Large Language Models

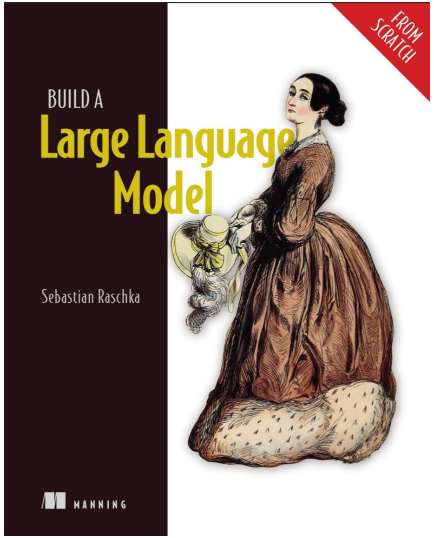
Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux

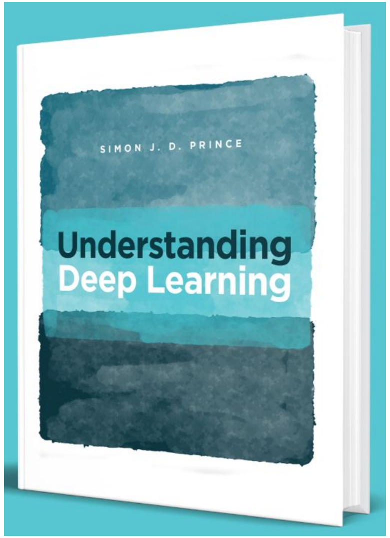
Goal of this course

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

Some references

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





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

WORDS != TOKENS

We will follow this tradition in this course, although sometimes it can be a bit misleading…

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 <u>integers</u> (which represent tokens).

GPT-2 has 50,257 tokens

What is a language model (LM)?

Input: a sentence (as a sequence of tokens)

Output: predict the next token

Basic examples:

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

N-GRAMS ARE LIMITED

Number of parameters: *vocab_size* ** *context_length*

vocab_size: total number of tokens

context_length: number of tokens considered for prediction

Think of it as a very large matrix…

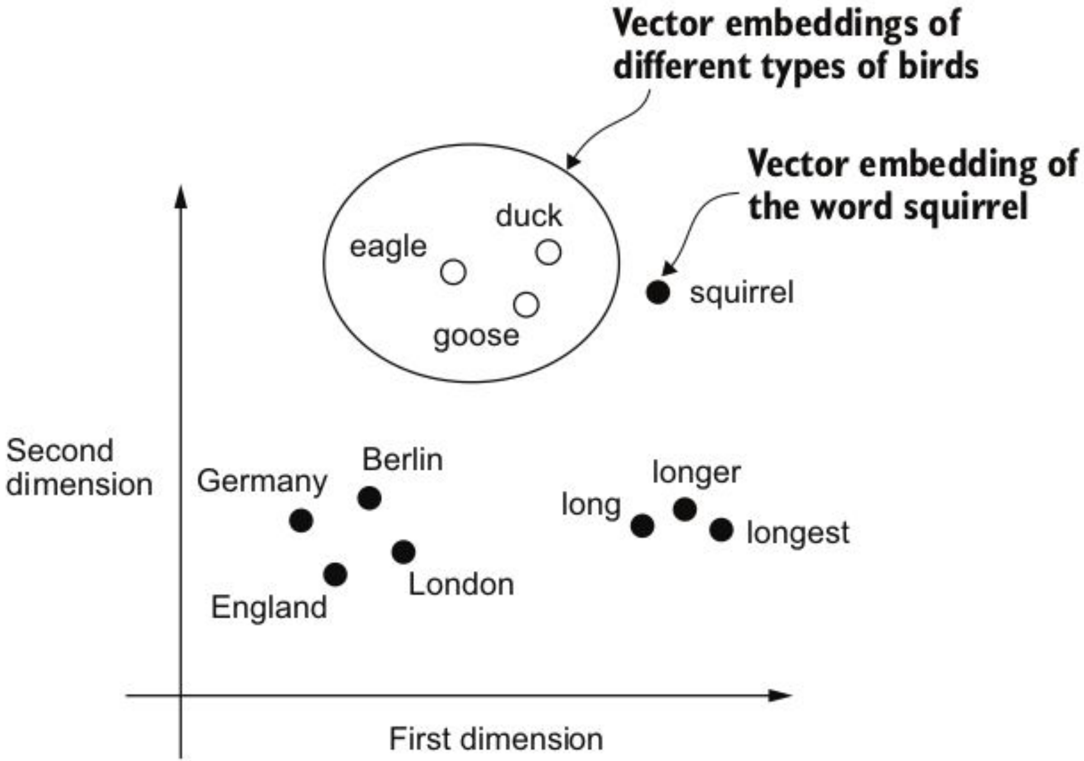
The 2003 (silent) breakthrough

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

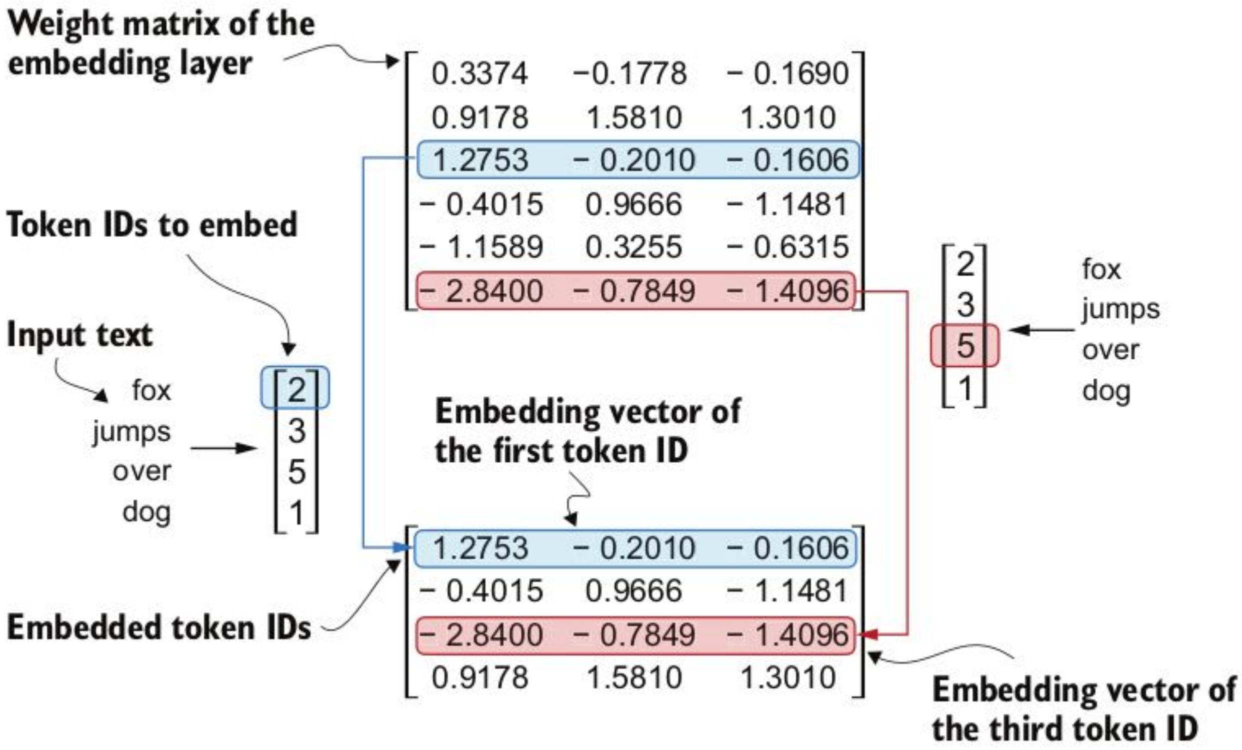
```
import torch
import torch.nn as nn
```

```
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>)
```

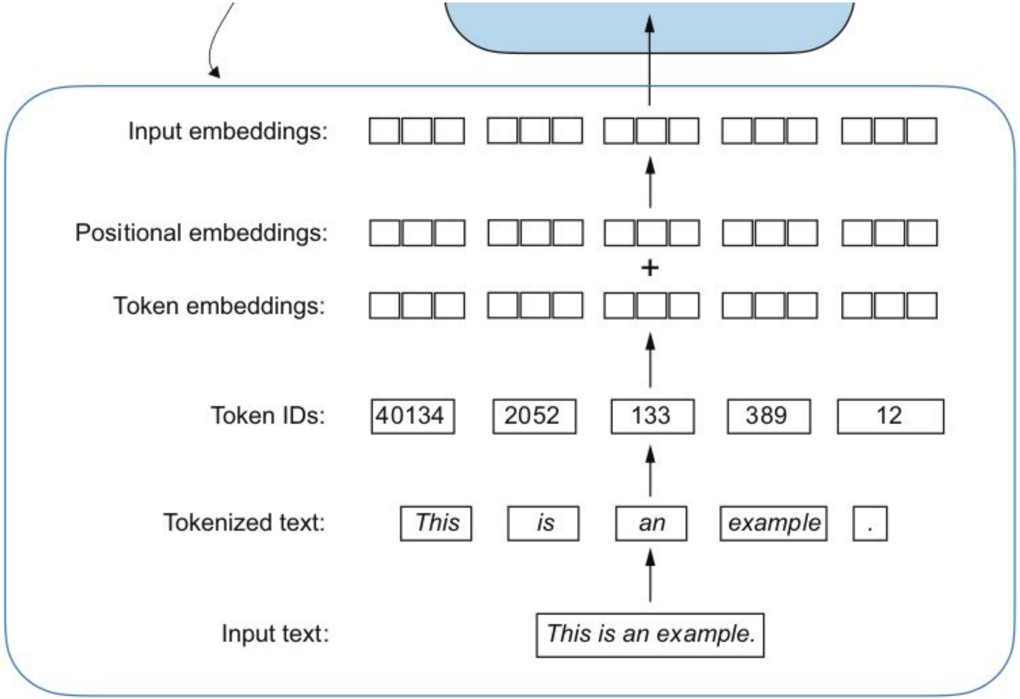
Advanced question: what is the difference between nn.embedding and nn.linear?

From text to vectors



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

(We will discuss later *positional embeddings*.)



Statistics

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

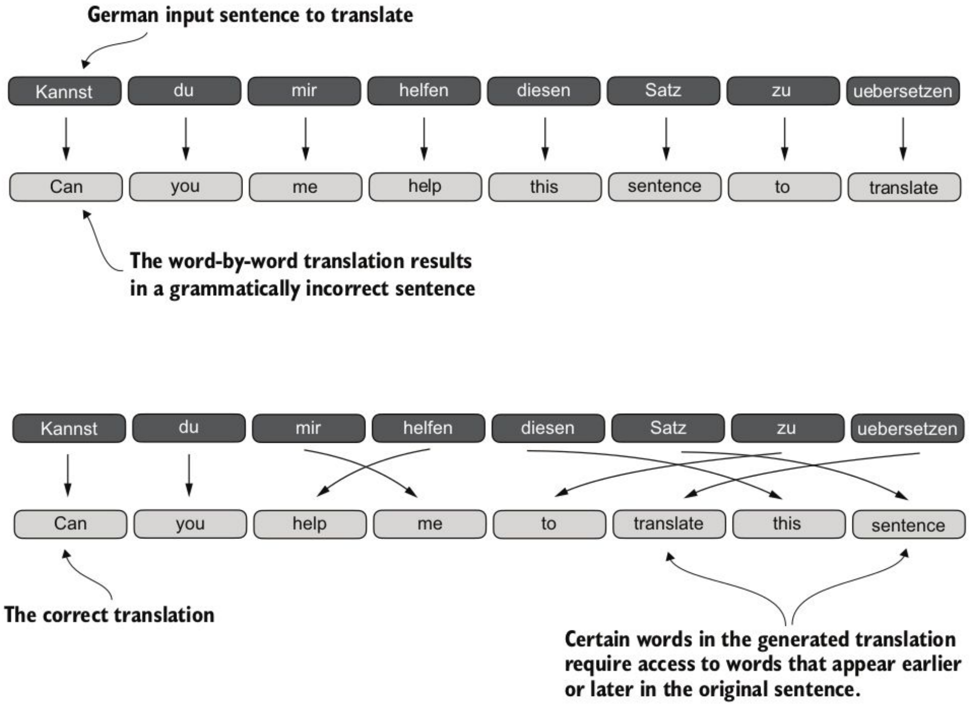
The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

Multi-layer Perceptron (MLP)

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

Two issues with MLPs

- We cannot have long contexts
- Struggle with long dependencies



Shall we look at some actual code?

The attention mechanism

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

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

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Attention is all you need

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

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

A SIDE-NOTE

OpenAI scientist Noam Brown:

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

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

A self-attention head

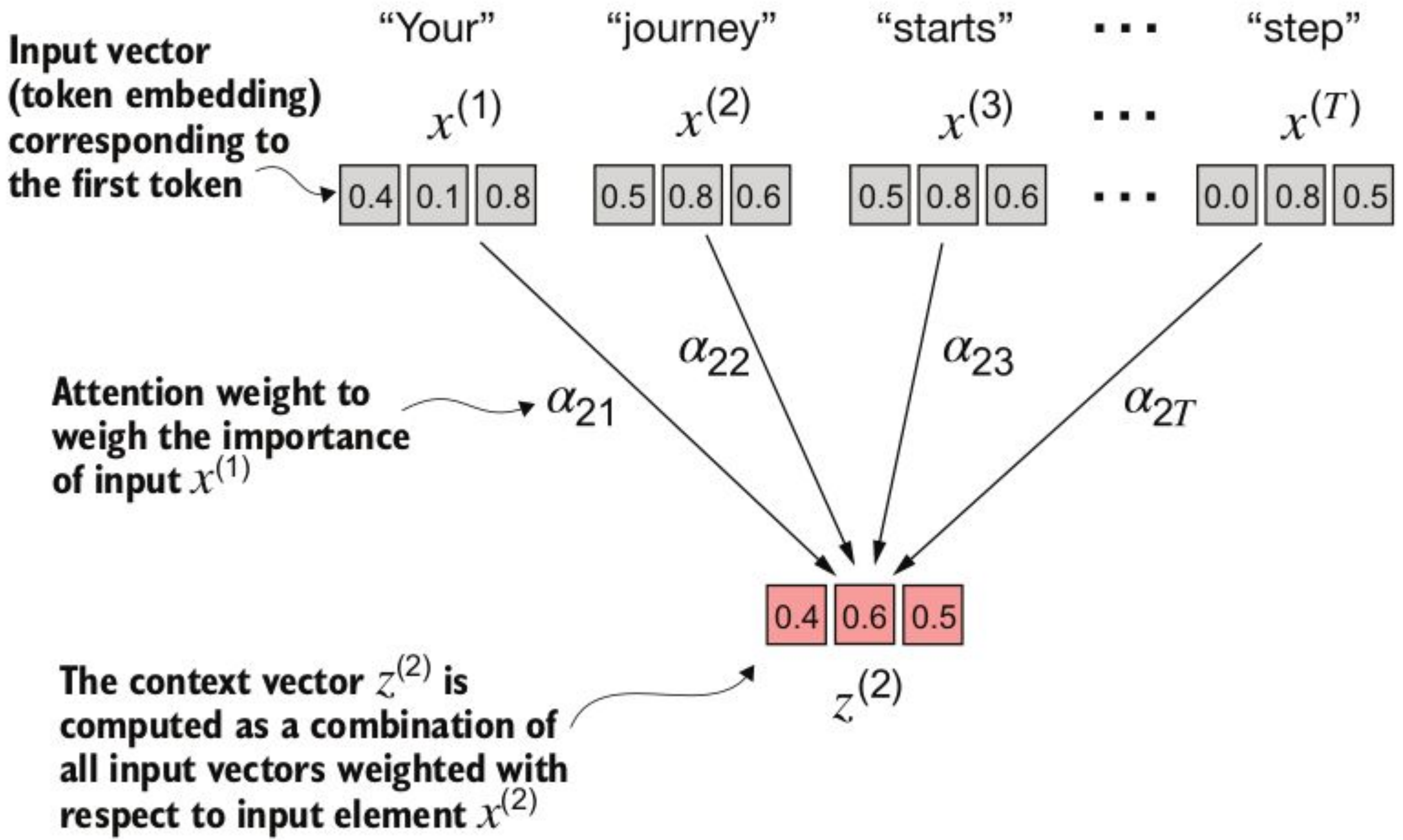
Input: an embedding vector x(i) for each token i

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

Intuition: z(i) gathers *contextual* information

Computing context vectors

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



Input vector (token embedding) corresponding to the first token
"Your"
"journey"
"starts"
"step"
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$
$x^{(T)}$
0.4 0.1 0.8
0.5 0.8 0.6
0.5 0.8 0.6
0.0 0.8 0.5
Attention weight to weigh the importance of input $x^{(1)}$
α_{21}
α_{22}
α_{23}
α_{2T}
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.

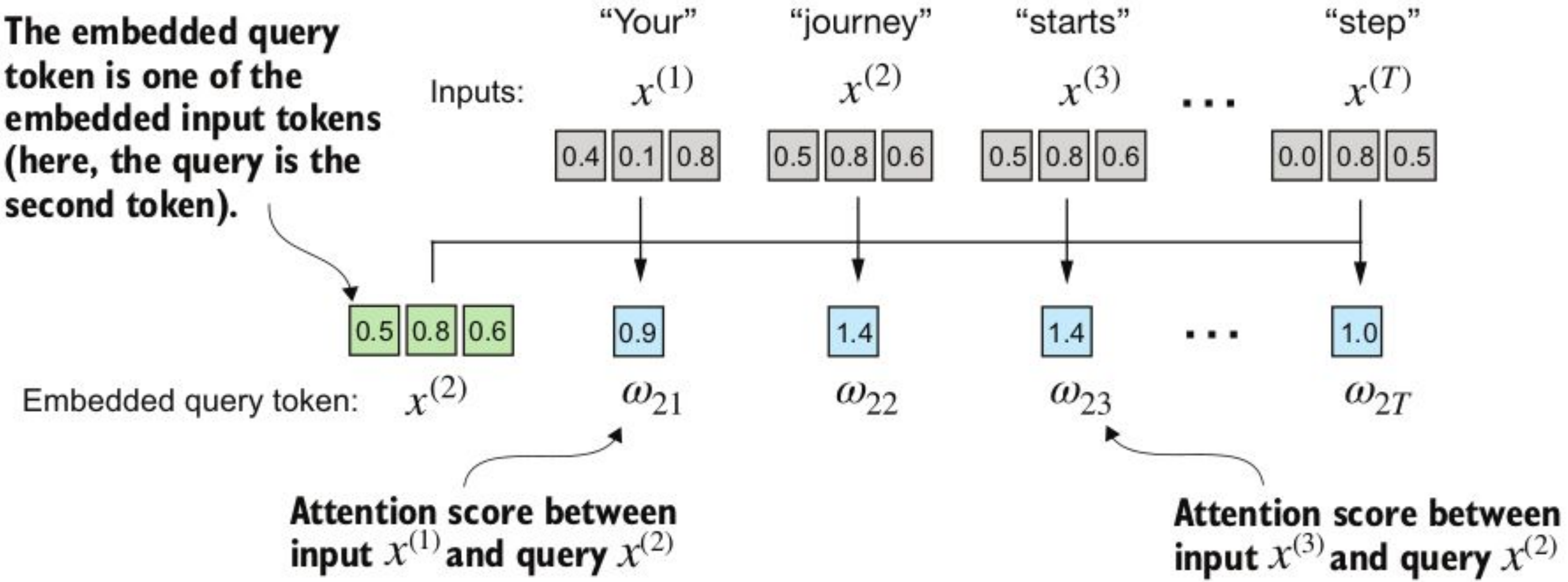
We first compute **attention scores**, and then normalise them into **attention weights**.

SIMPLIFICATION

As a starter, we begin with **non-trainable** attention weights.

This is ***only*** for the sake of explanation: the whole point of Transformers is to have trainable attention weights!

Computing <u>non-trainable</u> attention scores: dot-product



Again just a matrix multiplication…

```
context_length = 3
embed_dim = 2

x = torch.randn(context_length, embed_dim)
attention_scores = torch.empty(context_length, context_length)

for i, x_i in enumerate(x):
    for j, x_j in enumerate(x):
        attention_scores[i, j] = torch.dot(x_i, x_j)
attention_scores
```

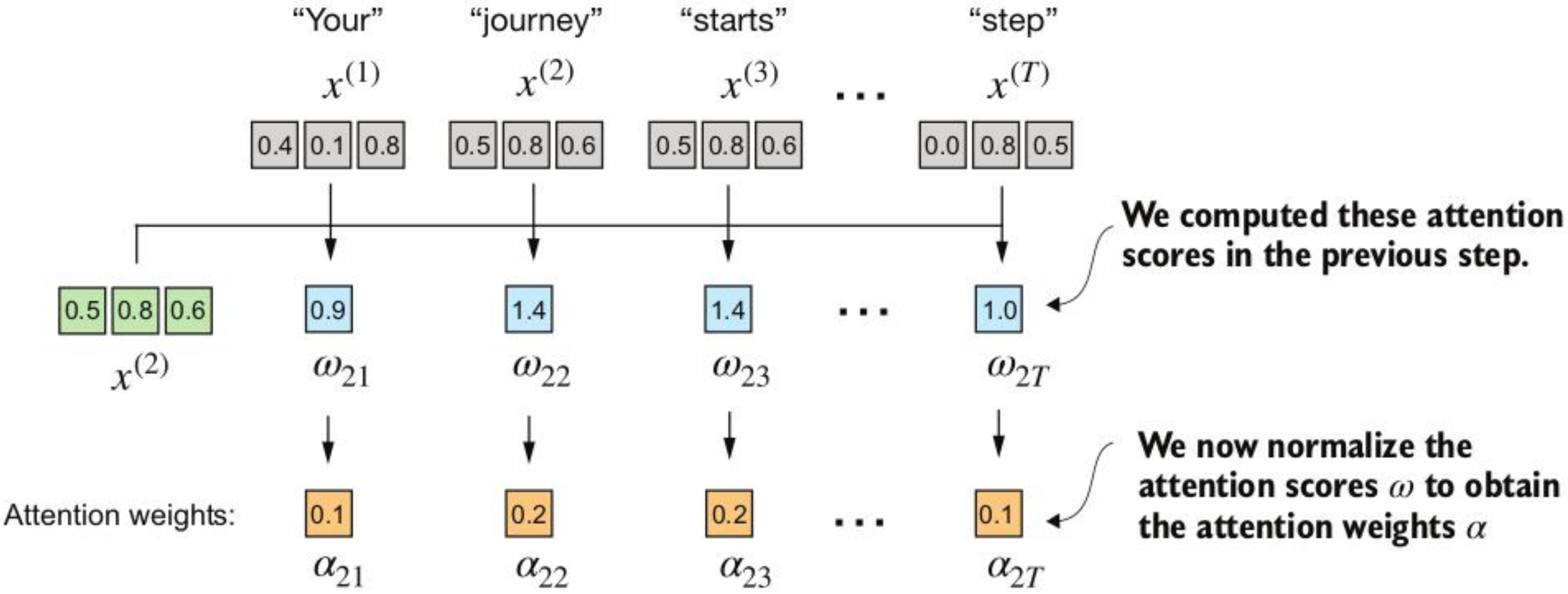
```
tensor([[ 1.3259, -0.3350, -0.4560],
        [-0.3350,  0.1948,  0.2814],
        [-0.4560,  0.2814,  0.4074]])
```

```
attention_scores = x @ x.T
attention_scores
```



```
tensor([[ 1.3259, -0.3350, -0.4560],
        [-0.3350,  0.1948,  0.2814],
        [-0.4560,  0.2814,  0.4074]])
```

From non-trainable attention scores to weights: softmax



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.

Illustration of softmax sensitivity to large numbers:

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

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

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

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

Scaled self-attention

Assume u,v are vectors of dimension d:

u,v ~ N(0,1)

What is the distribution of u \cdot v?

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

But: Var(u \cdot v / sqrt(d)) = 1

Self-attention head with (non-trainable!) attention weights

```
x = torch.randn(context_length, input_dim)
attention_scores = x @ x.T
attention_weights = torch.softmax(attention_scores * input_dim**-0.5, dim=-1)
context_vectors = attention_weights @ x
```

UN-SIMPLIFICATION

So far our attention weights were **<u>non-trainable</u>**.

We want attention weights to be data-dependent: depending on the embedding vector, the attention is put on different parts of the context.

Keys, queries, and values

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

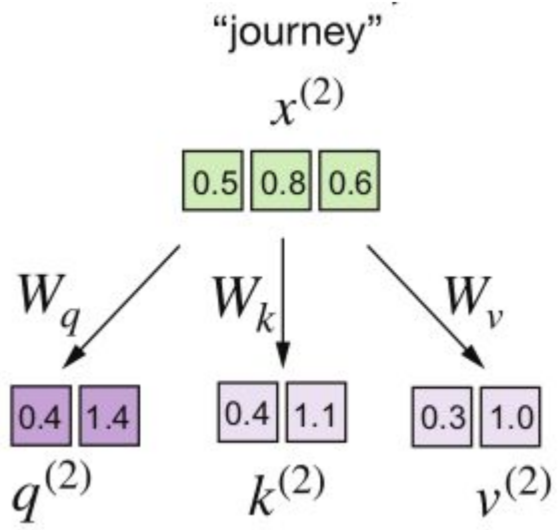
Output: for each token i:

- 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.

Computed by matrix multiplications...

We introduce three matrices with trainable parameters:

- Wq for query,
- Wk for key,
- Wv for value.



Self-attention head with (trainable!) attention weights

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

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

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

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

As a NN.Module

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

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

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

Complexity

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

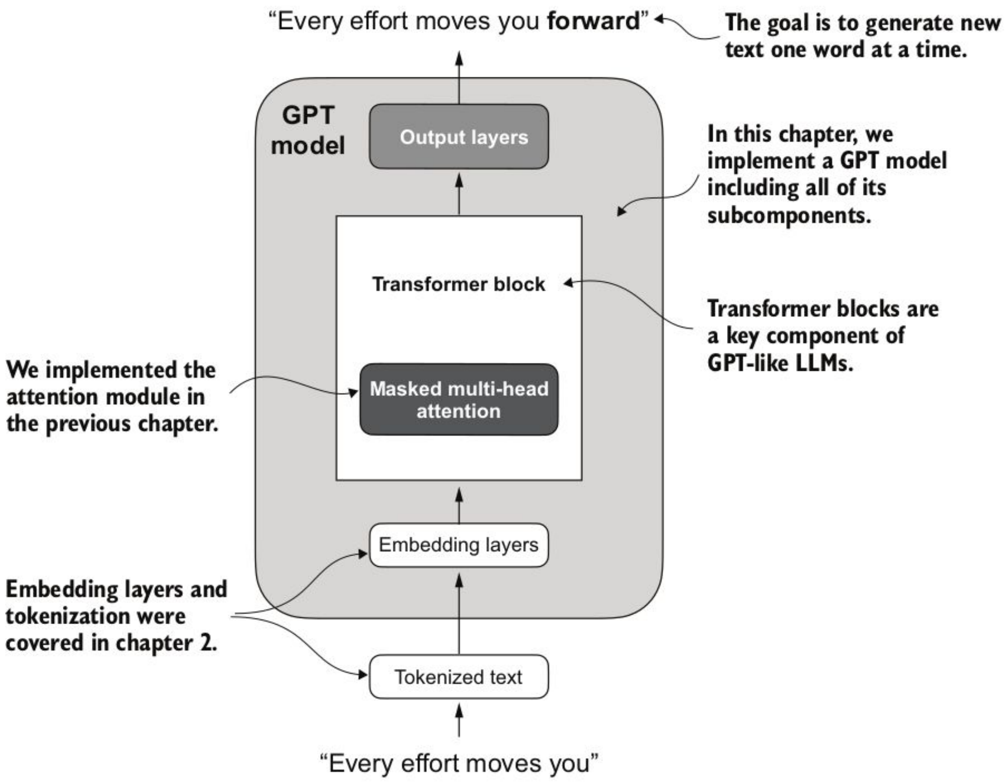
The memory footprint is quadratic in context length!

Shall we look at some actual code?

The transformer architecture

- Sliding windows
- Batching
- Cross entropy loss
- Residual connections
- Normalization layers
- Positional embeddings
- …

Attention heads as key components in a Transformer

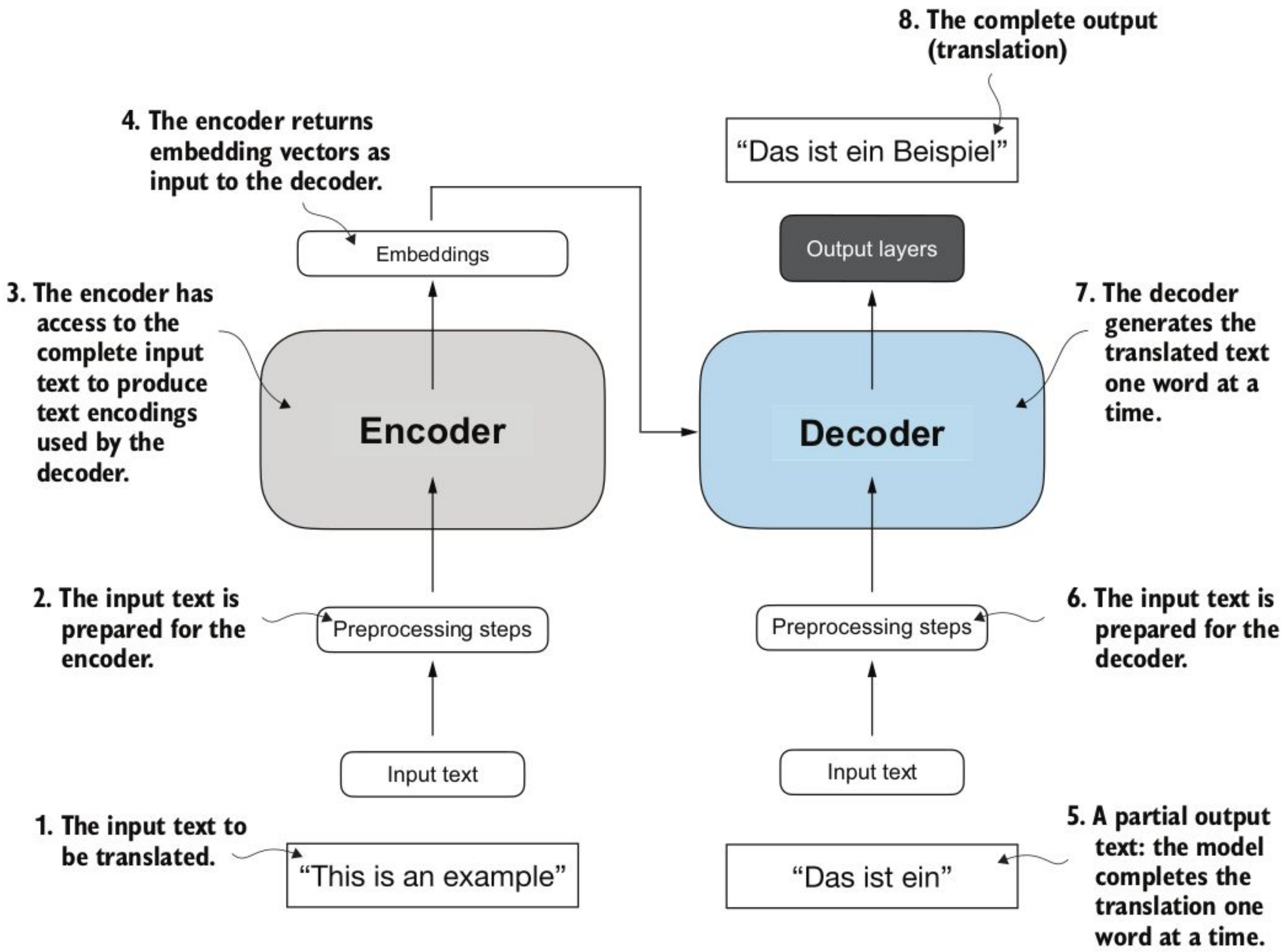


IMPORTANT

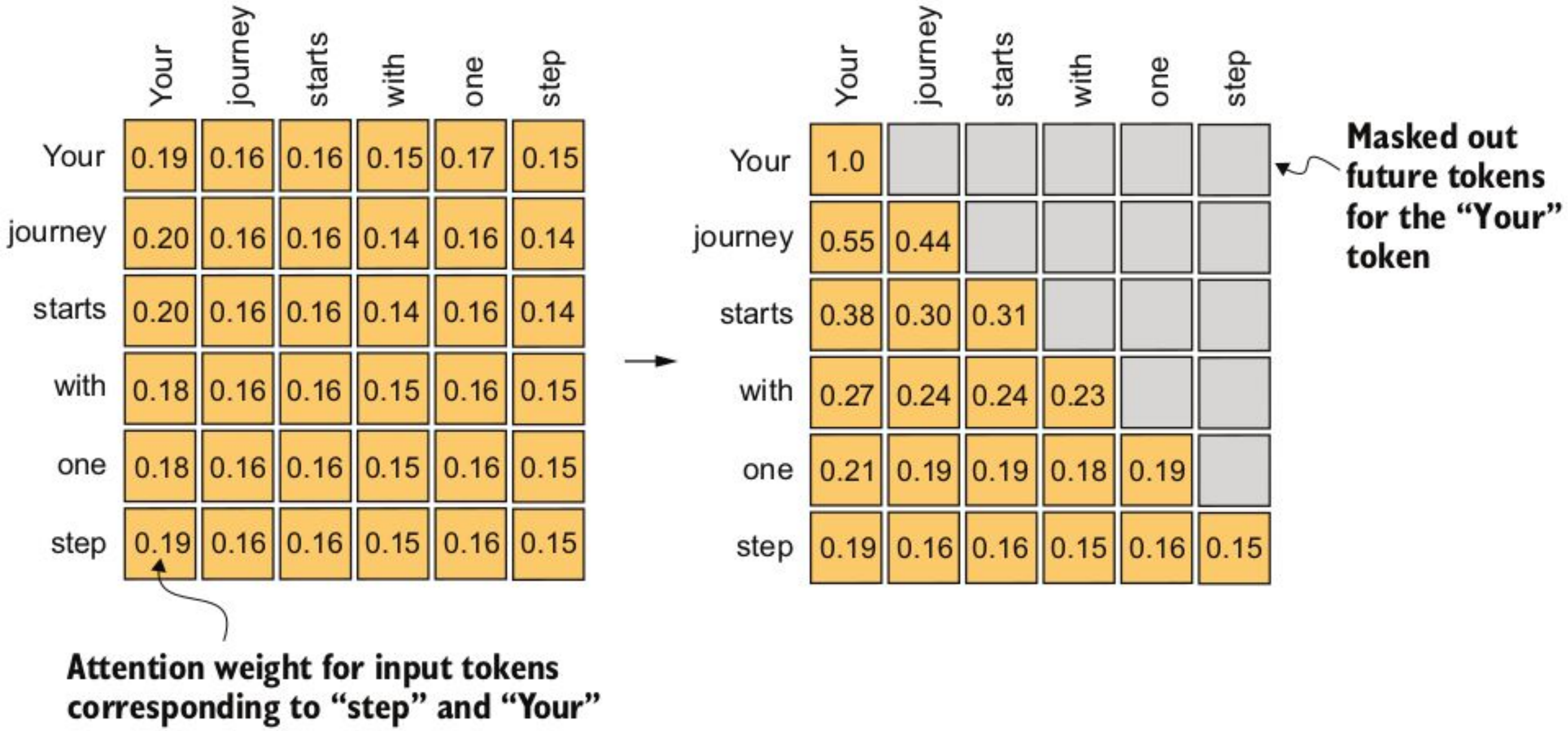
The Transformer consists of a number of “layers”, each with the same signature:

Input: a sequence of vectors, one for each token

Output: a sequence of vectors, one for each token

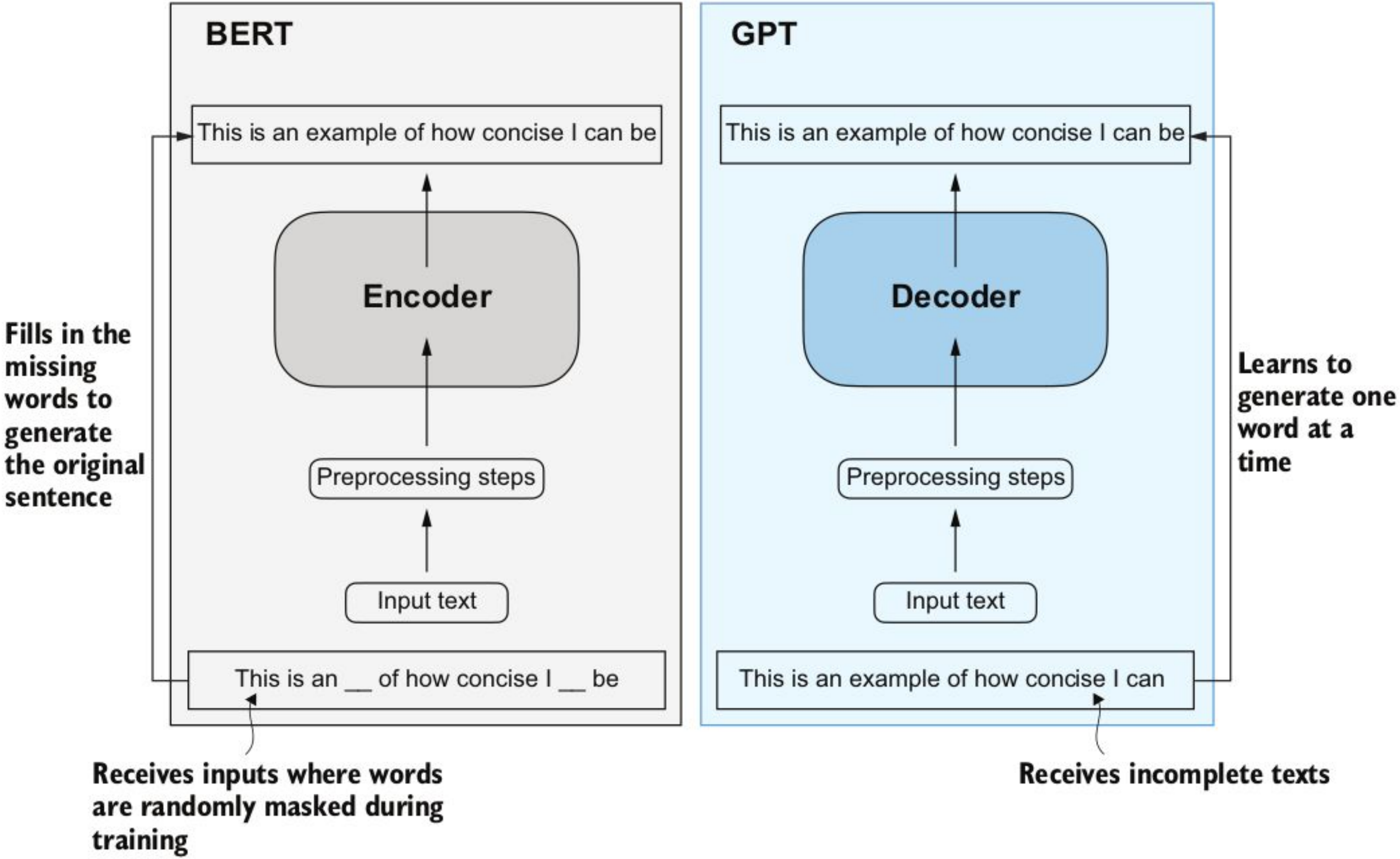

8. The complete output (translation)
"Das ist ein Beispiel"
4. The encoder returns embedding vectors as input to the decoder.
Embeddings
Output layers
3. The encoder has access to the complete input text to produce text encodings used by the decoder.
Encoder
Decoder
7. The decoder generates the translated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



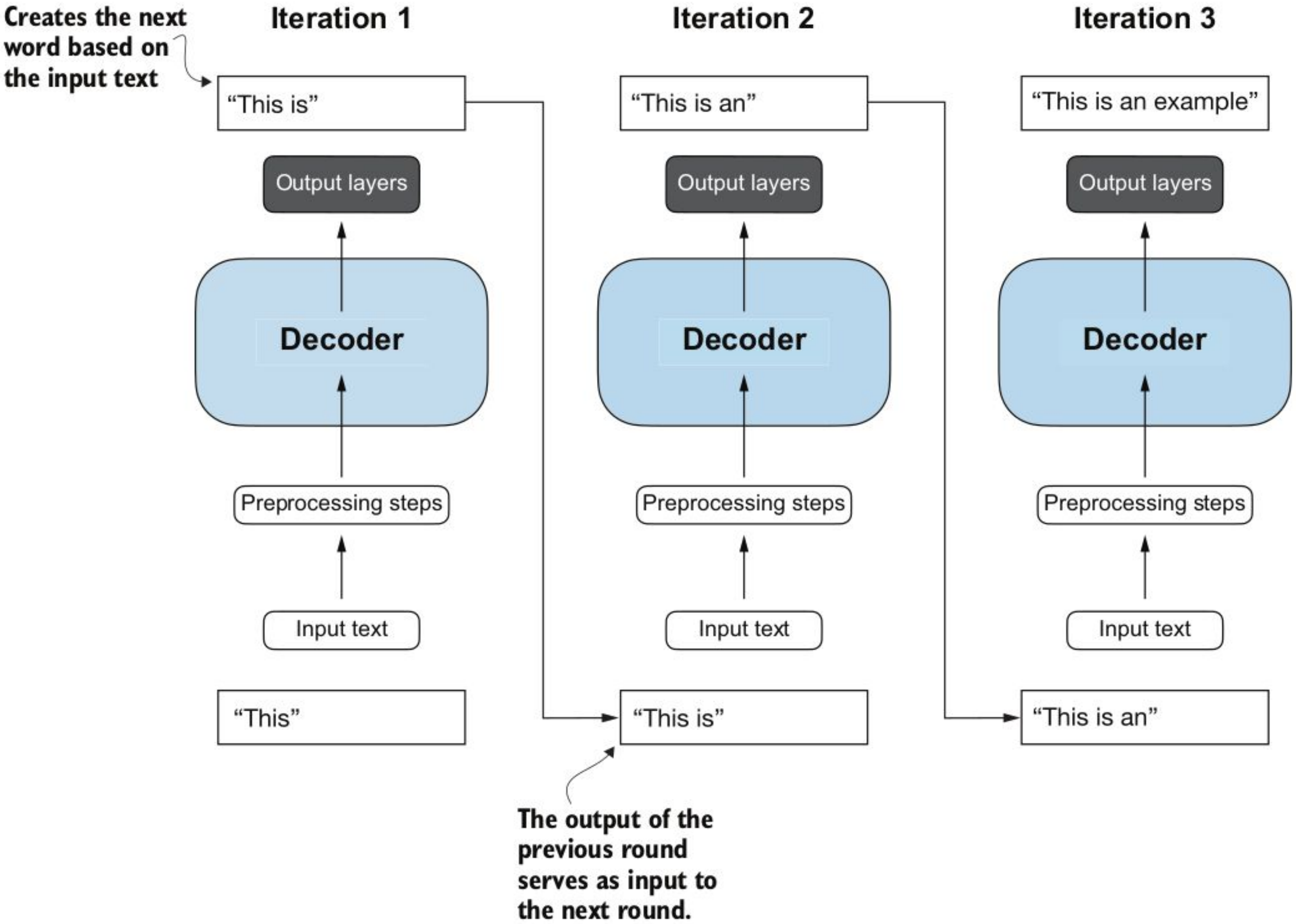
Implementation of the mask

```
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
```

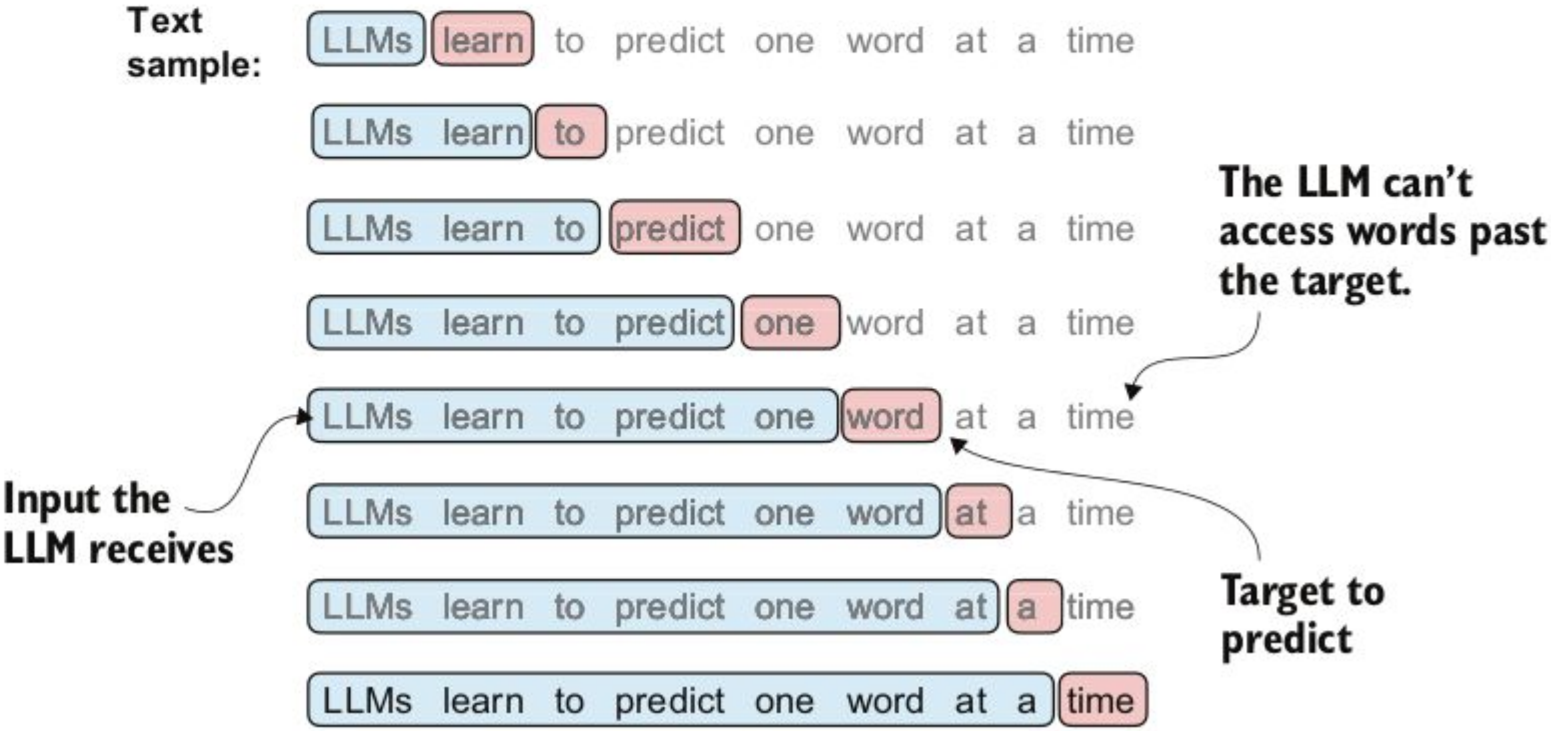

BERT
This is an example of how concise I can be
Encoder
Preprocessing steps
Input text
This is an __ of how concise I __ be
Fills in the missing words to generate the original sentence
Receives inputs where words are randomly masked during training
GPT
This is an example of how concise I can be
Decoder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

What does "auto-regressive" mean?

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


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

Sliding windows



What are the benefits of the sliding / expanding windows?

Fix c = context_length

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

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

Data collector

```
data = torch.tensor(tokenizer.encode(text), dtype=torch.long)
n = int(0.9*len(data))
train_data = data[:n]
val_data = data[n:]
```

```
def get_batch(split):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size - 1, (batch_size,))
    X = torch.stack([data[i:i+block_size] for i in ix])
    Y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    return X, Y
```

Models' signatures

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

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

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

For each window, make the prediction and compute the loss

Models' signatures with batching

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

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

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

A self-attention head with batching

```
class Head(nn.Module):
    def __init__(self, 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)
        # Some Pytorch way of defining a matrix without trainable parameters
        self.register_buffer('tril', torch.tril(torch.ones(context_length, context_length)))

    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)
        mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
        masked_attention_scores = attention_scores.masked_fill(mask.bool(), float('-inf')) # (B, T, T)
        attention_weights = torch.softmax(masked_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
```

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

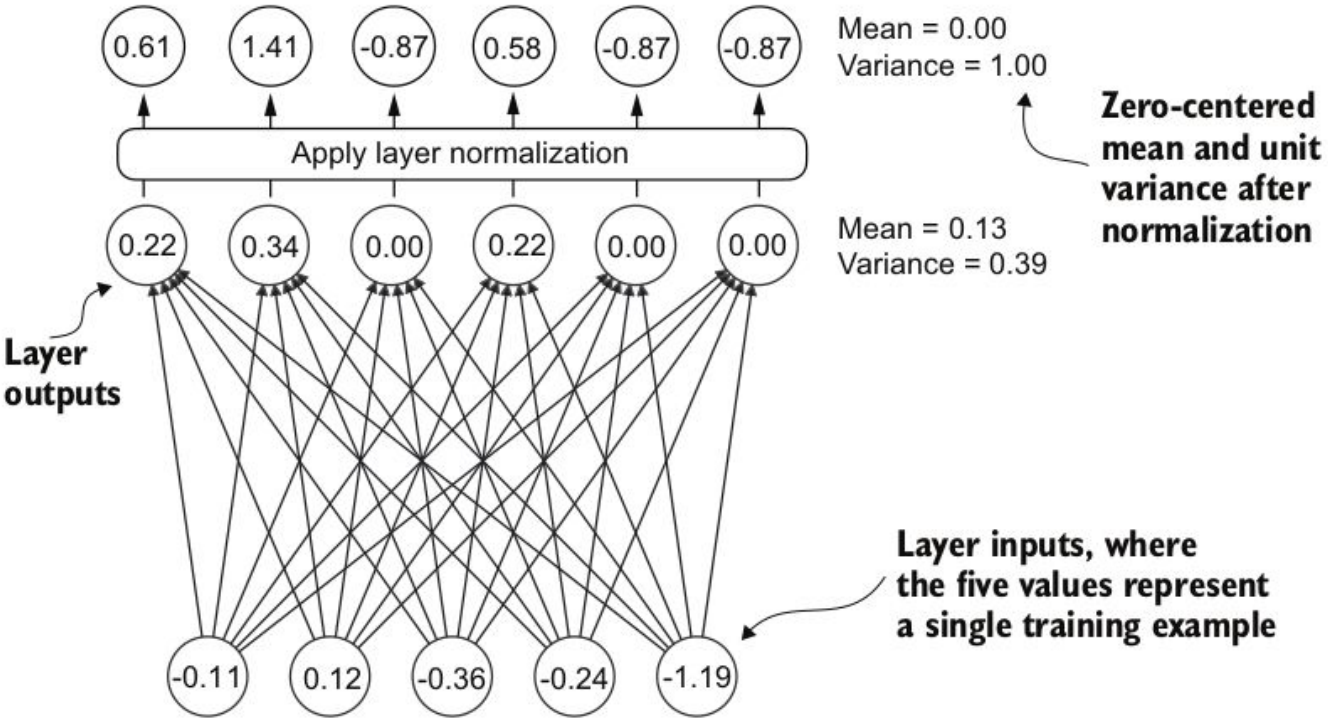
```python
vocab_size = 5

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?

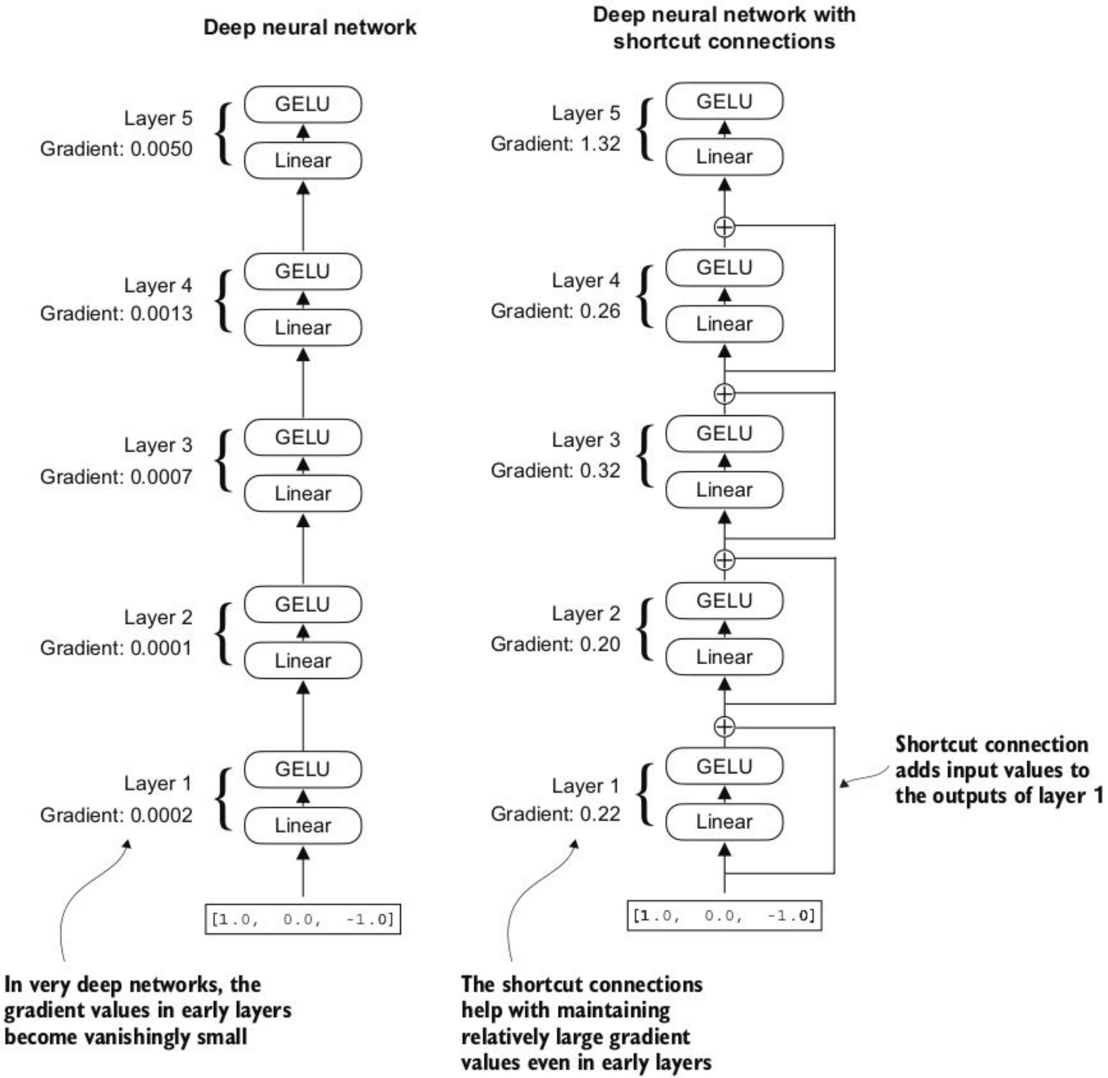
- **Maximum likelihood estimation:** Minimizing cross-entropy is equivalent to maximizing the likelihood of the observed data.
- **Encourages accurate probabilities:** It encourages the model to produce probabilities that closely match the true distribution, not just predict the correct class.
- **Smooth 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.

Layer normalization



What do "shortcut connections" (also called "residual connections") do?

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



Deep neural network
Deep neural network with shortcut connections
Layer 5
Gradient: 0.0050
Layer 4
Gradient: 0.0013
Layer 3
Gradient: 0.0007
Layer 2
Gradient: 0.0001
Layer 1
Gradient: 0.0002
GELU
Linear
[1.0, 0.0, -1.0]
Layer 5
Gradient: 1.32
Layer 4
Gradient: 0.26
Layer 3
Gradient: 0.32
Layer 2
Gradient: 0.20
Layer 1
Gradient: 0.22
Shortcut connection adds input values to the outputs of layer 1
In very deep networks, the gradient values in early layers become vanishingly small
The shortcut connections help with maintaining relatively large gradient values even in early layers

What is 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.

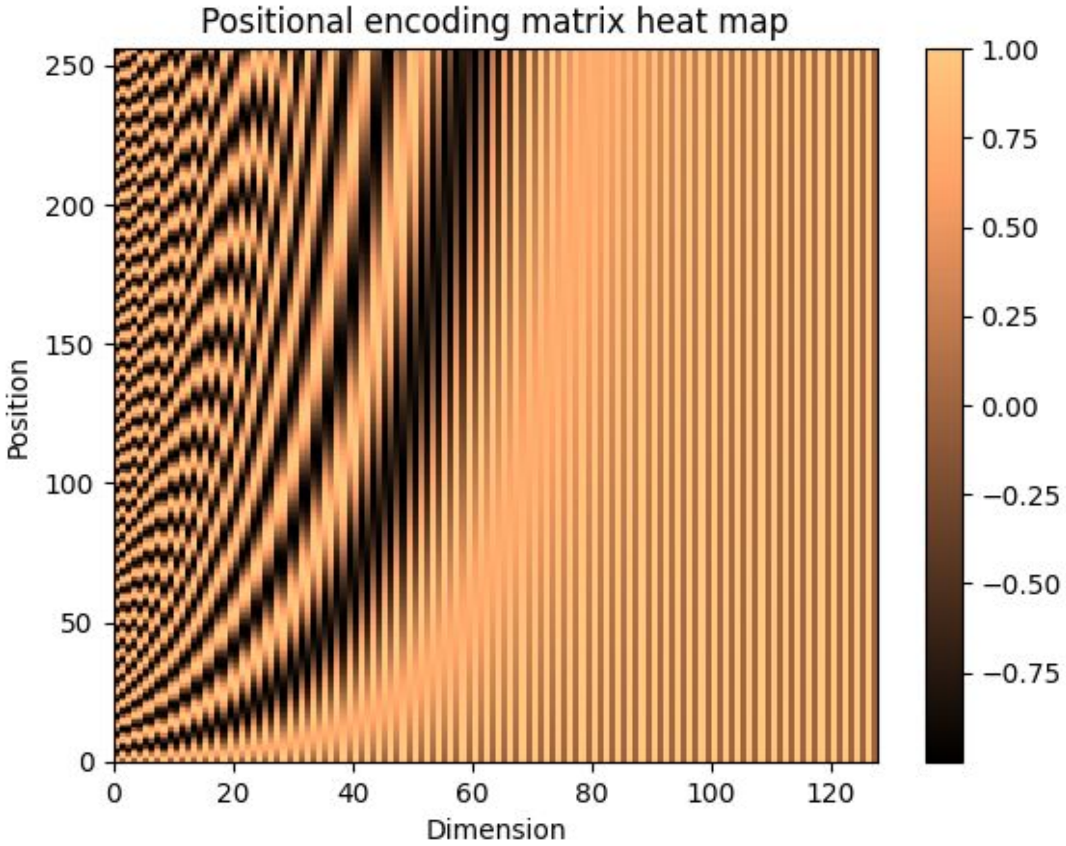
- **Prevents Overfitting:** By randomly dropping out neurons, dropout prevents the network from learning complex co-adaptations that are specific to the training data. This helps the model generalize better to unseen data.
- **Ensemble Effect:** Dropout can be seen as training an ensemble of multiple 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.

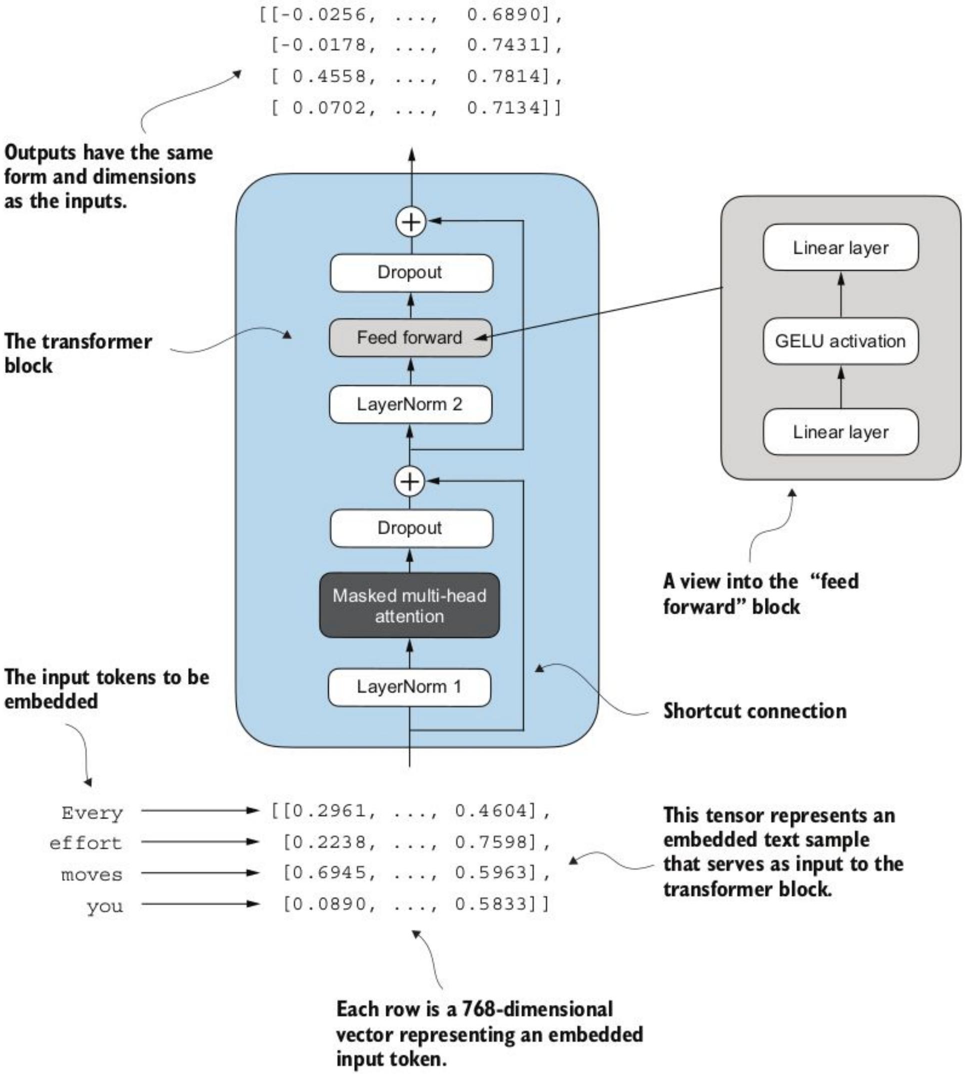
Positional embeddings

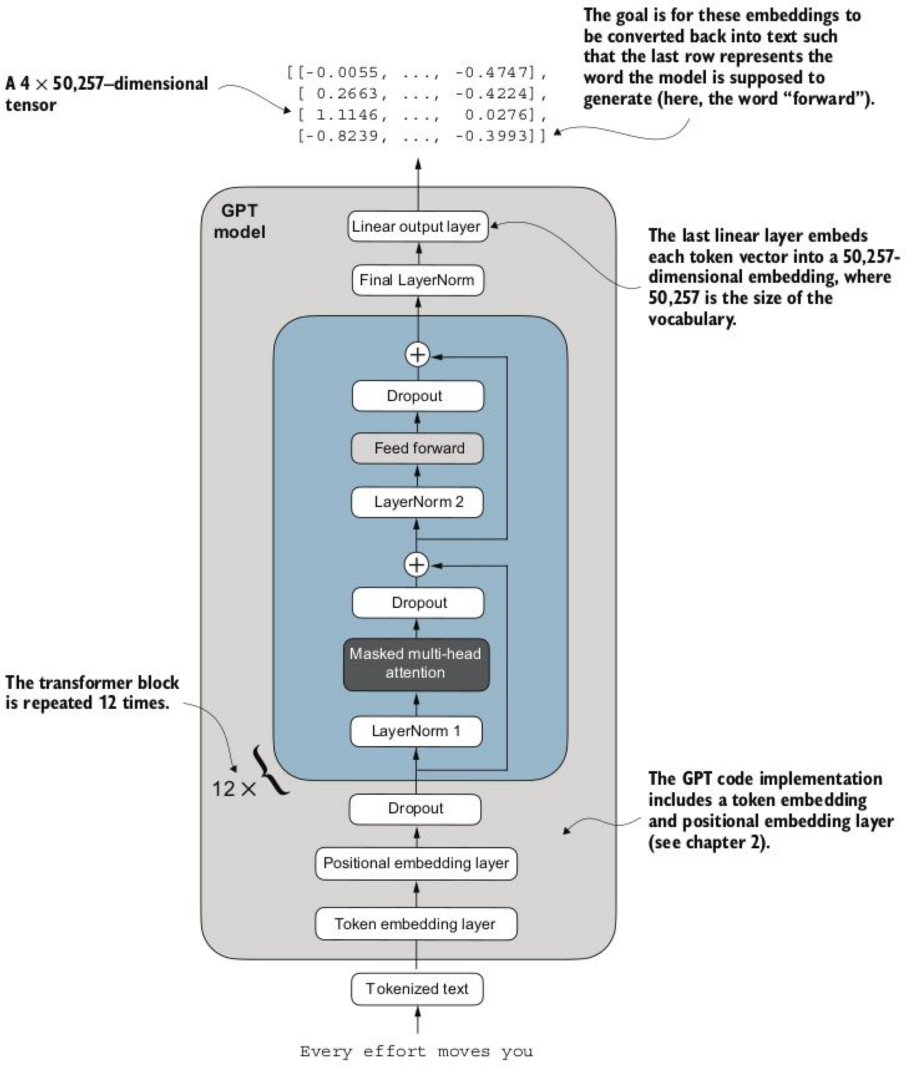
Attention scores are computed in the same way for all other tokens. But sometimes it is useful to be aware of *relative* positions (just before, just after,...), or absolute positions (at the beginning, at the end).

```
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)
```

Some visualization




[[-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.
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
Linear layer
GELU activation
Linear layer
The transformer block
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.



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").
A 4 × 50,257–dimensional tensor
[[-0.0055, ..., -0.4747],
[0.2663, ..., -0.4224],
[1.1146, ..., 0.0276],
[-0.8239, ..., -0.3993]]
GPT model
Linear output layer
The last linear layer embeds each token vector into a 50,257-dimensional embedding, where 50,257 is the size of the vocabulary.
Final LayerNorm
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
The transformer block is repeated 12 times.
12 ×
Dropout
The GPT code implementation includes a token embedding and positional embedding layer (see chapter 2).
Positional embedding layer
Token embedding layer
Tokenized text
Every effort moves you

Shall we look at some actual code?

Tokenization

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

Credits

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

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

Basics

A **bit** = 0 or 1

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

Character encodings:

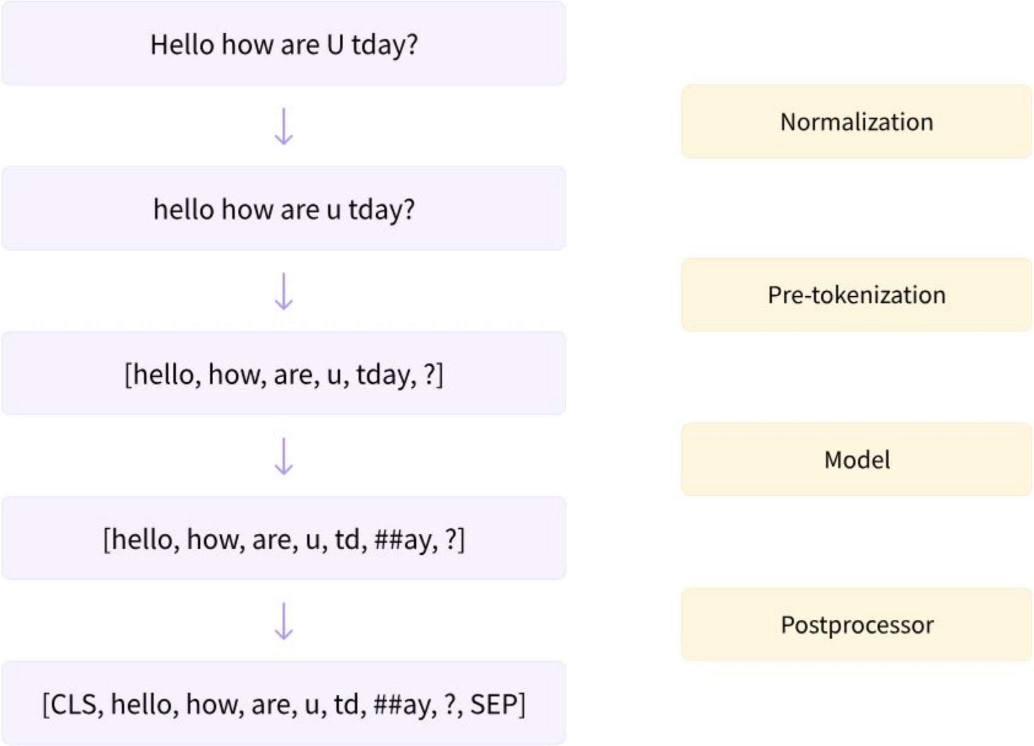
- ASCII (code unit: 7 bits)
- Unicode: UTF-8, UTF-16, UTF-32 (code unit: 8,16,32 bits)

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

Attention

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

The full tokenization pipeline



Normalization

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

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
print(type(tokenizer.backend_tokenizer))
```

```
<class 'tokenizers.Tokenizer'>
```

```
print(tokenizer.backend_tokenizer.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

```
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
```

```python
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

```
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
```

```python
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
```

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

```python
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
```

```python
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

Short practical session:
Train a tokenizer on code

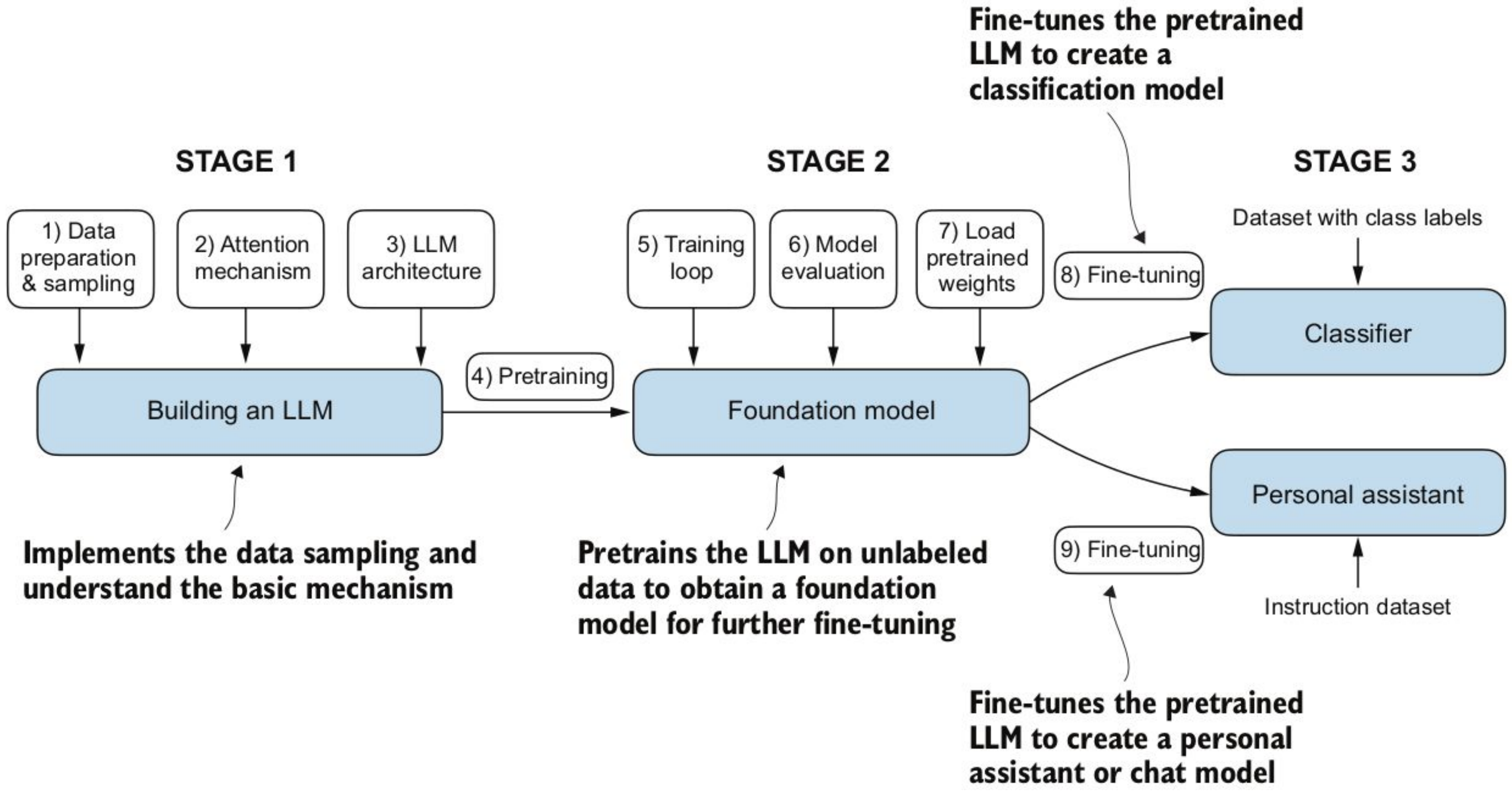
Fine-tuning

- 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
4) Pretraining
Building an LLM
5) Training loop
6) Model evaluation
7) Load pretrained weights
Foundation model
8) Fine-tuning
Dataset with class labels
Classifier
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.

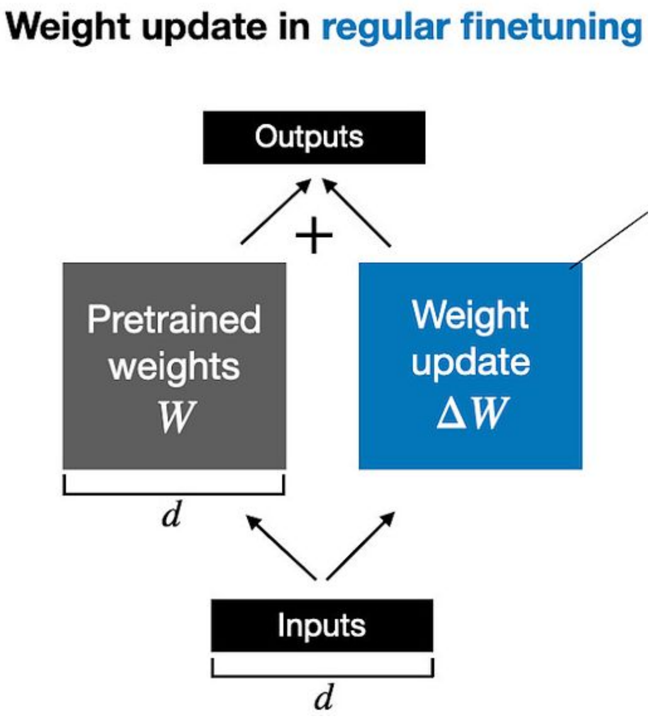
Low-rank Adaptation (LoRA)

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

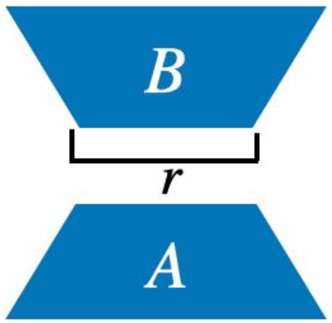


Rank approximations

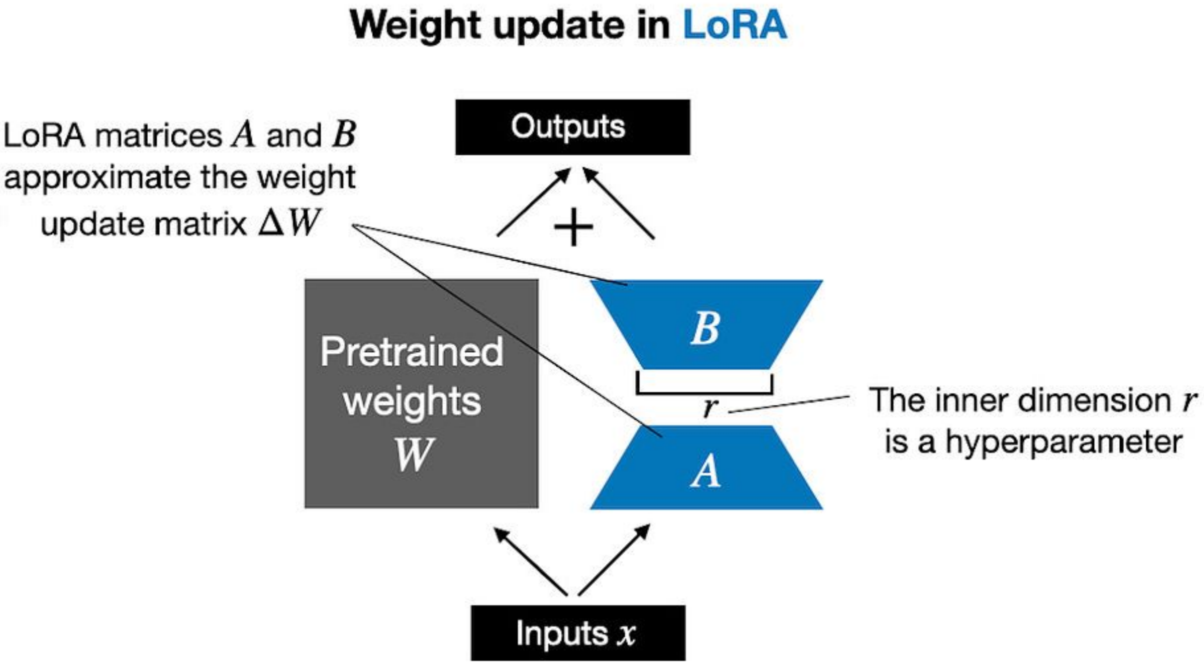
A matrix W of dimension dxd contains dxd parameters. It can be ***rank-r approximated*** by two matrices AxB with:

- A of dimension dxr
- B of dimension rxd

Instead of dxd parameters we now have 2xdxr parameters.



Weight update



LoRA Layer

```
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
```

```python
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()
        )
```