Large Language Models
Session 1

Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux
Co-teacher: Marc Lelarge

Goal of this course

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

LOGISTICS

- Website: https://llm.labri.fr/
- Please sign up on the Discord server!
- Each session will be two hours lecture + two hours practical
- Validation: project (presentation on 25/03)

Prerequisite

- Programmation: Python and Pytorch
- Linear algebra and Deep Learning

Python test:

```
"".join(map(lambda x:x[0], "Marc Le Large".split()))[::-1]
```

You did not laugh?

Introduction to Python:
https://scipy-lectures.org/intro/language/python_language.html

Introduction to NumPy and Matplotlib:
https://sebastianraschka.com/blog/2020/numpy-intro.html

Tentative course outline

It will surely evolve…

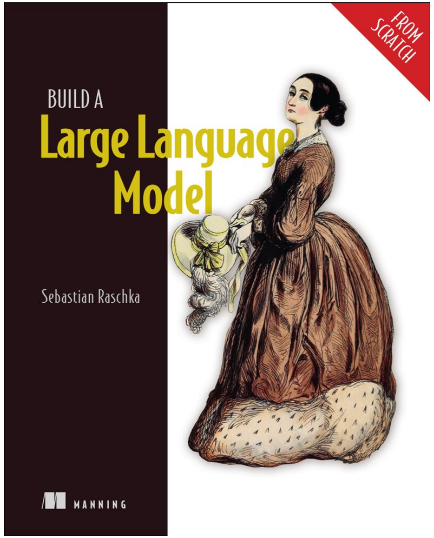
- 07.01 (Nath): Attention mechanism, Pre-training
- 14.01 (Nath): Tokenization, Fine-tuning
- 21.01 (Marc): Scaling laws, Probing
- 28.01 (Marc): Search strategies
- 04.02 (Nath): Retrieval-augmented generation
- 11.02 (Nath): Code generation
- 04.03 (Marc): Grammar decoders
- 11.03 (Marc): Value alignment (RLHF, RLCF)
- 25.03: Project presentations

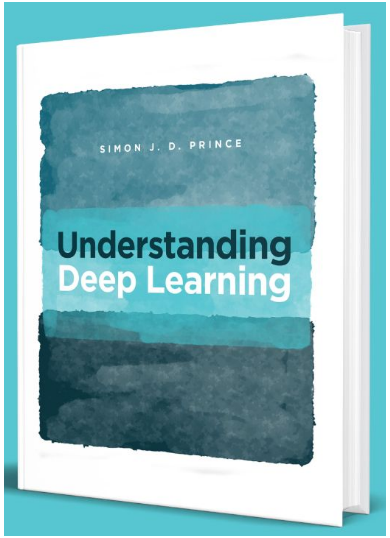
Outline for today

- Language models
- Attention mechanism
- Deep Learning magic

Some references for the first lectures

- 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 integers (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…

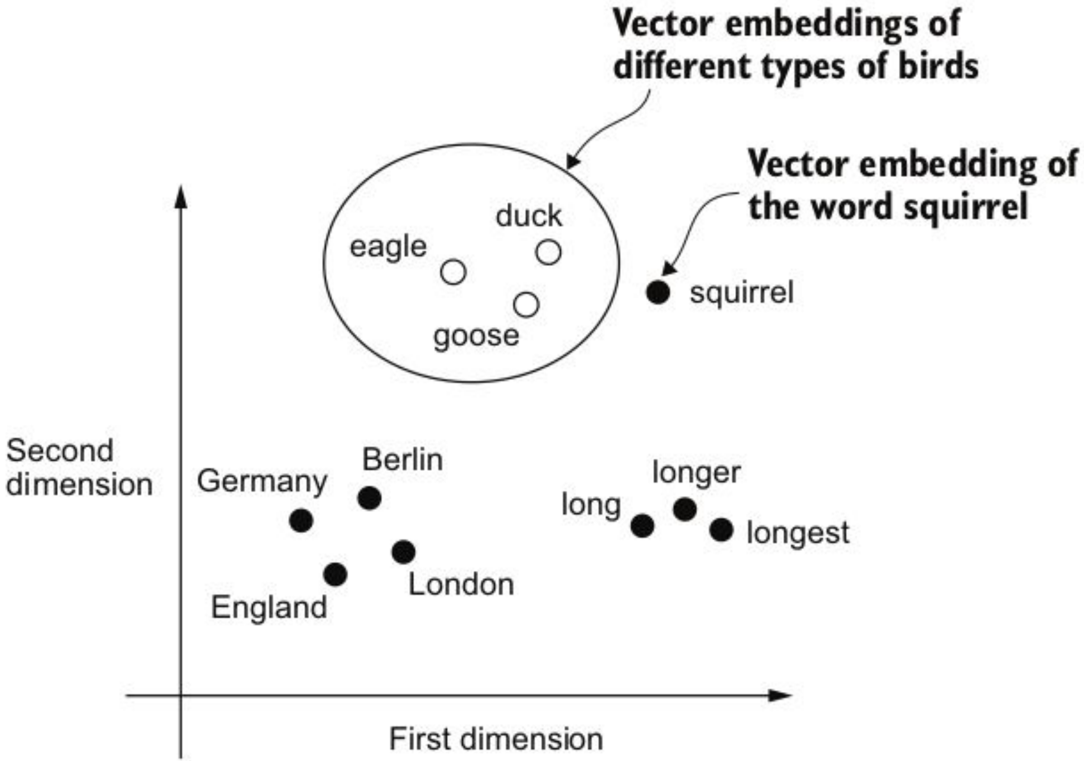
The 2003 (silent) breakthrough

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

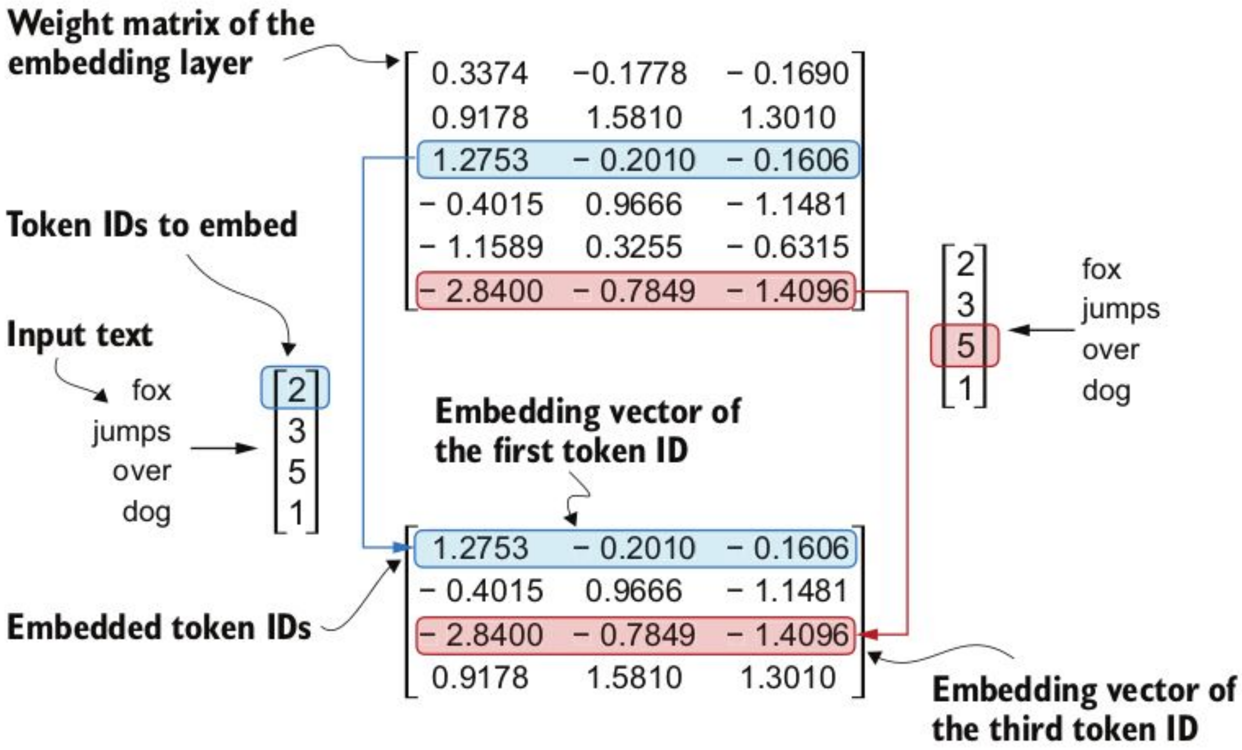
```python
import torch
import torch.nn as nn
```

```python
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>)
```

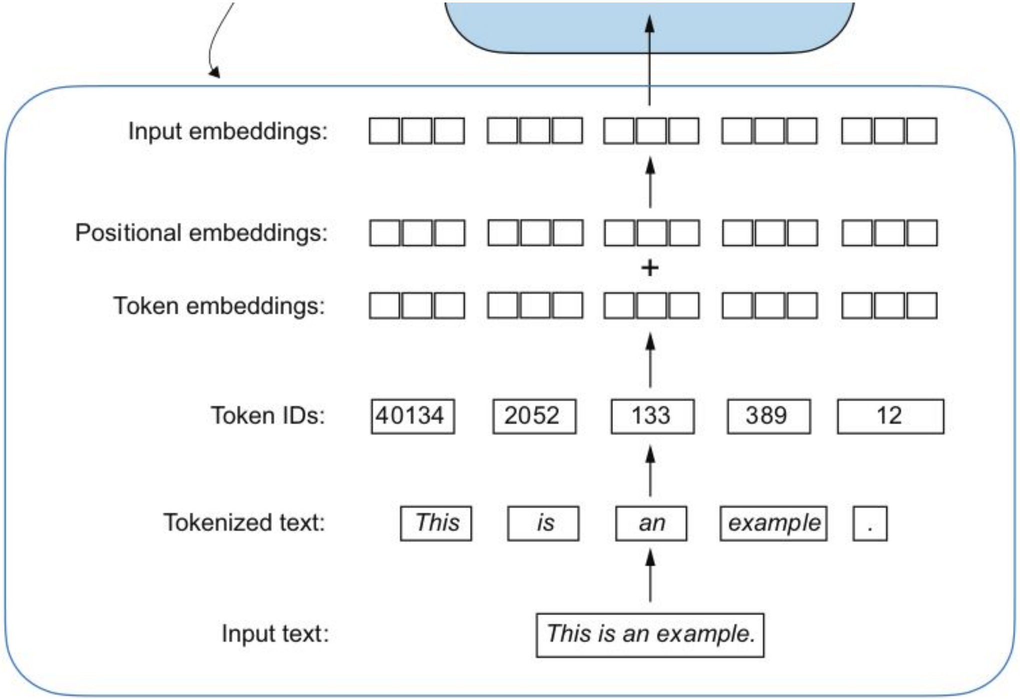
Advanced question: what is the difference between nn.embedding and nn.linear?

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.

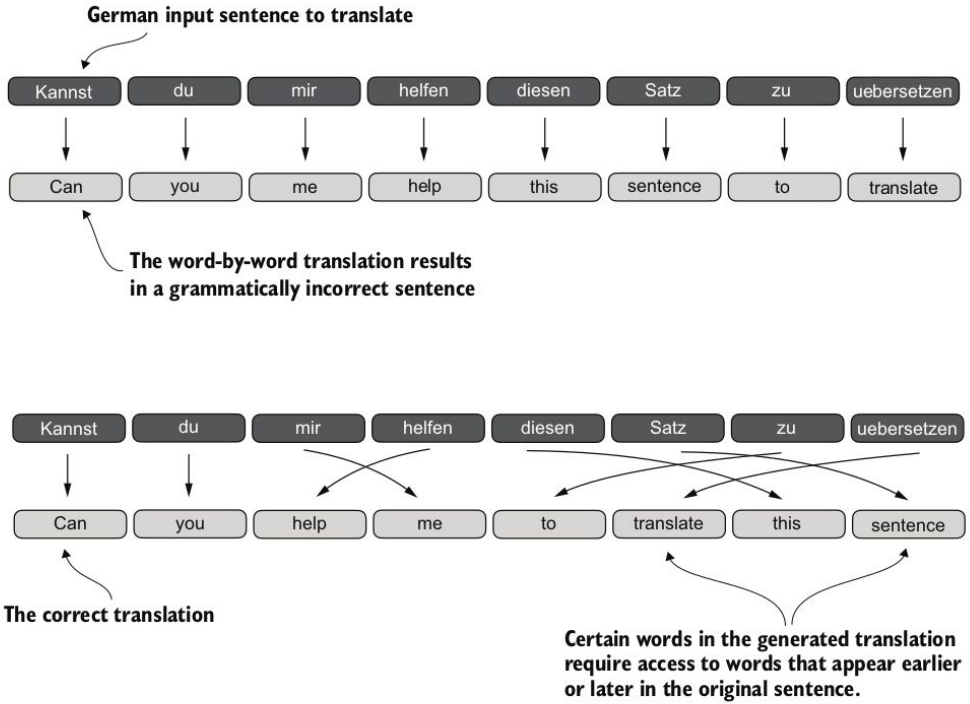
The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

Multi-layer Perceptron (MLP)

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.token_embedding_table = nn.Embedding(n_token, n_embed)
        self.net = nn.Sequential(
            nn.Linear(context_length * n_embed, n_embed2),
            nn.Tanh(),
            nn.Linear(n_embed2, n_token)
        )
```

Two issues with MLPs

- We cannot have long contexts
- Struggle with long dependencies



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

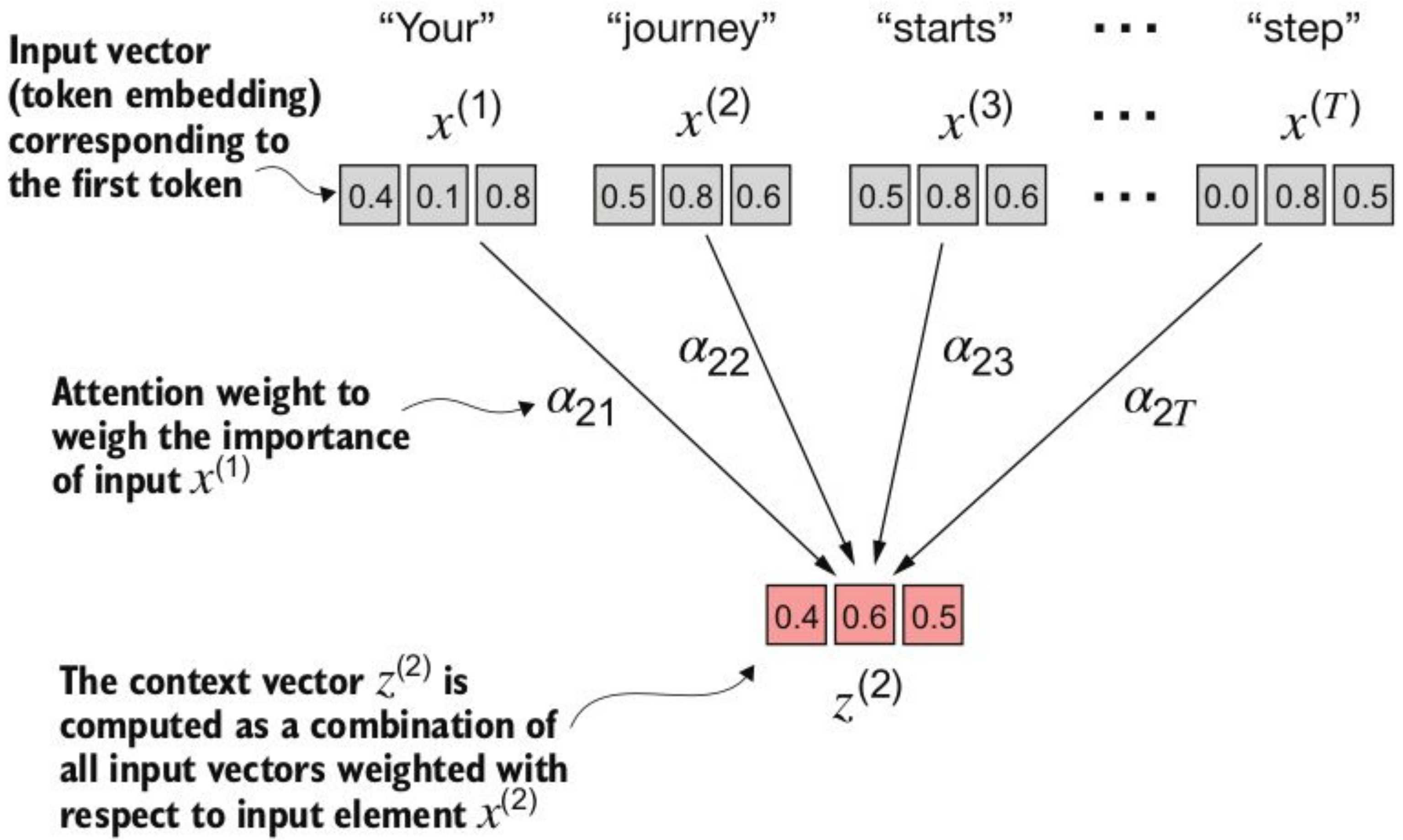
Input: an embedding vector x(i) for each token i

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.



"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.

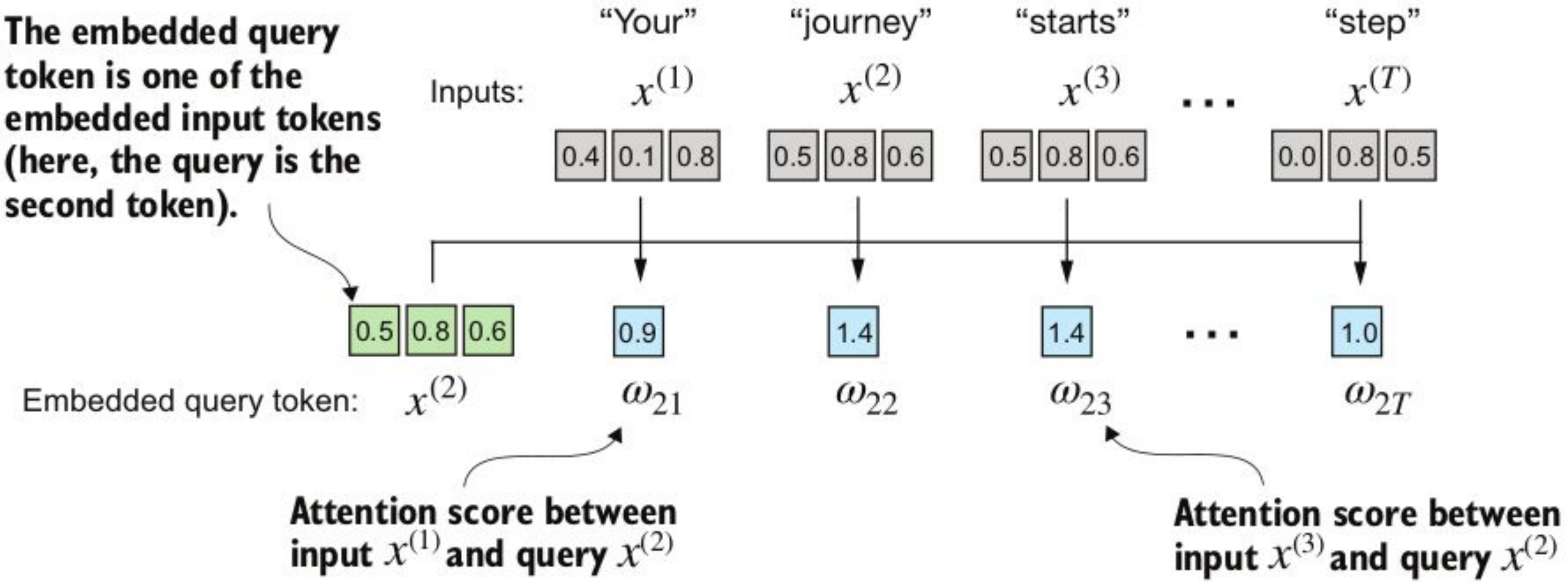
We first compute **attention scores**, and then normalise them into **attention weights**.

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 non-trainable attention scores: dot-product



Again just a matrix multiplication...

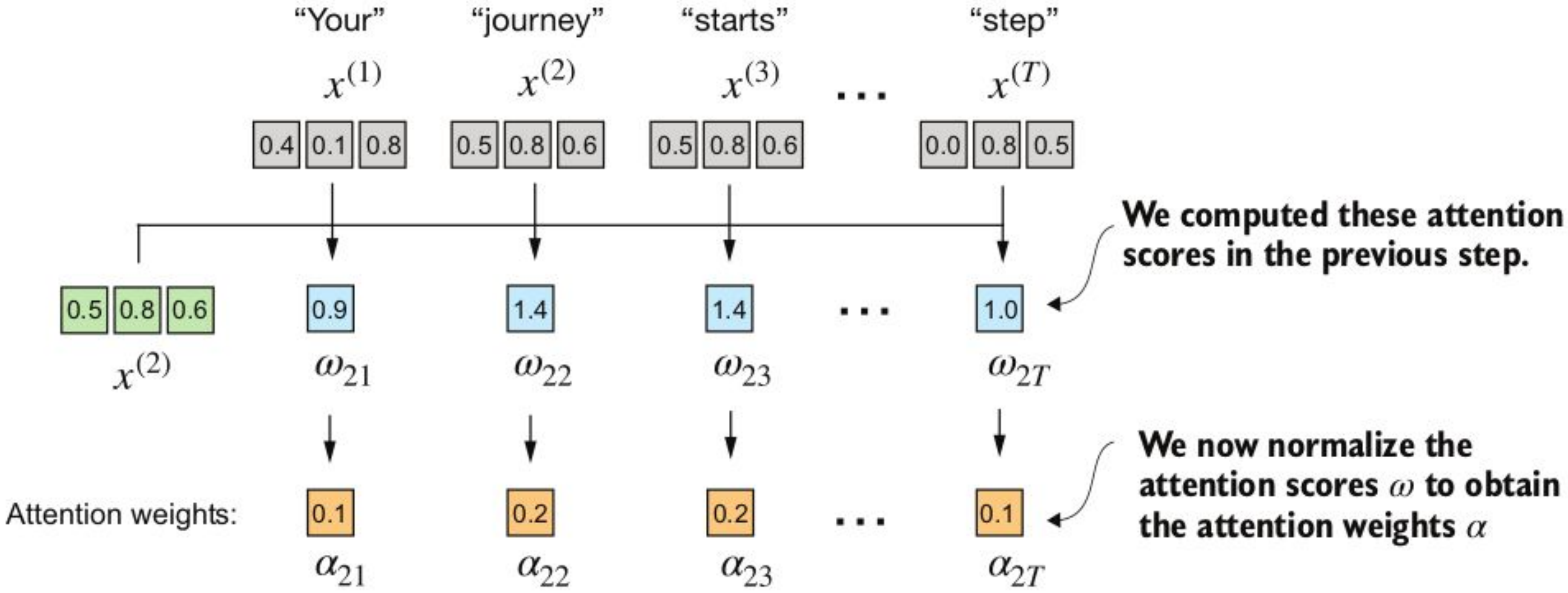
```
context_length = 3
embed_dim = 2

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 <u>non-trainable</u> 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

We divide the attention weights by sqrt(input_dim).

This implies that if x has unit variance, then the attention weights will have unit variance too and softmax will stay diffuse and not saturate too much.

Advanced question: formalise this claim.

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 **non-trainable**.

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

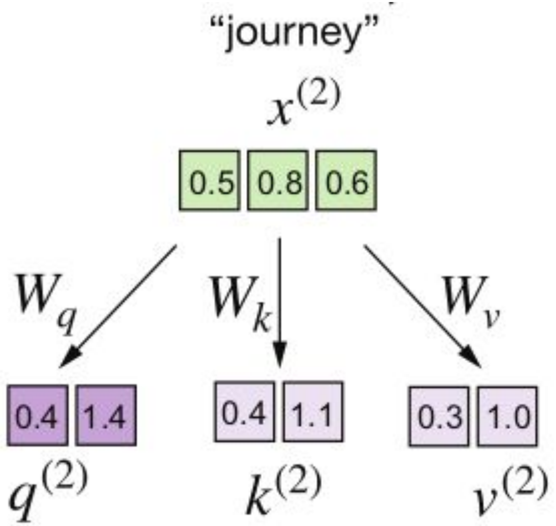
Output: for each token i:

- 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
```

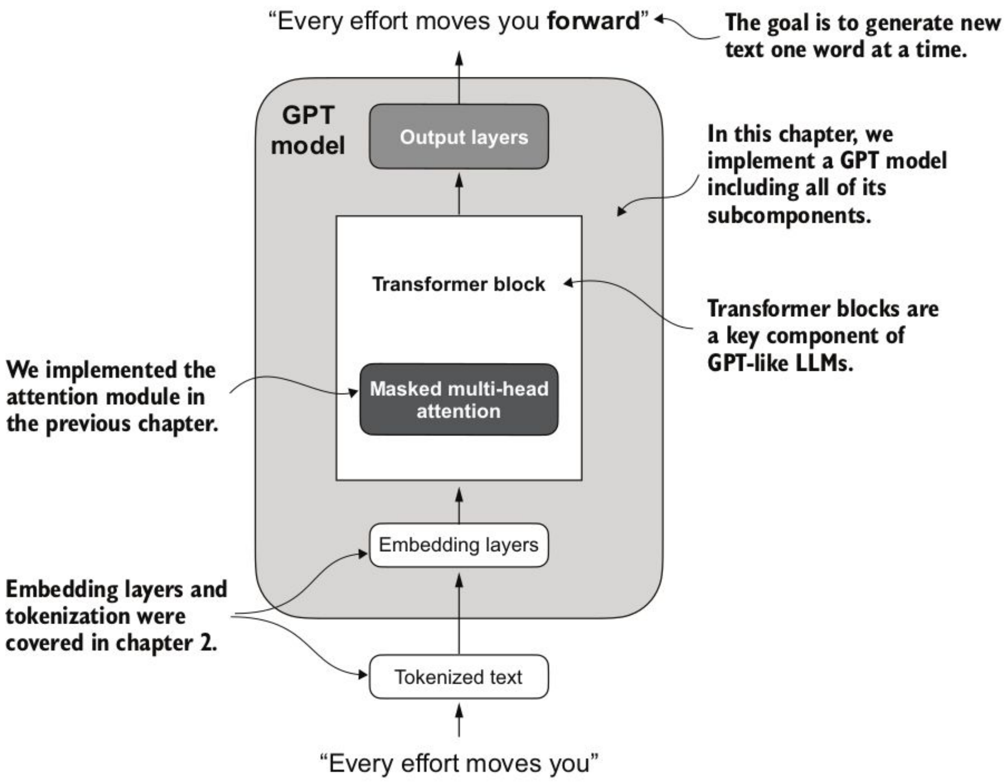
As a nn.Module

```python
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):
        T, C = x.shape
        # T = context_length
        # I = head_input_dim
        # H = head_size
        # O = head_output_dim

        k = self.key(x)   # (T, H)
        q = self.query(x) # (T, H)
        v = self.value(x) # (T, O)
        attention_scores = q @ k.T # (T, H) @ (H, T) -> (T, T)
        masked_attention_scores = attention_scores.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (T, T)
        attention_weights = torch.softmax(masked_attention_scores * self.head_size**-0.5, dim=-1) # (T, T)
        context_vectors = attention_weights @ v # (T, T) @ (T, O) -> (T, O)
        return context_vectors
```

We have a scaled-dot product self-attention head!!!

How can we use it?

Attention heads as key components in a Transformer

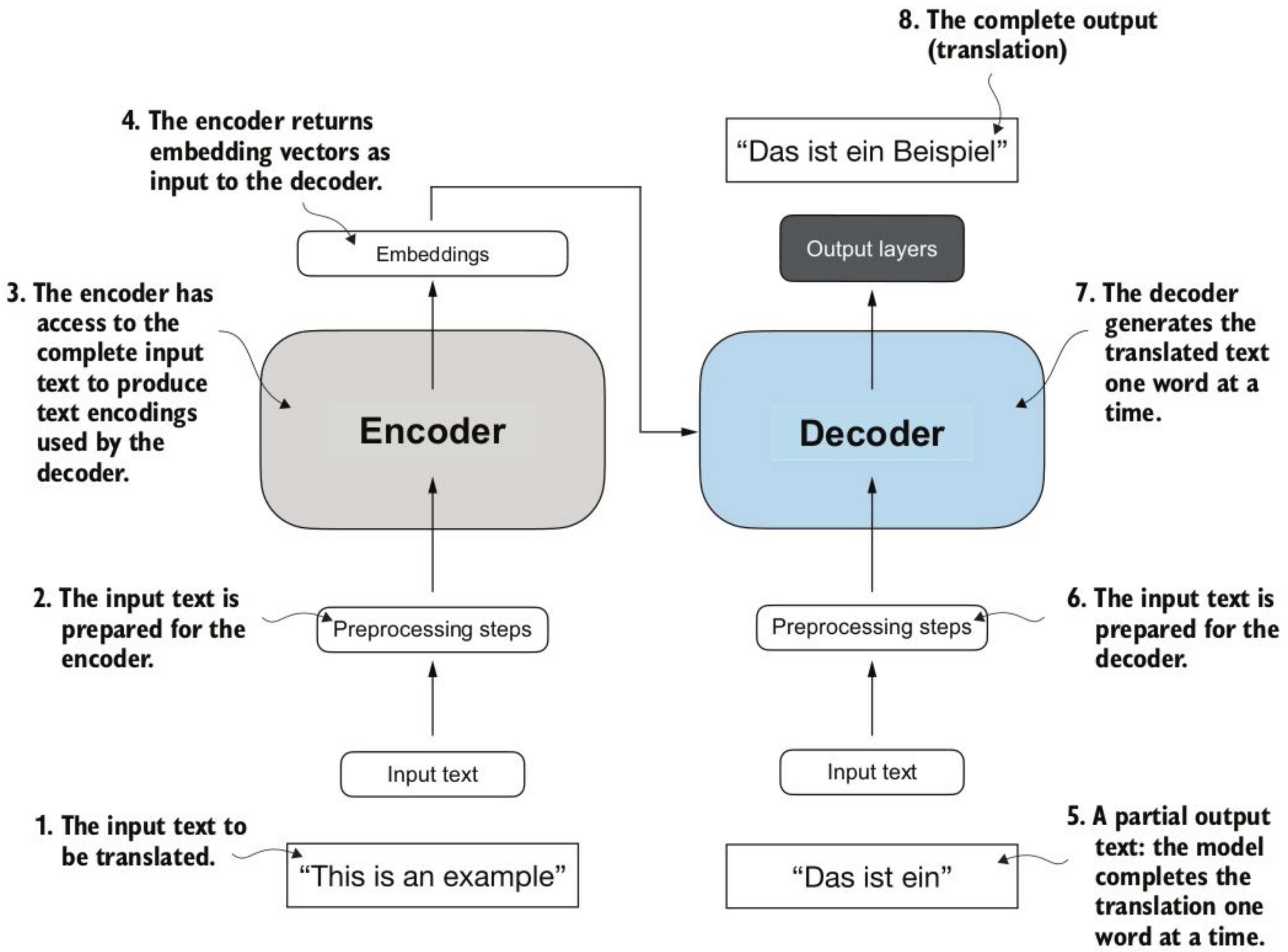


IMPORTANT

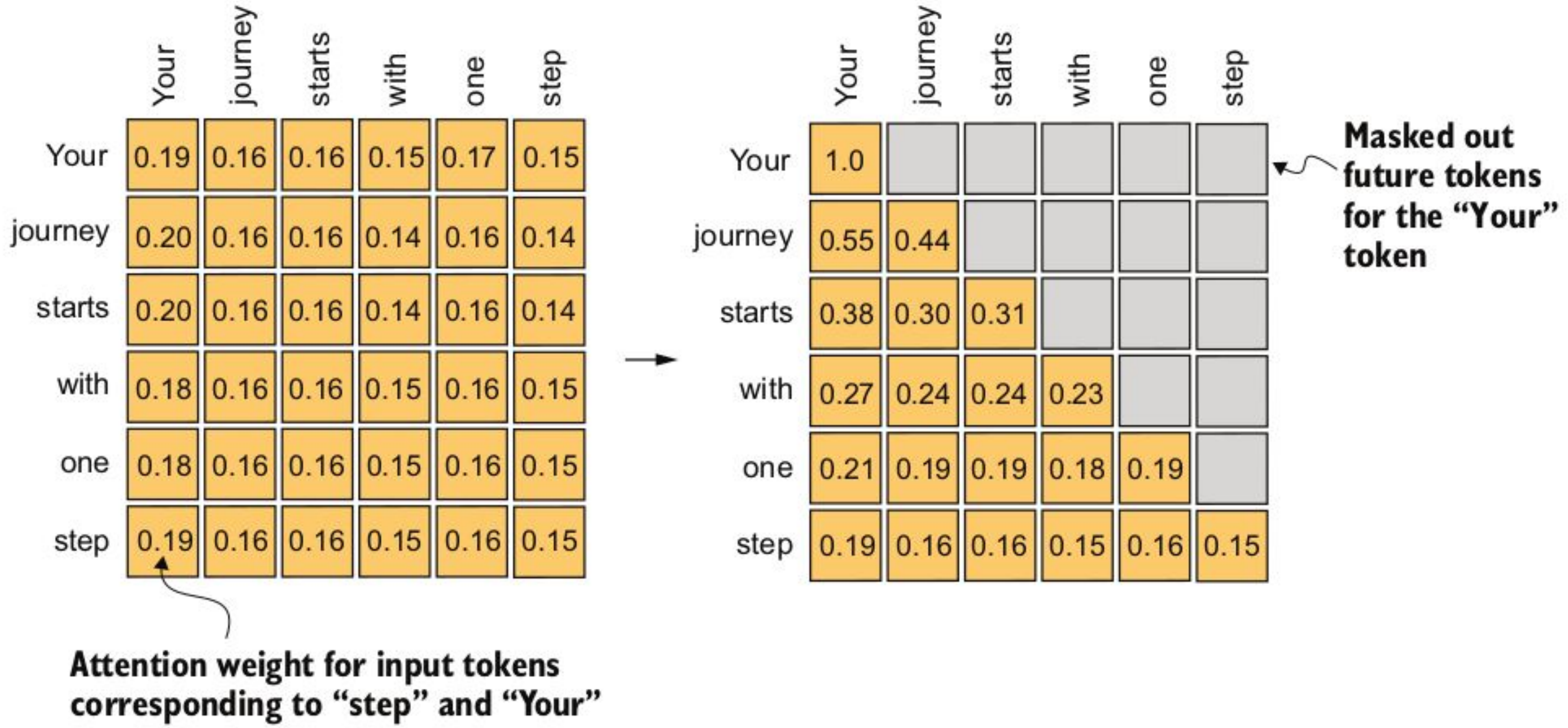
The Transformer consists of a number of “layers”, each with the same signature:

Input: a sequence of vectors, one for each token

Output: a sequence of vectors, one for each token

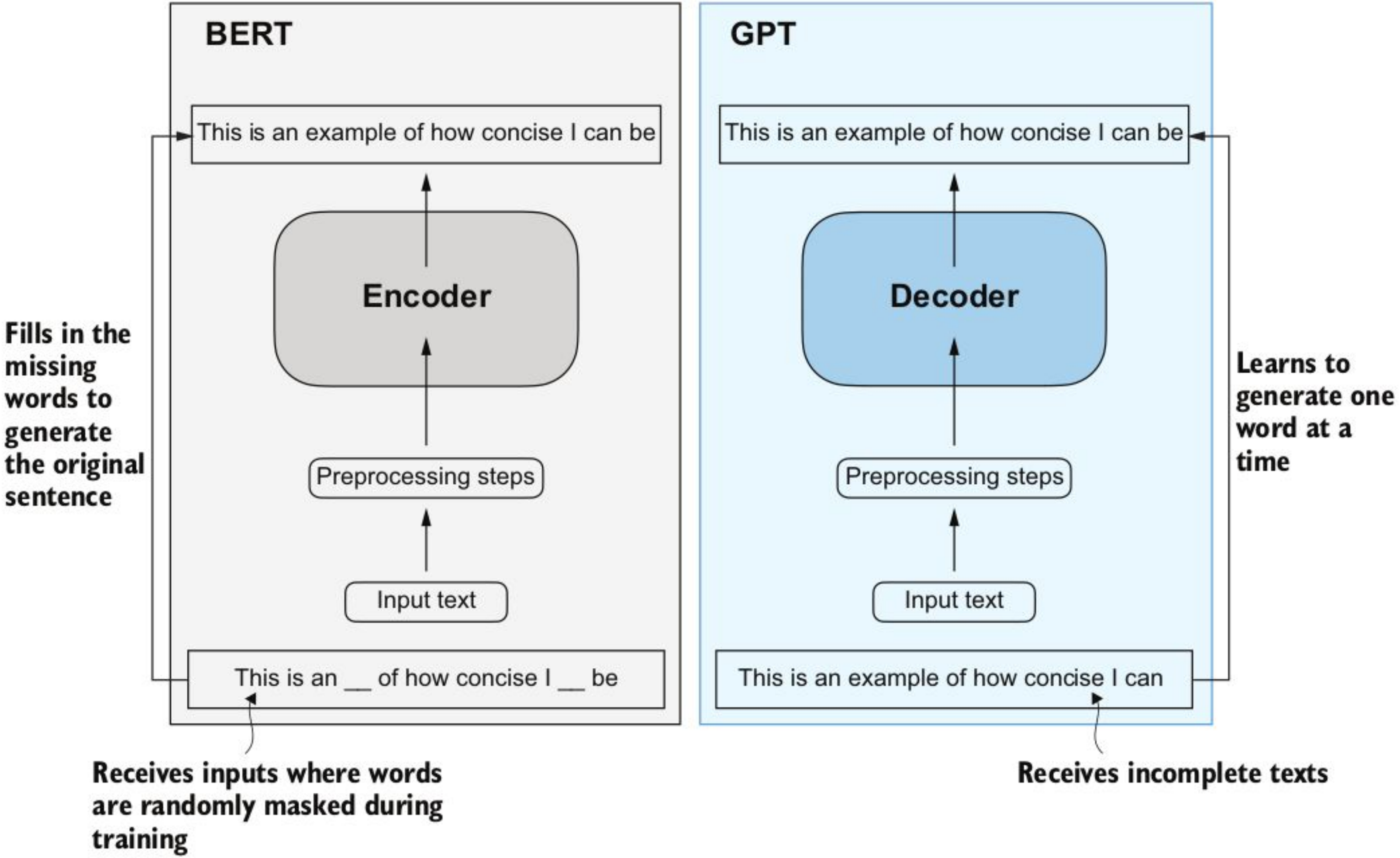

8. The complete output (translation)
"Das ist ein Beispiel"
4. The encoder returns embedding vectors as input to the decoder.
Embeddings
Output layers
3. The encoder has access to the complete input text to produce text encodings used by the decoder.
Encoder
Decoder
7. The decoder generates the translated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

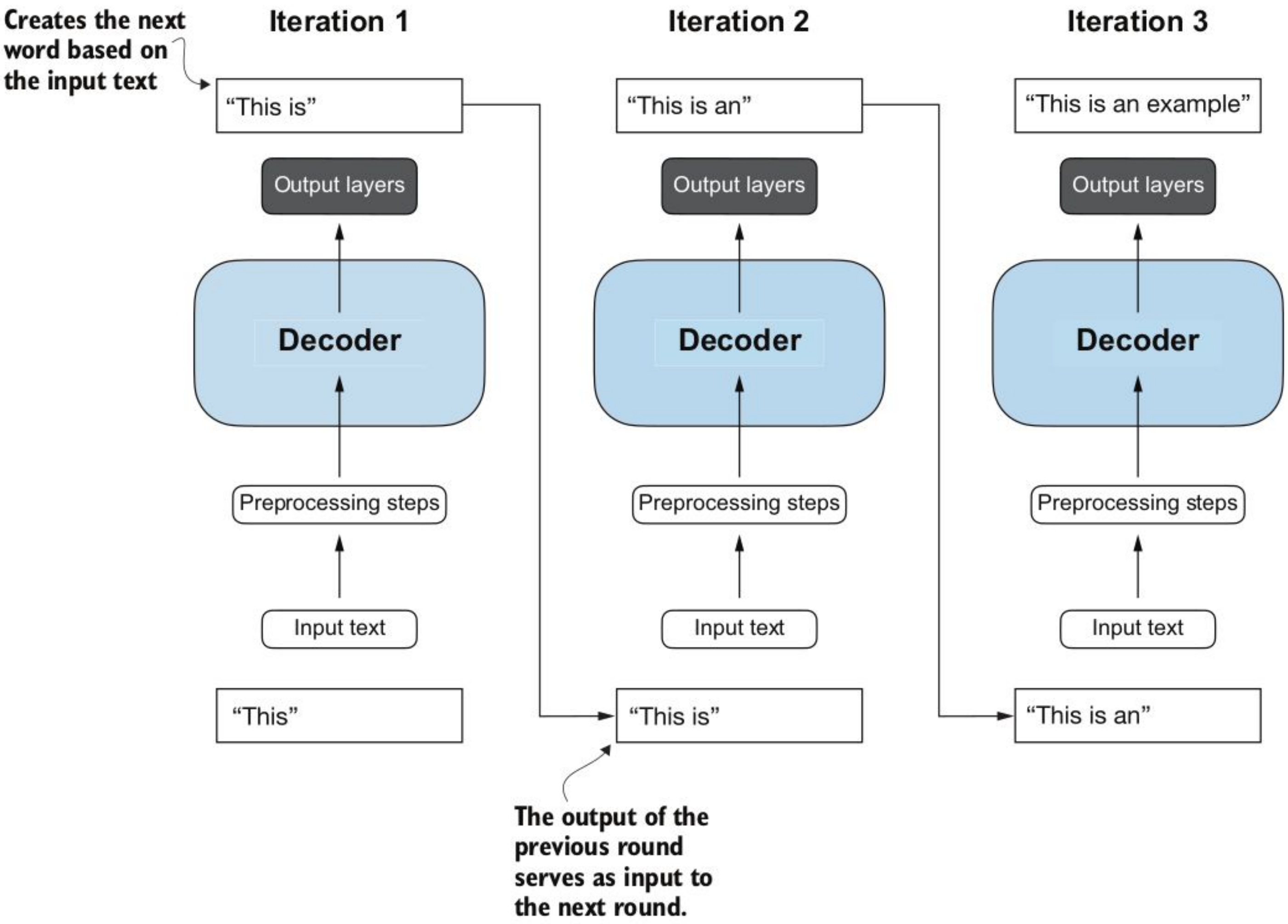
Decoders use causal attention



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

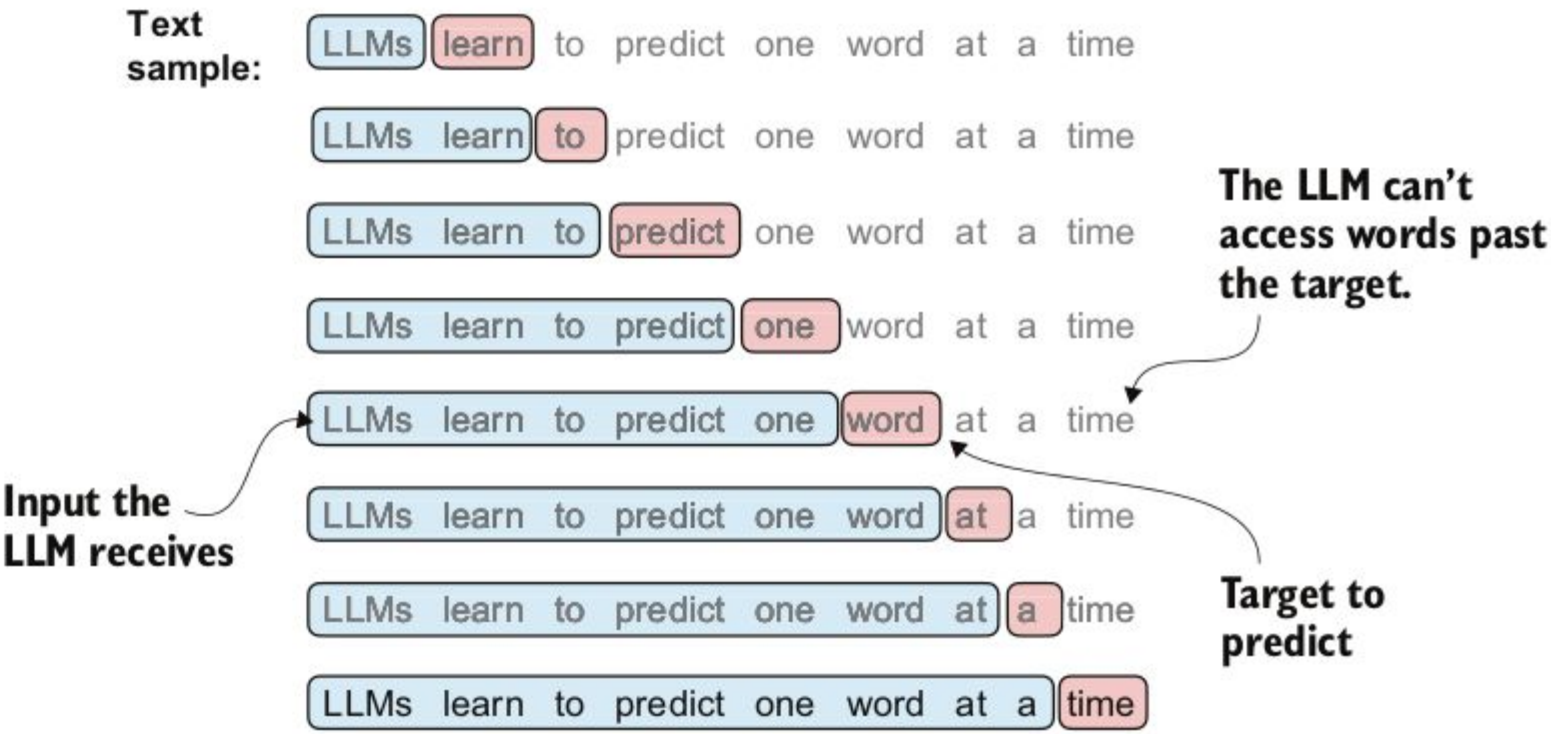

Creates the next word based on the input text
Iteration 1
Iteration 2
Iteration 3
"This is"
"This is an"
"This is an example"
Output layers
Output layers
Output layers
Decoder
Decoder
Decoder
Preprocessing steps
Preprocessing steps
Preprocessing steps
Input text
Input text
Input text
"This"
"This is"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

To make all of this useful we will need some more Deep Learning Magic

Deep Learning Magic

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

Sliding windows



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

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

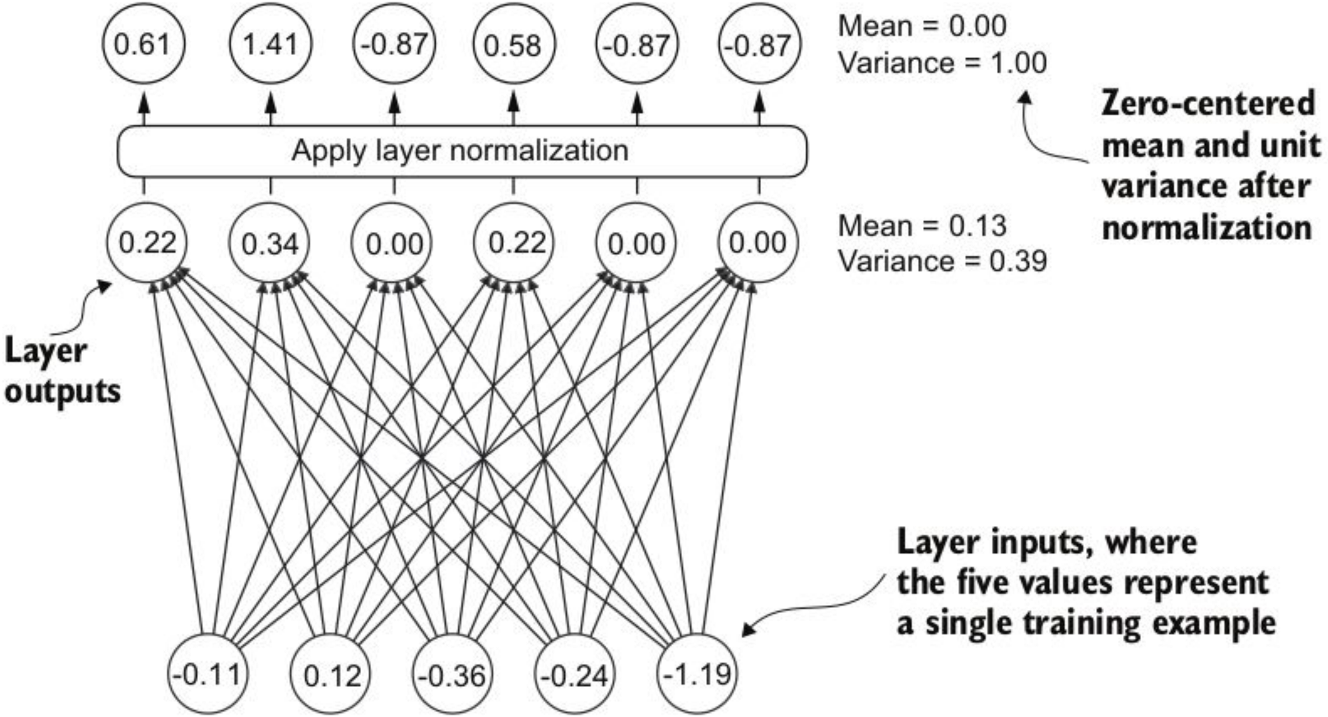
```
def train(model):
    # create a PyTorch optimizer
    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

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

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

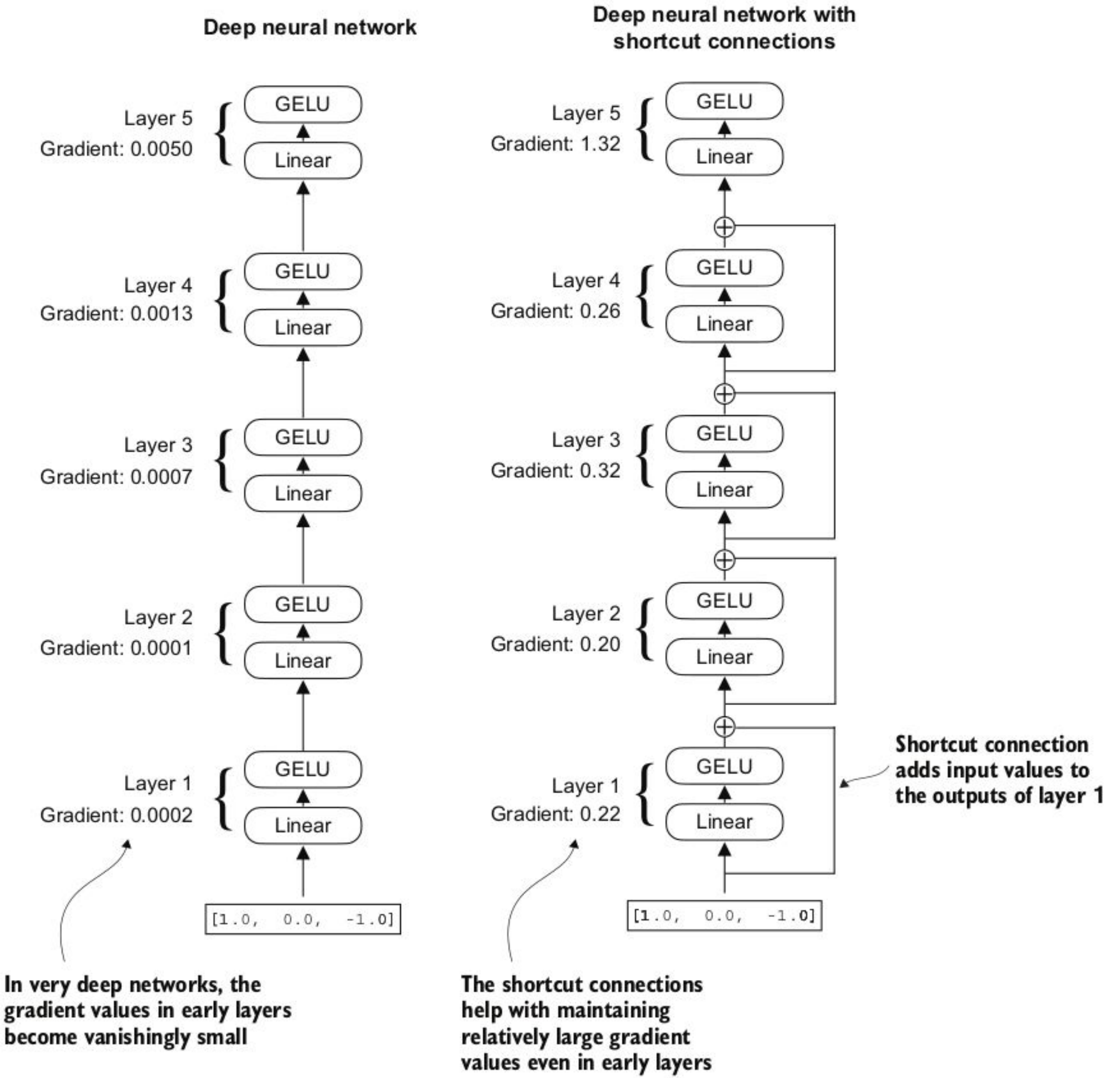
Cross entropy loss

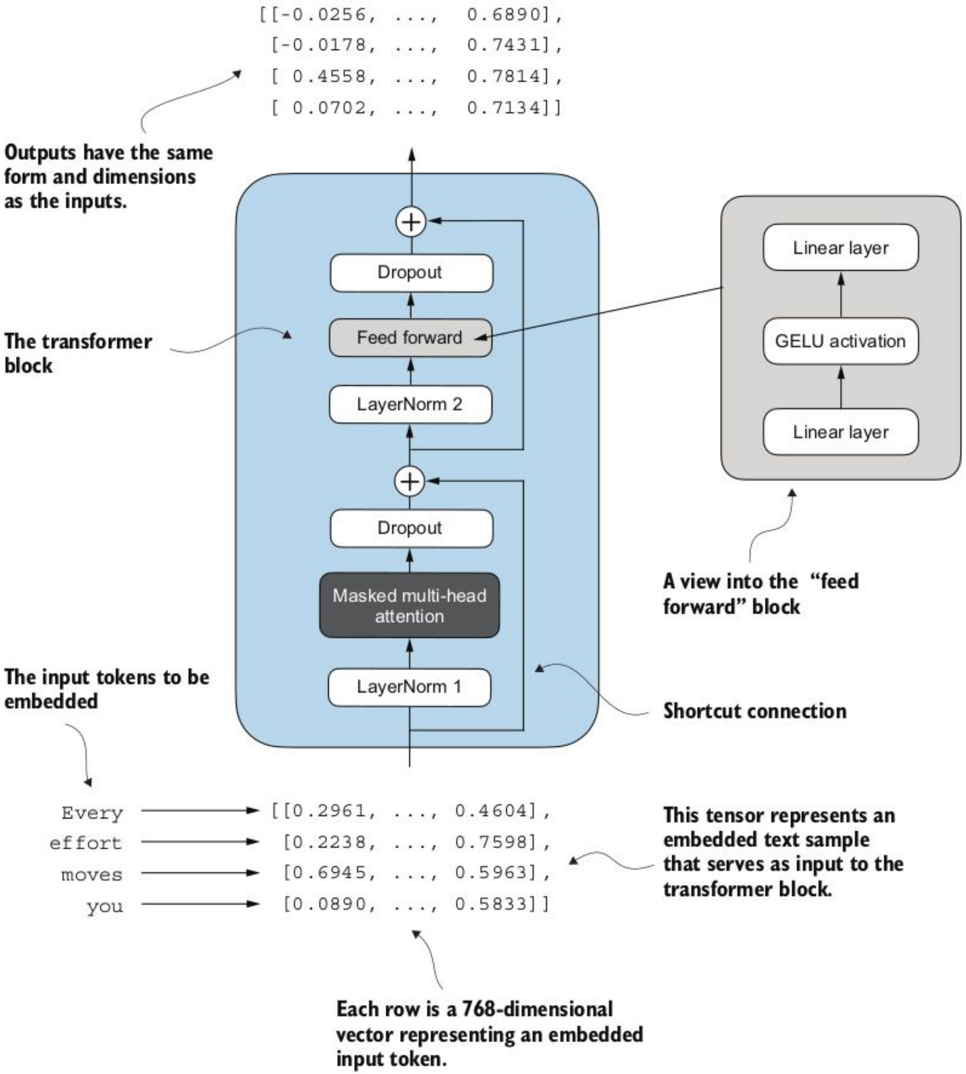
```python
vocab_size = 5

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
```

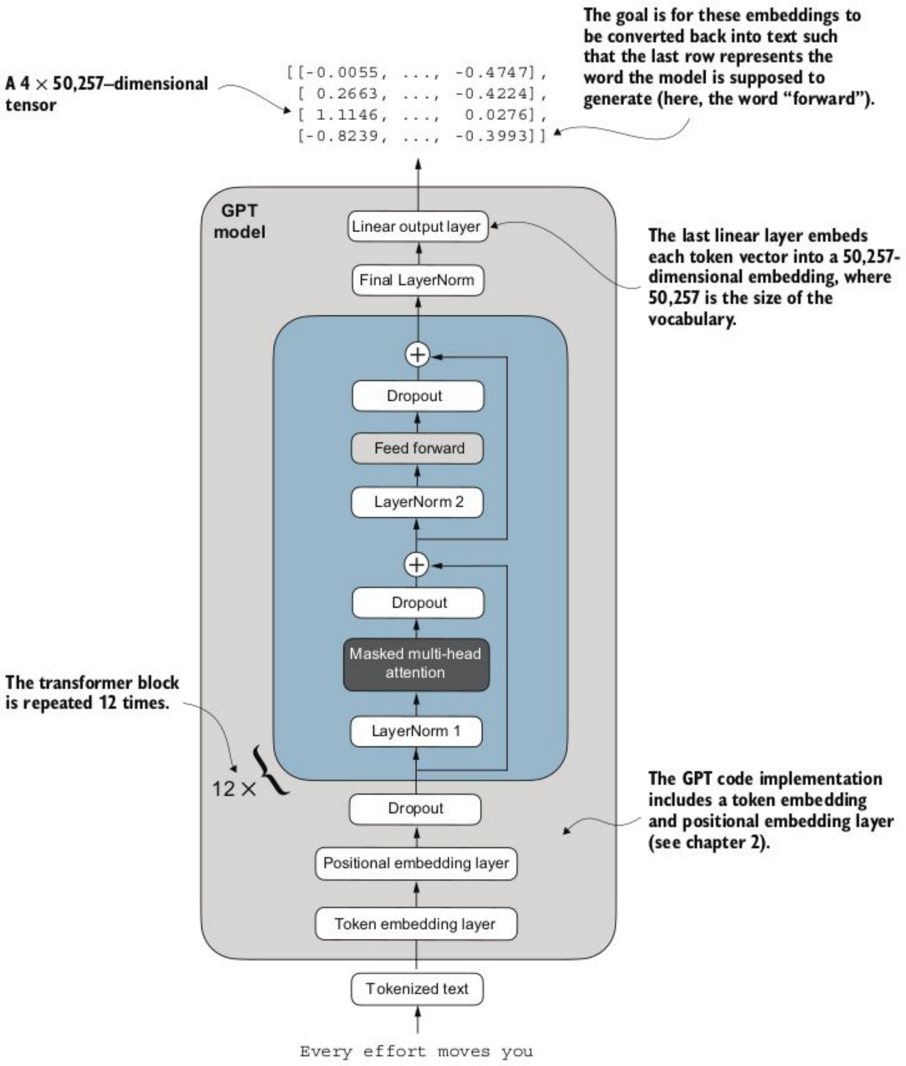
Layer normalization




Deep neural network
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]
In very deep networks, the gradient values in early layers become vanishingly small
Deep neural network with shortcut connections
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
[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



[[-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
The transformer block
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.



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?