

Large Language Models

Nathanaël Fijałkow

CNRS, LaBRI, Bordeaux

Co-teacher: Marc Lelarge



LaBRI

université
de 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** 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...

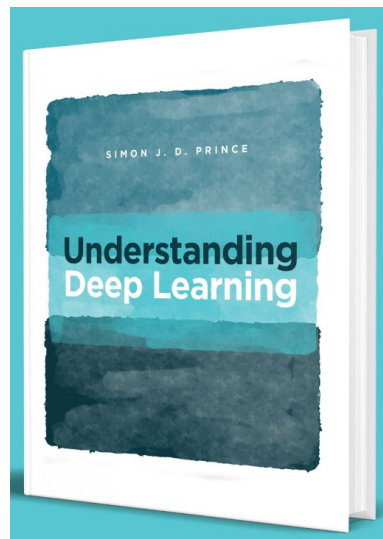
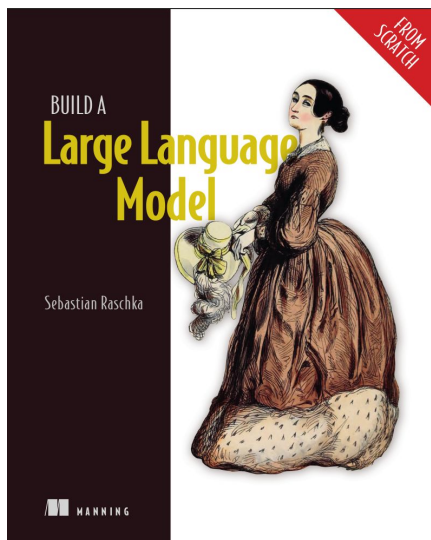
- 07.01 (Nath): Attention mechanism, Pre-training
- 14.01 (Nath): Tokenization, Fine-tuning
- 21.01 (Marc): Scaling laws, Probing
- 28.01 (Marc): Search strategies
- 04.02 (Nath): Retrieval-augmented generation
- 11.02 (Nath): Code generation
- 04.03 (Marc): Grammar decoders
- 11.03 (Marc): Value alignment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



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

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: $\text{vocab_size} \times \text{context_length}$

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

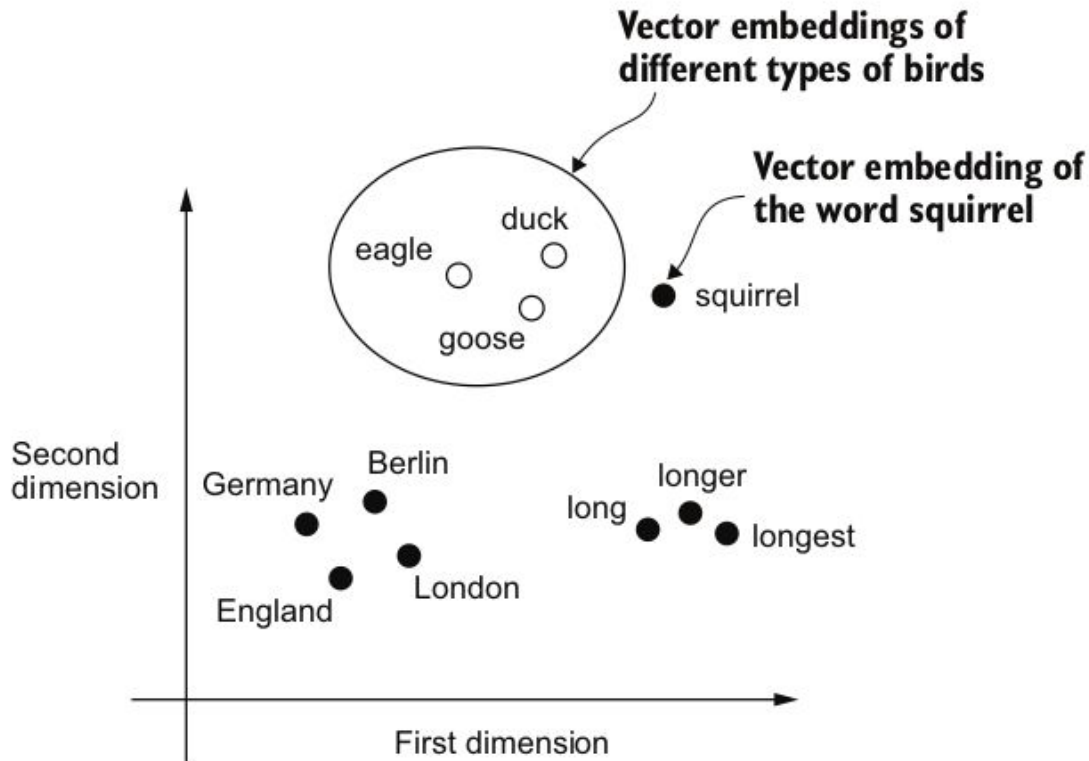
A Neural Probabilistic Language Model

Yoshua Bengio
Réjean Ducharme
Pascal Vincent
Christian Jauvin

Département d'Informatique et Recherche Opérationnelle
Centre de Recherche Mathématiques
Université de Montréal, Montréal, Québec, Canada

BENGIOY@IRO.UMONTREAL.CA
DUCHARME@IRO.UMONTREAL.CA
VINCENTP@IRO.UMONTREAL.CA
JAUVINC@IRO.UMONTREAL.CA

KEY IDEA: EMBEDDINGS



NN.EMBEDDING

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

```
n_token = 3
n_embed = 4
```

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

Weights of the embedding:

Parameter containing:

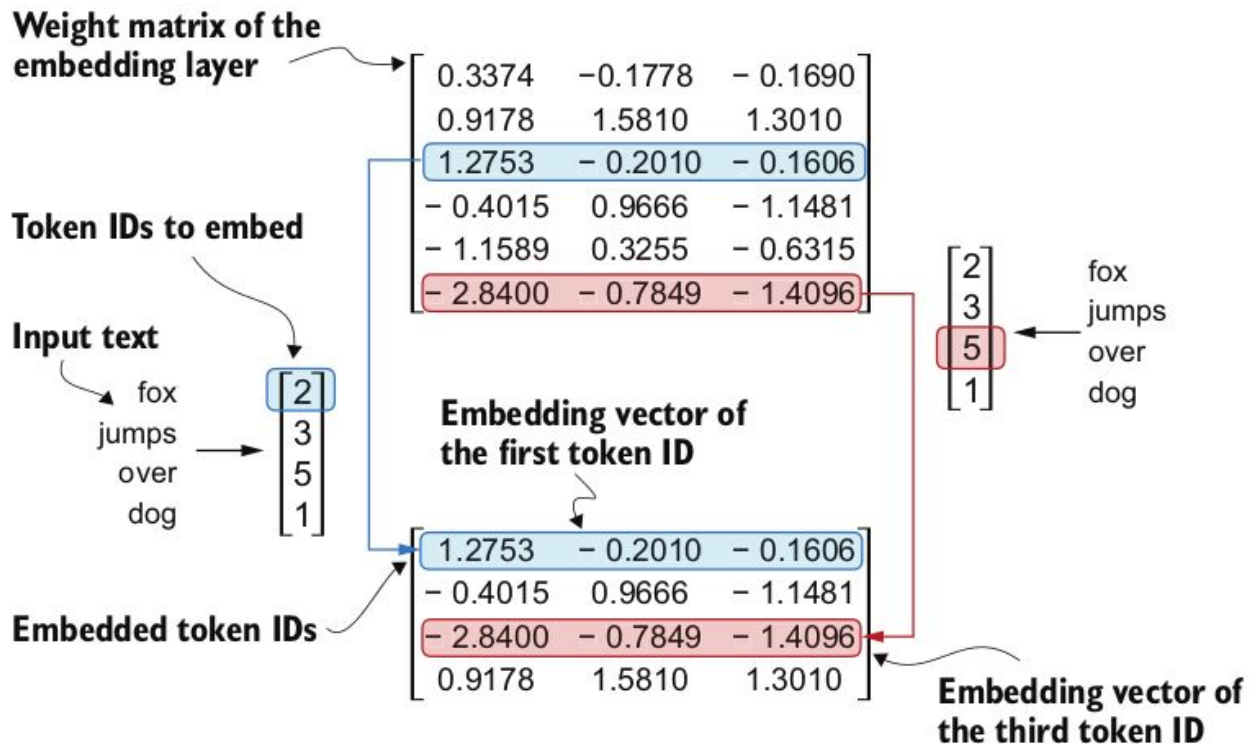
```
tensor([[ -0.9252,  0.8805, -0.0214,  0.9724],
        [ 0.1136,  0.2035,  1.1415,  0.0875],
        [ 0.4177,  0.6348,  0.6271,  0.1938]], requires_grad=True)
```

Result of embedding token number 1:

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

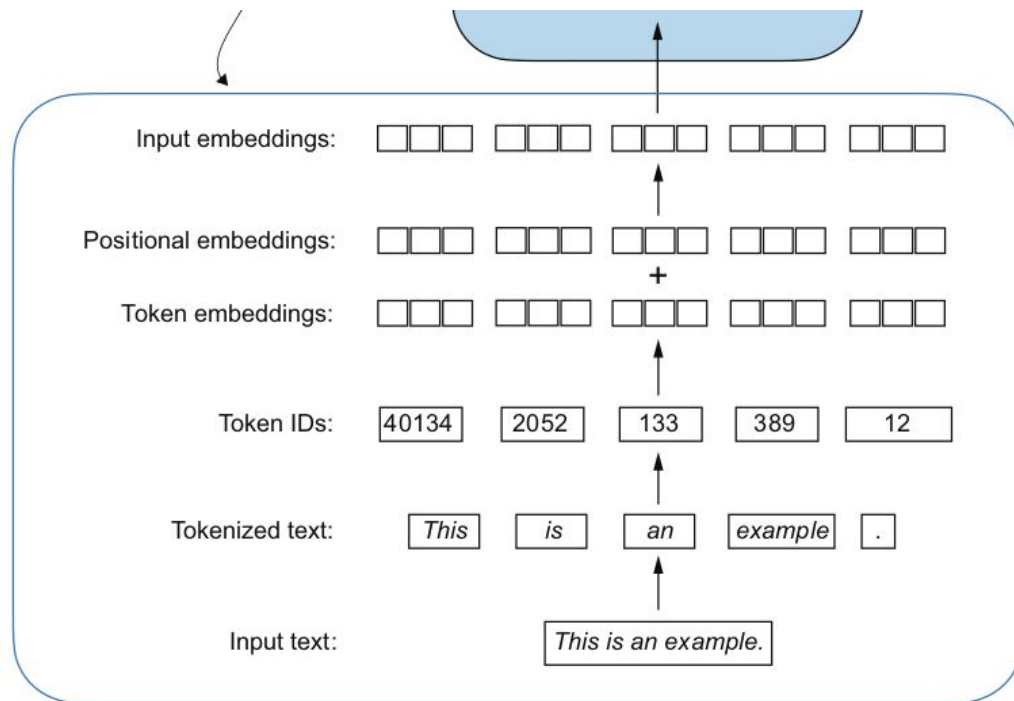
Advanced question: what is the difference between `nn.embedding` and `nn.linear`?

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.

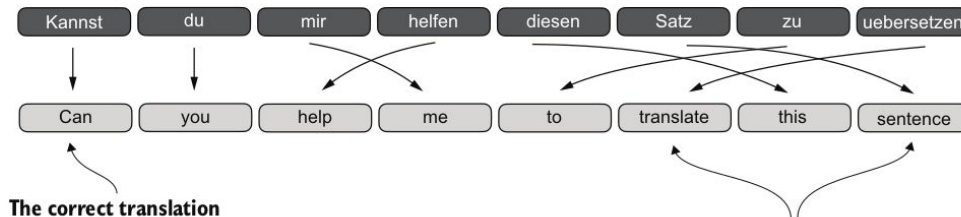
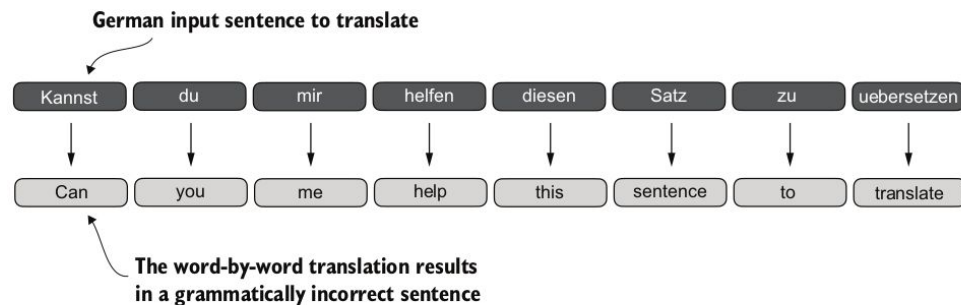
The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

MULTI-LAYER PERCEPTRON (MLP)

```
class MLP(nn.Module):
    def __init__(self):
        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



Certain words in the generated translation require access to words that appear earlier or later in the original sentence.

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

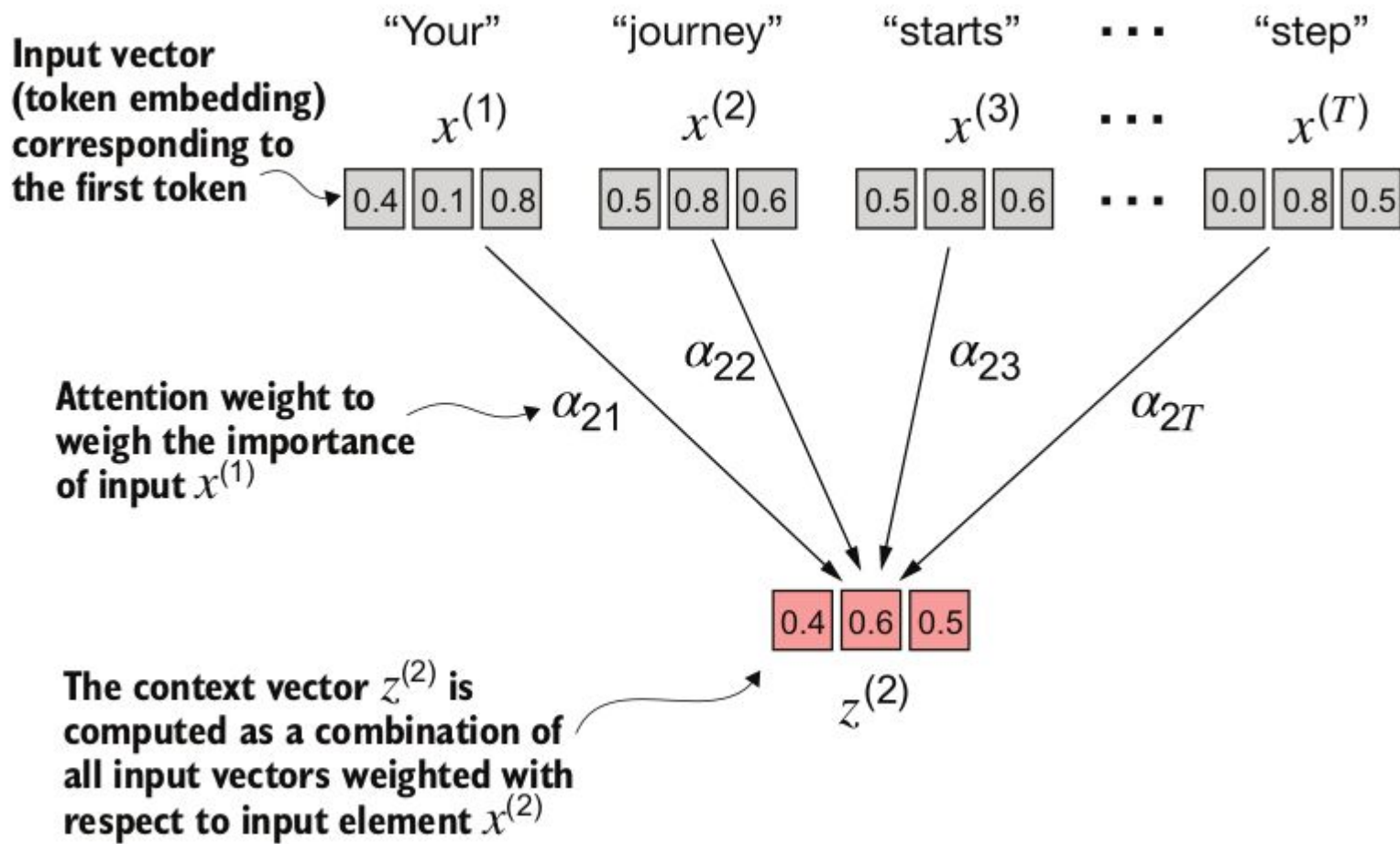
Input: an embedding vector $x(i)$ for each token i

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 .



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**.

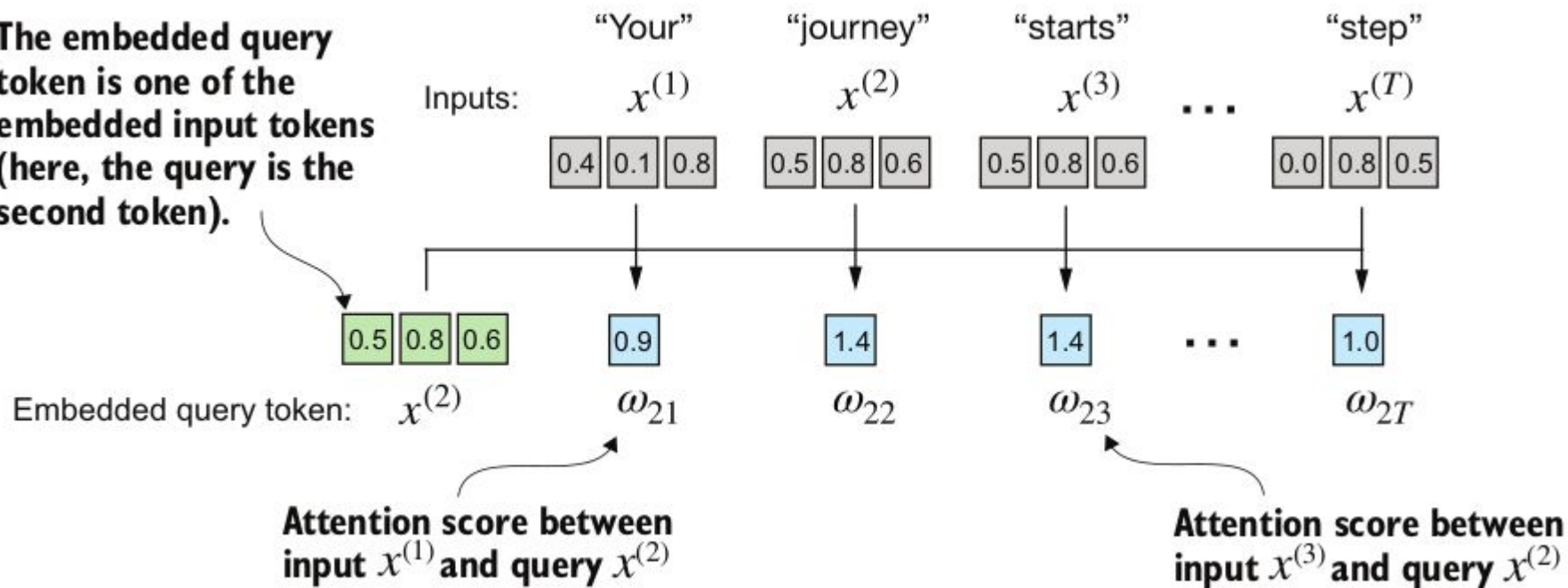
SIMPLIFICATION

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

The embedded query token is one of the embedded input tokens (here, the query is the second token).



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)

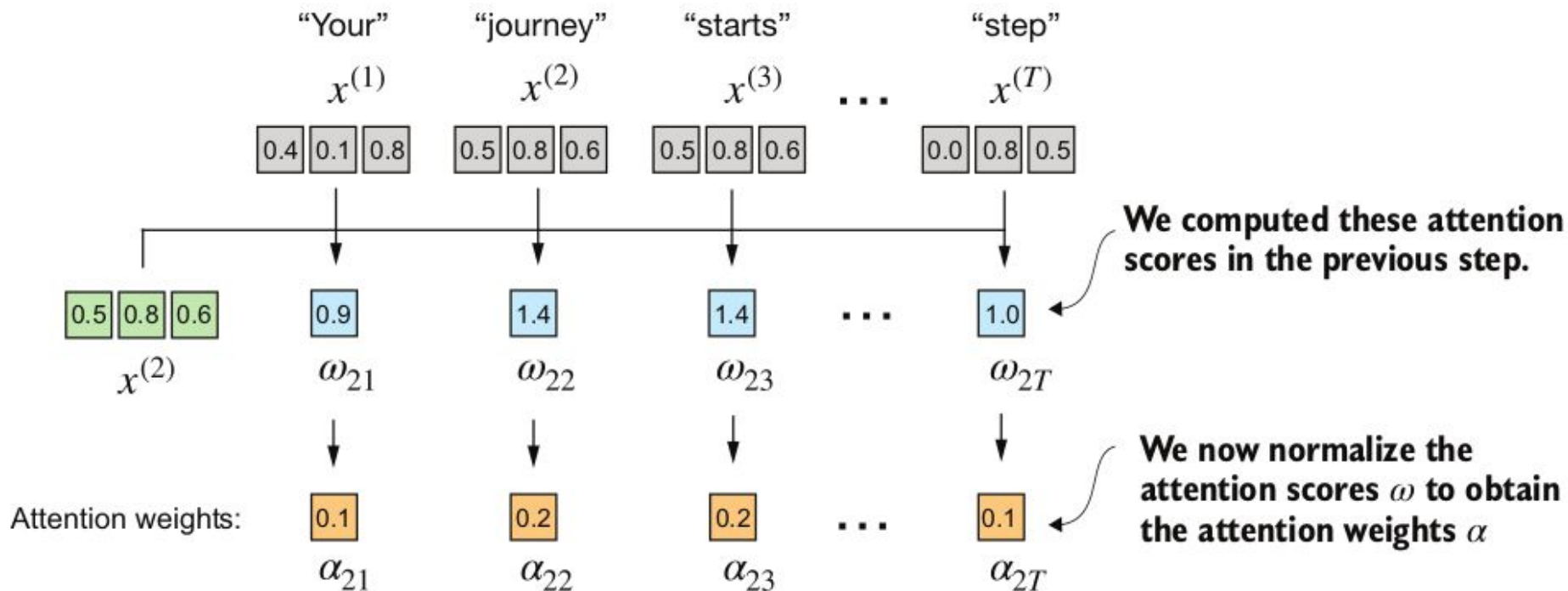
for i, x_i in enumerate(x):
    for j, x_j in enumerate(x):
        attention_scores[i, j] = torch.dot(x_i, x_j)
attention_scores
```

```
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_expded = attention_scores.exp()
print("After exponentiation: \n", scores_expded)
probs = scores_expded / scores_expded.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, *100.5])*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

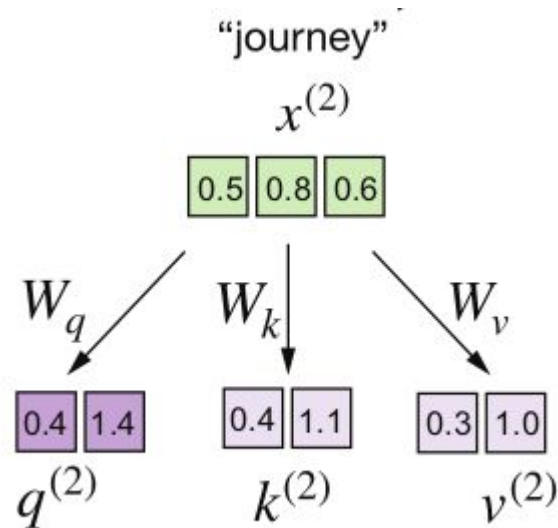
Output: for each token i :

- 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:

- W_q for query,
- W_k for key,
- W_v 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, 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)))

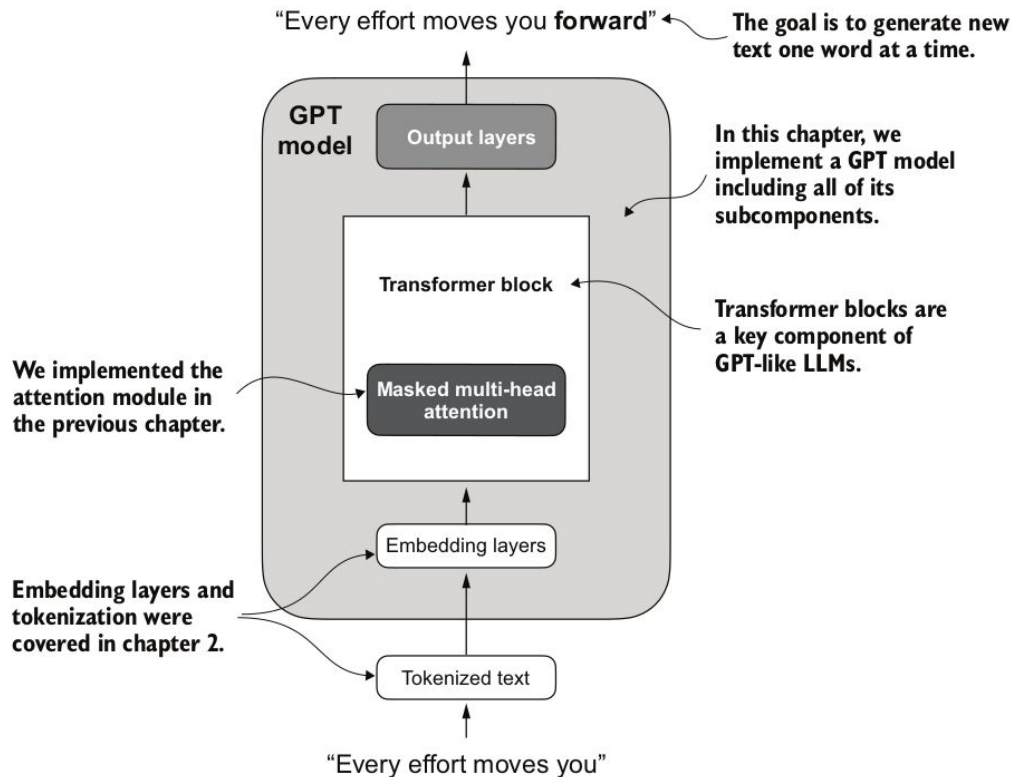
    def forward(self, x):
        T, C = x.shape
        # T = context_length
        # I = head_input_dim
        # H = head_size
        # O = head_output_dim

        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

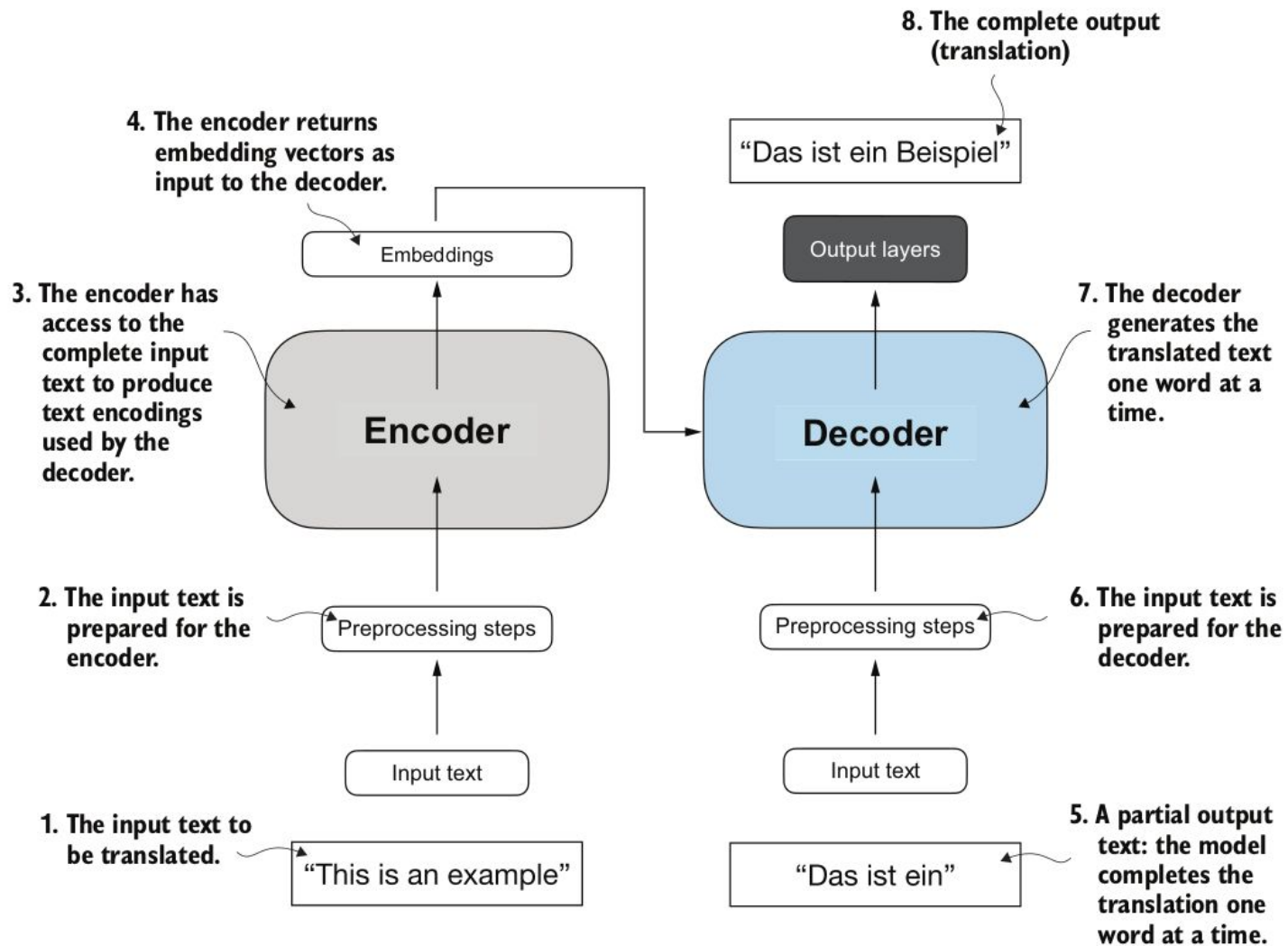


IMPORTANT

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



DECODERS USE CAUSAL ATTENTION

	Your	journey	starts	with	one	step
Your	0.19	0.16	0.16	0.15	0.17	0.15
journey	0.20	0.16	0.16	0.14	0.16	0.14
starts	0.20	0.16	0.16	0.14	0.16	0.14
with	0.18	0.16	0.16	0.15	0.16	0.15
one	0.18	0.16	0.16	0.15	0.16	0.15
step	0.19	0.16	0.16	0.15	0.16	0.15

Attention weight for input tokens
corresponding to “step” and “Your”



	Your	journey	starts	with	one	step
Your	1.0					
journey	0.55	0.44				
starts	0.38	0.30	0.31			
with	0.27	0.24	0.24	0.23		
one	0.21	0.19	0.19	0.18	0.19	
step	0.19	0.16	0.16	0.15	0.16	0.15

Masked out
future tokens
for the “Your”
token

IMPLEMENTATION OF THE MASK

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

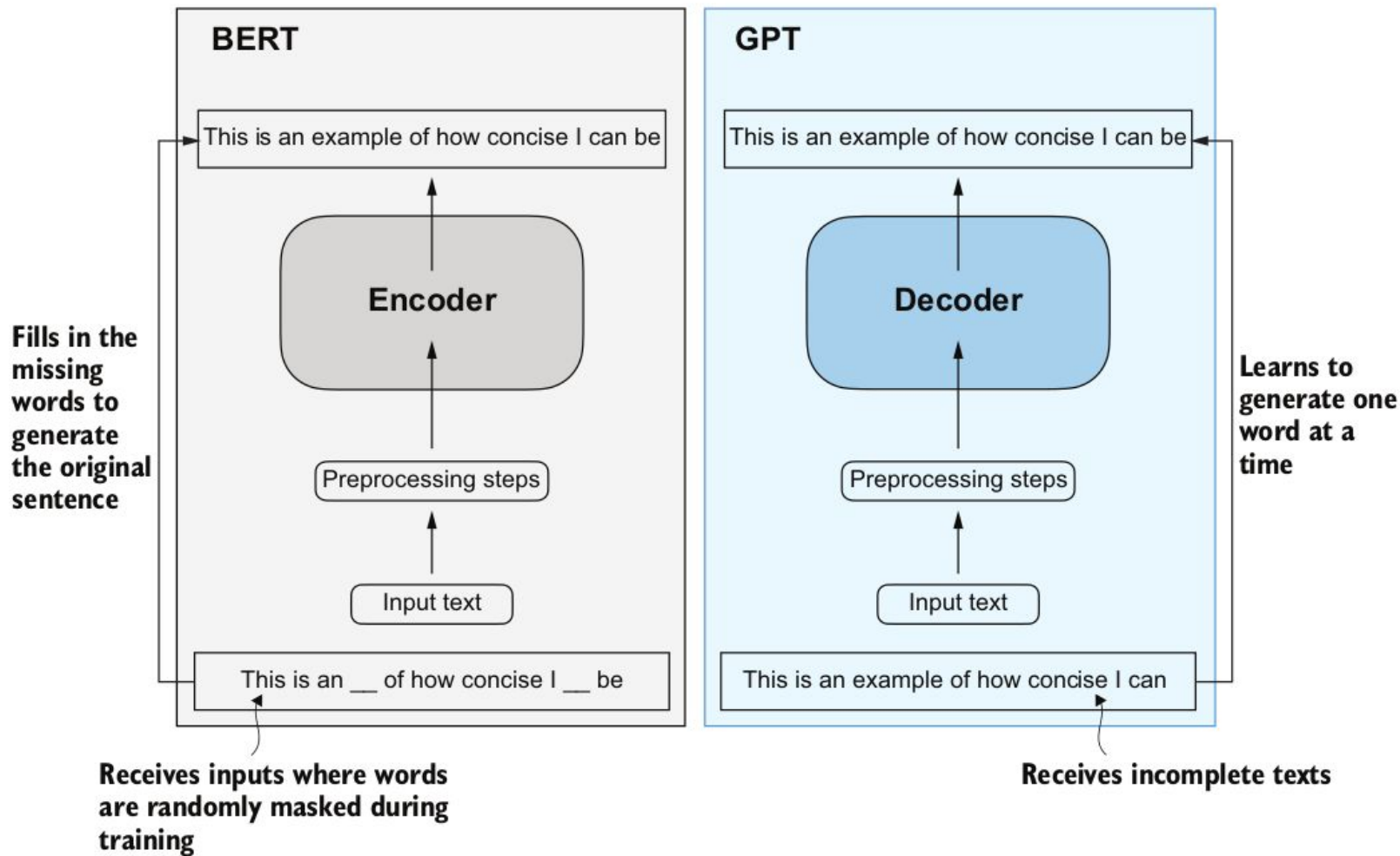
key = nn.Linear(input_dim, head_dim, bias=False)
query = nn.Linear(input_dim, head_dim, bias=False)
value = nn.Linear(input_dim, output_dim, bias=False)

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

attention_scores = q @ k.T

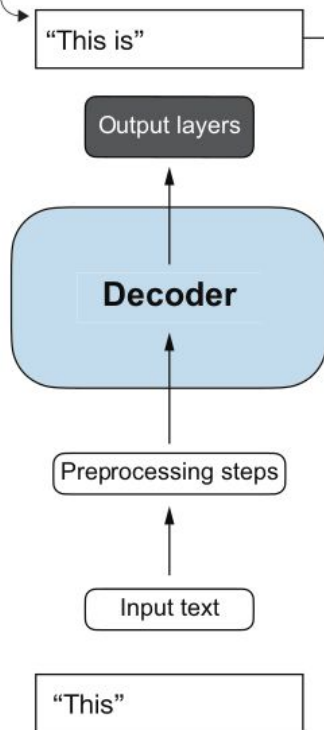
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked_attention_scores = attention_scores.masked_fill(mask.bool(), -torch.inf)

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

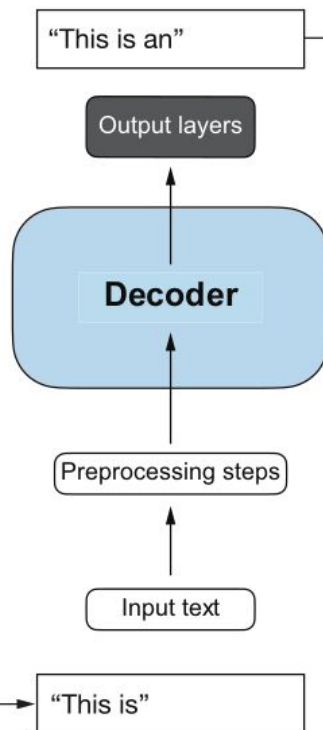


Creates the next word based on the input text

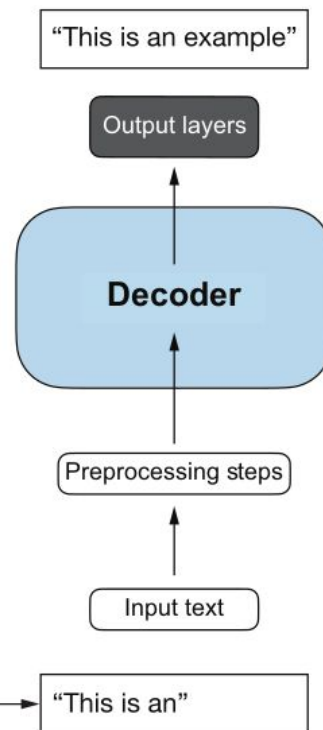
Iteration 1



Iteration 2



Iteration 3



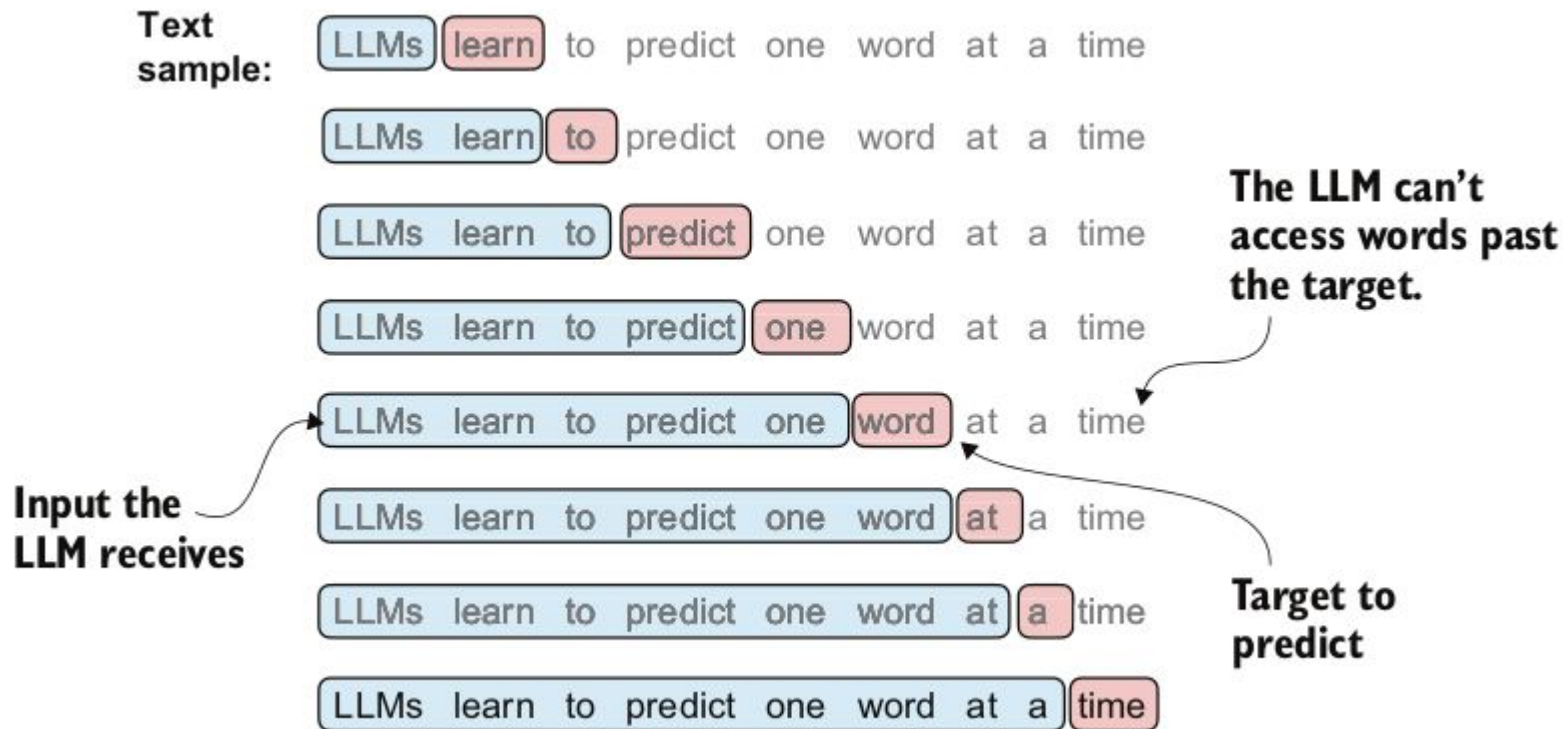
The output of the previous round serves as input to the next round.

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



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
```

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
```

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

```
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)

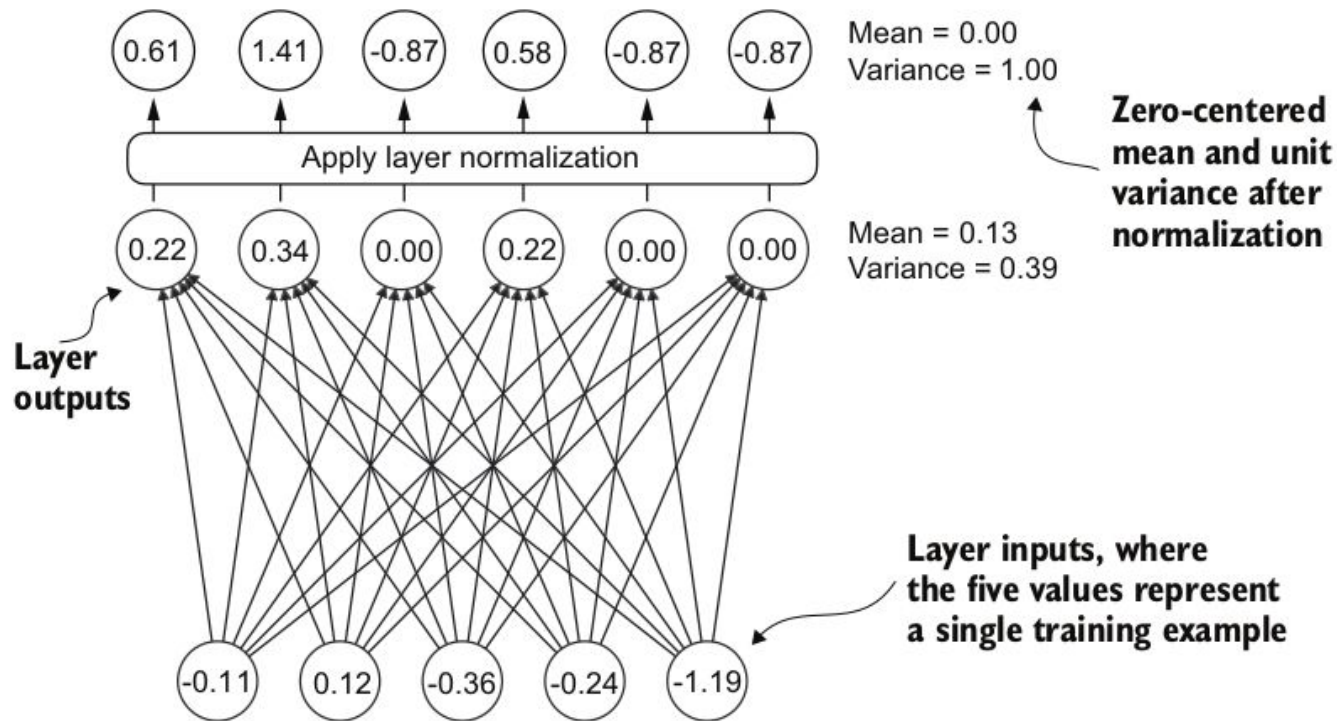
y = torch.randint(vocab_size, (), dtype=torch.int64)
print("\nLet us consider a target y: ", y.item())

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

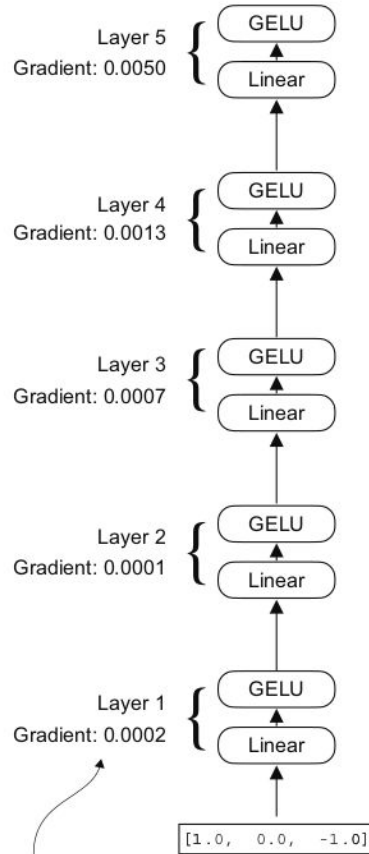
```
The logits:
  tensor([ 0.0465,  0.2514, -0.6639, -0.5434, -0.0025])
After softmax:
  tensor([0.2367, 0.2905, 0.1163, 0.1312, 0.2253])
The -log probabilities:
  tensor([1.4411, 1.2362, 2.1516, 2.0310, 1.4901])
```

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

LAYER NORMALIZATION

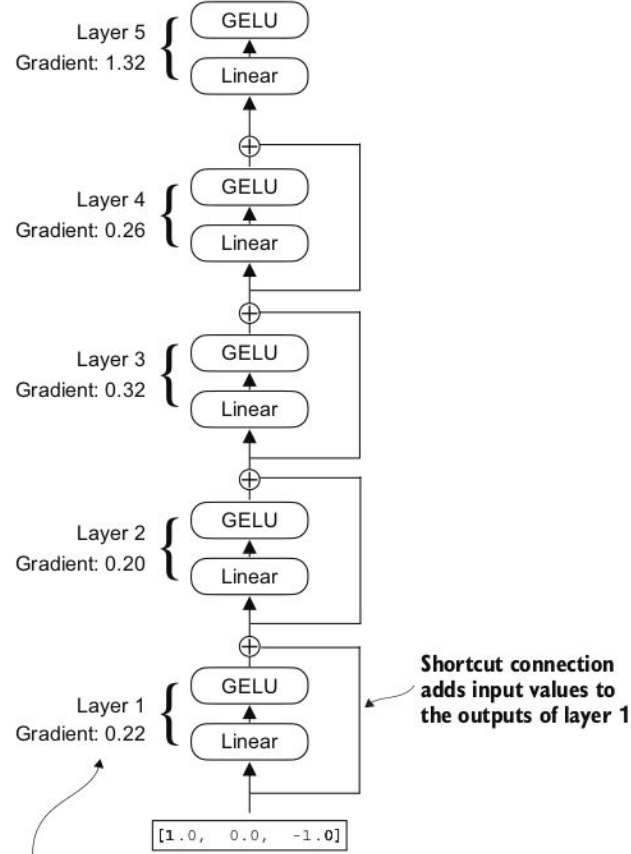


Deep neural network

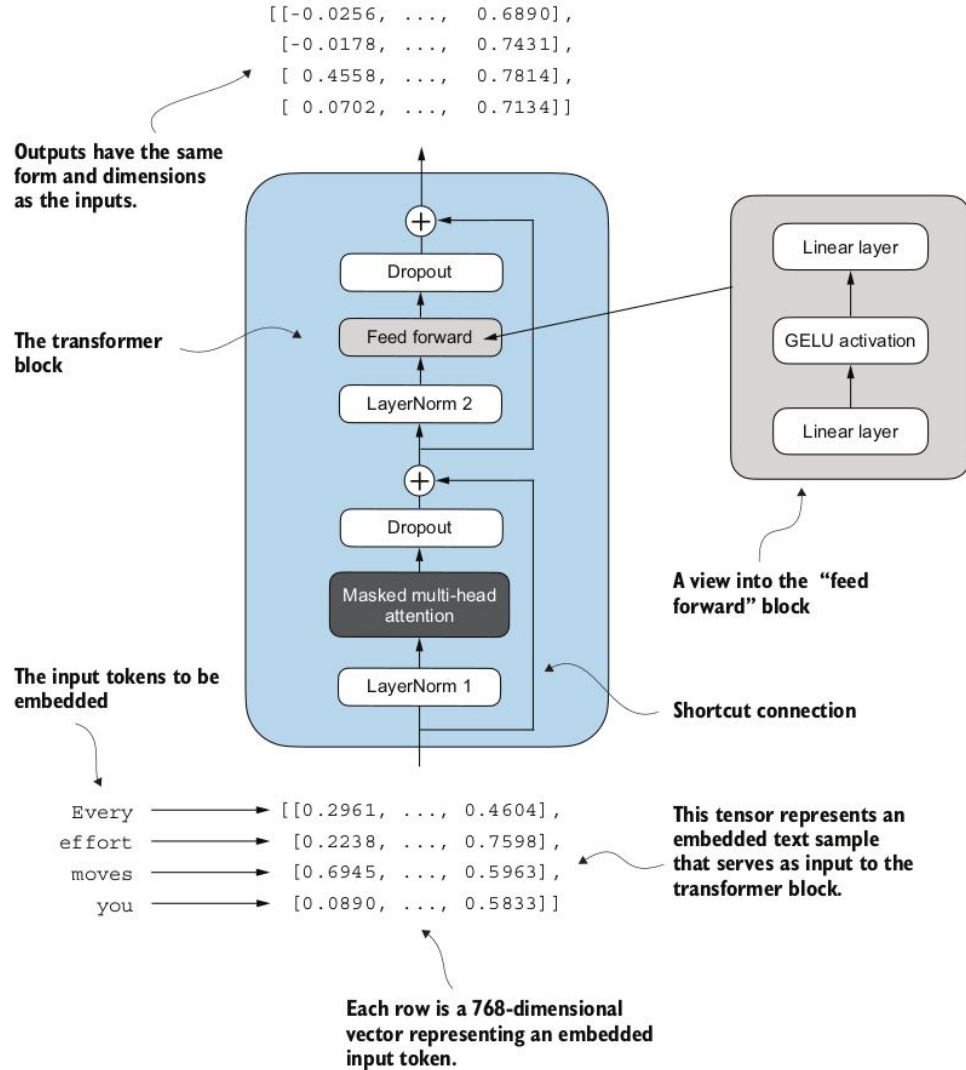


In very deep networks, the gradient values in early layers become vanishingly small

Deep neural network with shortcut connections



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

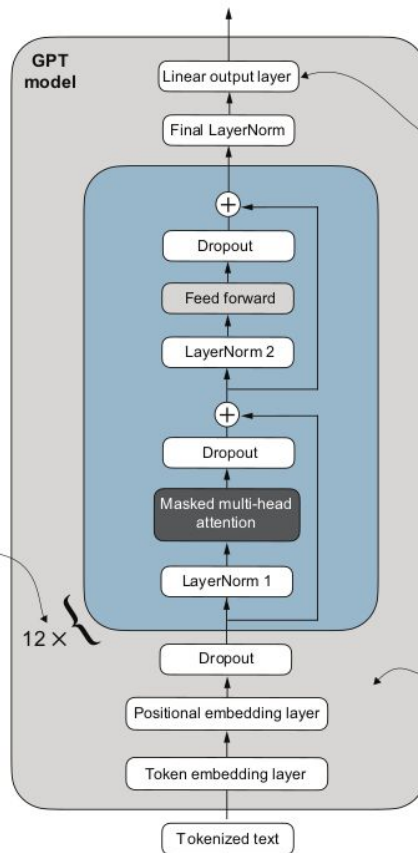


A $4 \times 50,257$ -dimensional tensor

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

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

The transformer block is repeated 12 times.



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

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

Every effort moves you

SHALL WE LOOK AT SOME ACTUAL CODE?