Large Language Models: Transformers from scratch

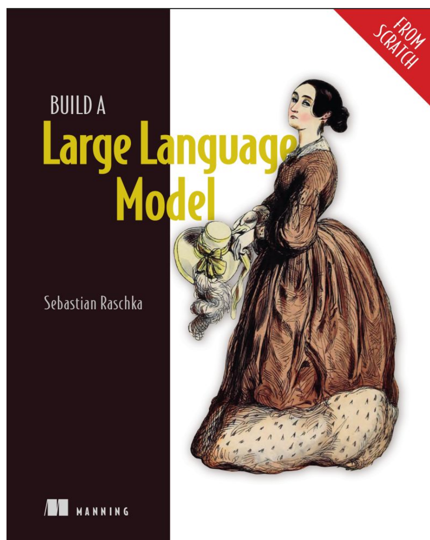
Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux

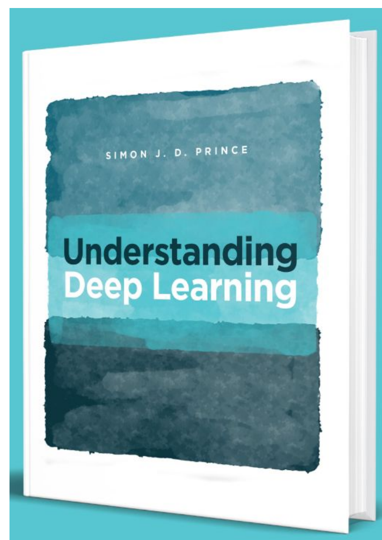
LaBRI

université de BORDEAUX

Some references

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





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

Other sources are mentioned when used

Language models

What is a language model (LM)?

Input: a sentence (as a sequence of tokens)

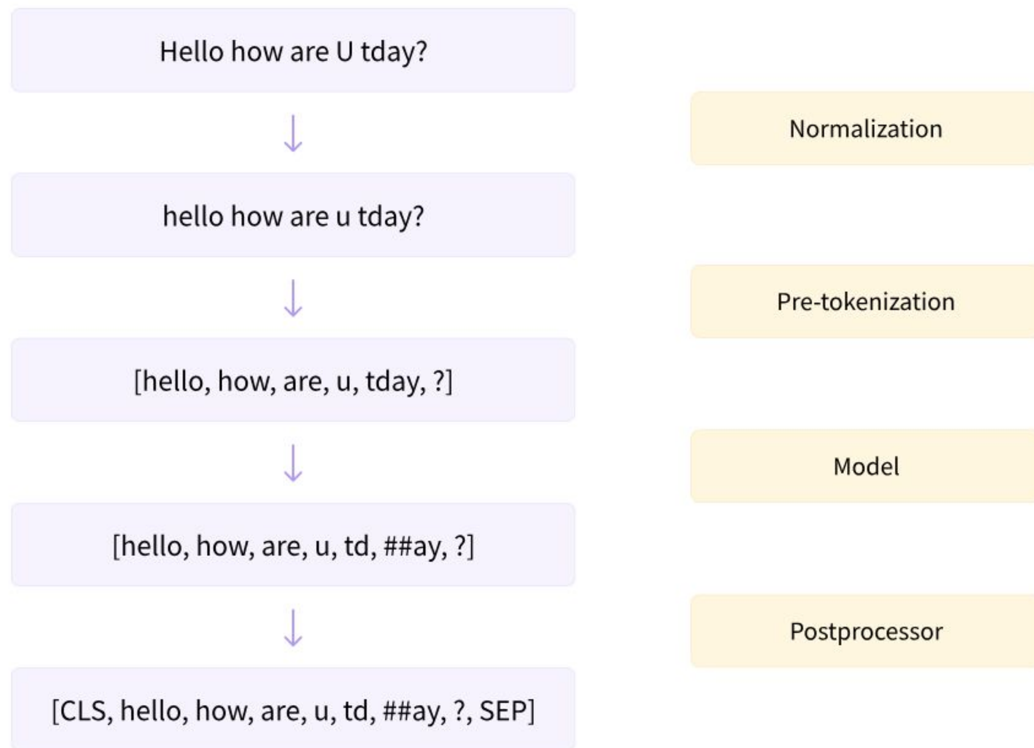
Output: predict the next token

Basic examples:

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

TOKENIZATION

Tokenization: decomposing a sentence into a sequence of tokens



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

Words != Tokens

What it actually looks like:

```
test = "hello world"
test_encoded = tokenizer.encode(test)
test_encoded, [tokenizer.decode([x]) for x in test_encoded], tokenizer.decode(test_encoded)
```

```
([258, 285, 111, 492], ['he', 'll', 'o', ' world'], 'hello world')
```

Tokenization is important, we'll talk about it later!

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

GPT-2 has 50,257 tokens

Embeddings and the Multi-layer Perceptron

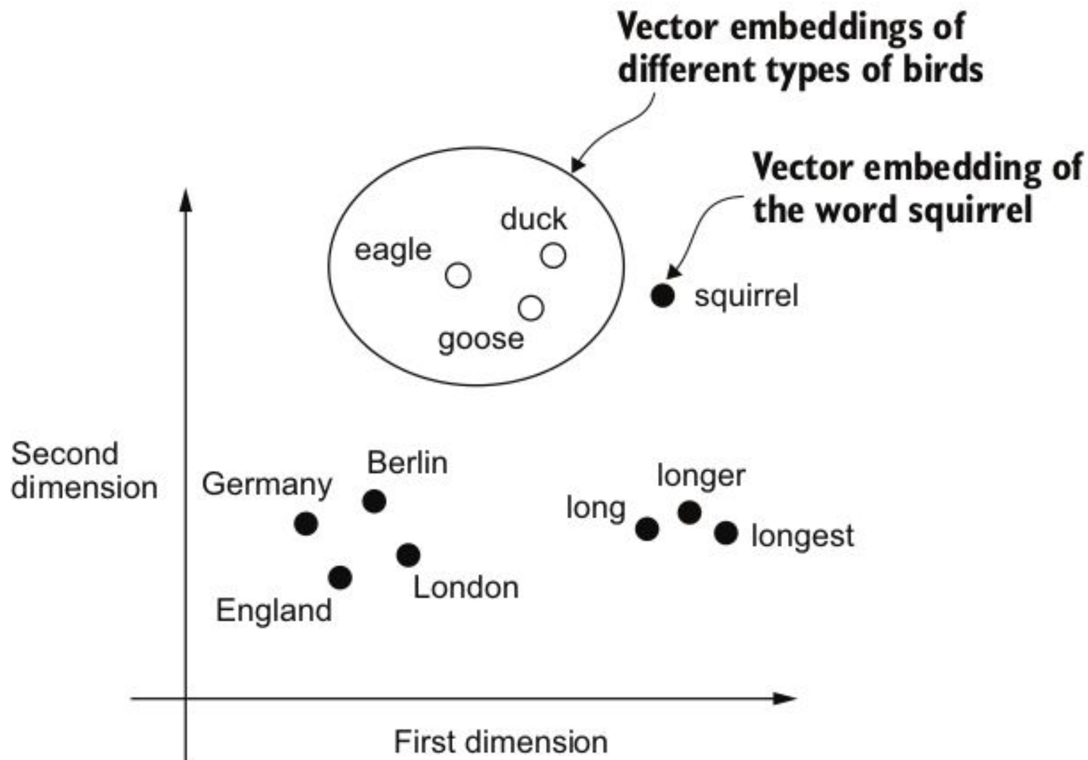
The 2003 (silent) breakthrough

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

A Neural Probabilistic Language Model

Yoshua Bengio BENGIOY@IRO.UMONTREAL.CA
Réjean Ducharme DUCHARME@IRO.UMONTREAL.CA
Pascal Vincent VINCENTP@IRO.UMONTREAL.CA
Christian Jauvin JAUVINC@IRO.UMONTREAL.CA
Département d'Informatique et Recherche Opérationnelle
Centre de Recherche Mathématiques
Université de Montréal, Montréal, Québec, Canada

Key idea: Embeddings



nn.Embedding

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

```
n_token = 3
n_embed = 4

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

```
Weights of the embedding:
 Parameter containing:
tensor([[-0.9252,  0.8805, -0.0214,  0.9724],
        [ 0.1136,  0.2035,  1.1415,  0.0875],
        [ 0.4177,  0.6348,  0.6271,  0.1938]], requires_grad=True)
Result of embedding token number 1:
 tensor([[0.1136, 0.2035, 1.1415, 0.0875]], grad_fn=<EmbeddingBackward0>)
```

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

What is the difference between nn.embedding and nn.linear?

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

`nn.Embedding`

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

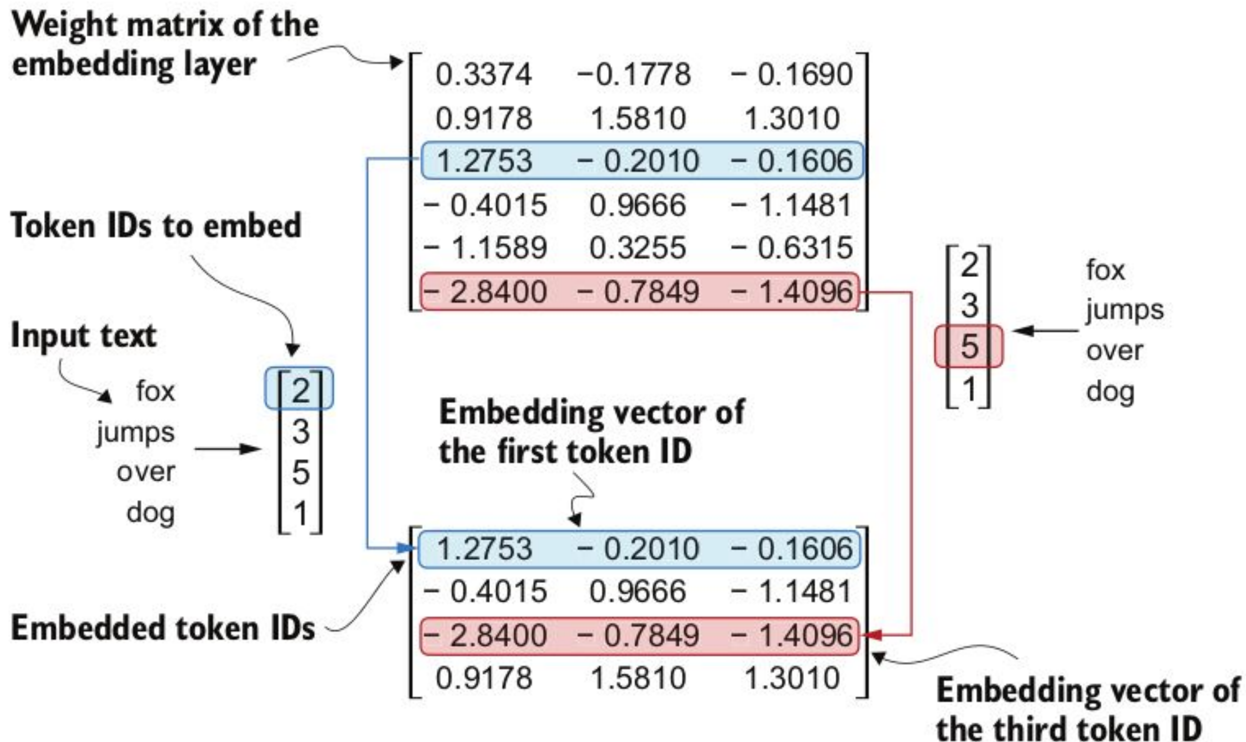
`nn.Linear`

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

In Summary:

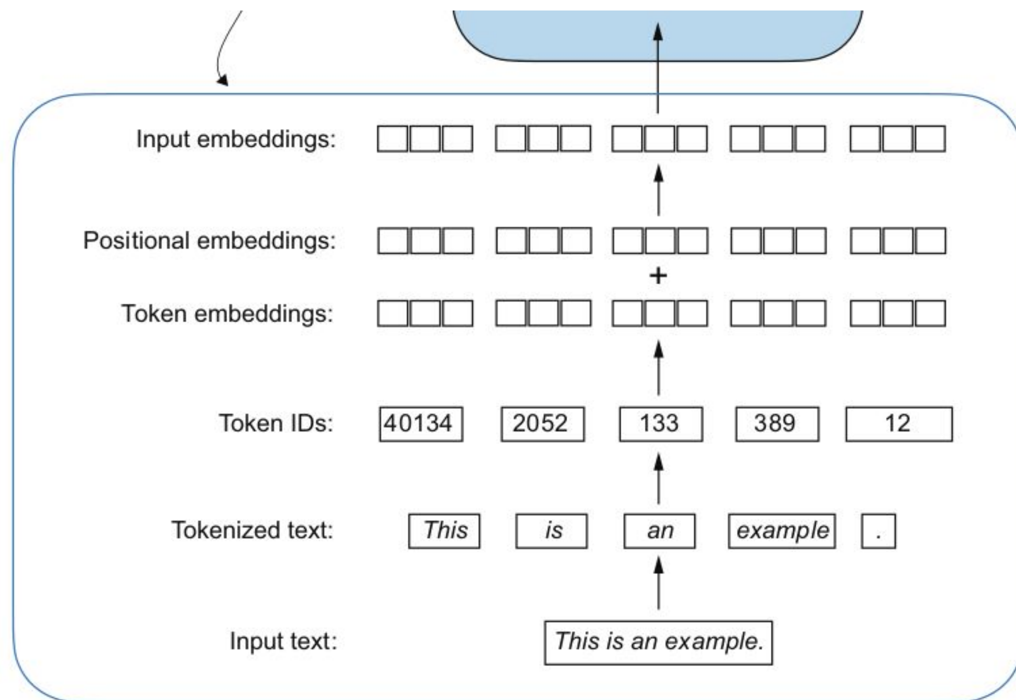
- Use `nn.Embedding` for representing categorical data as dense vectors.
- Use `nn.Linear` for performing linear transformations in neural networks.

From text to vectors



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

(We will discuss later *positional embeddings*.)



Statistics

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

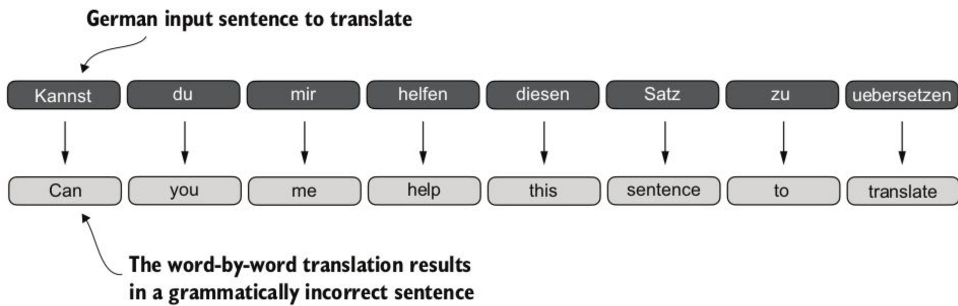
The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

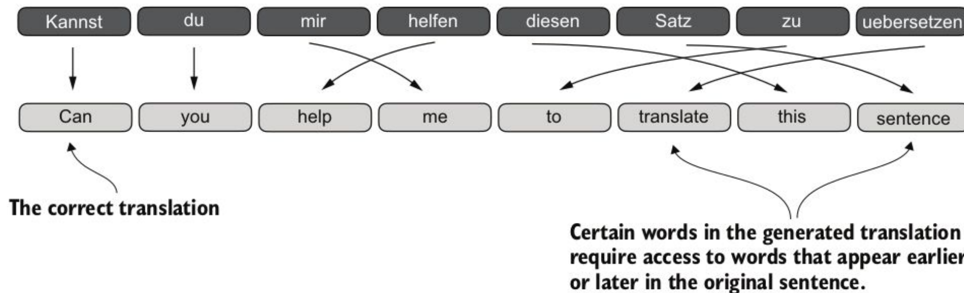
Multi-layer Perceptron (MLP)

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

Two issues with MLPs

- Long contexts require huge amount of compute
- Struggle with long-range dependencies





The attention mechanism

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

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

Łukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Attention is all you need

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

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

A side-note

OpenAI scientist Noam Brown:

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

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

A self-attention head

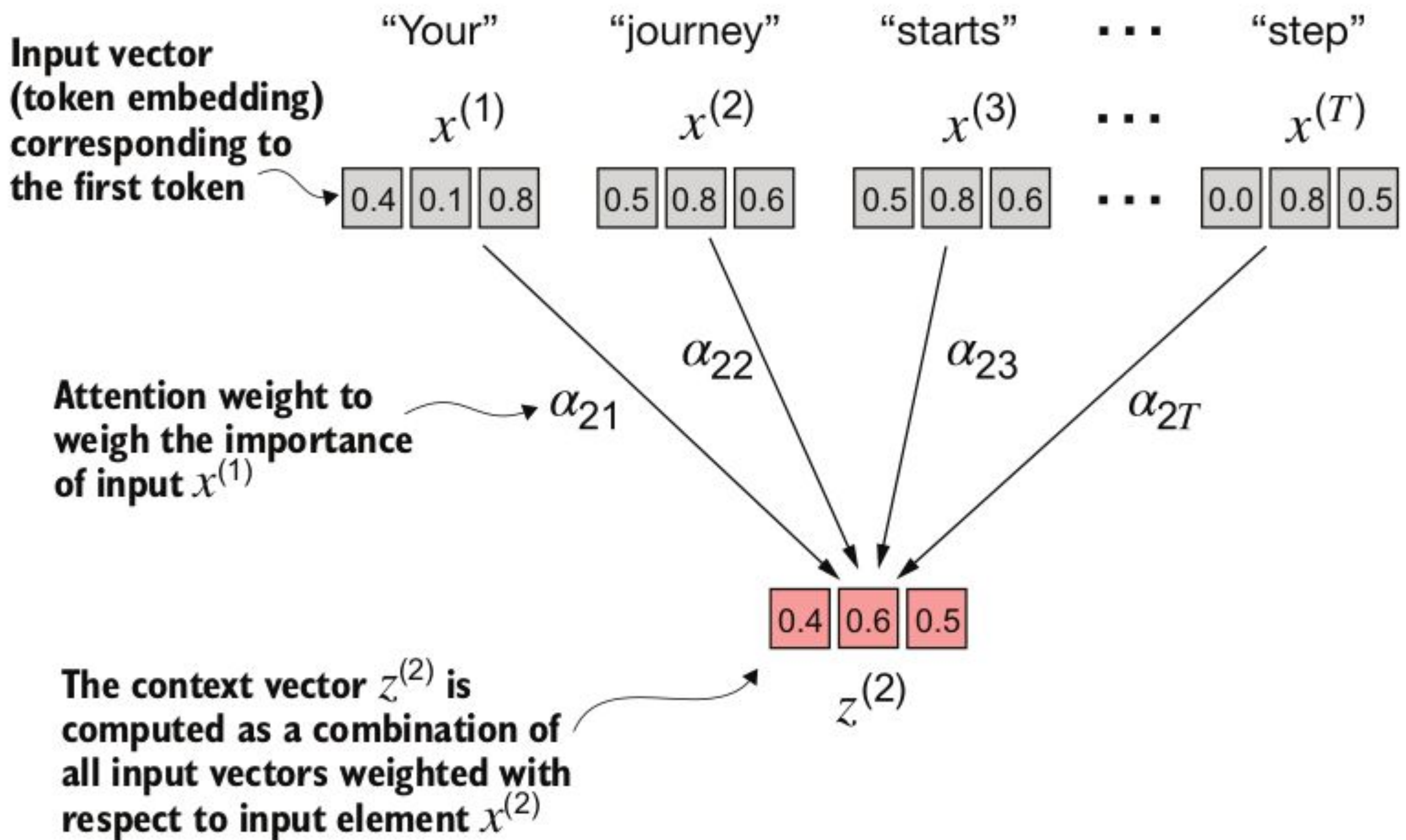
Input: an embedding vector x(i) for each token i

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

Intuition: z(i) gathers *contextual* information

Computing context vectors

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



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

Just a matrix multiplication...

```
context_length = 3
embed_dim = 2

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

context_vectors = attention_weights @ x
```

Computing attention scores and weights

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

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

Keys, queries, and values

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

Output: for each token i:

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

Information-retrieval intuition

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

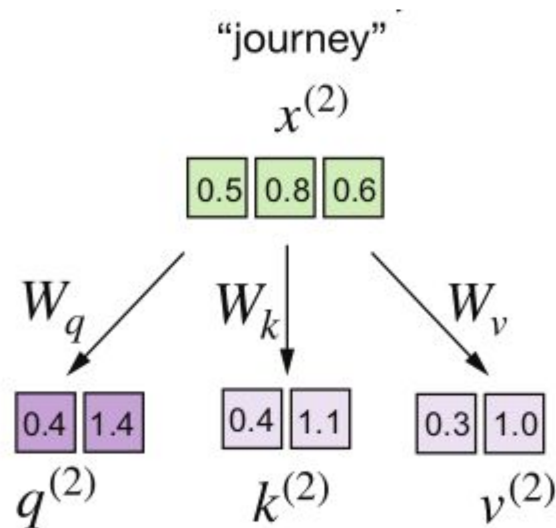
Here, keys, queries, and values are vectors. To match a query with a key we simply do a dot-product.

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

Keys, queries, and values are computed by matrix multiplications

We introduce three matrices with trainable parameters:

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



From attention scores to attention weights

Attention scores are computed by a single matrix multiplication:

```
query @ key.T
```

Now, how do we normalise scores into weights?

Softmax is vector normalisation

```
context_length = 5

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

```
The attention scores:
 tensor([ 1.4529,  0.3491, -0.8928,  0.2072, -0.3993])
After exponentiation:
 tensor([4.2757, 1.4177, 0.4095, 1.2302, 0.6708])
After normalisation:
 tensor([0.5342, 0.1771, 0.0512, 0.1537, 0.0838])

The two steps above are called softmax:
 tensor([0.5342, 0.1771, 0.0512, 0.1537, 0.0838])
```

We have to be careful with softmax

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

A second reason is softmax sensitivity to large numbers, illustrated below:

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

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

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

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

Scaled self-attention

Assume u,v are vectors of dimension d:

    u,v ~ 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

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

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

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

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

As a NN.module

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

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

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

The power of Pytorch broadcasting semantics

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

This is called broadcasting semantics:

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

Complexity of self-attention heads

```
C = context_length

I = input_dim

H = head_dim

O = output_dim

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

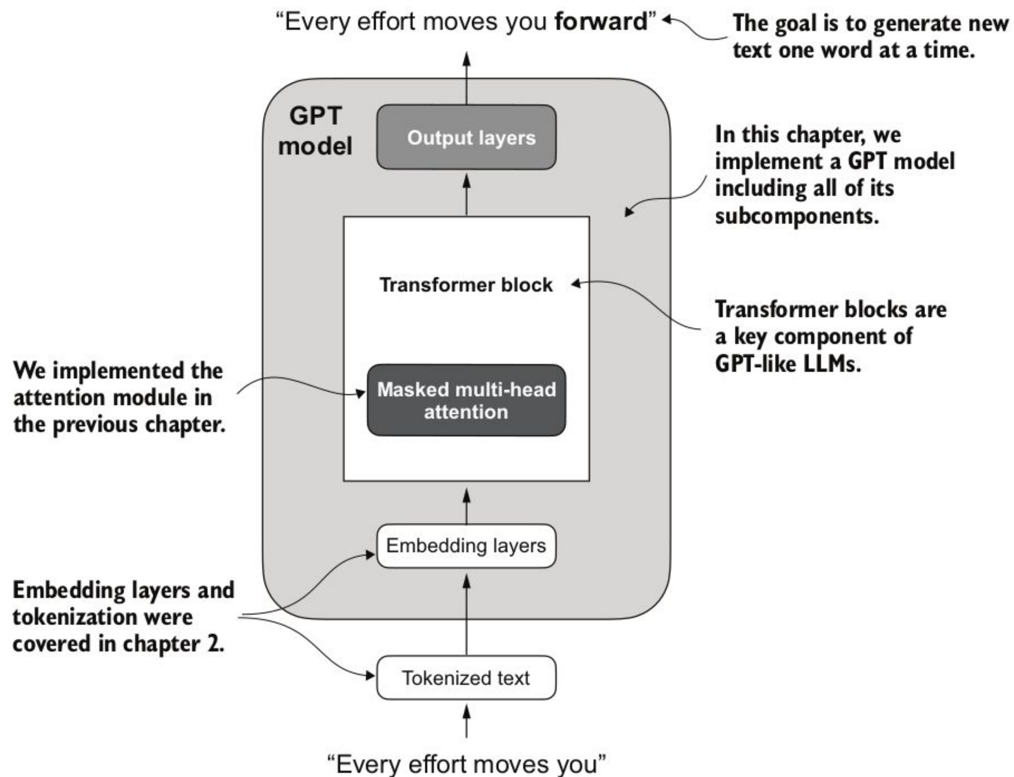
The memory footprint is quadratic in context length!

IMPORTANT

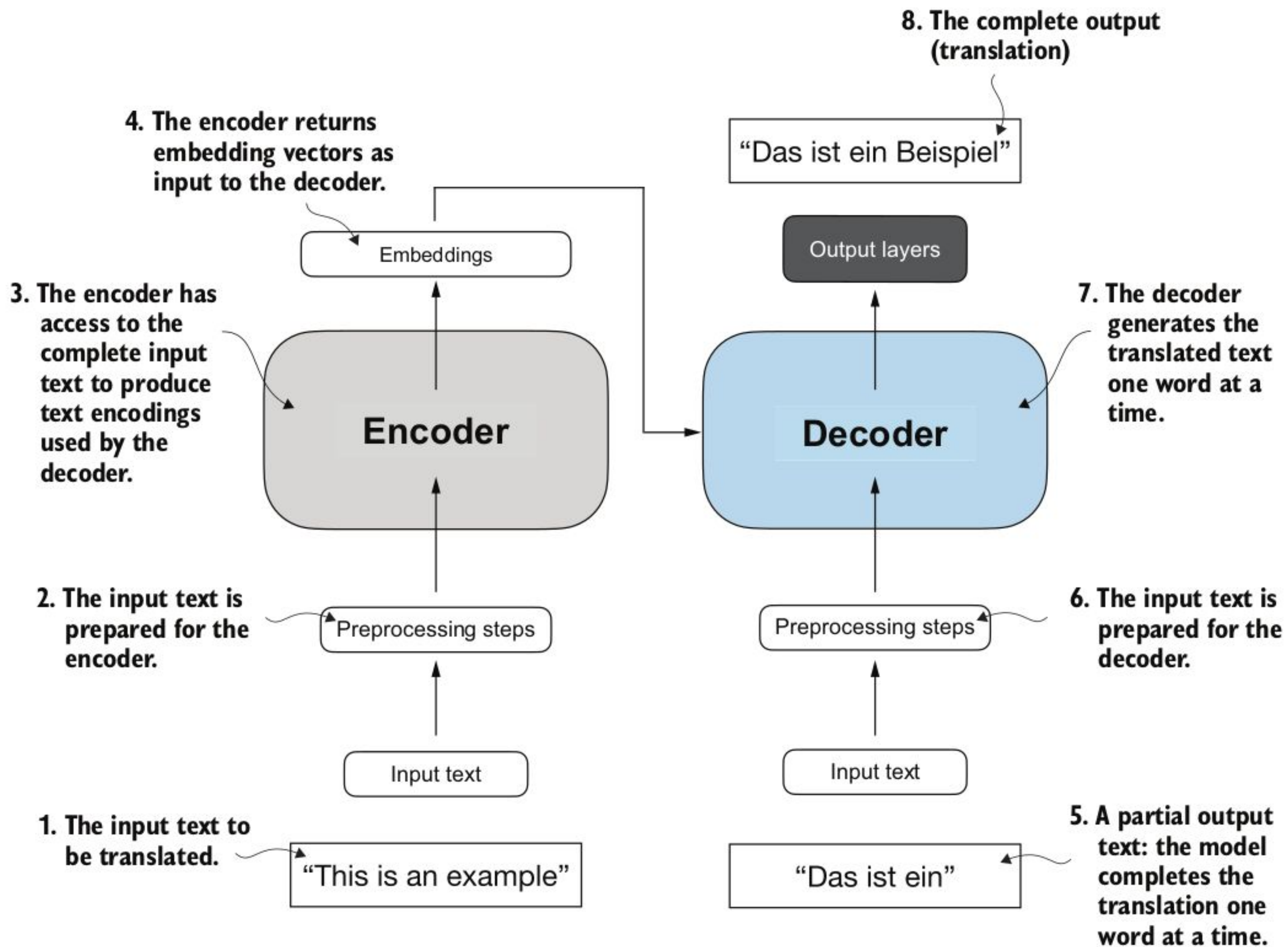
The matrices for computing keys, queries, and values include **trainable** parameters, so the attention mechanism **learns** where to put attention in a **data-driven way.**

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

Attention heads as key components in a Transformer



Encoder / decoder


8. The complete output (translation)
"Das ist ein Beispiel"
4. The encoder returns embedding vectors as input to the decoder.
Embeddings
Output layers
3. The encoder has access to the complete input text to produce text encodings used by the decoder.
Encoder
Decoder
7. The decoder generates the translated text one word at a time.
2. The input text is prepared for the encoder.
Preprocessing steps
Preprocessing steps
6. The input text is prepared for the decoder.
Input text
Input text
1. The input text to be translated.
"This is an example"
"Das ist ein"
5. A partial output text: the model completes the translation one word at a time.

Decoders use causal attention

	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

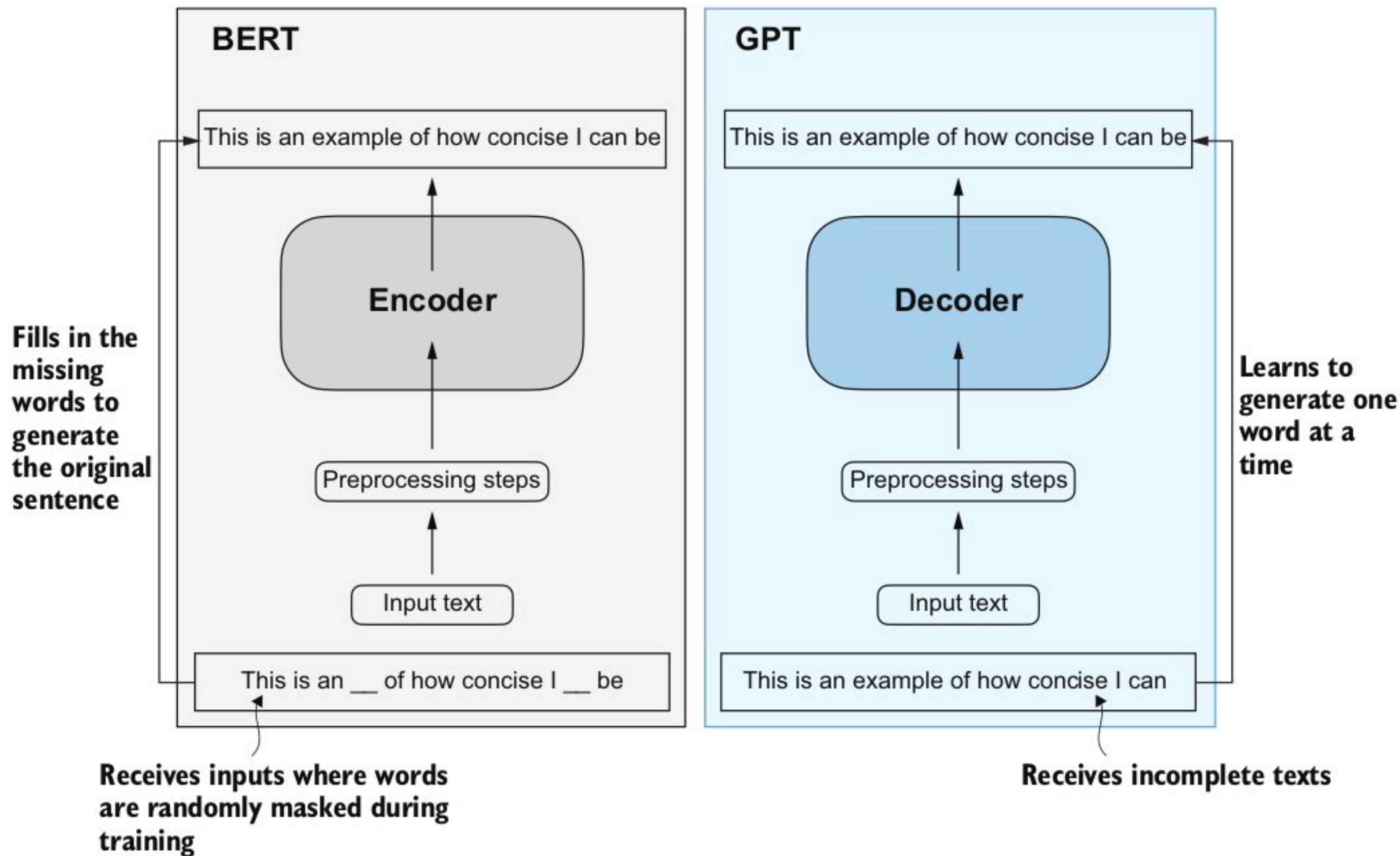
Attention weight for input tokens corresponding to "step" and "Your"

	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

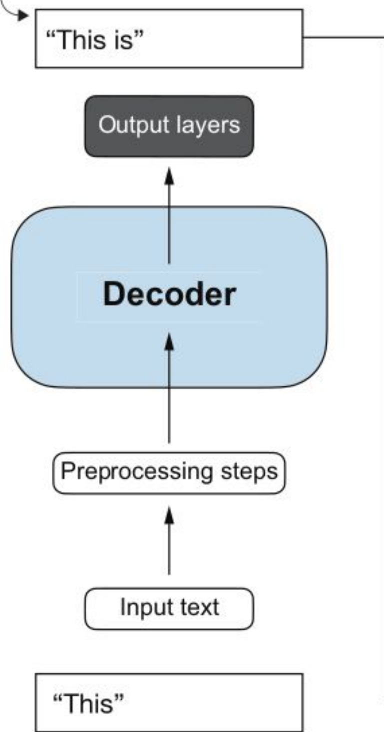
```
x = torch.randn(context_length, input_dim)

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
```

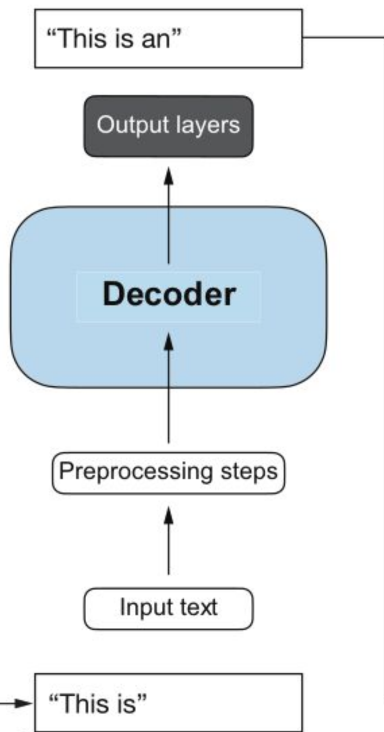

BERT
This is an example of how concise I can be
Encoder
Preprocessing steps
Input text
This is an __ of how concise I __ be
Fills in the missing words to generate the original sentence
Receives inputs where words are randomly masked during training
GPT
This is an example of how concise I can be
Decoder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

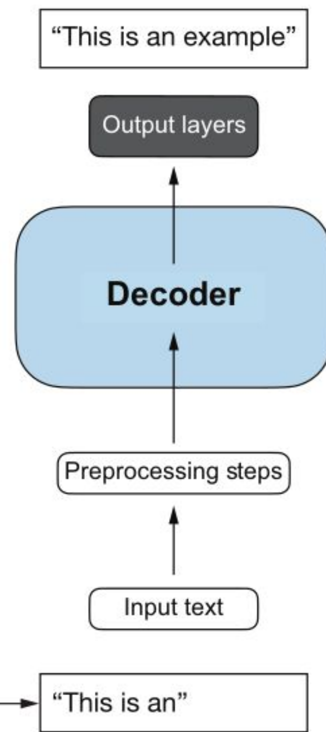
Autoregressive models

AUTOREGRESSIVE

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


Creates the next word based on the input text
Iteration 1
"This is"
Output layers
Decoder
Preprocessing steps
Input text
"This"


Iteration 2
"This is an"
Output layers
Decoder
Preprocessing steps
Input text
"This is"
The output of the previous round serves as input to the next round.


Iteration 3
"This is an example"
Output layers
Decoder
Preprocessing steps
Input text
"This is an"

Sliding windows

Sliding windows

Text sample:

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

The LLM can't access words past the target.

Input the LLM receives

Target to predict

IMPORTANT

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

Input: a sequence of vectors, one for each token

Output: a sequence of vectors, one for each token

What are the benefits of the sliding windows?

Fix c = context_length

A single data point (meaning, a sequence of c+1 tokens) becomes c data points, <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

Batching

Models' signatures (without batching)

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

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

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

For each window, make the prediction and compute the loss

Models' signatures with batching

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

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

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

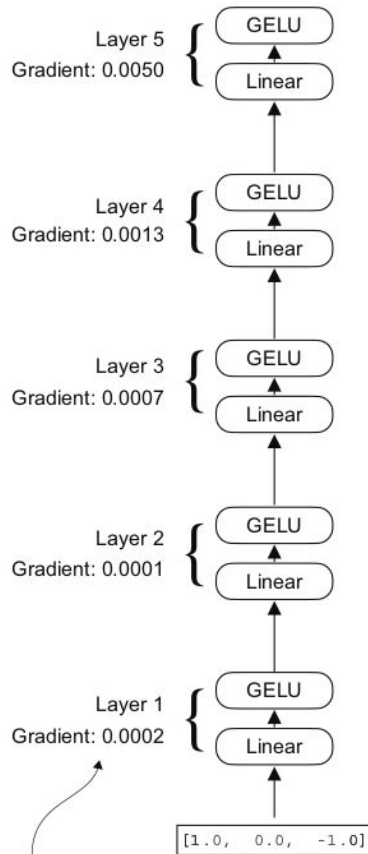
Note: this is called "batch-first", sometimes the models are "input-first" (just a matter of definitions)

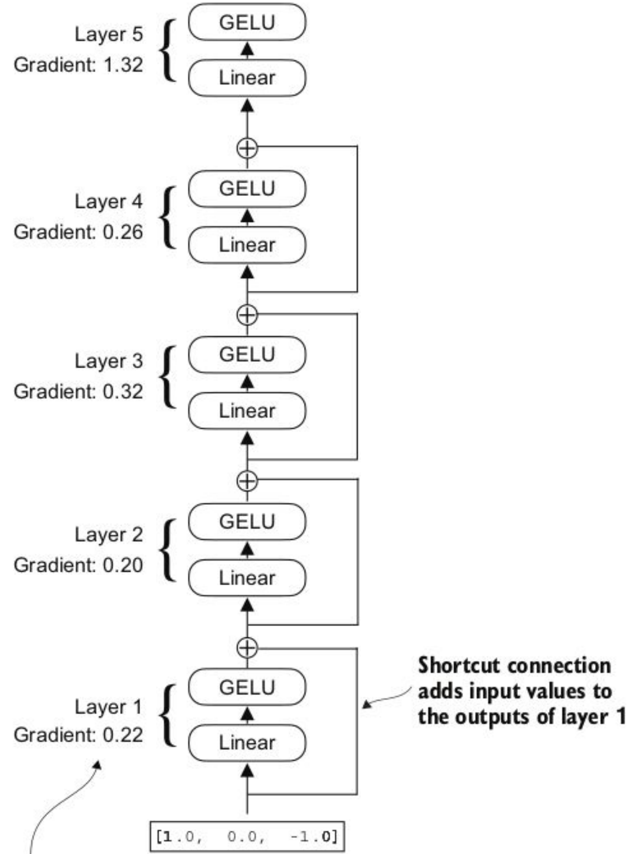
Shortcut connections

SHORTCUT CONNECTIONS

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

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


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


Deep neural network with shortcut connections
Layer 5
Gradient: 1.32
GELU
Linear
Layer 4
Gradient: 0.26
GELU
Linear
Layer 3
Gradient: 0.32
GELU
Linear
Layer 2
Gradient: 0.20
GELU
Linear
Layer 1
Gradient: 0.22
GELU
Linear
[1.0, 0.0, -1.0]
Shortcut connection adds input values to the outputs of layer 1
The shortcut connections help with maintaining relatively large gradient values even in early layers

Dropout

DROPOUT

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

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

DROPOUT

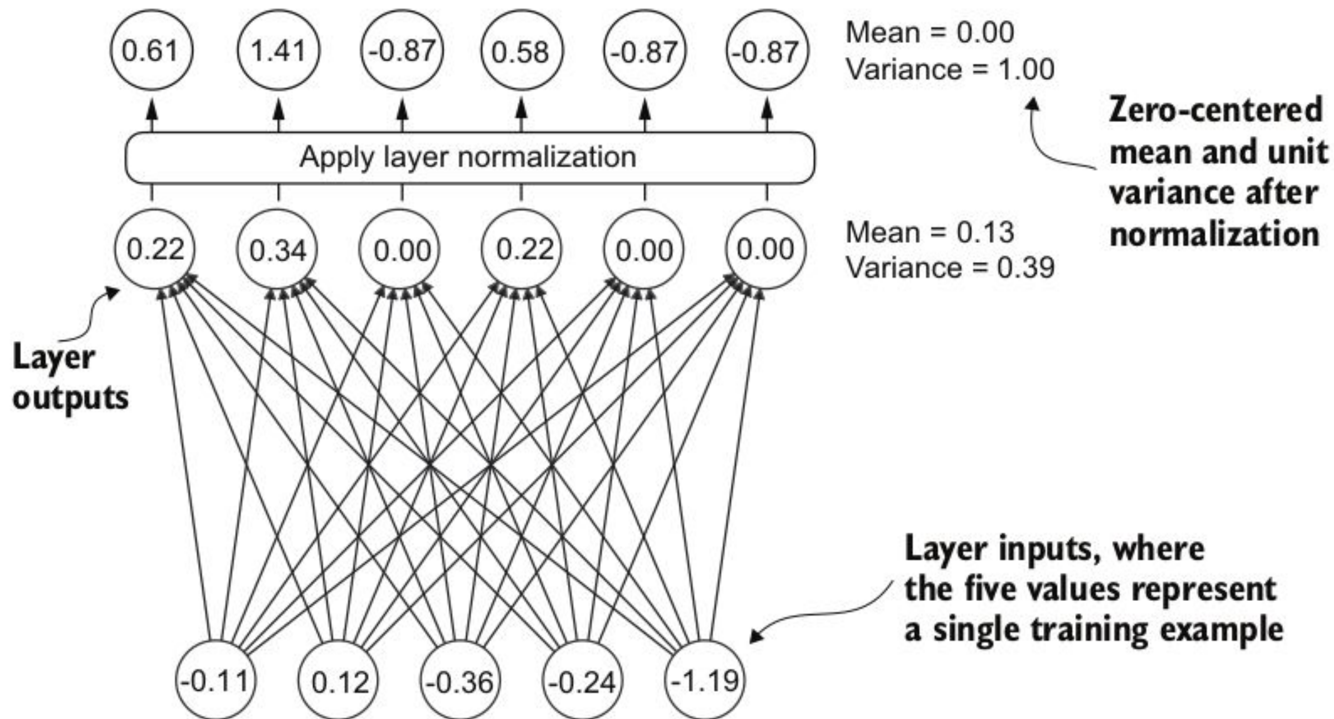
Dropout is only used during training (using *model.train*), it must be deactivated for inference, using *model.eval*

Layer normalization

Why renormalization?

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

Layer normalization



Positional embeddings

Important

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

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

Positional embeddings

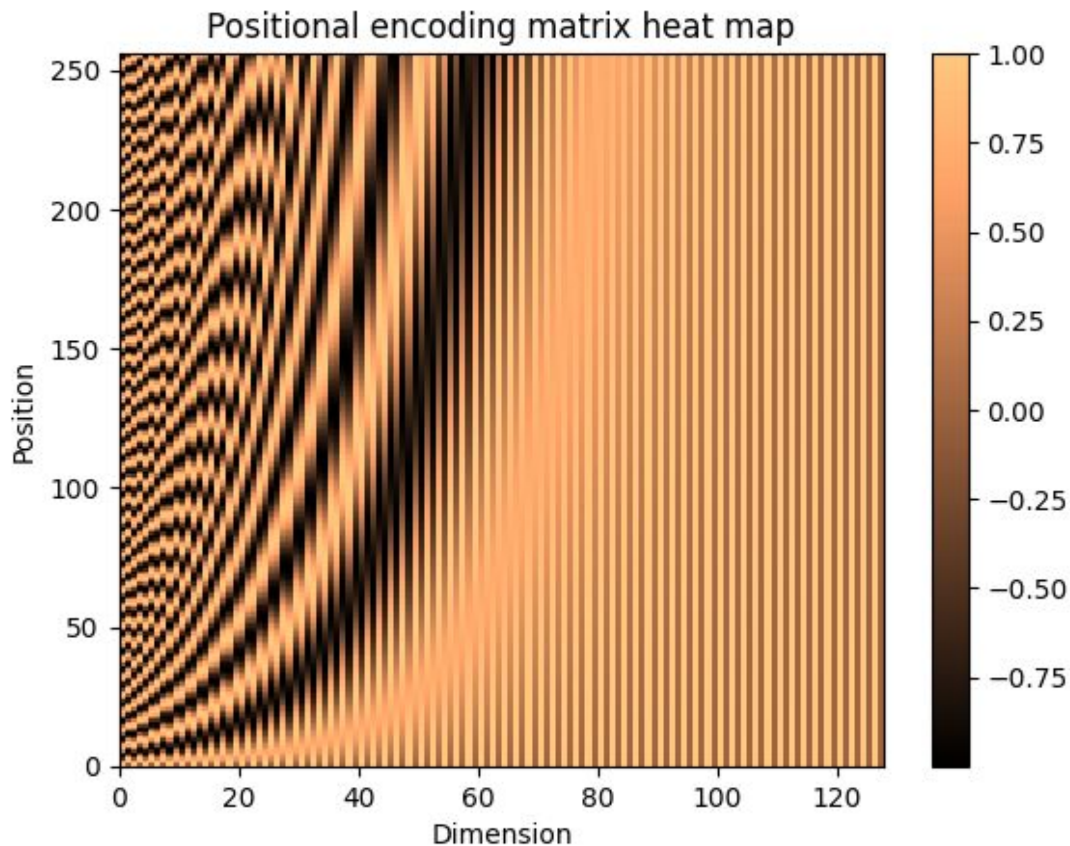
Positional embeddings are added to bring information about position of the tokens.

Simplest version: learned positional embeddings

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

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

The original positional embedding



The formula

Fix c = context_length and d = input_dim

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

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

Remarks:

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

The more recent RoPE (Rotary Positional Embeddings)

An important difference:

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

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

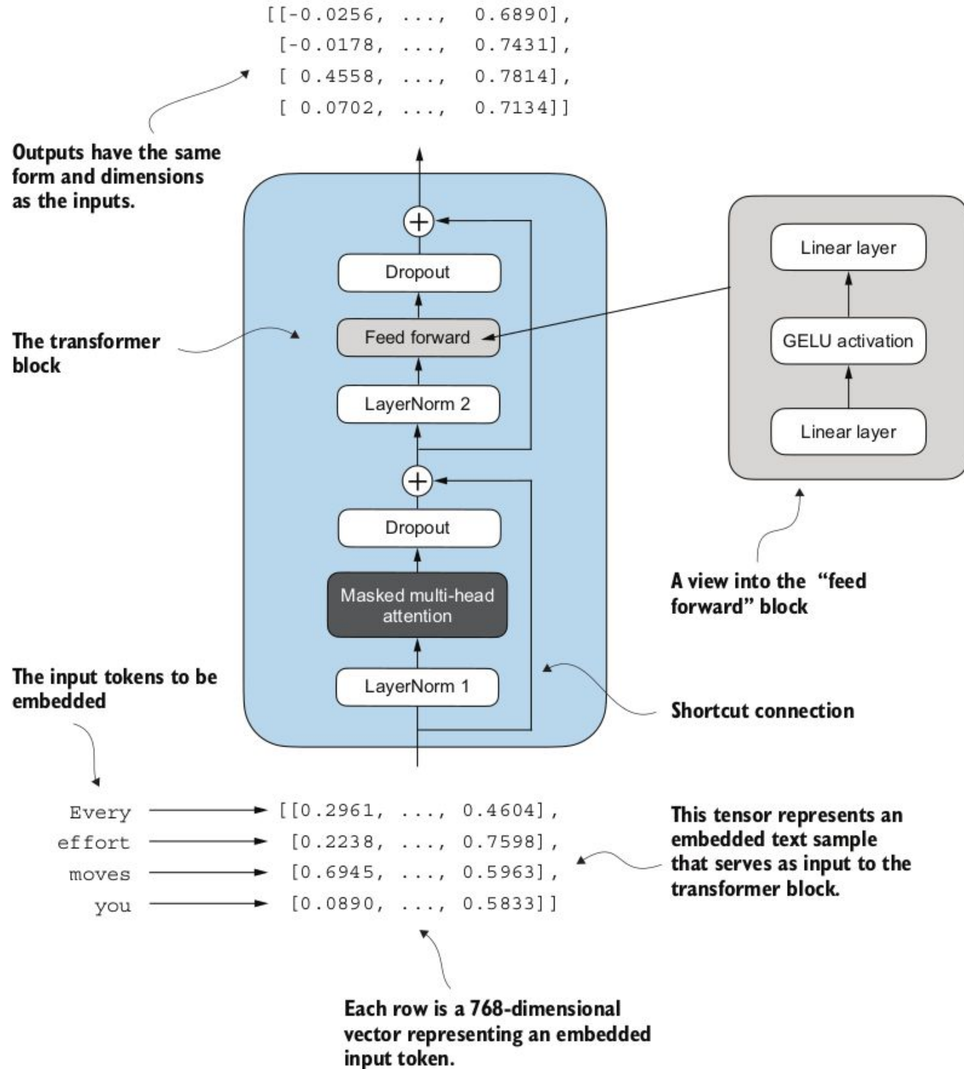
The formula

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

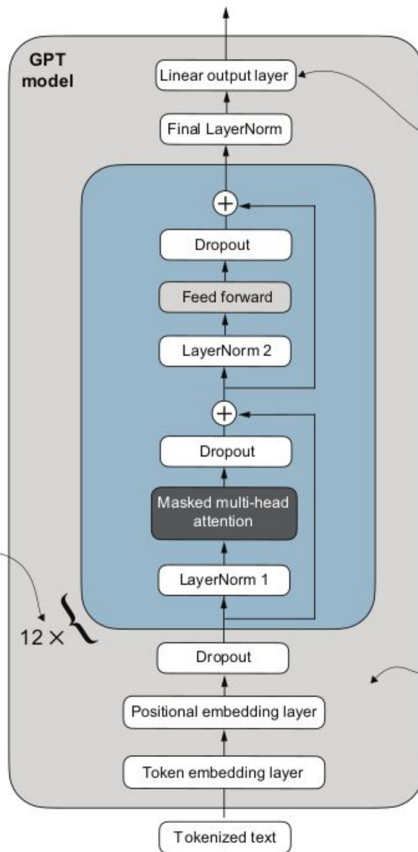
We group dimensions by pairs, and for pair (i,i+1) we apply a rotation of angle pos * theta_i where theta_i = 10_000^{-2(i-1)/d}:

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

Transformer architecture



[[-0.0256, ..., 0.6890],
[-0.0178, ..., 0.7431],
[0.4558, ..., 0.7814],
[0.0702, ..., 0.7134]]
Outputs have the same form and dimensions as the inputs.
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.



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

Pretraining

Boilerplate training code

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

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

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

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

What is cross entropy loss?

Cross entropy measures the difference between probability distributions: it quantifies the dissimilarity between the predicted probability distribution and the true probability distribution.

In language modelling we do not have the true distribution of words, it is approximated from a training set:

$$H(T, q) = -\sum_{i=1}^{N} \frac{1}{N} \log_2 q(x_i)$$

Where N is the number of tokens in the training set and q(x_i) is the probability that the model outputs x_i.

Cross entropy loss

```
vocab_size = 5

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

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

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

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

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

Why is cross entropy loss interesting?

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