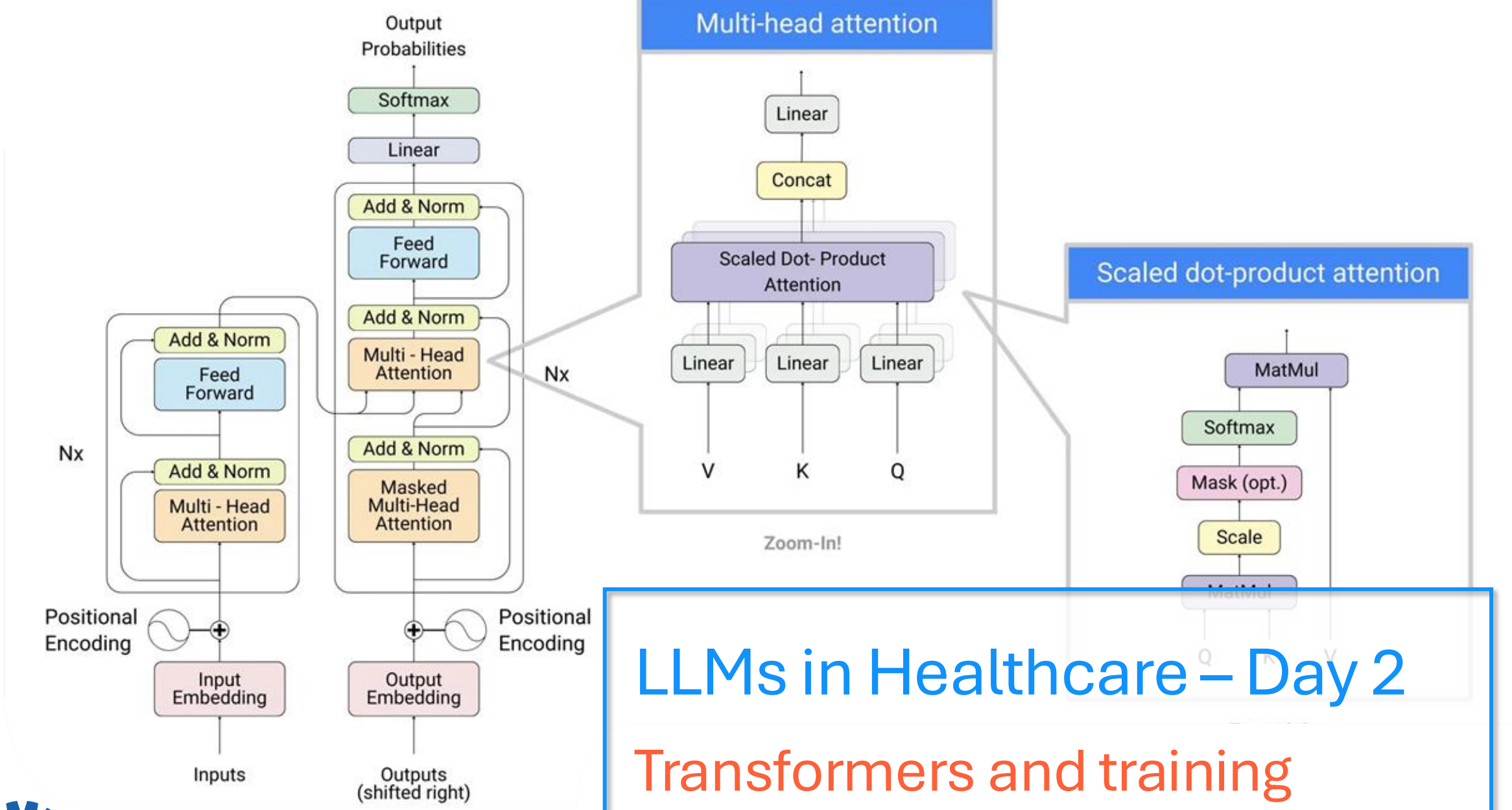


LLMs in Healthcare – Day 2

Transformers and training

Ruurd Kuiper – Julius Center, UMCU

UMC Utrecht

Image source: datasciencedojo.com

News update

The Turing test: refresh or retire?

Passing the Turing test — a thought experiment that asks whether a computer's written conversation can pass as a person's — is a piece of cake for the best artificial intelligence (AI) models. First proposed by mathematician Alan Turing 75 years ago, some researchers want to see an upgraded test to use as a benchmark for progress towards artificial general intelligence — models with the resourcefulness to match human cognitive ability. Others say the test should be scrapped altogether, and that developers should instead focus on evaluating AI safety and building specific capabilities that could be of benefit to the public.

Nature | 6 min read

Contents day 2

1. Presentations from day 1
2. Quick recap: What do LLMs do?
3. How do LLMs work?

Short break

4. Transformer architecture essentials
5. Tutorial: Building and training an LLM

Lunch

6. Practical: Build and train your own LLM

Why do we need to know how these models work?

Today we get an in-depth understanding of the inner workings of large language models, and more broadly speaking foundation models in general.

We need to know how these models work, to know what their capabilities and limitations are. But maybe even more important, we need to understand what their RISKS are.

What are LLMs?

A **large language model** (LLM) is a **deep learning neural network** trained on **large (text) datasets** to learn **statistical patterns of language** enabling it to **generate text** by **predicting** the **mostly likely** next **token** in a sequence.

They can exhibit emergent capabilities such as contextual **understanding (?)** and **reasoning (?)**, allowing them to perform diverse tasks without task-specific training

What's so special about transformers?

The transformer is the main component of almost all currently popular and available large language models.

This was not always the case, as before we had:

- CNNs for image data
- RNNs, and LSTMs for sequence data

However, these did not capture the long range dependencies well enough.

So, what's different about the transformer architecture? What causes this model to be so much more powerful?

What do LLMs do?

LLMs estimate the conditional probability distribution $P(w_{t+1}|w_1, w_2, \ldots, w_t)$, where $w_1, w_2, \ldots, w_t$ are the preceding words in a sequence, and w_{t+1} represents the next word to be predicted.

What do LLMs do?

LLMs estimate the conditional probability distribution $P(w_{t+1}|w_1, w_2, \ldots, w_t)$, where $w_1, w_2, \ldots, w_t$ are the preceding words in a sequence, and w_{t+1} represents the next word to be predicted.

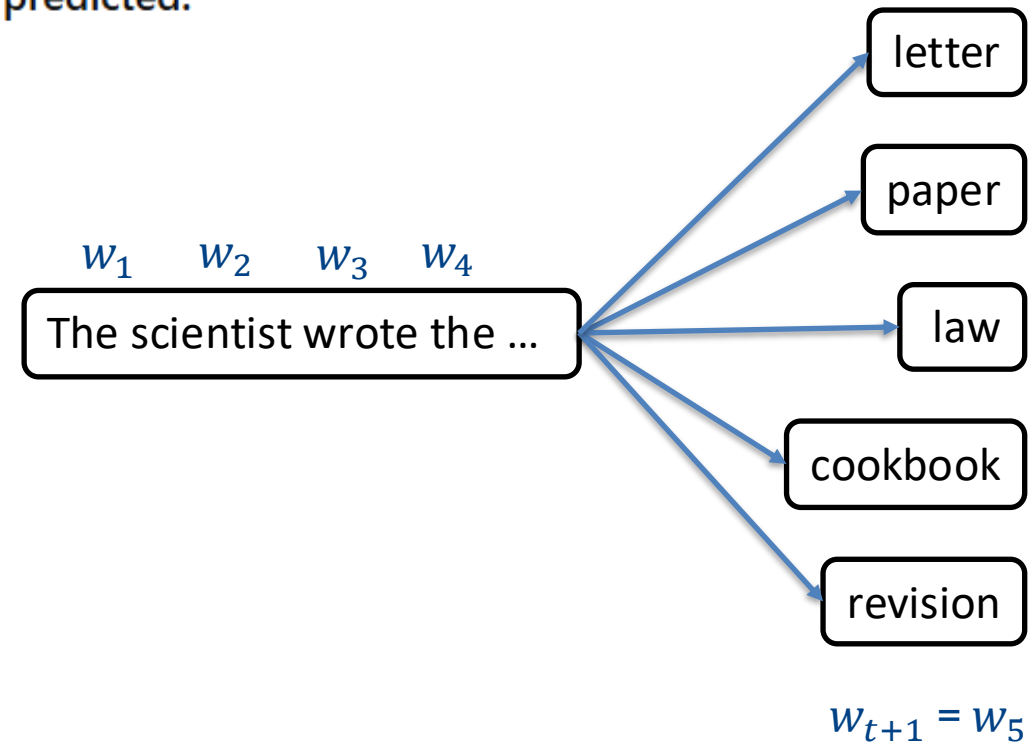
w_1 w_2 w_3 w_4 $w_{t+1} = w_5$

The scientist wrote the …

?

What do LLMs do?

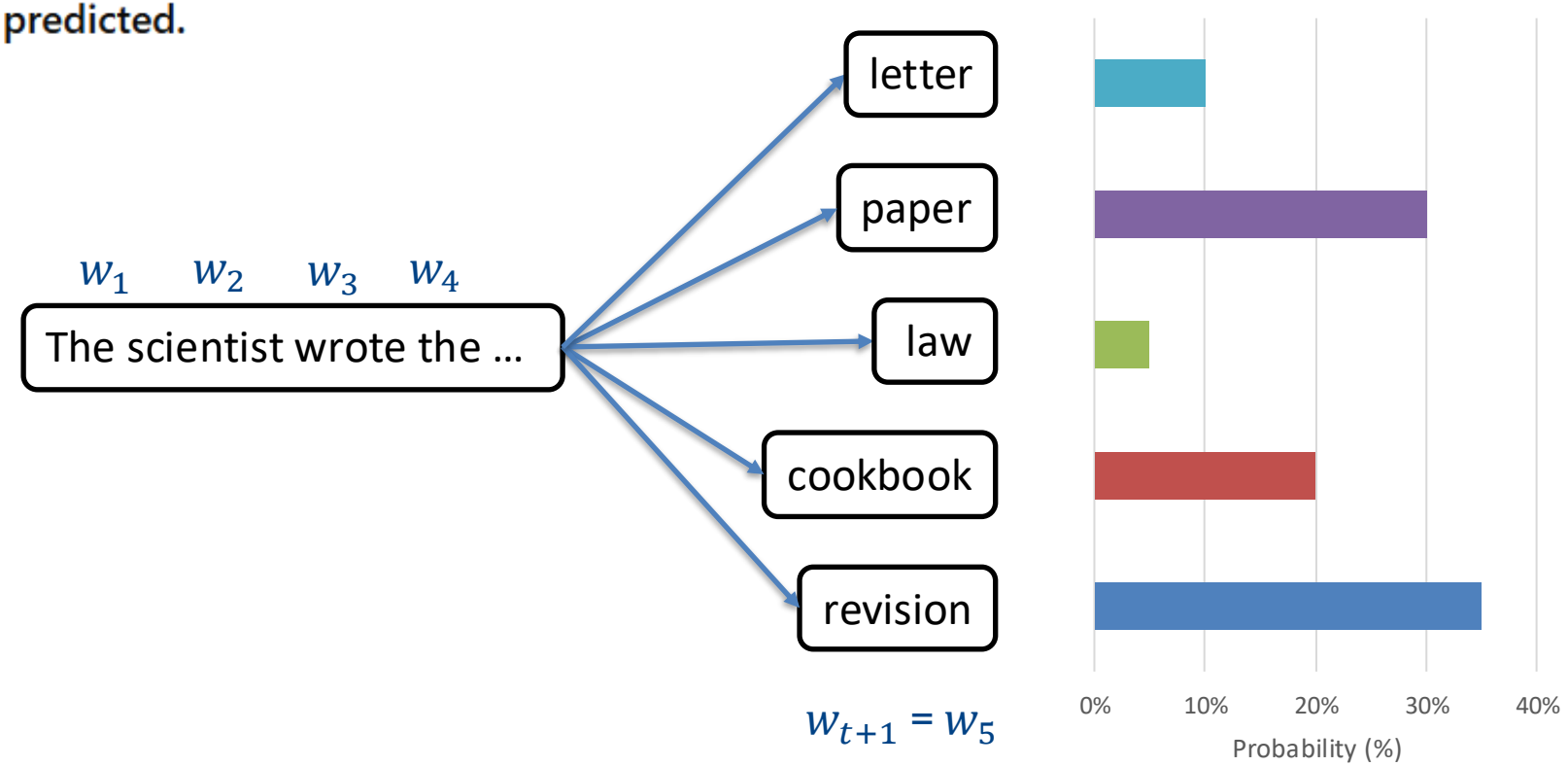
LLMs estimate the conditional probability distribution $P(w_{t+1}|w_1, w_2, \ldots, w_t)$, where $w_1, w_2, \ldots, w_t$ are the preceding words in a sequence, and w_{t+1} represents the next word to be predicted.



$w_{t+1} = w_5$

What do LLMs do?

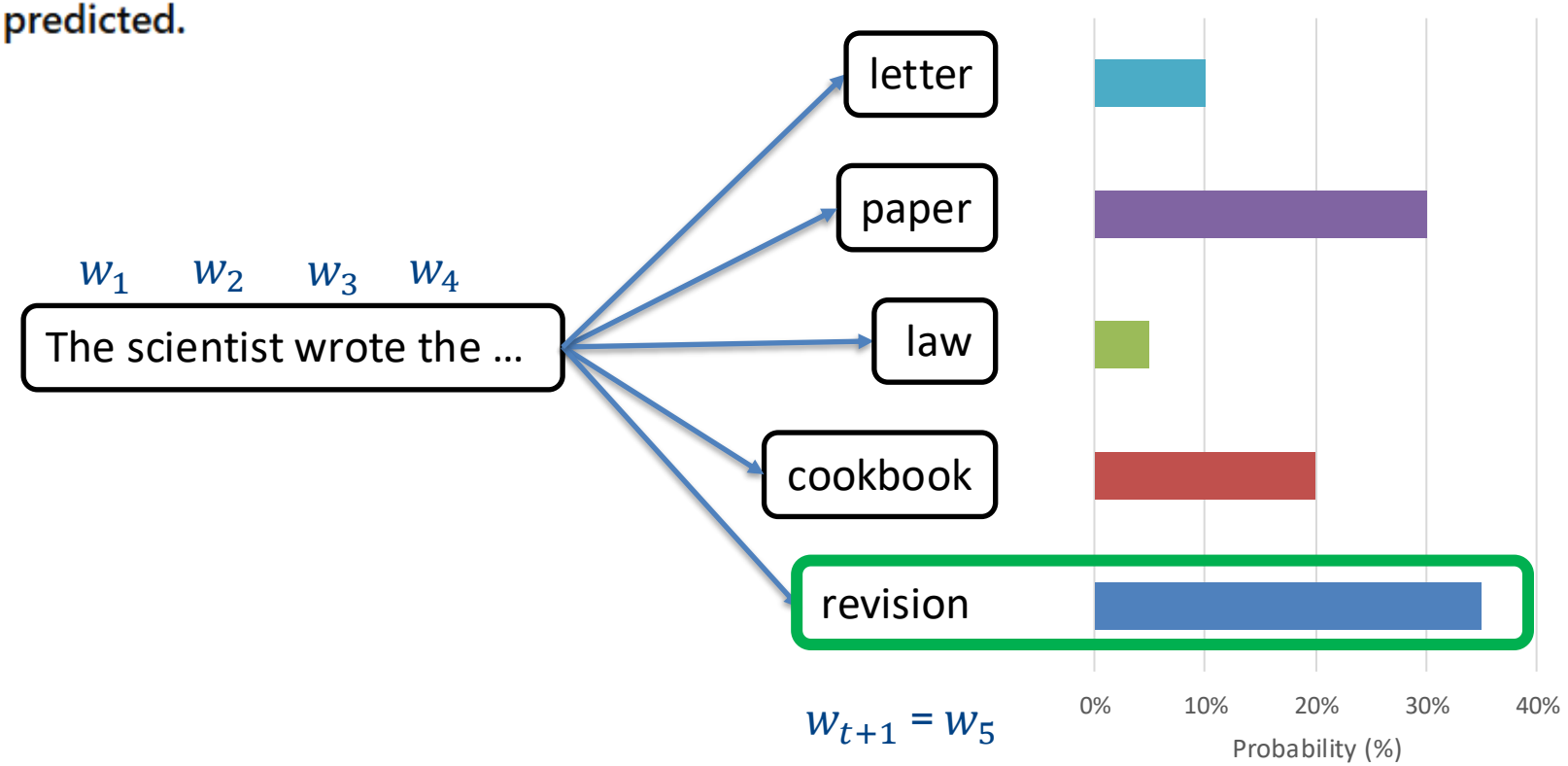
LLMs estimate the conditional probability distribution $P(w_{t+1}|w_1, w_2, \ldots, w_t)$, where $w_1, w_2, \ldots, w_t$ are the preceding words in a sequence, and w_{t+1} represents the next word to be predicted.



$P(w_{t+1}|w_1, w_2, \ldots, w_t)$

What do LLMs do?

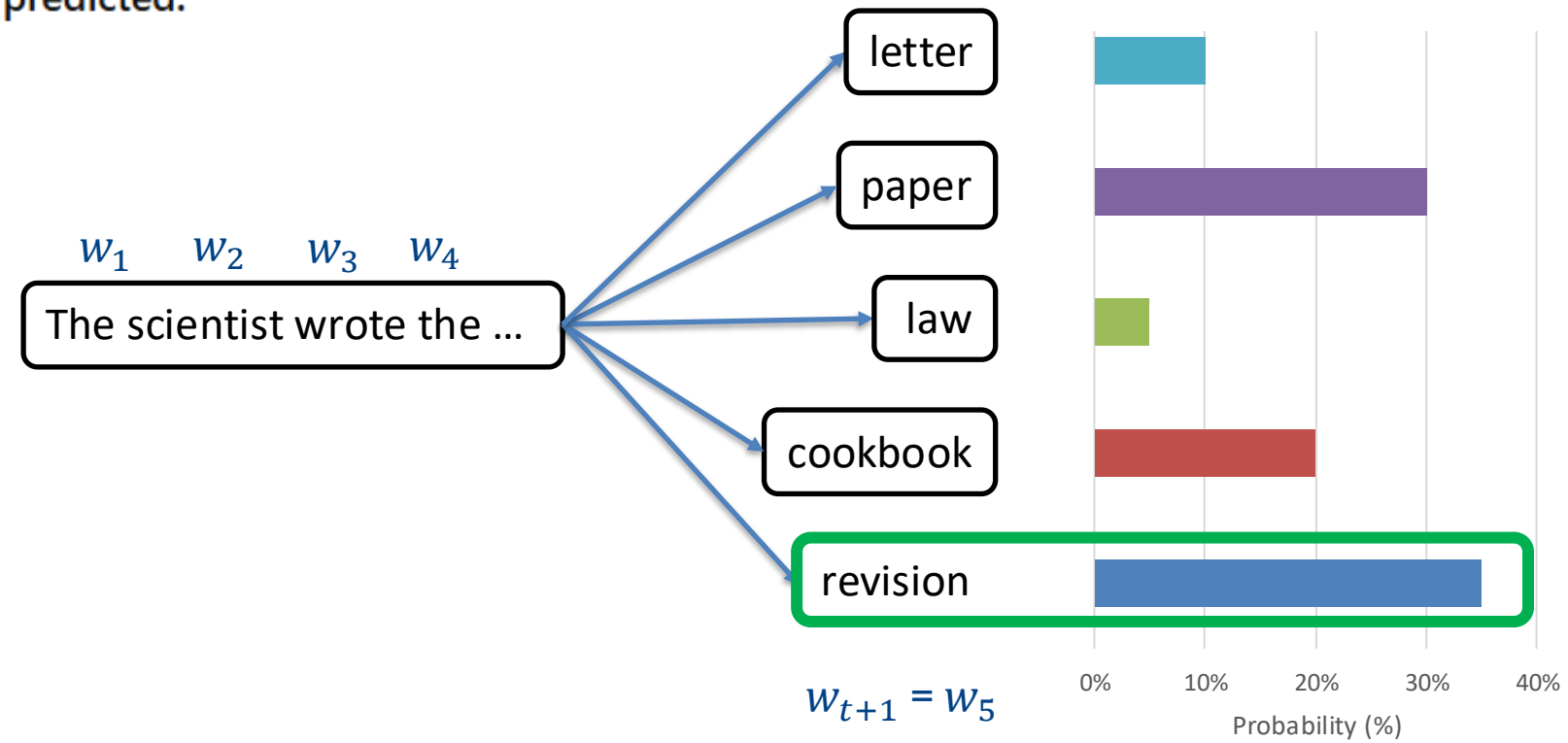
LLMs estimate the conditional probability distribution $P(w_{t+1}|w_1, w_2, \ldots, w_t)$, where $w_1, w_2, \ldots, w_t$ are the preceding words in a sequence, and w_{t+1} represents the next word to be predicted.



$P(w_{t+1}|w_1, w_2, \ldots, w_t)$

What do LLMs do?

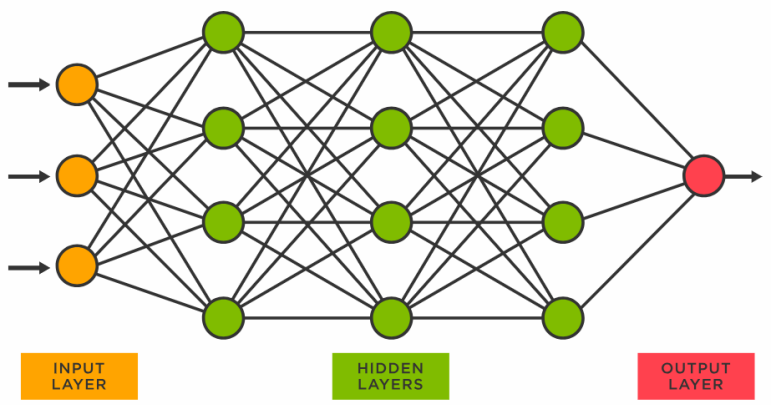
LLMs estimate the conditional probability distribution $P(w_{t+1}|w_1, w_2, \ldots, w_t)$, where $w_1, w_2, \ldots, w_t$ are the preceding words in a sequence, and w_{t+1} represents the next word to be predicted.



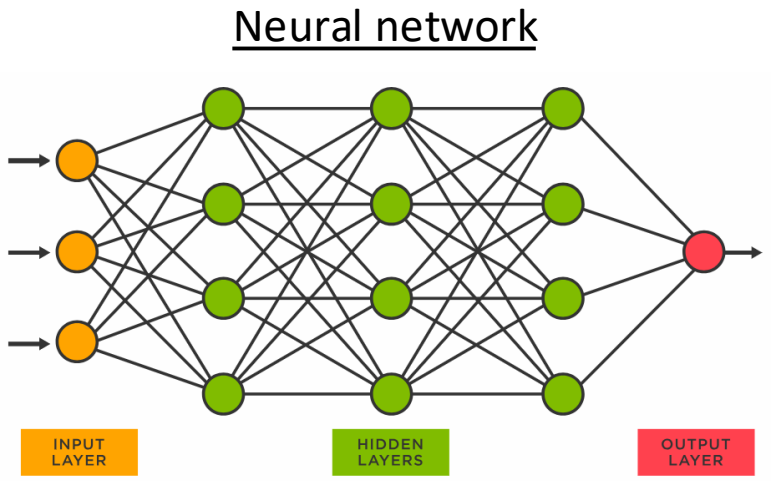
$w_{t+1} = w_5$

How do LLMs work?

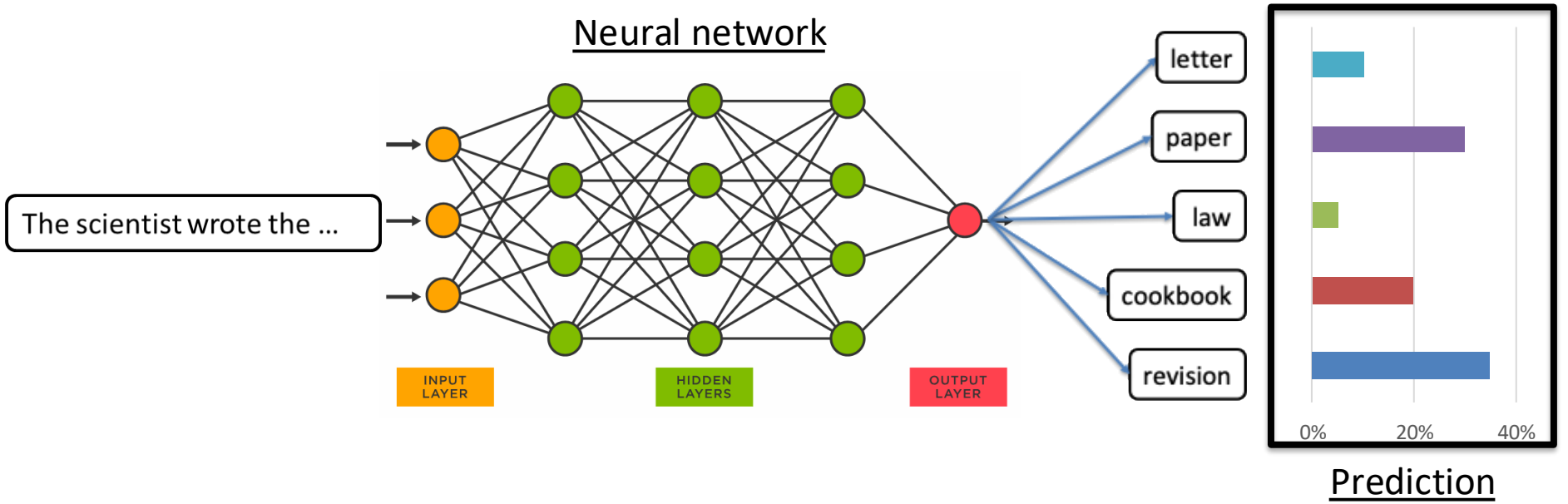
Neural network



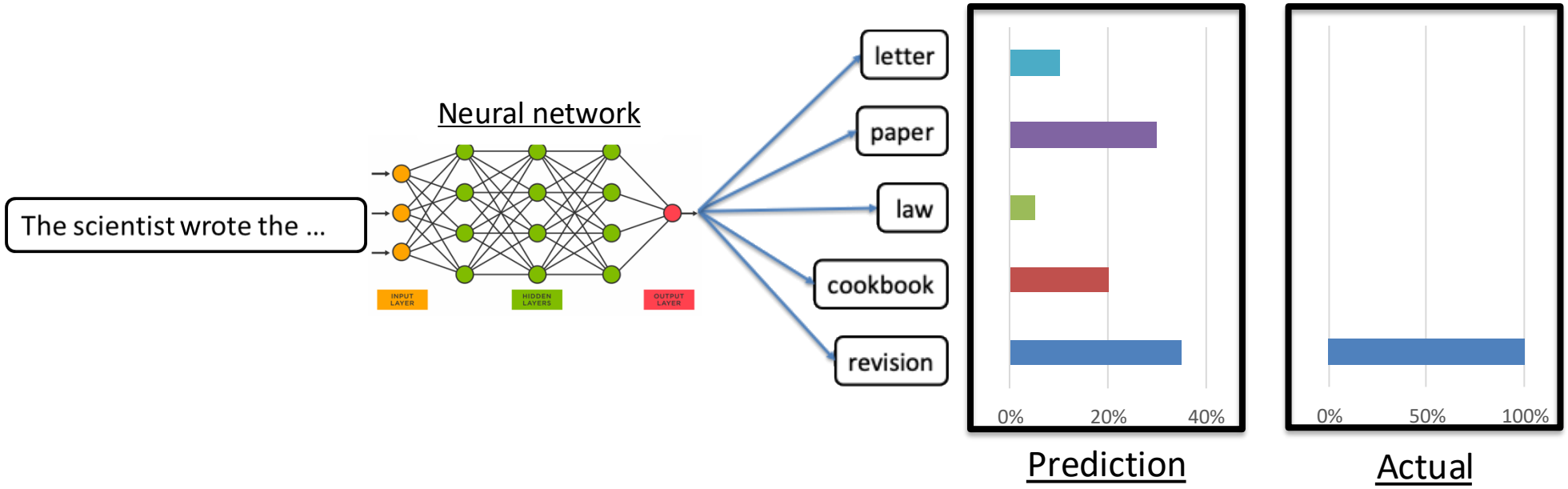
How do LLMs work? – Neural networks



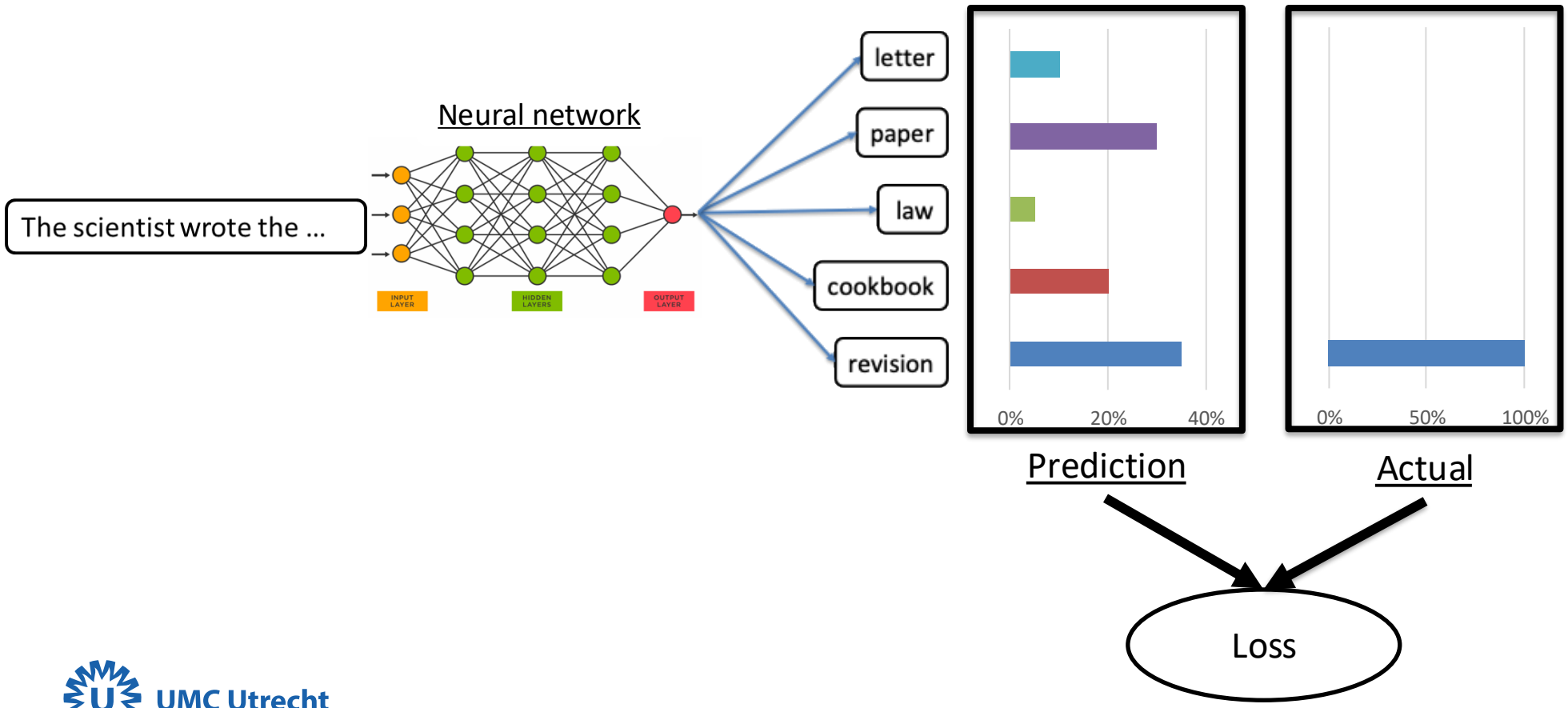
How do LLMs work? – Prediction



How do LLMs work? - Loss



How do LLMs work? - Loss



How do LLMs work? – Cross-entropy loss

Measures how well **predicted** predicted distribution matches **true** distribution.

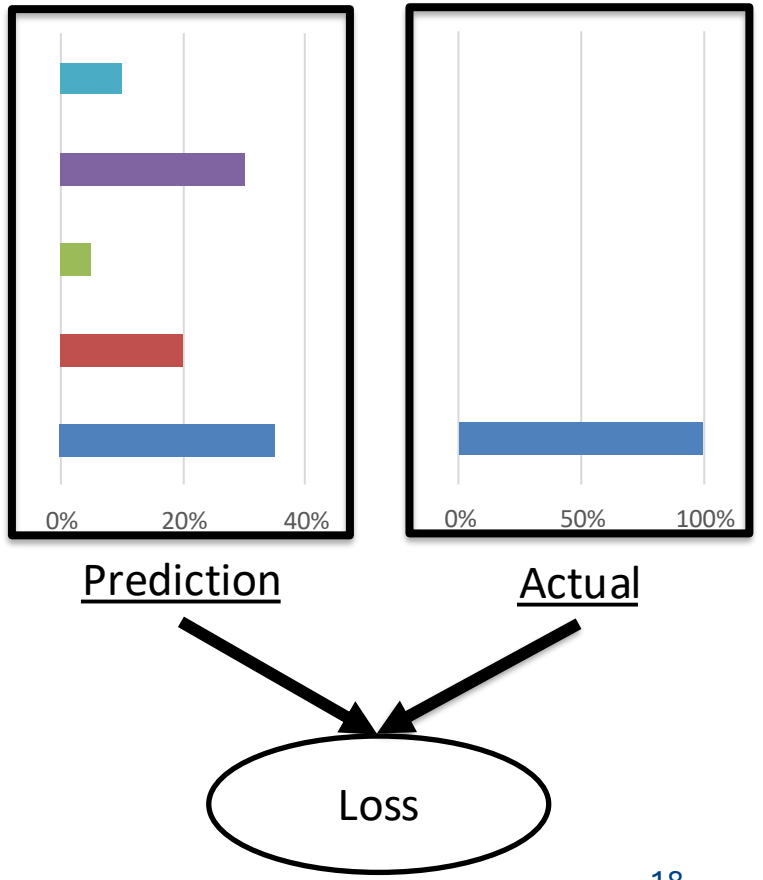
- Predicted distribution = soft-maxed output logits.
- True distribution = one-hot vector with actual token.

$$\mathcal{L} = -\sum_{i=1}^{V} y_i \log(p_i)$$

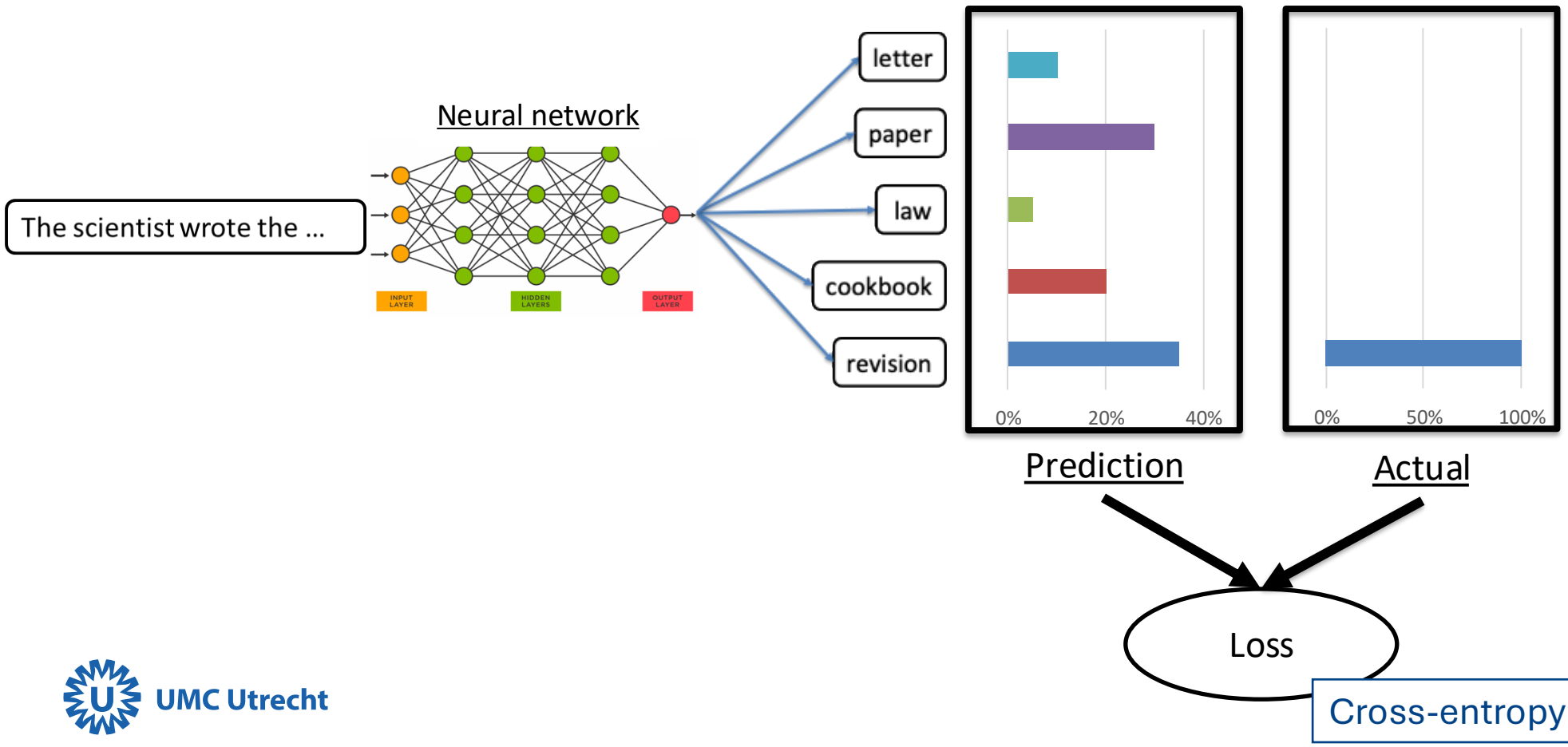
Where:
V= vocabulary size
y_i= 'true probability' for token i
p_i= predicted probability for token i

But since y_i = 100% = 1 for the actual token and 0 everywhere else:

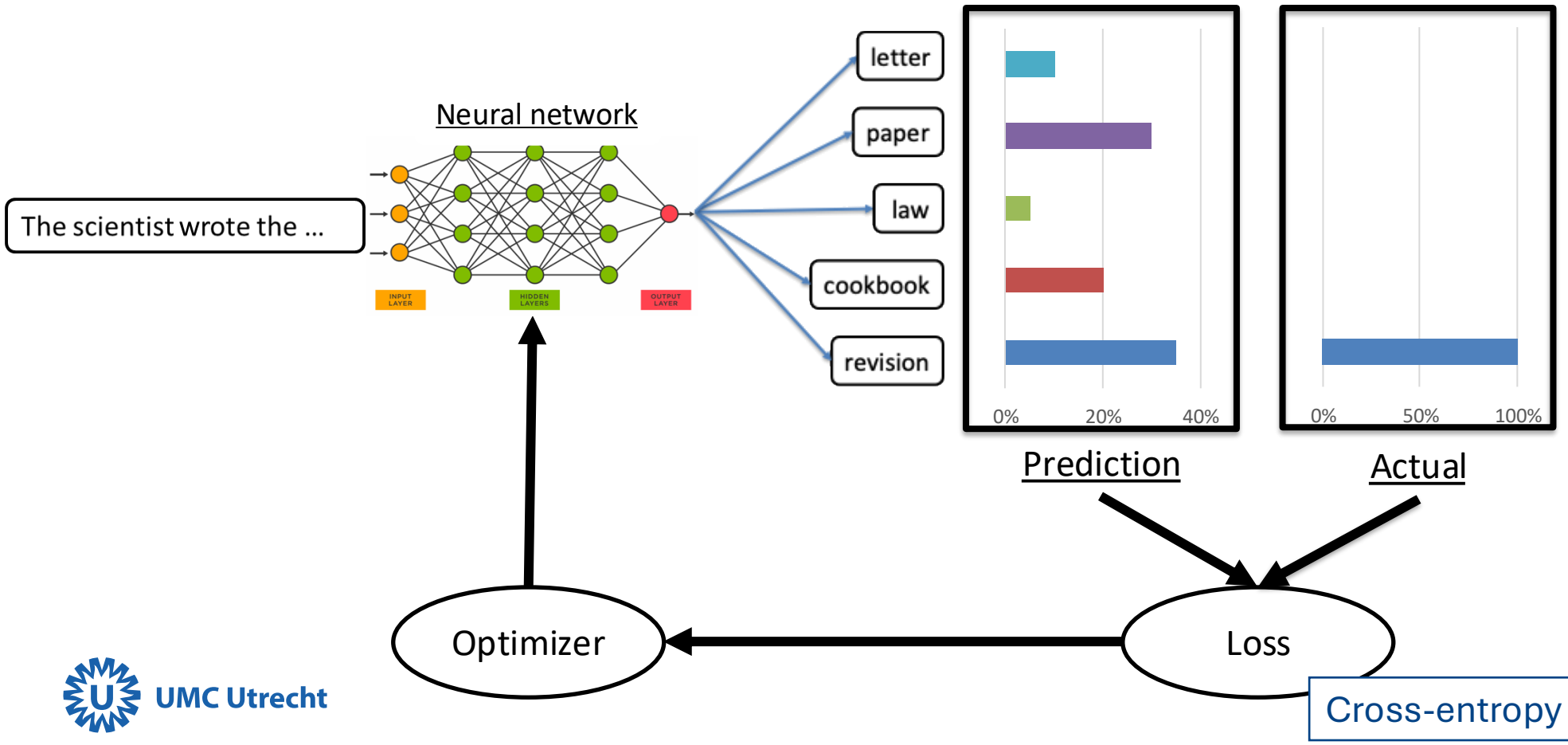
$$\mathcal{L} = -\log(p_{\text{true token}})$$



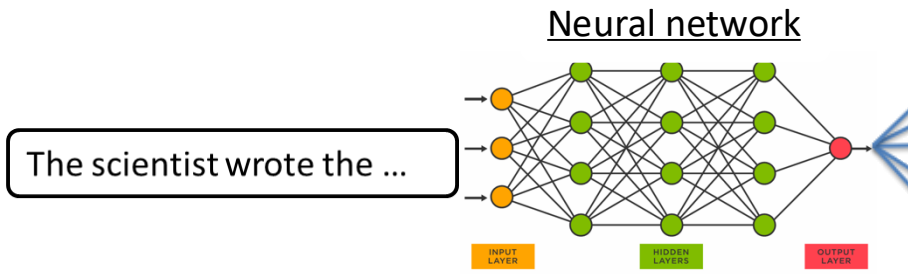
How do LLMs work? - Training



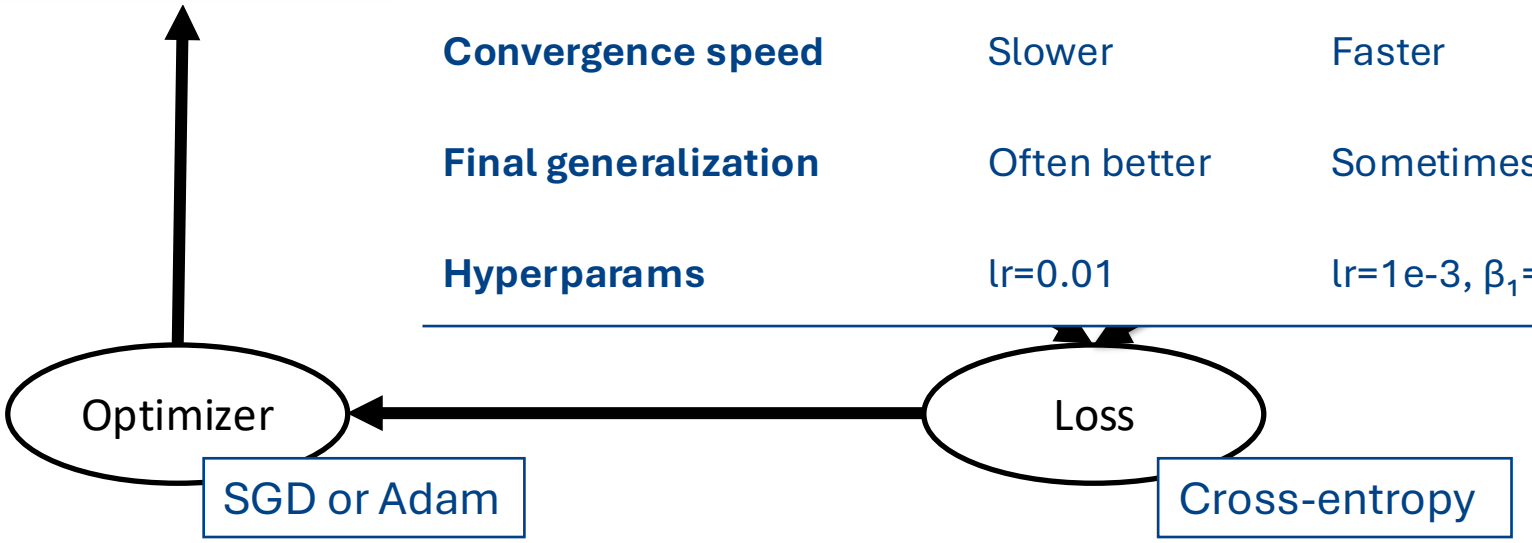
How do LLMs work? - Optimization



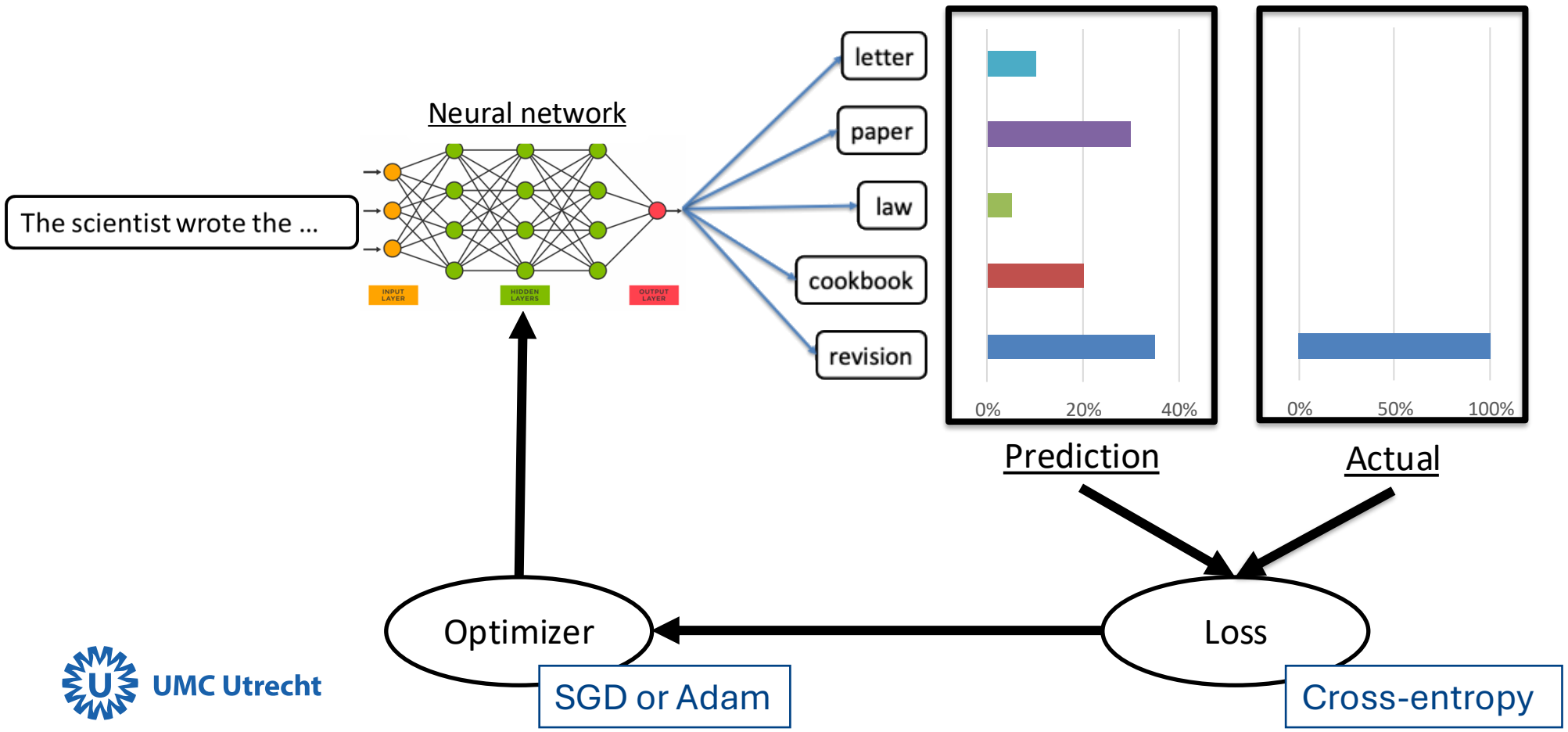
How do LLMs work? - Training



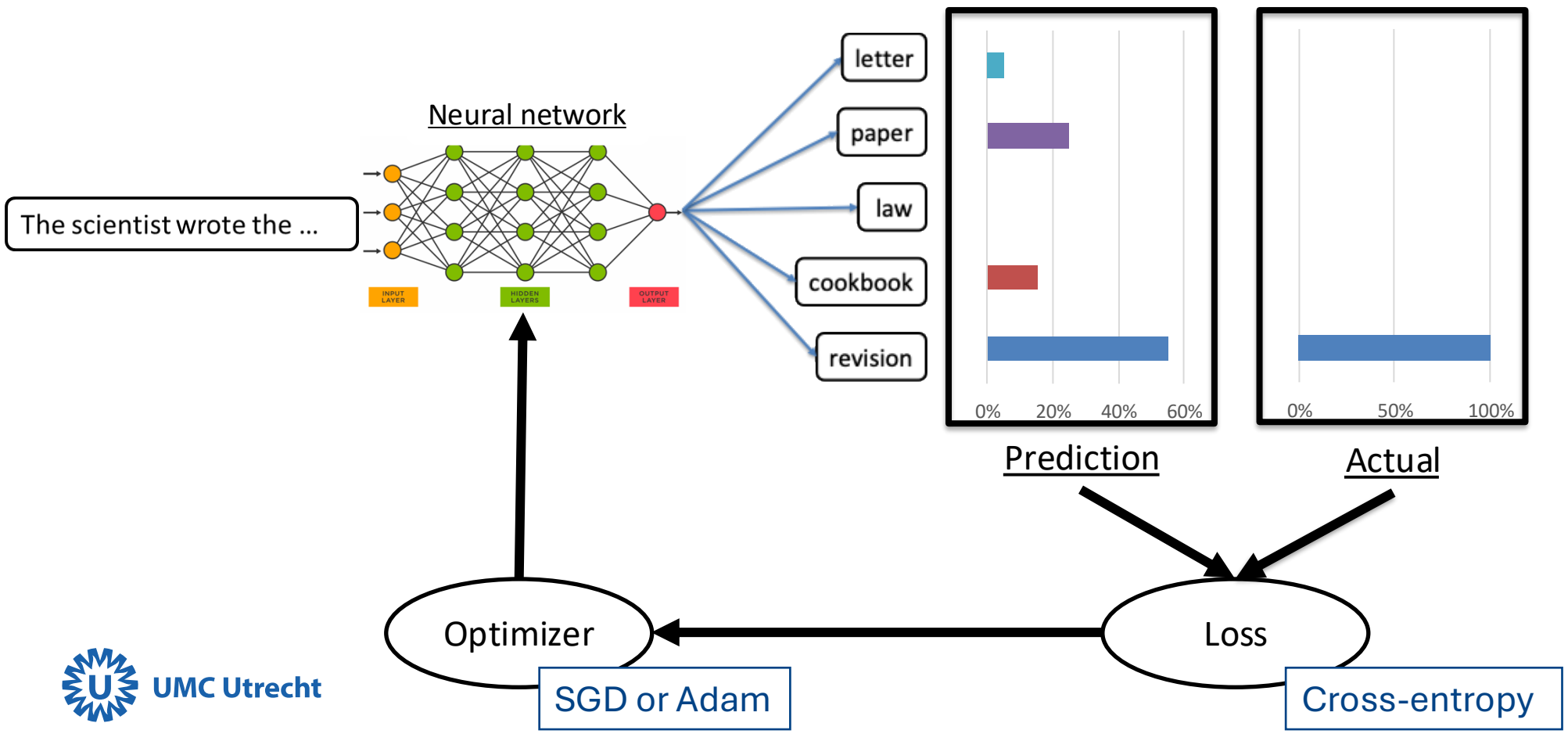
Feature	SGD	Adam
Learning rate	Global	Adaptive per parameter
Momentum	Optional	Built-in (1st moment)
Handles sparse gradients	Poorly	Very well
Convergence speed	Slower	Faster
Final generalization	Often better	Sometimes worse
Hyperparams	lr=0.01	lr=1e-3, β_1=0.9, β_2=0.999



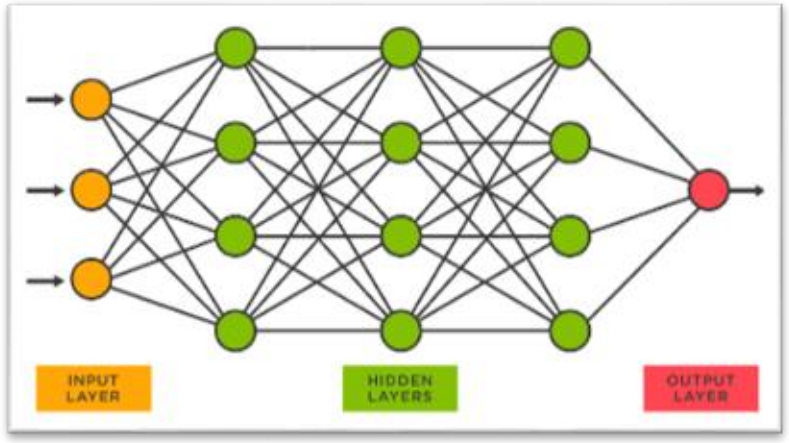
How do LLMs work? - Training

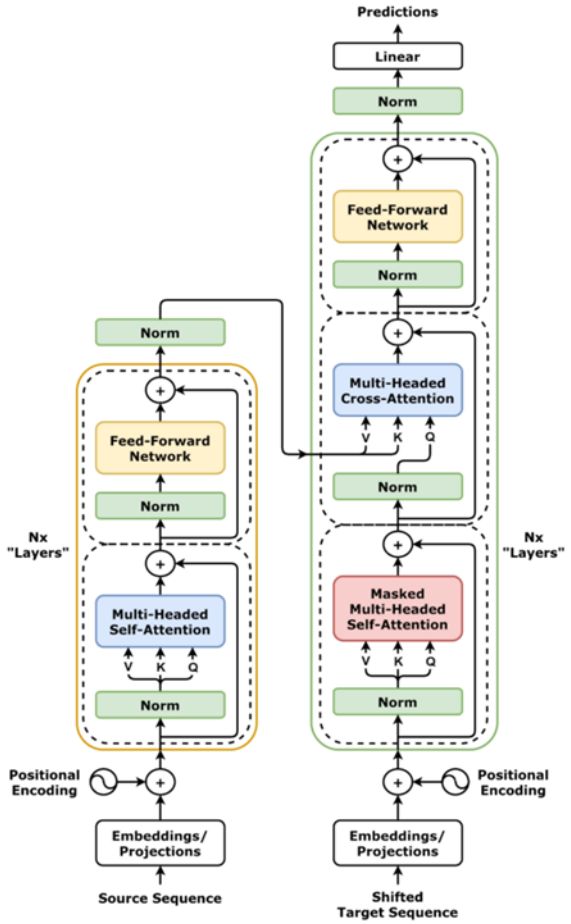


How do LLMs work? - Training



How do LLMs work? - Transformers



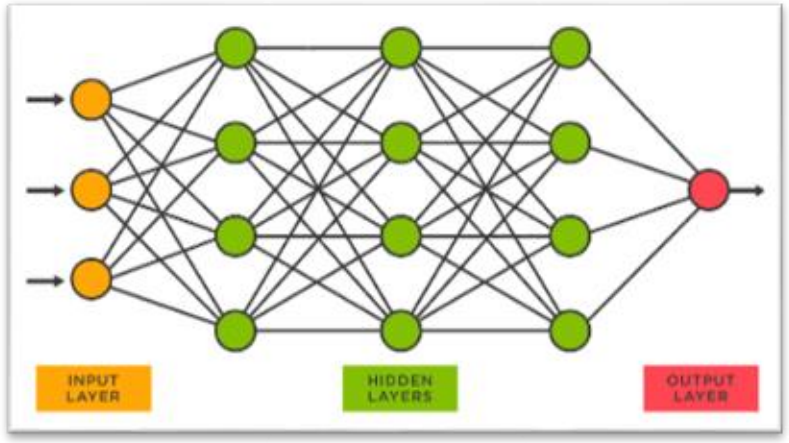


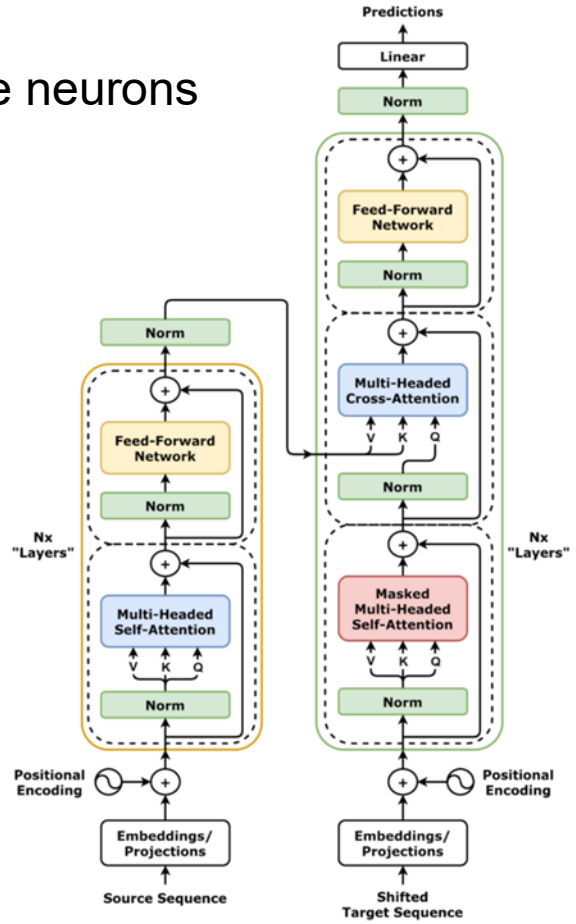
Transformer

UMC Utrecht

How do LLMs work? - Transformers

The ‘architecture’ is the way we connect the neurons



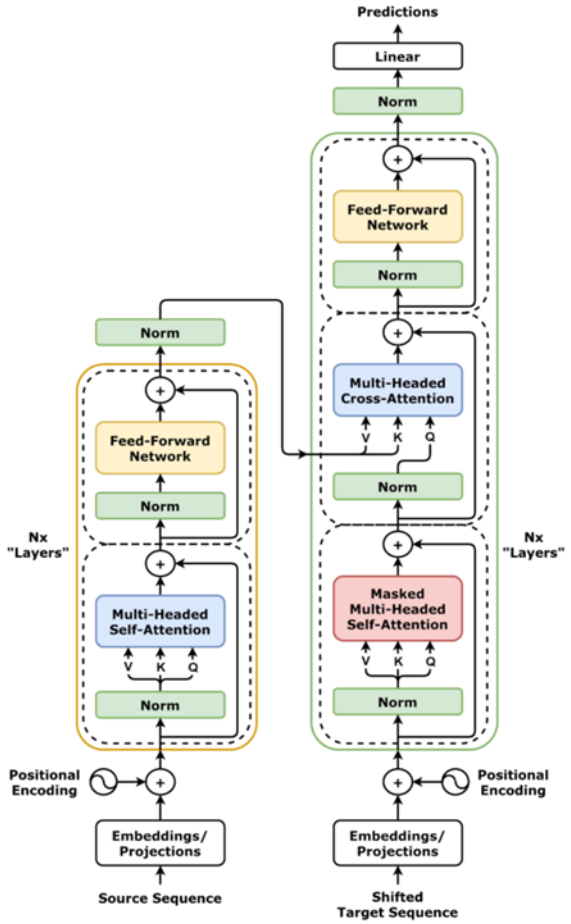


Transformer

UMC Utrecht

How do LLMs work? - Transformers

Let's walk through the architecture step-by-step…



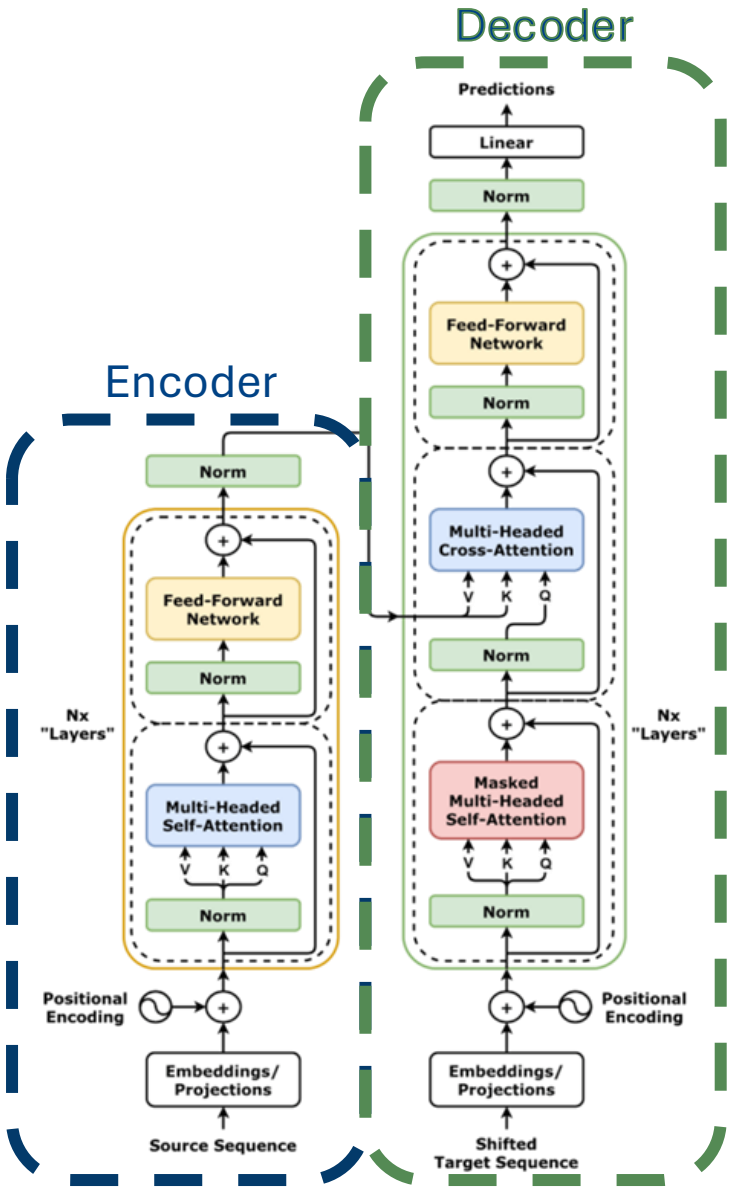
Transformer

UMC Utrecht

Core technical concepts

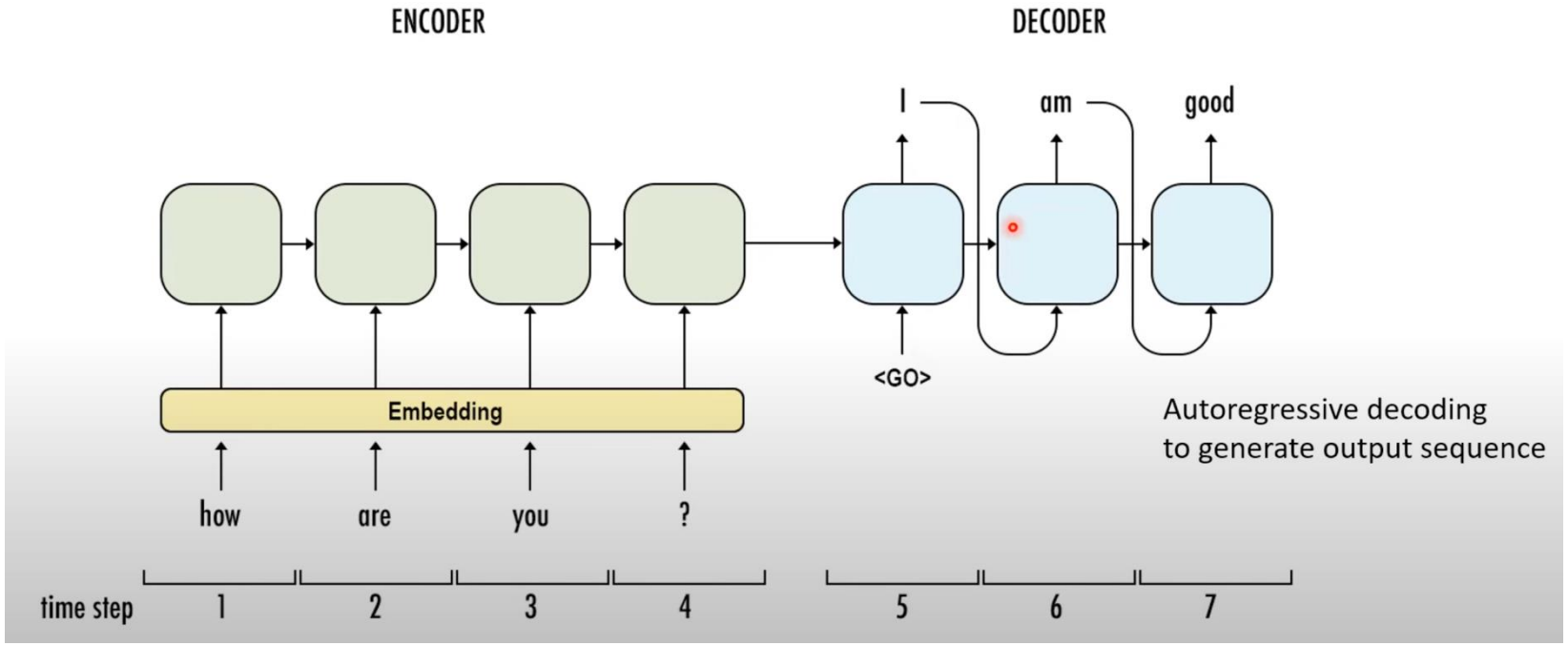
Encoder-decoder

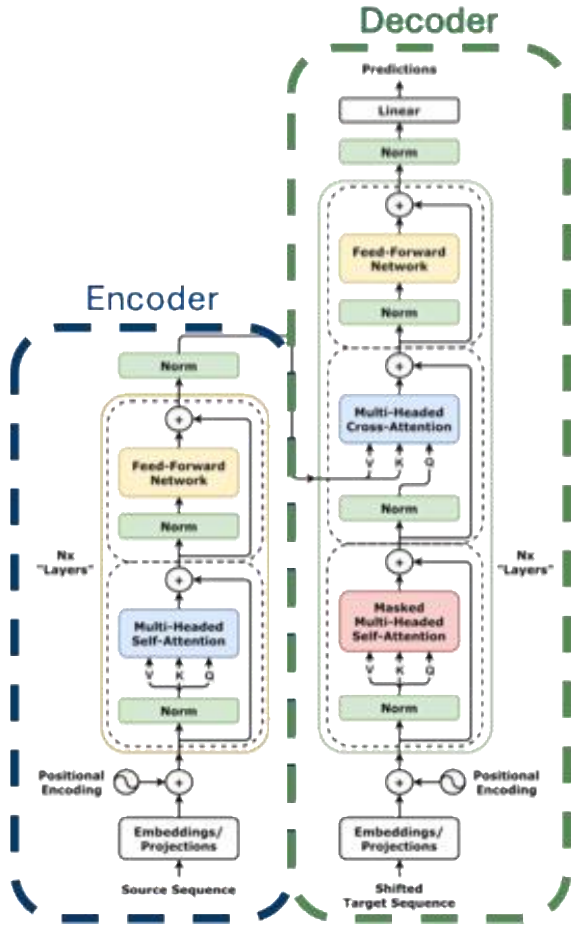
Originally, most models were encoder or encoder-decoder models



Core technical concepts

Encoder-decoder



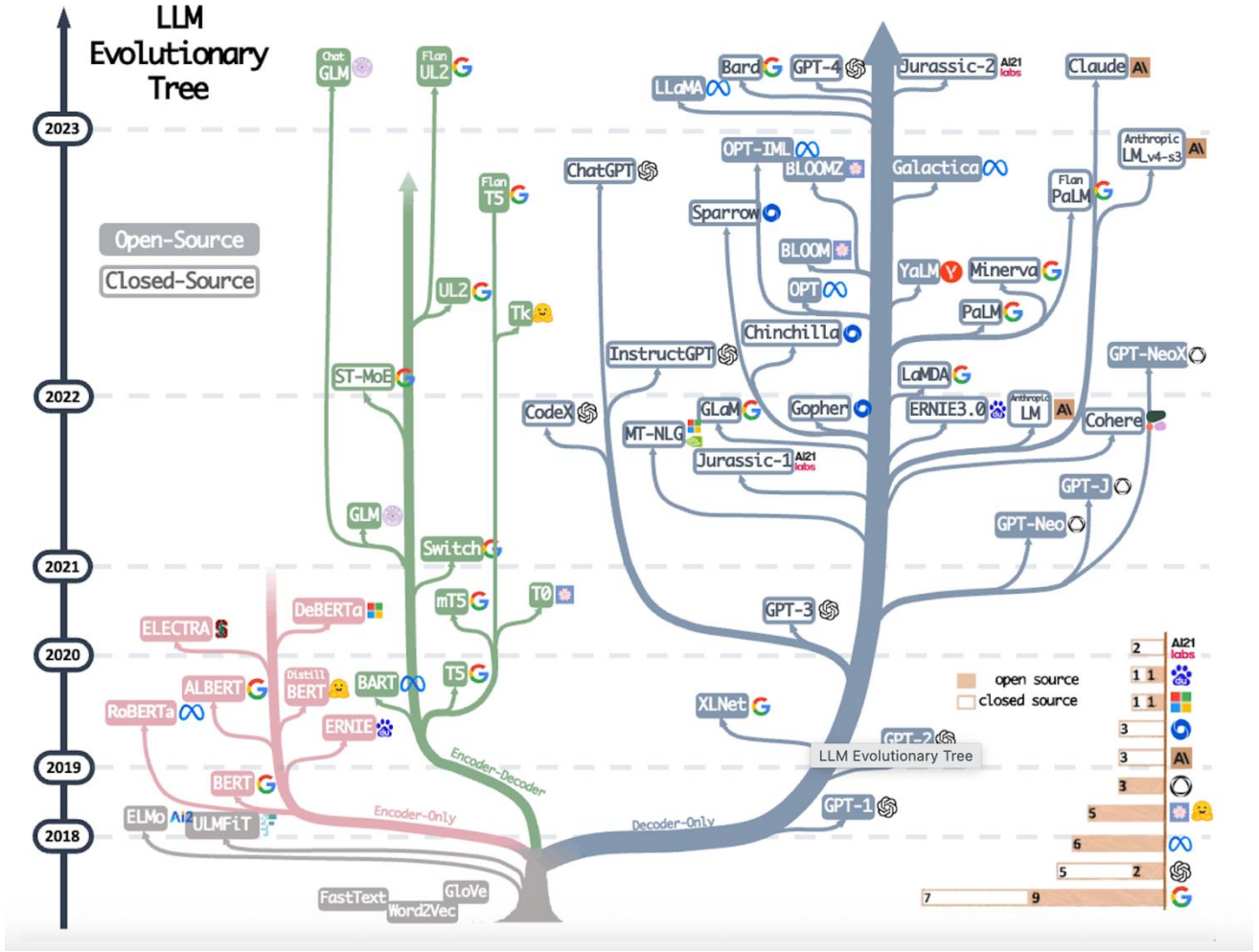


Source: https://www.youtube.com/watch?v=wOcbALDw0bU

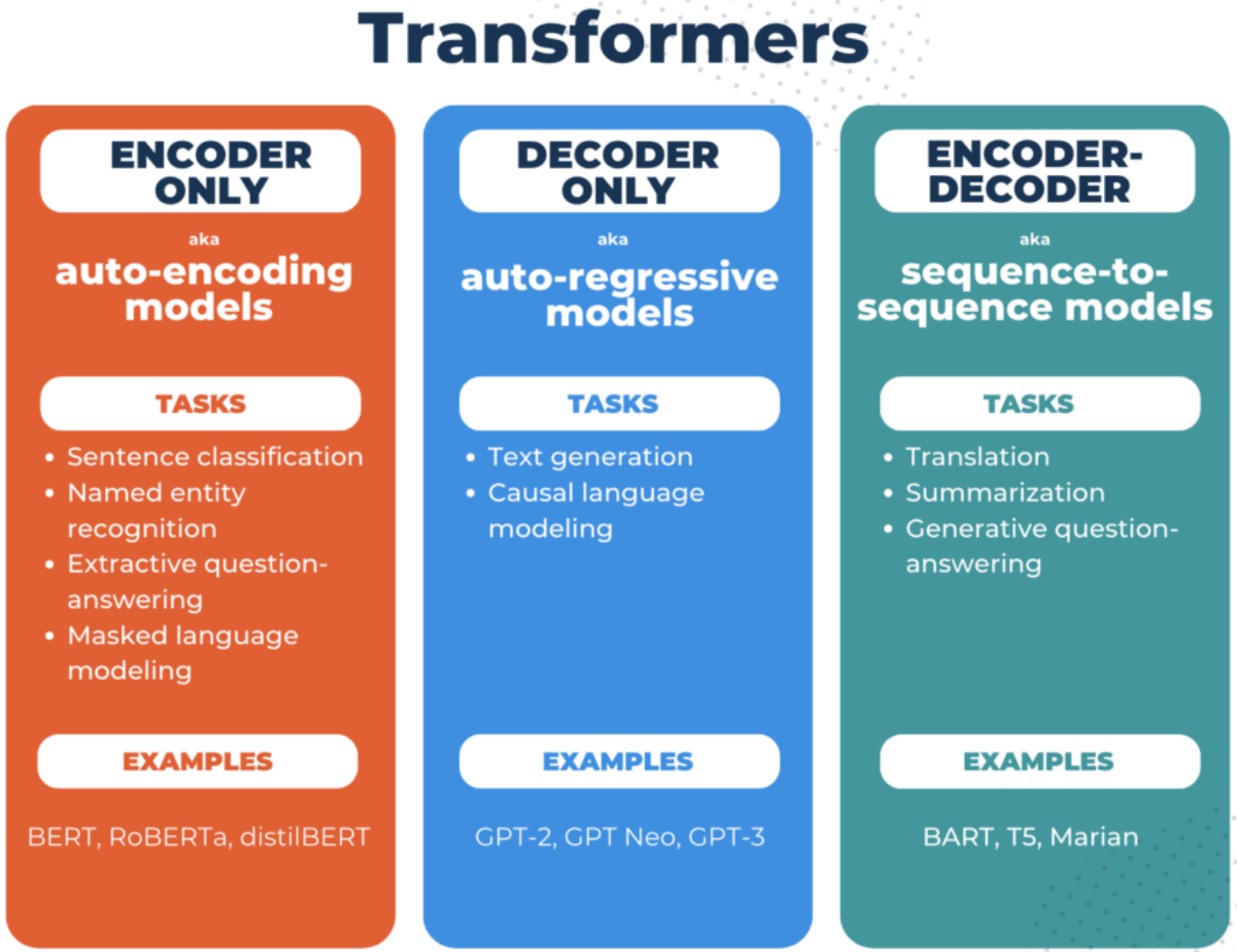
Core technical concepts

The transformer architecture

Originally, most models were encoder or encoder-decoder models

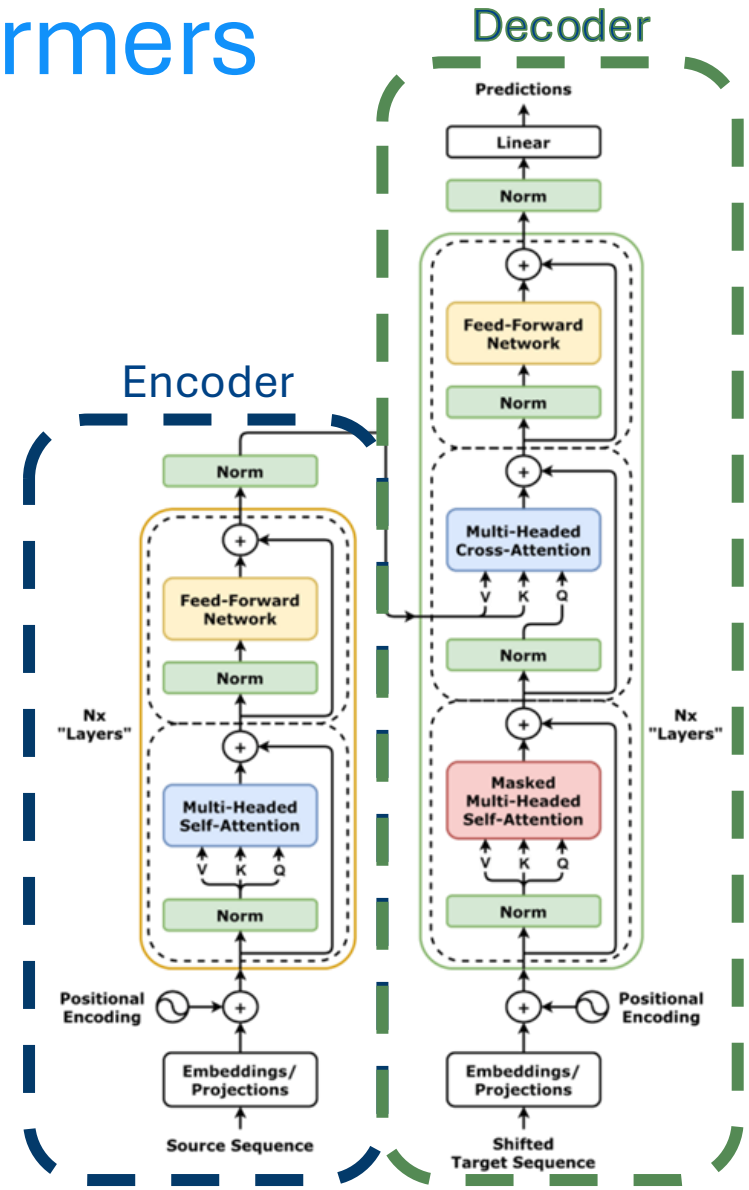


How do LLMs work? - Transformers



How do LLMs work? - Transformers

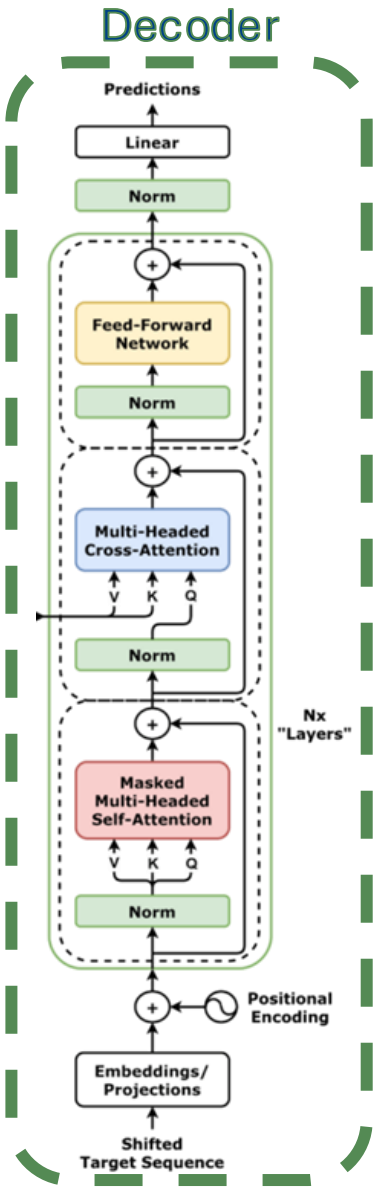
However, nowadays all LLMs only use decoders!



UMC Utrecht

How do LLMs work? - Transformers

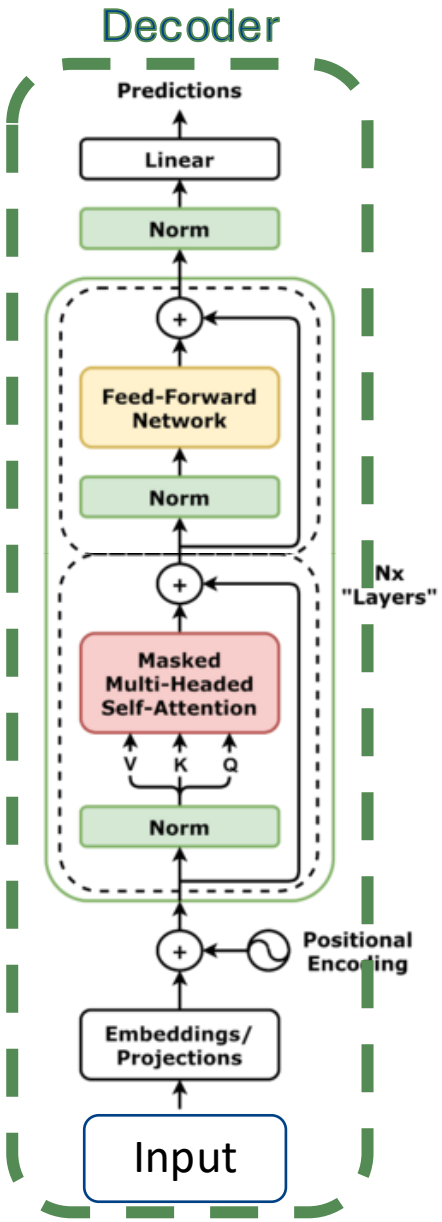
However, nowadays all LLMs only use decoders!



UMC Utrecht

How do LLMs work? - Transformers

However, nowadays all LLMs only use decoders!

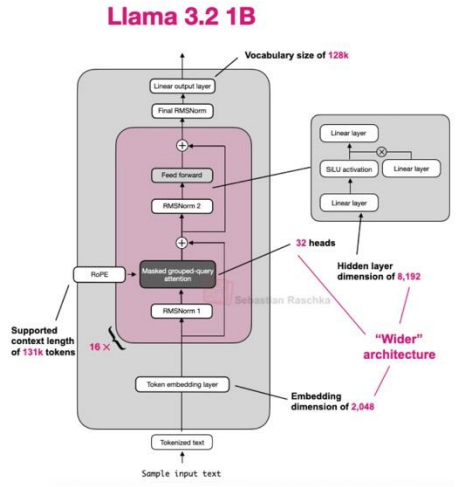


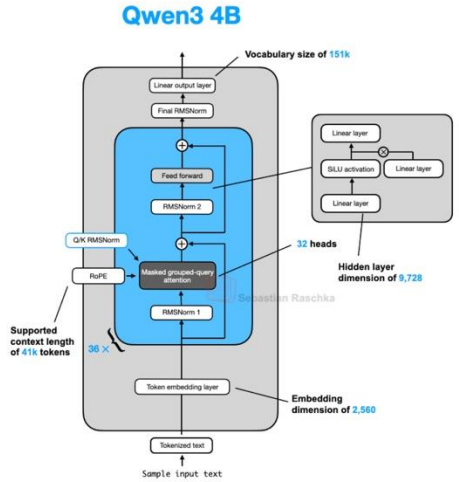
UMC Utrecht

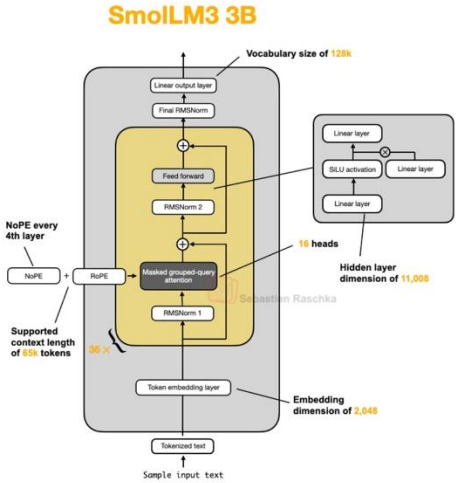
Core technical concepts

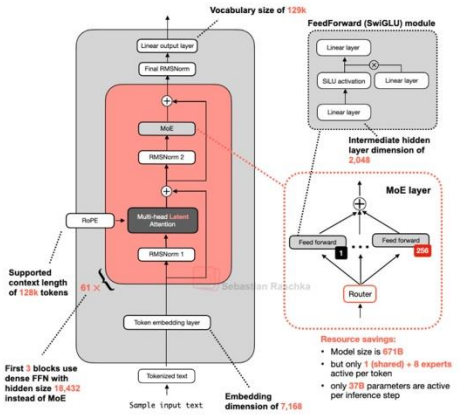
The transformer architecture

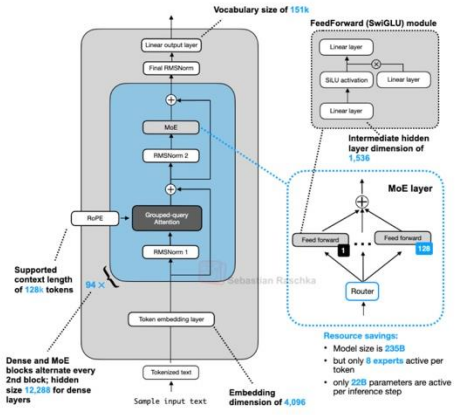
Almost all generative LLMs use the same (decoder-only) architecture!

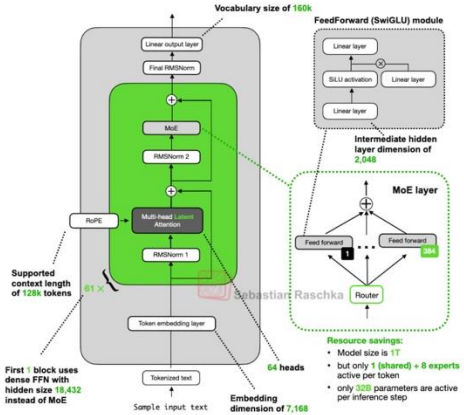










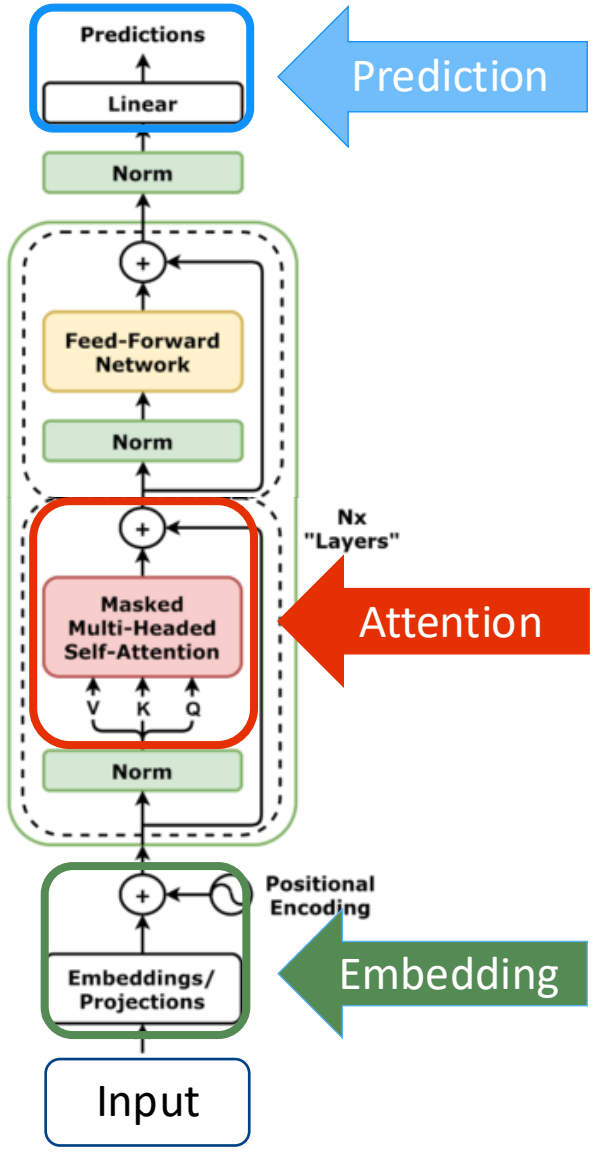


Source: https://magazine.sebastianraschka.com/p/the-big-llm-architecture-comparison

How do LLMs work? - Transformers

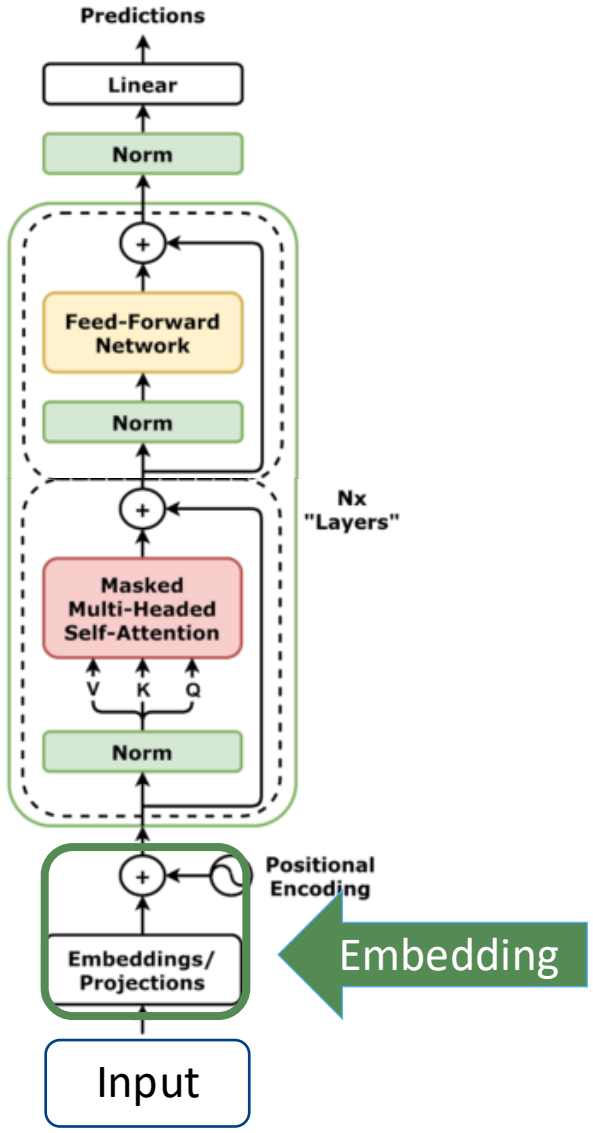
Three main components:

- **Embedding**
- **Attention**
- **Prediction**



UMC Utrecht

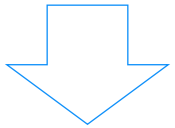
How do LLMs work? - Embedding



UMC Utrecht

How do LLMs work? - Tokenization

Short words each get their own token, but exquisite, convoluted, or infrequent words are partitioned into diminutive segments.



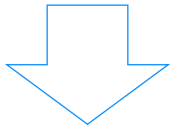
Tokenizer

Short words each get their own token, but exquisite, convoluted, or inf
requent words are partitioned into diminutive segments.

UMC Utrecht

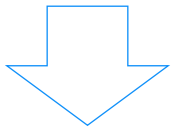
How do LLMs work? – Token IDs

```
Short words each get their own token, but exquisite, convoluted, or
infrequent words are partitioned into diminutive segments.
```



Tokenizer

```
Short words each get their own token, but exquisite, convoluted, or inf
requent words are partitioned into diminutive segments.
```

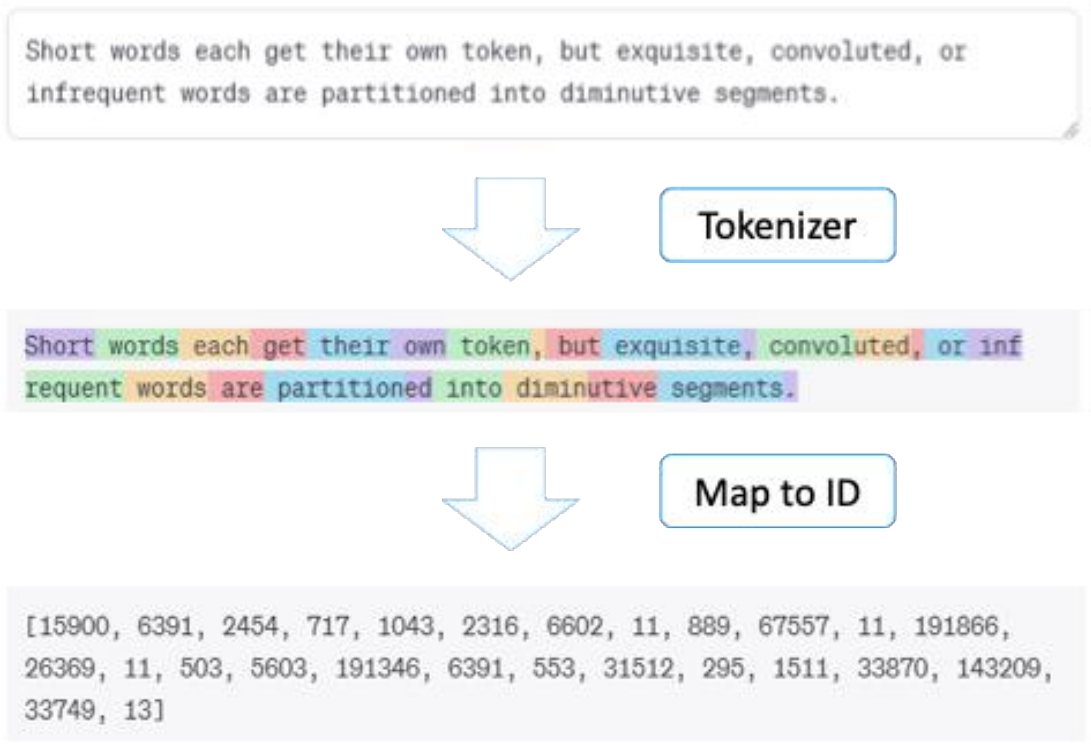


Map to ID

```
[15900, 6391, 2454, 717, 1043, 2316, 6602, 11, 889, 67557, 11, 191866,
26369, 11, 503, 5603, 191346, 6391, 553, 31512, 295, 1511, 33870, 143209,
33749, 13]
```

UMC Utrecht

How do LLMs work? – Token IDs



Try it yourself!

platform.openai.com/tokenizer

tokenizer.model.box

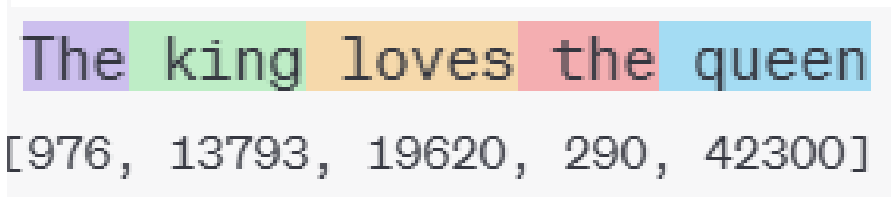
Try (for example):

- **A very long word**
- **A very short word**
- **Words in a different language**
- **Punctuation marks**

UMC Utrecht

How do LLMs work? – Token IDs

The king loves the queen



How do LLMs work? – Vector embedding

The king loves the queen

The king loves the queen

[976, 13793, 19620, 290, 42300]

Vector embedding (learned!)

1	0.02	0.93	0.04	0.02	0.94
2	0.01	0.80	0.40	0.01	0.21
3	0.01	0.21	0.60	0.01	0.83
N	…	…	…	…	…

For example: N = 4096 for Llama 3

How do LLMs work? – Vector embedding

Vector embeddings are learned

Vector embeddings give meaning to tokens

	The	King	loves	the	Queen
…	…	…	…	…	…
Royalness	0.02	0.93	0.04	0.02	0.94
Menliness	0.01	0.80	0.40	0.01	0.21
Womanliness	0.01	0.21	0.60	0.01	0.83
…	…	…	…	…	…

How do LLMs work? – Vector embedding

Vector embeddings are learned

Vector embeddings give meaning to tokens

They can also be a source of bias!

	The	King	loves	the	Queen
…	…	…	…	…	…
Royalness	0.02	0.93	0.04	0.02	0.94
Menliness	0.01	0.80	0.40	0.01	0.21
Womanliness	0.01	0.21	0.60	0.01	0.83
…	…	…	…	…	…

How do LLMs work? – Vector embedding

Vector embeddings are learned

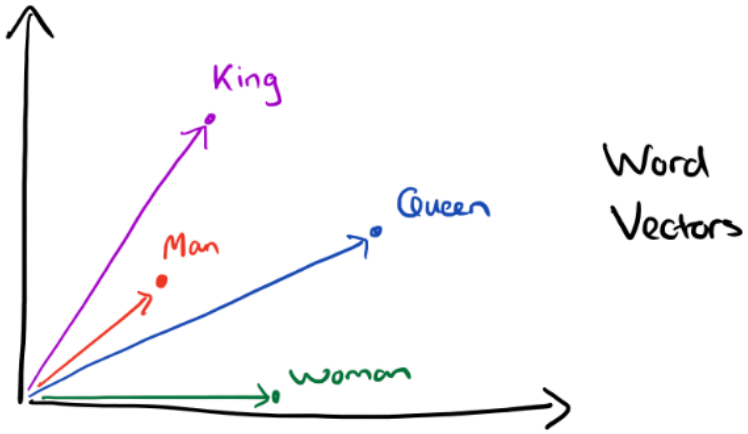
Vector embeddings give meaning to tokens

They can also be a source of bias!

Where does this bias come from?

	The	King	loves	the	Queen
…	…	…	…	…	…
Royalness	0.02	0.93	0.04	0.02	0.94
Menliness	0.01	0.80	0.40	0.01	0.21
Womanliness	0.01	0.21	0.60	0.01	0.83
…	…	…	…	…	…

How do LLMs work? – Vector embedding

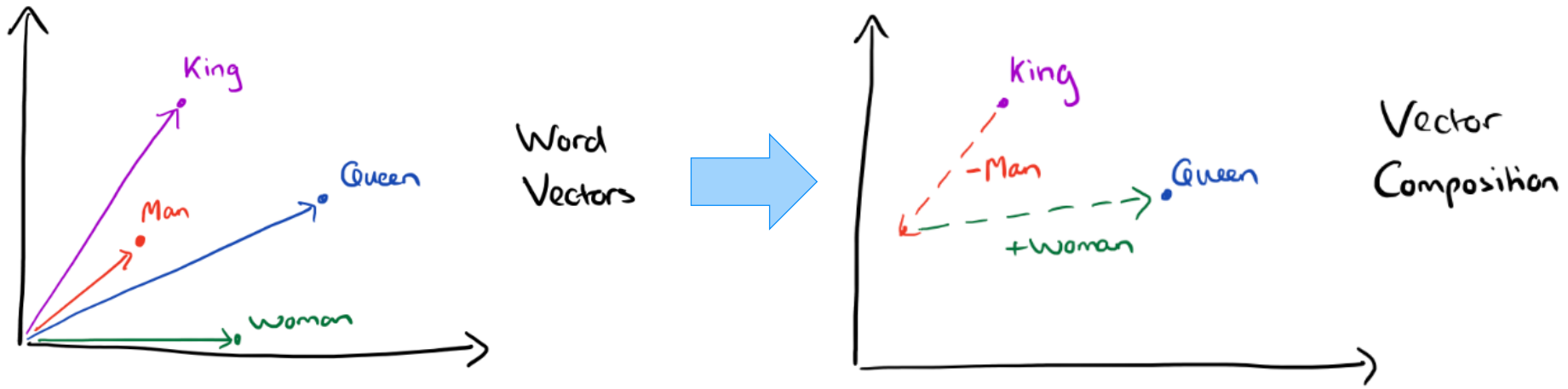


Vector embeddings have interesting interconnections…

UMC Utrecht

Images by Adrian Colyer: https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors/

How do LLMs work? – Vector embedding



Vector embeddings have interesting interconnections…

projector.tensorflow.org

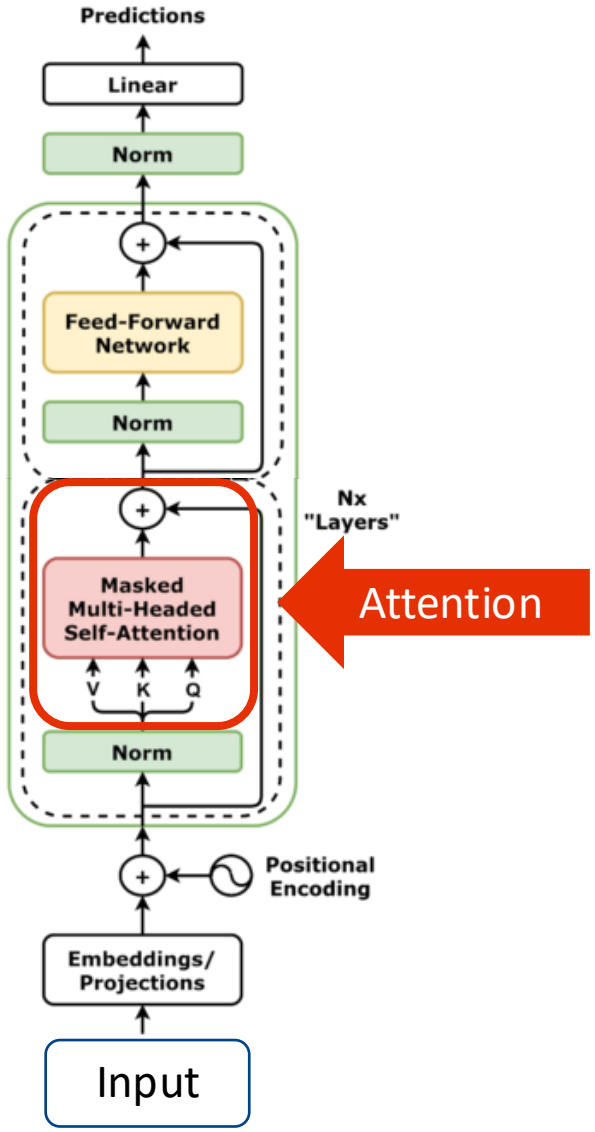
tokscope.site

UMC Utrecht

For people that want to work on classification using embeddings, check out this page:
https://platform.openai.com/docs/guides/embeddings

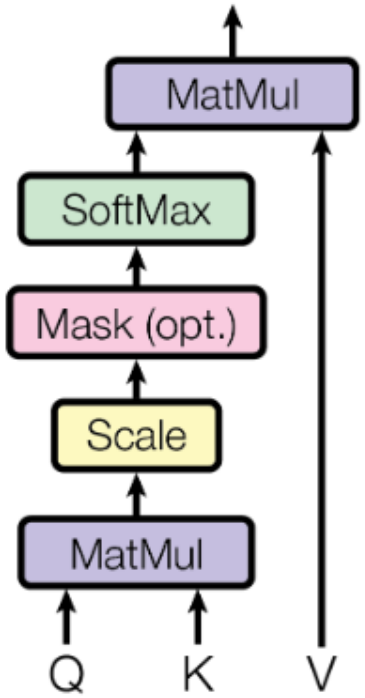
Images by Adrian Colyer: https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors/

How do LLMs work? – Attention

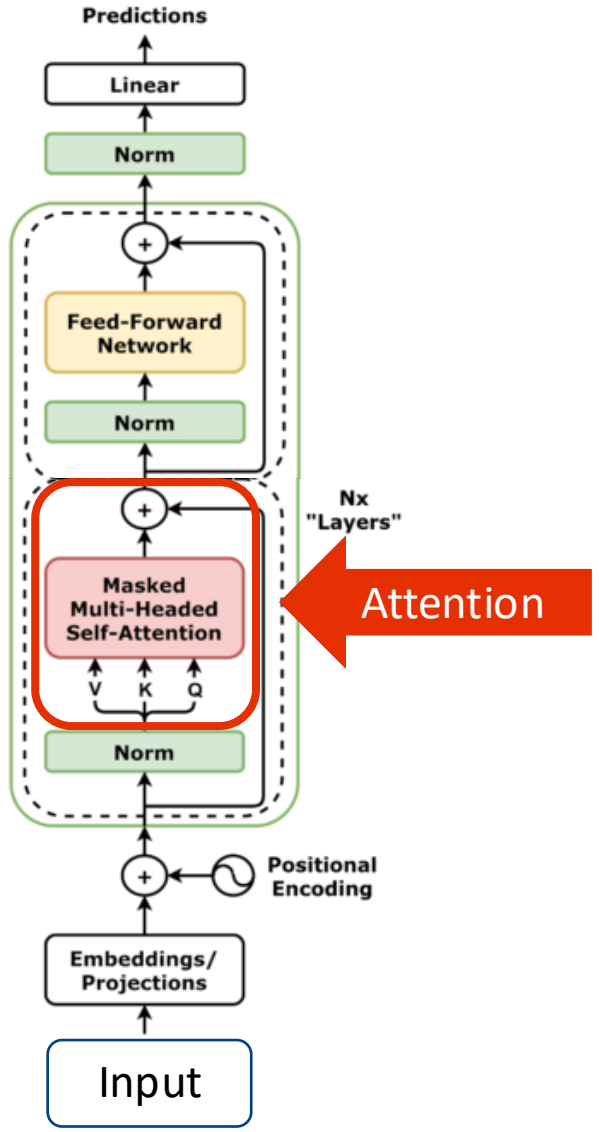


UMC Utrecht

How do LLMs work? – Query-Key-Value

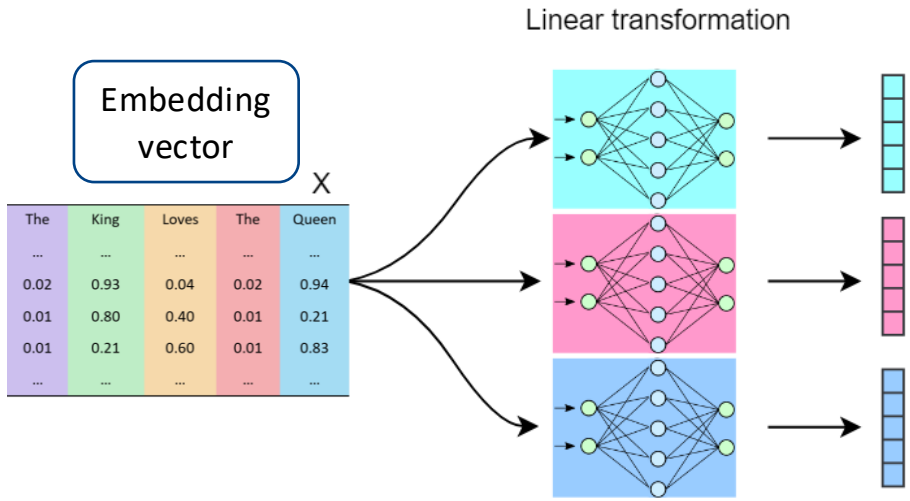


Query – **K**ey - **V**alue



UMC Utrecht

How do LLMs work? – Query-Key-Value

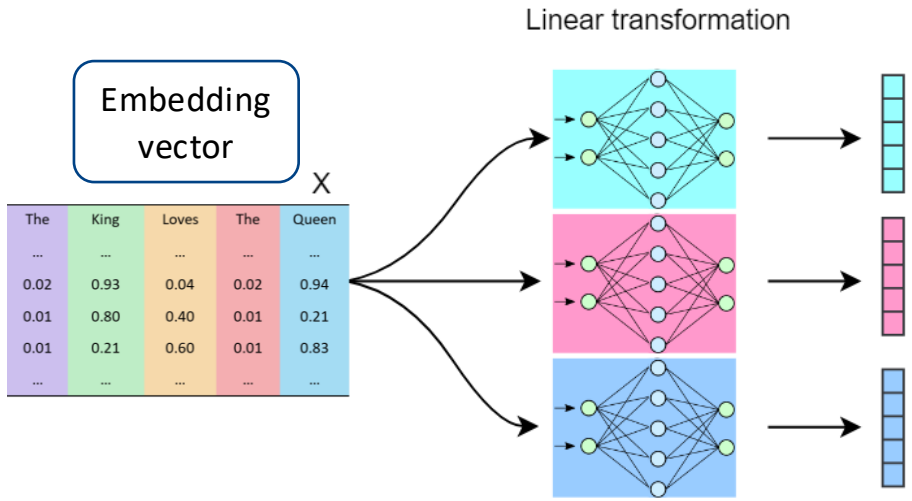


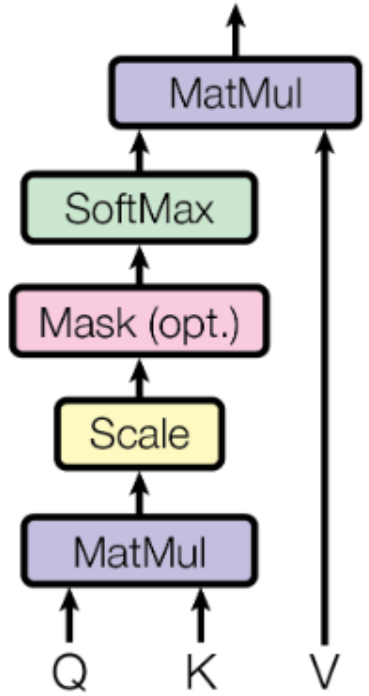
Query: represents what the current token is looking for.

Key: represents what each token offers that matches a query.

Value: contains the information to be passed along once attention is assigned.

How do LLMs work? – Query-Key-Value



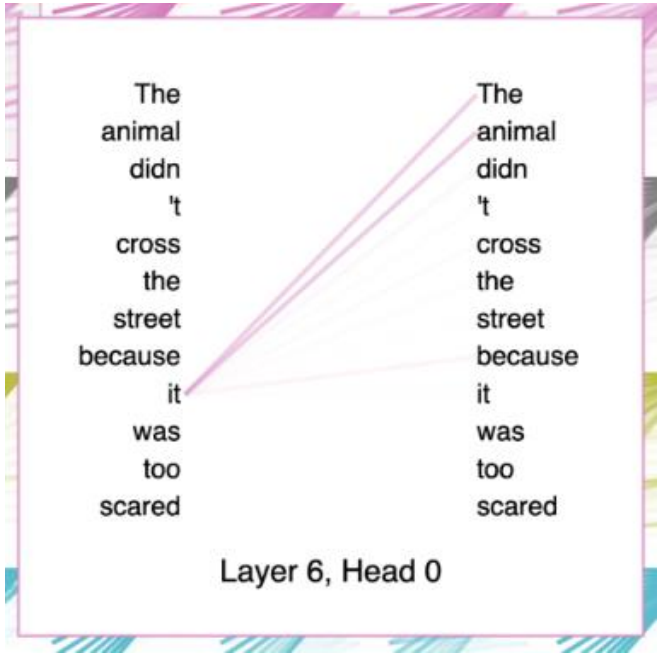


Query – **K**ey - **V**alue

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

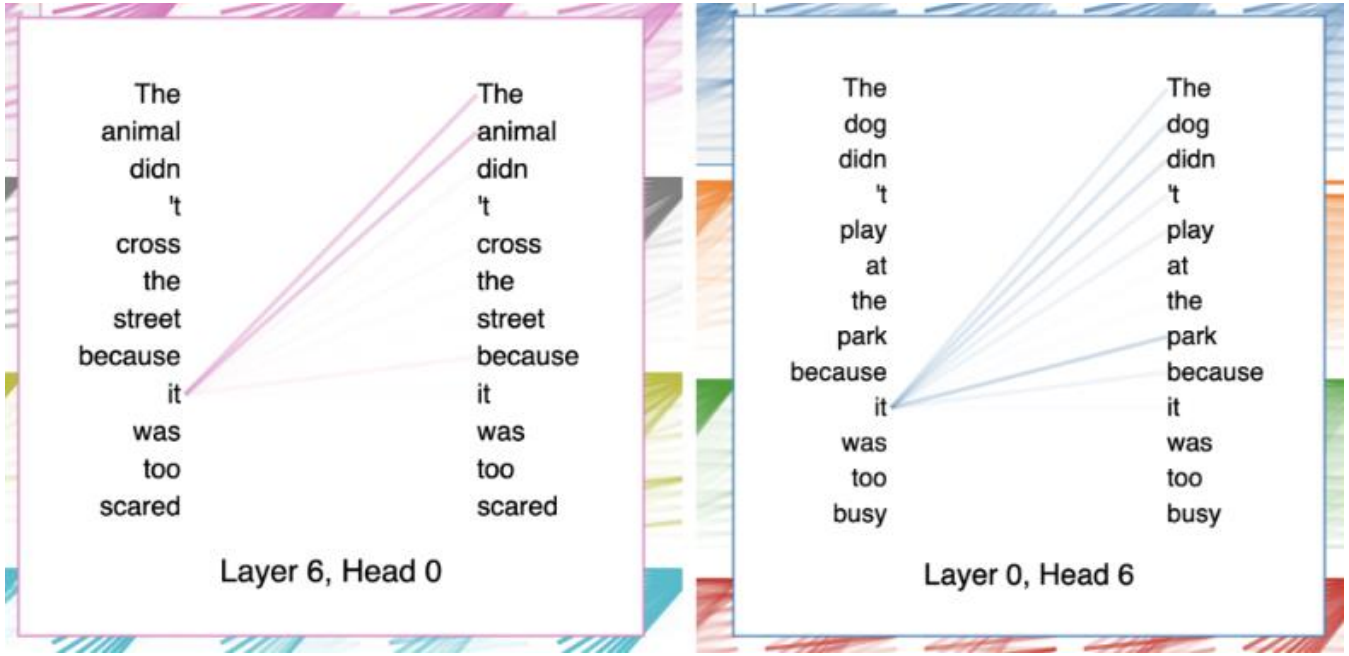
UMC Utrecht

How do LLMs work? – Attention



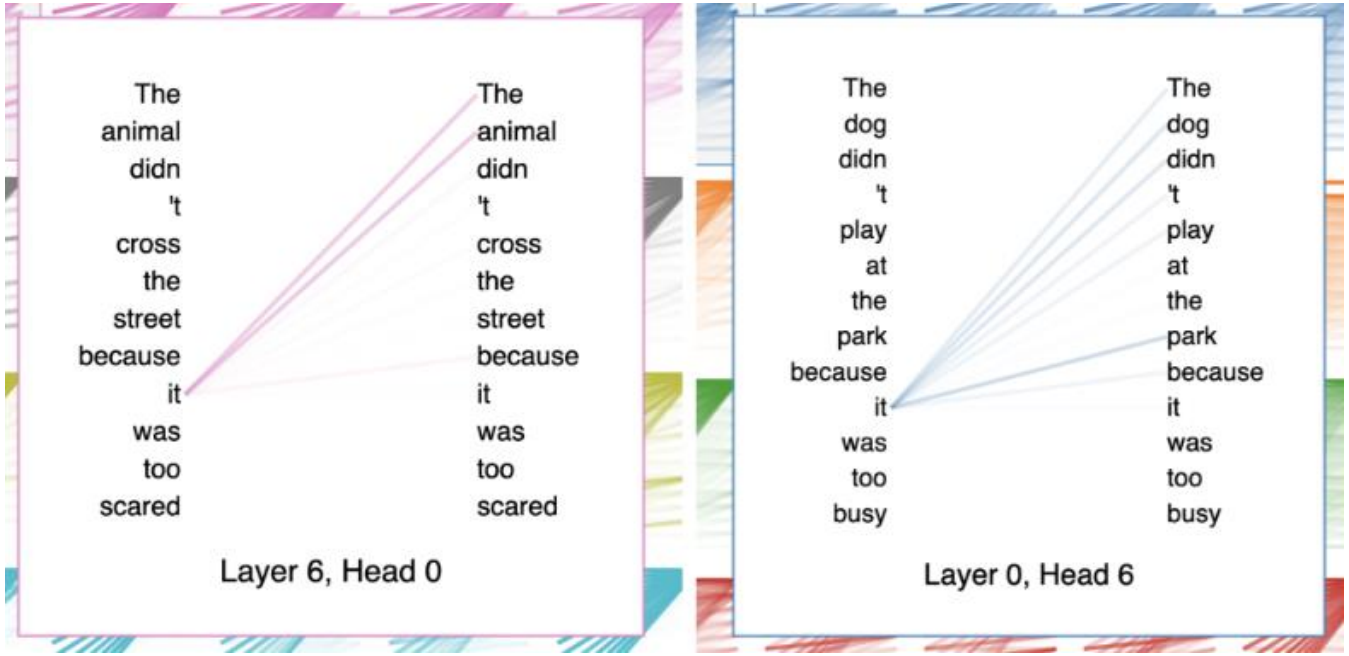
Source: https://www.comet.com/site/blog/explainable-ai-for-transformers/

How do LLMs work? – Attention



Source: https://www.comet.com/site/blog/explainable-ai-for-transformers/

How do LLMs work? – Attention



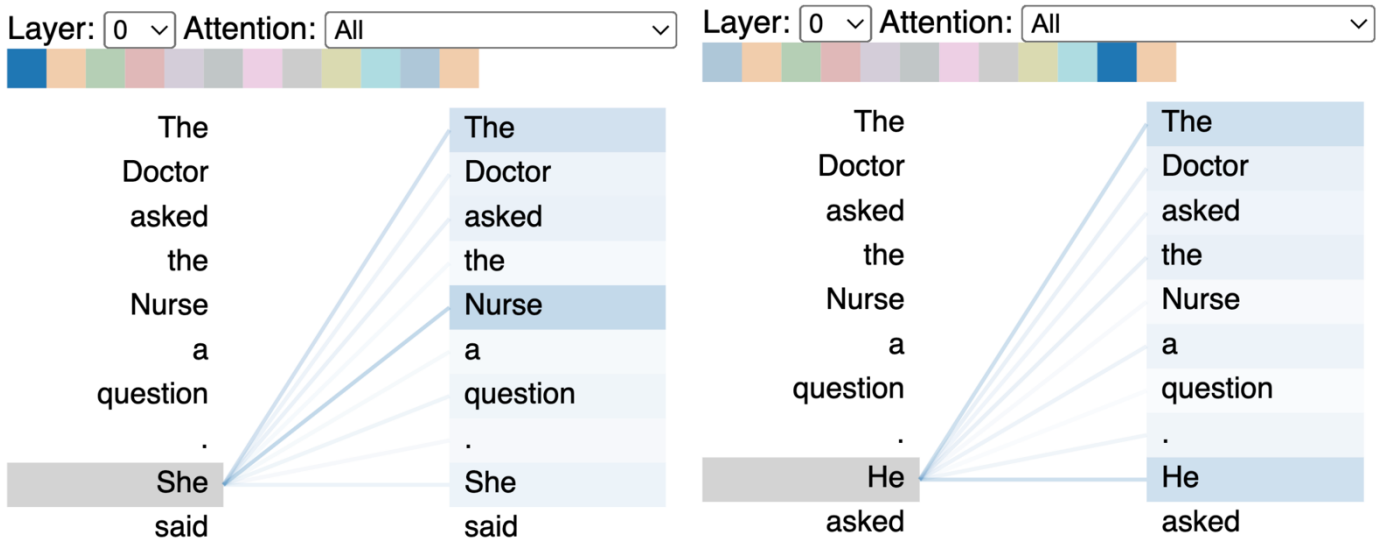
Source: https://www.comet.com/site/blog/explainable-ai-for-transformers/

Meaning + Attention = Understanding?

UMC Utrecht

How do LLMs work? – Risks

Why is this important to know?

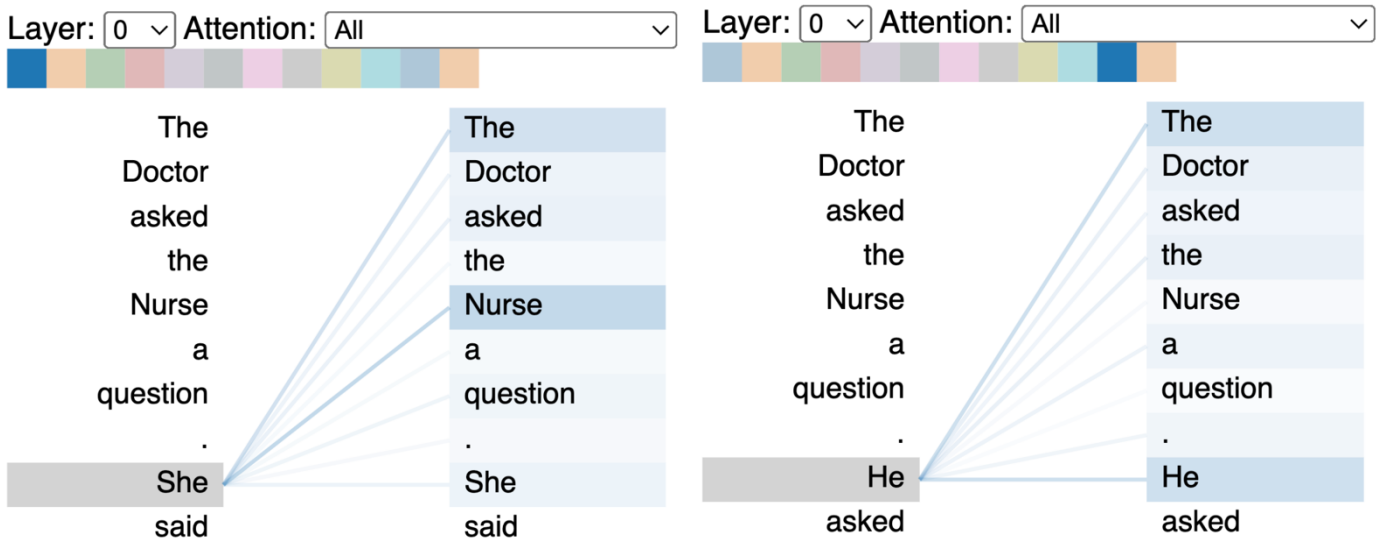


Source: https://www.comet.com/site/blog/explainable-ai-for-transformers/

UMC Utrecht

How do LLMs work? – Risks

Why is this important to know?
To understand the risks you need to understand the sources of error



Source: https://www.comet.com/site/blog/explainable-ai-for-transformers/

UMC Utrecht

From yesterday: example of learning bias

HA

In the sentence "the nurse married the doctor because she was pregnant", who was pregnant?

From yesterday: example of learning bias

HA

In the sentence "the nurse married the doctor because she was pregnant", who was pregnant?

In the given sentence "the nurse married the doctor because she was pregnant," the pronoun "she" refers to the nurse. Therefore, it means that the nurse was pregnant.

From yesterday: example of learning bias

HA

In the sentence "the nurse married the doctor because she was pregnant", who was pregnant?

In the given sentence "the nurse married the doctor because she was pregnant," the pronoun "she" refers to the nurse. Therefore, it means that the nurse was pregnant.

HA

Could "she" refer to the doctor instead?

HA

In the sentence "the nurse married the doctor because she was pregnant", who was pregnant?

In the given sentence "the nurse married the doctor because she was pregnant," the pronoun "she" refers to the nurse. Therefore, it means that the nurse was pregnant.

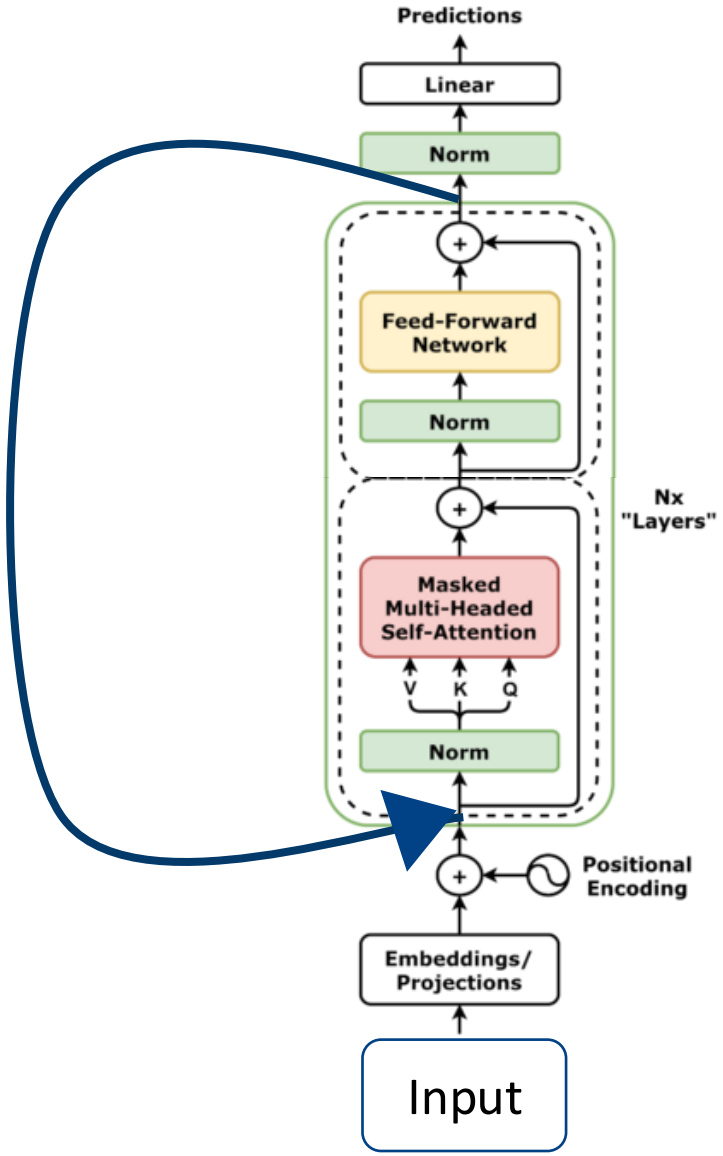
HA

Could "she" refer to the doctor instead?

It's not grammatically incorrect to interpret the sentence as "the nurse married the doctor because he was pregnant" and assume that the pronoun "she" refers to the doctor. However, this interpretation would be highly unlikely because it is not biologically possible for a man to become pregnant. Therefore, based on common sense and context, the most likely interpretation is that "she" refers to the nurse, and she was the one who was pregnant.

How do LLMs work?

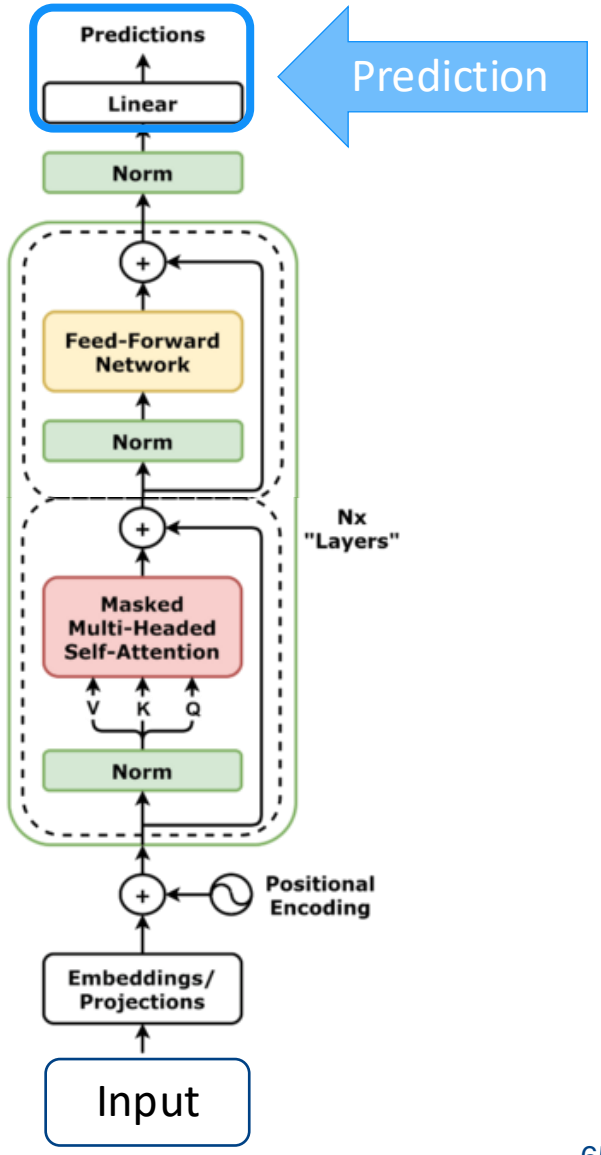
‘N’ times, with N the number of transformer layers



How do LLMs work? – Prediction

Final output layer:
- Transformer layers output final embedded vector.
- Linear (output) layer converts embedding to a score for each token in the vocabulary
- These scores are called logits.

Softmax function:
- Softmax turns logits into probabilities for each token in the vocabulary
- All probabilities will sum to 1.
- Token is chosen from multinomial distribution using argmax, top-p, top-k and/or temperature

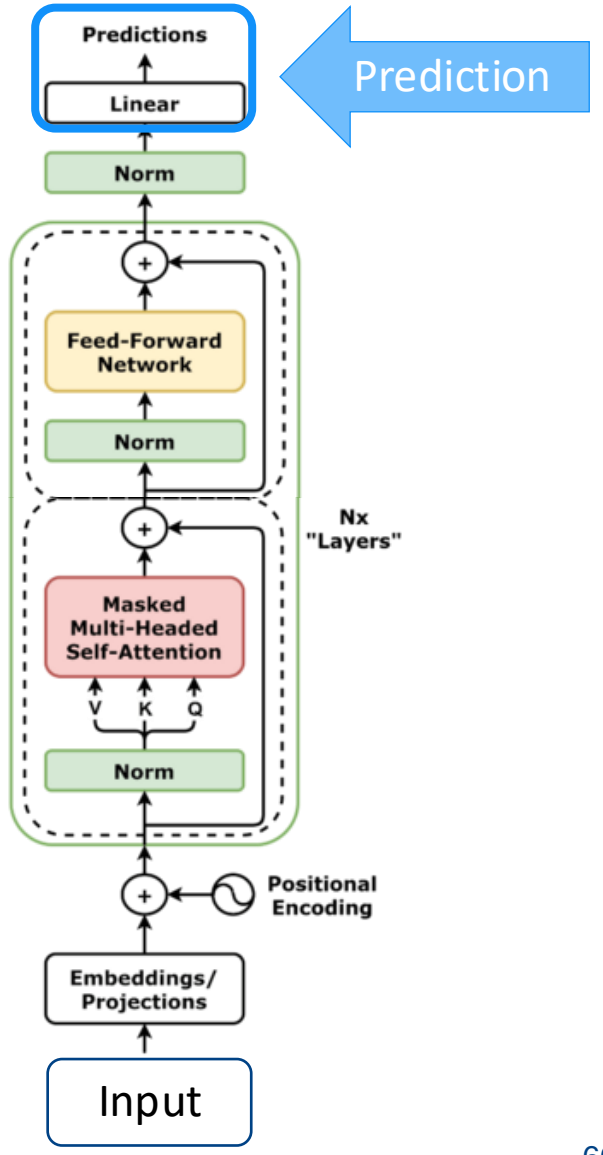


How do LLMs work? – Prediction

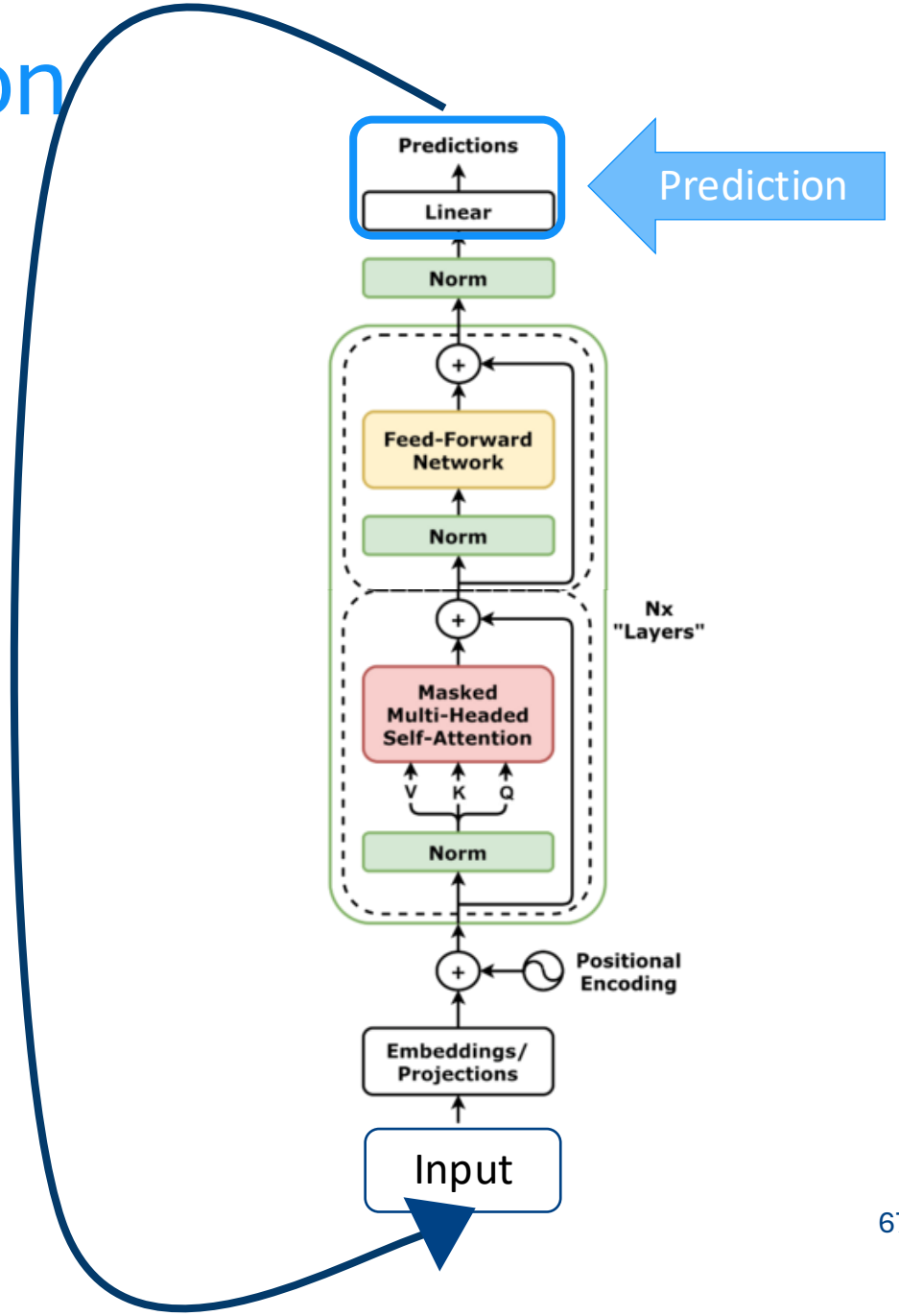
Top-k sampling:
- Keep only the **k** most probable next tokens
- Controls vocabulary size considered (e.g., k = 10).

Top-p sampling:
- Keep the smallest set of tokens whose cumulative probability ≥ *p*
- Controls cumulative probability (e.g., *p = 0.9*).

Temperature (T):
- Divides logits before softmax by a factor T
- Lower T < 1 => more deterministic
- Higher T > 1 => more random.

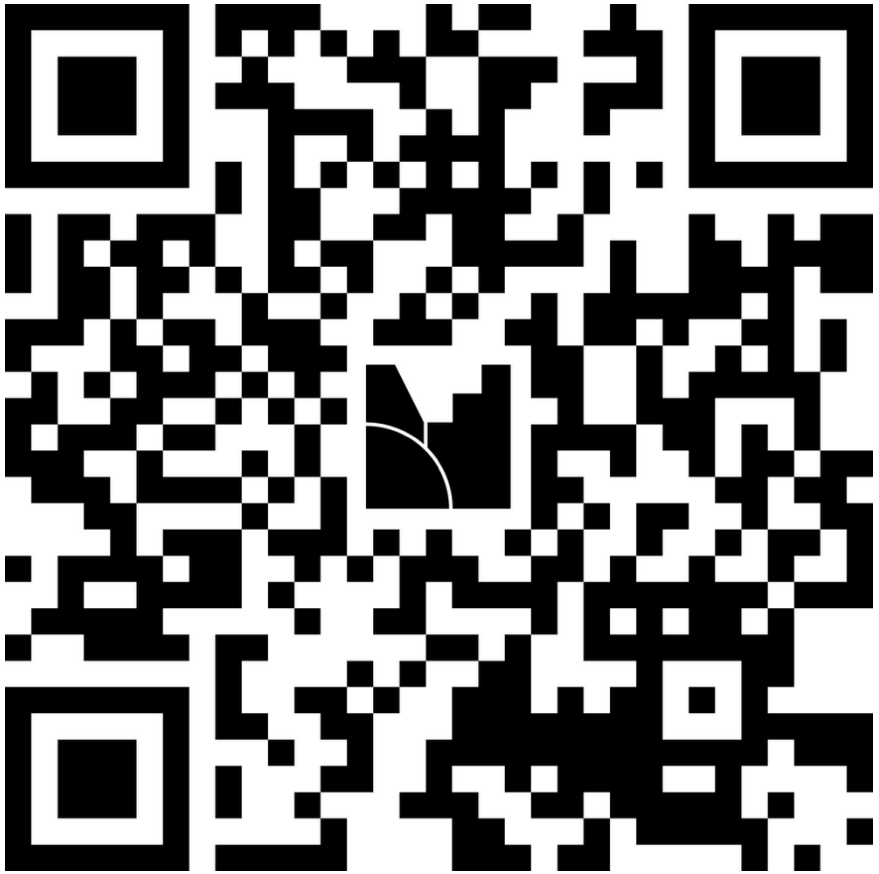


How do LLMs work? – Prediction



Knowledge quiz!

Code: 7935 2831



https://www.menti.com/alws5w7igmes

How do LLMs work?

Two online visualizations

Transformer explained: https://poloclub.github.io/transformer-explainer/

Model architecture: https://bbycroft.net/llm

We will use these while we go through the code!

BYOLLM: Build Your Own Large Language Model

1. Data
2. Tokenization
3. Embedding (vectorization, positional encoding)
4. Attention
5. Training
6. Validation and test
7. Next-word-prediction

LINK TO NOTEBOOK

BYOLLM: Data

First step is to load a dataset to learn from.

A few examples have been preloaded, you can add more if you like.

We load all data to one long string so we can conveniently cut it up into smaller pieces later.

BYOLLM: Data

Part 1: The dataset

The first step to training your own LLMs is finding a dataset that you want to train on.

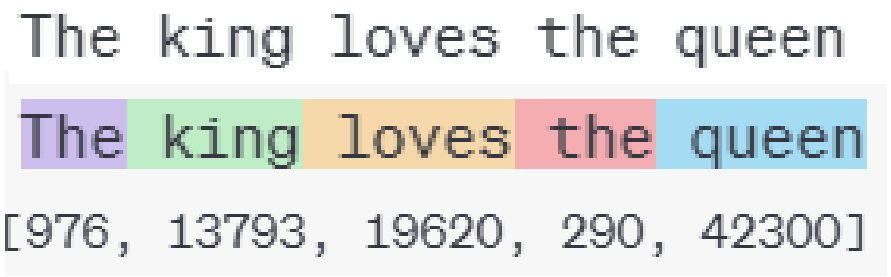
```python
# Just run this

from datasets import load_dataset

# Function to load the dataset based on user choice
def load_text_dataset(dataset_name):
    if dataset_name == 'imdb':
        dataset = load_dataset('imdb')
        text_key = 'text'
    elif dataset_name == 'shakespeare':
        dataset = load_dataset('sarnab/Shakespeare_Corpus')
        text_key = 'text'
    elif dataset_name == 'news':
        dataset = load_dataset('SetFit/20_newsgroups')
        text_key = 'text'
    elif dataset_name == 'wiki':
        dataset = load_dataset('wikitext', 'wikitext-2-raw-v1')
        text_key = 'text'
    elif dataset_name == 'python':
        dataset = load_dataset('mbpp')
        text_key = 'code'
    else:
        raise ValueError("Unknown dataset name. Please choose from 'imdb', 'news', or 'wikitext' or 'pythoncode'.")

    return dataset, text_key
```

BYOLLM: Tokenization

Next, we tokenize the dataset.

In this toy example, we assign a token to each letter.

Most LLMs use ‘sub-word’ tokenization.



BYOLLM: Tokenization

```python
# Just run this

# To find which tokens we will need, we can list all the unique characters in our text
chars = sorted(list(set(text)))
vocab_size = len(chars)
print(f"All unique characters: {''.join(chars)}")
print(f"Number of unique characters: \n{vocab_size}")
```

```
All unique characters:
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPRSTUVWXYZ[\]^_abcdefghijklmnopqrstuvwxyz{|}~
Number of unique characters:
96
```

BYOLLM: Tokenization

We will define each unique character to be a token. All these tokens together form the vocabulary. And by mapping each token to a token ID (an integer), we can then represent our text as a list of integers.

```python
# Just run this

# We define our vocabulary as a mapping of all unique characters to token IDs.

# Token to token ID
stoi = { ch:i for i,ch in enumerate(chars) }
# Token ID to token
itos = { i:ch for i,ch in enumerate(chars) }

# Function to tokenize text into token IDs
def tokenize(s):
    return [stoi[c] for c in s]

# Function to detokenize a list of token IDs back into a string (text)
def detokenize(l):
    return ''.join([itos[i] for i in l])

# Let's print the mapping so we can see which character corresponds to which token ID.
print(f"Mapping of characters to token IDs: {stoi}")
print(f"Mapping of token IDs to characters: {itos}")
```

```
Mapping of characters to token IDs: {'\t': 0, '\n': 1, '\r': 2, ' ': 3, '!': 4, '"': 5, '#': 6, '$': 7, '%': 8, '&': 9, "
Mapping of token IDs to characters: {0: '\t', 1: '\n', 2: '\r', 3: ' ', 4: '!', 5: '"', 6: '#', 7: '$', 8: '%', 9: '&', 1
```

BYOLLM: Embedding (vectorization)

Part 3: Vector embedding

We start with the embedding layer. Although we have already transformed our text into integers using tokenization, the numbers attributed to each token have no meaning. On top of that, if our dictionary would contain many words, our integer values would become very high.

The embedding layer gives transforms the tokenized sequences to a multi-dimensional vector that is more manageable for a deep learning network. It will also give similar (but not the same) embedding vectors to tokens that have a similar meaning.

At first, a tokenized sequence will be just a 1D array:

[sequence length]

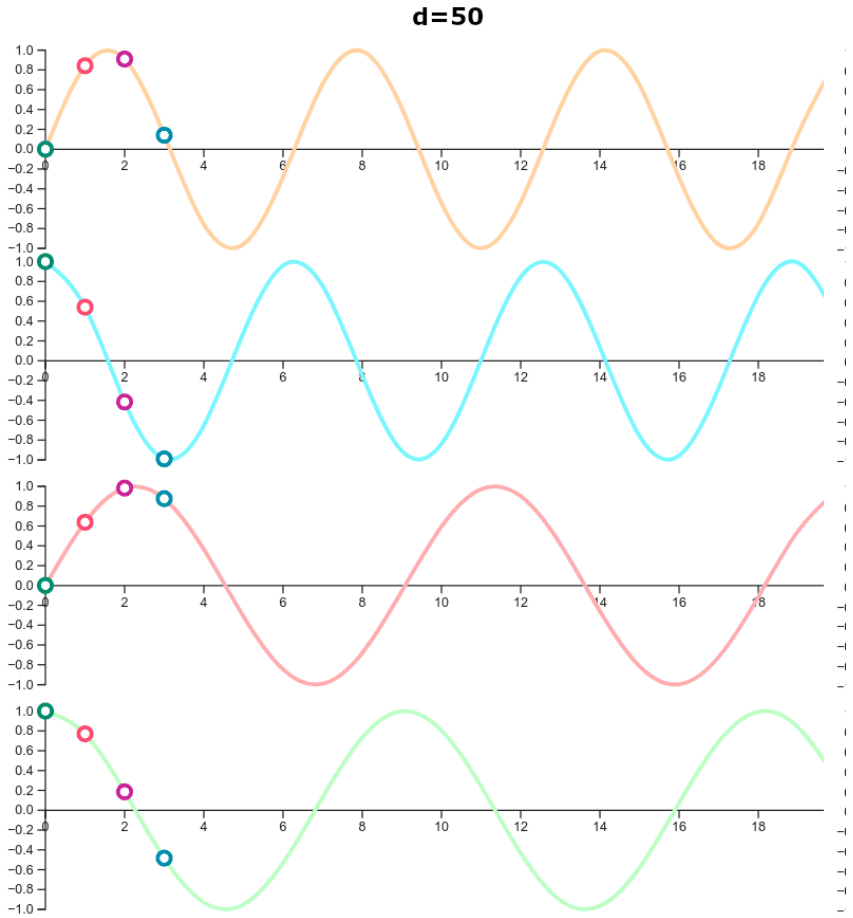
A sequence of tokenized text then turns into a 2D array after the embedding layer, with dimensions:

[sequence length x embedding dim]

Because we will process multiple sequences at once, in so-called 'batches', this will add another dimension to form:

[batches x sequence length x embedding dim]

BYOLLM: Embedding (positional encoding)



$$PE_{(p,os2i)} = \sin\left(\frac{pos}{10000^{2i/d}model}\right)$$

$$PE_{(p,os2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}model}\right)$$

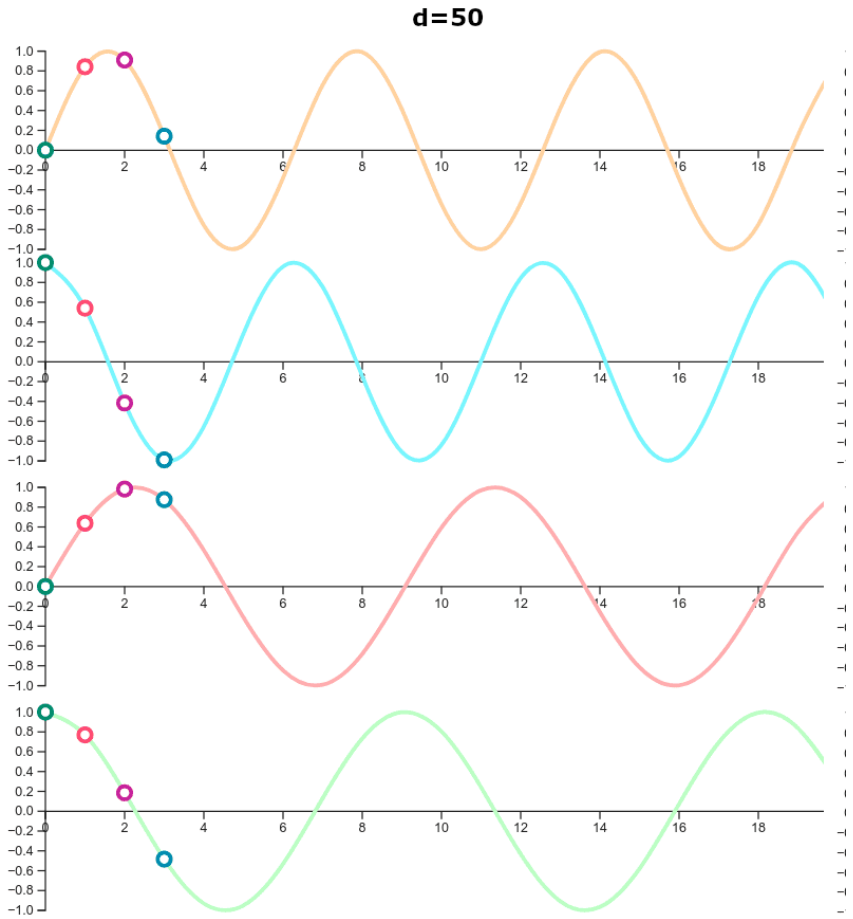
$\boldsymbol{pos}$: index of token position in the sequence (e.g., 0 for the first token, 1 for the second, and so on).

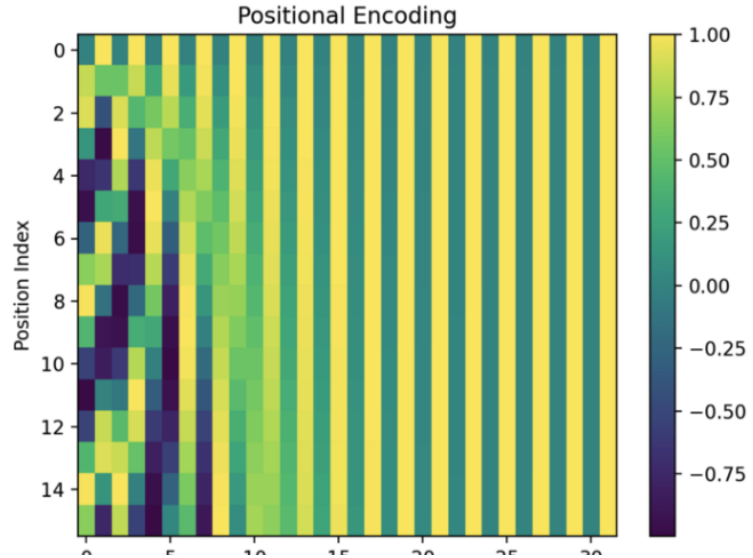
$\boldsymbol{i}$: This is the index of the dimension within the embedding vector. It ranges from 0 to $d_{model}/2 - 1$ .Notice that each pair of dimensions $(2, i2i + 1)$ uses the same frequency but different functions (sine and cosine).

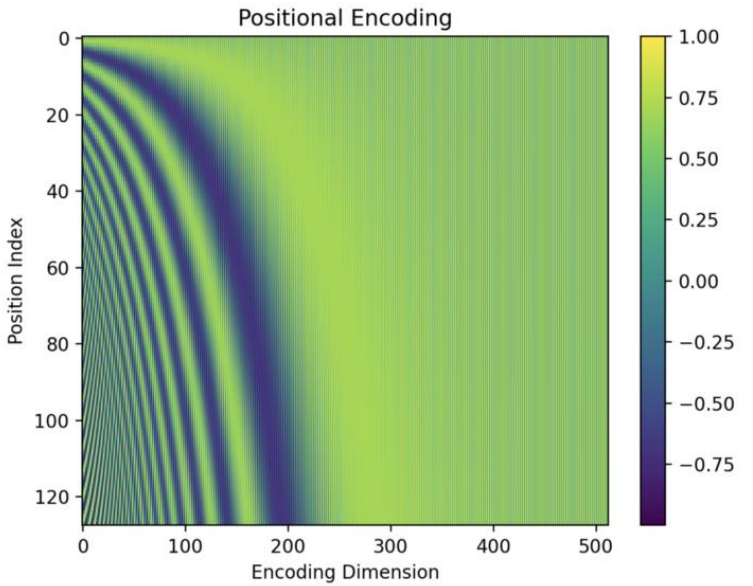
$\boldsymbol{d_{model}}$:This is the dimensionality of the input embeddings (and also the dimension of the positional encoding vector). A typical value might be 512 or 768.

$\mathbf{10000}$: This value is a hyperparameter chosen by the authors; it determines the range of frequencies used.

BYOLLM: Embedding (positional encoding)







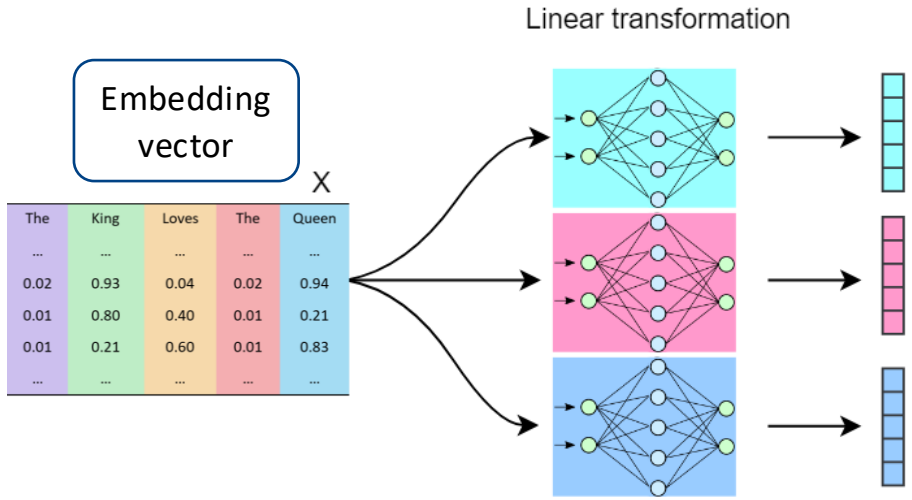
UMC Utrecht

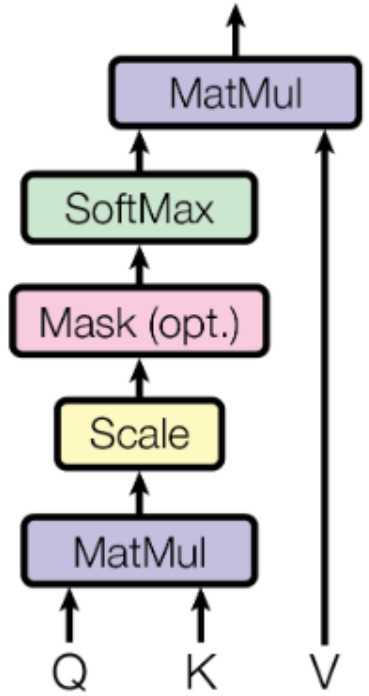
BYOLLM: Embedding (positional encoding)

```python
# Just run this

import math

class PositionalEncodingLayer(nn.Module):
    def __init__(self, embed_size, max_len=5000):
        super(PositionalEncodingLayer, self).__init__()
        self.embed_size = embed_size

        pe = torch.zeros(max_len, embed_size)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, embed_size, 2).float() * (-math.log(10000.0) / embed_size))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1), :].detach()
        return x

# Initialize the layer
pos_encoding_layer = PositionalEncodingLayer(embed_size)

# Use the layer to add position encoding to our embedded sample!
pos_encoded_sample = pos_encoding_layer(batched_embedded_sample)

# Show the output (optional) and the shape.
print(f"Positional encoded output:\n {pos_encoded_sample}")
print(f"Positional encoded sample shape:\n {pos_encoded_sample.shape}")
```

BYOLLM: Attention layers

Query-key-value





Query – **K**ey - **V**alue

BYOLLM: Attention layers

Part 4: The attention mechanism

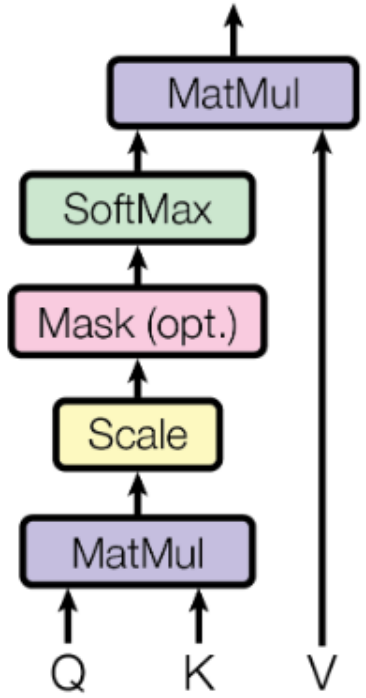
This is where the magic of transformers happens: self-attention. Attention is a way for the model to focus on different parts of the input sequence when making predictions, depending on the next word to be predicted.

In our decoder-only network, which generates text one token at a time, the attention mechanism helps the model decide which parts of the already generated tokens (the context) are important for predicting the next token.

A few key points about the attention mechanism:

1. The input embeddings are transformed into **three matrices, called the query (Q), key (K), and value (V)** matrices using a linear layer.
2. These linear layers have **trainable parameters** which are initialized randomly, but these (attention) parameters are trained during the training of the model.
3. The Q-K-V matrices are combined to produce **a context-aware representation of the input sequence**.
4. The entire process is performed in a sub-unit called the **'attention head'**.
5. **Multiple 'attention heads' are usually used in a parallel configuration** in a system called 'multi-head attention' to efficiently capture relations between tokens in the input sequence.
6. **Multiple 'multi-head attention' blocks are usually stacked in a serial configuration** to attain deeper networks.

BYOLLM: Attention layers

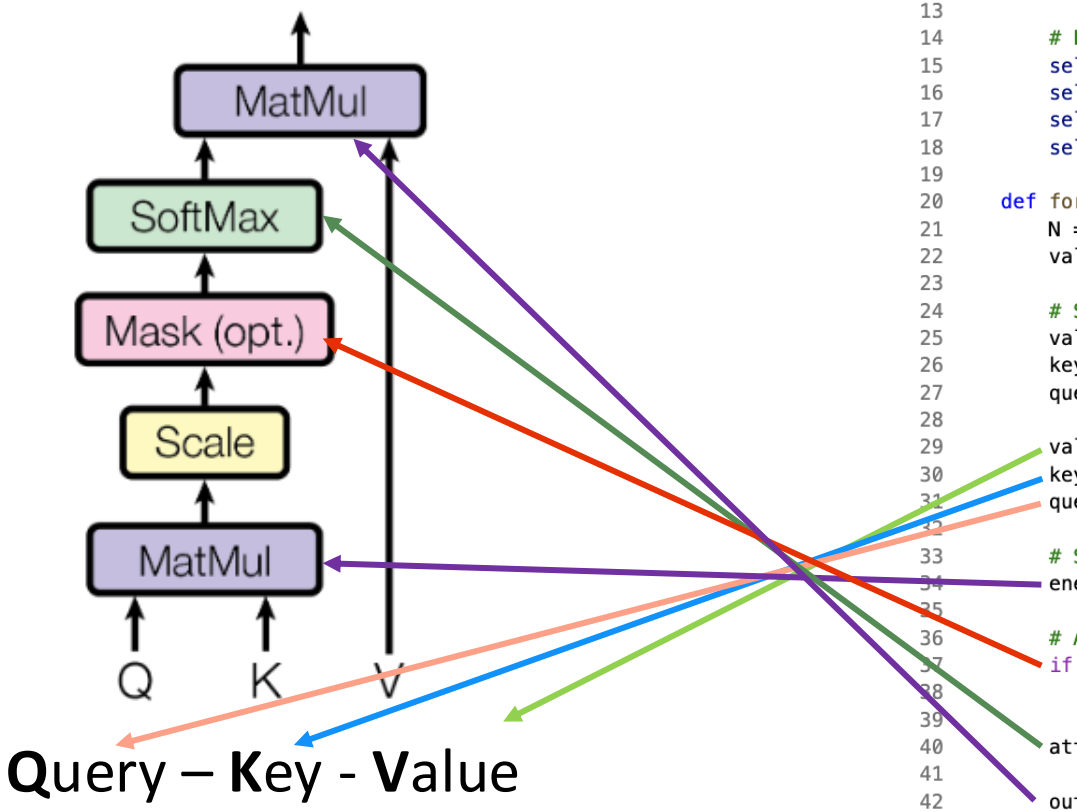


Query – **K**ey - **V**alue

```python
class MaskedAttentionLayer(nn.Module):
    def __init__(self, embed_size, heads = 4):
        super(MaskedAttentionLayer, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert (
            self.head_dim * heads == embed_size
        ), "Embedding size needs to be divisible by heads"

        # Each
        self.valuesLayer = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.keysLayer = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.queriesLayer = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.fc_out = nn.Linear(heads * self.head_dim, embed_size)

    def forward(self, values, keys, queries, mask = None):
        N = queries.shape[0]
        value_len, key_len, query_len = values.shape[1], keys.shape[1], queries.shape[1]

        # Split the embedding into self.heads different pieces
        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = queries.reshape(N, query_len, self.heads, self.head_dim)

        values = self.valuesLayer(values)
        keys = self.keysLayer(keys)
        queries = self.queriesLayer(queries)

        # Scaled dot-product attention
        energy = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

        # Apply the mask (add -1e10 to masked positions)
        if mask is not None:
            energy = energy.masked_fill(mask == 0, float('-1e10'))

        attention = torch.softmax(energy / (self.embed_size ** (1 / 2)), dim=3)

        out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(
            N, query_len, self.heads * self.head_dim
        )

        out = self.fc_out(out)
        return out
```

UMC Utrecht

BYOLLM: Attention layers



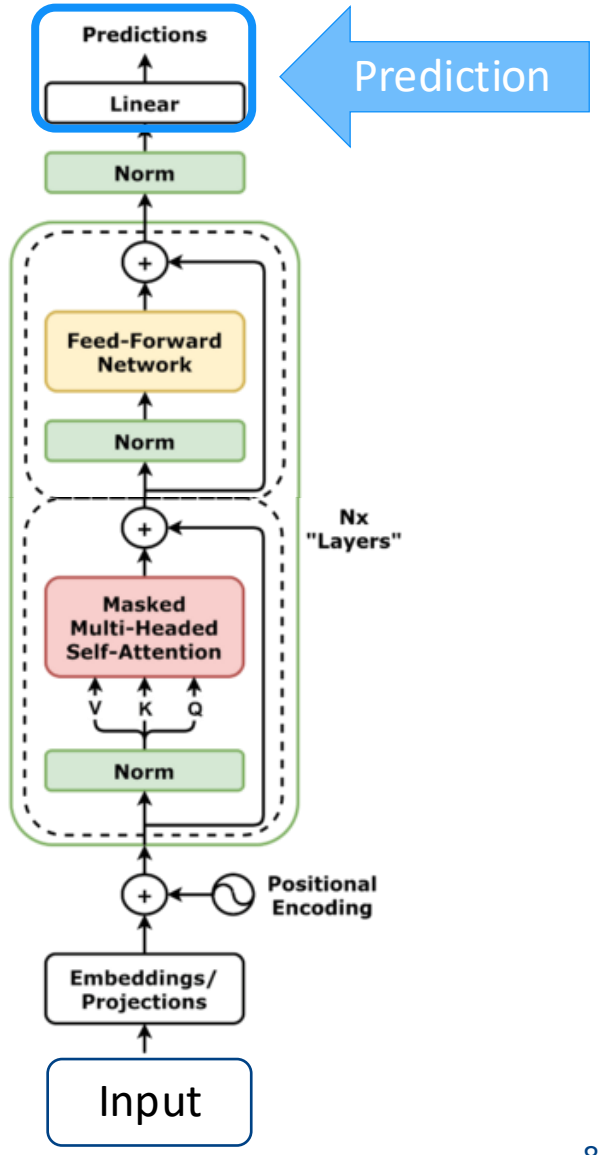

```python
class MaskedAttentionLayer(nn.Module):
    def __init__(self, embed_size, heads = 4):
        super(MaskedAttentionLayer, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert (
            self.head_dim * heads == embed_size
        ), "Embedding size needs to be divisible by heads"

        # Each
        self.valuesLayer = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.keysLayer = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.queriesLayer = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.fc_out = nn.Linear(heads * self.head_dim, embed_size)

    def forward(self, values, keys, queries, mask = None):
        N = queries.shape[0]
        value_len, key_len, query_len = values.shape[1], keys.shape[1], queries.shape[1]

        # Split the embedding into self.heads different pieces
        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = queries.reshape(N, query_len, self.heads, self.head_dim)

        values = self.valuesLayer(values)
        keys = self.keysLayer(keys)
        queries = self.queriesLayer(queries)

        # Scaled dot-product attention
        energy = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

        # Apply the mask (add -1e10 to masked positions)
        if mask is not None:
            energy = energy.masked_fill(mask == 0, float('-1e10'))

        attention = torch.softmax(energy / (self.embed_size ** (1 / 2)), dim=3)

        out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(
            N, query_len, self.heads * self.head_dim
        )

        out = self.fc_out(out)
        return out
```

UMC Utrecht

BYOLLM: Output

Final output layer:
- Transformer layers output final embedded vector.
- Linear (output) layer converts embedding to a score for each token in the vocabulary
- These scores are called logits.

Softmax function:
- Softmax turns logits into probabilities for each token in the vocabulary
- All probabilities will sum to 1.
- Token is chosen from multinomial distribution using argmax, top-p, top-k and/or temperature



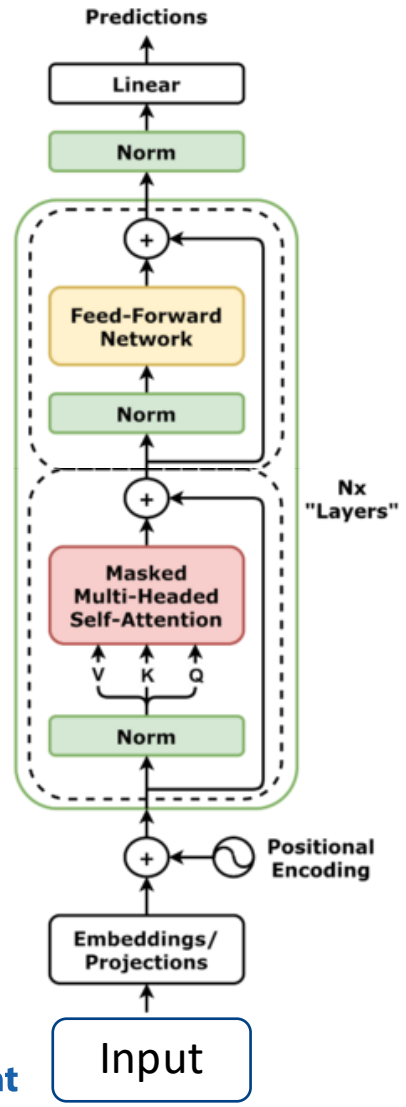
BYOLLM: Output

Final output layer:

- Transformer layers output final embedded vector.
- Linear (output) layer converts embedding to a score
  for each token in the vocabulary
- These scores are called logits.

```python
# Just run this

class OutputLayer(nn.Module):
    def __init__(self, embed_size, vocab_size):
        super(OutputLayer, self).__init__()
        self.fc_out = nn.Linear(embed_size, vocab_size)

    def forward(self, x):
        return self.fc_out(x)

# Example usage
output_layer = OutputLayer(embed_size, vocab_size)
output = output_layer(attention_sample)
print(f"Output:\n {output}")
print(f"Output shape: {output.shape}")
```

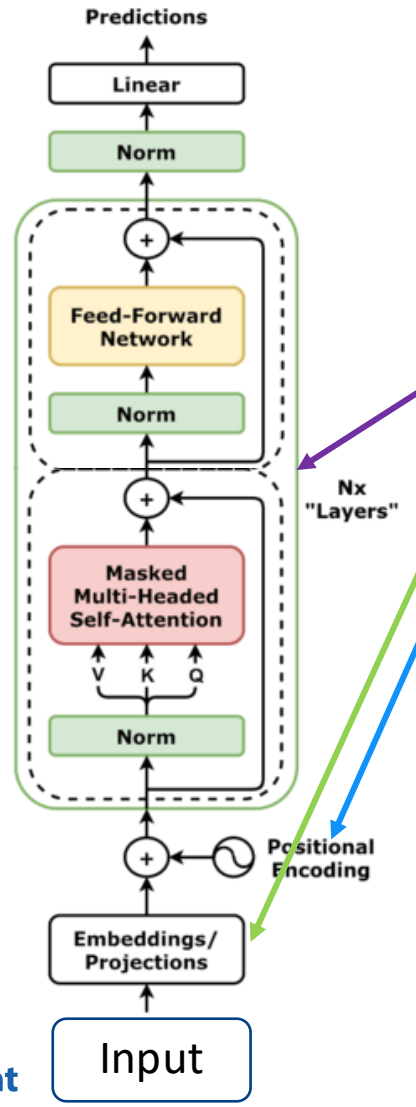
BYOLLM: Building the model




```python
class SimpleDecoderOnlyTransformerModel(nn.Module):
    def __init__(self, vocab_size, embed_size, num_heads, num_layers, context_window, max_len=5000):
        super(SimpleDecoderOnlyTransformerModel, self).__init__()

        # Initialize an embedding layer that converts token indices into dense vectors of size embed_size
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Initialize an embedding layer for positional encoding, giving a unique embedding to each position in the context window
        self.pos_encoding = PositionalEncodingLayer(embed_size, max_len)
        # Stack the transformer decoder layers (num_layers times)
        self.transformer_decoder = nn.TransformerDecoder(
            nn.TransformerDecoderLayer(embed_size, num_heads, dim_feedforward=4*embed_size, dropout=0.1, batch_first=True),
            num_layers
        )
        # A fully connected layer to map the output of the decoder to the vocabulary size
        self.fc_out = nn.Linear(embed_size, vocab_size)
        # Store the maximum sequence length
        self.max_len = max_len
        # Store the embedding size
        self.embed_size = embed_size

    def generate_square_subsequent_mask(self, size):
        # Generate a lower triangular matrix (mask) to prevent the decoder from attending to future tokens
        mask = torch.tril(torch.ones(size, size))  # Lower triangular matrix
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))  # Convert to float and mask
        return mask

    def forward(self, x):
        device = x.device

        # Trim to last context_window tokens
        x = x[:, -self.max_len:]  # (B, T)
        # Get token embeddings
        tok_emb = self.embedding(x)  # (B, T, E)
        # Apply sinusoidal positional encoding
        x = self.pos_encoding(tok_emb)  # (B, T, E)
        # Generate causal mask
        tgt_mask = self.generate_square_subsequent_mask(x.shape[1]).to(device)  # (T, T)
        # Pass through Transformer decoder
        x = self.transformer_decoder(tgt=x, memory=x, tgt_mask=tgt_mask, memory_mask=tgt_mask)  # (B, T, E)
        # Final projection to vocabulary logits
        x = self.fc_out(x)  # (B, T, V)

        return x
```

UMC Utrecht

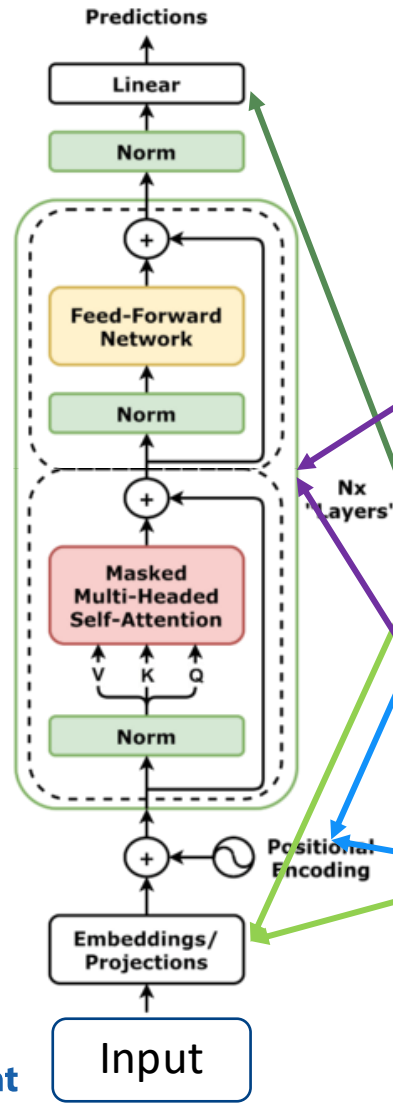
BYOLLM: Building the model




```python
class SimpleDecoderOnlyTransformerModel(nn.Module):
    def __init__(self, vocab_size, embed_size, num_heads, num_layers, context_window, max_len=5000):
        super(SimpleDecoderOnlyTransformerModel, self).__init__()

        # Initialize an embedding layer that converts token indices into dense vectors of size embed_size
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Initialize an embedding layer for positional encoding, giving a unique embedding to each position in the context window
        self.pos_encoding = PositionalEncodingLayer(embed_size, max_len)
        # Stack the transformer decoder layers (num_layers times)
        self.transformer_decoder = nn.TransformerDecoder(
            nn.TransformerDecoderLayer(embed_size, num_heads, dim_feedforward=4*embed_size, dropout=0.1, batch_first=True),
            num_layers
        )
        # A fully connected layer to map the output of the decoder to the vocabulary size
        self.fc_out = nn.Linear(embed_size, vocab_size)
        # Store the maximum sequence length
        self.max_len = max_len
        # Store the embedding size
        self.embed_size = embed_size

    def generate_square_subsequent_mask(self, size):
        # Generate a lower triangular matrix (mask) to prevent the decoder from attending to future tokens
        mask = torch.tril(torch.ones(size, size))  # Lower triangular matrix
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))  # Convert to float and mask
        return mask

    def forward(self, x):
        device = x.device

        # Trim to last context_window tokens
        x = x[:, -self.max_len:]  # (B, T)
        # Get token embeddings
        tok_emb = self.embedding(x)  # (B, T, E)
        # Apply sinusoidal positional encoding
        x = self.pos_encoding(tok_emb)  # (B, T, E)
        # Generate causal mask
        tgt_mask = self.generate_square_subsequent_mask(x.shape[1]).to(device)  # (T, T)
        # Pass through Transformer decoder
        x = self.transformer_decoder(tgt=x, memory=x, tgt_mask=tgt_mask, memory_mask=tgt_mask)  # (B, T, E)
        # Final projection to vocabulary logits
        x = self.fc_out(x)  # (B, T, V)

        return x
```

UMC Utrecht

BYOLLM: Building the model

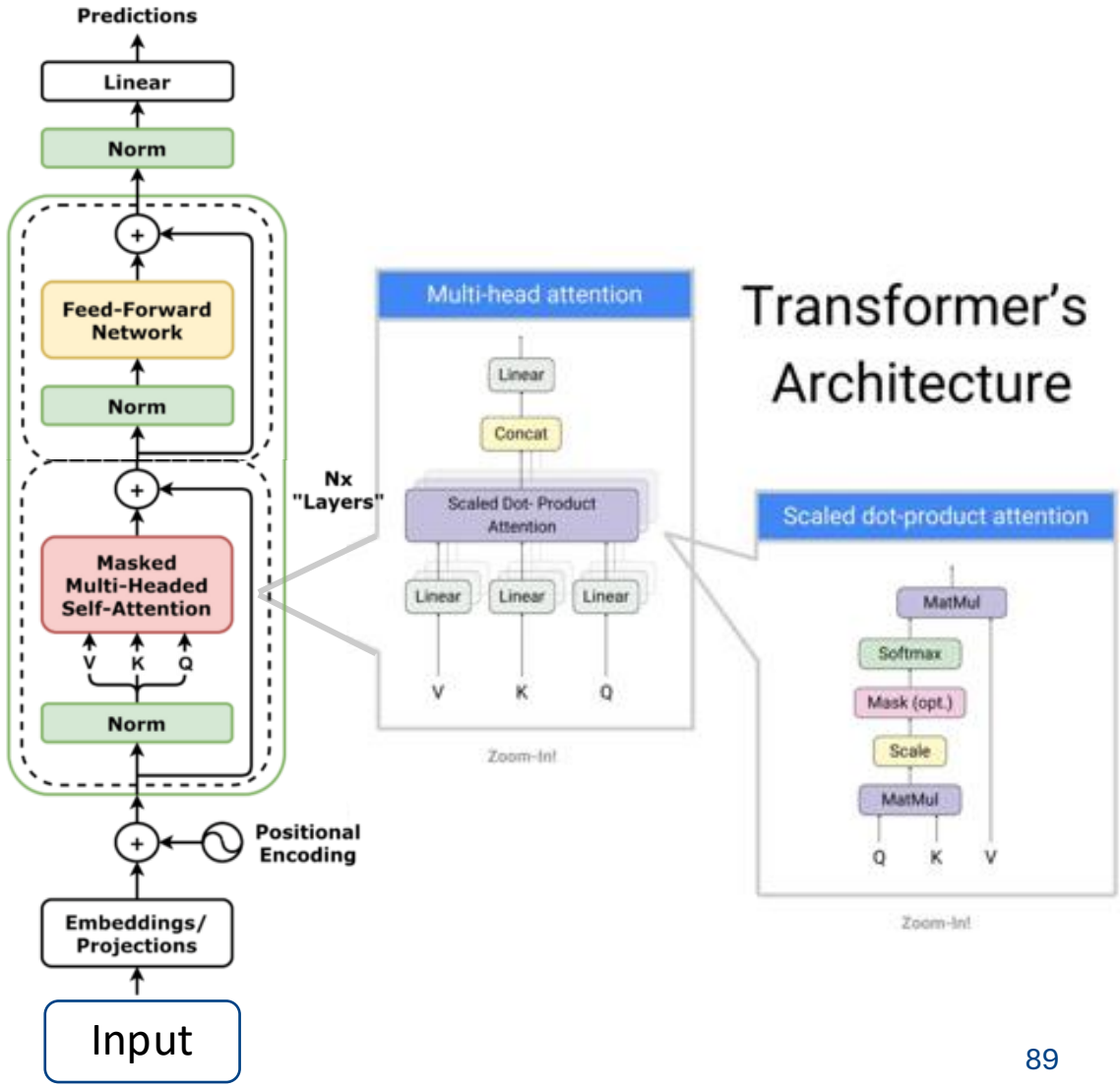



```python
class SimpleDecoderOnlyTransformerModel(nn.Module):
    def __init__(self, vocab_size, embed_size, num_heads, num_layers, context_window, max_len=5000):
        super(SimpleDecoderOnlyTransformerModel, self).__init__()

        # Initialize an embedding layer that converts token indices into dense vectors of size embed_size
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Initialize an embedding layer for positional encoding, giving a unique embedding to each position in the context window
        self.pos_encoding = PositionalEncodingLayer(embed_size, max_len)
        # Stack the transformer decoder layers (num_layers times)
        self.transformer_decoder = nn.TransformerDecoder(
            nn.TransformerDecoderLayer(embed_size, num_heads, dim_feedforward=4*embed_size, dropout=0.1, batch_first=True),
            num_layers
        )
        # A fully connected layer to map the output of the decoder to the vocabulary size
        self.fc_out = nn.Linear(embed_size, vocab_size)
        # Store the maximum sequence length
        self.max_len = max_len
        # Store the embedding size
        self.embed_size = embed_size

    def generate_square_subsequent_mask(self, size):
        # Generate a lower triangular matrix (mask) to prevent the decoder from attending to future tokens
        mask = torch.tril(torch.ones(size, size))  # Lower triangular matrix
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))  # Convert to float and mask
        return mask

    def forward(self, x):
        device = x.device

        # Trim to last context_window tokens
        x = x[:, -self.max_len:]  # (B, T)
        # Get token embeddings
        tok_emb = self.embedding(x)  # (B, T, E)
        # Apply sinusoidal positional encoding
        x = self.pos_encoding(tok_emb)  # (B, T, E)
        # Generate causal mask
        tgt_mask = self.generate_square_subsequent_mask(x.shape[1]).to(device)  # (T, T)
        # Pass through Transformer decoder
        x = self.transformer_decoder(tgt=x, memory=x, tgt_mask=tgt_mask, memory_mask=tgt_mask)  # (B, T, E)
        # Final projection to vocabulary logits
        x = self.fc_out(x)  # (B, T, V)

        return x
```

UMC Utrecht

BYOLLM: Building the model

```
# Example variables, change them around to see their effect!
vocab_size = len(stoi)
context_window = 32
embed_size = 48
num_heads = 3
num_layers = 3

# Instantiate the model
model = SimpleDecoderOnlyTransformerModel(vocab_size,
                                          embed_size,
                                          num_heads,
                                          num_layers,
                                          context_window)
# If you have a gpu, we will use that
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model.to(device)
print(device)

# Show a summary of the model
torchinfo.summary(model, depth = 4)
```



BYOLLM: Batching

Batching:

- **Group samples** for parallel processing -> more efficient training
- **Larger batches** smooth out gradients -> less noisy training

```
batch_size_dummy = 4

def generate_batch(data_split, batch_size, context_window):
    # Generate a small batch of input sequences and target sequences
    dataset = train_data if data_split == 'train' else val_data
    random_indices = torch.randint(len(dataset) - context_window, (batch_size,))
    input_sequences = torch.stack([dataset[i:i+context_window] for i in random_indices])
    target_sequences = torch.stack([dataset[i+1:i+context_window+1] for i in random_indices])
    return input_sequences, target_sequences

# A batch of data will then look like this
input_example, target_example = generate_batch('train', batch_size_dummy, context_window)
print(f"Input tokens:\n {input_example}")
print(f"Target tokens:\n {target_example}")
```

BYOLLM: Generation

Generation function

- **Prediction**:
  Model takes recent tokens as input.
- **Temperature**:
  Scales logits; higher T means more random sampling.
- **Softmax:**
  From logits to probabilities
- **Sampling**:
  Multinomial distribution is used to choose the next token. (could also be highest prob. token!)
- **Loop**:
  Process repeats until max length is reached. (could also be a stopping token!)

```python
def generate_text(model, max_length, vocab_size, start_token=None, temperature=1):

    model.eval()  # set model to evaluation mode

    # Determine starting token
    if start_token is None:
        start_token = [random.randint(0, vocab_size - 1)]  # random start if not provided
    else:
        # Convert string to token ID (if start_token is a string)
        if isinstance(start_token, str):
            start_token = tokenize(start_token)


    generated_tokens = start_token  # initialize list of generated tokens

    print(detokenize(start_token), end="", flush=True)

    for _ in range(max_length - 1):
        # Prepare input tensor of the last `context_window` tokens
        input_tensor = torch.tensor(generated_tokens[-context_window:]).unsqueeze(0)  # shape (1, seq_len)

        with torch.no_grad():  # no gradient needed for inference
            output = model(input_tensor.to(device))  # forward pass

        # Get logits for the next token (last position)
        next_token_logits = output[0, -1, :] / temperature
        # also applied temperature scaling (higher temperature is more random)

        # Convert logits to probabilities
        next_token_probs = F.softmax(next_token_logits, dim=-1)

        # Sample the next token from the probability distribution
        next_token = torch.multinomial(next_token_probs, num_samples=1).item()

        # Add the new token to the sequence
        generated_tokens.append(next_token)

        # Print the new token immediately (streaming output)
        print(detokenize([next_token]), end="", flush=True)

    print()  # final newline after finishing generation

    return generated_tokens
```

BYOLLM: Training

Transformer Training: Settings and Loop

- **Hyperparameters:** Learning rate and batch size are important for convergence.
- **Optimizer:** Adam algorithm adjusts model learn rate according to individual parameters gradients.
- **Loss Function:** CrossEntropy calculates the prediction error.
- **Update model:** Optimizer uses loss to calculate gradients and update model.
- **Evaluation:** Validation loss checks performance on unseen data.

```python
# Hyperparameters
num_epochs = 10
learning_rate = 0.0001
batch_size = 32
max_num_its_per_epoch = 300

# Initialize model, loss, and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
train_losses = []
val_losses = []

# Training and validation functions
def run_epoch(model, data, is_training):
    epoch_loss = []
    model.train() if is_training else model.eval()

    num_its_per_epoch = min(max_num_its_per_epoch, len(data) // (batch_size*context_window))
    iterator = tqdm(range(num_its_per_epoch), desc="Training" if is_training else "Validating")

    for i in iterator:
        inputs, targets = generate_batch('train' if is_training else 'validate', batch_size, context_window)
        outputs = model(inputs.to(device))

        B, T, V = outputs.shape
        outputs = outputs.view(B*T, V)
        targets = targets.view(B*T)
        loss = criterion(outputs, targets.to(device))
        epoch_loss.append(loss.item())

        if is_training:
            optimizer.zero_grad(set_to_none=True)
            loss.backward()
            optimizer.step()

        iterator.set_postfix({'loss': np.mean(epoch_loss)})

    return np.mean(epoch_loss)

# Training loop
for epoch in range(num_epochs):
    train_loss = run_epoch(model, train_data, is_training=True)
    val_loss = run_epoch(model, val_data, is_training=False)

    train_losses.append(train_loss)
    val_losses.append(val_loss)

    print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}')
    print(f'Quick example of current text output:')
    generate_text(model, max_length, vocab_size, temperature = 1, start_token = 'The')
    print()
```

Practical: BYOLLM

Now it's your turn!

You can find instructions and questions on Canvas

CHALLENGES:

- **Try running your model using the GPU for a major speed-up!**
- **Get the best validation loss on a dataset by tuning hyperparameters (compare with colleagues!)**
- **Make your model generate deterministically**
- **Try both character and sub-word tokenization (what works best?)**
- **With sub-word tokenization, try one of the new, larger datasets!**
- **Find, load and train your own dataset from Huggingface**

LINK TO NOTEBOOK (subword tokenization)