Google Classroom Code: mhxgl24

Transformer Architecture, Language Models

Deep Learning (DS-5006)

Dr. Adeel Mumtaz

Lecture 11

Fall, 2022

National University
Of Computer and Emerging Sciences

Contents

Transformer Models

NIPS 2017

ATTENTION IS ALL YOU NEED

Google Brain

Ashish Vaswani[*] Google Brain avaswani@google.com

Noam Shazeer[*] Google Brain noam@google.com

Niki Parmar[*] Google Research nikip@google.com

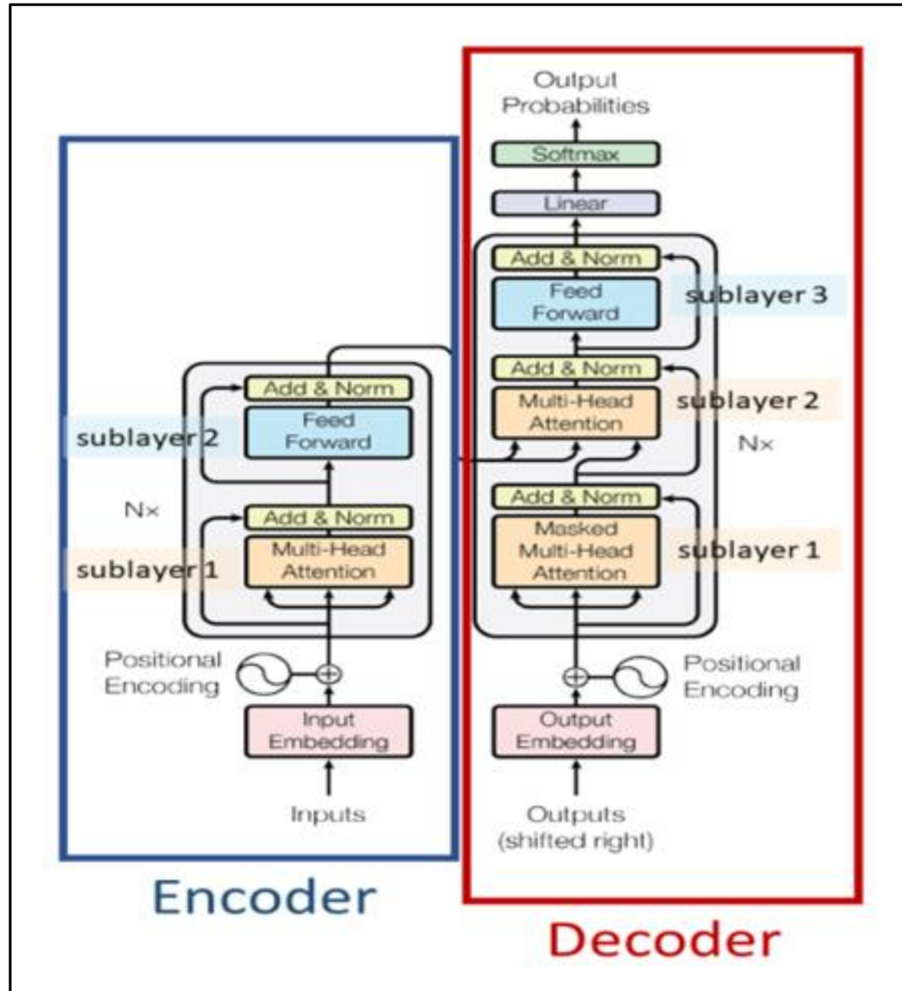
Jakob Uszkoreit[*] Google Research usz@google.com

Llion Jones[*] Google Research llion@google.com

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

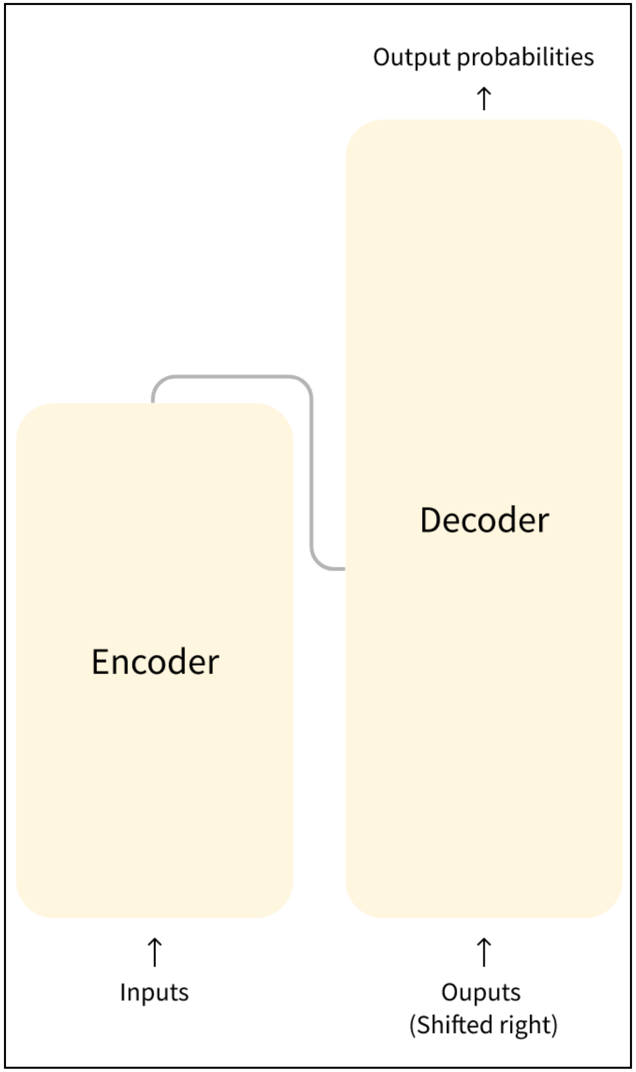
Łukasz Kaiser[*] Google Brain lukaszkaiser@google.com

Illia Polosukhin[* ‡] illia.polosukhin@gmail.com



Transformer Architecture

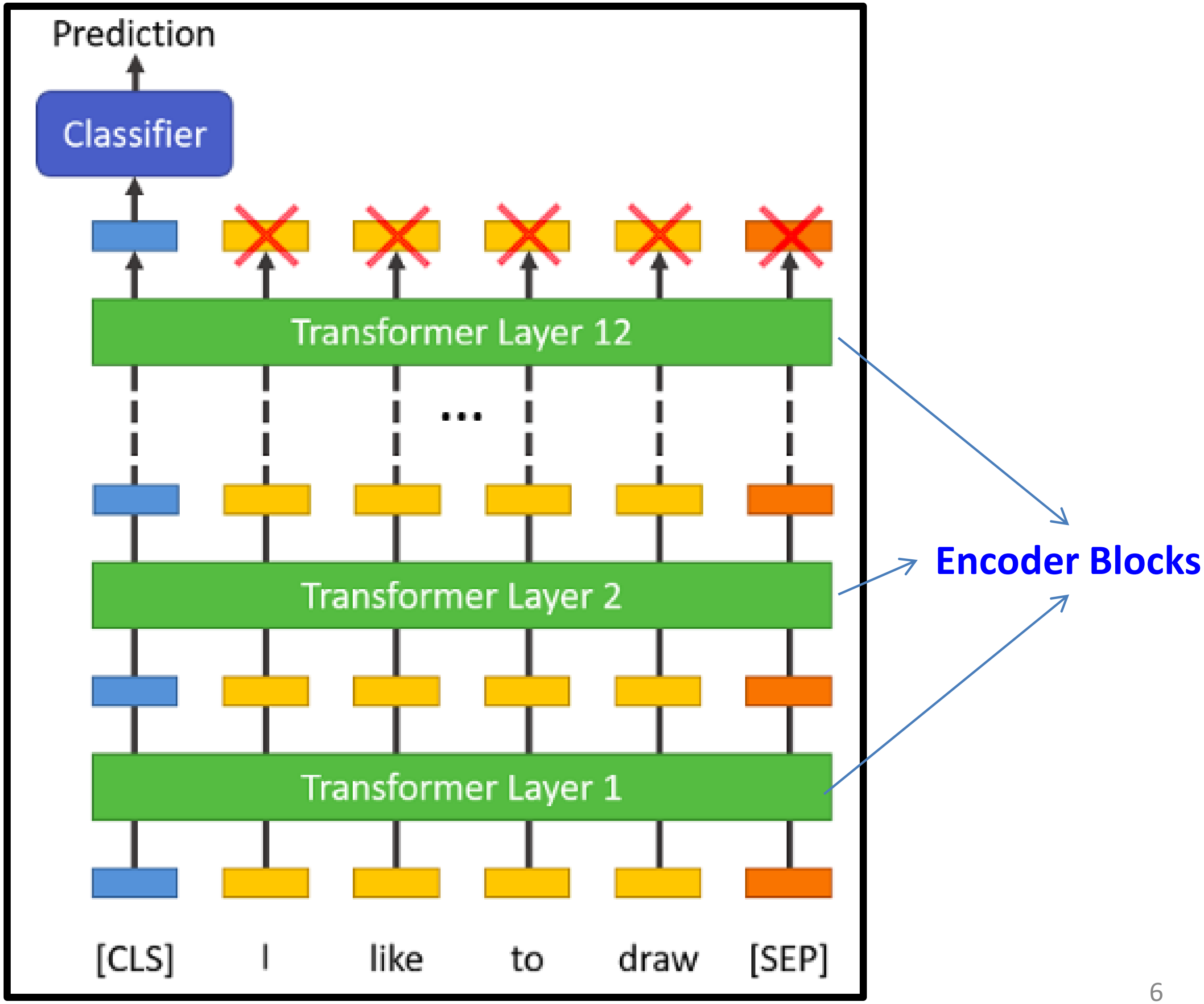
- Two blocks:
  - **Encoder (left):** The encoder receives an input and builds a representation of it (its **features**).
    - Optimized to acquire understanding from the input.
  - **Decoder (right):** The decoder uses the encoder's representation (features) along with other inputs to generate a **target sequence**.
    - Optimized for generating outputs.



Encoder-only models

- Encoder models use only the encoder of a Transformer model
- Attention layers can access all the words in the sentence called "**bi-directional**" attention
- Also called **auto-encoding models**.
- Used for tasks that require understanding of the input
  - Sentence **classification**
  - Named entity recognition
- Family of encoder models
  - ALBERT, **BERT**, DistilBERT, ELECTRA, RoBERTa

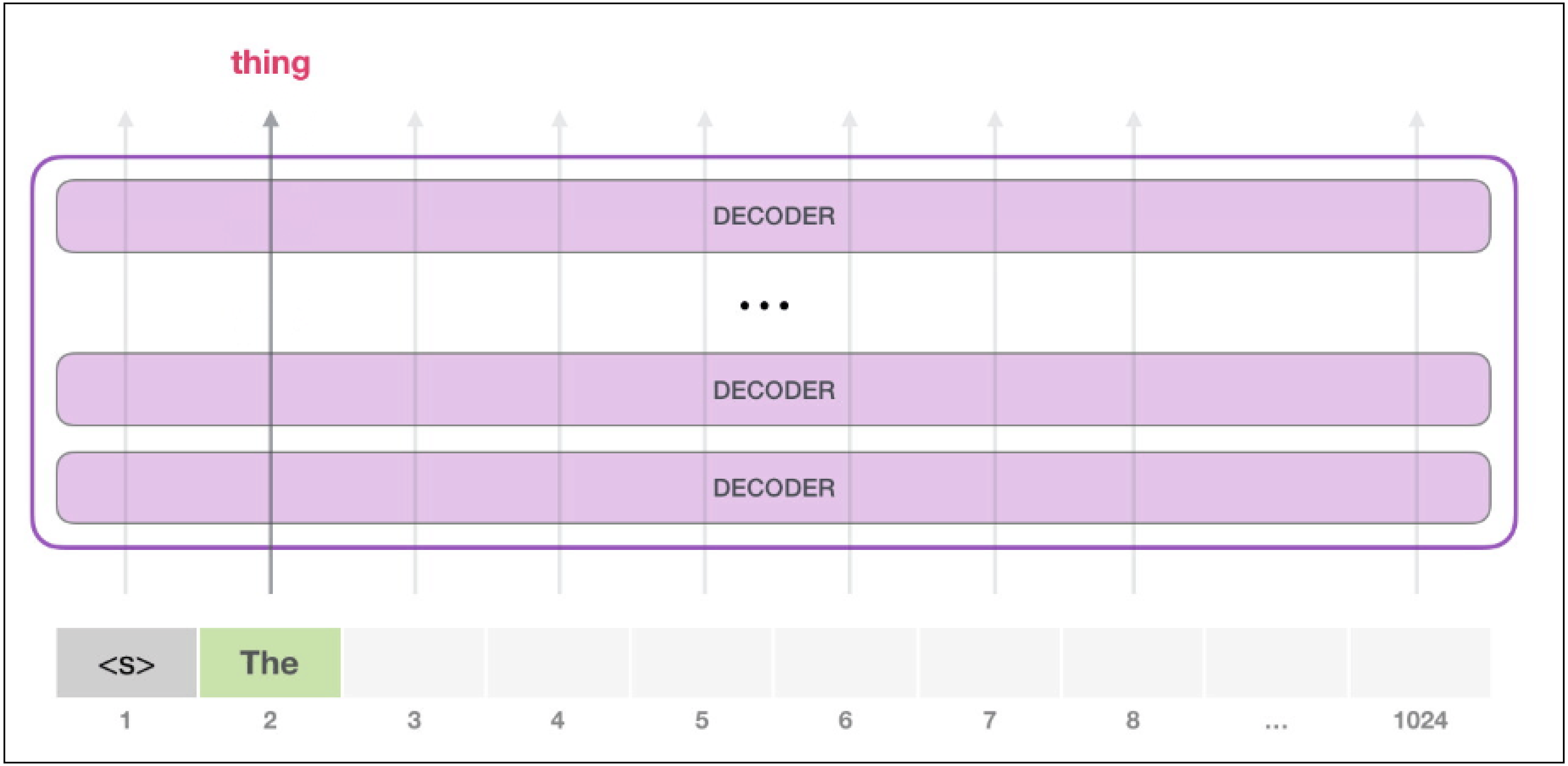
Encoder-only models



Decoder-only models

- Decoder models use **only the decoder** of a Transformer model.
- For each given word the attention layers can only access the **words positioned before** it in the sentence.
- These models are often called auto-regressive models.
- Also called **Language Models**
- Used for tasks involving text generation.
  - **Predicting the next word in the sentence**.
- Representatives of this family of models include:
  - CTRL, GPT, GPT-2, Transformer XL

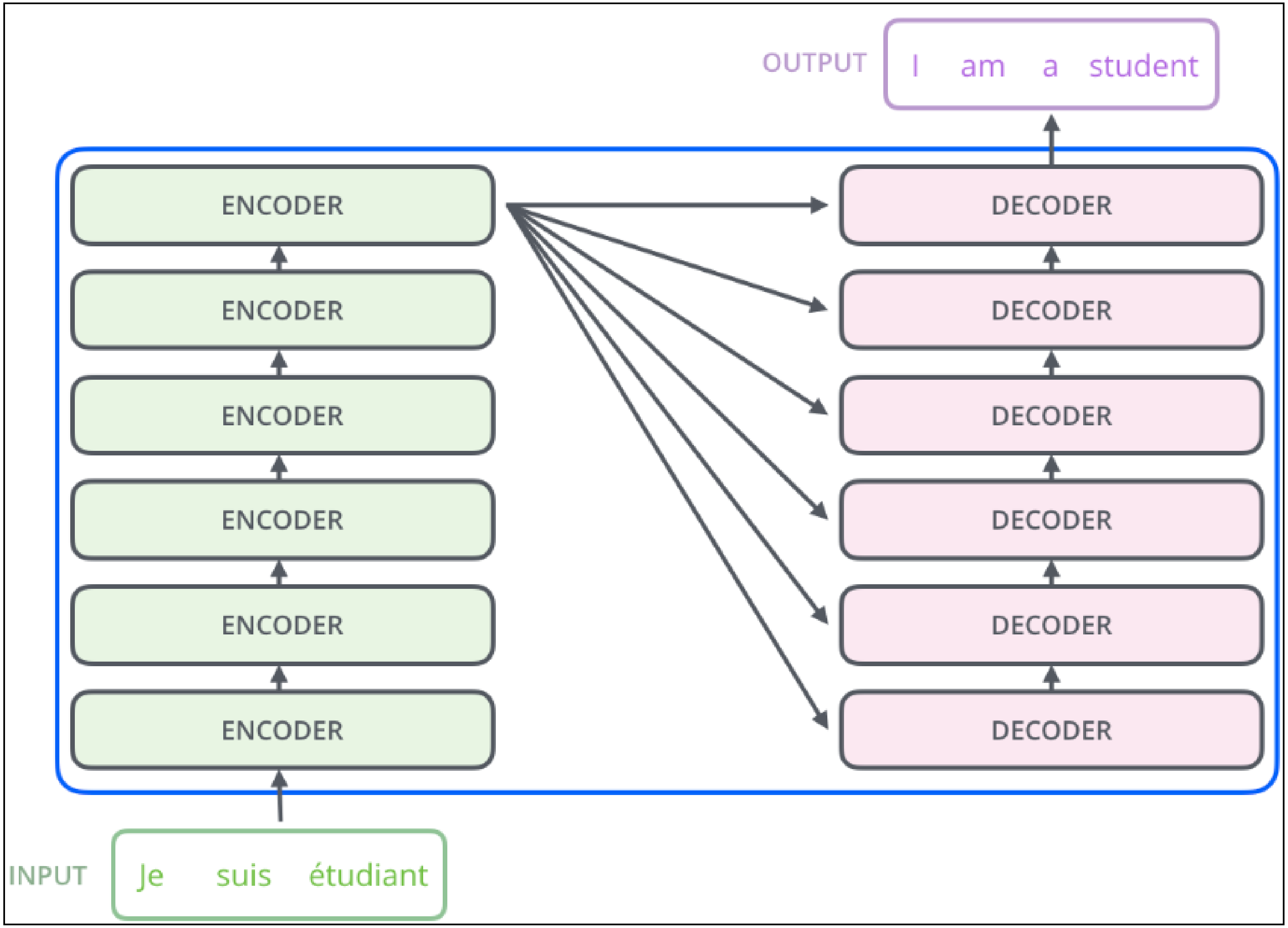
Decoder-only models



Sequence-to-Sequence Models

- **Encoder-decoder** models (also called sequence-to-sequence models) use both parts of the Transformer architecture.
- Sequence-to-sequence models are best suited for tasks revolving around **generating new sentences depending on a given input**
  - Summarization
  - Translation
  - Generative question answering.
- Representatives of this family of models include:
  - BART
  - mBART
  - Marian
  - T5

Sequence-to-Sequence Models



Transformer Types Summary

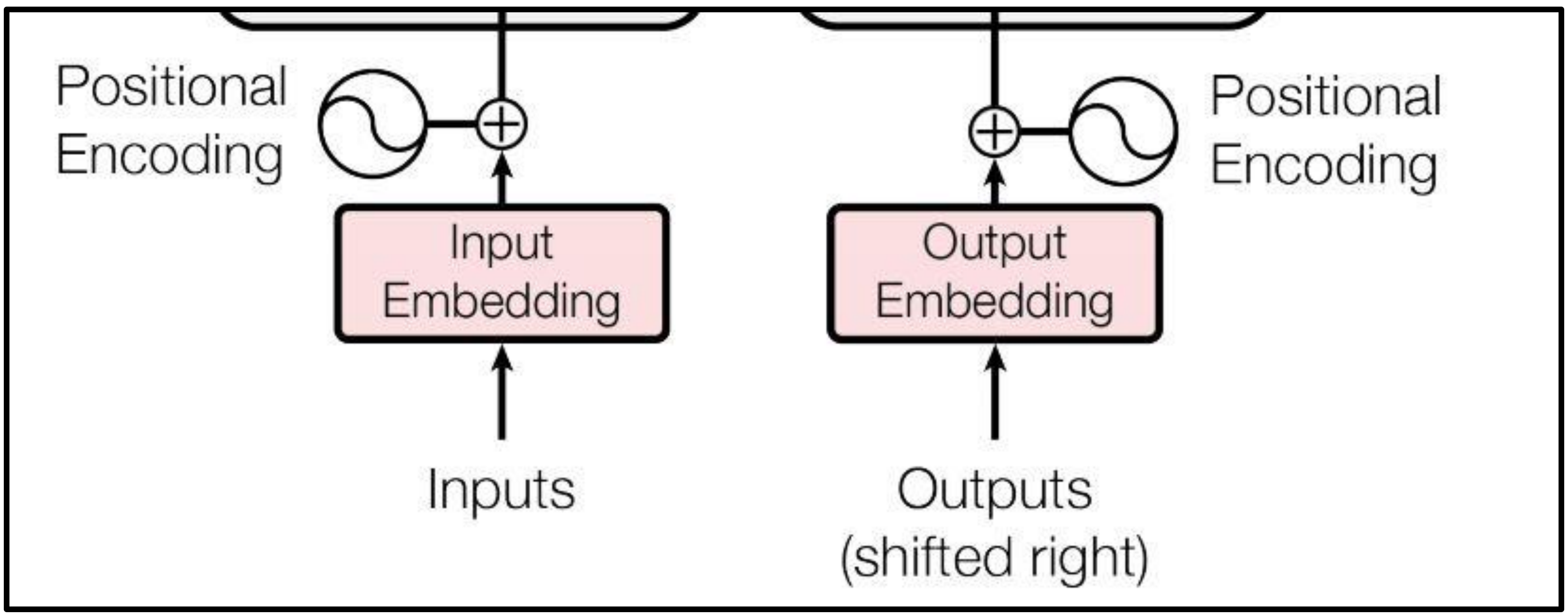
Model	Examples	Tasks
Encoder	ALBERT, BERT, DistilBERT, ELECTRA, RoBERTa	Sentence classification, named entity recognition, extractive question answering
Decoder	CTRL, GPT, GPT-2, Transformer XL	Text generation
Encoder-decoder	BART, T5, Marian, mBART	Summarization, translation, generative question answering

BREAKING DOWN TRANSFORMER ARCHITECTURE

TOKEN EMBEDDING AND POSITIONAL ENCODINGS

Token Embedding and Positional encodings

- All types of transformer model pass the sequence input ids from Token Embedding and Positional encodings layers at start
  - **BxLx1 to BxLxF**



Token Embedding Methods

- The Big Idea of Token/Word Embedding is to turn text into numbers
- Machine learning algorithms (including deep nets) require their input to be vectors of continuous values
- Objectives
  - **Dimensionality Reduction** —most efficient representation
  - **Contextual Similarity** — most expressive representation

1. One-hot Encoding

- Generate a vector for each token with length equal to size of vocabulary

	Restaurant Reviews
R1	Great restaurant and great service !
R2	They can do better to provide better service
R3	Only two thumbs up, worst service ever

Entire Corpus

Set of all the words in the corpus
great
restaurant
and
service
they
can
do
better
to
provide
only
Two
thumbs
up
worst
ever

1. One-hot Encoding

- Very large & sparse vectors
- Order & frequency information of words is lost

Set of all the words in the corpus	**R1:** Great Restaurant and great service !	**R2:** They can do better to provide better service	**R3:** Only two thumbs up, worst service ever
great	1	0	0
restaurant	1	0	0
and	1	0	0
service	1	1	0
they	0	1	0
can	0	1	0
do	0	1	0
better	0	1	0
to	0	1	0
provide	0	1	0
only	0	0	1
Two	0	0	1
thumbs	0	0	1
up	0	0	1
worst	0	0	1
ever	0	0	1

2. Frequency Encoding

- Count number of times a word appear in corpus

Restaurant Reviews
Great restaurant and great service!
They can do better to provide better service
Only two thumbs up, worst service ever

Sequence of words
(''Great'', ''restaurant'', ''and'', ''great'', ''service'')
(''They'', ''can'', ''do'', '' better'', ''to'', ''provide'', 'better'', ''service'')
(''Only'', ''two'', ''thumbs'', ''up'', ''worst'', ''service'', ''ever'')

2. Frequency Encoding

- Still large (can only keep the top-n words based on frequencies)
- Still No context/Semantic capturing of the words

Set of all the words in the corpus	**R1:** Great Restaurant and great service !	**R2:** They can do better to provide better service	**R3:** Only two thumbs up, worst service ever
great	2	0	0
restaurant	1	0	0
and	1	0	0
service	1	1	0
they	0	1	0
can	0	1	0
do	0	1	0
better	0	2	0
to	0	1	0
provide	0	1	0
only	0	0	1
Two	0	0	1
thumbs	0	0	1
up	0	0	1
worst	0	0	1
ever	0	0	1

3. TF-IDF Embedding

- Widely used in the search technologies

```
corpus = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?',
]
```

Term Frequency Table for each document

and	document	first	is	one	second	the	third	this
0	1	1	1	0	0	1	0	1
0	1	0	1	0	2	1	0	1
1	0	0	0	1	0	1	1	0
0	1	1	1	0	0	1	0	1

Inverse document Frequency Table for each token

$$\mathrm{idf}(t) = \log \frac{m}{\mathrm{df}(t)} + 1$$

	and	document	first	is	one	second	the	third	this
df(t)	1	3	2	3	1	1	4	1	3
idf(t)	2.39	1.29	1.69	1.29	2.39	2.39	1.00	2.39	1.29

3. TF-IDF Embedding

- Still Large equal to vocab size
- First & last same vector

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t)$$

Tf-IDF table for each document

and	document	first	is	one	second	the	third	this
0.00	1.29	1.69	1.29	0.00	0.00	1.00	0.00	1.29
0.00	1.29	0.00	1.29	0.00	4.77	1.00	0.00	1.29
2.39	0.00	0.00	0.00	2.39	0.00	1.00	2.39	0.00
0.00	1.29	1.69	1.29	0.00	0.00	1.00	0.00	1.29

Norm: `2.97, 5.36, 4.25, 2.97`

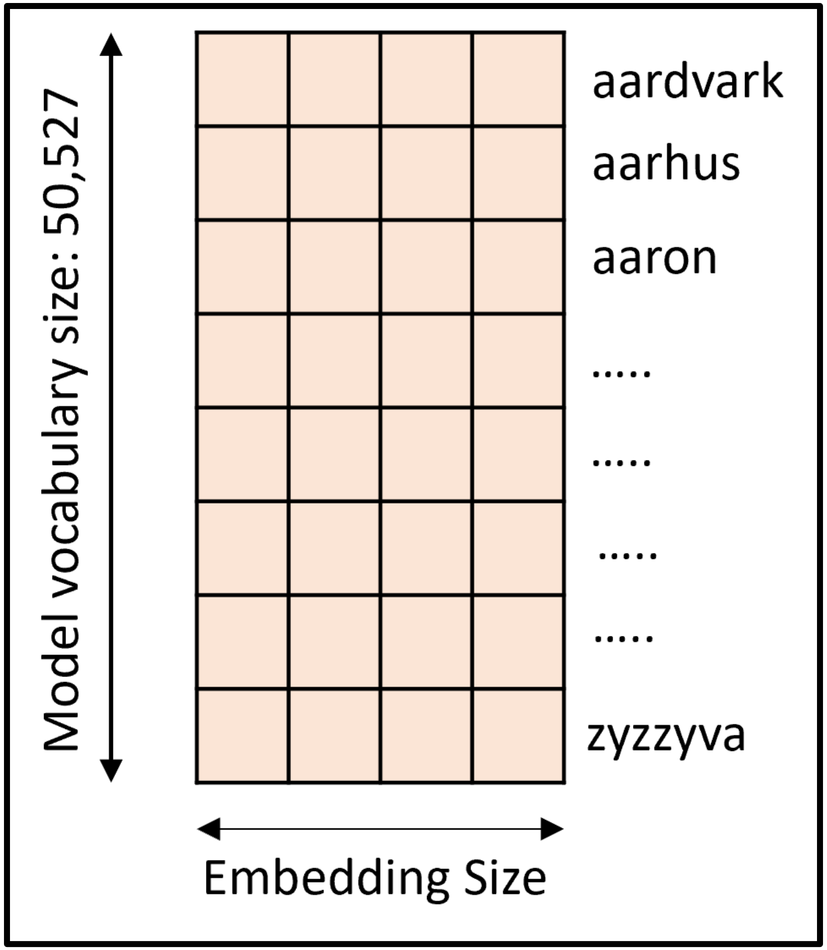
$$v_{\text{norm}} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{{v_1}^2 + {v_2}^2 + \cdots + {v_{\text{vocab\_size}}}^2}}$$

Normalized Tf-IDF table for each document

and	document	first	is	one	second	the	third	this
0.00	0.43	0.57	0.43	0.00	0.00	0.34	0.00	0.43
0.00	0.24	0.00	0.24	0.00	0.89	0.19	0.00	0.24
0.56	0.00	0.00	0.00	0.56	0.00	0.24	0.56	0.00
0.00	0.43	0.57	0.43	0.00	0.00	0.34	0.00	0.43

4. Word2Vec

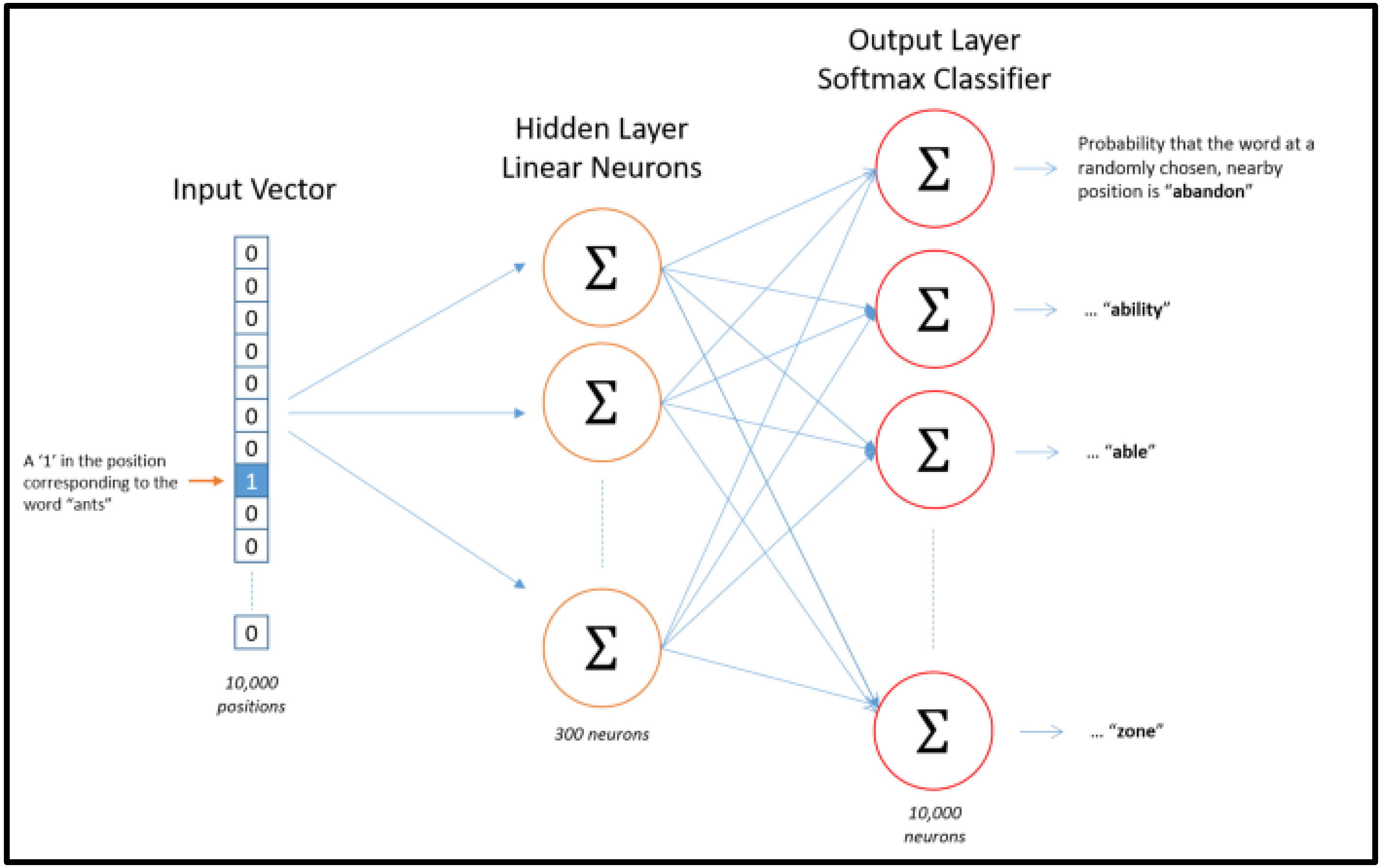
- Each token becomes a vector with
  - the length (typically 100–1000) called **embed_dim**



Output of word2vec

4. Word2Vec

- Train a ANN to learn context of similar words by using a window around the center word
- One hidden layer with size equal to embed_dim



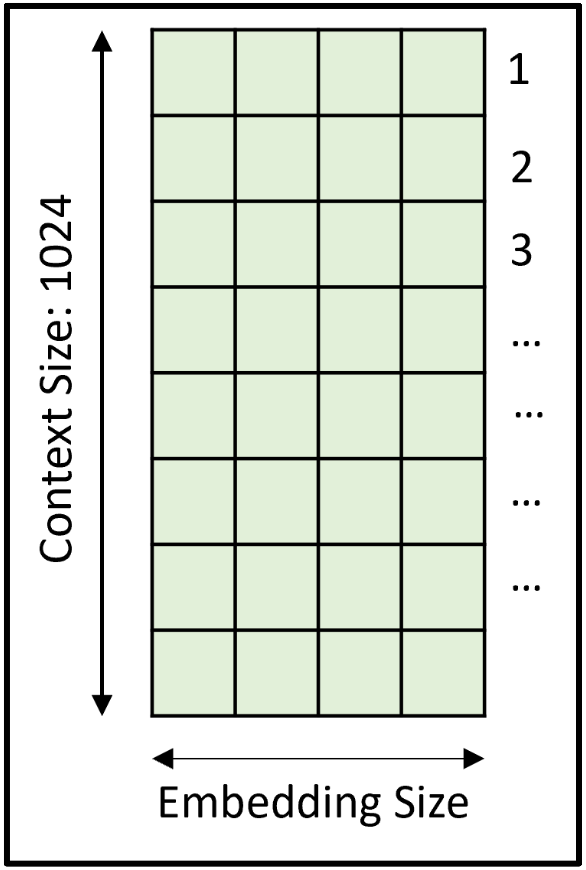
4. Word2Vec

- Use one hot vectors of center word as input and neighbor words as output
- After training Weight matrix will be the embedding matrix

Input text	Train data (window = 1)
This is the first document.	• (this, is)
This is the first document.	• (is, this) • (is, the)
This is the first document.	• (the, is) • (the, first)
...	...

Positional Encoding

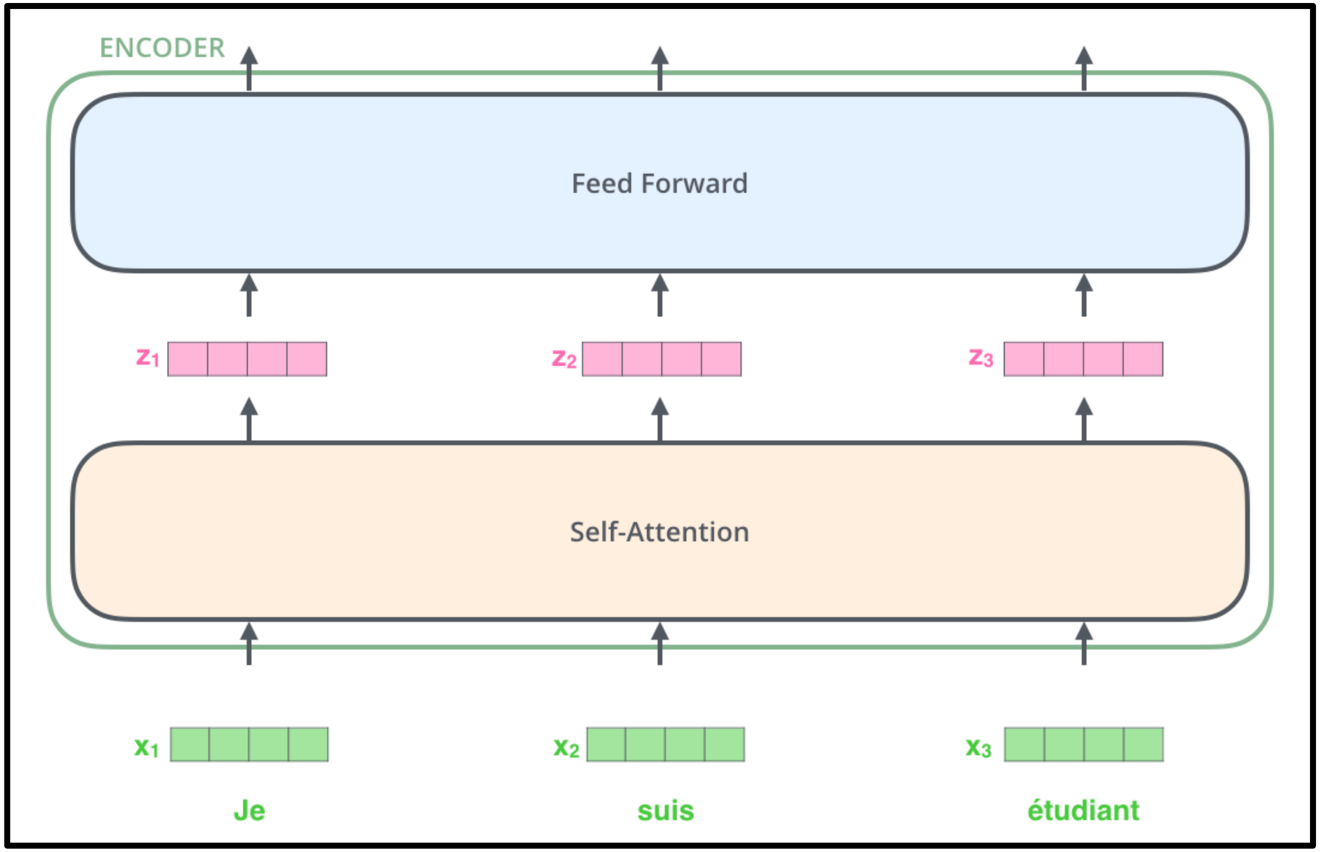
- Incorporates into each word its **positional information** in the input sequence.
- This positional information represents the order of the tokens in the sequence provided as input to the blocks of the transformer
- The **nth encoding is added to the embedding of the word present at the nth position.**



ENCODER BLOCK

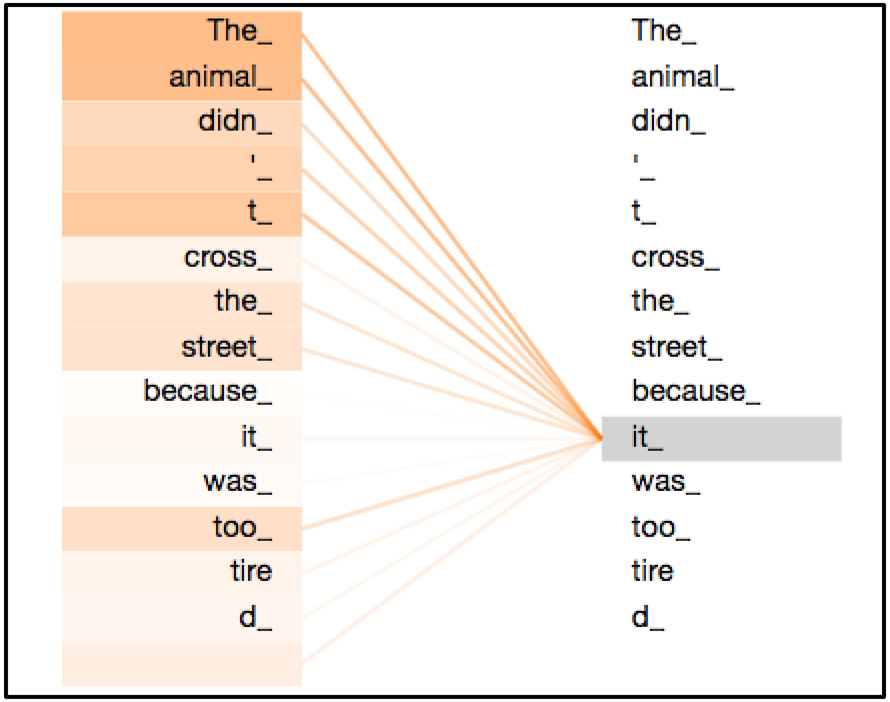
A single Encode Block

- Remember each block has its own weights
- **Hidden state, Context vectors, Sequence Length**



Self Attention Layer

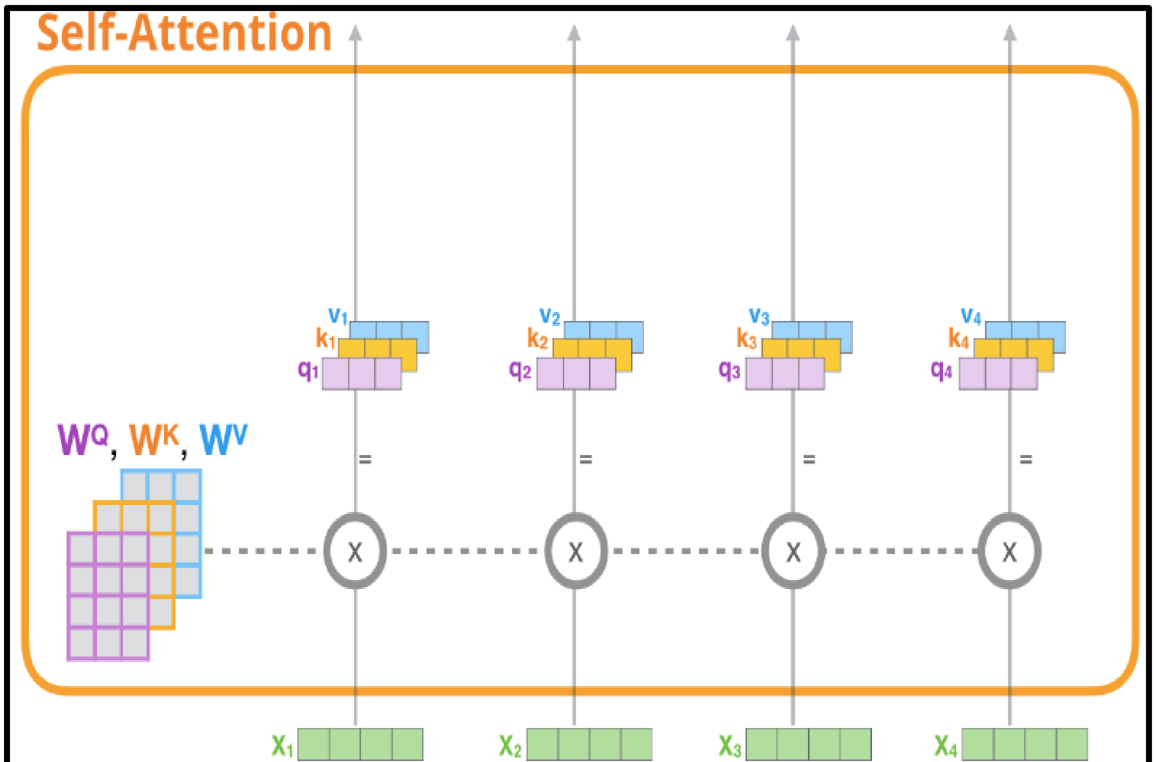
- As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

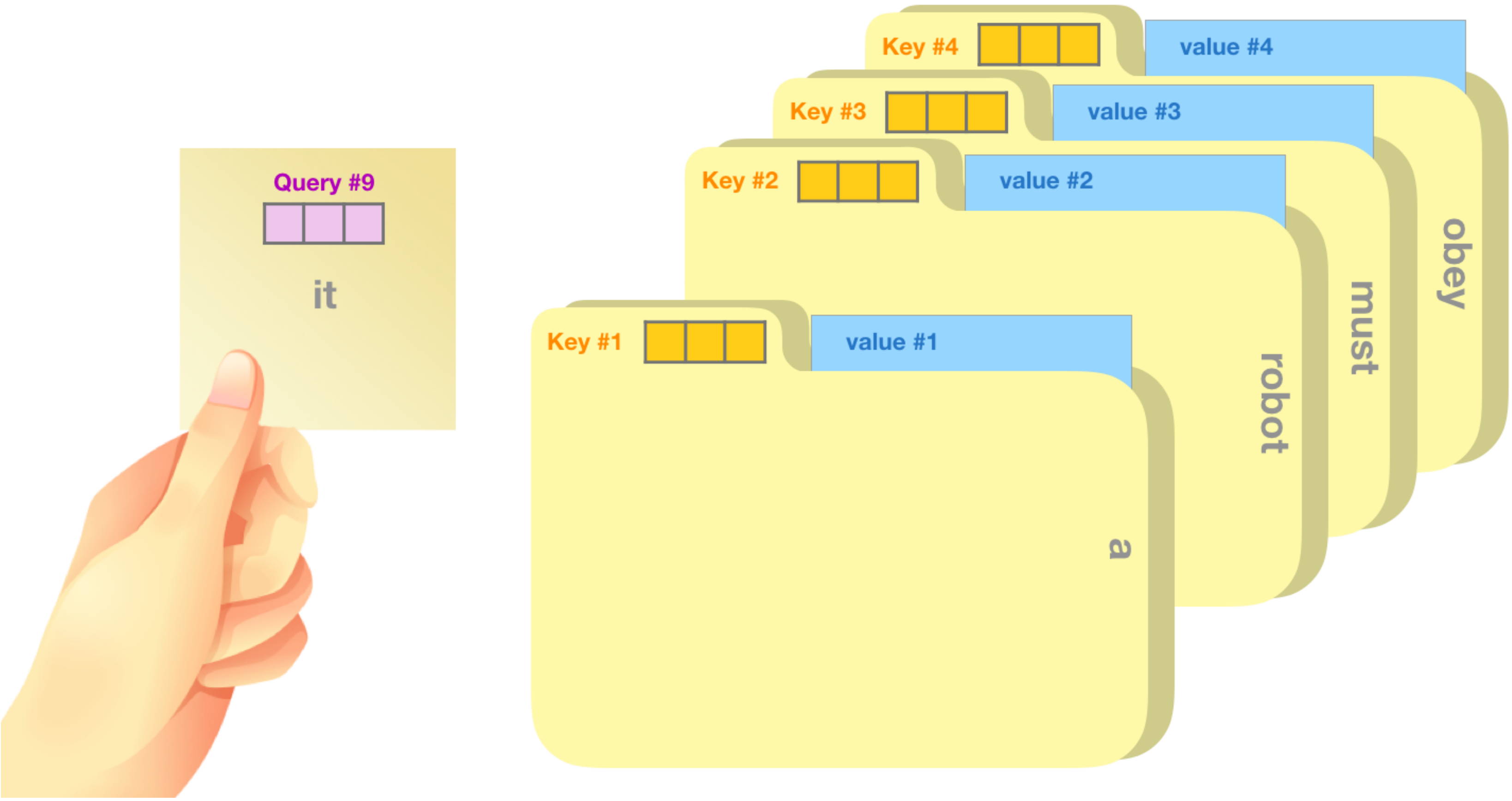


”The animal didn't coss the street because it was too tired”

Self Attention

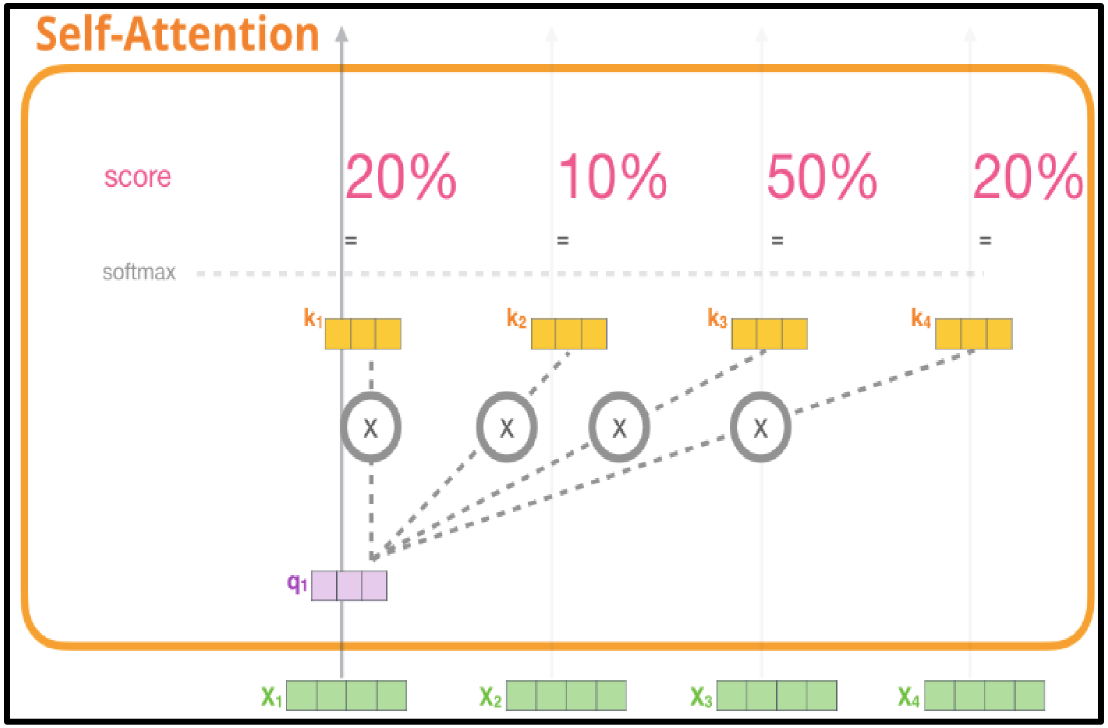
1. The **query, key and value** vectors are created through feed forward neural networks **For each token**

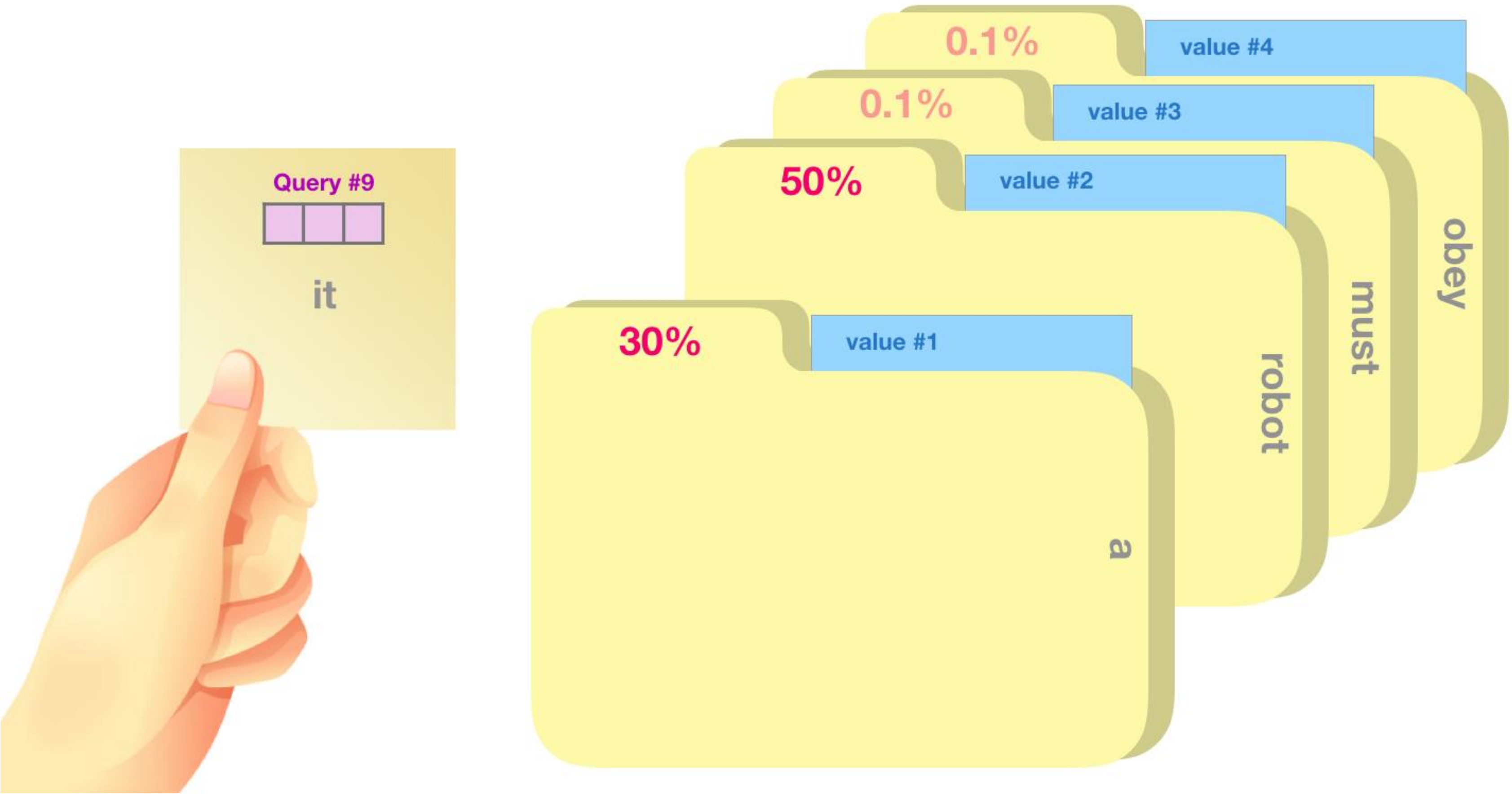



Query #9
it
Key #4
value #4
obey
Key #3
value #3
must
Key #2
value #2
robot
Key #1
value #1
a

Self Attention

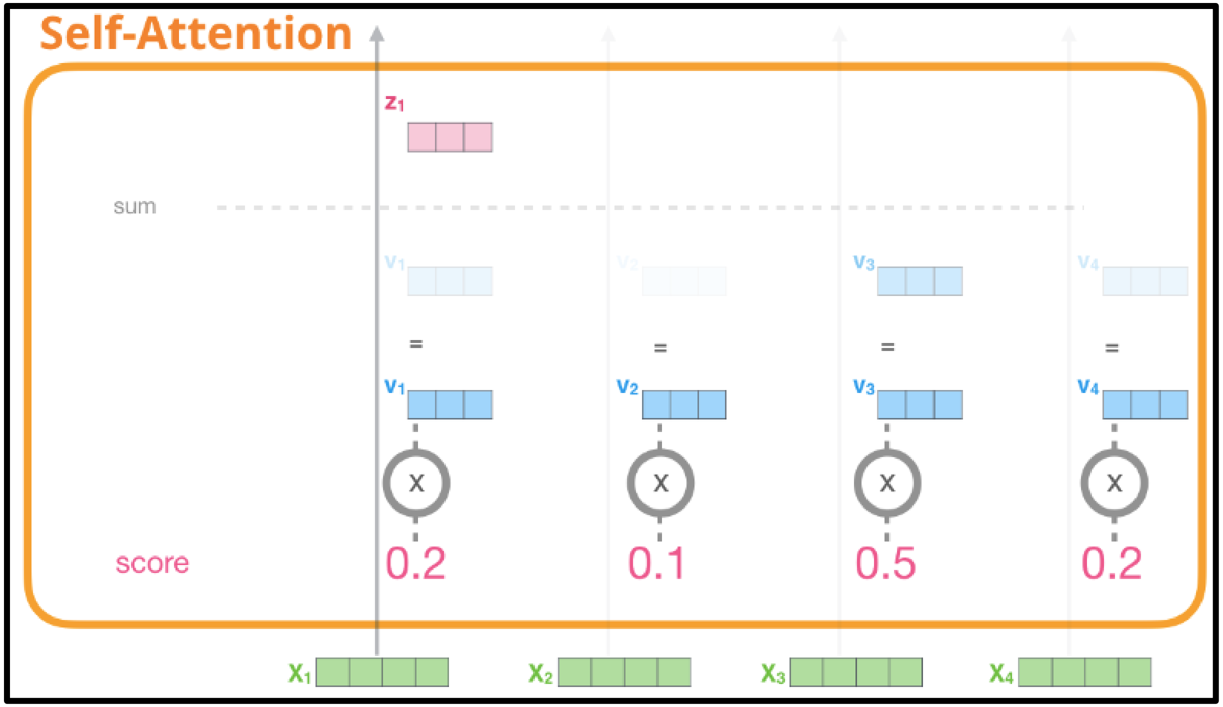
2. The current query vector is **dot multiplied** with all other key vectors to get the **attention scores (using a Softmax layer at end)**




Query #9
it
0.1%
value #4
obey
0.1%
value #3
must
50%
value #2
robot
30%
value #1
a

Self Attention

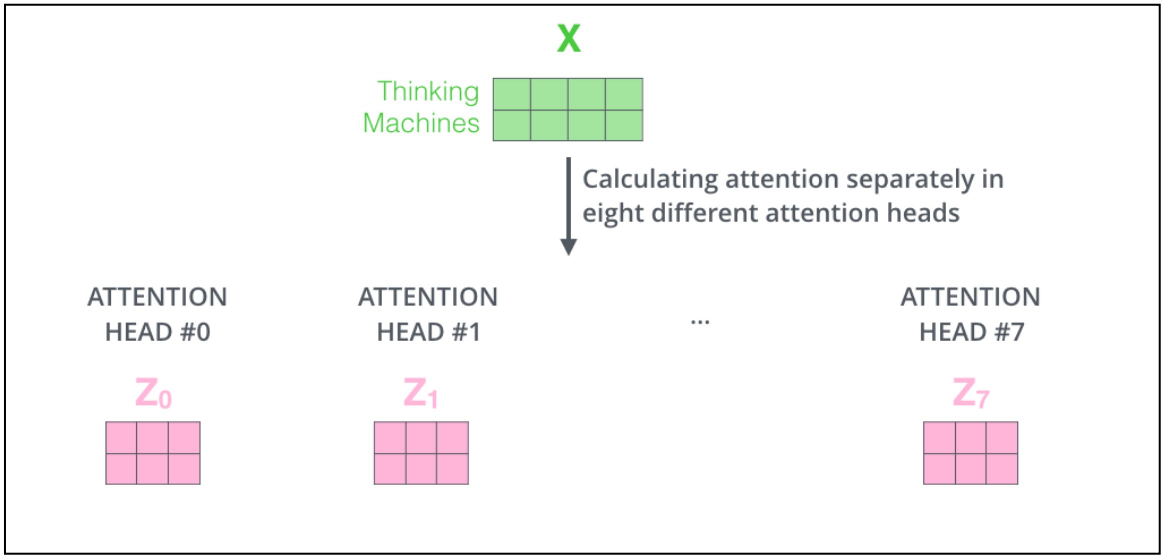
3. The value vectors of the tokens are **multiplied** with the respective scores and them summed up to generate **context vectors**



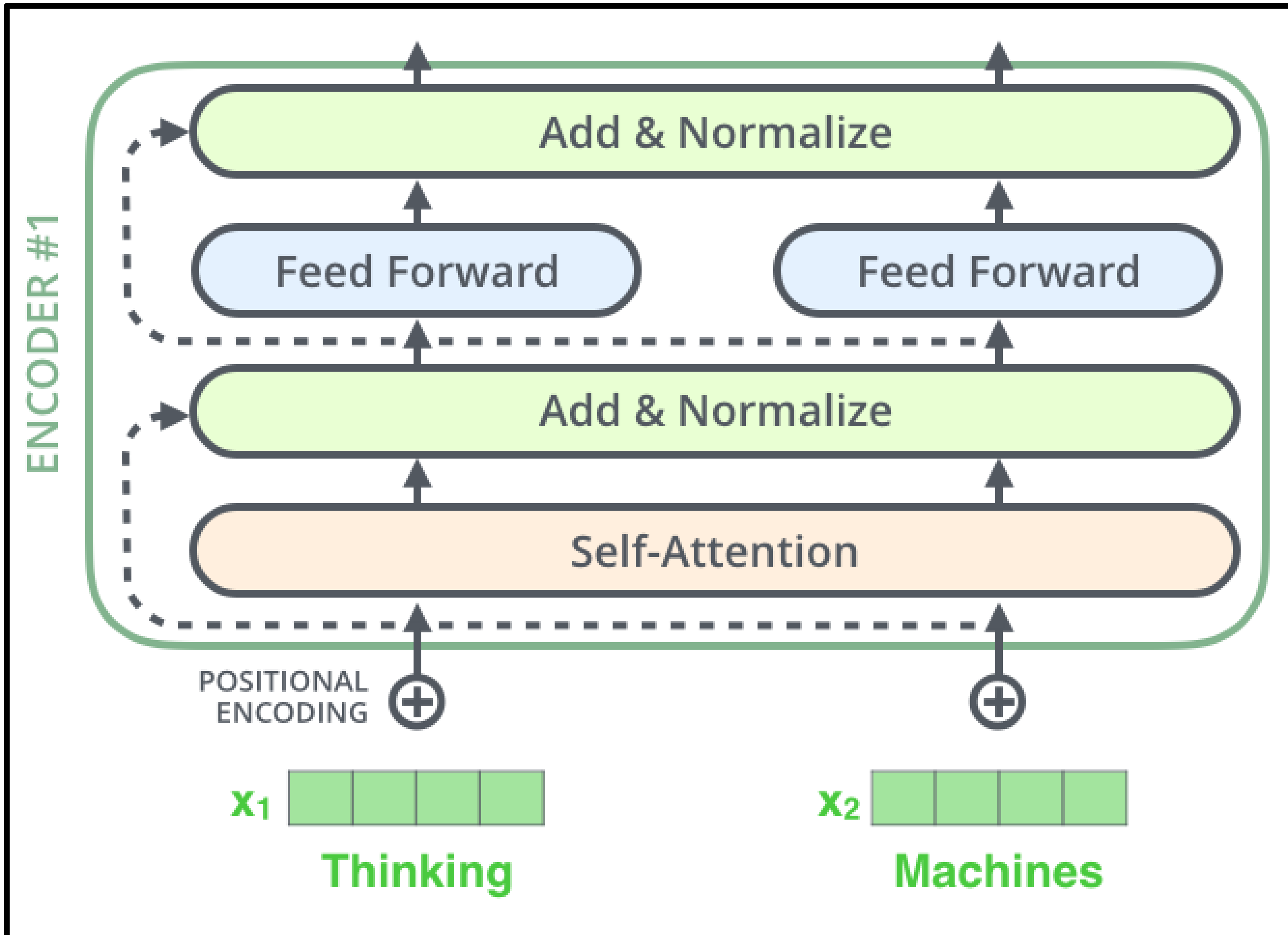
Word	Value vector	Score	Value X Score
<s>		0.001	
a		0.3	
robot		0.5	
must		0.002	
obey		0.001	
the		0.0003	
orders		0.005	
given		0.002	
it		0.19	
		Sum:	

Multi-headed Attention

- Two types of multi-headed attention:
  - Input is linearly transformed to get a different representation and context vectors are computed with different Q, K, V matrices
  - Input is equally divided and passed to different head
- Separate context vectors are computed using self attention
- The resulting attention heads are then concatenated together and passed through a feed-forward neural network, to give the output vector of the multi-headed attention block

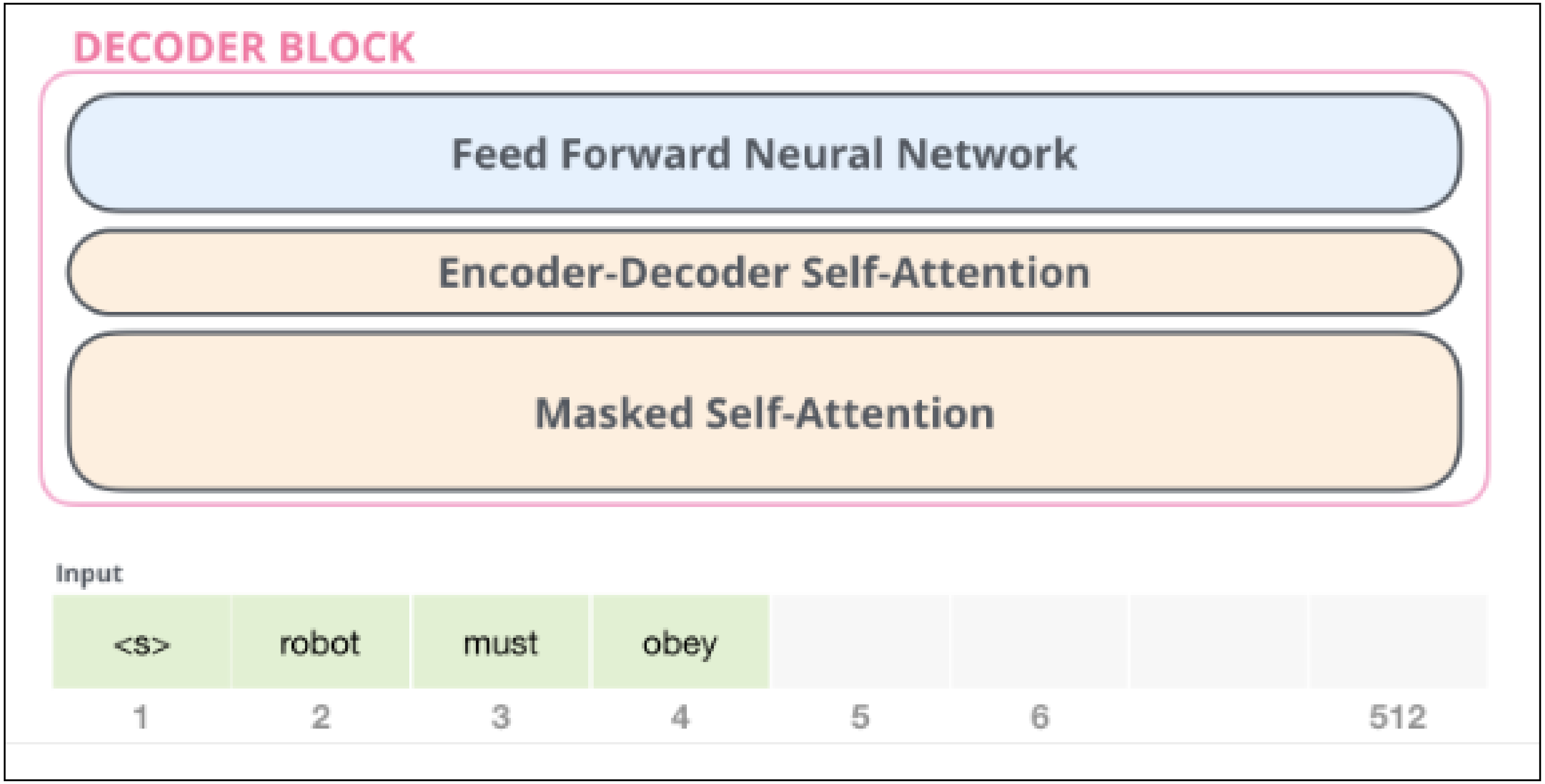


Skip Connections & Layer Normalization

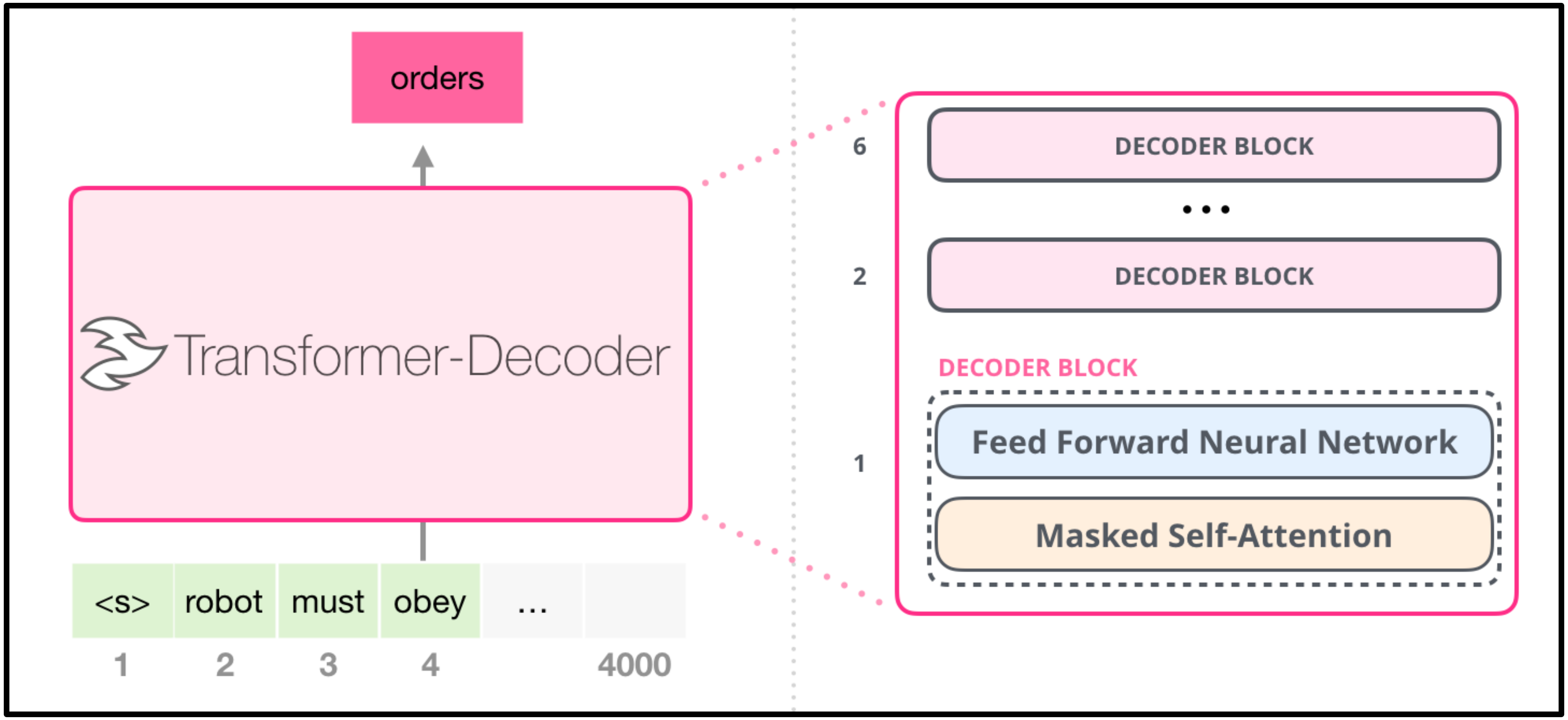


DECODER BLOCK

Transformer Decoder Block



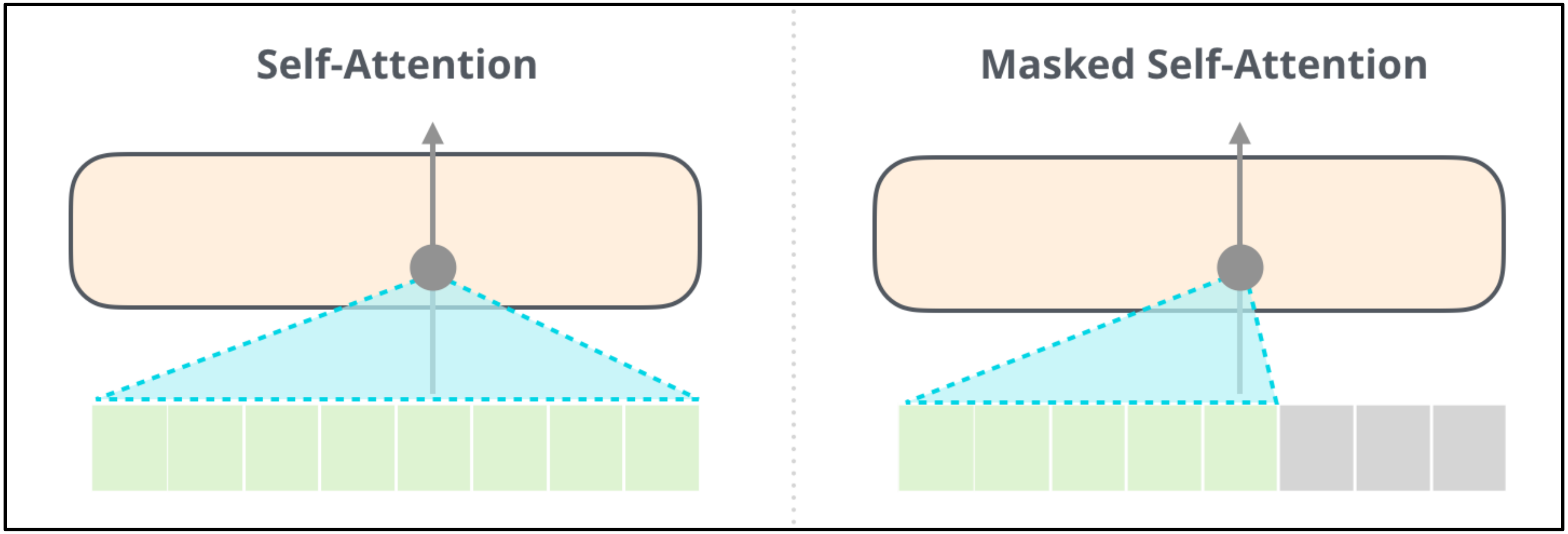
Decoder Only Architecture
GPT/Language Models



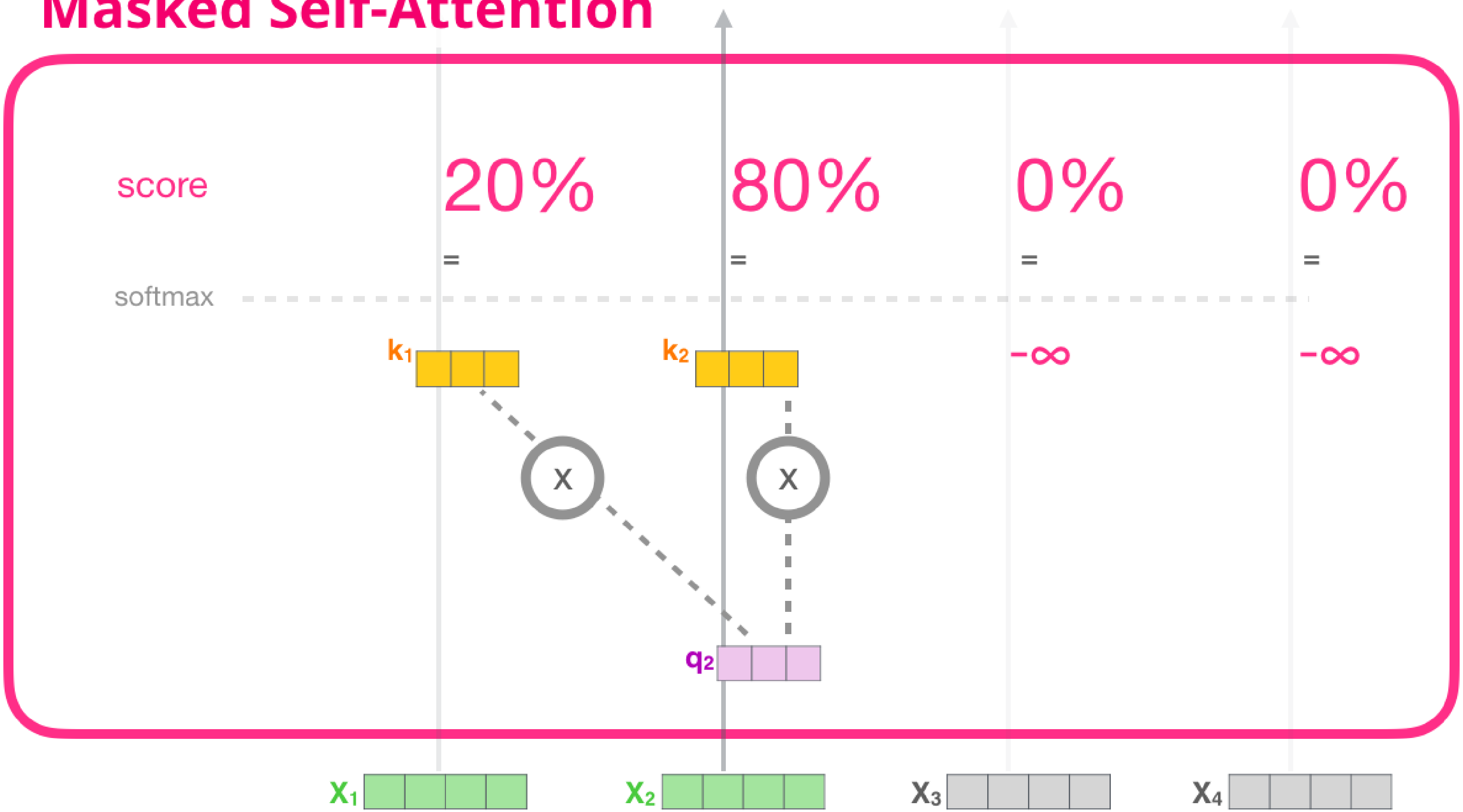
Spot the difference with Transformer Decoder?

Masked Self Attention

- Remember auto-regressive (**left side context)** and **bi-directional** (both side context) models?



Masked Self-Attention



Masked Self Attention

	Features				Labels
Example: / position:	1	2	3	4	
1	robot	must	obey	orders	must
2	robot	must	obey	orders	obey
3	robot	must	obey	orders	orders
4	robot	must	obey	orders	<eos>

Masked Self Attention

Queries

robot	must	obey	orders

X

Keys

robot	must	obey	orders
robot	must	obey	orders
robot	must	obey	orders
robot	must	obey	orders

=

Scores
(before softmax)

0.11	0.00	0.81	0.79
0.19	0.50	0.30	0.48
0.53	0.98	0.95	0.14
0.81	0.86	0.38	0.90

Scores
(before softmax)

0.11	0.00	0.81	0.79
0.19	0.50	0.30	0.48
0.53	0.98	0.95	0.14
0.81	0.86	0.38	0.90

Apply Attention Mask →

Masked Scores
(before softmax)

0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

Masked Scores
(before softmax)

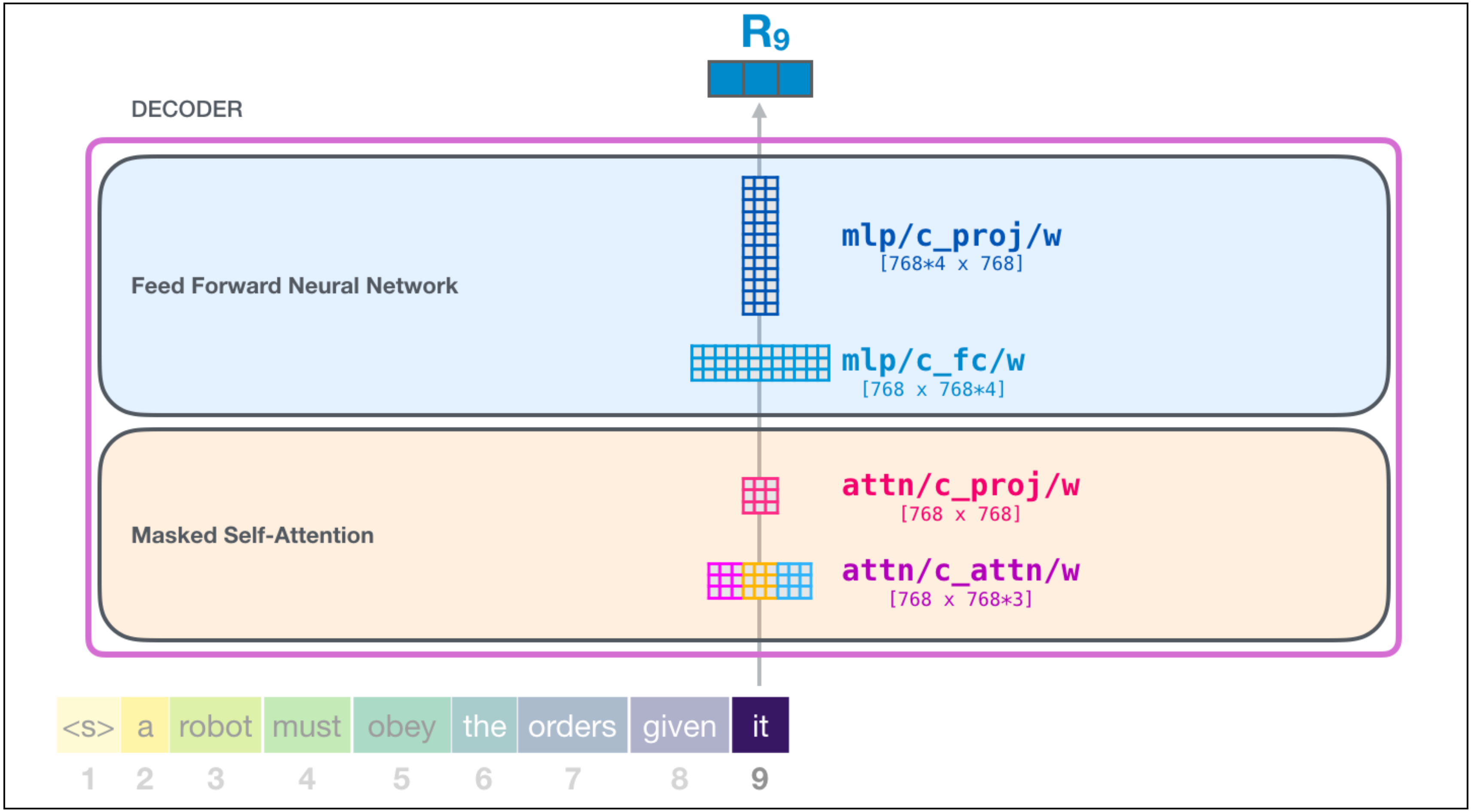
0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

Softmax
(along rows) →

Scores

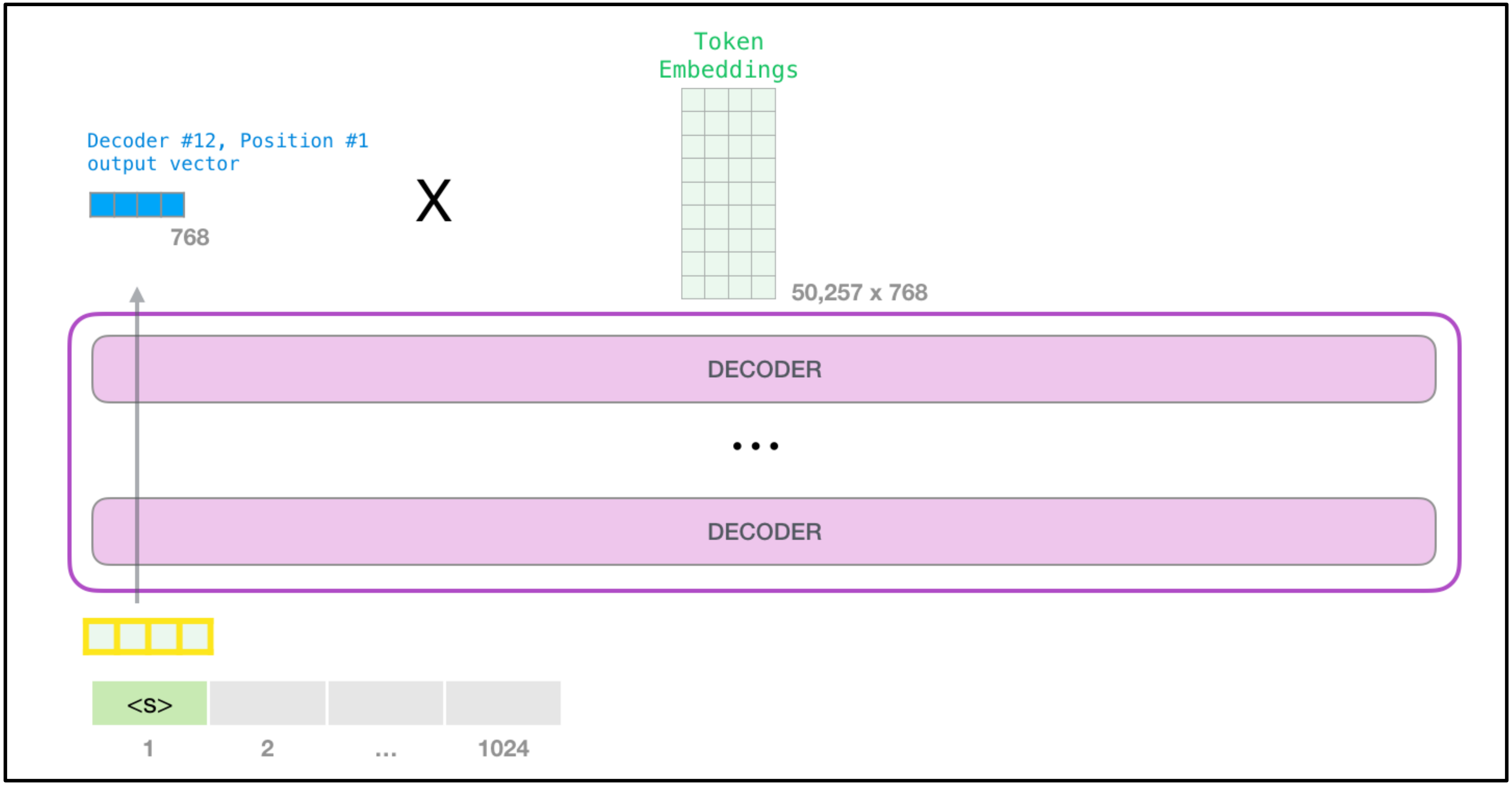
1	0	0	0
0.48	0.52	0	0
0.31	0.35	0.34	0
0.25	0.26	0.23	0.26

Feed Forward Network



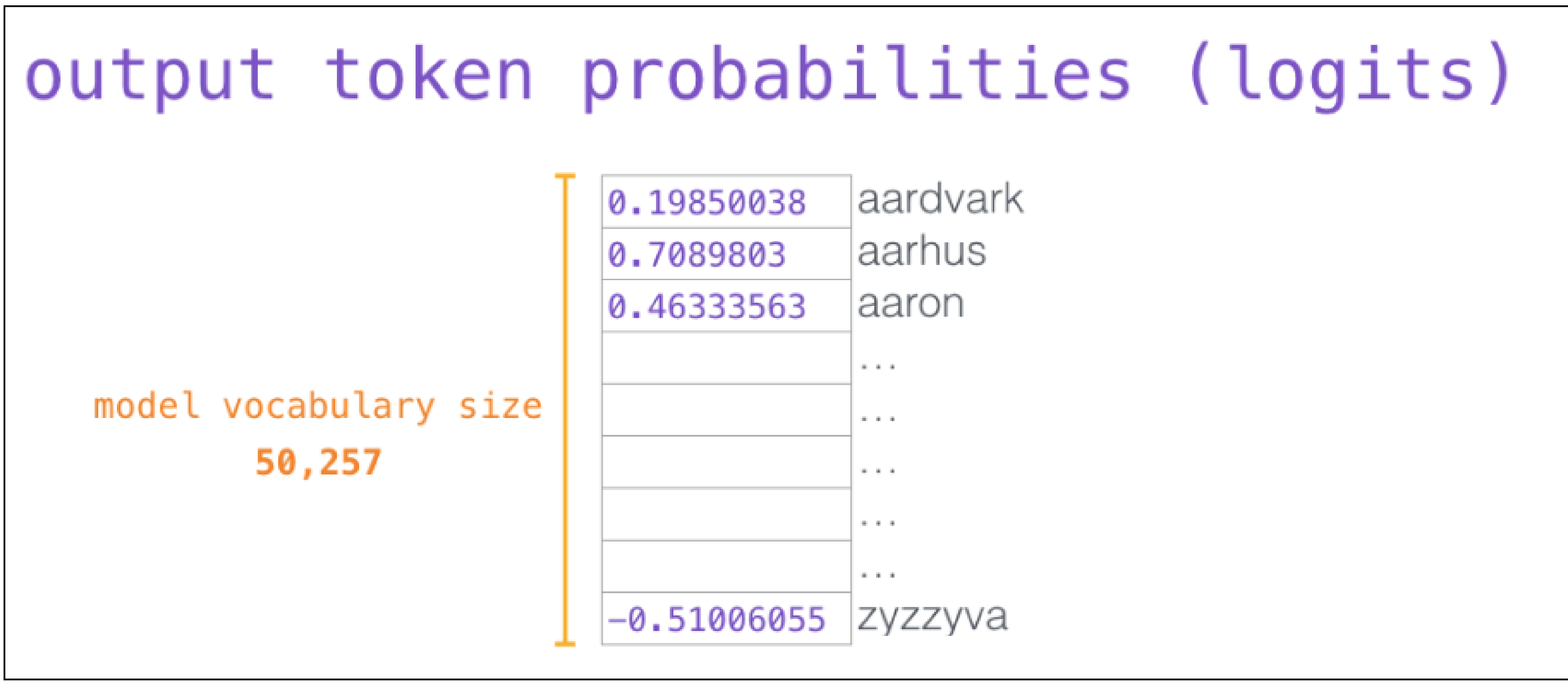
Saving Computations/Flow of GPT

Step-1: Predicting first word



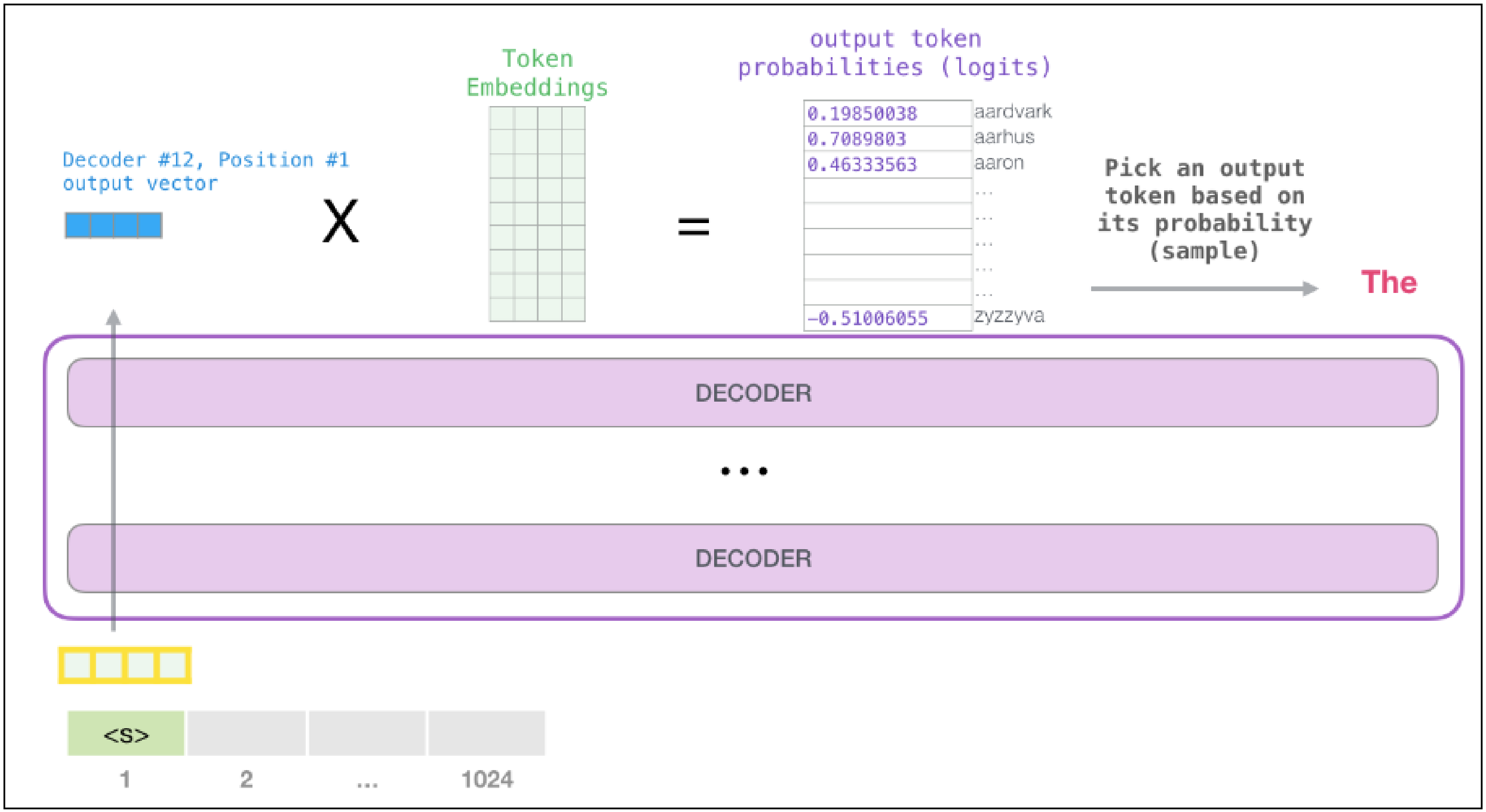
Saving Computations/Flow of GPT

Step-1: Predicting first word



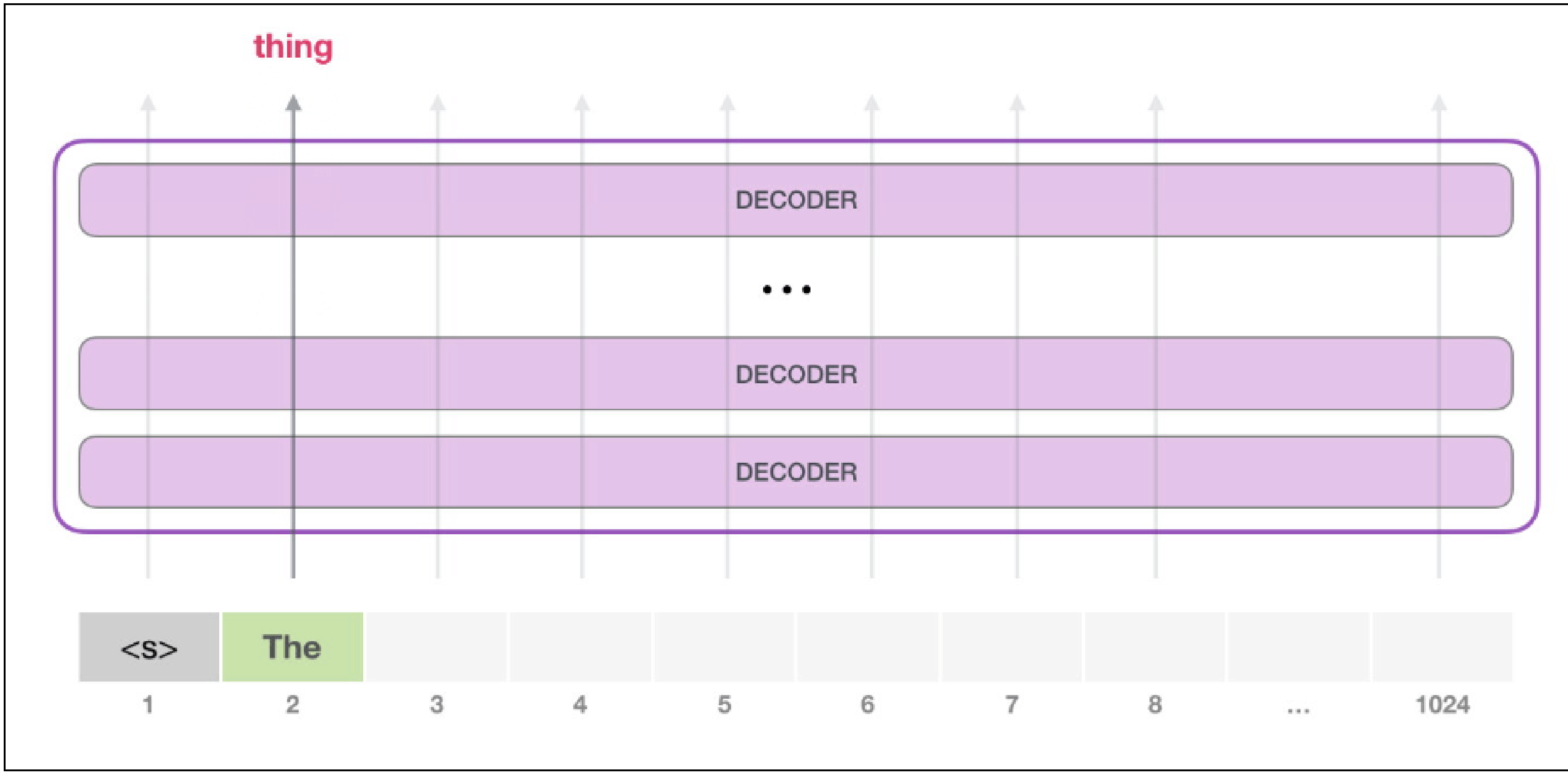
Saving Computations/Flow of GPT

Step-1: Predicting first word



Saving Computations/Flow of GPT

Step-2: Predicting second word



Guess where we saved computations?

How many word GPT predicts at once?

FINE TUNING GPT-2 FOR MOVIE NAME GENERATION

Dataset (movies.csv)

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy
...	...	...	...
9737	193581	Black Butler: Book of the Atlantic (2017)	Action|Animation|Comedy|Fantasy
9738	193583	No Game No Life: Zero (2017)	Animation|Comedy|Fantasy
9739	193585	Flint (2017)	Drama
9740	193587	Bungo Stray Dogs: Dead Apple (2018)	Action|Animation
9741	193609	Andrew Dice Clay: Dice Rules (1991)	Comedy

Dataset (movies.csv)

```
#loading dataset from a csv file of movie names
movies_file = "movies.csv"
max_length = 10
raw_df = pd.read_csv(movies_file)
movie_names = raw_df['title']  #only keeping the title column
movie_list = list(movie_names) #converting from pandas to list
for i in range(len(movie_list)):  #removing year from title
    movie_list[i]=re.sub("\([0-9]+\)", "", movie_list[i]).strip()
```

```
['Toy Story',
 'Jumanji',
 'Grumpier Old Men',
 'Waiting to Exhale',
 'Father of the Bride Part II']
```

Dataset (movies.csv)

```
class MovieDataset(Dataset):
    def __init__(self, tokenizer, init_token, movie_titles, max_len):
        self.max_len = max_len
        self.tokenizer = tokenizer
        self.eos = self.tokenizer.eos_token
        self.eos_id = self.tokenizer.eos_token_id
        self.movies = movie_titles
        self.allTokenizedTitles = []
        self.allTokenizedTitles2 = []
        tokenizer.pad_token = tokenizer.eos_token
        for movie in self.movies:
            # Encode the text using tokenizer.encode(). We ass EOS at the end
            tokenized_output=self.tokenizer(init_token + movie + self.eos,truncation=True,padding='max_length',max_length=13)
            self.allTokenizedTitles.append(torch.tensor(tokenized_output['input_ids']))

    def __len__(self):
        return len(self.allTokenizedTitles)


    def __getitem__(self, item):
        tmp = self.allTokenizedTitles[item]
        if(tmp[-1]!=50256):
            xx=0
        return self.allTokenizedTitles[item]
```

```
#datasets and dataloaders
dataset = MovieDataset(tokenizer, "movie: ", movie_list, max_length)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True, drop_last=True)
```

Model+Tokenizer

```
#LOADING GPT model and tokenizers
tokenizer = AutoTokenizer.from_pretrained("gpt2",cache_dir='cache/')
extra_length = len(tokenizer.encode("movie: "))
model = AutoModelWithLMHead.from_pretrained("gpt2",cache_dir='cache/')
model = model.to(device)
```

Training

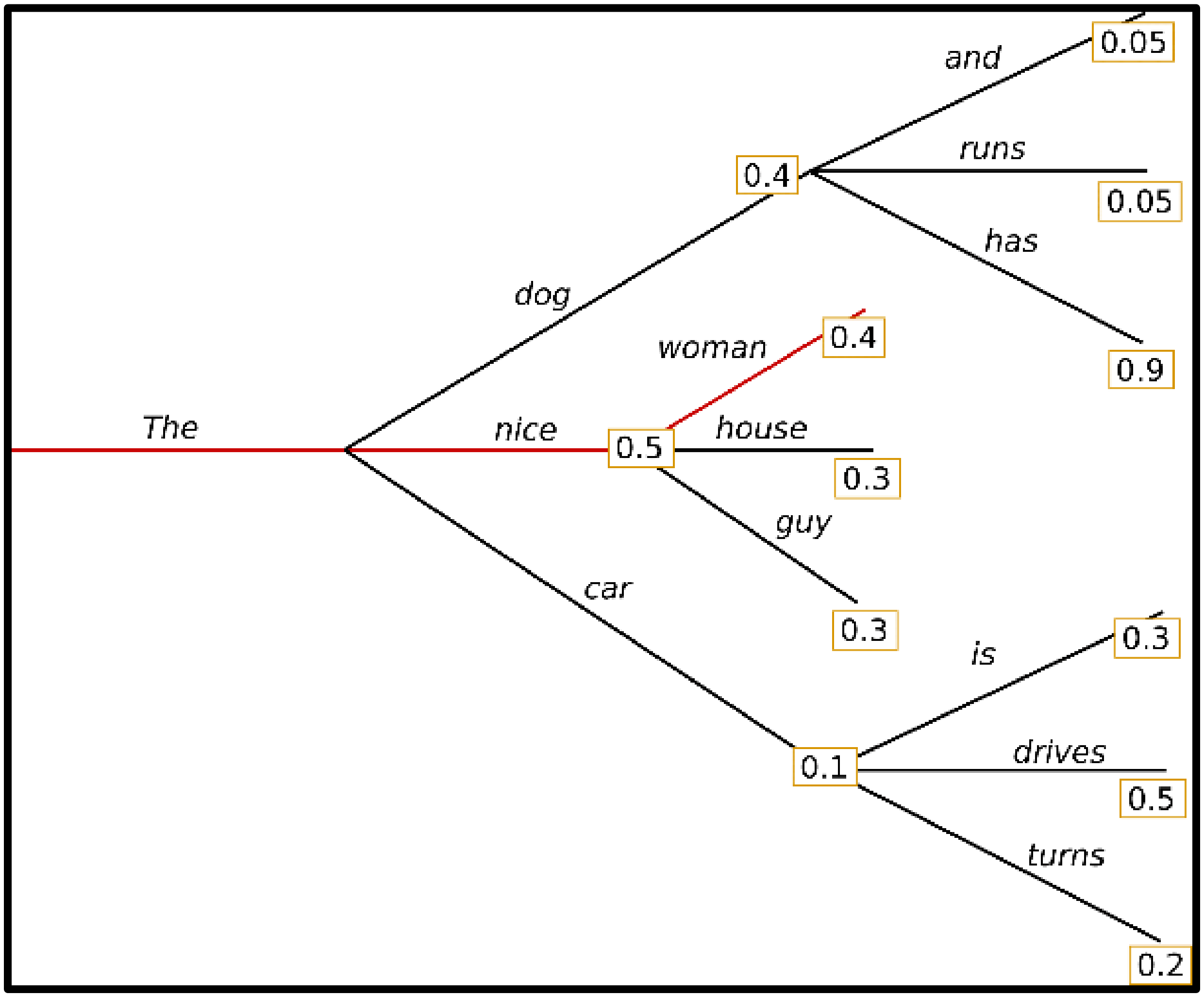
```
epochs=2
optimizer = optim.AdamW(model.parameters(), lr=3e-4)
for epoch in range(epochs):
    for idx, batch in enumerate(dataloader):
        batch = batch.to(device)
        output = model(batch, labels=batch)
        loss = output.loss
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        if idx % 50 == 0:
            print("loss: %f, %d"%(loss, idx))
```

Inference

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(device)
model=torch.load("/content/drive/MyDrive/lec11/model.pt")
model=model.to(device)
tokenizer = AutoTokenizer.from_pretrained("gpt2",cache_dir="/content/drive/MyDrive/HuggingFace/")


input_ids = tokenizer.encode('movie: ',return_tensors='pt')
input_ids=input_ids.to(device)
sample_outputs = model.generate(
    input_ids,
    do_sample=True,
    max_length=13,
    top_k=9,
    num_return_sequences=10

)

print("Output:\n" + 100 * '-')
for i, sample_output in enumerate(sample_outputs):
  print("{}: {}".format(i, tokenizer.decode(sample_output, skip_special_tokens=True)))
```

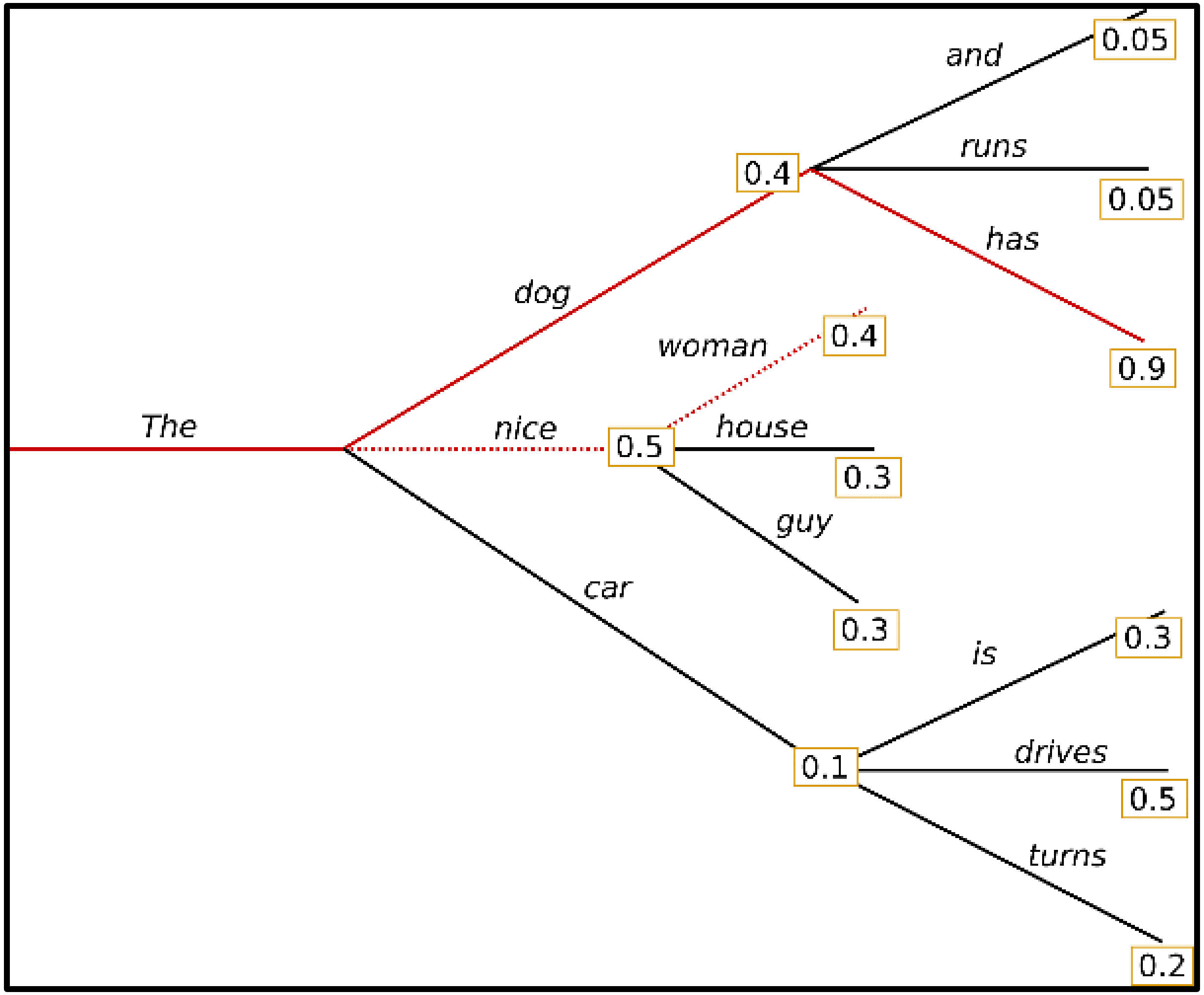
Inference (Greedy Search)

https://huggingface.co/blog/how-to-generate



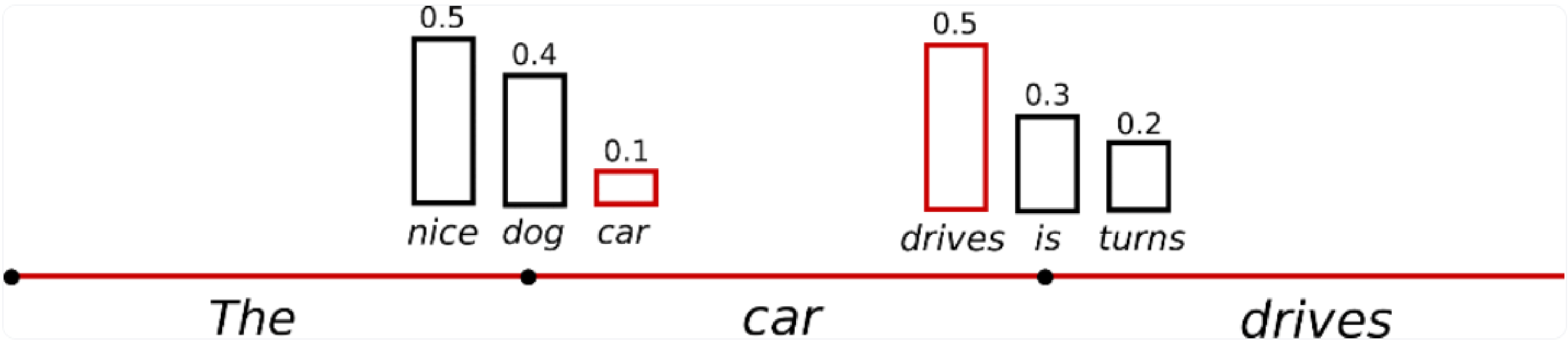
Inference (Beam Search)

https://huggingface.co/blog/how-to-generate



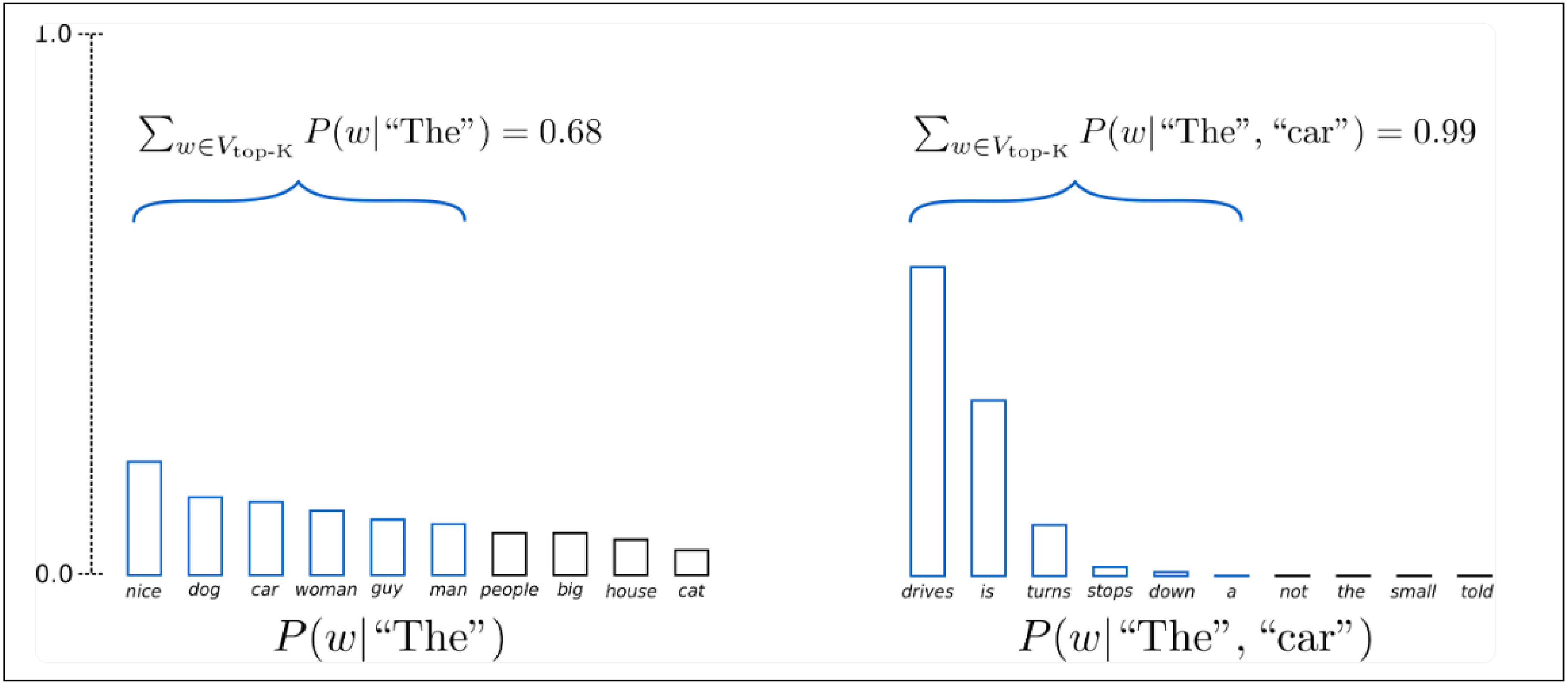
Inference (Sampling)

https://huggingface.co/blog/how-to-generate



Inference (Top-K Sampling)

https://huggingface.co/blog/how-to-generate



Home Task 7

- Train the movie title GPT model
- Generate titles using different sampling methods discussed
- Which one preformed best