CSI2121: Big Data

Ch4. Transformers and Langauge Models

Jinyoung Yeo

Yonsei AI

Outline

- Transformer
- Language Models

Transformer

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

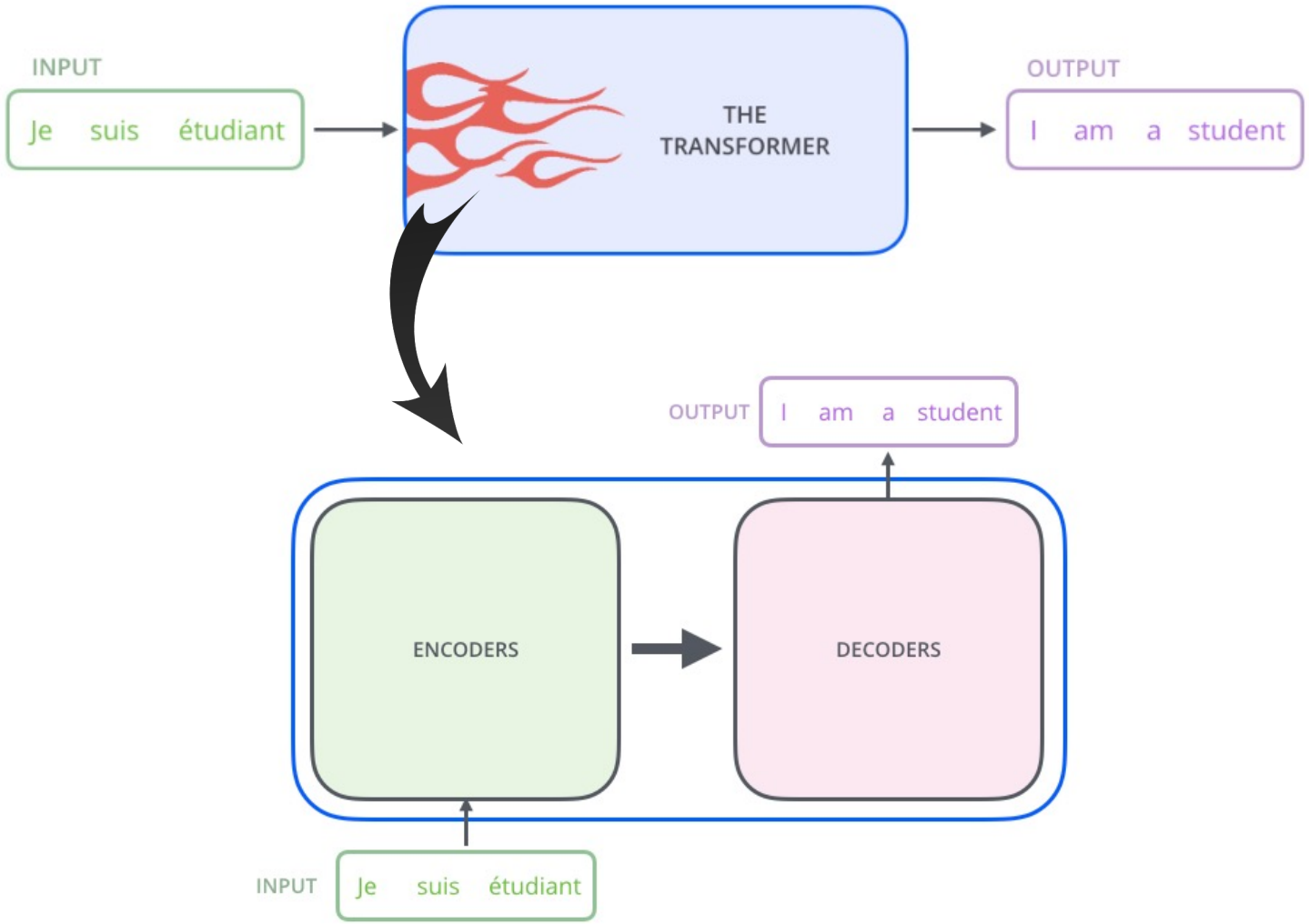
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

RNN → LSTM → LSTM + attention → Only attention (Transformer)

A High-level Look

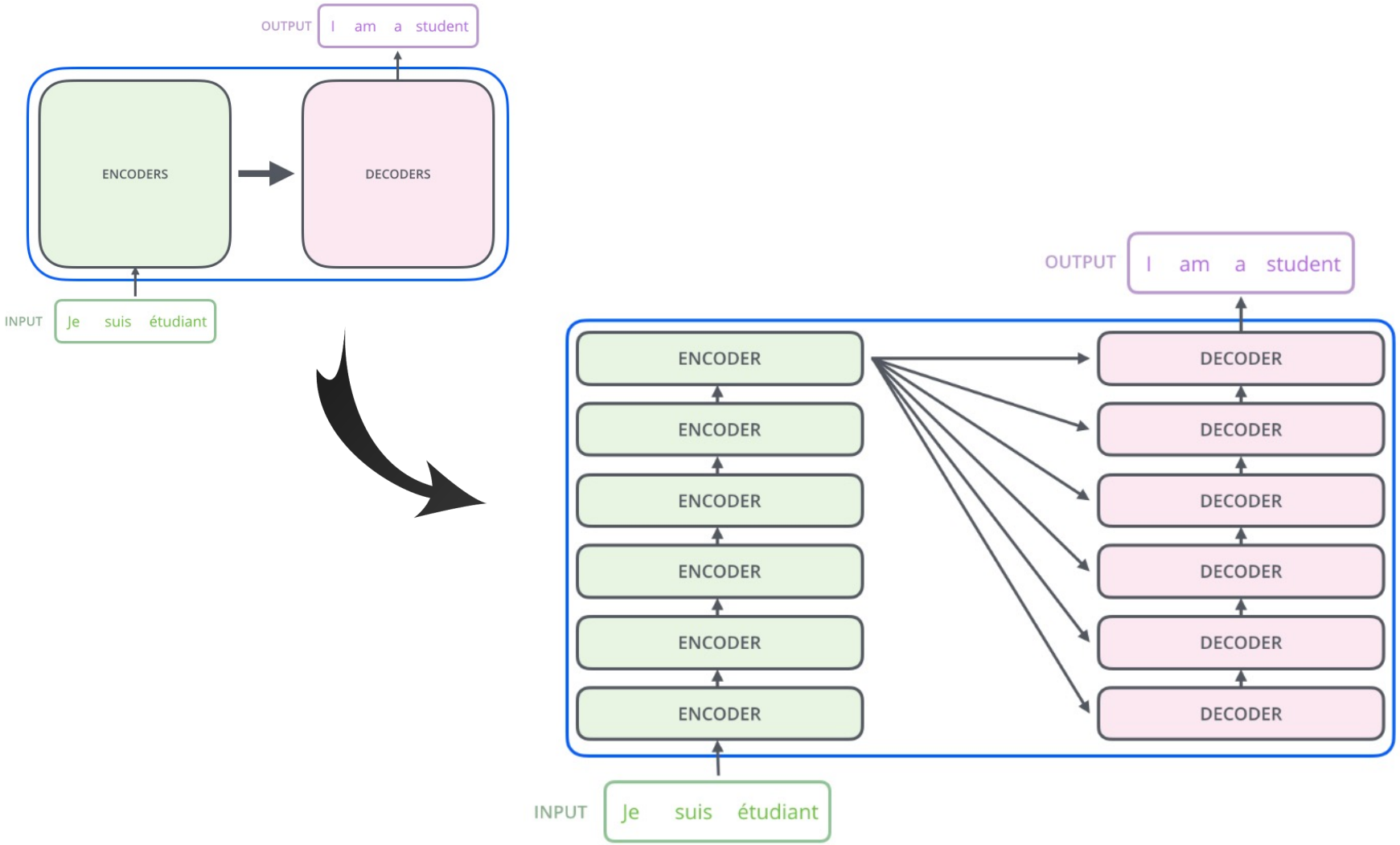


A High-level Look

```
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this and many
    other models.
    """
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask), src_mask,
                           tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

```
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return F.log_softmax(self.proj(x), dim=-1)
```

A High-level Look

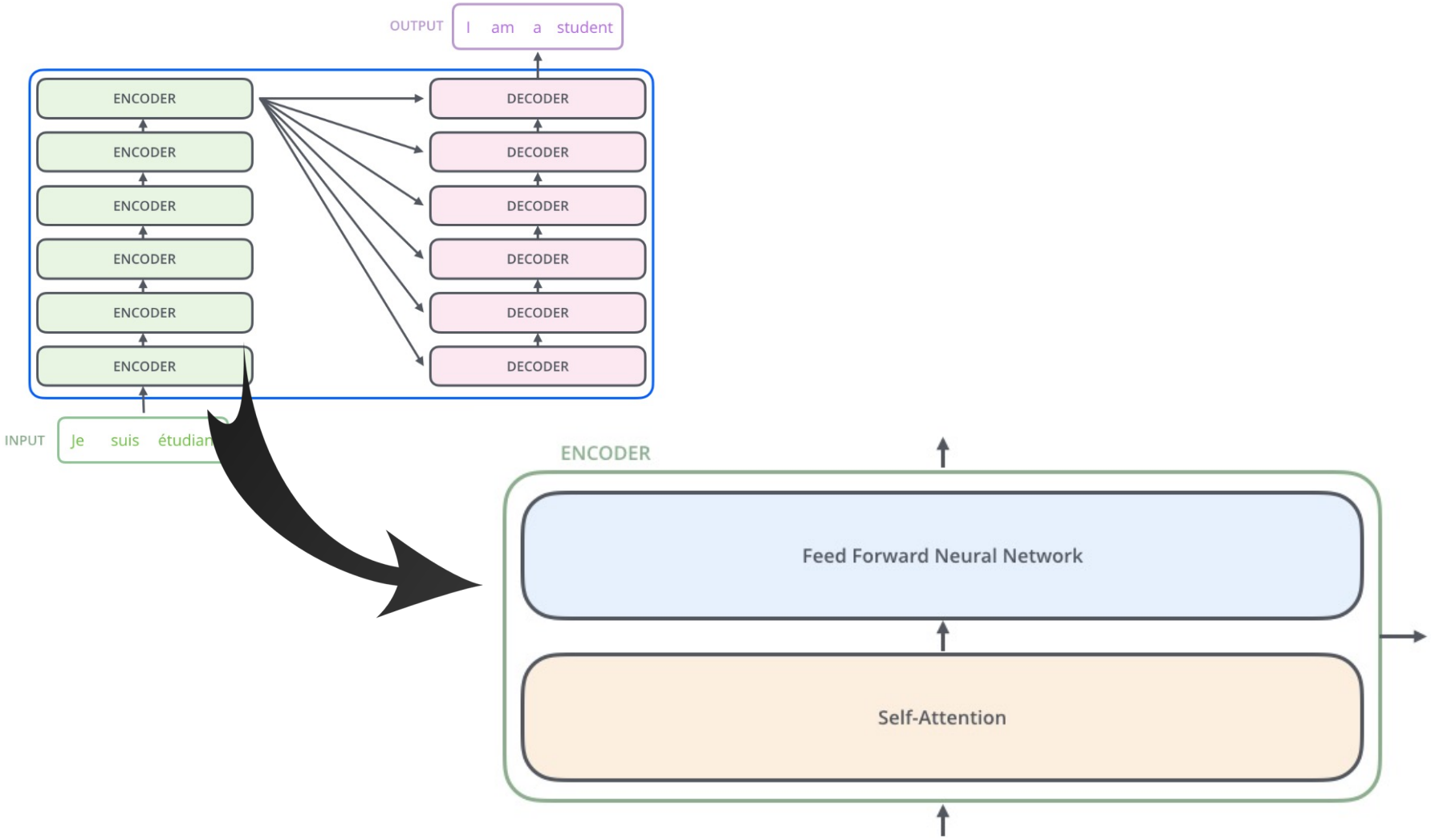


A High-level Look

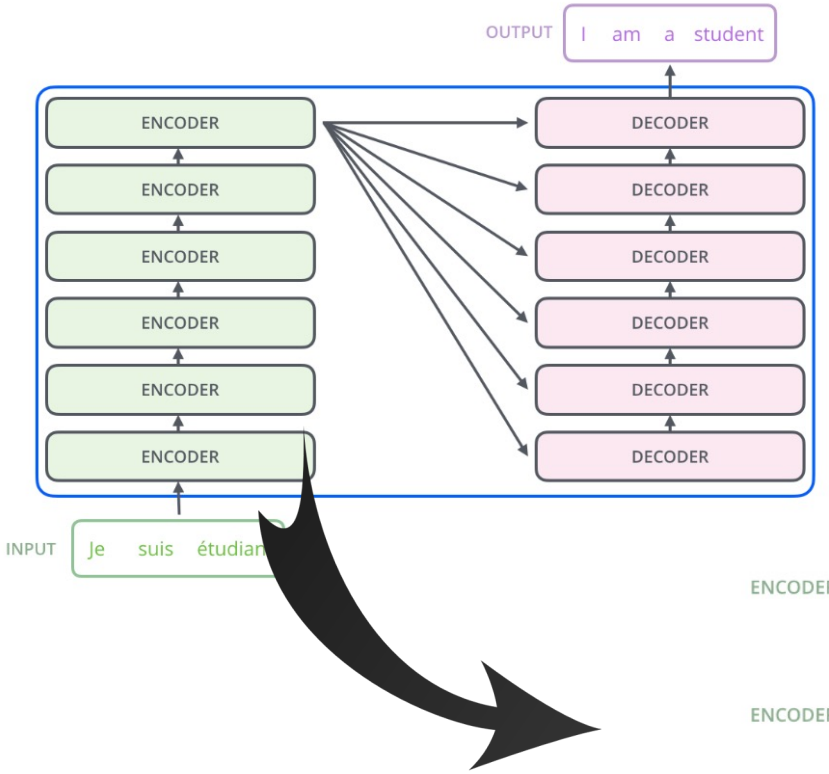
```
def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
```

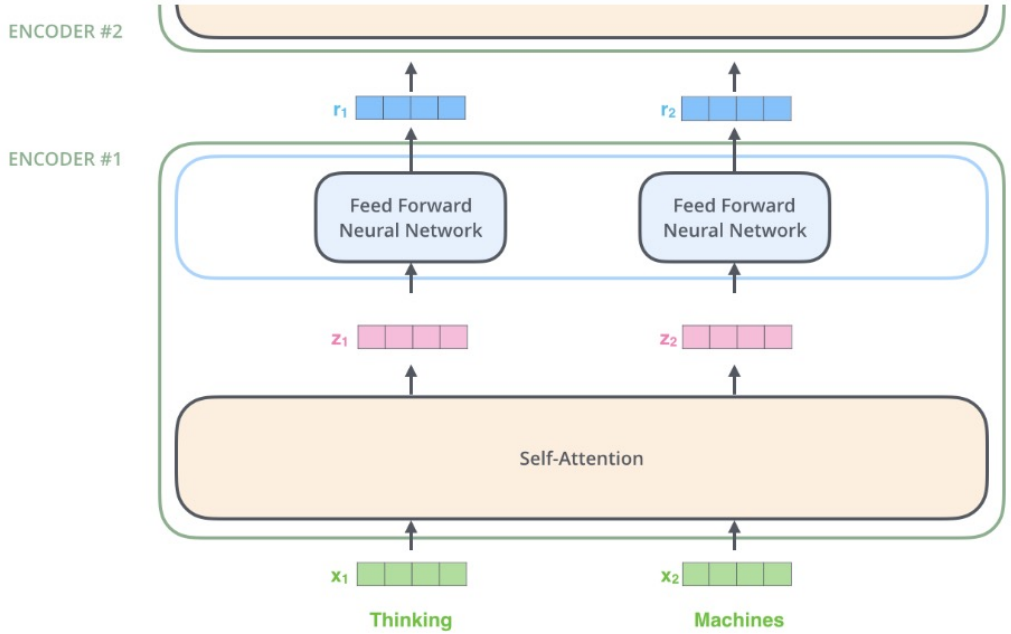
```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

A High-level Look

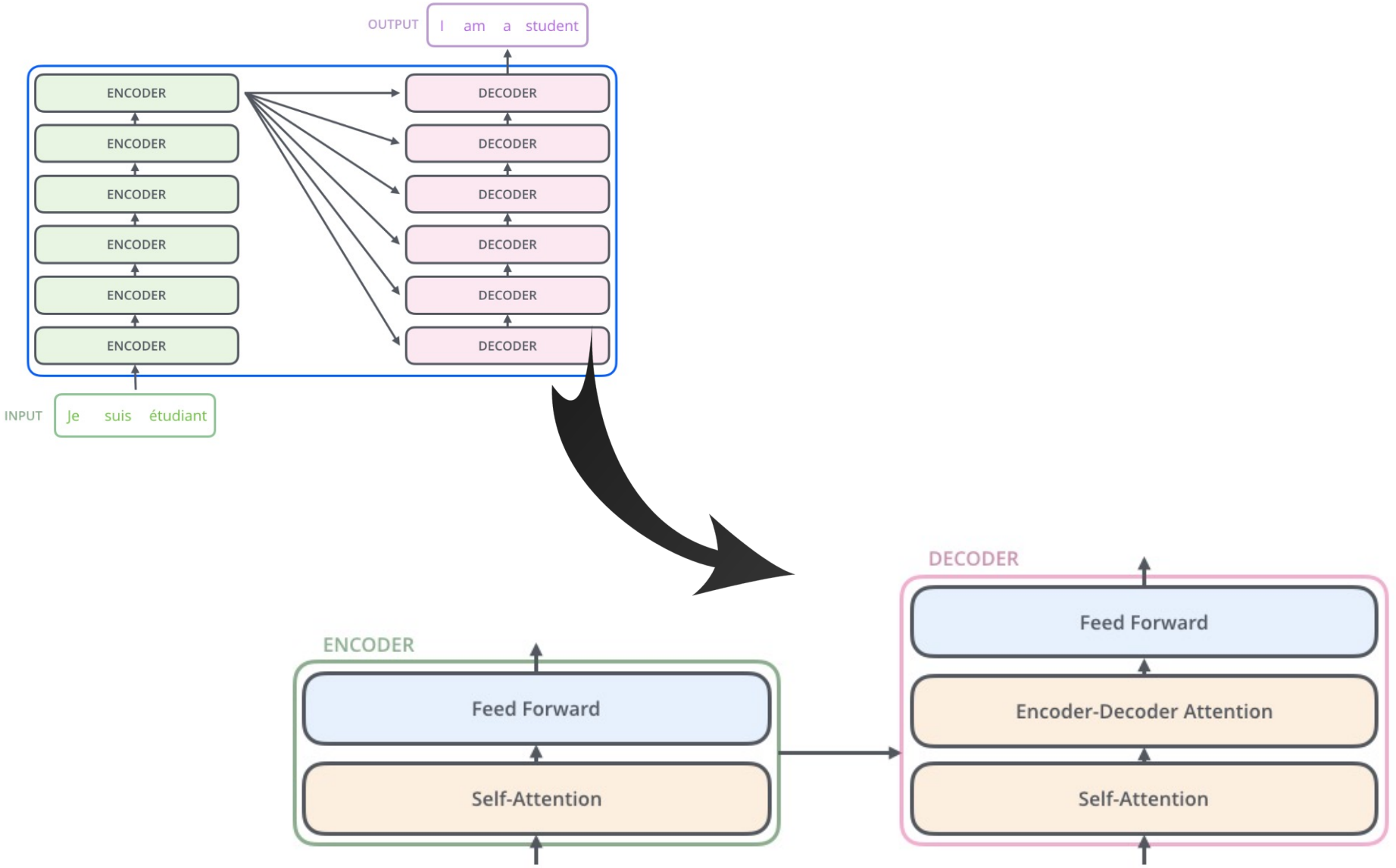


A High-level Look

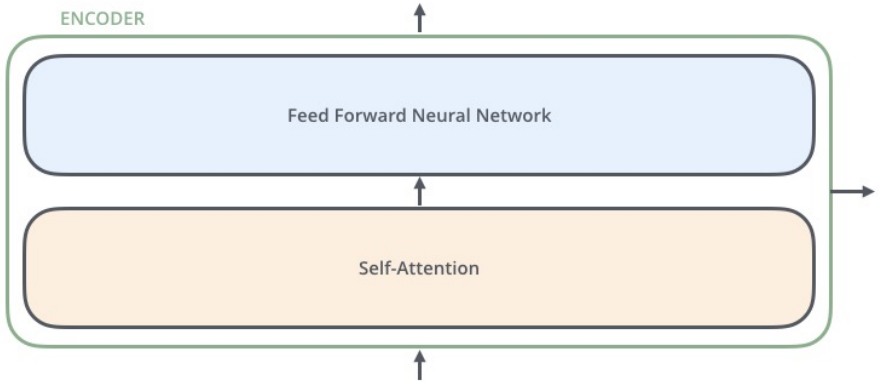


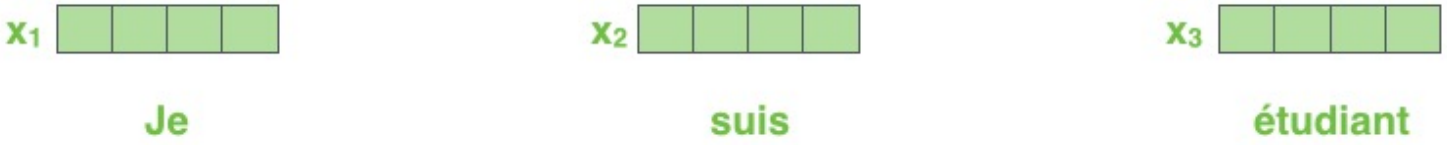


A High-level Look



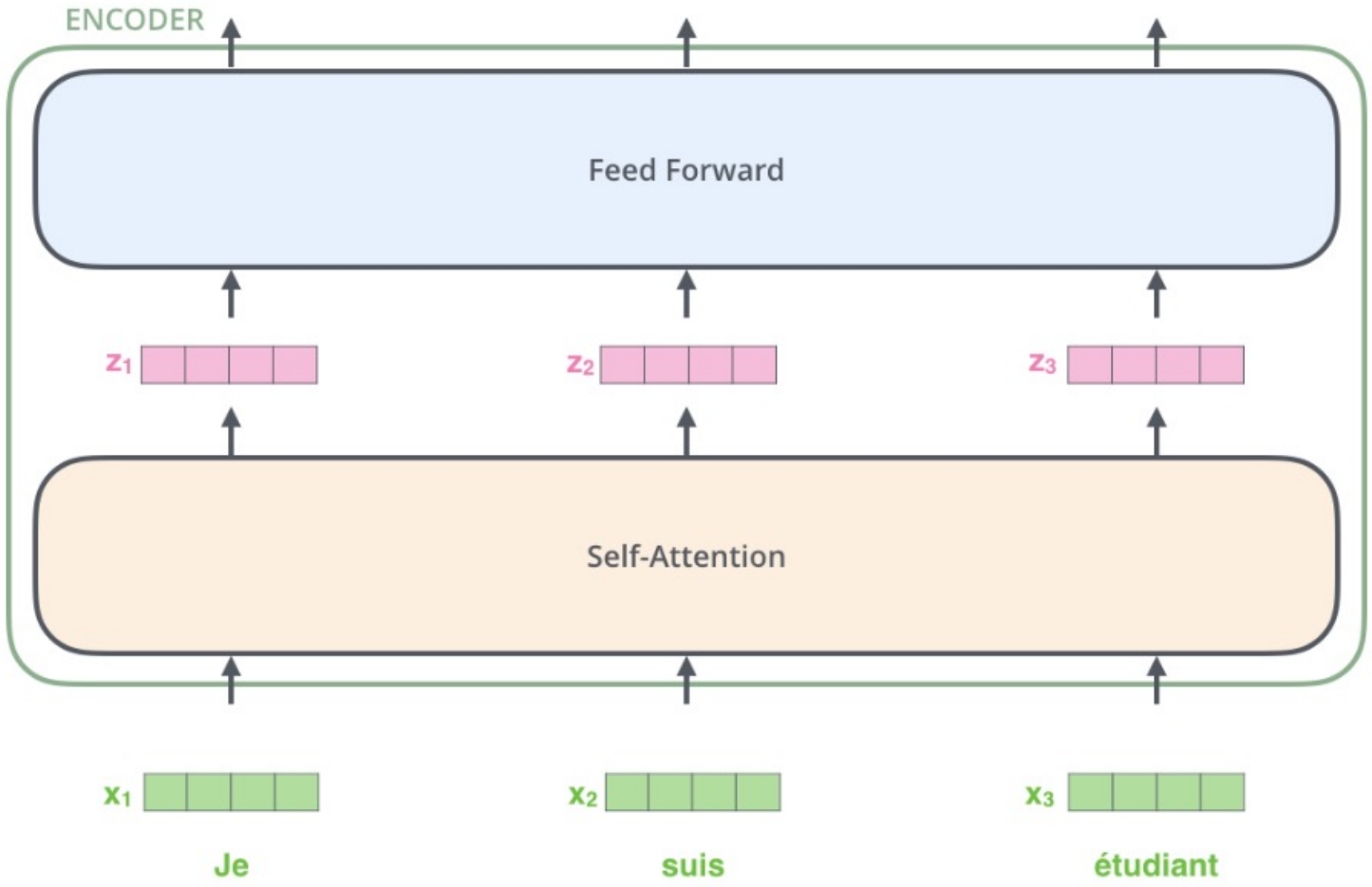
Bringing The Tensors Into The Picture



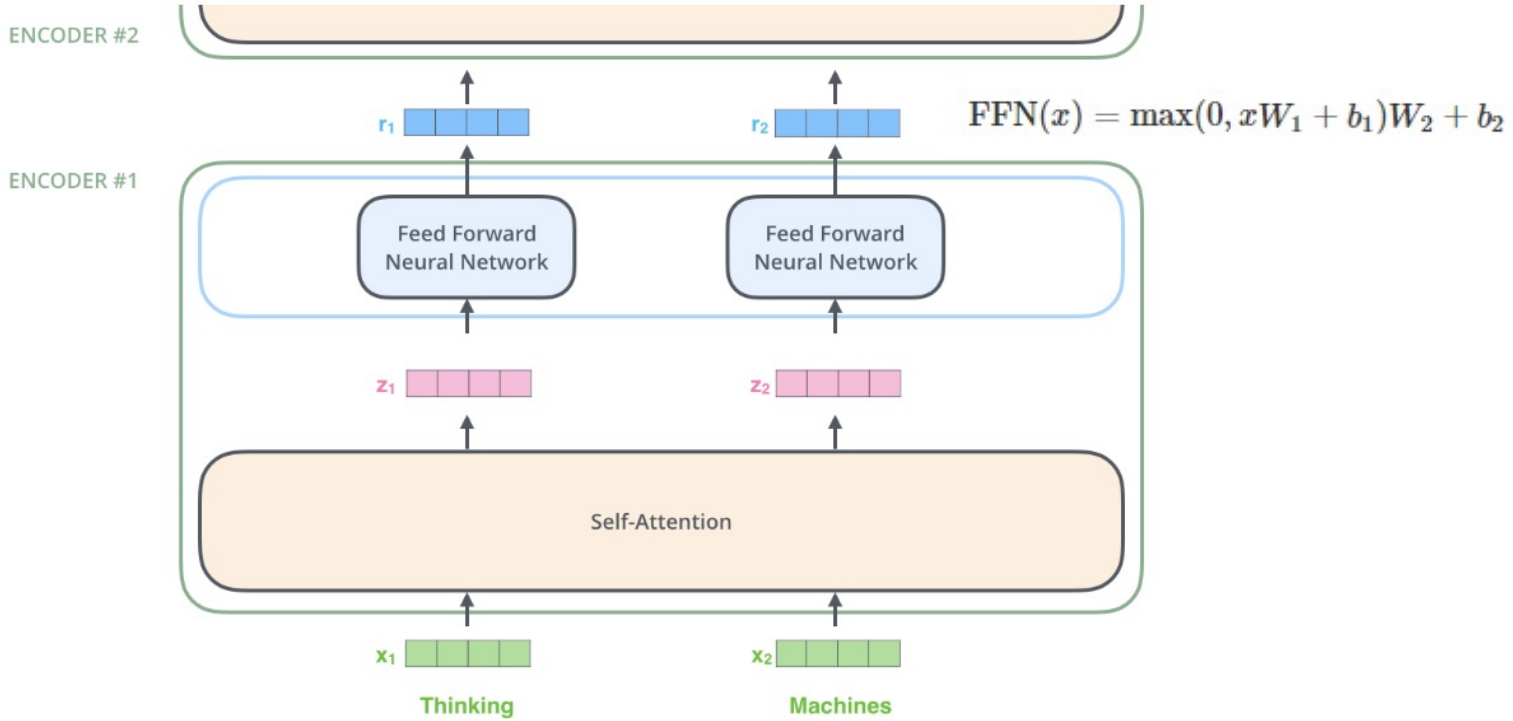


Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

Bringing The Tensors Into The Picture



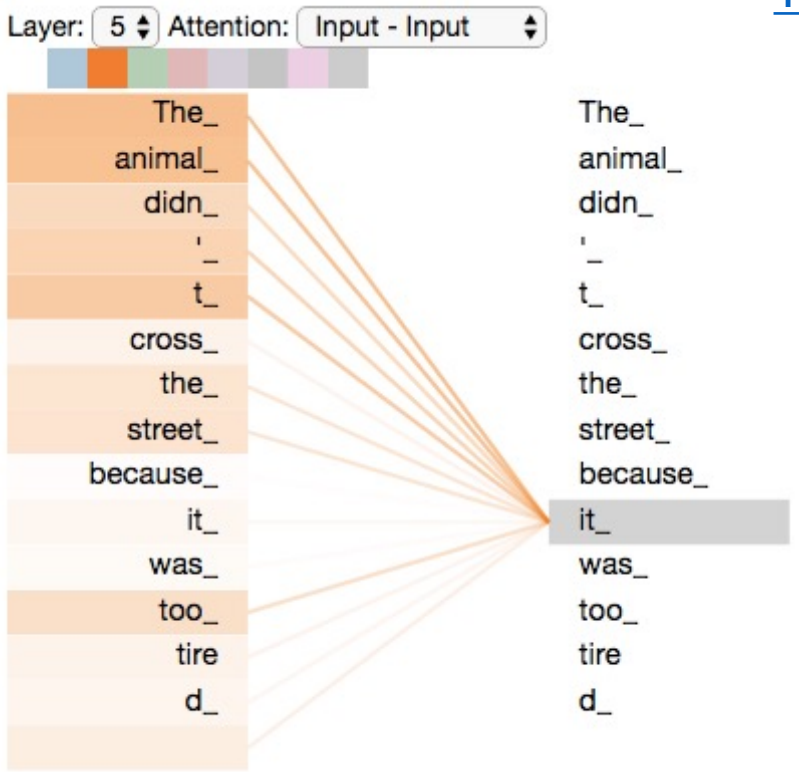
Now We're Encoding!



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

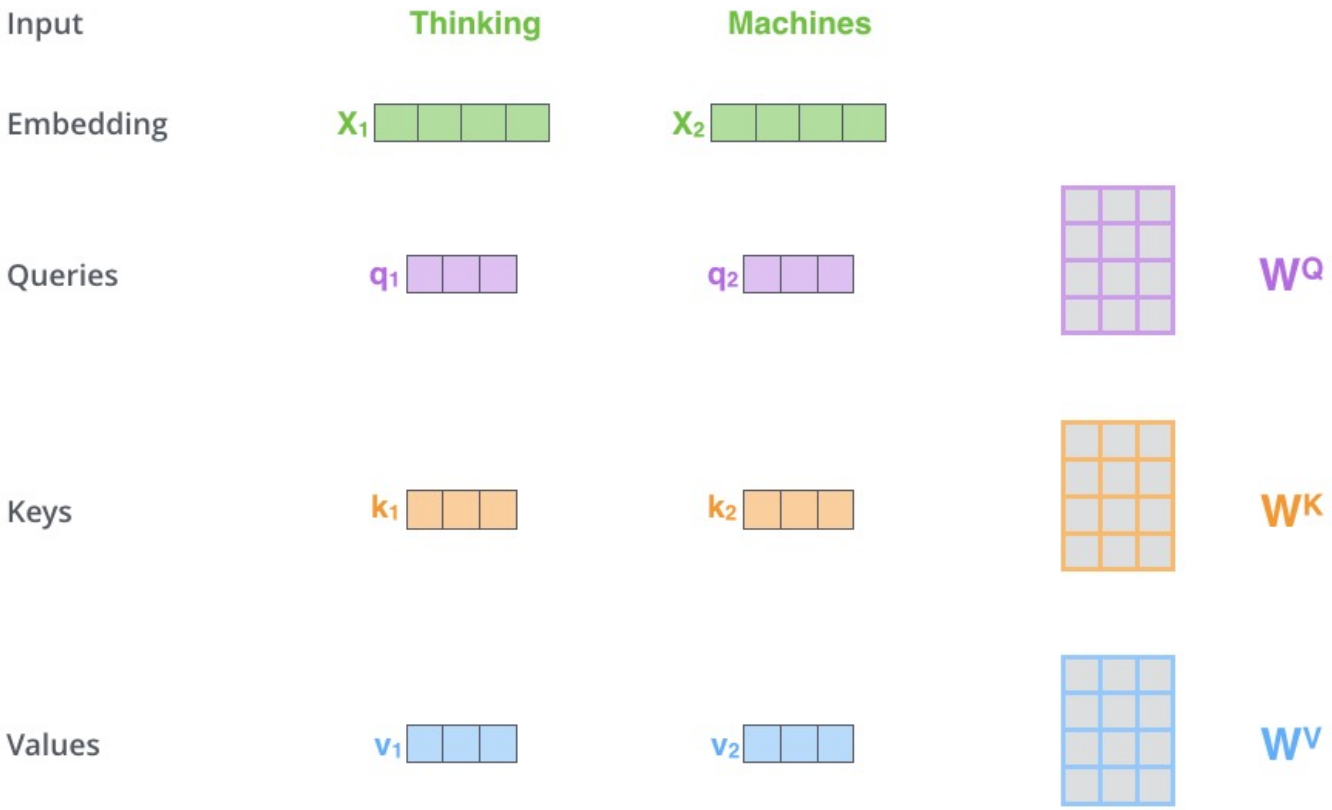
Self-Attention at a High Level

Tensor2Tensor notebook



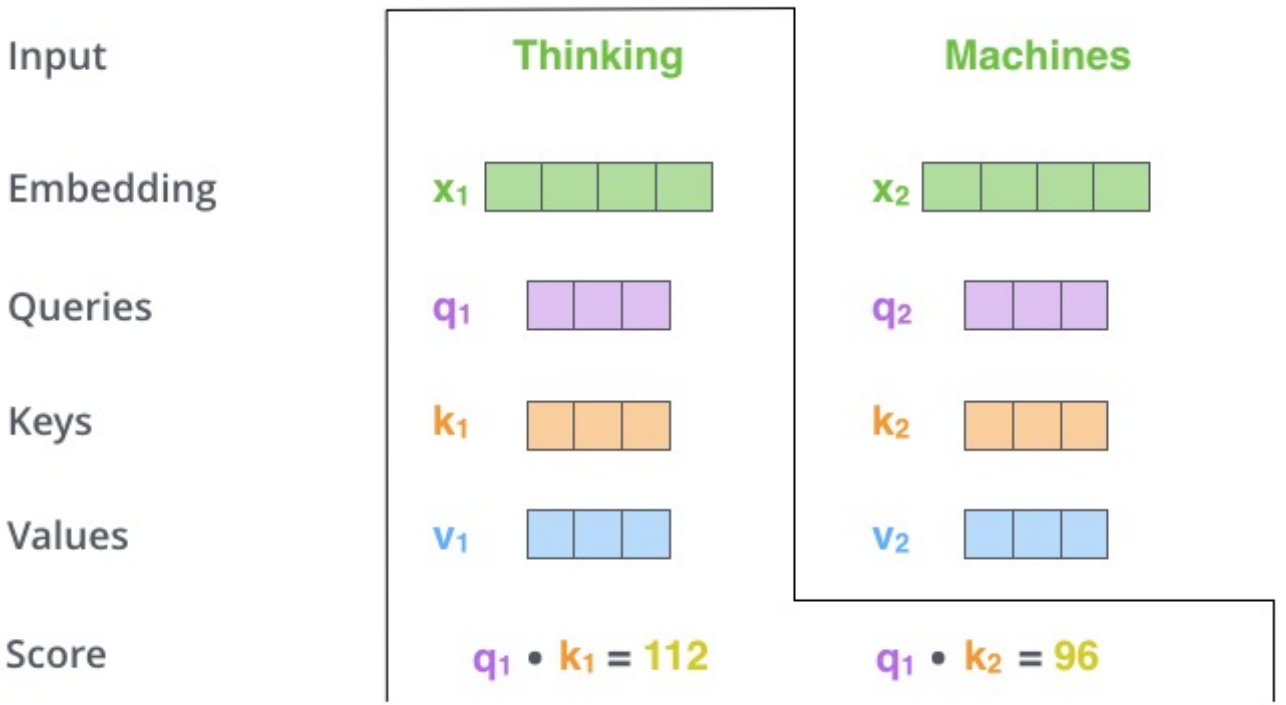
As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

Self-Attention in Detail

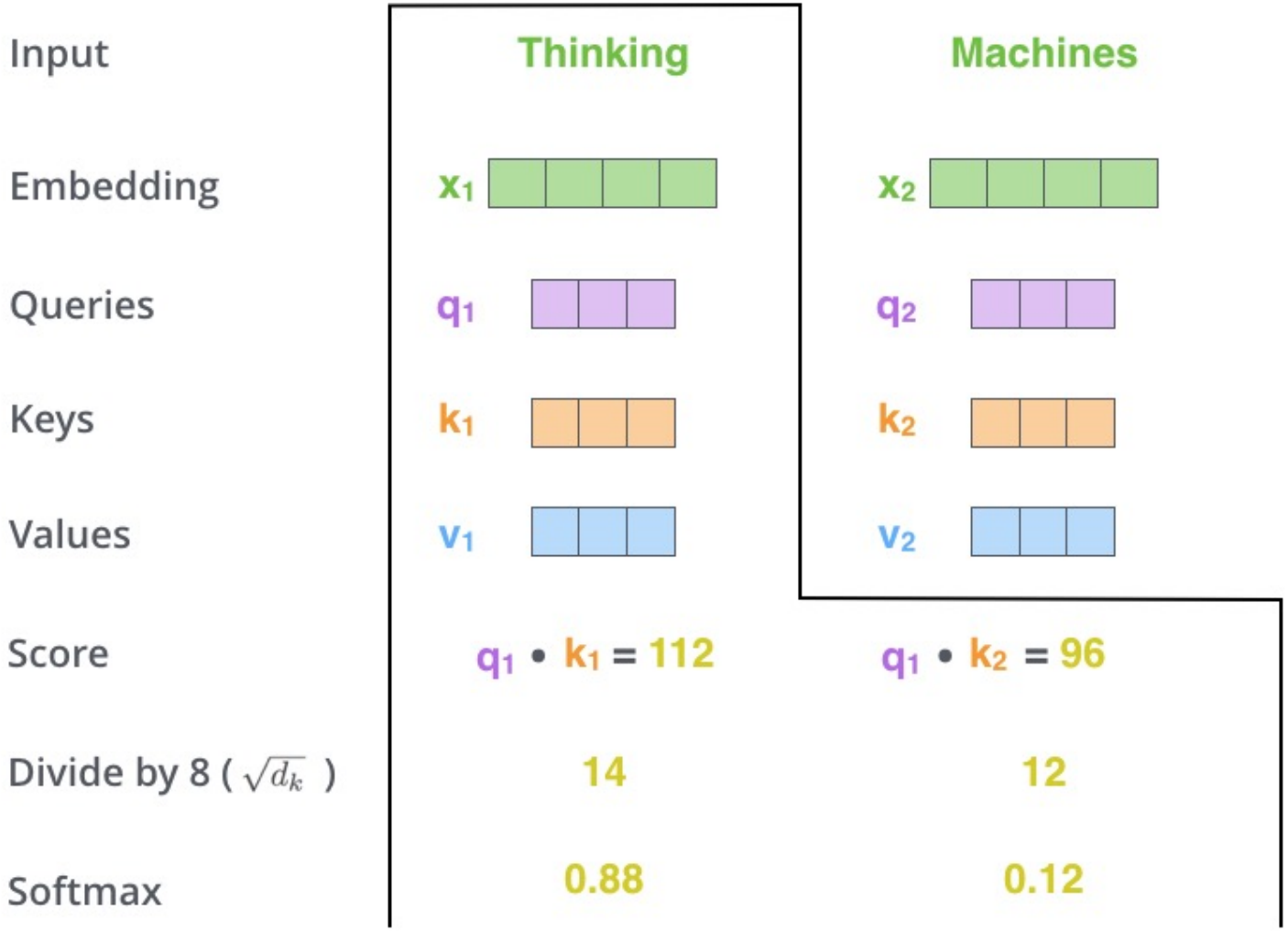


Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

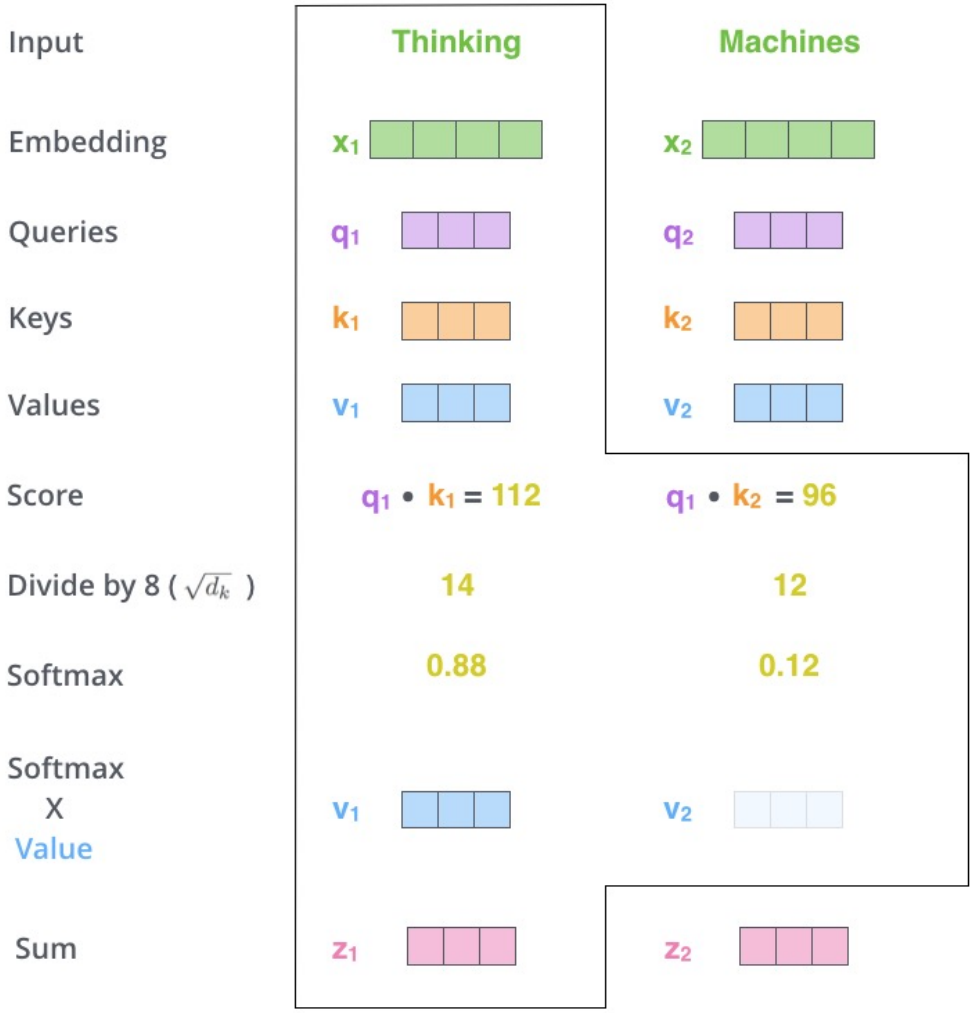
Self-Attention in Detail



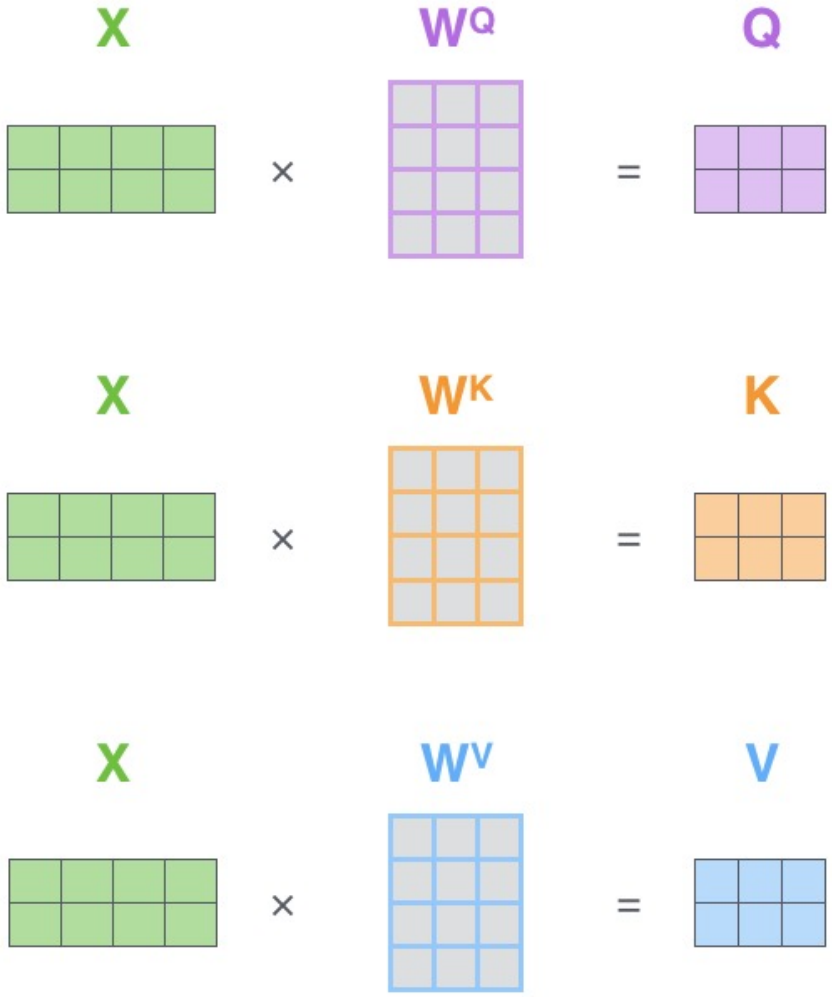
Self-Attention in Detail



Self-Attention in Detail

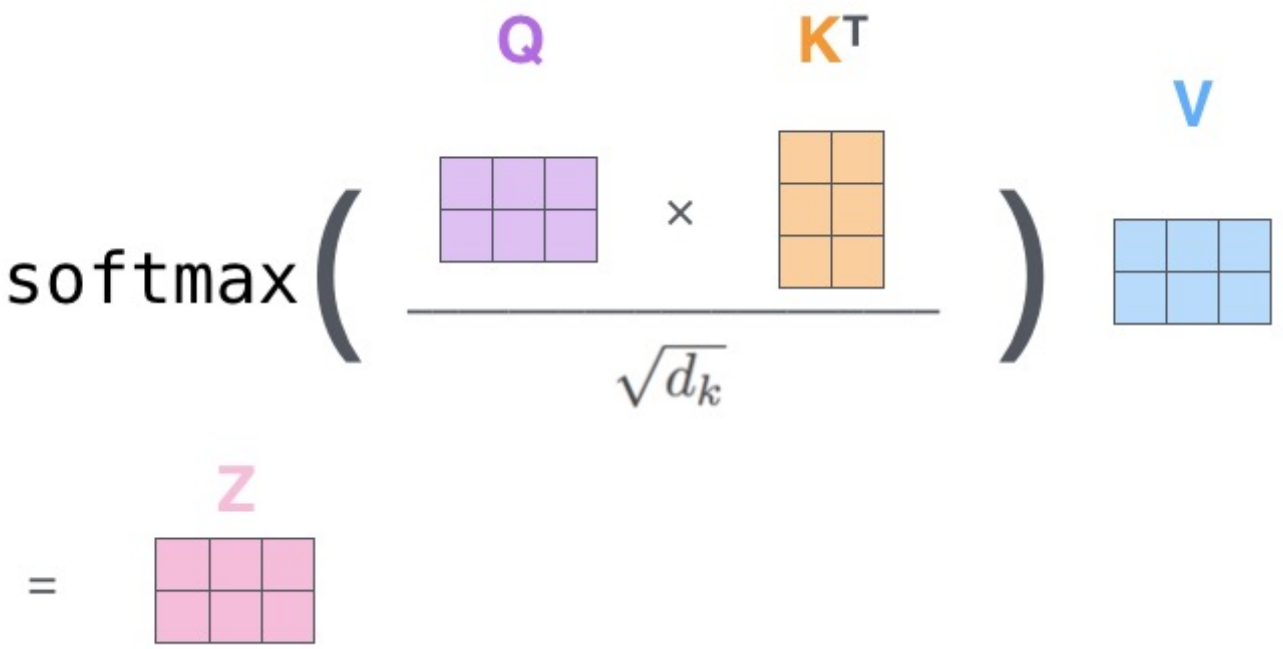


Matrix Calculation of Self-Attention

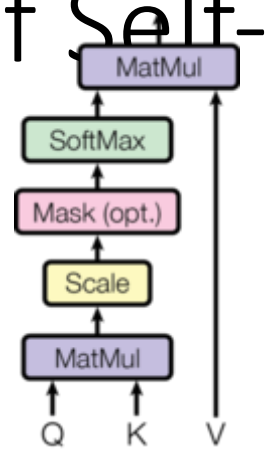


Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

Matrix Calculation of Self-Attention

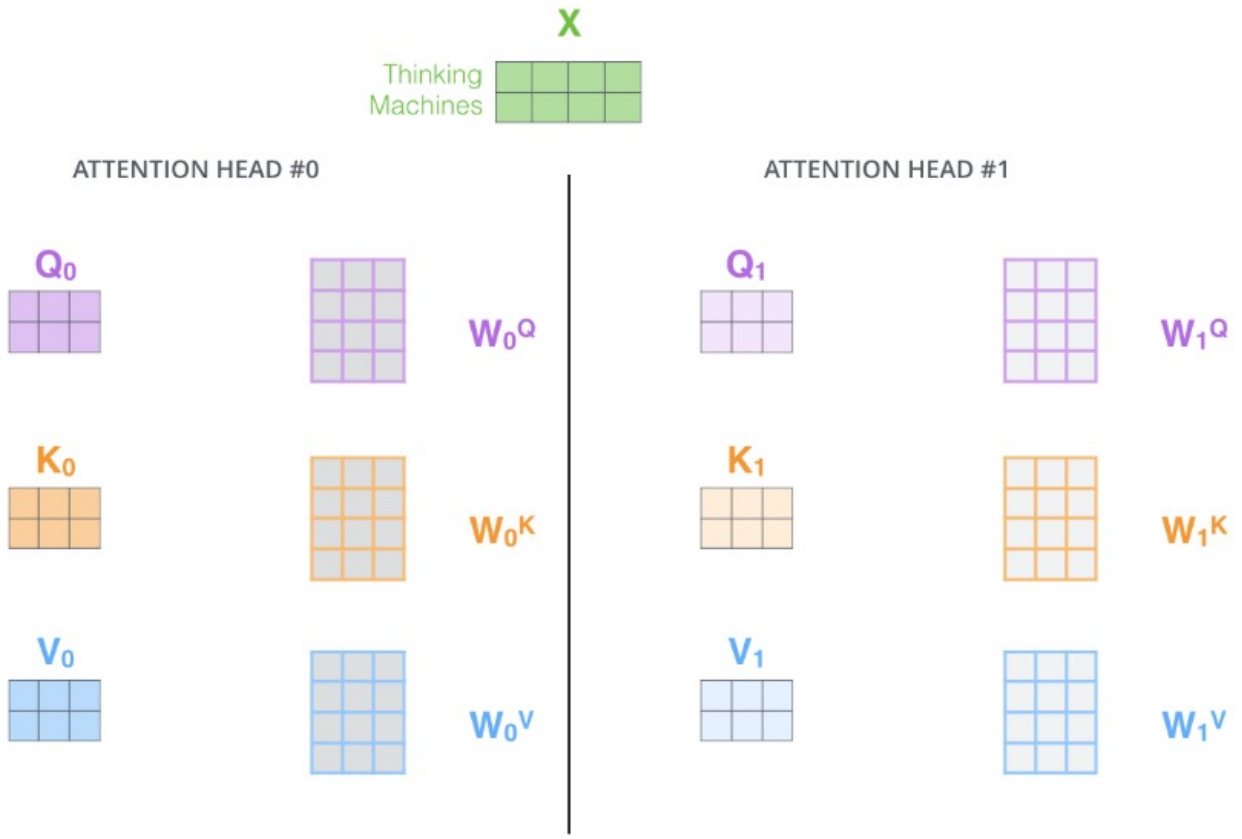


Matrix Calculation of Self-Attention



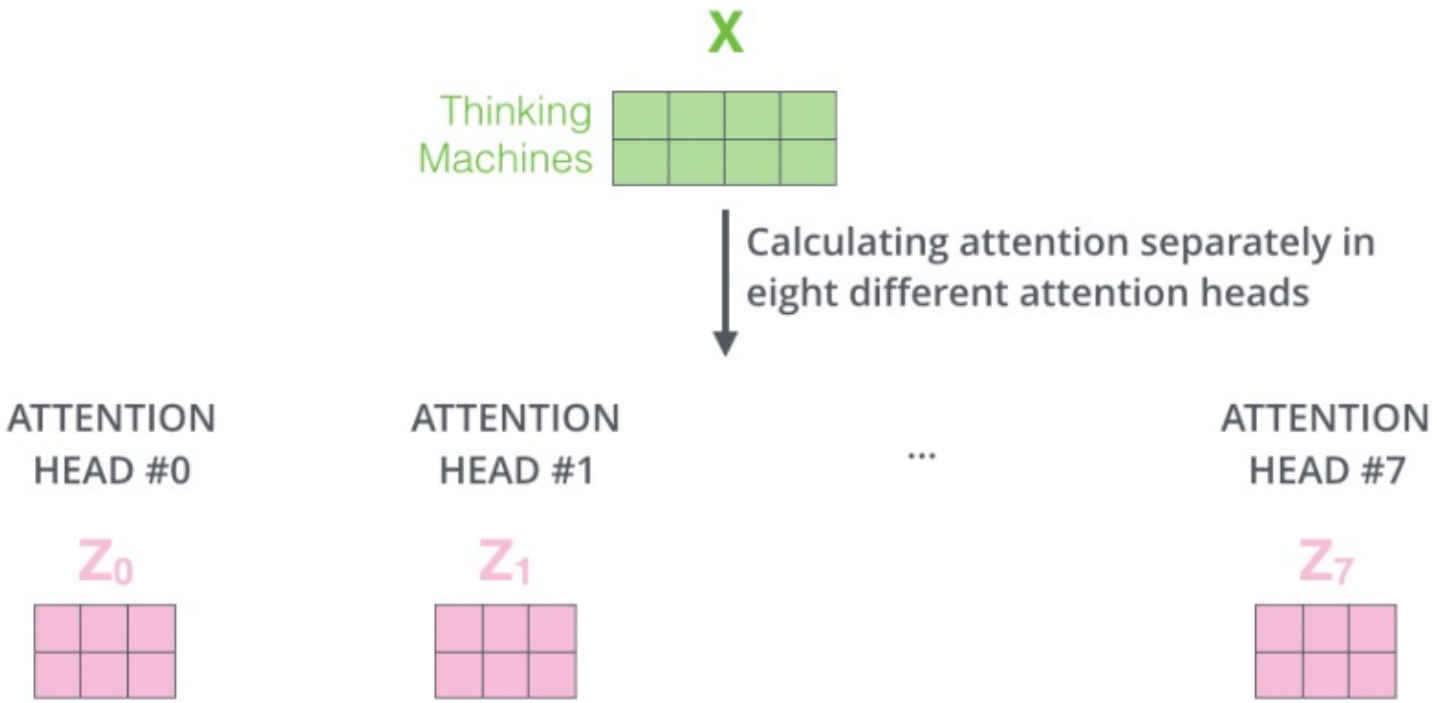

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

Multi-head Attention



With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

Multi-head Attention



Multi-head Attention

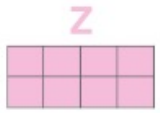
1) Concatenate all the attention heads

2) Multiply with a weight matrix W^O that was trained jointly with the model

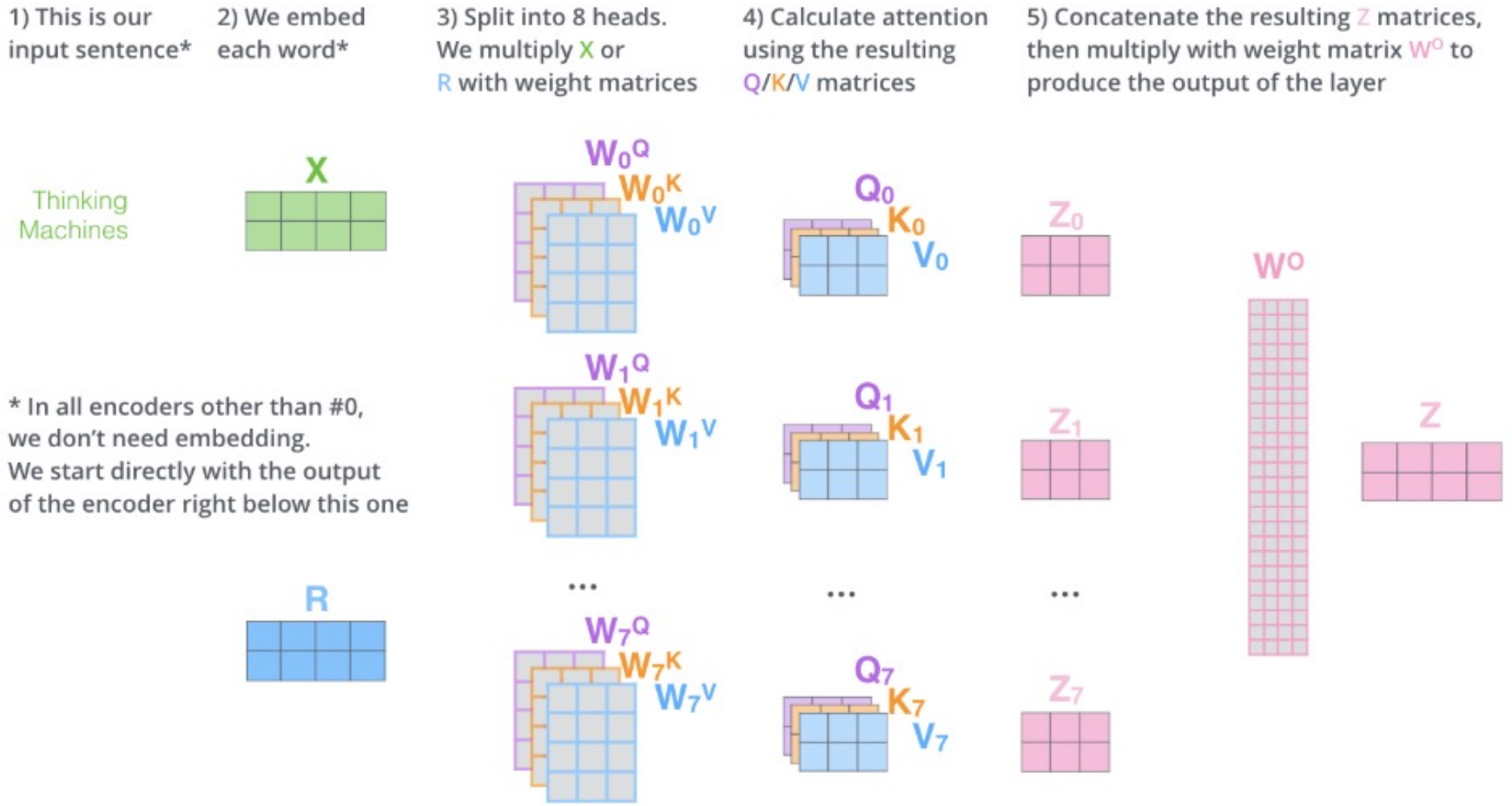
X

3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

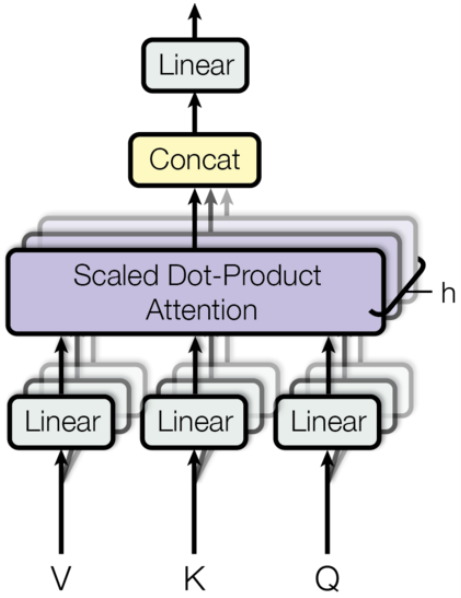
=



Multi-head Attention

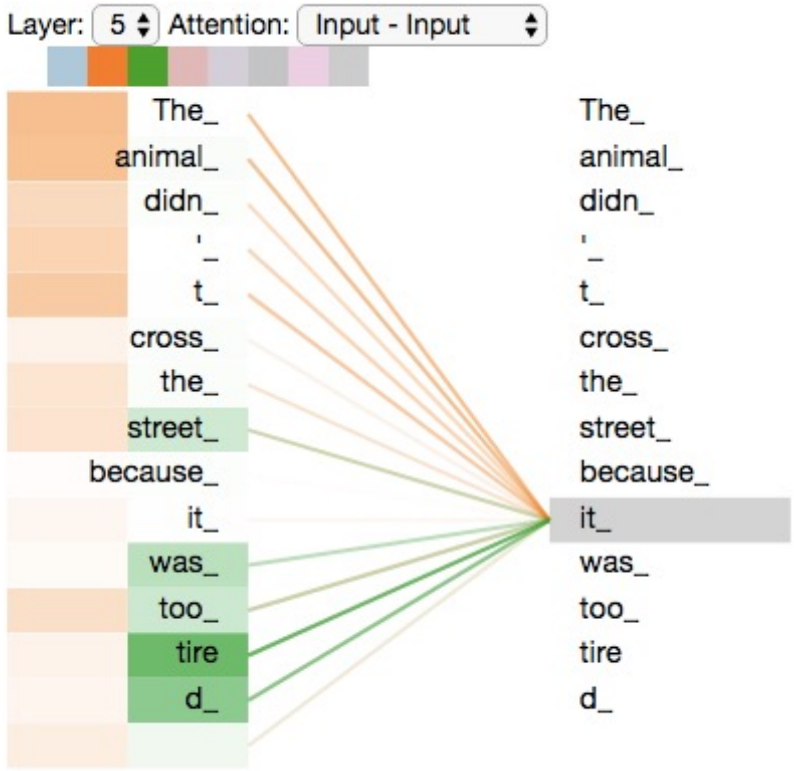


Multi-head Attention



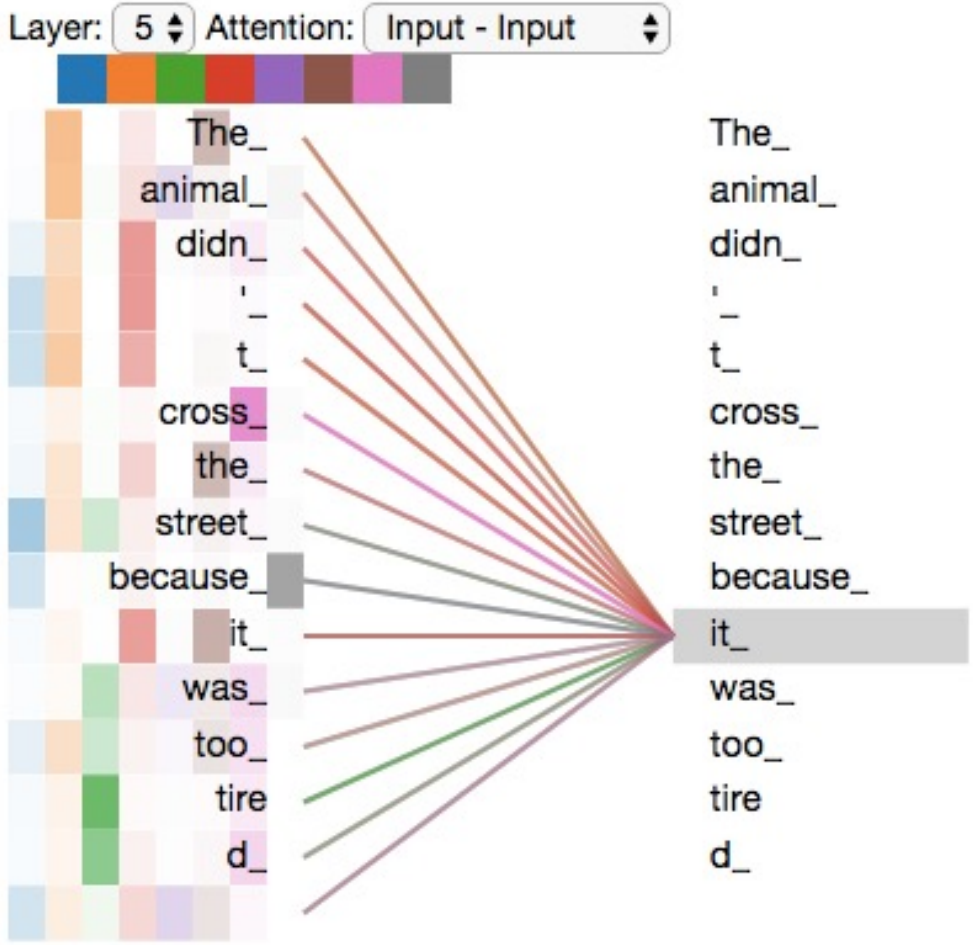

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous() \
             .view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

Multi-head Attention

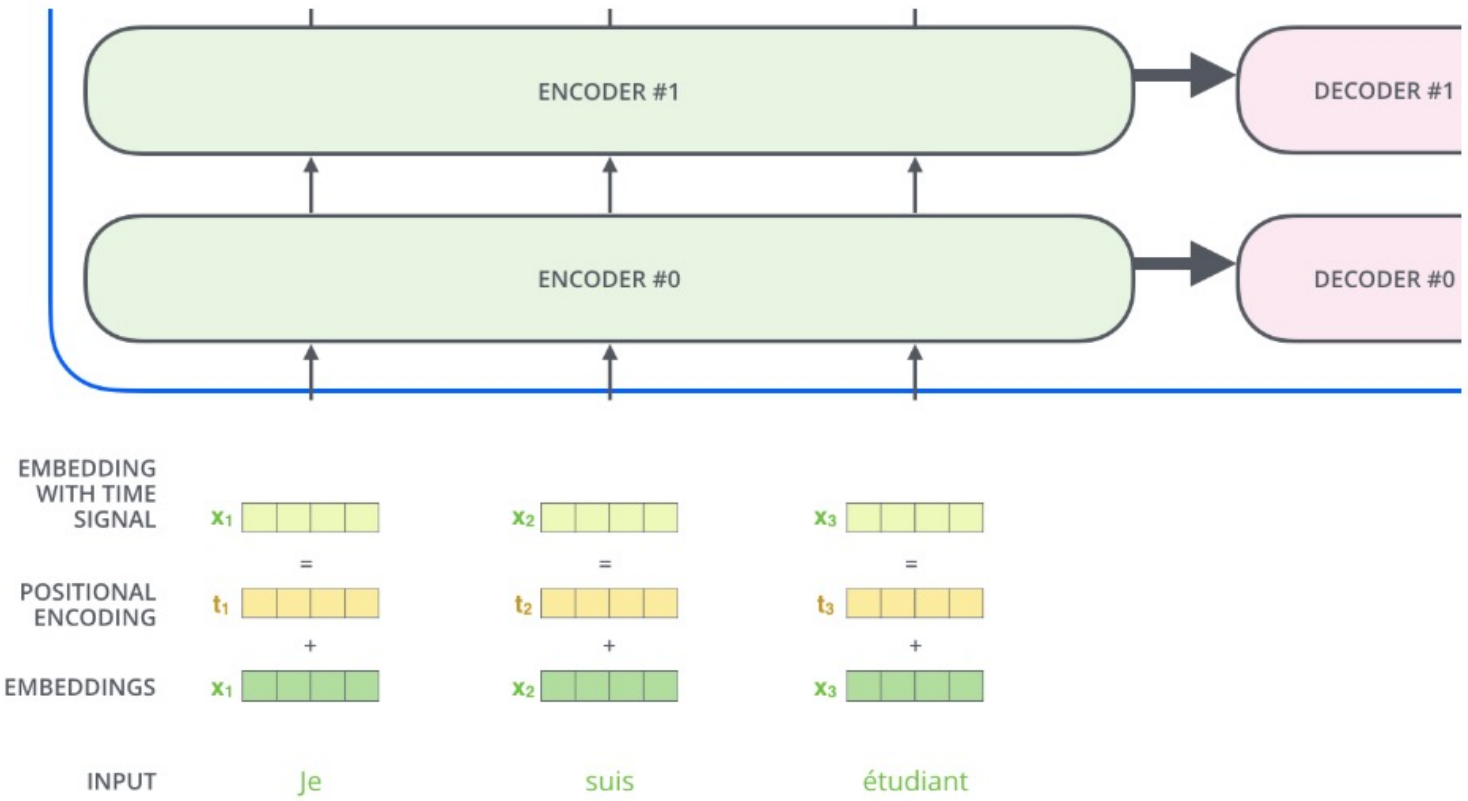


As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

Multi-head Attention

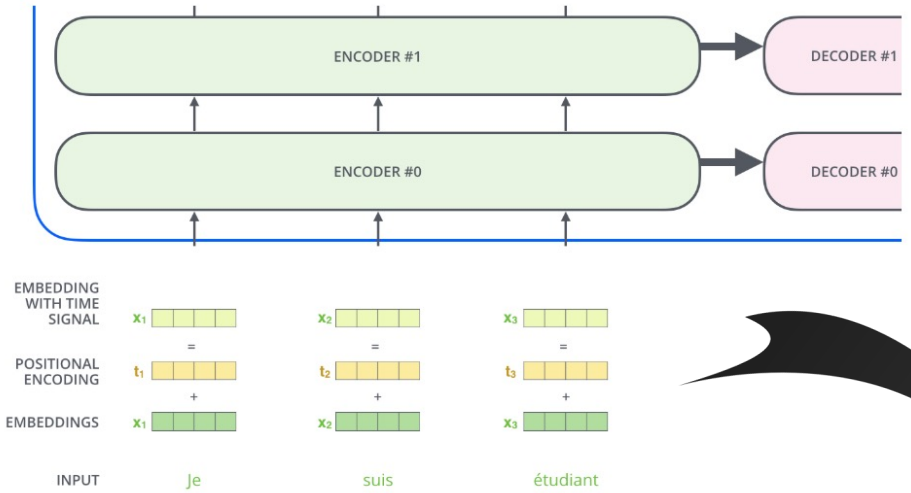


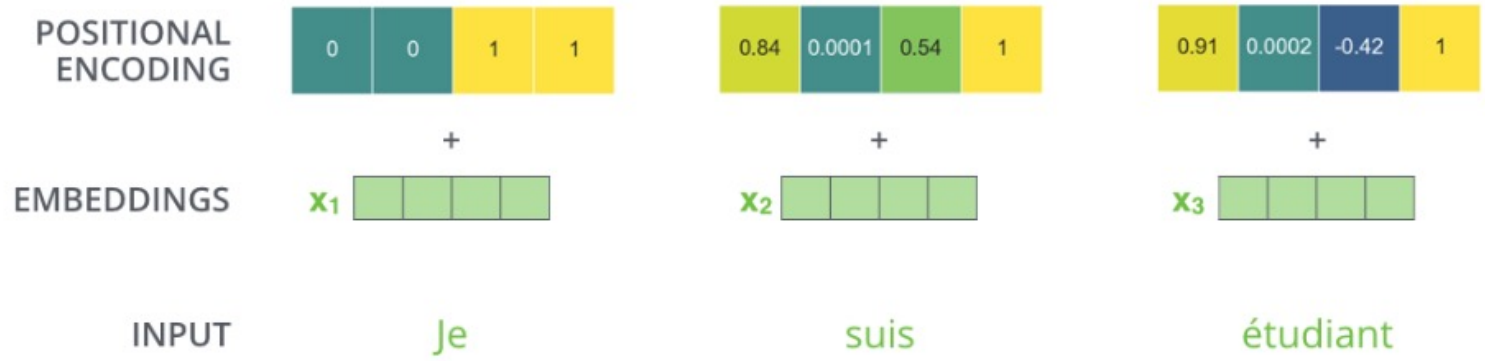
Representing The Order of The Sequence Using P



To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

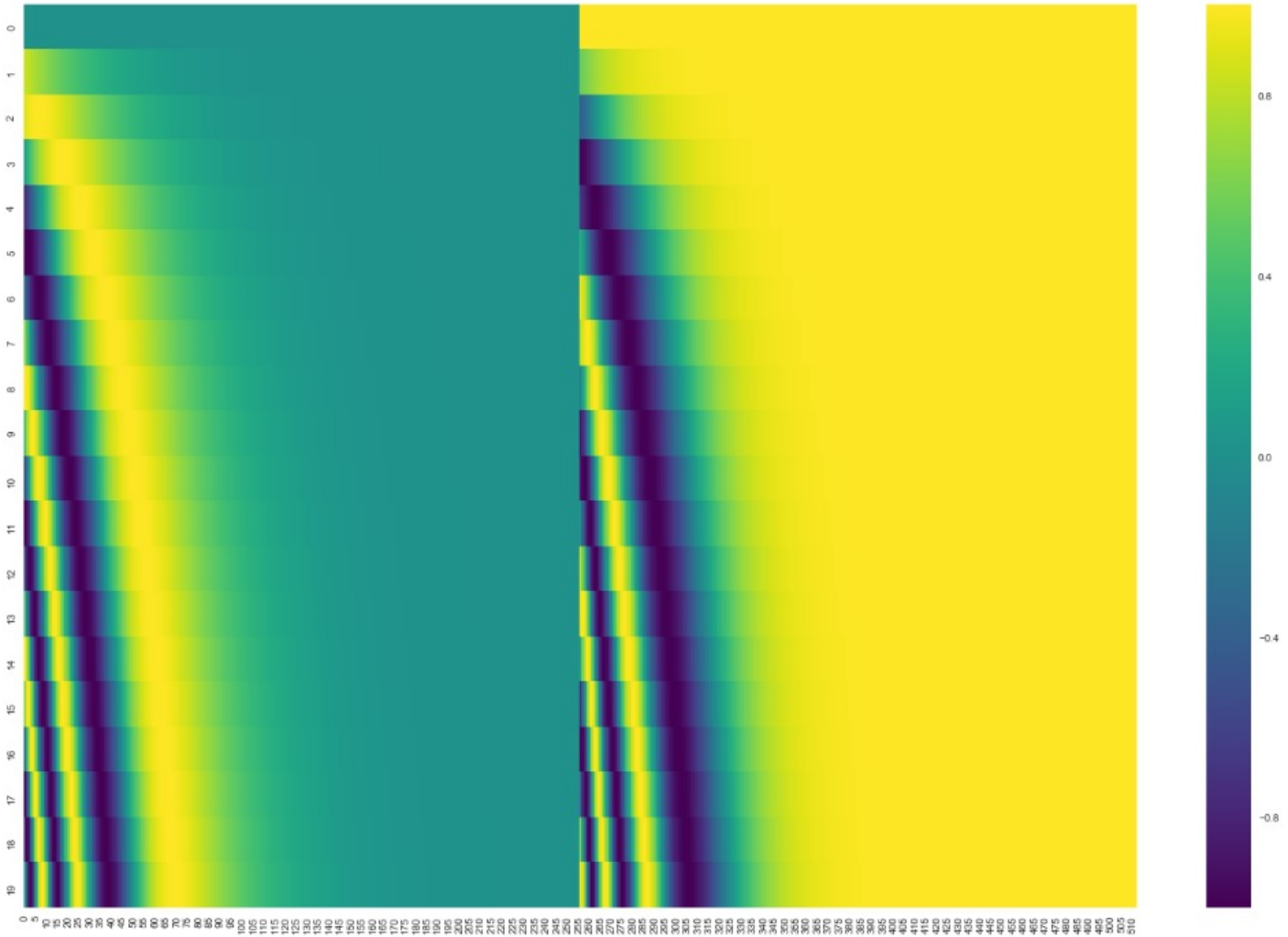
Representing The Order of The Sequence Usin g



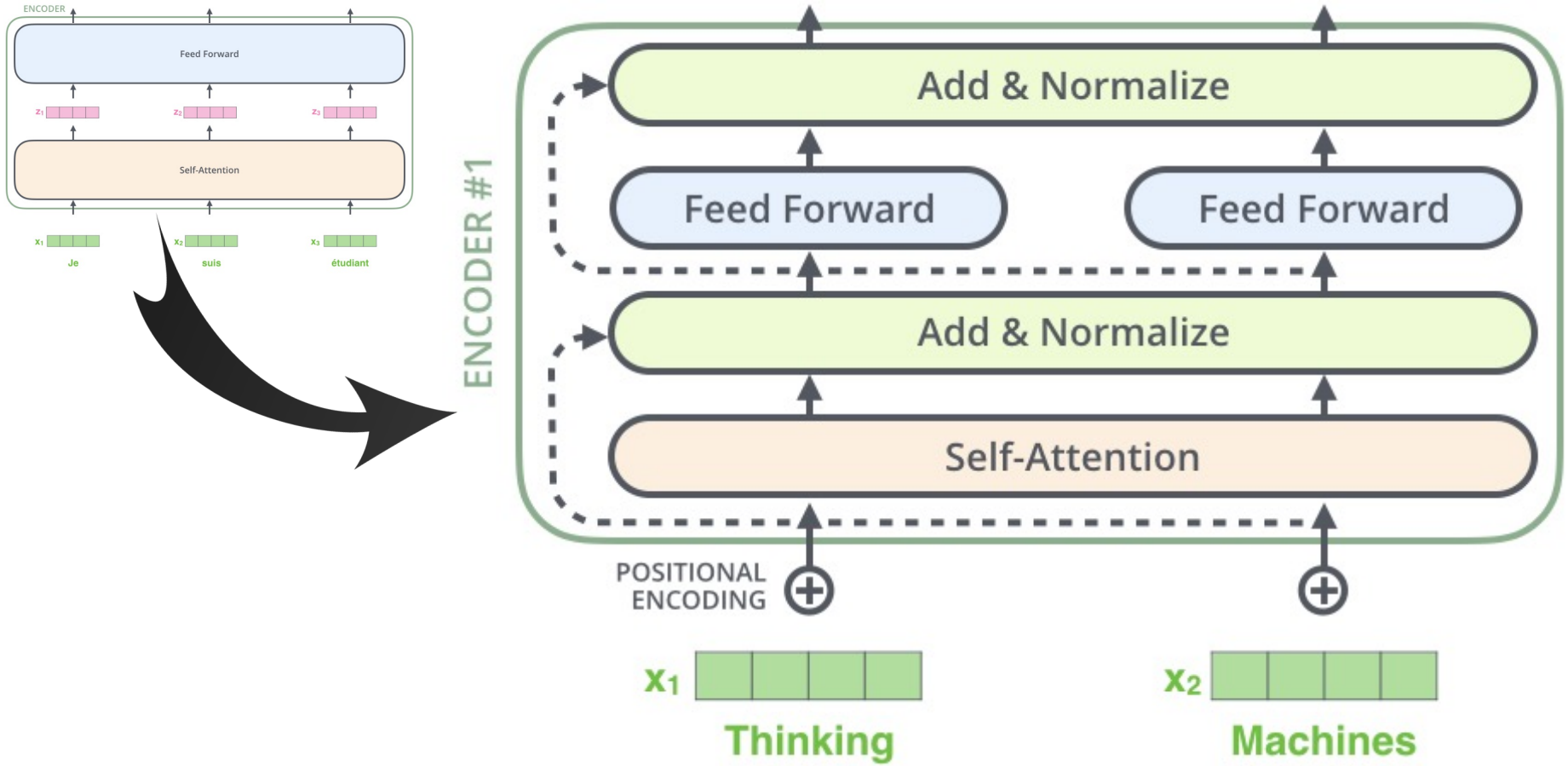


A real example of positional encoding with a toy embedding size of 4

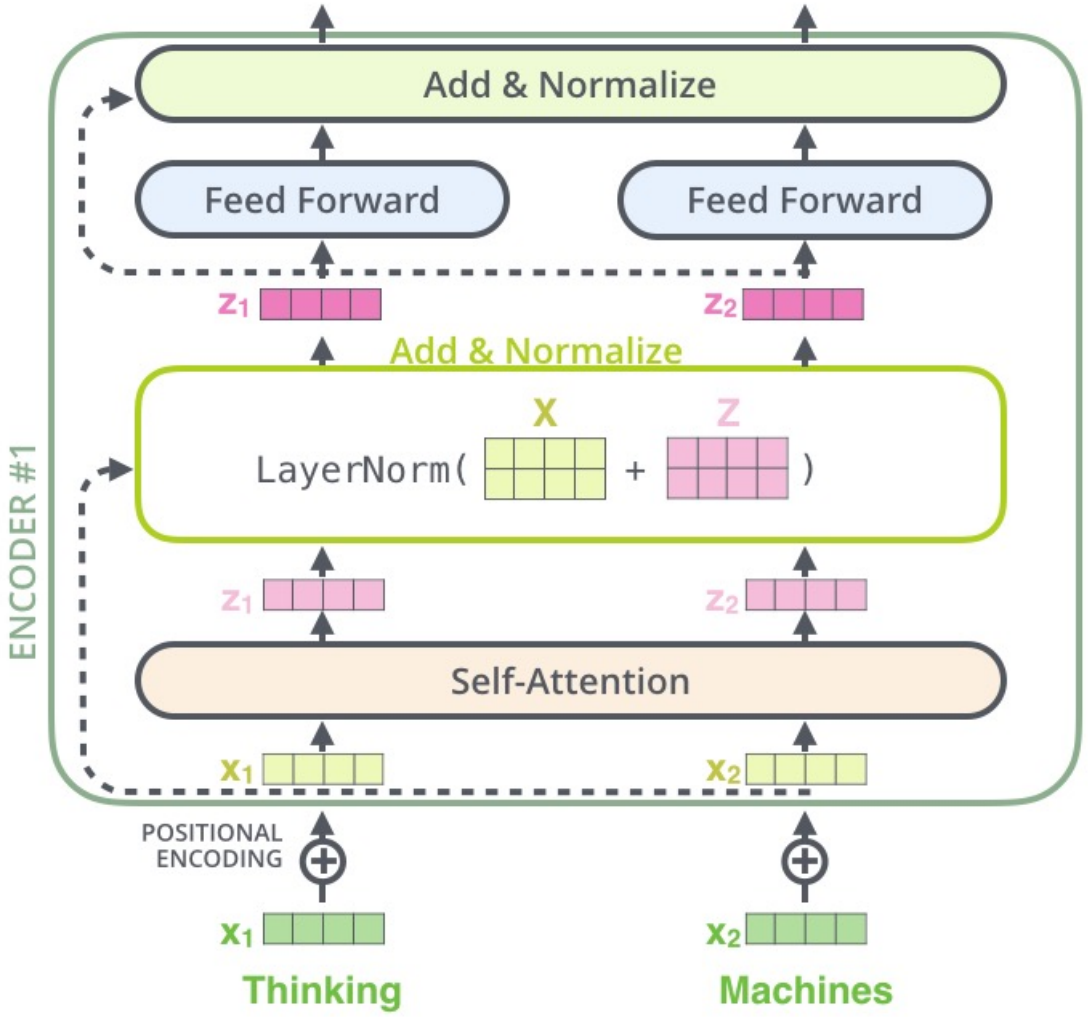
Representing The Order of The Sequence Using Positional Encoding



The Residuals



The Residuals

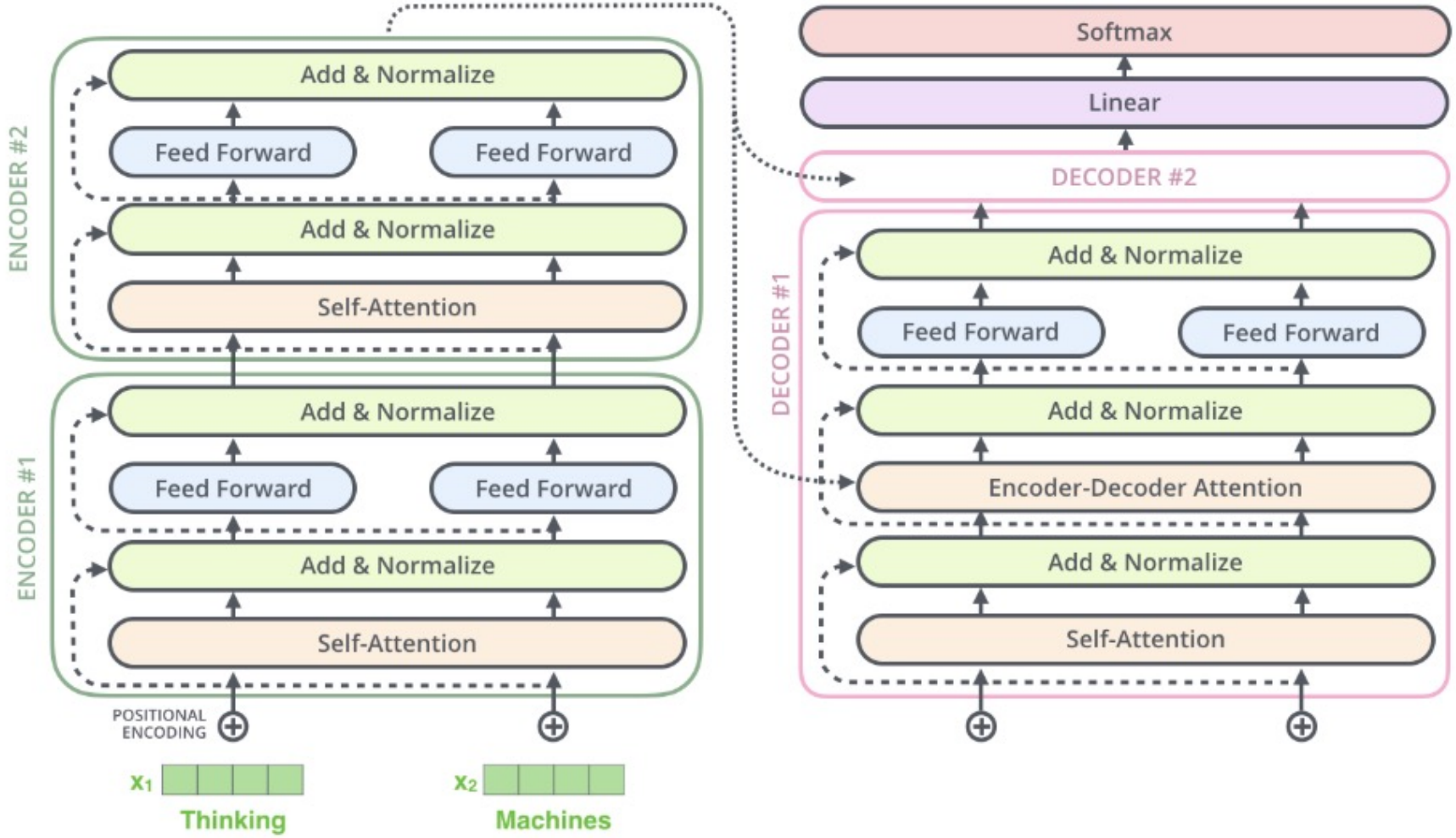


The Residuals

```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

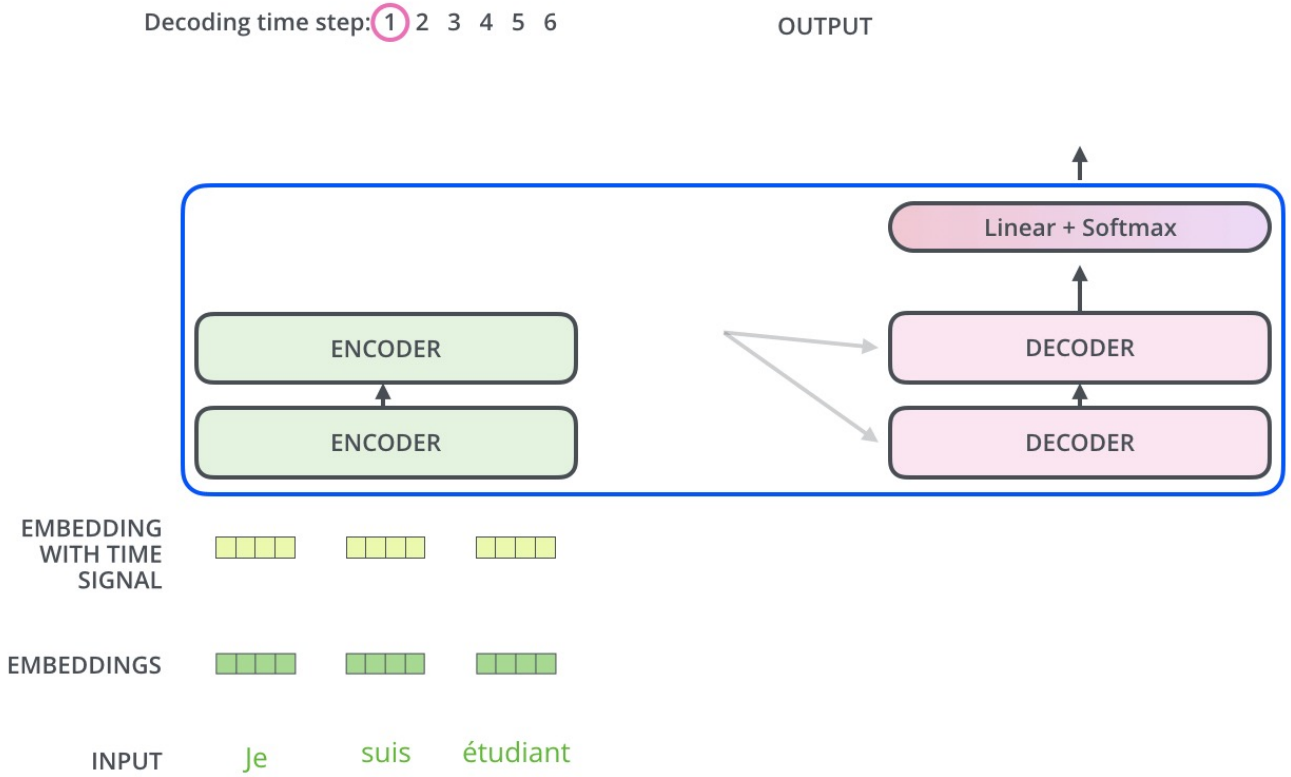
The Residuals



The Residuals

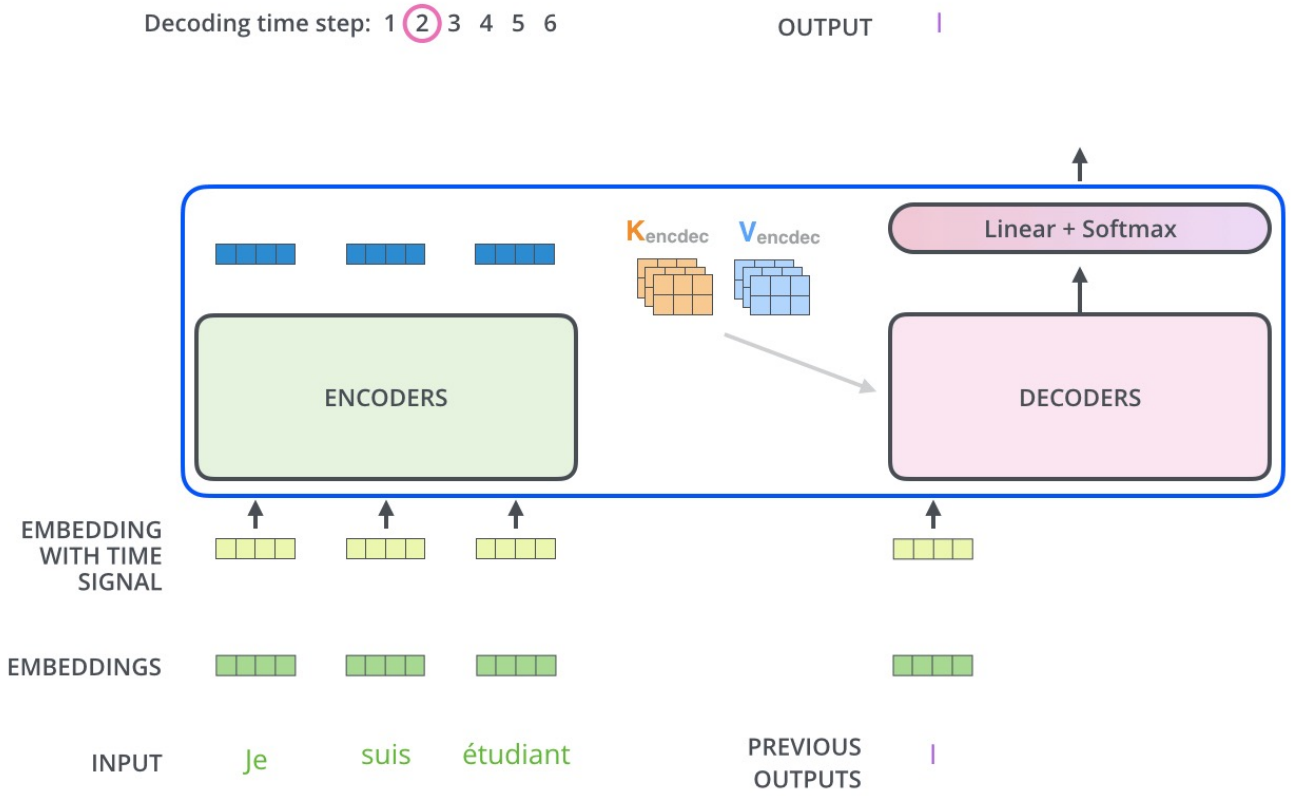
```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

The Decoder Side



After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

The Decoder Side

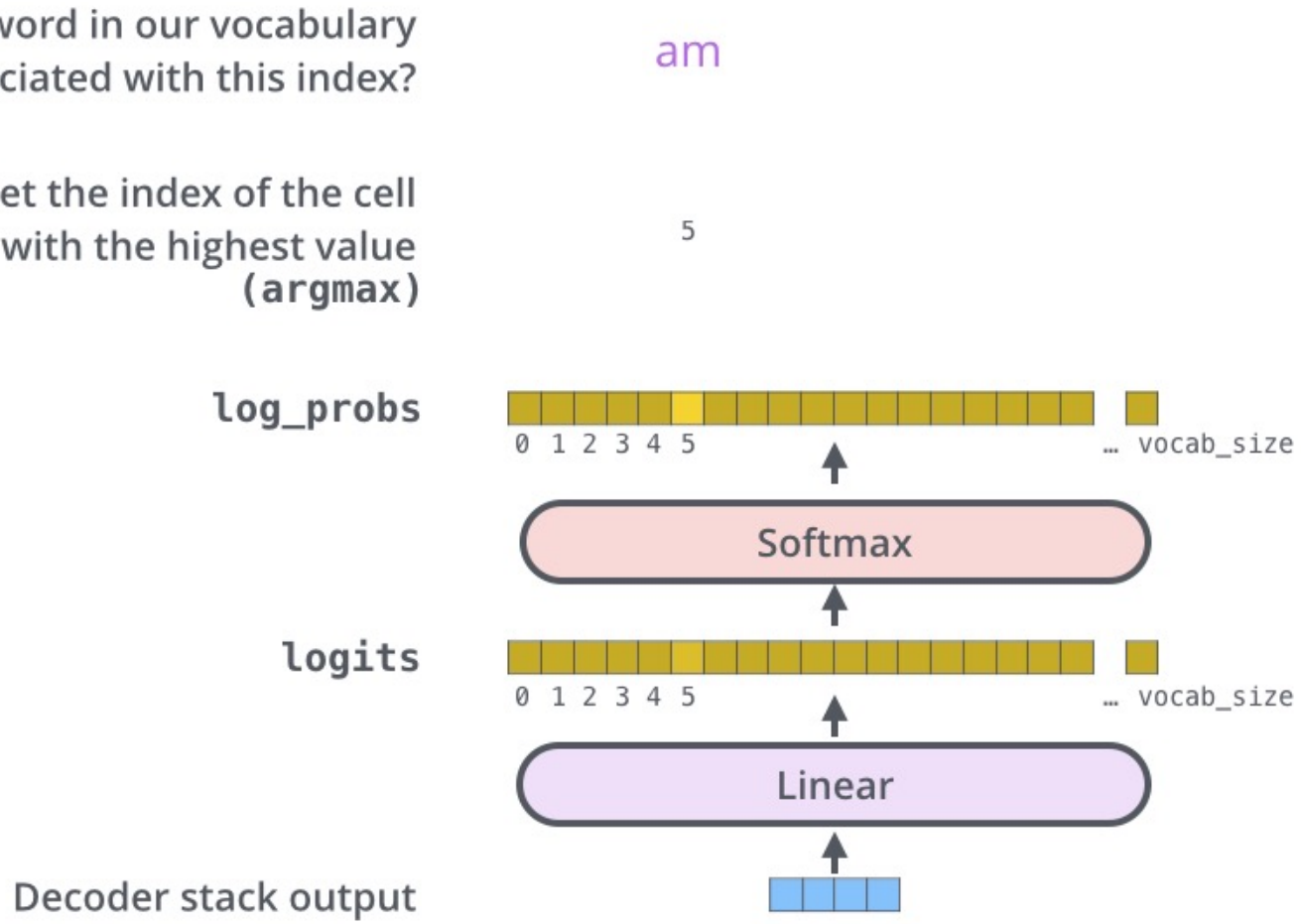


The Decoder Side

```
class Decoder(nn.Module):
    "Generic N layer decoder with masking."
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)
```

The Final Linear and Softmax Layer



This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

Training Procedure

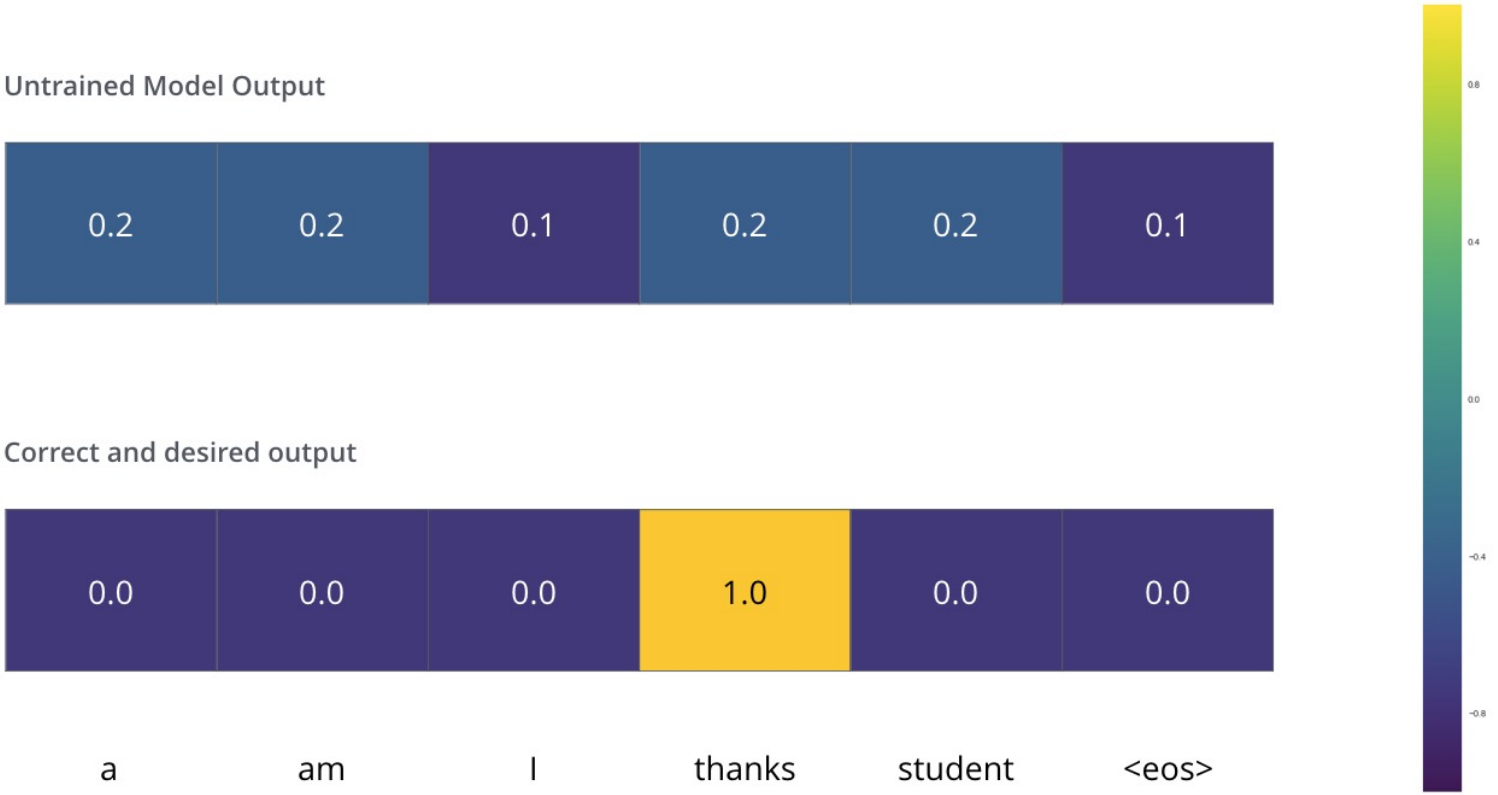
Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word “am”

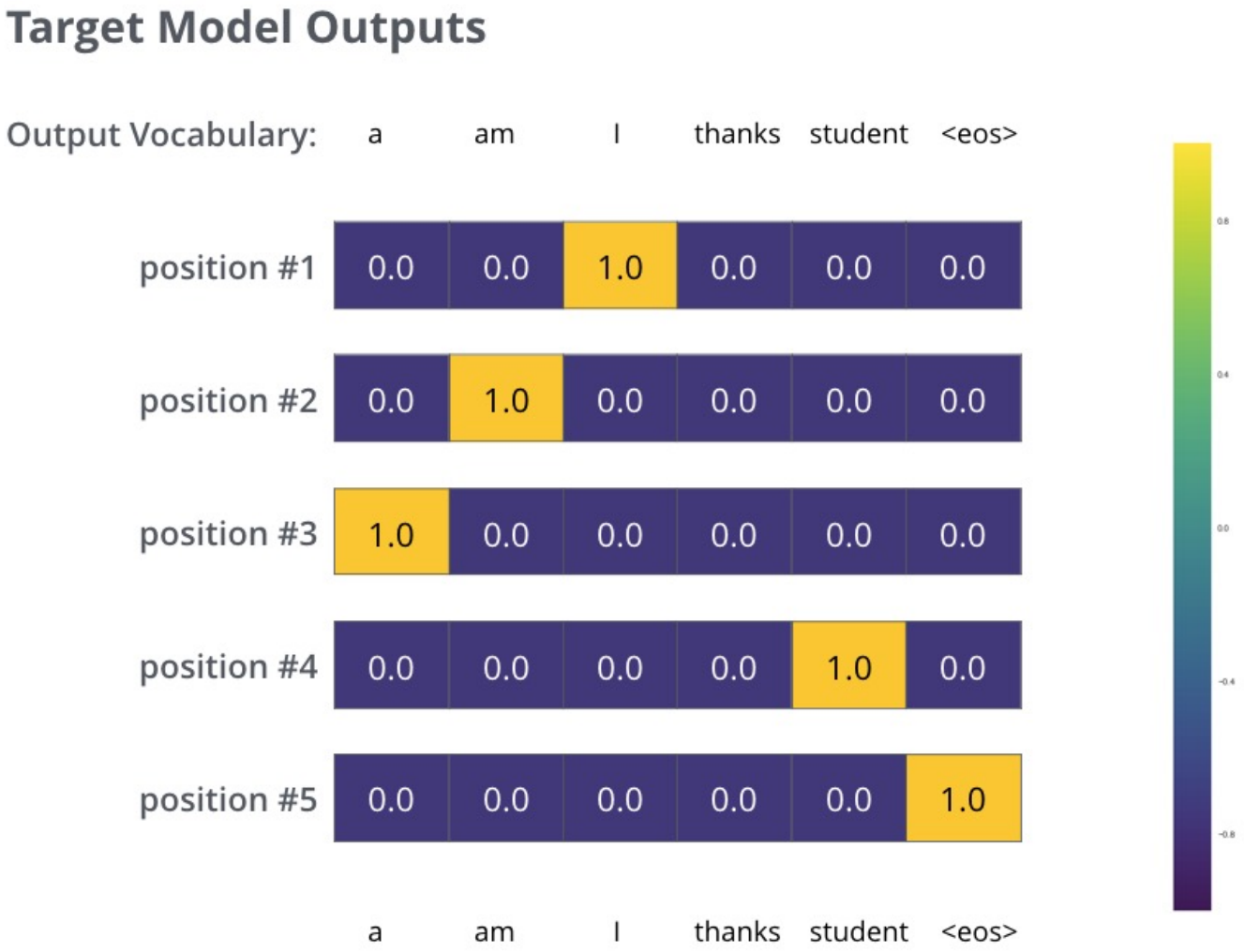
0.0	1.0	0.0	0.0	0.0	0.0

The Loss Function

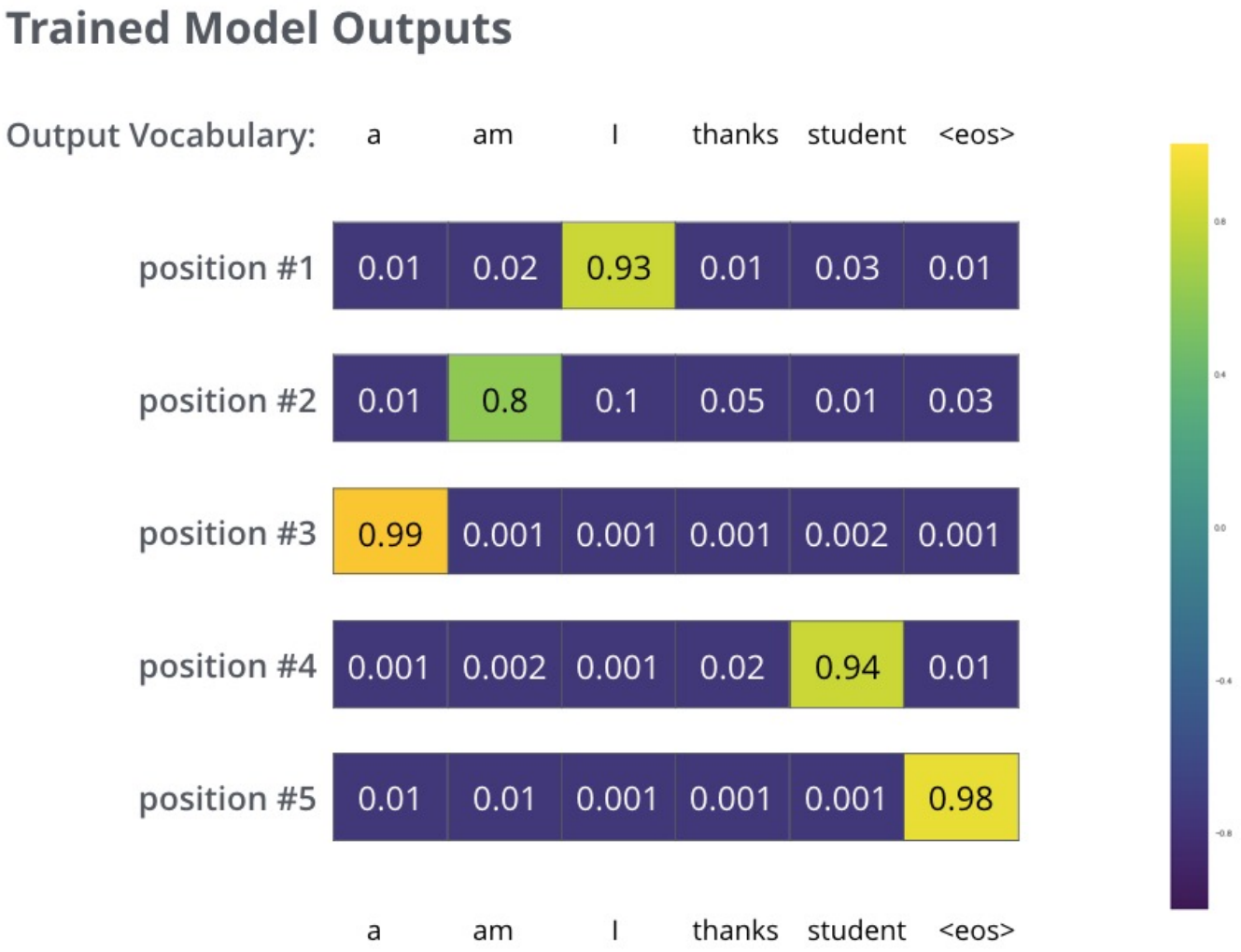


Since the model's parameters (weights) are all initialized randomly, the (untrained) model produces a probability distribution with arbitrary values for each cell/word. We can compare it with the actual output, then tweak all the model's weights using backpropagation to make the output closer to the desired output.

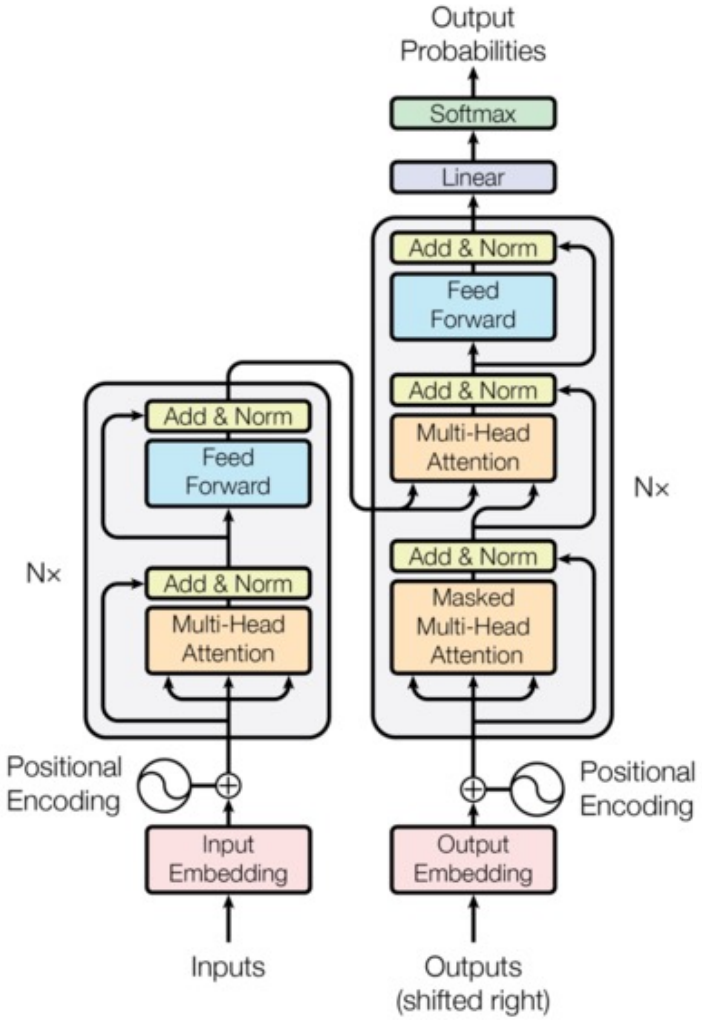
The Loss Function



The Loss Function



Wrap up!

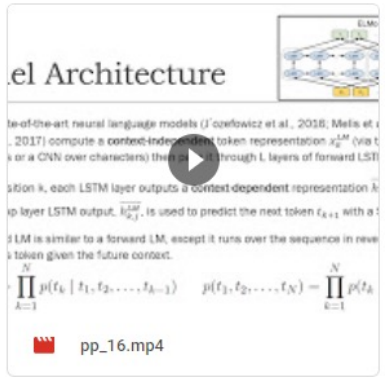

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

References

- The Illustrated Transformer
  - http://jalammar.github.io/illustrated-transformer/
- The Annotated Transformer
  - http://nlp.seas.harvard.edu/2018/04/03/attention.html
- Transformer Neural Networks - EXPLAINED! (Attention is all you need)
  - https://www.youtube.com/watch?v=TQQlZhbC5ps

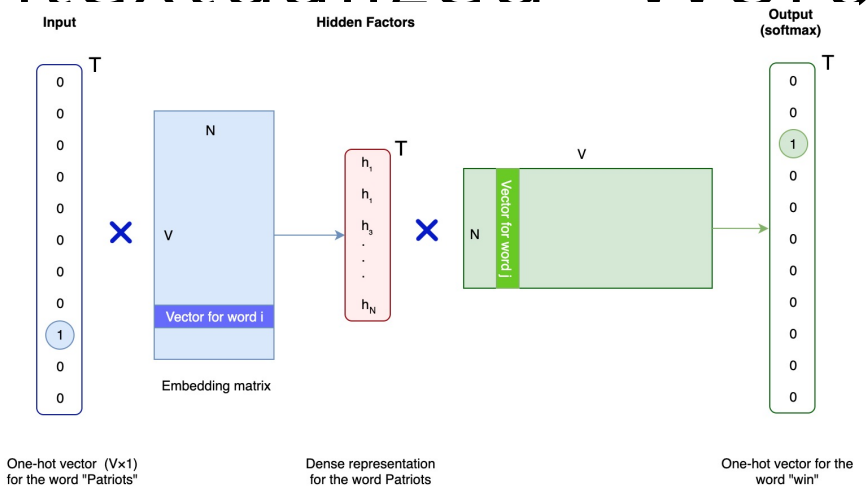
ELMo

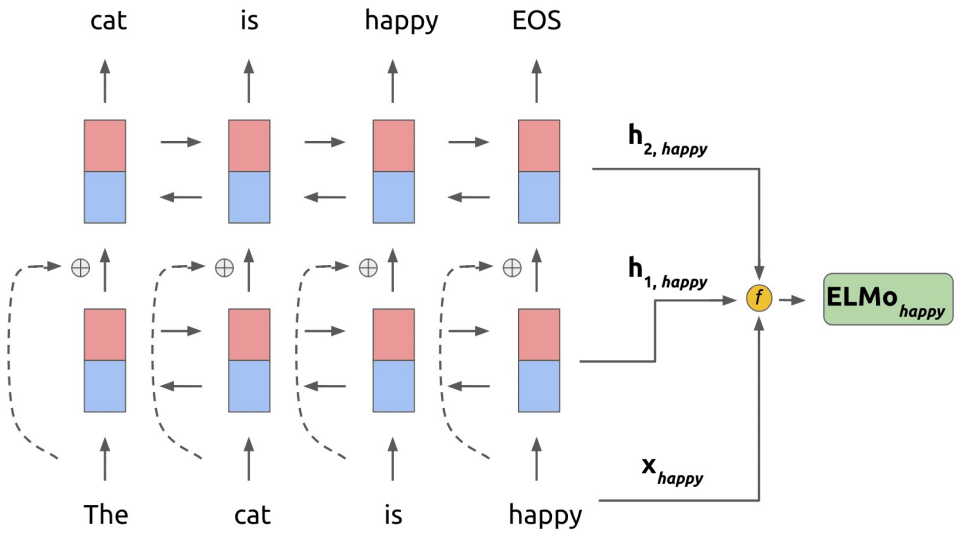

Hey ELMo, what's the embedding of the word "stick"?
There are multiple possible embeddings! Use it in a sentence.
Oh, okay. Here: "Let's stick to improvisation in this skit"
Oh in that case, the embedding is: -0.02, -0.16, 0.12, -0.1 ….etc


el Architecture
pp_16.mp4

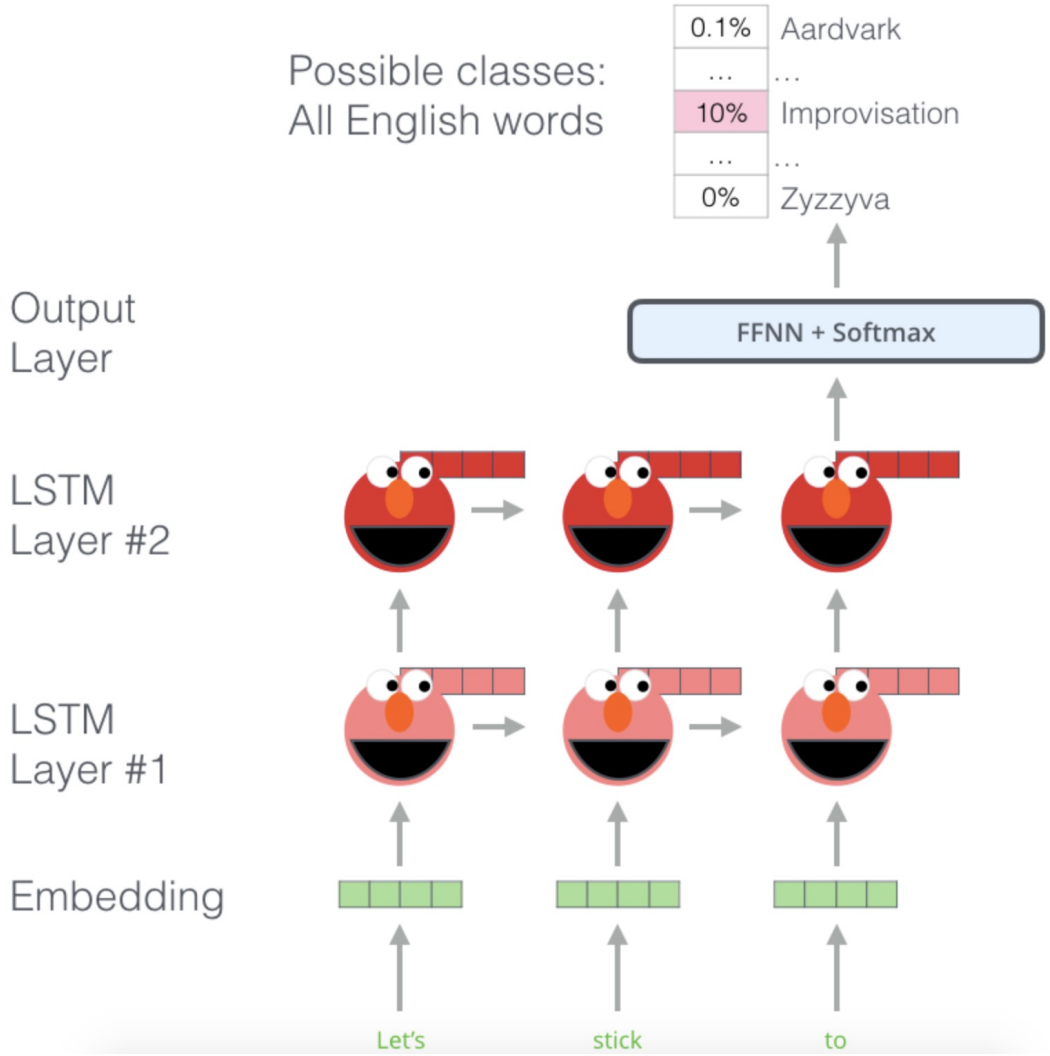

Deep
Conte
Word
Repre
2019521112
DONGYUP SH
pp_16.pdf

“Contextualized” Word Embedding

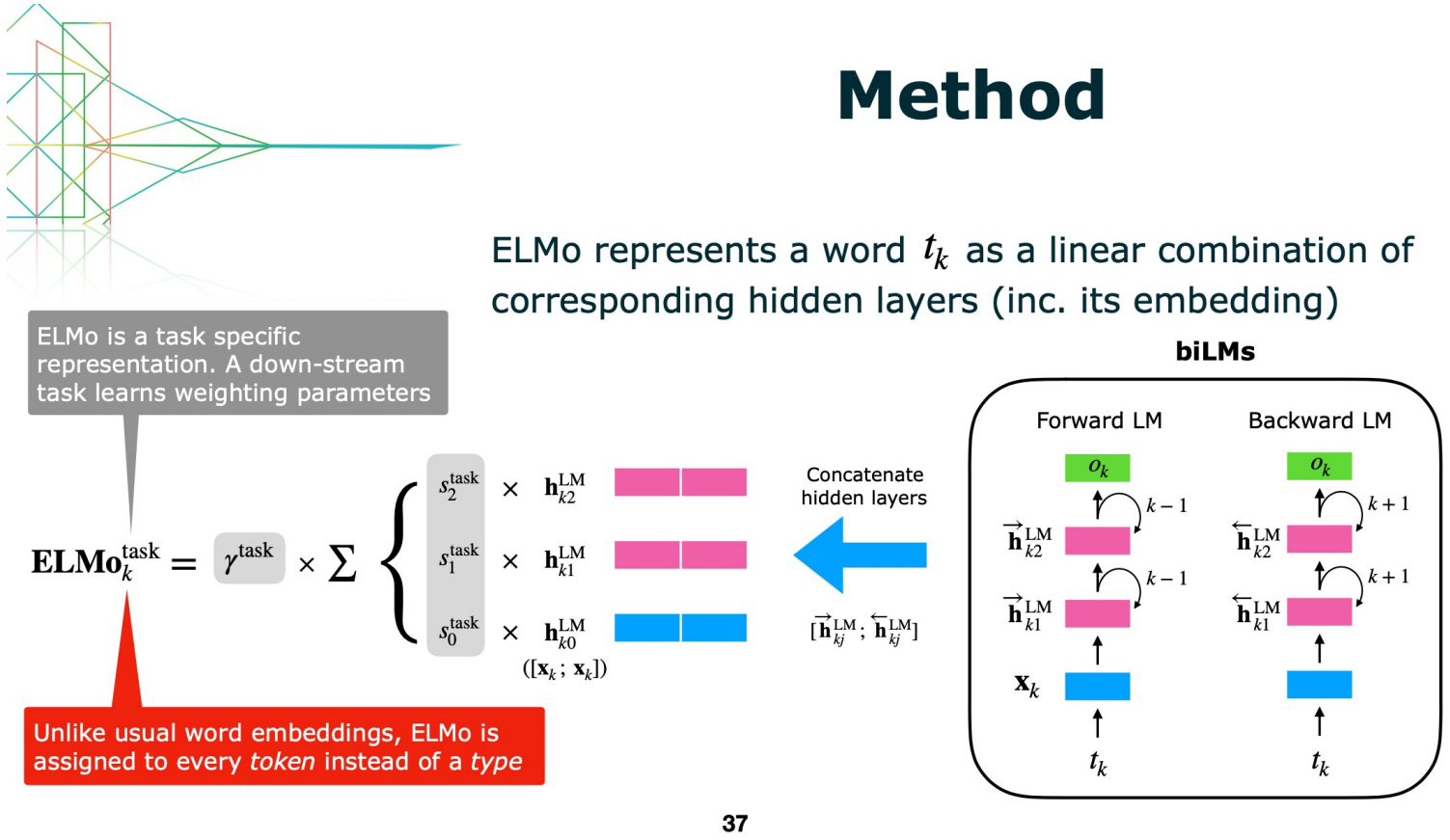




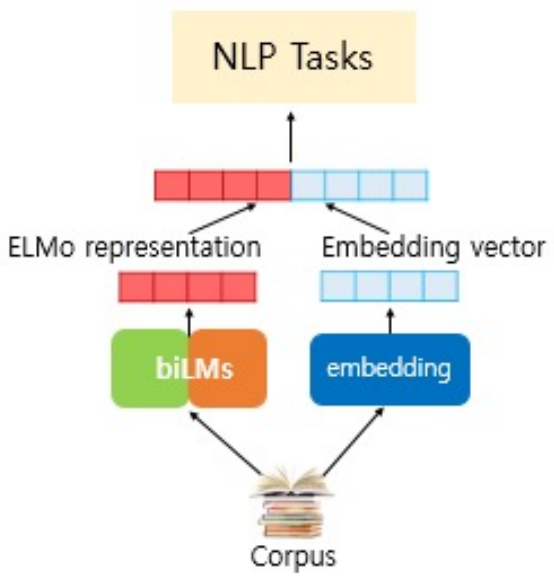
(Pre-)Training procedure of ELMo



(Pre-)Training procedure of ELMo



How to use ELMo when after pre-training



28 Nov 03, 2017	BiDAF + Self Attention + ELMo (single model) *Allen Institute for Artificial Intelligence*	78.580	85.833
46 Sep 20, 2017	BiDAF + Self Attention (single model) *Allen Institute for Artificial Intelligence* https://arxiv.org/abs/1710.10723	72.139	81.048

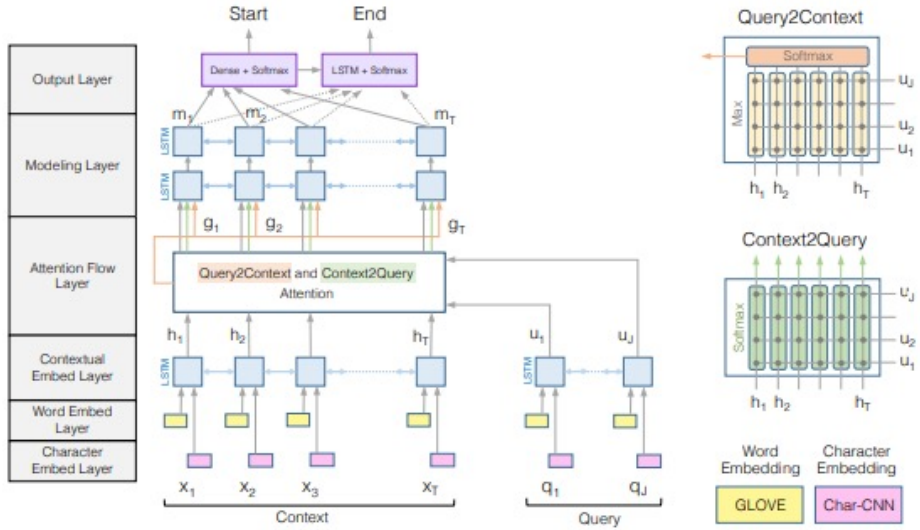


Figure 1: BiDirectional Attention Flow Model *(best viewed in color)*

How to use ELMo when after pre-training

Task	Previous SOTA		Our baseline	ELMo + Baseline	Increase (Absolute/Relative)
SQuAD	SAN	84.4	81.1	85.8	4.7 / 24.9%
SNLI	Chen et al (2017)	88.6	88.0	88.7 +/- 0.17	0.7 / 5.8%
SRL	He et al (2017)	81.7	81.4	84.6	3.2 / 17.2%
Coref	Lee et al (2017)	67.2	67.2	70.4	3.2 / 9.8%
NER	Peters et al (2017)	91.93 +/- 0.19	90.15	92.22 +/- 0.10	2.06 / 21%
Sentiment (5-class)	McCann et al (2017)	53.7	51.4	54.7 +/- 0.5	3.3 / 6.8%

Deep Pre-trained Language Models

- ~~2017 Jun: Transformer (not LM)~~

Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

- 2018 Feb: ELMo

Deep contextualized word representations

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer

- 2018 Jun: OpenAI GPT

Improving Language Understanding by Generative Pre-Training

- 2018 Oct: BERT

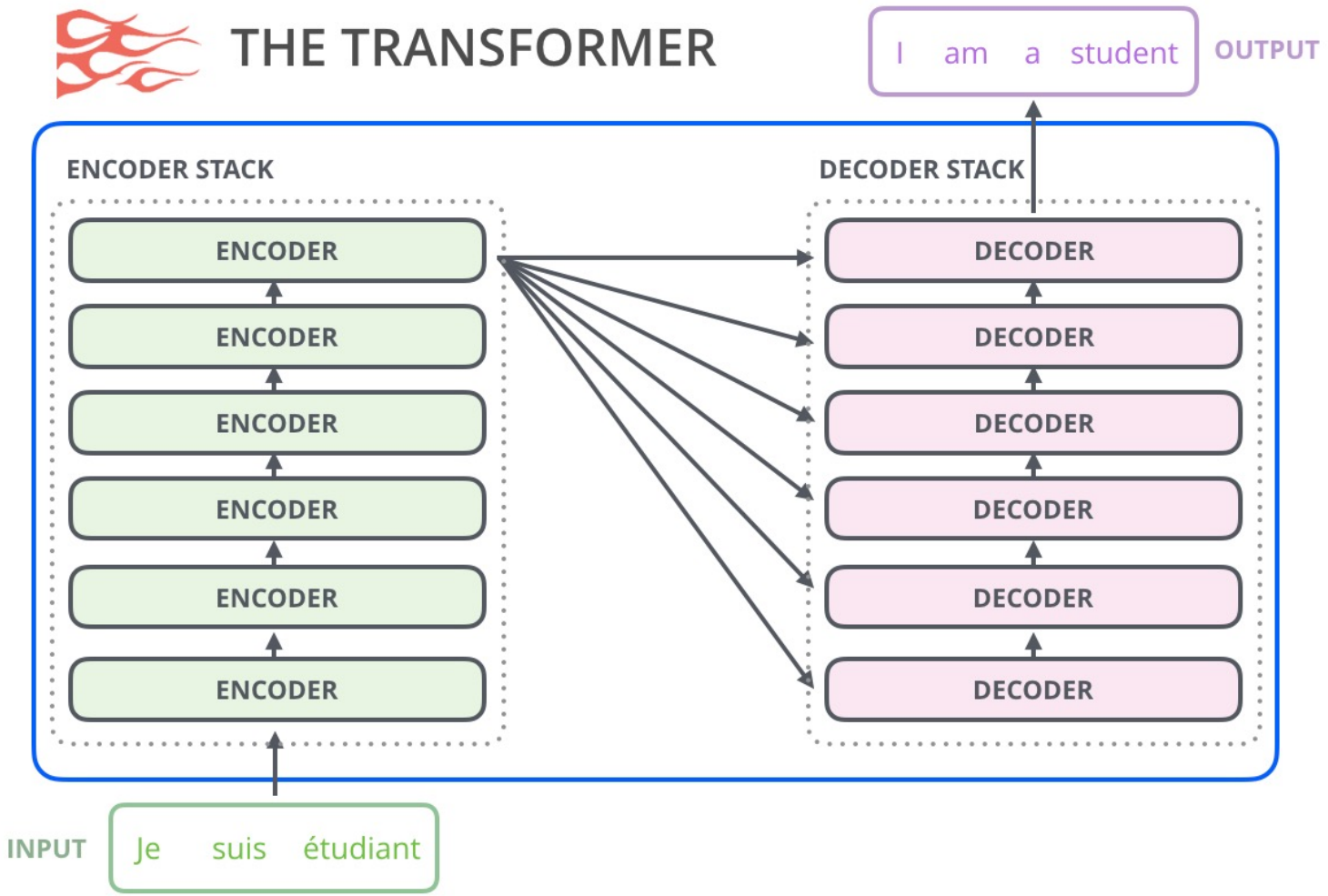
BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova

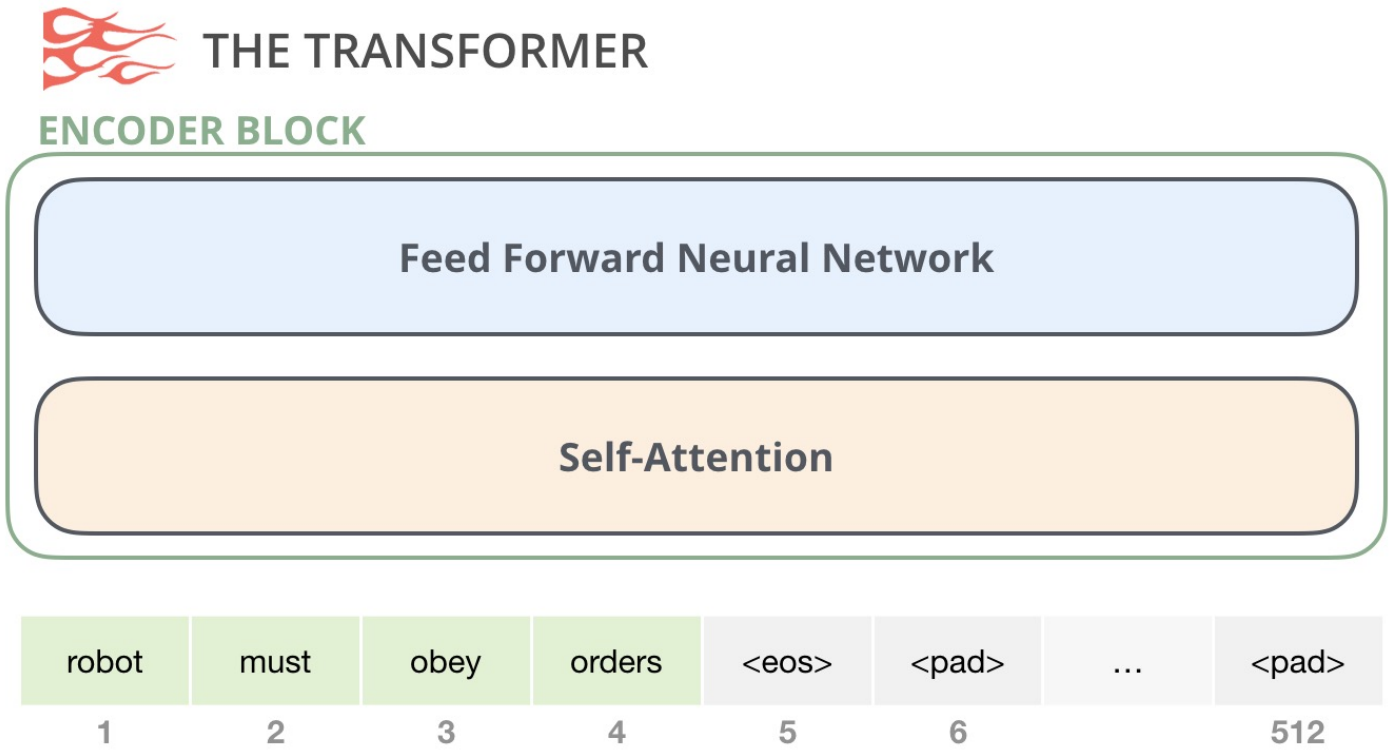
- 2019 Feb: OpenAI GPT-2

…

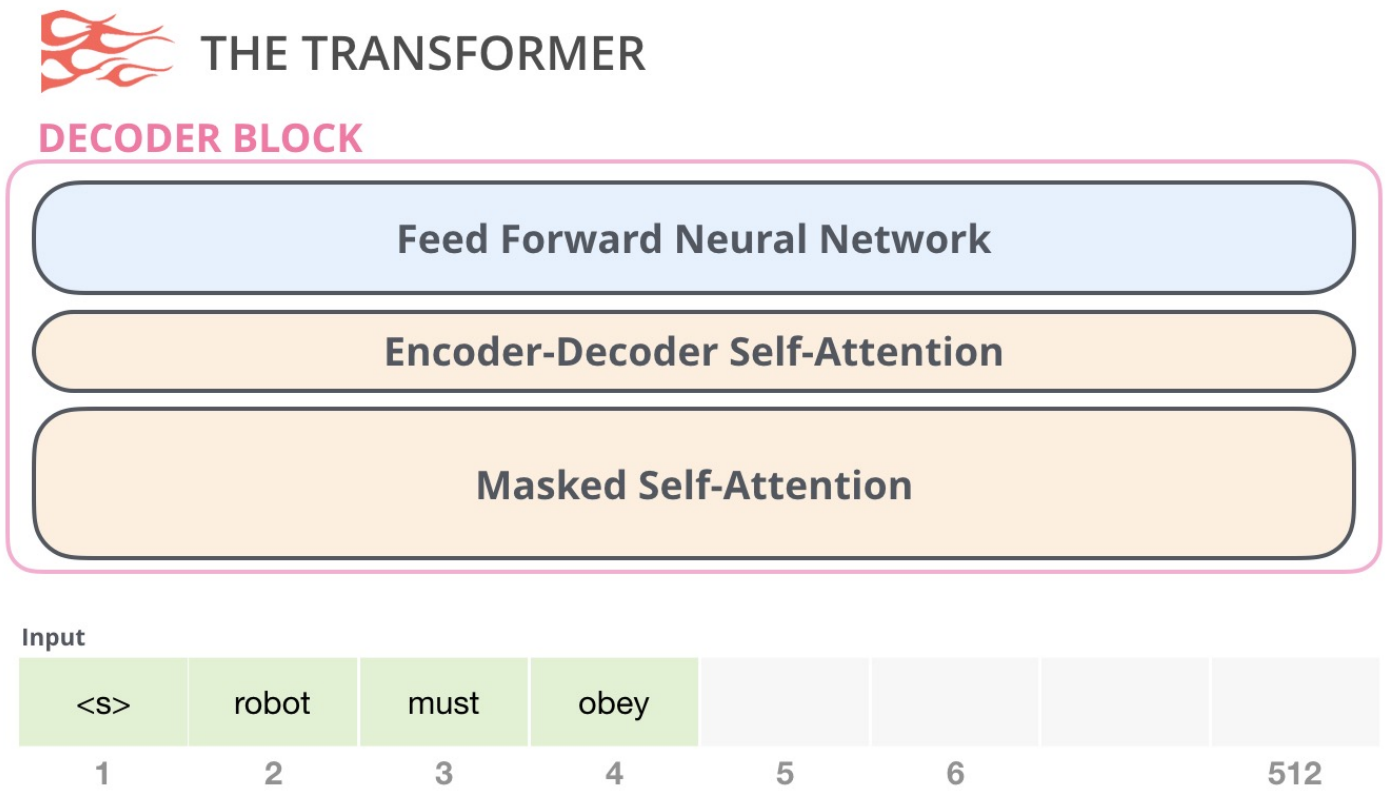
(Recap) Transformer



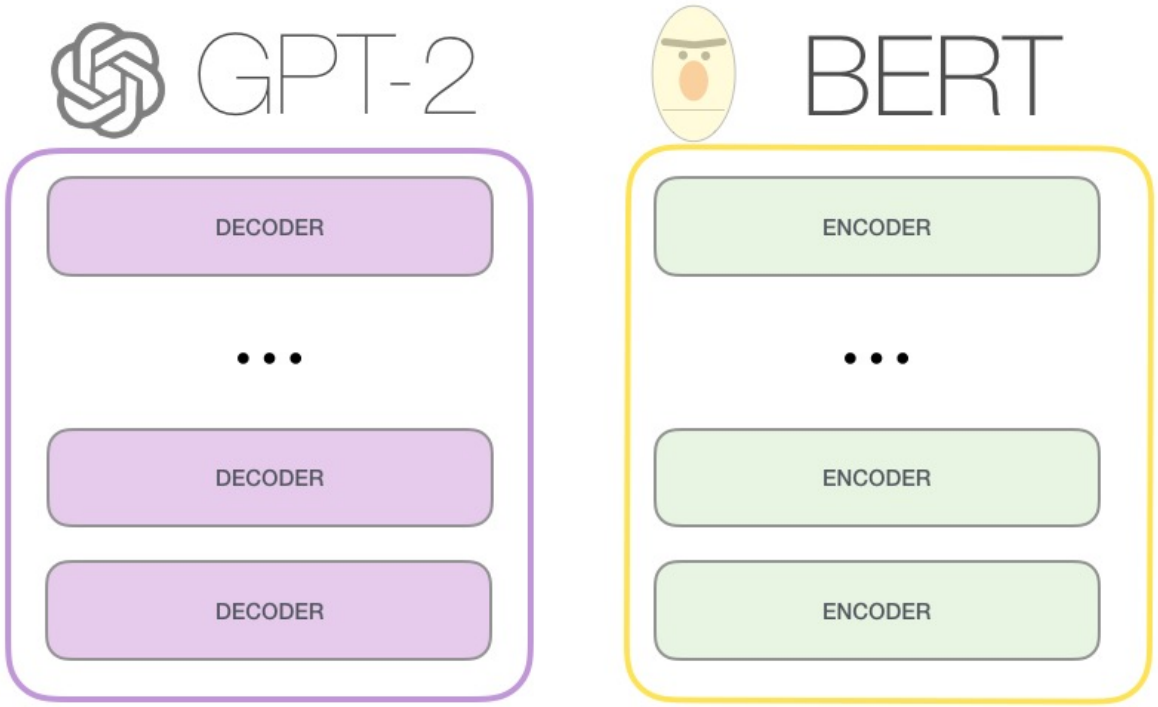
(Recap) Encoder Block



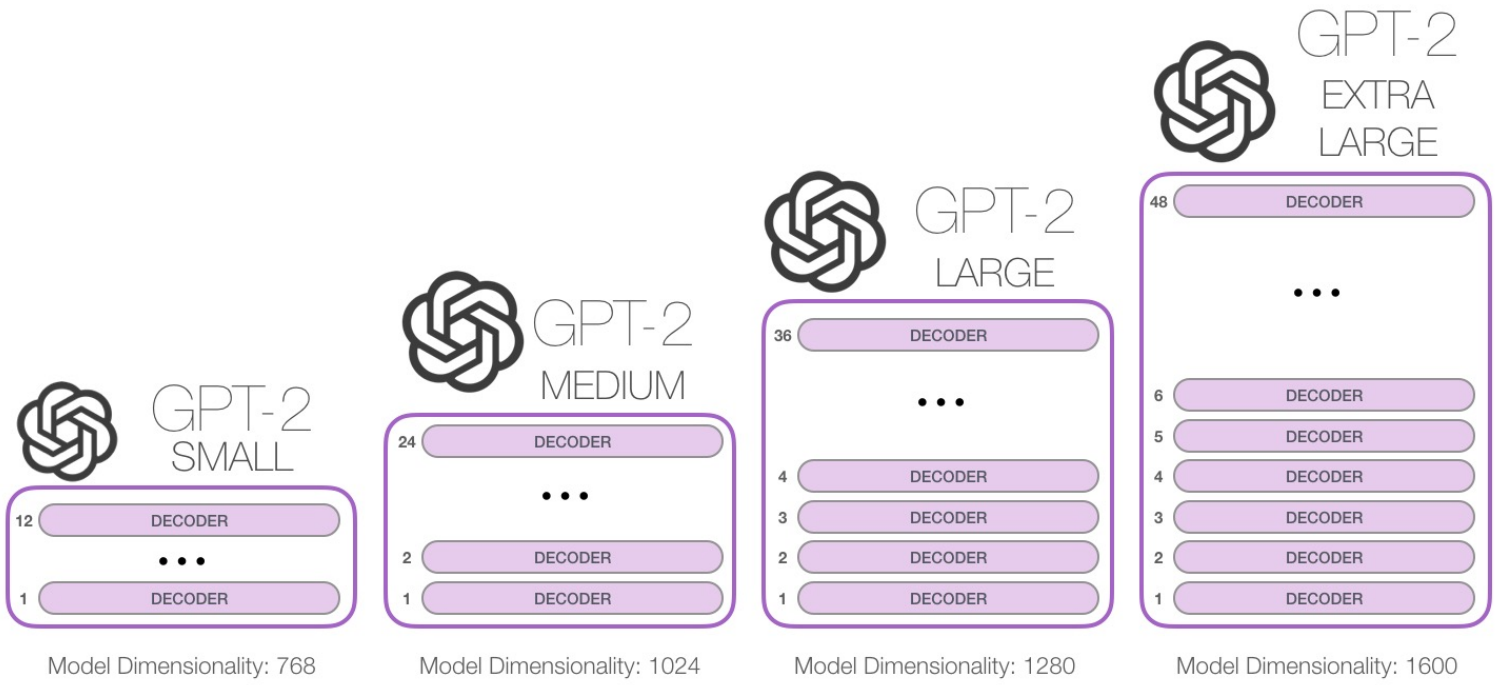
(Recap) Decoder Block



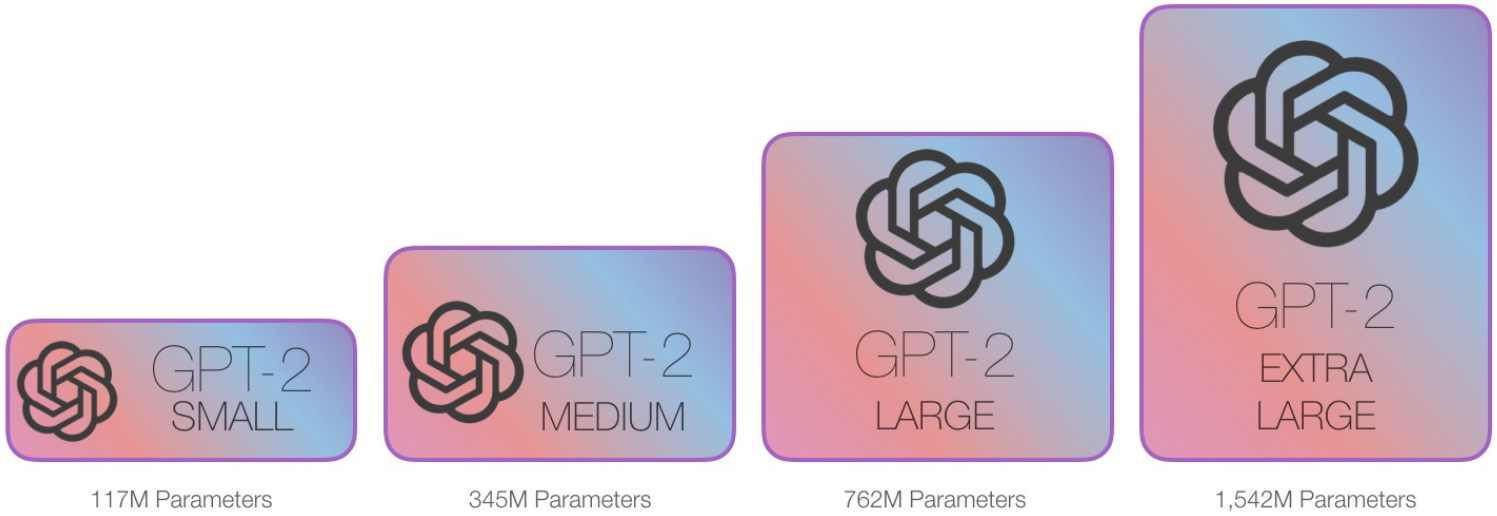
Transformer-based Language Models



How high can we stack up these blocks?



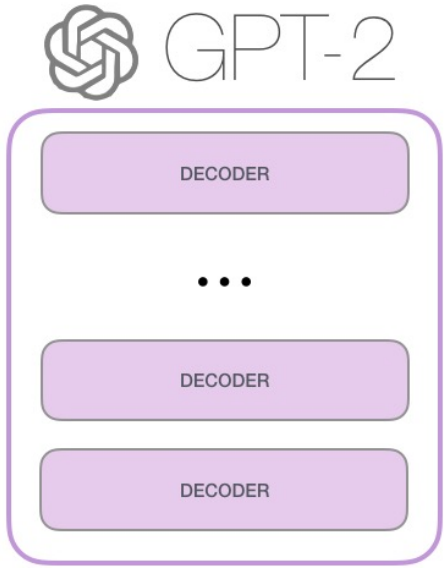
How high can we stack up these blocks?



GPT: Generative Pre-trained Transformer

Improving Language Understanding by Generative Pre-Training

Alec Radford OpenAI alec@openai.com
Karthik Narasimhan OpenAI karthikn@openai.com
Tim Salimans OpenAI tim@openai.com
Ilya Sutskever OpenAI ilyasu@openai.com

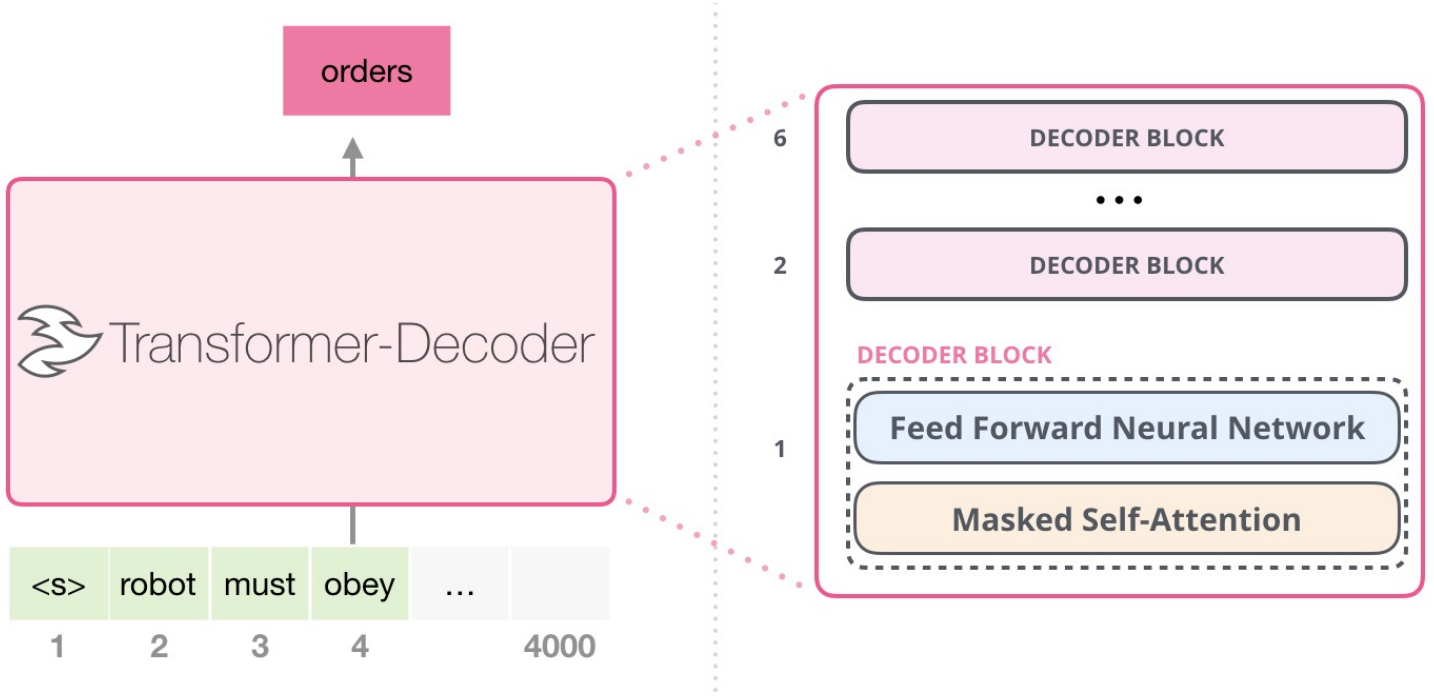


The Decoder-Only Block

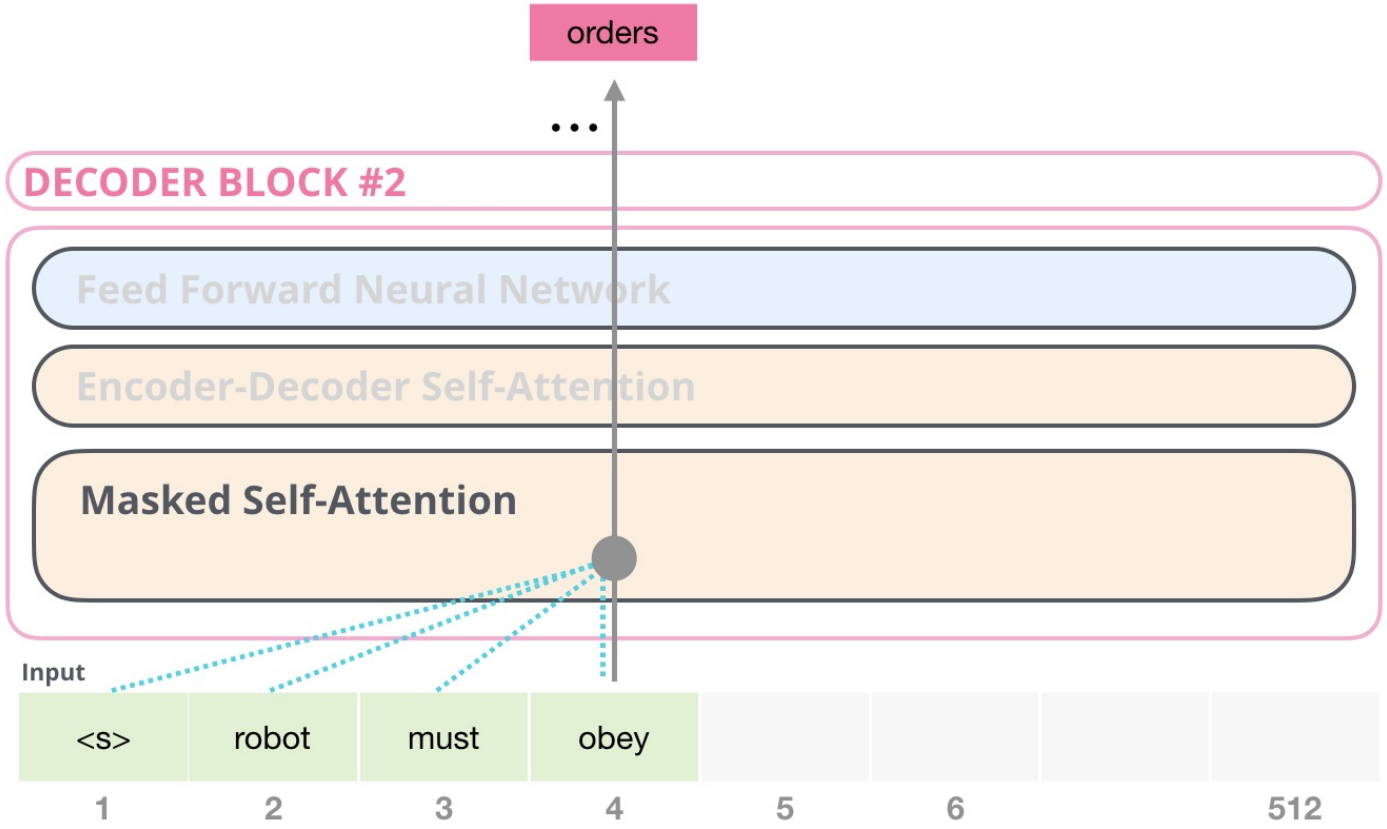
Published as a conference paper at ICLR 2018

GENERATING WIKIPEDIA BY SUMMARIZING LONG SEQUENCES

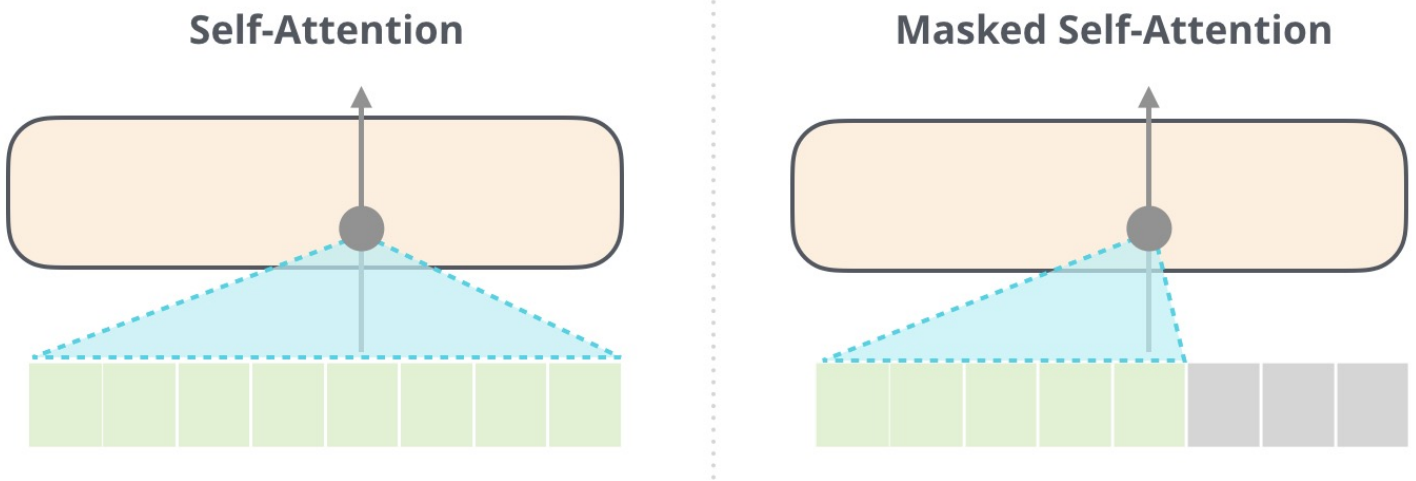
Peter J. Liu*, **Mohammad Saleh***,
Etienne Pot†, **Ben Goodrich, Ryan Sepassi, Łukasz Kaiser, Noam Shazeer**
Google Brain
Mountain View, CA
{peterjliu,msaleh,epot,bgoodrich,rsepassi,lukaszkaiser,noam}@google.com



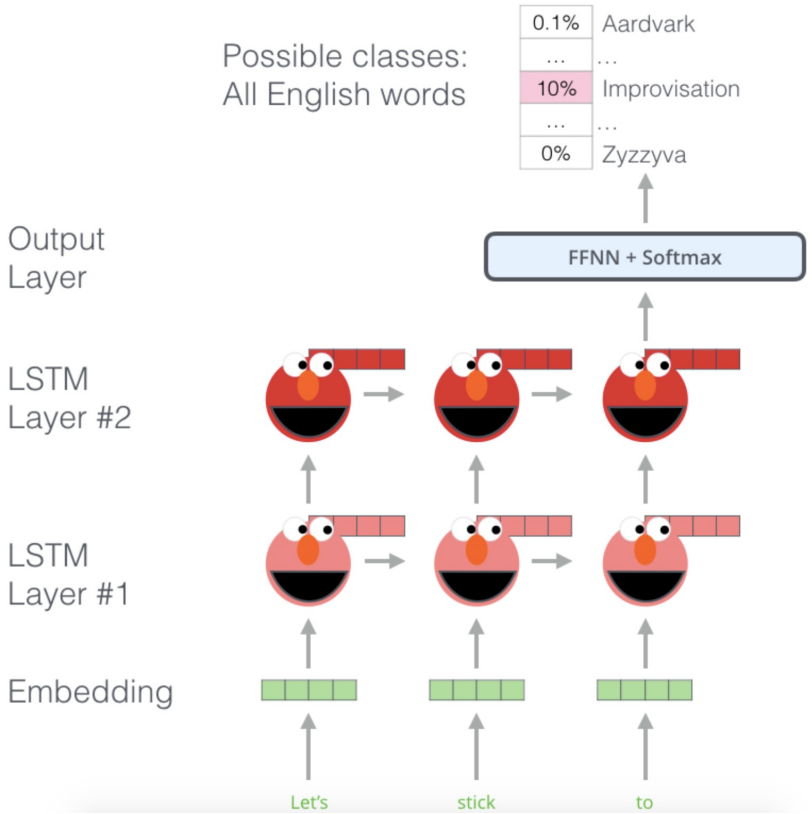
(Recap) Decoder Block

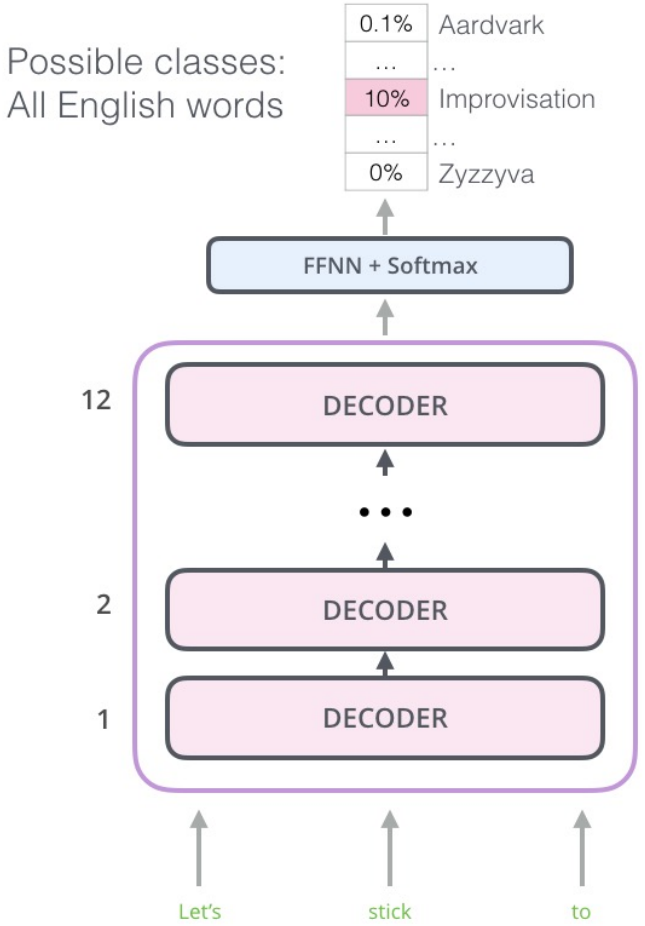


(Recap) Self-Attention

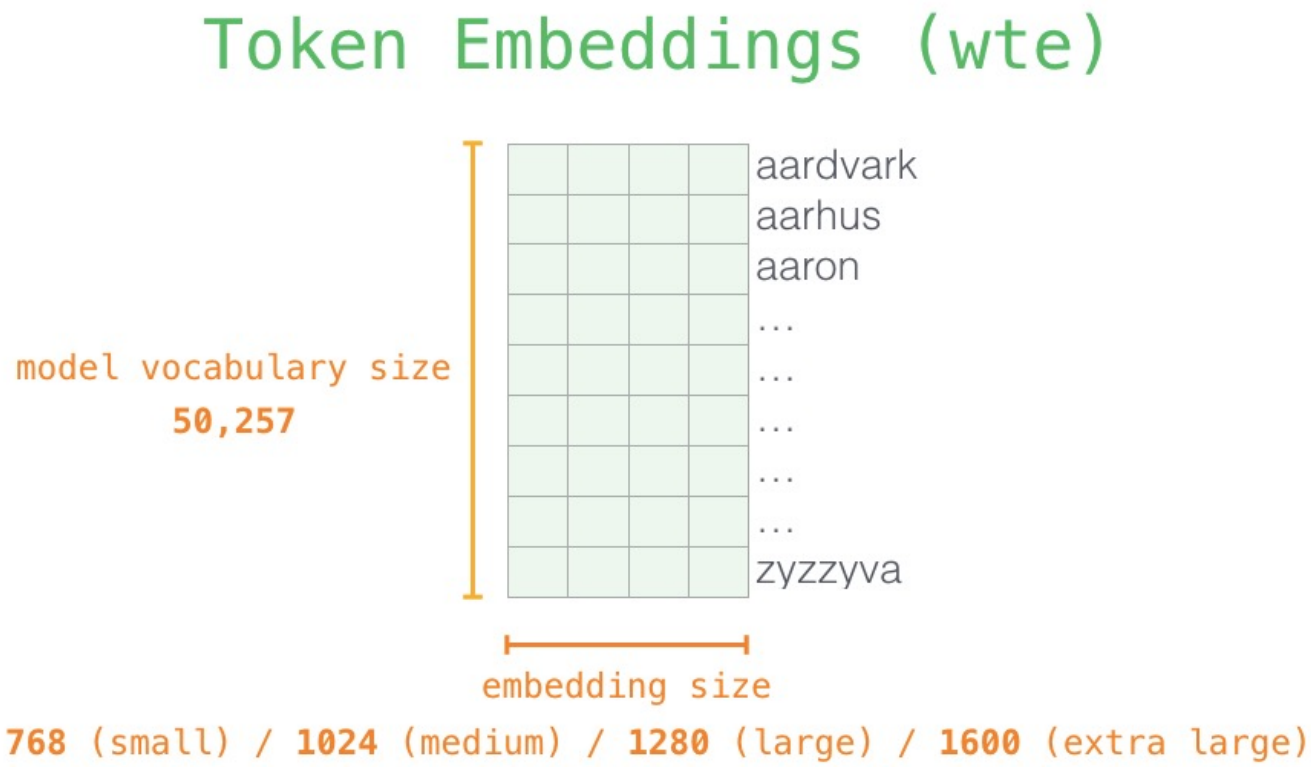


Pre-training: Language modeling (unsupervised)

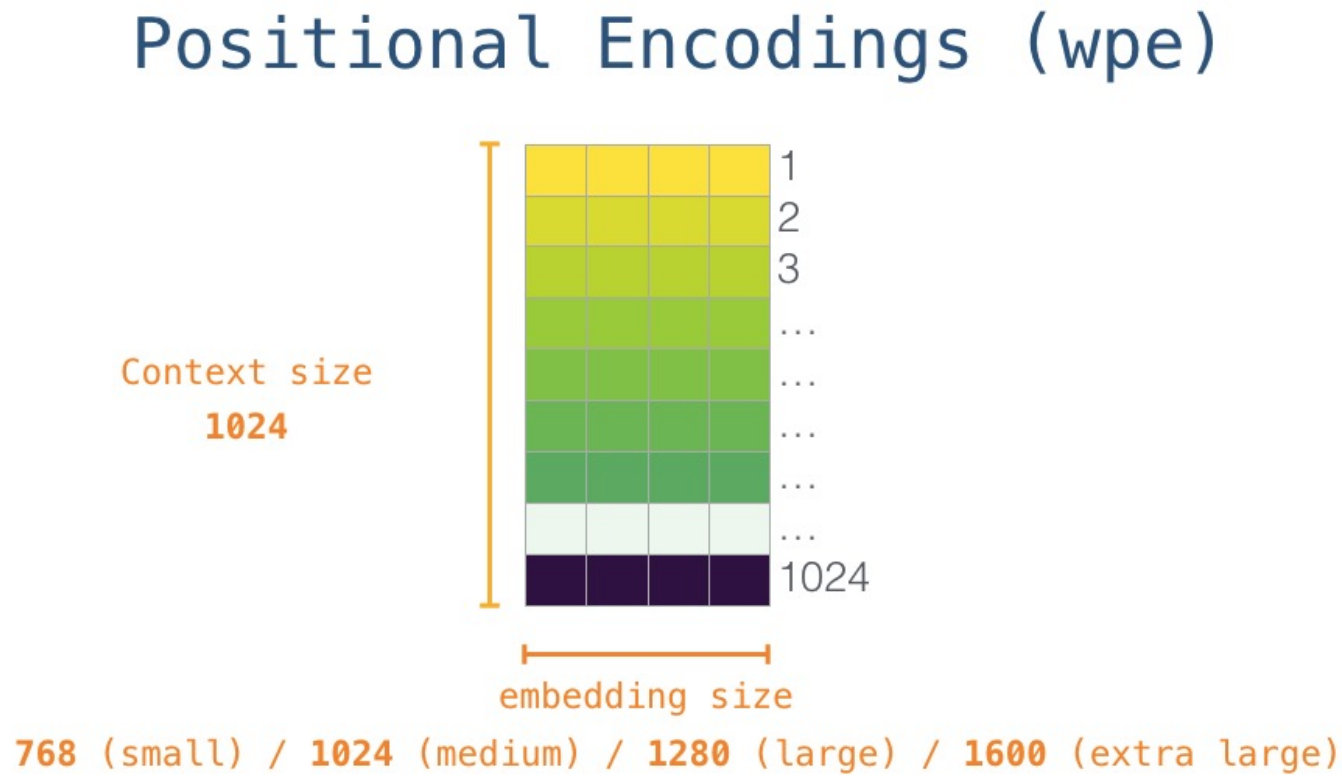




Input Encoding in GPT

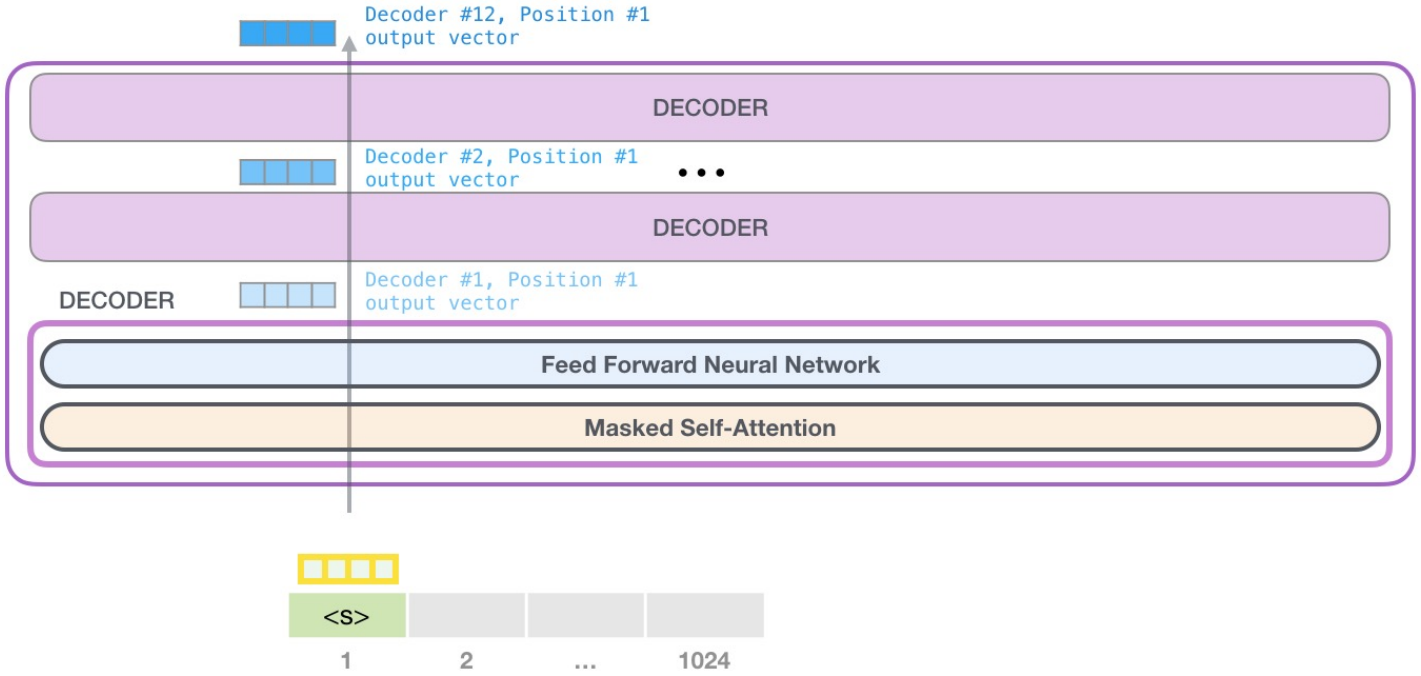


Input Encoding in GPT



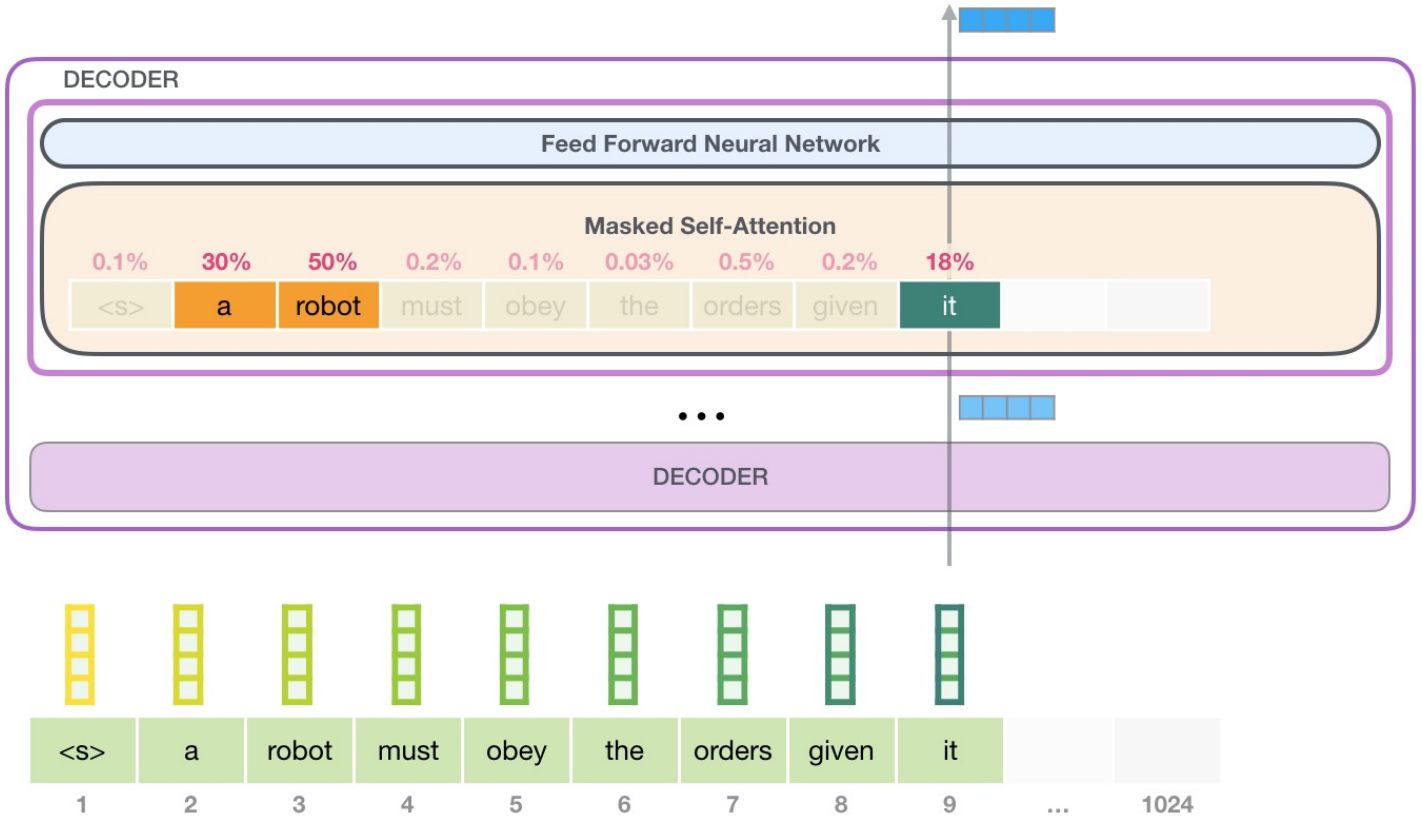
Input Encoding in GPT

Input Encoding in GPT

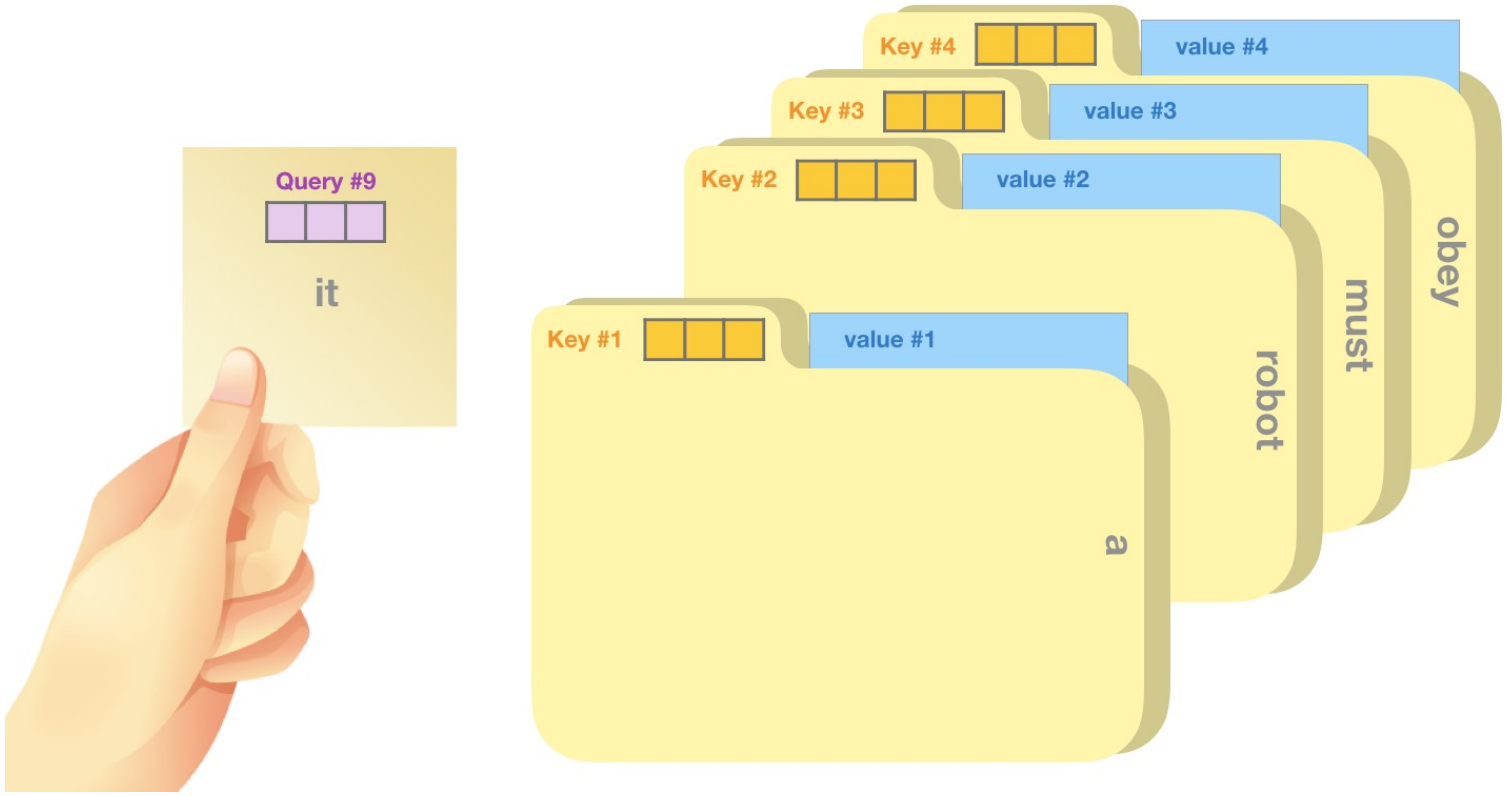


Self-Attention in GPT

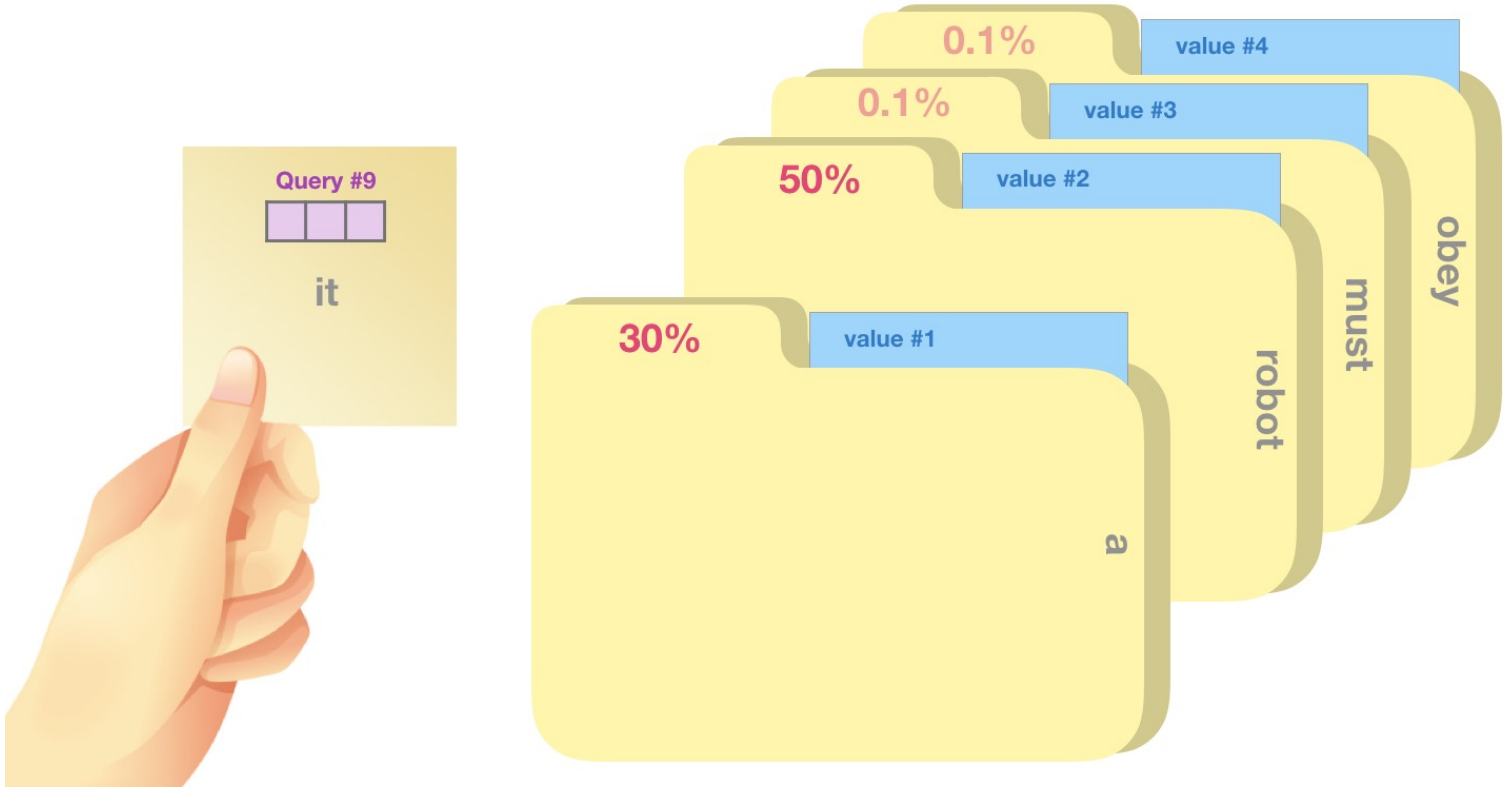
> ***Second Law of Robotics***
> *A robot must obey the orders given **it** by human beings except where **such orders** would conflict with the **First Law**.*



Self-Attention in GPT



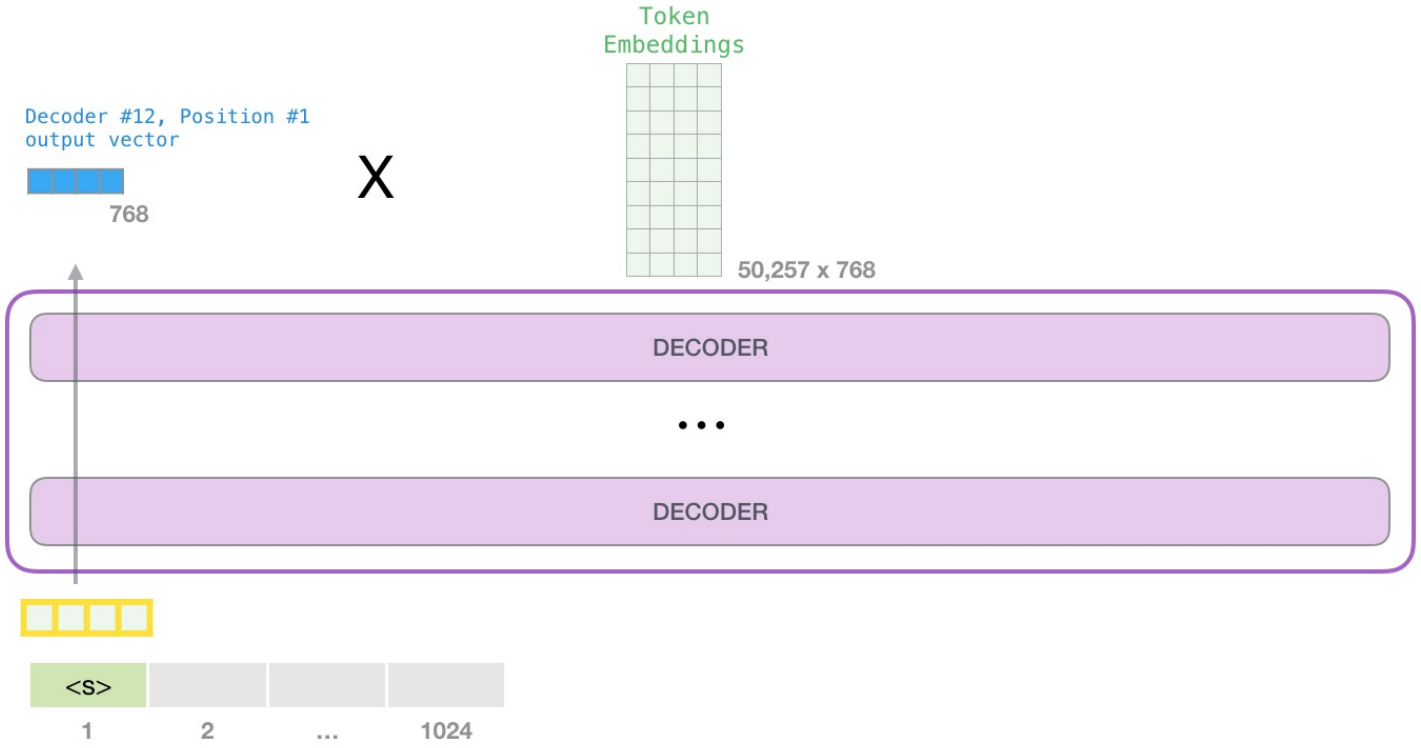
Self-Attention in GPT



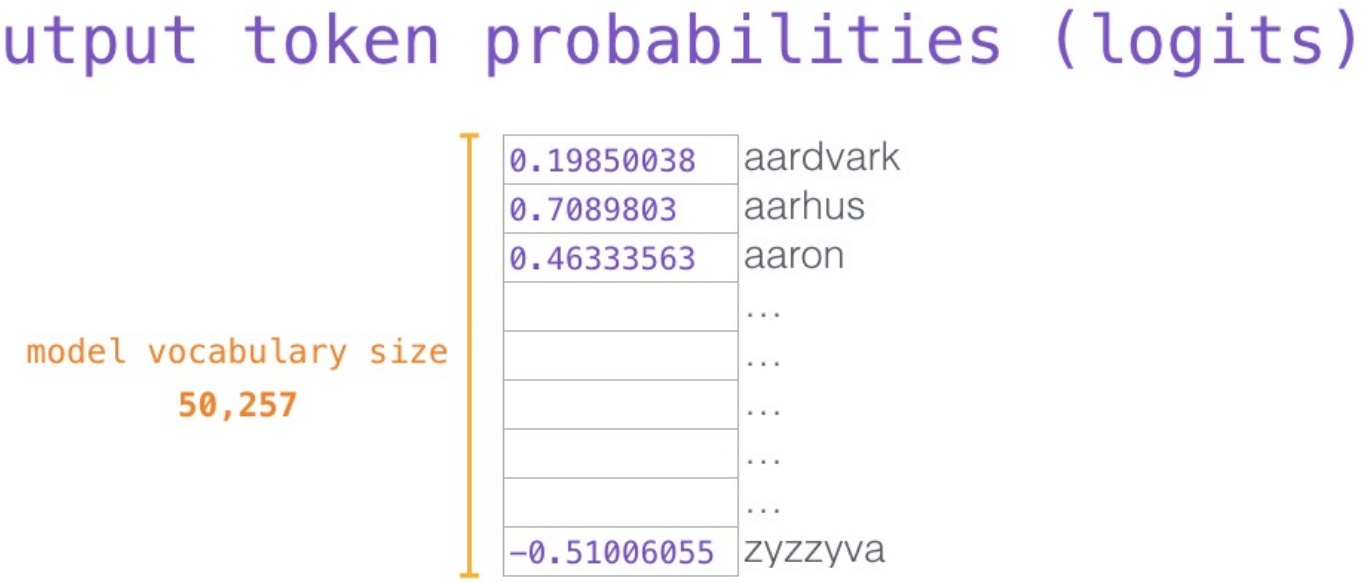
Self-Attention in GPT

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:	

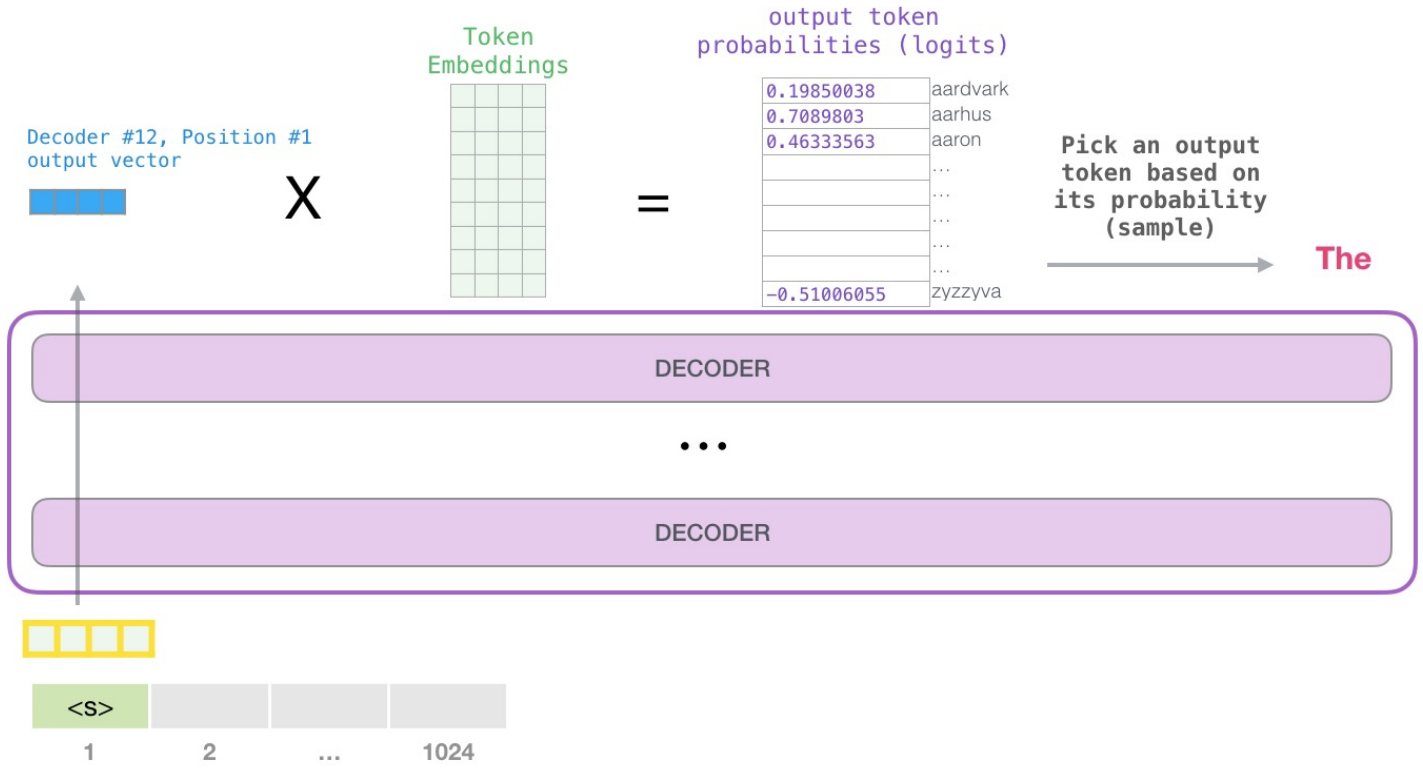
Output in GPT



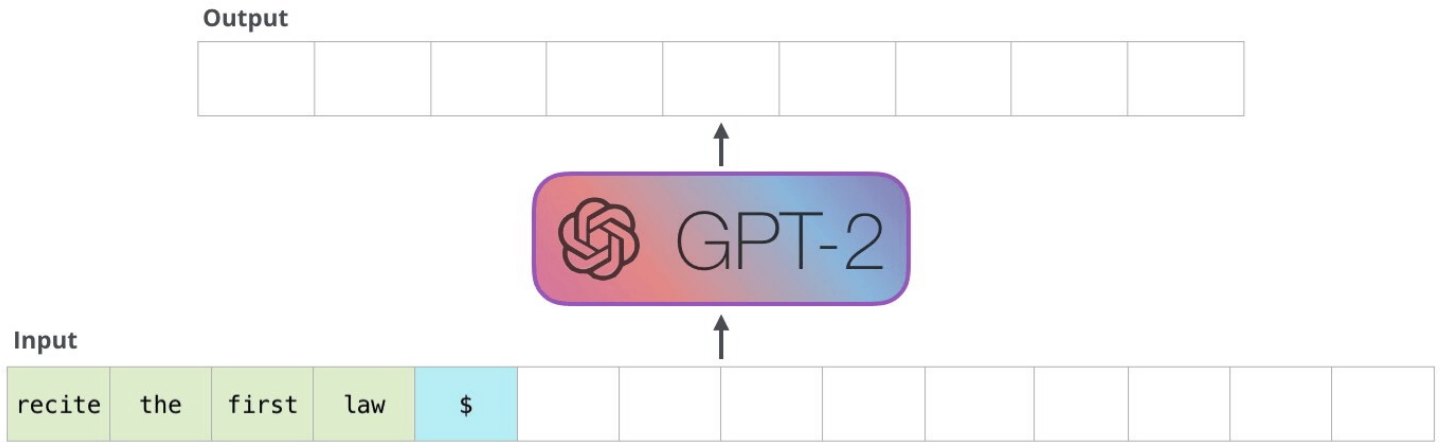
Output in GPT



Output in GPT



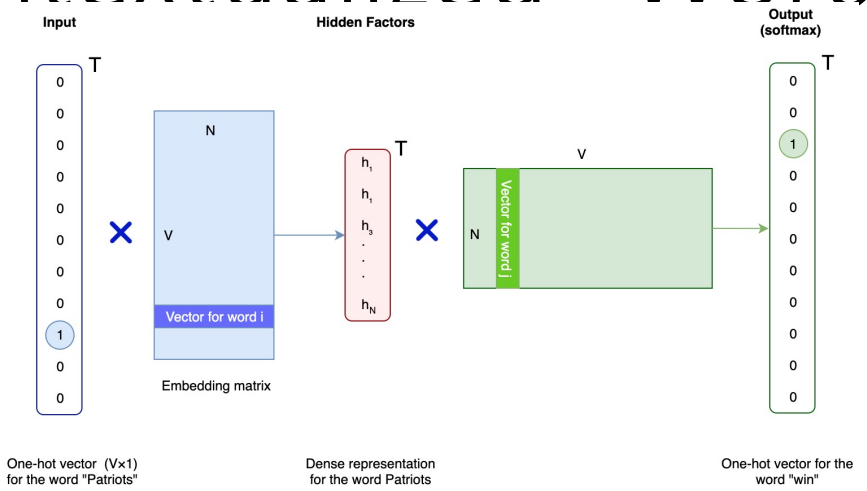
Auto-regressive Model for Language Modeling

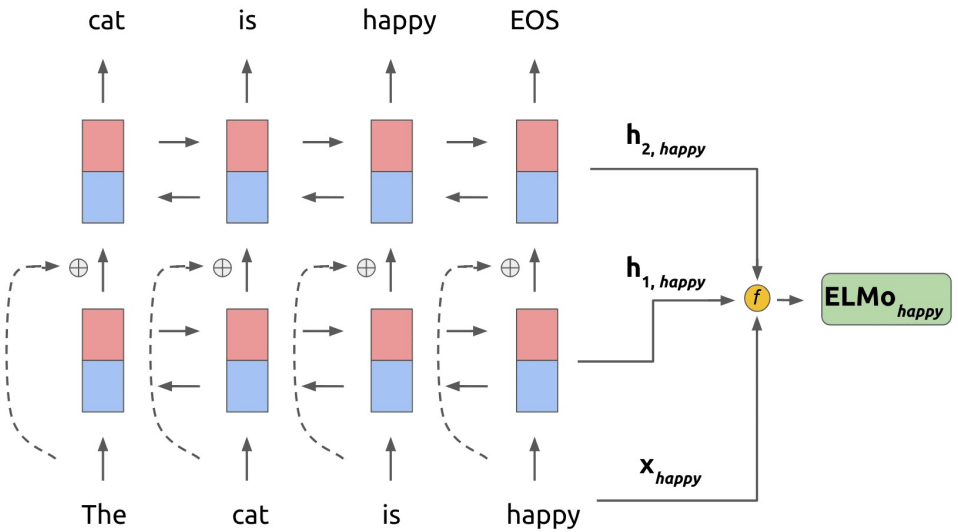


References

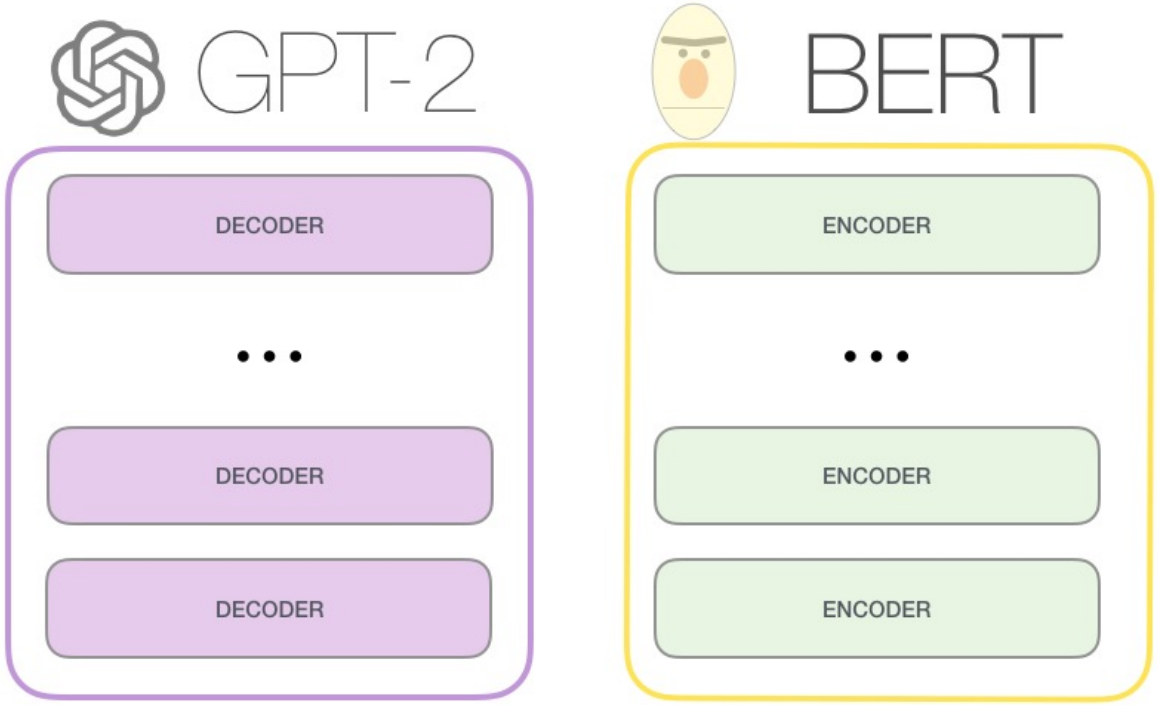
- **The Illustrated BERT, ELMo, and co.**
  - http://jalammar.github.io/illustrated-bert/
- **The Illustrated GPT-2**
  - http://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

“Contextualized” Word Embedding

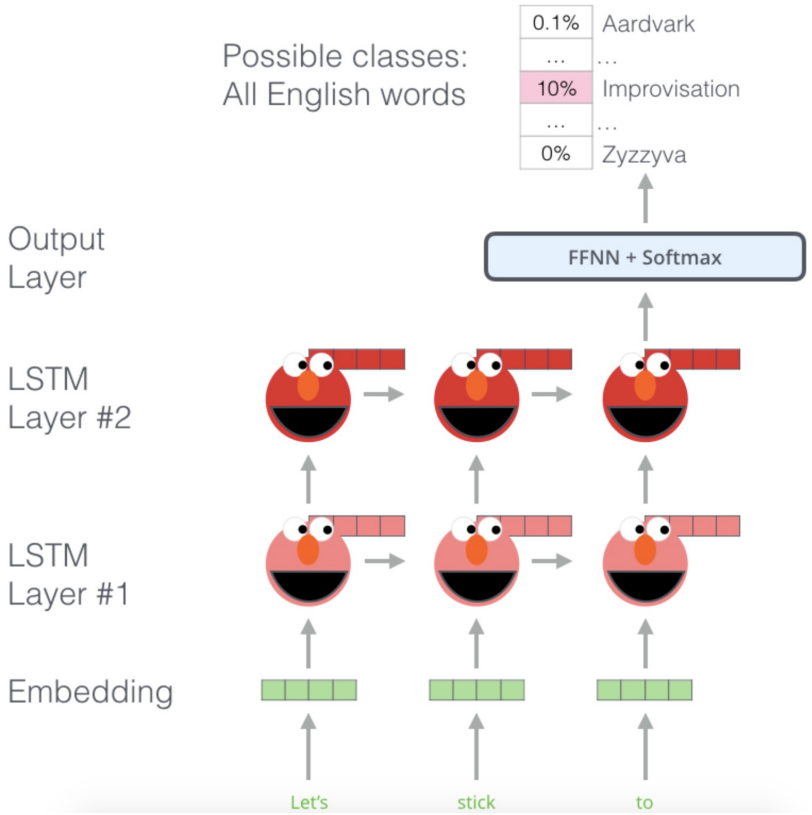


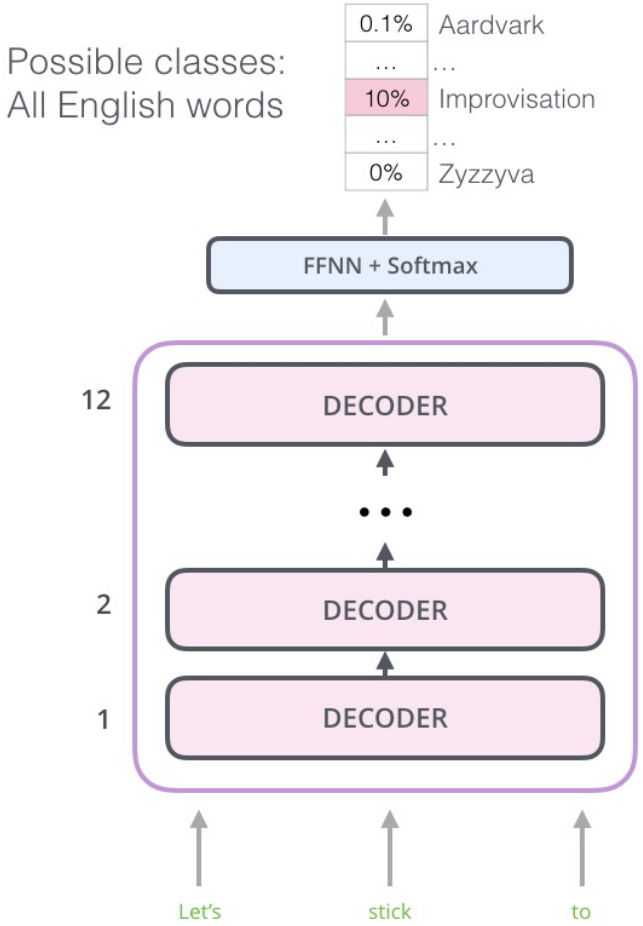


Transformer-based Language Models

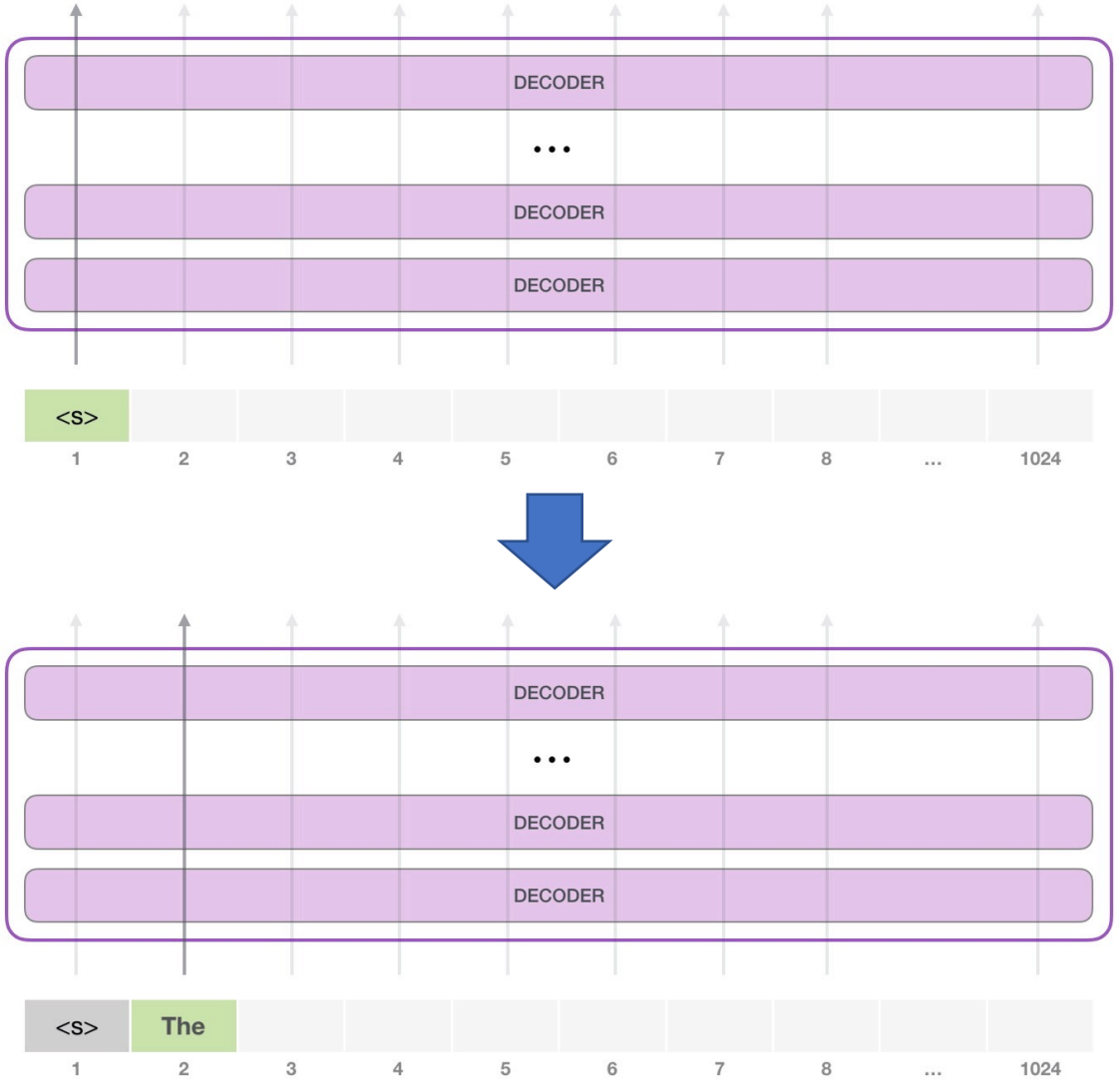


Pre-training: Language modeling (unsupervised)





Predicting next words in GPT



Predicting next words in GPT

Context:

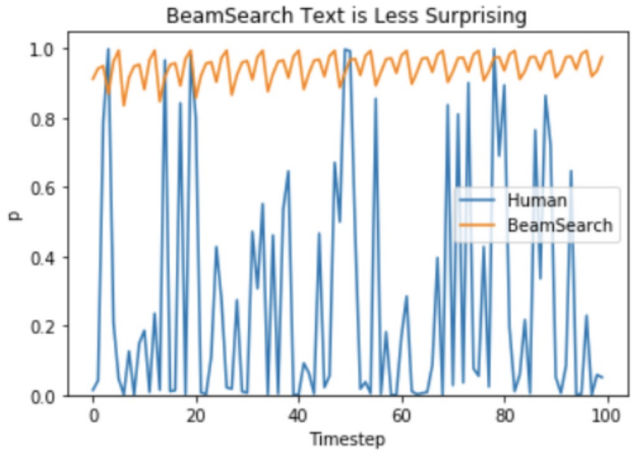
In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Continuation (BeamSearch, b=10):

"The unicorns were able to communicate with each other, they said unicorns. a statement that the unicorns. Professor of the Department of Los Angeles, the most important place the world to be recognition of the world to be a of the world to be a of the world to be a of the world to be a of the world to be a of the world to be a of the world to be a of the world to be a of the…

Figure 1: Beam search leads to degenerate text, even when generated from GPT-2-117M, in stark contrast with the admirable quality of the text decoded using *top-k* sampling (Radford et al., 2019). The *continuation* is machine generated, conditioned on the *context* provided by a human. Blue text highlights decoded words that have occurred previously in the text.

Predicting next words in GPT



Human	BeamSearch
…get your hopes up. I saw him once and I have no intention of being near him anytime soon. He sat on the edge, the wind tossing around his hair. It was going to be seriously wind-blown later. I sat down next to him and I was trying to forget the dwarfs mangled body. I shook and hugged myself. Are you cold? He asked, his voice full of concern. I just shrugged and squeezed my eyes shut. I saw Kojas glowing eyes and sword, the…	…looked at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds. He looks at the clouds…

Figure 2: The probability assigned to tokens generated by humans and beam search using GPT-2-117M. Note the increased variance that characterizes the richness of human text.

Predicting next words in GPT

🗣 **Context**: On days when he woke early, the president liked to have coffee in the oval office. There was something about watching the sky grow lighter and lighter as you sat your pajama'd behind in the most powerful chair in the free world, sipping marine strength coffee, that you just couldn't achieve anywhere else.

💰 **Greedy**: He'd been in the oval office for over a year now, and he'd never been so happy to be there. He'd been in the oval office for almost a year, and he'd never been so happy to be there.

🚦 **BeamSearch** (k=4): The President was a man of his word. He was a man of his word. He was a man of his word. He was a man of his word. He was a man of his word. He was a man of his word.

🔧 **Sampling** (t=1.0): You couldn't be sure if that's what you were really doing, and If you decided to take the day off. The president wanted you to take the day off, but he wanted to maintain a curfew and use his influence wisely.

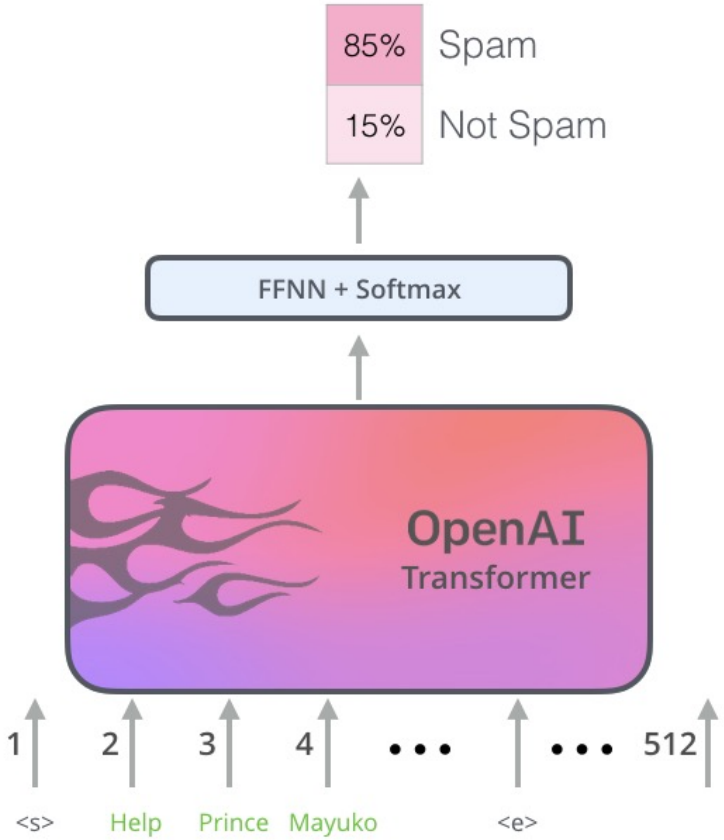
🆚 **Top-k Sampling** (k=8): "I'm sorry, sir." "No, it's okay, son," the president said. "I understand." "You're going to have to make a special trip down there to get that kid. He has no idea where he's going."

⚛ **Nucleus Sampling** (p=0.9): But that wasn't what drew the president's attention. He'd been seated for maybe a minute when he noticed the other man. What was the guy doing here?

🗣 **Gold**: He was therefore disagreeably surprised to find a man in an understated grey suit sitting in that selfsame chair sipping tea. The president turned around and went looking for his chief of staff.

Figure 10: Example generations from all discussed decoding strategies, hyperparameters were chosen by experts. All generations for all hyperparameters will be made publicly available.

Fine-tuning: Transfer learning to downstream tasks

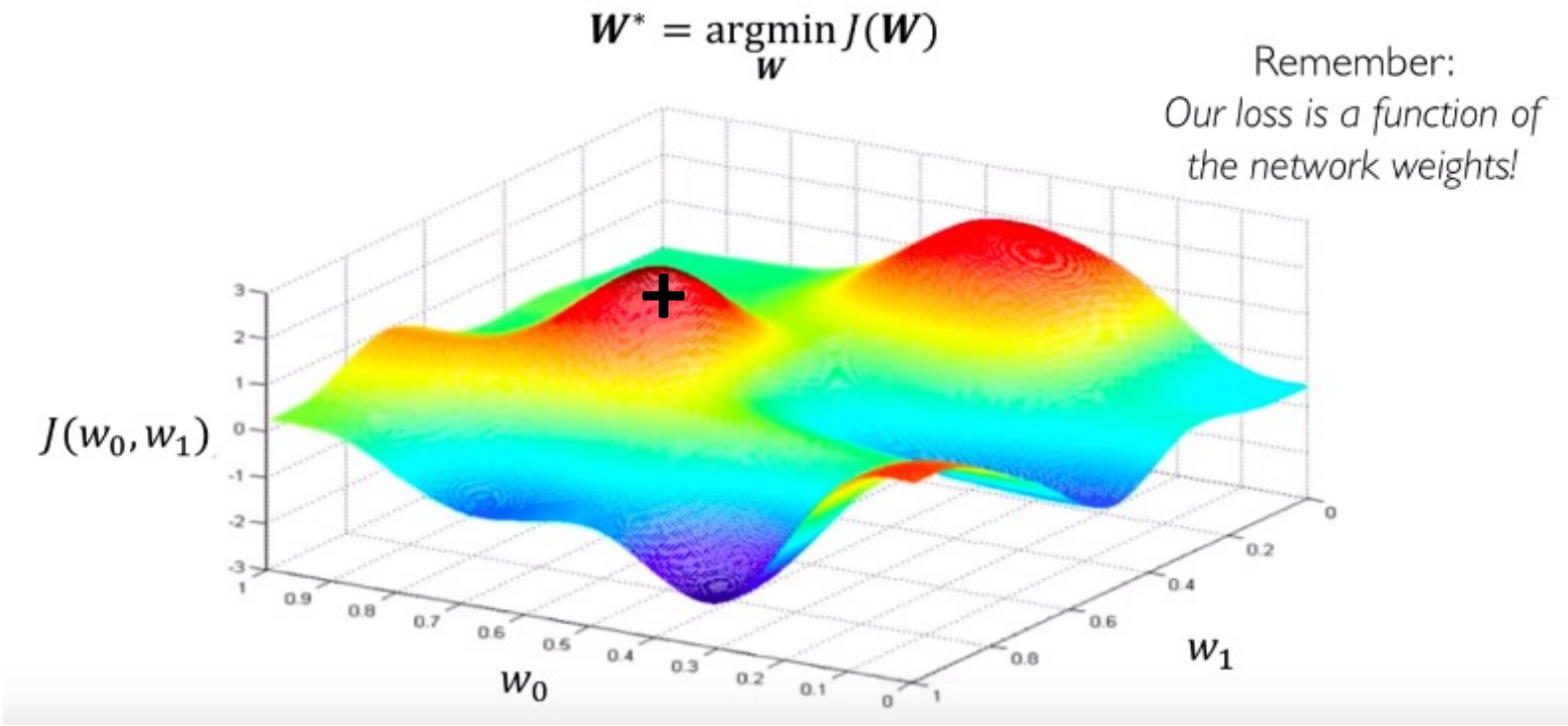


Why pre-training? Why fine-tuning?

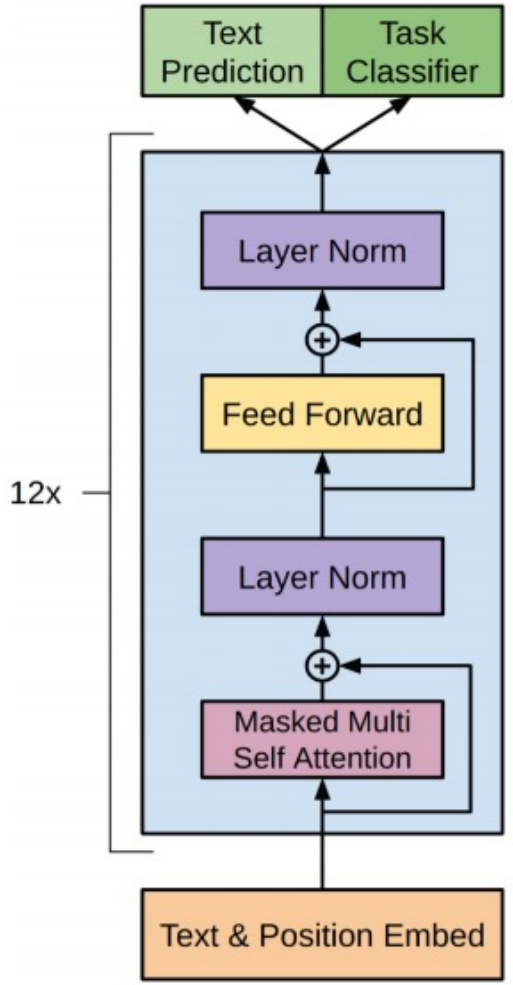
Randomly pick an initial (w_0, w_1)

$$W^* = \underset{W}{\text{argmin}}\, J(W)$$

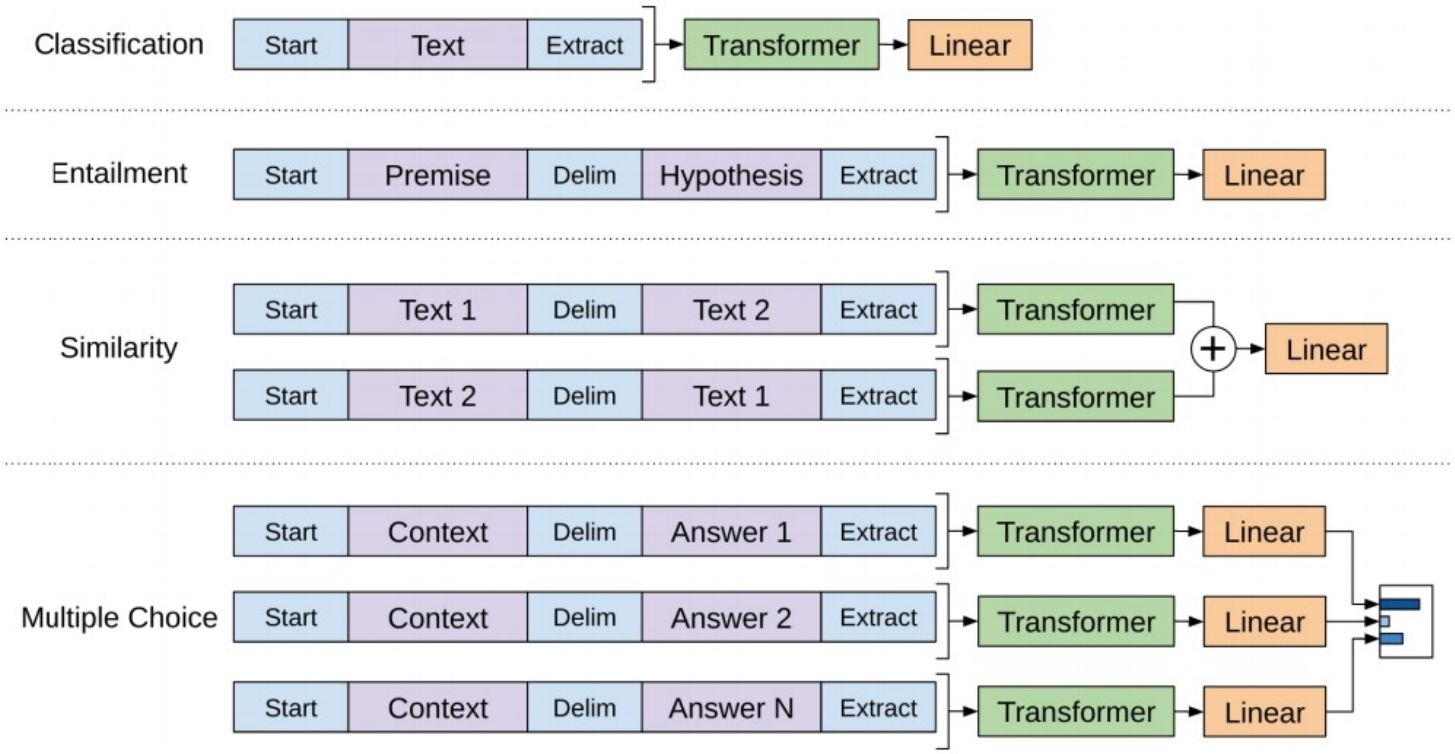
Remember:
Our loss is a function of the network weights!



Fine-tuning: Transfer learning to downstream tasks



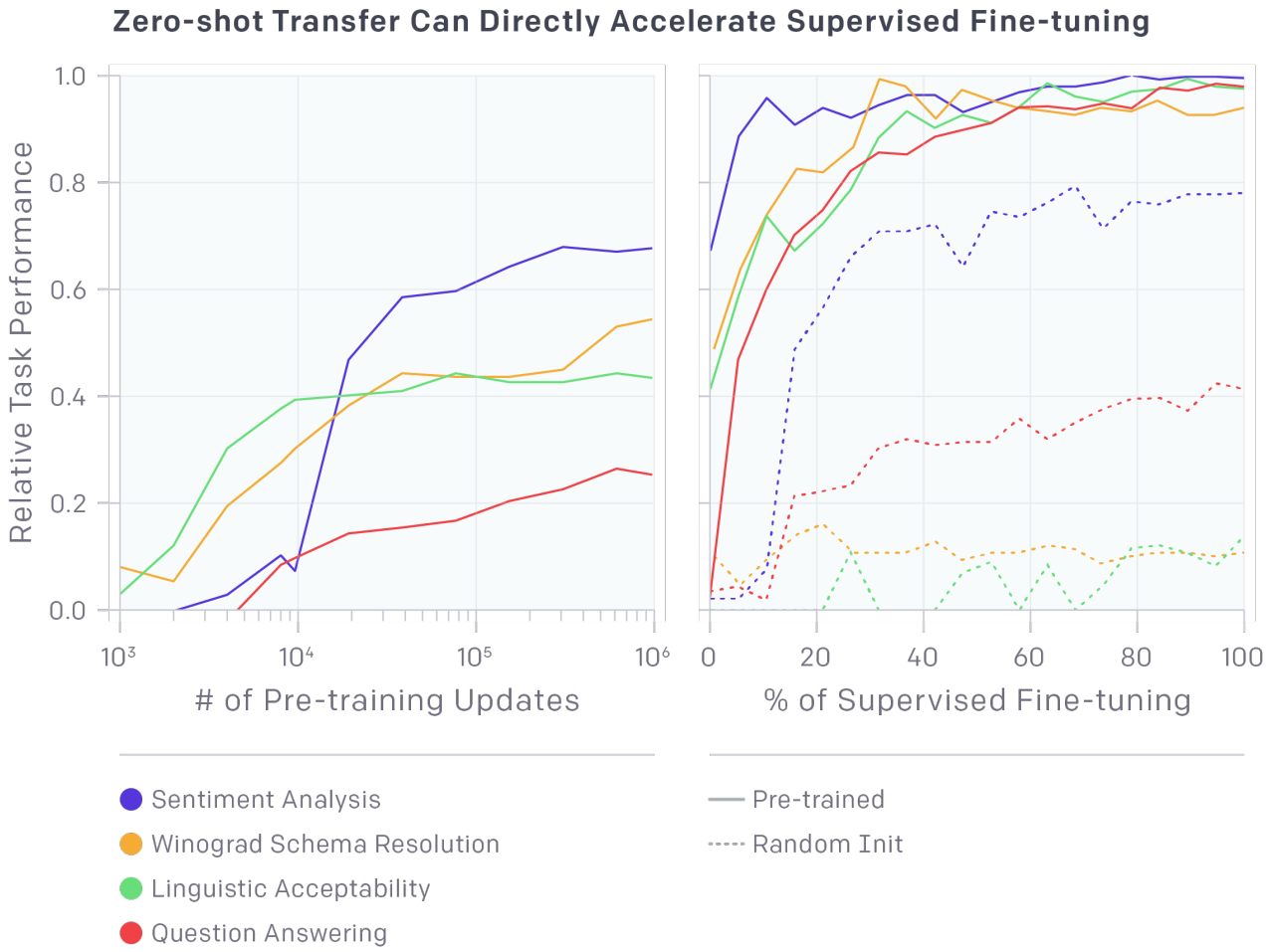
Fine-tuning: Transfer learning to downstream tasks



Experiment in GPT

DATASET	TASK	SOTA	OURS
SNLI	Textual Entailment	89.3	**89.9**
MNLI Matched	Textual Entailment	80.6	**82.1**
MNLI Mismatched	Textual Entailment	80.1	**81.4**
SciTail	Textual Entailment	83.3	**88.3**
QNLI	Textual Entailment	82.3	**88.1**
RTE	Textual Entailment	**61.7**	56.0
STS-B	Semantic Similarity	81.0	**82.0**
QQP	Semantic Similarity	66.1	**70.3**
MRPC	Semantic Similarity	**86.0**	82.3
RACE	Reading Comprehension	53.3	**59.0**
ROCStories	Commonsense Reasoning	77.6	**86.5**
COPA	Commonsense Reasoning	71.2	**78.6**
SST-2	Sentiment Analysis	**93.2**	91.3
CoLA	Linguistic Acceptability	35.0	**45.4**
GLUE	Multi Task Benchmark	68.9	**72.8**

Why Unsupervised Learning?



BERT: Bidirectional Encoder Representations from Transformers

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin **Ming-Wei Chang** **Kenton Lee** **Kristina Toutanova**
Google AI Language
{jacobdevlin,mingweichang,kentonl,kristout}@google.com

https://arxiv.org/pdf/1810.04805.pdf

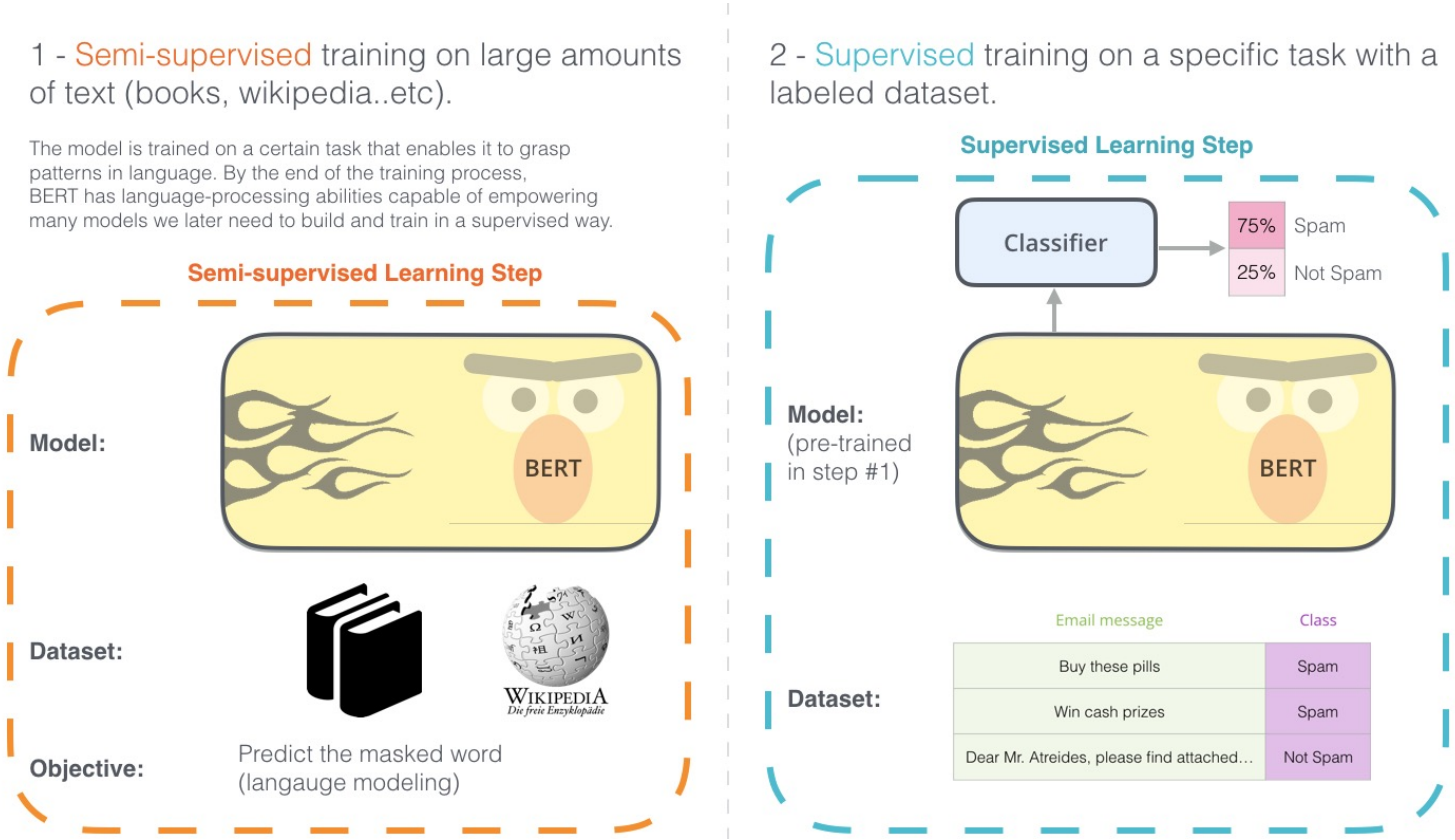
google-research / bert Watch 923 Star 22.6k

https://github.com/google-research/bert

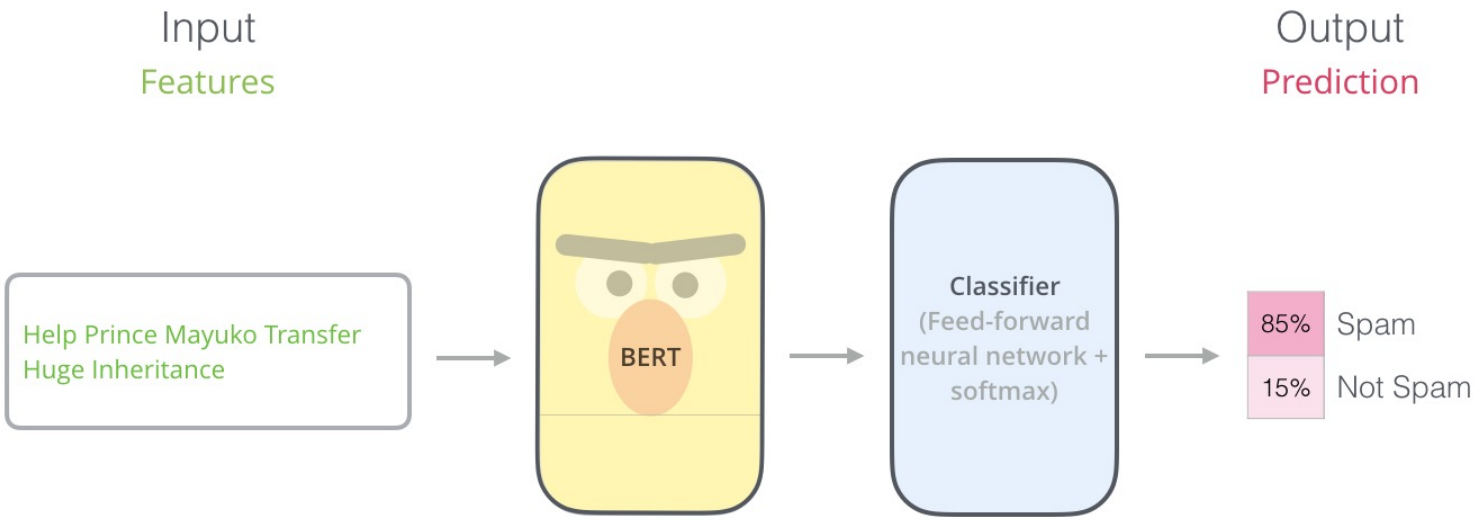
huggingface / transformers Watch 536 Star 25.5k

https://github.com/huggingface/transformers

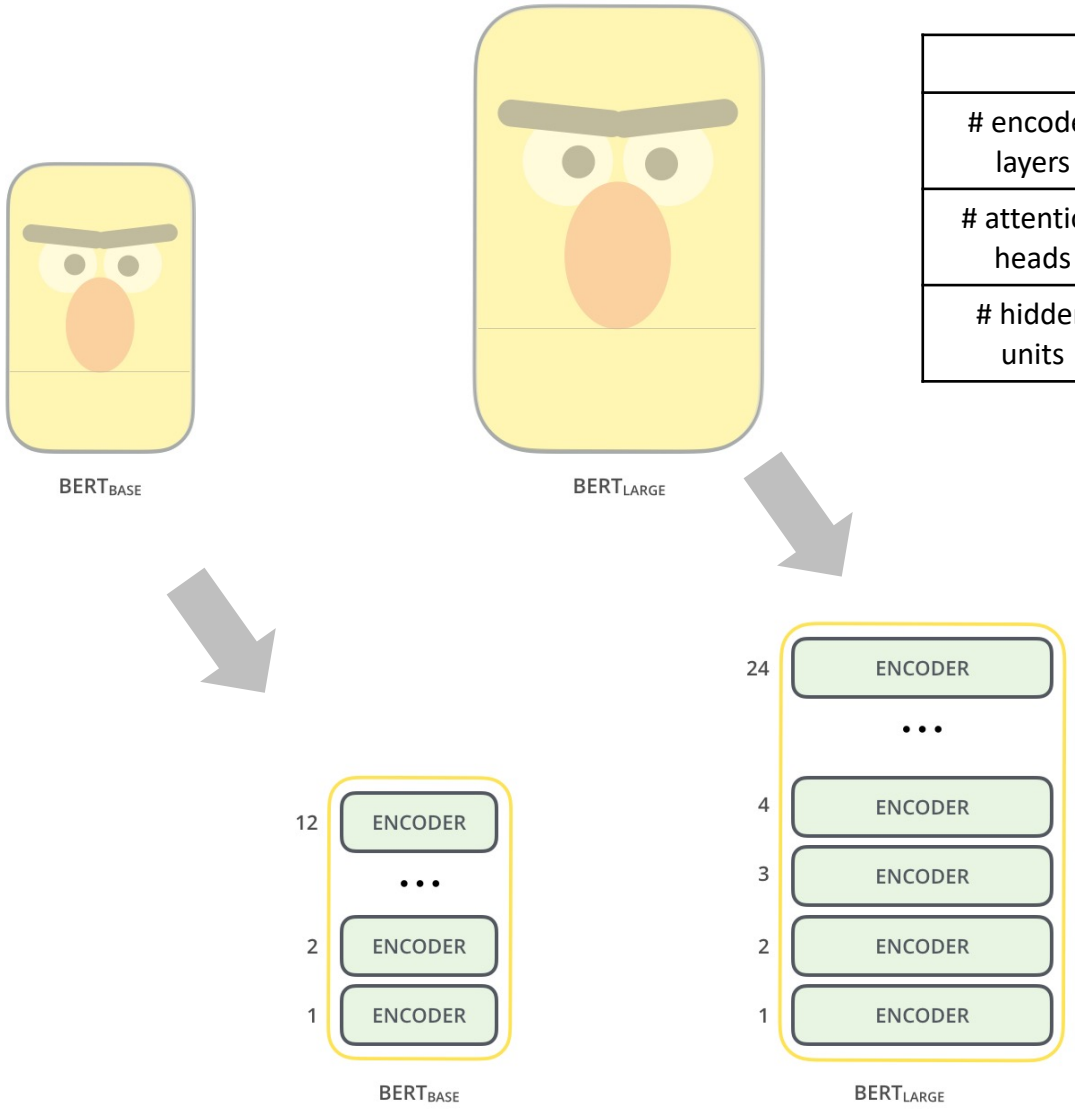
BERT: Bidirectional Encoder Representations from Transformers



BERT: Bidirectional Encoder Representations from Transformers

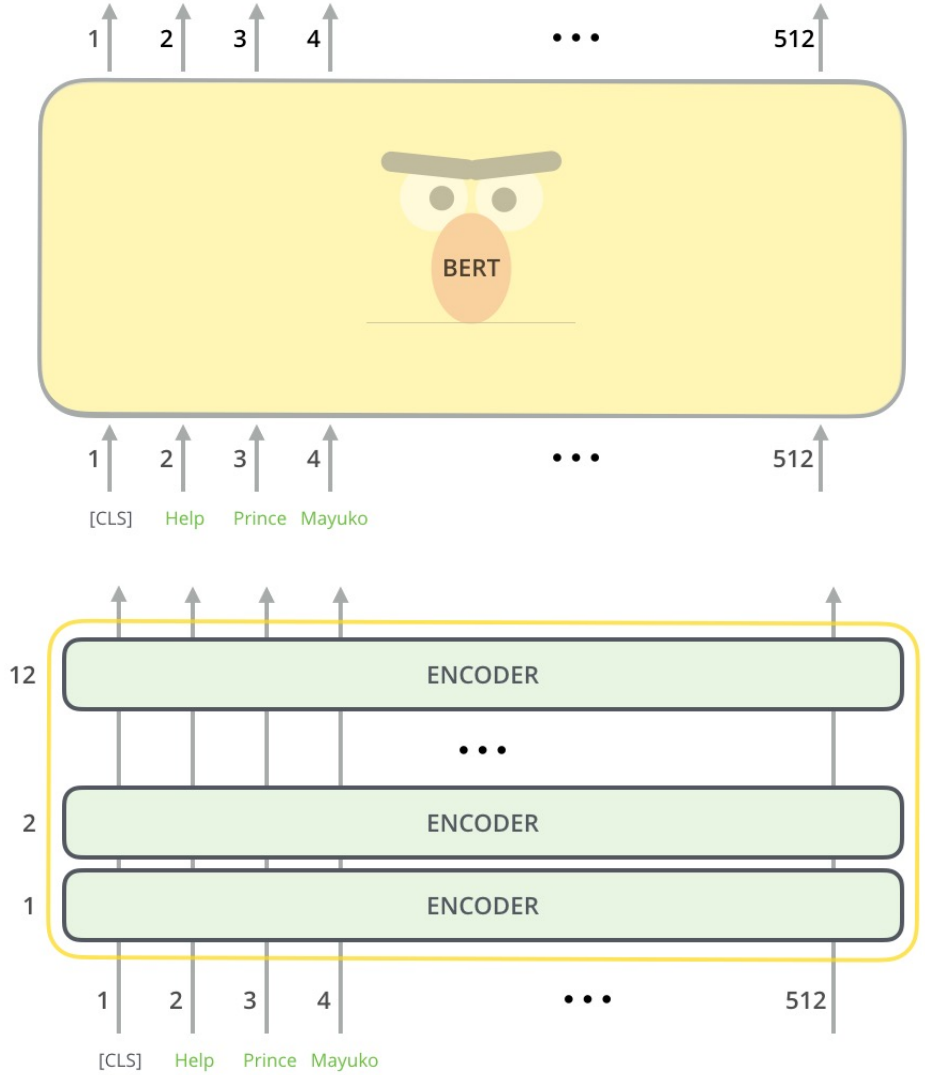


Architecture of BERT



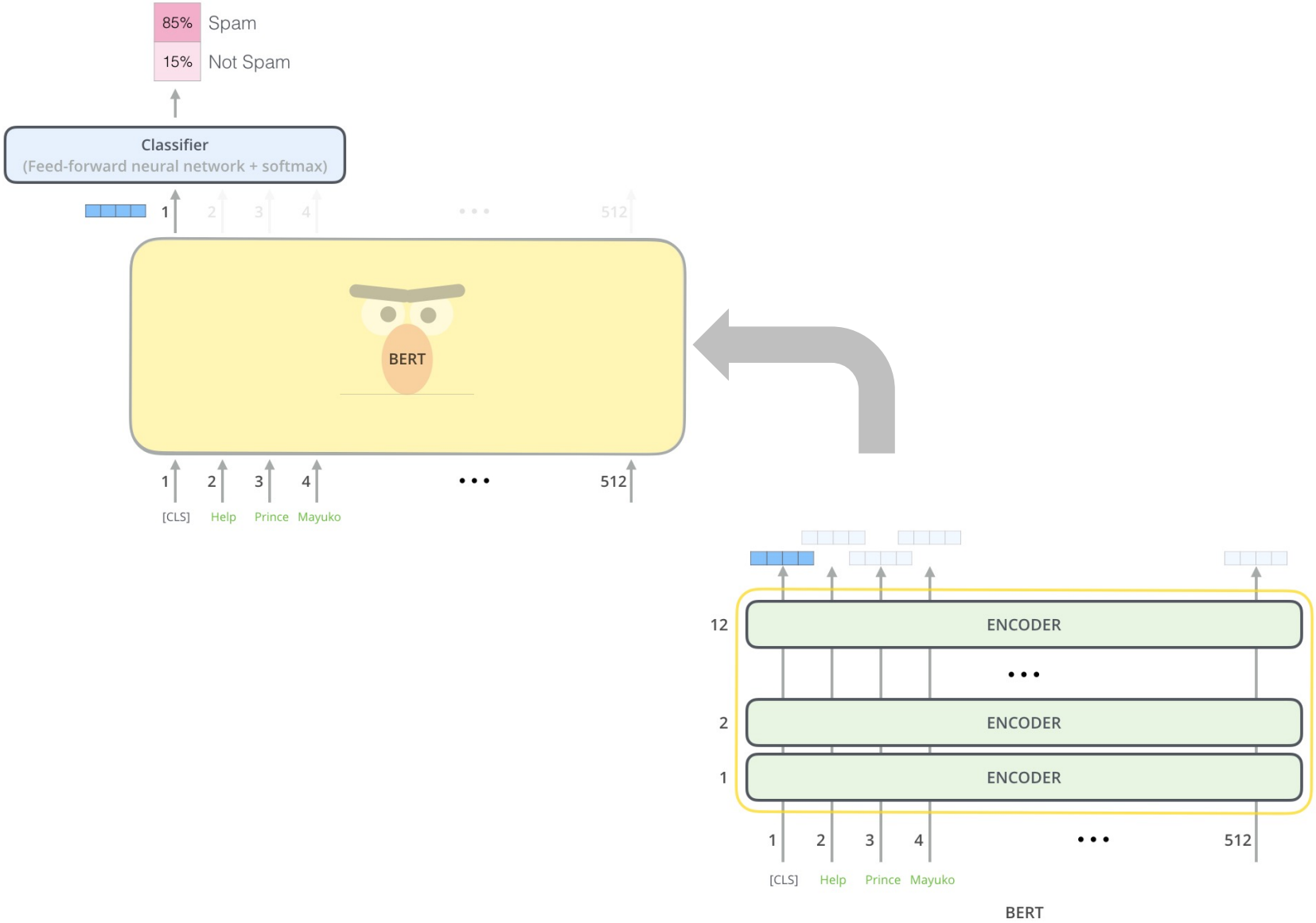
	Transformer	BERT Base	BERT Large
# encoder layers	6	12	24
# attention heads	8	12	16
# hidden units	512	768	1024

Architecture of BERT

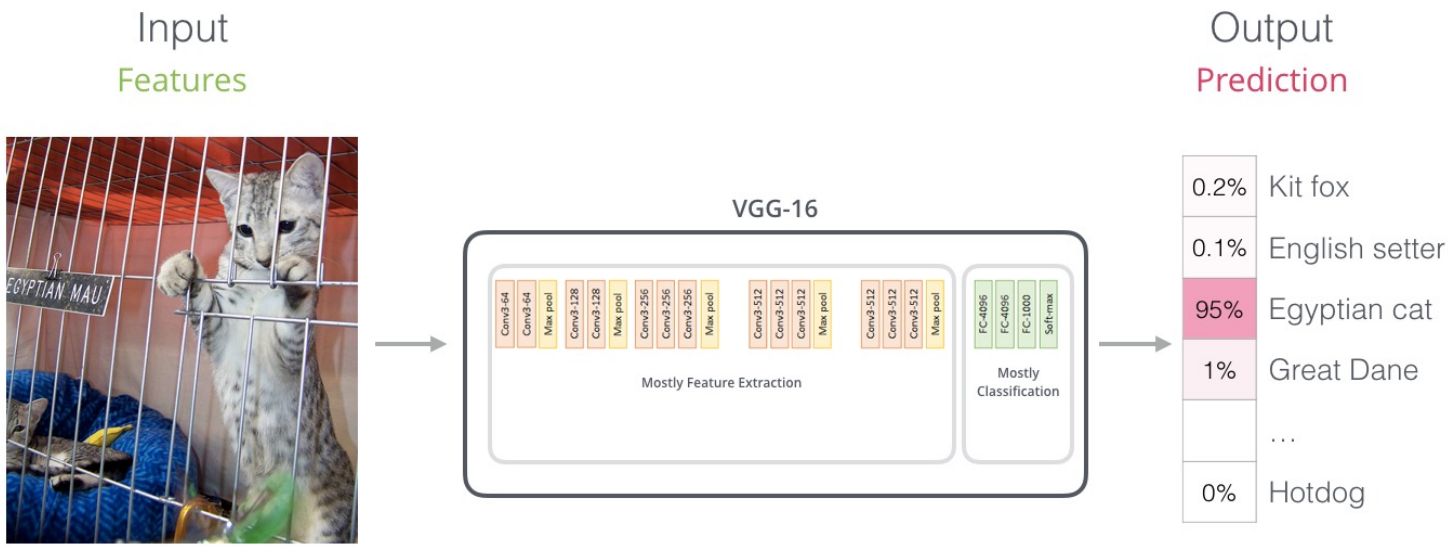


BERT

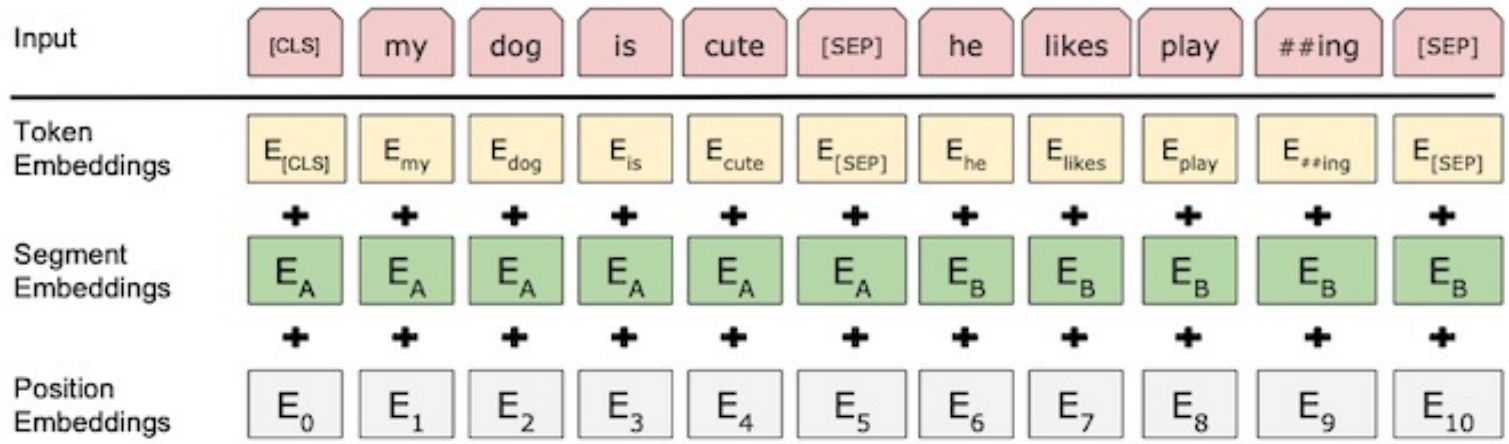
Architecture of BERT



Parallels with Convolutional Nets



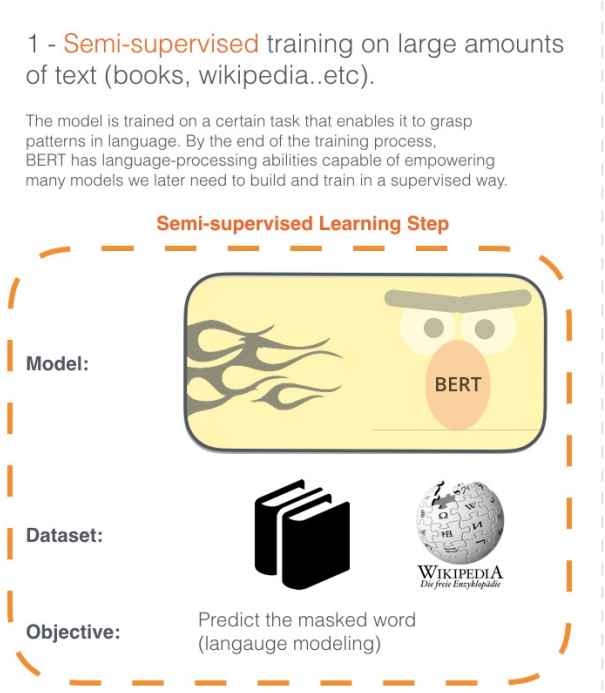
Difference with Vanilla Transformer Encoder



Difference with Vanilla Transformer Encoder

```python
""" encoder """
class Encoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        self.enc_emb = nn.Embedding(self.config.n_enc_vocab, self.config.d_hidn)
        self.pos_emb = nn.Embedding(self.config.n_enc_seq + 1, self.config.d_hidn)
        self.seg_emb = nn.Embedding(self.config.n_seg_type, self.config.d_hidn)

        self.layers = nn.ModuleList([EncoderLayer(self.config) for _ in range(self.confi

    def forward(self, inputs, segments):
        positions = torch.arange(inputs.size(1), device=inputs.device, dtype=inputs.dtype
        pos_mask = inputs.eq(self.config.i_pad)
        positions.masked_fill_(pos_mask, 0)

        # (bs, n_enc_seq, d_hidn)
        outputs = self.enc_emb(inputs) + self.pos_emb(positions)  + self.seg_emb(segments

        # (bs, n_enc_seq, n_enc_seq)
        attn_mask = get_attn_pad_mask(inputs, inputs, self.config.i_pad)

        attn_probs = []
        for layer in self.layers:
            # (bs, n_enc_seq, d_hidn), (bs, n_head, n_enc_seq, n_enc_seq)
            outputs, attn_prob = layer(outputs, attn_mask)
            attn_probs.append(attn_prob)
        # (bs, n_enc_seq, d_hidn), [(bs, n_head, n_enc_seq, n_enc_seq)]
        return outputs, attn_probs
```

Pre-training



$$\text{Input} = \texttt{[CLS] the man went to [MASK] store [SEP]}$$

$$\texttt{he bought a gallon [MASK] milk [SEP]}$$

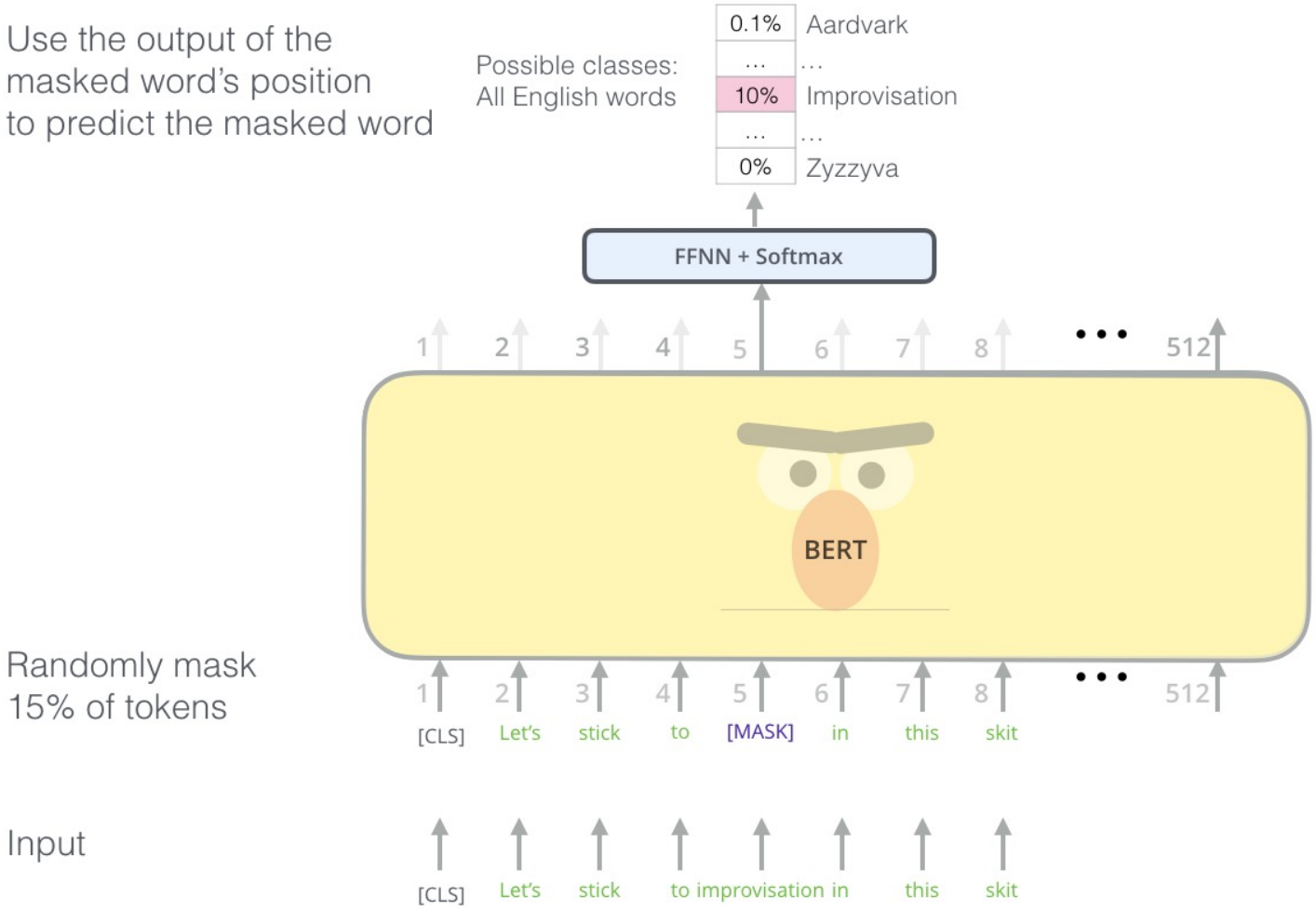
$$\text{Label} = \texttt{IsNext}$$

$$\text{Input} = \texttt{[CLS] the man [MASK] to the store [SEP]}$$

$$\texttt{penguin [MASK] are flight ##less birds [SEP]}$$

$$\text{Label} = \texttt{NotNext}$$

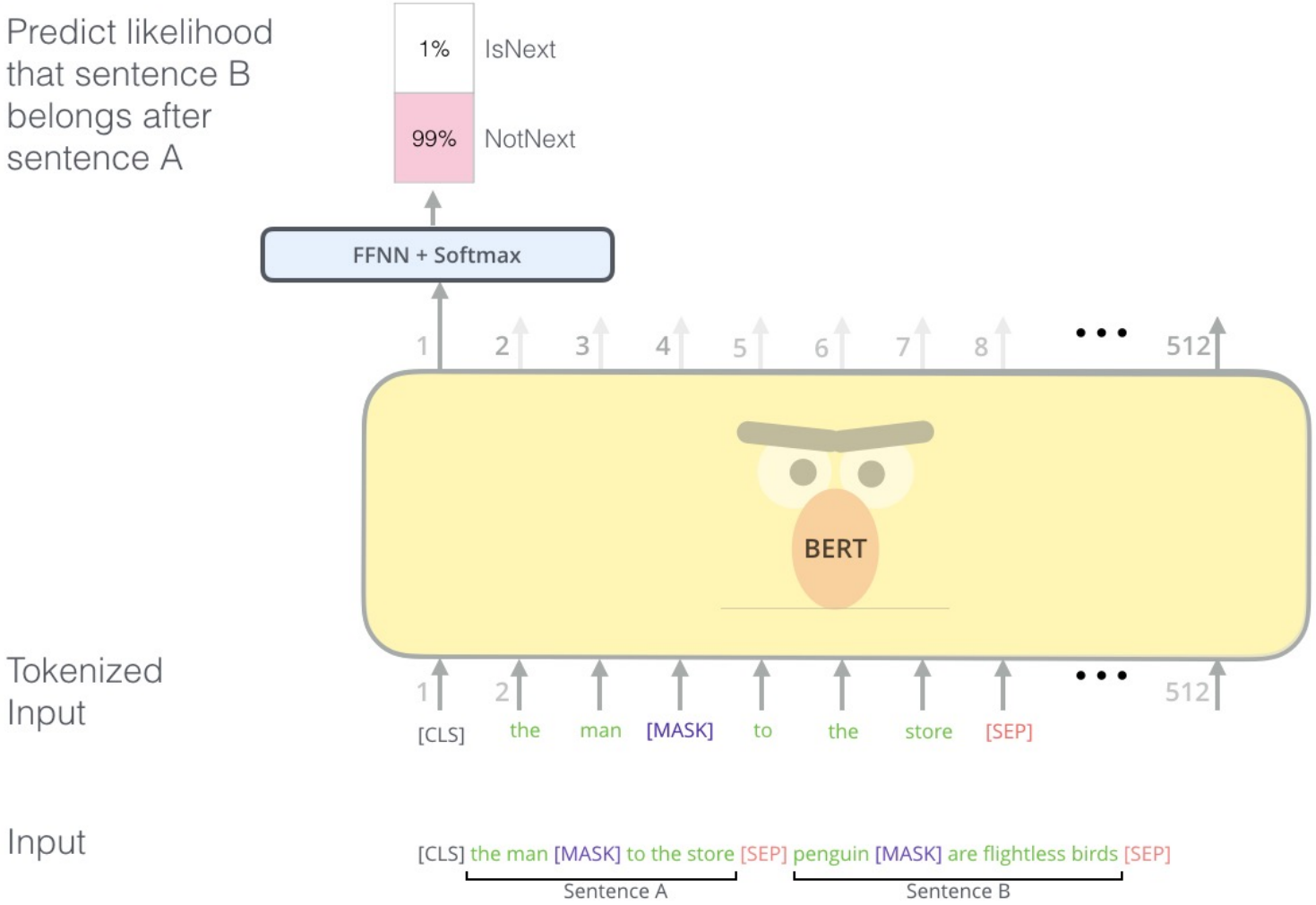
Pre-training Task: Mask LM



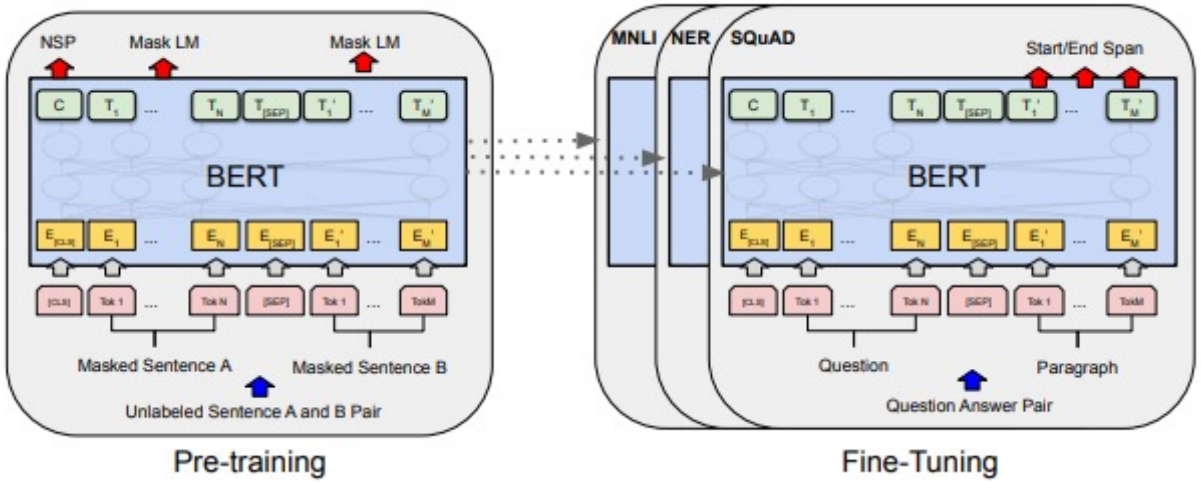
Pre-training Task: Mask LM

- Rather than *always* replacing the chosen words with `[MASK]`, the data generator will do the following:
- 80% of the time: Replace the word with the `[MASK]` token, e.g., `my dog is hairy` → `my dog is [MASK]`
- 10% of the time: Replace the word with a random word, e.g., `my dog is hairy` → `my dog is apple`
- 10% of the time: Keep the word unchanged, e.g., `my dog is hairy` → `my dog is hairy`. The purpose of this is to bias the representation towards the actual observed word.

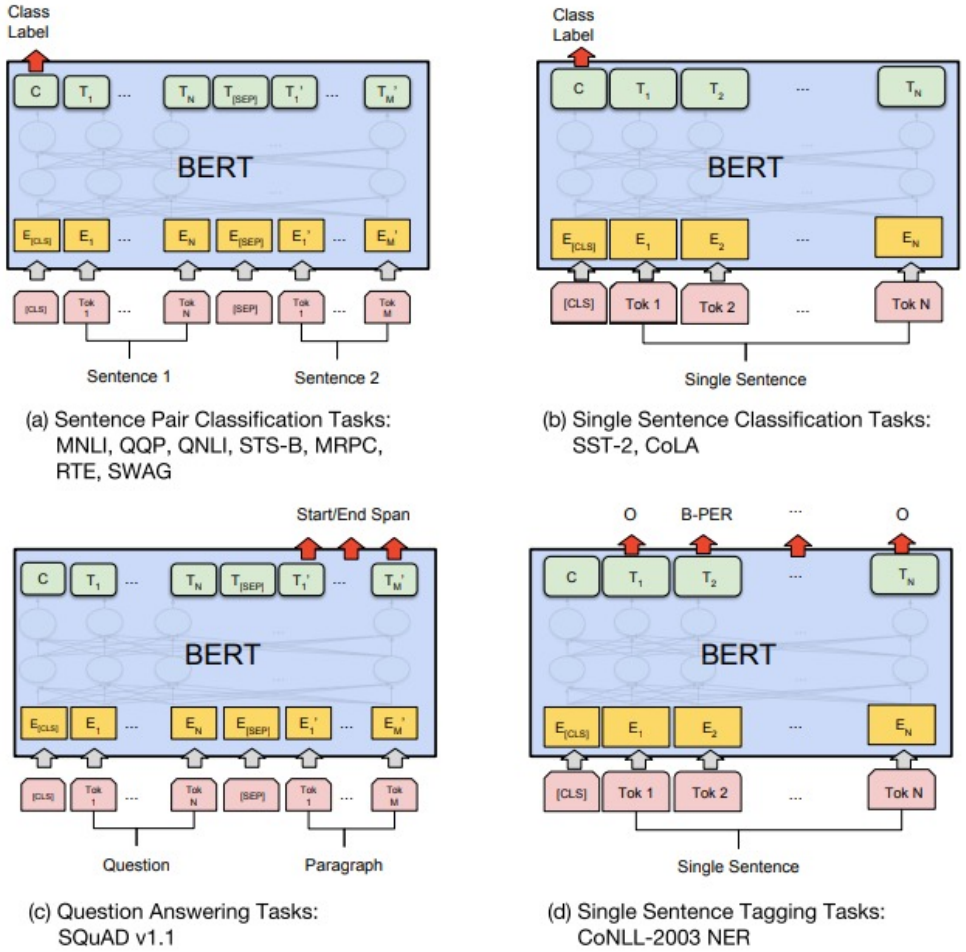
Pre-training Task: Next Sentence Prediction



Fine-tuning Tasks



Fine-tuning Tasks



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

(b) Single Sentence Classification Tasks: SST-2, CoLA

(c) Question Answering Tasks: SQuAD v1.1

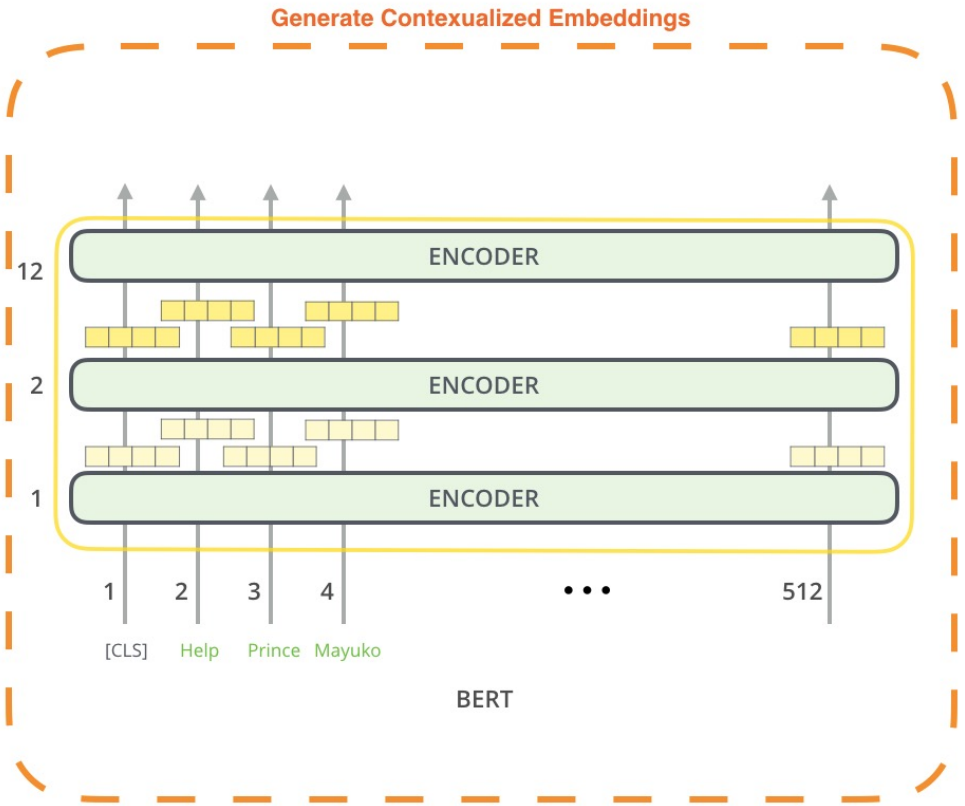
(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

Fine-tuning Tasks

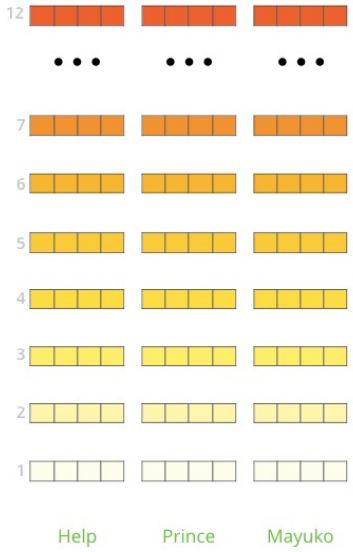
System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	**Average** -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

Table 1: GLUE Test results, scored by the evaluation server (https://gluebenchmark.com/leaderboard). The number below each task denotes the number of training examples. The "Average" column is slightly different than the official GLUE score, since we exclude the problematic WNLI set.[8] BERT and OpenAI GPT are single-model, single task. F1 scores are reported for QQP and MRPC, Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks. We exclude entries that use BERT as one of their components.

BERT for feature extraction

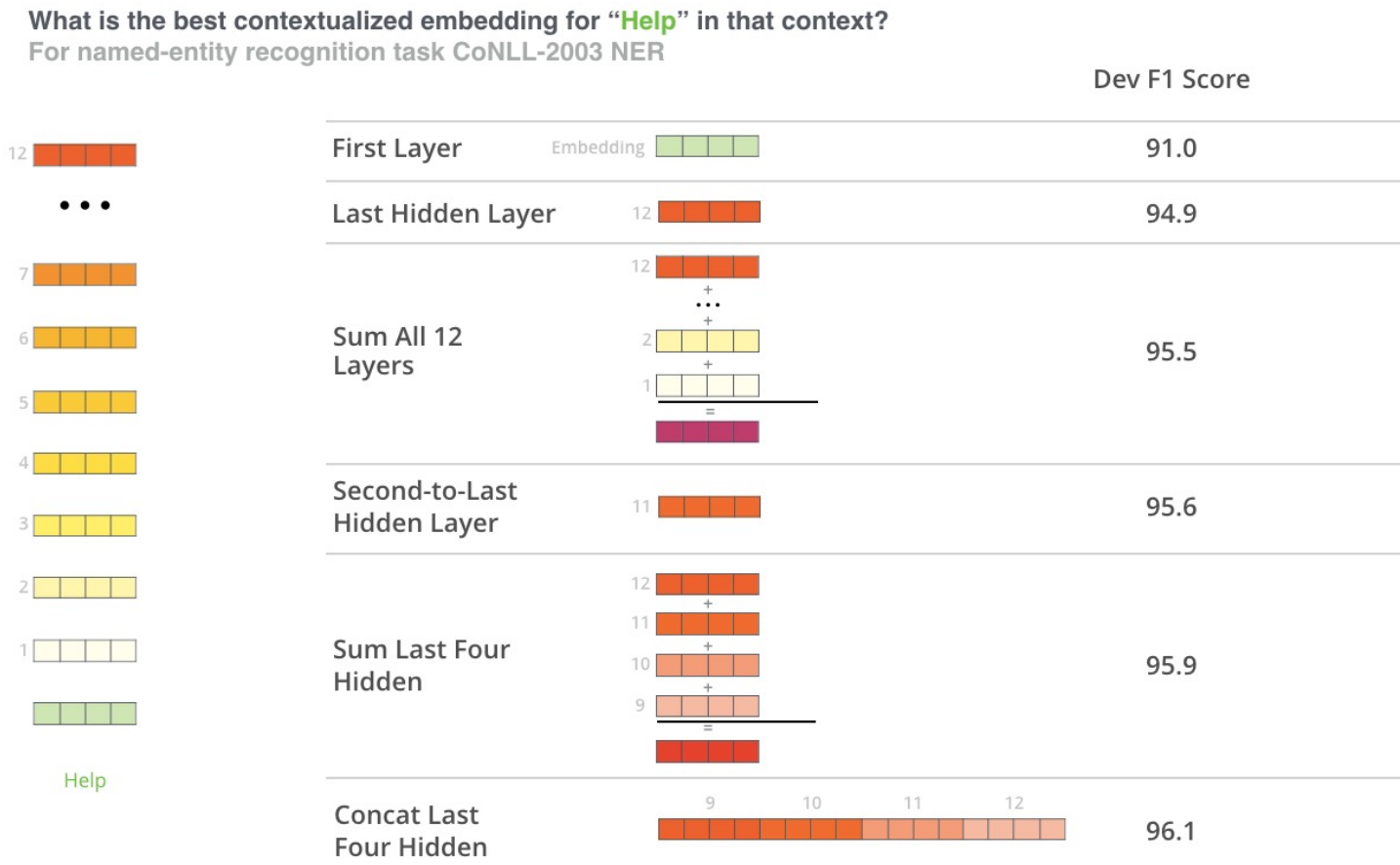


The output of each encoder layer along each token's path can be used as a feature representing that token.



But which one should we use?

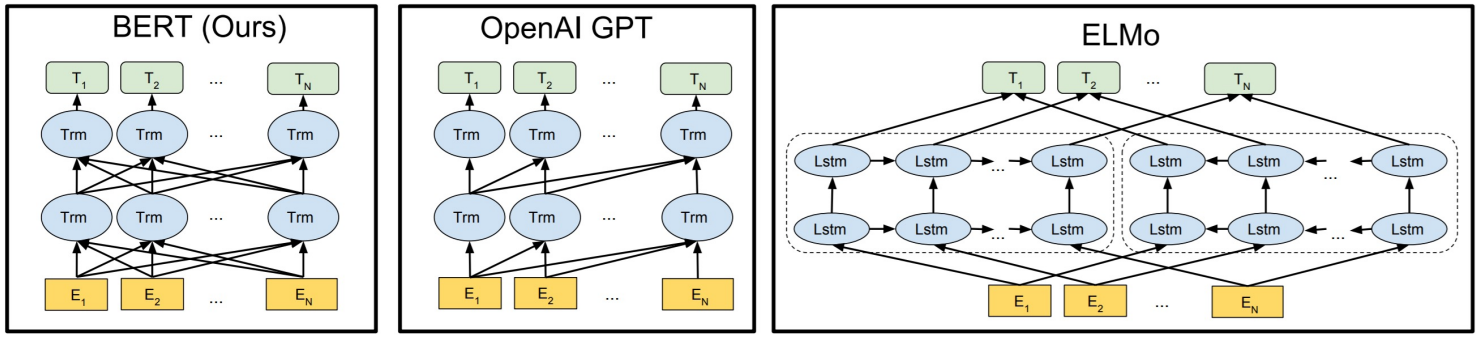
BERT for feature extraction

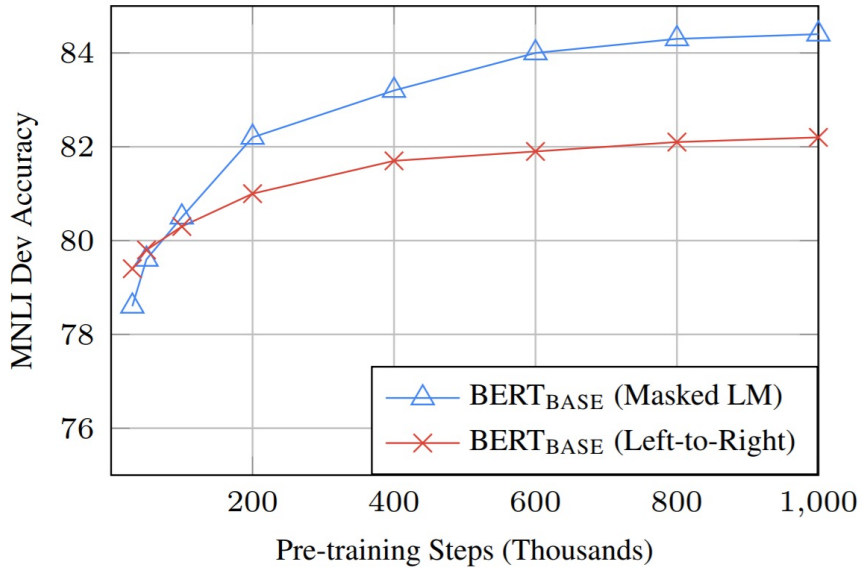


BERT for feature extraction

Layers	Dev F1
Finetune All	96.4
First Layer (Embeddings)	91.0
Second-to-Last Hidden	95.6
Last Hidden	94.9
Sum Last Four Hidden	95.9
Concat Last Four Hidden	96.1
Sum All 12 Layers	95.5

BERT vs. GPT vs. ELMo





Thank you for listening. Any question?

References

- **The Illustrated BERT, ELMo, and co.**
  - http://jalammar.github.io/illustrated-bert/
- **The Illustrated GPT-2**
  - http://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling