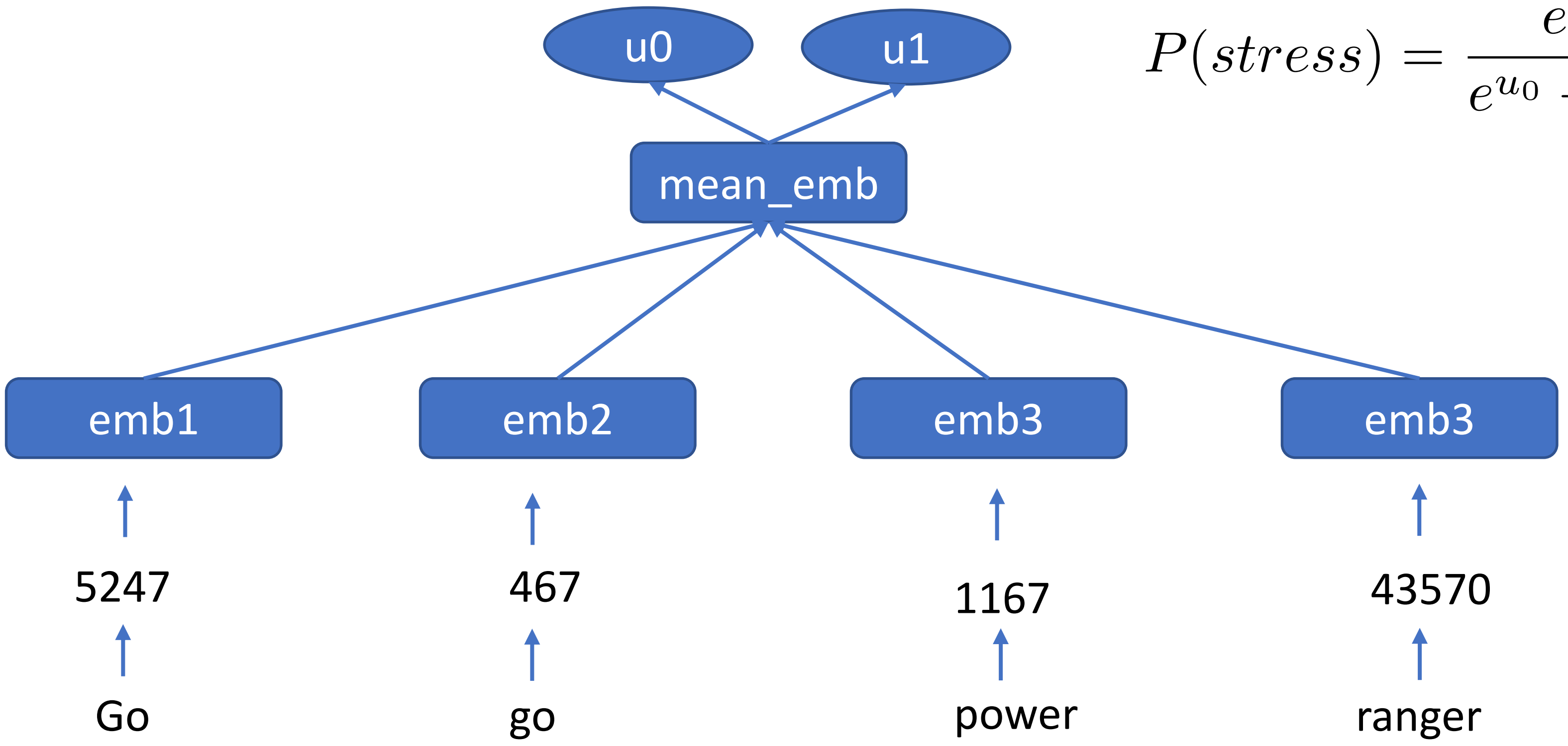
Gen-AI: Technical and Social

Lecture 03: Transformer and Text Generation

Recap: Text Classification

- A simple model



$$P(stress) = \frac{e^{u_1}}{e^{u_0} + e^{u_1}}$$

Tokenization

Recap: Text Classification

- A simple model

```
class EmbeddingModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_classes):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.fc1 = nn.Linear(embedding_dim, num_classes)

    def forward(self, inputs):
        output = self.embedding(inputs)
        output = torch.mean(output, dim=1)  # Mean pooling
        output = self.fc1(output)
        return output

model = EmbeddingModel(tokenizer.vocab_size, 128, 128, 2)
```

Issues of Word Embeddings?

- 1. Does not take the surrounding contexts into account
- 2. Does not take care of the order of the words

Context matters

Word	Example Contexts
it	The animal didn't cross the street because it was too tired The animal didn't cross the street because it was too wide
station	The train left the station on time The radio station was playing 60s hits I was stationed on a remote island in Polynesia

Order matter

She only eats pizza -> Only she eats pizza
The dog chased the cat -> The cat chased the dog.
….

Transformer

By failing to prepare, you are preparing to fail. ---Benjamin Franklin

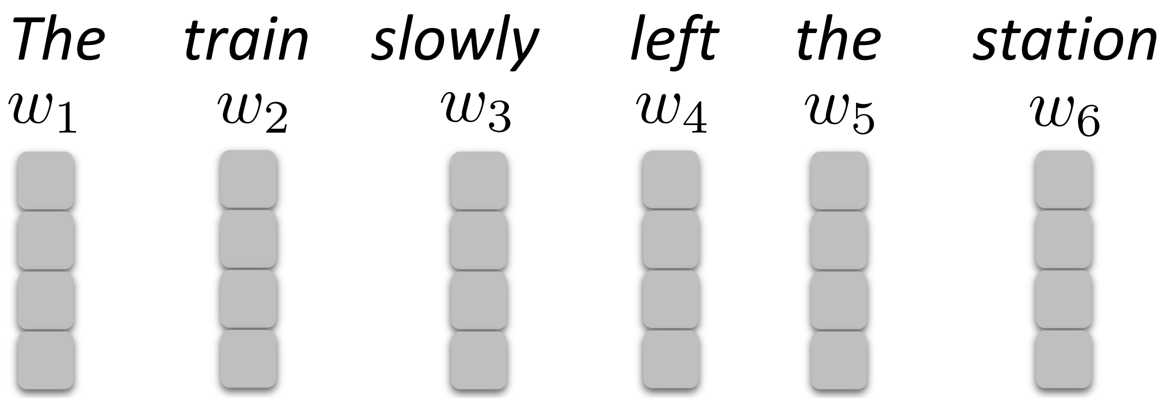
Let's prepare a right architecture…

We will focus on this first

How to take the surrounding context of each word into account

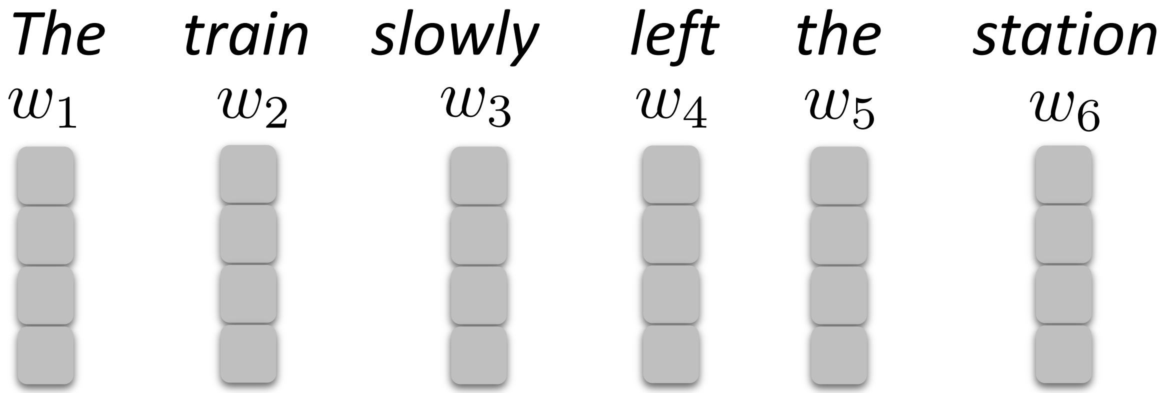
How to take the order of the words into account

From embeddings to *contextual embeddings*



- We can easily get stand-alone embeddings for all the words

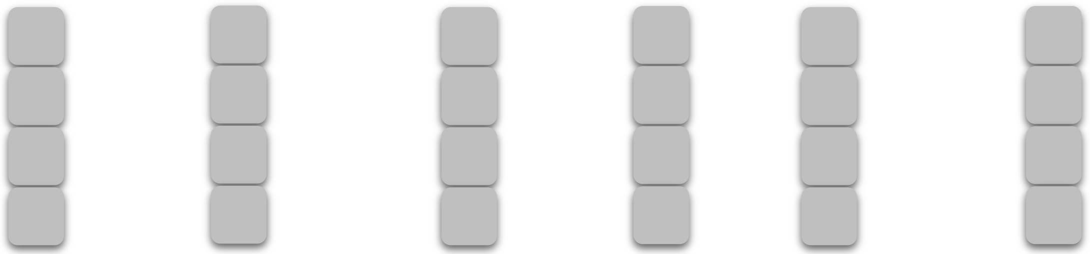
From embeddings to *contextual embeddings*



- We can easily get stand-alone embeddings for all the words
- How can we modify station's embedding so that it incorporates the other words?

From embeddings to *contextual embeddings*

The	*train*	*slowly*	*left*	*the*	*station*
w_1	w_2	w_3	w_4	w_5	w_6



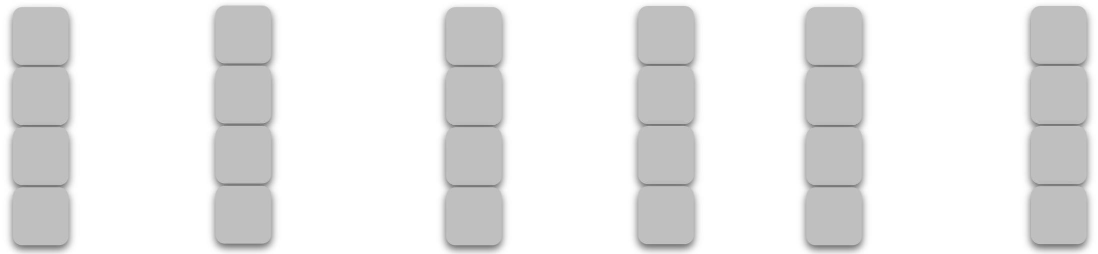
0.05	0.3	0.12	0.08	0.05	0.4

Imagine that we somehow know how much <u>attention</u> to give the other words i.e., how much <u>weight</u> to give the other words

From embeddings to *contextual embeddings*

The *train* *slowly* *left* *the* *station*

w_1 w_2 w_3 w_4 w_5 w_6

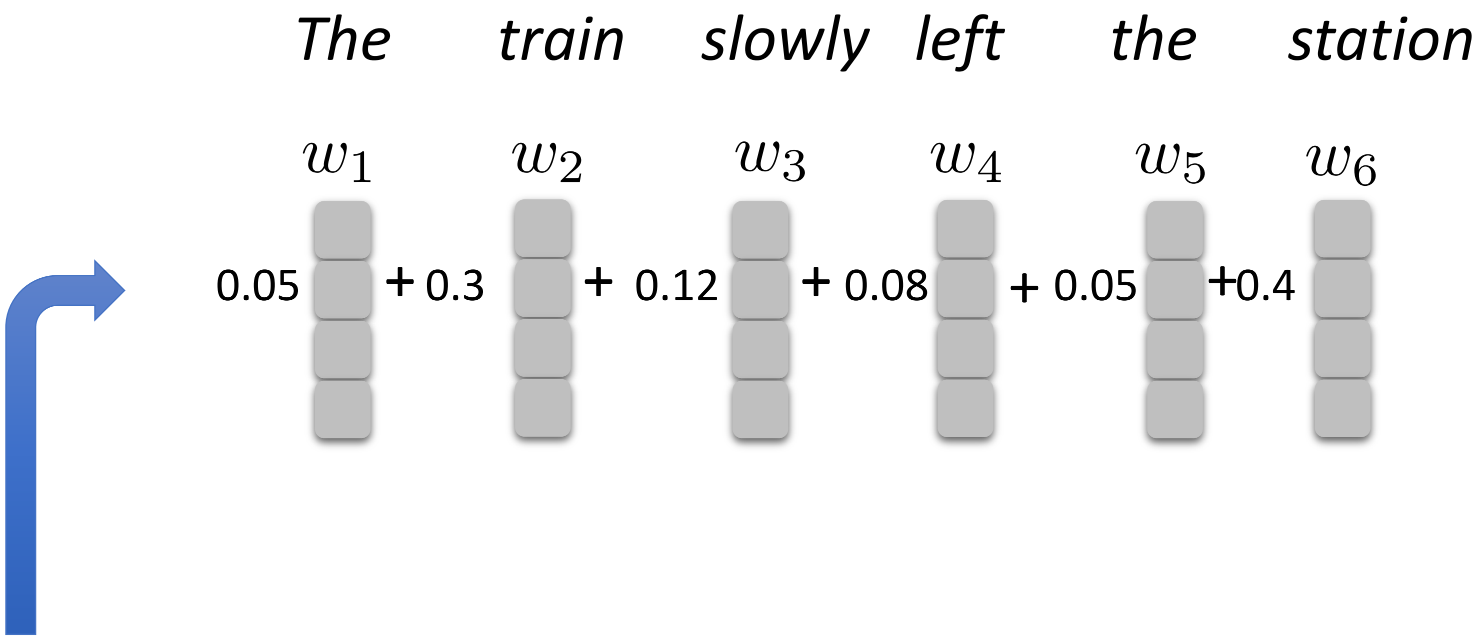


Imagine that we somehow know how much attention to give the other words i.e., how much weight to give the other words

0.05	0.3	0.12	0.08	0.05	0.4

Could we use these weights to "contextualize" the stand-alone embedding w_6 for "station"?

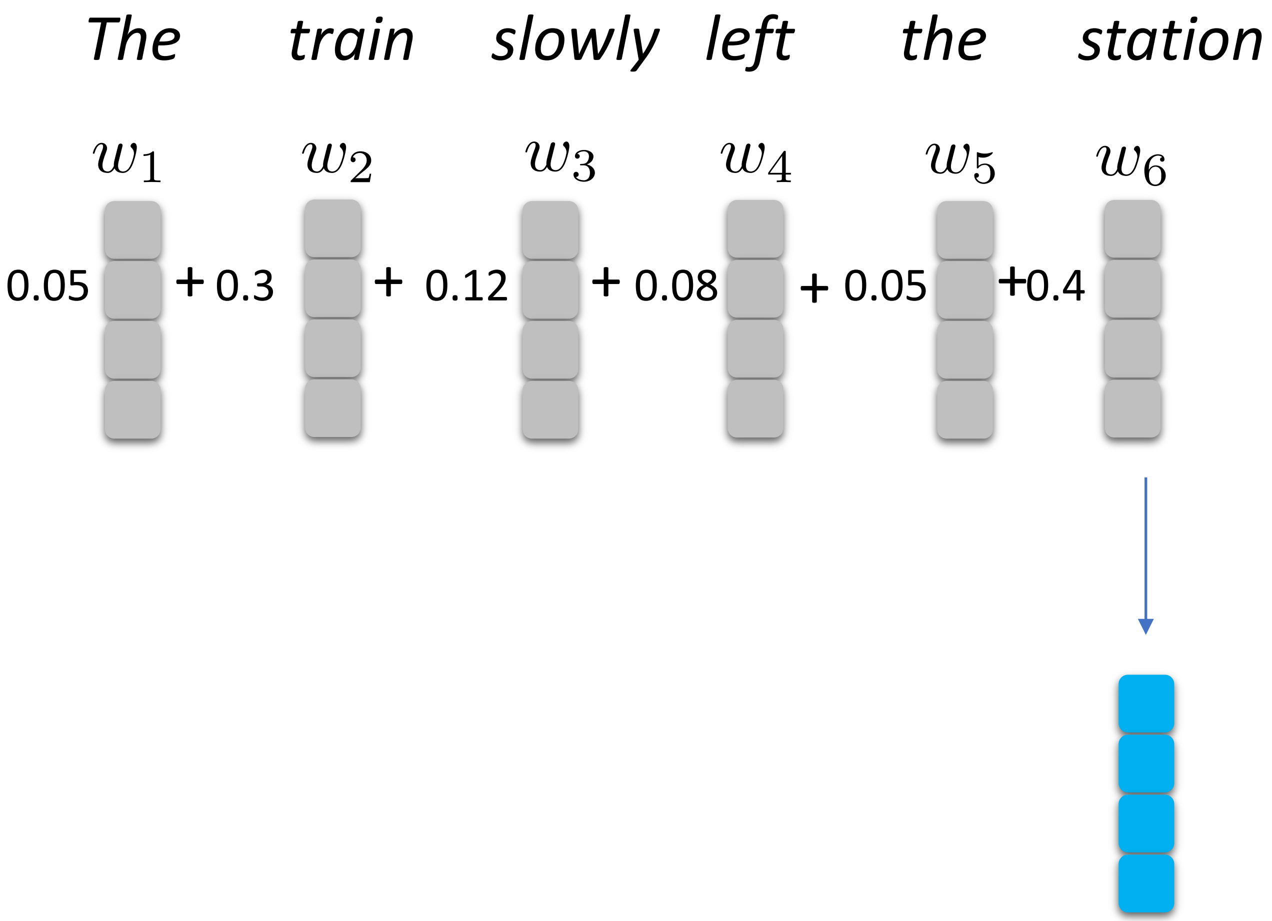
From embeddings to *contextual embeddings*



0.05	0.3	0.12	0.08	0.05	0.4

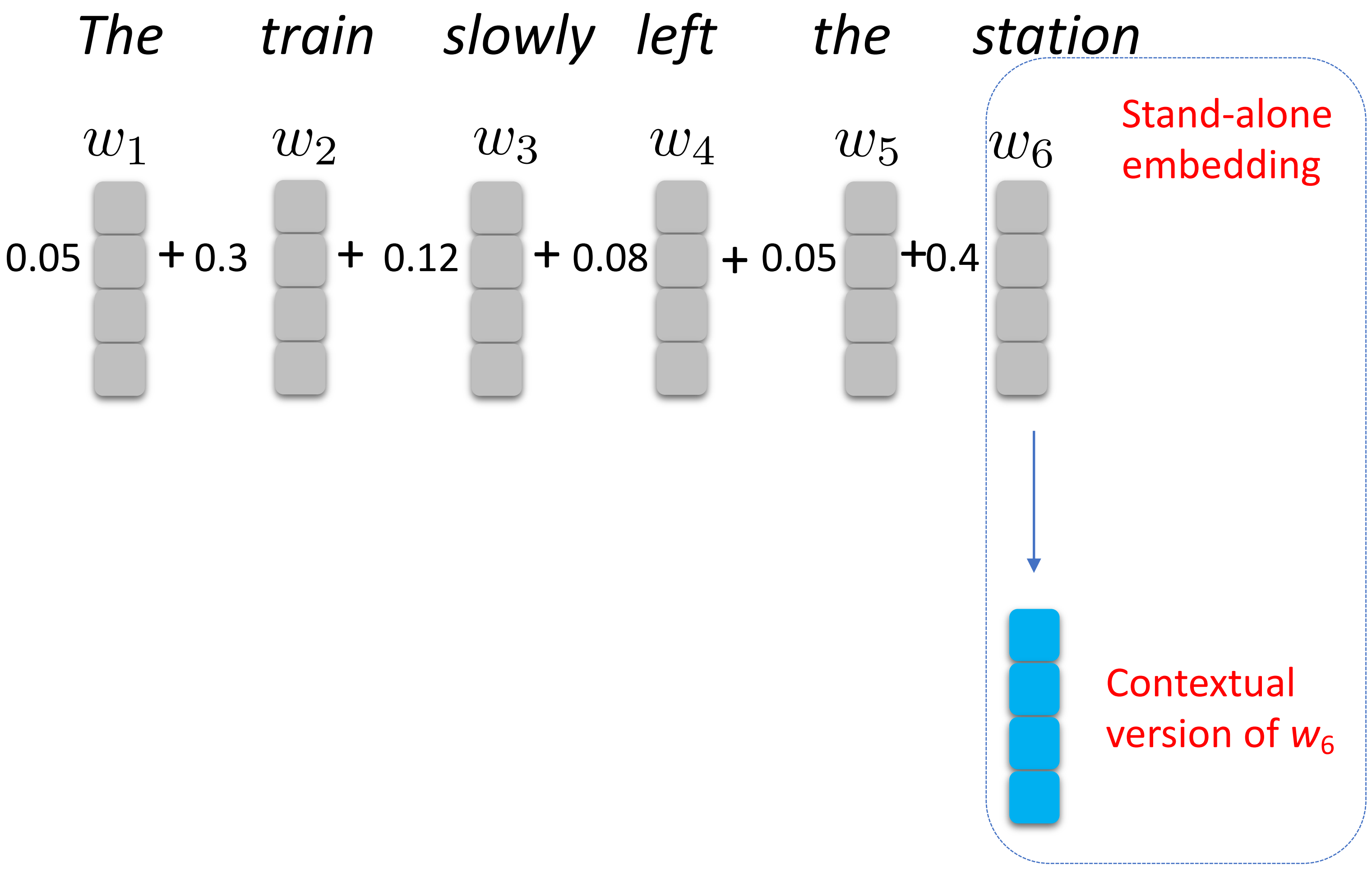
<u>Idea</u>: For each word, we can calculate a <u>weighted average</u> of the stand-alone embeddings of *all* the words in the sentence

From embeddings to *contextual embeddings*

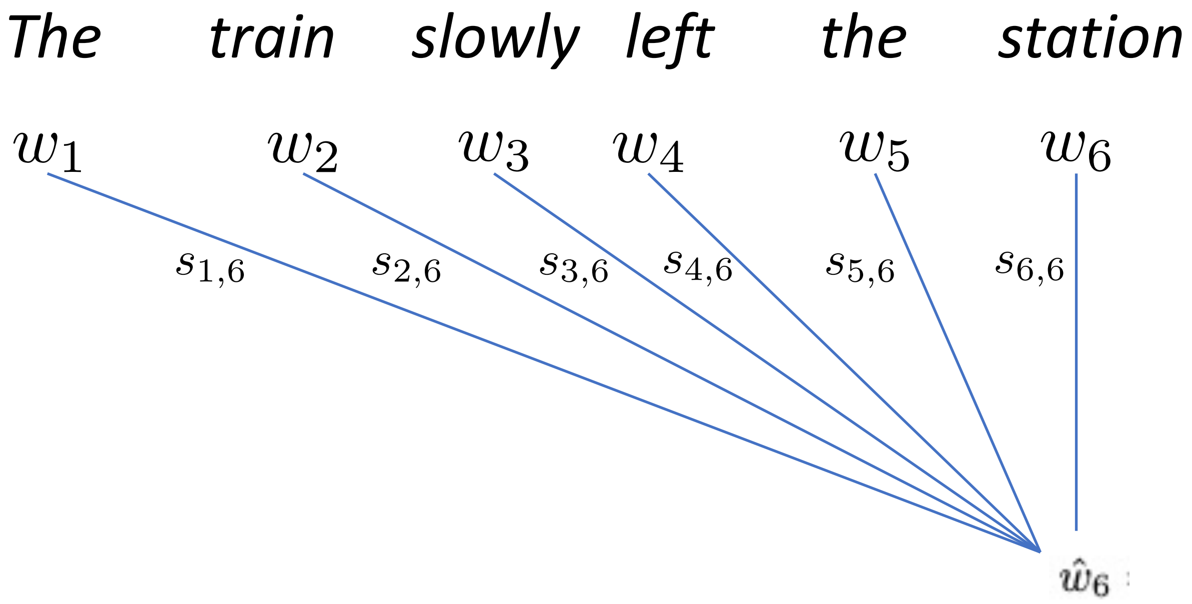


Idea: For each word, we can calculate a weighted average of the stand-alone embeddings of *all* the words in the sentence

From embeddings to *contextual embeddings*



Same thing but more abstract



Stand-alone embedding

$$\hat{w}_6 = s_{1,6}w_1 + s_{2,6}w_2 + s_{3,6}w_3 + s_{4,6}w_4 + s_{5,6}w_5 + s_{6,6}w_6$$

Contextual version of w_6

For a given word (e.g., ‘station’), how should the weights of the other words be chosen?

For a given word (e.g., ‘station’), how should the weights of the other words be chosen?

Intuition

- The weight of a word should be proportional to how related it is to the word “station”

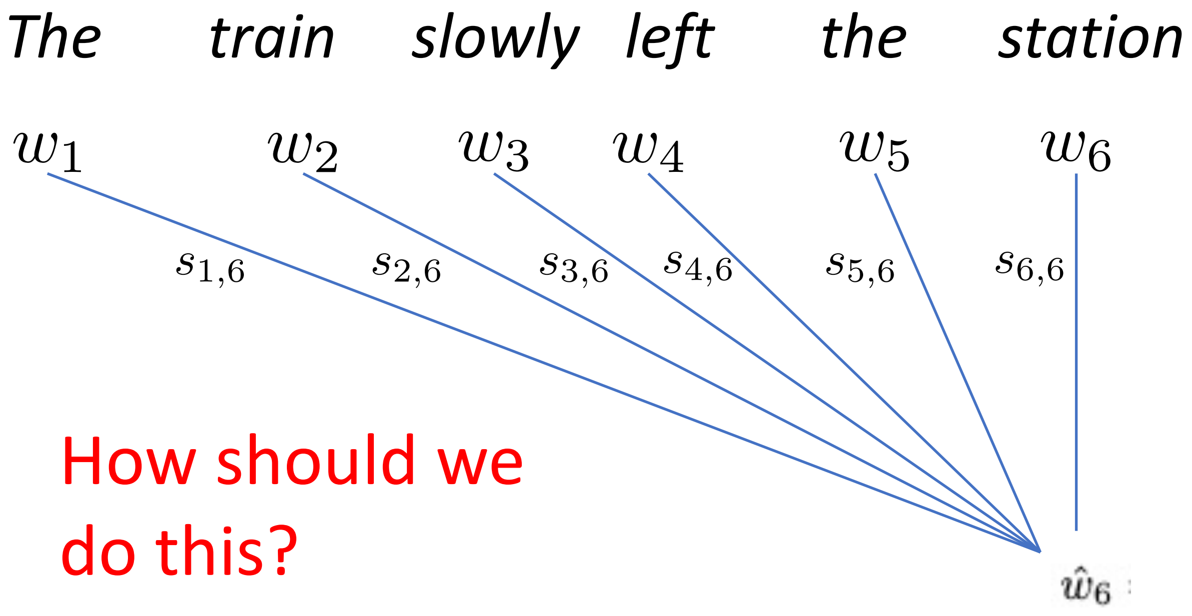
For a given word (e.g., ‘station’), how should the weights of the other words be chosen?

Intuition

- The weight of a word should be proportional to how related it is to the word “station”

- One way to quantify how “related” two words are: the *dot-product* of their stand-alone embeddings

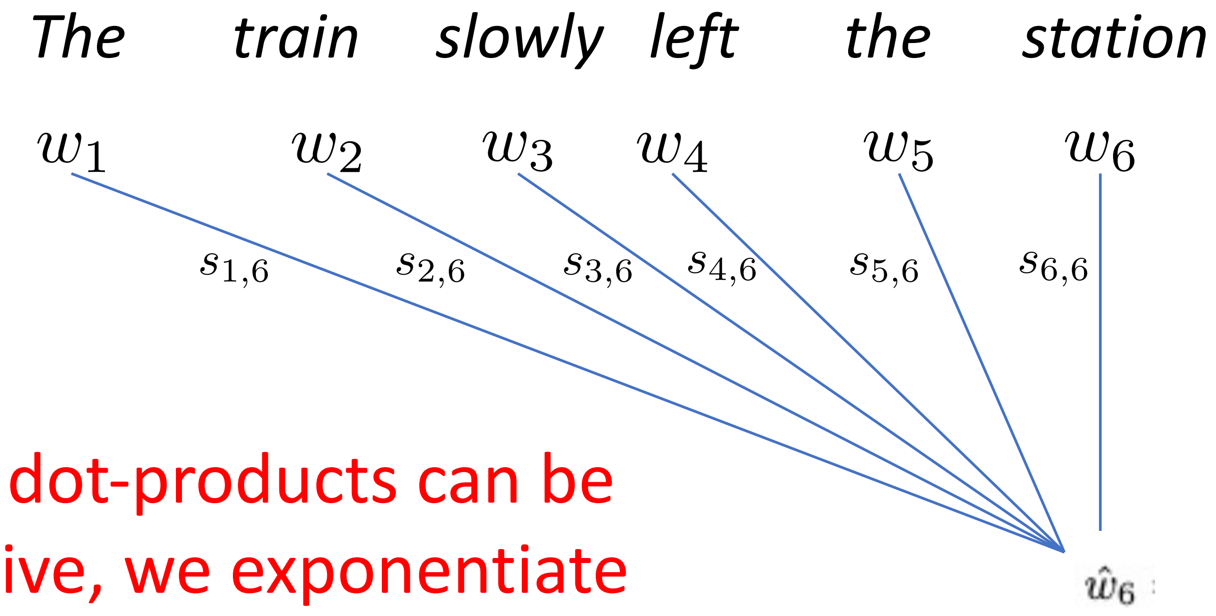
We need to convert dot-products to proper weights*



How should we do this?

* non-negative, and summing to 1.0

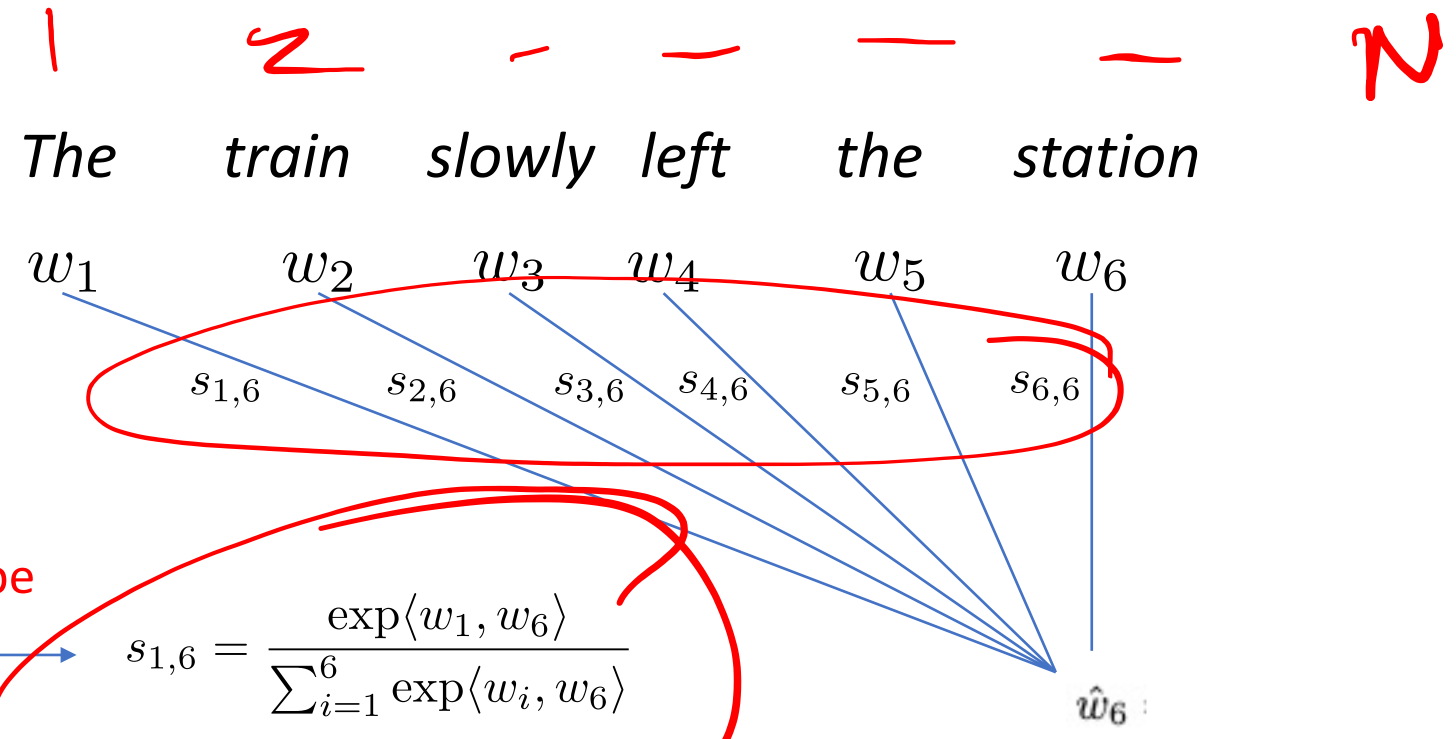
We need to convert dot-products to proper weights*



Since dot-products can be negative, we exponentiate them and then normalize (remember softmax?)

* non-negative, and summing to 1.0

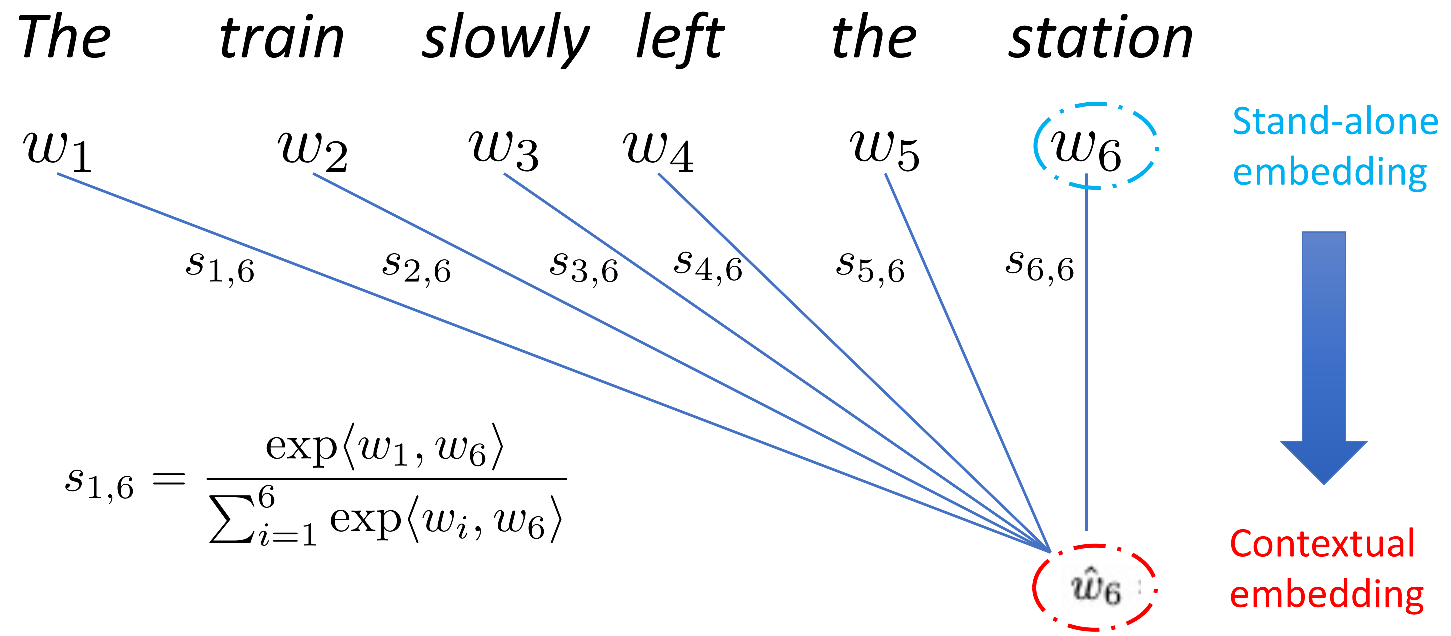
Normalized attention weights



Since dot-products can be negative, we exponentiate them and then normalize

$$s_{1,6} = \frac{\exp\langle w_1, w_6\rangle}{\sum_{i=1}^{6} \exp\langle w_i, w_6\rangle}$$

From embedding to *contextual* embedding!



$$s_{1,6} = \frac{\exp\langle w_1, w_6 \rangle}{\sum_{i=1}^{6} \exp\langle w_i, w_6 \rangle}$$

$$\hat{w}_6 = s_{1,6}w_1 + s_{2,6}w_2 + s_{3,6}w_3 + s_{4,6}w_4 + s_{5,6}w_5 + s_{6,6}w_6$$

By choosing weights in this manner, the embedding of a word moves closer to the embeddings of the other words in the current context, in proportion to how related they are

- The word ‘station’ has many contexts.

By choosing weights in this manner, the embedding of a word moves closer to the embeddings of the other words in the current context, in proportion to how related they are

- The word ‘station’ has many contexts.
    - In the current context, ‘train’ is closely related to ‘station’ and therefore exerts a strong “pull” on it

By choosing weights in this manner, the embedding of a word moves closer to the embeddings of the other words in the current context, in proportion to how related they are

- The word ‘station’ has many contexts.
  - In the current context, ‘train’ is closely related to ‘station’ and therefore exerts a strong “pull” on it
  - ‘radio’ is also related to ‘station’ but doesn’t appear in the current context so (automatically) has zero weight

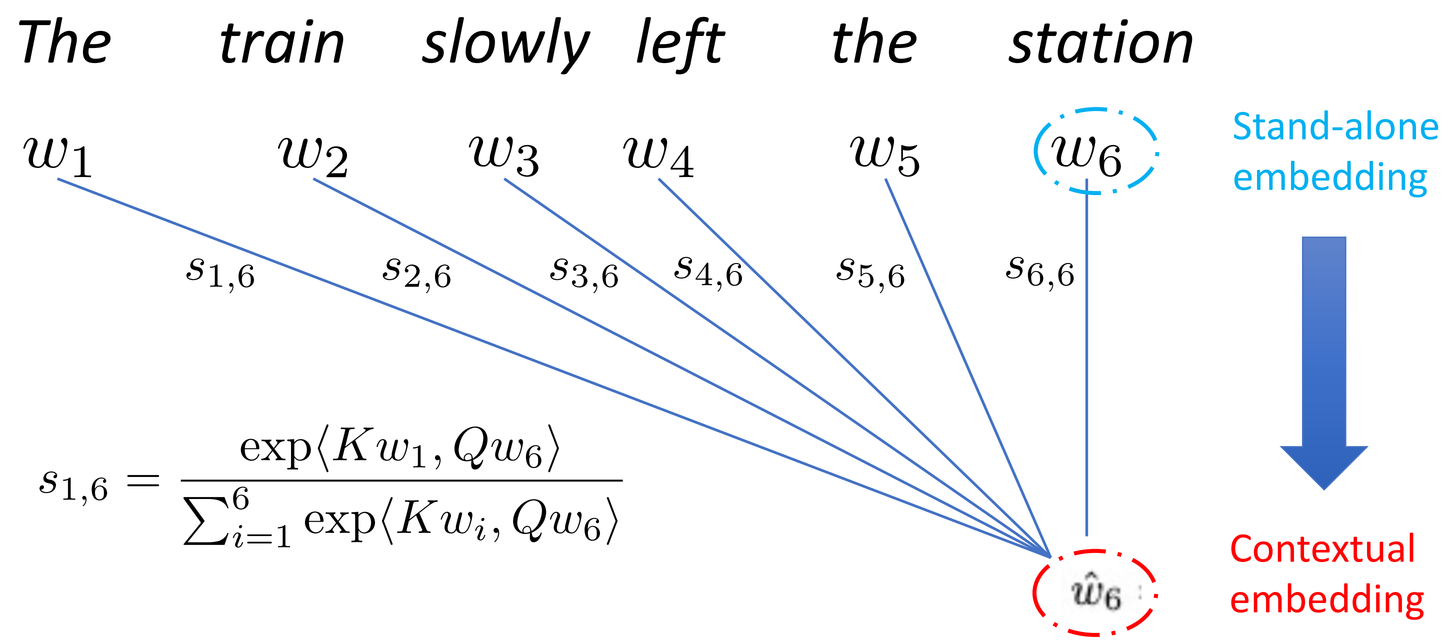
By choosing weights in this manner, the embedding of a word moves closer to the embeddings of the other words in the current context, in proportion to how related they are

- The word ‘station’ has many contexts.
  - In the current context, ‘train’ is closely related to ‘station’ and therefore exerts a strong “pull” on it
  - ‘radio’ is also related to ‘station’ but doesn’t appear in the current context so (automatically) has zero weight

- By moving station closer to train (equivalently – paying more “attention” to train), we are contextualizing station’s embedding to the context of trains, platforms, departures, etc.

From embedding to *contextual* embedding!

Make it more flexible

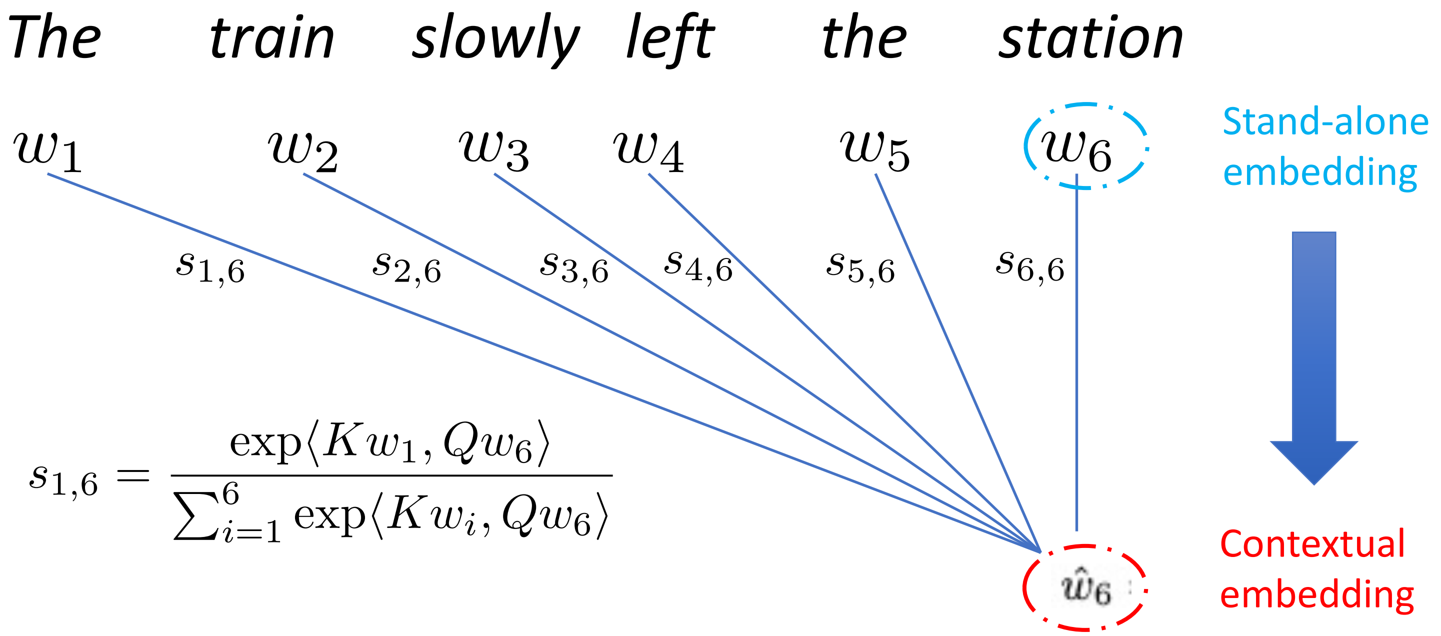


$$s_{1,6} = \frac{\exp\langle K w_1, Q w_6 \rangle}{\sum_{i=1}^{6} \exp\langle K w_i, Q w_6 \rangle}$$

$$\boxed{\hat{w}_6 = s_{1,6} w_1 + s_{2,6} w_2 + s_{3,6} w_3 + s_{4,6} w_4 + s_{5,6} w_5 + s_{6,6} w_6}$$

From embedding to *contextual* embedding!

Make it more flexible:
- e.g., in K=transportation subspace, Q=location subspace, how closely two words are related?



$$s_{1,6} = \frac{\exp\langle K w_1, Q w_6 \rangle}{\sum_{i=1}^{6} \exp\langle K w_i, Q w_6 \rangle}$$

$$\hat{w}_6 = s_{1,6} w_1 + s_{2,6} w_2 + s_{3,6} w_3 + s_{4,6} w_4 + s_{5,6} w_5 + s_{6,6} w_6$$

From embedding to *contextual* embedding!

The *train* *slowly* *left* *the* *station*

w_1 w_2 w_3 w_4 w_5 w_6

Stand-alone embedding

$s_{1,6}$ $s_{2,6}$ $s_{3,6}$ $s_{4,6}$ $s_{5,6}$ $s_{6,6}$

Make it more flexible:
- e.g., in K=transportation subspace, Q=location subspace, how closely two words are related?

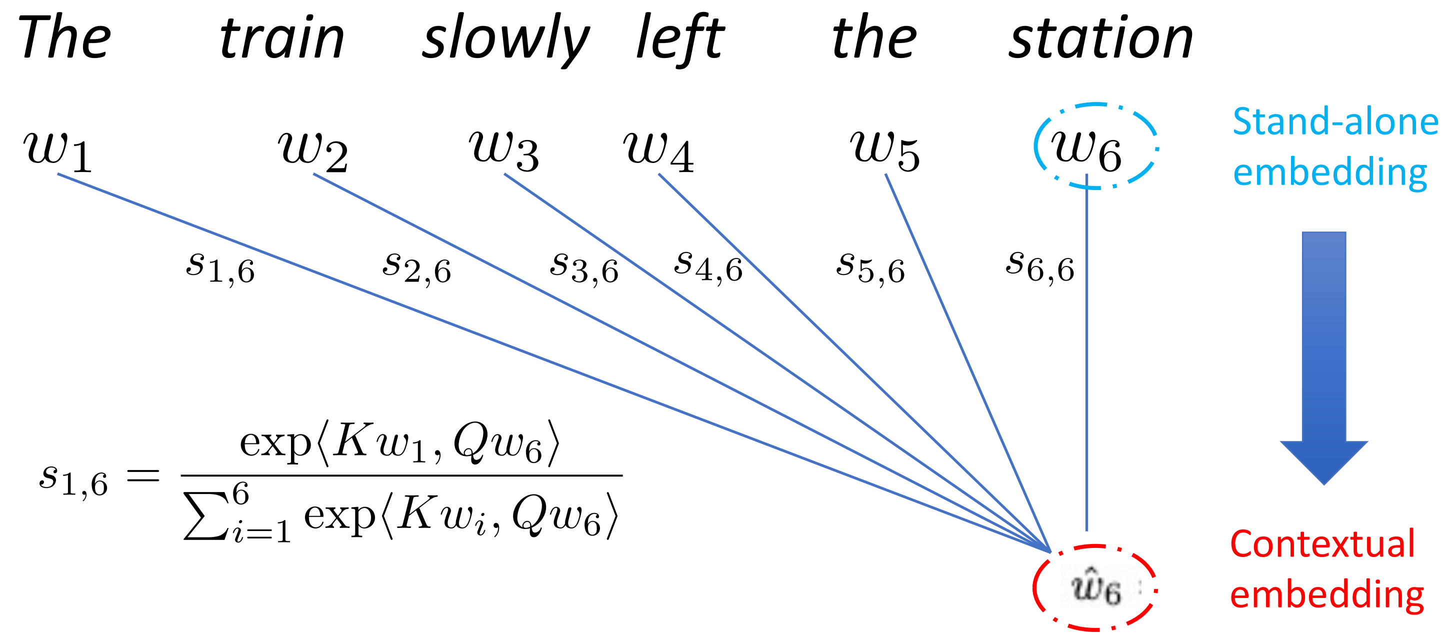
$$s_{1,6} = \frac{\exp\langle Kw_1, Qw_6\rangle}{\sum_{i=1}^{6} \exp\langle Kw_i, Qw_6\rangle}$$

$\hat{w}_6$

Contextual embedding

$$\boxed{\hat{w}_6 = s_{1,6}Vw_1 + s_{2,6}Vw_2 + s_{3,6}Vw_3 + s_{4,6}Vw_4 + s_{5,6}Vw_5 + s_{6,6}Vw_6}$$

From embedding to *contextual* embedding!



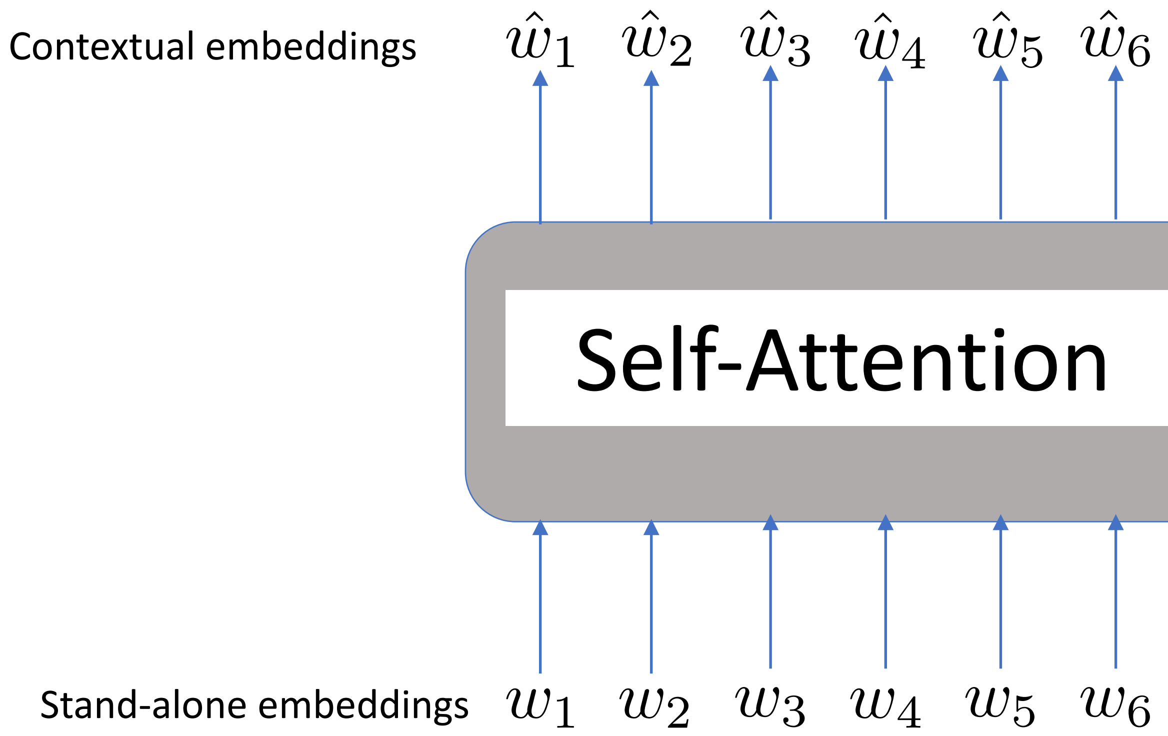
Make it more flexible:
- e.g., in K=transportation subspace, Q=location subspace, how closely two words are related?

$$s_{1,6} = \frac{\exp\langle Kw_1, Qw_6\rangle}{\sum_{i=1}^{6} \exp\langle Kw_i, Qw_6\rangle}$$

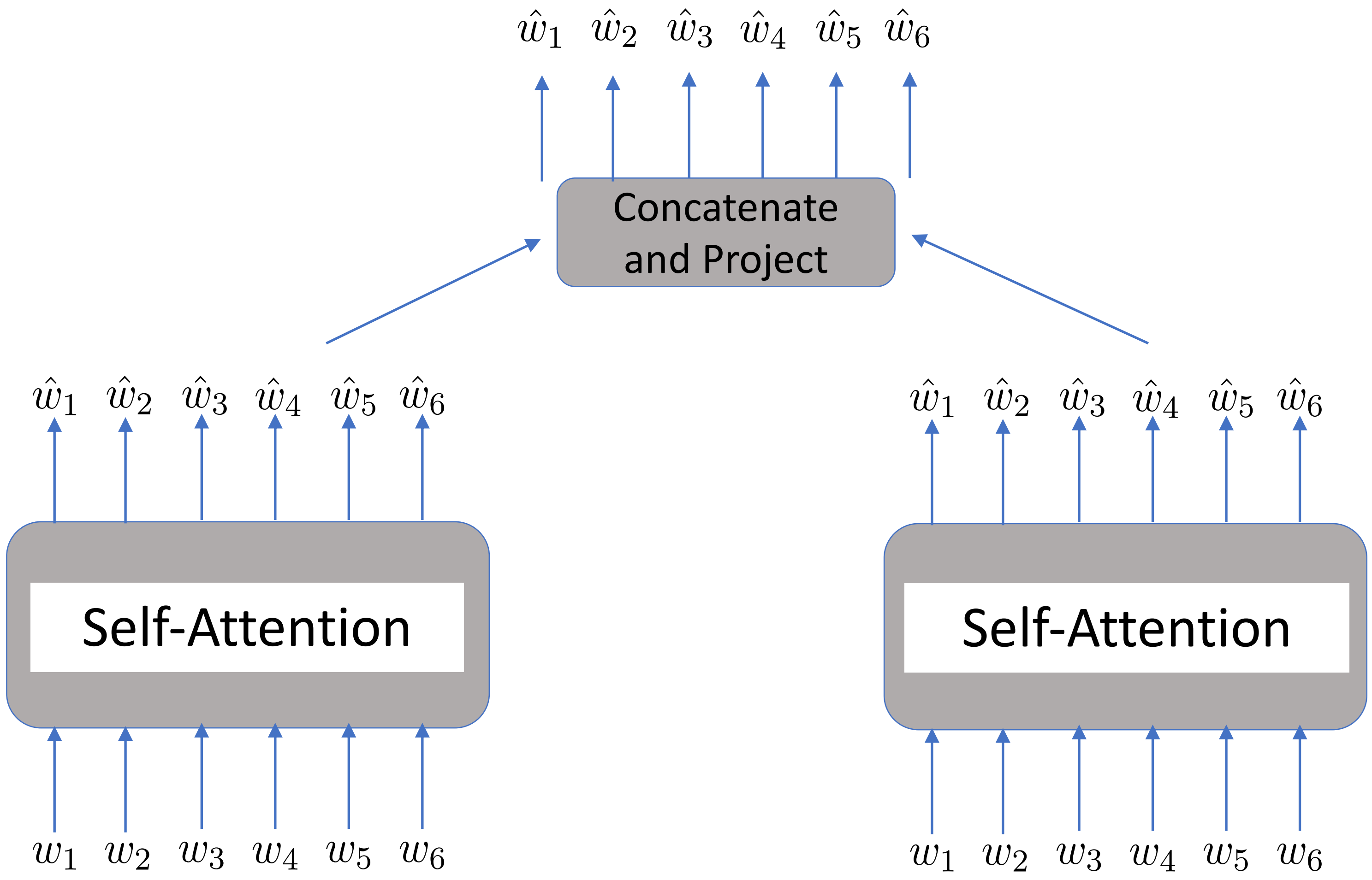
$$\boxed{\hat{w}_6 = s_{1,6}Vw_1 + s_{2,6}Vw_2 + s_{3,6}Vw_3 + s_{4,6}Vw_4 + s_{5,6}Vw_5 + s_{6,6}Vw_6}$$

Key (K), Query (Q), Value (V) projection are all tunable parameters: $K, Q, V \in R^{d \times d_{model}}$

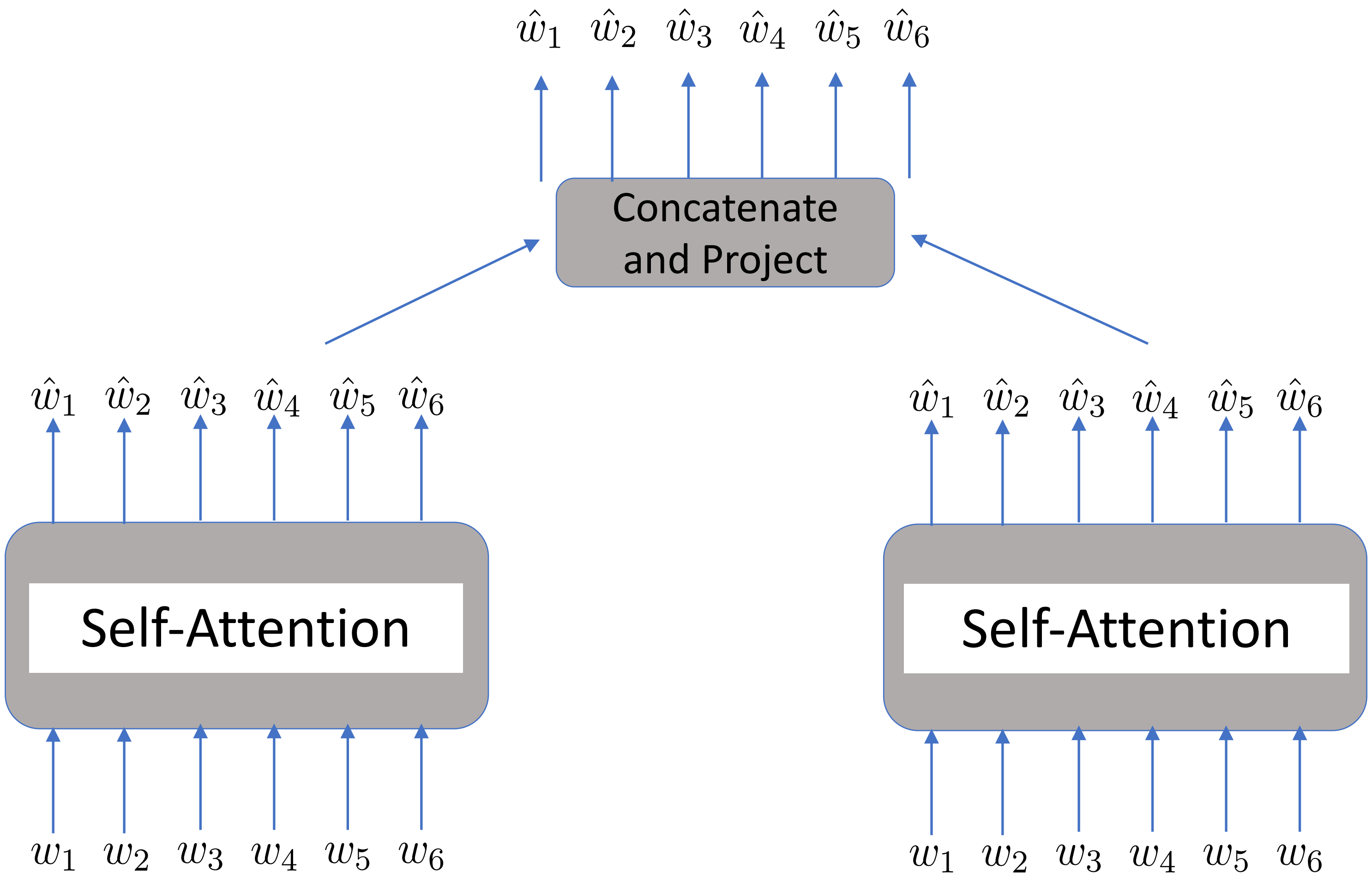
This operation is referred to as a 'Self Attention' layer and can be done very efficiently with matrix operations



Key Tweak: Multi*-Head Attention

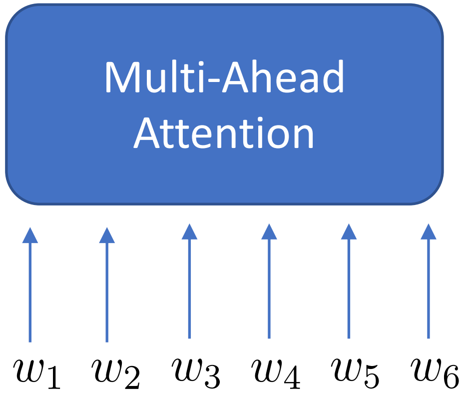


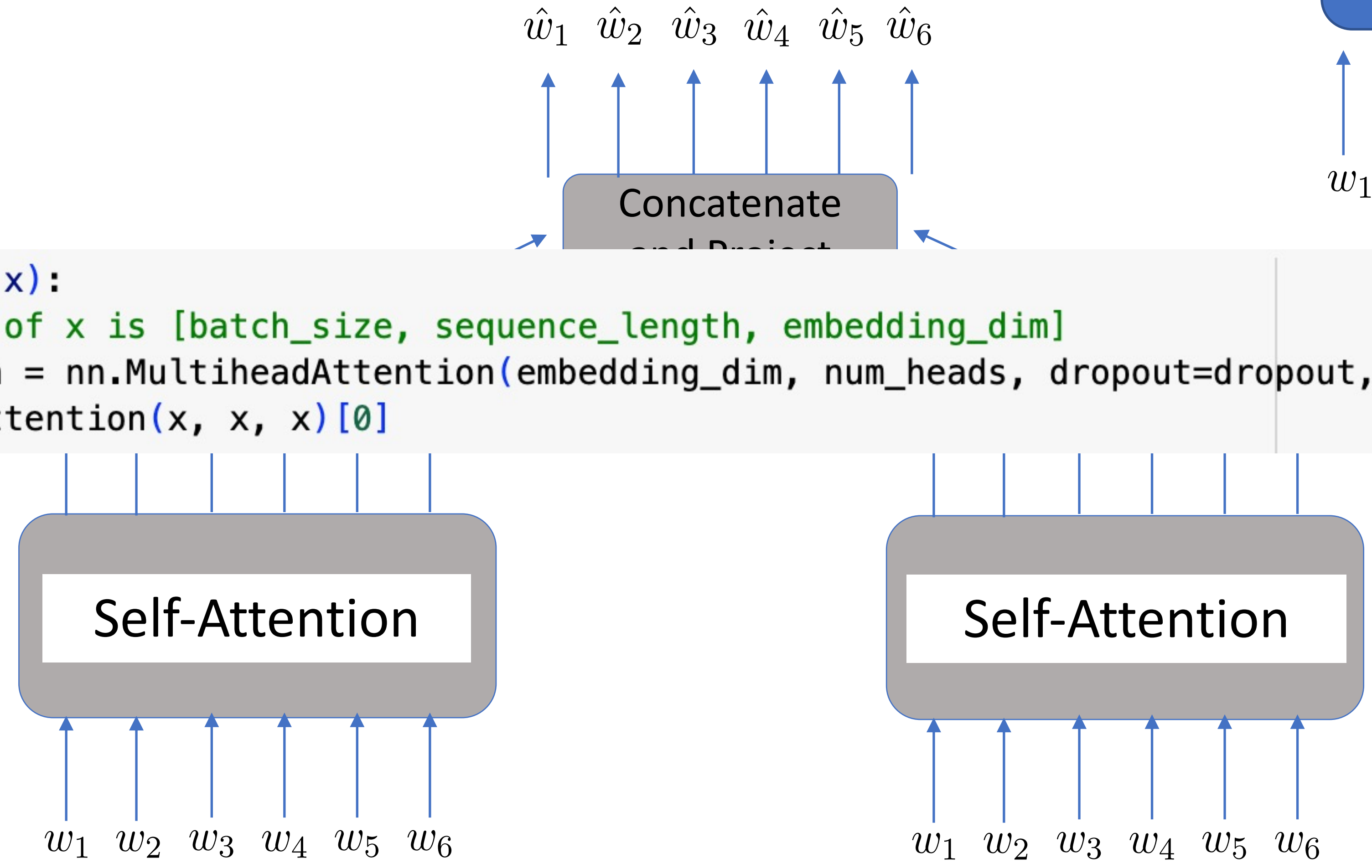
Key Tweak: Different attention 'heads' learn different patterns



Key Tweak: Different attention 'heads' learn different patterns

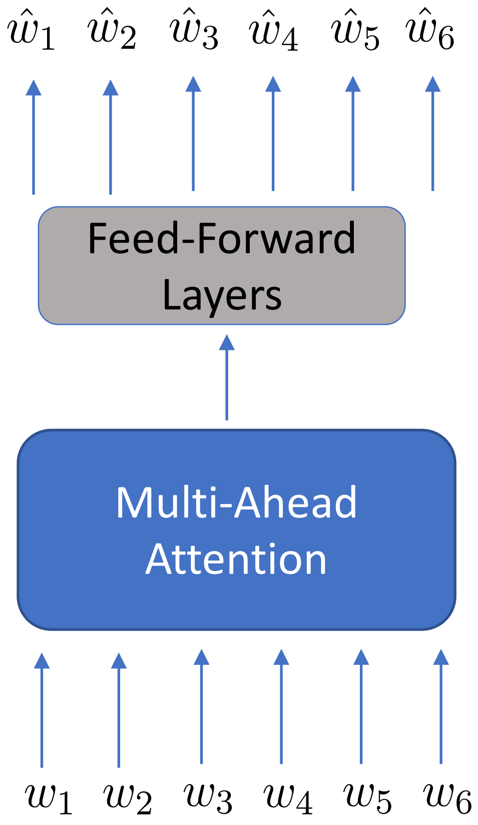
Combine together..



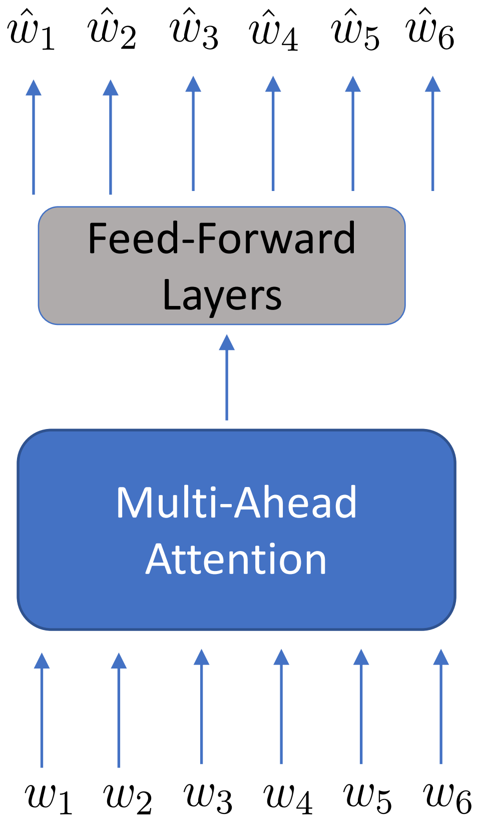



```python
def forward(self, x):
    ### dimension of x is [batch_size, sequence_length, embedding_dim]
    self.attention = nn.MultiheadAttention(embedding_dim, num_heads, dropout=dropout, batch_first=True)
    return self.attention(x, x, x)[0]
```

Key Tweak: Inject some non-linearity with feed-forward layers at the end

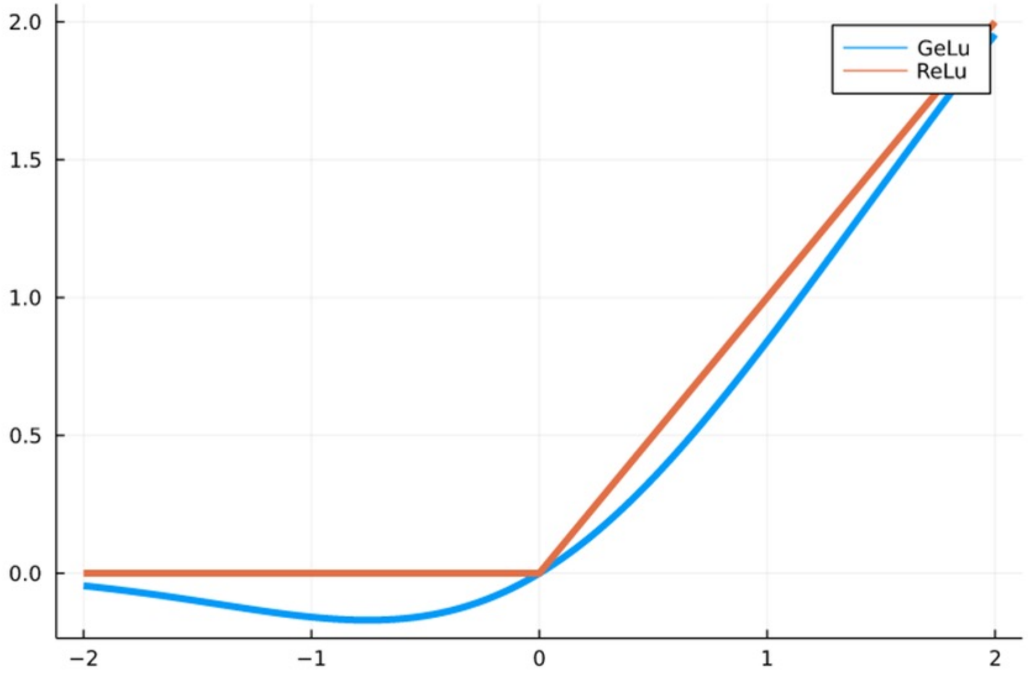


Key Tweak: Inject some non-linearity with feed-forward layers at the end



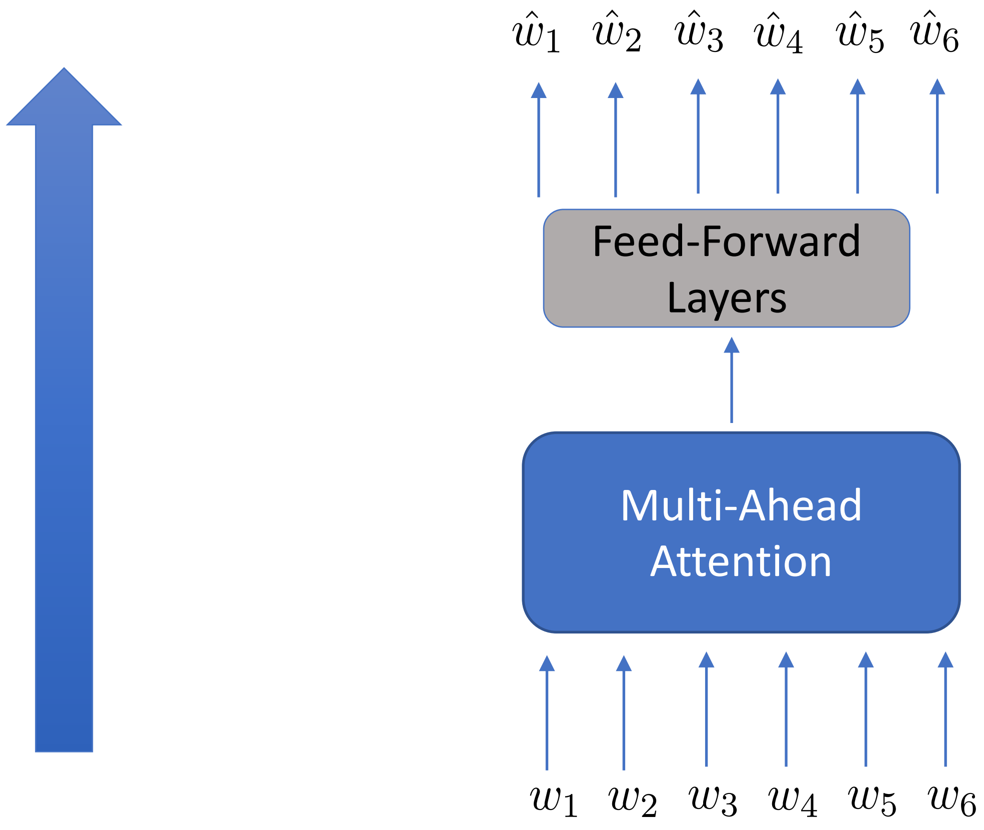

```
self.feed_forward = nn.Sequential(
    nn.Linear(embedding_dim, 4 * embedding_dim),
    nn.GELU(),
    nn.Linear(4 * embedding_dim, embedding_dim),
)
```

$$GELU(x) = xP(X \leq x), X \sim N(0,1)$$



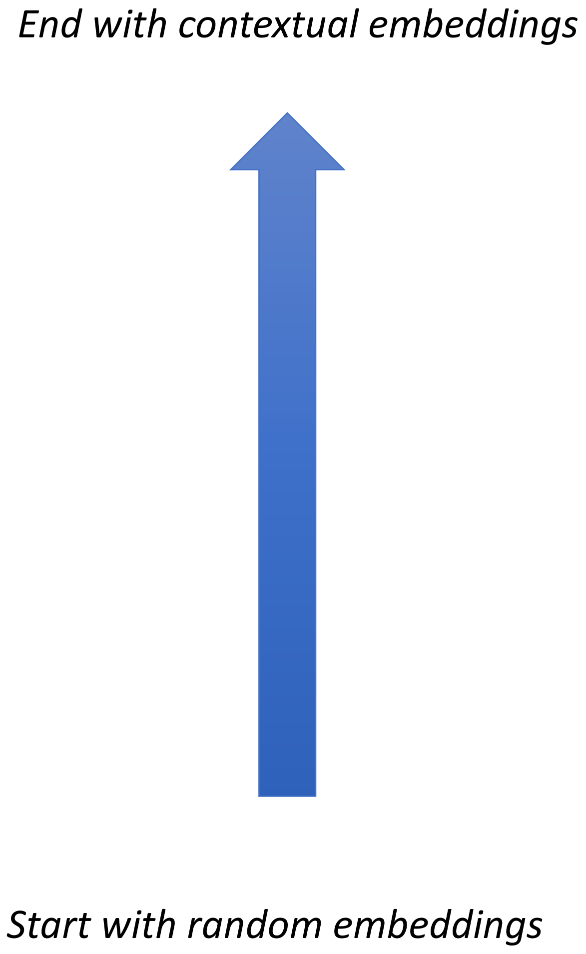
The story so far

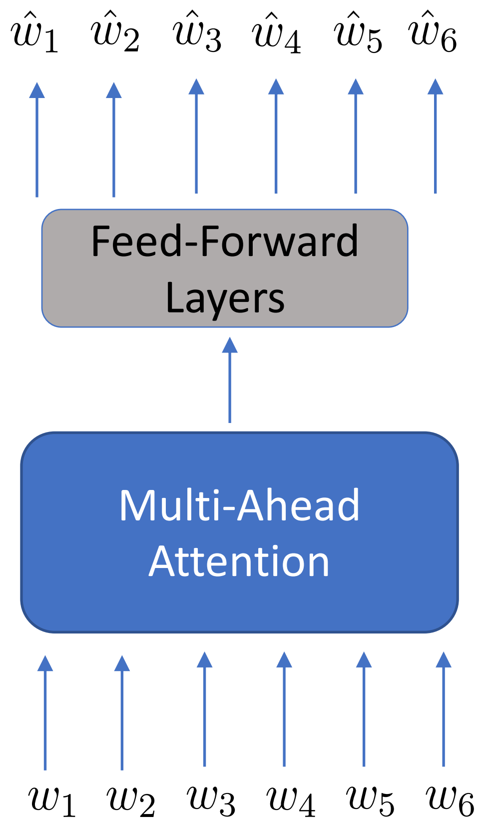
End with contextual embeddings



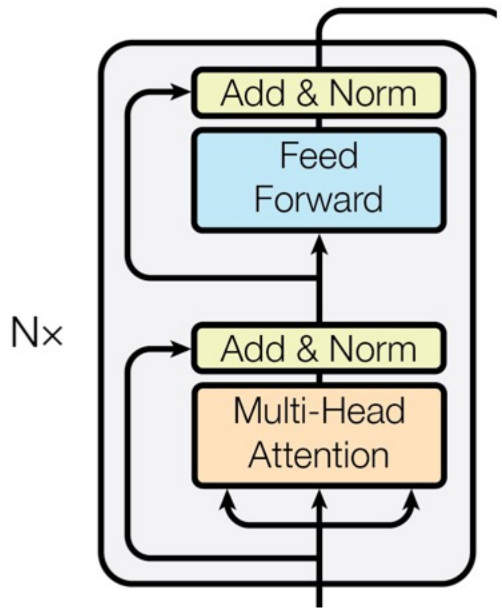
Start with random embeddings

The story so far





Transformer Layer

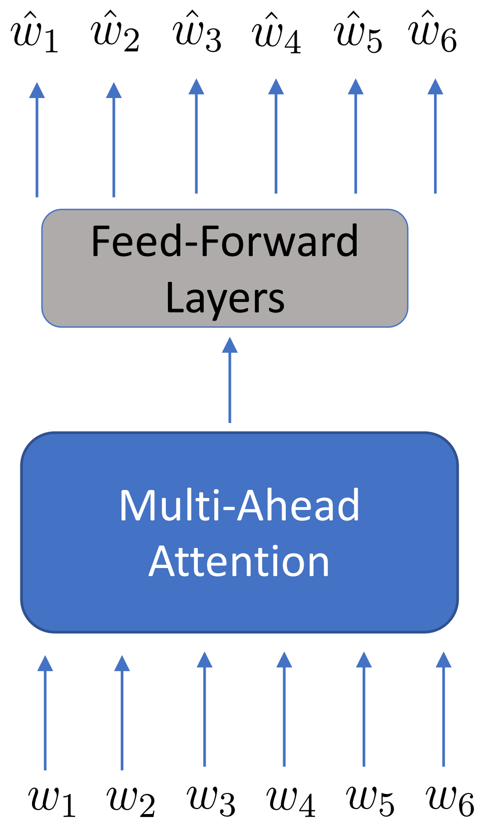


Attention is all you need

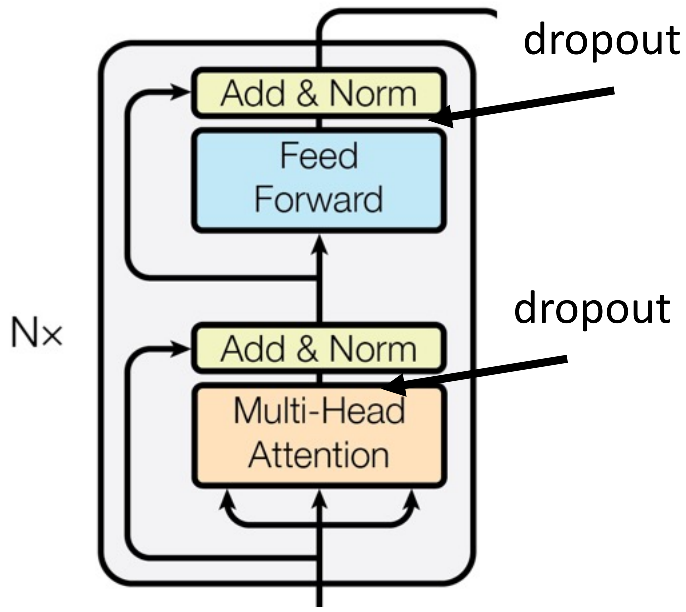
The story so far

End with contextual embeddings

Start with random embeddings



Transformer Layer



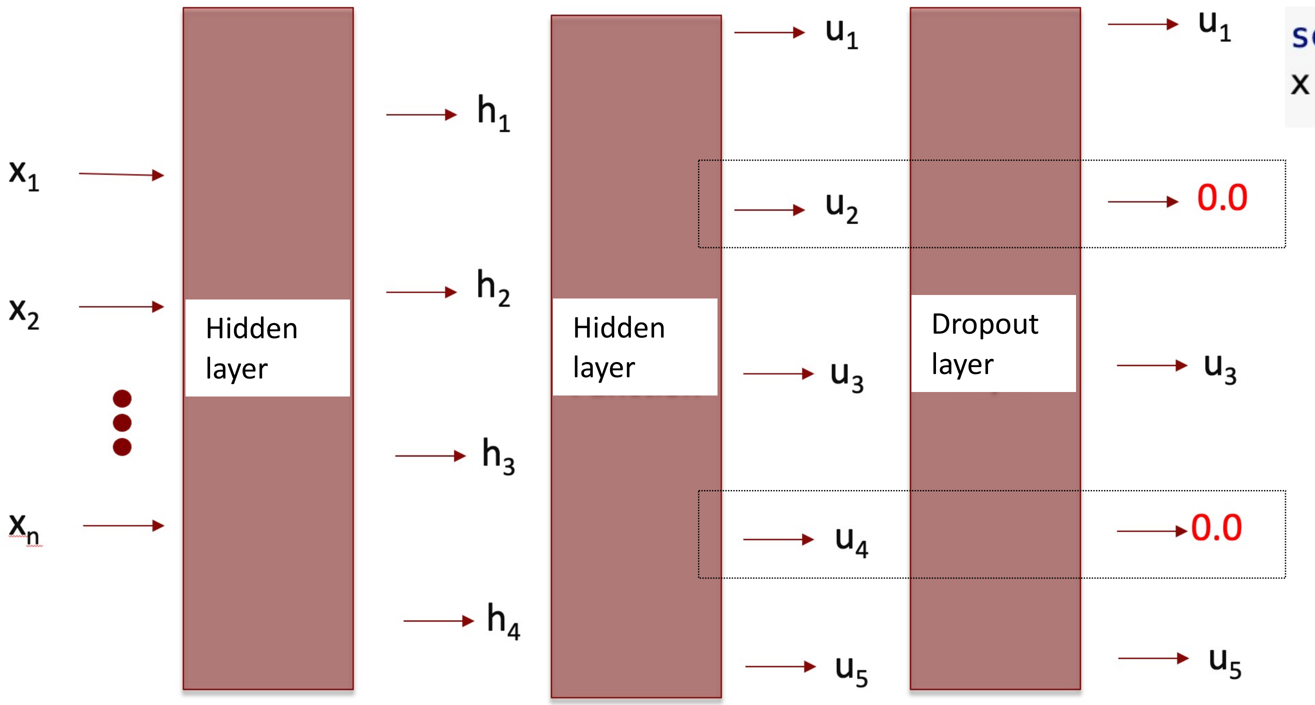
Attention is all you need

Three regularization techniques for Deep Learning

- Dropout
- Residual connections (Add)
- Layer normalization (Norm)

Dropout

<u>In Training, randomly</u> zero out the output from some of the nodes (typically 10% of the nodes) in a hidden layer (implemented as a “dropout layer” in PyTorch)

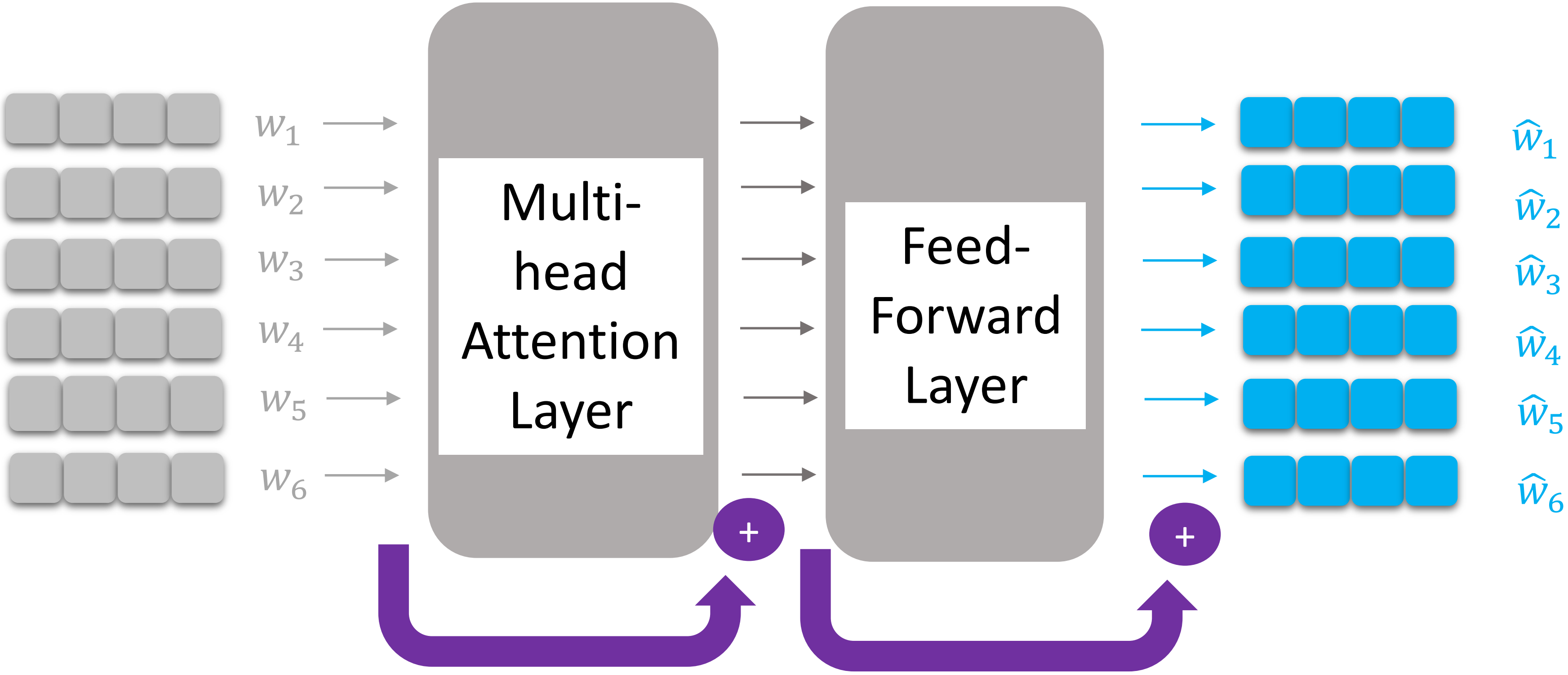



```
self.dropout = nn.Dropout(0.1)
x = self.dropout(x)
```

Residual Connection

```
y = Layer(x) + x
```

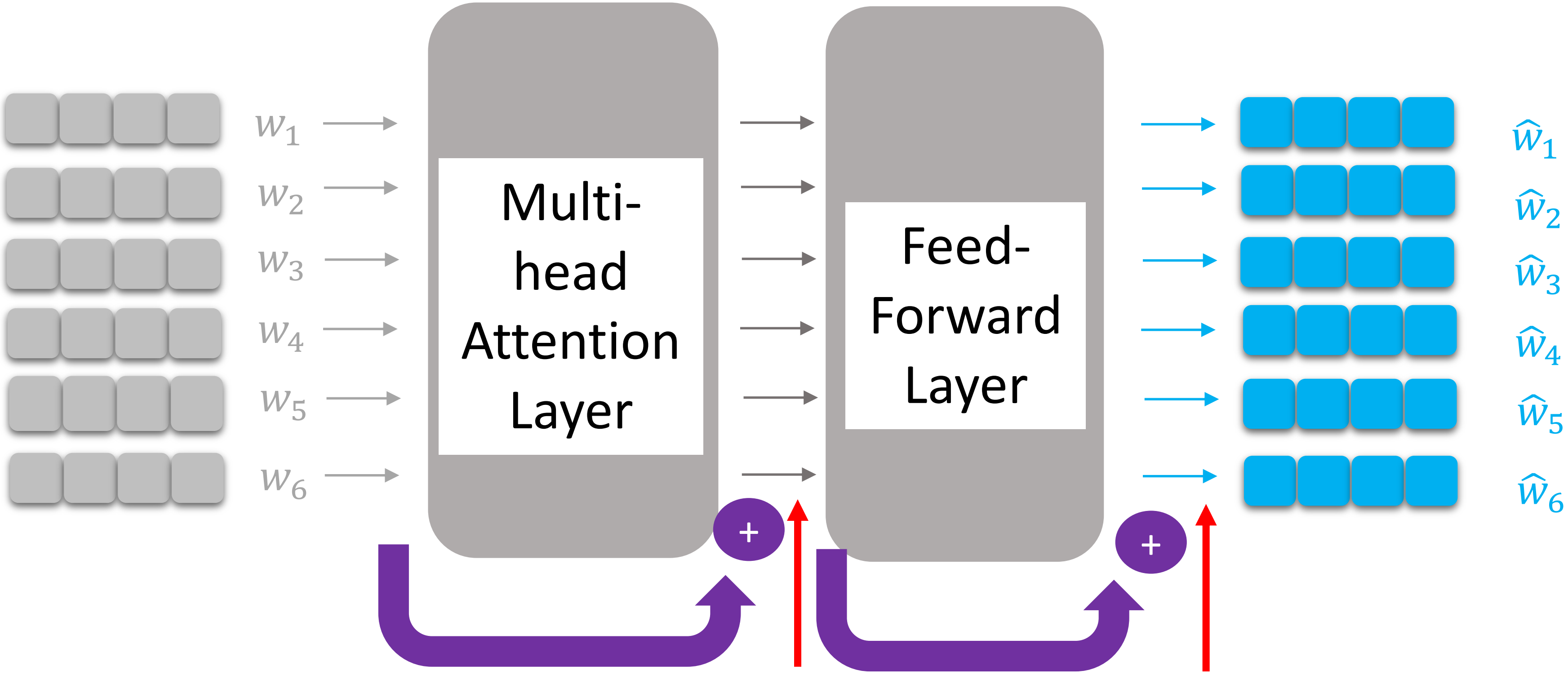
We sum the input embedding to the output embedding of the Attention / Feed-Forward Layers. This helps gradients flow better during backpropagation.



Layer Normalization

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

After the Attention / Feed-Forward Layers, we standardize (i.e., subtract mean and divide by std) each embedding. This ensures that the weights stay small.

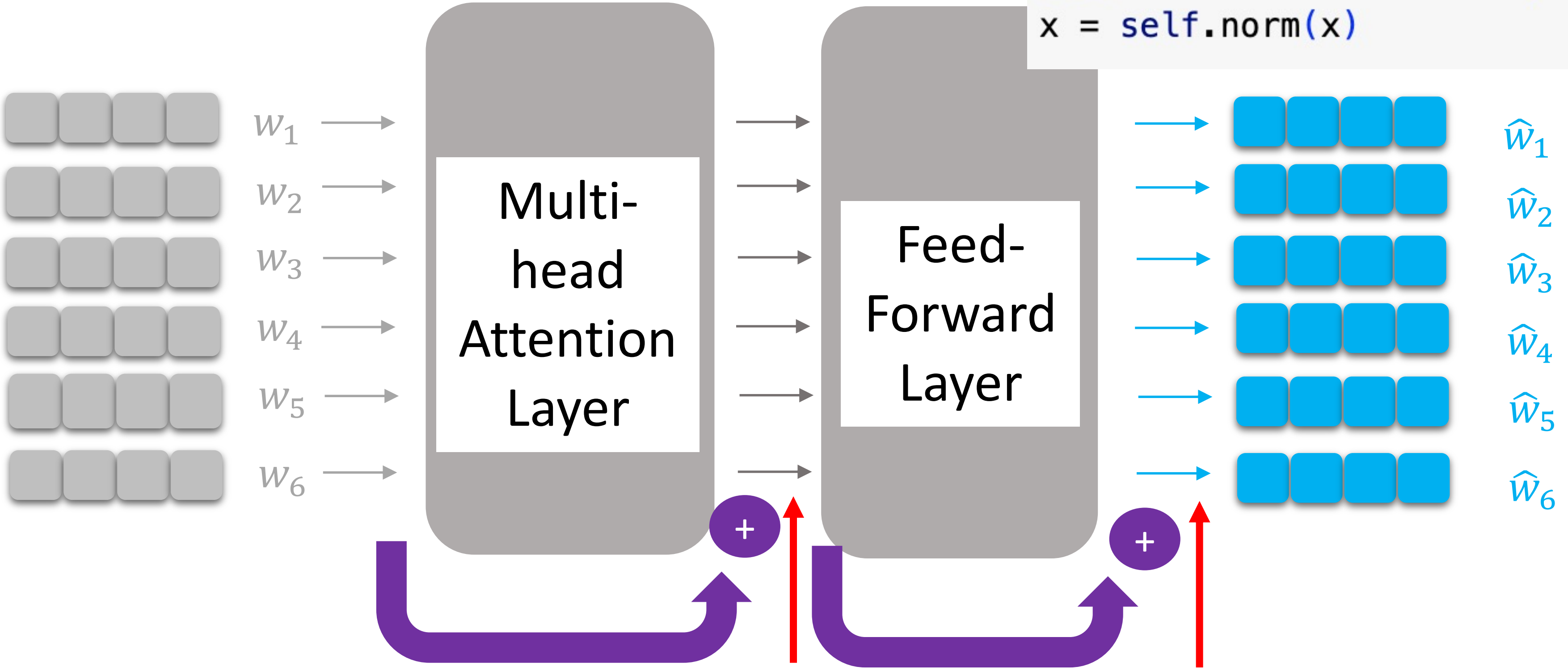


Layer Normalization

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

After the Attention / Feed-Forward Layers, we standardize (i.e., subtract mean and divide by std) each embedding. This ensures that the weights stay small.

```
self.norm = nn.LayerNorm(embedding_dim)
x = self.norm(x)
```

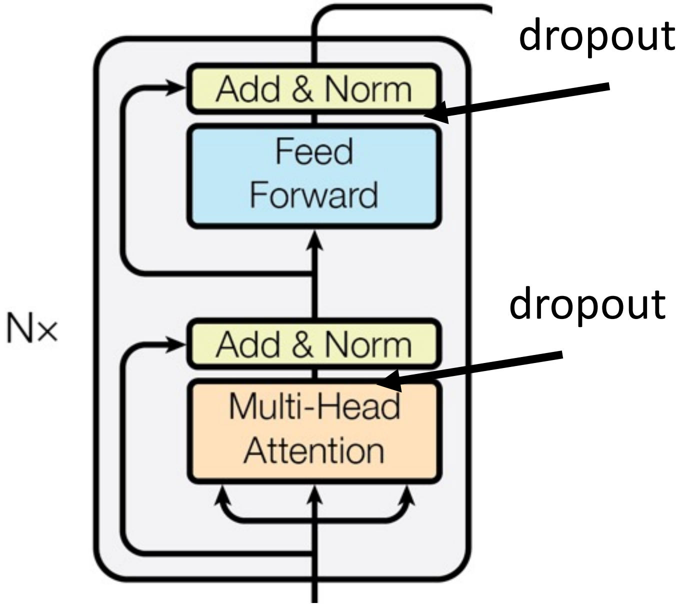


Taken Together: A Transformer Layer

```
y = MultiheadAttention(x) ## Attention Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm

y = FeedForward(x) ## Feed Forward Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm
```

Pseudo code



Attention is all you need

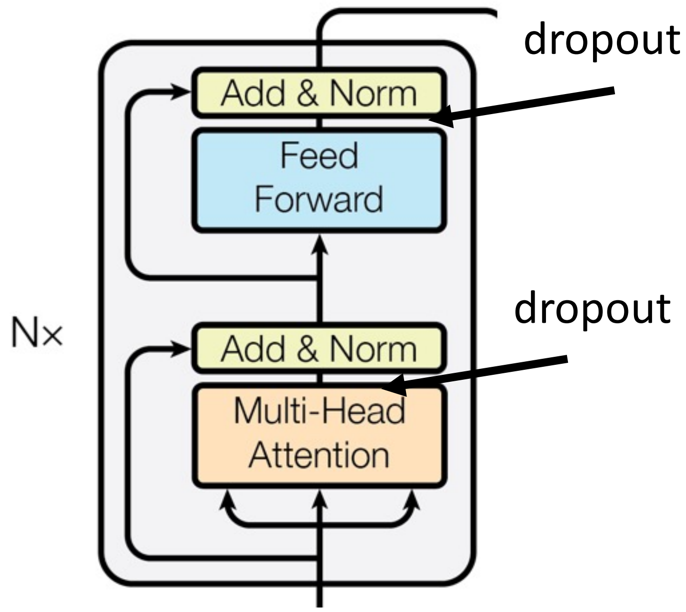
Taken Together: A Transformer Layer

What is dimension of x?

```
y = MultiheadAttention(x) ## Attention Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm

y = FeedForward(x) ## Feed Forward Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm
```

Pseudo code



Attention is all you need

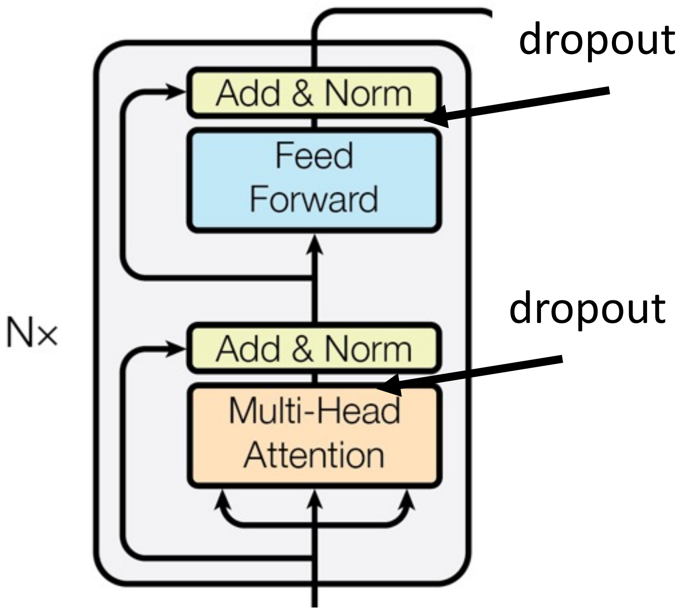
Taken Together: A Transformer Layer

What is dimension of x? [batch_size, seq_Len, embedding_dim]

```
y = MultiheadAttention(x) ## Attention Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm

y = FeedForward(x) ## Feed Forward Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm
```

Pseudo code



Attention is all you need

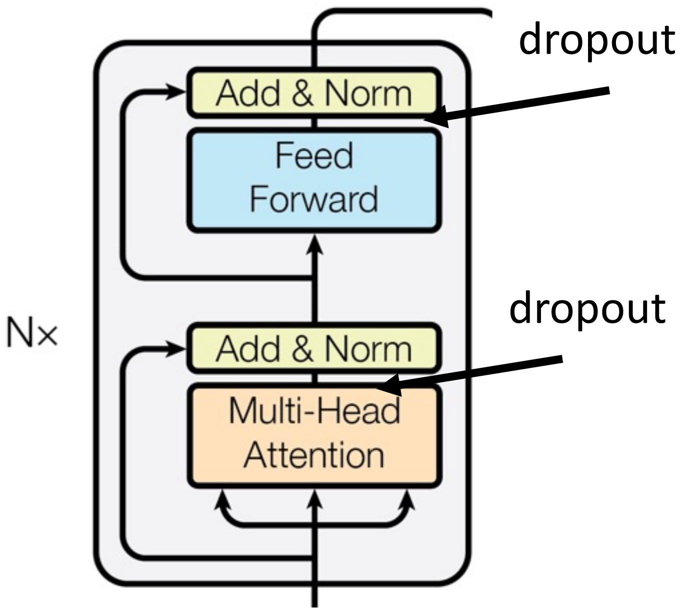
Taken Together: A Transformer Layer

What is dimension of x? [batch_size, seq_Len, embedding_dim]

```
y = MultiheadAttention(x) ## Attention Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm

y = FeedForward(x) ## Feed Forward Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm
```

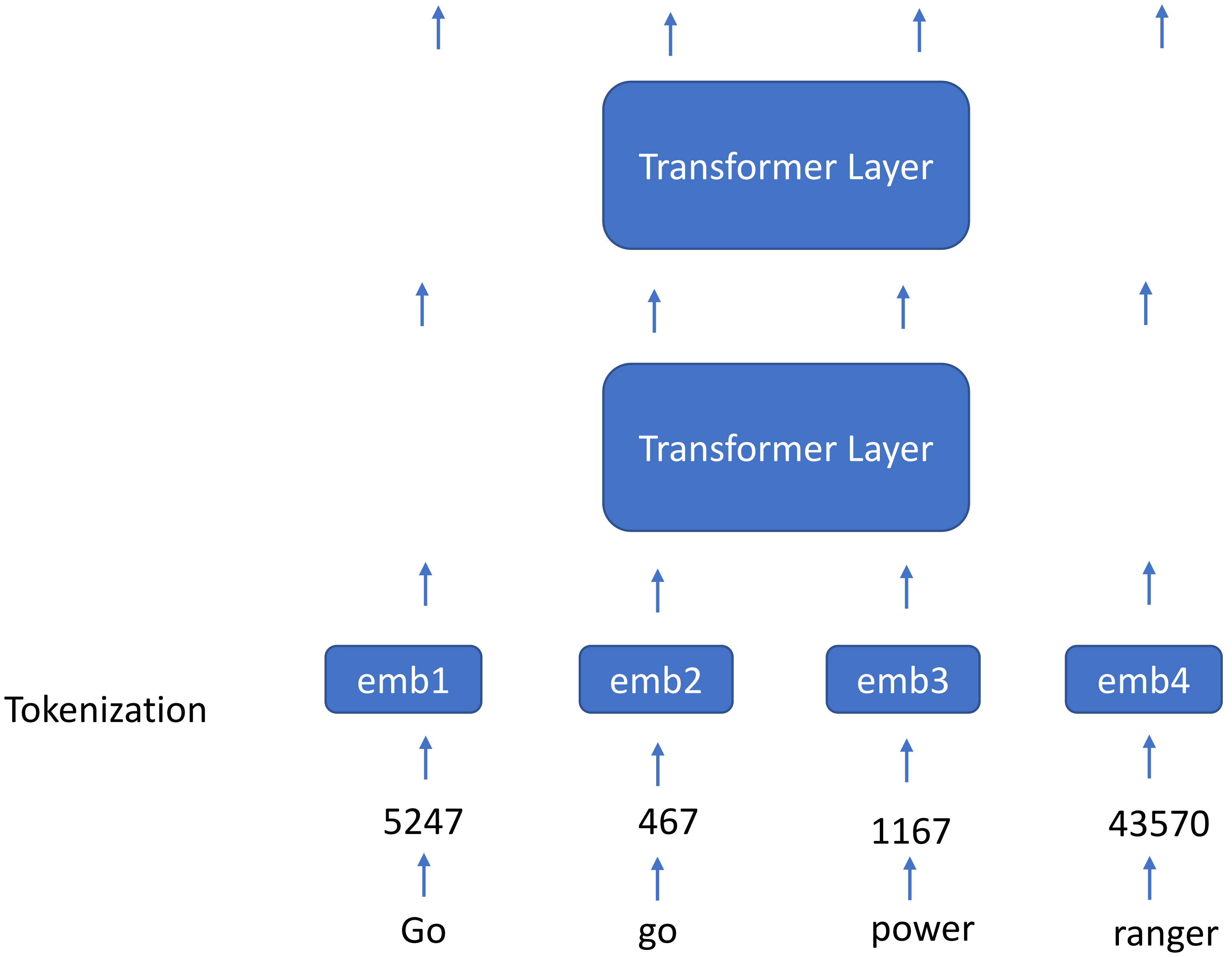
Pseudo code



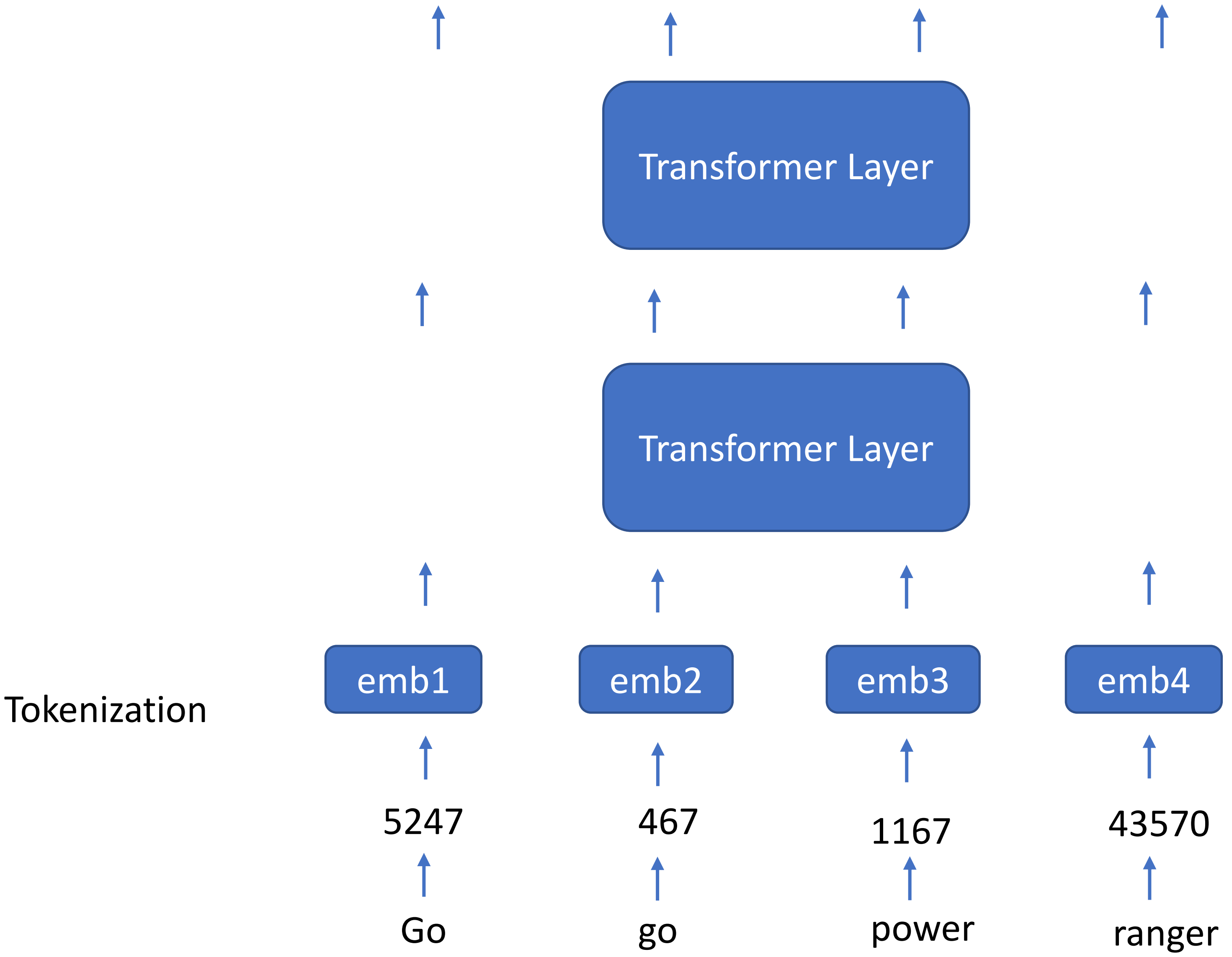
Attention is all you need

We can have multiple Transformer Layers stacked

Does everything look good?

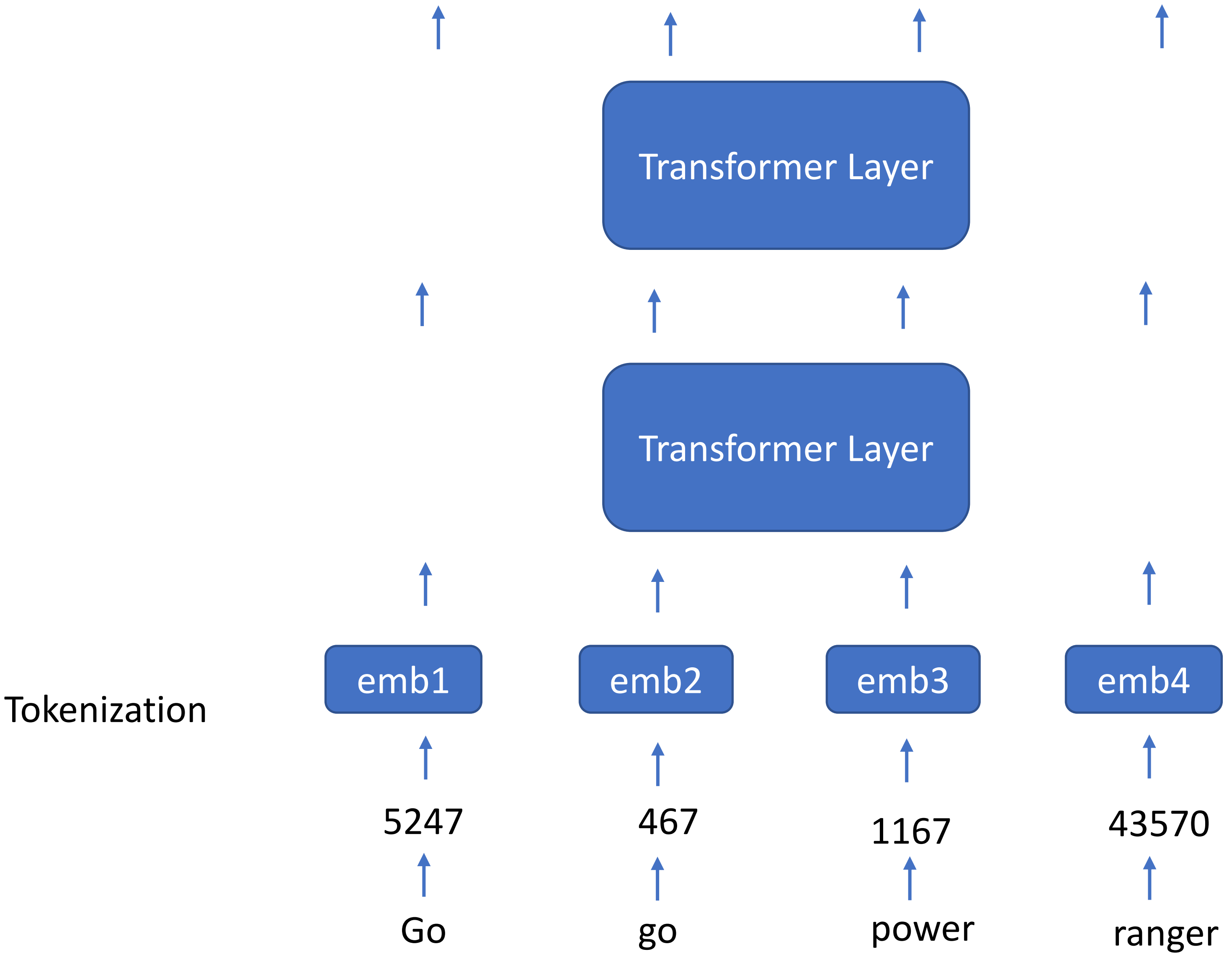


Does everything look good?



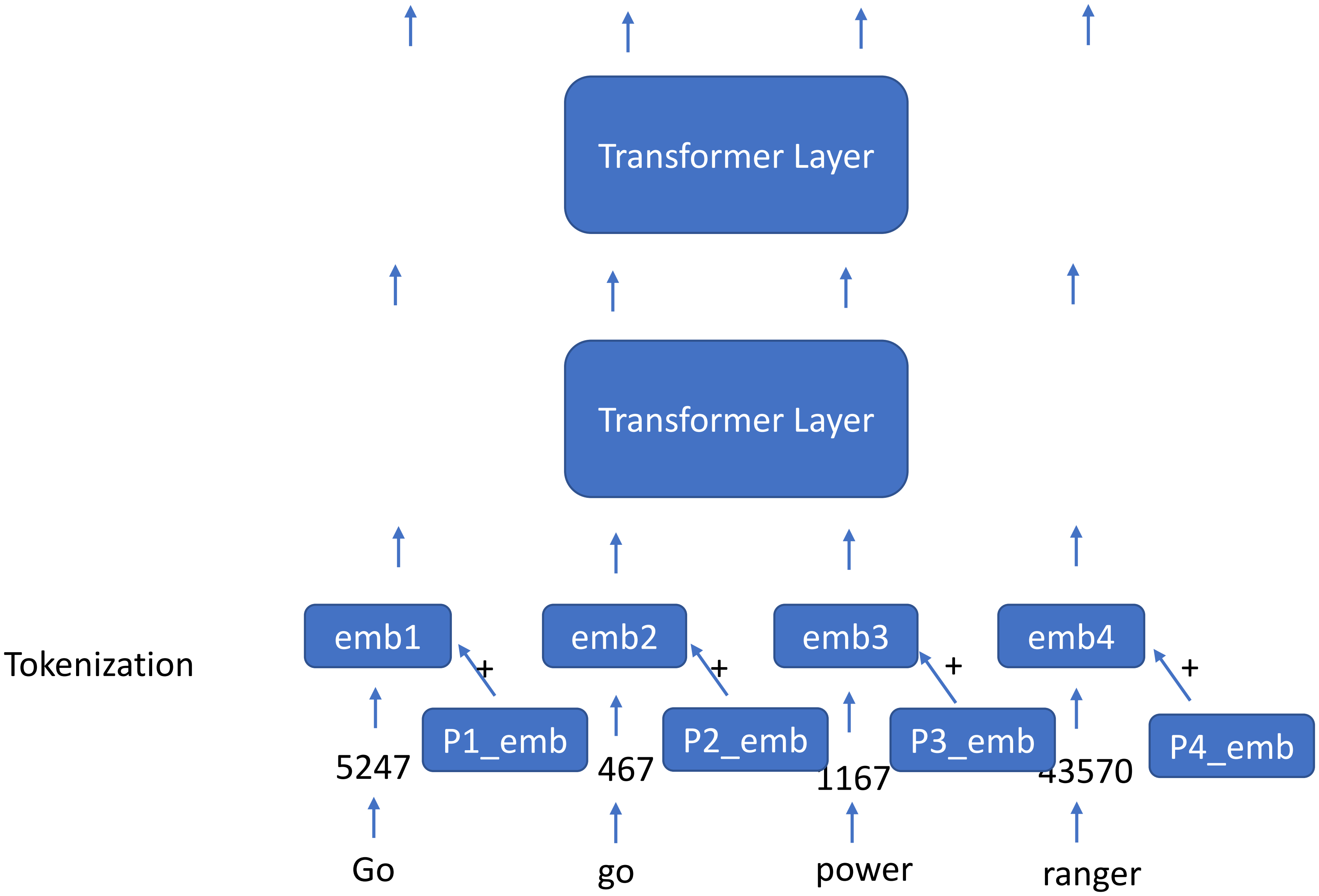
Order information of words are missing

Does everything look good?



Let's directly take positions as inputs and learn a position embedding!

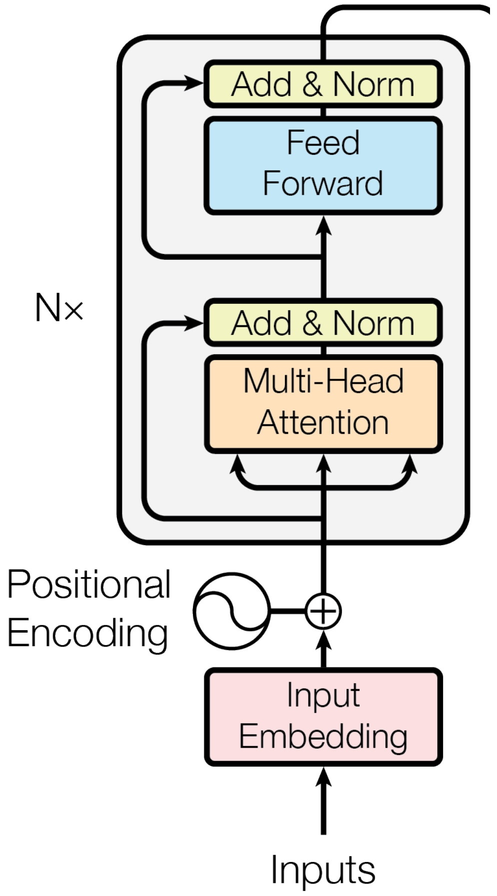
Does everything look good?



Let’s directly take positions as inputs and learn a position embedding!

This is called a Transformer (Encoder)!

What is dimension of x? [batch_size, seq_Len, embedding_dim]



Transformer Layer

```
y = MultiheadAttention(x) ## Attention Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm

y = FeedForward(x) ## Feed Forward Layer
y = Dropout(y) ## Dropout layer
x = x + y ## Add Residule
x = LayerNorm(x) ## Layer Norm
```

Input Layer

```
x = Embedding(inputs)
x = x + Embedding(torch.arange(seq_len))
x = LayerNorm(x)
x = Dropout(x)
```

https://arxiv.org/abs/1706.03762

Coding Exercise

- Text Classification: https://colab.research.google.com/drive/1hWA0Tf4DpeSbhzpb5DP6M_KxfvZOxEyw?usp=sharing

```
x = Transformer(inputs)
x = torch.mean(x, dim=1)
x = nn.Linear(embedding_dim, 2)(x)
```

Classification Head

A better training wrapper

- DataLoader
  - The DataLoader in PyTorch efficiently batches and shuffles data, while also enabling parallel processing and flexible collation, making data loading faster and more scalable for training large models.

- Pytorch_lightning
  - streamlines the training process by automating loops, logging, and device management, enabling cleaner, more efficient model development.

Text Generation

A journey of a thousand miles begins with a single step. ---- Laozi (老子)

How would we train a model to generate the next token/word?

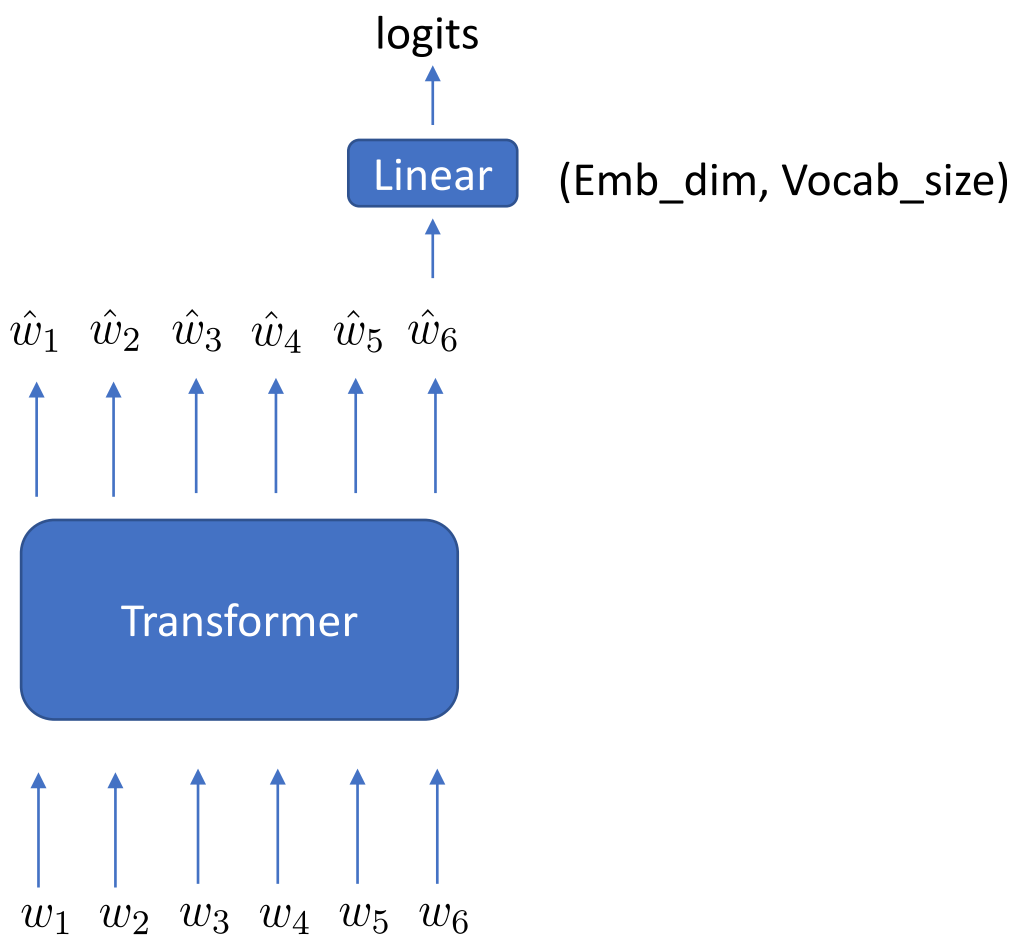
How would we train a model to generate the next token/word?

- Multi-class classification problem!

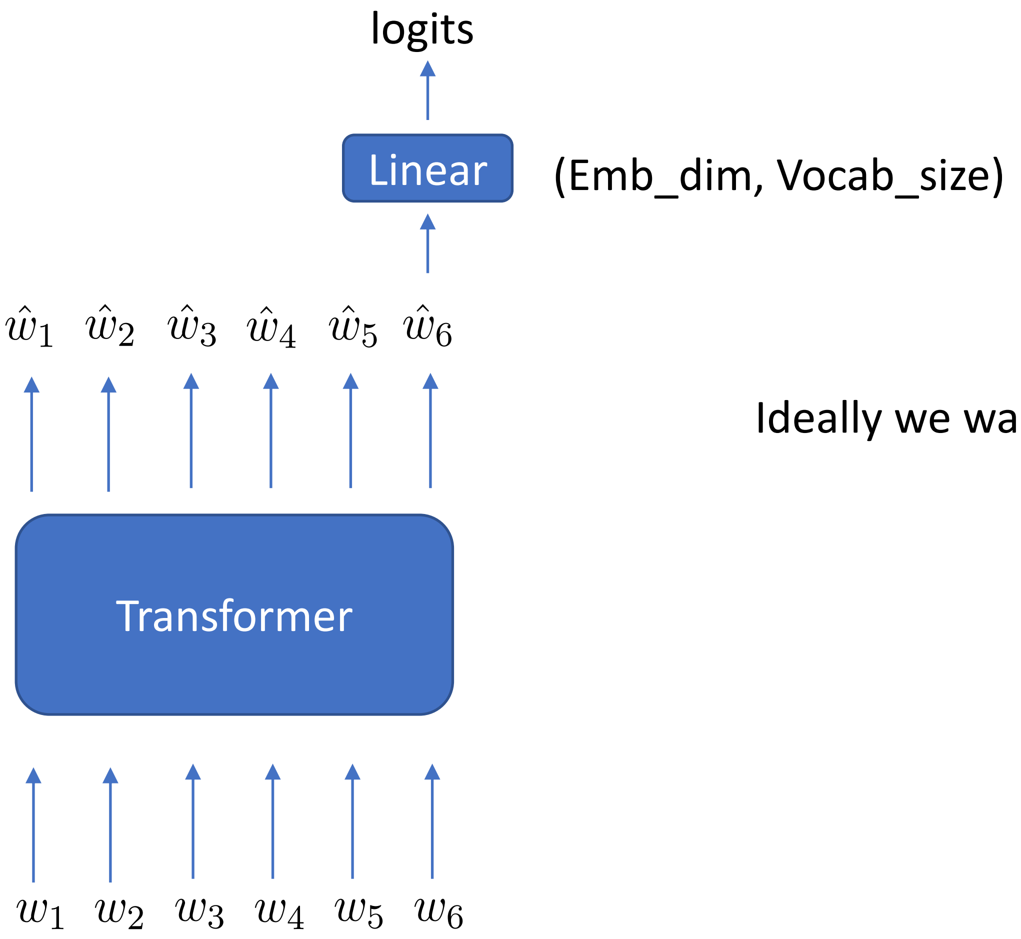
Next Word*	Probability
aardvark	0.0003
…	
rainy	0.3
…	
stormy	0.6
…	
zebra	0.00009

Cross Entropy Loss

How would we train a model to generate the next token/word?



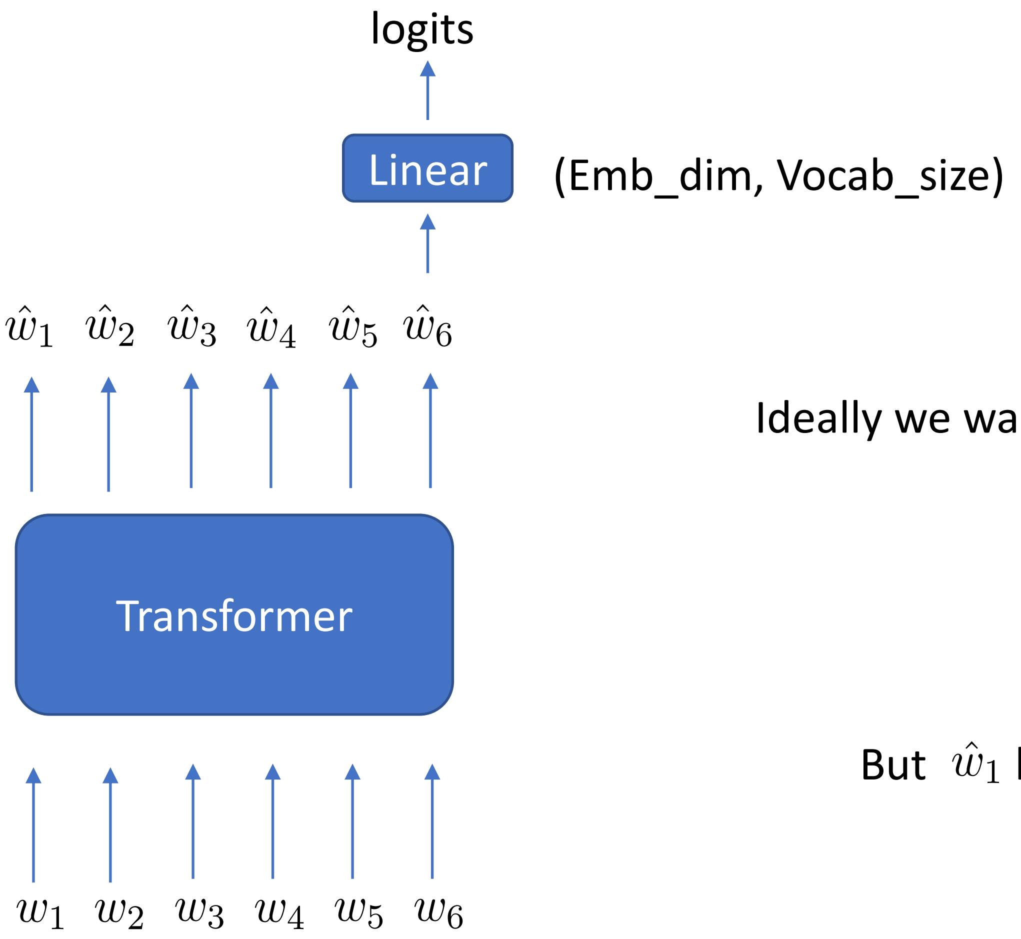
How would we train a model to generate the next token/word?



Ideally we want to use

$$\hat{w}_1 \rightarrow w_2$$
$$\hat{w}_2 \rightarrow w_3$$
$$\hat{w}_3 \rightarrow w_4$$
$$\hat{w}_4 \rightarrow w_5$$

How would we train a model to generate the next token/word?

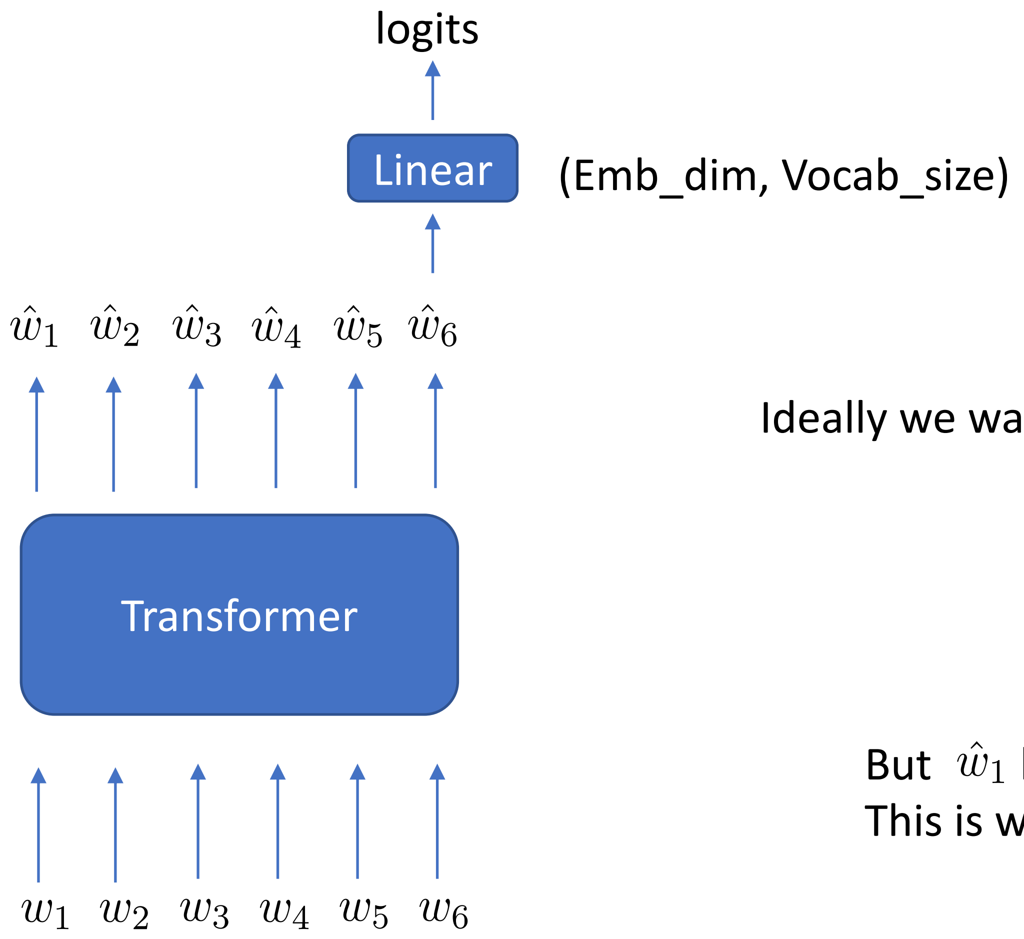


Ideally we want to use

$$\hat{w}_1 \to w_2$$
$$\hat{w}_2 \to w_3$$
$$\hat{w}_3 \to w_4$$
$$\hat{w}_4 \to w_5$$

But $\hat{w}_1$ has information of other words

How would we train a model to generate the next token/word?

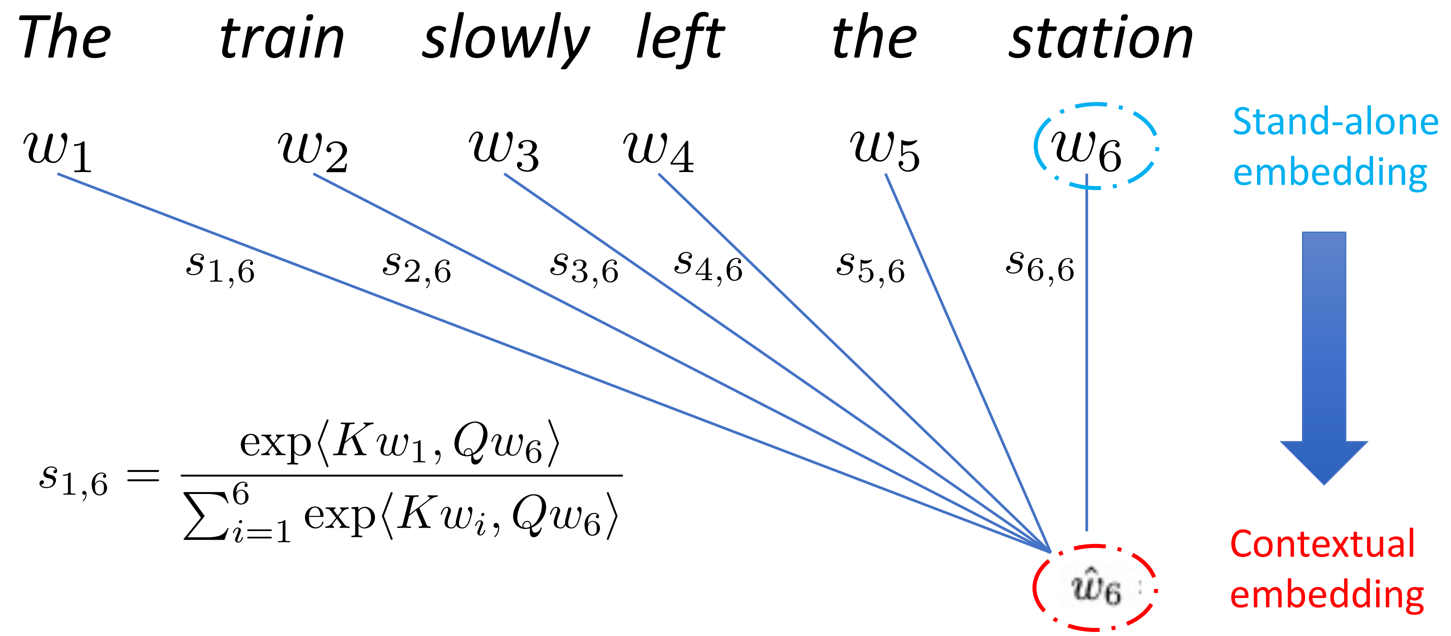


Ideally we want to use

$$\hat{w}_1 \to w_2$$
$$\hat{w}_2 \to w_3$$
$$\hat{w}_3 \to w_4$$
$$\hat{w}_4 \to w_5$$

But $\hat{w}_1$ has information of other words
This is where Causal Mask comes in

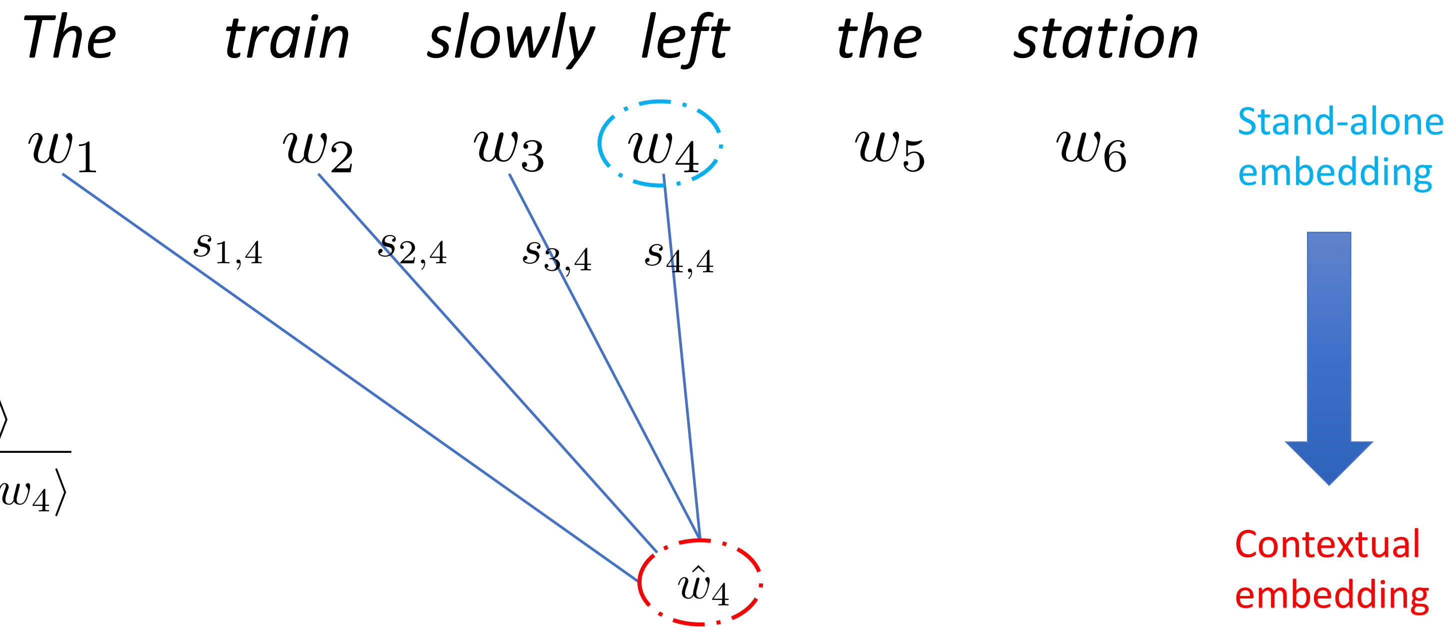
In transformer, the information spread across tokens by Self-Attention



$$s_{1,6} = \frac{\exp\langle K w_1, Q w_6 \rangle}{\sum_{i=1}^{6} \exp\langle K w_i, Q w_6 \rangle}$$

$$\boxed{\hat{w}_6 = s_{1,6} V w_1 + s_{2,6} V w_2 + s_{3,6} V w_3 + s_{4,6} V w_4 + s_{5,6} V w_5 + s_{6,6} V w_6}$$

Causal Mask requires a token cannot use the information of a future token



$$s_{1,4} = \frac{\exp\langle Kw_1, Qw_4 \rangle}{\sum_{i=1}^{4} \exp\langle Kw_i, Qw_4 \rangle}$$

$$\hat{w}_4 = s_{1,4} V w_1 + s_{2,4} V w_4 + s_{3,4} V w_3 + s_{4,4} V w_4$$

Causal Mask requires a token cannot use the information of a future token

$$A_{ij} = \mathrm{mask}_{ij} \cdot \exp \langle Kw_i, Qw_j \rangle$$

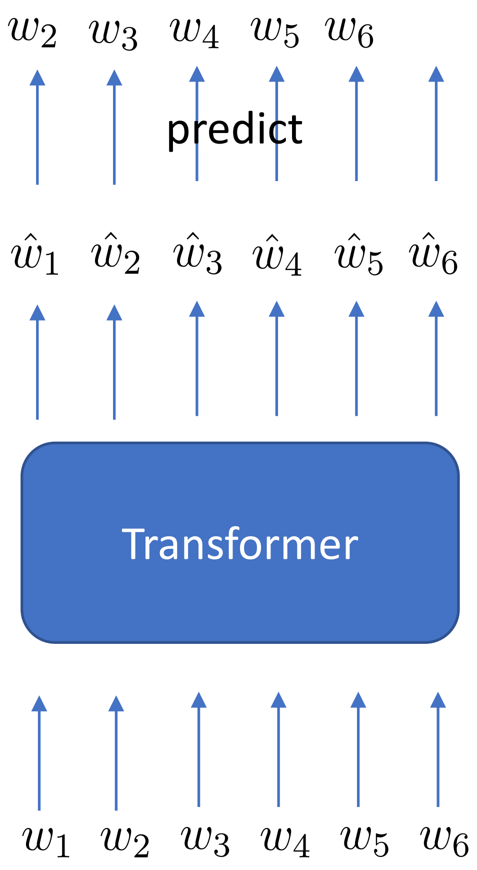
$$s_{i,k} = \frac{A_{ij}}{\sum_{i=1}^{n} A_{ij}}$$

	the	cat	sat	on	the
the	1				
cat	1	1			
sat	1	1	1		
on	1	1	1	1	
the	1	1	1	1	1

mask

Causal Self-Attention, Or Masked Self Attention

With Causal Mask, *training* is much more efficient



All predictions/loss calculations can be done by one pass

Autoregressive Language Model

With this training, essentially we obtain an oracle: $p(x_k|x_{<k})$, an autoregressive model for language

This is what we called GPT!

GPT: Generative Pretrained Transformers

1. Transformer-based text-generation model
2. Pre-trained on **Massive** amount data

Autoregressive Language Model

With this training, essentially we obtain an oracle: $p(x_k|x_{<k})$, an autoregressive model for language

GPT2 Model Architecture

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

Autoregressive Language Model

With this training, essentially we obtain an oracle: $p(x_k|x_{<k})$, an autoregressive model for language

GPT2 Model Architecture

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

GPT2 Dataset

How would you collect high quality data from Internet?

Autoregressive Language Model

With this training, essentially we obtain an oracle: $p(x_k|x_{<k})$, an autoregressive model for language

GPT2 Model Architecture

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

GPT2 Dataset

Links from Reddit Posts that have at least 3 karmas

WebText: 45M links -> 8M docs (40GB)

Autoregressive Language Model

With this training, essentially we obtain an oracle: $p(x_k|x_{<k})$, an autoregressive model for language

GPT2 Model Architecture

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

GPT2 Dataset

Links from Reddit Posts that have at least 3 karmas

WebText: 45M links -> 8M docs (40GB)

Right Data + Right Model

Recap: Transformer

- Transformer
    - Inputs: Input embedding + Position embedding
    - Transformer Layers
        - MultiHeadAttention
        - Regularizations
            - Dropout
            - LayerNorm
            - Residual Connection
        - Fast Forward Neural Networks
- Text Generation
    - Causal Mask for Transformer

Assignment