Transformers

Charles Lovering - charles_lovering@brown.edu

Transformers

1. Attention
2. Multi-head Attention
3. Transformer Block
4. Other Modules
5. Models

Transformers

1. Attention
2. Multi-head Attention
3. Transformer Block
4. Other Modules
5. Models

Transformers

1. Attention
2. Multi-head Attention
3. Transformer Block
4. Other Modules
5. Models

Transformers

1. Attention
2. Multi-head Attention
3. Transformer Block
4. Other Modules
5. Models

Transformers

Attention

- Attention is the primary mechanic used in the transformer network.

Transformers

Attention

- Attention is the primary mechanic used in the transformer network.
- A lot models use it in a range of ways.

Transformers

Attention

“An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.”

Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems.* 2017.

Transformers

Attention: Working Example

the cat danced $\longrightarrow$ die Katze tanzte

Transformers

Attention: Working Example

the cat danced $\longrightarrow$ die Katze $\boxed{\text{tanzte}}$

Assume that we have generated the first two words, and now are trying to generate: “tanzte”.

Transformers

Attention: Working Example

the cat danced → die Katze tanzte

the cat danced

The input words are the keys K and the values V.

Transformers

Attention: Working Example

the cat danced $\longrightarrow$ $\boxed{\text{die Katze}}$ tanzte

The words generated so far are the queries Q.

Transformers

Attention: Working Example

Q: die Katze

```
[
    [-1.0, -2.5],   # q1: die
    [ 4.0,  3.0],   # q2: Katze
]
```

K: the cat danced

```
[
    [-2.0, -4.0],   # k1: the
    [-2.5, -0.5],   # k2: cat
    [ 4.5,  2.5]    # k3: danced
]
```

V: the cat danced

```
[
    [-2.0, -4.0],   # v1: the
    [-2.5, -0.5],   # v2: cat
    [ 4.5,  2.5]    # v3: danced
]
```

Transformers

Attention: Working Example

- Note: the keys and the values correspond 1:1.

Transformers

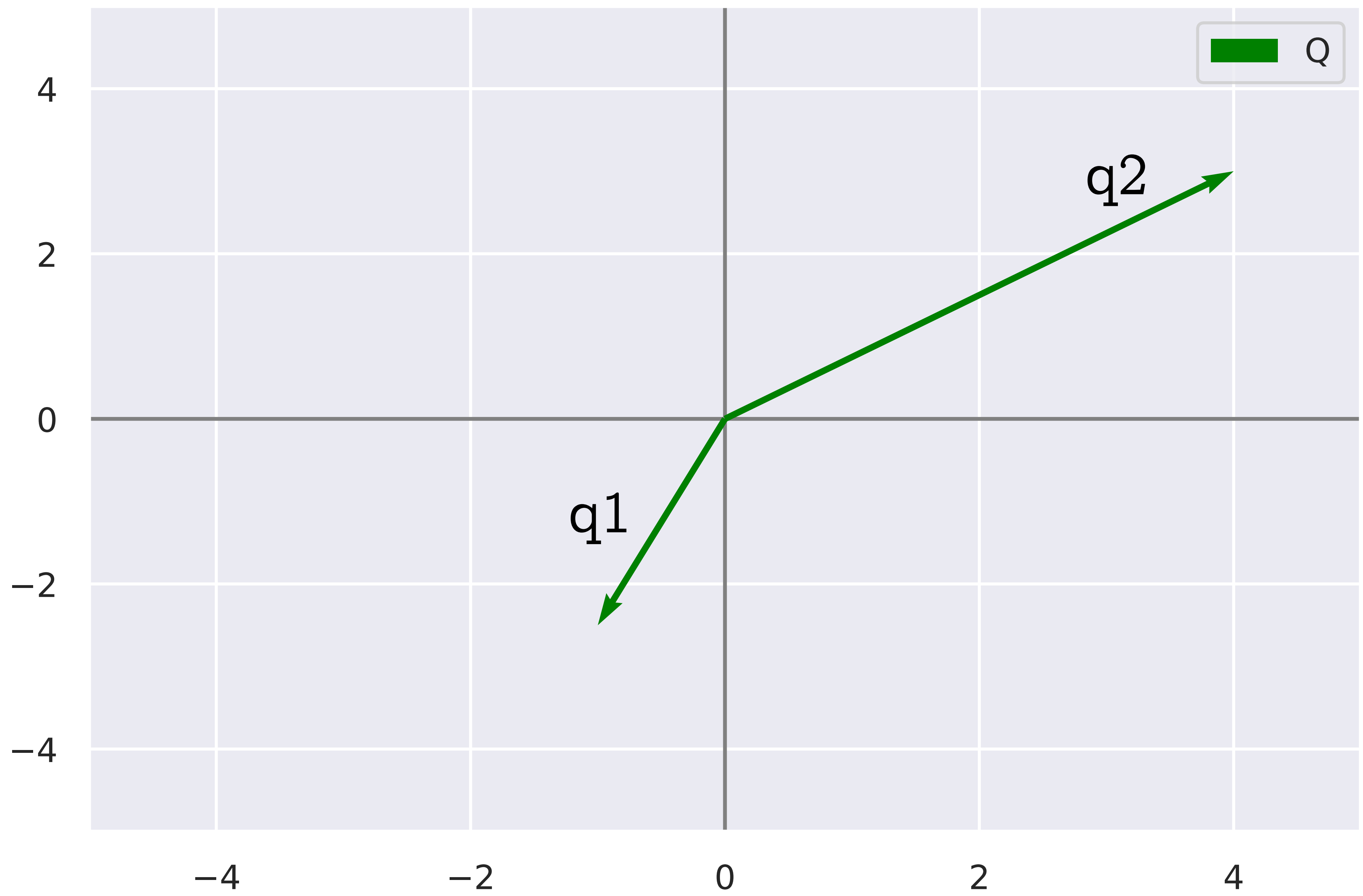
Attention: Working Example

- Note: the keys and the values correspond 1:1.
- They don't have to be the same (although in the transformer models they always are.)

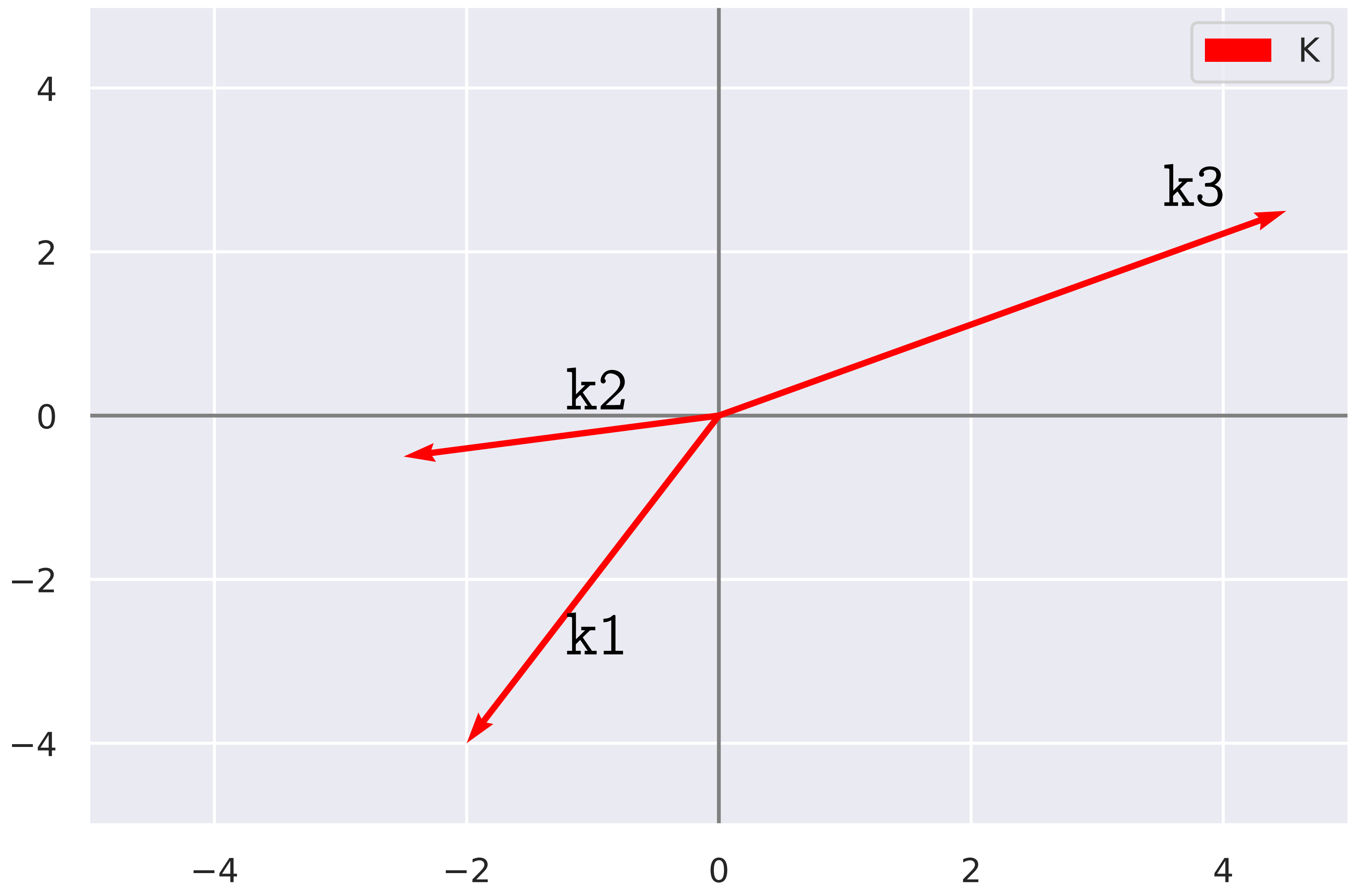
Transformers

Attention: Working Example

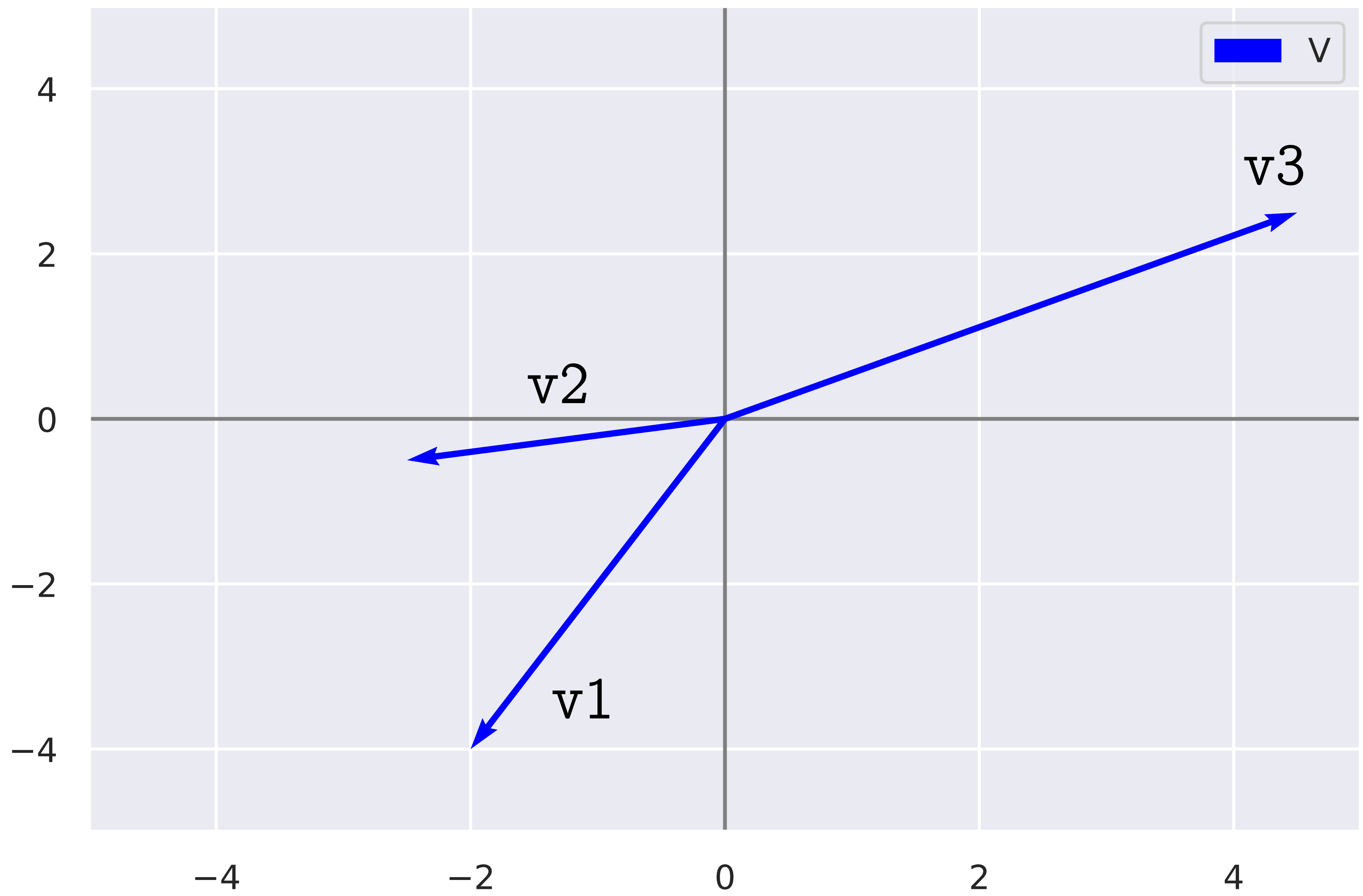
- Note: the keys and the values correspond 1:1.
- They don't have to be the same (although in the transformer models they always are.)
- In fact, the queries, keys, and values can all be the same vectors. This is self-attention.




```
[
    [-1.0, -2.5],   # q1: die
    [4.0,   3.0],   # q2: Katze
]
```



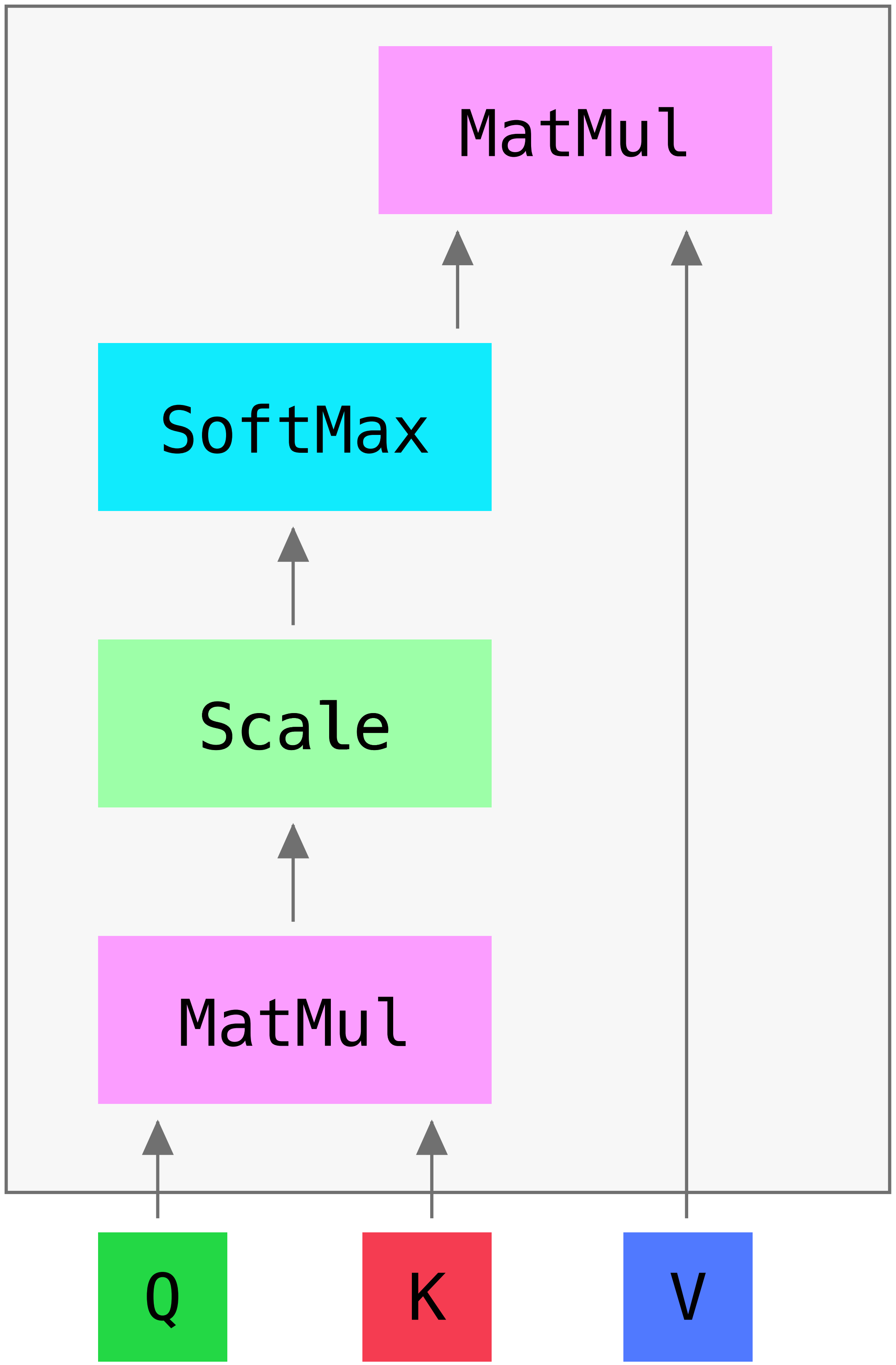

```
[
    [-2, -4],         # k1: the
    [-2.5, -0.5],     # k2: cat
    [4.5 ,2.5]        # k3: danced
]
```




```
[
    [-2, -4],          # v1: the
    [-2.5, -0.5],      # v2: cat
    [4.5 ,2.5]         # v3: danced
]
```

Transformers

Scaled Dot-Product Attention



Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  -> N x D_2

  scores = Q · K.T             # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V          # N x d_2
```

```
Q = [
  [-1.0, -2.5],   # q1: die
  [4.0,   3.0],   # q2: Katze
]
```

```
K.T = [
  [-2, -2.5, 4.5],
  [-4, -0.5, 2.5],
]
```

```
# k1: the
# k2: cat
# k3: danced
```

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  -> N x D_2

  scores = Q · K.T             # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V          # N x d_2
```

```
scores = [
  [
     (-1.0 * -2.0 + -2.5 * -4.0),
     (-1.0 * -2.5 + -2.5 * -0.5),
     (-1.0 *  4.5 + -2.5 *  2.5),
  ],
  [
     ( 4.0 * -2.0 +  3.0 * -4.0),
     ( 4.0 * -2.5 +  3.0 * -0.5),
     ( 4.0 *  4.5 +  3.0 *  2.5),
  ],
]
```

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  —> N x D_2

  scores = Q · K.T              # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V            # N x d_2
```

```
scores = [
  [  12.00,   3.75, -10.75 ],
  [ -20.00, -11.50,  25.50 ]]
```

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  —> N x D_2

  scores = Q · K.T            # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V          # N x d_2
```

```
scores = [
  [  12.00,   3.75, -10.75 ],
  [ -20.00, -11.50,  25.50 ]]
```

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  -> N x D_2

  scores = Q · K.T             # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V          # N x d_2
```

```
scores = [
  [  12.00,   3.75, -10.75 ],
  [ -20.00, -11.50,  25.50 ]]
```

Each index in this NxM matrix represents the *compatibility* between query n and key m.

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  -> N x D_2

  scores = Q · K.T             # N x M
  scaled = scores / sqrt(D_1)  # N x M
  alpha  = softmax(scores)     # N x M
  out    = alpha · V           # N x d_2
```

```
scores = [
  [  12.00,   3.75, -10.75 ],
  [ -20.00, -11.50,  25.50 ]]
```

Each index in this NxM matrix represents the *compatibility* between query n and key m.

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  —> N x D_2

  scores = Q · K.T             # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V           # N x d_2
```

```
scores = [
  [  12.00,   3.75, -10.75 ],
  [ -20.00, -11.50,  25.50 ]]
```

Each index in this NxM matrix represents the *compatibility* between query n and key m.

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  —> N x D_2

  scores = Q · K.T              # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V            # N x d_2
```

```
scores = [
  [  12.00,   3.75, -10.75 ],
  [ -20.00, -11.50,  25.50 ]]

scaled = [
  [ 4.0000,  1.2500, -3.5833],
  [-6.6667, -3.8333,  8.5000]]
```

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  -> N x D_2

  scores = Q · K.T              # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V            # N x d_2
```

```
scaled = [
  [ 4.0000,  1.2500, -3.5833],
  [-6.6667, -3.8333,  8.5000]]
```

The motivation for scaling the product is that the dot product gets larger as the dimensionality gets larger: the variance of the dot-product of 0-1 random variables is the length of the vector.

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  -> N x D_2

  scores = Q · K.T              # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)     # N x M
  out    = alpha · V            # N x d_2
```

```
scaled = [
  [ 4.0000,  1.2500, -3.5833],
  [-6.6667, -3.8333,  8.5000]]
```

When the values are large, the gradients of the softmax will be small (which can hurt learning).

Transformers

Softmax

$$\frac{e^{x_i}}{\sum_j e^{x_j}}$$

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  -> N x D_2

  scores = Q · K.T              # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V          # N x d_2
```

```
scaled = [
  [ 4.0000,  1.2500, -3.5833],
  [-6.6667, -3.8333,  8.5000]]

alpha = [
  [0.94, 0.06, 0.00],
  [0.00, 0.00, 0.99]
]
```

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  -> N x D_2

  scores = Q · K.T               # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)    # N x M
  out    = alpha · V          # N x d_2
```

```
scaled = [
  [ 4.0000,  1.2500, -3.5833],
  [-6.6667, -3.8333,  8.5000]]
```

```
alpha = [
  [0.94, 0.06, 0.00],
  [0.00, 0.00, 0.99]
]
```

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  # Q : N x D_1
  # K : M x D_1
  # V : M x D_2
  #  -> N x D_2

  scores = Q · K.T              # N x M
  scaled = scores / sqrt(D_1) # N x M
  alpha  = softmax(scores)     # N x M
  out    = alpha · V           # N x d_2
```

```
alpha = [
  [0.94, 0.06, 0.00],
  [0.00, 0.00, 0.99]
]

V = [
    [-2.0, -4.0],    # v1: the
    [-2.5, -0.5],    # v2: cat
    [ 4.5,  2.5]     # v3: danced
]
```

```
out = [
  [
    (0.94 * -2.0 + 0.06 * -2.5 + 0.00 * 4.5),
    (0.00 * -4.0 + 0.00 * -0.5 + 0.99 * 2.5),
  ],
  [
    (0.94 * -2.0 + 0.06 * -2.5 + 0.00 * 4.5),
    (0.00 * -4.0 + 0.00 * -0.5 + 0.99 * 2.5),
  ]
]
```

Transformers

Attention: Working Example

```
def scaled-dot-product-attention(Q,K,V):
  …
  out    = alpha · V            # N x d_2
```

```
out = [
  [-2.02,  -3.78],
  [ 4.49 ,  2.49]]
```

Transformers

Attention: Working Example

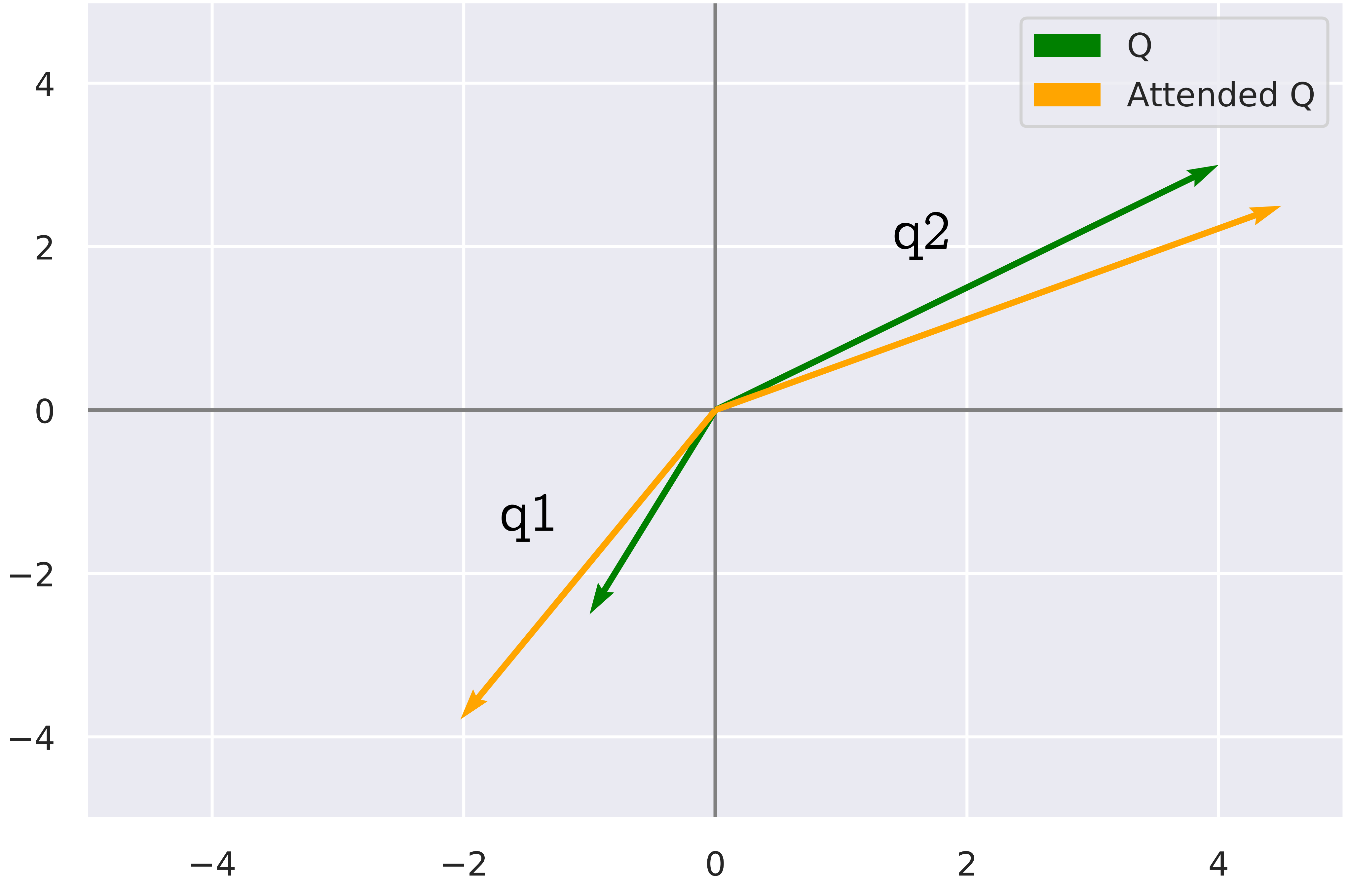
```
def scaled-dot-product-attention(Q,K,V):
  …
  out     = alpha · V            # N x d_2
```

```
out = [
  [-2.02,  -3.78],
  [ 4.49 ,  2.49]]
```

- Thus, each query is mapped to a linear combination of the values.
- Note: the weight of each value depends on the compatibility between the corresponding key and query.

Transformers

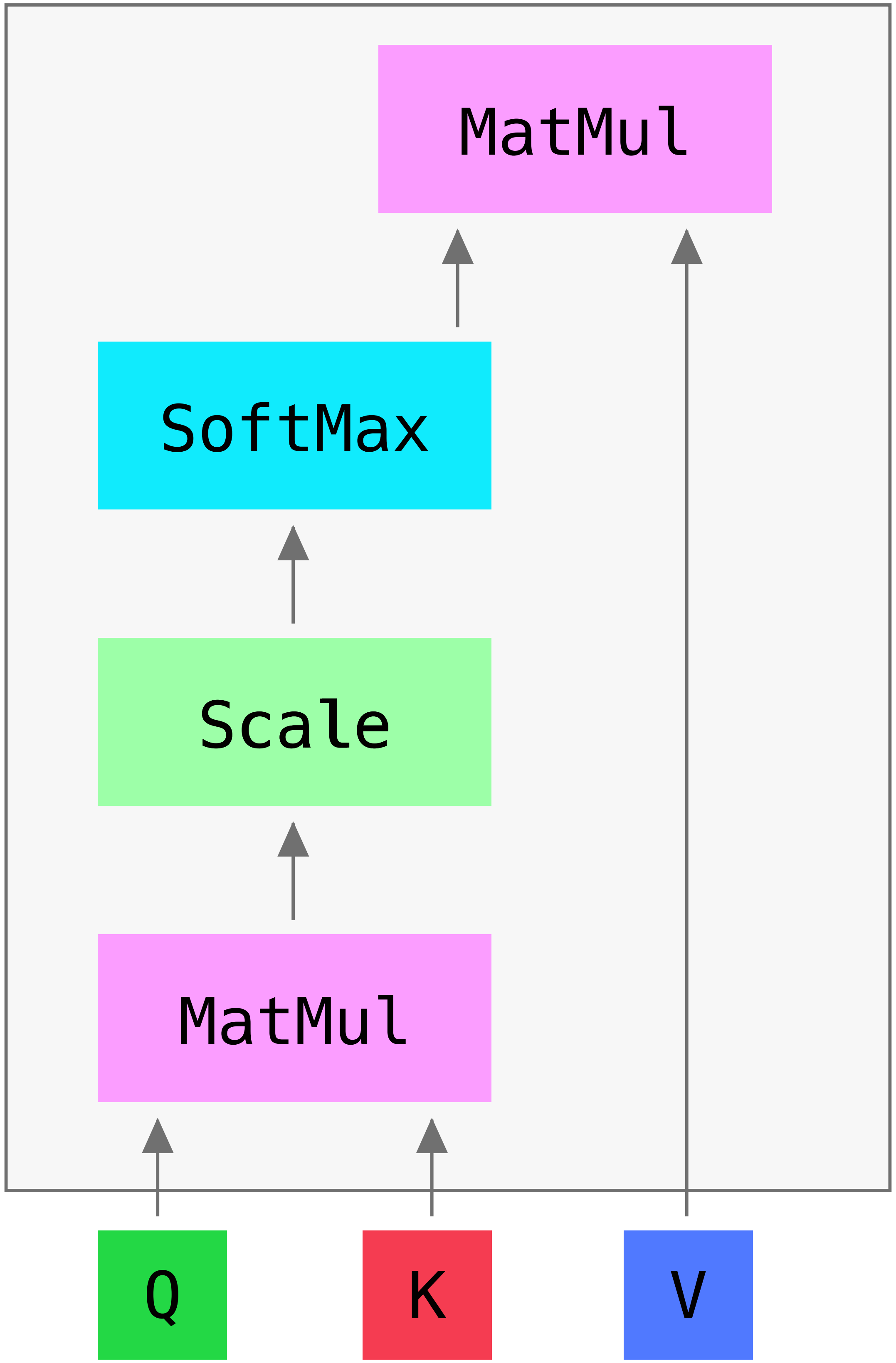
Attention: Working Example



Transformers

Attention

"An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key."



Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems.* 2017.

Transformers

1. Attention
2. Multi-head Attention
3. Transformer Block
4. Other Modules
5. Models

Transformers

Multi-head Attention Intuition

- No parameters in scaled dot-product attention.

Transformers

Multi-head Attention Intuition

- No parameters in scaled dot-product attention.
- Thus, to influence how we attend the vectors, downstream functions have to be updated.

Transformers

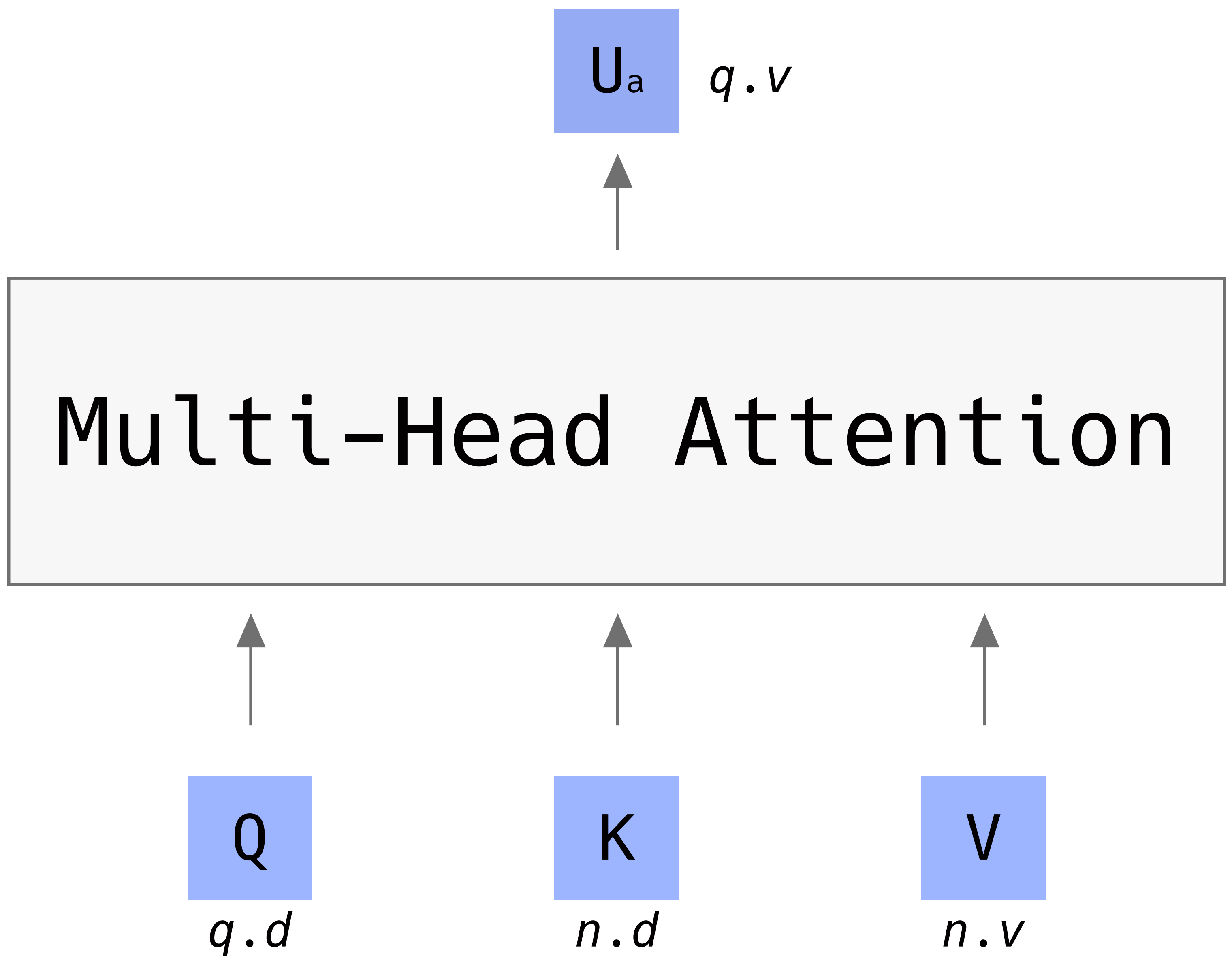
Multi-head Attention Intuition

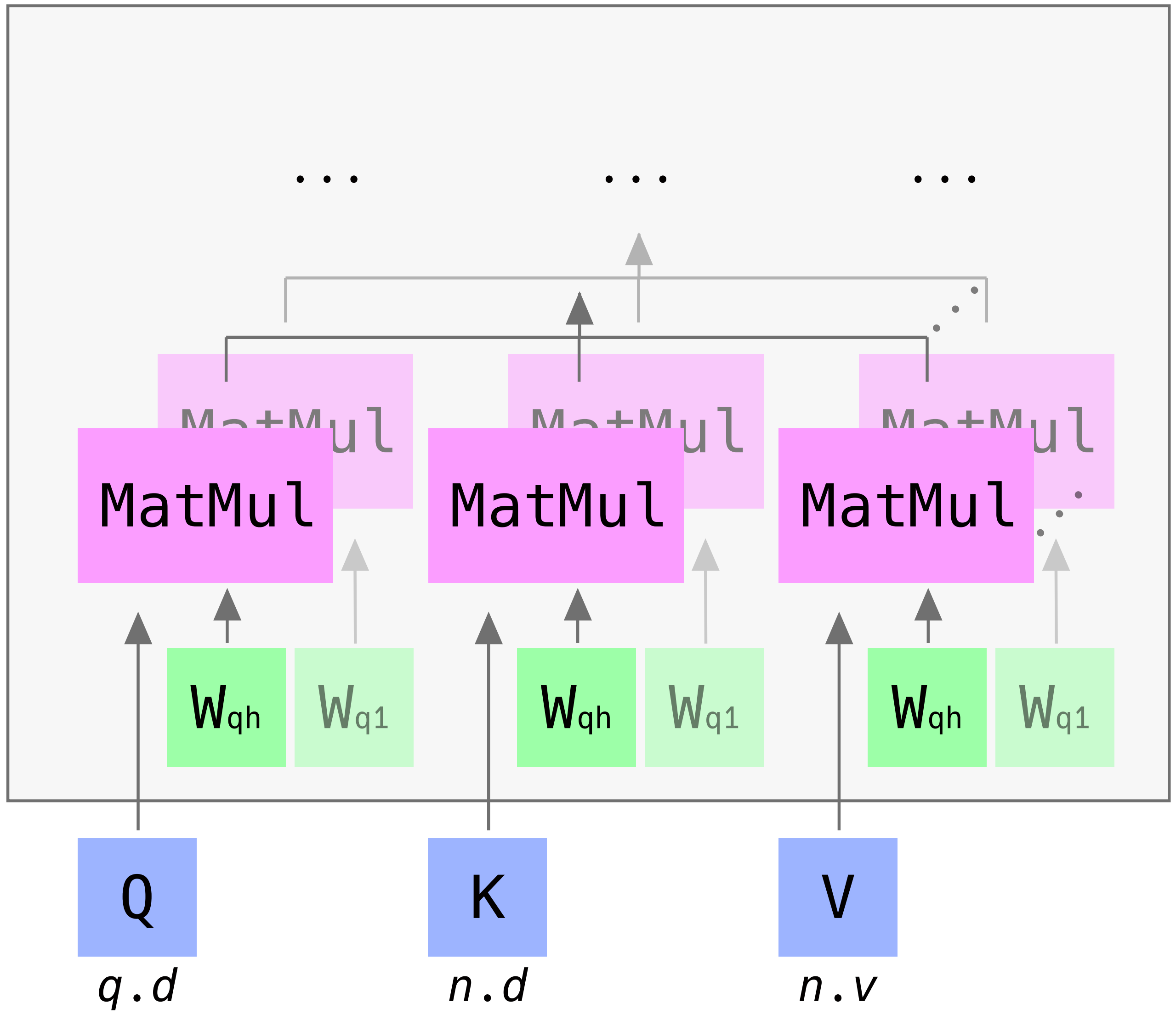
- Dot-product isn't flexible; it's difficult to attend to different aspects of a representation.

Transformers

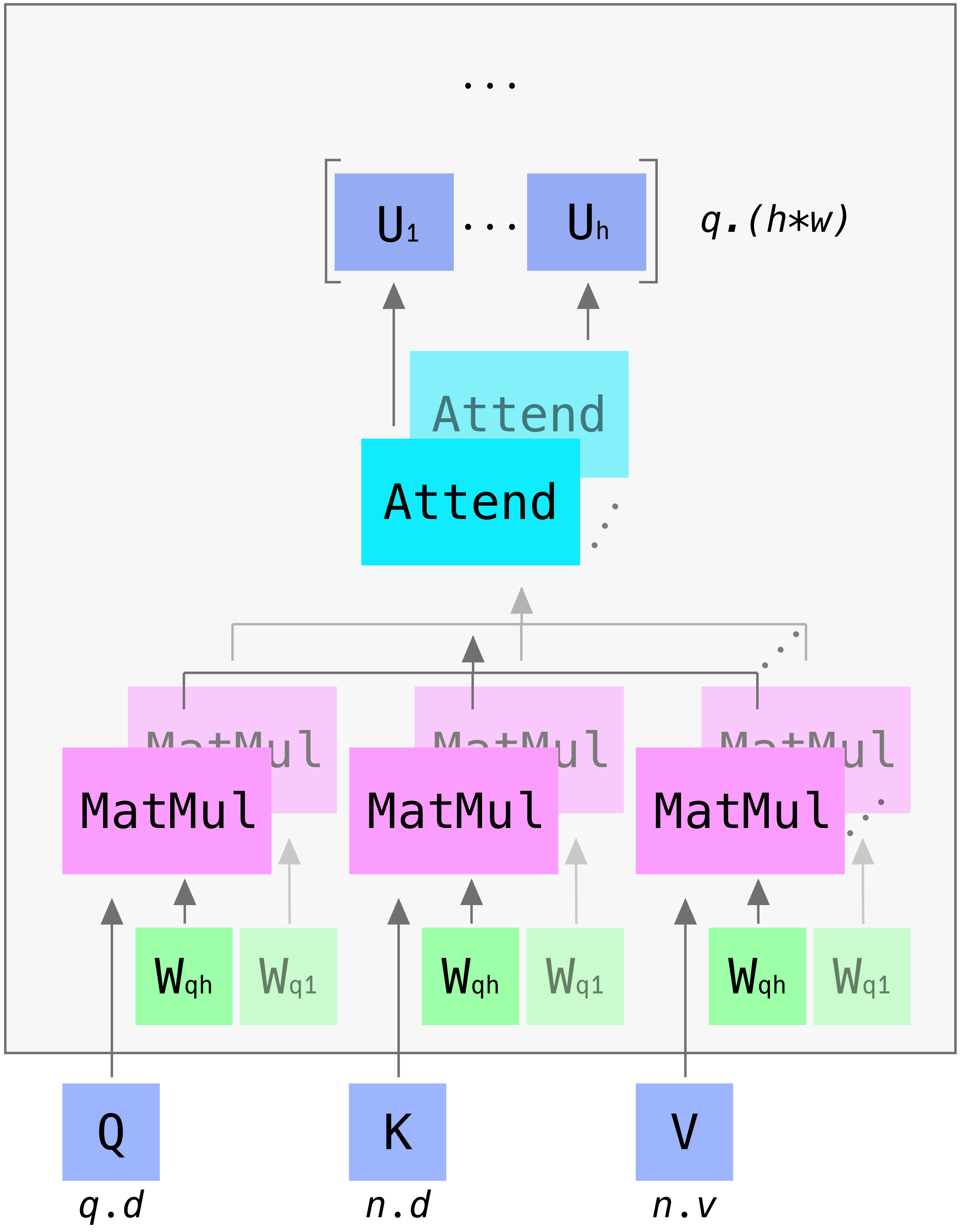
Multi-head Attention Intuition

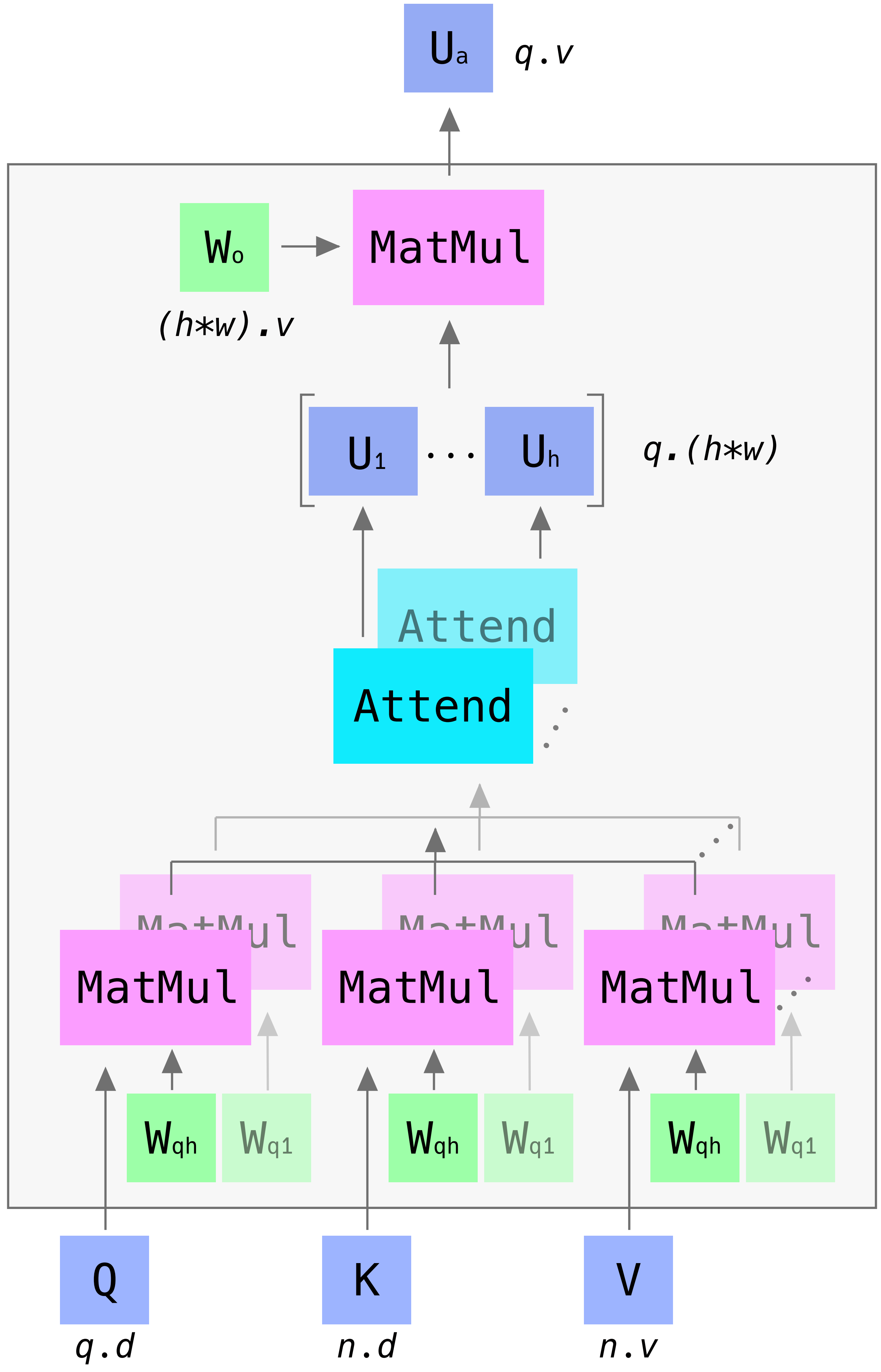
- Dot-product isn't flexible; it's difficult to attend to different aspects of a representation.
- How can we get the model to attend differently based upon the context?


U_a
$q.v$
Multi-Head Attention
Q
K
V
$q.d$
$n.d$
$n.v$



...
...
...
MatMul
MatMul
MatMul
MatMul
MatMul
MatMul
W_{qh}
W_{q1}
W_{qh}
W_{q1}
W_{qh}
W_{q1}
Q
K
V
q.d
n.d
n.v


...
U1 ... Uh
q.(h*w)
Attend
Attend
MatMul
MatMul
MatMul
MatMul
MatMul
MatMul
Wqh
Wq1
Wqh
Wq1
Wqh
Wq1
Q
K
V
q.d
n.d
n.v



Ua
q.v
Wo
MatMul
(h*w).v
U1
Uh
q.(h*w)
Attend
Attend
MatMul
MatMul
MatMul
MatMul
MatMul
MatMul
Wqh
Wq1
Wqh
Wq1
Wqh
Wq1
Q
K
V
q.d
n.d
n.v

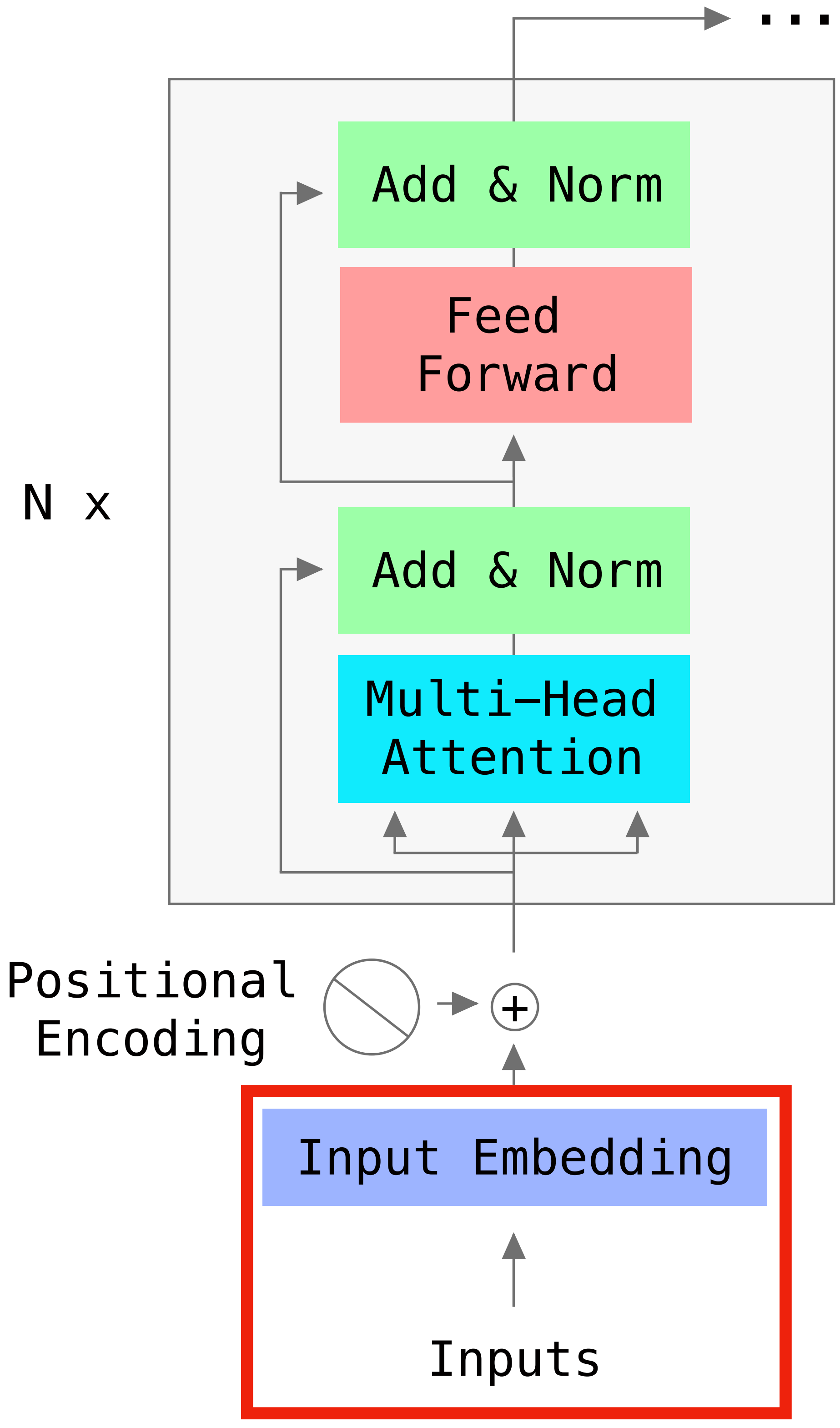
Transformers

1. Attention
2. Multi-head Attention
3. Transformer Block
4. Other Modules
5. Models

Transformers

Transformer Block

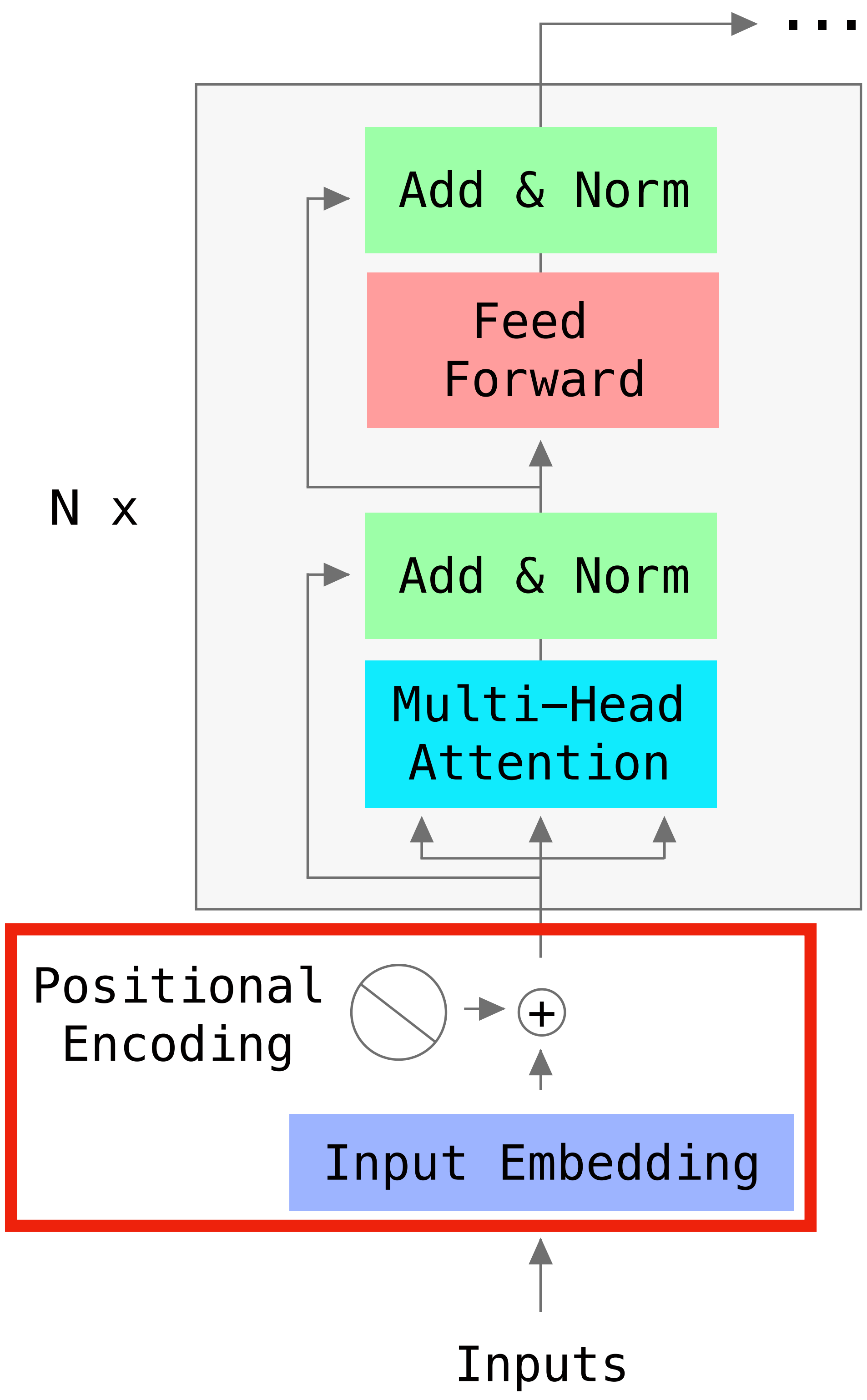
1. Input embedding.



Transformers

Transformer Block

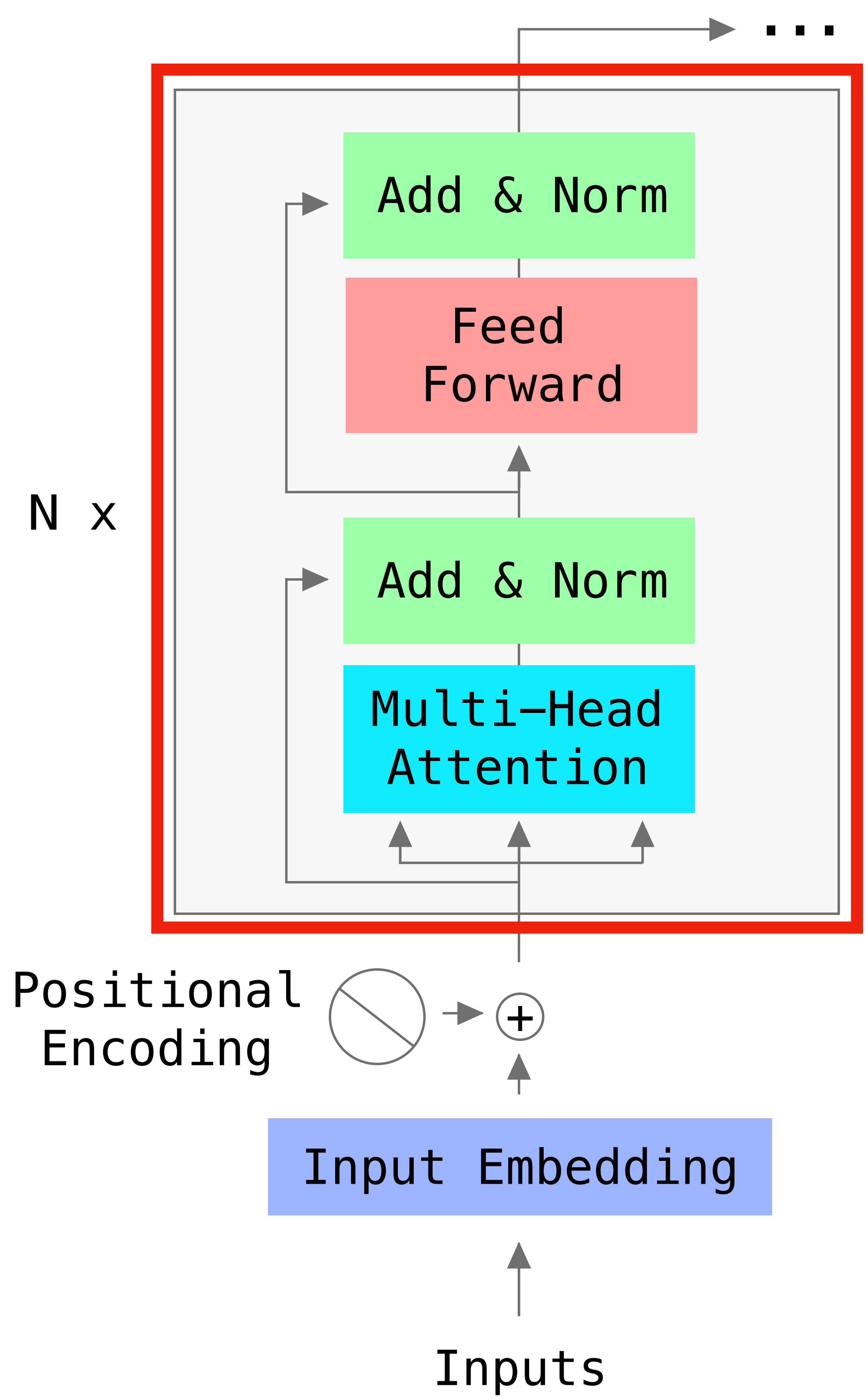
1. Input embedding.
2. Position encodings are element-wise added to the embeddings.



Transformers

Transformer Block

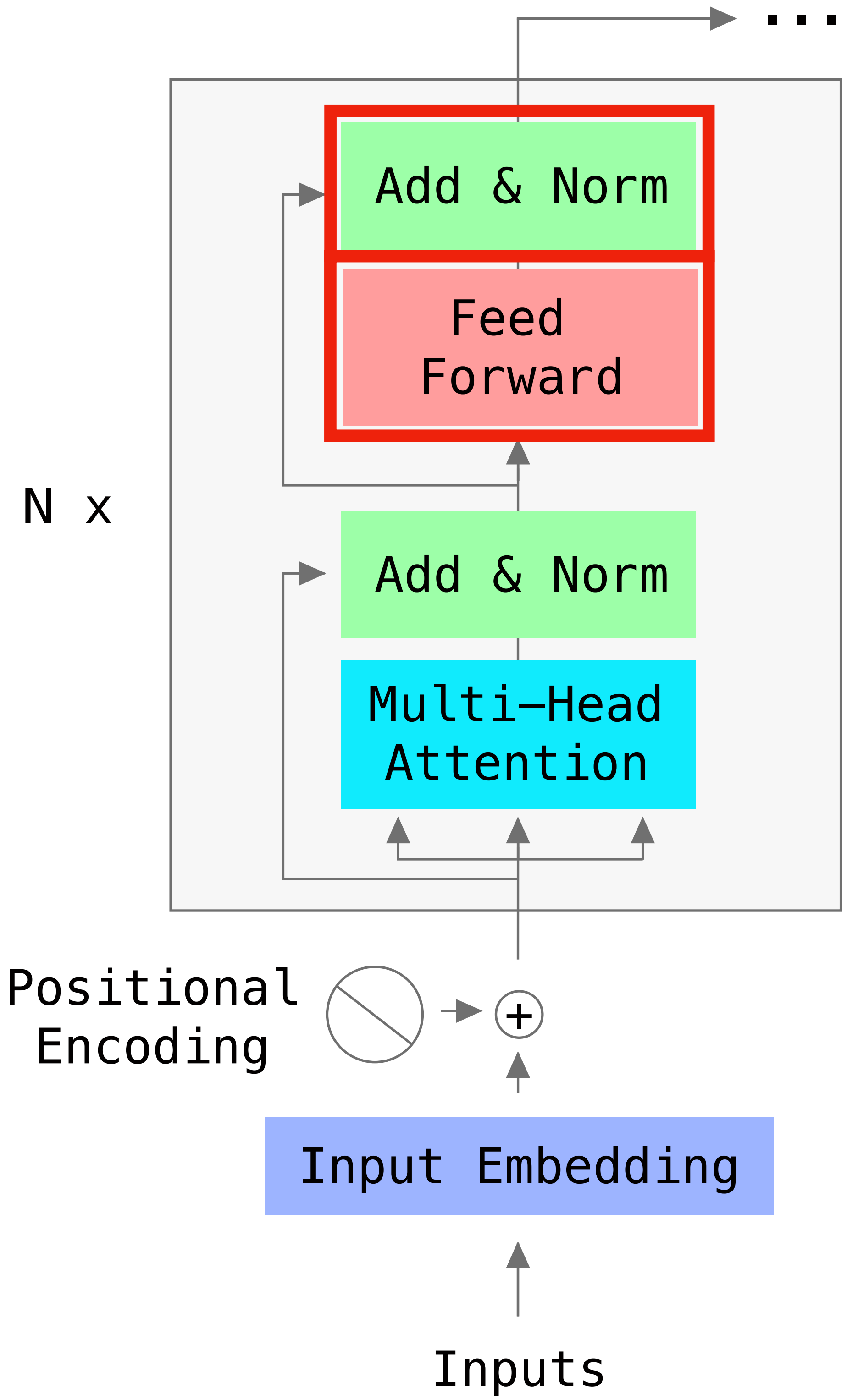
1. Input embedding.
2. Position encodings are element-wise added to the embeddings.
3. Stacked transformer blocks.



Transformers

Transformer Block

You may have noticed: there are two modules in the block we haven't covered.



Transformers

1. Attention
2. Multi-head Attention
3. Transformer Block
4. Other Modules
5. Models

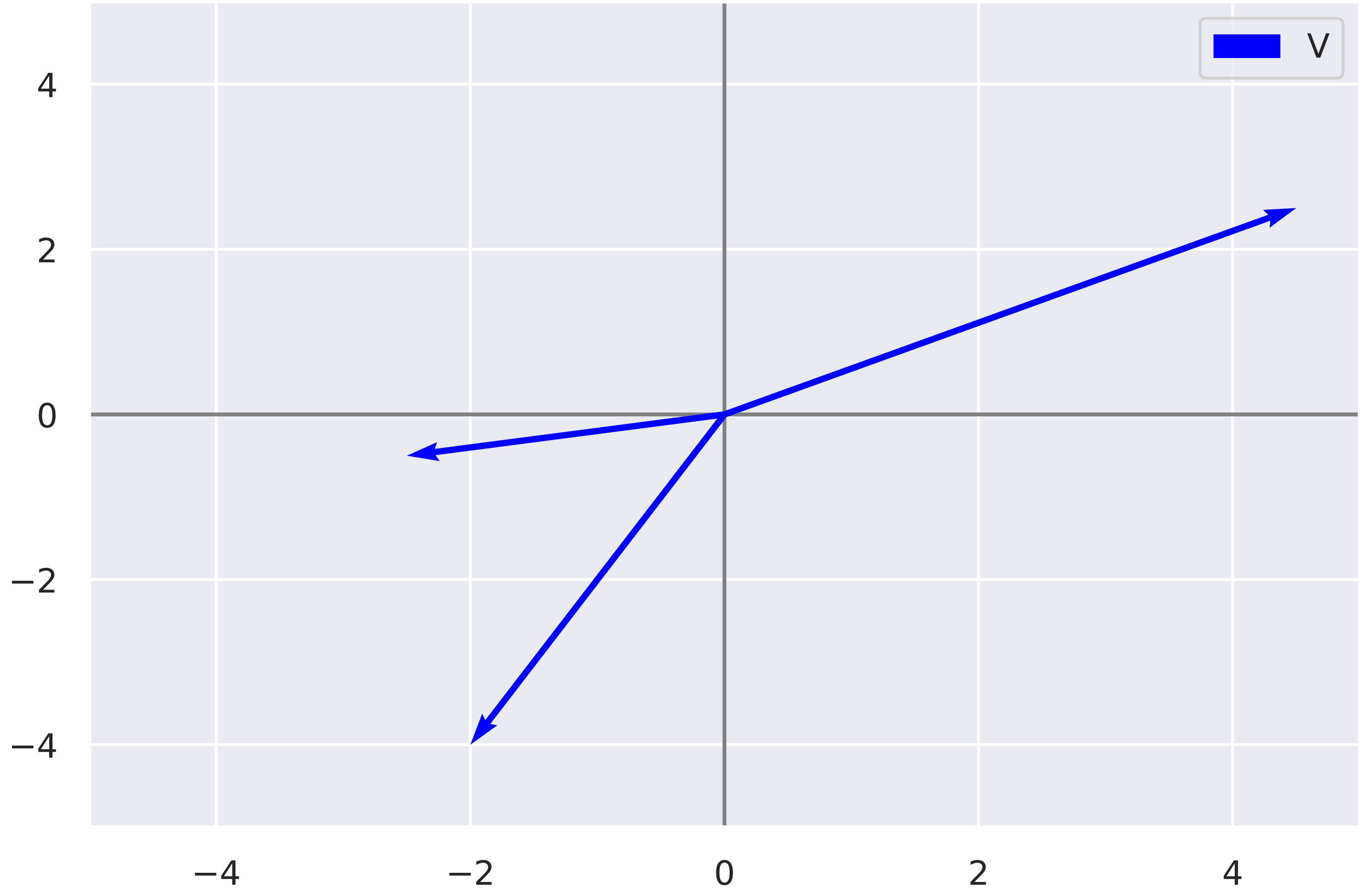
Transformers

Positional Encoding

- Positional Encodings are meant to replace the ordering information lost (as all the vectors are operated on in parallel).
- These could directly learned; but the authors opt for a “simpler” approach.

Transformers

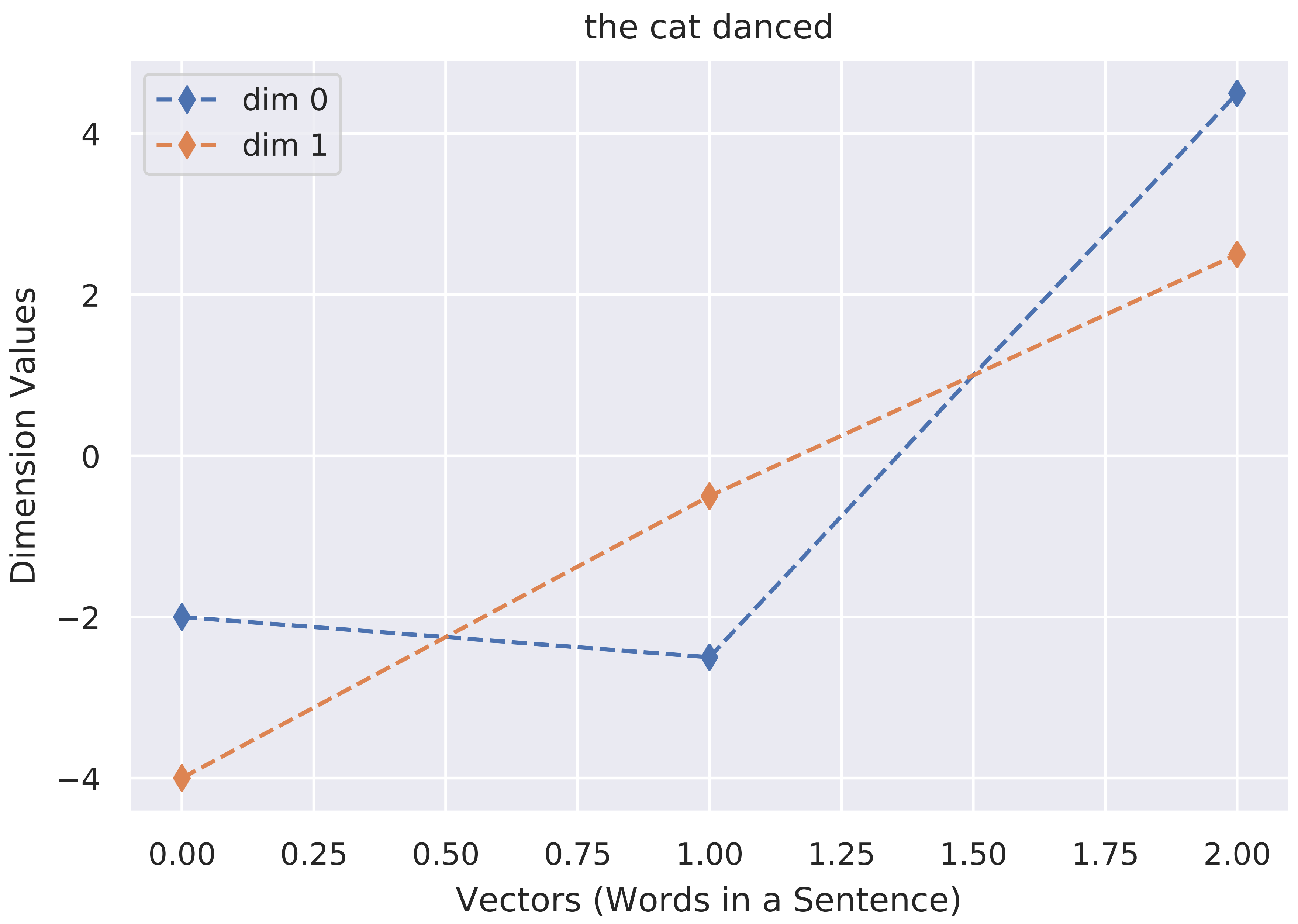
Positional Encoding




```
[
    [-2, -4],          # v1: the
    [-2.5, -0.5],      # v2: cat
    [4.5 ,2.5]         # v3: danced
]
```

Transformers

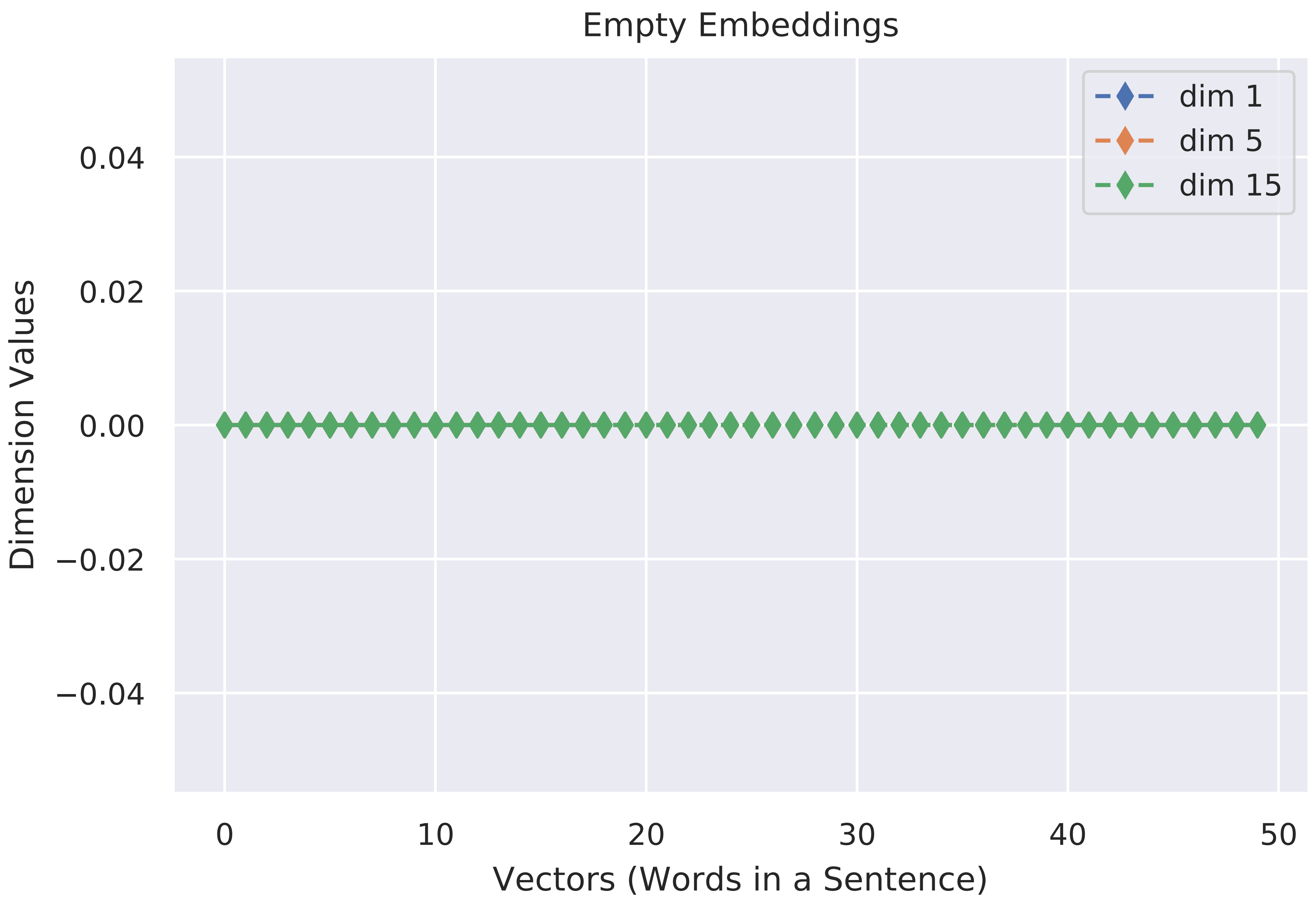
Positional Encoding




```
[
    [-2, -4],         # v1: the
    [-2.5, -0.5],     # v2: cat
    [4.5 ,2.5]        # v3: danced
]
```

Transformers

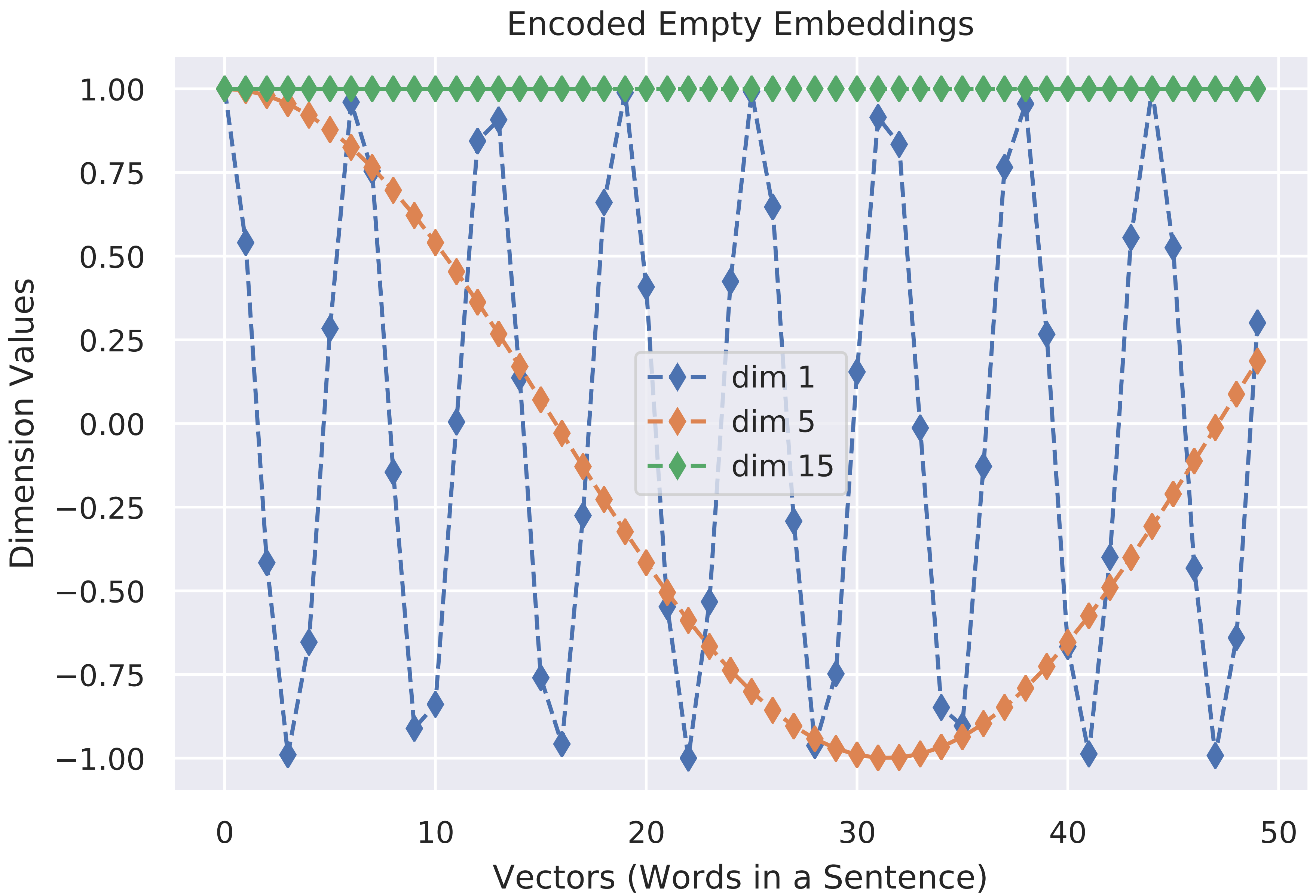
Positional Encoding




```
[
   [0, … 0],   # v1: the
   [0, … 0],   # v2: cat
   [0, … 0]    # v3: danced
   …
]
```

Transformers

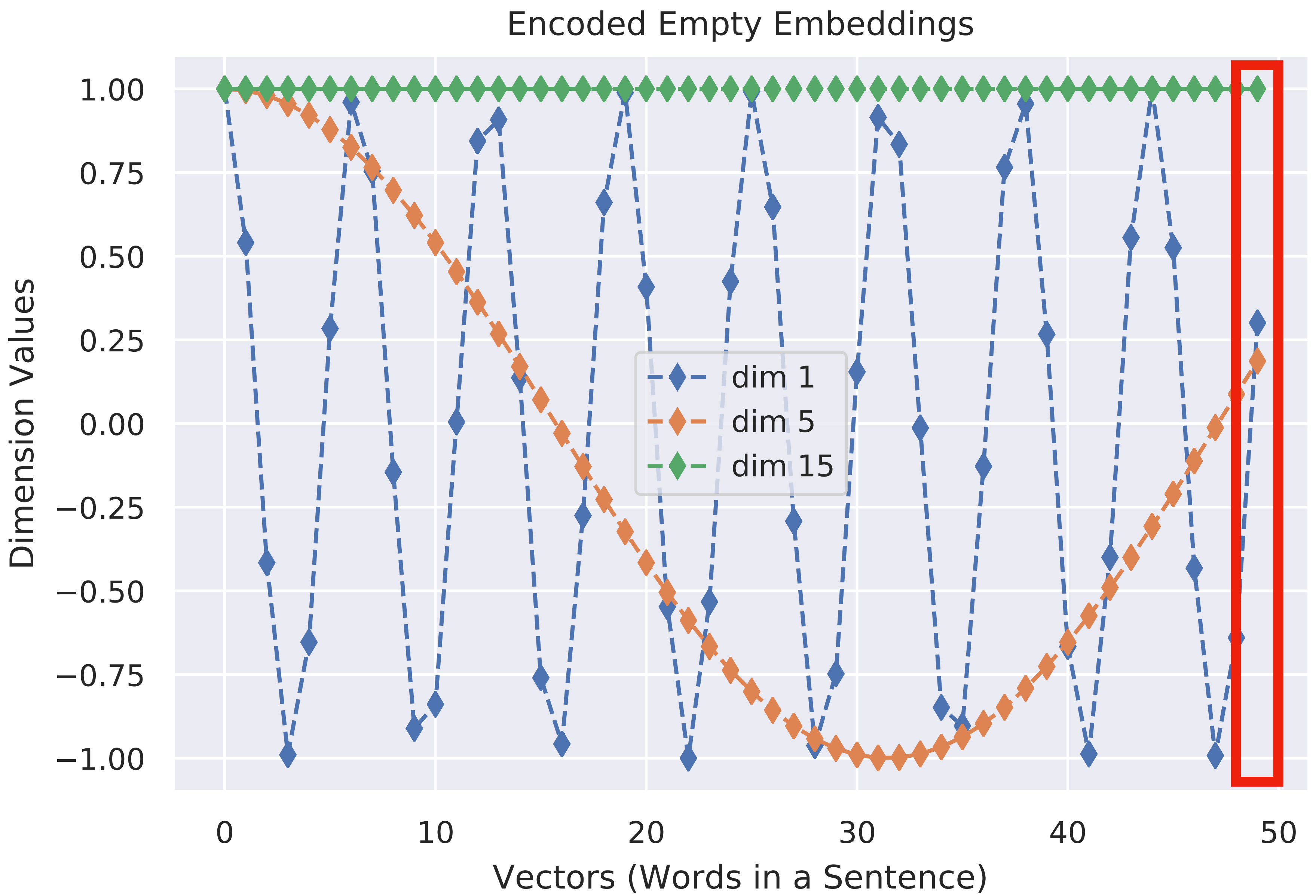
Positional Encoding




```
[
   [0, … 0],    # v1: the
   [0, … 0],    # v2: cat
   [0, … 0]     # v3: danced
   …
]
```

Transformers

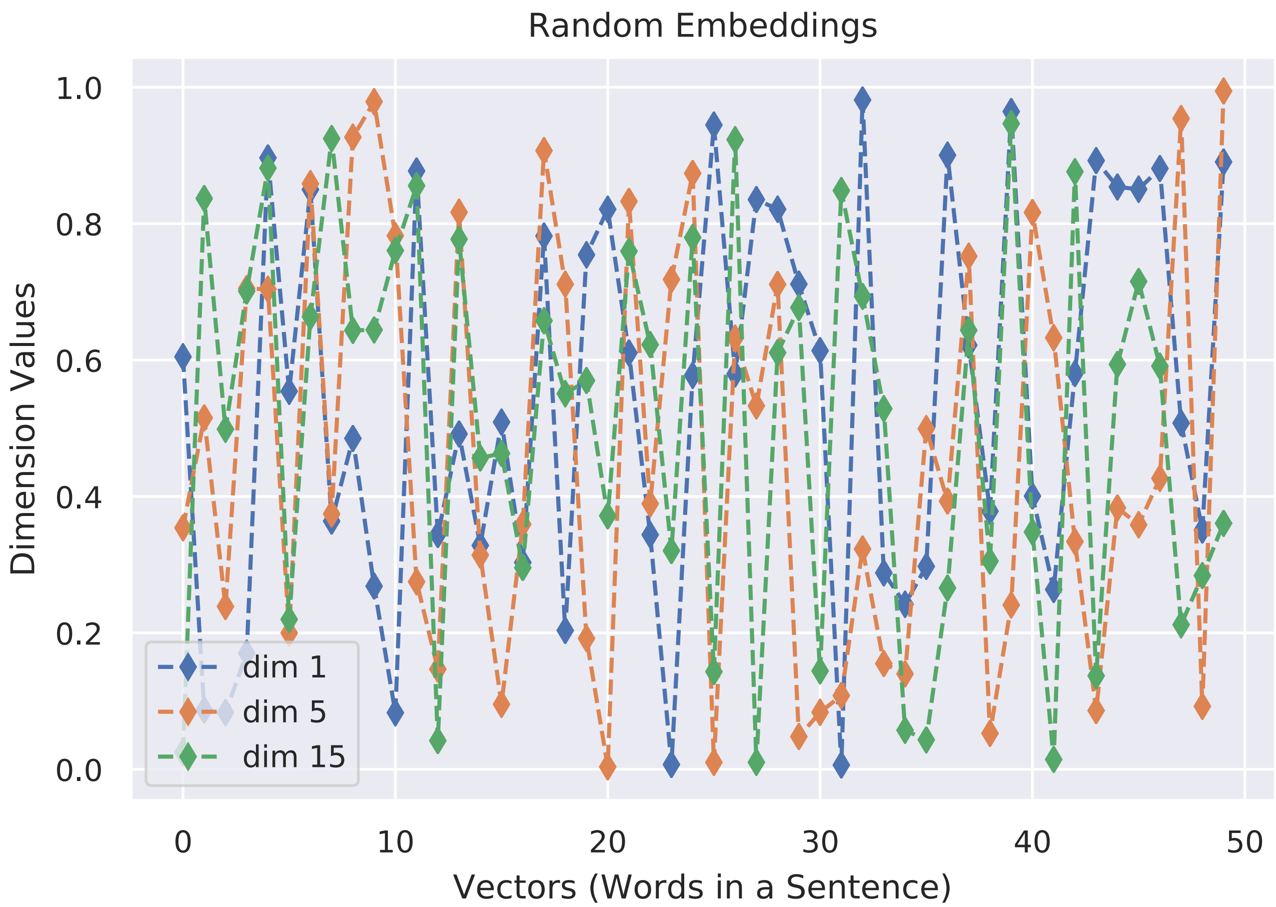
Positional Encoding

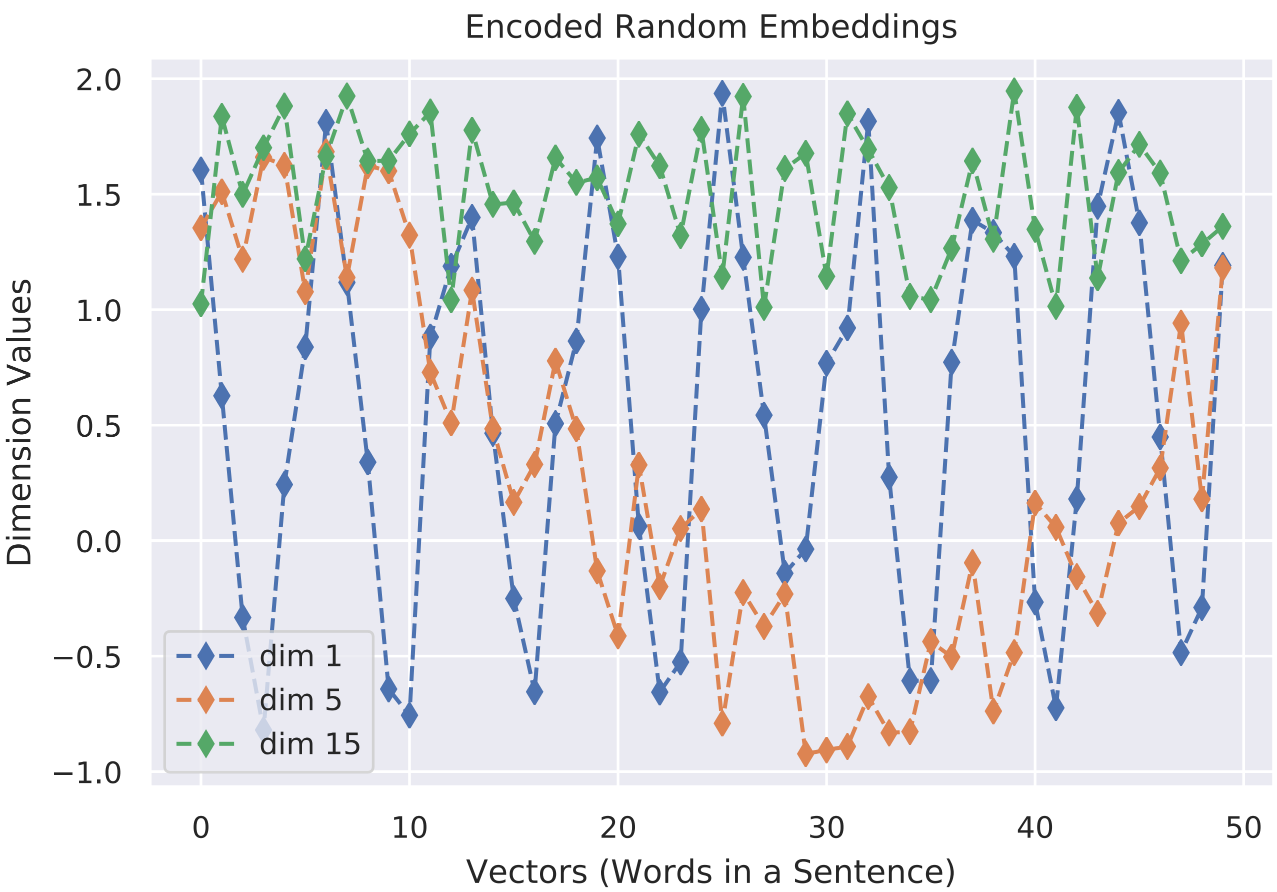



```
[
   [0, … 0],   # v1: the
   [0, … 0],   # v2: cat
   [0, … 0]    # v3: danced
   …
]
```

Transformers

Positional Encoding





Transformers

Add & Norm

```
def add-norm(sublayer, x):
  return LayerNorm(x + sublayer(x))
```

- Sub-layer connection between the input of the layer and the output of the layer.
- This structure has been used in a wide range of networks; its effective at “stabilizing the gradient”, and letting us build deeper networks.

Srivastava, Rupesh Kumar, Klaus Greff, and Jürgen Schmidhuber. "Highway networks." *arXiv preprint arXiv:1505.00387* (2015).

He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

Transformers

Add & Norm

```
def add-norm(sublayer, x):
  return LayerNorm(x + sublayer(x))
```

"While the traditional plain neural architectures become increasingly difficult to train with increasing network depth (even with variance-preserving initialization), our experiments show that optimization of highway networks is not hampered even as network depth increases to a hundred layers."

Srivastava, Rupesh Kumar, Klaus Greff, and Jürgen Schmidhuber. "Highway networks." *arXiv preprint arXiv:1505.00387* (2015).

Transformers

Add & Norm

```
def add-norm(sublayer, x):
  return LayerNorm(x + sublayer(x))
```

- This is layer normalization across the residual connection between the input and the output of the sublayer (e.g. multi-head attention).
- It's similar to batch-normalization, except that all variables are normalized per layer.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer normalization." *arXiv preprint arXiv:1607.06450* (2016).

https://pytorch.org/docs/stable/nn.html#layernorm

Transformers

Add & Norm

```
def add-norm(sublayer, x):
  return LayerNorm(x + sublayer(x))
```

"Layer normalization is very effective at stabilizing the hidden state dynamics in recurrent networks. Empirically, we show that layer normalization can substantially reduce the training time compared with previously published techniques."

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer normalization." ***arXiv preprint arXiv:1607.06450*** **(2016).**

https://pytorch.org/docs/stable/nn.html#layernorm

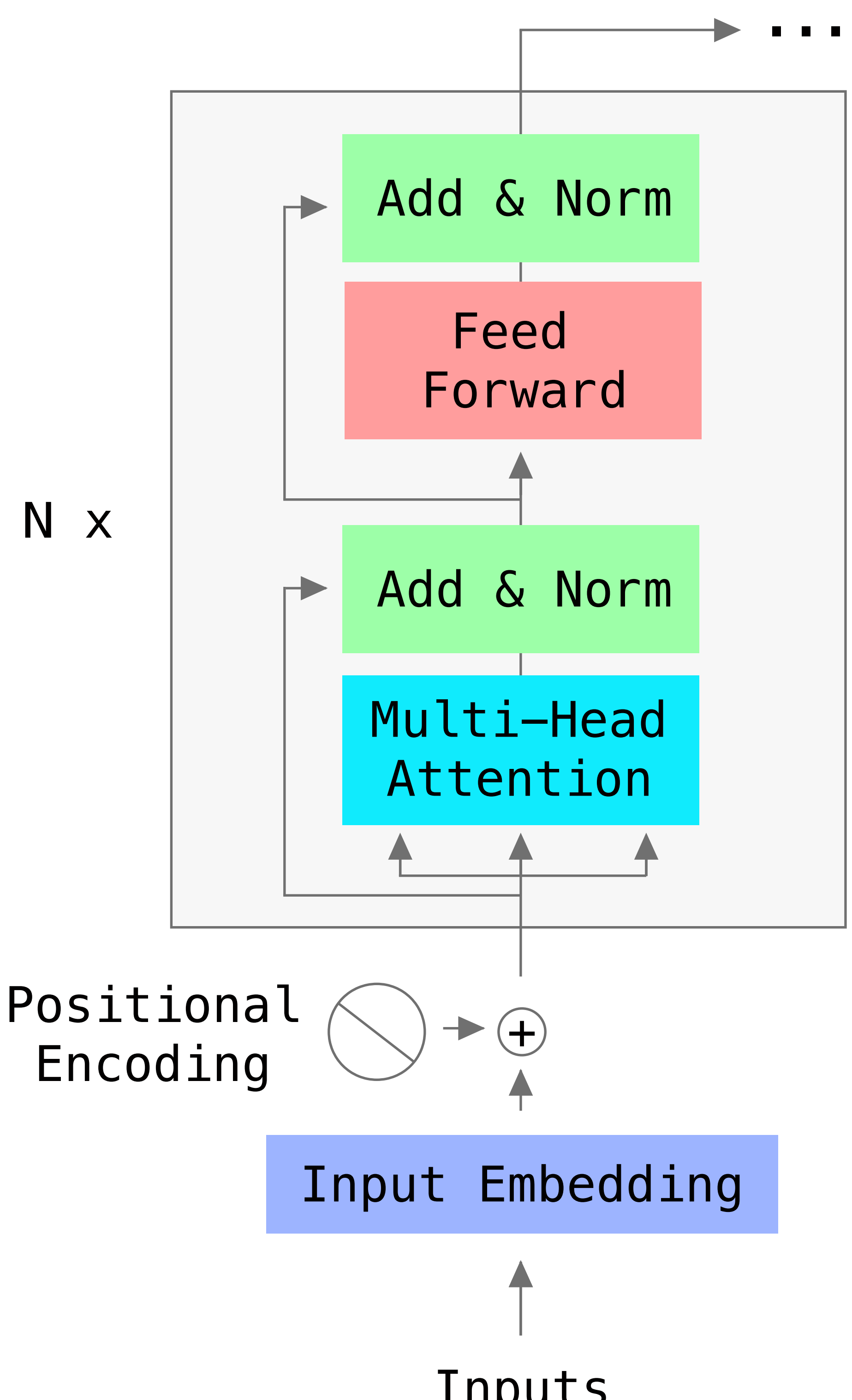
Transformers

Position-wise Feed Forward

$$\mathrm{FFN}(x) = \sigma(x \cdot W_1 + b_1) \cdot W_2 + b_2$$

Transformers

Transformer Block Recap



1. Attention is used for *Self-Attention*.
2. Transformers don't require the size of inputs to match or be padded.
3. The operations are all *parallel* across the inputs.

Transformers

1. Attention
2. Multi-head Attention
3. Transformer Block
4. Other Modules
5. Models

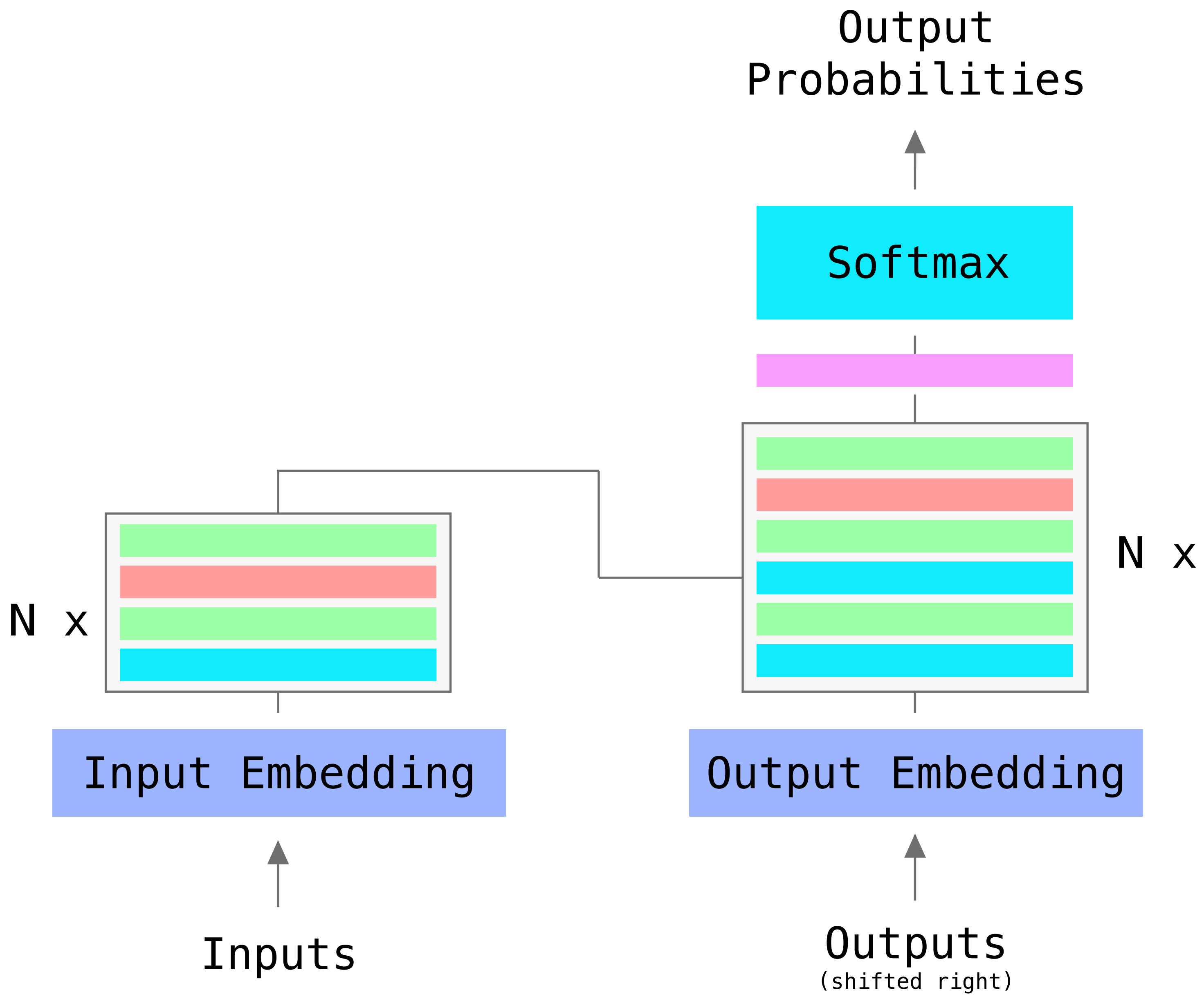
Transformers

Models

Transformer Encoder-Decoder	(T-ED)
Transformer Decoder	(T-D)
Generative Pre-Training	(GPT-1)
Generative Pre-Training	(GPT-2)
Bidirectional Transformers	(BERT)

Transformers

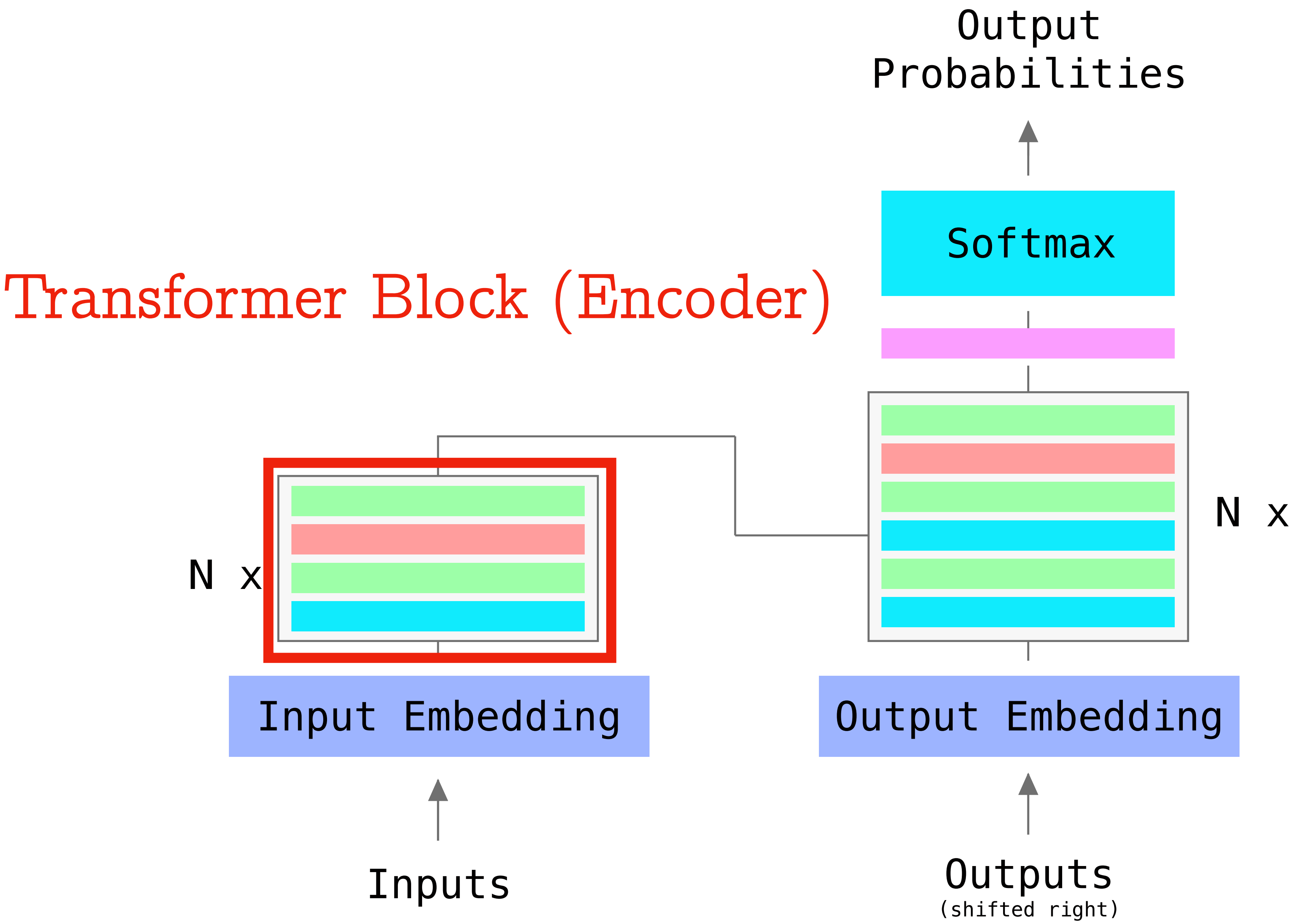
T-ED



Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems*. 2017.

Transformers

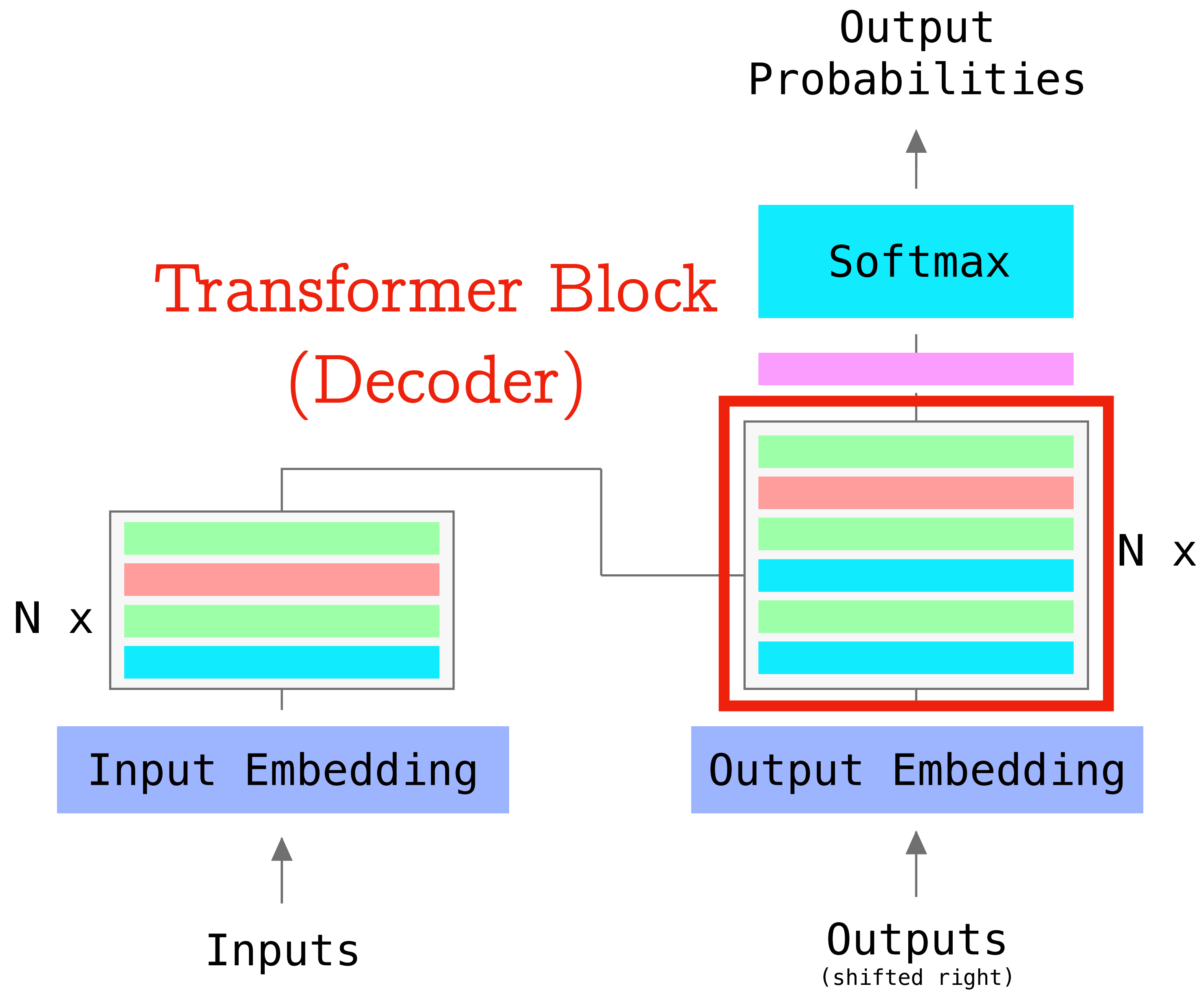
T-ED



Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems.* 2017.

Transformers

T-ED

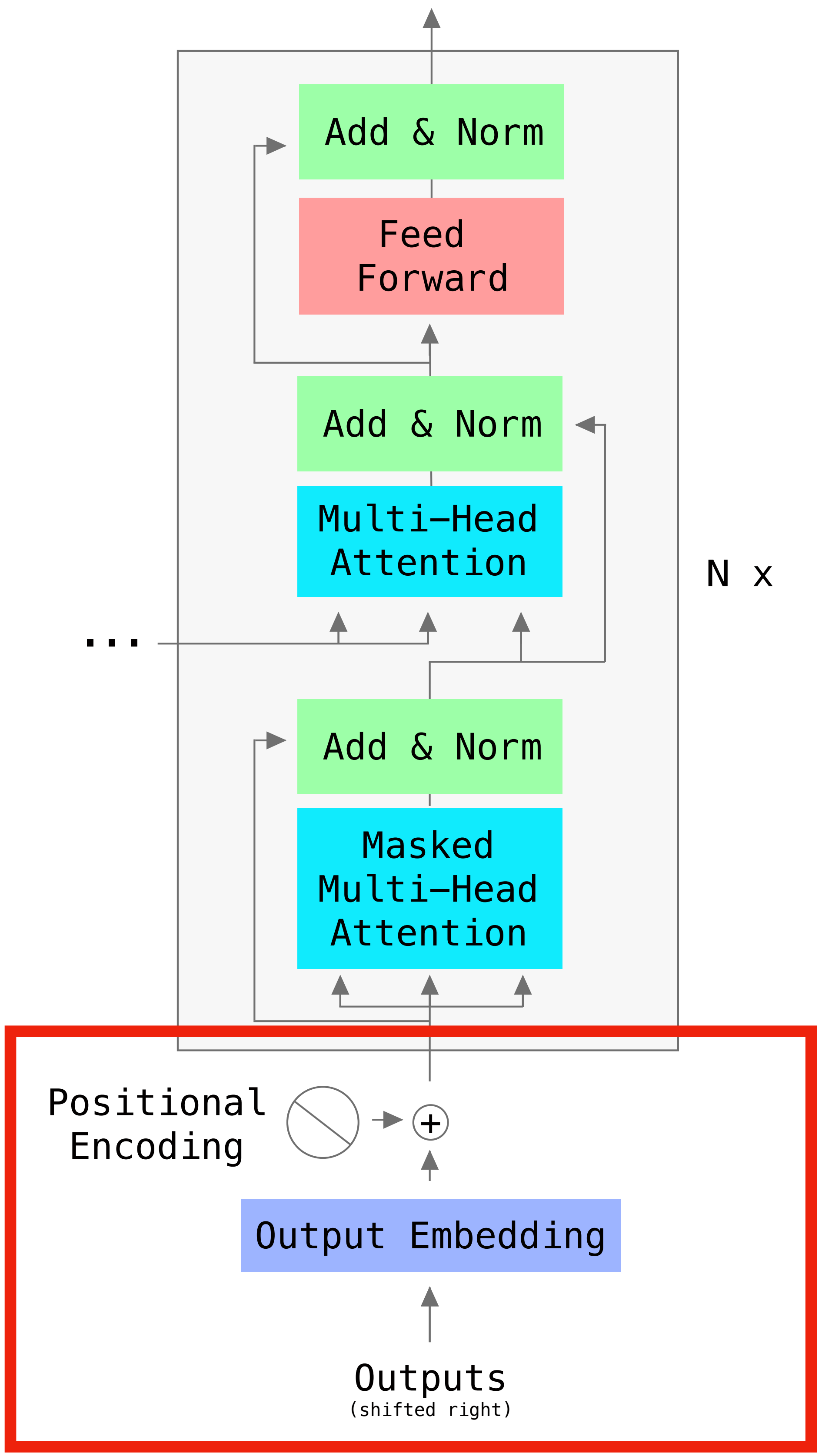


Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems*. 2017.

Transformers

T-ED

The outputs (so-far) are embedded in the same way.

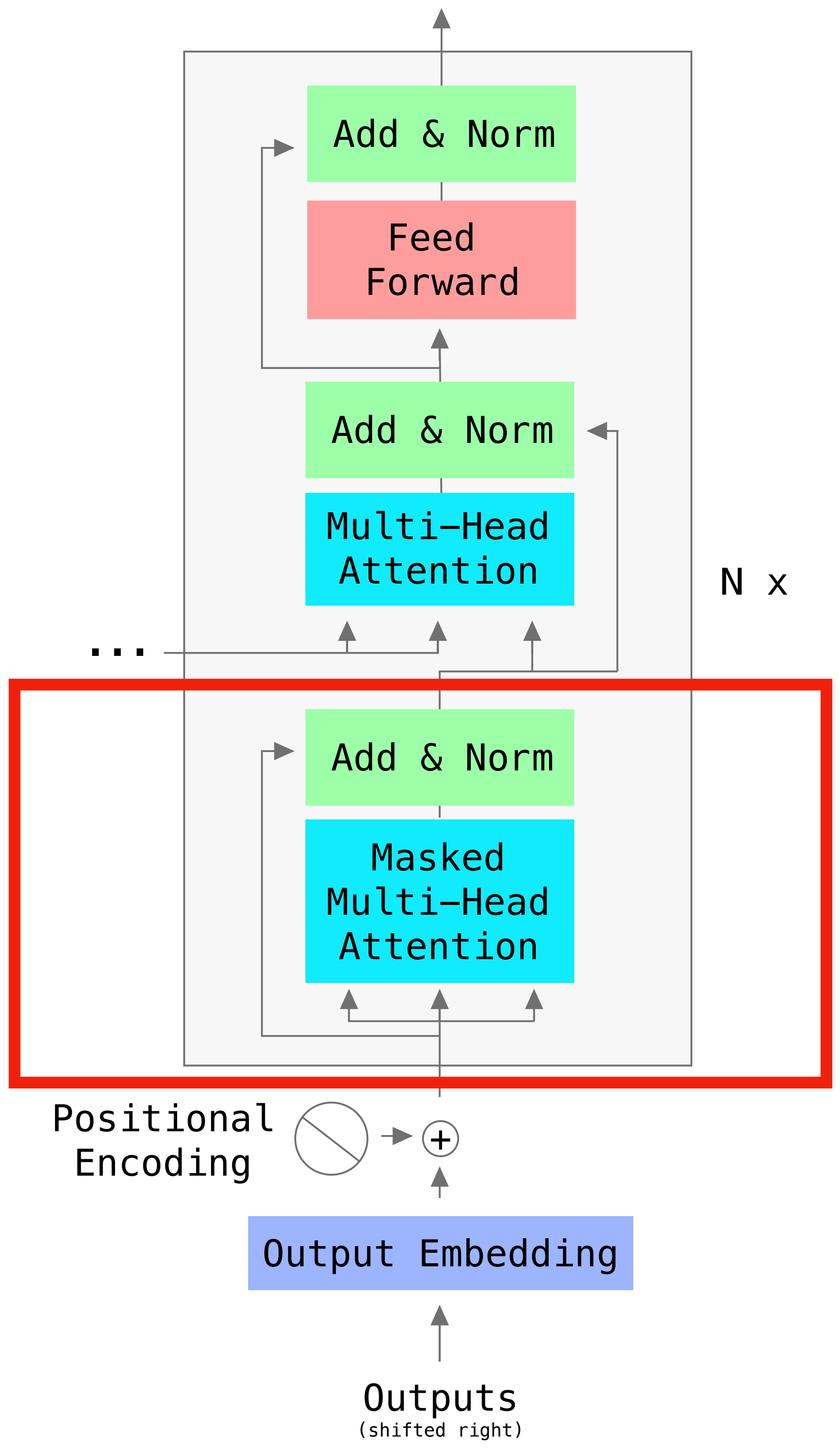


Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems.* 2017.

Transformers

T-ED

Self-attention starts each decoder block.

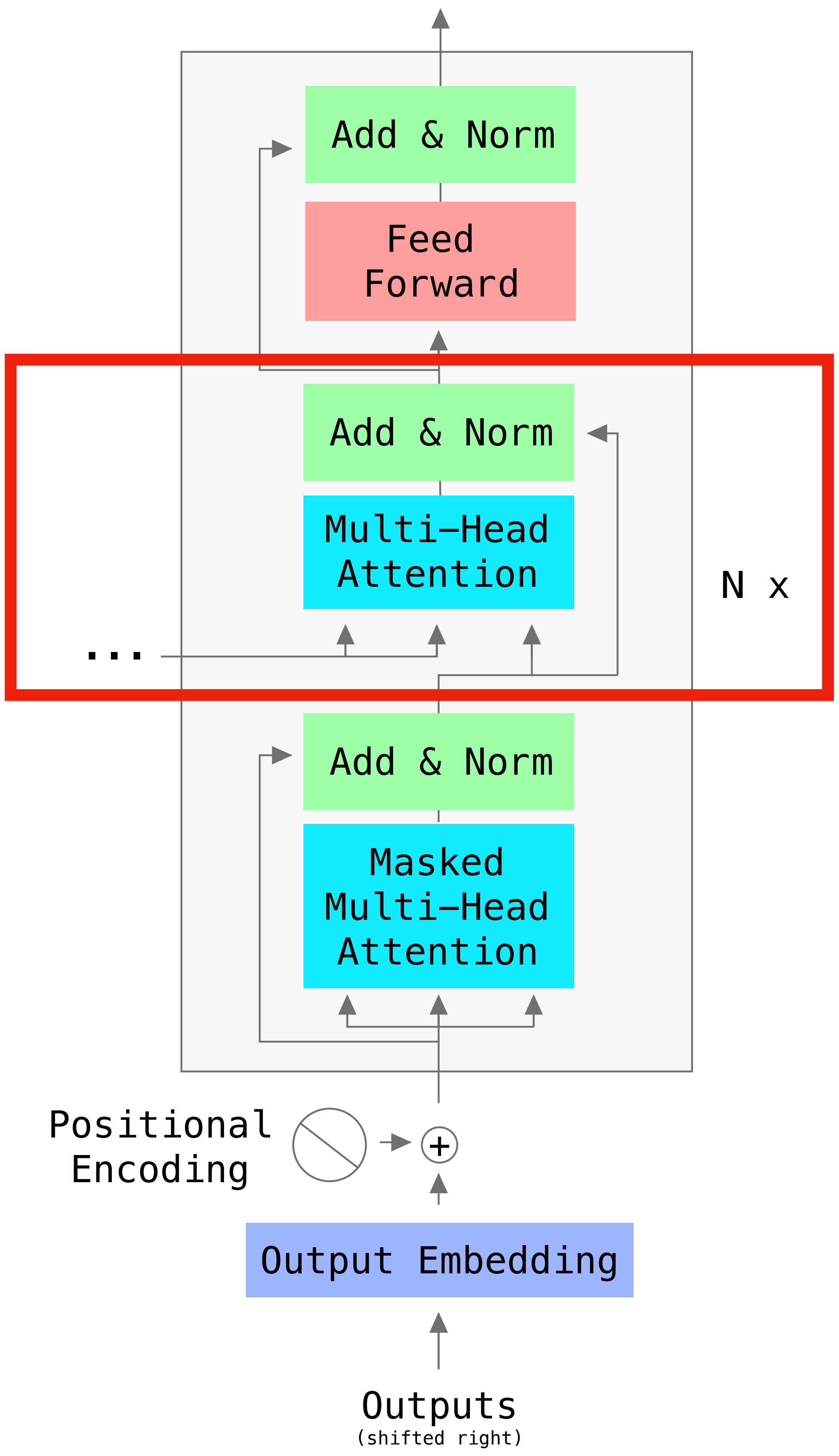


Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems.* 2017.

Transformers

T-ED

Next, attention between the encoded inputs and the encoded outputs, and the outputs are mapped to the inputs.

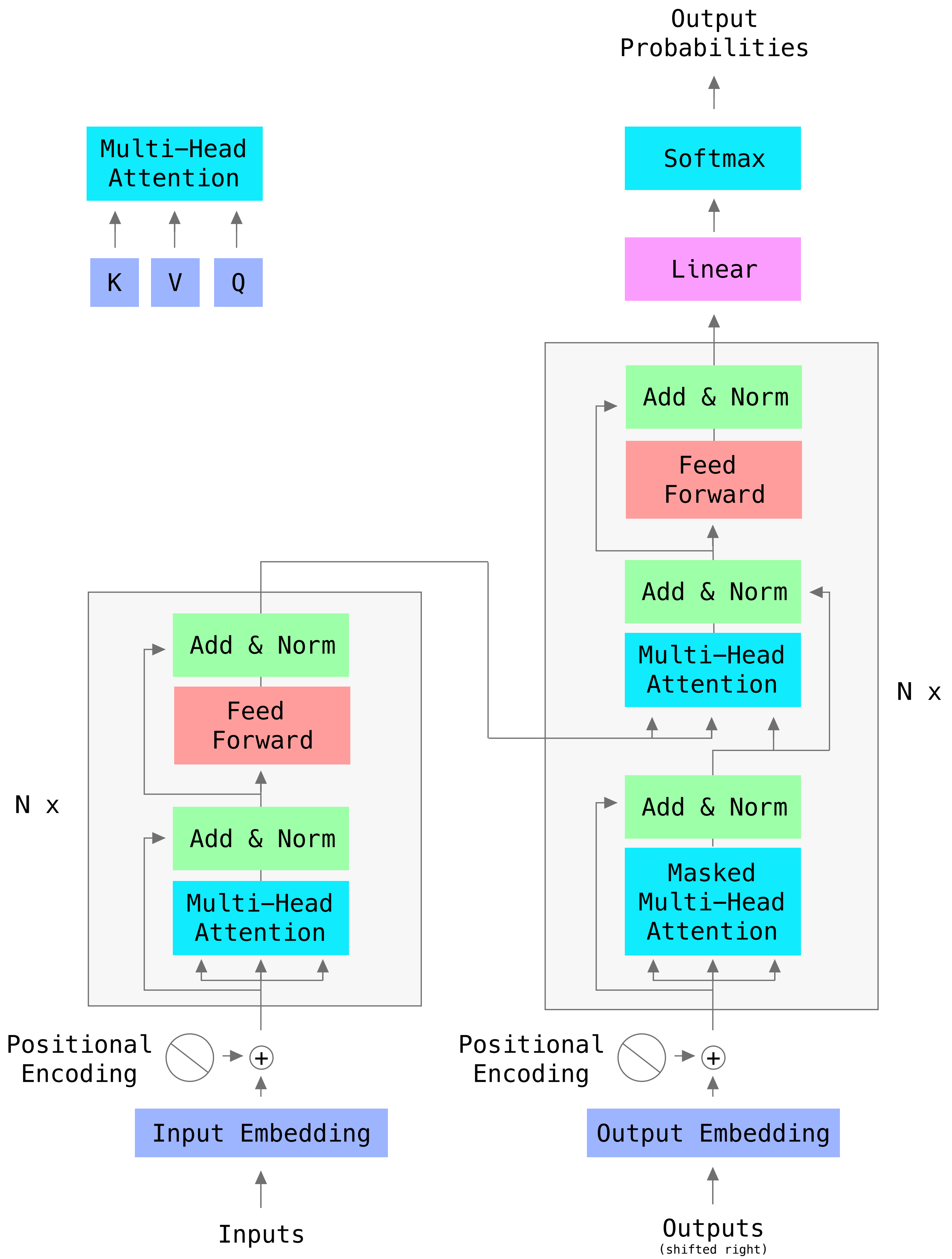


Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems.* 2017.

Transformers

T-ED

Each output is produced auto-regressively.

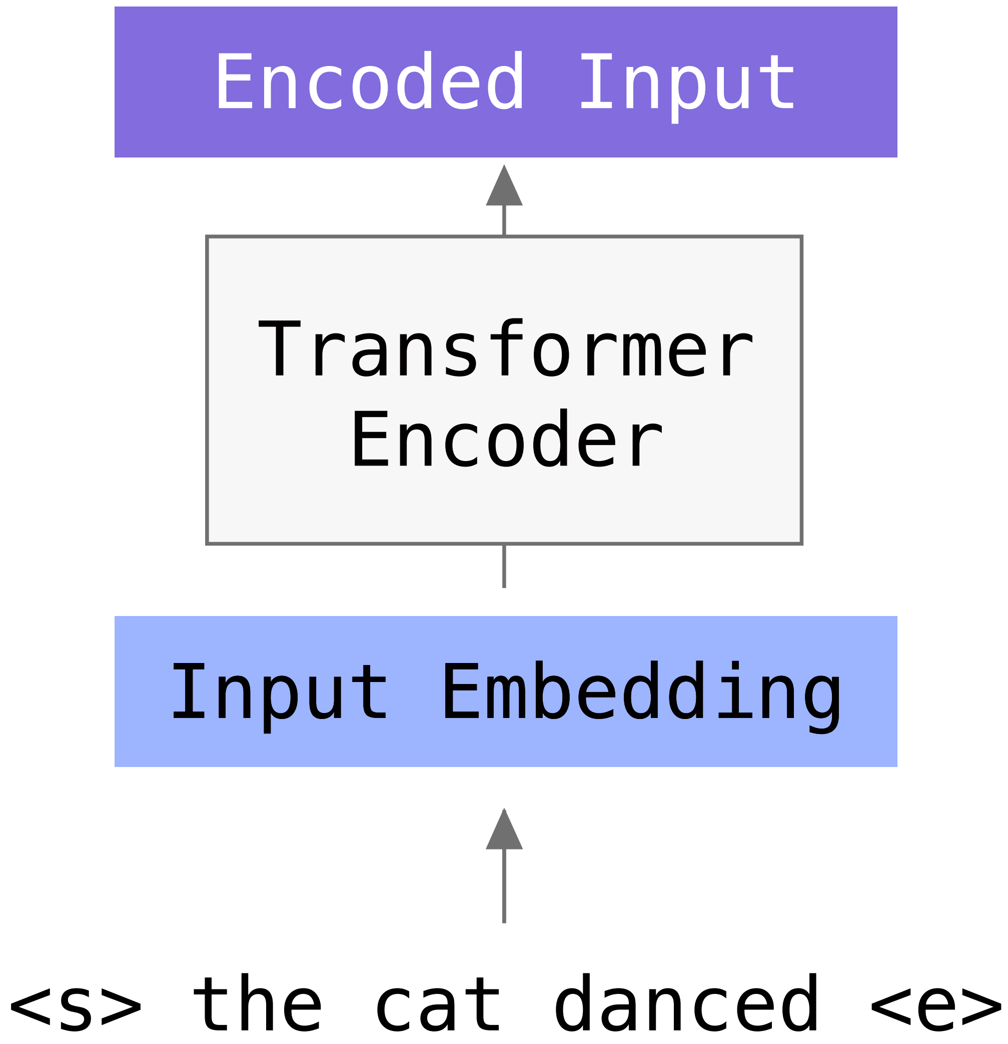


Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems.* 2017.

Test / Live

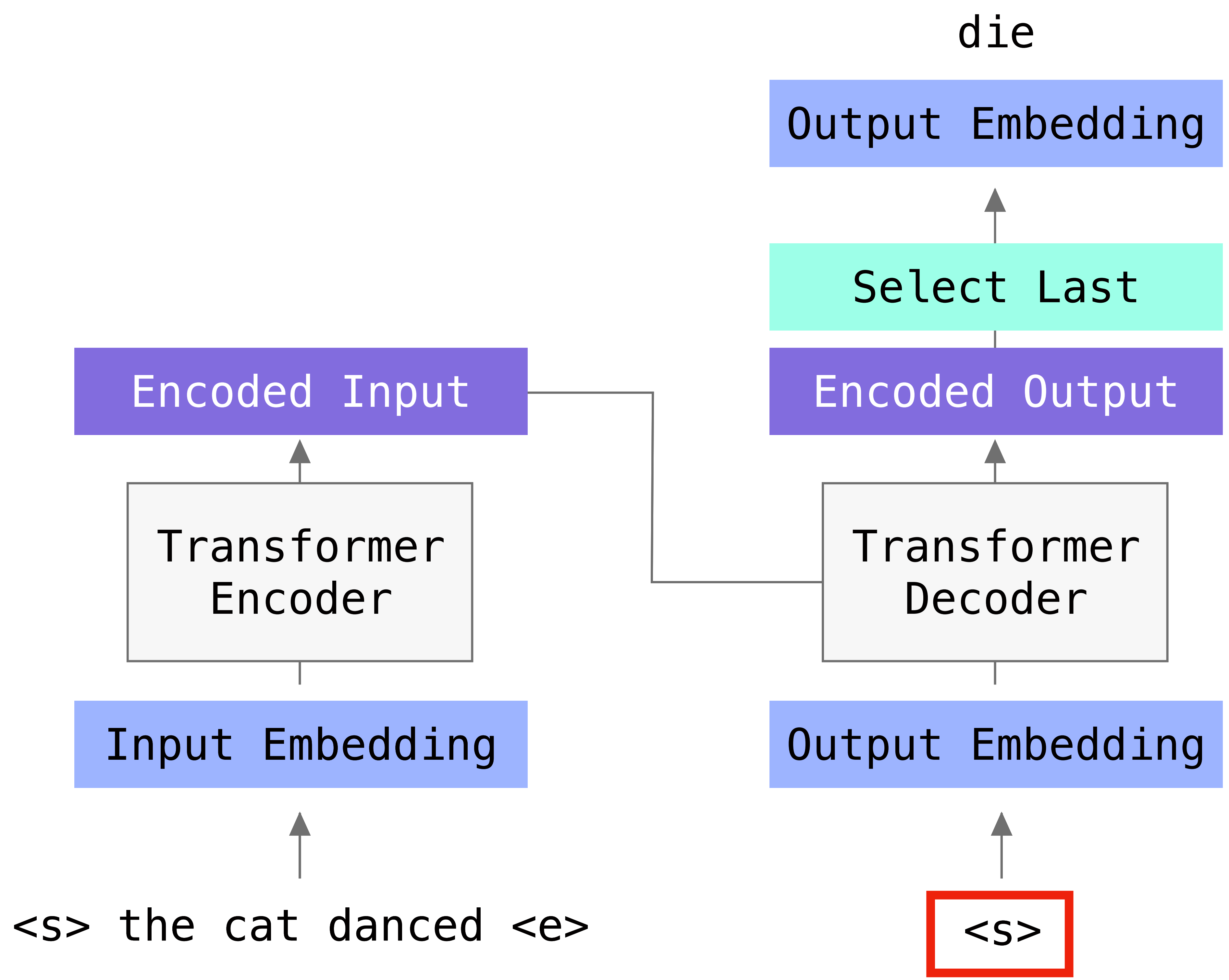
Transformers

Generating Outputs (Greedy)



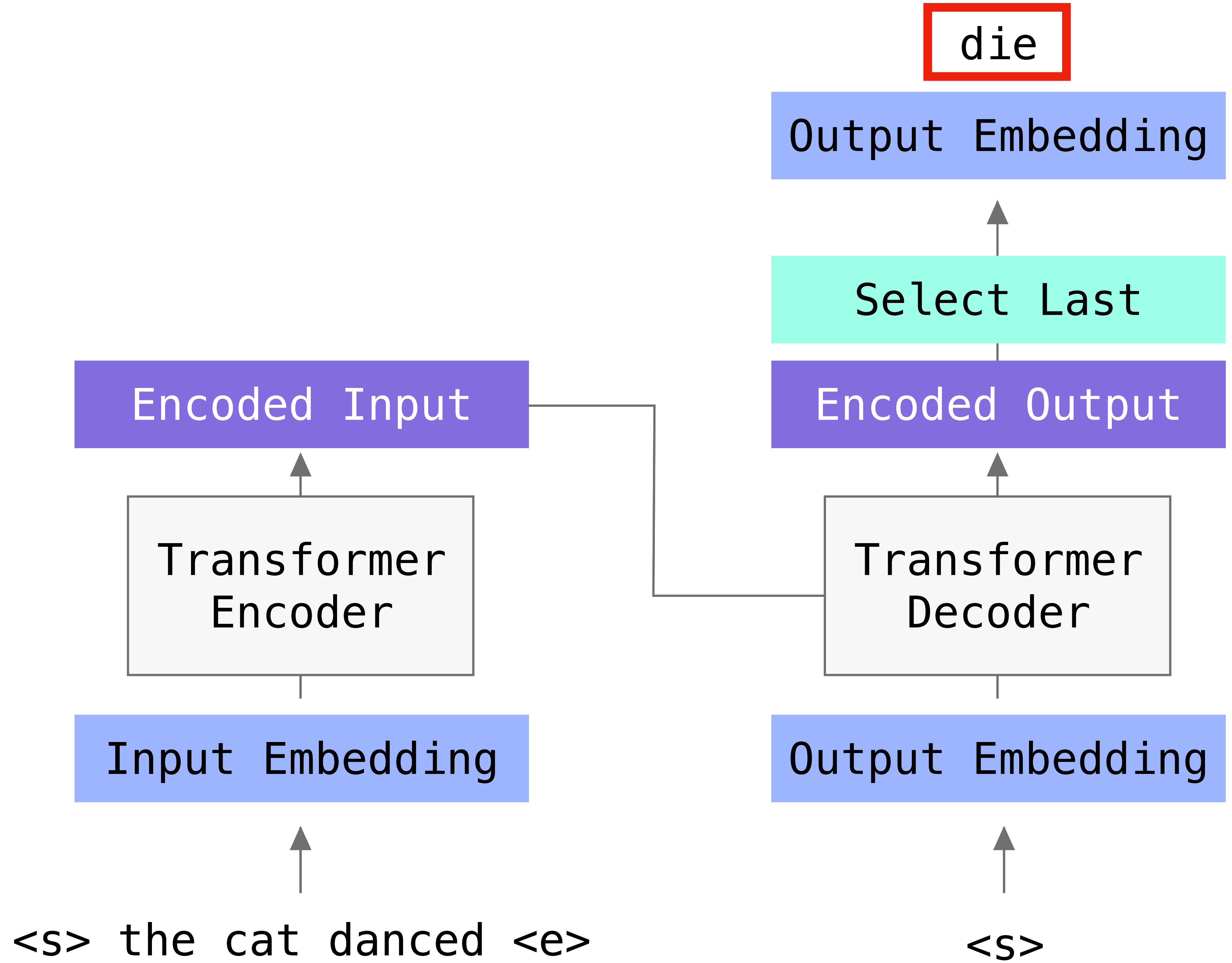
Transformers

Generating Outputs (Greedy)



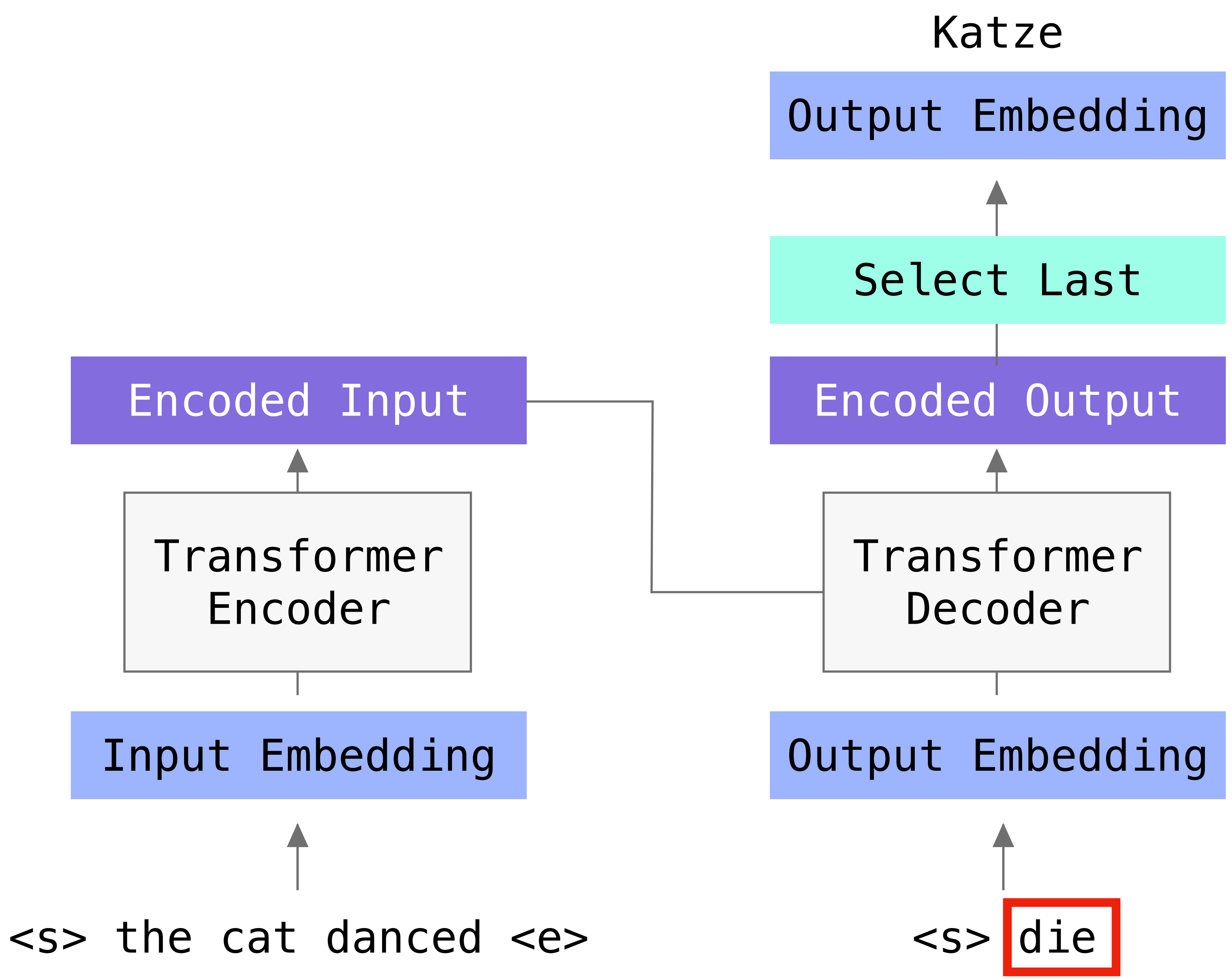
Transformers

Generating Outputs (Greedy)



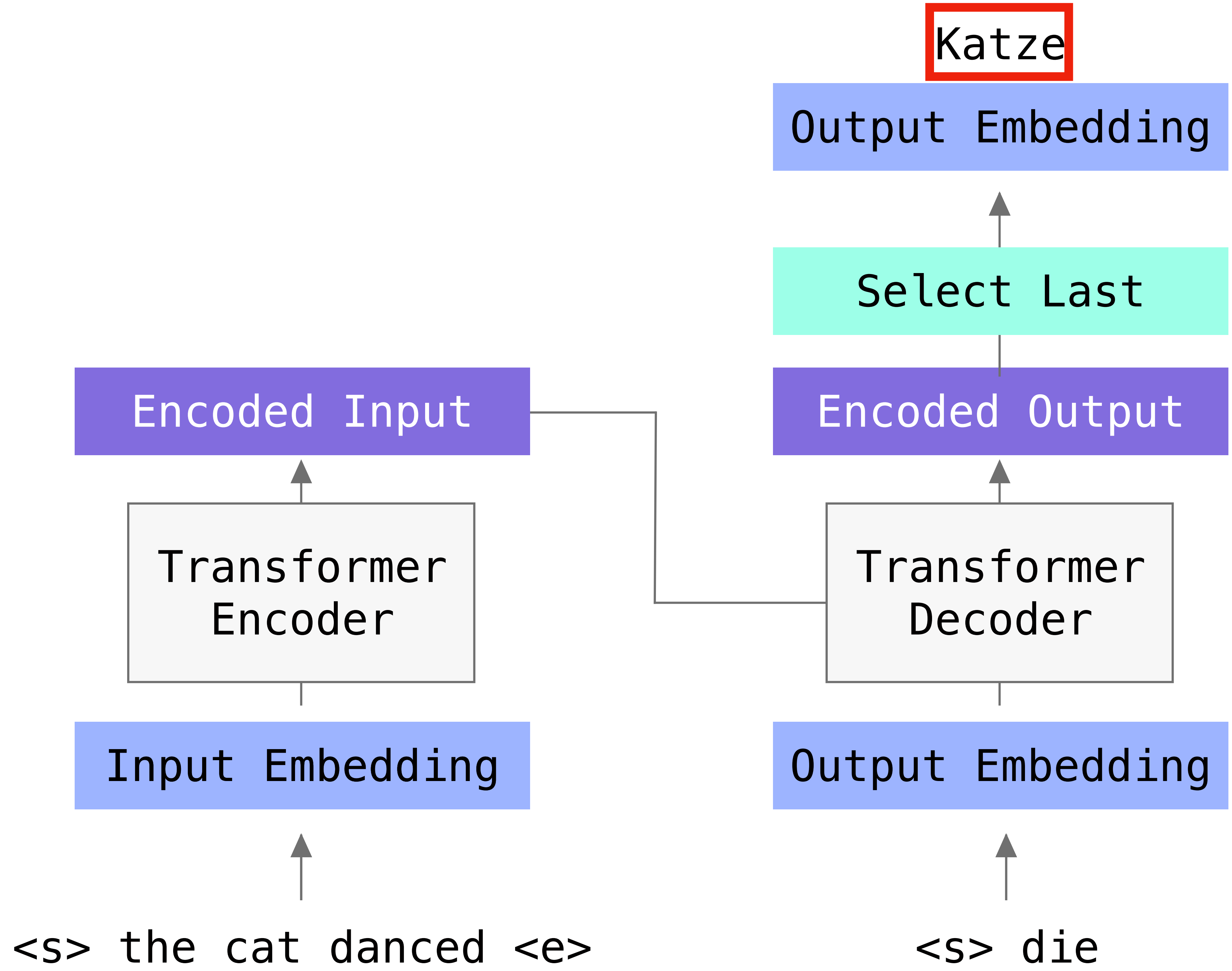
Transformers

Generating Outputs (Greedy)



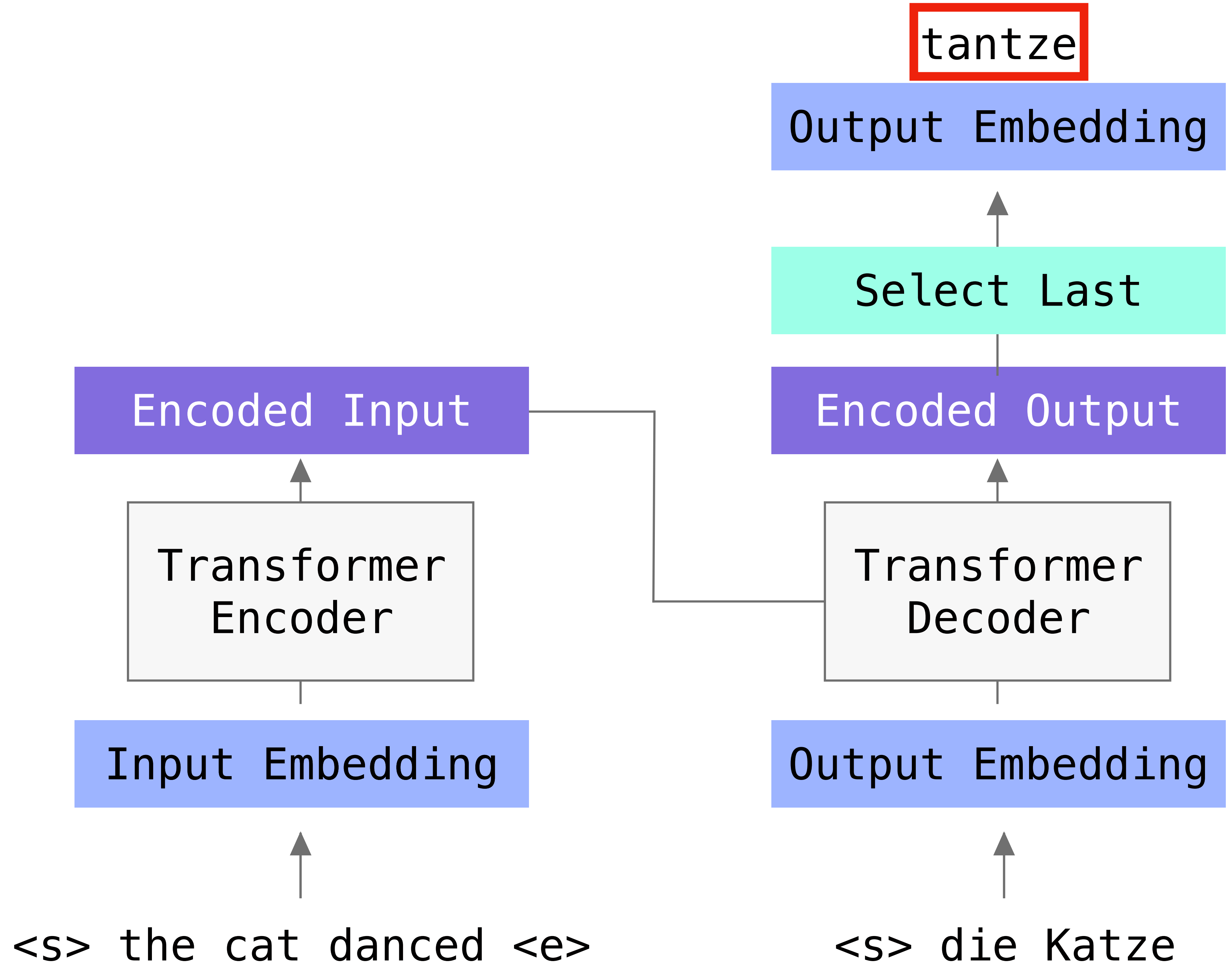
Transformers

Generating Outputs (Greedy)



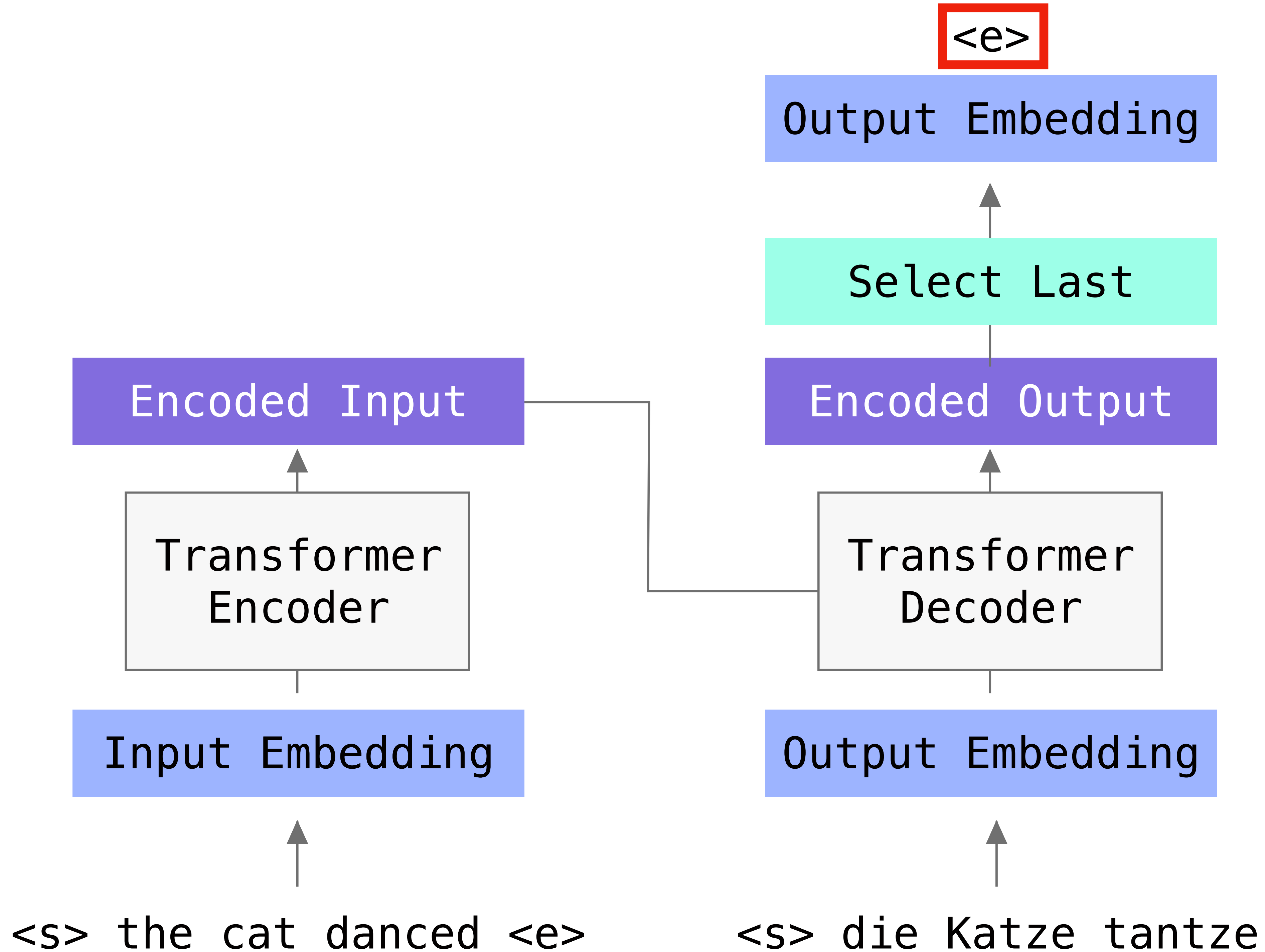
Transformers

Generating Outputs (Greedy)



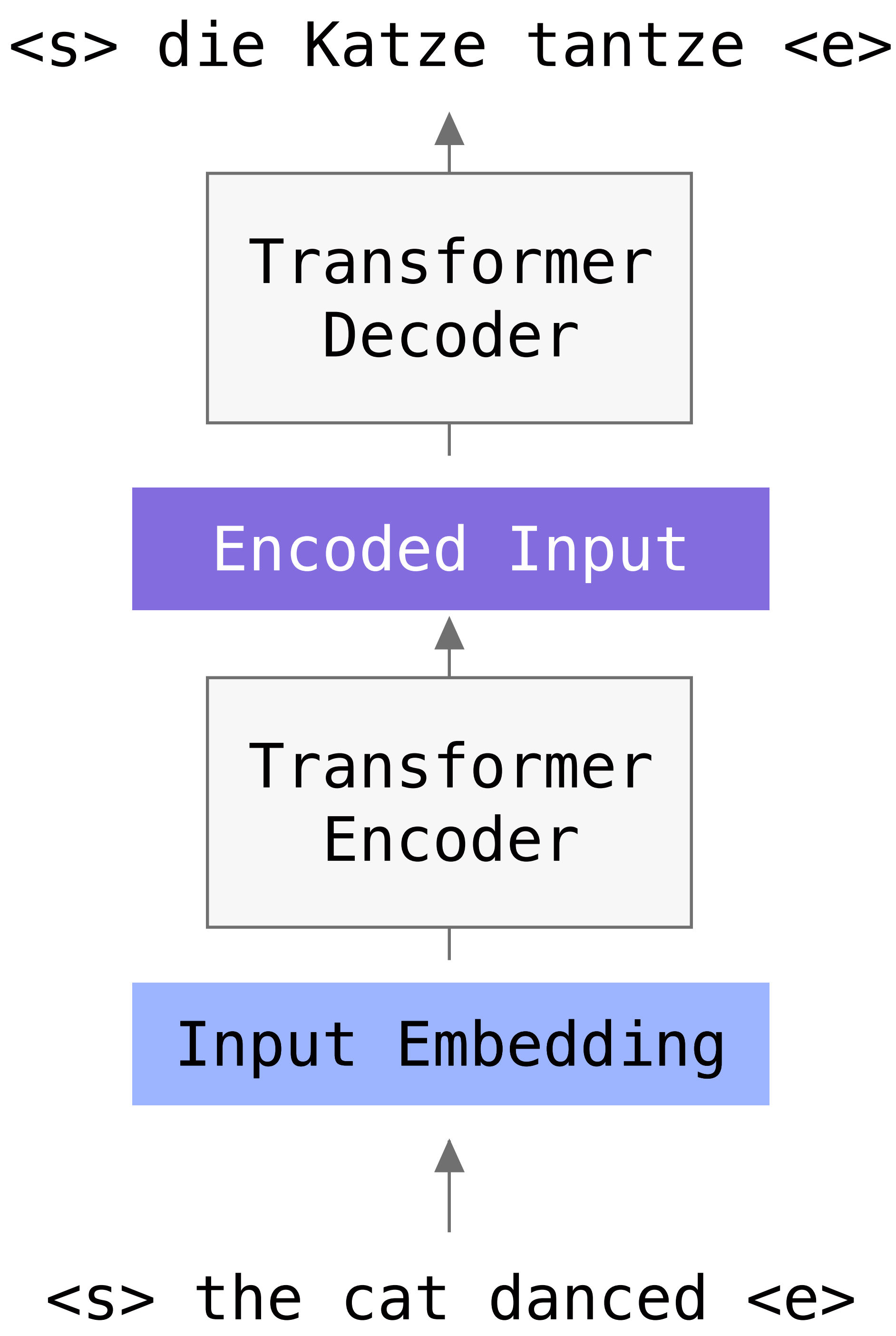
Transformers

Generating Outputs (Greedy)



Transformers

Generating Outputs (Greedy)

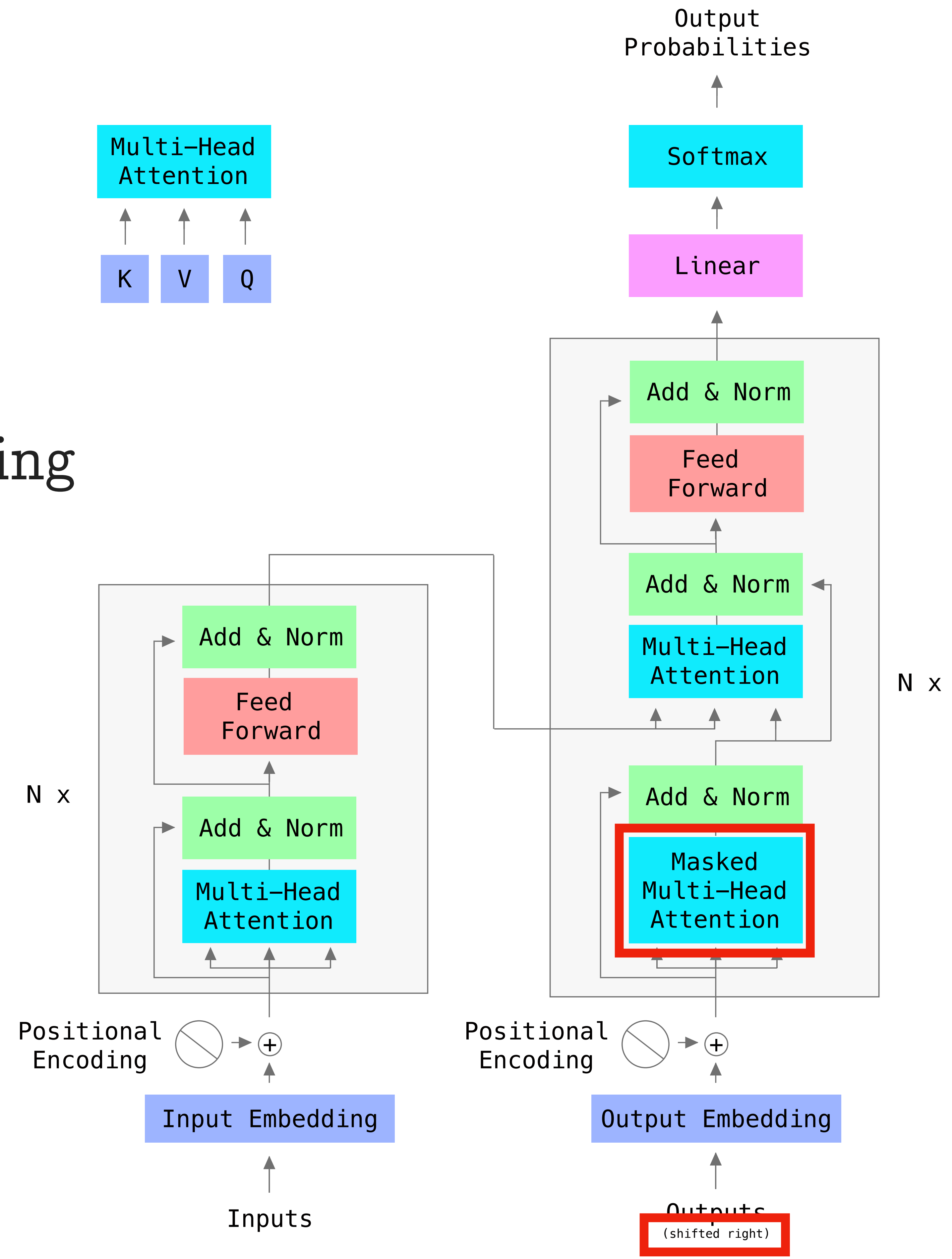


Training

Transformers

T-ED

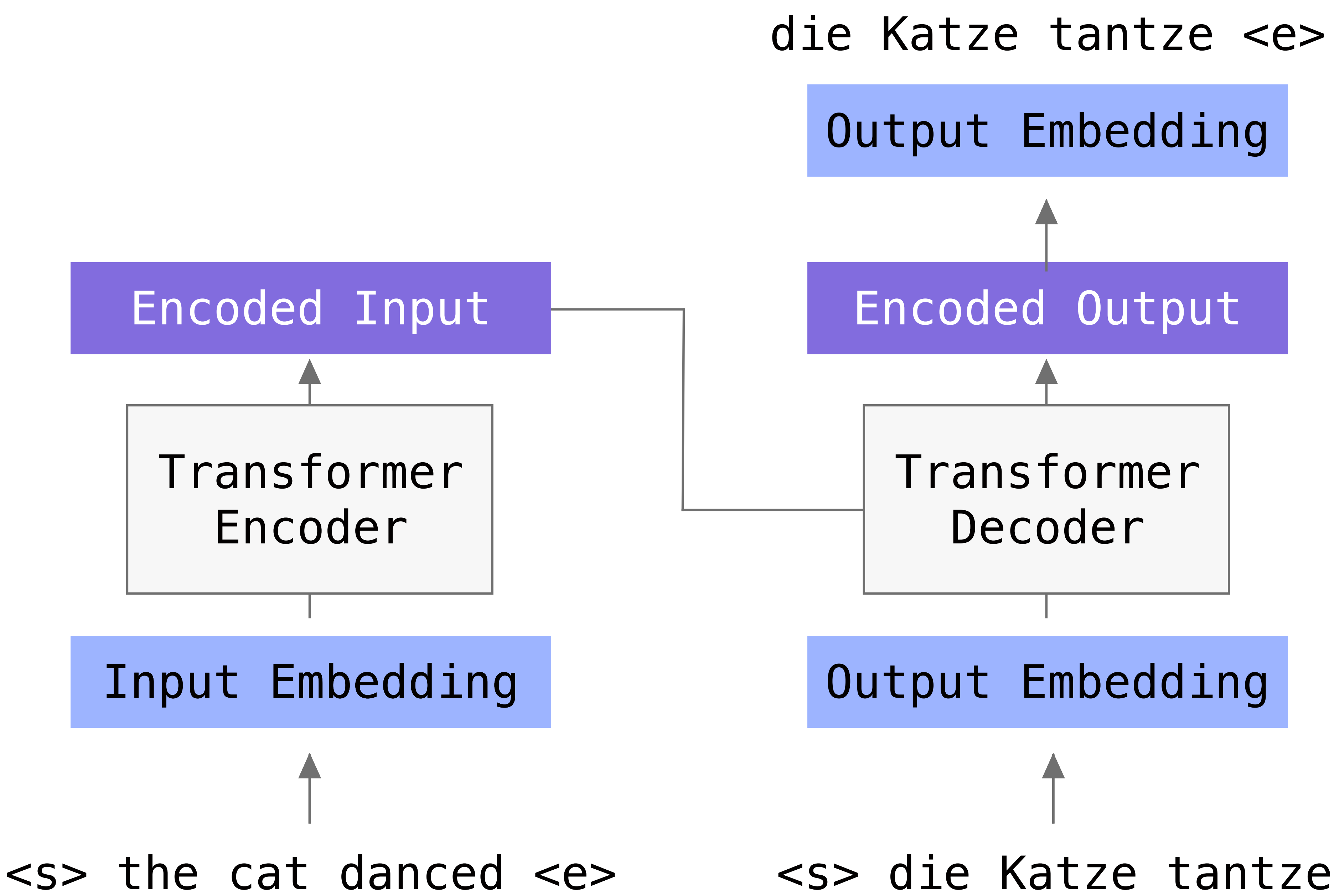
Two minor details: masking & shifting.



Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems.* 2017.

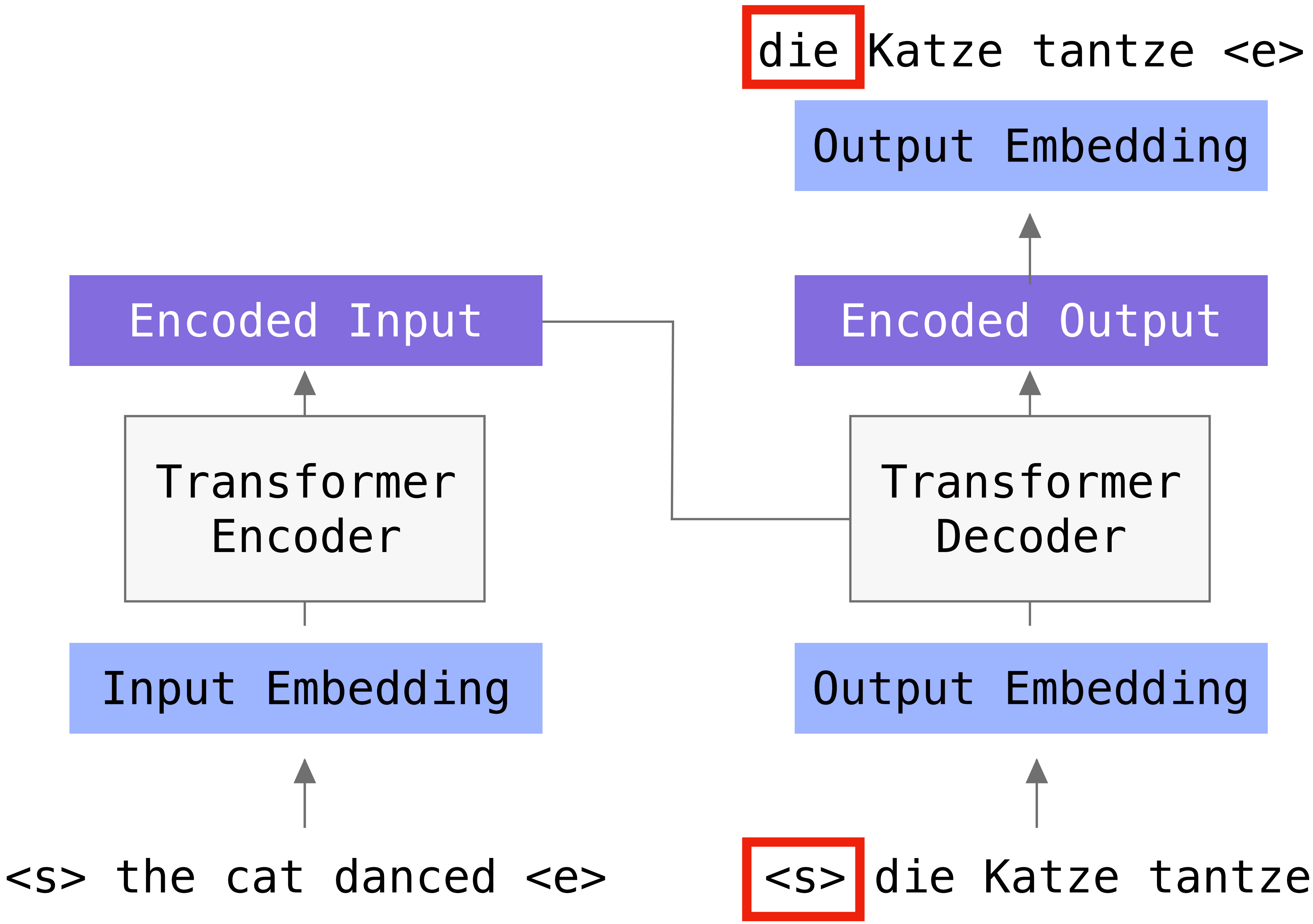
Transformers

Masked Training



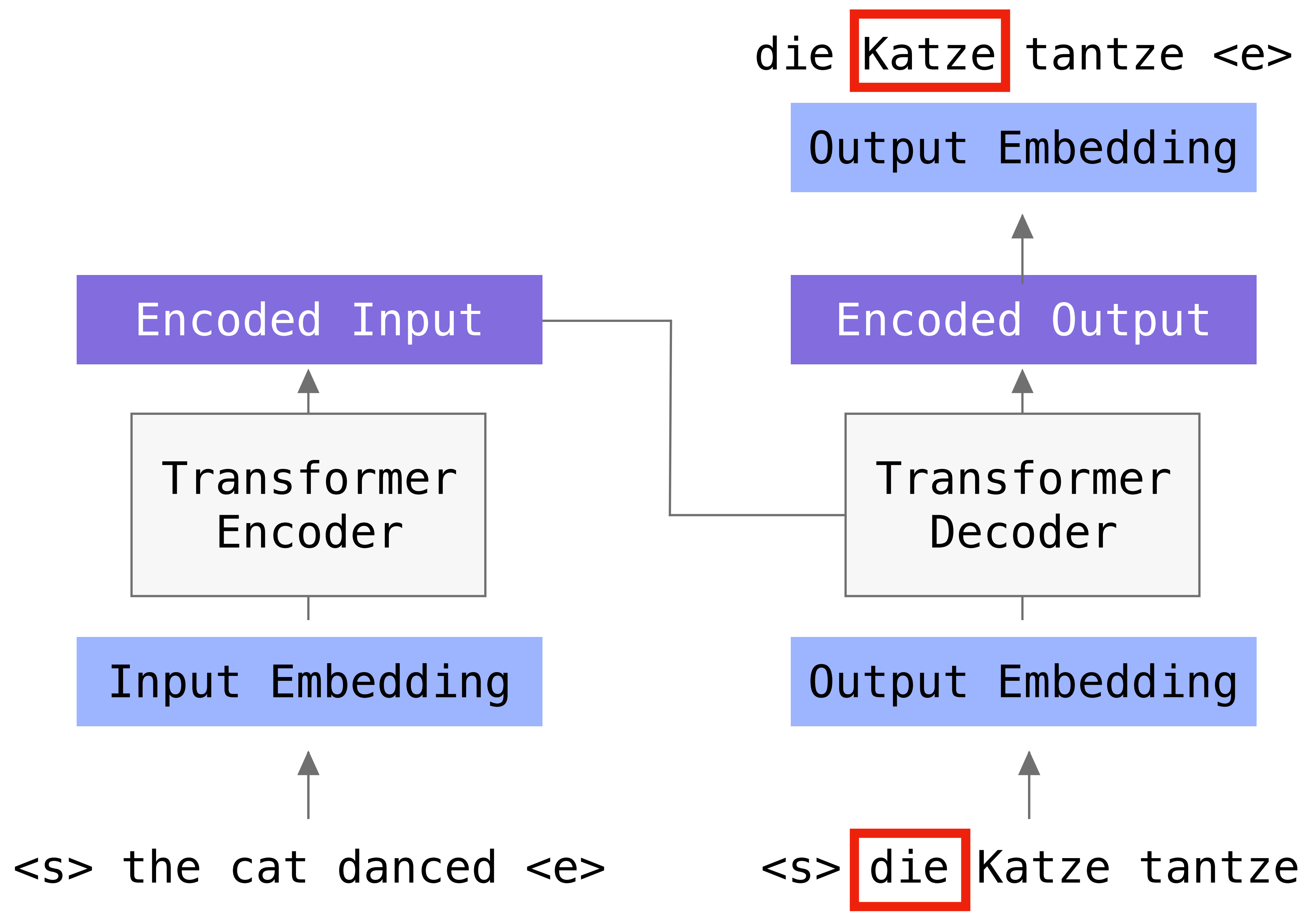
Transformers

Masked Training



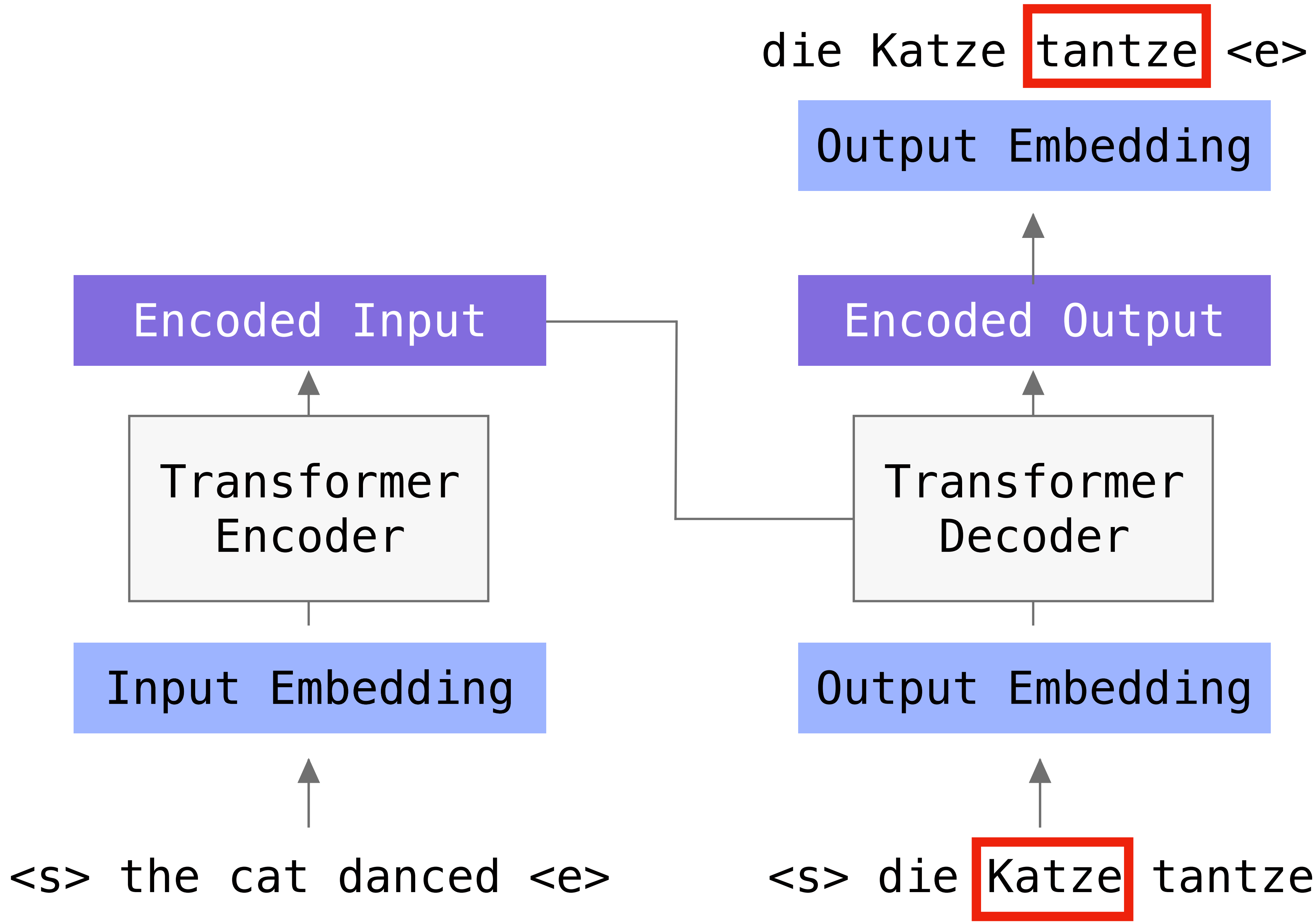
Transformers

Masked Training



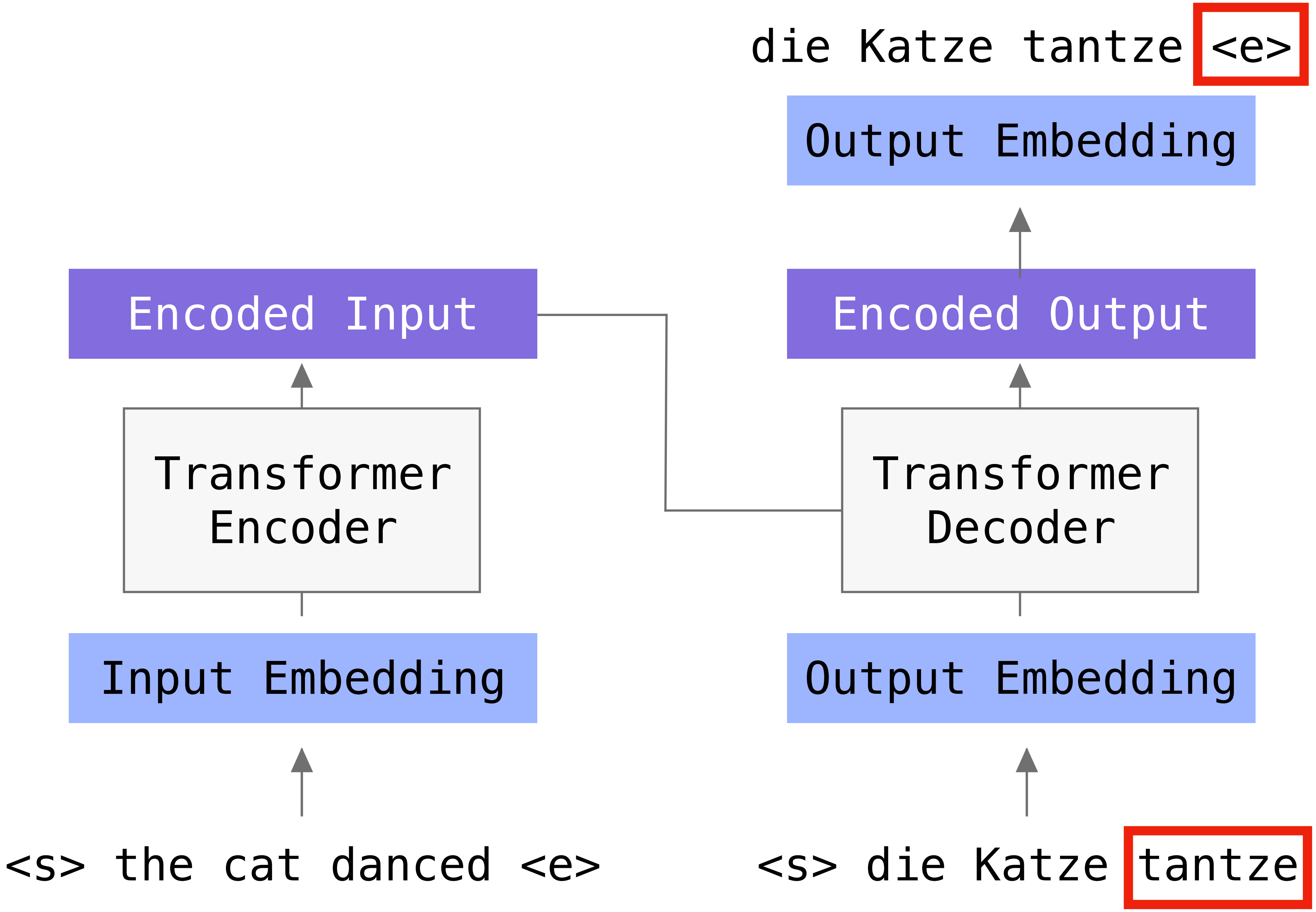
Transformers

Masked Training



Transformers

Masked Training



Transformers

Masked Training

<s>	1	0	0	0
die	1	1	0	0
Katze	1	1	1	0
tantze	1	1	1	1
	<s>	die	Katze	tantze

Transformers

Attention: Working Example

```
def masked-self-attention(Q, mask):
  # Q : N x D_1
  # mask : N x N
  K = V = Q

  scores = Q · K.T            # N x N
  scaled = scores / sqrt(D_1) # N x N
  masked = scores.masked_fill(
          mask == 0, -1e9)
  alpha  = softmax(masked)    # N x N
  out    = alpha · V          # N x d_2
```

```
Q =    [[ 1.0000,  1.0000],   mask =  [[1, 0, 0, 0],
        [-1.0000, -2.5000],            [1, 1, 0, 0],
        [ 4.0000,  3.0000],            [1, 1, 1, 0],
        [ 2.0000, -3.0000]]            [1, 1, 1, 1]])
```

Transformers

Attention: Working Example

```
def masked-self-attention(Q, mask):
  # Q : N x D_1
  # mask : N x N
  K = V = Q

  scores = Q · K.T              # N x N
  scaled = scores / sqrt(D_1) # N x N
  masked = scores.masked_fill(
          mask == 0, -1e9)
  alpha  = softmax(masked)    # N x N
  out    = alpha · V          # N x d_2
```

```
scores = tensor([
        [  2.0000,  -3.5000,   7.0000,  -1.0000],
        [ -3.5000,   7.2500, -11.5000,   5.5000],
        [  7.0000, -11.5000,  25.0000,  -1.0000],
        [ -1.0000,   5.5000,  -1.0000,  13.0000]])
```

Transformers

Attention: Working Example

```
def masked-self-attention(Q, mask):
  # Q : N x D_1
  # mask : N x N
  K = V = Q

  scores = Q · K.T            # N x N
  scaled = scores / sqrt(D_1) # N x N
  masked = scores.masked_fill(
          mask == 0, -1e9)
  alpha  = softmax(masked)    # N x N
  out    = alpha · V          # N x d_2
```

```
scaled = tensor(
       [[ 0.5000, -0.8750,  1.7500, -0.2500],
        [-0.8750,  1.8125, -2.8750,  1.3750],
        [ 1.7500, -2.8750,  6.2500, -0.2500],
        [-0.2500,  1.3750, -0.2500,  3.2500]])
```

Transformers

Attention: Working Example

```
def masked-self-attention(Q, mask):
  # Q : N x D_1
  # mask : N x N
  K = V = Q

  scores = Q · K.T             # N x N
  scaled = scores / sqrt(D_1) # N x N
  masked = scores.masked_fill(
           mask == 0, -1e9)
  alpha  = softmax(masked)    # N x N
  out    = alpha · V          # N x d_2
```

Recall: Each index in this NxM matrix represents the *compatibility* between query n and key m.

```
masked = tensor(
       [[ 0.5000,    -1e9,    -1e9,    -1e9],
        [-0.8750,  1.8125,    -1e9,    -1e9],
        [ 1.7500, -2.8750,  6.2500,    -1e9],
        [-0.2500,  1.3750, -0.2500,  3.2500]])
```

Transformers

Attention: Working Example

```
def masked-self-attention(Q, mask):
  # Q : N x D_1
  # mask : N x N
  K = V = Q

  scores = Q · K.T              # N x N
  scaled = scores / sqrt(D_1) # N x N
  masked = scores.masked_fill(
           mask == 0, -1e9)
  alpha  = softmax(masked)    # N x N
  out    = alpha · V          # N x d_2
```

Thus, as we generate the output for the first query, there is *no* compatibility between it and subsequent queries (and so on).

```
masked = tensor(
       [[ 0.5000,    -1e9,    -1e9,    -1e9],
        [-0.8750,  1.8125,    -1e9,    -1e9],
        [ 1.7500, -2.8750,  6.2500,    -1e9],
        [-0.2500,  1.3750, -0.2500,  3.2500]])
```

Transformers

Attention: Working Example

```
def masked-self-attention(Q, mask):
  # Q : N x D_1
  # mask : N x N
  K = V = Q

  scores = Q · K.T             # N x N
  scaled = scores / sqrt(D_1) # N x N
  masked = scores.masked_fill(
          mask == 0, -1e9)
  alpha  = softmax(masked)    # N x N
  out    = alpha · V          # N x d_2
```

```
alpha = tensor([[
         [1.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 1.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 1.0000, 0.0000],
         [0.0000, 0.0006, 0.0000, 0.9994]]])
```

Transformers

Attention: Working Example

```
def masked-self-attention(Q, mask):
  # Q : N x D_1
  # mask : N x N
  K = V = Q

  scores = Q · K.T            # N x N
  scaled = scores / sqrt(D_1) # N x N
  masked = scores.masked_fill(
          mask == 0, -1e9)
  alpha  = softmax(masked)    # N x N
  out    = alpha · V          # N x d_2
```

```
out = [
  [ 1.         1.        ]
  [-0.9999571 -2.499925  ]
  [ 4.         3.        ]
  [ 1.9983424 -2.9997153]]
```

Transformers

Attention: Working Example

```
def masked-self-attention(Q, mask):
  # Q : N x D_1
  # mask : N x N
  K = V = Q

  scores = Q · K.T            # N x N
  scaled = scores / sqrt(D_1) # N x N
  masked = scores.masked_fill(
           mask == 0, -1e9)
  alpha  = softmax(masked)    # N x N
  out    = alpha · V          # N x d_2
```

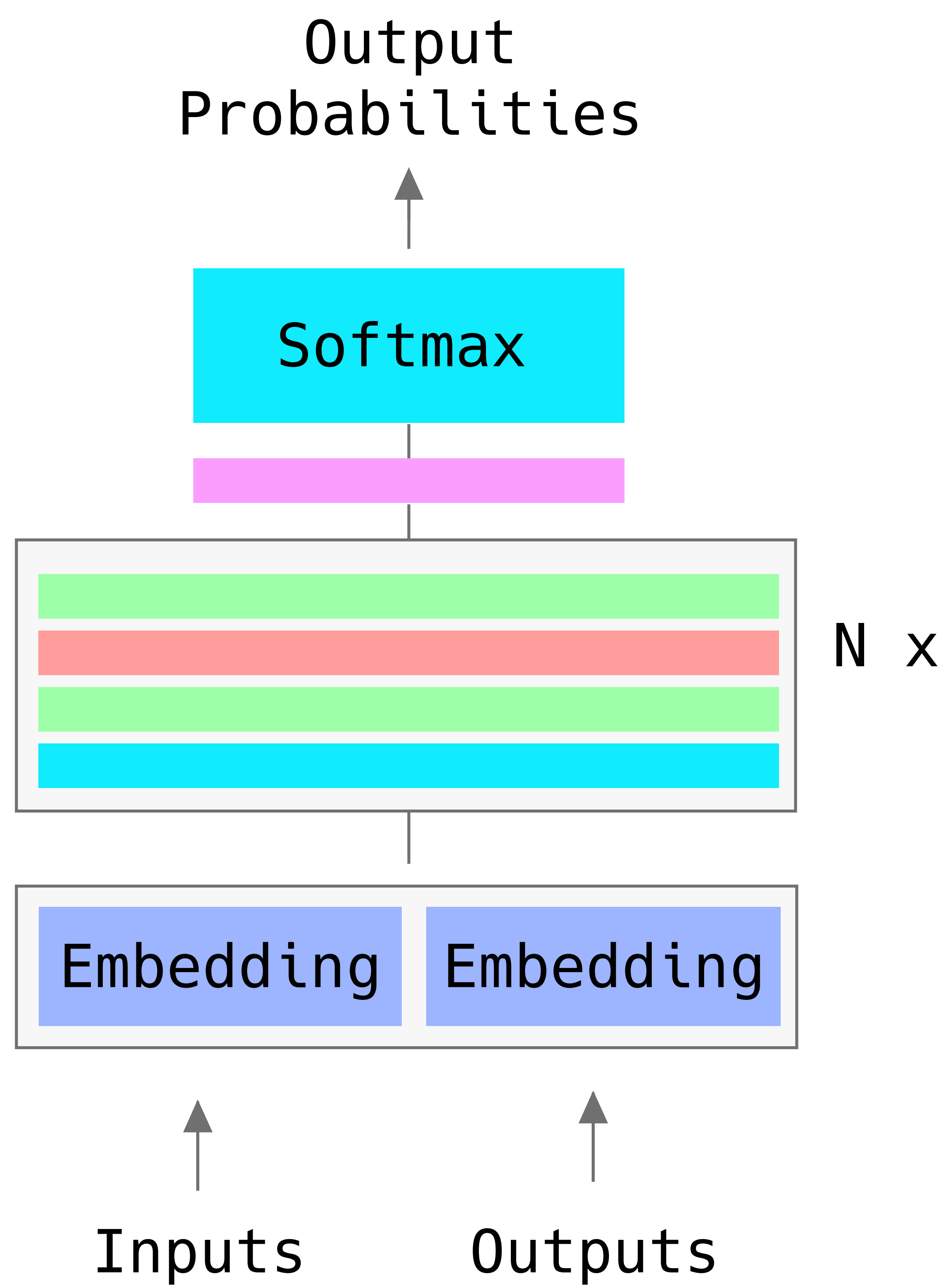
```
out = [
  [ 1.         1.        ]
  [-0.9999571 -2.499925 ]
  [ 4.         3.        ]
  [ 1.9983424 -2.9997153]]
```

```
Q =    [[ 1.0000,  1.0000],
        [-1.0000, -2.5000],
        [ 4.0000,  3.0000],
        [ 2.0000, -3.0000]]
```

Transformers

T-D

“We also suspect that for monolingual text-to-text tasks redundant information is re-learned about language in the encoder and decoder.”



Liu, Peter J., et al. "Generating wikipedia by summarizing long sequences." *arXiv preprint arXiv:1801.10198* (2018).

Transformers

GPT (1)

- This is the first model that uses a transformer uses LM as pre-training for future use.
- T-D blocks are used as encoders, and then a single weight matrix is learned on top for fine-tuned tasks (at most 3 epochs of training).

Radford, Alec, et al. "Improving language understanding by generative pre-training." URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/languageunsupervised/language understanding paper. pdf (2018).

https://blog.openai.com/language-unsupervised/

Transformers

GPT (1)

- They continue to use the LM as an auxiliary loss which speeds up convergence.
- They demonstrate zero-shot capacity on many simple tasks (sentiment analysis).

Radford, Alec, et al. "Improving language understanding by generative pre-training." URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/languageunsupervised/language understanding paper. pdf (2018).

https://blog.openai.com/language-unsupervised/

Transformers

GPT (2)

- There are minor modifications to the transformer block, but it's basically just the T-D.
- Predominately, they only test on LM; they get SOTA on 7/8 datasets with zero-shot evaluation.

Radford, Alec, et al. "Language Models are Unsupervised Multitask Learners" URL https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. pdf (2019).

https://blog.openai.com/better-language-models/

Transformers

GPT (2)

- They demonstrated some capacity for zero-shot learning in other tasks (including reading comprehension and question answering). Both results were impressive, but not close to SOTA.

Radford, Alec, et al. "Language Models are Unsupervised Multitask Learners" URL https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. pdf (2019).

https://blog.openai.com/better-language-models/

Transformers

BERT

- Similar to GPT, but it is stacked T-E.
- It used two pre-training tasks.

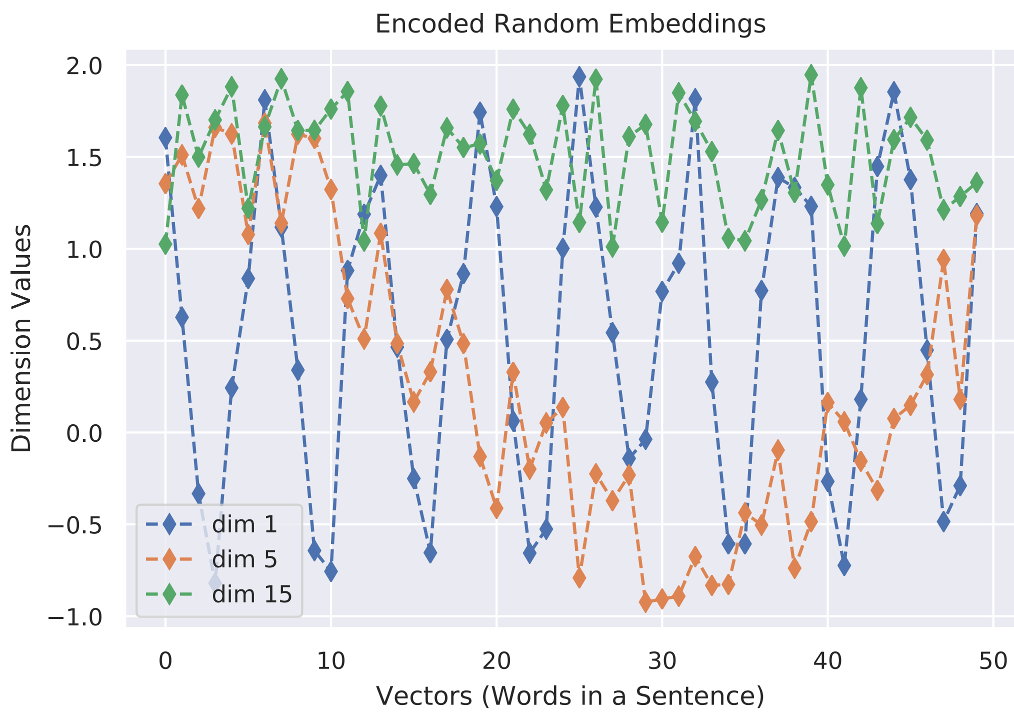
BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding Devlin, J., Chang, M., Lee, K. and Toutanova, K., 2018. arXiv preprint arXiv:1810.04805.

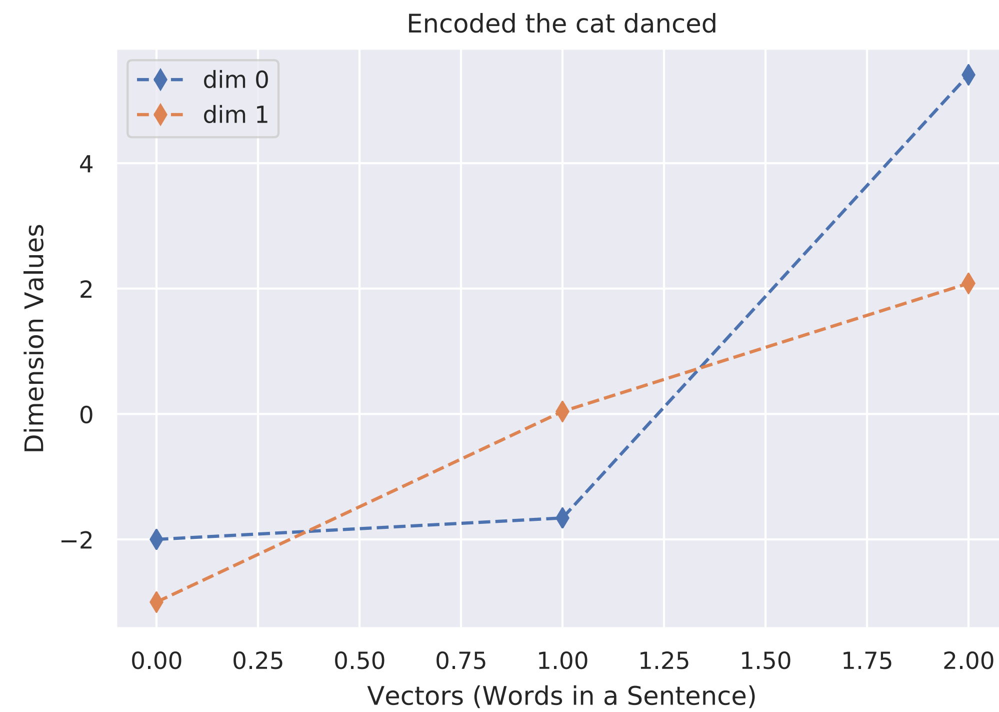
Transformers

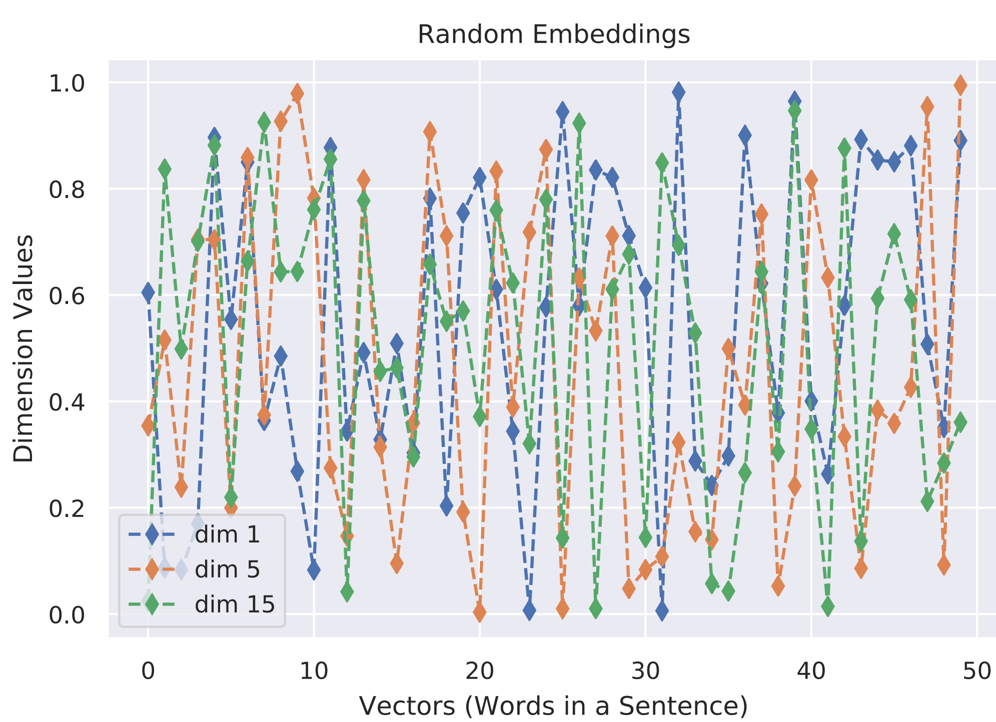
BERT

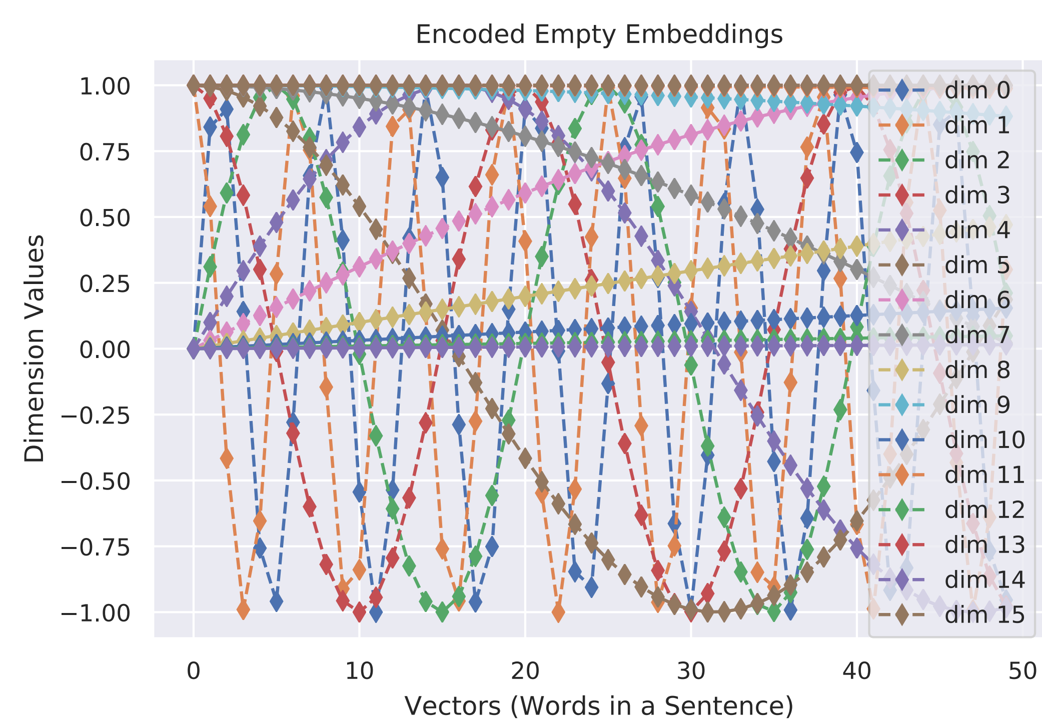
- The task is like LM, but instead the model has to predict words which were randomly masked.
- Like skip-thought vectors, they train the model to predict if two sentences should follow one another.

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding Devlin, J., Chang, M., Lee, K. and Toutanova, K., 2018. arXiv preprint arXiv:1810.04805.


Encoded Random Embeddings
Dimension Values
Vectors (Words in a Sentence)
dim 1
dim 5
dim 15


Encoded the cat danced
Dimension Values
Vectors (Words in a Sentence)
dim 0
dim 1


Random Embeddings
Dimension Values
Vectors (Words in a Sentence)
dim 1
dim 5
dim 15


Encoded Empty Embeddings
Dimension Values
Vectors (Words in a Sentence)
dim 0
dim 1
dim 2
dim 3
dim 4
dim 5
dim 6
dim 7
dim 8
dim 9
dim 10
dim 11
dim 12
dim 13
dim 14
dim 15

Transformers

RNN Comparison

Transformers	RNNs
Parallel across inputs.	Sequential across inputs.
Constant path length from input to output.	Path length from output symbol to input depends on the length of the sequence, making it difficult to learn long range dependencies.
Few parameters.	