Relation Network Various Attention mechanisms

모두의 연구소

- 2018. 10. 29 -

Attention Mechanism
&
Self-attention

Seq2seq & its limit

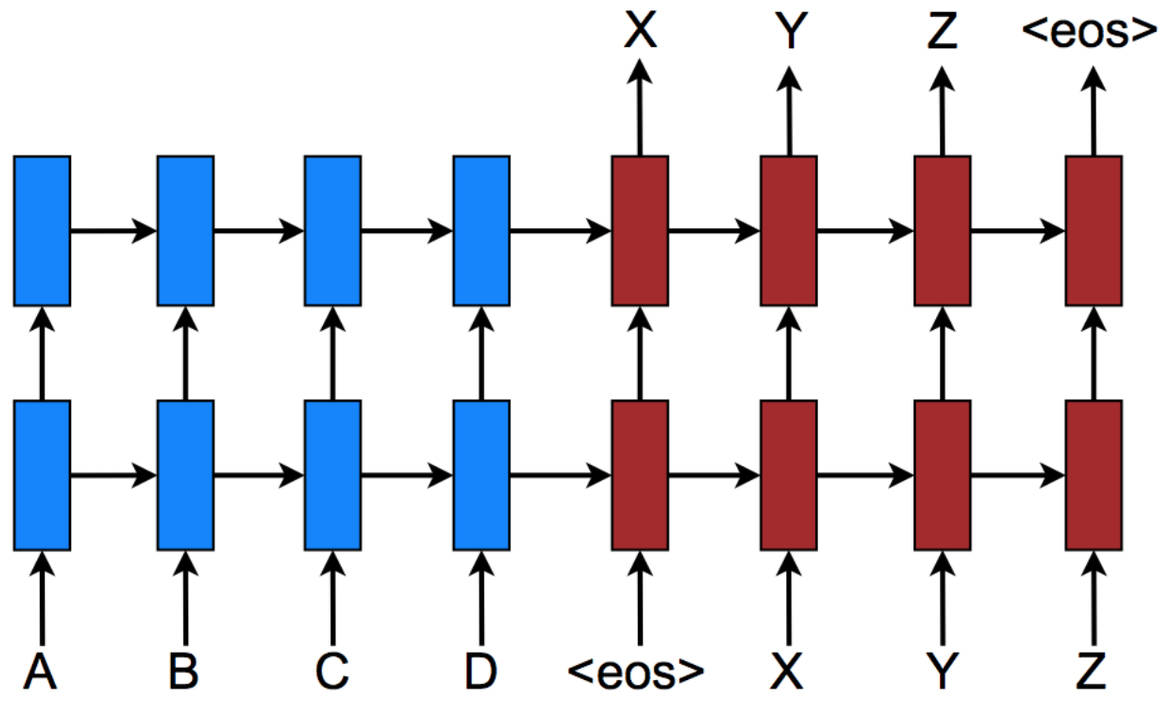


Figure 1: **Neural machine translation** – a stacking recurrent architecture for translating a source sequence A B C D into a target sequence X Y Z. Here, <eos> marks the end of a sentence.

When the source sequence is too long and contains multiple information-rich phrases apart from each other

What if we had a mechanism to allow the decoder to selectively (dynamically) focus on the information-rich phrases in the source sequence?

Attention mechanism

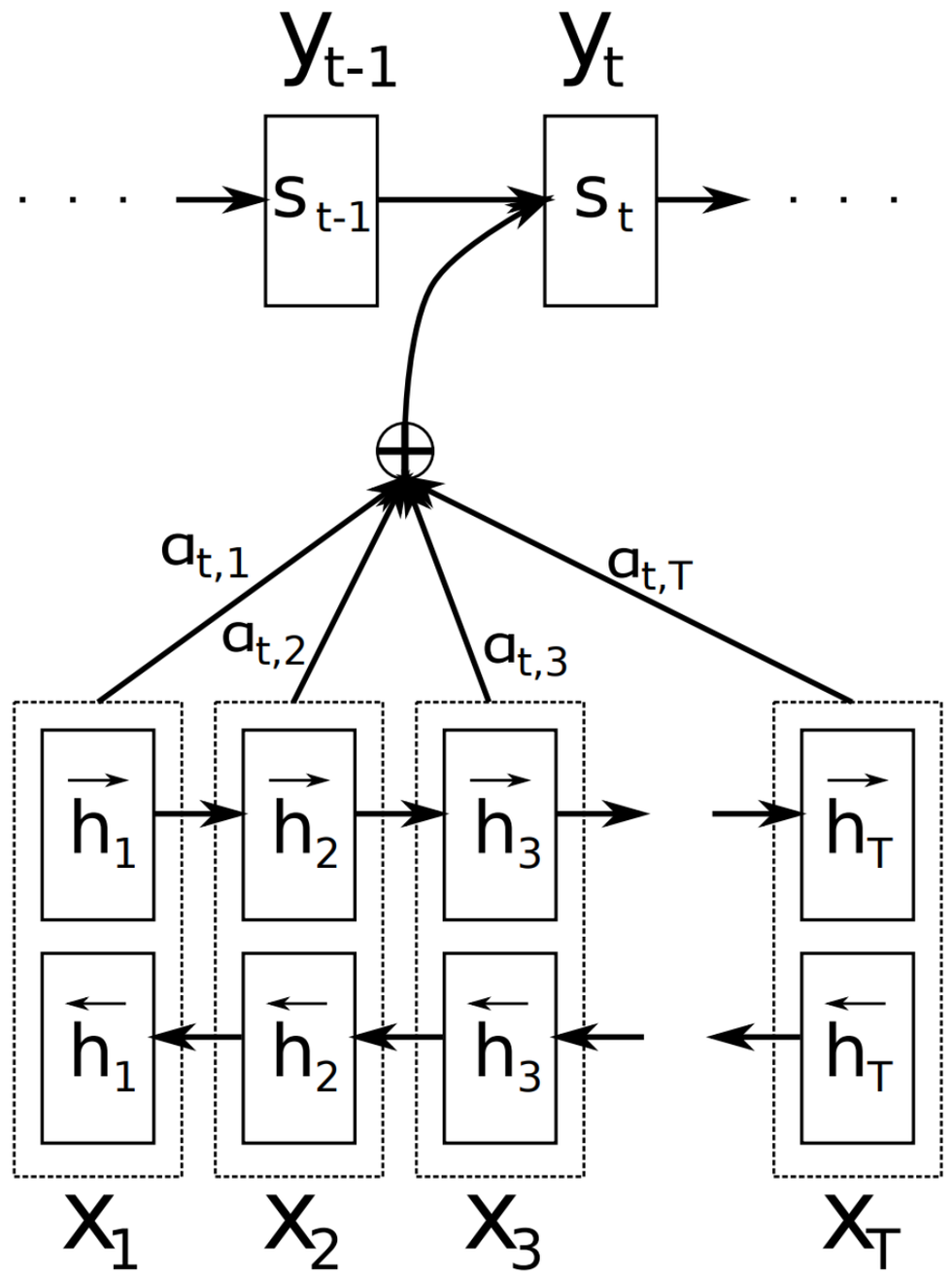


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

Attention mechanism can do

Long term memories - attending to memories

- Dealing with gradient vanishing problem

Exceeding limitations of a global representation

- Attending/focusing to smaller parts of data
- patches in images - words or phrases in sentences

http://download.mpi-inf.mpg.de/d2/mmalinow-slides/attention_networks.pdf

Attention mechanism can do

Decoupling representation from a problem

- Different problems required different sizes of representations
- LSTM with longer sentences requires larger vectors

Overcoming computational limits for visual data

- Focusing only on the parts of images
- Scalability independent of the size of images

Adds some interpretability to the models (error inspection)

Attention mechanism

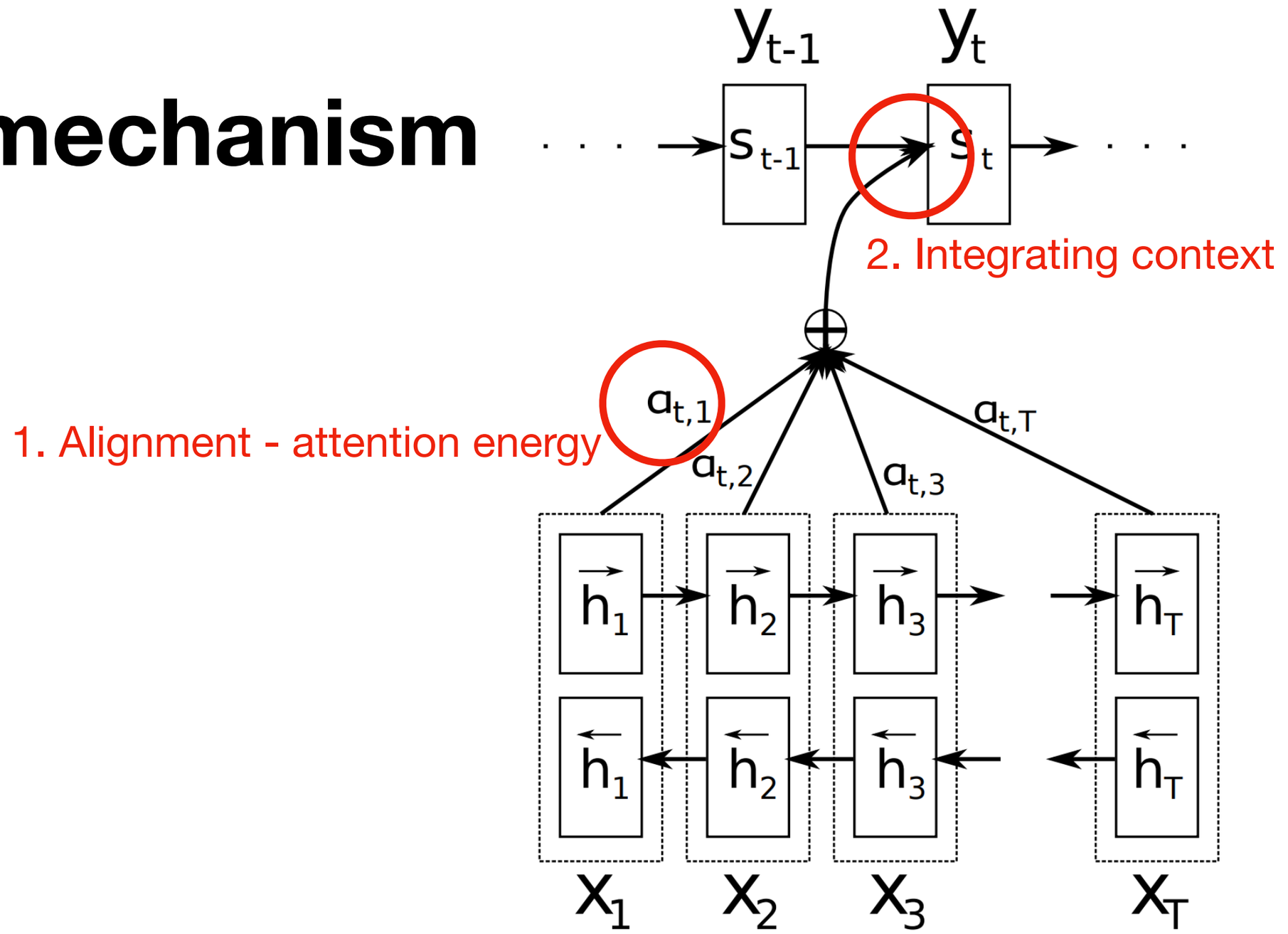


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

How to implement Attention

Alignment

Neural Machine Translation by Jointly Learning to Align and Translate
(https://arxiv.org/pdf/1409.0473.pdf)

A.1.2 Alignment Model

The alignment model should be designed considering that the model needs to be evaluated $T_x \times T_y$ times for each sentence pair of lengths T_x and T_y. In order to reduce computation, we use a single-layer multilayer perceptron such that

$$a(s_{i-1}, h_j) = v_a^\top \tanh\left(W_a s_{i-1} + U_a h_j\right),$$

where $W_a \in \mathbb{R}^{n \times n}, U_a \in \mathbb{R}^{n \times 2n}$ and $v_a \in \mathbb{R}^n$ are the weight matrices. Since $U_a h_j$ does not depend on i, we can pre-compute it in advance to minimize the computational cost.

Alignment

These are basically ***unnormalized scores*** of alignment between decoder state ***s*** and the hidden states H.

The mapping from $(\{h_i\}_{i\in[0,T]}, s_j)$ to the attention energies is known as the *alignment* model.

Effective Approaches to Attention-based Neural Machine Translation (https://arxiv.org/pdf/1508.04025.pdf)

The idea of a global attentional model is to consider all the hidden states of the encoder when deriving the context vector c_t. In this model type, a variable-length alignment vector $\boldsymbol{a}_t$, whose size equals the number of time steps on the source side, is derived by comparing the current target hidden state $\boldsymbol{h}_t$ with each source hidden state $\bar{\boldsymbol{h}}_s$:

$$\begin{aligned} \boldsymbol{a}_t(s) &= \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \qquad (7) \\ &= \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \end{aligned}$$

Alignment models

Here, score is referred as a *content-based* function for which we consider three different alternatives:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W_a} \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_a}[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

Multiplicative model

Additive model

Effective Approaches to Attention-based Neural Machine Translation (https://arxiv.org/pdf/1508.04025.pdf)

Alignment models

Here, score is referred as a *content-based* function for which we consider three different alternatives:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W_a} \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_a}[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

Multiplicative model

Additive model

Equivalent to

$$v_a^T \tanh\left(\mathbf{U}_a \mathbf{h}_t + \mathbf{W}_a \bar{\mathbf{h}}_s\right)$$

Effective Approaches to Attention-based Neural Machine Translation (https://arxiv.org/pdf/1508.04025.pdf)

Alignment models

Multiplicative Models

$$score(h_i, s_j) = \begin{cases} h_i^T s_j & dot \\ h_t^T W_a s_j & general \end{cases}$$

Additive Models

$$score(h_i, s_j) = \begin{cases} v_a^T \tanh(u_a h_i + W_a s_j) & linear \\ v_a^T \tanh(W_a [h_i; s_j]) & general \end{cases}$$

In general, the performance of multiplicative and additive functions are similar but the multiplicative function is faster and more space-efficient.

Integrating context

Specifically, given the target hidden state $\boldsymbol{h}_t$ and the source-side context vector $\boldsymbol{c}_t$, we employ a simple concatenation layer to combine the information from both vectors to produce an attentional hidden state as follows:

$$\tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W_c}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \quad (5)$$

The attentional vector $\tilde{\boldsymbol{h}}_t$ is then fed through the softmax layer to produce the predictive distribution formulated as:

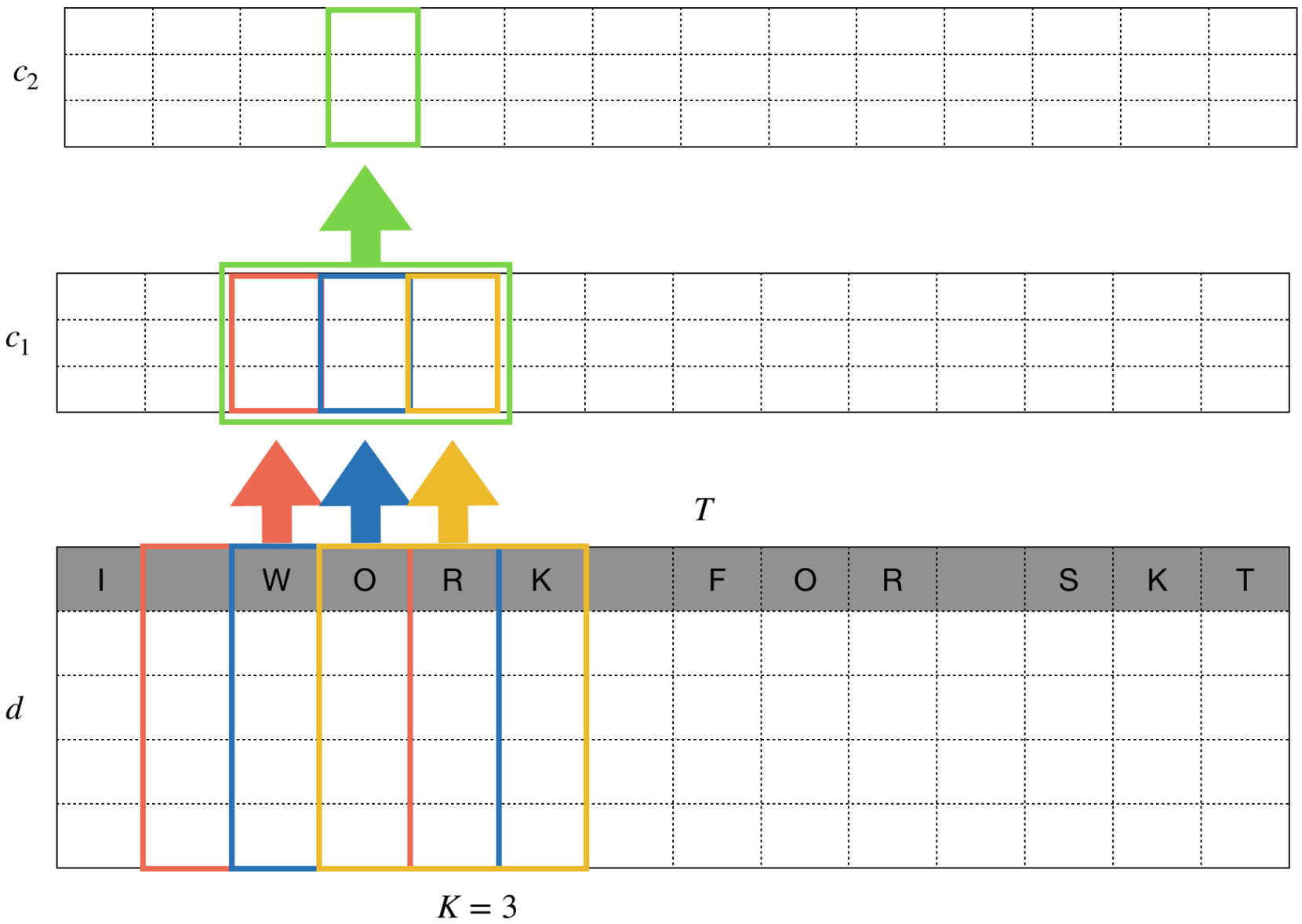
$$p(y_t|y_{<t}, x) = \text{softmax}(\boldsymbol{W_s}\tilde{\boldsymbol{h}}_t) \quad (6)$$

We now detail how each model type computes the source-side context vector $\boldsymbol{c}_t$.

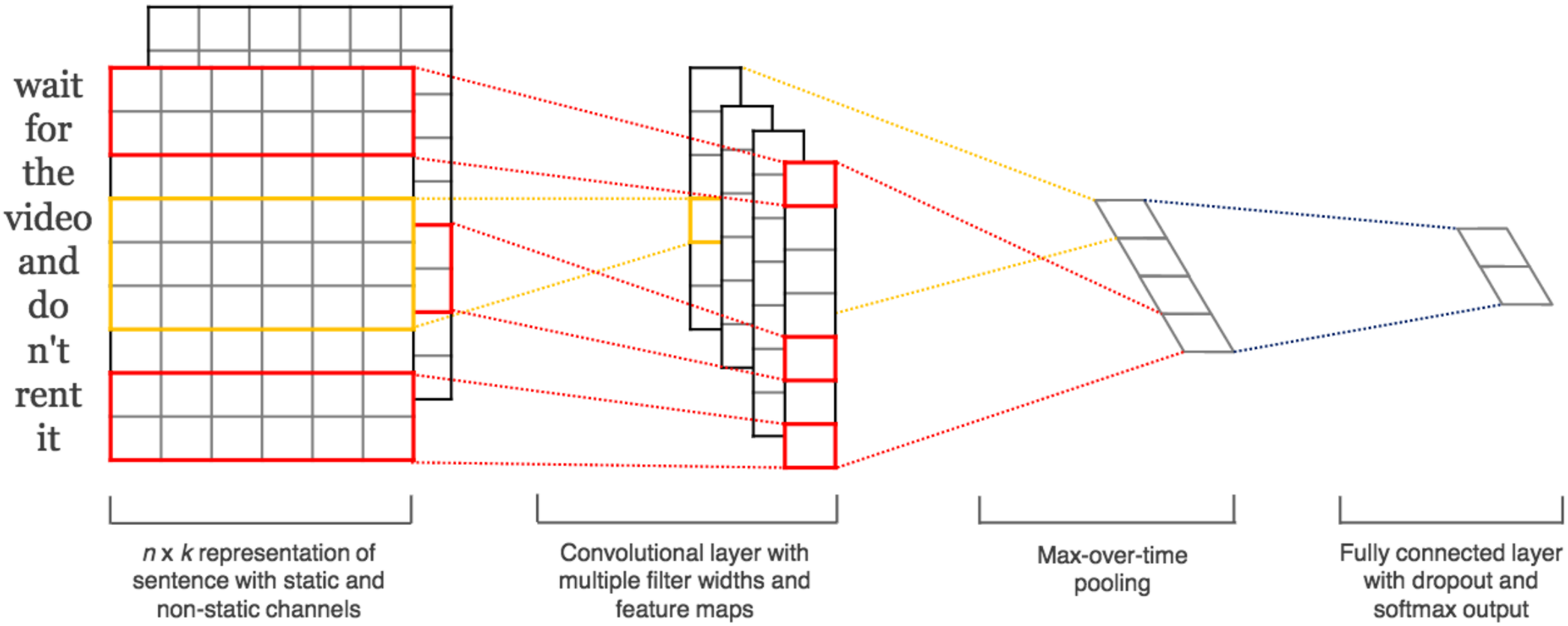
Self Attention

N-GRAM & CNN

***n*-gram is a contiguous sequence of *n* items from a given sample of text**



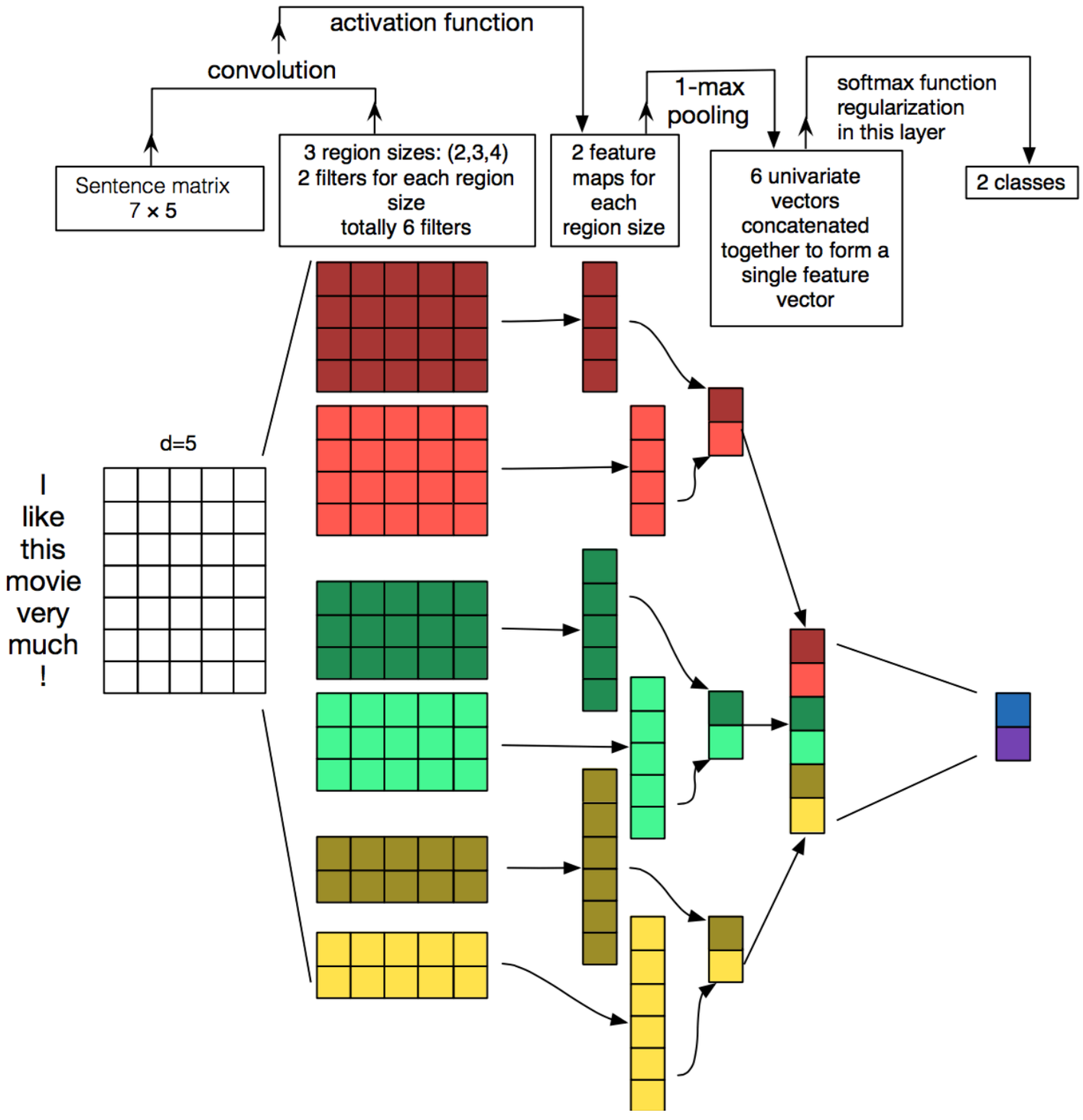
CNN for NLP



Yoon Kim (2014)

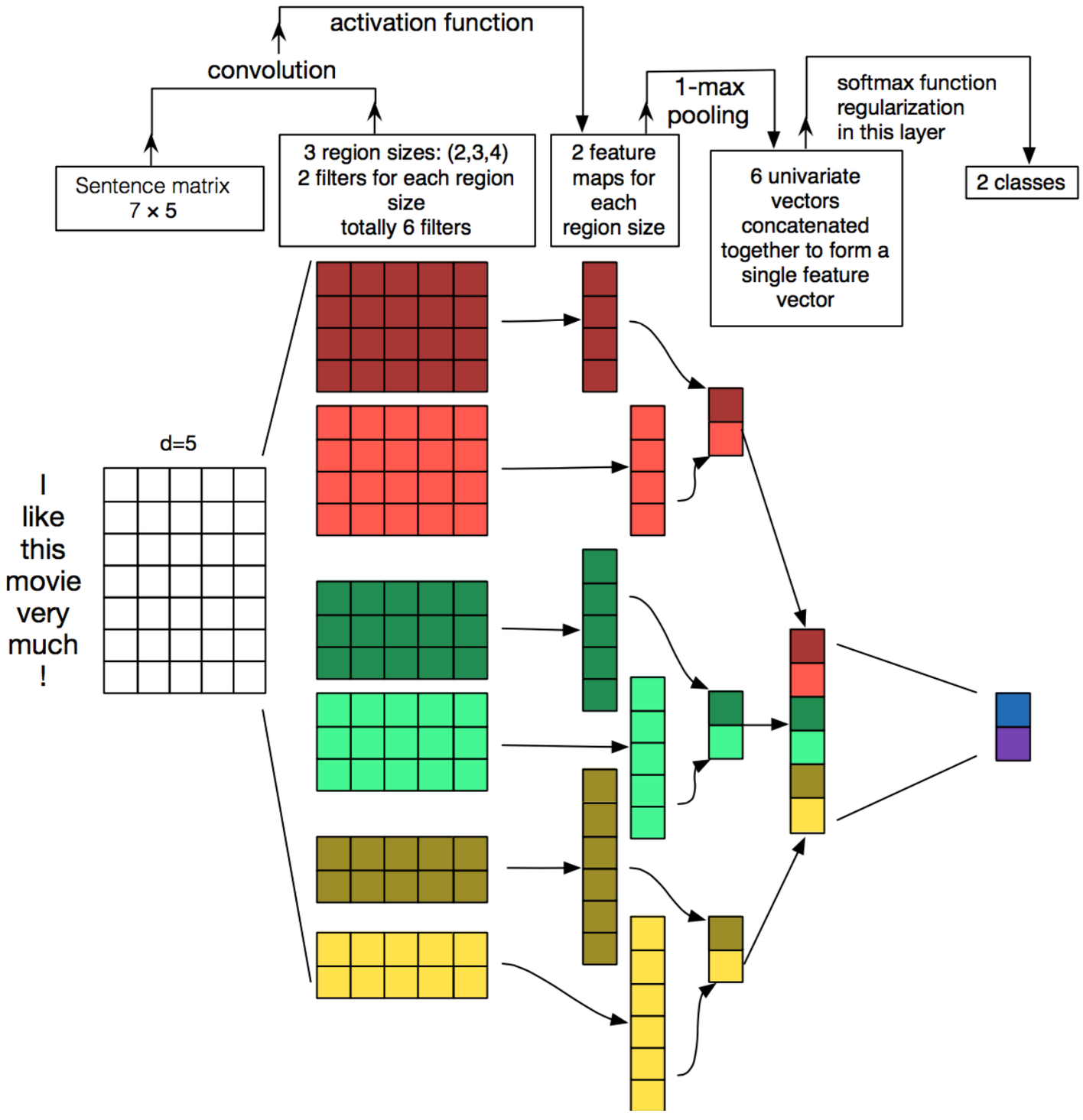
https://arxiv.org/abs/1408.5882

CNN for NLP



[A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification]

CNN for NLP

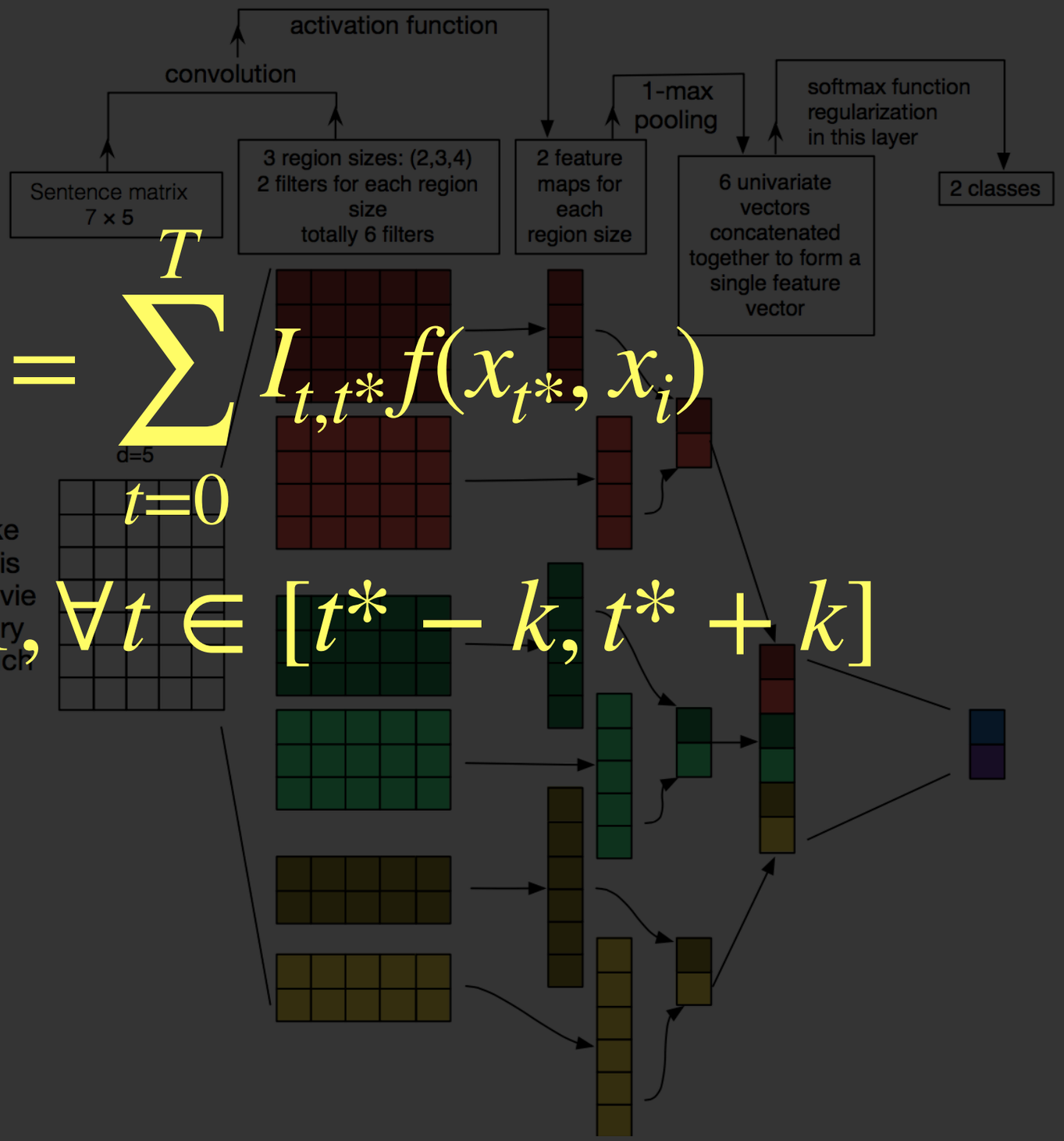


Wants to increase Receptive fields?

Dilated convolution (wavenet)

[A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification]

CNN for NLP



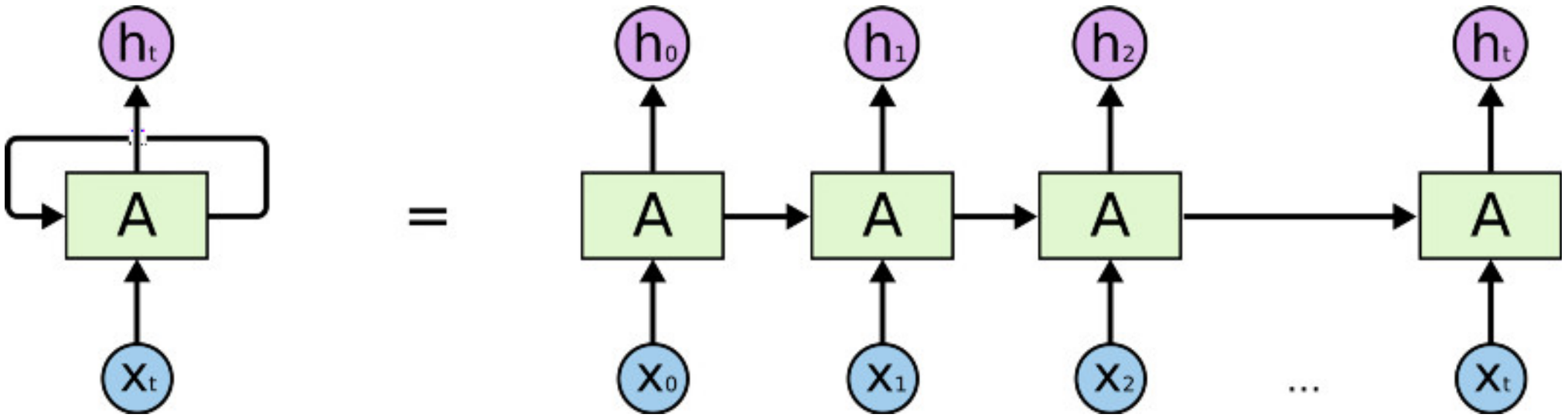
$$h_{t*} = \sum_{t=0}^{T} I_{t,t*} f(x_{t*}, x_i)$$

$$I_{t,t*} = 1, \forall t \in [t* - k, t* + k]$$

A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification

RNN for NLP

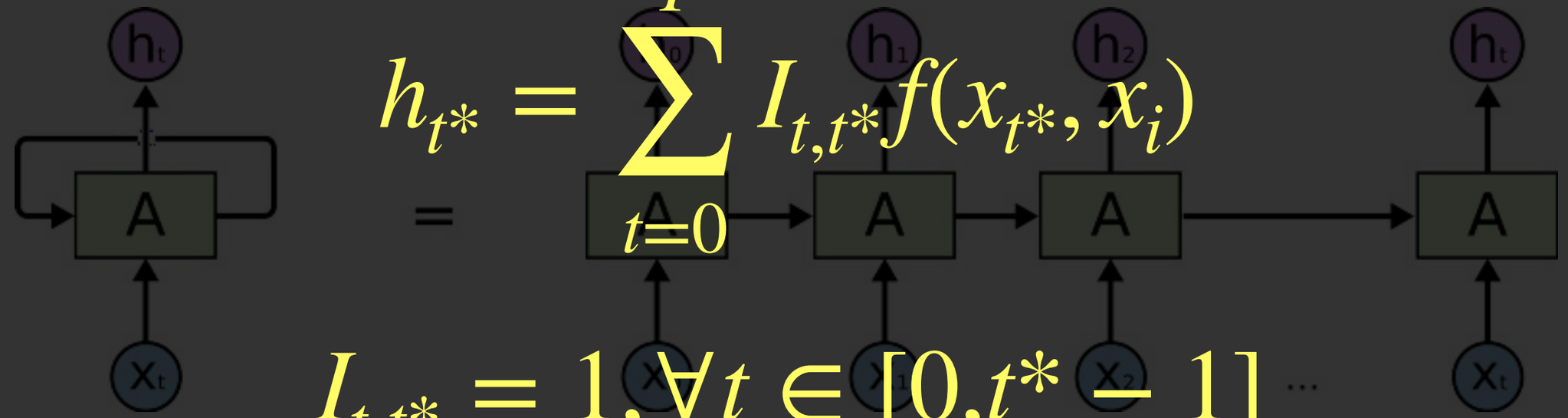
Unfolding **Computational Graphs**



$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

RNN for NLP

Unfolding **Computational Graphs**



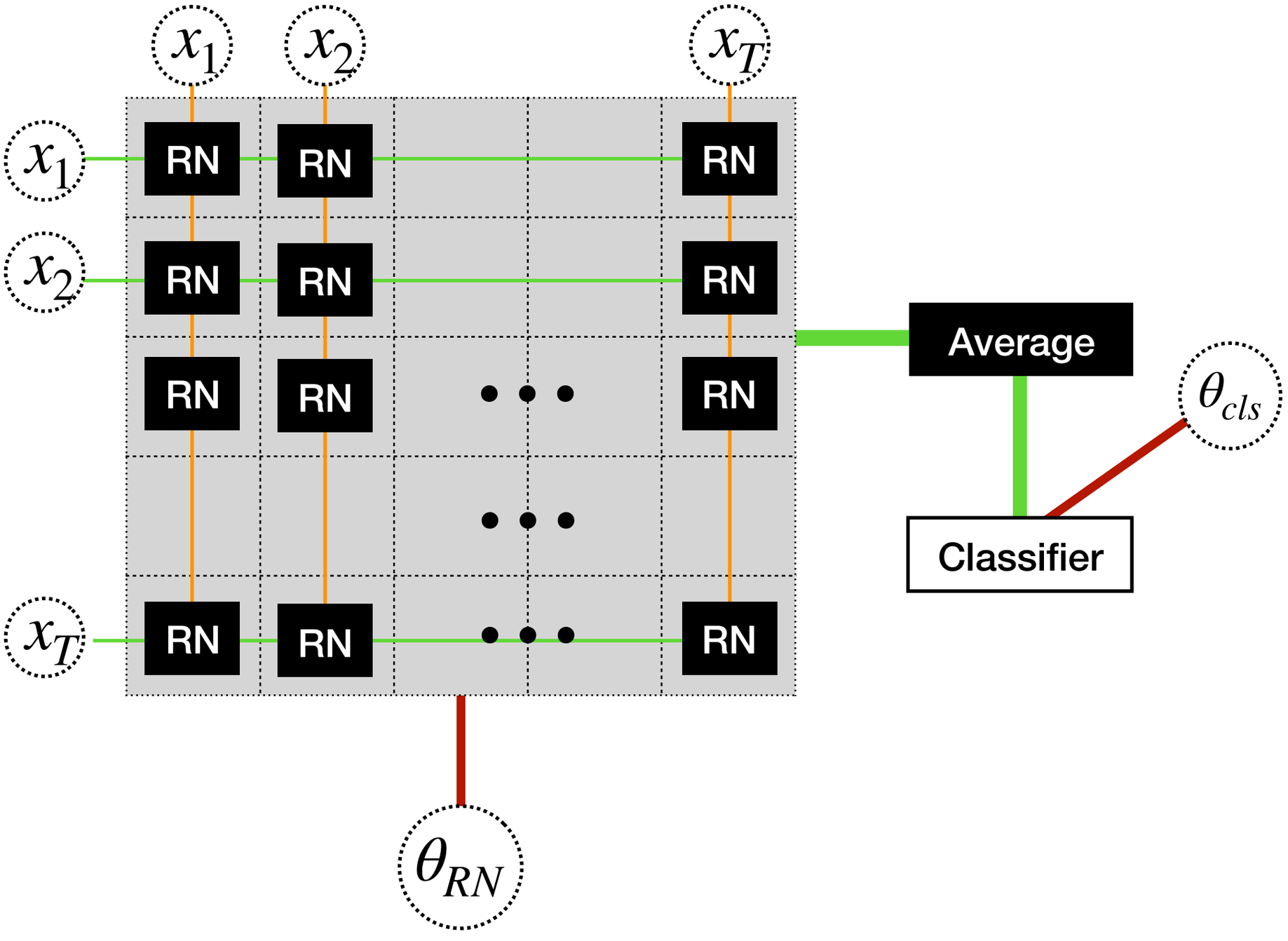
$$h_{t*} = \sum_{t=0}^{T} I_{t,t*} f(x_{t*}, x_i)$$

$$I_{t,t*} = 1, \forall t \in [0, t* - 1]$$

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

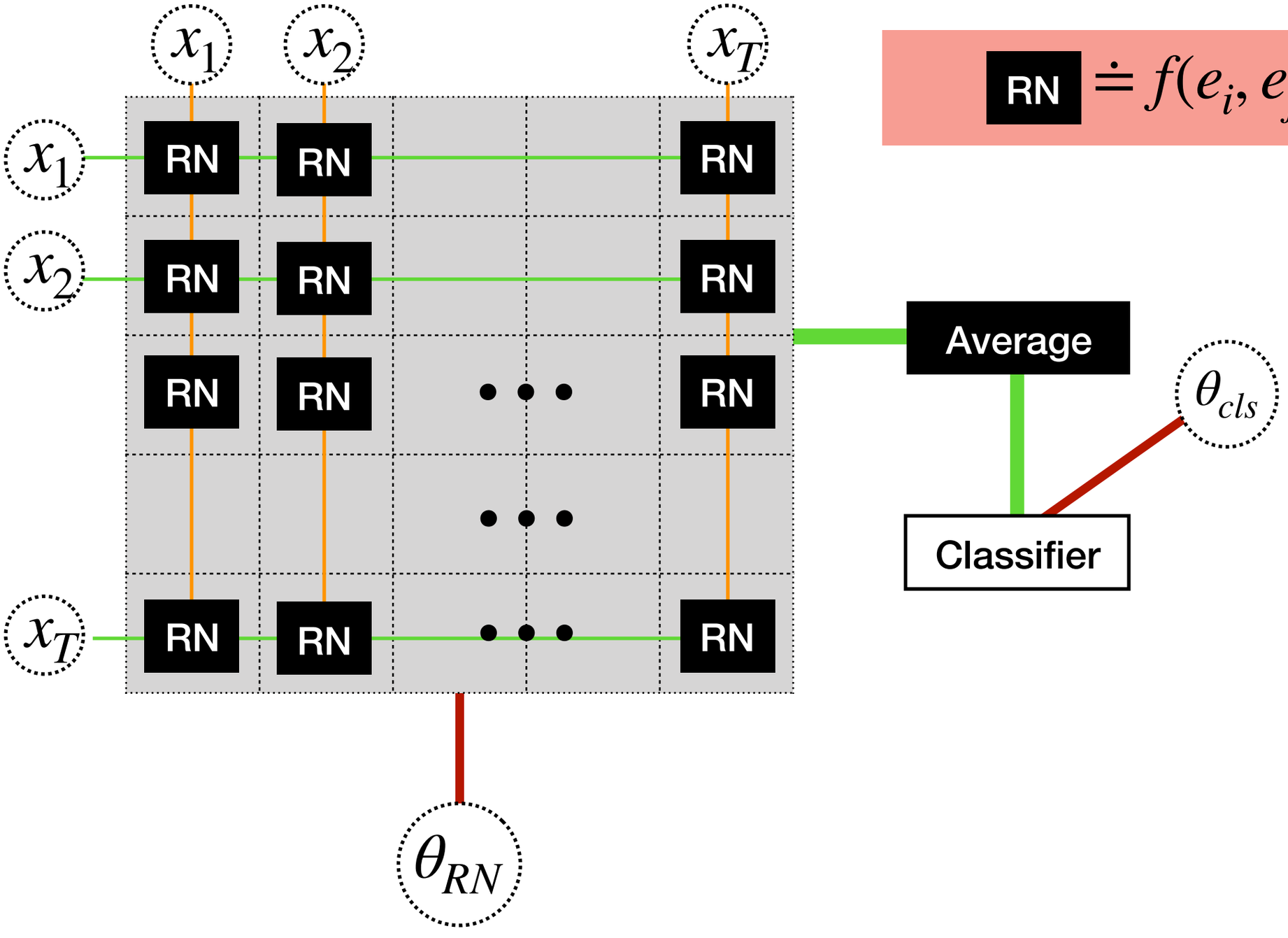
Relation Network (Skip bigram)

To summarize sentence based on pairwise relationship



Relation Network (Skip bigram)

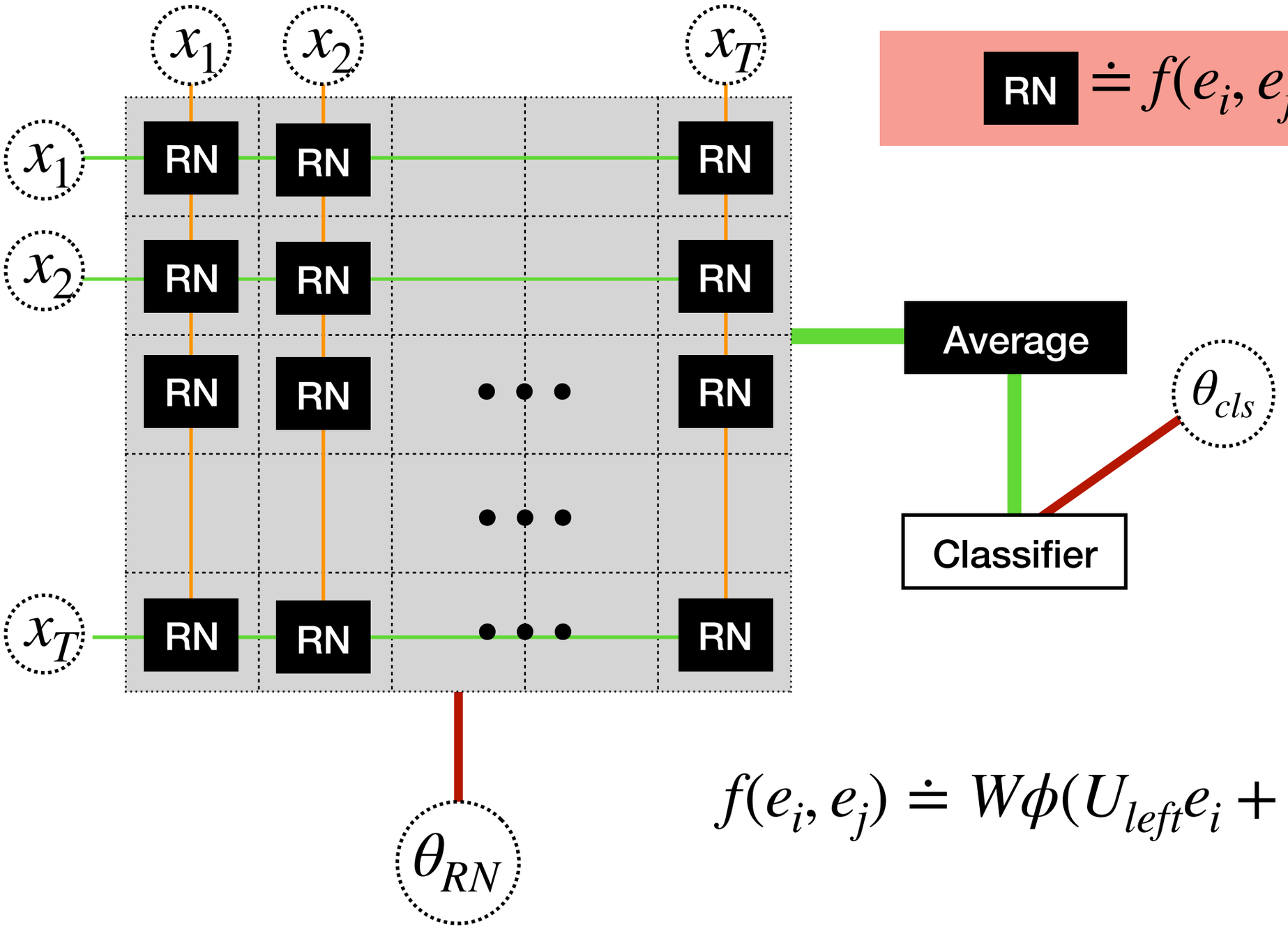
To summarize sentence based on pairwise relationship



$$\text{RN} \doteq f(e_i, e_j)$$

Relation Network (Skip bigram)

To summarize sentence based on pairwise relationship

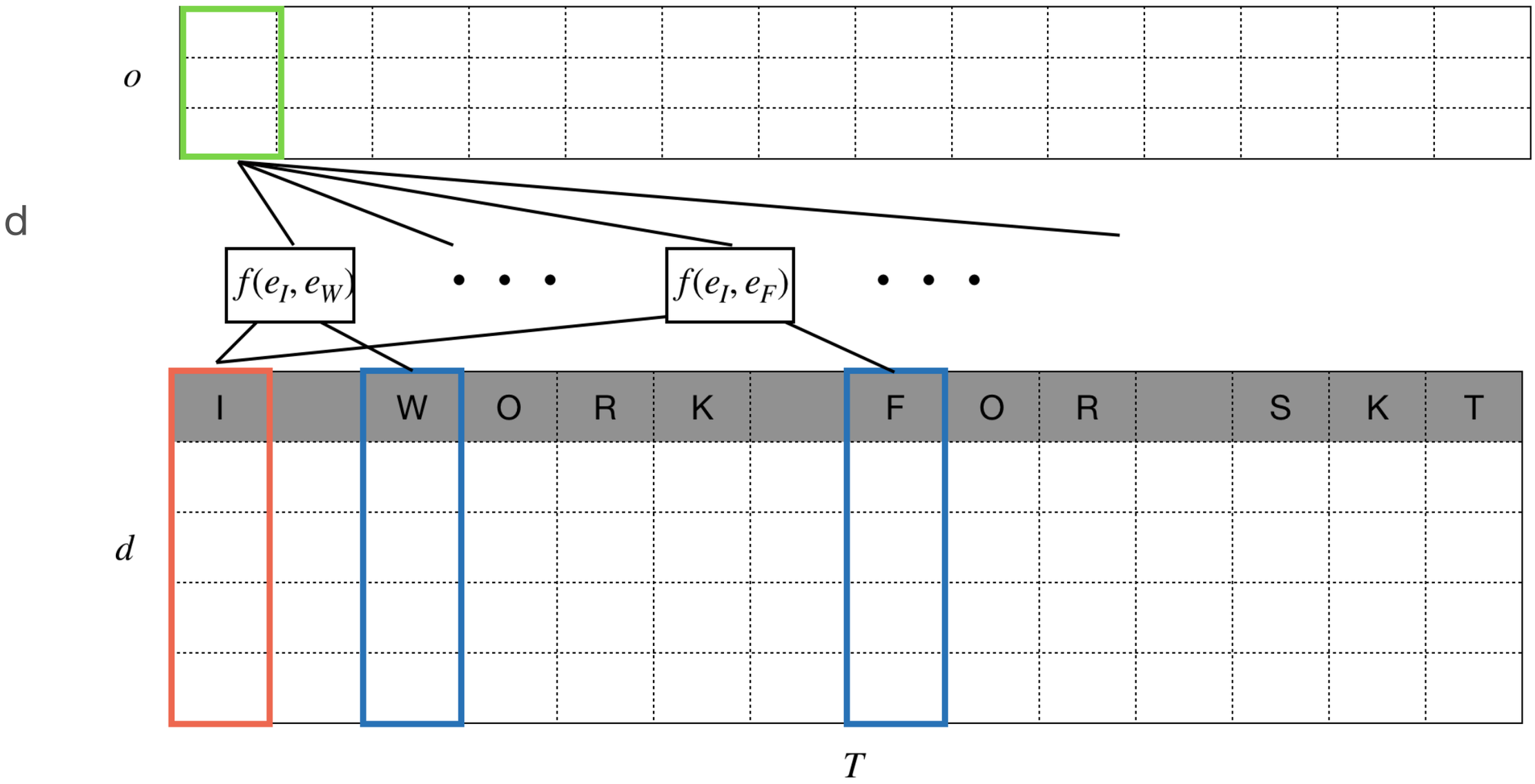


$$\text{RN} \doteq f(e_i, e_j)$$

$$f(e_i, e_j) \doteq W\phi(U_{left}e_i + U_{right}e_j)$$

Skip bigram - Relation Net

A pair of words may give us more clear information about the sentence.
ex) ‘like’ in ‘I like’ vs ‘like’ in ‘like this’

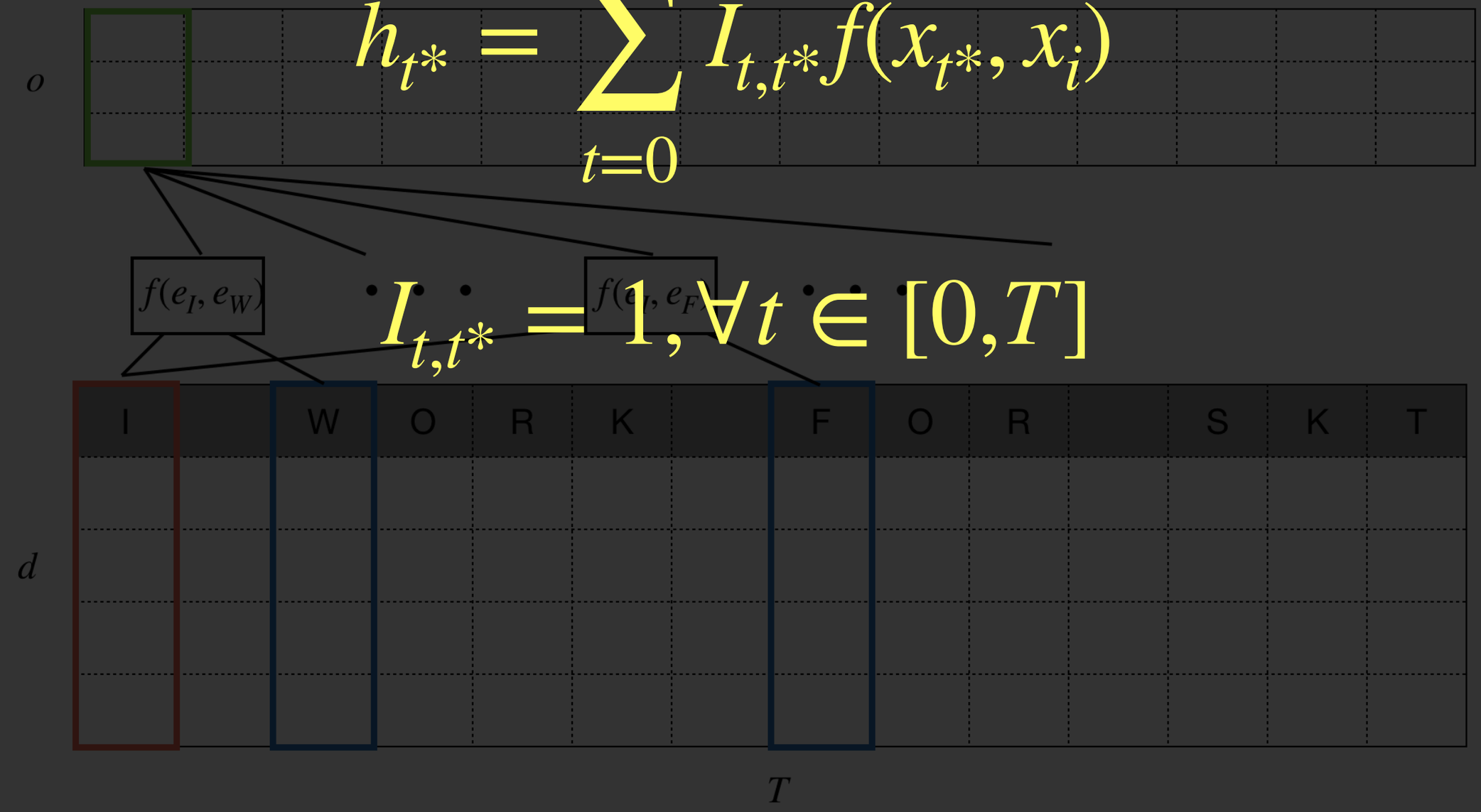


Skip bigram - Relation Net

A pair of words may give us more clear information about the sentence.
ex)'like' in 'I like' vs 'like' in 'like this'

$$h_{t*} = \sum^{T} I_{t,t*} f(x_{t*}, x_i)$$

$$I_{t,t*} = 1, \forall t \in [0, T]$$

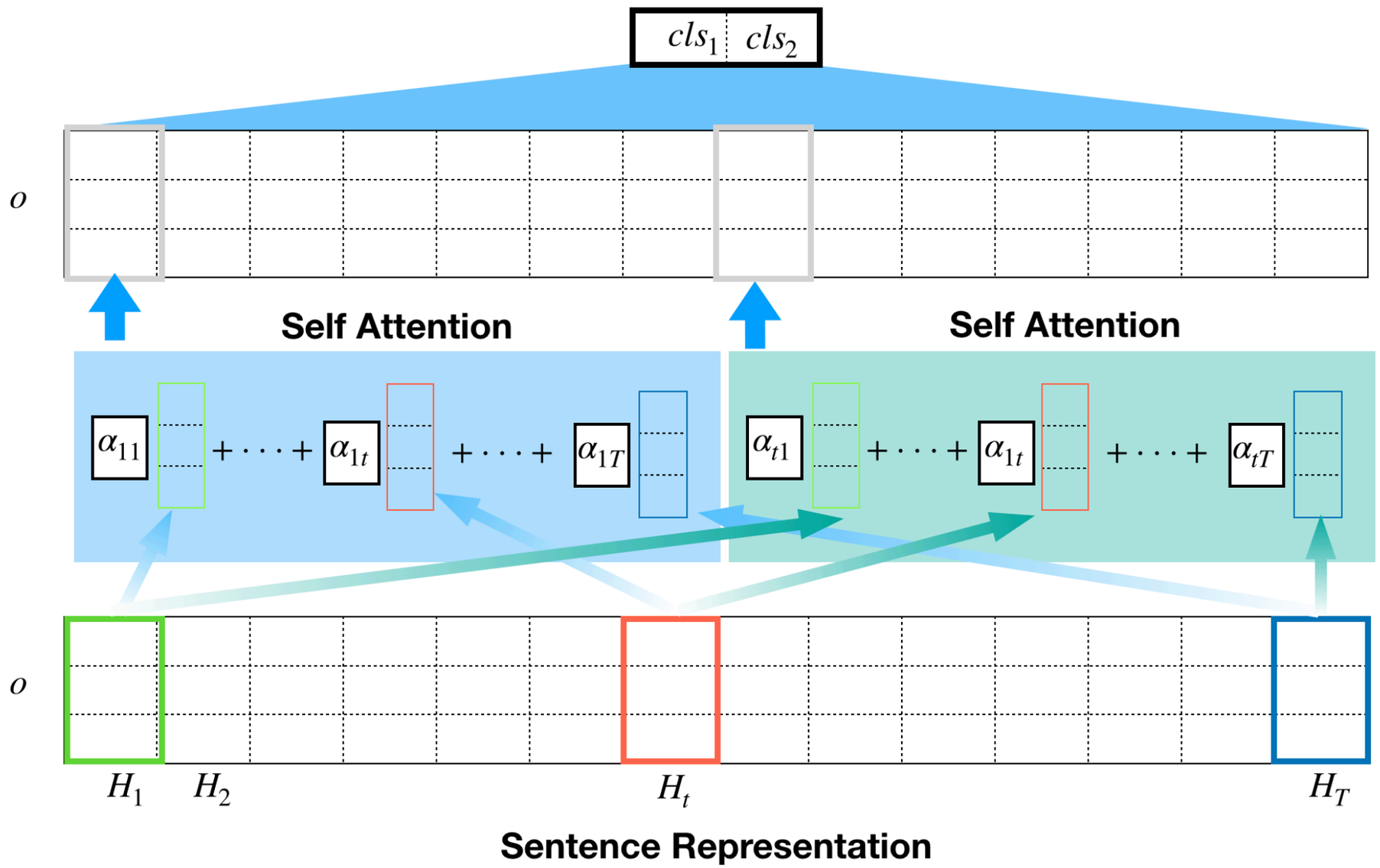


Idea of Self Attention

Why don’t we have more generator function than just indicator?

$$h_{t*} = \sum_{t=0}^{T} g(t, t^*) \cdot f(x_t, x_{t^*})$$

Self - Attention



Self - Attention

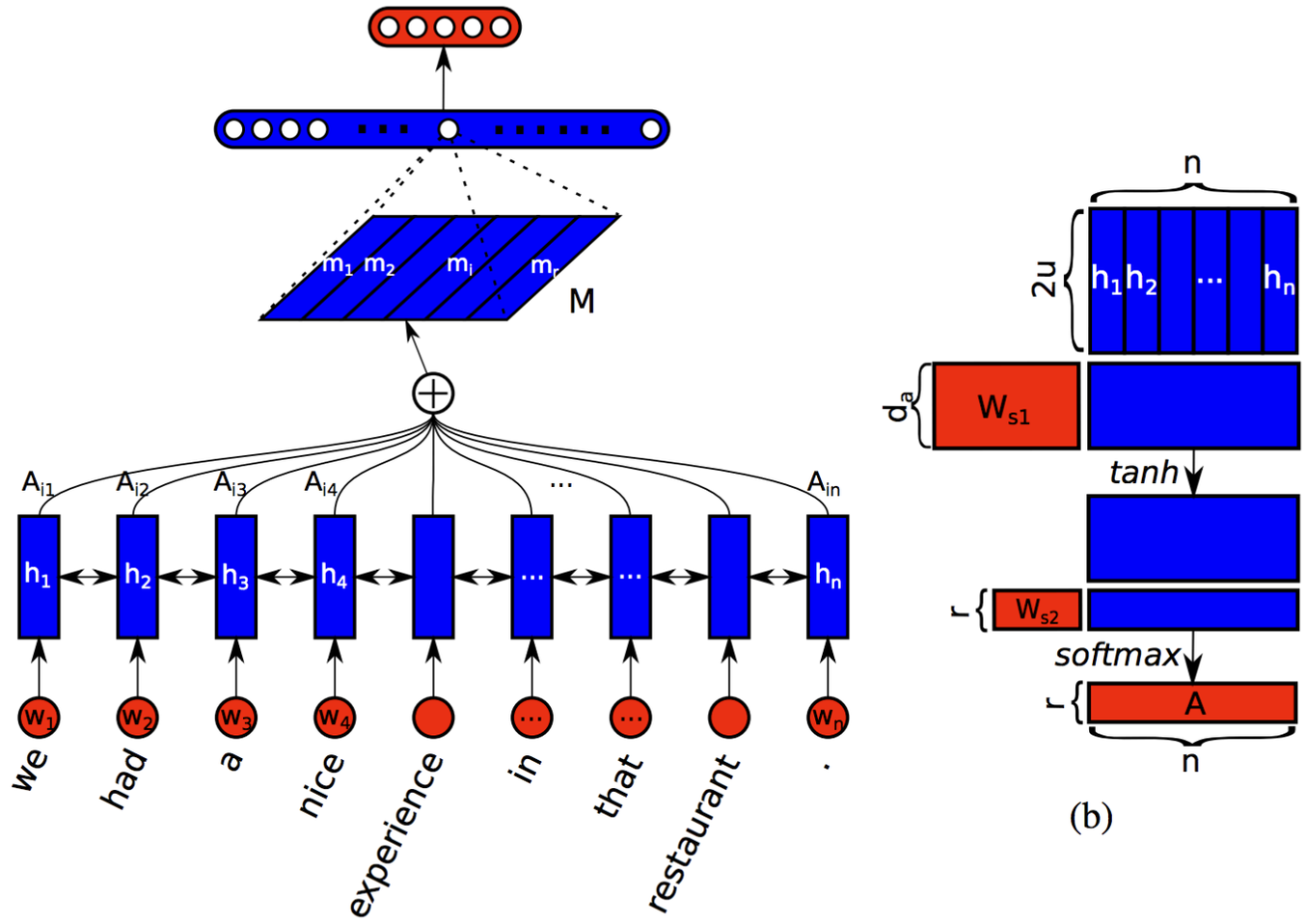


Figure 1: A sample model structure showing the sentence embedding model combined with a fully connected and softmax layer for sentiment analysis (a). The sentence embedding M is computed as multiple weighted sums of hidden states from a bidirectional LSTM ($\mathbf{h_1}, ..., \mathbf{h_n}$), where the summation weights ($A_{i1}, ..., A_{in}$) are computed in a way illustrated in (b). Blue colored shapes stand for hidden representations, and red colored shapes stand for weights, annotations, or input/output.

Transformer

Sorely based on attention-mechanism without underlying RNNs

RNNs are ***slow***, especially for LSTM and ***loose long-term dependency***

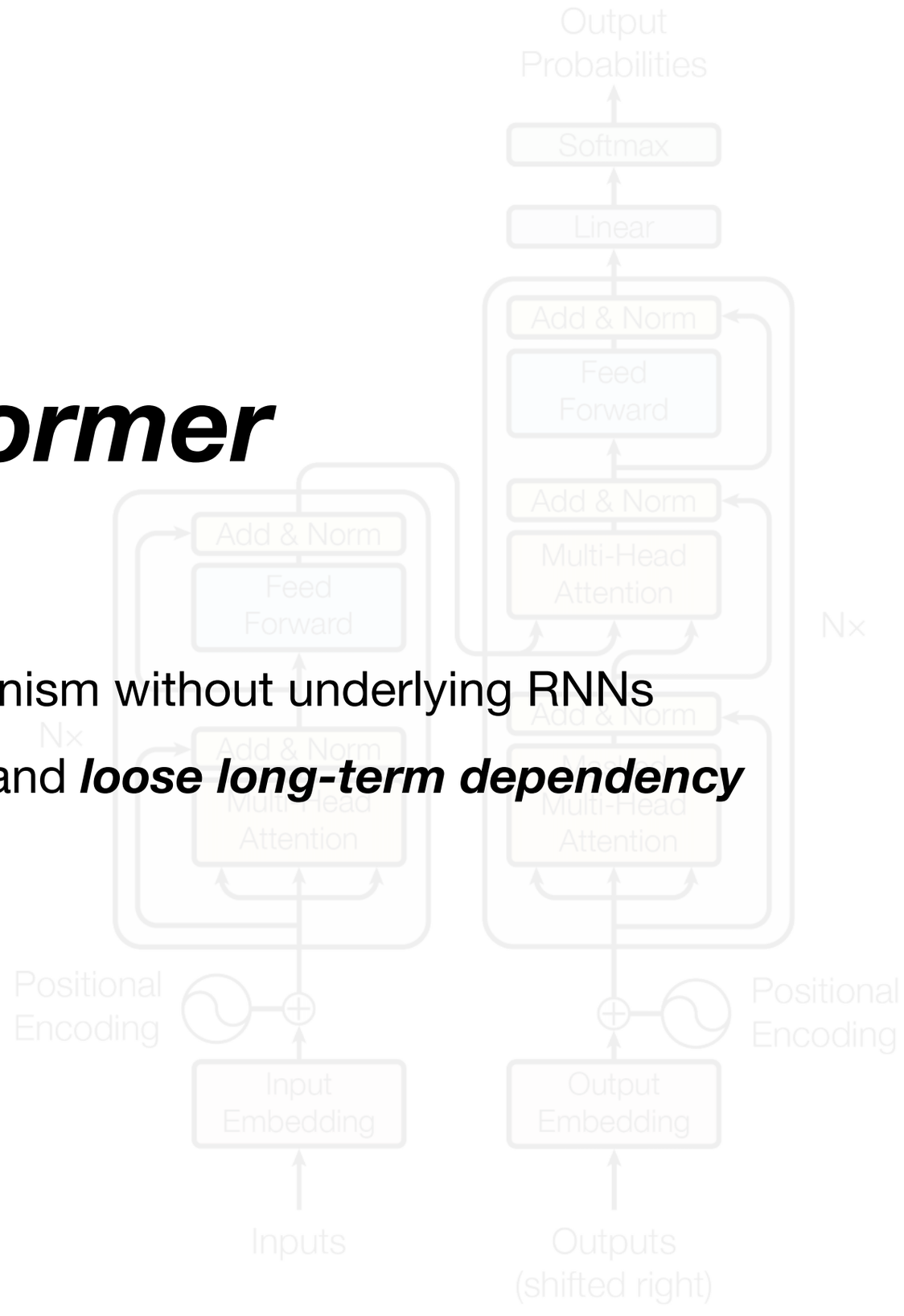


Figure 1: The Transformer - model architecture.

Transformer

Sorely based on attention-mechanism without underlying RNNs

RNNs are ***slow***, especially for LSTM and ***loose long-term dependency***

Much ***faster*** than RNN by putting ***sequence data all at once***

Can capture ***long-term dependency*** by employing ***attention-mechanism***

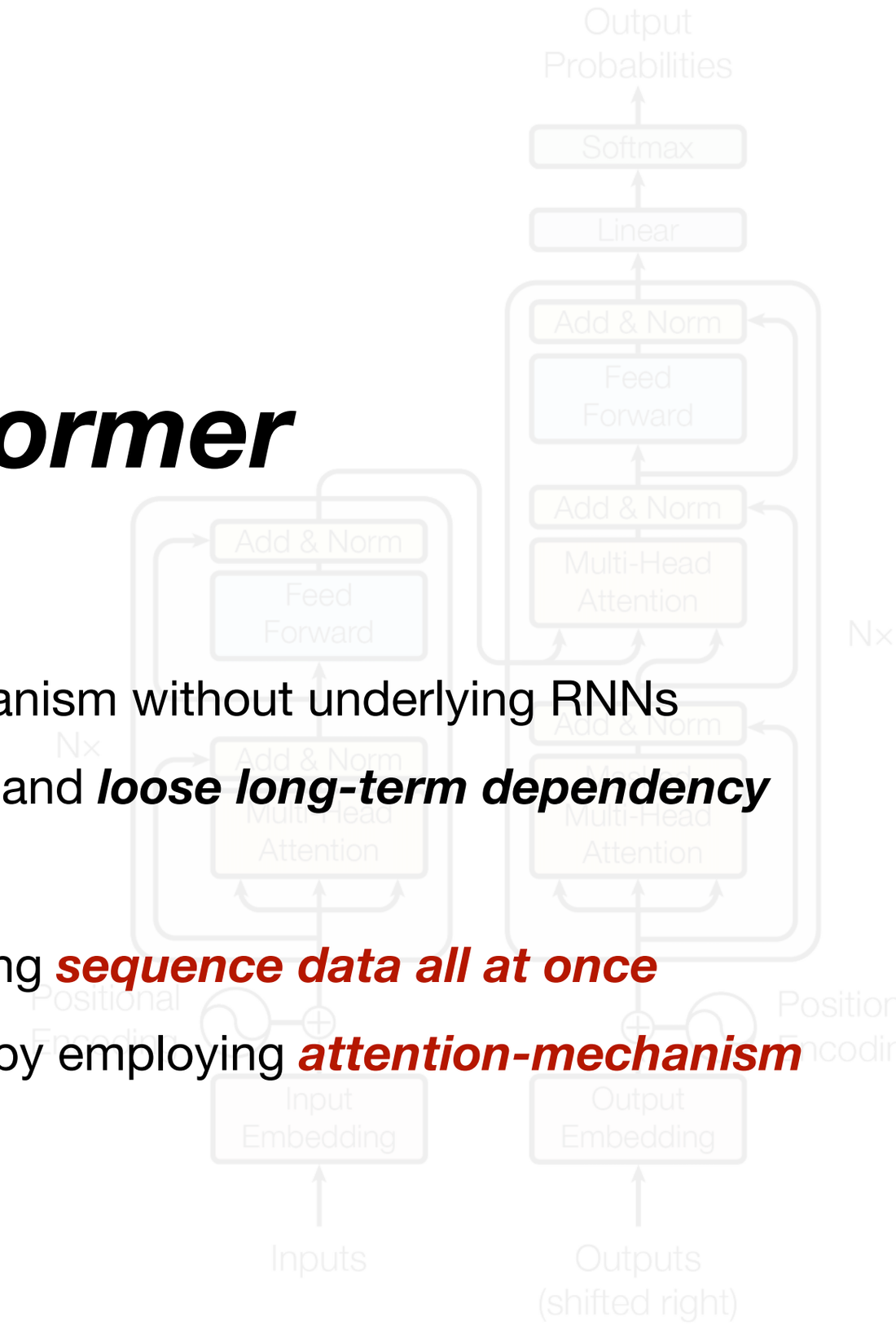


Figure 1: The Transformer - model architecture.

Transformer

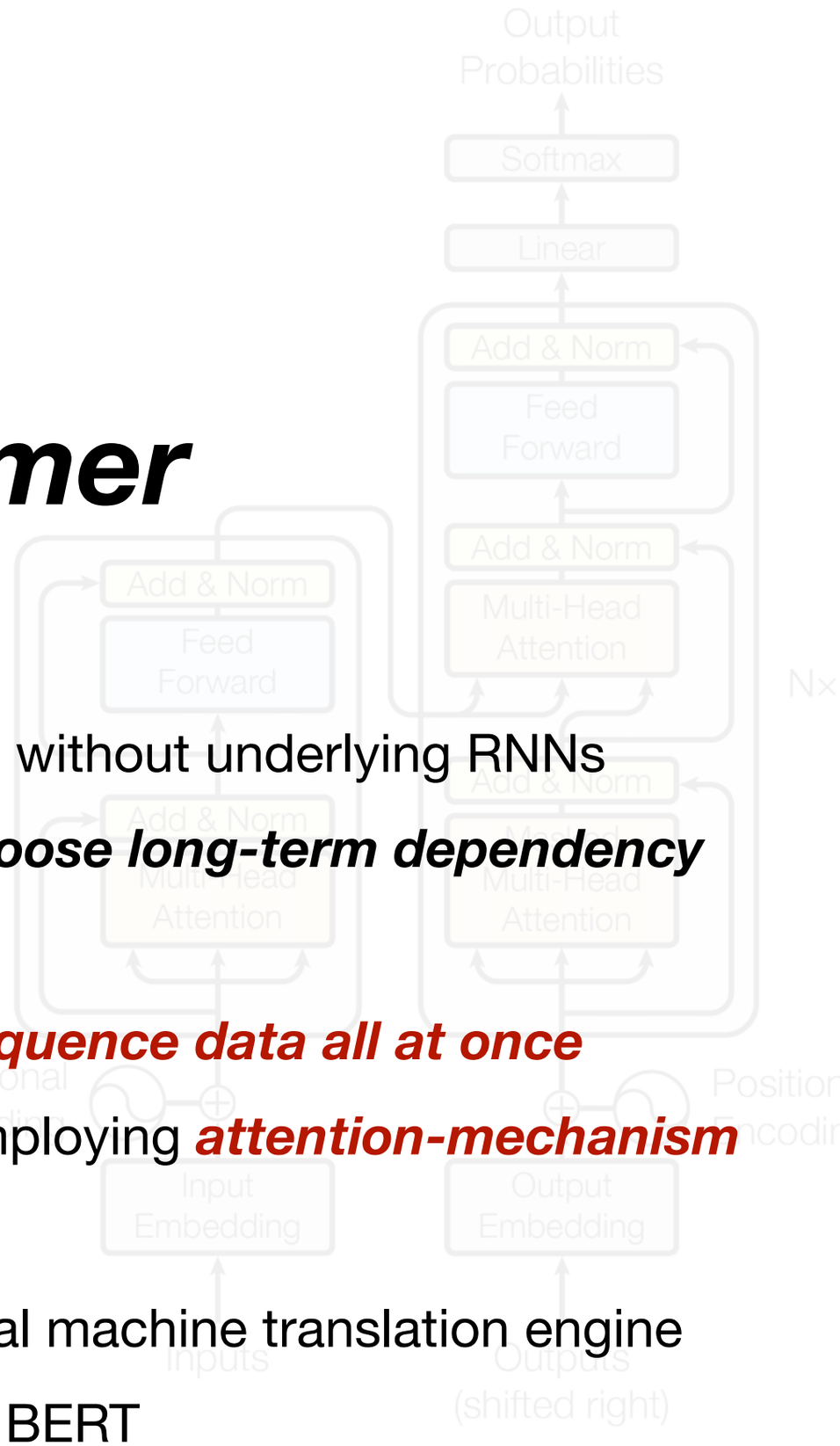


Figure 1: The Transformer - model architecture.

Sorely based on attention-mechanism without underlying RNNs

RNNs are ***slow***, especially for LSTM and ***loose long-term dependency***

Much ***faster*** than RNN by putting ***sequence data all at once***

Can capture ***long-term dependency*** by employing ***attention-mechanism***

Known as the core engine for STOA neural machine translation engine

Building blocks of BERT

Transformer Architecture

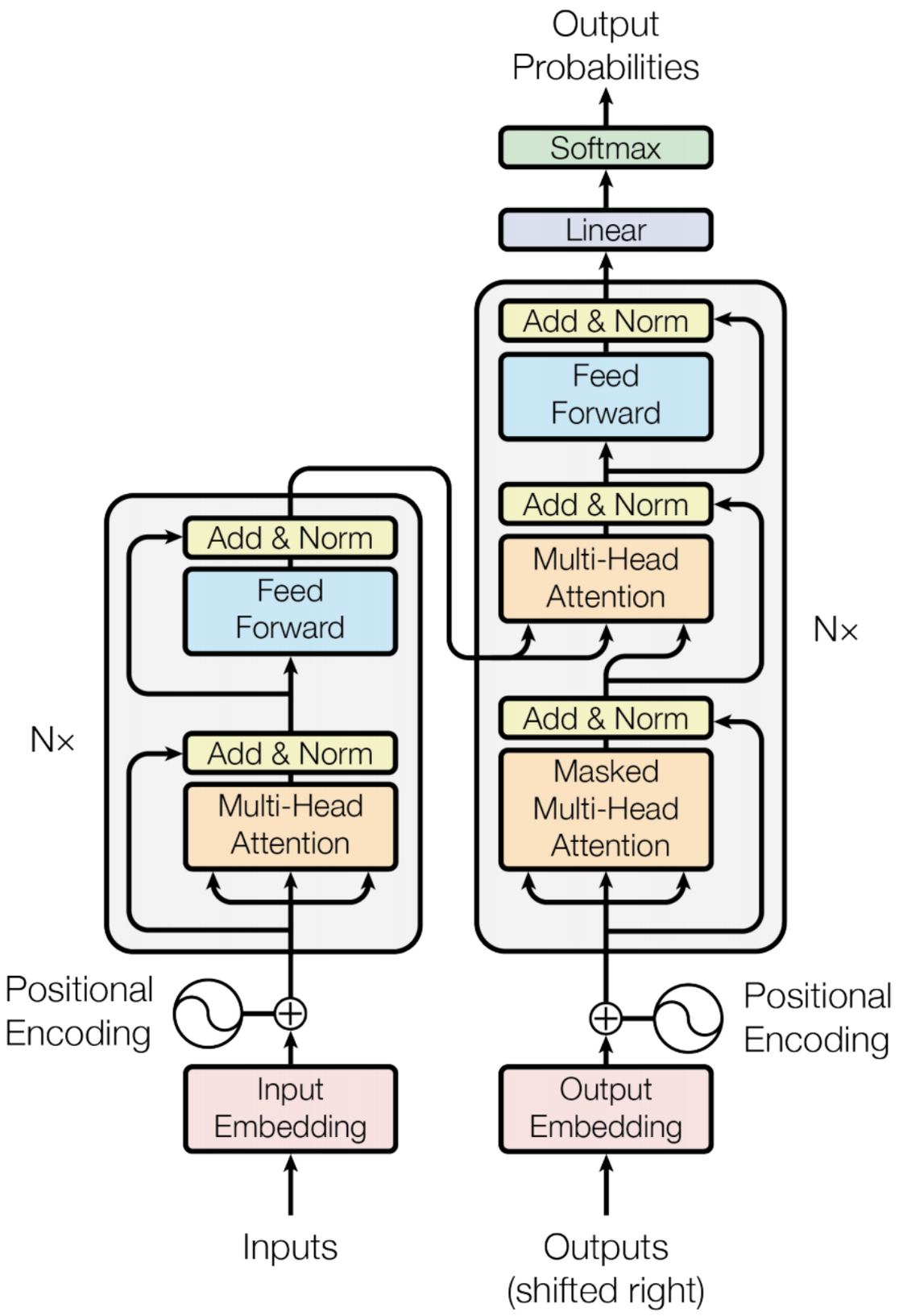


Figure 1: The Transformer - model architecture.

Introduced several tricks

- Self-attention
- Multi-head attention
- Masking
- Positional Encoding
- Residual connection
- Label smoothing
- Layer normalization

Introduced several tricks

- Self-attention
- Multi-head attention

To extract information effectively

Introduced several tricks

- Self-attention
- Multi-head attention
- Masking

To extract information effectively

Not to include tokens not seen yet

Introduced several tricks

- Self-attention
- Multi-head attention
- Masking
- Positional Encoding

To extract information effectively

Not to include tokens not seen yet

Not to loose order information

Introduced several tricks

- Self-attention
- Multi-head attention
- Masking
- Positional Encoding
- Residual connection

To extract information effectively

Not to include tokens not seen yet

Not to loose order information

To go deeper

Introduced several tricks

- Self-attention
- Multi-head attention
- Masking
- Positional Encoding
- Residual connection
- Label smoothing

To extract information effectively

Not to include tokens not seen yet

Not to loose order information

To go deeper

To Regularize loss

Introduced several tricks

- Self-attention
- Multi-head attention
- Masking
- Positional Encoding
- Residual connection
- Label smoothing
- Layer normalization

To extract information effectively

Not to include tokens not seen yet

Not to loose order information

To go deeper

To Regularize loss

For better convergence

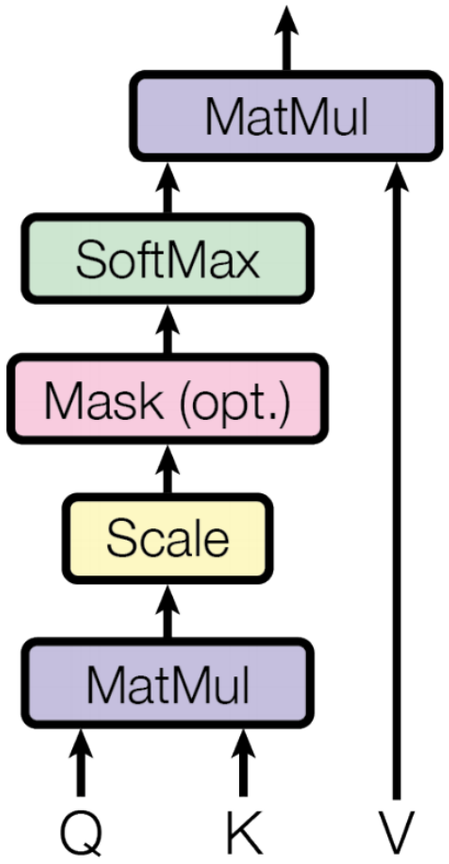
Attention

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.

Query, Key, Value

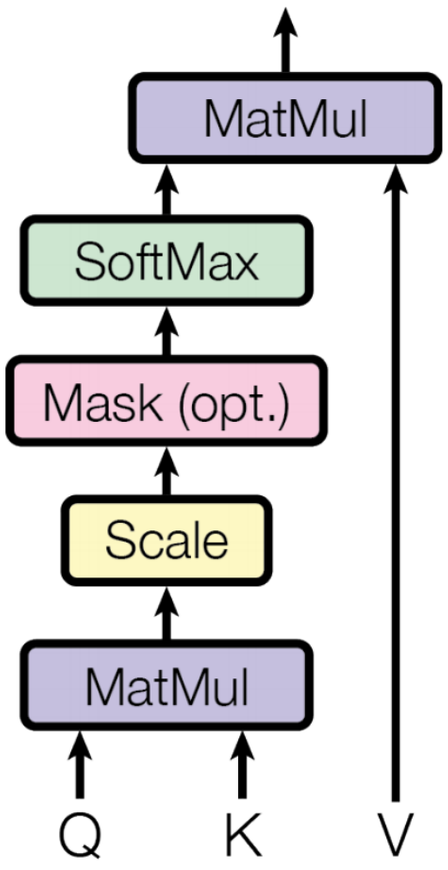
My opinion: To put attention & self-attention under the same umbrella

Scaled Dot-Product Attention

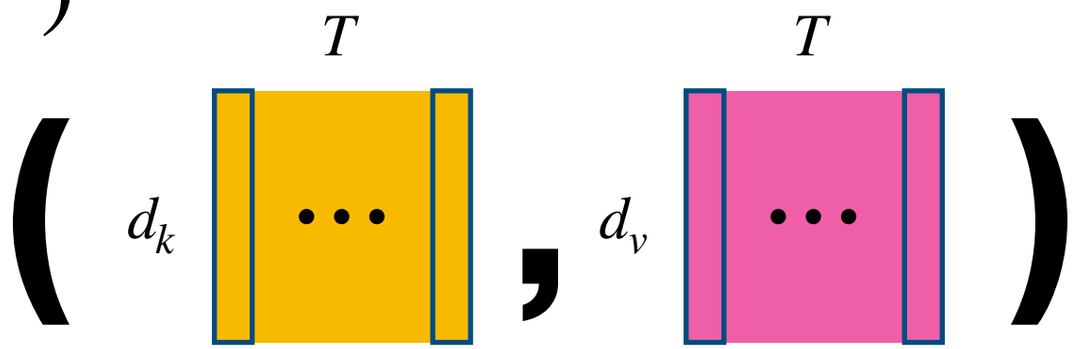


$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

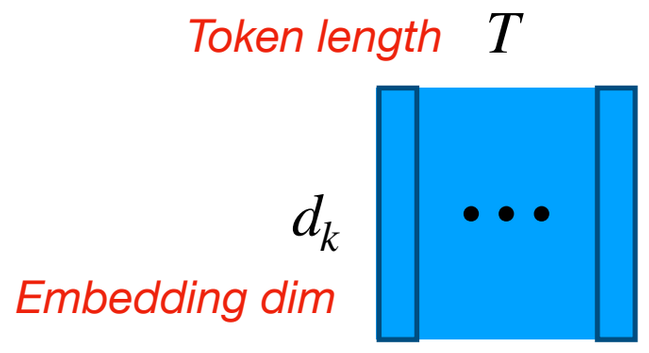
Scaled Dot-Product Attention



(K^T, V^T)

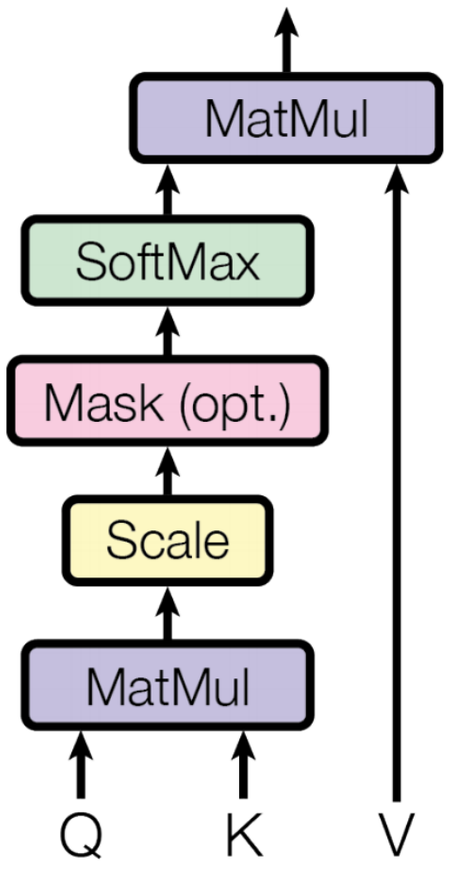


Q^T

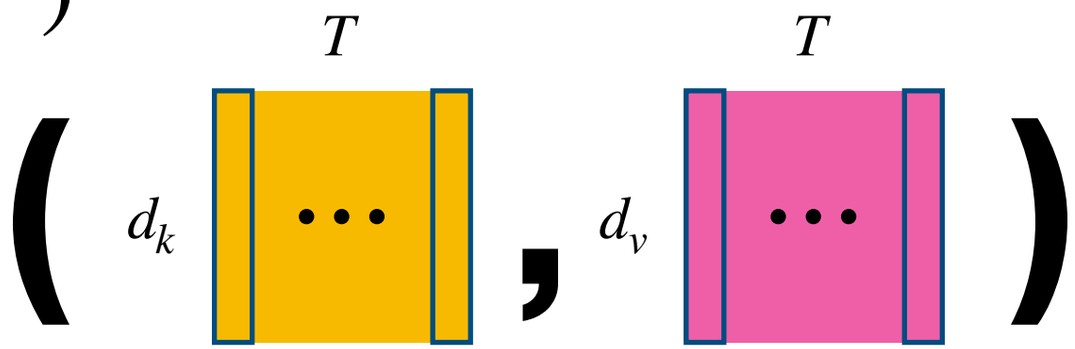


$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

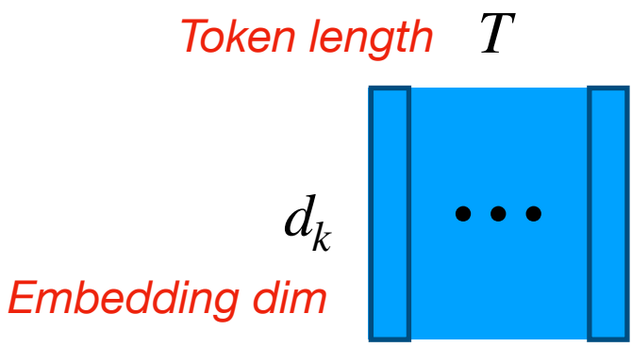
Scaled Dot-Product Attention



(K^T, V^T)



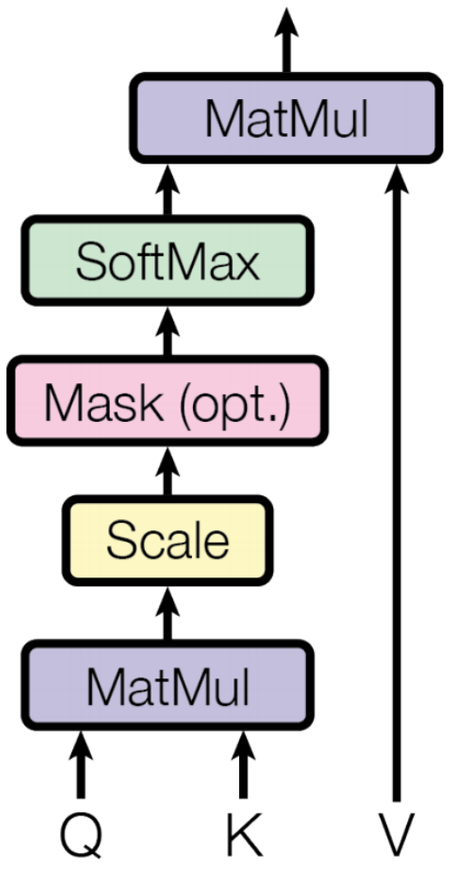
Q^T



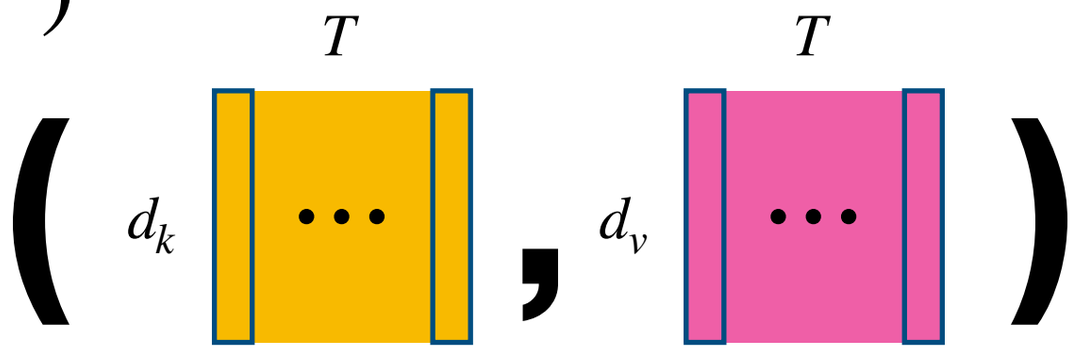
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

dot-product attention is much faster and more space-efficient in practice

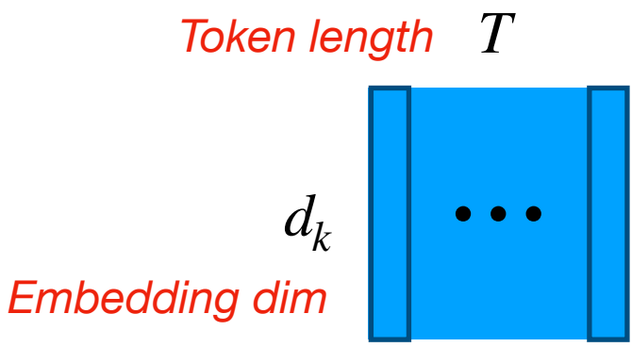
Scaled Dot-Product Attention



(K^T, V^T)



Q^T

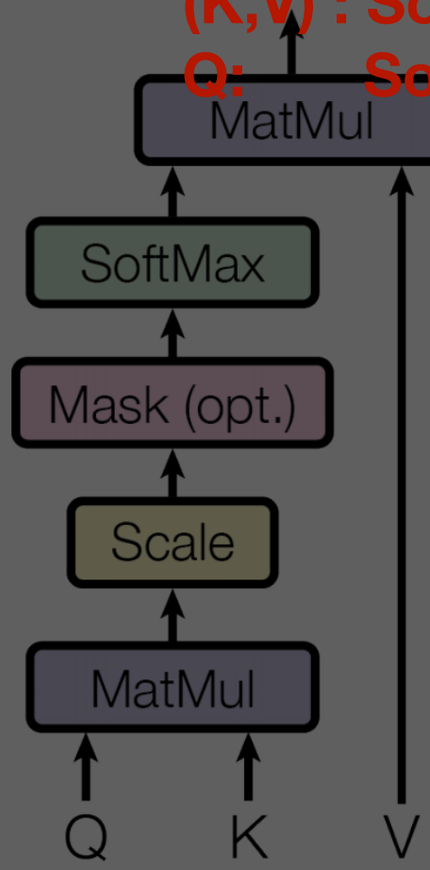


$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

dot-product attention is much faster and more space-efficient in practice

Not to collapse most of values to zero

Scaled Dot-Product Attention

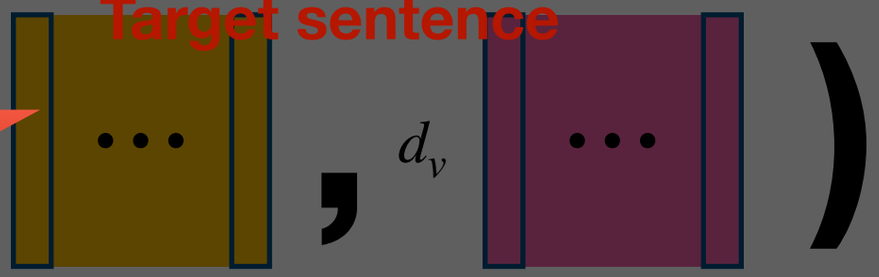


(K^T, V^T)

(K,V) : Source sentence
Q: Source sentence

Or

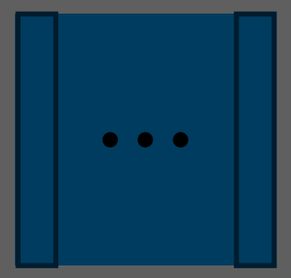
(K,V) : Target sentence
Q: Target sentence



Self-attention

Q

Token length T



Embedding dim

(K,V) : Source sentence
Q: Target sentence

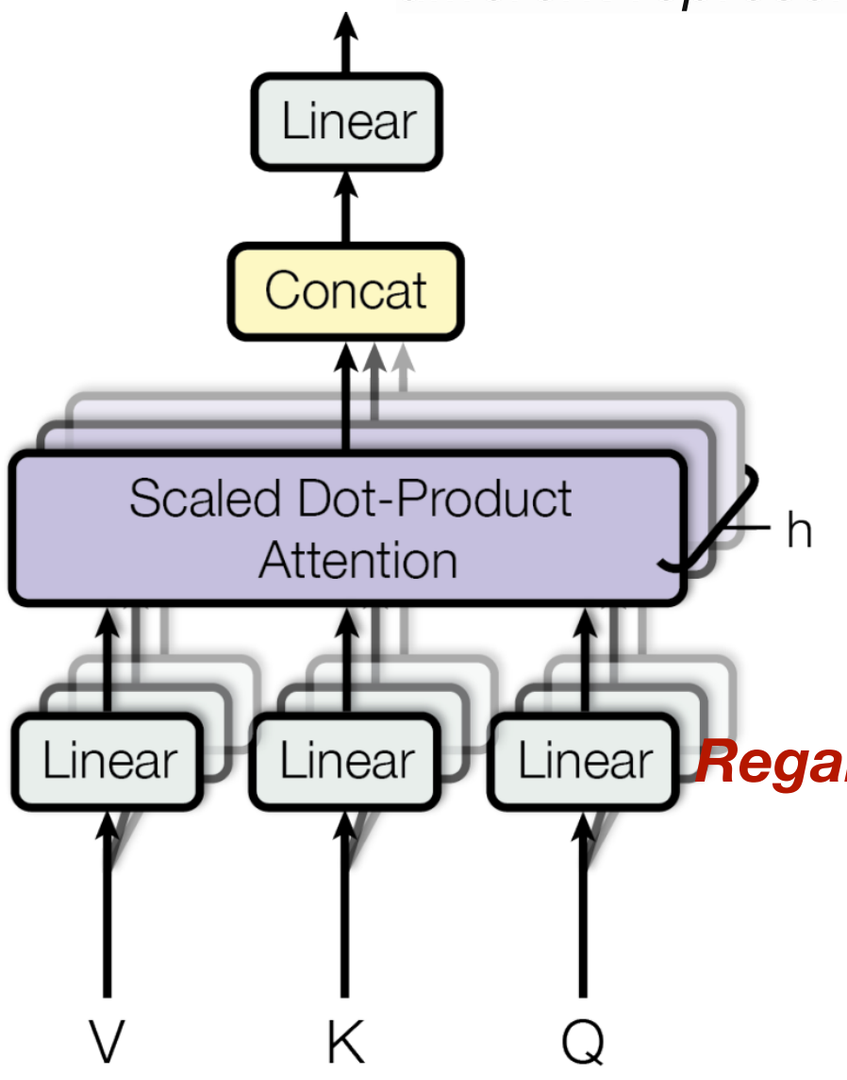
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Attention

- dot-product attention is much faster
- pace-efficient in practice
- Not to collapse most of values to zero

Multi-Head Attention

Multiple representations are better than just one huge representation
*To **jointly** attend to information from*
different representation subspaces at different positions.



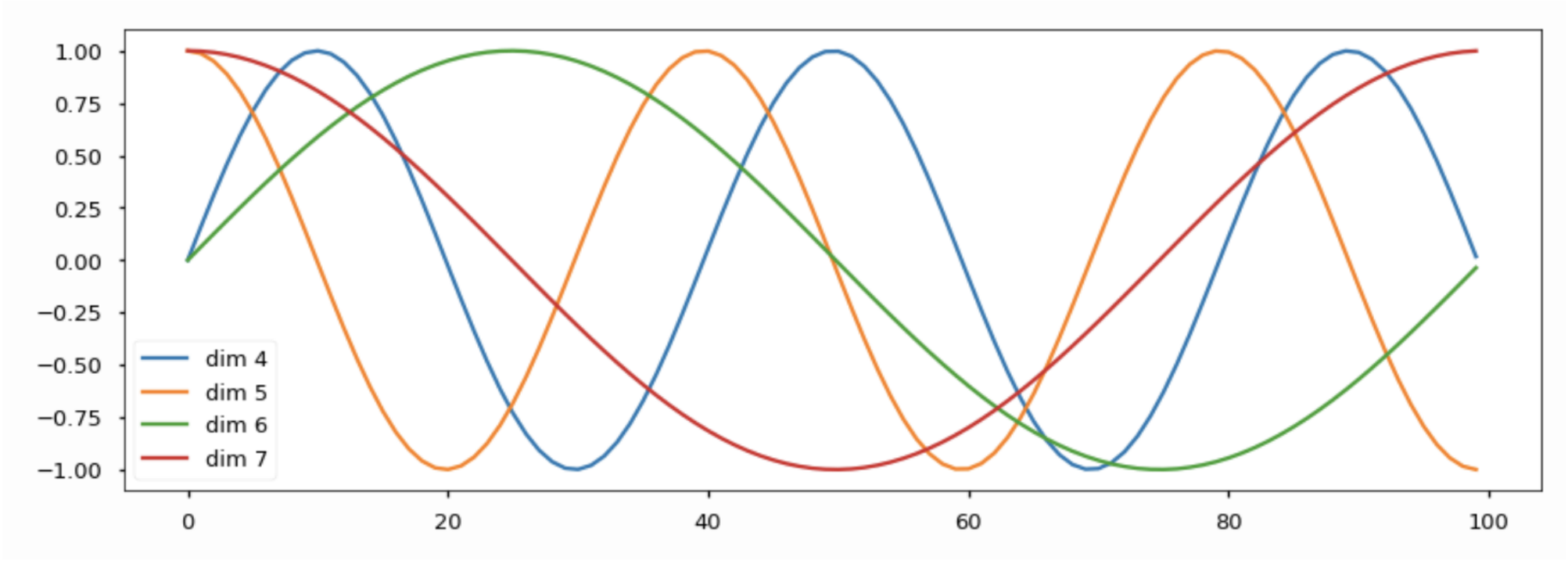
To keep hidden dimension the same
for computational cost saving,
each head has d/h many nodes as input

$$d_k = d_{model}/h$$

Regardless of position

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Positional Encoding



Transformer Architecture

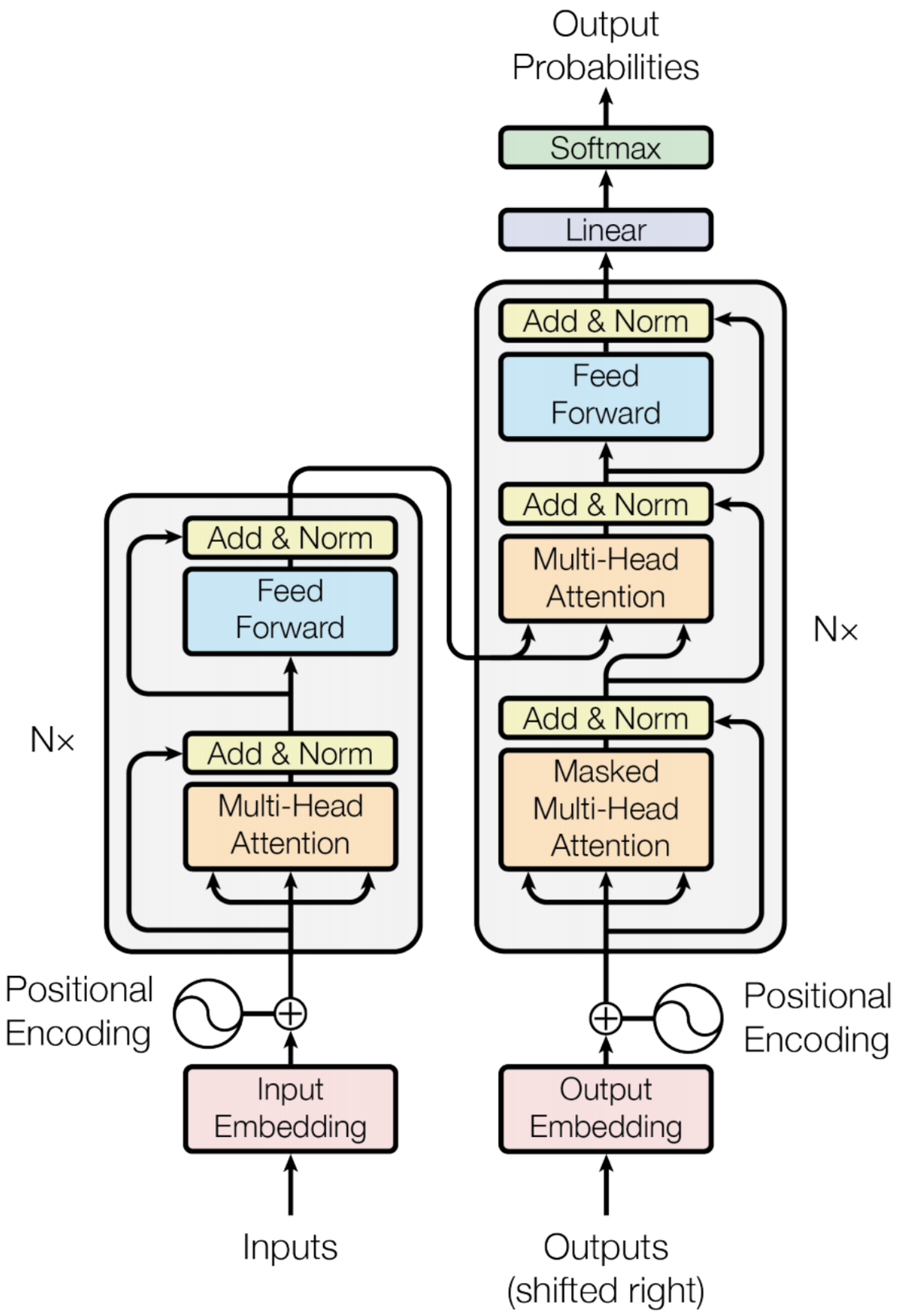
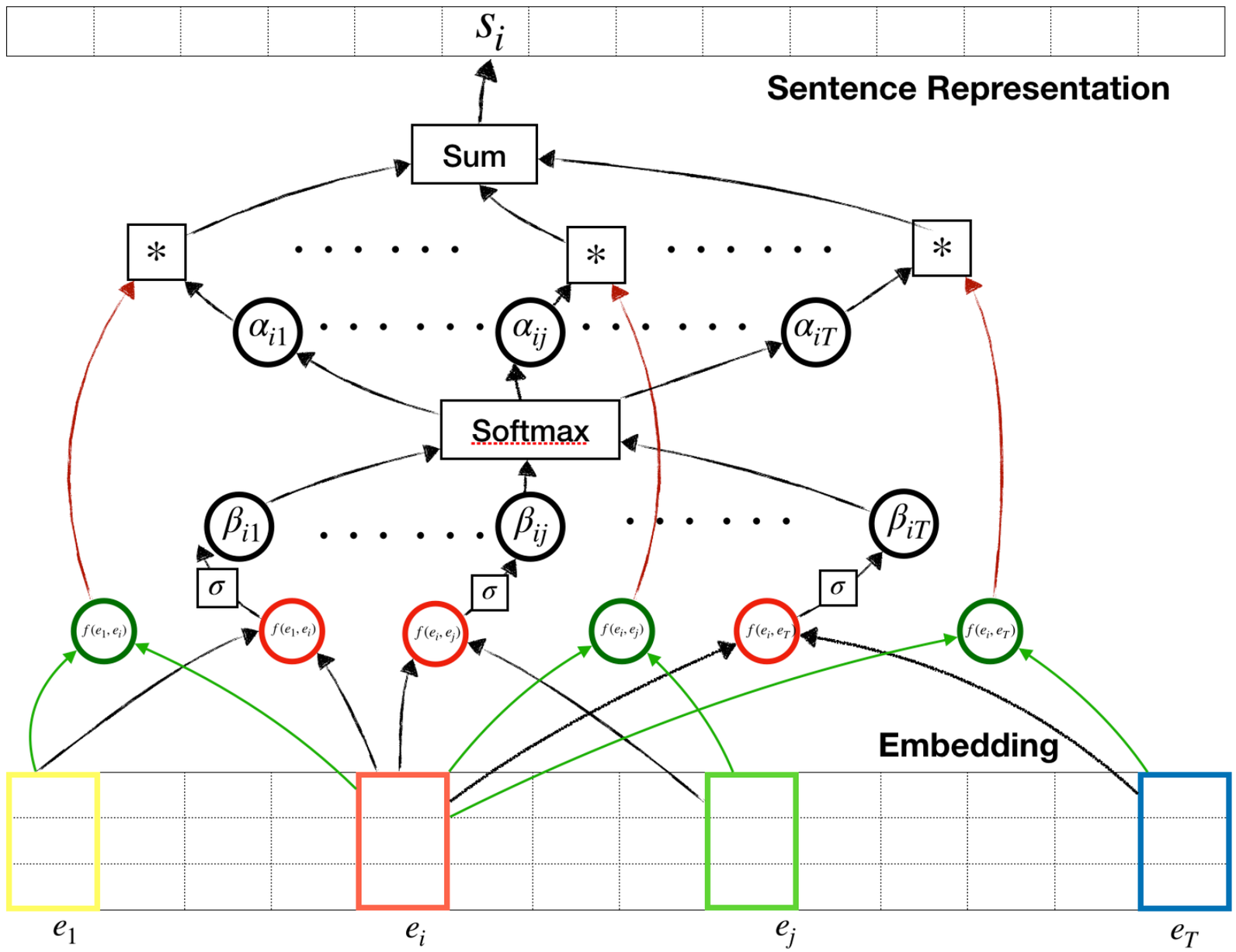


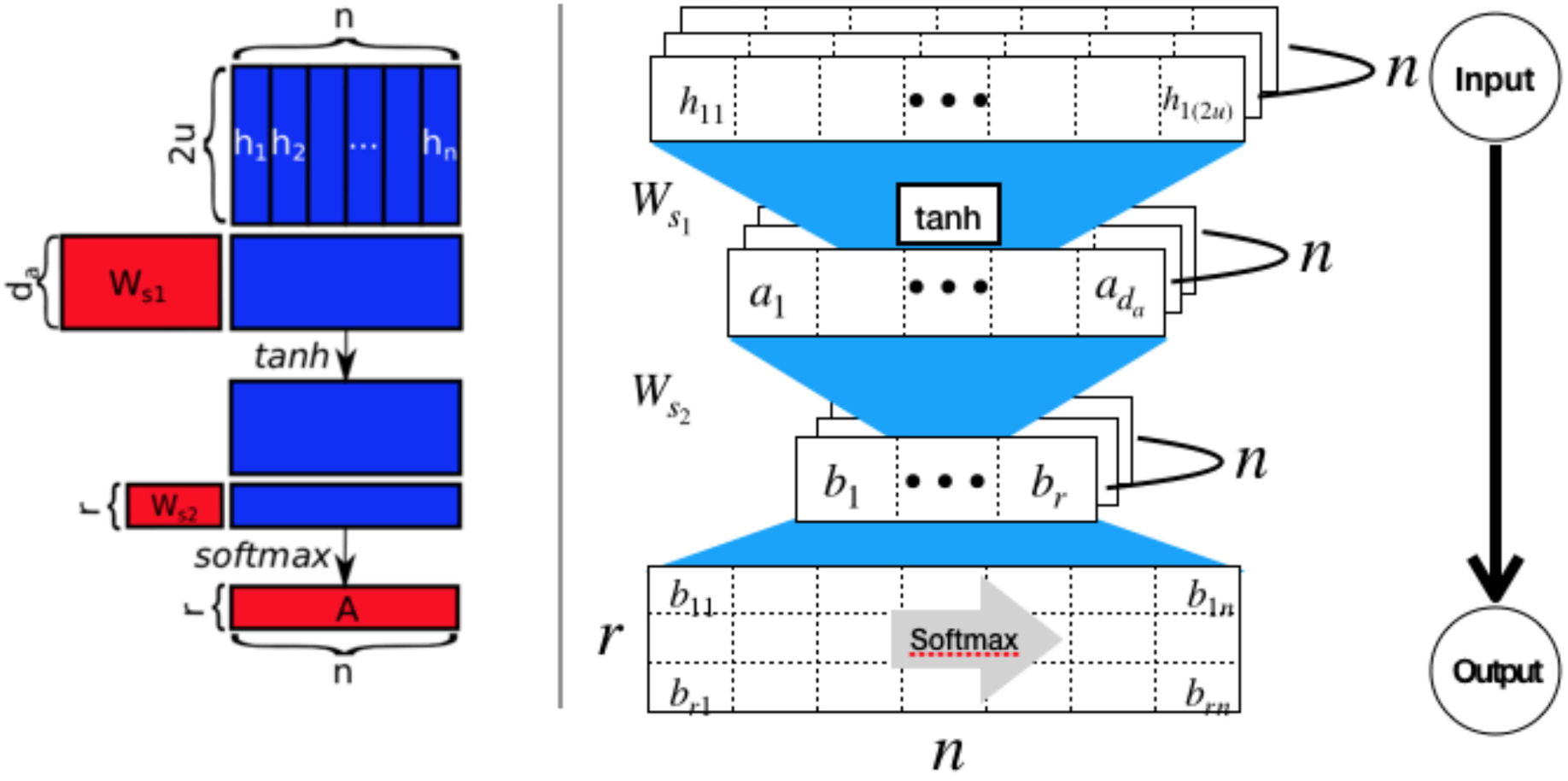
Figure 1: The Transformer - model architecture.

Sentiment analysis with Self attention

Self - Attention



Self - Attention Implementation

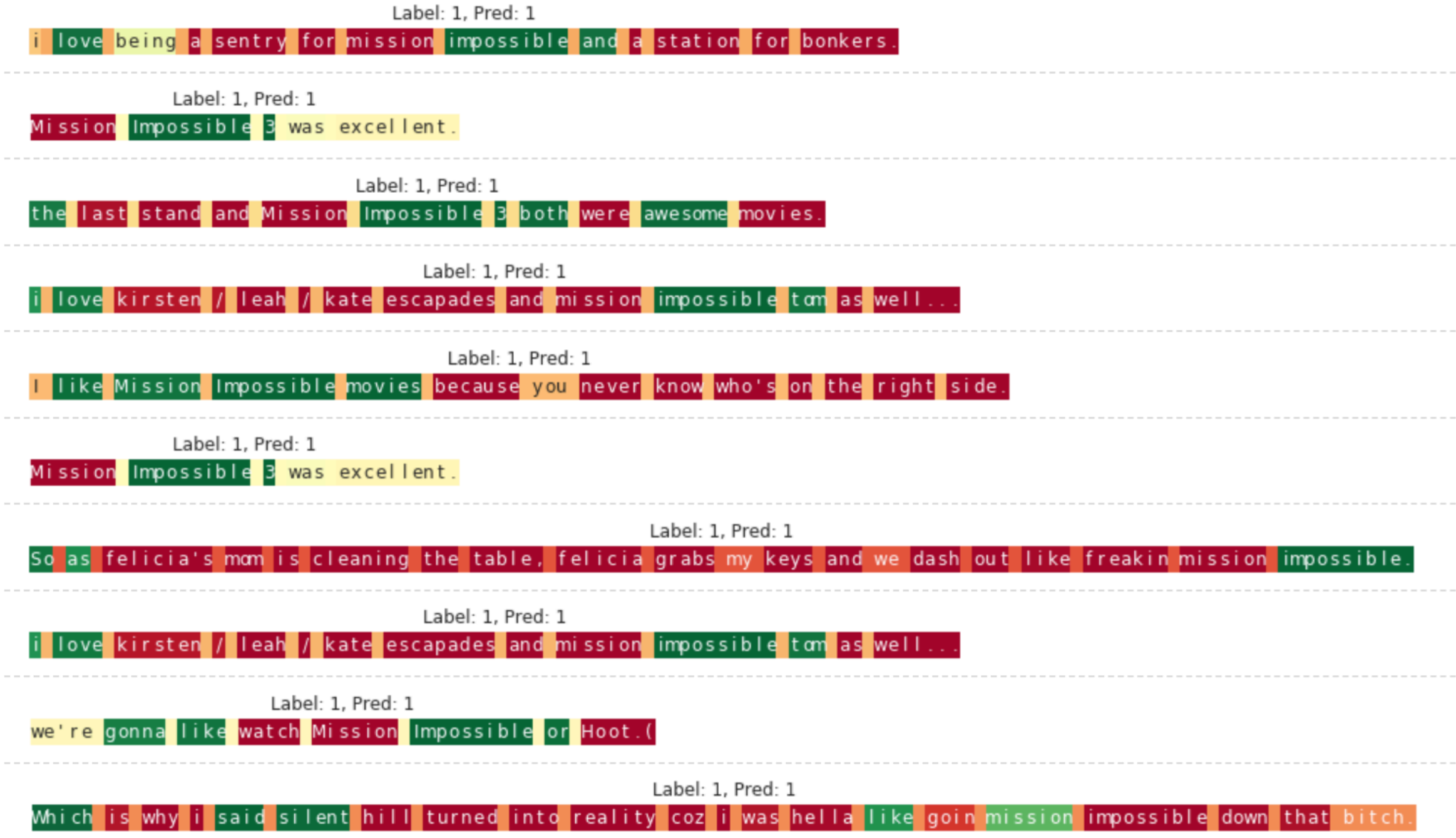


Weight를 layer로 인식!!

Self - Attention Implementation

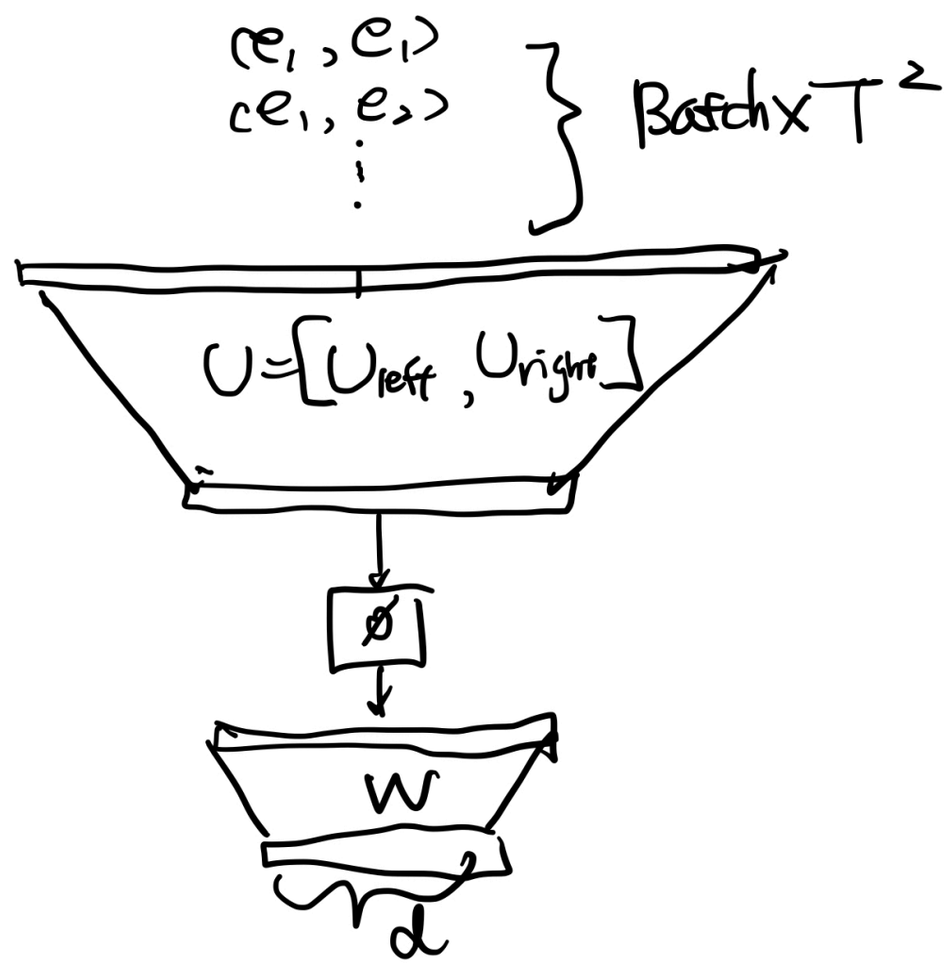
```
class Sentence_Representation(nn.Block):
    def __init__(self, **kwargs):
        super(Sentence_Representation, self).__init__()
        for (k, v) in kwargs.items():
            setattr(self, k, v)

        with self.name_scope():
            self.embed = nn.Embedding(self.vocab_size, self.emb_dim)
            self.drop = nn.Dropout(.2)
            self.bi_rnn = rnn.BidirectionalCell(
                 rnn.LSTMCell(hidden_size = self.hidden_dim // 2),
                 rnn.LSTMCell(hidden_size = self.hidden_dim // 2)
            )
            self.w_1 = nn.Dense(self.d, use_bias = False)
            self.w_2 = nn.Dense(self.r, use_bias = False)

    def forward(self, x, hidden):
        embeds = self.embed(x) # batch * time_step * emb_dim
        h, _ = self.bi_rnn.unroll(length = embeds.shape[1] \
                                  , inputs = embeds \
                                  , layout = 'NTC' \
                                  , merge_outputs = True)
        # For understanding
        batch_size, time_step, _ = h.shape
        # get self-attention
        _h = h.reshape((-1, self.hidden_dim))
        _w = nd.tanh(self.w_1(_h)) # Batch * time_step * r
        w = self.w_2(_w) # Batch * time_step * d
        _att = w.reshape((-1, time_step, self.r)) # Batch * time_step * r
        att = nd.softmax(_att, axis = 1)
        x = gemm2(att, h, transpose_a = True)  # h = Batch * time_step * (2 * hidden_dim)
        return x, att
```

Self - Attention some results

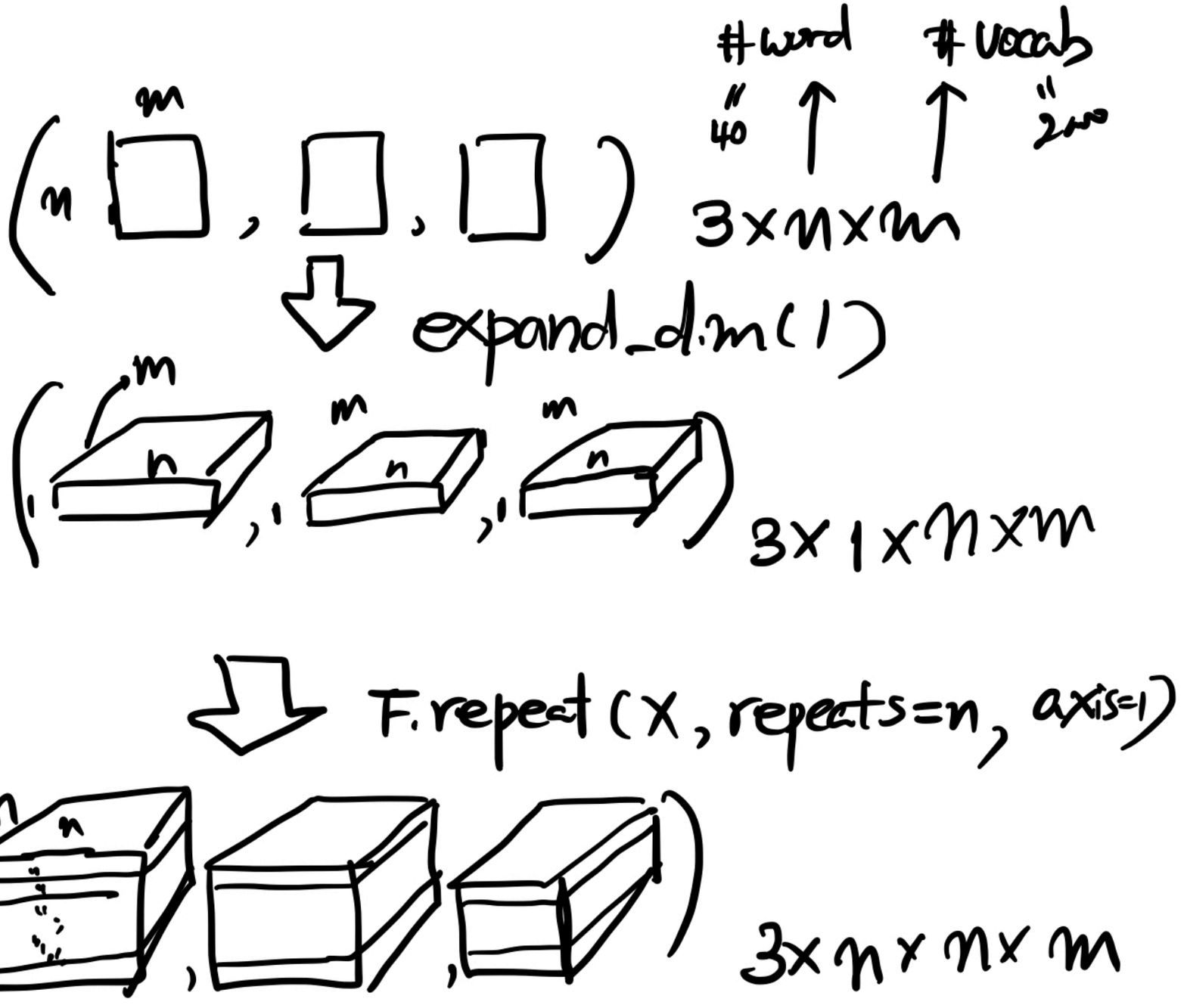


Sentiment analysis with Relation Nework

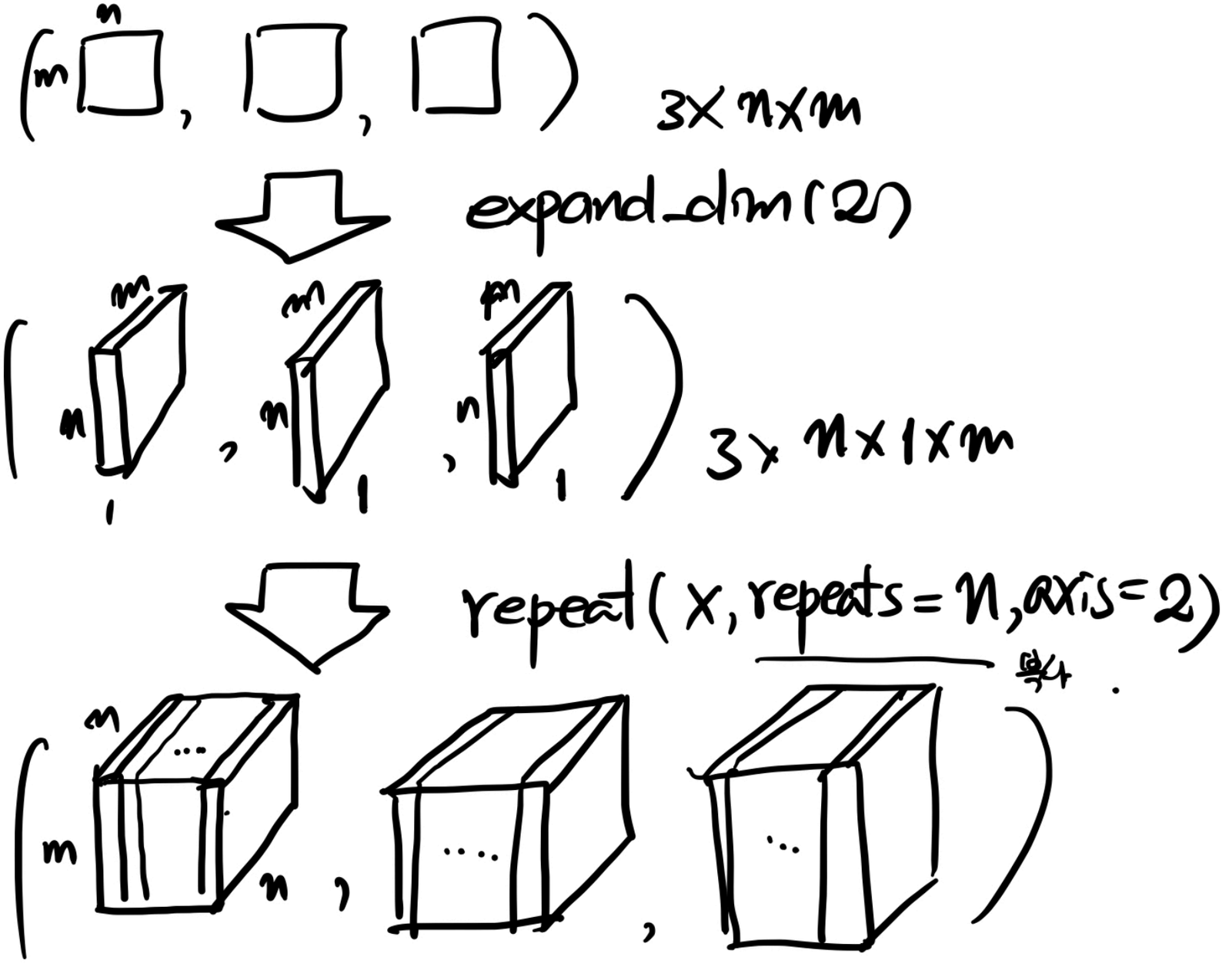
$$f(t_i, t_j) = W\phi\left\{(U_{left}, U_{right}) \cdot \begin{pmatrix} e_i \\ e_j \end{pmatrix}\right\}$$



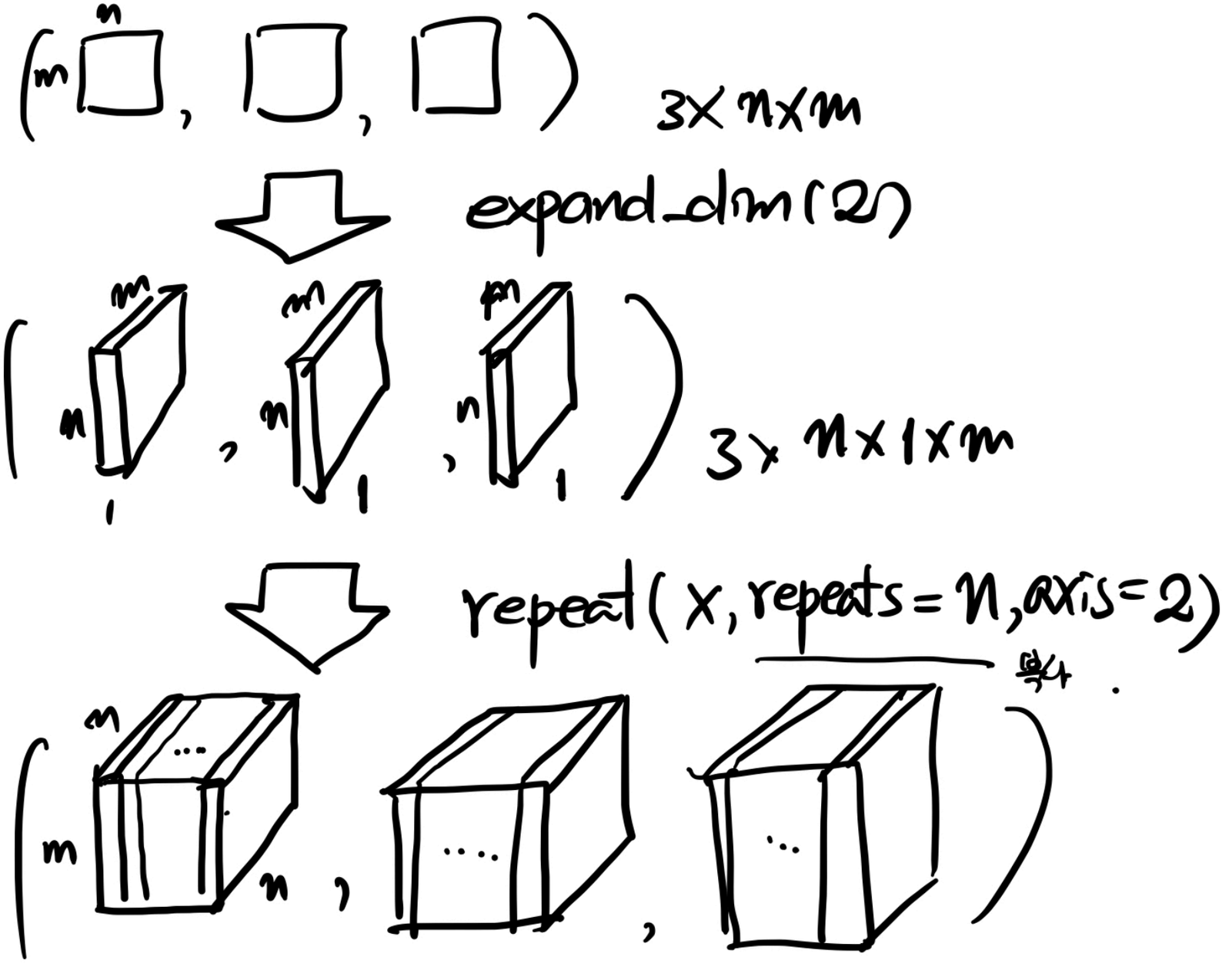
$$f(t_i, t_j) = W\phi\left\{(U_{left}, U_{right}) \cdot \begin{pmatrix} e_i \\ e_j \end{pmatrix}\right\}$$



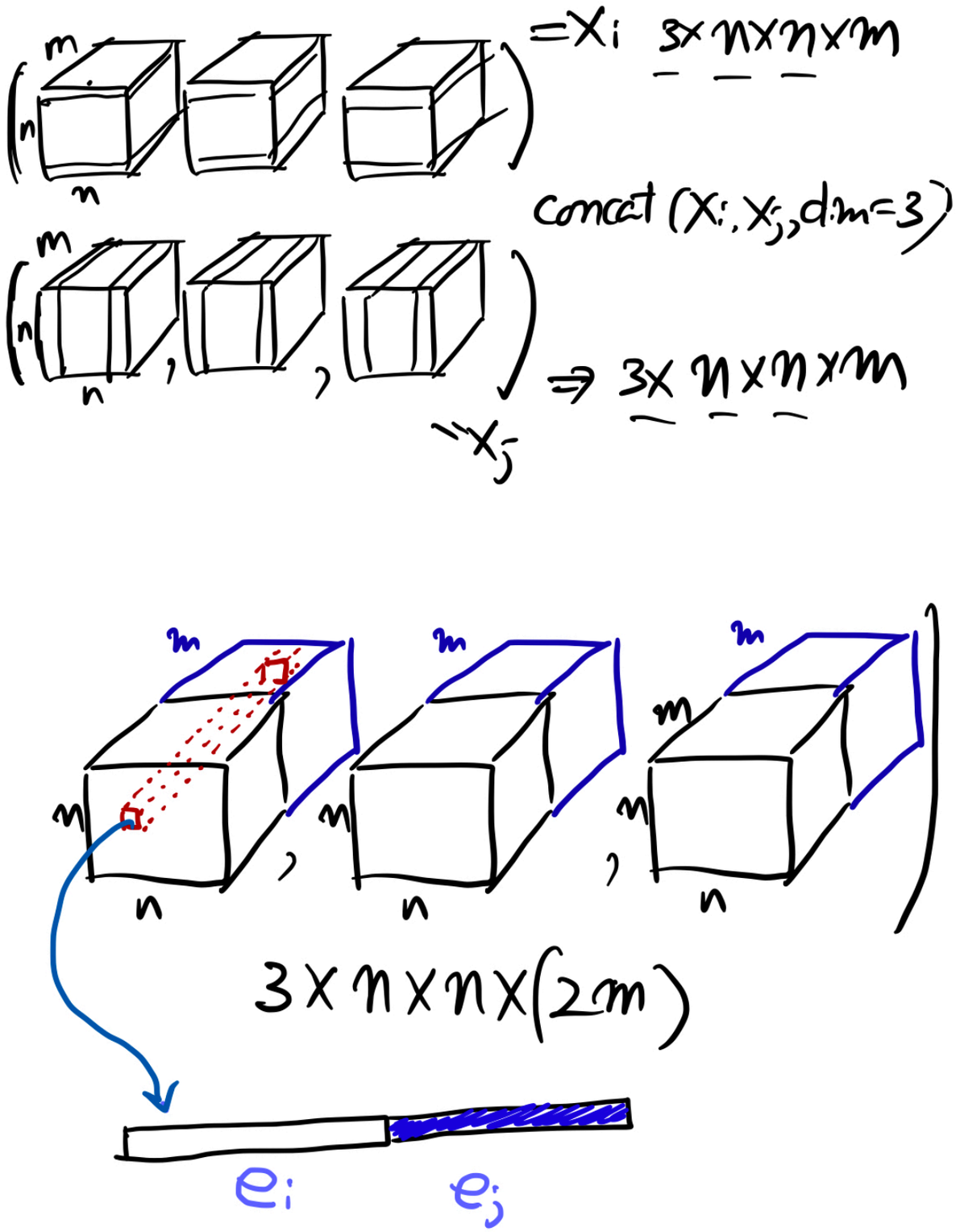
$$f(t_i, t_j) = W\phi\left\{(U_{left}, U_{right}) \cdot \begin{pmatrix} e_i \\ e_j \end{pmatrix}\right\}$$



$$f(t_i, t_j) = W\phi\left\{(U_{left}, U_{right}) \cdot \begin{pmatrix} e_i \\ e_j \end{pmatrix}\right\}$$



$$f(t_i, t_j) = W\phi\left\{(U_{left}, U_{right}) \cdot \begin{pmatrix} e_i \\ e_j \end{pmatrix}\right\}$$



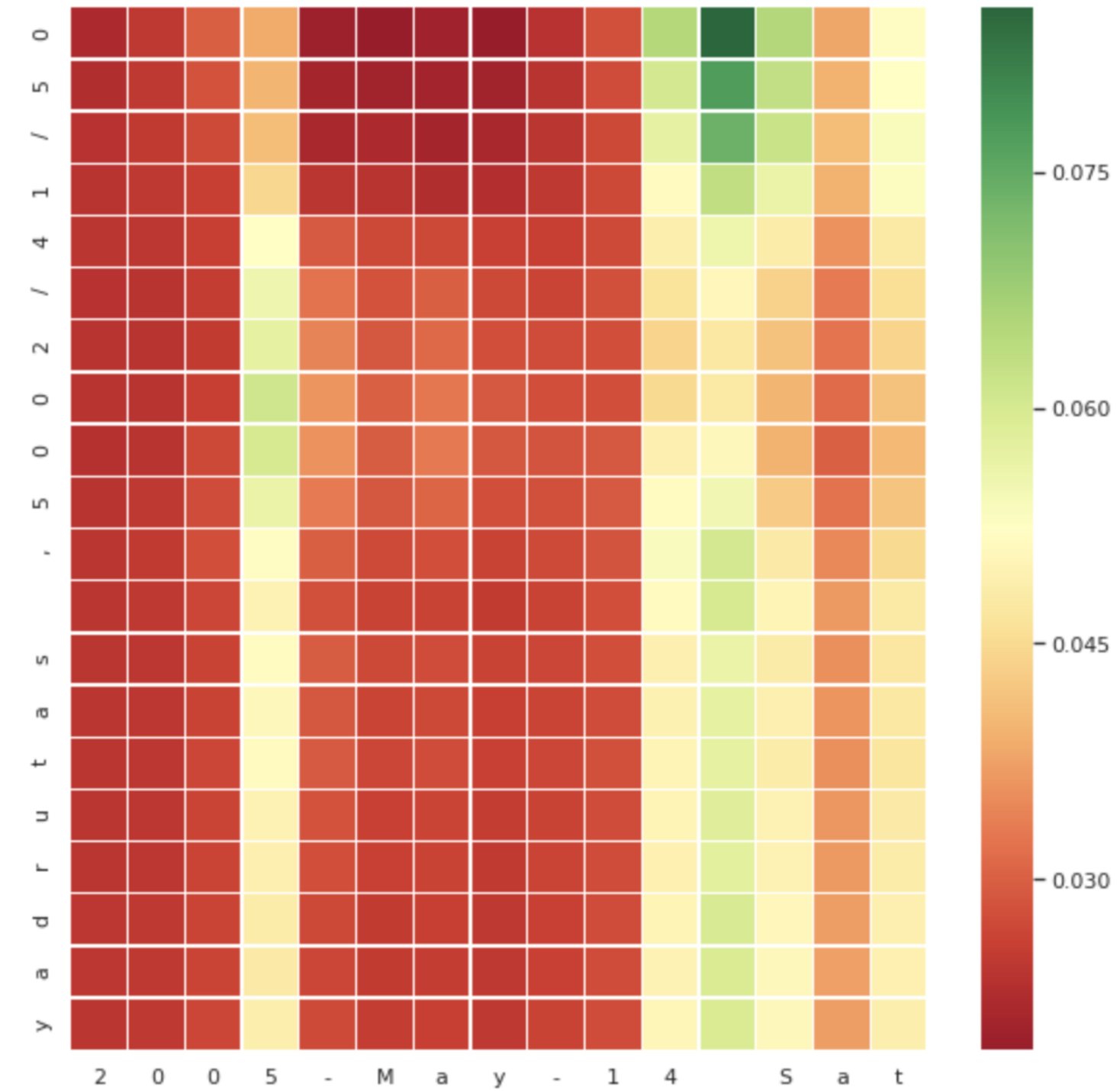
Relation Network

```
class Sentence_Representation(nn.Block):
    def __init__(self, **kwargs):
        super(Sentence_Representation, self).__init__()
        for (k, v) in kwargs.items():
            setattr(self, k, v)

        with self.name_scope():
            self.embed = nn.Embedding(self.vocab_size, self.emb_dim)
            self.g_fc1 = nn.Dense(self.hidden_dim, activation='relu', flatten = False)
            self.g_fc2 = nn.Dense(self.hidden_dim, activation='relu', flatten = False)

    def forward(self, x):
        embeds = self.embed(x) # batch * time_step * emb_dim
        x_i = embeds.expand_dims(1) # batch * 1 * time_step * emb_dim
        x_i = nd.repeat(x_i,repeats= self.sentence_length, axis=1)
        x_j = embeds.expand_dims(2) # batch * time_step * 1 * emb_dim
        x_j = nd.repeat(x_j,repeats= self.sentence_length, axis=2)
        x_full = nd.concat(x_i,x_j,dim=3) # batch * time_step * time_step * (2 * emb_dim)
        _x = self.g_fc1(x_full) # (batch * time_step * time_step) * (hidden_dim)
        _x = self.g_fc2(_x) # (batch * time_step * time_step) * (hidden_dim)
        print('_x shape after g_2 = {}'.format(_x.shape))
        x_g = _x.reshape((-1, self.sentence_length * self.sentence_length, self.hidden_dim))
        sentence_rep = x_g.sum(1) # (time_step, emb_dim): ignorable
        return sentence_rep
```

Seq2seq with attention

Dot product attention

```
class pattern_matcher(gluon.Block):
    def __init__(self, n_hidden, in_seq_len, out_seq_len, vocab_size, ctx, **kwargs):
        super(pattern_matcher, self).__init__(**kwargs)
        self.in_seq_len = in_seq_len
        self.out_seq_len = out_seq_len
        self.n_hidden = n_hidden
        self.vocab_size = vocab_size

        with self.name_scope():
            self.encoder = rnn.LSTMCell(hidden_size = n_hidden)
            self.decoder = rnn.LSTMCell(hidden_size = n_hidden)
            self.attn_weight = nn.Dense(self.in_seq_len, in_units = self.in_seq_len)
            self.batchnorm = nn.BatchNorm(axis = 2)
            self.dense = nn.Dense(self.vocab_size, flatten = False)

    def forward(self, inputs, outputs):
        self.batch_size = inputs.shape[0]
        enout, (next_h, next_c) = self.encoder.unroll(inputs = inputs \
                                                      , length = self.in_seq_len \
                                                      , merge_outputs = True)
        # enout: (n_batch * time_step * n_hidden), next_h, next_c: (n_batch * n_hidden)
        for i in range(self.out_seq_len):
            # For each time step, caclculate context for attention
            _n_h = next_h.expand_dims(axis = 2)
            ####### Attention part: To get context vector at jth point of output sequence
            score_i = gemm2(enout, next_h.expand_dims(axis = 2))  # n_batch * time_step * 1
            alpha_i = nd.softmax(self.attn_weight(score_i)) # n_batch * time_step
            alpha_expand = alpha_i.expand_dims(2) # (n_batch * 1 * time_step)
            alpha_expand = nd.repeat(alpha_expand,repeats= self.n_hidden, axis=2) # n_batch * time_step * n_hidden
            context = nd.multiply(alpha_expand, enout) # n_batch * time_step * n_hidden
            context = nd.sum(context, axis = 1) # n_batch * n_hidden
            _in = nd.concat(outputs[:, i, :], context)
            deout, (next_h, next_c) = self.decoder(_in, [next_h, next_c],)
            if i == 0:
                deouts = deout
            else:
                deouts = nd.concat(deouts, deout, dim = 1)
        deouts = nd.reshape(deouts, (-1, self.out_seq_len, self.n_hidden))
        deouts = self.batchnorm(deouts)
        deouts_fc = self.dense(deouts)
        return deouts_fc
```

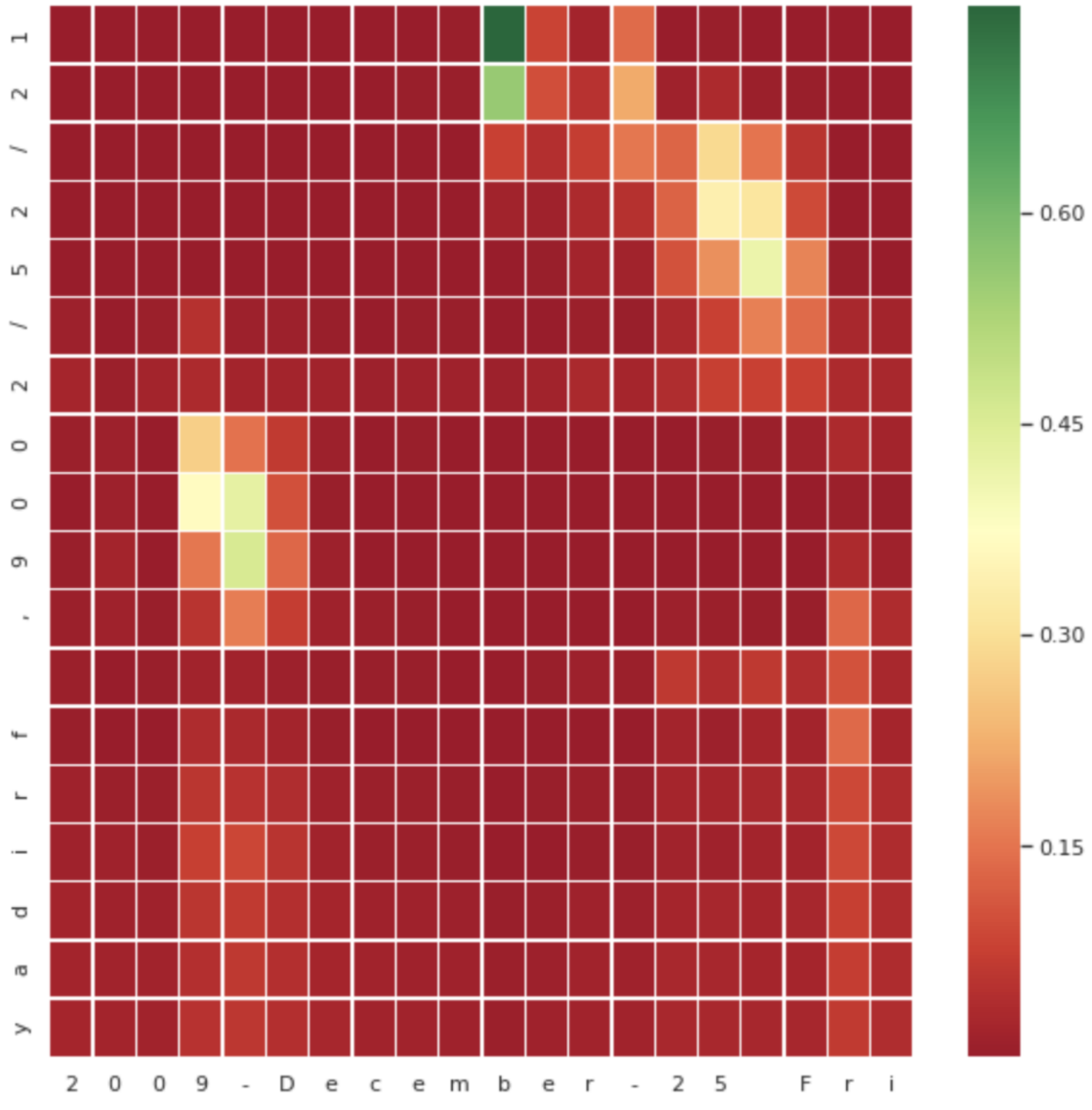

0
5
/
1
4
/
2
0
0
5
-
0
0
5
-
M
a
y
-
1
4
S
a
t
0.075
0.060
0.045
0.030

Generalized dot product attention

```
class alignment(gluon.HybridBlock):
    def __init__(self, n_hidden, **args):
        super(alignment, self).__init__(**args)
        with self.name_scope():
            self.weight = self.params.get('weight', shape = (n_hidden, n_hidden), allow_deferred_init = True)

    def hybrid_forward(self, F, inputs, output, weight):
        _s = F.dot(inputs, weight)
        return gemm2(_s, output)
```

Generalized dot product attention

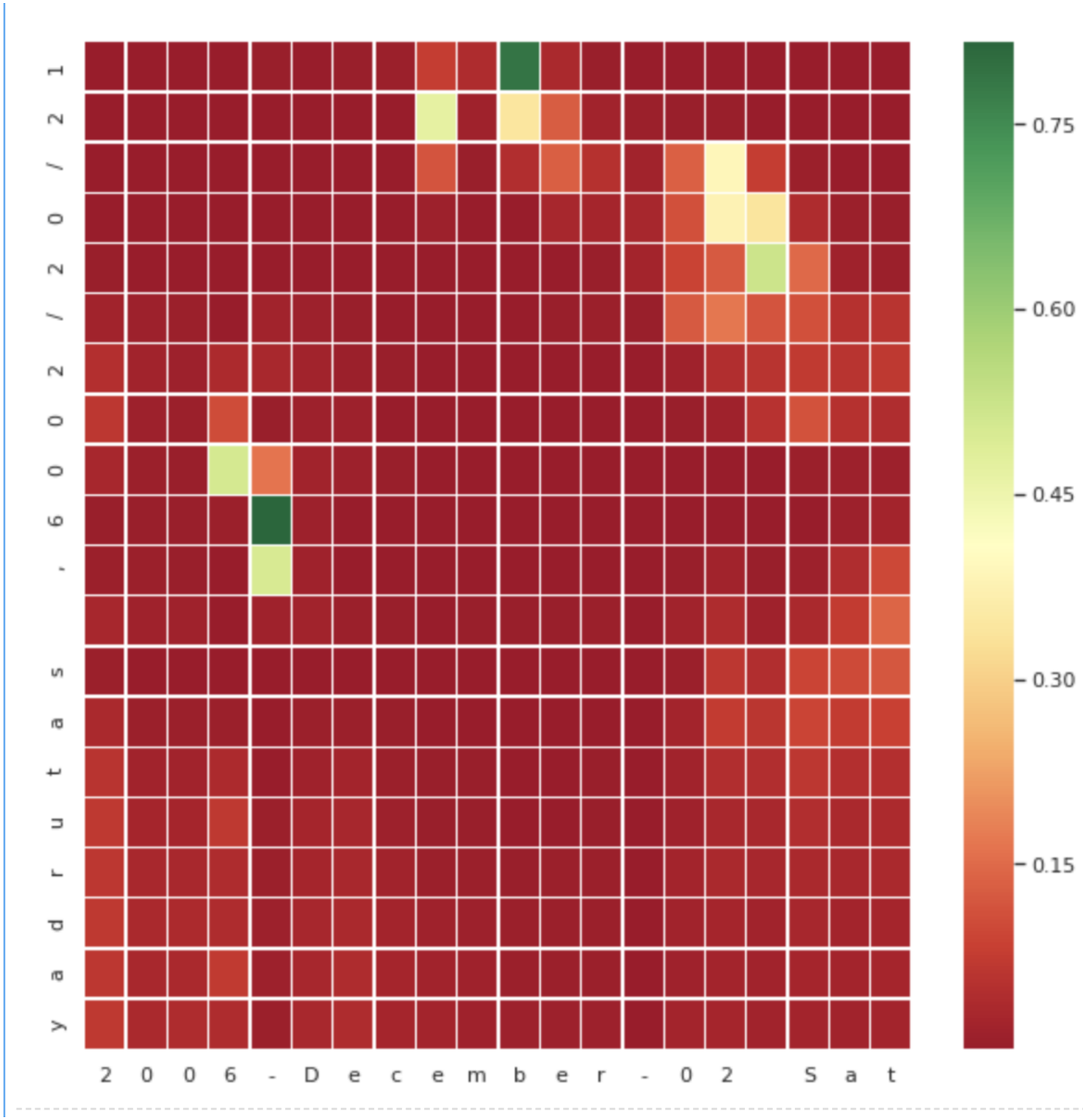
```
class format_translator(gluon.Block):
    def __init__(self, n_hidden, in_seq_len, out_seq_len, vocab_size, ctx, **kwargs):
        super(format_translator, self).__init__(**kwargs)
        self.in_seq_len = in_seq_len
        self.out_seq_len = out_seq_len
        self.n_hidden = n_hidden
        self.vocab_size = vocab_size
        self.ctx = ctx

        with self.name_scope():
            self.encoder = rnn.LSTMCell(hidden_size = n_hidden)
            self.decoder = rnn.LSTMCell(hidden_size = n_hidden)
            self.alignment = alignment(n_hidden)
            self.attn_weight = nn.Dense(self.in_seq_len, in_units = self.in_seq_len)
            self.batchnorm = nn.BatchNorm(axis = 2)
            self.dense = nn.Dense(self.vocab_size, flatten = False)

    def forward(self, inputs, outputs):
        self.batch_size = inputs.shape[0]
        enout, (next_h, next_c) = self.encoder.unroll(inputs = inputs, length = self.in_seq_len, merge_outputs = True)

        for i in range(self.out_seq_len):
            # For each time step, caclculate context for attention
            # Use enout(batch_size * in_seq_len * n_hidden)
            _n_h = next_h.expand_dims(axis = 2)
            score_i = self.alignment(enout, _n_h)
            alpha_i = nd.softmax(self.attn_weight(score_i))
            alpha_expand = alpha_i.expand_dims(2) # (n_batch * in_seq_len * n_hidden)
            alpha_expand = nd.repeat(alpha_expand,repeats= self.n_hidden, axis=2) # n_batch * in_seq_len * n_hidden
            context = nd.multiply(alpha_expand, enout) # n_batch * in_seq_len * n_hidden
            context = nd.sum(context, axis = 1) # n_batch * n_hidden
            _in = nd.concat(outputs[:, i, :], context)
            deout, (next_h, next_c) = self.decoder(_in, [next_h, next_c],)
            if i == 0:
                deouts = deout
            else:
                deouts = nd.concat(deouts, deout, dim = 1)
        deouts = nd.reshape(deouts, (-1, self.out_seq_len, self.n_hidden))
        deouts = self.batchnorm(deouts)
        deouts_fc = self.dense(deouts)
        return deouts_fc
```


0.60
0.45
0.30
0.15
2009-December-25 Friday

Additive attention

```
class pattern_matcher(gluon.Block):
    def __init__(self, n_hidden, in_seq_len, out_seq_len, vocab_size, ctx, **kwargs):
        super(pattern_matcher, self).__init__(**kwargs)
        self.in_seq_len = in_seq_len
        self.out_seq_len = out_seq_len
        self.n_hidden = n_hidden
        self.vocab_size = vocab_size
        self.ctx = ctx
        with self.name_scope():
            self.encoder = rnn.LSTMCell(hidden_size = n_hidden)
            self.decoder = rnn.LSTMCell(hidden_size = n_hidden)
            self.attn_w = nn.Dense(self.n_hidden)
            self.attn_v = nn.Dense(self.in_seq_len)
            self.batchnorm = nn.BatchNorm(axis = 2)
            self.dense = nn.Dense(self.vocab_size, flatten = False)

    def forward(self, inputs, outputs):
        self.batch_size = inputs.shape[0]
        enout, (next_h, next_c) = self.encoder.unroll(inputs = inputs, length = self.in_seq_len, merge_outputs = True)
        for i in range(self.out_seq_len):
            _n_h = next_h.expand_dims(axis = 1)
            ####### Attention part: To get context vector at jth point of output sequence
            _n_h_s = nd.repeat(_n_h, repeats = enout.shape[1], axis = 1)
            _in_attn = nd.concat(enout, _n_h_s, dim = 2)
            _align = self.attn_w(_in_attn)
            _align = nd.tanh(_align)
            score_i = self.attn_v(_align)
            alpha_i = nd.softmax(score_i) # (n_batch * in_seq_len)
            alpha_expand = alpha_i.expand_dims(2) # (n_batch * in_seq_len * n_hidden)
            alpha_expand = nd.repeat(alpha_expand,repeats= self.n_hidden, axis=2) # n_batch * time step * n_hidden
            context = nd.multiply(alpha_expand, enout)
            context = nd.sum(context, axis = 1) # n_batch * n_hidden
            _in = nd.concat(outputs[:, i, :], context)
            deout, (next_h, next_c) = self.decoder(_in, [next_h, next_c],)
            if i == 0:
                deouts = deout
            else:
                deouts = nd.concat(deouts, deout, dim = 1)
        deouts = nd.reshape(deouts, (-1, self.out_seq_len, self.n_hidden))
        deouts = self.batchnorm(deouts)
        deouts_fc = self.dense(deouts)
        return deouts_fc
```

Additive attention



Reference

Websites

https://www.weibo.com/ttarticle/p/show?id=2309403960559677656612
Deep Learning book, Chapter 10 Sequence Modeling: Recurrent and Recursive Nets
https://isaacchanghau.github.io/post/lstm-gru-formula/

Papers

http://docs.likejazz.com/cnn-text-classification-tf/
https://arxiv.org/abs/1408.5882
https://arxiv.org/abs/1404.2188
https://arxiv.org/pdf/1508.04025.pdf
https://arxiv.org/pdf/1409.0473.pdf