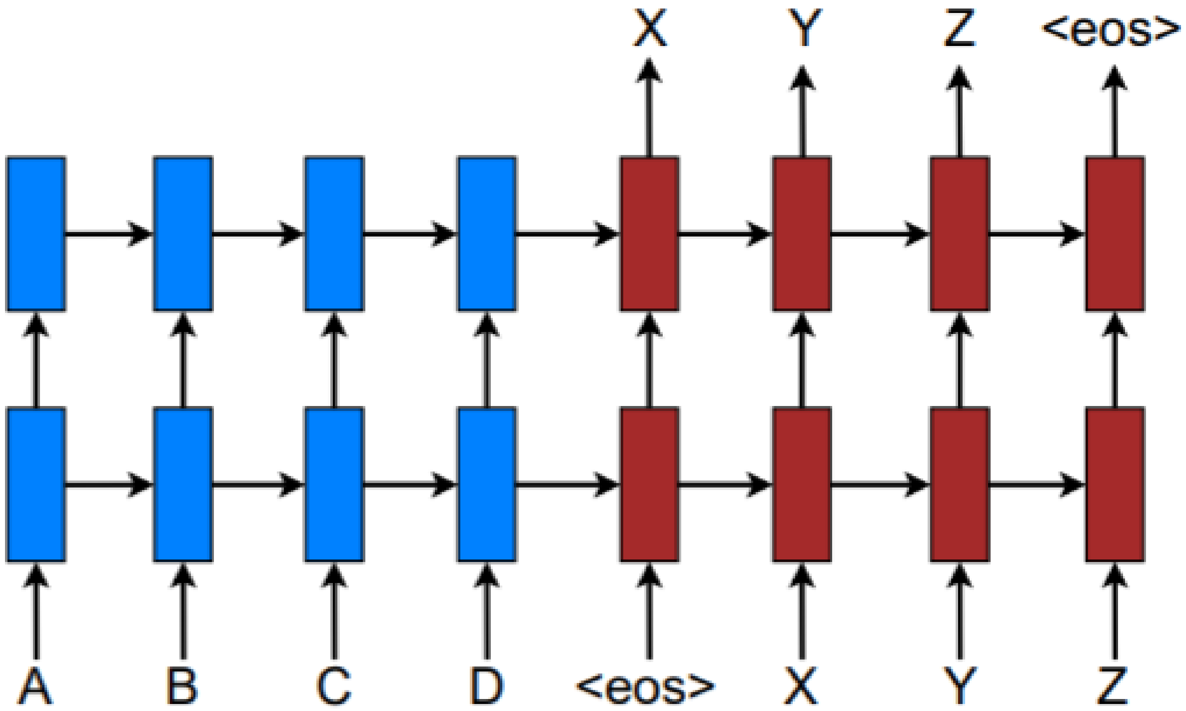
Effective Approaches to Attention-based Neural Machine Translation

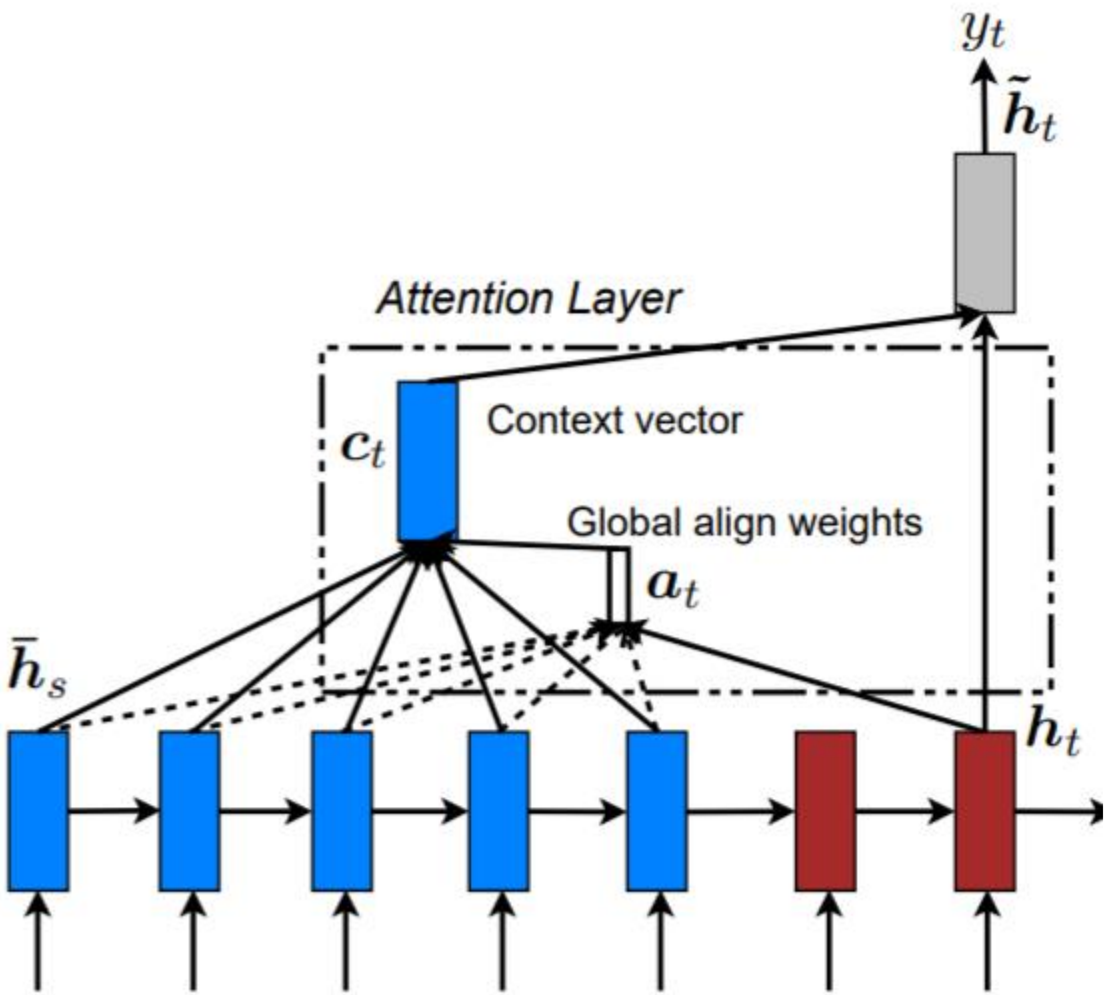
1. Introduction

- NML (neural machine translation)
  - 기존 ML에 비해 강력하고 간단하고 쉬운 모델



1. Introduction

- Attention



2. NMT

- X1, X2, … ,Xn : source sentence
- Y1, Y2, … ,Yn : target sentence

$$\log p(y|x) = \sum_{j=1}^{m} \log p\left(y_j | y_{<j}, \boldsymbol{s}\right)$$

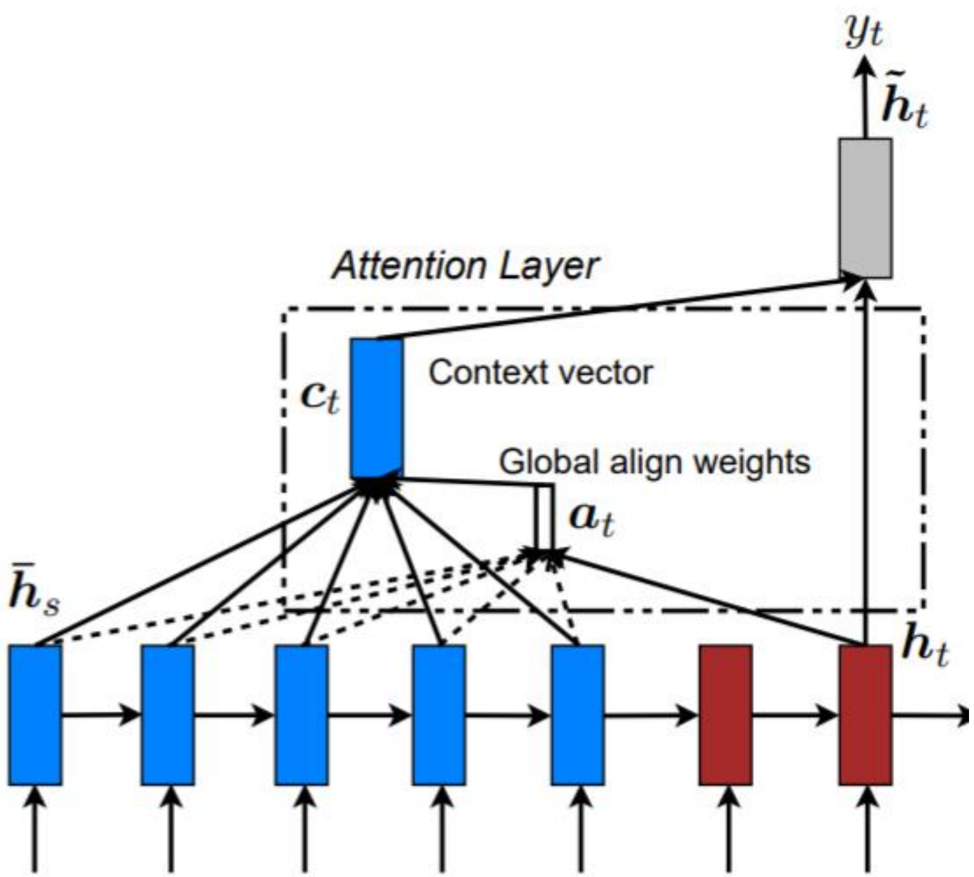
S: 초기 상태, x

$$p\left(y_j | y_{<j}, \boldsymbol{s}\right) = \operatorname{softmax}\left(g\left(\boldsymbol{h}_j\right)\right)$$

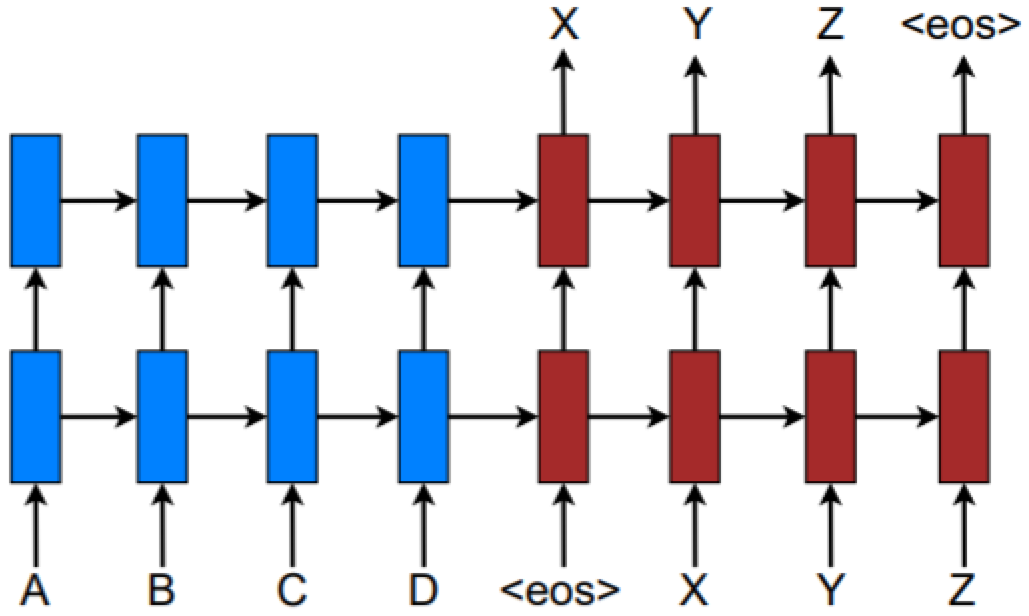
g: RNN output(h_j) > 단어

$$J_t = \sum_{(x,y)\in\mathbb{D}} -\log p(y|x)$$

3. Attention

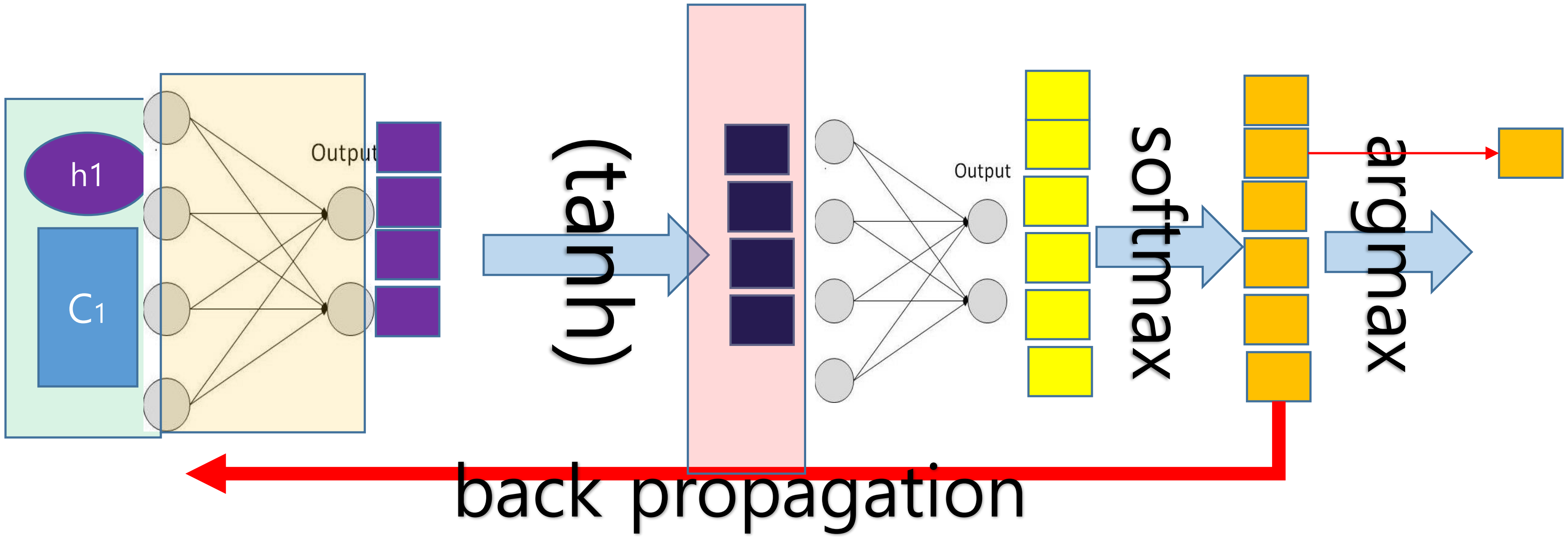


y_t
$\tilde{h}_t$
Attention Layer
Context vector
c_t
Global align weights
a_t
$\bar{h}_s$
h_t

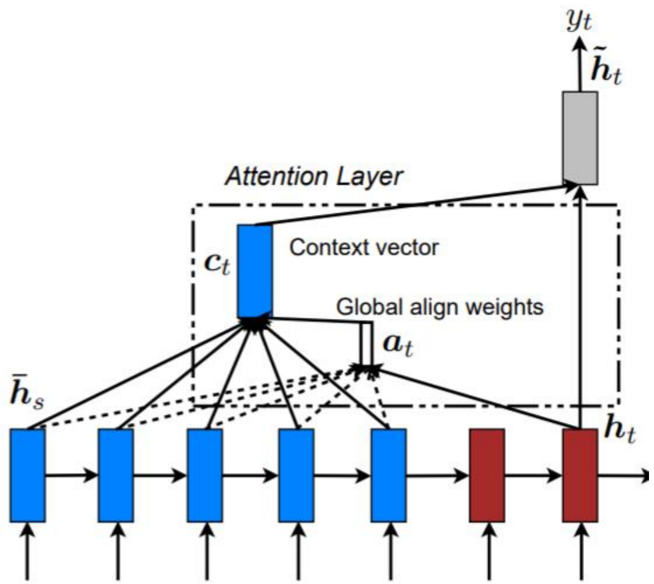


X Y Z <eos>
A B C D <eos> X Y Z

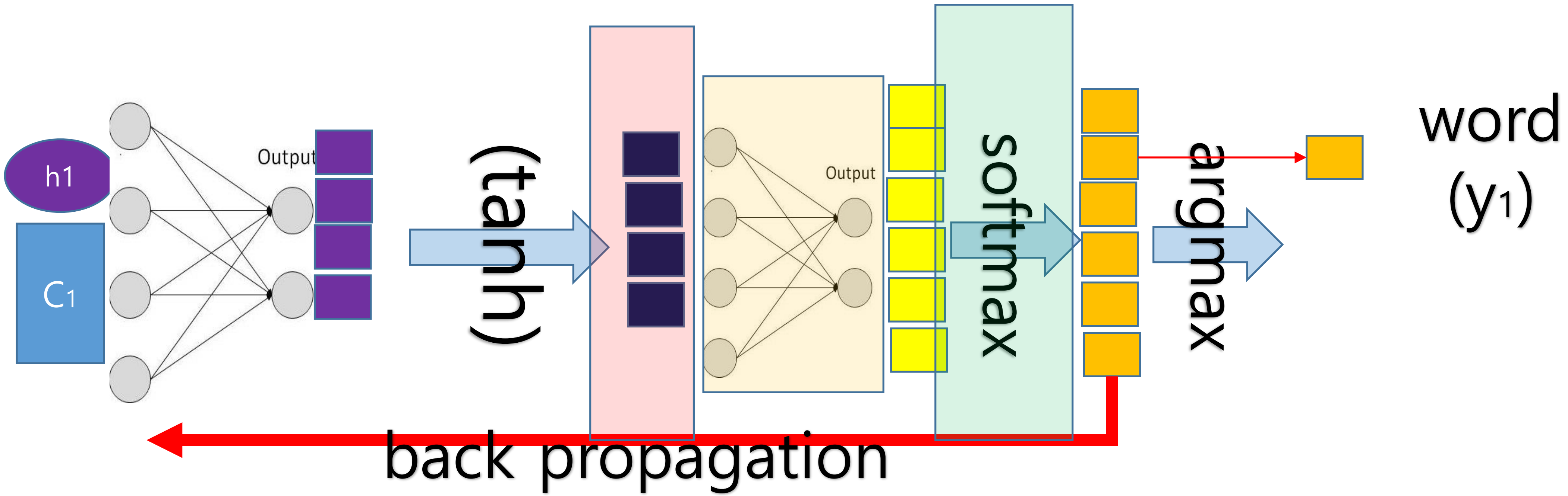
3. Attention

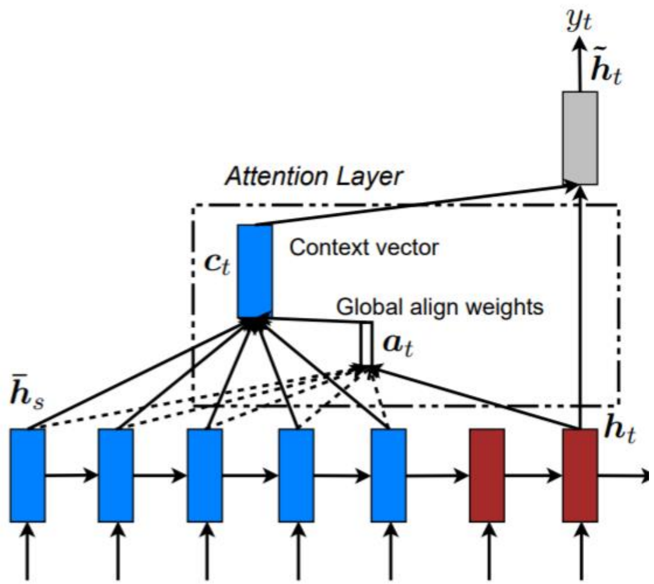


$$\tilde{h}_t = \tanh(W_c[c_t;h_t])$$



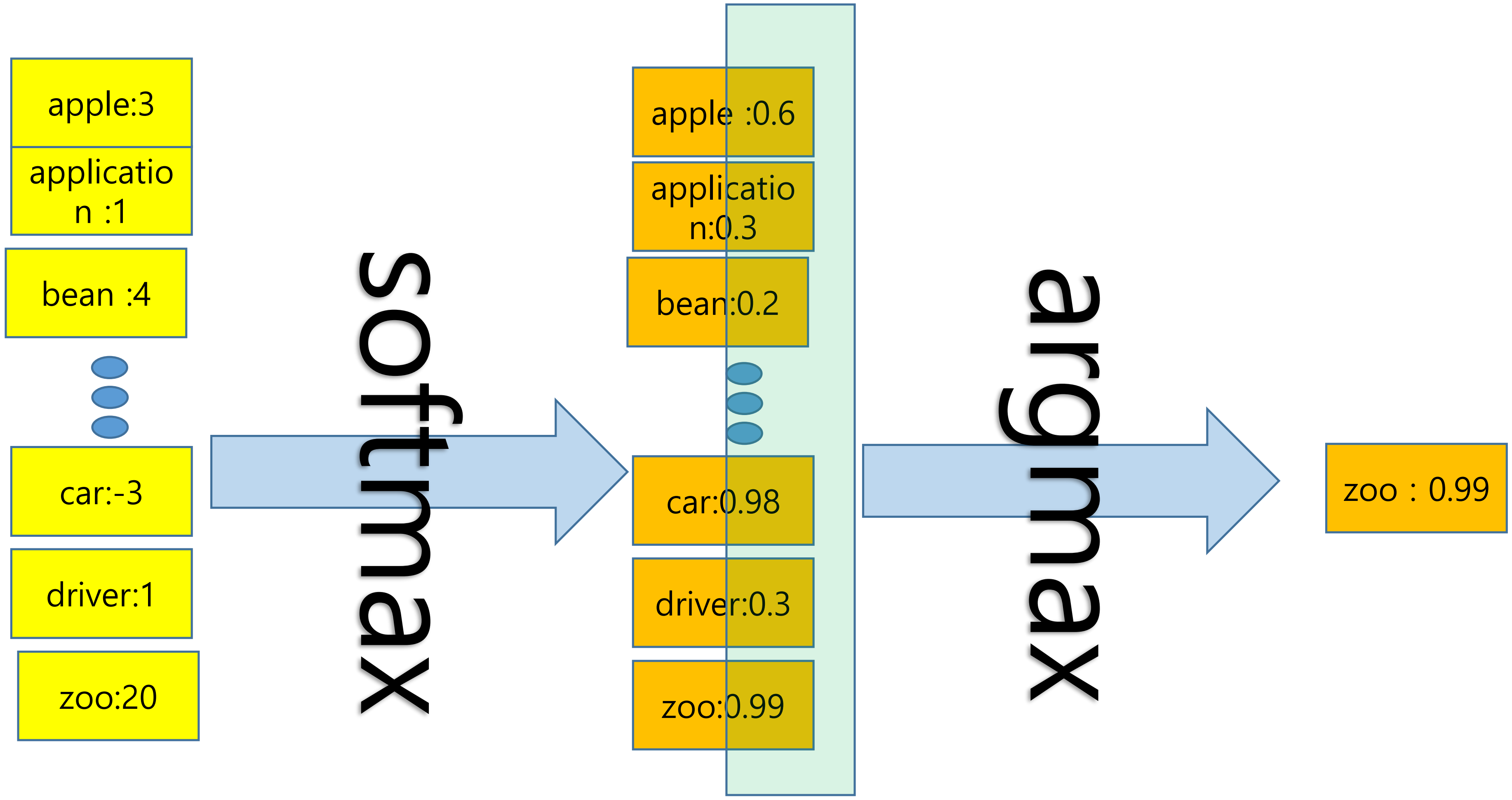
3. Attention



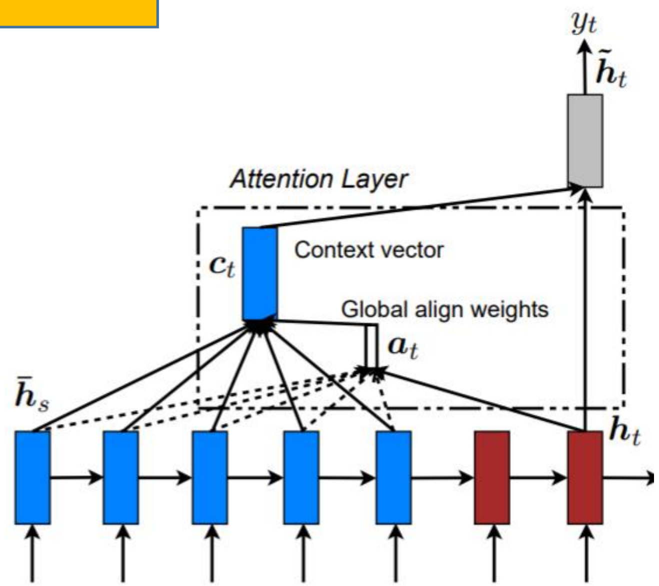


$$p(y_t|y_{<t}, x) = \mathrm{softmax}(W_s\tilde{h}_t)$$

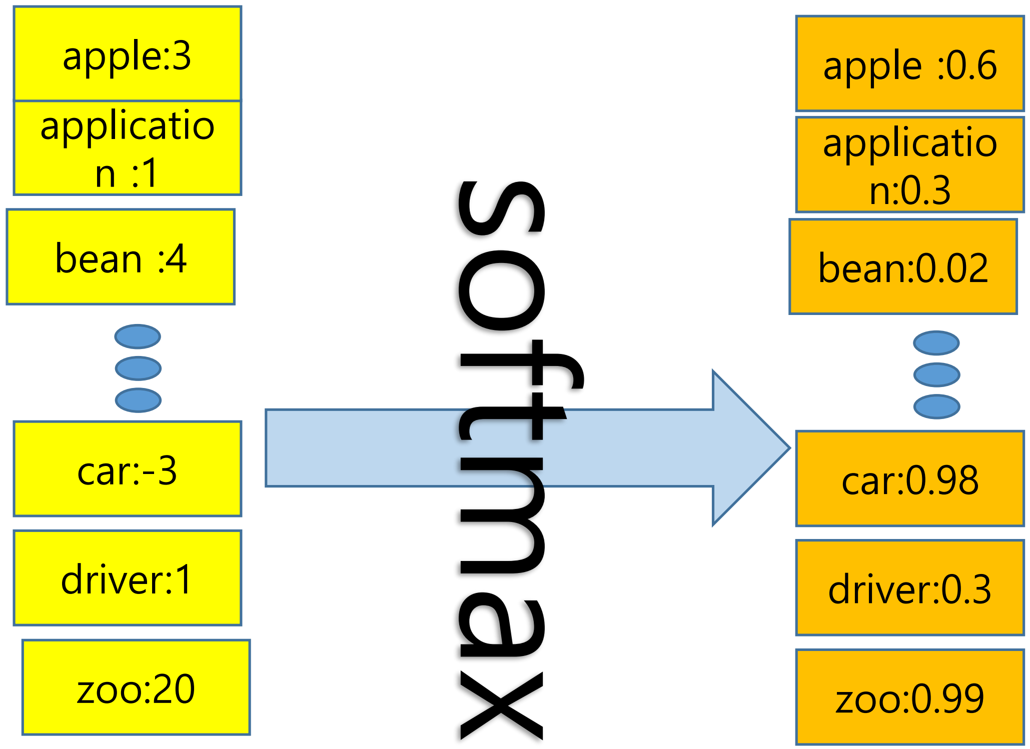
3. Attention - using



$$\boxed{p(y_t|y_{<t}, x)} = \mathrm{softmax}(\boldsymbol{W}_s\tilde{\boldsymbol{h}}_t)$$



3. Attention - training



$$J_t = \sum_{(x,y)\in\mathbb{D}} -\log p(y|x)$$

if Ground Truth : car

$-\log p(y|x)$ =-log(0.98)
=0.008

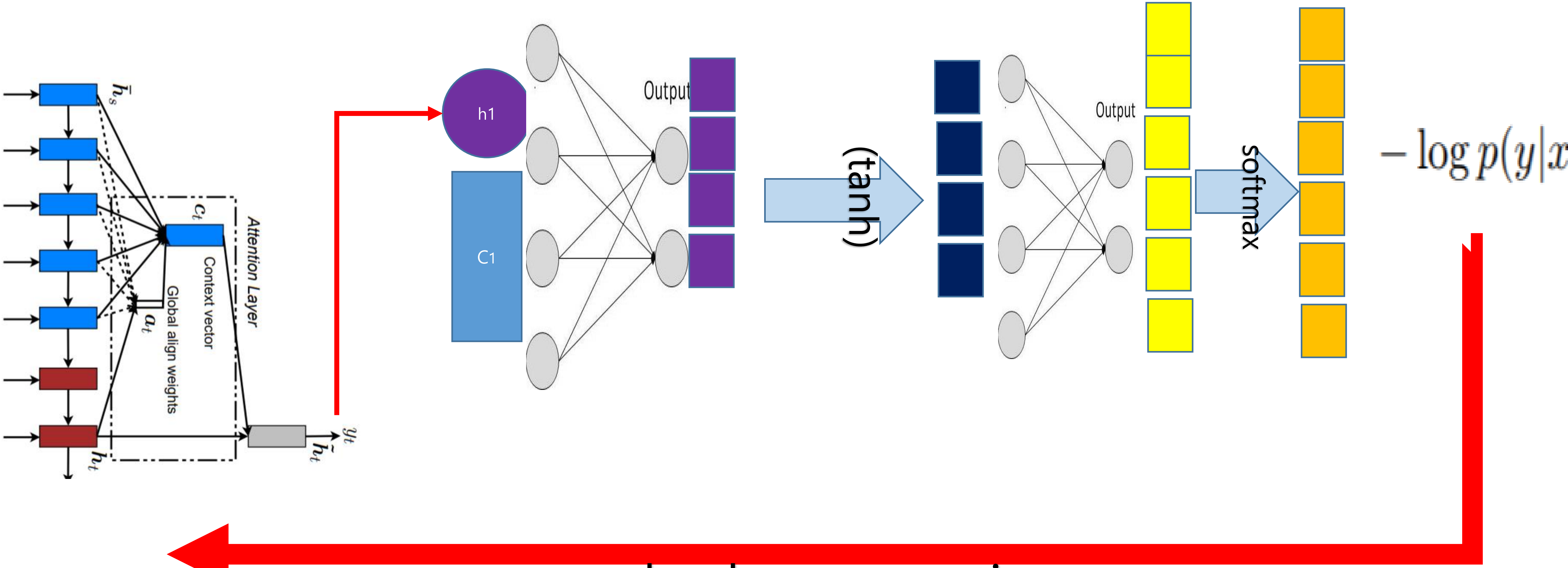
if Ground Truth : application

$-\log p(y|x)$ =-log(0.6)
=0.22

if Ground Truth : bean

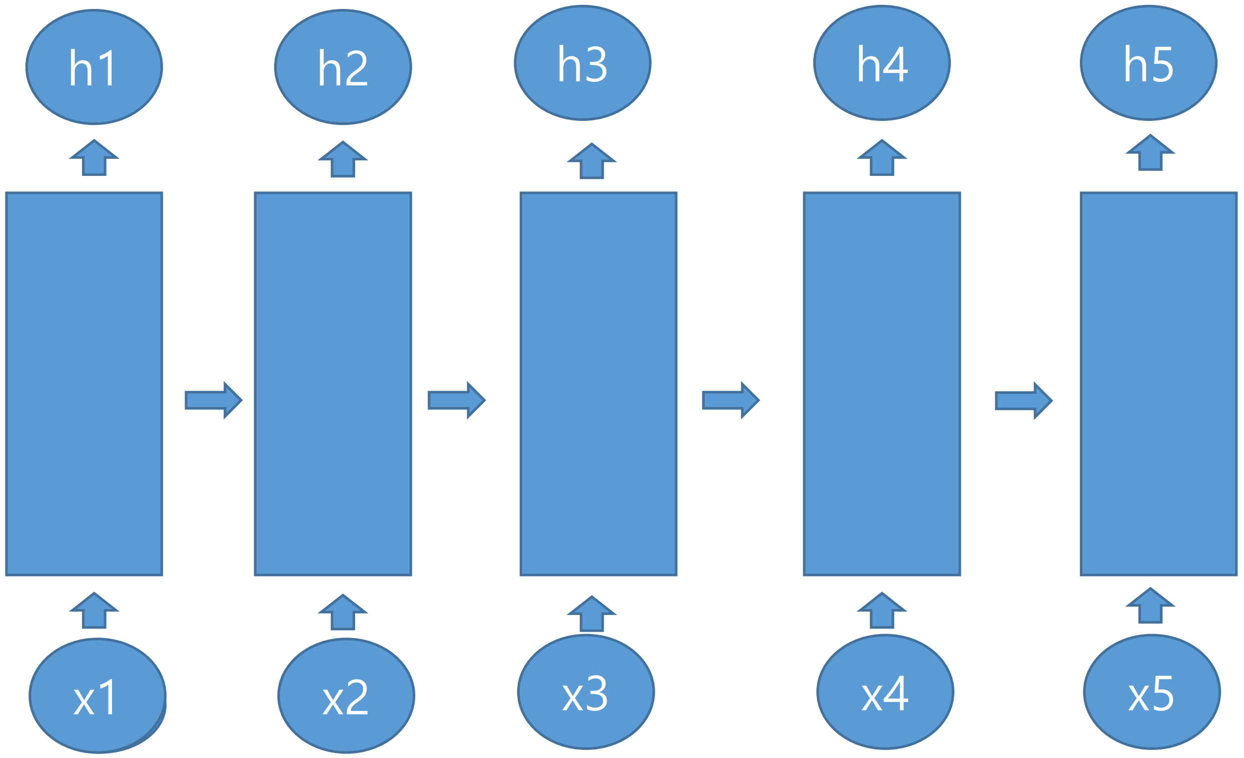
$-\log p(y|x)$ =-log(0.02)
=0.22

3. Attention - training



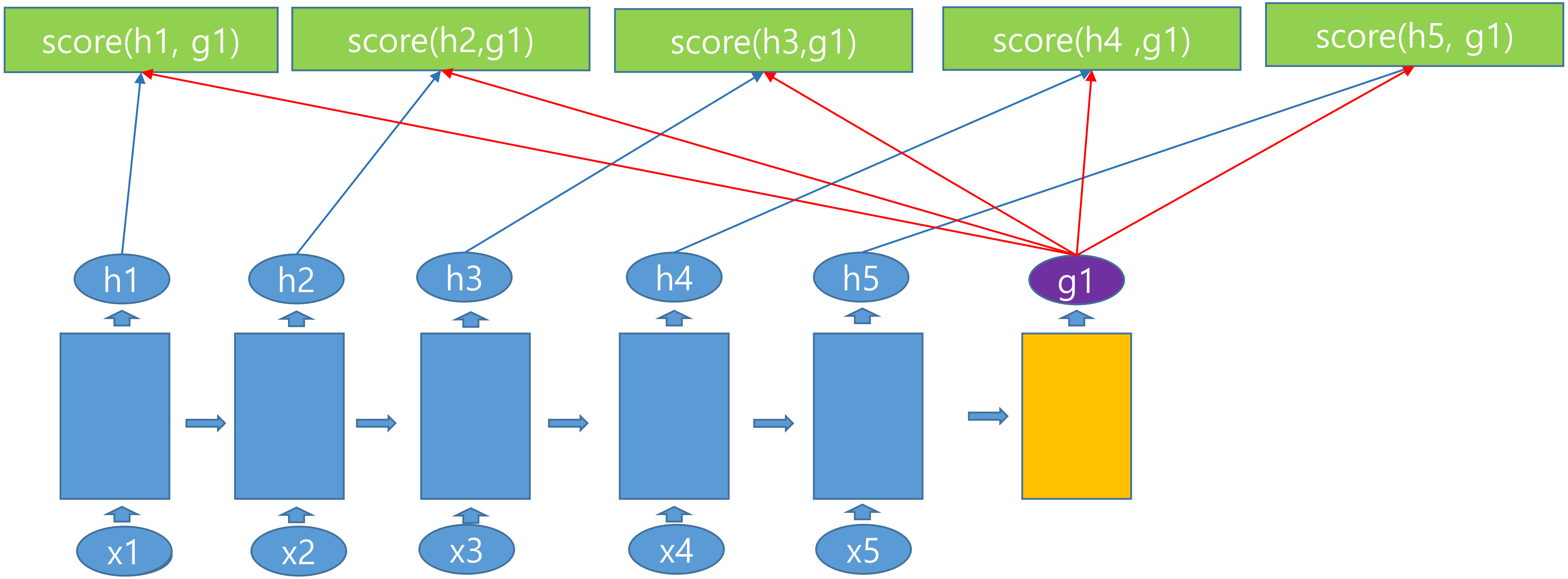
back propagation

3.1 global Attention
-Encoding



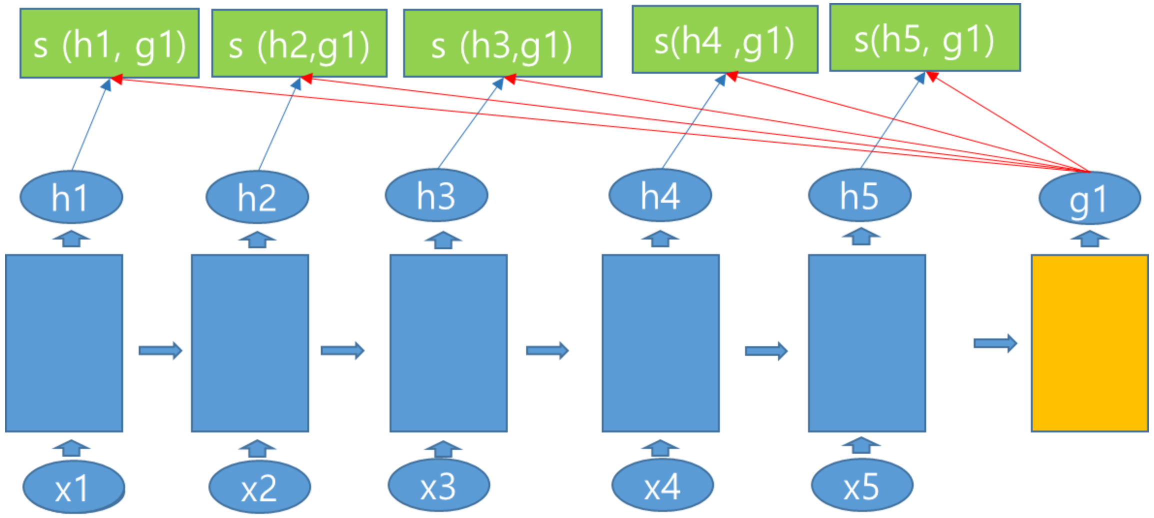
3.1 global Attention
- score

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



3.1 global Attention
-alignment vector a, softmax

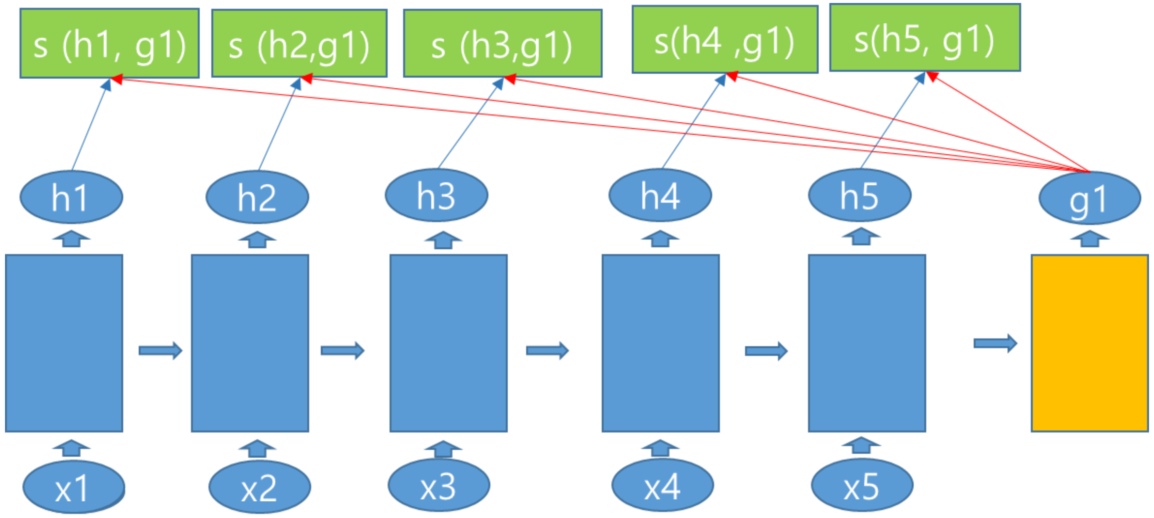
$$a_1(1) = \frac{\exp\ s(h1\ ,g1)}{\exp\ s(h1\ ,g1) + \exp\ s(h2\ ,g1) + \exp\ s(h3\ ,g1) + \exp\ s(h4\ ,g1) + \exp\ s(h5\ ,g1)}$$



$$\boldsymbol{a}_t(s) = \operatorname{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)$$
$$= \frac{\exp\left(\operatorname{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'} \exp\left(\operatorname{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)}$$

3.1 global Attention
-alignment vector a, softmax

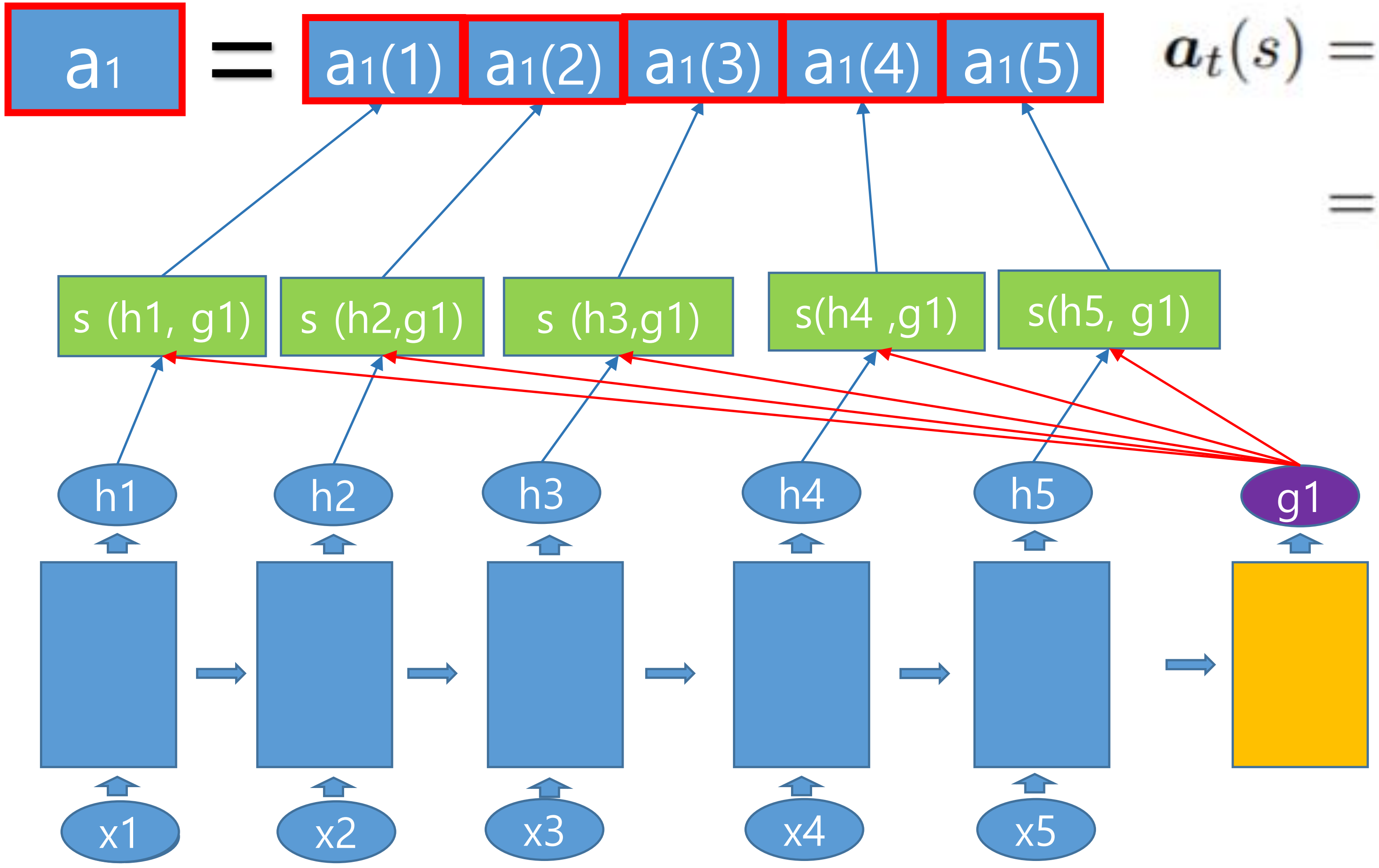
$$a_1(2) = \frac{s(h2, g1)}{s(h1, g1) + s(h2, g1) + s(h3, g1) + s(h4, g1) + s(h5, g1)}$$



$$\boldsymbol{a}_t(s) = \operatorname{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)$$

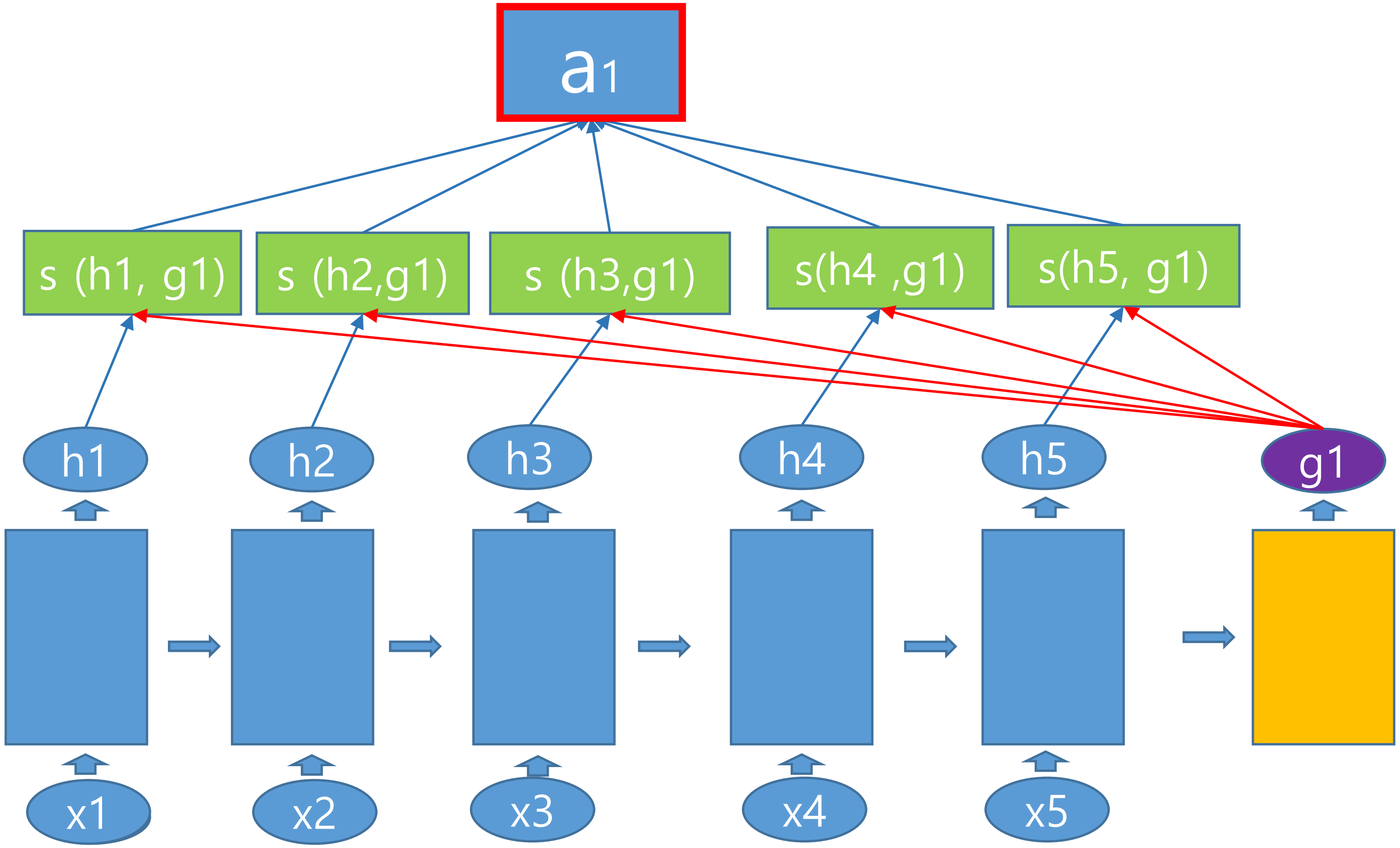
$$= \frac{\exp\left(\operatorname{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'} \exp\left(\operatorname{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)}$$

3.1 global Attention
alignment vector a



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

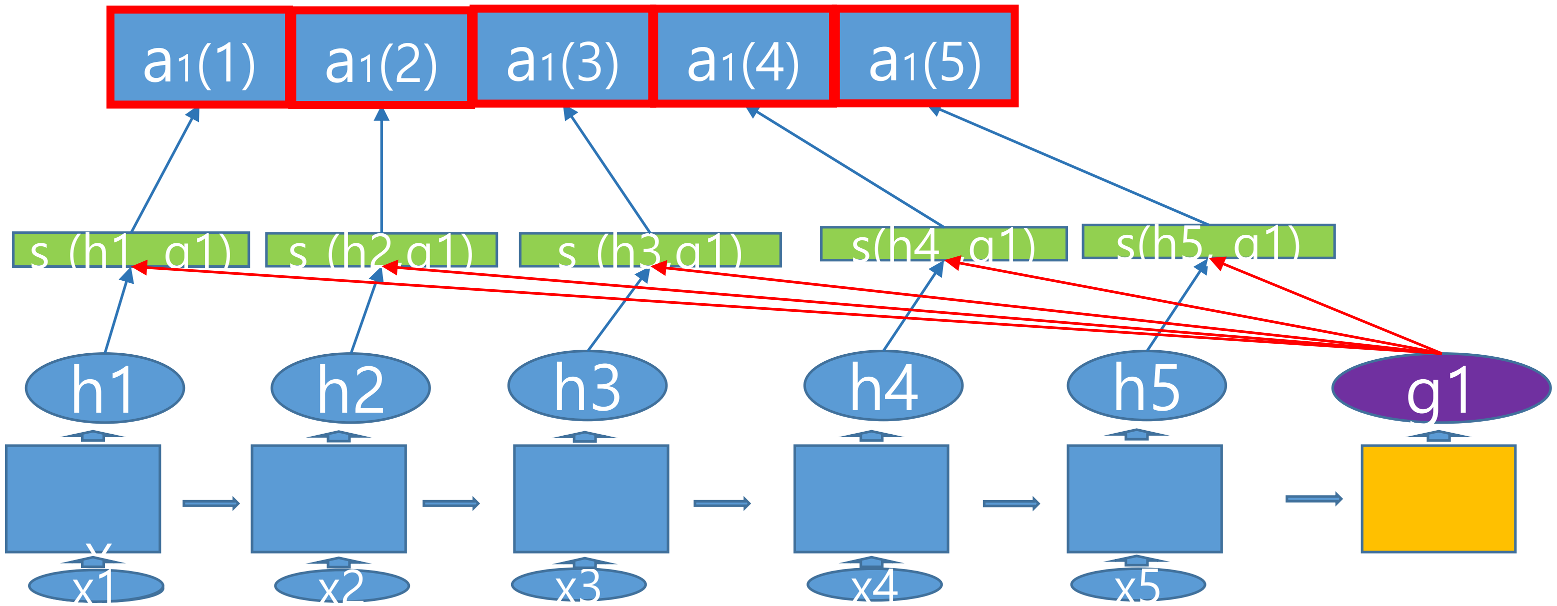
3.1 global Attention



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

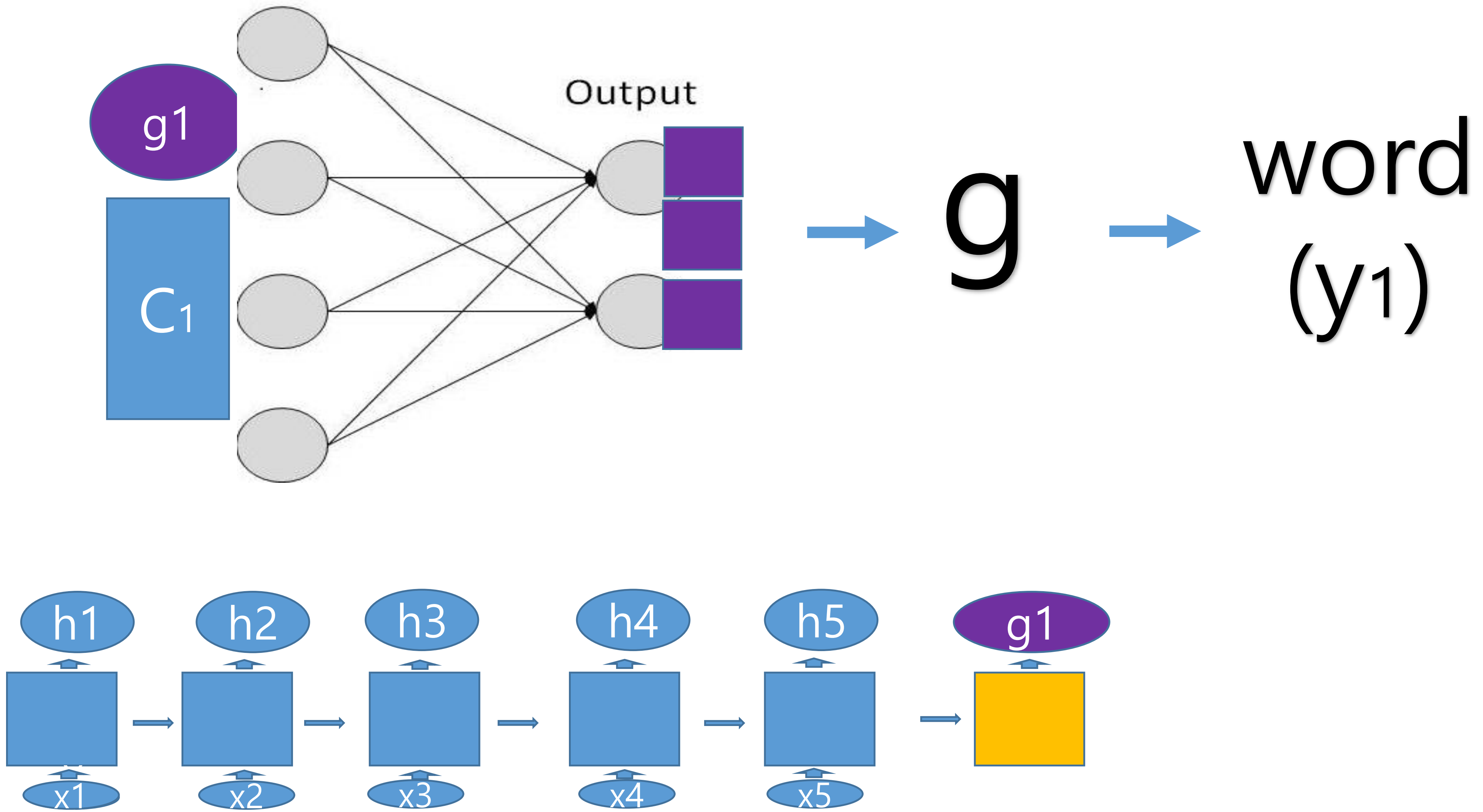
3.1 global Attention
context vector

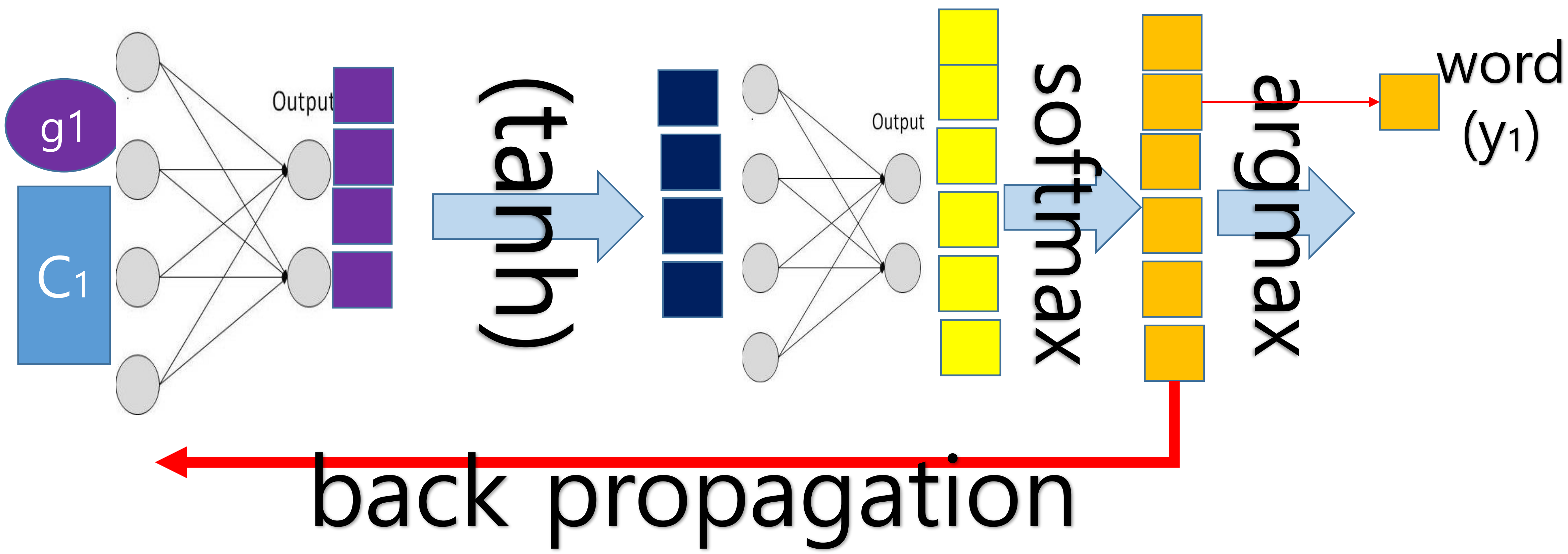
$$a_1(1)\,h1 + a_1(2)\,h2 + a_1(3)\,h3 + a_1(4)\,h4 + a_1(5)\,h5 = C_1 \text{ or } C_{g1}$$

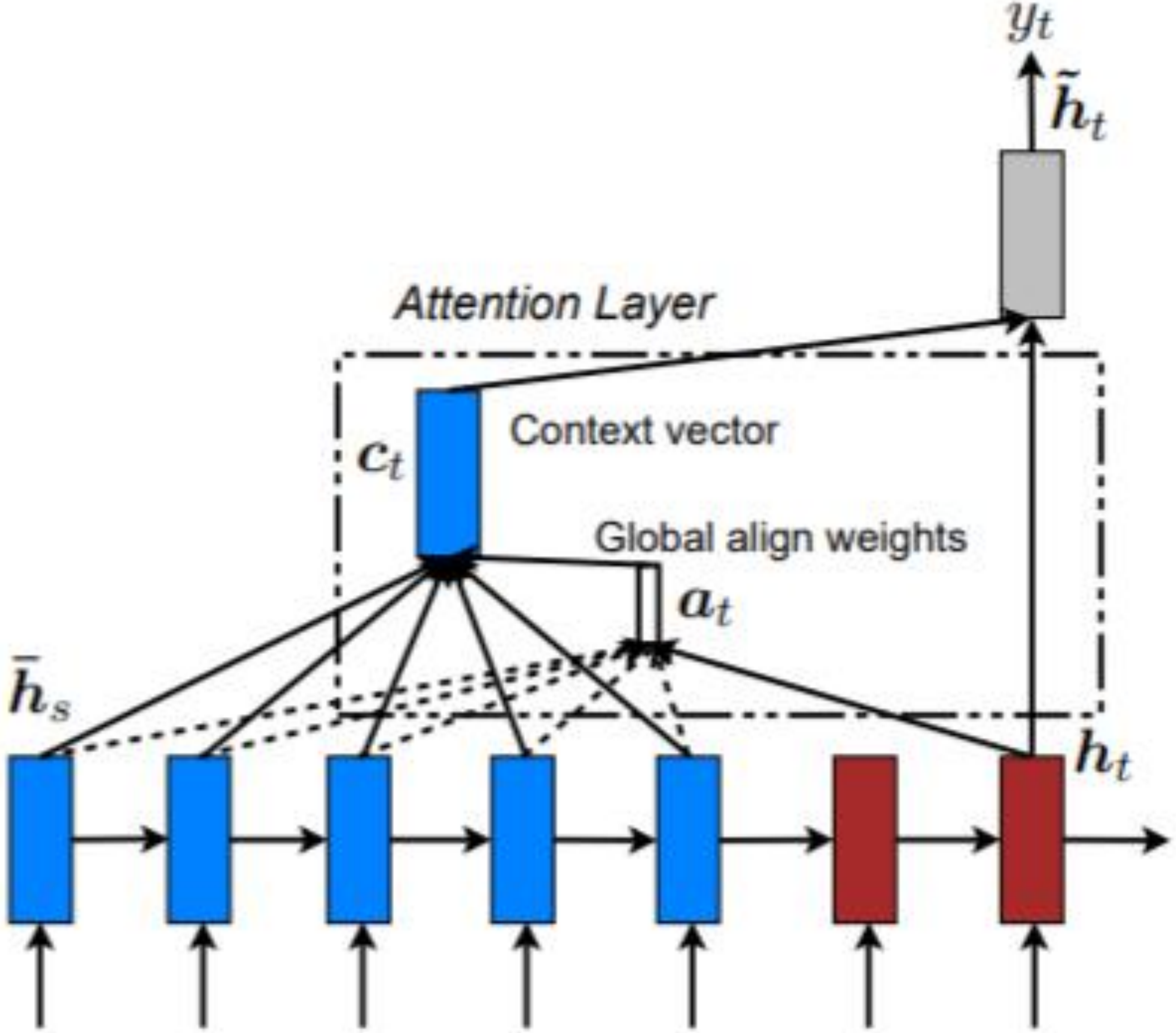


가중치가 a인 h들의 가중평균
= context vector

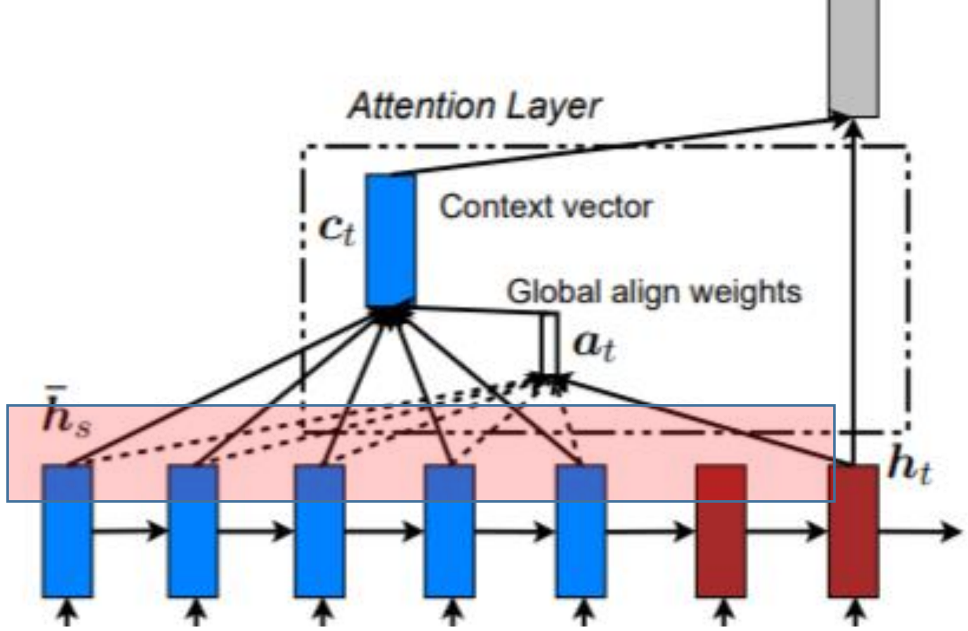
3.1 global Attention-prediction



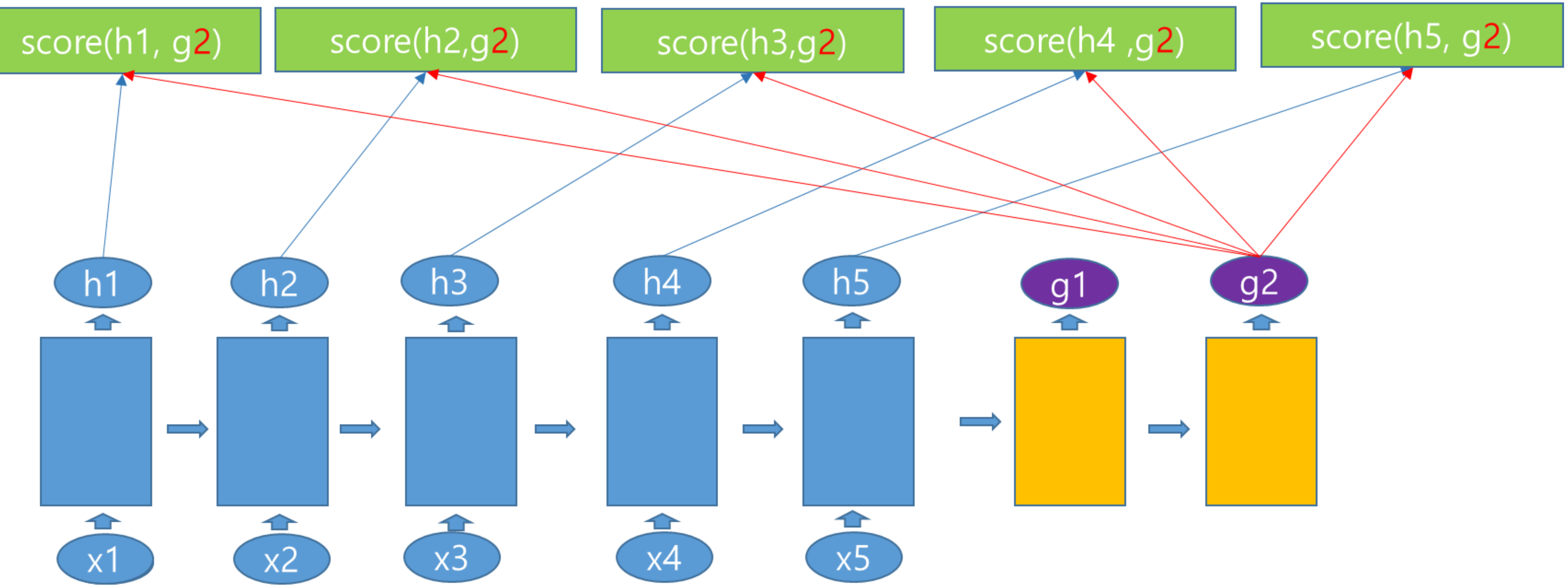

g1
C1
Output
(tanh)
Output
softmax
argmax
word
(y1)
back propagation



y_t
$\tilde{h}_t$
Attention Layer
c_t
Context vector
Global align weights
a_t
$\bar{h}_s$
h_t

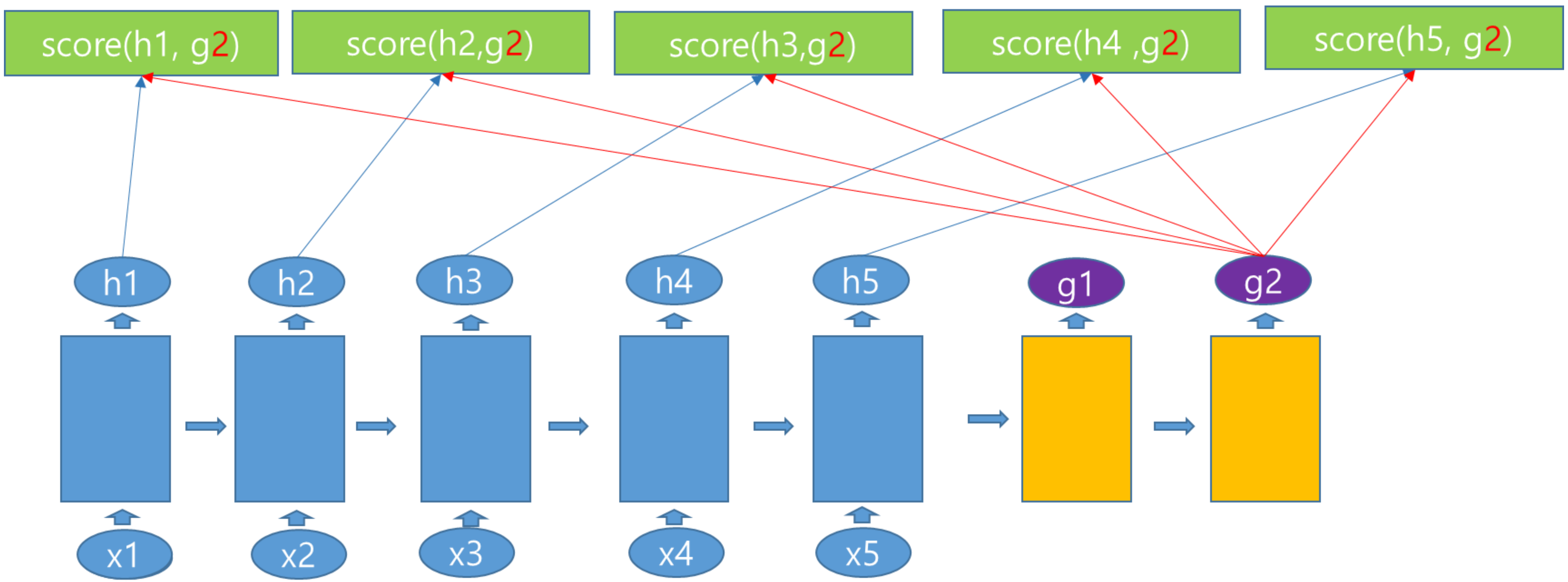


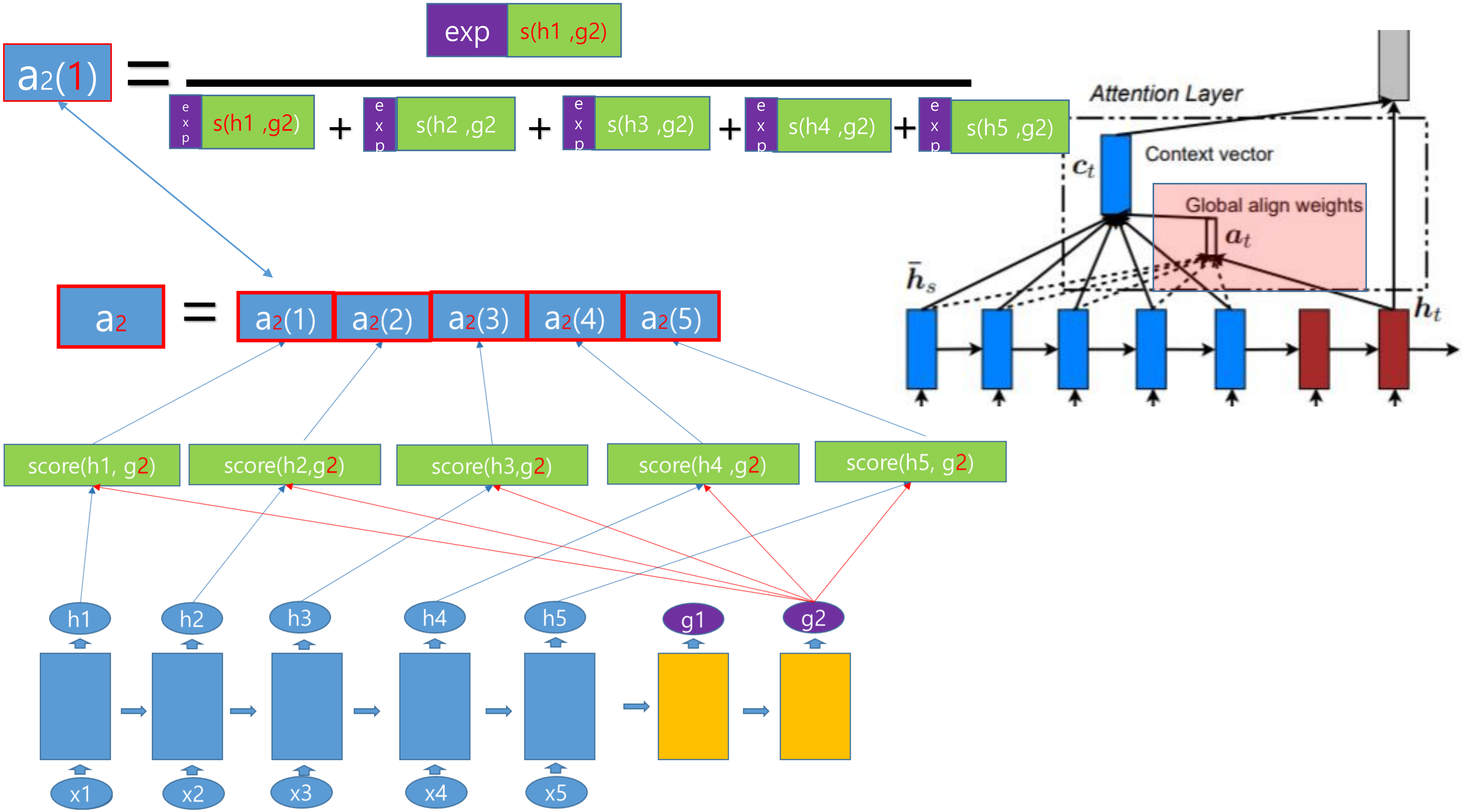
Attention Layer
c_t
Context vector
Global align weights
a_t
$\bar{h}_s$
h_t



score(h1, g2)
score(h2,g2)
score(h3,g2)
score(h4 ,g2)
score(h5, g2)
h1
h2
h3
h4
h5
g1
g2
x1
x2
x3
x4
x5

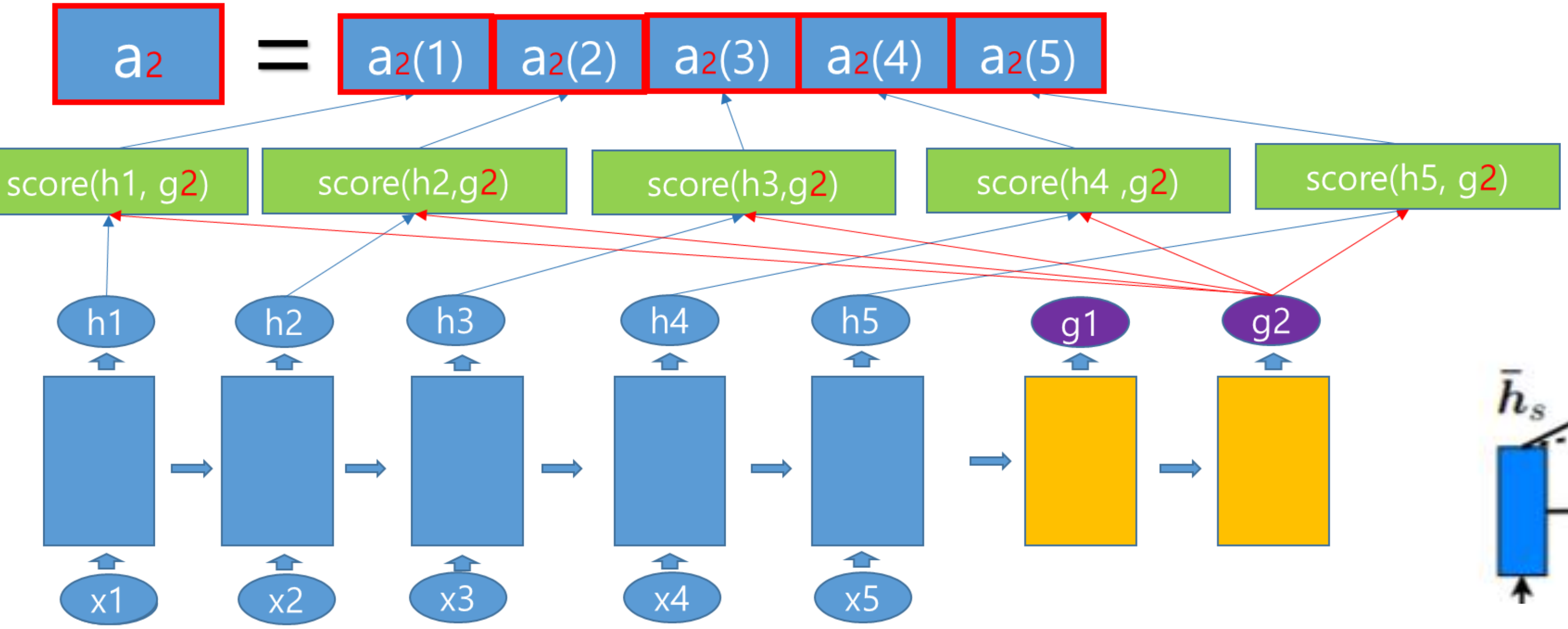
$$a_2(1) = \frac{\exp\ s(h1\ ,g2)}{\exp\ s(h1\ ,g2) + \exp\ s(h2\ ,g2 + \exp\ s(h3\ ,g2) + \exp\ s(h4\ ,g2) + \exp\ s(h5\ ,g2)}$$

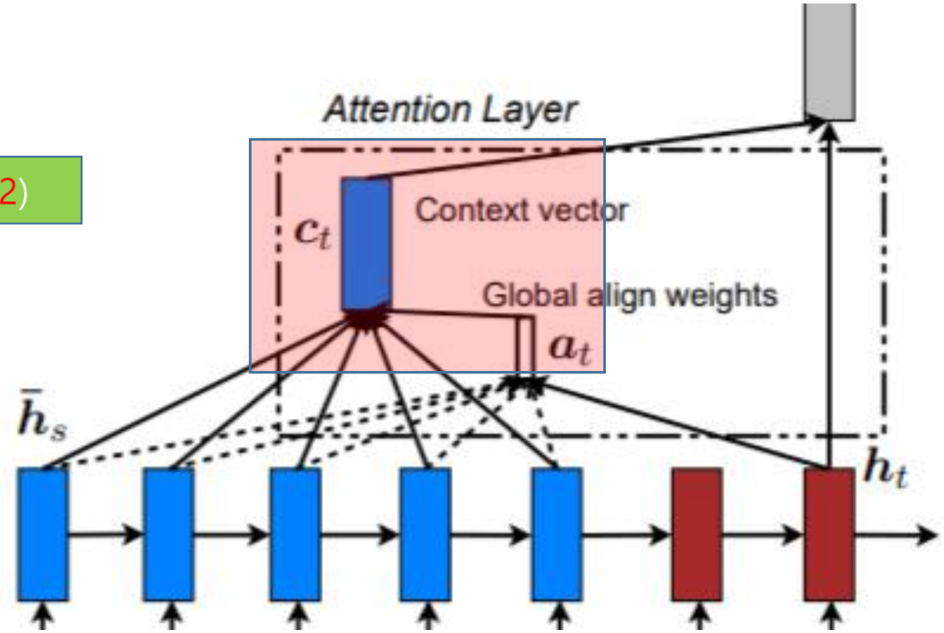


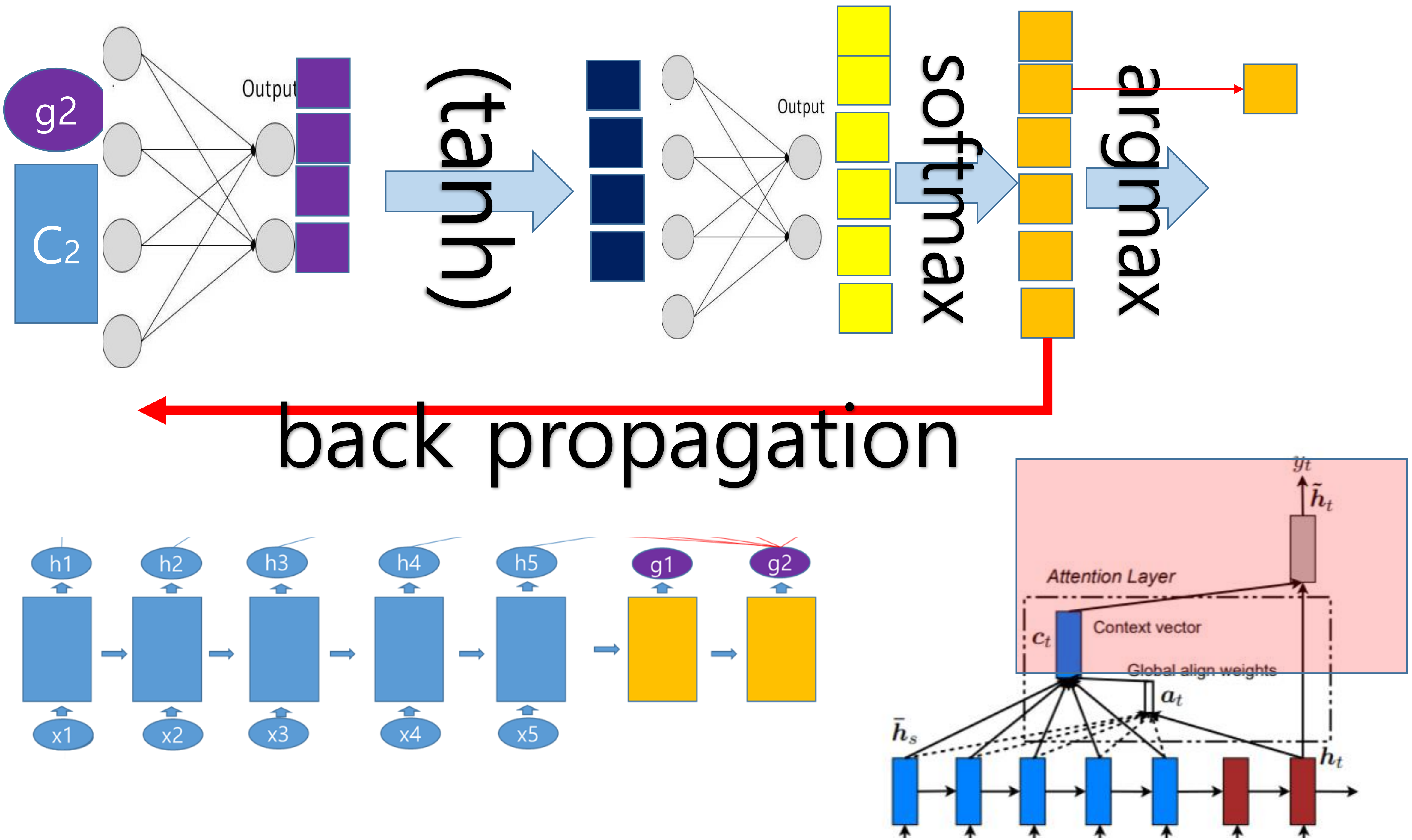


a2(1) =
exp s(h1 ,g2)
exp s(h1 ,g2) + exp s(h2 ,g2 + exp s(h3 ,g2) + exp s(h4 ,g2) + exp s(h5 ,g2)
a2 = a2(1) a2(2) a2(3) a2(4) a2(5)
score(h1, g2)
score(h2,g2)
score(h3,g2)
score(h4 ,g2)
score(h5, g2)
h1
h2
h3
h4
h5
g1
g2
x1
x2
x3
x4
x5
Attention Layer
Context vector
Global align weights
c_t
a_t
$\bar{h}_s$
h_t

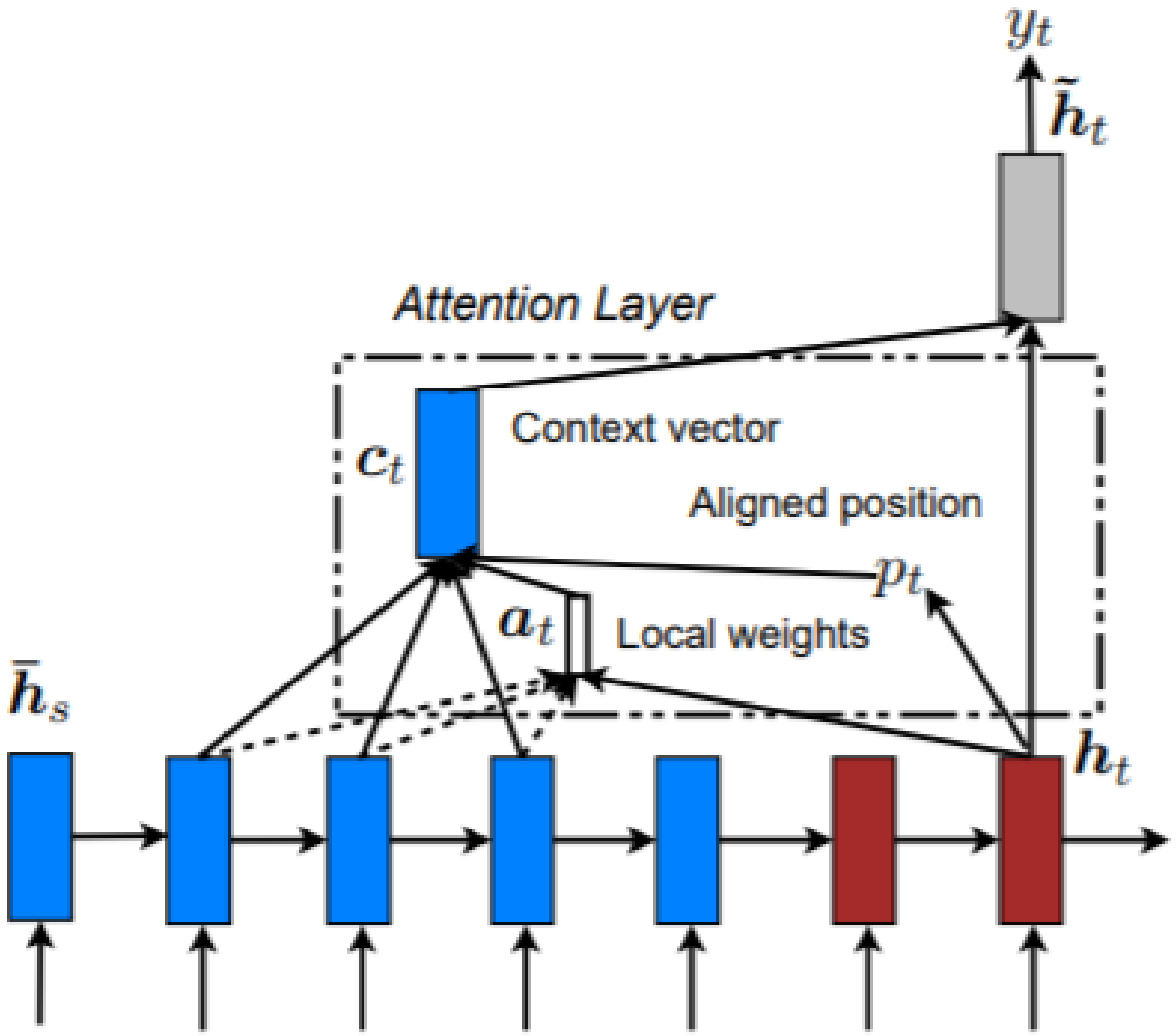
$$a_2(1)\,h1 + a_2(2)\,h2 + a_2(3)\,h3 + a_2(4)\,h4 + a_2(5)\,h5 = C_2 \text{ or } C_{g2}$$



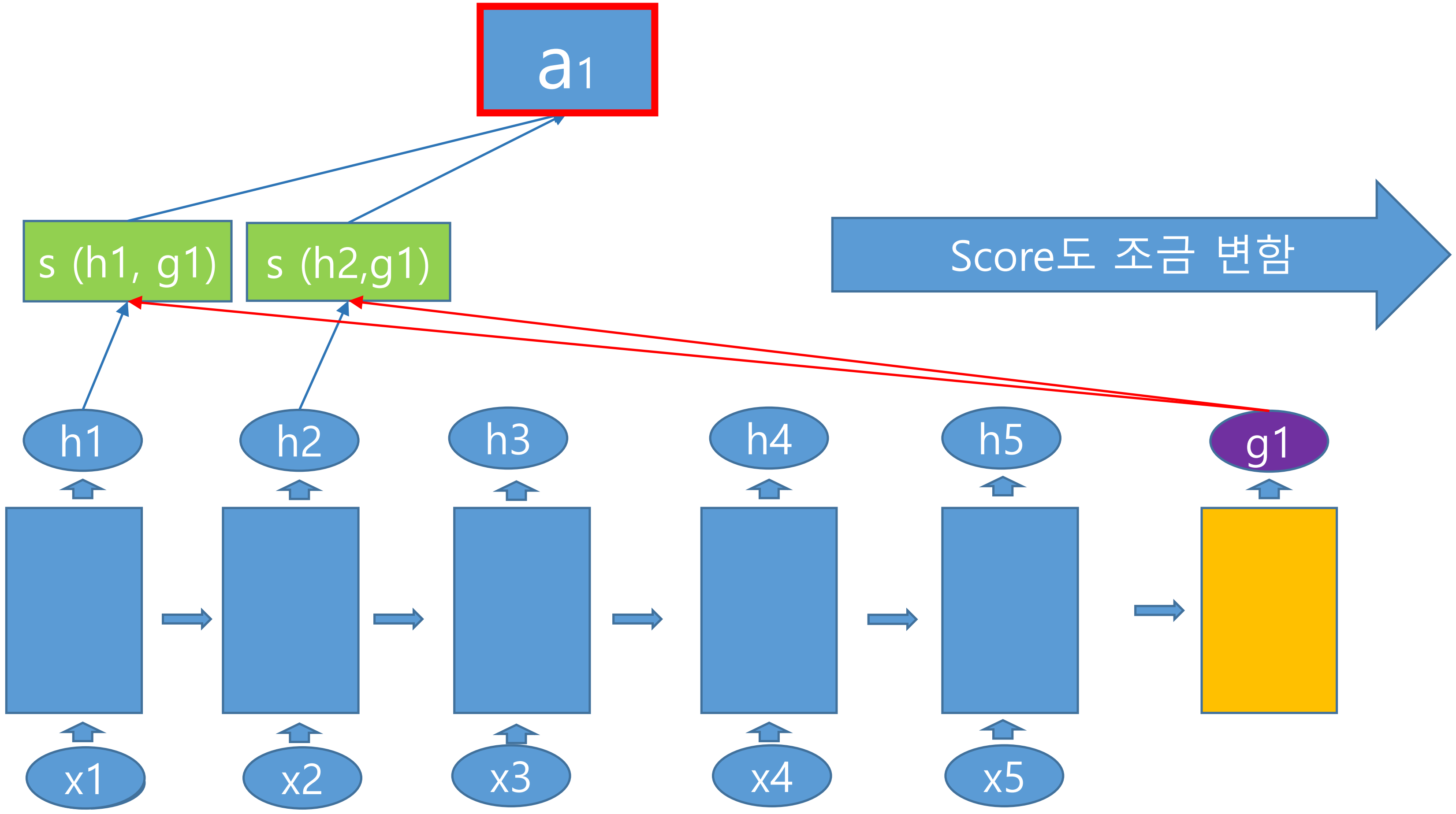



g2
C2
Output
(tanh)
Output
softmax
argmax
back propagation
h1
h2
h3
h4
h5
g1
g2
x1
x2
x3
x4
x5
y_t
$\tilde{h}_t$
Attention Layer
c_t
Context vector
Global align weights
a_t
$\bar{h}_s$
h_t

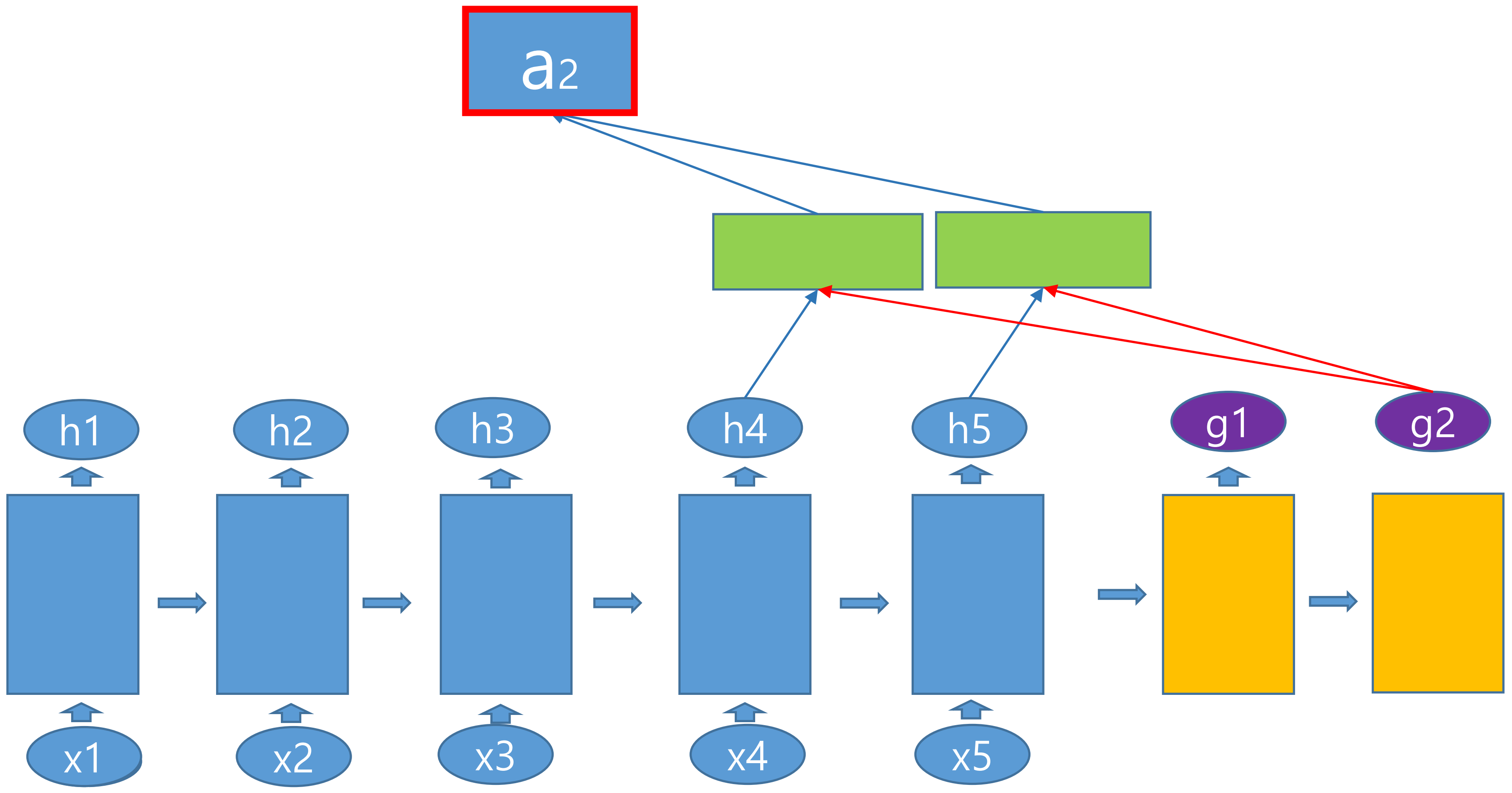
3.2 Local attention



3.2 Local attention



3.2 Local attention



3.2 Local attention

Monotonic alignment (**local-m**) – we simply set $p_t = t$ assuming that source and target sequences are roughly monotonically aligned. The alignment vector $\boldsymbol{a}_t$ is defined according to Eq. (7).[9]

pt: 어디를 중심으로 뽑을까???

local-m : 그냥 시간 순서대로 다음 단어가 중심

3.2 Local attention

Predictive alignment (**local-p**) – instead of assuming monotonic alignments, our model predicts an aligned position as follows:

$$\boxed{p_t = S \cdot \text{sigmoid}(\boldsymbol{v}_p^\top \tanh(\boldsymbol{W}_p \boldsymbol{h}_t)),} \quad (9)$$

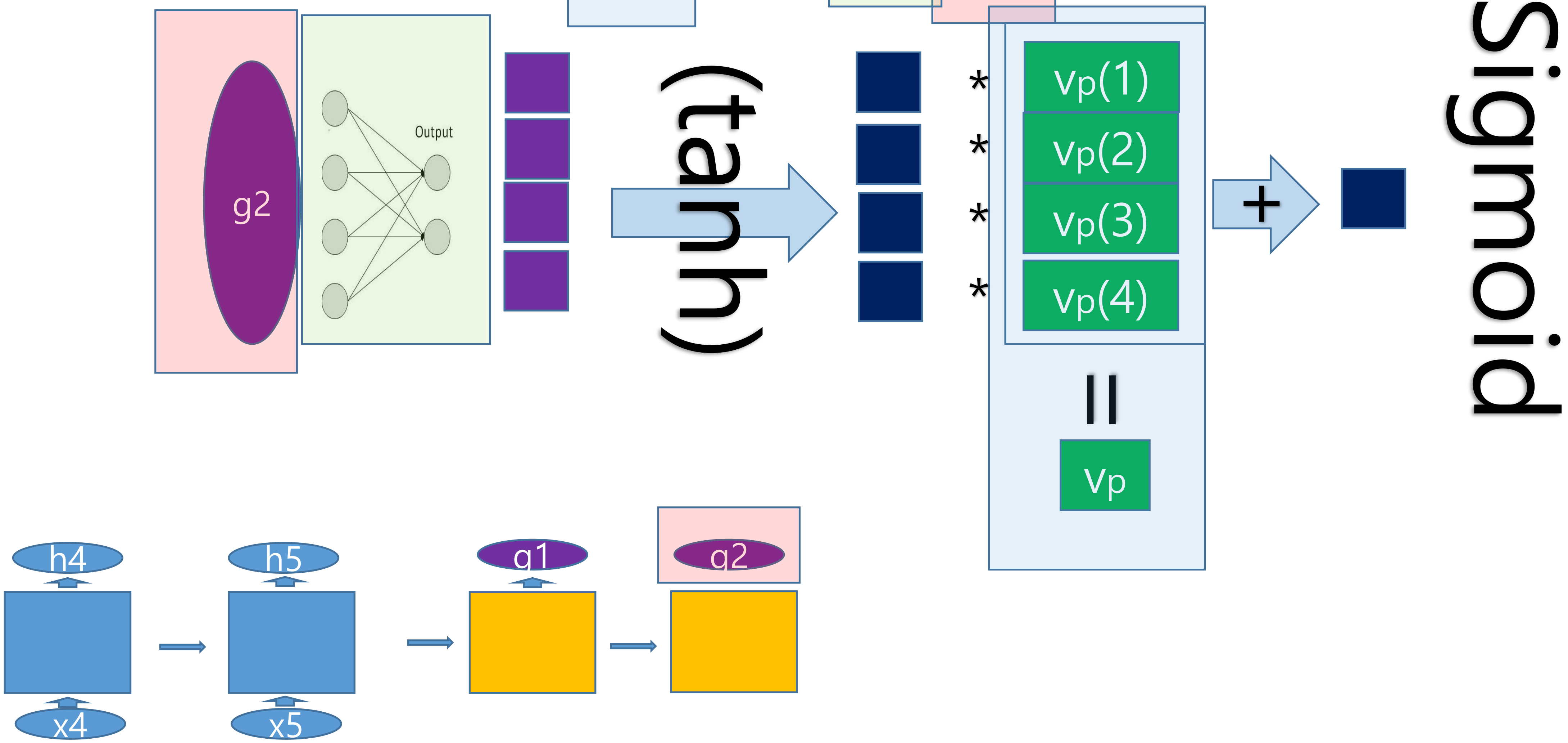
pt: 어디를 중심으로 뽑을까???

local-p : 예측 해서 씀

S:번역 대상 단어 갯수

3.2 Local attention

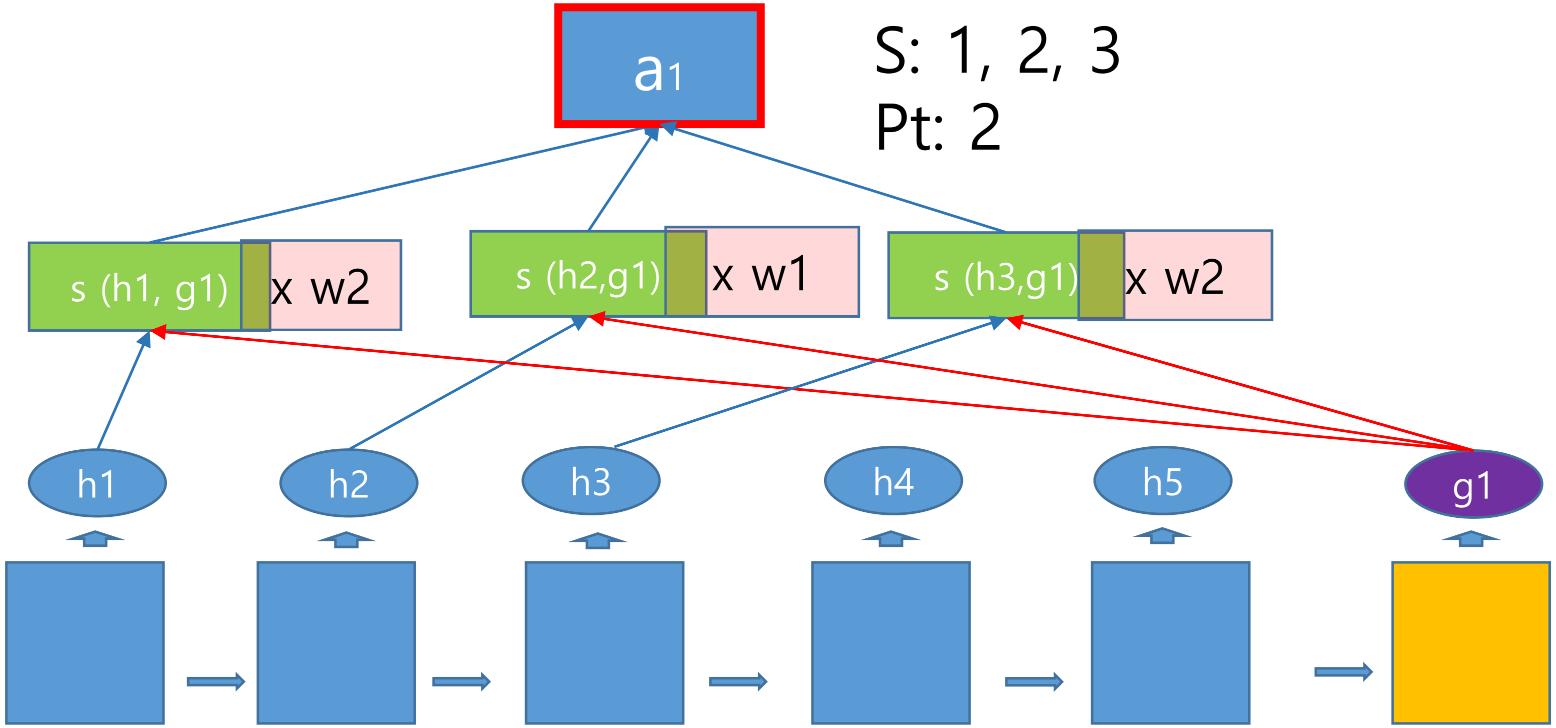
$$p_t = S \cdot \text{sigmoid}(\boldsymbol{v}_p^\top \tanh(\boldsymbol{W}_p \boldsymbol{h}_t)),$$

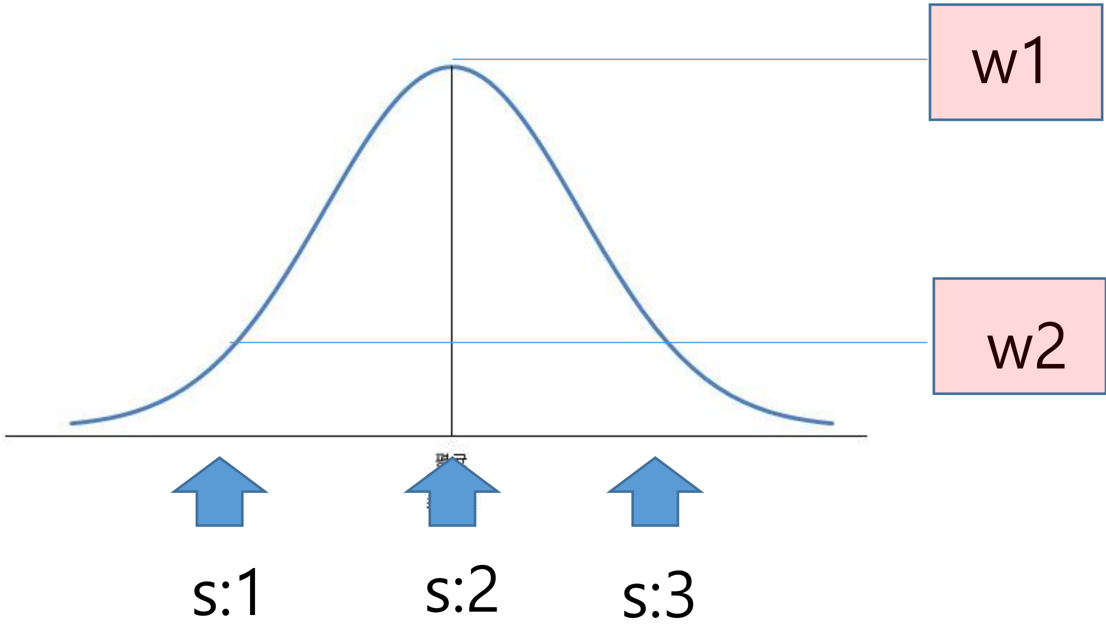


3.2 Local attention

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\boxed{\exp\left(-\frac{(s-p_t)^2}{2\sigma^2}\right)}$$

중심에 가중치주려는 의도임





3.3 Input-feeding Approach

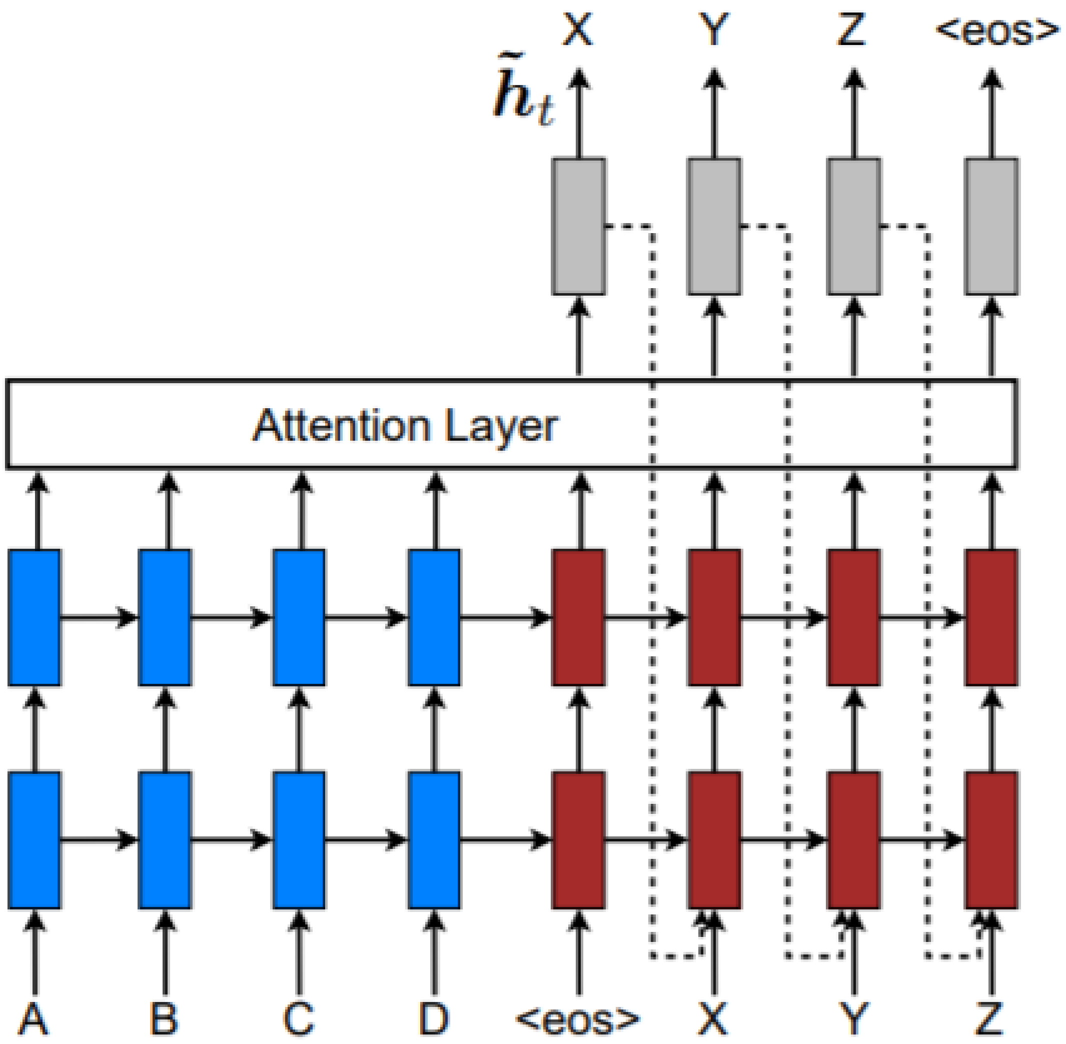


Figure 4: **Input-feeding approach** – Attentional vectors $\tilde{\boldsymbol{h}}_t$ are fed as inputs to the next time steps to inform the model about past alignment decisions.

System	Ppl	BLEU
Winning WMT'14 system – *phrase-based + large LM* (Buck et al., 2014)		20.7
Existing NMT systems		
RNNsearch (Jean et al., 2015)		16.5
RNNsearch + unk replace (Jean et al., 2015)		19.0
RNNsearch + unk replace + large vocab + *ensemble* 8 models (Jean et al., 2015)		**21.6**
Our NMT systems		
Base	10.6	11.3
Base + reverse	9.9	12.6 (+*1.3*)
Base + reverse + dropout	8.1	14.0 (+*1.4*)
Base + reverse + dropout + global attention (*location*)	7.3	16.8 (+*2.8*)
Base + reverse + dropout + global attention (*location*) + feed input	6.4	18.1 (+*1.3*)
Base + reverse + dropout + local-p attention (*general*) + feed input	5.9	19.0 (+*0.9*)
Base + reverse + dropout + local-p attention (*general*) + feed input + unk replace		20.9 (+*1.9*)
Ensemble 8 models + unk replace		**23.0** (+*2.1*)

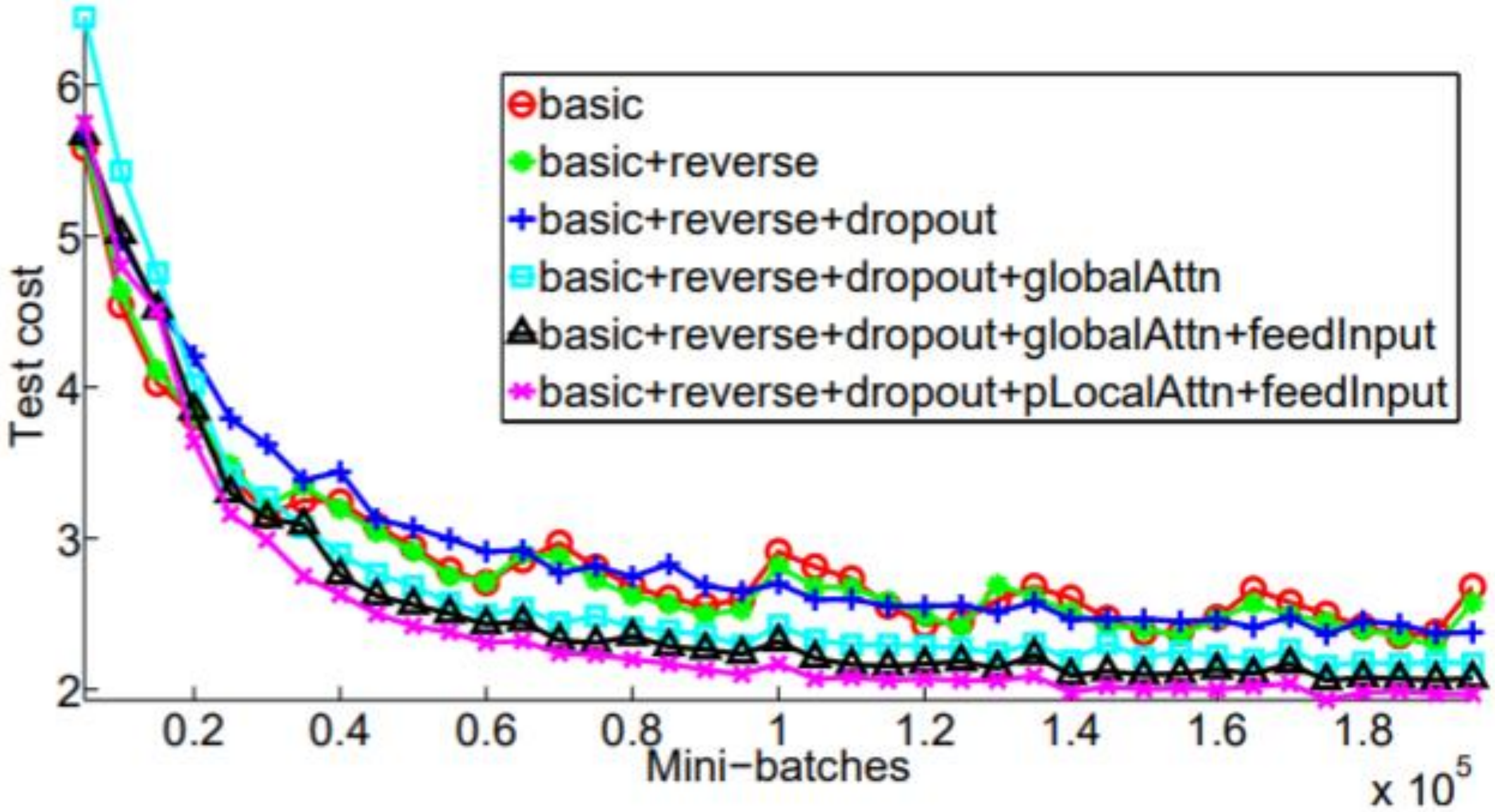


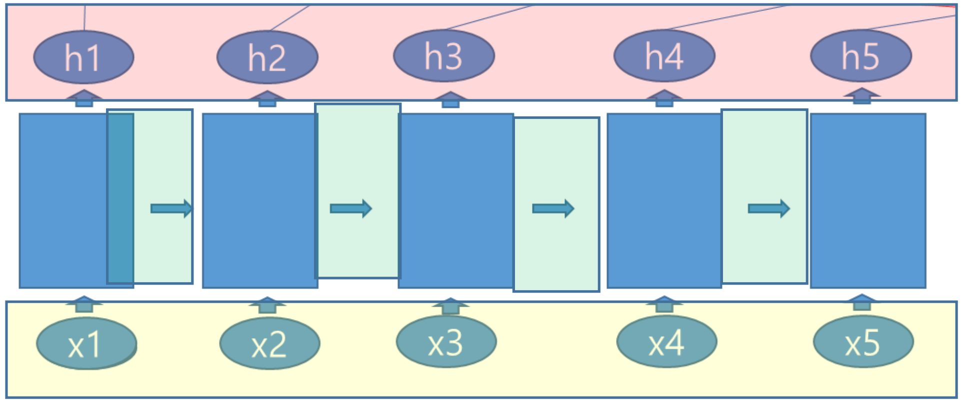
Figure 5: **Learning curves** – test cost (ln perplexity) on newstest2014 for English-German NMTs as training progresses.

Source code Introduce(pytorch, only global)

```
# Initialize models
encoder = EncoderRNN(input_lang.n_words, hidden_size, n_layers)
decoder = AttentionDecoderRNN(attn_model, hidden_size, output_lang.n_words, n_layers, dropout_p=dropout_p)

# Run the train step
loss = train(input_variable, target_variable, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion)
```

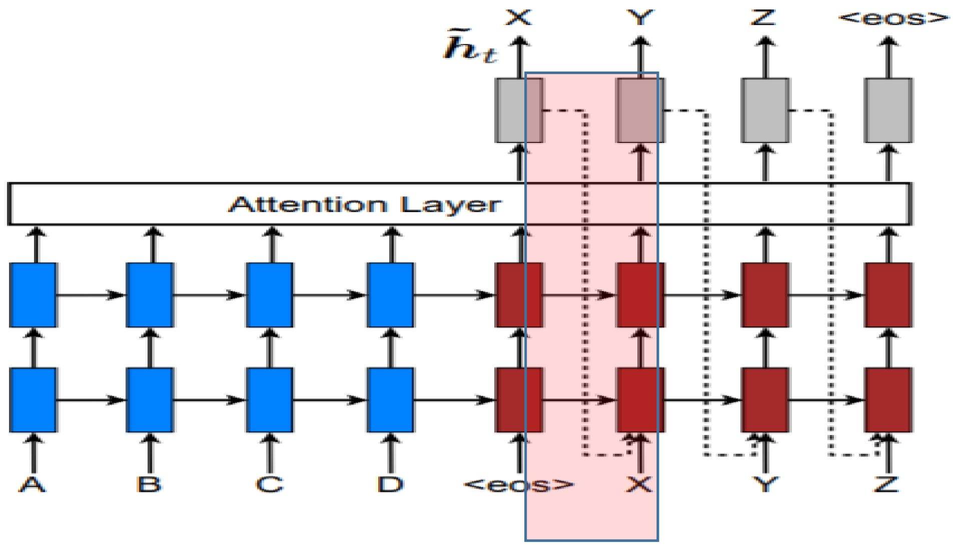
Source code Introduce frame

```
def train(input_var, target_var, encoder, decoder, encoder_opt, decoder_opt, criterion):
    do soemthing
    encoder_outputs, encoder_hidden = encoder(input_var, encoder_hidden)
    do soemthing

    for di in range(target_length):
        decoder_output, decoder_context, decoder_hidden, decoder_attention = decoder(decoder_input,
                                                                                     decoder_context,
                                                                                     decoder_hidden,
                                                                                     encoder_outputs)

        loss += criterion(decoder_output[0], target_var[di])

        topv, topi = decoder_output.data.topk(1)
        ni = topi[0][0]

        decoder_input = Variable(torch.LongTensor([[ni]]))
        decoder_input = decoder_input.cuda()
```

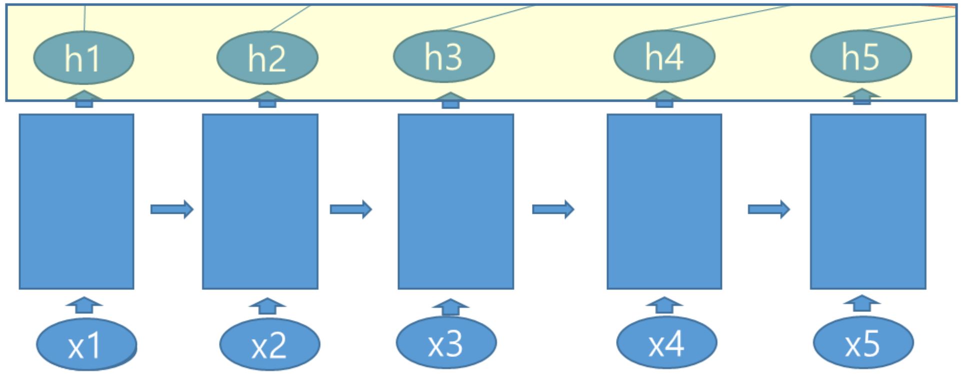


Source code Introduce frame

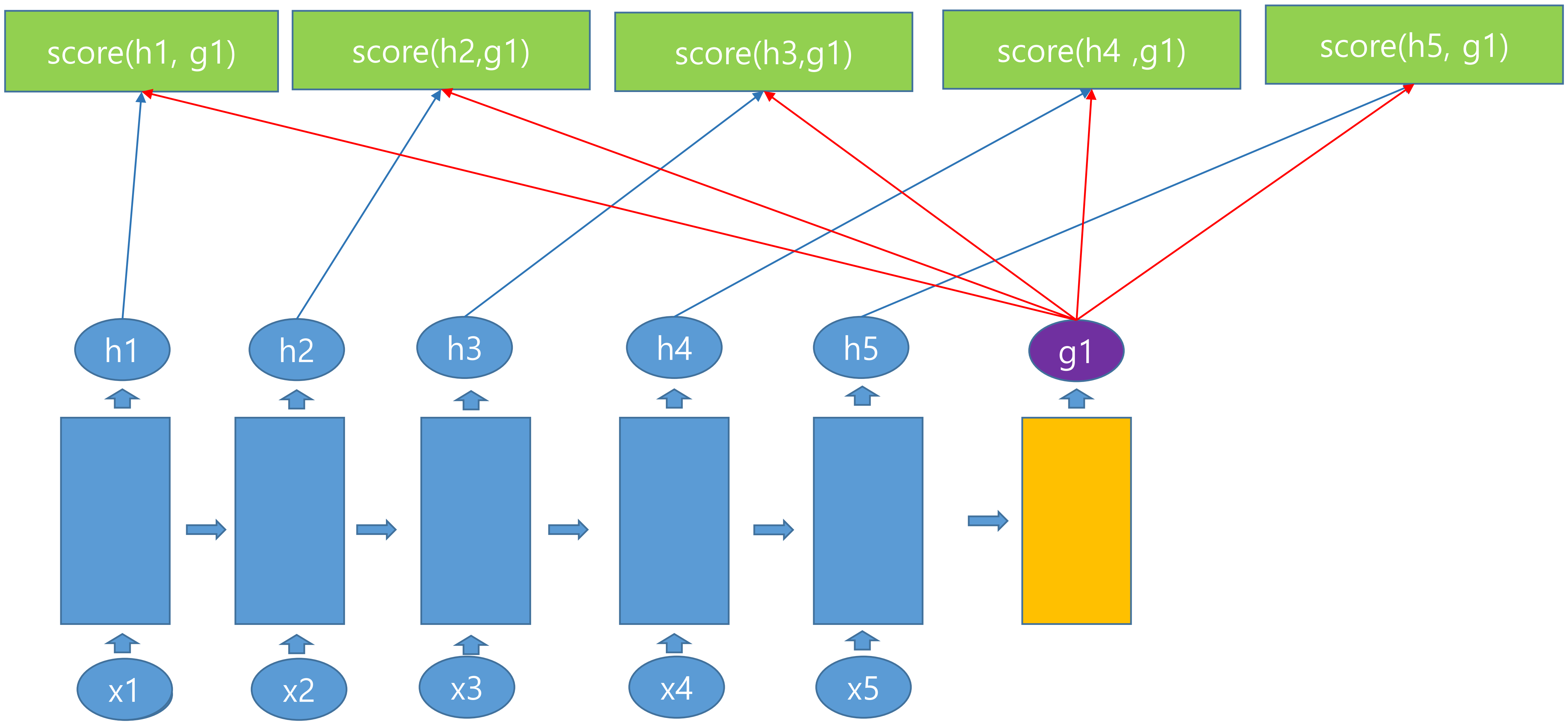
```
def train(input_var, target_var, encoder, decoder, encoder_opt, decoder_opt, criterion):
        do soemthing

        encoder_outputs, encoder_hidden = encoder(input_var, encoder_hidden)

        do soemthing

         for di in range(target_length):
             decoder_output, decoder_context, decoder_hidden, decoder_attention = decoder(decoder_input,
                                                                                           decoder_context,
                                                                                           decoder_hidden,
                                                                                           encoder_outputs)
                                                                    )
```



Source code Introduce Encoder

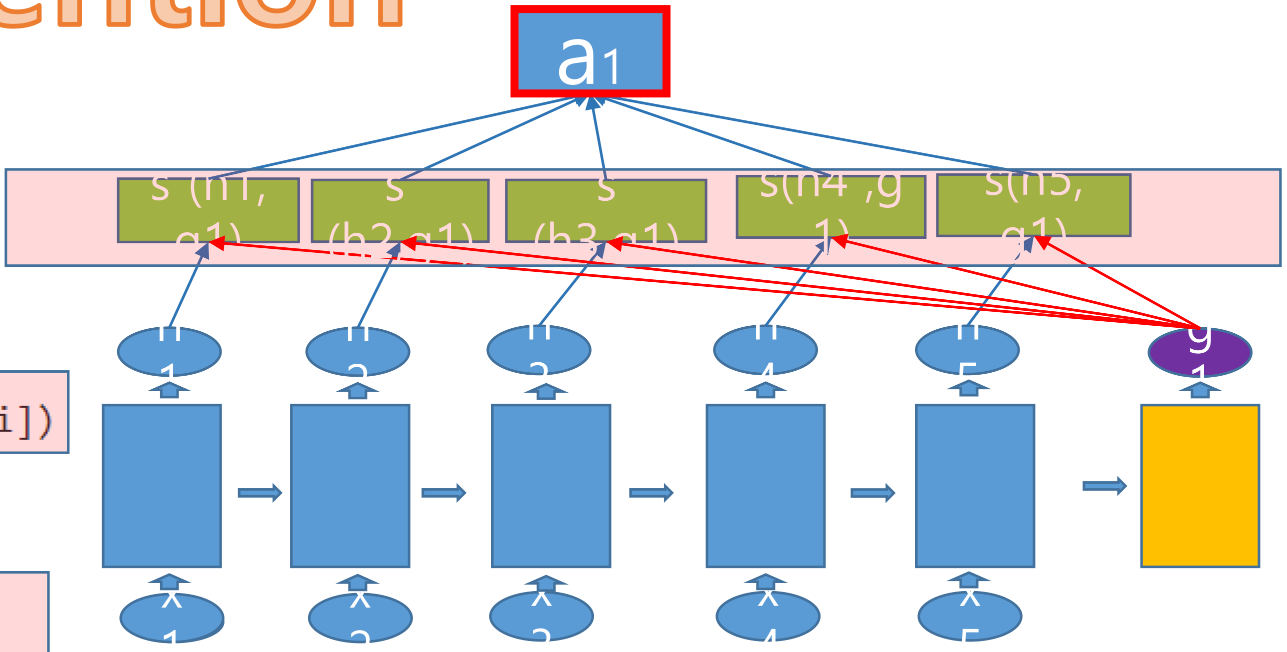



```python
class EncoderRNN(nn.Module):
    """Recurrent neural network that encodes a given input sequence."""

    def __init__(self, input_size, hidden_size, n_layers=1):
        super(EncoderRNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)

    def forward(self, word_inputs, hidden):
        seq_len = len(word_inputs)
        embedded = self.embedding(word_inputs).view(seq_len, 1, -1)
        output, hidden = self.gru(embedded, hidden)
        return output, hidden

    def init_hidden(self):
        hidden = Variable(torch.zeros(self.n_layers, 1, self.hidden_size))
        hidden = hidden.cuda()
        return hidden
```


score(h1, g1)
score(h2,g1)
score(h3,g1)
score(h4 ,g1)
score(h5, g1)
h1
h2
h3
h4
h5
g1
x1
x2
x3
x4
x5

Source code Introduce attention

```python
def forward(self, hidden, encoder_outputs):

    seq_len = len(encoder_outputs)
    energies = Variable(torch.zeros(seq_len)).cuda()
    for i in range(seq_len):
        energies[i] = self._score(hidden, encoder_outputs[i])
    return F.softmax(energies).unsqueeze(0).unsqueeze(0)
```

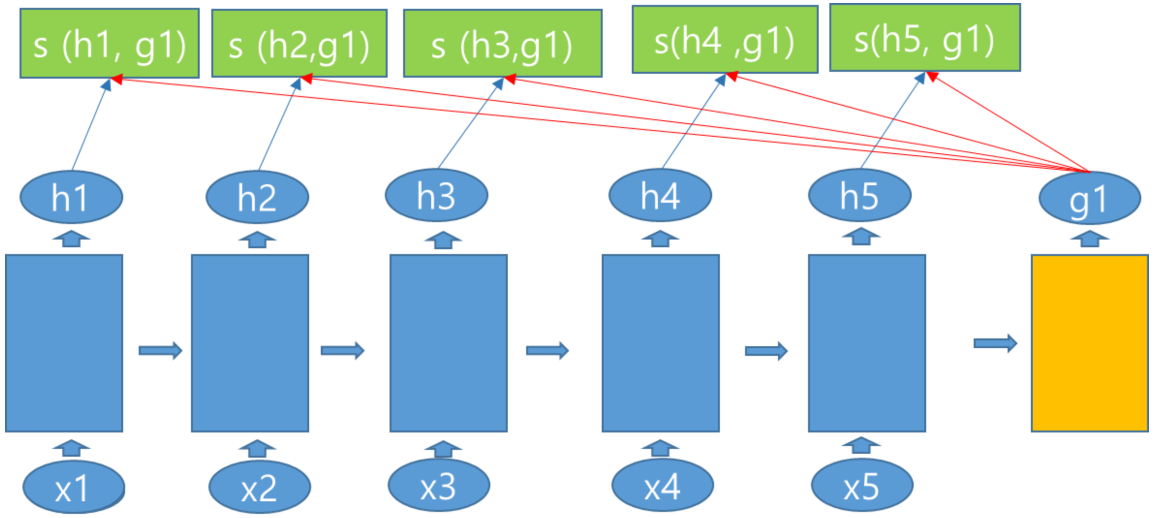



```python
def _score(self, hidden, encoder_output):
    """Calculate the relevance of a particular encoder output in respect to


    if self.method == 'dot':
        energy = hidden.dot(encoder_output)
    elif self.method == 'general':
        energy = self.attention(encoder_output)
        energy = hidden.dot(energy)
    elif self.method == 'concat':
        energy = self.attention(torch.cat((hidden, encoder_output), 1))
        energy = self.other.dor(energy)
    return energy
```

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

Source code Introduce Attention

$$a_1(1) = \frac{\exp\ s(h1,g1)}{\exp\ s(h1,g1) + \exp\ s(h2,g1) + \exp\ s(h3,g1) + \exp\ s(h4,g1) + \exp\ s(h5,g1)}$$



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

Source code Introduce Attention

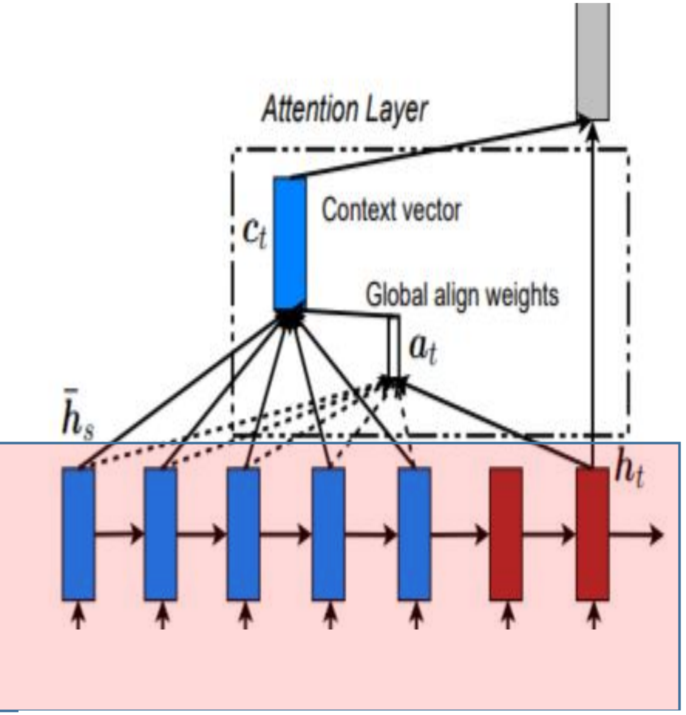
$$a1(1) = \frac{\exp\, s(h1, g1)}{\exp\, s(h1, g1) + \exp\, s(h2, g1) + \exp\, s(h3, g1) + \exp\, s(h4, g1) + \exp\, s(h5, g1)}$$

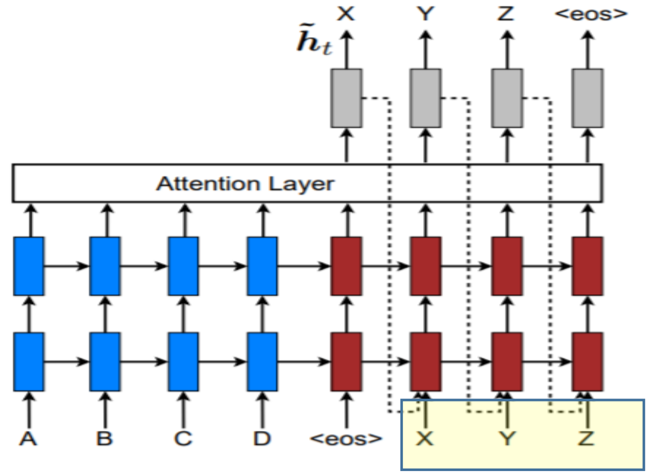
```python
def forward(self, hidden, encoder_outputs):

    seq_len = len(encoder_outputs)
    energies = Variable(torch.zeros(seq_len)).cuda()
    for i in range(seq_len):
        energies[i] = self._score(hidden, encoder_outputs[i])
    return F.softmax(energies).unsqueeze(0).unsqueeze(0)


def _score(self, hidden, encoder_output):
    """Calculate the relevance of a particular encoder output in respect to

    if self.method == 'dot':
        energy = hidden.dot(encoder_output)
    elif self.method == 'general':
        energy = self.attention(encoder_output)
        energy = hidden.dot(energy)
    elif self.method == 'concat':
        energy = self.attention(torch.cat((hidden, encoder_output), 1))
        energy = self.other.dor(energy)
    return energy
```

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

Source code Introduce

Decoder

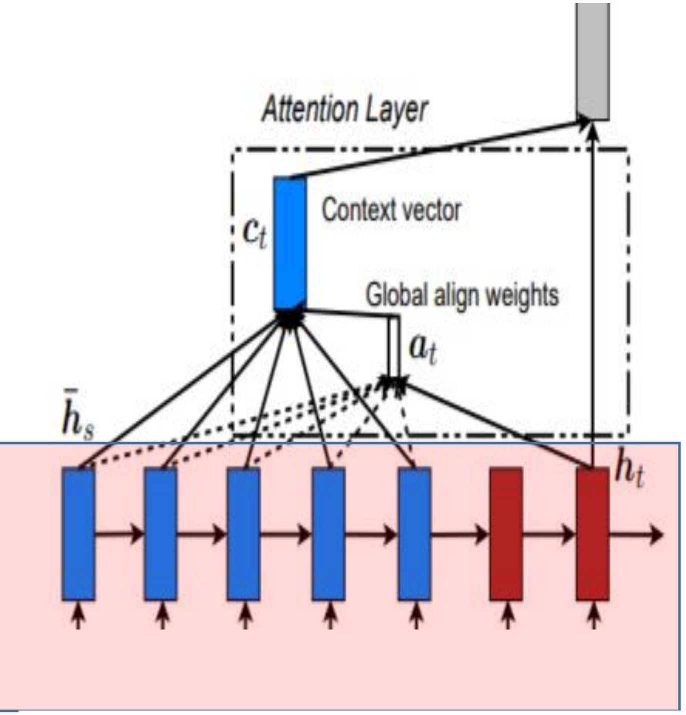


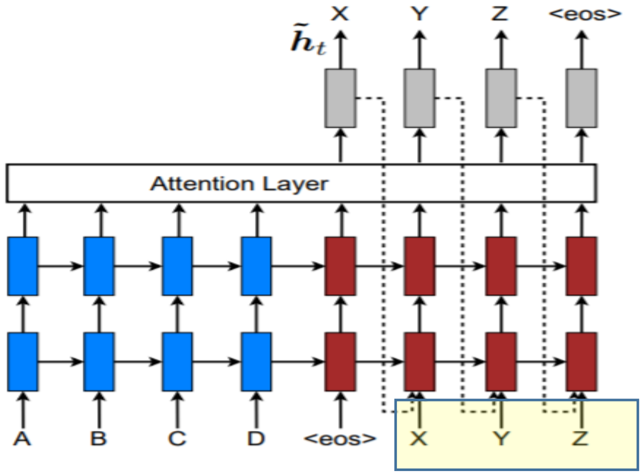



```
# Run through RNN
word_embedded = self.embedding(word_input).view(1, 1, -1)
rnn_input = torch.cat((word_embedded, last_context.unsqueeze(0)), 2)
rnn_output, hidden = self.gru(rnn_input, last_hidden)

# Calculate attention
attention_weights = self.attention(rnn_output.squeeze(0), encoder_outputs)
context = attention_weights.bmm(encoder_outputs.transpose(0, 1))

# Predict output
rnn_output = rnn_output.squeeze(0)
context = context.squeeze(1)
output = F.log_softmax(self.out(torch.cat((rnn_output, context), 1)))
return output, context, hidden, attention_weights
```

Source code Introduce

Decoder

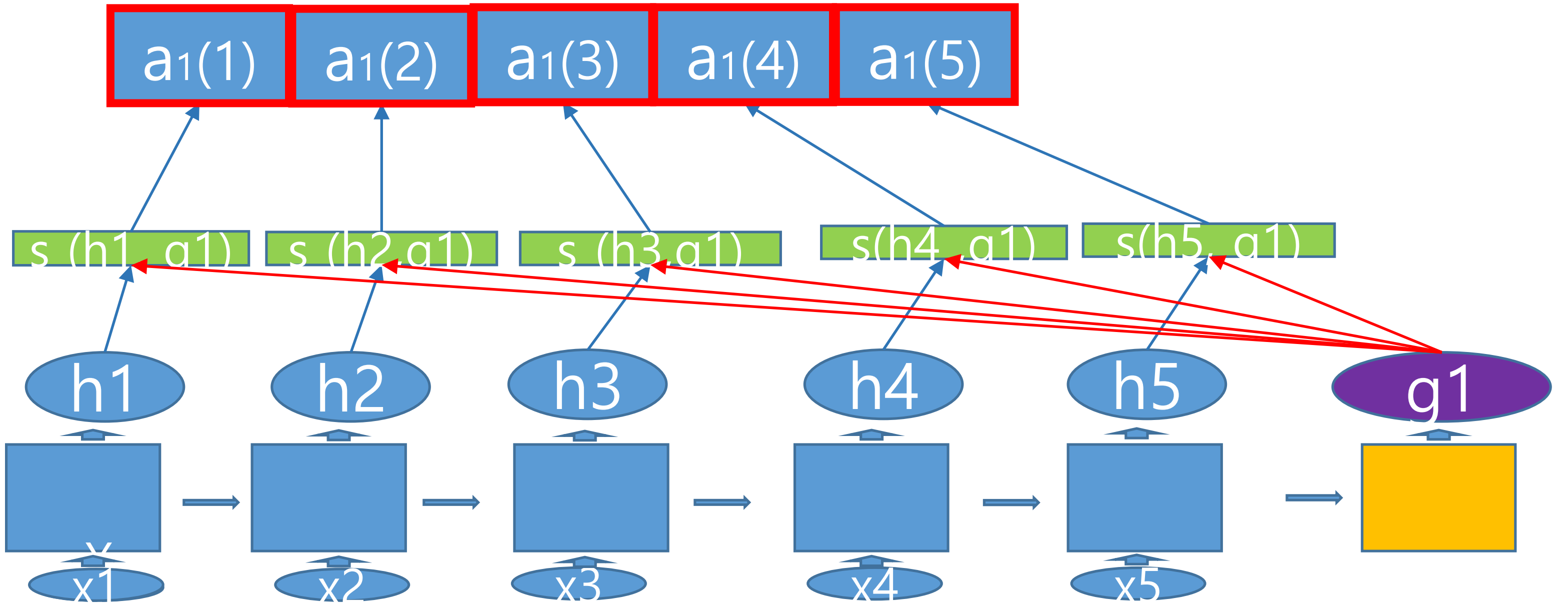





```
# Run through RNN
word_embedded = self.embedding(word_input).view(1, 1, -1)
rnn_input = torch.cat((word_embedded, last_context.unsqueeze(0)), 2)
rnn_output, hidden = self.gru(rnn_input, last_hidden)

# Calculate attention
attention_weights = self.attention(rnn_output.squeeze(0), encoder_outputs)
context = attention_weights.bmm(encoder_outputs.transpose(0, 1))

# Predict output
rnn_output = rnn_output.squeeze(0)
context = context.squeeze(1)
output = F.log_softmax(self.out(torch.cat((rnn_output, context), 1)))
return output, context, hidden, attention_weights
```

3.1 global Attention
context vector

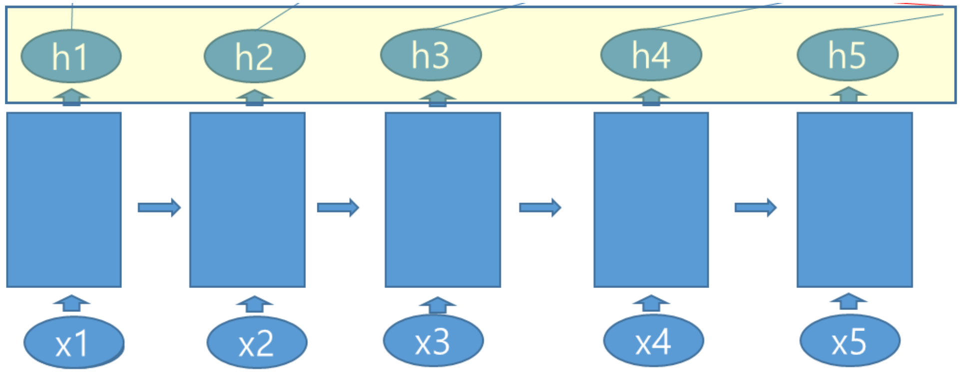
$$a_1(1)\,h1 + a_1(2)\,h2 + a_1(3)\,h3 + a_1(4)\,h4 + a_1(5)\,h5 = C_1 \text{ or } C_{g1}$$

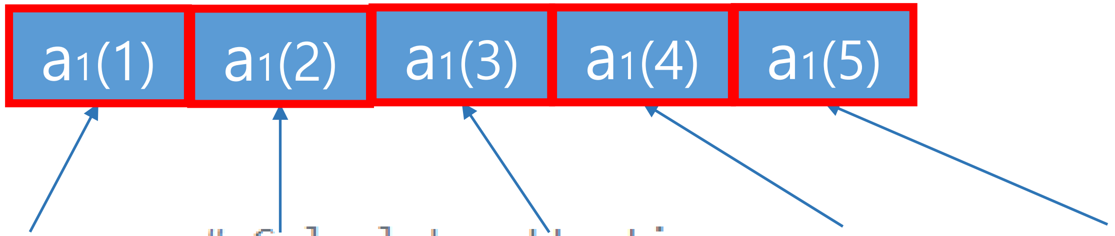


가중치가 a인 h들의 가중평균
= context vector

Source code Introduce

Decoder

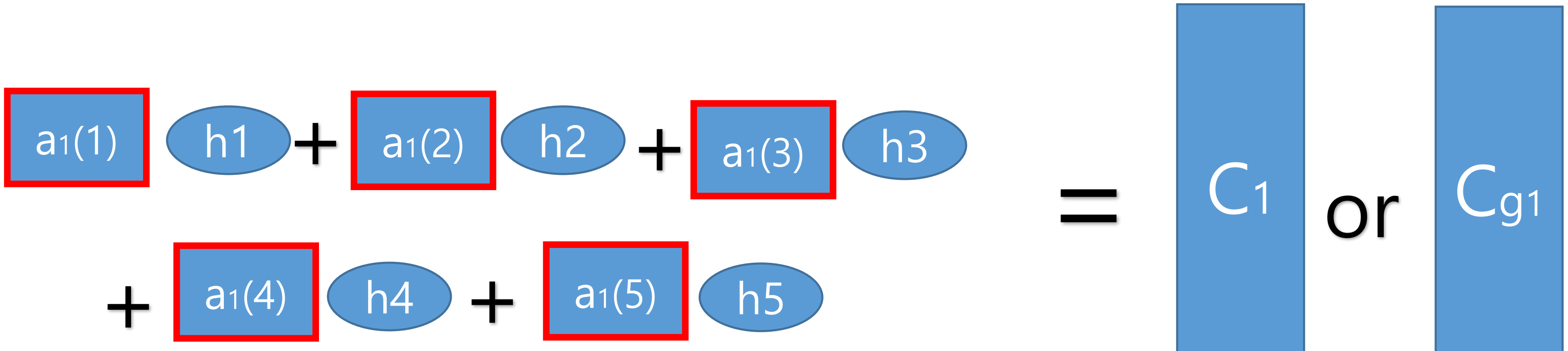





```
# Calculate attention
attention_weights = self.attention(rnn_output.squeeze(0), encoder_outputs)
context = attention_weights.bmm(encoder_outputs.transpose(0, 1))

# Predict output
rnn_output = rnn_output.squeeze(0)
context = context.squeeze(1)
output = F.log_softmax(self.out(torch.cat((rnn_output, context), 1)))
return output, context, hidden, attention_weights
```

Source code Introduce

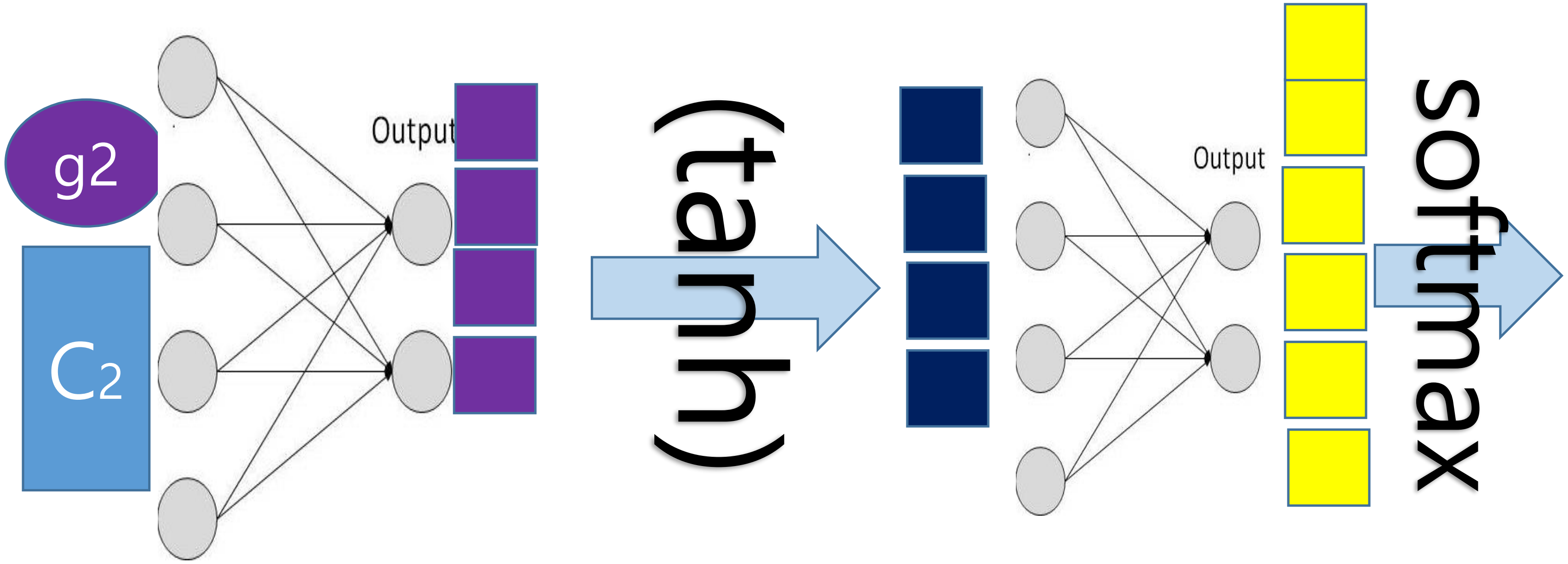
Decoder




```
# Calculate attention
attention_weights = self.attention(rnn_output.squeeze(0), encoder_outputs)
context = attention_weights.bmm(encoder_outputs.transpose(0, 1))

# Predict output
rnn_output = rnn_output.squeeze(0)
context = context.squeeze(1)
output = F.log_softmax(self.out(torch.cat((rnn_output, context), 1)))
return output, context, hidden, attention_weights
```

Source code Introduce

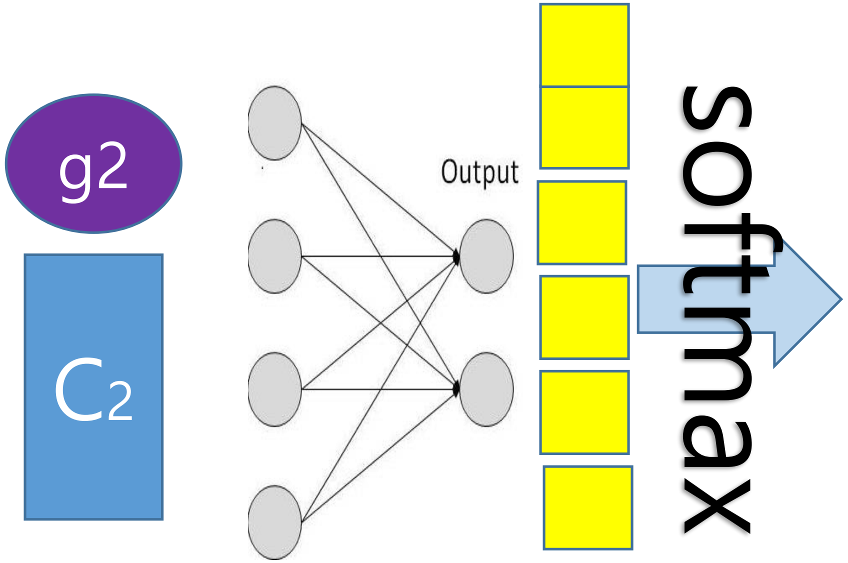
Decoder




```
# Predict output
rnn_output = rnn_output.squeeze(0)
context = context.squeeze(1)
output = F.log_softmax(self.out(torch.cat((rnn_output, context), 1)))
return output, context, hidden, attention_weights
```

Source code Introduce

Decoder




```
# Predict output
rnn_output = rnn_output.squeeze(0)
context = context.squeeze(1)
output = F.log_softmax(self.out(torch.cat((rnn_output, context), 1)))
return output, context, hidden, attention_weights
```