C. Attention and Transformers

C.1 Origins of attention

C.2 Attention as a standalone operator

C.3 Transformer architectures

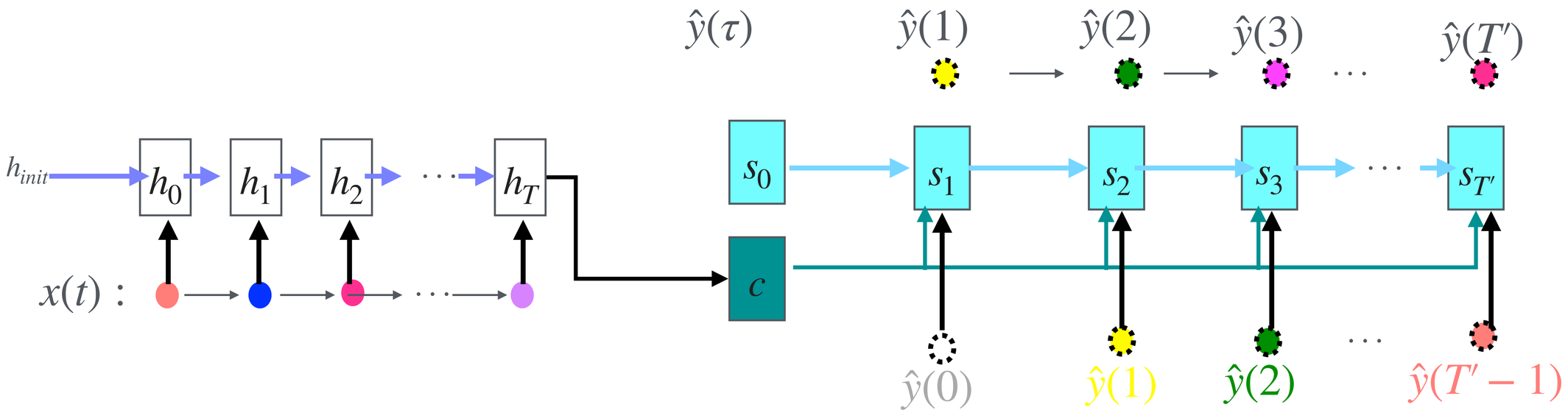
C.4 Transformers across different data structures

C.5 Training transformers

where the attention mechanism first appeared and why it was introduced

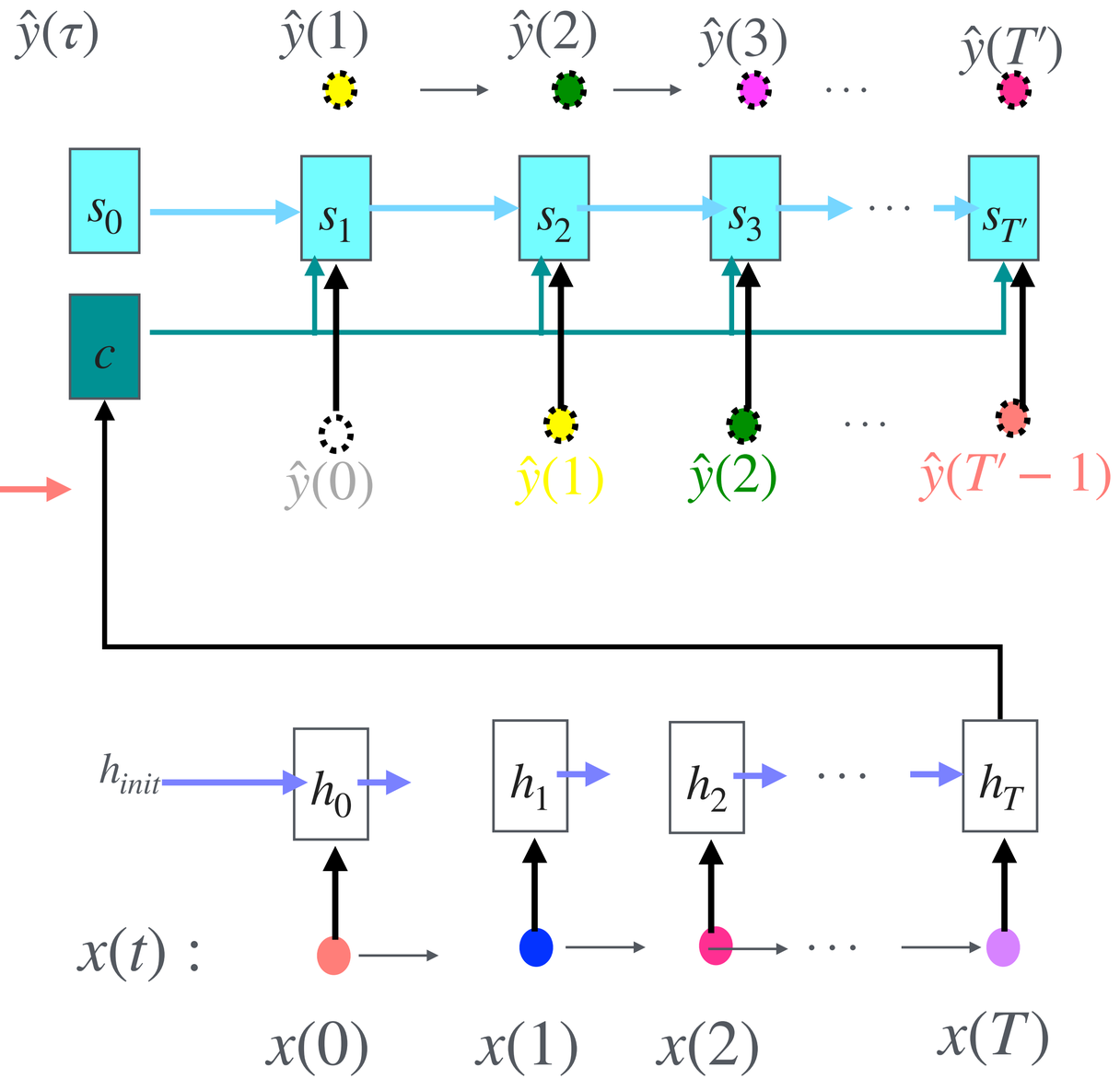
Tiam Heydari, PhD
Postdoctoral Fellow, UBC

- Remember from our RNN lecrue, specifically the **Sequence → sequence**. Out methods was:
- The encoder processes the full input sequence
- Its final state is used as a single context vector c
- This vector must summarize the entire input
- The decoder uses this same summary (c) to generate all outputs Important information can be lost in this compression



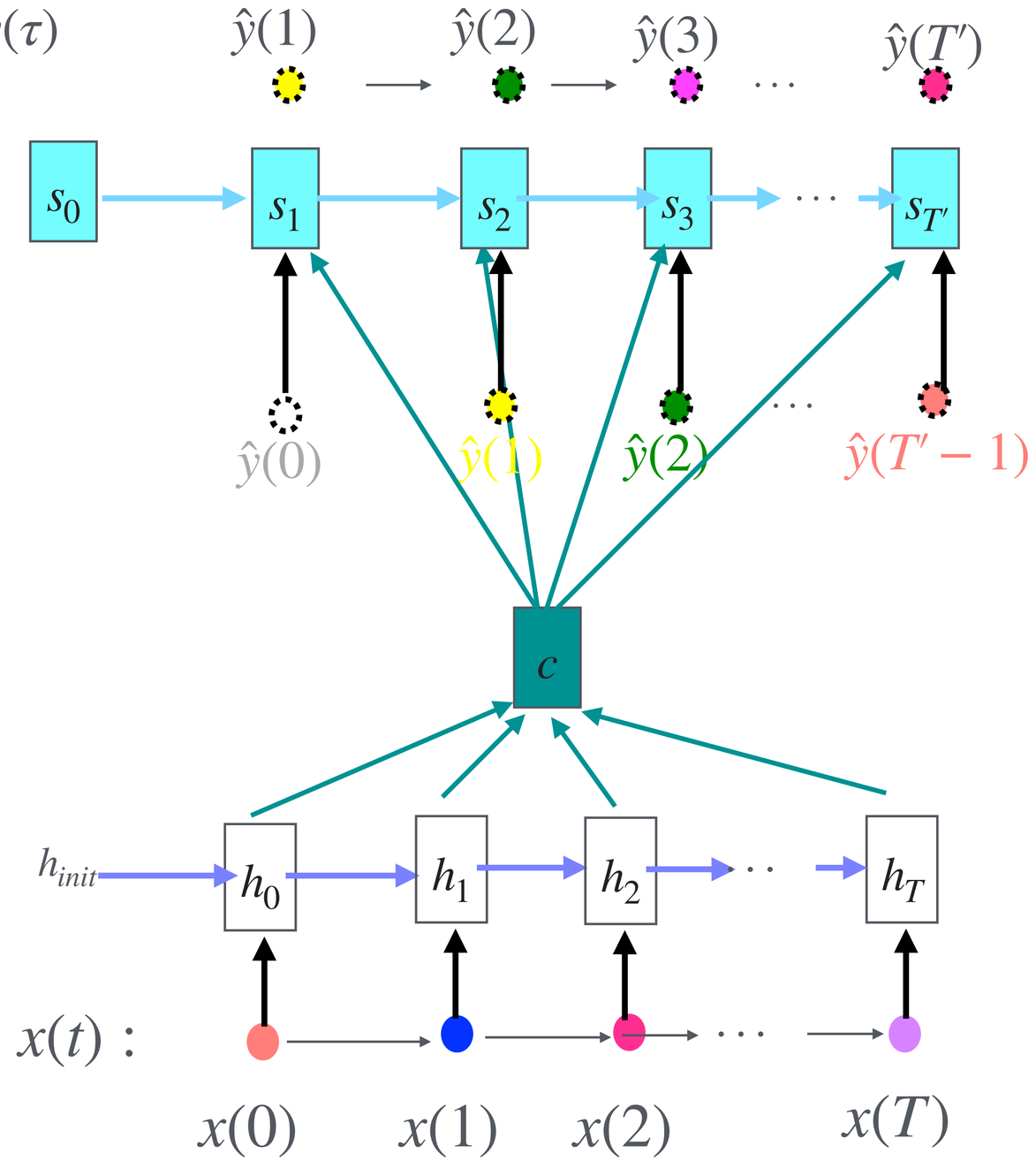
Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." *Advances in neural information processing systems* 27 (2014).

- This means a large amount of information must be squeezed into one vector, c
- This introduces a serious **bottleneck problem**
- For very long sequences, such as DNA or a book, etc, this becomes unrealistic



Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." *Advances in neural information processing systems* 27 (2014).

- A natural next idea is to let the context vector c use information from all encoder hidden states, not only the final one
- A schematic idea is shown on the right
- However, there is still a problem, We still have only one context vector c
- Even if c depends on all encoder states $(h_1, \ldots, h_T)$, it still must compress them into a single vector for all decoder steps!



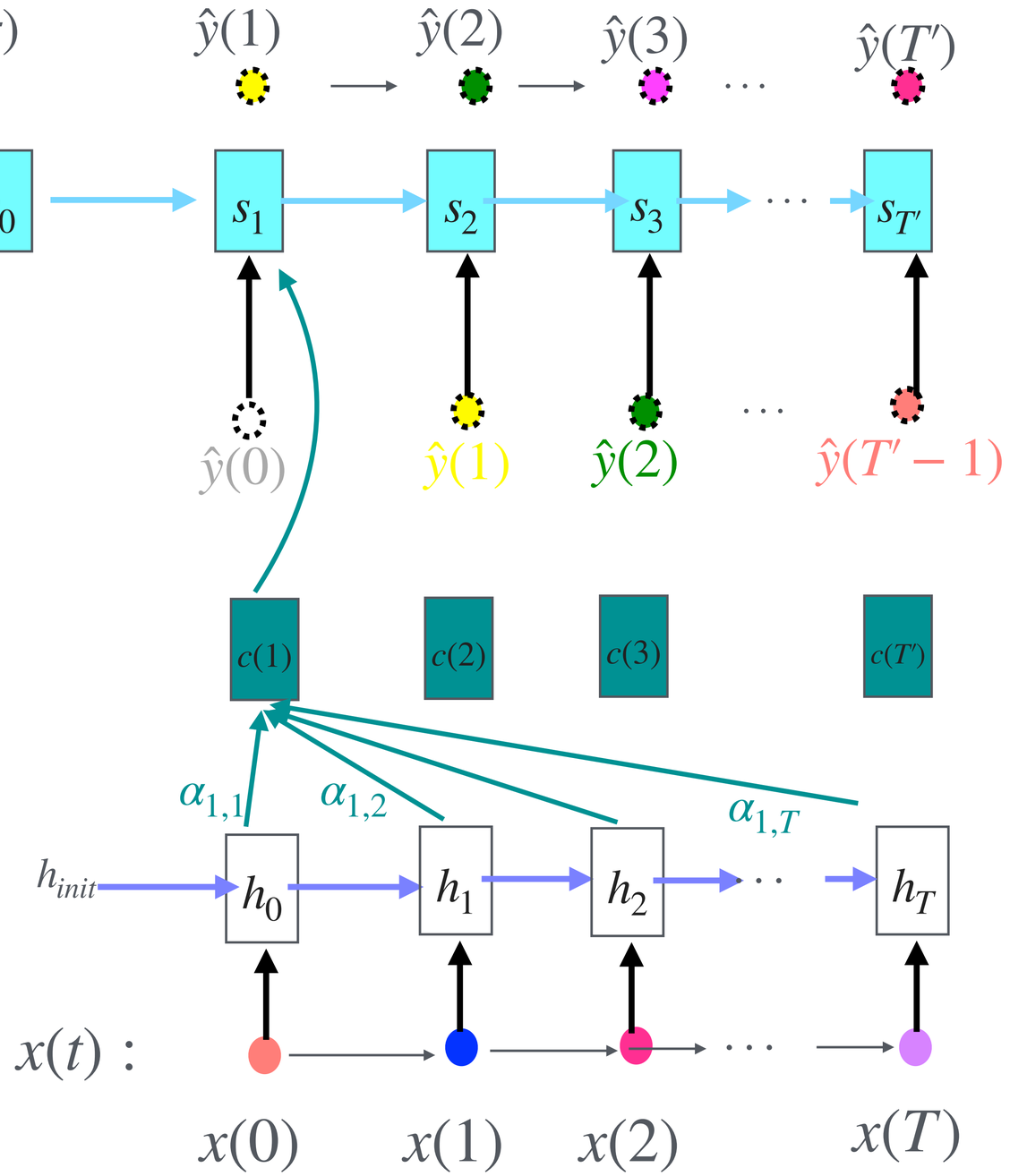
- Ok, so let's have multiple context vectors, one for each decoder step:

$$c(1), c(2), \ldots . c(T')$$

Each context vector, does something simple, it calcite a weighted sum of all hidden states of the encoder:

$$c(i) = \sum \alpha_{i,j} h_j$$

Where $\alpha_{i,j}$ for now can just be learnable weights(basically scalar values...)



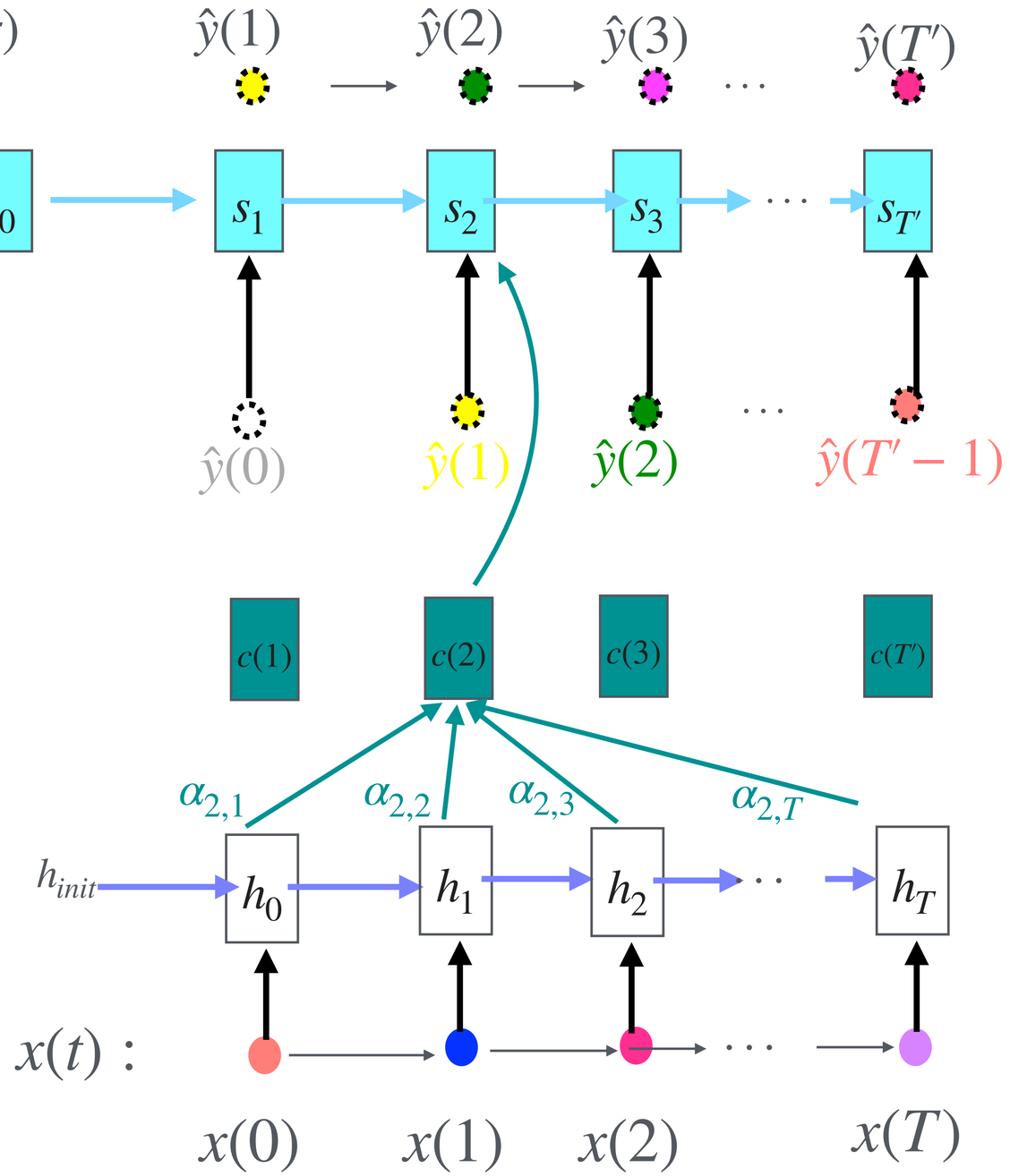
- Ok, so let's have multiple context vectors, one for each decoder step:

$$c(1), c(2), \ldots . c(T')$$

Each context vector, does something simple, it calcite a weighted sum of all hidden states of the encoder:

$$c(i) = \sum \alpha_{i,j} h_j$$

Where $\alpha_{i,j}$ for now can just be learnable weights(basically scalar values...)



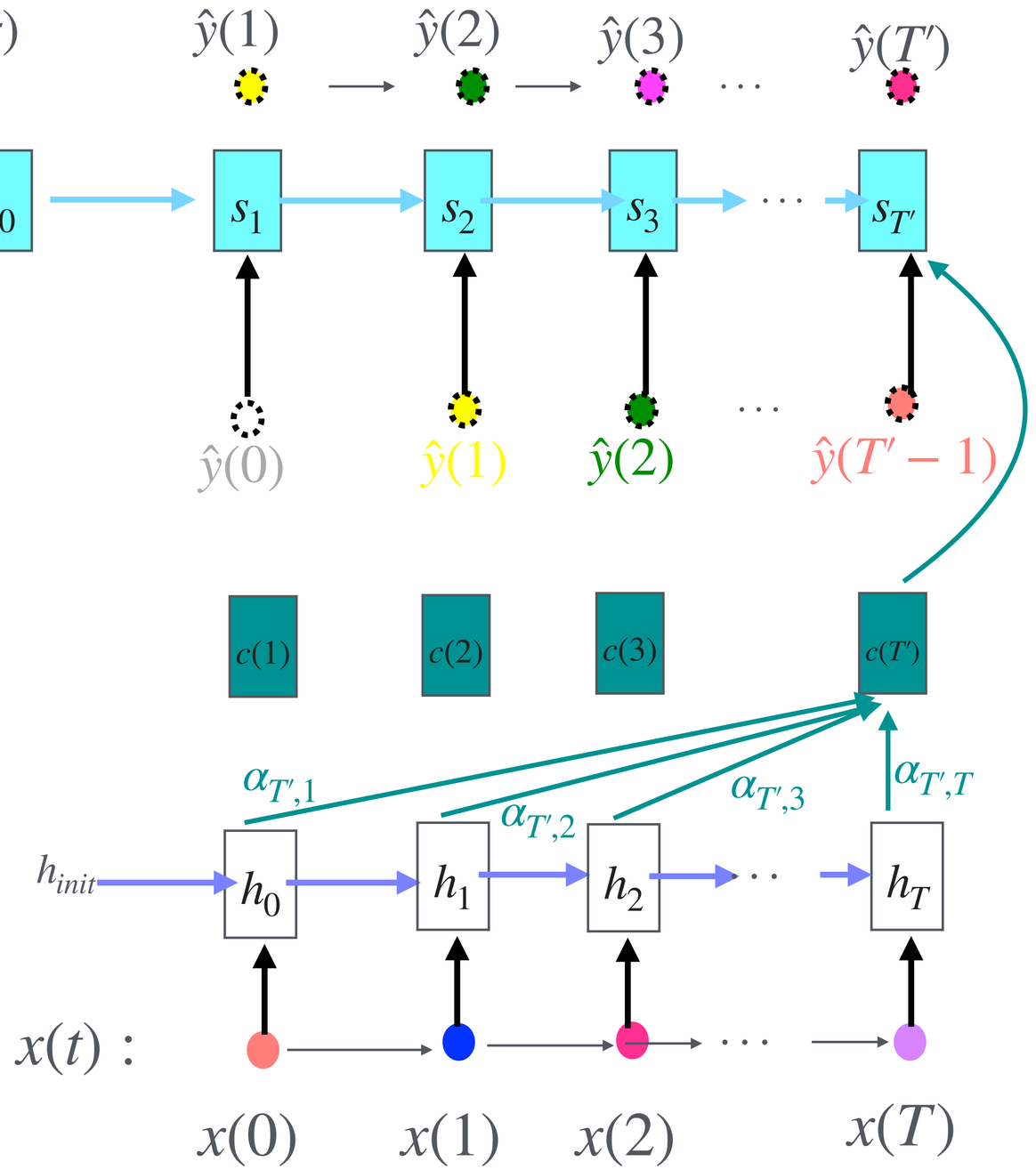
- Ok, so let's have multiple context vectors, one for each decoder step:

$$c(1), c(2), \ldots . c(T')$$

Each context vector, does something simple, it calcite a weighted sum of all hidden states of the encoder:

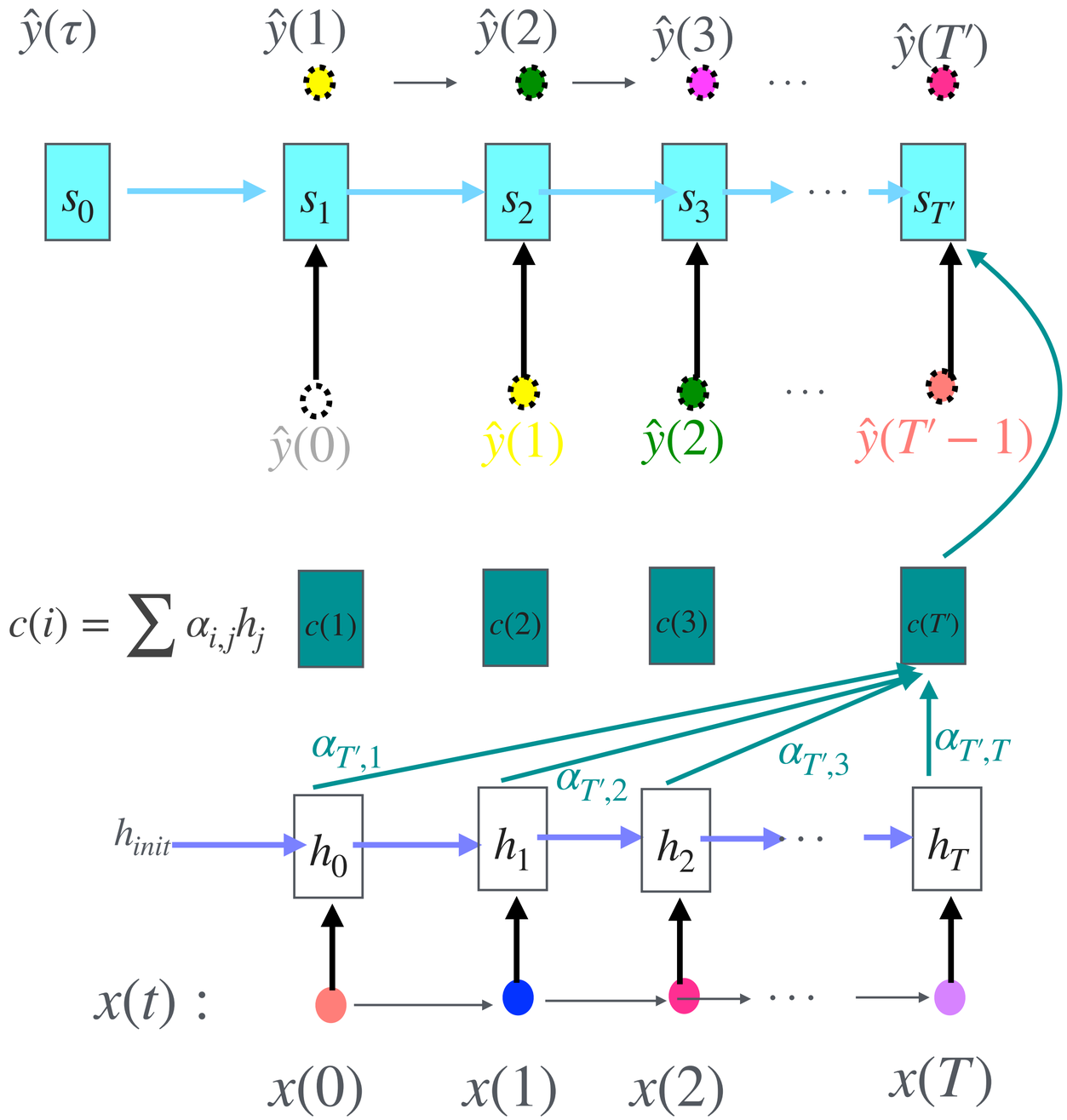
$$c(i) = \sum \alpha_{i,j} h_j$$

Where $\alpha_{i,j}$ for now can just be learnable weights(basically scalar values...)



- But there is still **a problem**
- If $\alpha_{i,j}$ are fixed learned scalars, they are the same for every input sequence!
- Then decoder step would always look at the same pattern of encoder states, regardless of what is going on in the data
- This is too rigid, because the relevant encoder states should depend on the **actual hidden stated** h_t and the current **decoder states** $s_{t'}$.

"So the weights $\alpha_{i,j}$ should **be computed dynamically**, not fixed in advance"

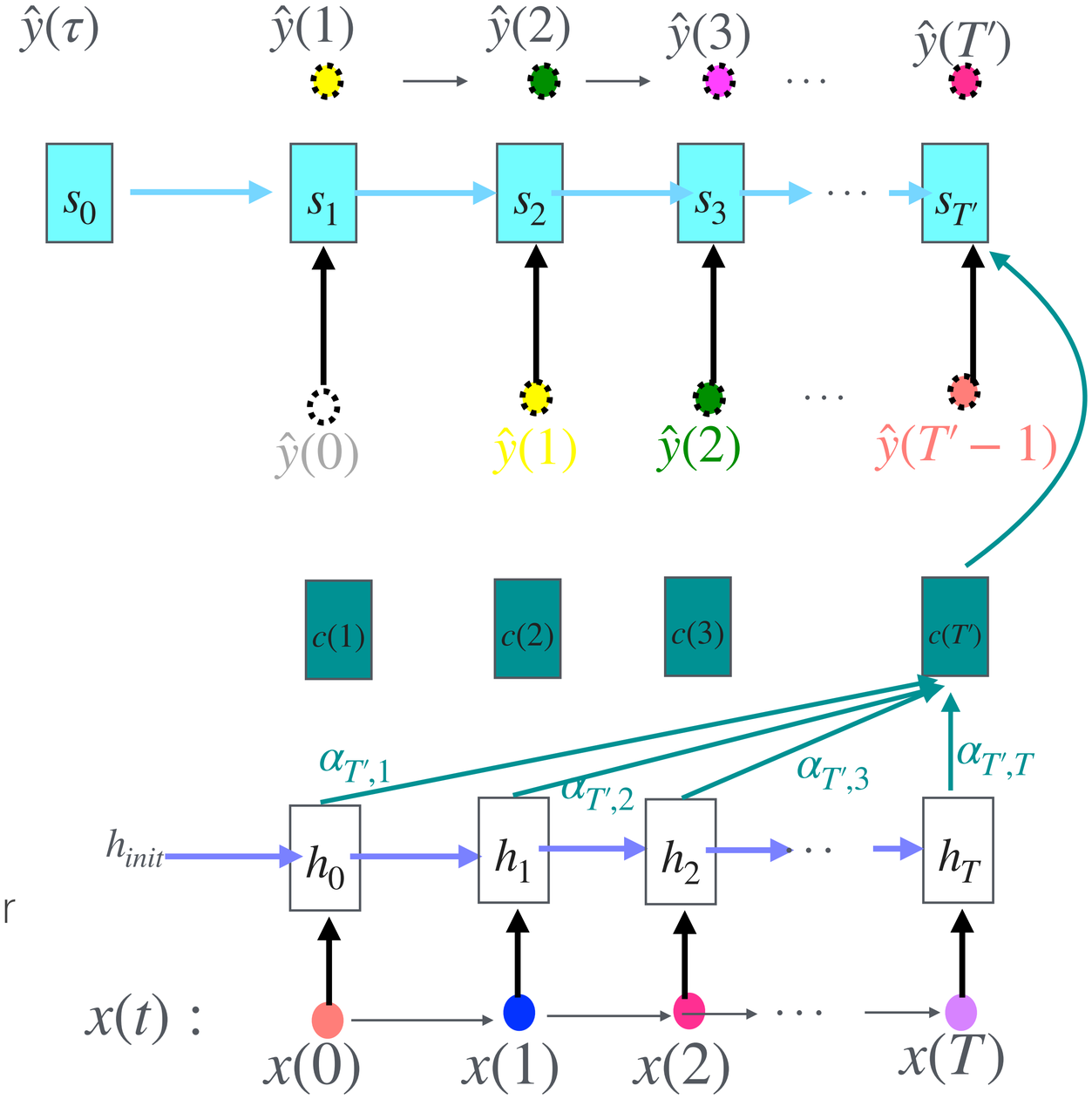


- We can fix the problem, easily!
- The weights $\alpha_{i,j}$ should be calculable functions based the depends on respective decoder state before the update (s_{i-1}) and the corresponding hidden states ($\{h_j\}$).
- They can have a form like:

$$\alpha_{i,j} = F(s_{i-1}, h_j)$$
$$c(i) = \sum \alpha_{i,j} h_j$$

- Function F, Intuitively need something to behave as:

(**a**)We want larger score ⇒ more influence
(**b**)All encoder positions should compete with each other
(**c**)The weights should be comparable across positions
(**d**)the final context should stay on a controlled scale



Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." *arXiv preprint arXiv:1409.0473* (2014).

- all of these are achieved by applying **softmax** to the scores across encoder positions! These are steps:
- 1) We first compute a relevance score between the current decoder state and each encoder state with a learnable function F: (F's parameters will be learned by the model) , (addresses **a**)

$$e_{i,j} = F(s_{i-1}, h_j)$$

- 2) Then we convert these scores into attention weights by normalizing across all encoder positions: (addresses **b,c,d**)

$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_k \exp(e_{i,k})} = softmax_j(e_{i,j})$$

- 3) Finally, we build the context vector as a weighted sum:

$$c_i = \sum_j \alpha_{i,j} h_j$$

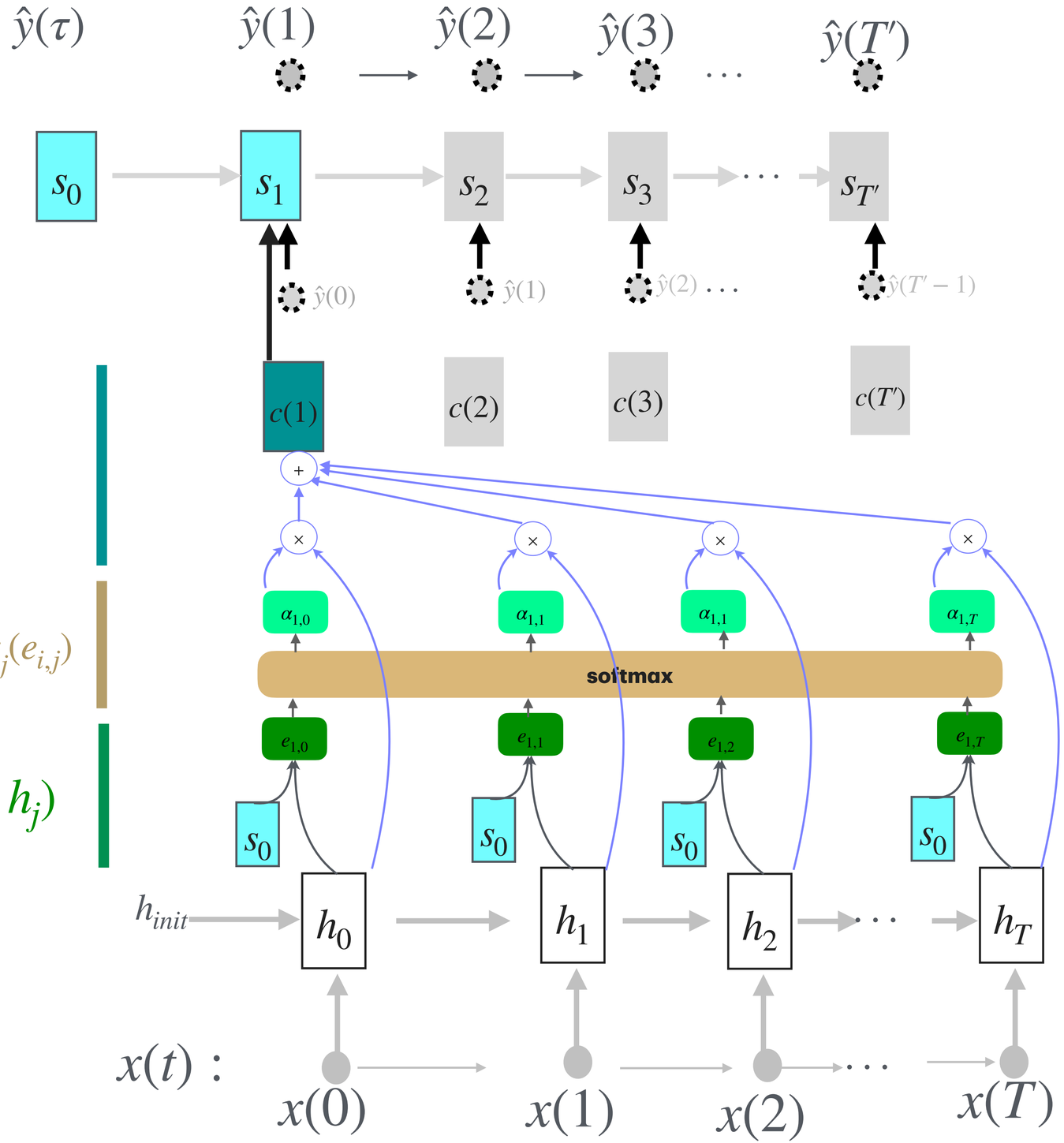
Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." *arXiv preprint arXiv:1409.0473* (2014).

- Now we can update the diagram accordingly:

$$c_i = \sum_j \alpha_{i,j} h_j$$

$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_k \exp(e_{i,k})} = softmax_j(e_{i,j})$$

$$e_{i,j} = F(s_{i-1}, h_j)$$

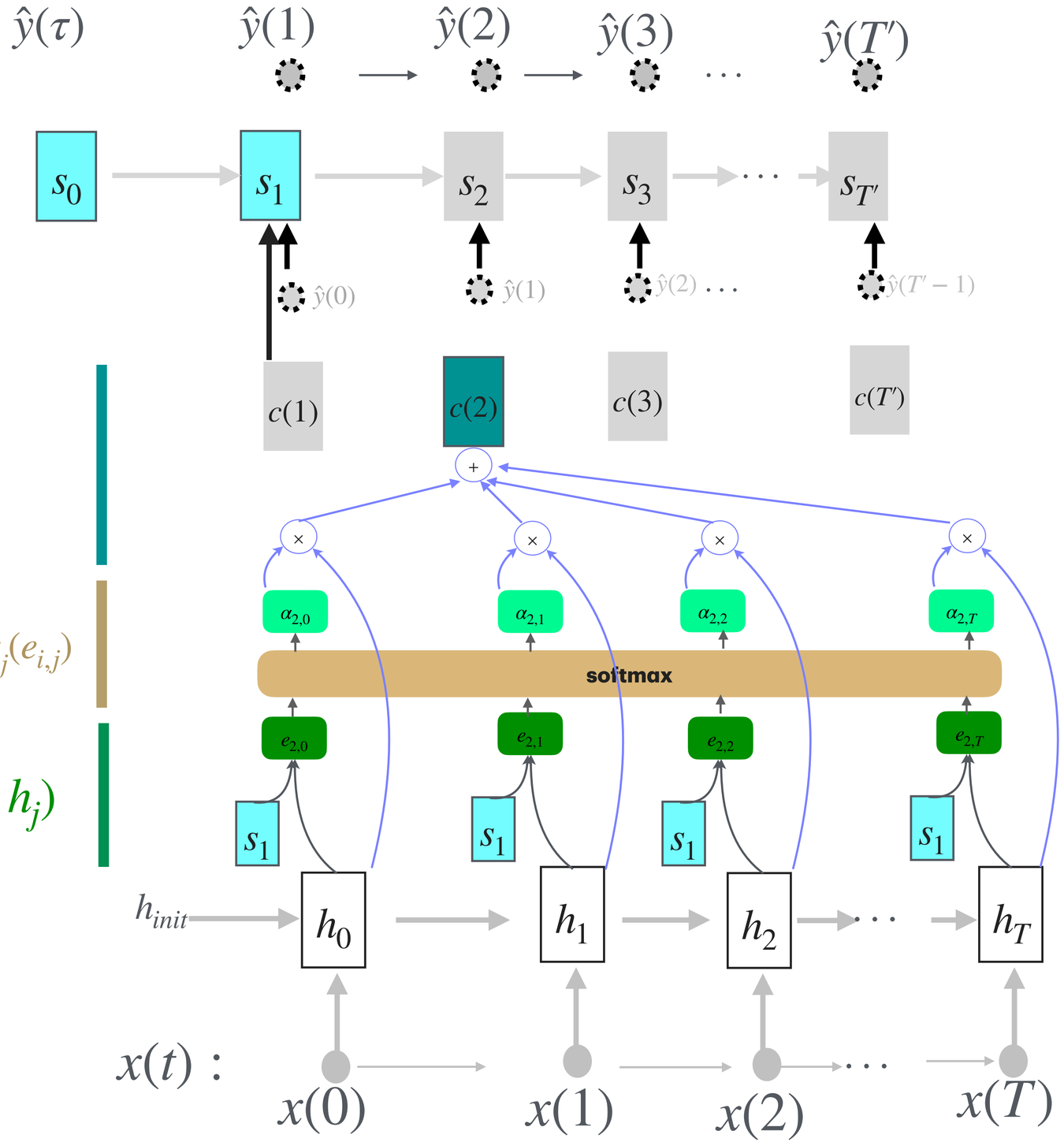


- Now we can update the diagram accordingly:

$$c_i = \sum_j \alpha_{i,j} h_j$$

$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_k \exp(e_{i,k})} = softmax_j(e_{i,j})$$

$$e_{i,j} = F(s_{i-1}, h_j)$$

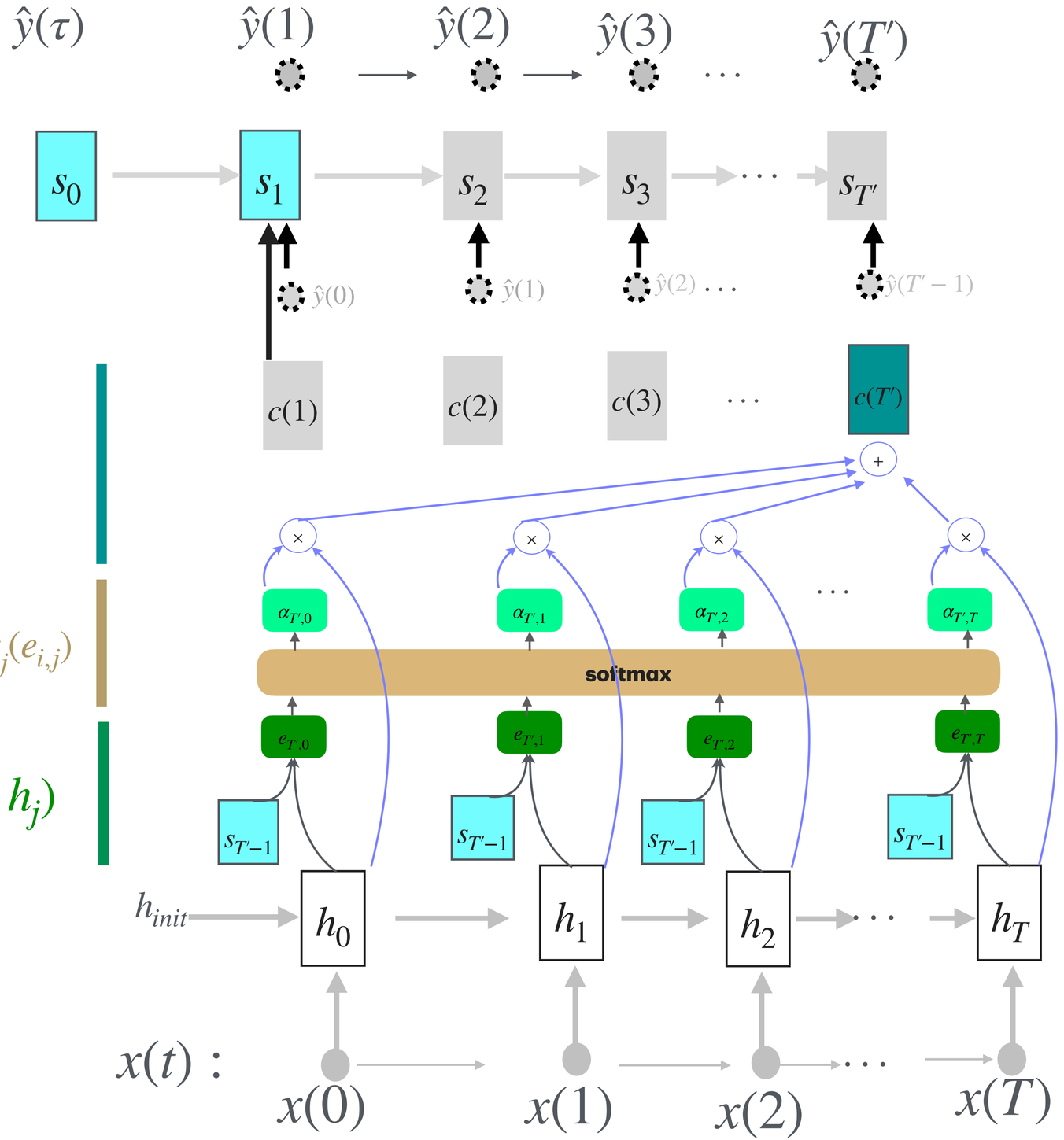


- Now we can update the diagram accordingly:

$$c_i = \sum_j \alpha_{i,j} h_j$$

$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_k \exp(e_{i,k})} = softmax_j(e_{i,j})$$

$$e_{i,j} = F(s_{i-1}, h_j)$$



- Now we can update the diagram accordingly:

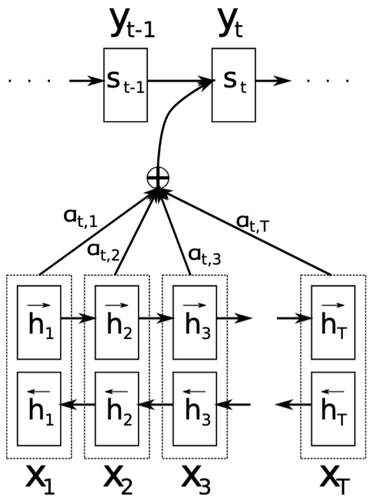
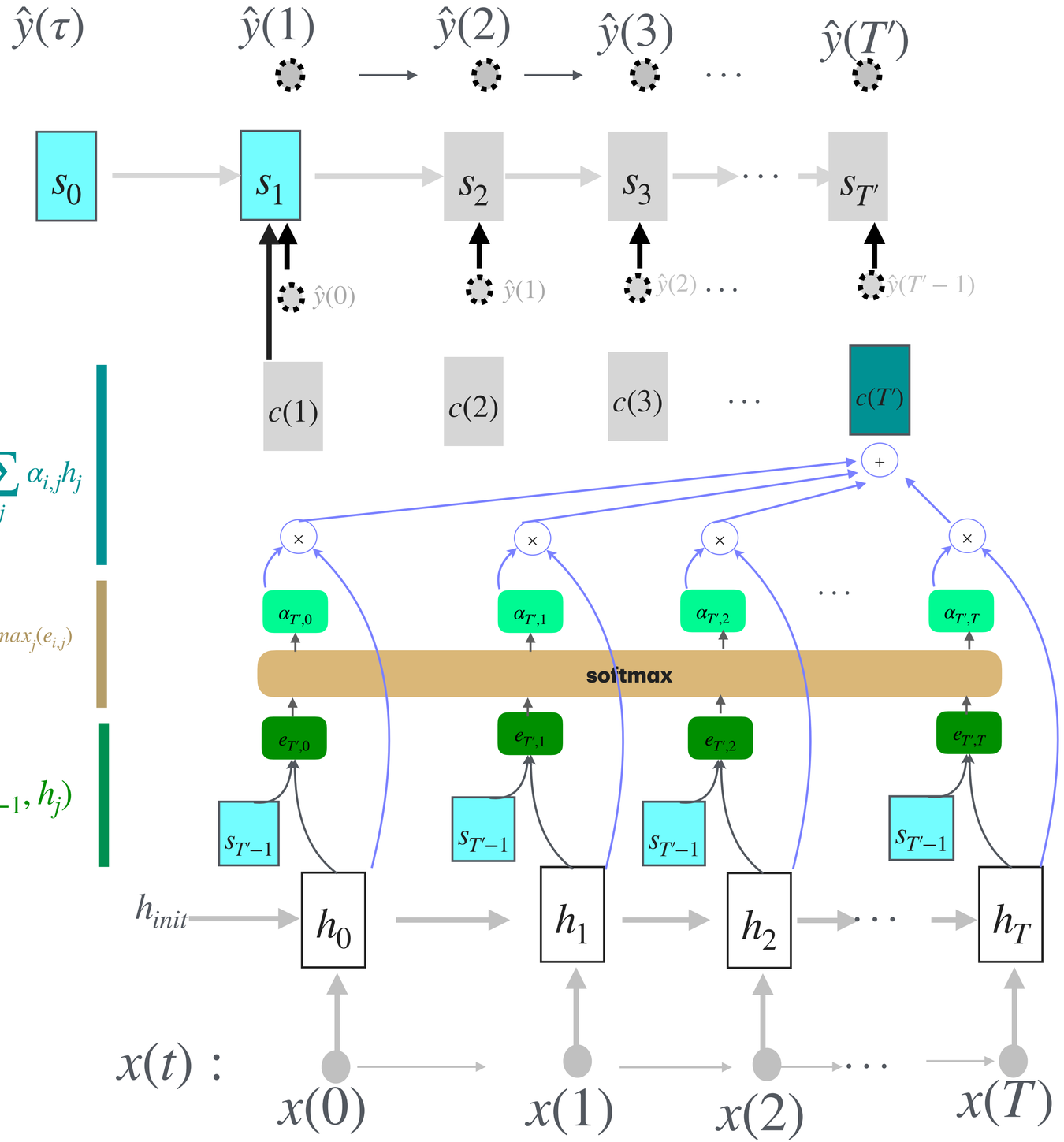


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

$$c_i = \sum_j \alpha_{i,j} h_j$$

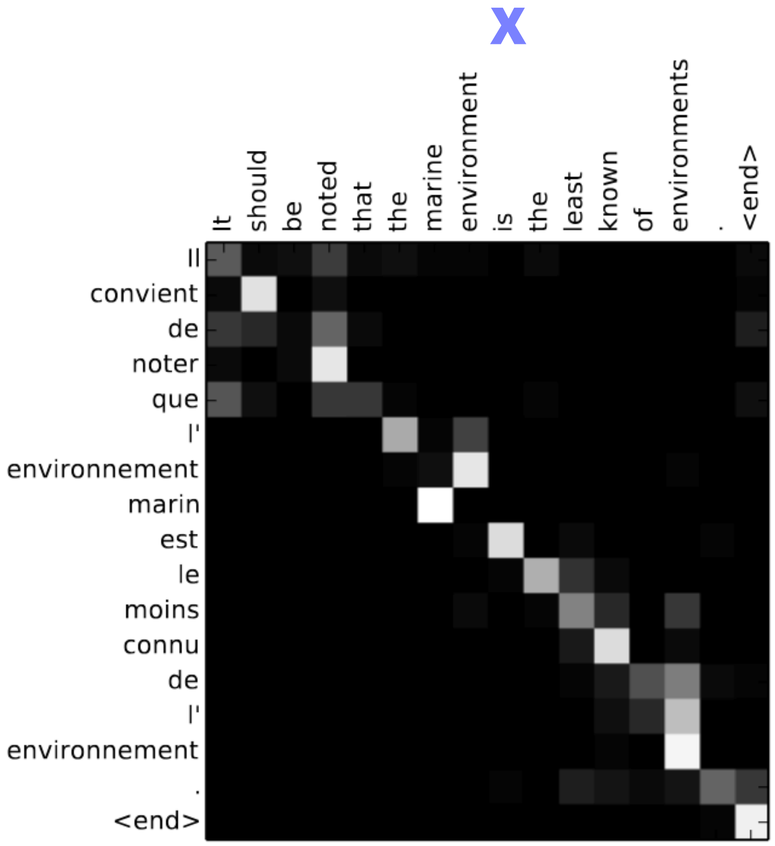
$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_k \exp(e_{i,k})} = softmax_j(e_{i,j})$$

$$e_{i,j} = F(s_{i-1}, h_j)$$



Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." *arXiv preprint arXiv:1409.0473* (2014).

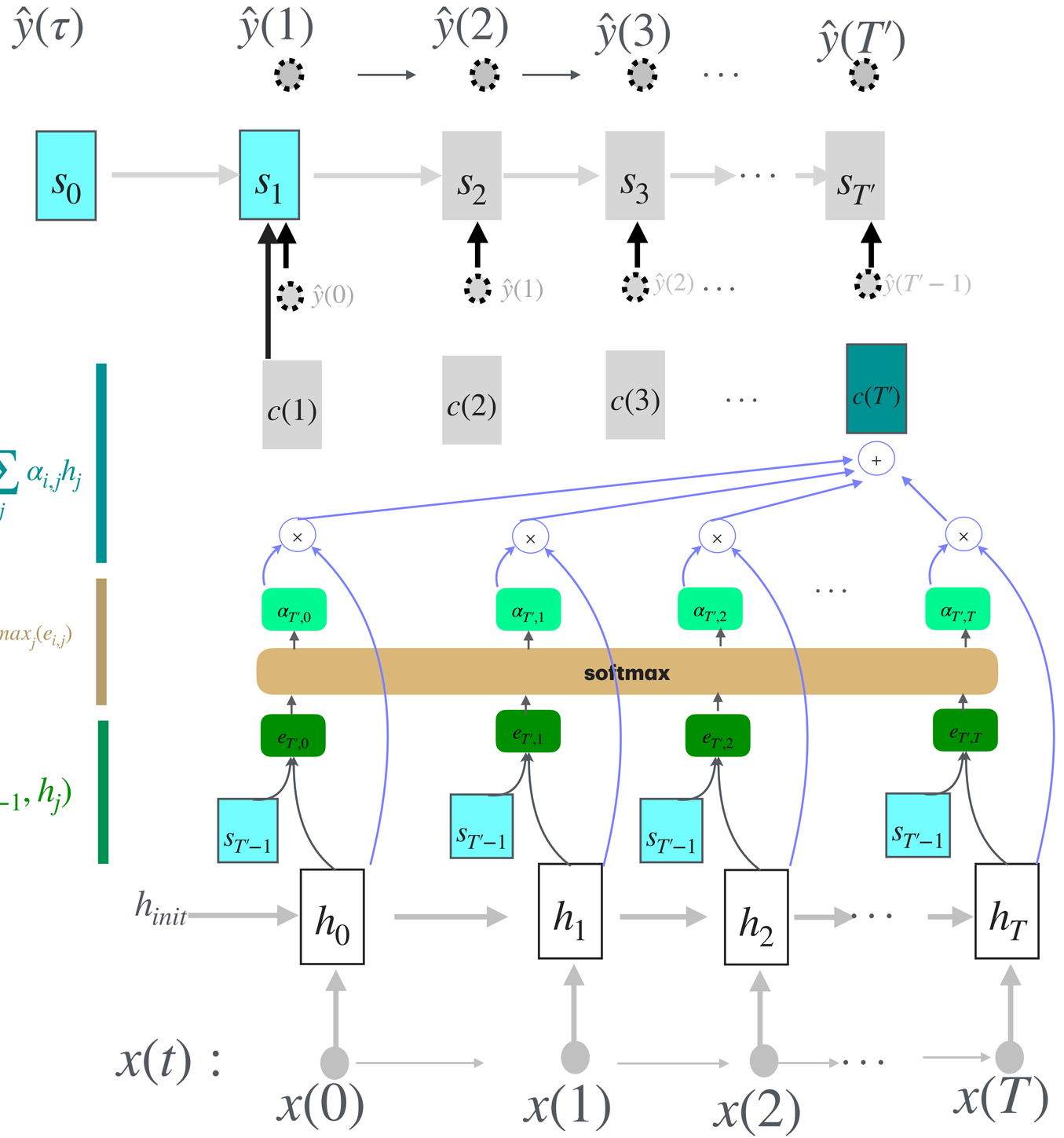
- Now we can update the diagram accordingly:



$$c_i = \sum_j \alpha_{i,j} h_j$$

$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_k \exp(e_{i,k})} = softmax_j(e_{i,j})$$

$$e_{i,j} = F(s_{i-1}, h_j)$$



Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." *arXiv preprint arXiv:1409.0473* (2014).

Recipe Card: **Autoregressive Seq2Seq + Attention**

Type of Learning: Supervised + Autoregressive
Data set: Input sequence **+** output sequence

Data

1- Load libraries: Basics + PyTorch
2-Load your sequence dataset **D:**
-Same as Autoregressive Recurrent Models (seq2seq)
3- Now set up the DataLoader for training/validation/testing.
-Same as **Autoregressive Recurrent Models** (seq2seq)

NN model

4-Build the *autoregressive seq2seq + attention model* (It has 4 parts):

A) Encoder (choose nn.RNN(...) nn.GRU(...) nn.LSTM(...))

$$x \rightarrow [\text{RNN / GRU / LSTM encoder}] \rightarrow H$$

-Input:$x \in [B, T, F]$. output is (Unlike seq2seq, we keep all encoder hidden states):

$$H = [h_1, \ldots, h_{T_x}], \qquad H \in [B, T_x, d_h],$$

B) Attention module: At decoder step τ, calculate the context vector c_τ:

1)elevance score**: $e_{\tau,j} = F(s_{\tau-1}, h_j)$

2)attention weights: $\alpha_{\tau,j} = \text{softmax}_j(e_{\tau,j})$

3)context vector for decoder step τ: $c_\tau = \sum_j \alpha_{\tau,j} h_j$

**note: the function F in PyTorch ca=ould be a small linear based module: nn.Linear(...)

C) Decoder Generate one output step at a time using previous decoder state, previous output, and current context:

$$s_\tau = f_{dec}(s_{\tau-1}, \hat{y}_{\tau-1}, c_\tau)$$

D) Output head:

$$s_\tau \rightarrow [\text{Linear layer}] \rightarrow out,\ \hat{y}_\tau$$

So, overall:

$x_1, \ldots, x_{T_x} \rightarrow$ Encoder $\rightarrow H \rightarrow$ Attention at each decoder step $\rightarrow c_1, c_2, \ldots \rightarrow \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{T_y}$

Training

5-Define a proper loss function based on Y

a-If Y is continuous, then use MSELoss or L1Loss

b-If Y is categorical, CrossEntropyLoss on raw logits

c-For autoregressive part:
1) compute loss over all relevant *decoder* steps
2) ignore padded positions using masking if needed
3) total loss = sum or mean over decoder steps

-Add regularization to prevent overfitting
-The loss will be the sum of all relevant losses

6-Define the train and test functions:

The details of forward rollout:

a-Encoder reads the input sequence and outputs all encoder states context $H = [h_1, \ldots h_{T_x}]$

b-At each decoder step τ calculate the context c_τ

c-update decoder state

d-generate output $\hat{y}(\tau)$

training: teacher forcing can be used by feeding true previous target to model $y(\tau - 1)$

Inference: use model's previous prediction $\hat{y}(\tau - 1)$

7-Define training and run the training:

-Same as standard recurrent network card