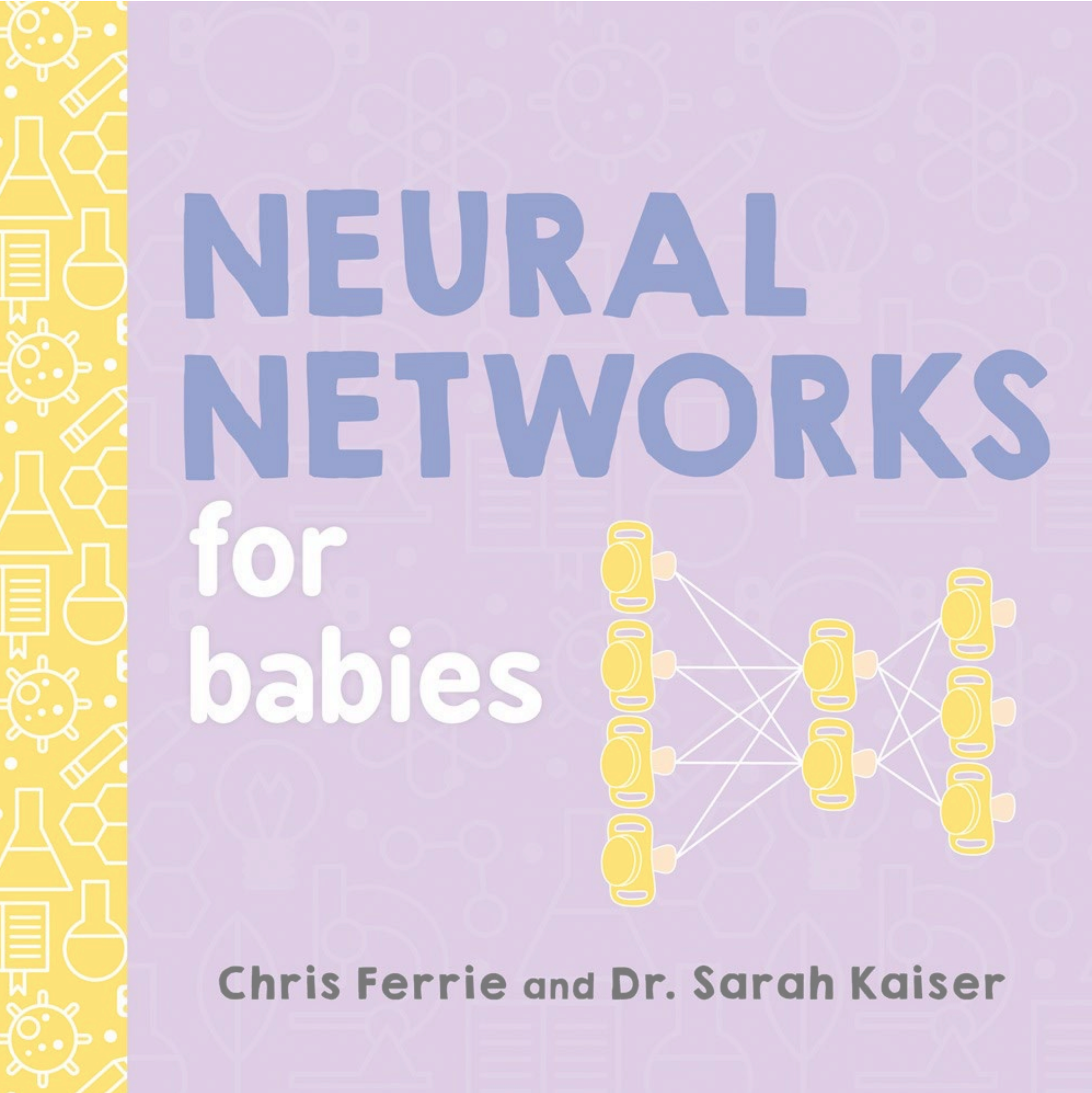
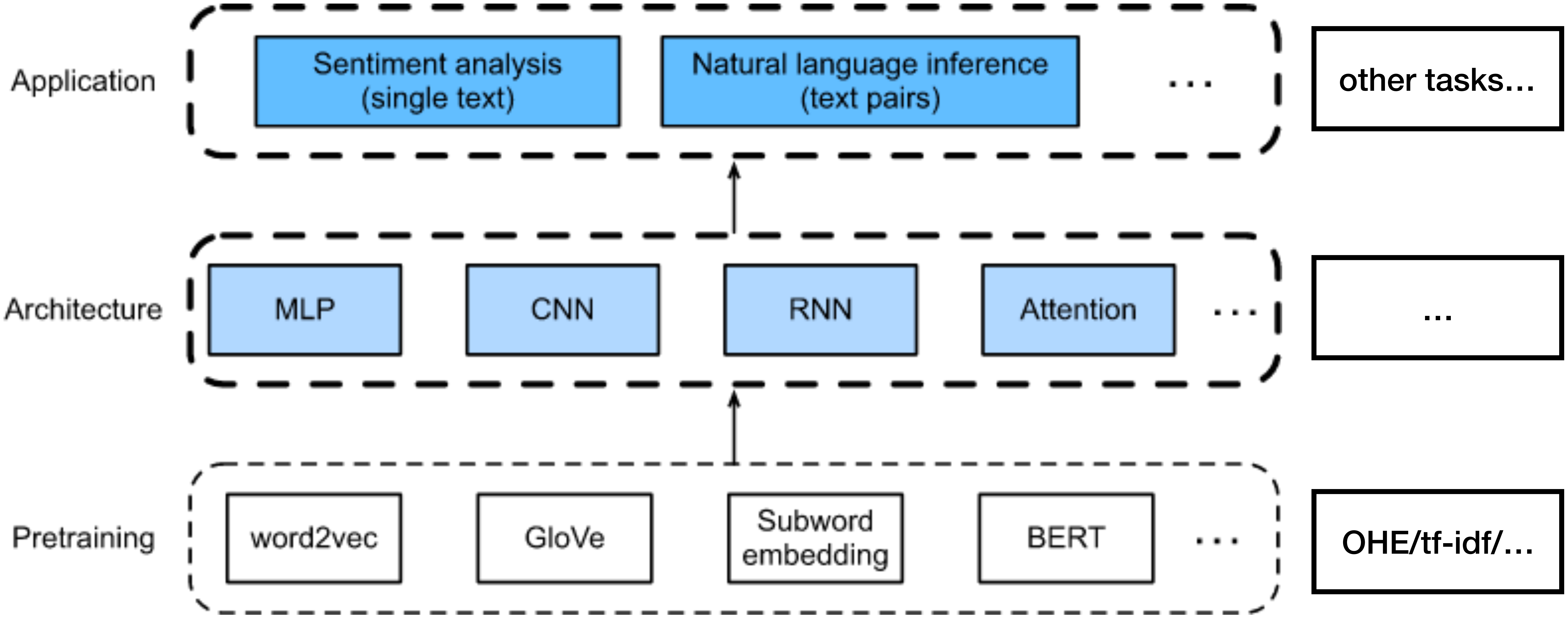
Neural networks NLP-applications

Eugeny Malyutin



Schema


Application
Sentiment analysis
(single text)
Natural language inference
(text pairs)
...
other tasks...
Architecture
MLP
CNN
RNN
Attention
...
...
Pretraining
word2vec
GloVe
Subword
embedding
BERT
...
OHE/tf-idf/...

RNN (recurrent neural networks):

From recSys:

- There is anchoring, based on someone else's opinion. Oscar lifts 0.5 avg rating for a movie. (The Revenant case)
- There is a *Hedonic adaptation*.
- There is seasonality.
- Movies minused due to social activity.

Another examples:

- BOW omits word order. The headline *dog bites man* is much less surprising than *man bites dog*
- Many users have particular time-dependent behavioural patterns connected to applications.
- Earthquakes are strongly correlated
- Humans interact with each other in a sequential nature

RNN vs LM:

- Language **n-gram** model with Markov assumption: $p(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$
- **Problems?**

RNN vs LM:

- Language **n-gram** model with Markov assumption: $p(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$
- **Problems:**
    - We need to retrain all model if we want to check the possible effect earlier then (t - n + 1)
    - Easy to overfit / need much bigger corpora for our |V|^n parameters
    - A lot of memory for |V|^n
    - Heuristics and tricks for unknown **bi-grams**
-

RNN vs LM:

- Language **n-gram** model with Markov assumption: $p(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$
- **Problems:**
    - …
- **Idea:**
    - Switch to **latent** model: $p(x_t \mid x_{t-1}, \ldots, x_1) \approx p(x_t \mid x_{t-1}, h_t)$.
    - Latent **state** h_t: $h_t = f(x_t, h_{t-1})$
- **Note**: hidden state != hidden layer

RNN with hidden states:

- Assume that we have $\mathbf{X}_t \in \mathbb{R}^{n\times d}$, $t = 1,\ldots,T$ in iteration and $\mathbf{H}_t \in \mathbb{R}^{n\times h}$ as hidden variable.
- We already saved H_{t-1} from prev iteration.

  Welcome $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ as mapping for H_{t-1} to H_t.

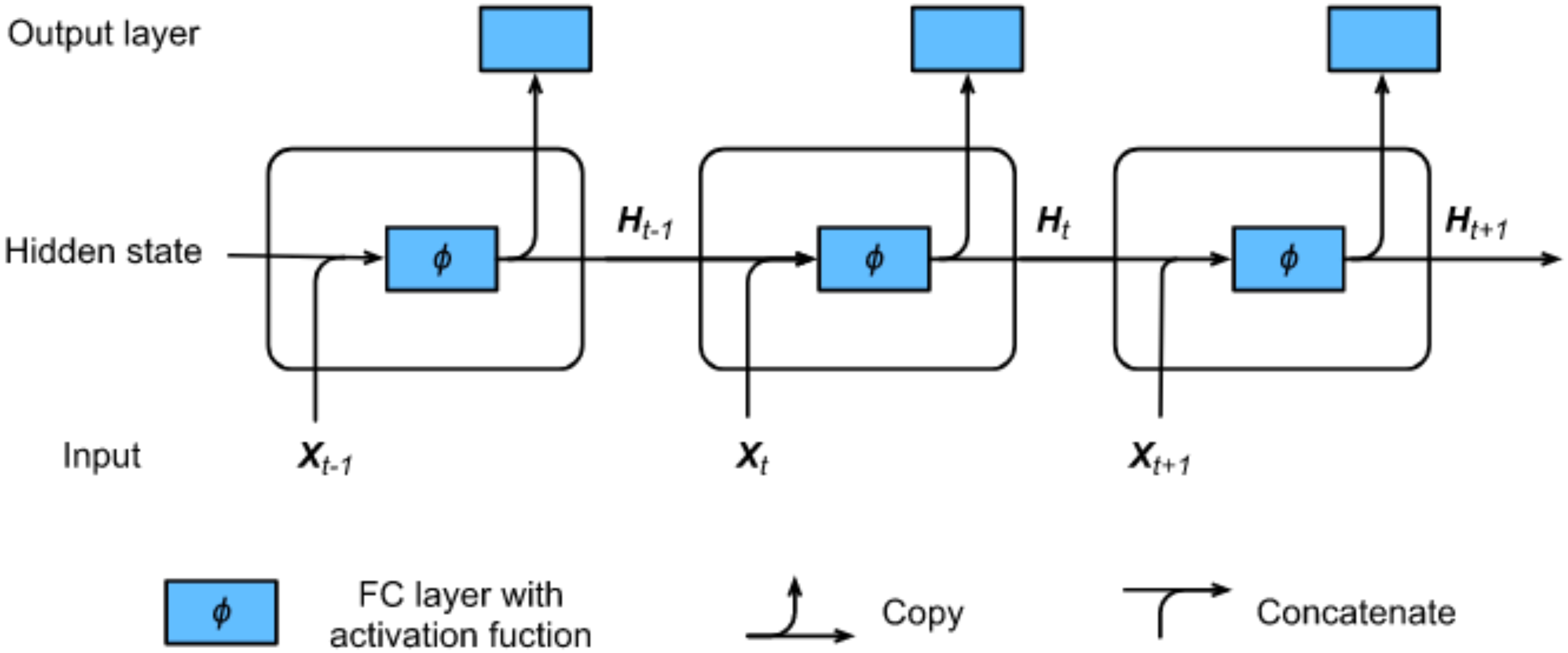
  Now we use $\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h)$

  to derive H_t from H_{t-1} and current input X_t.
  $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$ - additional weight parameter, $b_h -$ bias parameter.

- Finally the output layer: $\mathbf{O}_t = \mathbf{H}_t\mathbf{W}_{hq} + \mathbf{b}_q$ with their own parameters W_hq and b_q
- **Note:** RNNs always use these model parameters, even for different timesteps.

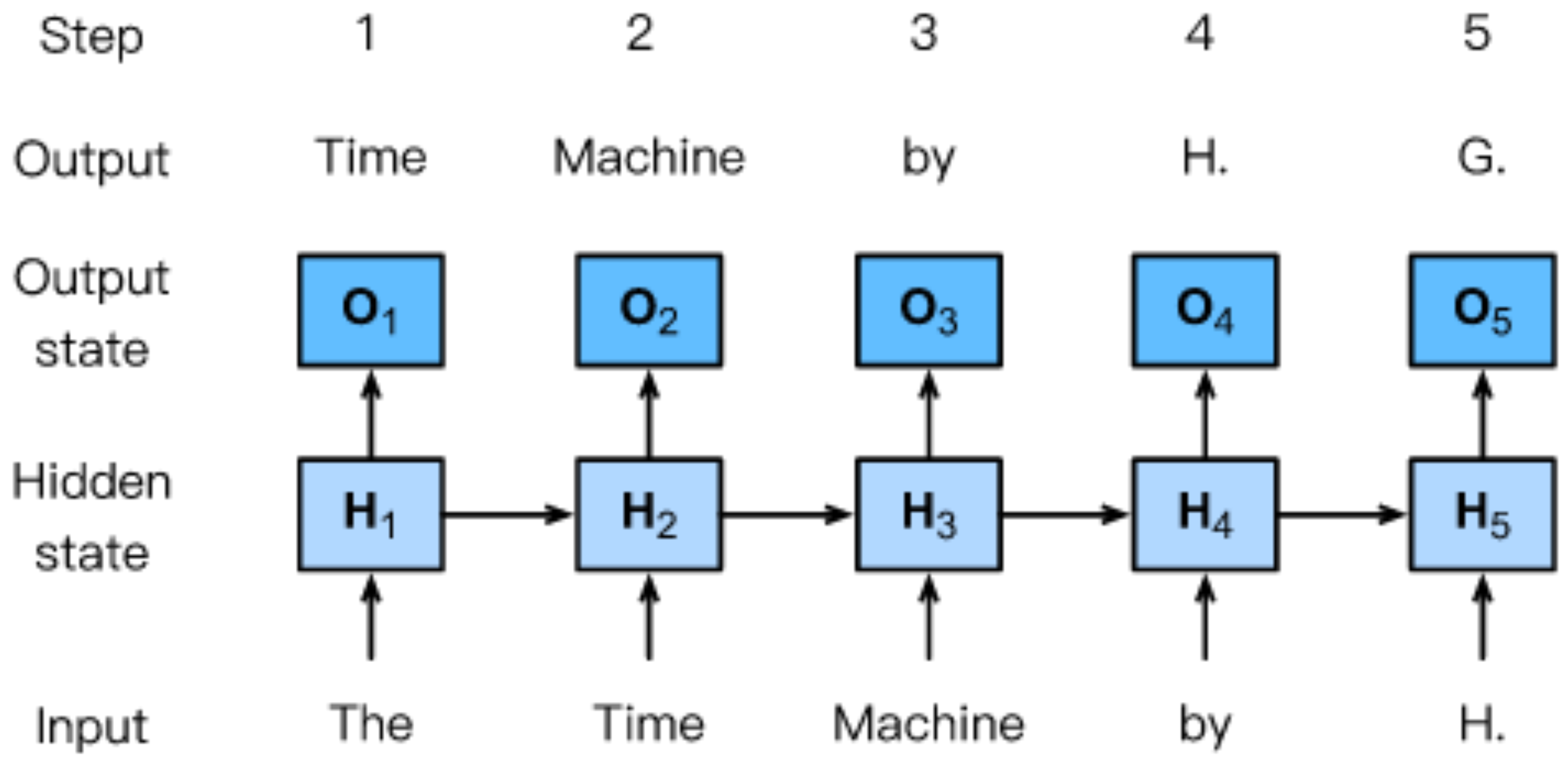
RNN:

- $\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h)$
- $\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q$



RNN as LM:

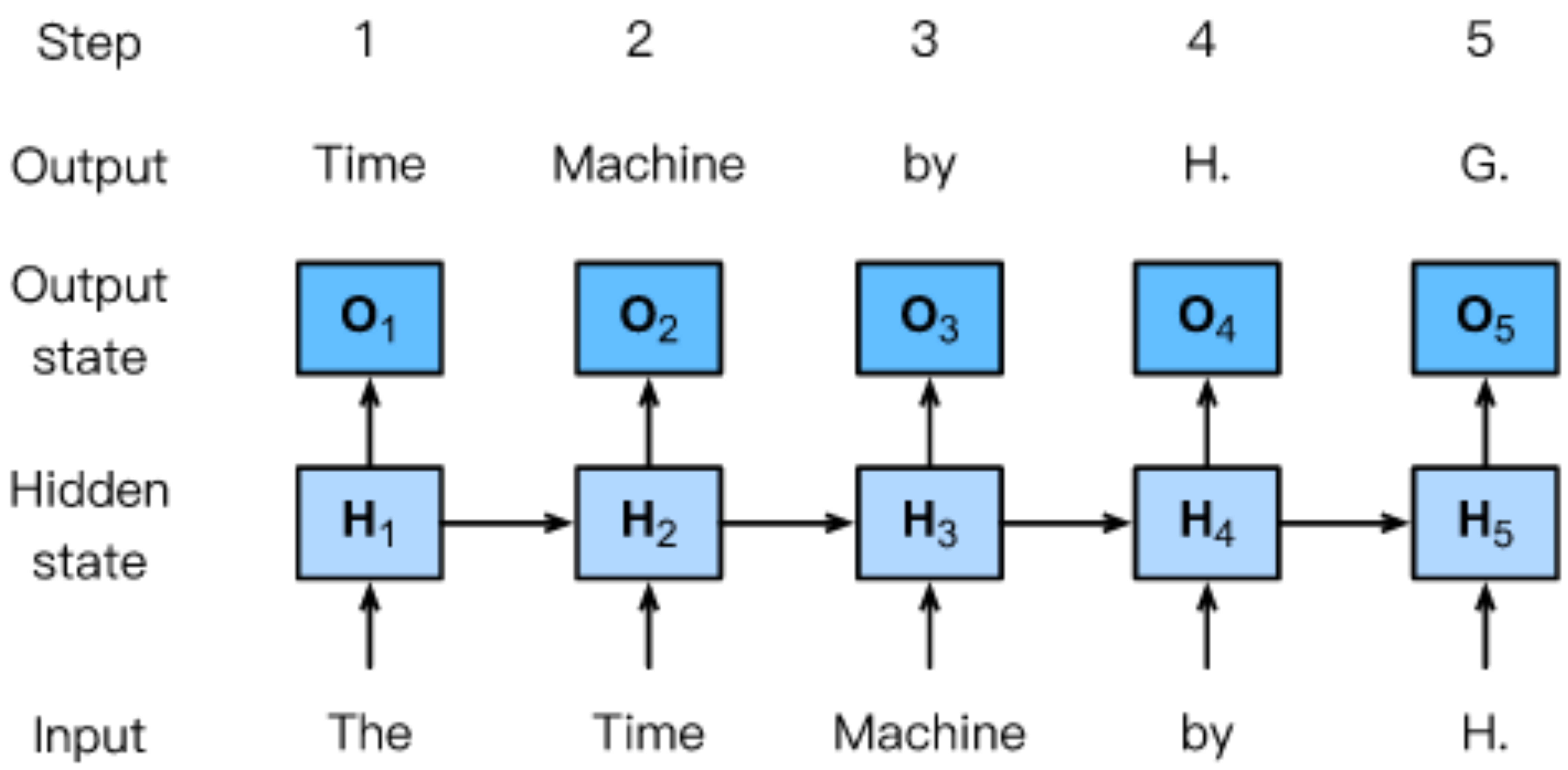
- Generate «The Time Machine by H. G. Wells» word by word. From t-1 previous word and state H_t-1 through H_t and X_t to predicted O_t (predicted X_t+1)



Start H state initialization discussion

RNN as LM:

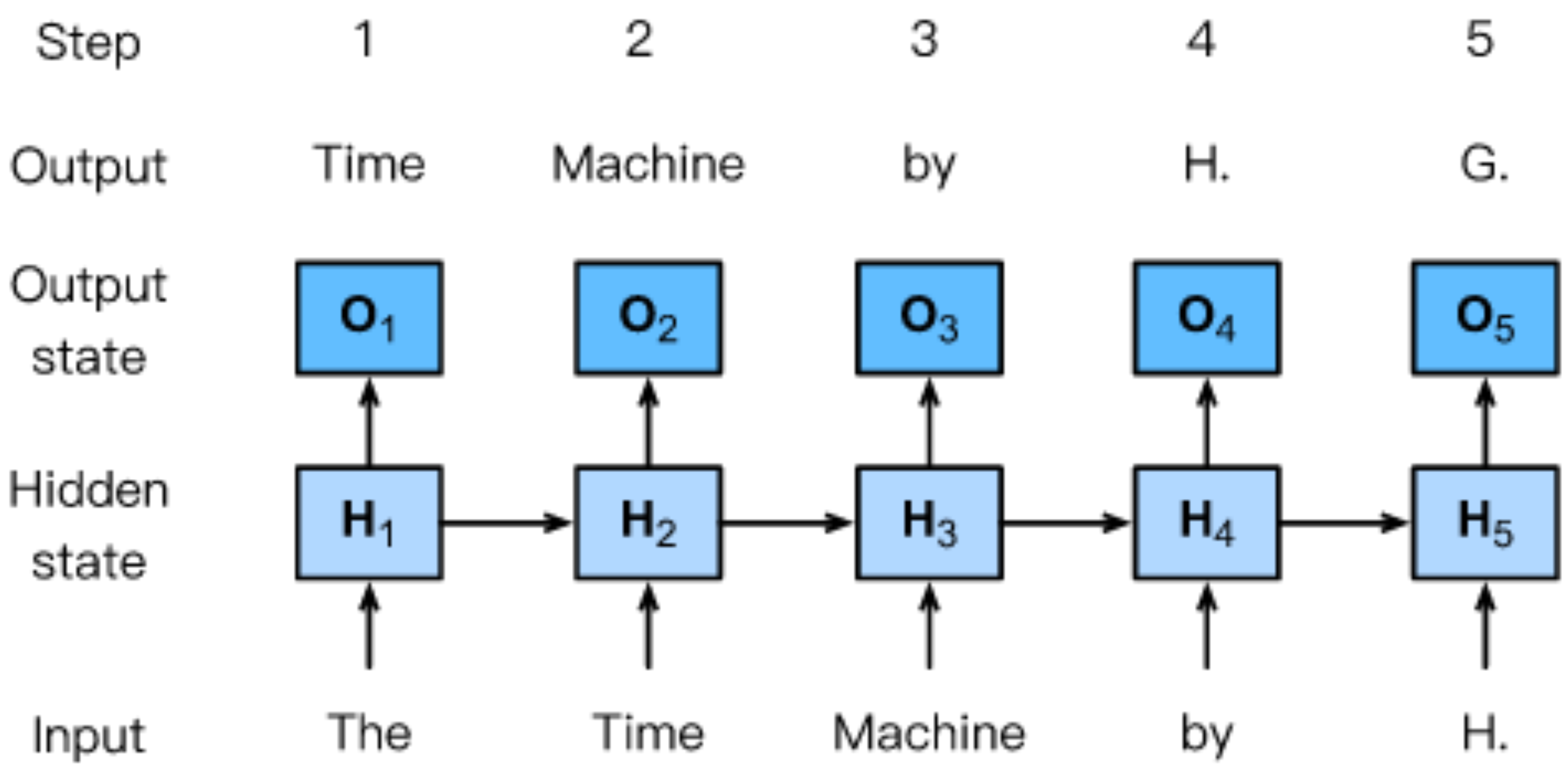
- Generate «The Time Machine by H. G. Wells»
- In practice, each word is presented by a d-dimensional and batch size n>1 so X_t at timestamp t becomes n x d matrix



Start H state initialization discussion

RNN as LM:

- A network that uses recurrent computation is called a recurrent neural network (RNN).
- The hidden state of the RNN can capture historical information of the sequence up to the current timestep.
- The number of RNN model parameters does not grow as the number of timesteps increases.
- We can create language models using a character-level RNN.



Start H state initialization discussion

BPTT (Backpropagation Through Time):

- Here is our network: $h_t = f(x_t, h_{t-1}, w_h)$ and $o_t = g(h_t, w_o)$.
- And here iterating over triplets (h_t, x_t, o_t) we can compute our loss:

  $$L(x, y, w_h, w_o) = \sum_{t=1}^{T} l(y_t, o_t).$$

- And through a chain rule

$$\begin{aligned}\partial_{w_h} L &= \sum_{t=1}^{T} \partial_{w_h} l(y_t, o_t) \\ &= \sum_{t=1}^{T} \partial_{o_t} l(y_t, o_t) \partial_{h_t} g(h_t, w_h) \left[\partial_{w_h} h_t\right].\end{aligned}$$

BPTT (Backpropagation Through Time):

$$\partial_{w_h} L = \sum_{t=1}^{T} \partial_{w_h} l(y_t, o_t)$$

Through a chain rule

- $$= \sum_{t=1}^{T} \partial_{o_t} l(y_t, o_t) \partial_{h_t} g(h_t, w_h) \left[\partial_{w_h} h_t \right].$$

- $$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w_h) + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \partial_{h_{j-1}} f(x_j, h_{j-1}, w_h) \right) \partial_{w_h} f(x_i, h_{i-1}, w_h).$$

BPTT (Backpropagation Through Time):

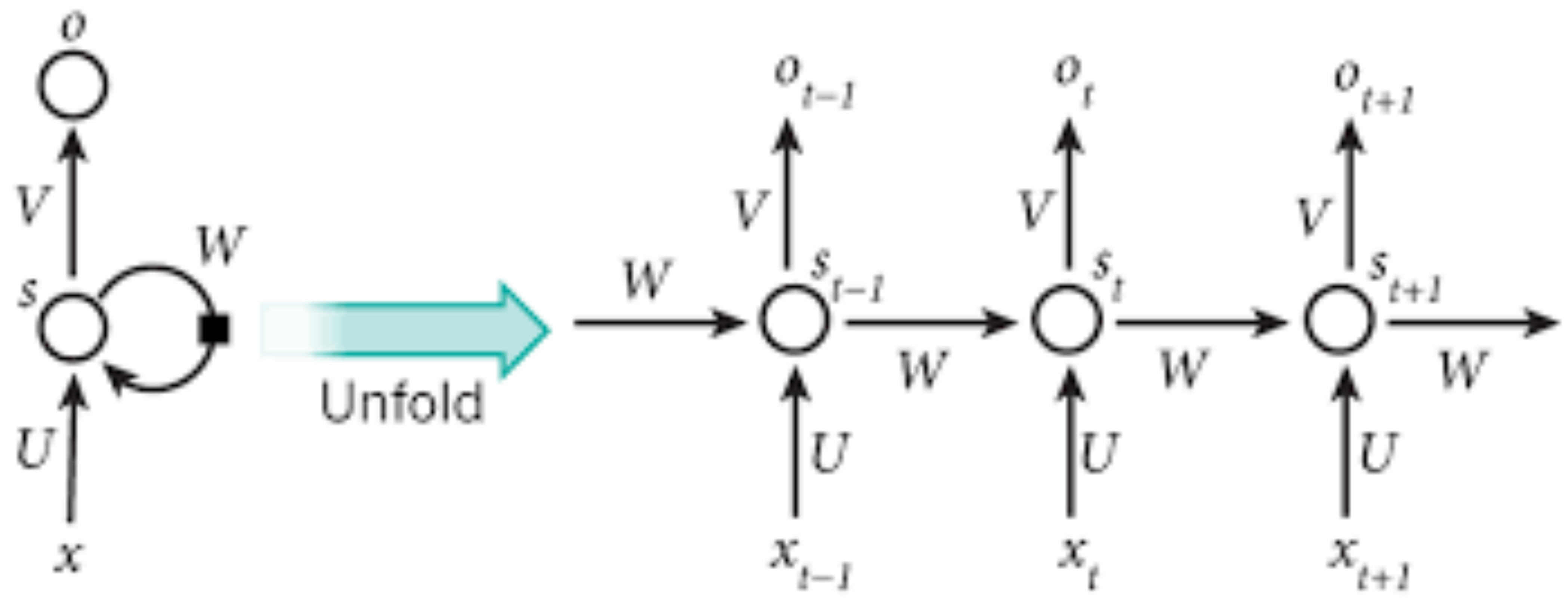
- Through a chain rule

$$\partial_{w_h} L = \sum_{t=1}^{T} \partial_{w_h} l(y_t, o_t)$$
$$= \sum_{t=1}^{T} \partial_{o_t} l(y_t, o_t) \partial_{h_t} g(h_t, w_h) \left[\partial_{w_h} h_t \right].$$

- $$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w_h) + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \partial_{h_{j-1}} f(x_j, h_{j-1}, w_h) \right) \partial_{w_h} f(x_i, h_{i-1}, w_h).$$

- **Problems?**

BPTT (Backpropagation Through Time):

- 

$$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w_h) + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \partial_{h_{j-1}} f(x_j, h_{j-1}, w_h) \right) \partial_{w_h} f(x_i, h_{i-1}, w_h) \,.$$

- **Problems?**

BPTT (Backpropagation Through Time):

- **Compute the full sum:** slow, gradients blow up, unstable
- **Truncate sum after τ steps:** terminating the sum above at $\partial wh\_\{t{-}\tau\}$.
  - The approximation **error** is thus given by $\partial_h f(x_t, h_{t-1}, w) \partial_w h_{t-1}$ (multiplied by a product of gradients involving $\partial h f$).
  - Called as **truncated BPTT**
  - Works well in practice.
  - Leads for short-term truncated models.

$$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w_h) + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \partial_{h_{j-1}} f(x_j, h_{j-1}, w_h) \right) \partial_{w_h} f(x_i, h_{i-1}, w_h) \, .$$

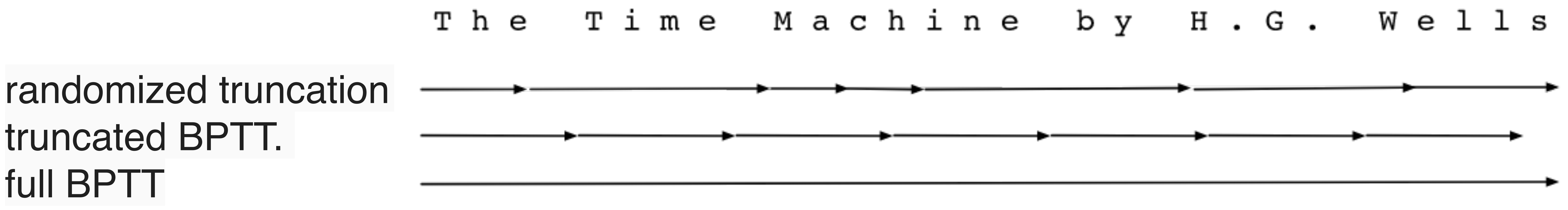
BPTT (Backpropagation Through Time):

- **Compute the full sum:** slow, gradients blow up, unstable
- **Truncate sum after τ steps:** terminating the sum above at $\partial wh\_\{t{-}\tau\}$.
- **Randomized Truncation:** Replace gradient to randomised variable:
  - $P(\xi_t = 0) = 1 - \pi$ and $P(\xi_t = \pi^{-1}) = \pi$
  - Replace gradient to: $z_t = \partial_w f(x_t, h_{t-1}, w) + \xi_t \partial_h f(x_t, h_{t-1}, w) \partial_w h_{t-1}$. and $E[z_t] = \partial_w h_t$
  - In theory better then truncated. **Practically not.**

$$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w_h) + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \partial_{h_{j-1}} f(x_j, h_{j-1}, w_h) \right) \partial_{w_h} f(x_i, h_{i-1}, w_h) \, .$$

BPTT (Backpropagation Through Time):

- **Compute the full sum:** slow, gradients blow up, unstable
- **Truncate sum after τ steps:** terminating the sum above at $\partial w h\_\{t-\tau\}$.
- **Randomized Truncation:** Replace gradient to randomised variable:



$$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w_h) + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \partial_{h_{j-1}} f(x_j, h_{j-1}, w_h) \right) \partial_{w_h} f(x_i, h_{i-1}, w_h) .$$

BPTT in details:

- Decomposing **W:** $\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1}$ **and** $\mathbf{o}_t = \mathbf{W}_{oh}\mathbf{h}_t$.
- And computing: $\dfrac{\partial L}{\partial \mathbf{W}_{hh}}$
- $\partial_{\mathbf{W}_{oh}} L = \sum_{t=1}^{T} \text{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{h}_t\right),$

$$\partial_{\mathbf{W}_{hh}} \mathbf{h}_t = \sum_{j=1}^{t} \left(\mathbf{W}_{hh}^\top\right)^{t-j} \mathbf{h}_j$$

(…some inference…)

- $$\partial_{\mathbf{W}_{hx}} \mathbf{h}_t = \sum_{j=1}^{t} \left(\mathbf{W}_{hh}^\top\right)^{t-j} \mathbf{x}_j \,.$$

BPTT in details:

$$\partial_{\mathbf{W}_{hh}} \mathbf{h}_t = \sum_{j=1}^{t} \left(\mathbf{W}_{hh}^\top\right)^{t-j} \mathbf{h}_j$$

$\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1}$ **and** $\mathbf{o}_t = \mathbf{W}_{oh}\mathbf{h}_t$.

- $\partial_{\mathbf{W}_{hx}} \mathbf{h}_t = \sum_{j=1}^{t} \left(\mathbf{W}_{hh}^\top\right)^{t-j} \mathbf{x}_j$.
- Store intermediate
- Large powers of (W^T_{hh})^{t-j} leads to numerical problems:
  - With eigenvalues > 1 gradients diverge (aka blow up)
  - With eigenvalues <1 gradients vanishes
- Of course we can fix diverge (not vanishing
  by clipping $\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}$. , but it hurts convergence.

GRU (Gate recurrent unit):

Motivation:

- We might encounter a situation where an **early observation is highly significant** for predicting all future observations. (Checksum case)

- We might encounter situations where **some symbols carry no pertinent observation** (HTML code in sentiment analysis case).

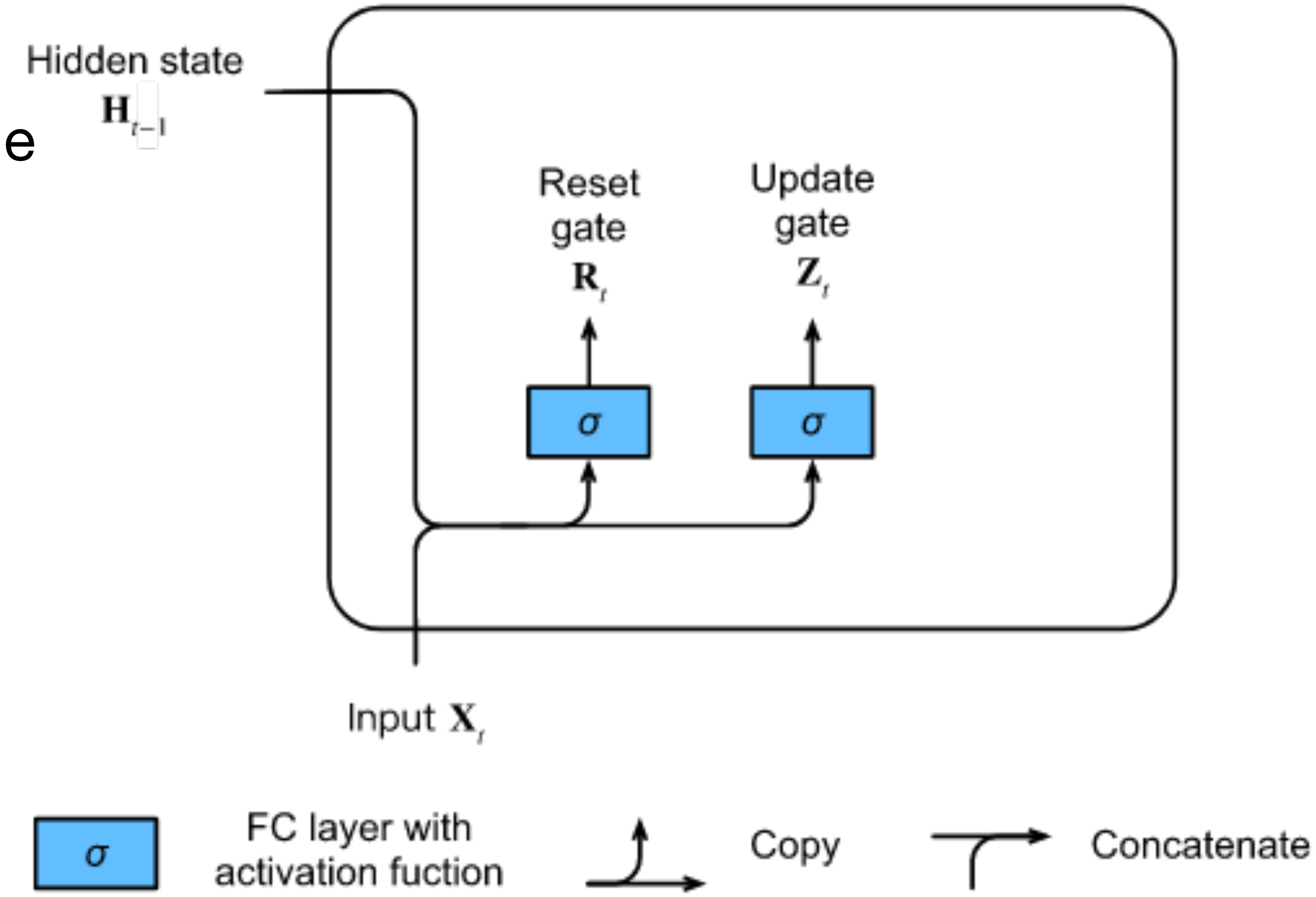
- We might encounter situations where there is a logical break between parts of a sequence. (Chapter X. case)

Reset gates and update gates:

- Reset and update variables are vectors with entries in (0, 1) and size h.
- Reset variable: **how much of the previous state** we **still want to remember:**
- Update variable: allow us to control how much of the new state is just a copy of the old state

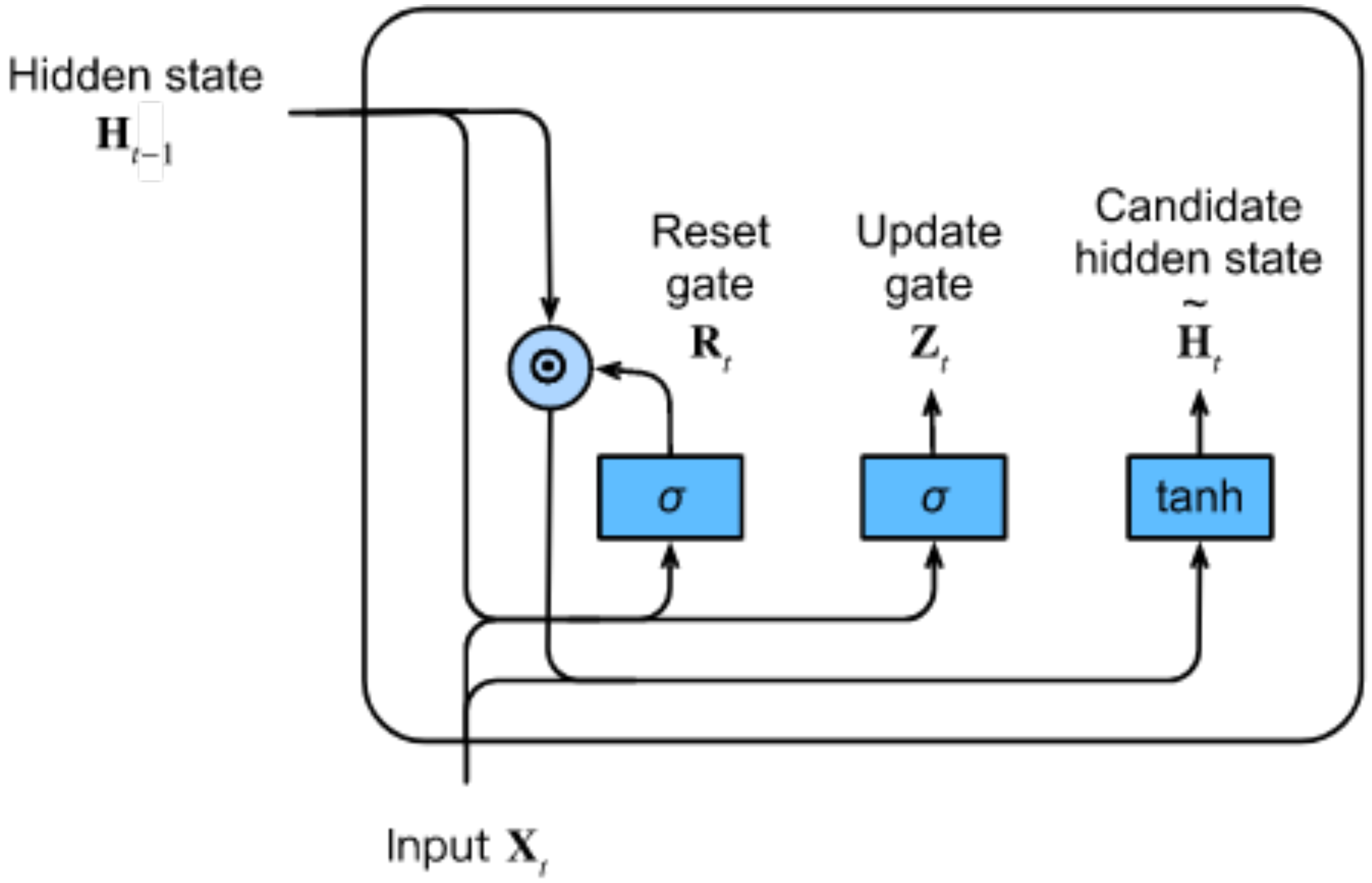
- Hidden state: $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$
  Reset variable: $\mathbf{R}_t \in \mathbb{R}^{n \times h}$
  Update variable: $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$

$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z)\ .$$

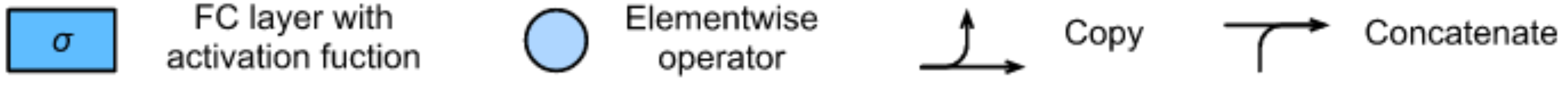


Reset gates in action:

- In a convenient RNN: $\mathbf{H}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h)$.
- Now we use *candidate* hidden state: $\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + \left(\mathbf{R}_t \odot \mathbf{H}_{t-1}\right)\mathbf{W}_{hh} + \mathbf{b}_h)$.
- If R_t is close to 1: we recover a convinien RNN
- If R_t is close to 0 we omit any pre-existent state and accept new MLP(X_t) as a state



$$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z) .$$

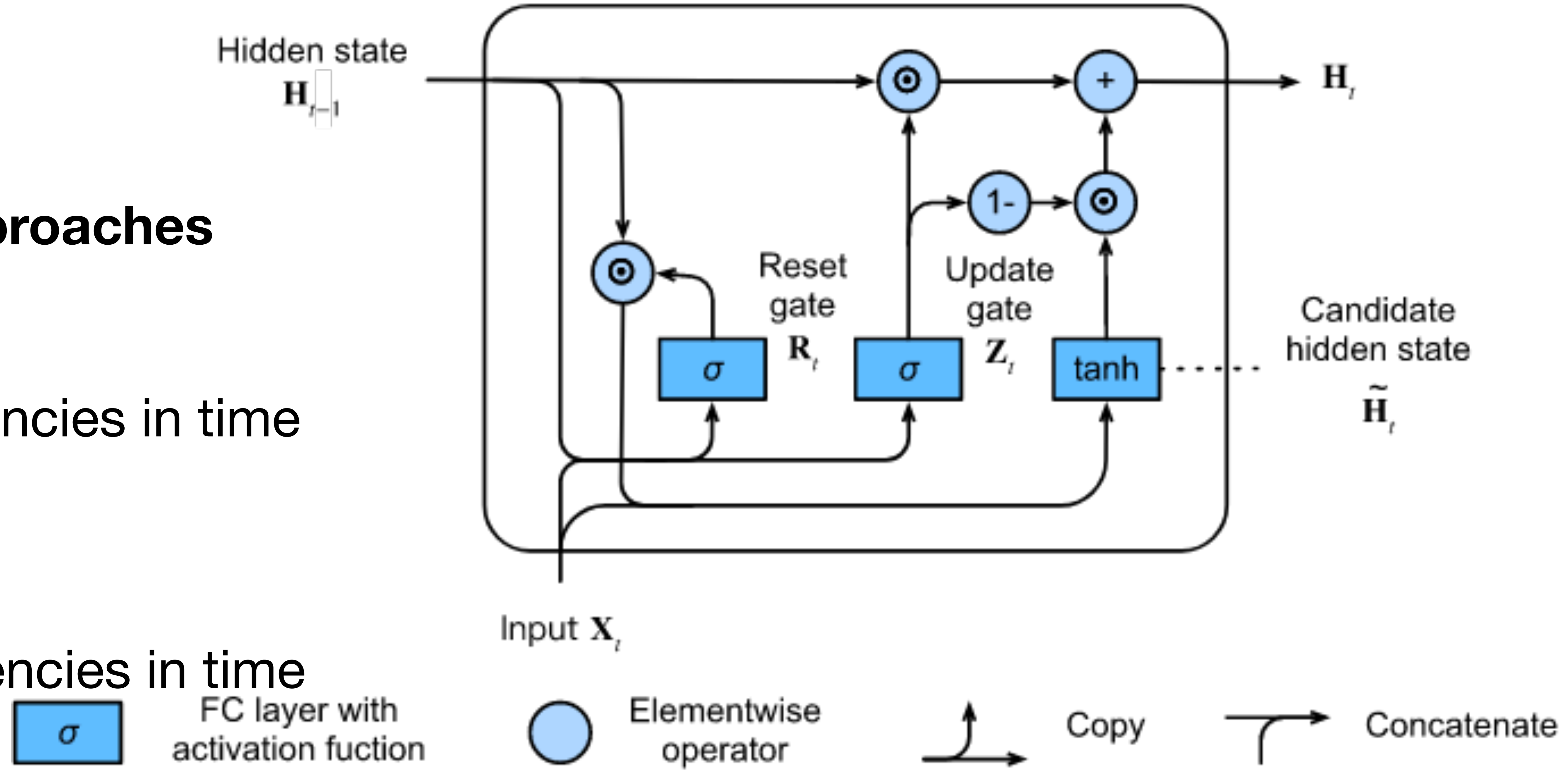


Update gates in action:

- Update gate determines an extent of how much new state H_t is just H_{t-1} and how much candidate $\tilde{\mathbf{H}}_t$ is used:
$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$.

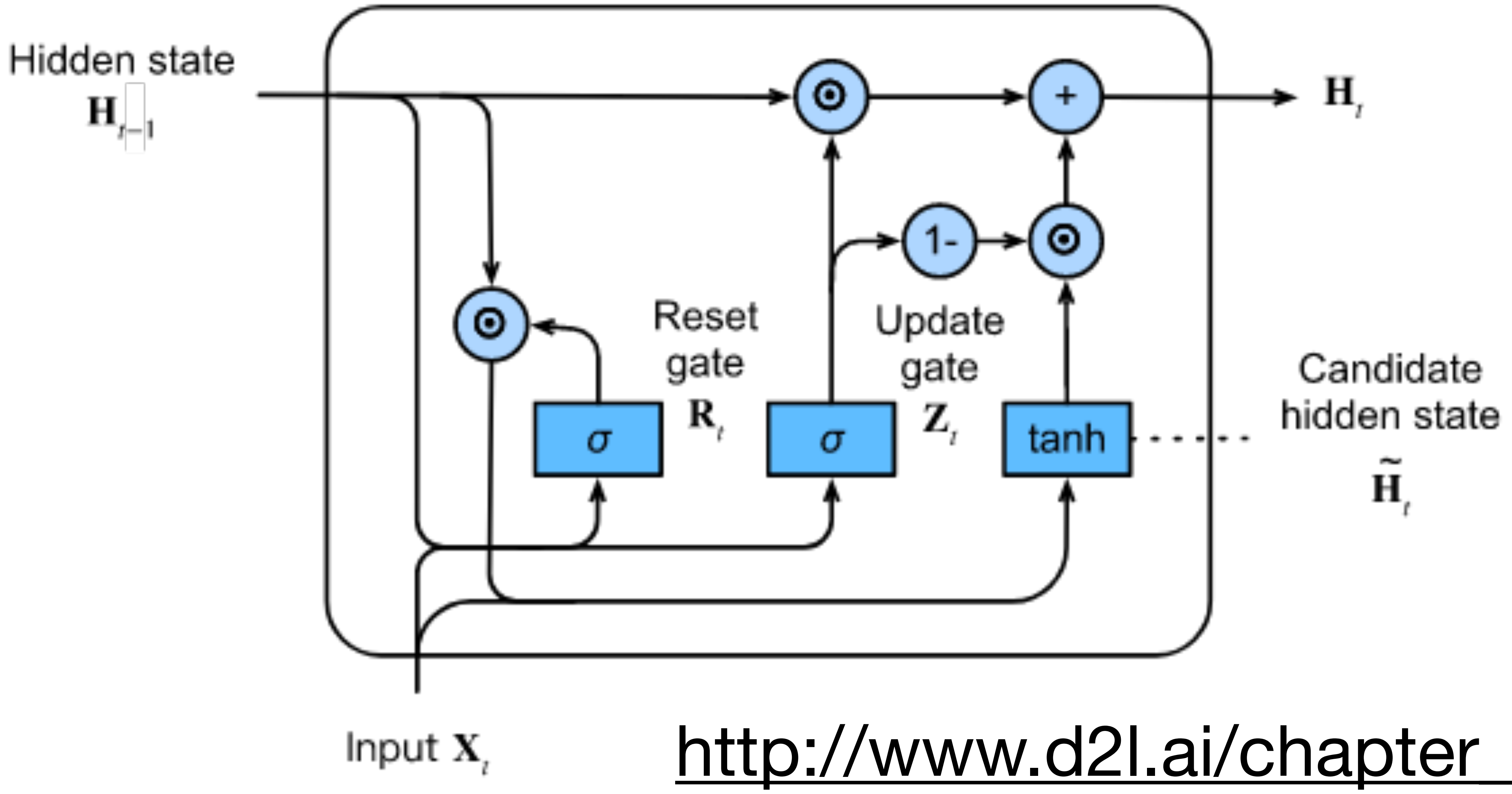
$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z) .$$

- Update gate Z:
    - Whenever **Z** is **close** to **1** — we omit current timestamp updates and just **recover previous state**
    - Whenever **Z** is **close** to **0** — **new state approaches candidate one**.
- .Reset gates help capture short-term dependencies in time series.
- Update gates help capture long-term dependencies in time series



GRU summary:

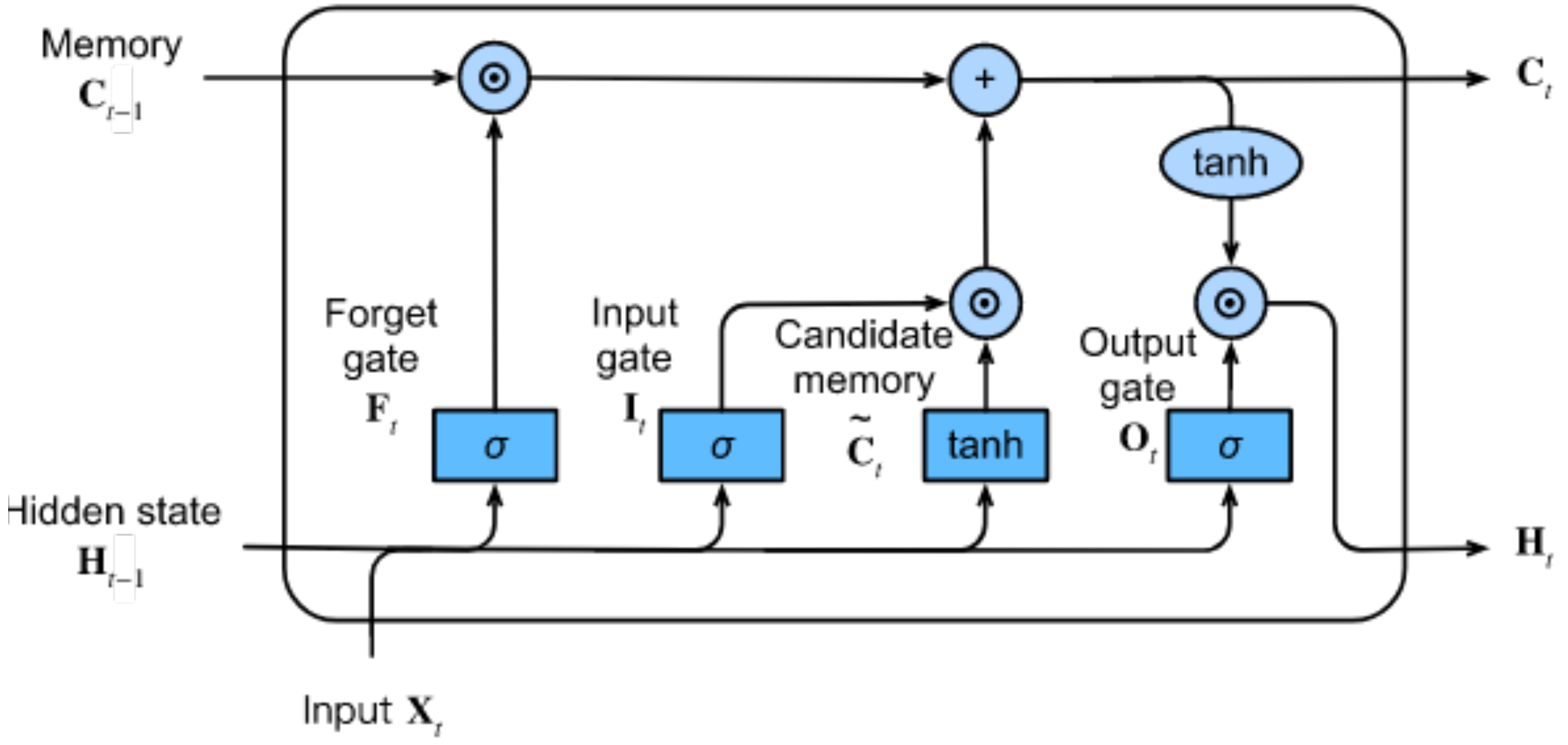
- Gated recurrent neural networks are better at capturing dependencies for time series with large timestep distances.
- Reset gates help capture short-term dependencies in time series.
- Update gates help capture long-term dependencies in time series.
- GRUs contain basic RNNs as their extreme case whenever the reset gate is switched on. They can ignore sequences when needed.

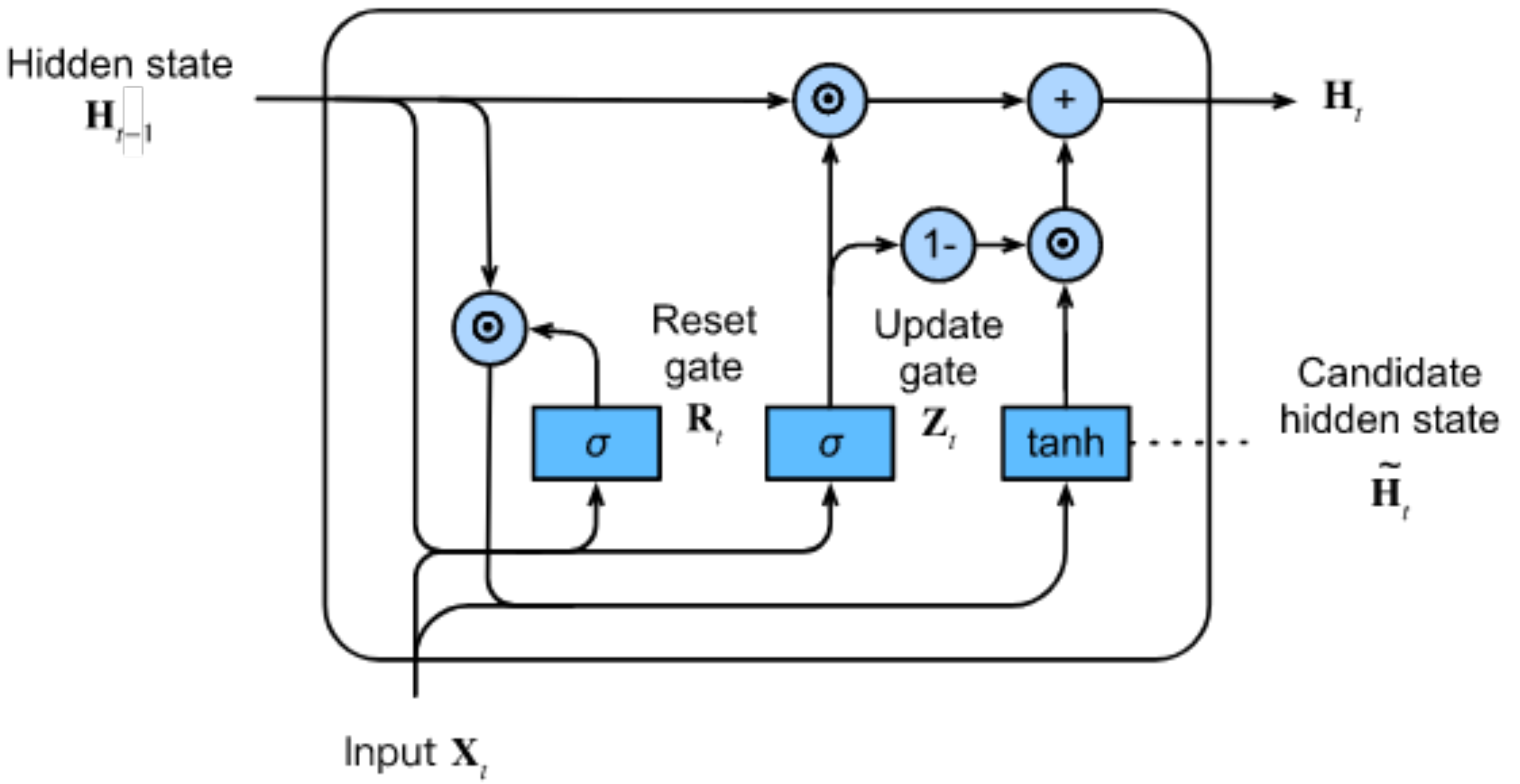


http://www.d2l.ai/chapter_recurrent-modern/gru.html

GRU vs LSTM:

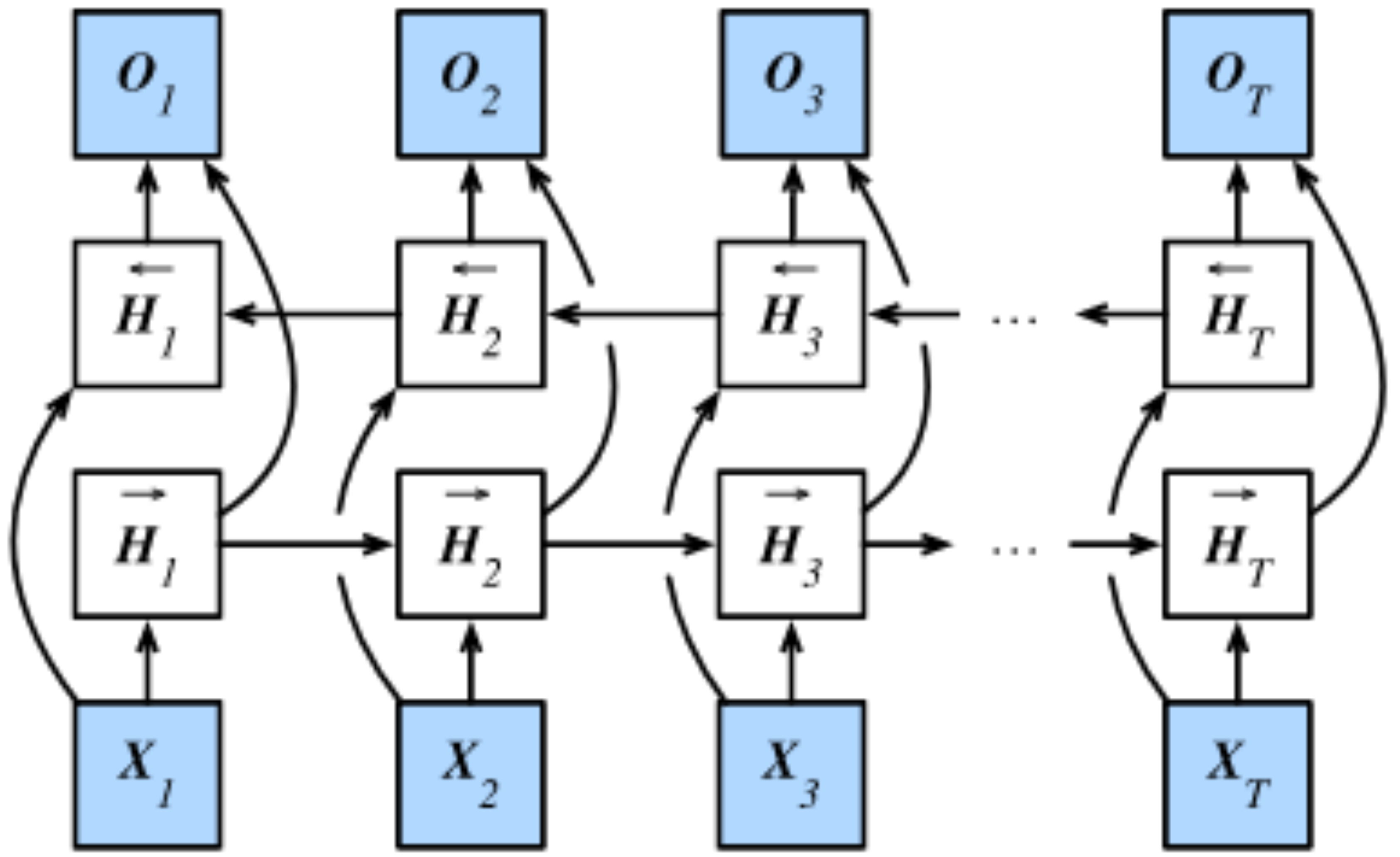
- There is earlier and more sophisticated structure LSTM
- Theoretically better for capturing long-term sequences, practically harder for computation.
- May beat exploding gradients better.





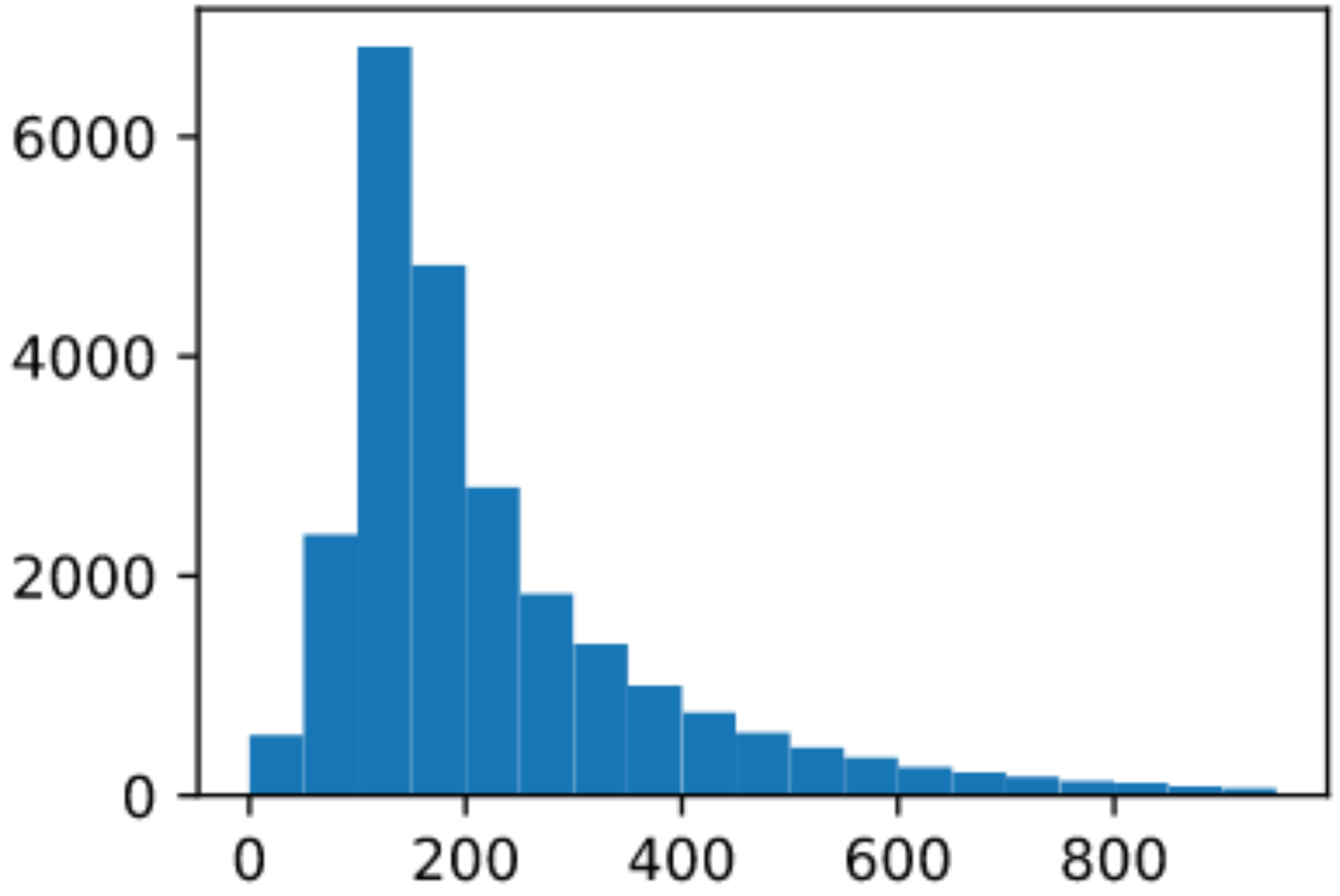
BiRNN

- Hidden state for each timestep is simultaneously determined by the data prior to and after the current timestep
- Mostly useful for sequence embedding and the estimation of observations given bidirectional context
- Very costly to train due to long gradient chains.



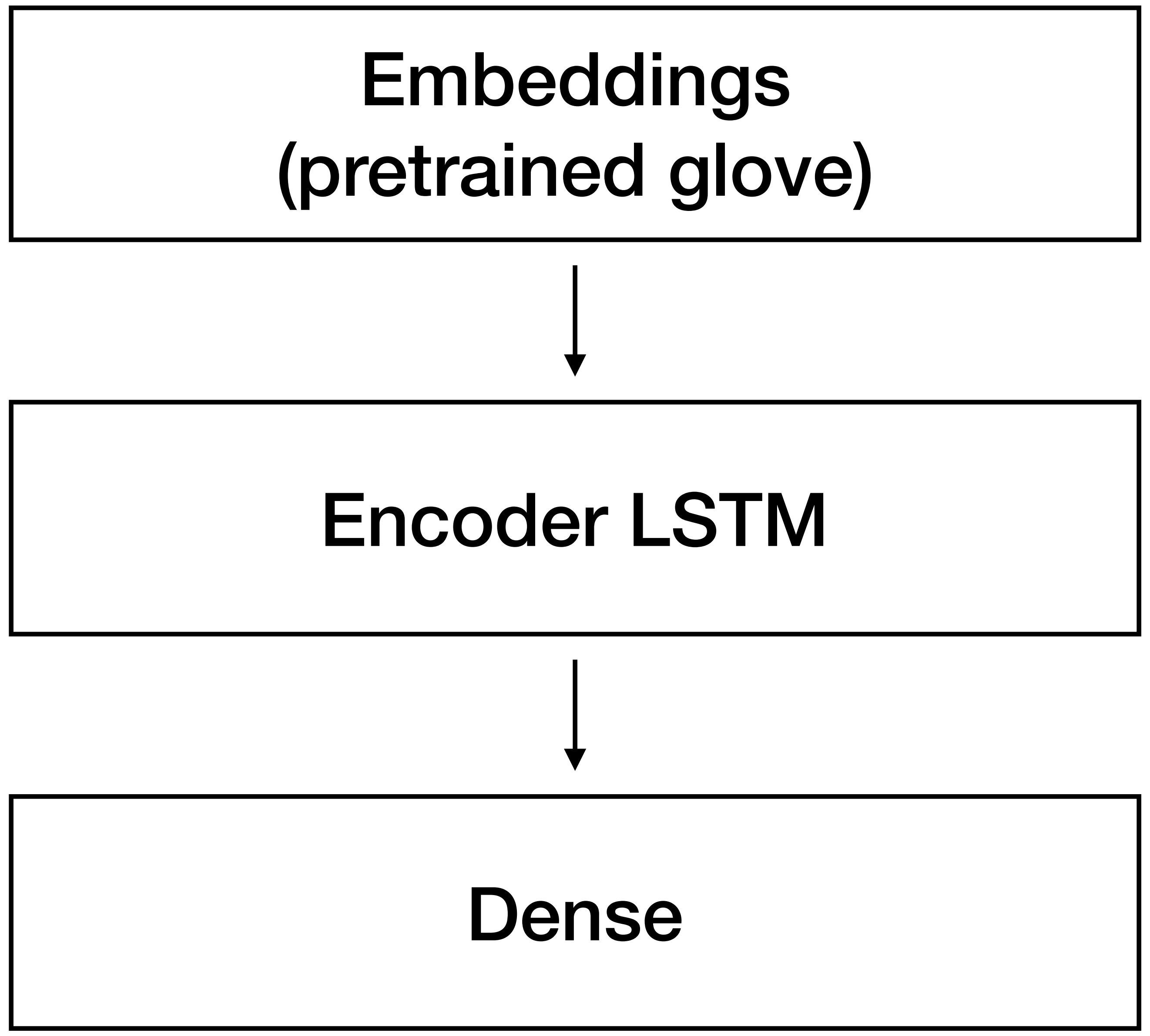
Sentiment analysis dataset:

- Movie reviews dataset
  https://ai.stanford.edu/~amaas/data/sentiment/ . 25k «positive» and «negative» reviews.
- Preparation steps:
  - Split to words
  - Vectorization (min_df = 5)
  - Padding to the same length (500)



Text original length histogram

Sentiment using RNN:

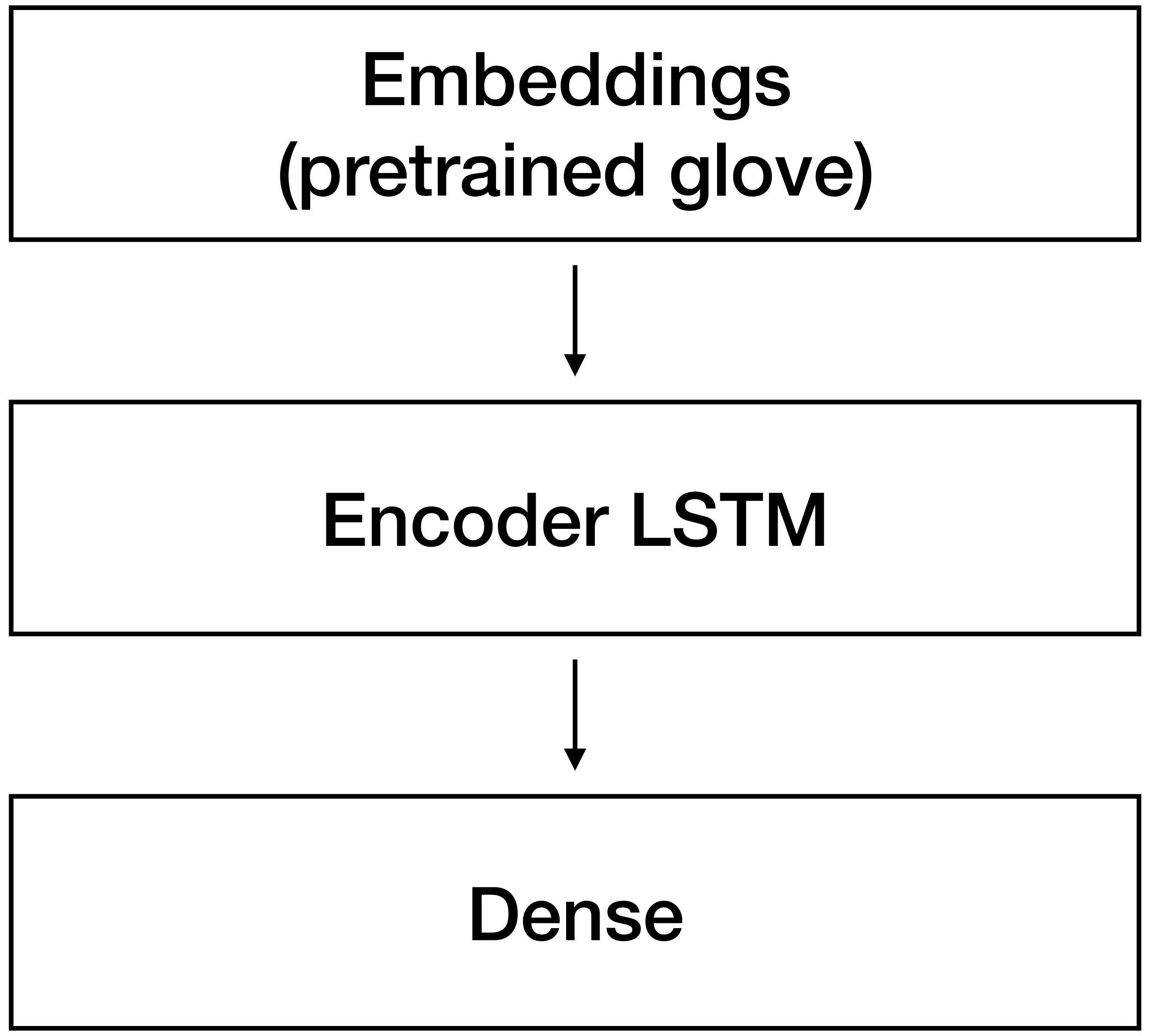


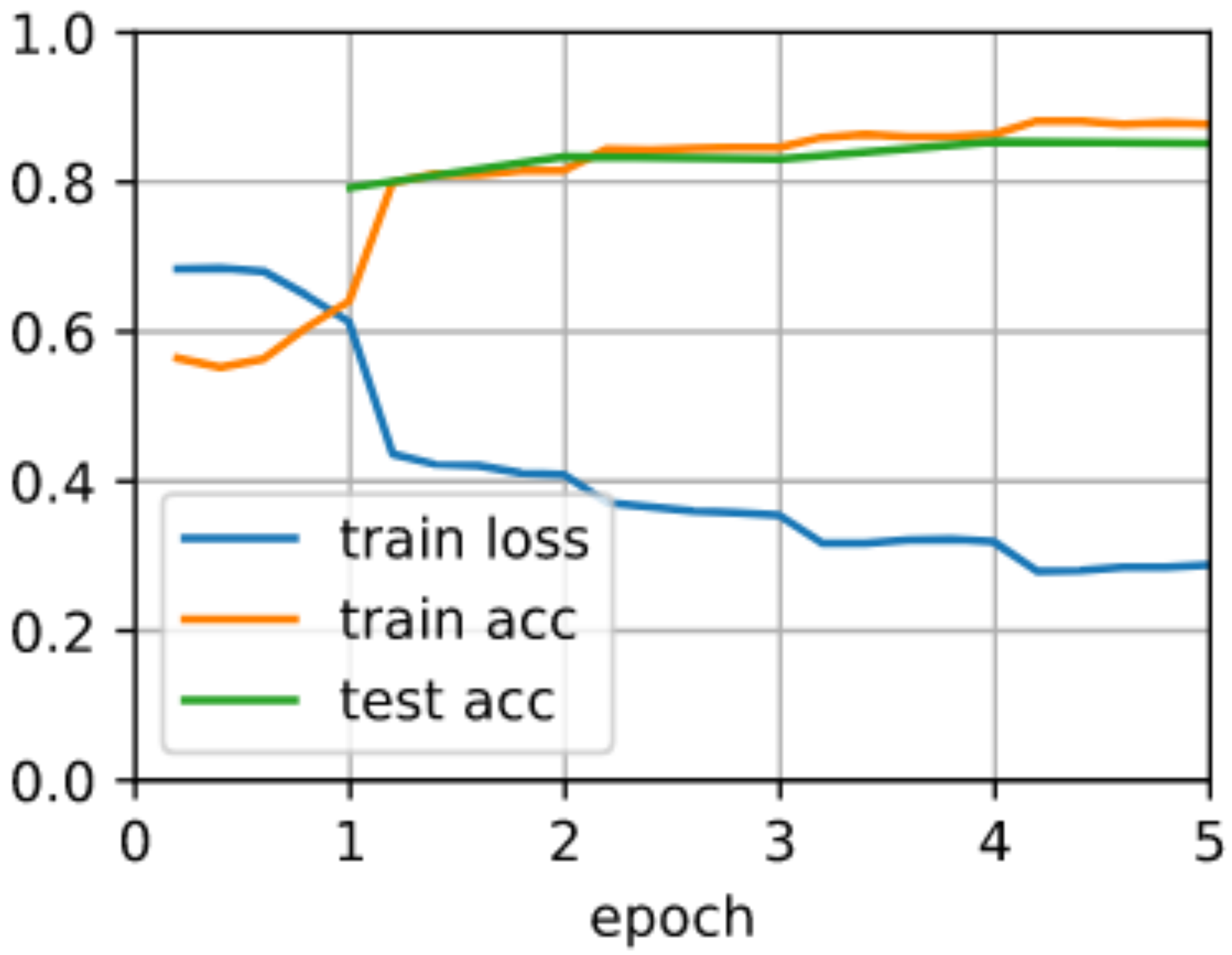

```python
class BiRNN(nn.Block):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set Bidirectional to True to get a bidirectional recurrent neural
        # network
        self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,
                                bidirectional=True, input_size=embed_size)
        self.decoder = nn.Dense(2)

    def forward(self, inputs):
        # The shape of inputs is (batch size, number of words). Because LSTM
        # needs to use sequence as the first dimension, the input is
        # transformed and the word feature is then extracted. The output sha
        # is (number of words, batch size, word vector dimension).
        embeddings = self.embedding(inputs.T)
        # Since the input (embeddings) is the only argument passed into
        # rnn.LSTM, it only returns the hidden states of the last hidden lay
        # at different timestep (outputs). The shape of outputs is
        # (number of words, batch size, 2 * number of hidden units).
        outputs = self.encoder(embeddings)
        # Concatenate the hidden states of the initial timestep and final
        # timestep to use as the input of the fully connected layer. Its
        # shape is (batch size, 4 * number of hidden units)
        encoding = np.concatenate((outputs[0], outputs[-1]), axis=1)
        outs = self.decoder(encoding)
        return outs
```

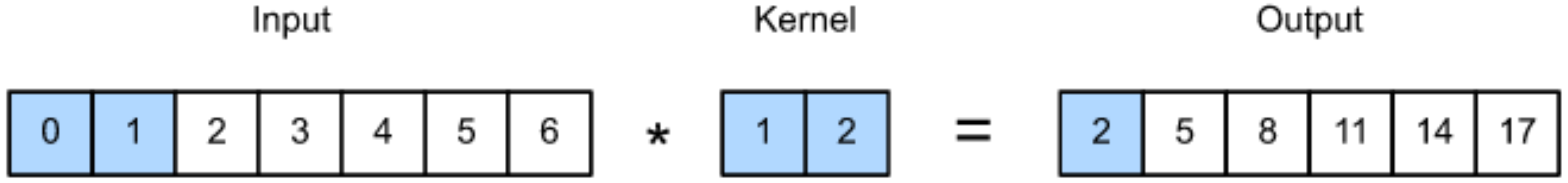
Sentiment using RNN:

- Text classification transforms a sequence of text of indefinite length into a category of text. This is a downstream application of word embedding.
- We can apply pre-trained word vectors and recurrent neural networks to classify the emotions in a text.

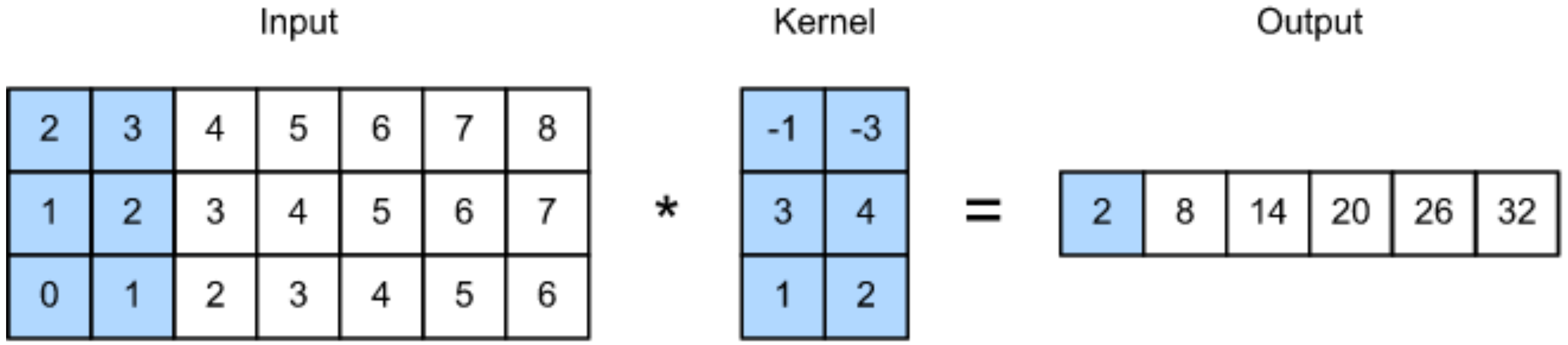




Sentiment using CNN:



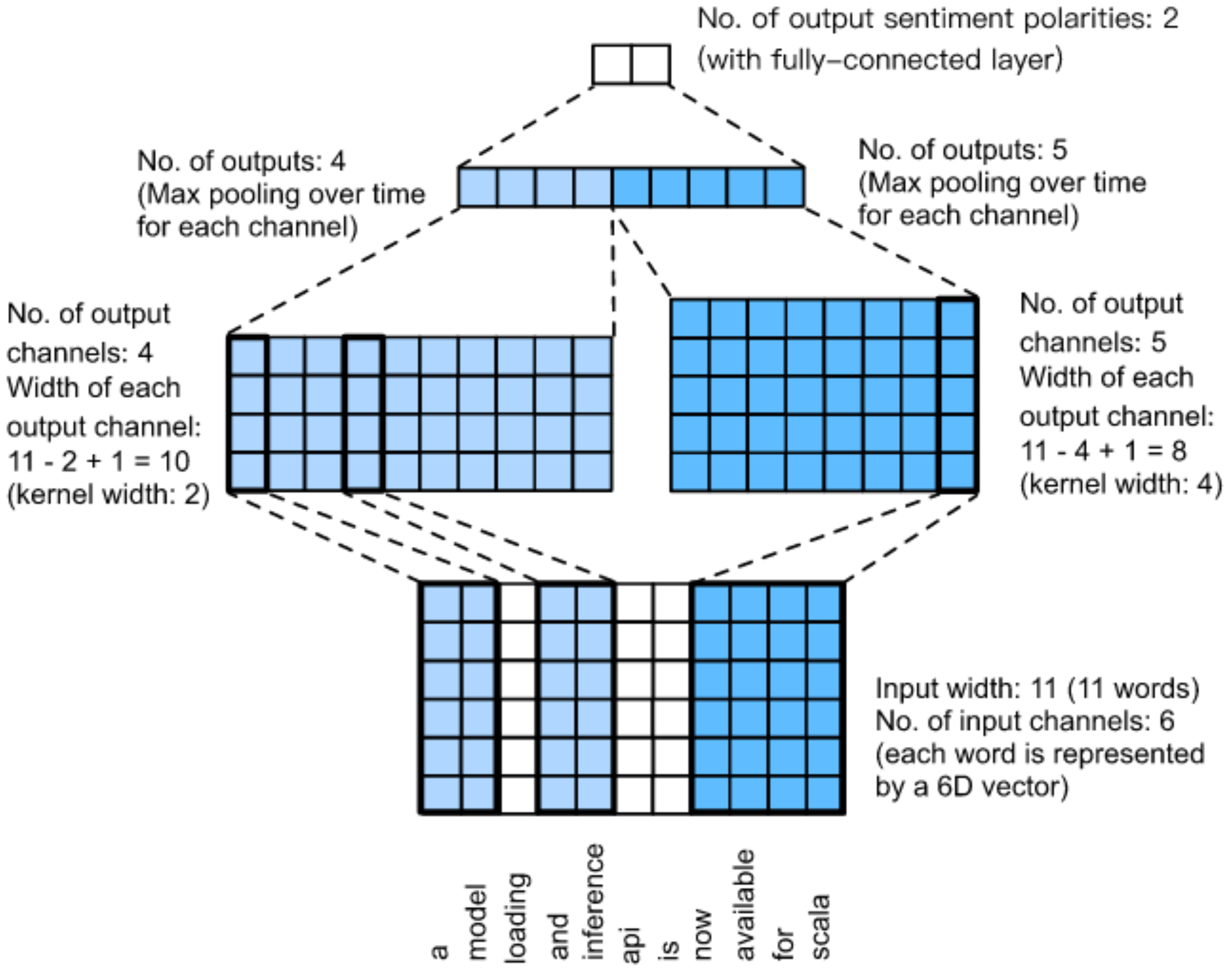
One-dimensional cross-correlation operation. The shaded parts are the first output element as well as the input and kernel array elements used in its calculation: 0×1+1×2=2



Two-dimensional cross-correlation operation with a single input channel. The highlighted parts are the first output element and the input and kernel array elements used in its calculation: 2×(−1)+3×(−3)+1×3+2×4+0×1+1×2=2

Text-CNN:

- Input: n words, and each word is represented by a d-dimension word vector
- Treated as: example has a width of n, a height of 1, and d input channels



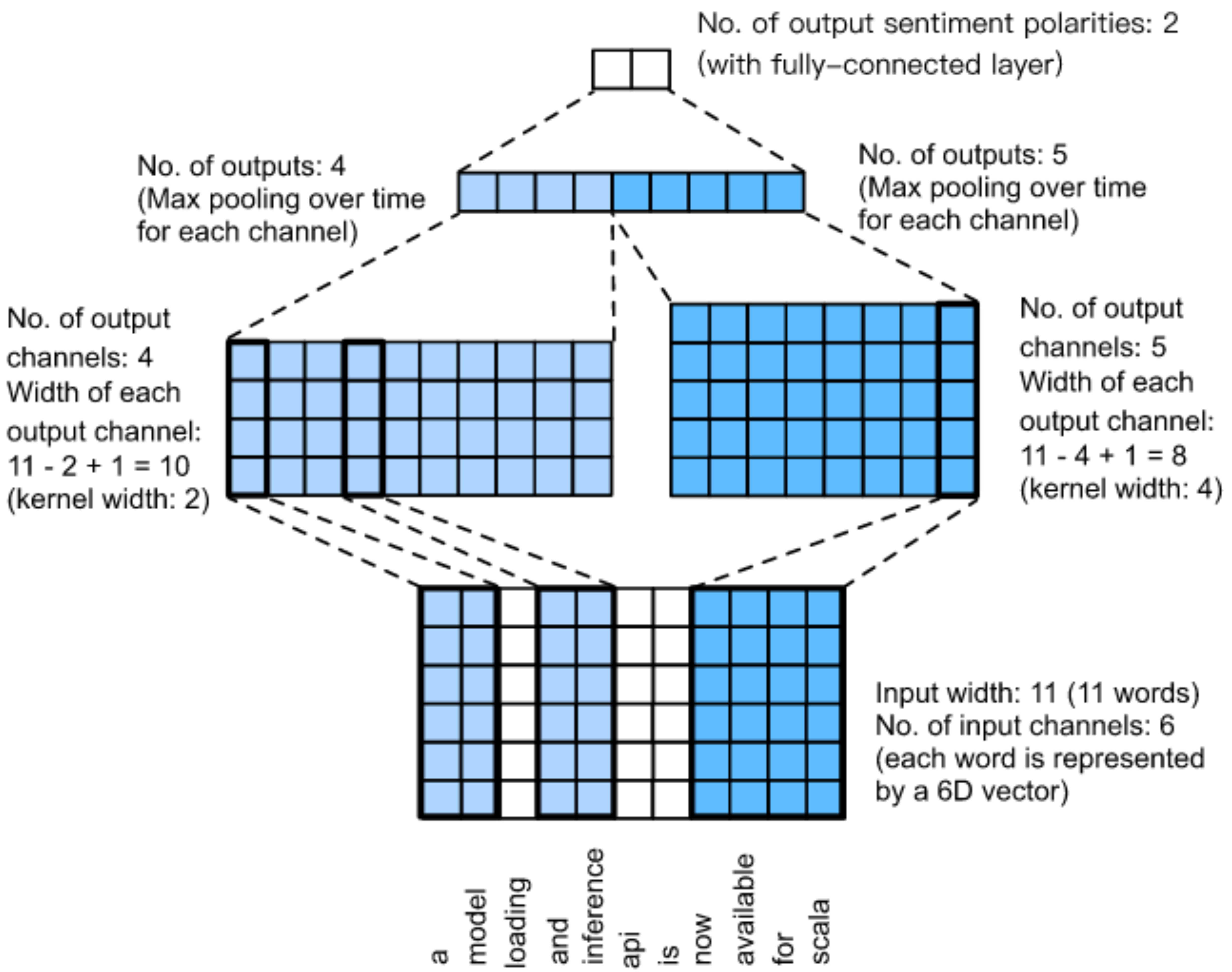
Text-CNN:

Computation:

1. Define multiple one-dimensional convolution kernels. Different width kernels - dependency of seq with different num of words.

2. Perform Max-Over-Time pooling.

3. Concatenate and transform through fully-connected layer.



Text-CNN:

Summary:

- We can use one-dimensional convolution to process and analyze timing data.
- A one-dimensional cross-correlation operation with multiple input channels can be regarded as a two-dimensional cross-correlation operation with a single input channel.
- The input of the max-over-time pooling layer can have different numbers of timesteps on each channel.
- TextCNN mainly uses a one-dimensional convolutional layer and max-over-time pooling layer.