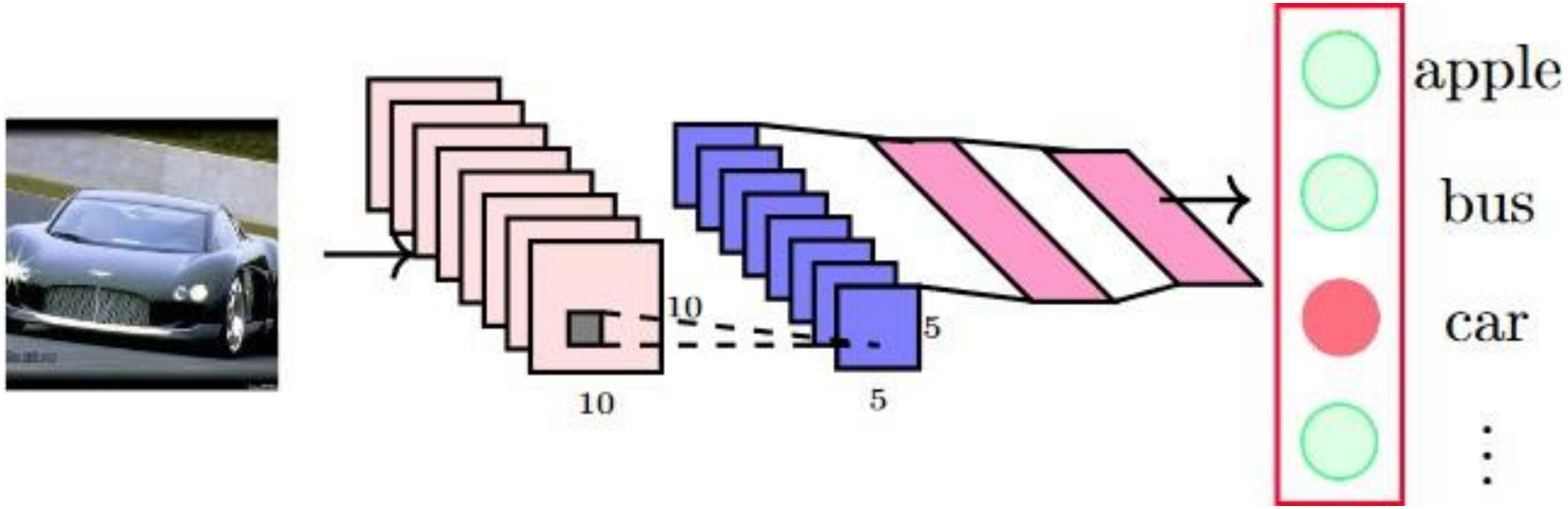
Sequence modeling – recurrent networks

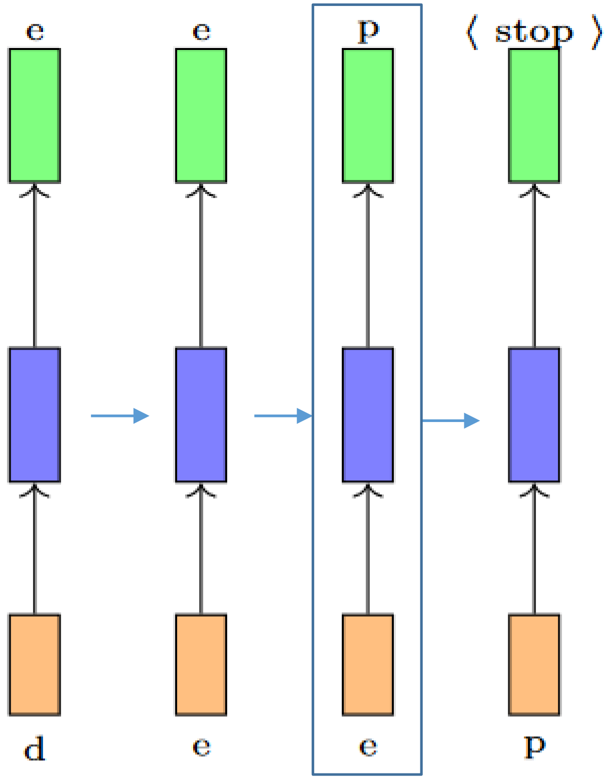
Biplab Banerjee

Sequential learning problem - motivation



- ✓ Each input is of fixed size
- ✓ Inputs are independent

Sequential learning problem



Auto-correct application in e-mail

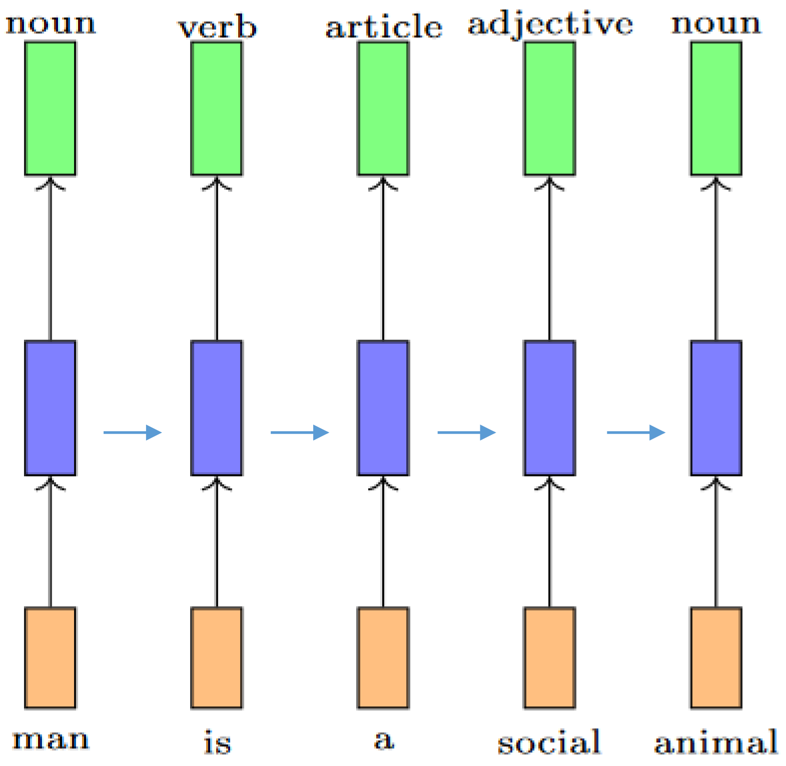
✓ Input lengths can vary

✓ Inputs are no longer independent

✓ At each time stamp, the same operation is Carried out

Given previous **information**
And current input, **predict**
Next input

Sequential learning problem



<u>Parts of speech tagging</u>

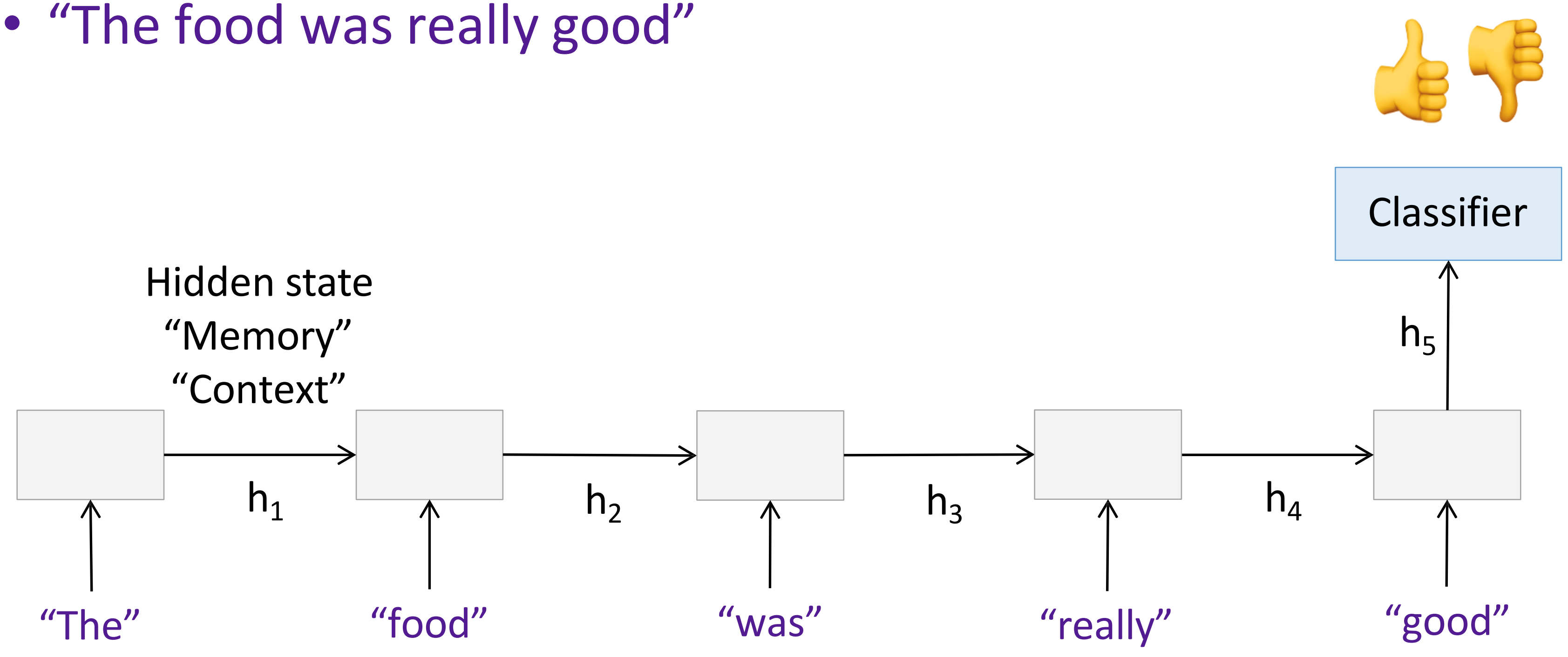
A set of previous time-stamps provide useful information
About the next time-stamp

Text classification

- **Sentiment classification:** classify a restaurant or movie or product review as positive or negative
  - "The food was really good"
  - "The vacuum cleaner broke within two weeks"
  - "The movie had slow parts, but overall was worth watching"
- What feature representation or predictor structure can we use for this problem?

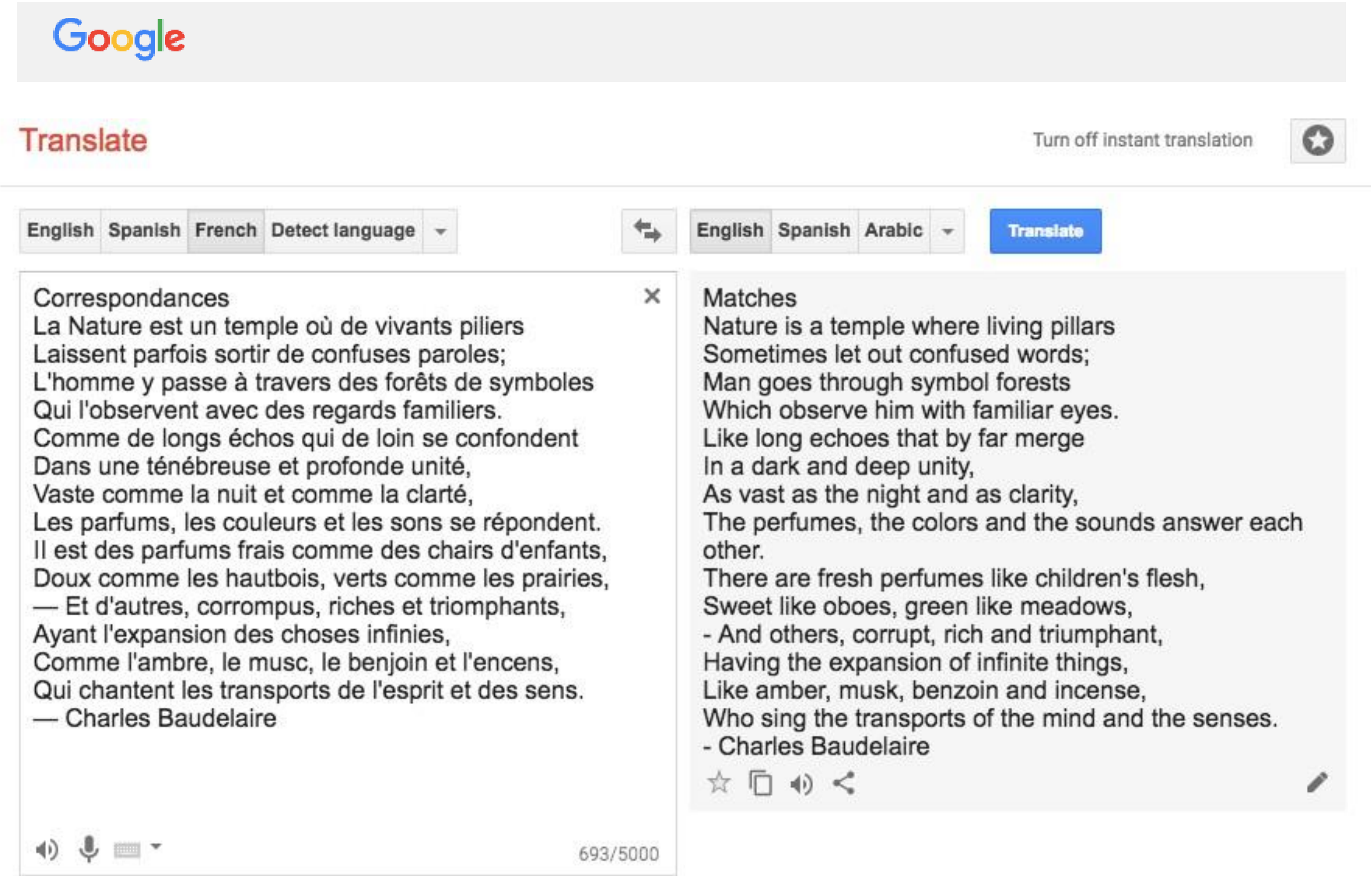
Sentiment classification

- “The food was really good”



Recurrent Neural Network (RNN)

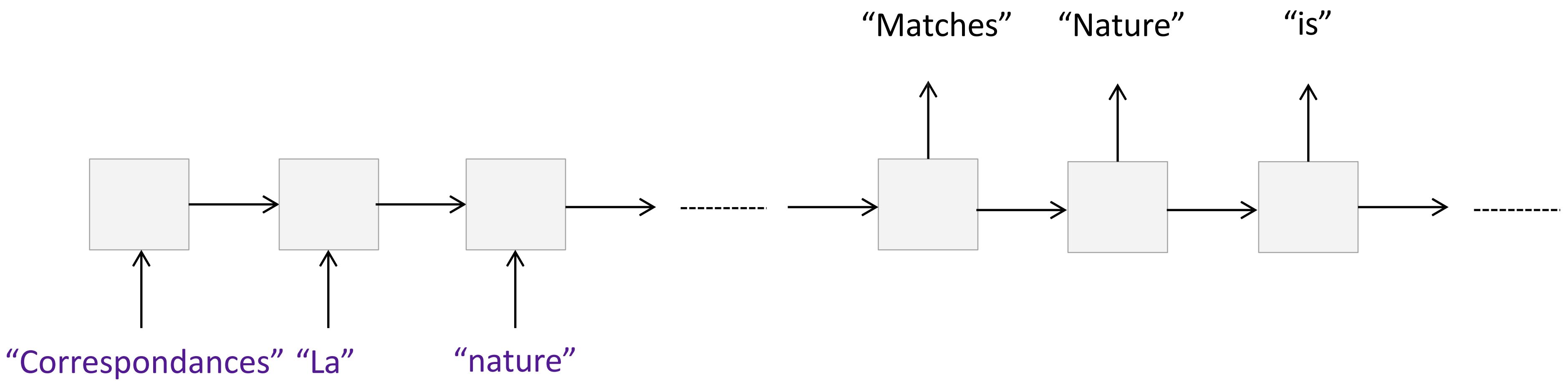
Machine translation



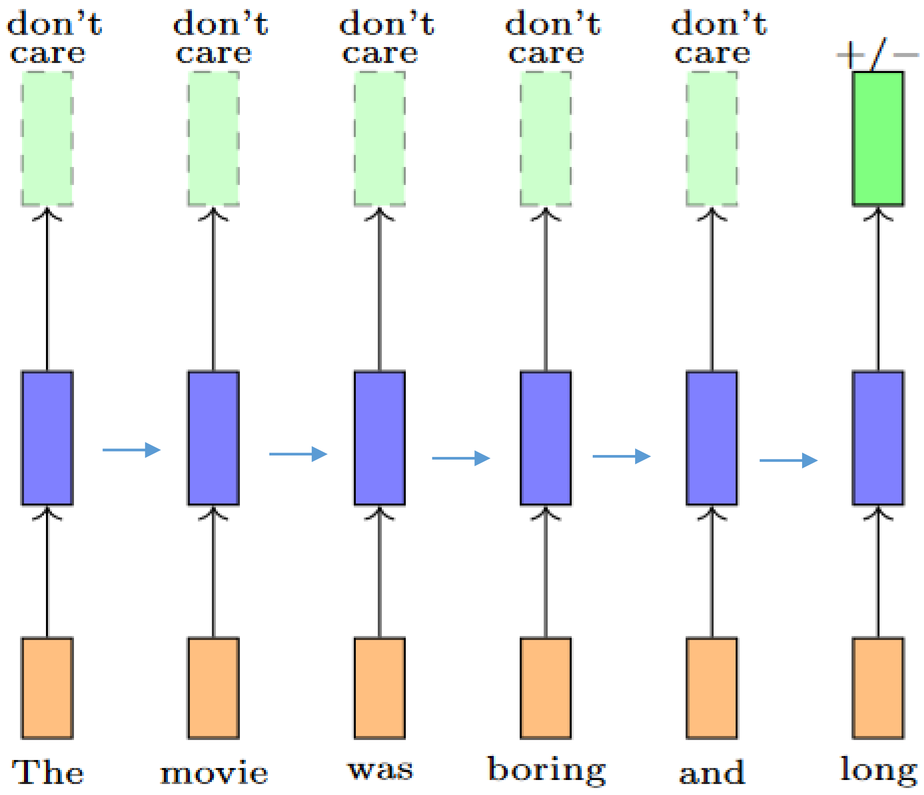
https://translate.google.com/

Machine translation

- Multiple input – multiple output (or sequence to sequence)

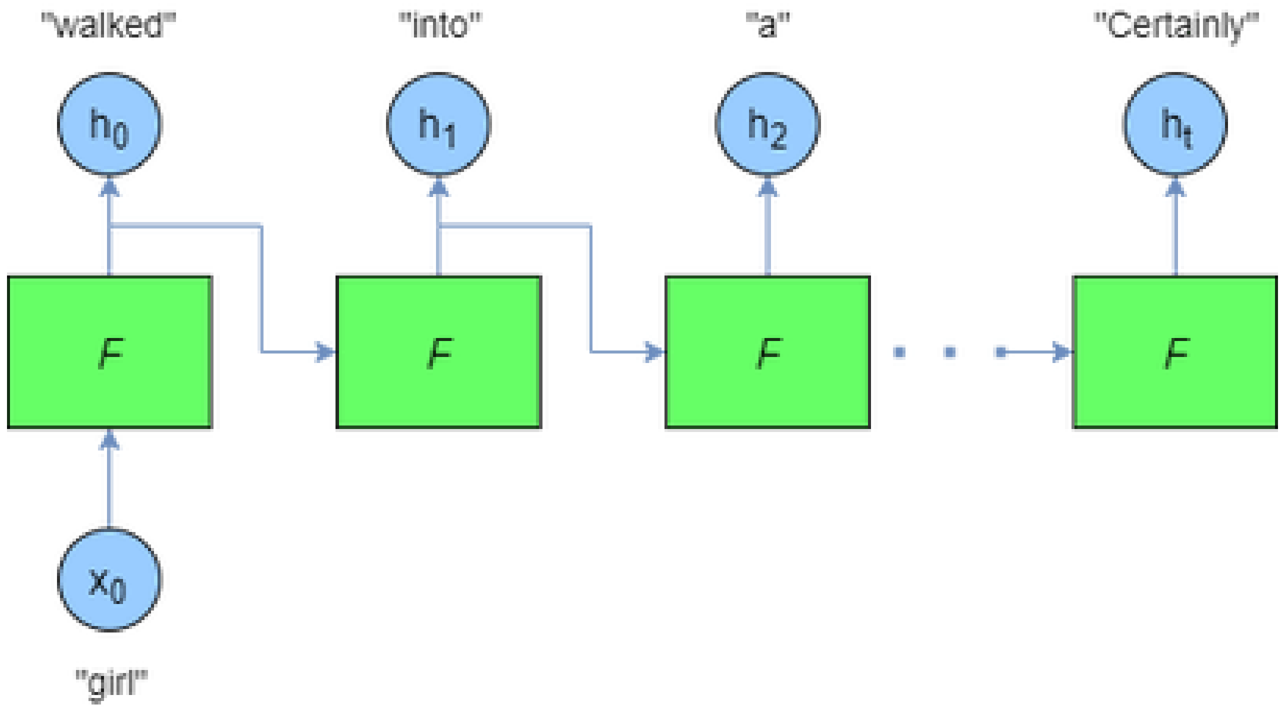


Different input output combinations



Movie review – many to one

Different input output combinations



Many to Many -

Machine translation

Dialogue

Different input output combinations

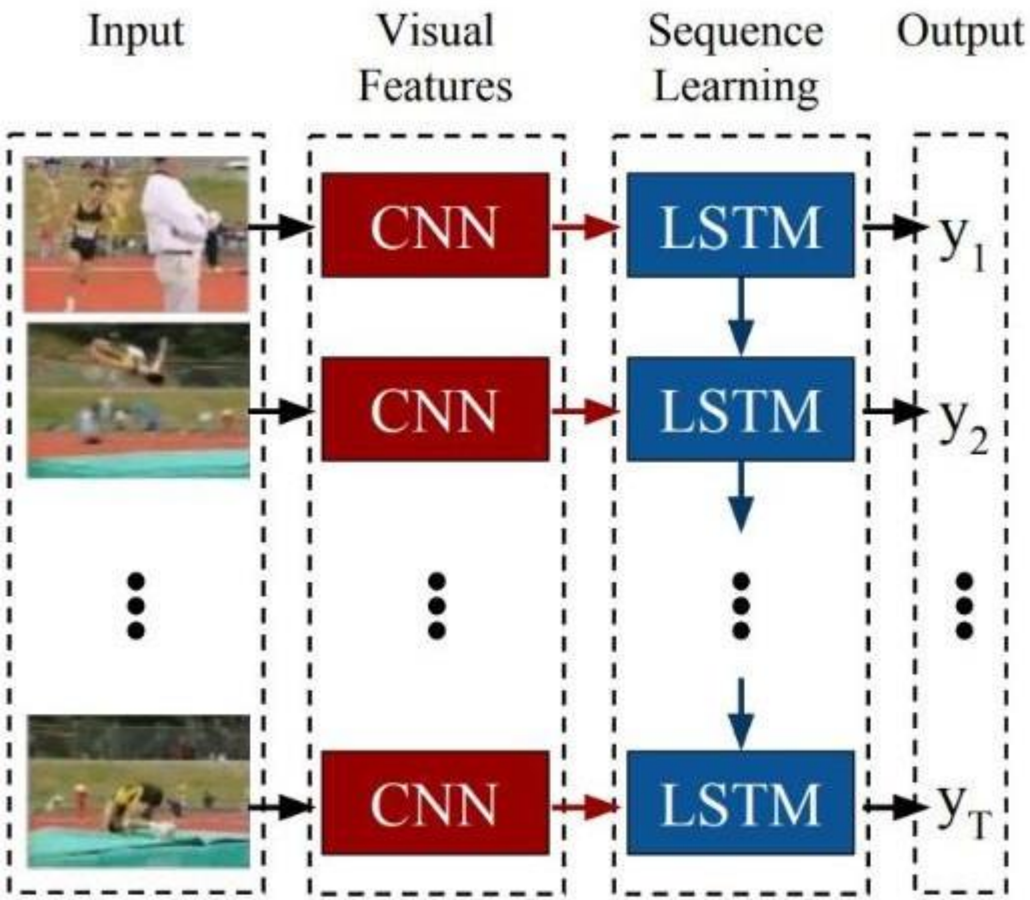
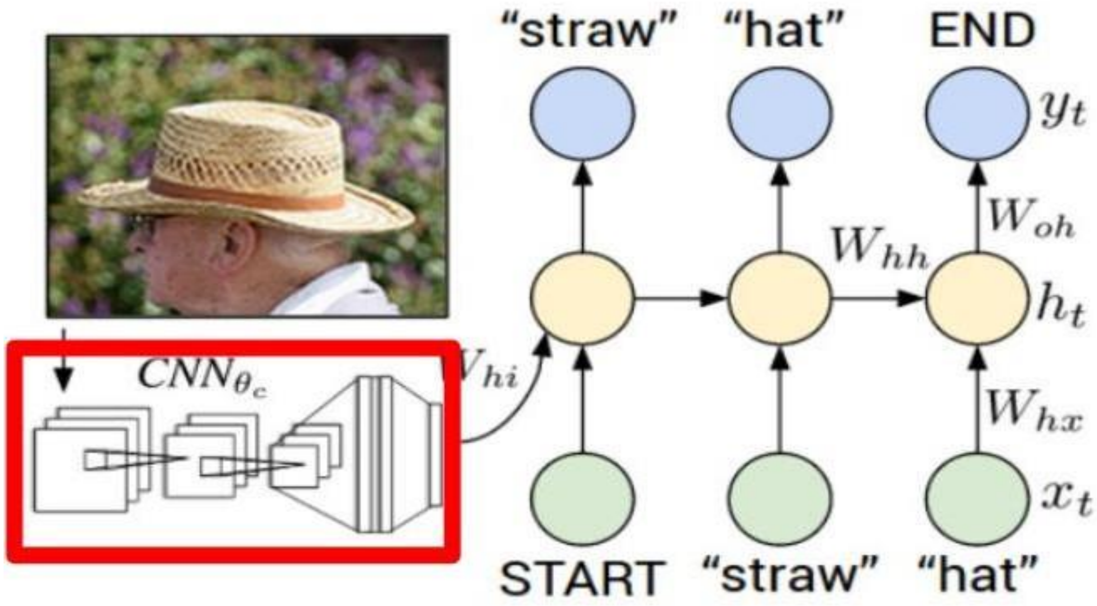
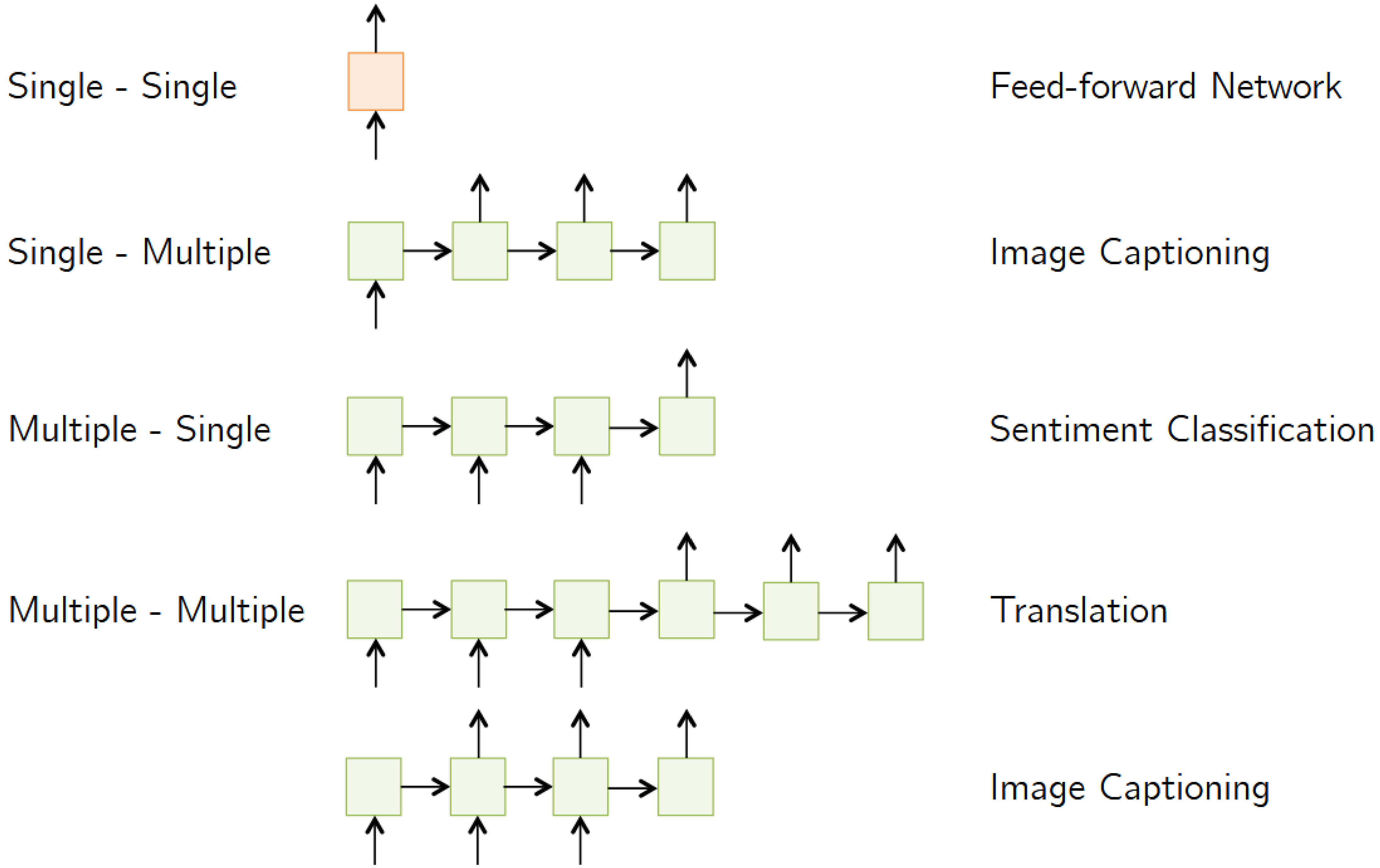


Image and video captioning



Many to many and one to many


Single - Single
Feed-forward Network
Single - Multiple
Image Captioning
Multiple - Single
Sentiment Classification
Multiple - Multiple
Translation
Image Captioning

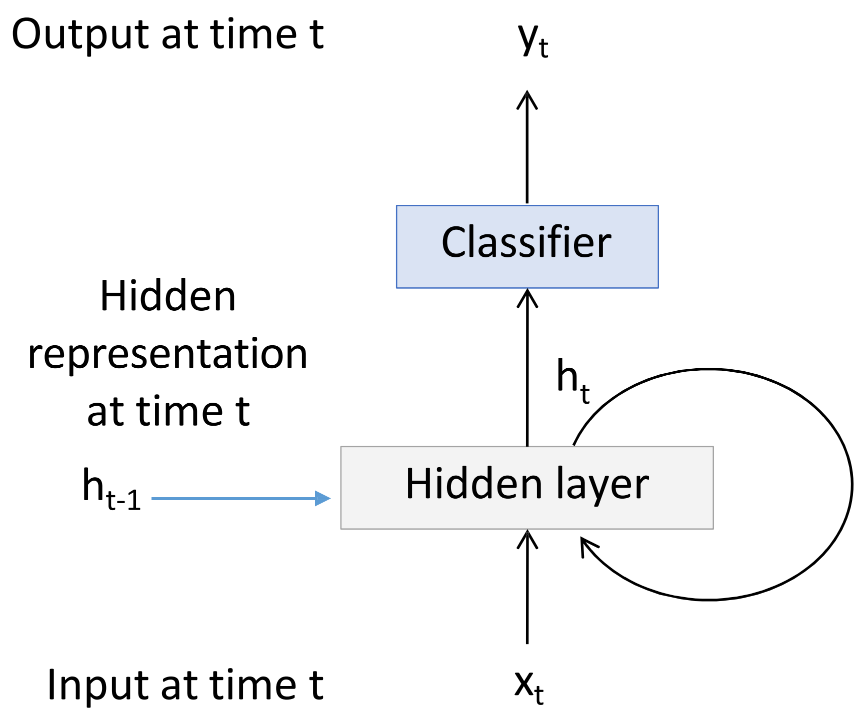
Sequential learning problem

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1})$$

The predicted word at a certain point can be inferred given a few previous words

Markovian Property of languages

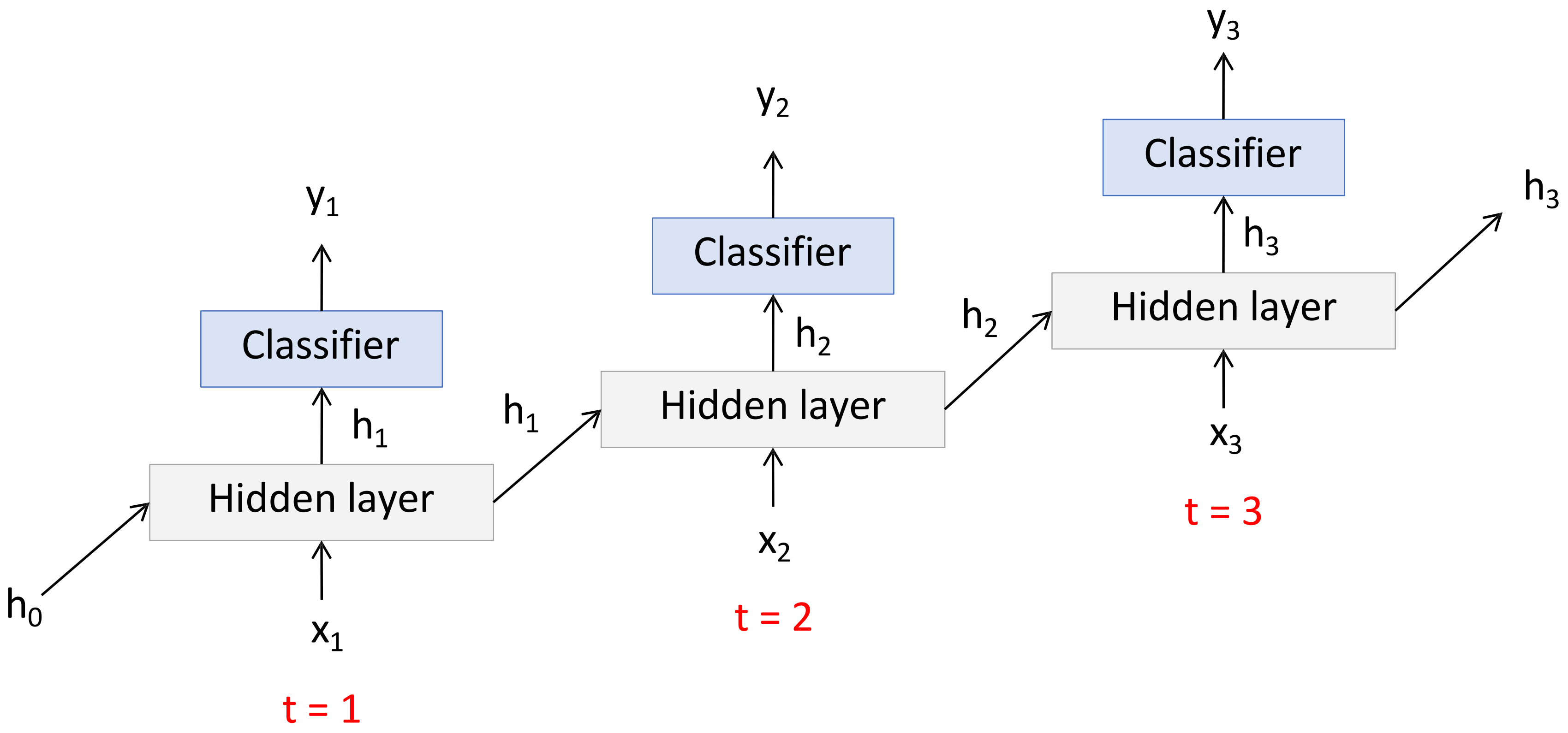
Idea of hidden state in RNN



Recurrence:

$$\underbrace{h_t}_{\text{new state}} = \underbrace{f_W}_{\text{function of W}}(\underbrace{x_t}_{\text{input at time t}}, \underbrace{h_{t-1}}_{\text{old state}})$$

Unrolling the RNN



Formulation

h(t) = σ (W h(t-1) + U x(t))

hat(y) (t) = softmax(V h(t))

$$\hat{P}(x_{t+1} = v_j \mid x_t, \ldots, x_1) \quad = \quad \hat{y}_{t,j}$$

Encodes the long-term dependency

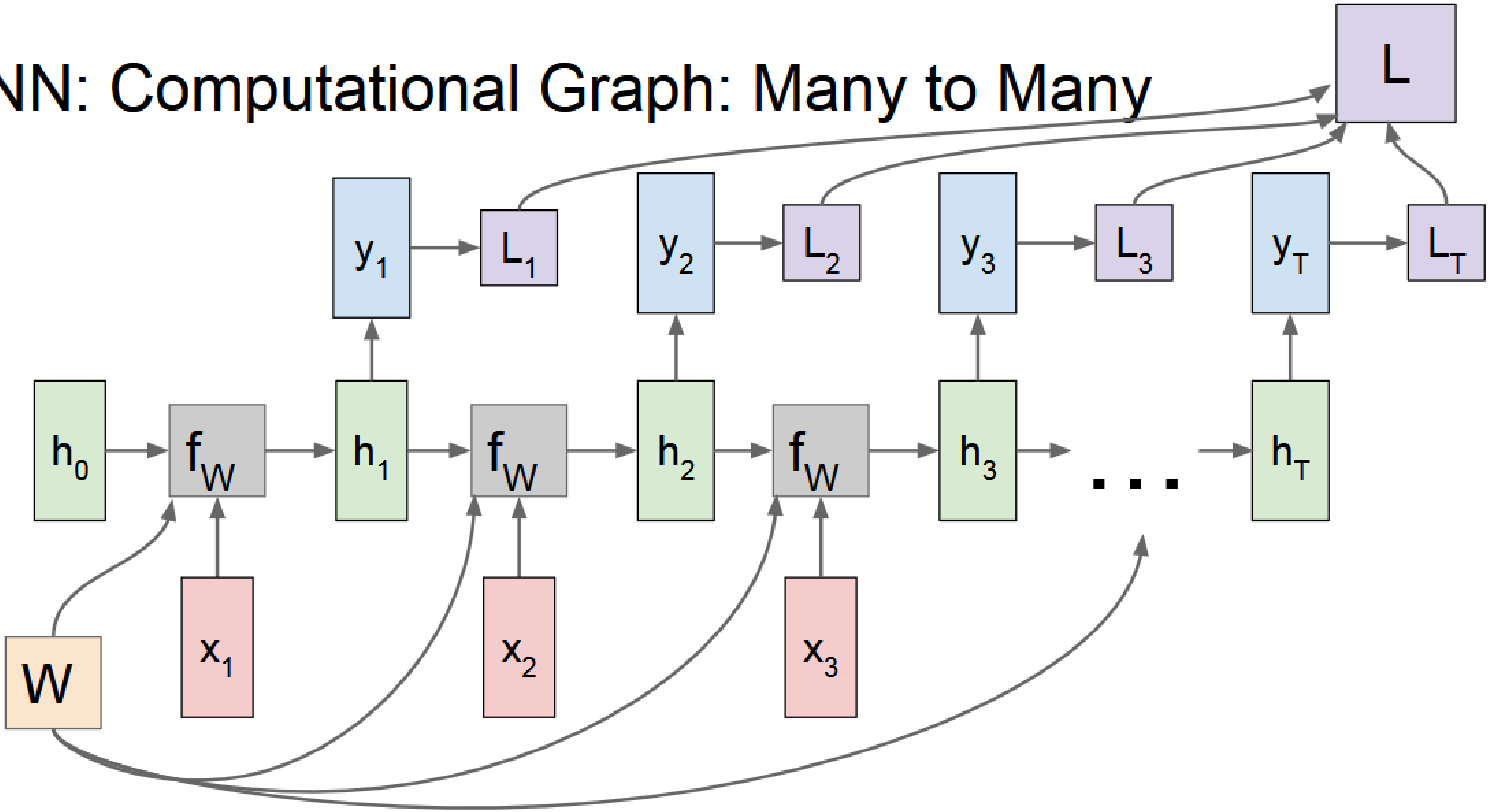
$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

The loss function to be minimized

- ✓ Model for Many to Many prediction
- ✓ One output per time stamp
- ✓ |V| denotes the number of words
- ✓ Training with softmax cross-entropy
- ✓ **The weights are shared for all the time-stamps**

Training is **Back-Propagation Through Time**

RNN: Computational Graph: Many to Many



How to define the classifier

- There could be multiple possibilities for defining the classifier:
  - Ensemble of classifiers for all the states?
  - Single classifier at the final state?

- It is application dependent. Let us see two examples for sentiment classification and captioning

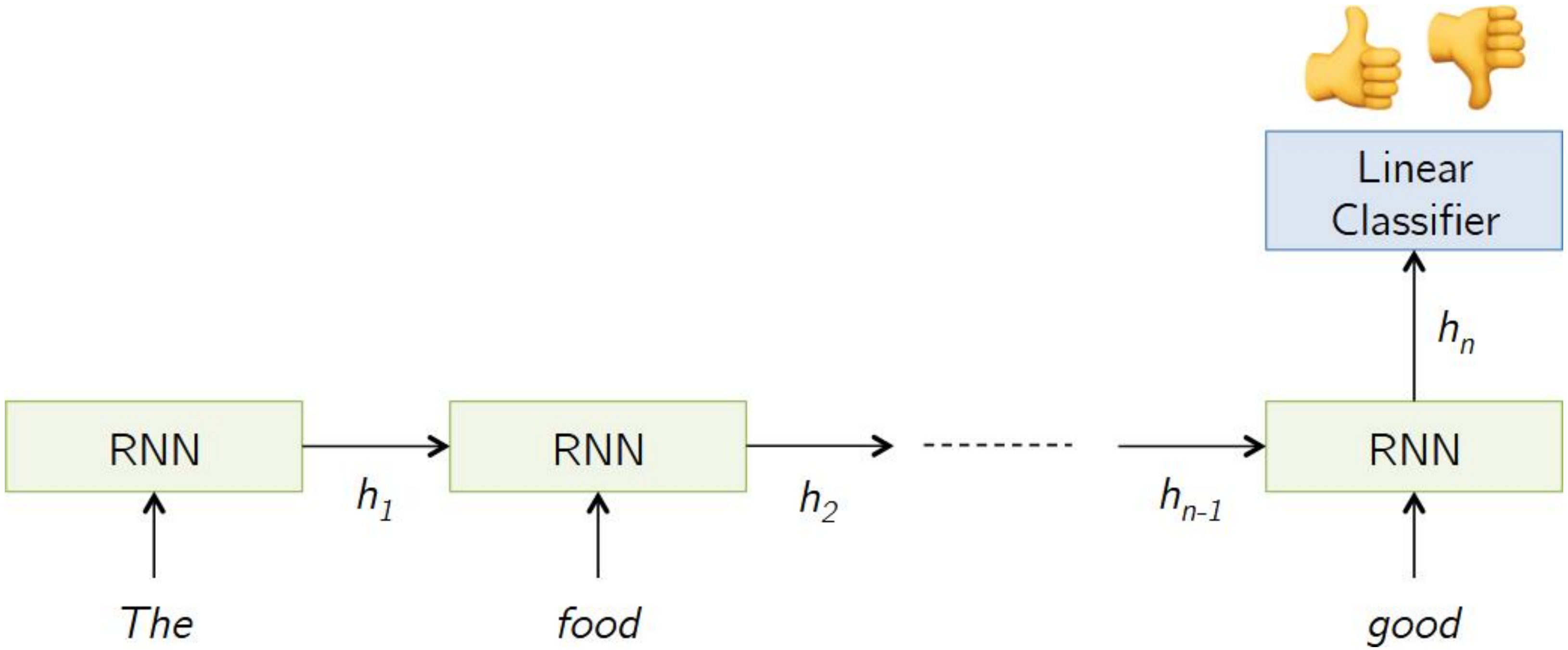
Example – sentiment classification

- Classify a
  restaurant review from Yelp! OR
  movie review from IMDB OR
  ...
  as positive or negative

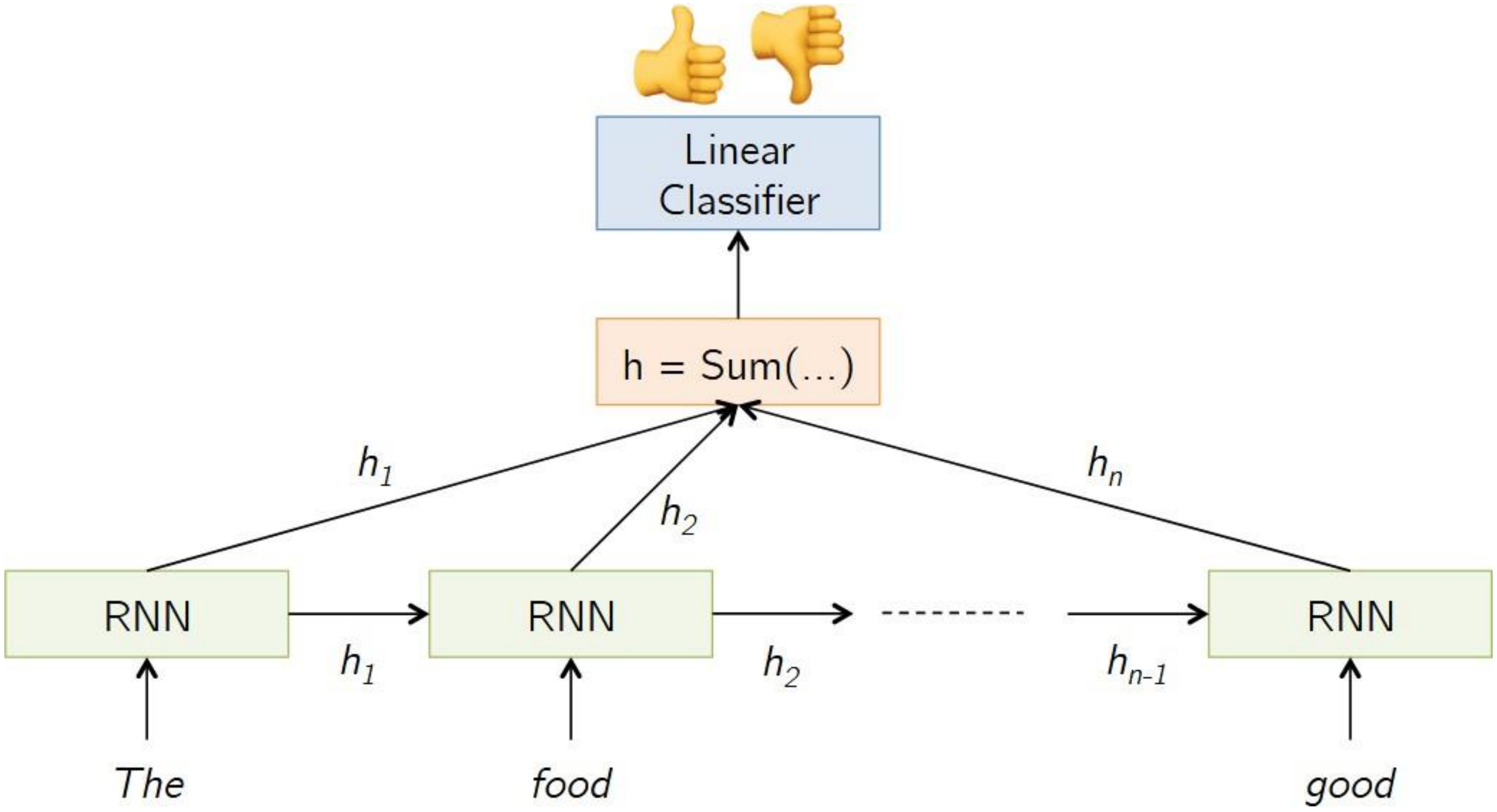
- **Inputs:** Multiple words, one or more sentences
- **Outputs:** Positive / Negative classification

- "The food was really good"
- "The chicken crossed the road because it was uncooked"

Model-1



Model-2



Linear Classifier
h = Sum(...)
h_1
h_2
h_n
RNN
RNN
RNN
h_1
h_2
h_{n-1}
The
food
good

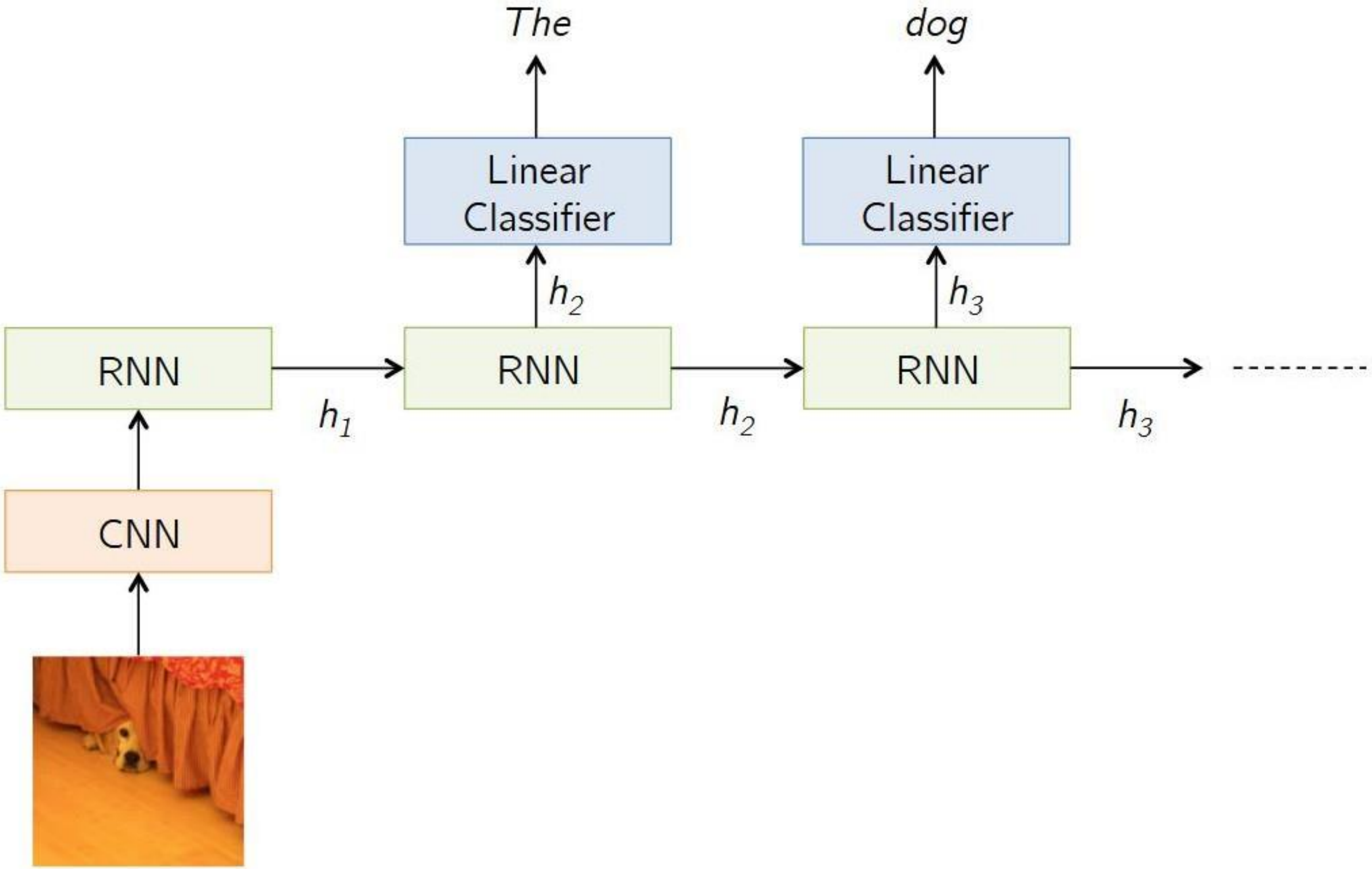
Image captioning

- Given an image, produce a sentence describing its contents

- **Inputs:** Image feature (from a CNN)
- **Outputs:** Multiple words (let's consider one sentence)

: The dog is hiding

Model



The
dog
Linear Classifier
Linear Classifier
h_2
h_3
RNN
RNN
RNN
h_1
h_2
h_3
CNN

A person riding a motorcycle on a dirt road.

Two dogs play in the grass.

A herd of elephants walking across a dry grass field.

A group of young people playing a game of frisbee.

Two hockey players are fighting over the puck.

A close up of a cat laying on a couch.

[Show and Tell: A Neural Image Caption Generator, CVPR 15]

Some highlights

- RNN takes the previous output or hidden state as inputs at every time together with the usual input. This means that the composite input at time t has some historical information about the happenings at time T < t.

- RNNs gained popularity in sequence modeling without forgetting important past information as their intermediate values (state) can store information about the past inputs for a time that is not fixed a priori.

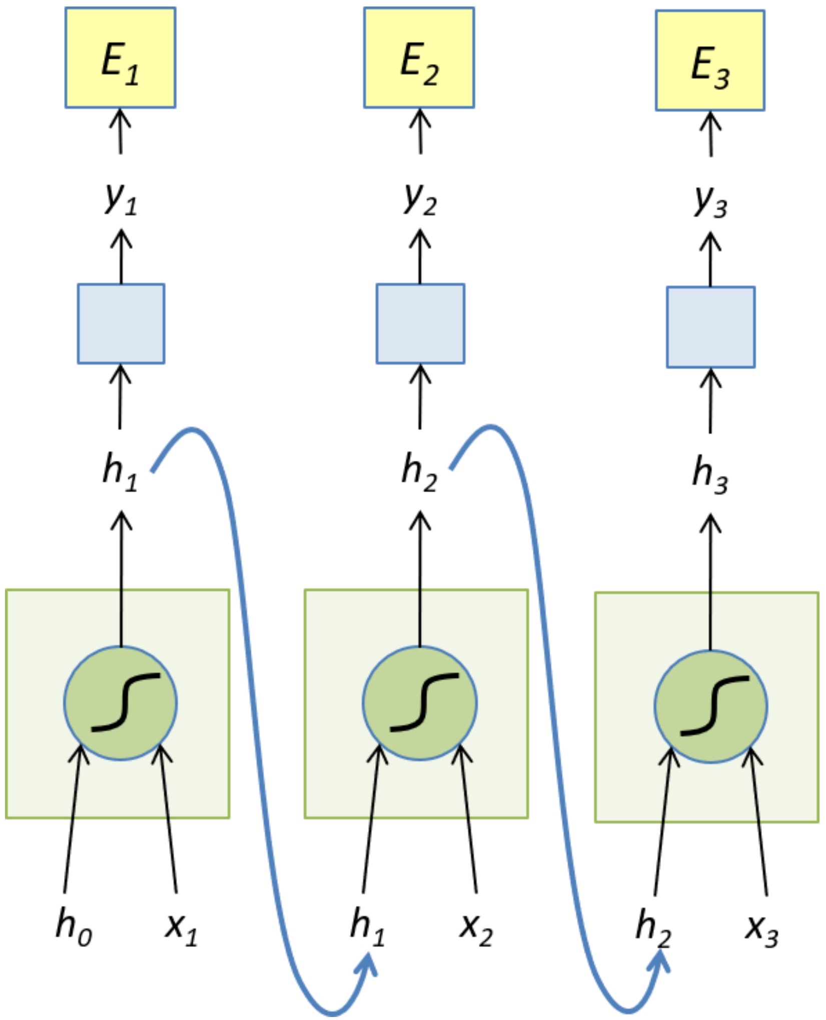
Back propagation through time

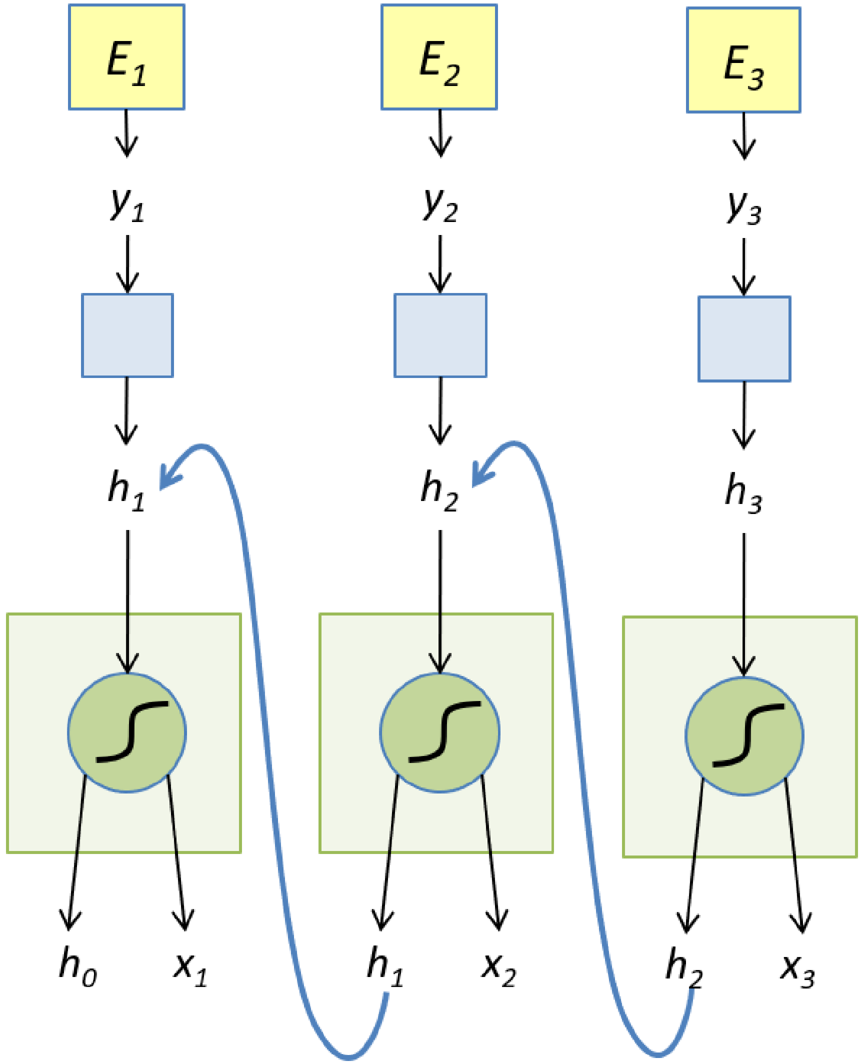
- Considered to be the standard method used to train RNNs.

- The unfolded network (used during forward pass) is treated as one feed-forward network that accepts the whole time series as input.

- The weight updates are computed for each copy in the unfolded network, then accumulated and applied to the model weights and biases.

Forward and backward pass





$$
\begin{aligned}
h_t &= f(x_t, h_{t-1}, w_\mathrm{h}),\\
o_t &= g(h_t, w_\mathrm{o}),
\end{aligned}
$$

where f and g are transformations of the hidden layer and the output layer, respectively. Hence, we have a chain of values $\{\ldots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \ldots\}$ that depend on each other via recurrent computation. The forward propagation is fairly straightforward. All we need is to loop through the (x_t, h_t, o_t) triples one time step at a time. The discrepancy between output o_t and the desired target y_t is then evaluated by an objective function across all the T time steps as

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_\mathrm{h}, w_\mathrm{o}) = \frac{1}{T}\sum_{t=1}^T l(y_t, o_t).$$

$$
\begin{aligned}
\frac{\partial L}{\partial w_\mathrm{h}} &= \frac{1}{T}\sum_{t=1}^T \frac{\partial l(y_t, o_t)}{\partial w_\mathrm{h}}\\
&= \frac{1}{T}\sum_{t=1}^T \frac{\partial l(y_t, o_t)}{\partial o_t} \frac{\partial g(h_t, w_\mathrm{o})}{\partial h_t} \frac{\partial h_t}{\partial w_\mathrm{h}}.
\end{aligned}
$$

$$\frac{\partial h_t}{\partial w_\mathrm{h}} = \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial w_\mathrm{h}} + \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_\mathrm{h}}.$$

To derive the above gradient, assume that we have three sequences $\{a_t\}, \{b_t\}, \{c_t\}$ satisfying $a_0 = 0$ and $a_t = b_t + c_t a_{t-1}$ for $t = 1, 2, \ldots$. Then for $t \geq 1$, it is easy to show

$$a_t = b_t + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} c_j \right) b_i.$$

By substituting a_t, b_t, and c_t according to

$$\begin{aligned} a_t &= \frac{\partial h_t}{\partial w_\mathrm{h}}, \\ b_t &= \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial w_\mathrm{h}}, \\ c_t &= \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial h_{t-1}}, \end{aligned}$$

$$\frac{\partial h_t}{\partial w_\mathrm{h}} = \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial w_\mathrm{h}} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial f(x_j, h_{j-1}, w_\mathrm{h})}{\partial h_{j-1}} \right) \frac{\partial f(x_i, h_{i-1}, w_\mathrm{h})}{\partial w_\mathrm{h}}.$$

The repeated multiplication of the gradients cause the derivative to explode

The exploding gradient problem

Imagine an RNN that just does:

$$
\begin{aligned}
h_1 &= w \cdot h_0, \\
h_2 &= w \cdot h_1, \\
h_3 &= w \cdot h_2, \\
&\cdots
\end{aligned}
$$

where:

- h_t is the hidden state at timestep t (just a single number here).
- w is a single learned parameter (instead of a matrix).
- h_0 is some initial state.

We define a **loss** L that depends on the final hidden state (say, h_3 in this tiny example). For simplicity, let's say L is just $L(h_3)$. When we do Backpropagation Through Time, we need $\frac{\partial L}{\partial w}$.

$$
\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h_3} \times \frac{\partial h_3}{\partial w}.
$$

1. $h_3 = w \cdot h_2$

$$\frac{\partial h_3}{\partial w} = h_2 \; + \; w \, \frac{\partial h_2}{\partial w} \quad \text{(Product Rule)}$$

2. $h_2 = w \cdot h_1$

$$\frac{\partial h_2}{\partial w} = h_1 \; + \; w \, \frac{\partial h_1}{\partial w}.$$

3. $h_1 = w \cdot h_0$

$$\frac{\partial h_1}{\partial w} = h_0 \quad \text{(no extra term, since } h_0 \text{ is constant).}$$

Putting these together, we go from the inside out:

$$\frac{\partial h_2}{\partial w} = h_1 + w \underbrace{\frac{\partial h_1}{\partial w}}_{=h_0} = w \cdot h_0 + w \cdot h_0 = 2\, w\, h_0.$$

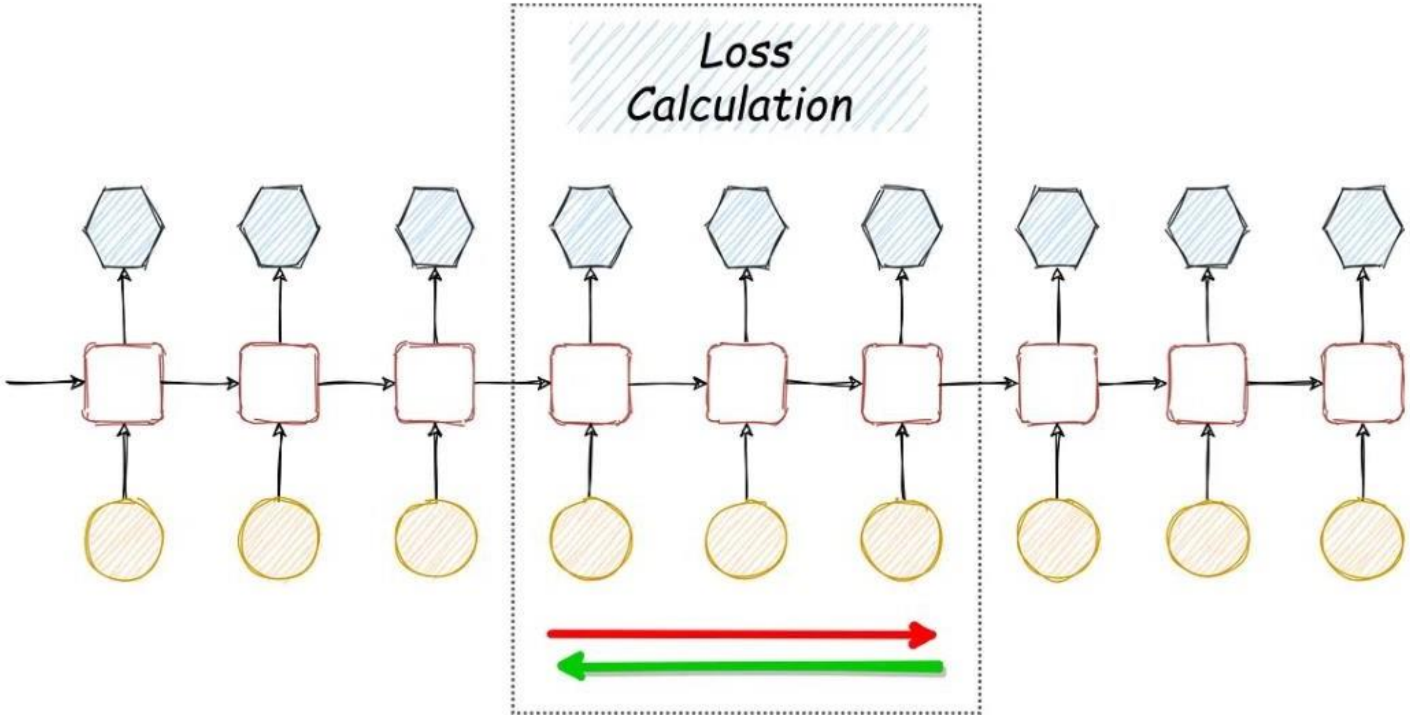
Then,

$$\frac{\partial h_3}{\partial w} = h_2 + w \, \frac{\partial h_2}{\partial w} = \underbrace{w \cdot h_1}_{=w^2 h_0} \; + \; w \, \bigl(2\, w\, h_0\bigr) = w^2 h_0 + 2\, w^2 h_0 = 3\, w^2\, h_0.$$

$$\frac{\partial L}{\partial w} = \delta_3 \times \bigl(3\, w^2\, h_0\bigr).$$

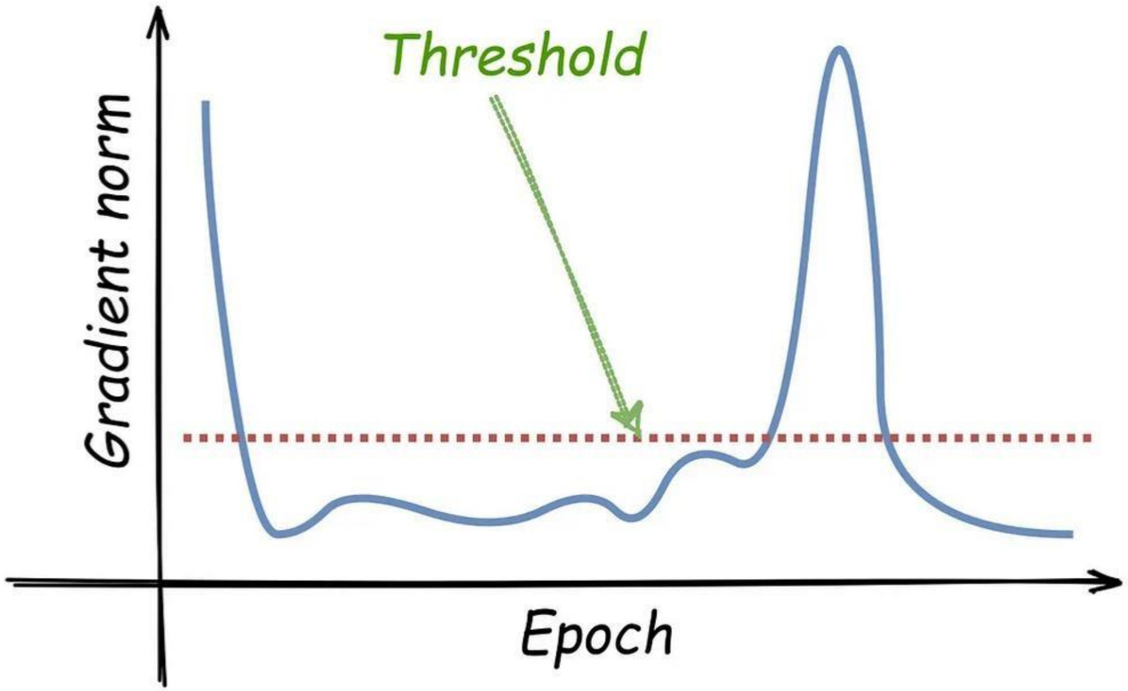
For **3 timesteps**, we got a factor of **3** and a factor of w^2. If we had T **timesteps**, you'd see terms that grow roughly like $T \cdot w^{T-1}$. If $|w| > 1$, this can become **very large** for even moderate T.

Solutions



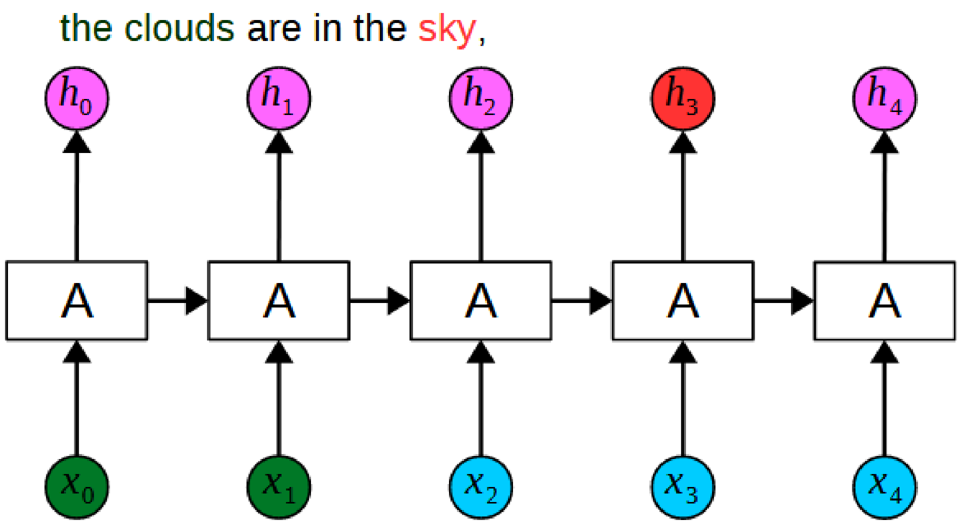
Instead of processing the entire sequence, process the forward/backward passes in chunks

Solutions



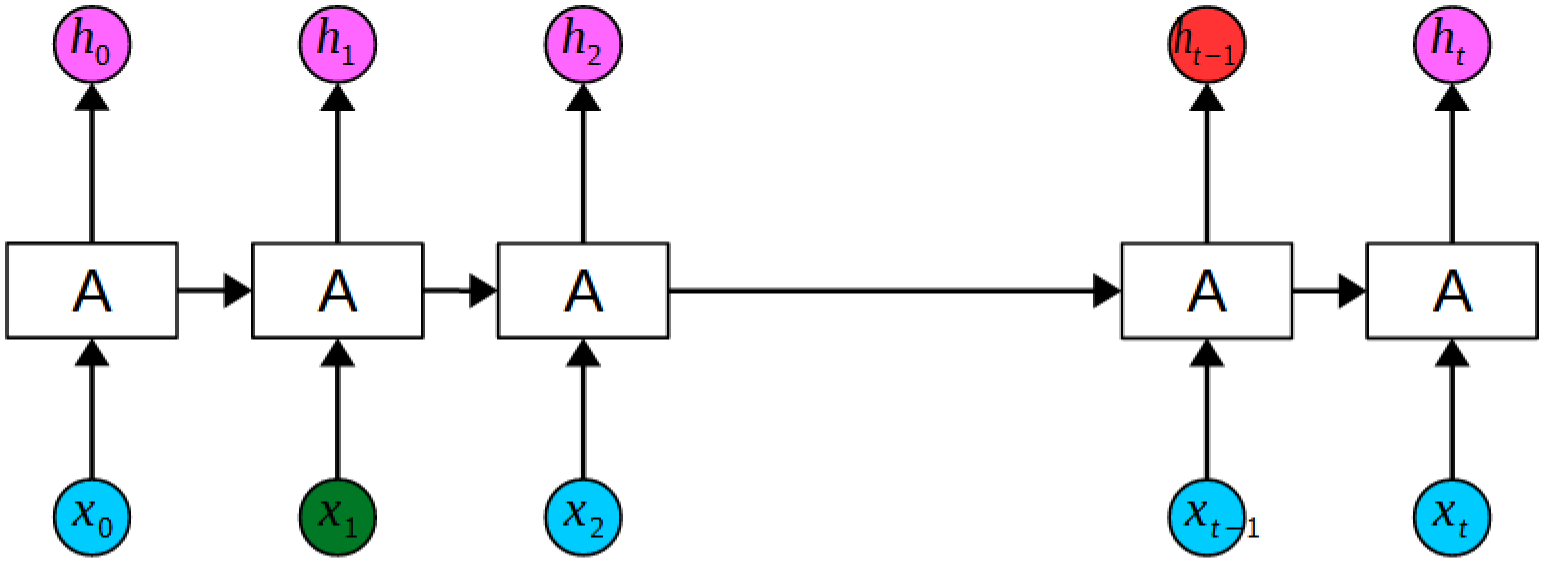
Short term dependency

Language model trying to predict the next word based on the previous ones



Long term dependency

I grew up in India… I speak fluent Hindi.



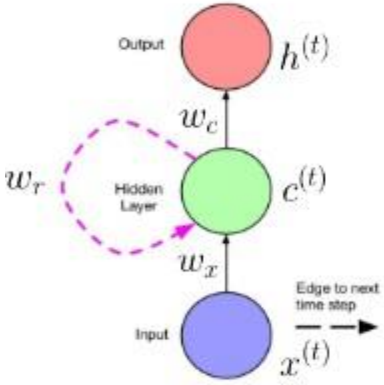
RNN fails to preserve the long term dependency due to the vanishing gradient problem

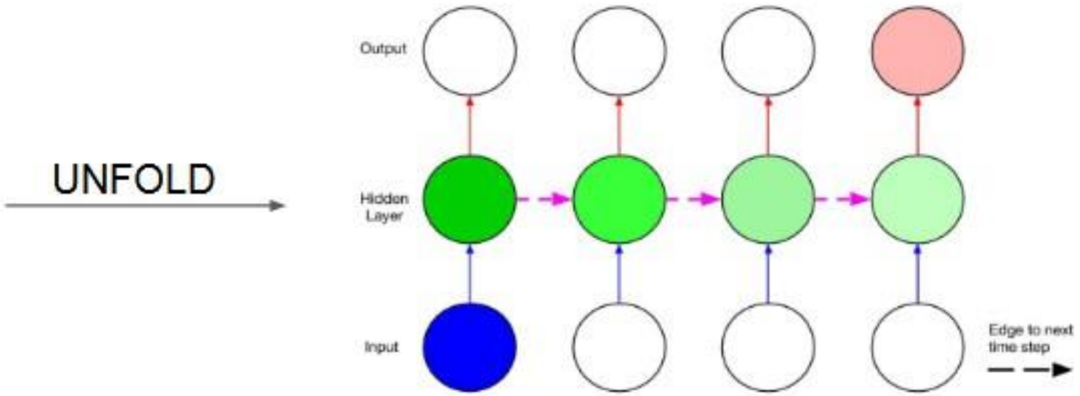
We should distinguish between what to remember and what to forget more wisely

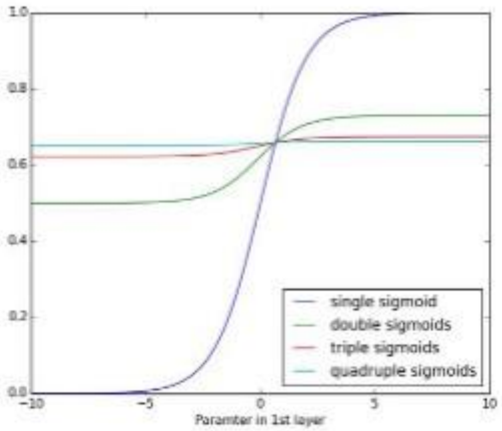
LSTM (Long short term memory) to the rescue

The RNN problem revisited

- Problem: can't capture long-term dependencies due to vanishing/exploding gradients during backpropagation

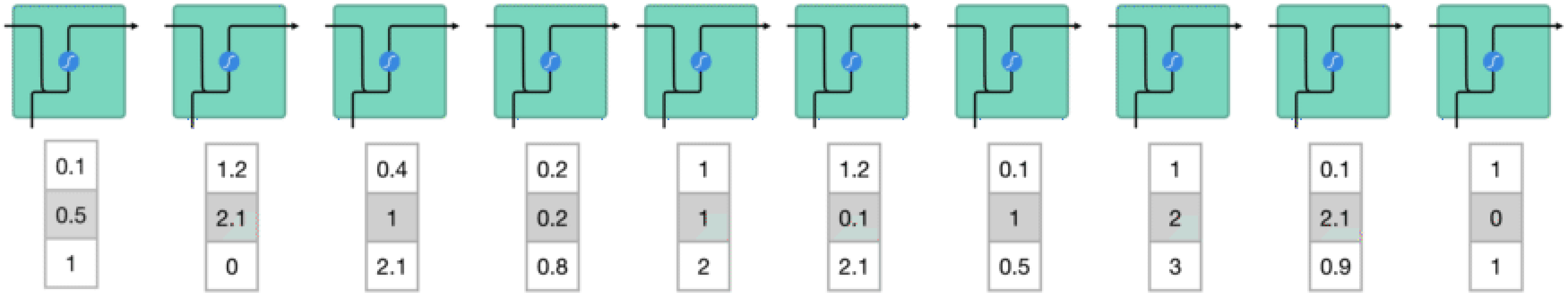


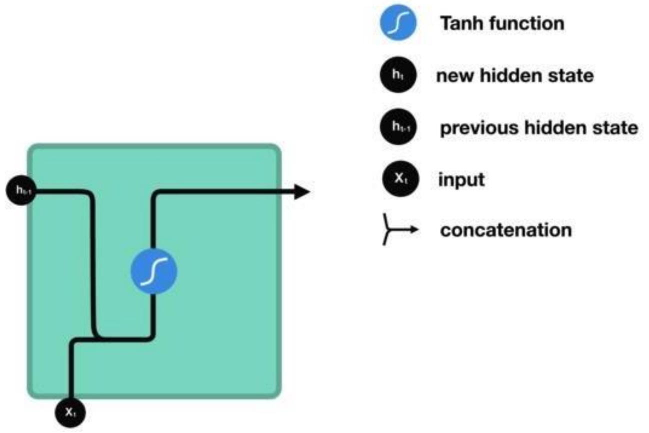




$$h^{(t)} = \sigma(w_c \cdot c^{(t)})$$
$$c^{(t)} = \sigma(w_r \cdot c^{(t-1)} + w_x \cdot x^{(t)})$$

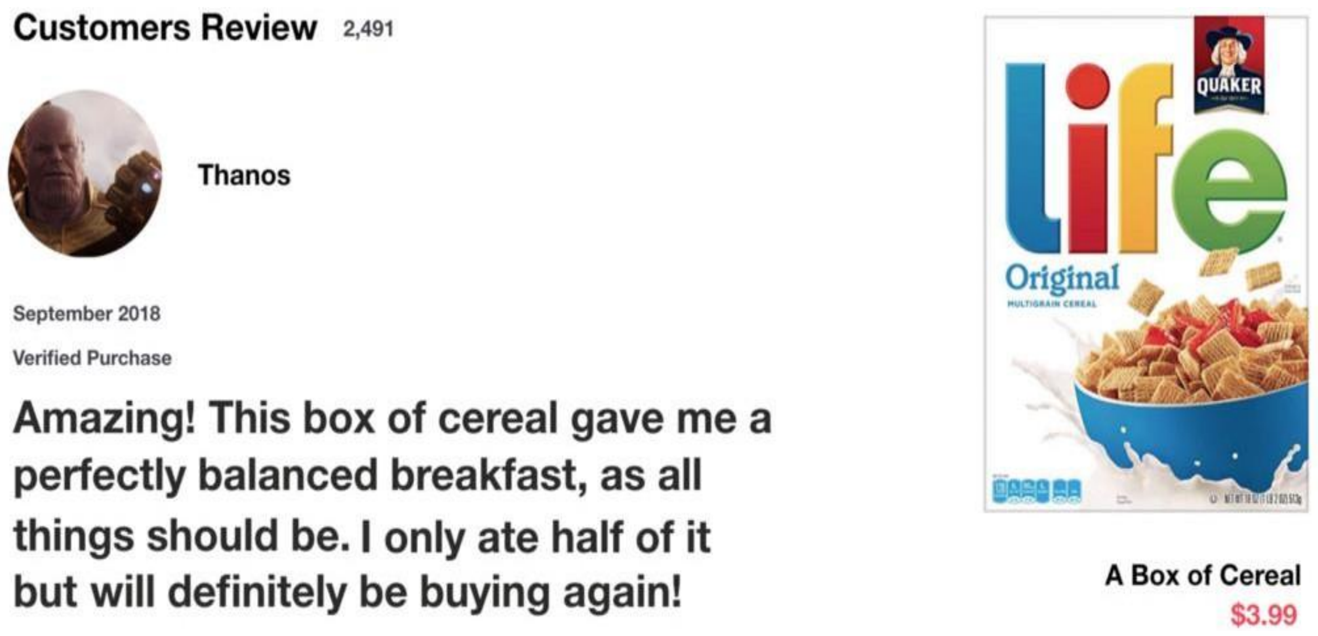
$$\begin{aligned}
h^{(3)} &= \sigma(w_c \cdot c^{(3)}) \\
&= \sigma(w_c \cdot \sigma(w_x \cdot x^{(3)} + w_r \cdot c^{(2)})) \\
&= \sigma(w_c \cdot \sigma(w_x \cdot x^{(3)} + w_r \cdot \sigma(w_x \cdot x^{(2)} + w_r \cdot c^{(1)}))))) \\
&= \sigma(w_c \cdot \sigma(w_x \cdot x^{(3)} + w_r \cdot \sigma(w_x \cdot x^{(2)} + w_r \cdot \sigma(w_x \cdot x^{(1)} + w_r \cdot c^{(0)}))))))
\end{aligned}$$





How RNN works! – trying to process all the words and update memory on that basis

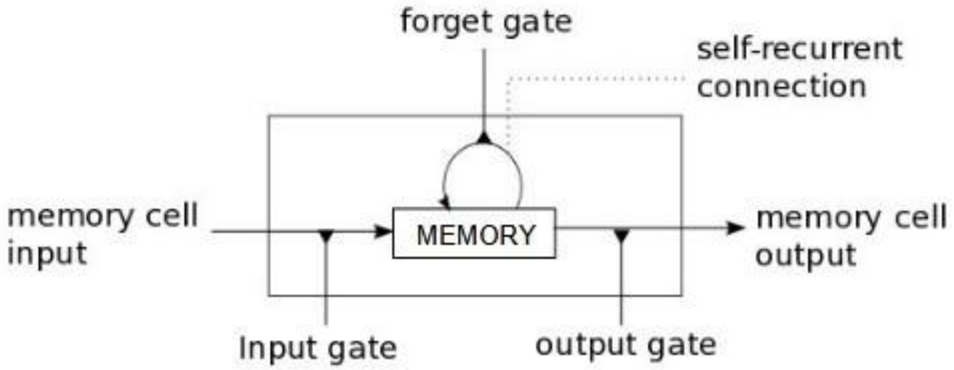
Some intuition



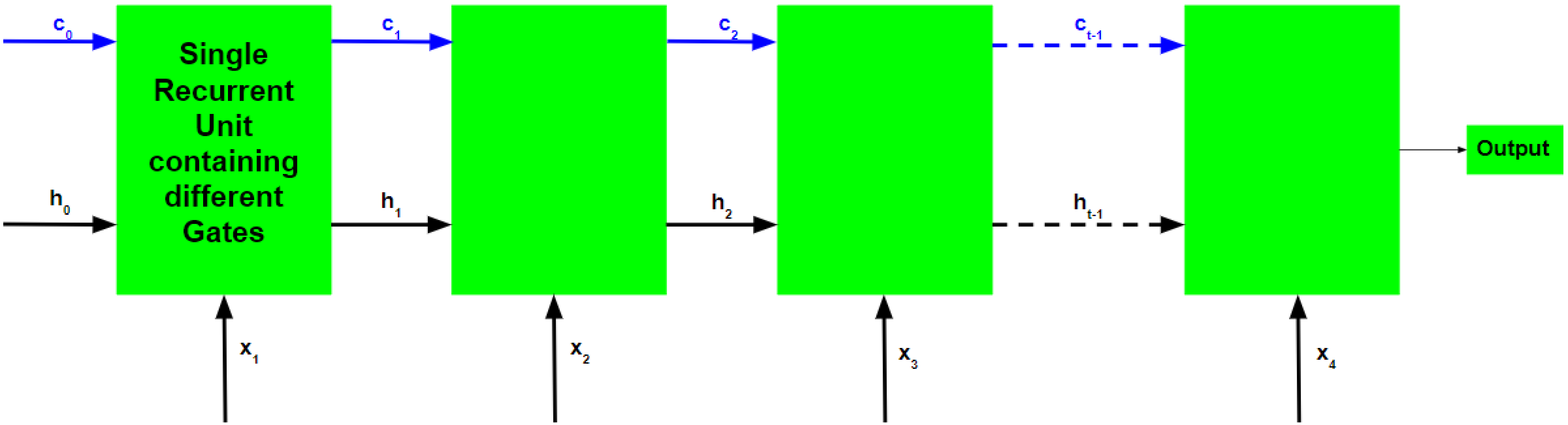
We look for certain words, while ignoring most of the other words

LSTM

Central Idea: A **memory cell** (interchangeably **block**) which can maintain its state over time, consisting of an explicit memory (aka the **cell state vector**) and **gating units** which regulate the information flow into and out of the memory.



LSTM Memory Cell



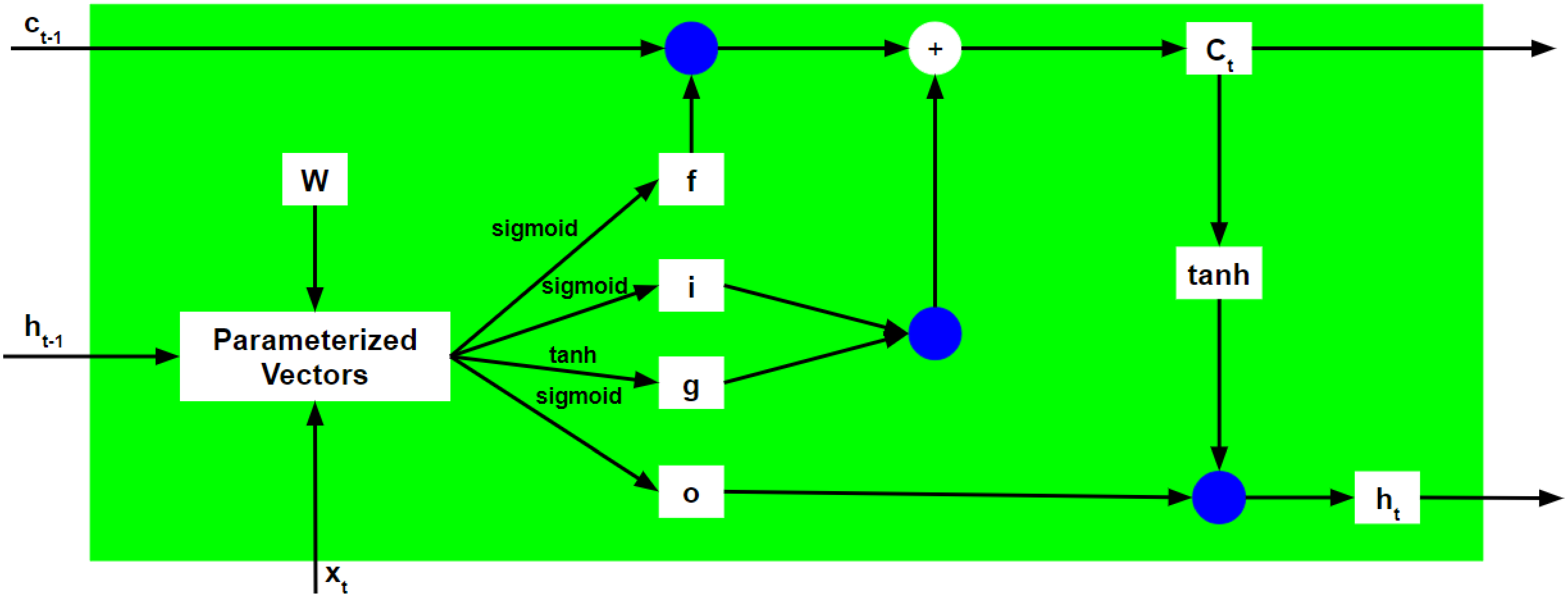
c_0
h_0
Single Recurrent Unit containing different Gates
c_1
h_1
c_2
h_2
c_{t-1}
h_{t-1}
x_1
x_2
x_3
x_4
Output

1. Hidden State (h_t)

- **Short-Term Signal**: The hidden state is effectively the "working" or "immediate" memory. It carries information that is directly used to predict the output at the current timestep (and to help compute the next states).
- **Frequently Updated**: Because it directly connects to the network's output, h_t changes more rapidly and captures shorter-term patterns.

2. Cell State (c_t)

- **Long-Term Memory:** The cell state is designed to propagate information over much longer sequences with fewer modifications. It flows more linearly through the network (with some gating).
- **Reduced Vanishing/Exploding Gradients**: By providing a nearly linear path for gradients, c_t helps mitigate the vanishing and exploding gradient problems, which commonly arise in vanilla RNNs.



c_{t-1}
h_{t-1}
W
Parameterized Vectors
x_t
sigmoid
sigmoid
tanh
sigmoid
f
i
g
o
+
C_t
tanh
h_t

Forget Gate

Purpose: Controls which information from c_{t-1} should be discarded before forming c_t.

$$f_t = \sigma(W_f \cdot [h_{t-1},\, x_t] + b_f)$$

- $\sigma(\cdot)$ is the sigmoid function, which outputs values between 0 and 1.
- f_t is multiplied elementwise with the previous cell state c_{t-1}. Values close to 0 mean "forget most of this," while values close to 1 mean "keep most of this."

Applied to c_{t-1}:

$$\tilde{c}_{t-1} = f_t \odot c_{t-1}$$

Input Gate

Purpose: Determines how much new information from the current timestep's input (and the previous hidden state) is added to the cell state.

This gate is actually split into two parts—an **input gate** and a **candidate cell state**. We first compute a "candidate" update, then use the input gate to decide how much of this candidate actually goes into c_t.

1. **Candidate Cell State** ($\tilde{c}_t$):

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1},\, x_t] + b_c)$$

   This is the new, potential information that could be added to the cell state.

2. **Input Gate** (i_t):

$$i_t = \sigma(W_i \cdot [h_{t-1},\, x_t] + b_i)$$

   Outputs a value between 0 and 1 for each component, controlling how much of $\tilde{c}_t$ is added.

Combined effect:

$$i_t \odot \tilde{c}_t$$

This value is added to the (partially) "forgotten" previous cell state $\tilde{c}_{t-1}$ to form the new cell state.

With the forget gate and the input gate, we can update c_t as follows:

$$c_t = \underbrace{(f_t \odot c_{t-1})}_{\text{forgotten old state}} + \underbrace{(i_t \odot \tilde{c}_t)}_{\text{new information}}$$

Output Gate

Purpose: Determines how much of the new cell state c_t will be used to compute the hidden state h_t.

$$o_t = \sigma(W_o \cdot [h_{t-1},\, x_t] + b_o)$$

- $\sigma(\cdot)$ is the sigmoid function (again producing values between 0 and 1).

We then apply a $\tanh$ function to the newly updated cell state c_t to push values between -1 and +1, and multiply elementwise by o_t to get the final hidden state h_t:

$$h_t = o_t \odot \tanh(c_t)$$

Summarizing the equations

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma\left(W_o \ [\,h_{t-1}, x_t] \ + \ b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Training LSTM

- Backpropagation through time

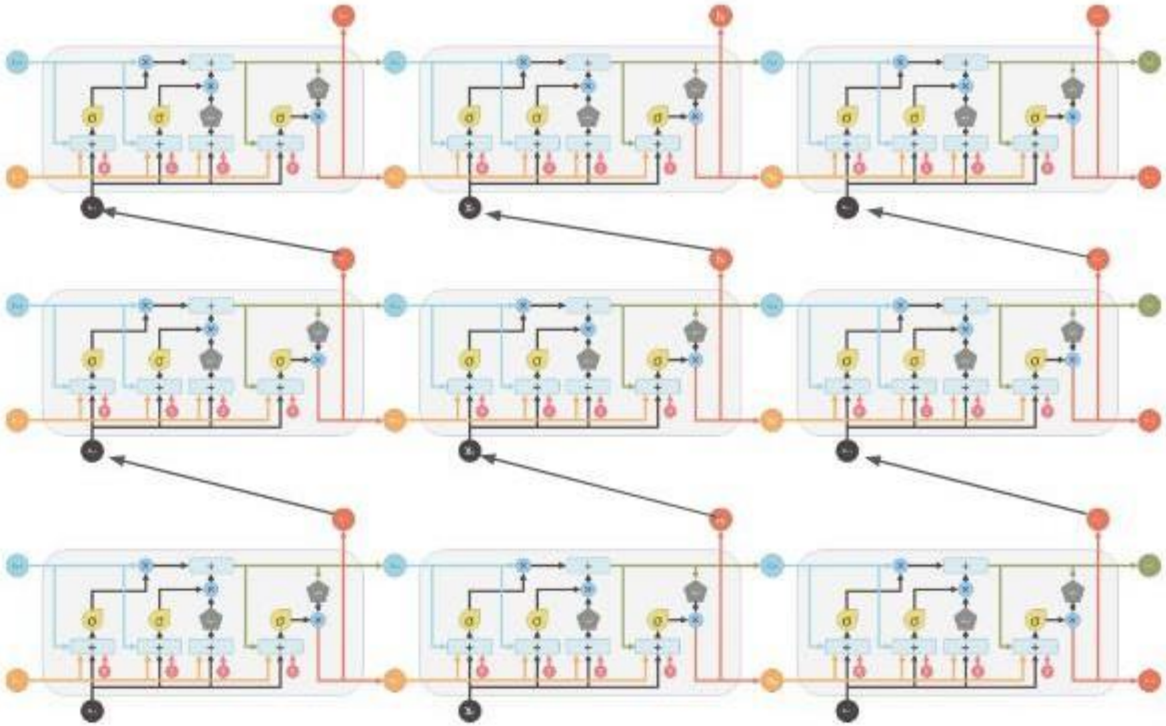
- Each cell has many parameters (W_f, W_i, W_C, W_o)
  - Generally requires lots of training data.
  - Requires lots of compute time that exploits GPU clusters.

Deep LSTM

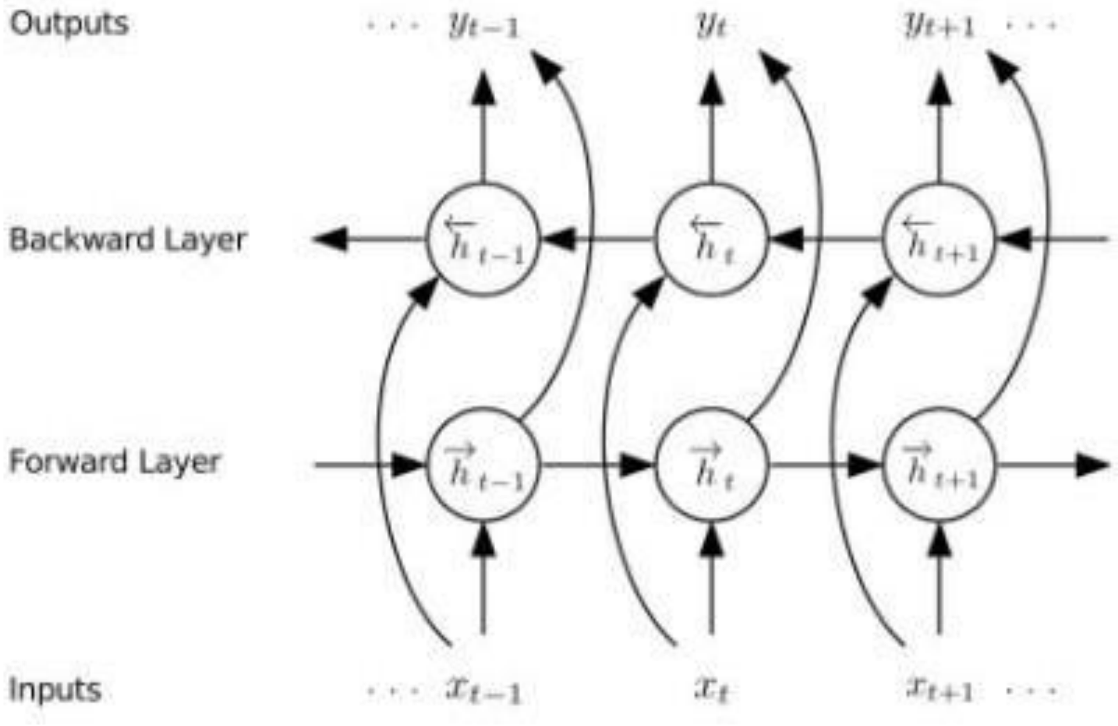
- Deep LSTMs can be created by stacking multiple LSTM layers vertically, with the output sequence of one layer forming the input sequence of the next (in addition to recurrent connections within the same layer)

- Increases the number of parameters - but given sufficient data, performs significantly better than single-layer LSTMs (Graves et al. 2013)

- Dropout usually applied only to non-recurrent edges, including between layers



Bi-directional recurrent model



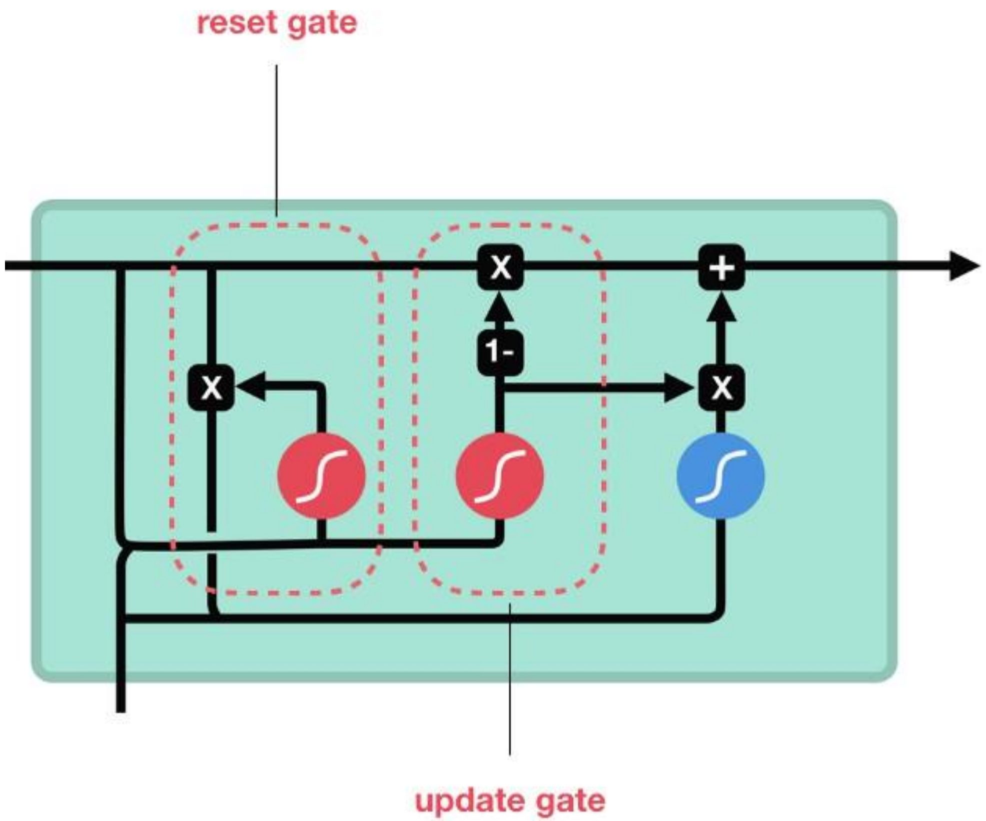
1. **Forward Pass:**
   - Processes the input sequence $(x_1, x_2, \ldots, x_T)$ from $t = 1$ to $t = T$.
   - Produces a set of "forward" hidden states $(\overrightarrow{h}_1, \overrightarrow{h}_2, \ldots, \overrightarrow{h}_T)$.
2. **Backward Pass:**
   - Processes the sequence in reverse $(x_T, x_{T-1}, \ldots, x_1)$.
   - Produces a set of "backward" hidden states $(\overleftarrow{h}_T, \overleftarrow{h}_{T-1}, \ldots, \overleftarrow{h}_1)$.
3. **Combining Forward and Backward States:**

   At each timestep t, the forward hidden state $\overrightarrow{h}_t$ and the backward hidden state $\overleftarrow{h}_t$ are combined (often by **concatenation** or **addition**) to form a single output representation:

   $$h_t = \left[\overrightarrow{h}_t; \overleftarrow{h}_t\right]$$

   where `;` denotes concatenation. This final h_t captures information from **both past and future context** relative to timestep t.

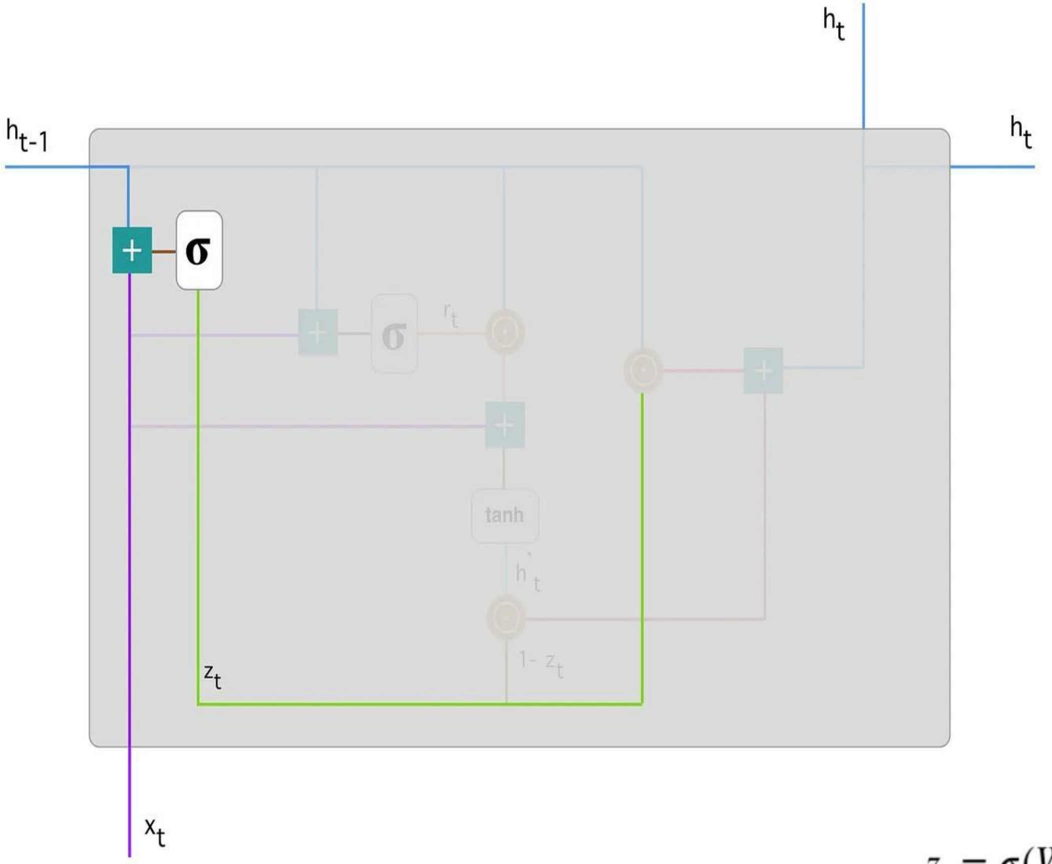
Gated recurrent units



GRU’s got rid of the cell state and used the hidden state to transfer information. It also only has two gates, a reset gate and update gate.

Update gate

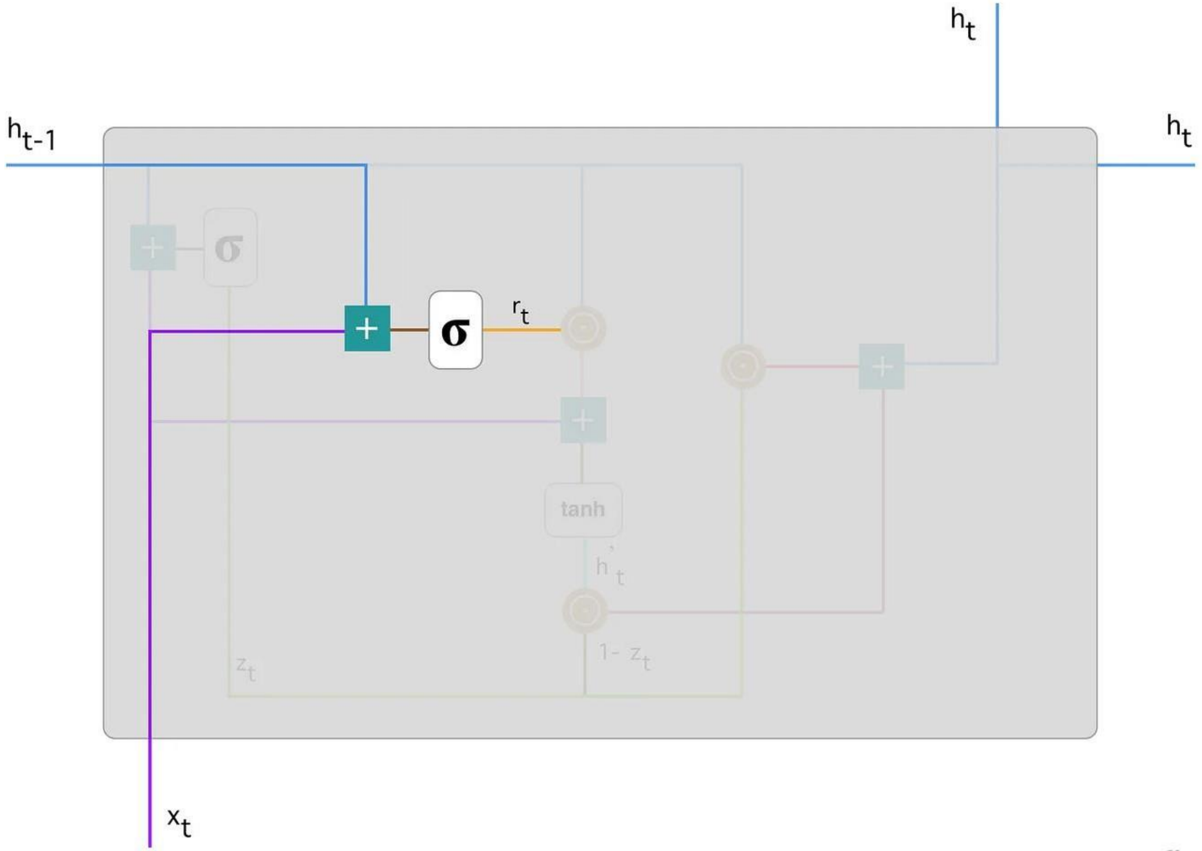
The update gate decides how much of the previous hidden state h_{t-1} should be carried over to the new hidden state. A higher value of z_t means “keep more of the past information,” while a lower value means “let in more of the new candidate information.”



$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

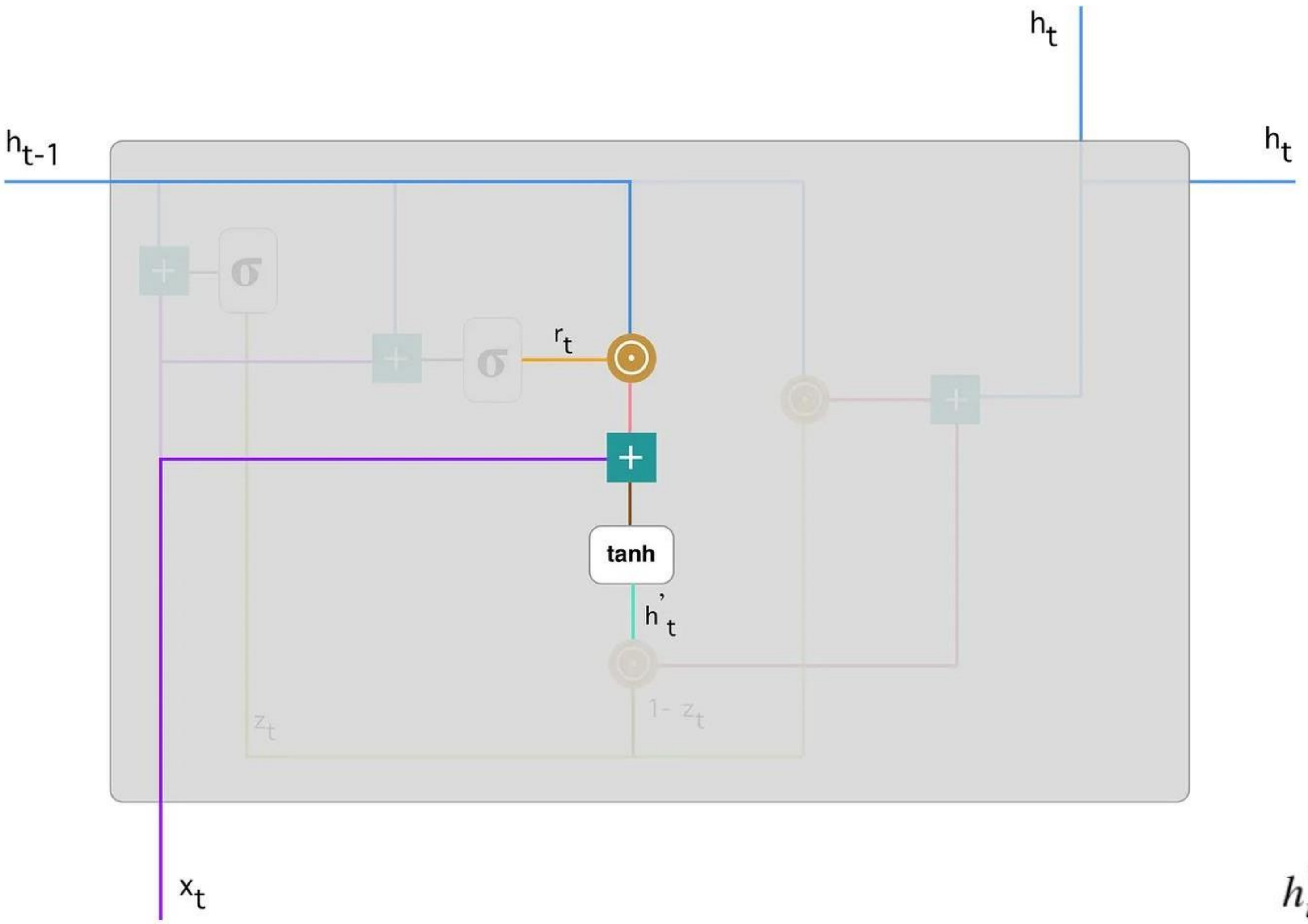
Reset gate

The reset gate determines how much of the past hidden state h_{t-1} to "forget" (or reset) before calculating the new candidate hidden state. A lower value of r_t means the model puts less emphasis on the previous hidden state, effectively forgetting more past context.



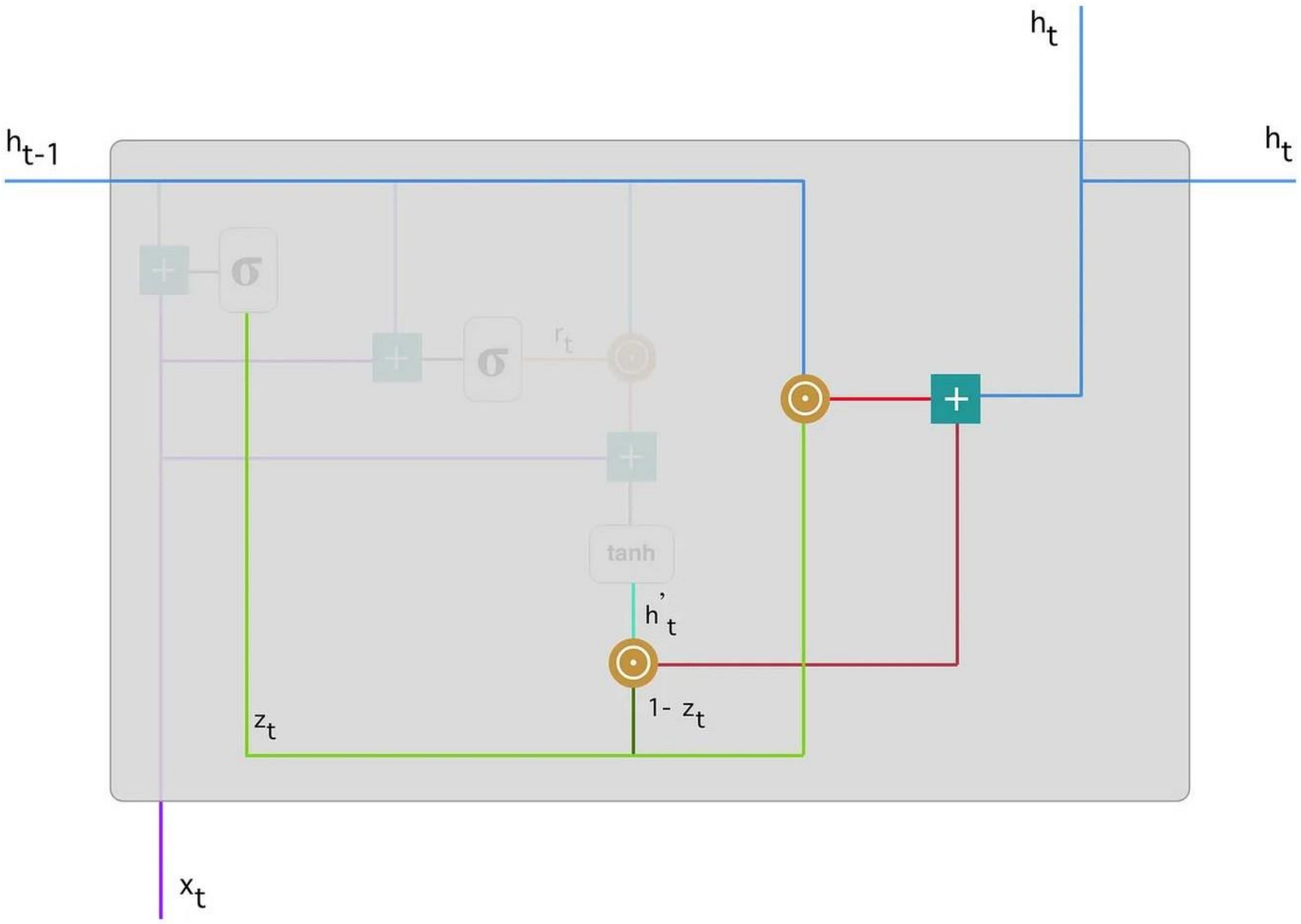
$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

Creating intermediate memory



$$h_t' = \tanh(Wx_t + r_t \odot Uh_{t-1})$$

Outputting the next hidden state



$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t^{'}$$

```
initialize hidden state h_0 = 0
for t in 1 to T:
    z_t = sigmoid(W_z * [h_{t-1}, x_t] + b_z)
    r_t = sigmoid(W_r * [h_{t-1}, x_t] + b_r)
    h_t_candidate = tanh(W_h * [r_t * h_{t-1}, x_t] + b_h)
    h_t = (1 - z_t) * h_{t-1} + z_t * h_t_candidate
```

Consider a linear autoencoder with:

- An input layer of dimension d.
- A hidden/"bottleneck" layer of dimension k (with $k < d$).
- A reconstruction layer also of dimension d.

Let the encoder be described by a weight matrix $W \in \mathbb{R}^{k \times d}$ and bias $\mathbf{b} \in \mathbb{R}^k$, and the decoder by another weight matrix $V \in \mathbb{R}^{d \times k}$ and bias $\mathbf{c} \in \mathbb{R}^d$. For a dataset $\{\mathbf{x}^{(n)}\}_{n=1}^N$ with $\mathbf{x}^{(n)} \in \mathbb{R}^d$, define the (mean-squared) reconstruction loss as:

$$\mathcal{L} = \sum_{n=1}^{N} \left\| \mathbf{x}^{(n)} - \hat{\mathbf{x}}^{(n)} \right\|^2,$$

where

$$\hat{\mathbf{x}}^{(n)} = V \left(W \, \mathbf{x}^{(n)} + \mathbf{b} \right) + \mathbf{c}.$$

1. **Show** that, after removing any mean offset of the data via preprocessing (so you can assume $\sum_{n=1}^N \mathbf{x}^{(n)} = \mathbf{0}$ and drop the bias terms $\mathbf{b}, \mathbf{c}$ for simplicity), the weight matrices W and V that **minimize** $\mathcal{L}$ satisfy

$$V = W^\top.$$

2. **Prove** that the optimal matrix W^* (which also implies $V^* = (W^*)^\top$) forms an orthonormal basis for the **top** k principal components of the data covariance matrix

$$\Sigma = \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}^{(n)} (\mathbf{x}^{(n)})^\top.$$

   In other words, the rows of W^* span the same subspace as the eigenvectors of Σ associated with its largest k eigenvalues.