UNSW
THE UNIVERSITY OF NEW SOUTH WALES

5a. Recurrent Neural Networks

Never Stand Still | Faculty of Engineering | COMP9444 Week 5

Hao Xue
School of Computer Science and Engineering
Faculty of Engineering
The University of New South Wales, Sydney, Australia
cs9444@cse.unsw.edu.au

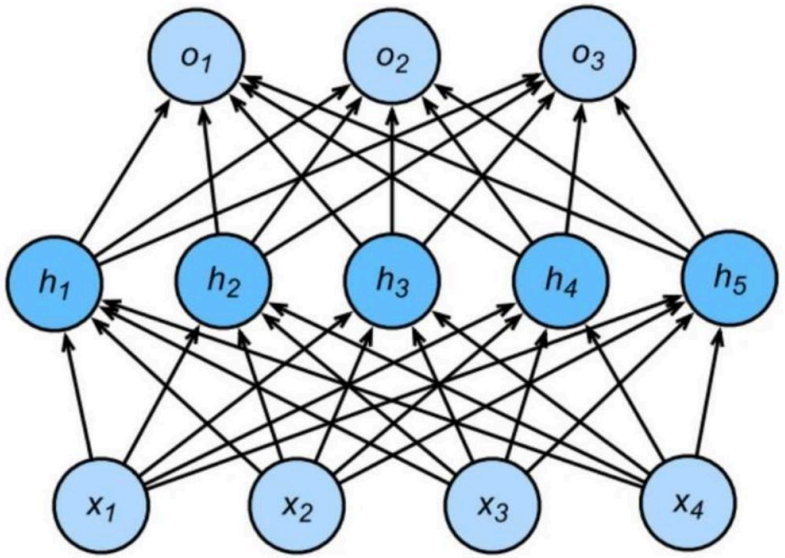
Outline

- Processing Temporal Sequences
- Sliding Window
- Recurrent Network Architectures
- Long Short Term Memory (LSTM)
- Gated Recurrent Unit (GRU)
- Applications

Processing Temporal Sequences

- An important domain: There are many tasks which require a sequence of inputs to be processed rather
- Sequential Data:
  - Different from image data
  - Each data instance: a sequence of data points, x_t , $for\ 1 \leq t \leq T$
  - Various length T
  - Label: can be a scalar, a vector, or even a sequence
- Speech Recognition, Time Series Prediction, Machine Translation
- So: How can neural network models be adapted for these tasks?

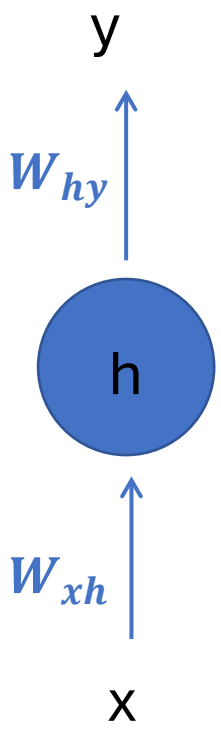
Feedforward Neural Network



Output layer

Hidden layer

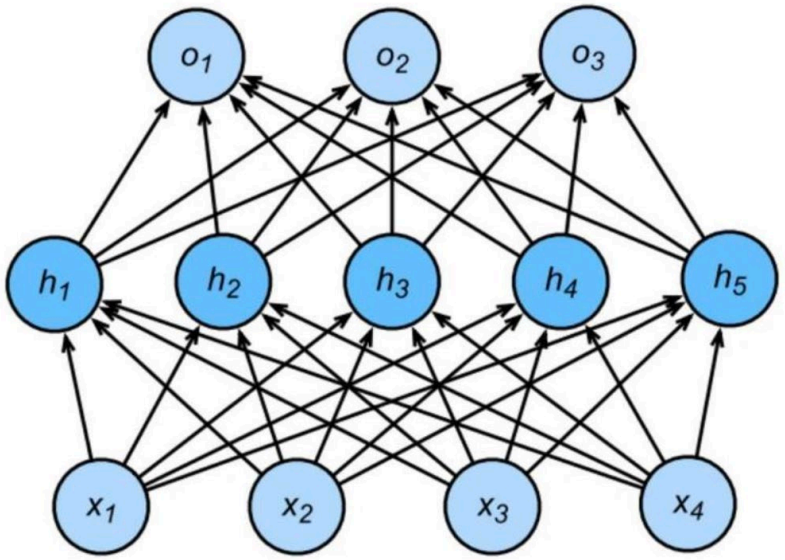
Input layer



Any issues for sequential data?

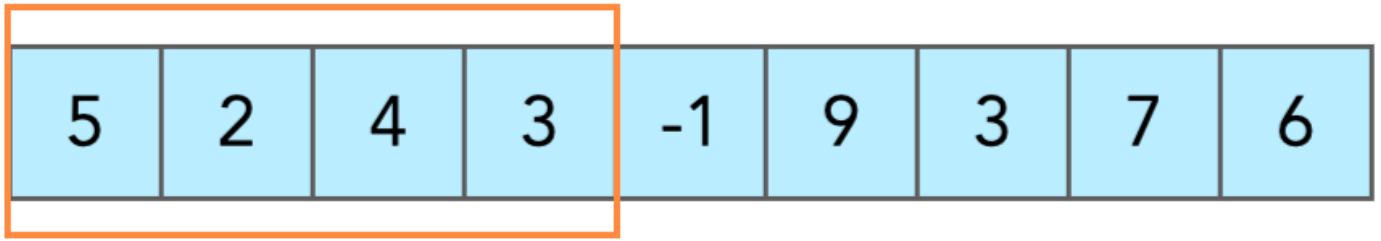
Issue 1

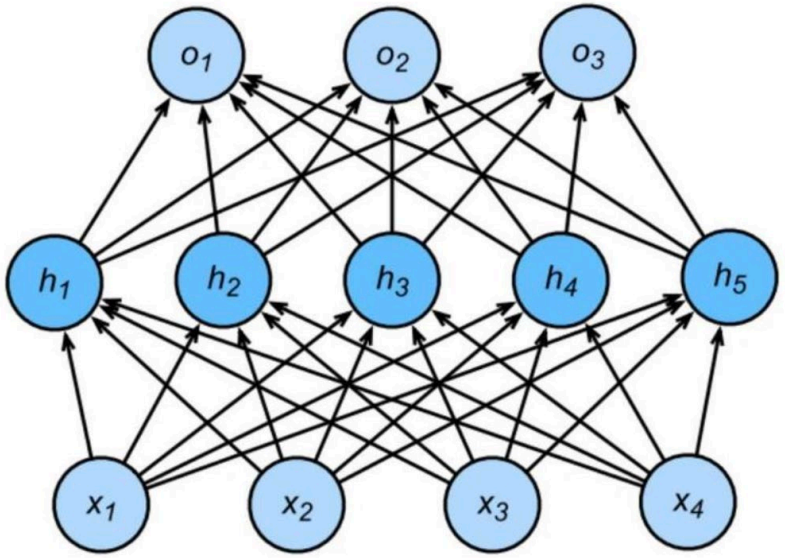
- Feedforward neural networks require fixed-size inputs
- However
  - Temporal data often comes in long sequences
  - Real-world sequences (e.g., sensor readings) often vary in length, making it harder to batch them directly



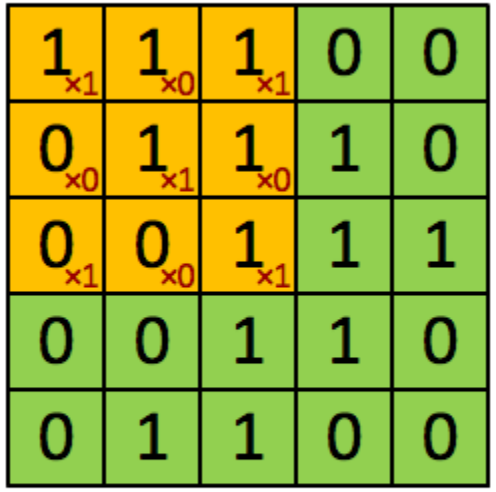
Sliding Window

Sliding window helps create fixed-size input/output pairs from sequences

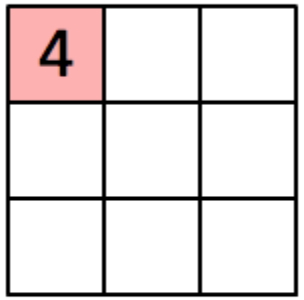




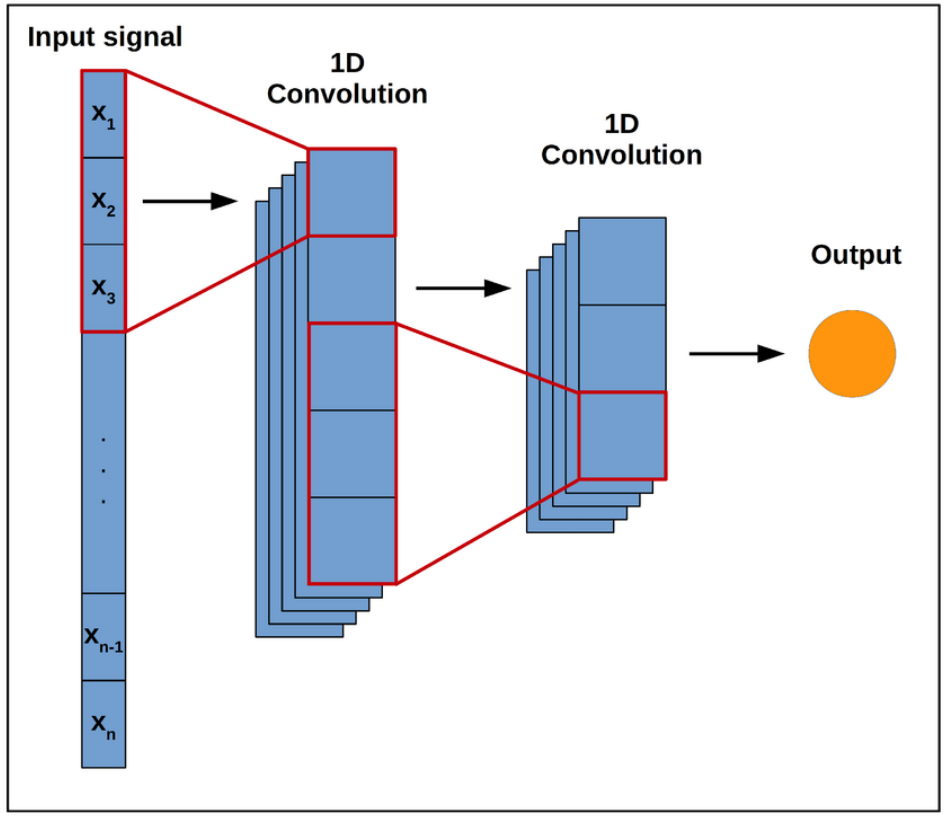
Sliding Window



Image



Convolved Feature



Issue 2: temporal dependencies

- Feedforward networks treat inputs as fixed-size vectors without order
- No inherent mechanism to model temporal dependencies or sequence dynamics
- But order is critical in time series and language data
  - E.g., “How are you” != “You are how”

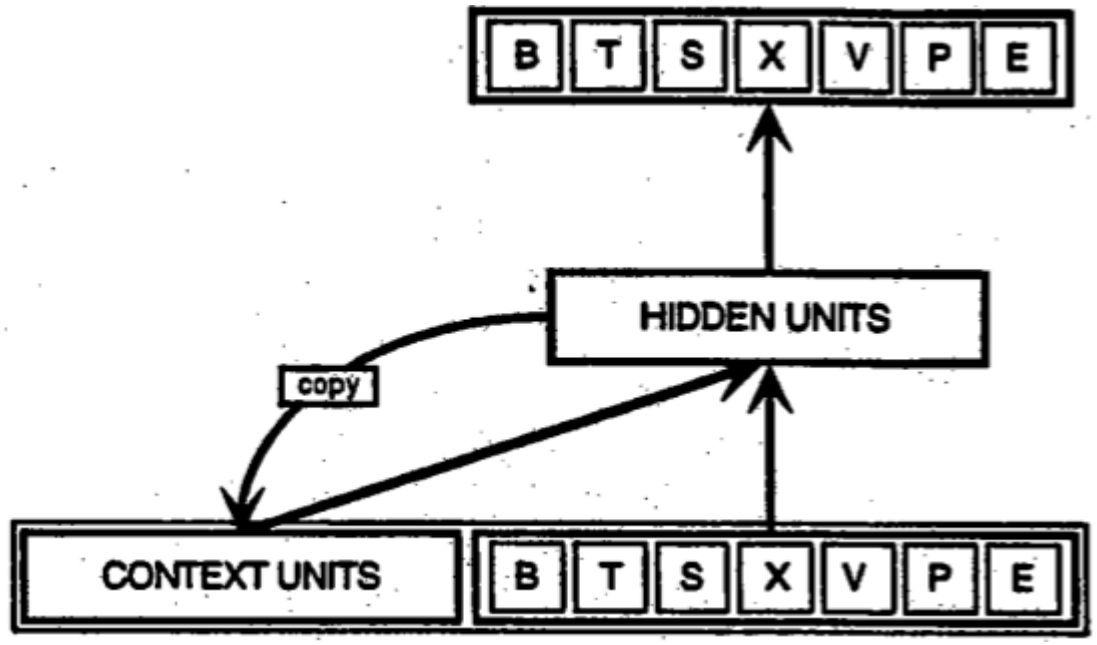
- What follows after: “I study at UNSW and I am” “a student”
- What follows after: “I UNSW at and study I am” “a student” ???
- The order of words matters. This is true for most sequential data.
  - A fully connected network will not distinguish the order well
  - Hence, missing some information, missing context!

Recurrent Neural Networks

RNNs: designed for sequential data

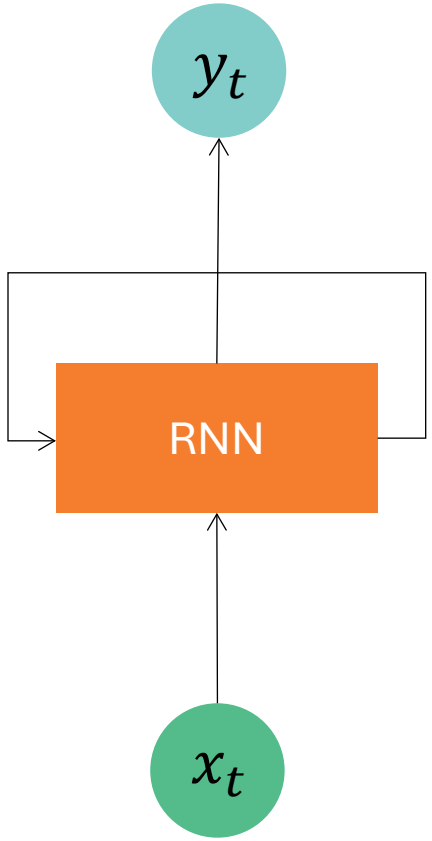
- Recurrent in nature: performs the same function for every input of data while the output of the current input depends on the past one computation
- RNNs can handle arbitrary input / output lengths
- Unlike feed-forward neural networks, RNNs can use their internal state (memory) to process sequences of inputs

Simple Recurrent Network (Elman, 1990)



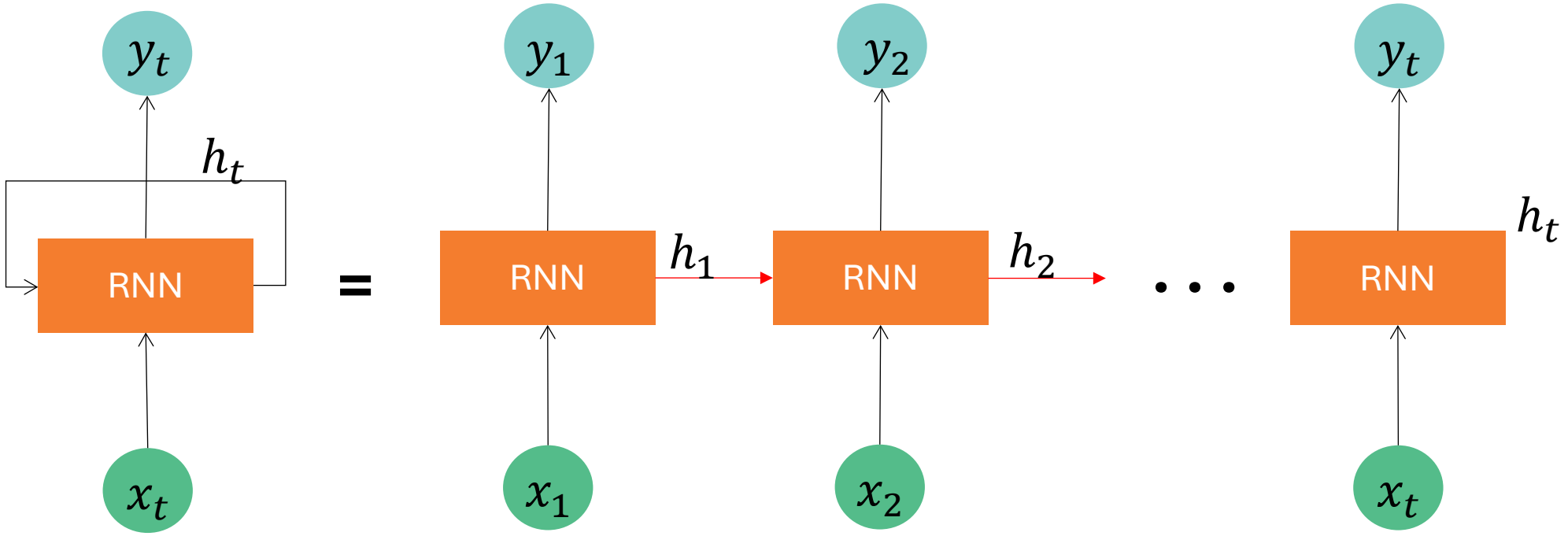
- at each time step, hidden layer activations are copied to “context” layer
- hidden layer receives connections from input and context layers
- the inputs are fed one at a time to the network, it uses the context layer to “remember” whatever information is required for it to produce the correct output

RNNs

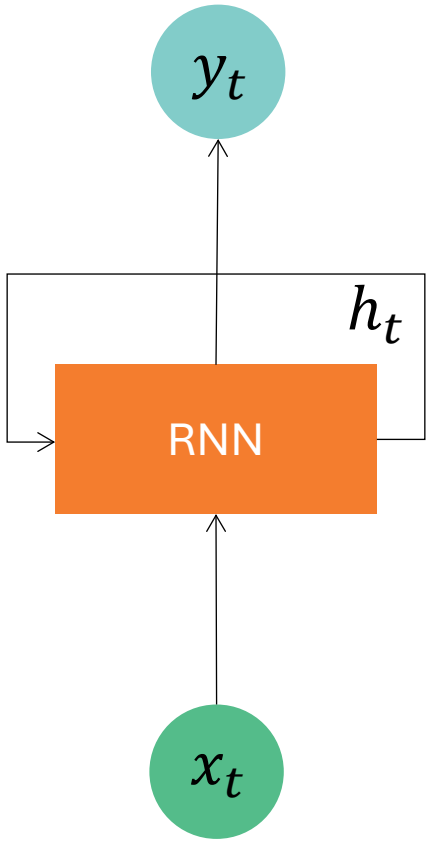


Key idea: "**internal state**" that is updated as a sequence is processed

Unfolded RNNs



Understanding the Recurrence: hidden state



Process a sequence of data points x_t by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

new state

function with parameters *W*

old state (last time step)

input vector at time step *t*

Notice: the same function and the same set of parameters are used at every time step.

RNNs: hidden state calculation

$$h_t = f_w(h_{t-1}, x_t)$$

Vanilla RNN

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

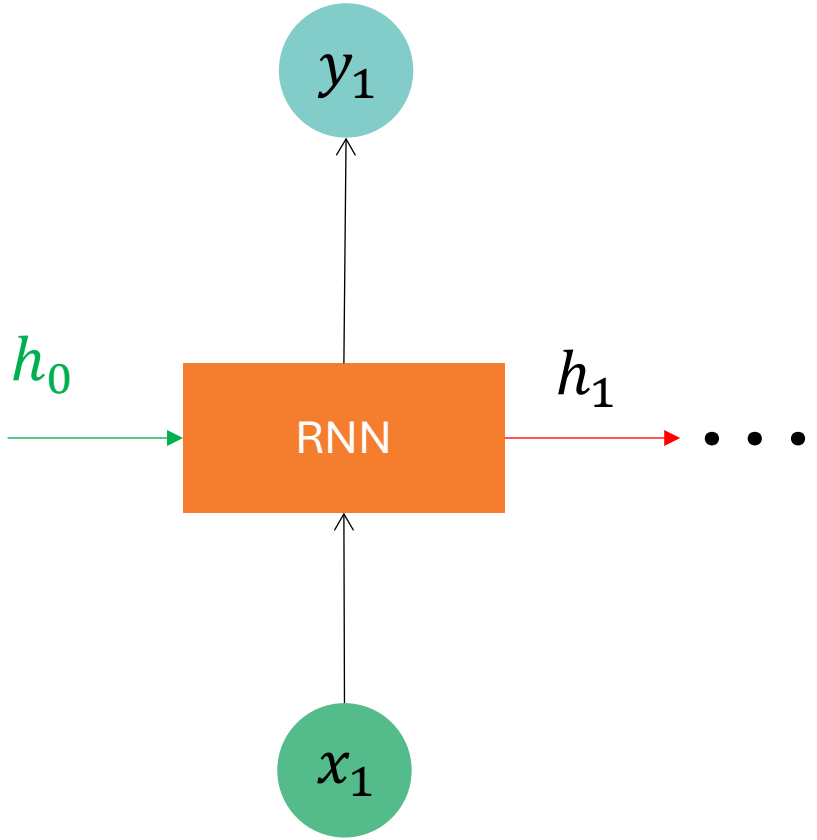
$$y_t = W_{hy}h_t$$

RNNs

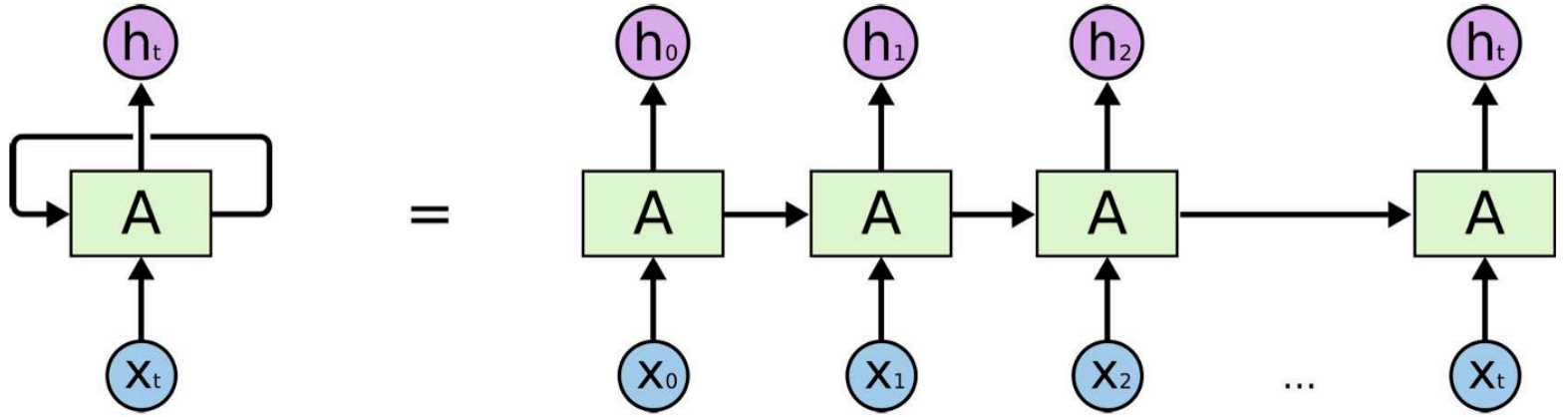
$$h_t = f_w(h_{t-1}, x_t)$$

Q: How to get the first hidden state h_1 ?

```
# Initialize the hidden state
hidden = torch.zeros(num_layers, batch_size, hidden_size)
```

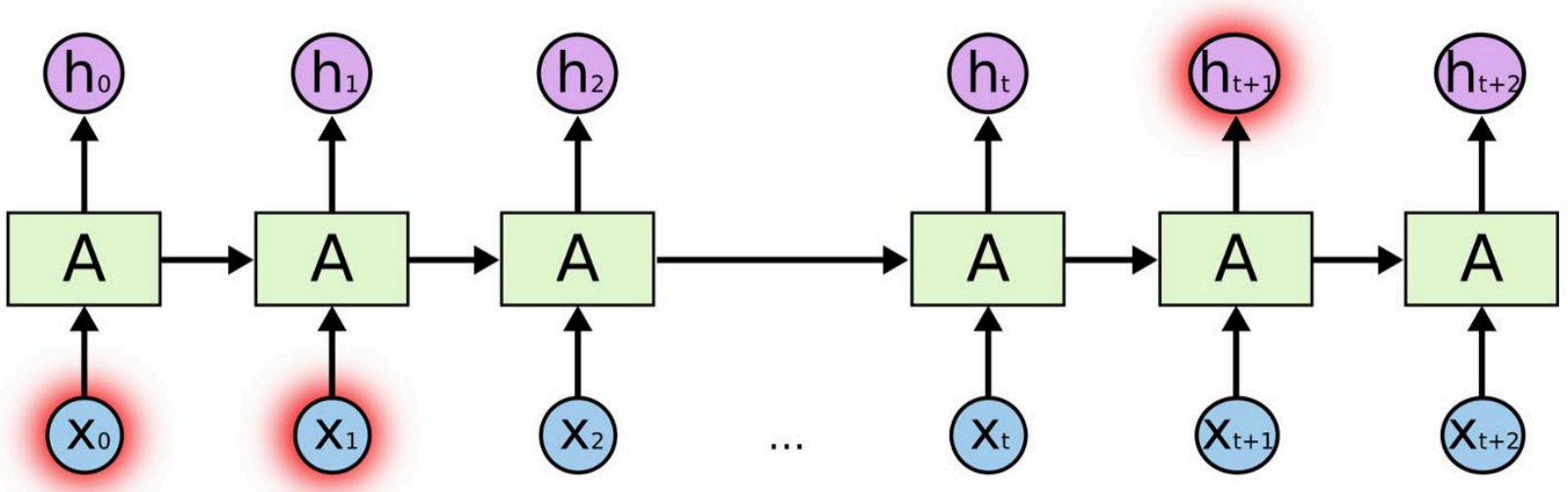


Back Propagation Through Time



- We can “unroll” a recurrent architecture into an equivalent feedforward architecture, with shared weights
- Applying backpropagation to the unrolled architecture is reffered to as “backpropagation through time”
- We can back propagate just one timestep, or a fixed number of timesteps, or all the way back to beginning of the sequence

Long Range Dependencies



- Basic RNNs/ Simple Recurrent Network can learn medium-range dependencies but have difficulty learning long range dependencies
- Why modelling long range is hard?

Vanishing and Exploding Gradients

- Gradients are computed by moving backward through the unfolded network
- The key challenge is that the same parameters (e.g., W_{hh}) are shared across all time steps.
- The gradient of the loss with respect to a weight (e.g., W_{hh}) is the sum of gradients at each time step
- Because h_t depends on h_{t-1}, gradients are computed recursively using the chain rule

$$h_t = \tanh(W_{ih}x_t + W_{hh}h_{t-1})$$

$$y_t = W_{ho}h_t$$

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t(y_t, \hat{y}_t)$$

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}}$$

$$\frac{\partial L_t}{\partial W_{hh}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}$$

Vanishing and Exploding Gradients

- Involves repeated multiplication

$$\frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k+1}}{\partial h_k}$$

- This long chain of multiplications of derivatives and weight matrices:
  - < 1 → multiplication shrinks the gradient exponentially → vanishing gradient
  - > 1 → multiplication grows the gradient exponentially → exploding gradient

$$0.9 \times 0.9 \times \cdots \times 0.9 = 0.9^T \to 0 \quad \text{as } T \to \infty$$

Vanishing and Exploding Gradients

- Vanishing Gradients:
  - When gradients become too small, earlier time steps (far from *t*) receive negligible updates
  - The RNN "forgets" long-range dependencies.

$$W_{hh} \leftarrow W_{hh} - \eta \frac{\partial L}{\partial W_{hh}}$$

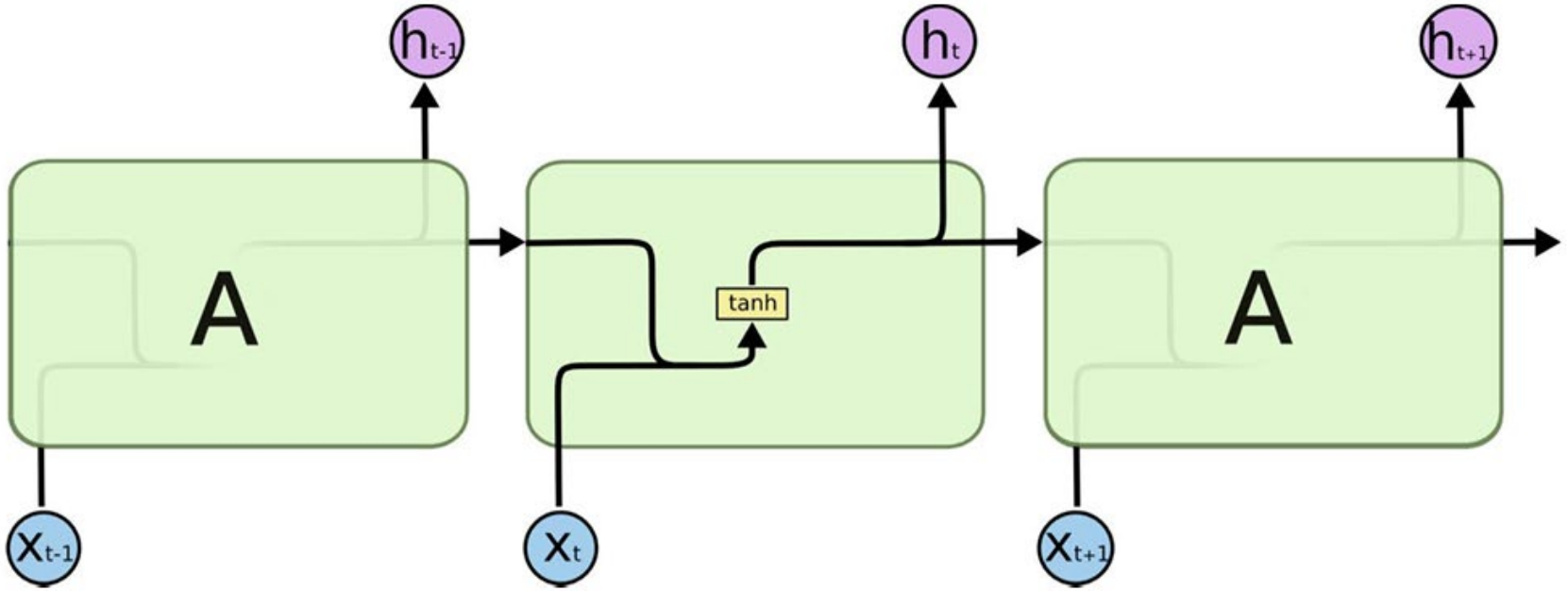
- Exploding Gradients:
  - When gradients grow too large, parameter updates destabilize training
  - May cause NaN errors or chaotic behavior.
  - Can be mitigated with gradient clipping

LSTM & GRU

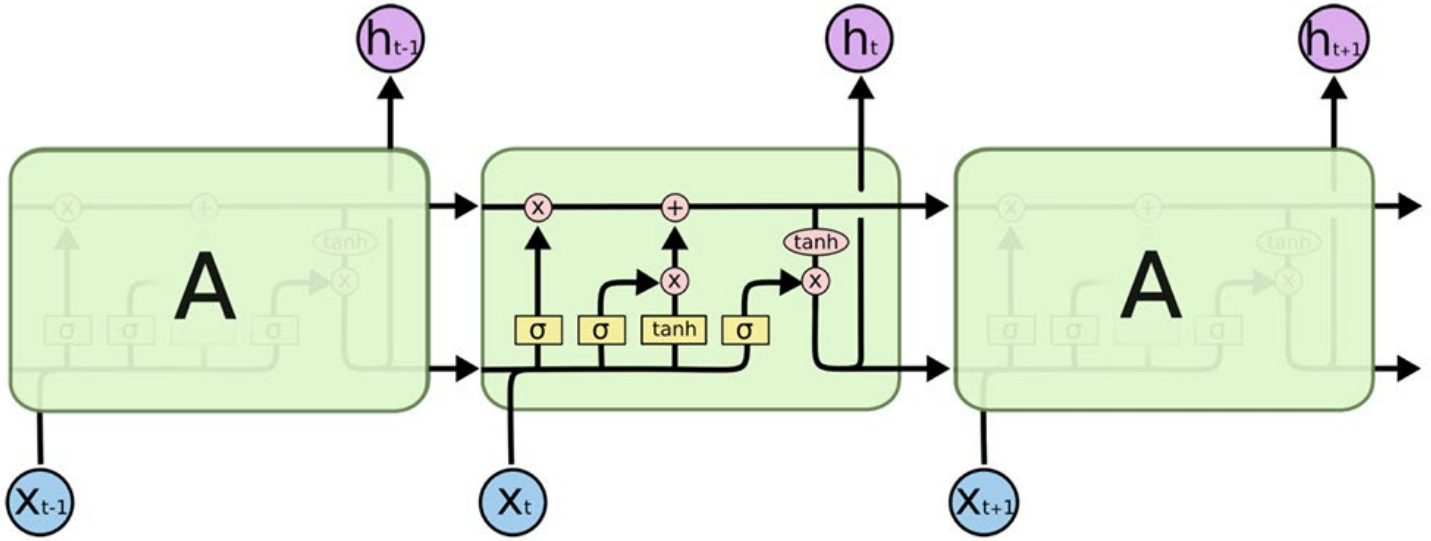
- To address the long range dependencies, Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU) are designed.
- Designed with gates to control information flow (what to keep, forget, or output)

Recall: Vanilla RNN

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

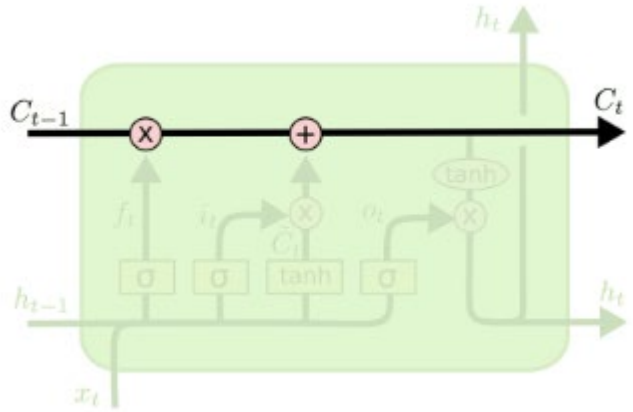


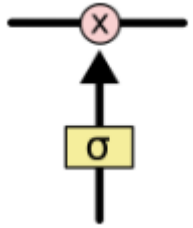
LSTM



- LSTM–context layer is modulated by three gating mechanisms: forget gate, input gate and output gate, plus a cell state that carries long-term memory
  - Forget gate: What to erase from memory
  - Input gate: What to add to memory
  - Output gate: What to output from memory

LSTM: Cell State & Gating Operation



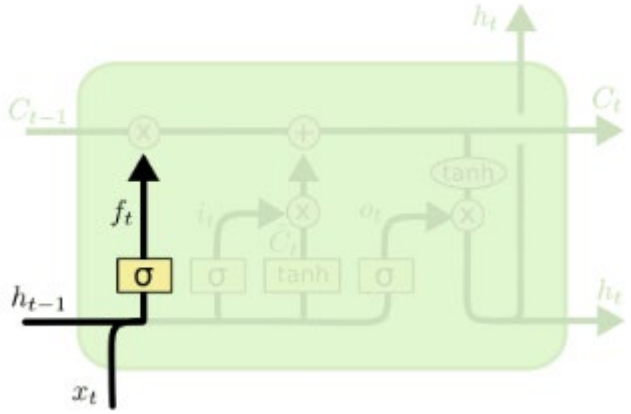


- Cell state C_t is the LSTM's memory , which carries long-term information
- It flows with minimal modification, which helps preserve gradients across time

- To remove or add information to the cell state, with gates.
- Gating is like a switch (using Sigmoid):
  - A value of zero means "let nothing through"
  - A value of one means "let everything through"

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: Forget Gate

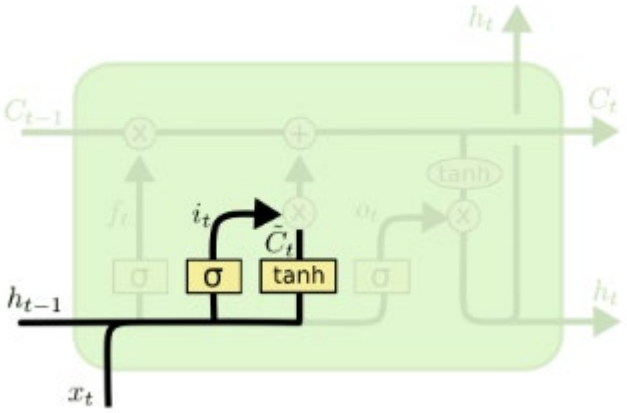


$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

- What to Remove: what information we're going to throw away from the cell state
- Decides what portion of the previous memory C_{t-1} to retain
- Acts as a gate between the past and present
- Helps the model learn when to forget outdated or irrelevant info
- Close to 1: retain memory
- Close to 0: discard memory

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: Input Gate & Candidate Cell State

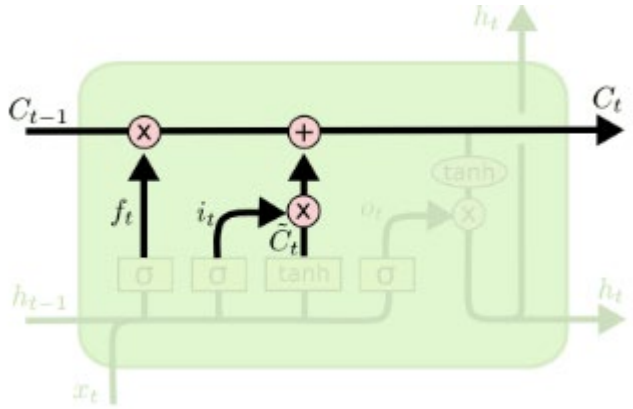


$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

- Controls how much new information (i.e., x_t) is written to the cell state
- The candidate state $\tilde{C}_t$ is a proposal for what could be added to the memory at time
- But is not added directly to the memory, it's filtered by the input gate
  - Candidate: What to write into memory (content)
  - Input Gate: How much to write into memory (strength)

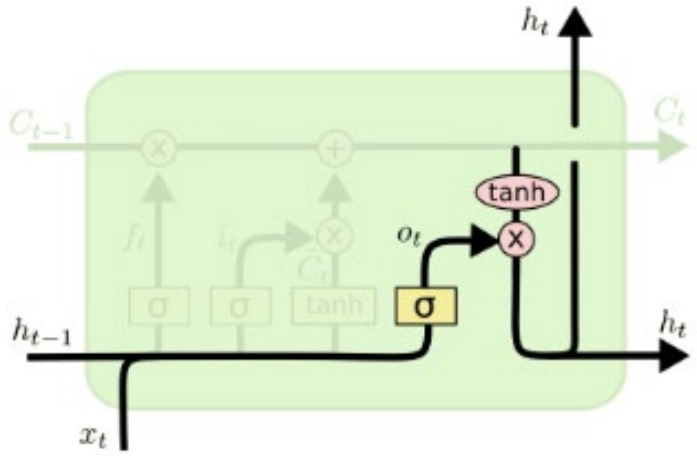
LSTM: Updating Cell Memory



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

- forget gate decides what to keep from the past
- input gate decides what new information to add
- They are combined additively to form the new memory

LSTM: Output Gate

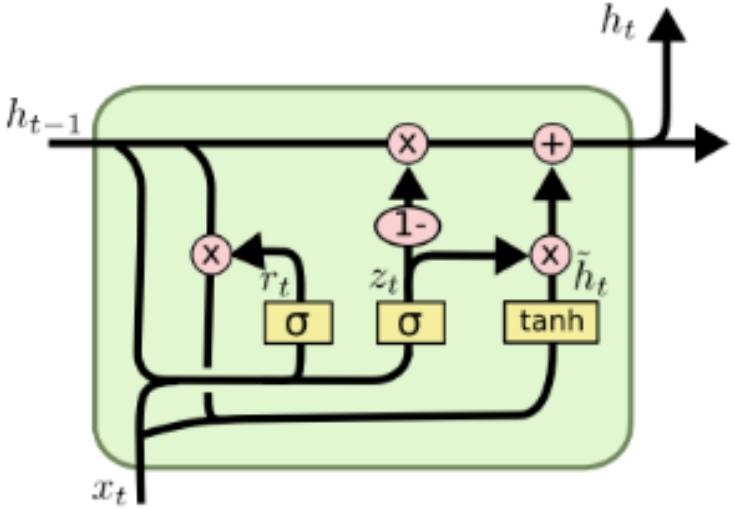


$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

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

- The output gate decides how much of the current memory should go to output

Gated Recurrent Unit (GRU): A Simpler Alternative to LSTM



$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \quad \text{(Update gate)}$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \quad \text{(Reset gate)}$$
$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \quad \text{(Candidate state)}$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \quad \text{(Final hidden state)}$$

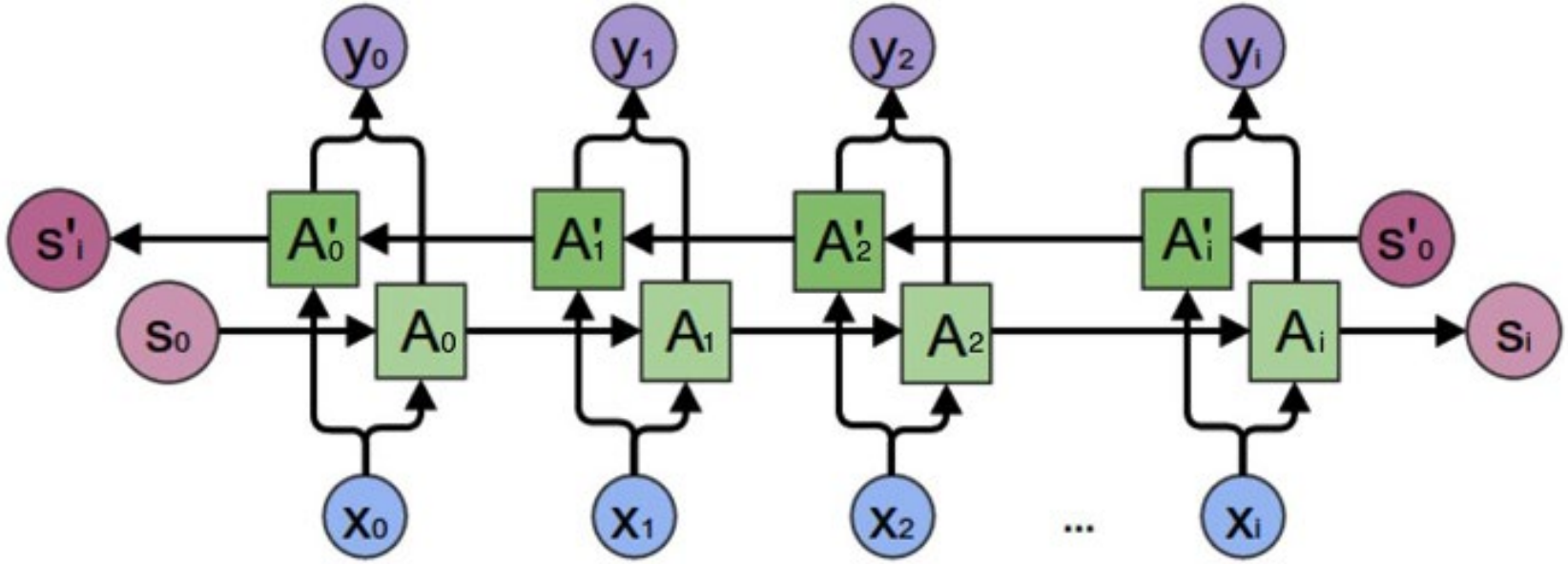
- Merging input & forget gates into a single update gate
- It also merges the cell state and hidden state
- Update gate: used in hidden state updating, controlling memory flow (close to 1: embrace new information)
- Reset gate: used in generating candidate state, adapting to new inputs (close to 0: new input more important)

Applications

Applications: Bidirectional RNNs

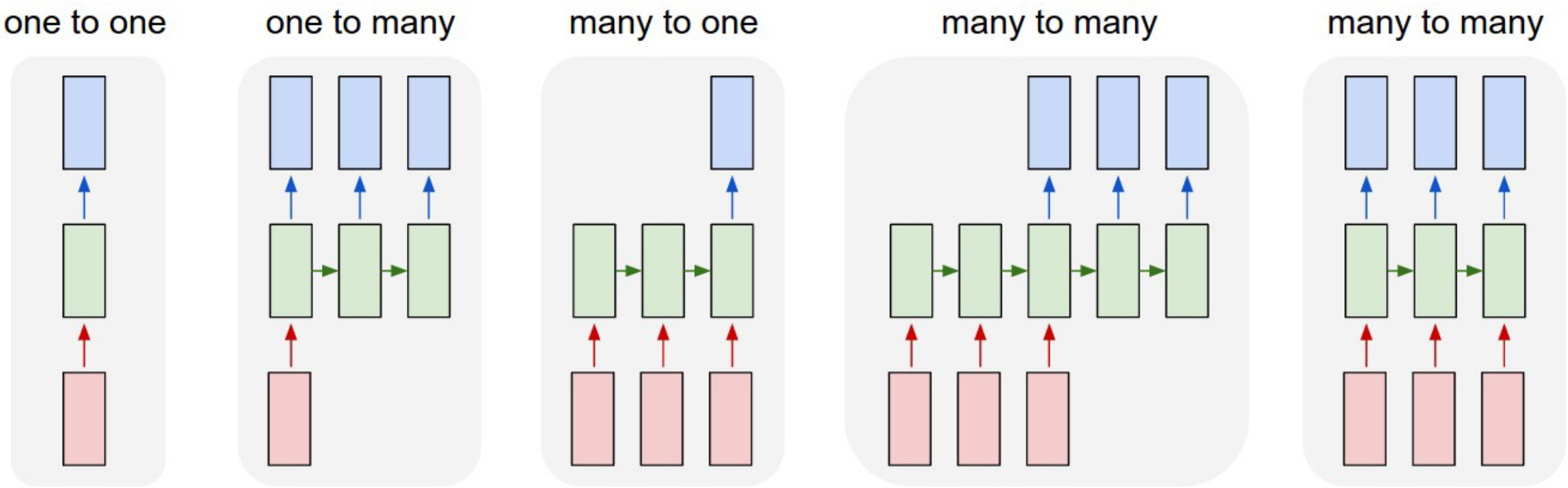
- For language models, it's often insufficient to look only at the words before a given point.
- **“My phone is broken, I’m planning to ____ a new phone.”**
- If we only look at the words before the blank — “My phone is broken” — are we planning to repair it? Replace it? Cry about it? These possibilities are all ambiguous.
- But if we also see the words after the blank — “a new phone” — then the probability of filling the blank with “buy” becomes much higher.
- In such tasks, we can improve the model’s performance by adding a network layer that passes information backward in time, enhancing the network’s ability to understand context.

Applications: Bidirectional RNNs



- A Bidirectional RNN is composed of two RNNs:
  - One processes the sequence forward (from past to future)
  - The other processes it backward (from future to past)
- Both layers receive the same input sequence
- Their outputs are usually concatenated or combined at each time step

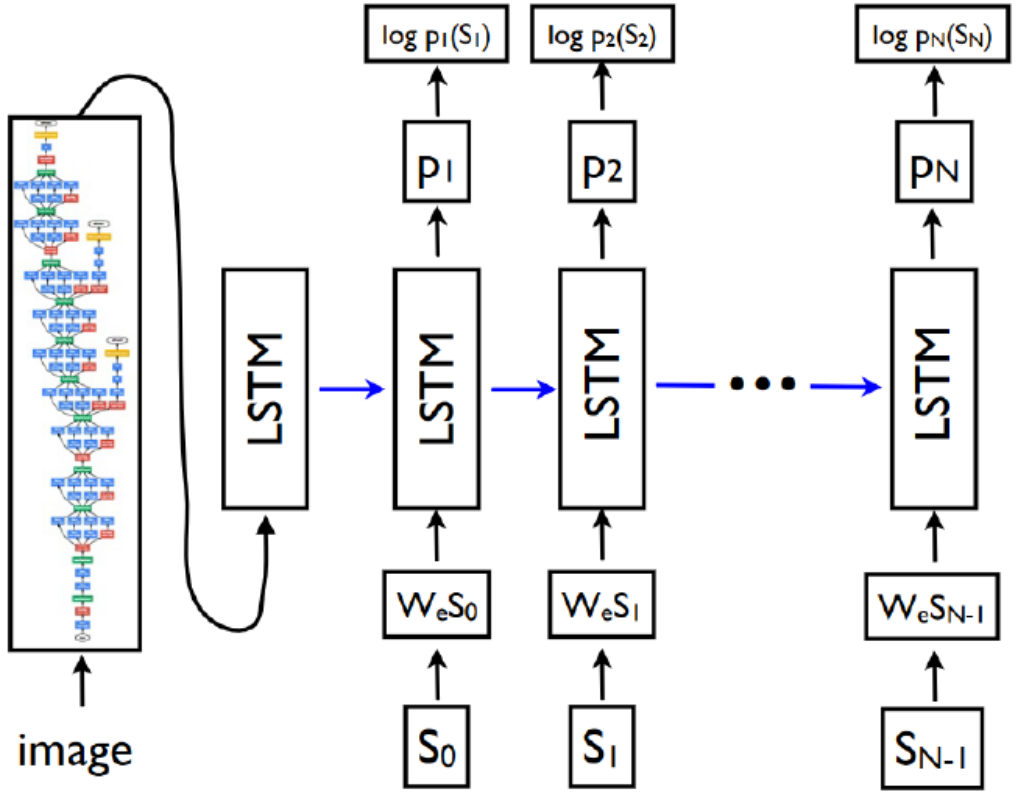
Applications



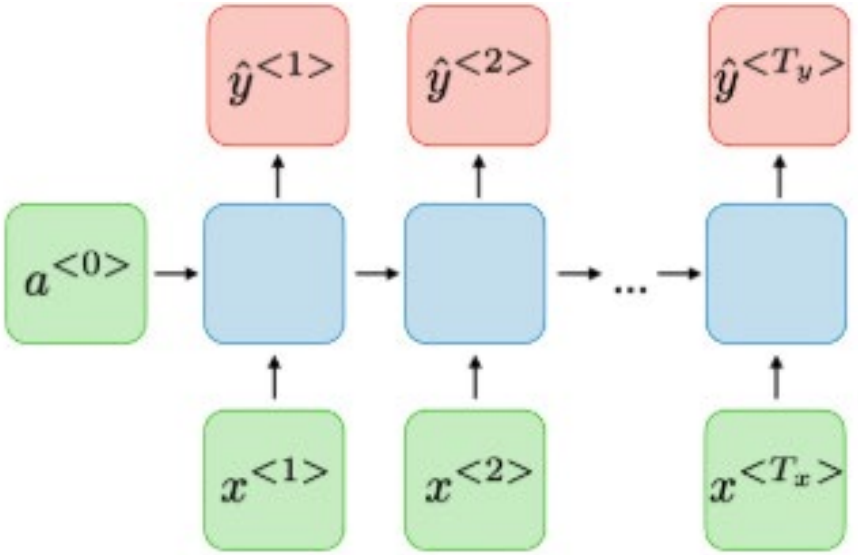
- One to one: Vanilla Neural Networks
- One to many: Image Caption (e.g., one image to a sequence of words)
- Many to one: Sequence Classification (e.g., a sentence comment review → positive/negative)

CNNs+ LSTM: Image Captioning

Show and Tell: Neural Image Caption Generator (Vinyals et al. 2015)

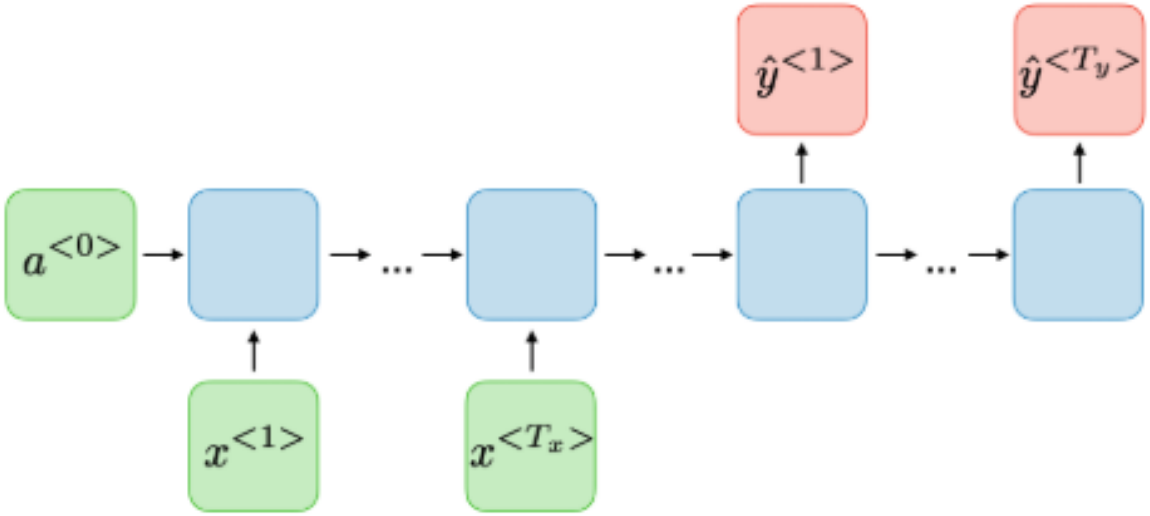


Sequence input and sequence output



- Equal Size: In this case, the input and output layer size is exactly the same.
- e.g., Video classification on frame level

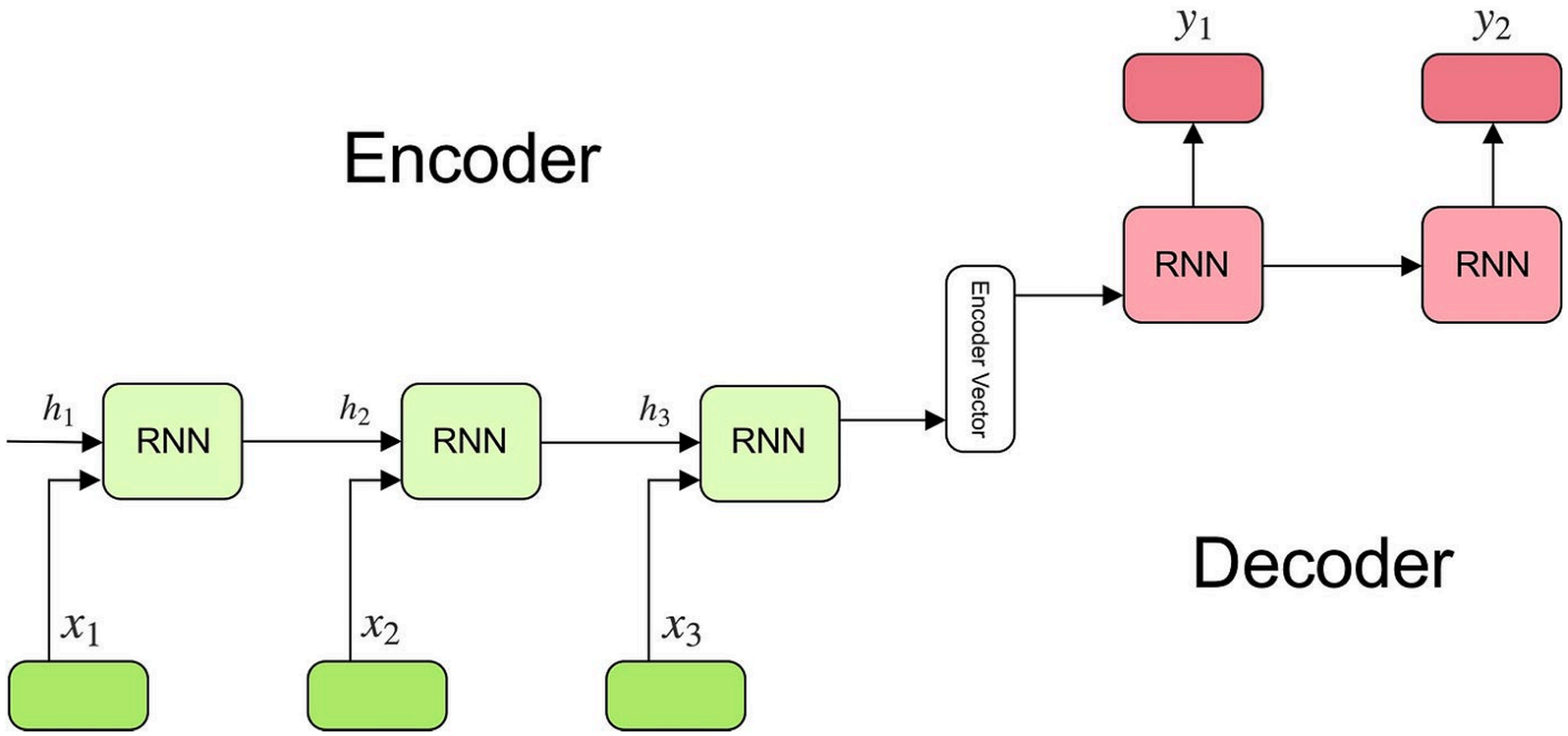
Sequence input and sequence output



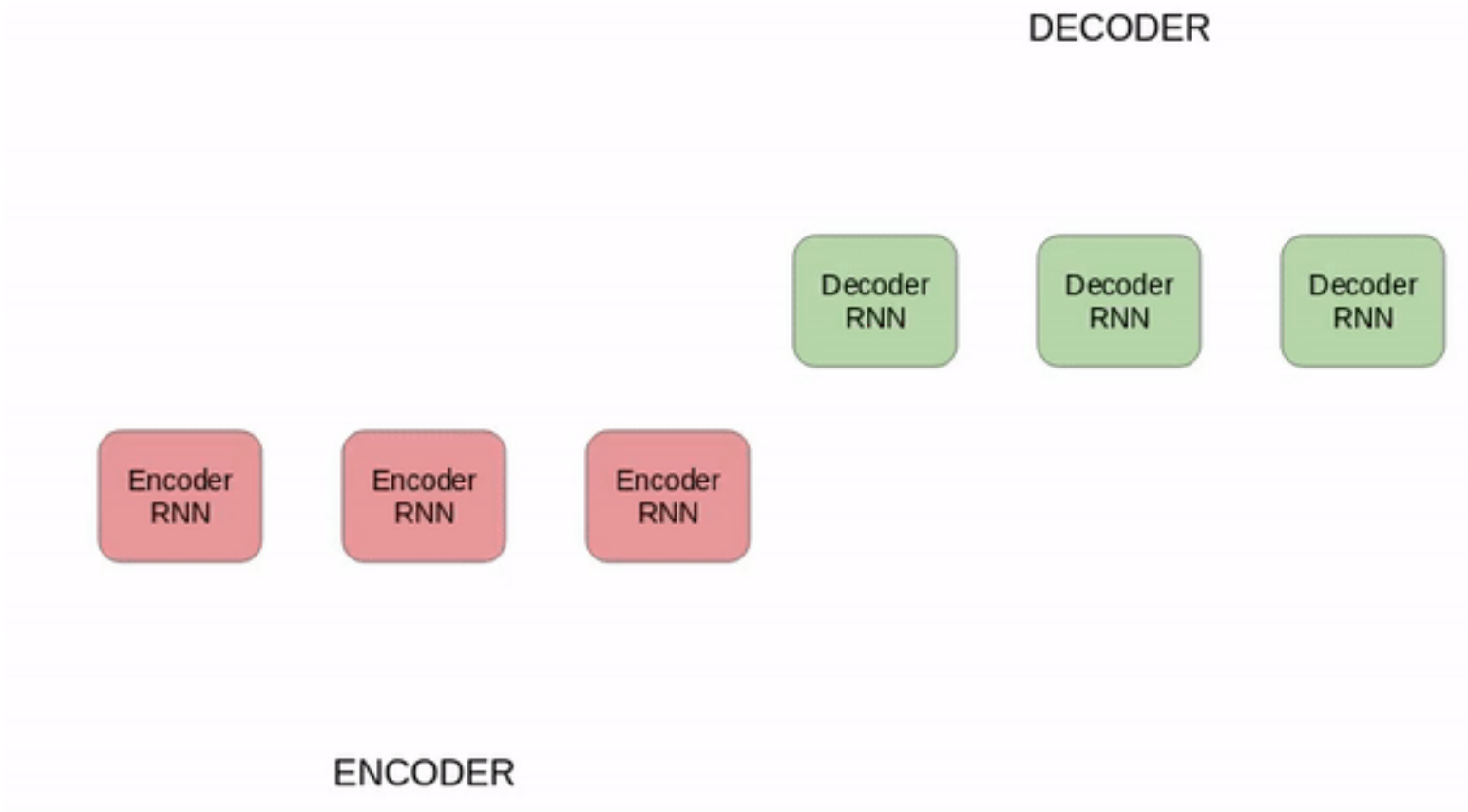
- Unequal Size: In this case, inputs and outputs have different numbers of units. Its application can be found in Machine Translation.
- Machine Translation

Seq2Seq Model

Encoder-Decoder Framework



Seq2Seq Model



Summary

- Recurrent Neural Networks (RNNs) are specialised neural networks suitable for modelling sequential or time-series data.
- RNNs have a looping mechanism that acts as a highway to allow information to flow from one step to the next. This information is the hidden state, which is a representation of previous inputs.
- Simple RNNs suffers from vanishing gradient problem
  - As the RNNs processes more steps, it has troubles retaining information from previous steps.
  - Due to back-propagation, the earlier layers fail to do any learning as the internal weights are barely being adjusted due to extremely small gradients.
  - Does not learn the long-range dependencies across time steps
- LSTMs and GRUs are two special RNNs, capable of learning long-term dependencies using mechanisms called gates.
- These gates are different tensor operations that can learn what information to add or remove to the hidden state.