Recurrent Neural Networks (RNN)

David I. Inouye

Sequential Data Is Natural in Many Applications

- Text analysis
- Speech recognition
- Medical time series
- Stock prices

Windowing Is a Simple Approach to Handle Sequential Data With Standard NNs

- Break up sequence into multiple **fixed-length** sequences:

$$split(x, y) = \{(w_j, y_j)\}_{j=1}^{\dim(x)/W}$$

  - "This is a great movie." → `{("This is", +1), ("is a", +1), ("a great", +1), ("great movie", +1)}`
  - "Hello world!" → `{("###", "H"), (“##H", "e"), (“#He", "l"), ("Hel", "l"), (“ell", "o"), ...}`
- Apply model f to each window

$$\forall (w_j, y_j) \in split(x, y), \quad \hat{y}_j = f(w_j)$$

- *Training*: Compute loss on each term or an aggregate term

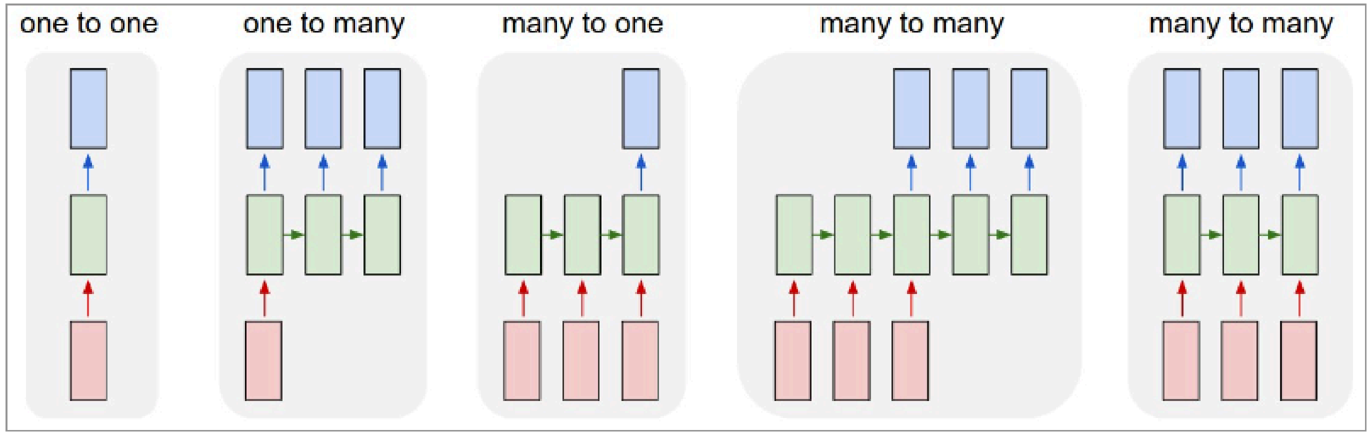
$$\sum_j l(\hat{y}_j, y_j) \quad \text{or} \quad l(\sum_j \hat{y}_j, y)$$

- *Test-time*: Concatenate or average predictions for all windows.

While a Good Baseline for Sequences, the Windowing Approach Has Several Issues

- Fixed-window size
  - How do you choose the fixed-window size?
  - If too big, computational cost is high and learning could be slow.
  - If too small, the window may lack sufficient history to predict.
- Lacks long-range dependencies (limited to window)
  - Cannot model dependencies beyond the window size.
- Predictions on each window are assumed to be independent
  - Window overlap can help as the inputs are implicitly dependent.
  - Yet the outputs are not explicitly dependent.

Recurrent Neural Networks (RNNs) Process Data **Sequentially** and Can Handle **Variable-Sized** Input/Output Sequences



Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words). **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

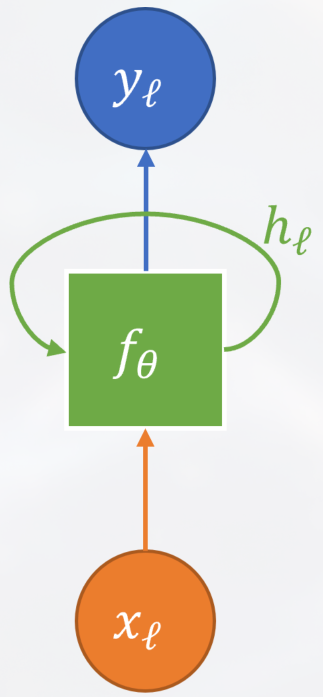
Quoted from excellent article on RNNs with examples: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

RNNs Take an Input + Old Hidden State and Produce Output + New Hidden State

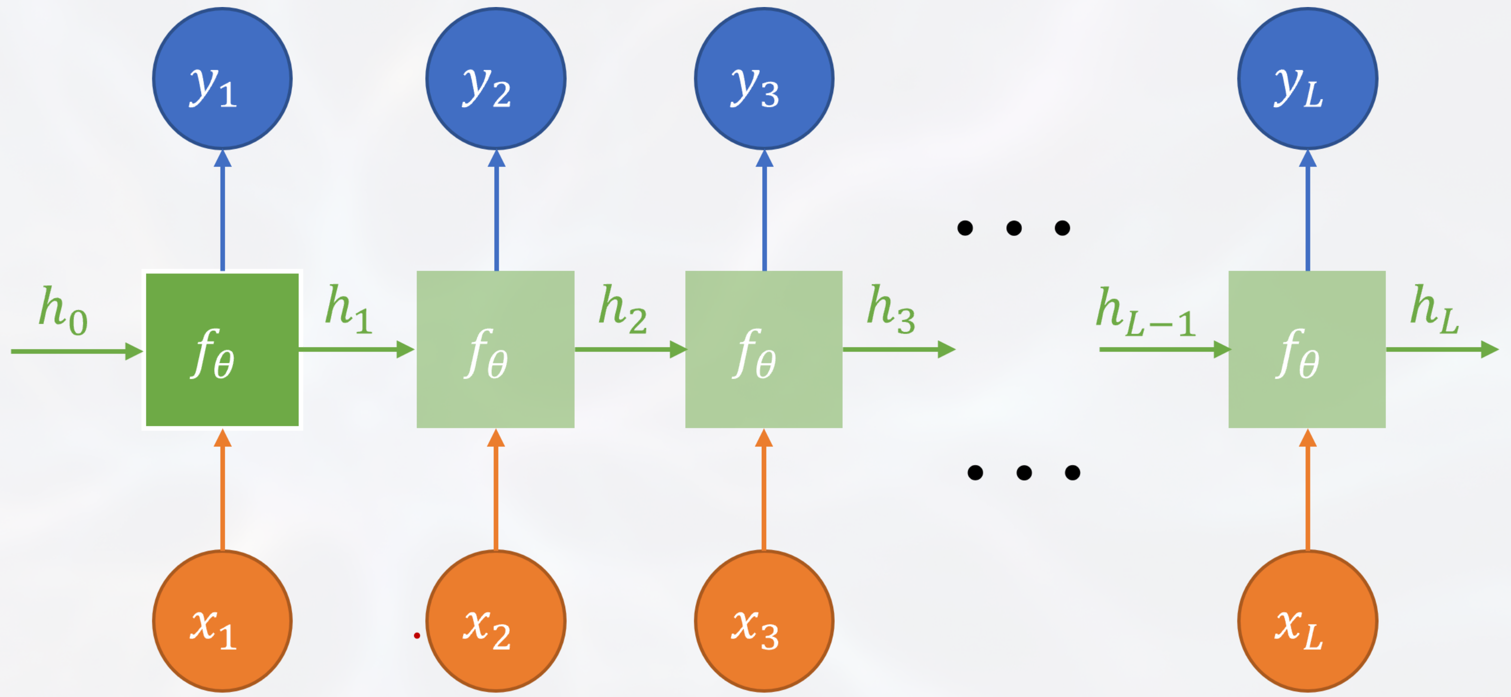
- Let x, y, and h denote the input, output, and hidden state sequences.
  - Each element in sequence could be any format including a vector, a discrete integer, or even a full tensor itself (e.g., video processing).
- Let L corresponds to length of the sequence.
  - For example, $x = (x_1, x_2, \cdots, x_l, \cdots, x_L)$
  - Note this can be different for each sample.
  - For one-to-many or many-to-one, the sequences can be padded to be the same length.
- RNN (parametrized by θ) written as recursion, where z_0 is initialized to some default value:

$$(y_l, h_l) = f_\theta(x_l, h_{l-1})$$

RNNs Can Be Visualized With Loop Arrows or Unrolled With Model Copies



Compact recursive visualization shows the RNNs simple form.

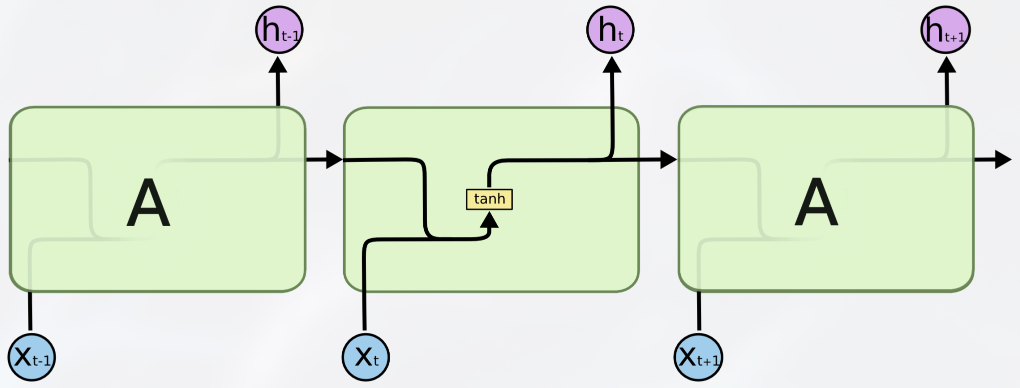


Unrolled visualization shows that the same network is used multiple times but with different inputs and different hidden states.

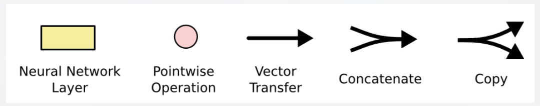
A Vanilla RNN Can Be Made With Linear and Activation Layers

- The RNN module can be written as $f_\theta(h_{l-1}, x_l) = (h_l, y_l)$
  - $h_l = \tanh(W_h h_{l-1} + W_x x_l + b_h)$
  - $y_l = W_y h_l + b_y = W_y \tanh(W_h h_{l-1} + W_x x_l + b_h) + b_y$
  - The parameters of the model are the weights and biases

$$\theta = (W_h, W_x, W_y, b_h, b_y)$$



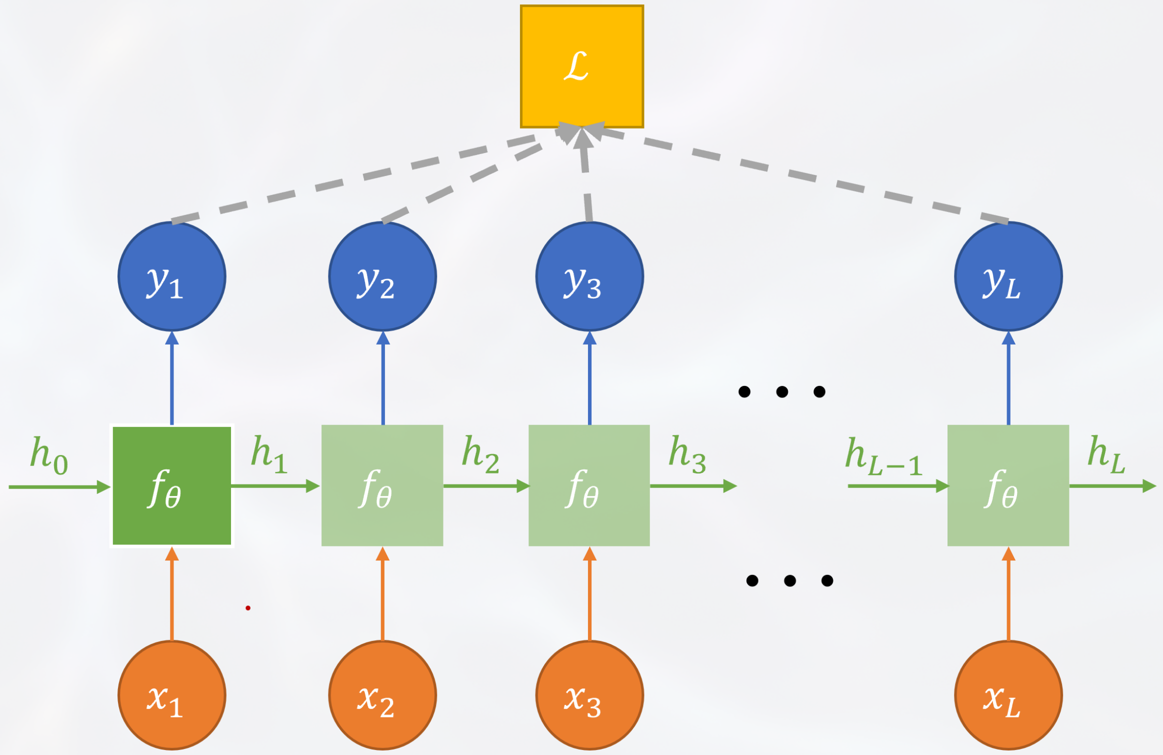
In these figures $t \equiv l$.



Images and content adapted from excellent blogpost on LSTMS: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

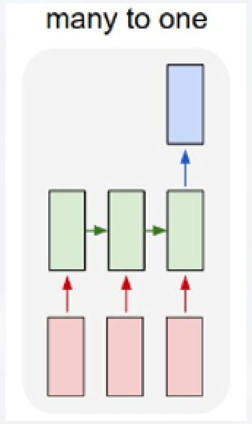
Training RNNs Can Simply Use Backpropagation Where the Whole Chain Is “Backpropagated”

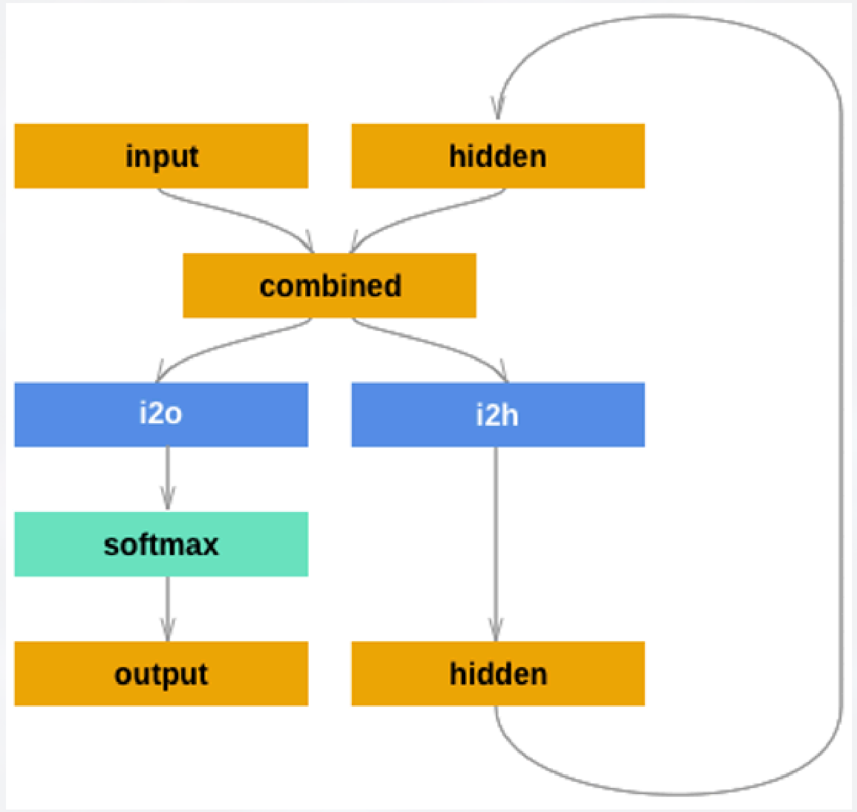
- Backpropagate gradients of the **loss function** through the computation graph.
- PyTorch’s **dynamic** computation graph enables backprop for any length sequence.
- Each RNN model evaluation f_θ is logically the same model so the gradients **accumulate**.



Demo of Simple RNNs for Sequence Classification

- Task is many to one.
- Architecture is simple.

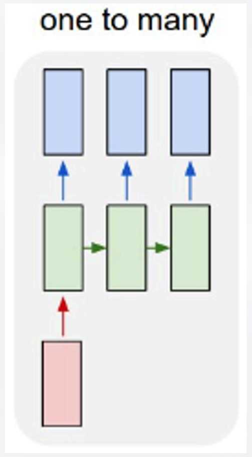


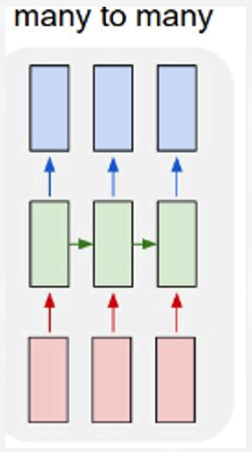


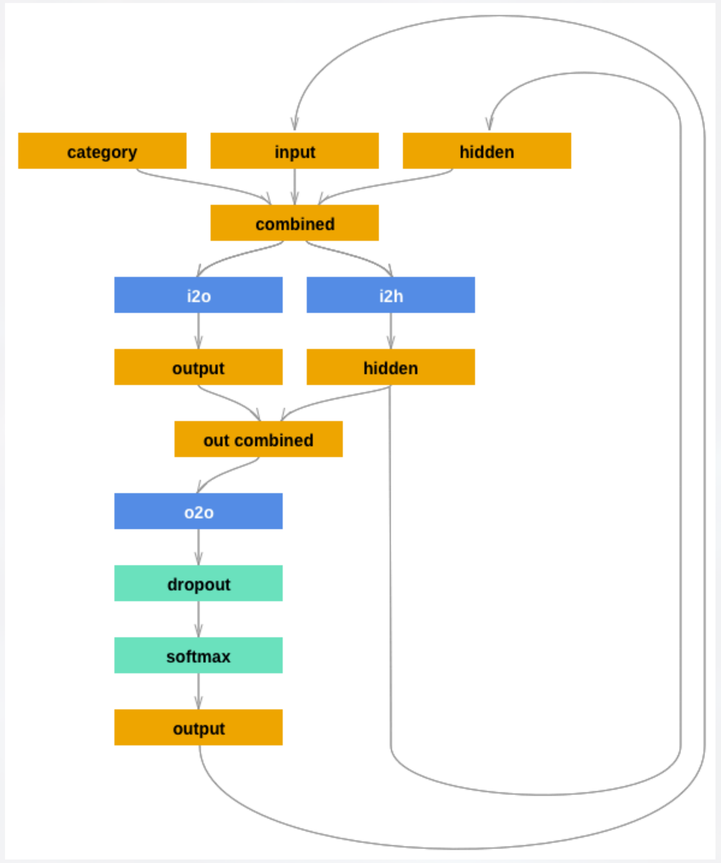
Right image from PyTorch tutorial https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html; See previous slides for credit of image on left.

Demo of Simple RNNs for Sequence Generation

- Task is conceptually one to many but can be implemented as many to many.
- Architecture is a little more complex.







Right image from PyTorch https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html Accessed on 3-28-2023; See previous slides for credit of image on left.

However, Vanilla RNNs Suffer From **Vanishing and Exploding Gradients**

Resulting in Only Learning Short-Term Dependencies

- Vanishing or exploding gradient are caused by recursive definition of hidden state.
- For simplicity, let's assume that $w_x = 0$ so that we see the core issue.
- The last prediction is as follows:

$$\hat{y}_L = w_y h_L + w_x x_L = w_y h_L = w_y(w_h h_{L-1} + w_x x_{L-1}) = w_y w_h h_{L-1} = w_y w_h^2 h_{L-2} = \ldots = w_y w_h^L h_0$$

- The gradient of MSE loss for the last term is:

$$\frac{d}{dw_y} l(y, \hat{y}_L) = \frac{d}{dw_y} ||y - \hat{y}_L||_2^2 = 2(y - \hat{y}_L)\frac{d\hat{y}_L}{dw_y} = 2(y - \hat{y}_L) w_h^L h_0$$

- If $w_h > 1.0$, then the gradient **exponentially increases** w.r.t. sequence length L.
- If $w_h < 1.0$, then the gradient **exponentially decreases** w.r.t. sequence length L.

Demo: Tiny RNN Implementation to demonstrate vanishing/exploding gradients

```python
import torch
import torch.nn as nn
class TinyRNN(nn.Module):
  def __init__(self, wh_init=1):
    super().__init__()
    self.h_init = nn.Parameter(torch.tensor(1.0))
    self.wh = nn.Parameter(torch.tensor(wh_init).float())
    self.wx = torch.tensor(0.0)
    self.wy = nn.Parameter(torch.tensor(1.0))

  def forward(self, x_seq):
    # Single RNN step
    def _single_step(x, h):
      new_h = torch.relu(self.wh * h + self.wx * x)
      y = self.wy * new_h
      return y, new_h
    h = self.h_init
    h_list = [h]
    y_list = []
    for i, x in enumerate(x_seq):
      y, h = _single_step(x, h)
      y_list.append(y)
      h_list.append(h)
    return torch.stack(y_list), torch.stack(h_list)
```

Vanishing Gradients in RNNs when wh_init < 1

```python
def show_grads(wh_init):
  criterion = torch.nn.MSELoss()
  print(f'wh_init = {wh_init}')
  for L in [1, 2, 3, 4, 5, 10]:
    rnn = TinyRNN(wh_init=wh_init)
    x_seq = torch.ones(L) # Init input sequence
    y_seq_pred, h_seq = rnn(x_seq)

    # Fake target to ensure MSE is 1
    y_true = y_seq_pred[-1].detach() - 1
    loss = criterion(y_seq_pred[-1], y_true)
    loss.backward()
    print(f'  Length={L:2d}  ' + ', '.join([f'{name}.grad = {p.grad:6.6f}' for name, p in rnn.named_parameters()]))

show_grads(wh_init=0.1)
```

```
wh_init = 0.1
  Length= 1  h_init.grad = 0.200000, wh.grad = 2.000000, wy.grad = 0.200000
  Length= 2  h_init.grad = 0.020000, wh.grad = 0.400000, wy.grad = 0.020000
  Length= 3  h_init.grad = 0.002000, wh.grad = 0.060000, wy.grad = 0.002000
  Length= 4  h_init.grad = 0.000200, wh.grad = 0.008000, wy.grad = 0.000200
  Length= 5  h_init.grad = 0.000020, wh.grad = 0.001000, wy.grad = 0.000020
  Length=10  h_init.grad = 0.000000, wh.grad = 0.000000, wy.grad = 0.000000
```

Stable Gradients in RNNs when wh_init = 1

```python
show_grads(wh_init=1)
```

```
wh_init = 1
  Length= 1  h_init.grad = 2.000000, wh.grad = 2.000000, wy.grad = 2.000000
  Length= 2  h_init.grad = 2.000000, wh.grad = 4.000000, wy.grad = 2.000000
  Length= 3  h_init.grad = 2.000000, wh.grad = 6.000000, wy.grad = 2.000000
  Length= 4  h_init.grad = 2.000000, wh.grad = 8.000000, wy.grad = 2.000000
  Length= 5  h_init.grad = 2.000000, wh.grad = 10.000000, wy.grad = 2.000000
  Length=10  h_init.grad = 2.000000, wh.grad = 20.000000, wy.grad = 2.000000
```

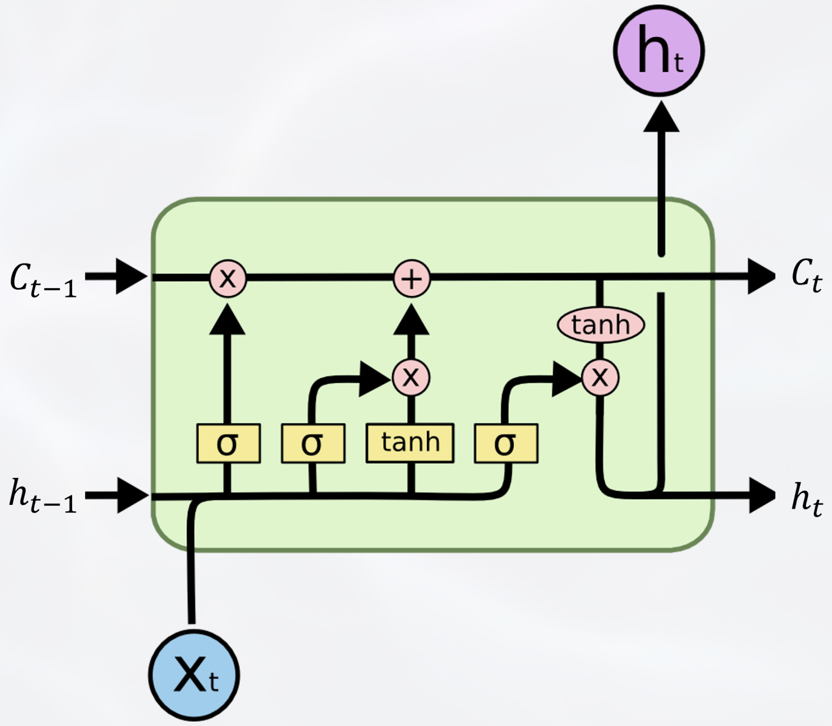
Exploding Gradients in RNNs when wh_init > 1

```python
show_grads(wh_init=10)
```

```
wh_init = 10
  Length= 1  h_init.grad = 20.000000, wh.grad = 2.000000, wy.grad = 20.000000
  Length= 2  h_init.grad = 200.000000, wh.grad = 40.000000, wy.grad = 200.000000
  Length= 3  h_init.grad = 2000.000000, wh.grad = 600.000000, wy.grad = 2000.000000
  Length= 4  h_init.grad = 20000.000000, wh.grad = 8000.000000, wy.grad = 20000.000000
  Length= 5  h_init.grad = 200000.000000, wh.grad = 100000.000000, wy.grad = 200000.000000
  Length=10  h_init.grad = 0.000000, wh.grad = 0.000000, wy.grad = 0.000000
```

Long Short-Term Memory (LSTM) Units Alleviate Vanishing Gradient Problem and Enable Learning of Long-Term Dependencies

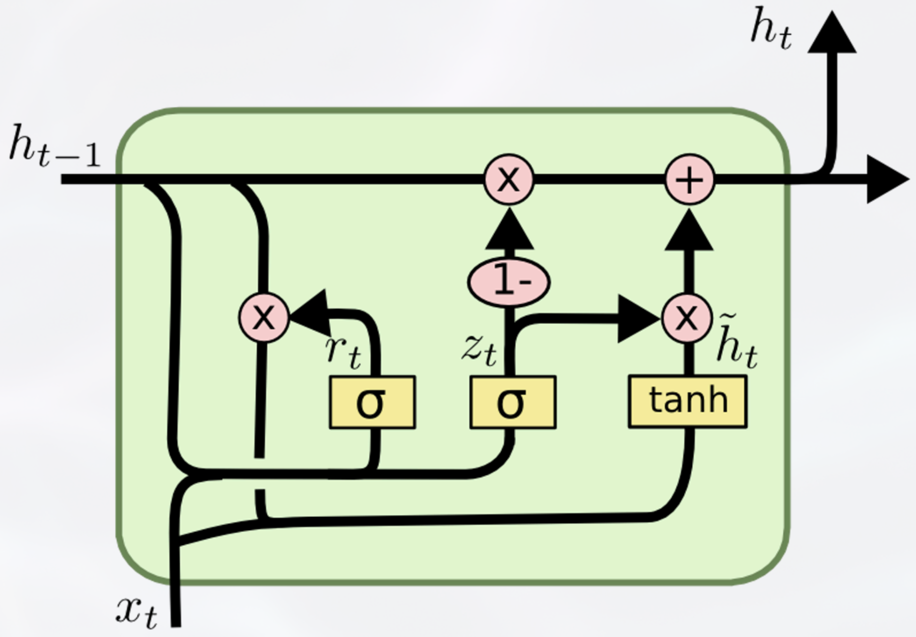
- $h'_{t-1} = [h_{t-1}, x_t]$ (concatenate)
- $\tilde{C}_t = \tanh(W_c h'_{t-1} + b_c)$ (new cell state information)
- $f_t = \sigma(W_f h'_{t-1} + b_f)$ (forget gate)
- $i_t = \sigma(W_i h'_{t-1} + b_i)$ (input gate)
- $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$ (update cell state)
- $o_t = \sigma(W_o h'_{t-1} + b_o)$ (output gate)
- $h_t = o_t \odot \tanh(C_t)$



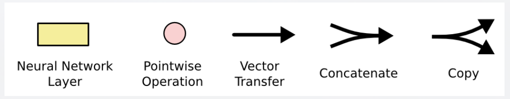
Images and content adapted from excellent blogpost on LSTMs: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Gated Recurrent Units (GRU) Simplify the LSTM Structure and Seem to Have Better Performance

- $h'_{t-1} = [h_{t-1}, x_t]$ (concatenate)
- $z_t = \sigma(W_z h'_{t-1})$ (forget/input gate)
- $r_t = \sigma(W_r h'_{t-1})$ (hidden gate)
- $\tilde{h}_t = \tanh(W[r_t \odot h_{t-1}, x_t])$ (new hidden information)
- $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$ (update hidden)

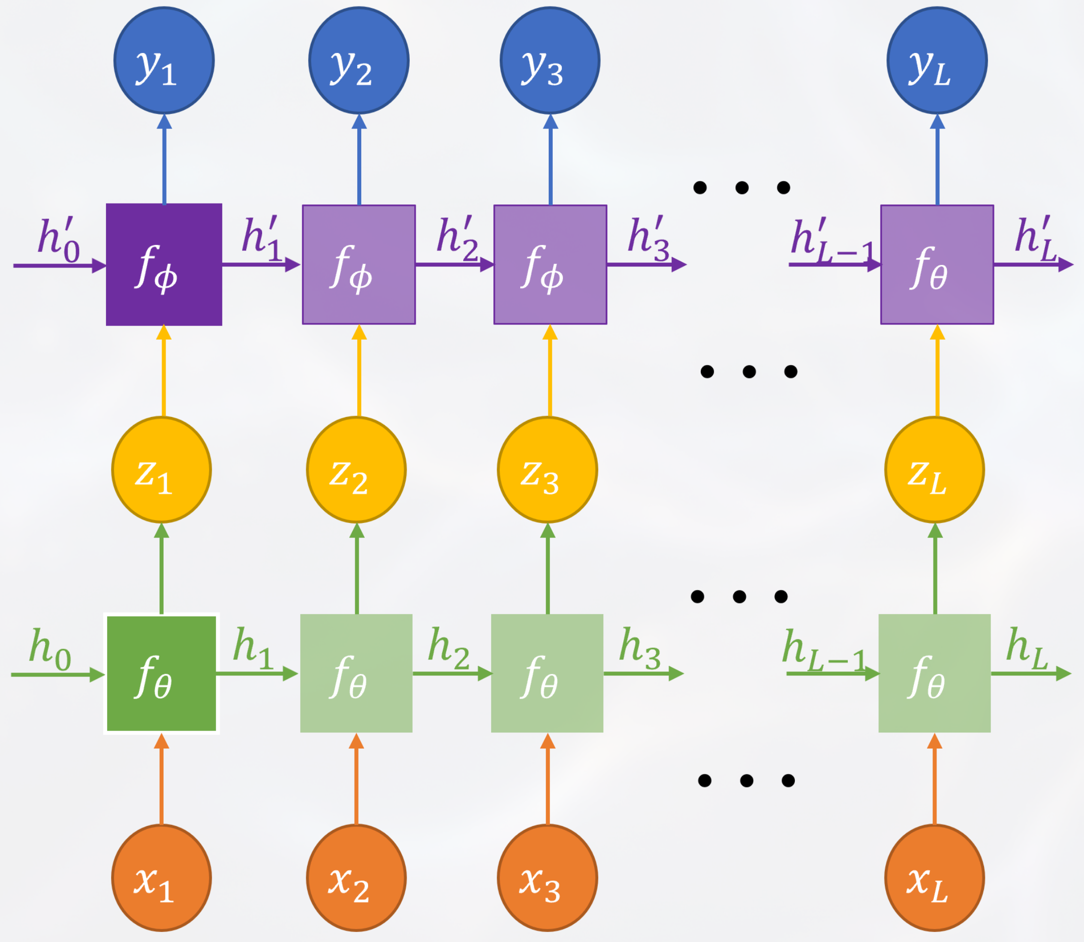


In these figures $t \equiv l$.

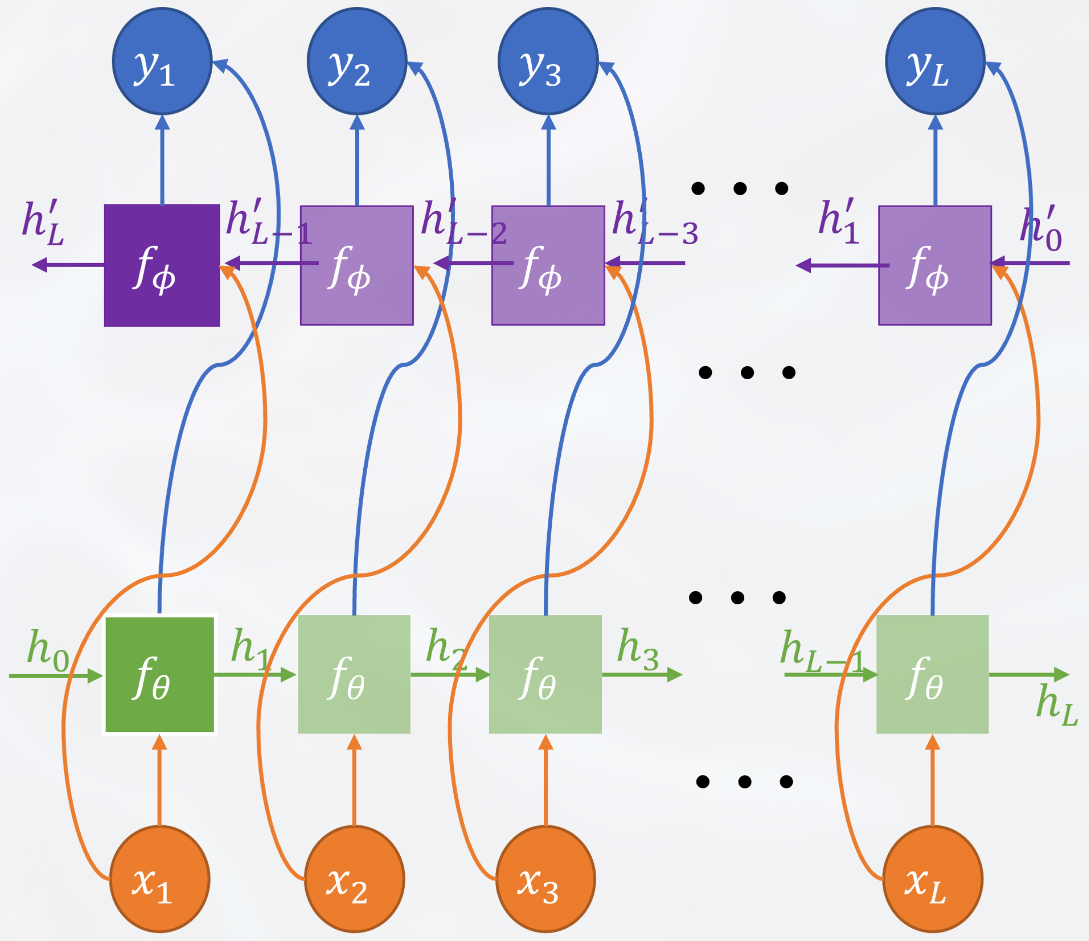


Images and content adapted from excellent blogpost on LSTMs: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

RNNs Can Be Stacked Into Deep RNNs and Even Bidirectional RNNs



Deep RNN



Bidirectional RNN

Summary of Recurrent Neural Networks

- **RNNs for Sequential Data:** RNNs are specialized for processing sequences by maintaining a hidden state that acts as a memory.
- **Limitations of Vanilla RNNs:** They struggle with long-term dependencies due to the vanishing and exploding gradient problems.
- **LSTMs and GRUs as a Solution:** Gated architectures like LSTMs and GRUs were introduced to solve this, using gates to control the flow of information and preserve long-range context.
- **Key Architectures:**
  - **LSTM:** Uses a dedicated cell state with forget, input, and output gates.
  - **GRU:** A simpler model that merges the cell state and hidden state and uses fewer gates.
- **Advanced Models:** RNNs can be stacked (Deep RNNs) or made bidirectional to process sequences in both forward and backward directions, capturing more context.