Modeling Sequential Data Using Recurrent Neural Networks

Dr. Huiping Cao

Outline

- Sequential data
- RNNs for modeling sequences
- Long short-term memory (LSTM)

Sequence data

- Elements in a sequence appear in a certain **order** and are not independent of each other.
  - Typical machine learning algorithms for supervised learning assume that the input is **independent and identically distributed** (**IID**) data.
- **Example**: predicting the market value of a particular stock
  - Each training example represents the market value of a certain stock on a particular day
  - Predict the stock market value for the next three days.
    - It makes more sense to consider the previous stock prices in a date-sorted order to derive trends rather than utilize these training examples in a randomized order

Time series data

- Time series data is a special type of sequential data where each example is associated with a dimension for time.
- **Not all sequential data has the time** dimension.
  - Text data or DNA sequences

Representation

- A sequence is represented at $< x^{(1)}, x^{(2)}, \ldots, x^{(T)} >$
  - Subscript represents the order
  - T: length

RNNs

- The standard NN models and CNNs assume that the training examples are independent of each other and thus do not incorporate *ordering information.*
  - Such models do not have a *memory* of previously seen training examples.
- RNNs are designed for modeling sequences and are capable of **remembering past information** and processing new events.

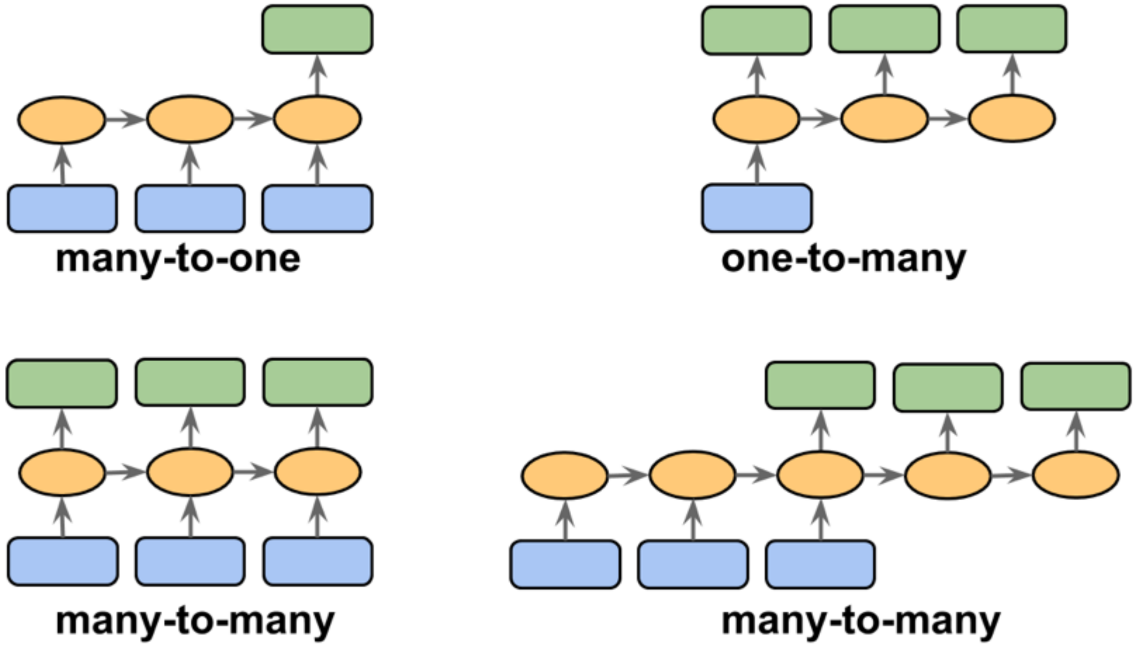
Applications

- Sequence modeling has many fascinating **applications**
  - language translation (for example, translating text from English to German)
  - Image captioning
  - Text generation
- **Three most common** sequence models based on an article*.
  - Either the input or the output data represent sequences.

* *The Unreasonable Effectiveness of Recurrent Neural Networks*, by *Andrej Karpathy*, 2015 (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

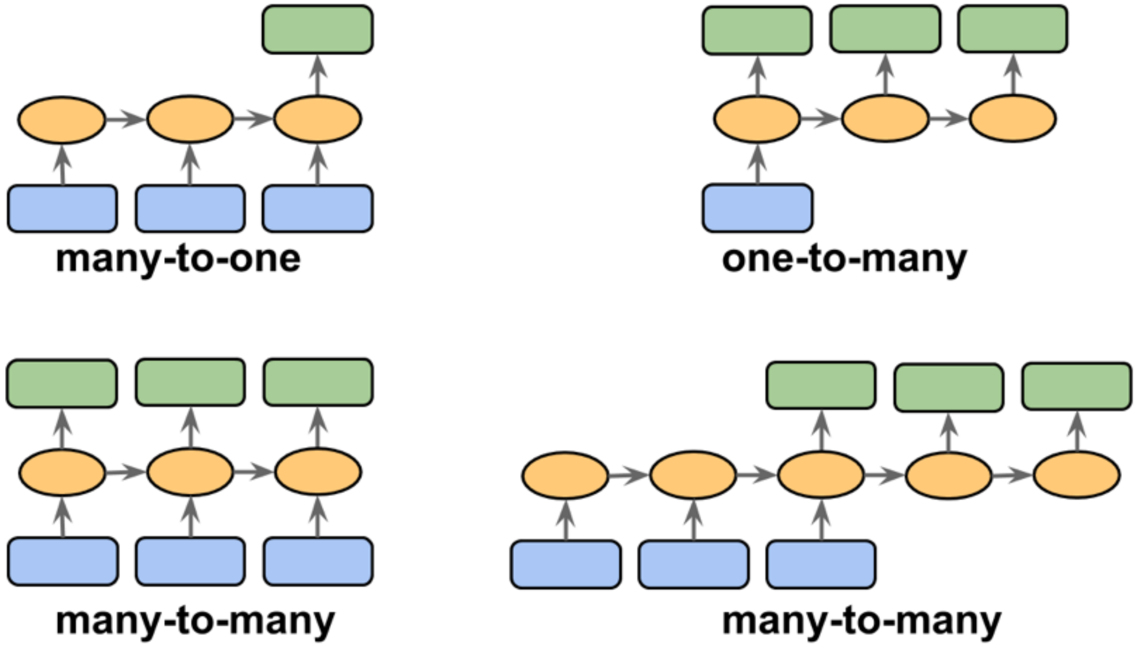
Sequence models

- **Many-to-one**: The input data is a sequence, but the output is a fixed-size vector or scalar, not a sequence.
  - Example: in sentiment analysis, the input is text-based (for example, a movie review) and the output is a class label (for example, a label denoting whether a reviewer liked the movie).
- **One-to-many**: The input data is in standard format and not a sequence, but the output is a sequence.
  - Example: image captioning—the input is an image and the output is an English phrase summarizing the content of that image.



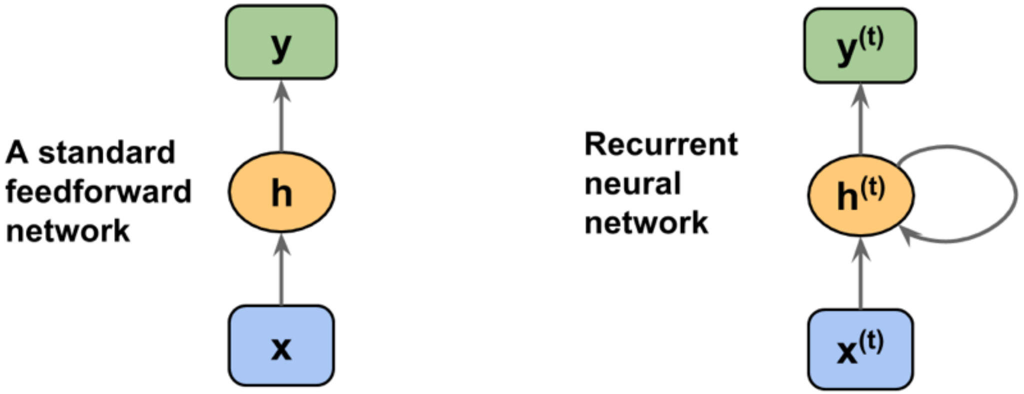
Sequence models

- **Many-to-many**: Both the input and output arrays are sequences.
  - Divided based on whether the input and output are **synchronized**.
  - **Synchronized example**: video classification, where each frame in a video is labeled.
  - ***Delayed* many-to-many modeling** task would be translating one language into another .
    - The entire English sentence must be read and processed by a machine before its translation into German is produced.



Dataflow in RNNs

- Both of these networks have only one hidden layer.
- We assume that the input layer (***x***), hidden layer (***h***), and output layer (***o***) are vectors that contain many units.
- This generic RNN architecture could correspond to the many to one and many to many sequence modeling.
  - A recurrent layer can return a sequence as output, $< o^{(1)}, o^{(2)}, \ldots, o^{(T)} >$, or simply return the last output at $t = T$, o^{T}.

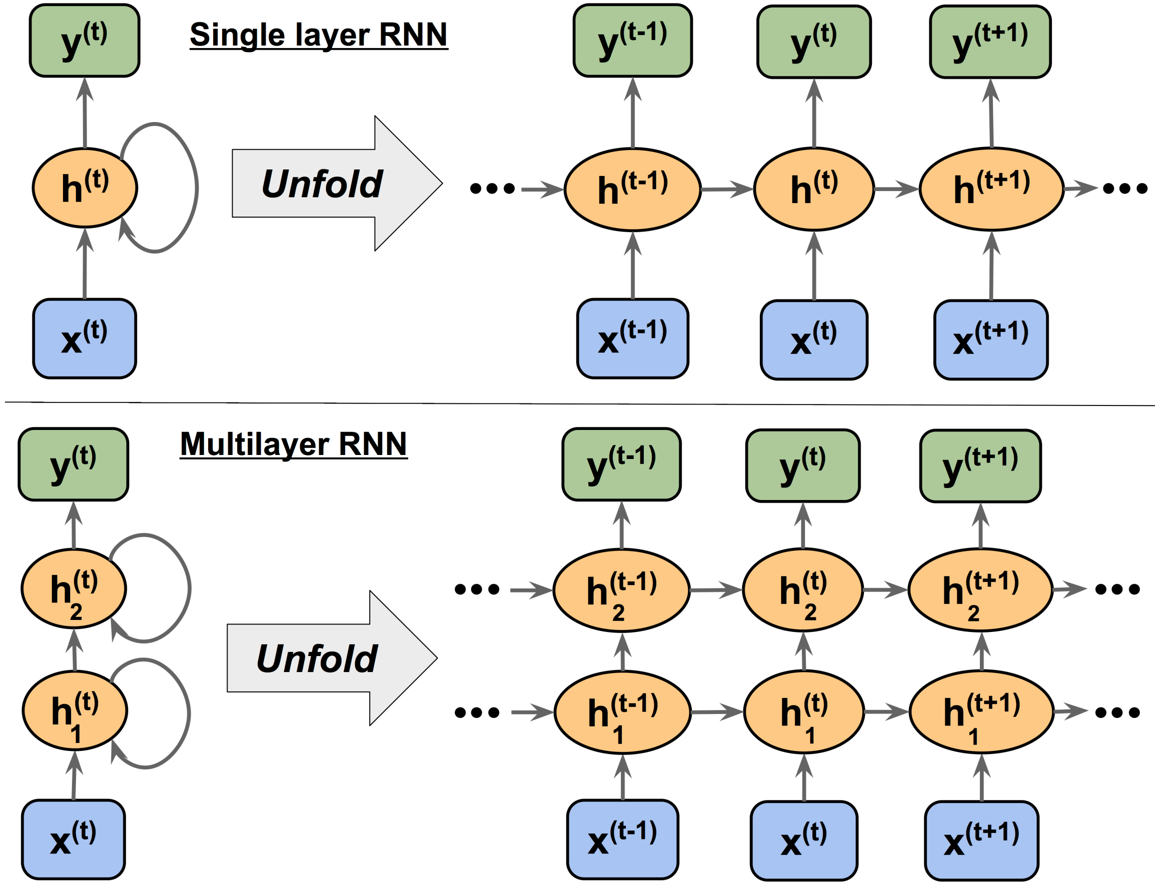


Dataflow in RNNs

- In an RNN, the hidden layer receives its input from both the input layer of the current time step and the hidden layer from the previous time step.
  - The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events.
  - This flow of information is usually displayed as a loop, also known as a **recurrent edge** in graph notation.
- In a standard feedforward network, information flows from the input to the hidden layer, and then from the hidden layer to the output layer.
- RNNs can consist of multiple hidden layers.
  - It is a common convention to refer to RNNs with one hidden layer as a *single-layer RNN,* which is different from single-layer NNs without a hidden layer

RNNs

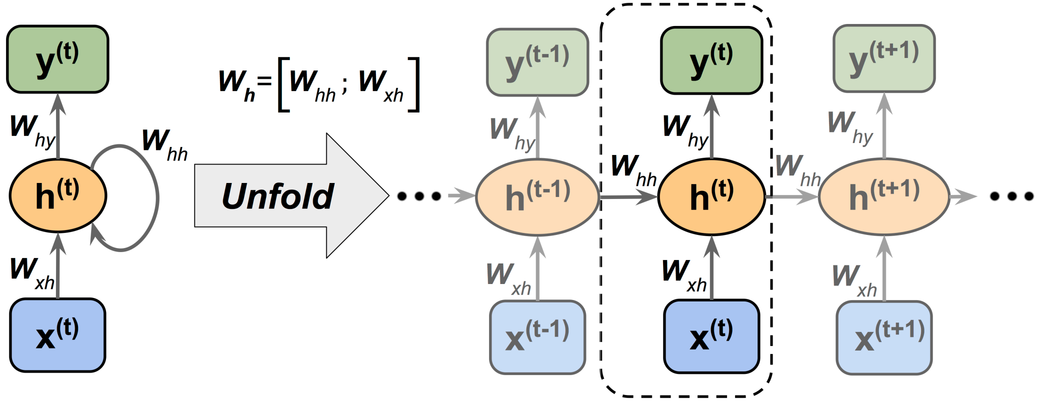
- RNN with one hidden layer (top) and an RNN with two hidden layers (bottom)



RNNs

- Each hidden unit in an RNN receives two *distinct* sets of input—the preactivation from the input layer and the activation of the same hidden layer from the previous time step.
  - Each hidden unit in a standard NN receives only one input—the net preactivation associated with the input layer.
- At the first time step, $t = 0$, the hidden units are initialized to zeros or small random values.
- At a time step where $t > 0$, the hidden units receive their input from the data point at the current time, $\boldsymbol{x}^{(t)}$, and the previous values of hidden units at $t - 1$, indicated as $\boldsymbol{h}^{(t-1)}$.

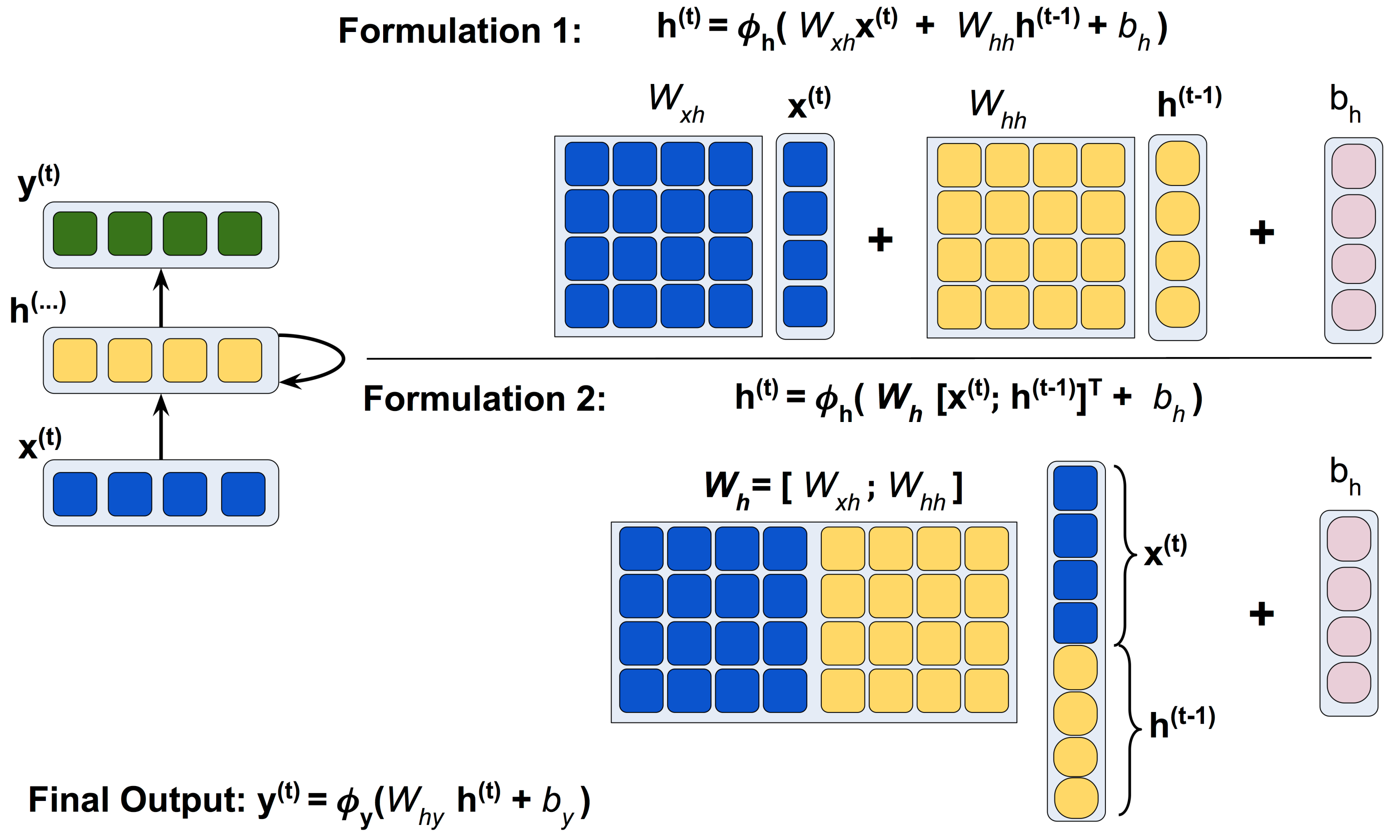
Computing activations in an RNN



- Consider just a single hidden layer.
- Each directed edge (the connections between boxes) is associated with a weight matrix.
  - These weights do not depend on time.
  - They are shared across the time axis.
- The different weight matrices in a single-layer RNN.
  - $\mathbf{W}_{xh}$: The weight matrix between the input, $\boldsymbol{x}^{(t)}$, and the hidden layer, ***h.***
  - $\mathbf{W}_{hh}$: he weight matrix associated with the recurrent edge.
  - $\mathbf{W}_{ho}$: he weight matrix between the hidden layer and output layer.

Computing activations in an RNN

- In certain implementations, you may observe that the weight matrices, $\boldsymbol{W}_{xh}$ and $\boldsymbol{W}_{hh}$, are concatenated to a combined matrix, $\boldsymbol{W}_h = [\boldsymbol{W}_{xh}; \boldsymbol{W}_{hh}]$.

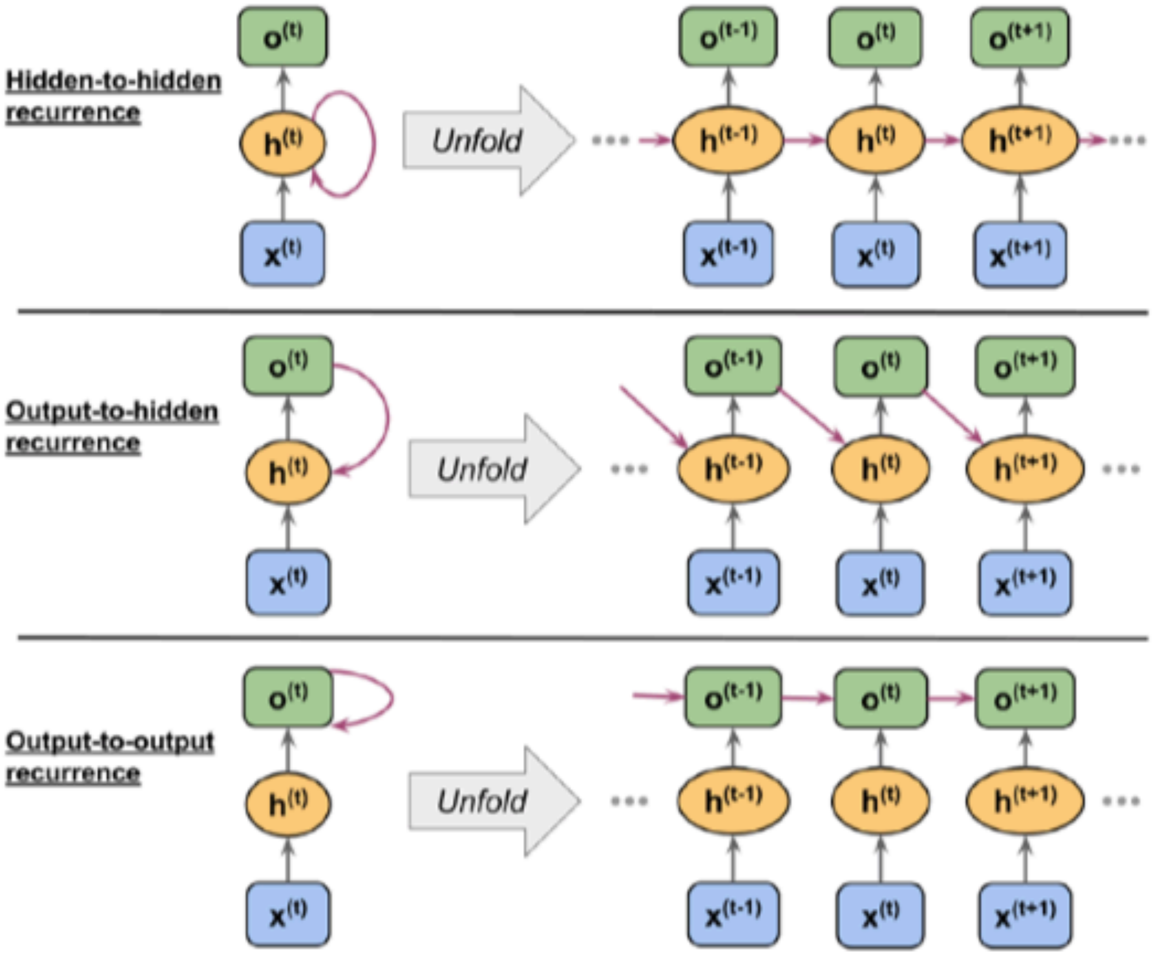


Computing activations in an RNN

- Computing the activations
  - For the hidden layer, **the net input**, $\boldsymbol{z}_h$ (preactivation), is computed through a linear combination
    - Compute the sum of the multiplications of the weight matrices with the corresponding vectors and add the bias unit. $z_h^{(t)} = W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h$
  - The **activations of the hidden units** at the time step, t, is calculated as $h^{(t)} = \sigma_h(z_h^{(t)}) = \sigma_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$
  - The activations of the output units is computed $o^{(t)} = \sigma_o(W_{ho}h^{(t)} + b_o)$

Hidden recurrence versus output recurrence

- Hidden layer has the recurrent property.
- **Alternative models**: the recurrent connection comes from the output layer



Create the layer

- torch.nn.RNN
- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
  - RNN uses one layer by default.
  - Set *num_layers* to stack multiple RNN layers together to form a stacked RNN.
- **batch_first** – If True, then the input and output tensors are provided as *(batch, seq, feature)* instead of *(seq, batch, feature)*. Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: False
  - The input shape for this layer is (batch_size, sequence_length, 5). The last dimension 5 represent the number of features.

```
torch.manual_seed(1)
rnn_layer = nn.RNN(input_size=5,  \
        hidden_size=2,num_layers=1, batch_first=True)

w_xh = rnn_layer.weight_ih_l0
w_hh = rnn_layer.weight_hh_l0
b_xh = rnn_layer.bias_ih_l0
b_hh = rnn_layer.bias_hh_l0

print('W_xh shape:', w_xh.shape)
print('W_hh shape:', w_hh.shape)
print('b_xh shape:', b_xh.shape)
print('b_hh shape:', b_hh.shape)
```

```
W_xh shape: torch.Size([2, 5])
W_hh shape: torch.Size([2, 2])
b_xh shape: torch.Size([2])
b_hh shape: torch.Size([2])
```

torch.nn.RNN

- **RNN.weight_ih_l[k]** – the learnable input-hidden weights of the k-th layer, of shape *(hidden_size, input_size)* for *k = 0*. Otherwise, the shape is *(hidden_size, num_directions * hidden_size)*
- **RNN.weight_hh_l[k]** – the learnable hidden-hidden weights of the k-th layer, of shape *(hidden_size, hidden_size)*
- **RNN.bias_ih_l[k]** – the learnable input-hidden bias of the k-th layer, of shape *(hidden_size)*
- **RNN.bias_hh_l[k]** – the learnable hidden-hidden bias of the k-th layer, of shape *(hidden_size)*

Call the forward pass on the rnn_layer

- Call the forward pass on the rnn_layer
- Manually compute the outputs at each time step and compare them
- To convert a Torch tensor with gradient to a Numpy array, first we have to detach the tensor from the current computing graph. To do it, we use the **Tensor.detach()** operation. This operation detaches the tensor from the current computational graph.
- After the detach() operation, we use **the .numpy() method** to convert it to a Numpy array.
- If a tensor with **requires_grad=True** is defined on **GPU**, then to convert this tensor to a Numpy array, we have to perform one more step.
    - First we have to move the tensor to CPU,
    - then we perform **Tensor.detach()** operation and finally use **.numpy()** method to convert it to a Numpy array.

```
x_seq = torch.tensor([[1.0]*5, [2.0]*5,
[3.0]*5]).float()
print(x_seq)

x_seq_reshaped = torch.reshape(x_seq, (1, 3, 5))
print(x_seq_reshaped)
```

```
tensor([[1., 1., 1., 1., 1.],
        [2., 2., 2., 2., 2.],
        [3., 3., 3., 3., 3.]])

tensor([[[1., 1., 1., 1., 1.],
         [2., 2., 2., 2., 2.],
         [3., 3., 3., 3., 3.]]])
```

Manually compute the outputs

```
output, hn = rnn_layer(x_seq_reshaped)
out_man = []
for t in range(3):
    xt = torch.reshape(x_seq[t], (1, 5))
    print(f'Time step {t} =>’)
    print(' Input :', xt.numpy())
    ht = torch.matmul(xt, torch.transpose(w_xh, 0, 1)) + b_xh
    print(' Hidden :', ht.detach().numpy())
    if t>0:
        prev_h = out_man[t-1]
    else:
        prev_h = torch.zeros((ht.shape))
    ot = ht + torch.matmul(prev_h, torch.transpose(w_hh, 0, 1)) + b_hh
    ot = torch.tanh(ot)
    out_man.append(ot)
    print(' Output (manual) :', ot.detach().numpy())
    print(' RNN Output :', output[:, t].detach().numpy())
    print()
```

```
Time step 0 =>
   Input : [[1. 1. 1. 1. 1.]]
   Hidden : [[-0.4701929 0.58639044]]
   Output (manual) : [[-0.3519801 0.52525216]]
   RNN Output :      [[-0.3519801 0.52525216]]
Time step 1 =>
   Input : [[2. 2. 2. 2. 2.]]
   Hidden : [[-0.88883156 1.2364398 ]]
   Output (manual) : [[-0.68424344 0.76074266]]
   RNN Output :      [[-0.68424344 0.76074266]]
Time step 2 =>
   Input : [[3. 3. 3. 3. 3.]]
   Hidden : [[-1.3074702 1.8864892]]
   Output (manual) : [[-0.8649416 0.9046636]]
   RNN Output :      [[-0.8649416 0.9046636]]
```

- The outputs from the manual forward computations **exactly match** the output of the RNN layer at each time step

The challenges of learning long-range interactions

- Because of the multiplicative factor, $\frac{\partial \boldsymbol{h}^{(t)}}{\partial h^{(k)}}$ in computing the gradients of a loss function, the so-called **vanishing** and **exploding gradient problems arise.**

- **Overall loss L** is the sum of all the loss functions at times t=1 to t=T.

$$L = \sum_{t=1}^{T} L^{(t)}$$

- The loss at time *t* is dependent on the hidden units at all previous time steps 1 : *t,* the gradient will be computed as follows.

$$\frac{\partial L^{(t)}}{\partial \boldsymbol{W}_{hh}} = \frac{\partial L^{(t)}}{\partial \boldsymbol{o}^{(t)}} \times \frac{\partial \boldsymbol{o}^{(t)}}{\partial h^{(t)}} \times \left(\sum_{k=1}^{t} \frac{\partial h^{(t)}}{\partial h^{(k)}} \times \frac{\partial h^{(k)}}{\partial \boldsymbol{W}_{hh}}\right)$$

- $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ is calculated as a multiplication of adjacent time steps

$$\frac{\partial h^{(t)}}{\partial h^{(k)}} = \prod_{i=k+1,\ldots,t} \frac{\partial h^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}$$

Vanishing and exploding gradient problems

- $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ has t-k multiplications
- A large $t - k$ refers to long-range dependencies.

- Multiplying the weight, w, by itself $t - k$ times results in a factor, w^{t-k}.
  - If $|w| < 1$, this factor becomes very small when $t - k$ is large.
  - If the weight of the recurrent edge is $|w| > 1$, then w^{t-k} becomes very large when $t - k$ is large.
- A naive solution to avoid vanishing or exploding gradients can be reached by ensuring $|w| = 1$.

Vanishing and exploding gradient problems

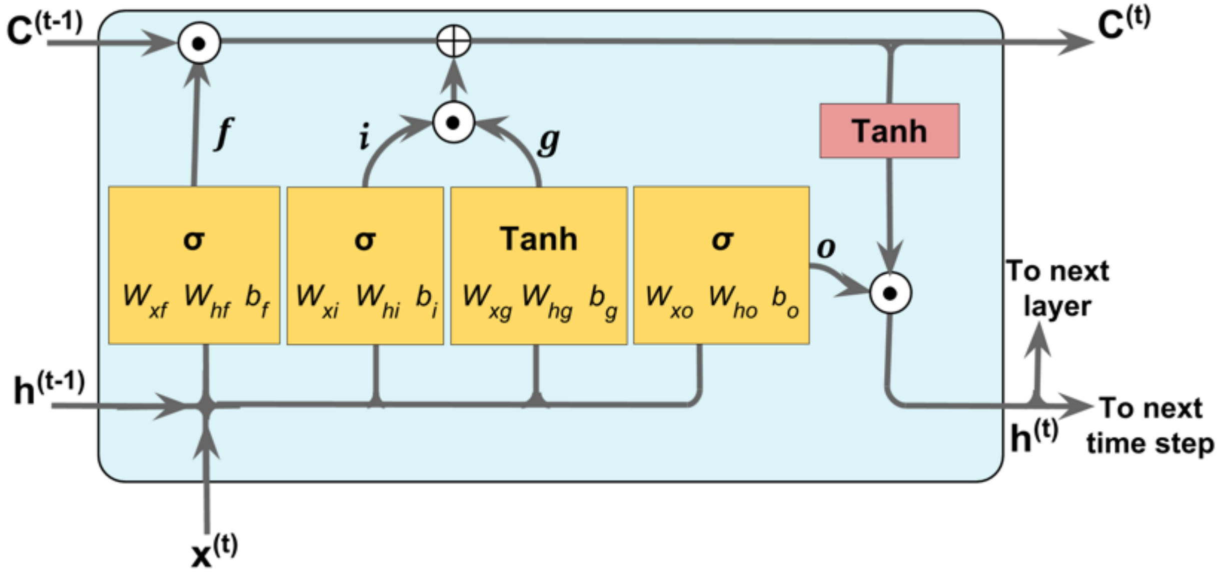
- There are at least three solutions to this problem:
  - Gradient clipping
    - We specify a cut-off or threshold value for the gradients, we assign this cut-off value to gradient values that exceed this value.
  - Truncated backpropagation through time (TBPTT)
    - Limits the number of time steps that the signal can backpropagate after each forward pass.
    - For example, even if the sequence has 100 elements or steps, we may only backpropagate the most recent 20 time steps.
  - LSTM
    - LSTM, designed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber, has been more successful in vanishing and exploding gradient problems while modeling long-range dependencies through the use of memory cells.

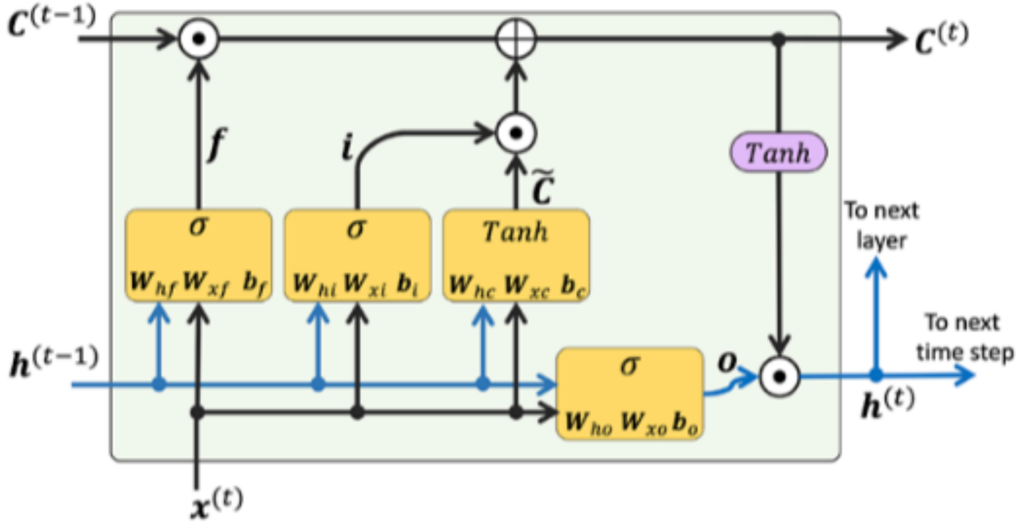
Long short-term memory cells

- LSTMs were first introduced to overcome the vanishing gradient problem.
  - *Long Short-Term Memory* by *S. Hochreiter* and *J. Schmidhuber*, *Neural Computation*, 9(8): 1735-1780, 1997.
- The building block of an LSTM is a **memory cell**, which essentially represents or replaces the hidden layer of standard RNNs.
- In each memory cell, there is a recurrent edge that has the desirable weight, $w = 1$, to overcome the vanishing and exploding gradient problems.
- The values associated with this recurrent edge are collectively called the **cell state**.

The structure of an LSTM cell

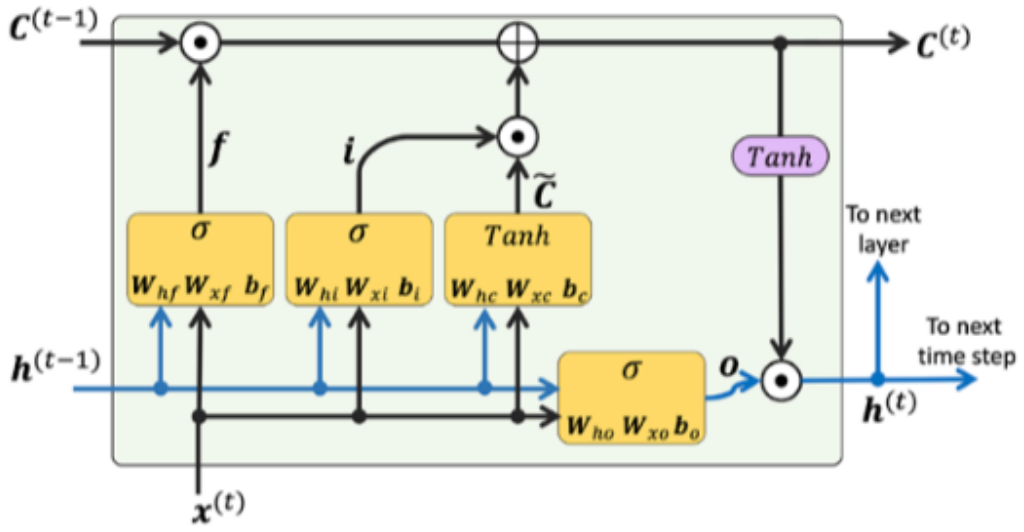
- The memory cell is responsible for keeping track of the dependencies between the elements in the input sequence.
- The cell state from the previous time step, $\boldsymbol{C}^{(t-1)}$, is modified to get the cell state at the current time step, $\boldsymbol{C}^{(t)}$.
- The flow of information in this memory cell is controlled by several computation units (often called **gates**).



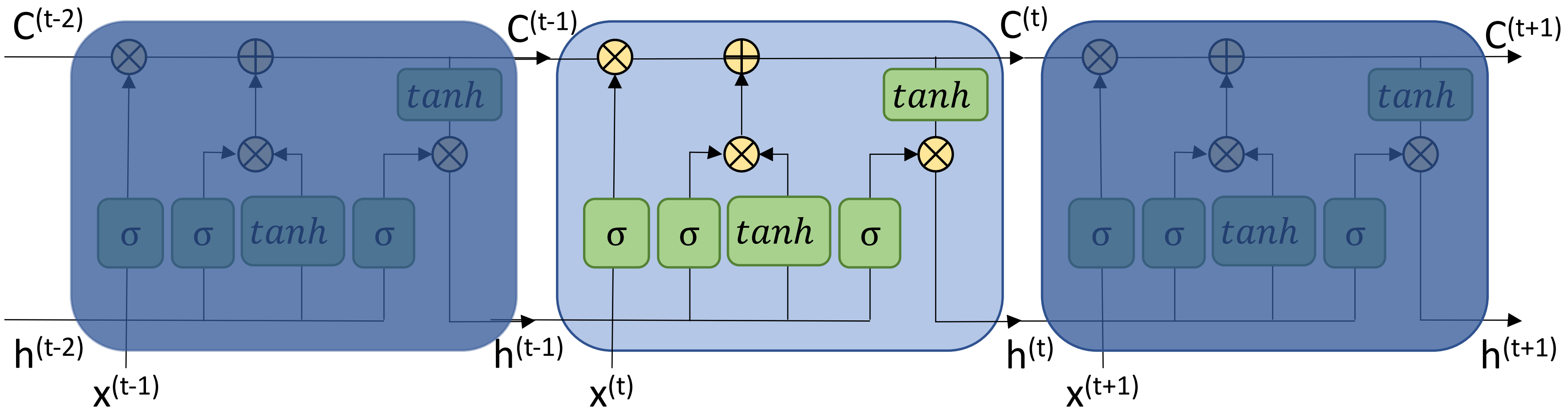


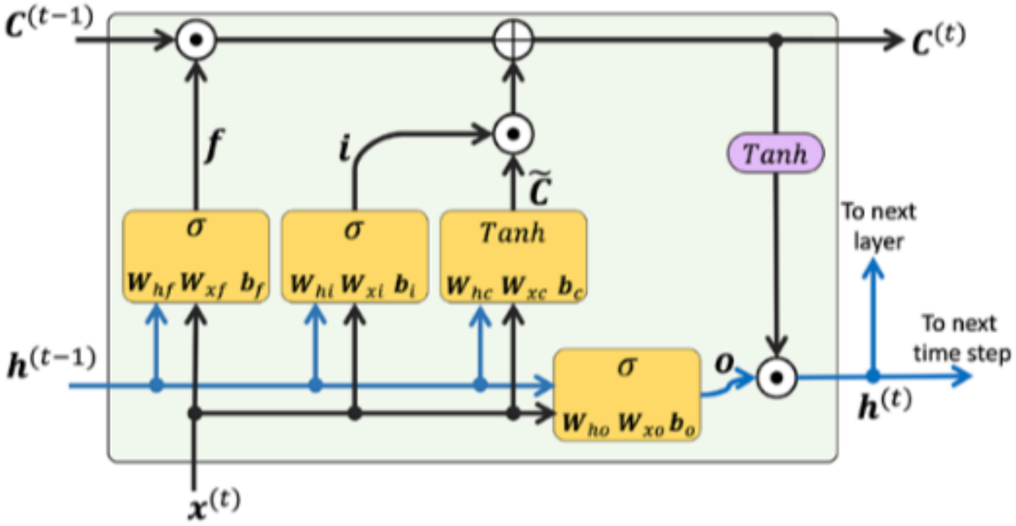
The structure of an LSTM cell

- $\oplus$: element-wise summation
- $\odot$: element-wise product
- $x^{(t)}$: input data at time t
- $h^{(t-1)}$: hidden units at time t-1
- The four boxes:
  - Activation functions: σ or tanh
  - Apply a linear combination by performing matrix-vector multiplications on their input



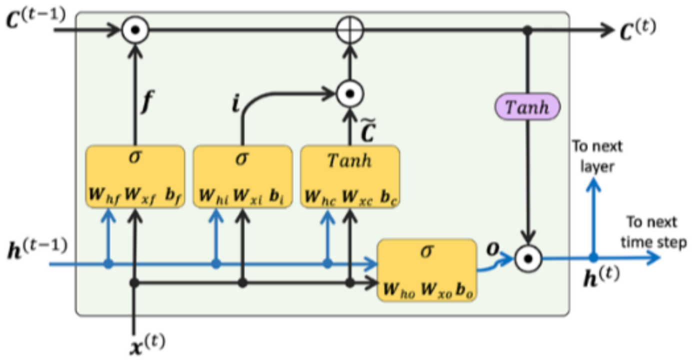
LSTM cells





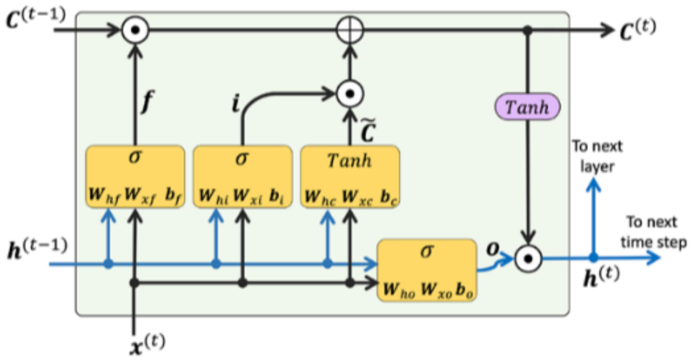
- Different symbol: $\odot = \otimes$

LSTM cell



- In an LSTM cell, there are three different types of gates, which are known as the forget gate, the input gate, and the output gate.
- The **forget gate** ($\boldsymbol{f}_t$) allows the memory cell to reset the cell state without growing indefinitely.
  - The forget gate decides which information is allowed to go through and which information to suppress.
  - $\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{xf}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hf}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f)$
  - The forget gate was not part of the original LSTM cell; it was added a few years later to im- prove the original model.
    - *Learning to Forget: Continual Prediction with LSTM* by *F. Gers*, *J. Schmidhuber*, and *F. Cummins*, *Neural Computation 12*, 2451-2471, 2000).

LSTM cell



- The **input gate** ($\boldsymbol{i}_t$) and **candidate value** ($\tilde{C}_t$) are responsible for updating the cell state. They are computed as follows.
  - $\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_{xi}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i\right)$
  - $\widetilde{\boldsymbol{C}}_t = tanh\left(\boldsymbol{W}_{xc}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hc}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_c\right)$
- The **cell state** at time t is computed as
  - $\boldsymbol{C}^{(t)} = (\boldsymbol{C}^{(t-1)} \odot \boldsymbol{f}_t) \oplus (\boldsymbol{i}_t \odot \widetilde{\boldsymbol{C}}_t)$
- The **output gate** $\boldsymbol{o}_t$ decides how to update the values of hidden units
  - $\boldsymbol{o}^{(t)} = \sigma(\boldsymbol{W}_{xo}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_{\boldsymbol{o}})$
- The **hidden units** at the current time step is computed as
  - $\boldsymbol{h}^{(t)} = \boldsymbol{o}_t \odot \tanh(\boldsymbol{C}^{(t)})$

Other advanced RNN models

- LSTMs provide a basic approach for modeling long-range dependencies in sequences
- There are many variations of LSTMs.
- A more recent approach, **gated recurrent unit** (**GRU**), which was proposed in 2014.
  - GRUs have a simpler architecture than LSTMs; therefore, they are computationally more efficient.
  - While their performance in some tasks, such as polyphonic music modeling, is comparable to LSTMs.

References

- Chapter 15: By Sebastian Raschka , Yuxi (Hayden) Liu , Vahid Mirjalili: Machine Learning with PyTorch and Scikit-Learn, Packt.
- PyTorch RNN: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html