Deep Learning for COMP6714 – Part II

Wei Wang @ CSE, UNSW

October 8, 2018

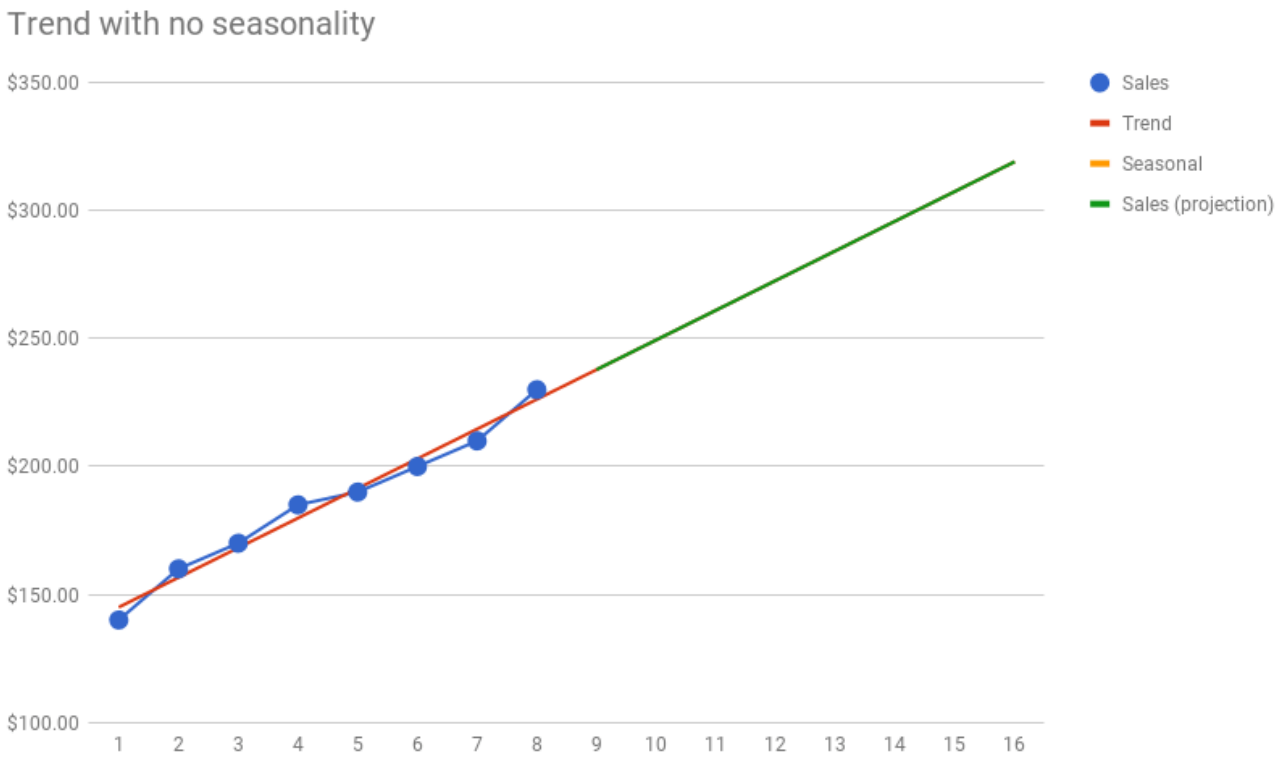
Outline

- RNN
- LSTM

RNN

- Goal: How to model sequence?
  - e.g., $\mathbf{y} = f([\mathbf{x}_1, \mathbf{x}_2, \ldots])$
- Challenge: How to deal with variable input length?

Example: Time Series Data



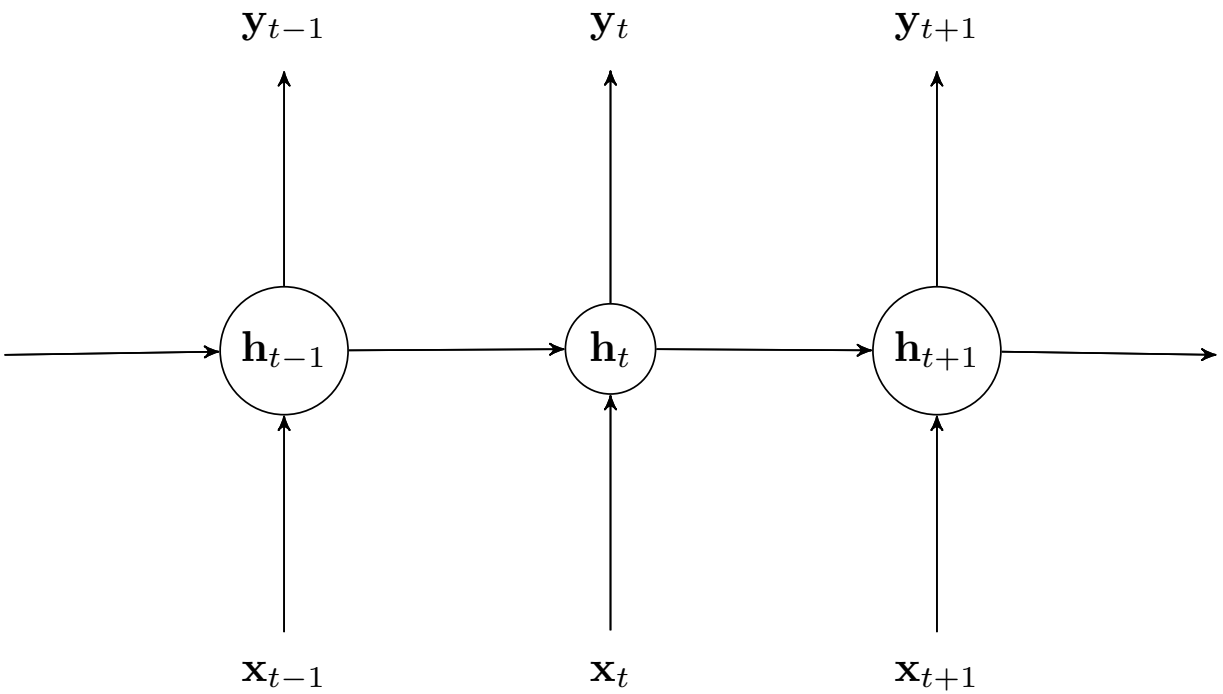
$y_t = ?$

- What should it depdend on?

How to accept variable-length input?
Proposals:

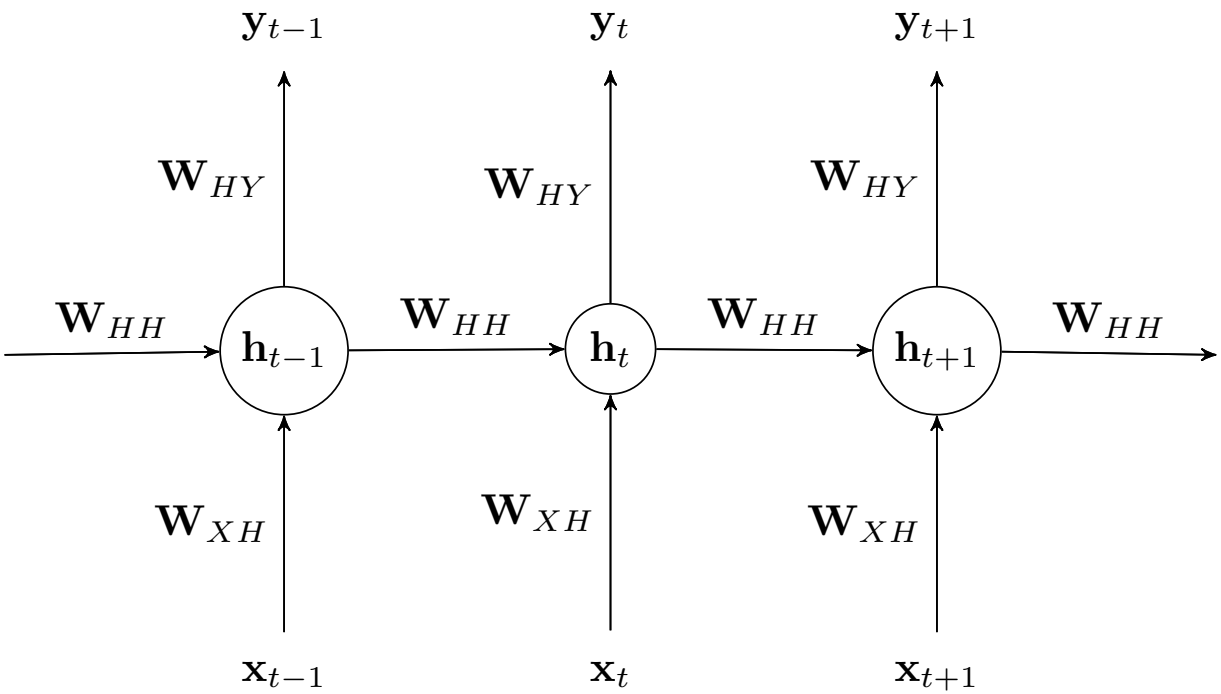
- Naïve 1: $yy_i = f_i(\mathbf{x}_i)$?
- Naïve 2: $yy_i = f(\mathbf{x}_i)$?
  - Consider `The dog bit Harry` vs `Harry bit the dog`.
- Recursive (aka. Jordan): $yy_i = f(\mathbf{y}_{i-1}, \mathbf{x}_i)$?
  - Allow $\mathbf{y}_j$ to have different shape than that of $\mathbf{x}_i$:
  - Allow $\mathbf{y}_j$ to have different semantics than that of $\mathbf{x}_i$:
  - Allow high dimensionality of $\mathbf{y}_j$.
- Recursive (aka. Elman): Use $\mathbf{h}_i$ as latent variables:
  - $\mathbf{h}_i = f(\mathbf{h}_{i-1}, \mathbf{x}_i)$
  - $\mathbf{y}_i = g(\mathbf{h}_i)$
  - Both f and g are affine function of the inputs.

RNN Diagram



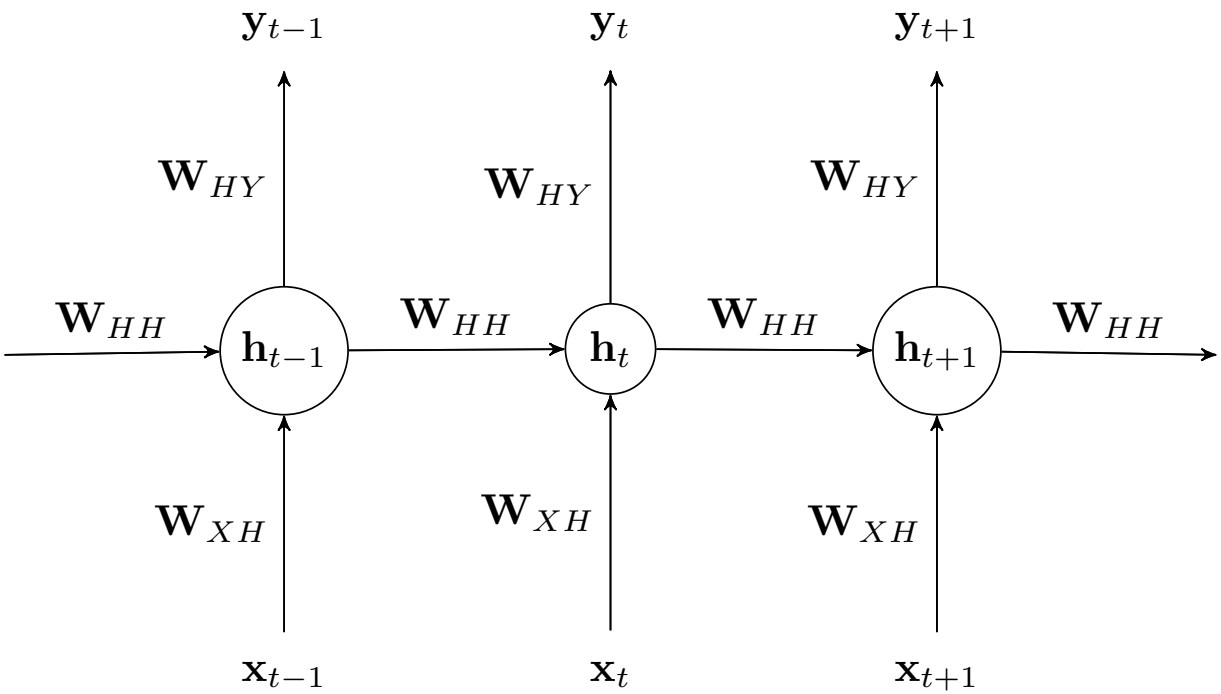
- Note: parameters are shared.

RNN Diagram



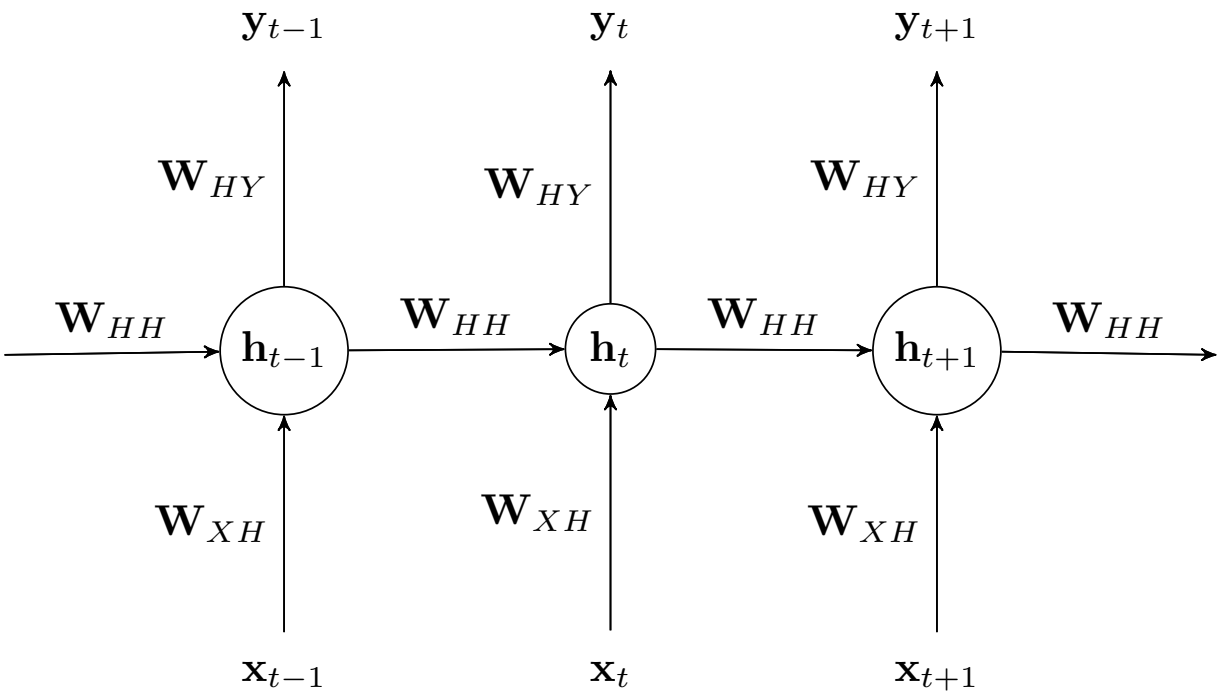
- $\mathbf{y}_t = f(\mathbf{h}_t)$
- $\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_t)$

RNN Diagram



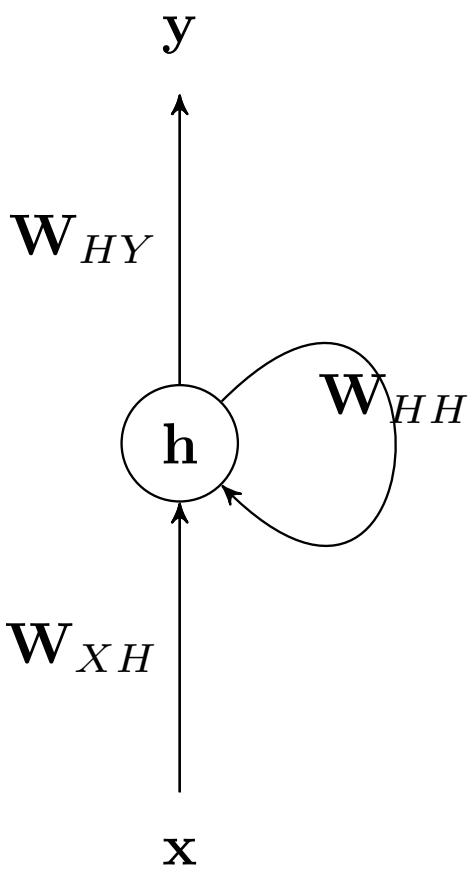
- $\mathbf{y}_t = \sigma(\mathbf{W}_{HY}\mathbf{h}_t + \mathbf{b}_Y)$
- $\mathbf{h}_t = \sigma(\mathbf{W}_{HH}\mathbf{h}_{t-1} + \mathbf{W}_{XH}\mathbf{x}_t + \mathbf{b}_H)$, or
  $\mathbf{h}_t = \sigma(\mathbf{W}\,[\mathbf{h}_{t-1}\,;\,\mathbf{x}_t] + \mathbf{b}_H)$

RNN Diagram



- $\mathbf{y}_t = f(\mathbf{h}_t)$
- $\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_t)$

RNN Diagram



- $\mathbf{y}_t = \sigma(\mathbf{W}_{HY}\mathbf{h}_t + \mathbf{b}_Y)$
- $\mathbf{h}_t = \sigma(\mathbf{W}_{HH}\mathbf{h}_{t-1} + \mathbf{W}_{XH}\mathbf{x}_t + \mathbf{b}_H)$

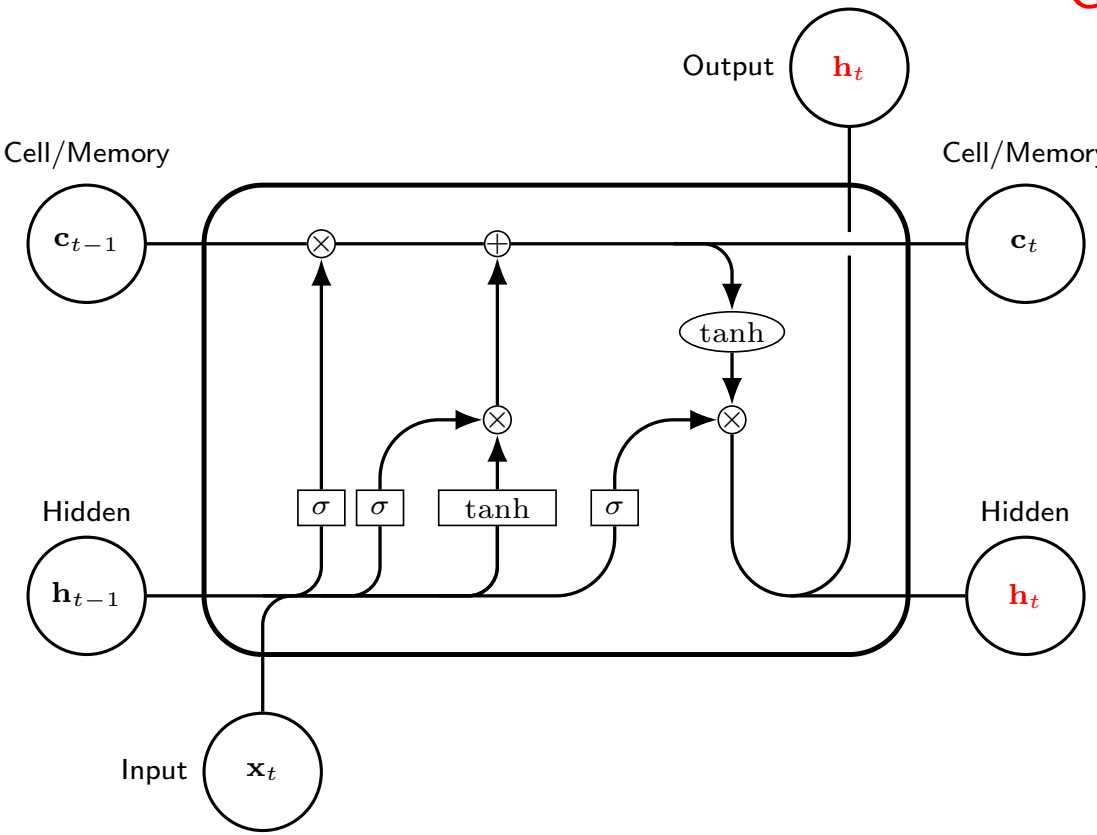
Additional Notes about RNNs

- $\mathbf{y}_t$ is determined by $\mathbf{h}_t$.
  - Better to think of the output of an RNN as a sequence of **h**s.
- About $\mathbf{h}_i$:
  - Better to think of it as some learned feature that **summarizes** $[\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_i]$

Problems with RNN

- Gradient vanishing / exploding problems
  - $\delta^{(l)} = \left((\mathbf{W}^{(l)})^\top \delta^{(l+1)}\right) * f'(z^{(l)})$
  - Multiplying the same matrix again and again can result in very small values (gradient vanishing) or very large values (gradient exploding)
- Can only capture very limited history in practice.

LSTM (Long Short-term Memory)



Gates:

- output gate
  $\mathbf{o}_t = f_o(\mathbf{x}_t, \mathbf{h}_{t-1})$
- input gate
  $\mathbf{i}_t = f_i(\mathbf{x}_t, \mathbf{h}_{t-1})$
- forget gate
  $\mathbf{f}_t = f_f(\mathbf{x}_t, \mathbf{h}_{t-1})$

Input transformation:

- $\mathbf{c}_t^{\text{new}} = g(\mathbf{x}_t, \mathbf{h}_{t-1})$

State update:

- $\mathbf{c}_t = \mathbf{f}_t * \mathbf{c}_{t-1} + \mathbf{i}_t * \mathbf{c}_t^{\text{new}}$
- $\mathbf{h}_t = \mathbf{o}_t * \tanh(\mathbf{c}_t)$

Activation functions:

- Three σ : to implement the "gates".
- Two tanh: allow both postive and negative values.

Another form:

$$\begin{bmatrix} \mathbf{i}_t \\ \mathbf{f}_t \\ \mathbf{o}_t \\ \mathbf{c}_t^{\text{new}} \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} \left(\mathbf{W} \begin{bmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \end{bmatrix} \right)$$

State update:

- $\mathbf{c}_t = \mathbf{f}_t * \mathbf{c}_{t-1} + \mathbf{i}_t * \mathbf{c}_t^{\text{new}}$
- $\mathbf{h}_t = \mathbf{o}_t * \tanh(\mathbf{c}_t)$

Comments:

- Simpler to remember.
- Better performance (exploiting parallel-GEMM (General Matrix to Matrix Multiplication)).

Implementation in PyTorch

```
def LSTMCell(input, hidden, w_ih, w_hh, b_ih=None, b_hh=None):
    if input.is_cuda:
        ...
    hx, cx = hidden
    gates = F.linear(input, w_ih, b_ih) + F.linear(hx, w_hh, b_hh)
    ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)
    ingate     = F.sigmoid(ingate)
    forgetgate = F.sigmoid(forgetgate)
    cellgate   = F.tanh(cellgate)
    outgate    = F.sigmoid(outgate)
    cy = (forgetgate * cx) + (ingate * cellgate)
    hy = outgate * F.tanh(cy)
    return hy, cy
```

- `input`: $\mathbf{x}_t$
- `...gate`: $\mathbf{x}_t$
- `hidden`: `(hx, cx)` of the last step.
- `chunk(chunks, dim)`: Splits a tensor into a specific number of chunks.

Deep LSTM

- Signature of LSTM: $[\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_t] = \mathrm{LSTM}([\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t])$
- Can stack LSTM layers: use $[\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_t]$ (multiplied by Bernoulli random variables determined by dropout if any) as the input $\mathbf{x}_i$s of the next layer
- Usually improves the performance at the expense of longer training time.

Bi-LSTM

- Often the dependencies within sequential data are not just in one direction, but may be observed in both!
- Bi-LSTM: combining hidden states obtained from both directions simultaneously.
  - $\mathbf{h}_t = [\mathbf{h}_t^{\rightarrow} \; ; \; \mathbf{h}_t^{\leftarrow}]$

LSTM in PyTorch

- All implemented in one high-level class: `torch.nn.LSTM`.
  - `num_layers`
  - `dropout`
  - `bidirectional`

Additional Notes about LSTM

- Initialization:
  - Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target)
  - Initialize weights $\sim \mathrm{Uniform}(r, r)$, r inversely proportional to fan-in (previous layer size) and fan-out (next layer size):
    - $\sqrt{6/(\mathrm{fan\_in} + \mathrm{fan\_out})}$ for tanh units,
    - 4x bigger for sigmoid units
- Regularization:
  - traditional regularization mechanisms;
  - early stop;
  - dropout on non-recurrent connections;
  - encourage sparse activiation by $KL(\frac{1}{n}\sum_{i=1}^{N} a_i^{(n)} \| 0.00001)$
  - add more data
- Optimizer: SGD (with momentum), Adam, RMSProp

The Input Embedding Layer

- What if the input sequences are words?
  - Naive solution: one-hot encoding of size $|V|$.
  - Embedding: use each words' embedding as its input vector. Or mathematically, use $\mathbf{Ex}$, where $\mathbf{x}$ is the one-hot encoding vector of x.
    - $\mathbf{E}$ of size $(d, |V|)$.
    - $\mathbf{E}$ can be learned by models such as word2vec or glove.