NEURAL NETWORKS – PART 3

DATA/MSML 603: Principles of Machine Learning

Last Time

- Training Neural Networks
  - Stochastic gradient descent
  - Backpropagation – forward propagation & backward propagation
- Activation functions
  - Vanishing gradient & exploding gradient
- Loss functions
- Underfitting & Overfitting

Ensemble Methods

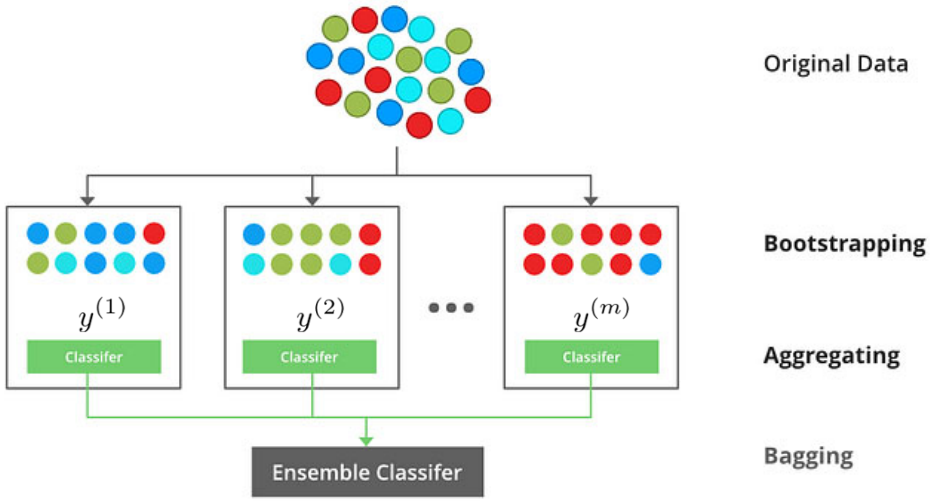
Bootstrap Aggregating (Bagging)

- Recall bias-variance trade-off
  - More flexible a model, the lower its bias will be and the higher the risk of overfitting (i.e., higher model variance)
- Model variance of flexible models can be reduced by using them as base (or ensemble) models in bootstrap aggregating (bagging) without increasing bias
- Bootstrap – method for artificially creating multiple datasets from one dataset
  - All datasets have the same size (as the original dataset)
  - Intuition: Suppose the original dataset is representative of real samples/distribution, i.e., if we collected more training data, the new data points/samples would likely be similar to the existing training data. Then, randomly choosing samples from the original dataset would be a reasonable way to simulate new training datasets
  - Each bootstrapped dataset obtained using **sampling with replacement**
    - May contain more than one copy of some of the original training data points/samples

Bootstrap Aggregating (Bagging)

- Bootstrap Algorithm used to generate m datasets
- Multiple (m) classifiers are trained in parallel using m datasets
- Output of ensemble classifier given by average

$$\hat{y}_{bag}(\mathbf{x}) = \frac{1}{m}\sum_{l=1}^{m}\hat{y}^{(l)}(\mathbf{x})$$



[source: medium.com]

$$\hat{y}_{bag}(\mathbf{x}) = \frac{1}{m}\sum_{l=1}^{m}\hat{y}^{(l)}(\mathbf{x})$$

Bootstrap Aggregating (Bagging)

- But, how does it reduce model variance?
- Suppose $\hat{y}^{(1)}(\mathbf{x}), \hat{y}^{(2)}(\mathbf{x}), \ldots, \hat{y}^{(m)}(\mathbf{x})$ denote the predictions from individual ensemble members
- We can view them as identically distributed random variables, but not necessarily independent with mean $\mathbb{E}[\hat{y}^{(1)}(\mathbf{x})] = \mu$ and $Var(\hat{y}^{(1)}(\mathbf{x})) = \sigma^2$

$$\mathbb{E}\left[\hat{y}_{bag}(\mathbf{x})\right] = \mathbb{E}\left[\frac{1}{m}\sum_{l=1}^{m}\hat{y}^{(l)}(\mathbf{x})\right] = \mu$$

$$Var\left(\frac{1}{m}\sum_{l=1}^{m}\hat{y}^{(l)}(\mathbf{x})\right) = \frac{1-\rho}{m}\sigma^2 + \rho\sigma^2 < \sigma^2 \quad \text{if } \rho < 1$$

$\rho-$ Correlation coefficient between predictions of two ensemble members

Boosting

- Primarily used for reducing bias in high-bias base models (often simple models)
  - Reduce the bias by turning ensemble of simple/weak models into one strong model
- Unlike in bagging where multiple models are trained in parallel, boosting uses a sequential construction of ensemble members (cannot be done in parallel)
  - Each model tries to correct the mistakes of previous model by modifying training dataset at each iteration – put larger weights on data points/samples for which the previous models performed poorly
- Two key details that need to be determined
  - How to compute the weights for training data points/samples at each iteration
  - How to combine ensemble models to construct the final model

AdaBoost (Adaptive Boosting)

- First successful implementation of boosting
- Hard prediction for binary classification: $\hat{y}^{(1)}(\mathbf{x}), \hat{y}^{(2)}(\mathbf{x}), \ldots, \hat{y}^{(m)}(\mathbf{x})$
- Boosted classifier using a weighted majority vote

$$\hat{y}^{(m)}_{boost}(\mathbf{x}) = \text{sign}\left\{\sum_{l=1}^{m} \alpha^{(l)} \hat{y}^{(l)}(\mathbf{x})\right\}$$

  - Each ensemble member votes either -1 or +1
  - Coefficients $\alpha^{(l)} > 0$ can be viewed as a level of **confidence** in the prediction by l-th ensemble member

AdaBoost (Adaptive Boosting)

- Ensemble members and their coefficients $\alpha^{(l)}$ are trained greedily by minimizing exponential loss of boosted classifier at each iteration
- For each $l = 0, 1, \ldots, m$, define $f^{(l)}(\mathbf{x}) = \sum_{k=1}^{l} \alpha^{(k)} \hat{y}^{(k)}(\mathbf{x})$ with $f^{(0)}(\mathbf{x}) = 0$
- Boosted classifier after l iterations $\hat{y}^{(l)}_{boost}(\mathbf{x}) = \text{sign}\{f^{(l)}(\mathbf{x})\}$
- $\hat{y}^{(l)}$ is selected to minimize the exponential loss

$$\sum_{i=1}^{n} L\left(y^i \cdot f^{(l)}(\mathbf{x}^i)\right) = \sum_{i=1}^{n} \exp\left(-y^i \left(\underbrace{f^{(l-1)}(\mathbf{x}^i) + \alpha^{(l)} \hat{y}^{(l)}(\mathbf{x}^i)}_{= f^{(l)}(\mathbf{x}^i)}\right)\right)$$

AdaBoost (Adaptive Boosting)

- At the l-th iteration, need to find

$$\begin{aligned}
(\alpha^{(l)}, \hat{y}^{(l)}) &= \arg\min_{(\alpha,\hat{y})} \sum_{i=1}^{n} \exp\left(-y^i\left(f^{(l-1)}(\mathbf{x}^i) + \alpha\hat{y}(\mathbf{x}^i)\right)\right) && (\alpha > 0) \\
&= \arg\min_{(\alpha,\hat{y})} \sum_{i=1}^{n} \underbrace{\exp\left(-y^i f^{(l-1)}(\mathbf{x}^i)\right)}_{=:\, w_i^{(l)}} \exp\left(-y^i\alpha\hat{y}(\mathbf{x}^i)\right) && \left(e^{a+b} = e^a \cdot e^b\right) \\
&= \arg\min_{(\alpha,\hat{y})} \sum_{i=1}^{n} w_i^{(l)} \exp\left(-y^i\alpha\hat{y}(\mathbf{x}^i)\right)
\end{aligned}$$

- Define $W = \sum_{i=1}^{n} w_i^{(l)}$ be the total weight sum

AdaBoost (Adaptive Boosting)

- Break the function to be minimized into two terms – one for data points that are correctly classified by candidate $\hat{y}$ and the other for incorrectly classified data points

$$\sum_{i=1}^{n} w_i^{(l)} \exp\left(-y^i \alpha \hat{y}(\mathbf{x}^i)\right) = e^{-\alpha} \underbrace{\sum_{i=1}^{n} w_i^{(l)} \mathbf{1}\left\{y^i = \hat{y}(\mathbf{x}^i)\right\}}_{=: W_c} + e^{\alpha} \underbrace{\sum_{i=1}^{n} w_i^{(l)} \mathbf{1}\left\{y^i \neq \hat{y}(\mathbf{x}^i)\right\}}_{=: W_e}$$

$$(W = W_c + W_e) \qquad = e^{-\alpha} W + \underbrace{(e^{\alpha} - e^{-\alpha})}_{>0} W_e$$

Weighted misclassification loss

- If we want to minimize the exponential loss, we need to minimize W_e, i.e.,

$$\hat{y}^{(l)} = \arg\min_{\hat{y}} \sum_{i=1}^{n} w_i^{(l)} \mathbf{1}\left\{y^i \neq \hat{y}(\mathbf{x}^i)\right\}$$

AdaBoost (Adaptive Boosting)

$$\hat{y}^{(l)} = \arg\min_{\hat{y}} \sum_{i=1}^{n} w_i^{(l)} \mathbf{1}\left\{y^i \neq \hat{y}(\mathbf{x}^i)\right\}$$

- The l-th ensemble member should be trained by minimizing the weighted misclassification loss with weights $w_i^{(l)} = \exp\left(-y^i f^{(l-1)}(\mathbf{x}^i)\right)$
  - At iteration l, should focus our attention on the data points/samples previously misclassified to correct the mistakes of the first l-1 classifiers

- What about the coefficient $\alpha^{(l)}$?

AdaBoost (Adaptive Boosting)

- Recall that the function we want to minimize is given by

$$\sum_{i=1}^{n} w_i^{(l)} \exp\left(-y^i \alpha \hat{y}(\mathbf{x}^i)\right) = e^{-\alpha} + (e^{\alpha} - e^{-\alpha}) W_e \qquad (10\text{-}1)$$

- Once we choose $\hat{y}^{(l)}$, $W_e = \sum_{i=1}^{n} w_i^{(l)} \mathbf{1}\left\{y^i \neq \hat{y}^{(l)}(\mathbf{x}^i)\right\}$ is a fixed constant

- Differentiate (10-1) w.r.t. α and set the derivative to zero

$$-e^{-\alpha} W + W_e(e^{\alpha} - (-1)e^{-\alpha}) = -e^{-\alpha} W + W_e(e^{\alpha} + e^{-\alpha}) = 0$$

$$\Leftrightarrow \quad W = W_e(e^{2\alpha} + 1)$$

$$\Leftrightarrow \quad \alpha^{(l)} = \frac{1}{2} \log\left(\frac{W}{W_e} - 1\right)$$

Reflects the confidence in the classifier's predictions

Depends on the training error of $\hat{y}^{(l)}$

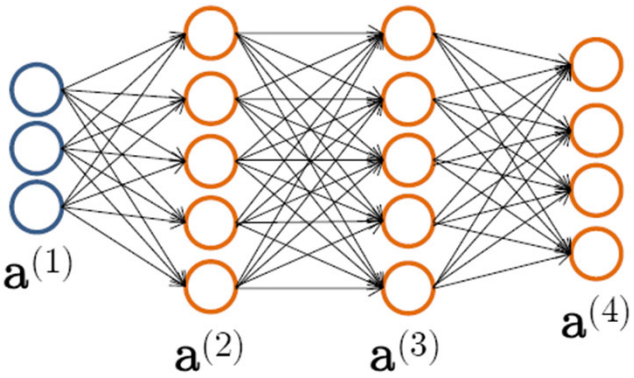
Recurrent Neural Networks

Feedforward NN

- Neural network used to approximate the relationship between the input (e.g., feature values) and the output (e.g., label)
  - For each example, input values determine output value, independently of other examples

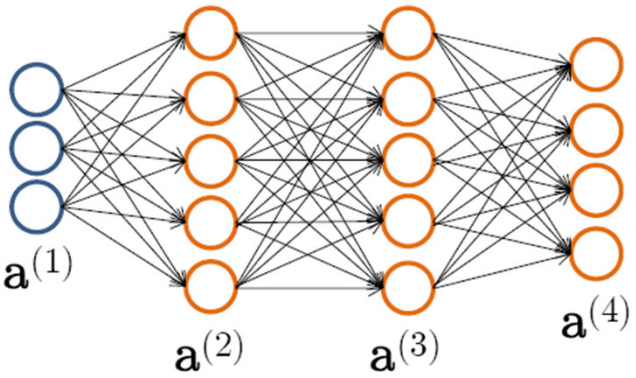
$$\mathbf{y} = f(\mathbf{x})$$

  - Universal function approximator – when the neural network size is sufficiently large, it can approximate a large class of functions using appropriate weights



Feedforward NN

- In some cases, inputs are sequential and outputs cannot be computed independently
  - Often need to discover the pattern in sequential data (e.g., time series forecasting)
  - Requires some “memory” of the past
- In addition, a neural network is trained using fixed-size inputs, i.e., the dimension of inputs is pre-determined and cannot be changed on the fly
- Question: What if we don’t know the exact length of the inputs that need to be handled by a neural network or the inputs are sequential data of variable length?



Example

- Predicting the next word in the sequence
  - *He was born in Germany and grew up in Hamburg till he was a teenager. It is no wonder that he is fluent in …...*
  - *Tom and Mary are married, and Tom is her …….*
- Machine translation
  - *Come arrivare al Duomo di Milano?* ⟹ *How do I get to Duomo di Milano?*
- Time series prediction
- Sentiment analysis

Recurrent Neural Networks

- Dates back to (Rumelhart *et al.*, 1986)
- A family of neural networks for handling **sequential data**, which involves variable length inputs or outputs

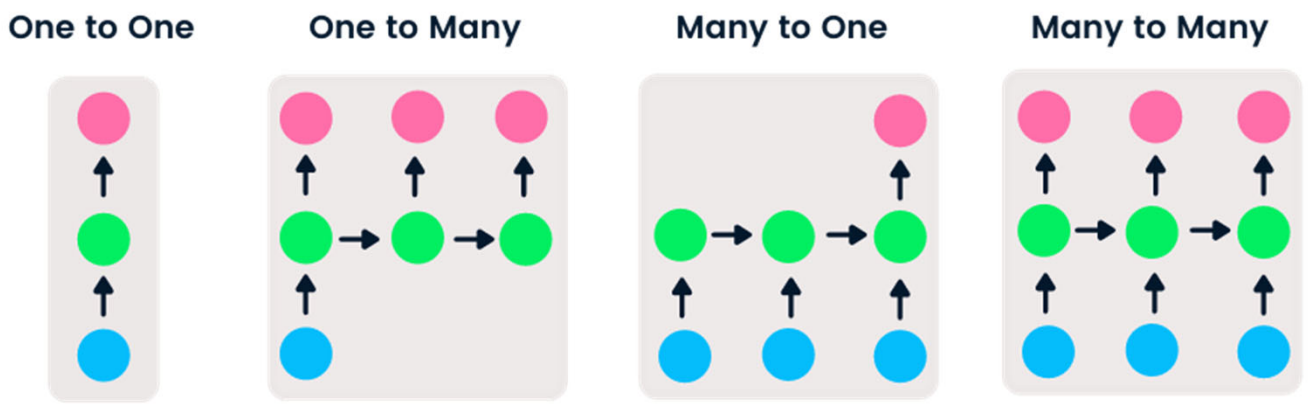
Sequential Data

- Each data point: A sequence of vectors $x(t)$, for $1 \leq t \leq \tau$
- Batch data: many sequences with different lengths τ
- Label: can be a scalar, a vector, or even a sequence

Types of RNNs

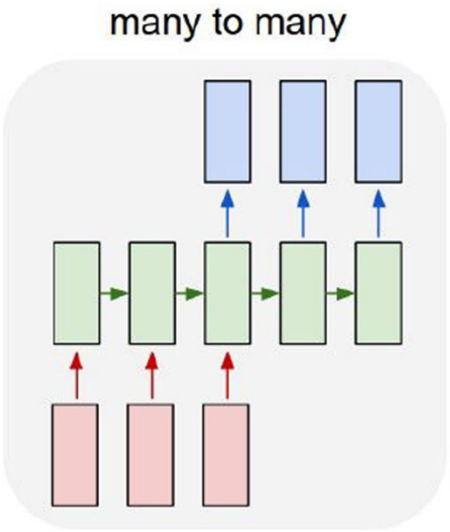
- Depending on the lengths/number of inputs and outputs
  - One-to-one – single input and single output
  - One-to-many – single input and multiple outputs
    - Generating image captions
  - Many-to-one – takes a sequence of multiple inputs and predicts a single output
    - Sentiment classification (e.g., input = text with multiple words, output = category)
  - Many-to-many – takes multiple inputs and produces multiple outputs
    - Machine language translation

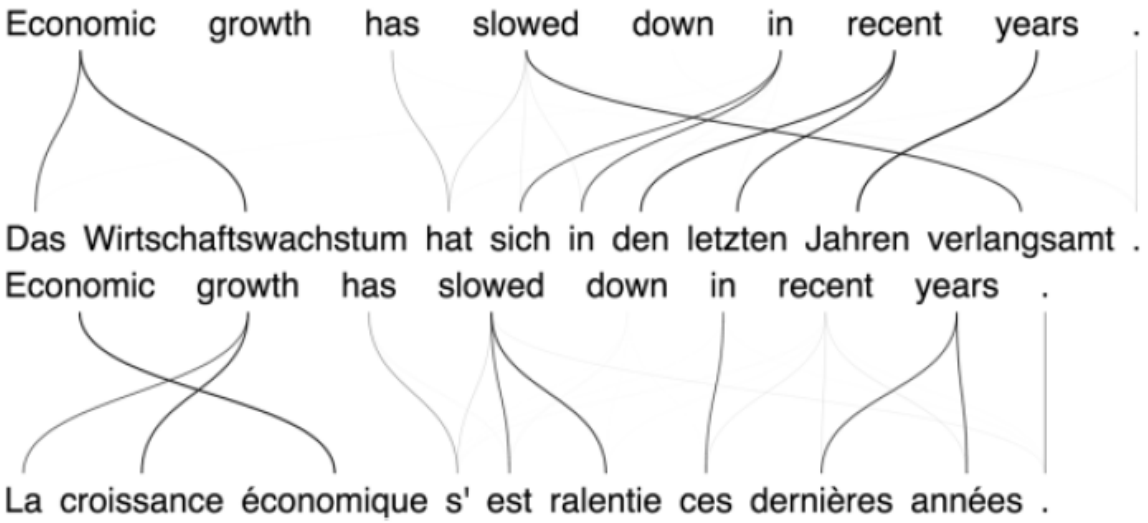
Types of RNNs



[source: datacamp.com]

Sequential Data: Machine Translation



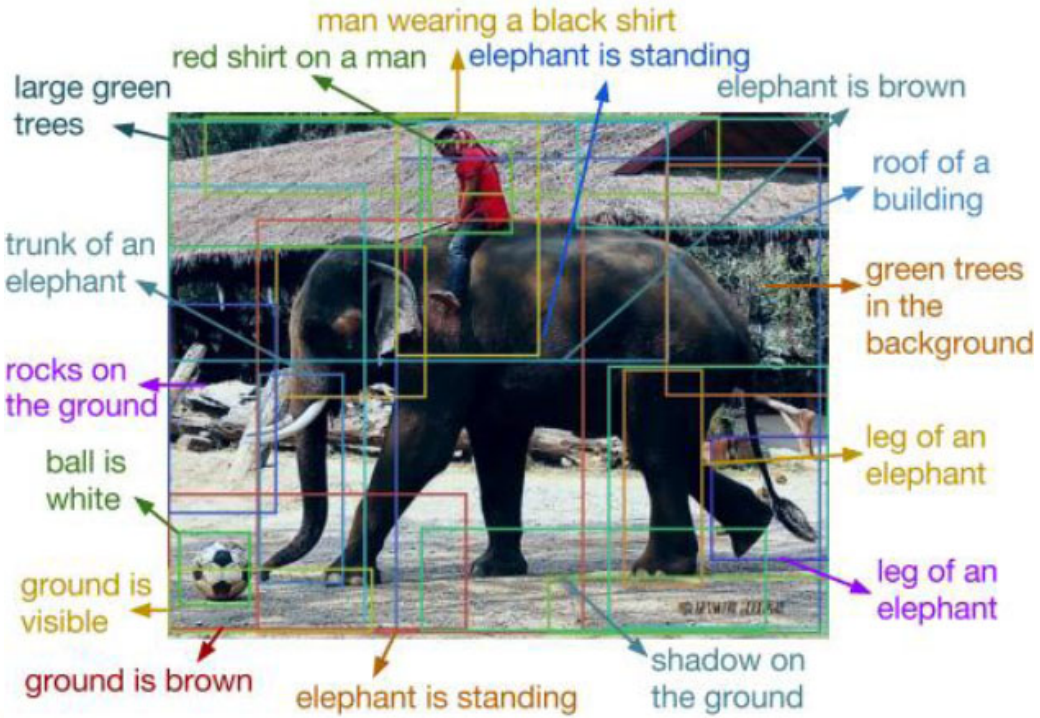


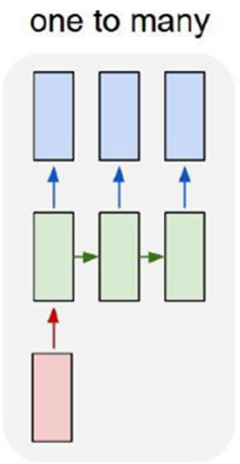
e.g. **Machine Translation**
seq of words -> seq of words

More Complicated Sequential Data

- **Data point:** two dimensional sequences like images
- **Label:** different type of sequences like text sentences
- Example: image captioning

Image Captioning





e.g. **Image Captioning**
image -> sequence of words

Image source: "DenseCap: Fully Convolutional Localization Networks for Dense Captioning," by Justin Johnson, Andrej Karpathy, Li Fei-Fei

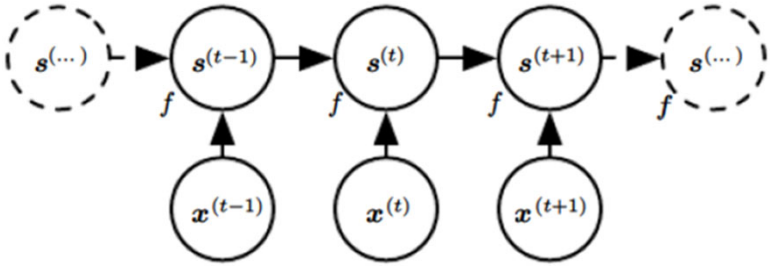
Key Features

- Can handle inputs of variable lengths with a fixed number of parameters
- Parameter sharing: same weights used for all inputs (reduced complexity for generalization)
- Hidden state or memory state (computed for each input)
  - Used to keep memory of past inputs
  - Evolves over time steps
  - Weights updated using backpropagation through time (BPTT)

A System Driven By External Data

- Discrete-time dynamical system driven by external input
  - $s^{(t)}$: state of the system at time t
  - $x^{(t)}$: input at time t

- State of the system at time t+1 depends on two things
  - State of the system at time t
  - Input at time t

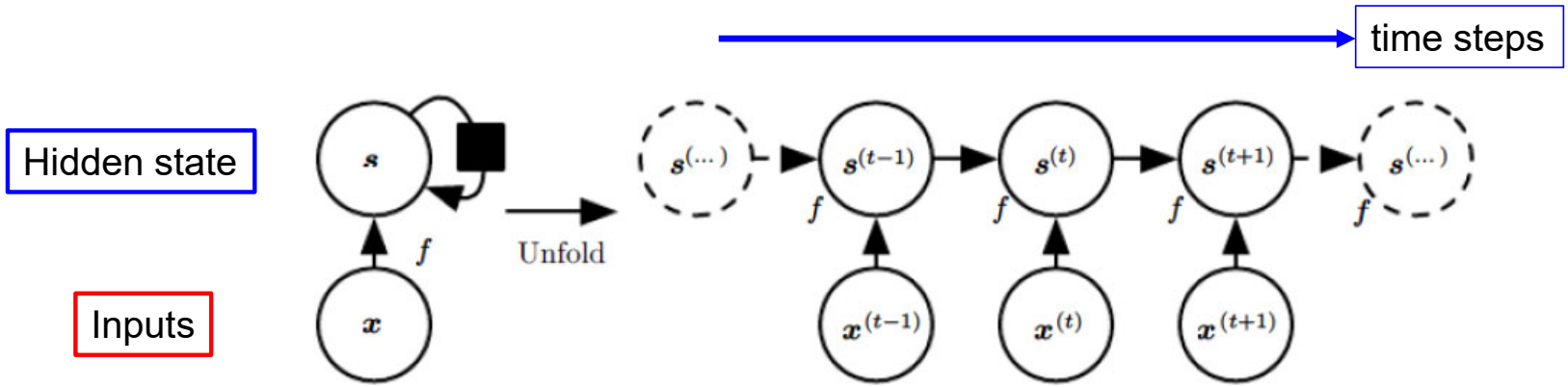


$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Fixed parameters

Figure from Deep Learning, Goodfellow, Bengio and Courville

Compact View

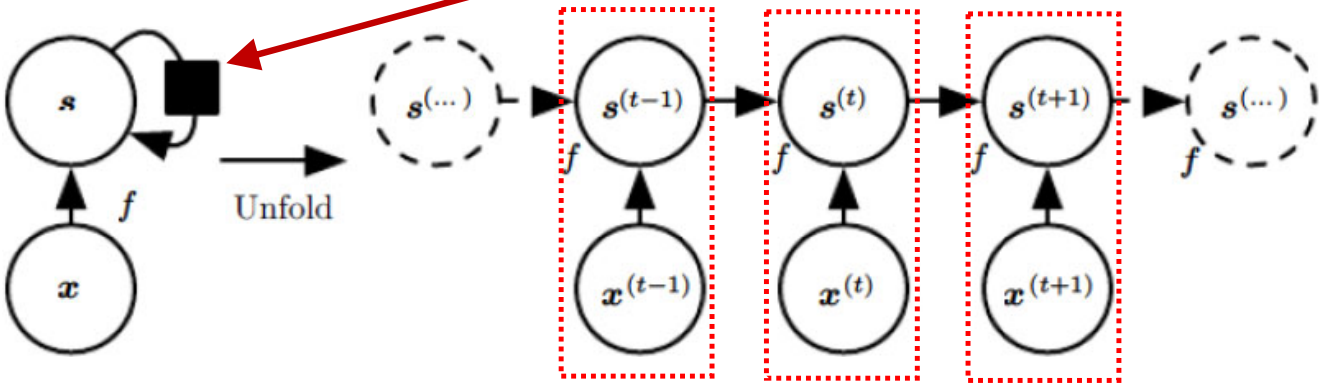


$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Figure from Deep Learning, Goodfellow, Bengio and Courville

Compact View

Square: One step time delay



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Key: the same f and θ for all time steps

Figure from Deep Learning, Goodfellow, Bengio and Courville

Recurrent Neural Networks

- Use **the same computational function (f) and parameters (θ)** across different time steps of the sequence (for better generalization)
- Each time step: takes the input entry and **the previous hidden state** to compute the output entry
- Loss: typically computed at every time step

Recurrent Neural Networks (single layer)

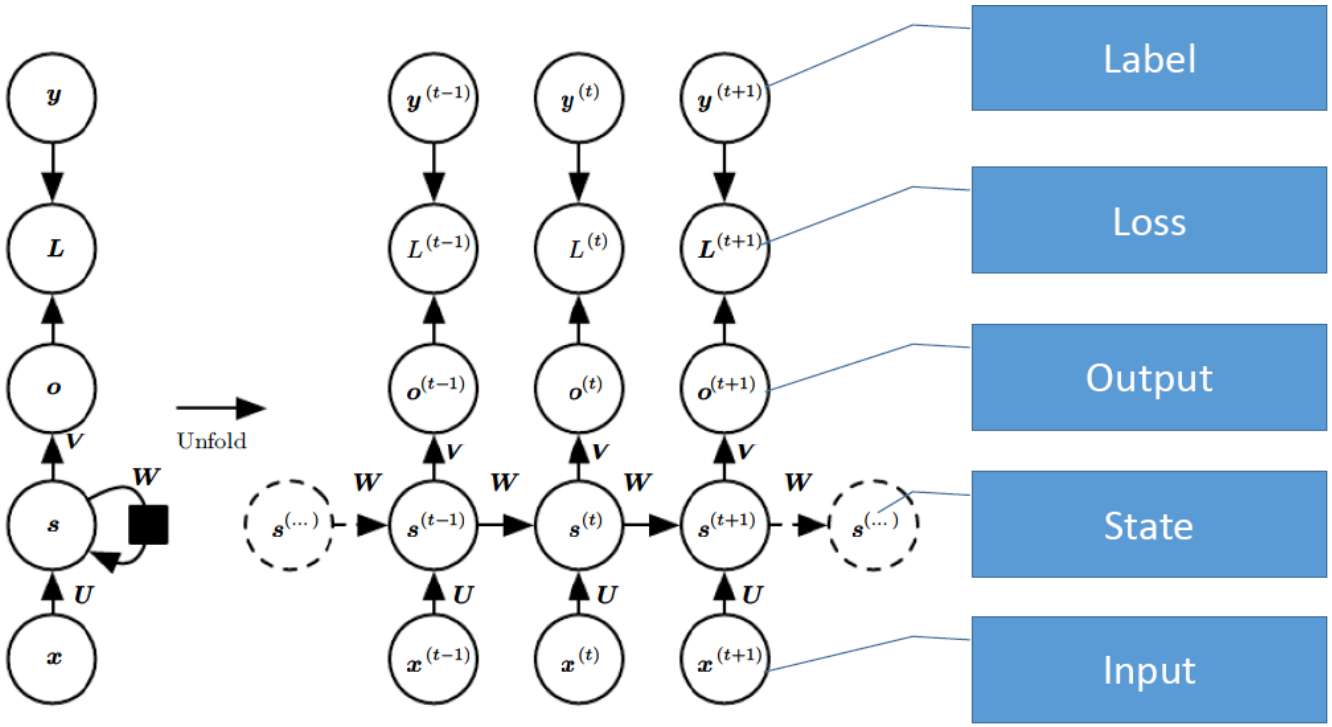
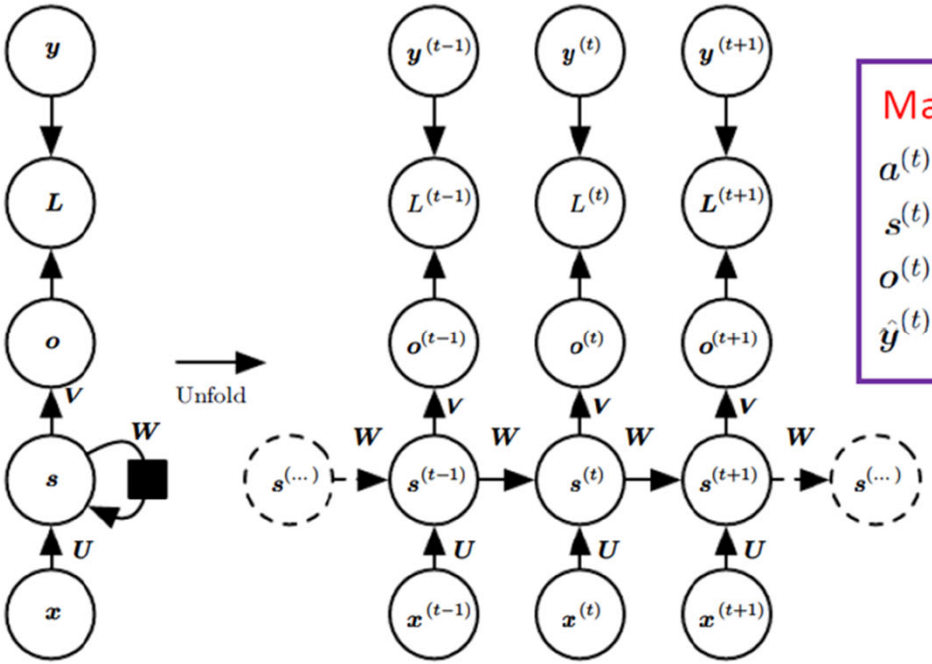


Figure from Deep Learning, Goodfellow, Bengio and Courville

Recurrent Neural Networks



Math formula:

$$a^{(t)} = b + Ws^{(t-1)} + Ux^{(t)}$$
$$s^{(t)} = \tanh(a^{(t)})$$
$$o^{(t)} = c + Vs^{(t)}$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

state transition with a single layer of tanh

Figure from Deep Learning, Goodfellow, Bengio and Courville

Advantages

- Hidden state: lossy summary of the past
- Shared functions and parameters: greatly reduce the capacity and good for **generalization** in learning
- Explicitly use the **prior knowledge** that the sequential data can be processed in the same way at different time step (e.g., NLP)
- Yet still powerful (Turing-complete): any function computable by a Turing machine can be computed by such a recurrent network of a finite size (see, e.g., Siegelmann and Sontag [1995])

Training RNN

- Principle: unfold the computational graph, and use backpropagation over time steps
- Called backpropagation through time (BPTT) algorithm
- Can then apply any general-purpose gradient-based techniques
- Conceptually: first compute the gradients of **the internal nodes**, then compute the gradients of **the parameters**

Training RNN

- RNN dynamics

$$s^{(t)} = f(x^{(t)}, s^{(t-1)}, \mathbf{W}_s),\ \mathbf{W}_s = [\mathbf{W}\ \mathbf{U}\ \mathbf{b}]$$
$$o^{(t)} = g(s^{(t)}, \mathbf{V}_o),\ \mathbf{V}_o = [\mathbf{V}\ \mathbf{c}]$$

- Loss: $L(\mathbf{x}, \mathbf{y}, \mathbf{W}_s, \mathbf{V}) = \sum_{t=1}^{T} L^{(t)}(y^{(t)}, o^{(t)})$

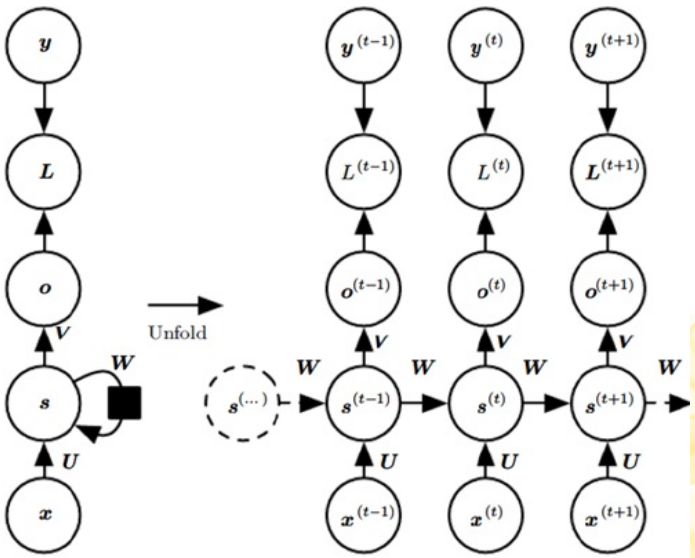
- Derivative w.r.t. $\mathbf{V}_o$

$$\frac{\partial L}{\partial \mathbf{V}} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial L^{(t)}(y^{(t)}, o^{(t)})}{\partial \mathbf{V}}$$
$$= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial L^{(t)}(y^{(t)}, o^{(t)})}{\partial o^{(t)}} \frac{\partial g(s^{(t)}, \mathbf{V})}{\partial \mathbf{V}}$$

Math formula:

$$a^{(t)} = b + Ws^{(t-1)} + Ux^{(t)}$$
$$s^{(t)} = \tanh(a^{(t)})$$
$$o^{(t)} = c + Vs^{(t)}$$
$$\hat{y}^{(t)} = \mathrm{softmax}(o^{(t)})$$

state transition formula



Training RNN

Math formula:

$$
\begin{aligned}
a^{(t)} &= b + W s^{(t-1)} + U x^{(t)} \\
s^{(t)} &= \tanh(a^{(t)}) \\
o^{(t)} &= c + V s^{(t)} \\
\hat{y}^{(t)} &= \mathrm{softmax}(o^{(t)})
\end{aligned}
$$

state transition formula

- RNN dynamics

$$s^{(t)} = f(x^{(t)}, s^{(t-1)}, \mathbf{W}_s), \ \mathbf{W}_s = [\mathbf{W}\ \mathbf{U}]$$
$$o^{(t)} = g(s^{(t)}, \mathbf{V})$$

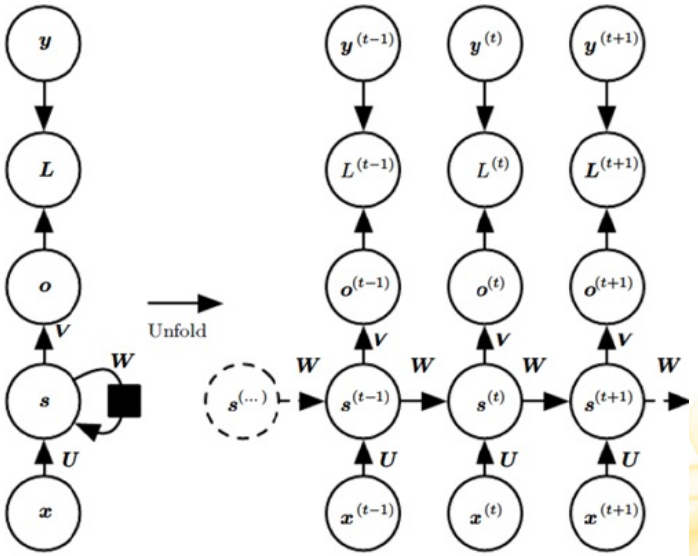
- Loss: $L(\mathbf{x}, \mathbf{y}, \mathbf{W}_s, \mathbf{V}) = \sum_{t=1}^{T} L^{(t)}(y^{(t)}, o^{(t)})$

- Derivatives w.r.t. $\mathbf{W}_s$

$$\frac{\partial L}{\partial \mathbf{W}_s} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial L^{(t)}(y^{(t)}, o^{(t)})}{\partial \mathbf{W}_s}$$
$$= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial L^{(t)}(y^{(t)}, o^{(t)})}{\partial o^{(t)}} \frac{\partial g(s^{(t)}, \mathbf{V})}{\partial s^{(t)}} \frac{\partial s^{(t)}}{\partial \mathbf{W}_s}$$

Need to recurrently Compute the effect

Can be computed easily



Training RNN

Math formula:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

state transition formula

- RNN dynamics

$$s^{(t)} = f(x^{(t)}, s^{(t-1)}, \mathbf{W}_s),\ \mathbf{W}_s = [\mathbf{W}\ \mathbf{U}]$$

$$\frac{\partial s^{(t)}}{\partial \mathbf{W}_s} = \frac{\partial f(x^{(t)}, s^{(t-1)}, \mathbf{W}_s)}{\partial \mathbf{W}_s} + \frac{\partial f(x^{(t)}, s^{(t-1)}, \mathbf{W}_s)}{\partial s^{(t-1)}} \frac{\partial s^{(t-1)}}{\partial \mathbf{W}_s}$$

recursion

$$= \frac{\partial f(x^{(t)}, s^{(t-1)}, \mathbf{W}_s)}{\partial \mathbf{W}_s} + \sum_{\tau=1}^{t-1} \left(\prod_{t'=\tau+1}^{t} \frac{\partial f(x^{(t')}, s^{(t'-1)}, \mathbf{W}_s)}{\partial s^{(t'-1)}} \right) \frac{\partial f(x^{(\tau)}, s^{(\tau-1)}, \mathbf{W}_s)}{\partial \mathbf{W}_s}$$

Often truncate this term in practice

Recurrent Neural Networks

- Use **the same** computational function and parameters across different time steps of the sequence
  - Reduced complexity for improved generalization
- Each time step: takes the input entry and **the previous hidden state** to compute the output entry
- Loss: typically computed at every time step
- Many variants
  - Information about the past can be in many other forms
  - Only output at the end of the sequence

RNN Variations

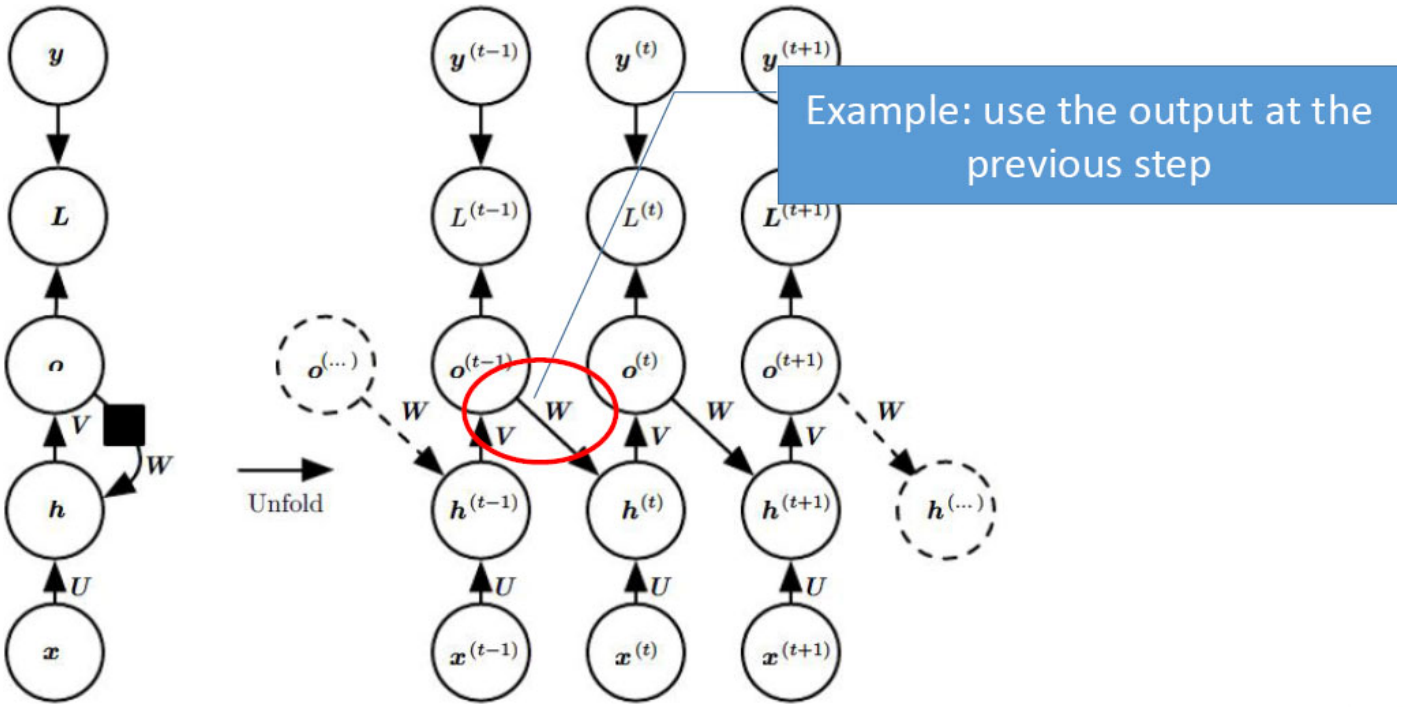


Figure from *Deep Learning,* Goodfellow, Bengio and Courville

RNN Variations

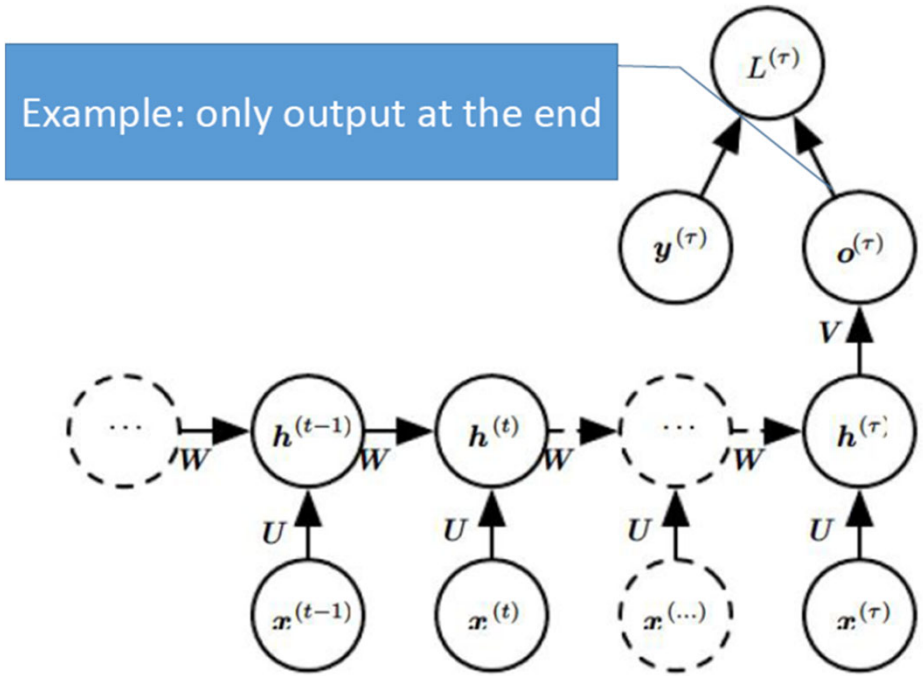


Figure from *Deep Learning,* Goodfellow, Bengio and Courville

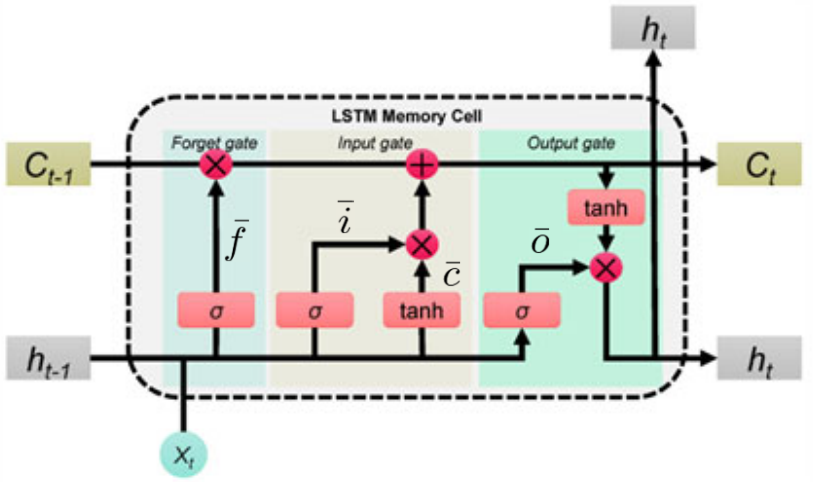
LSTM & GRU

Shortcoming of RNNs

- Training a RNN generally a difficult task (cannot be done in parallel due to the sequential nature of data)
- Short memory
  - Called long-term dependency problem
  - Caused by vanishing and exploding gradient problems
- Handled by Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU)
  - Introduces memory cells and gates

Long Short-Term Memory (LSTM)

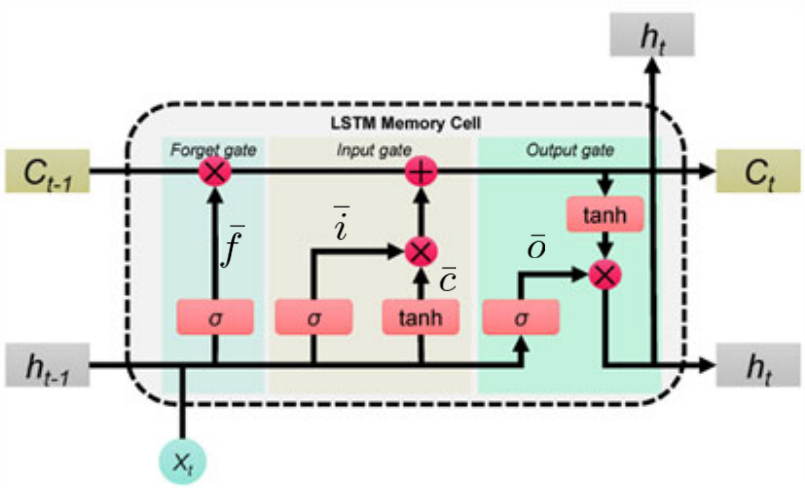
- Enhancement of RNN architecture
  - Change how hidden states are propagated with the help of additional hidden vector called the "cell state"
- Cell state maintains long-term memory
  - Cell states in different temporal layers share more similarity through long-term memory
- Introduces four (4) intermediate vector variables
  - $\bar{i}$ - input variable
  - $\bar{f}$ - forget variable
  - $\bar{o}$ - output variable



Long Short-Term Memory (LSTM)

- Cell state updated in two steps
- Intermediate variables
  - Input gate:
  - Forget gate:
  - Output gate:
  - New C-state:

$$\begin{bmatrix} \bar{i} \\ \bar{f} \\ \bar{o} \\ \bar{c} \end{bmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} \begin{bmatrix} \mathbf{W}_h & \mathbf{W}_x \end{bmatrix} \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}$$



- Update of cell state and hidden state

$$c_t = \bar{f} \odot c_{t-1} + \bar{i} \odot \bar{c} \qquad \text{[Selectively forget and add to long-term memory (cell state)]}$$

$$h_t = \bar{o} \odot tanh(c_t) \qquad \text{[Selectively leak long-term memory to hidden state]}$$

$\odot$ – element-wise product

Long Short-Term Memory (LSTM)

- Update of cell state and hidden state

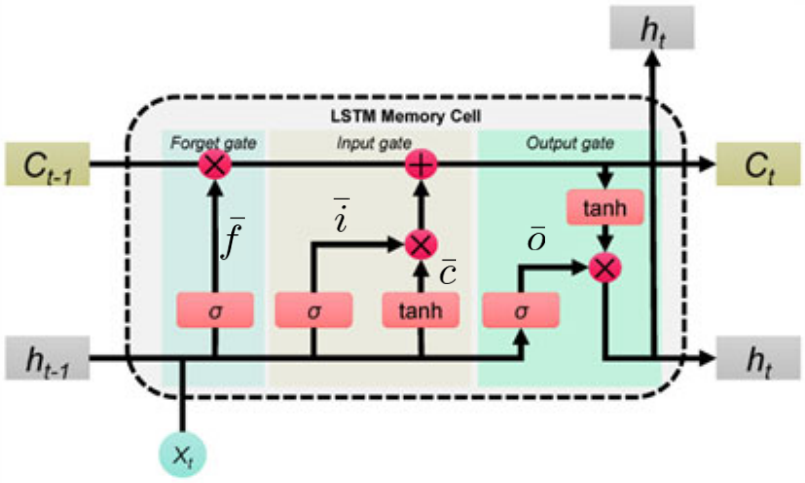
$$c_t = \bar{f} \odot c_{t-1} + \bar{i} \odot \bar{c} \quad \text{[Selectively forget and add to long-term memory (cell state)]}$$

$$h_t = \bar{o} \odot tanh(c_t) \quad \text{[Selectively leak long-term memory to hidden state]}$$

- Input gate:
- Forget gate:
- Output gate:
- New C-state:

$$\begin{bmatrix} \bar{i} \\ \bar{f} \\ \bar{o} \\ \bar{c} \end{bmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} \begin{bmatrix} \mathbf{W}_h & \mathbf{W}_x \end{bmatrix} \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}$$

- $\bar{c}$ contains newly proposed contents of cell state



Gated Recurrent Unit (GRU)

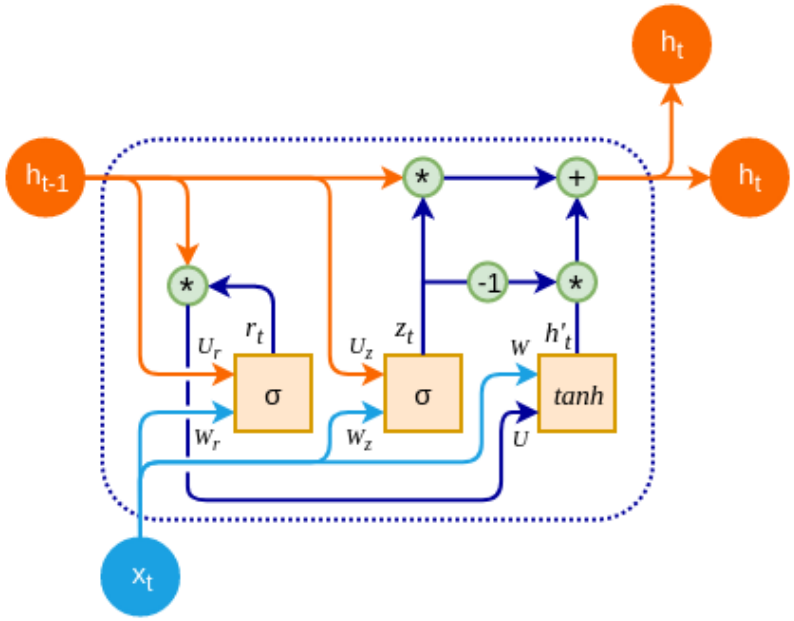
- Can be considered a simplification of LSTM without explicit cell state
  - Uses a single “reset” gate to achieve the functions of “forget” and “output” gates in LSTM
- Hidden state updated in two steps
  - Update gate:
  - Reset gate:

$$\begin{bmatrix} \bar{z} \\ \bar{r} \end{bmatrix} = \begin{pmatrix} \sigma \\ \sigma \end{pmatrix} \begin{bmatrix} \mathbf{W}_h \mathbf{W}_x \end{bmatrix} \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}$$

  - Hidden state update

$$h_t = \bar{z} \odot h_{t-1} + (1 - \bar{z}) \odot tanh \left(\mathbf{V} \begin{bmatrix} x_t \\ \bar{r} \odot h_{t-1} \end{bmatrix} \right)$$

- Reset gate determines how much of hidden state to carry over
- Update gate decides the relative strength of contributions of hidden state



Bidirectional RNNs

- Many applications: output at time t may depend on the whole input sequence
- Example in speech recognition: correct interpretation of the current sound may depend on the next few phonemes, potentially even the next few words
- Bidirectional RNNs are introduced to address this

BiRNNs

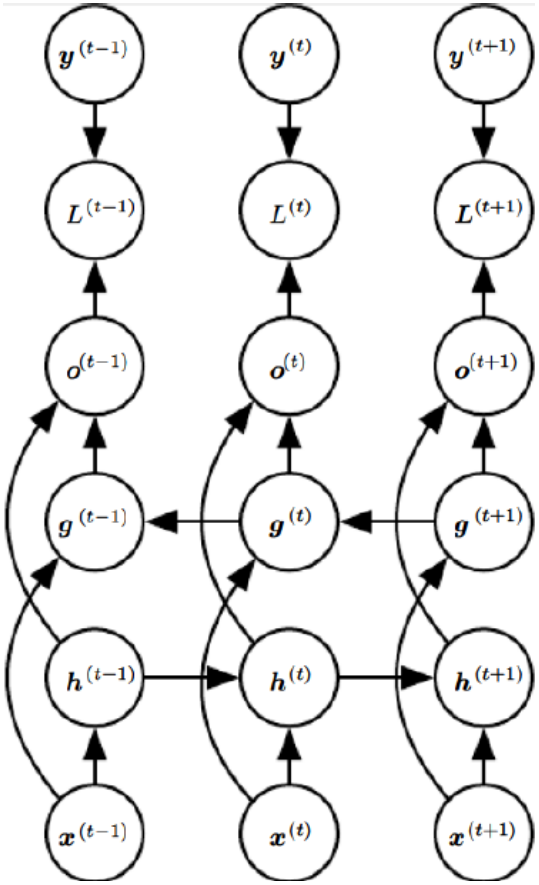


Image source: *Deep Learning,* Goodfellow, Bengio and Courville