Recurrent neural networks and Long-short term memory (LSTM)

Jeong Min Lee
CS3750

Outline

- RNN
  - RNN
  - Unfolding Computational Graph
  - Backpropagation and weight update
  - Explode / Vanishing gradient problem
- LSTM
- GRU
- Tasks with RNN
- Software Packages

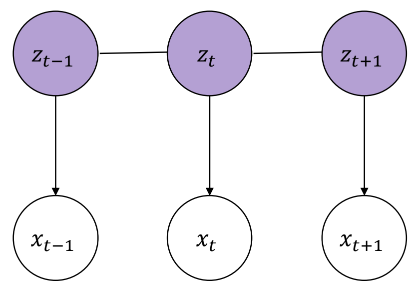
So far we are

- Modeling sequence (time-series) and predicting future values by probabilistic models (AR, HMM, LDS, Particle Filtering, Hawkes Process, etc)
- E.g. LDS
  - Observation x_t is modeled as **emission** matrix C, hidden state z_t with Gaussian noise w_t

    $$x_t = Cz_t + w_t \,;\, w_t \sim N(w|0, \Sigma)$$

  - The hidden state is also probabilistically computed with **transition** matrix A and Gaussian noise v_t

    $$z_t = Az_{t-1} + v_t \;\;;\, v_t \sim N(w|0, \Gamma)$$



Paradigm Shift to RNN

- We are moving into a new world where no probabilistic component exists in a model
- That is, you cannot do inference like in LDS and HMM
  - In RNN, hidden states bear no probabilistic form or assumption
- Given fixed input and target from data, RNN is to learn intermediate association between them and also the real-valued vector representation

RNN

- RNN's input, output, and internal representation (hidden states) are all real-valued vectors

$$h_t = \tanh(Ux_t + Wh_{t-1})$$

$$\hat{y} = \lambda(Vh_t)$$

- h_t: hidden states; real-valued vector
- x_t: input vector (real-valued)
- Vh_t: real-valued vector
- $\hat{y}$: output vector (real-valued)

RNN

- RNN consists of three parameter matrices (U, W, V) with activation functions

$$h_t = \tanh(Ux_t + Wh_{t-1})$$

$$\hat{y} = \lambda(Vh_t)$$

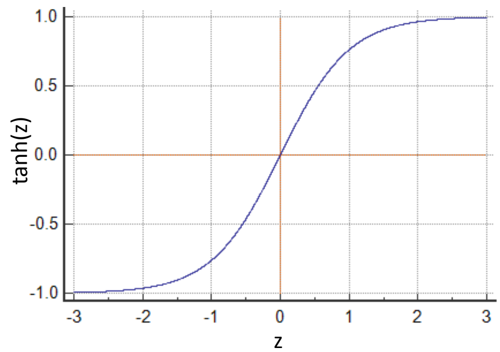
- U: input-hidden matrix
- W: hidden-hidden matrix
- V: hidden-output matrix

RNN

- $\tanh(\cdot)$ is a tangent hyperbolic function. It models non-linearity.

$$h_t = \tanh(Ux_t + Wh_{t-1})$$

$$\hat{y} = \lambda(Vht)$$



RNN

- $\lambda(\cdot)$ is output transformation function
- It can be any function and selected for a task and type of target in data
- It can be even another feed-forward neural network and it makes RNN to model anything without restriction

$$h_t = \tanh(Ux_t + Wh_{t-1})$$

$$\hat{y} = \lambda(Vht)$$

- Sigmoid: binary probability distribution
- Softmax: categorical probability distribution
- ReLU: positive real-value output
- Identity function: real-value output

Make a prediction

- Let's see how it makes a prediction
- In the beginning, initial hidden state h_0 is filled with zero or random value
- Also we assume the model is already trained. (we will see how it is trained soon)

Make a prediction

- Assume we currently have observation x_1 and want to predict x_2
- We compute hidden states h_1 first

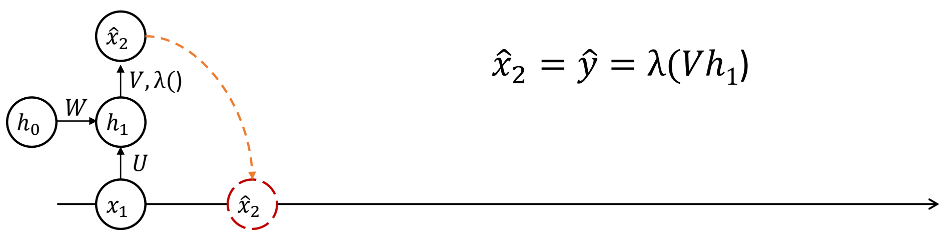
$$h_1 = \tanh(Ux_1 + Wh_0)$$

Make a prediction

- Then we generate prediction:
- Vh_1 is a real-valued vector or scalar value (depends on the size of output matrix V)

$$h_1 = \tanh(Ux_1 + Wh_0)$$

$$\hat{x}_2 = \hat{y} = \lambda(Vh_1)$$

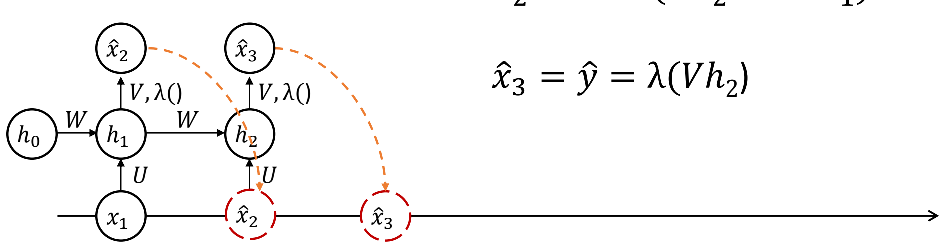


Make a prediction multiple steps

- In prediction for multiple steps a head, predicted value $\hat{x}_2$ from previous step is considered as input x_2 at time step 2

$$h_2 = \tanh(U\hat{x}_2 + Wh_1)$$

$$\hat{x}_3 = \hat{y} = \lambda(Vh_2)$$

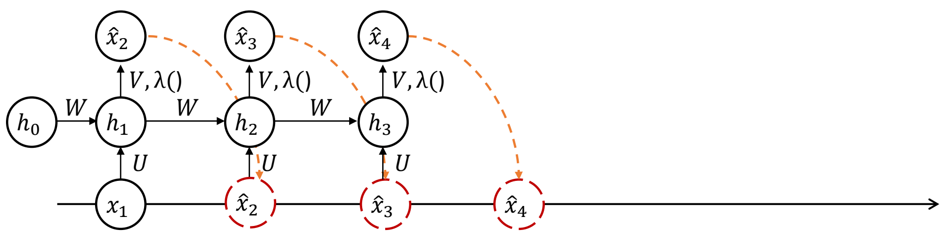


Make a prediction multiple steps

- Same mechanism applies forward in time..

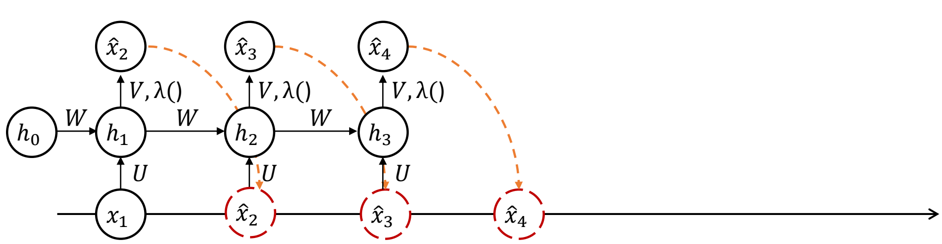
$$h_3 = \tanh(U\hat{x}_3 + Wh_2)$$

$$\hat{x}_4 = \hat{y} = \lambda(Vh_3)$$



RNN Characteristic

- You might observed that…
- Parameters U, V, W are shared across all time steps
- No probabilistic component (random number generation) is involved
- So, everything is deterministic



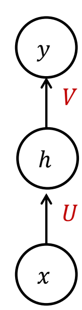
Another way to see RNN

- RNN is a type of neural network

Neural Network

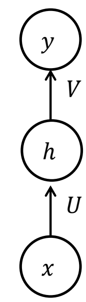
- Cascading several linear weights with nonlinear activation functions in between them

- y: output
- V: Hidden-Output matrix
- h: hidden units (states)
- U: Input-Hidden matrix
- x: input



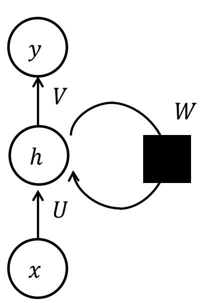
Neural Network

- In traditional NN, it is assumed that every input is independent each other
- But with sequential data, input in current time step is highly likely depends on input in previous time step
- We need some additional structure that can **model dependencies** of inputs **over time**

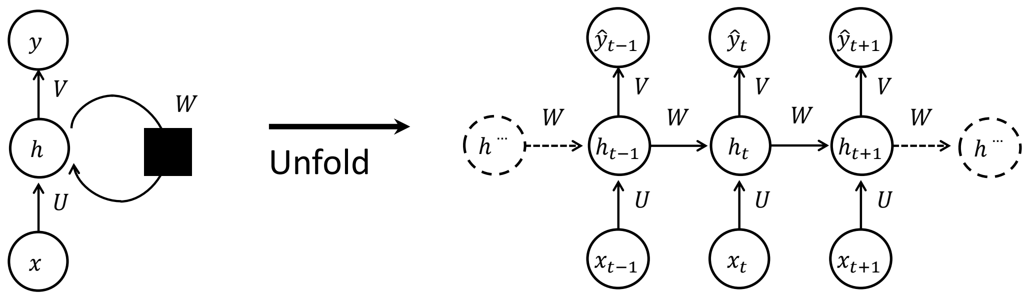


Recurrent Neural Network

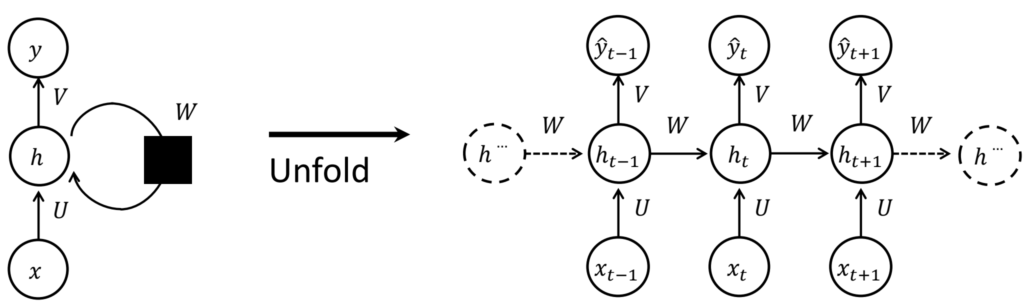
- A type of a neural network that has a recurrence structure
- The recurrence structure allows us to operate over a sequence of vectors



RNN as an Unfolding Computational Graph



RNN as an Unfolding Computational Graph

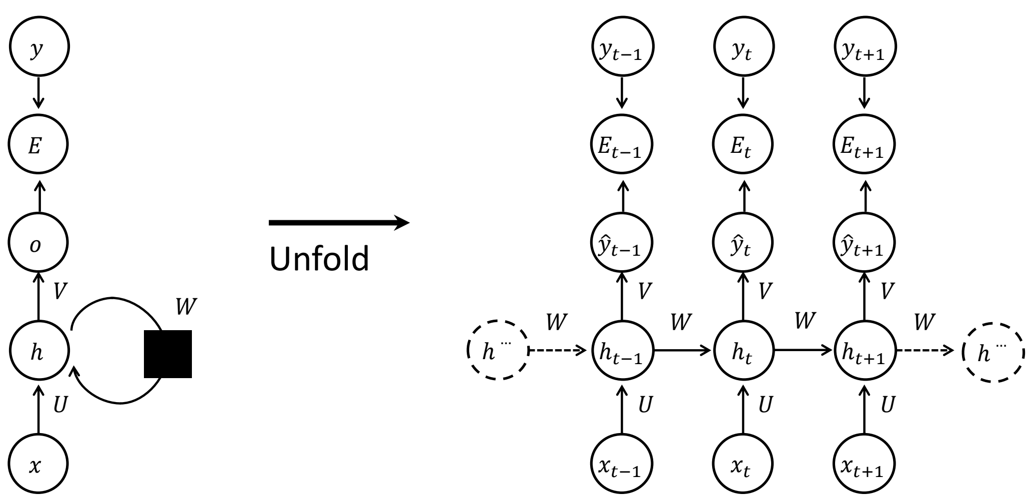


RNN can be converted into a feed-forward
neural network by **unfolding over time**

How to train RNN?

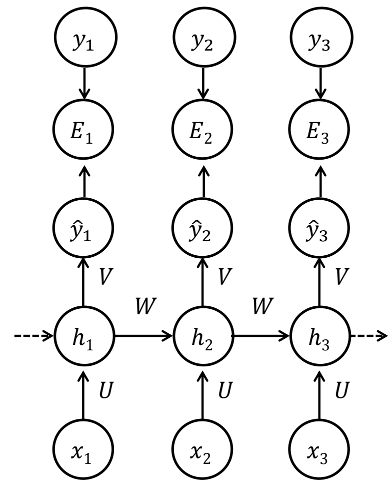
- Before make train happen, we need to define these:
- y^t : true target
- $\hat{y}_t$: output of RNN (=prediction for true target)
- E_t: error (loss); difference between the true target and the output
- As the output transformation function λ is selected by the task and data, so does the loss: (e.g.)
  - Binary Classification: Binary Cross Entropy
  - Categorical Classification: Cross Entropy
  - Regression: Mean Squared Error

With the loss, the RNN will be like:



Back Propagation Through Time (BPTT)

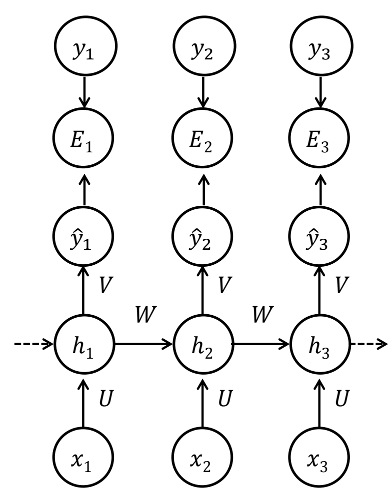
- Extension of standard backpropagation that performs gradient descent on an unfolded network
- Goal is to calculate gradients of the error with respect to parameters U, V, and W and learn desired parameters using Stochastic Gradient Descent



Back Propagation Through Time (BPTT)

- To update in one training example (sequence), we sum up the gradients at each time of the sequence:

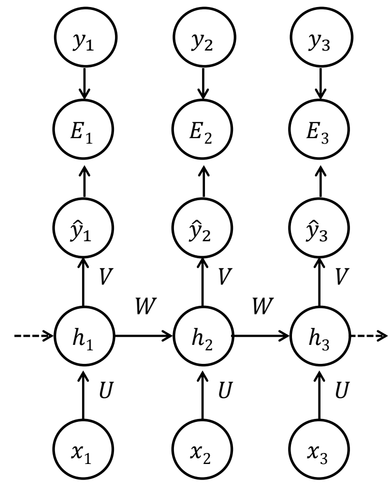
$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$



Learning Parameters

$$h_t = \tanh(Ux_t + Wh_{t-1})$$
$$z_t = Ux_t + Wh_{t-1}$$
$$h_t = \tanh(z_t)$$

- Let $\left\{ \begin{array}{l} \lambda_k = \dfrac{\partial h_k}{\partial W} \quad \alpha_k = \dfrac{\partial h_k}{\partial z_k} = 1 - {h_k}^2 \\ \beta_k = \dfrac{\partial E_k}{\partial h_k} = (o_k - y_k)V \end{array} \right.$

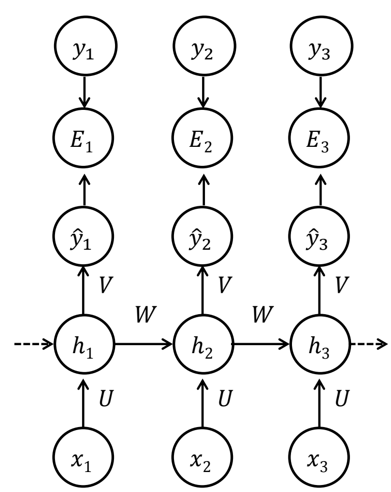


Learning Parameters

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial h_k}\frac{\partial h_k}{\partial W} = \beta_k \lambda_k$$

$$\lambda_k = \frac{\partial h_k}{\partial W} = \frac{\partial h_k}{\partial z_k}\frac{\partial z_k}{\partial W} = \alpha_k(h_{k-1} + W\lambda_{k-1})$$

$$\psi_k = \frac{\partial h_k}{\partial U} = \alpha_k \frac{\partial z_k}{\partial U} = \alpha_k(x_k + W\psi_{k-1})$$



Initialization:

$$\alpha_0 = 1 - {h_0}^2;\ \lambda_0 = 0;\ \psi_0 = \alpha_0 \cdot x_0$$
$$\Delta w = 0\ ;\ \Delta u = 0\ ; \Delta v = 0$$

For k= 1…T (T; length of a sequence):

$$\alpha_k = 1 - {h_k}^2$$
$$\lambda_k = \alpha_k(h_{k-1} + W\lambda_{k-1})$$
$$\beta_k = (o_k - y_k)V$$
$$\Delta w = \Delta w + \beta_k\lambda_k$$
$$\psi_k = \alpha_k(x_k + W\psi_{k-1})$$
$$\Delta u = \Delta u + \beta_k\psi_k$$
$$\Delta v = \Delta v + (o_k - y_k) \otimes h_k$$

Then,

$$V_{new} = Vol_d + \alpha\Delta v$$
$$W_{new} = Wol_d + \alpha\Delta w$$
$$U_{new} = Uol_d + \alpha\Delta u$$

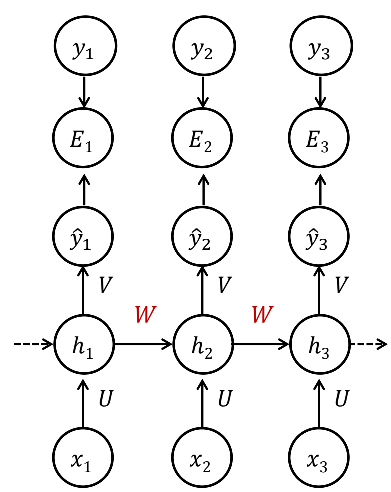
α: learning rate

Exploding and Vanishing Gradient Problem

- In RNN, we repeatedly multiply W along with a input sequence

$$h^t = \tanh(Uxt + Wh^{t-1})$$

- The recurrence multiplication can result in difficulties called exploding and vanishing gradient problem



Exploding and Vanishing Gradient Problem

- For example, we can think of simple RNN with lacking inputs $\boldsymbol{x}$

$$h^t = Wh^{t-1}$$

- It can be simplified to

$$h^t = W^t h^0$$

- If W^t has an Eigen decomposition

$$W^t = (V \operatorname{diag}(\lambda) V^{-1})^t = V \operatorname{diag}(\lambda^t) V^{-1}$$

Exploding and Vanishing Gradient Problem

$$W^t = (V \operatorname{diag}(\lambda) V^{-1})^t = V \operatorname{diag}(\lambda^t) V^{-1}$$

- Any eigenvalues λ_i that are not near an absolute value of 1 will either
  - explode if they are greater than 1 in magnitude
  - vanish if they are less than 1 in magnitude
- The gradients through such a graph are also scaled according to $\operatorname{diag}(\lambda^t)$

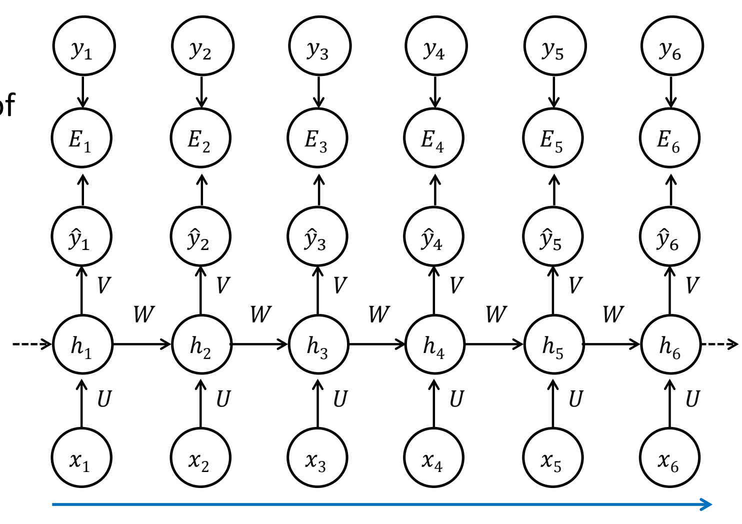
Exploding and Vanishing Gradient Problem

$$W^t = (V \, \mathrm{diag}(\lambda) V^{-1})^t = V \, \mathrm{diag}(\lambda^t) V^{-1}$$

- Whenever the model is able to represent long-term dependencies, the gradient of a long-term interaction has exponentially smaller magnitude than the gradient of a short-term interaction
- That is, it is not impossible to learn, but that it might take a very long time to learn long-term dependencies:
- Because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies

Solution1: Truncated BPTT

- Run forward as it is, but run the backward in the chunk of the sequence instead of the whole sequence



Solution2: Gating mechanism (LSTM;GRU)

- Add gates to produce paths where gradients can flow more constantly in longer-term without vanishing nor exploding
- We'll see in next chapter

Outline

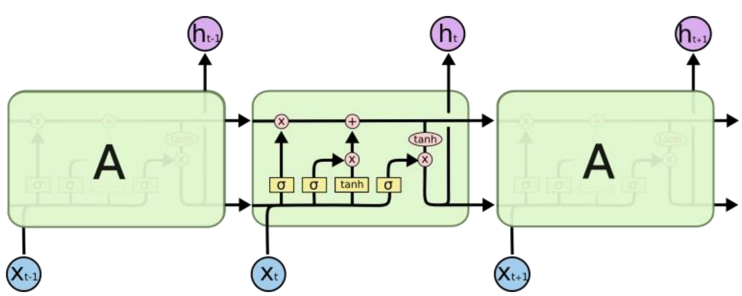
- RNN
- LSTM
- GRU
- Tasks with RNN
- Software Packages

Long Short-term Memory (LSTM)

- Capable of modeling longer term dependencies by having memory cells and gates that controls the information flow along with the memory cells

Long Short-term Memory (LSTM)

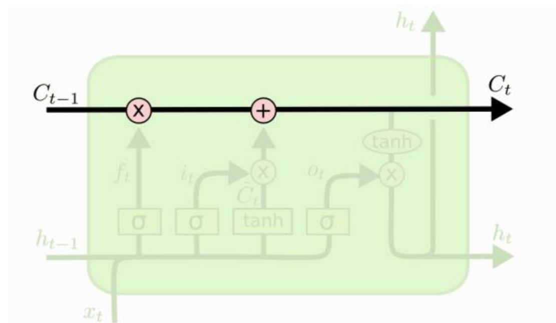
- Capable of modeling longer term dependencies by having memory cells and gates that controls the information flow along with the memory cells



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

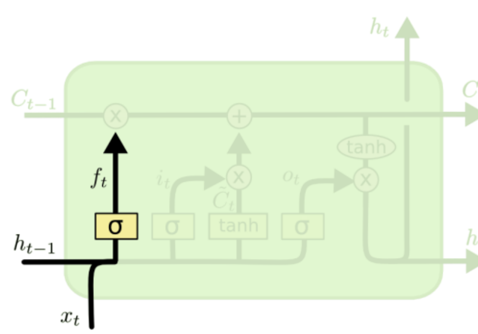
Long Short-term Memory (LSTM)

- The contents of the memory cells C_t are regulated by various gates:
  - Forget gate f_t
  - Input gate i_t
  - Reset gate r_t
  - Output gate o_t
- Each gates are composed of affine transformation with Sigmoid activation function



Forget Gate

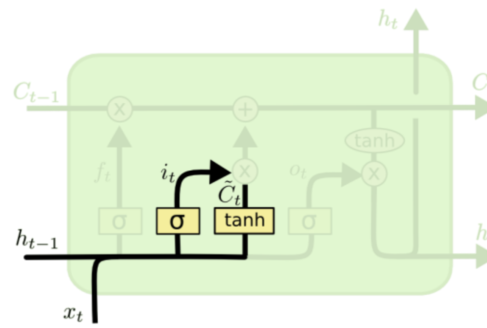
- It determines how much contents from previous cell C_{t-1} will be erased (we will see how it works in next a few slides)
- Linear transformation of concatenated previous hidden states and input are followed by Sigmoid function
- The sigmoid generates values 0 and 1:
  - 0 : completely remove info in the dimension
  - 1 : completely keep info in the dimension



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

New Candidate Cell and Input Gate

- New candidate cell states $\tilde{C}_t$ are created as a function of h_{t-1} and x_t
- Input gates i_t decides how much of values of the new candidate cell states $\tilde{C}_t$ are combined into the cell states

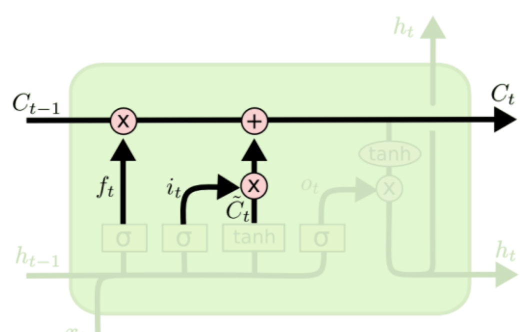


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \qquad \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Update Cell States

- The previous cell states C_{t-1} are updated to the new cell states C_t by using the input and forget gates with new candidate cell states

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



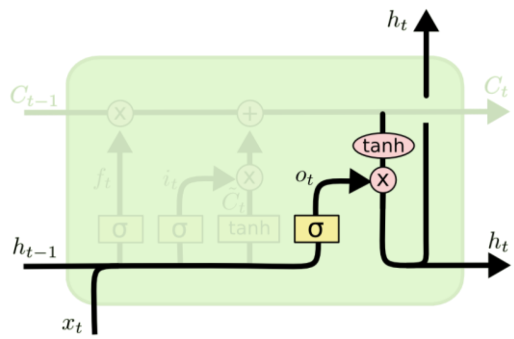
Generate Output

- Output will be based on cell state C_t with filter from output gate o_t
- The output gate o_t decides which part of cell state C_t will be in the output

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

- Then the final output is generated from tanh-ed cell states filtered by o_t

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



Outline

- RNN
- LSTM
- GRU
- Tasks with RNN
- Software Packages

Gated Recurrent Unit (GRU)

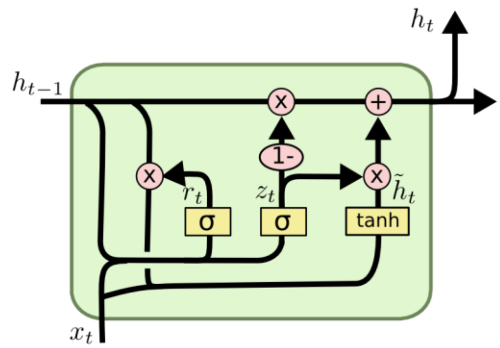
- Simplify LSTM by combining forget and input gate into update gate z_t
- z_t controls the forgetting factor and the decision to update the state unit

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

$$\widetilde{h_t} = \tanh(W \cdot [r_t * h_{t-1}, x_t] + b)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \widetilde{h_t}$$



Gated Recurrent Unit (GRU)

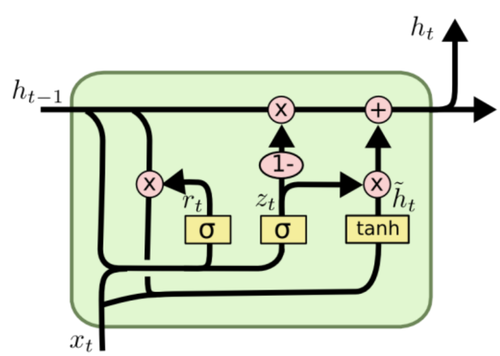
- Reset gates r_t control which parts of the state get used to compute the next target state
- It introduces additional nonlinear effect in the relationship between past state and future state

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

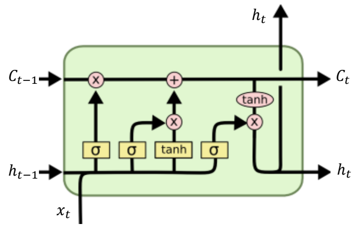
$$\widetilde{h_t} = \tanh(W \cdot [r_t * h_{t-1}, x_t] + b)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \widetilde{h_t}$$



Comparison LSTM and GRU

LSTM



$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$

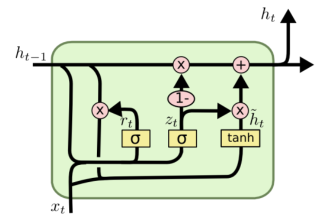
$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$

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

$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$

$h_t = o_t * \tanh(C_t)$

GRU



$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$

$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$

$\widetilde{h_t} = \tanh(W \cdot [r_t * h_{t-1}, x_t] + b)$

$h_t = (1 - z_t) * h_{t-1} + z_t * \widetilde{h_t}$

Comparison LSTM and GRU

- Greff, et al. (2015) compared LSTM, GRU and several variants on thousands of experiments and found that none of the variants can improve upon the standard LSTM architecture significantly, and demonstrate the forget gate and the output activation function to be its most critical components.

- Greff, et al. (2015): LSTM: A Search Space Odyssey

Outline

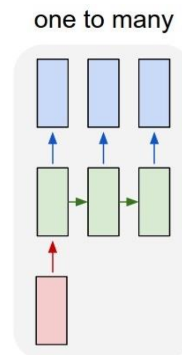
- RNN
- LSTM
- GRU
- Tasks with RNN
  - One-to-Many
  - Many-to-One
  - Many-to-Many
  - Encoder-Decoder Seq2Seq Model
  - Attention Mechanism
  - Bidirectional RNN
- Software Packages

Tasks with RNN

- One of strengths of RNN is flexibility in modeling any task with any data type
- By composing the input and output as either sequence or non-sequence data, you can model many different tasks
- Here are some of the examples:

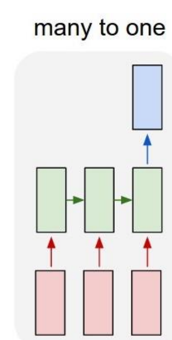
One-to-Many

- Input: non-sequence vector / Output: sequence of vectors
- After the first time step, hidden states are updated with only previous step's hidden states
- Example: Sentence generation given image
  - Typically the input image is processed with CNN to generate a real-valued vector representation
  - During training, true target is a sentence (sequence of words) about the training image



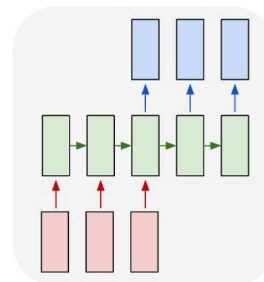
Many-to-One

- Input: sequence of vectors / Output: non-sequence vector
- Only the last time step's hidden states is used as the output
- Example: Sequence classification, sentiment classification



Many-to-Many

- Input: sequence of vectors / Output: sequence of vectors
- Generate a sequence given another sequence
- Example: Machine translation
  - Especially parameterized by what is called "Encoder-Decoder" model

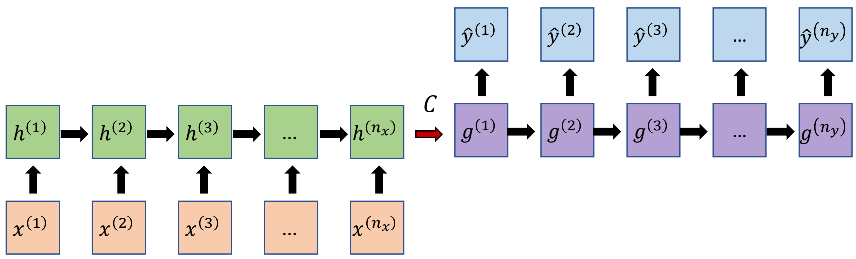


Encoder-Decoder (Seq2Seq) Model

- Key idea:
  - Encoder RNN generates a fixed-length context vector C from input sequence $\boldsymbol{X} = (x^{(1)}, \dots, x^{(n_x)})$
  - Decoder RNN generates an output sequence $\boldsymbol{Y} = (y^{(1)}, \dots, y^{(n_y)})$ conditioned on the context C

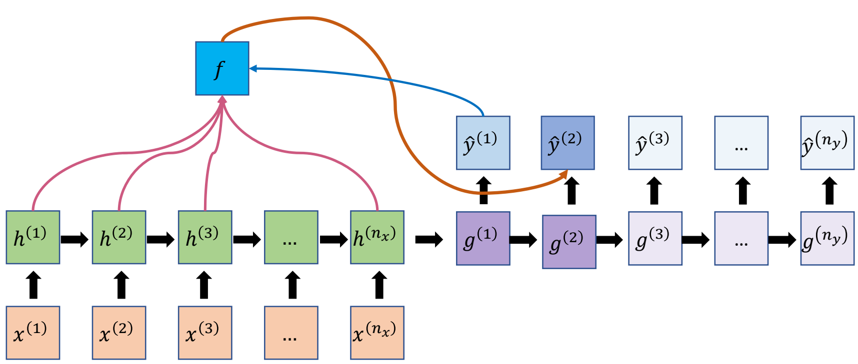
- The two RNNs are trained jointly to maximize the average of $\log P(y^{(1)}, \dots, y^{(n_y)} | x^{(1)}, \dots, x^{(n_x)})$ over all sequence in training set

Encoder-Decoder (Seq2Seq) Model



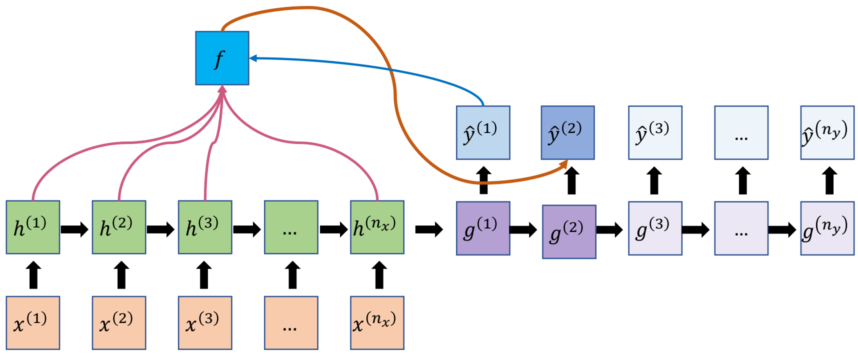
- Typically, the last hidden states of encoder RNN $h^{(n_x)}$ is used as context C
- But when the context C has smaller dimension or lengths of sequences are longer, C can be a bottleneck; it cannot properly summarize the input sequence

Attention Mechanism



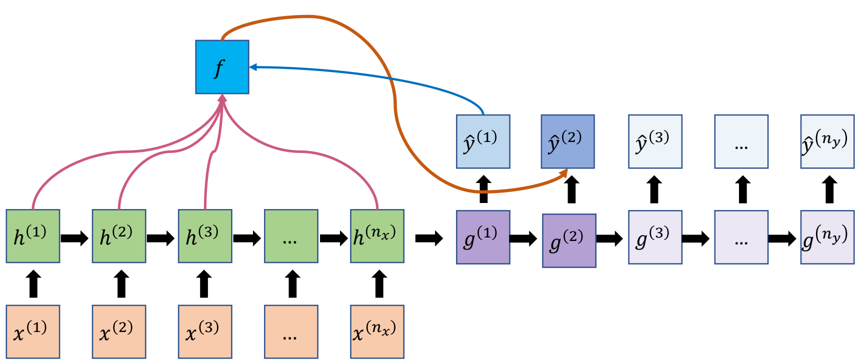
- Attention mechanism learns to associate hidden states of input sequence to generation of each step of the target sequence

Attention Mechanism



- The association is modeled as additional feed-forward network f gets input sequence's hidden states and predicted target on previous time step

Attention Mechanism



- In f, Softmax is used to generate the weights among the hidden states of the input sequences

Outline

- RNN
- LSTM
- GRU
- Encoder-Decoder Seq2Seq Model
- **Bidirectional RNN**
- Software Packages

Bidirectional RNN

- In some applications, such as speech recognition or machine translation, dependencies over time not only lie in forward in time but also lie in backward in time

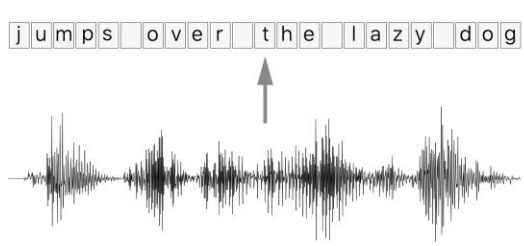
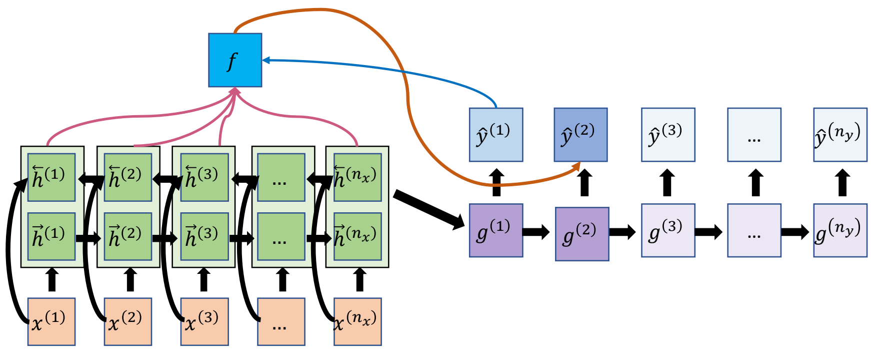


Image: https://distill.pub/2017/ctc/

Bidirectional RNN

- To model these, two RNNs are trained together forward RNN and backward RNN
- Each time step's hidden states from both RNNs are concatenated to form a final output



Outline

- RNN
- LSTM
- GRU
- Tasks with RNN
- Software Packages

Software Packages for RNN

- Many recent Deep Learning packages are supporting RNN/LSTM/GRU:

- PyTorch: https://pytorch.org/docs/stable/nn.html#recurrent-layers
- TensorFlow: https://www.tensorflow.org/tutorials/sequences/recurrent
- Caffe2: https://caffe2.ai/docs/RNNs-and-LSTM-networks.html
- Keras: https://keras.io/layers/recurrent/