



SDU Summer School

# Deep Learning

Summer 2018

## Recurrent Neural Networks



# Recurrent Neural Networks

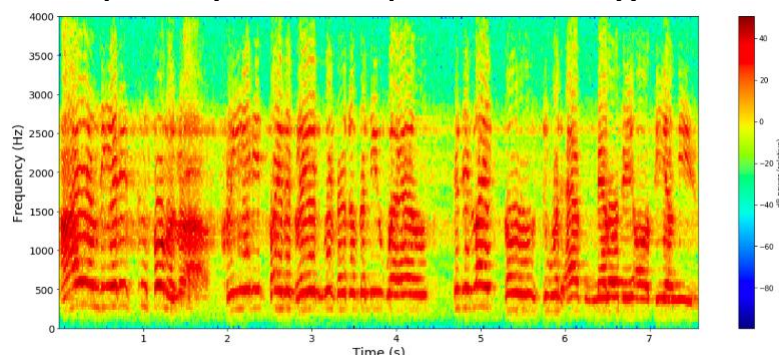
- **Introduction**
- **Unfolding a Graph**
- **Recurrent Neural Networks**
- **Long Term Memory**
  - Leaky Units
  - Long-Short Term Memory
- **Working with Texts in Keras**

# RNNs process sequential data

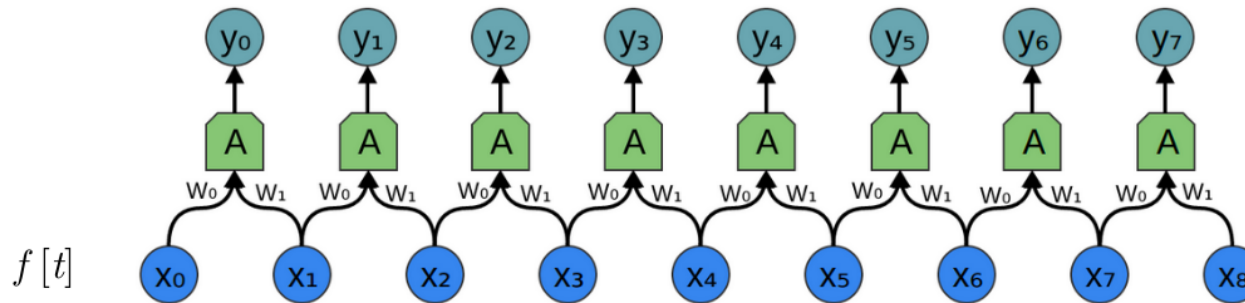
- Recurrent Neural Networks are a family of neural networks for processing sequential data
- Just as CNN is specialized for processing grid of values, e.g., image
  - RNN is specialized for processing a sequence of values  $x^{(1)}, \dots, x^{(\tau)}$
- Just as CNNs can readily scale images with large width/height and process variable size images
  - RNNs can scale to much longer sequences than would be practical for networks without sequence-based specialization
  - RNNs can also process variable-length sequences

# Examples of Sequential Data and Tasks

- Sequence data: sentences, speech, stock market, signal data
- Sequence-to-sequence Tasks
  - Speech recognition
    - decompose sound waves into frequency and amplitude using Fourier transforms yielding a spectrogram
  - Named Entity Recognition
    - Input: Jim bought 300 shares of Acme Corp. in 2006
    - NER: [Jim]<sub>Person</sub> bought 300 shares of [Acme Corp.]<sub>Organization</sub> in [2006]<sub>Time</sub>
- Sequence-to-symbol
  - Sentiment
  - Speaker recognition



# Recall: 1-D Convolution



Kernel  $g(t)$ :  
 $[\dots, 0, w_1, w_0, 0\dots]$

Equations for outputs of this network:

$$y_0 = \sigma(W_0x_0 + W_1x_1 - b)$$

$$y_1 = \sigma(W_0x_1 + W_1x_2 - b) \text{ etc. upto } y_8$$

Note that kernel gets flipped in convolution

We can also write the equations in terms of elements of a general  $8 \times 8$  weight matrix  $W$  as:

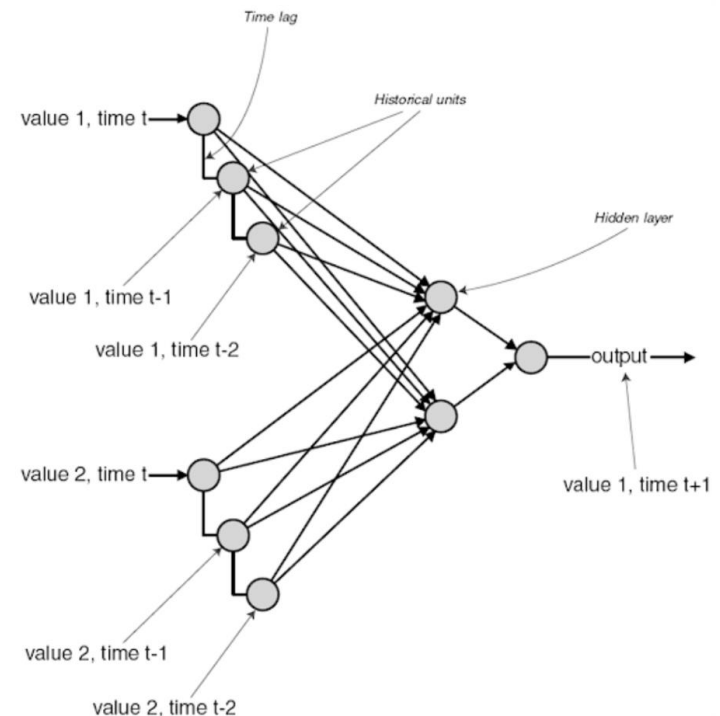
$$y_0 = \sigma(W_{0,0}x_0 + W_{0,1}x_1 + W_{0,2}x_2\dots)$$

$$y_1 = \sigma(W_{1,0}x_0 + W_{1,1}x_1 + W_{1,2}x_2\dots)$$

where  $W = \begin{bmatrix} w_0 & w_1 & 0 & 0 & \dots \\ 0 & w_0 & w_1 & 0 & \dots \\ 0 & 0 & w_0 & w_1 & \dots \\ 0 & 0 & 0 & w_0 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$

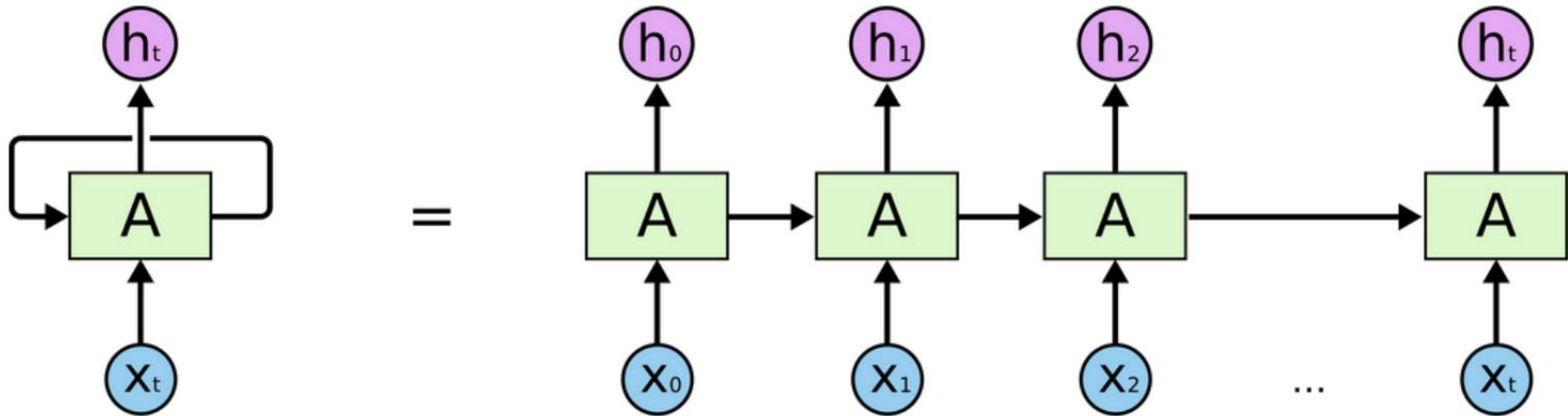
# Time Delay Neural Networks

- Time-delay neural networks perform convolution across 1-D temporal sequence
  - Convolution operation allows a network to share parameters across time, but is shallow
    - Each member of output is dependent upon a small no. of neighboring members of the input
    - Parameter sharing manifests in the application of the same convolutional kernel at each time step
- A TDNN remembers the previous few training examples and uses them as input into the network.



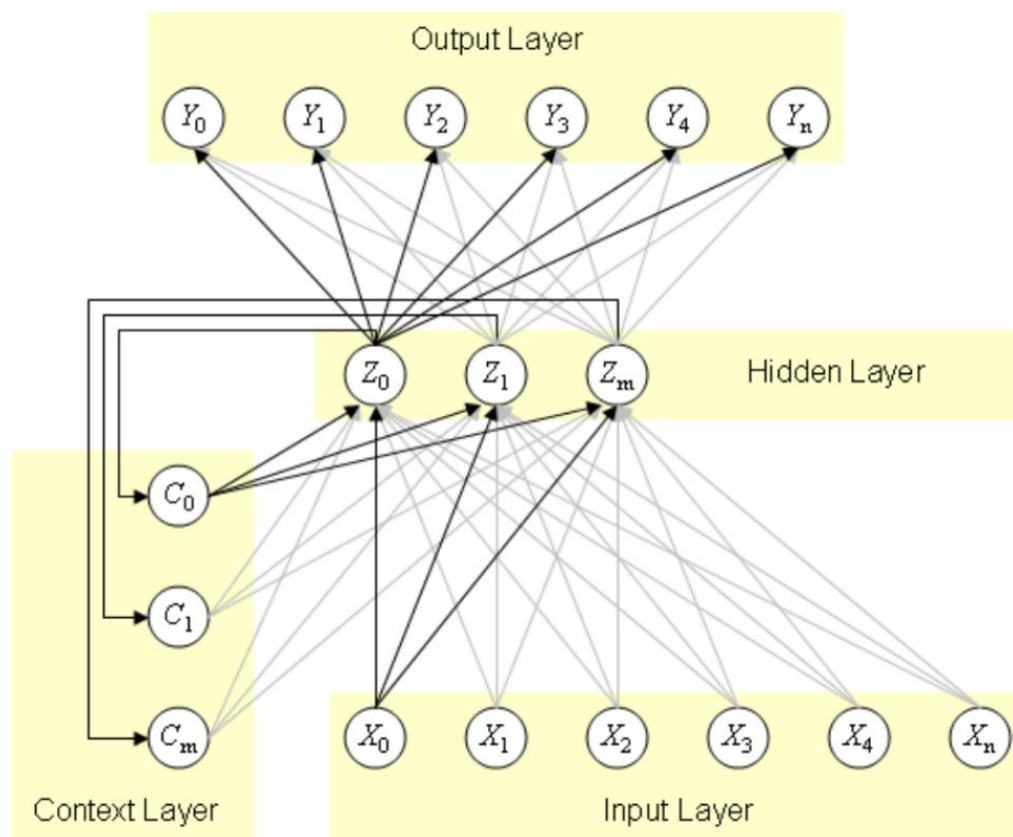
# RNN vs. TDNN

- RNNs share parameters in a different way
  - Each member of output is a function of previous members of output
  - Each output produced using same update rule applied to previous outputs
  - This recurrent formulation results in sharing of parameters through a very deep computational graph
- An unrolled RNN



# RNN as a network with cycles

- An RNN is a class of neural networks where connections between units form a directed cycle
- This creates an internal state of the network which allows it to exhibit dynamic temporal behavior
- The internal memory can be used to process arbitrary sequences of inputs



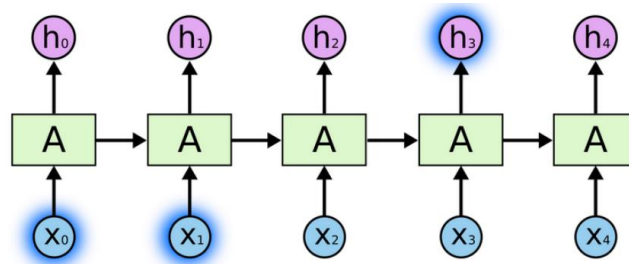


# RNNs share same weights across Time Step

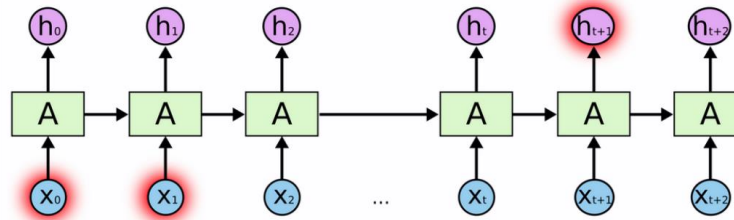
- To go from multi-layer networks to RNNs:
  - Need to share parameters across different parts of a model
  - Separate parameters for each value of input cannot generalize to sequence lengths not seen during training
  - Share statistical strength across different sequence lengths and across different positions in time
- Sharing important when information can occur at multiple positions in the sequence
  - Given “I went to Nepal in 1999 ” and “In 1999, I went to Nepal ”, an ML method to extract year, should extract 1999 whether in position 6 or 2
  - A feed-forward network that processes sentences of fixed length would have to learn all of the rules of language separately at each position
  - An RNN shares the same weights across several time steps

# Problem of Long-Term Dependencies

- Easy to predict last word in “the clouds are in the sky”
  - When gap between relevant information and place that it’s needed is small, RNNs can learn to use the past information



- “I grew up in France... I speak fluent French.”
  - Large gap between relevant information and point where it is needed



- In principle RNNs can handle it, but fail in practice
  - LSTMs offer a solution

# RNN operating on a sequence

- RNNs operate on a sequence that contain vector  $x^{(t)}$  with time step index  $t$ , ranging from 1 to  $\tau$ 
  - Sequence  $x^{(1)}, \dots, x^{(\tau)}$
  - RNNs operate on minibatches of sequences of length  $\tau$
- Some remarks about sequences
  - The steps need not refer to passage of time in the real world
  - RNNs can be applied in two-dimensions across spatial data such as image
  - Even when applied to time sequences, network may have connections going backwards in time, provided entire sequence is observed before it is provided to network



# Recurrent Neural Networks

- Introduction
- **Unfolding a Graph**
- Recurrent Neural Networks
- Long Term Memory
  - Leaky Units
  - Long-Short Term Memory

# Unfolding Computational Graphs

- Recall: A Computational Graph is a way to formalize the structure of a set of computations
  - Such as mapping inputs and parameters to outputs and loss
- We can unfold a recursive or recurrent computation into a computational graph that has a repetitive structure
  - Corresponding to a chain of events
- Unfolding this graph results in sharing of parameters across a deep network structure

# Example of Unfolding a Recurrent Equation

- Classical form of a dynamical system is

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \boldsymbol{\theta})$$

- where  $\mathbf{s}^{(t)}$  is called the state of the system
  - Equation is recurrent because the definition of  $\mathbf{s}$  at time  $t$  refers back to the same definition at time  $t - 1$
- For a finite no. of time steps  $\tau$ , the graph can be unfolded by applying the definition  $\tau - 1$  times

- E.g, for  $\tau = 3$  time steps we get

$$\mathbf{s}^{(3)} = f(\mathbf{s}^{(2)}; \boldsymbol{\theta}) = f(f(\mathbf{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta})$$

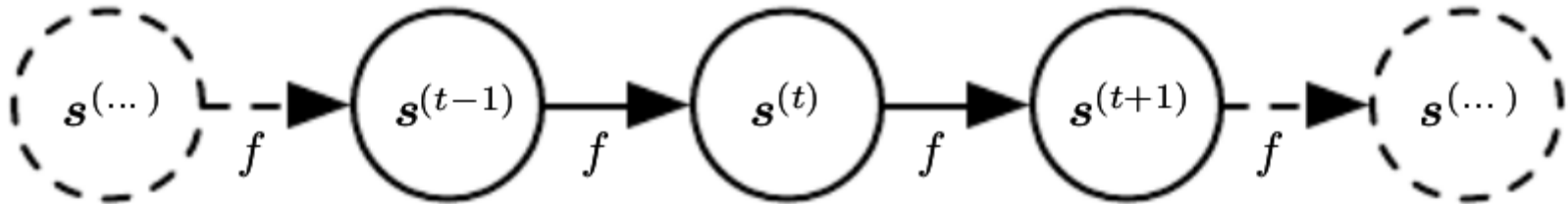
- Unfolding equation by repeatedly applying the definition in this way has yielded expression without recurrence
  - $\mathbf{s}^{(1)}$  is ground state, the other states are calculated by applying  $f$
  - Can now be represented by a traditional acyclic computational graph

# Unfolded dynamical system

- The classical dynamical system described by

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \boldsymbol{\theta})$$

is illustrated as an unfolded computational graph:



- Each node represents state at some time  $t$
- Function  $f$  maps state at time  $t$  to the state at  $t + 1$
- The same parameters (the same value of  $\theta$  used to parameterize  $f$ ) are used for all time steps

# Dynamical system driven by external signal

- As another example, consider a dynamical system driven by external (input) signal  $\mathbf{x}^{(t)}$

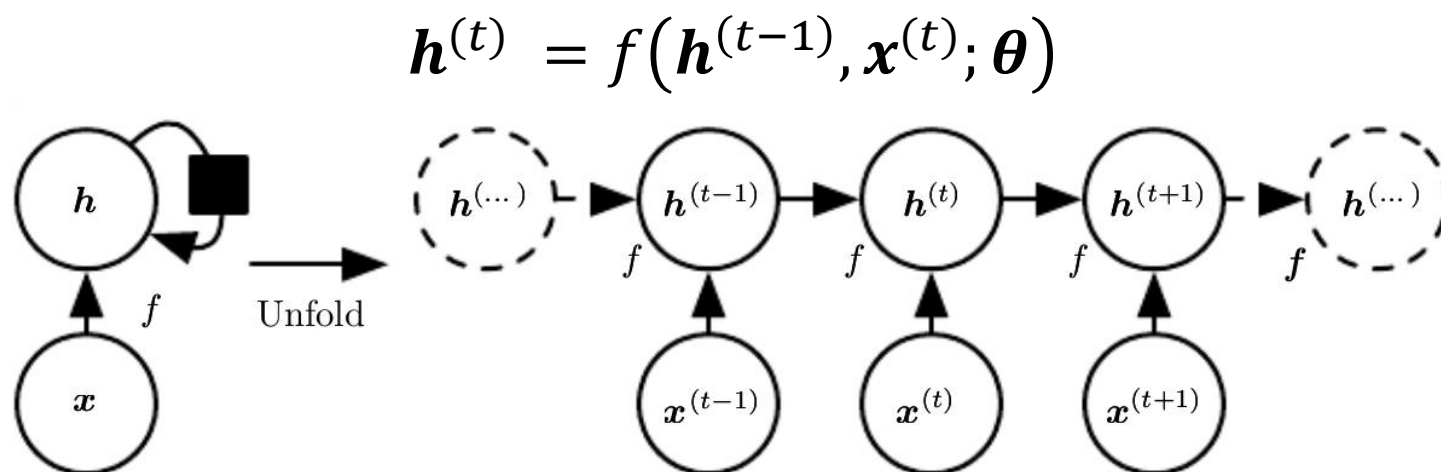
$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta})$$

- State now contains information about the whole past input sequence
- Recurrent neural networks can be built in many ways
  - Much as almost any function is a feedforward neural network, any function involving recurrence can be considered to be a recurrent neural network



# Defining values of hidden units in RNNs

- Many recurrent neural nets use same equation (as dynamical system with external input) to define values of hidden units
  - To indicate that the state is hidden rewrite using variable  $\mathbf{h}$  for state



- Typical RNNs have extra architectural features such as output layers that read information out of state  $\mathbf{h}$  to make predictions

# Predicting the Future from the Past

- When RNN is required to perform a task of predicting the future from the past, network typically learns to use  $\mathbf{h}^{(t)}$  as a lossy summary of the past sequence of inputs upto time point  $t$
- The summary is in general lossy since it maps a sequence of arbitrary length  $(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)})$  to a fixed length vector  $\mathbf{x}^{(1)}$
- Depending on criterion, summary keeps some aspects of past sequence more precisely than other aspects
  - RNN used in statistical language modeling, typically to predict next word from past words
  - it may not be necessary to store all information upto time  $t$  but only enough information to predict rest of sentence

# Process of Unfolding

- We can represent unfolded recurrence after  $t$  steps with a function  $g^{(t)}$ :

$$\mathbf{h}^{(t)} = g^{(t)}(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)}; \boldsymbol{\theta}) = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta})$$

- Function  $g^{(t)}$  takes in whole past sequence  $(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)})$  as input and produces the current state but the unfolded recurrent structure allows us to factorize  $g^{(t)}$  into repeated application of a function  $f$
- This has several advantages

# Unfolding process allows learning a single model

- The unfolding process introduces two major advantages:
  1. Regardless of sequence length, learned model has same input size
    - because it is specified in terms of transition from one state to another state rather than specified in terms of a variable length history of states
  2. Possible to use same function  $f$  with same parameters at every step
- These two factors make it possible to learn a single model  $f$ 
  - that operates on all time steps and all sequence lengths
  - rather than needing separate model  $g^{(t)}$  for all possible time steps
- Learning a single shared model allows:
  - Generalization to sequence lengths that did not appear in the training
  - Allows model to be estimated with far fewer training examples than would be required without parameter sharing



# Recurrent Neural Networks

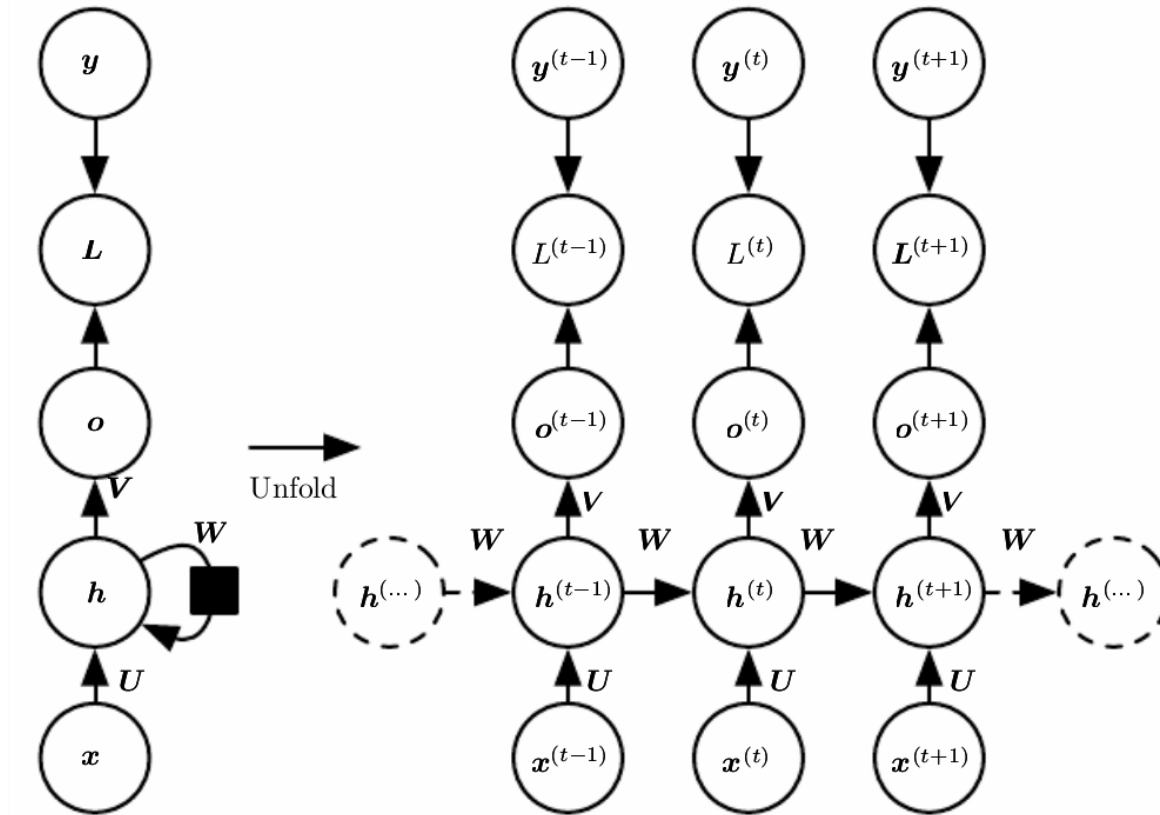
- Introduction
- Unfolding a Graph
- **Recurrent Neural Networks**
- Long Term Memory
  - Leaky Units
  - Long-Short Term Memory

# Three design patterns of RNNs

- Design 1: Output: each time step; Recurrence: hidden units
- Design 2: Output: each time step; Recurrence: output units
- Design 3: Output: one at the end; Recurrence: hidden units

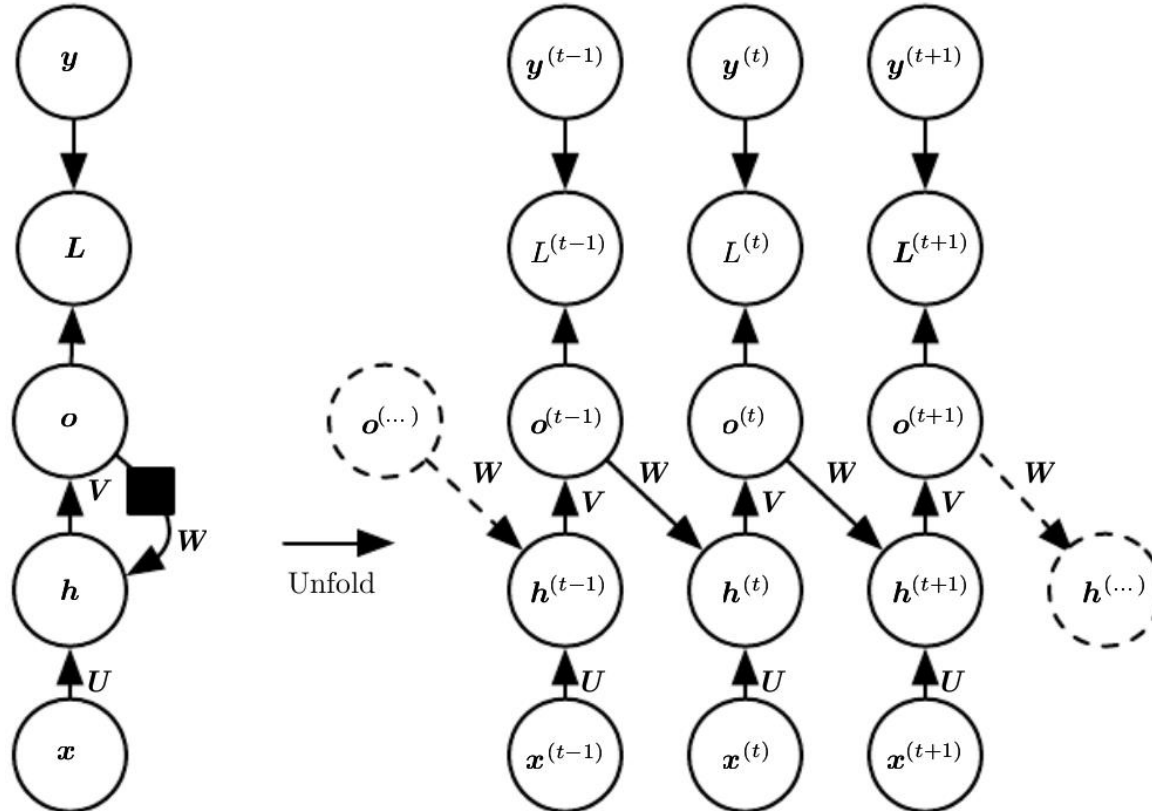
# Three design patterns of RNNs

- **Design 1: Output: each time step; Recurrence: hidden units**
- Design 2: Output: each time step; Recurrence: output units
- Design 3: Output: one at the end; Recurrence: hidden units



# Three design patterns of RNNs

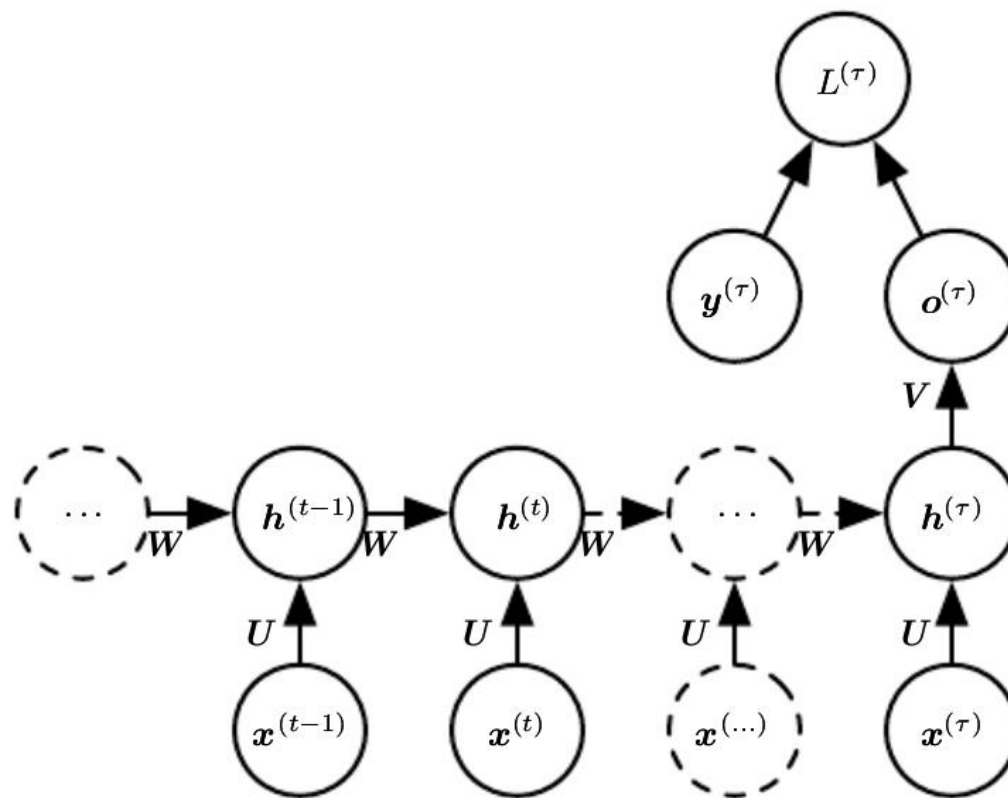
- Design 1: Output: each time step; Recurrence: hidden units
- **Design 2: Output: each time step; Recurrence: output units**
- Design 3: Output: one at the end; Recurrence: hidden units





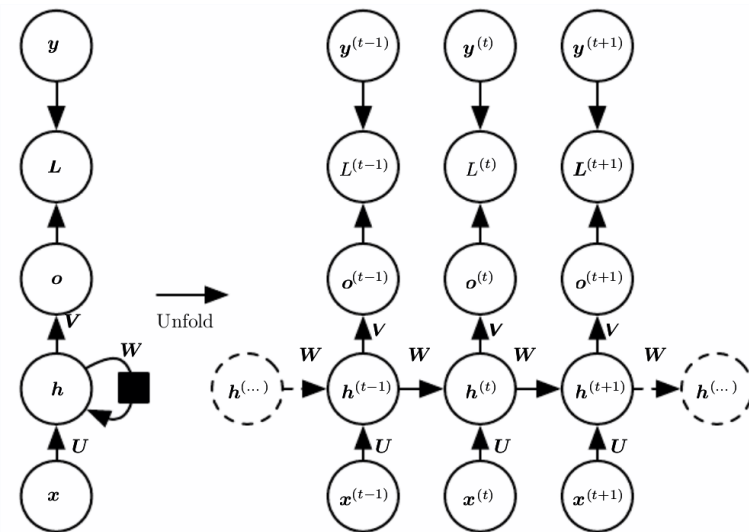
# Three design patterns of RNNs

- Design 1: Output: each time step; Recurrence: hidden units
- Design 2: Output: each time step; Recurrence: output units
- **Design 3: Output: one at the end; Recurrence: hidden units**



# Design 1: RNN with recurrence between hidden units

- Maps input sequence  $\mathbf{x}$  to output  $\mathbf{o}$  values
  - Loss  $L$  measures how far each  $\mathbf{o}$  is from the corresponding target  $\mathbf{y}$
  - With softmax outputs we assume  $\mathbf{o}$  is the unnormalized log probabilities
  - Loss  $L$  internally computes  $\mathbf{y} = \text{softmax}(\mathbf{o})$  and compares to target  $\mathbf{y}$
  - Let's assume we have tanh activation function
- Any function computable by a Turing Machine is computable by such an RNN of finite size



# Design 1: Forward Calculation

- We need to specify initial state  $\mathbf{h}^{(0)}$ , then for each time point:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)})$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$$

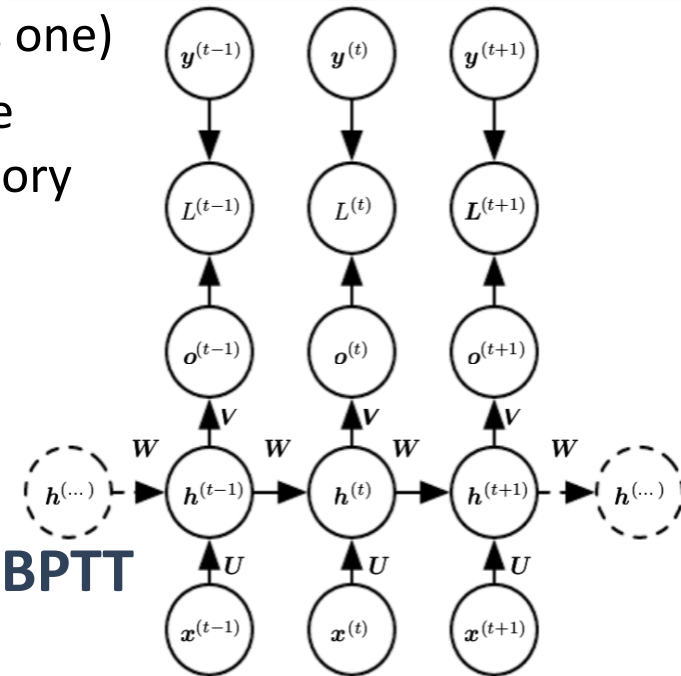
$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)})$$

- The Loss function can simply defined as the sum of the loss at each timepoint:

$$L = \sum_t L^{(t)} = - \sum_t \log p_{model}(\mathbf{y}^{(t)} | \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\})$$

# Design 1: Gradient Calculation

- Computing the gradient of this loss function with respect to the parameters is an expensive operation.
  - Requires forward propagation pass through the unrolled graph
  - followed by a backward pass
  - The runtime is  $O(\tau)$  and cannot be reduced by parallelization (each time step may only be computed after the previous one)
  - The forward pass must be stored until they are reused during the backward pass, so the memory cost is also  $O(\tau)$ .
- The back-propagation algorithm applied to the unrolled graph with  $O(\tau)$  cost is called **back-propagation through time** or **BPTT**





# Teacher Forcing

- Teacher forcing during training means
  - Instead of summing activations from incoming units (possibly erroneous)
  - Each unit sums correct teacher activations as input for the next iteration
- It emerges from the maximum likelihood criterion

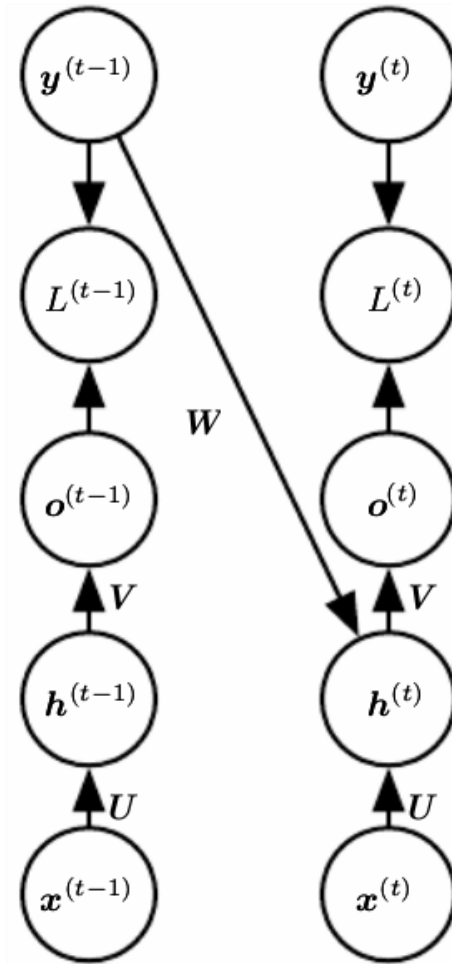
$$\log p(\mathbf{y}^{(1)}, \mathbf{y}^{(2)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) = \log p(\mathbf{y}^{(2)} | \mathbf{y}^{(1)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) + \log p(\mathbf{y}^{(1)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)})$$

- At time  $t = 2$ , model is trained to maximize cond. prob. of  $\mathbf{y}^{(2)}$  given both the  $\mathbf{x}$  sequence so far and the previous  $\mathbf{y}$  value from the training set.
  - We are using  $\mathbf{y}^{(1)}$  as teacher forcing, rather than only  $\mathbf{x}^{(1)}$ .
- Maximum likelihood specifies that during training, rather than feeding the model's own output back to itself, target values should specify what the correct output should be

# Illustration of Teacher Forcing

## Train time:

We feed the correct output  $y^{(t)}$  (from teacher) drawn from the training set as input to  $h^{(t+1)}$

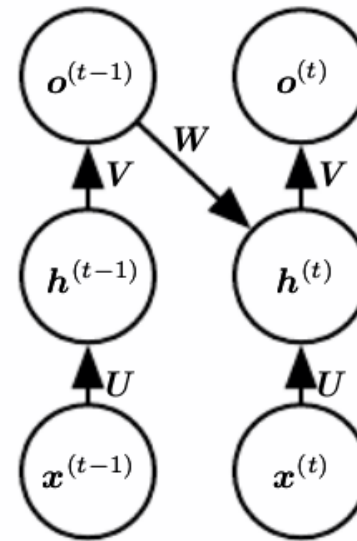


Train time

## Test time:

True output is not known.

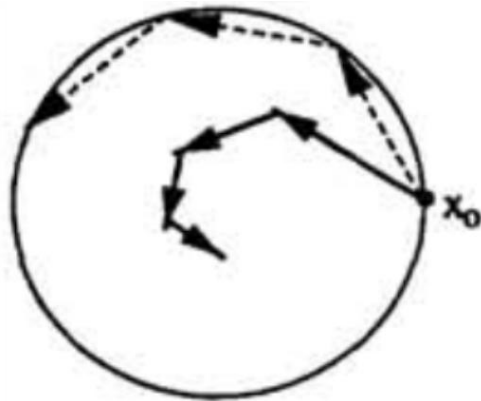
We approximate the correct output  $y^{(t)}$  with the model's output  $o^{(t)}$  and feed the output back to the model



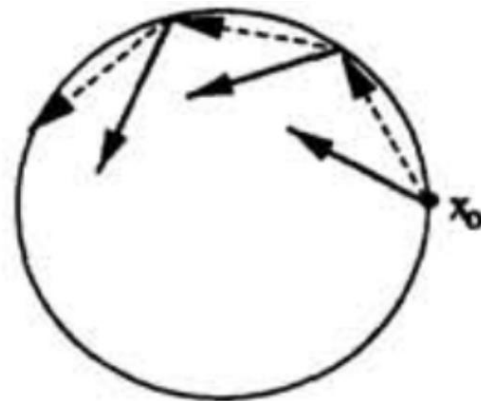
Test time

# Visualizing Teacher Forcing

- Imagine that the network is learning to follow a trajectory
- It goes astray (because the weights are wrong) but teacher forcing puts the net back on its trajectory



**a**



**b**





# Recurrent Neural Networks

- Introduction
- Unfolding a Graph
- Recurrent Neural Networks
- **Long Term Memory**
  - Leaky Units
  - Long-Short Term Memory

# Challenge of Long-Term Dependencies

- Neural network optimization face a difficulty when computational graphs become deep, e.g.,
  - Feedforward networks with many layers
  - RNNs that repeatedly apply the same operation at each time step of a long temporal sequence
- Gradients propagated over many stages tend to either vanish (most of the time) or explode (damaging optimization)
- The difficulty with long-term dependencies arise from exponentially smaller weights given to long-term interactions (involving multiplication of many Jacobians)

# Vanishing and Exploding Gradient Problem

- Suppose a computational graph consists of repeatedly multiplying by a matrix  $W$
- After  $t$  steps this is equivalent to multiplying by  $W^t$
- Suppose  $W$  has an eigendecomposition  $W = V\text{diag}(\lambda)V^{-1}$ :

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

- Any eigenvalues  $\lambda_i$  that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude and vanish if they are less than 1 in magnitude
- Vanishing gradients make it difficult to know which direction the parameters should move to improve cost
- Exploding gradients make learning unstable



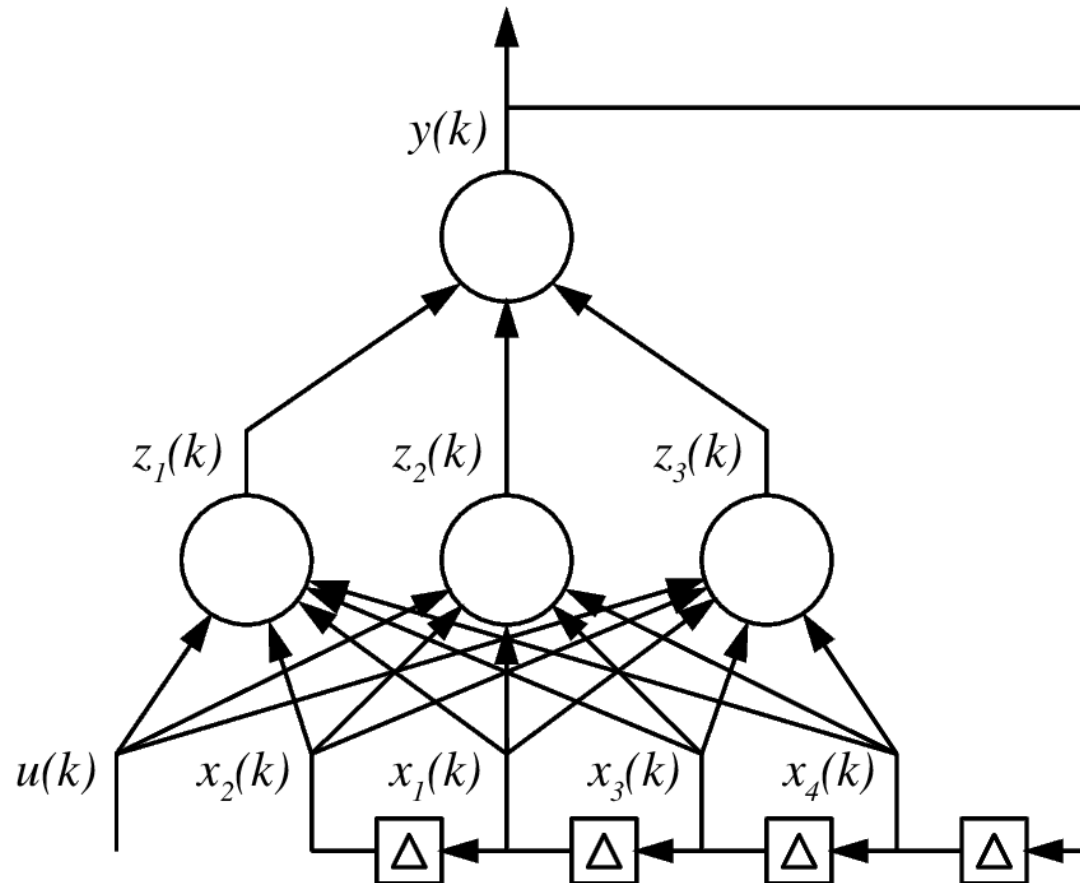
# Recurrent Neural Networks

- Introduction
- Unfolding a Graph
- Recurrent Neural Networks
- Long Term Memory
  - **Leaky Units**
  - Long-Short Term Memory

# Adding skip connections through time

- One way to obtain coarse time scales is to add direct connections from variables in the distant past to variables in the present
- In an ordinary RNN, recurrent connection goes from time  $t$  to time  $t + 1$ .
  - Gradients can vanish/explode exponentially wrt no. of time steps
- Introduce time delay of  $d$  to mitigate this problem
- Gradients diminish as a function of  $\frac{\tau}{d}$  rather than  $\tau$
- Allows learning algorithm to capture longer dependencies
  - Not all long-term dependencies can be captured this way

# Network with 4 output delays

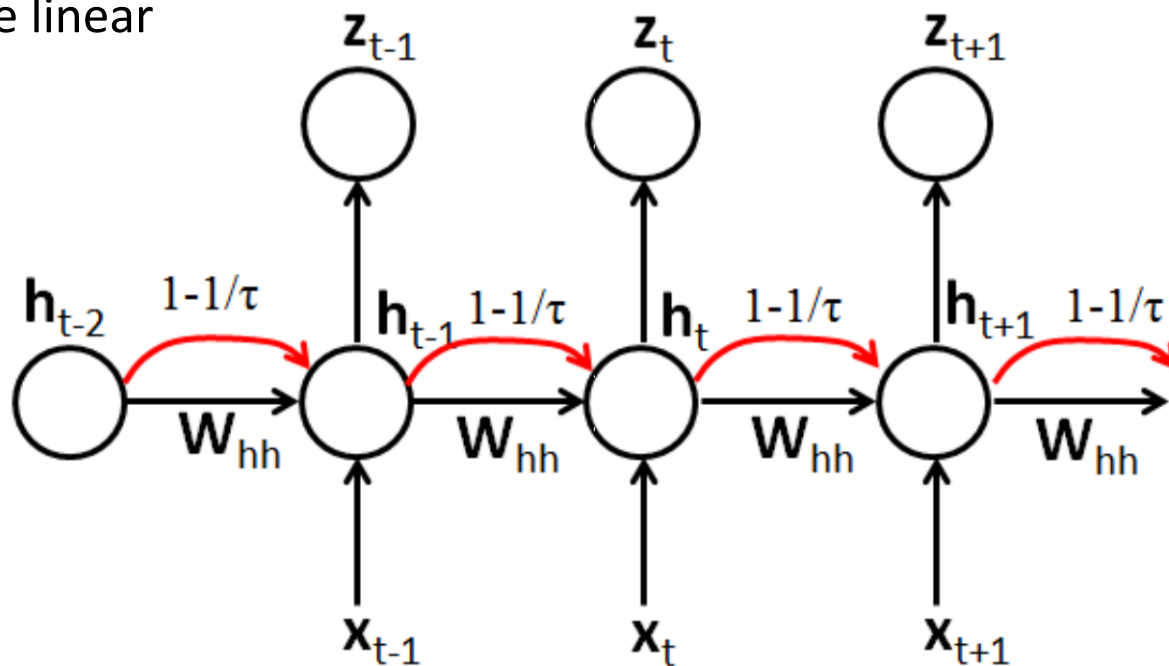


$$y(t) = \Psi \left( u(t), y(t-1), \dots, y(t-D) \right)$$

# More General

$$h_{t+1} = \left(1 - \frac{1}{\tau}\right) h_t + \frac{1}{\tau} g(W_{xh}x_t + W_{hh}h_t + b_h)$$

- The new value of the state  $h_{t+1}$  is a combination of linear and non-linear parts of  $h_t$
- The errors are easier to be back propagated through the paths of red lines, which are linear



# Leaky units and a spectrum of time scales

- Rather than an integer skip of  $d$  time steps, the effect can be obtained smoothly by adjusting a real-valued  $\alpha$
- Example: Running Average
  - Running average  $\mu^{(t)}$  of some value  $v^{(t)}$  is
$$\mu^{(t)} = \alpha\mu^{(t-1)} + (1 - \alpha)v^{(t)}$$
  - Called a linear self-correction
  - When  $\alpha$  is close to 1, running average remembers information from the past for a long time and when it is close to 0, information is rapidly discarded.
- Hidden units with linear self connections behave similar to running average. They are called leaky units.
- Can obtain product of derivatives close to 1 by having linear self-connections and a weight near 1 on those connections



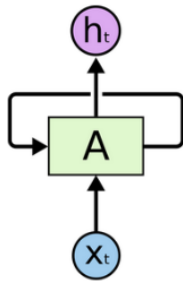


# Recurrent Neural Networks

- Introduction
- Unfolding a Graph
- Recurrent Neural Networks
- Long Term Memory
  - Leaky Units
  - **Long-Short Term Memory**

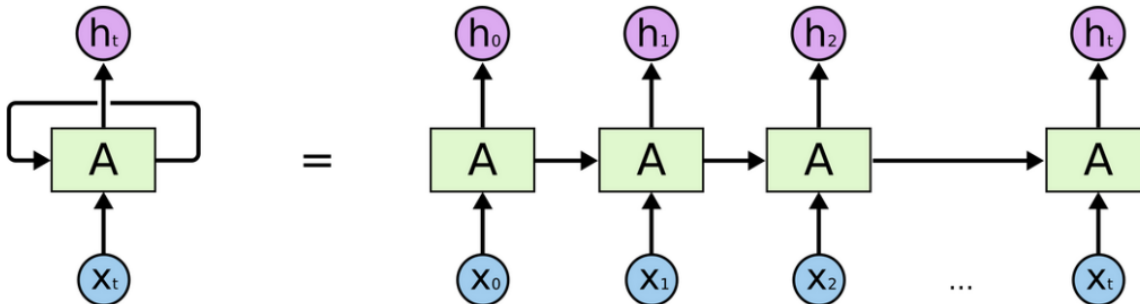
# From RNN to LSTM

## 1. RNNs have loops



A chunk of neural network A looks at some input  $x_t$  and outputs a value  $h_t$ . A loop allows information to be passed from one step of the network to the next

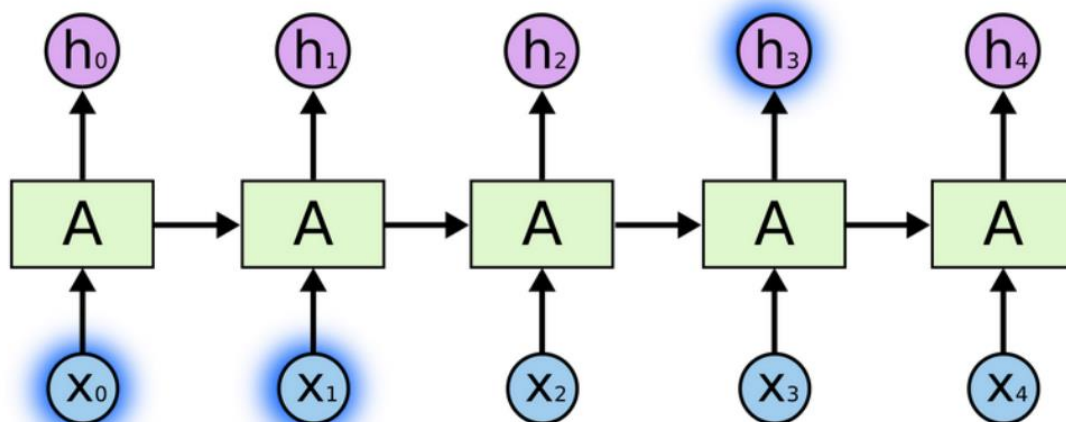
## 2. An unrolled RNN



Chain-like structure reveals that RNNs are intimately related to sequences and lists

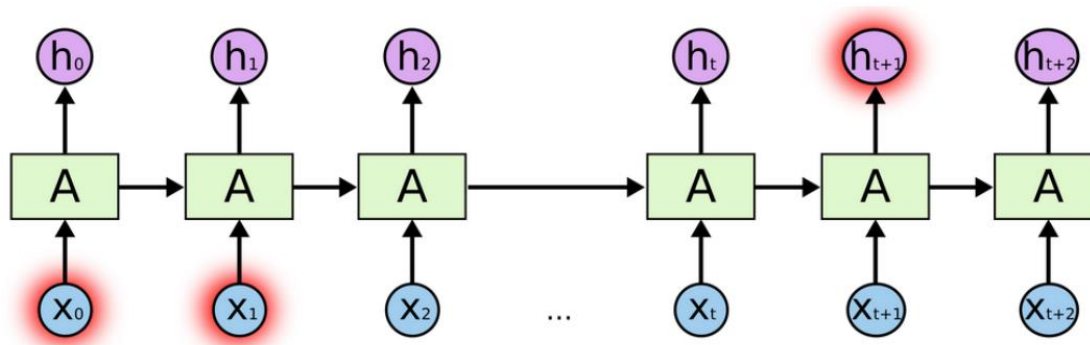
# Prediction with only recent previous information

- RNNs connect previous information to the present task
  - Previous video frames may help understand present frame
- Sometimes we need only look at recent previous information to predict
  - To predict the last word of “The clouds are in the sky” we don’t need any further context. It is obvious that the word is “sky”



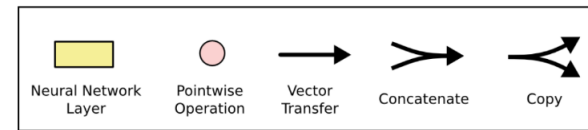
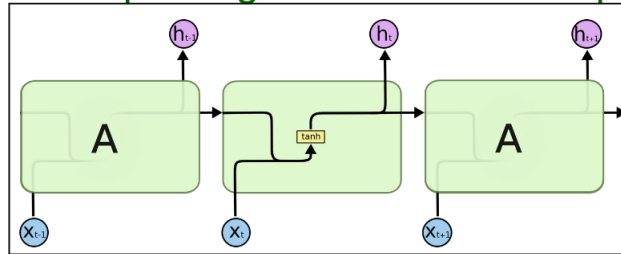
# Long-term Connections not Possible

- here are cases where we need more context
- To predict the last word in the sentence “I grew up in France....I speak fluent French”
  - Using only recent information suggests that the last word is the name of a language. But more distant past indicates that it is French
  - It is possible that the gap between the relevant information and where it is needed is very large

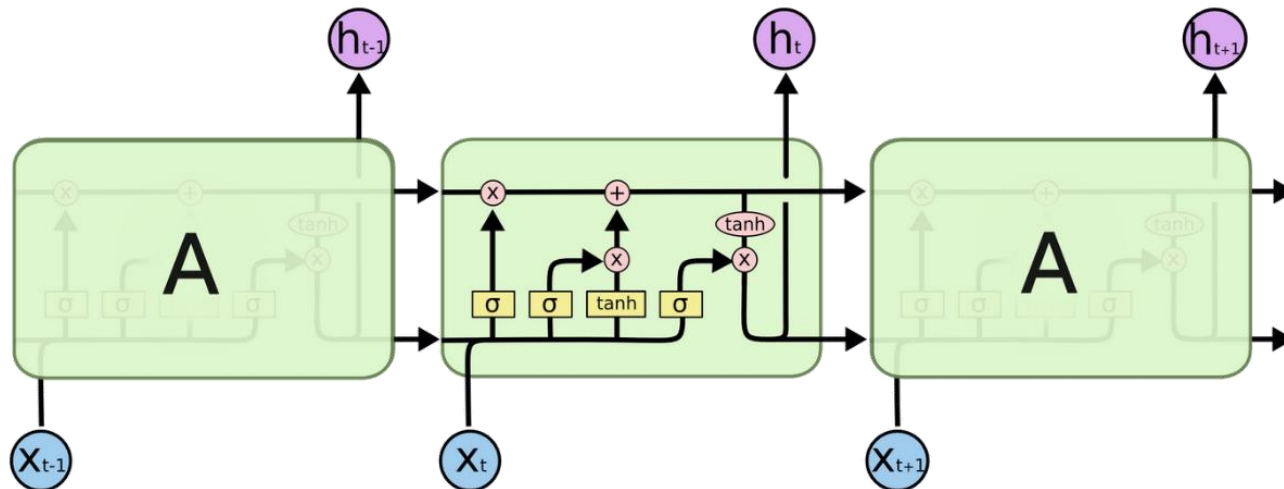


# Long Short Term Memory

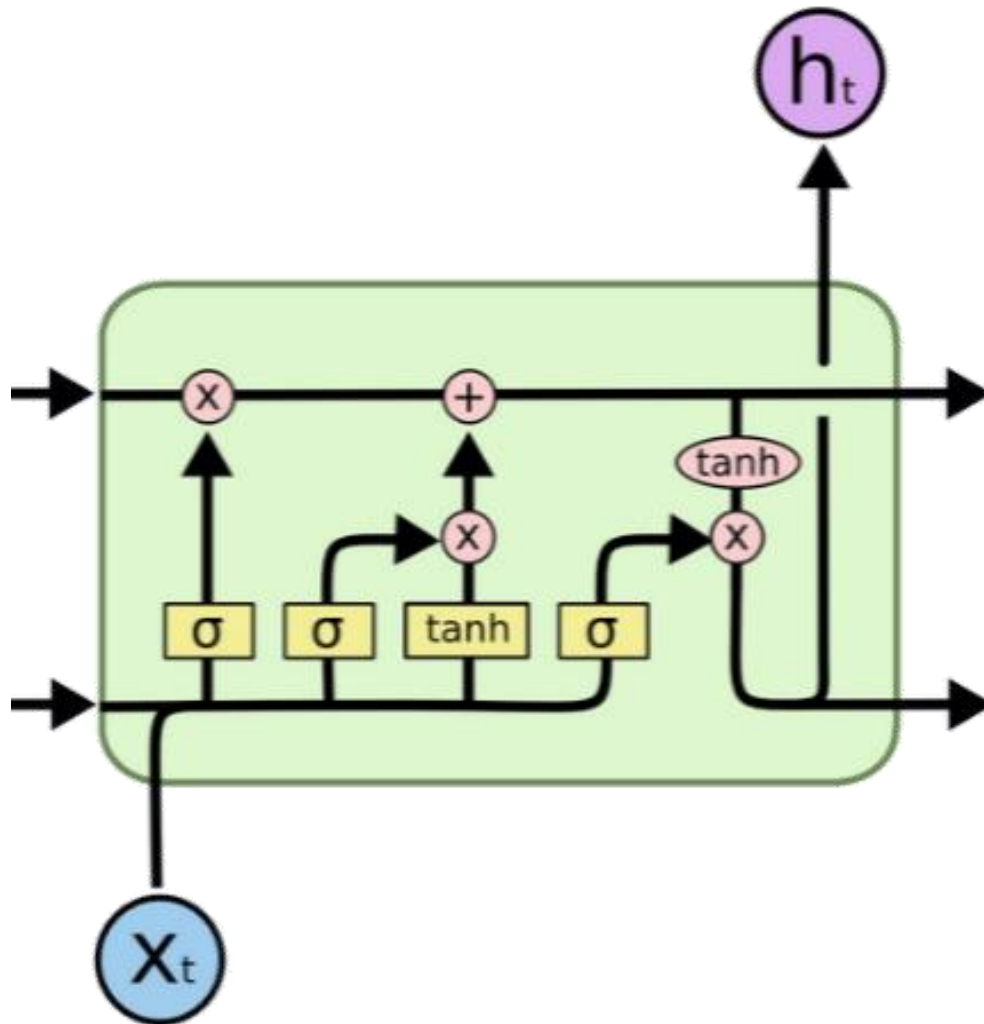
- Explicitly designed to avoid the long-term dependency problem
- RNNs have the form of a repeating chain structure
  - The repeating module has a simple structure such as tanh



- LSTMs also have a chain structure

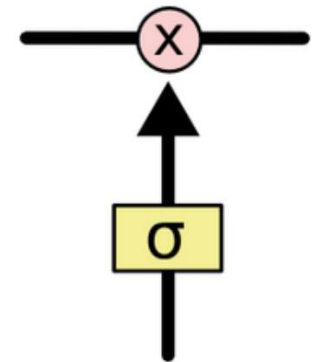
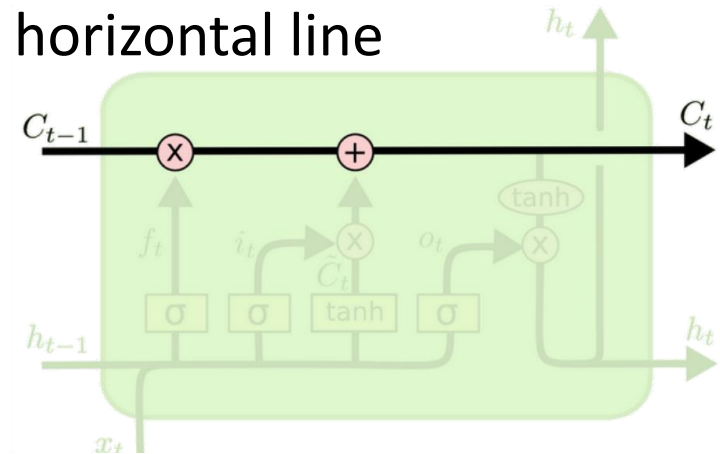


# LSTM Unit



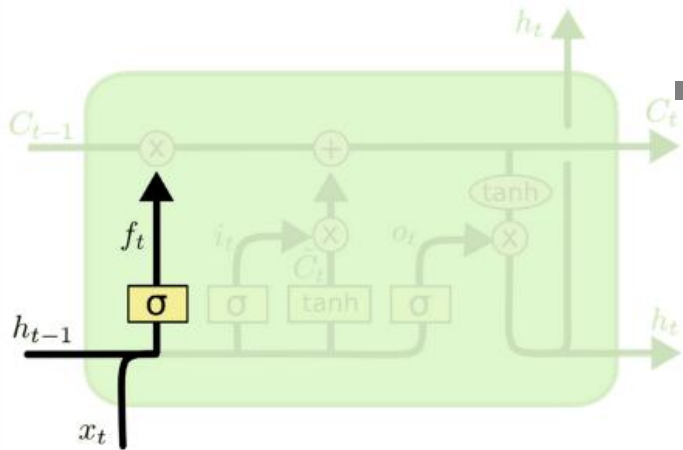
# Core idea behind LSTM

- The key to LSTM is the cell state,  $C_t$ , the horizontal line running through the top of the diagram
  - Like a conveyor belt
  - Runs through entire chain with minor interactions
  - LSTM does have the ability to remove/add information to cell state regulated by structures called gates
- Gates are an optional way to let information through
- Consist of a sigmoid and a multiplication operation
- Sigmoid outputs a value between 0 and 1
  - 0 means let nothing through
  - 1 means let everything through
- LSTM has three of these gates, to protect and control cell state



# Step-by-step LSTM walk through

- Example: predict next word based on previous ones
  - Cell state may include the gender of the present subject
- First step: information to throw away from cell state



- Called forget gate layer

- Uses  $h^{(t-1)}$  and  $x^{(t)}$  and outputs a number between 0 and 1 for each  $C^{(t-1)}$

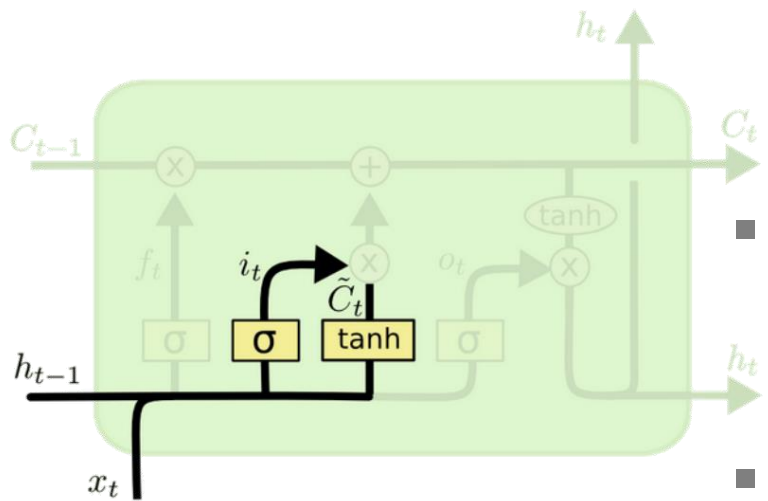
$$f^{(t)} = \sigma(\mathbf{W}_f \cdot [\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}] + b_f)$$

- In language model
  - Consider trying to predict the next word based on all previous ones.
  - The cell state may include the gender of the present subject so that the proper pronouns can be used
  - When we see a new subject we want to forget old subject



# LSTM walk through: Second step

- Next step is to decide as to what new information we're going to store in the cell state



- In language model:

we'd want to add the gender of the new subject to the cell state, to replace the old One we are forgetting

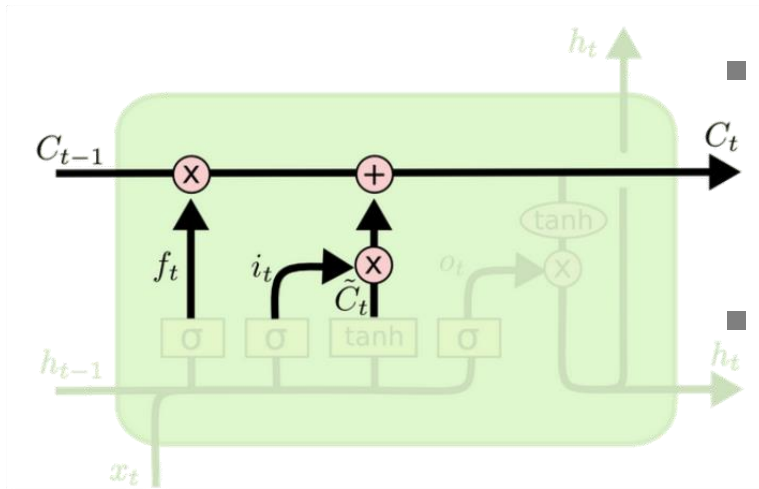
- First a sigmoid layer called Input gate layer:
  - decides which values we will update
- Next a tanh layer creates a vector of new candidate values  $\tilde{C}^{(t)}$  that could be added to the state.
- In the third step we will combine these two to create an update to the state

$$i^{(t)} = \sigma(W_i \cdot [h^{(t-1)}, x^{(t)}] + b_i)$$

$$\tilde{C}^{(t)} = \tanh(W_C \cdot [h^{(t-1)}, x^{(t)}] + b_C)$$

# LSTM walk through: Third step

- It's now time to update old cell state  $C_{t-1}$  into new cell state  $C_t$ 
  - The previous step decided what we need to do
  - We just need to do it



- We multiply the old state by  $f^{(t)}$ , forgetting the things we decided to forget earlier.

- Then we add  $i^{(t)} \cdot \tilde{C}^{(t)}$

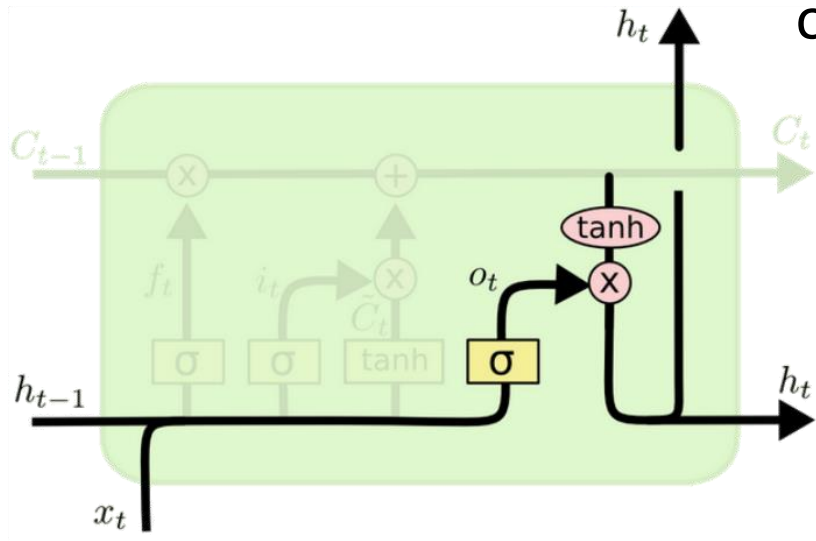
$$C^{(t)} = f^{(t)} \cdot C^{(t-1)} + i^{(t)} \cdot \tilde{C}^{(t)}$$

- In language model:

this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in previous steps

# LSTM walk through: Fourth step

- Finally we decide what we are going to output
  - This output will be a filtered version of our cell state



- First we run a sigmoid layer which decides what parts of cell state we're going to output.
- Then we put the cell state through tanh (to push values to be between -1 and 1)

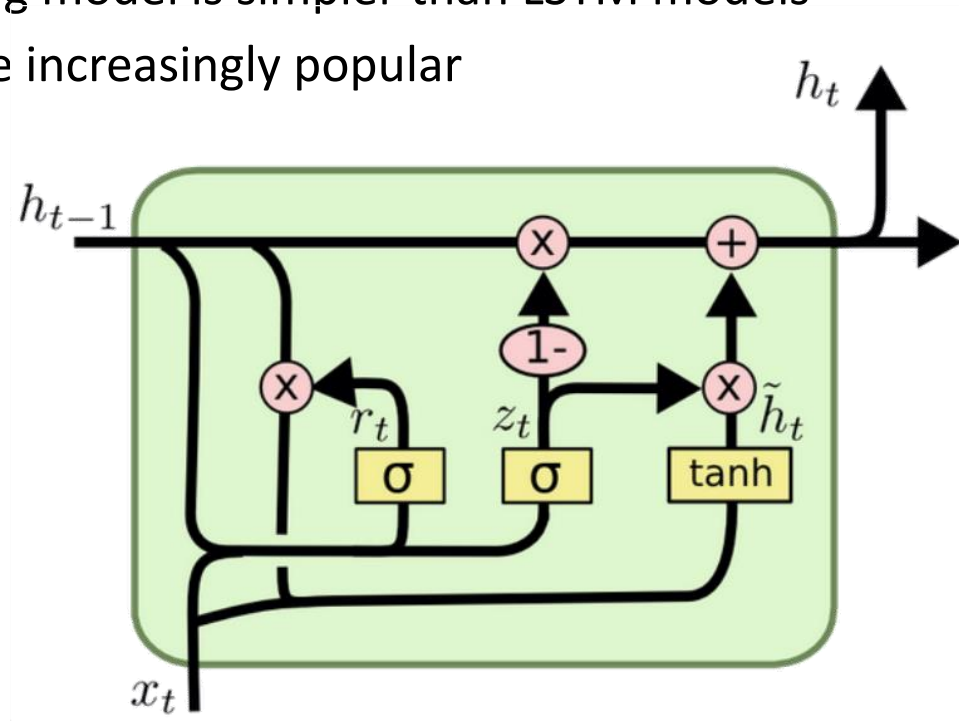
$$\begin{aligned}o^{(t)} &= \sigma(W_o \cdot [h^{(t-1)}, x^{(t)}] + b_o) \\h^{(t)} &= o_t \cdot \tanh(C^{(t)})\end{aligned}$$

- In language model:

Since it just saw a subject it might want to output information relevant to a verb, in case that is what is coming next, e.g., it might output whether the subject is singular or plural so that we know what form a verb should be conjugated into if that's what follows next.

# Gated Recurrent Unit (GRU)

- A dramatic variant of LSTM
  - It combines the forget and input gates into a single update gate
  - It also merges the cell state and hidden state, and makes some other changes
  - The resulting model is simpler than LSTM models
  - Has become increasingly popular



# Accumulating Information over Longer Duration

- Leaky Units Allow the network to accumulate information
  - Such as evidence for a particular feature or category
  - Over a long duration
- However, once the information has been used, it might be useful to forget the old state
  - E.g., if a sequence is made of sub-sequences and we want a leaky unit to accumulate evidence inside each sub-sequence, we need a mechanism to forget the old state by setting it to zero
- Gated/LSTM RNN:
  - Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it

# LSTM

- Contribution of LSTMs:
  - clever idea of introducing self-loops to produce paths where the gradient can flow for long durations
- A crucial addition: make weight on this self-loop conditioned on the context, rather than fixed
  - By making weight of this self-loop gated (controlled by another hidden unit), time-scale can be changed dynamically
  - Even for an LSTM with fixed parameters, time scale of integration can change based on the input sequence
    - Because time constants are output by the model itself
- LSTM found extremely successful in:
  - Unconstrained handwriting recognition, Speech recognition
  - Handwriting generation, Machine Translation
  - Image Captioning, Parsing



# Recurrent Neural Networks

- Introduction
- Unfolding a Graph
- Recurrent Neural Networks
- Long Term Memory
  - Leaky Units
  - Long-Short Term Memory
- **Working with Texts in Keras**

# About Text

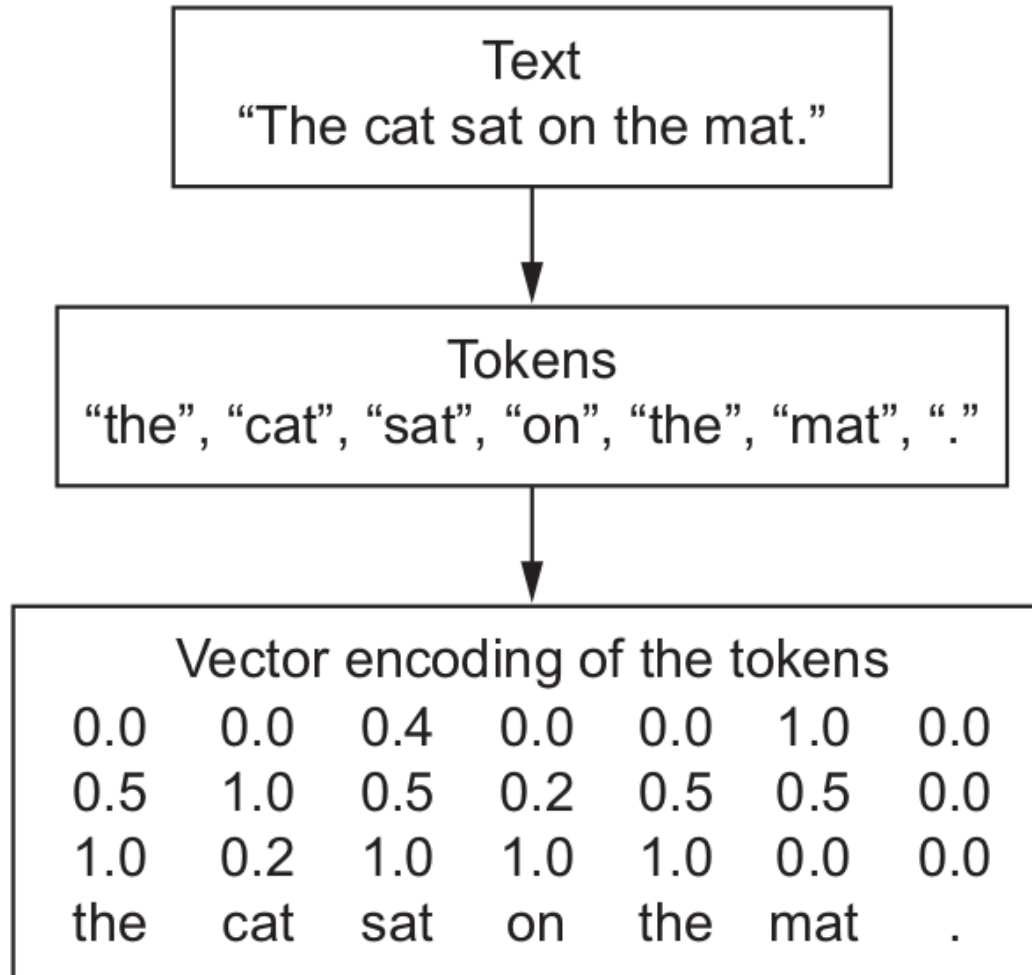
- Text is one of the most widespread forms of sequence data.
- Can be interpreted as sequence of characters or a sequence of words; interpretation as words more common.
- The deep-learning sequence-processing models can use text to produce a basic form of natural-language understanding
  - Document classification
  - Sentiment analysis
  - Author identification
  - question-answering ( QA ) (in a constrained context).



# Text as Input

- We cannot feed text directly into networks
- They only “eat” numerical tensors
- We need to **vectorize** text into numeric tensors
  - Segment text into words, and transform each word into a vector.
  - Segment text into characters, and transform each character into a vector.
  - Extract n-grams of words or characters, and transform each n-gram into a vector.
    - N-grams are overlapping groups of multiple consecutive words or characters.

# Example



# What are N-grams

- Word n-grams are groups of N (or fewer) consecutive words that you can extract from a sentence.
- Example: “The cat sat on the mat.”
- Set of 2-grams:
  - {"The", "The cat", "cat", "cat sat", "sat", "sat on", "on", "on the", "the", "the mat", "mat"}
- Set of 3-grams:
  - {"The", "The cat", "cat", "cat sat", "The cat sat", "sat", "sat on", "on", "cat sat on", "on the", "the", "sat on the", "the mat", "mat", "on the mat"}

# One-hot Encoding of Words and Characters

- One-hot encoding is the most common, most basic way to turn a token into a vector.
  - Associate a unique integer index with every word (or token)
  - Turn the integer index  $i$  into a binary vector of size  $N$  (the size of the vocabulary); the vector is all zeros except for the  $i$ th entry, which is 1.
  - Like we have done for the encoding of categorical data



# Using Keras for One-Hot Encoding

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# Tokenizer, only consider the 1000 most common words
tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(samples)

# Turn strings into list of integers
sequences = tokenizer.texts_to_sequences(samples)

# Or directly get the one hot encoding
one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

# Retrieve the word index
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

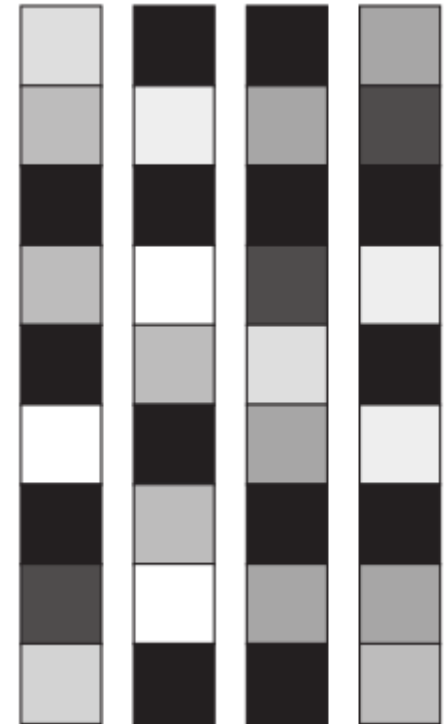
# Vectorize a Sequence

```
def vectorize_sequences(sequences, dimension=10000):  
    results = np.zeros((len(sequences), dimension))  
    for i, sequence in enumerate(sequences):  
        results[i, sequence] = 1.  
    return results
```

- You cannot feed a list of integer or one-hot encoded Matrix of different shapes into your model.
- Solution: One-hot encode your lists to turn them into vectors of 0s and 1s.
  - This would mean, for instance, turning the sequence [3, 5] into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s.
  - Then you could use as the first layer in your network a Dense layer, capable of handling floating-point vector data.

# Word Embeddings

- Vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (same dimensionality as the number of words in the vocabulary)
- Word embeddings are low-dimensional floating-point vectors (that is, dense vectors, as opposed to sparse vectors)



# How to obtain Embeddings?

- Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction).
  - In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network.
- Load into your model word embeddings that were precomputed using a different machine-learning task than the one you're trying to solve. These are called pretrained word embeddings.



# Embeddings in Keras

```
from keras.layers import Embedding  
embedding_layer = Embedding(1000, 64)
```

- The Embedding layer takes at least two arguments: the number of possible tokens (here, 1,000: 1 + maximum word index)
- The dimensionality of the embeddings (here, 64).
- The Embedding layer takes as input a 2D tensor of integers, of shape (samples, sequence\_length)
  - Sequences that are shorter must be padded with zeros, sequences that are longer must be truncated.
- Returns a 3D floating-point tensor of shape (samples, sequence\_length, embedding\_dimensionality)

# RNNs in Keras

- The simple RNN takes inputs of the shape (batch\_size, timesteps, input\_features)
- Can be run in two different modes:
  - It can return either the full sequences of successive outputs for each timestep:  
(a 3D tensor of shape (batch\_size, timesteps, output\_features) )
  - Only the last output for each input sequence  
(a 2D tensor of shape (batch\_size, output\_features) ).
- These two modes are controlled by the return\_sequences constructor argument.
- **Available RNNs: SimpleRNN, GRU, LSTM, ...**

# RNNs in Keras

```
model.add(Embedding(10000, 32))  
model.add(SimpleRNN(32))  
model.summary()
```

Layer (type)	Output Shape	Param #
embedding_22 (Embedding)	(None, None, 32)	320000
simplernn_10 (SimpleRNN)	(None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

- Creates a simple RNN which only produces one output

# RNNs in Keras

```
model.add(Embedding(10000, 32))  
model.add(SimpleRNN(32, return_sequences=True))  
model.summary()
```

Layer (type)	Output Shape	Param #
embedding_22 (Embedding)	(None, None, 32)	320000
simplernn_10 (SimpleRNN)	(None, None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

- Creates a simple RNN which produces an output at every position

# Stack RNNs

```
model = Sequential()
model.add(Embedding(10000, 32))

# All RNNs need to return a value every timestep
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))

# The last layer of RNN can only return one value
model.add(SimpleRNN(32))
```