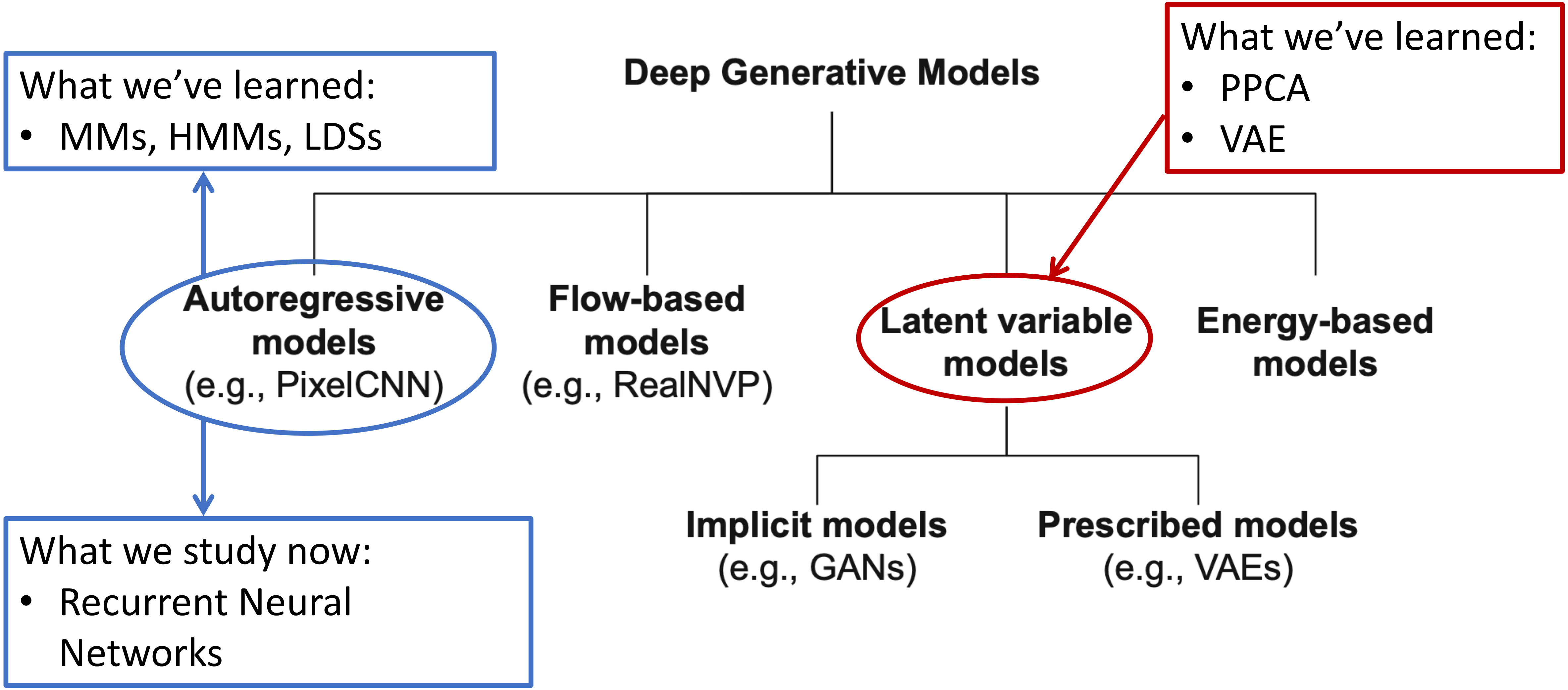
Deep Generative Models: Recurrent Neural Networks

Fall Semester 2024

René Vidal
Director of the Center for Innovation in Data Engineering and Science (IDEAS),
Rachleff University Professor, University of Pennsylvania
Amazon Scholar & Chief Scientist at NORCE

Penn
UNIVERSITY of PENNSYLVANIA

Taxonomy of Generative Models



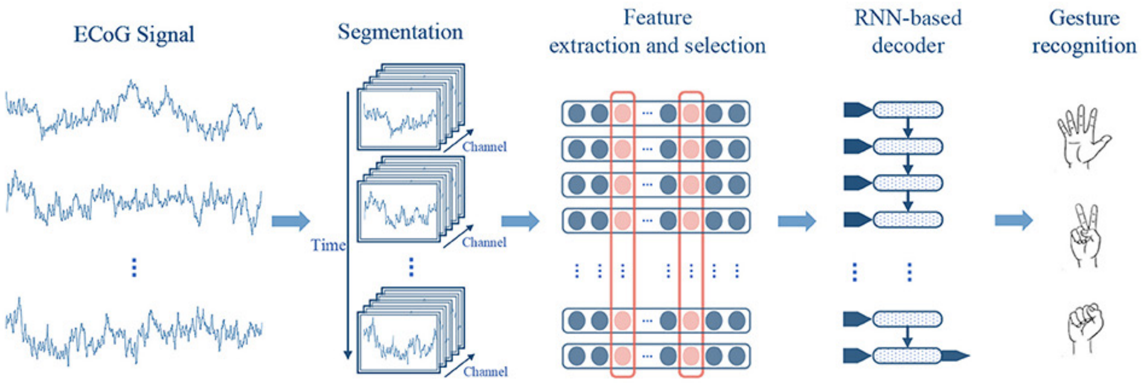
Autoregressive Models

- Many kinds of models
  - Markov Chains
  - Hidden Markov Models
  - Markov Random Fields
  - Linear Dynamical Systems
  - **Recurrent Neural Networks**
  - Transformers

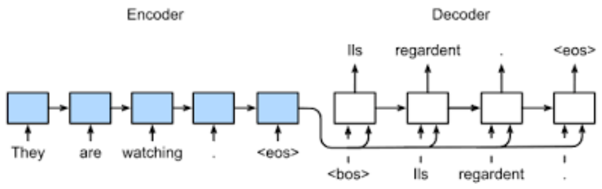
- This lecture: we focus on **Recurrent Neural Networks**
  - Vanilla RNNs
  - Basic applications for Language Modeling
  - Training and Issues with RNNs
  - LSTMs and GRUs

Applications of RNNs

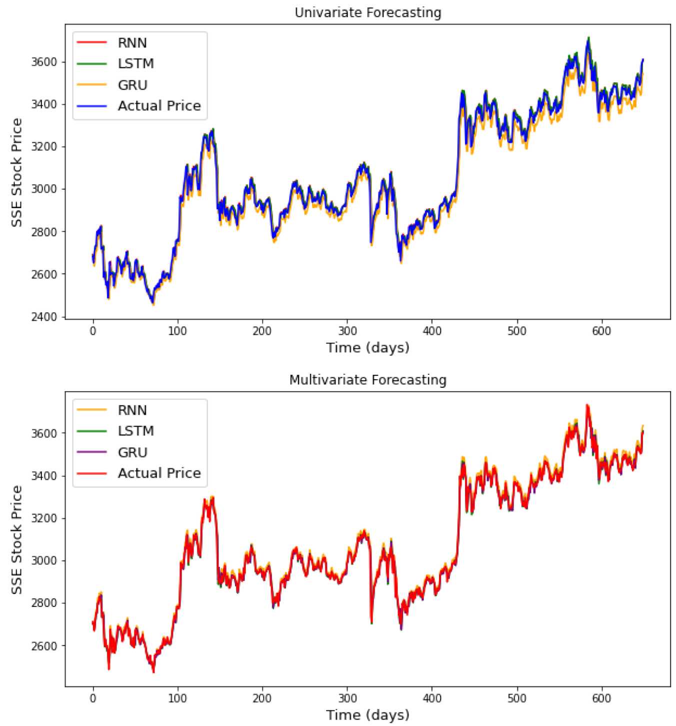
- NLP: Machine Translation, Text Classification, POS Tagging
- Healthcare: Gesture Forecasting, EGG
- Computer Vision: Self-driving, Image/Texture Classification
- Finance: Stock Price Forecasting
- Many, many more



Healthcare: Gesture Forecasting



NLP: Machine Translation

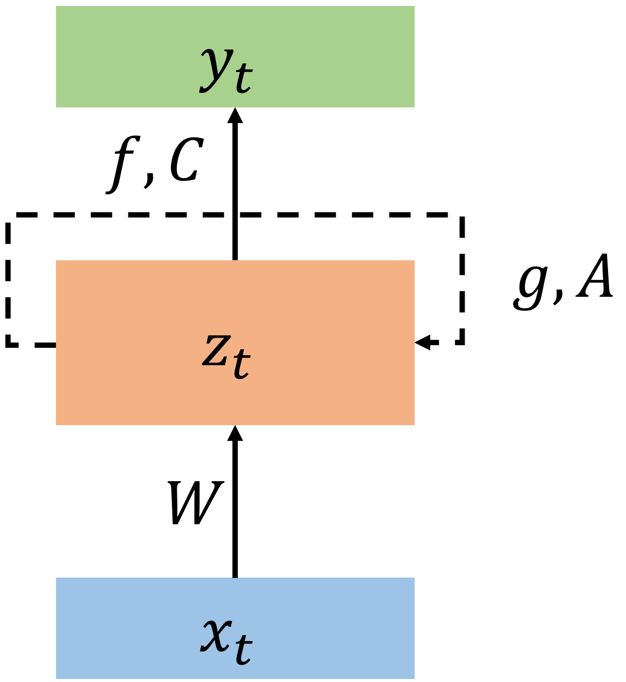


Finance: Stock Forecasting

Recurrent Neural Network (RNNs)

- Recurrent Neural Networks is a neural network architecture for sequential data
  - Often seen as a generalization of a feed-forward neural network (MLP)
- Let's denote
  - $x_0, \dots, x_T \in \mathbb{R}^D$ as the inputs
  - $y_0, \dots, y_T \in \mathbb{R}^m$ as the outputs
  - $z_0, \dots, z_T \in \mathbb{R}^d$ as the hidden states
- RNN can be described by

$$z_{t+1} = g(Az_t + Wx_t) + w_t$$
$$y_t = f(Cz_t) + v_t$$



- Where
  - $A \in \mathbb{R}^{d \times d}, W \in \mathbb{R}^{d \times D}, C \in \mathbb{R}^{m \times d}$ are weight matrices
  - f and g are nonlinear functions (e.g. f can be a softmax function for soft classification)
  - No noise w_t , v_t when RNN used for prediction instead of generation.

RNNs vs LDSs

- Linear Dynamic Systems

$$z_t = Az_{t-1} + Bx_t + w_t, \qquad w_t \sim N(0, Q)$$
$$y_t = Cz_t + Dx_t + v_t, \qquad v_t \sim N(0, R)$$

- Everything is linear
- Can be deterministic or stochastic
- Distributions of z_t and y_t has closed-form due the Gaussian assumption
- Exact inference via **Kalman filter**
- Parameter learning via **EM algorithm**

- Recurrent Neural Networks

$$z_{t+1} = g(Az_t + Wx_t) + w_t, \quad w_t \sim N(0, Q)$$
$$y_t = f(Cz_t) + v_t, \qquad v_t \sim N(0, R)$$

- Has nonlinearity from f and g
- Can be deterministic or stochastic
- Distributions of z_t and y_t does not necessarily admit a closed form
- Approximate inference via extended Kalman filter, **particle filter,** etc.
- Parameter learning via **Backpropagation Through Time**

Extended Kalman Filters for RNNs

$$z_{t+1} = g(Az_t) + w_t$$
$$y_t = f(Cz_t) + v_t$$

- Let us consider an RNN with no inputs and with noise added to the state and output.
- Can we use EM and the Kalman filter for learning and inference with RNNs?

$$p(z_{t+1} \mid z_t) = \mathcal{N}(g(Az_t), Q)$$
$$p(y_t \mid z_t) = \mathcal{N}(f(Cz_t), R)$$

- On the one hand, we can write a probabilistic model with Gaussian conditionals
- On the other hand, even if z_0 is Gaussian, $z_1 = g(Az_0) + w_t$ may not!
  - **Reason**: a linear transformation of a Gaussian is Gaussian, but the non-linearity breaks that.
- Why is this a problem?
  - A Gaussian is uniquely determined by its mean and covariance (μ, Σ)
  - The Kalman filter tracks the evolution of the mean and covariance of $z_{t+1} \mid y_{0:t}$. If this is not Gaussian, then we cannot track that anymore.

$$K_t = \hat{\Sigma}_{t|t-1} C^\top \left(C \hat{\Sigma}_{t|t-1} C^\top + R \right)^{-1}$$
$$\hat{z}_{t+1|t} = A\hat{z}_{t|t-1} + AK_t(y_0 - C\hat{z}_{t|t-1})$$
$$\hat{\Sigma}_{t+1|t} = A\left(\hat{\Sigma}_{t|t-1} - K_t C \hat{\Sigma}_{t|t-1}\right)A^\top + Q$$

Extended Kalman Filters for RNNs

$$z_{t+1} = g(Az_t) + w_t$$
$$y_t = f(Cz_t) + v_t$$

$$\tilde{z}_{t+1} = \tilde{A}_t \tilde{z}_t + w_t$$
$$y_t = \tilde{C}_t \tilde{z}_t + v_t$$

$$\tilde{A}_t = \nabla_z g(A\hat{z}_{t|t})A^\top$$
$$\tilde{C}_t = \nabla_z f(C\hat{z}_{t|t})C^\top$$

- How do we apply the Kalman filter to RNNs?
  - We linearize f and g around current estimate of mean and covariance using first-order Taylor expansion and then we can run a Kalman filtering step using the Jacobian of f and g.

- Prediction

$$\hat{z}_{t+1|t} = A\hat{z}_{t|t}$$
$$\hat{\Sigma}_{t+1|t} = A\hat{\Sigma}_{t|t}A^\top + Q$$

$$\hat{z}_{t+1|t} = g(A\hat{z}_{t|t})$$
$$\hat{\Sigma}_{t+1|t} = \tilde{A}_t\hat{\Sigma}_{t|t}\tilde{A}_t^\top + Q$$

Update

$$K_t = \hat{\Sigma}_{t|t-1}C^\top\left(C\hat{\Sigma}_{t|t-1}C^\top + R\right)^{-1}$$
$$\hat{z}_{t|t} = \hat{z}_{t|t-1} + K_t(y_0 - C\hat{z}_{t|t-1})$$
$$\hat{\Sigma}_{t|t} = \hat{\Sigma}_{t|t-1} - K_tC\hat{\Sigma}_{t|t-1}$$

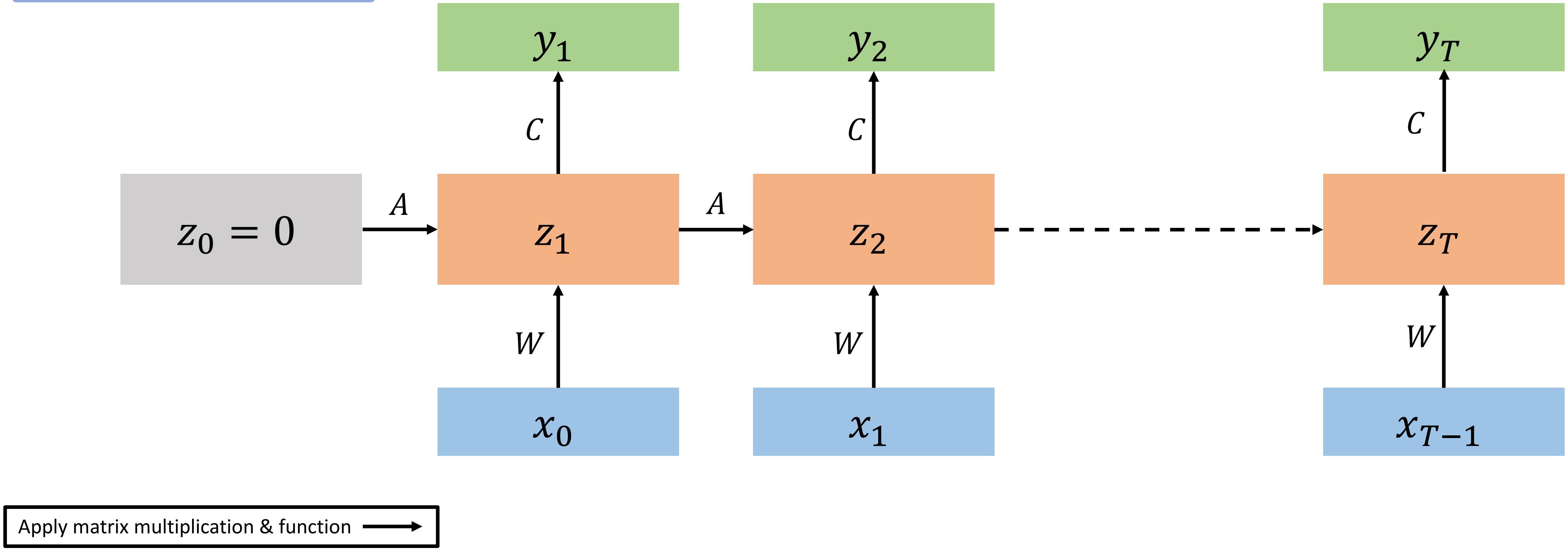
$$K_t = \hat{\Sigma}_{t|t-1}\tilde{C}_t^\top\left(\tilde{C}_t\hat{\Sigma}_{t|t-1}\tilde{C}_t^\top + R\right)^{-1}$$
$$\hat{z}_{t|t} = \hat{z}_{t|t-1} + K_t(y_0 - f(C\hat{z}_{t|t-1}))$$
$$\hat{\Sigma}_{t|t} = \hat{\Sigma}_{t|t-1} - K_t\tilde{C}_t\hat{\Sigma}_{t|t-1}$$

- Therefore, we don't have any optimality guarantees.

Unrolling and Parameter Tying

- Rather than treating it as a neural network with recurrent inputs and outputs, one can *unroll* the network such that it becomes one feed-forward pass
- Here A, C, W are the same matrices for all timestep, known as **Parameter Tying**

$$z_{t+1} = g(Az_t + Wx_t)$$
$$y_t = f(Cz_t)$$



Backpropagation Through Time (BTT)

- The unrolled graph is a well-formed (DAG) computation graph, so we can run backpropagation
- Parameters are tied across time, derivatives are aggregated across all time steps
- This is known as **Backpropagation Through Time**

- Question: Why do we want to tie the parameters?
  - Reduce the number of parameters to be learned
  - Deal with arbitrarily long sequences

- What if we always have short sequences?
  - We may untie the parameters, but then we would simply have a Feedforward Neural Network instead

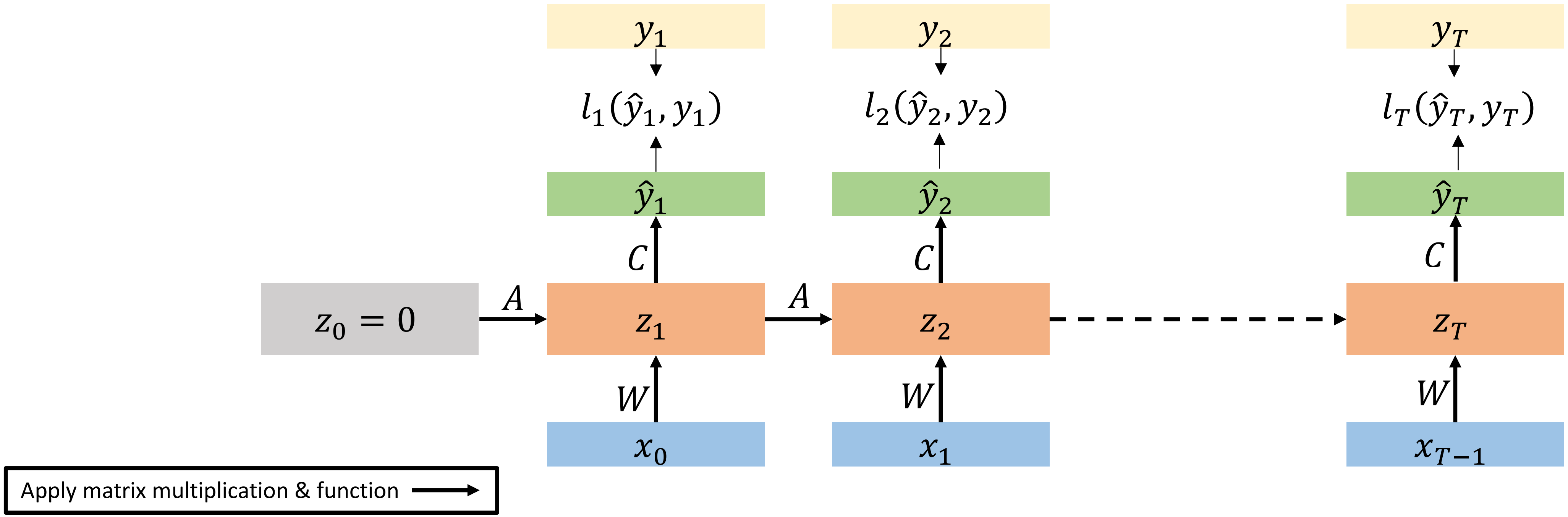
Backpropagation in Time

$z_{t+1} = g(Az_t + Wx_t)$
$y_t = f(Cz_t)$

- For a given sample $(\boldsymbol{x}, \boldsymbol{y})$, with $\boldsymbol{x} = \{x_t\}_{t=1}^{T}$ and $\boldsymbol{y} = \{y_t\}_{t=1}^{T}$,
- For prediction at each time step $\hat{y}_t$, we can compute the loss $l_t(\hat{y}_t, y_t)$ for each timestep and sum over all timesteps

$$L(\hat{y}, y) = \sum_{t=1}^{T} l_t\,(\hat{y}_t, y_t)$$

- For single prediction, we can compute loss at the final step $L(\hat{y}, y_T)$



Application of RNNs: Next Word Prediction

- Let's consider applying RNN for language modeling task, When given some preceding context, we want the language model to predict the next word:

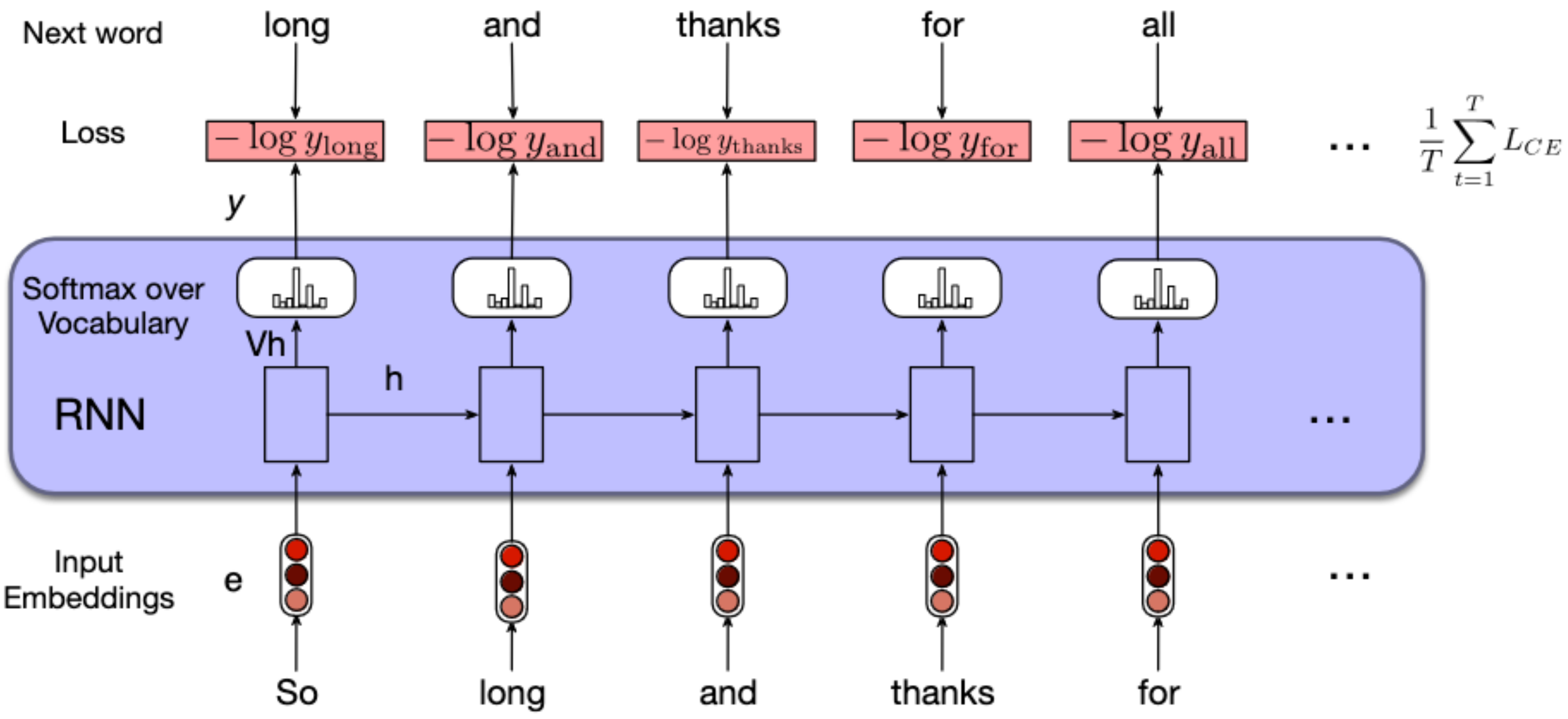
$$P(\underbrace{y_t = \text{week}}_{\text{Next word}} | \underbrace{y_{1:t-1} = \text{Homework 2 is due next}}_{\text{Context}})$$

- Suppose we have a set of N sentences $\{x^{(i)}\}_{i=1}^{N}$, where $x^{(i)} = [x_1, \dots, x_{T_i}]$ is a sentence of length T_i
- If V is the set of all possible words, then we can represent each word using a one-hot vector with size $|V| \times 1$
- Then using a word embedding matrix E, we can retrieve the word embedding associated to the current word
- This provides a way for us to go from a word to its mathematical representation

Application of RNNs: Next Word Prediction

- We want each time step of the RNN to select the next word y_{t+1} from our vocabulary, which is a discrete choice. In this case, we can use the softmax function for modeling the distribution $P(y_t|z_t)$
- Using BTT, we apply cross-entropy loss on the prediction of each timestep

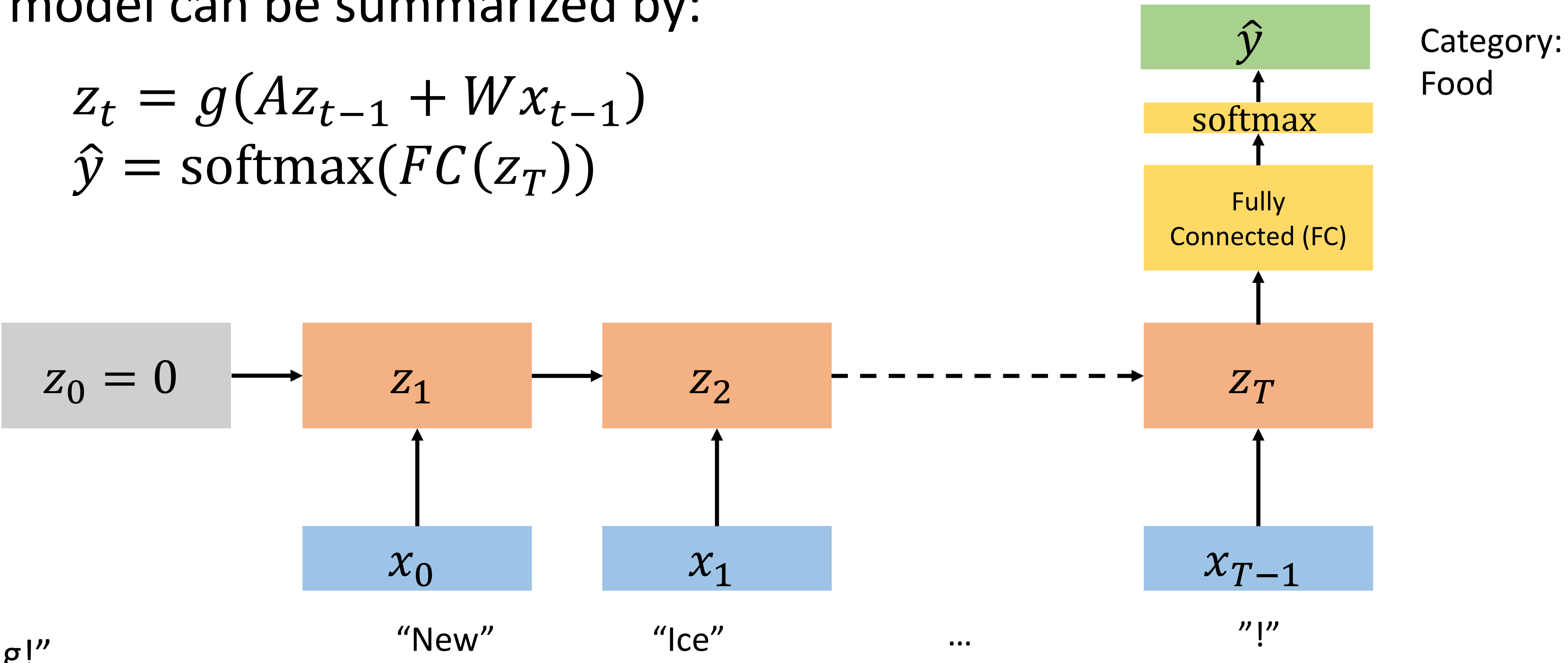
$$e_t = Ex_t$$
$$z_t = g(Az_{t-1} + We_t)$$
$$y_t = \text{softmax}(Cz_t)$$



Application of RNNs: Text Summarization

- Another application of RNNs is to summarize the whole sequence into a single category.
- For example, given the title of a news article, predict the news category
- The entire model can be summarized by:

$$z_t = g(Az_{t-1} + Wx_{t-1})$$
$$\hat{y} = \text{softmax}(FC(z_T))$$



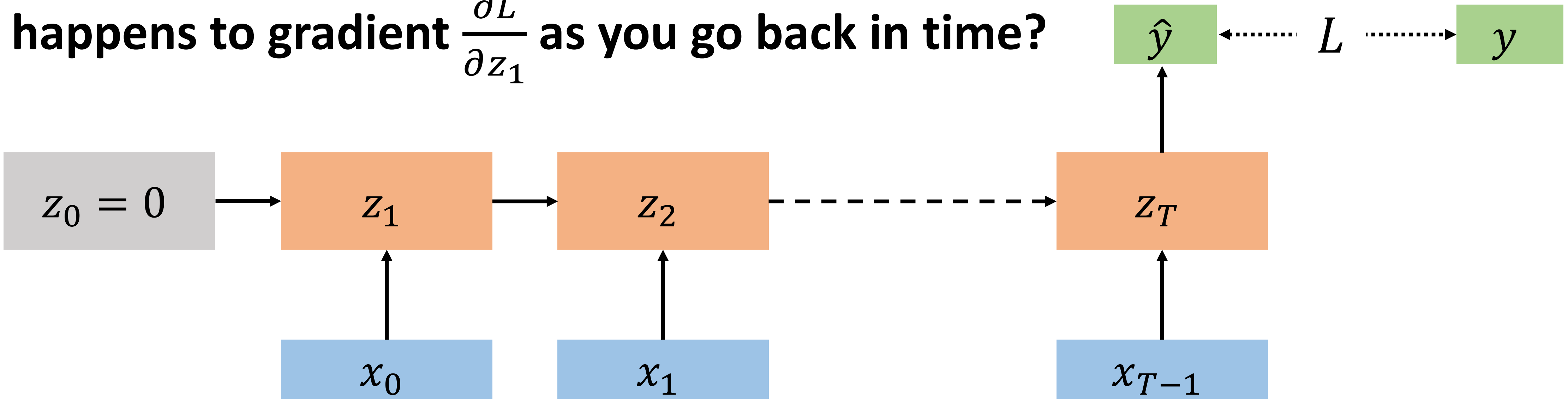
Title: "New Ice Cream truck showed up in Walnut Building!"

Issues with RNN: Exploding/Vanishing Gradients

- While RNNs can capture long-term dependencies, training can be challenging
- Consider a simple RNN model with output at the last iteration:

$$z_t = g(Az_{t-1} + Wx_{t-1})$$
$$y = Cz_T$$

- **What happens to gradient $\frac{\partial L}{\partial z_1}$ as you go back in time?**



$$\frac{\partial L}{\partial z_0} = \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_3}{\partial z_2} \cdots \frac{\partial \hat{y}}{\partial z_T} \cdot \frac{\partial L}{\partial \hat{y}} = A^\top A^\top A^\top \cdots C^\top \frac{\partial L}{\partial \hat{y}} = (CA^T)^\top \frac{\partial L}{\partial \hat{y}}$$

Assuming g = identity

Apply matrix multiplication & function →

Exploding/Vanishing Gradients: LDS case

$$\frac{\partial L}{\partial z_0} = \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_3}{\partial z_2} \cdots \frac{\partial \hat{y}}{\partial z_T} \cdot \frac{\partial L}{\partial \hat{y}} = A^\top A^\top A^\top \cdots C^\top \frac{\partial L}{\partial \hat{y}} = (CA^T)^\top \frac{\partial L}{\partial \hat{y}}$$

- Let $\lambda_1(A)$ be the maximum eigenvalue of A.
- For any initial condition z_0 and a large $T \to \infty$
  - Exploding: If $|\lambda_1(A)| > 1$, A^T will grow to infinity
  - Vanishing: If $|\lambda_1(A)| < 1$, A^T will diminish to zero
- Hence, the gradient involving A^T terms will also either explode or vanish.

Issues with RNN: Vanishing Gradients

- We have to backpropagate through many gradient terms to get back to the first time step
- This means long-range dependencies are difficult to learn (although in theory they are learnable)

- Solutions:
  - Better optimizers (second order methods, approximate second order methods)
  - Normalization (at each layer to keep gradient norms stable)
  - Clever initializations such that gradients don't go to zero (e.g. start with random orthonormal matrices)
- **Alternative parameterization: LSTMs and GRUs**

Long Short Term Memory (LSTM)

- So how does LSTM work? And how does it address the issue of vanishing gradients?
- Intuition: Vanishing gradients happen because we multiply many gradients across time, we want some ways to prevent that

- Long Short Term Memory (LSTM) can be described as a sequence of **memory cells**, which we will go step by step

$$c_t = f_t \odot c_{t-1} + i_t \odot f([x_t; z_{t-1}])$$

$$z_t = o_t \odot g(c_t)$$

$$f_t = \sigma\left(f_{\text{forget}}([x_t; z_{t-1}])\right) \quad \text{“forget gate”}$$

$$i_t = \sigma(f_{\text{input}}([x_t; z_{t-1}])) \quad \text{“input gate”}$$

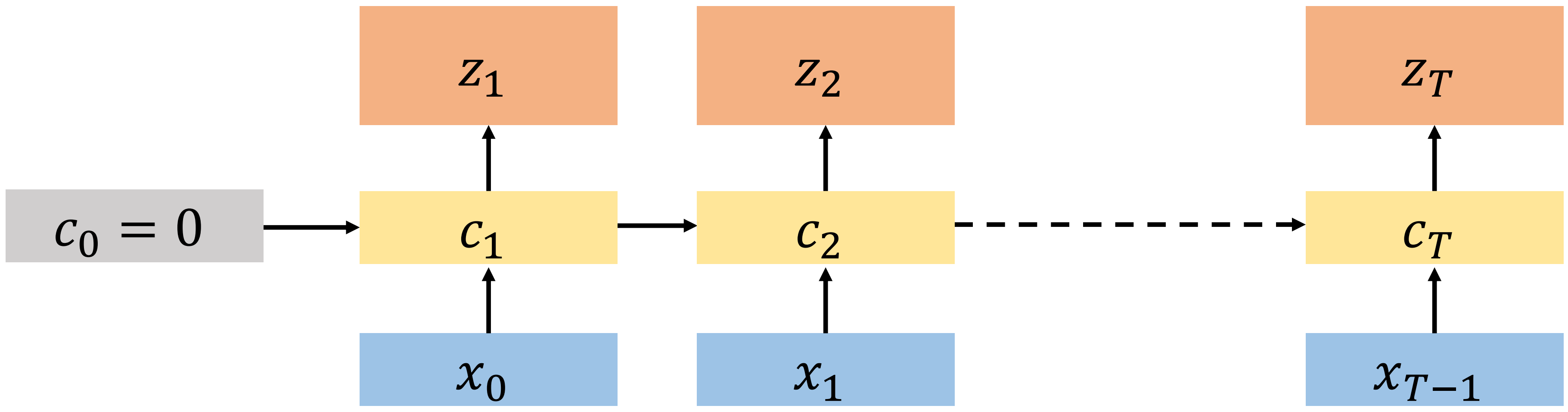
$$o_t = \sigma(f_{\text{output}}([x_t; z_{t-1}])) \quad \text{“output gate”}$$

Long Short Term Memory: Memory Cells

- Information learned by the LSTM are stored in “cells”, represented by c_t
- New information comes from the $f(x_t)$

$$c_t = \; c_{t-1} + f(x_t) \qquad \text{where } f(v) = \tanh(Wv + b)$$
$$h_t = g(c_t)$$

- Note from the formulation, $\frac{\partial c_t}{\partial c_{t-1}} = I$

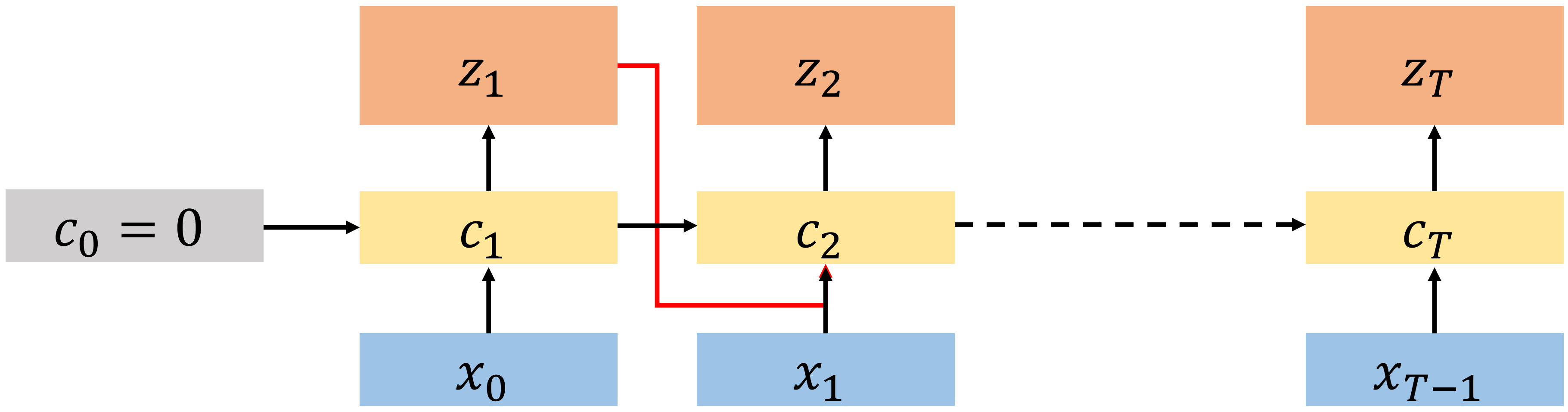


Long Short Term Memory: Memory Cells

- Now if we concatenate what's been learned in the hidden states h_t with new information from f (highlighted arrow in red)

$$c_t = c_{t-1} + f([x_t; h_{t-1}])$$
$$h_t = g(c_t)$$

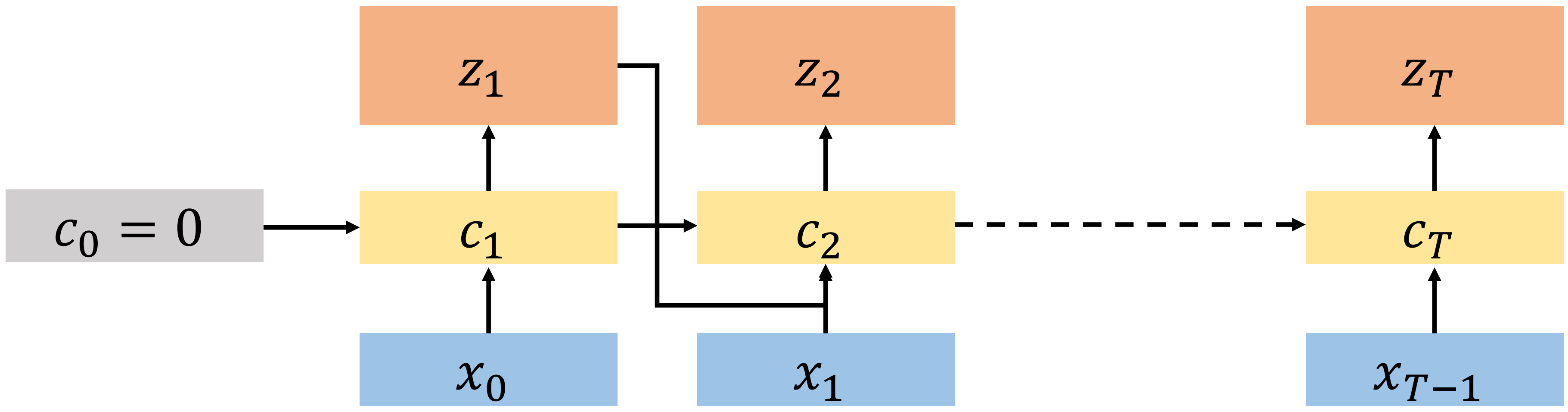
- Instead of gradient being identity, $\frac{\partial c_t}{\partial c_{t-1}} = I + \varepsilon$, with ε being small



Long Short Term Memory: Forget and Input gate

- Now we need some way to control what to input and what to forget

$$c_t = f_t \odot c_{t-1} + i_t \odot f([x_t; z_{t-1}])$$
$$z_t = g(c_t)$$
$$f_t = \sigma\left(f_{\text{forget}}([x_t; z_{t-1}])\right)$$ "forget gate"
$$i_t = \sigma(f_{\text{input}}([x_t; z_{t-1}]))$$ "input gate"



Long Short Term Memory: Output gate

- Finally, we need some way to decide what to store in our hidden state

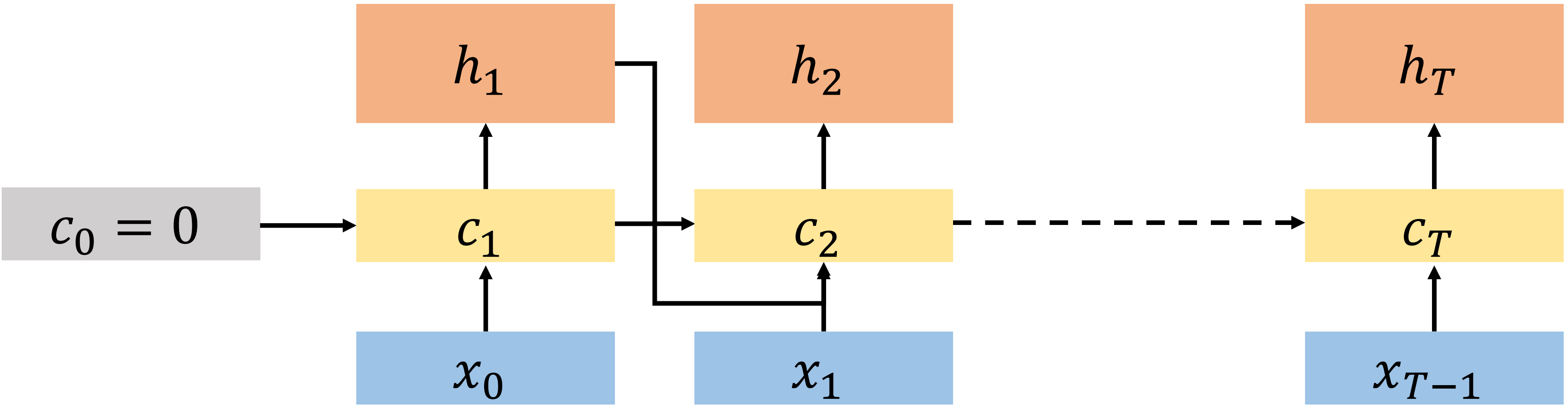
$$c_t = f_t \odot c_{t-1} + i_t \odot f([x_t; z_{t-1}])$$ "updating the cell"

$$z_t = o_t \odot g(c_t)$$

$$f_t = \sigma\left(f_{\text{forget}}([x_t; z_{t-1}])\right)$$ "forget gate"

$$i_t = \sigma(f_{\text{input}}([x_t; z_{t-1}]))$$ "input gate"

$$o_t = \sigma(f_{\text{output}}([x_t; z_{t-1}]))$$ "output gate"

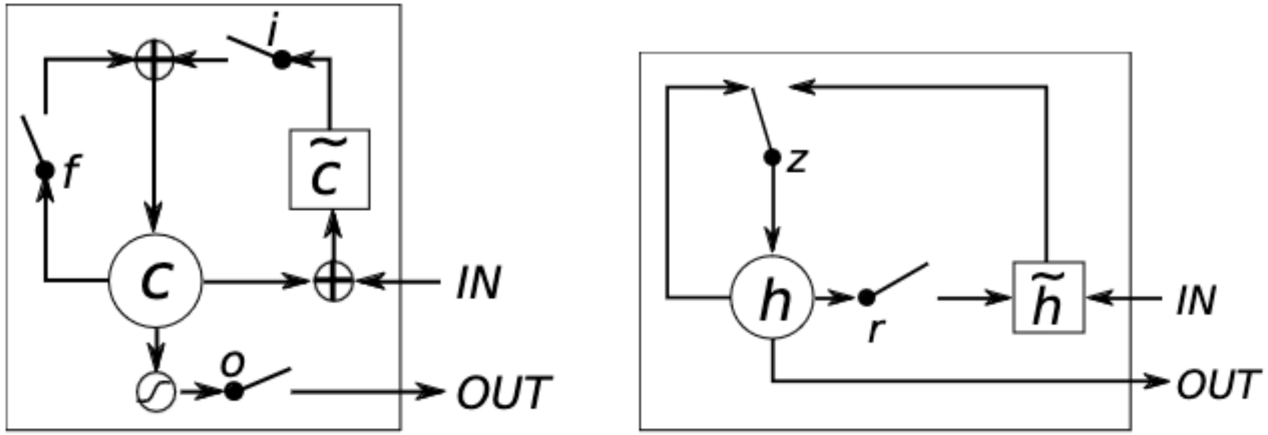


Other Variants: Gated Recurrent Neural Networks

- Another famous variant of the vanilla RNNs is Gated Recurrent Neural Network
  - Instead of a memory cell, it uses what's known as a **Gated Recurrent United (GRU)**
- On a high level, rather than using forget, input and output gates like LSTM
- GRU uses a weighted sum of two hidden states

$$z_t = (1 - s_t) \odot z_{t-1} + s_t \odot f([x_t\,; r_t \odot z_{t-1}])$$

- Empirically, GRUs perform just as well as LSTMs, but much more efficient because of it has fewer gates

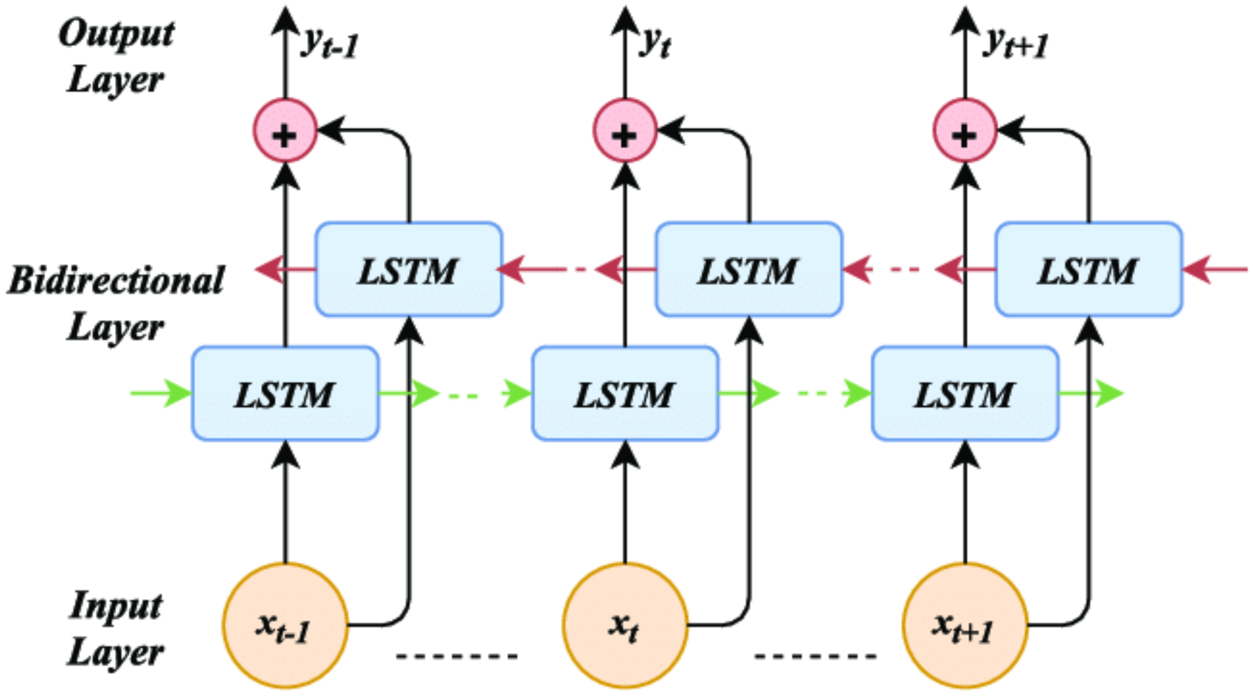


(a) Long Short-Term Memory (b) Gated Recurrent Unit

Chung et al. (2014) Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling: https://arxiv.org/pdf/1412.3555v1.pdf

Other Variants: Bidirectional-RNNs

- Vanilla RNNs/LSTMs only go forward in time $t = 1, 2, \ldots, T$
  - This makes it hard trajectories with long histories, i.e. when T is large
- Proposed Modification: To have another trajectory that goes backward in time
  - And the output $P(y_t \mid h_{\text{forward}}, h_{\text{backward}})$ depends on forward and backward hidden states
- Intuition from NLP: knowing a word means knowing what comes before and after the word
- Experiments show this reduces the vanishing gradient problem



Huang, Zhiheng, Wei Xu, and Kai Yu. "Bidirectional LSTM-CRF models for sequence tagging." (2015).

Other Variants

- Conclusion: Once you know what the building blocks are, you can create different variants that are suitable for your task

- This is also not limited to RNNs. As we will see in next lecture, for example, we can combine RNNs with VAEs for more complicated tasks

Next Lecture: Generative RNNs?

- So far we have only learned a discriminative model for RNNs
  - Simpliet RNN model

$$z_{t+1} = g(z_t, x_t)$$
$$y_t = f(z_t)$$

  - And learning using some loss function on $(x_{0:T}, y_{0:T})$ and gradient descent.
  - Is there a generative approach to RNNs?

- Learning a ”generative” RNN would allow us to:
  - Sample new trajectories
  - Explicitly model the trajectories with known distributions
  - Compute the likelihood of trajectories