Deep Generative Models: Recurrent Neural Networks

Fall Semester 2025

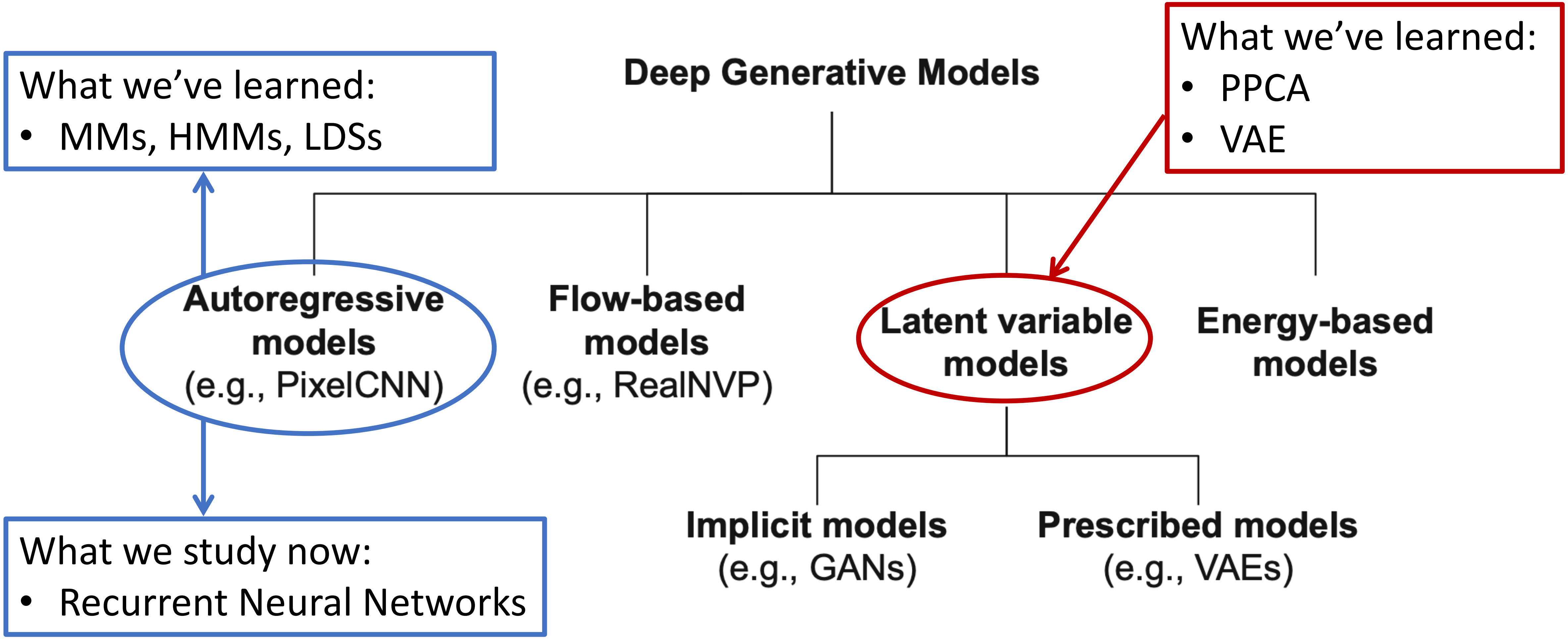
René Vidal

Director of the Center for Innovation in Data Engineering and Science (IDEAS),

Rachleff University Professor, University of Pennsylvania

Amazon Scholar & Chief Scientist at NORCE

Taxonomy of Generative Models



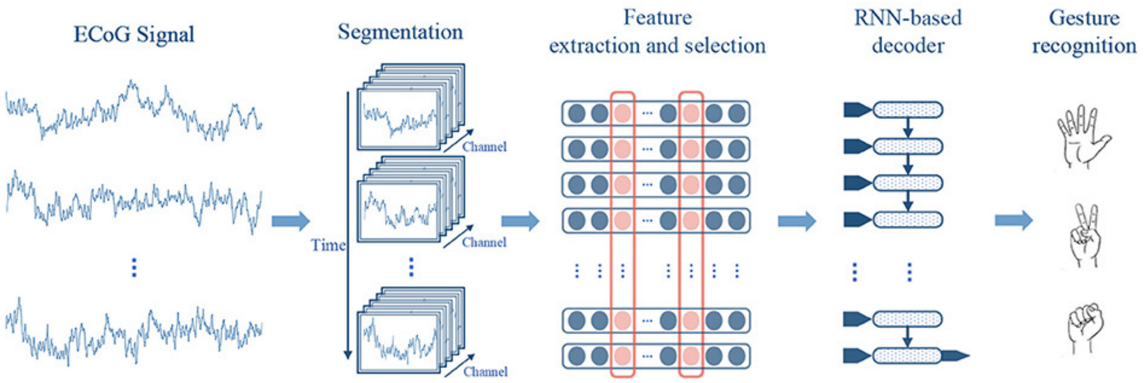
Autoregressive Models

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

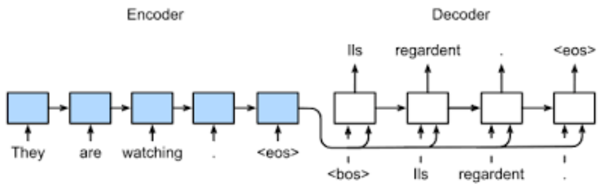
- This lecture: we focus on **Recurrent Neural Networks**
  - Vanilla RNNs
  - Basic applications for Language Modeling
  - Training and Issues with RNNs
  - LSTMs and GRUs

Applications of RNNs

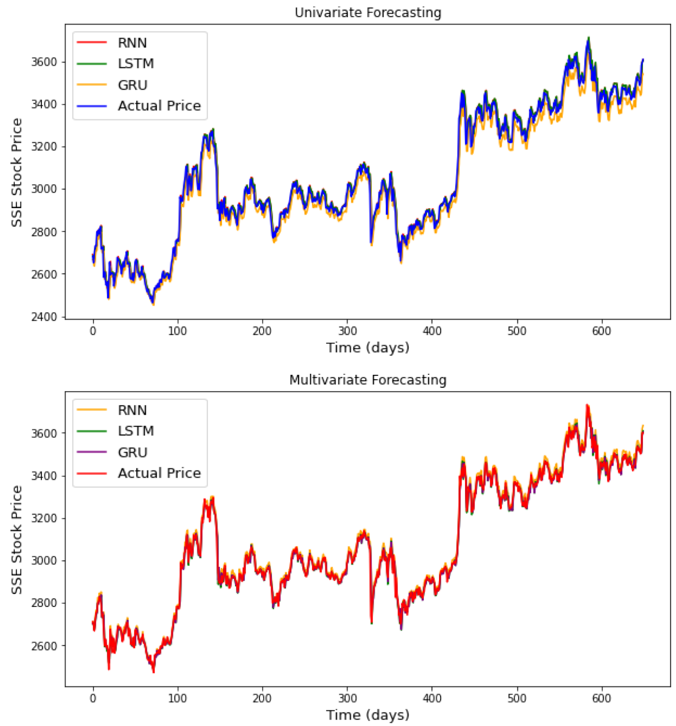
- NLP: Machine Translation, Text Classification, POS Tagging
- Healthcare: Gesture Forecasting, EGG
- Computer Vision: Self-driving, Image/Texture Classification
- Finance: Stock Price Forecasting
- Many, many more



Healthcare: Gesture Forecasting



NLP: Machine Translation

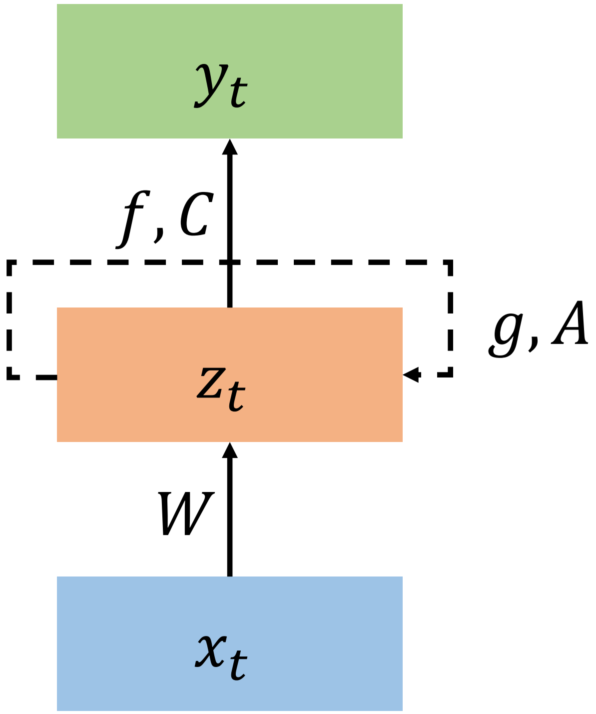


Finance: Stock Forecasting

Recurrent Neural Network (RNNs)

- Recurrent Neural Networks (RNNs) are **non-linear dynamical systems** described by

$$z_t = g(Az_{t-1} + Wx_t) + w_t$$
$$y_t = f(Cz_t) + v_t$$



- Here
  - $x_1, \ldots, x_T \in \mathbb{R}^D$ denote the **inputs**
  - $y_0, \ldots, y_T \in \mathbb{R}^m$ denote the **outputs**
  - $z_0, \ldots, z_T \in \mathbb{R}^d$ denote the **hidden states**, with z_0 the initial state
  - $A \in \mathbb{R}^{d \times d}, W \in \mathbb{R}^{d \times D}, C \in \mathbb{R}^{m \times d}$ 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

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

- 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

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

- 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

- Let us consider an RNN with no inputs and with noise added to the state and output.

$$z_t = g(Az_{t-1}) + w_t$$
$$y_t = f(Cz_t) + v_t$$

- Can we use EM and the Kalman filter for learning and inference with RNNs?
- On the one hand, we can write a probabilistic model with Gaussian conditionals

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

- 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 $z_t \mid y_{1:t-1}$. If this is not Gaussian, then we cannot track that anymore.

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

Extended Kalman Filters for RNNs

$$z_t = g(Az_{t-1}) + w_t$$
$$y_t = f(Cz_t) + v_t$$

- 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
  - We run a Kalman filtering step using the Jacobians J_f, J_g of f and g.

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

$$\tilde{A}_t := J_g\left(A\hat{z}_{t-1|t-1}\right)A$$
$$\tilde{C}_t := J_f\left(C\hat{z}_{t|t-1}\right)C$$

- 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

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

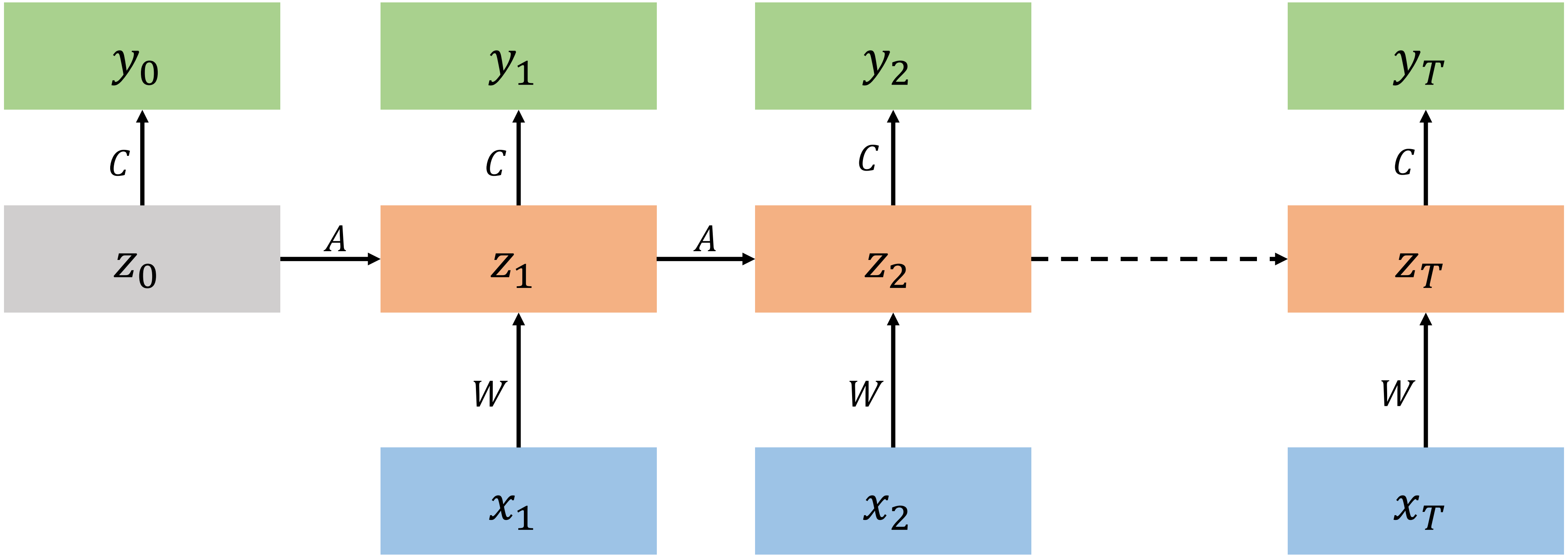
$$K_t = \hat{\Sigma}_{t|t-1}\tilde{C}_t^\top\left(\tilde{C}_t\hat{\Sigma}_{t|t-1}\tilde{C}_t^\top + R\right)^{-1}$$
$$\hat{z}_{t|t} = \hat{z}_{t|t-1} + K_t(y_t - f(C\hat{z}_{t|t-1}))$$
$$\hat{\Sigma}_{t|t} = \hat{\Sigma}_{t|t-1} - K_t\tilde{C}_t\hat{\Sigma}_{t|t-1}$$

- We do not have any optimality guarantees.

Unrolling and Parameter Tying

$$z_t = g(Az_{t-1} + Wx_t)$$
$$y_t = f(Cz_t)$$

- Rather than treating an RNN as a neural network with recurrent inputs and outputs, we can *unroll* the network such that it becomes a feed-forward network



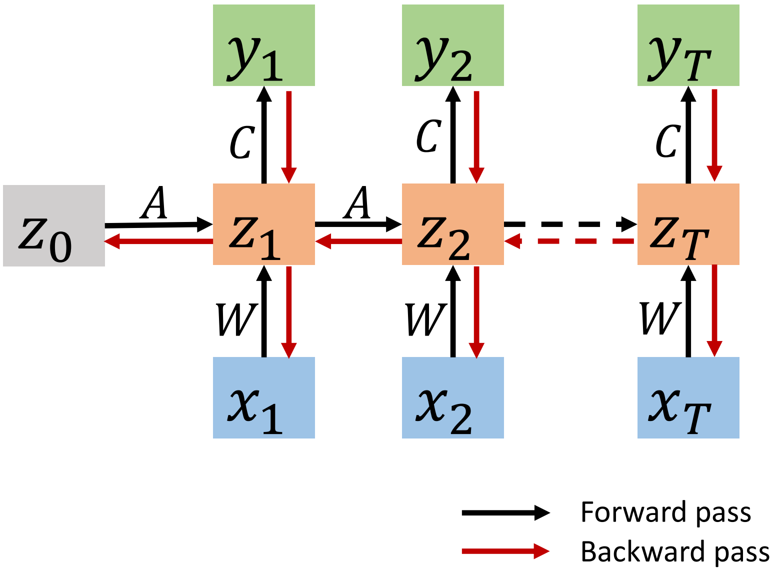
- Here A, C, W are the same matrices for all timestep, known as **Parameter Tying**

Note: Here we omit the noise terms w_t and v_t for simplicity.

→ Apply matrix multiplication & function

Backpropagation Through Time (BPTT)

- The unrolled graph is a well-formed **computation graph** (which is a directed acyclic graph), so we can run backpropagation on it
- 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, then we would simply have a **standard feedforward neural network** instead

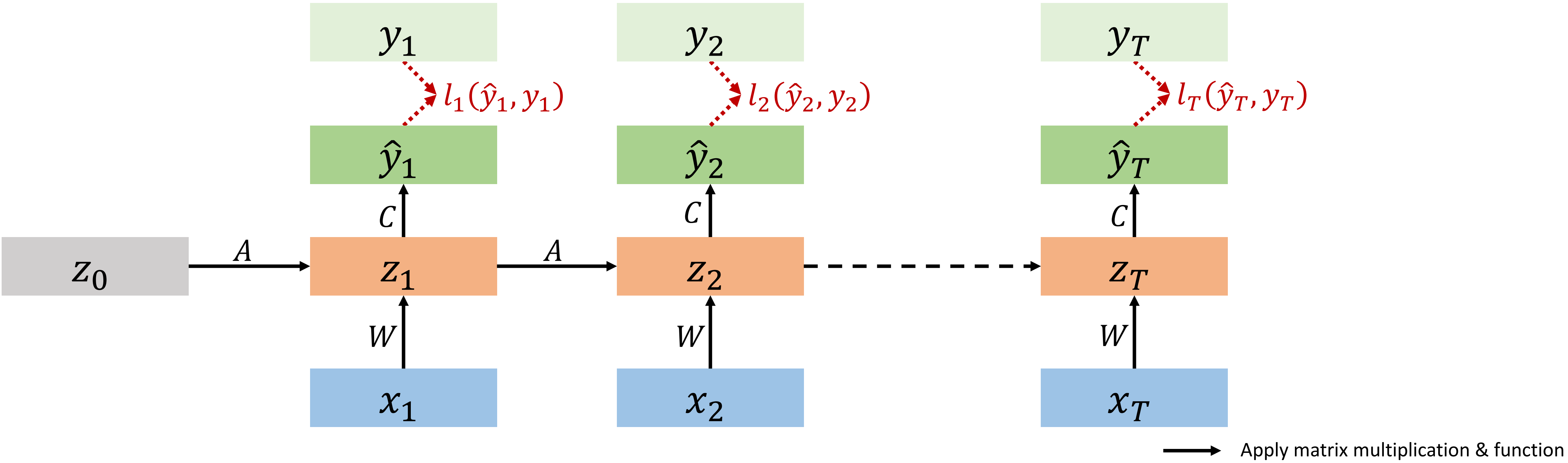
Loss Computation in Time

$$z_t = g(Az_{t-1} + Wx_t)$$
$$y_t = f(Cz_t)$$

- Given $(\boldsymbol{x}, \boldsymbol{y})$, with $\boldsymbol{x} = \{x_t\}_{t=1}^{T}$ and $\boldsymbol{y} = \{y_t\}_{t=1}^{T}$, we can define different losses
- For a task that requires 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

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

- For a task that needs a single prediction, we can compute the final loss $\mathcal{L}(\hat{y}, y_T)$



Application of RNNs: Next Word Prediction

- Let us consider using an RNN for a **language modeling task**. Given some preceding context, we want the language model to predict the next word:

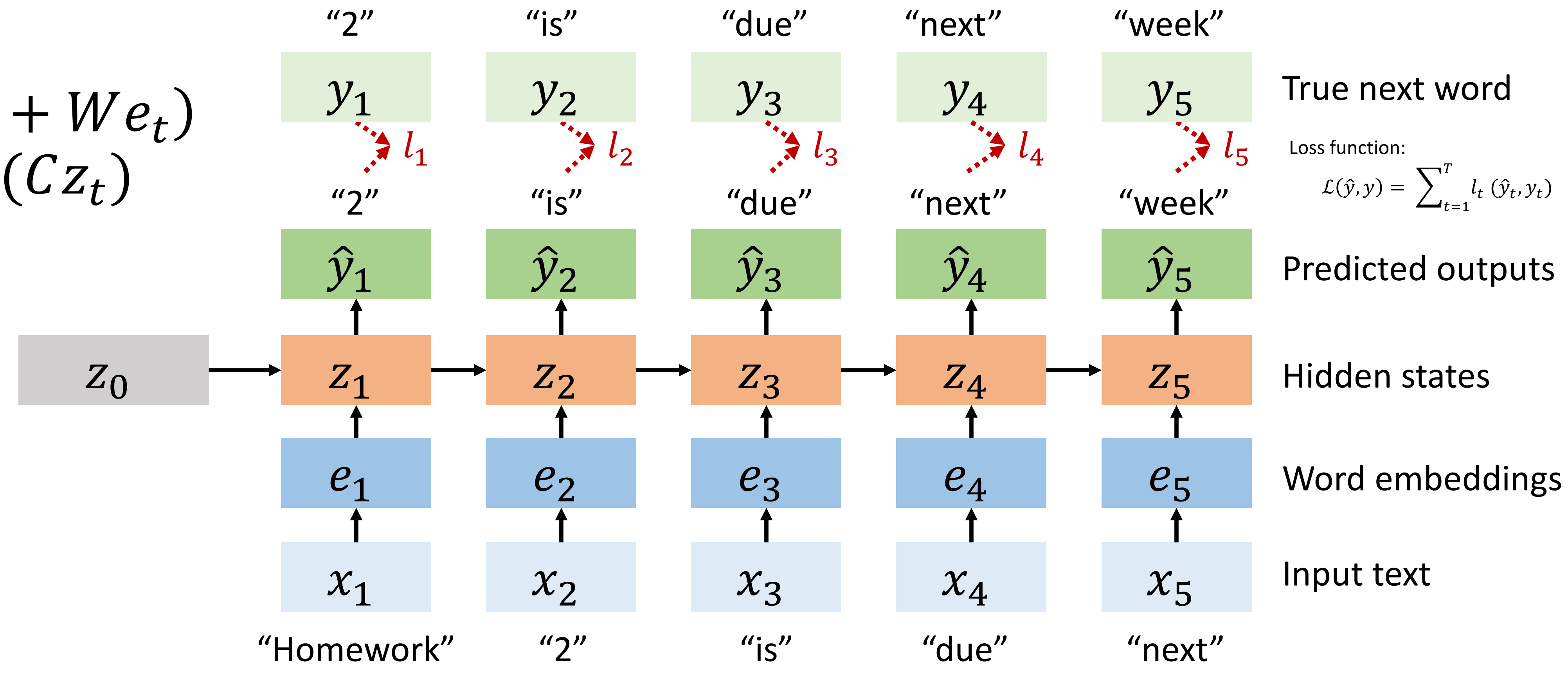
$$P(\underbrace{y_t = \text{“week”}}_{\text{Next word}} \mid y_{0:t-1} = \underbrace{\text{“Homework 2 is due next”}}_{\text{Context}})$$

- Suppose we have a set of N sentences $\{x^{(i)}\}_{i=1}^{N}$, where $x^{(i)} = [x_1, \ldots, x_{T_i}]$ 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** $E \in \mathbb{R}^{D \times |V|}$, 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 from our vocabulary, which is a discrete choice. In this case, we can use the Softmax function for modeling the distribution $P(y_t \mid z_t)$
- Using BPTT, we apply cross-entropy loss on the prediction of each timestep

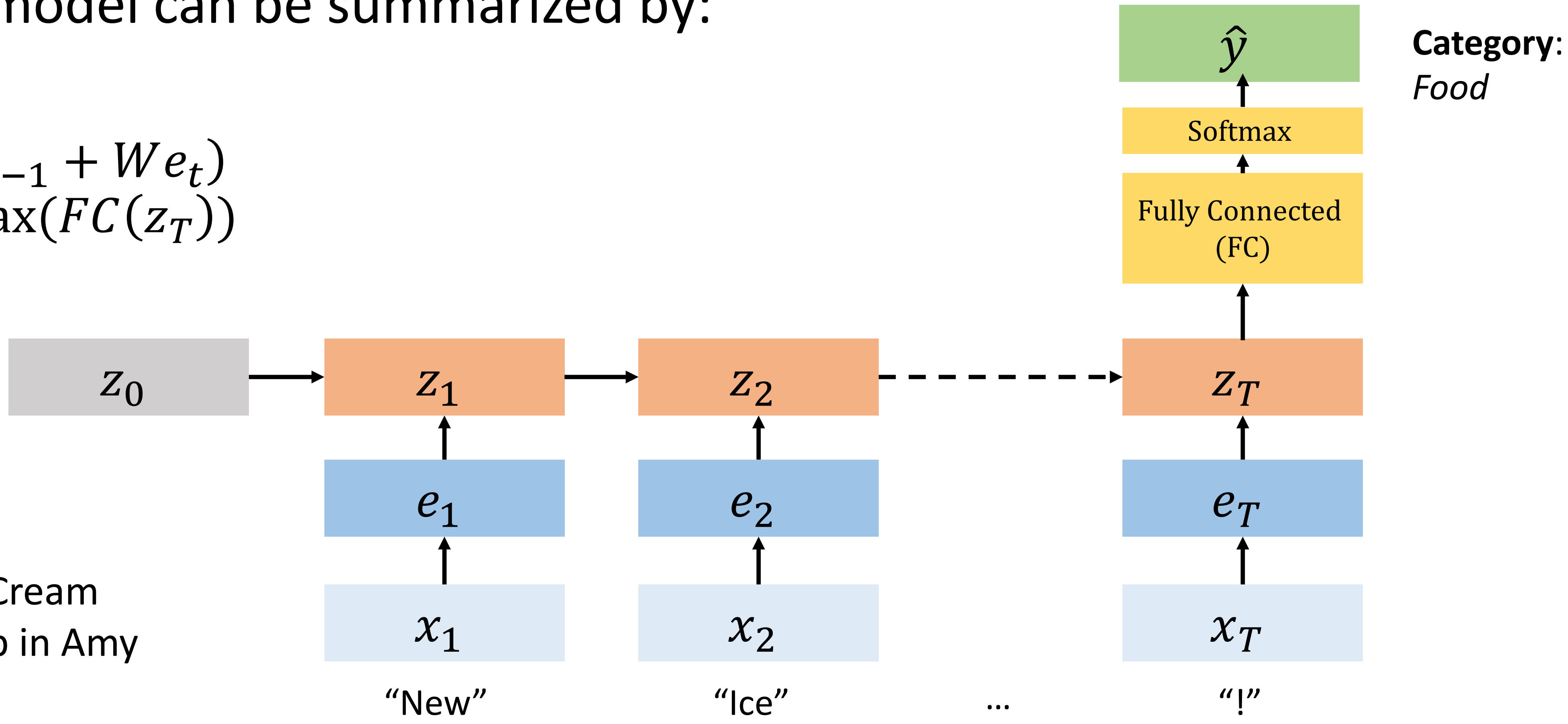
$$e_t = Ex_t$$
$$z_t = g(Az_{t-1} + We_t)$$
$$y_t = \text{Softmax}(Cz_t)$$



Application of RNNs: Text Classification

- 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:

$$
\begin{aligned}
e_t &= E x_t \\
z_t &= g(A z_{t-1} + W e_t) \\
\hat{y} &= \mathrm{Softmax}(FC(z_T))
\end{aligned}
$$



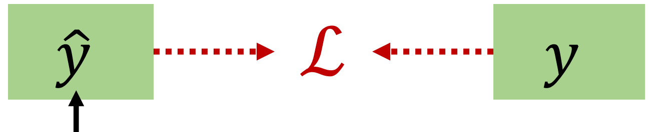
Title: "New Ice Cream truck showed up in Amy Gutmann Hall!"

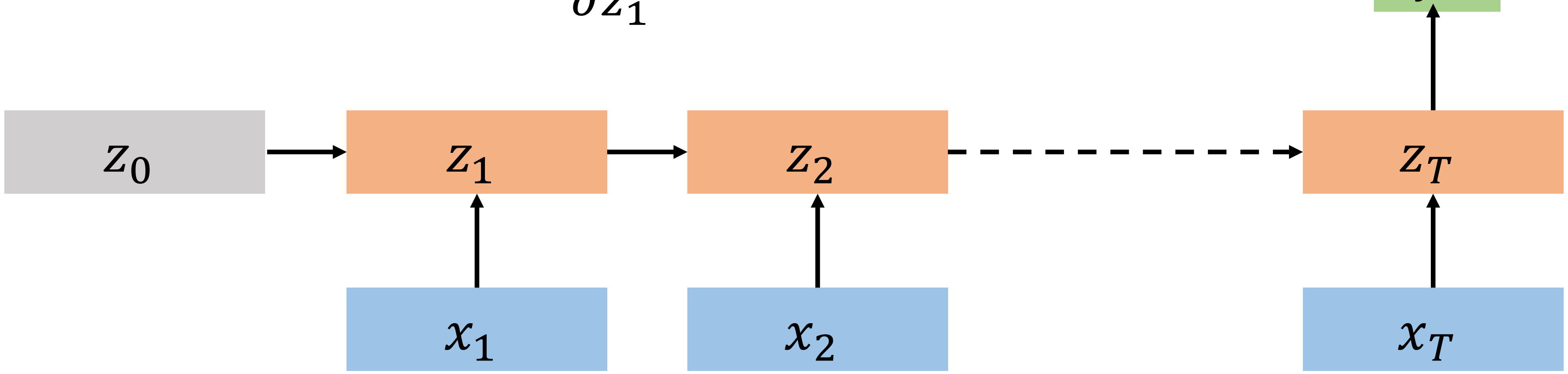
Issues with RNN: Exploding/Vanishing Gradients

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

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

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





$$\frac{\partial \mathcal{L}}{\partial z_1} = \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_3}{\partial z_2} \cdots \frac{\partial \hat{y}}{\partial z_T} \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}} = A^\top A^\top A^\top \cdots C^\top \frac{\partial \mathcal{L}}{\partial \hat{y}} = (CA^{T-1})^\top \frac{\partial \mathcal{L}}{\partial \hat{y}}$$

$\boxed{\text{Assuming } g = \text{identity}}$

If the eigenvalues of A are less than 1, **gradients vanish**;
If greater than 1, **gradients explode**.

Exploding/Vanishing Gradients: LDS case

- More generally,

$$\frac{\partial \mathcal{L}}{\partial z_t} = (CA^{T-t})^\top \frac{\partial \mathcal{L}}{\partial \hat{y}} \implies \frac{\partial \mathcal{L}}{\partial A} = \sum_t \frac{\partial z_t}{\partial A} \cdot \frac{\partial \mathcal{L}}{\partial z_t} = \sum_t \frac{\partial z_t}{\partial A} \cdot (CA^{T-t})^\top \frac{\partial \mathcal{L}}{\partial \hat{y}}$$

- 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.

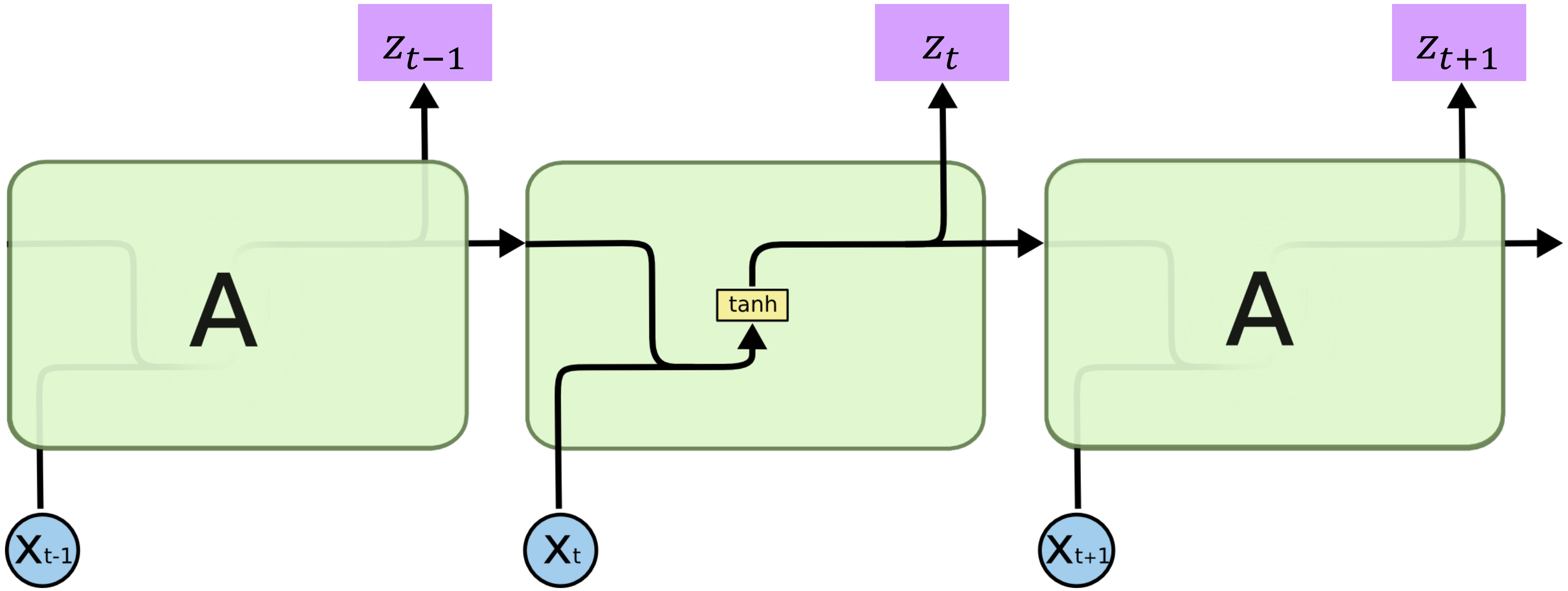
Issues with RNN: Vanishing Gradients

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

- **Solutions**:
  - **Better optimizers** (e.g., second order or approximate second order methods)
  - **Normalization** (at each layer to keep gradient norms stable)
  - **Clever initializations** (e.g., start with random orthonormal matrices to prevent gradients from vanishing)
- **Alternative parameterization**: **LSTMs** and **GRUs**

Introduction to Long Short-Term Memory (LSTM)

- Recap of RNN: chain of repeating modules of neural network
  - In standard RNN, the repeating network is just a single tanh layer

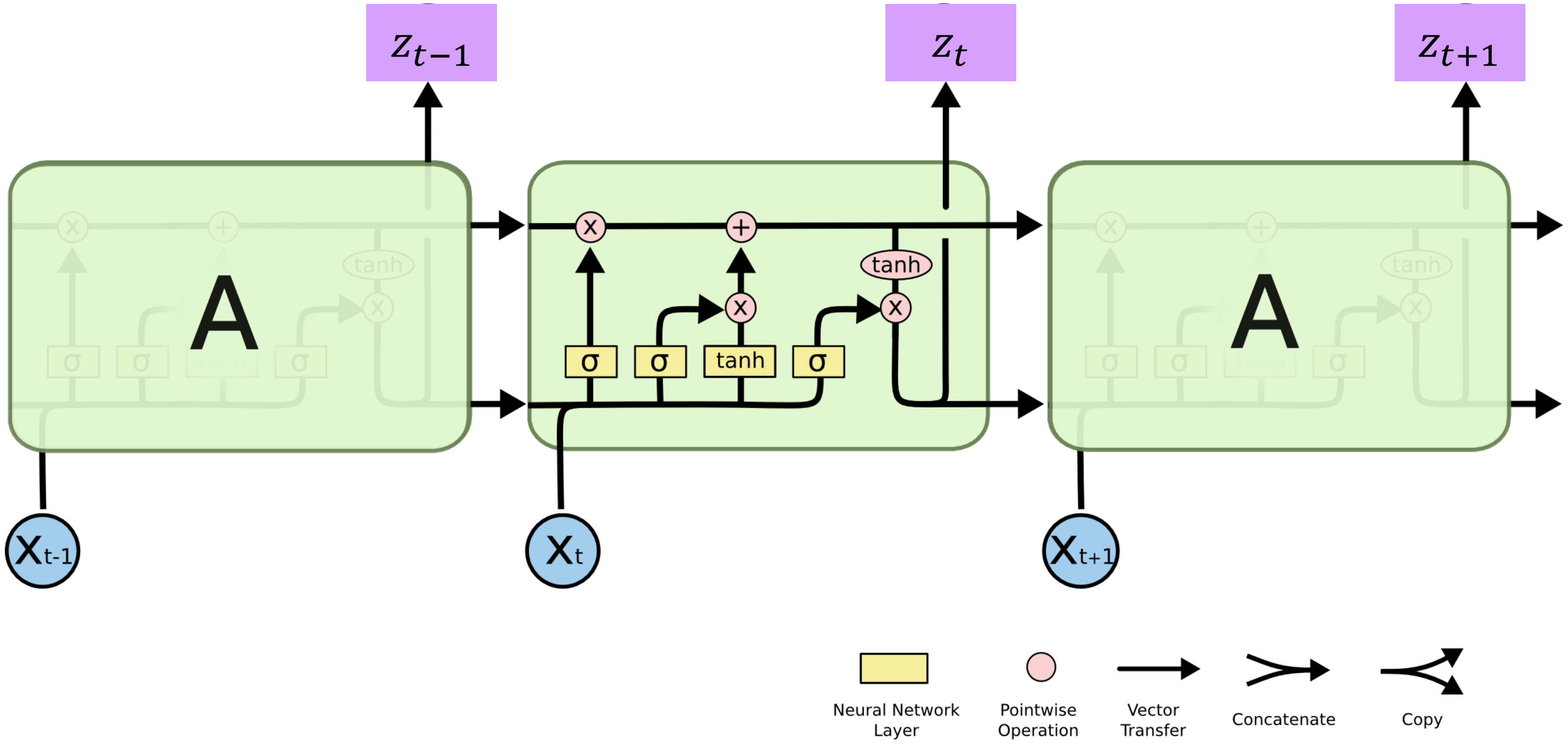


- **Motivation**: Vanishing gradients happen because we multiply many gradients across time, and we want some ways to prevent that

Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory (LSTM)

- LSTM introduces **cell states** c_t
  - Each repeating module has **three gates** to update and control the cell state: **forget gate**, **input gate**, and **output gate**

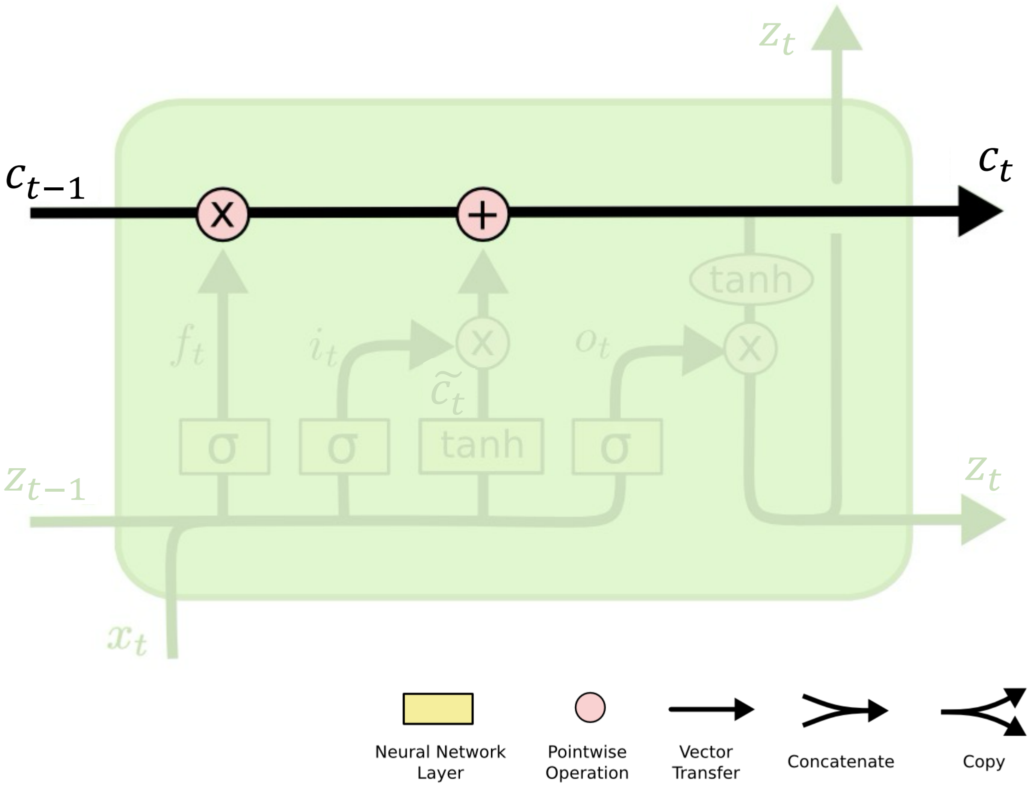


$$f_t = \sigma\left(W_f[z_{t-1}, x_t]\right)$$
$$i_t = \sigma(W_i[z_{t-1}, x_t])$$
$$\widetilde{c_t} = \tanh(W_c[z_{t-1}, x_t])$$
$$c_t = f_t * c_{t-1} + i_t * \widetilde{c_t}$$
$$o_t = \sigma(W_o[z_{t-1}, x_t])$$
$$z_t = o_t * \tanh(c_t)$$

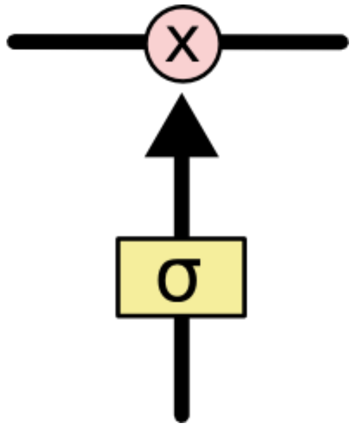
Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: Cell State

- **Cell states c_t**
  - Runs straight down the entire chain, with only some linear interactions
  - In this way, information can flow through time without gradients vanishing



- To update and control the cell states, LSTM introduced **gates**:
  - A sigmoid neural network and a pointwise multiplication operation
  - The **sigmoid layer** outputs numbers between 0 and 1, controlling **how much information could go through**

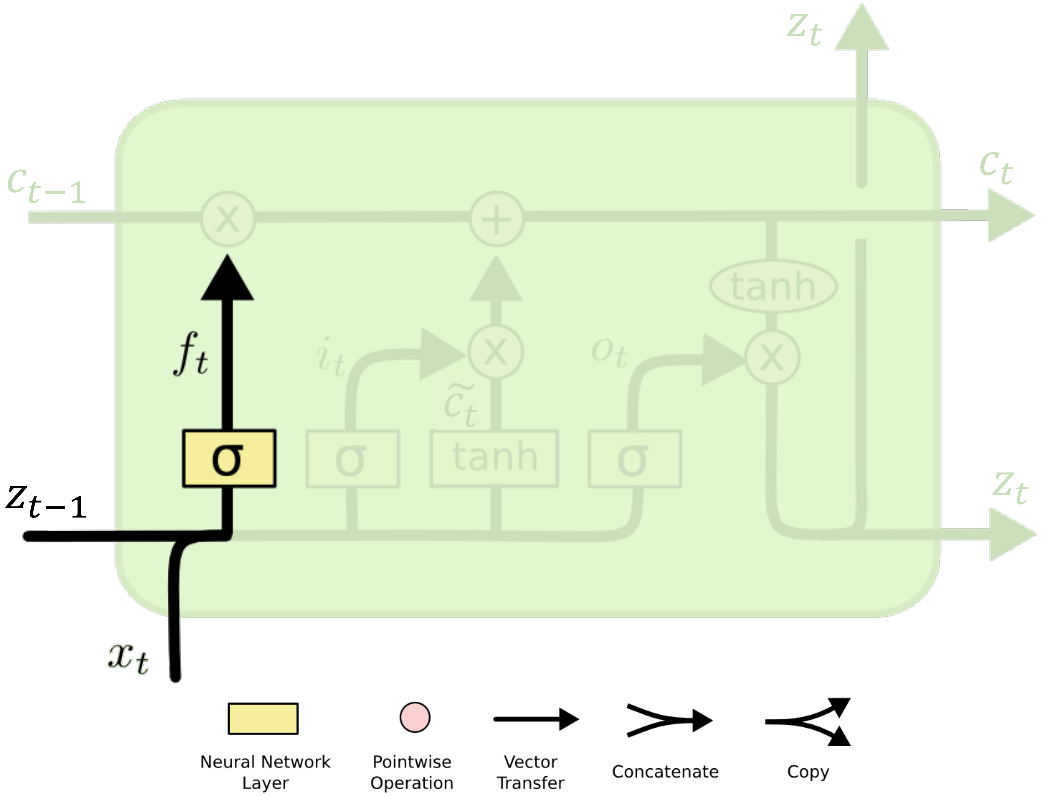


Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: Forget Gate

- Forget gate controls how much information to forget from the previous cell state c_{t-1}

$$f_t = \sigma\left(W_f\left[z_{t-1}, x_t\right]\right)$$



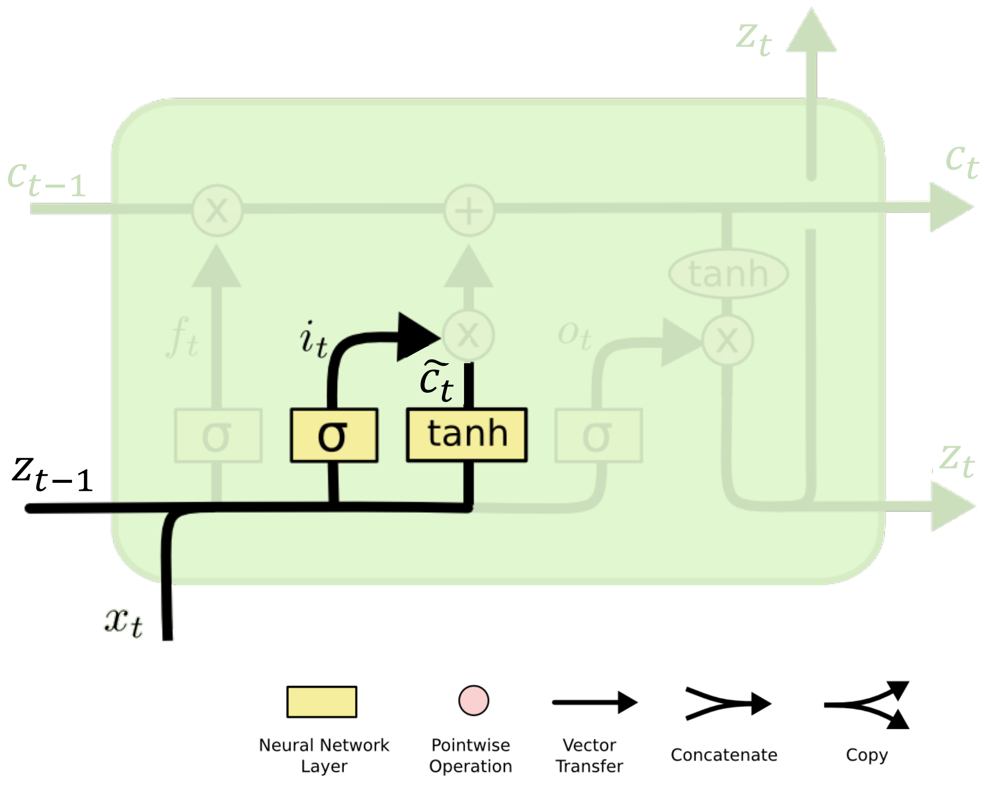
Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: Input Gate

- Input gate decides which new information to store in the cell state

$$i_t = \sigma(W_i[z_{t-1}, x_t])$$
$$\tilde{c}_t = \tanh(W_c[z_{t-1}, x_t])$$

Then update cell state by
$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$

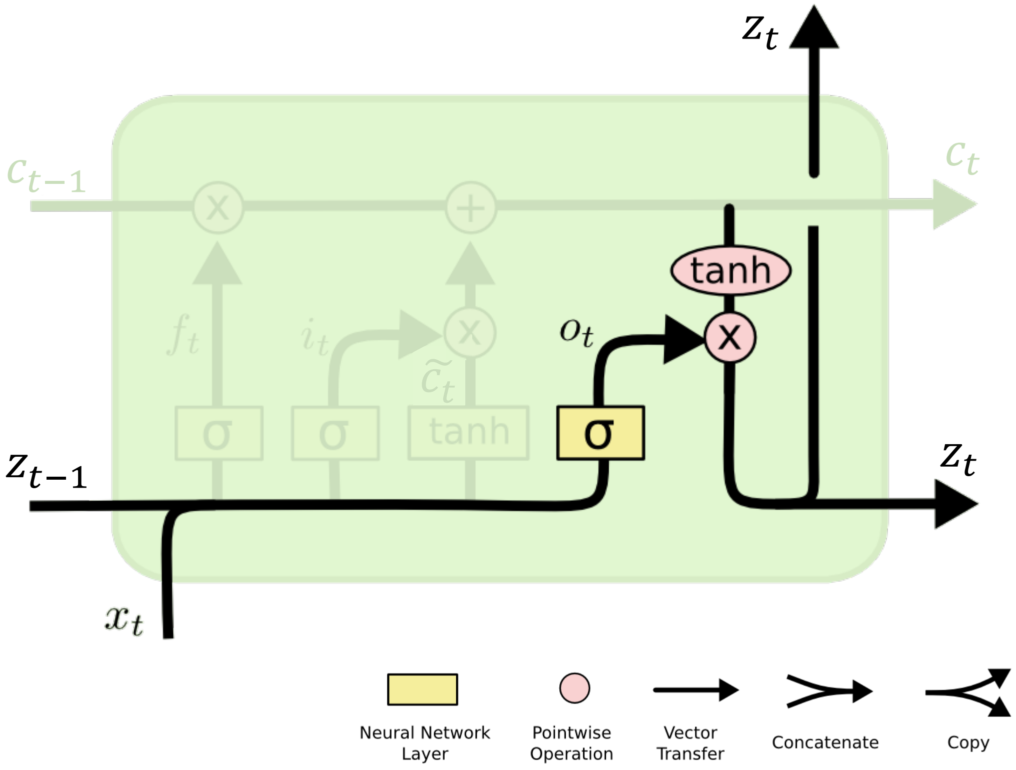


Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: Output Gate

- Output gate decides what to output

$$o_t = \sigma(W_o[z_{t-1}, x_t])$$
$$z_t = o_t * \tanh(c_t)$$



Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Why LSTMs work

- **Forget gate** f_t: decides what to forget from the previous cell state
- **Input gate** i_t: decides what new information to add
- **Output gate** o_t: decides what to output to the next layer
- Constant path through **cell states** c_t helps prevent vanishing gradients

$$f_t = \sigma\left(W_f[z_{t-1}, x_t]\right) \qquad \text{Forget gate}$$

$$i_t = \sigma(W_i[z_{t-1}, x_t]) \qquad \text{Input gate}$$

$$\widetilde{c_t} = \tanh(W_c[z_{t-1}, x_t])$$

$$c_t = f_t * c_{t-1} + i_t * \widetilde{c_t} \qquad \text{Update cell state}$$

$$o_t = \sigma(W_o[z_{t-1}, x_t]) \qquad \text{Output gate}$$

$$z_t = o_t * \tanh(c_t)$$

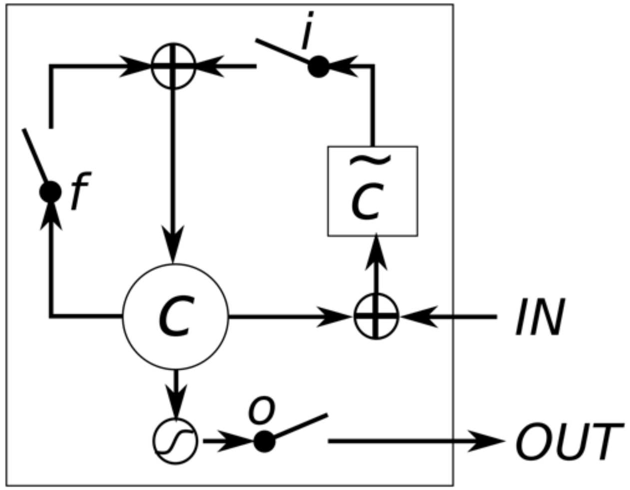
Other Variants: Gated Recurrent Neural Networks

- 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 Unit (GRU)**
- On a high level, rather than using forget, input and output gates like LSTM
- GRU uses a weighted sum of two hidden states

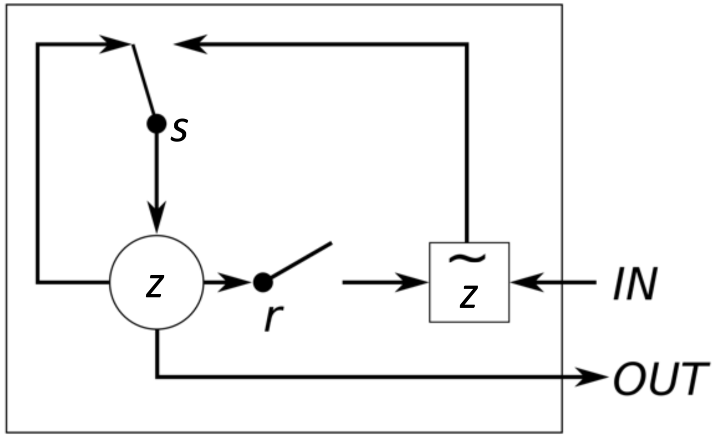
$$z_t = (1 - s_t) \odot z_{t-1} + s_t \odot \widetilde{z}_t, \qquad \widetilde{z}_t = \tanh(W[x_t\,; r_t \odot z_{t-1}])$$

where s_t denotes the update gate

- Empirically, GRUs perform just as well as LSTMs, but much **more efficient** because they have **fewer gates**



(a) Long Short-Term Memory



(b) Gated Recurrent Unit

Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling." (2014) : 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

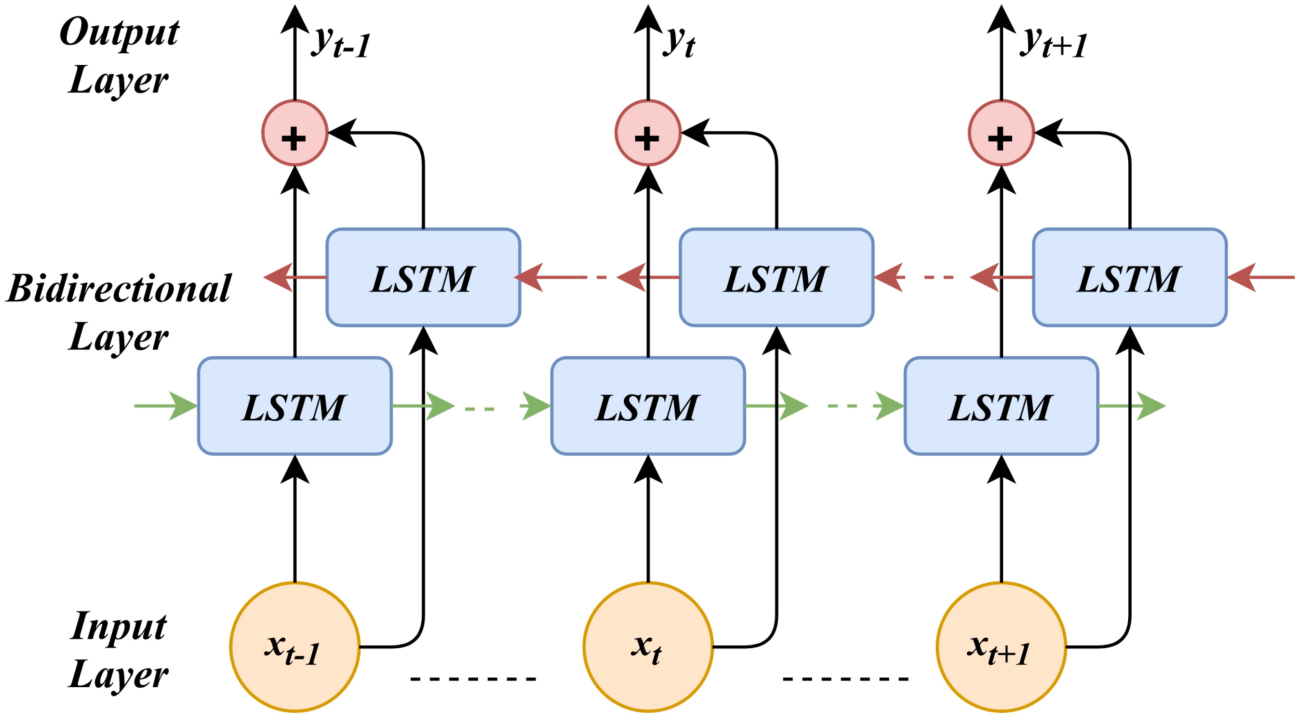


*Figure source: Ihianle et al. "A Deep Learning Approach for Human Activities Recognition From Multimodal Sensing Devices." (2020).

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