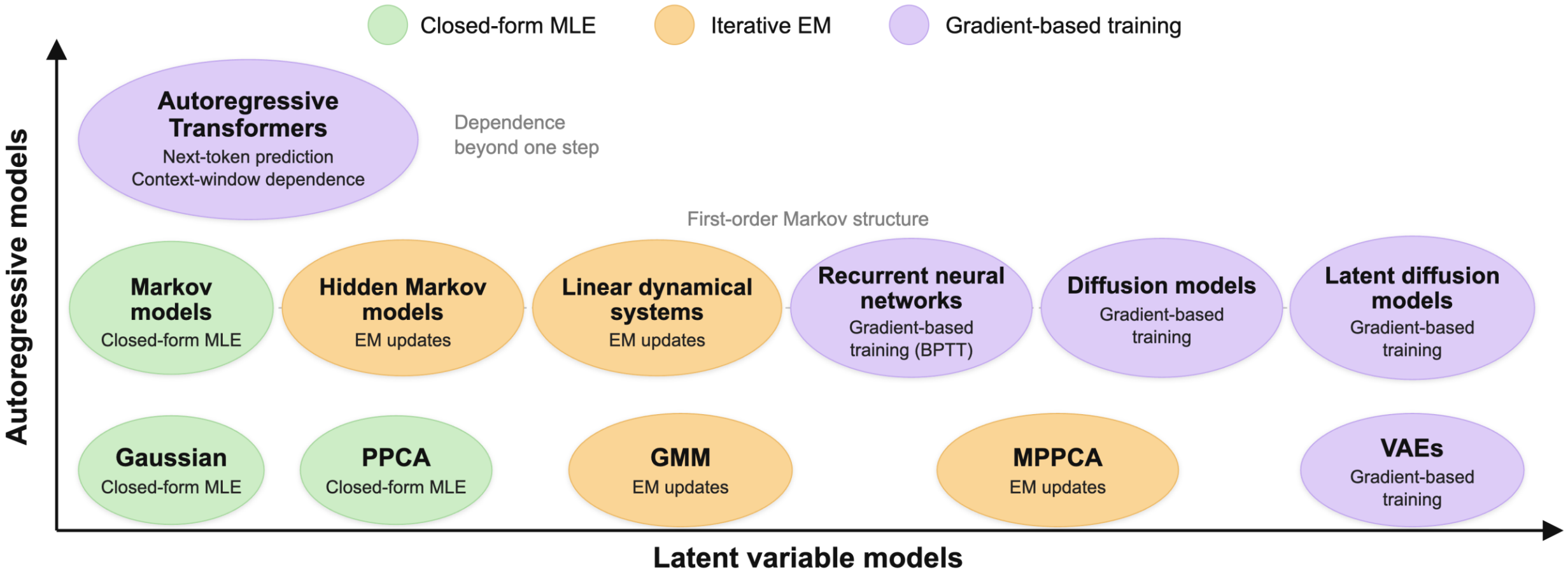
Deep Generative Models: Recurrent Neural Networks

Fall Semester 2026

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



Penn
University of Pennsylvania

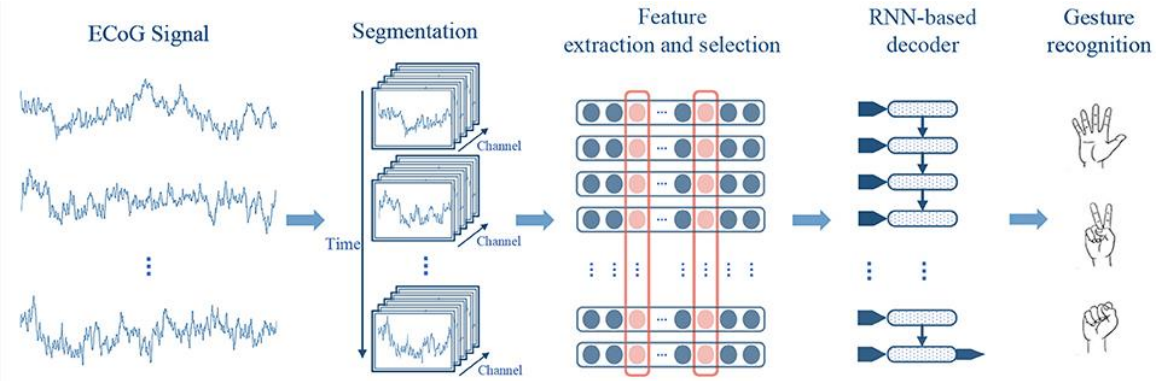
Autoregressive Models

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

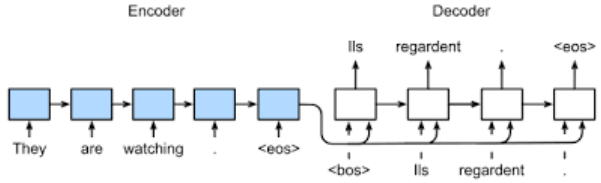
- This lecture: we focus on **Recurrent Neural Networks**
  - Vanilla RNNs
  - Basic applications for Language Modeling
  - Training and Issues with RNNs
  - LSTMs and GRUs

Applications of RNNs

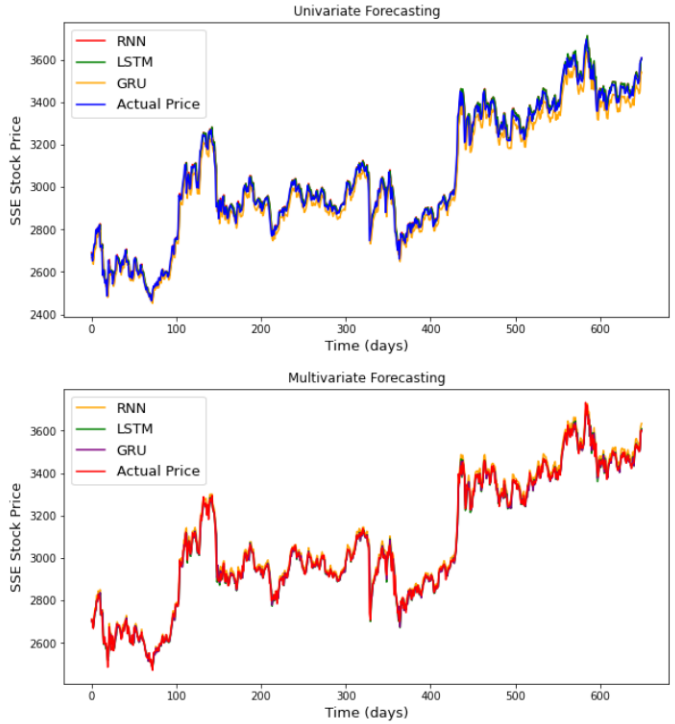
- NLP: Machine Translation, Text Classification, POS Tagging
- Healthcare: Gesture Forecasting, EGG
- Computer Vision: Self-driving, Image/Texture Classification
- Finance: Stock Price Forecasting
- Many, many more



Healthcare: Gesture Forecasting



NLP: Machine Translation

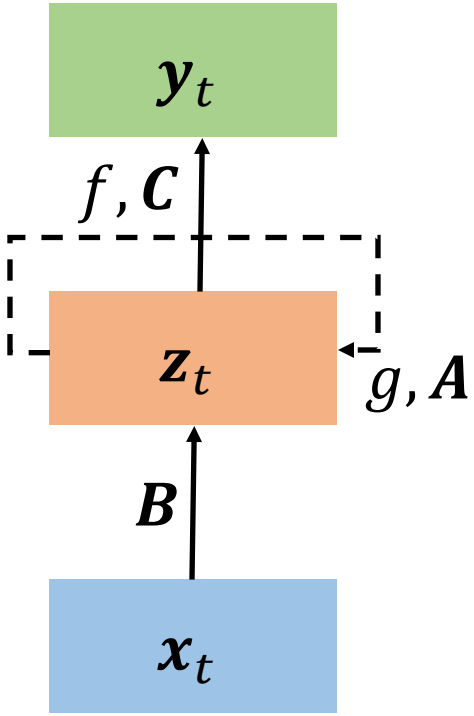


Finance: Stock Forecasting

Recurrent Neural Network (RNNs)

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

$$\boldsymbol{z}_t = g(\boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t) + \boldsymbol{w}_t$$
$$\boldsymbol{y}_t = f(\boldsymbol{C}\boldsymbol{z}_t) + \boldsymbol{v}_t$$



- Here
  - $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T \in \mathbb{R}^D$ denote the **inputs**
  - $\boldsymbol{y}_0, \ldots, \boldsymbol{y}_T \in \mathbb{R}^m$ denote the **outputs**
  - $\boldsymbol{z}_0, \ldots, \boldsymbol{z}_T \in \mathbb{R}^d$ denote the **hidden states**, with $\boldsymbol{z}_0$ the initial state
  - $\boldsymbol{A} \in \mathbb{R}^{d\times d}, \boldsymbol{B} \in \mathbb{R}^{d\times D}, \boldsymbol{C} \in \mathbb{R}^{m\times d}$ are **weight matrices**
  - f and g are nonlinear functions (e.g. f can be a Softmax function for soft classification)
  - No noise $\boldsymbol{w}_t$, $\boldsymbol{v}_t$ when RNN used for prediction instead of generation.

RNNs vs LDSs

- Linear Dynamic Systems

$$\boldsymbol{z}_t = \boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t + \boldsymbol{w}_t, \qquad \boldsymbol{w}_t \sim \mathcal{N}(0, \boldsymbol{Q})$$
$$\boldsymbol{y}_t = \boldsymbol{C}\boldsymbol{z}_t + \boldsymbol{v}_t, \qquad \boldsymbol{v}_t \sim \mathcal{N}(0, \boldsymbol{R})$$

- Everything is linear
- Can be deterministic or stochastic
- Distributions of $\boldsymbol{z}_t$ and $\boldsymbol{y}_t$ have closed-form due the Gaussian assumption
- Exact inference via **Kalman filter**
- Parameter learning via **EM algorithm**

- Recurrent Neural Networks

$$\boldsymbol{z}_t = g(\boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t) + \boldsymbol{w}_t, \qquad \boldsymbol{w}_t \sim \mathcal{N}(0, \boldsymbol{Q})$$
$$\boldsymbol{y}_t = f(\boldsymbol{C}\boldsymbol{z}_t) + \boldsymbol{v}_t\,, \qquad \boldsymbol{v}_t \sim \mathcal{N}(0, \boldsymbol{R})$$

- Has nonlinearity from f and g
- Can be deterministic or stochastic
- Distributions of $\boldsymbol{z}_t$ and $\boldsymbol{y}_t$ do 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.

$$\boldsymbol{z}_t = g(\boldsymbol{A}\boldsymbol{z}_{t-1}) + \boldsymbol{w}_t$$
$$\boldsymbol{y}_t = f(\boldsymbol{C}\boldsymbol{z}_t) + \boldsymbol{v}_t$$

- 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

$$p(\boldsymbol{z}_t \mid \boldsymbol{z}_{t-1}) = \mathcal{N}(g(\boldsymbol{A}\boldsymbol{z}_{t-1}), \boldsymbol{Q})$$
$$p(\boldsymbol{y}_t \mid \boldsymbol{z}_t) = \mathcal{N}(f(\boldsymbol{C}\boldsymbol{z}_t), \boldsymbol{R})$$

- On the other hand, even if $\boldsymbol{z}_0$ is Gaussian, $\boldsymbol{z}_1 = g(\boldsymbol{A}\boldsymbol{z}_0) + \boldsymbol{w}_1$ 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 $(\boldsymbol{\mu}, \boldsymbol{\Sigma})$
  - The Kalman filter tracks the evolution of the mean and covariance of $\boldsymbol{z}_t \mid \boldsymbol{y}_{1:t-1}$. If this is not Gaussian, then we cannot track that anymore.

$$\boldsymbol{K}_t = \hat{\boldsymbol{\Sigma}}_{t|t-1}\boldsymbol{C}^\top\left(\boldsymbol{C}\hat{\boldsymbol{\Sigma}}_{t|t-1}\boldsymbol{C}^\top + \boldsymbol{R}\right)^{-1}$$
$$\hat{\boldsymbol{z}}_{t+1|t} = \boldsymbol{A}\hat{\boldsymbol{z}}_{t|t-1} + \boldsymbol{A}\boldsymbol{K}_t(\boldsymbol{y}_t - \boldsymbol{C}\hat{\boldsymbol{z}}_{t|t-1})$$
$$\hat{\boldsymbol{\Sigma}}_{t+1|t} = \boldsymbol{A}\left(\hat{\boldsymbol{\Sigma}}_{t|t-1} - \boldsymbol{K}_t\boldsymbol{C}\hat{\boldsymbol{\Sigma}}_{t|t-1}\right)\boldsymbol{A}^\top + \boldsymbol{Q}$$

Extended Kalman Filters for RNNs

$$\boldsymbol{z}_t = g(\boldsymbol{A}\boldsymbol{z}_{t-1}) + \boldsymbol{w}_t$$
$$\boldsymbol{y}_t = f(\boldsymbol{C}\boldsymbol{z}_t) + \boldsymbol{v}_t$$

- 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 $\boldsymbol{J}_f, \boldsymbol{J}_g$ of f and g.

$$\tilde{\boldsymbol{z}}_t = \tilde{\boldsymbol{A}}_t \tilde{\boldsymbol{z}}_{t-1} + \boldsymbol{w}_t$$
$$\boldsymbol{y}_t = \tilde{\boldsymbol{C}}_t \tilde{\boldsymbol{z}}_t + \boldsymbol{v}_t$$

$$\tilde{\boldsymbol{A}}_t := \boldsymbol{J}_g\left(\boldsymbol{A}\hat{\boldsymbol{z}}_{t-1|t-1}\right)\boldsymbol{A}$$
$$\tilde{\boldsymbol{C}}_t := \boldsymbol{J}_f\left(\boldsymbol{C}\hat{\boldsymbol{z}}_{t|t-1}\right)\boldsymbol{C}$$

- Prediction

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

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

Update

$$\boldsymbol{K}_t = \hat{\boldsymbol{\Sigma}}_{t|t-1}\boldsymbol{C}^\top\left(\boldsymbol{C}\hat{\boldsymbol{\Sigma}}_{t|t-1}\boldsymbol{C}^\top + \boldsymbol{R}\right)^{-1}$$
$$\hat{\boldsymbol{z}}_{t|t} = \hat{\boldsymbol{z}}_{t|t-1} + \boldsymbol{K}_t(\boldsymbol{y}_t - \boldsymbol{C}\hat{\boldsymbol{z}}_{t|t-1})$$
$$\hat{\boldsymbol{\Sigma}}_{t|t} = \hat{\boldsymbol{\Sigma}}_{t|t-1} - \boldsymbol{K}_t\boldsymbol{C}\hat{\boldsymbol{\Sigma}}_{t|t-1}$$

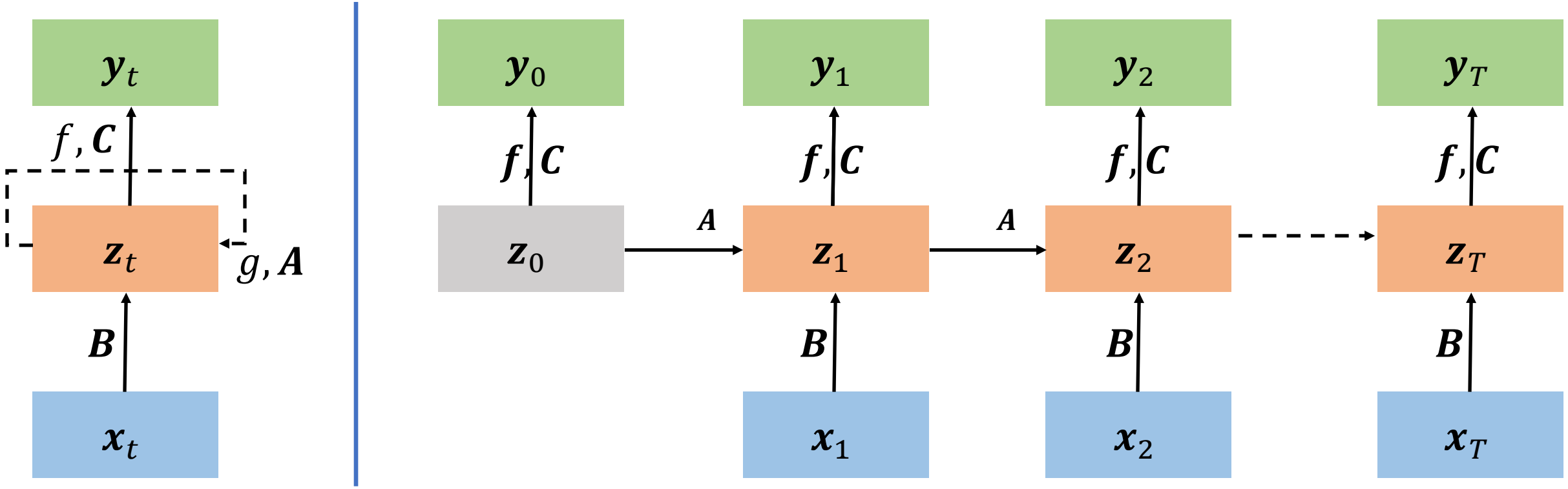
$$\boldsymbol{K}_t = \hat{\boldsymbol{\Sigma}}_{t|t-1}\tilde{\boldsymbol{C}}_t^\top\left(\tilde{\boldsymbol{C}}_t\hat{\boldsymbol{\Sigma}}_{t|t-1}\tilde{\boldsymbol{C}}_t^\top + \boldsymbol{R}\right)^{-1}$$
$$\hat{\boldsymbol{z}}_{t|t} = \hat{\boldsymbol{z}}_{t|t-1} + \boldsymbol{K}_t(\boldsymbol{y}_t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

- We do not have any optimality guarantees.

Unrolling and Parameter Tying

$$\boldsymbol{z}_t = g(\boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t)$$
$$\boldsymbol{y}_t = f(\boldsymbol{C}\boldsymbol{z}_t)$$

- 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



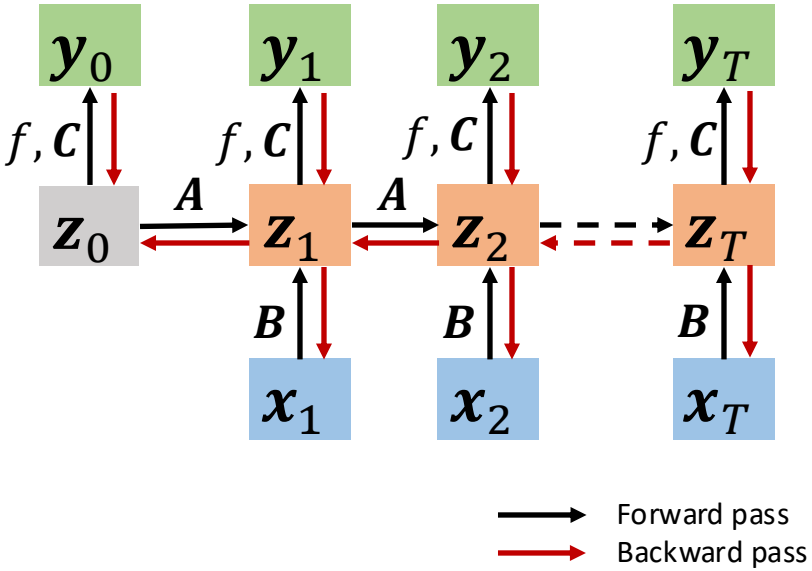
- Here $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ are the same matrices for all timestep, known as **Parameter Tying**

Note: Here we omit the noise terms $\boldsymbol{w}_t$ and $\boldsymbol{v}_t$ 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

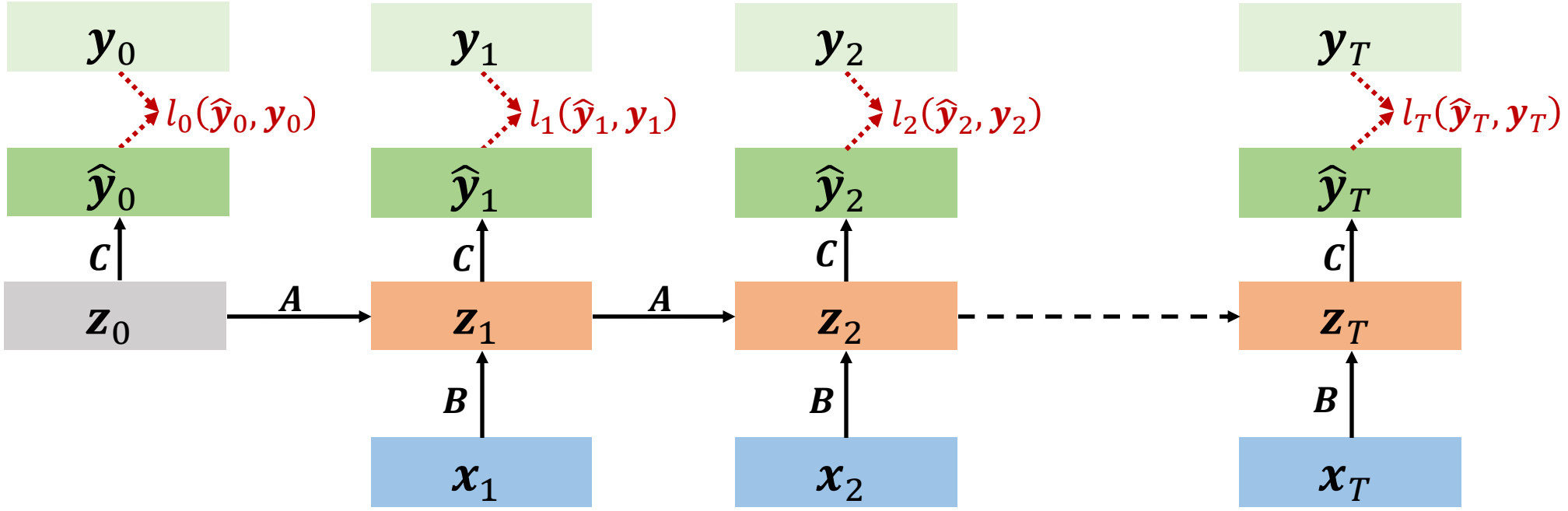
Loss Computation in Time

$$\boldsymbol{z}_t = g(\boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t)$$
$$\hat{\boldsymbol{y}}_t = f(\boldsymbol{C}\boldsymbol{z}_t)$$

- Given $(\boldsymbol{x}_{1:T}, \boldsymbol{y}_{0:T})$, we can define different losses
- For a task that requires prediction at each time step $\hat{\boldsymbol{y}}_t$, we can compute the loss $l_t(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t)$ for each timestep and sum over all timesteps

$$\mathcal{L}(\hat{\boldsymbol{y}}_{0:T}, \boldsymbol{y}_{0:T}) = \sum_{t=0}^{T} l_t\,(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t)$$

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



Application of RNNs: Next Word Prediction

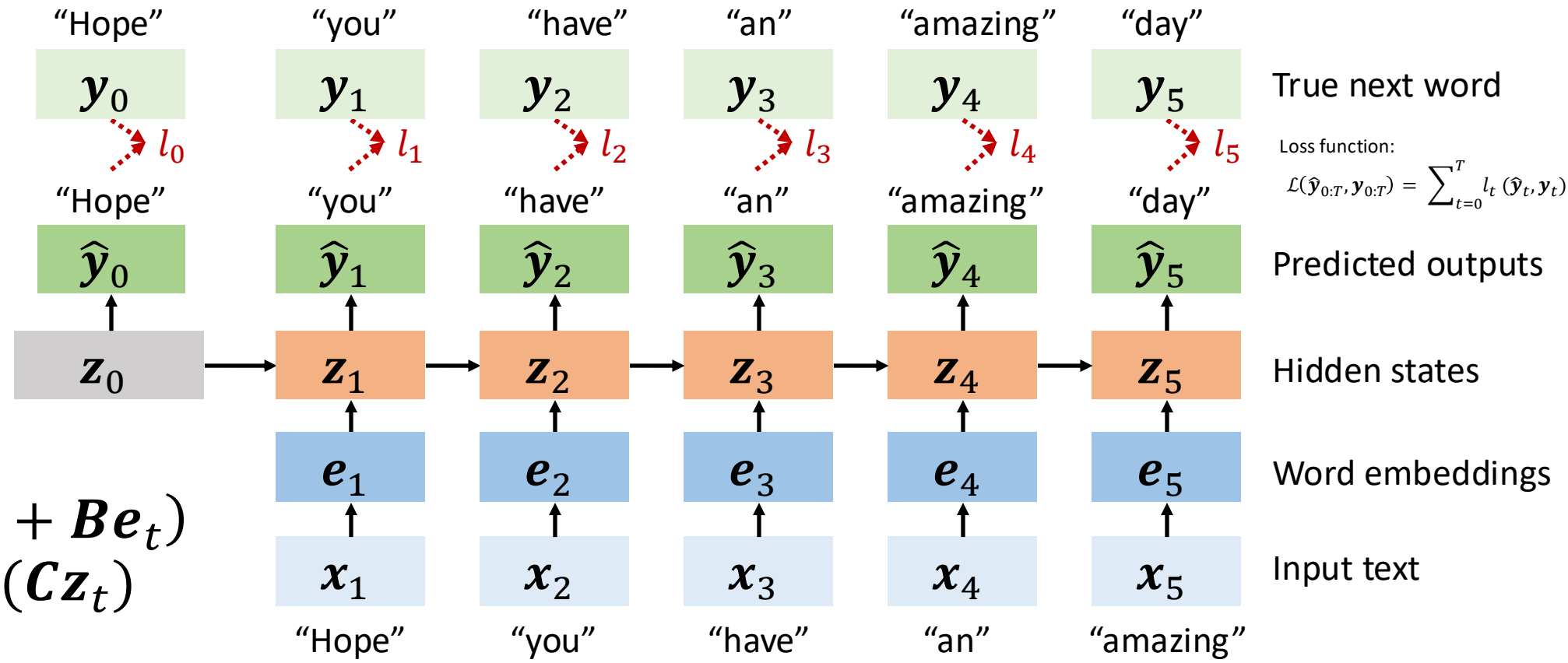
- Let us consider using an RNN for a **language modeling task**. Given some preceding context, we want the language model to predict the next word:

$$p(\boldsymbol{y}_t = \underbrace{\text{“day”}}_{\text{Next word}} | \, \boldsymbol{y}_{0:t-1} = \underbrace{\text{“Hope you have an amazing”}}_{\text{Context}})$$

- Suppose we have a set of N sentences $\left\{\boldsymbol{y}_{0:T_i}^{(i)}\right\}_{i=1}^{N}$, where $\boldsymbol{y}_{0:T_i}^{(i)} = \left[\boldsymbol{y}_0, \ldots, \boldsymbol{y}_{T_i}\right]$ 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** $\boldsymbol{E} \in \mathbb{R}^{D \times |V|}$, 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 $\boldsymbol{y}_t$ from our vocabulary, which is a discrete choice. In this case, we can use the Softmax function for modeling the distribution $p(\boldsymbol{y}_t \mid \boldsymbol{z}_t)$
- Using BPTT, we apply cross-entropy loss on the prediction of each timestep

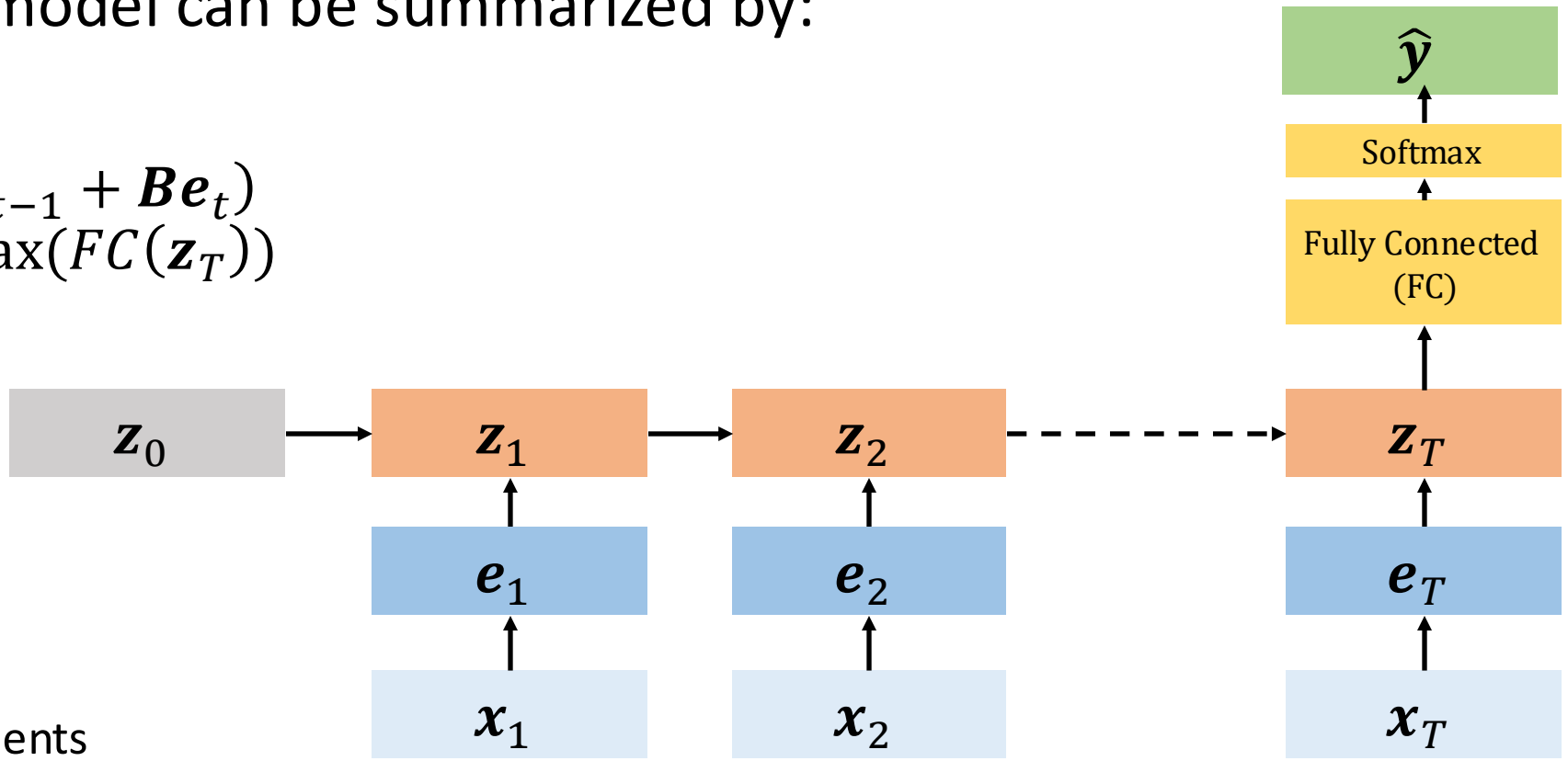


$$\boldsymbol{e}_t = \boldsymbol{E}\boldsymbol{x}_t$$
$$\boldsymbol{z}_t = g(\boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{B}\boldsymbol{e}_t)$$
$$\hat{\boldsymbol{y}}_t = \mathrm{Softmax}(\boldsymbol{C}\boldsymbol{z}_t)$$

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:

$$\boldsymbol{e}_t = \boldsymbol{E}\boldsymbol{x}_t$$
$$\boldsymbol{z}_t = g(\boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{B}\boldsymbol{e}_t)$$
$$\hat{\boldsymbol{y}} = \text{Softmax}(FC(\boldsymbol{z}_T))$$



Category: *Technology*

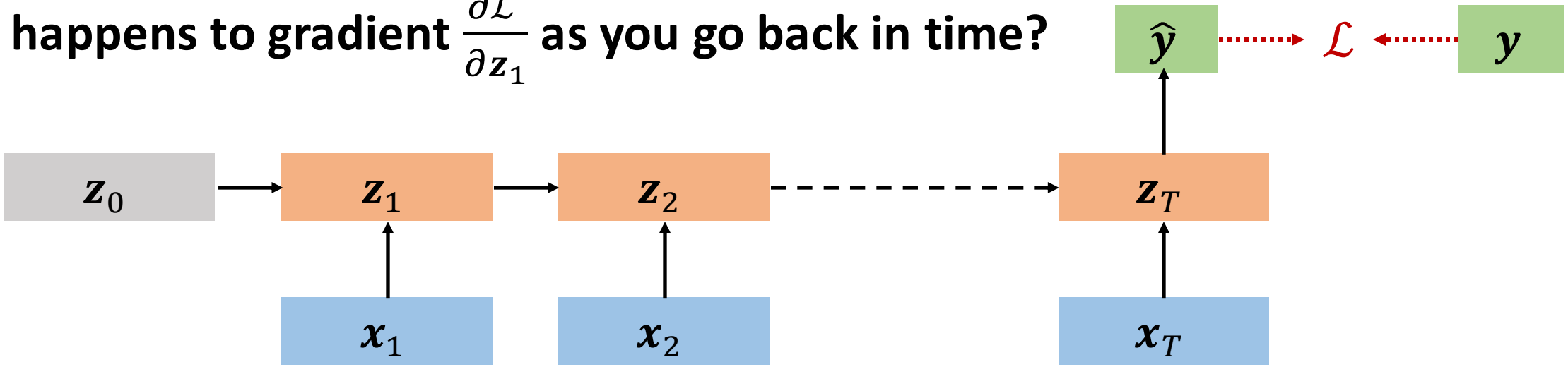
Title: "Penn students innovate in the time of AI!"

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:

$$\boldsymbol{z}_t = g(\boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t)$$
$$\hat{\boldsymbol{y}} = \boldsymbol{C}\boldsymbol{z}_T$$

- **What happens to gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_1}$ as you go back in time?**



$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_1} = \frac{\partial \boldsymbol{z}_2}{\partial \boldsymbol{z}_1} \cdot \frac{\partial \boldsymbol{z}_3}{\partial \boldsymbol{z}_2} \cdots \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}_T} \cdot \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} = \boldsymbol{A}^\top \boldsymbol{A}^\top \boldsymbol{A}^\top \cdots \boldsymbol{C}^\top \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} = (\boldsymbol{C}\boldsymbol{A}^{T-1})^\top \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}}$$

Assuming g = identity

If the eigenvalues of $\boldsymbol{A}$ are less than 1, **gradients vanish**;
If greater than 1, **gradients explode**.

Exploding/Vanishing Gradients: LDS case

- More generally,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_t} = (\boldsymbol{C}\boldsymbol{A}^{T-t})^\top \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \Rightarrow \frac{\partial \mathcal{L}}{\partial \boldsymbol{A}} = \sum_t \frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{A}} \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_t} = \sum_t \frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{A}} \cdot (\boldsymbol{C}\boldsymbol{A}^{T-t})^\top \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}}$$

- Let $\lambda_1(\boldsymbol{A})$ be the maximum eigenvalue of $\boldsymbol{A}$.
- For any initial condition $\boldsymbol{z}_0$ and a large $T \to \infty$
  - Exploding: If $|\lambda_1(\boldsymbol{A})| > 1$, $\boldsymbol{A}^T$ will grow to infinity
  - Vanishing: If $|\lambda_1(\boldsymbol{A})| < 1$, $\boldsymbol{A}^T$ will diminish to zero
- Hence, the gradient involving $\boldsymbol{A}^T$ terms will also either explode or vanish.

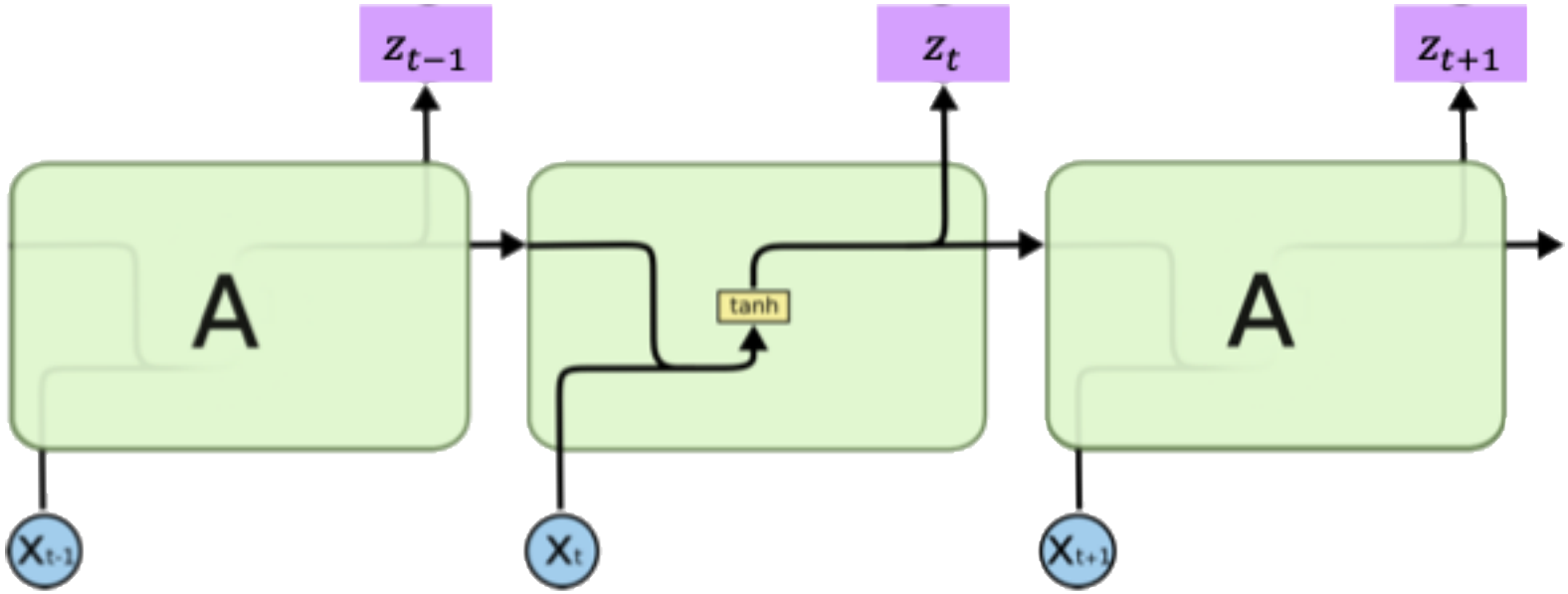
Issues with RNN: Exploding/Vanishing Gradients

- 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**:
  - **Gradient clipping**: rescale the gradient if its norm exceeds a threshold (fixes exploding)
  - **Better optimizers**: e.g., second order or approximate second order methods
  - **Layer normalization**: at each layer to keep gradient norms stable
  - **Clever initializations:** e.g., start with random orthonormal matrices to prevent gradients from vanishing or exploding
  - **Skip connections through time**: give gradients a shorter path to early time steps
- **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

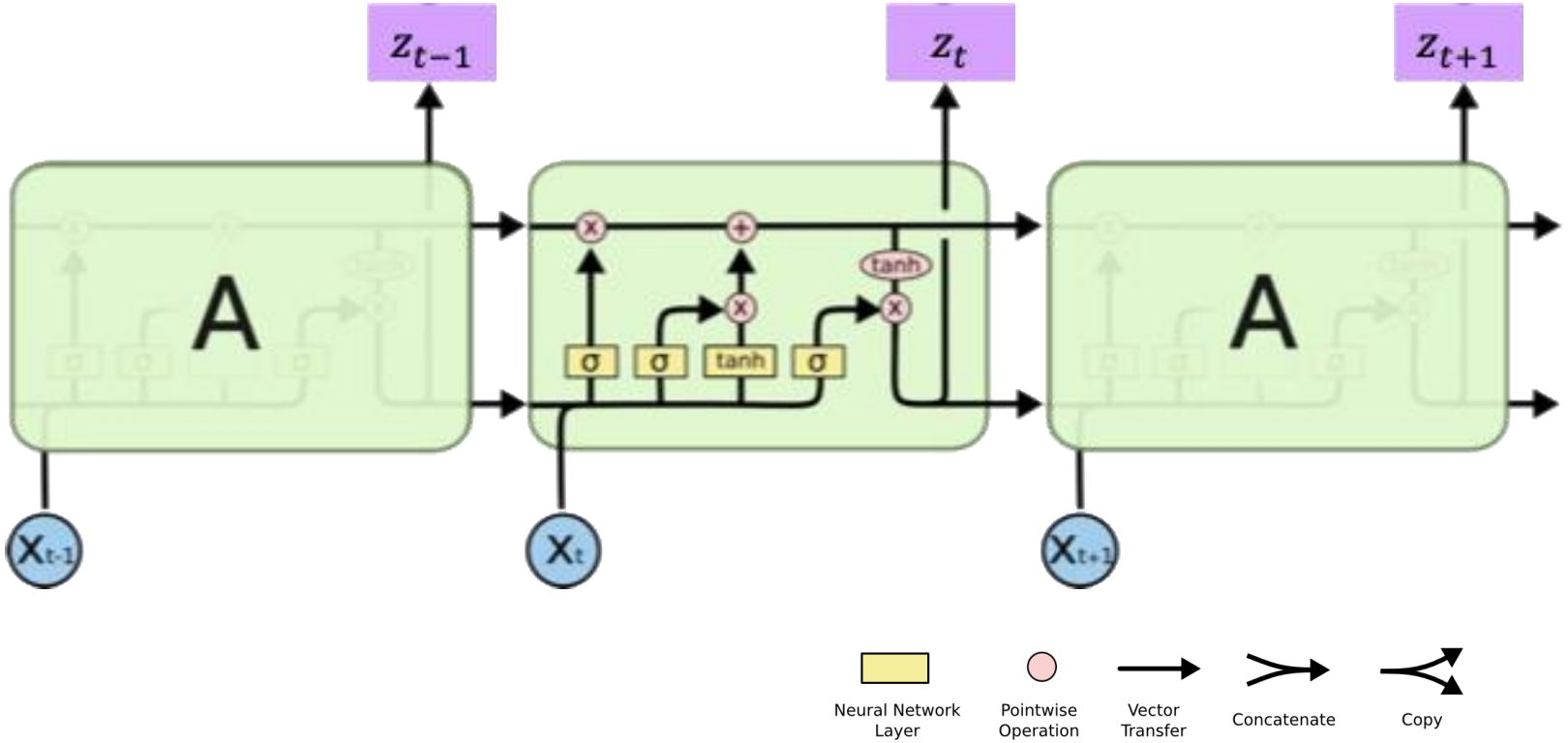


- **Motivation**: Vanishing gradients happen because we multiply many gradients across time, and we want some ways to prevent that

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

Long Short-Term Memory (LSTM)

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

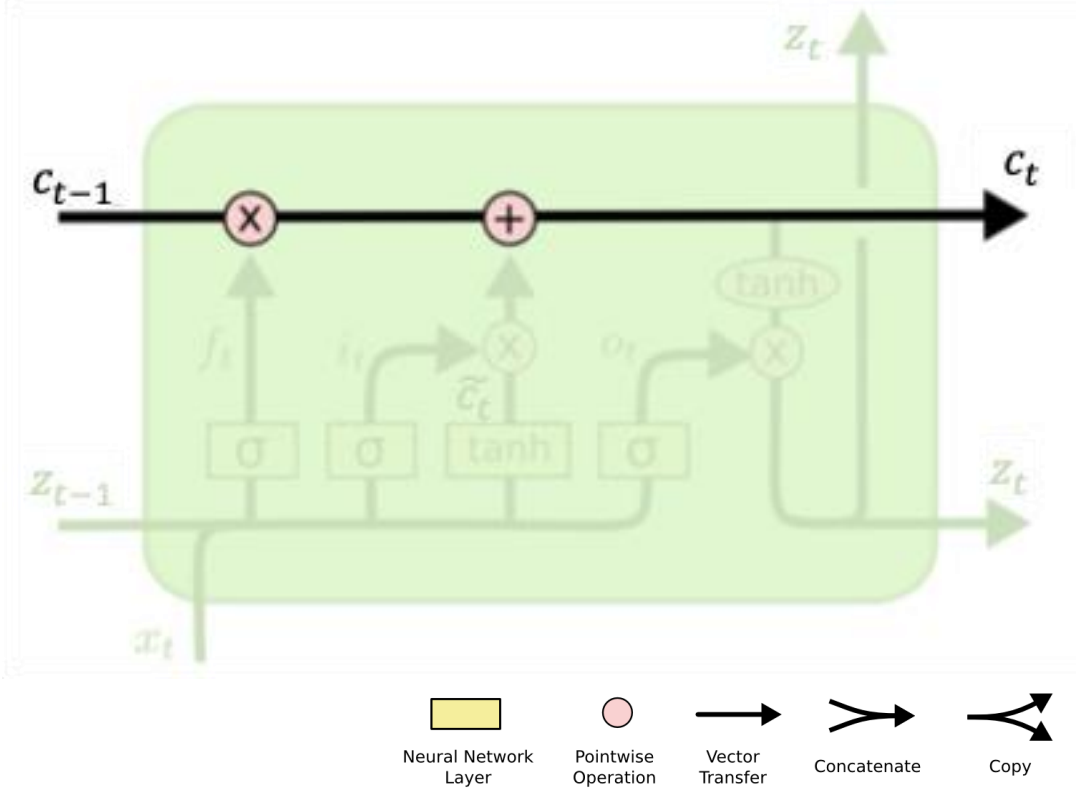


$$\boldsymbol{f}_t = \sigma\big(\boldsymbol{W}_f[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t]\big)$$
$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t])$$
$$\widetilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}_c[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t])$$
$$\boldsymbol{c}_t = \boldsymbol{f}_t * \boldsymbol{c}_{t-1} + \boldsymbol{i}_t * \widetilde{\boldsymbol{c}}_t$$
$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t])$$
$$\boldsymbol{z}_t = \boldsymbol{o}_t * \tanh(\boldsymbol{c}_t)$$

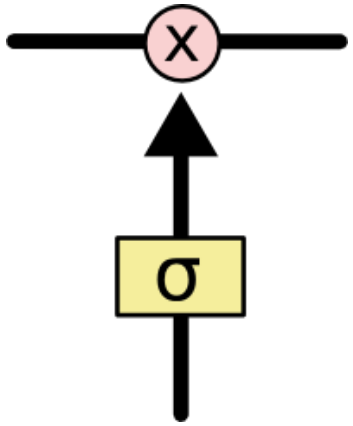
Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

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**

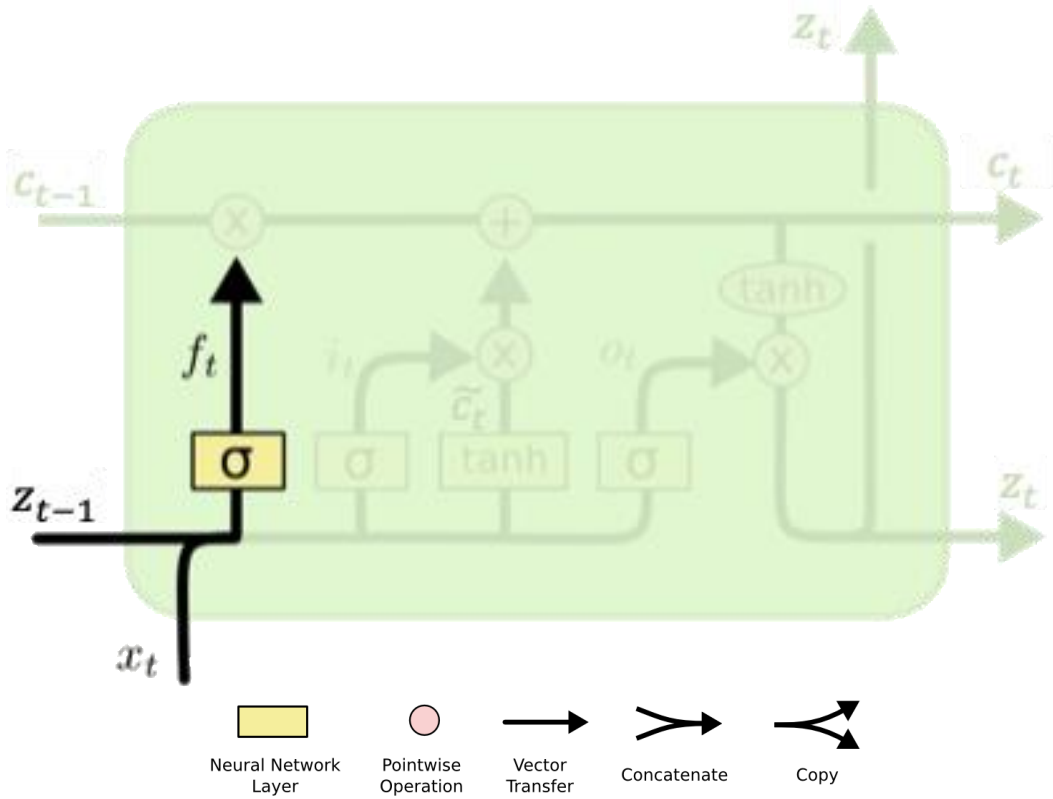


Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: Forget Gate

- Forget gate controls how much information to forget from the previous cell state $\boldsymbol{c}_{t-1}$

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_f[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t]\right)$$



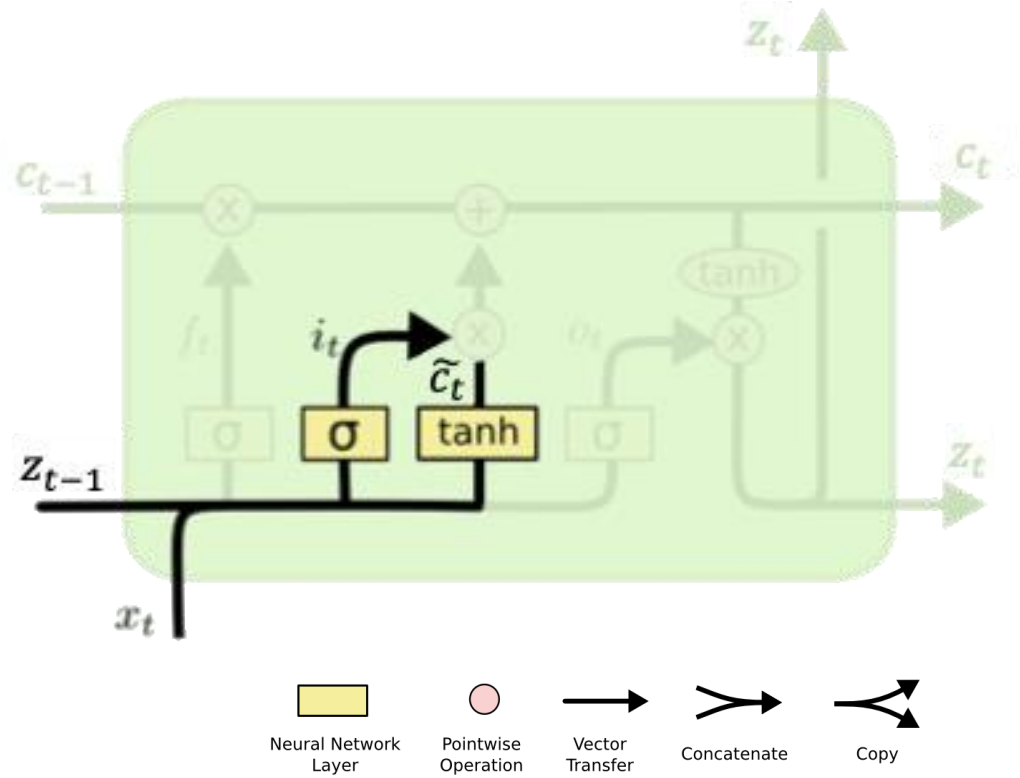
Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: Input Gate

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

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t])$$
$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}_c[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t])$$

Then update cell state by
$\boldsymbol{c}_t = \boldsymbol{f}_t * \boldsymbol{c}_{t-1} + \boldsymbol{i}_t * \tilde{\boldsymbol{c}}_t$

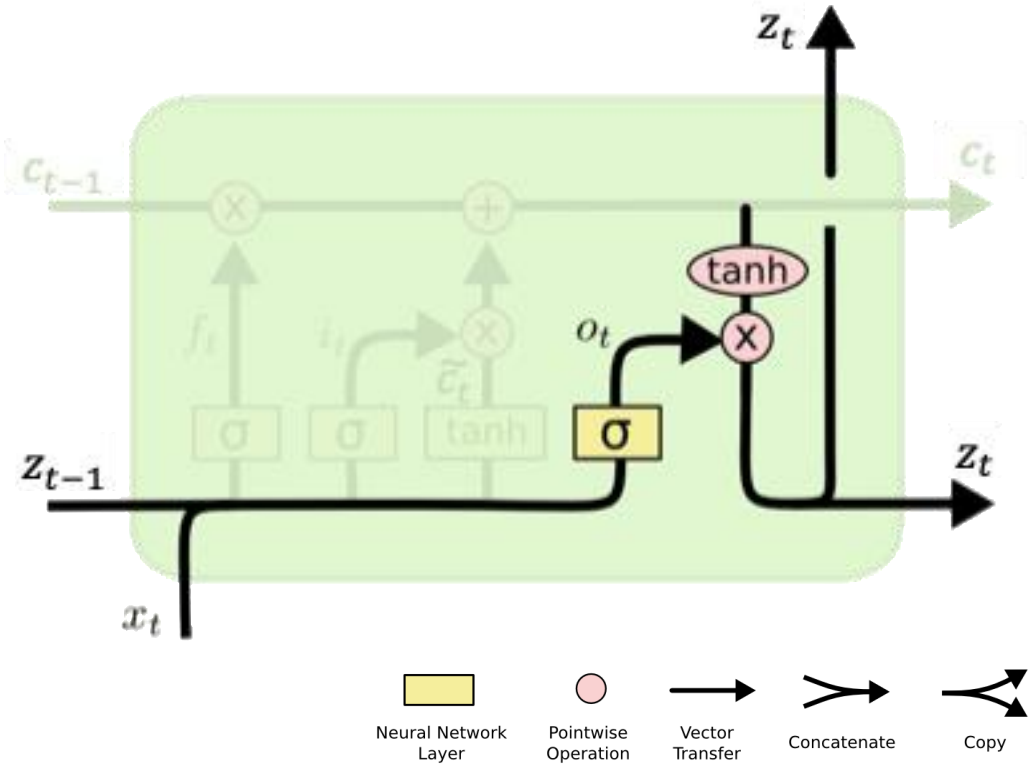


Colah. Blog. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM: Output Gate

- Output gate decides what to output

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t])$$
$$\boldsymbol{z}_t = \boldsymbol{o}_t * \tanh(\boldsymbol{c}_t)$$



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

Why LSTMs work

- **Forget gate $\boldsymbol{f}_t$**: decides what to forget from the previous cell state
- **Input gate $\boldsymbol{i}_t$**: decides what new information to add
- **Output gate $\boldsymbol{o}_t$**: decides what to output to the next layer
- Constant path through **cell states $\boldsymbol{c}_t$** helps prevent vanishing gradients

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_f[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t]\right) \qquad \text{Forget gate}$$

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t]) \qquad \text{Input gate}$$

$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}_c[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t])$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t * \boldsymbol{c}_{t-1} + \boldsymbol{i}_t * \tilde{\boldsymbol{c}}_t \qquad \text{Update cell state}$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t]) \qquad \text{Output gate}$$

$$\boldsymbol{z}_t = \boldsymbol{o}_t * \tanh(\boldsymbol{c}_t)$$

Other Variants: Gated Recurrent Neural Networks

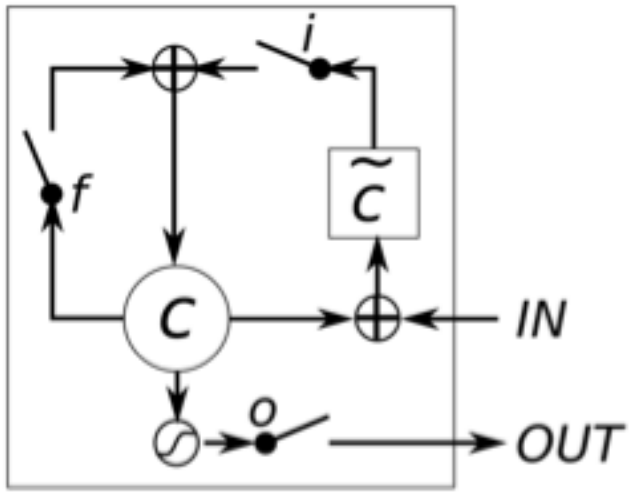
- Another famous variant of the vanilla RNNs is **Gated Recurrent Neural Network**
  - Instead of a memory cell, it uses what's known as a **Gated Recurrent Unit (GRU)**
- At a high level, rather than using forget, input and output gates like LSTM
- GRU uses a weighted sum of two hidden states

$$\boldsymbol{z}_t = (1 - \boldsymbol{s}_t) \odot \boldsymbol{z}_{t-1} + \boldsymbol{u}_t \odot \tilde{\boldsymbol{z}}_t, \qquad \text{where } \tilde{\boldsymbol{z}}_t = \tanh(\boldsymbol{W}[\boldsymbol{x}_t ; \boldsymbol{r}_t \odot \boldsymbol{z}_{t-1}])$$

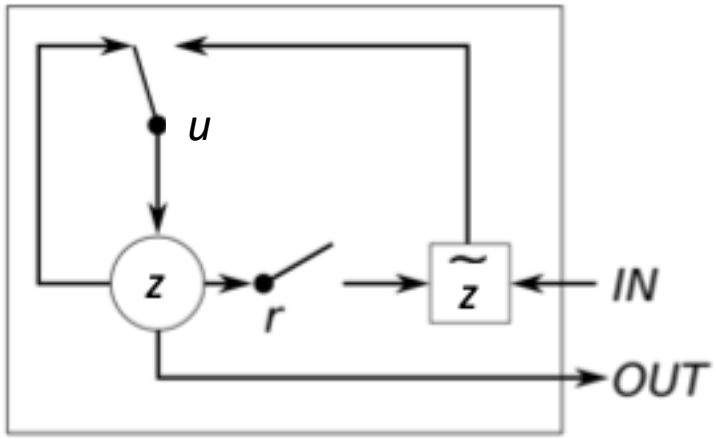
where $\boldsymbol{u}_t$ denotes the **update gate**,

$\boldsymbol{r}_t$ denotes the **reset 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

Gated Recurrent Neural Networks

- **Gated Recurrent Unit (GRU)** has two gates:
  - **Update gate $\boldsymbol{u}_t$**: decides how much the unit update its activation

$$\boldsymbol{u}_t = \sigma(\boldsymbol{W}_u \boldsymbol{x}_t + \boldsymbol{U}_u \boldsymbol{z}_{t-1})$$

  - **Reset gate $\boldsymbol{r}_t$**: decides how much information to forget (reset)

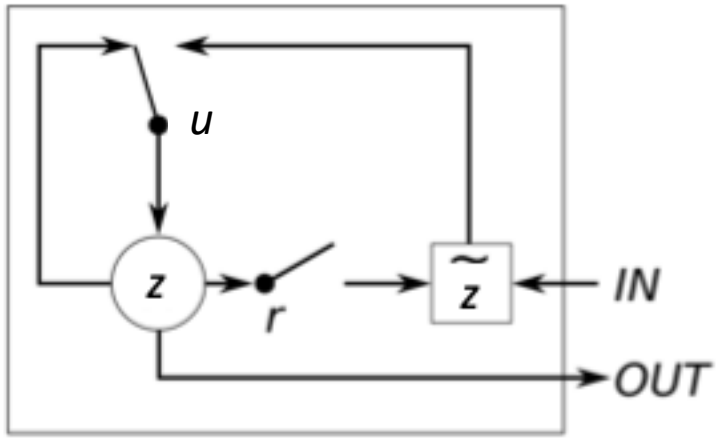
$$\boldsymbol{r}_t = \sigma(\boldsymbol{W}_r \boldsymbol{x}_t + \boldsymbol{U}_r \boldsymbol{z}_{t-1})$$

- Hidden state $\boldsymbol{z}_t$ is updated by a weighted sum of the previous activation $\boldsymbol{z}_{t-1}$ and a candidate activation $\tilde{\boldsymbol{z}}_t$:

$$\boldsymbol{z}_t = (1 - \boldsymbol{u}_t) \odot \boldsymbol{z}_{t-1} + \boldsymbol{u}_t \odot \tilde{\boldsymbol{z}}_t$$

$$\tilde{\boldsymbol{z}}_t = \tanh(\boldsymbol{W}[\boldsymbol{x}_t ; \boldsymbol{r}_t \odot \boldsymbol{z}_{t-1}])$$

where $\odot$ is an element-wise multiplication



(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 $p(\boldsymbol{y}_t \mid \boldsymbol{z}_{\text{forward}}, \boldsymbol{z}_{\text{backward}})$ depends on forward and backward hidden states
- Intuition from **NLP**: knowing a word means knowing what comes before and after the word

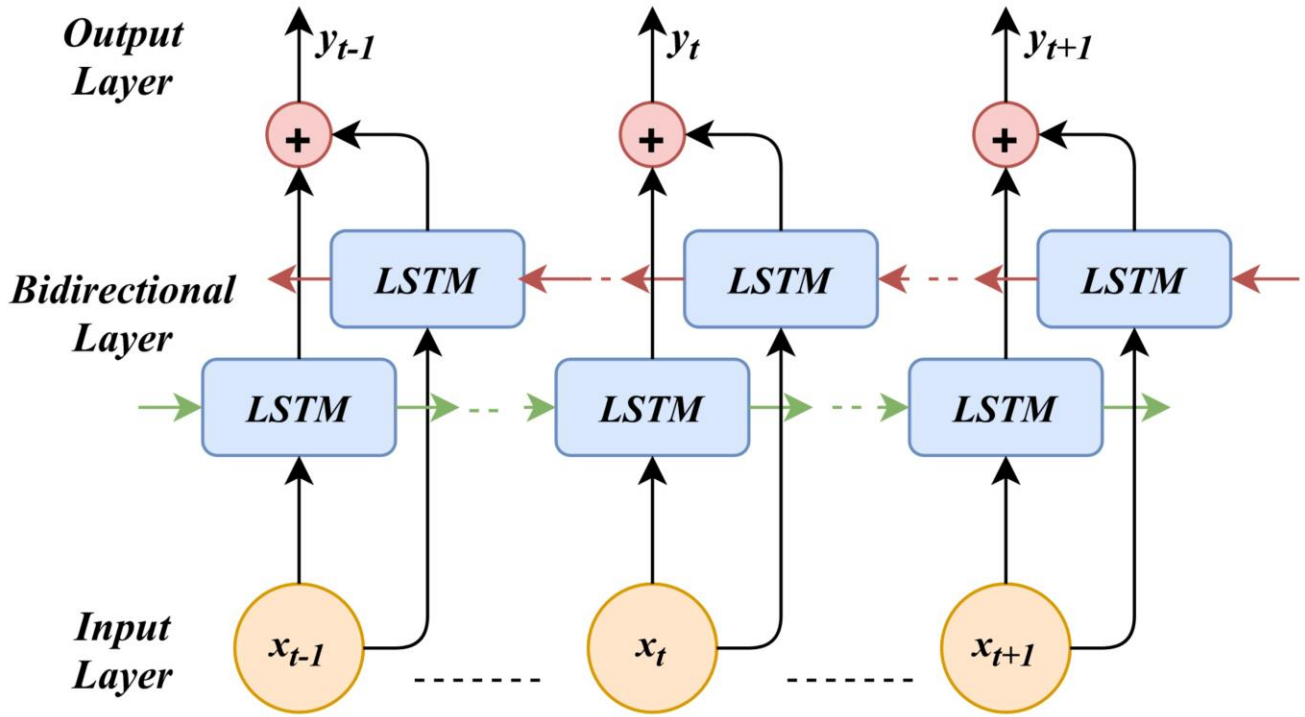


*Figure source: Ihianle et al. "A Deep Learning Approach for Human Activities Recognition From Multimodal Sensing Devices." (2020).

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