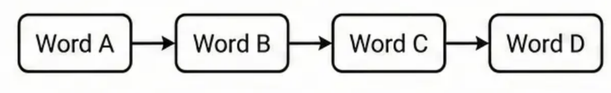
B. Data Structures, Symmetries, and Neural Network Architectures

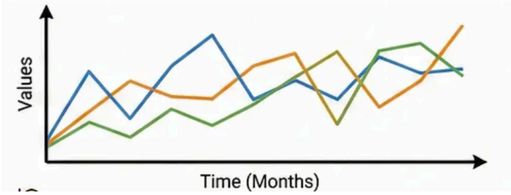
B.1 Tabular data: MLPs, residual connections, and autoencoders
B.2 Grid-structured data: CNNs
B.3 Sequential data: RNNs, LSTMs, and GRUs
B.4 Graph-structured data: GNNs and message passing
B.5 Set-structured data: Deep Sets

Tiam Heydari, PhD
Postdoctoral Fellow, UBC

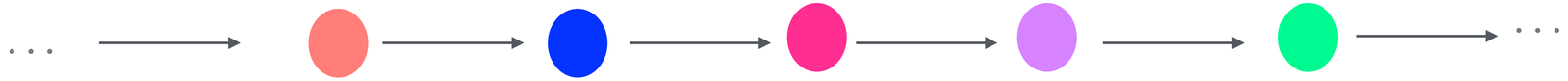
Sequences and memories:

- RNNs are designed for sequential data, where the order of inputs matters.
- At each time step t, the model updates a **hidden state** $h(t)$ **(memory)** that carries information from earlier inputs to later ones.
- This lets the network model temporal dependencies, so the prediction at time t can depend on both the current input $x(t)$and past context.





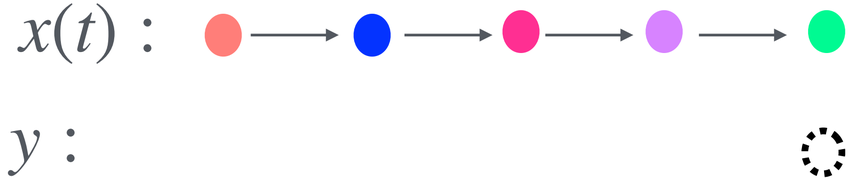
$x(t)$:



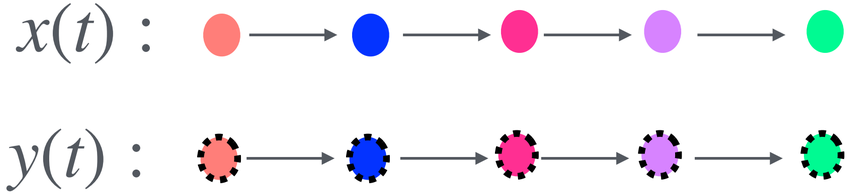
Sequences and memories:

The tasks could be different:

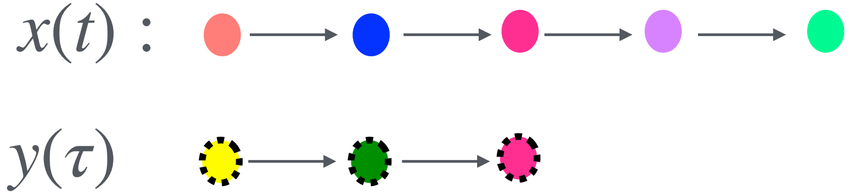
- **Sequence → one**: the model reads an entire sequence and predicts a single label or value.
  - *Examples: sentiment from a sentence, activity from a video, diagnosis from a time series.*



- **Sequence → sequence (same length):** the model outputs one prediction for each input time step.
  - *Examples: per-time-step signal classification, Predicting output signal from input signal in time.*



- **Sequence → sequence (different length):** the model encodes one sequence and generates another sequence.
  - *Examples: machine translation, speech-to-text, text summarization.*

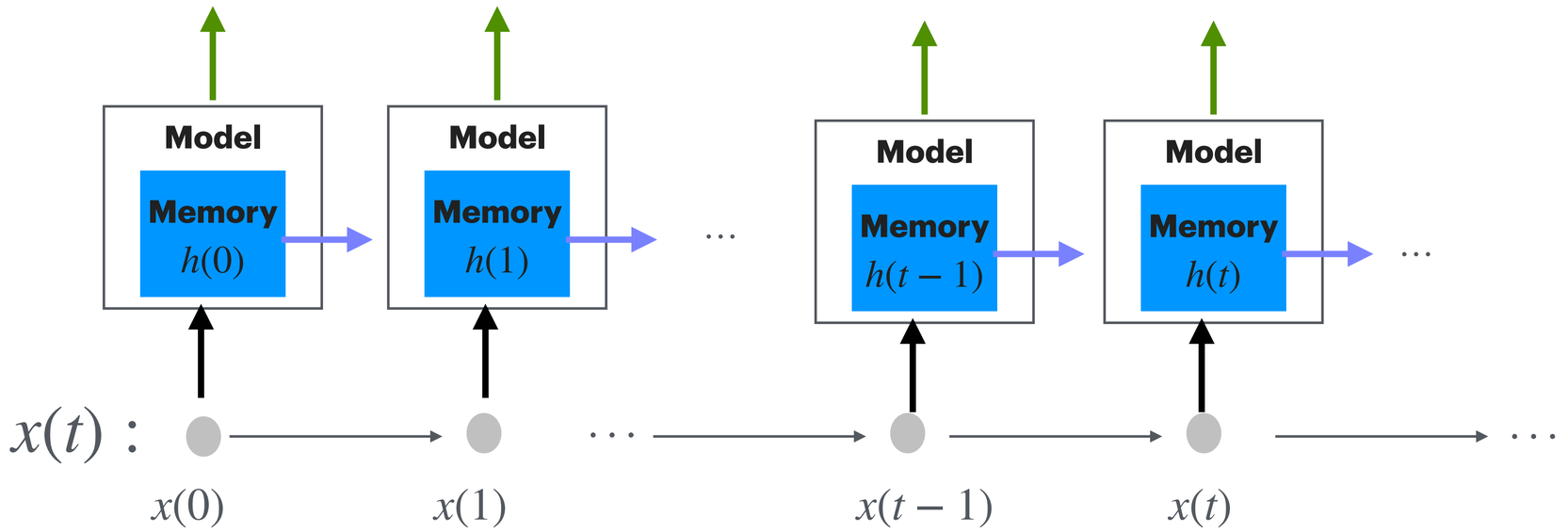


Sequences and memories:

Concept of memory

Memory: a compressed representation of the past carried forward through time.

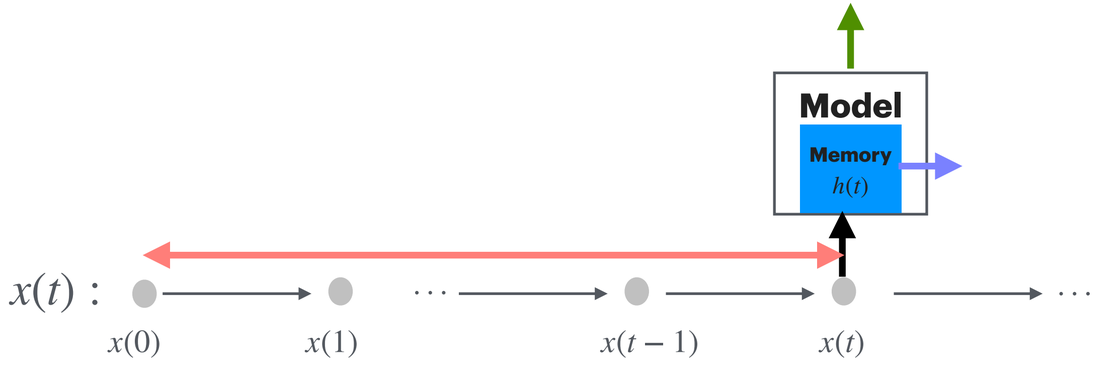
- In sequential problems, the current input is often not enough to make a good prediction.
- The model therefore maintains an internal state (memory) that summarizes relevant information from previous time steps.
- This state is updated as new inputs arrive, so the model can use the past to interpret the present and predict the future.
- This memory can be represented by the hidden state $h(t)$.

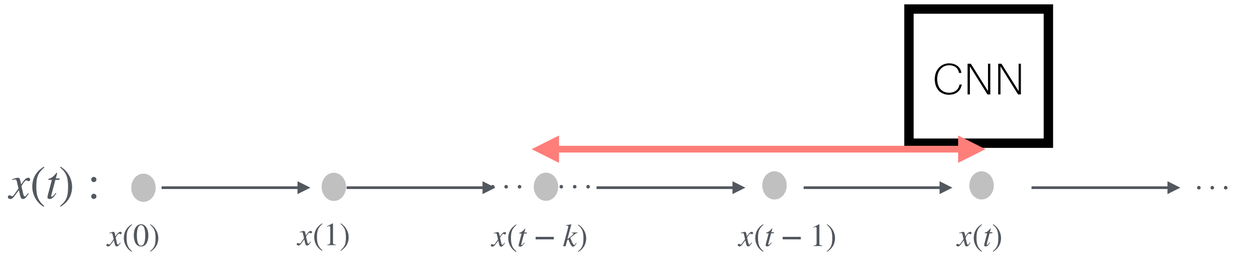


Sequences and memories:

Memory vs. CNN receptive field

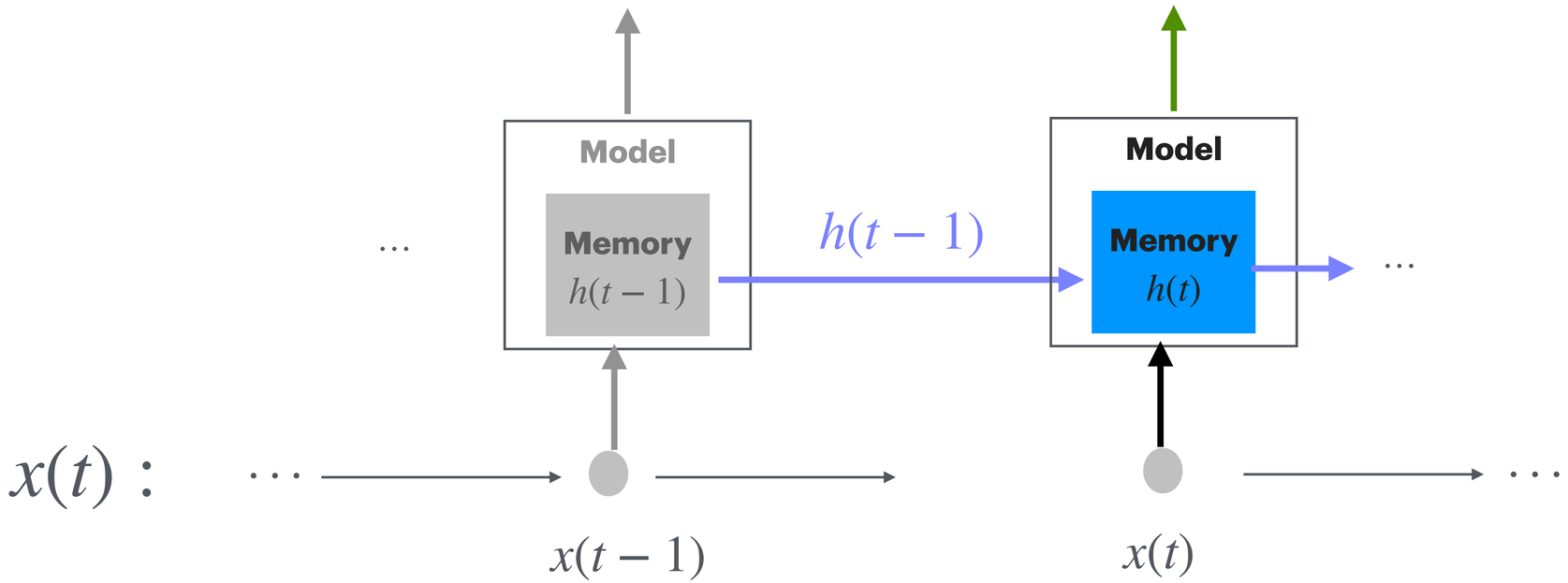
- In CNNs, context is gathered through a local receptive field.
- The amount of past information available depends on kernel size, depth, stride, and dilation.
- To capture long-range dependencies, CNNs must expand the receptive field across layers.
- In memory based modelling, past information is carried through an explicit hidden state $h(t)$ (memory) updated at each time step.





Sequences and memories:

How to define a model with memory? Intuitive building



- We want:

  1- At the current time t, we have take the memory from the previous step $h(t-1)$

  2-Also At the current time t, we be able to take the input sequence of this time point, $x(t)$

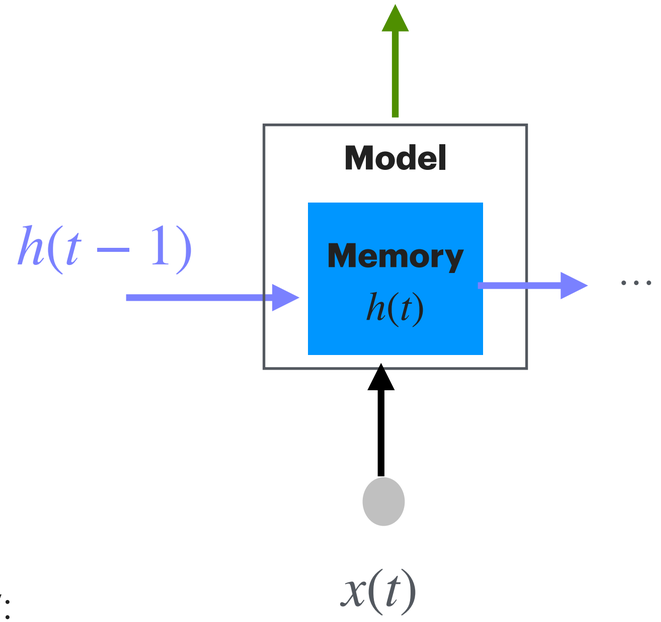
  3-The output of the model (if any) should come from the current memory $h(t)$. Which has both the memory of the past and the information of the current input store in it

Sequences and memories:

Building a model with memory

- At time t, the model receives the current input $x(t)$.
- It also keeps the memory from the previous step $h(t-1)$.
- These two are combined to produce an updated memory $h(t)$:

$$h_t = f(h_{t-1}, x_t)$$



- The new memory $h(t)$ summarizes both the past and the current input.

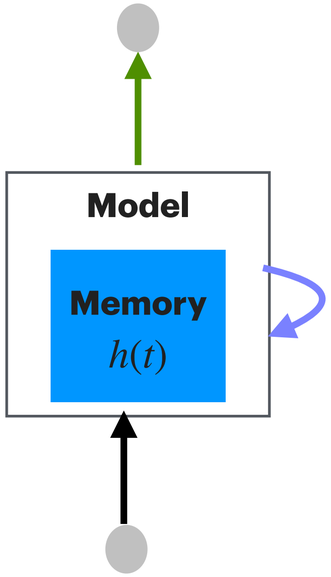
- The model output, if needed, can then be computed from this current memory:

$$y_t = g(h_t)$$

previous memory + current input → new memory → output

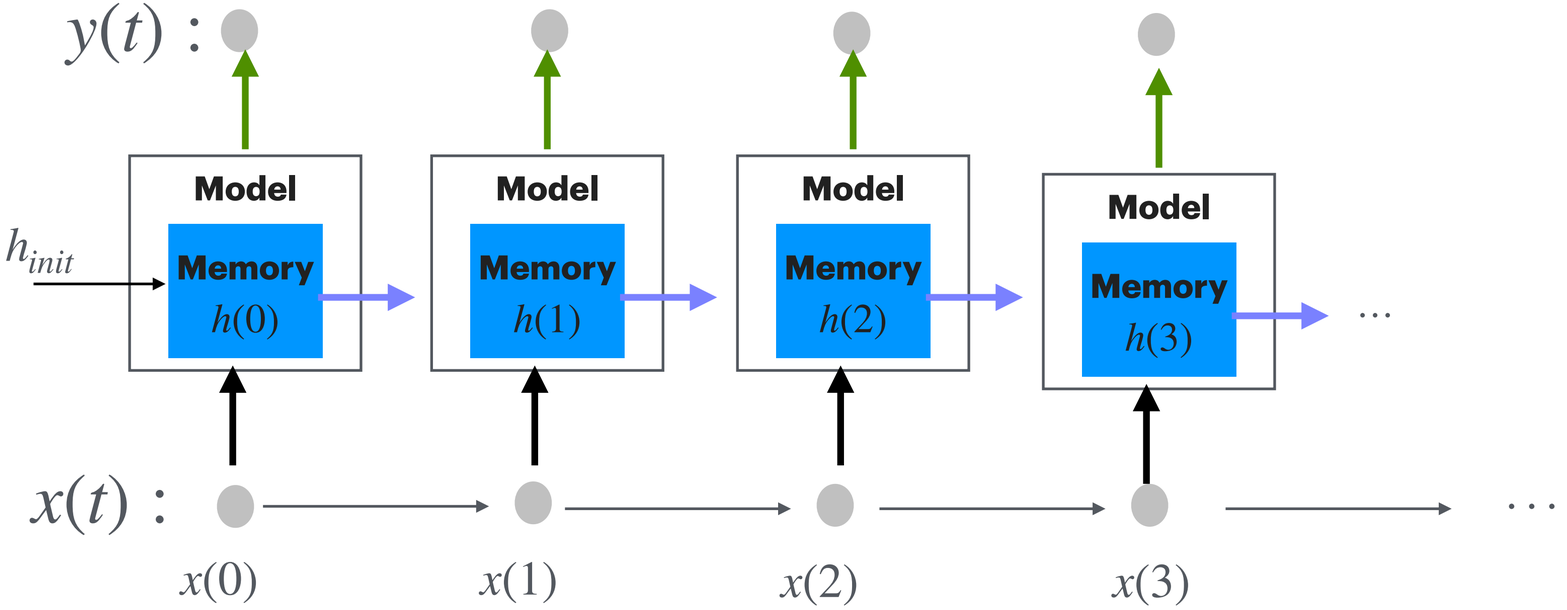
Sequences and memories:

Compact:



$$h_t = f(h_{t-1}, x_t)$$
$$y_t = g(h_t)$$

Unroll:



$$h(0) = f\big(h_{\text{init}}, x(0)\big), \qquad y(0) = g\big(h(0)\big)$$

$$h(1) = f\big(h(0), x(1)\big), \qquad y(1) = g\big(h(1)\big)$$

$$h(2) = f\big(h(1), x(2)\big), \qquad y(2) = g\big(h(2)\big)$$

$$h(3) = f\big(h(2), x(3)\big), \qquad y(3) = g\big(h(3)\big)$$

...

Sequences and memories:

Note the dependency at later times:

Note: For how many steps back should we unroll? It is a hyper parameter

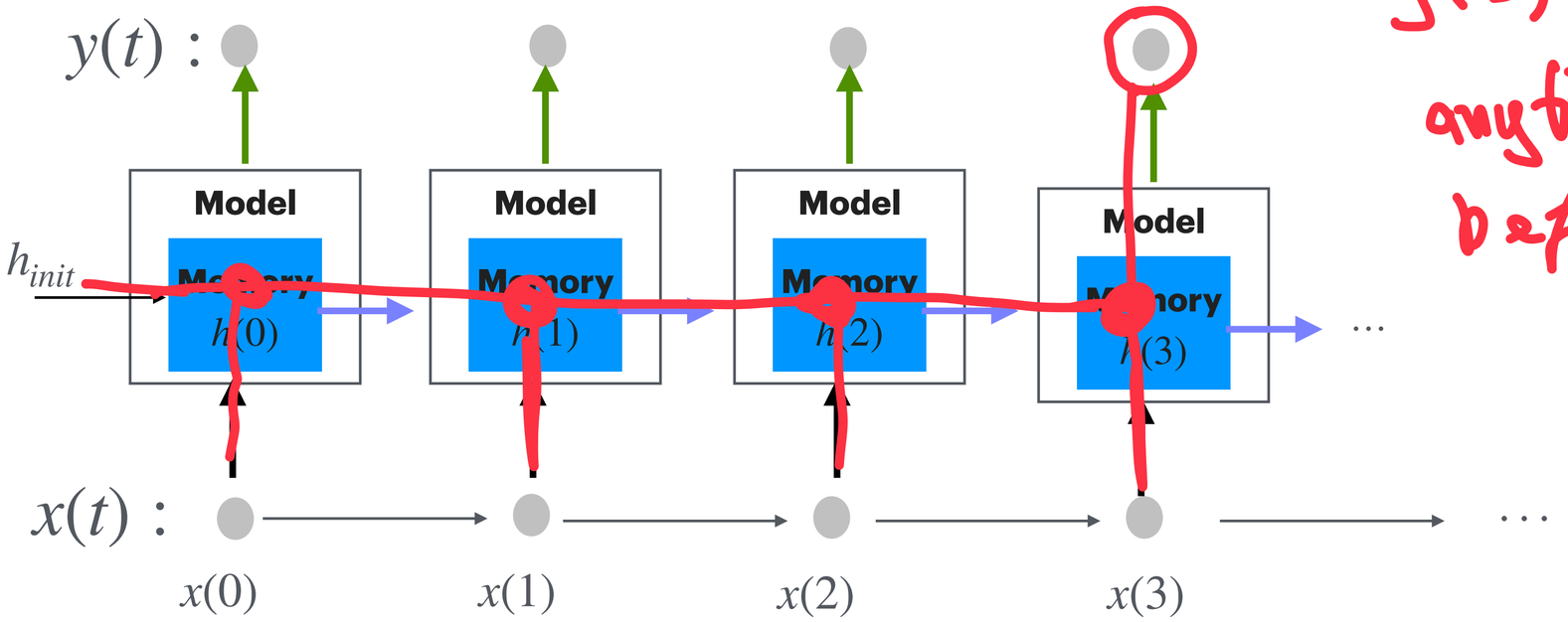
$$h(0) = f\big(h_{\text{init}}, x(0)\big), \qquad y(0) = g\big(h(0)\big)$$

$$h(1) = f\big(h(0), x(1)\big), \qquad y(1) = g\big(h(1)\big)$$

$$h(2) = f\big(h(1), x(2)\big), \qquad y(2) = g\big(h(2)\big)$$

$$h(3) = f\big(h(2), x(3)\big), \qquad y(3) = g\big(h(3)\big)$$

$$\dots$$



y(3) depends on anything came before!

A) Simplest choices for functions f and g: **Recurrent Neural Networks (RNN)**

Sequences and memories:

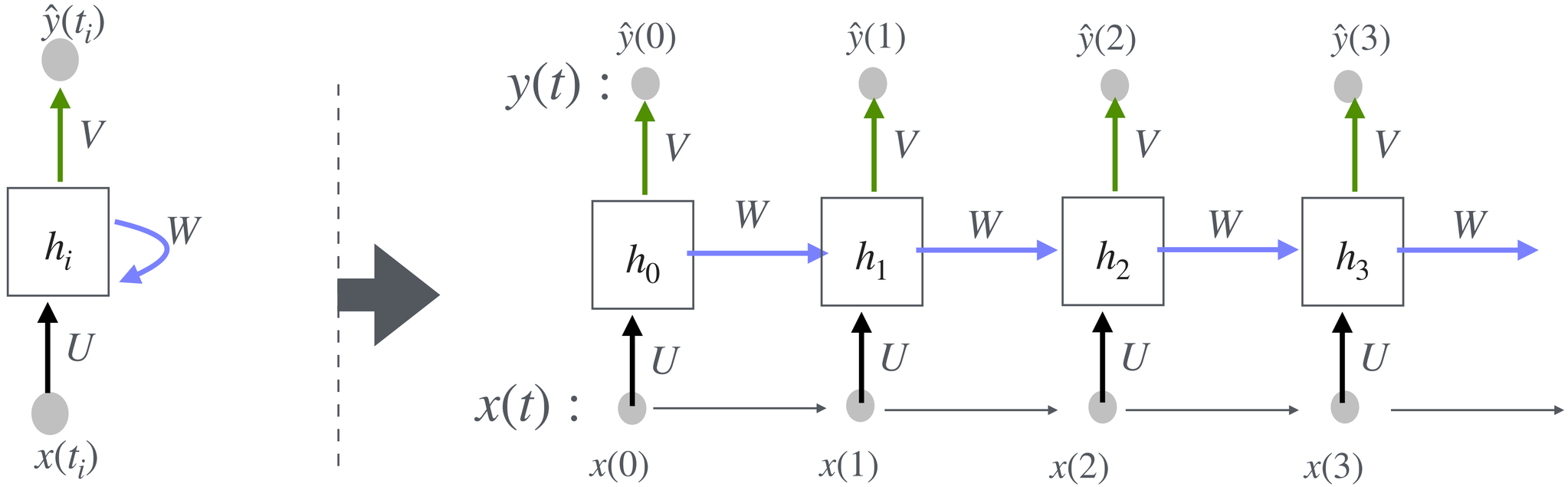
Simplest choices for functions f and g: **Recurrent Neural Networks (RNN)**

The simplest case for choosing f and g a linear transformations and integration, followed by a simple nonlinearity $\sigma(\cdot)$, for example tanh()... :

$$h_t = f(h_{t-1}, x(t)) = \sigma(Wh_{t-1} + Ux_t + b)$$

$$y_t = g(h_t) = \sigma(Vh_t + c)$$

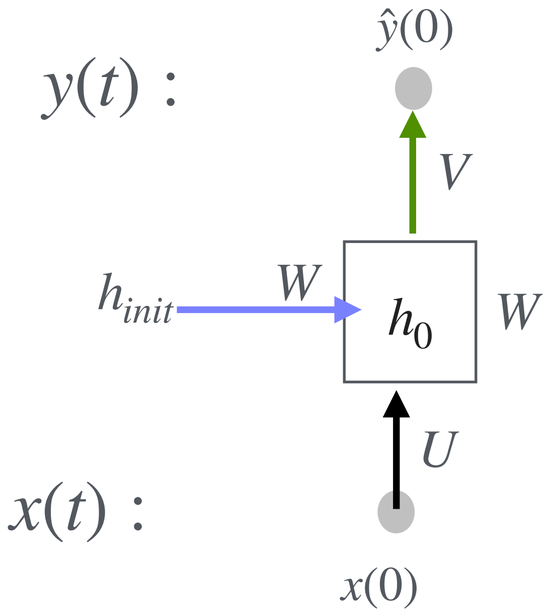
Note the parameter sharing: U, W, V, b, c are shared across time



Sequences and memories:

Practice 1: Calculate Back propagation on W for only for x(0). and look at the output at that state $\hat{y}(0)$:

$$\mathscr{L}_0 = \ell(\hat{y}_0, y_0)$$



$$\frac{\partial \mathscr{L}_0}{\partial W} = \frac{\partial \mathscr{L}_0}{\partial \hat{y}_0} \cdot \frac{\partial \hat{y}_0}{\partial h_0} \cdot \frac{\partial h_0}{\partial W}$$

$$\frac{\partial h_0}{\partial W} = \sigma'(Wh_{\text{init}} + Ux_0 + b)\, h_{\text{init}}$$

$$\Rightarrow \quad \frac{\partial \mathscr{L}_0}{\partial W} = \frac{\partial \mathscr{L}_0}{\partial \hat{y}_0} \cdot \frac{\partial \hat{y}_0}{\partial h_0} \cdot \sigma'(Wh_{\text{init}} + Ux_0 + b)\, h_{\text{init}}$$

$$h_0 = \sigma(Wh_{\text{init}} + Ux_0 + b)$$

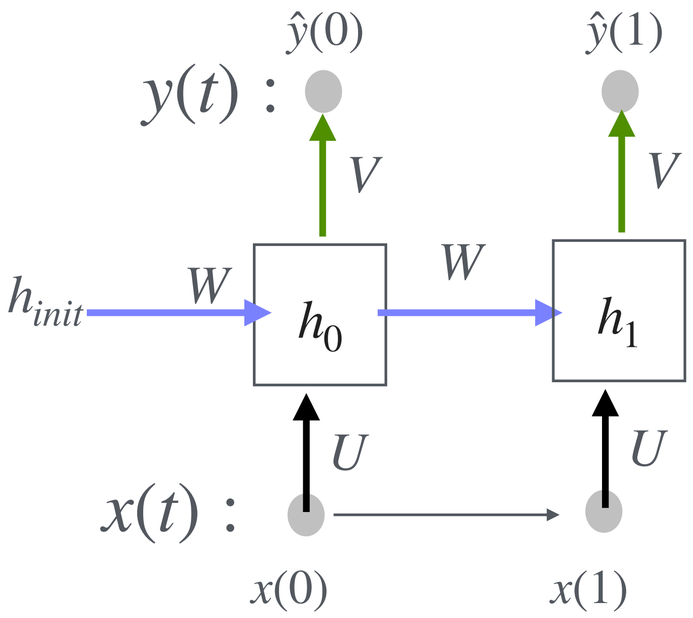
$$\hat{y}_0 = \sigma(Vh_0 + c)$$

In one step, W acts like an ordinary parameter inside one hidden update.

Sequences and memories:

Practice 2: Calculate Back propagation on W for only two steps x(1), and look at the output at that state $\hat{y}(1)$:

$$\mathscr{L}_1 = \ell(\hat{y}_1, y_1)$$



$$\frac{\partial \mathscr{L}_1}{\partial W} = \frac{\partial \mathscr{L}_1}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial h_1} \cdot \frac{\partial h_1}{\partial W}$$

because W affects h_1 in two ways: **1**-directly through the term Wh_0 **2**-indirectly through h_0

$$\frac{\partial h_1}{\partial W} = \sigma'(Wh_0 + Ux_1 + b)\frac{\partial (Wh_0 + Ux_1 + b)}{\partial W}$$

$$\frac{\partial (Wh_0 + Ux_1 + b)}{\partial W} = h_0 + W\frac{\partial h_0}{\partial W}$$

$$\frac{\partial h_0}{\partial W} = \sigma'(Wh_{\text{init}} + Ux_0 + b)\, h_{\text{init}}$$

$$\frac{\partial \mathscr{L}_1}{\partial W} = \frac{\partial \mathscr{L}_1}{\partial \hat{y}_1} \cdot V \cdot \sigma'(Wh_0 + Ux_1 + b)\left[h_0 + W\,\sigma'(Wh_{\text{init}} + Ux_0 + b)\, h_{\text{init}}\right]$$

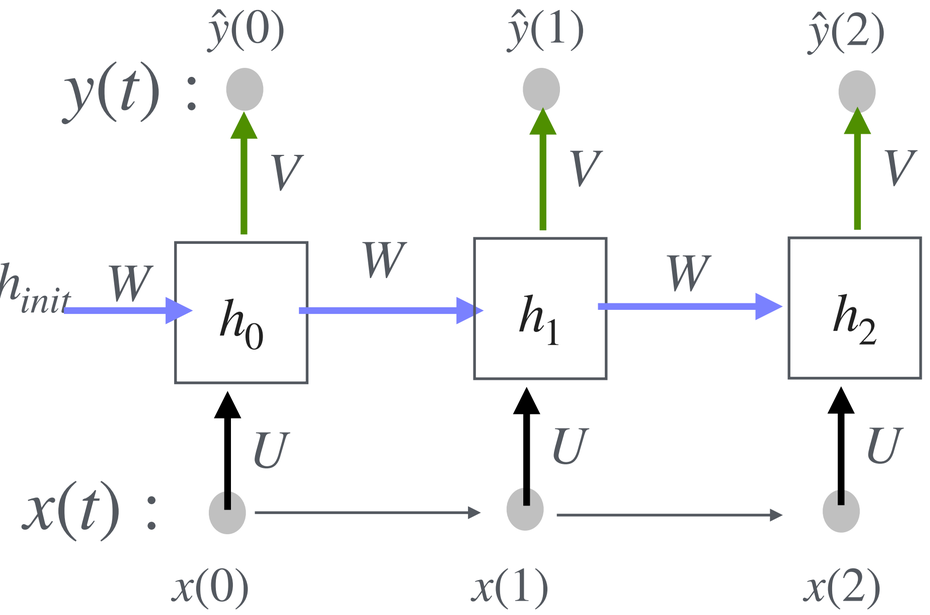
$$h_0 = \sigma(Wh_{\text{init}} + Ux_0 + b)$$
$$h_1 = \sigma(Wh_0 + Ux_1 + b)$$
$$\hat{y}_1 = Vh_1 + c$$

Sequences and memories:

Practice 2: Calculate Back propagation on W for only two steps $x(2)$, and look at the output at that state $\hat{y}(2)$:

$$\mathscr{L}_2 = \ell(\hat{y}_2, y_2)$$

$$\frac{\partial \mathscr{L}_2}{\partial W} = \frac{\partial \mathscr{L}_2}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial h_2} \cdot \frac{\partial h_2}{\partial W}$$



$$h_0 = \sigma(Wh_{\text{init}} + Ux_0 + b)$$
$$h_1 = \sigma(Wh_0 + Ux_1 + b)$$
$$h_2 = \sigma(Wh_1 + Ux_2 + b)$$
$$\hat{y}_2 = Vh_2 + c$$

$$\frac{\partial h_2}{\partial W} = \sigma'(Wh_1 + Ux_2 + b)\left(h_1 + W\frac{\partial h_1}{\partial W}\right)$$

$$\frac{\partial h_1}{\partial W} = \sigma'(Wh_0 + Ux_1 + b)\left(h_0 + W\frac{\partial h_0}{\partial W}\right)$$

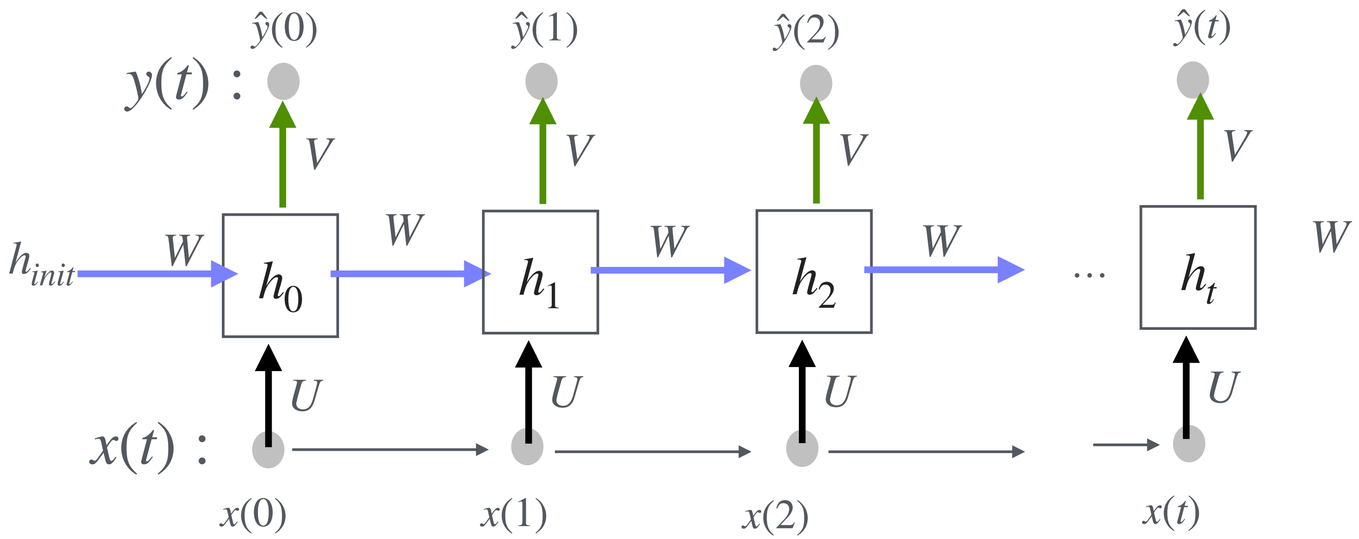
$$\frac{\partial h_0}{\partial W} = \sigma'(Wh_{init} + Ux_0 + b)\, h_{init}$$

$$\frac{\partial \mathscr{L}_2}{\partial W} = \frac{\partial \mathscr{L}_2}{\partial \hat{y}_2} V\sigma'(Wh_1 + Ux_2 + b)\Big[h_1 + W\sigma'(Wh_0 + Ux_1 + b)\big(h_0 + W\sigma'(Wh_{\text{init}} + Ux_0 + b)\, h_{\text{init}}\big)\Big]$$

current step + W(1 step back) + W^2(2 steps back) + ...

Sequences and memories:

Practice 3: Calculate Back propagation on W for only many steps $x(t)$, and look at the output at that state $\hat{y}(t)$:



$\frac{\partial \mathcal{L}_t}{\partial W}$ = contribution from step t + contribution from step $t-1$ + $\cdots$ + contribution from earlier steps

Contribution from k steps back $\propto \left(W\sigma'(\cdot)\right)^k$

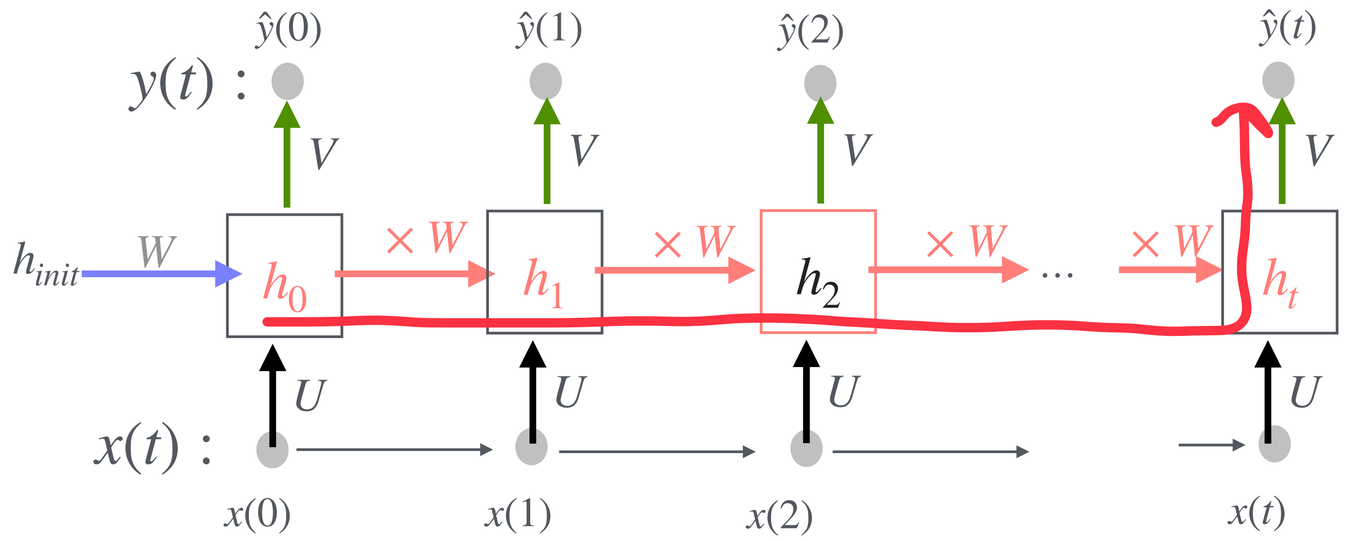
Very unstable to train!

$|W| < 1 \Rightarrow$ vanishing gradients, goes to zero

$|W| > 1 \Rightarrow$ exploding gradients, goes to inf

Sequences and memories:

What's the root cause? Let's look at the diagnosis of our problem that causes keeping track of long term memory hard:



Every time memory passes through a step, it gets multiplied by W again

NOTE: RNNs are still work for simple problems, so keep them in mind!

Sequences and memories:

Solution? Instead of every time that we pass the memory, multiply it with a weight, we can pass some part of it intact!

Two popular recurrent architectures that do it are:

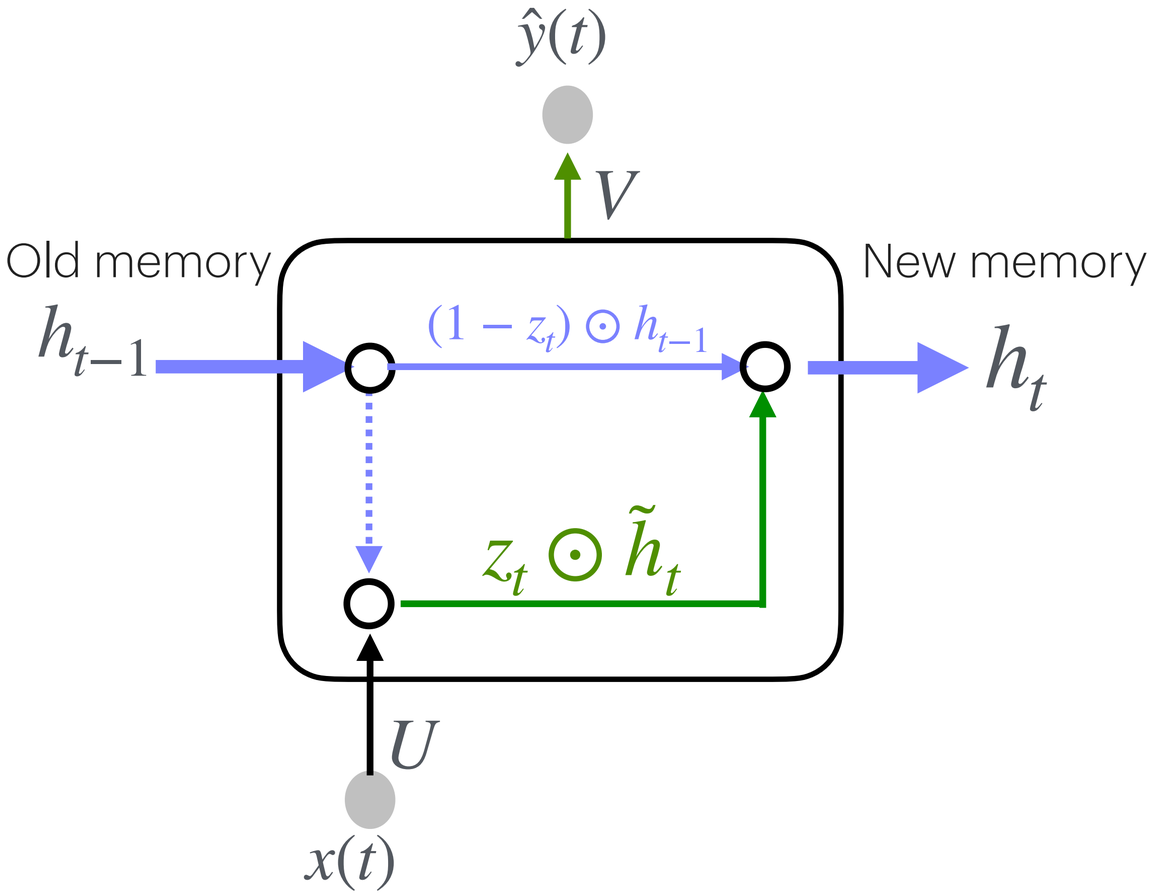
B) Gated Recurrent units (GRUs)
C) Long-Short term memory (LSTMs)

B) Choices for functions f and g: **Gated Recurrent units (GRUs)**

Sequences and memories:

B1) Minimal Gated Recurrent units (min-GRUs)

- Instead of fully rewriting memory at each step, a min-GRU forms the new memory using two paths:
  - A carry path that passes part of the old memory forward
  - An update path that builds a candidate new memory from the current input
- The new memory is a gated (via z_t) combination of these two paths.



Learnable gater value:

$$z_t = \sigma(W_z h_{t-1} + U_z x_t)$$

Learnable canditate new memory:

$$\tilde{h}_t = \tanh(W h_{t-1} + U x_t + b)$$

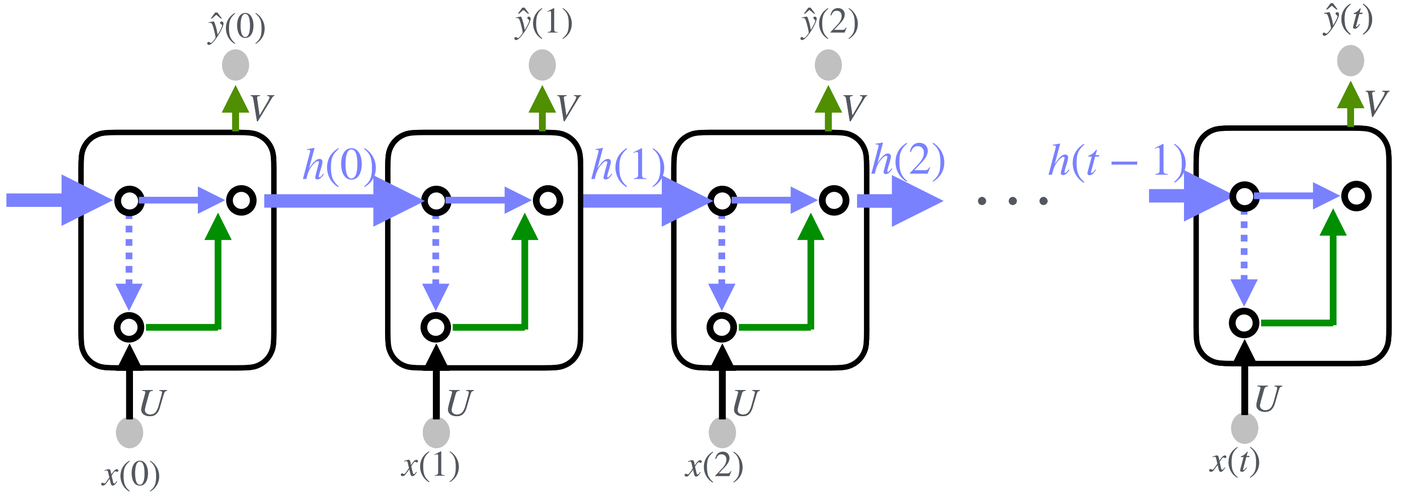
Learnable new memory:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

$$y_t = \sigma(V h_t + c)$$

Sequences and memories:

B1) How min-GRUs improve gradient flow?

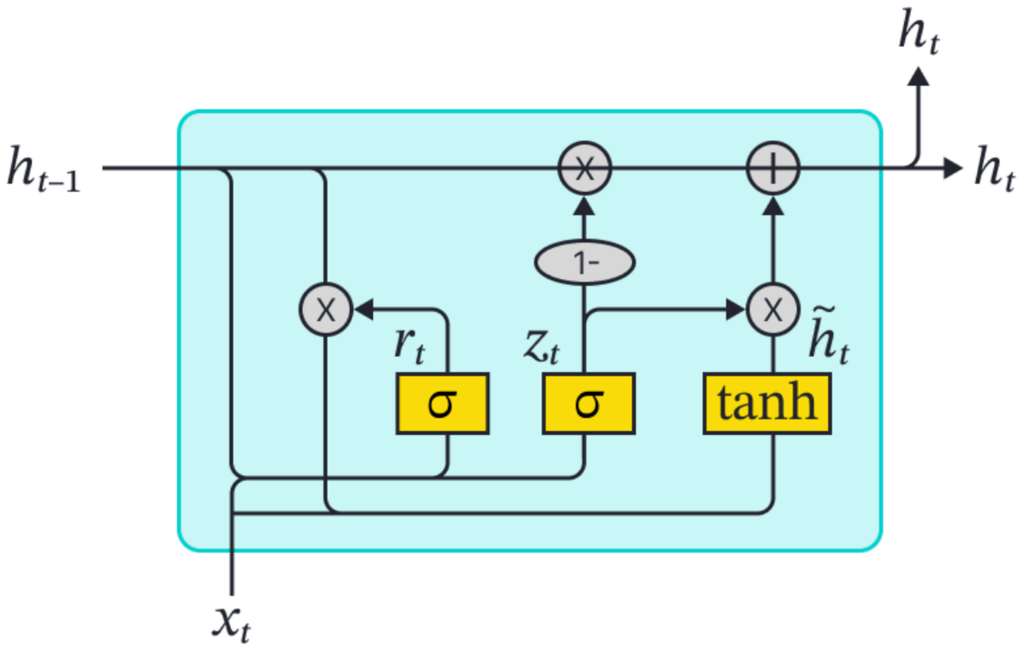


- Note that $\frac{\partial \mathscr{L}_t}{\partial h_0} = \frac{\partial \mathscr{L}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_0}$. Meaning that the effect of early memories on final result could be captured by looking at $\frac{\partial h_t}{\partial h_0}$
- In a vanilla RNN, long-range gradients must pass through repeated recurrent transformations, roughly behaving like W^t, $\frac{\partial h_t}{\partial h_0} \approx W^t$
- In a min-GRU, part of the old memory passes directly through the carry path. Backward signals can also use this path, where they are modulated by $(1 - z_t)$ instead of relying mainly on repeated powers of W
- . Because Z_t is learned at each step independently, the network can choose to preserve memory when it matters!

$$\frac{\partial h_t}{\partial h_0} \approx \prod_{k=1}^{t} (1 - z_k)$$

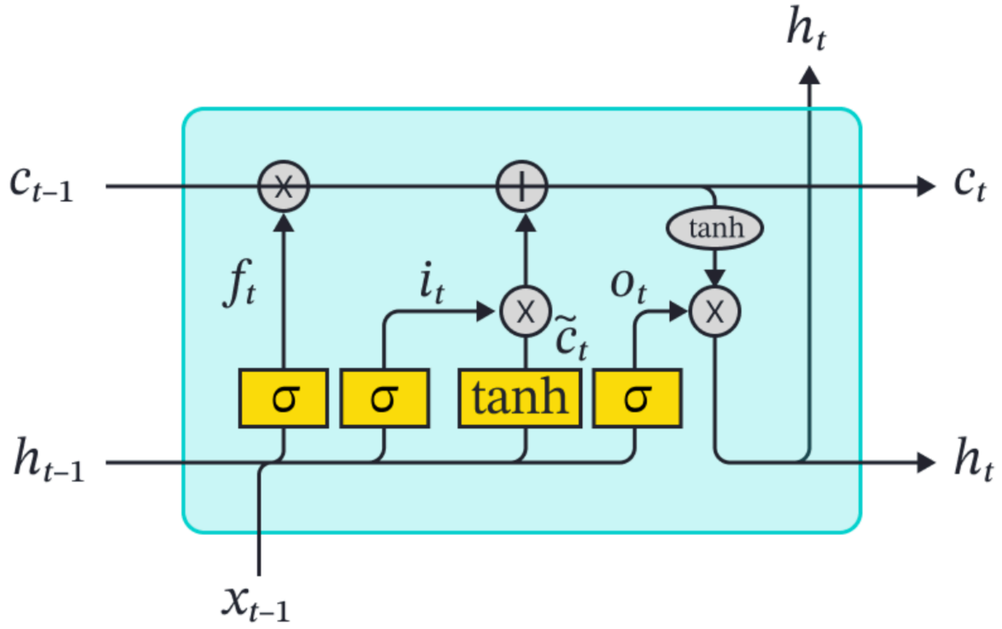
Popular sequence models (pre-transformer era!):

B) **GRUs**



Gated Recurrent Unit (GRU)

C) **LSTMS**



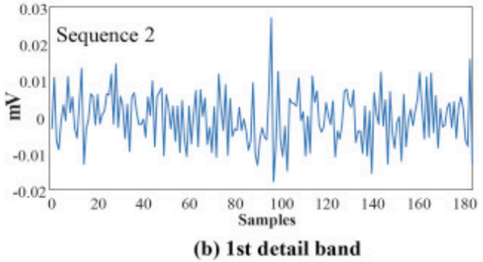
Long Short Term Memory (LSTM)

Sequences and memories:

Example usage in Biology (RNN/LSTM/GRU)

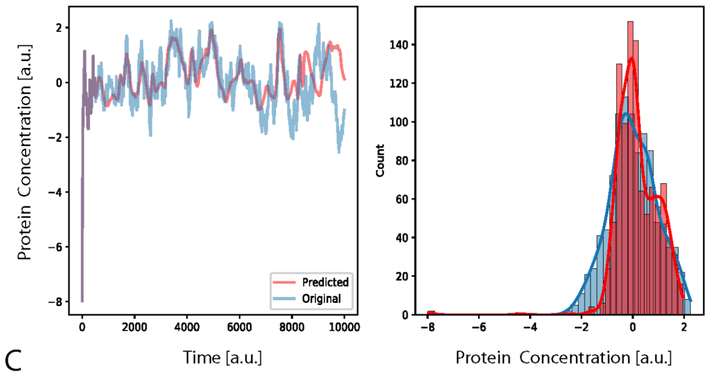
In clinical diagnostics, RNNs are used to process physiological time-series data where the timing between peaks (like the R-peak in a heartbeat) is critical for diagnosis

Yildirim, Özal. "A novel wavelet sequence based on deep bidirectional LSTM network model for ECG signal classification." *Computers in biology and medicine* 96 (2018): 189-202.



In clinical diagnostics, RNNs are used to proGenomics researchers use RNNs to understand how genes "turn on and off" over time in response to stimuli like drugs or environmental cess physiological time-series data where the timing between peaks (like the R-peak in a heartbeat) is critical for diagnosis

Monti, Michele, et al. "Prediction of time series gene expression and structural analysis of gene regulatory networks using recurrent neural networks." *Entropy* 24.2 (2022): 141.



....

Recipe Card: **Recurrent Models**

Type of Learning: Supervised
Data set: Sequence

Data

1- Load libraries: Basics + PyTorch

2-Load your sequence dataset **D.** Sequence data usually comes in two forms:

i)Many sequences:

$D = \{(x_1, y_1), \ldots, (x_k, y_k)\}, \qquad x_i \in \mathbb{R}^{T_i \times F}, \quad T_i$: seq length, F: num of features

ii) One long sequence: Then we usually split it into many shorter windows for training, again:

$x = [x_1, x_2, \ldots, x_T], y = [y_1, y_2, \ldots, y_T] \rightarrow split \rightarrow D = \{(x_1, y_1), \ldots, (x_k, y_k)\}$

3- Now set up the DataLoader for training/validation/testing.

a) If all sequences have the same length: stack them directly into batches

b) If sequences have different lengths: pad them inside a custom collate_fn

-A common batch shape is: $x \in [B, T, F]$

NN model

4-Build the recurrent Model:

Note: Recurrent models have two main parts: recurrent backbone + task head.

A)Define the Recurrent backbone (choose nn.RNN(...) nn.GRU(...) nn.LSTM(...))

-Input:$x \in [B, T, F]$

-output: hidden states for all time steps, shape: $h_T \in [B, H]$, (H is the dimension of hidden)

For LSTM also final cell state: $c_T \in [B, H]$-B) define task head as a module [Task

B) Define Task head:

i) Sequence-to-one: $h_T \rightarrow$ [Linear layer] $\rightarrow out, \hat{y}$

ii) Sequence-to-sequence: $h_t \rightarrow$ [Linear layer] $\rightarrow out, \hat{y}_t$

Eventually your network should look like:

$$x \rightarrow \text{[RNN / GRU / LSTM]} \rightarrow \begin{cases} h_T \rightarrow \text{Linear} \rightarrow \hat{y} & amp; \text{sequence-to-one} \\ output \rightarrow \text{Linear} \rightarrow \hat{y}_t & amp; \text{sequence-to-sequence} \end{cases}$$

Training

5-Define a proper loss function based on Y

a-If Y is continuous, then use MSELoss or L1Loss

b-If Y is categorical, use CrossEntropyLoss. note that we feed raw logits directly to Loss.

c-For sequence-to-sequence tasks:

1) compute loss over all relevant time steps

2) ignore padded positions using masking if needed

-Add regularization to prevent overfitting

-The loss will be the sum of all relevant losses

6-Define the train and test functions:

-Train/Validate/test functions are standard

-For each batch: move data to device, run forward pass, compute loss, backpropagate, optimizer step.

-For variable-length sequences: use padding + masking or pack_padded_sequence(...)

7-Define training and run the training:

-Same as standard.

-For each epoch: run train() on training data, then run test() on training/validation data

-At the end: load the best saved model and plot training/validation loss curves if needed

Sequence-to-sequence with encoder-decoder recurrent models

Up to now, we used recurrent models to map a sequence into one final output. But sometimes the output is itself another sequence. A common solution is encoder-decoder recurrence: the encoder compresses the input into a context, and the decoder unfolds that context into an output sequence.

1) How it works? At a high level it looks like this (remember the concept of Autoencoders...):

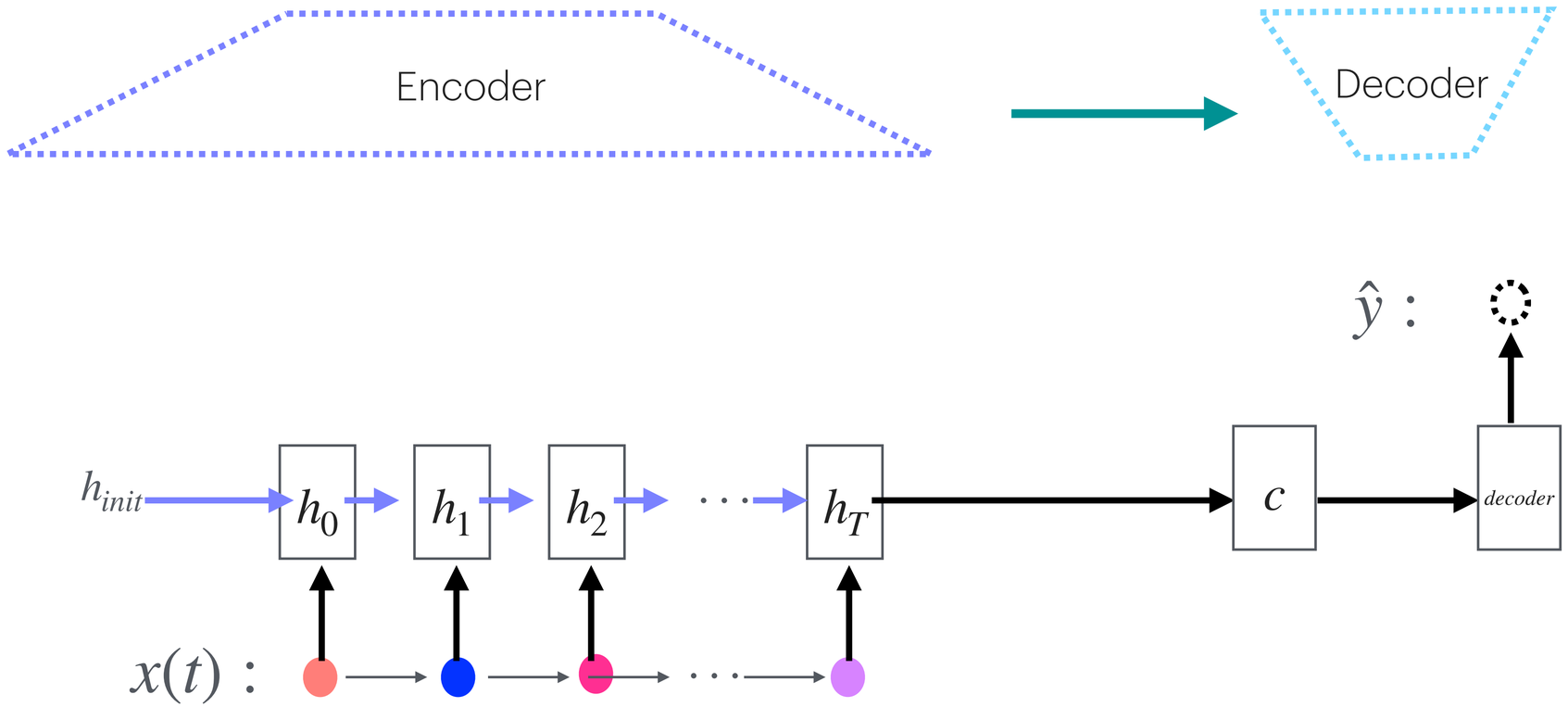
2) The decoder generates the output sequence (if it is a sequence) **autoregressively**. meaning it produces one output at a time and then uses that information to generate the next one.

sequence encoder + prediction head

- **Sequence → one**: the model reads an entire sequence and predicts a single label or value.
  - *Examples: sentiment from a sentence, activity from a video, diagnosis from a time series.*

Usually we have something like and encoder, and a decoder:

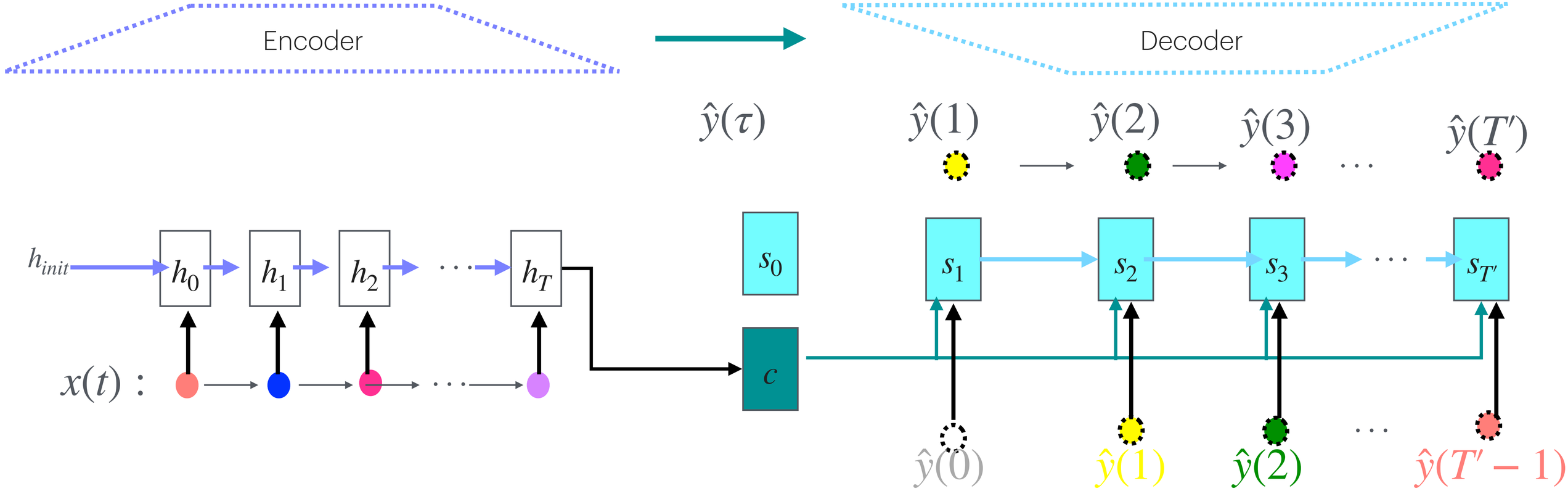
c : the context vector. It summarizes the entire input sequence into one vector, usually $c = h_T$



Sequence-to-sequence with encoder-decoder recurrent models

- **Sequence → sequence (same or different length):** the model encodes one sequence and generates another sequence.

  We have:
  - An encoder that recurrently reads the input sequence.
  - A recurrent decoder that auto regressively generates outputs. At each decoder step, the model uses the previous decoder state (S_i), the context (c).
    - c : the context vector. It summarizes the entire input sequence into one vector, usually $c = h_T$
    - s_0 : initial decoder state, initial decoder state, often initialized from c



Recipe Card: **Autoregressive Recurrent Models**

(seq2seq)

Type of Learning: Supervised + Autoregressive
Data set: Input sequence **+** output sequence

Data

1- Load libraries: Basics + PyTorch

2-Load your sequence dataset **D:**

$$D = \{(x_1, y_1), \ldots, (x_k, y_k)\}$$

Where:

$x_i \in \mathbb{R}^{T_x \times F_x}$:input sequence

$y_i \in \mathbb{R}^{T_y \times F_y}$:target sequence

3- Now set up the DataLoader for training/validation/testing.

a) If all sequences have the same length: stack them directly into batches

b) If sequences have different lengths: pad them inside a custom collate_fn

-A common batch shape is: $x \in [B, T_x, F_x], \qquad y \in [B, T_y, F_y]$

NN model

4-Build the autoregressive recurrent model (3 parts, A, B, and C):

A)Encoder (choose nn.RNN(...) nn.GRU(...) nn.LSTM(...))

$$x \rightarrow [\text{RNN / GRU / LSTM encoder}] \rightarrow c$$

-Input:$x \in [B, T, F]$. output: is usually : $c = h_T \in [B, H]$, (H is the dimension of hidden)

-Dont confuse c (context) with LSTMs cell states, c_t...

B) Decoder. Generates output one step at a time:

$$s_\tau = f_{dec}(s_{\tau-1}, \hat{y}_{\tau-1}, c)$$

C) Output head:

$$s_\tau \rightarrow [\text{Linear layer}] \rightarrow out, \hat{y}_\tau$$

So overa:

$$x_1, x_2, \ldots, x_T \rightarrow \text{Encoder} \rightarrow c \rightarrow \text{Decoder} \rightarrow \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{T_y}$$

Training

5-Define a proper loss function based on Y

a-If Y is continuous, then use MSELoss or L1Loss

b-If Y is categorical, use CrossEntropyLoss. note that we feed raw logits directly to Loss.

c-For autoregressive part:

1) compute loss over all relevant *decoder* time steps
2) ignore padded positions using masking if needed

-Add regularization to prevent overfitting

-The loss will be the sum of all relevant losses

6-Define the train and test functions:

-Encoder reads the input sequence and outputs context c

-Decoder rolls forward one step at a time, generates output

-**Training**: often use teacher forcing: feed the true previous target $y(\tau - 1)$ into the decoder

-**Inference**: use the model's own previous prediction, $\hat{y}(\tau - 1)$

-the rest as normal recurrent networks..

7-Define training and run the training:

-Same as standard recurrent network card

Read more:

J.L. Elman, Finding structure in time., Cognitive Science 14 (1990) 179–211.

S. Hochreiter, J. Schmidhuber, Long short-term memory., Neural Computation 9 (1997) 1735–1780.

https://visionbook.mit.edu/recurrent_neural_nets.html