# Recurrent Neural Networks

CSE 849 Deep Learning
Spring 2025

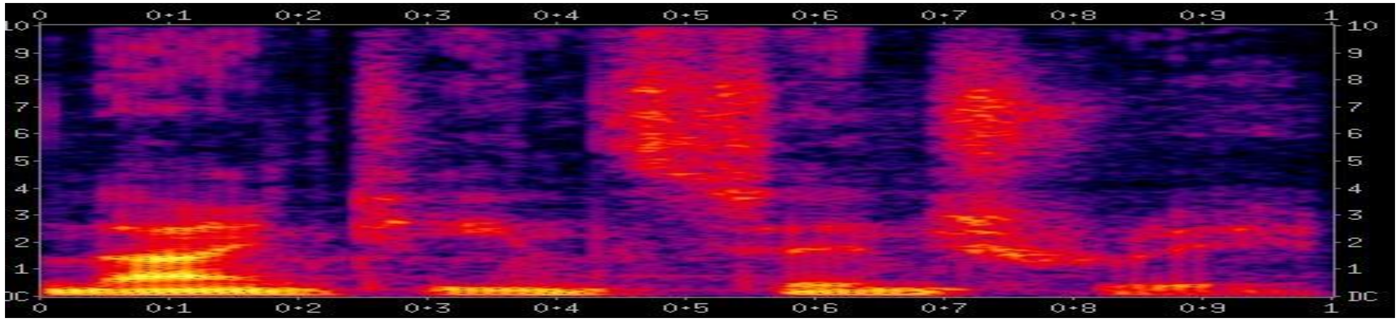
Zijun Cui

Partial slides are adapted from CMU 11-785 Introduction to Deep Learning by Bhiksha Raj

# Modelling Series

- In many situations one must consider a *series* of inputs to produce an output
  - Outputs too may be a series

# What did I say?




- Speech Recognition
  - Analyze a series of spectral vectors, determine what was said
- Note: Inputs are sequences of vectors.  Output is a classification result

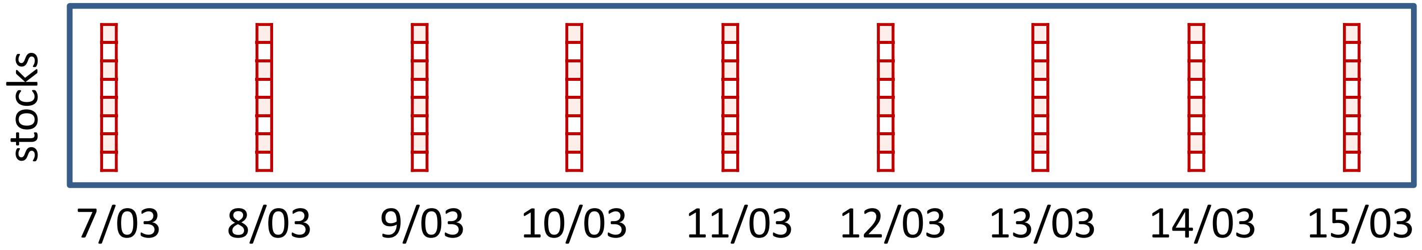
# What is he talking about?

The Steelers, meanwhile, continue to struggle to make stops on defense. They've allowed, on average, 30 points a game, and have shown no signs of improving anytime soon.

"Football" or "basketball"?

- Text analysis
  - E.g. analyze document, identify topic
    - Input series of words, output classification output
  - E.g. read English, output French
    - Input series of words, output series of words

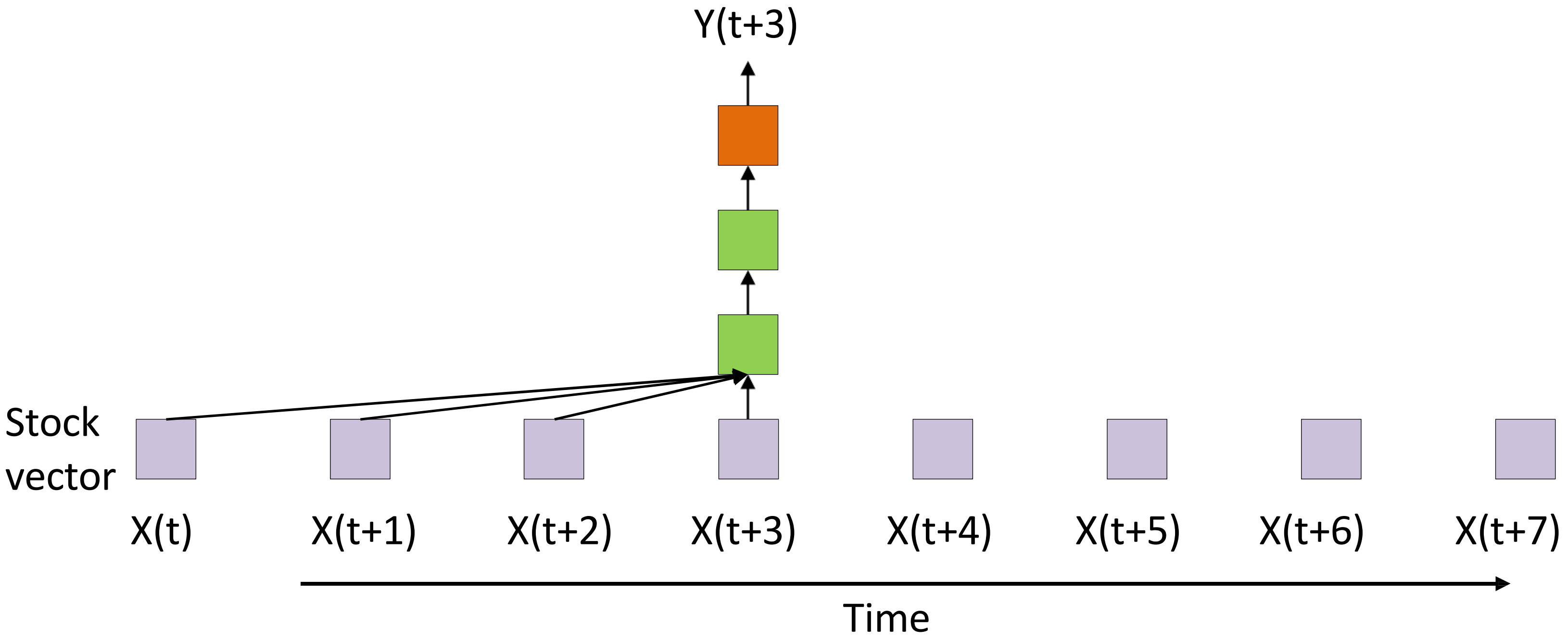
# Should I invest..




- Note: Inputs are sequences of vectors.   Output may be scalar or vector
  - Should I invest, vs. should I not invest in X?
  - Decision must be taken considering how things have fared over time

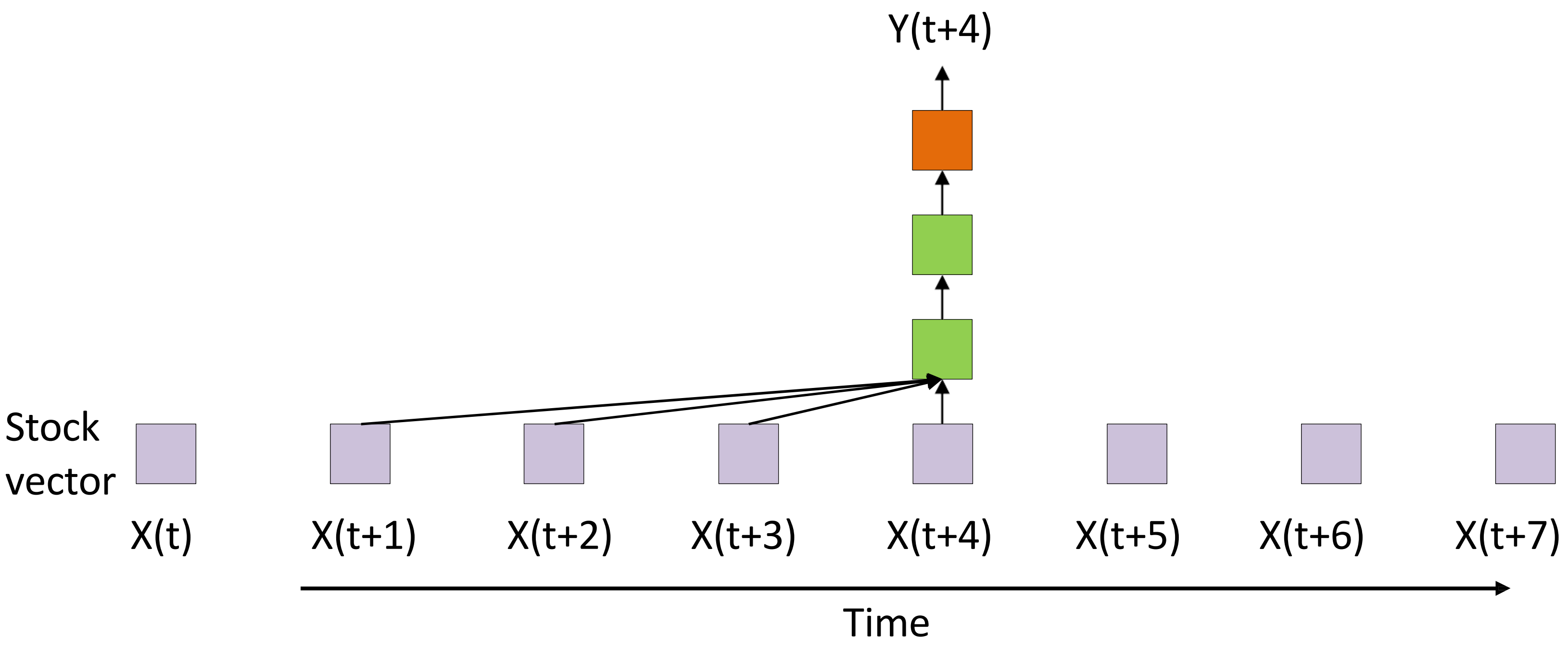
- Must consider the series of stock values in the past several days to decide if it is wise to invest today

# The stock predictor network



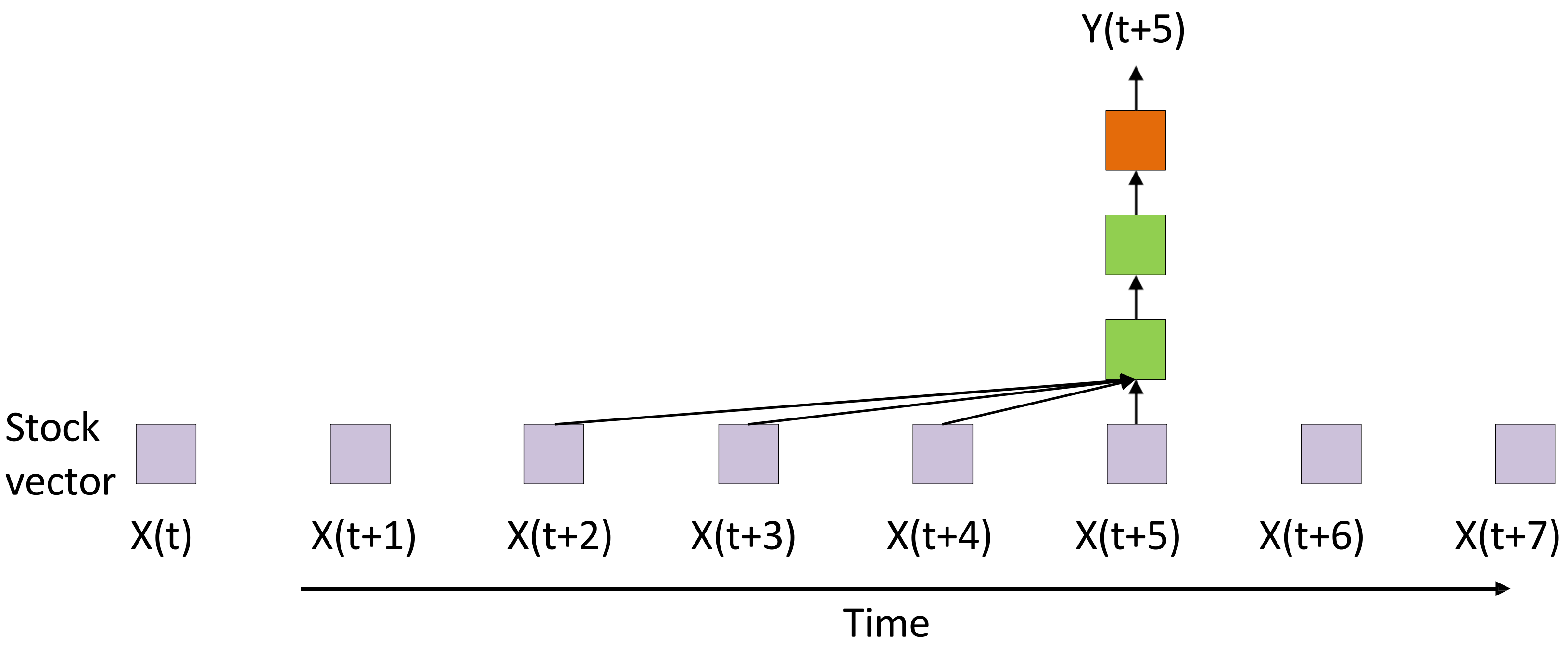

- The sliding predictor
  - Look at the last few days
  - This is just a convolutional neural net applied to series data
    - Also called a *Time-Delay neural network*

# The stock predictor network



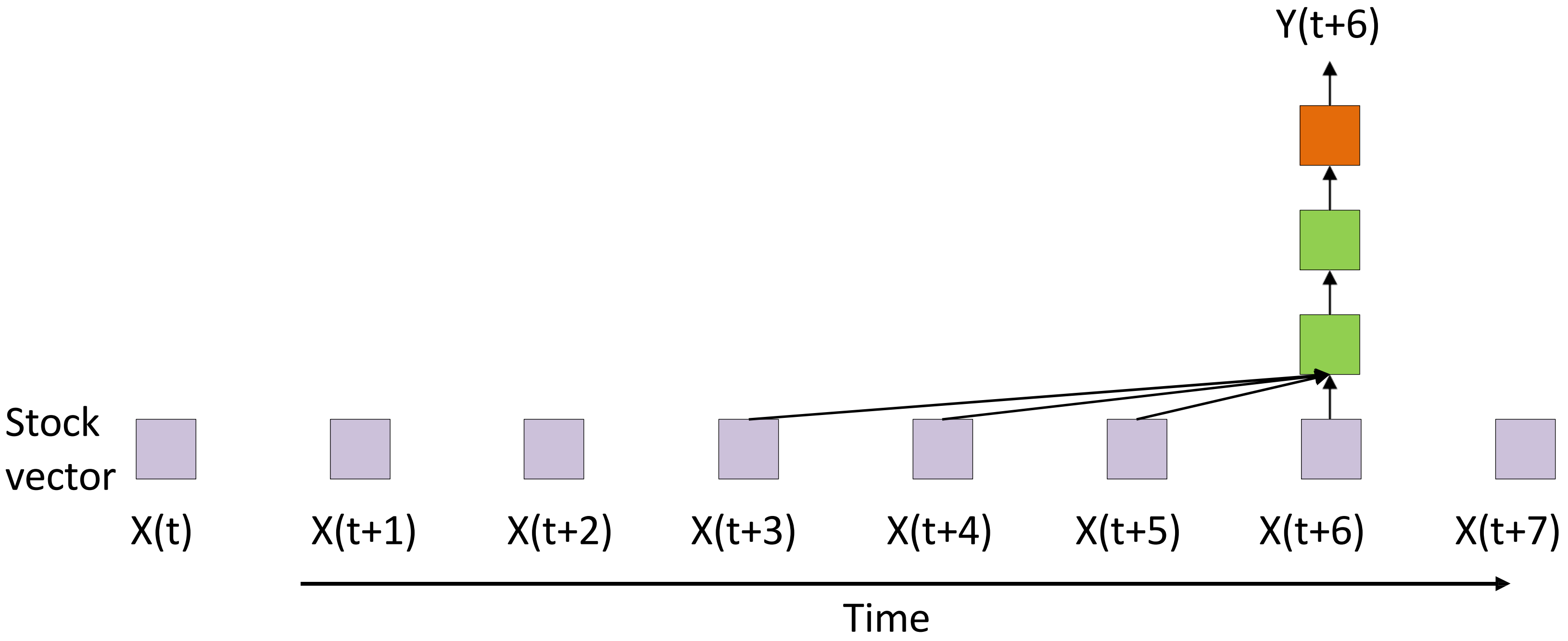

- The sliding predictor
  - Look at the last few days
  - This is just a convolutional neural net applied to series data
    - Also called a *Time-Delay neural network*

# The stock predictor network



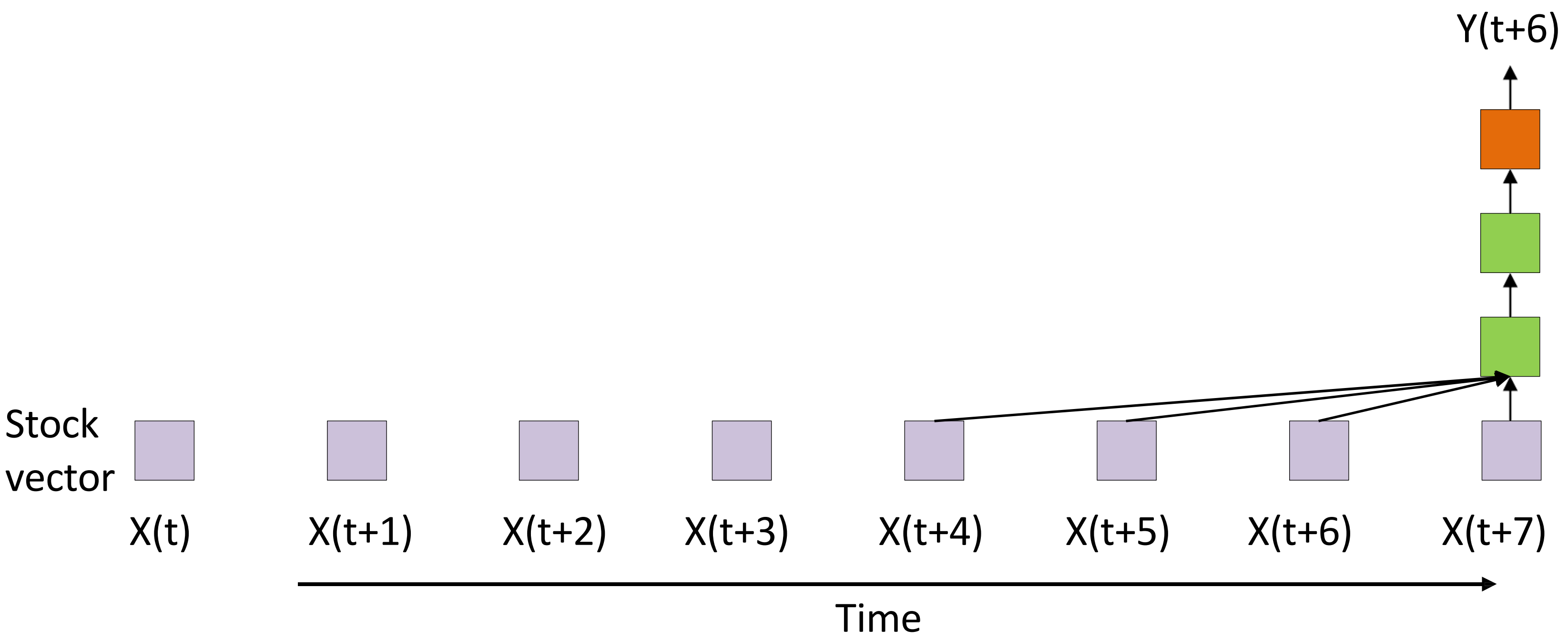

- The sliding predictor
  - Look at the last few days
  - This is just a convolutional neural net applied to series data
    - Also called a *Time-Delay neural network*

# The stock predictor network




- The sliding predictor
  - Look at the last few days
  - This is just a convolutional neural net applied to series data
    - Also called a *Time-Delay neural network*

# The stock predictor network



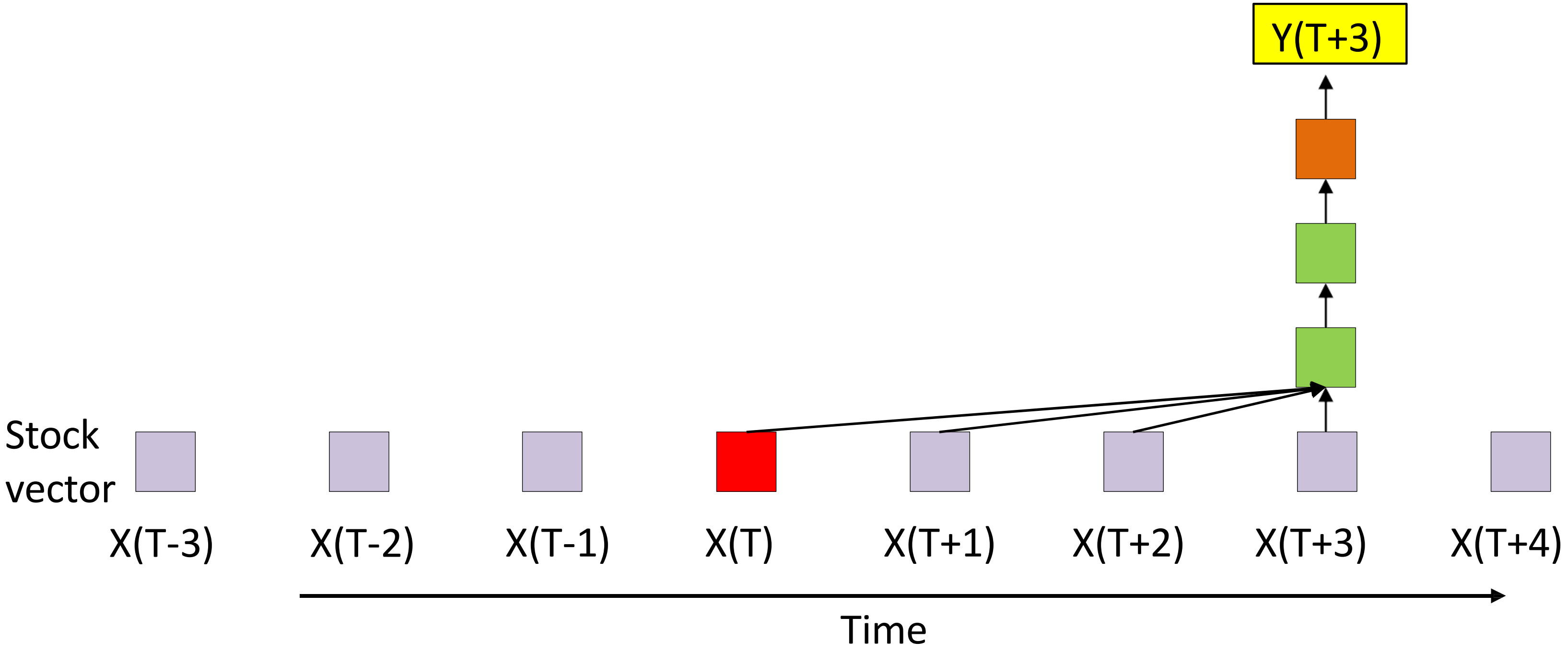

- The sliding predictor
  - Look at the last few days
  - This is just a convolutional neural net applied to series data
    - Also called a *Time-Delay neural network*

# Finite-response model

- This is a *finite response* system
  - Something that happens *today* only affects the output of the system for $N$ days into the future
    - $N$ is the *width* of the system

$$Y_t = f(X_t, X_{t-1}, \dots, X_{t-N})$$

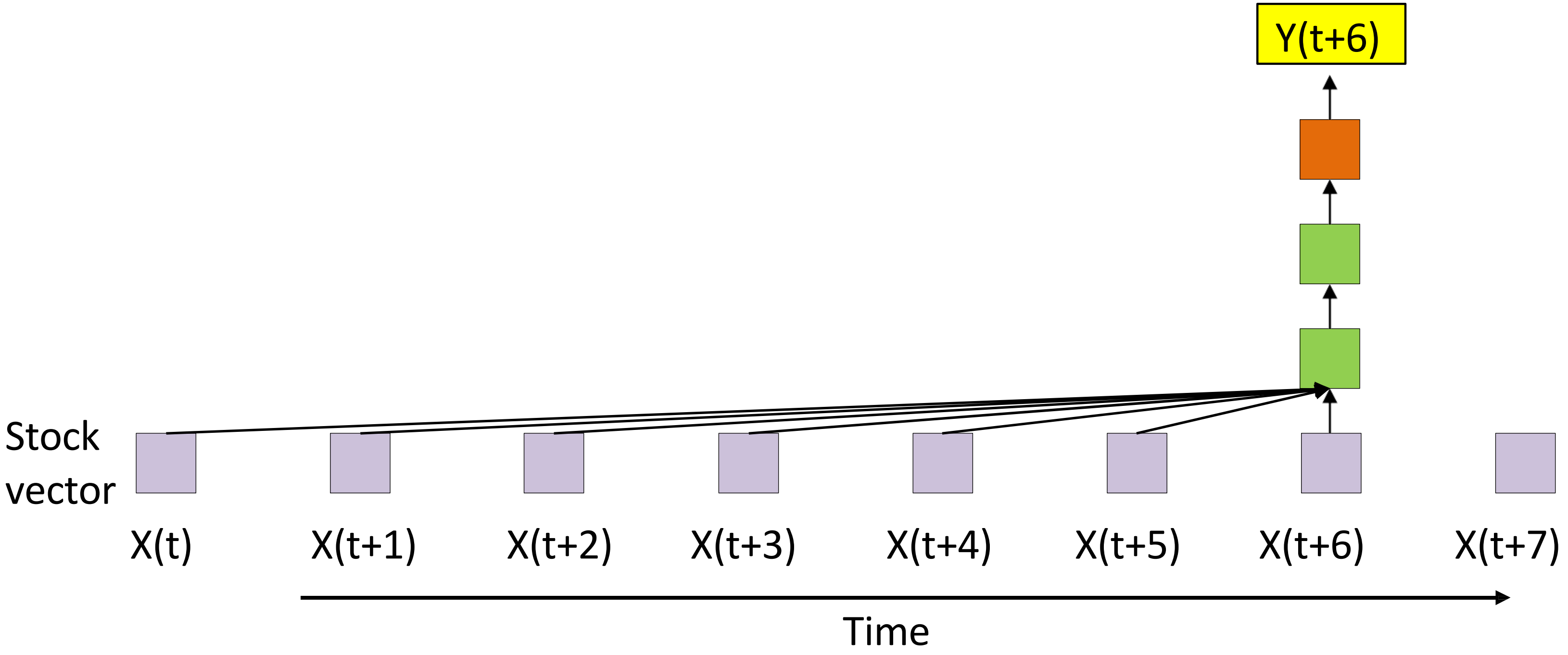
# Finite-response model




- Something that happens *today* only affects the output of the system for $N$ days into the future
  - **Predictions consider *N* days of history**

- To consider more of the past to make predictions, you must increase the "history" considered by the system

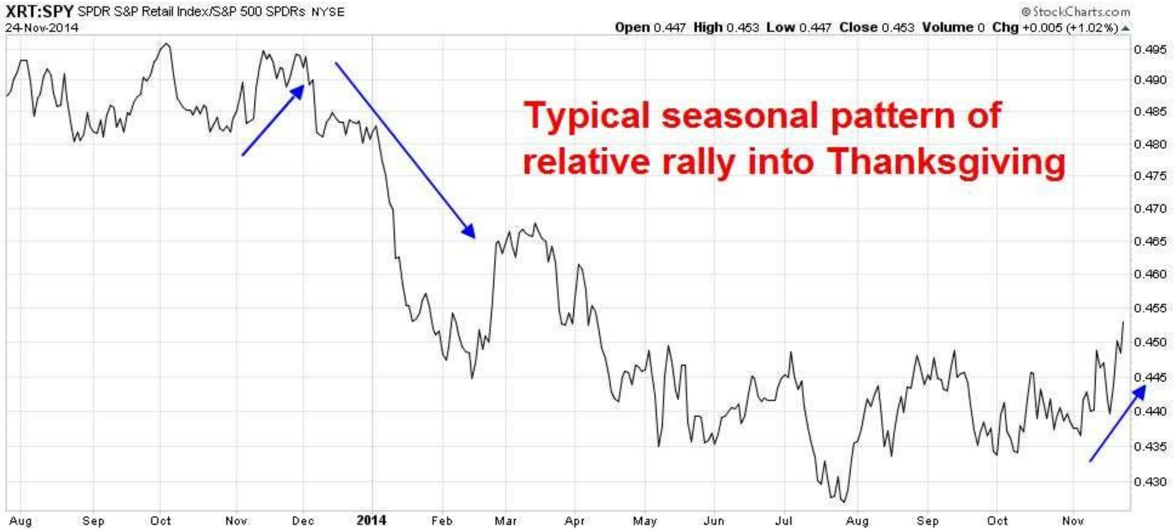
# Finite-response




- Problem: Increasing the "history" makes the network more complex

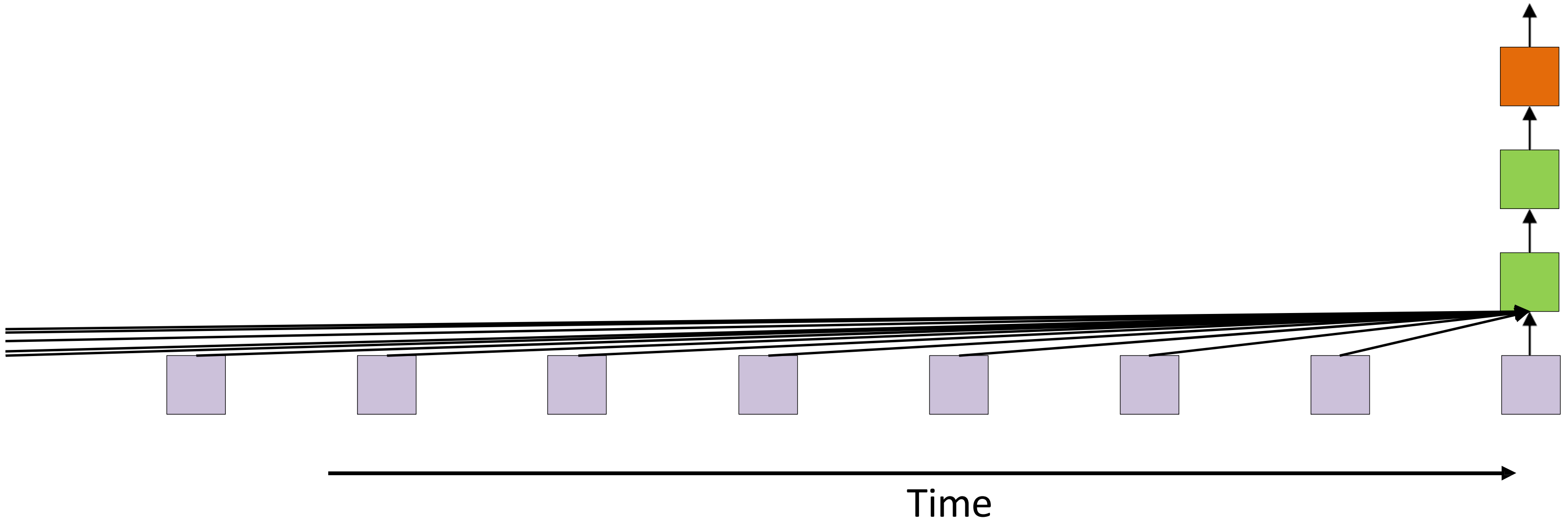
  -- require more computational resources

# Systems often have long-term dependencies




- Longer-term trends –
  - Weekly trends in the market
  - Monthly trends in the market
  - Annual trends

# We want *infinite* memory




- Required: *Infinite* response systems
  - What happens today can continue to affect the output forever
    - Possibly with weaker and weaker influence

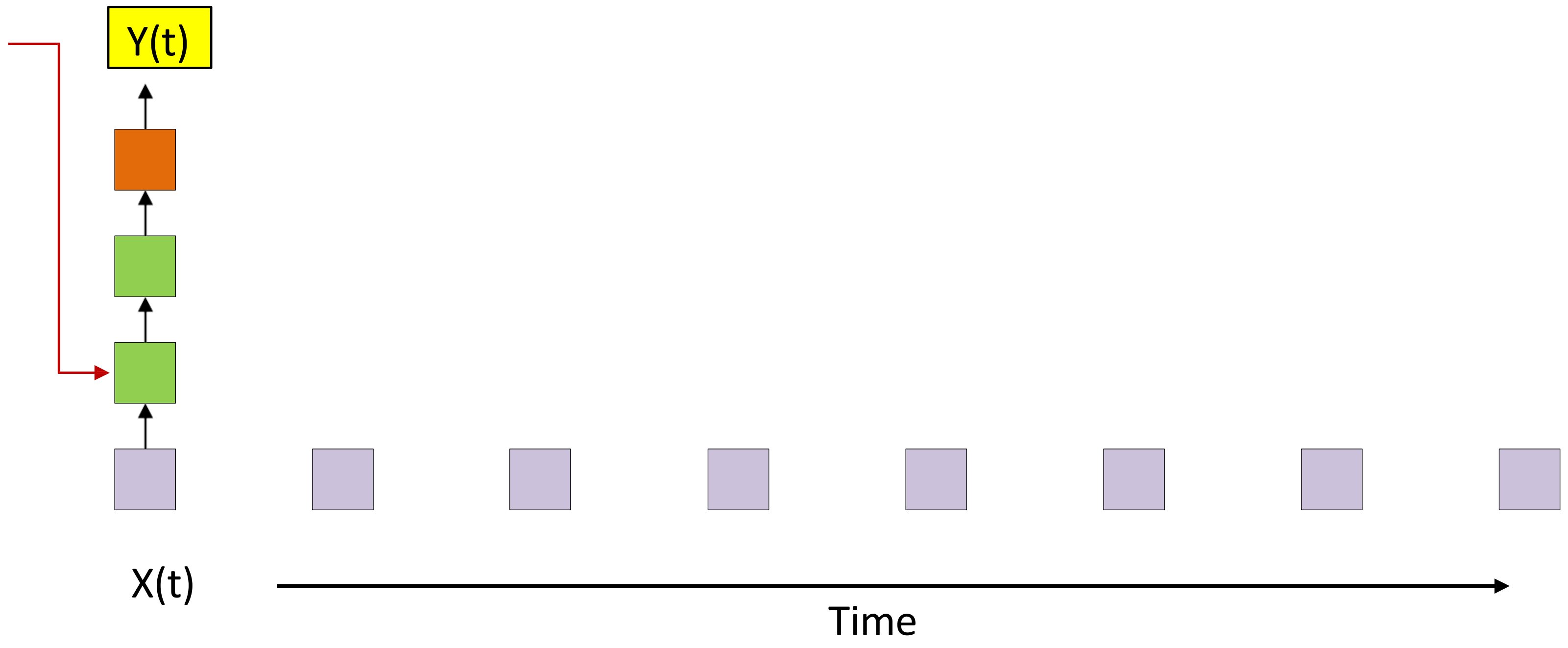
$$Y_t = f(X_t, X_{t-1}, \dots, X_{t-\infty})$$

# Examples of infinite response systems

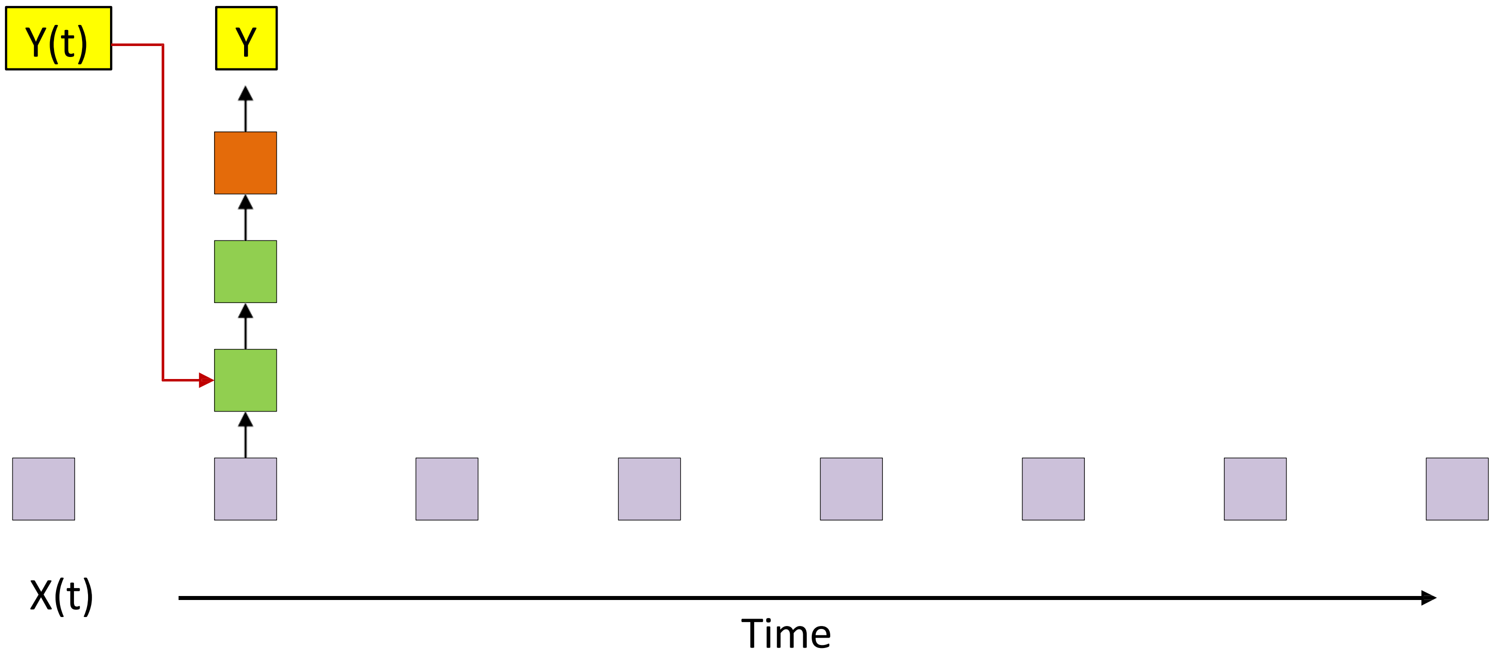
$$Y_t = f(X_t, Y_{t-1})$$

- Required: Define initial state: $Y_{-1}$ for $t = 0$
- An input $X_0$ at $t = 0$ produces $Y_0$
- $Y_0$ produces $Y_1$ which produces $Y_2$ and so on until $Y_\infty$ *even if* $X_1 \dots X_\infty$ *are 0*
  - i.e. even if there are no further inputs!
- **A single input influences the output for the rest of time!**

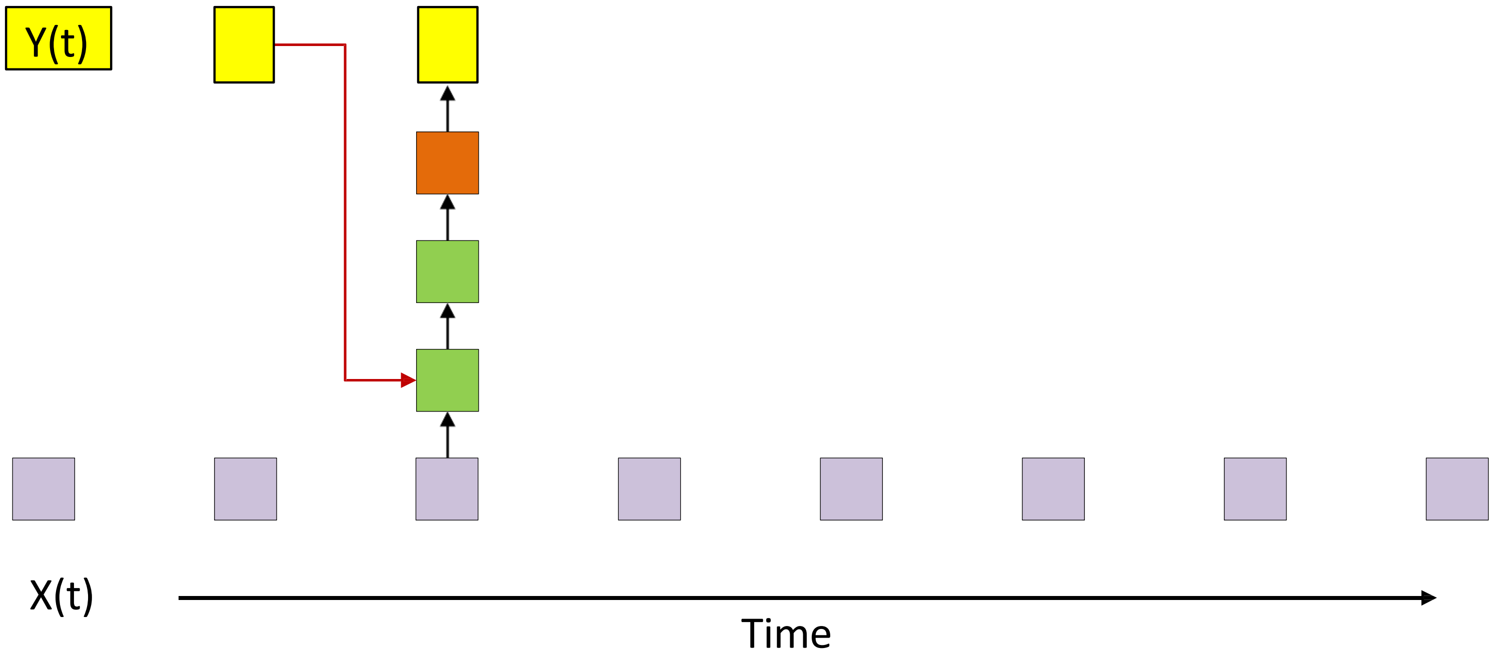
- This is an instance of a NARX network
  - "nonlinear autoregressive network with exogenous inputs"
  - $Y_t = f(X_{0:t}, Y_{0:t-1})$
- *Output* contains information about the entire past



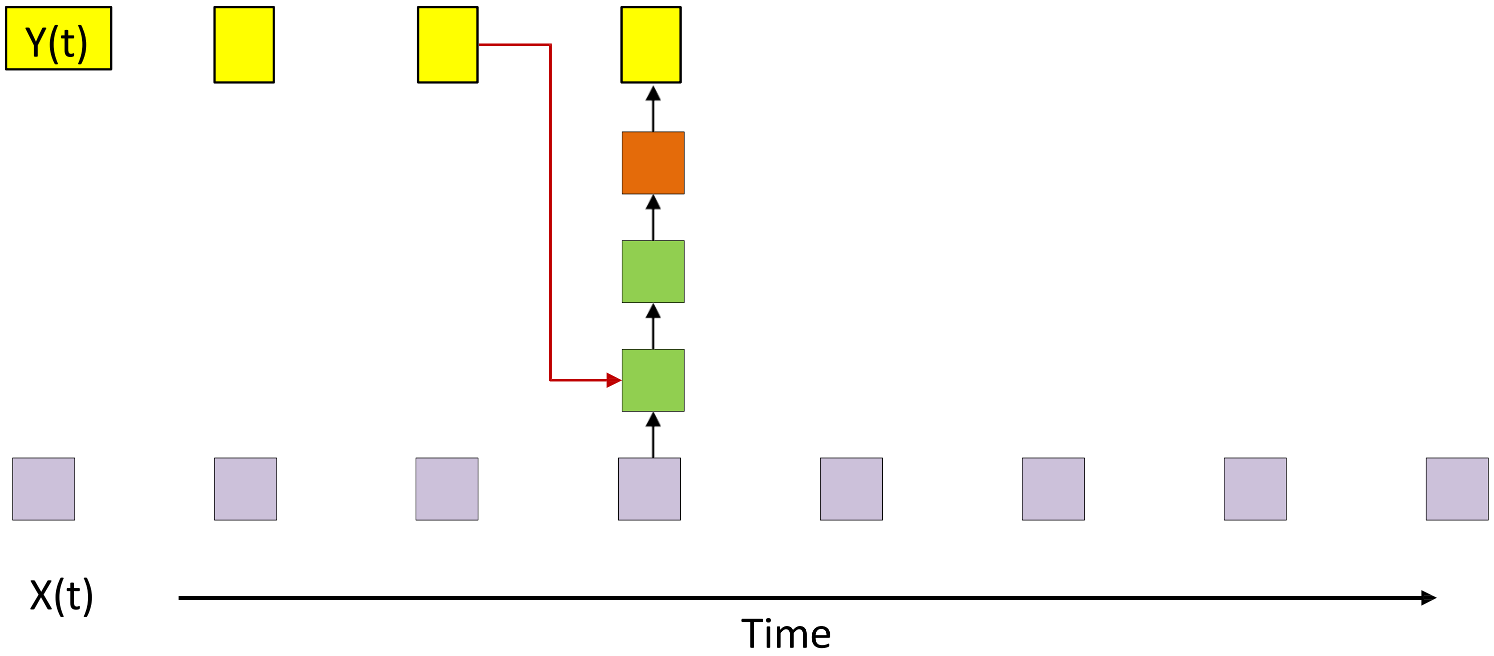

- A NARX net with recursion from the output



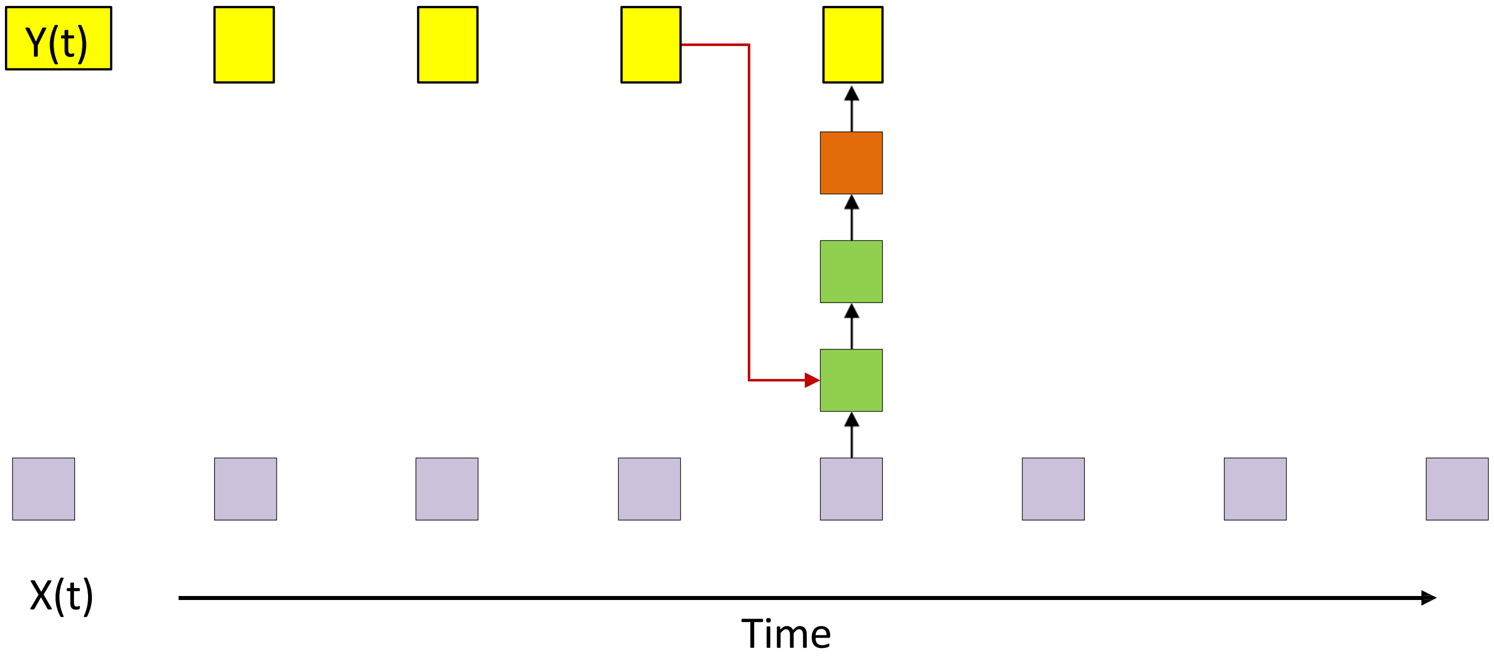

- A NARX net with recursion from the output



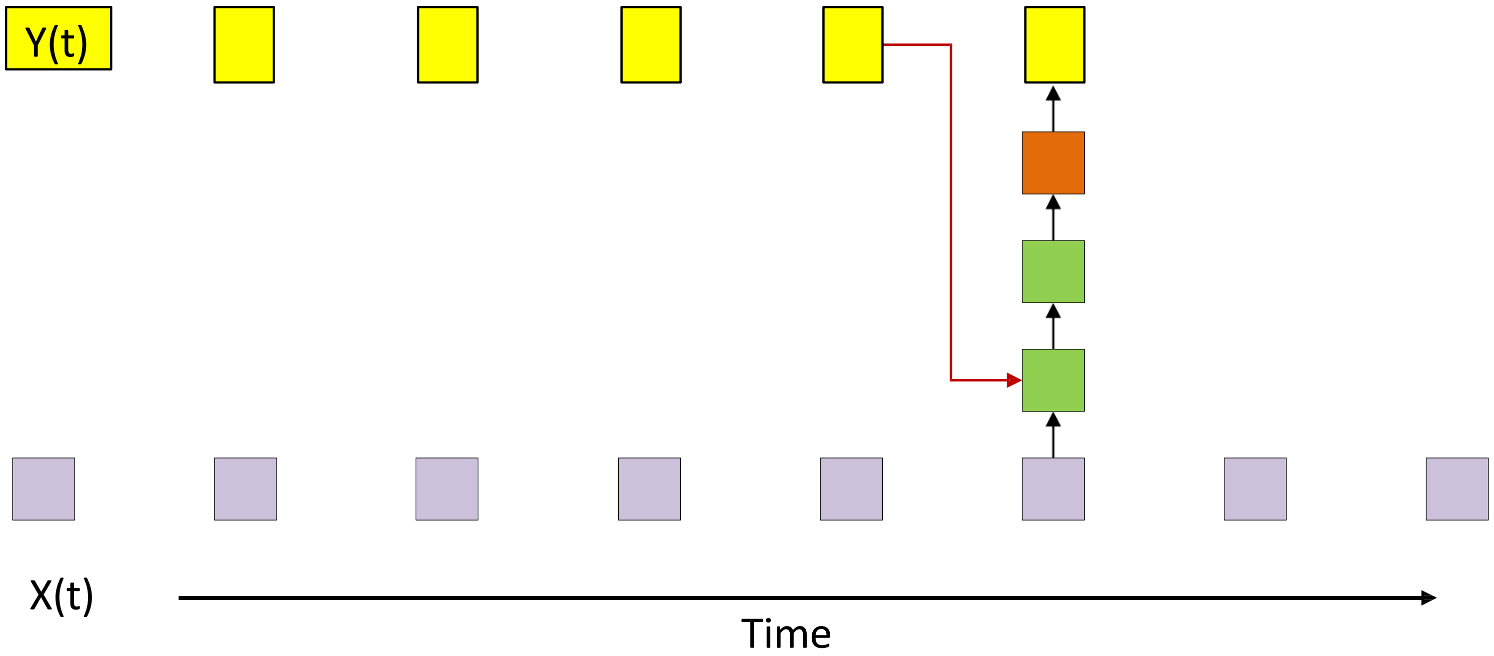

- A NARX net with recursion from the output



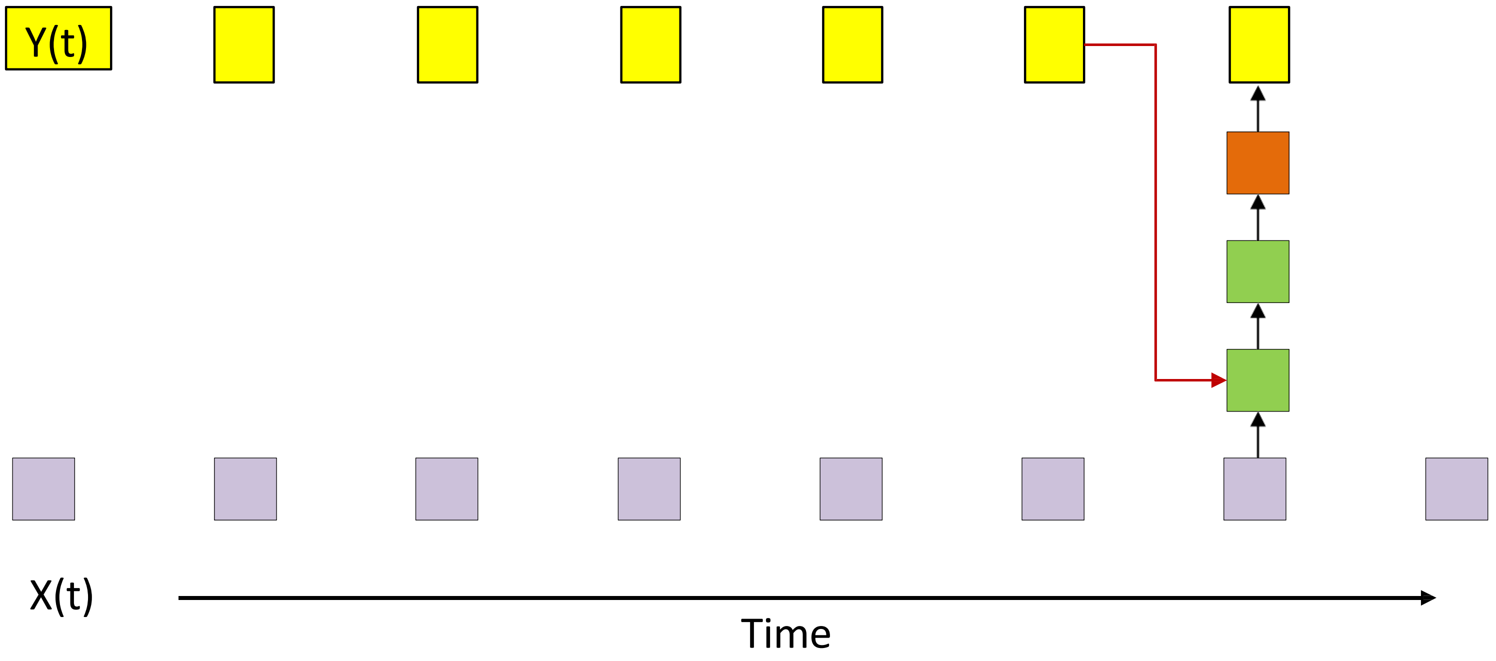

- A NARX net with recursion from the output



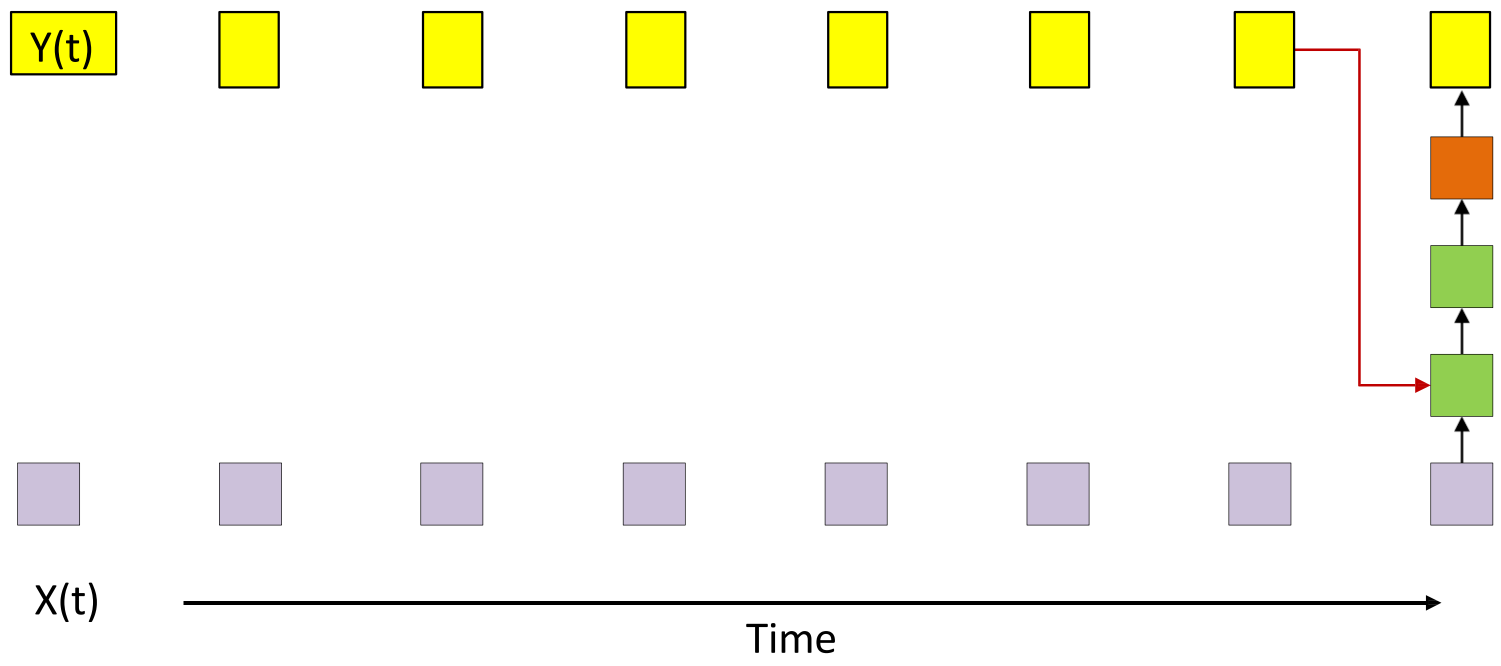

- A NARX net with recursion from the output




- A NARX net with recursion from the output

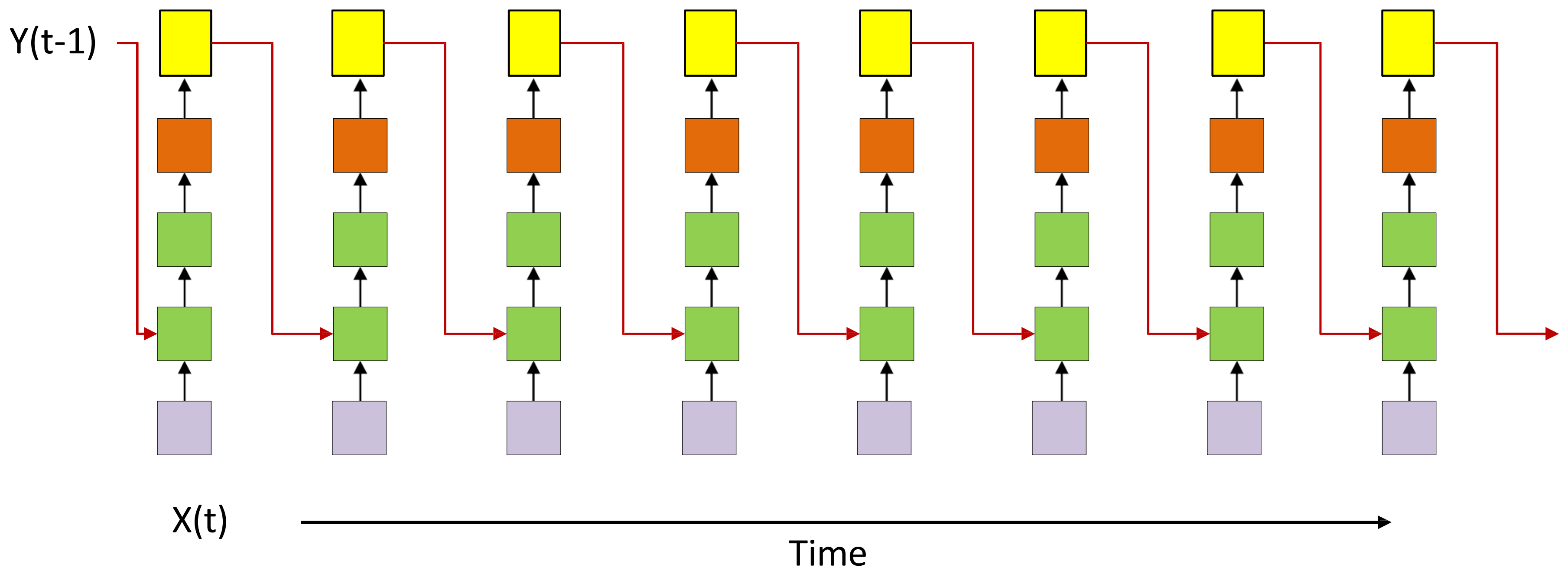



- A NARX net with recursion from the output




- A NARX net with recursion from the output

# A more complete representation




- A NARX net with recursion from the output
- Showing all computations
- All columns are identical
- *An input at t=0 affects outputs forever*

# NARX Networks

- Very popular for time-series prediction
  - Weather
  - Stock markets
  - As alternate system models in tracking systems
  - ***Language***

- Note: here the “memory” of the past is in the output itself, and *not in the network*

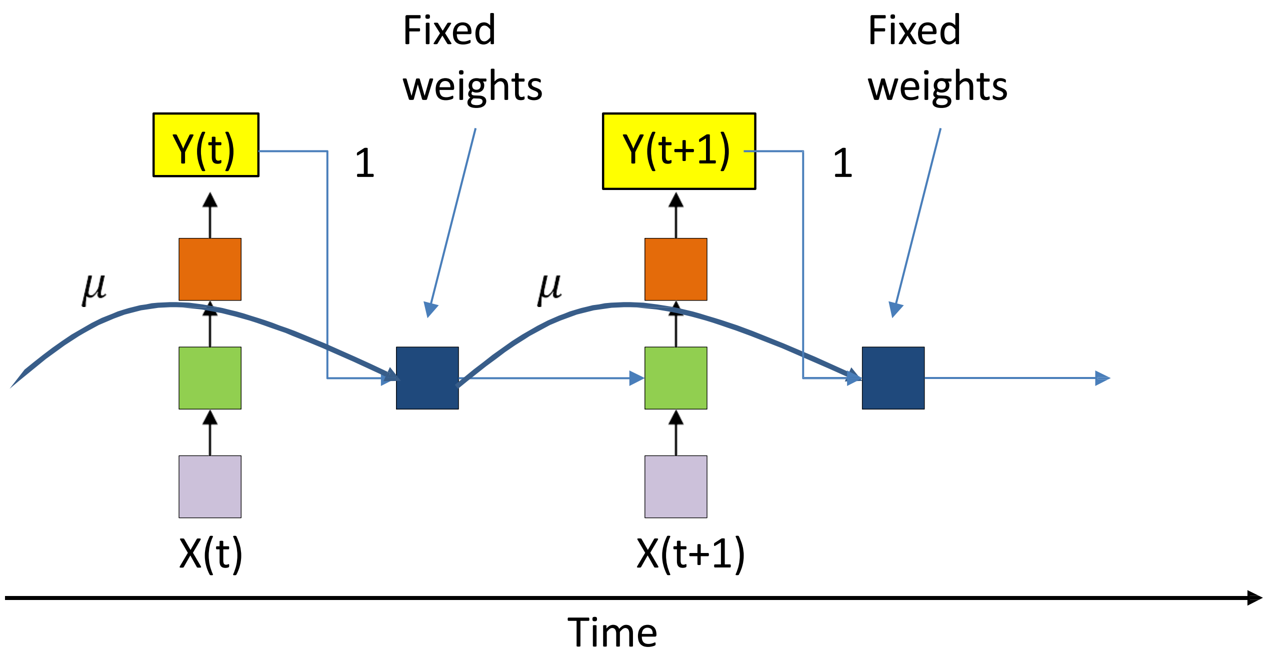
# Let’s make memory more explicit

- Task is to “remember” the past
- Introduce an explicit *memory* variable whose *job* it is to remember

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

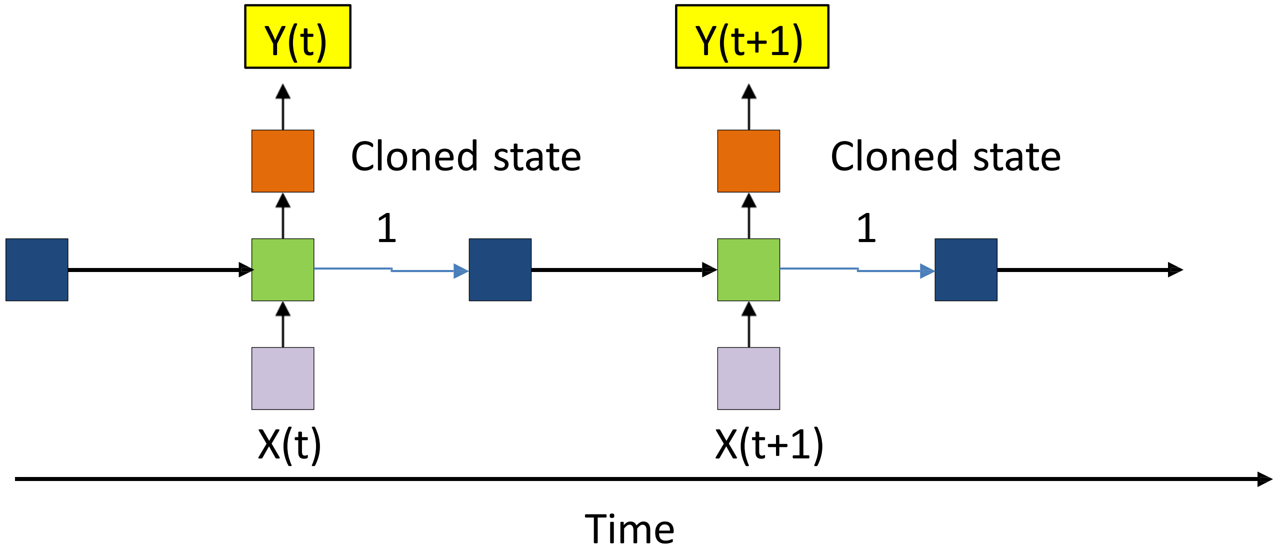
- $m_t$ is a “memory” variable
  - Generally stored in a “memory” unit
  - Used to “remember” the past

# Jordan Network




- Memory unit simply retains a running average of past outputs
  - "Serial order: A parallel distributed processing approach", M.I.Jordan, 1986
  - Memory has fixed structure; does not "learn" to remember
    - The running average of outputs considers entire past, rather than immediate past

# Elman Networks




- Separate memory state from output
  - "Context" units that carry historical state
  - "Finding structure in time", Jeffrey Elman, Cognitive Science, 1990
- Only the weight *from* the memory unit to the hidden unit is learned
  - But during training no gradient is backpropagated over the "1" link

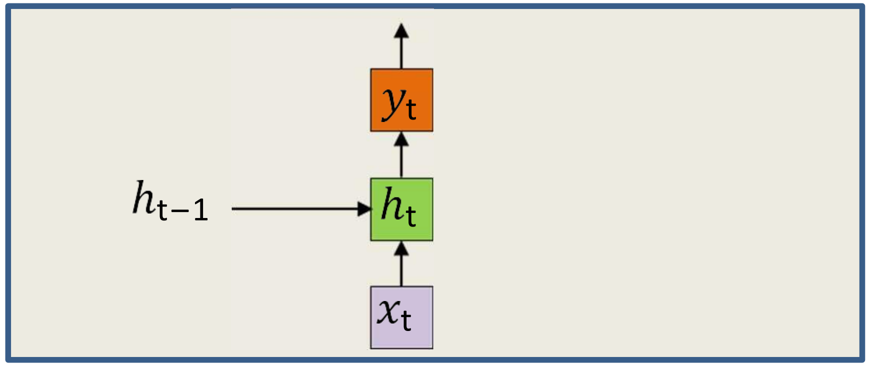
# An alternate model for infinite response systems: the state-space model

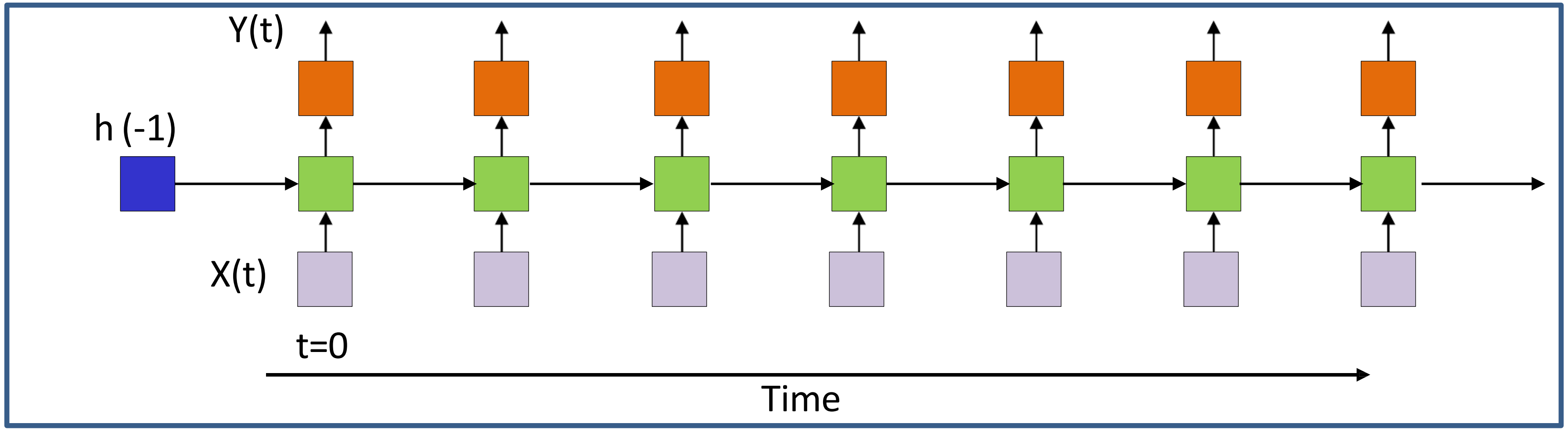
$$h_t = f(x_t, h_{t-1})$$
$$y_t = g(h_t)$$

- $h_t$ is the *state* of the network
  - *State* summarizes information about the entire past
    - Model directly embeds the memory in the state
- Need to define initial state $h_{-1}$

- This is a *recurrent* neural network

# The simple state-space model



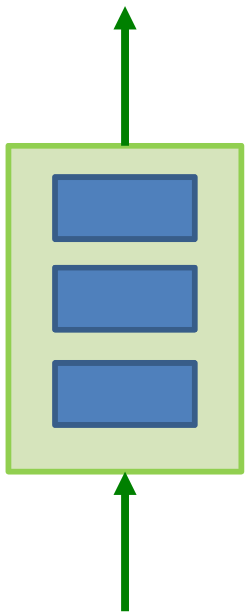

- The state (green) at any time is determined by the input at that time, and the state at the previous time
- *An input at t=0 affects outputs forever*
- Also known as a recurrent neural net

# An alternate model for infinite response systems: the state-space model

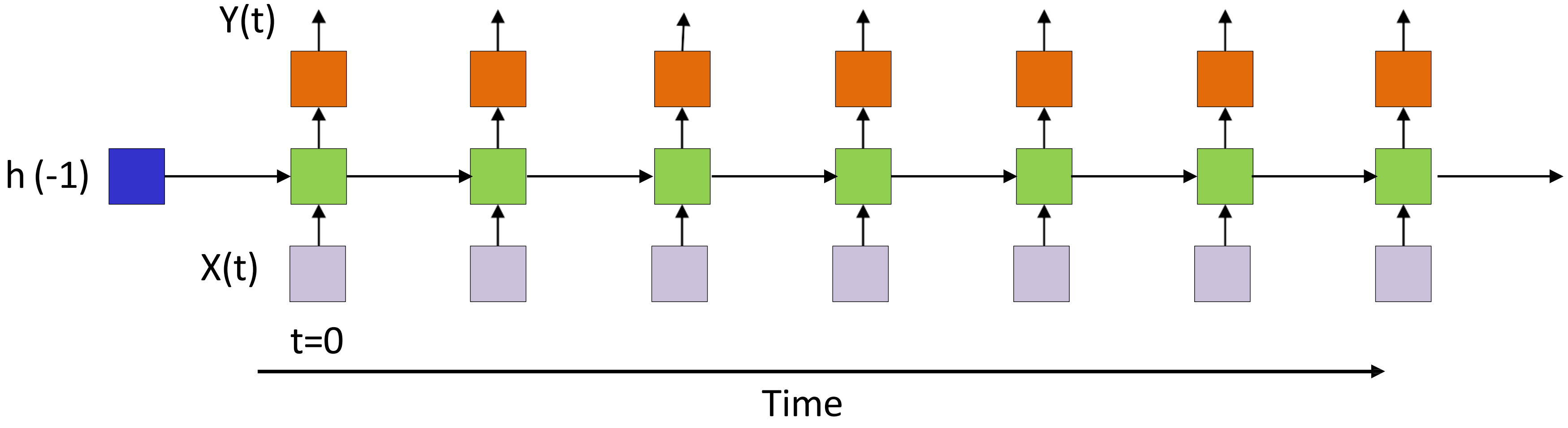
$$h_t = f(x_t, h_{t-1})$$
$$y_t = g(h_t)$$

- $h_t$ is the *state* of the network
- Need to define initial state $h_{-1}$

- The state can be *arbitrarily complex*

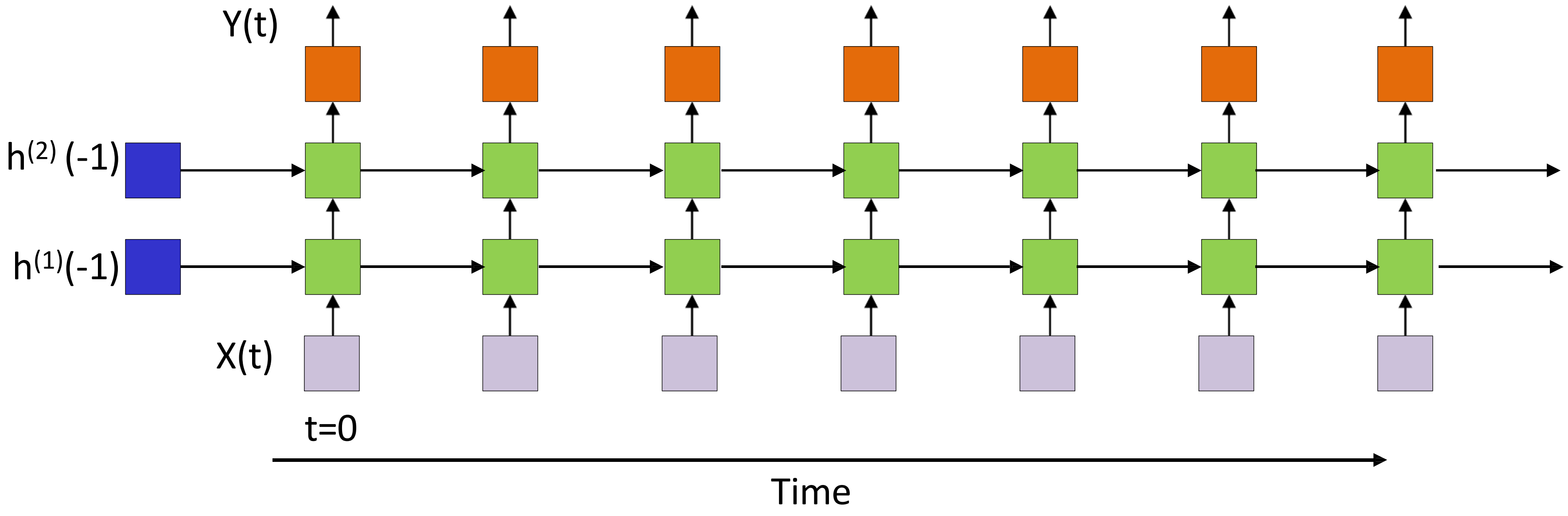


# Single hidden layer RNN



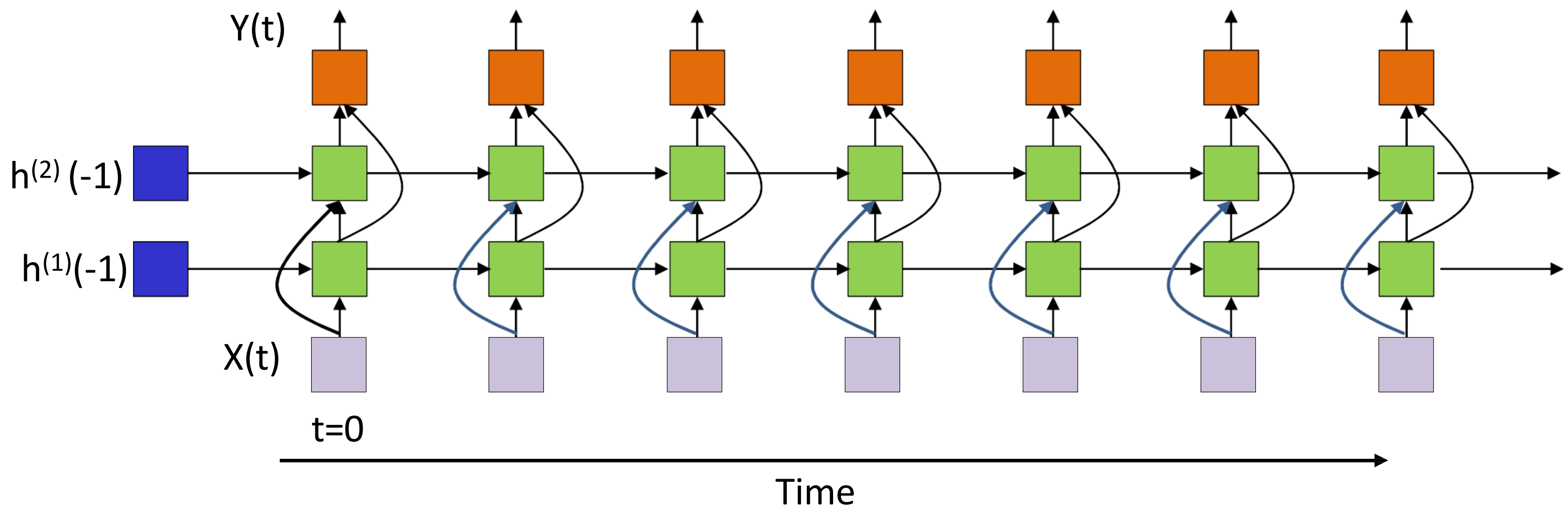

- Recurrent neural network
- All columns are identical
- *An input at t=0 affects outputs forever*

# Multiple recurrent layer RNN



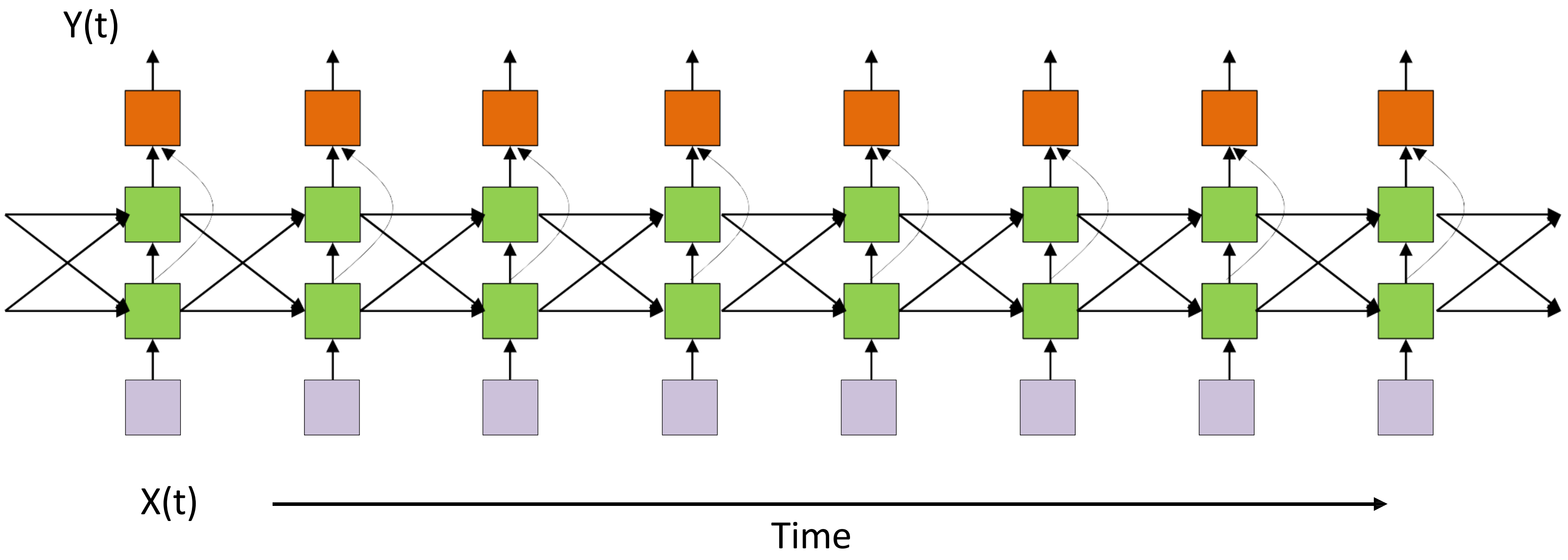

- Recurrent neural network
- All columns are identical
- *An input at t=0 affects outputs forever*

# Multiple recurrent layer RNN



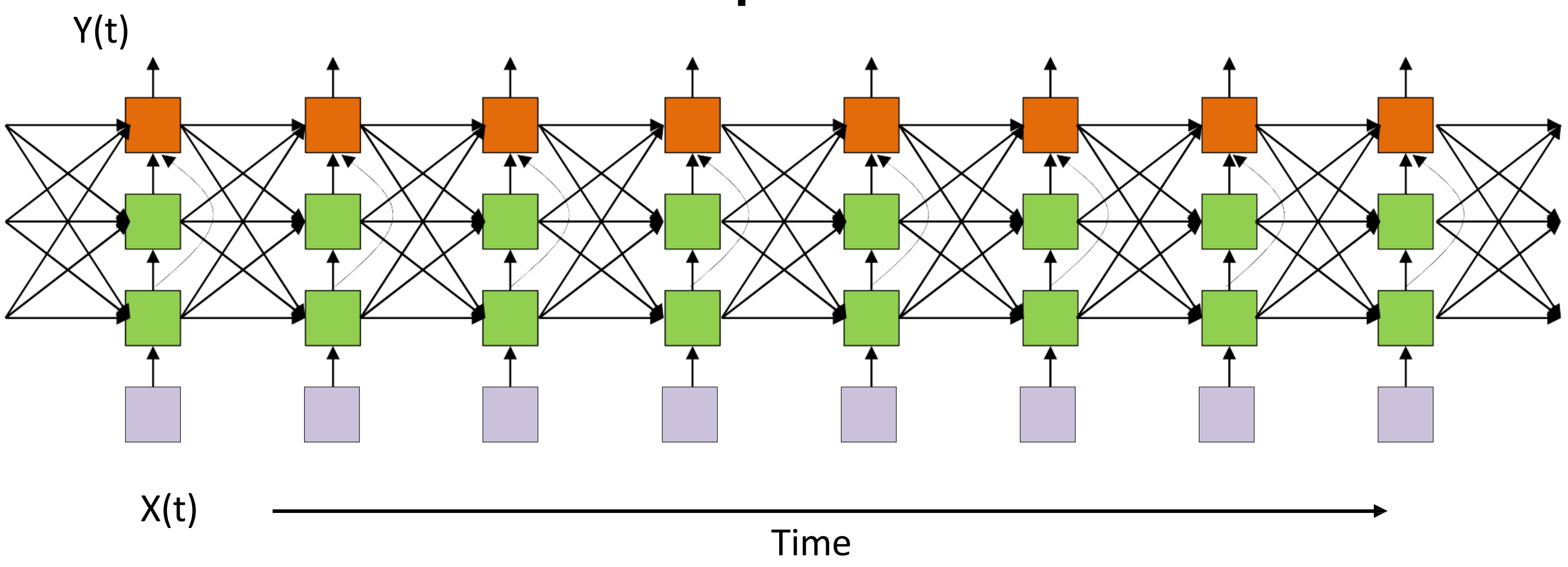

- We can also have skips..

# A more complex state



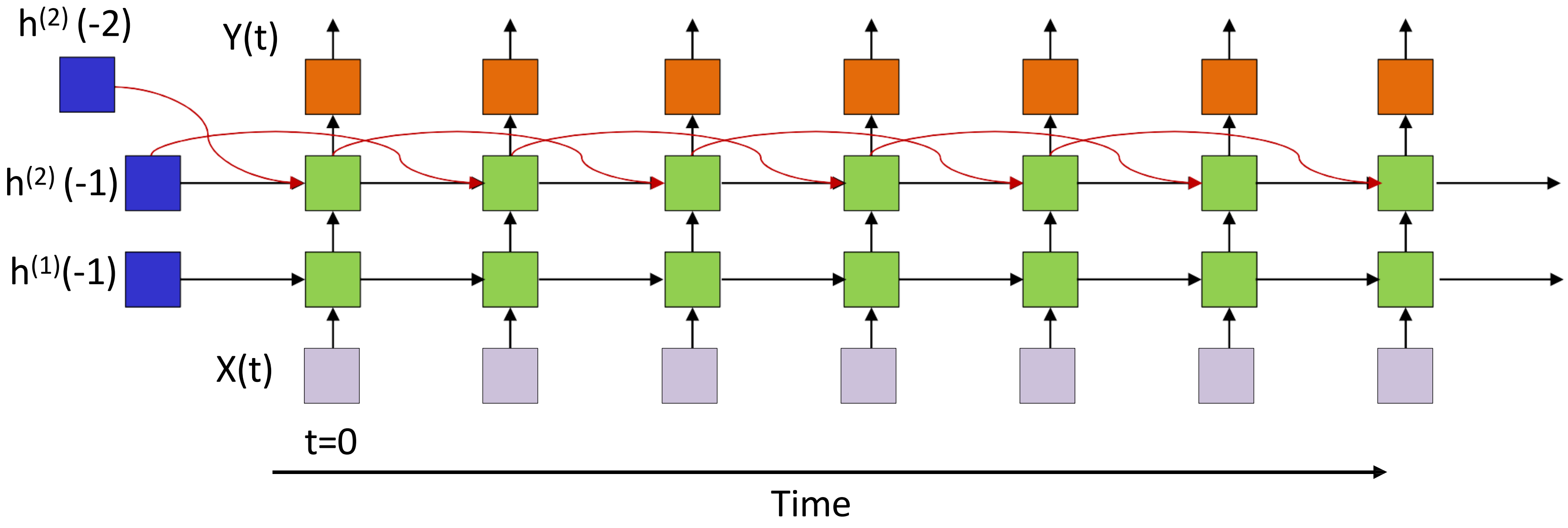

- All columns are identical
- *An input at t=0 affects outputs forever*

# Or the network may be even more complicated



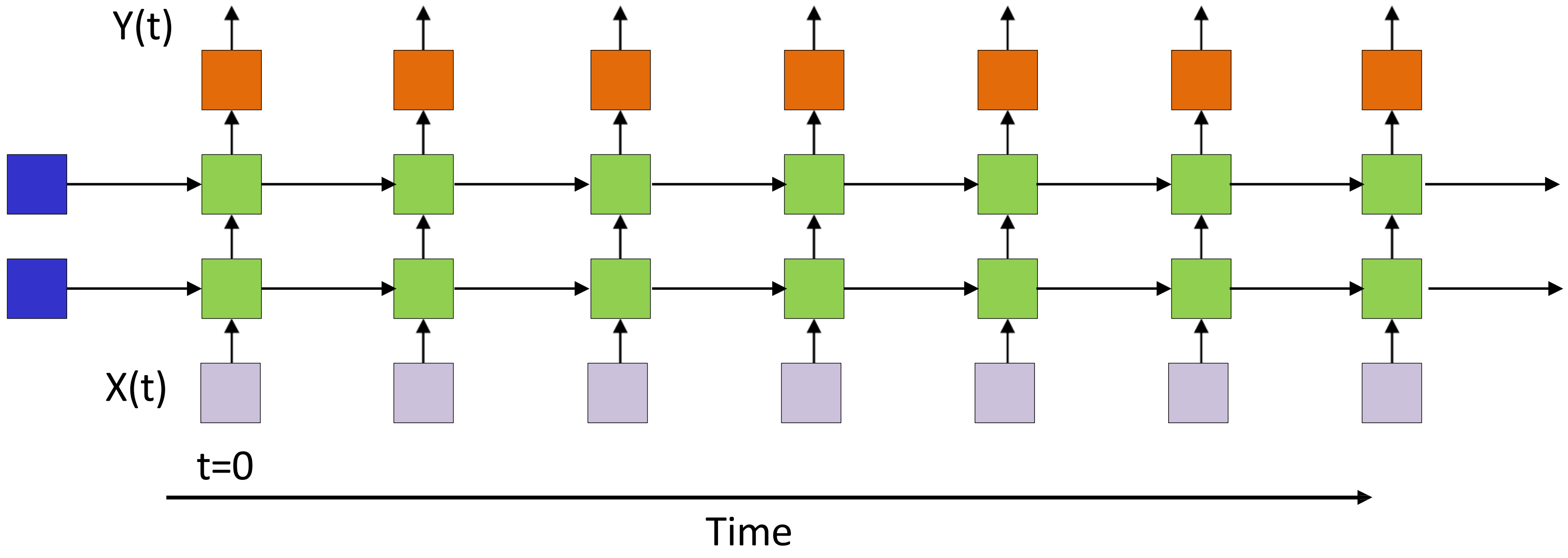

- All columns are identical
- *An input at t=0 affects outputs forever*

# Generalization with other recurrences



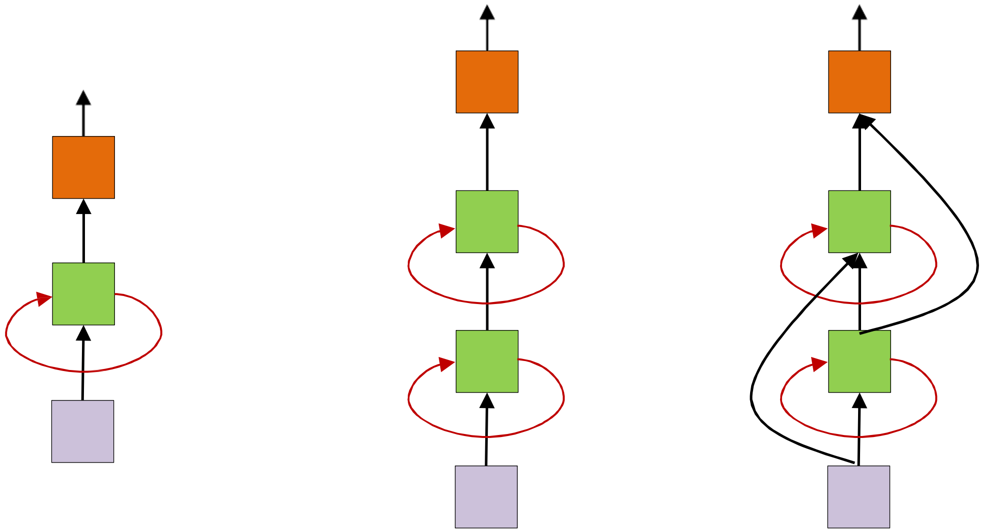

- All columns (including incoming edges) are identical

# The simplest structures are most popular



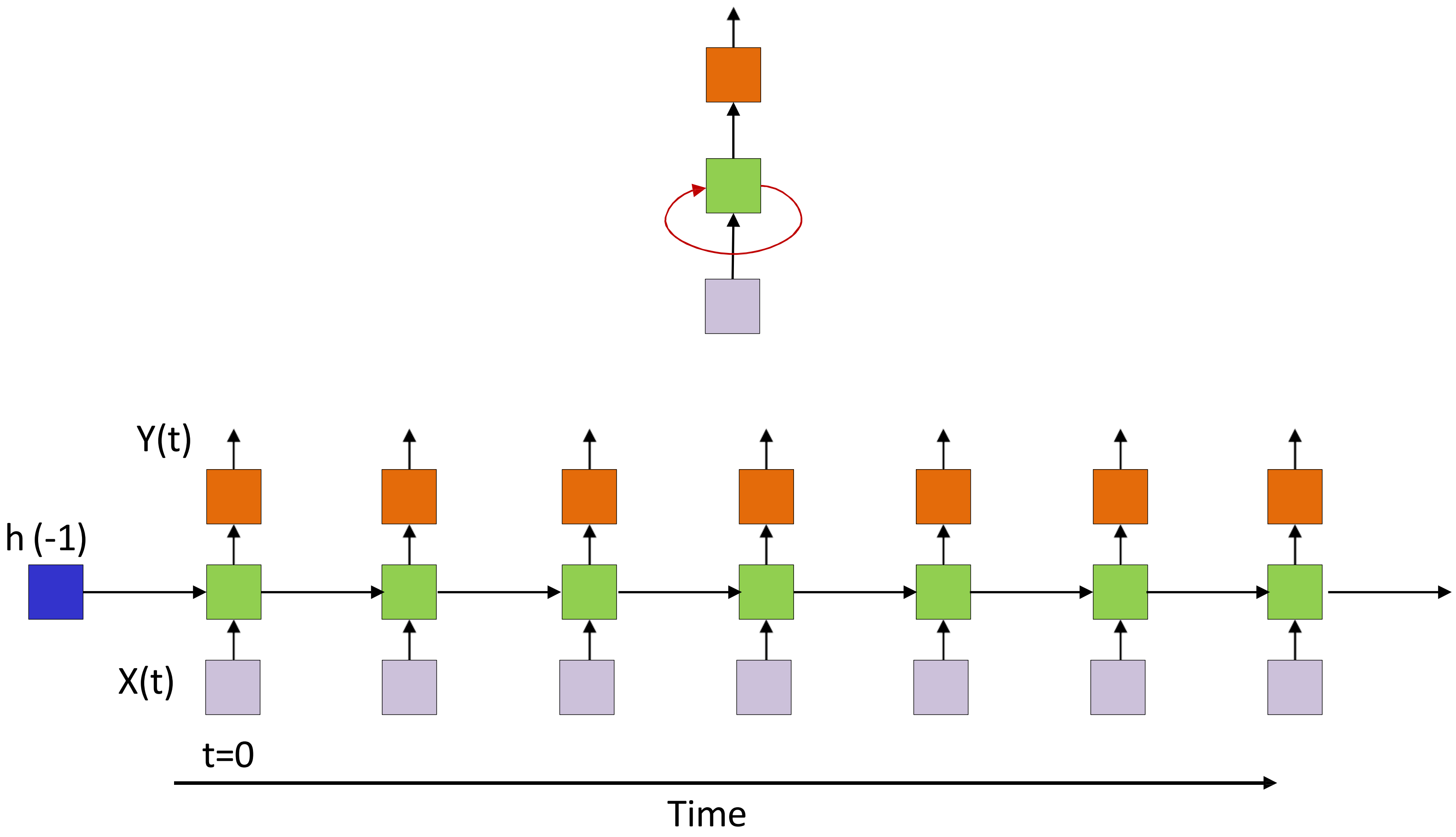

- Recurrent neural network
- All columns are identical
- *An input at t=0 affects outputs forever*

# A Recurrent Neural Network

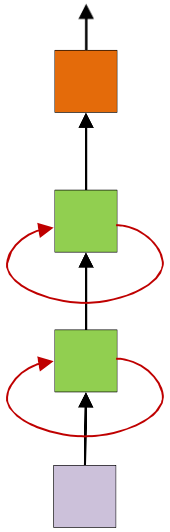


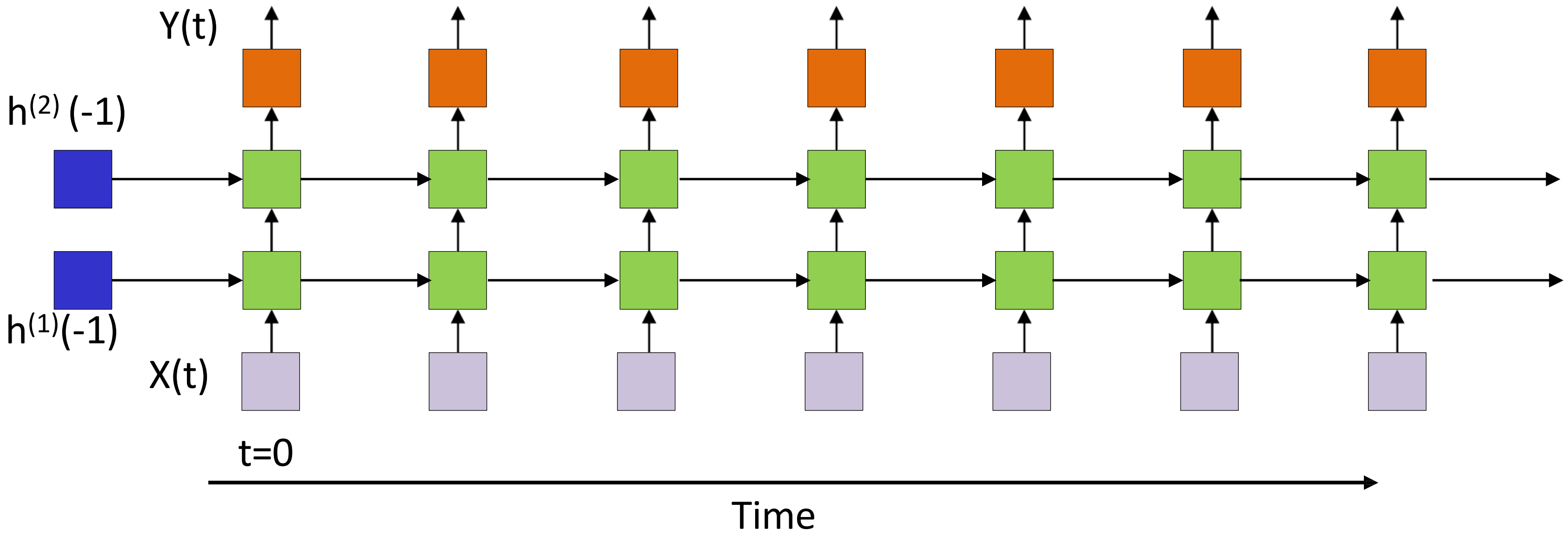
- Simplified models often drawn
- The loops imply recurrence

# The detailed version of the simplified representation

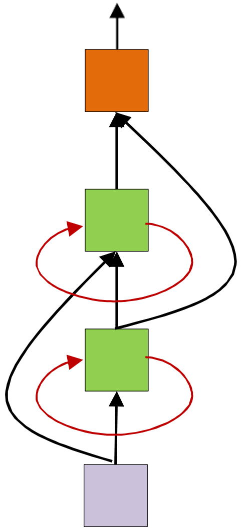


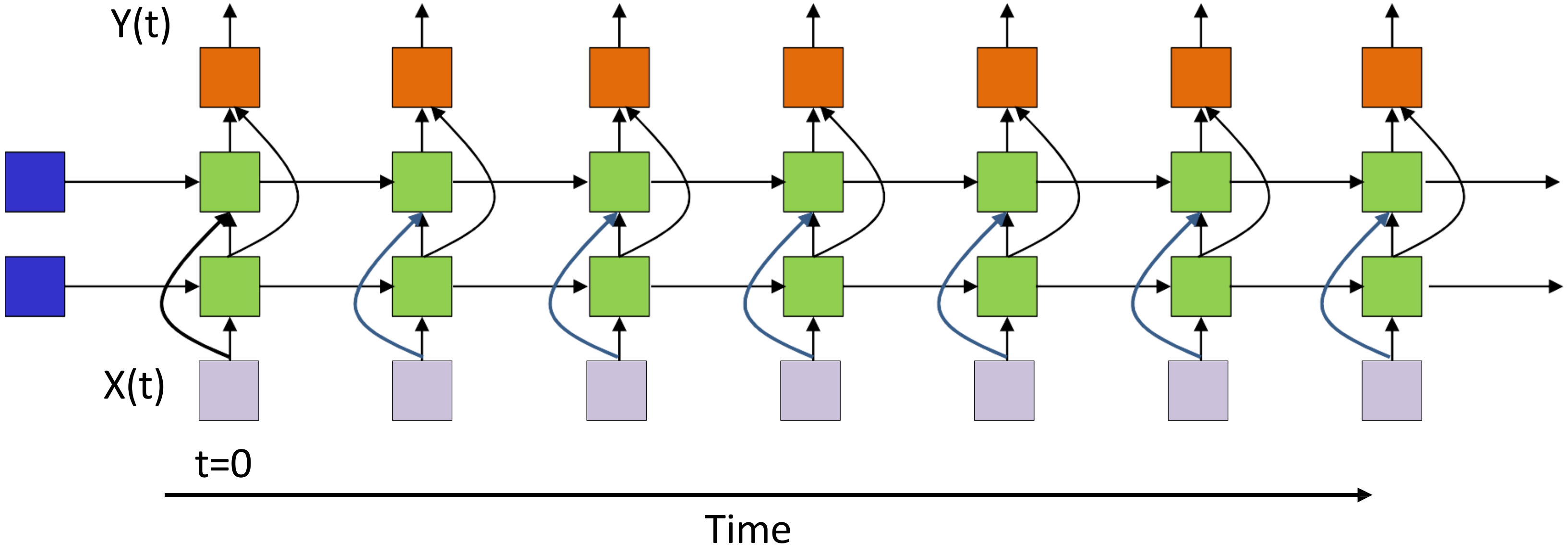
# Multiple recurrent layer RNN



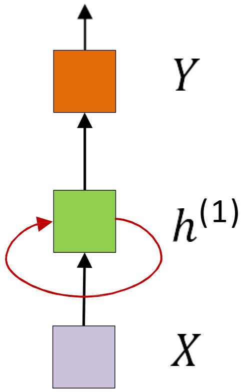


# Multiple recurrent layer RNN





# Equations




Current weights

Recurrent weights

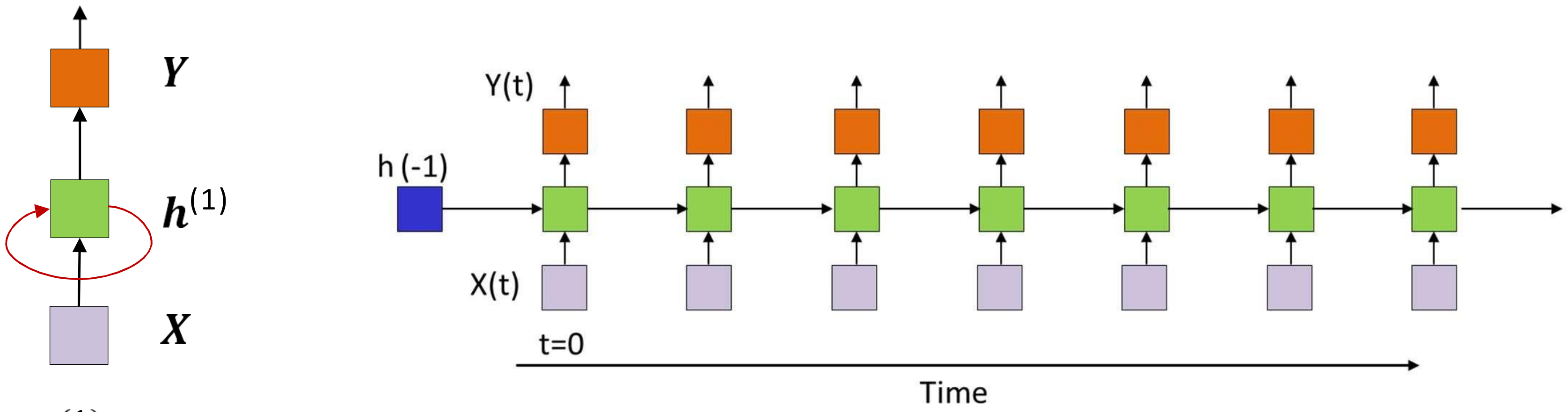
$$h_i^{(1)}(-1) = part\ of\ network\ parameters$$

$$h_i^{(1)}(t) = f_1\left(\sum_j w_{ji}^{(1)} X_j(t) + \sum_j w_{ji}^{(11)} h_j^{(1)}(t-1) + b_i^{(1)}\right)$$

$$Y(t) = f_2\left(\sum_j w_{jk}^{(2)} h_j^{(1)}(t) + b_k^{(2)}, k = 1..M\right)$$

- Note superscript in indexing, which indicates layer of network from which inputs are obtained
- Assuming vector function at output, e.g. softmax
- The *state* node activation, $f_1()$ is typically $\tanh()$
- Every neuron also has a *bias* input

# Equations




$\boldsymbol{h}^{(1)}(-1) = part\ of\ network\ parameters$

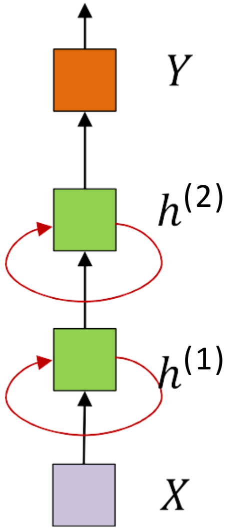
- Computation:

$$\boldsymbol{h}^{(1)}(t) = f_1\left(\boldsymbol{W}^{(1)}\boldsymbol{X}(t) + \boldsymbol{W}^{(11)}\boldsymbol{h}^{(1)}(t-1) + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{Y}(t) = f_2\left(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)}(t) + \boldsymbol{b}^{(2)}\right)$$

- The recurrent state activation $f_1()$ is typically $\tanh()$

# Equations




$$h_i^{(1)}(-1) = part\ of\ network\ parameters$$

$$h_i^{(2)}(-1) = part\ of\ network\ parameters$$

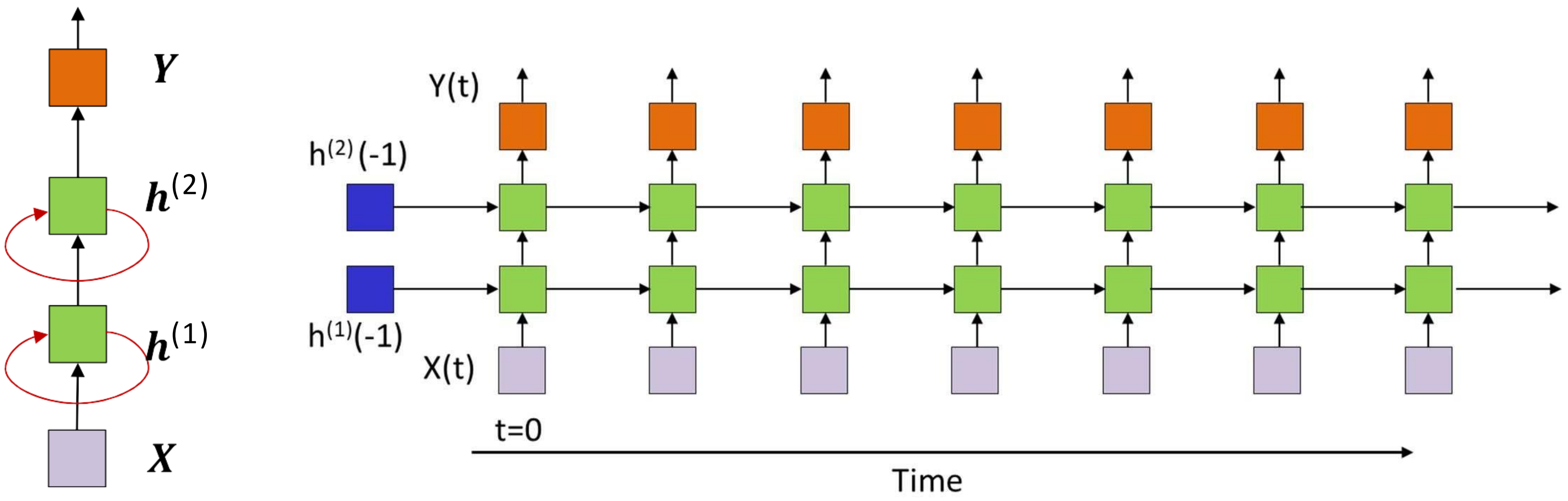
$$h_i^{(1)}(t) = f_1\left(\sum_j w_{ji}^{(1)} X_j(t) + \sum_j w_{ji}^{(11)} h_j^{(1)}(t-1) + b_i^{(1)}\right)$$

$$h_i^{(2)}(t) = f_2\left(\sum_j w_{ji}^{(2)} h_j^{(1)}(t) + \sum_j w_{ji}^{(22)} h_j^{(2)}(t-1) + b_i^{(2)}\right)$$

$$Y(t) = f_3\left(\sum_j w_{jk}^{(3)} h_j^{(2)}(t) + b_k^{(3)}, k = 1..M\right)$$

- Assuming vector function at output, e.g. softmax $f_3()$
- The *state* node activations, $f_k()$ are typically $\tanh()$
- Every neuron also has a *bias* input

# Equations




$$\boldsymbol{h}^{(1)}(-1)\ and\ \boldsymbol{h}^{(2)}(-1) = part\ of\ network\ parameters$$

- Computation:

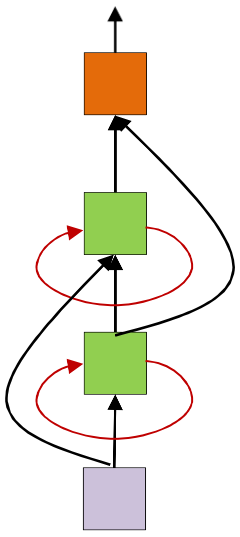
$$\boldsymbol{h}^{(1)}(t) = f_1\left(\boldsymbol{W}^{(1)}\boldsymbol{X}(t) + \boldsymbol{W}^{(11)}\boldsymbol{h}^{(1)}(t-1) + \boldsymbol{b}^{(1)}\right)$$
$$\boldsymbol{h}^{(2)}(t) = f_2\left(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)}(t) + \boldsymbol{W}^{(22)}\boldsymbol{h}^{(2)}(t-1) + \boldsymbol{b}^{(2)}\right)$$
$$\boldsymbol{Y}(t) = f_3\left(\boldsymbol{W}^{(3)}\boldsymbol{h}^{(2)}(t) + \boldsymbol{b}^{(3)}\right)$$

- The recurrent state activation is typically $\tanh()$

# Equations

$$\boldsymbol{h}^{(1)}(-1) = part\ of\ network\ parameters$$

$$\boldsymbol{h}^{(2)}(-1) = part\ of\ network\ parameters$$

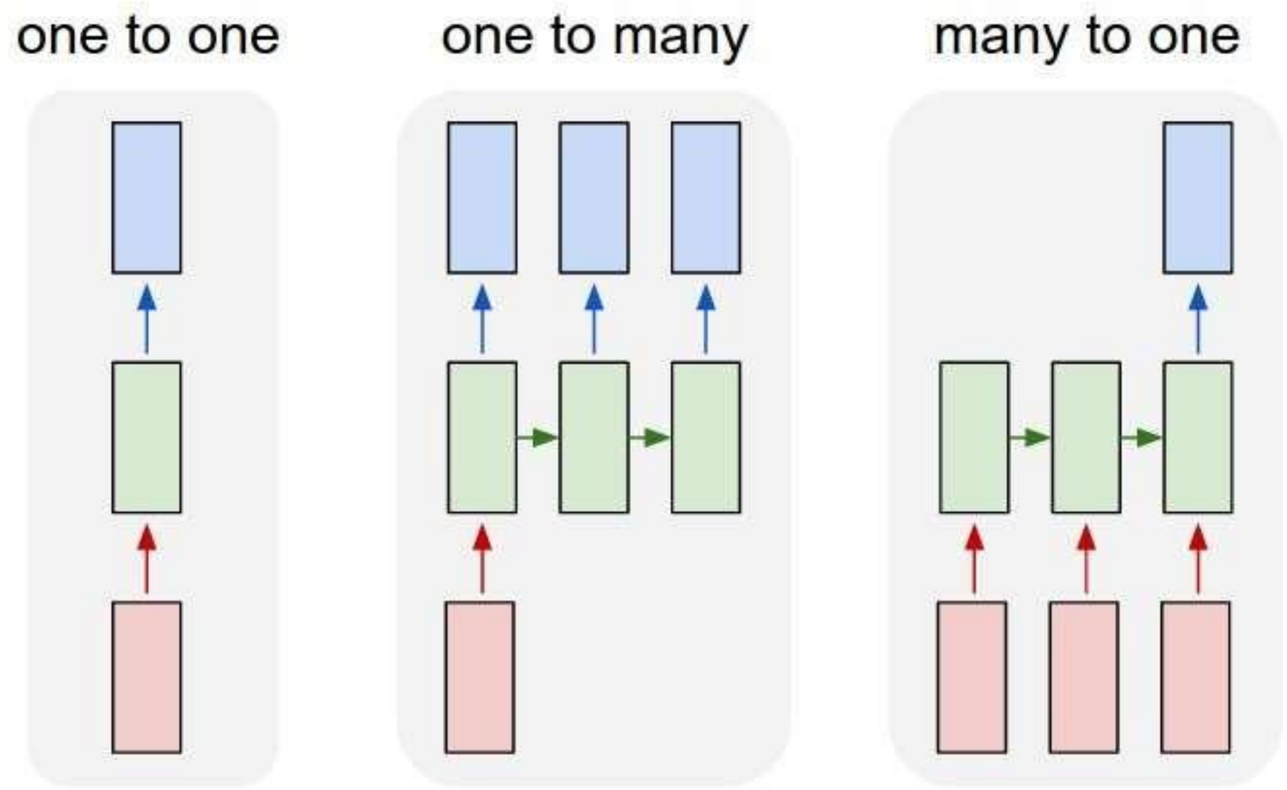


$$\boldsymbol{h}^{(1)}(t) = f_1\left(\boldsymbol{W}^{(01)}\boldsymbol{X}(t) + \boldsymbol{W}^{(11)}\boldsymbol{h}^{(1)}(t-1) + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{h}^{(2)}(t) = f_2\left(\boldsymbol{W}^{(12)}\boldsymbol{h}^{(1)}(t) + \boldsymbol{W}^{(02)}\boldsymbol{X}(t) + \boldsymbol{W}^{(22)}\boldsymbol{h}^{(2)}(t-1) + \boldsymbol{b}^{(2)}\right)$$

$$\boldsymbol{Y}(t) = f_3\left(\boldsymbol{W}^{(23)}\boldsymbol{h}^{(2)}(t) + \boldsymbol{W}^{(13)}\boldsymbol{h}^{(1)}(t) + \boldsymbol{b}^{(3)}\right)$$

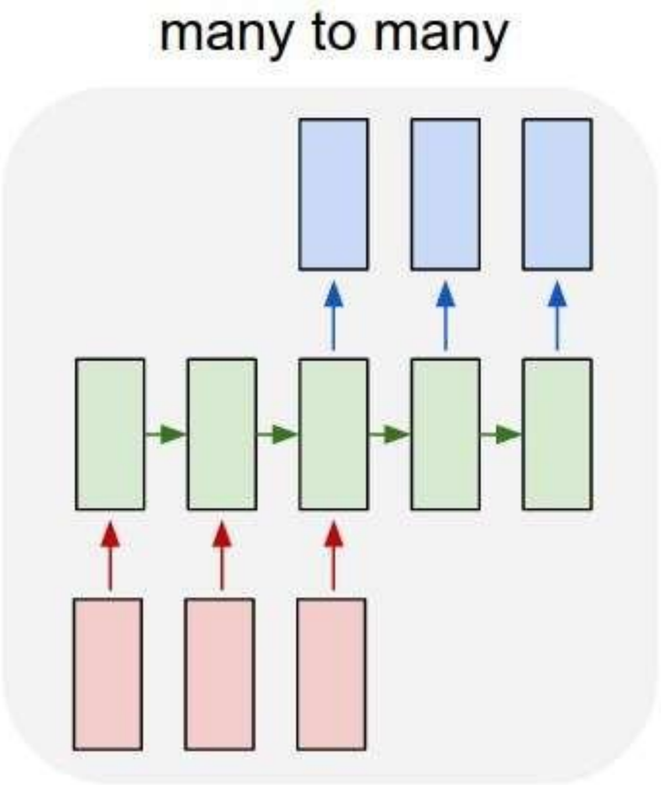
# Variants on recurrent nets

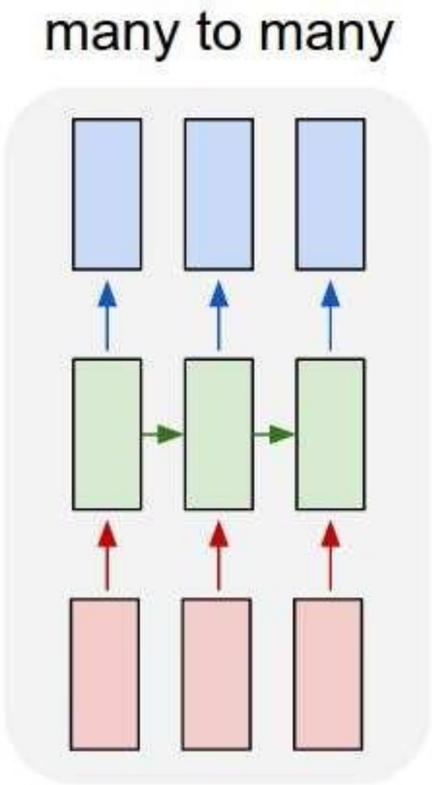



Images from Karpathy

- 1: Conventional MLP
- 2: Sequence *generation*, e.g. image to caption
- 3: Sequence based *prediction or classification,* e.g. Speech recognition, text classification

# Variants

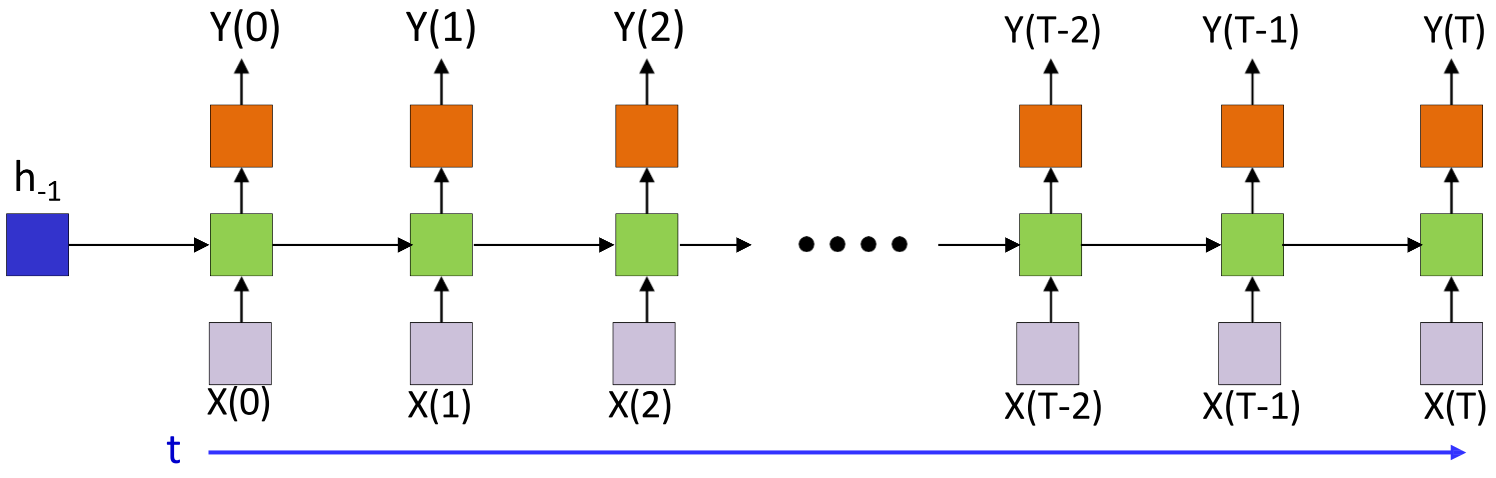



Images from
Karpathy

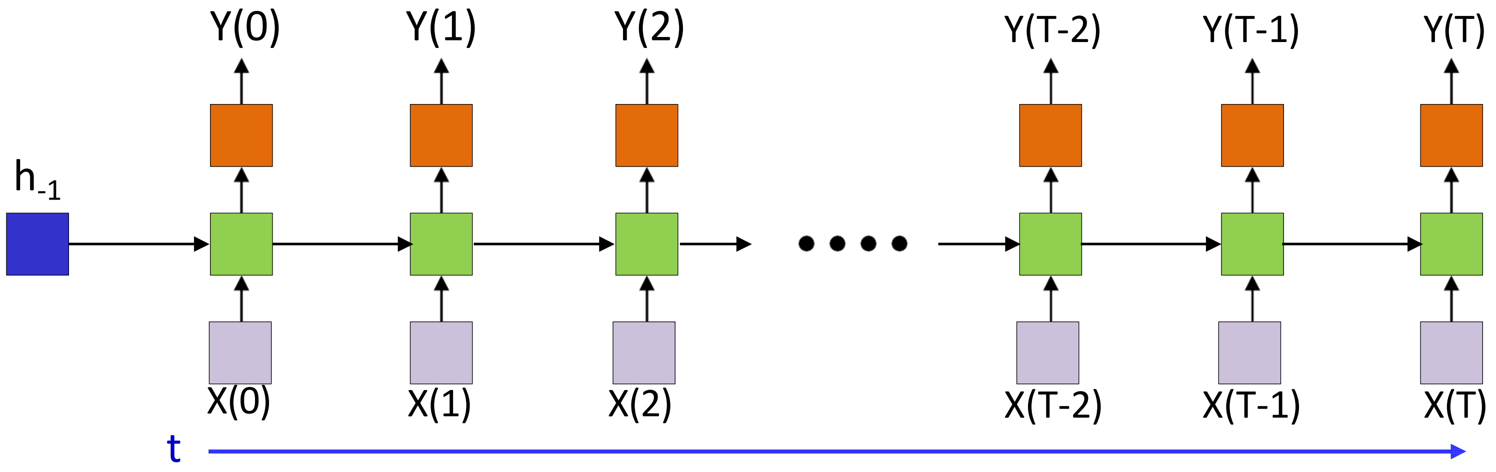
- 1: *Delayed* sequence to sequence, e.g. machine translation
- 2: Sequence to sequence, e.g. stock problem, label prediction
- Etc…

# How do we *train* the network




- Back propagation through time (BPTT)
- Given a collection of *sequence* inputs
  - $(\mathbf{X}_i, \mathbf{D}_i)$
  - $\mathbf{X}_i = X_{i,0}, \ldots, X_{i,T}$
  - $\mathbf{D}_i = D_{i,0}, \ldots, D_{i,T}$
- Train network parameters to minimize the error between the output of the network $\mathbf{Y}_i = Y_{i,0}, \ldots, Y_{i,T}$ and the desired outputs
  - This is the most generic setting.

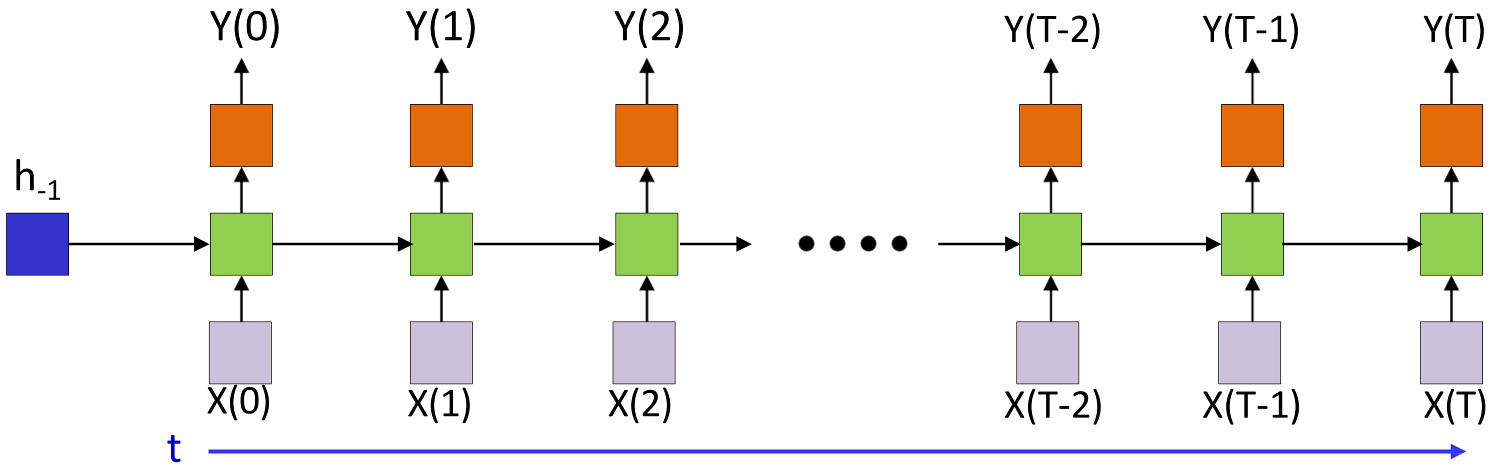
# Training the RNN




- The “unrolled” computation is just a giant shared-parameter neural network
  – All columns are identical and share parameters

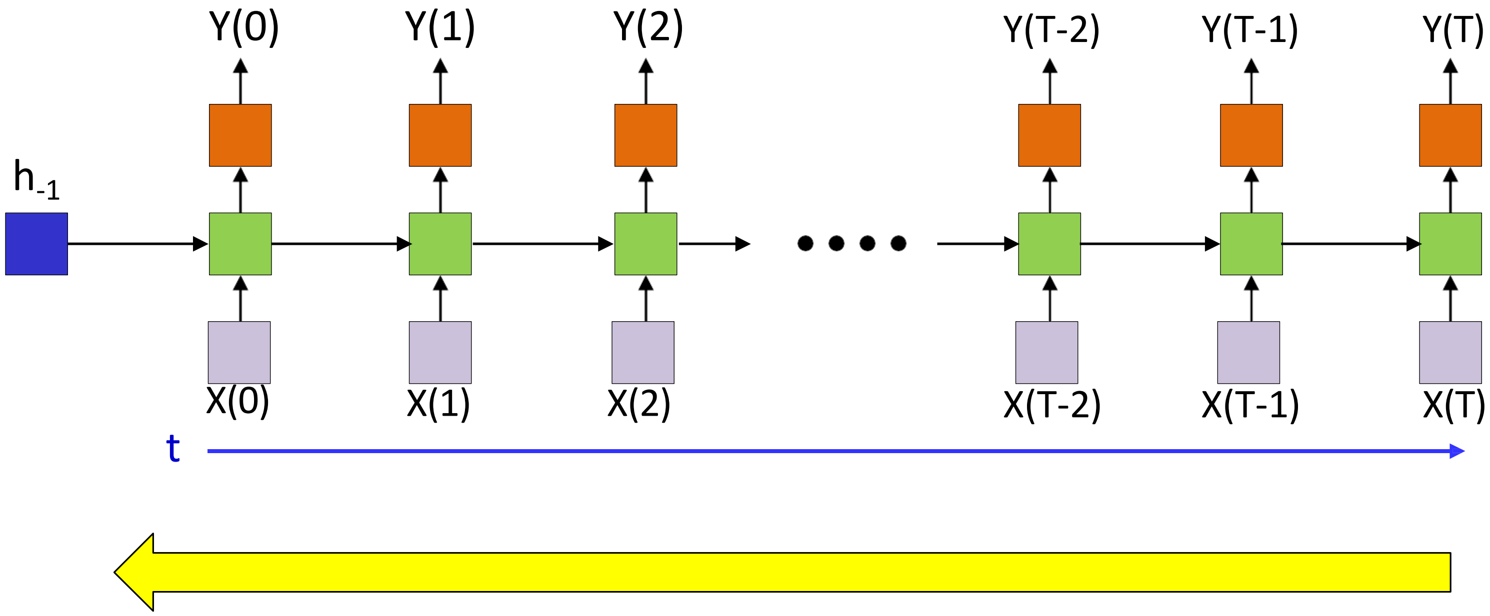
- Network parameters can be trained via gradient-descent (or its variants) using shared-parameter gradient descent rules
  – Gradient computation requires a forward pass, back propagation, and pooling of gradients (for parameter sharing)

# Training: Forward pass



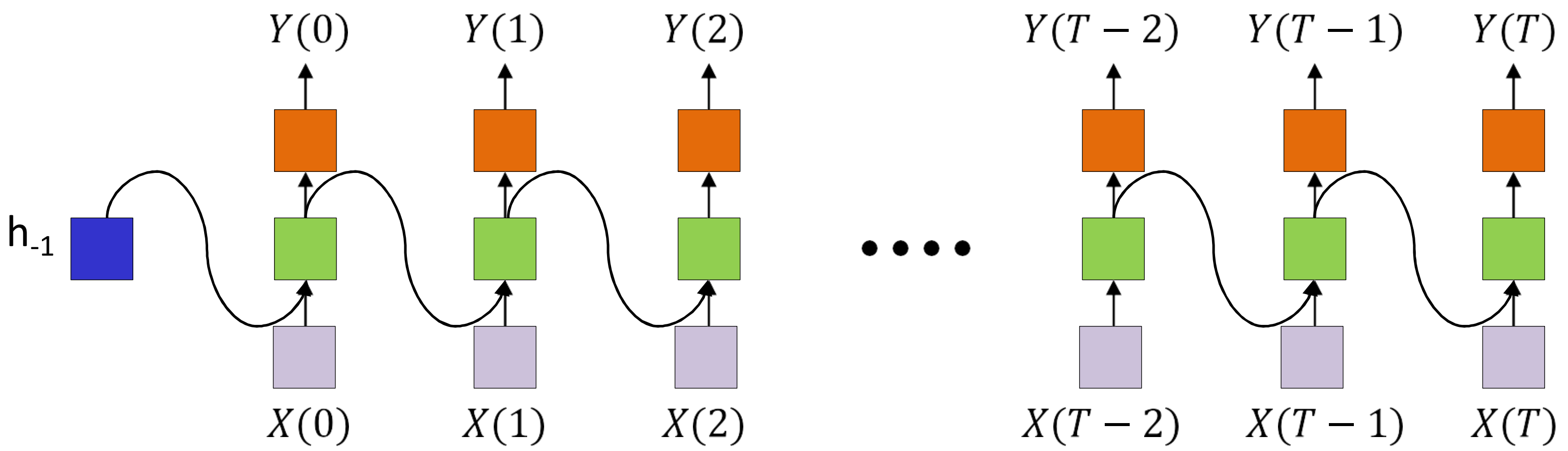

- For each training input:
- Forward pass: pass the entire data sequence through the network, generate outputs

# Training: Computing gradients




- For each training input:
- Backward pass: Compute gradients via backpropagation
  - *Back Propagation Through Time*

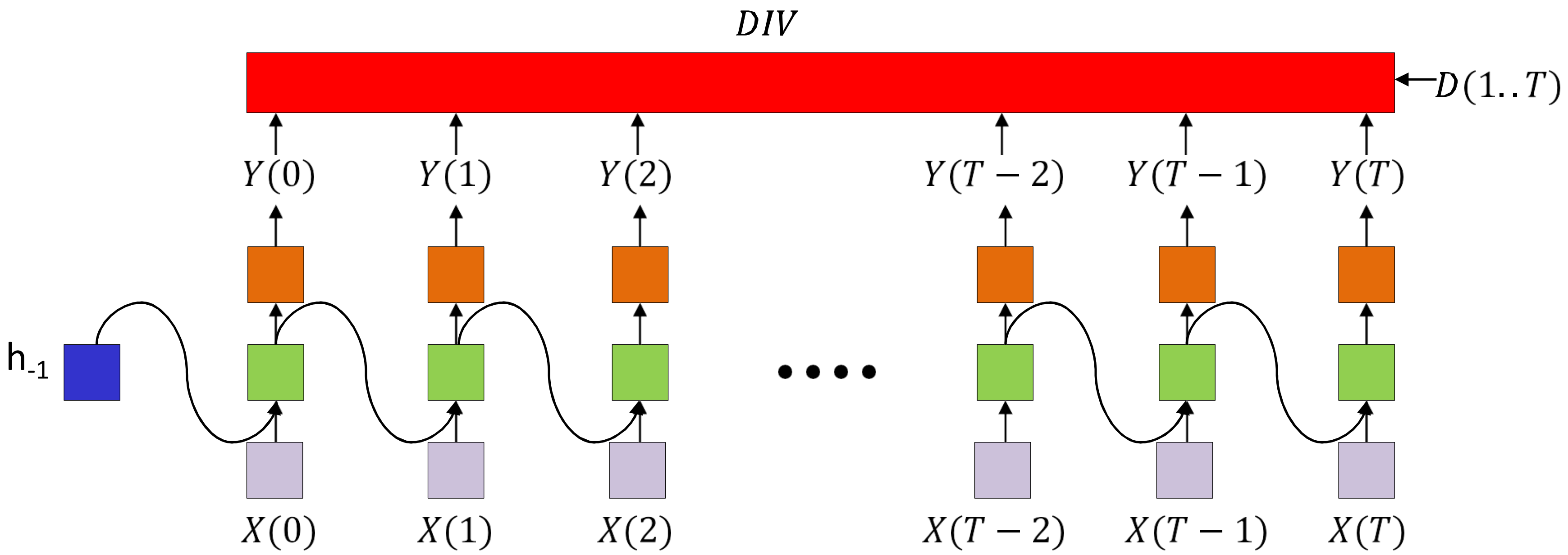
# Back Propagation Through Time




Will only focus on *one* training instance

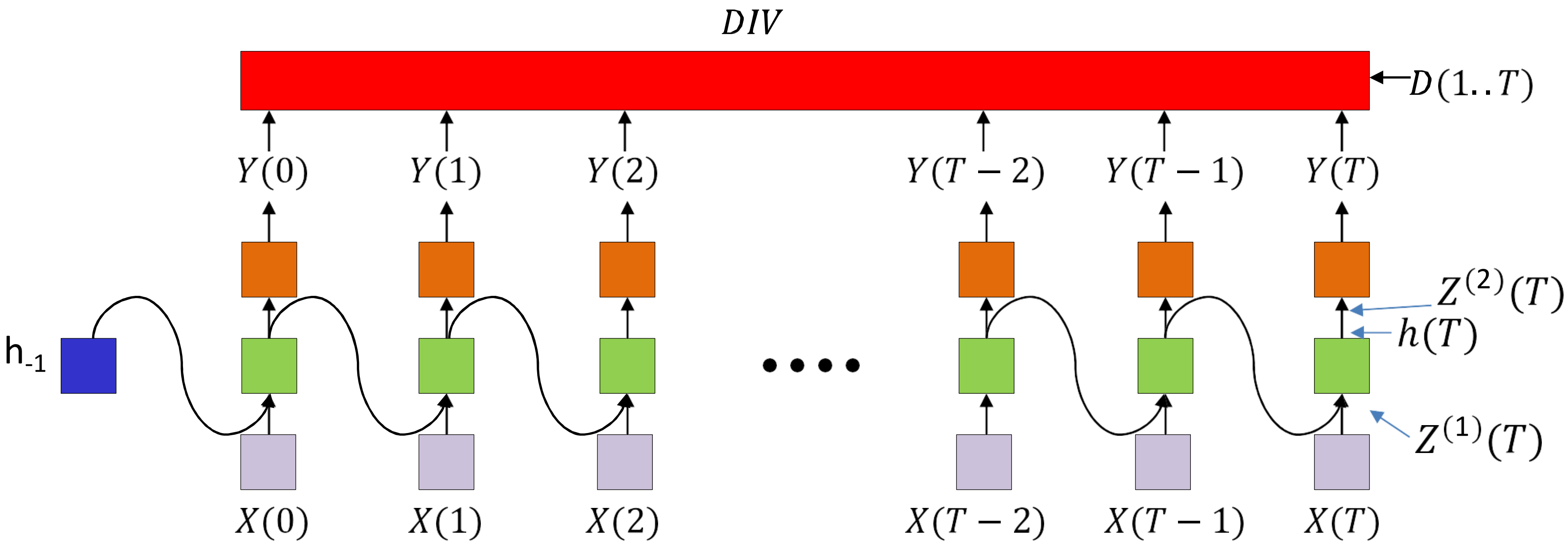
All subscripts represent *components* and not training instance index

# Back Propagation Through Time



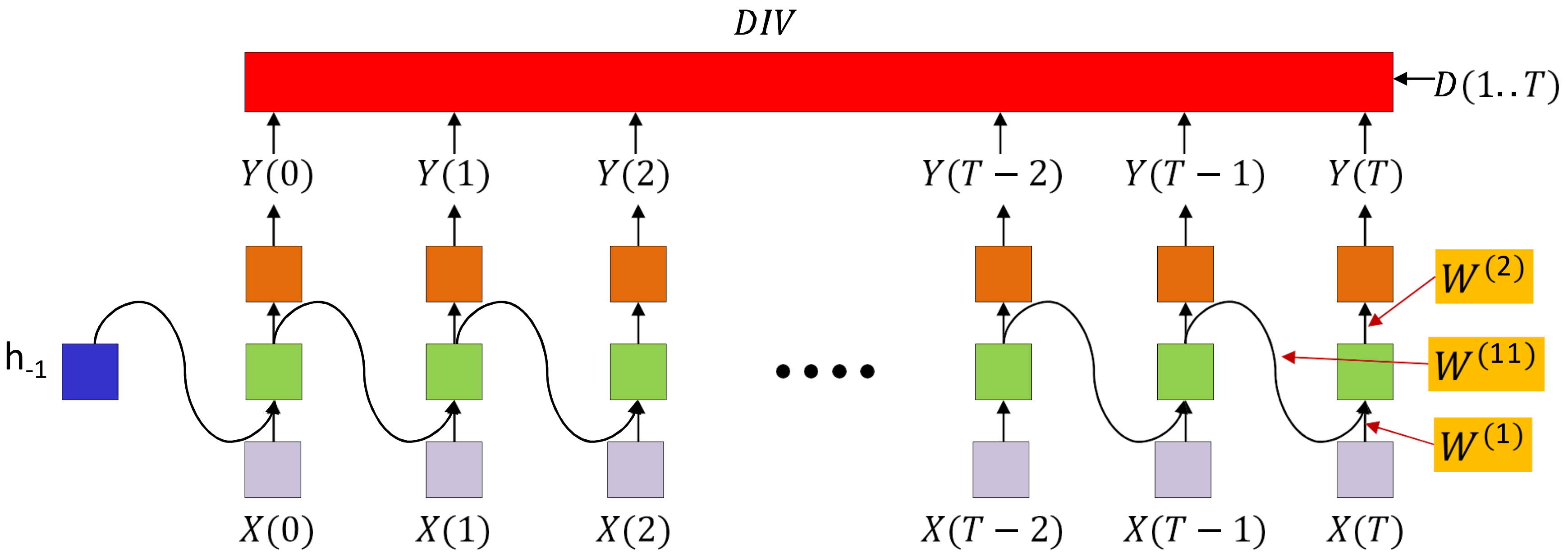

- The divergence computed is between the *sequence of outputs* by the network and the *desired sequence of outputs*
  - DIV is a scalar function of a series of vectors!
- This is *not* just the sum of the divergences at individual times
  - Unless we explicitly define it that way

# Notation



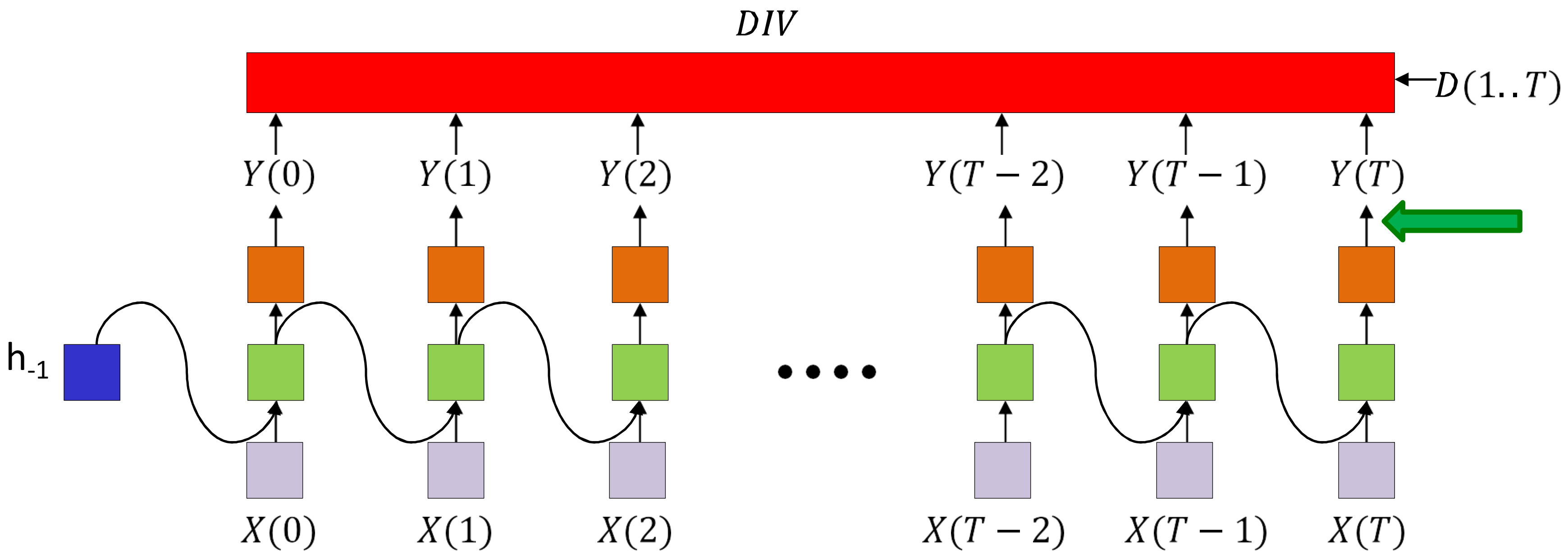

- $Y(t)$ is the output at time $t$
  - $Y_i(t)$ is the ith output
- $Z^{(2)}(t)$ is the pre-activation value of the neurons at the output layer at time t
- $h(t)$ is the output of the hidden layer at time $t$
  - Assuming only one hidden layer in this example
- $Z^{(1)}(t)$ is the pre-activation value of the hidden layer at time $t$

# Notation




- $W^{(1)} = \left[w_{ij}^{(1)}\right]$ is the matrix of *current* weights from the input to the hidden layer.
- $W^{(2)} = \left[w_{ij}^{(2)}\right]$ is the matrix of *current* weights from the hidden layer to the output layer
- $W^{(11)} = \left[w_{ij}^{(11)}\right]$ is the matrix of *recurrent* weights from the hidden layer to itself

# Back Propagation Through Time

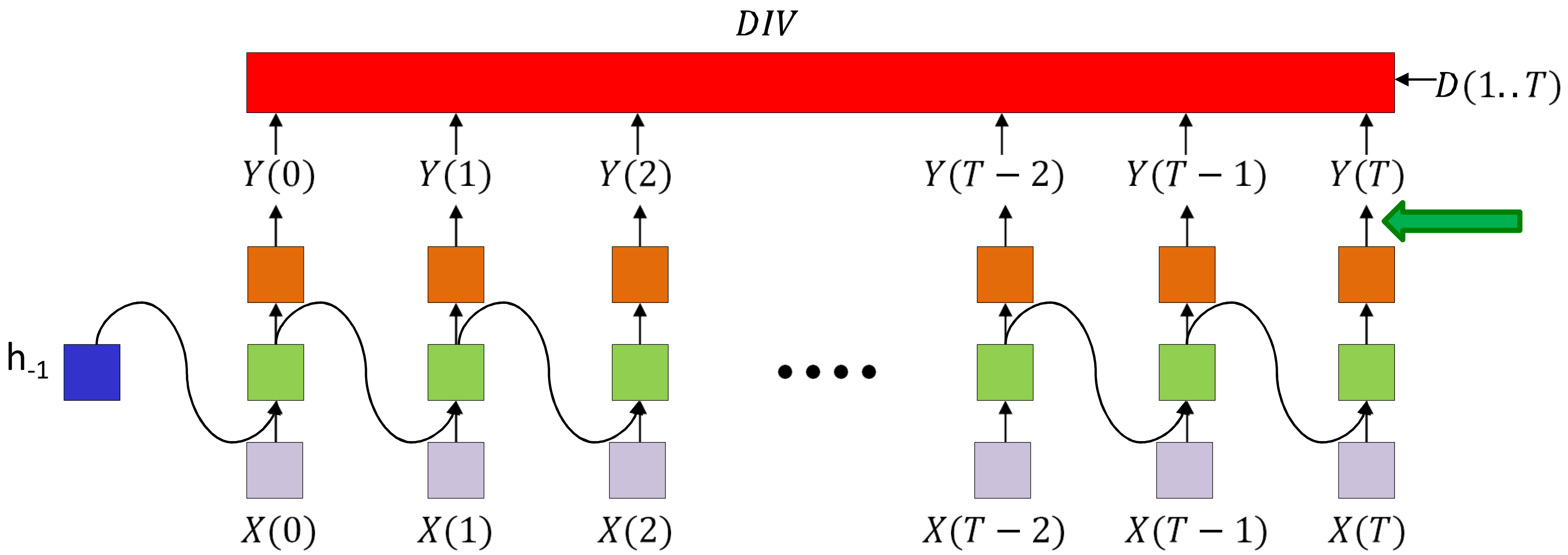



First step of backprop: Compute $\nabla_{Y(T)} DIV$ (Compute $\frac{dDIV}{dY_i(T)}$ for all $i$)

Note: DIV is a function of *all* outputs Y(0) … Y(T)

In general we will be required to compute $\frac{dDIV}{dY_i(t)}$ for all $i$ and $t$. This can be a source of significant difficulty in many scenarios.

# Back Propagation Through Time

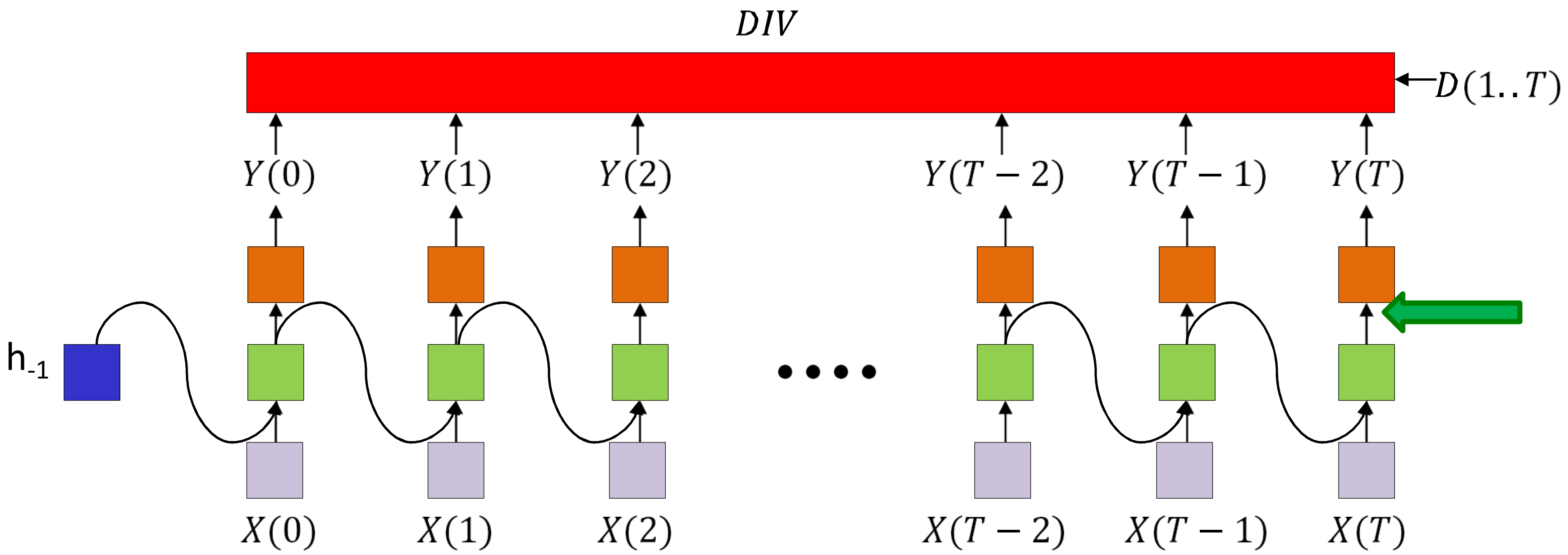



Special case, when the overall divergence is a simple sum of local divergences at each time: $DIV = \sum_t Div(t)$

Will get $\nabla_{Y(t)} Div(t)$

$$\frac{\partial DIV}{\partial Y_i(t)} = \frac{\partial Div(t)}{\partial Y_i(t)}$$

# Back Propagation Through Time

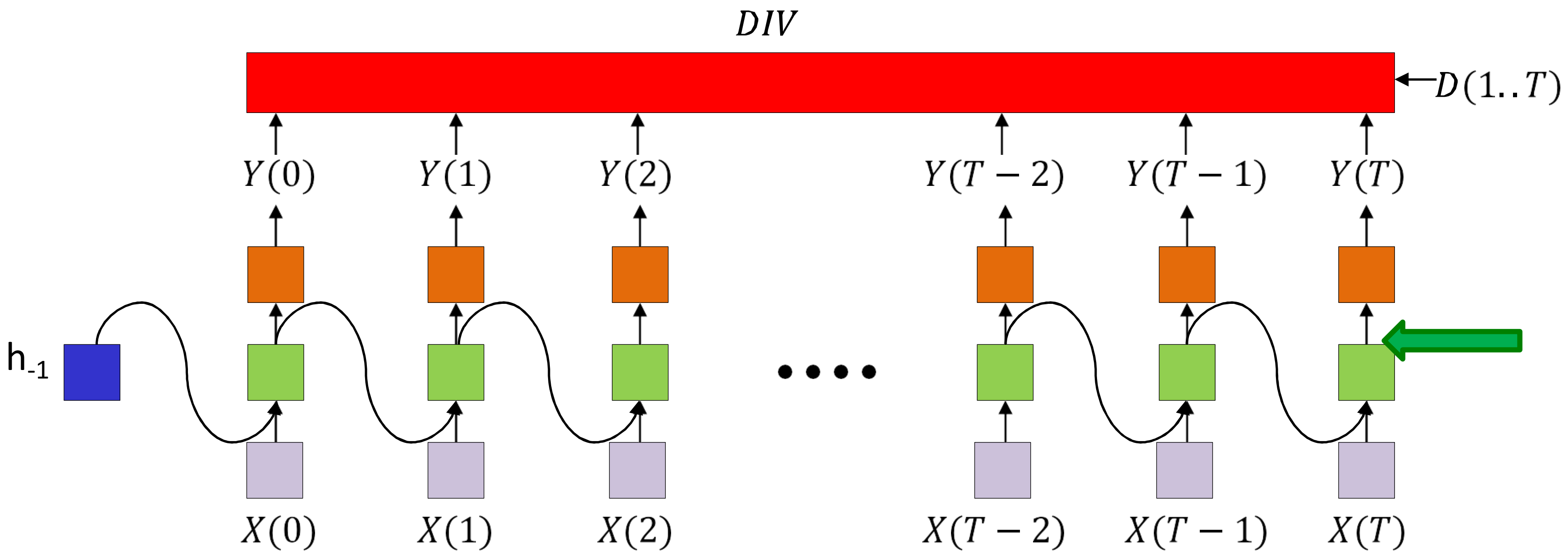



$$\nabla_{Z^{(2)}(T)} DIV = \nabla_{Y(T)} DIV \nabla_{Z^{(2)}(T)} Y(T)$$

Vector output activation

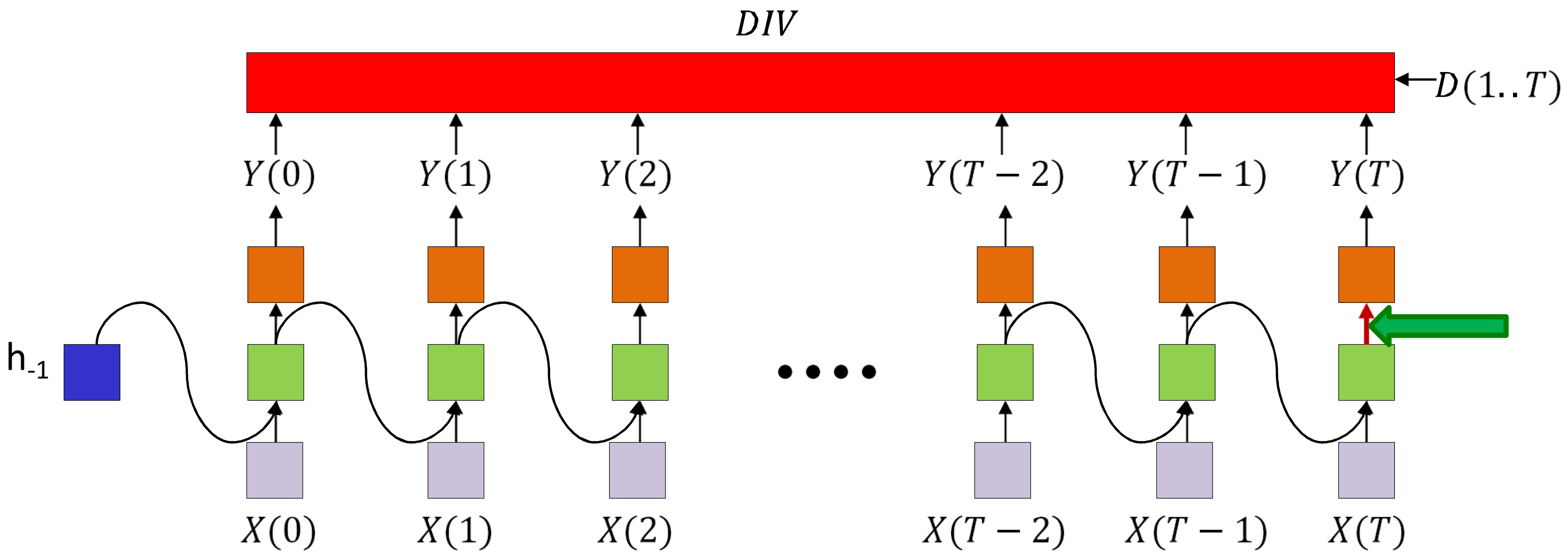
$$\frac{dDIV}{dZ_i^{(2)}(T)} = \frac{dDIV}{dY_i(T)} \frac{dY_i(T)}{dZ_i^{(2)}(T)} \quad \text{OR} \quad \frac{dDIV}{dZ_i^{(2)}(T)} = \sum_j \frac{dDIV}{dY_j(T)} \frac{dY_j(T)}{dZ_i^{(2)}(T)}$$

# Back Propagation Through Time



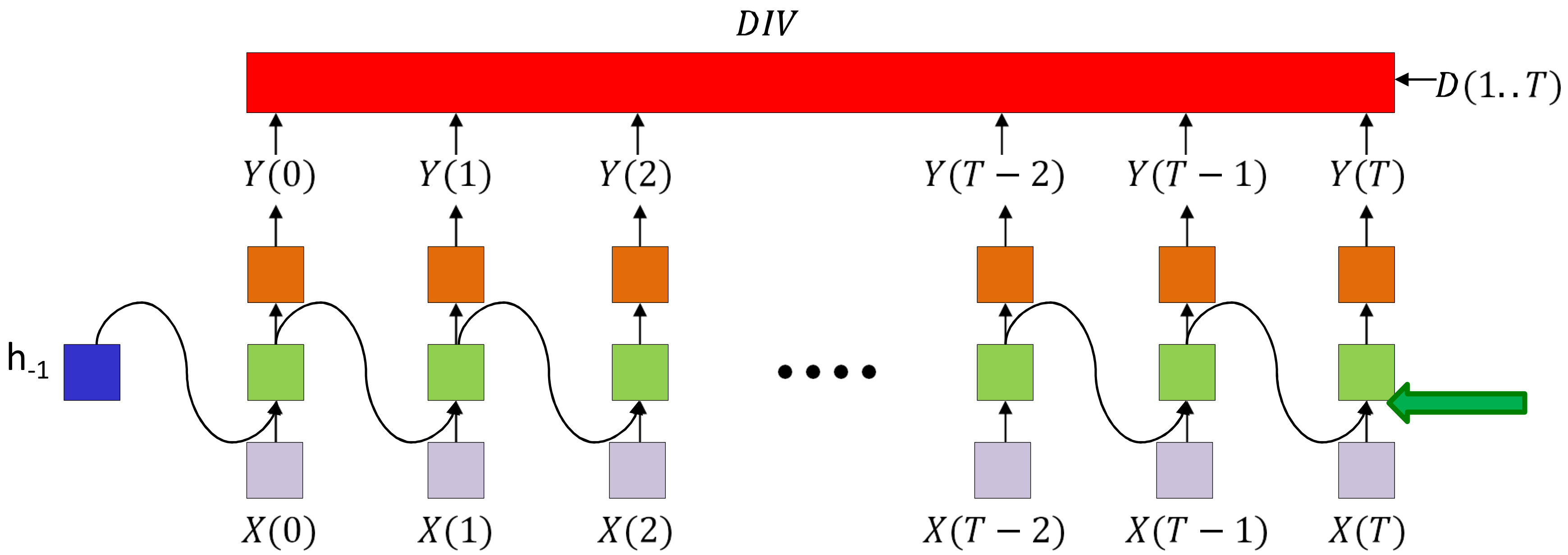

$$\frac{dDIV}{dh_i(T)} = \sum_j \frac{dDIV}{dZ_j^{(2)}(T)} \frac{dZ_j^{(2)}(T)}{dh_i(T)} = \sum_j w_{ij}^{(2)} \frac{dDIV}{dZ_j^{(2)}(T)}$$

# Back Propagation Through Time



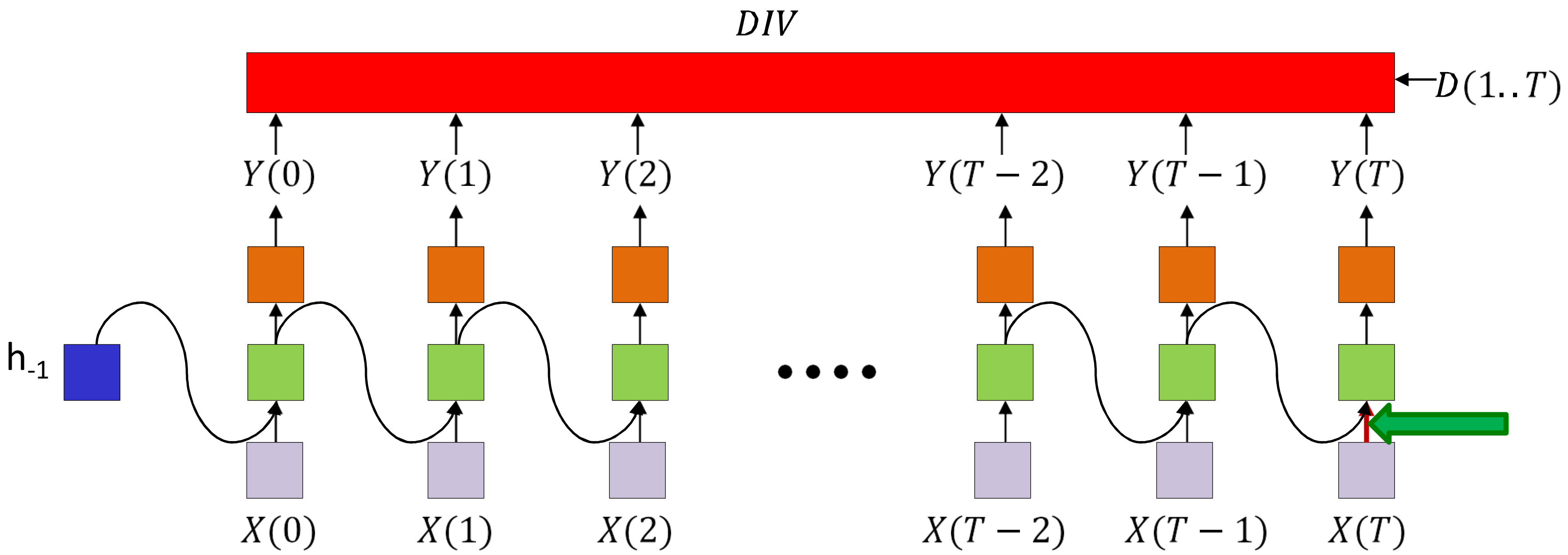

$$\frac{dDIV}{dw_{ij}^{(2)}} = \frac{dDIV}{dZ_j^{(2)}(T)} h_i(T)$$

# Back Propagation Through Time



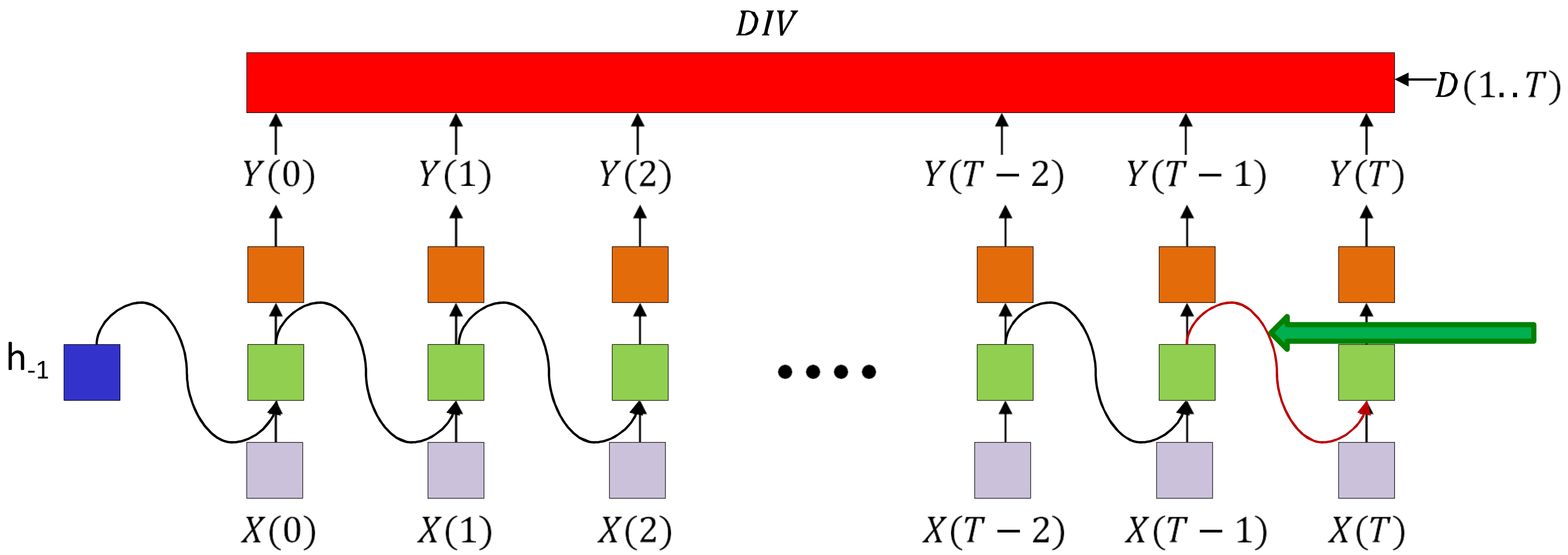

$$\frac{dDIV}{dZ_i^{(1)}(T)} = \frac{dDIV}{dh_i(T)} \frac{dh_i(T)}{dZ_i^{(1)}(T)}$$

# Back Propagation Through Time



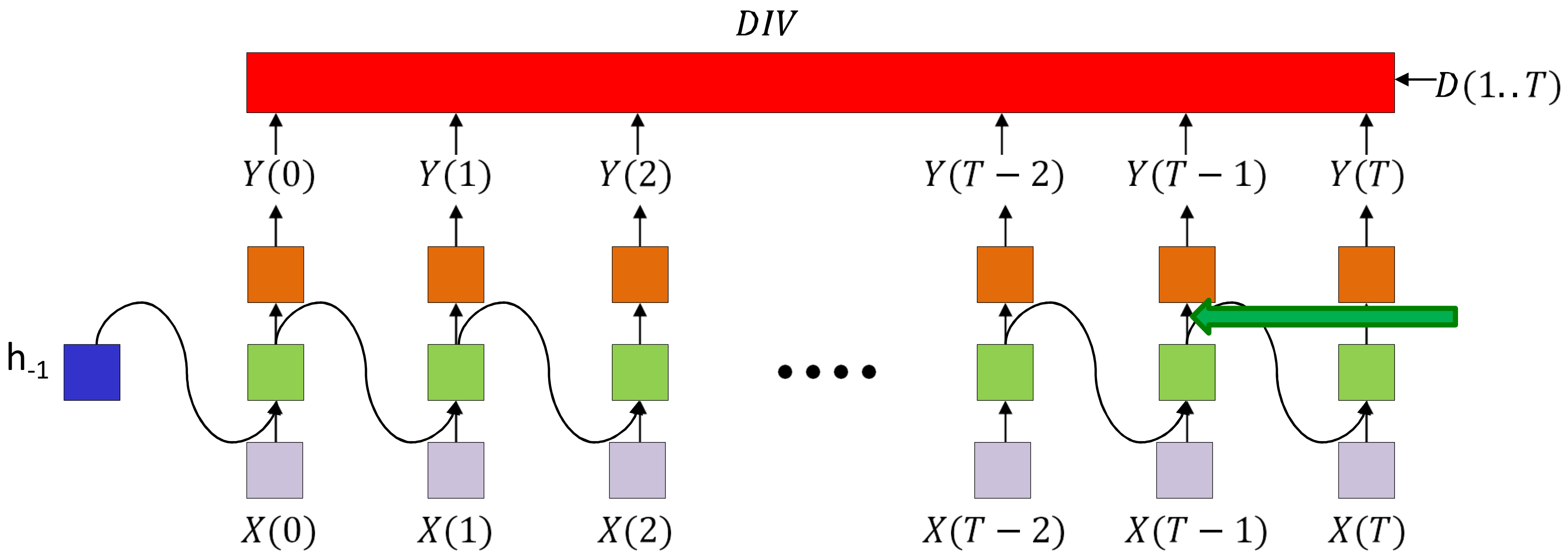

$$\frac{dDIV}{dw_{ij}^{(1)}} = \frac{dDIV}{dZ_j^{(1)}(T)} X_i(T)$$

# Back Propagation Through Time




$$\frac{dDIV}{dw_{ij}^{(11)}} = \frac{dDIV}{dZ_j^{(1)}(T)} h_i(T-1)$$

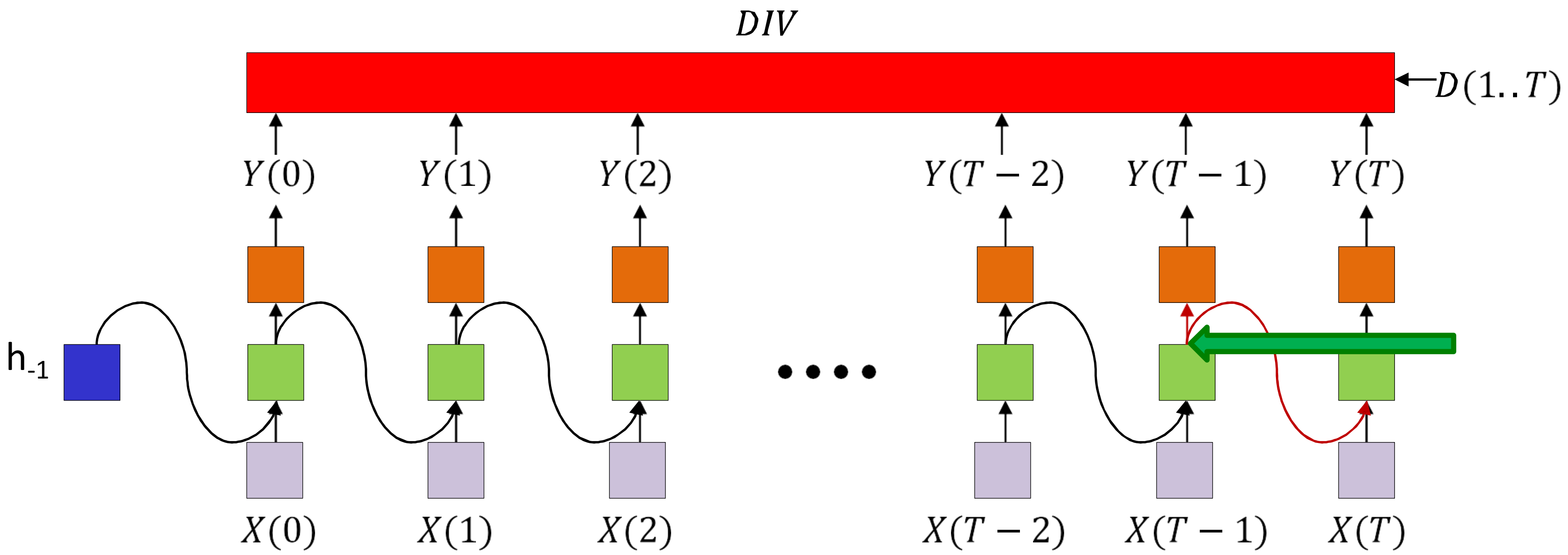
# Back Propagation Through Time




Vector output activation

$$\frac{dDIV}{dZ_i^{(2)}(T-1)} = \frac{dDIV}{dY_i(T-1)}\frac{dY_i(T-1)}{dZ_i^{(2)}(T-1)} \quad \text{OR} \quad \frac{dDIV}{dZ_i^{(2)}(T-1)} = \sum_j \frac{dDIV}{dY_j(T-1)}\frac{dY_j(T-1)}{dZ_i^{(2)}(T-1)}$$

# Back Propagation Through Time

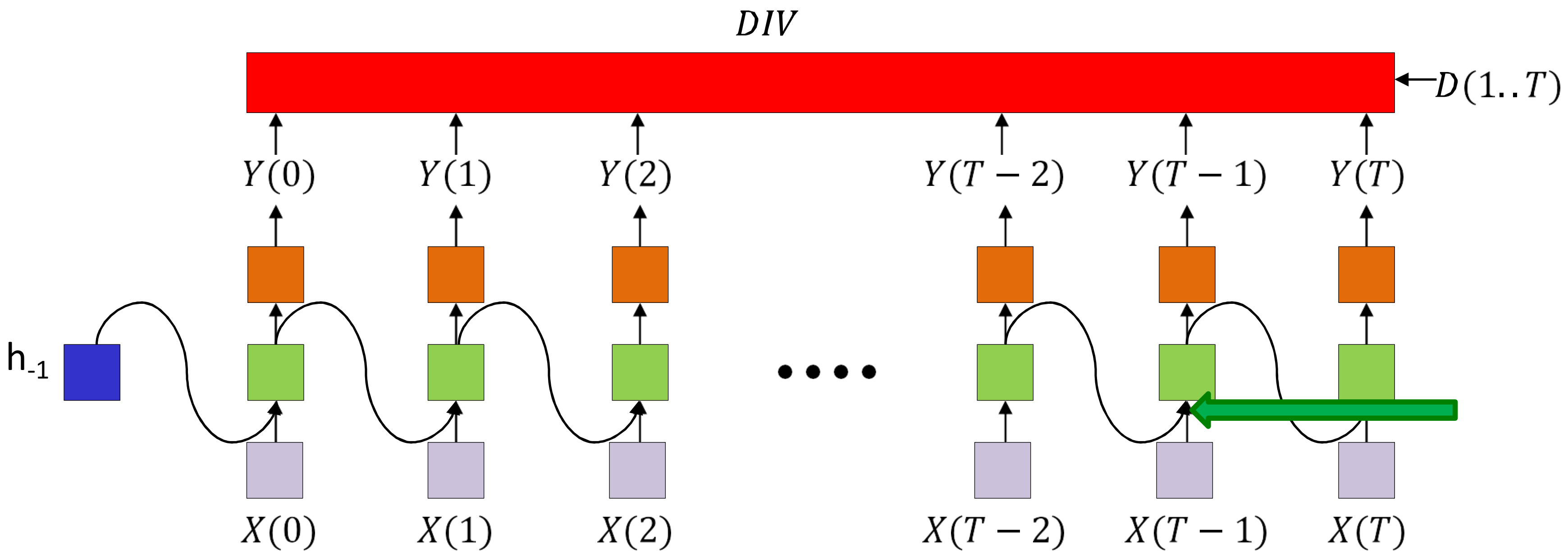



$$\frac{dDIV}{dh_i(T-1)} = \sum_j w_{ij}^{(2)} \frac{dDIV}{dZ_j^{(2)}(T-1)} + \sum_j w_{ij}^{(11)} \frac{dDIV}{dZ_j^{(1)}(T)}$$

Note the addition $\Rightarrow$ $$\frac{dDIV}{dw_{ij}^{(2)}} += \frac{dDIV}{dZ_j^{(2)}(T-1)} h_i(T-1)$$

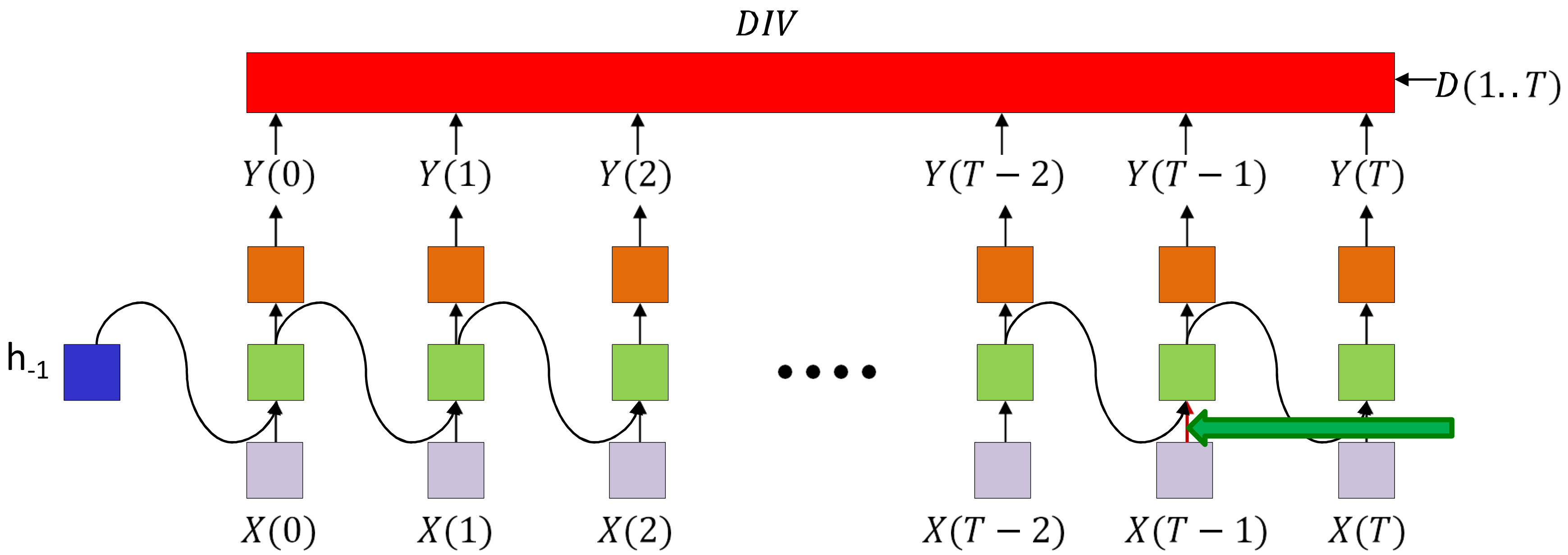
Note the addition $\Rightarrow$ $$\nabla_{W^{(2)}} DIV += h(T-1)\nabla_{Z^{(2)}(T-1)} DIV$$

# Back Propagation Through Time




$$\frac{dDIV}{dZ_i^{(1)}(T-1)} = \frac{dDIV}{dh_i(T-1)} \frac{dh_i(T-1)}{dZ_i^{(1)}(T-1)}$$

# Back Propagation Through Time

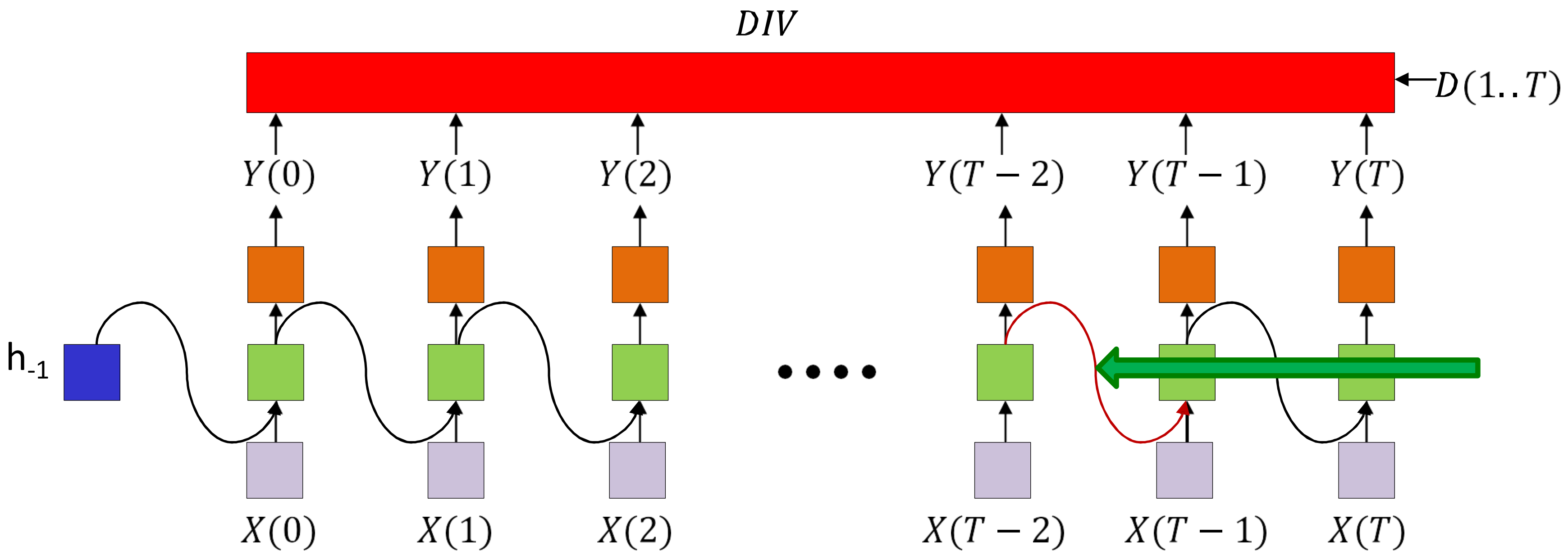



$$\frac{dDIV}{dZ_i^{(1)}(T-1)} = \frac{dDIV}{dh_i(T-1)} \frac{dh_i(T-1)}{dZ_i^{(1)}(T-1)}$$

$$\frac{dDIV}{dw_{ij}^{(1)}} += \frac{dDIV}{dZ_j^{(1)}(T-1)} X_i(T-1)$$

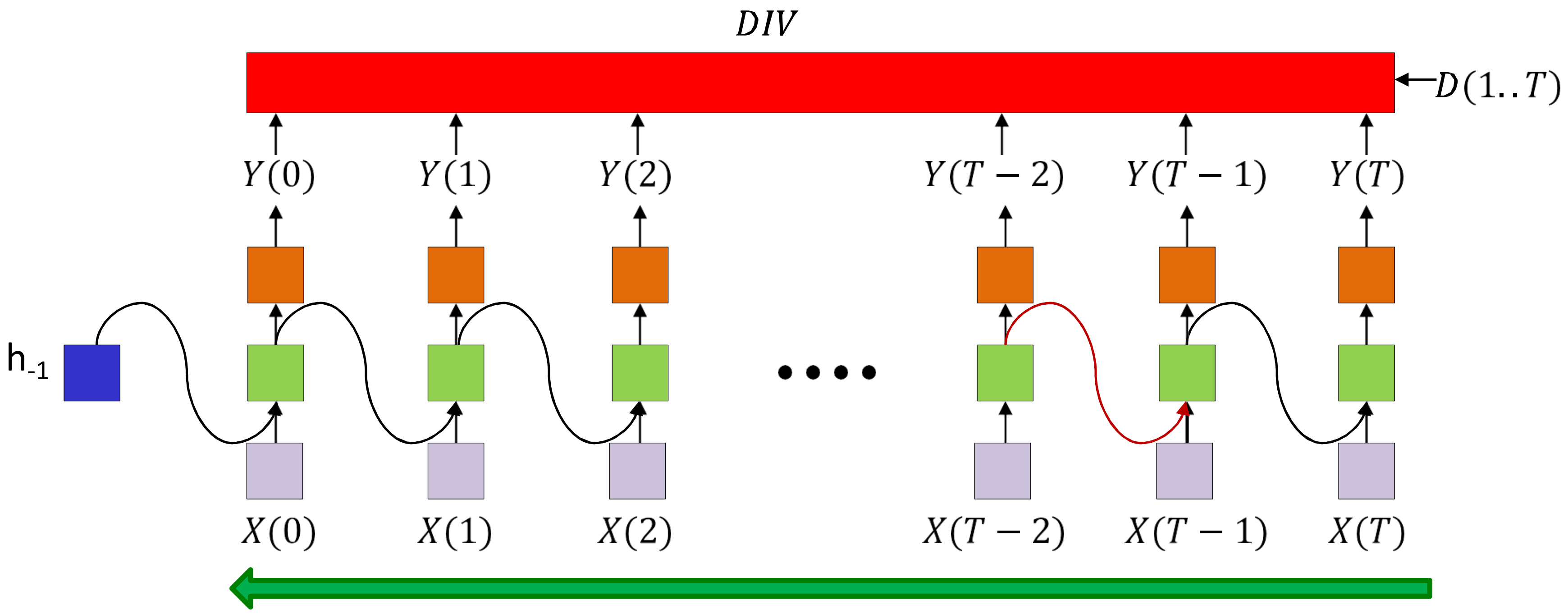
Note the addition

# Back Propagation Through Time




Note the addition $\Rightarrow$ $$\frac{dDIV}{dw_{ij}^{(11)}} += \frac{dDIV}{dZ_j^{(1)}(T-1)} h_i(T-2)$$

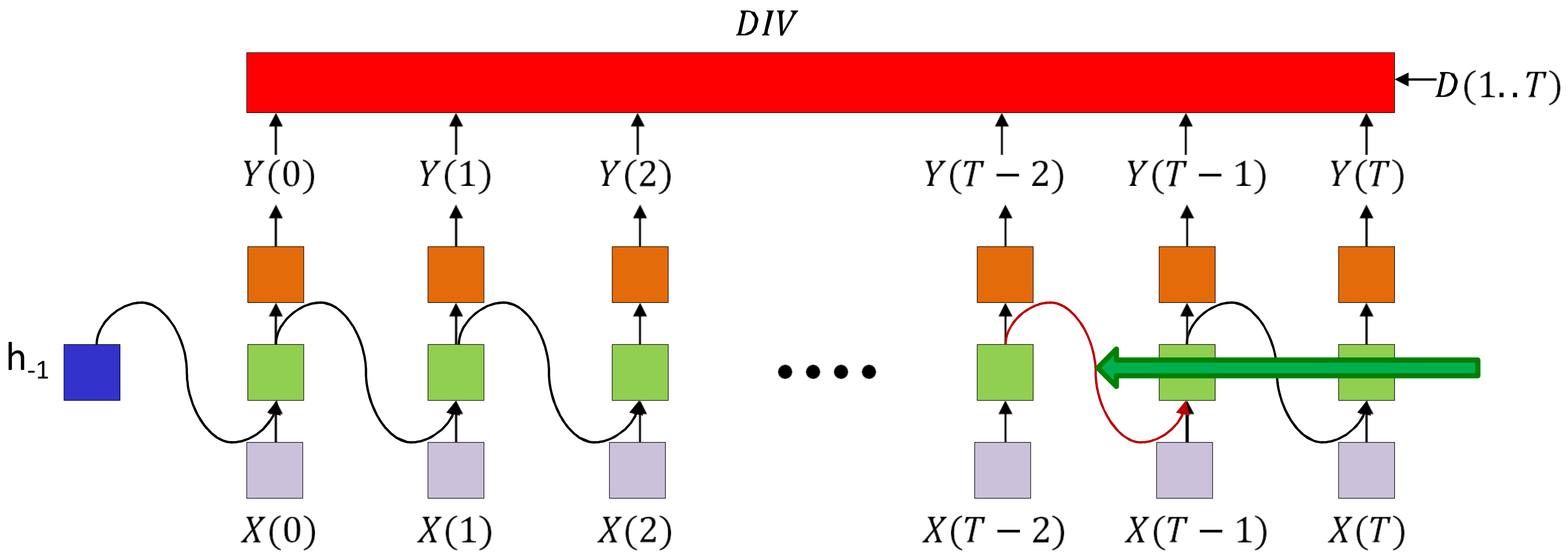
# Back Propagation Through Time




Continue computing derivatives going backward through time until..

$$\frac{dDIV}{dh_i(-1)} = \sum_j w_{ij}^{(11)} \frac{dDIV}{dZ_j^{(1)}(0)}$$

# Back Propagation Through Time




Initialize all derivatives to 0

For t = T downto 0

$$\nabla_{Z^{(2)}(t)} DIV = \nabla_{F(t)} DIV \, \nabla_{Z^{(2)}(t)} Y(t)$$

$$\nabla_{h(t)} DIV = \nabla_{Z^{(2)}(t)} DIV \, W^{(2)} + \nabla_{Z^{(1)}(t+1)} DIV \, W^{(11)}$$

$$\nabla_{Z^{(1)}(t)} DIV = \nabla_{h(t)} DIV \, \nabla_{Z^{(1)}(t)} h(t)$$

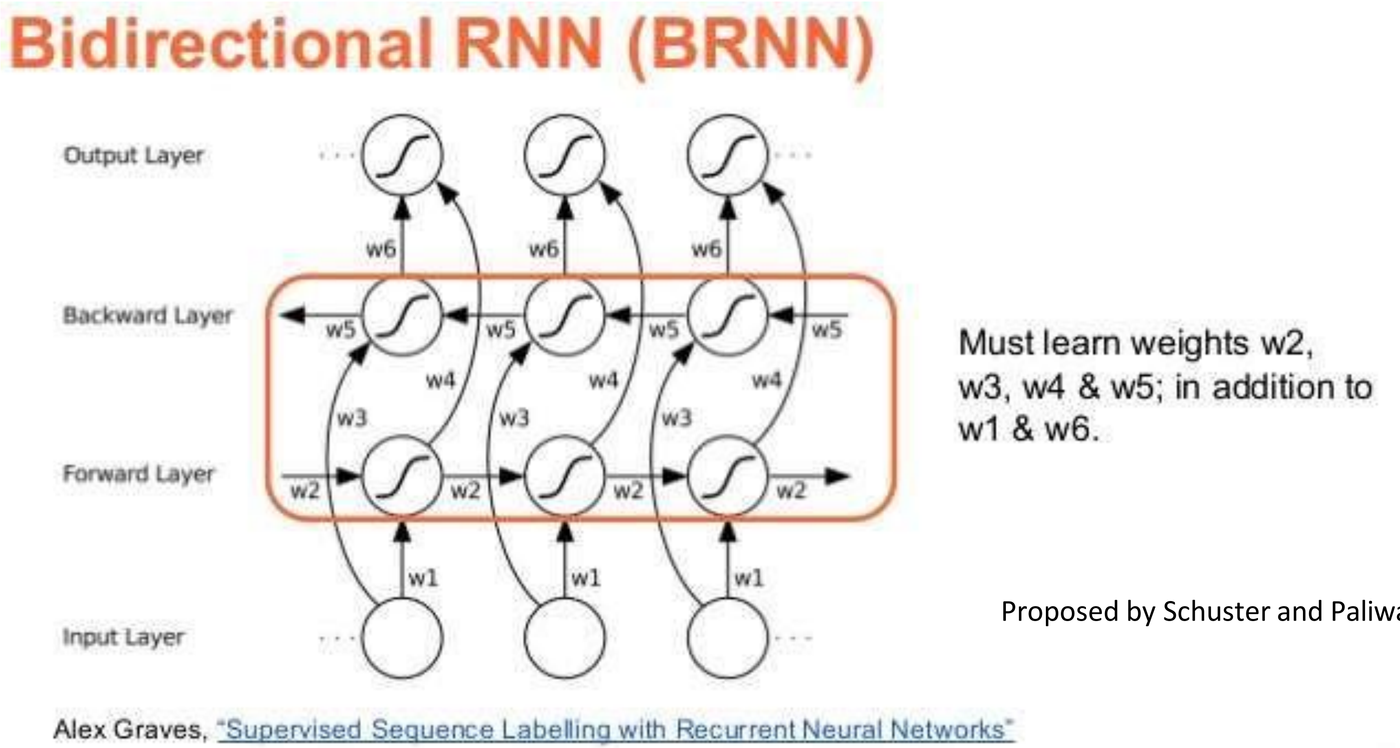
$$\nabla_{W^{(2)}} DIV \mathrel{+}= h(t) \nabla_{Z^{(2)}(t)} DIV$$

$$\nabla_{W^{(11)}} DIV \mathrel{+}= h(t-1) \nabla_{Z^{(1)}(t)} DIV$$

$$\nabla_{W^{(1)}} DIV \mathrel{+}= X(t) \nabla_{Z^{(1)}(t)} DIV$$

$$\nabla_{h_{-1}} DIV = \nabla_{Z^{(1)}(0)} DIV W^{(11)}$$

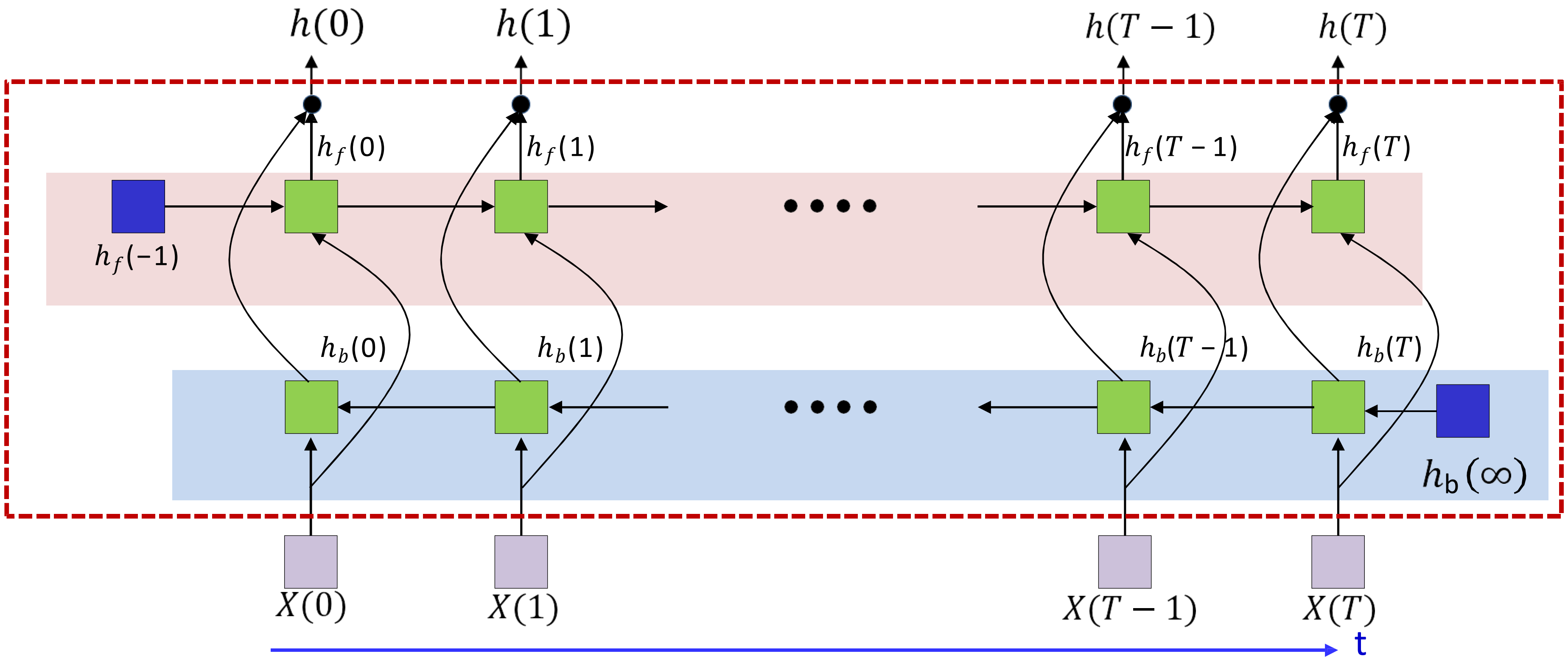
# Extensions to the RNN: *Bidirectional RNN*




Proposed by Schuster and Paliwal 1997

- In problems where the entire input sequence is available, RNNs can be bidirectional
- RNN with both forward and backward recursion
- Explicitly models the fact that just as the future can be predicted from the past, the past can be deduced from the future

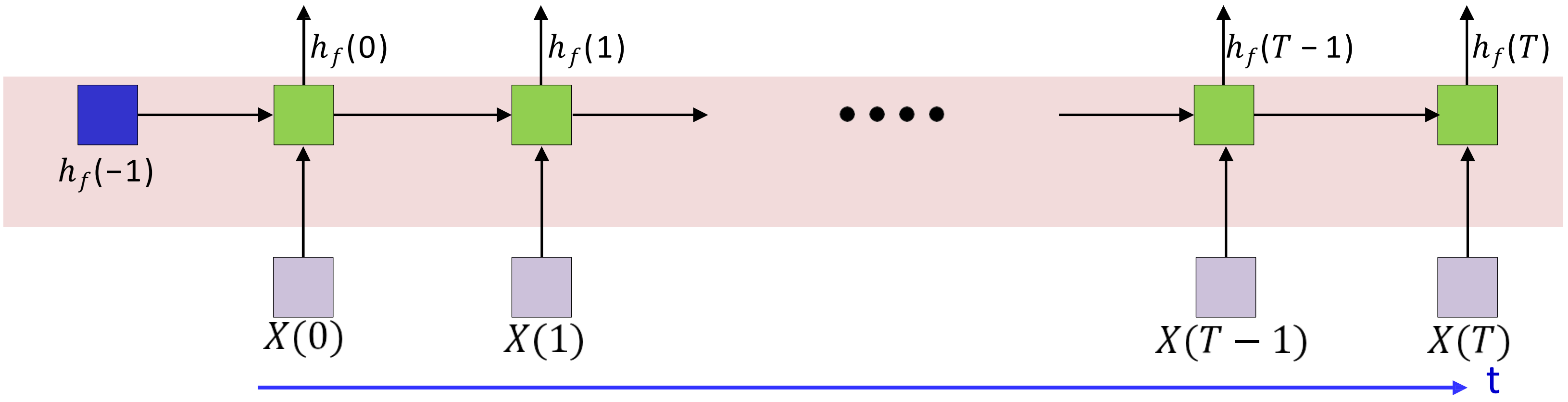
# Bidirectional RNN




- "Block" performs bidirectional inference on input
  - "Input" could be input series X(0)…X(T) or the output of a previous layer (or block)

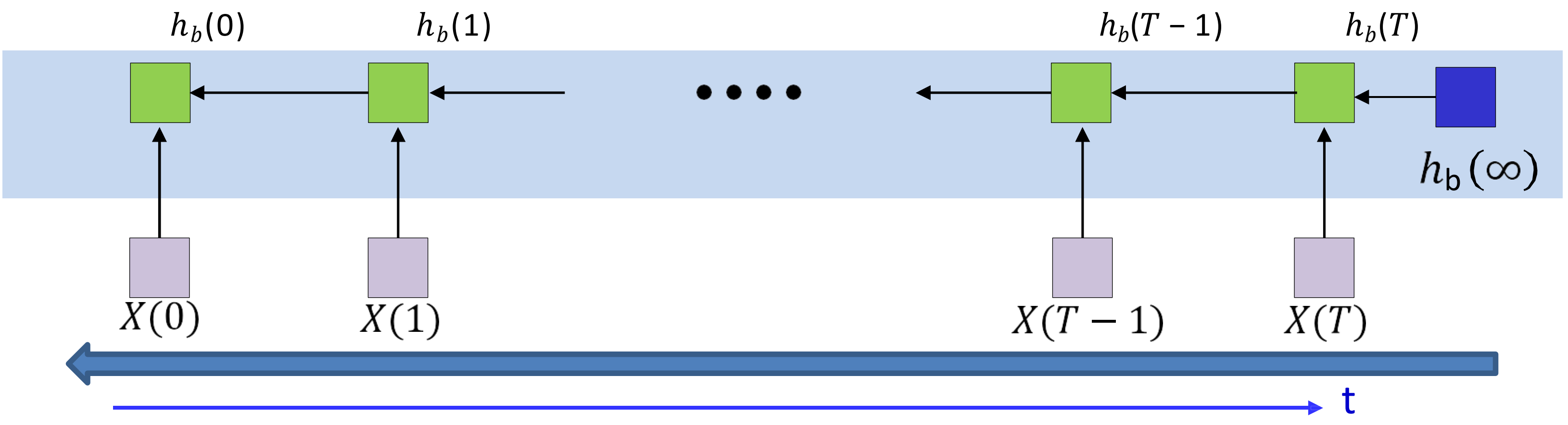
- The Block has two components
  - A forward net process the data from t=0 to t=T
  - A backward net processes it backward from t=T down to t=0

# Bidirectional RNN block



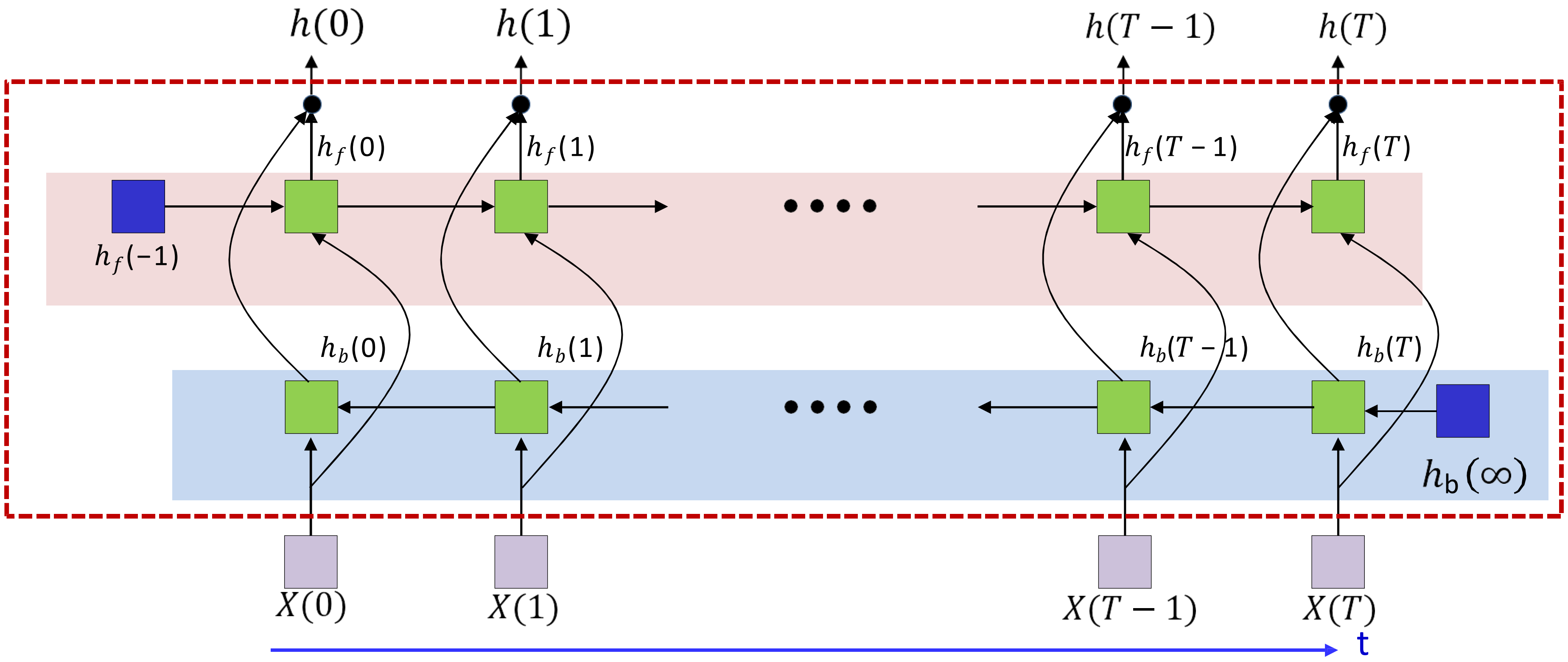

- The forward net process the data from t=0 to t=T
  - Only computing the hidden state values.

# Bidirectional RNN block




- The backward nets processes the input data in *reverse time,* end to beginning
  - Initially only the hidden state values are computed
    - Clearly, this is not an online process and requires the *entire* input data
  - Note: *This is not the backward pass of backprop.*

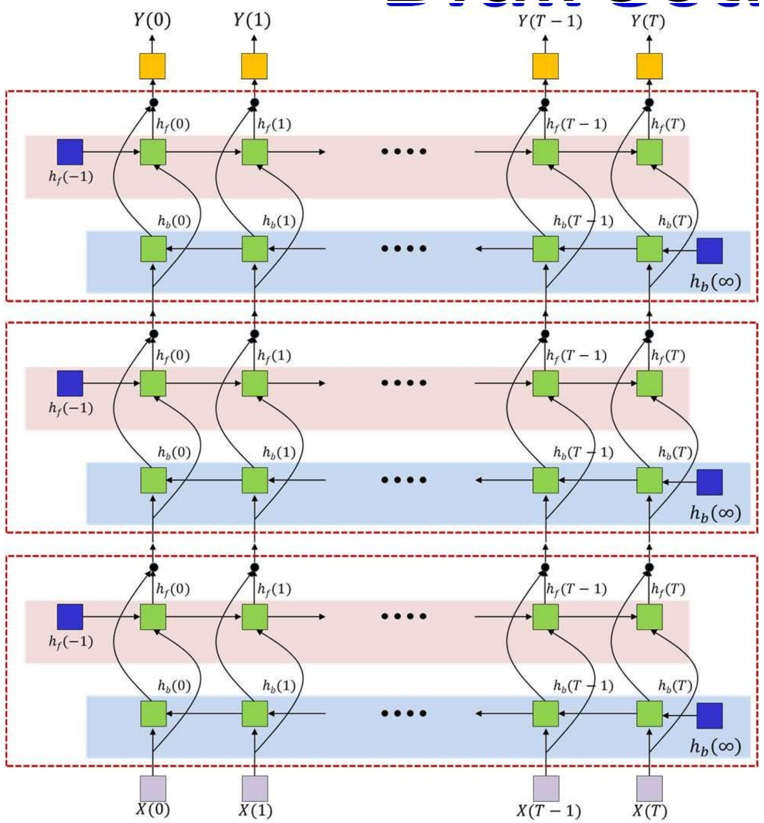
# Bidirectional RNN block

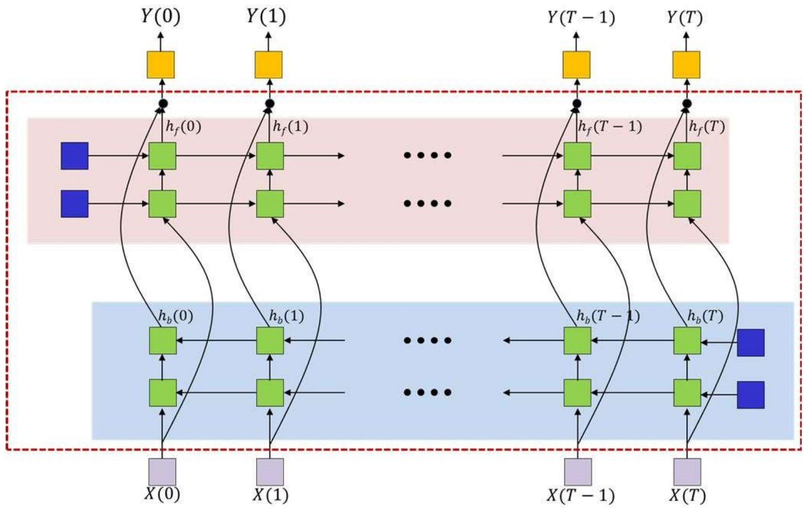



- The computed states of both networks are combined to give you the output of the bidirectional block
  - Typically just concatenate them

$$h(t) = [h_f(t); h_b(t)]$$

# Bidirectional RNN




- Actual network may be formed by stacking many independent bidirectional blocks followed by an output layer
  - Forward and backward nets in each block are a single layer
- Or by a single bidirectional block followed by an output layer
  - The forward and backward nets may have several layers
- In either case, it’s sufficient to understand forward inference and backprop rules for a single block
  - Full forward or backprop computation simply requires repeated application of these rules