Deep Learning
Recurrent Networks : 1
Fall 2024

Instructor: Bhiksha Raj

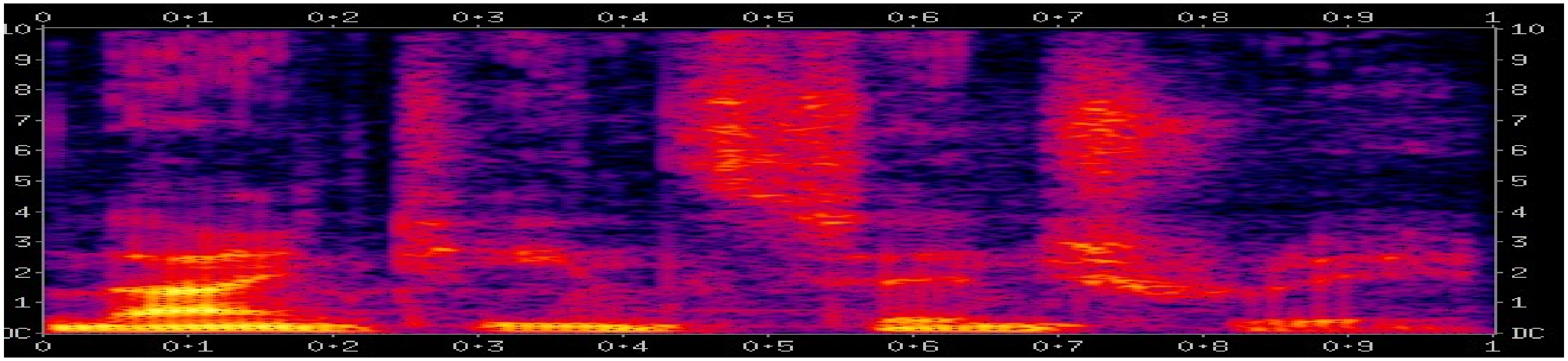
Attendance poll: 611

Modelling Series

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

What did I say?

“To be” or not “to be”??



- 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?

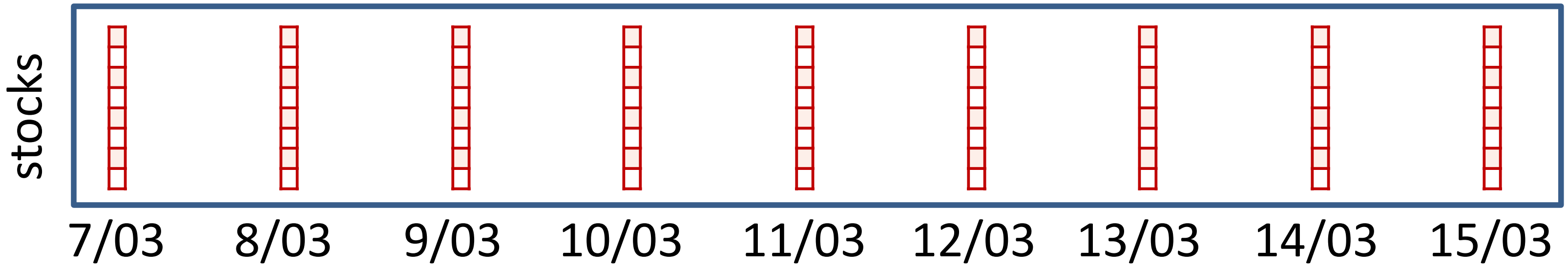
“Football” or “basketball”?

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.

- 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..

To invest or not to invest?



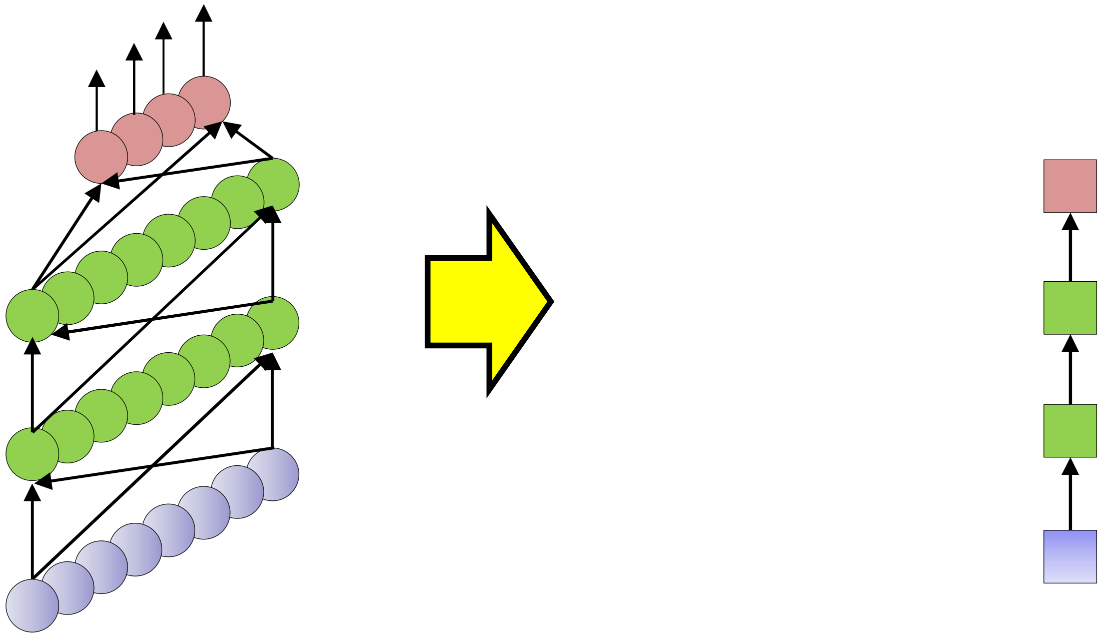
- Note: Inputs are sequences of vectors. Output may be scalar or vector
  - Should I invest, vs. should I not invest in X?
  - Decision must be taken considering how things have fared over time

These are classification and prediction problems

- Consider a sequence of inputs
  - Input vectors
- Produce one or more outputs

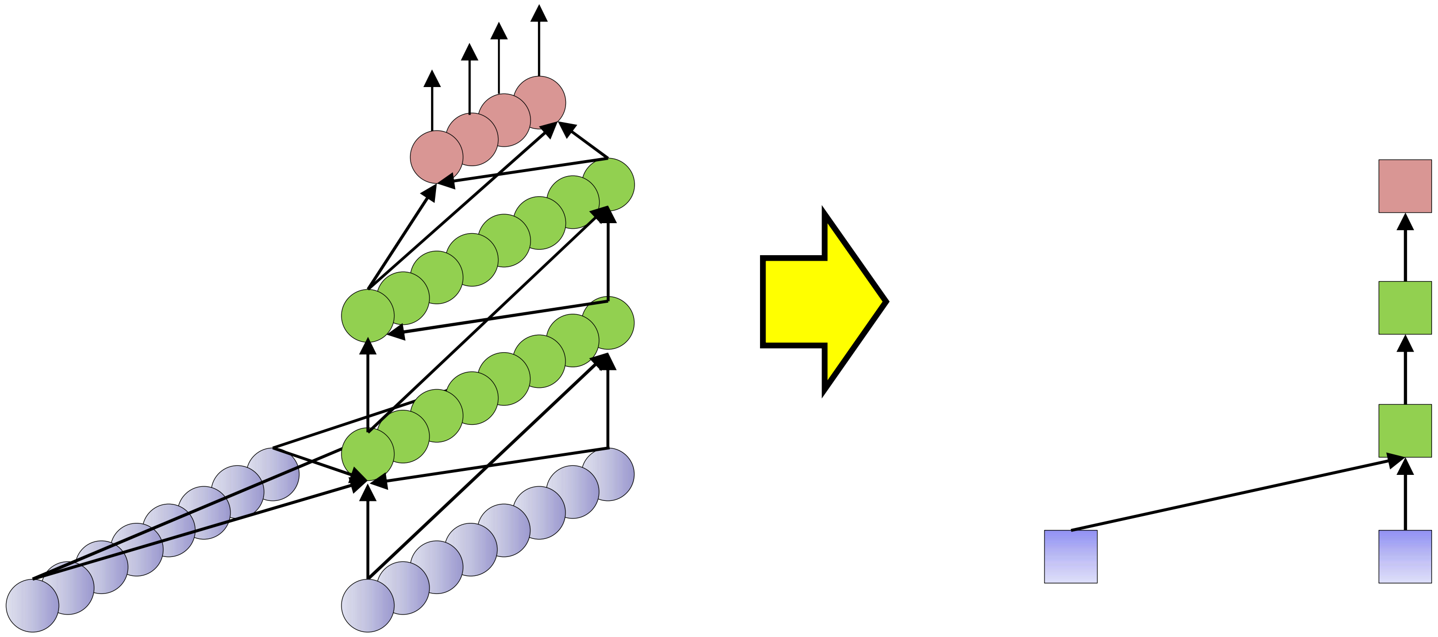
- This can be done with neural networks
  - Obviously

Representational shortcut



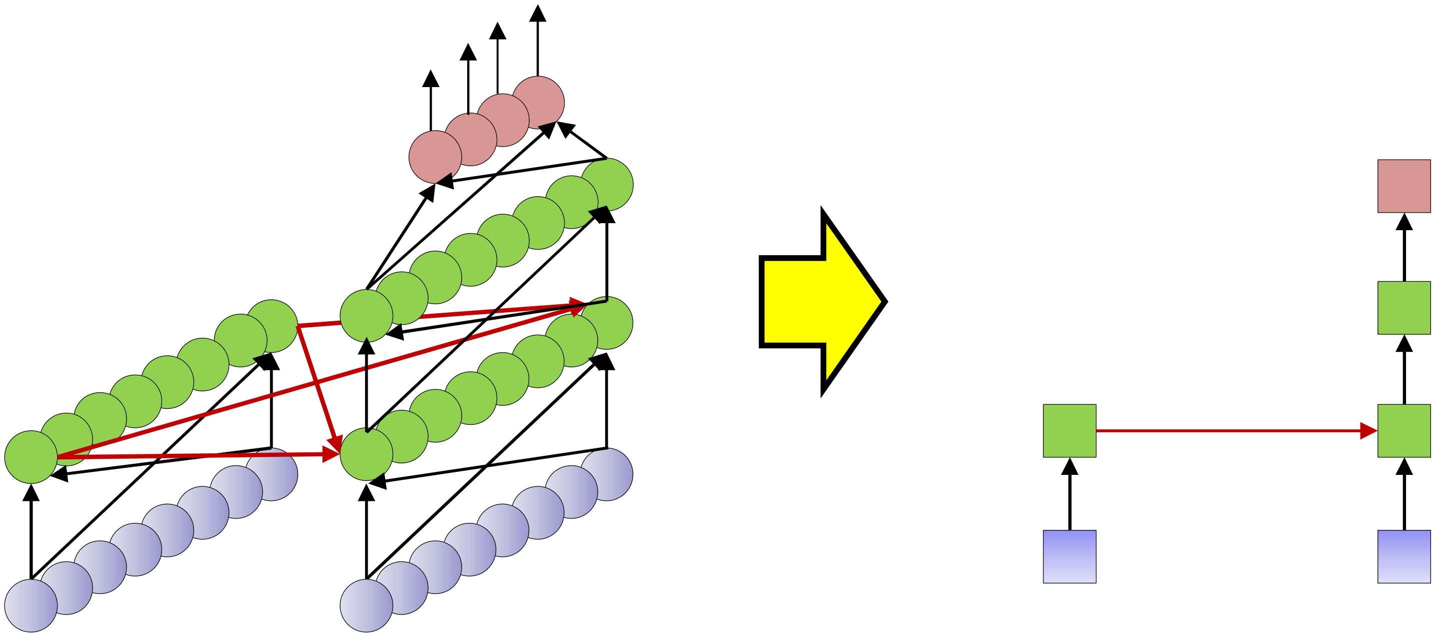
- Input at each time is a *vector*
- Each layer has many neurons
  - Output layer too may have many neurons
- But will represent everything by simple boxes
  - Each box actually represents an entire *layer with many units*

Representational shortcut



- Input at each time is a *vector*
- Each layer has many neurons
  - Output layer too may have many neurons
- But will represent everything by simple boxes
  - Each box actually represents an entire *layer with many units*

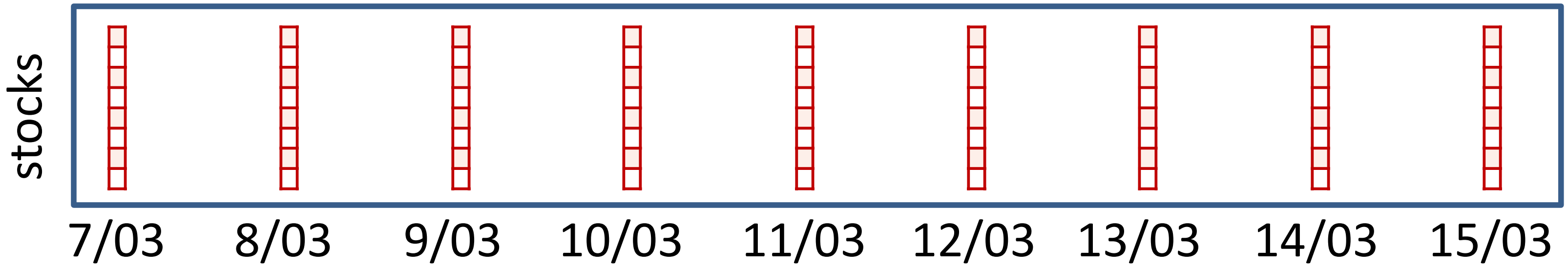
Representational shortcut



- Input at each time is a *vector*
- Each layer has many neurons
  - Output layer too may have many neurons
- But will represent everything as simple boxes
  - Each box actually represents an entire *layer with many units*

The stock prediction problem…

To invest or not to invest?

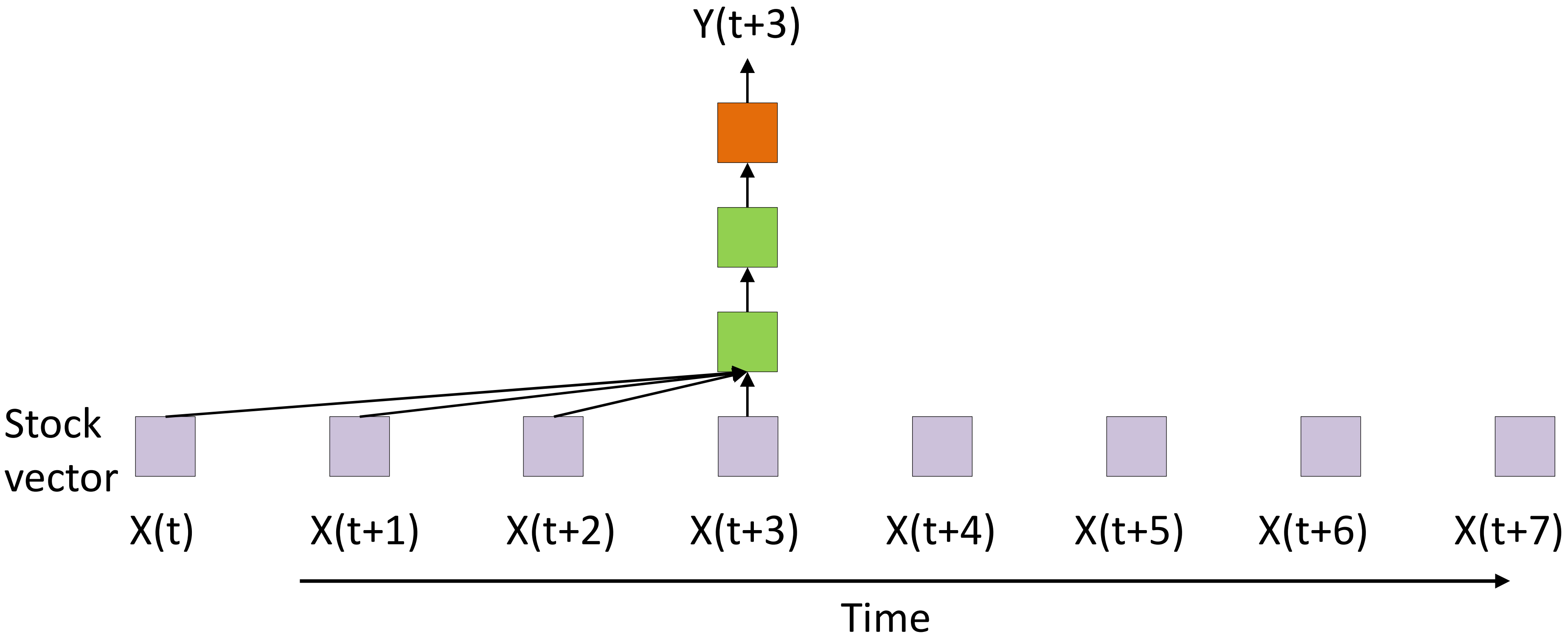


stocks

7/03 8/03 9/03 10/03 11/03 12/03 13/03 14/03 15/03

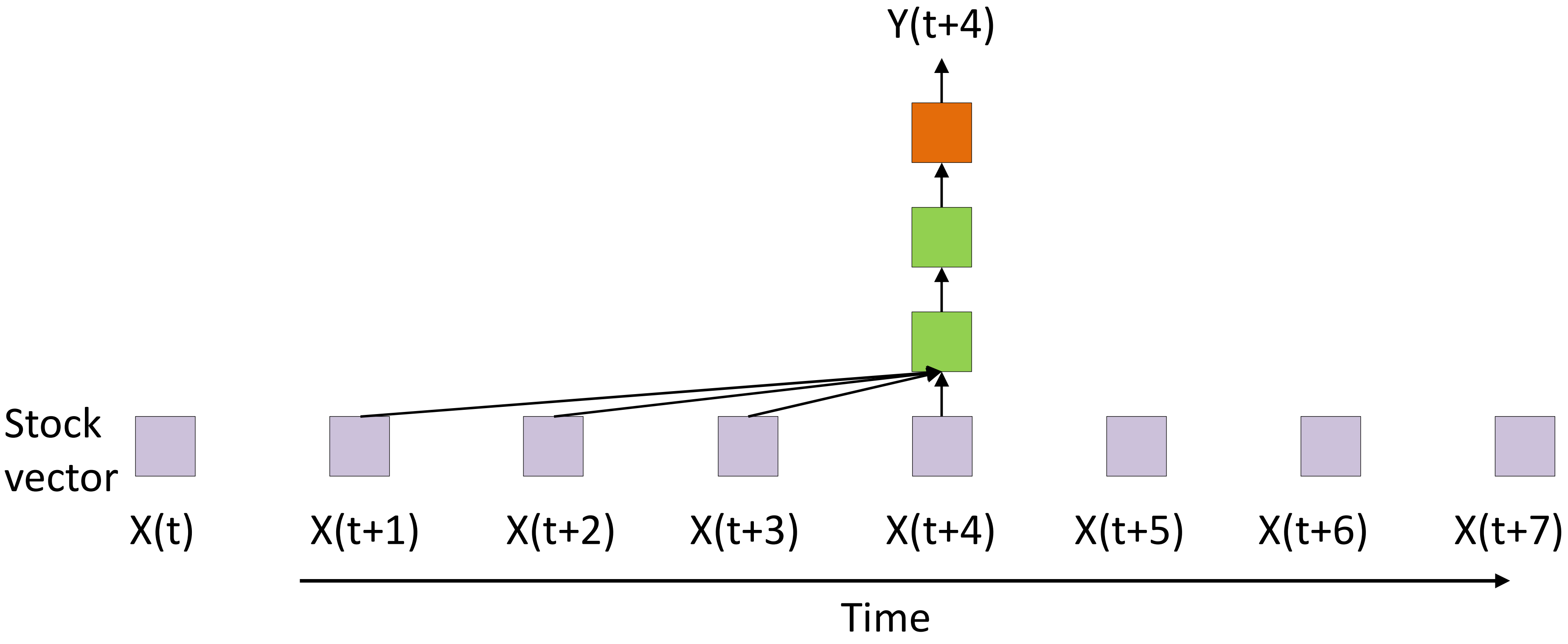
- Stock market
  - – Must consider the series of stock values in the past several days to decide if it is wise to invest today

The stock predictor network



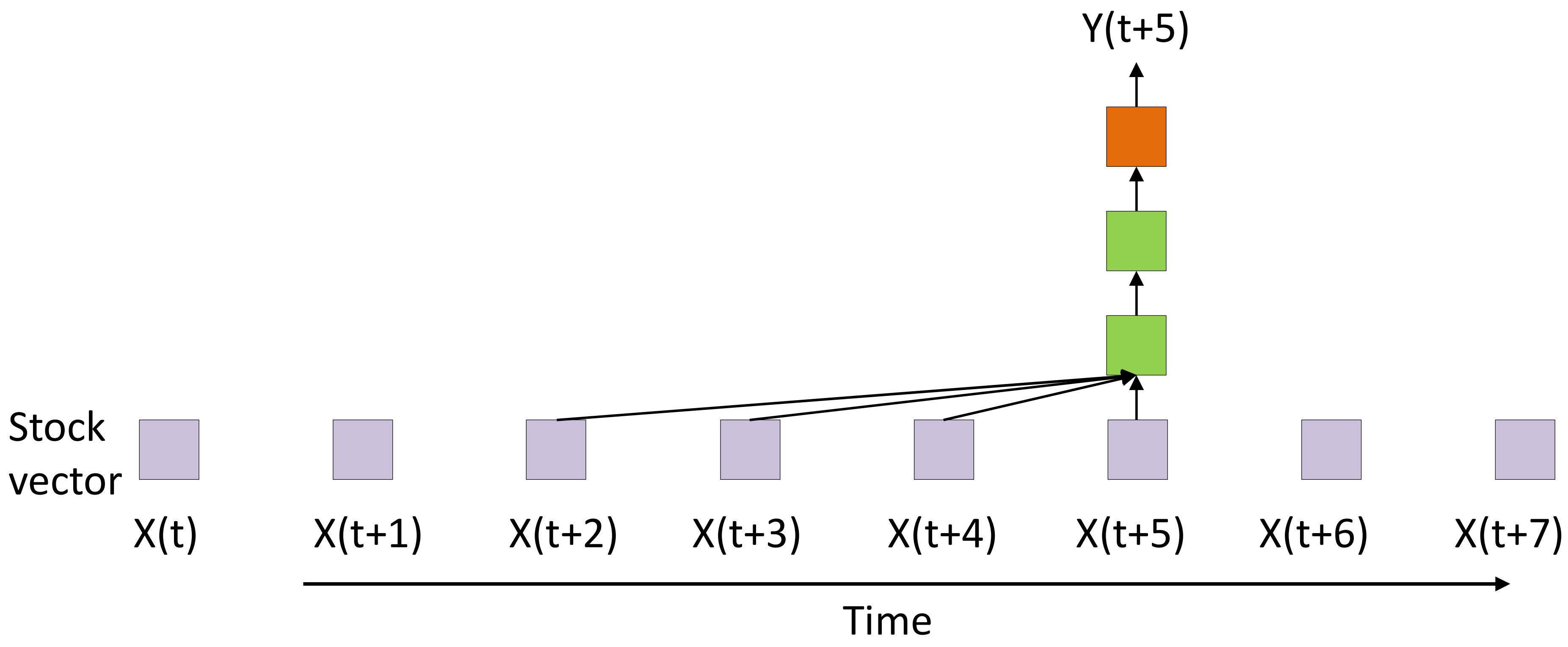
- The sliding predictor
  - Look at the last few days
  - This is just a convolutional neural net applied to series data
    - Also called a *Time-Delay neural network*

The stock predictor network



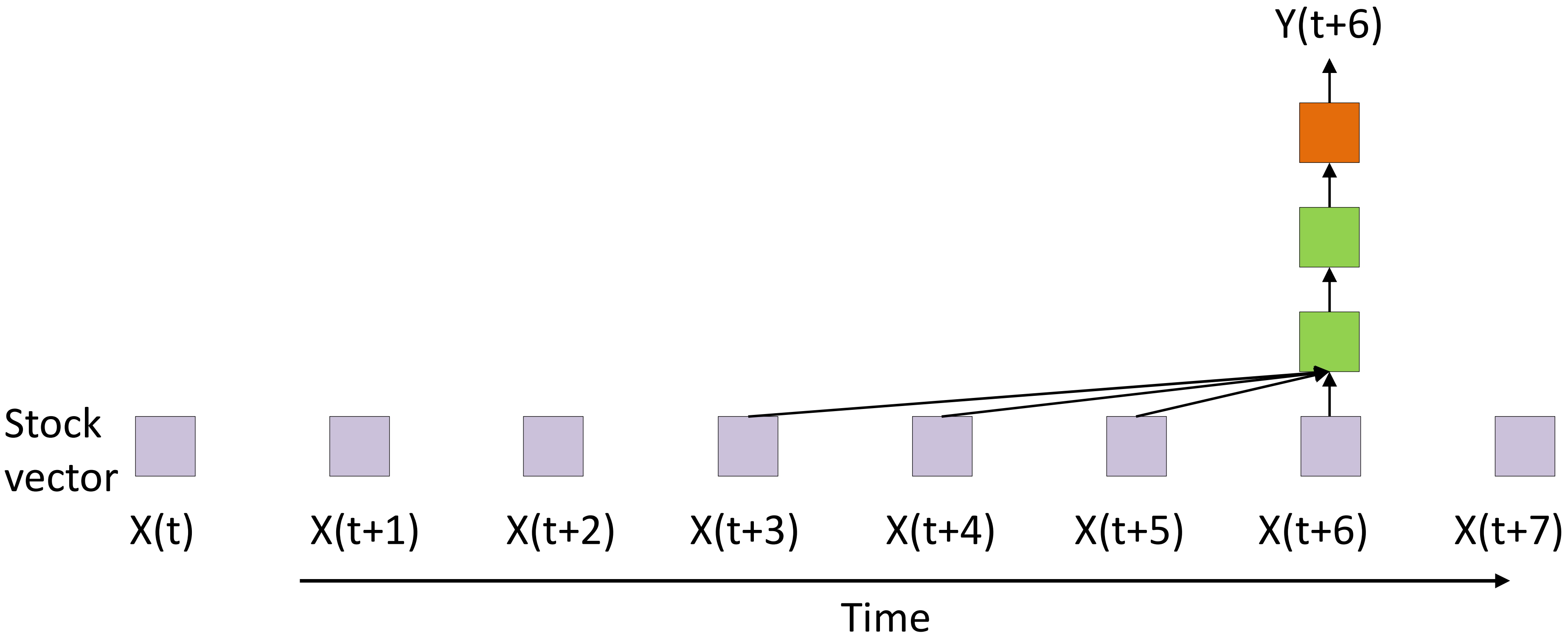
- The sliding predictor
  - Look at the last few days
  - This is just a convolutional neural net applied to series data
    - Also called a *Time-Delay neural network*

The stock predictor network



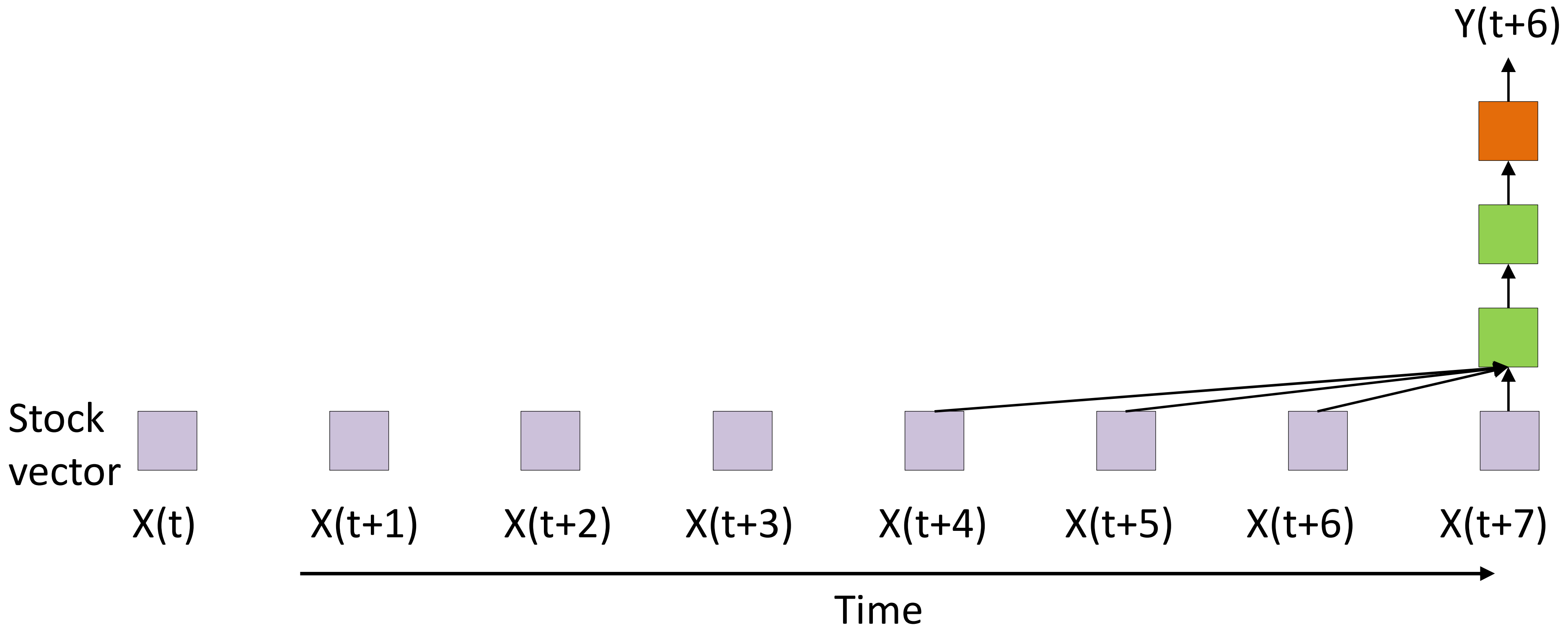
- The sliding predictor
  - Look at the last few days
  - This is just a convolutional neural net applied to series data
    - Also called a *Time-Delay neural network*

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



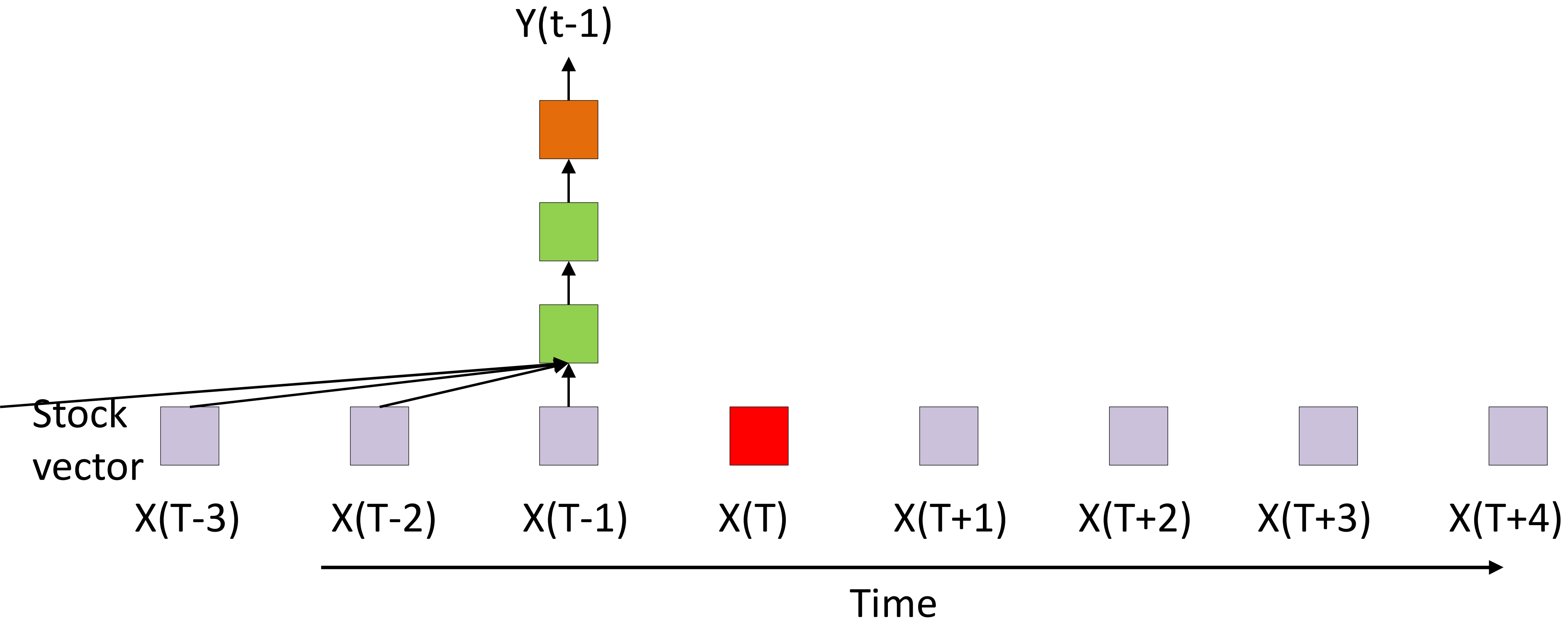
- The sliding predictor
  - Look at the last few days
  - This is just a convolutional neural net applied to series data
    - Also called a *Time-Delay neural network*

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

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

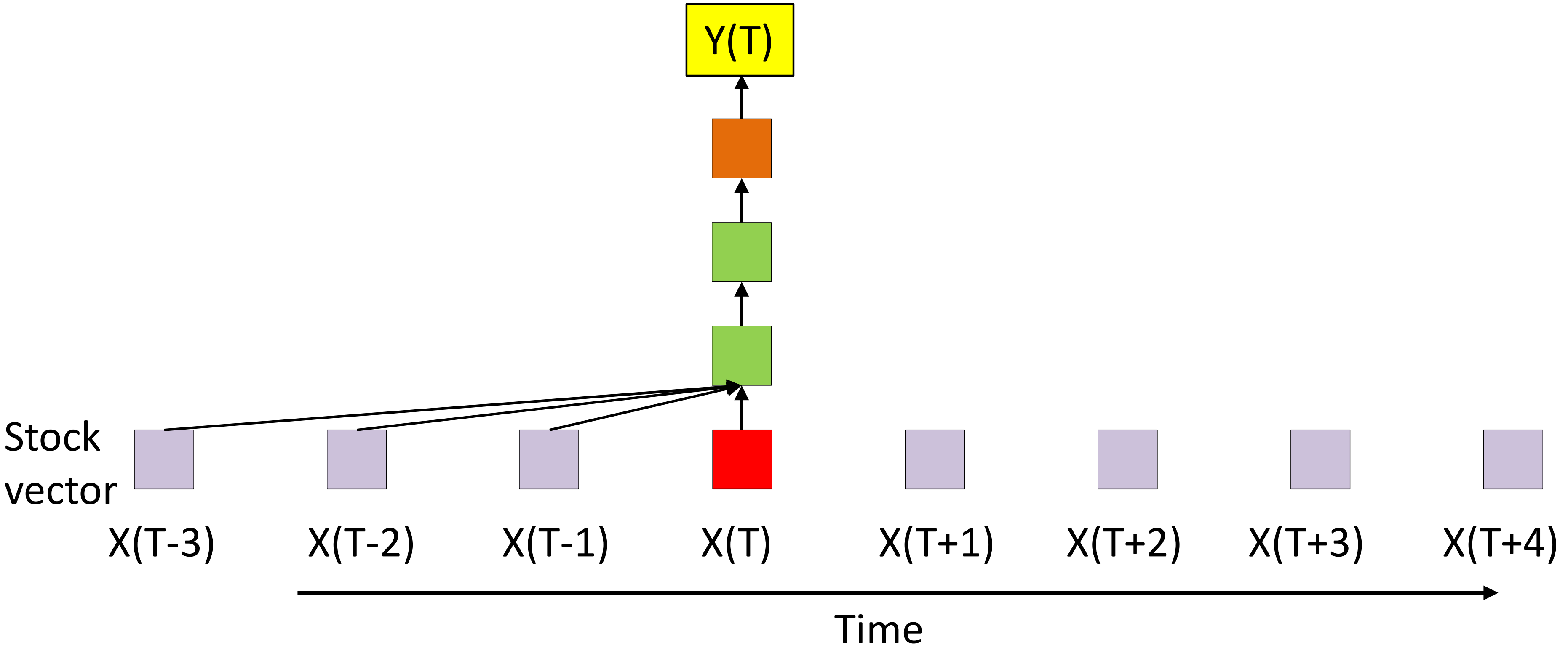
The stock predictor



- 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

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

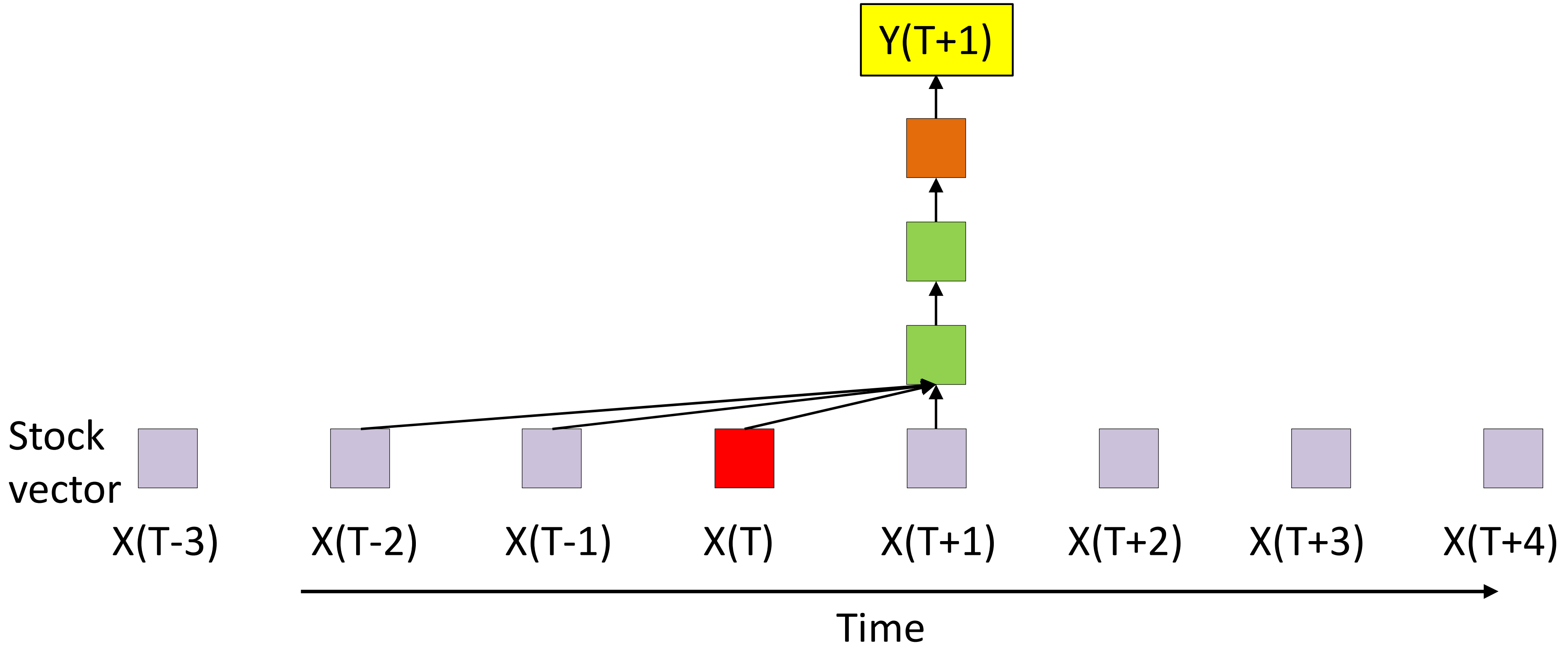
The stock predictor



- 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

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

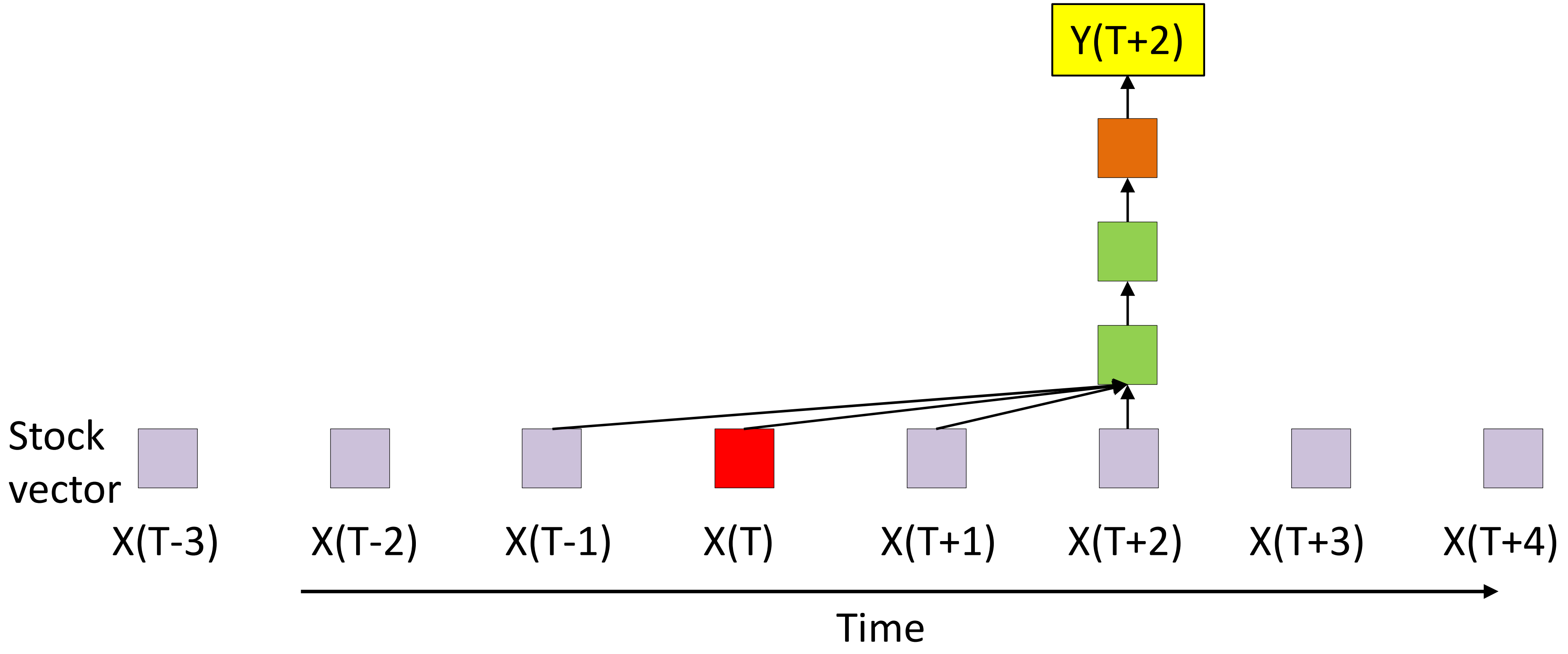
The stock predictor



- 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

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

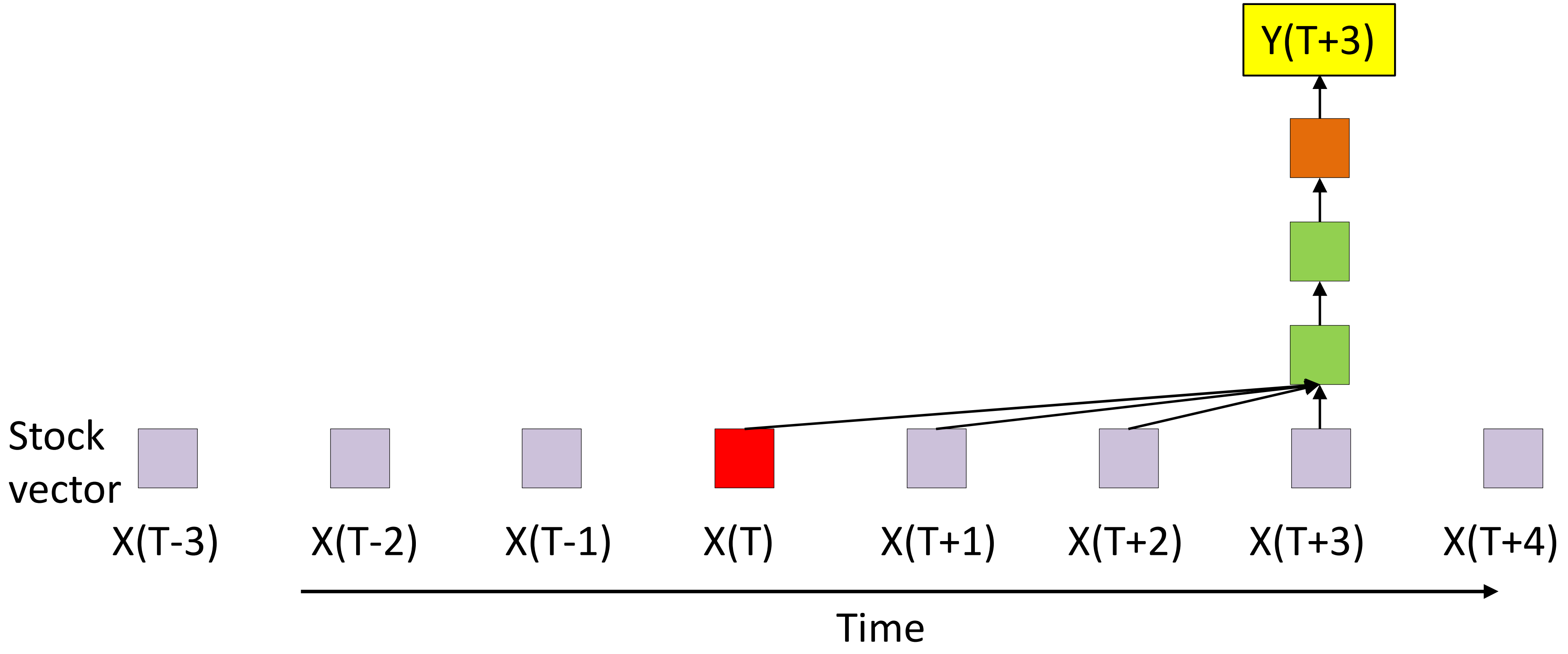
The stock predictor



- 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

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

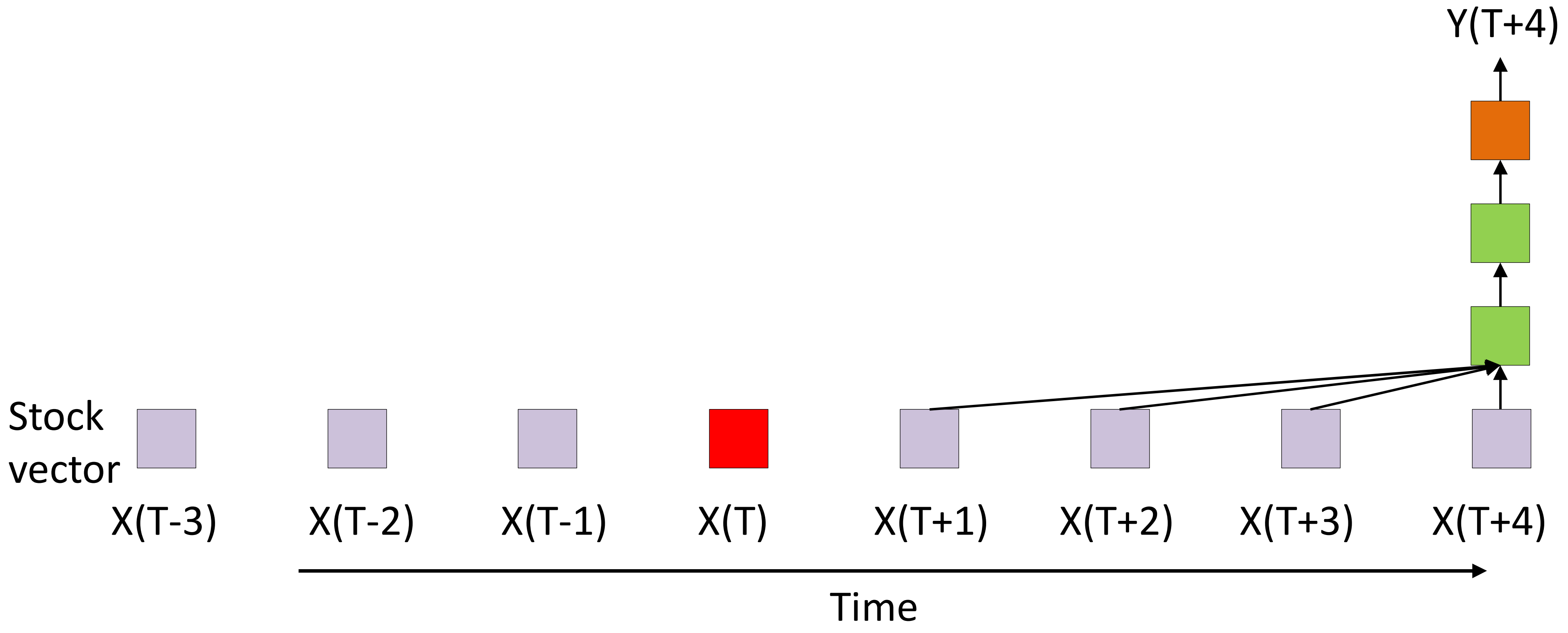
The stock predictor



- 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

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

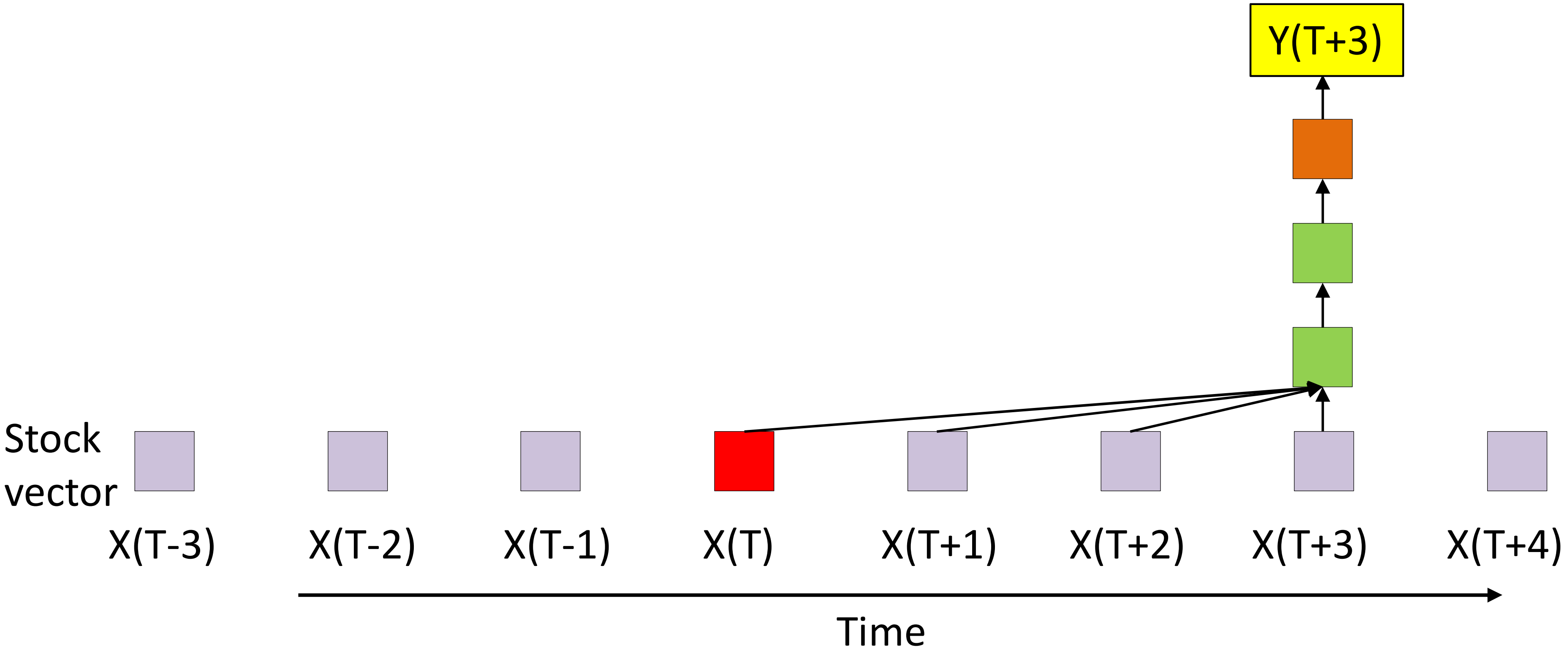
The stock predictor



- 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

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

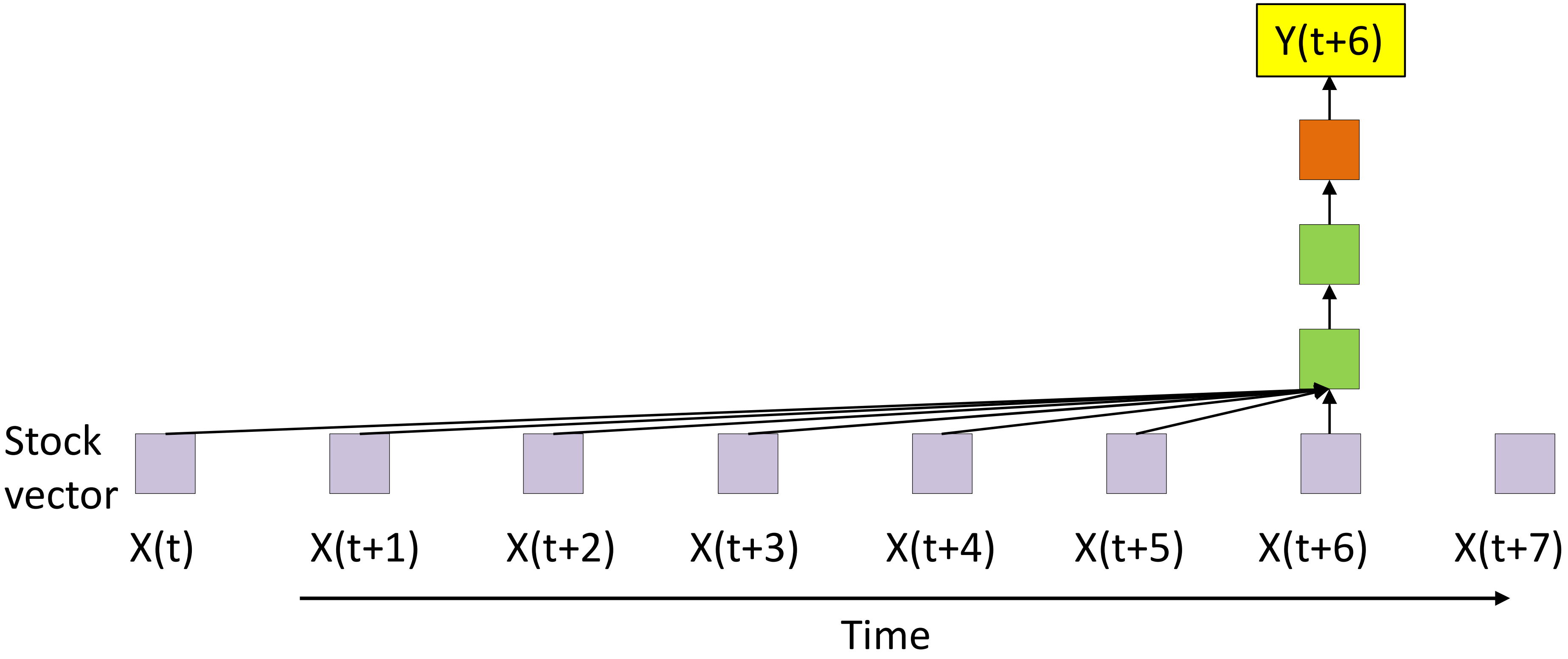
Finite-response model



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

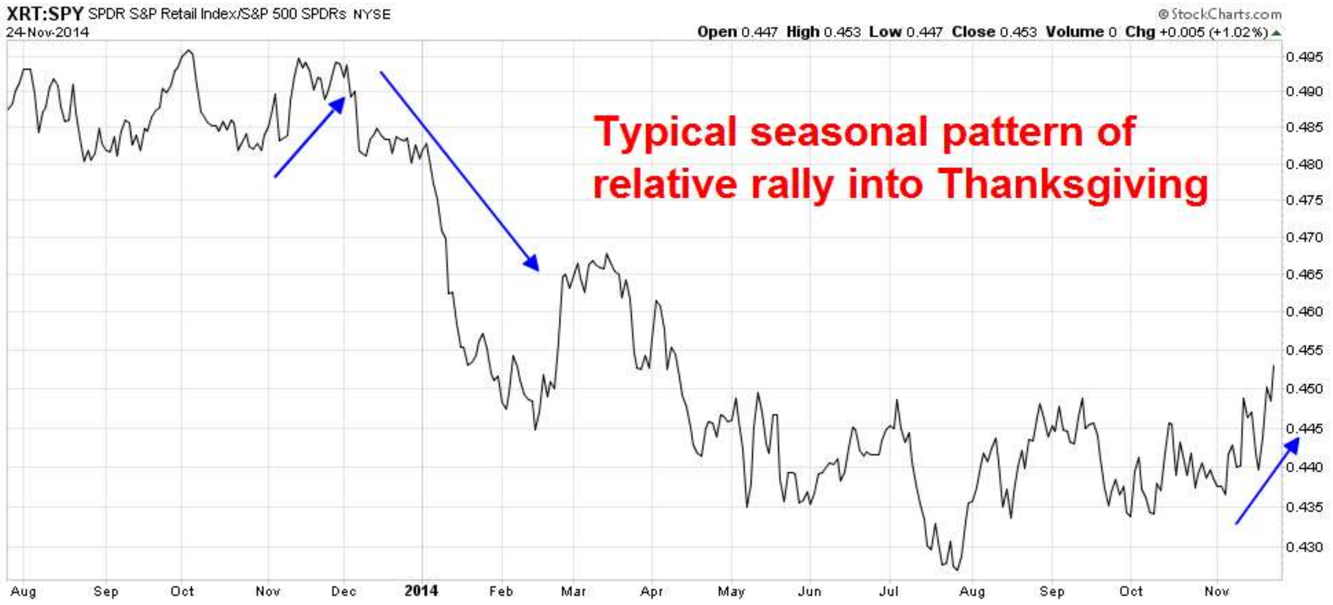
- To consider more of the past to make predictions, you must increase the "history" considered by the system

Finite-response



- Problem: Increasing the “history” makes the network more complex
  - – No worries, we have the CPU and memory
    - Or do we?

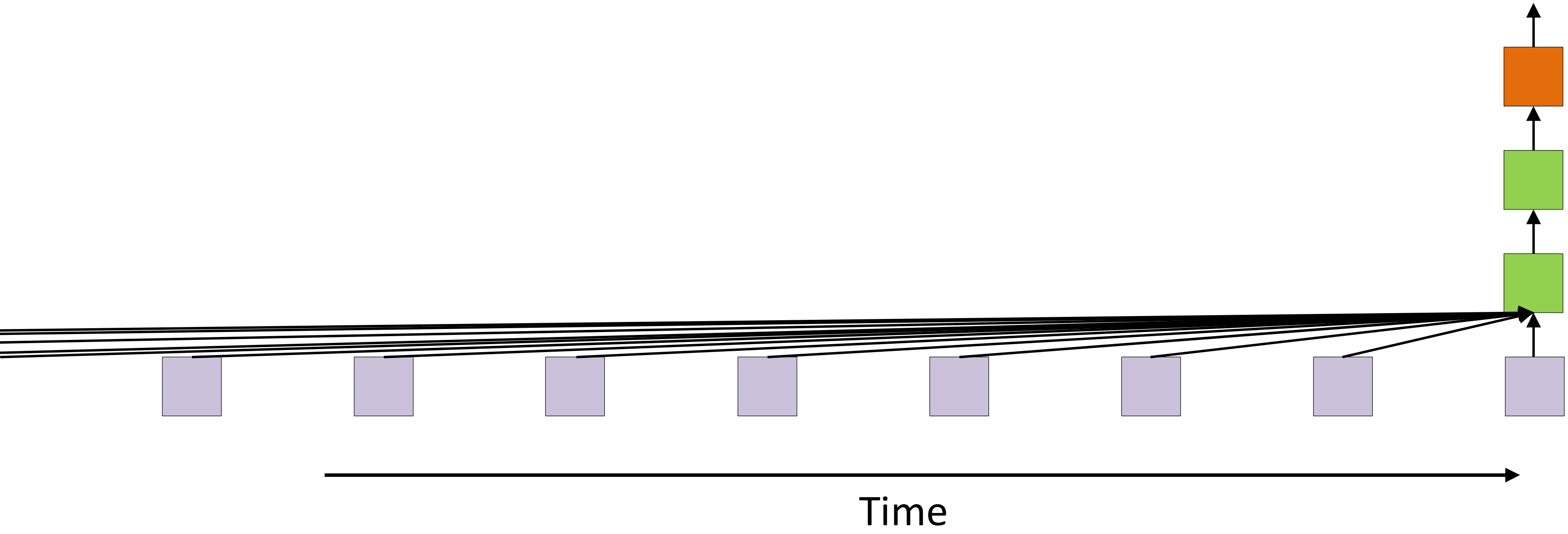
Systems often have long-term dependencies



Typical seasonal pattern of relative rally into Thanksgiving

- Longer-term trends –
  - Weekly trends in the market
  - Monthly trends in the market
  - Annual trends
  - Though longer historic tends to affect us less than more recent events..

We want *infinite* memory



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

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

Poll 1 (@613 @614)

Convolutional neural networks are finite response systems, true or false

- True
- False

An input at time T affects the output of the convolutional layers of the network for all time, true or false

- True
- False

Poll 1

Convolutional neural networks are finite response systems, true or false

- **True**
- False

An input at time T affects the output of the convolutional layers of the network for all time, true or false

- True
- **False**

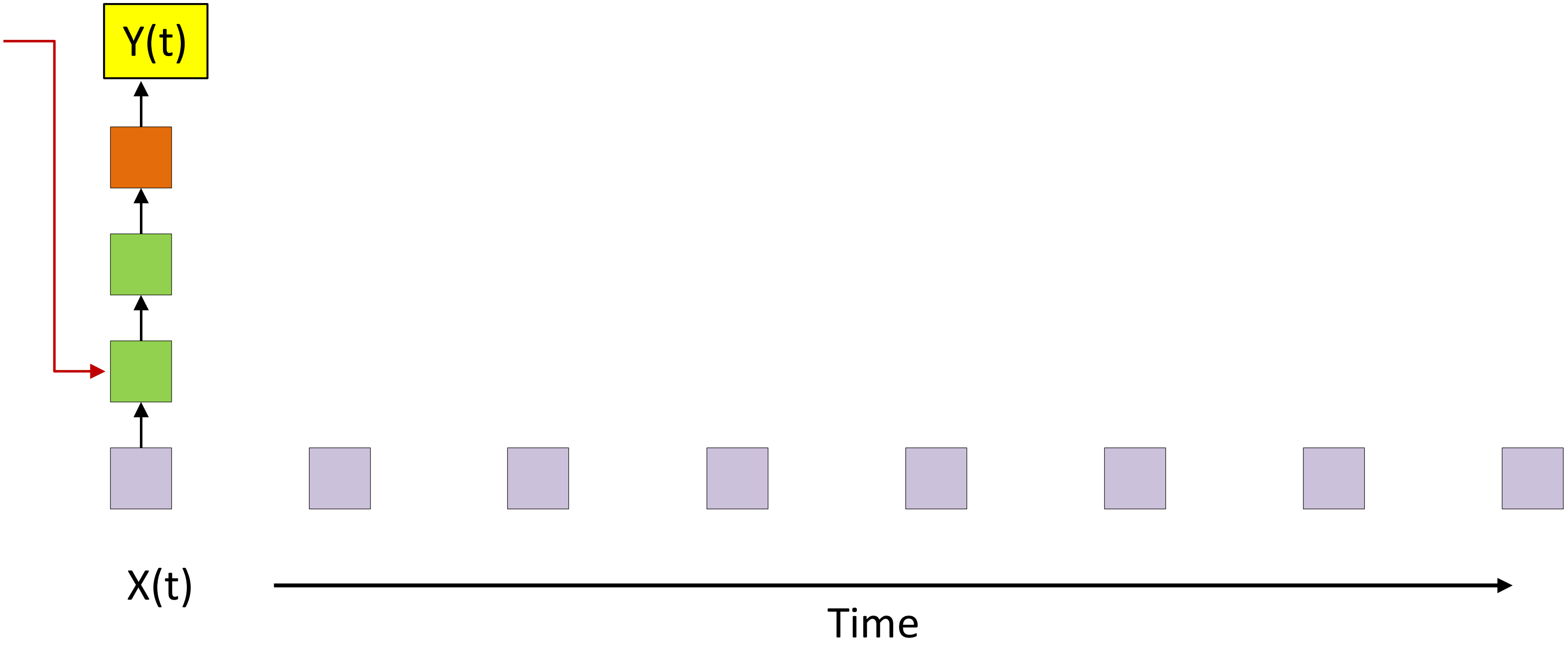
Examples of infinite response systems

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

- Required: Define initial state: Y_{-1} for $t = 0$
- An input at X_0 at $t = 0$ produces Y_0
- Y_0 produces Y_1 which produces Y_2 and so on until Y_∞ *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!**

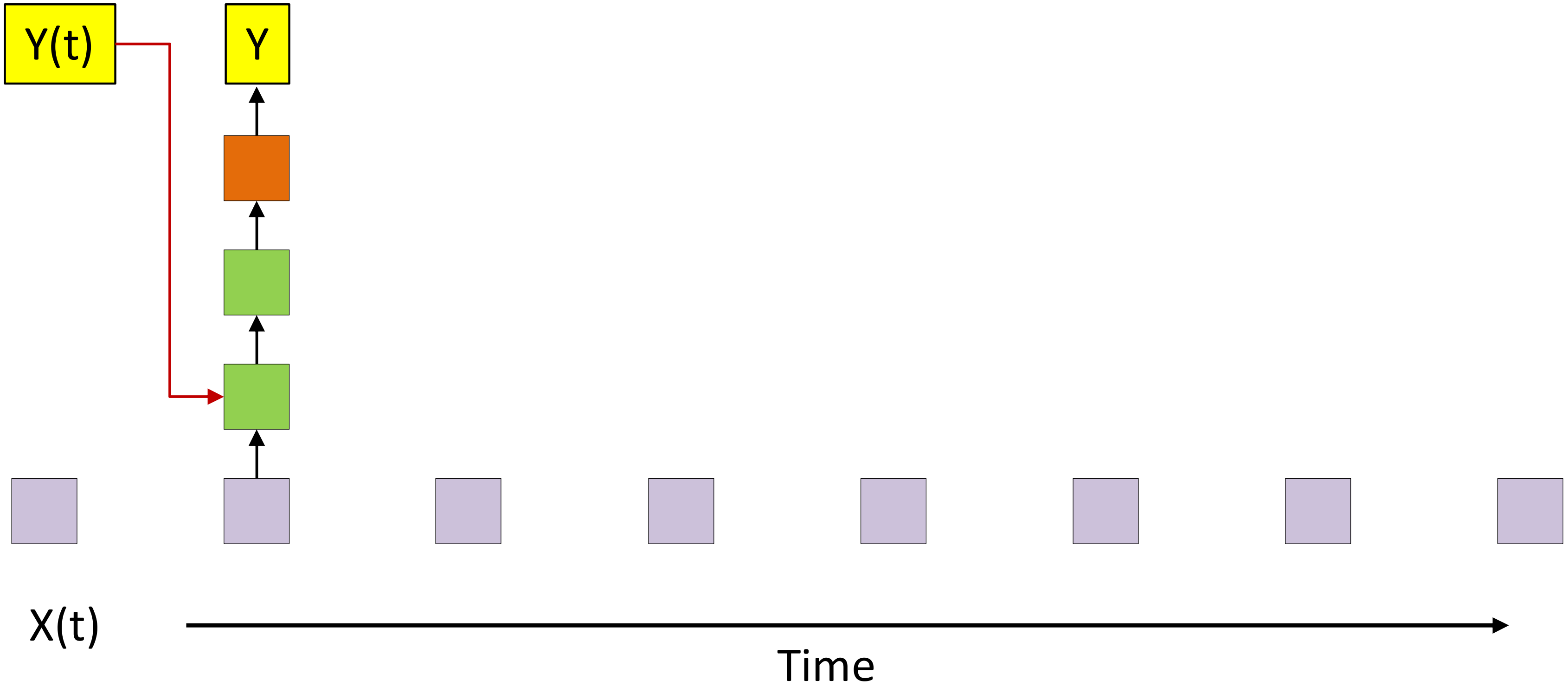
- This is an instance of a NARX network
  - "nonlinear autoregressive network with exogenous inputs"
  - $Y_t = f(X_{0:t}, Y_{0:t-1})$
- *Output* contains information about the entire past

A one-tap NARX network



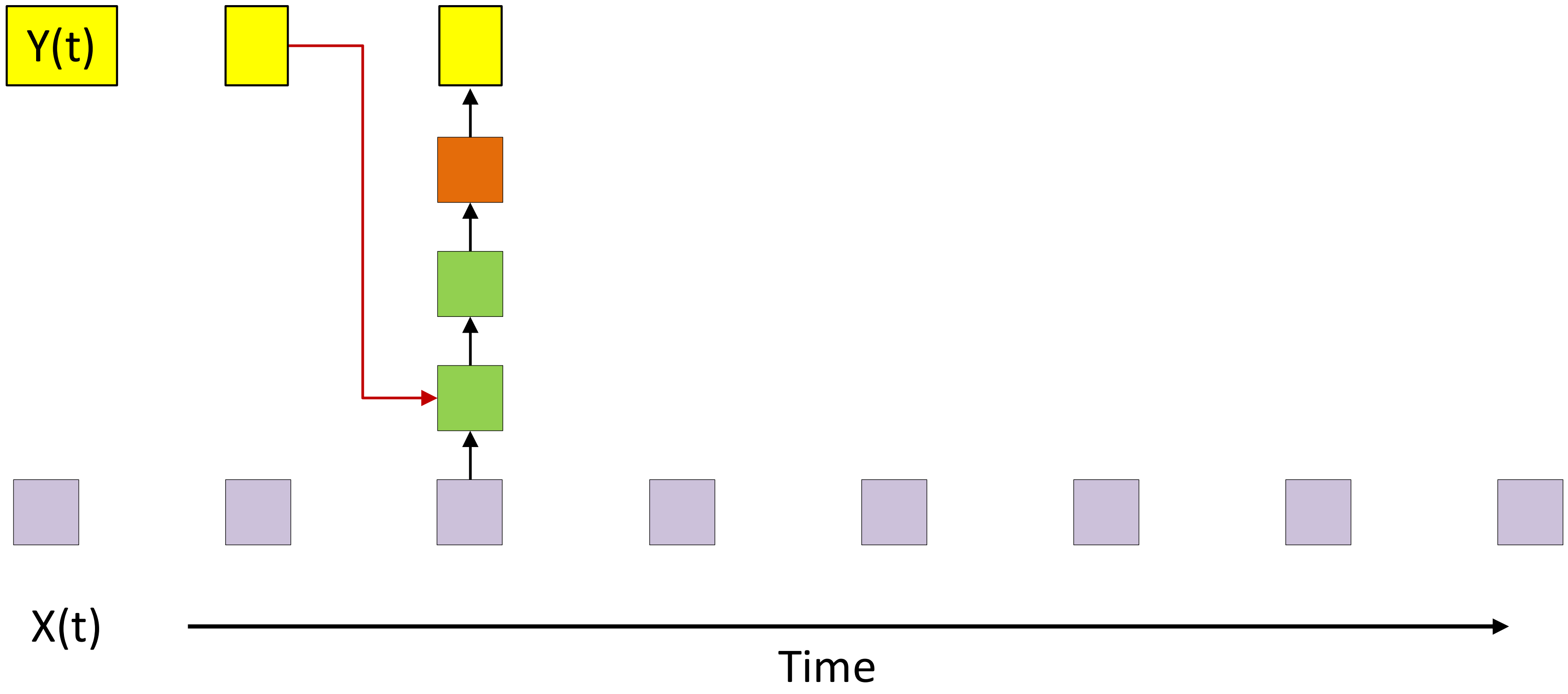
- A NARX net with recursion from the output

A one-tap NARX network



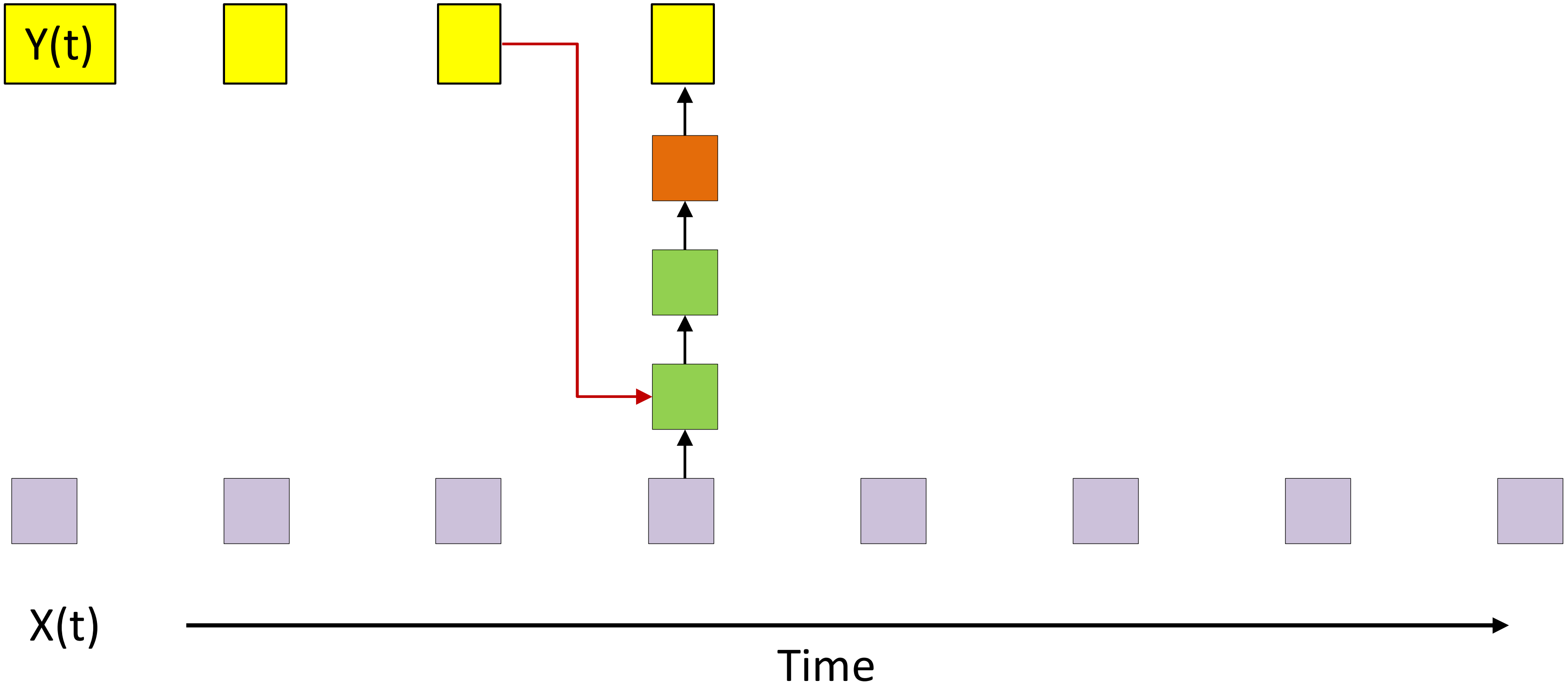
- A NARX net with recursion from the output

A one-tap NARX network



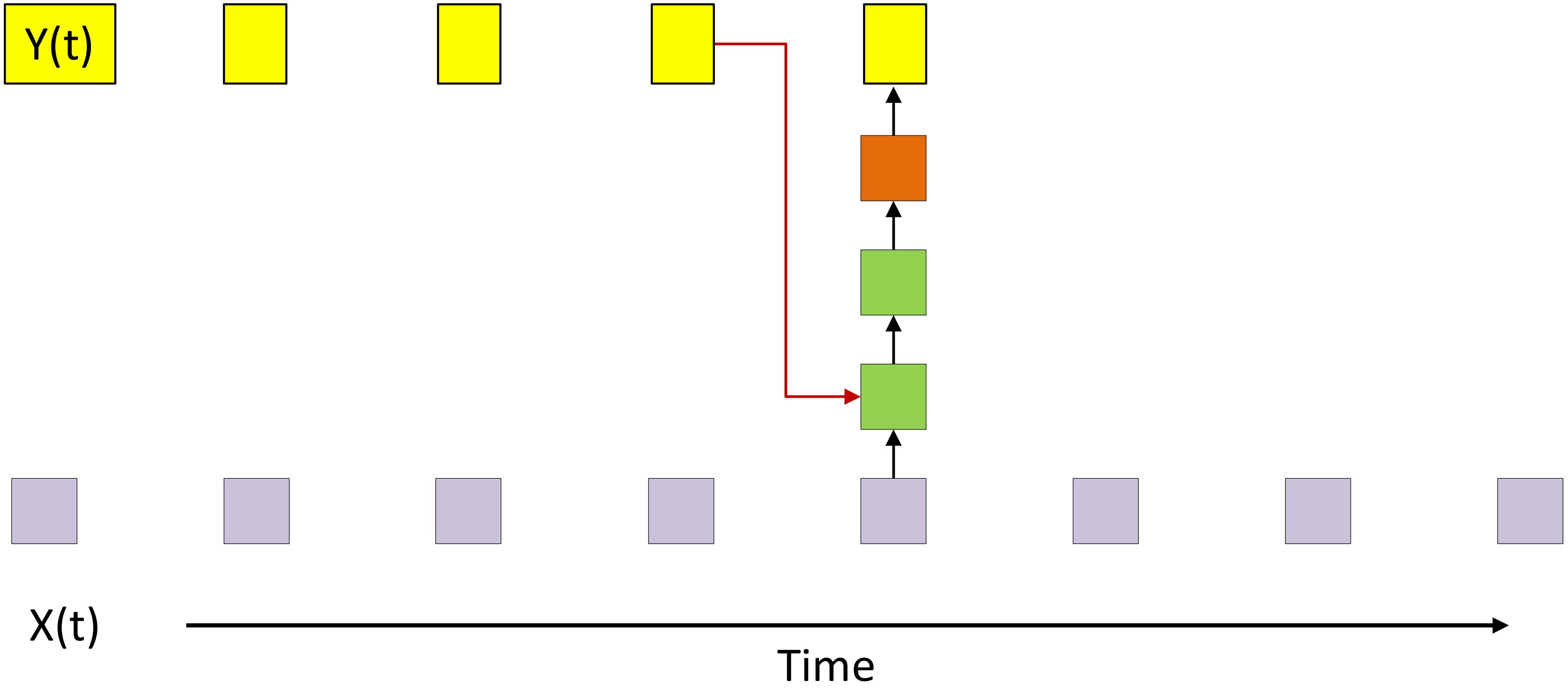
- A NARX net with recursion from the output

A one-tap NARX network



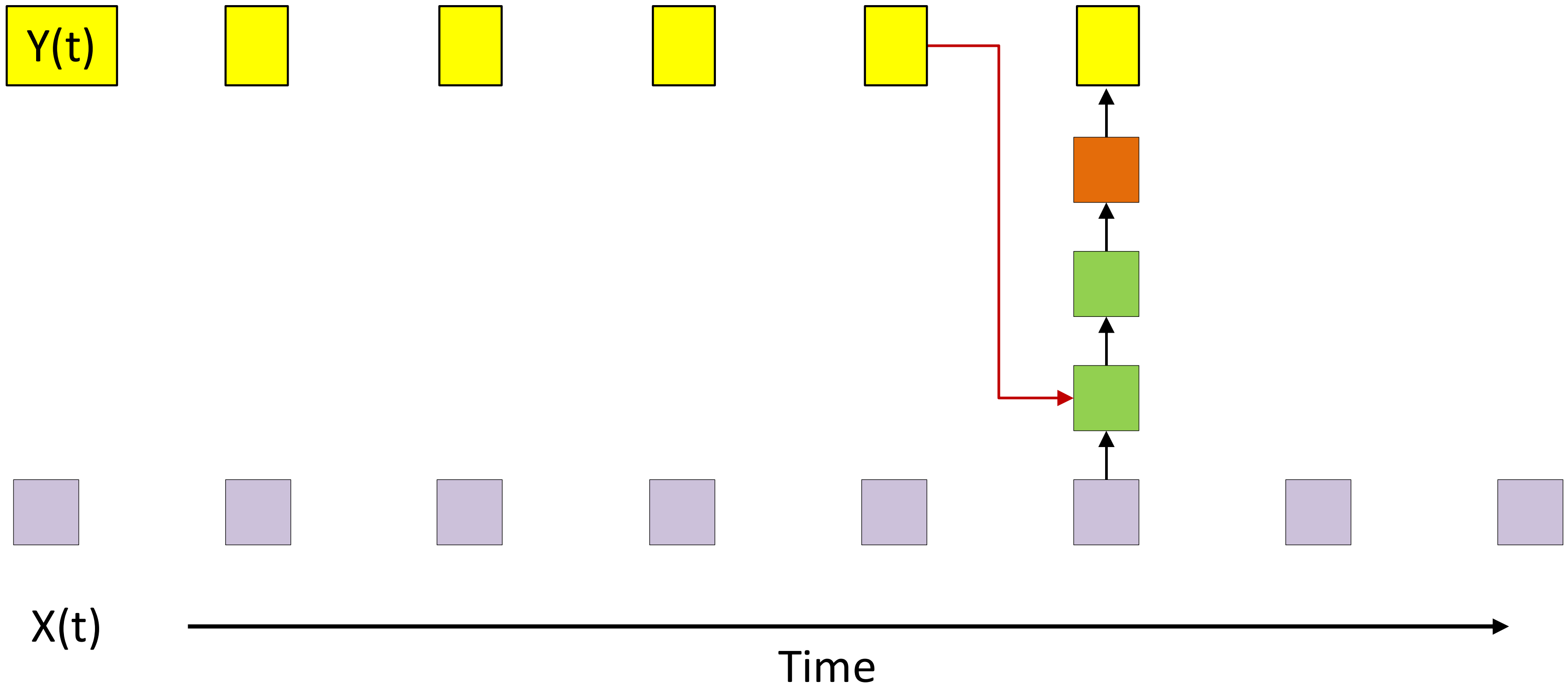
- A NARX net with recursion from the output

A one-tap NARX network



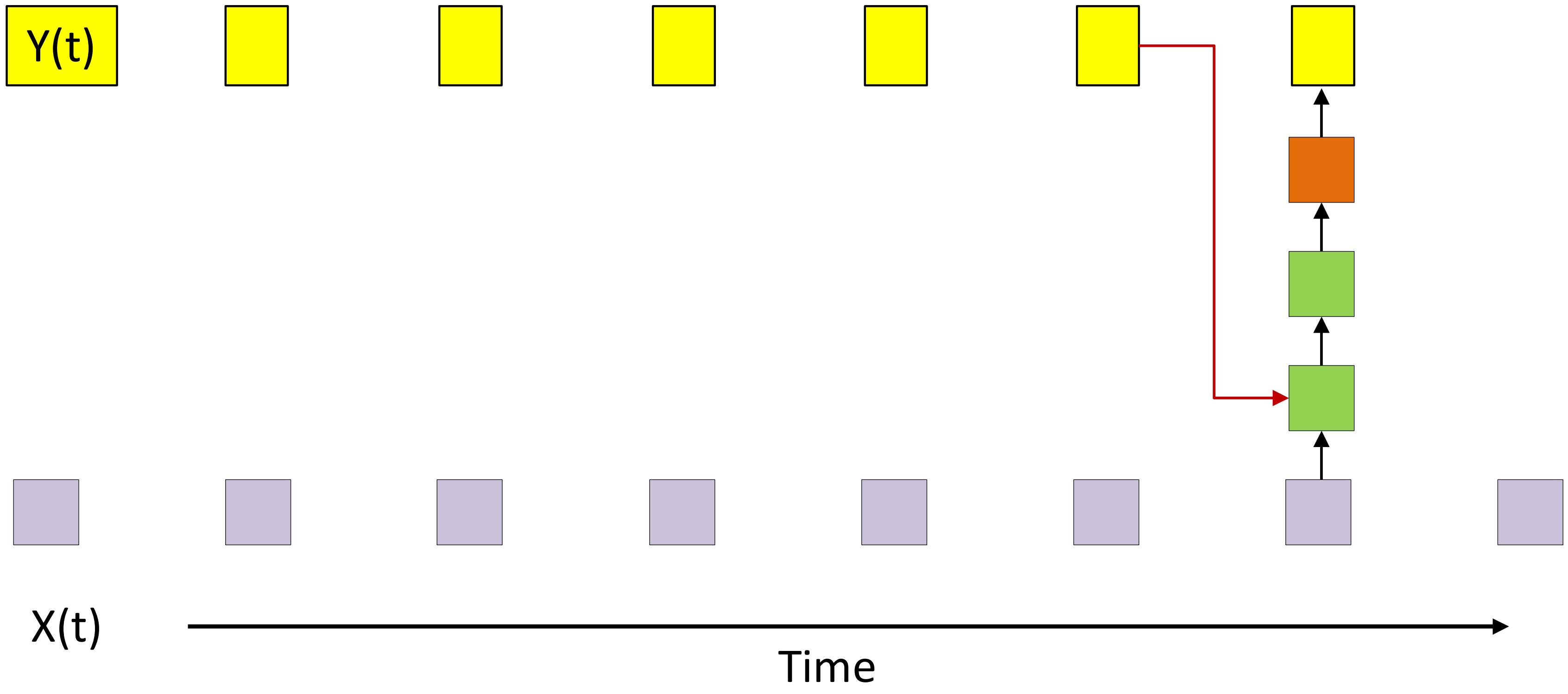
- A NARX net with recursion from the output

A one-tap NARX network



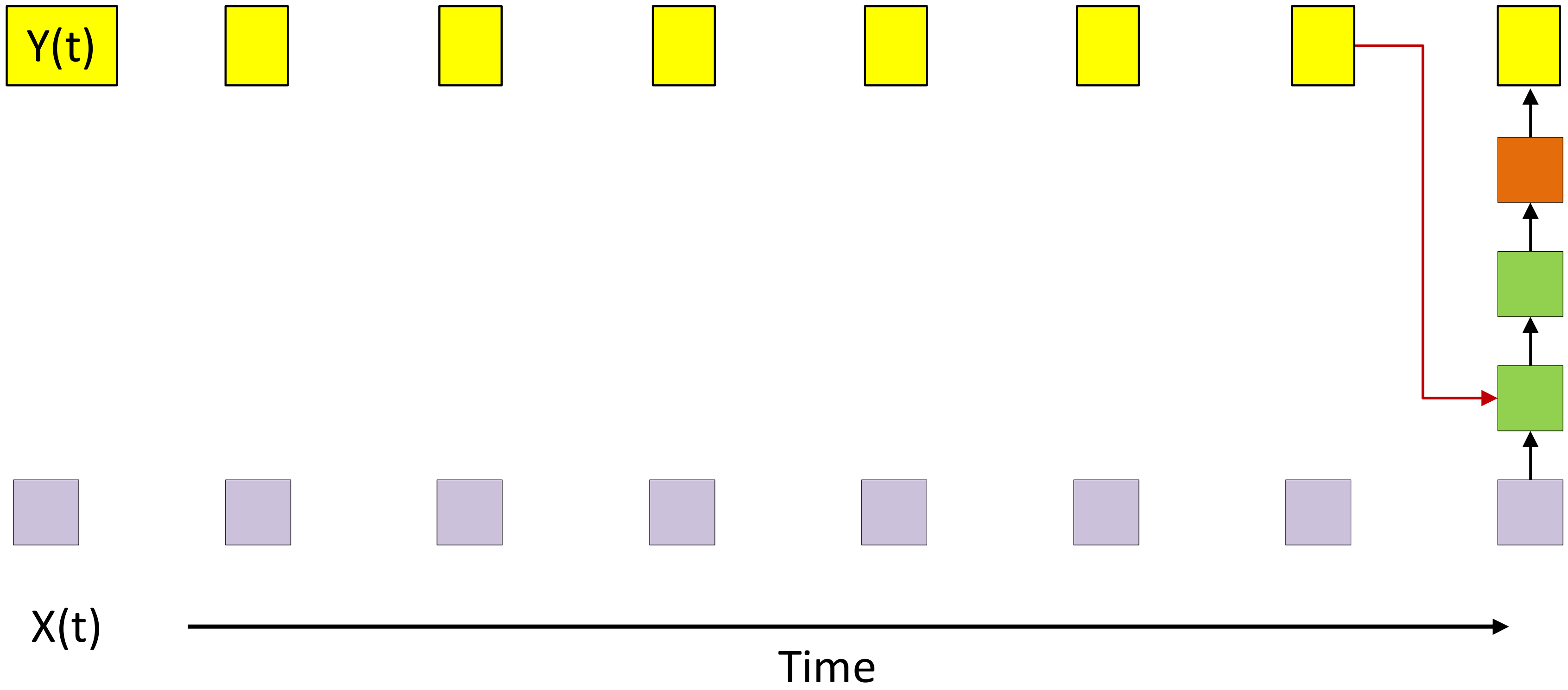
- A NARX net with recursion from the output

A one-tap NARX network



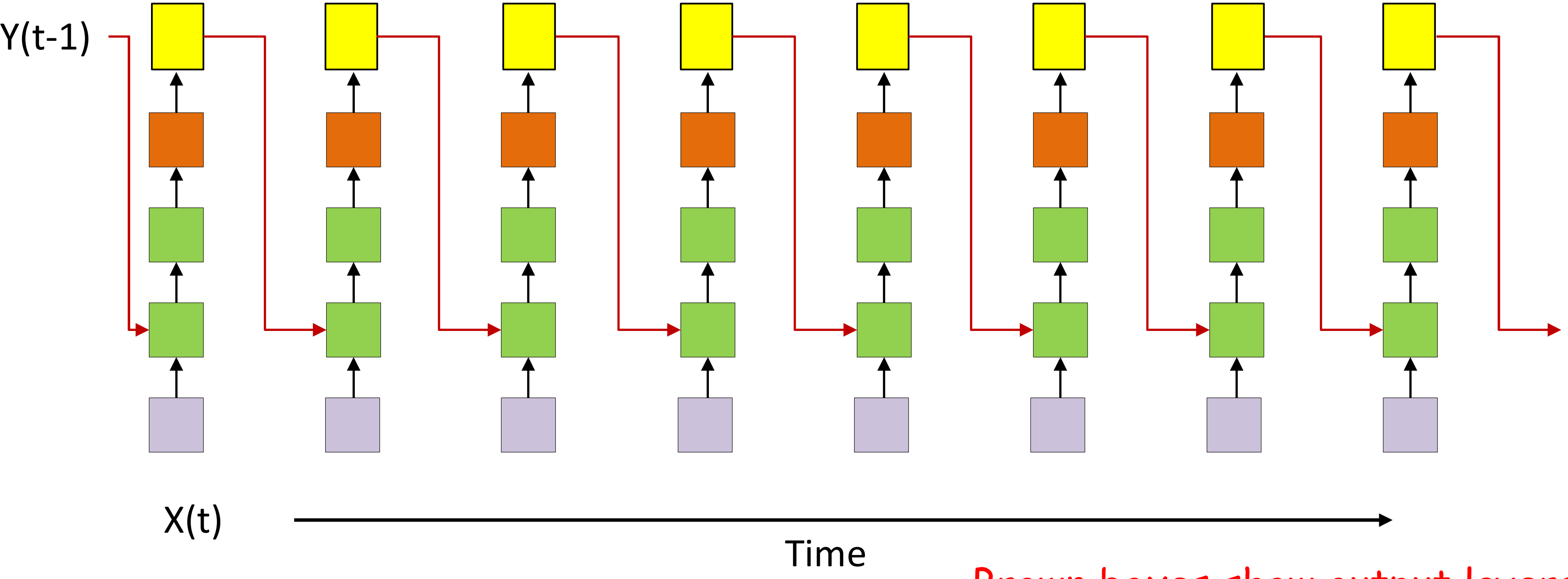
- A NARX net with recursion from the output

A one-tap NARX network



- A NARX net with recursion from the output

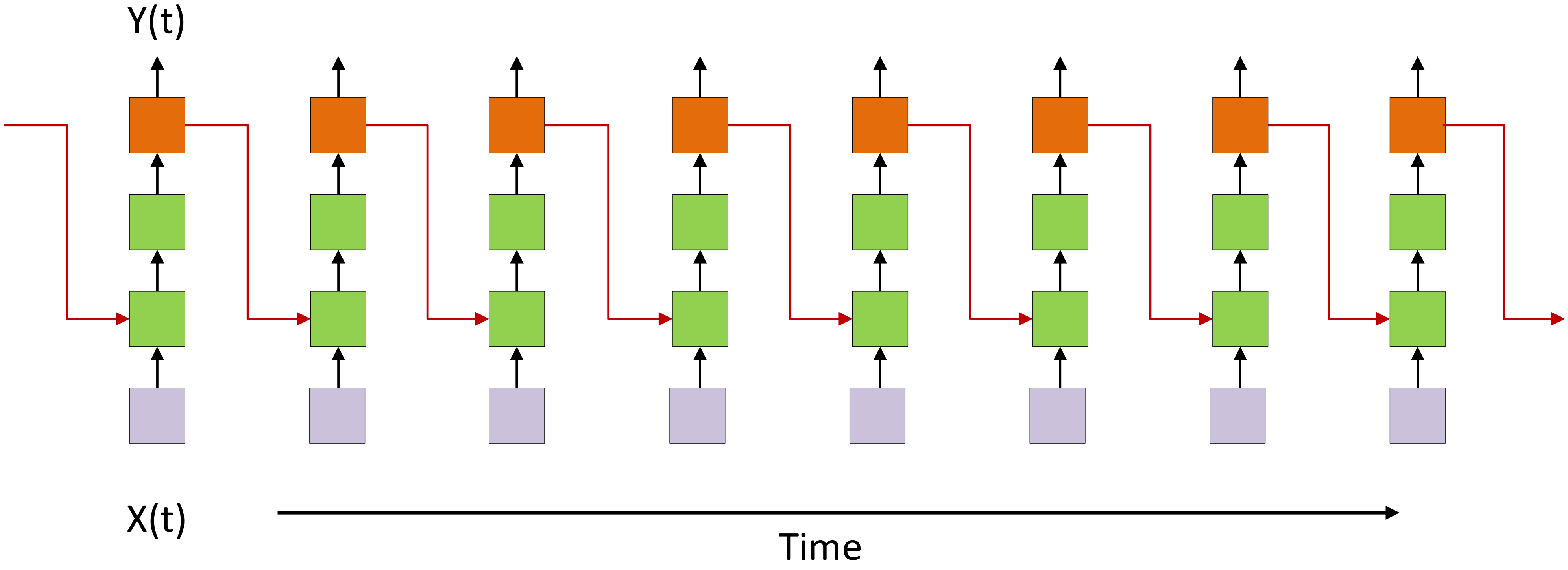
A more complete representation



Brown boxes show output layers
Yellow boxes are outputs

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

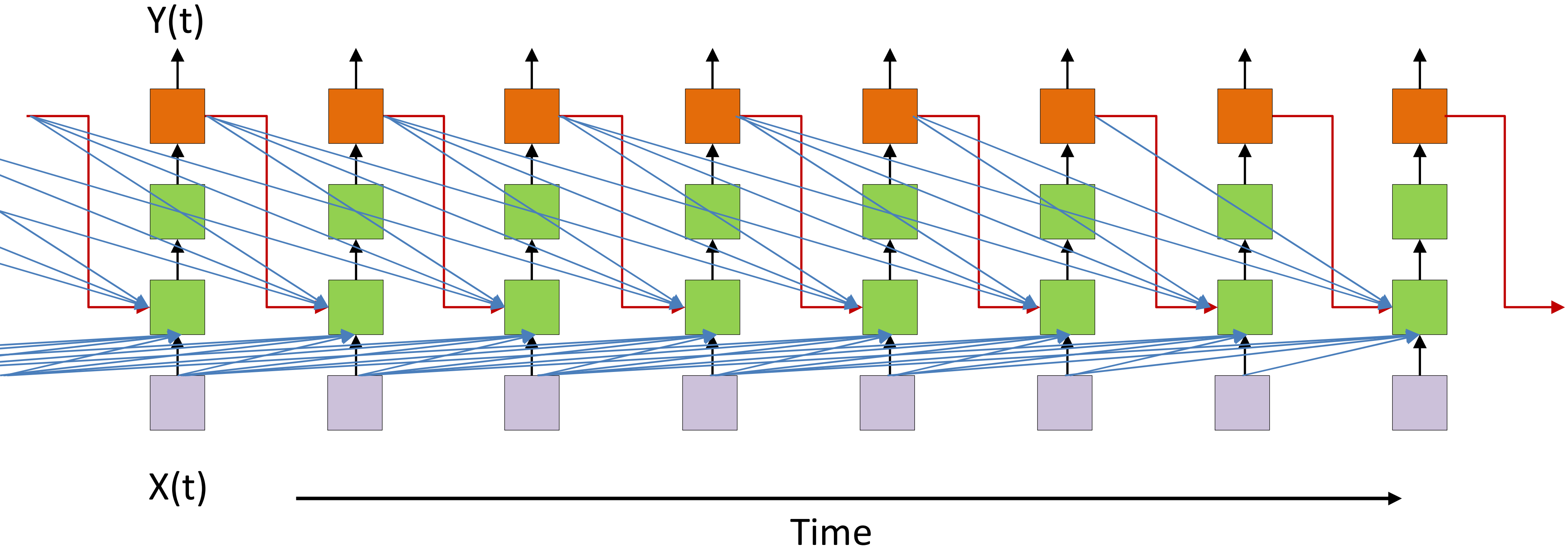
Same figure redrawn



Brown boxes show output layers
All outgoing arrows are the same output

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

A more generic NARX network



- The output Y_t at time t is computed from the past K outputs $Y_{t-1}, \ldots, Y_{t-K}$ and the current and past L inputs $X_t, \ldots, X_{t-L}$

NARX Networks

- Very popular for time-series prediction
  - Weather
  - Stock markets
  - As alternate system models in tracking systems
  - ***Language***
- Any phenomena with distinct “innovations” that “drive” an output
- Note: here the “memory” of the past is in the output itself, and not in the network

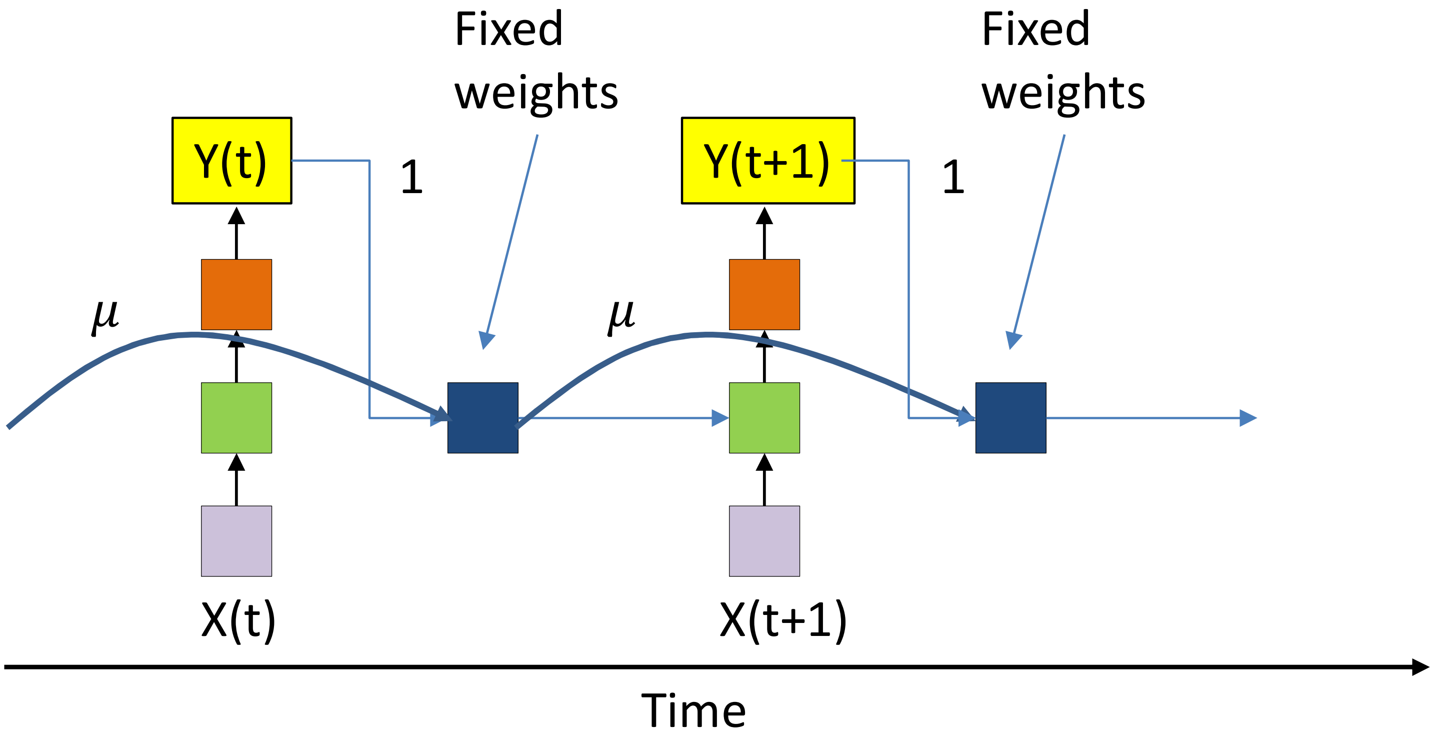
Let’s make memory more explicit

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

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

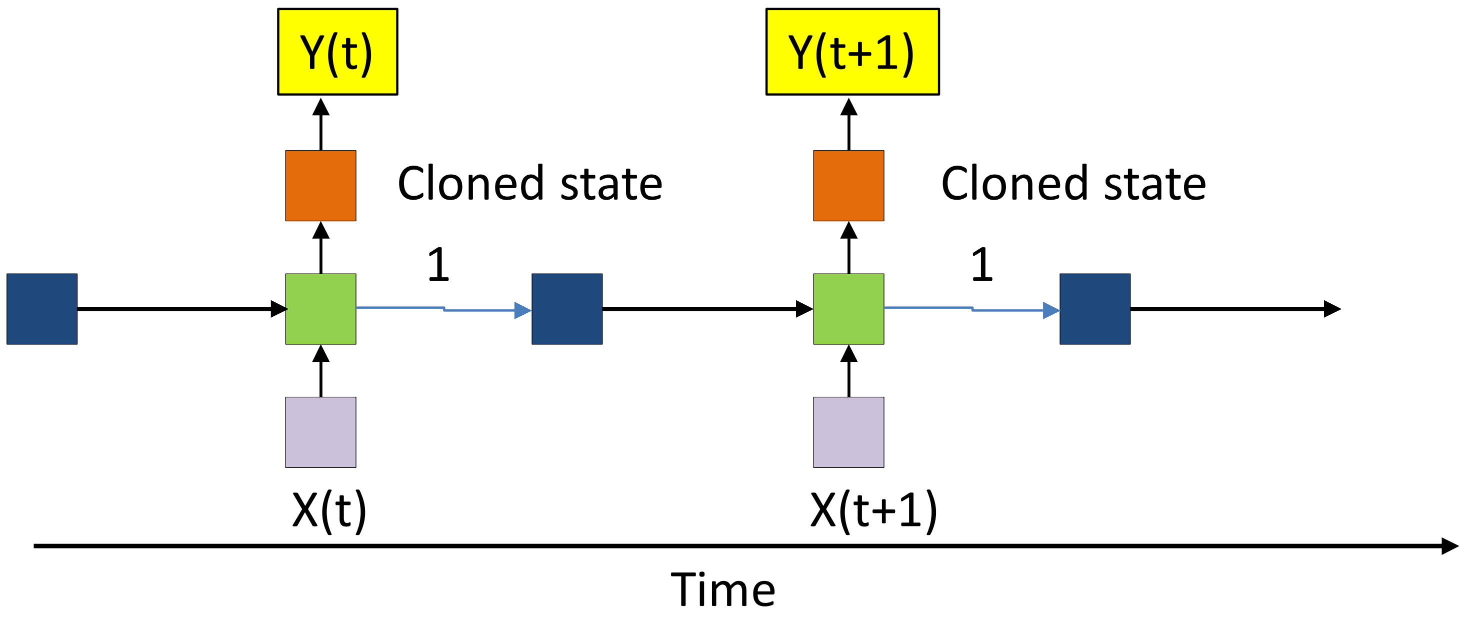
- m_t is a “memory” variable
  - Generally stored in a “memory” unit
  - Used to “remember” the past

Jordan Network



- Memory unit simply retains a running average of past outputs
  - "Serial order: A parallel distributed processing approach", M.I.Jordan, 1986
    - Input is constant (called a "plan")
    - Objective is to train net to produce a specific output, given an input plan
  - 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
    - For the purpose of training, this was approximated as a set of T independent 1-step history nets
- Only the weight *from* the memory unit to the hidden unit is learned
  - But during training no gradient is backpropagated over the "1" link

Story so far

- In time series analysis, models must look at past inputs along with current input
  - Looking at a finite horizon of past inputs gives us a convolutional network
- Looking into the infinite past requires recursion

- NARX networks recurse by feeding back the output to the input
  - May feed back a finite horizon of outputs

- "Simple" recurrent networks:
  - Jordon networks maintain a running average of outputs in a "memory" unit
  - Elman networks store *hidden* unit values for one time instant in a "context" unit
  - "Simple" (or partially recurrent) because during *learning* current error does not actually propagate to the past
    - "Blocked" at the memory units in Jordan networks
    - "Blocked" at the "context" unit in Elman networks

Poll 2 (@615 @616)

Memory neuron models have true recurrence, true or false

- True
- False

Memory neuron networks dedicate neurons specifically to store past history, true or false

- True
- False

Poll 2

Memory neuron models have true recurrence, true or false

- True
- **False**

Memory neuron networks dedicate neurons specifically to store past history, true or false

- **True**
- False

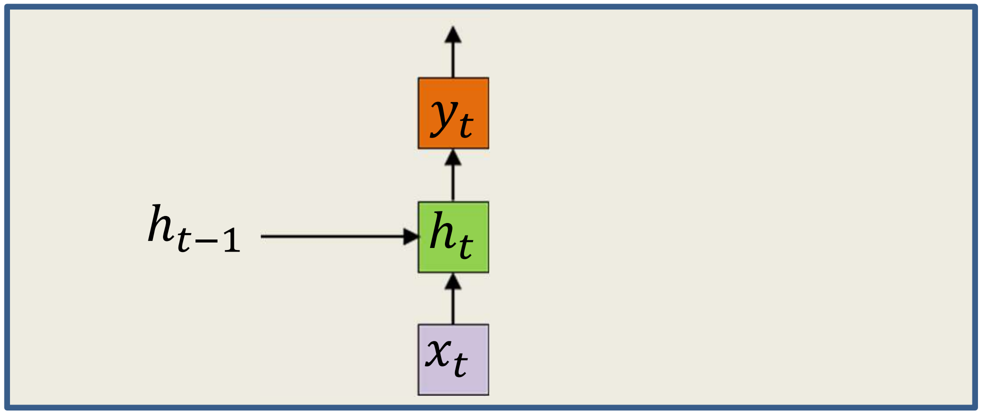
An alternate model for infinite response systems: the state-space model

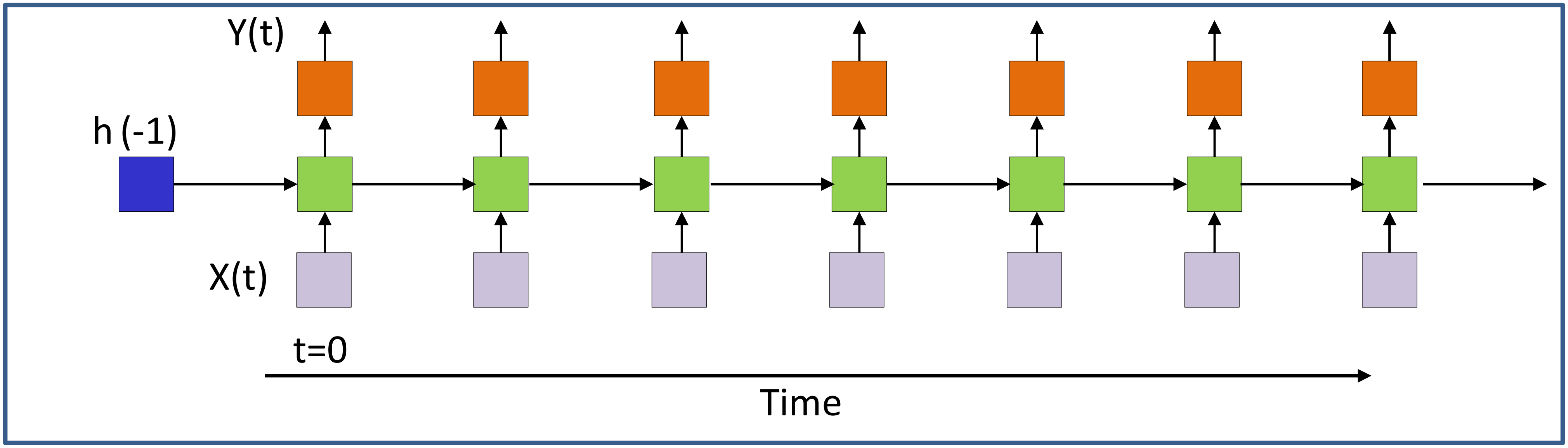
$$h_t = f(x_t, h_{t-1})$$
$$y_t = g(h_t)$$

- 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 *fully recurrent* neural network
  - Or simply a *recurrent neural network*

The simple state-space model

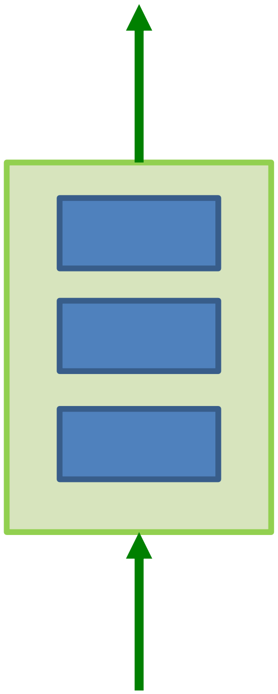




- The state (green) at any time is determined by the input at that time, and the state at the previous time
- *An input at t=0 affects outputs forever*
- Also known as a recurrent neural net

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

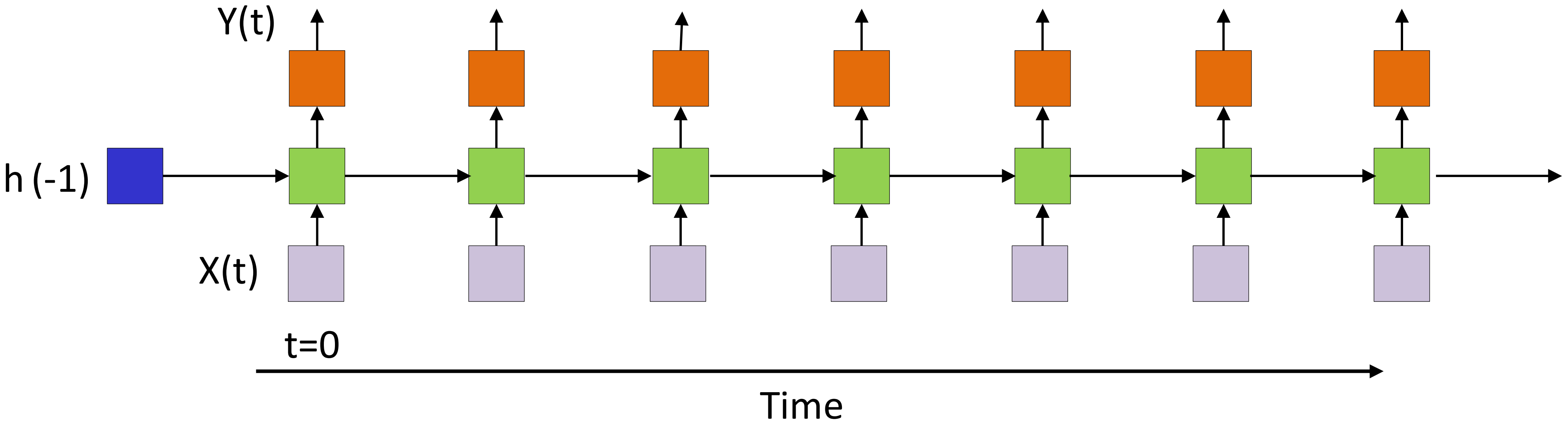
$$h_t = f(x_t, h_{t-1})$$
$$y_t = g(h_t)$$



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

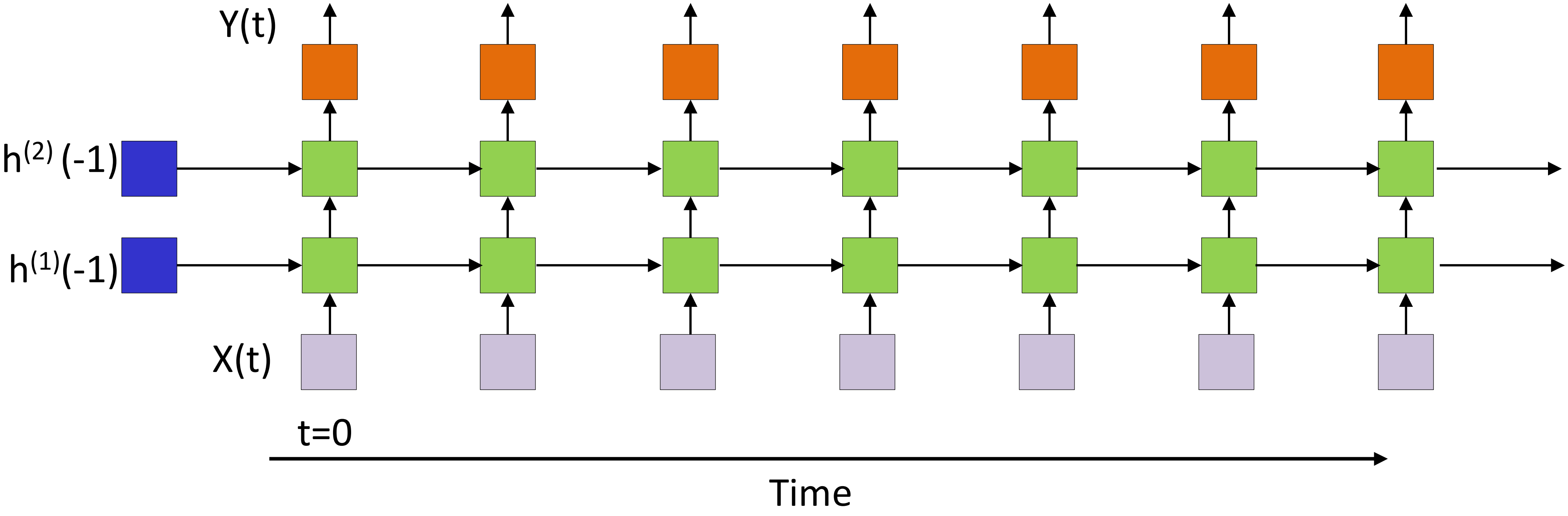
- The state an be arbitrarily complex

Single hidden layer RNN



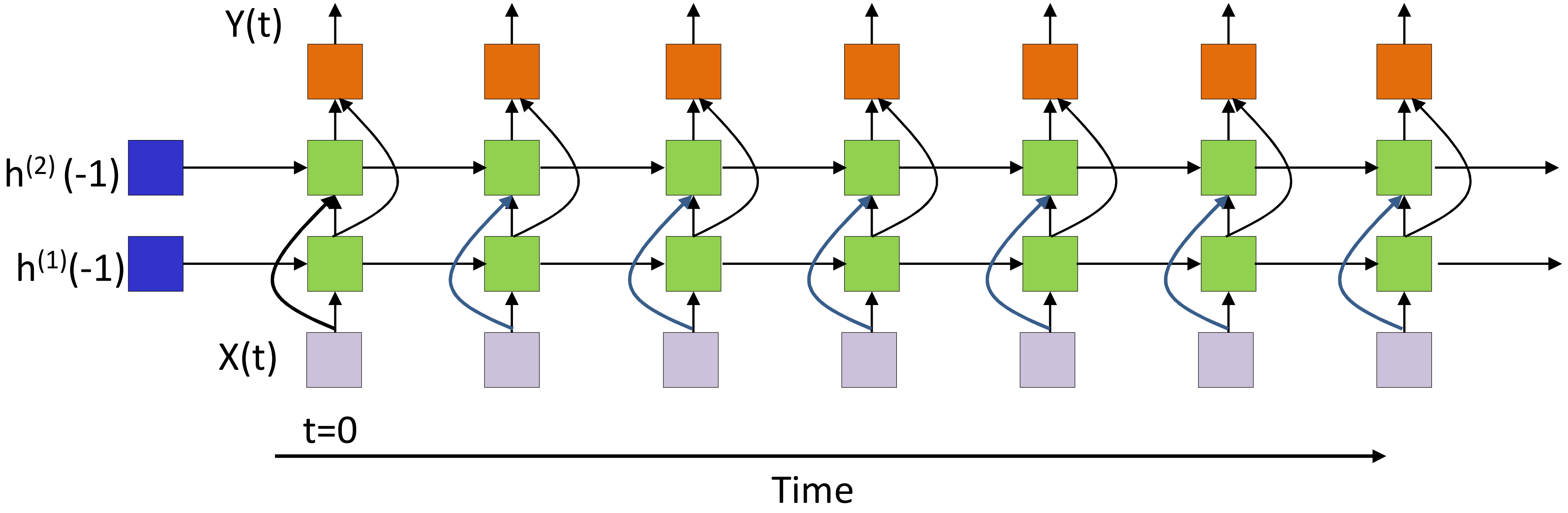
- Recurrent neural network
- All columns are identical
- *An input at t=0 affects outputs forever*

Multiple recurrent layer RNN



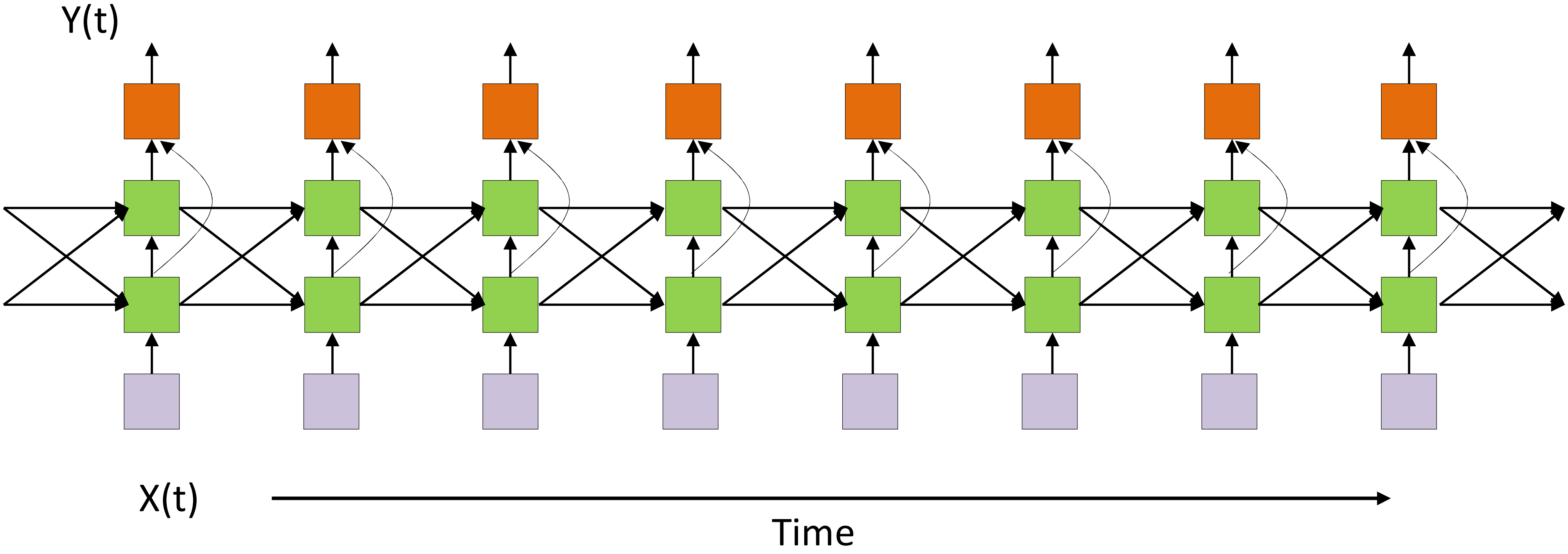
- Recurrent neural network
- All columns are identical
- *An input at t=0 affects outputs forever*

Multiple recurrent layer RNN



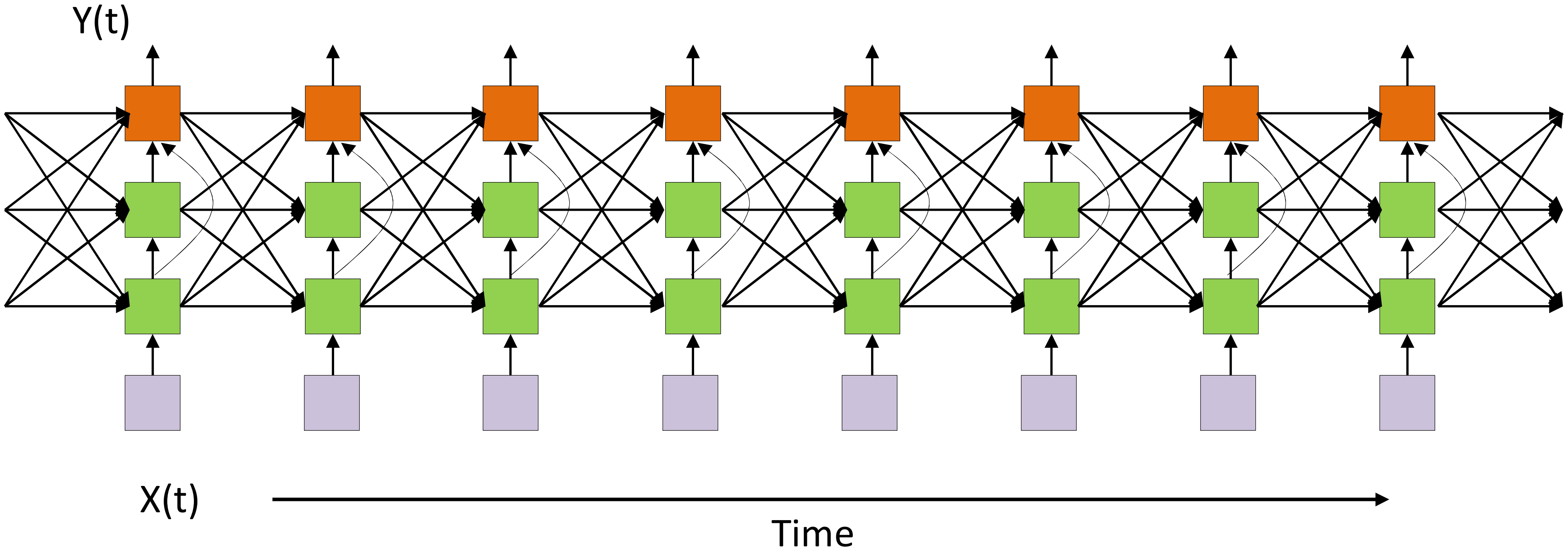
- We can also have skips..

A more complex state



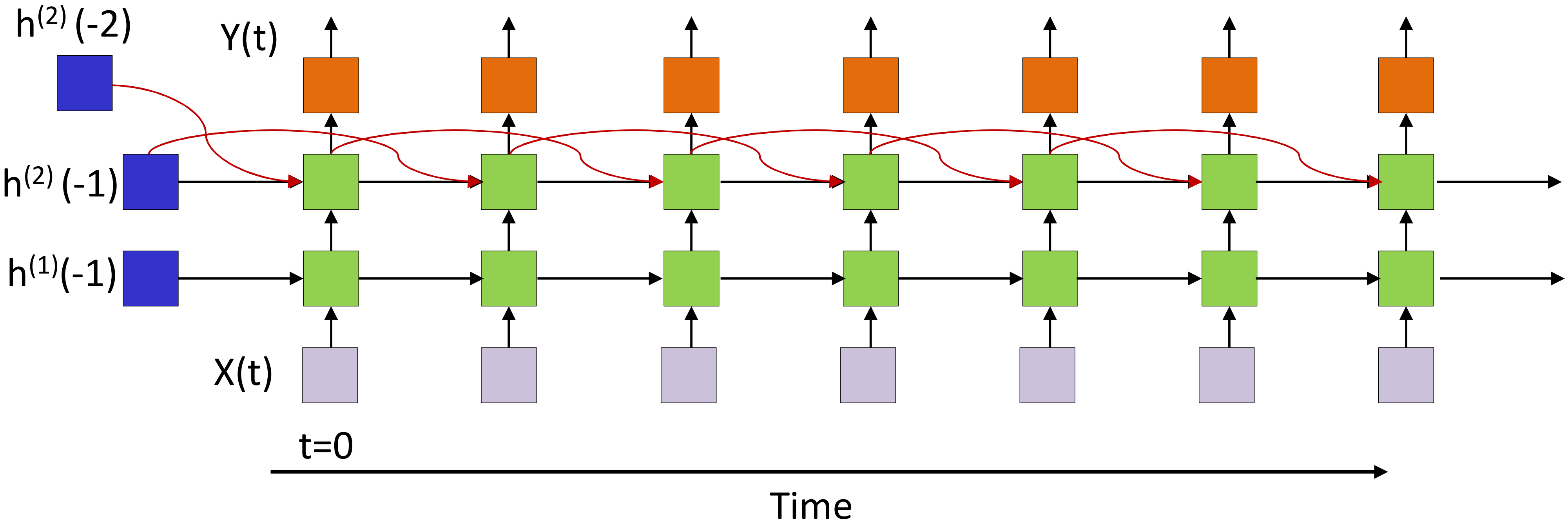
- All columns are identical
- *An input at t=0 affects outputs forever*

Or the network may be even more complicated



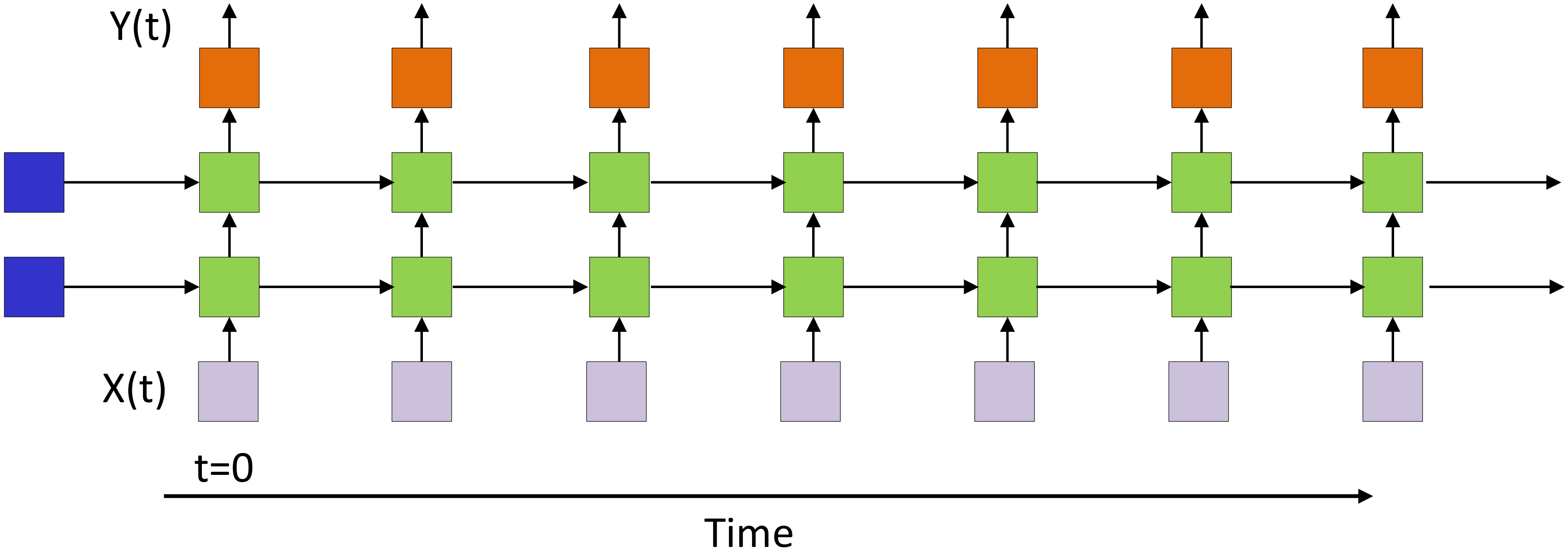
- Shades of NARX
- All columns are identical
- *An input at t=0 affects outputs forever*

Generalization with other recurrences



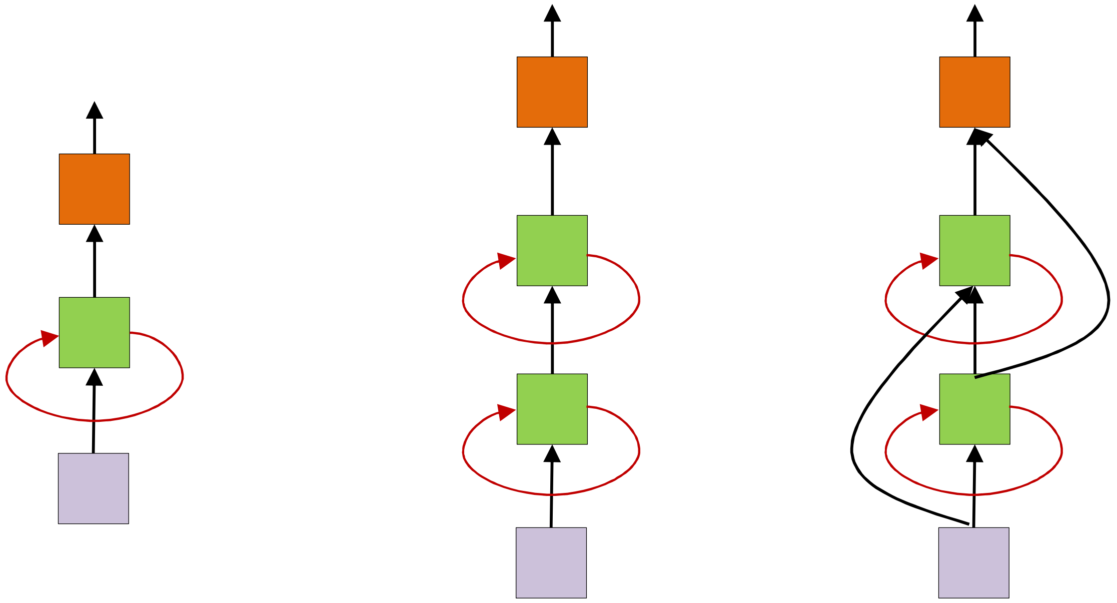
- All columns (including incoming edges) are identical

The simplest structures are most popular



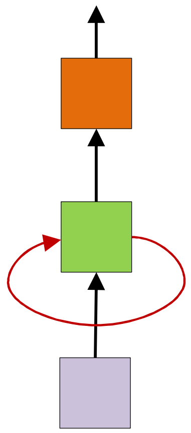
- Recurrent neural network
- All columns are identical
- *An input at t=0 affects outputs forever*

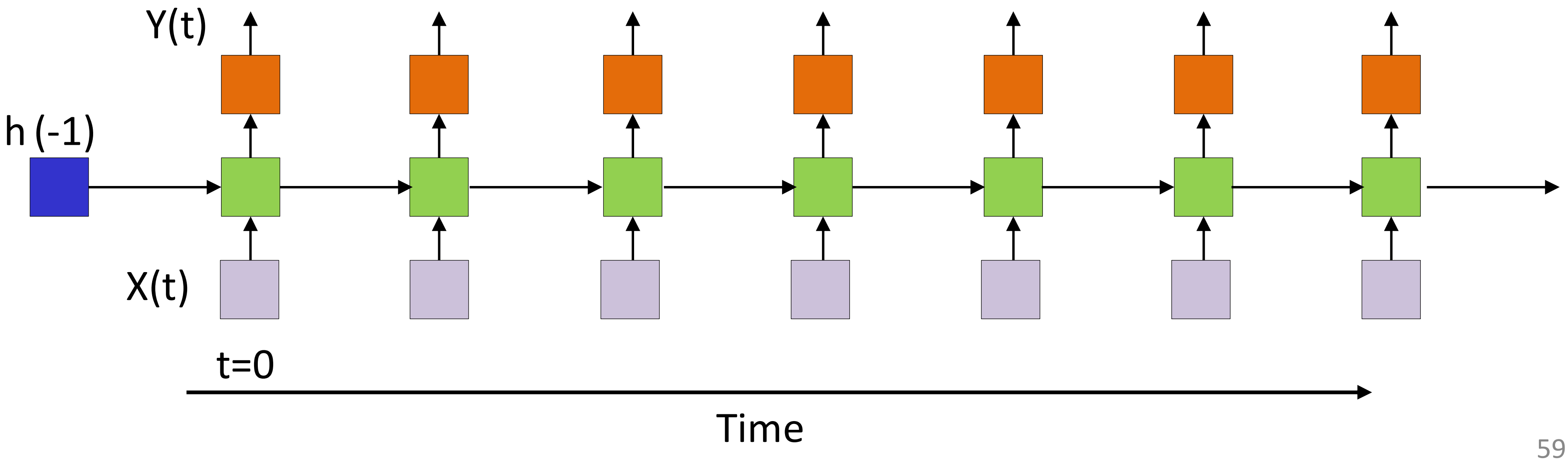
A Recurrent Neural Network



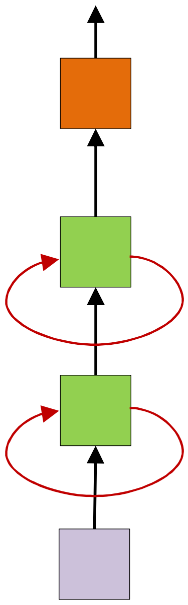
- Simplified models often drawn
- The loops imply recurrence

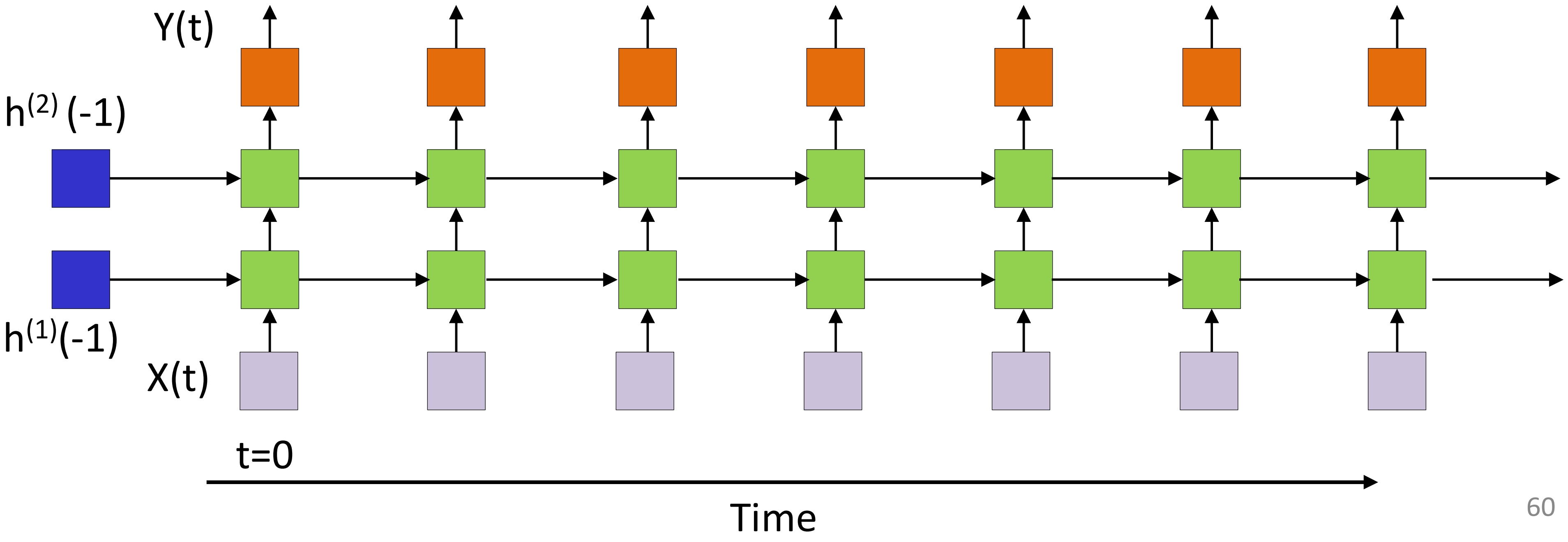
The detailed version of the simplified representation



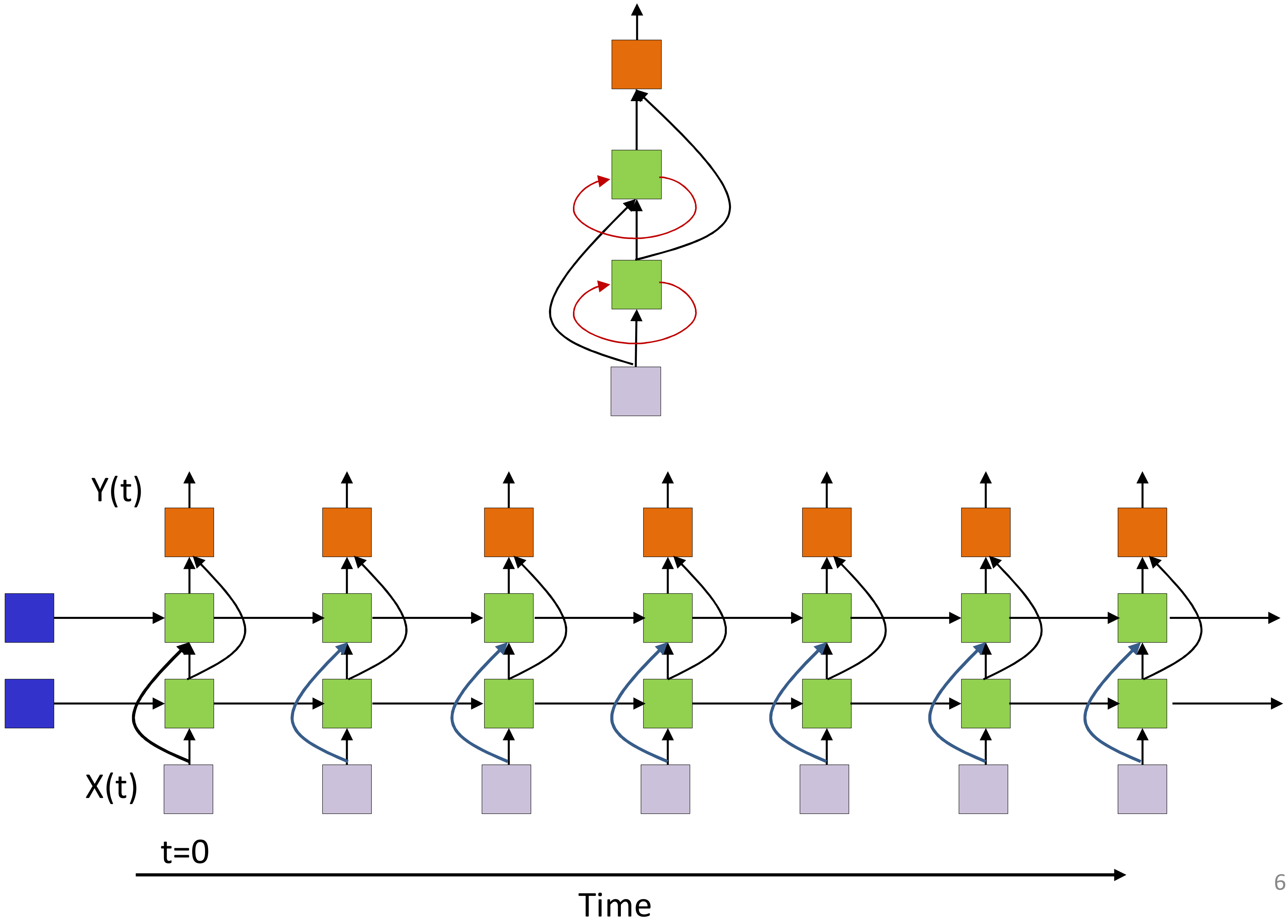


Multiple recurrent layer RNN





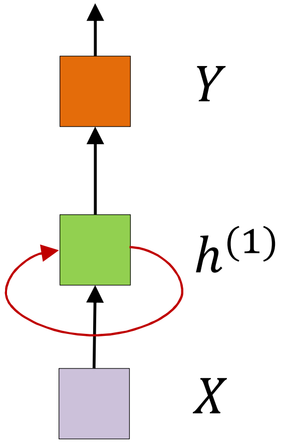
Multiple recurrent layer RNN



Equations

Current weights

Recurrent weights



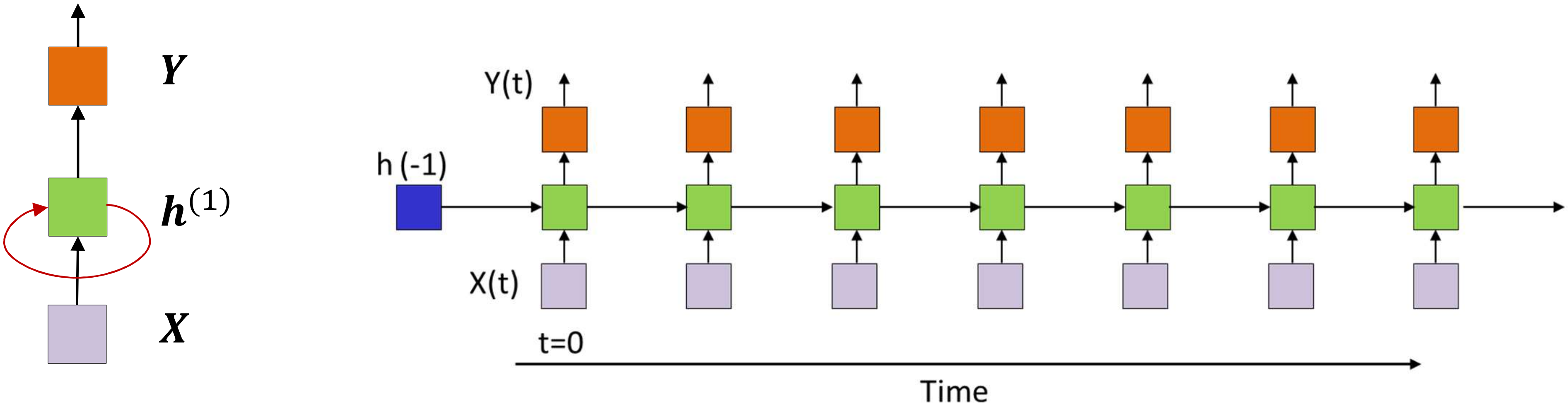
$$h_i^{(1)}(-1) = part\ of\ network\ parameters$$

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

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

- 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



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

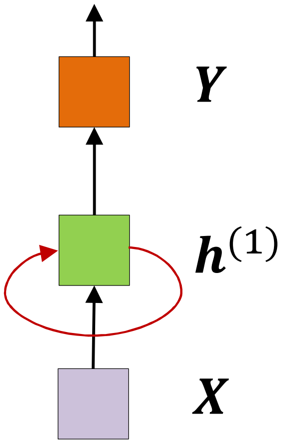
- Computation:

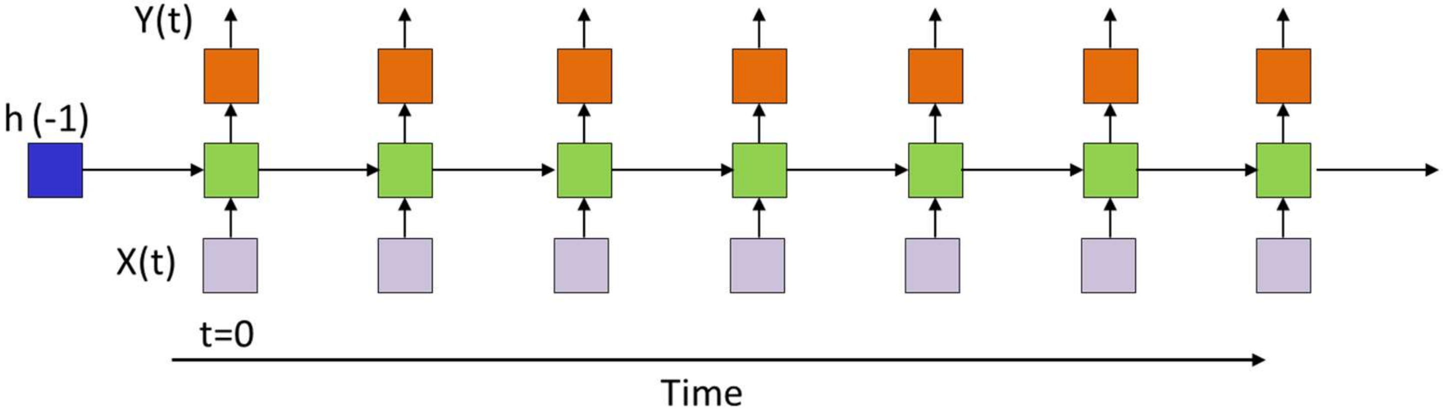
$$\boldsymbol{h}^{(1)}(t) = f_1\left(\boldsymbol{W}^{(1)}\boldsymbol{X}(t) + \boldsymbol{W}^{(11)}\boldsymbol{h}^{(1)}(t-1) + \boldsymbol{b}^{(1)}\right)$$

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

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

Equations





Current weights $= part\ of\ network\ parameters$

Recurrent weights

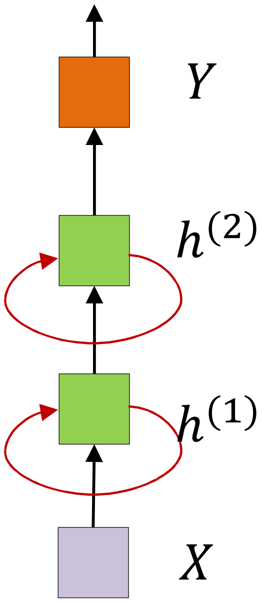
- Computation:

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

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

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

Equations



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

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

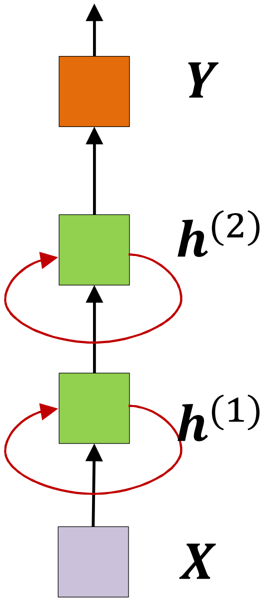
$$h_i^{(1)}(t) = f_1\left(\sum_j w_{ji}^{(1)} X_j(t) + \sum_j w_{ji}^{(11)} h_j^{(1)}(t-1) + b_i^{(1)}\right)$$

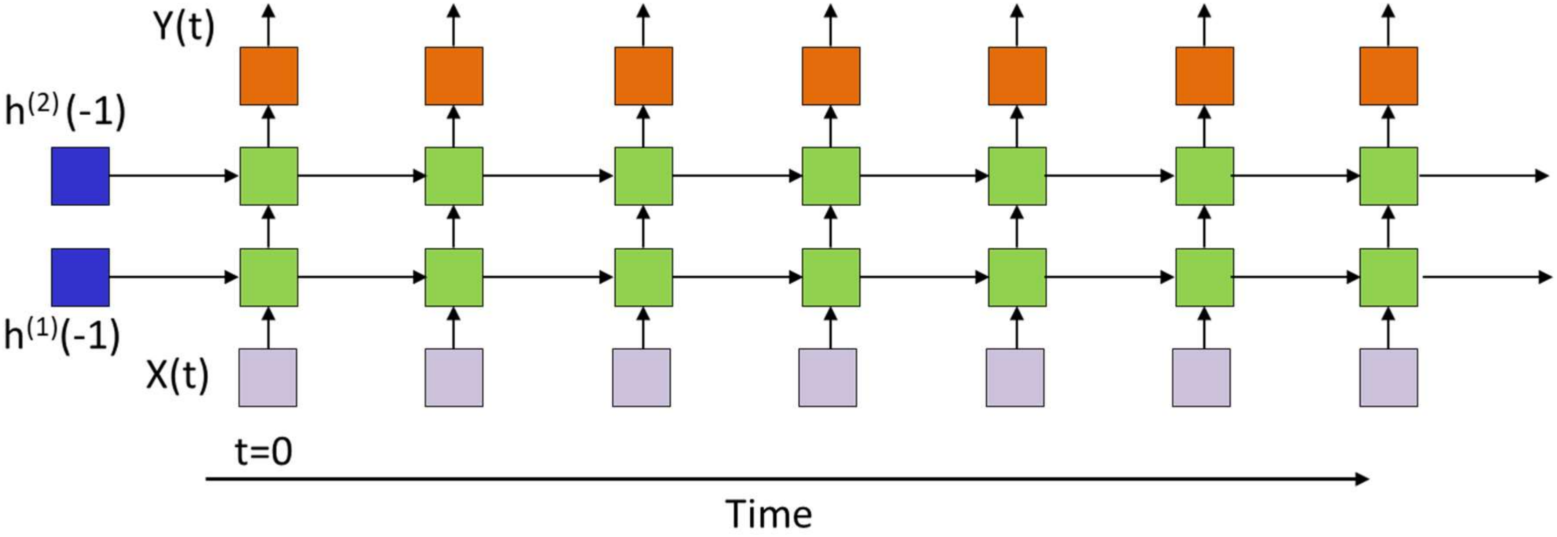
$$h_i^{(2)}(t) = f_2\left(\sum_j w_{ji}^{(2)} h_j^{(1)}(t) + \sum_j w_{ji}^{(22)} h_j^{(2)}(t-1) + b_i^{(2)}\right)$$

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

- 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:

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

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

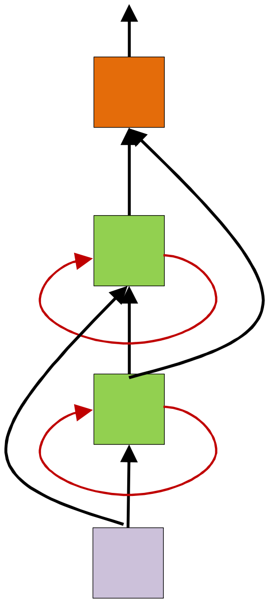
$$\boldsymbol{Y}(t) = f_3\left(\boldsymbol{W}^{(3)}\boldsymbol{h}^{(2)}(t) + \boldsymbol{b}^{(3)}\right)$$

- The recurrent state activation is typically tanh()

Equations

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

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

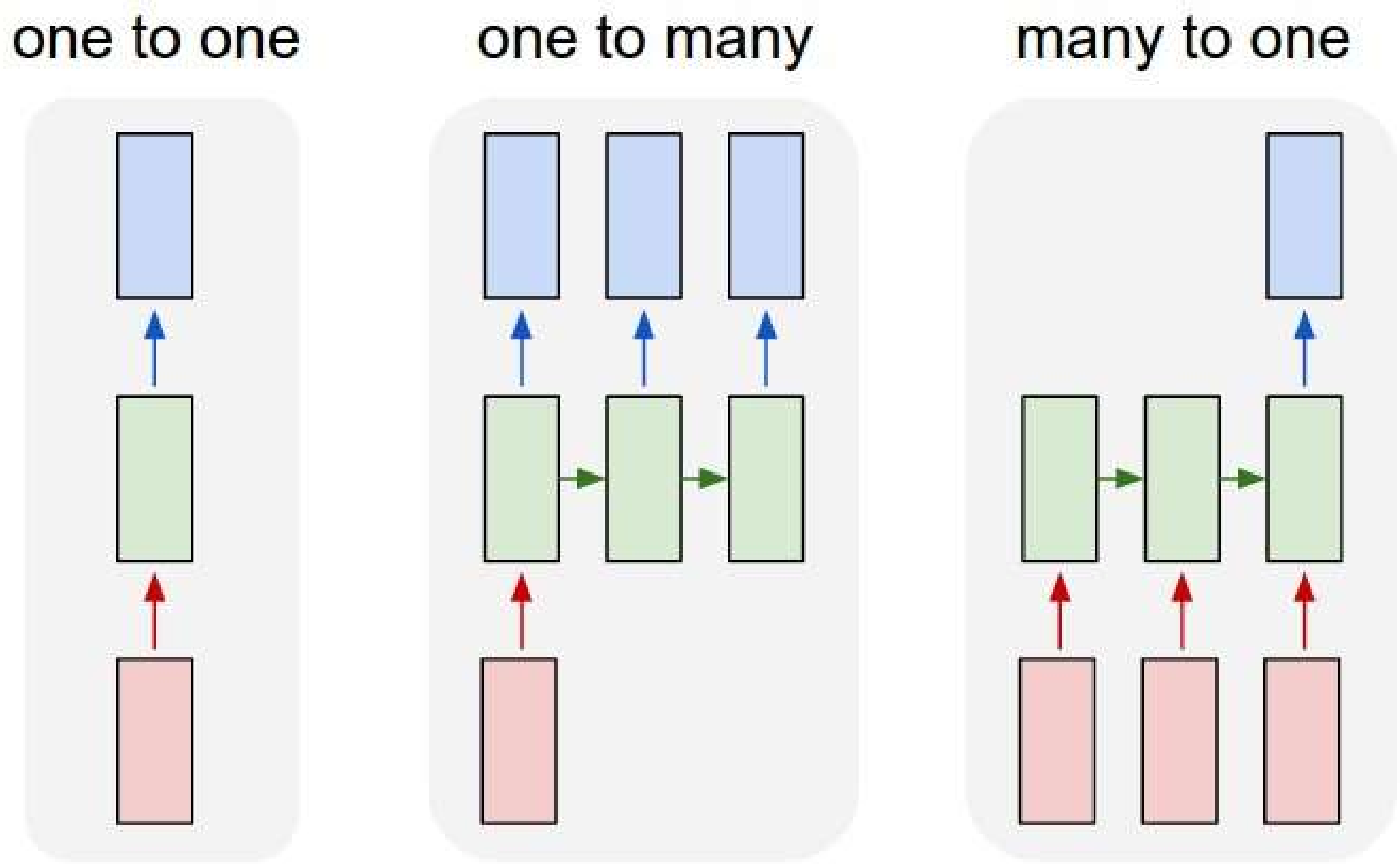


$$\boldsymbol{h}^{(1)}(t) = f_1\left(\boldsymbol{W}^{(01)}\boldsymbol{X}(t) + \boldsymbol{W}^{(11)}\boldsymbol{h}^{(1)}(t-1) + \boldsymbol{b}^{(1)}\right)$$

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

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

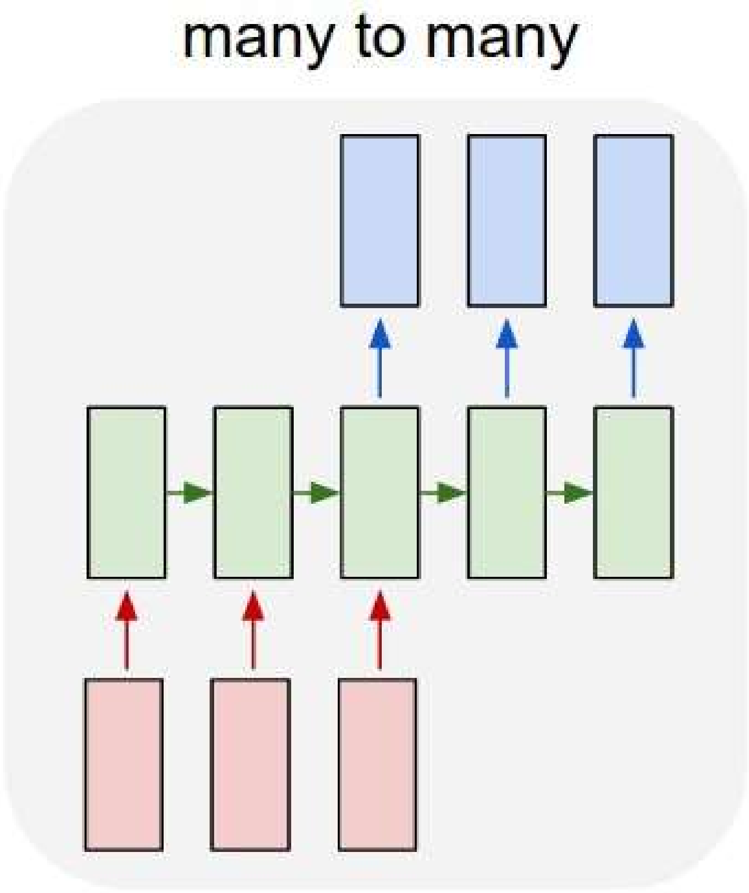
Variants on recurrent nets

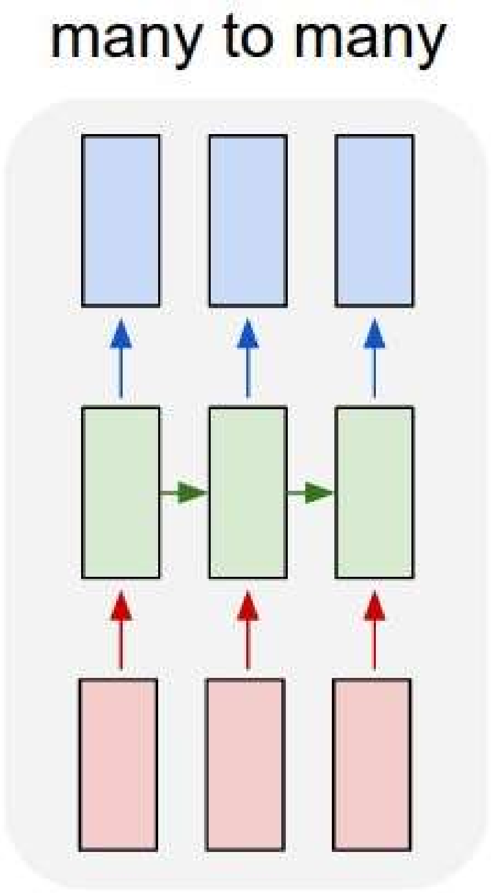


Images from Karpathy

- 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…

Story so far

- Time series analysis must consider past inputs along with current input
- Looking into the infinite past requires recursion

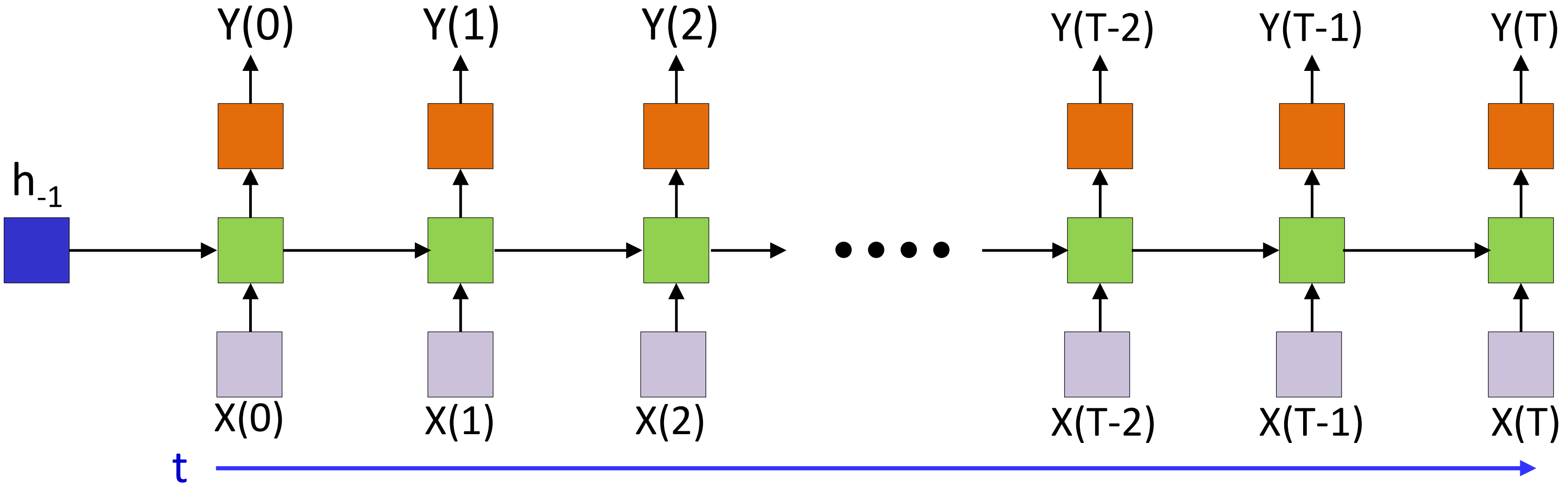
- NARX networks achieve this by feeding back the output to the input

- “Simple” recurrent networks maintain separate “memory” or “context” units to retain some information about the past
  - But during learning the current error does not influence the past

- State-space models retain information about the past through recurrent hidden states
  - These are “fully recurrent” networks
  - The initial values of the hidden states are generally learnable parameters as well

- State-space models enable current error to update parameters in the past

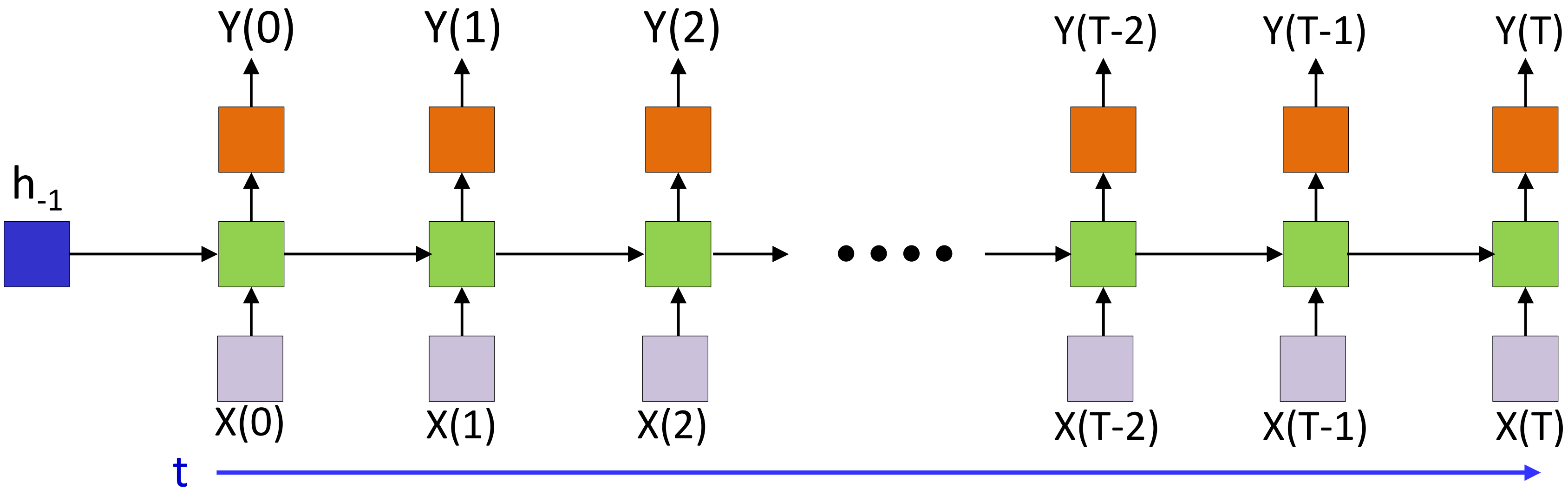
How do we *train* the network



- Back propagation through time (BPTT)

- Given a collection of *sequence* inputs
  - $(\mathbf{X}_i, \mathbf{D}_i)$, where
  - $\mathbf{X}_i = X_{i,0}, \dots, X_{i,T}$
  - $\mathbf{D}_i = D_{i,0}, \dots, D_{i,T}$
- Train network parameters to minimize the error between the output of the network $\mathbf{Y}_i = Y_{i,0}, \dots, Y_{i,T}$ and the desired outputs
  - This is the most generic setting. In other settings we just “remove” some of the input or output entries

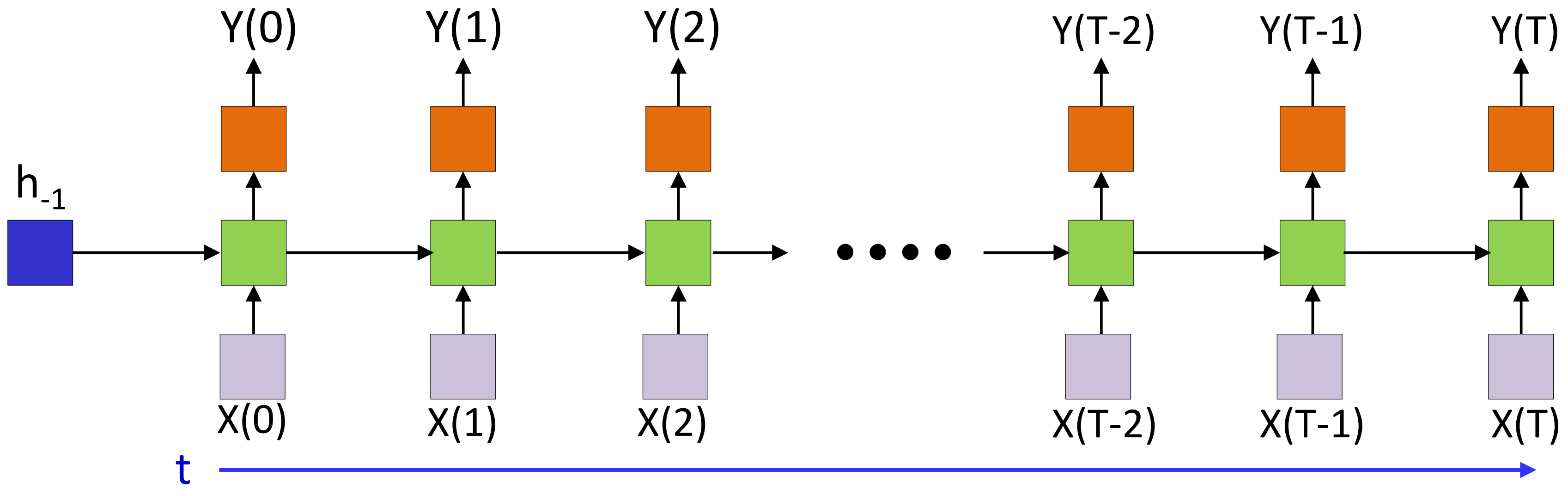
Training the RNN



- 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



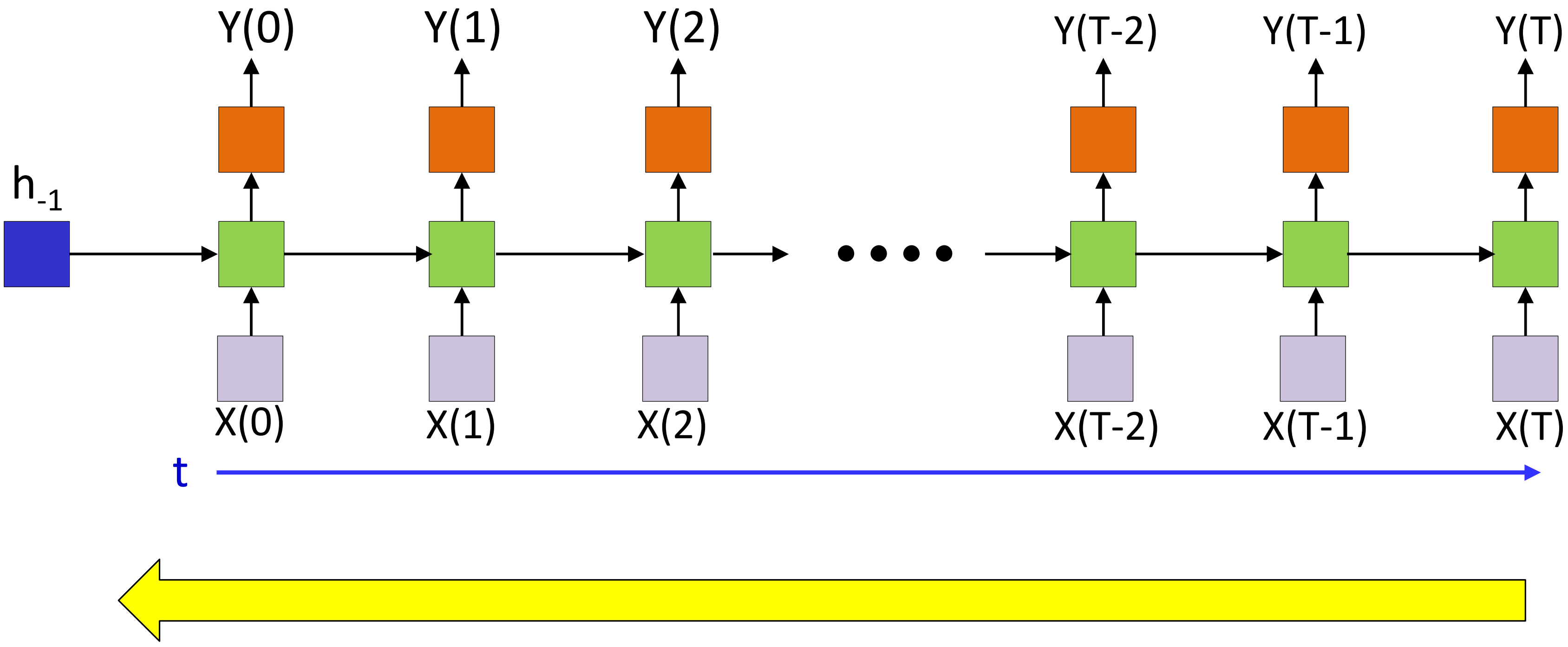
- For each training input:
- Forward pass: pass the entire data sequence through the network, generate outputs

Recurrent Neural Net
Assuming time-synchronous output

```
# Assuming h(-1,*) is known
# Assuming L hidden-state layers and an output layer
# $W_c(*)$ and $W_r(*)$ are matrics, b(*) are vectors
# $W_c$ are weights for inputs from current time
# $W_r$ is recurrent weight applied to the previous time
# $W_o$ are output layre weights

for t = 0:T-1  # Including both ends of the index
    h(t,0) = x(t) # Vectors. Initialize h(0) to input
    for l = 1:L  # hidden layers operate at time t
        z(t,l) = $W_c$(l)h(t,l-1) + $W_r$(l)h(t-1,l) + b(l)
        h(t,l) = tanh(z(t,l)) # Assuming tanh activ.
    $z_o$(t) = $W_o$h(t,L) + $b_o$
    Y(t) = softmax( $z_o$(t) )
```

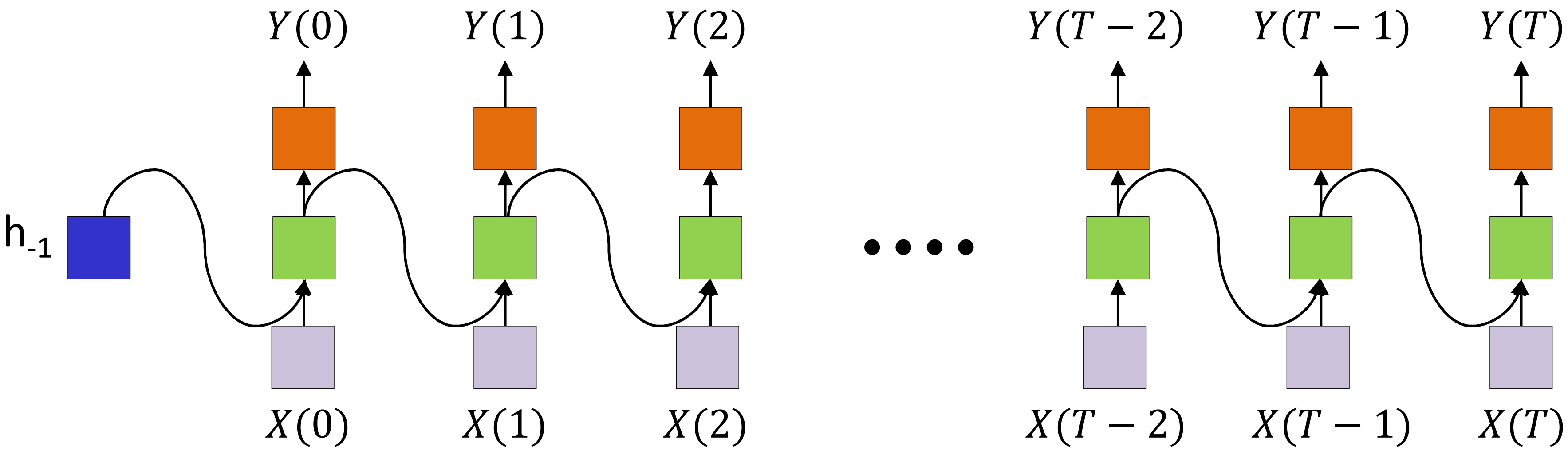
Subscript "c" - current
Subscript "r" - recurrent

Training: Computing gradients



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

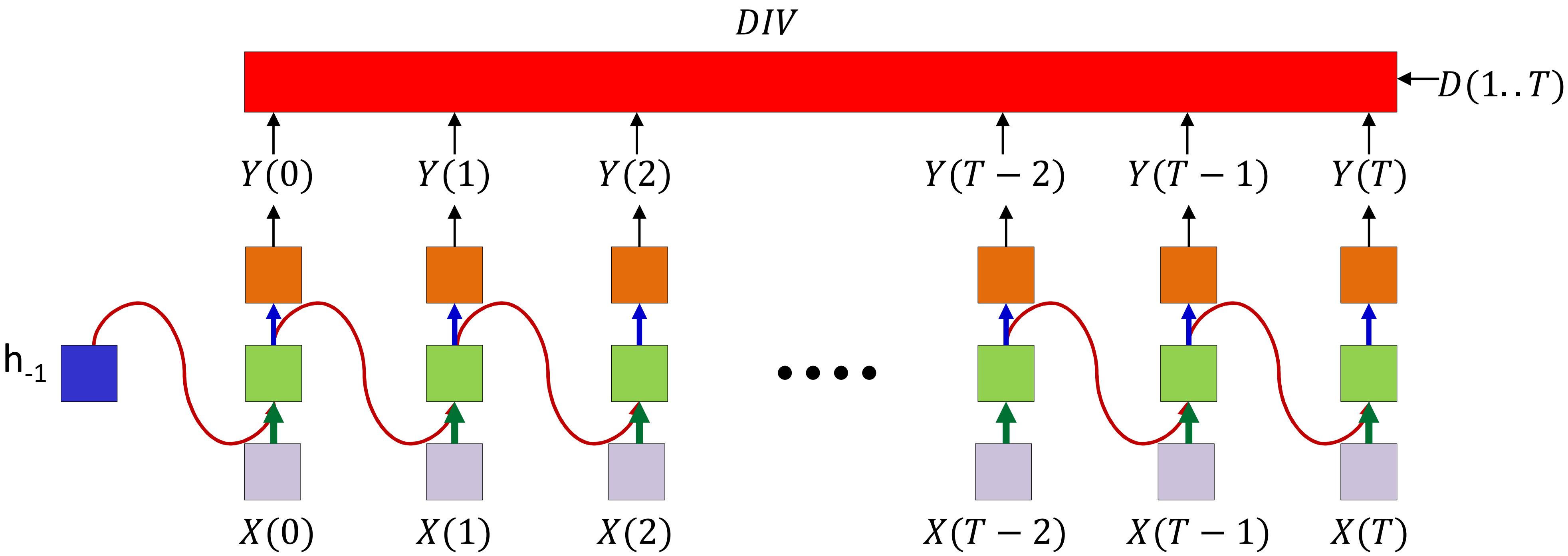
Back Propagation Through Time



Will only focus on *one* training instance

All subscripts represent *components* and not training instance index

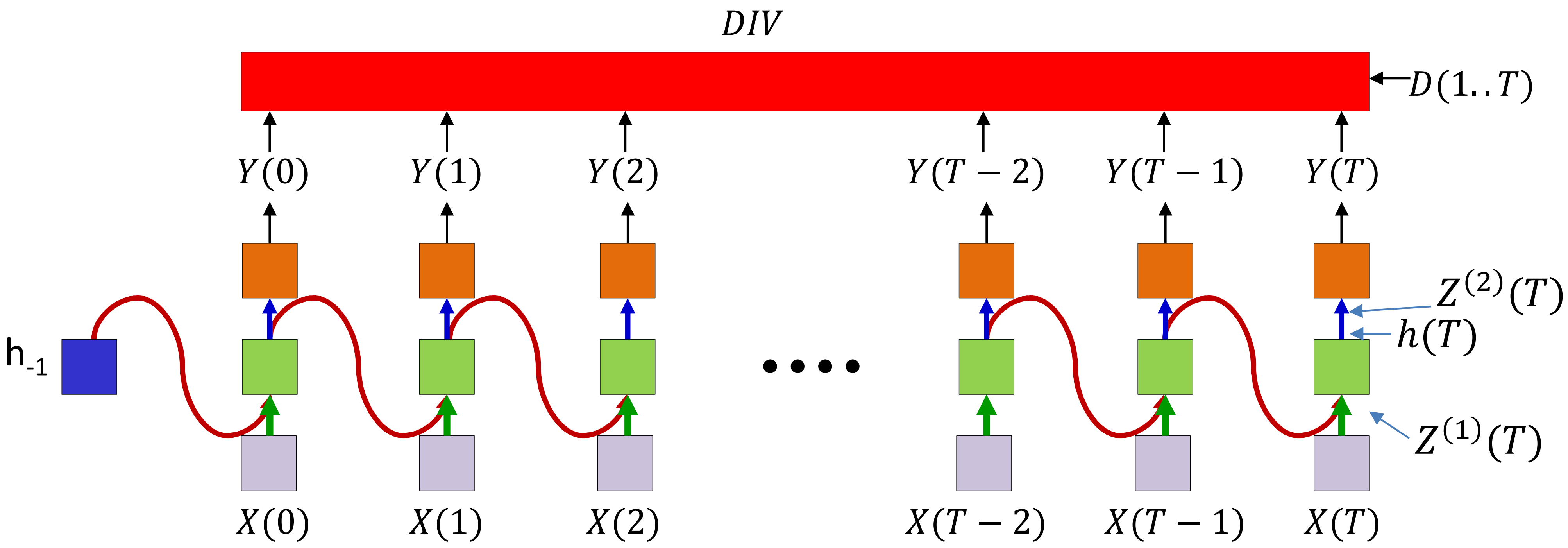
Back Propagation Through Time



- 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!

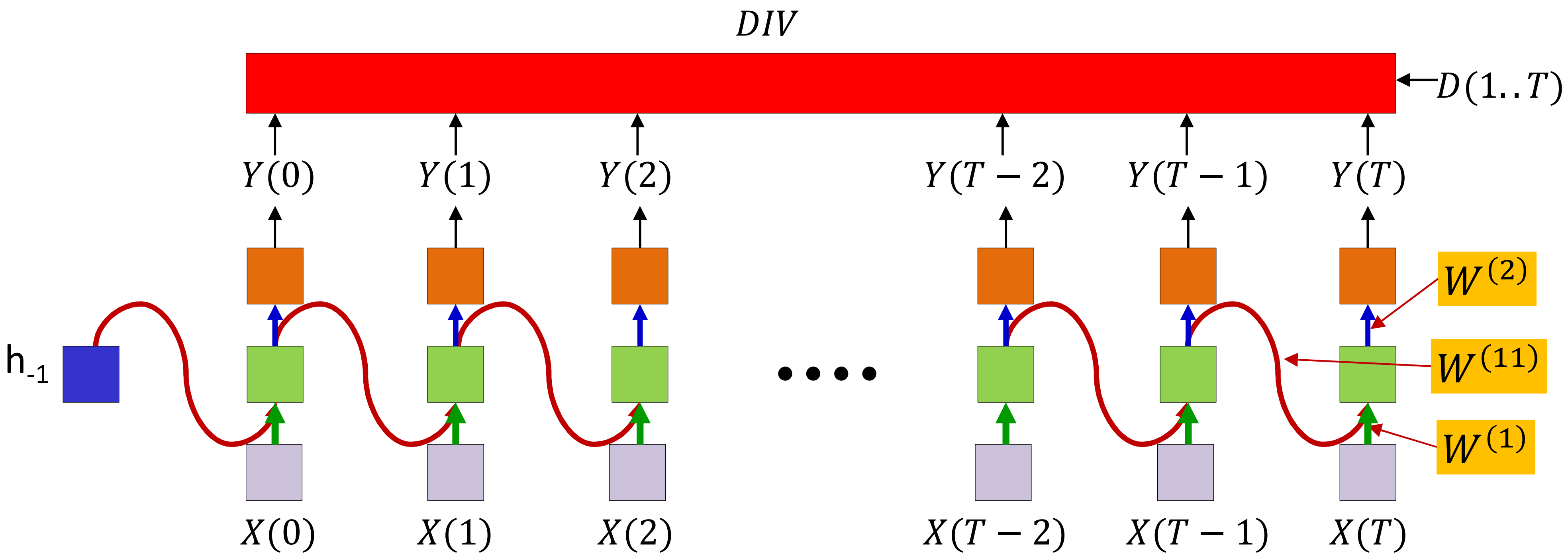
- This is *not* just the sum of the divergences at individual times
  - Unless we explicitly define it that way

Notation



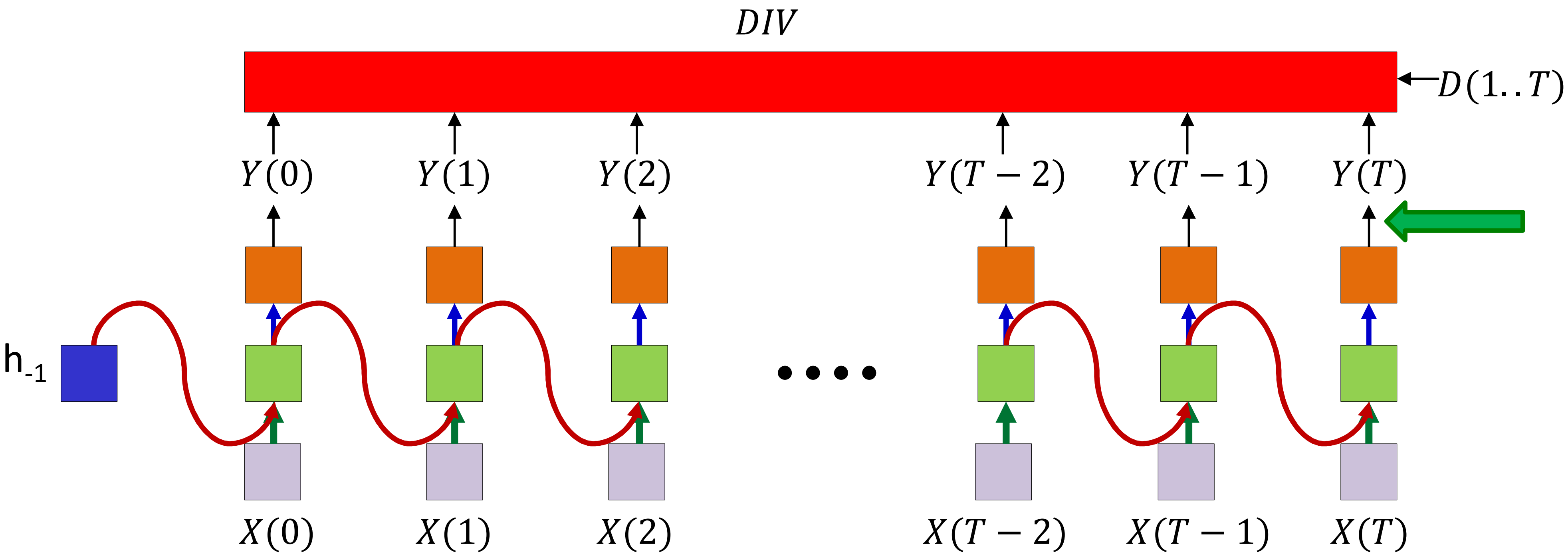
- $Y(t)$ is the output at time t
  - $Y_i(t)$ is the ith output
- $Z^{(2)}(t)$ is the pre-activation value of the neurons at the output layer at time t
- $h(t)$ is the output of the hidden layer at time t
  - Assuming only one hidden layer in this example
- $Z^{(1)}(t)$ is the pre-activation value of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

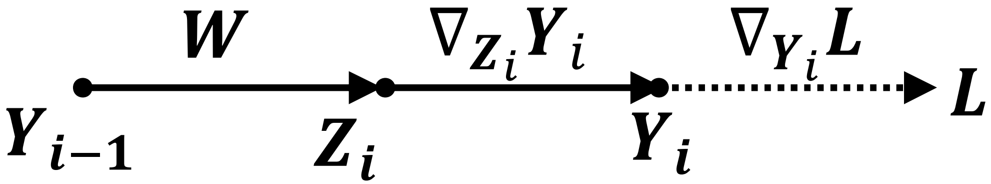


First step of backprop: Compute $\nabla_{Y(T)} DIV$ (Compute $\frac{dDIV}{dY_i(T)}$ for all i)

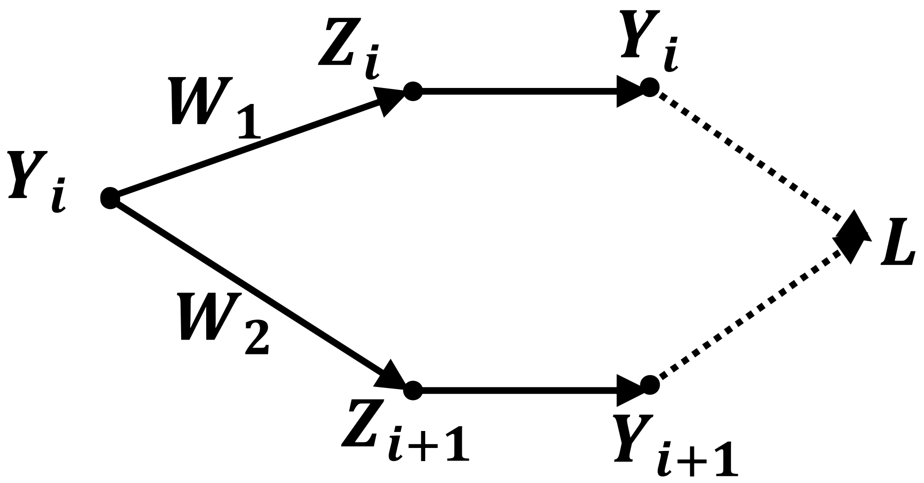
Note: DIV is a function of *all* outputs Y(0) … Y(T)

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

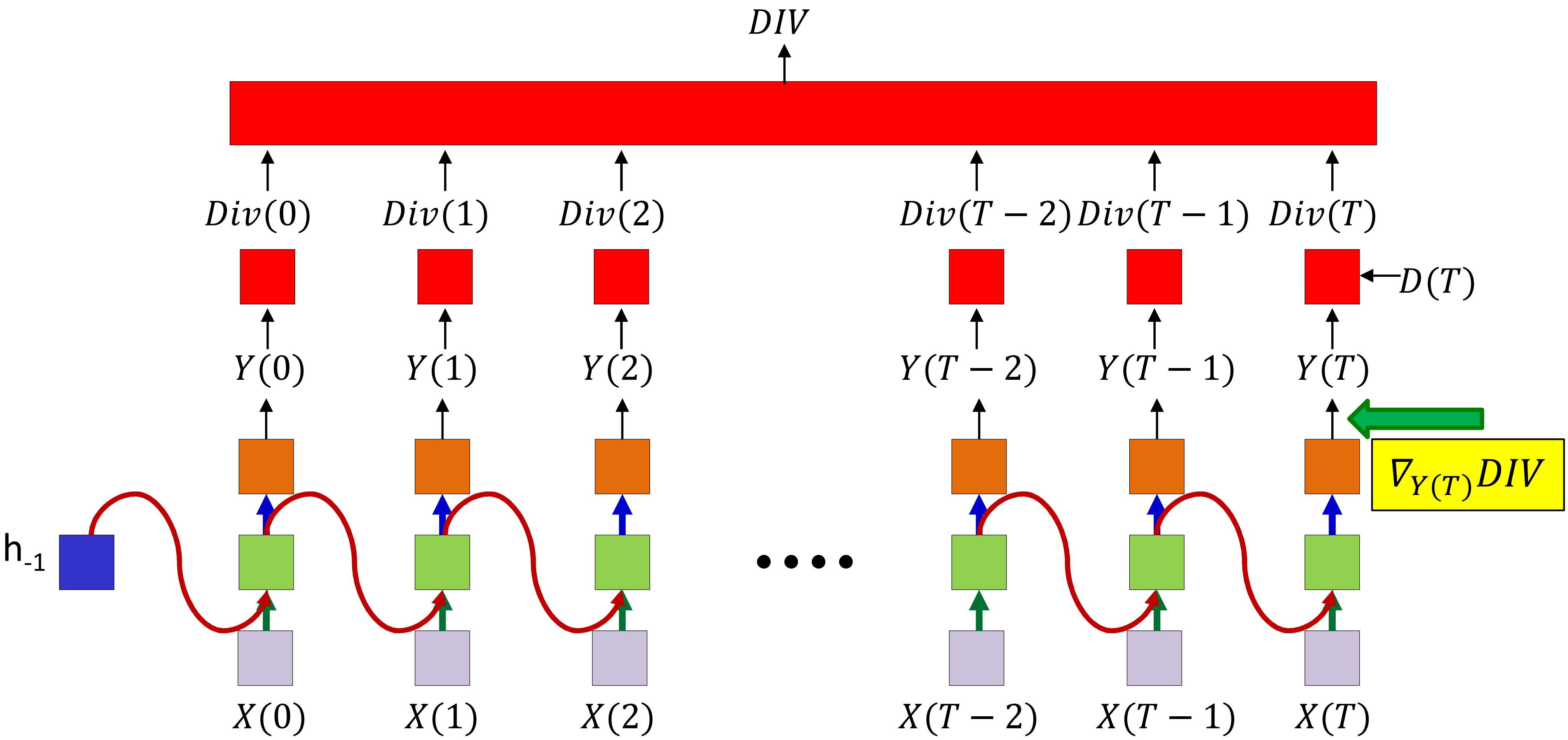
Rules we will use



- $\nabla_{Z_i}L = \nabla_{Y_i}L\,\nabla_{Z_i}\boldsymbol{Y}_i$
- $\nabla_{Y_{i-1}}L = \nabla_{Z_i}L\,\boldsymbol{W}$
- $\nabla_{W}L = \boldsymbol{Y}_{i-1}\nabla_{Z_i}L$



- $\nabla_{Y_i}L = \nabla_{Z_i}L\,\boldsymbol{W}_1 + \nabla_{Z_{i+1}}L\,\boldsymbol{W}_2$



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

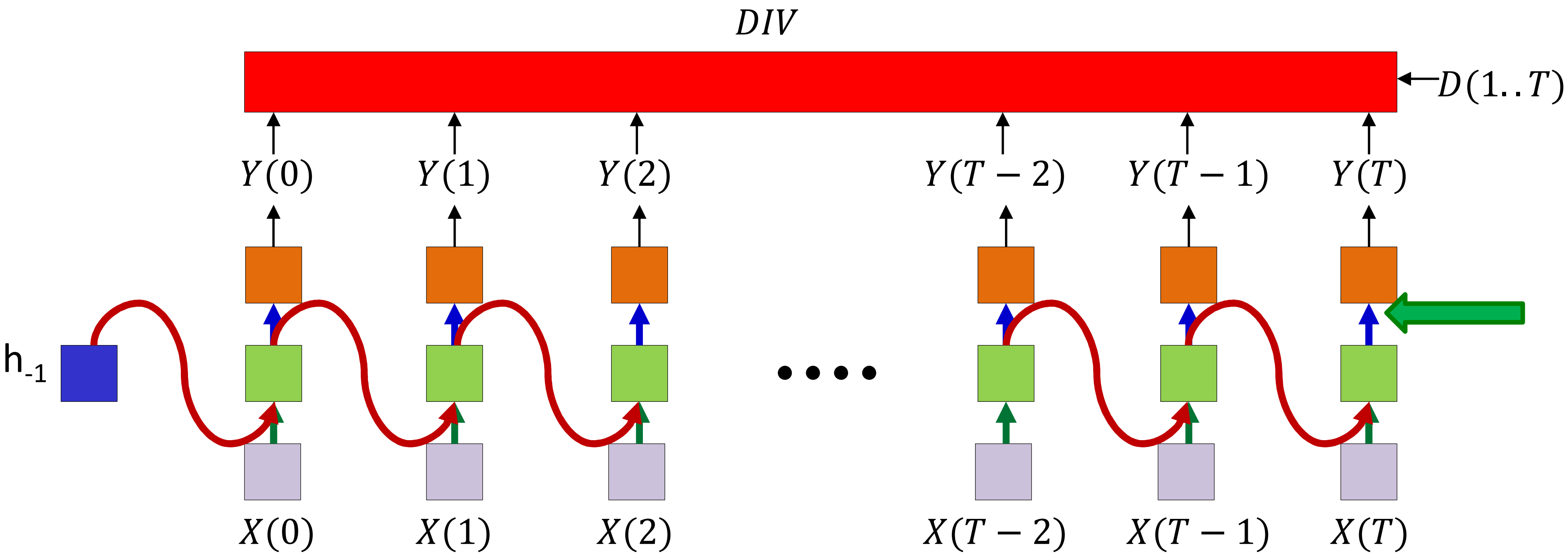
Must compute $\nabla_{Y(t)} DIV$

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

$\frac{\partial DIV}{\partial Y_i(t)}$ for all i for all T

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

Back Propagation Through Time



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

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

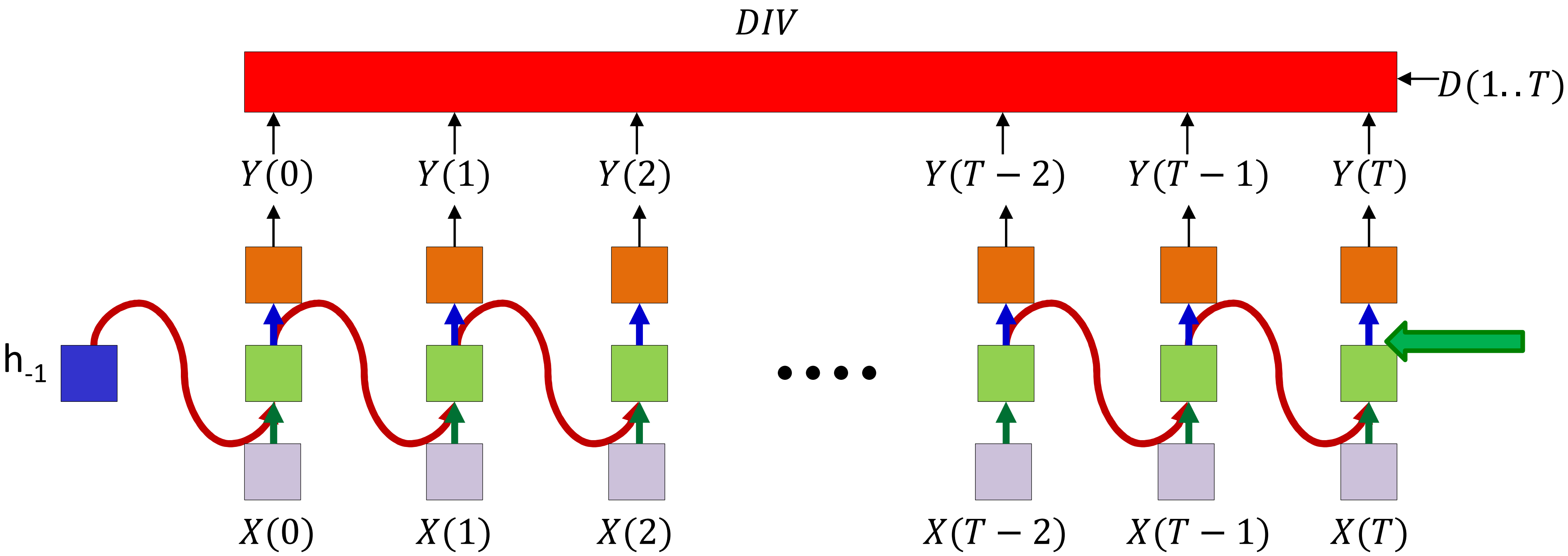
Vector output activation

$$\frac{dDIV}{dZ_i^{(2)}(T)} = \frac{dDIV}{dY_i(T)}\frac{dY_i(T)}{dZ_i^{(2)}(T)}$$

OR

$$\frac{dDIV}{dZ_i^{(2)}(T)} = \sum_j \frac{dDIV}{dY_j(T)}\frac{dY_j(T)}{dZ_i^{(2)}(T)}$$

Back Propagation Through Time



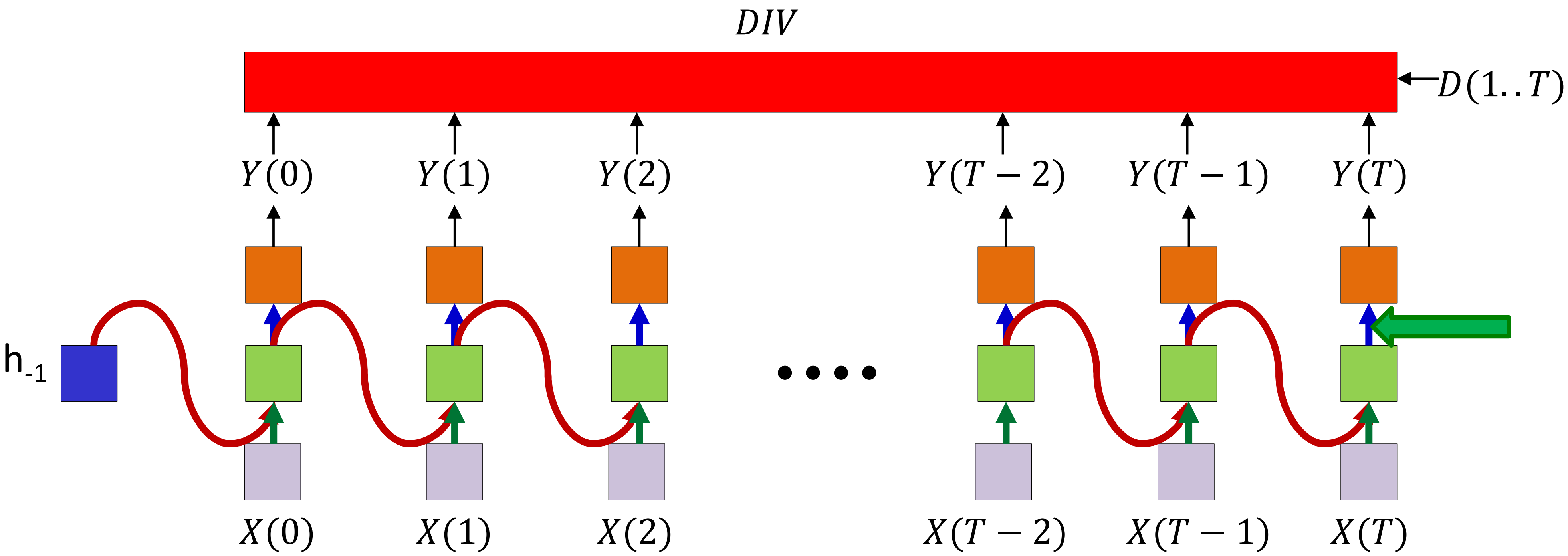
$$\frac{dDIV}{dY_i(T)} \text{ for all i}$$

$$\frac{dDIV}{dZ_i^{(2)}(T)} = \frac{dDiv(T)}{dY_i(T)} \frac{dY_i(T)}{dZ_i^{(2)}(T)}$$

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

$$\nabla_{h(T)} DIV = \nabla_{Z^{(2)}(T)} DIV \, W^{(2)}$$

Back Propagation Through Time



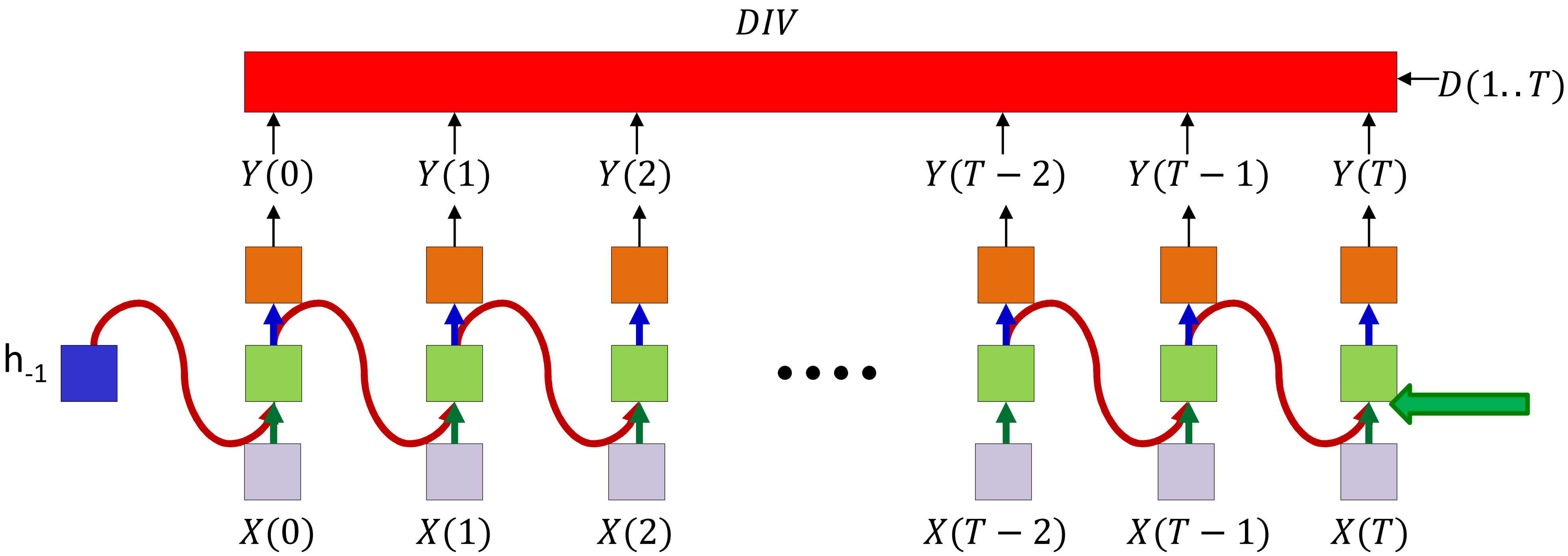
$$\frac{dDIV}{dZ_i^{(2)}(T)} = \frac{dDiv(T)}{dY_i(T)} \frac{dY_i(T)}{dZ_i^{(2)}(T)}$$

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

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

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

Back Propagation Through Time



jacobian

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

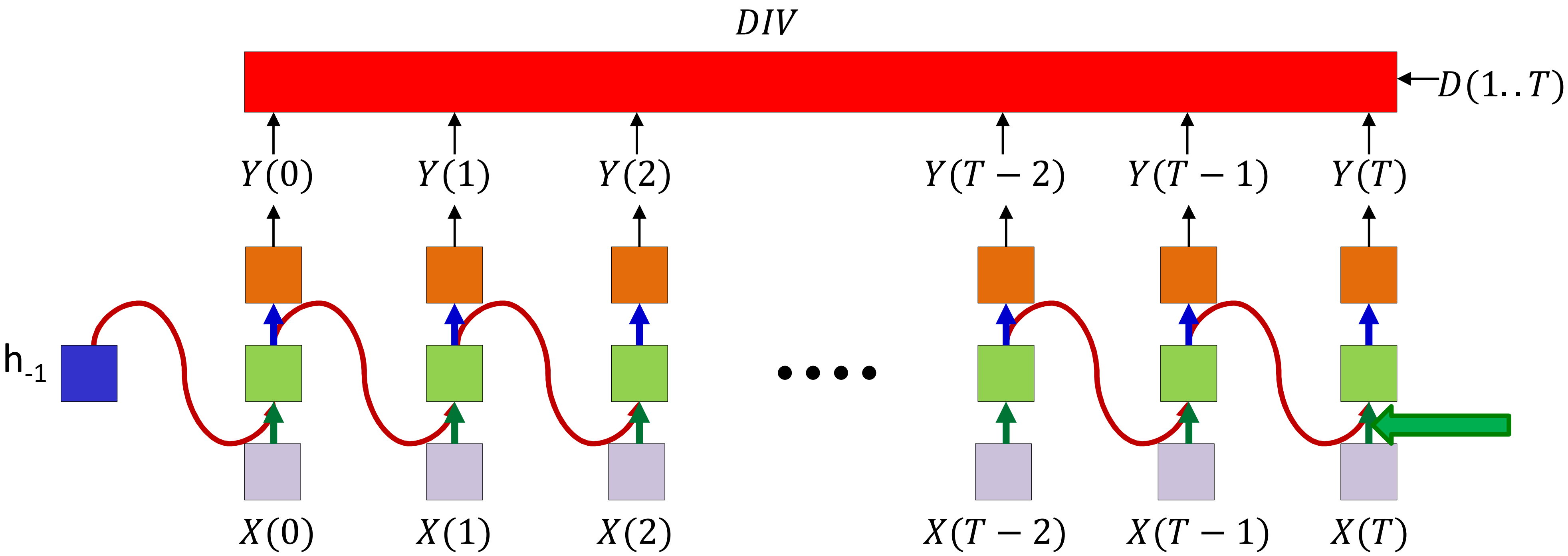
$$\frac{dDIV}{dZ_i^{(1)}(T)} = \frac{dDIV}{dh_i(T)} \frac{dh_i(T)}{dZ_i^{(1)}(T)}$$

$$\frac{dDIV}{dZ_i^{(2)}(T)} = \frac{dDIV}{dY_i(T)} \frac{dY_i(T)}{dZ_i^{(2)}(T)}$$

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

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

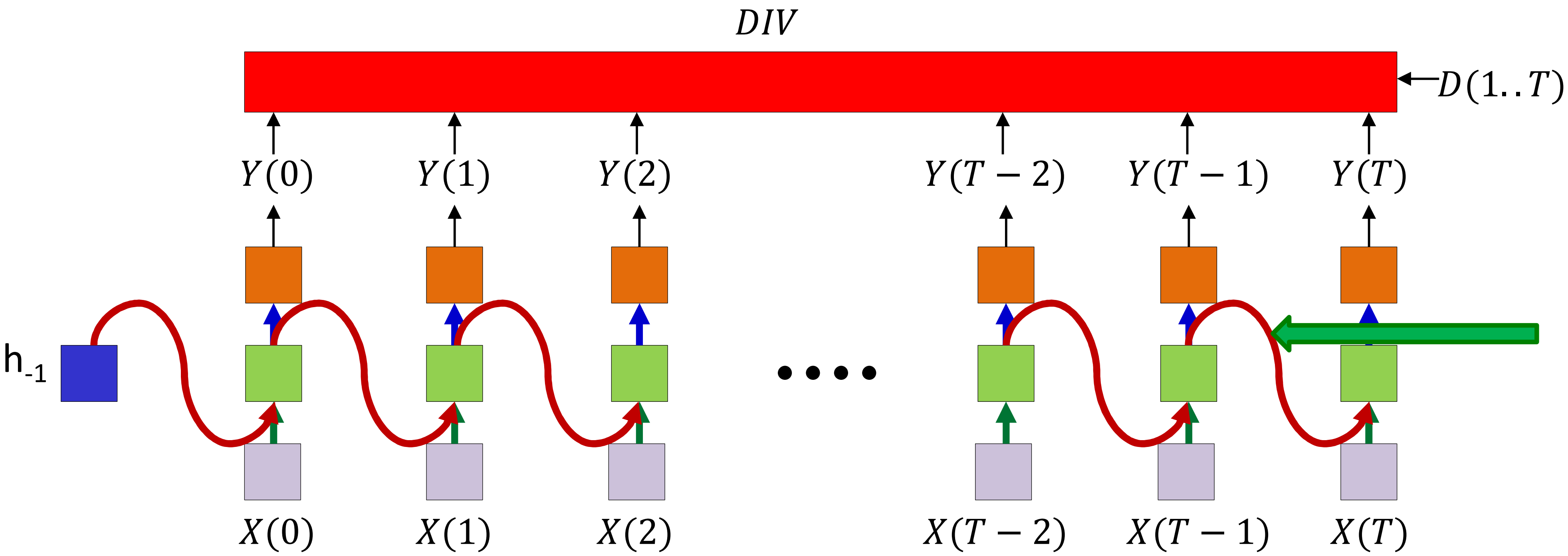
Back Propagation Through Time



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

$$\nabla_{W^{(1)}} DIV = X(T) \nabla_{Z^{(1)}(T)} DIV$$

Back Propagation Through Time

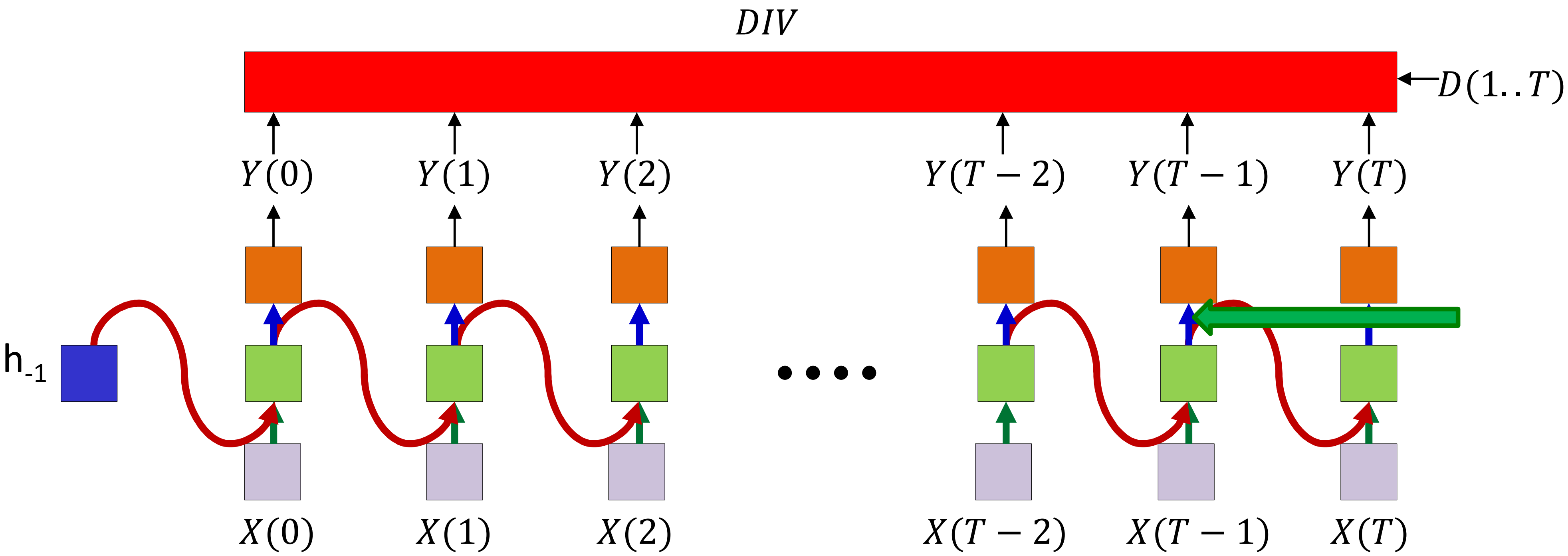


$$\nabla_{W^{(11)}} DIV = h(T-1)\nabla_{Z^{(1)}(T)} DIV$$

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

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

Back Propagation Through Time



$$\nabla_{Z^{(2)}(T-1)}DIV = \nabla_{Y(T-1)}DIV \, \nabla_{Z^{(2)}(T-1)}Y(T-1)$$

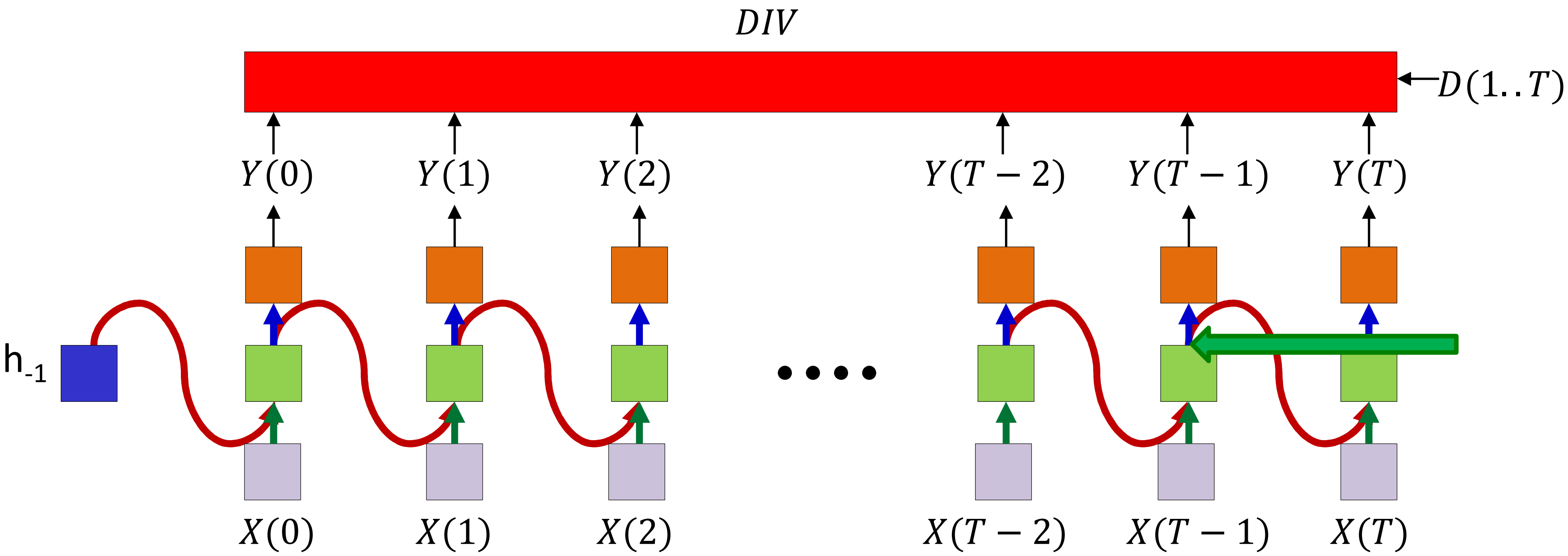
Vector output activation

$$\frac{dDIV}{dZ_i^{(2)}(T-1)} = \frac{dDIV}{dY_i(T-1)} \frac{dY_i(T-1)}{dZ_i^{(2)}(T-1)}$$

OR

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

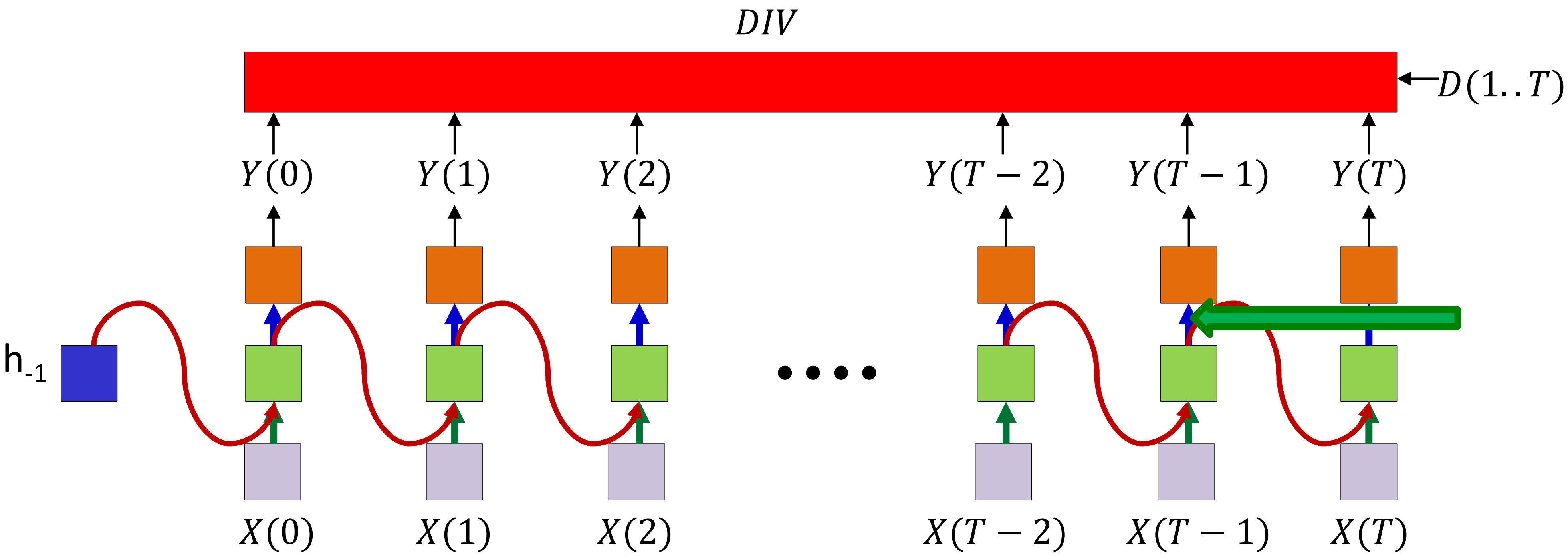
Back Propagation Through Time



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

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

Back Propagation Through Time



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

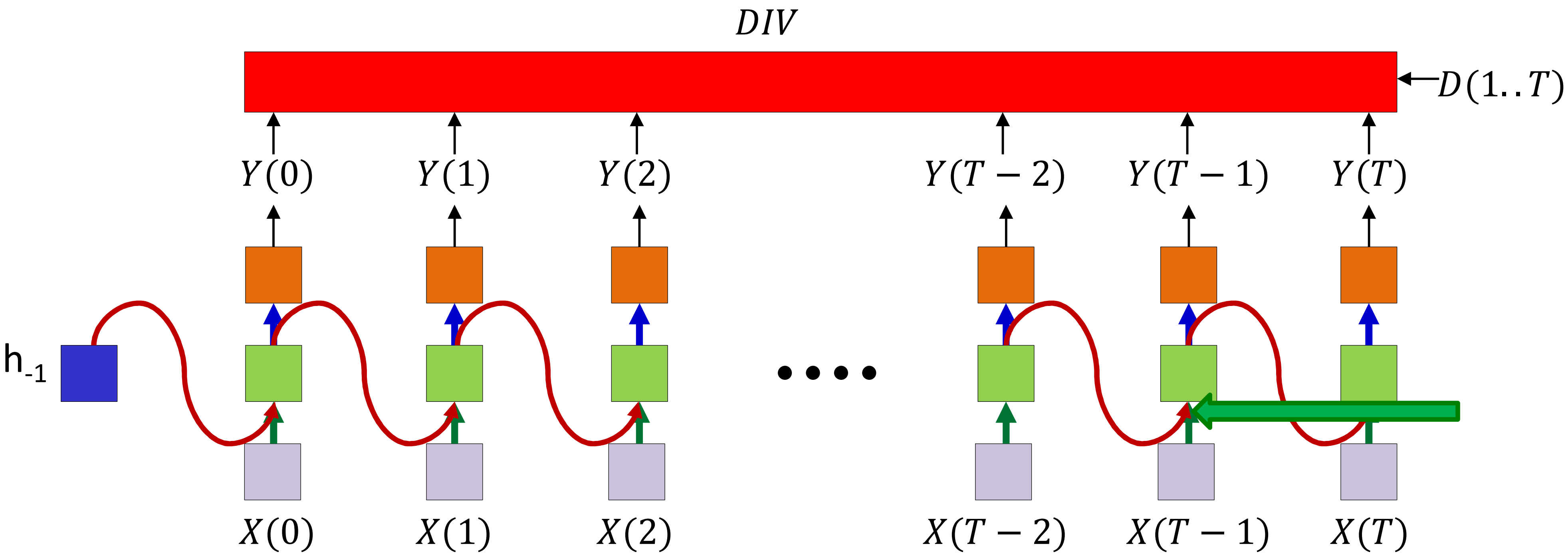
Note the addition

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

Note the addition

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

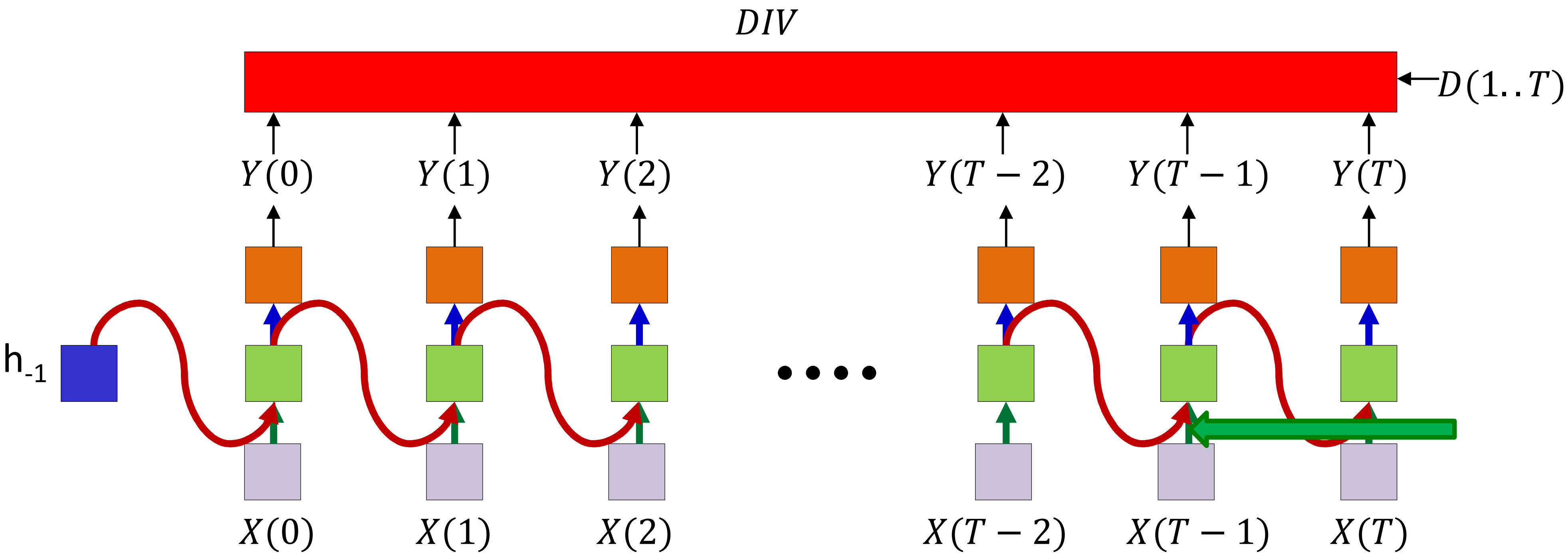
Back Propagation Through Time



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

$$\boxed{\nabla_{Z^{(1)}(T-1)}DIV = \nabla_{h(T-1)}DIV\ \nabla_{Z^{(1)}(T-1)}h(T-1)}$$

Back Propagation Through Time



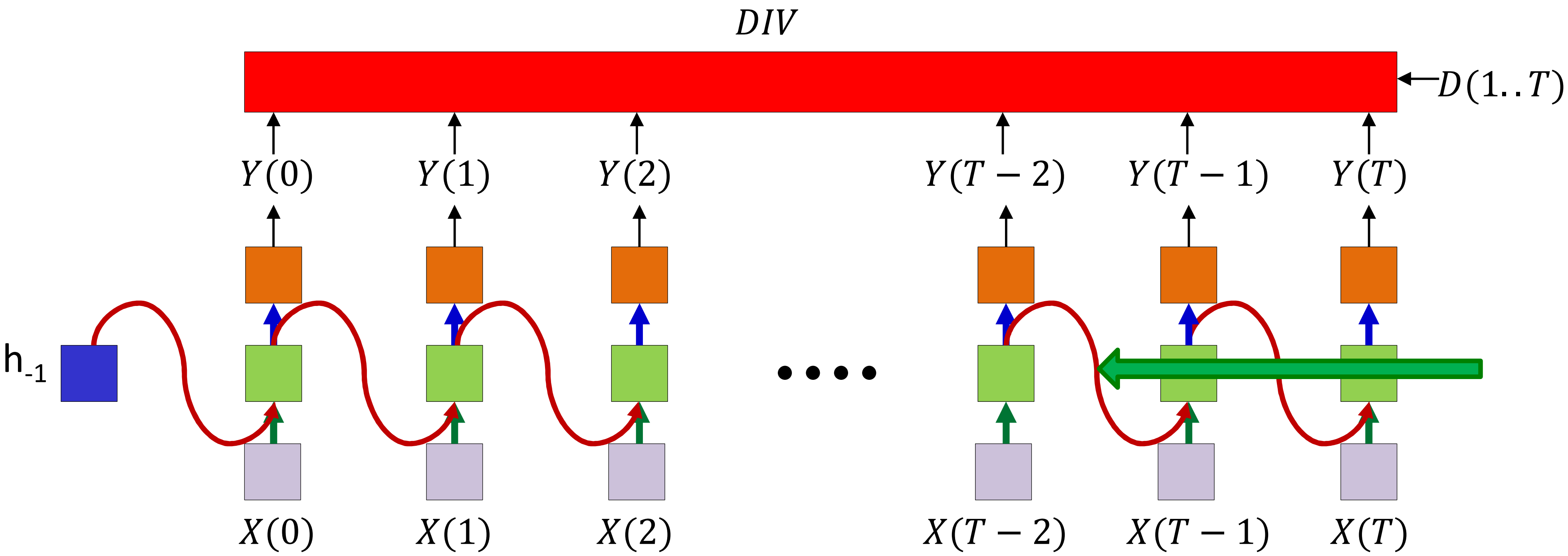
$$\frac{dDIV}{dZ_i^{(1)}(T-1)} = \frac{dDIV}{dh_i(T-1)} \frac{dh_i(T-1)}{dZ_i^{(1)}(T-1)}$$

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

Note the addition

$$\nabla_{W^{(1)}} DIV += X(T-1) \nabla_{Z^{(1)}(T-1)} DIV$$

Back Propagation Through Time



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

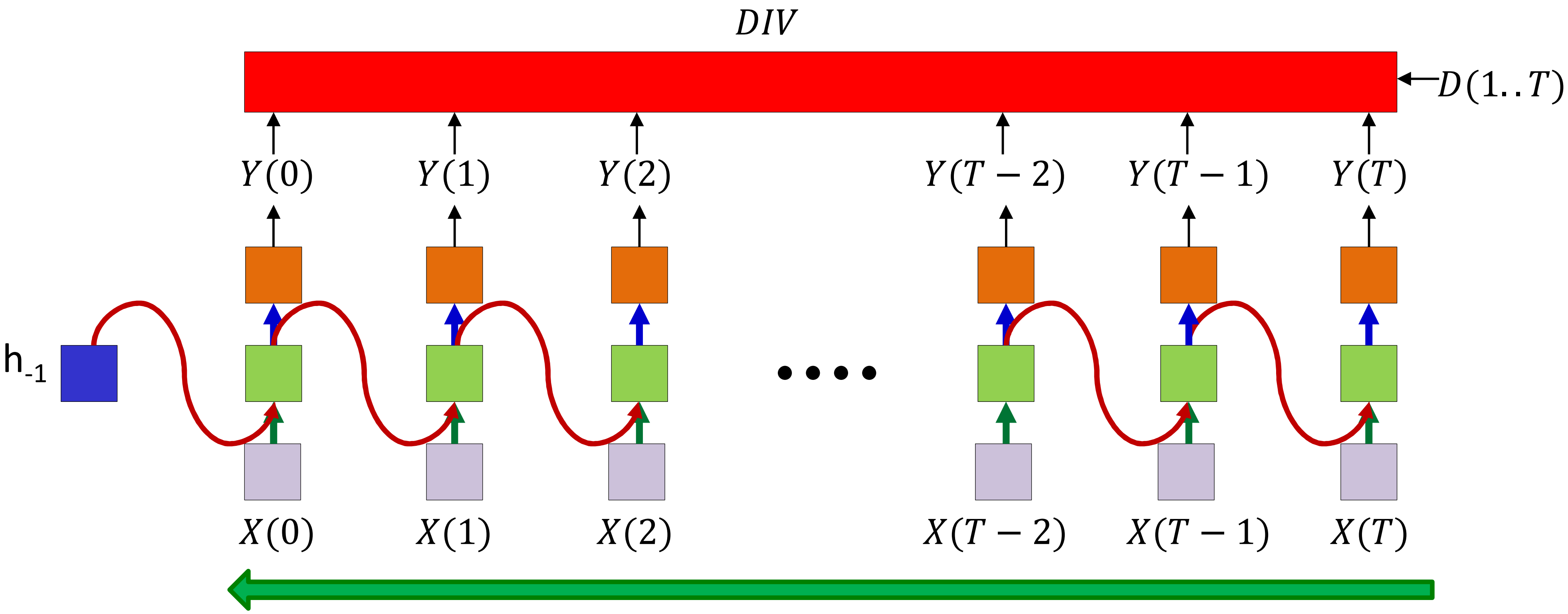
Note the addition

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

Note the addition

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

Back Propagation Through Time

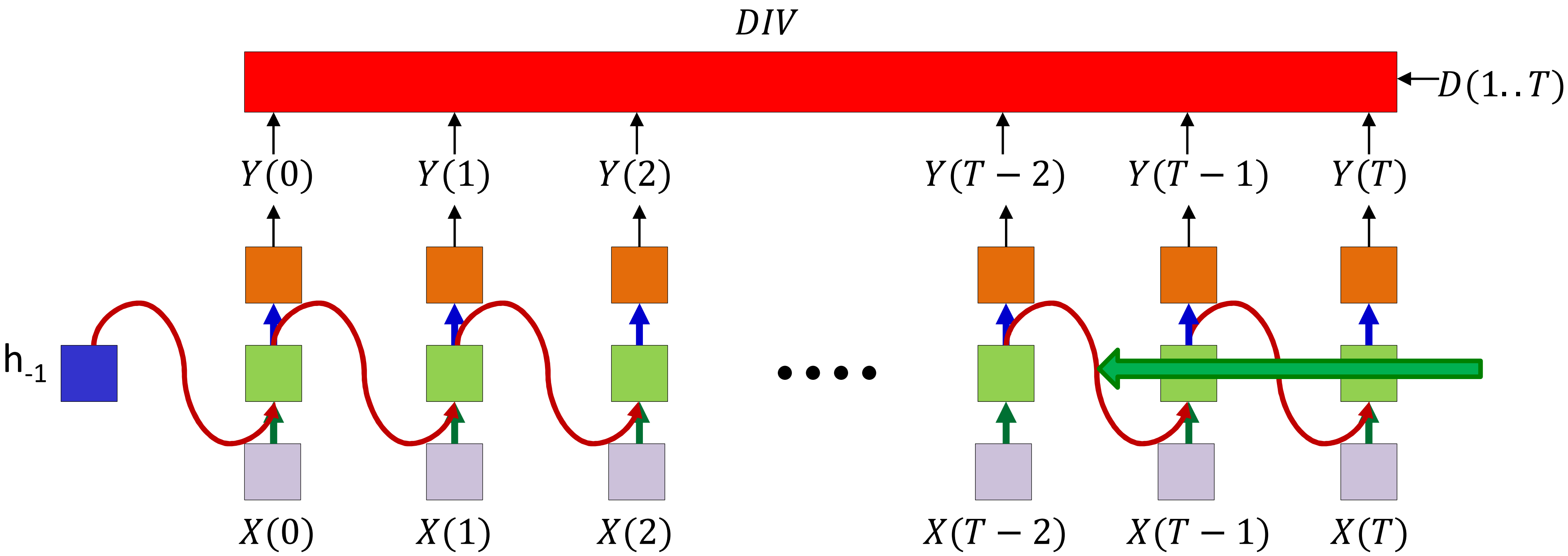


Continue computing derivatives going backward through time until..

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

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

Back Propagation Through Time



Initialize all derivatives to 0

For t = T downto 0

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

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

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

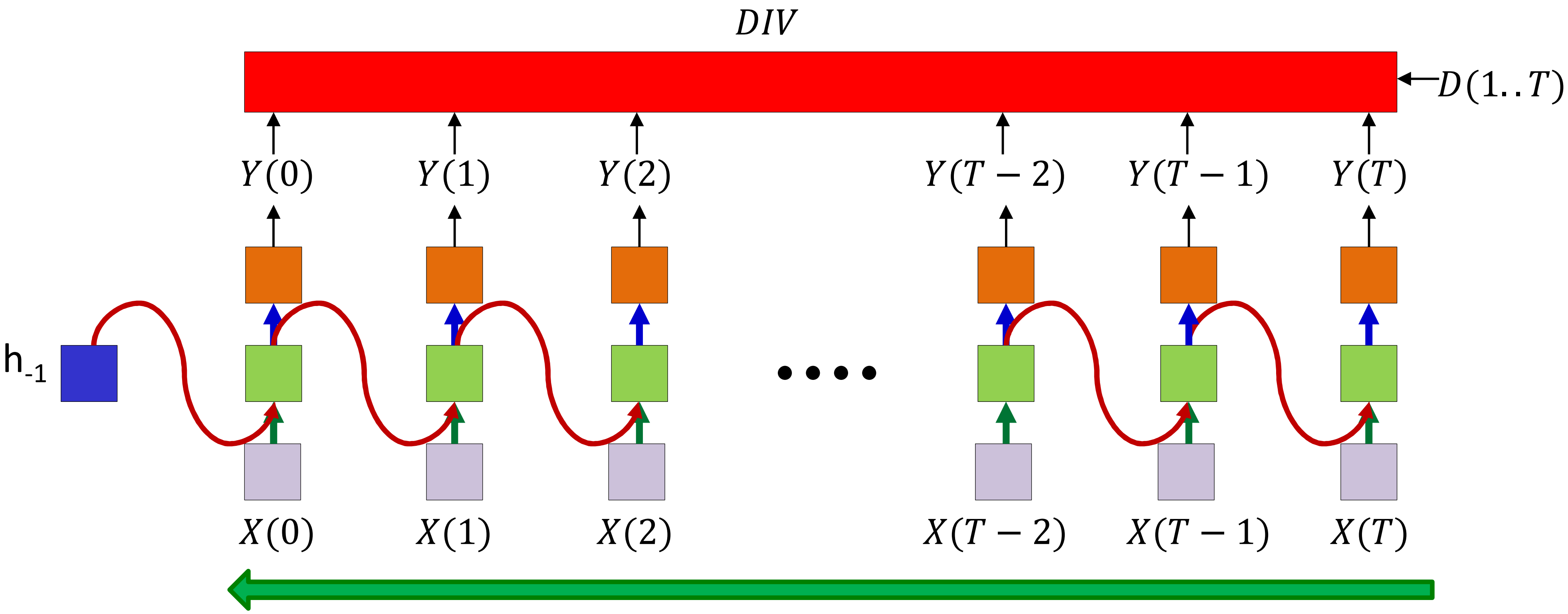
$$\nabla_{W^{(2)}}DIV \mathrel{+}= h(t)\nabla_{Z^{(2)}(t)}DIV$$

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

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

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

Back Propagation Through Time

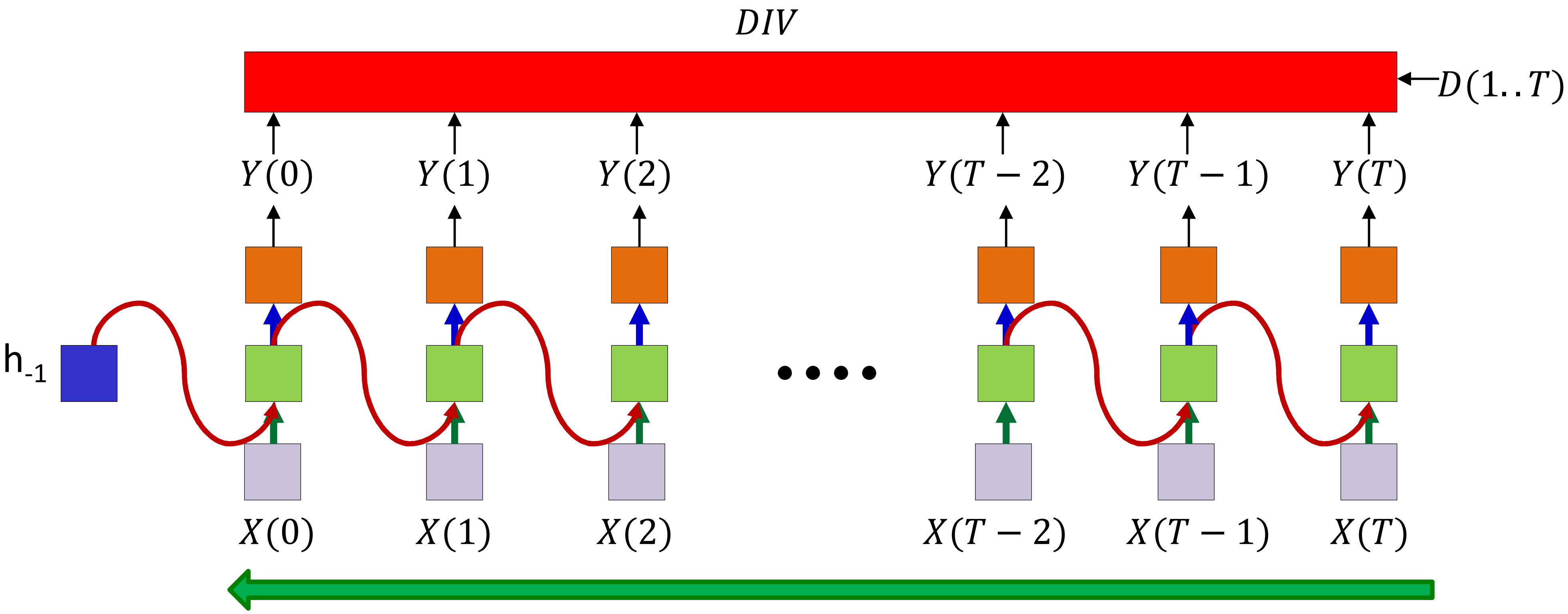


$$\boxed{\frac{dDIV}{dh_i^{(k)}(t)} = \sum_j w_{i,j}^{(k+1)} \frac{dDIV}{dZ_j^{(k+1)}(t)} + \sum_j w_{i,j}^{(k,k)} \frac{dDIV}{dZ_j^{(k)}(t+1)}}$$

Not showing derivatives at output neurons

$$\boxed{\frac{dDIV}{dZ_i^{(k)}(t)} = \frac{dDIV}{dh_i^{(k)}(t)} f_k'\left(Z_i^{(k)}(t)\right)}$$

Back Propagation Through Time



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

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

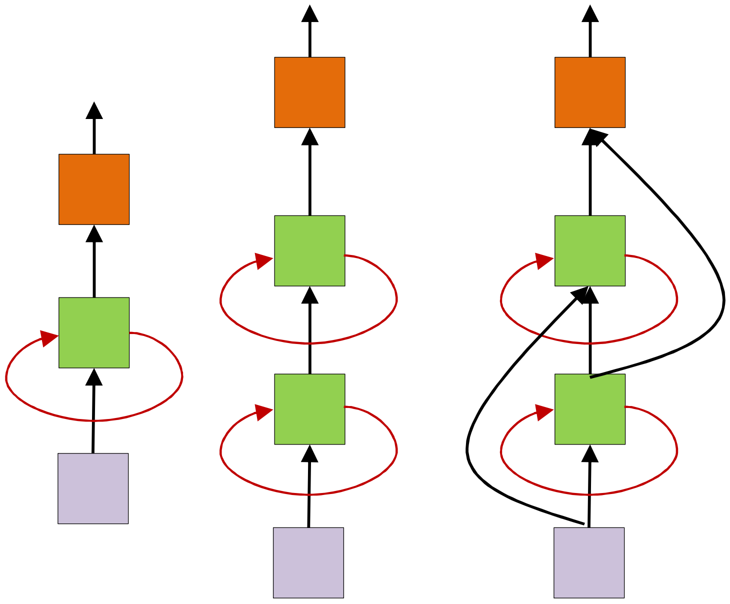
$$\frac{dDIV}{dw_{ij}^{(11)}} = \sum_t \frac{dDIV}{dZ_j^{(1)}(t)} h_i(t-1)$$

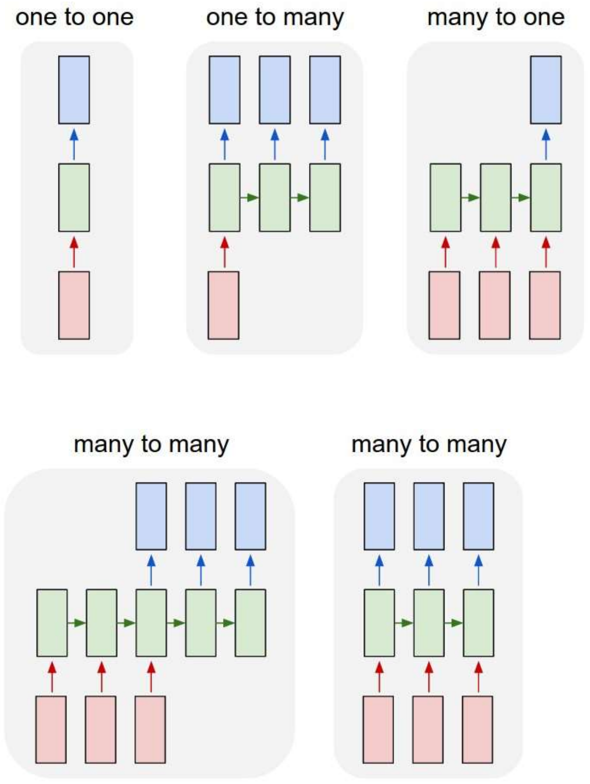
BPTT

```
# Assuming forward pass has been completed
# Jacobian(x,y) is the jacobian of x w.r.t. y
# Assuming dY(t) = gradient(div,Y(t)) available for all t
# Assuming all dz, dh, dW and db are initialized to 0

for t = T-1:downto:0  # Backward through time
    $dz_o$(t) = dY(t)Jacobian(Y(t),$z_o$(t))
    $dW_o$ += h(t,L)$dz_o$(t)
    $db_o$ += $dz_o$(t)
    dh(t,L) += $dz_o$(t)$W_o$

    for l = L:1  # Reverse through layers
        dz(t,l) = dh(t,l)Jacobian(h(t,l),z(t,l))
        dh(t,l-1) += dz(t,l) $W_c$(l)
        dh(t-1,l) = dz(t,l) $W_r$(l)

        $dW_c$(l) += h(t,l-1)dz(t,l)
        $dW_r$(l) += h(t-1,l)dz(t,l)
        db(l) += dz(t,l)
```

Subscript “c” – current
Subscript “r” – recurrent

BPTT





- Can be generalized to any architecture

Poll 3 (@617 @618)

SGD trains neural networks one input at a time, rather than over batches of inputs. The corresponding equivalent for RNNs would be to update the network after each input vector: True or False

- True
- False

Select all that are true:

- In RNNs the divergence we minimize is the sum of the divergences for the individual inputs in the time series
- The divergence is the divergence between the actual sequence of outputs and the desired sequence of outputs and cannot always be decomposed into the sum of divergences at individual time steps.

Poll 3

SGD trains neural networks one input at a time, rather than over batches of inputs. The corresponding equivalent for RNNs would be to update the network after each input vector: True or False

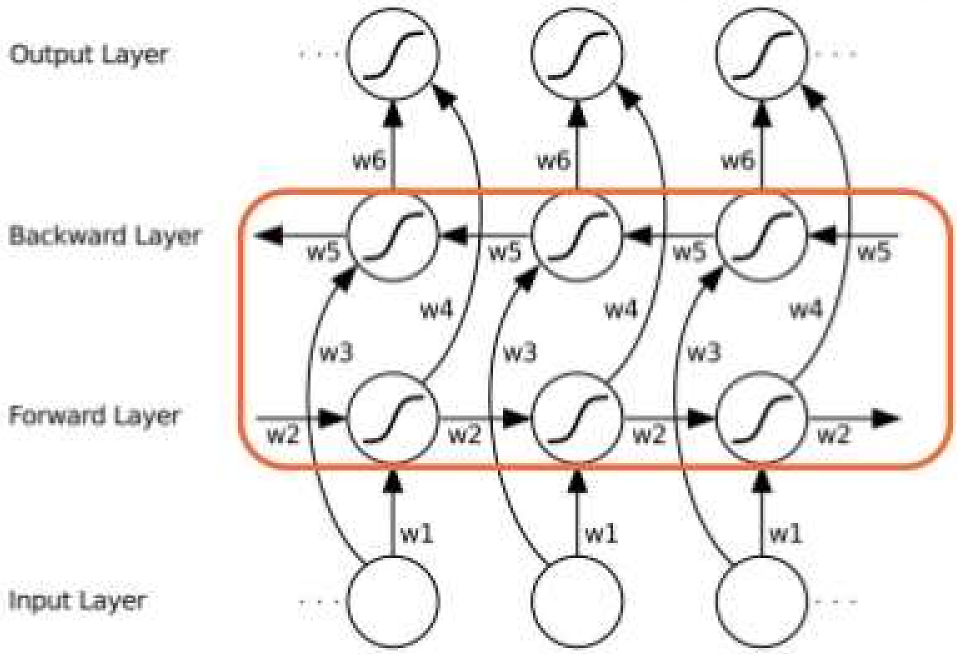
- True
- **False**

Select all that are true:

- In RNNs the divergence we minimize is the sum of the divergences for the individual inputs in the time series
- **The divergence is the divergence between the actual sequence of outputs and the desired sequence of outputs and cannot always be decomposed into the sum of divergences at individual time steps.**

Extensions to the RNN: *Bidirectional RNN*

Bidirectional RNN (BRNN)



Must learn weights w2, w3, w4 & w5; in addition to w1 & w6.

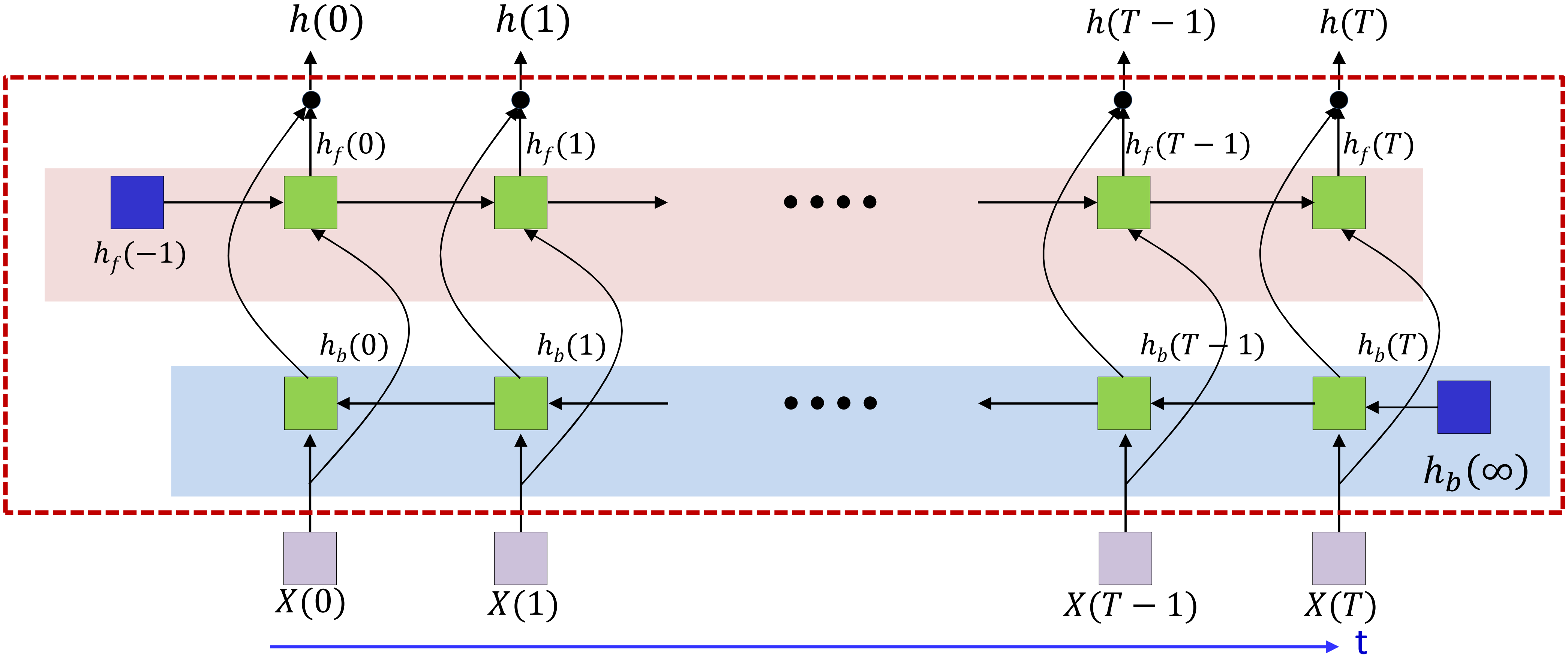
Proposed by Schuster and Paliwal 1997

Alex Graves, "Supervised Sequence Labelling with Recurrent Neural Networks"

- In problems where the entire input sequence is available before we compute the output, 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

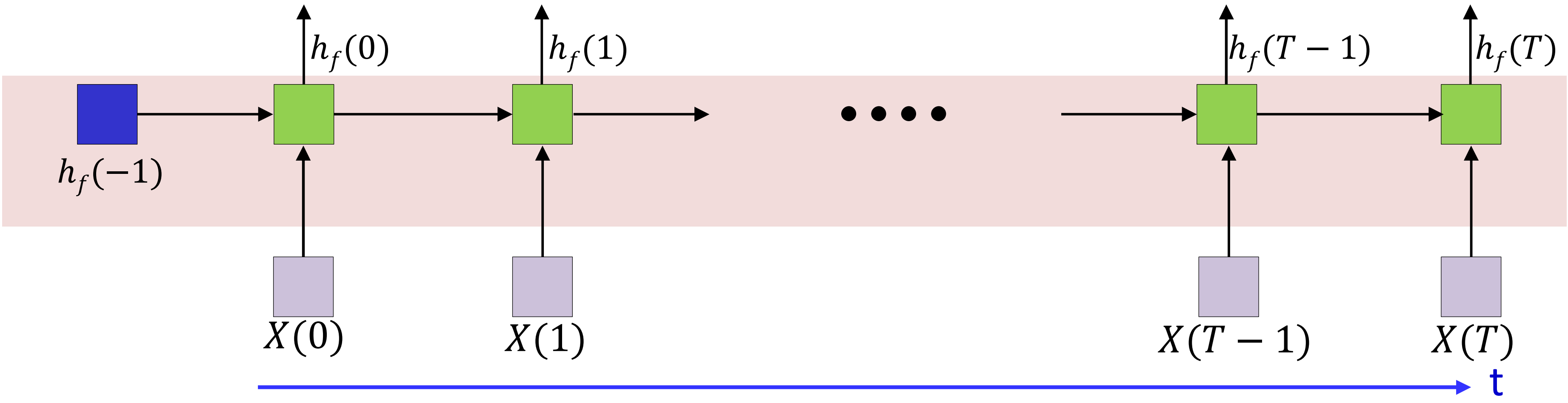
Bidirectional RNN



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

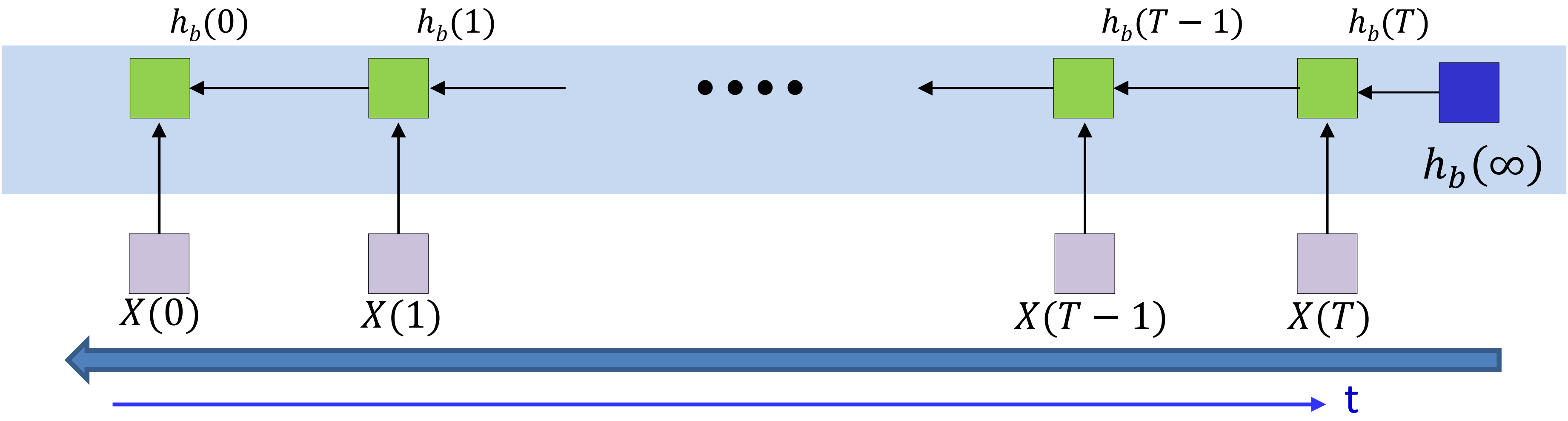
- The Block has two components
  - A forward net process the data from t=0 to t=T
  - A backward net processes it backward from t=T down to t=0

Bidirectional RNN block



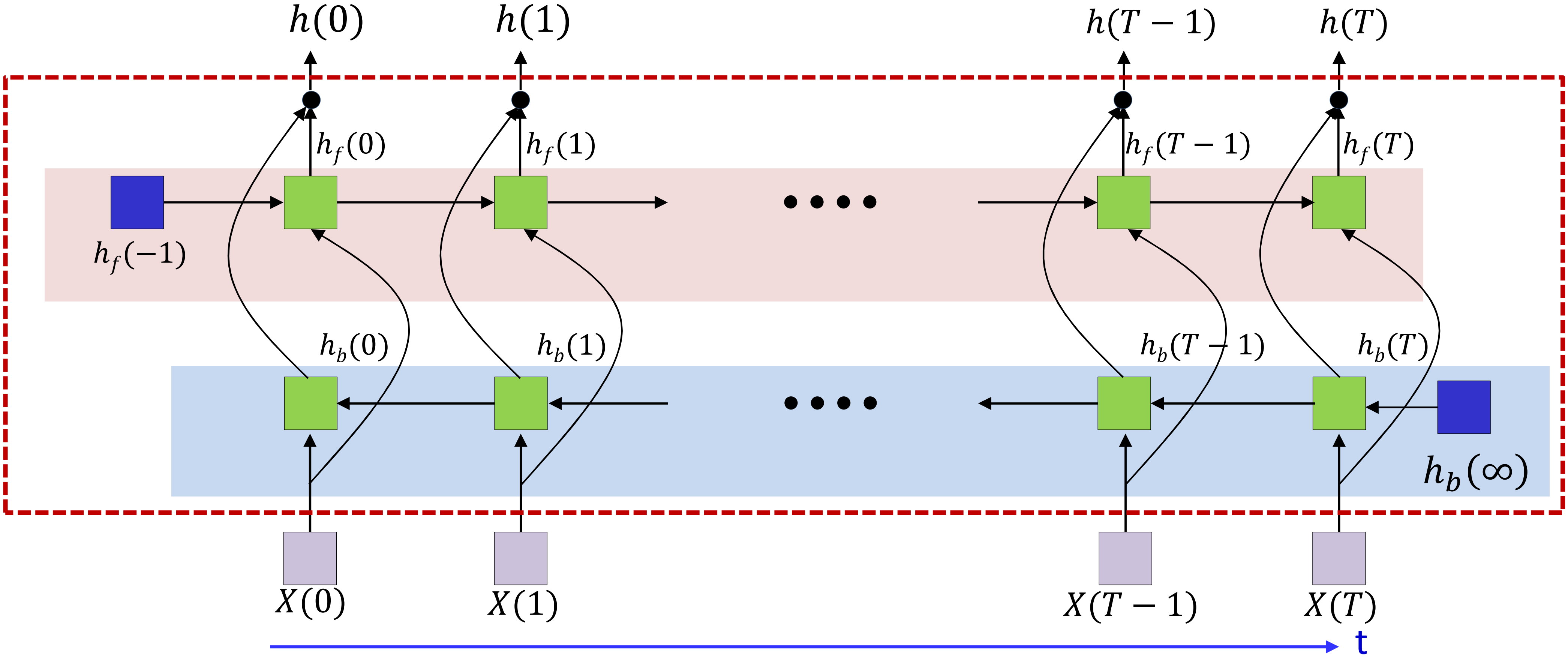
- The forward net process the data from t=0 to t=T
  - Only computing the hidden state values.

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.*

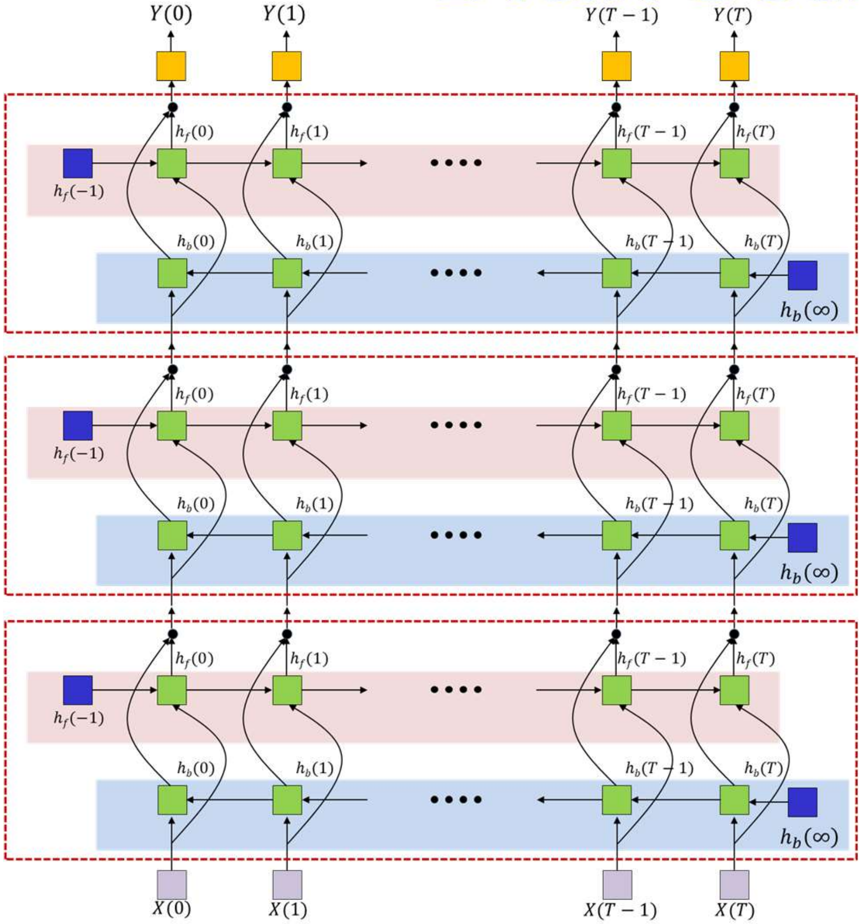
Bidirectional RNN block

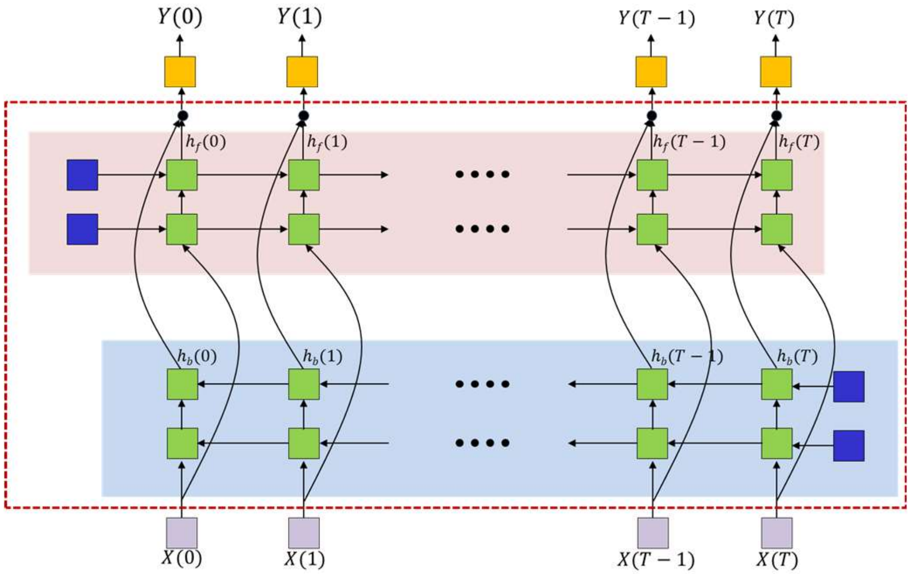


- The computed states of both networks are combined to give you the output of the bidirectional block
  - Typically just concatenate them

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

Bidirectional RNN





- Actual network may be formed by stacking one or more bidirectional blocks followed by an output layer
  - Forward and backward nets in each block are a single layer
  - Alternately, the forward and backward nets within each block 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

Poll 4 (@619)

A day trader in the stock exchange uses RNNs to make predictions from historical market data to decide which stocks to buy. He can use a bidirectional RNN for his task.

- True
- False

Poll 4

A day trader in the stock exchange uses RNNs to make predictions from historical market data to decide which stocks to buy. He can use a bidirectional RNN for his task.

- True
- **False**

Bidirectional RNN block: inference

```
# Subscript f represents forward net, b is backward net
# Assuming $h_f(-1,*)$ and $h_b(inf,*)$ are known
# x(t) is the input to the block (which could be from a lower layer)

#forward recurrence
for t = 0:T-1 # Going forward in time
    $h_f(t,0) = x(t)$ # Vectors. Initialize $h_f(0)$ to input
    for l = 1:$L_f$  # $L_f$ is depth of forward network hidden layers
        $z_f(t,l) = W_{fc}(l)h_f(t,l-1) + W_{fr}(l)h_f(t-1,l) + b_f(l)$
        $h_f(t,l) = tanh(z_f(t,l))$ # Assuming tanh activ.

#backward recurrence
$h_b(T,:,:) = h_b(inf,:,:)$ # Just the initial value
for t = T-1:downto:0 # Going backward in time
    $h_b(t,0) = x(t)$ # Vectors. Initialize $h_b(0)$ to input
    for l = 1:$L_b$ # $L_b$ is depth of backward network hidden layers
        $z_b(t,l) = W_{bc}(l)h_b(t,l-1) + W_{br}(l)h_b(t+1,l) + b_b(l)$
        $h_b(t,l) = tanh(z_b(t,l))$ # Assuming tanh activ.

for t = 0:T-1  # The output combines forward and backward
     $h(t) = [h_f(t,L_f); h_b(t,L_b)]$
```

Bidirectional RNN: Simplified code

- Code can be made modular and simplified for better interpretability…

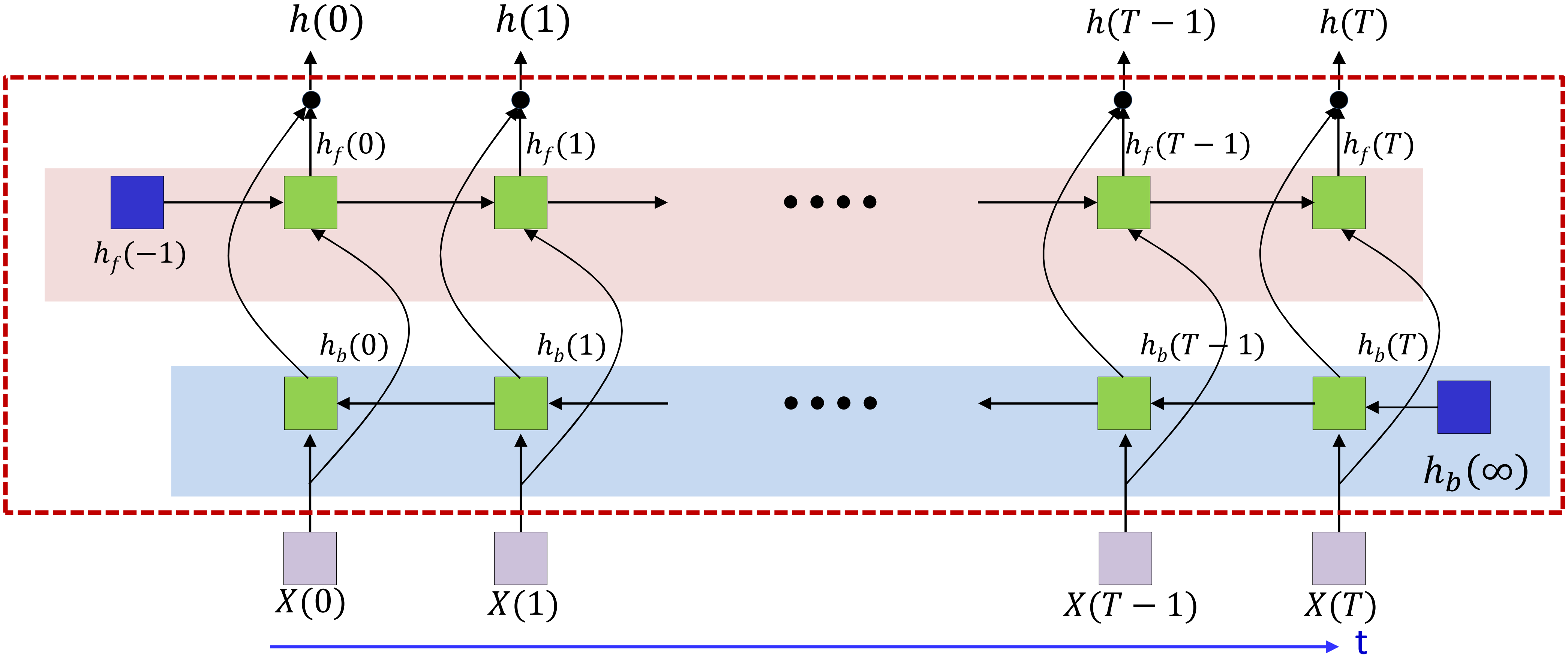
First: Define forward recurrence

```
# Inputs:
#    L : Number of hidden layers
#    $W_c$,$W_r$,b: current weights,  recurrent weights, biases
#    hinit:  initial value of h(representing h(-1,*))
#    x: input vector sequence
#    T: Length of input vector sequence
# Output:
#    h, z: sequence of pre-and post activation hidden
#          representations from all layers of the RNN

function RNN_forward(L, $W_c$, $W_r$, b, hinit, x, T)
    h(-1,:) = hinit  # hinit is the initial value for all layers
    for t = 0:T-1 # Going forward in time
        h(t,0) = x(t) # Vectors. Initialize h(0) to input
        for l = 1:L
            z(t,l) = $W_c$(l)h(t,l-1) + $W_r$(l)h(t-1,l) + b(l)
            h(t,l) = tanh(z(t,l)) # Assuming tanh activ.
    return h
```

Bidirectional RNN block

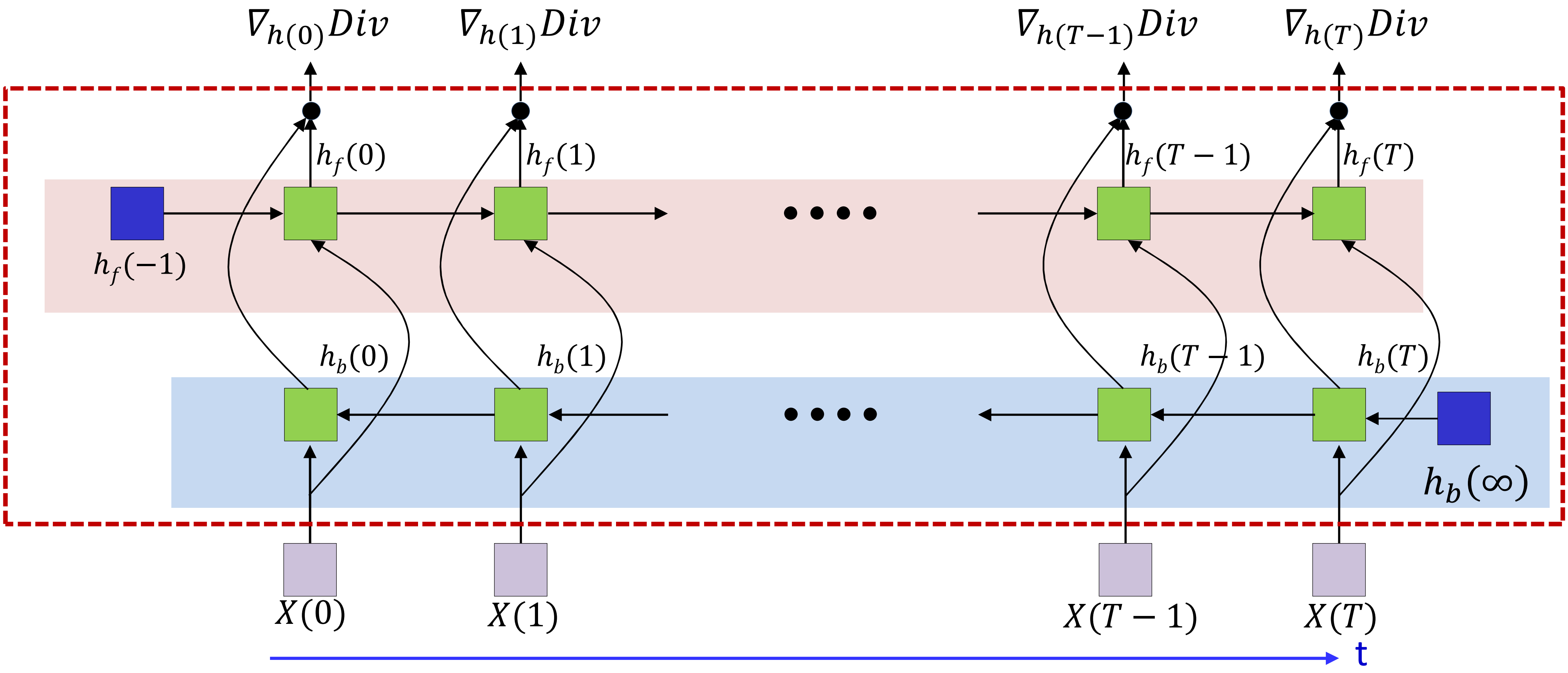
```
# Subscript f represents forward net, b is backward net
# Assuming $h_f$(-1,*) and $h_b$(inf,*) are known

#forward pass
$h_f$ = RNN_forward($L_f$, $W_{fc}$, $W_{fr}$, $b_f$, $h_f$(-1,:), x, T)

#backward pass
$x_{rev}$ = fliplr(x)   # Flip it in time
$h_{brev}$ = RNN_forward($L_b$, $W_{bc}$, $W_{br}$, $b_b$, $h_b$(inf,:), $x_{rev}$, T)
$h_b$ = fliplr($h_{brev}$)   # Flip back to straighten time

#combine the two for the output
for t = 0:T-1  # The output combines forward and backward
    h(t) = [$h_f$(t,$L_f$); $h_b$(t,$L_b$)]
```

Backpropagation in BRNNs



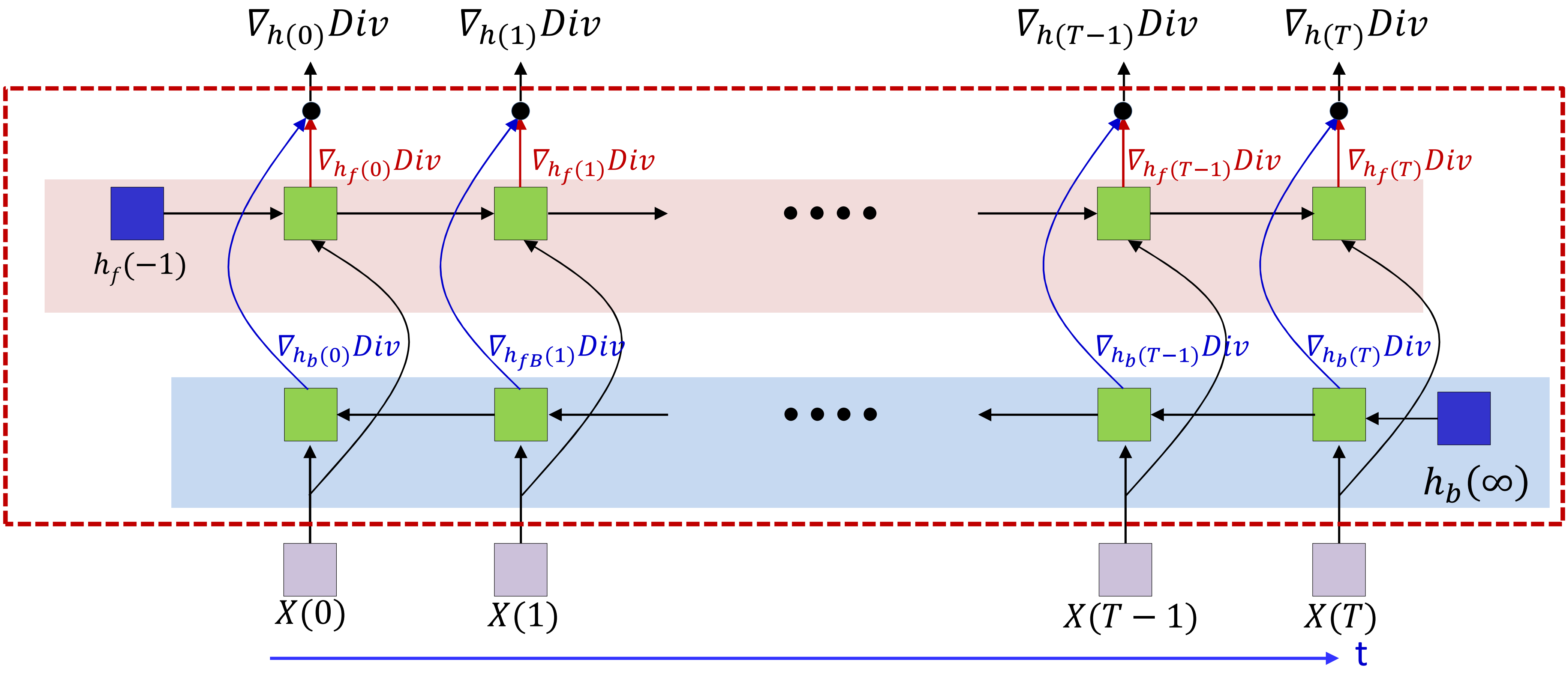
- Forward pass: Compute both forward and backward networks and final output

Backpropagation in BRNNs



- Backward pass: Assume gradients of the divergence are available for the block outputs $h(t)$
  - Obtained via backpropagation from network output
  - Will have the same dimension (length) as $h(t)$
    - Which is the sum of the dimensions of $h_f(t)$ and $h_b(t)$

Backpropagation in BRNNs



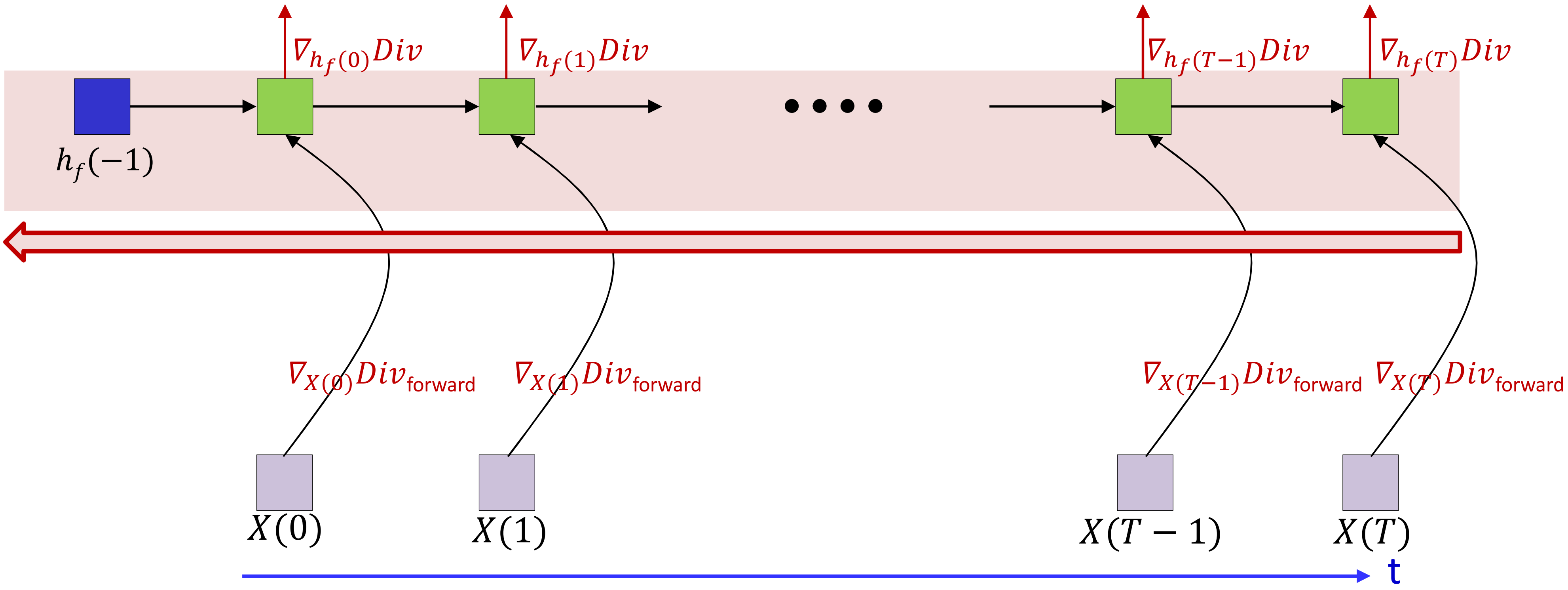
- Separate gradient into forward and backward components

$$\nabla_{h(t)}Div = [\nabla_{h_f(t)}Div; \nabla_{h_b(t)}Div]$$

  - Extract $\nabla_{h_f(t)}Div$ and $\nabla_{h_b(t)}Div$ from $\nabla_{h(t)}Div$.

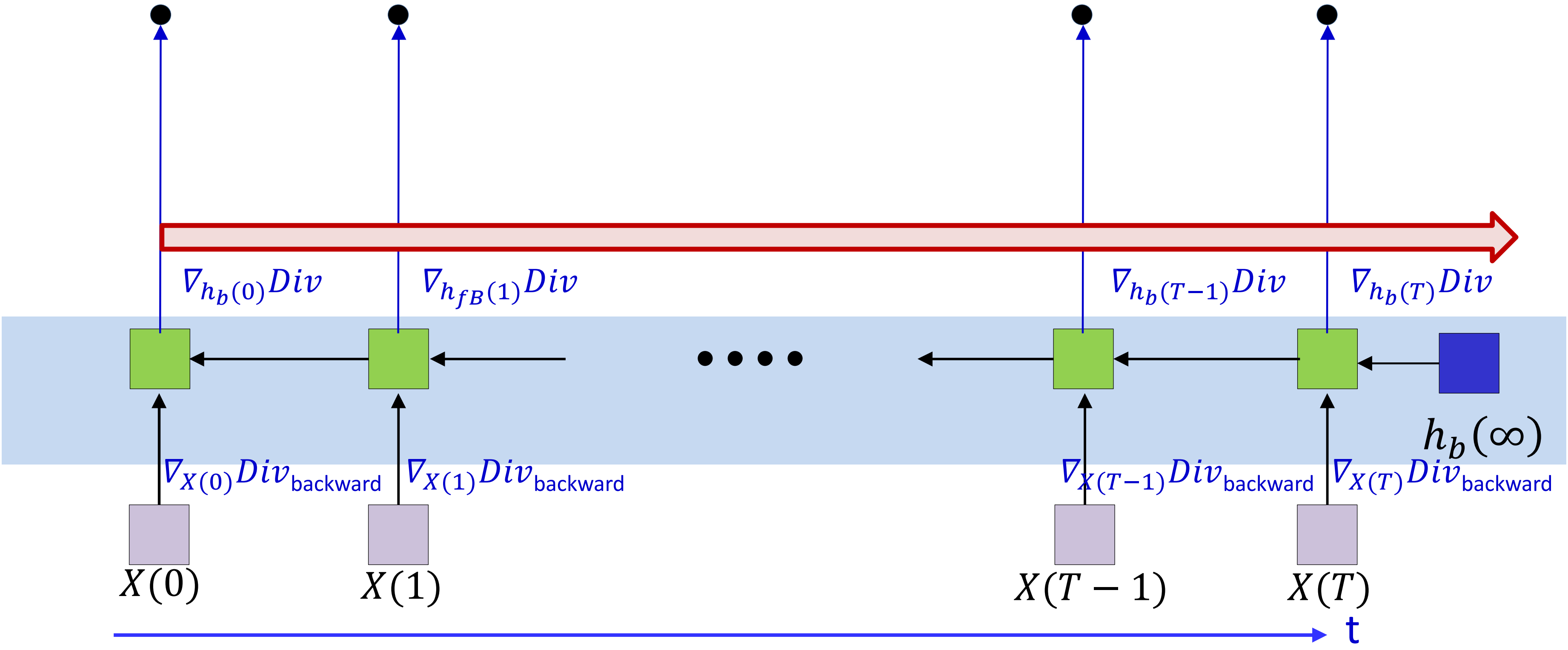
- Separately perform backprop on the forward and backward nets

Backpropagation in BRNNs


$\nabla_{h_f(0)}Div$
$\nabla_{h_f(1)}Div$
$\nabla_{h_f(T-1)}Div$
$\nabla_{h_f(T)}Div$
$h_f(-1)$
$\nabla_{X(0)}Div_{\text{forward}}$
$\nabla_{X(1)}Div_{\text{forward}}$
$\nabla_{X(T-1)}Div_{\text{forward}}$
$\nabla_{X(T)}Div_{\text{forward}}$
$X(0)$
$X(1)$
$X(T-1)$
$X(T)$
t

- Backprop for forward net:
  - Backpropagate $\nabla_{h_f(t)}Div$ from $t = T$ down to $t = 0$ in the usual way
  - Will obtain derivatives for all the parameters of the forward net
  - Will also get $\nabla_{X(t)}Div_{forward}$ at the input to the forward net
    - Partial derivative of the gradient for $X(t)$ computed through the forward net

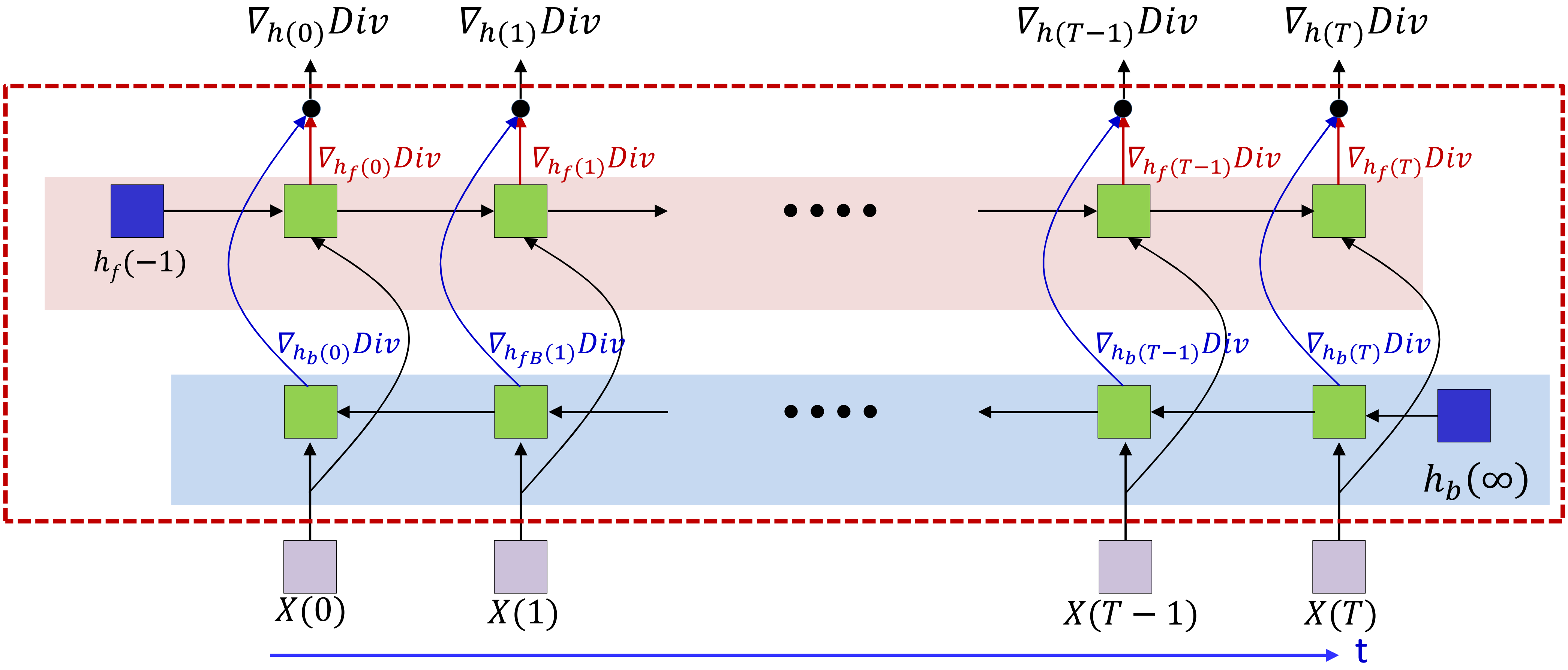
Backpropagation in BRNNs



$\nabla_{h_b(0)}Div$
$\nabla_{h_{fB}(1)}Div$
$\nabla_{h_b(T-1)}Div$
$\nabla_{h_b(T)}Div$
$h_b(\infty)$
$\nabla_{X(0)}Div_{\text{backward}}$
$\nabla_{X(1)}Div_{\text{backward}}$
$\nabla_{X(T-1)}Div_{\text{backward}}$
$\nabla_{X(T)}Div_{\text{backward}}$
$X(0)$
$X(1)$
$X(T-1)$
$X(T)$
t

- Backprop for backward net:
  - Backpropagate $\nabla_{h_b(t)}Div$ *forward* from $t = 0$ *up* to $t = T$
  - Will obtain derivatives for all the parameters of the forward net
  - Will also get $\nabla_{X(t)}Div_{backward}$
    - Partial derivative of the gradient for $X(t)$ computed through the backward net

Backpropagation in BRNNs



- Finally add up the forward and backward partial derivatives to get the full gradient for $X(t)$

$$\nabla_{X(t)}Div = \nabla_{X(t)}Div_{backward} + \nabla_{X(t)}Div_{forward}$$

Backpropagation: Pseudocode

- As before we will use a 2-step code:
  - A basic backprop routine that we will call
  - Two calls to the routine within a higher-level wrapper

First: backprop through a recurrent net

```
# Inputs:
#    (In addition to inputs used by L : Number of hidden layers
#    $dh_{top}$:  derivatives ddiv/$dh_*$(t,L) at each time (* may be f or b)
#    h, z:  h and z values returned by the forward pass
#    T: Length of input vector sequence
# Output:
#    $dW_c$, $dW_b$, db $dh_{init}$: derivatives w.r.t current and recurrent weights,
#                             biases, and initial h.
# Assuming all dz, dh, $dW_c$, $dW_r$ and db are initialized to 0

function RNN_bptt(L, $W_c$, $W_r$, b, hinit, x, T, $dh_{top}$, h, z)
    dh = zeros
    for t = T-1:downto:0  # Backward through time
        dh(t,L) += $dh_{top}$(t)
        h(t,0) = x(t)
        for l = L:1  # Reverse through layers
            dz(t,l) = dh(t,l)Jacobian(h(t,l),z(t,l))
            dh(t,l-1) += dz(t,l) $W_c$(l)
            dh(t-1,l) += dz(t,l) $W_r$(l)

            $dW_c$(l) += h(t,l-1)dz(t,l)
            $dW_r$(l) += h(t-1,l)dz(t,l)
            db(l) += dz(t,l)
         dx(t)= dh(t,0)

    return dx, dWc, dWr, db, dh(-1)  # dh(-1) is actually dh(-1,1:L,:)
```

BRNN block: gradient computation

```
# Subscript f represents forward net, b is backward net
# Given dh(t), t=0…T-1 : The sequence of gradients from the upper layer
# Also assumed available:
#      x(t), t=0…T-1 : the input to the BRNN block
#      $z_f(t)$, $h_f(t)$ : Complete forward-computation outputs for all layers of the forward net
#      $z_b(t)$, $h_b(t)$ : Complete backward-computation outputs for all layers of the backward net
# $L_f$ and $L_b$ are the number of components in $h_f(t)$ and $h_b(t)$

for t = 0:T-1  # Separate out forward and backward net gradients
    $dh_f(t) = dh(t,1:L_f)$
    $dh_b(t) = dh(t,L_f+1:L_f+L_b)$

#forward net
$[d\mathbf{x}_f\ d\mathbf{W}_{fc}, d\mathbf{W}_{fr}, d\mathbf{b}_f, dh_f(-1)]$ = RNN_bptt($L$, $\mathbf{W}_{fc}$, $\mathbf{W}_{fr}$, $\mathbf{b}_f$, $h_f(-1)$, x, T, $dh_f$, $h_f$, $z_f$)

#backward net
$\mathbf{x}_{rev}$ = fliplr(x)  # Flip it in time
$dh_{brev}$ = fliplr($dh_b$)
$h_{brev}$ = fliplr($h_b$)
$z_{brev}$ = fliplr($z_b$)
$[d\mathbf{x}_{brev}, d\mathbf{W}_{bc}, d\mathbf{W}_{br}, d\mathbf{b}_b, dh_b(inf)]$ = RNN_bptt($L$, $\mathbf{W}_{bc}$, $\mathbf{W}_{br}$, $\mathbf{b}_b$, $h_b(inf)$, $\mathbf{x}_{rev}$, T, $dh_{brev}$, $h_{brev}$, $z_{brev}$)
$d\mathbf{x}_b$ = fliplr($d\mathbf{x}_{brev}$)

for t = 0:T-1  # Add the partials
    $d\mathbf{x}(t) = d\mathbf{x}_f(t) + d\mathbf{x}_b(t)$
```

Story so far

- Time series analysis must consider past inputs along with current input

- Recurrent networks look into the infinite past through a state-space framework
  - Hidden states that recurse on themselves

- Training recurrent networks requires
  - Defining a divergence between the actual and desired output *sequences*
  - Backpropagating gradients over the entire chain of recursion
    - Backpropagation through time
  - Pooling gradients with respect to individual parameters over time

- Bidirectional networks analyze data both ways, begin→end and end→beginning to make predictions
  - In these networks, backprop must follow the chain of recursion (and gradient pooling) separately in the forward and reverse nets

RNNs..

- Excellent models for series data analysis tasks
  - Time-series prediction
  - Time-series classification
  - Sequence generation..