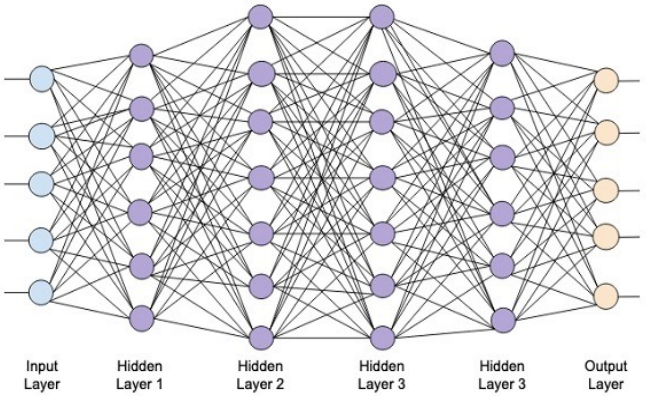
MATH 7339 - Machine Learning and Statistical Learning Theory 2

Section- Time Series using Deep Learning

1. Neural Network Basics
2. Backpropagation
3. Convolutional Neural Networks
4. Recurrent Neural Networks
5. Long Short-Term Memory (LSTM)
6. Deep Learning for Time Series

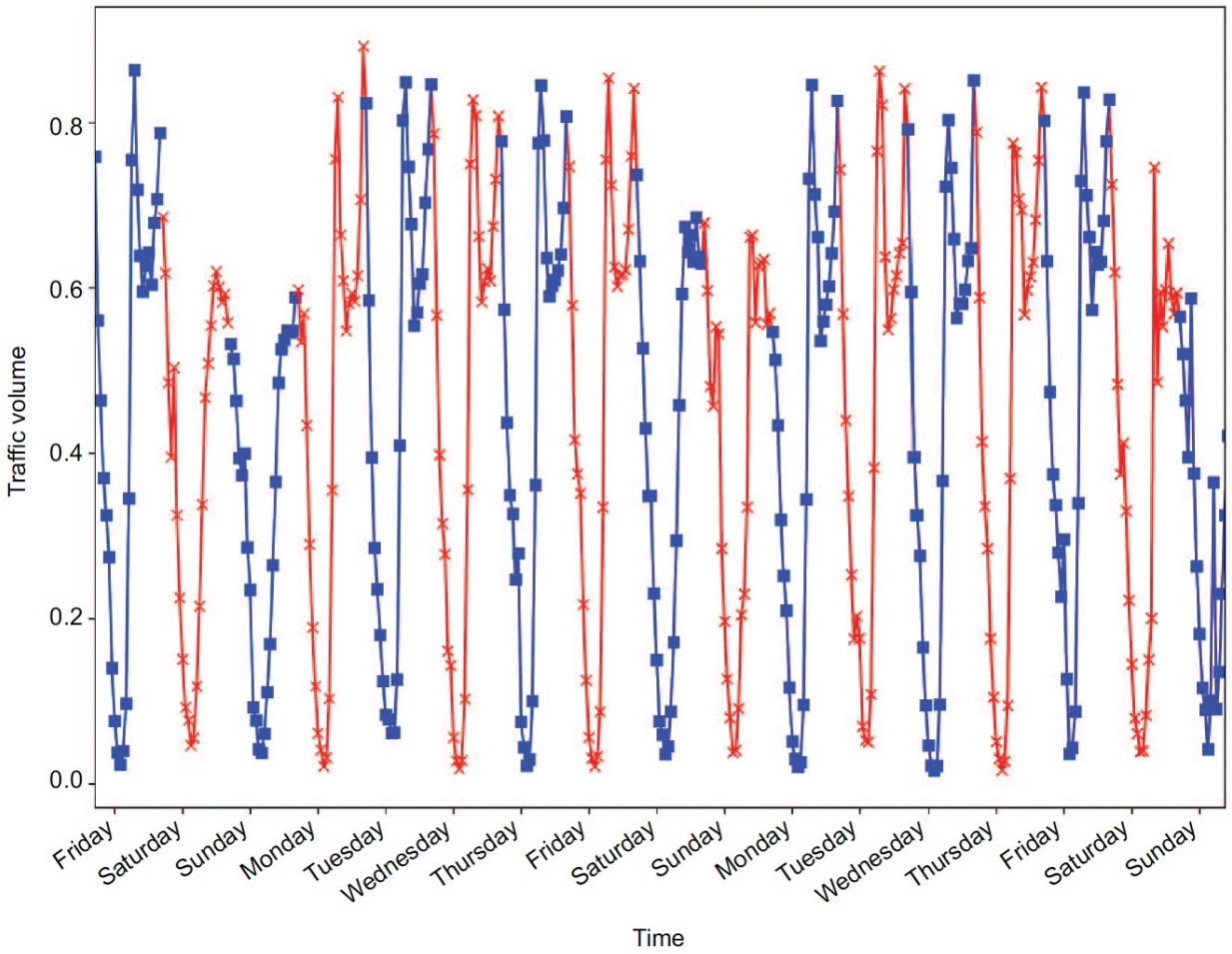
❑ Neural Network Model:



Machine Learning Framework/Structure:

1. Data: $\mathcal{D} = \left(\vec{x}^{(i)}, y^{(i)}\right)$ for $i = 1 \ldots n$.
2. **Model $\boldsymbol{h_\Theta(\vec{x})}$**
3. **Cost Function $\boldsymbol{J(\Theta)}$**
4. **Optimization**
5. Prediction $\boldsymbol{h_{\hat{\Theta}}(\vec{x})}$

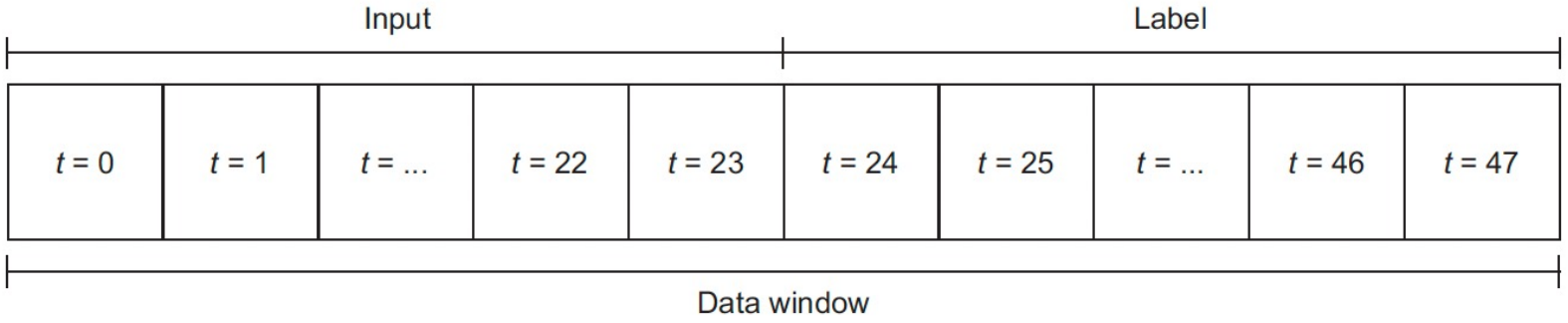
Neural network for time series:



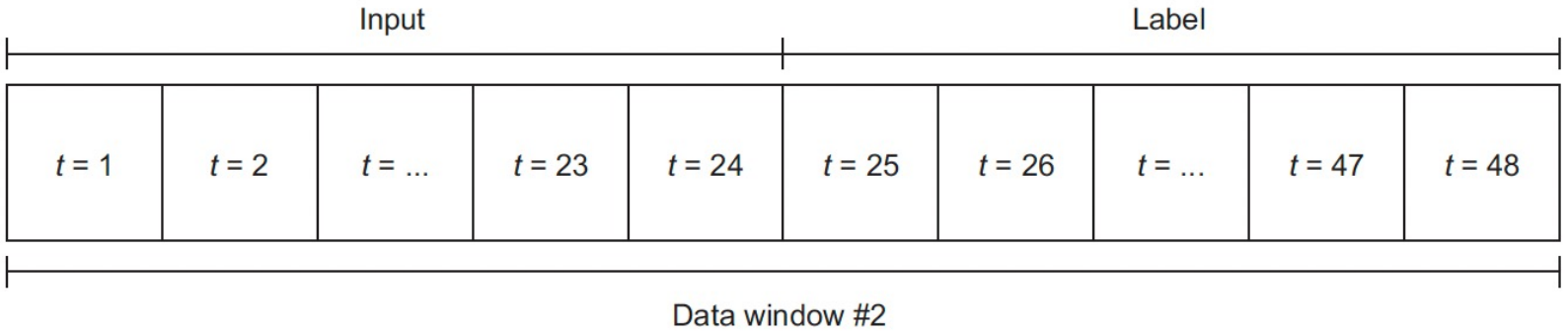
The inputs are shown with square markers, and the labels are shown with crosses. Each data window consists of 24 timesteps with square markers followed by 24 labels with crosses.

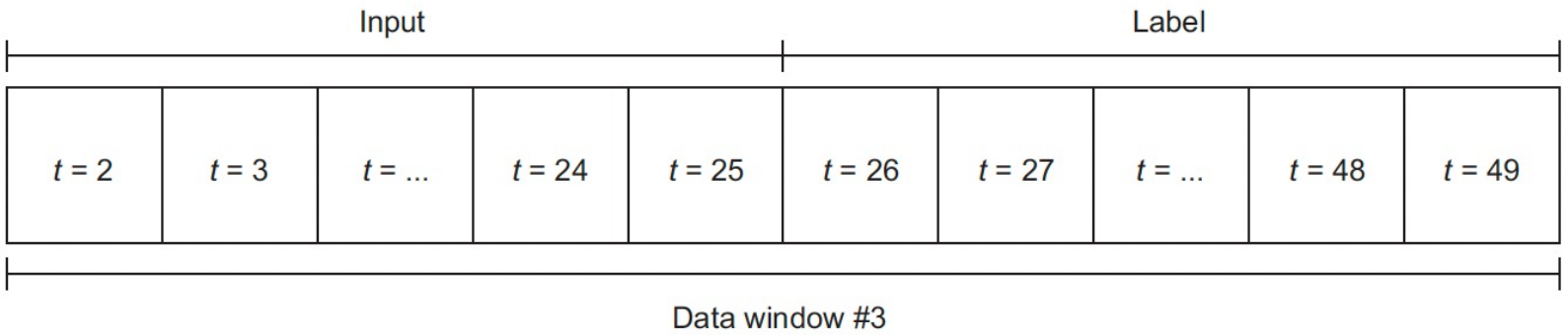
Data window.

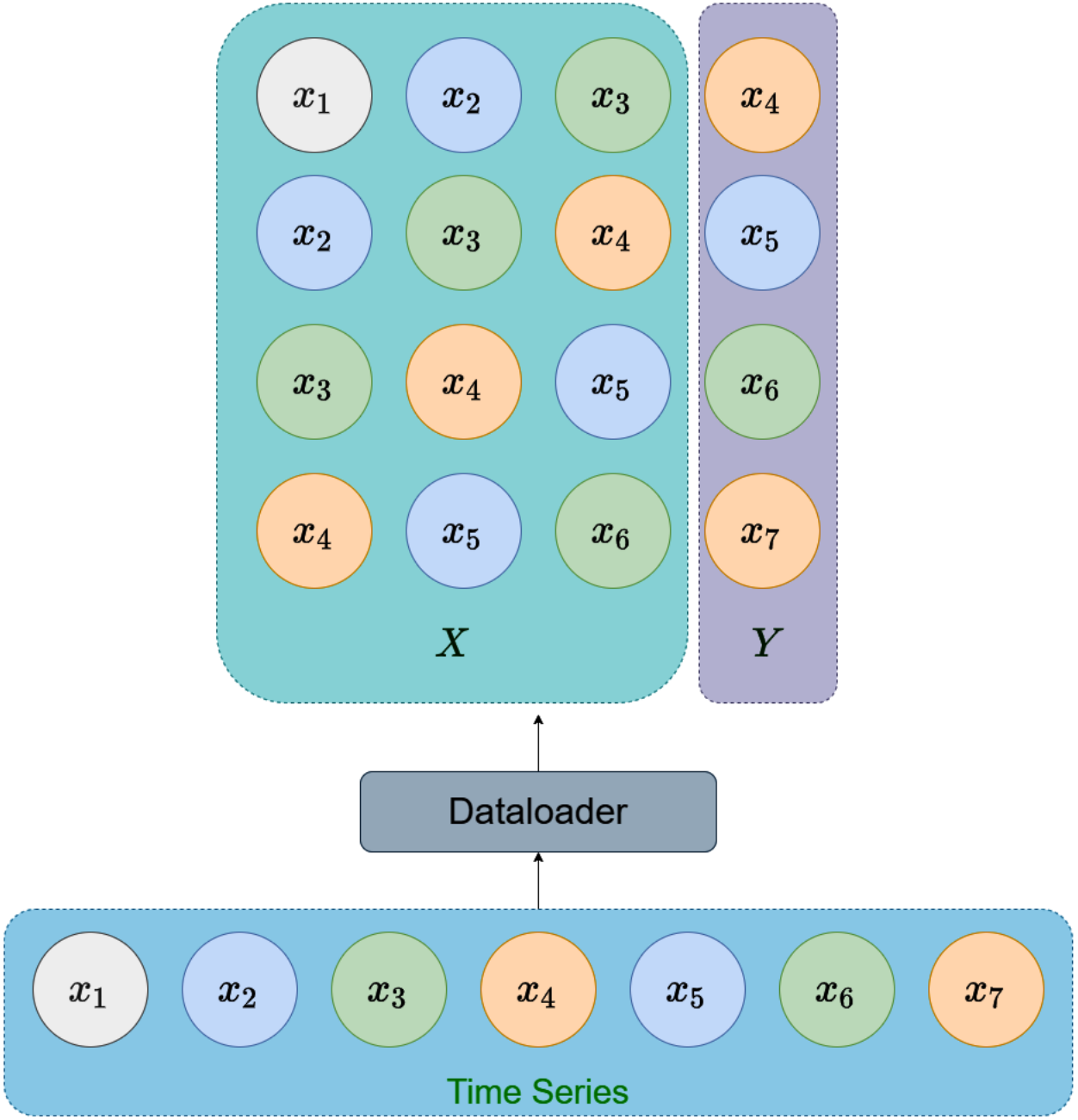
Our data window has 24 timesteps as input and 24 timesteps as output.



You might think that we are wasting a lot of training data, but






x_1 x_2 x_3 x_4
x_2 x_3 x_4 x_5
x_3 x_4 x_5 x_6
x_4 x_5 x_6 x_7
X
Y
Dataloader
x_1 x_2 x_3 x_4 x_5 x_6 x_7
Time Series

2 Neural Network Model:

$$h_\Theta(\vec{x}) \coloneqq F^{[m]} \circ \Theta^{[m]} \circ \cdots \circ F^{[2]} \circ \Theta^{[2]} \circ F^{[1]} \circ \Theta^{[1]}$$

3 Cost Functions :

$J(\Theta) \coloneqq L(h_\Theta(X), \vec{y})$, where $L(-,-)$ is a **metric**.

For example:

1. Mean Square Error for regression

$$J(\Theta) = \frac{1}{n}\|h_\Theta(X) - \vec{y}\|^2 = \frac{1}{n}\sum_{i=1}^{n}\left(h_\Theta\left(\vec{x}^{(i)}\right) - y^{(i)}\right)^2$$

2. Cross-Entropy cost for classification

$$J(\Theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\mathbb{I}\left(y^{(i)} = k\right)\ln\left(h_\Theta\left(\vec{x}^{(i)}\right)_k\right)$$

3. Hinge loss, 0–1 loss, …

4 Optimization-Gradient Descent:

Goal: Minimize the loss function $J(\Theta)$ by gradient descent:

$$\Theta^{j+1} = \Theta^{j} - \alpha \nabla J(\Theta^{j})$$

The key calculation is the **gradient** $\nabla J(\Theta)$ by **chain rule**.

For example: $F = F\left(\vec{z}\left(\vec{x}\left(\vec{t}\right)\right)\right)$: $\mathbb{R}^{a} \to \mathbb{R}^{s} \to \mathbb{R}^{n} \to \mathbb{R}^{m}$

By Chain Rule: $$\frac{\partial F}{\partial \vec{t}} = \frac{\partial F}{\partial \vec{z}} \frac{\partial \vec{z}}{\partial \vec{x}} \frac{\partial \vec{x}}{\partial \vec{t}}$$

Difficulty: Too much calculation/memory in formula $\nabla J(\Theta)$

Solution: **automatic differentiation (**Back-propagation)

Computation Methods:

1. Numerical differentiation using finite difference approximations (e.g., set $h = 0.001$);

2. Manually working out **analytical** derivatives and coding them directly;

3. Derive analytic gradient, check your implementation with numerical gradient to avoid redundant computations; (called **automatic differentiation (**Autodiff**)**, also called **algorithmic differentiation**), e.g., Back-propagation method.

Comments:

- **Numerical Finite differences** are expensive, since you need to do a forward pass for each derivative. It also induces huge numerical error. Normally, we only use it for testing. *Autodiff is not finite differences.*
- **Symbolic** differentiation (e.g. Mathematica) can result in complex and redundant expressions. *Autodiff is not symbolic differentiation.*

- **Autodiff** is not a formula, but a **procedure** for computing derivatives.
- **Autodiff** is both efficient (linear in the cost of computing the value) and numerically stable.

Example: One layer neural network:

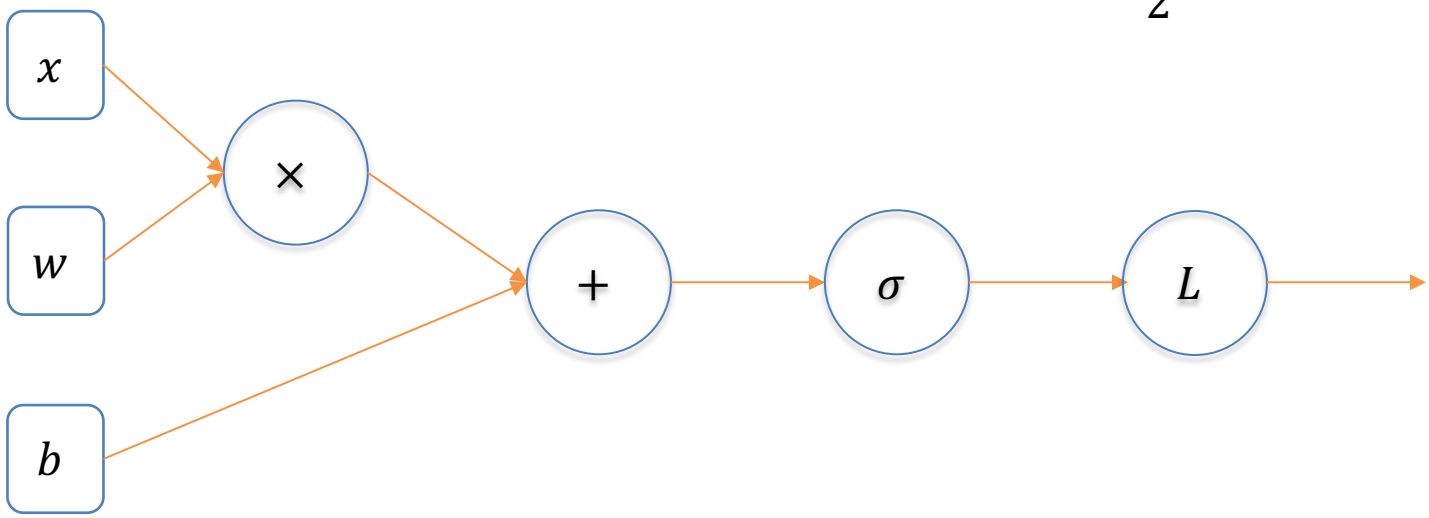
$$z = wx + b$$

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$L(y) = \frac{1}{2}(y - c)^2$$

Computational graphs

$$L(w, b) = \frac{1}{2}(\sigma(wx + b) - c)^2$$



Analytical Derivatives:

$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}\left(\frac{1}{2}(\sigma(wx + b) - t)^2\right) = \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx + b) - t)^2$$

$$= (\sigma(wx + b) - t)\frac{\partial}{\partial w}(\sigma(wx + b) - t)$$

$$= (\sigma(wx + b) - t)\,\sigma'(wx + b)\frac{\partial}{\partial w}(wx + b)$$

$$\frac{\partial L}{\partial w} = \frac{dL}{dy}\frac{dy}{dz}\frac{\partial z}{\partial w} = (\sigma(wx + b) - t)\,\sigma'(wx + b)x$$

$$\frac{\partial L}{\partial b} = \frac{\partial}{\partial b}\left(\frac{1}{2}(\sigma(wx + b) - t)^2\right) = \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx + b) - t)^2$$

$$= (\sigma(wx + b) - t)\frac{\partial}{\partial b}(\sigma(wx + b) - t)$$

$$= (\sigma(wx + b) - t)\,\sigma'(wx + b)\frac{\partial}{\partial b}(wx + b)$$

$$\frac{\partial L}{\partial b} = \frac{dL}{dy}\frac{dy}{dz}\frac{\partial z}{\partial b} = (\sigma(wx + b) - t)\,\sigma'(wx + b)$$

Disadvantages?

Efficient algorithmic differentiation :

- Computing the loss functions:

$$z = wx + b$$

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$L(y) = \frac{1}{2}(y - c)^2$$

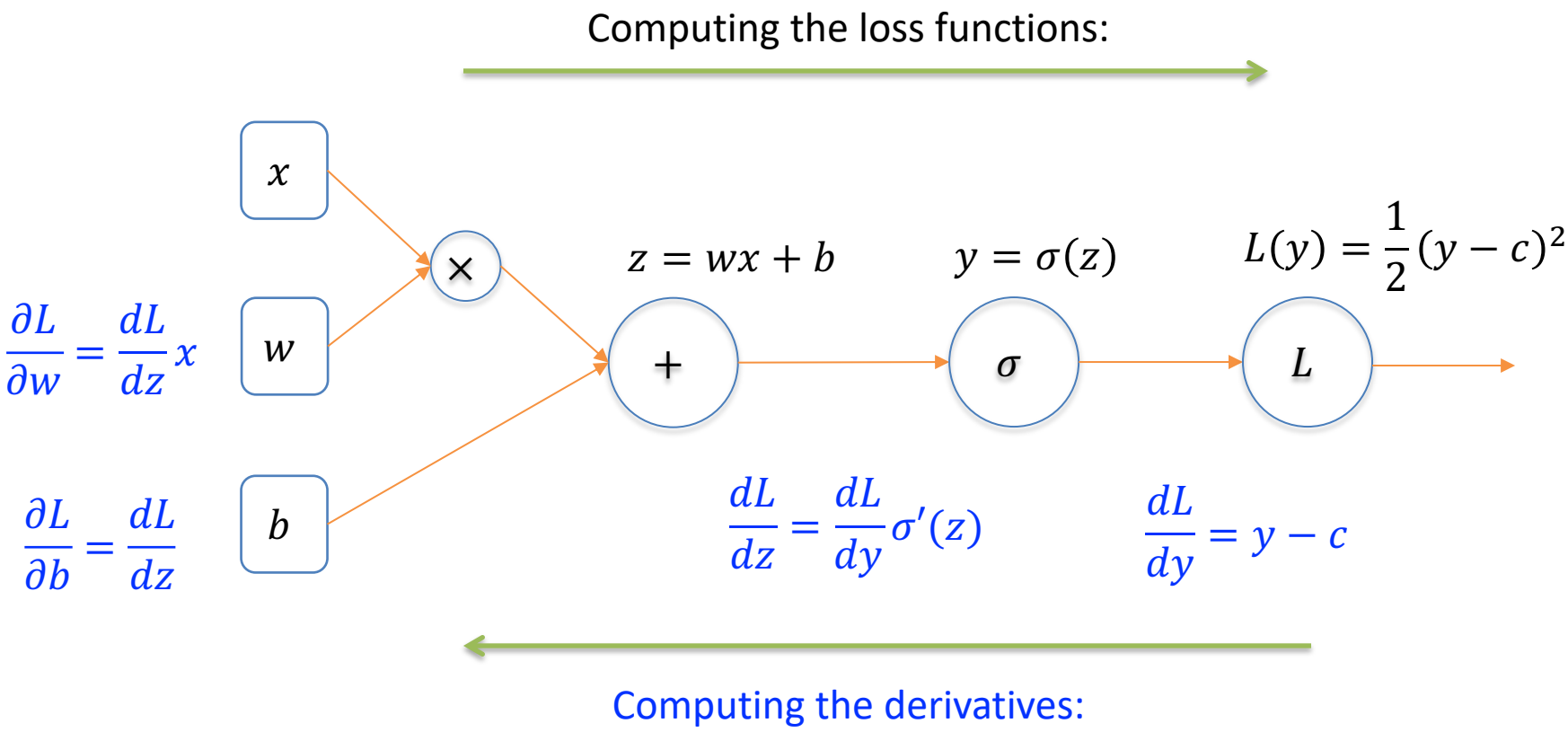
- Computing the derivatives:

$$\frac{dL}{dy} = y - c$$

$$\frac{dL}{dz} = \frac{dL}{dy}\sigma'(z) = \frac{dL}{dy}\sigma(z)\big(1 - \sigma(z)\big)$$

$$\frac{\partial L}{\partial w} = \frac{dL}{dz}x \quad \text{and} \quad \frac{\partial L}{\partial b} = \frac{dL}{dz}$$

Computation Graphs



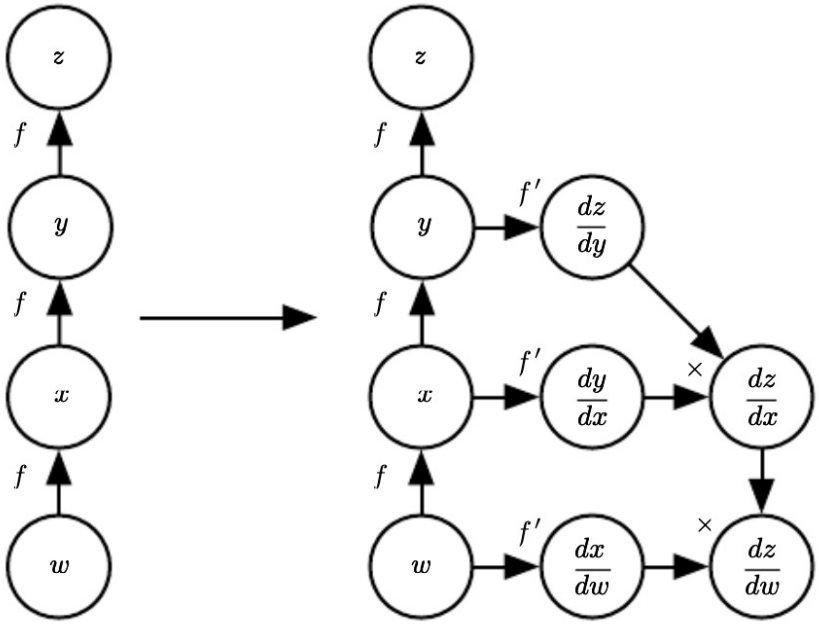
The goal isn't to obtain closed-form solutions for derivative.

The goal is to write a program that efficiently computes the derivatives.

Computational Graphs

- Formalize computation as graphs
- Nodes indicate variables (scalar, vector, tensor or another variable)
- Operations are simple functions of one or more variables
- Our graph language comes with a set of allowable operations

$$z = f(f(f(w)))$$

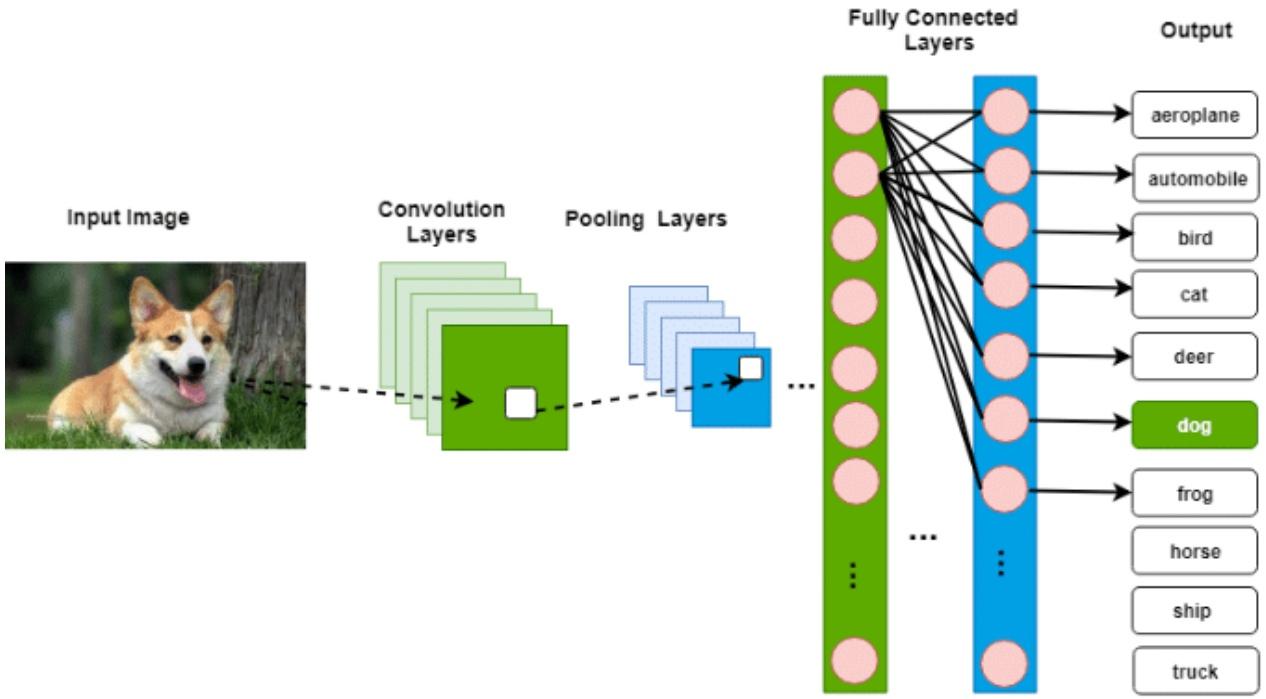


https://www.deeplearningbook.org/contents/mlp.html Sec 6.5

➢ Convolution Neural Networks (CNN)

CNNs are a specific type of neural networks that are generally composed of **convolution layers** and **pooling layers.**

Typical CNN architecture

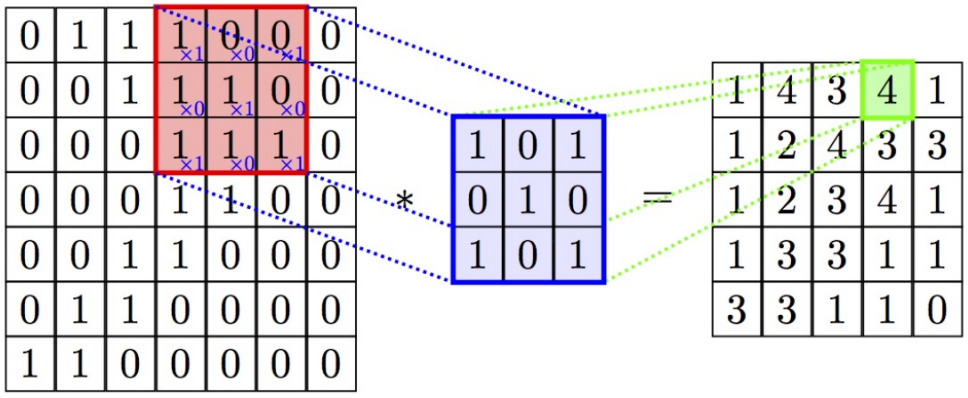


➢ Convolution layers: (generate feature maps.)

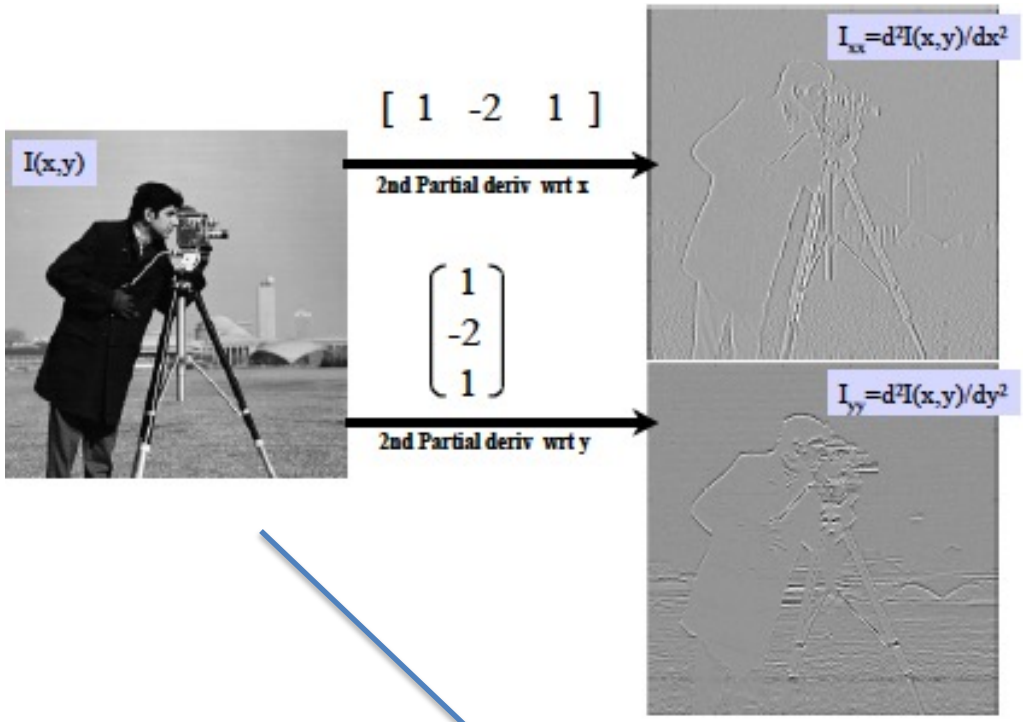
Mathematically, a **convolution** combines two matrices to make a third, by taking the **dot product** of the smaller matrix with every block of the larger.

Suppose B is an $m \times n$ matrix.

$$(A * B)_{i,j} = \sum_{s=0}^{m-1} \sum_{r=0}^{n-1} A_{i+s,j+r} B_{s,r}$$



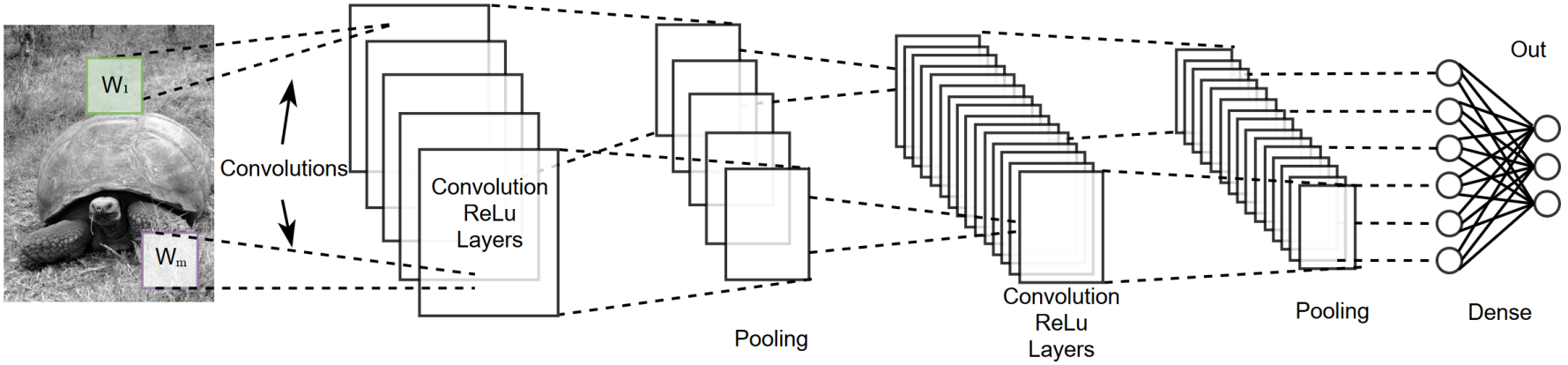
https://cs231n.github.io/convolutional-networks/



Edge detect:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} * I$$

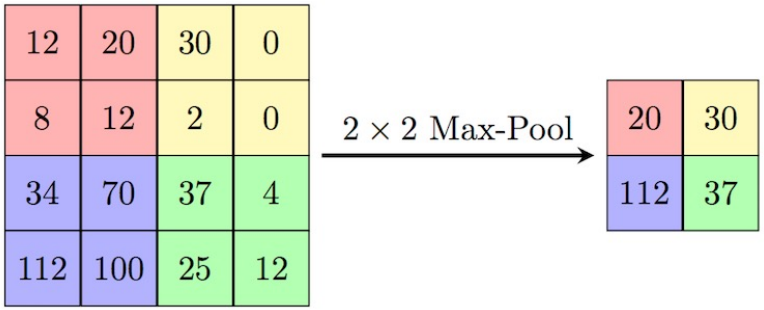
Convolutions In Neural Networks



$$W_1 = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{bmatrix}$$

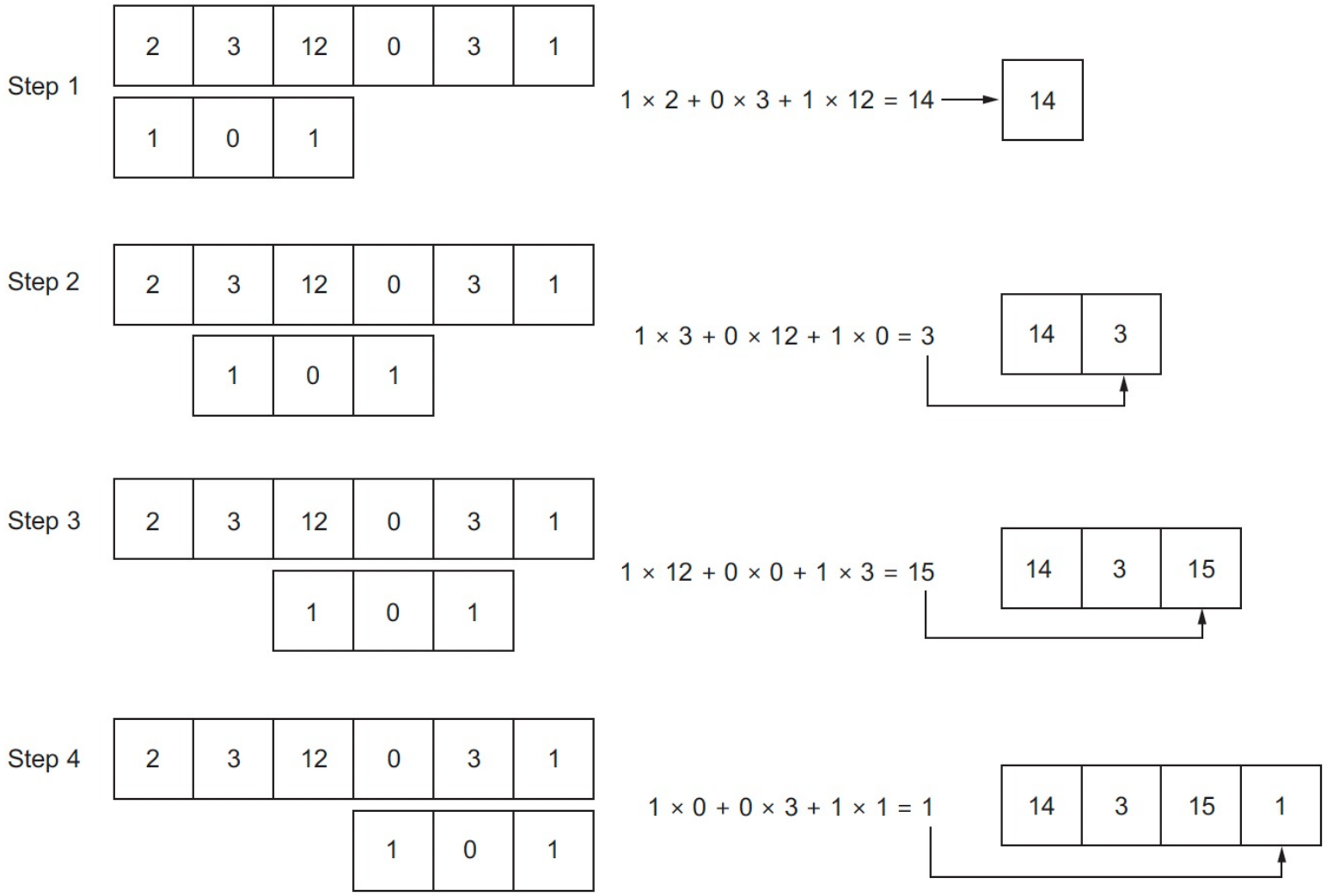
$$\vdots$$

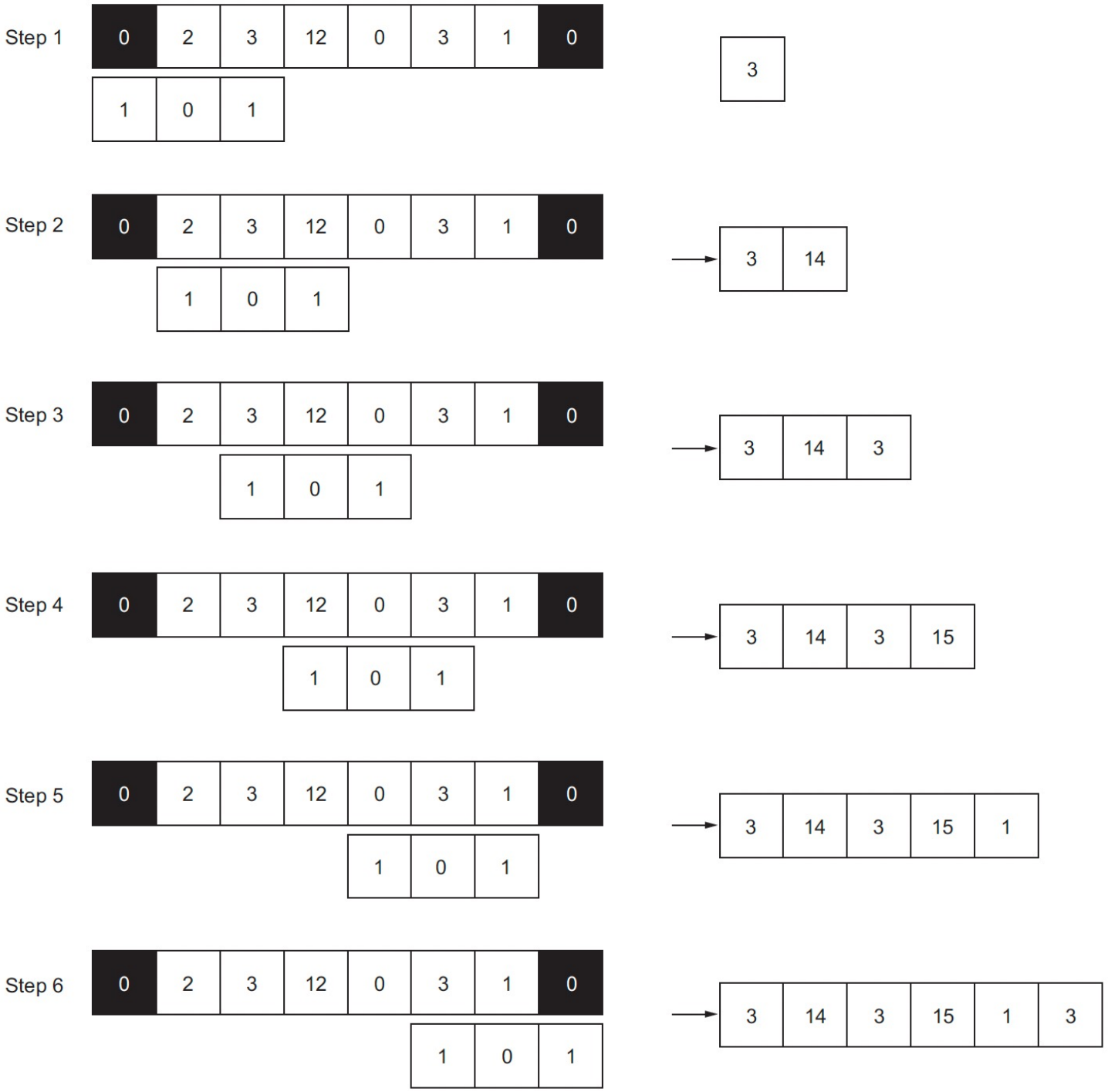
$$W_m = \begin{bmatrix} w_{11}^{(m)} & w_{12}^{(m)} & w_{12}^{(m)} \\ w_{21}^{(m)} & w_{22}^{(m)} & w_{23}^{(m)} \\ w_{31}^{(m)} & w_{32}^{(m)} & w_{33}^{(m)} \end{bmatrix}$$



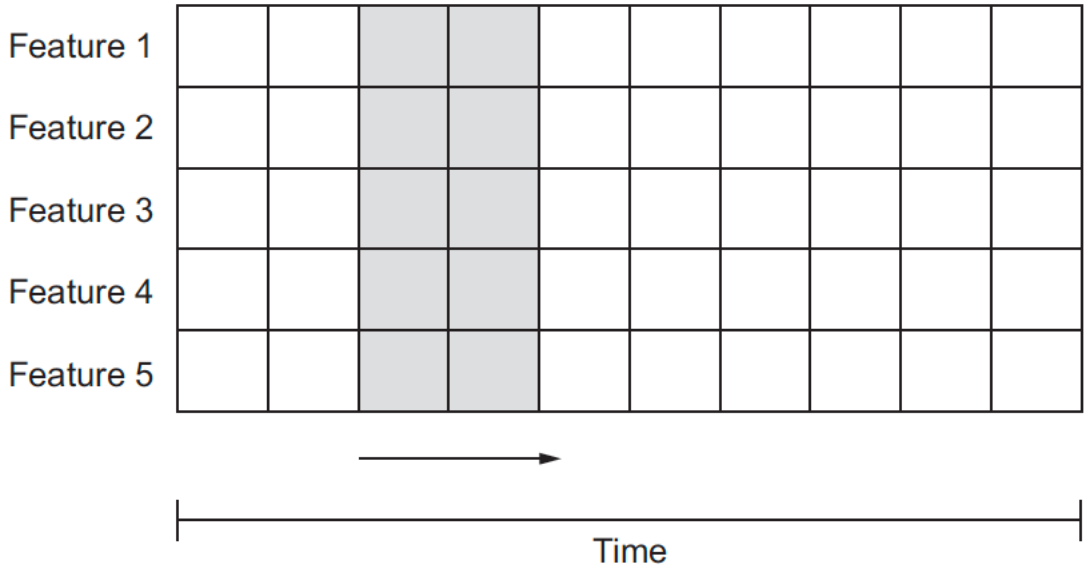
CNN for time series:

1-D CNN



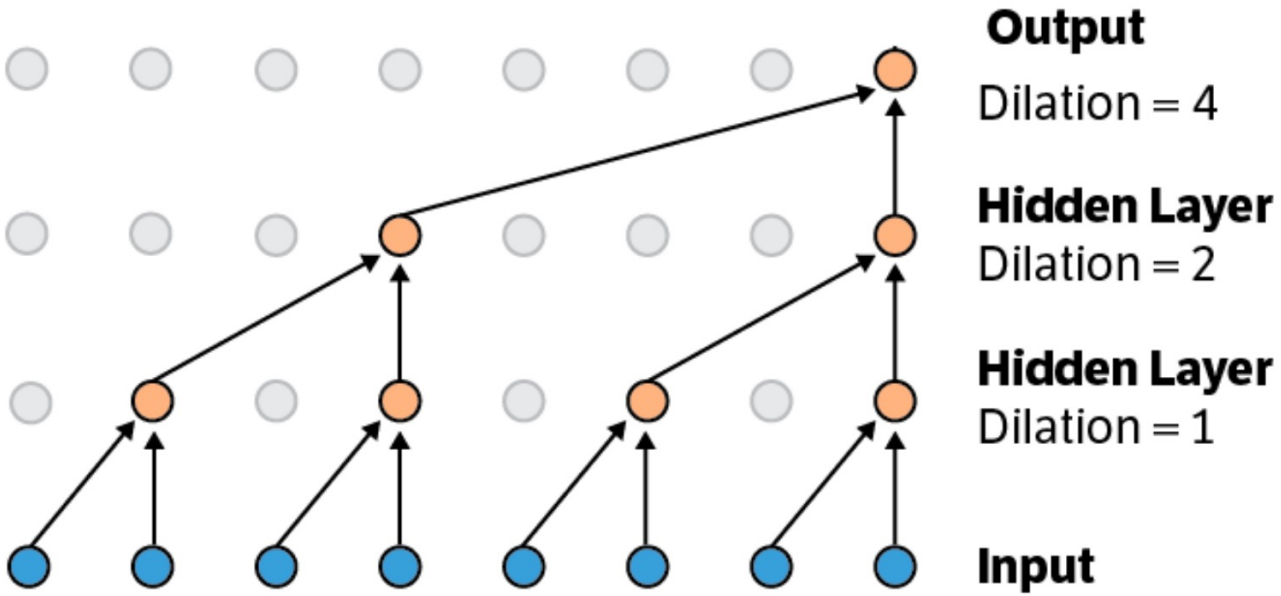

Step 1
0 2 3 12 0 3 1 0
1 0 1
3
Step 2
0 2 3 12 0 3 1 0
1 0 1
3 14
Step 3
0 2 3 12 0 3 1 0
1 0 1
3 14 3
Step 4
0 2 3 12 0 3 1 0
1 0 1
3 14 3 15
Step 5
0 2 3 12 0 3 1 0
1 0 1
3 14 3 15 1
Step 6
0 2 3 12 0 3 1 0
1 0 1
3 14 3 15 1 3

2-D CNN for multiple features:



Dilated Convolutional Neural Networks (DCNN)

DCNN are a specific kind of CNN that allow for a longer reach back to historical information by connecting an exponential number of input values to the output (using dilated convolutional layers of varied dimensions)



Conditional time series forecasting with convolutional neural networks

https://arxiv.org/pdf/1703.04691.pdf

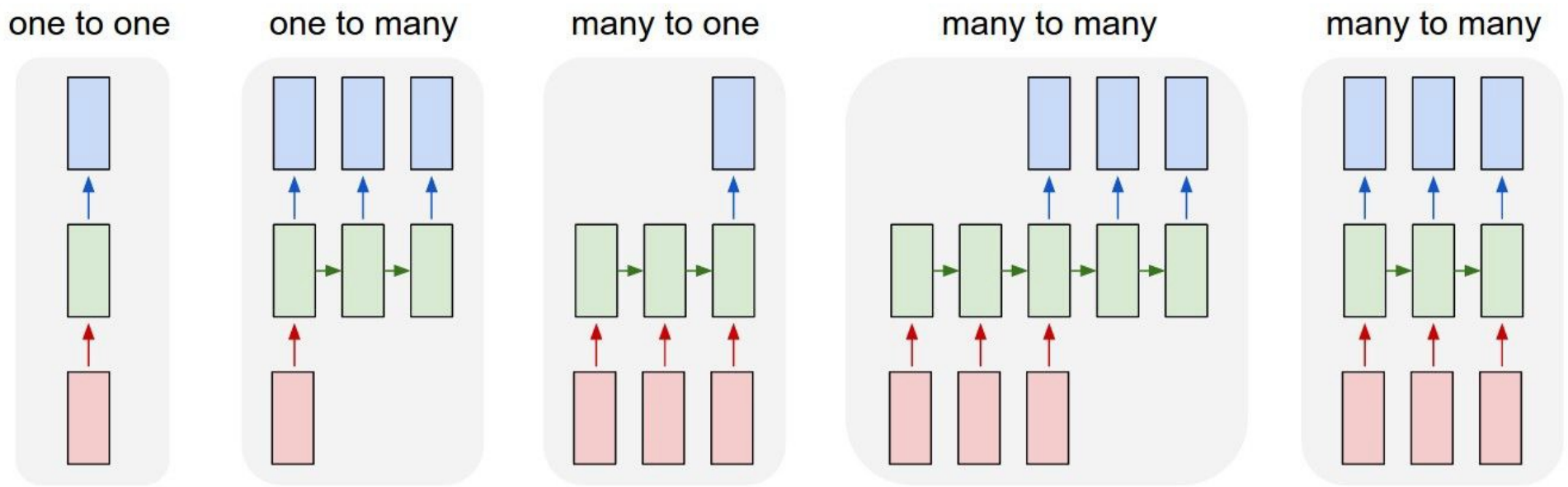
➢ Recurrent Neural Networks: Process Sequences (for sequence data.)

A recurrent neural network (RNN) is a deep learning architecture especially adapted to processing sequences of data.

Long short-term memory (LSTM) is a particular case of a recurrent neural network (RNN). Gated recurrent unit (GRU) are another subtypes of RNN.

One common application of RNN and LSTM is in natural language processing, where words in a sentence have an order.

- **Recurrent Neural Networks examples**

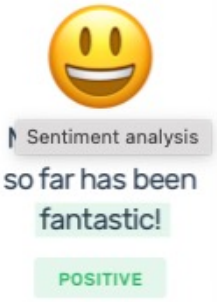


Vanilla Neural Networks

Image Captioning

A baseball player throws a ball

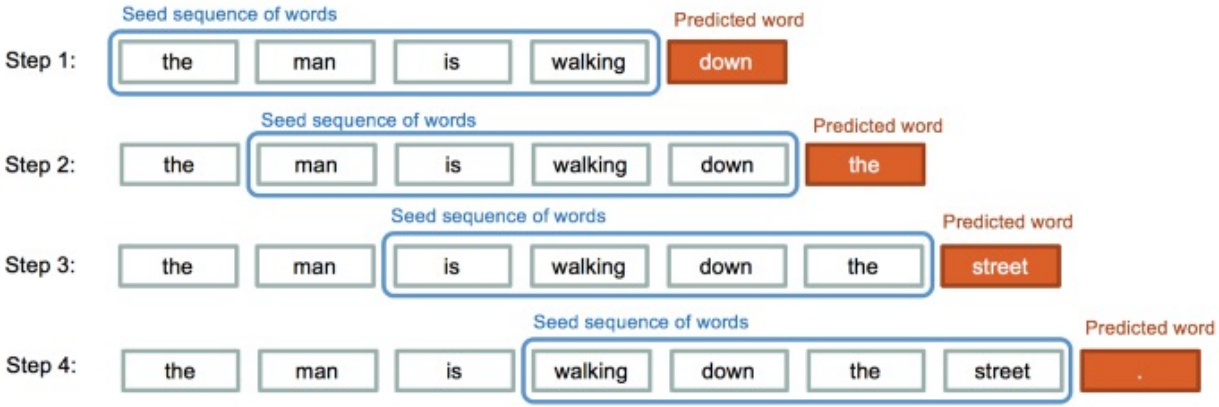
Sentiment Classification



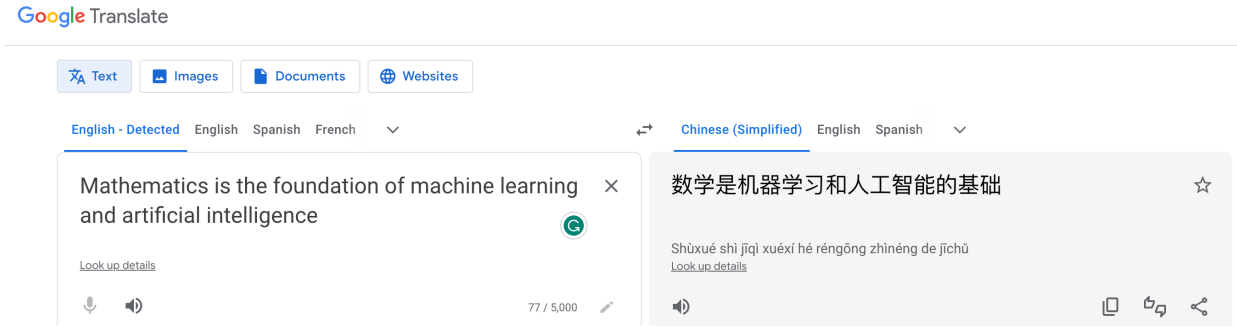
Machine Translation

Video classification on frame level

- Word prediction:



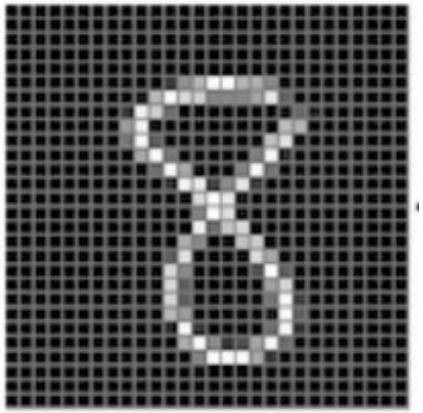
- Machine Translation:



- Sequential Processing of Non-Sequence Data.

1. Classify images by taking a series of “glimpses”

2. Generate images one piece at a time.



Ba, Mnih, and Kavukcuoglu, “Multiple Object Recognition with Visual Attention”, ICLR 2015.
Gregor et al, “DRAW: A Recurrent Neural Network For Image Generation”, ICML 2015

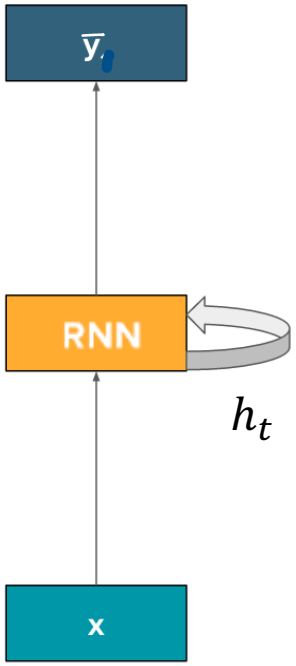
A **recurrent neural network** has **state variables** that can be changed at runtime and that persist between prediction runs.

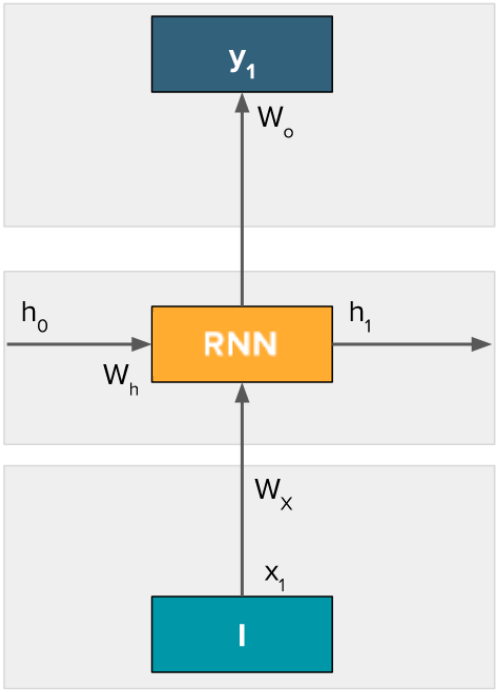
For example, in text prediction an RNN may predict one word at a time while "remembering" its previous predictions.

A **recurrent node** is often represented in “wrapped” form. RNN’s are used extensively in *time series prediction* and *natural language processing*.

Usually want to predict a vector at some time steps

State variables





Sequence of vectors $\vec{x}$ by applying a recurrence formula at every time step:

RNN new hidden state

$$h_t = f_W(h_{t-1}, \vec{x}_t)$$

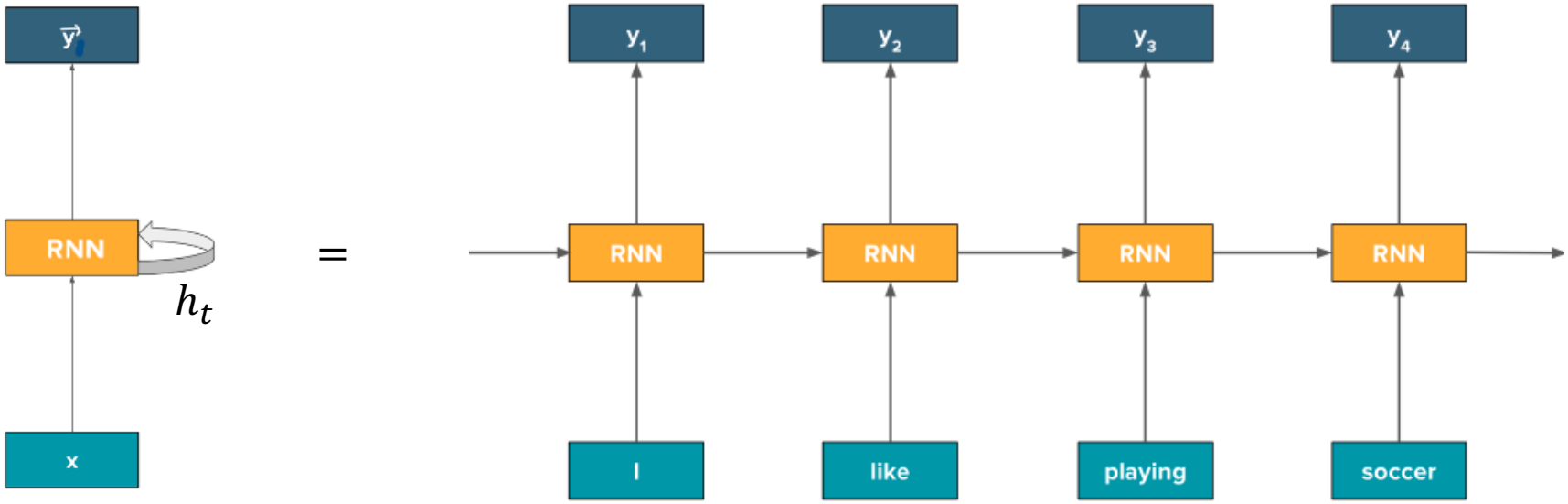
f_W : Function with parameters W.

$\vec{x}_t$: input vector at some time step.

RNN output

$$y_t = g_{W_{hy}}(h_t)$$

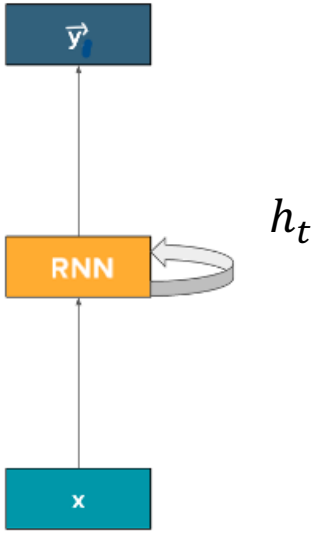
g_W : Function with parameters W_{hy}.



Vanilla Recurrent Neural Networks

State space equations in feedback dynamical systems.

The state consists of a single “hidden” vector h:



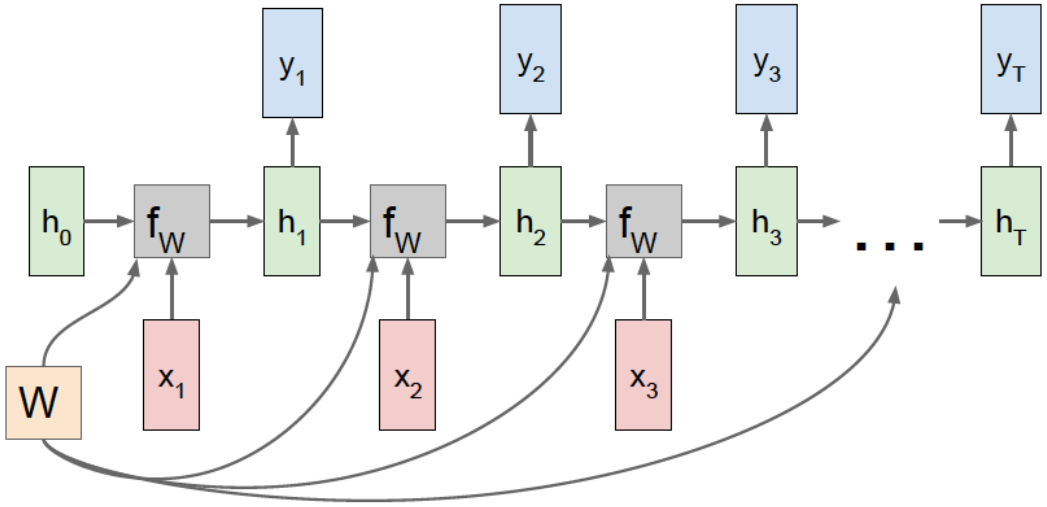
h_t

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

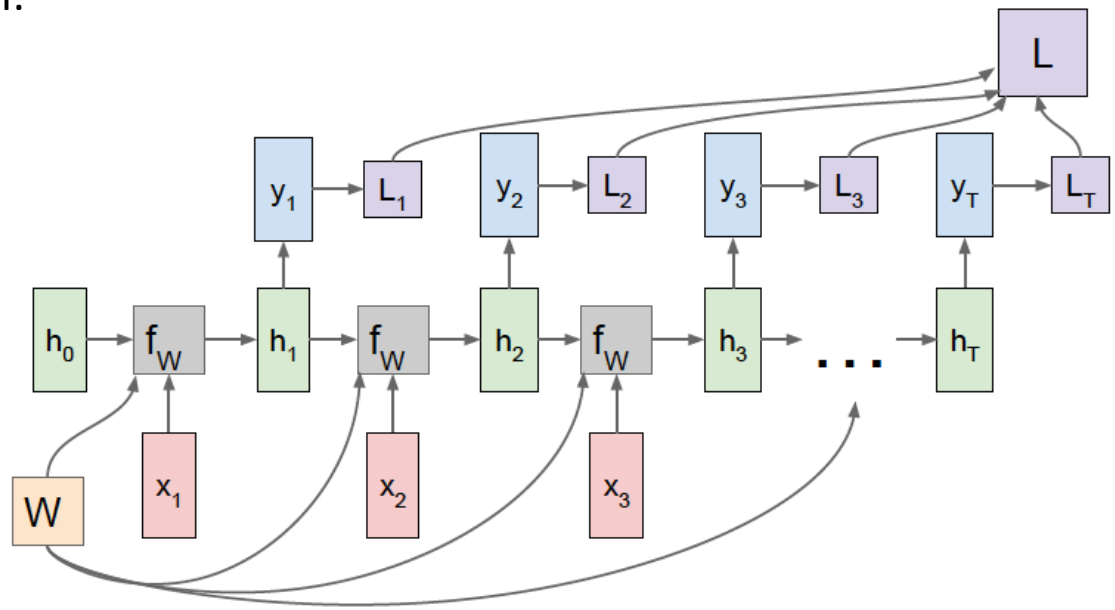
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t \qquad \text{Or } y_t = \text{softmax}\,(W_{hy}h_t)$$

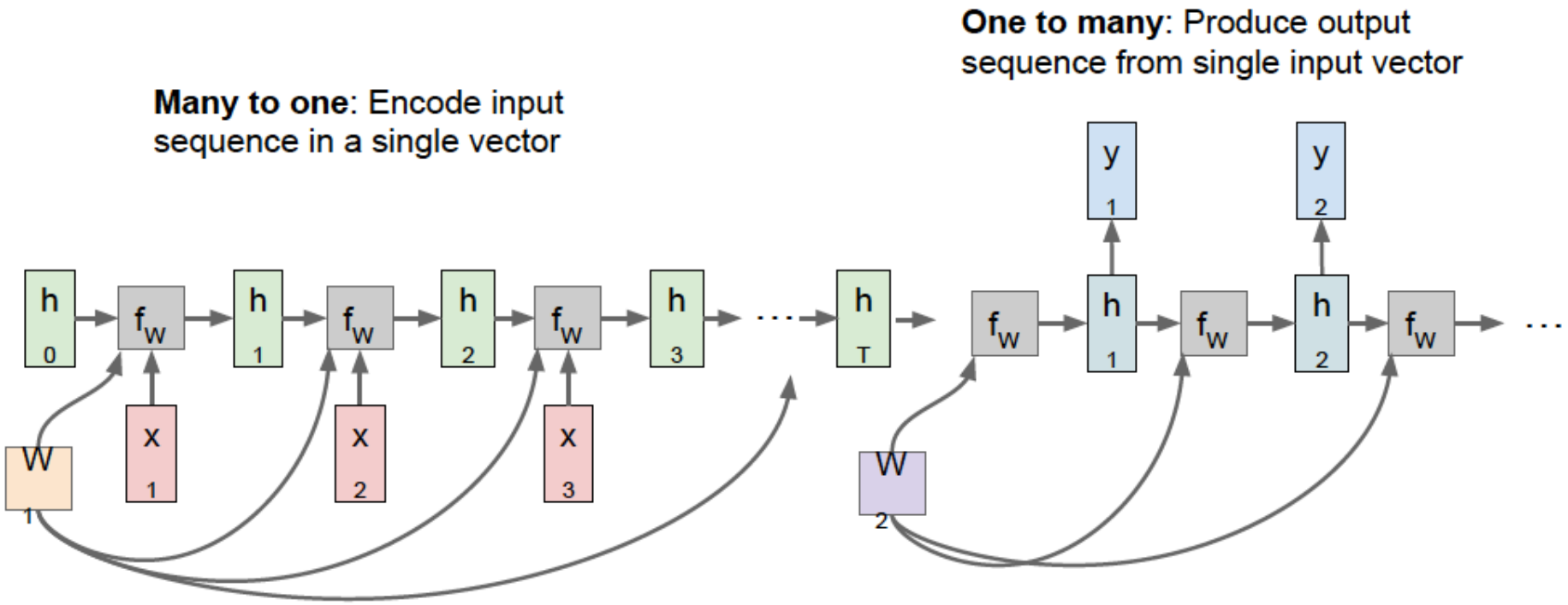
RNN: Computational Graph: Many to Many (Sequence-to-Sequence) (Seq2Seq)



Loss function:



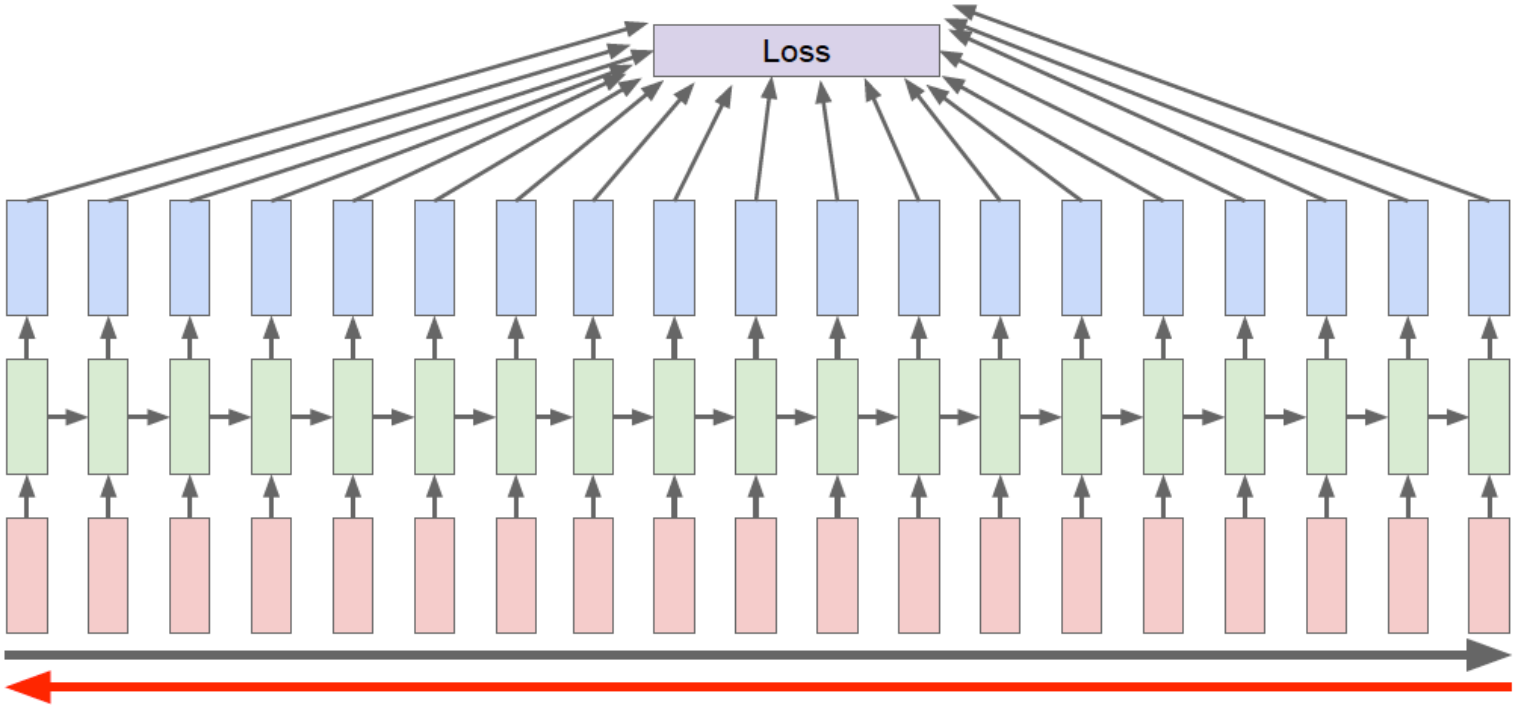
Sequence to Sequence: **Many-to-one + one-to-many**



Backpropagation through time

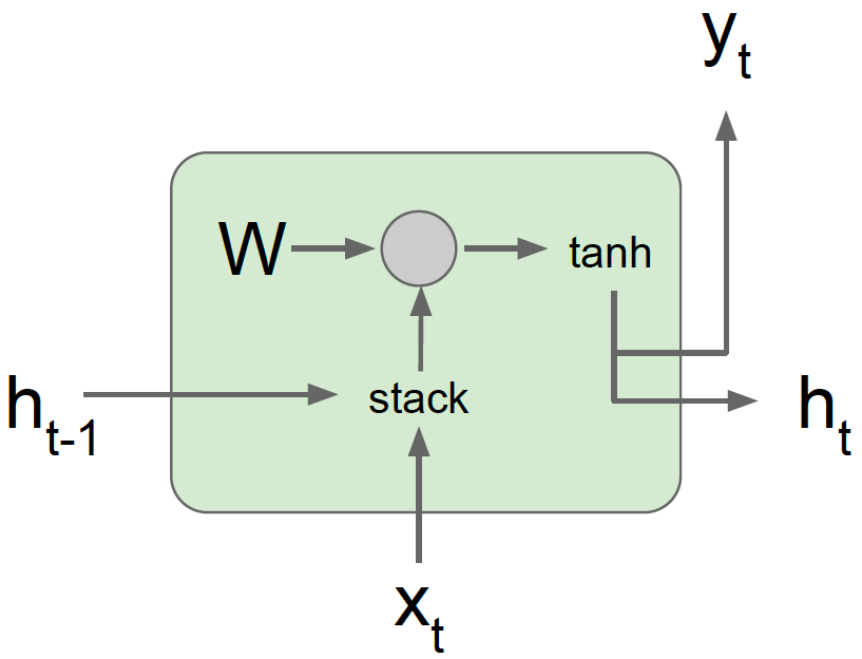
To train an RNN, we unroll it to the the number of steps we require to match out input data shape and then perform standard autodiff backpropagation with input vector and output vector.

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient.



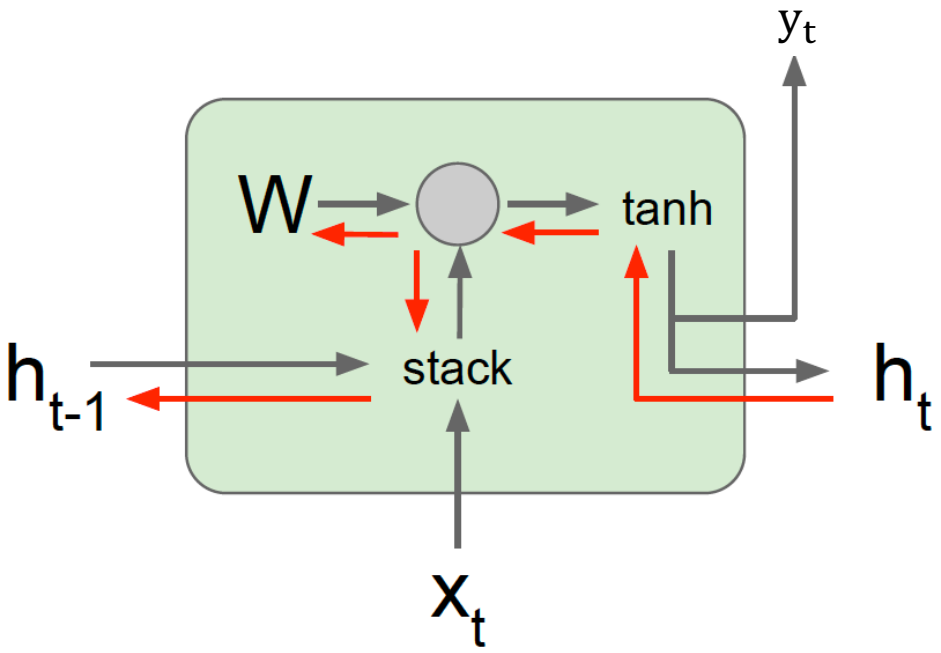
Standard Vanilla RNN Forward Function:

$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)
\end{aligned}
$$



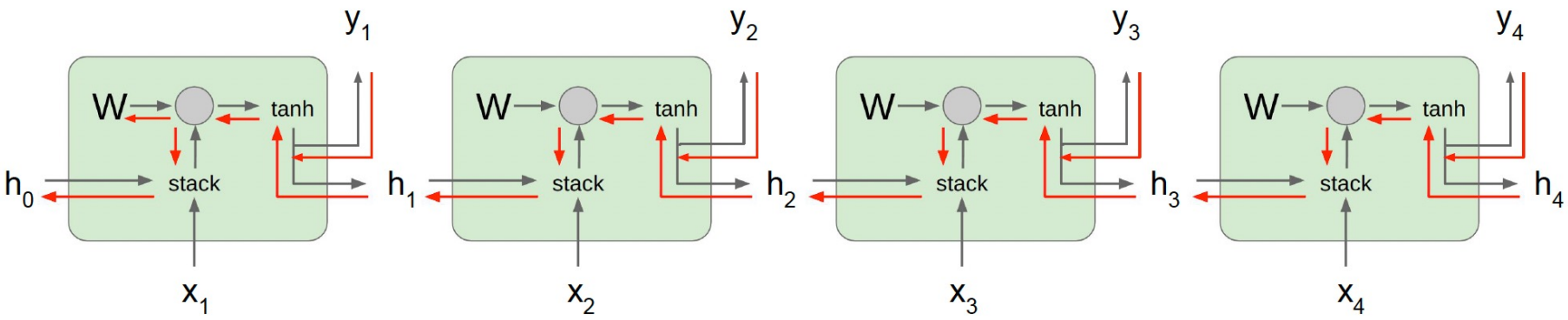
Vanilla RNN Gradient Flow- Backward Gradient

$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}$$



Bengio et al, “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al, “On the difficulty of training recurrent neural networks”, ICML 2013

Vanilla RNN Gradient Flow-Backpropagation



Total Cost: $L = L_1 + L_2 + \cdots + L_T$

$$\frac{\partial L}{\partial W} = \frac{\partial L_1}{\partial W} + \frac{\partial L_2}{\partial W} + \cdots + \frac{\partial L_T}{\partial W}$$

$$\frac{\partial L_t}{\partial W} = \frac{\partial L_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}$$

$$\tanh'(z) = 1 - \tanh^2(z)$$

Explosion and Vanishing of Gradients

Computing gradient of h_0 involves many factors of W (and repeated $\tanh$). The main challenge with RNN's is that training is highly susceptible to gradient **explosion** and **vanishing**, because recurrent nodes lead to highly nonlinear networks.

1. Largest singular value > 1: **Exploding** gradients

   Gradient clipping: Scale gradient if its norm is too big

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```

2. Largest singular value < 1: **Vanishing** gradients

   Change RNN architecture, e.g., Long Short Term Memory (LSTM), Gated recurrent units (GRUs)

➢ Long Short Term Memory (LSTM)

Standard Vanilla RNN

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$= \tanh\left(W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}\right)$$

LSTM

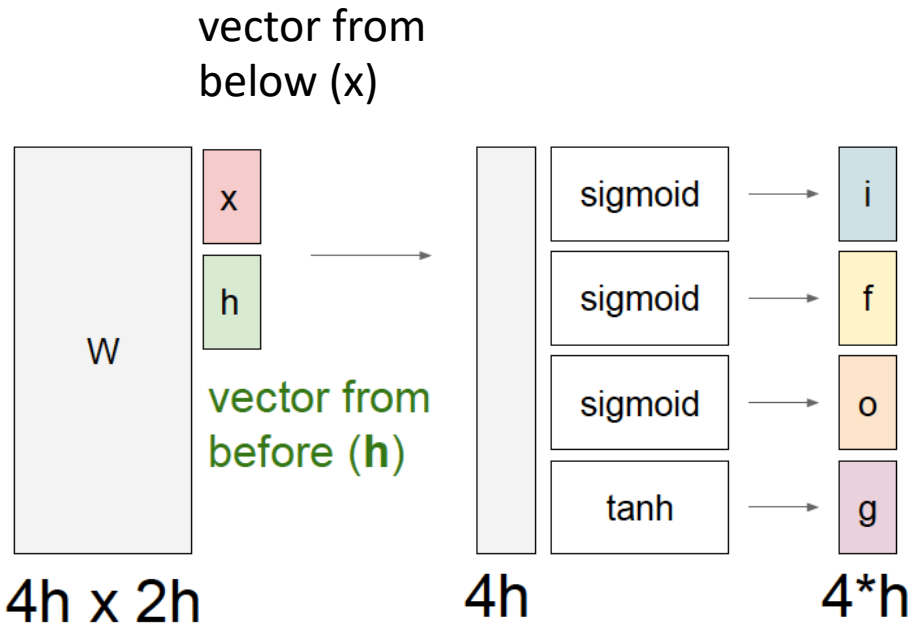
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation 1997

➢ **Long Short Term Memory (LSTM).** [Hochreiter et al., 1997]



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

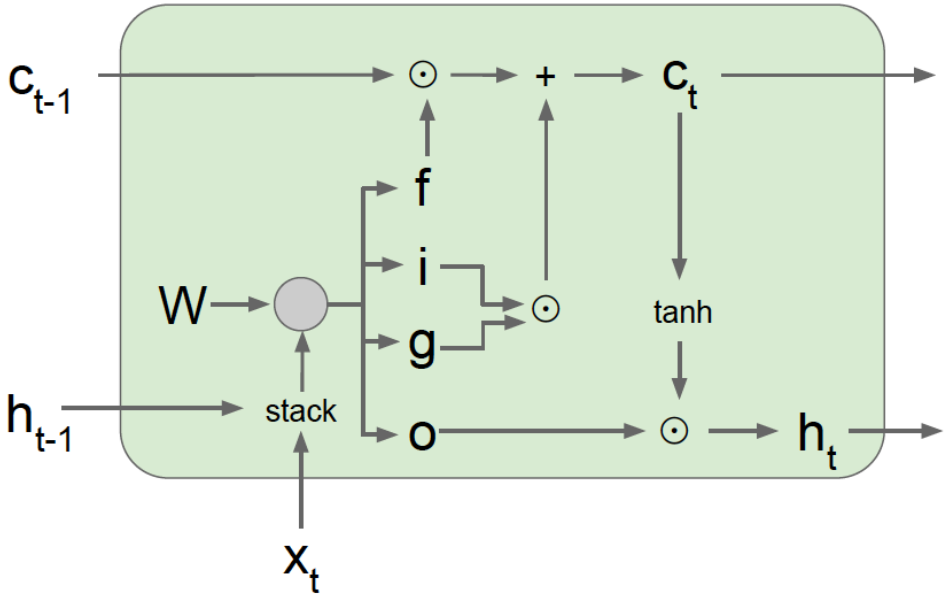
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Gates are a way to **optionally** let information through. They are composed out of a **sigmoid** neural net layer and a **pointwise multiplication** operation.

f: **Forget gate**, Whether to erase cell. (**forget** irrelevant information)
i: **Input gate**, whether to write to cell. (**store** relevant information from current input)
g: **Gate gate**, How much to write to cell.
o: **Output gate**, How much to reveal cell. (Return a filtered version of the cell state.)

The long term memory c_t is a vector whose length is the same as the output.



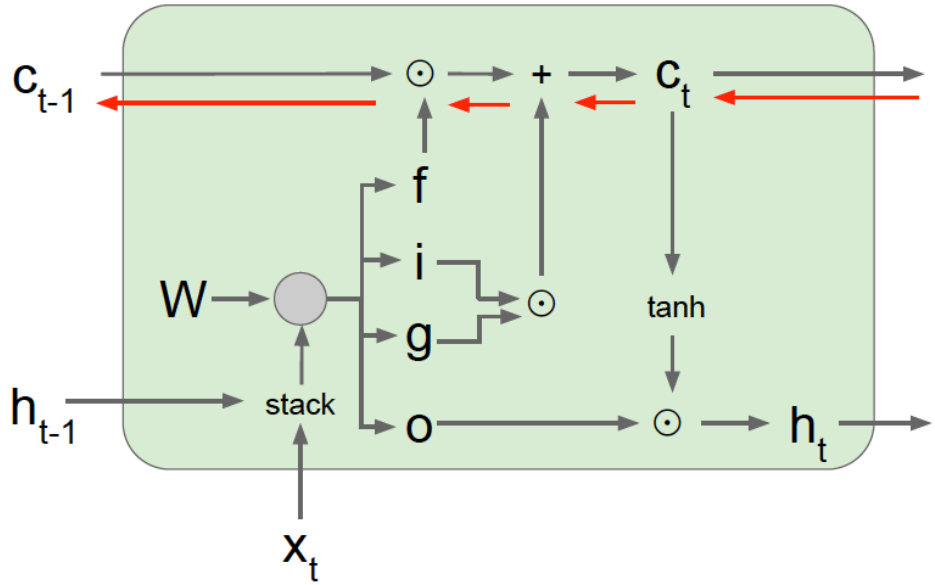
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

In the diagram, the product and sum are the component wise product and sum.

We only need to stipulate how to update the long term memory c_t . We allow the long term memory to "forget" by making at the first multiplication, and to then store new information in the memory, by adding on a masked (non-liner) term dependent on the input x_t , and the previous output h_{t-1}.



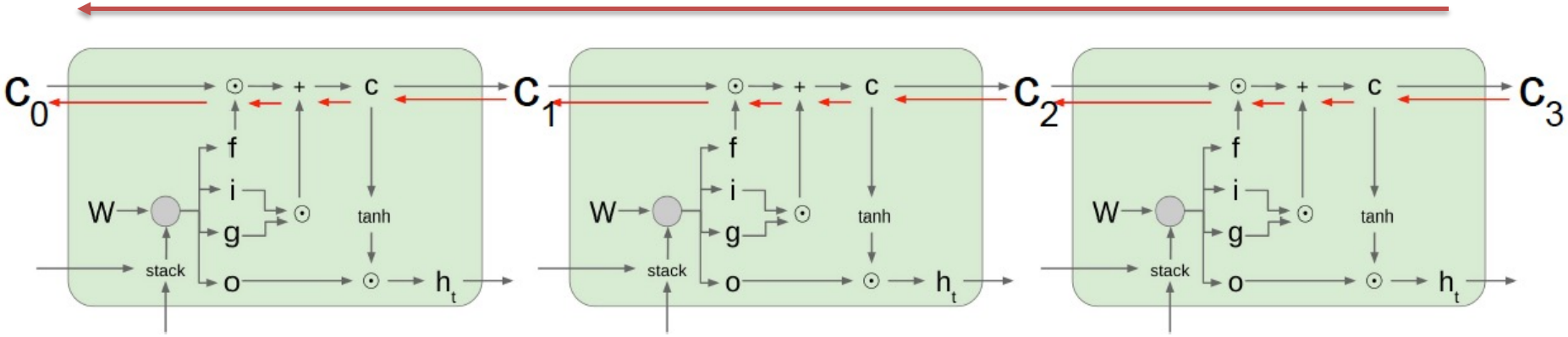
Backpropagation from c_t *to* c_{t-1} only elementwise multiplication by f, no matrix multiply by W.

- The gradient contains the f gate's vector of activations: allows better control of gradients values, using suitable parameter updates of the forget gate f.
- The $f, i, g, and\ o\ gates$ better balance of gradient values.

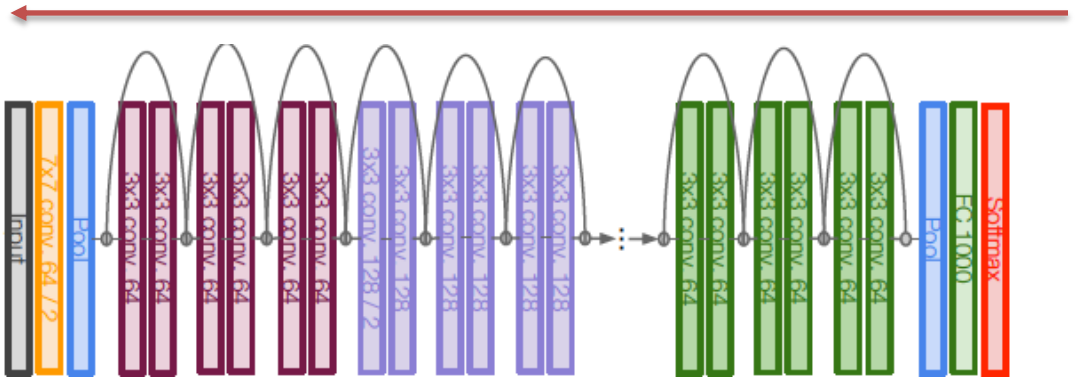
Remarks:

- The LSTM architecture makes it easier for the RNN to preserve information over many timesteps, e.g. if the f = 1 and the i = 0, then the information of that cell is preserved indefinitely.
- By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix W_h that preserves info in hidden state.
- LSTM doesn't guarantee that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies

Uninterrupted gradient flow.



Similar to **ResNet**.



In between: Srivastava et al, “Highway Networks”, ICML DL Workshop 2015

Deep RNN Network:

1. Multilayer RNN:

$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$h \in \mathbb{R}^n$ $\quad W^l \ [n \times 2n]$

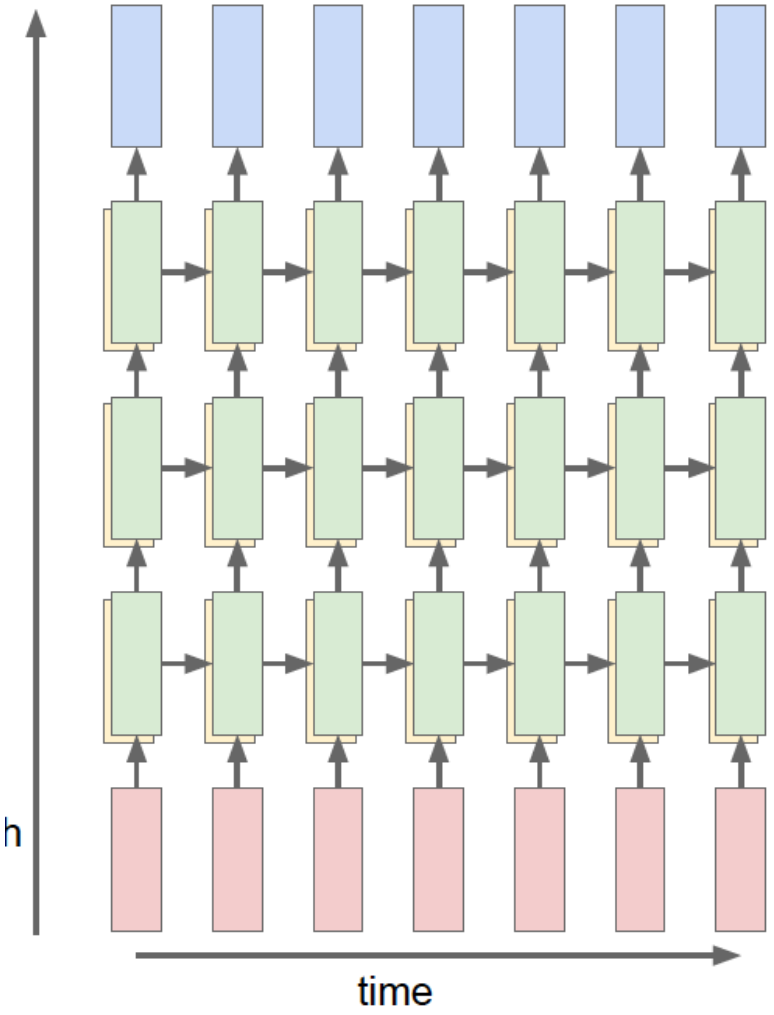
2. LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

$W^l \ [4n \times 2n]$



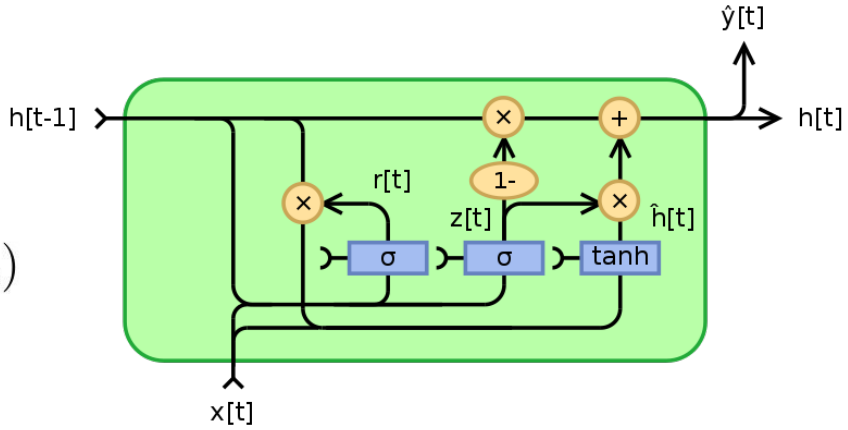
tf.keras.layers.LSTM(num_units))

Gated recurrent units (GRU)

Gated recurrent units (**GRU)** [Learning phrase representations using RNN encoder-decoder for statistical machine translation, Cho et al. 2014]

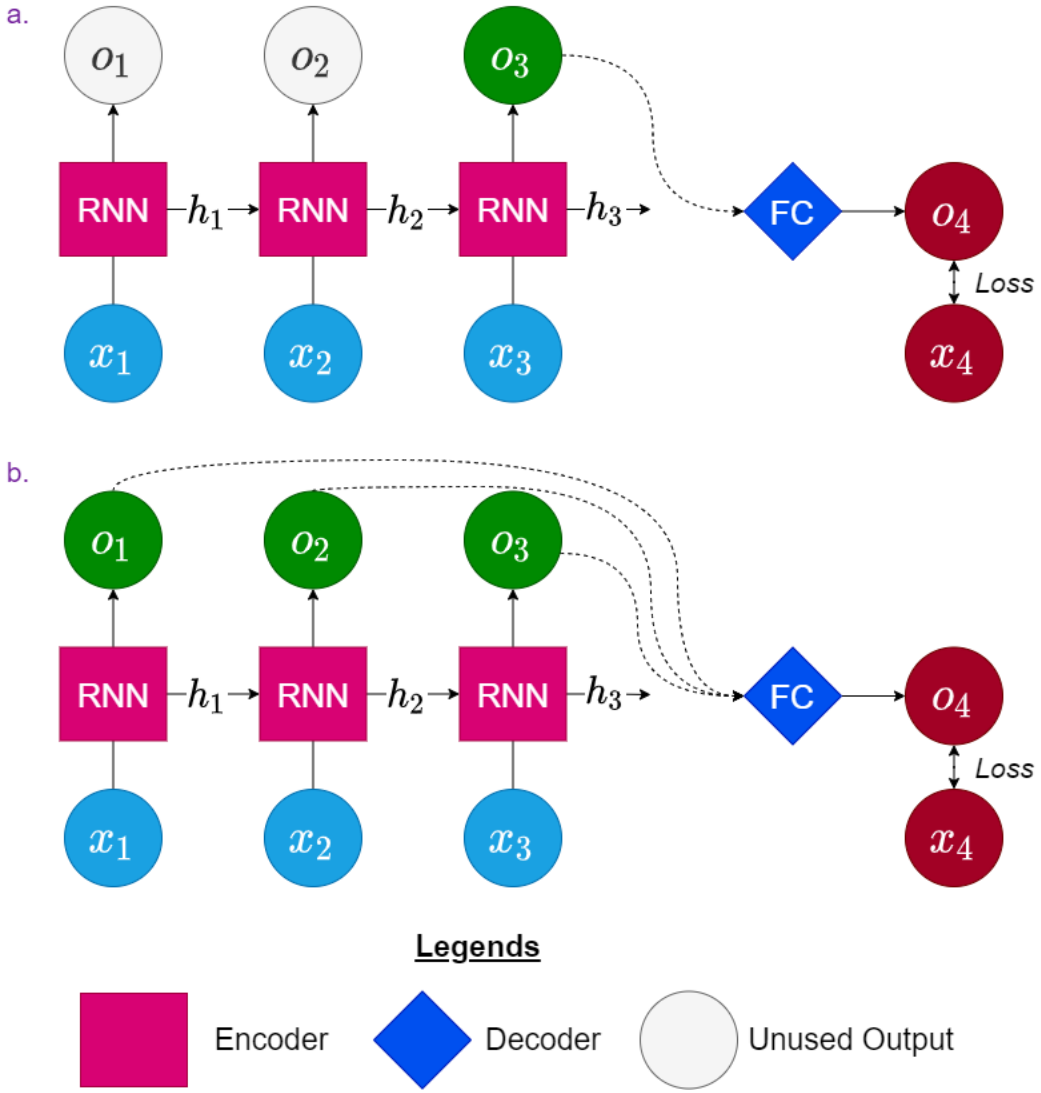
GRU is a simplified version of LSTM.

$$
\begin{aligned}
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\
\tilde{h}_t &= \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t
\end{aligned}
$$



LSTM's are stronger than GRU's: https://arxiv.org/abs/1805.0490856

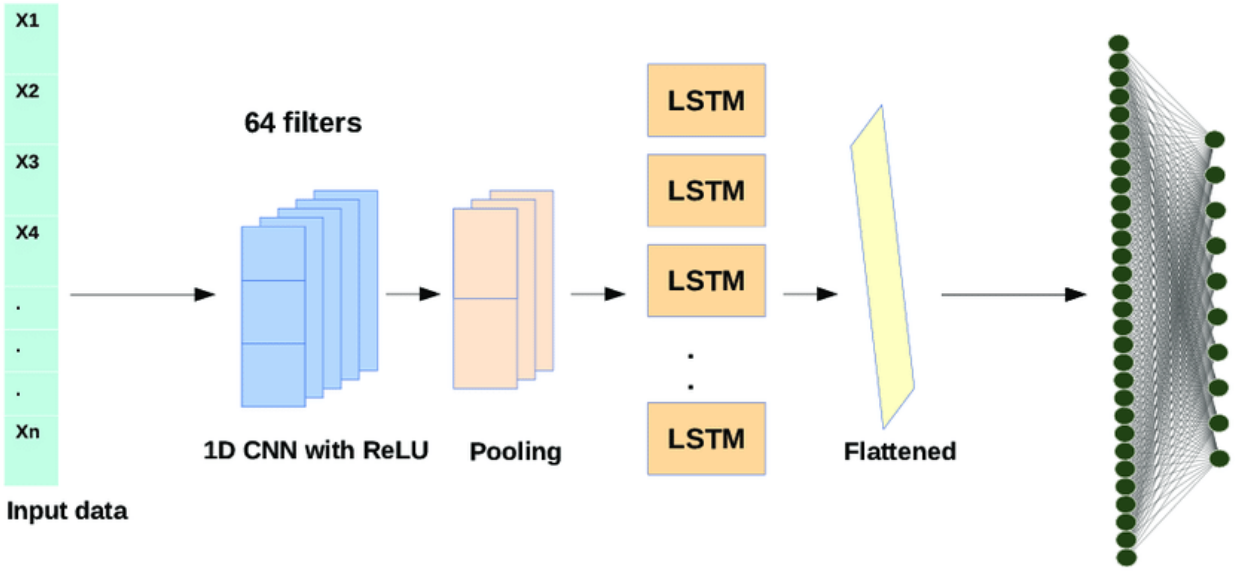
RNN as the encoder and a fully connected layer as the decoder



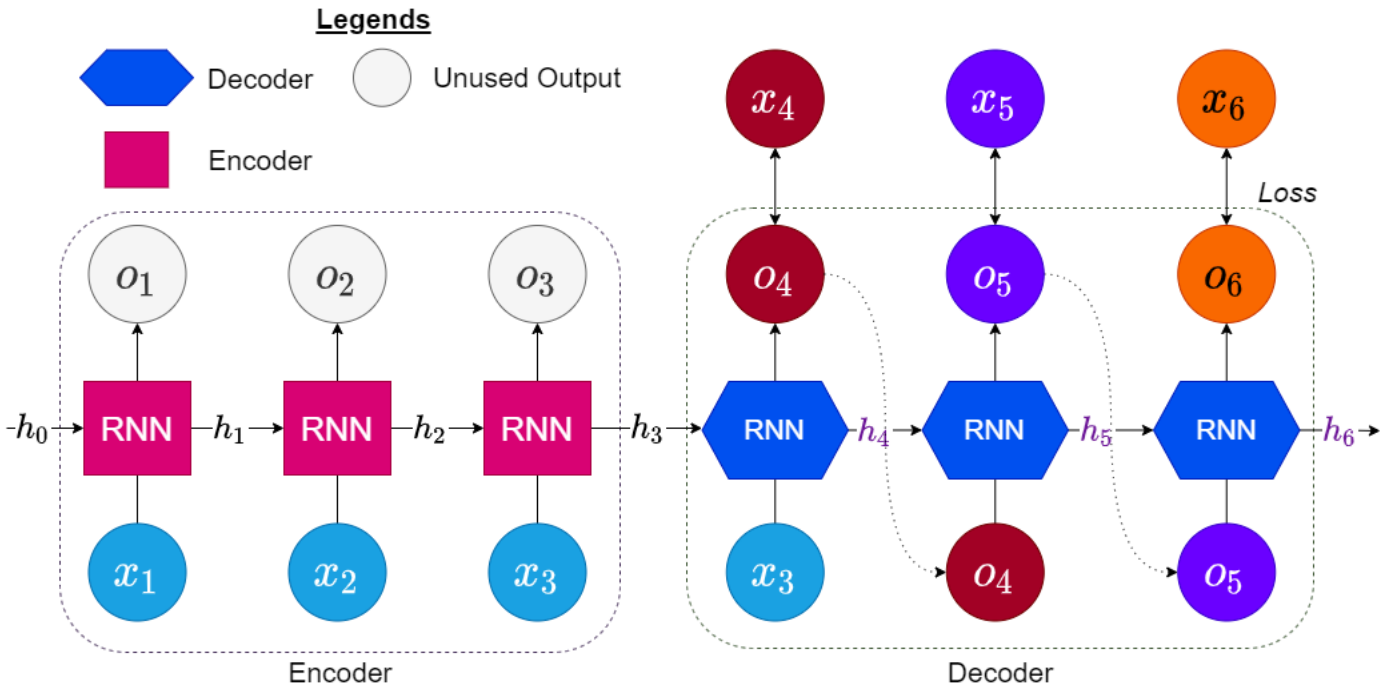
RNN +CNN for Time Series

Sequence-to-sequence modeling for time series has been fairly popular for the past several years. These methods range from vanilla models to advanced industry competitors.

We can also combine the CNN architecture with the LSTM architecture to further improve the performance of our deep learning models.



RNN-to-RNN


Legends
Decoder
Unused Output
Encoder
o_1
o_2
o_3
h_0
RNN
h_1
RNN
h_2
RNN
h_3
x_1
x_2
x_3
Encoder
x_4
x_5
x_6
Loss
o_4
o_5
o_6
RNN
h_4
RNN
h_5
RNN
h_6
x_3
o_4
o_5
Decoder

❑ Attention and Transformers for Time Series

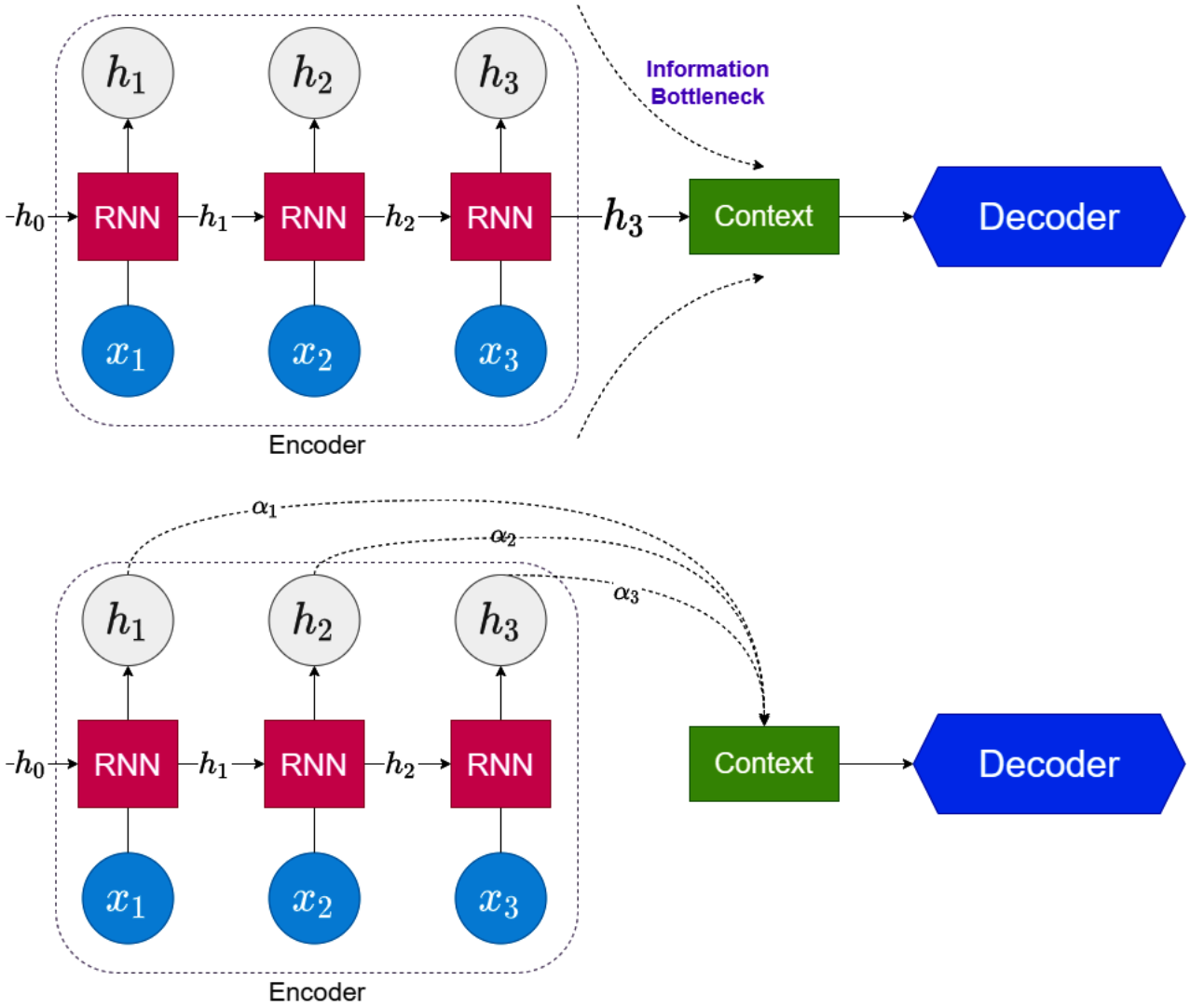
Attention is key concept in deep learning in recent years. Attention can be thought of as a mechanism that allows a neural network to selectively focus on certain parts of an input, while ignoring others.

In 2015, Bahdanau et al. proposed the first known attention model
https://arxiv.org/abs/1409.0473

One of the most popular attention mechanisms is the "self-attention" mechanism, also known as the **Transformer**, introduced by Vaswani et al. in 2017. The Transformer has since become a cornerstone of many state-of-the-art deep learning models, including the famous GPT-3 language model.

Attention Is All You Need: https://arxiv.org/pdf/1706.03762.pdf

Traditional v.s. Attention model in Seq2Seq models

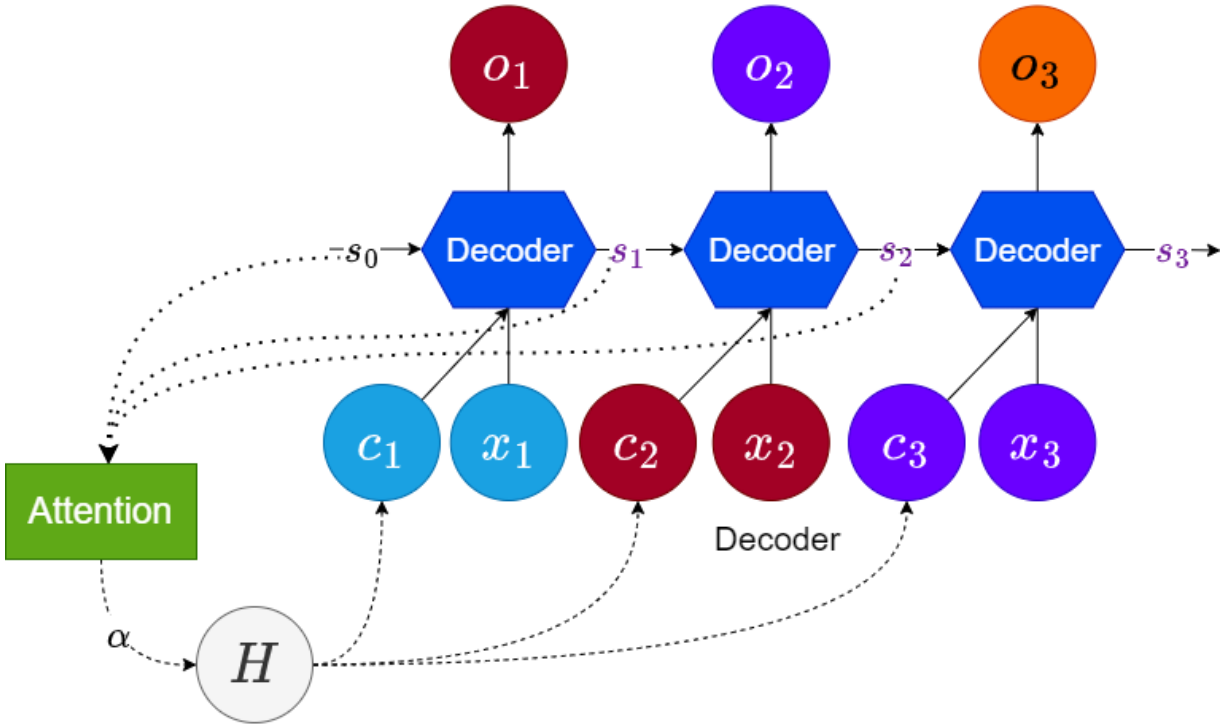


Learn attention weights, $\vec{\alpha}$, for each hidden state corresponding to the input sequence and combine them into a single context vector while decoding

Decoding using attention

h_i: Hidden states generated during encoding process

s_i: Hidden states generated during decoding process



$$c_j = \sum_{i=1}^{T_S} \alpha_{i,j} h_i$$

How these attention weights, α, are calculated?

Attention mechanism -Queries, keys and values

Query $\vec{q}$: A learned parameter that represents the current token in the sequence that we want to compute the attention weights for.

Key K: A set of learned parameter that represents each token in the sequence. The key vector is used to compute the similarity between the query vector and each token in the sequence.

Value V: A set of learned parameter that represents each token in the sequence. The value vector is used to compute the weighted sum of the other tokens' values. (In many cases, K and V are the same).

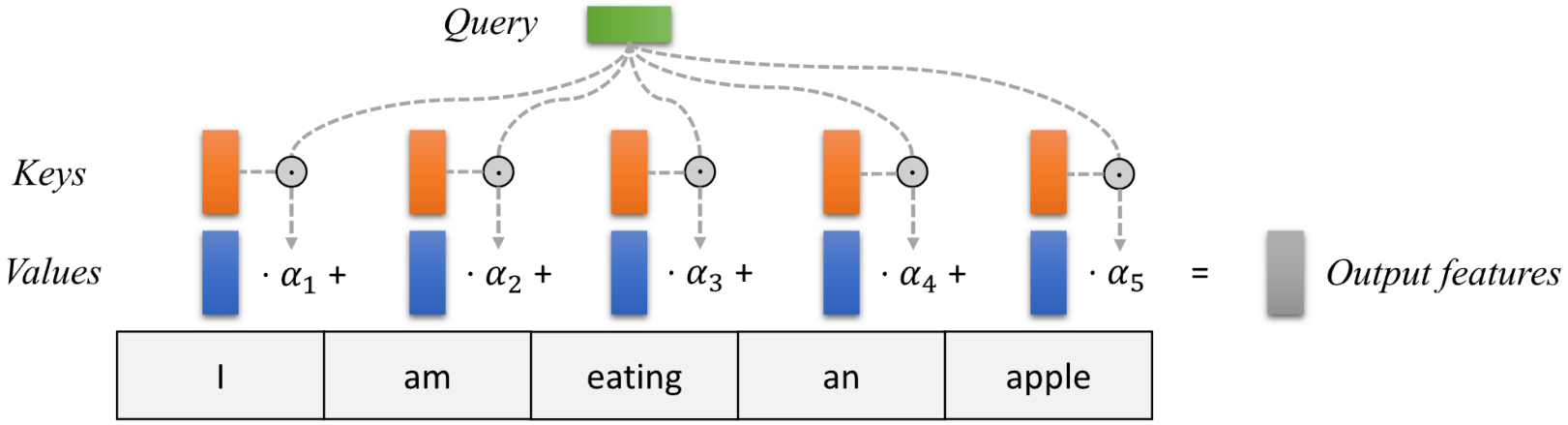
Think of an attention model as learning an attention distribution (α)

$$T(\vec{q}, K, V) = \sum_i p\left(a\left(\overrightarrow{k_i}, \vec{q}\right)\right) \times \overrightarrow{v_i}$$

The **alignment function** $a(-,-)$ that calculates a similarity **score** between the queries and keys. (Usually it is dot product $\vec{k}_i^T \vec{q}$ or weighted dot product $\vec{k}_i^T W \vec{q}$)

A distribution function $p(-)$ converts this **score** into attention weights that sum up to 1.

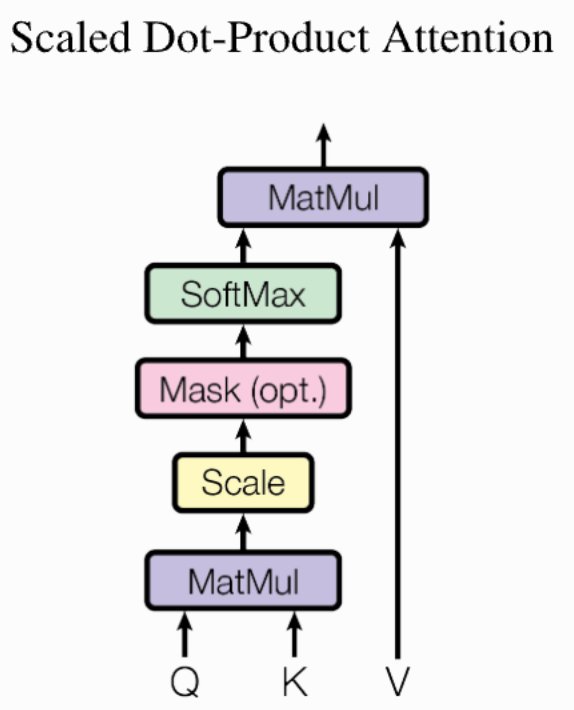
Illustration of with key, query and value transformations.



$$\alpha_i = \frac{\exp\left(f_{attn}\left(\text{key}_i, \text{query}\right)\right)}{\sum_j \exp\left(f_{attn}\left(\text{key}_j, \text{query}\right)\right)}, \quad \text{out} = \sum_i \alpha_i \cdot \text{value}_i$$

https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

Scaled Dot Product Attention



$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Attention Is All You Need https://arxiv.org/abs/1706.03762

Transformers – Attention is all you need

Just as the title of the paper implies, they explored an architecture that used attention (scaled dot product attention) and threw away recurrent networks altogether. *Both the encoder and decoder are non-recurrent.*

(a) Animals in a playground

(b) Our attention is drawn to animals

Transformers in time series

Time series have a lot of similarities with NLP because both deal with sequences data.

Transformers in Time Series: A Survey https://arxiv.org/pdf/2202.07125.pdf

Are Transformers Effective for Time Series Forecasting? https://arxiv.org/pdf/2205.13504.pdf

Textbooks with examples:

Time Series Forecasting in Python:
https://github.com/marcopeix/TimeSeriesForecastingInPython

Modern Time Series Forecasting with Python
https://github.com/PacktPublishing/Modern-Time-Series-Forecasting-with-Python

Conditional time series forecasting with convolutional neural networks,(2017)

https://arxiv.org/pdf/1703.04691.pdf

Time Series Forecasting Examples Using Deep Learning

https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras (LSTM-Python)

https://machinelearningmastery.com/how-to-develop-convolutional-neural-network-models-for-time-series-forecasting/ (CNN-Python)

MATLAB-Forecast time series data using a long short-term memory (LSTM) network

https://www.mathworks.com/help/deeplearning/ug/time-series-forecasting-using-deep-learning.html