Business Forecasting

ADIA Course

Day 4 – Session 1

Dr. Tanujit Chakraborty

Sorbonne University, Abu Dhabi, UAE

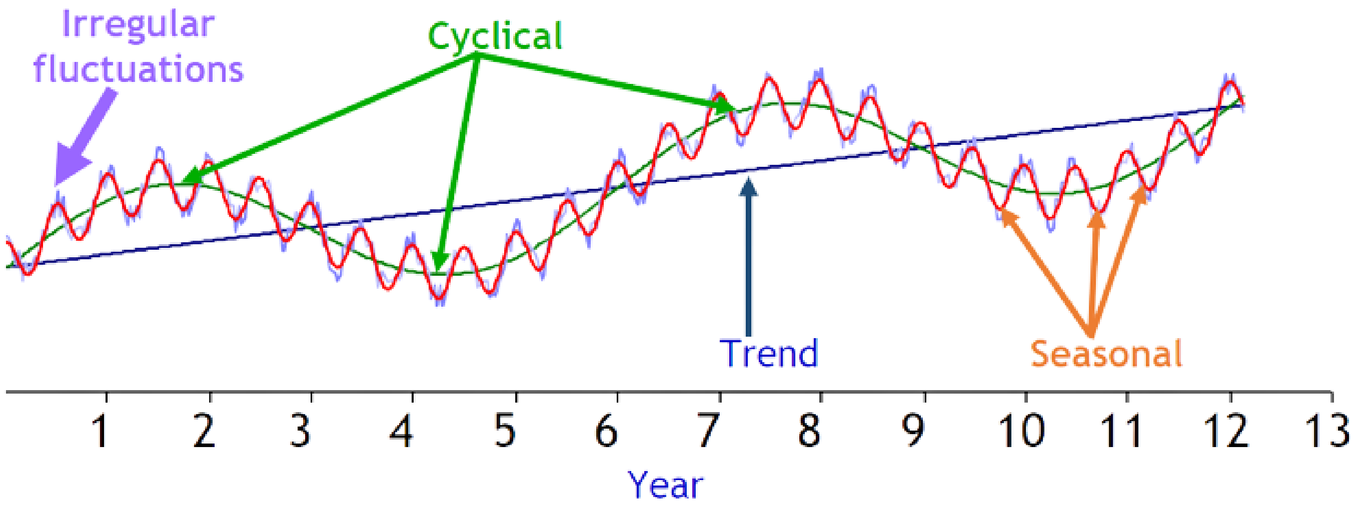
Sorbonne Centre for Artificial Intelligence, Paris, France

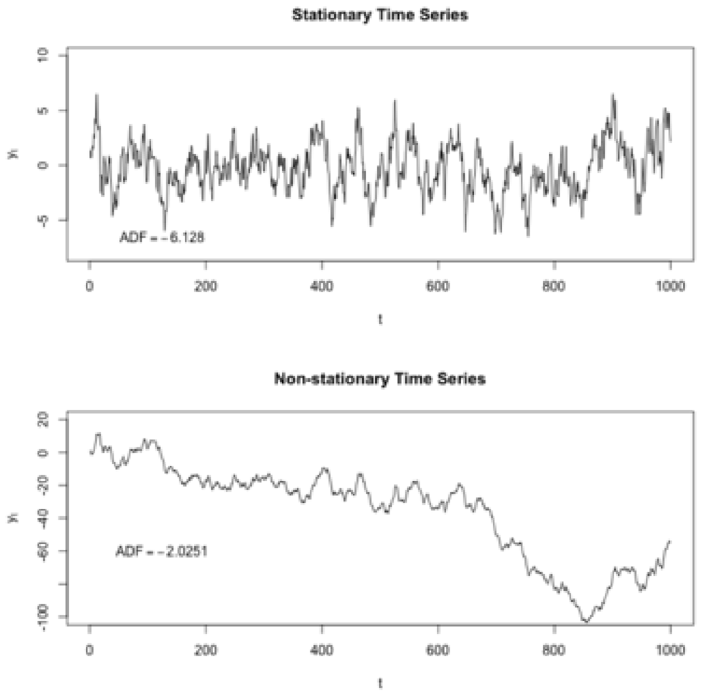
Research Areas: Time Series Forecasting, Machine Learning, Econometrics, Health Data Science

July 3, 2025

Learning from Time-Series Data

- The input is a sequence of (non-i.i.d.) examples $y_1, y_2, \dots, y_t$.
- The problem may be supervised or unsupervised, e.g.,
  - Forecasting: Predict y_{t+1} using $y_1, y_2, \dots, y_t$
  - Cluster the examples or perform dimensionality reduction / Anomaly detection
- Evolution of time-series data can be attributed to several factors



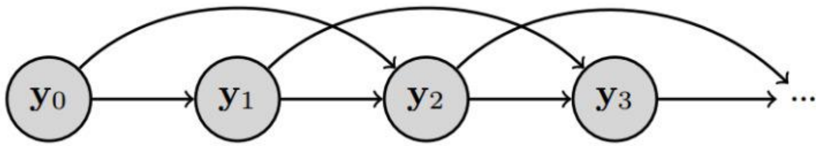


- Teasing apart these factors of variation is also an important problem.

Auto-regressive Models

- Auto-regressive (AR): Regress each example on p previous lagged values - AR(p) model

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \varepsilon_t,$$



Auto-regressive Model (shown above: 2nd order AR)

- Moving Average (MA): Regress each example on q previous stochastic errors - MA(q) model

$$y_t = c + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_q \varepsilon_{t-q},$$

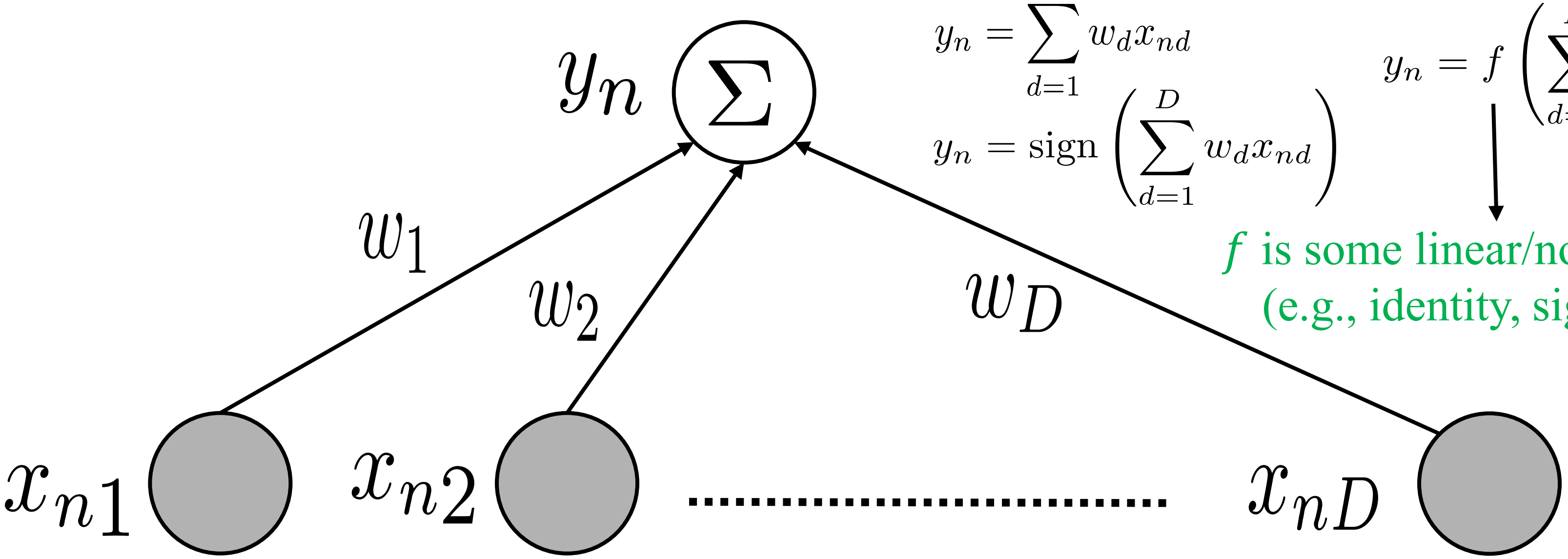
- Auto-regressive Integrated Moving Average (ARMA): Regress each example of p previous lagged values and q previous stochastic errors

$$y'_t = c + \phi_1 y'_{t-1} + \cdots + \phi_p y'_{t-p} + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q} + \varepsilon_t,$$

where y_t' is the differenced series (if the data is nonstationary and differencing is applied). We call this an **ARIMA**(p, d, q) model.

Limitation of Linear Models

- Linear models: Output produced by taking a linear combination of input features



$$y_n = \sum_{d=1}^{D} w_d x_{nd}$$

$$y_n = \text{sign}\left(\sum_{d=1}^{D} w_d x_{nd}\right)$$

$$y_n = f\left(\sum_{d=1}^{D} w_d x_{nd}\right)$$

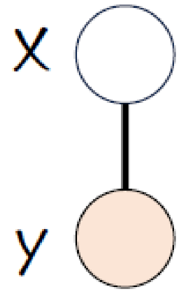
f is some linear/nonlinear function (e.g., identity, sign, or sigmoid)

- A basic unit of the form $y = f(\boldsymbol{w}^\top \boldsymbol{x})$ is known as the "Perceptron" (not to be confused with the Perceptron "algorithm", which learns a linear classification model)
- This can't however learn nonlinear functions or nonlinear decision boundaries

Exercise: Linear Layer

$$\begin{array}{cc|c} W & b & \\ 3 & 0 & y \end{array} \qquad \begin{array}{c} X \\ 4 \\ 1 \end{array}$$

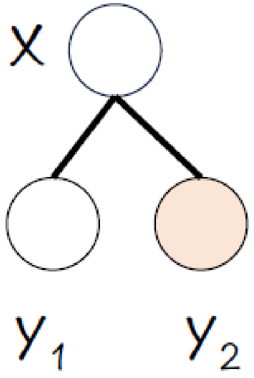
$$y = [W|b]\cdot[X|1] = WX+b$$



Exercise 1

12

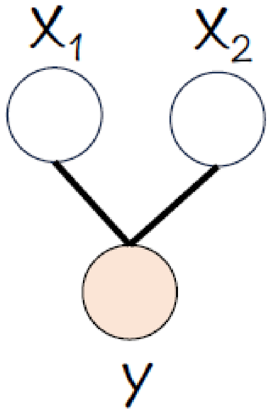
$$\begin{array}{cc} W & b \\ 3 & 2 \\ 3 & -2 \end{array} \qquad \begin{array}{cc} X & \\ 4 & \\ 1 & \\ 14 & y_1 \\ & y_2 \end{array}$$



Exercise 2

10

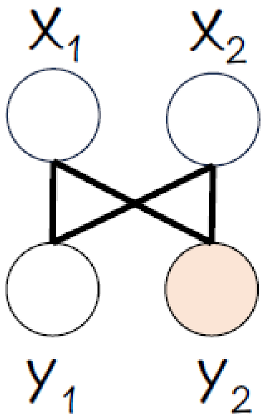
$$\begin{array}{ccc} W & & b \\ 1 & -1 & 0 \end{array} \qquad \begin{array}{cc} 2 & X_1 \\ 1 & X_2 \\ 1 & \\ & y \end{array}$$



Exercise 3

1

$$\begin{array}{ccc} W & & b \\ 1 & 1 & 2 \\ 1 & -1 & 2 \end{array} \qquad \begin{array}{cc} 2 & X_1 \\ 1 & X_2 \\ 1 & \\ 5 & y_1 \\ & y_2 \end{array}$$



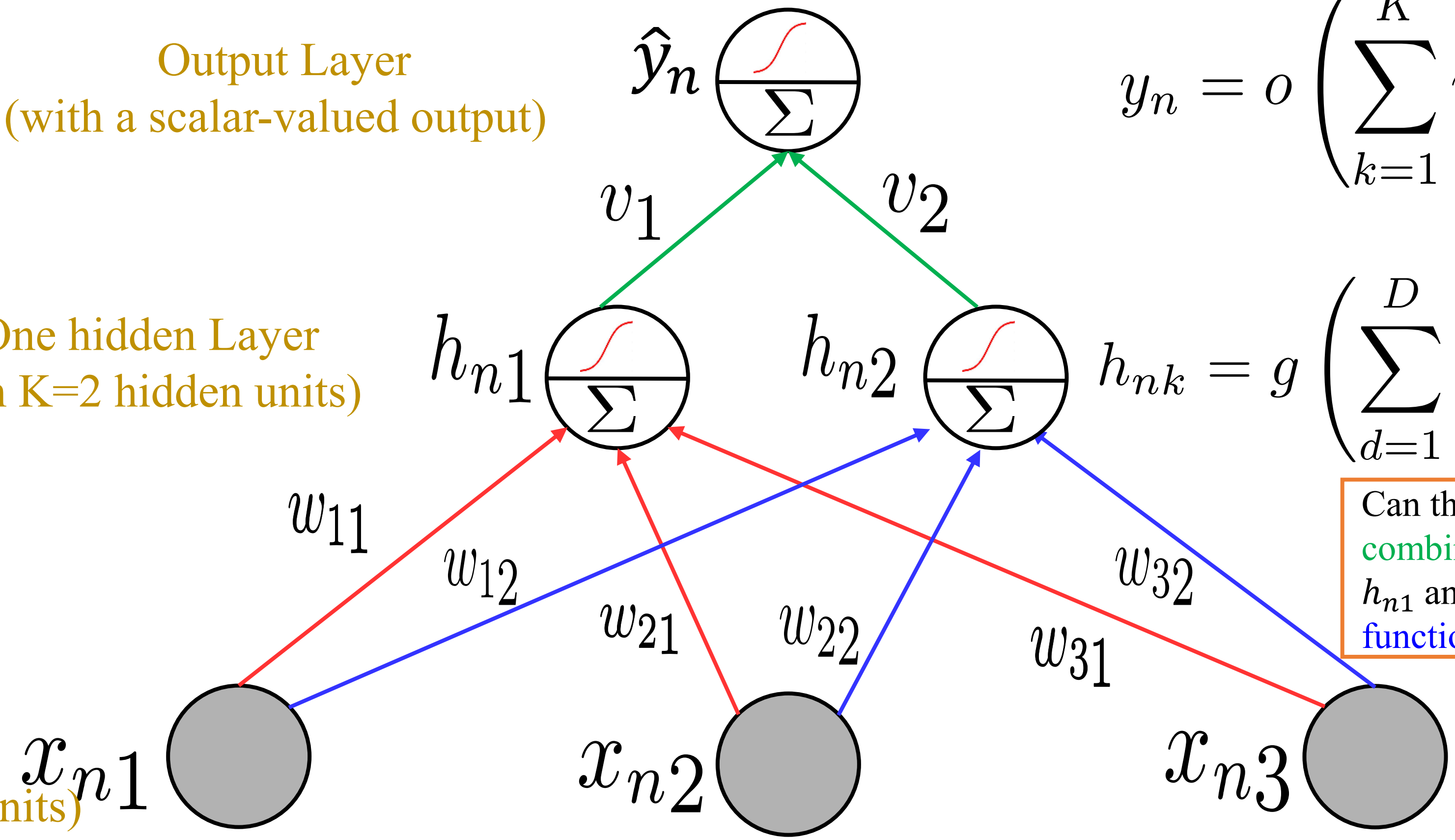
Exercise 4

3

Neural Networks: Multi-layer Perceptron (MLP)

- An MLP consists of an input layer, an output layer, and one or more hidden layers

Hidden layer units/nodes (a.k.a. “neurons”) act as new features



$$y_n = o\left(\sum_{k=1}^{K} v_k h_{nk}\right)$$

Learnable weights

$$h_{nk} = g\left(\sum_{d=1}^{D} w_{dk} x_{nd}\right)$$

Can think of this model as a combination of two predictions h_{n1} and h_{n2} of two simple functions (red and blue weights)

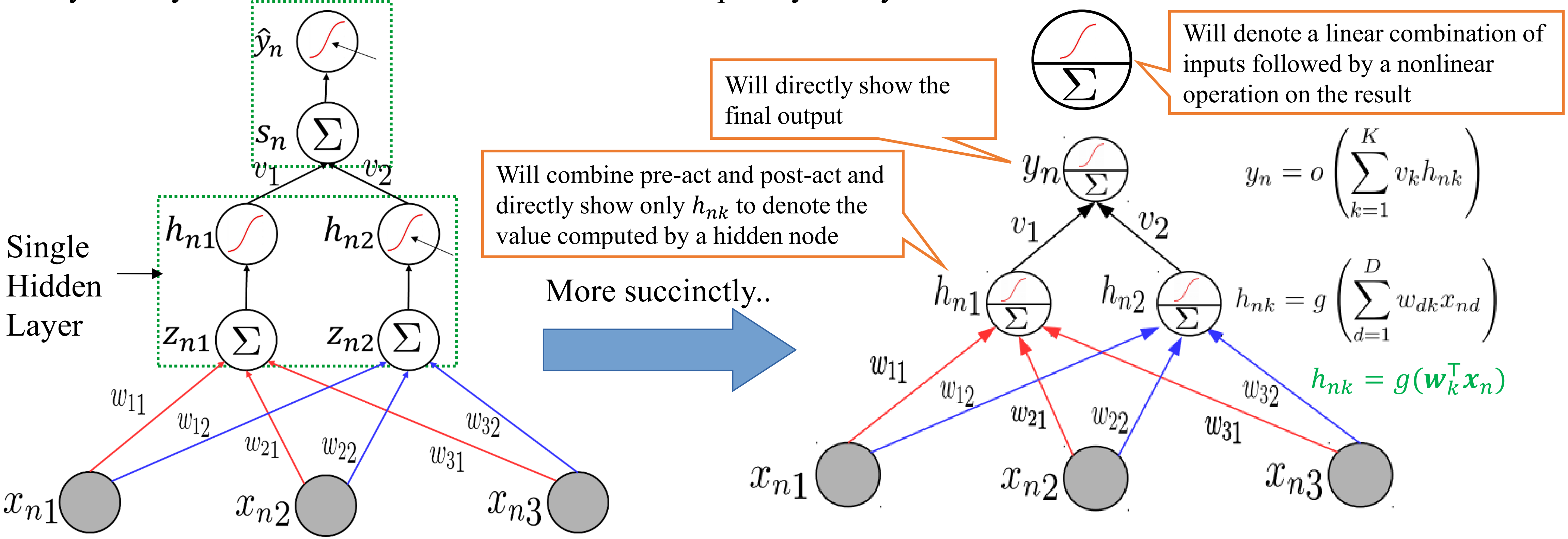
The effective x to y mapping is nonlinear (will see justification shortly)

Input layer units/nodes denote the original features of input $\boldsymbol{x}_n$

MLP is also called feedforward fully-connected network

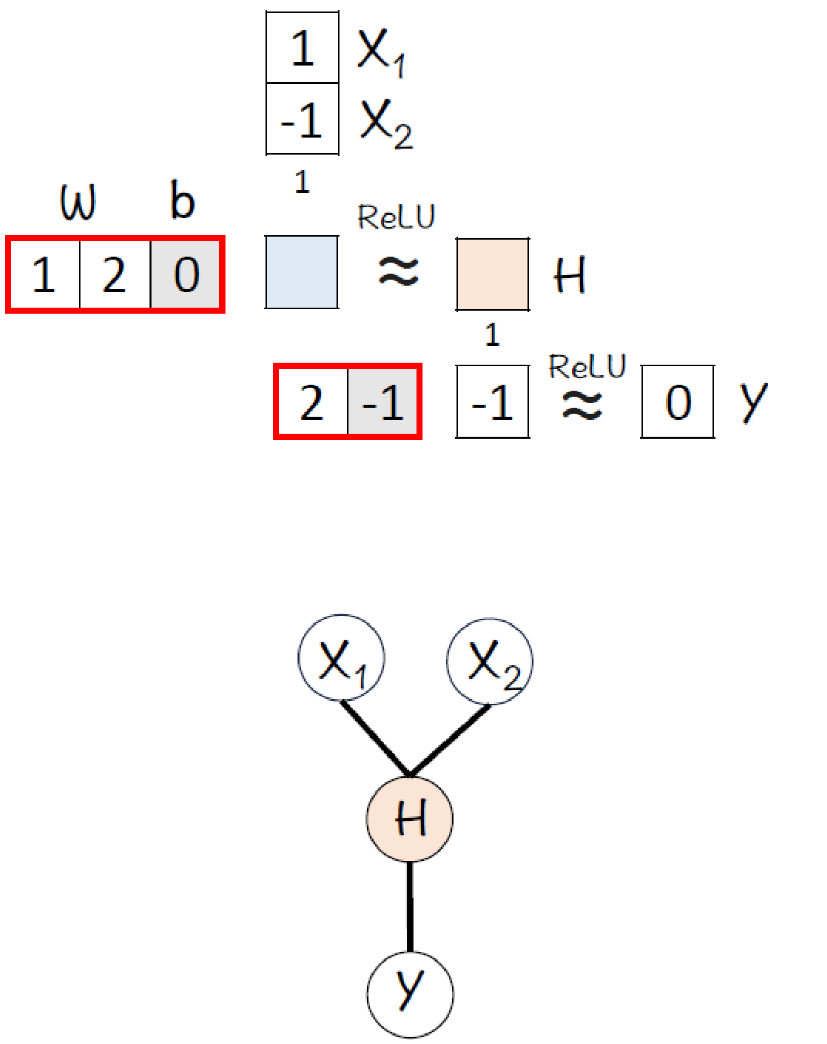
Neural Nets: A Compact Illustration

- Note: Hidden layer pre-act z_{nk} and post-act h_{nk} will be shown together for brevity
- Different layers may use different non-linear activations. Output layer may have none.



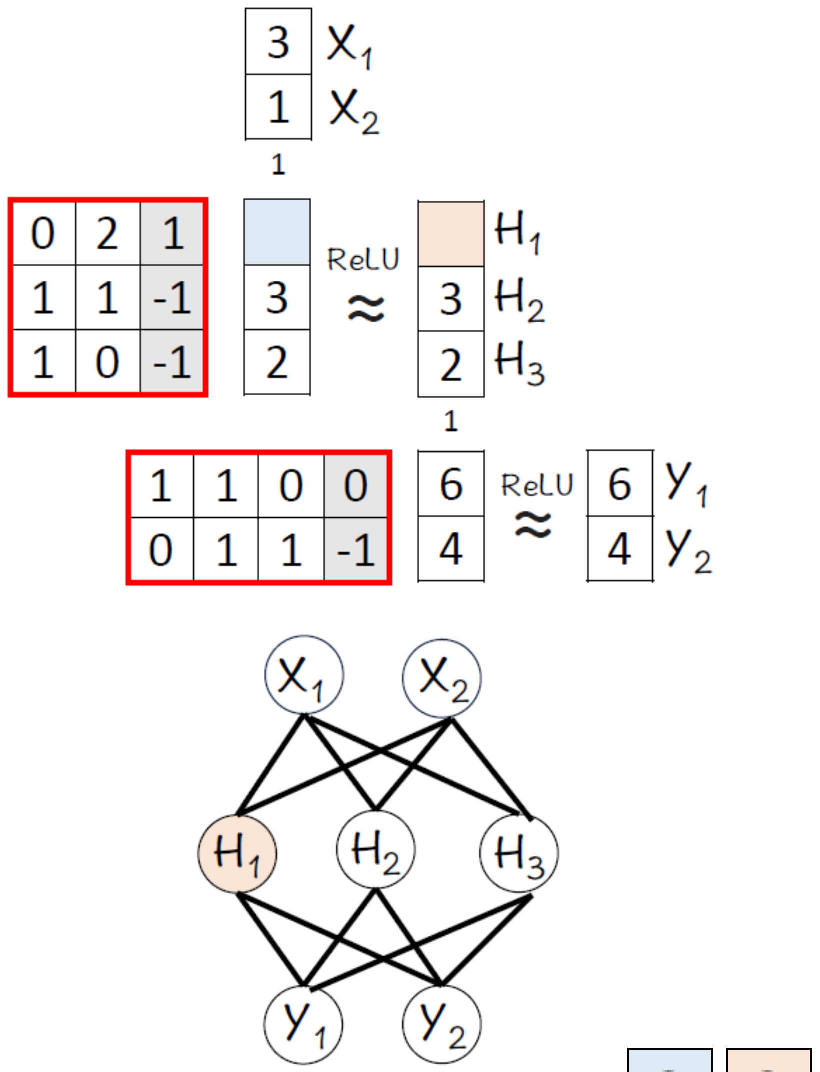
- Denoting $\boldsymbol{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_K]$, $\boldsymbol{w}_k \in \mathbb{R}^D$, $\boldsymbol{h}_n = g(\boldsymbol{W}^\top \boldsymbol{x}_n) \in \mathbb{R}^K$ ($K = 2, D = 3$ above).
- Note: g applied elementwise on pre-activation vector $z_n = W^\top x_n$

Exercise: Hidden Layer



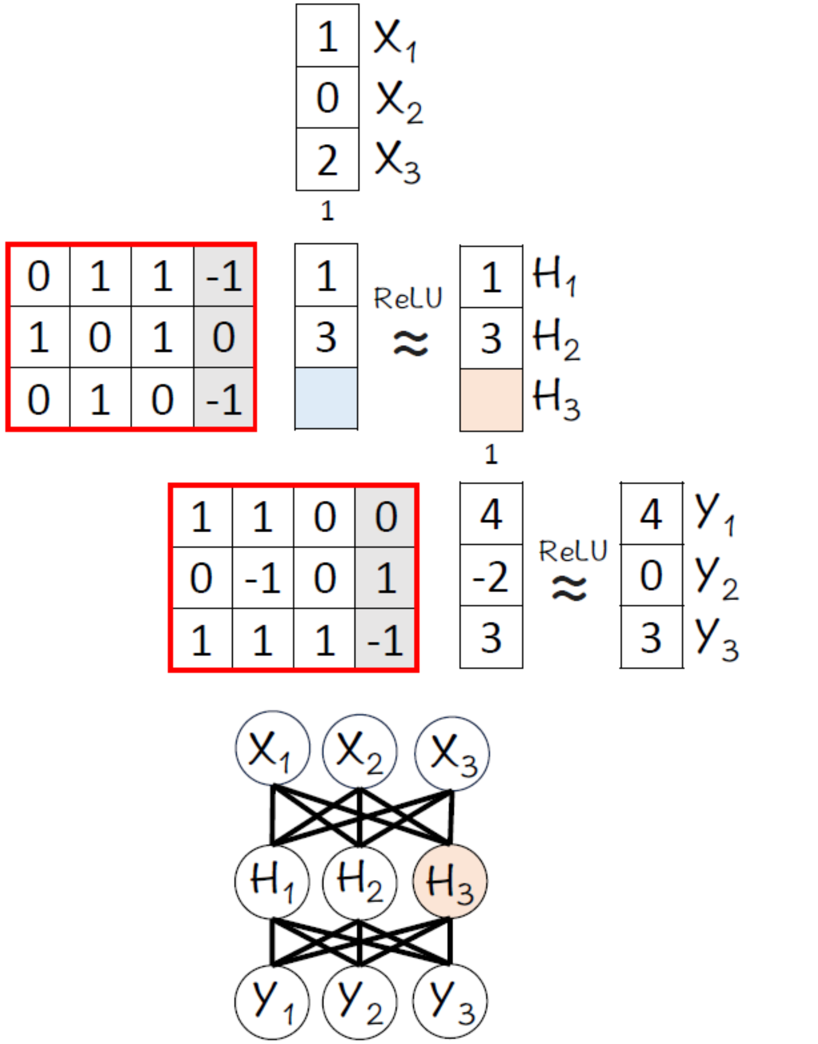
Exercise 1

-1 0



Exercise 2

3 3

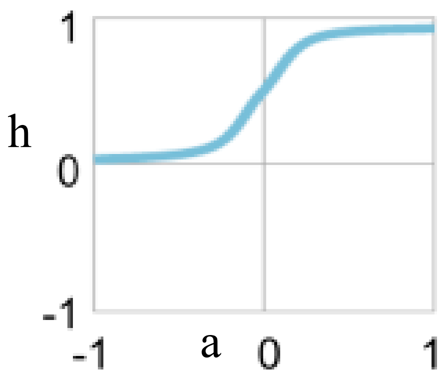


Exercise 3

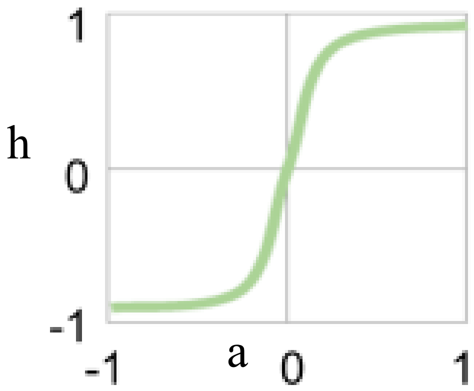
-1 0

Activation Functions: Some Common Choices

sigmoid



tanh



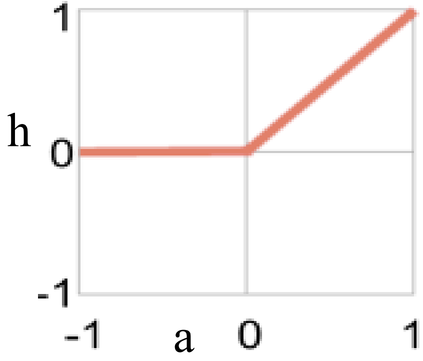
Preferred more than sigmoid. Helps keep the mean of the next layer's inputs close to zero (with sigmoid, it is close to 0.5)

For sigmoid as well as tanh, gradients saturate (become close to zero as the function tends to its extreme values)

Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$

tanh (tan hyperbolic): $h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$

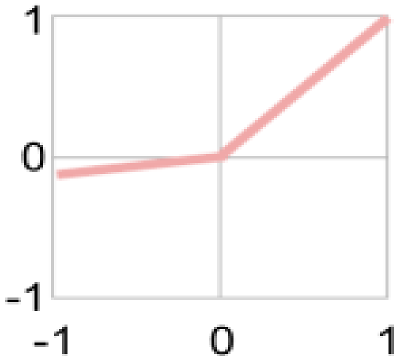
ReLU



ReLU and Leaky ReLU are among the most popular ones (also efficient to compute)

Helps fix the dead neuron problem of ReLU when a is a negative number

Leaky ReLU



Without nonlinear activation, a deep neural network is equivalent to a linear model no matter how many layers we use

ReLU (Rectified Linear Unit): $h = \max(0, a)$

Leaky ReLU: $h = \max(\beta a, a)$

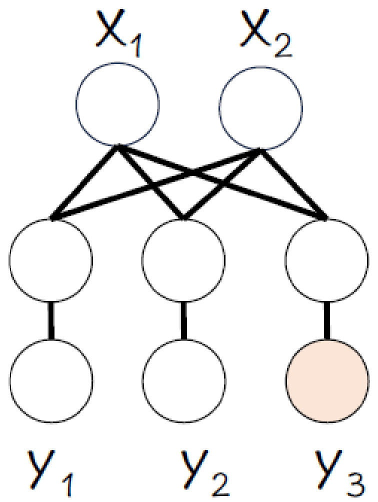
where β is a small postive number

Exercises: Activation Functions-ReLU, Sigmoid, Tanh

$$\begin{array}{c} \\ \\ \end{array}\begin{bmatrix}3\\2\end{bmatrix}\begin{array}{l}x_1\\x_2\end{array}$$

$$\begin{array}{ccc} w & & b \\ \end{array} \qquad 1$$

$$\begin{bmatrix}1 & -1 & 0\\ 1 & -2 & 0\\ 1 & -3 & 0\end{bmatrix} \quad \begin{bmatrix}1\\-1\\-3\end{bmatrix} \quad \underset{\approx}{\text{ReLU}} \quad \begin{bmatrix}1\\0\\ \ \end{bmatrix}\begin{array}{l}y_1\\y_2\\y_3\end{array}$$



Exercise 1

0

$$\begin{bmatrix}3\\2\end{bmatrix}\begin{array}{l}x_1\\x_2\end{array}$$

$$\begin{array}{ccc} w & & b \\ \end{array} \qquad 1$$

$$\begin{bmatrix}1 & -1 & 0\\ 1 & -2 & 0\\ 1 & -3 & 0\end{bmatrix} \quad \begin{bmatrix}1\\-1\\-3\end{bmatrix} \quad \underset{\approx}{\begin{array}{c}\text{Sigmoid}\\ \sigma\{3\}\end{array}} \quad \begin{bmatrix}0.5\\ \ \\0\end{bmatrix}\begin{array}{l}y_1\\y_2\\y_3\end{array}$$

3 Level Quantization
for hand calculation

Integer	Sigmoid
>= 2	1
-1, 0, 1	0.5
<= -2	0

Exercise 2

0.5

$$\begin{bmatrix}3\\2\end{bmatrix}\begin{array}{l}x_1\\x_2\end{array}$$

$$\begin{array}{ccc} w & & b \\ \end{array} \qquad 1$$

$$\begin{bmatrix}1 & -1 & -2\\ 1 & -2 & -2\\ 1 & -3 & -2\end{bmatrix} \quad \begin{bmatrix}-1\\-3\\-5\end{bmatrix} \quad \underset{\approx}{\text{Tanh}\{3\}} \quad \begin{bmatrix}0\\ \ \\-1\end{bmatrix}\begin{array}{l}y_1\\y_2\\y_3\end{array}$$

3 Level Quantization
for hand calculation

Integer	Sigmoid	Tanh
>= 2	1	1
-1, 0, 1	0.5	0
<= -2	0	-1

Exercise 3

-1

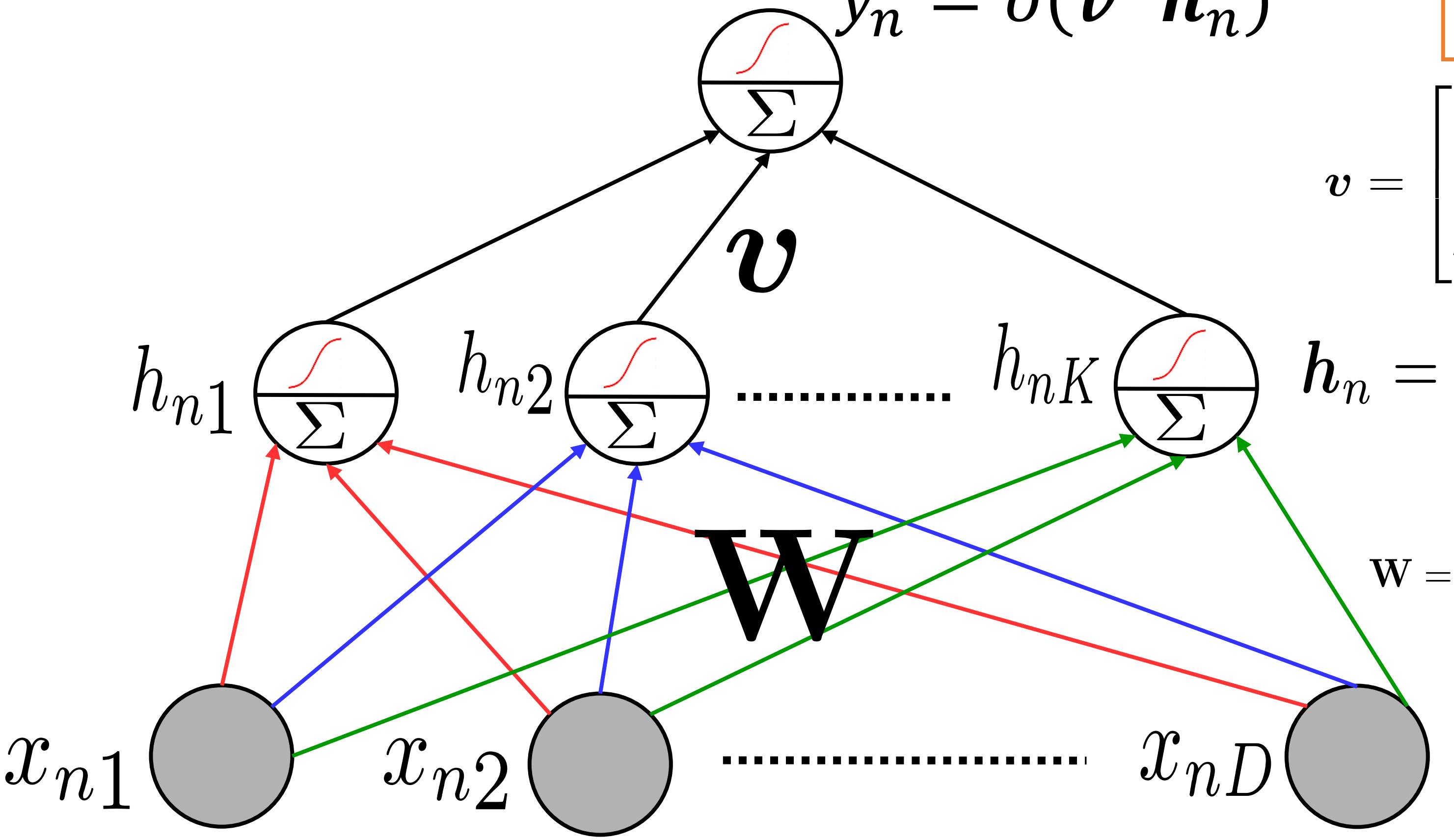
Examples of some basic NN/MLP architectures

Single Hidden Layer and Single Output

- One hidden layer with K nodes and a single output (e.g., scalar-valued regression or binary classification)

$$\hat{y}_n = o(\boldsymbol{v}^\top \boldsymbol{h}_n)$$

o is a suitable function (e.g., identity, sign, or sigmoid, etc) to convert the score into the actual response



$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_K \end{bmatrix}$$

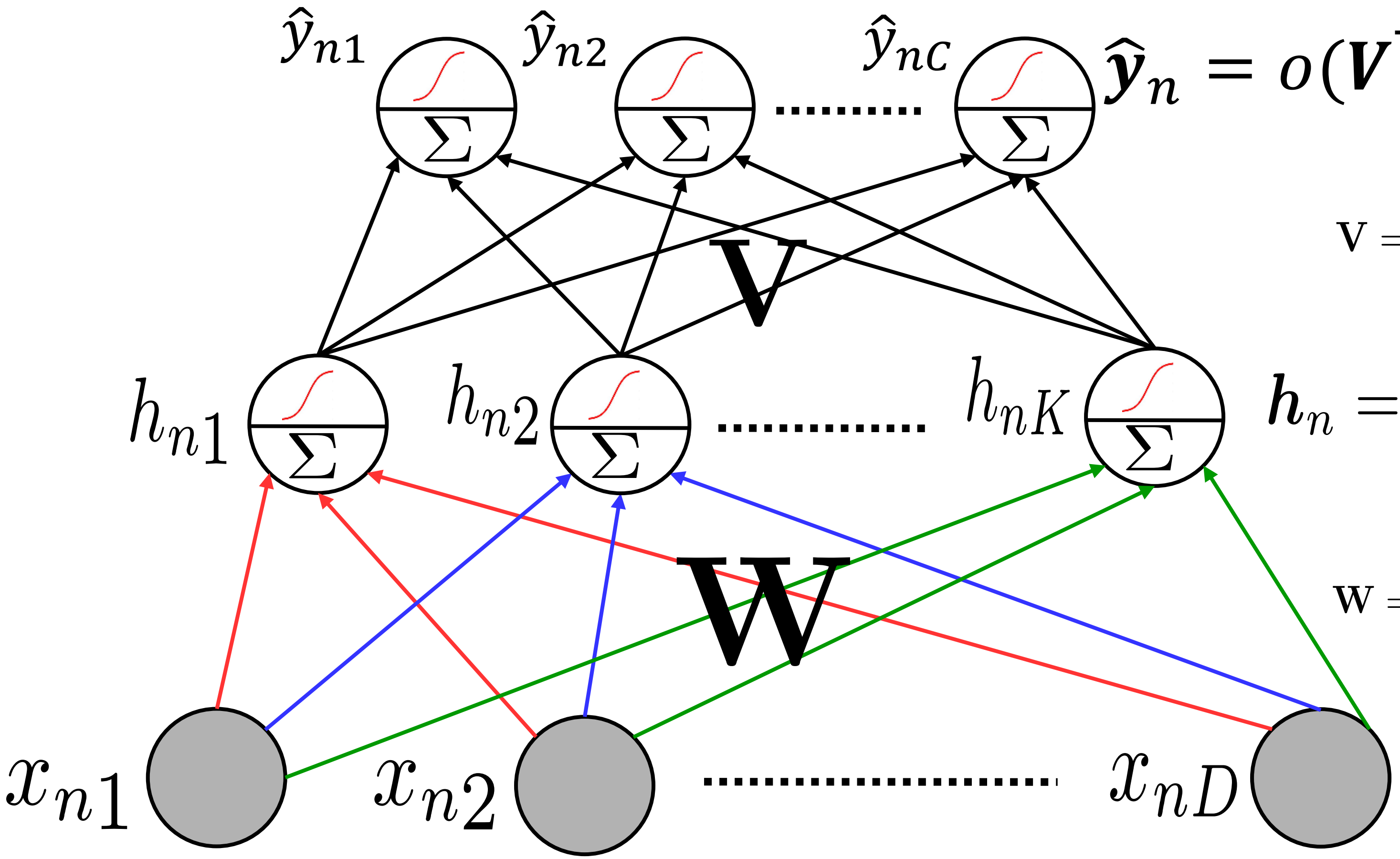
g is a nonlinear activation function

$$\boldsymbol{h}_n = g(\mathbf{W}^\top \boldsymbol{x}_n)$$

$$\mathbf{W} = \begin{bmatrix} w_{11} & \cdots & w_{1K} \\ \vdots & \ddots & \vdots \\ w_{D1} & \cdots & w_{DK} \end{bmatrix}$$

Single Hidden Layer and Multiple Outputs

- One hidden layer with K nodes and a vector of C outputs (e.g., vector-valued regression or multi-class classification or multi-label classification)



$$\hat{\boldsymbol{y}}_n = o(\boldsymbol{V}^\top \boldsymbol{h}_n)$$

o is a suitable function (e.g., identity, sign, or softmax, etc) to convert the score into the actual response vector for x_n

$$\mathbf{V} = \begin{bmatrix} v_{11} & \cdots & v_{1C} \\ \vdots & \ddots & \vdots \\ v_{K1} & \cdots & v_{KC} \end{bmatrix}$$

$$\boldsymbol{h}_n = g(\mathbf{W}^\top \boldsymbol{x}_n)$$

g is a nonlinear activation function

$$\mathbf{W} = \begin{bmatrix} w_{11} & \cdots & w_{1K} \\ \vdots & \ddots & \vdots \\ w_{D1} & \cdots & w_{DK} \end{bmatrix}$$

Multiple Hidden Layers (One/Multiple Outputs)

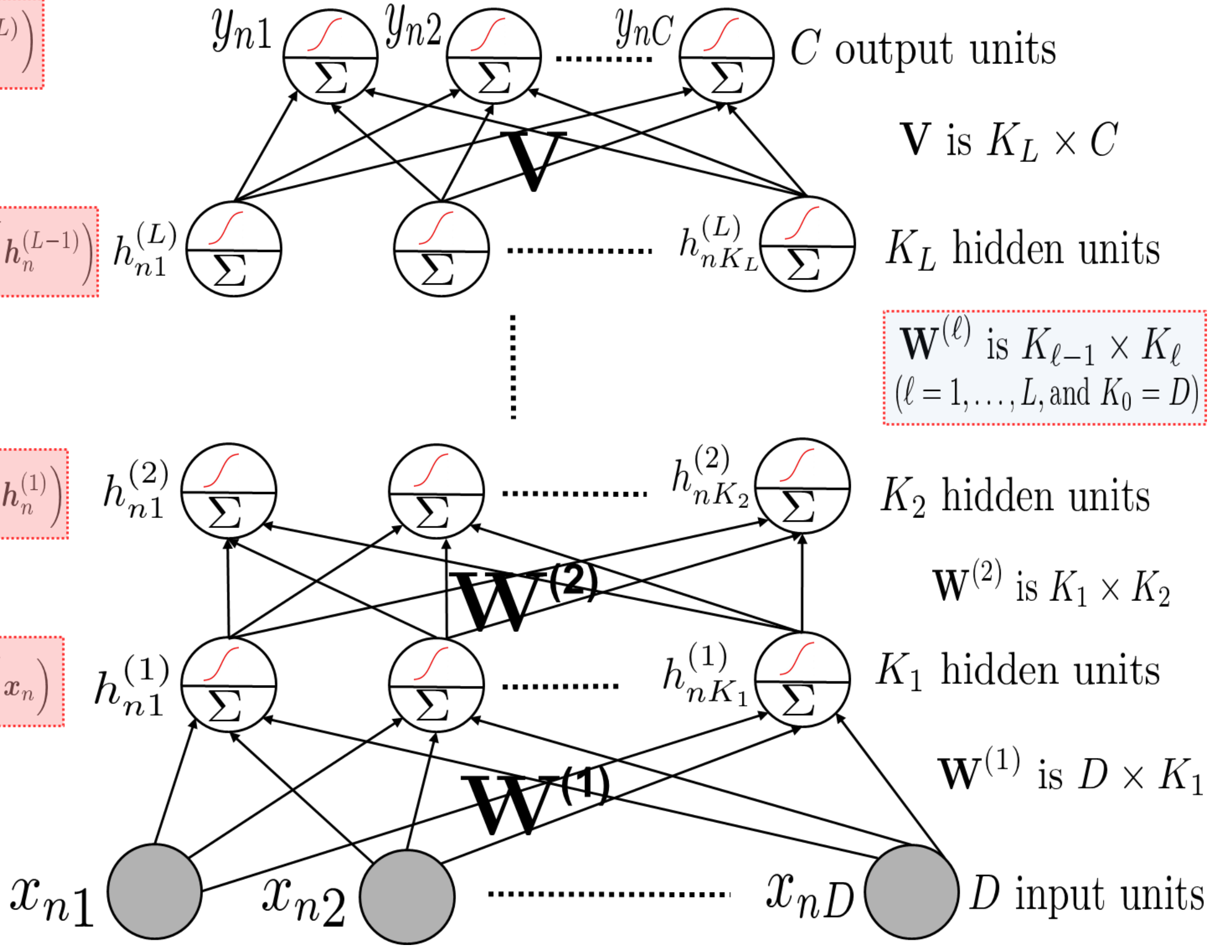
- Most general case: Multiple hidden layers with (with same or different number of hidden nodes in each) and a scalar or vector-valued output

$\boldsymbol{y}_n = o\left(\mathbf{V}^\top \boldsymbol{h}_n^{(L)}\right)$

$\boldsymbol{h}_n^{(L)} = g\left({\mathbf{W}^{(L)}}^\top \boldsymbol{h}_n^{(L-1)}\right)$

$\boldsymbol{h}_n^{(2)} = g\left({\mathbf{W}^{(2)}}^\top \boldsymbol{h}_n^{(1)}\right)$

$\boldsymbol{h}_n^{(1)} = g\left({\mathbf{W}^{(1)}}^\top \boldsymbol{x}_n\right)$



Each hidden layer uses a nonlinear activation function g (essential, otherwise the network can't learn nonlinear functions and reduces to a linear model)

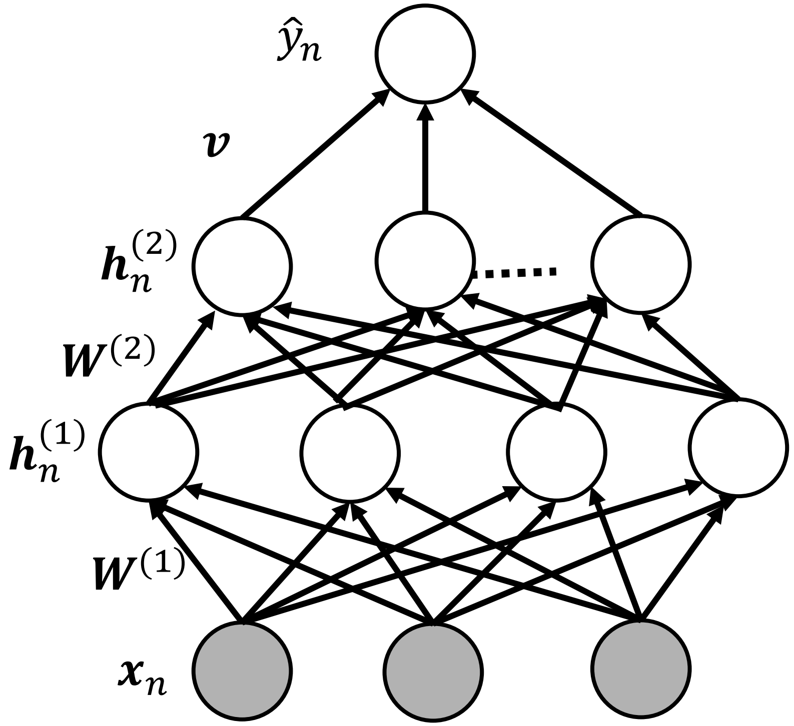
Note: Nonlinearity g is applied element-wise on its inputs so $h_n^{(\ell)}$ has the same size as vector $W^{(\ell)} h_n^{(\ell-1)}$

The Bias Term

- Each layer's pre-activations $\boldsymbol{z}_n^{(\ell)}$ have a an add bias term $\boldsymbol{b}^{(\ell)}$ (has the same size as $\boldsymbol{z}_n^{(\ell)}$ and $\boldsymbol{h}_n^{(\ell)}$) as well

$$\boldsymbol{z}_n^{(\ell)} = \boldsymbol{W}^{(\ell)\top} \boldsymbol{x}_n^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$$

$$\boldsymbol{h}_n^{(\ell)} = g(\boldsymbol{z}_n^{(\ell)})$$



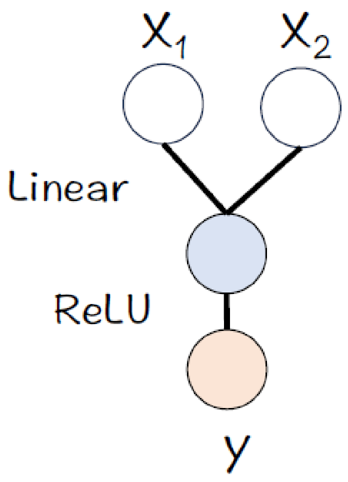
- Bias term increases the expressiveness of the network and ensures that we have nonzero activations/pre-activations even if this layer's input is a vector of all zeros

- Note that the bias term is the same for all inputs (does not depend on n)

- The bias term $\boldsymbol{b}^{(\ell)}$ is also learnable

Exercises: Artificial Neuron

$$\begin{array}{c} W \qquad\quad b \\ \left[\begin{array}{cc|c} 2 & 1 & -5 \end{array}\right] \end{array} \begin{array}{cl} \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} & \begin{array}{l} x_1 \\ x_2 \\ 1 \end{array} \end{array} = \boxed{} \overset{\text{ReLU}}{\approx} \boxed{}\ y$$

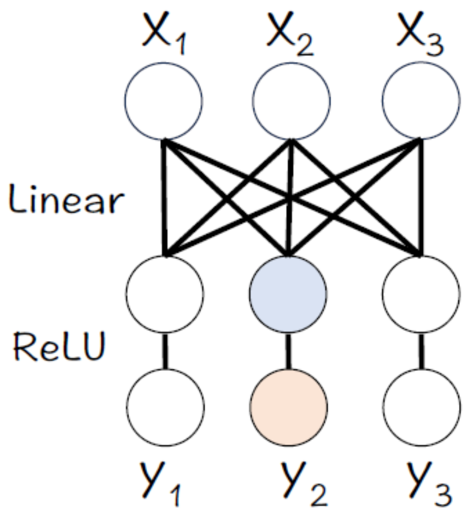


$\boxed{-2}$

$\boxed{0}$

Exercise 1

$$\begin{array}{c} W \qquad\qquad\quad b \\ \left[\begin{array}{ccc|c} 1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & -1 & 0 & 0 \end{array}\right] \end{array} \begin{array}{cl} \begin{bmatrix} 2 \\ 3 \\ 4 \\ 1 \end{bmatrix} & \begin{array}{l} x_1 \\ x_2 \\ x_3 \\ 1 \end{array} \end{array} = \begin{bmatrix} 5 \\ \\ -1 \end{bmatrix} \overset{\text{ReLU}}{\approx} \begin{array}{cl} \begin{bmatrix} 3 \\ \\ 0 \end{bmatrix} & \begin{array}{l} y_1 \\ y_2 \\ y_3 \end{array} \end{array}$$

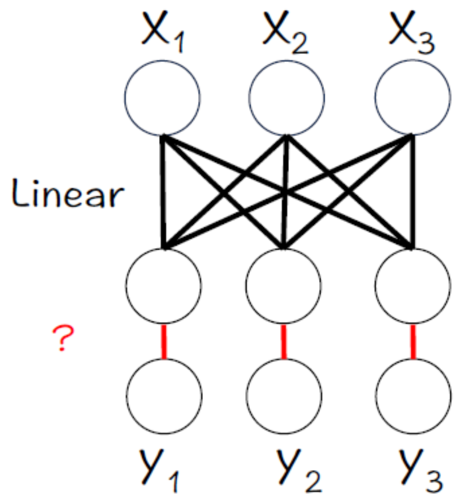


$\boxed{-2}$

$\boxed{0}$

Exercise 2

$$\begin{array}{c} W \qquad\qquad\quad b \\ \left[\begin{array}{ccc|c} 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{cl} \begin{bmatrix} 2 \\ 4 \\ 3 \\ 1 \end{bmatrix} & \begin{array}{l} x_1 \\ x_2 \\ x_3 \\ 1 \end{array} \end{array} = \begin{bmatrix} -2 \\ -1 \\ 2 \end{bmatrix} \overset{?}{\approx} \begin{array}{cl} \begin{bmatrix} 0 \\ 0.5 \\ 1 \end{bmatrix} & \begin{array}{l} y_1 \\ y_2 \\ y_3 \end{array} \end{array}$$

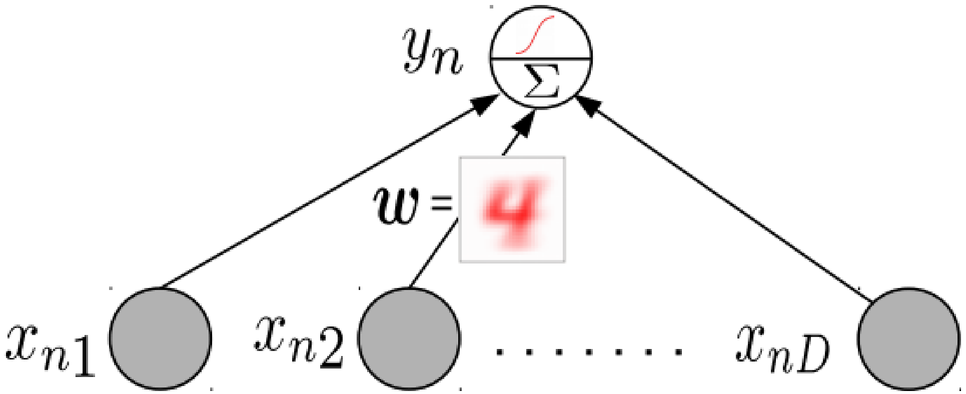


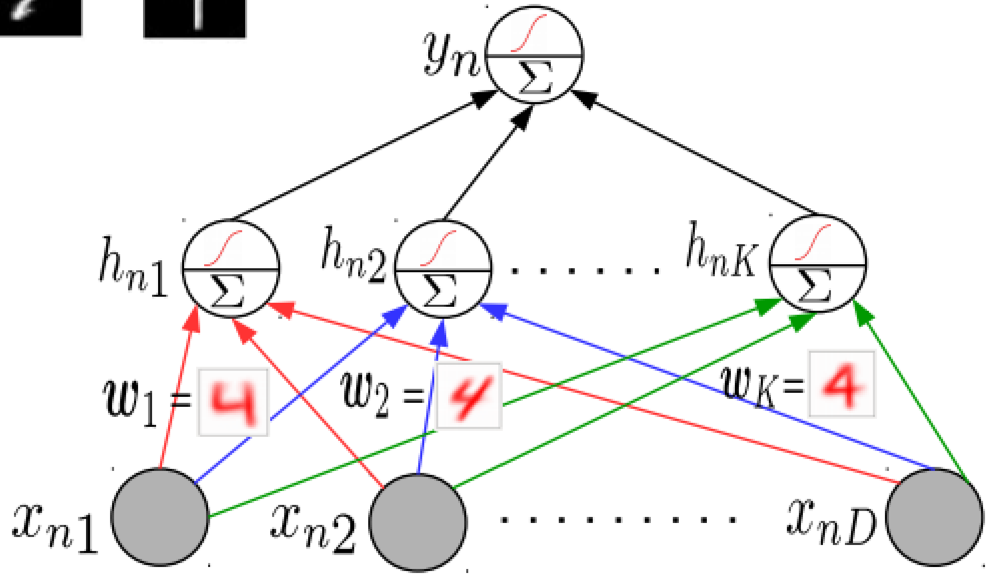
Exercise 3

$\boxed{\text{Sigmoid}}$

Why Neural Networks Work Better?

- Linear models tend to only learn the "average" pattern
  - E.g., Weight vector of a linear classification model represent average pattern of a class
- Deep models can learn multiple patterns (each hidden node can learn one pattern)
  - Thus, deep models can learn to capture more subtle variations that a simpler linear model





Backpropagation

- Backpropagation = Gradient descent using chain rule of derivatives
- Chain rule of derivatives: Example, if $y = f_1(x)$ and $x = f_2(z)$ then $\frac{\partial y}{\partial z} = \frac{\partial y}{\partial x}\frac{\partial x}{\partial z}$

Start taking the derivatives of the loss function w.r.t. params of the last layer and then proceed backwards

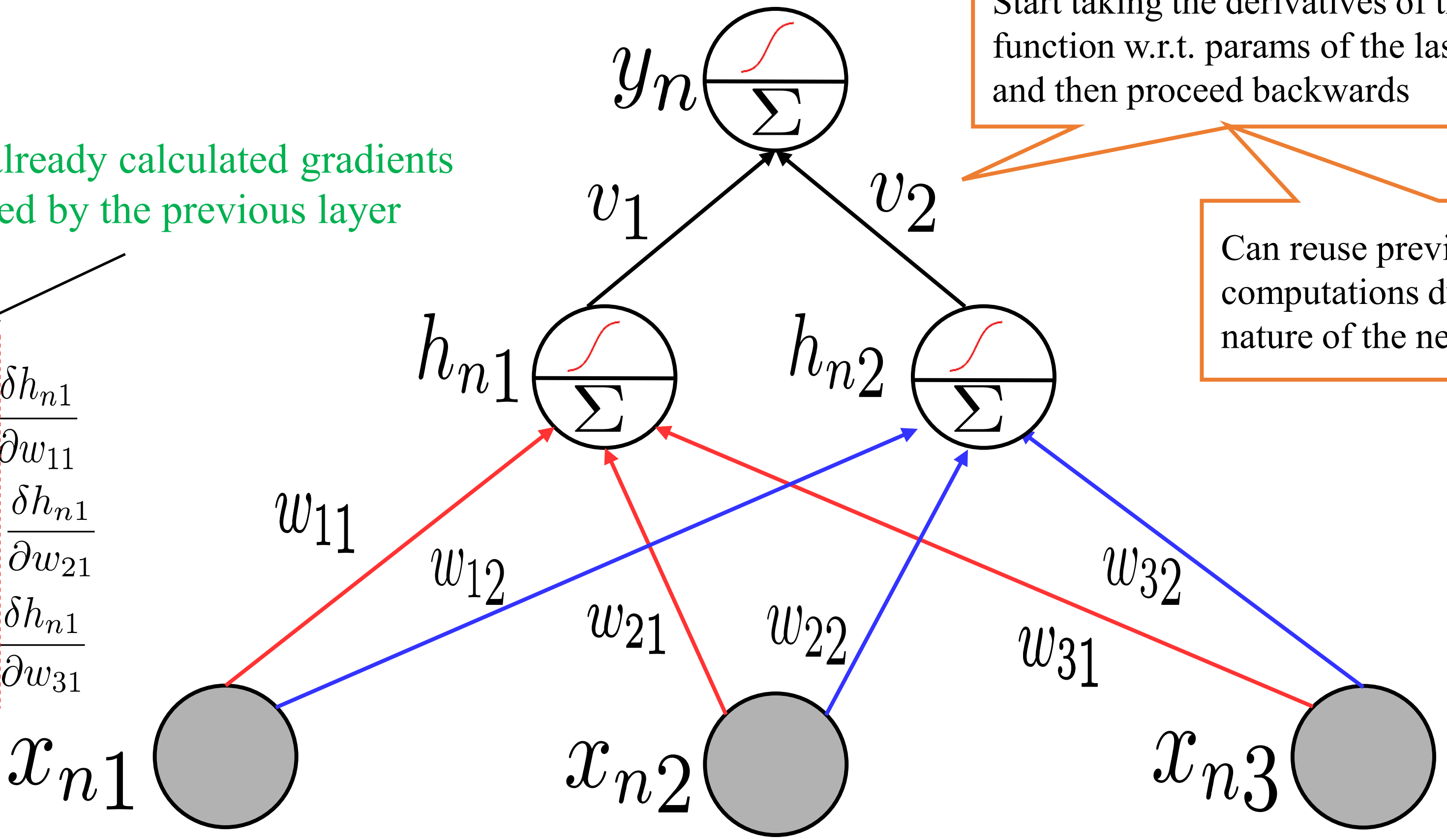
Can reuse previous derivative computations due to the recursive nature of the neural net architecture

Reuse already calculated gradients computed by the previous layer

$$\frac{\delta \mathcal{L}}{\partial w_{11}} = \boxed{\frac{\delta \mathcal{L}}{\partial h_{n1}}}\frac{\delta h_{n1}}{\partial w_{11}}$$

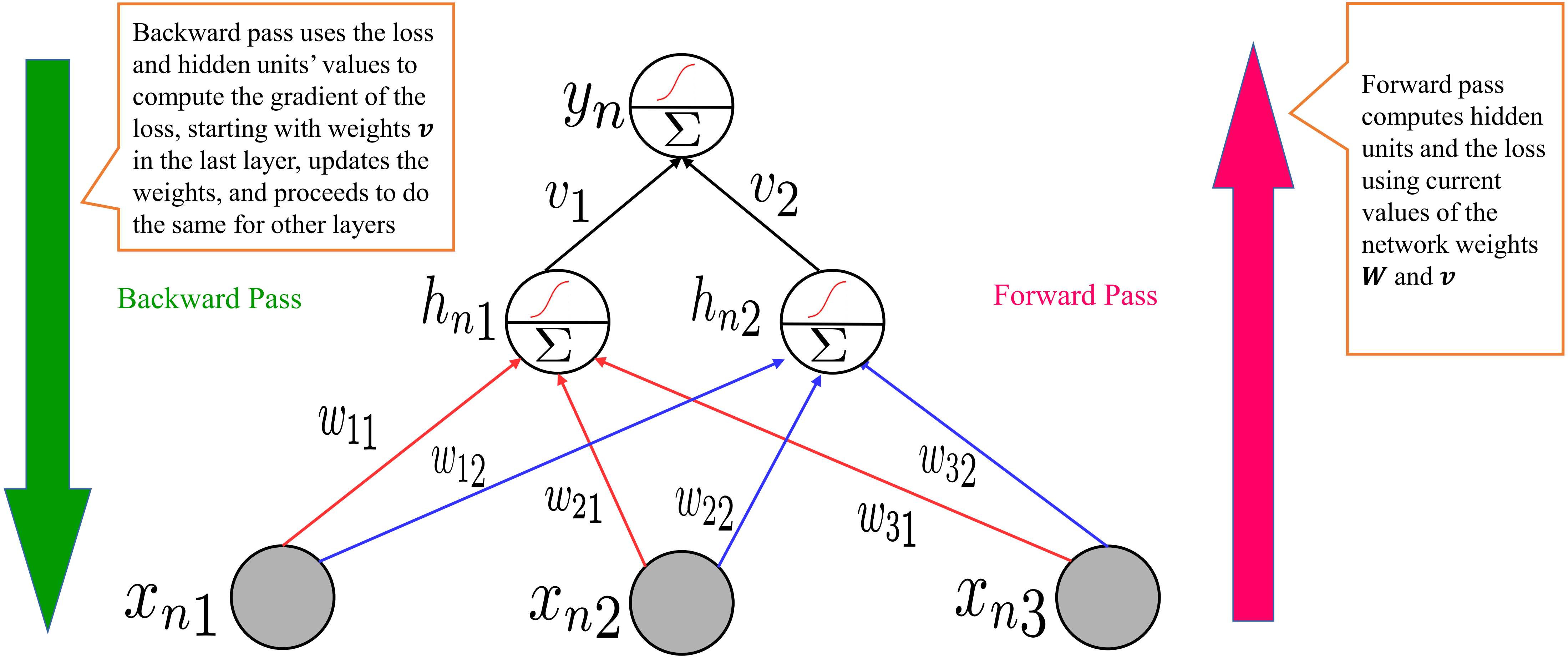
$$\frac{\delta \mathcal{L}}{\partial w_{21}} = \boxed{\frac{\delta \mathcal{L}}{\partial h_{n1}}}\frac{\delta h_{n1}}{\partial w_{21}}$$

$$\frac{\delta \mathcal{L}}{\partial w_{31}} = \boxed{\frac{\delta \mathcal{L}}{\partial h_{n1}}}\frac{\delta h_{n1}}{\partial w_{31}}$$



Backpropagation

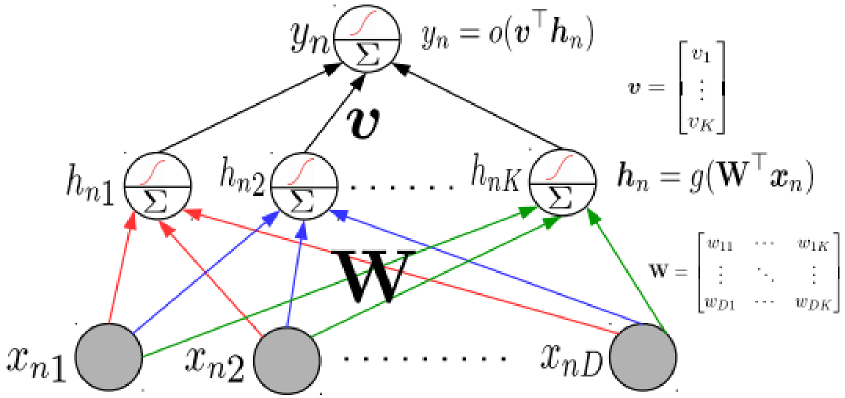
- Backprop iterates between a forward pass and a backward pass



- Software frameworks such as Tensorflow and PyTorch support this

Backpropagation through an example

Consider a single hidden layer MLP



Assuming regression (o = identity), the loss function for this model

$$
\begin{aligned}
\mathcal{L} &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2
\end{aligned}
$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

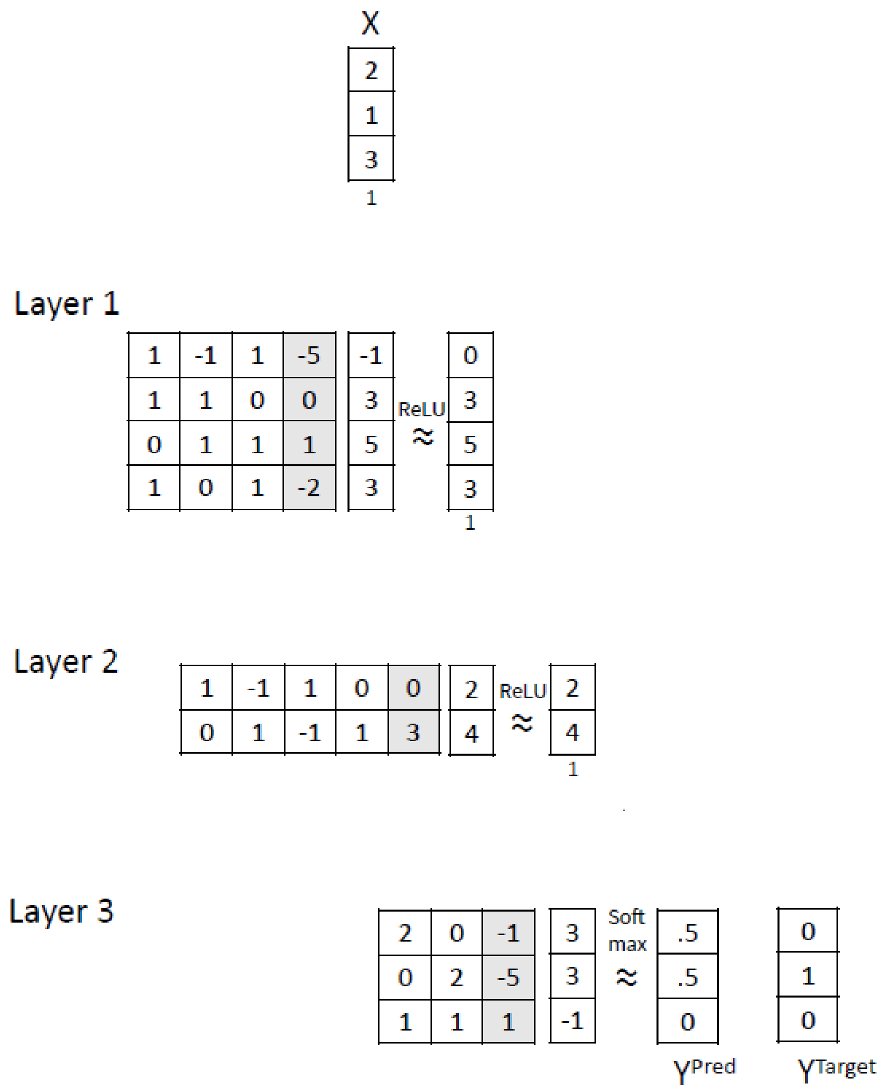
$$
\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}
$$

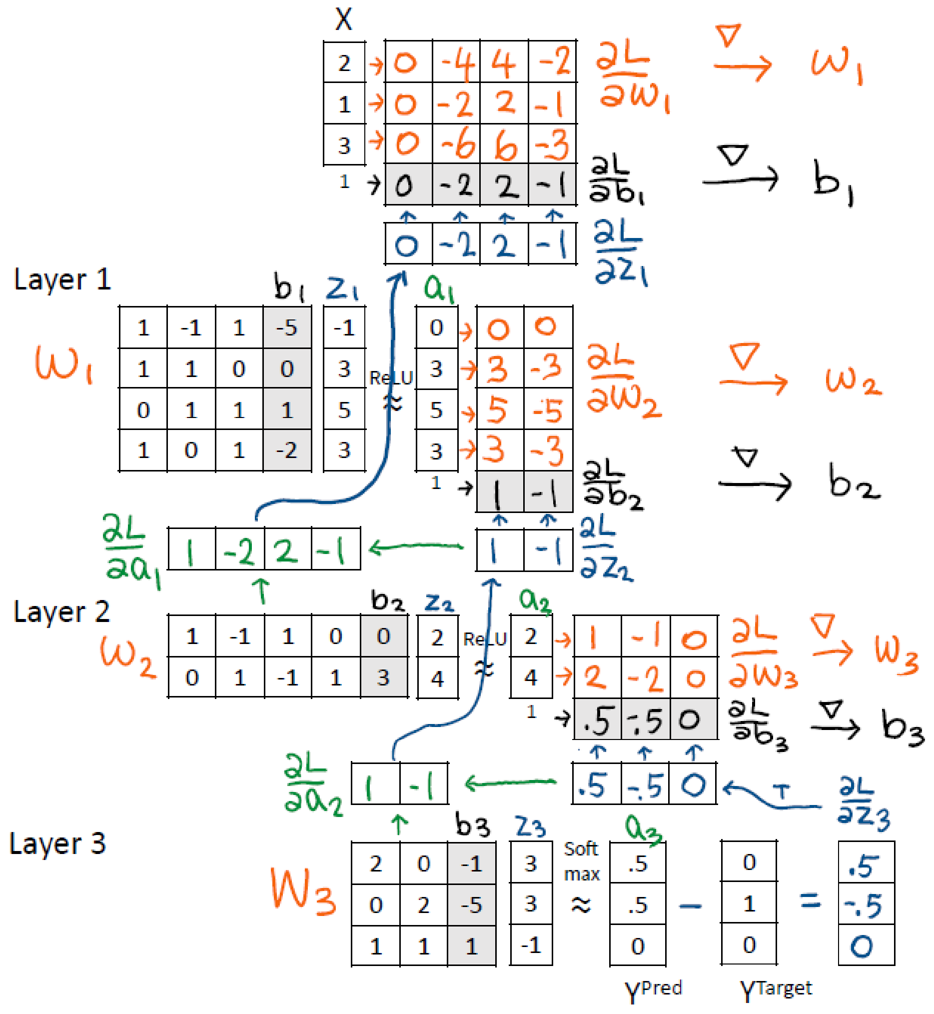
- Gradient of $\mathcal{L}$ w.r.t. $\mathbf{W}$ requires chain rule

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_{dk}} &= \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}} \\
\frac{\partial \mathcal{L}}{\partial h_{nk}} &= -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)) v_k = -\boldsymbol{e}_n v_k \\
\frac{\partial h_{nk}}{\partial w_{dk}} &= g'(\boldsymbol{w}_k^\top \boldsymbol{x}_n) x_{nd} \qquad (\text{note: } h_{nk} = g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))
\end{aligned}
$$

- Forward prop computes errors $\boldsymbol{e}_n$ using current $\mathbf{W}$, $\boldsymbol{v}$.
  Backprop updates NN params $\mathbf{W}$, $\boldsymbol{v}$ using grad methods
- Backprop caches many of the calculations for reuse

Exercise: Backpropagation





L: Cross-Entropy Loss

Exercise: Backpropagation

Steps:

1. Forward Pass: Given a multi layer perceptron (3 levels), an input vector X, predictions Y^{Pred} = [0.5,0.5,0], and ground truth label Y^{Target} =[0,1,0].

2. Backpropagation: Insert cells to hold our calculations.

3. Layer 3 - Softmax (blue): Calculate ∂L/∂z3 directly using the simple equation: $Y^{Pred} - Y^{Target}$ = [0.5,-0.5,0]. This simple equation is the benefit of using Softmax and Cross Entropy Loss together.

4. Layer 3 - Weights (orange) & Biases (black): Calculate ∂L/∂W3 and ∂L/∂b3 by multiplying ∂L/∂z3 and [a2|1].

5. Layer 2 - Activations (green): Calculate ∂L/∂a2 by multiplying ∂L/∂z3 and W3.

6. Layer 2 - ReLU (blue): Calculate ∂L/∂z2 by multiplying ∂L/∂a2 with 1 for positive values and 0 otherwise.

7. Layer 2 - Weights (orange) & Biases (black): Calculate ∂L/∂W2 and ∂L/∂b2 by multiplying ∂L/∂z2 and [a1|1].

8. Layer 1 - Activations (green): Calculate ∂L/∂a1 by multiplying ∂L/∂z2 and W2.

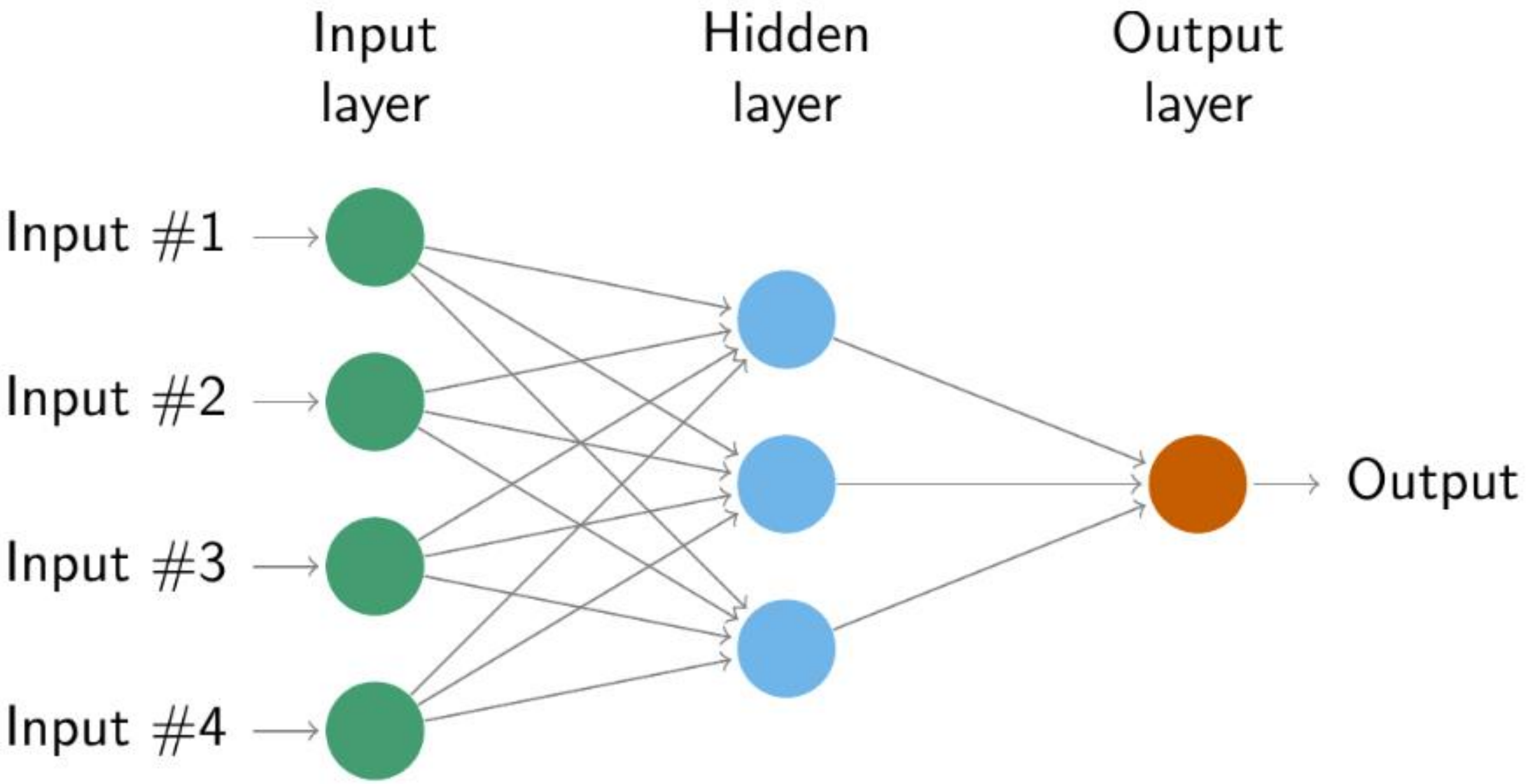
9. Layer 1 - ReLU (blue): Calculate ∂L/∂z1 by multiplying ∂L/∂a1 with 1 for positive values and 0 otherwise.

10. Layer 1 - Weights (orange) & Biases (black): Calculate ∂L/∂W1 and ∂L/∂b1 by multiplying ∂L/∂z1 and [x|1].

11. Gradient Descent: Update weights and biases (typically a learning rate is applied here).

Autoregressive Neural Network

Feed-forward Neural Network



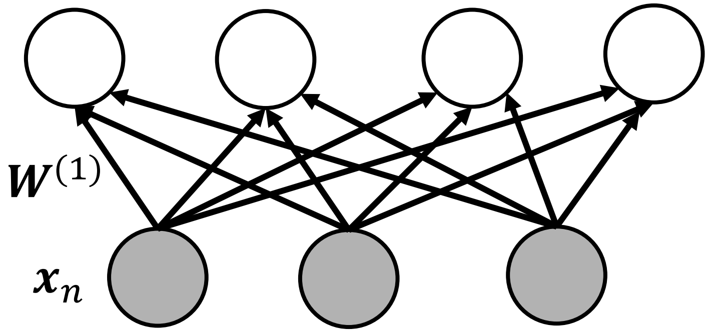
ARNN for Forecasting

- With time series data, lagged values of the time series can be used as inputs to a neural network, typically named as Autoregressive neural network (ARNN)
- $\text{ARNN}(p, \text{k})$ is a feed-forward network with one hidden layer, with p lagged inputs and k nodes in the hidden layer.
- $\text{ARNN}(p, 0)$ model is equivalent to an $\text{ARIMA}(p, 0,0)$ model, but without the restrictions on the parameters to ensure stationarity.
- With seasonal data, it is useful to also add the last observed values from the same season as inputs.
- For time series, the default is the optimal number of lags (according to the AIC) for a linear $\text{AR}(p)$ model. If k is not specified, it is set to $k = (p + 1)/2$ (rounded to the nearest integer).
- When it comes to forecasting, the network is applied iteratively.

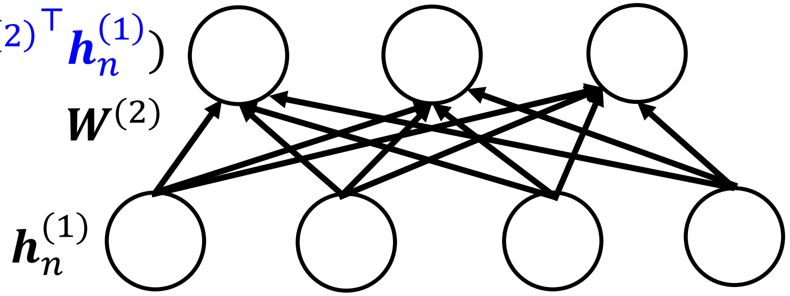
Limitations/Shortcomings of MLP & ARNN

- MLP uses fully connected layers defined by matrix multiplications + nonlinearity

$$h_n^{(1)} = g({W^{(1)}}^\top x_n)$$



$$h_n^{(2)} = g({W^{(2)}}^\top h_n^{(1)})$$

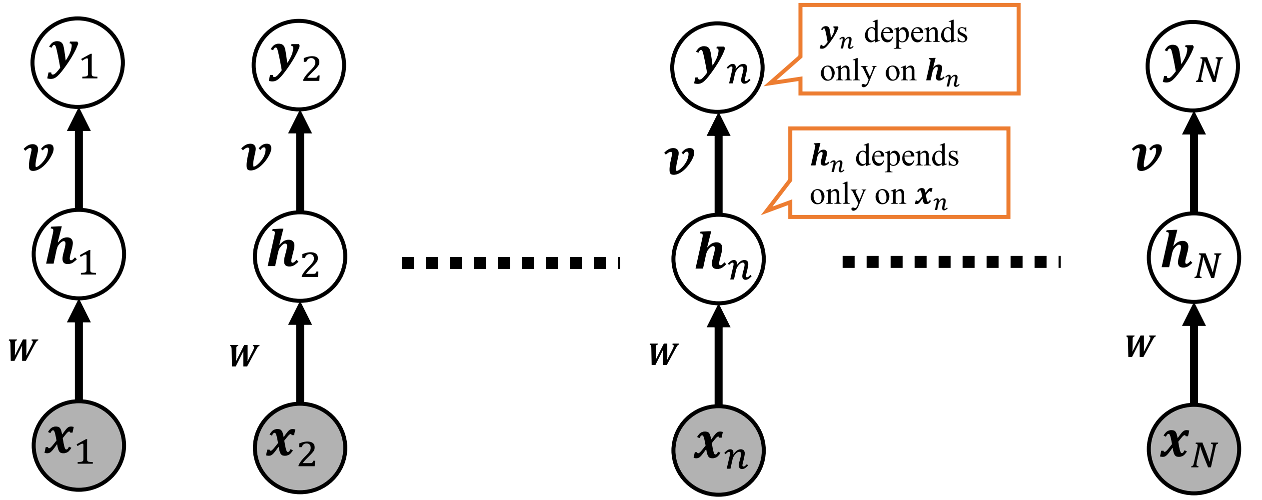


- MLP ignores structure (e.g., spatial/sequential) in the inputs
  - Not ideal for data such as images, text, etc. which are flattened as vectors when used with MLP

- Fully connected nature of MLP requires massive number of weights
  - Recall that each layer is fully connected so each layer needs a massive number of weights!

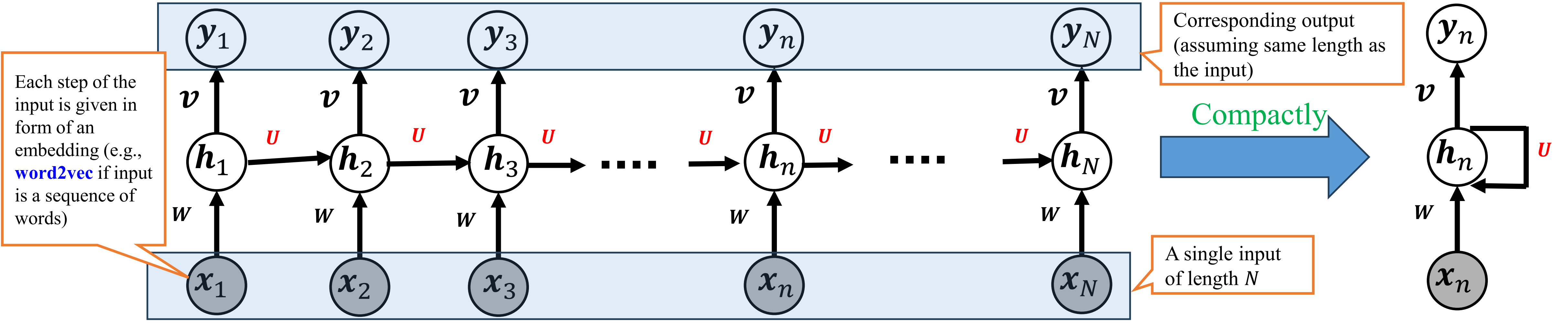
Recurrent Connections in Deep Neural Networks

- Feedforward nets such as MLP assume independent observations



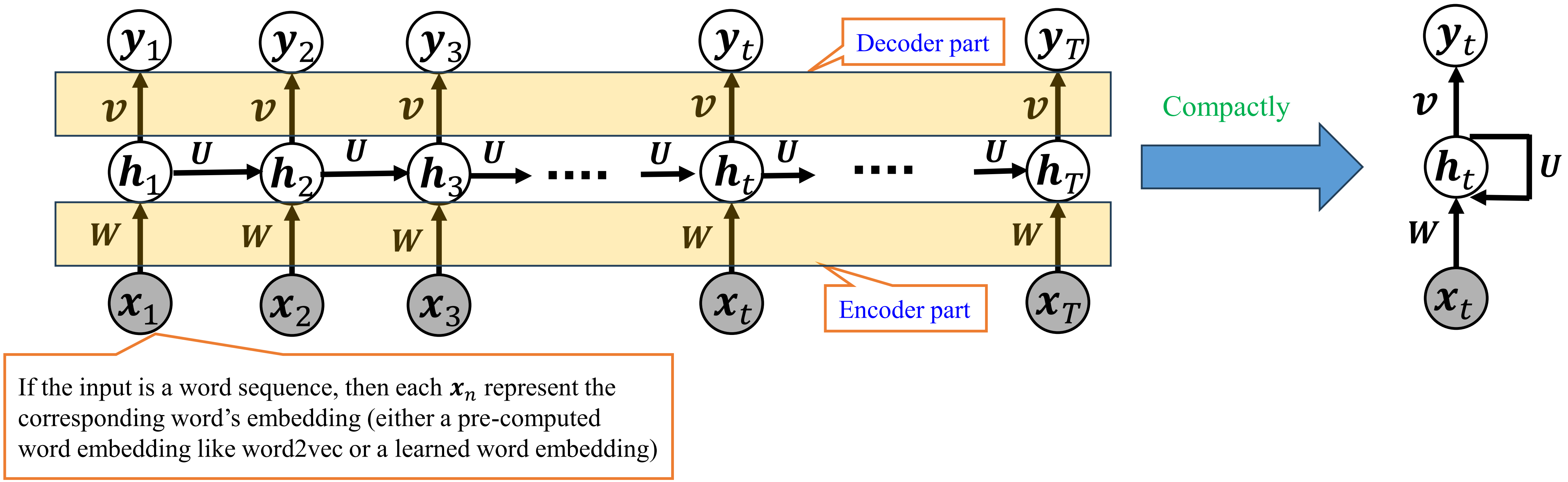
Feedforward neural networks are not ideal when inputs $[x_1, x_2, \ldots, x_N]$ and/or outputs $[y_1, y_2, \ldots, y_N]$ represent sequential data (e.g., sequence of words, video (sequence of frames), etc.

- A recurrent structure can be helpful if each input and/or output is a sequence



RNNs

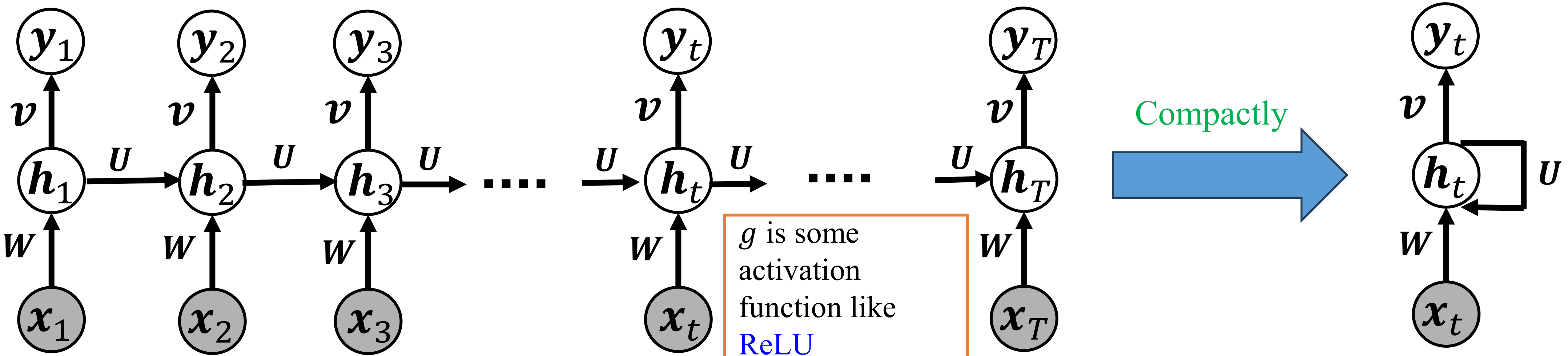
- RNNs are used when each input or output or both are sequences of tokens



- Hidden state $\boldsymbol{h}_t$ is supposed to remember everything up to time $t-1$. However, in practice, RNNs have difficulties remembering the distant past
  - Variants such as LSTM, GRU, etc mitigate this issue to some extent
- Slow processing is another major issue (e.g., can't compute $\boldsymbol{h}_t$ before computing $\boldsymbol{h}_{t-1}$)

Recurrent Neural Networks

- A basic RNN's architecture (assuming input and output sequence have same lengths)
- RNN has three sets of weights $\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{v}$



g is some activation function like ReLU

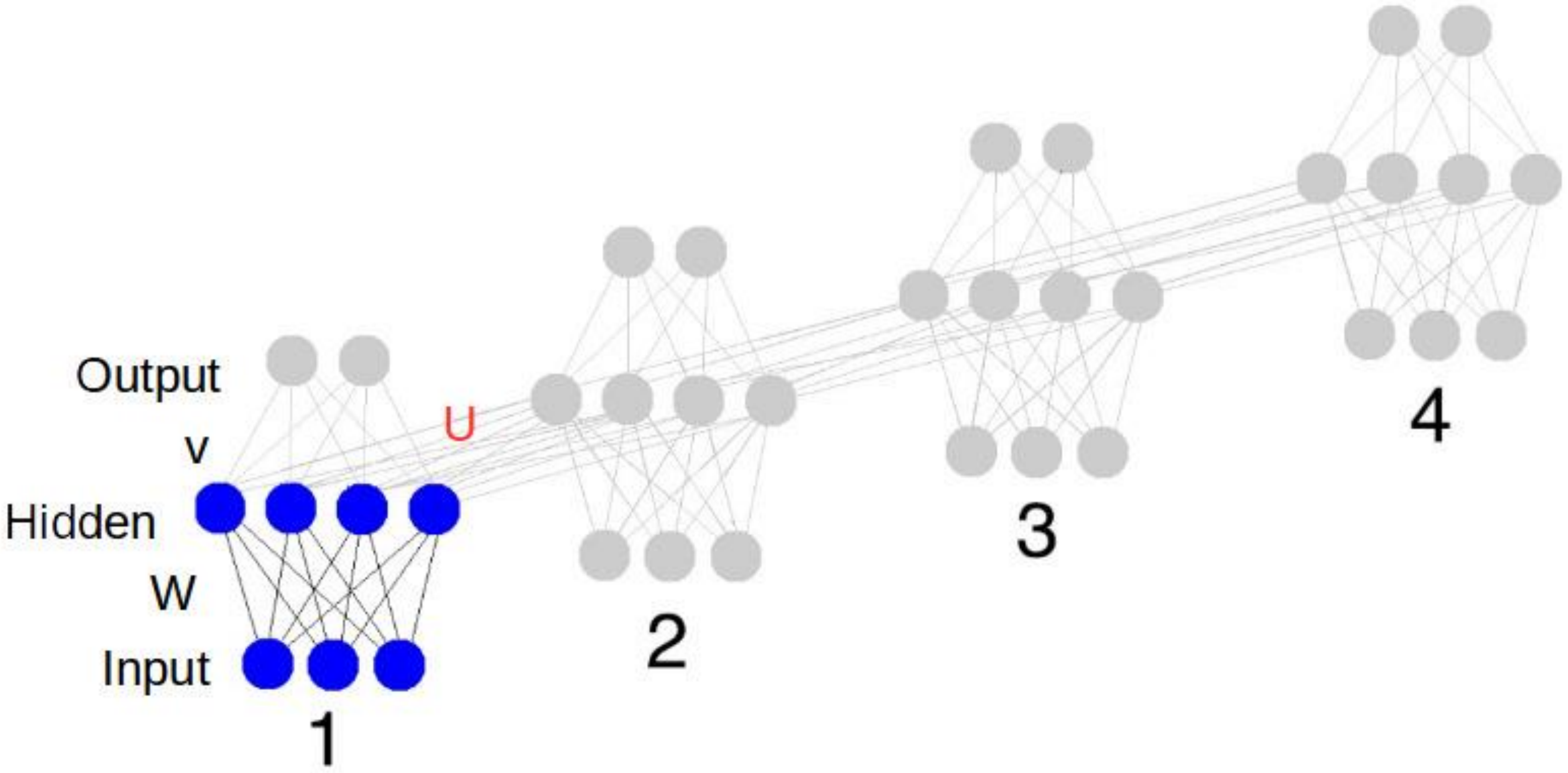
- $\boldsymbol{W}$ and $\boldsymbol{U}$ model how h_t at step t is computed: $\boldsymbol{h}_t = g(\boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{U}\boldsymbol{h}_{t-1})$
- $\boldsymbol{v}$ models the hidden layer to output mapping, e.g., $\boldsymbol{y}_t = o(\boldsymbol{v}\boldsymbol{h}_t)$
- Important: Same $\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{v}$ are used at all steps of the sequence (weight sharing)

o depends on the nature of y_t. If it is categorical then o can be softmax

Given in form of an embedding (e.g., word embedding if x_1 is a word)

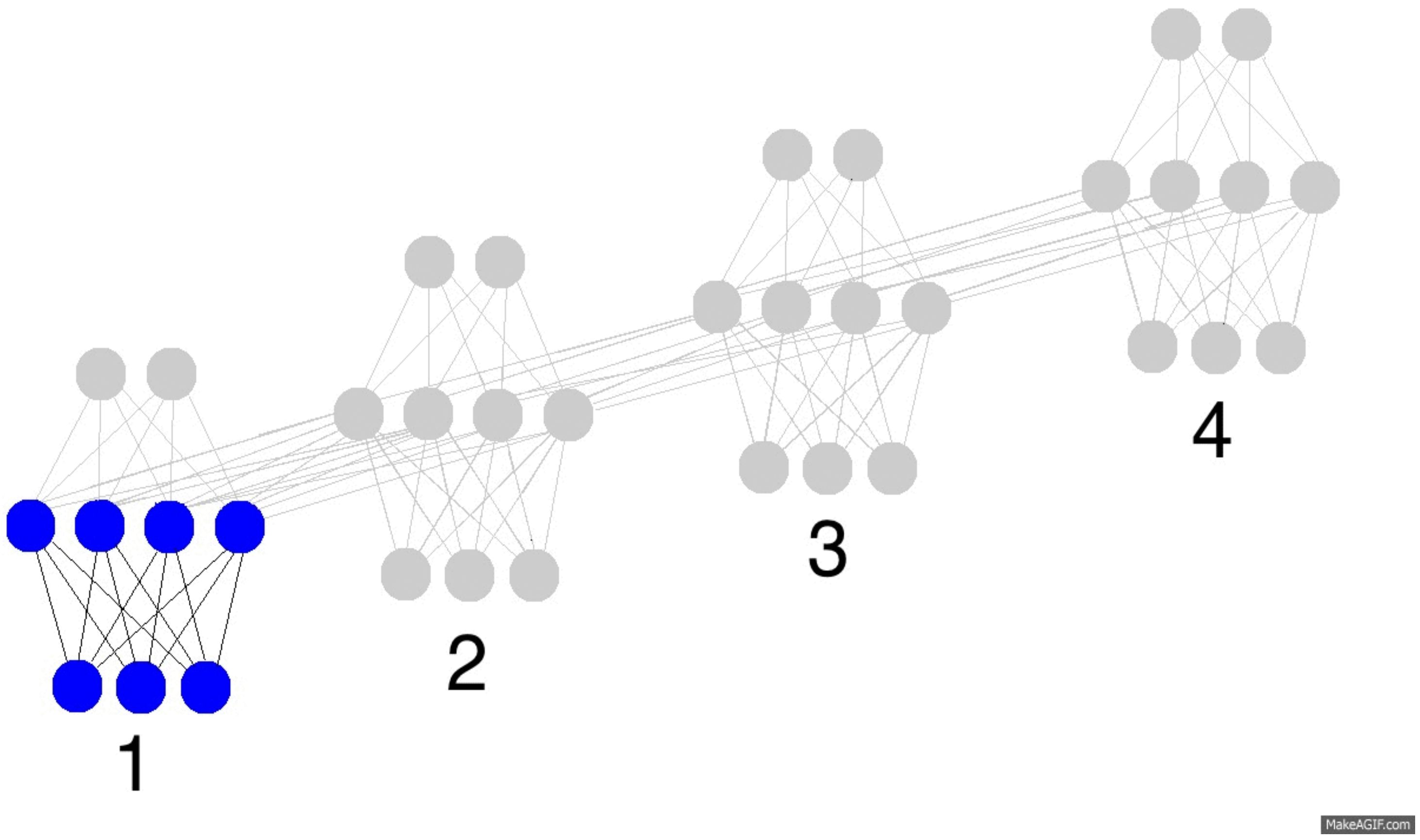
Recurrent Neural Nets (RNN)

- A more “micro” view of RNN (the transition matrix U connects the hidden states across observations, propagating information along the sequence)



Pic source: https://iamtrask.github.io/

RNN in Action



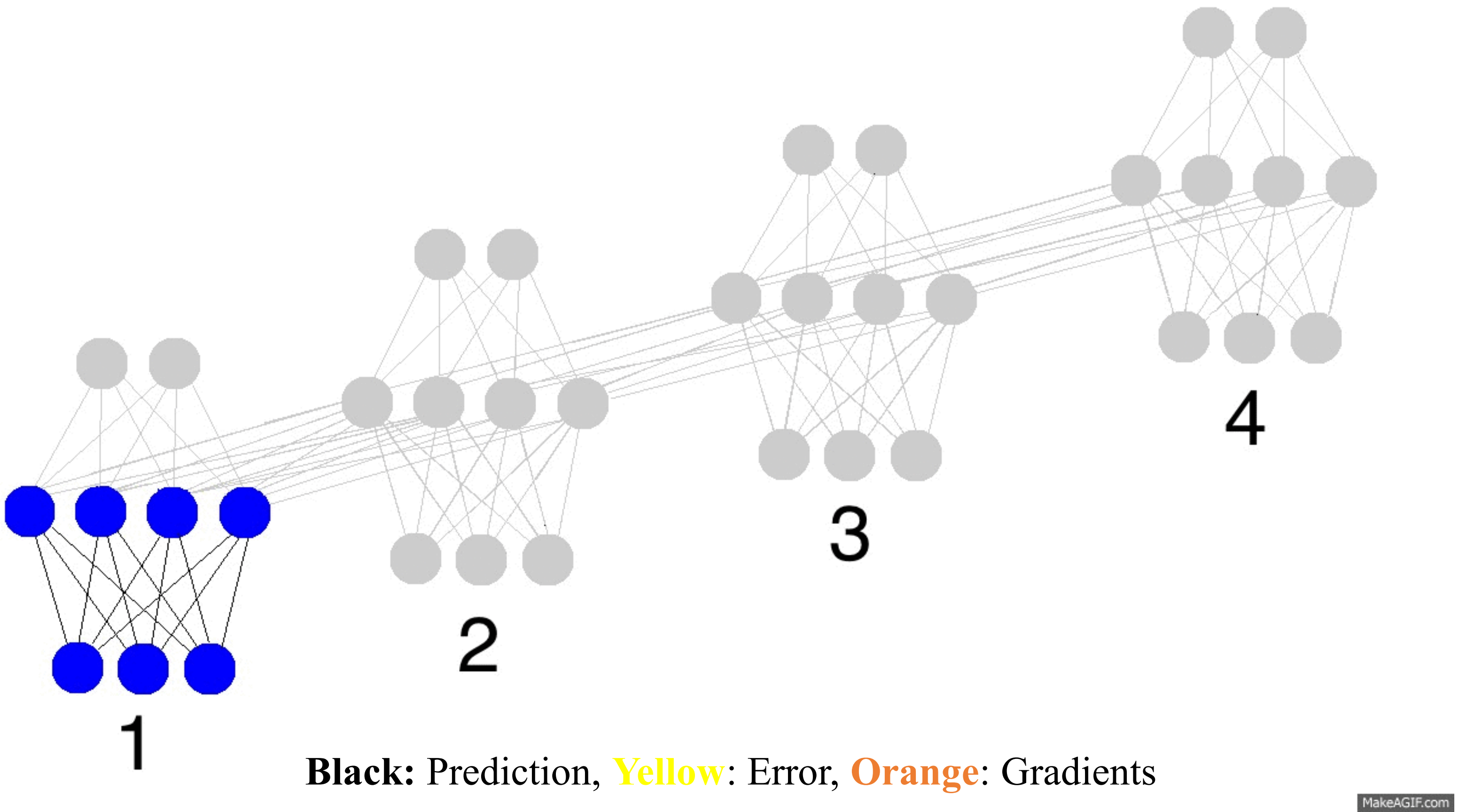
Pic source: https://iamtrask.github.io/

Workflow of RNN:

- The gif above reflects the magic of recurrent networks.
- It depicts 4 timesteps. The first is exclusively influenced by the input data.
- The second one is a mixture of the first and second inputs. This continues on.
- You should recognize that, in some way, network 4 is "full".
- Presumably, timestep 5 would have to choose which memories to keep and which ones to overwrite.
- This is very real. It's the notion of memory "capacity".
- As you might expect, bigger layers can hold more memories for a longer period of time.

Training RNN

- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- They learn by fully propagating forward from 1 to 4 (through an entire sequence of arbitrary length), and then backpropagating all the derivatives from 4 back to 1.
- You can also pretend that it's just a funny shaped normal neural network, except that we're re-using the same weights (synapses 0,1,and h) in their respective places.
- Other than that, it's normal backpropagation.

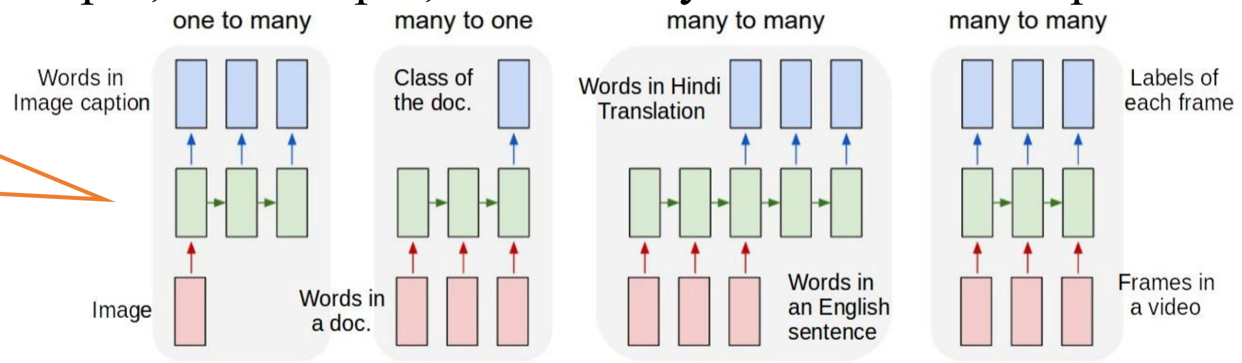
Black: Prediction, Yellow: Error, **Orange**: Gradients

Pic source: https://iamtrask.github.io/

RNN Applications

- In many problems, each input, each output, or both may be in form of sequences

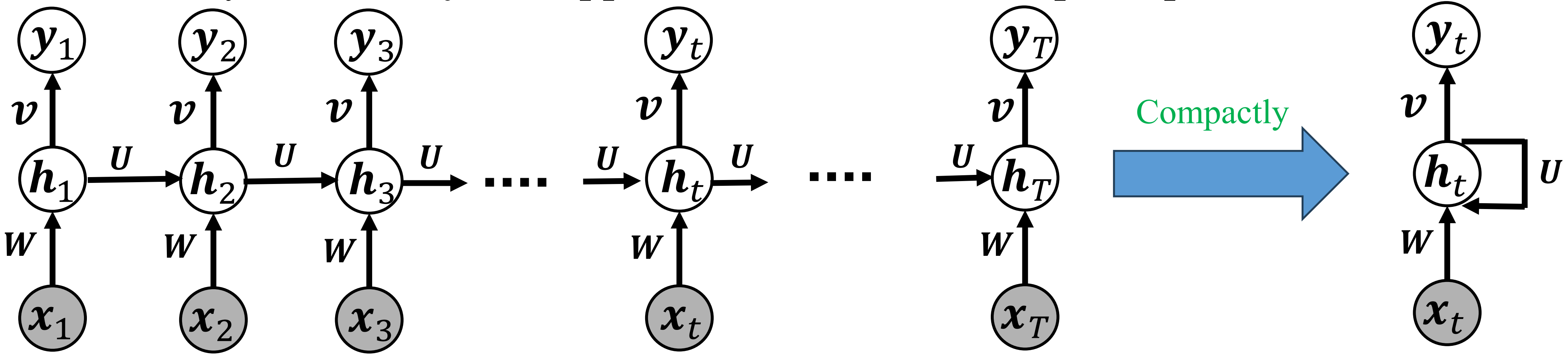
These green cells are some feature representation (e.g., hidden layer of a deep neural network) of the inputs



- Different inputs or outputs need not have the same length
- Some examples of prediction tasks in such problems
  - Image captioning: Input is image (not a sequence), output is the caption (word sequence)
  - Document classification: Input is a word sequence, output is a categorical label
  - Machine translation: Input is a word sequence, output is a word sequence (in different language)
  - Stock price prediction: Input is a sequence of stock prices, output is its predicted price tomorrow
  - No input – just output (e.g., generation of random but plausible-looking text)

RNNs: Long Distant Past is Hard to Remember

- The hidden layer nodes h_t are supposed to summarize the past up to time $t-1$



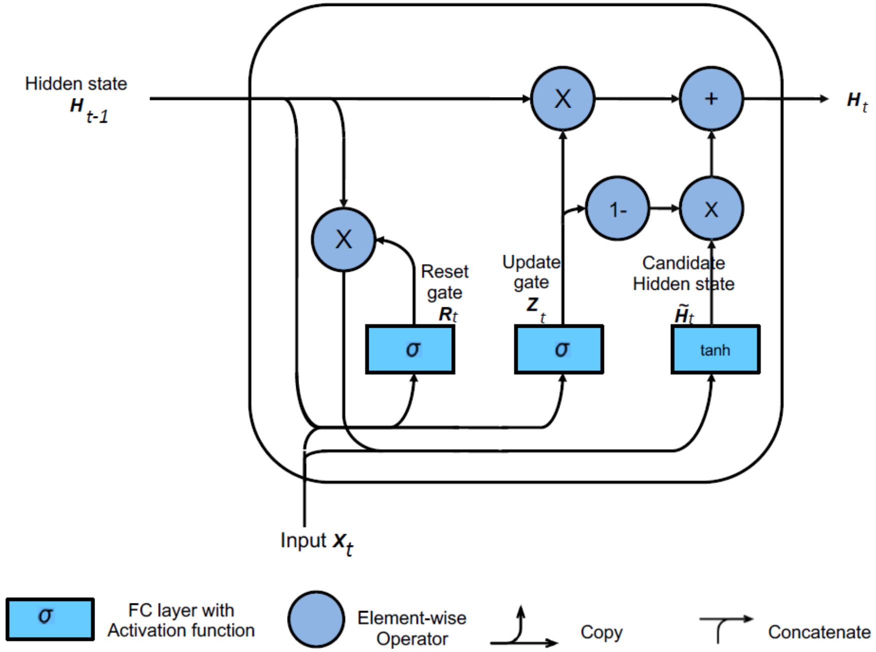
- In theory, they should. In practice, they can't. Some reasons
  - Vanishing gradients along the sequence too (due to repeated multiplications)
    – past knowledge gets "diluted"
  - Hidden nodes also have limited capacity because of their finite dimensionality
- Various extensions of RNNs have been proposed to address forgetting
  - Gated Recurrent Units (GRU), Long Short Term Memory (LSTM)

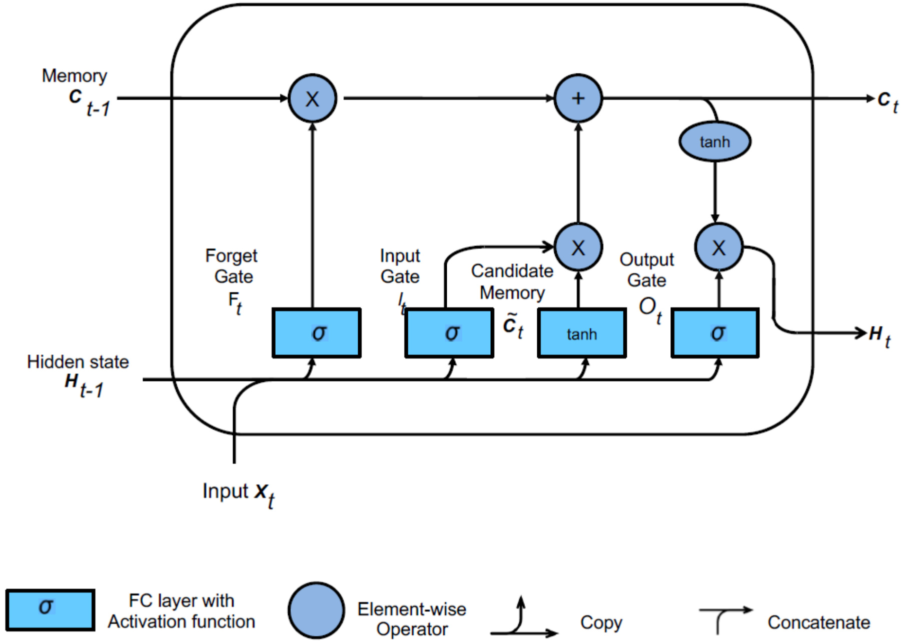
GRU and LSTM

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = tanh(\boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)
- GRU and LSTM contain specialized units and “memory” which modulate what/how much information from the past to retain/forget





Pic source: https://d2l.ai/

LSTM

- In contrast, LSTM maintains a "context" $\boldsymbol{C}_t$ and computes hidden states as

$$
\begin{aligned}
\hat{C}_t &= \tanh(\mathbf{W}^c \boldsymbol{x}_t + \mathbf{U}^c \boldsymbol{h}_{t-1}) && \text{("local" context, only up to immediately preceding state)} \\
i_t &= \sigma(\mathbf{W}^i \boldsymbol{x}_t + \mathbf{U}^i \boldsymbol{h}_{t-1}) && \text{(how much to take in the local context)} \\
f_t &= \sigma(\mathbf{W}^f \boldsymbol{x}_t + \mathbf{U}^f \boldsymbol{h}_{t-1}) && \text{(how much to forget the previous context)} \\
o_t &= \sigma(\mathbf{W}^o \boldsymbol{x}_t + \mathbf{U}^o \boldsymbol{h}_{t-1}) && \text{(how much to output)} \\
C_t &= C_{t-1} \odot f_t + \hat{C}_t \odot i_t && \text{(a modulated additive update for context)} \\
h_t &= \tanh(C_t) \odot o_t && \text{(transform context into state and selectively output)}
\end{aligned}
$$

- Note: $\odot$ represents elementwise vector product. Also, state updates now additive, not multiplicative. Training using backpropagation through time.

- Many variants of LSTM exists, e.g., using $\boldsymbol{C}_t$ in local computations, Gated Recurrent Units (GRU), etc. Mostly minor variations of basic LSTM above.

Exercise: RNN

Recurrent Neural Network (RNN)

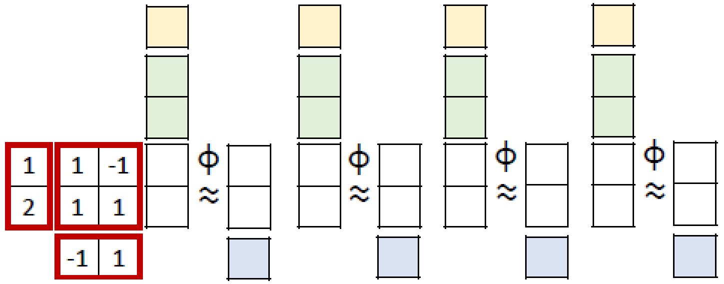
Input Sequence $X = \begin{bmatrix} 3 & 4 & 5 & 6 \end{bmatrix}$

Parameters $A = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$ $B = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ $C = \begin{bmatrix} -1 & 1 \end{bmatrix}$

Activation Function ϕ: ReLU

Hidden States $H_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Output Sequence $Y = \begin{bmatrix} \ & \ & \ & \ \end{bmatrix}$



Recurrent Neural Network (RNN)

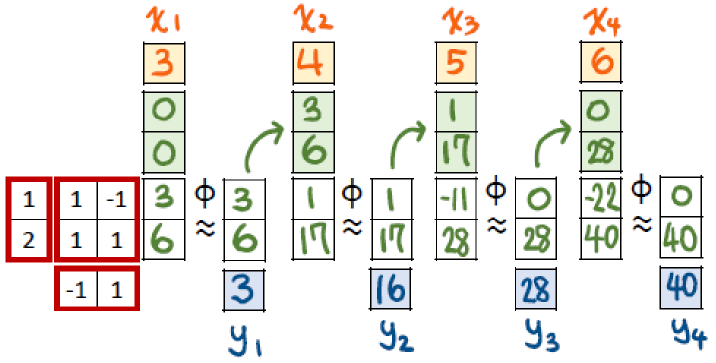
Input Sequence $X = \begin{bmatrix} 3 & 4 & 5 & 6 \end{bmatrix}$

Parameters $A = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$ $B = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ $C = \begin{bmatrix} -1 & 1 \end{bmatrix}$

Activation Function ϕ: ReLU

Hidden States $H_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

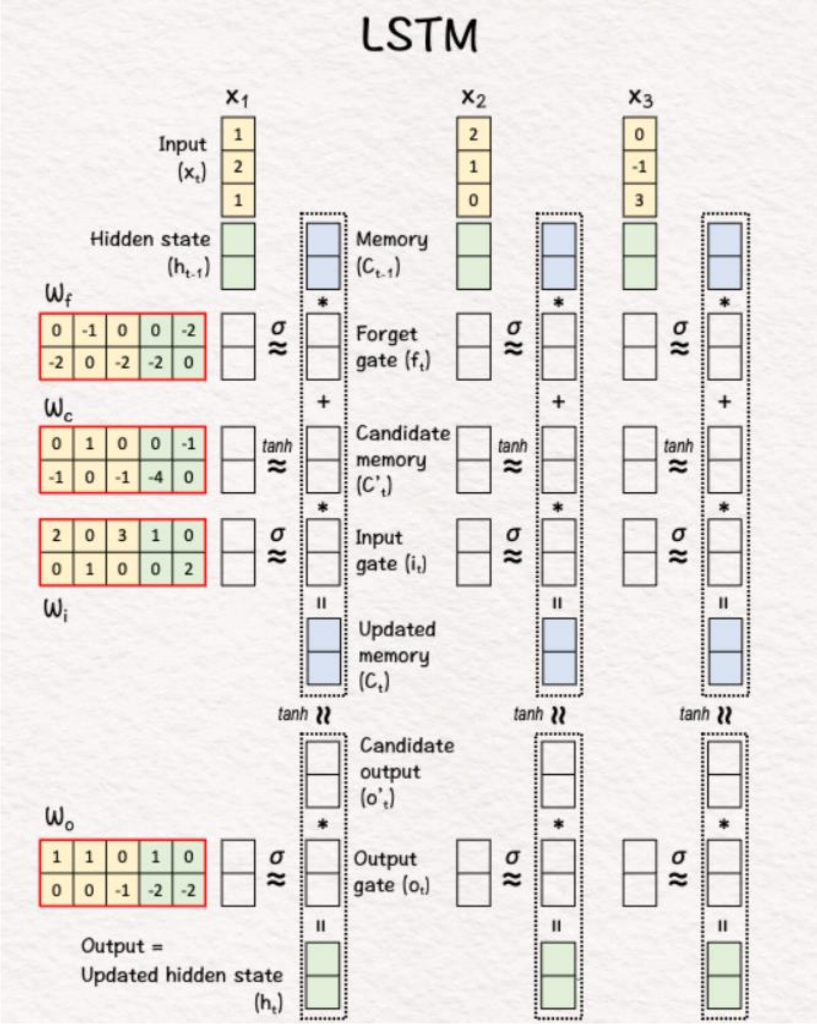
Output Sequence $Y = \begin{bmatrix} \ & \ & \ & \ \end{bmatrix}$

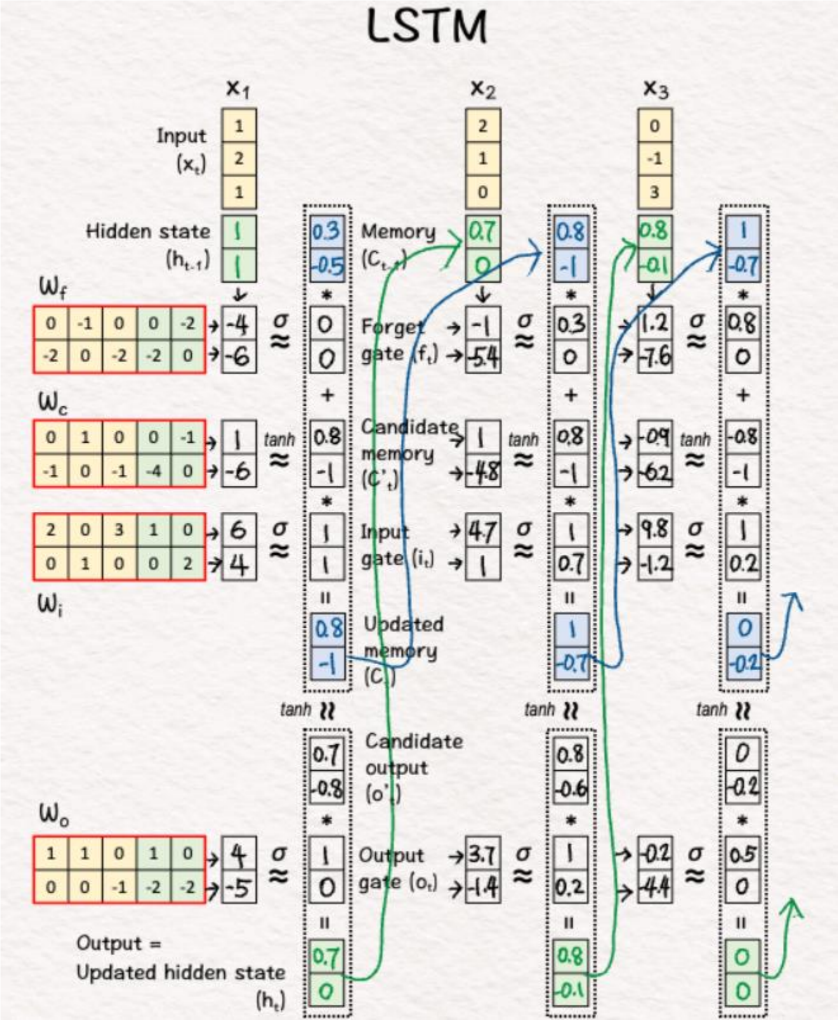


Pic source: https://www.byhand.ai/

Exercise: LSTM

Initialize: Randomly set the previous hidden state h0 to [1, 1] and memory cells C0 to [0.3, -0.5]



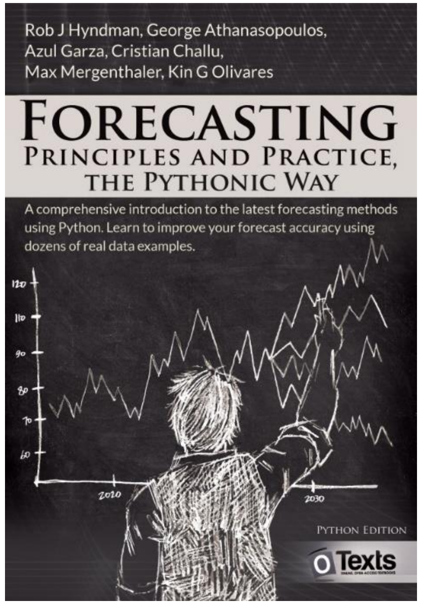


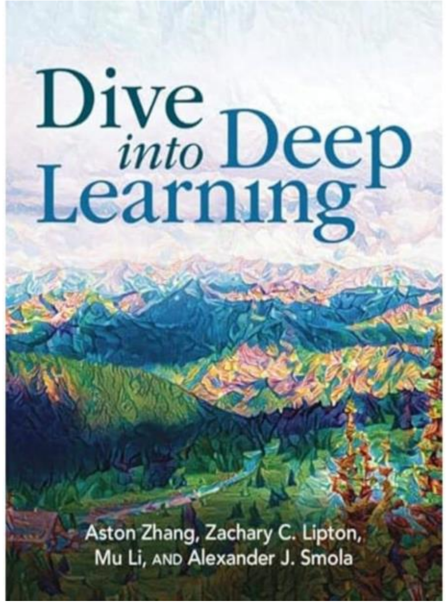
Pic source: https://www.byhand.ai/

Reference

HAPPY FORECASTING

"A good forecaster is not smarter than everyone else, he merely has his ignorance better organised."





Read Online: https://otexts.com/fpppy/ and https://d2l.ai/


If you're brave enough to say goodbye, life will reward you with a new hello.
Paulo Coelho