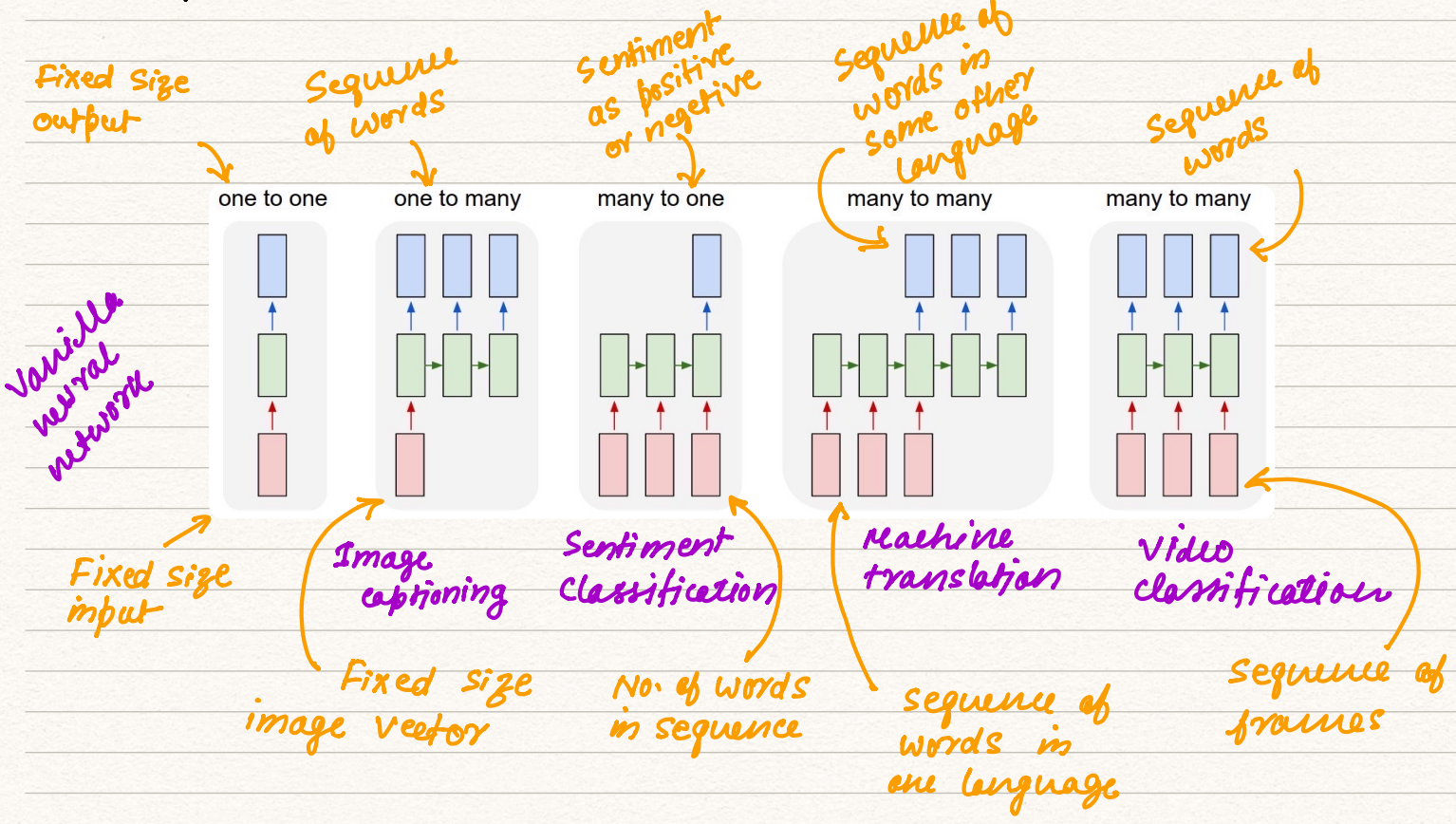
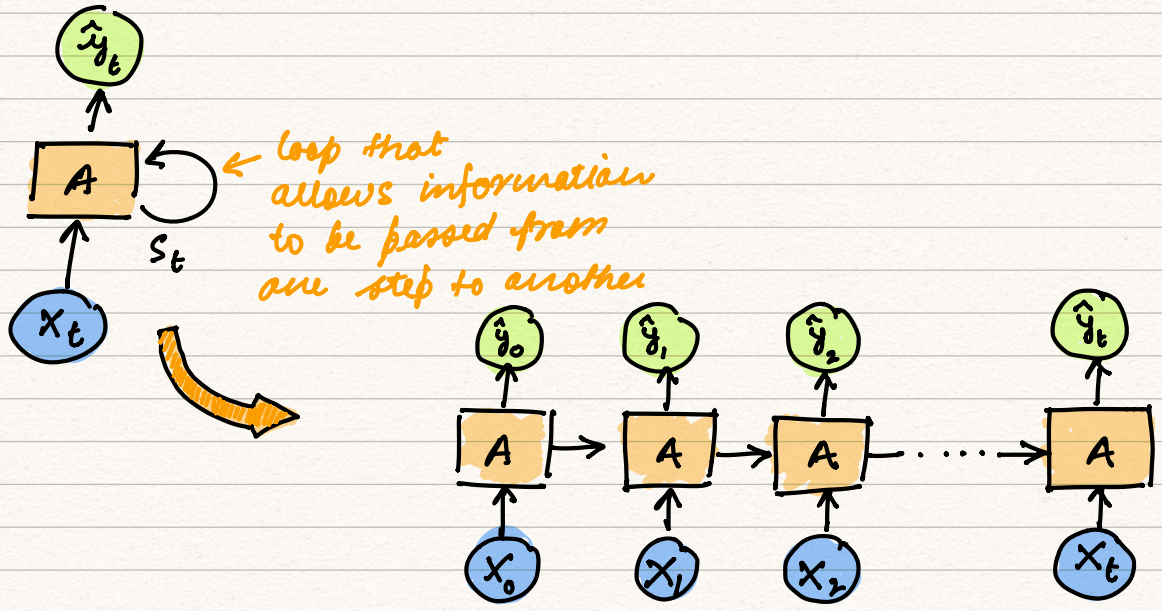
RNN → provides lot of flexibility in how you wire your neural net





$$s_t = f_w(s_{t-1}, x_t)$$

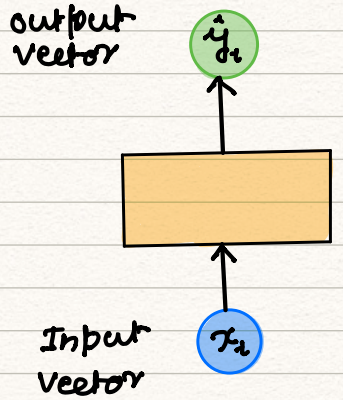
- s_t: New State
- f_w: Recurrence function with parameters w
- s_{t-1}: old State
- x_t: input vector at some time step

Allows us to use same function to every sequence of input irrespective of size of input/output sequence

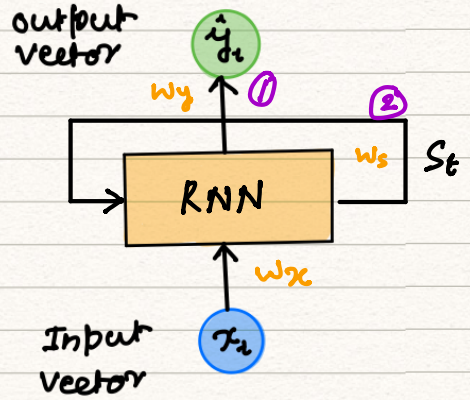
Same function and the same set of parameters are used at every time step.

4 Sequence modelling problem

1. Handle variable length sequence
2. Track long term dependencies.
3. Maintain information about order
4. Share parameter across the sequence



In vanilla feed forward neural network, we are going from input to output in one direction and it can not maintain information about sequence of data.



RNN computation involves:

1. output calculation
2. state update

$$s_t = \phi\left(w_s \cdot s_t + w_x \cdot x_t\right)$$

2 weights

ϕ → activation fn

s_t → A standard neural net operation:
→ Multiplication by weight metrices
→ Apply non-linearity

$$\hat{y}_t = w_y \cdot s_t$$

$\hat{y}_t$ → output vector

Total loss, $L = (L_1 + L_2 + L_3 + \cdots + L_t)$

Major limitation of RNN

Unable to capture relationships that span more than 8 or 10 steps back

↳ called "vanishing gradient problem" ← LSTM solves this problem

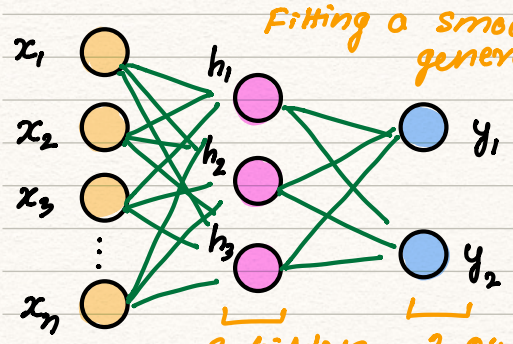
↓

contribution of information decays geometrically over time

Feed Forward Refresher

↳ It is non-linear function approximation

Fitting a smooth curve over the given data such that it generalises well for even the new set of data



n features — 3 hidden layers — 2 outputs

As there are only 1 hidden layer, steps:
① Finding $\bar{h}$ → from input x and weight w_1
② Finding $\bar{y}$ → from $\bar{h}$ and weights w_2

Feed forward.

Both these steps involve linear combination
↳ use matrix multiplication

w_1 w_2

(w_{ij}) → weight that connects x_i to h_j [$i \to 1$ to n ; $j \to 1$ to 3]

$x = [x_1, x_2, x_3, \dots, x_n]$ ← n cols →

$$w_1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ \vdots & & \\ w_{n1} & w_{n2} & w_{n3} \end{bmatrix}$$ n rows ↓ ; ← 3 columns →

h'

$\therefore [h_1', h_2', h_3'] = x \cdot w_1$

$1 \times 3 \quad = \quad 1 \times n \quad \cdot \quad n \times 3$ (same)

$h_1' = x_1 \cdot w_{11} + x_2 \cdot w_{21} + x_3 \cdot w_{31} + \dots + x_n \cdot w_{n1}$

← Activation function

$h = \phi(h')$

↳ Allow the network to represent non linear relationships between its inputs and its outputs

Most world data is non-linear

$$[y_1, y_2] = [h_1, h_2, h_3] \cdot \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

$1 \times 2 \quad = \quad 1 \times 3 \quad \cdot \quad 3 \times 2$ (3 rows ↓ ; ← 2 cols →)

$y = h \cdot w_2$

y ← output ← Activation fn is optional here.

↳ you can use softmax function if you want probability score

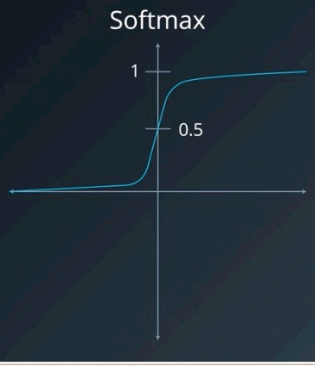
Softmax function

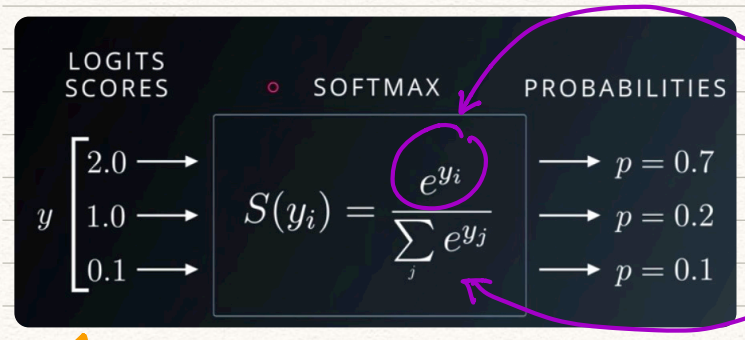
A activation function that turns numbers (logits) into probabilities that sum to one

↓

outputs a vector that represents the probability distributions of a list of potential outcomes.

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_1^k e^{x_k}}$$





Converts all outputs to 0 or positive

divide it by sum to normalise

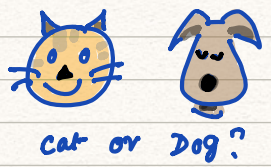
Logit layers are used as last layer neuron layer for neural net for classification task

↳ produces raw prediction values as real numbers ranging from $-\infty$ to $+\infty$

Loss Function → Cross Entropy Loss is the key loss function used with softmax function

Note that sigmoid function can be seen as special case of softmax where logit values are $[0, x]$ or for binary classification.

$$\therefore S(x_i) = \frac{e^0}{(e^0 + e^x)} = \frac{1}{1+e^x}$$



Python implementation

$x = [x_1, x_2, x_3, \ldots, x_m]$

exps = np.exp(x)

$S = \frac{exps}{np.sum(exps)}$

Note that for large values of x, the respective exp function value will be nan

↓

use normalization

Standard: $\frac{x - np.mean(x)}{np.std(x)}$

min-max: $\frac{x - np.min(x)}{np.max(x) - np.max(x)}$

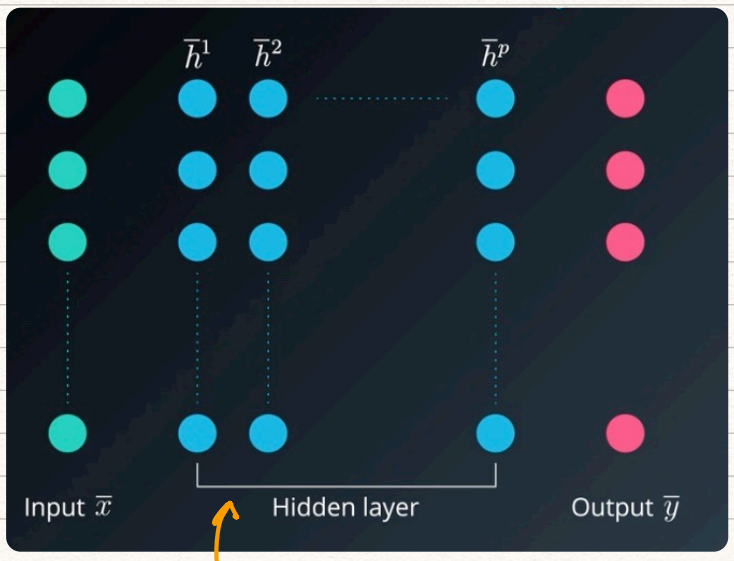
Feed Forward ... cont

For a good approximation of f there will be more hidden layers

Lets say there are p hidden layers

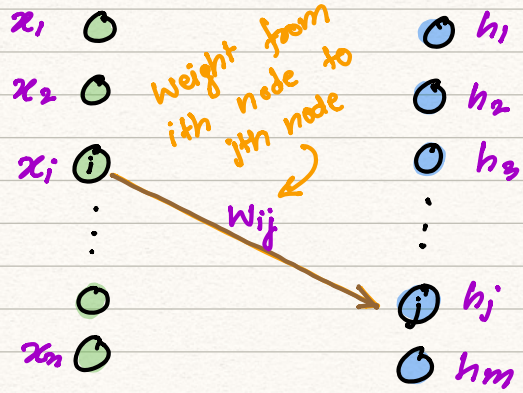
$$h^p = \phi(X \cdot W_p)$$

- h^p: output of a hidden layer (vector)
- X: input (vector)
- W_p: weights at layer p (matrices)
- ϕ: activation function



No. of neurons can be different in each layers

Input Layer k → Hidden Layer $k+1$



$$h_1 = \phi\left(\sum_{i=1}^{n} x_i \times w_{i1}\right)$$

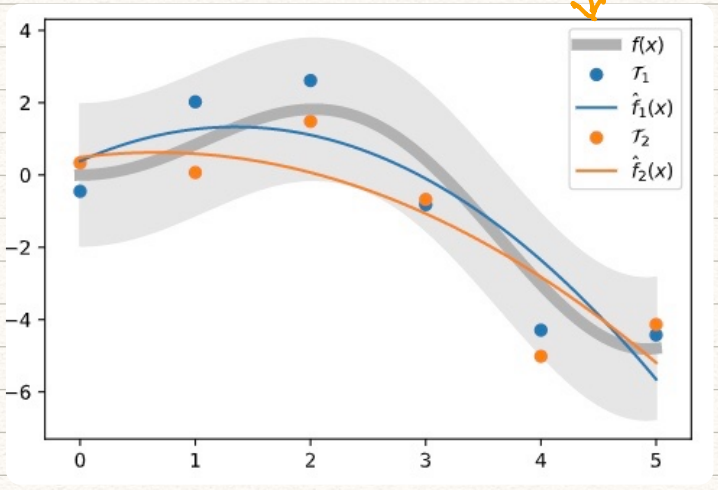
$$h_2 = \phi\left(\sum_{i=1}^{n} x_i \times w_{i2}\right)$$

$$h_m = \phi\left(\sum_{i=1}^{n} x_i \times w_{im}\right)$$

Note that bias is just another input

Error

T_1 & T_2 are 2 different Experiments



Lets say, we want to find an approximation to $f(x)$

we can never observe this function directly.

We observe some training set T (T_1 & T_2)

we use this to arrive at approximation → $\hat{f}_T(x)$

In general, approximation won't be a perfect fit, because of 2 source of error:

1. Systematic error (bias) → comes from the choice of model
2. Random error (variance) → comes up from the randomness inherent in the training set

Overall, we want to estimate $f(x)$ by observing T we come up with estimator $\hat{f}_T(x)$ and expect $[\hat{f}_T(x) - f(x)]$ to be low

MSE (Mean Squared Error) → common approach to measure average error in regression.

$\hat{f}_T(x) = \hat{y}$ ← predicted value

$f(x) = y$ ← Actual value

$$MSE = \underbrace{\mathbb{E}_T}_{\text{average over training set}} \overbrace{(\hat{y} - y)^2}^{\text{sum of error squared}}$$

$$Bias(x) = \mathbb{E}_T(\hat{y} - y)$$

Lets consider, average prediction $= \mu(x) = \mathbb{E}_T(\hat{y})$

$$Bias(x) = \mu(x) - f(x)$$

↳ Here we have distributed expectation over the two terms (because expectations are linear).

$$Variance / Var(x) = \mathbb{E}_T(\hat{y} - \mu(x))^2$$

The Bias-Variance decomposition

$$MSE(x) = \mathbb{E}_T(\hat{y} - y)^2$$

$$= \mathbb{E}_T(\hat{y}^2 - 2\hat{y}y + y^2)$$

$$= \mathbb{E}_T(\hat{y}^2) - 2\underbrace{\mathbb{E}_T\hat{y}}_{=\mu(x)} \cdot y + \mathbb{E}_T y^2$$

$$= \mathbb{E}_T(\hat{y})^2 - 2\mu(x)f(x) + f(x)^2$$

$$= \mathbb{E}_T(\hat{y})^2 - \mu(x)^2 + \mu(x)^2 - 2\mu(x)f(x) + f(x)^2$$

$$= \underbrace{\mathbb{E}_T(\hat{y})^2 - \mu(x)^2}_{\text{variance}} + \underbrace{(\mu(x) - f(x))^2}_{\text{Bias}}$$

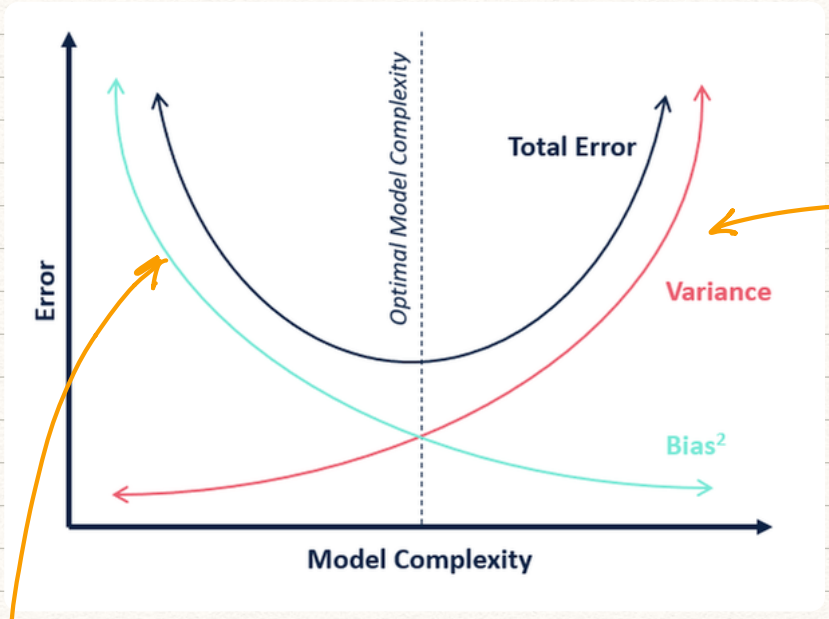
$$= Var(x) + Bias(x)^2$$

As model complexity increases, more of MSE can be attributed to variance

↳ using regularization, we artificially increase bias to reduce variance.

So, Expected prediction error or $EPE(x) = Var(x) + Bias(x)^2 + \sigma^2$

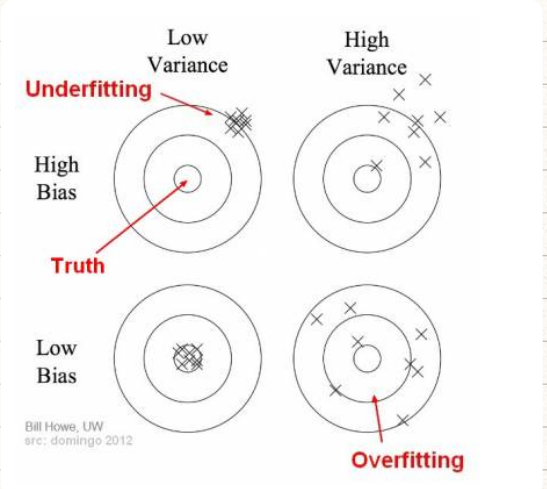
irreducible error (Noise)



Model with high Variance pays a lot of attention to training data and does not generalize on new data

Model with high bias pays very little attention to the training data and oversimplifies the model.

↳ leads to high error on both training and test data



Backpropagation

Goal → find weights that minimises error

Backpropagation algorithm looks for the min of the error function in weight space using the method of gradient descent

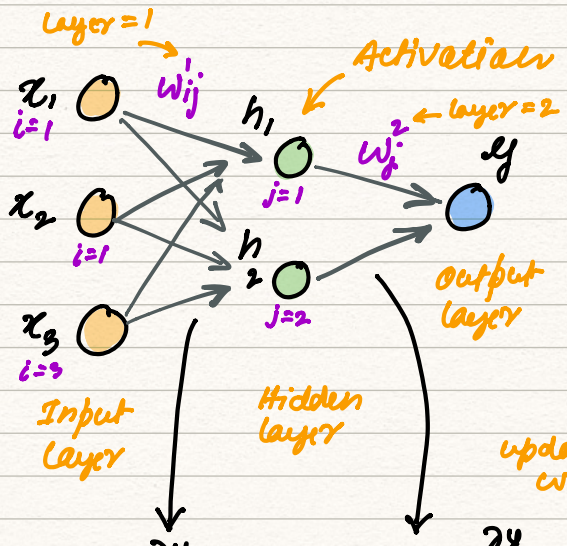
For gradient calculation, we must guarantee the continuity and differentiability of the error function.

$w_{ij} = w_{ij} + \Delta w_{ij}$ ← gradient

$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = -\alpha \frac{\partial \frac{(\hat{y}-y)^2}{2}}{\partial w_{ij}} \Rightarrow \boxed{\Delta w_{ij} = \alpha (\hat{y}-y) \frac{\partial \hat{y}}{\partial w_{ij}}}$

$E = \frac{(\hat{y}-y)^2}{2}$

$= -\alpha \frac{2(\hat{y}-y) \times \partial(\hat{y}-y)}{\partial w_{ij}}$ (2 cancels with $\frac{1}{2}$; $\partial(\hat{y}-y) = -1\left(\frac{\partial \hat{y}}{\partial w_{ij}}\right)$)



Forward pass

$$\hat{h}_j = \Sigma(w_{ij} \cdot x_i) \quad h_j = \phi(\hat{h}_j)$$

$$y = \Sigma w_j h_j$$

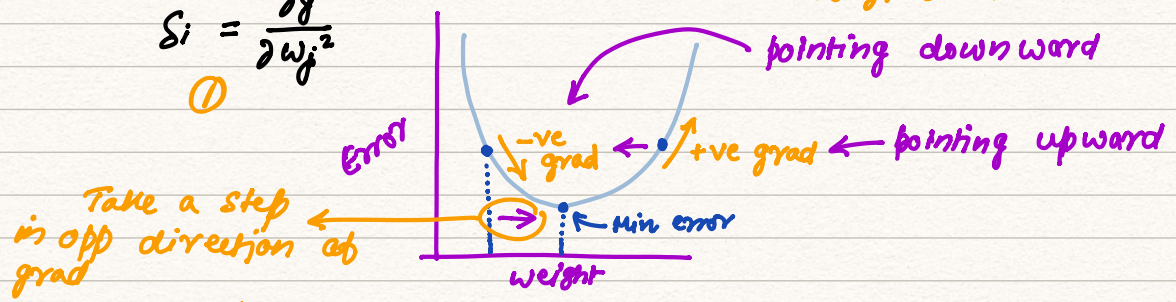
learning rate

$$w_{new} = w + \alpha\left(-\frac{\partial E}{\partial w}\right)$$

w_{new} ← updated weight; w ← current weight; α ← step in opp direction to gradient; $\frac{\partial E}{\partial w}$ ← gradient (partial derivative); E ← error

$$S_{ij} = \frac{\partial y}{\partial w_{ij}^1} \quad ②$$

$$S_i = \frac{\partial y}{\partial w_j^2} \quad ①$$

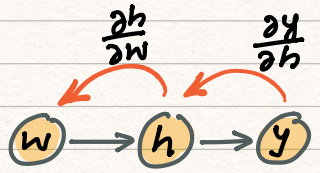


① $\frac{\partial y}{\partial w_i^2} = \frac{\partial(w_i^2 h_i)}{\partial w_i^2}$ ← Linear summation of terms

$= h_i$

$\frac{\partial y}{\partial h} = \frac{\partial(w_j^2 h_j)}{\partial h} = w_j^2$

② $\frac{\partial y}{\partial w_{ij}^1}$



← Chain rule

$\frac{\partial y}{\partial w} = \frac{\partial y}{\partial h} \times \frac{\partial h}{\partial w}$

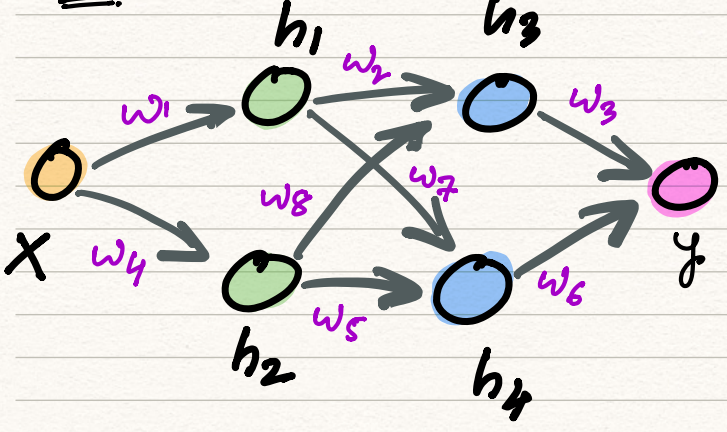
$\frac{\partial h}{\partial w} = \frac{\partial(\phi_j(x_i w_{ij}))}{\partial w_{ij}}$

$= \frac{\partial \phi_j}{\partial} \times \frac{\partial(x_i w_{ij})}{\partial w_{ij}}$

Partial derivative of activation fu → $= \phi_j' \times x_i$

$= w_j^2 \times \phi_j' \times x_i$

Example



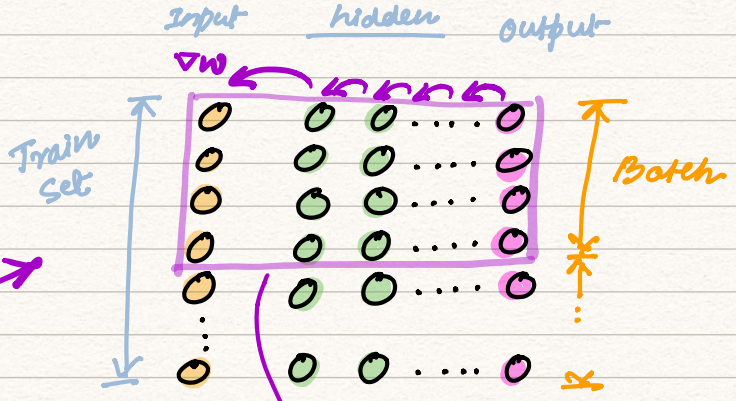
What is update rule for weight matrix w_1

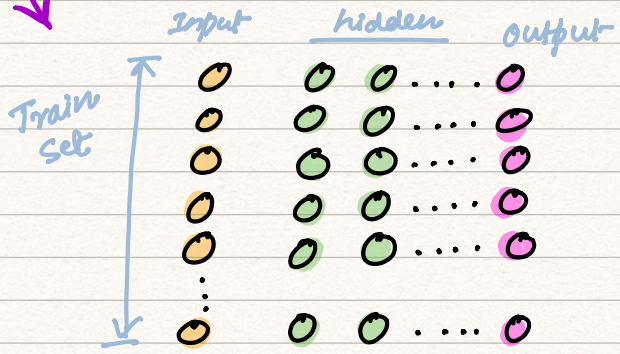
$$\frac{\partial y}{\partial w_1} = \left[\frac{\partial y}{\partial h_3} \times \frac{\partial h_3}{\partial h_1} \times \frac{\partial h_1}{\partial w_1} + \frac{\partial y}{\partial h_4} \times \frac{\partial h_4}{\partial h_1} \times \frac{\partial h_1}{\partial w_1}\right]$$

Gradient Descent

Stochastic gradient descent (SGD)

Mini Batch Gradient Descent





→ calculate gradient for every input

→ update weight

$$w_{new} = w + \alpha(-\nabla w)$$

→ calculate gradient for every input

→ calculate mean gradient for a batch

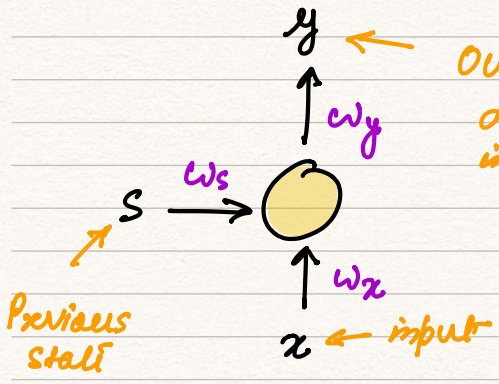
→ update weight

$$w_{new} = w + \alpha\left(\sum_{batch} -\nabla w\right)$$

Back to ... **RNN** (Recurrent Neural Network).

↳ Takes care of temporal dependencies

↳ dependencies over time

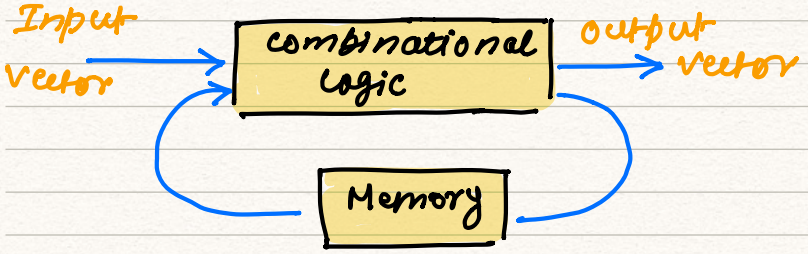


Output not only depends on input but also on previous state

usage

1. Sentiment Analysis
2. Speech recognition
3. Time Series prediction
4. NLP
5. Gesture recognition

State Machine

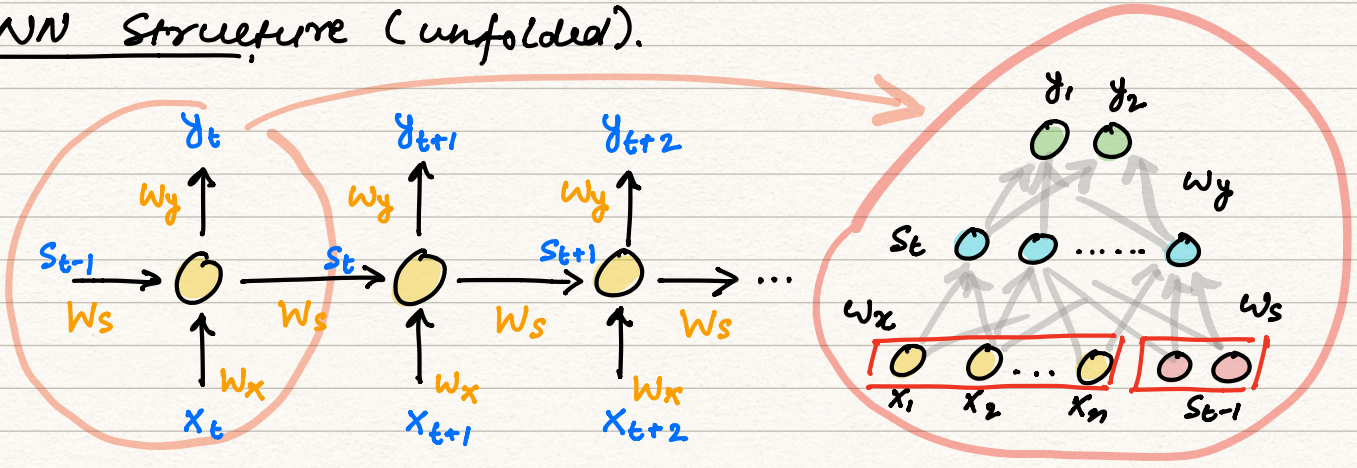


2 key feature of a RNN →

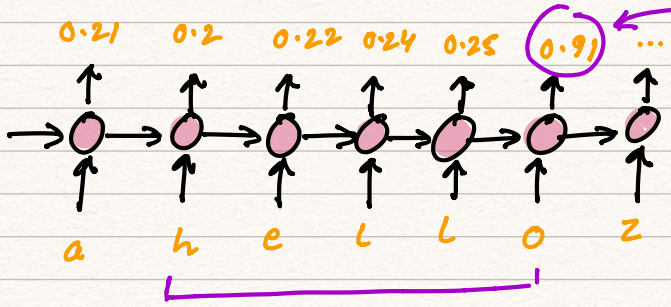
1. Sequences as input in training phase
2. memory elements — output of hidden layer neurons, which will serve as additional input to the network during next training step

Elman Network → the basic 3 layer neural network with feedback that serve as memory inputs

RNN Structure (unfolded).

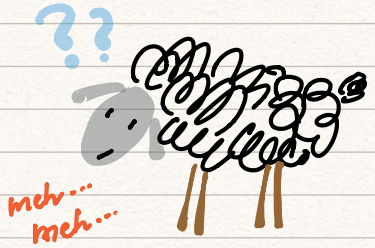


Example → we train model on word 'hello'



Indicates that sequence was observed. as output is close to 1.

Note that we can train RNN on different length of words. RNN can deal with varying sequence length.



Backpropagation through Time (BPTT)

activation fn

$$S_t = \tanh(x_t \cdot w_x + S_{t-1} \cdot w_s)$$

S_t: state at t — x_t: input — S_{t-1}: state at $(t-1)$

$y_t = S_t \cdot w_y$; In case you are using softmax

$$y_t = S(S_t \cdot w_y)$$

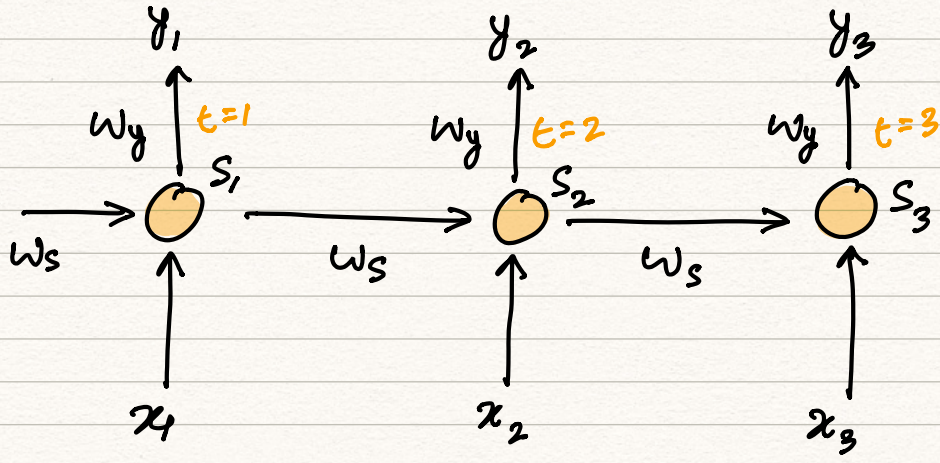
y_t: output at t — S: softmax

prediction

$$E_t = (\hat{y}_t - y_t)^2$$

E_t: Error

$$E_1 = (\hat{y}_1 - y_1)^2 \qquad E_2 = (\hat{y}_2 - y_2)^2 \qquad E_3 = (\hat{y}_3 - y_3)^2$$

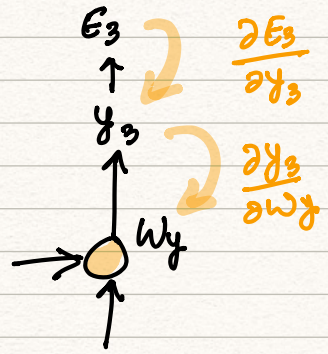


Lets adjust weight at $t=3$.

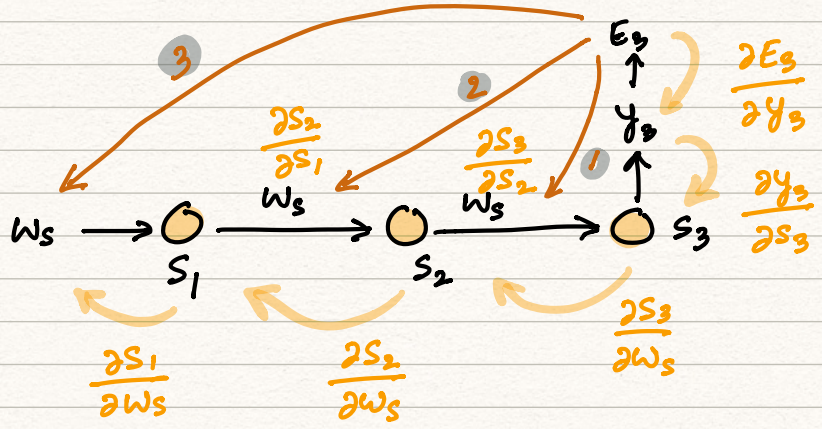
① Adjust weight matrix W_y

$$E_3 = (\hat{y}_3 - y_3)^2$$

$$\frac{\partial E_3}{\partial W_y} = \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial W_y}$$



② Adjust weight matrix W_s



$$\frac{\partial E_3}{\partial W_s} = 1+2+3$$

1) $$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial s_3} \times \frac{\partial s_3}{\partial W_s}$$

2) $$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial s_3} \times \frac{\partial s_3}{\partial s_2} \times \frac{\partial s_2}{\partial W_s}$$

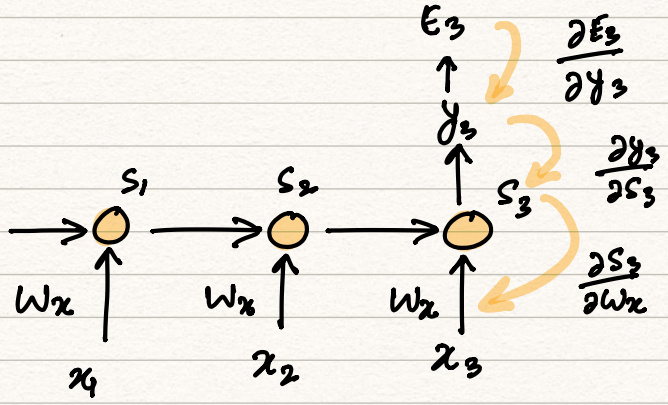
3) $$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial s_3} \times \frac{\partial s_3}{\partial s_2} \times \frac{\partial s_2}{\partial W_s}$$

$$\therefore \frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial s_3} \times \frac{\partial s_3}{\partial W_s} + \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial s_3} \times \frac{\partial s_3}{\partial s_2} \times \frac{\partial s_2}{\partial W_s} + \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial s_3} \times \frac{\partial s_3}{\partial s_2} \times \frac{\partial s_2}{\partial s_1} \times \frac{\partial s_1}{\partial W_s}$$

General Equation

$$\frac{\partial E_n}{\partial W_s} = \sum_{i=1}^{n} \frac{\partial E_n}{\partial y_n} \times \frac{\partial y_n}{\partial s_i} \times \frac{\partial s_i}{\partial W_s}$$

③ Adjust weight matrix W_x



Gradient = accumulative gradient of each of the previous steps.

State S_3 State S_2

$$\frac{\partial E_3}{\partial w_x} = \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial s_3} \times \frac{\partial s_3}{\partial w_x} + \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial s_3} \times \frac{\partial s_3}{\partial s_2} \times \frac{\partial s_2}{\partial w_x}$$

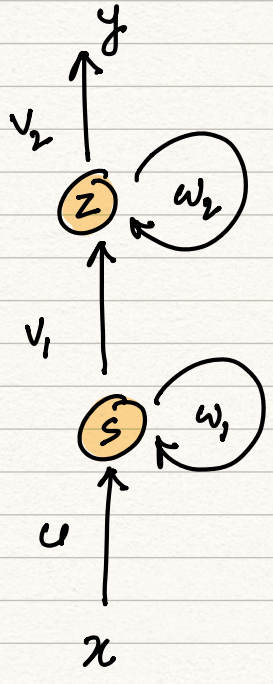
$$+ \frac{\partial E_3}{\partial y_3} \times \frac{\partial y_3}{\partial s_3} \times \frac{\partial s_3}{\partial s_2} \times \frac{\partial s_2}{\partial s_1} \times \frac{\partial s_1}{\partial w_x}$$

State S_1

General Equation

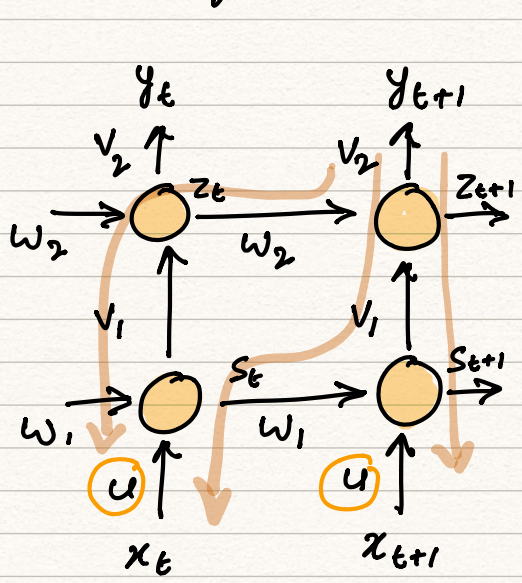
$$\frac{\partial E_n}{\partial w_x} = \sum_{i=1}^{n} \frac{\partial E_n}{\partial y_n} \times \frac{\partial y_n}{\partial s_i} \times \frac{\partial s_i}{\partial w_x}$$

Example



Assume that error is 'E'. What is update rule of weight matrix 'U' at time $t+1$

Solution the diagram in unfolded —



$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial y_{t+1}} \times \frac{\partial y_{t+1}}{\partial z_{t+1}} \times \frac{\partial z_{t+1}}{\partial s_{t+1}} \times \frac{\partial s_{t+1}}{\partial U}$$

$$+ \frac{\partial E_{t+1}}{\partial y_{t+1}} \times \frac{\partial y_{t+1}}{\partial z_{t+1}} \times \frac{\partial z_{t+1}}{\partial s_{t+1}} \times \frac{\partial s_{t+1}}{\partial s_t} \times \frac{\partial s_t}{\partial U}$$

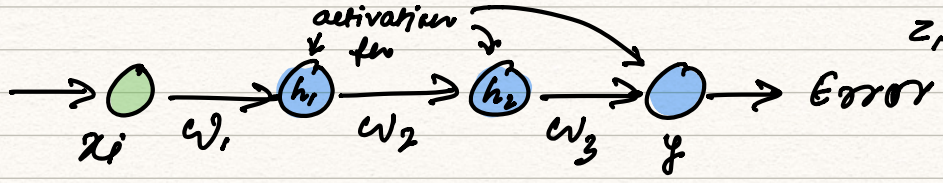
$$+ \frac{\partial E_{t+1}}{\partial y_{t+1}} \times \frac{\partial y_{t+1}}{\partial z_{t+1}} \times \frac{\partial z_{t+1}}{\partial z_t} \times \frac{\partial z_t}{\partial s_t} \times \frac{\partial s_t}{\partial U}$$

Vanishing gradient problem → If we backpropagate more than 8-10 time steps, then gradient will become too small (contribution of information decays exponentially over time).

↳ In summary, temporal dependency spanning more than 8-10 timestep will be disregarded.

↓

LSTM addresses this issue.



z_1 and z_2 → outputs of hidden layers

Assume, we are using sigmoid activation fn.

$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial y} \times \frac{\partial y}{\partial z_2} \times \frac{\partial z_2}{\partial z_1} \times \frac{\partial z_1}{\partial W_1}$$

$\hat{z}_2 = h_2 \times w_3 \mid z_2 = \sigma(\hat{z}_2)$

$\frac{\partial(\sigma \cdot \hat{z}_2)}{\partial \hat{z}_2} \times w_3 = \boxed{\sigma' \cdot w_3}$

$\boxed{\sigma' \times w_2}$

$\sigma' = 1/4$

weights → chosen using Gaussian dist with mean = 0 & Std = 1 ∴ all terms here are < 1 hence prod will be very small and will with increasing layers

Exploding Gradients → the gradient grows uncontrollably.