DIFFERENCE BETWEEN CONVOLUTION LAYER AND FULLY/ DENSELY CONNECTED LAYER

Dense layers learn "global" patterns in their input feature space (e.g. for a MNIST digit, patterns involving all pixels)

Conv layers learn "local" patterns (e.g. for a MNIST digit, patterns involving small 2D windows of inputs)

KEY CHARACTERISTICS/PROPERTIES OF CONVNETS:

- Convnets learn patterns that are 'translation invariant': i.e. After learning a pattern in the lower right corner of a picture, a convnet can recognise it anywhere

- Convnets learn 'spatial hierarchies' of patterns: A first convolution layer will learn small local patterns such as edges, a second conv. layer will learn larger patterns made of features of the first layer and so on. This allows convnets to learn increasingly complex visual concepts

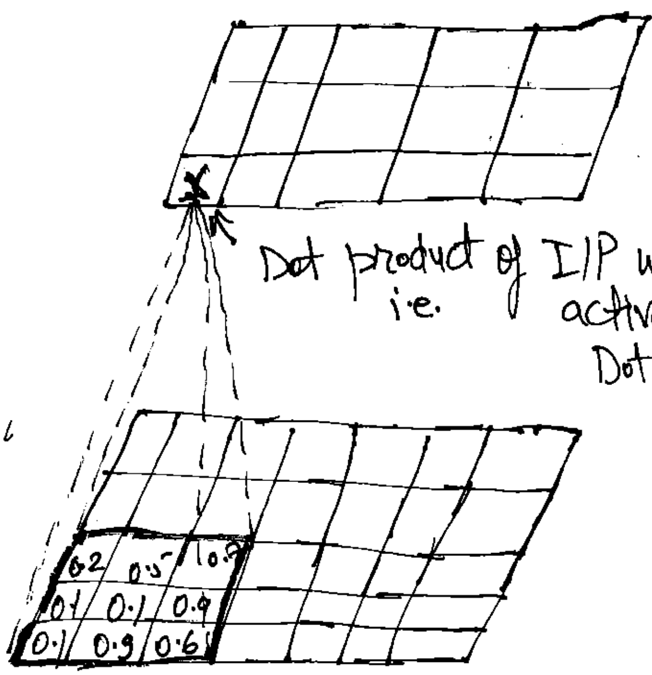
CONVOLUTION LAYER

- Convolution layer is defined by two key params:
  i) Size of patches extracted from inputs = filter size (typically 3X3)

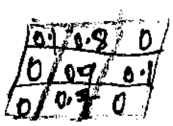
  ii) # of filters = # of O/P feature maps
  conv. kernels

Note: If the I/P to conv. layer is n-dimensional (e.g. 3), the filter depth is n-dimensional

Example: in case of RGB image, the first conv layer's filters will have depth 3
And if conv2D layer is used then O/P of this layer is 2D and the # of filters determine the depth of O/P from this layer i.e. # of feature maps



Dot product of I/P with filter followed by activation
i.e. activation (dot product I/P with filter)
Dot product results in scalar hence one value



→ Filter (3X3)

Stride: → The way the filter moves from one position to next on the I/P

→ The factor by which width and ht. of I/P feature maps are downsampled (in addition to any changes induced by padding)

e.g. stride = 2 means width and ht. of I/P feature map is downsampled by ~~factor~~ of 2

→ Typically we keep stride = 1 in conv. layer

Padding: → When the filter moves across the I/P feature map, the output feature map's width & ht. are still reduced (even when stride = 1)

a	b	c
d	e	f
g	h	i

I/P feature map
3x3

using 2x2 filters with stride = 1

ab de	bc ef
de gh	ef hi

2 x 2
O/P feature map

Zero Padding → To make O/P feature map's width & ht = I/P feature map's width & ht

i.e. I/P feature map is padded with cells left, right, top & bottom in such a way that O/P feature map's width & ht = I/P feature map's width & ht

Note: Depth of O/P feature map depends on # of filters

→ Filters = receptive field

e.g. if the filter is
$$\begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline \end{array}$$
$\xrightarrow{\text{will detect}}$ Central vertical line (if the image is grayscale i.e. matrix of values b/w 0 & 1)

Thus a layer full of neurons using the same filter O/Ps a feature map which highlights the areas in an image that activate the filter the most.

→ The values in the filter are the wts. that are learnt during the training process.

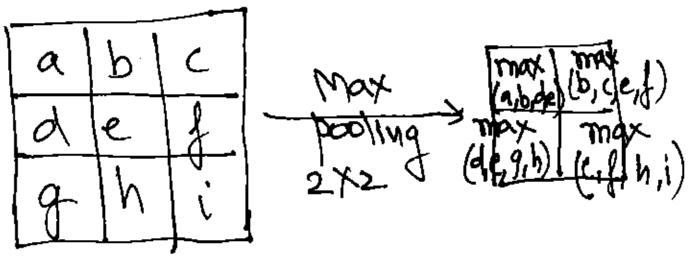
→ One filter results in one feature map
That's why all neurons in the feature map share the same parameters

→ When one conv. layer is directly connected to another conv. layer (i.e. without any other layer in between), the O/P feature map of one conv. layer goes as I/P feature map to next conv. layer

POOLING LAYER

→ Pooling layer is defined by:
  i) size of window/patches (typically 2×2)
  ii) stride (typically 2)

→ Pooling consists of extracting windows from I/P feature map and performing the pooling operation (max, avg) on the extracted window ← O/P of each channel (feature map)



→ Downsamples width & ht of I/P feature map by the factor of stride, does not downsample depth/# of feature maps (typically)

→ However, downsampling by depth can also be done (not provided in Keras, need to use low level TF APIs)

→ Pooling layer has no wts ⇒ all it does is aggregate the inputs using an agg. func. such as mean, max etc.

CNN ARCHITECTURES

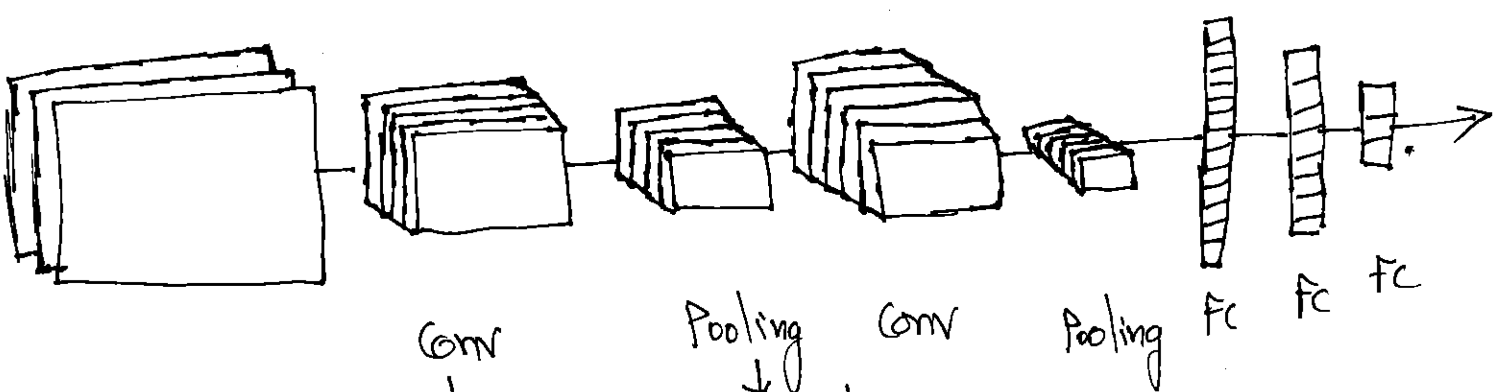
Typical CNN architecture involves:

1) (Convolutional layer (+ReLU) + Pooling layer) x n + Fully connected layer

Or

2) (Convolutional layer (+ReLU) x n + Pooling layer) x n + Fully connected layer

The image gets smaller and smaller as it progresses thru the n/w but it typically gets deeper and deeper (i.e. with more feature maps)
typically # of filters increase as the n/w size increases



# of filters = # of O/P feature maps

⇒ width and ht. decreases but # O/P feature maps increases (since # of filters typically increase when going deep)

CONVNET CODE (KERAS)

```
model = models.sequential ( )
model.add (layers.Conv2D (32, (3,3) ,activation ='relu',
                          input_shape =(28,28,1)))

model.add (layers. MaxPooling2D ((2,2)))
model.add (layers. Conv2D (64, (3,3) , activation = 'relu'))
model.add ( layers. MaxPooling2D ( (2,2))
model.add (layers.Conv2D (64,(3,3), activation = 'relu'))
model.add (layers. Flatten ( ) )
model.add (layers. Dense (64, activation = 'relu' ))
model.add (layers. Dense (10, activation= 'softmax'))
```

```
model.summary.( )
```

Layer	O/P Shape	Param #
conv2d_1	(None, 26,26,32)	320 $(\underset{\text{\# of filters}}{32} \times \underbrace{3\times3}_{\text{filter size}} + \underbrace{32}_{\substack{\text{bias} \\ = \\ \text{\# of filters}}})$
maxpooling2d_1	(None, 13, 13, 32)	0 (No trainable params for pooling)

Conv2d_2	(None, 11, 11, 64)	18496 $(\underbrace{32}_{\text{\# of I/P filters from prev. conv layer}} \times \underbrace{64}_{\text{\# of filters in this layer}} \times \underbrace{3\times3}_{\text{size of filters}} + \underbrace{64}_{\text{bias = \# of filters in this layer}})$
maxpooling2d_2	(None, 5, 5, 64)	0
Conv2d_3	(None, 3, 3, 64)	$64\times 64\times3\times3 + 64 = 36928$
Flatten_1	(None, $\underset{3\times3\times64}{576}$)	0
Dense_1	(None, 64)	36928 $(576\times64 + 64)$
Dense_2	(None, 10)	650 $(\underbrace{64}_{\text{I/P from prev. layer}} \times \underbrace{10}_{\text{\# of nodes in this layer}} + \underbrace{10}_{\text{bias = \# of nodes in this layer}})$

Total params: 93,322

IMP: Pooling layer has zero trainable params
= all it does is aggregate

LEARNABLE PARAMS CALC: $\underset{\downarrow}{\text{Inputs}} \times \underset{\downarrow}{\text{outputs}} + \text{biases} \rightarrow$ # of filters

Inputs:
i) If last layer is dense = # of nodes
ii) If last layer is conv = # of filters

outputs: # of filters x size of filters

→ Filters define the wt and biases

→ One feature map share wts & biases i.e. different feature maps use different parameters

→ The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model.

⇒ Once the CNN has learned to recognize a pattern in one location, it can recognize in any other location

TYPES OF COMPUTER VISION PROBLEMS:

i) Image Classification – Image is of cat vs dog

ii) Object detection – In an image, what are the different objects e.g. wall, cat, vessel, shoes etc.

iii) Semantic Segmentation – In an image, what are the boundaries of different objects e.g. self driving cars → how diff. objects are de-lineated

PURPOSE OF 1X1 CONVOLUTION

Pooling layer 'typically' downsamples width & ht of I/P feature map with no change in depth / # of output feature maps

Conv. layer 'typically' increases depth with no change in width or ht of I/P feature map (if zero padding)

1X1 conv. is used to downsample or reduce the depth or # of O/P feature maps

So, 1X1 conv → Channel-wise / depth-wise pooling

can be thought of as:

↓

dimensionality reduction

(Shrinks the number of channels)

PURPOSE OF SKIP CONNECTIONS (RESIDUAL NETWORKS)

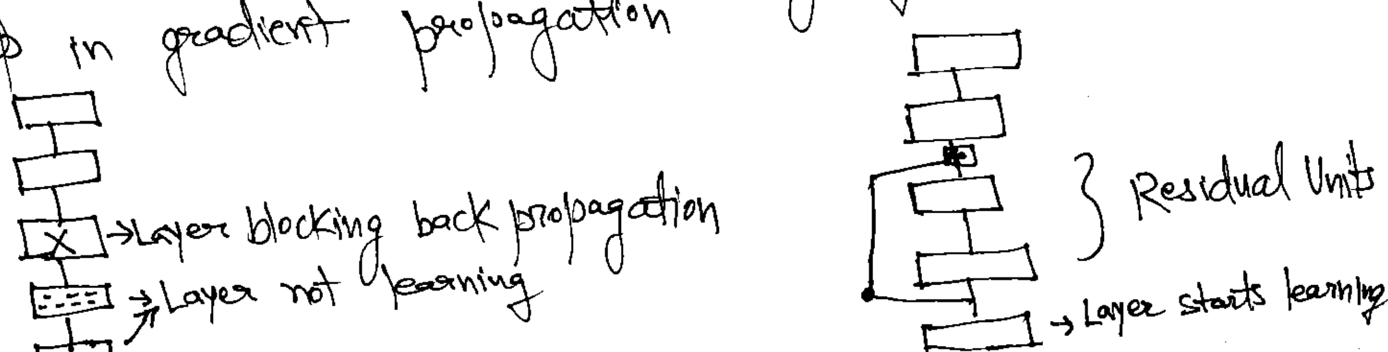
SKIP CONNECTIONS: The signal feeding into a layer is also added to the o/p of a layer located a bit higher up

The idea is if you take a "shallow" network and just stack on more layers to create a deeper network, the performance of the deeper network should be at least as good as the shallow network since the layers in b/w shallow & deep n/w should be at least able to learn the identity fn. However in practice, this was not the case → hence residual learning or skip connections

So this motivated the use of "deep residual layers" to allow network to learn deviations from the identity layer, hence the term "residual", residual here referring to difference from identity.

PURPOSE:

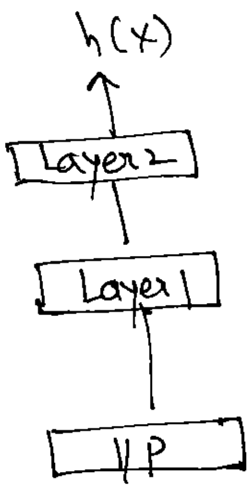
1) They overcome the issue of vanishing gradients and hence help in gradient propagation

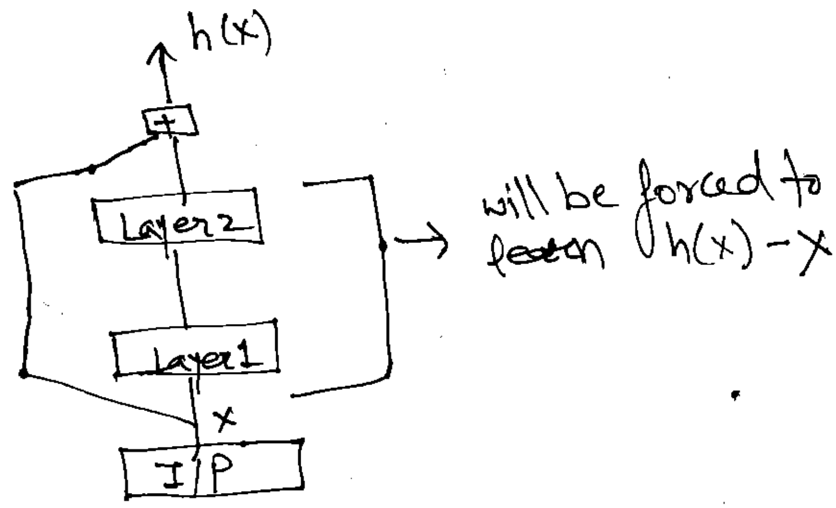


2) They make the skipped layers learn an identity function means the higher layer n/w will perform atleast as good as the lower layer n/w

RESIDUAL LEARNING:

When training a neural n/w, the goal is to make it model a target function h(X). If you add the input x to the output of n/w (i.e. you add a skip connection) then the n/w will be forced to model h(x) − x rather than h(x) → residual learning





RNN

Diff. b/w Feedforward Neural N/W & RNN:

Feedforward neural n/w do not maintain info about order of features in an instance

Whereas

RNNs maintain info about order of elements within a sequence. RNNs assume elements within a sequence have some order but different sequences are independent.

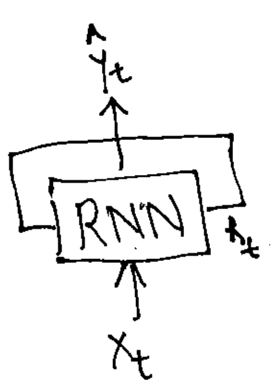
Instance / Example in FF Neural N/W = Sequence in RNNs

↓

RNNs internally loop over every element in the sequence

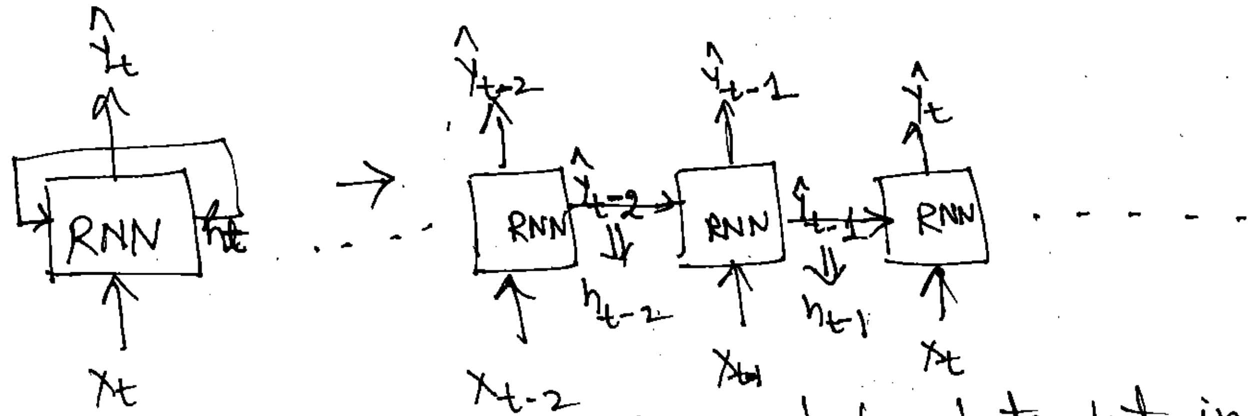
UNROLLING THE NETWORK THRU TIME:

Every element in a sequence is at a different time-step meaning 1^{st} element is at timestep 1
2^{nd} element is at timestep 2



At every timestep t, every neuron receives both the Input vector X_t and output vector from previous timestep Y_{t-1}

Since this O/P vector Y_{t-1} keeps accumulating info from prev timesteps e.g. $Y_{t-2}, Y_{t-3}, Y_{t-4}$ .. etc



In basic cells, the o/p is simply equal to state but in complex cells, hidden state may be different than o/p.

Recurrent : info is passed from one timestep to next in a seq.

RNNs:

Apply a recurrance relation at every timestep to process a seq.

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

h_t → cell state

f_W → fn parameterized by W

h_{t-1} → old state

x_t → I/P vector at time step t

RNNs maintain an internal state h_t. At every time step they apply a function f (parametrized by a set of wts. W) to update the state h.

This state update is dependent on prev. state h_{t-1} as well as current input.

Imp. to note : the same function (with the same set of params) are used at every timestep in a sequence

→ Just like in feed forward neural nets, wts are learnt across instances, in RNNs wts. are learnt across sequences i.e. wts and biases (params) are shared for a sequence

→ Every recurrent cell has two sets of wts: one depending of I/P (W_x) and other depending on prev. hidden state (W_h)

→ I/P vector : x_t

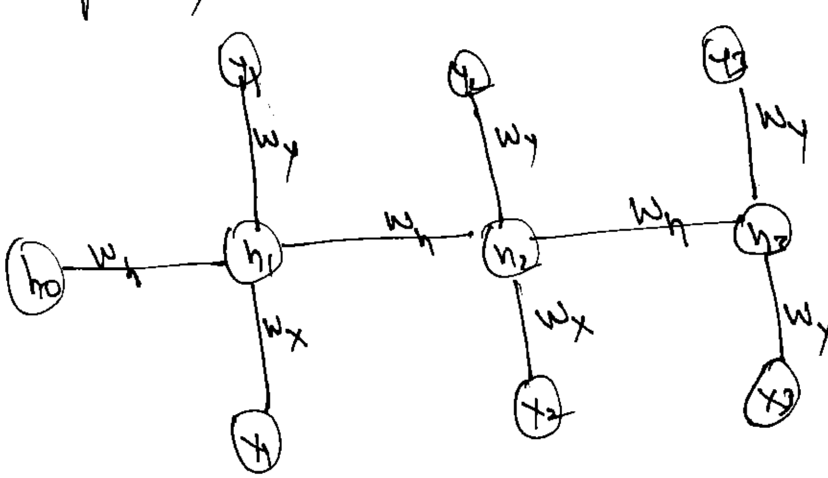
↓

update hidden state : $h_t = \tanh(W_h^T h_{t-1} + W_x^T x_t)$

↓

O/P vector : $\hat{y}_t = W_y^T h_t$

→ We have W_y since same input can produce diff. O/Ps depending on hidden state (or prev. inputs in the sequence)



TYPES OF RNN BASED ON I/P & O/P (APPLICATIONS OF RNN)

1. Sequence-to-Seq (Many-to-Many)

   An RNN can take a sequence of I/Ps and produce a sequence of O/Ps.

   Example: Time Series - you feed last N days of data and it will produce next N days predictions

2. Sequence-to-Vector (Many-to-One)

   An RNN can take sequence of I/Ps and ignore all O/Ps except the last one.

   Example: you could feed a seq of words corresponding to a movie review and the network would O/P a sentiment score

3. Vector-to-Sequence (One-to-Many)

   You could feed the n/w the same I/P vector over and over again at each timestep and let it O/P a sequence.

   Example: I/P Could be an image and O/P could be caption for that image

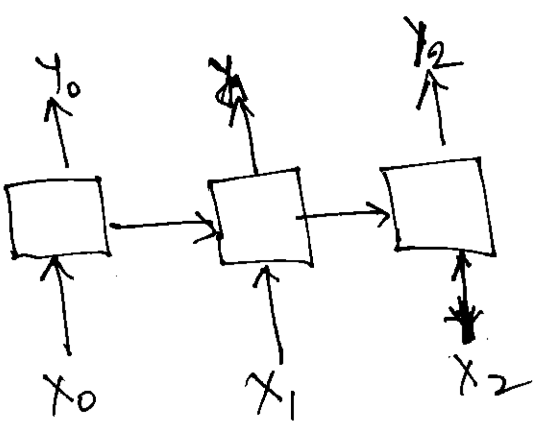
4. Seq-to-Vector followed by Vector-to-Seq

Seq-to-Vector → encoder

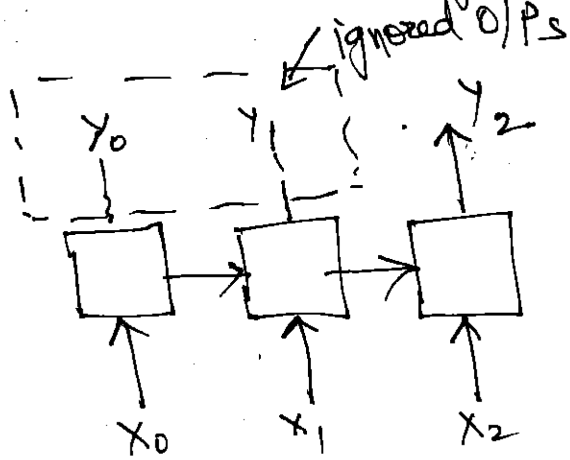
Vector-to-Seq → decoder

Example: Translating a sentence from one lang. to another

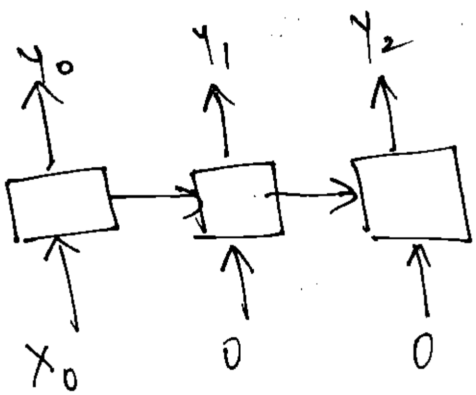
You could feed the n/w a sent. in one lang and the encoder would convert this sent. into a single vector representation, and then the decoder would decode this vector into a sentence in another lang.



Seq-to-Seq



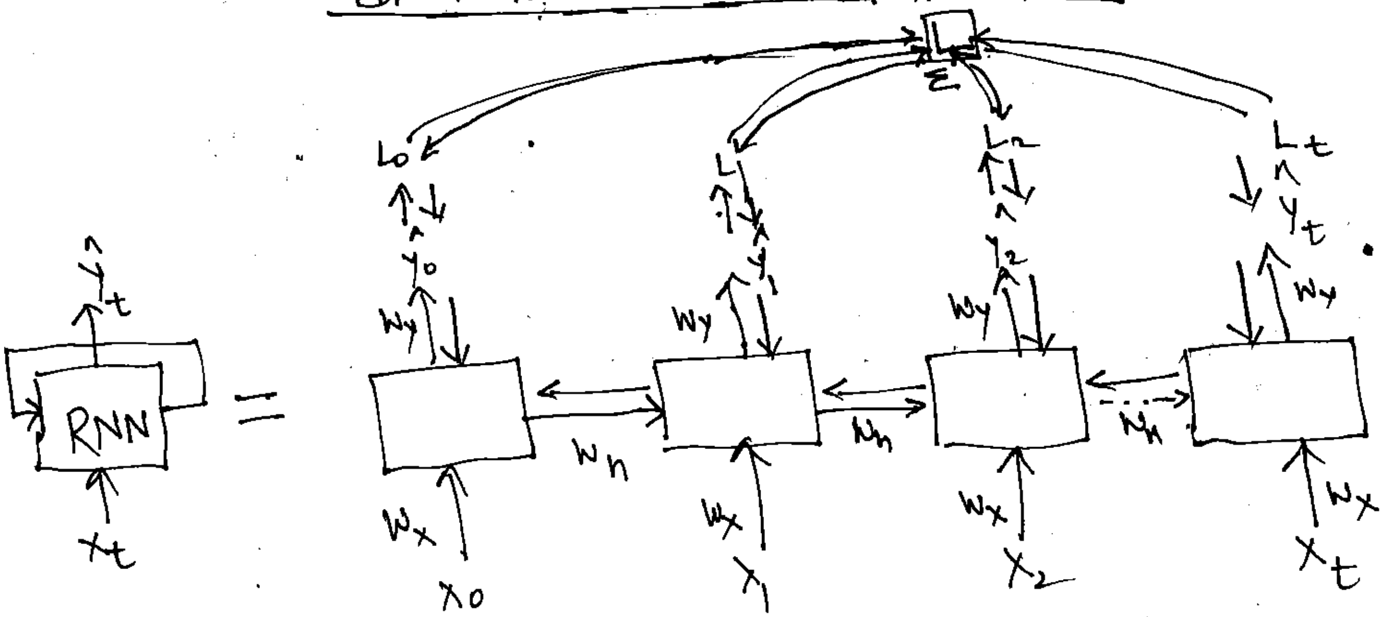
Seq-to-Vector



Vector-to-Seq

TRAINING RNNs

BACK-PROPAGATION THRU TIME



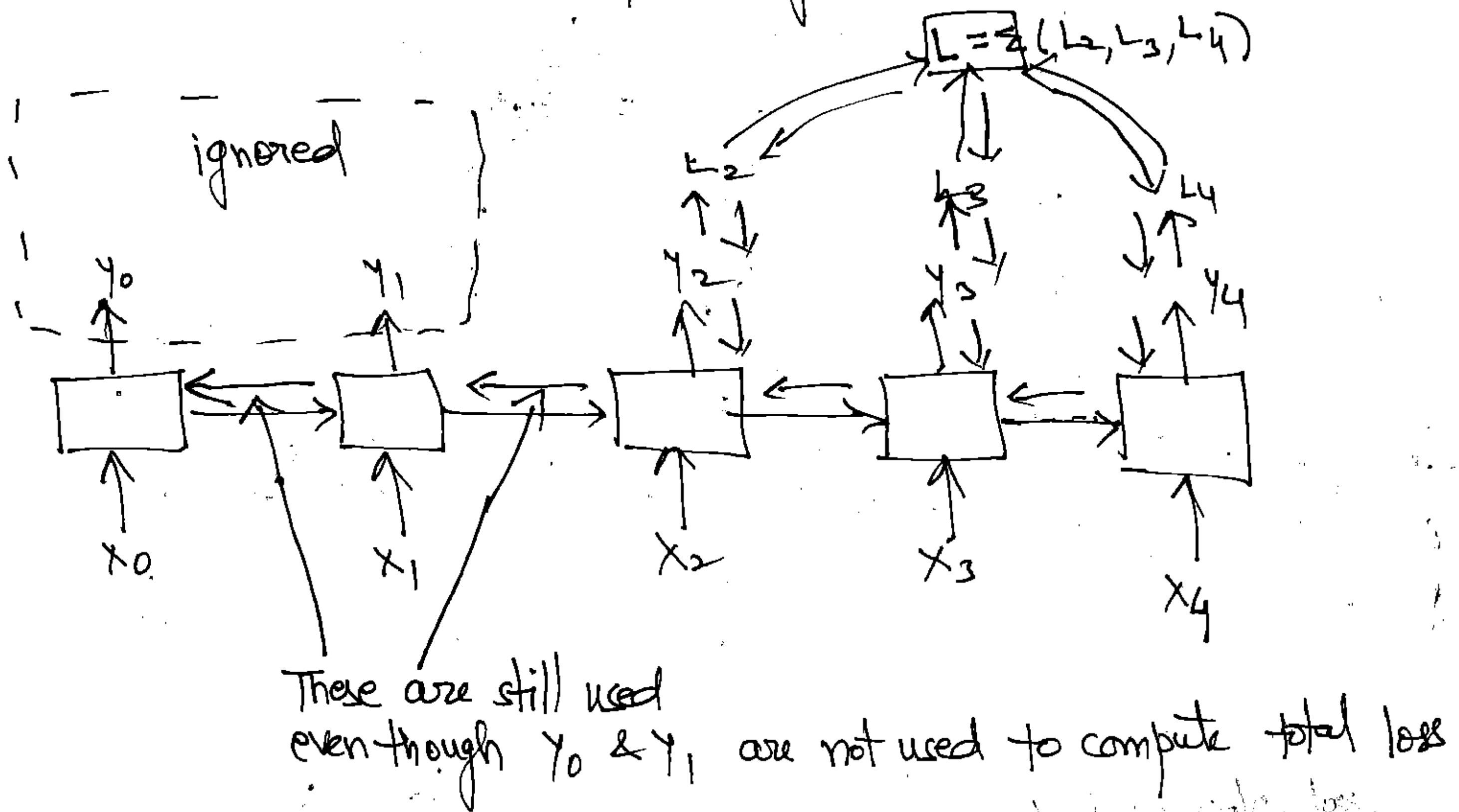
Cost Fn : $L(\hat{y}_0, \hat{y}_1 \ldots \hat{y}_t) = \Sigma(L_0, L_1, L_2 \cdots L_t)$

t = max timestep

This cost/loss fn may ignore some O/Ps depending on the configuration of RNN ex. if seq-to-vector conf. is used, only L_t is taken into account and rest of the O/Ps L_0, L_1, etc. are ignored. Also the error is propagated back only to these O/Ps (which the cost fn uses)

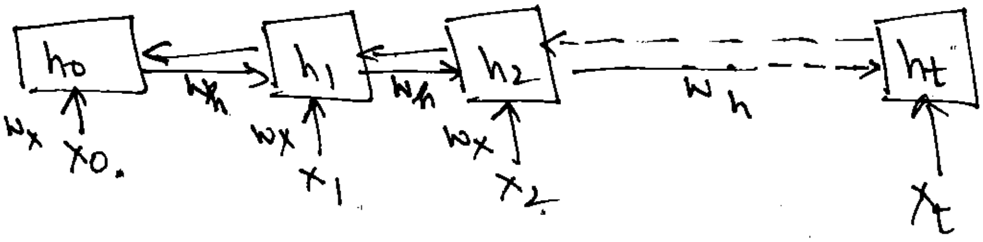
Errors are back-propagated at each timestep back upto the first timestep

In the scenario when y_0 & y_1 are ignored:



Since the parameters W_x, W_y, W_h are used at each time step, backpropagation will sum over all time steps

PROBLEMS IN TRAINING RNNs



Between each time step we need to perform matrix multiplication involving wt. matrix W_x & W_h (and W_y if seq to seq)

Also, cell update involves matrix multiplication with non-linear activation fn

⇒ Computation of gradient i.e. derivative of loss w.r.t. parameters all the way back to first time step requires many repeated matrix multiplication of wt. matrix + repeated use of derivative of activation fn → Problematic

Problems in training RNNs

exploding gradient

Many values involved in this repeated multiplication > 1

Soln: Gradient Clipping

Vanishing gradient

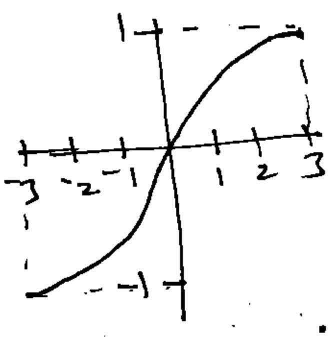
Many values in this repeated multiplication < 1

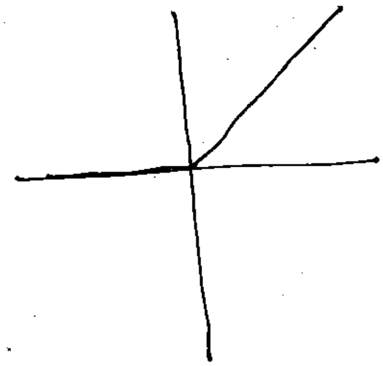
Soln: 1) Activation fn - relu instead of tanh / sigmoid

2) Wt. initialization - initialize wt to 1 & bias to 0

3) Network Architecture - LSTM (long term dependencies)

TANH VS RELU in RNNs



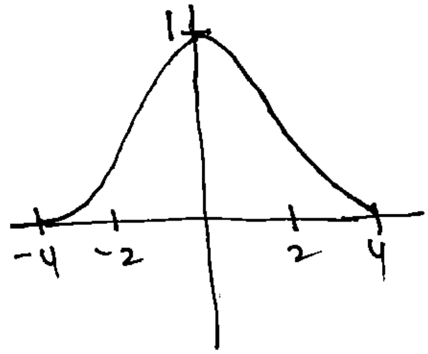


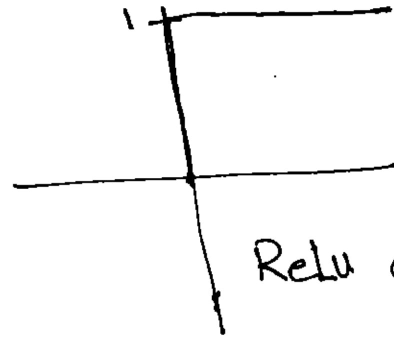
TANH
Always b/w -1 & 1
↓
Saturating Act. fn

RELU
If $x > 0$, unbounded
↓
non-saturating act. fn

To AVOID, exploding gradient problem : tanh is used as a default act. fn
since for $x > 0$, relu is unbounded
and if x goes greater than 1 then repeated multiplication will blow up.

However, the story is different for derivatives of tanh & relu and how they affect the vanishing gradient problem





Relu derivative

TANH Derivative

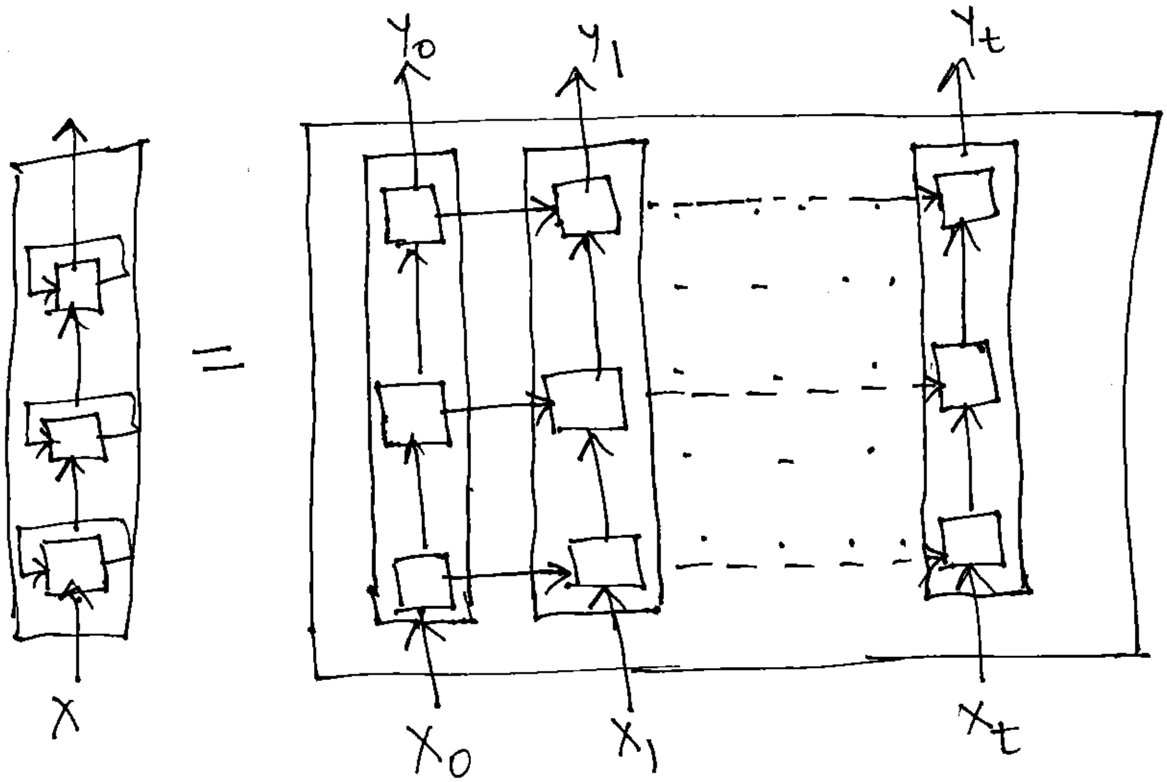
So when $x > 0$, relu prevents f' from shrinking the gradients.

✓ To AVOID Vanishing gradient problem: relu is preferred

But since other solⁿ exist for vanishing gradient problem like using LSTM etc., tanh is typically used as the activation fn.

So tanh prevents exploding gradient problem & LSTM (and other architectures e.g. GRU) prevent vanishing gradient problem.

DEEP RNN UNROLLED THRU TIME: → stacking multiple layers



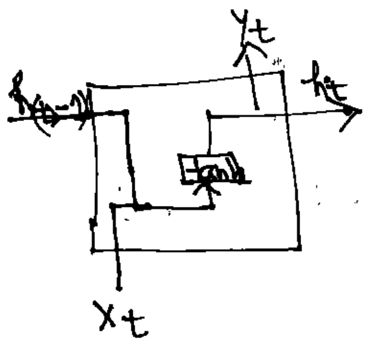
In Classical/traditional ML when modelling time series data ARIMA, MA models are used. To apply these models, time-series have to be stationary i.e. remove trend and seasonality. After the model is trained, you add trend or seasonality back to get final predictions

✓ When using RNN it is generally not needed to do all this. but it may improve performance in some cases.

LSTM

- To learn long-term dependencies
- To avoid vanishing gradient problem in RNNs.

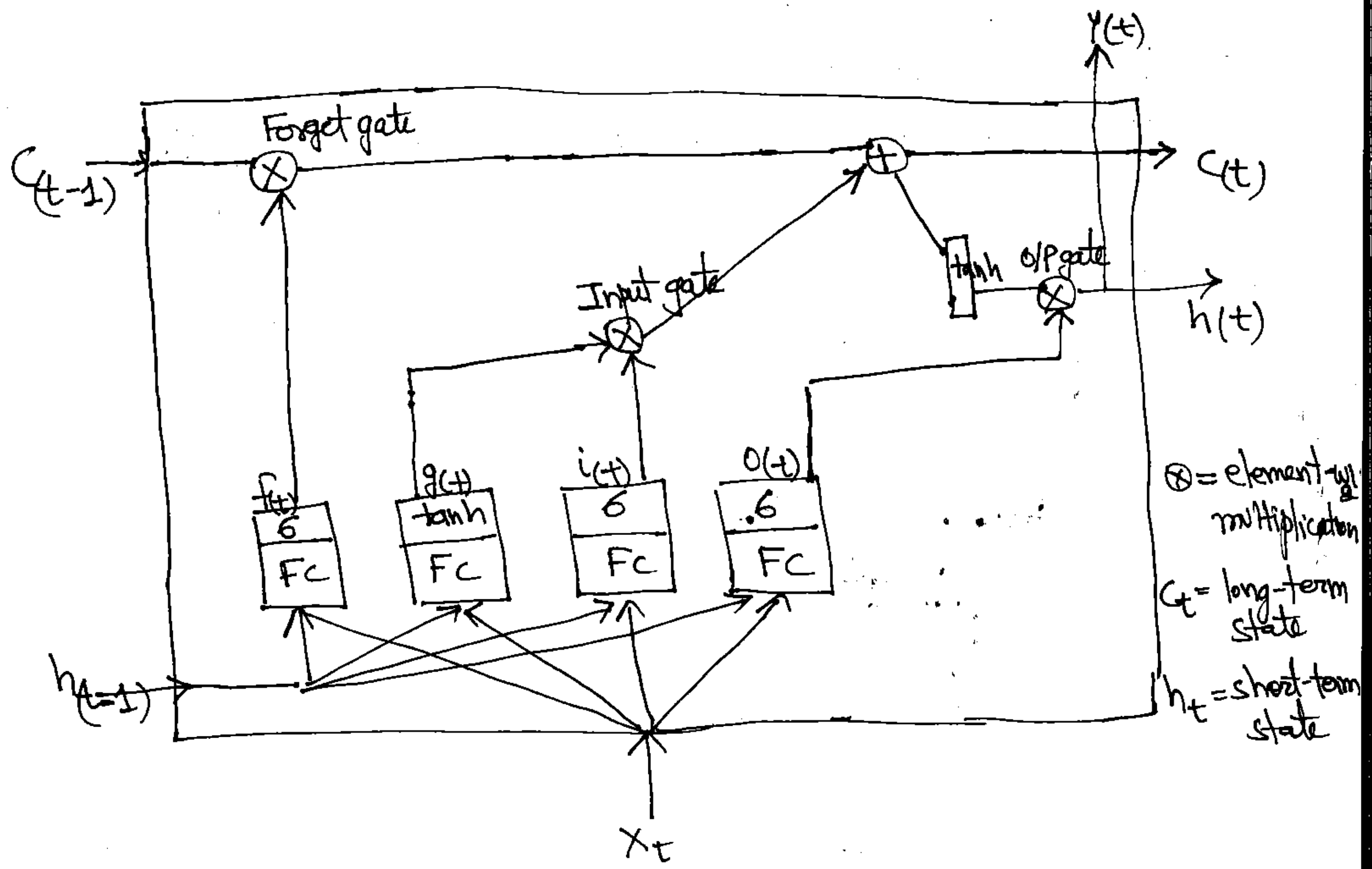
Standard RNN



LSTM

Steps in LSTM:

1) Forget - forget / get rid of irrelevant info
2) store - store relevant info from current I/P
3) Update - selectively update cell state (forget irrelevant info and add relevant info)
4) Output - output filtered version of cell state



Long-term state $C_{(t-1)}$: First goes thru forget gate, dropping irr. info then adds some info that were selected by input gate → $C_{(t)}$

Short-term state $h_{(t)}$: $C_{(t)}$ → tanh → filtered by o/p gate → $h_{(t)}$

||

$Y_{(t)}$

Cell's o/p at this time step

$x_{(t)}$ & $h_{(t-1)}$ feeds into $f_{(t)}$, $g_{(t)}$, $i_{(t)}$, $o_{(t)}$

$f_{(t)}$, $i_{(t)}$, $o_{(t)}$ are gate controllers:

3 GATES:

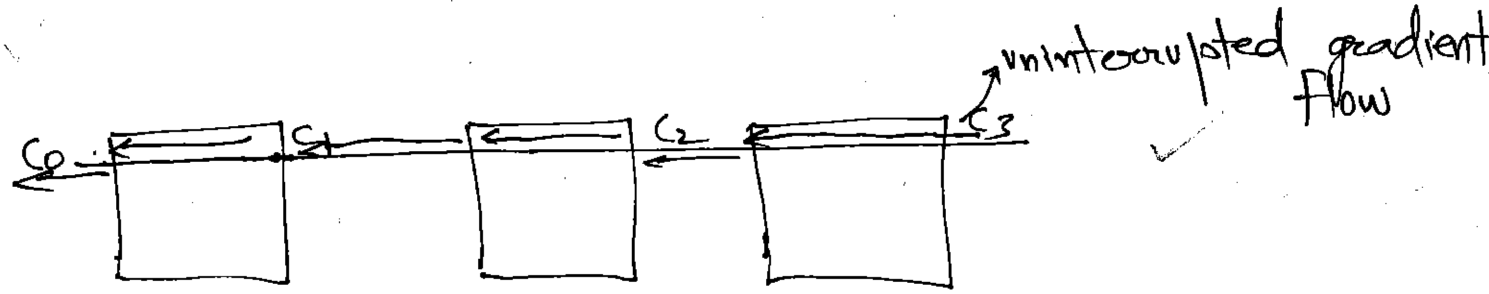
i) forget gate is controlled by $f_{(t)}$: which parts of long-term state to erase

ii) Input gate is controlled by $i_{(t)}$: which parts of $g_{(t)}$ should be added to long-term state

iii) Output gate is controlled by $o_{(t)}$: which parts of long-term state should be read and O/P at this time step both to $h_{(t)}$ & $y_{(t)}$

$g_{(t)}$: → Main layer's O/P: g_t takes x_t & h_{t-1} as I/Ps and determines what part of I/P should be retained after passing thru I/P gate and added to long-term state

* All these gating and update mechanisms actually work to create cell state C which allows for uninterrupted flow of gradient thru time



GRU: → Similar to LSTMs but less complicated structure

→ 2 gates : reset & update (instead of 3 - input, forget, output in LSTM)

→ Reset gate : LSTM's input & forget gate functionality is done by reset gate in GRU

→ Update gate : To update the cell state

GRU Vs LSTM:

1. Performance-wise (detecting dependencies over long-term) ≈ Similar

2. GRU is computationally efficient than LSTM
$\Downarrow$
Less complex structure of cell

1D CONVOLUTION LAYERS TO PROCESS SEQ

LSTMs & GRUs can process long sequences but they still have limited short-term memory and they have hard time learning long-term patterns in sequences of 100 time steps or more such as audio samples, long sentences etc.

✓ 1D convolutional layers can be used to <u>downsample</u> a very long seq while still retaining imp. info. (<u>using strides more than 1</u>)

1	2	3	4	5	6	7	8	9	10

→ seq (10 elements)

↓ 1D conv layer with filter size = 5
stride = 2

1-5	3-7	5-9	

→ seq (only 3 elements)

EMBEDDINGS

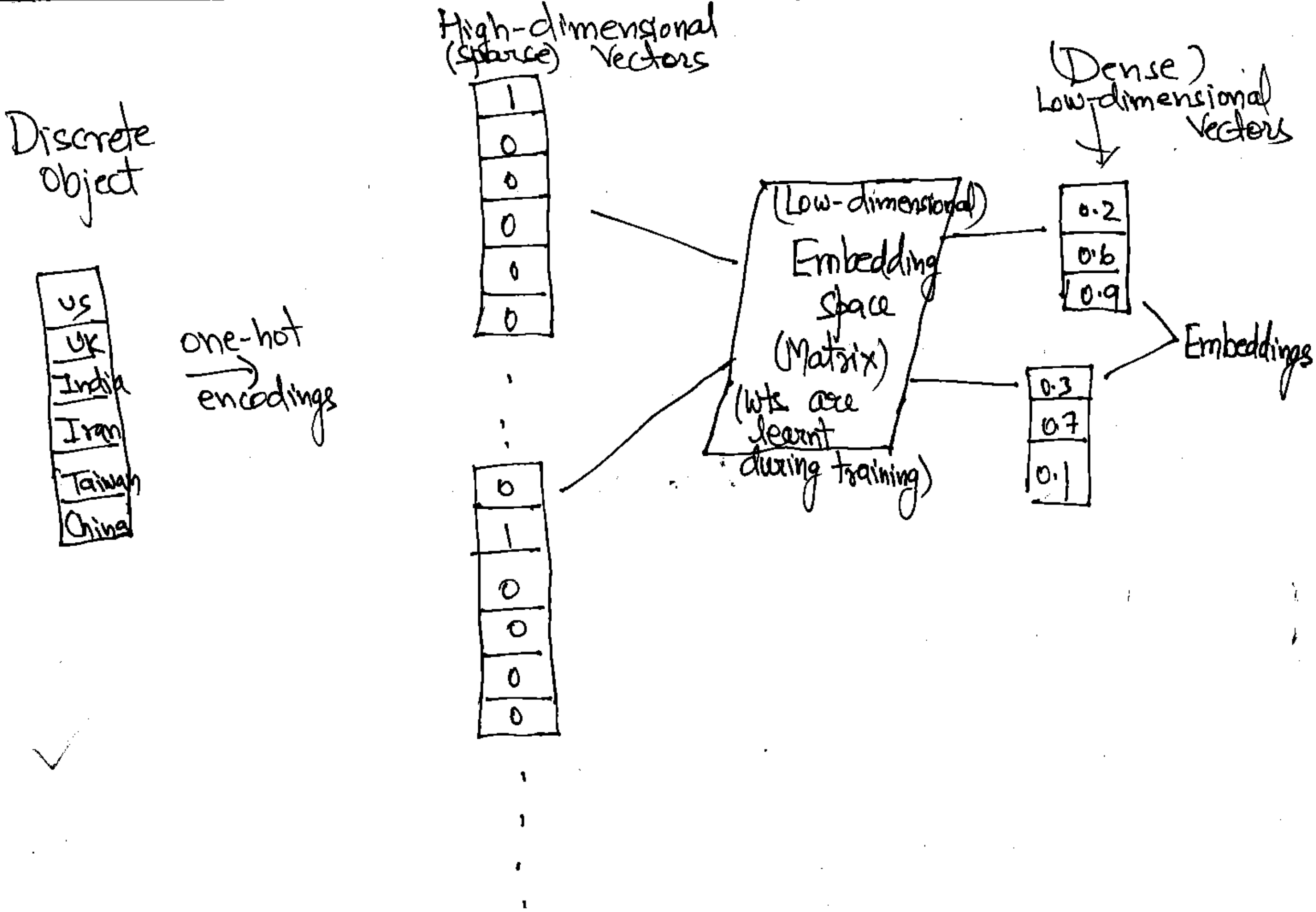
→ An embedding is a mapping from discrete objects, such as words or categorical var. value, to vectors of real numbers

→ A discrete object ~~first~~ needs to be converted to some form of vector representation (e.g. one hot encoding etc.) before embedding can be applied;

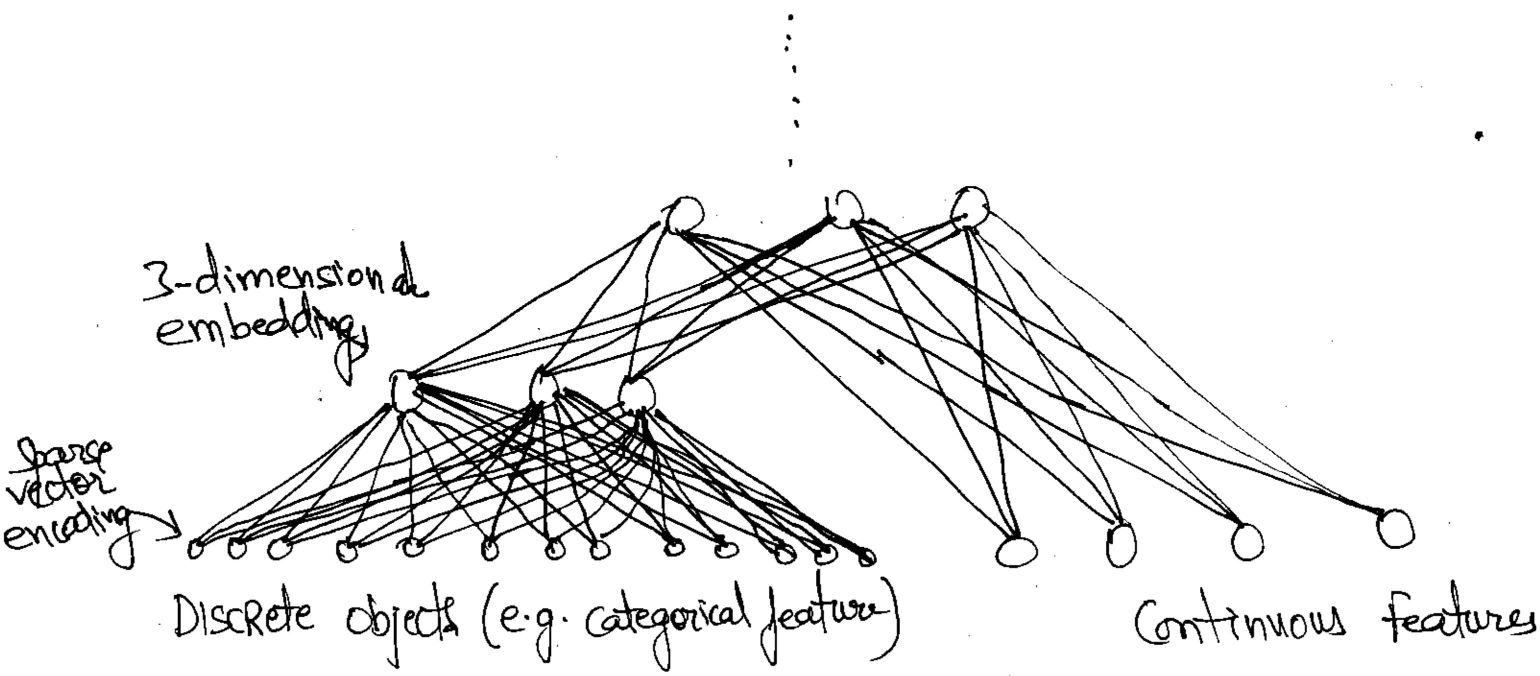
→ Converting a discrete object first to some vector representation ~~typically~~ results in sparse vectors (e.g. one-hot encoding) of high-dimensionality. Now when embedding space is applied to these high-dimensional sparse vectors, it results in low-dimensional dense vectors called as embeddings

→ Ideally, an embedding captures some of the semantics of the input by placing semantically similar inputs close together in the embedding space. (distance b/w vectors and direction of vectors determine this semantic similarity)

→ An embedding space is a low-dimensional space that translates high-dimensional vectors into low-dimensional vectors.



TRAINING AN EMBEDDING AS PART OF LARGER MODEL



Embeddings can be

1. Pre-trained : e.g. the model has learnt the embedding space using some corpus (domain)

   These are domain-specific (also called as unsupervised embedding)

   Now you can take this model, ingest your discrete objects/words to this model and model outputs the embeddings.
   Example: word2Vec is trained on Google News dataset.

2. Learnt as part of training for your task (supervised embedding)

Note: The concepts of word2Vec can be used to learn embeddings as part of your task too.

Note: If you require any kind of distance calculation for discrete objects, embeddings is the way to go. Example, if some form of KNN is needed on categorical values $\Rightarrow$ embeddings first then find distance

WORD2VEC

- Combination of two tech
  - CBOW (continuous bag of words)
  - skip-gram

- Both of these two techniques use "shallow" neural n/w i.e. one hidden layer which maps an input word to a target word.

- How I/P word and o/p word is defined is determined by these two techniques

Example: Hey, this is great

CBOW: context window = 2

I/P	O/P	
Hey	this	← Datapoint 1
Hey	is	← Datapoint 2
this	Hey	
this	is	
this	great	
is	Hey	
is	this	
is	great	
great	this	
great	this	

i.e. I/P : a word (repeated)
O/P : take context window words (before & after, if they exist)

Skip-gram: (Multiple O/Ps)

Context window = 1

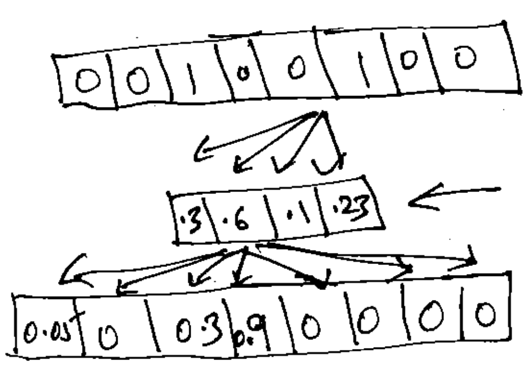
I/P	O/P	O/P	
Hey	this	<padding>	← Datapoint 1
this	Hey	is	← Datapoint 2
is	this	great	⋮
great	is	<padding>	

I/P: a word

O/Ps: All words in context window (before & after, if they exist)

→ Since a shallow network is used, the hidden layer's wt is the embedding space

And after the model is trained, the I/P words go thru this model and the O/P of hidden layer is the embedding for the I/P word



← I/P

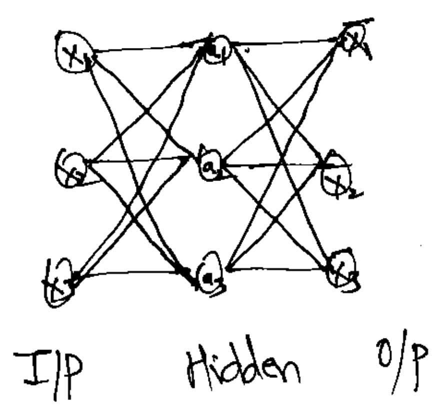
← Hidden layer (Embedding space wts.)

← O/P

AUTO ENCODERS (Unsupervised)

- A neural n/w architecture that imposes a bottleneck/constraint in the n/w which forces a $\underbrace{\text{compressed / latent representation}}_{\text{codings}}$ of the original I/P.

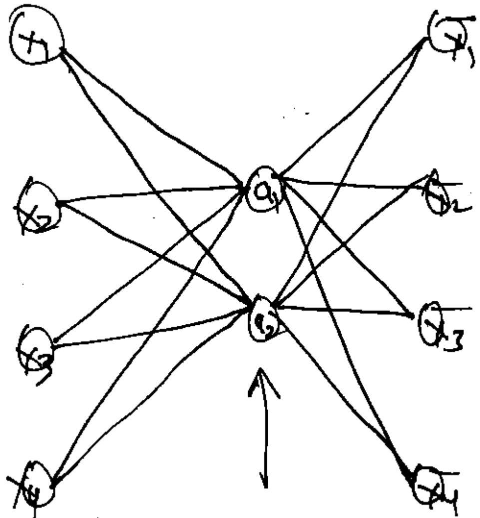
- The bottleneck/constraint is a key attribute of our n/w design; without the presence of information bottleneck our n/w could easily learn to simply memorize the input values by passing these values along thru the n/w. E.g.



No bottleneck (i.e. Hidden layer size = I/P layer size = O/P layer size)

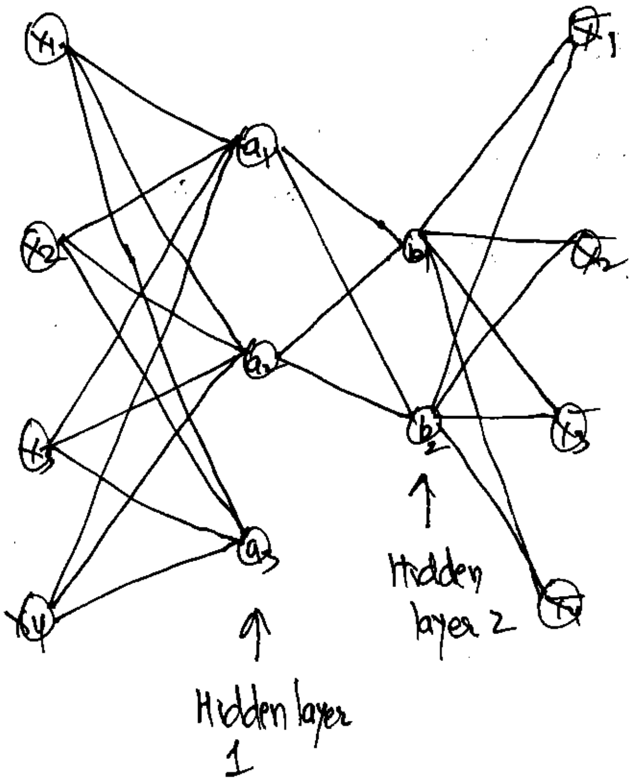
- An autoencoder looks at the input, converts them to an efficient latent representation and then spits out something that looks very close to the inputs.

- An autoencoder is always composed of two parts:
  i) encoder: that converts I/P to latent rep
  ii) decoder: that converts latent rep. to the O/Ps (= I/Ps)



← Undercomplete Autoencoder
(has no explicit regularization term)

bottleneck (Hidden layer size $\underset{<}{\neq}$ I/P or O/P layer size)



← Stacked / Deep Autoencoder
(Multiple hidden layers whose size is less than I/P or O/P layer)
⇓
Adding more hidden layers helps the autoencoder to learn more complex codings
(has no explicit regularization term in theory)

Objective fn of Autoencoders: Minimize reconstruction error + Regularization

typically, MSE or Cross-entropy ←

$$L(x, \bar{x}) + \text{regularizer}$$

↓ to prevent copying of I/P to O/P

When
i) Regularizer = L1 or KL divergence → Sparse Auto-encoders (avoiding overfitting)
ii) Regularizer = Dropout layer (or Gaussian noise) → Denoising Auto-encoders

CODE (KERAS)

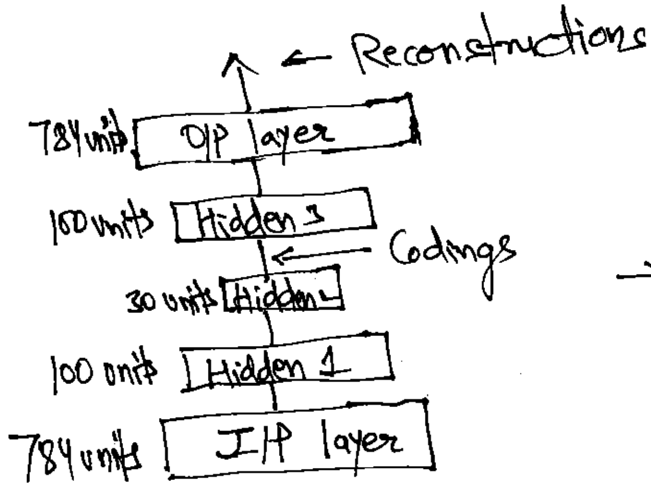
```
encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])

autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

I/P & O/P are both X_train

```
history = autoencoder.fit(X_train, X_train, epochs=20)
Codings = encoder.predict(X_train)
```

latent rep^r of I/P after training

TYPICAL AUTOENCODER ARCHITECTURE:



→ Symmetry around hidden 2 layer

APPLICATIONS :

i) Dim. Reduction / Feature Extraction
(O/P of encoder after training)

ii) Denoising data

iii) Generative Models: (VAEs)

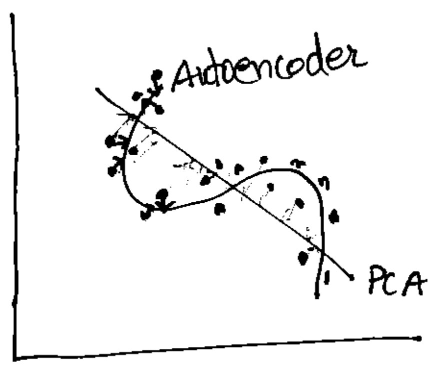
VAEs learn the parameters of prob. dist. modelling the I/P data instead of learning an arbitrary function in case of vanilla autoencoders. By sampling points from this distribution, we can use VAE as a generative model.

PCA Vs AUTOENCODERS

Similarity: Both attempt to discover lower dimensional rep of I/Ps (dimensionality reduction)

Differences:

PCA	Autoencoders
i) Linear	Non-linear



PCA	Autoencoders
Discovers lower dimensional hyperplane which describes the original data	Capable of discovering non-linear manifolds (a manifold is a continuous non-intersecting surface

PCA = AUTOENCODER

i) Activation in autoencoders = linear
(recall bcz of non-linear act. fn, neural n/w can learn non-linear fns)

ii) Loss fn = MSE (mean squared error)

SPARSE AUTOENCODERS

- Another type of constraint that can be used to create the bottleneck → sparsity constraint
- In particular, we add a penalty term to the cost fn so that only a fraction of nodes become active
- This forces the autoencoder to represent each input as a comb. of smaller number of nodes and demands it to discover interesting structure in data.
- Penalty term
  - L1
  - KL divergence
- This method works even if code size (size of last layer of encoder) is large since only a subset of nodes will be active at any time

DENOISING AUTOENCODERS

Another type of constraint that can be used to force autoencoder to learn useful features → adding random noise to its inputs and making it recover the original noise-free data (during training phase)