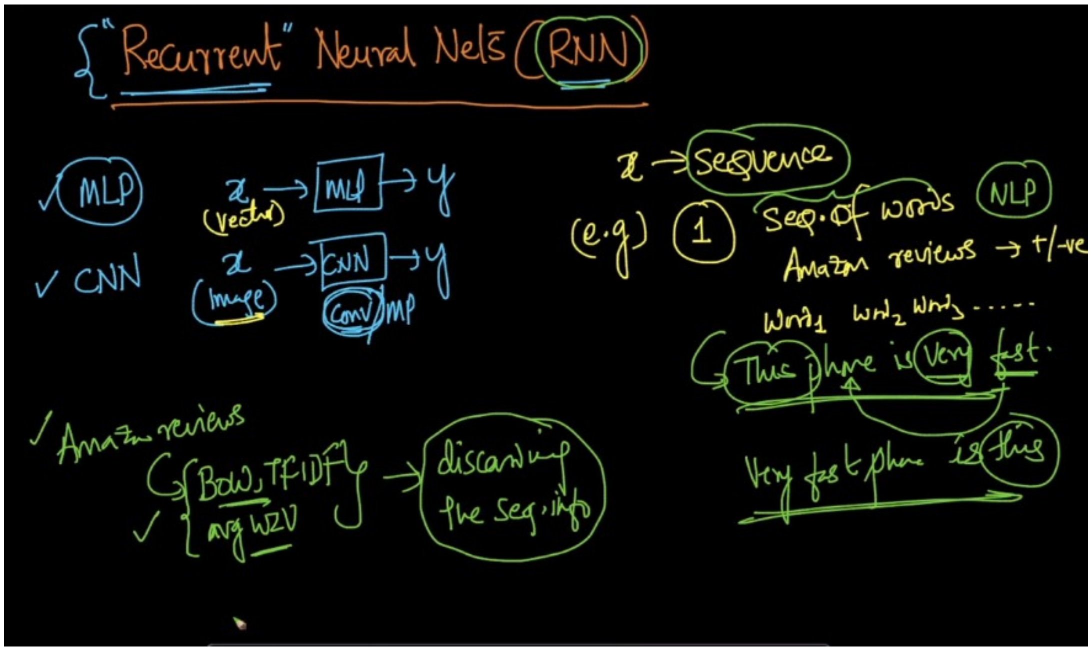
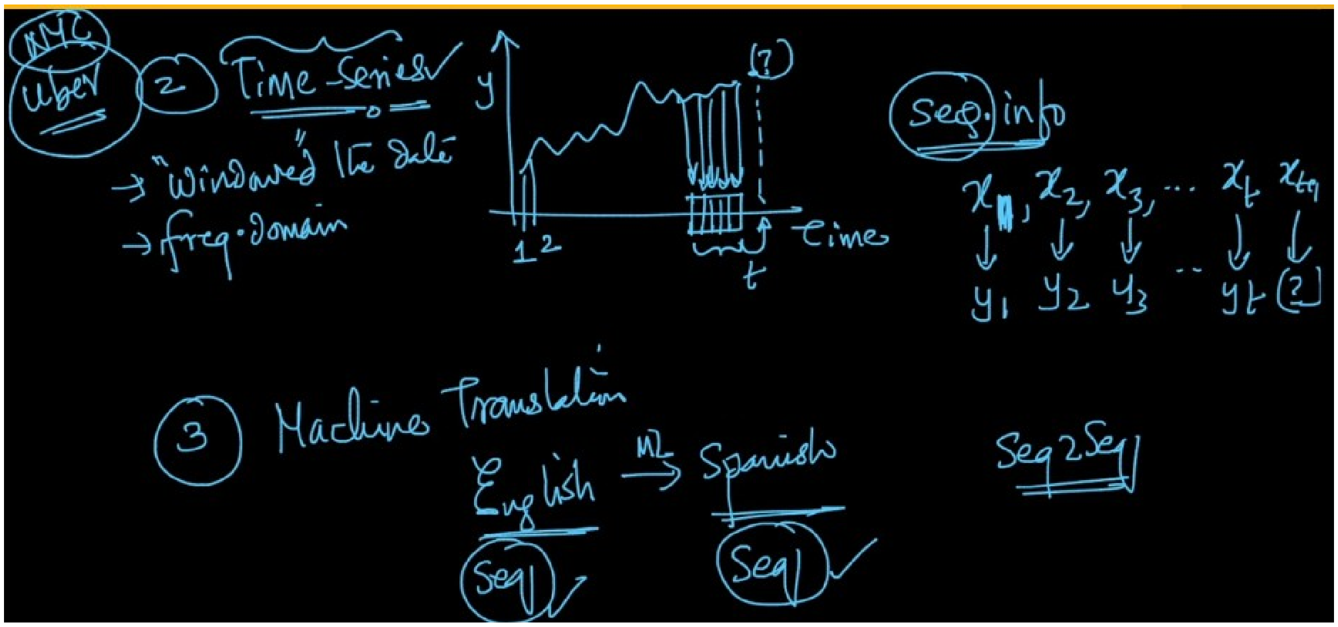
Recurrent Neural networks

Sequence matters for making a meaningful sentence.
Types of data various NN can handle:

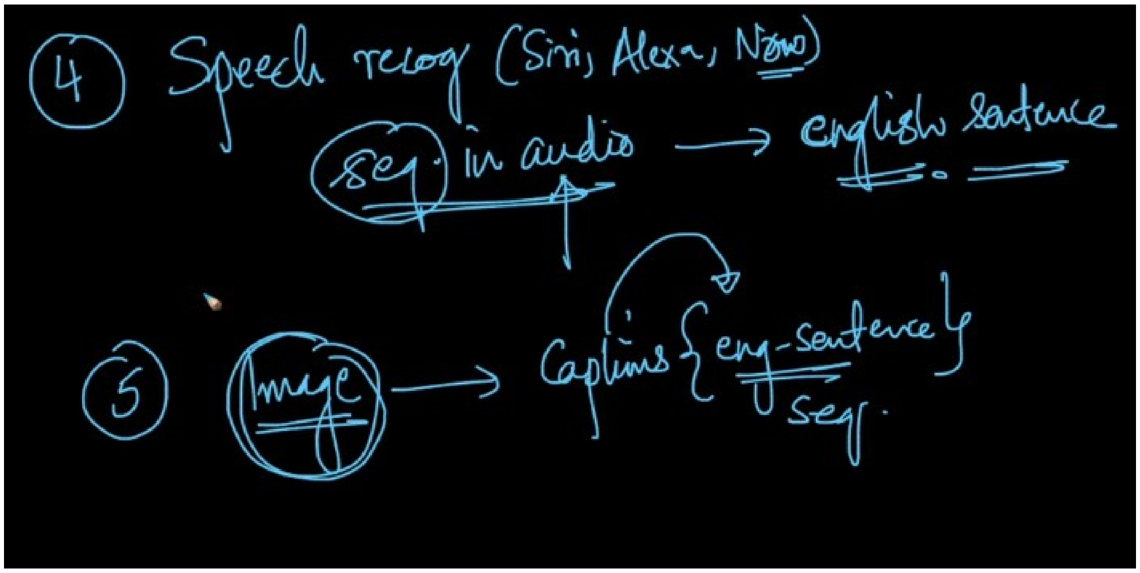
In BOW,TFIDF,Aavg w2v ignores the sequence information. This is actually important to make most out of the sequence of words.



In case of time series data, we windowed the data and we try to predict the next point. Time series is also a sequence information.
We partially used the sequence information using the windowing technique.
Machine translation is the sequence information, sequence input and sequence output.



Speech recognition and Image captions:



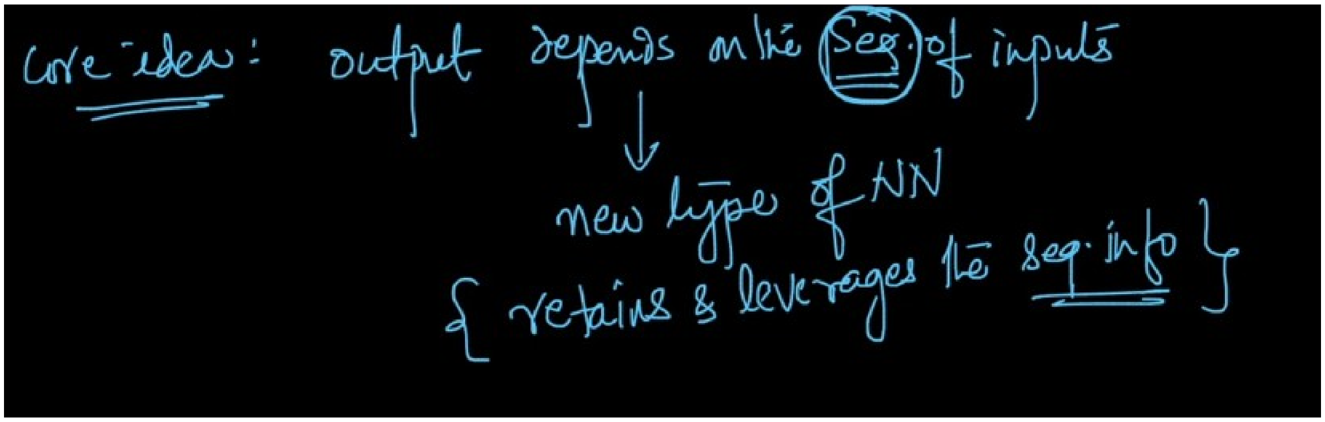
4 Speech recog (Siri, Alexa, Now)

seq. in audio → english sentence

5 Image → Captions { eng-sentence } seq.

Image captioning can be used in google image search, google text search.

The importance of RNNs:



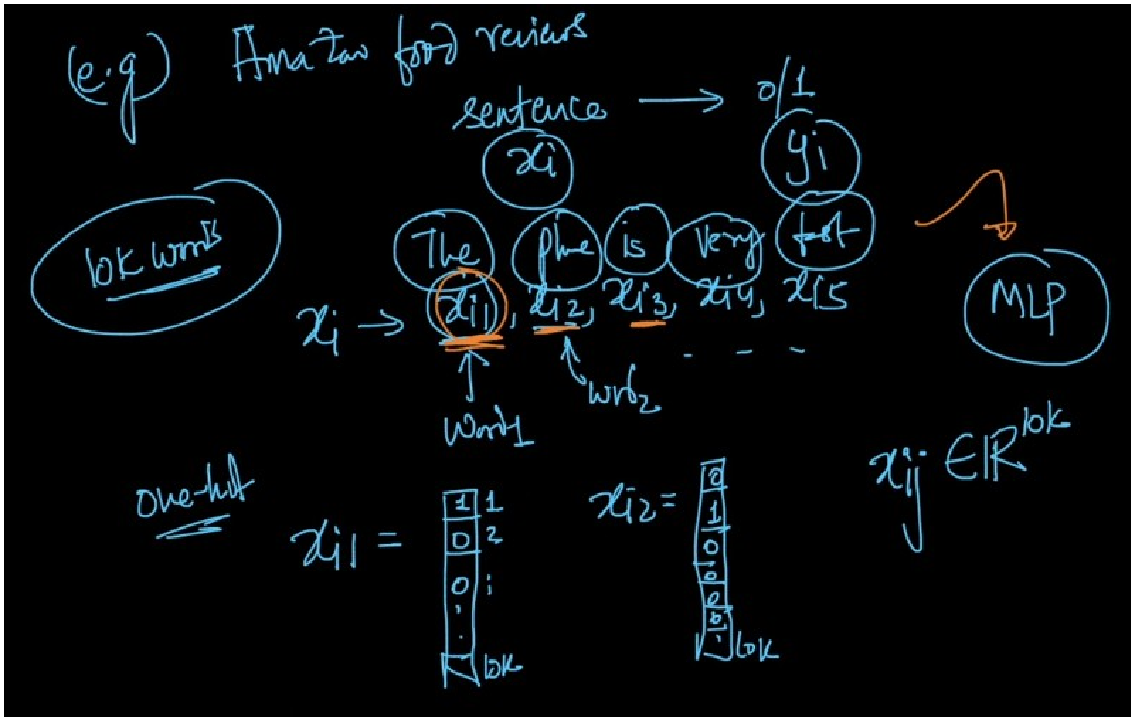
Core idea: output depends on the seq. of inputs

↓

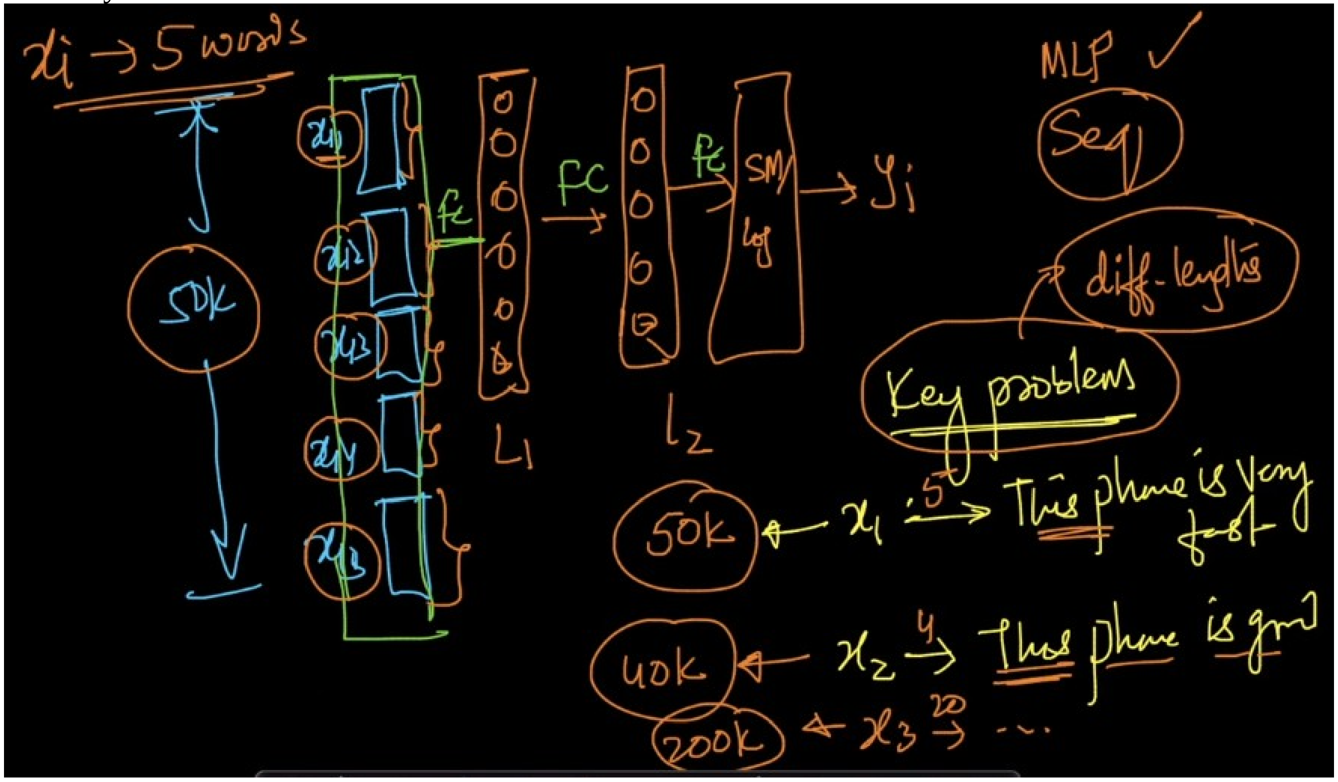
new type of NN

{ retains & leverages the seq. info }

Representing the words in the vector.

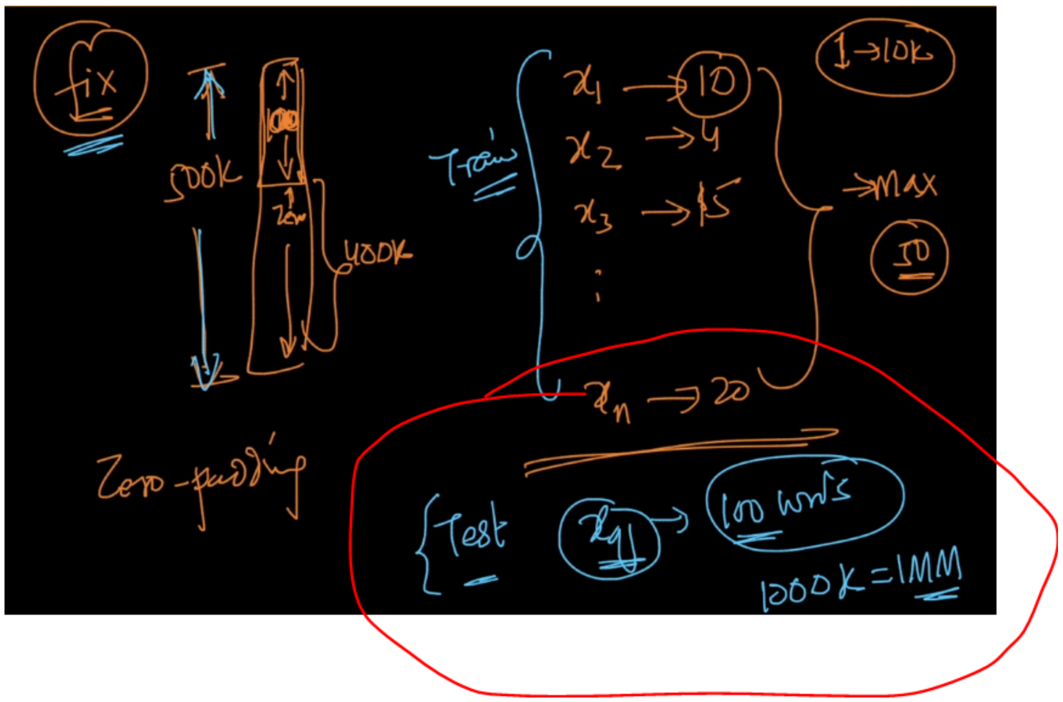


Why not we train MLP ?
Process of training MLP on the sequence data. Every word is one-hot encoded.
Xi has 5 words and there are 10k unique words in the data set. The number of words in the sentence will vary a lot.

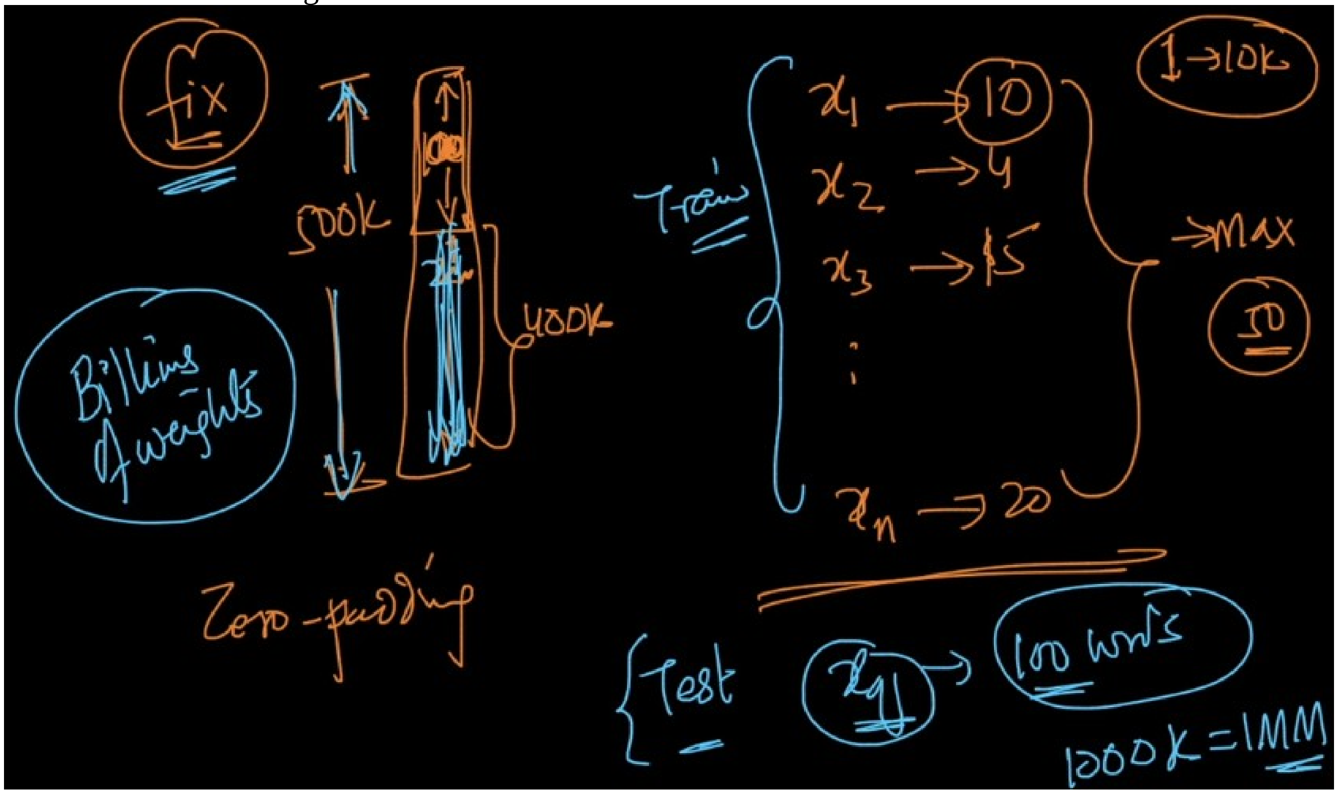


We can fix the variable length of the sentences, We will pad the sentence with zeros by taking the maximum length of the sentence.

Even with the fixed length, there will bw problem in case of the text data if we have the large dimensions.



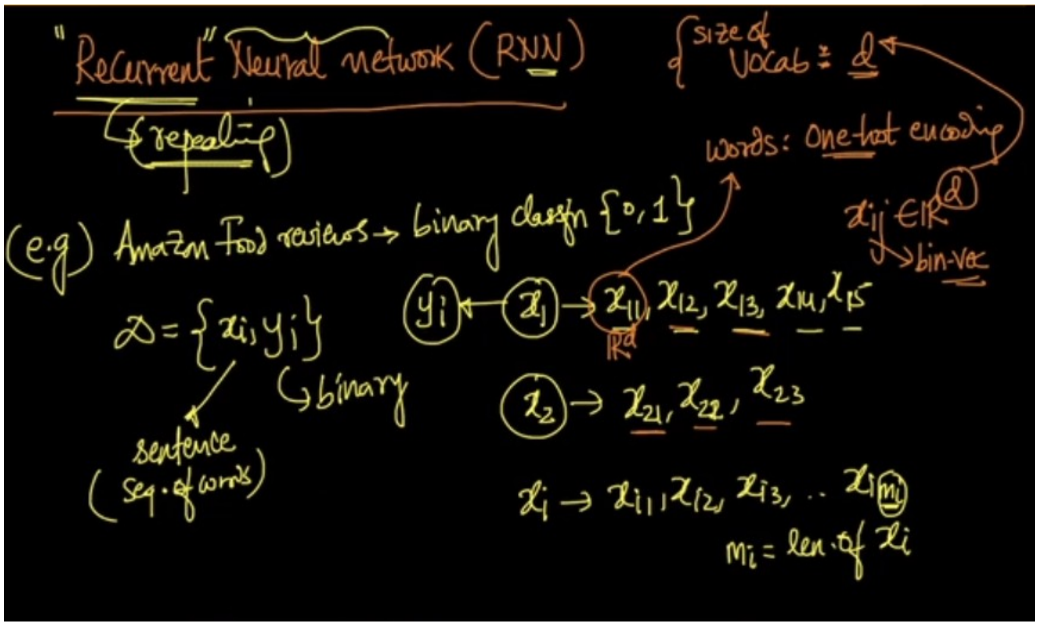
The input size will become massive. We are building un necessarily large vectors.
There will be more weights.



1We can use all the word vectors for RNN’s

Amazon fine food reviews is the binary classification task (0, 1).

We are representing the words in the one hot encoding, let say the size of vocab has the d words and each word belongs to a sentence.


"Recurrent" Neural network (RNN)
(repeating)
size of Vocab = d
words: One-hot encoding
(e.g) Amazon Food reviews → binary classfn {0, 1}
$x_{ij} \in \mathbb{R}^d$
bin-vec
$D = \{x_i, y_i\}$
binary
sentence (seq. of words)
$y_i \leftarrow x_1 \rightarrow x_{11}, x_{12}, x_{13}, x_{14}, x_{15}$
$\mathbb{R}^d$
$x_2 \rightarrow x_{21}, x_{22}, x_{23}$
$x_i \rightarrow x_{i1}, x_{i2}, x_{i3}, \ldots x_{im_i}$
m_i = len. of x_i

Task: To classify the seq of words into two classes.
In the first stage we take the first word and then second word and so on.

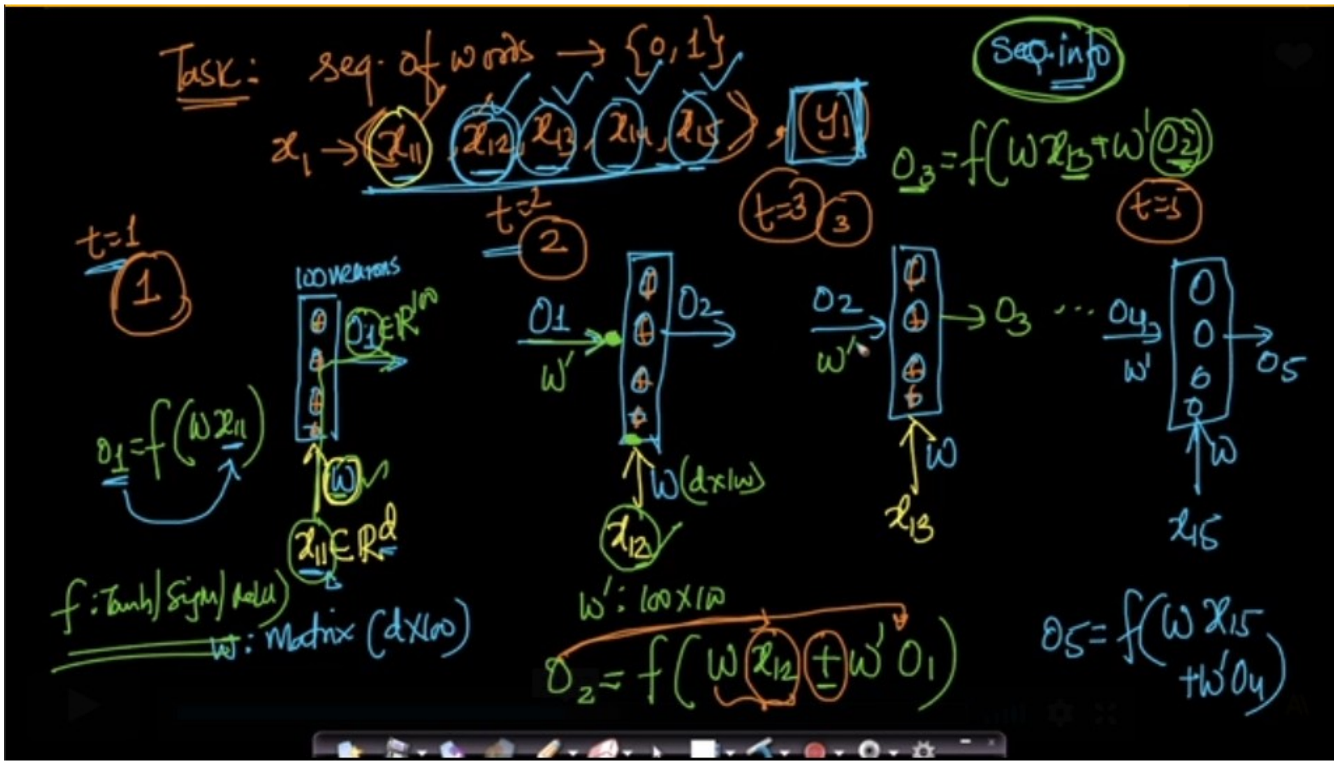
The same weight for the first word hidden layer is used for the next word and the out put of the first word.

We want to keep the seq information. So, we use the output of the first layer and pass it to the second layer with also a next word in the sequence.

Recurrent means repeating.

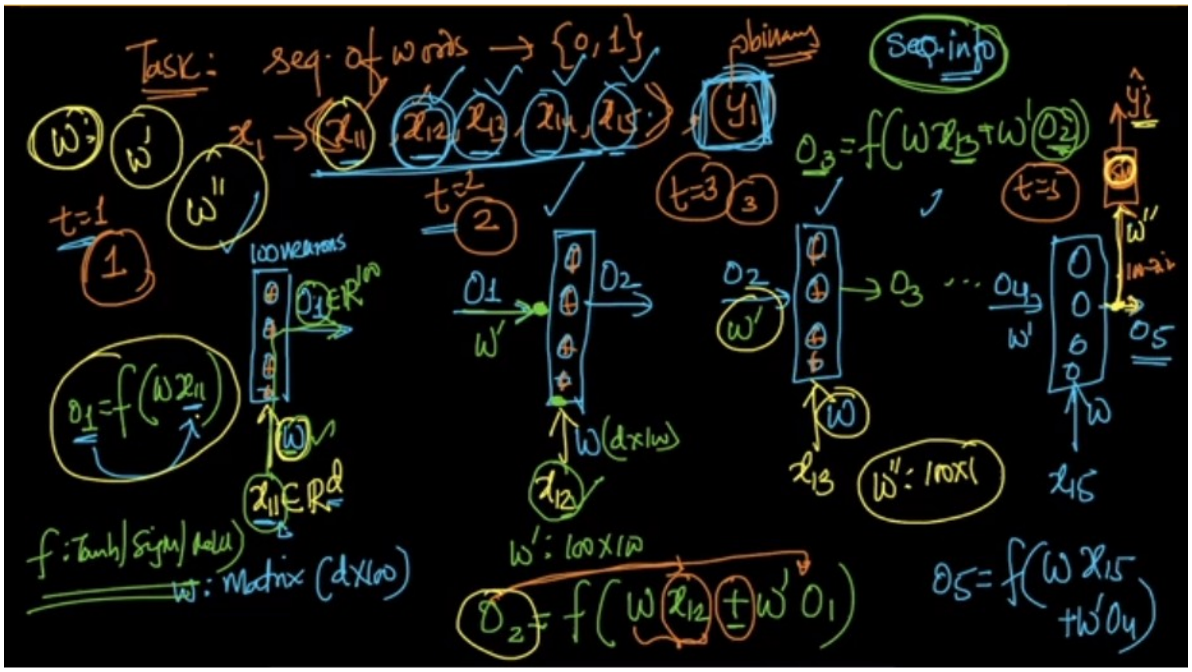
Explanation of recurrent NN:
Each word in the sentence is represented as one hot encoding vectors. They are binary vectors.

Training the sentence:

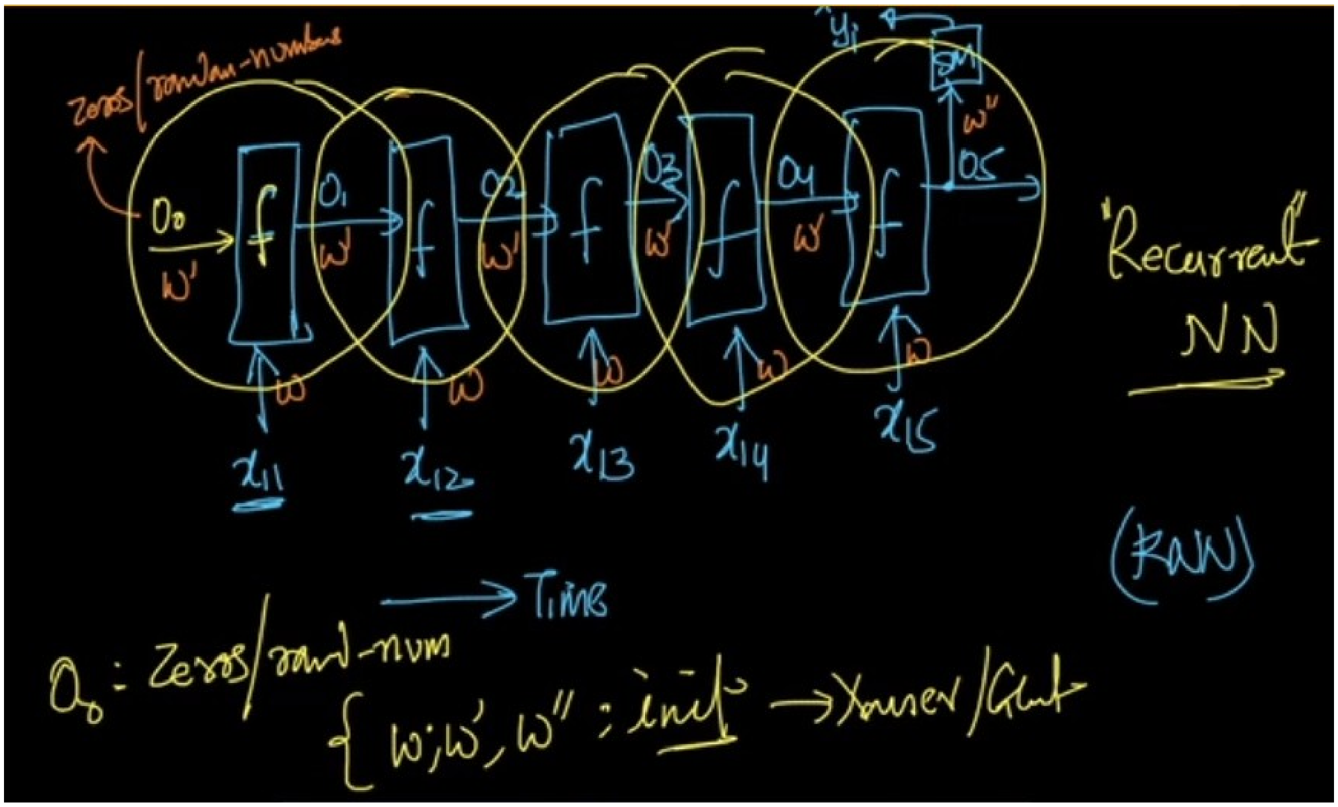


Determining the label yi of the sequence of words:
We take the output of the last layer and we put a softmax layer to give the label yi.

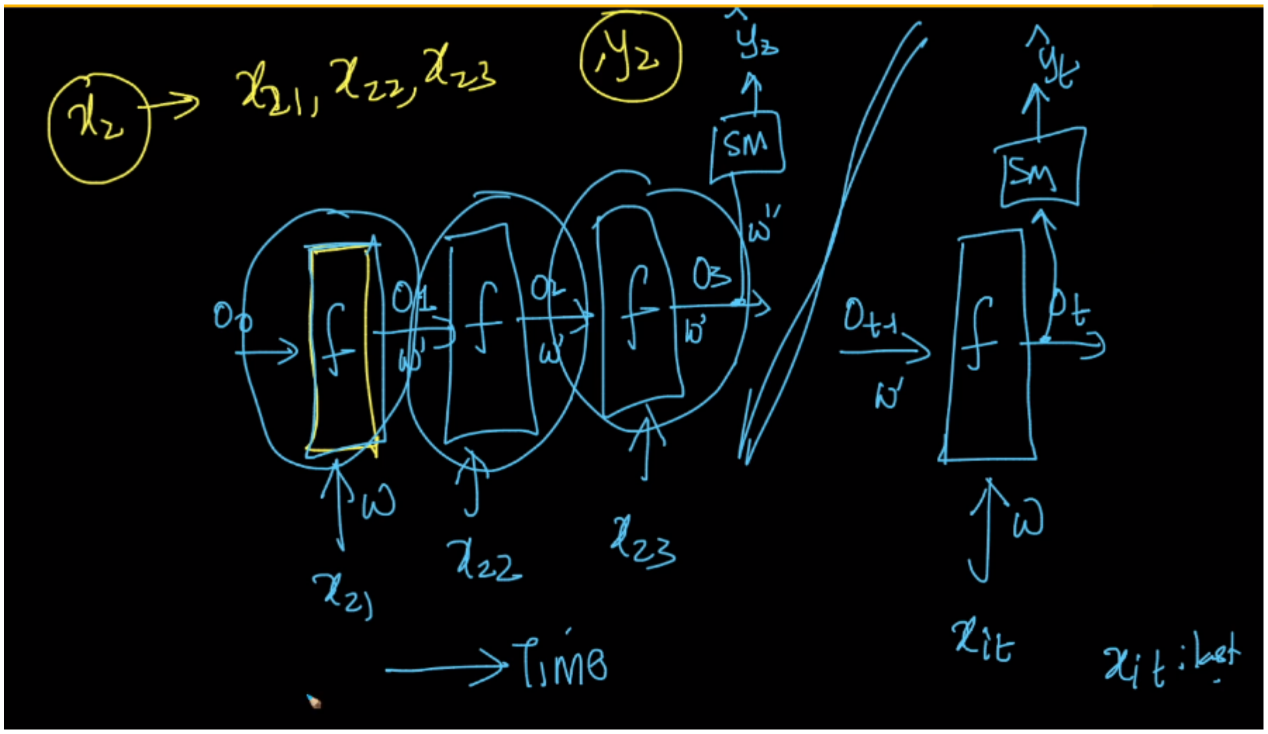
We will have the weight vector at the end, this is used for determining the label of the sequence.



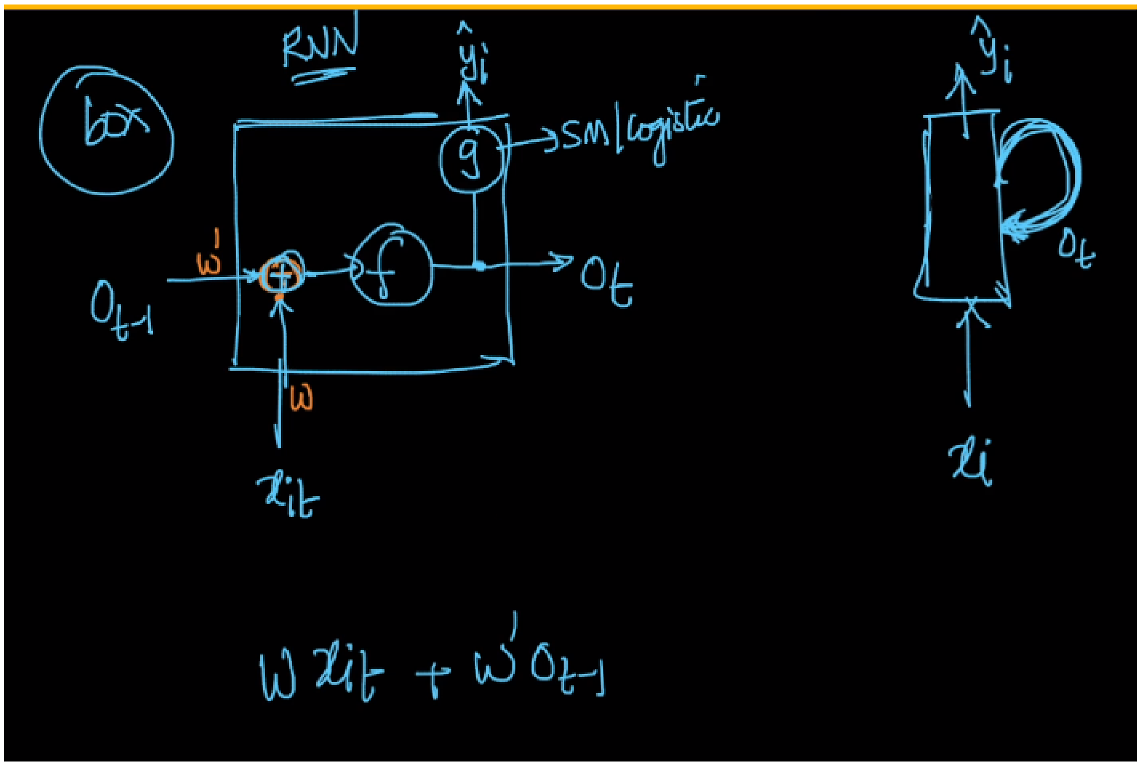
Simplified notation of RNN:



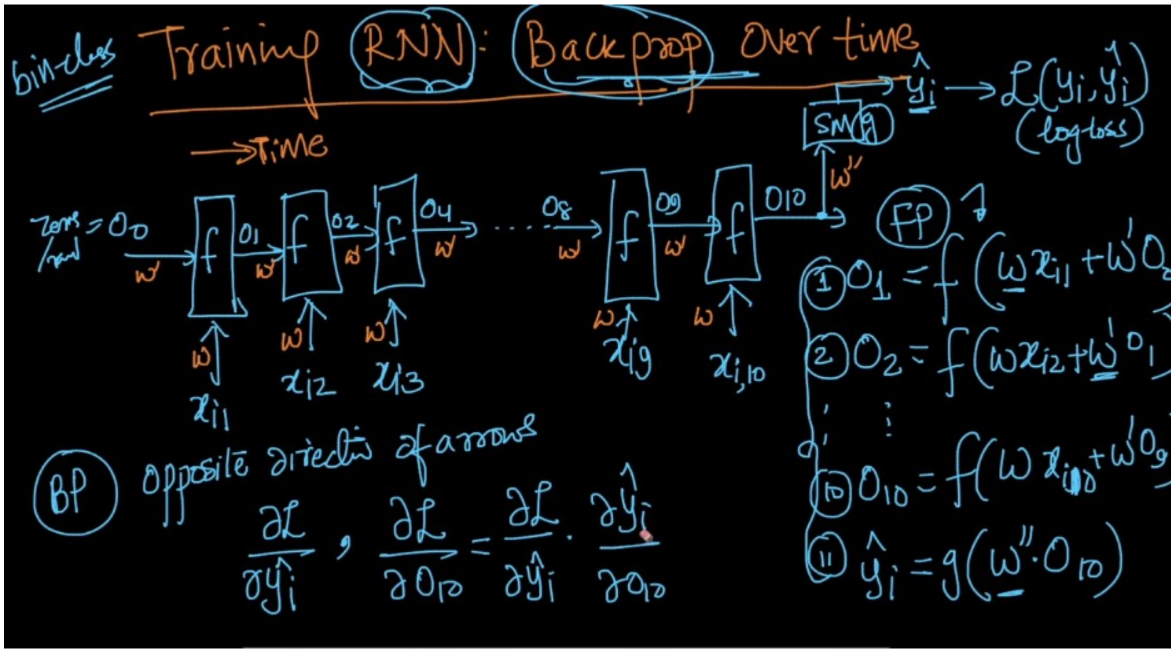
Training with the next sequence of words:

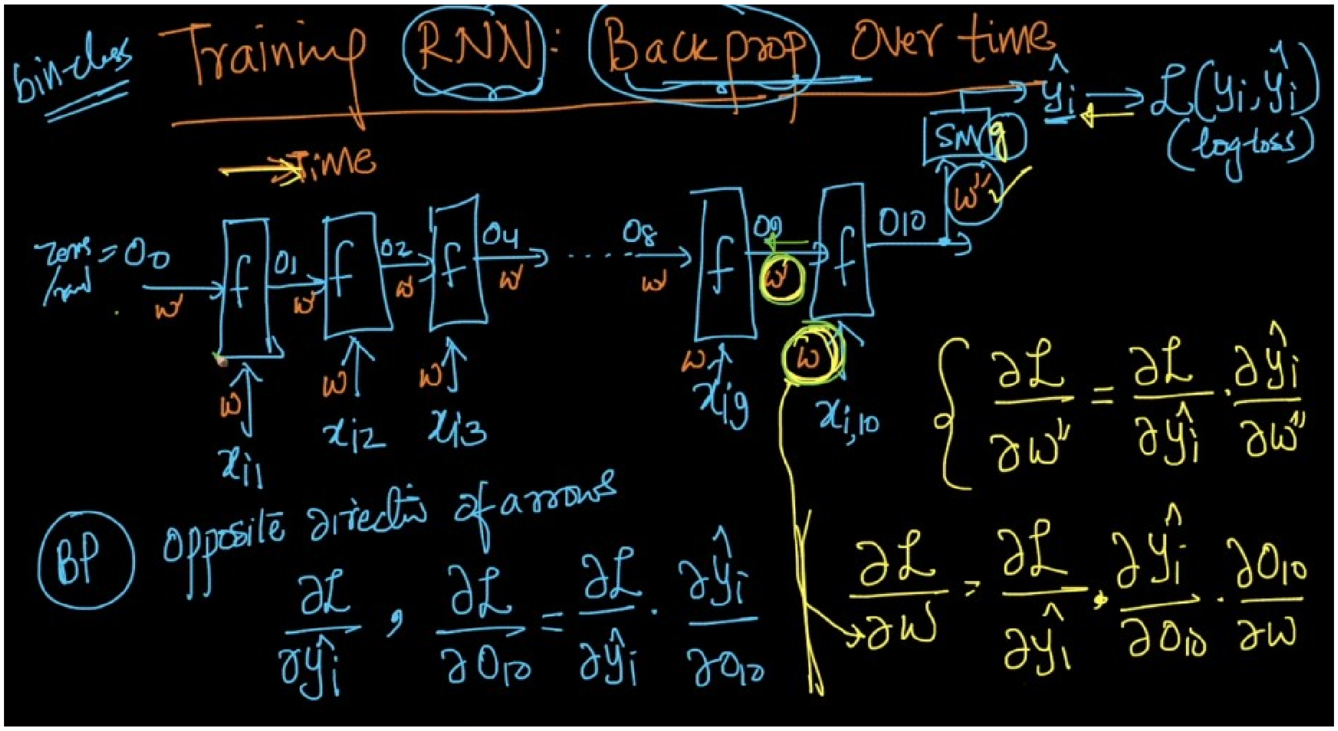


The other way of representing the RNN's:



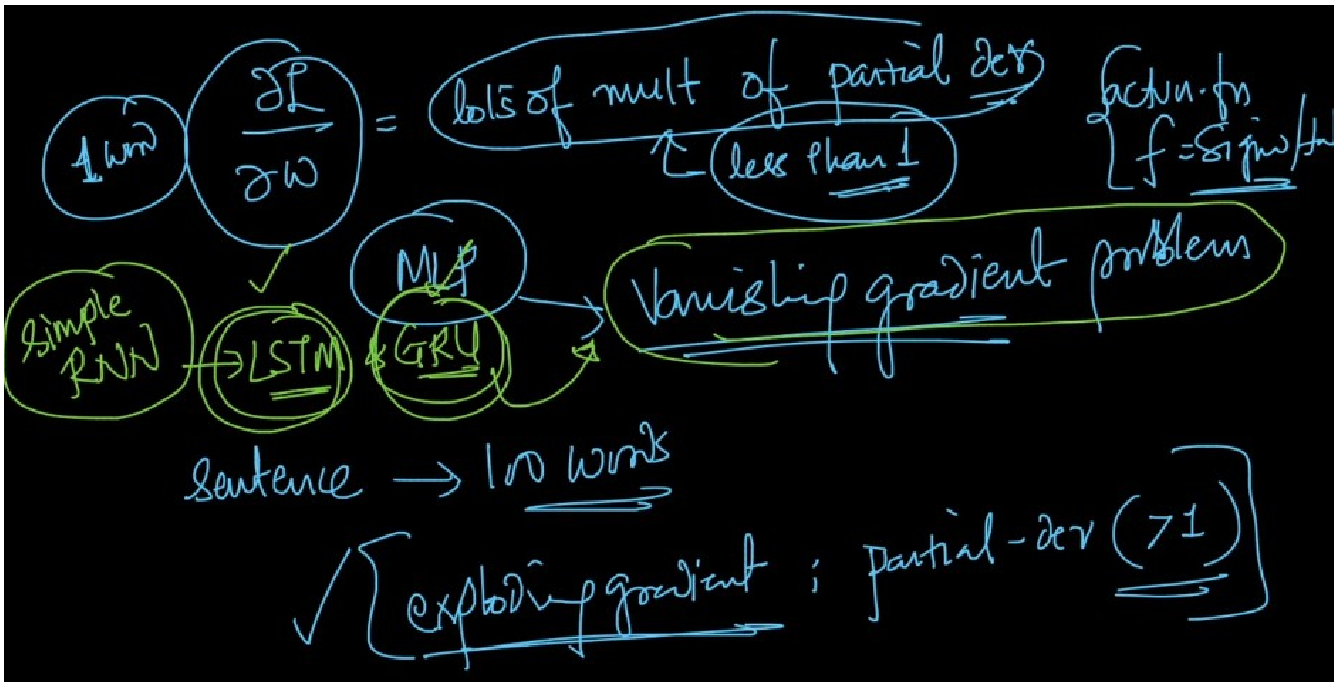
Training RNN’s using back propagation:
This is a modification over the normal back propagation. Forward prop use the direction of arrows, to make training.
Backward prop of the opposite of the arrows.


bin-class
Training RNN: Back prop Over time
Time
zero /rand = O_0
w'
f
O_1 O_2 O_4 O_8 O_9 O_{10}
x_{i1} x_{i2} x_{i3} x_{i9} $x_{i,10}$
SM/g
w''
$\hat{y}_i \rightarrow \mathcal{L}(y_i, \hat{y}_i)$ (log-loss)
FP
① $O_1 = f(w x_{i1} + w' O_0$
② $O_2 = f(w x_{i2} + w' O_1$
⑩ $O_{10} = f(w x_{i10} + w' O_9$
⑪ $\hat{y}_i = g(w'' \cdot O_{10})$
BP opposite direction of arrows
$\frac{\partial L}{\partial \hat{y}_i}$, $\frac{\partial L}{\partial O_{10}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial O_{10}}$


bin-class
Training RNN: Back prop Over time
Time
zero /rand = O_0
O_1 O_2 O_4 O_8 O_9 O_{10}
x_{i1} x_{i2} x_{i3} x_{i9} $x_{i,10}$
SM/g
w''
$\hat{y}_i \rightarrow \mathcal{L}(y_i, \hat{y}_i)$ (log-loss)
$\frac{\partial L}{\partial w''} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w''}$
BP opposite direction of arrows
$\frac{\partial L}{\partial \hat{y}_i}$, $\frac{\partial L}{\partial O_{10}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial O_{10}}$
$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial O_{10}} \cdot \frac{\partial O_{10}}{\partial w}$

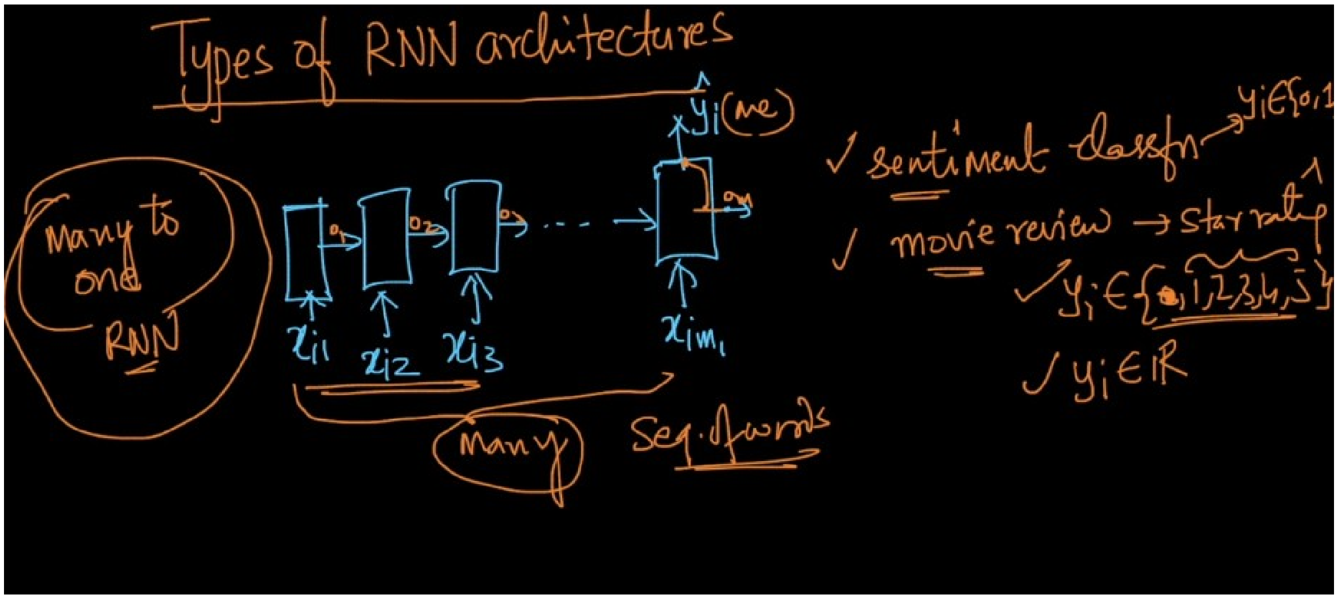
This is over time and we have the three derivatives w, w1, w11.

Due to the Vanishing gradient problems the LSTMs and GRUs are used to avoid this.

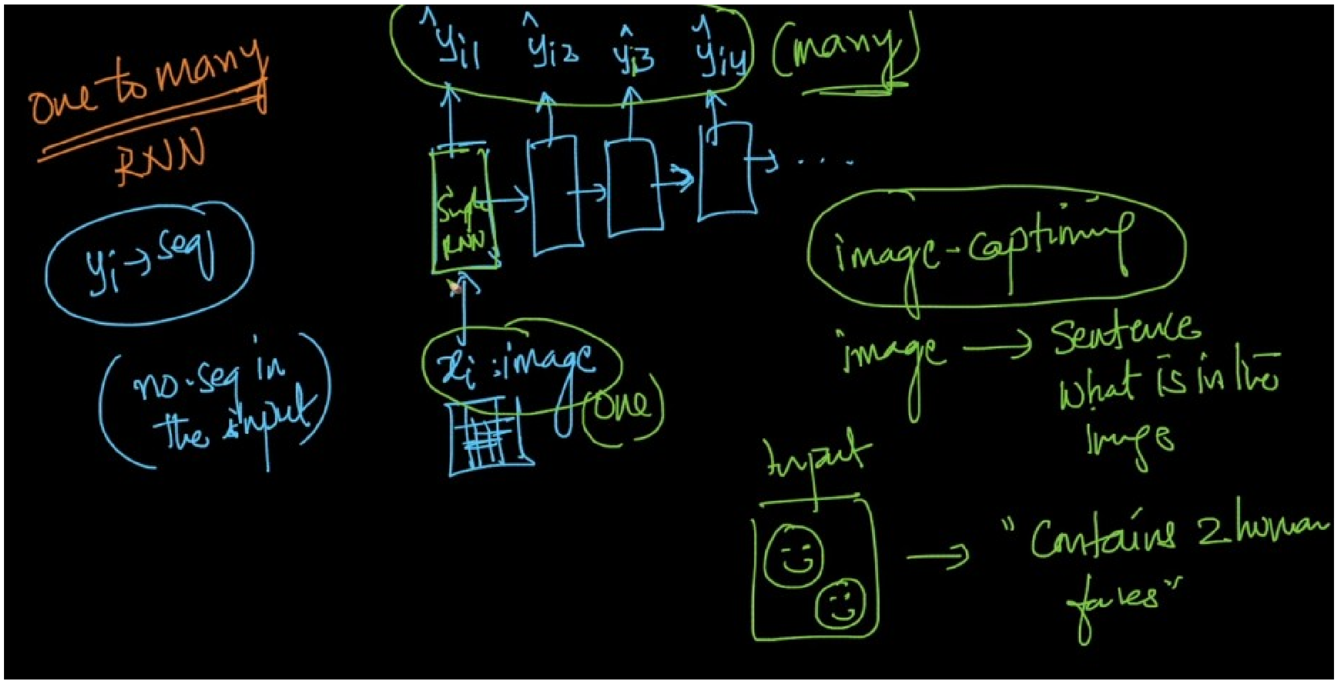

1 wm
∂L/∂w = lots of mult of partial ders
less than 1
actvn. fn
f = sigmoid/tanh
simple RNN → LSTM & GRU
MLP
Vanishing gradient problems
sentence → 100 words
exploding gradient ; partial-der (>1)

Types of RNN’s:

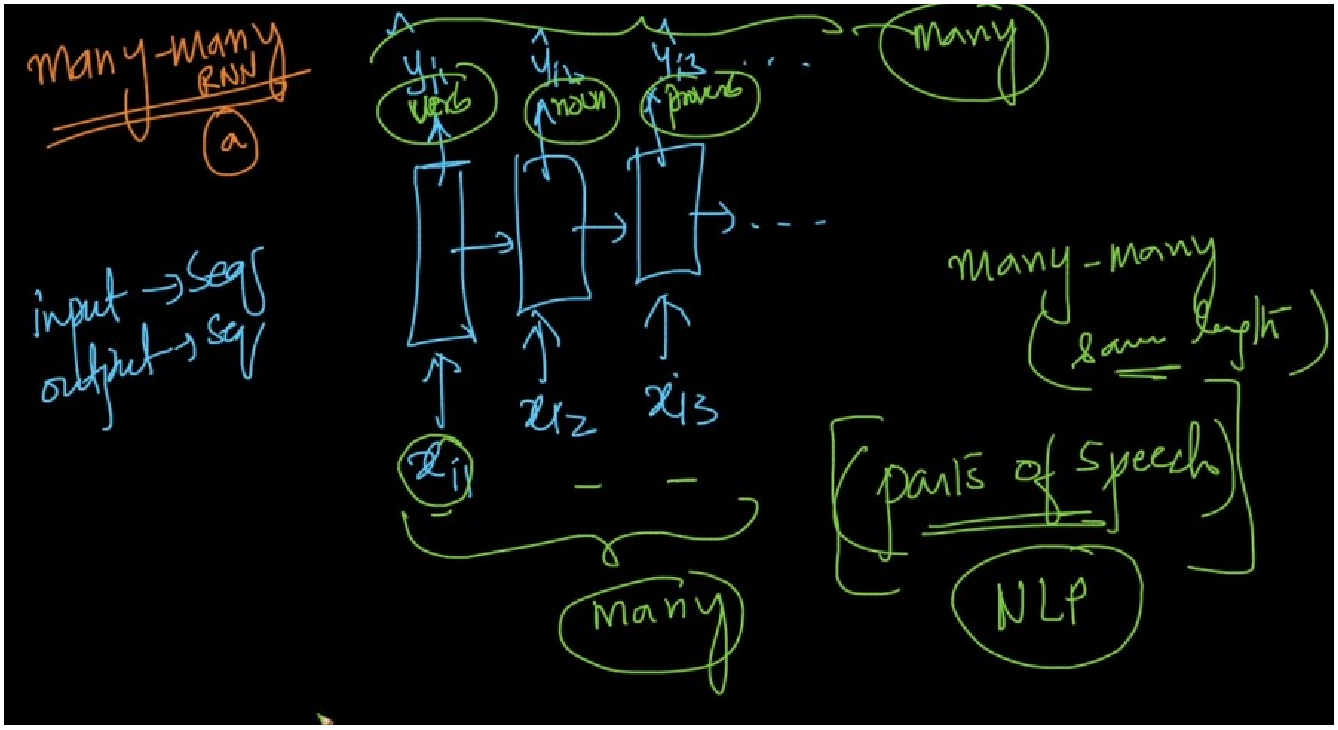
Many – one Rnns:



One to many:

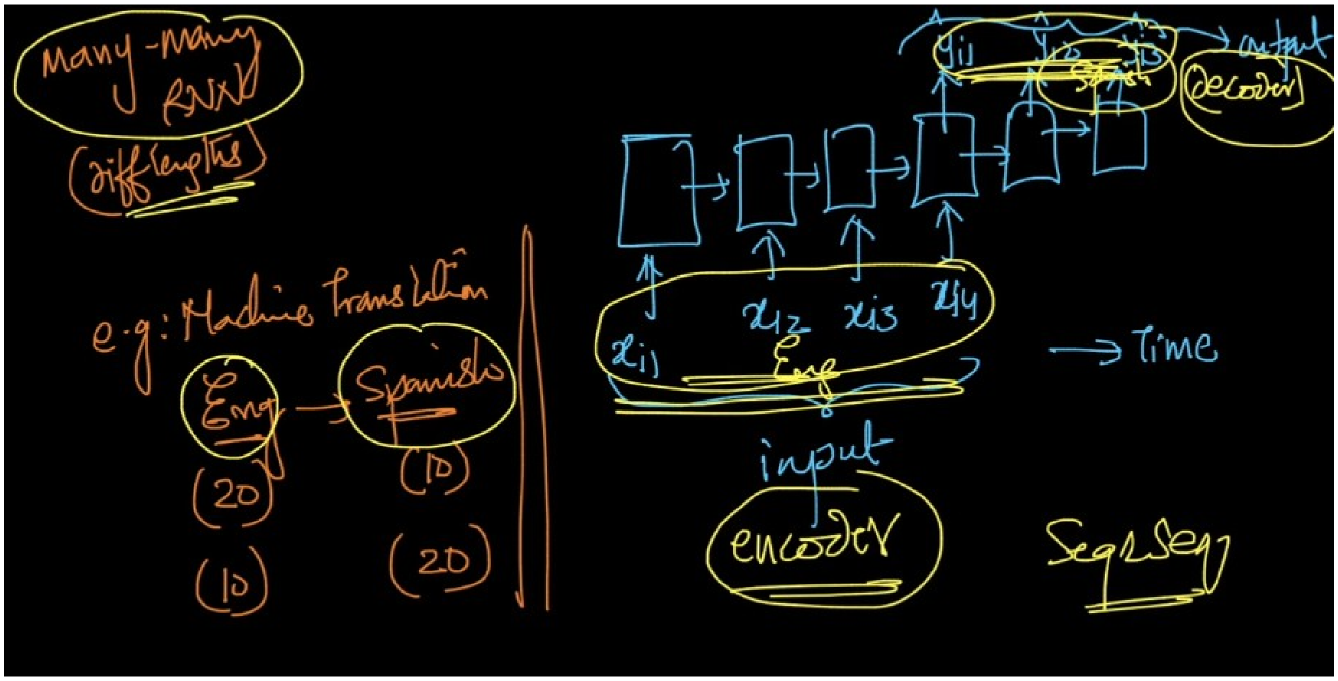


Many to Many:
Same length.
For determining the parts of speech of a given sequence.

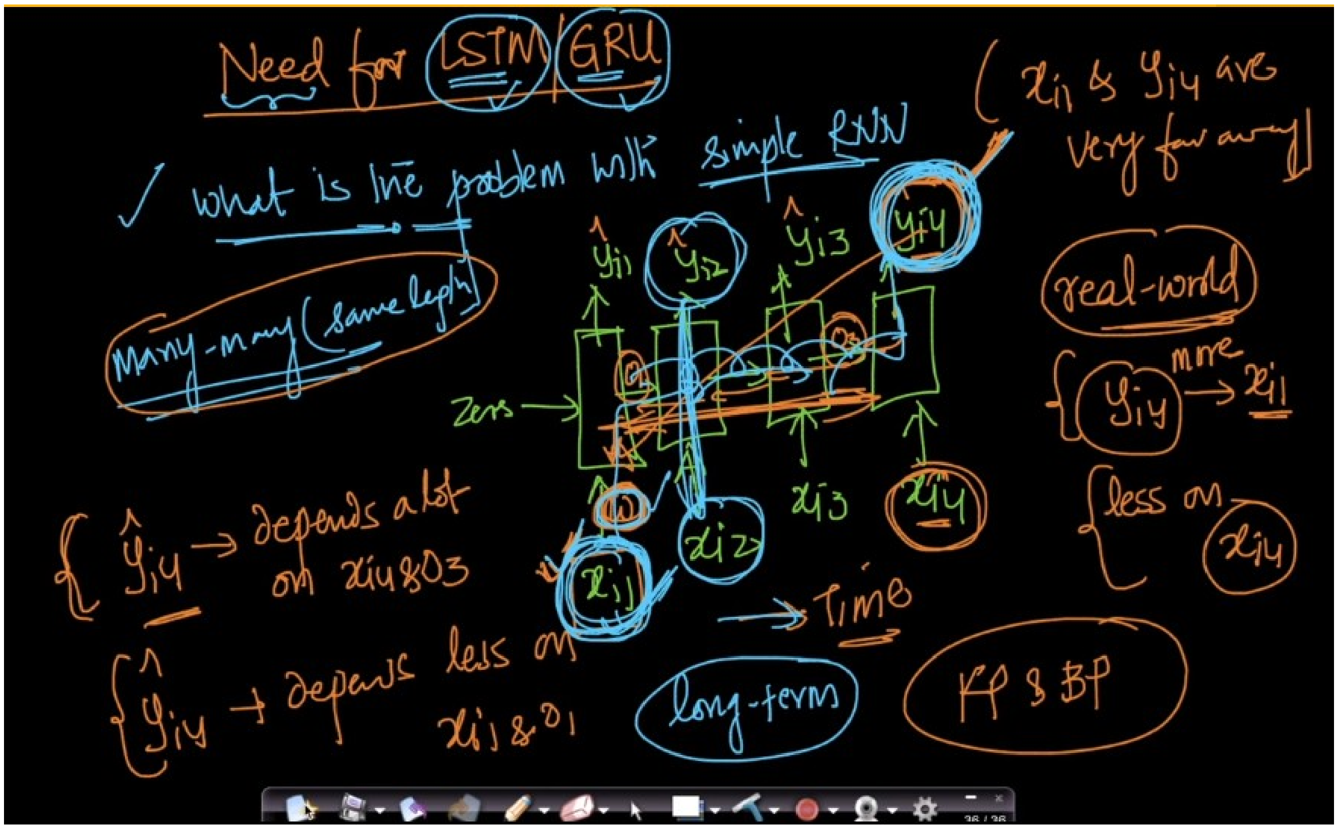


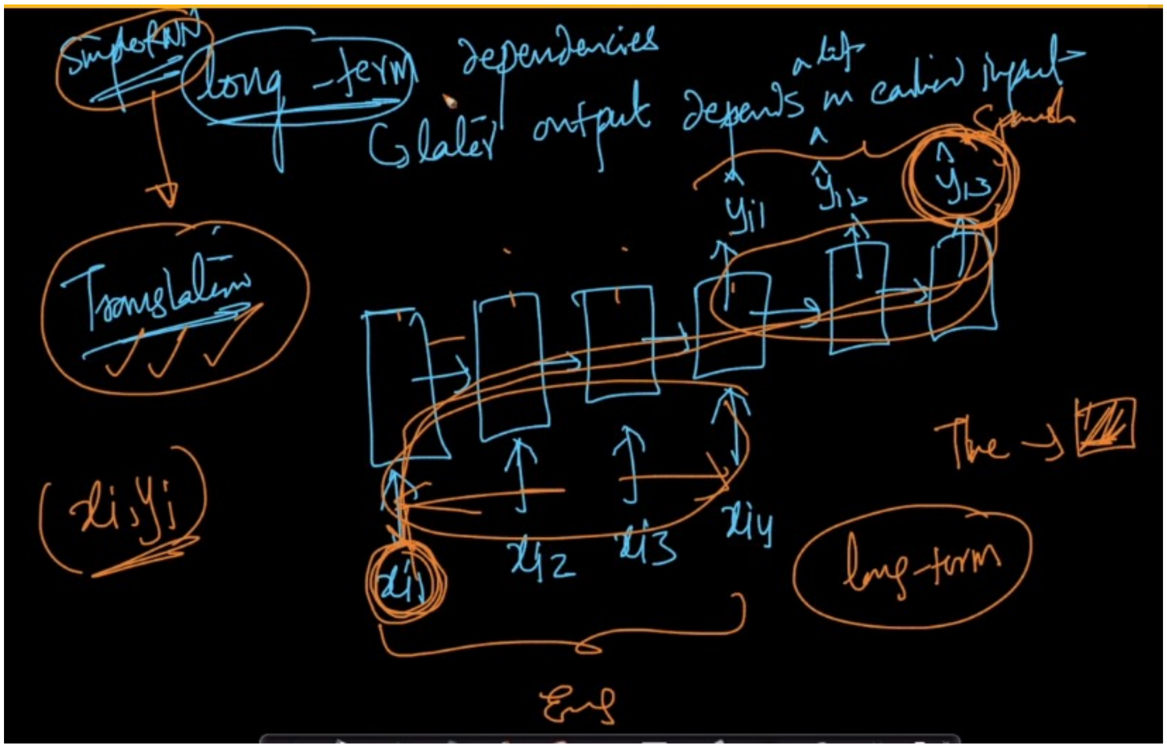
Different length.

Machine Translation is example of different lengths. This is also called the Encoder – Decoder Rnn’s.

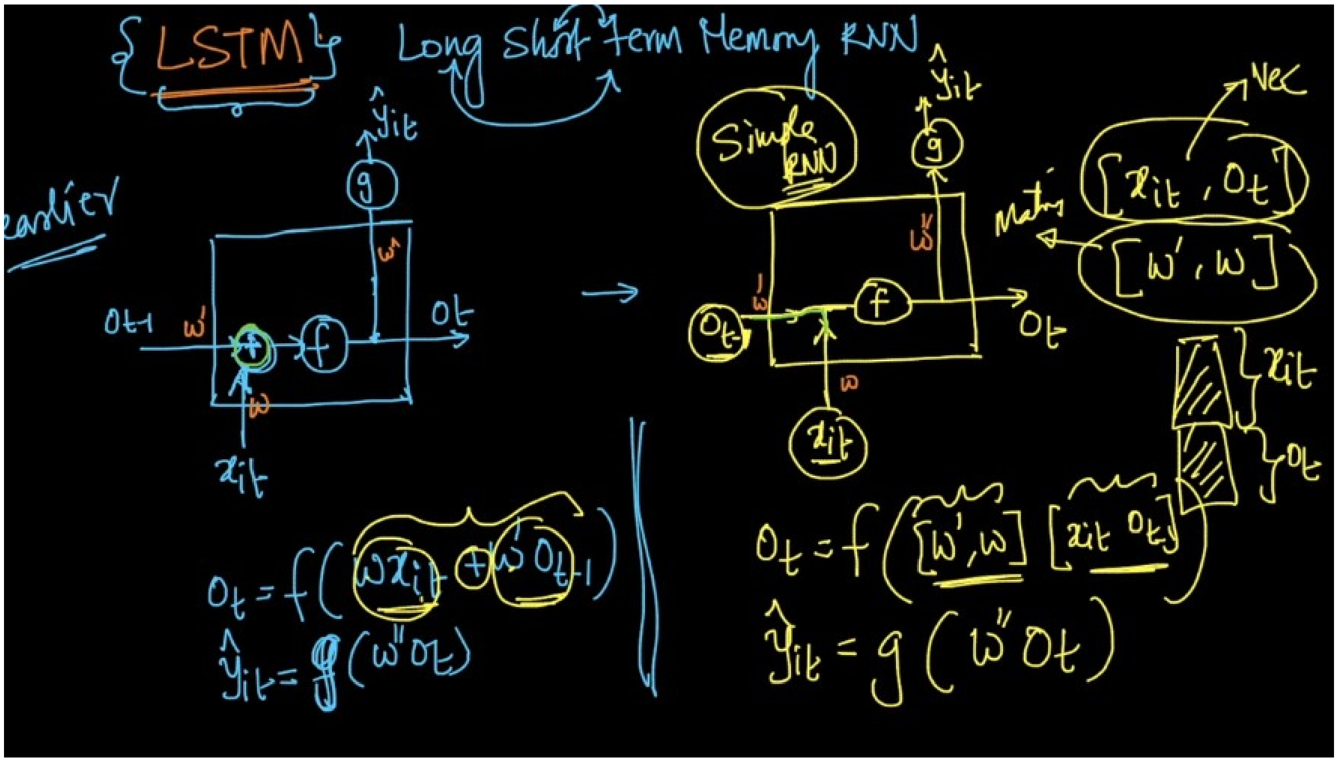


One – one is also present.
Need for LSTM/GRU:

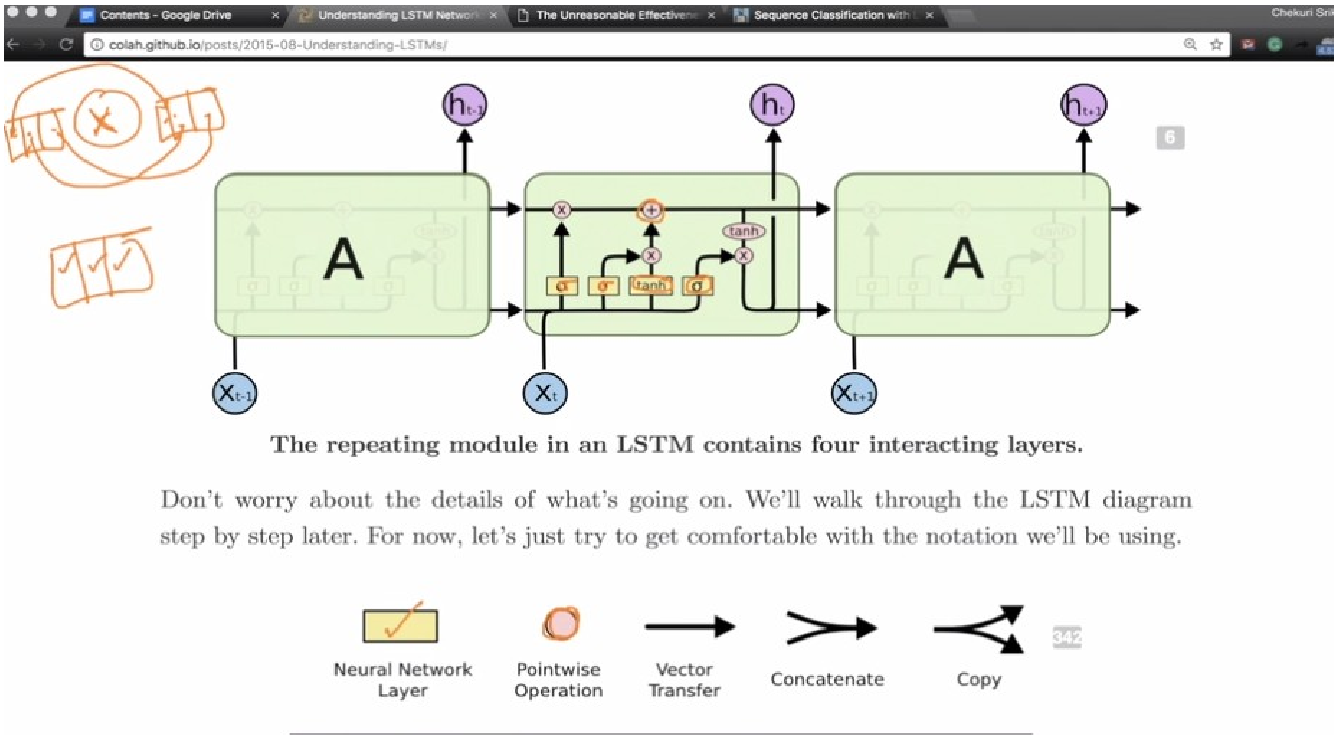

Need for LSTM/GRU
✓ what is the problem with simple RNN
Many-many (same length)
$\hat{y}_{i1}$ $\hat{y}_{i2}$ $\hat{y}_{i3}$ $\hat{y}_{i4}$
Zeros →
x_{i1} x_{i2} x_{i3} x_{i4}
→ Time
(x_{i1} & y_{i4} are very far away)
real-world
y_{i4} more → x_{i1}
less on x_{i4}
$\hat{y}_{i4}$ → depends a lot on x_{i4} & O_3
$\hat{y}_{i4}$ → depends less on x_{i1} & O_1
long-term
FP & BP



LSTM’s: In lstms we concatenate instead of summation.


{LSTM}
Long Short Term Memory RNN
earlier
Simple RNN
Vec
$[x_{it}, O_t]$
Matrix
$[w', w]$
O_{t-1}
O_t
x_{it}
$\hat{y}_{it}$
$O_t = f(wx_i + w'O_{t-1})$
$\hat{y}_{it} = g(w''O_t)$
$O_t = f([w', w][x_{it}\ O_t])$
$\hat{y}_{it} = g(w''O_t)$

Blog: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

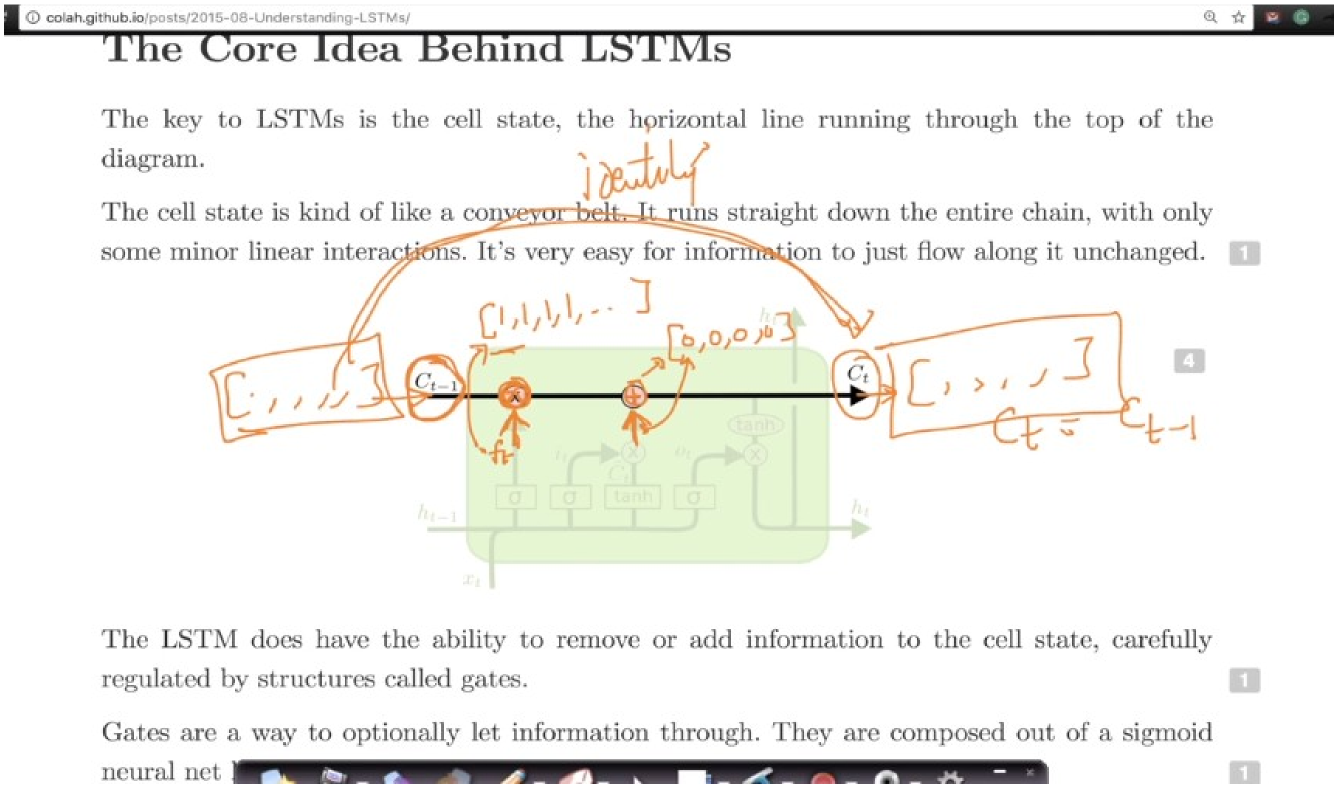


Contents - Google Drive
Understanding LSTM Network
The Unreasonable Effectivene
Sequence Classification with
colah.github.io/posts/2015-08-Understanding-LSTMs/
A
A
tanh
tanh
h_{t-1}
h_t
h_{t+1}
X_{t-1}
X_t
X_{t+1}

The repeating module in an LSTM contains four interacting layers.

Don't worry about the details of what's going on. We'll walk through the LSTM diagram step by step later. For now, let's just try to get comfortable with the notation we'll be using.

First path:



The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

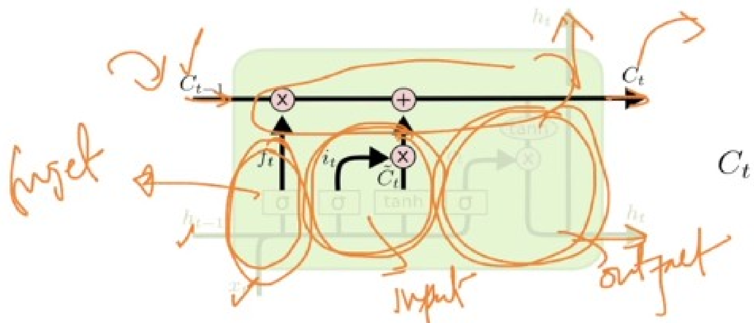
The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net

We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

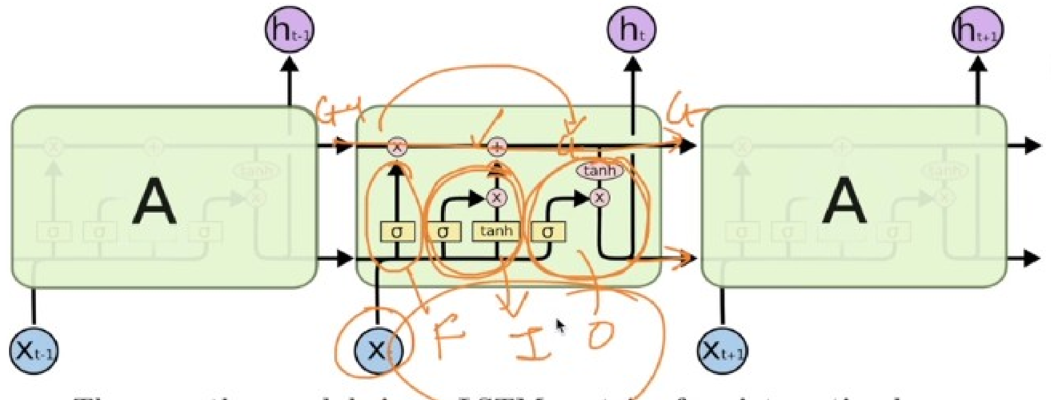
In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the

special way.



The repeating module in an LSTM contains four interacting layers.

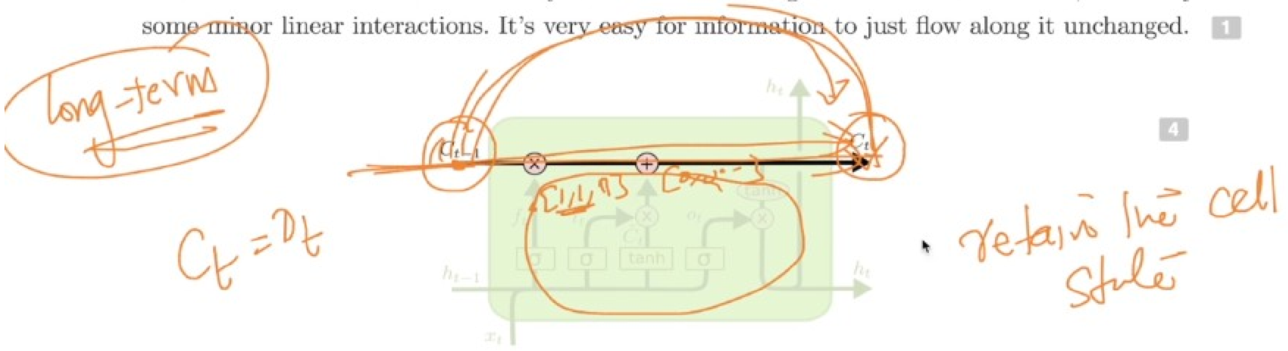
Don't worry about the details of what's going on. We'll walk through the LSTM diagram step by step later. For now, let's just try to get comfortable with the notation we'll be using.

We have the bypassing the input, by this the long term dependency is achieved.

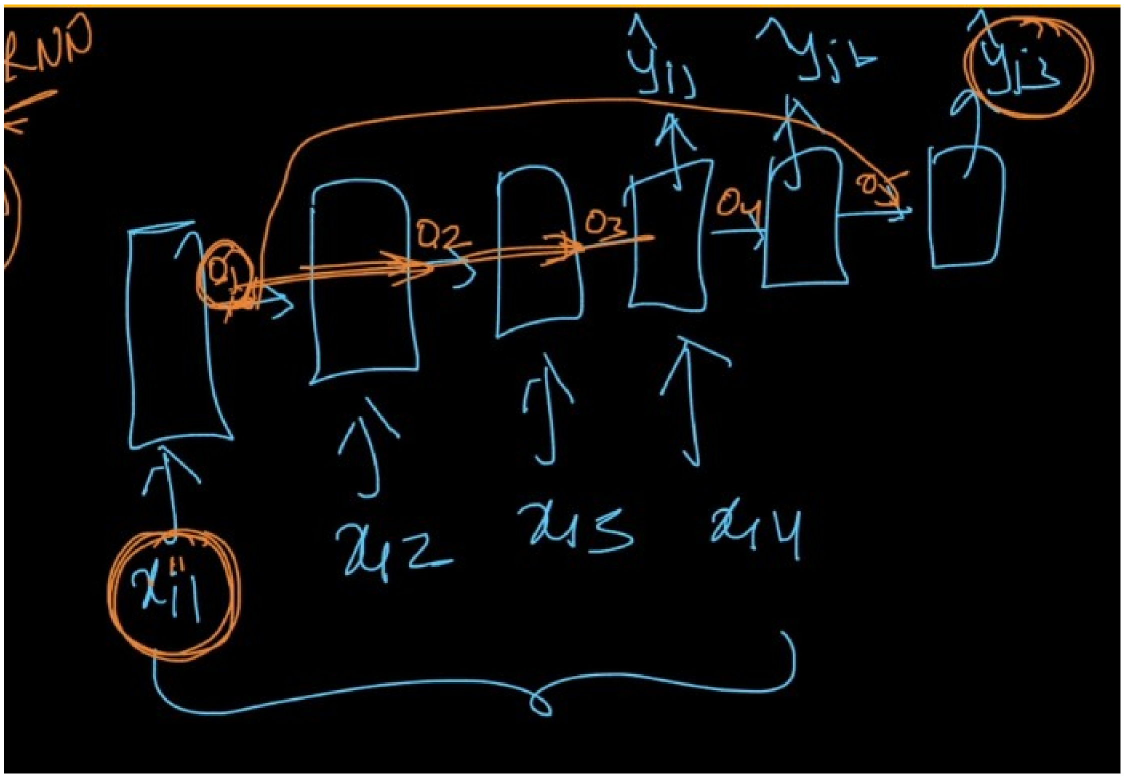
If the last layer needs the first layers output information, we will run into vanishing gradient problem. That is why we use the LSTM network to bypass some of the information. By this long term dependency worked out.

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

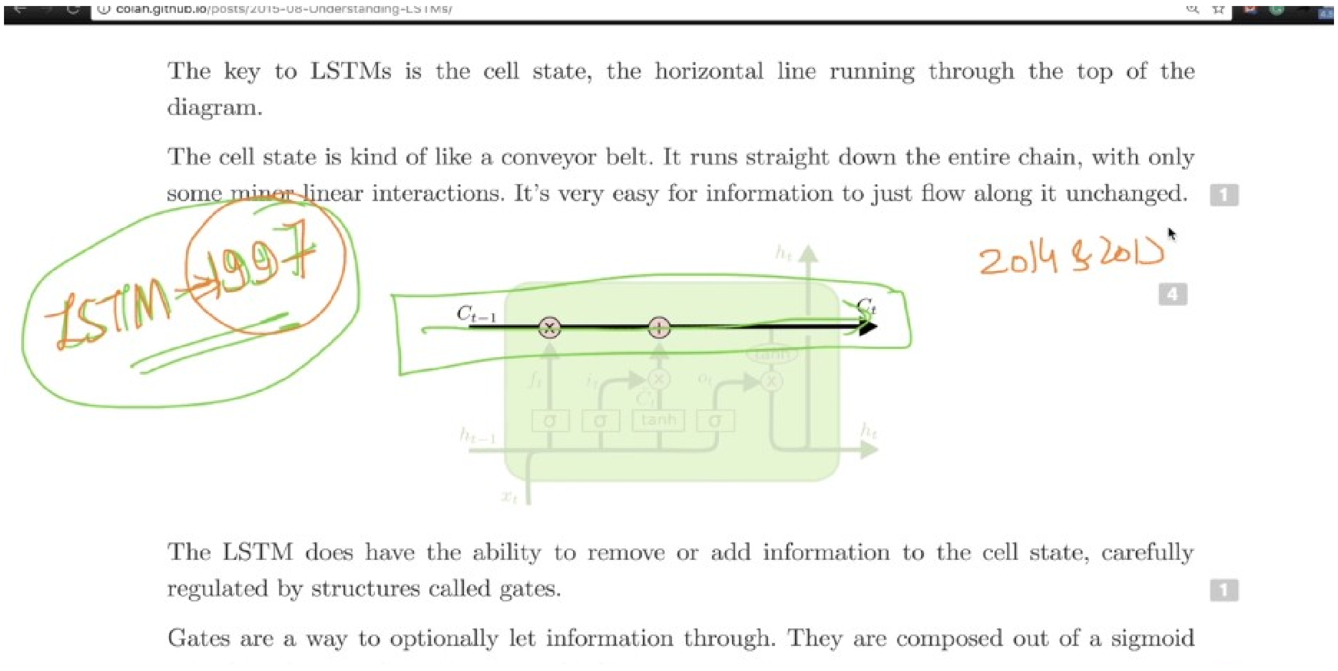
The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. 1



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. 1



By short circuiting the RNN network, we can impact something which is far away.



colah.github.io/posts/2015-08-Understanding-LSTMs/

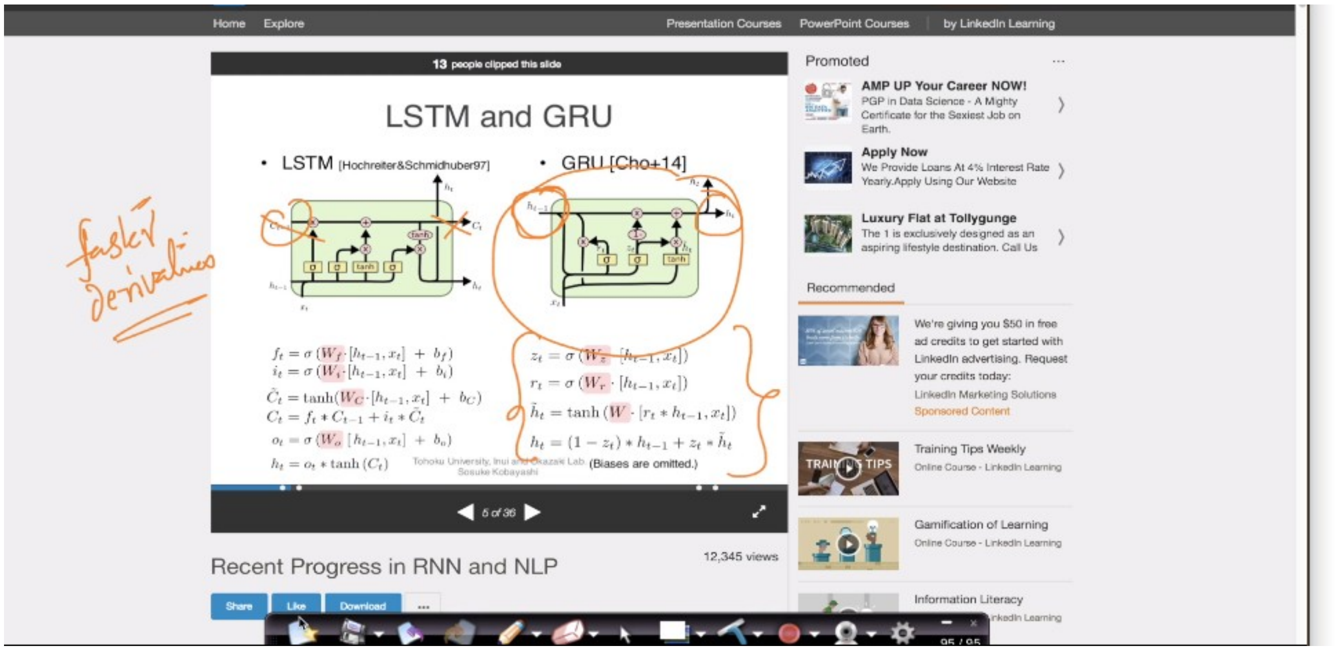
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid

GRU's:
These are more modern recurrent NN.



LSTM and GRU

- LSTM [Hochreiter&Schmidhuber97]
- GRU [Cho+14]

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$
$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Tohoku University, Inui and Okazaki Lab. Sosuke Kobayashi (Biases are omitted.)

5 of 36

Recent Progress in RNN and NLP

12,345 views

Share Like Download ...

Promoted

AMP UP Your Career NOW! PGP in Data Science - A Mighty Certificate for the Sexiest Job on Earth.

Apply Now We Provide Loans At 4% Interest Rate Yearly.Apply Using Our Website

Luxury Flat at Tollygunge The 1 is exclusively designed as an aspiring lifestyle destination. Call Us

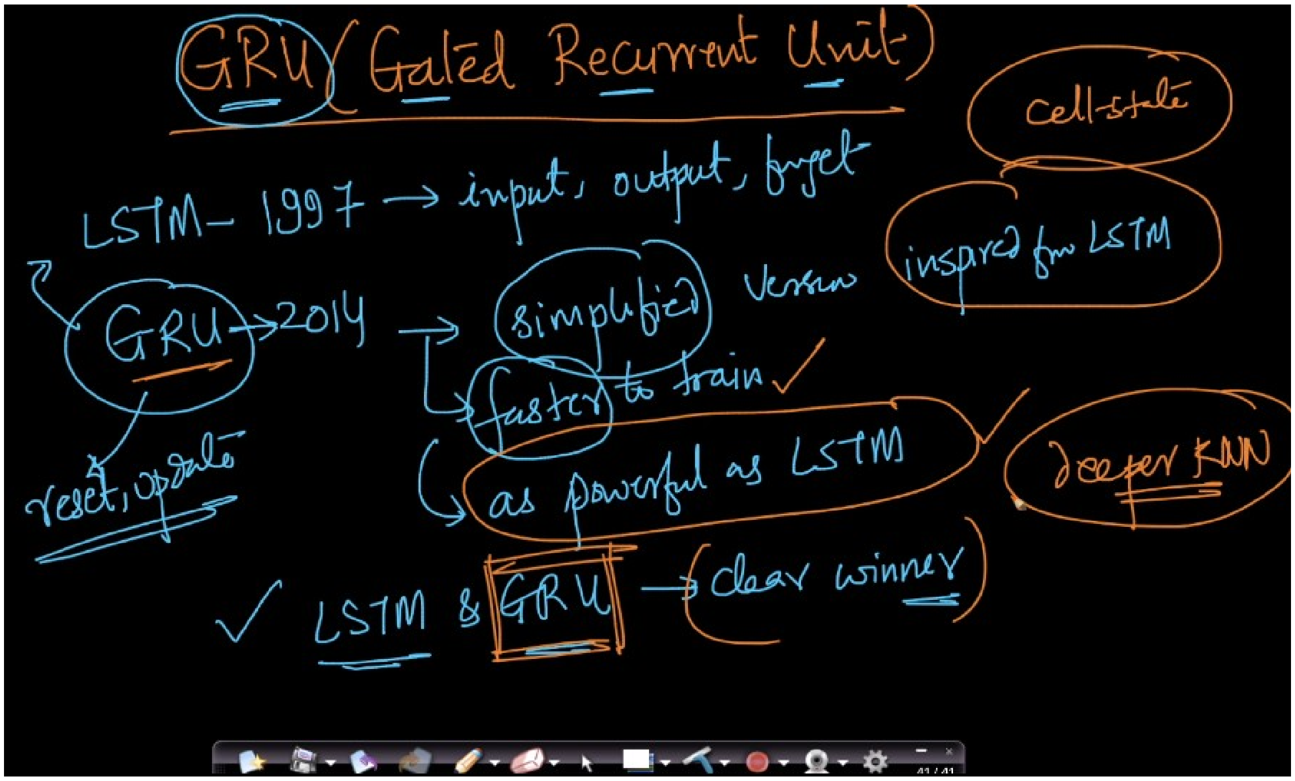
Recommended

We're giving you $50 in free ad credits to get started with LinkedIn advertising. Request your credits today: LinkedIn Marketing Solutions Sponsored Content

Training Tips Weekly Online Course - LinkedIn Learning

Gamification of Learning Online Course - LinkedIn Learning

Information Literacy Online Course - LinkedIn Learning



GRU (Gated Recurrent Unit)

cell-state

LSTM - 1997 → input, output, forget

GRU → 2014 → simplified version

inspired from LSTM

→ faster to train ✓

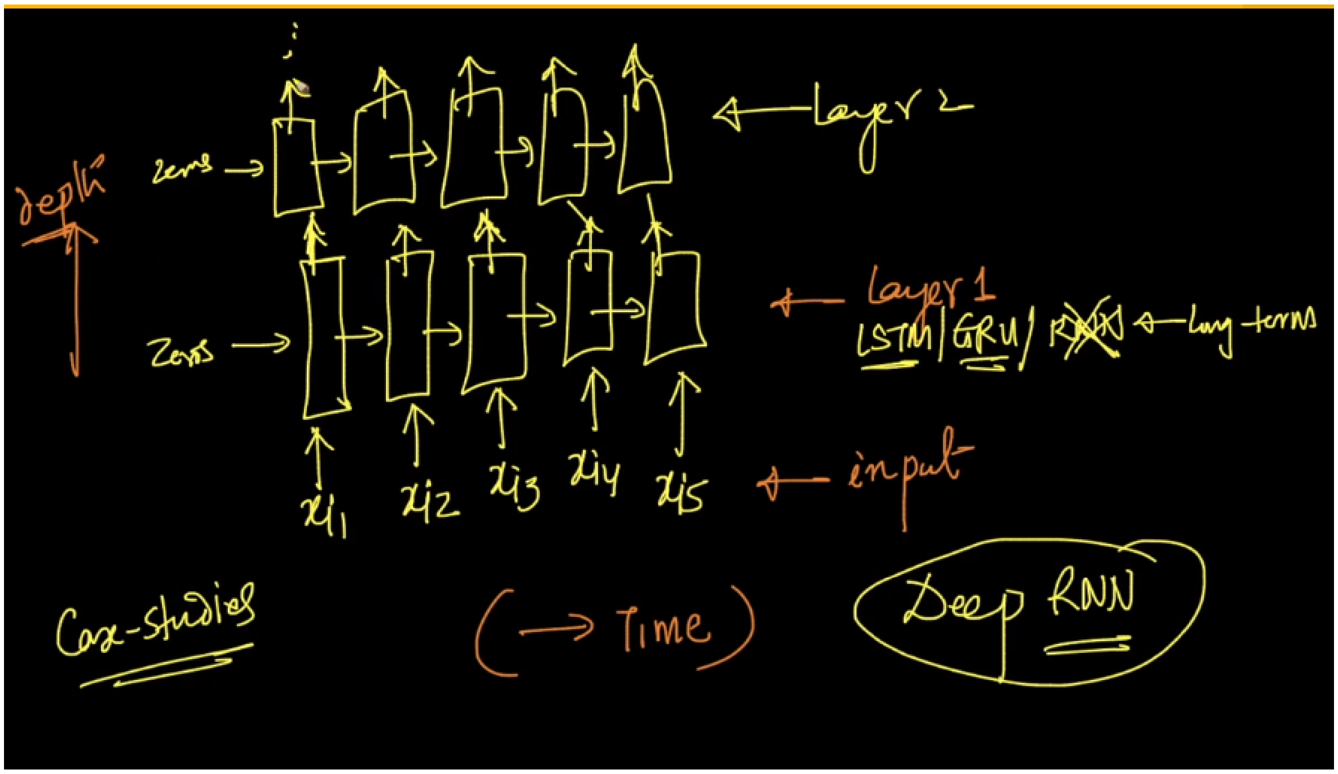
→ as powerful as LSTM ✓

Deeper RNN

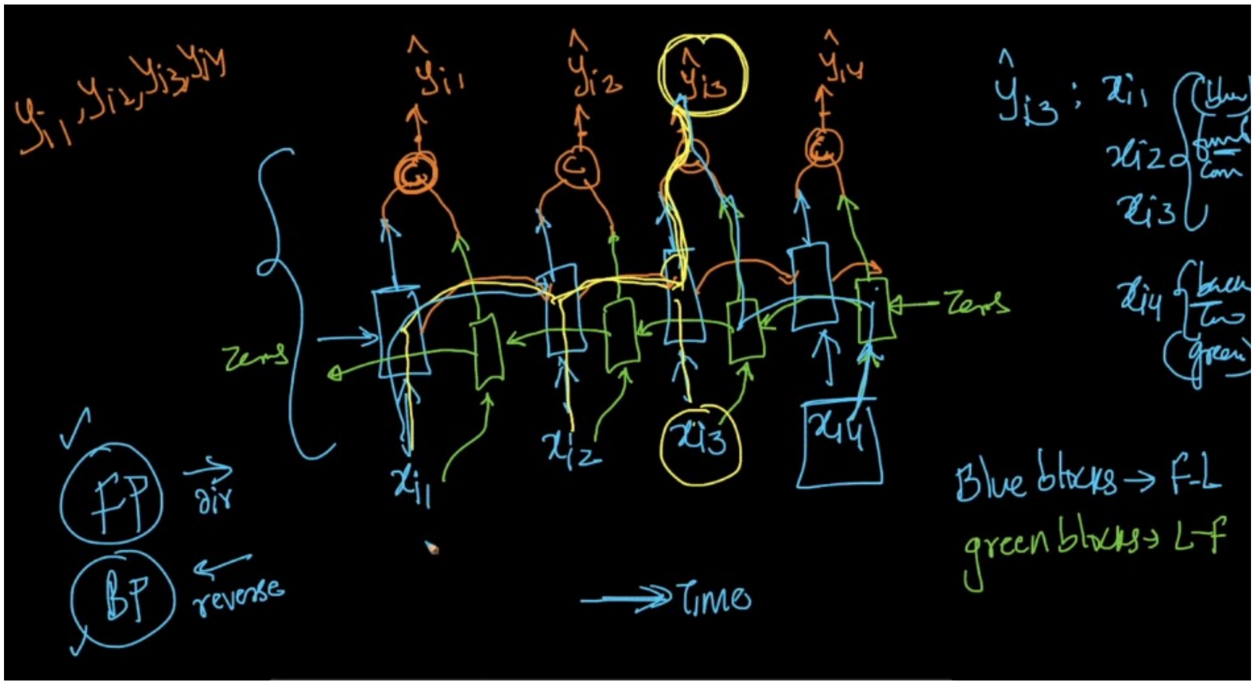
reset, update

✓ LSTM & GRU → (clear winner)

Most often we use the set of LSTMs or GRU cells than simple RNN's.



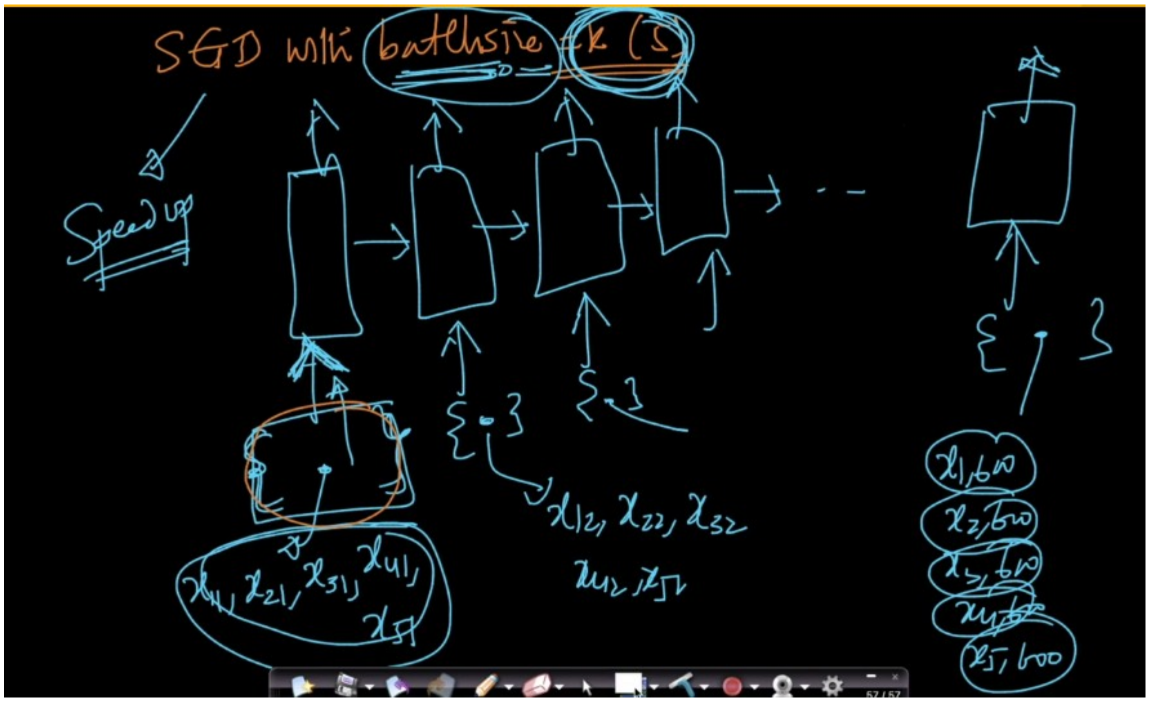
Bidirectional RNN's:



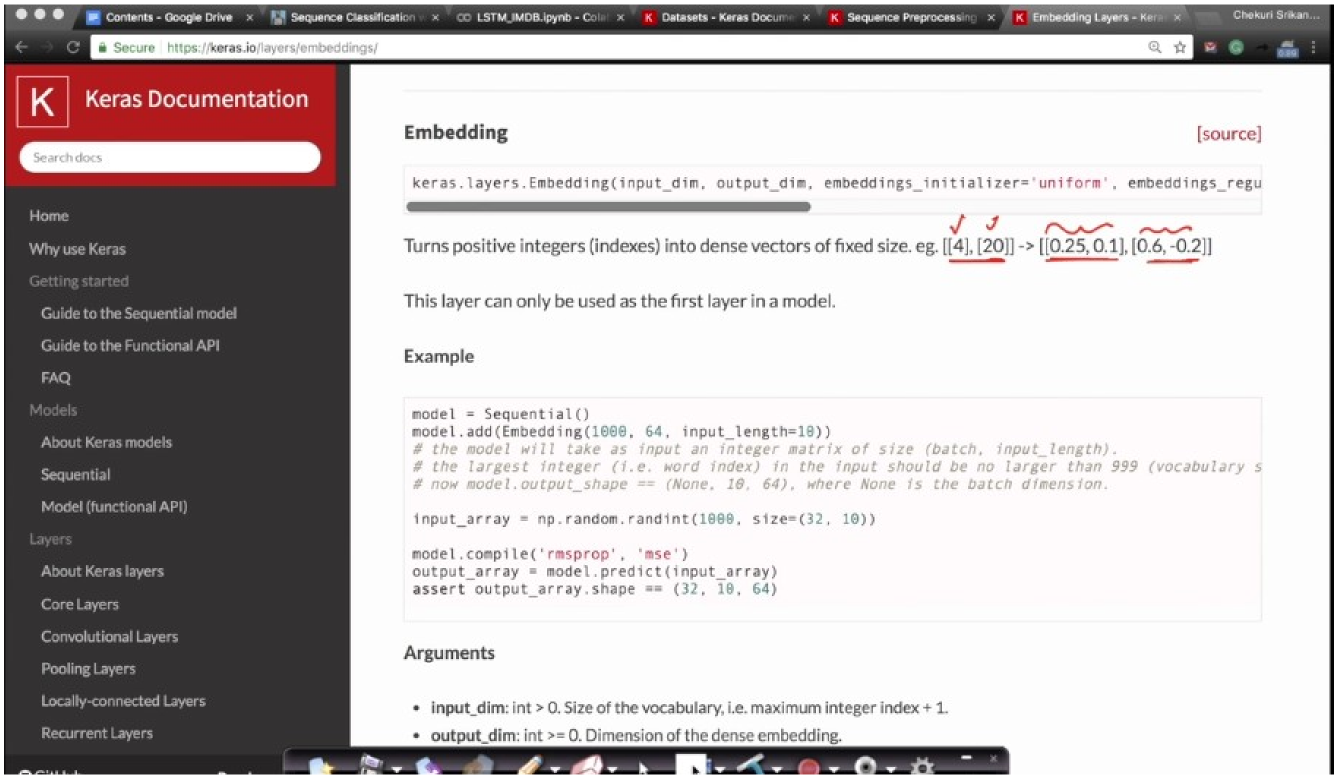
These are more common in NLP tasks.

When the sequences are of different lengths, it is hard to learn the weights because, we have the arbitary length of sequences.

So, batch would be better option to train the RNN's, If we pad all the sequences. This will speed up the LSTM because the sequences are of same length.



The first layer we apply Is the embedding layer. These are used a lot in NLP tasks.



Embedding

```
keras.layers.Embedding(input_dim, output_dim, embeddings_initializer='uniform', embeddings_regu
```

Turns positive integers (indexes) into dense vectors of fixed size. eg. [[4], [20]] -> [[0.25, 0.1], [0.6, -0.2]]

This layer can only be used as the first layer in a model.

Example

```
model = Sequential()
model.add(Embedding(1000, 64, input_length=10))
# the model will take as input an integer matrix of size (batch, input_length).
# the largest integer (i.e. word index) in the input should be no larger than 999 (vocabulary s
# now model.output_shape == (None, 10, 64), where None is the batch dimension.

input_array = np.random.randint(1000, size=(32, 10))

model.compile('rmsprop', 'mse')
output_array = model.predict(input_array)
assert output_array.shape == (32, 10, 64)
```

Arguments

- **input_dim**: int > 0. Size of the vocabulary, i.e. maximum integer index + 1.
- **output_dim**: int >= 0. Dimension of the dense embedding.

Each of my words can be represented as a dense vector.
LSTM 100 means, each of the LSTM takes the complete sequence of input one word after the other.