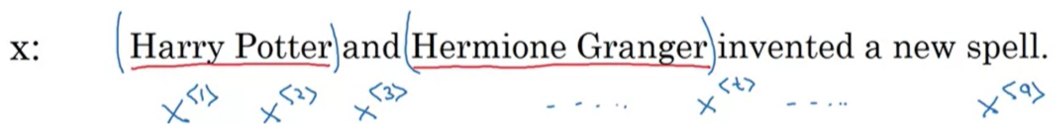
dl.ai_RNNs

1) A simple s to s binary classification

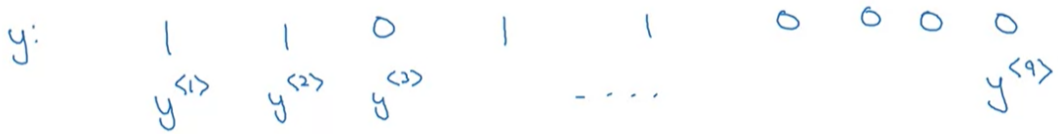
a) Notation

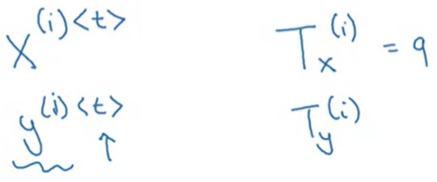
We are trying to create a simple named word identification model which on given a sentence as the input gives back as binary sequence as the output in which Each word in the text represents a token.So x will be denoted as



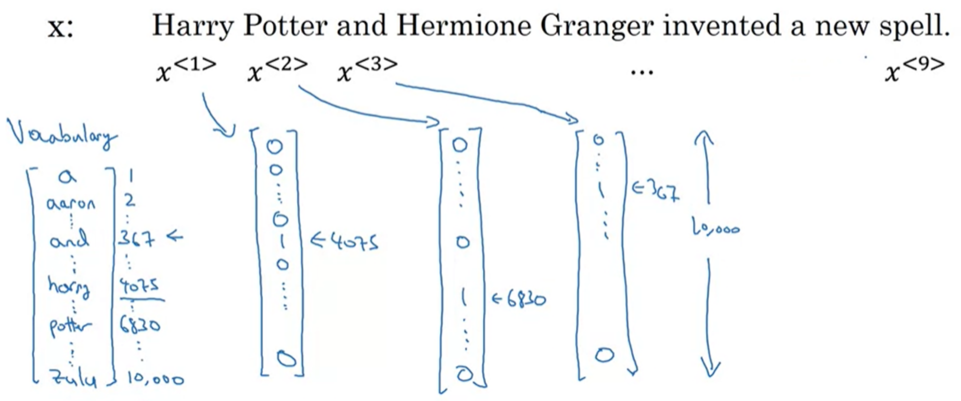
with $x^{<i>}$ denoting the i th indexed token

The target Y would be a sequence of zeros and ones denoting the presence of a named entity.Tx represents the no of token in the sentence



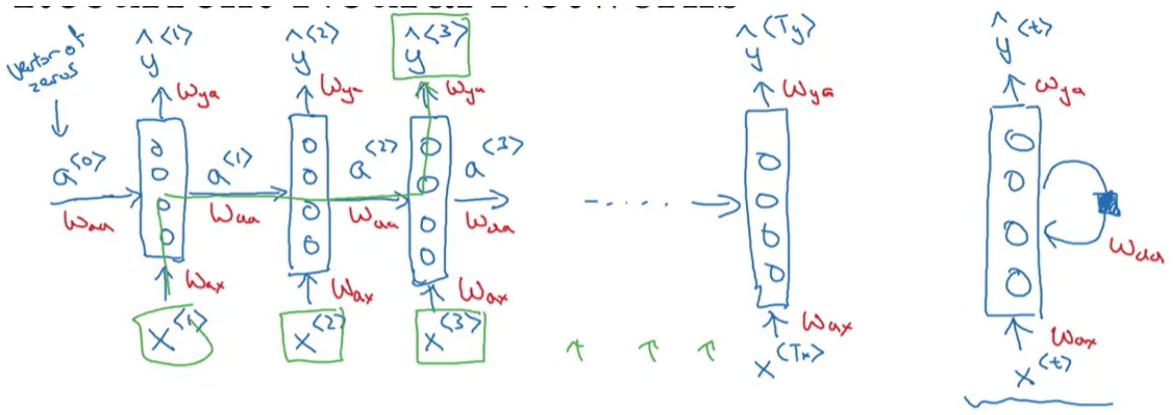


Each individual token in the Input X is represented as vector based on a dictionary of words having 1 whenever the word is present.



b)RNN-Forward propagation and Backward propagation

How rnn differs from a standard ann is its ability to find local correlation in data.It is also independent of the length of the sequence



We pass the input over the same neuron its just that the inputs that the neuron get changes every-time. The weight are shared between each tokens.

For the forward propagation we first find multiply the wax weight with x, waa weight with a0 add them along with the bias ba and apply the activation function over it. We get a1 the activation function for the next layer. The activation is generally a relu or tanh. Waa is smaller compared to wax.

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

we can stack a0 and x1, similarly we can stack waa and wax to get wa.

$$[W_{aa} ; W_{ax}] \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} = \underline{W_{aa} a^{<t-1>} + W_{ax} x^{<t>}}$$

Technically there are 4 parameters wa,ba,wy and by.

To find y1 we multiply a1 with wya add by to it and apply activation over it. The activation is generally a sigmoid function as y1 is generally a one hot encoded vector representing a word in the dictionary.

$$\hat{y}^{<t>} = g(W_{ya} a^{<t>} + b_y)$$

In ann we backpropagate through layers but in rnn we there is essentially only one layer, that is we only have 4 parameters which is shared across all tokens through time.

We instead back propagate through time since the weight is shared between all tokens we take the sum of each gradient.

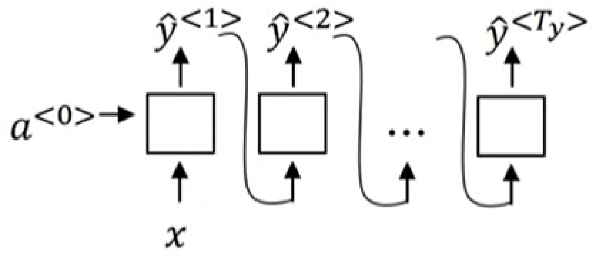
$$\frac{\partial E_3}{\partial W_S} = \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial W_S}\right) +$$

$$\left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial S_2} \cdot \frac{\partial S_2}{\partial W_S}\right) +$$

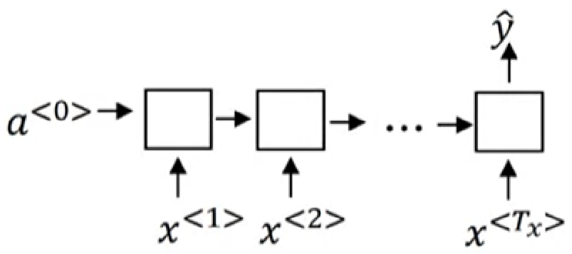
$$\left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial S_2} \cdot \frac{\partial S_2}{\partial S_1} \cdot \frac{\partial S_1}{\partial W_S}\right)$$

$$\frac{\partial E_N}{\partial W_S} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial Y_N} \cdot \frac{\partial Y_N}{\partial S_i} \cdot \frac{\partial S_i}{\partial W_X}$$

c) Types of RNN



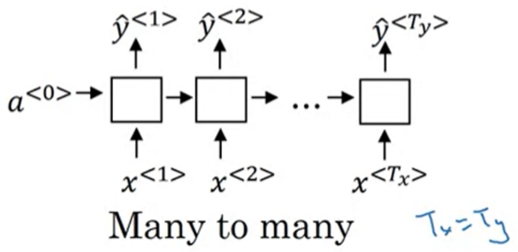
One to many



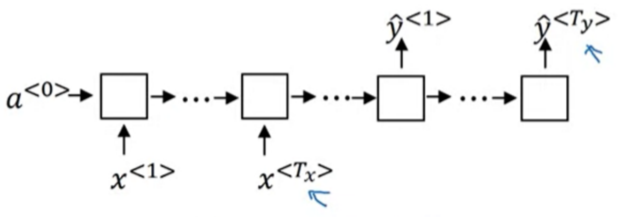
Many to one

One to many takes in one input and generates a sequence based on it. example music generation which only requires one input the genre.

Many to one takes in a sequence as an input and generates a single output based on it. example genre identification which only sends one output the genre.



Many to many



Many to many

Many to many Tx=ty takes in multiple inputs (sequence) and convert them into multiple outputs in a point to point basis. Example trigger word detection where for each token we classify whether its a trigger word or not.

Many to many Tx not= ty (RNN as a encoder decoder) which takes in multiple input (sequence) creates a single representation of the entire sequence and decodes it into another sequence.

2) Language Modeling

A language model basically models the probability of a sentence. That is it models the probability of a particular sequence of words irrespective of the input. Example the probability of “I and grandma ate cabbage” is more than ” cabbage and I ate grandma” irrespective of the input.

$$P(\text{The apple and pair salad}) = 3.2 \times 10^{-13}$$

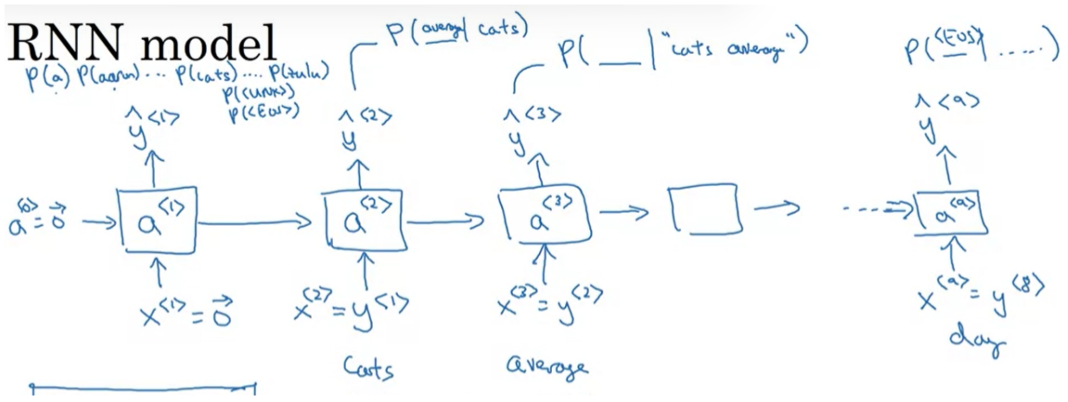
$$P(\text{The apple and pear salad}) = 5.7 \times 10^{-10}$$

$$P(\text{sentence}) = ? \qquad P(y^{<1>}, y^{<2>}, \ldots, y^{<T_y>})$$

Language models can be implemented using one to many rnn which take outputs a sequence of tokens(or their probabilities).Every unit of rnn will take the target output of previous unit as input. So each rnn predicts the probability of the the next word in a sequence given actual sequence as the input.

a)Sequence generation Training-RNN

Since we have a finite text vocabulary the model may encounter some words during training not present in the vocabulary. An unknown word is represented through token <unk> and end of a sentence through <eos>



The first unit model the probability P(y<1>) the second unit the probability P(<y2>|<y1>) and the third unit the probability P(<y3>|<y1>,<y2>).The final probability of P(<y1>,<y2>,<y3>) is given by the product of all the above probs.

$$P(y^{<1>}, y^{<2>}, y^{<3>}) = \underline{P(y^{<1>})}\, \underline{P(y^{<2>} \mid y^{<1>})}\, \underline{P(y^{<3>} \mid y^{<1>}, y^{<2>})}$$

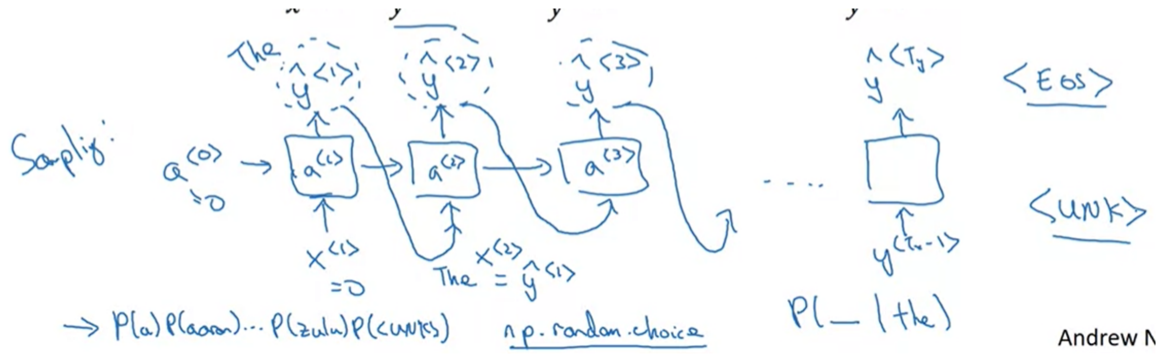
The loss function is defined based on sum of entropy of each prediction

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

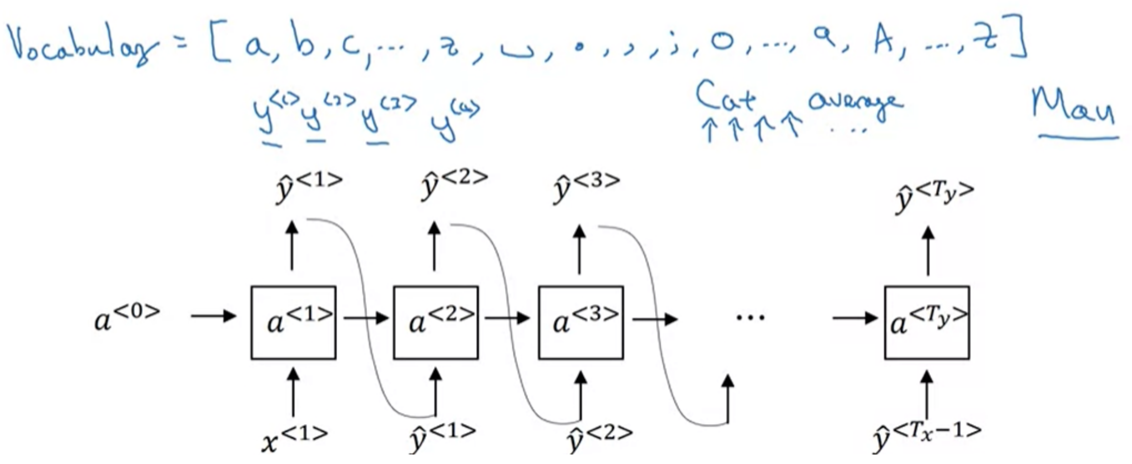
$$\mathcal{L} = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

b) Sampling a sequence-RNN

The above part was for training a sequence generation model during training we have access to the sequence and we essentially predict the next word in a sequence using the given actual previous word in the sequence. During training we don’t have access to the actual sequence hence we use whatever output s we have produced in the previous time steps as the input.



We stop the model when we get the <eos> taken as an output from a time step. The <unk> token must be however handled during sampling.A good way to avoid <unk> token is to use a character level sequence generation model.



However since the model now has to make exponentially more predictions for generating teh same sequence the computational cost is very high. Other issue with character based sequence generation is that it amplifies the vanishing gradient effect.

Rnns are more prone to vanishing gradient effect as it uses more number of units(very deep) compared to cnns.

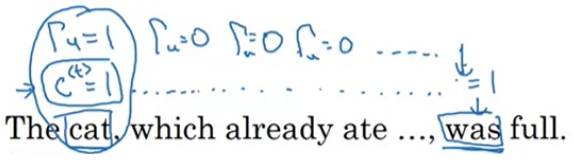
3) Variants of RNN

a)Gated Recurrent Unit

GRU came a whole 15 years after LSTM in-fact they are just a simpler version of LSTM .We can say that in LSTM we technically have two hidden states c<t> and a<t> in addition to y<t>,which means we have to train 6 parameters but what if we don’t use a separate state a<t> and just use c<t> for (i.e) every unit will only output a c<t> and y<t>.

c<t> or the memory cell just provides a way to store a specific context(plurality, gender etc) , not alter it and use it across the sentence. a<t> which we generally use gets altered in each unit so it stores a representation of the entire sentence rather than a specific context.

For example we can see in the below sentence the word c<t> is updated to a representation of cat. Since the update gates for rest of the sentence remains zero it gets carried over and can be used to decide the word “was” from among “were”, ”is” etc.

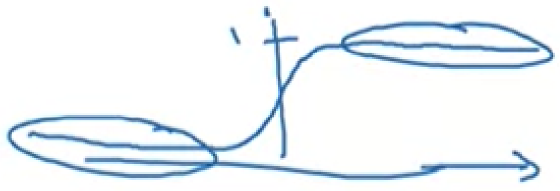


We define a update gate which is used to decide whether the memory cell gets updated by the candidate cell or not. The gate uses a sigmoid function over a linear function of concatenated c<t-1> and x<t> as it either has a value of 0 or 1.The candidate cell use the regular tanh activation over linear function of concatenated c<t-1> and x<t>.

$$\rightarrow \underline{c}^{<t>} = \underline{a}^{<t>}$$

$$\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$$

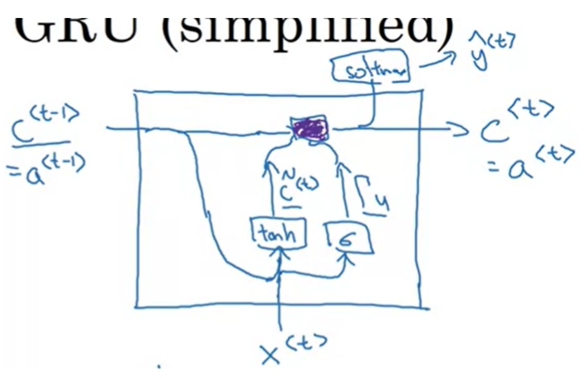
$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$



We then decide whether the memory cell gets updated by using the following

$$\underline{c^{<t>}} = \underset{\uparrow}{\Gamma_u} * \underline{\tilde{c}^{<t>}} + (1-\Gamma_u) * c^{<t-1>}$$

A Simplified version of GRU has the following flow



A full version of GRU has one more gate called reset gate why? In the above explanation for "cat" we can notice that if we get one more update gate =1(a contextual word) the existing c<t-1>and the new x<t> are concatenated and given equal importance in c<t>.But what if we want to reset the memory cell completely?. Drum rolls That's why we use use the reset gate.

$$\underline{\tilde{c}^{<t>}} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\underline{\Gamma_u} = \sigma(W_u[\, c^{<t-1>}, x^{<t>}] + b_u)$$

$$\underline{\Gamma_r} = \sigma(W_r[\, c^{<t-1>}, x^{<t>}] + b_r)$$

$$\underline{c^{<t>}} = \underline{\Gamma_u} * \underline{\tilde{c}^{<t>}} + \underline{(1 - \Gamma_u)} * c^{<t-1>}$$

$$\underline{a^{<t>} = c^{<t>}}$$

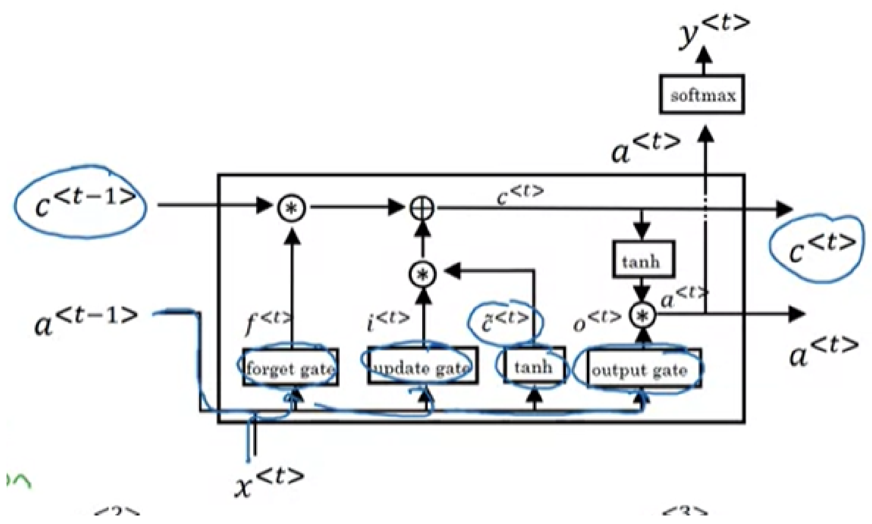
b)Long short term memory

In LSTM we have two gates to control c<t> the update gate to decide whether the new candidate must be added and forgot gate to decide whether the existing memory cell must be retained. The direct value of c<t-1> is always forgotten when new value gets updated(the new value c'<t> still might have c<t-1> in it).

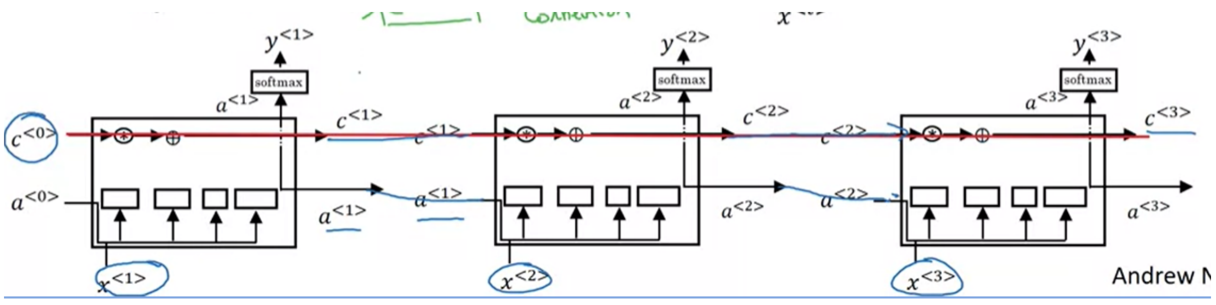
In addition to both of these gates we also have a output gate which basically ensures that the memory cell is different from the activation cell. The activation cell can be either be equal to memory cell or zero. a<t-1> is basically Γ_{o-1}* c<t-1> just like reset gate in GRU.

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$
$$\Gamma_u = \sigma(W_u[\, a^{<t-1>}, x^{<t>}] + b_u)$$
$$\Gamma_f = \sigma(W_f[\, a^{<t-1>}, x^{<t>}] + b_f)$$
$$\Gamma_o = \sigma(W_o[\, a^{<t-1>}, x^{<t>}] + b_o)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$

So if the output gate is zero for the previous unit then the information from c<t-1> will be won't be used in defining all three gates and the candidate cell for the next unit. In GRU the reset gate only has control over the c<t> for determining u<t> we always take both c<t-1> and x<t>.I cant figure out why a tanh has been added to c<t>.

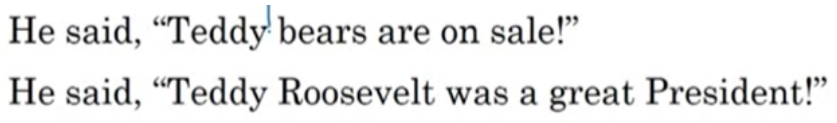


If value of the forgot gate is always 1 we can that the value of c<0> is carried till the end of the sentence.



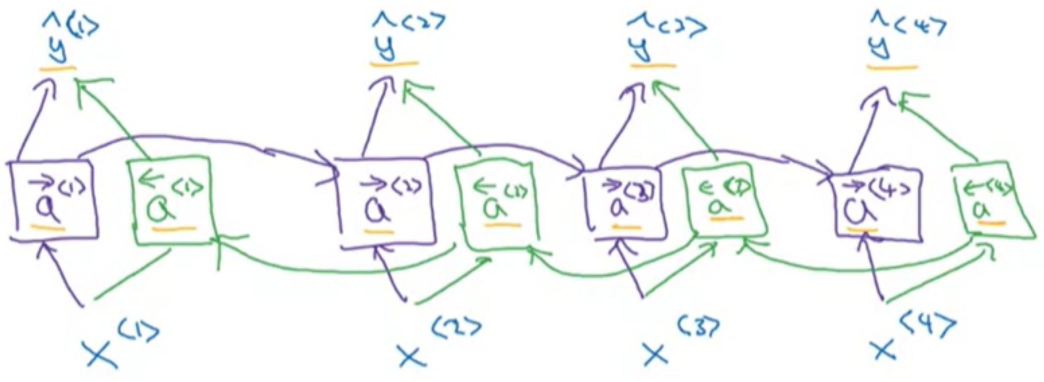
A variation called peephole connection basically involves directly using c<t-1> when finding the gates for the next unit in addition to a<t-1> and x<t>.

c)BRNN


He said, “Teddy bears are on sale!”

He said, “Teddy Roosevelt was a great President!”

In the above sentences if we have to do named word identification we would have to know the context of the sequence after the word “teddy”. However since rnns move from left to right we would not be able to capture this difference.



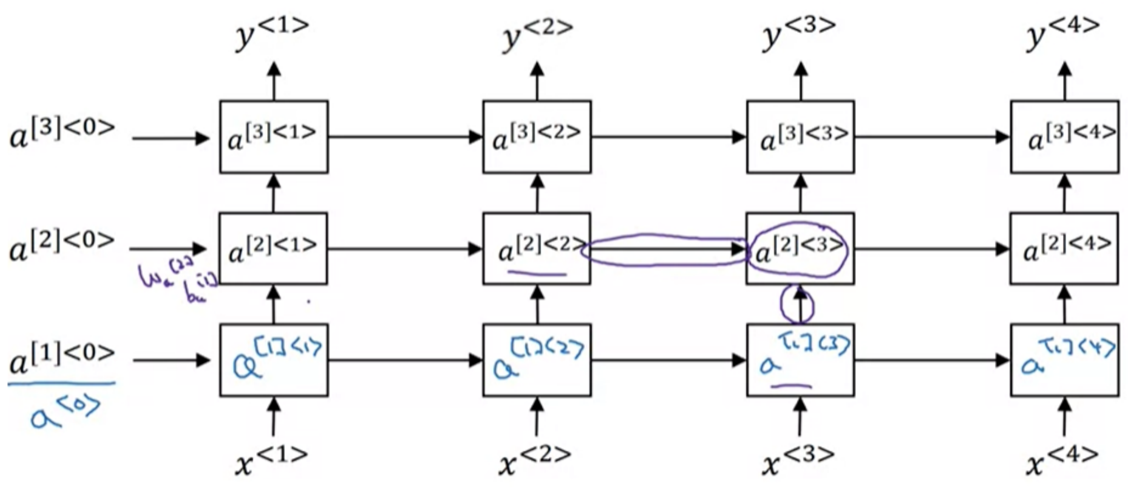
We move from left to right finding a→<i> for each unit till a→<4>.Then we move from right to left finding a←<i> for each unit. We don't loop at the fourth unit hence we can parallize it. We predict y<i> based on both a→<i> and a←<i>.

$$\hat{y}^{<t>} = g(W_y[\vec{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$$

Even GRUs and LSTMs can be made bidirectional. However for real time language generation BRNN tend to be slower as the entire sequence has to be passed once before generating an output. It is not word by word.

d)Deep RNN

If you notice in RNN we only use 4 parameters in each unit this limits the complexity of the model. GRUS or LSTM are not meant to increase the number of parameters (complexity).For a task like named word recognition it doesn't mater much but for a task like music generation we would need more complex units. We can extract more information from each individual unit by stacking layers of RNN hence called deep RNN.



We use a[j]<i> to denote the i th time unit for the j th layer. The units belonging to layers above the first layer get activation of the bottom unit of the below layer and the left unit unit of its own layer.

$$a^{[2]<3>} = g\left(W_a^{[2]}\left[a^{[2]<2>}, a^{[1]<3>}\right] + b_a^{[2]}\right)$$