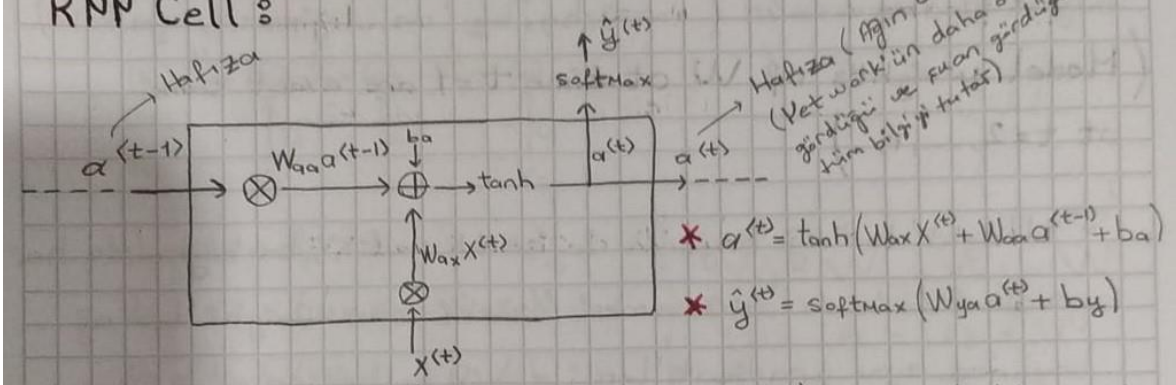
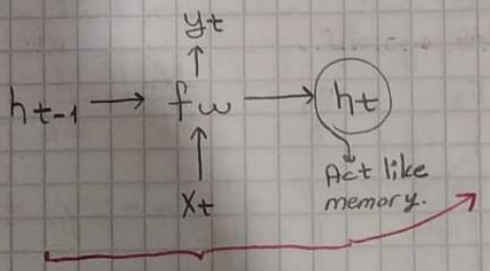
17) RNN ve LSTM

RNN Cell:



* $a^{\langle t \rangle} = \tanh(W_{ax}X^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$
* $\hat{y}^{\langle t \rangle} = \text{softMax}(W_{ya}a^{\langle t \rangle} + b_y)$

Takes as input $X^{\langle t \rangle}$ (current input) and $a^{\langle t-1 \rangle}$ (previous hidden state containing information from the past), outputs $a^{\langle t \rangle}$ which is given to the next RNN cell and also used to predict $y^{\langle t \rangle}$.

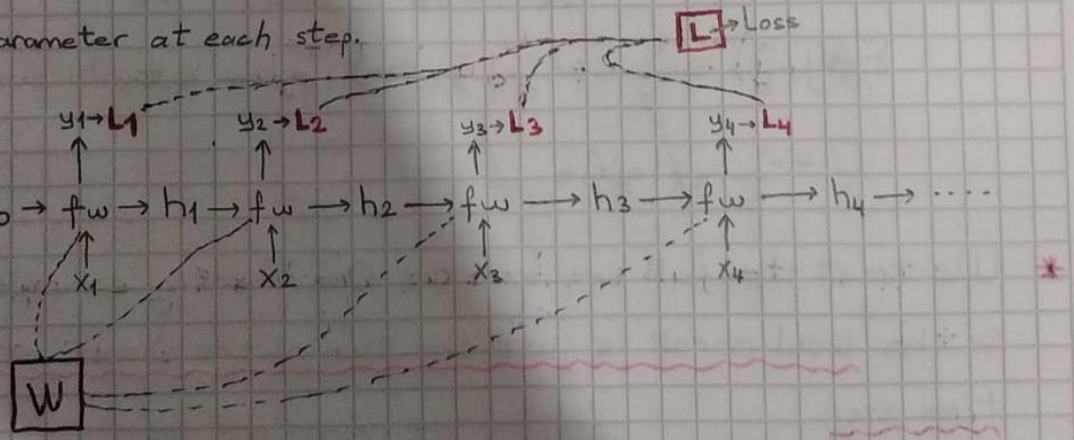


$h_t = f_w(h_{t-1}, x_t)$

$h_t = \tanh(W_{hh}h_{t-1} + W_{hx}X_t)$

$y_t = W_{hy}h_t$

* Unlike other deep neural networks which uses a *different parameter for each hidden layer*, RNN shares the same weight parameter at each step.



→ RNN where loss function L is sum of all the loss across Layers.

RNN'de W_{aa}, W_{ax}, W_{ay} ağırlıklarının her biri tüm layer'larda "Aynıdır". Yani tüm adımlarda aynı parametreler kullanılır. Bu sayede öğrenilmesi gereken parametre sayısı büyük ölçüde azalır. (Not ⇒ ba, by 'de değişmez, aynı kalır)

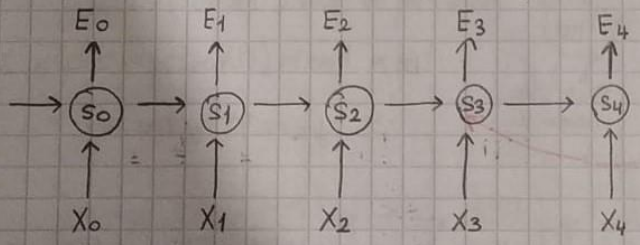
(Model weights of W at time $t=1$ are exact same at $t=2,\dots,n$.)

↳ Weight'ler ağ boyunca değişmez dedik, bu bağlamda her W_{ij} için gradient'lerin de değişmemesi lazım ki aynı Weight'ler ağ boyunca aynı gradient'ler ile güncellensin.

Backpropagation in RNN:

Many to Many:

Mantık tüm problem tiplerinde aynıdır.



$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$

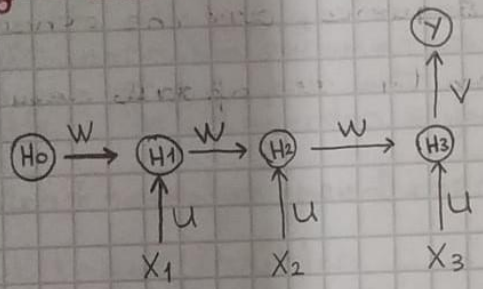
$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t) \quad \} \text{ Toplam Hata}$$

$$= -\sum_t y_t \log \hat{y}_t$$

* We treat the full sequence as One Training example, so the total error is just the sum of the Errors at each time step.

→ We sum up the gradients at each time step for one training example: $\underbrace{\frac{\partial E}{\partial W}}_{\text{Gradient of } W_{a,x,y}} = \sum_t \frac{\partial E_t}{\partial W}$

Many to one:



$$H_1 = \tanh(W_1 H_0 + U_1 X_1)$$
$$H_2 = \tanh(W_2 H_1 + U_2 X_2)$$
$$H_3 = \tanh(W_3 H_2 + U_3 X_3)$$
$$Y = V H_3$$

$$W_1 = W_2 = W_3 = W$$
$$U_1 = U_2 = U_3 = U$$

L or E is loss of the network. We want to minimize L.

During backpropogation, given $\frac{\partial L}{\partial Y}$ we want to calculate $\frac{\partial L}{\partial W}, \frac{\partial L}{\partial U}, \frac{\partial L}{\partial V}$.

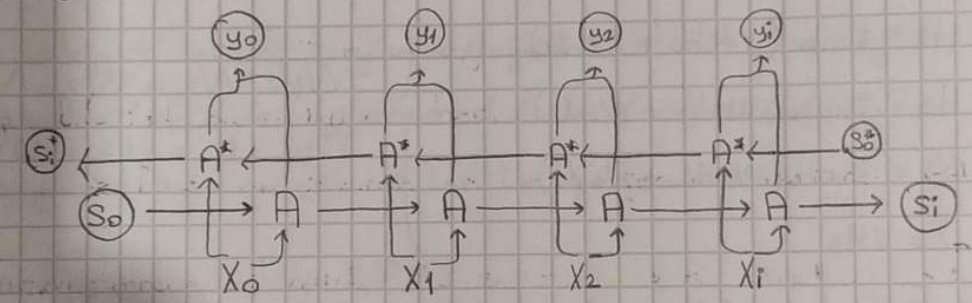
- $W = W - \alpha \frac{\partial L}{\partial W}$ $\qquad \frac{\partial L}{\partial W} = \frac{\partial L}{\partial W_1} + \frac{\partial L}{\partial W_2} + \frac{\partial L}{\partial W_3}$
- $U = U - \alpha \frac{\partial L}{\partial U}$ $\qquad \frac{\partial L}{\partial U} = \frac{\partial L}{\partial U_1} + \frac{\partial L}{\partial U_2} + \frac{\partial L}{\partial U_3}$
- $V = V - \alpha \frac{\partial L}{\partial V}$ $\qquad \frac{\partial L}{\partial V} = \frac{\partial L}{\partial Y} H_3^T$

Sonuç olarak; amaç toplam L'i minimize etmek.

Bidirectional RNNs :

BRNN is a combination of 2 RNN's - one RNN moves forward, beginning from the start of data sequence, and the other, moves reverse beginning from the end of data sequence.

Both the forward and reverse passes together train a BRNN.

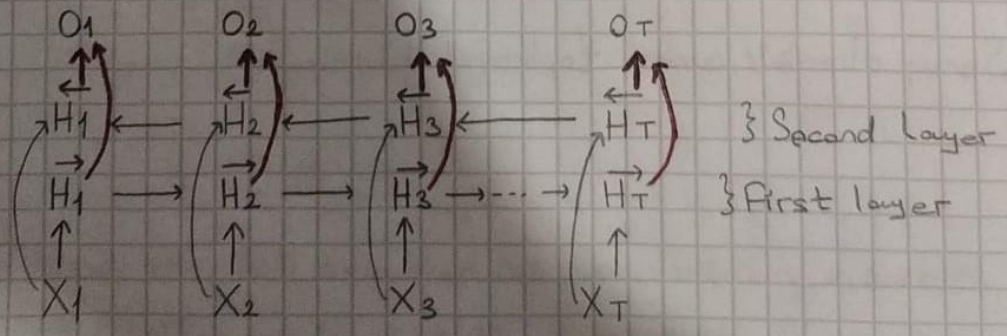


To produce the output of this BRNN, we simply concatenate together the corresponding outputs of 2 underlying RNN layers.

$$\overrightarrow{H}_t = \tanh\left(X_t W_{xh}^{(f)} + \overrightarrow{H}_{t-1} W_{hh}^{(f)} + b_h^{(f)}\right)$$

$$\overleftarrow{H}_t = \tanh\left(X_t W_{xh}^{(b)} + \overleftarrow{H}_{t+1} W_{hh}^{(b)} + b_h^{(b)}\right)$$

* In BRNN, there are 2 training process as forward and reverse happening simultaneously.



First Layer → 1st input X_1 and last input X_T

Second Layer → 1st input X_T and last input X_1

Bu 2 layer'ın; W değerleri birbirinden farklıdır.
Their learnable parameters are updated seperately.
Özetle; Tek layerdan ziyade elimizde 2 layer var
ve bu yapı Deep RNN diye de nitelendirilebilir.
İlk layer inputları ⟶ yönde alırken;
İkinci layer inputları ⟵ yönde alır.
Bu 2 layer'ın W parametreleri birbirinden farklıdır
Backpropogation ile öğrenilir ve güncellenir her ikisi de.

The Hidden state at time t is given by a combination of At (forward) and At (Reverse)

$$O_t = H_t * W_{ay} + b_y$$

(O_t: Output; H_t: Combination (At forward and At Reverse))

* Forward and Reverse RNN's Weights are updated seperately, while outputs of these 2 RNN layers are concatenate

LSTM : (Youtube ; Murat Karakaya 'iyi kaynak)

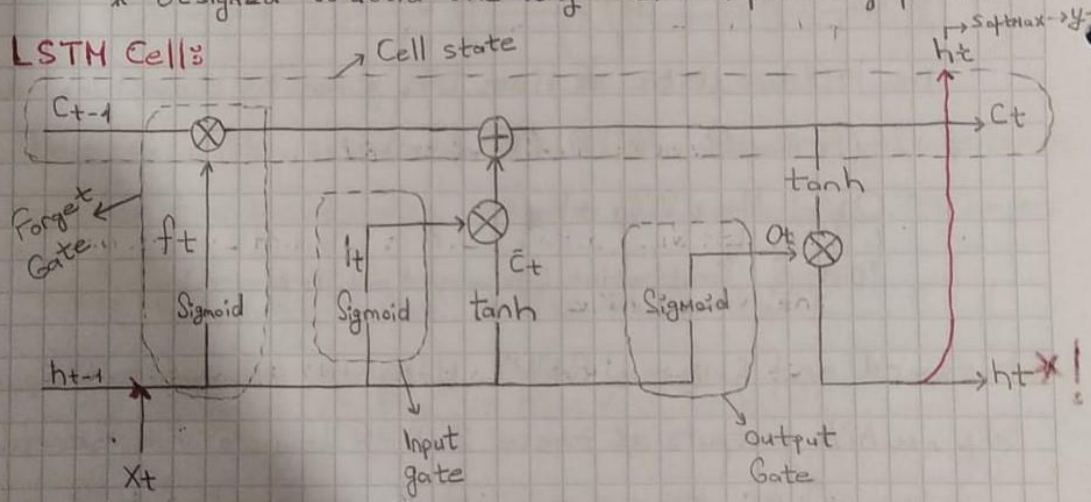
LSTM learn to keep only relevant information to make predictions, and forget non relevant data.

→ RNN "long term dependencies" i anlamakta zorluk çeker, nedeni "Vanishing Gradient Problem" dir. Modeller gradientlerin değişmesi (güncellenmesi) ile öğrenir. Gradientlerin güncellenmemesi modelin öğrememesine yol açar.

↳ **LSTM** bu problemi çözer.

* Designed to avoid the long-term dependency problem.

LSTM Cell:



* LSTM require 4 linear layer per cell to run at and for each sequence time step.

a) Forget Gate : (How much Past Data it should remember)

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

Decides, which information needs attention and which can be ignored. Generates values between 0 and 1.

Sigmoid result as 1 ⇒ Completely keep this.
" " as 0 ⇒ " get rid of this.

Özetle: Filters the information from input and previous output and decides which one remembered or forgotten.

- It concludes whether the part of the old output is necessary or not.

Bir daha özetle;

Forget Gate decides which pieces of the long-term memory should now be forgotten given the previous hidden state and the new data point in the sequence.

b) Input Gate: (How much this unit adds to the current state)

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

Input Gate decides what new information we are gonna store in the Cell State.

"The goal of this step is to determine what new information should be added to the networks long-term memory (cell state), given the previous hidden state and new input data."

* We use tanh here, its values lie in $[-1, 1]$ and so can be negative. The possibility of negative values is necessary if we wish to reduce impact of a component in the cell state.

Bu adımda tanh yardımıyla "new memory update vector" oluşturulur. Bu vektör bize, ağın uzun süreli belleğinin (cell state) her bir bileşeninin ne kadar güncelleneceğini söyler.

Yani bu adımdaki **tanh** katmanı cell state'e eklenebilecek yeni aday değerlerin bir vektörü olan $\tilde{C}_t$'yi yaratır.

Peki bu adımdaki i_t yani Sigmoid çıktısı ne yapar?

"Yani bellek vektörünün (tanh tarafından oluşturulan) hangi bileşenlerinin tutulmaya değer olduğunu belirleyen filtre görevi görür." [0,1] aralığında bir vektör üretecek ve noktasal çarpma yoluyla filtre görevi görecek.

↳ tanh çıktısı ($\tilde{C}_t$) (yeni bilgiler) ile Sigmoid çıktısı (i_t) noktasal çarpılır ve $\tilde{C}_t$'nin büyüklüğünün düzenlenmesine yardımcı olur.

c) Cell State:

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

f_t → forget gate

i_t → Input gate

$\tilde{C}_t$ → Generated by tanh (new memory update vector)

f_t → forgetting the things we decided to forget earlier.

$i_t * \tilde{C}_t$ → New candidate values, scaled by how much we decided to update each state value.

Pot:

- **Hidden state** is concerned with the most recent time step. (Working memory capability)
- **Cell state** is more concerned with the entire data, basically the global memory of the LSTM network over all time steps. (Long Term Memory capability)

RNN suffer from vanishing gradients problem which prevent learning of long data sequences when the gradients becomes smaller and smaller.

"Vanishing gradient" bölümünde anlattigimiz üzere sigmoid or tanh derivatives which are smaller than 1 cause this problem. (Bu sebeple, no significant learning will be done in reasonable time.)

The long term dependencies and relations are encoded in the cell state vectors and it's the cell state derivative that prevent LSTM gradients from vanishing.

Yani RNN'deki bu problemi "Cell State" çözer.

$$C_t = (C_{t-1} \otimes f_t) \oplus (\tilde{C}_t \otimes i_t)$$

$$\frac{\partial C_t}{\partial C_{t-1}} = \left(\frac{\partial f_t}{\partial C_{t-1}} \cdot C_{t-1}\right) + \left(\frac{\partial C_{t-1}}{\partial C_{t-1}} \cdot f_t\right) + \left(\frac{\partial i_t}{\partial C_{t-1}} \cdot \tilde{C}_t\right) + \left(\frac{\partial \tilde{C}_t}{\partial C_{t-1}} \cdot i_t\right)$$

Activation of Forget Gate

* The existence of the Forget Gate's activation vector in the gradient term, and additive structure of C_t solves Gradient Vanishing Problem.

d) Output Gate :

$$O_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = O_t * \tanh(C_t)$$

Output gate determines the value of the next hidden state. This state contains information on previous inputs.

New hidden state'e karar verirken;

- ✓ Newly updated Cell state
- ✓ Previous hidden state
- ✓ New input (x_t) data

kullanılır.

↳ Ancak; Forget Gate ve Cell state'de yaptığımız gibi Sigmoid ile filter görevi gören vektör yaratılır yani O_t.

- ✗ Buradaki O_t, Cell state'in hangi kısımlarının output olarak alınacağına karar verir.
- ✗ Hidden state contains information on previous inputs and also used for predictions.

Özetle

↳ Forget Gate decides what is relevant to keep from prior steps. Input Gate decides what information is relevant to add from the current step. The Output Gate determines what the next hidden state should be.

Önemli Bilgi :

The output of a LSTM is not a SoftMax. Dimensionality of LSTM output is equals to the number of unit, which is probably not the dimensionality of your desired target.
That is why you have to specify a last dense layer, which correspond to the $y_t = W * h_t$ equation.