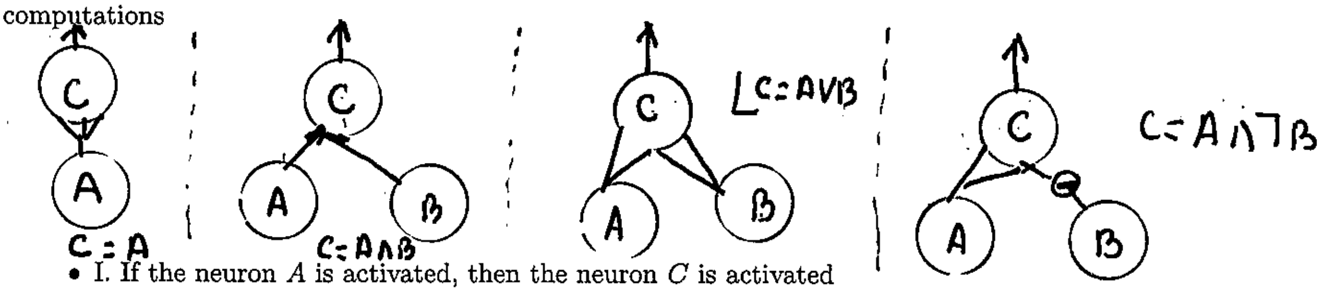
18. Artificial Neural Networks

18.1. Artificial Neuron. An artificial neuron has one or more binary inputs and one binary output. Here is an example of an Artificial Neural Network, which performs logical computations



- I. If the neuron A is activated, then the neuron C is activated
- II.The neuron C is activated only if the neuron A **and** the neuron B are activated
- III. The neuron C is activated only if the neuron A **or** the neuron B are activated
- IV. The neuron C is activated only if the neuron A is activated **and** the neuron B is not activated

18.2. Perceptron. For a perceptron inputs and outputs are numbers.

18.2.1. *Threshold Logic Unit (TLU).* Let the inputs be $x_1, x_2, ..., x_n$.

- The TLU computes a weighted sum

$$z = w_1x_1 + w_1x_1 + ... + w_nx_n$$

  the parameters $w_1, w_2, ..., w_n$ are called weights.
- Applies the step function to z and outputs the result

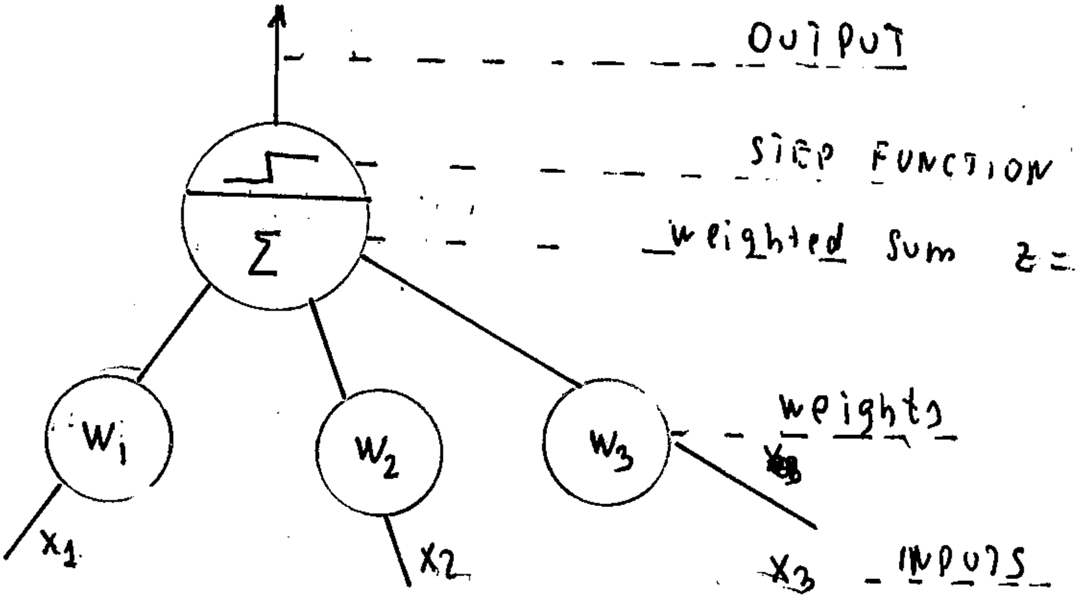
Examples of a step function

- Heaviside function

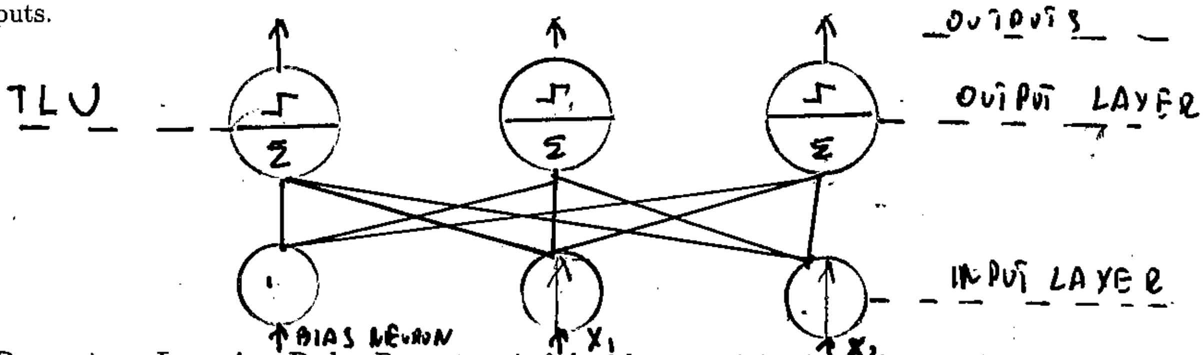
$$H(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

- Sign function

$$sgn(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ 1 & \text{if } z > 0 \end{cases}$$



A perceptron is composed of a single layer of TLU-s, with each TLU connected with all the inputs.



18.3. **Perceptron Learning Rule.** Perceptron is fed with one training instance at a time. Then it makes a prediction. If the prediction is wrong, it reinforces the connection weights from the inputs, which would have contributed to the correct prediction. It updates the weights according to the rule

$$w_{i,j}^{next.} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

Here

- $w_{i,j}$ is a connection weight between i-th input neuron and j-th output neuron
- $\hat{y}_j$ is the output of j-th neuron at the current training instance
- y_j is the target output of j-th neuron at the current training instance
- η is a learning rate

18.4. **The Deep Network.** Deep network consists of several layers. The first layer of the network is called the input layer, the last one is called the output layer. They layers which are between them are called hidden layers. The number of neurons in the input layer is equal to the number of features. When all neurons in a layer are connected to every neuron in the previous layer, then the layer is called a fully connected layer (dense layer).

18.4.1. *Cost functions.* Given a data point (x_i, y_i), where $x_i \in \mathbb{R}^{d+1}$, a Neural Network makes a prediction $\hat{y}_i(w)$, where w are the parameters (weights). It order to train the network we need cost functions. They are basically the same that we used previously. For continuous data we can use the means squared error

$$E(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i(w))^2$$

For cathegorical data we can use the cross entropy between predicted labels and true labels. For example for a binary outcome $y_i \in \{0, 1\}$ we have

$$E(w) = \sum_{i=1}^{n}(y_i \log(\hat{y}_i(w))(1 - y_i)\log(1 - \hat{y}_i(w)))$$

18.4.2. *Backpropagation algorithm.* Let l label a layer of the network, and let j label a neuron in this layer. Let us define a weighted sum which generalizes the one that was introduced above for a single neuron

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \tag{3}$$

where

$$a_j^l = \sigma(z_j^l) = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{4}$$

The expression b_j^l is a bias, which is specific to each neuron. Here σ is an activation function. It can be either a step function, or

- Sigmoid
$$\sigma(z) = \frac{1}{1+e^{-z}} .$$
- Hyperbolic Tangent
$$\sigma(z) = tanh(z) = \frac{e^{2z}-1}{e^{2z}+1}$$
- Rectified Linear Unit (ReLU)
$$\sigma(z) = \begin{cases} 0 & \text{if} \quad z \leq 0 \\ z & \text{if} \quad z \geq 0 \end{cases}$$
- Leaky ReLU
$$\sigma(z) = \begin{cases} 0.1z & \text{if} \quad z \leq 0 \\ z & \text{if} \quad z \geq 0 \end{cases}$$
- ELU
$$\sigma(z) = \begin{cases} 0 & \text{if} \quad e^{z-1} \leq 0 \\ z & \text{if} \quad z \geq 0 \end{cases}$$

The error of neuron j at the layer l is defined as

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial a_j^l}\sigma'(z_j^l) \tag{5}$$

The backpropagation algorithm goes as follows

- Calculate the activation functions a_j^1 for all neurons in the first layer.
- Feedforward: Compute z_j^l and a_j^l in each subsequent layer using (3) and (4).
- Calculate the error at the output level L using
$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial a_j^l}\sigma'(z_j^l)$$

- Backpropagate the error, which means to compute Δ_j^l for all layers using

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \sum_k \frac{\partial E}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_k^{l+1}} = \sum_k \Delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_k^{l+1}} = \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1} \right) \sigma'(z_j^l)$$

  ($\sigma'(x)$ is a derivative of $\sigma(x)$ with respect to the input evaluated at x)
- Calculate the gradient using

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial b_j^l}$$

  and

$$\Delta_j^l = \frac{\partial E}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1}$$

18.4.3. *To summarize.* The algorithm works as follows: we feed the data to the input layer, and using the network architecture propagate it to the outcome layer and see what is the error, comparing the output with the training data. Then we can determine the errors produced by each neuron, backpropagating the error. We update the wights using the gradient descent method and propagate the data forward again. And so on. A cycle over all datapoints is called an epoch. A number of hidden layers, a number of the neurons in hidden layers and the number of epochs is a priori arbitrary, and is determined by the problem at hand.

18.5. **Glorot initialization of the weights.** Let fan_{in} denote the number of inputs in the layer and fan_{out} denote the number of neurons in the layer. Finally,

$$fan_{avg.} = \frac{fan_{in} + fan_{out}}{2}$$

One uses the random initialization for the weights, with

- Normal distribution with mean 0 and the variance equal to

$$\sigma^2 = \frac{1}{fan_{avg.}}$$

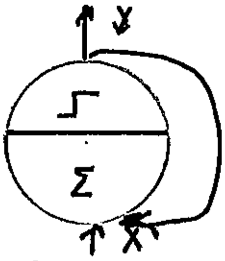
- Or uniform distribution between $-r$ and r, with

$$r = \sqrt{\frac{3}{fan_{avg.}}}$$

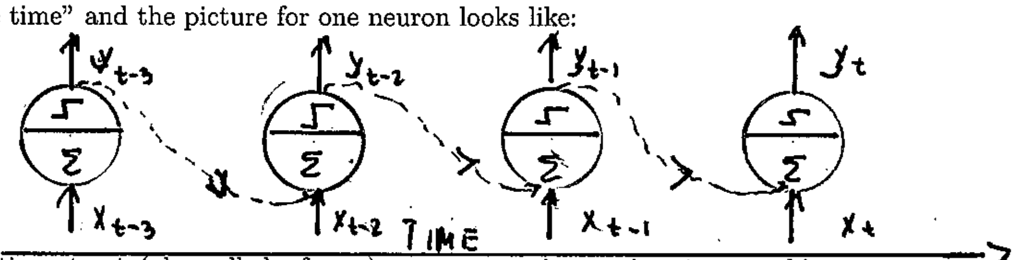
The activation functions are logistic, hyperbolic tangent, softmax or None. This is called Glorot initialization.

19. Recurrent Neural Networks (RNN)

RNN works like a feedforward Neural Network considered above, but it has connections pointing backwards



A neuron is getting an input, producing an output and feeding it back to itself. One can "unroll the time" and the picture for one neuron looks like:



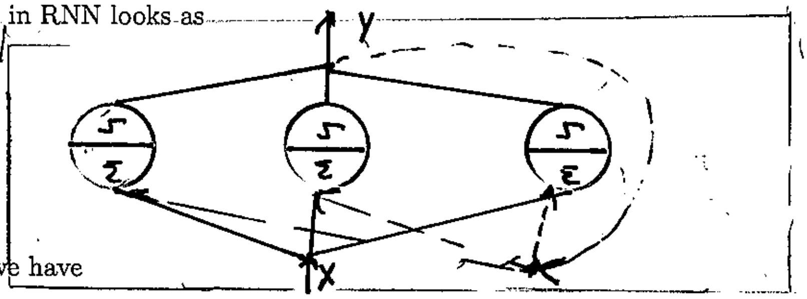
At each time step t, (also called a frame) a neuron receives an input $x_{(t)}$ and its own output $y_{(t-1)}$ at the previous time step $t-1$.

A neuron has two sets of weights:

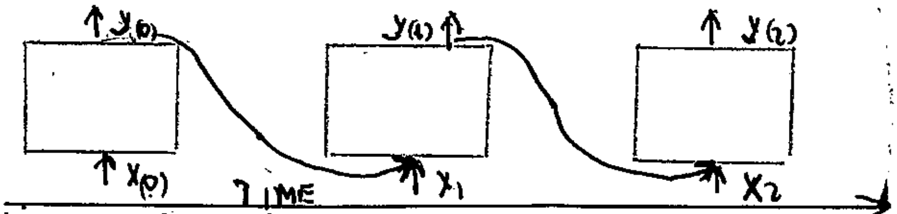
- For inputs $x_{(t)}$, which is a vector w_x
- For inputs $y_{(t-1)}$, which is a vector w_y

19.1. A layer.

A layer in RNN looks as



Unrolling the time, we have



For a layer the corresponding weights W_x and W_y are matrices. An output for a recurrent layer for a single instance is

$$y_{(t)} = \sigma(W_x^T\, x_{(t)} + W_y^T\, y_{(t-1)} + b)$$

where "T" means transpose matrix.

If we have several instances ("a minibatch") then $x_{(t)}$ becomes an input matrix. For all outputs in a minibatch we have

$$Y_{(t)} = \sigma(W_x^T X_{(t)} + W_y^T Y_{(t-1)} + b)$$

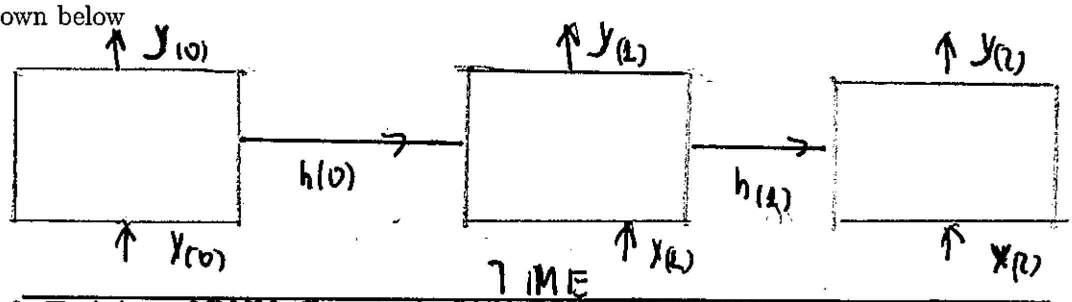
In this equation

- $Y_{(t)}$ is an $m \times n_{neurons}$ matrix, where m is a number of the instances in the minibatch.
- $X_{(t)}$ is an $m \times n_{inputs}$ matrix, where n_{inputs} is a number of inputs i.e., a number of features.
- W_x is an $n_{inputs} \times n_{neurons}$ matrix. It is a matrix of connection weights for inputs at the current time step.
- W_y is an $n_{neurons} \times n_{neurons}$ matrix. It is a matrix of connection weights for outputs at the previous time step.
- b is a vector of size $n_{neurons}$ and represents a bias.

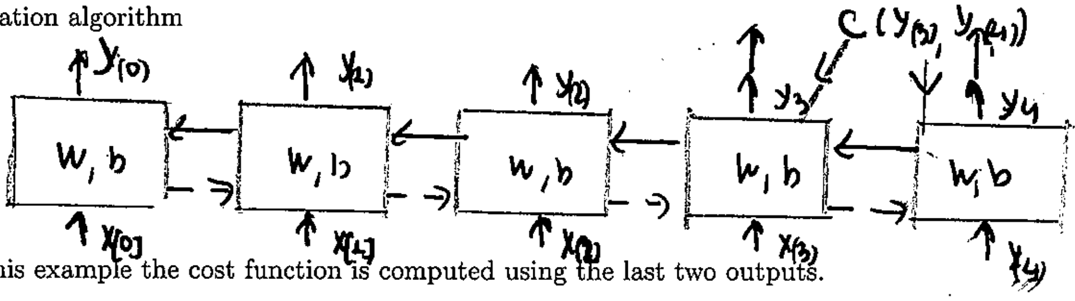
19.2. **Memory cells.** An output at time t is a function of all previous time steps. A part of a neural network some state across time steps is called a memory cell. A state at time t, denoted as $h_{(t)}$ is a function of the state at the previous time step $h_{(t-1)}$ and of the current input $x_{(t)}$

$$h_{(t)} = f(h_{(t-1)}, x_{(t)})$$

For basic cells the output $y_{(t)}$ simply equals to $h_{(t)}$. But in general they can be different, as shown below

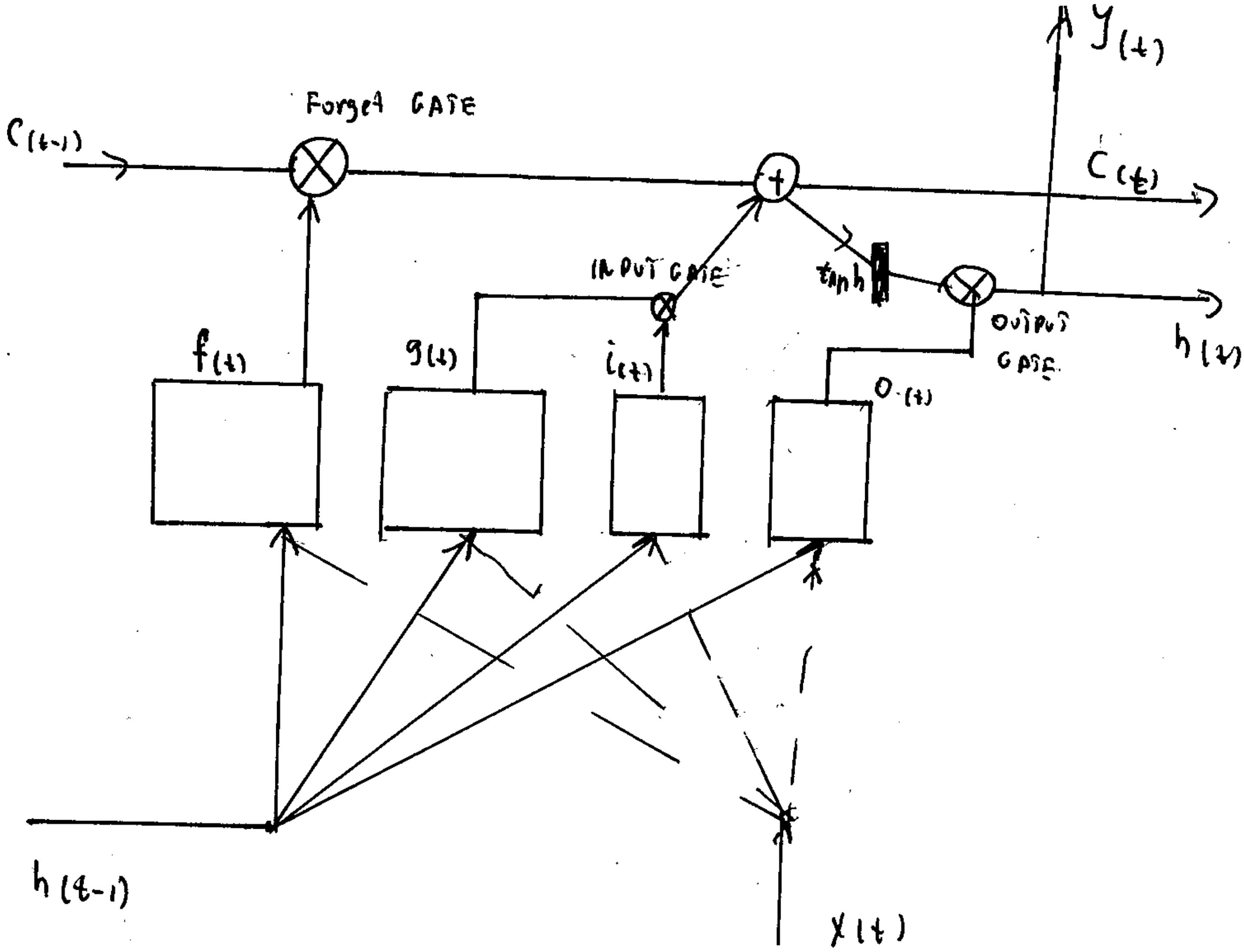


19.3. **Training of RNN.** To train the RNN one can unroll the time and then use the back-propagation algorithm



In this example the cost function is computed using the last two outputs.

19.4. **Long-Sort Term Memory (LSTM) cell.** LSTM cell operates with the long term state $c_{(t)}$ and a short term state $h_{(t)}$. It decides what part of the long-term state to keep, what to throw away, and how. It works as follows



In order to clarify the picture, we write the relevant equations and explain both equations and the picture

$$i_{(t)} = \sigma(W_{x,i}^T x_{(t)} + W_{h,,i}^T h_{(t-1)} + b_i) \tag{6}$$

$$f_{(t)} = \sigma(W_{x,f}^T x_{(t)} + W_{h,f}^T h_{(t-1)} + b_f) \tag{7}$$

$$o_{(t)} = \sigma(W_{x,o}^T x_{(t)} + W_{h,o}^T h_{(t-1)} + b_o) \tag{8}$$

$$g_{(t)} = \tanh(W_{x,g}^T x_{(t)} + W_{h,g}^T h_{(t-1)} + b_g) \tag{9}$$

$$c_{(t)} = (f_{(t)} \otimes c_{(t-1)}) \;\oplus\; (i_{(t)} \otimes g_{(t)}) \tag{10}$$

$$y_{(t)} = h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)}) \tag{11}$$

- There are three gates, denoted by $\otimes$ (the same $\otimes$ is in the equations (10)–(11), they correspond to the element-wise multiplication). The gates are the Forget gate, the Input gate and the Output Gate
- The long term memory $c_{(t-1)}$ runs through the network form left to right. It drops some memories at the Input gate, adds some memories at the addition operation $\oplus$, thus getting modified and then gets out. The modification happens as follows
- The main layer is the one that outputs $g_{(t)}$. It is the layer that we described earlier, when we introduced RNN. It analyses the input $x_{(t)}$, and the previous short term memory state $h_{(t-1)}$. The activation function is the hyperbolic tangent.
- The other three layers control the gates. Their activation functions are logistic ones. If their output is 0, they close the gate, if their output is 1, they open the gate.
- The Forget gate controls (eq. (7)) which part of the long term state should be erased (the first term in the eq. (10)). The Input gate controls (eq. (6)) which part of $g_{(t)}$ should be added to the long term state (the second term in the eq. (10)).
- After the addition operation (eq. (10)) the long term memory is copied, then passed through the tanh function (eq. (10)) and the controlled by the Output Gate (eq. (8)). The result is going out as an output $y_{(t)}$ and as a short-term state $h_{(t)}$. Here they are equal to each other.

20. BOLTZMANN MACHINE

Boltzmann Machines are based on stochastic neurons. Their output is 1 with some probability and 0 otherwise. They use Boltzmann distribution as a probability function.

The probability that the neuron i will output 1 is

$$P(s_i^{nextstep} = 1) = \sigma\left(\frac{\sum_{j=1}^{N}(w_{i.j}s_j + b_1)}{T}\right)$$

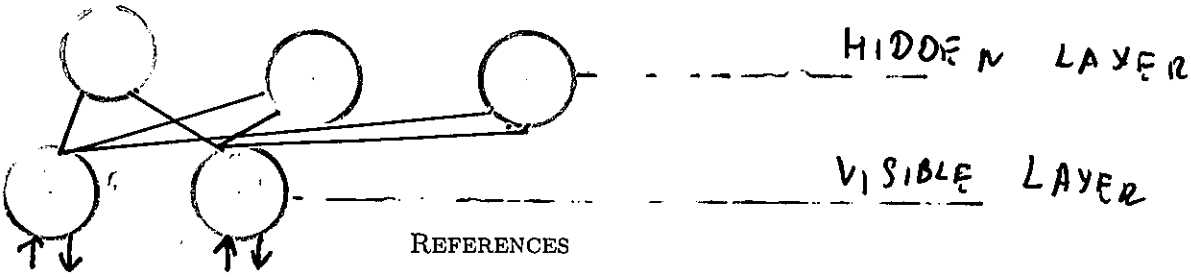
In this equation

- s_j is a state of j-th neuron, which is either 0 or 1
- $w_{i,j}$ are connection weights between the neuron i and the neuron j Therefore, $w_{i,i} = 0$
- b_i is a bias of the neuron i
- N is number of the neurons in the network
- T is a number, called temperature
- σ is a logistic function

The neurons in the network are divided between hidden and visible units. All inputs and outputs are through the visible units only.

The idea behind having hidden units is that, introducing of the "hidden" particles can often simplify the interaction between "visible" particles. In some sense, the hidden particles encode the interaction between the visible ones, and this kind of description of interactions between visible particles turns out to simplify the problem in physics.

20.1. **Restricted Boltzmann Machines.** In the restricted Boltzmann Machines the neurons in the hidden units are connected with visible units. But there are no connections ("interactions") between visible units or between hidden units.



REFERENCES

[1] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani
"An Introduction to Statistical Learning With Applications in R"
Springer Science & Business Media, 2013/06/24 - 426.

[2] Áurélien Géron
"Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow "
O'Reilly, 2019.

[3] Pankaj Mehta, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Marin Bukov, Charles K.Fisher, David J. Schwab
"A high-bias, low-variance introduction to Machine-Learning for physicists,"
arXiv:1803.08823 [physics.comp-ph].