FNNs ,RNNs ,LSTM and BLSTM

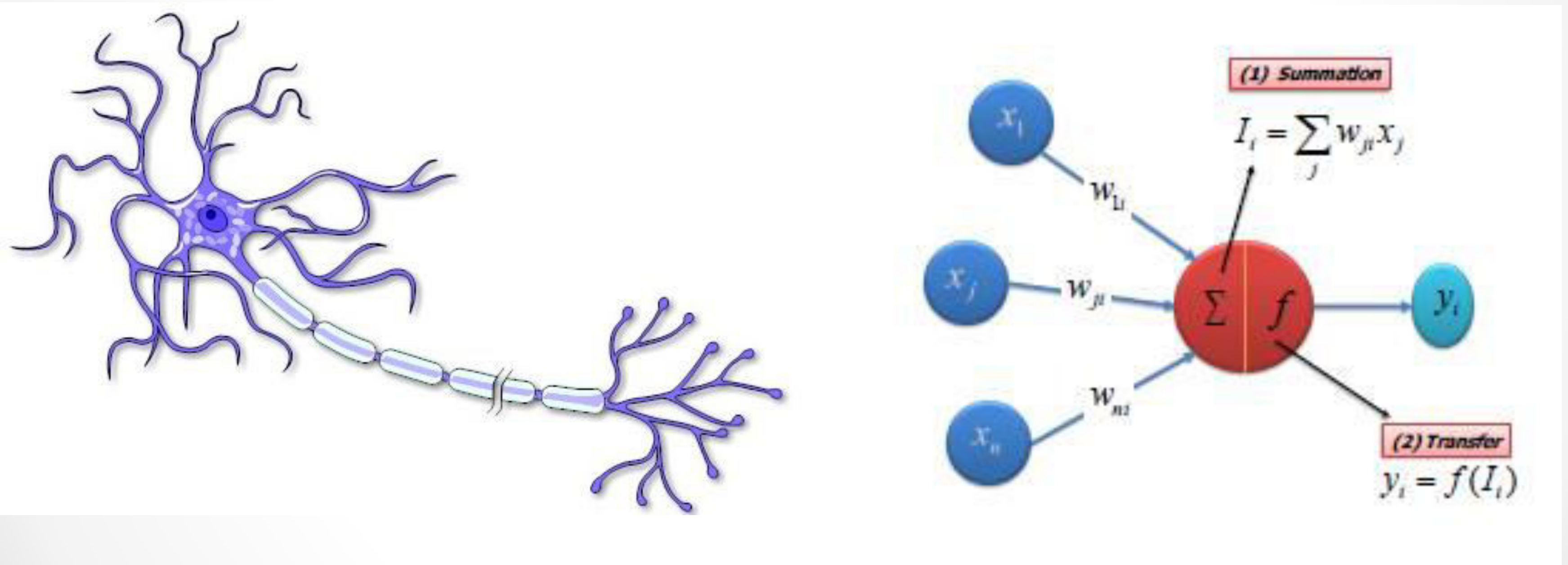
Sudeep Raja

Presentation Summary

- Artificial Neuron Structure
- Feed forward Neural Networks (FNN)
- Recurrent Neural Networks (RNN) and Bidirectional RNNs
- Long Short Term Memory (LSTM) architecture
- Using LSTM and BLSTM from lstm.py from OCROPUS library to recognize Embedded Reber Grammar
- How to Parallelize neural networks?

Artificial Neuron Structure

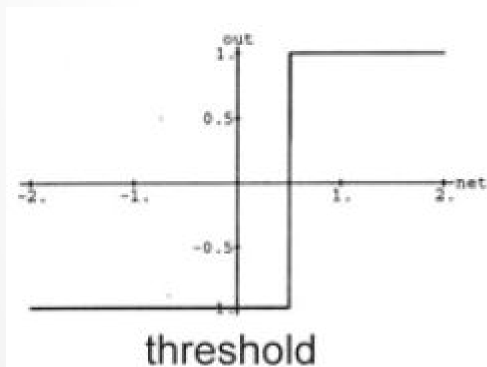
- Artificial neurons were first conceived in the 1940s to mimic the functioning of biological neurons.

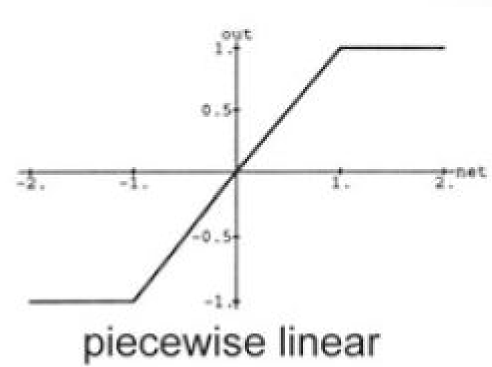


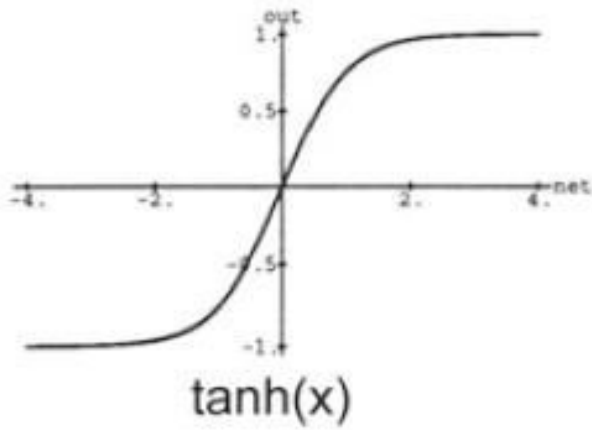
$$a_h = \sum_{i=1}^{I} w_{ih} x_i$$

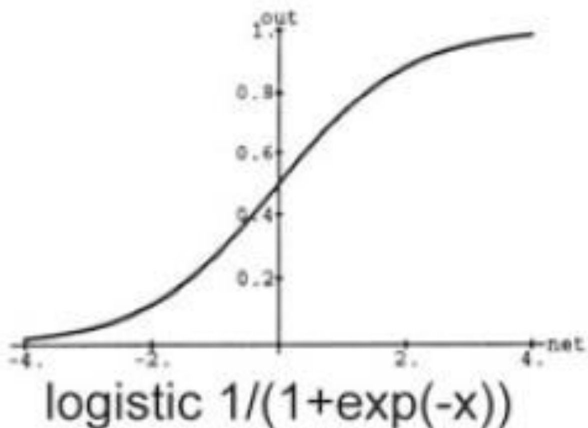
$$b_h = \theta_h(a_h)$$

The transfer function can be a threshold function or a piecewise linear function or a sigmoid function.









$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

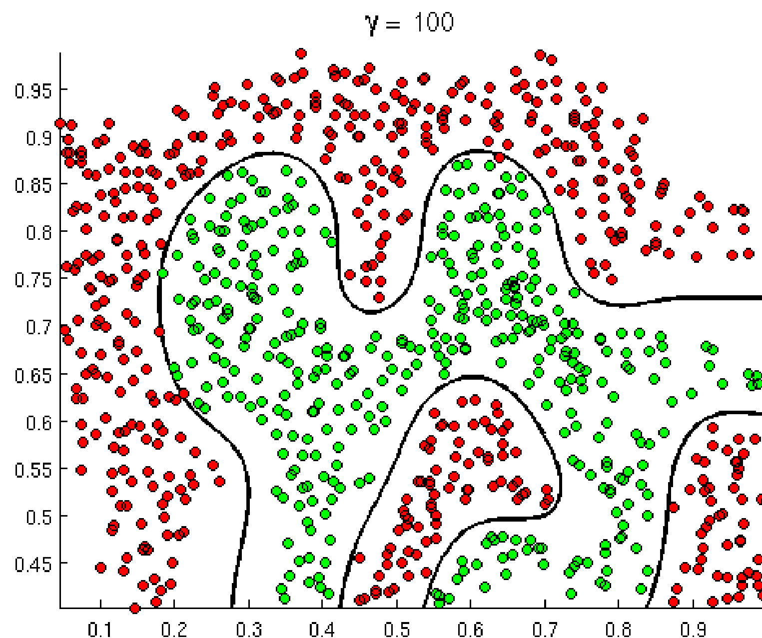
$$tanh(x) = 2\sigma(2x) - 1$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

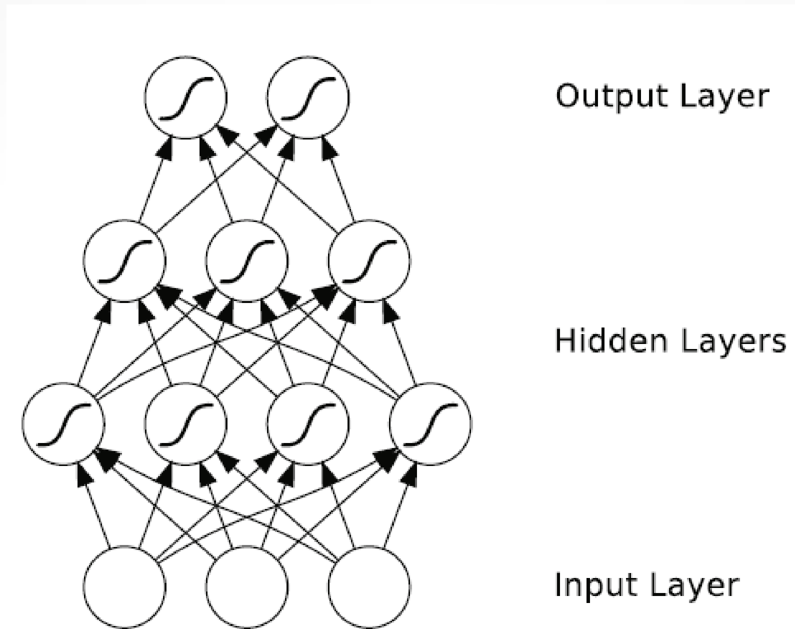
Since they are linearly related , any function computed by a neural network of tanh units can be computed by another network with logistic sigmoid units and vice-versa.

- Tanh and logistic functions are non linear. Nonlinear neural networks are more powerful than linear ones since they can find nonlinear classification boundaries and model nonlinear equations.

- Any combination of linear operators is itself a linear operator, which means that any MLP(Multi Layer Perceptron) with multiple linear hidden layers is exactly equivalent to some other MLP with a single linear hidden layer.



Feed Forward Neural Networks (FNN) Or Multi Layer Perceptions (MLP)



- Input patterns are presented to the *input layer*, then propagated through the *hidden layers* to the *output layer*. This process is known as the *forward pass* of the network.
- Since the output of an MLP depends only on the current input, and not on any past or future inputs, MLPs are more suitable for pattern classification than for sequence labeling.
- Typical classification problems include classifying emails as Spam/Non Spam and Tumors as Benign or Malignant

- For binary classification tasks, the standard configuration is a single unit with a logistic sigmoid activation . Since the range of the logistic sigmoid is the open interval (0, 1), the activation of the output unit can be interpreted as the probability that the input vector belongs to the first class C_1 (and conversely, one minus the activation gives the probability that it belongs to the second class C_2)

$$p(C_1|x) = y = \sigma(a)$$
$$p(C_2|x) = 1 - y$$

- We can also introduce a coding a scheme to represent these equations easily.
- For binary classification, if we use a coding scheme for the target vector z where z = 1 if the correct class is C_1 and z = 0 if the correct class is C_2, we can write:

$$p(z|x) = y^z (1 - y)^{1-z}$$

- For classification problems with K > 2 classes, the convention is to have K output units, and normalize the output activations with the softmax function to obtain the class probabilities.

$$p(C_k|x) = y_k = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}}$$

- A 1-of-K coding scheme represent the target class z as a binary vector with all elements equal to zero except for element k, corresponding to the correct class C_k, which equals one. For example, if K = 6 and the correct class is C_2, z is represented by [0 1 0 0 0 0]. Using this scheme we obtain the following convenient form for the target probabilities:

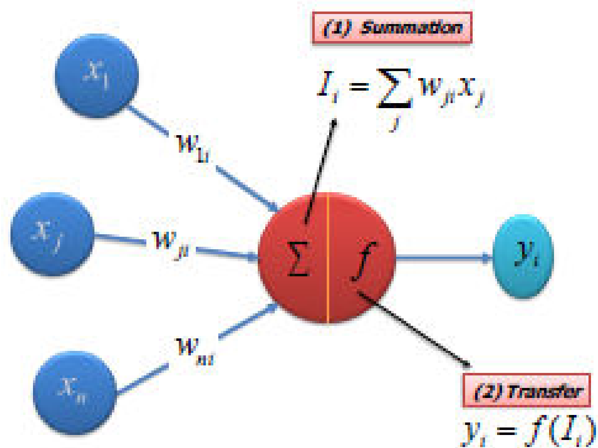
$$p(z|x) = \prod_{k=1}^{K} y_k^{z_k}$$

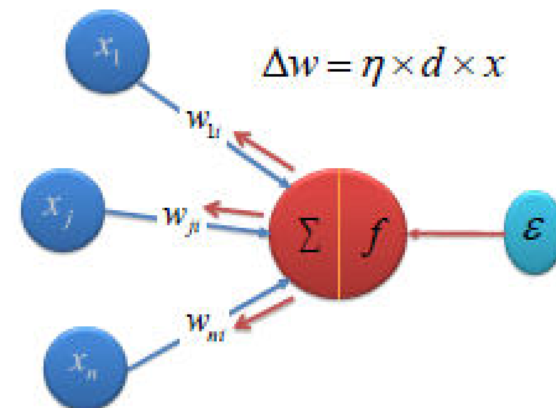
- So to classify the input vector, simply feed in the input vector, activate the network, and choose the class label corresponding to the most active output unit

- Network training has two parts, Forward Pass and Backward Pass
- Since sigmoid functions are differentiable, the network can be trained with gradient descent.

$$\frac{\partial tanh(x)}{\partial x} = 1 - tanh(x)^2$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$





- First we define a loss function for the output target vector z as:

$$\mathcal{L}(x, z) = -\ln p(z|x)$$

- For Binary classification this is:

$$\mathcal{L}(x, z) = (z - 1)\ln(1 - y) - z\ln y$$

- For Multi-class classification this is:

$$\mathcal{L}(x, z) = -\sum_{k=1}^{K} z_k \ln y_k$$

- Then we perform gradient descent through BackPropagation. The basic idea of gradient descent is to find the derivative of the loss function with respect to each of the network weights, then adjust the weights in the direction of the negative slope. This is achieved through repeated use of chain rule.

Output Layer

$$\frac{\partial \mathcal{L}(x,z)}{\partial a_k} = y_k - z_k$$

Hidden Layer

$$\delta_h = \theta'(a_j)\sum_{k=1}^{K} \delta_k w_{hk}$$

$$\delta_j \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}(x,z)}{\partial a_j}$$

Partial differential wrt w_{ij}

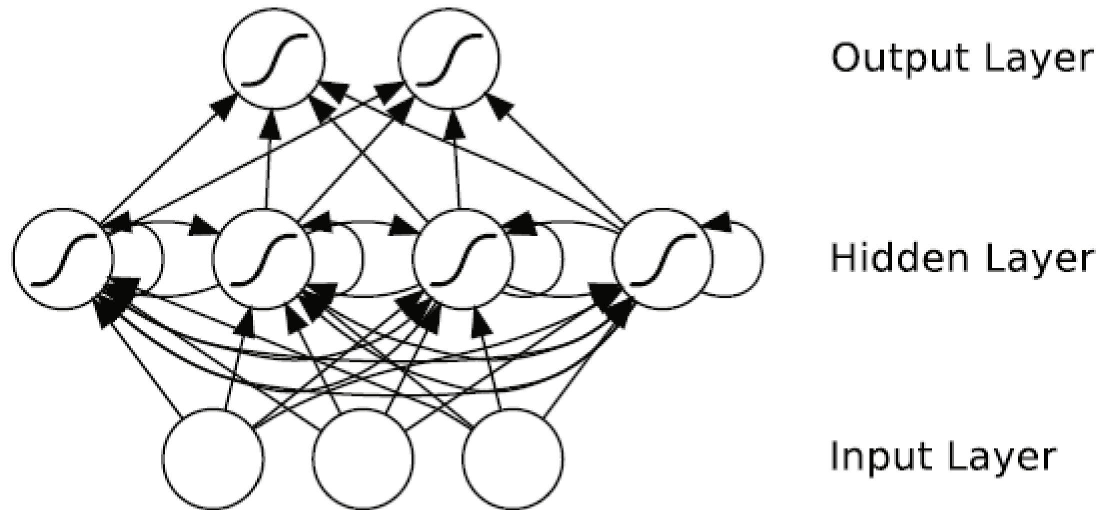
$$\frac{\partial \mathcal{L}(x,z)}{\partial w_{ij}} = \frac{\partial \mathcal{L}(x,z)}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} = \delta_j b_i$$

Δw

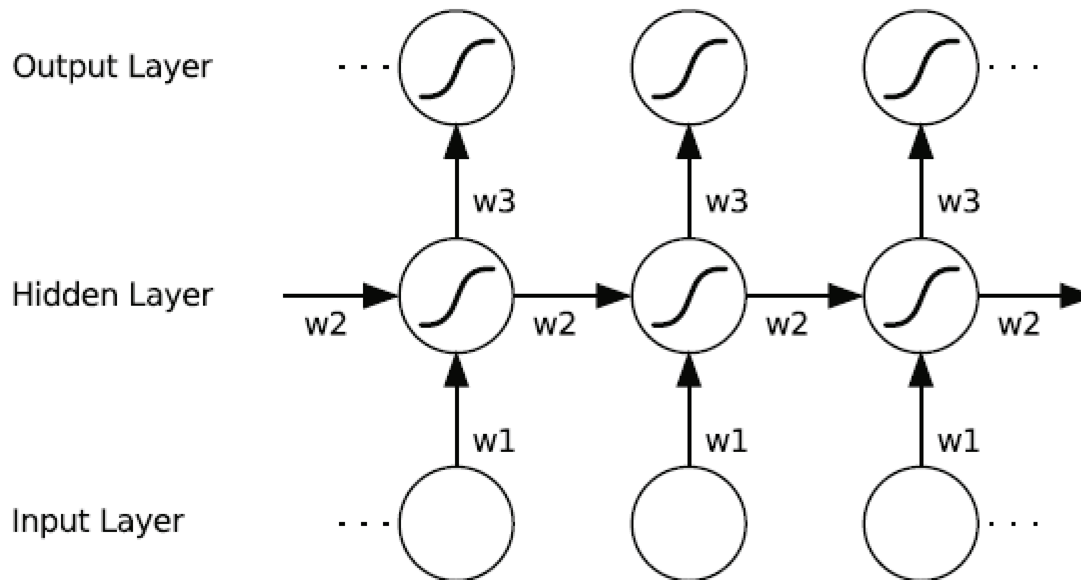
$$\Delta w^n = m\Delta w^{n-1} - \alpha \frac{\partial \mathcal{L}}{\partial w^n}$$

$$w^{n+1} = w^n + \Delta w^n$$

Recurrent Neural Networks (RNN)



- An MLP can only map from input to output vectors, whereas an RNN can in principle map from the entire *history* of previous inputs to each output.
- The idea is that the recurrent connections allow a 'memory' of previous inputs to persist in the network's internal state, and thereby influence the network output.



Consider a length T input sequence x presented to an RNN with I input units, H hidden units, and K output units. Let x^t_i be the value of input i at time t, and let a^t_j and b^t_j be respectively the network input to unit j at time t and the activation of unit j at time t. Denoting the weight from unit i to unit j as w_{ij}

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1} \qquad b_h^t = \theta_h(a_h^t)$$

The forward pass is calculated for a length T input sequence x by starting at t = 1 and iteratively applying the update equations while incrementing t

Backward Propagation Through Time (BPTT)

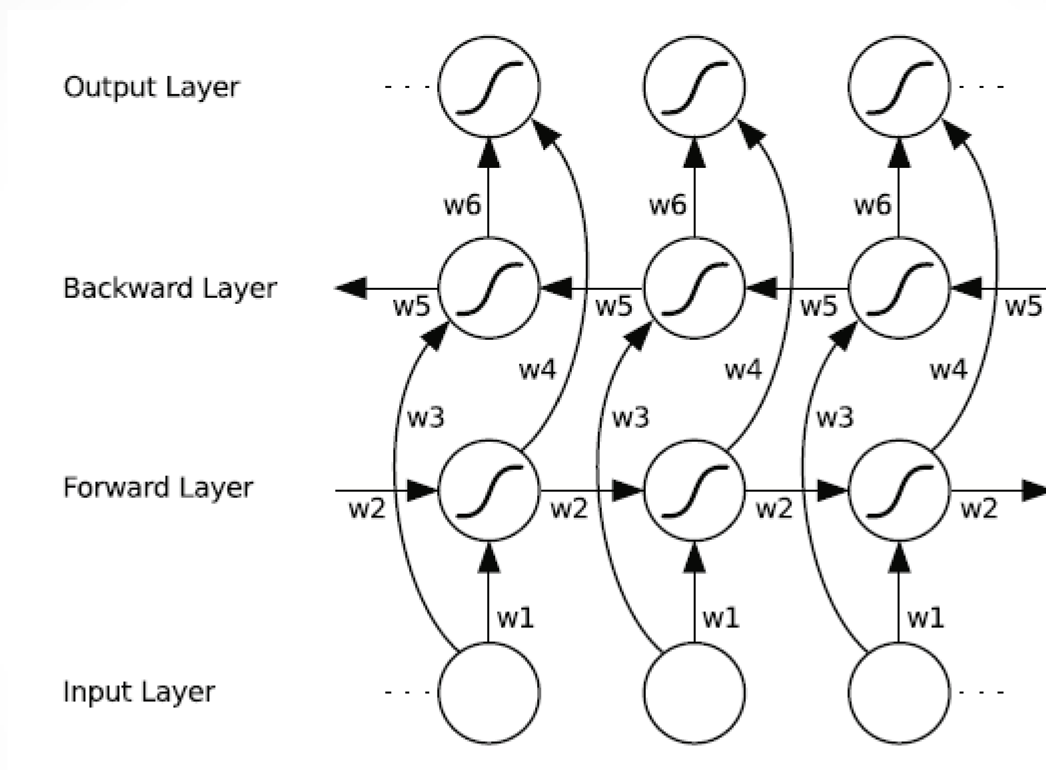
- The time step diagram is the same but with all the arrows inverted.
- The loss function depends on the activation of the hidden layer not only through its influence on the output layer, but also through its influence on the hidden layer at the next time step.

$$\delta_h^t = \theta'(a_h^t)\left(\sum_{k=1}^{K}\delta_k^t w_{hk} + \sum_{h'=1}^{H}\delta_{h'}^{t+1} w_{hh'}\right) \qquad \delta_j^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_j^t}$$

- The complete sequence of δ terms can be calculated by starting at t = T and iteratively applying above equation decrementing t at each step.
- Finally we sum over the whole sequence to get the derivatives with respect to the network weights.

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_{t=1}^{T}\frac{\partial \mathcal{L}}{\partial a_j^t}\frac{\partial a_j^t}{\partial w_{ij}} = \sum_{t=1}^{T}\delta_j^t b_i^t$$

Bidirectional RNNs



The forward pass for the BRNN hidden layers is the same as for a unidirectional RNN, except that the input sequence is presented in opposite directions to the two hidden layers, and the output layer is not updated until both hidden layers have processed the entire input sequence

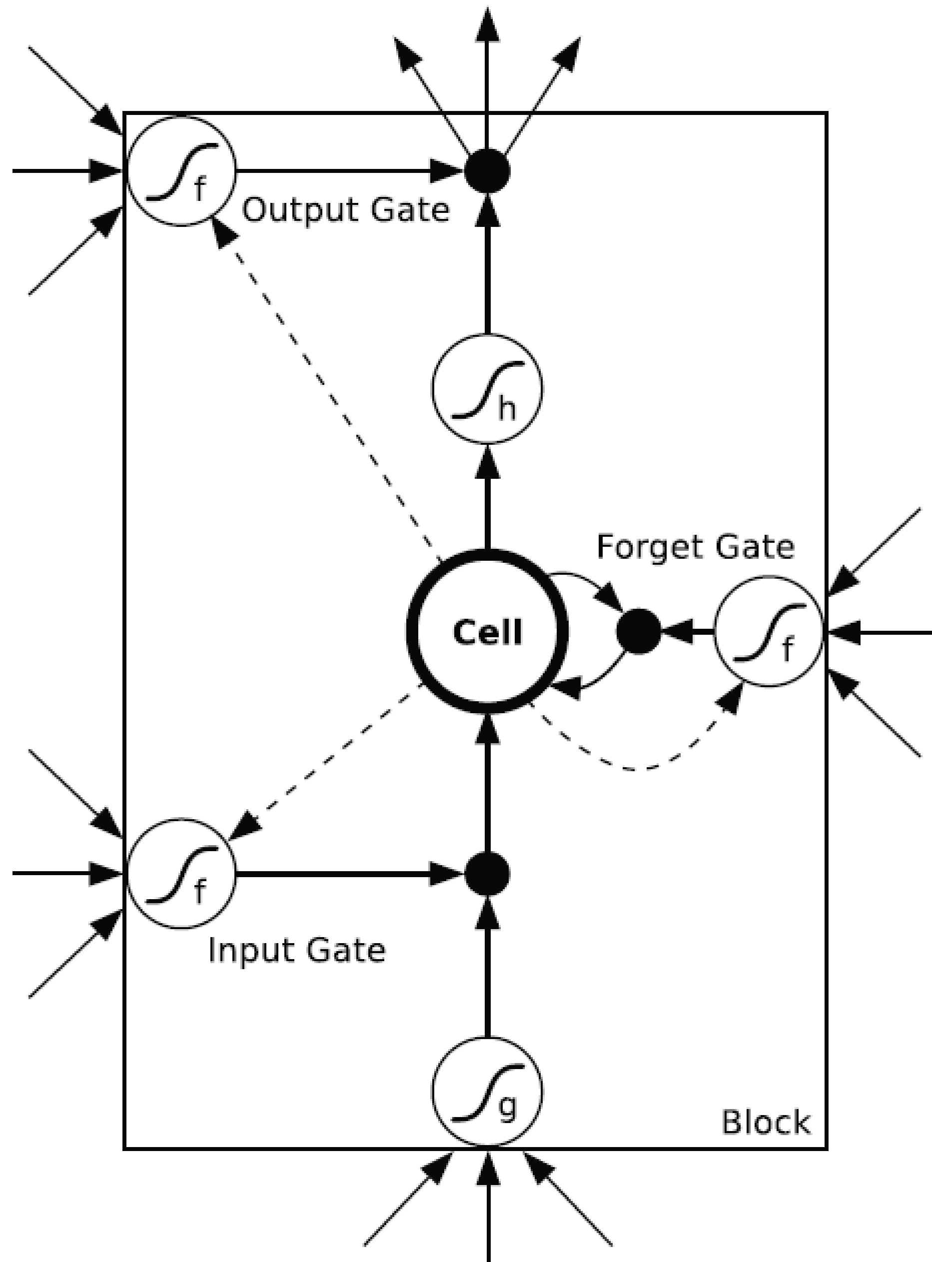
for $t = 1$ to T **do**
    Forward pass for the forward hidden layer, storing activations at each time step
for $t = T$ to 1 **do**
    Forward pass for the backward hidden layer, storing activations at each time step
for all t, in any order **do**
    Forward pass for the output layer, using the stored activations from both hidden layers

Similarly, the backward pass proceeds as for a standard RNN trained with BPTT, except that all the output layer δ terms are calculated first, then fed back to the two hidden layers in opposite directions

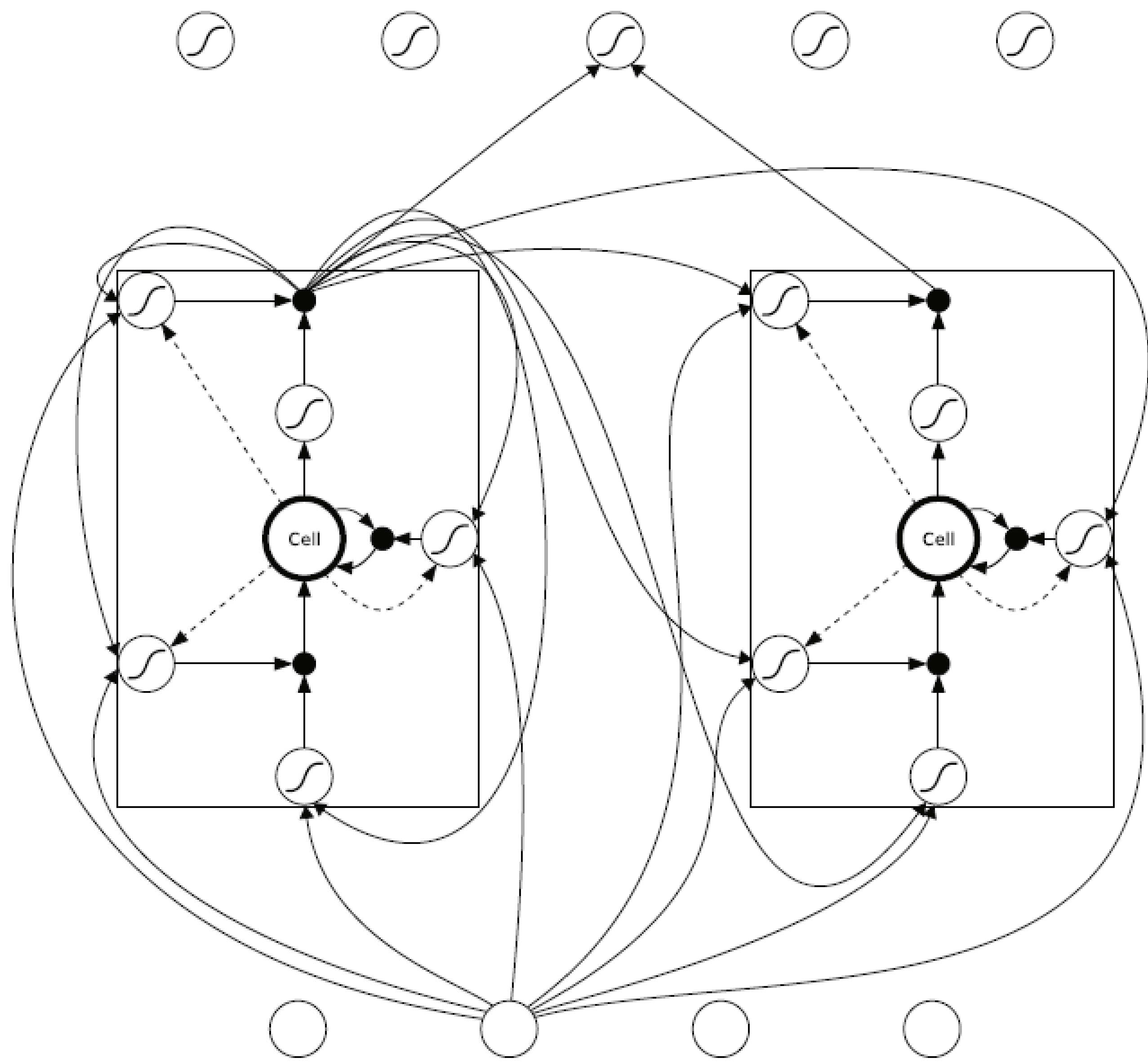
for all t, in any order do
    Backward pass for the output layer, storing δ terms at each time step
for t = T to 1 do
    BPTT backward pass for the forward hidden layer, using the stored δ terms from the output layer
for t = 1 to T do
    BPTT backward pass for the backward hidden layer, using the stored δ terms from the output layer

Long Short Term Memory (LSTM) architecture

- RNNs suffer from the problem of *Vanishing Gradients*
- The sensitivity of the network decays over time as new inputs overwrite the activations of the hidden layer, and the network ‘forgets’ the first inputs.
- This problem is remedied by using LSTM blocks instead of sigmoid cells in the hidden layer.
- LSTM blocks can choose to retain their memory over arbitrary periods of time and also ‘forget ‘ if necessary.



Output Gate
f
h
Forget Gate
Cell
f
f
Input Gate
g
Block


Cell
Cell

- w_{ij} is the weight of the connection from unit i to unit j.
- The network input to unit j at time t is denoted a^t_j and activation of unit j at time t is b^t_j .
- The subscripts ι, ϕ and ω refer respectively to the input gate, forget gate and output gate of the block.

- The subscripts c refers to one of the C memory cells. The peephole weights from cell c to the input, forget and output gates are denoted $w_{c\iota}$, $w_{c\phi}$ and $w_{c\omega}$ respectively.

- s^t_c is the state of cell c at time t (i.e. the activation of the linear cell unit).

- f is the activation function of the gates, and g and h are respectively the cell input and output activation functions.

- Let I be the number of inputs, K be the number of outputs and H be the number of cells in the hidden layer

As with standard RNNs the forward pass is calculated for a length T input sequence **x** by starting at $t = 1$ and applying the update equations while incrementing t, and the BPTT backward pass is calculated by starting at $t = T$, and calculating the unit derivatives while decrementing t to one. The final weight derivatives are found by summing over the derivatives at each time step.

Input Gates

$$a_\iota^t = \sum_{i=1}^{I} w_{i\iota} x_i^t + \sum_{h=1}^{H} w_{h\iota} b_h^{t-1} + \sum_{c=1}^{C} w_{c\iota} s_c^{t-1}$$

$$b_\iota^t = f(a_\iota^t)$$

Forget Gates

$$a_\phi^t = \sum_{i=1}^{I} w_{i\phi} x_i^t + \sum_{h=1}^{H} w_{h\phi} b_h^{t-1} + \sum_{c=1}^{C} w_{c\phi} s_c^{t-1}$$

$$b_\phi^t = f(a_\phi^t)$$

Cells

$$a_c^t = \sum_{i=1}^{I} w_{ic} x_i^t + \sum_{h=1}^{H} w_{hc} b_h^{t-1}$$

$$s_c^t = b_\phi^t s_c^{t-1} + b_\iota^t g(a_c^t)$$

Output Gates

$$a_\omega^t = \sum_{i=1}^{I} w_{i\omega} x_i^t + \sum_{h=1}^{H} w_{h\omega} b_h^{t-1} + \sum_{c=1}^{C} w_{c\omega} s_c^t$$

$$b_\omega^t = f(a_\omega^t)$$

Cell Outputs

$$b_c^t = b_\omega^t h(s_c^t)$$

$$\epsilon_c^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial b_c^t} \qquad \epsilon_s^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial s_c^t}$$

Cell Outputs

$$\epsilon_c^t = \sum_{k=1}^{K} w_{ck}\delta_k^t + \sum_{h=1}^{H} w_{ch}\delta_h^{t+1}$$

Output Gates

$$\delta_\omega^t = f'(a_\omega^t)\sum^{C} h(s_c^t)\epsilon_c^t$$

States

$$\epsilon_s^t = b_\omega^t h'(s_c^t)\epsilon_c^t + b_\phi^{t+1}\epsilon_s^{t+1} + w_{c\iota}\delta_\iota^{t+1} + w_{c\phi}\delta_\phi^{t+1} + w_{c\omega}\delta_\omega^t$$

Cells

$$\delta_c^t = b_\iota^t g'(a_c^t)\epsilon_s^t$$

Forget Gates

$$\delta_\phi^t = f'(a_\phi^t)\sum_{c=1}^{C} s_c^{t-1}\epsilon_s^t$$

Input Gates

$$\delta_\iota^t = f'(a_\iota^t)\sum_{c=1}^{C} g(a_c^t)\epsilon_s^t$$

Computational Complexity

- In Forward pass, we evaluate the weighted sum of inputs from previous layer to the next.
- In Backward pass, we calculate the errors and modify these weights.
- Hence for all networks thus far, FNNs, RNNs, BRNNs, LSTM and BLSTM computational complexity is $O(W)$ i.e., the total number of edges in the network.

- For FNN: $W = IH + HK$

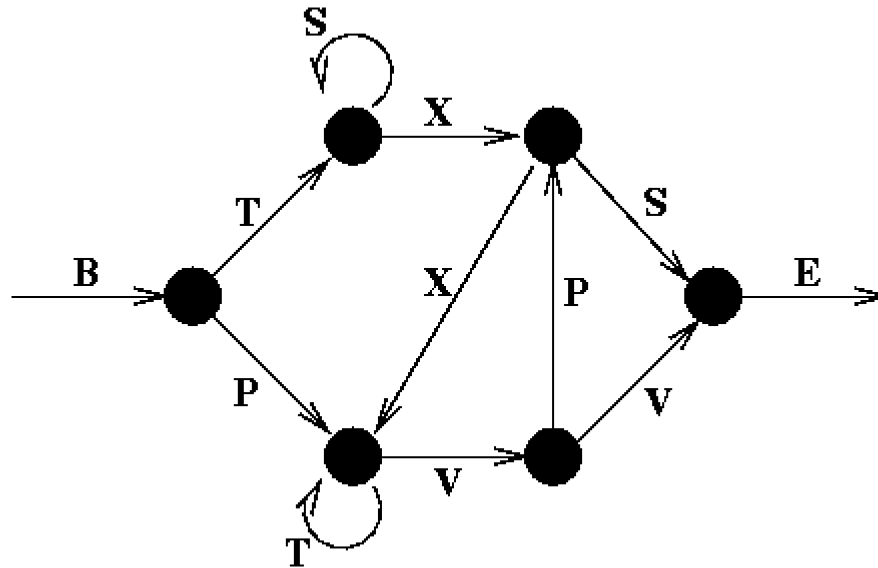
- For RNN: $W = IH + H^2 + HK$

- For LSTM : $W = 4IH + 4H^2 + 3H + HK$

- For Bidirectional networks, $W = 2$(W for unidirectional network)

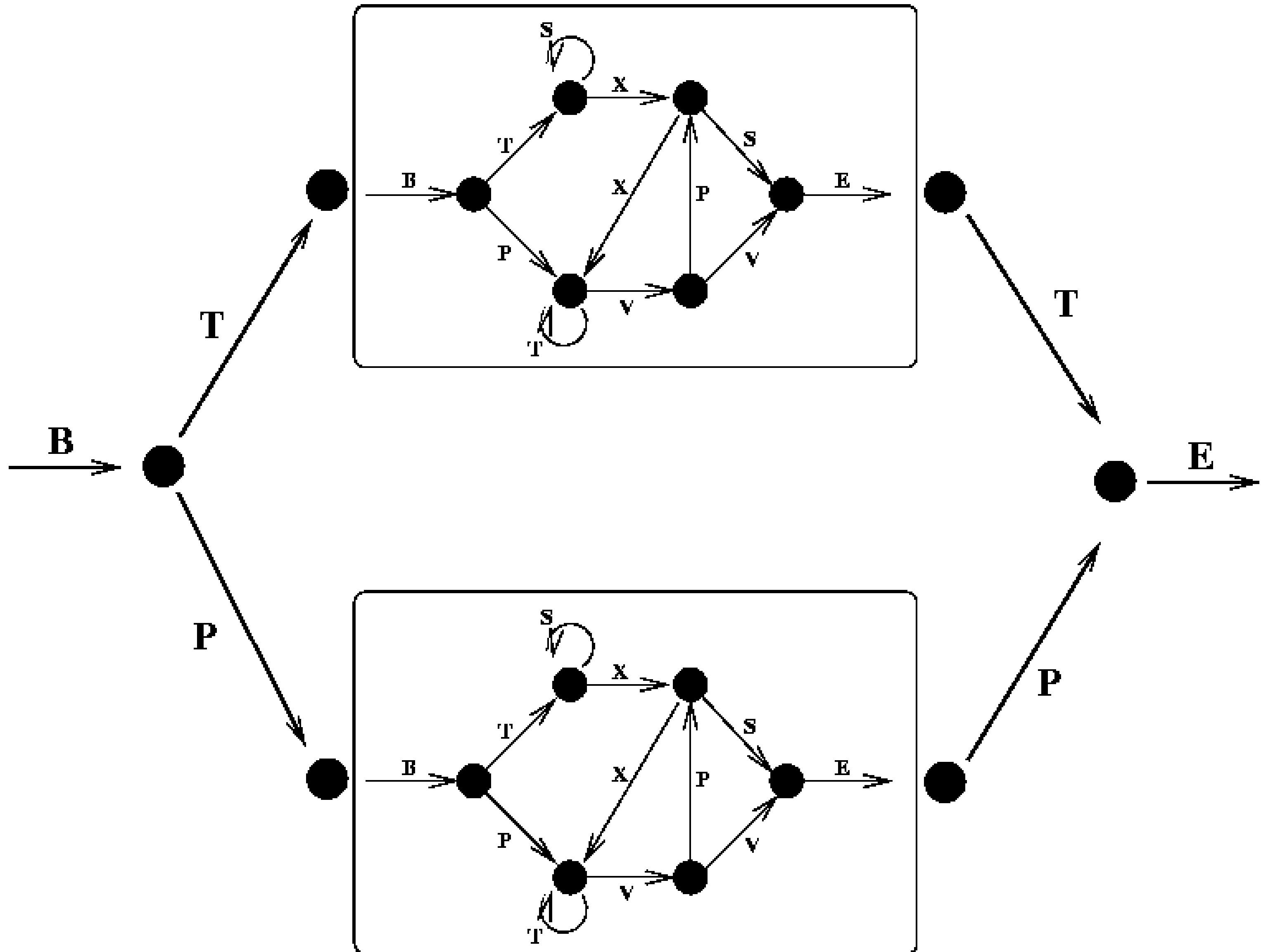
- In most applications , I<< H~ K
- Take Japanese character recognition for instance
- Total number of kanji characters ~ 85,000
- But most of them are archaic and 13,000 are listed in Japanese Industrial Standards for kanji of which 5000 are regularly used
- So K=5,000
- Number of features extracted for each time step is at max 30.
- Number of hidden cells used are of the order of the total classes (there is no definite answer to how many hidden cell to use , find optimal number by trial and error)
- So computational complexity is ~ O (KH)

Reber Grammar



- We start at **B**, and move from one node to the next, adding the symbols we pass to our string as we go. When we reach the final **E**, we stop. If there are two paths we can take, e.g. after **T** we can go to either **S** or **X**, we randomly choose one (with equal probability).
- An S can follow a T, but only if the immediately preceding symbol was a B. A V or a T can follow a T, but only if the symbol immediately preceding it was either a T or a P.
- In order to know what symbol sequences are legal, therefore, any system which recognizes reber strings must have some form of memory, which can use not only the current input, but also fairly recent history in making a decision. RNNs can recognize a Reber Grammar.

Embedded Reber Grammar



- Using this grammar two types of strings are generated: one kind which is made using the top path through the graph: **BT<reber string>TE** and one kind made using the lower path: **BP<reber string>PE**.

- To recognize these as legal strings, and learn to tell them from illegal strings such as **BP<reber string>TE**, a system must be able to remember the second symbol of the series, regardless of the length of the intervening input ,and to compare it with the second last symbol seen.

- A simple RNN can no longer solve this task, but LSTM can solve it with after about 10000 training examples . BLSTM can solve it after 1000 training examples

Programming example

- Here we use an implementation of LSTM in OCROPUS which is an open source document analysis and OCR system.
- OCROPUS is written in Python, NumPy, and SciPy focusing on the use of large scale machine learning for addressing problems in document analysis.
- https://code.google.com/p/ocropus/

http://www.felixgers.de/SourceCode_Data.html

- Input is (T x 7) matrix which represents the present move and output is also a (T x 7) matrix which represents the possible next moves
- Encoding: B T P S X V E
- Now can we present the output directly as target vector ?

```
1 0 0 0 0 0 0   0 1 1 0 0 0 0
0 1 0 0 0 0 0   1 0 0 0 0 0 0
1 0 0 0 0 0 0   0 1 1 0 0 0 0
0 0 1 0 0 0 0   0 1 0 0 0 1 0
0 0 0 0 0 1 0   0 0 1 0 0 1 0
0 0 0 0 0 1 0   0 0 0 0 0 0 1
0 0 0 0 0 0 1   0 1 0 0 0 0 0
0 1 0 0 0 0 0   0 0 0 0 0 0 1
```

- The answer is NO!
- One might say we can create 7 output classes and take the top 2 most active outputs as the answer.

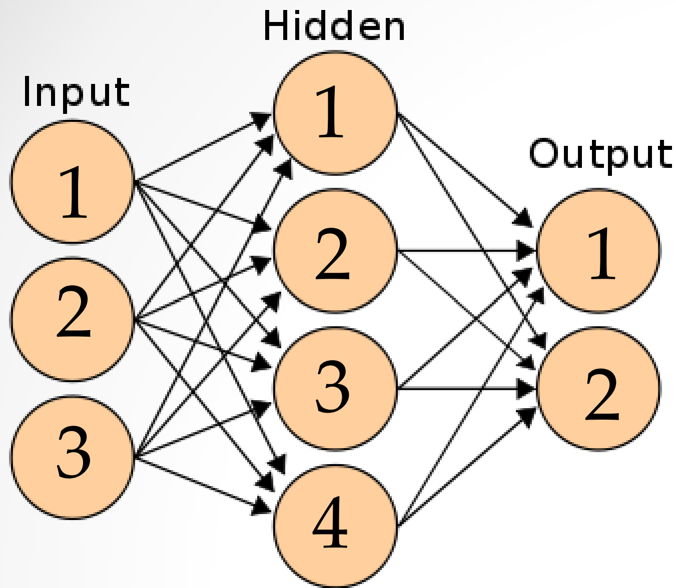
- But training wont work as multi class output layer is softmax which follows a 1-of-K encoding scheme

- So we need to identify the different output classes:

- TP , SX ,TV ,PV ,B ,T ,P and E are all possible next moves. So total of 8 output classes not 7.

		encoding	
• TP:	0110000		10000000
• SX:	0001100		01000000
• TV:	0100010		00100000
• PV:	0010010		00010000
• B:	1000000		00001000
• T:	0100000		00000100
• P:	0010000		00000010
• E:	0000001		00000001

Parallelizing FNNs



I H K

$$B_H = f(A_H) \qquad A_H = W_{IH}X_I + b_I$$

$$Y_K = f(A_K) \qquad A_K = W_{HK}B_H + b_H$$

- The weights between two layers can be represented by a matrix
- W_{IH} is a (HxI) matrix and input X_I is a (Ix1) vector
- Their product is a (HX1) vector A_H representing the activations of the 4 Hidden cells.
- Applying the sigmoid function to individual elements of this vector, we get a vector B_H representing the outputs of Hidden Layer.
- This is the input to the next layer.

- In BackPropagation, first we find error vector (Kx1) at output layer and then the error matrix (KxH)of the edges between hidden and output layer

$$\frac{\partial L}{\partial A_K} = Y_K - Z_K \qquad \frac{\partial L}{\partial W_{HK}} = \frac{\partial L}{\partial A_K} B^T{}_H \qquad \frac{\partial L}{\partial b_H} = \frac{\partial L}{\partial A_K}$$

- Next we find error vector (Hx1)at the hidden layer and find the error matrix (HxI) for the edges between input and output layer

$$\frac{\partial L}{\partial A_H} = \acute{f}(A_H)\ .^*\ (W_{HK}{}^T \frac{\partial L}{\partial A_K}) \qquad \frac{\partial L}{\partial W_{IH}} = \frac{\partial L}{\partial A_H} X^T{}_I \qquad \frac{\partial L}{\partial b_I} = \frac{\partial L}{\partial A_H}$$

- Weight updates are done after Back Propagation

Parallelizing RNNs

- Consider an input X having T time steps
- The Weight matrices of the network change only after the backward pass, so we can find the excitations of a layer for all T time steps one after the other, instead of completely propagating through the layers for every single time step.

- $[W_{IH} \mid W_{HH}]$ is a (H x (I+H)) matrix and input $[X^t_I \mid B^{t-1}_H]$ is a ((I+H)x1) vector. Their product is A^t_H and applying sigmoid to this we get B^t_H. This is done for t = 1 to T.

- Now we have a matrix B_H which has dimensions (HxT)

- Now the product $W_{HK} B_H$ gives the output (KxT) for all T time steps

- For Backprop, the output error δ_K (KxT) for all time steps is $Y_K - Z_K$, so error matrix (KxH) is the product $\delta_K B'_H$

- For the hidden layer error δ_H (KxT), repeatedly find the vector from t =T to 1

$$\delta^t_H = f'(A^t_H) .* \left([W'_{IH} \mid W'_{HH}] \begin{bmatrix} \delta^t_K \\ \delta^{t+1}_H \end{bmatrix} \right)$$

- After finding the complete error over time T find the product $\delta_H X'_I$ hang dimensions (HxI)

Citations:

1. Supervised Sequence Labelling with Recurrent Neural Networks - Alex Graves
2. Long Short-Term Memory in Recurrent Neural Networks - Felix Gers
3. Long Short-Term Memory - Sepp Hochreiter and Jürgen Schmidhuber
4. The OCRopus Open Source OCR System - Thomas M. Breuel
5. Parallel Implementations of Recurrent Neural Network Learning -Uros Lotric and Andrej Dobnikar