5 – Introduction to Neural Networks

Computational Intelligence for the Internet of Things (2019-20)

João Paulo Carvalho
joao.carvalho@inesc-id.pt

INESC-ID
Instituto Superior Técnico, Universidade de Lisboa

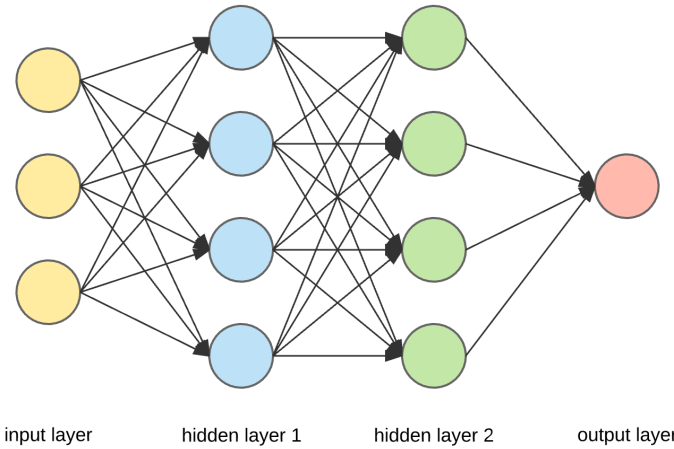
Neural Networks

- What are NN
- The Perceptron
- NN: Multilayer perceptron
- Recurrent Neural Nets
- Advanced Topics
- Conclusions

What are NN?

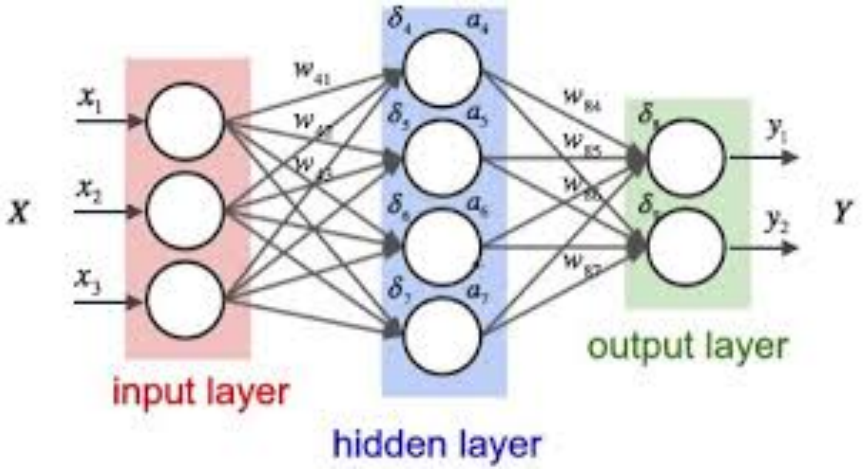
What is an Artificial Neural Network?

- An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information
- The key element of this paradigm is the novel structure of the information processing system:
  - A large number of highly interconnected processing elements (neurons) working in unison to solve specific problems



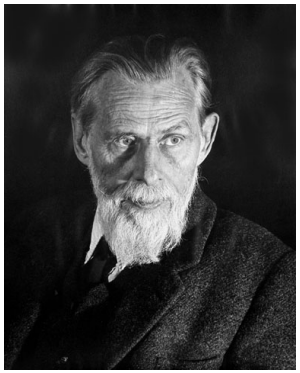
What is an Artificial Neural Network? (II)

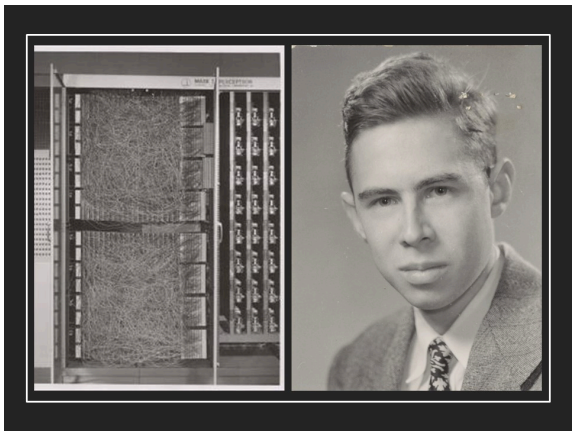
- ANNs, like people, learn by example:
  - An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process
- Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones
  - This is true of ANNs as well!



Historical Background

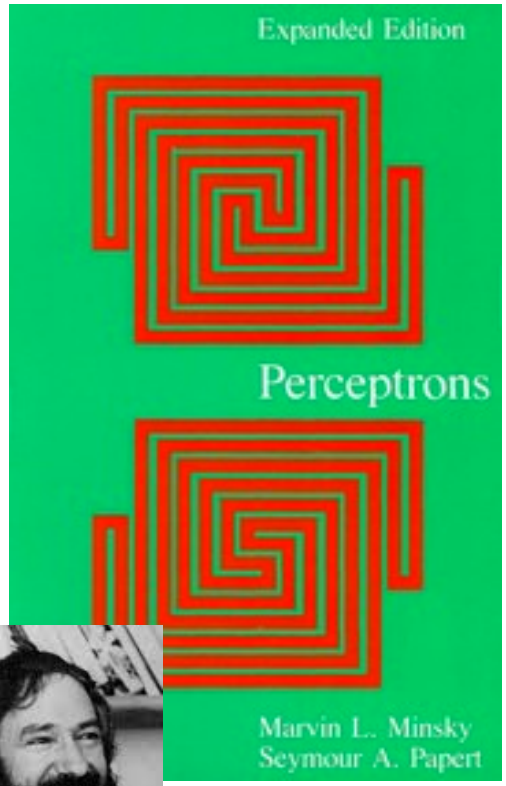
- The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pits
  - The technology available at that time did not allow them to do too much
- In the 50's Rosenblatt introduced the perceptron
  - It could be trained, and the learning algorithm converged as long as the perceptron could solve the problem





Historical Background (II)

- 1959 – ADALINE and MADALINE:
  - First commercial applications of NN (eliminate echoes in phone lines – still in use today)
- 1969 – Minsky and Papert published a book in which they proof perceptron limitations and summed up a general feeling of frustration against neural networks among researchers
- NN research went into decline through the 70's and early 80's



Historical Background (III)

- In the mid 80's, aided by computer development, Yann LeCun, D. B. Parker and others (e.g. L. Borges de Almeida), independently discovered backpropagation, a learning algorithm for multi-layer networks that could solve perceptron's limitations, and techniques to accelerate learning that made practical implementations possible using the available processing power

?

Historical Background

A Theoretical Framework for Back-Propagation *

Yann le Cun †
Department of Computer Science, University of Toronto
Toronto, Ontario, M5S 1A4. CANADA.

Acknowledgments

This work has been supported by a grant from the Fyssen foundation and a grant from the Sloan foundation. The author wishes to thank Geoff Hinton, Sue Becker, Mike Mozer, Steve Nowlan and Didier Georges for helpful discussions. Special thanks to Léon-Yves Bottou, the neural network simulator SN is the result of our collaboration.

References

[Almeida, 1987] Luis B. Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Proceedings ICNN 87*, San Diego, 1987. IEEE, IEEE.

[Athans and Falb, 1966] M. Athans and P. L. Falb. *Optimal Control Theory*. McGraw Hill, 1966.

[Bryson and Ho, 1969] A. E. Jr. Bryson and Yu-Chi Ho. *Applied Optimal Control*. Blaisdell Publishing Co., 1969.

[Fogelman-Soulié *et al.*, 1986] F. Fogelman-Soulié, P. Gallinari, Y. le Cun, and S. Thiria. Automata networks and artificial intelligence. Technical report, Laboratoire de Dynamique des Reseaux, 1986.

[Fogelman-Soulié *et al.*, 1987] F. Fogelman-Soulié, P. Gallinari, Y. le Cun, and S. Thiria. Automata networks and artificial intelligence. In *Automata networks in computer science, theory and applications*, pages 133–186. Princeton University Press, 1987.

[le Cun, 1985] Y. le Cun. A learning scheme for asymmetric threshold networks. In *Proceedings of Cognitiva 85*, pages 599–604, Paris, France, 1985.

[le Cun, 1986] Y. le Cun. Learning processes in an asymmetric threshold network. In F. Fogelman-Soulié E. Bienenstock and G. Weisbuch, editors, *Disordered systems and biological organization*, pages 233–240, Les Houches, France, 1986. Springer-Verlag.

[le Cun, 1987] Y. le Cun. *Modèles Connexionnistes de l'Apprentissage*. PhD thesis, Université Pierre et Marie Curie, Paris, France, 1987.

[Noton, 1965] A. R. M. Noton. *Introduction to Variational Methods in Control Engineering*. Pergamon Press, 1965.

[Parker, 1985] D. B. Parker. Learning-logic. Technical report, TR-47, Sloan School of Management, MIT, Cambridge, Mass., April 1985.

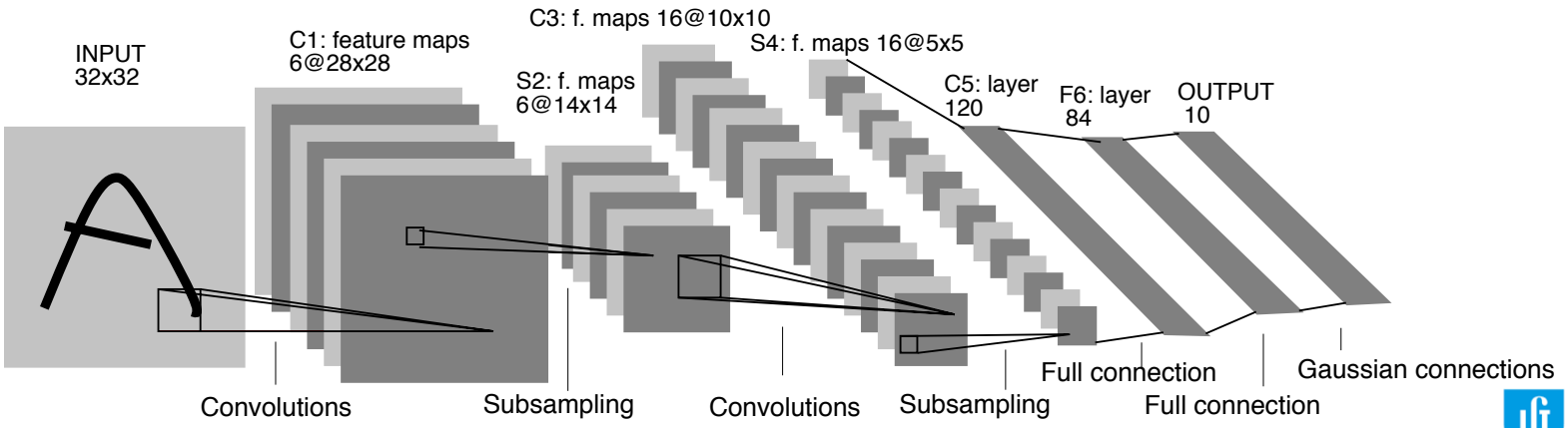
[Pineda, 1987] F.J. Pineda. Generalization of back propagation to recurrent and higher order neural networks. In *Proceedings of IEEE Conference on Neural Information Processing Systems*, Denver, Colorado, November 1987. IEEE.

[Rumelhart *et al.*, 1986] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel distributed processing: Explorations in the microstructure of cognition*, volume I. Bradford Books, Cambridge, MA, 1986.

[Werbos, 1974] P. Werbos. *Beyond Regression*. Phd thesis, Harvard University, 1974.

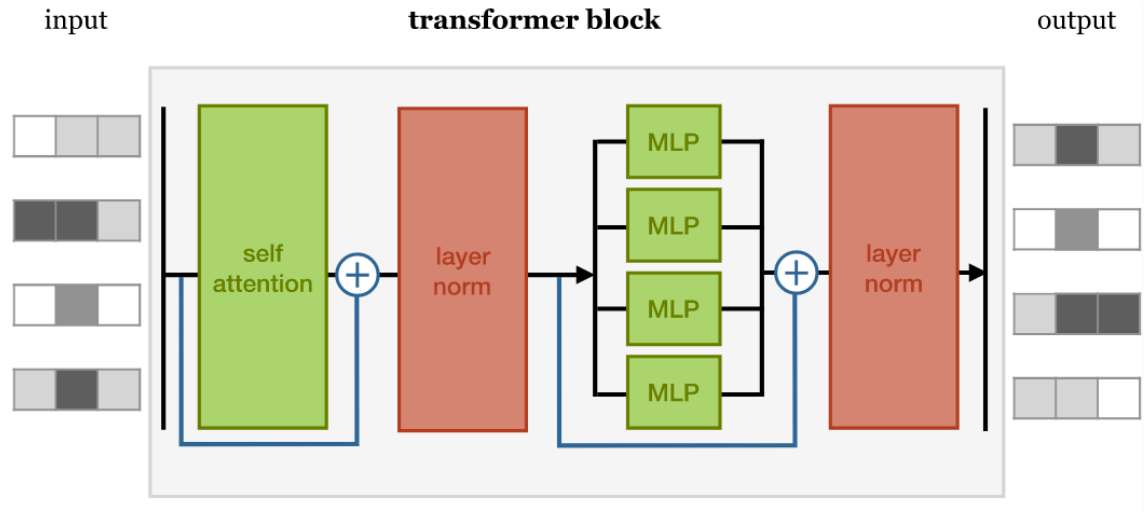
Historical Background (IV)

- Many in developments in research and applications during the 90's:
  - 1997 – A recurrent neural network framework, Long Short-Term Memory (LSTM) was proposed by Schmidhuber & Hochreiter
  - 1998 – Yann LeCun published Gradient-Based Learning Applied to Document Recognition (first deep learning convolutional architecture)



Historical Background (V)

- Plateau and decline in the 00's until…
- …the 10's, when the combination of processing power, computational mathematics and new fast learning algorithms [Hinton et. al 2006], enabled the blooming of complex deep recurrent architectures



Why use neural networks?

- Neural networks, with their ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or traditional computer techniques
- ...and they are also able to perform tasks that are trivial to humans but very difficult using other computational techniques, such as, e.g., handwritten character recognition
- A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze
- This expert can then be used to provide projections given new situations of interest and answer "what if" questions

Why use neural networks? (II)

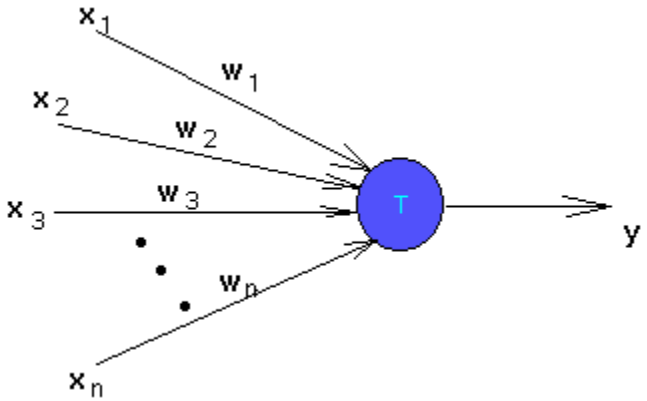
- Other advantages include:
  - Adaptive learning: an ability to learn how to do tasks based on the data given for training or initial experience;
  - Self-Organization: an ANN can create its own organization or representation of the information it receives during learning time;
  - Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices can be designed and manufactured to take advantage of this capability;
  - Fault Tolerance via Redundant Information Coding: partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with significant network damage

The Perceptron

The Perceptron

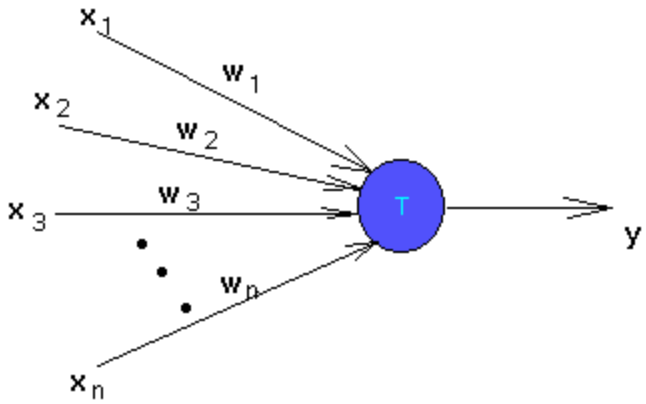
- The perceptron is a very simple model that consists of a single trainable neuron
  - Trainable means that its threshold and input weights are modifiable

$x_1, x_2, ..., x_n$ are inputs (real or boolean values)
y is the boolean output
$w_1, w_2, ..., w_n$ are weights of the edges (real values)
T is the threshold (real value)



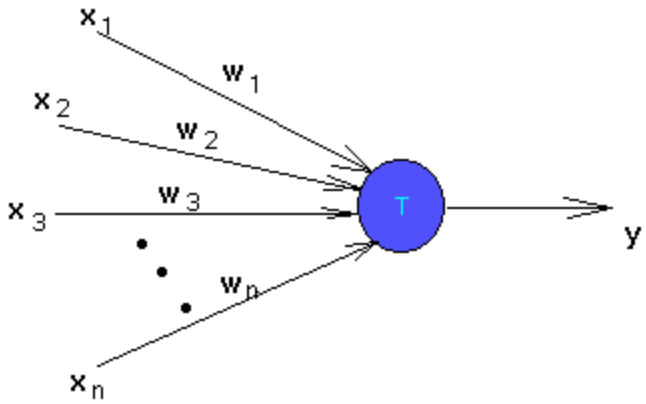
- The output is:
  - '1' if $w_1 x_1 + w_2 x_2 + ... + w_n x_n$ is greater than the threshold T;
  - '0' otherwise

The Perceptron (II)



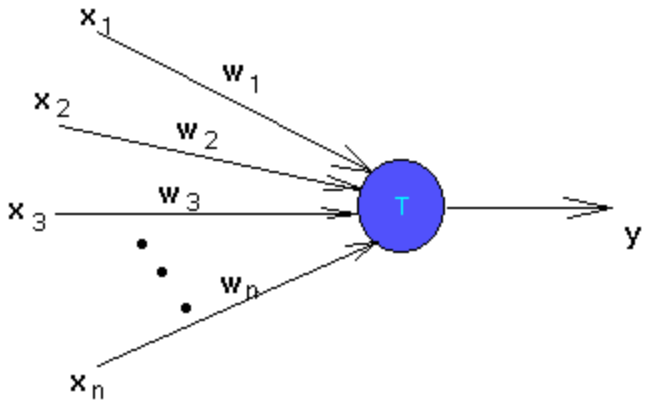
- It is possible to train the perceptron to respond to certain inputs with certain desired outputs
- After the training period, it should be able to give reasonable outputs for any kind of input
  - If it wasn't trained for that input, then it should try to find the best possible output depending on how it was trained
- During the training period we present the perceptron with inputs one at a time and see what output it gives:
  - If the output is wrong, we tell it that it has made a mistake
  - It should then change its weights and/or threshold properly to avoid making the same mistake later

Examples



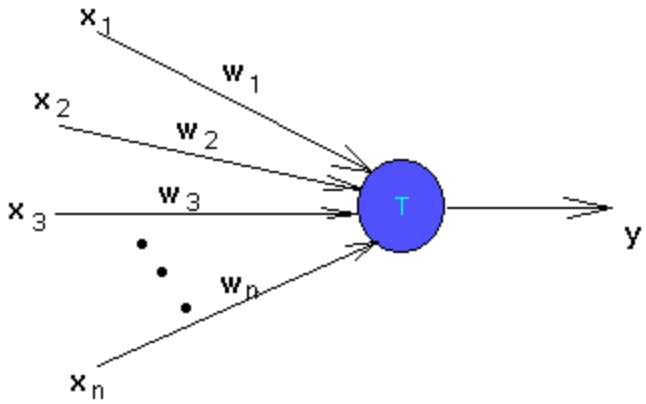
- Computing “AND”:
  - There are n inputs, each either a 0 or 1
  - To compute the logical ”AND" of these n inputs, the output should be 1 if and only if all the inputs are 1
  - This can be easily achieved by setting the threshold of the perceptron to n and maintaining the weights of all edges at 1
    - The net input can be n only if all the inputs are active
- Computing “OR”:
  - It is also simple to see that if the threshold is set to 1 then, the output will be 1 if at least one input is active
  - The perceptron in this case acts as the logical "or"

Training the Perceptron



- A series of $(x_1, x_2, ..., x_n)$ inputs are presented to the perceptron.
- For each such input, there is a desired output, either 0 or 1
- The actual output is compared with the desired output. If the perceptron gives a wrong (undesirable) output, then one of two things could have happened:
  - The desired output is 0, but the net input is above threshold. So the actual output becomes 1. In such a case one should decrease the weights
  - The desired output is 1, but the net input is below threshold. In this case one should increase the weights

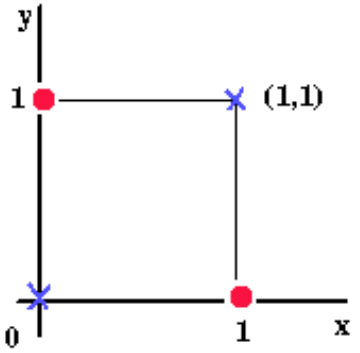
Training the Perceptron (II)



- The perceptron learning algorithm says that the decrease in weight of an edge should be directly proportional to the input through that edge
  - $new\_w_i = w_i - cx_i$, where c is a constant in the simpler cases
- The idea is that if the input through some edge was very high, then that edge must have contributed to most of the error
  - So we reduce the weight of that edge more (i.e. proportional to the input along that edge)
- The same rationale applies if there should be an increase the weight of an edge ($new\_w_i = w_i + cx_i$)

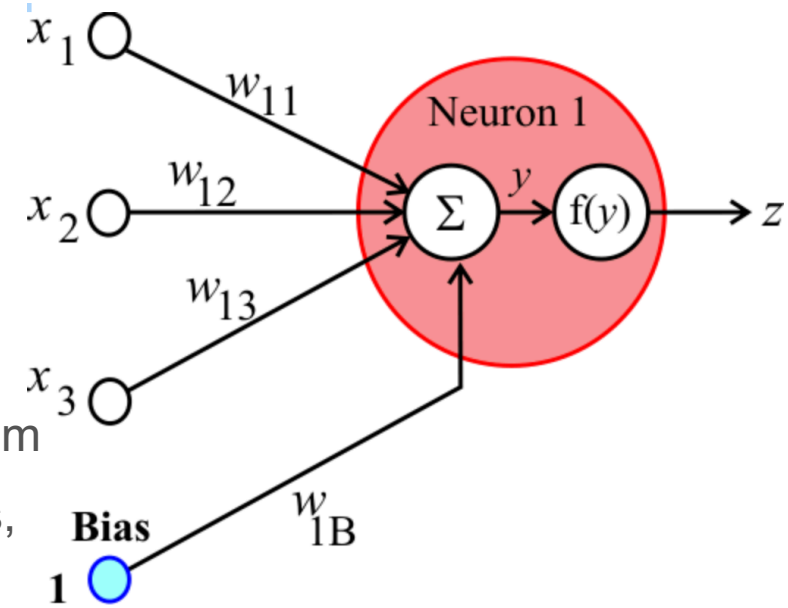
Computing "XOR"

- One cannot find any combination of weights or T that allows the perceptron to implement a function that outputs 1 if two inputs are different, and 0 otherwise!
- The perceptron cannot solve any problem that is not linearly separable:
  - Equation $w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = \mathrm{T}$ defines a hyperplane;
  - The perceptron can only implement functions where all points on one side of the hyperplane belong to the same class;
- On the 2-D XOR example, the hyperplane is a straight line, and no single straight line can divide the XOR space (Minsk and Papert):



Improving the perceptron

- A perceptron is able to learn, and the training algorithm converges as long as the perceptron can solve the problem…
- A "neuron" is a perceptron that includes a bias and replaces the threshold by a non-linear activation function:
  - The bias allows the neuron to increase its learning flexibility by shifting the value of y as required for the specific conditions of the problem
  - The nonlinear function must be real, continuous, limited and have a positive derivative (usually a sigmoid)

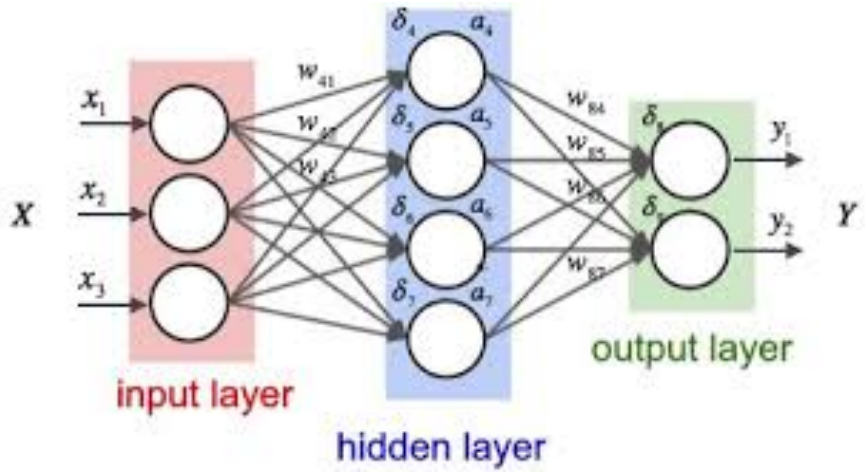


The Multilayer Perceptron

Feed Forward NN

Multilayer Perceptron: “Feed Forward” Neural Networks

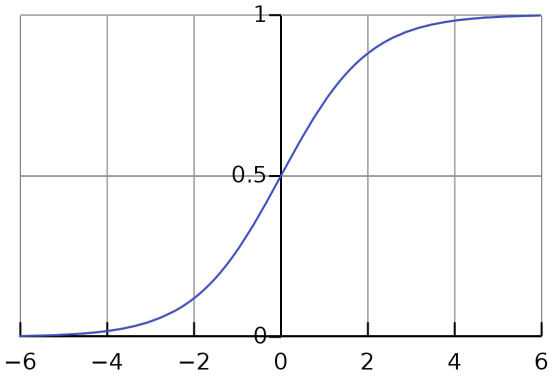
- A single perceptron cannot deal with non-linear problems
- However, by organizing a given number of perceptrons into at least 3 layers, one obtains a trainable Neural Network that can behave as a universal approximator
  - E.g.: 3 layer, 3 input, 2 outputs NN:

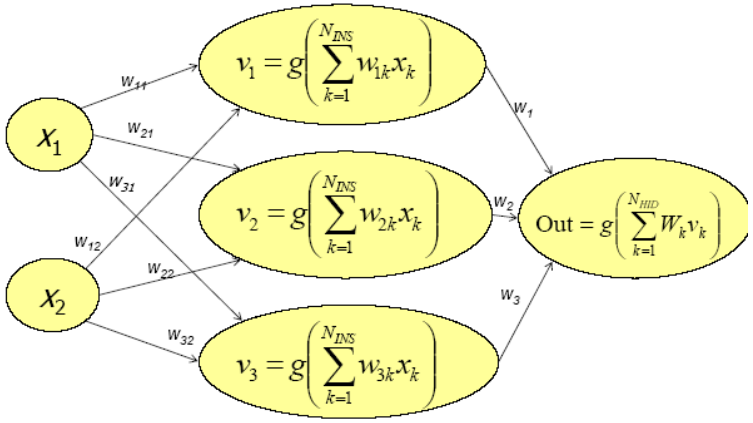


Multilayer Perceptron

- Assume the following activation function and 3 layer NN:

$$g(h) = \frac{1}{1+\exp(-h)}$$





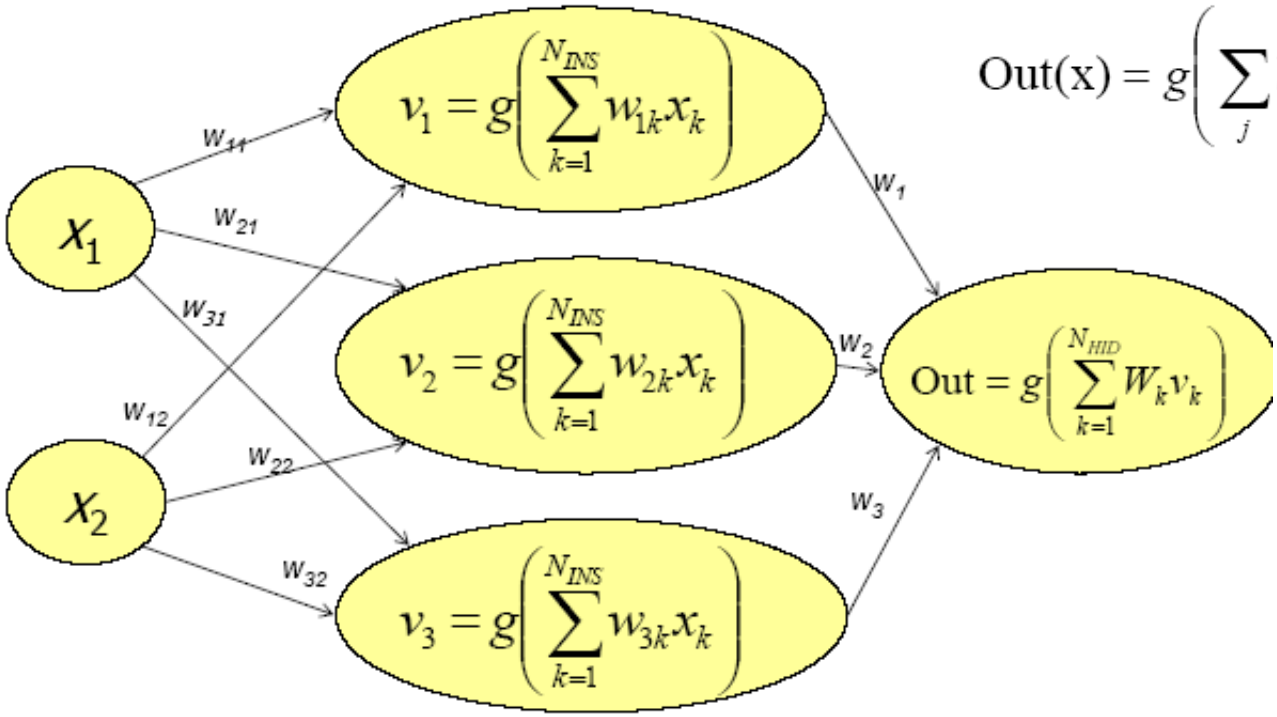
- The output of the NN is given by:

$$\text{Out(x)} = g\left(\sum_j W_j g\left(\sum_k w_{jk}x_{jk}\right)\right)$$

This is a nonlinear function
Of a linear combination
Of non linear functions
Of linear combinations of inputs

Multilayer Perceptron (II)

- Input Layer = 2; Hidden Layer = 3; Output Layer = 1



$$\text{Out(x)} = g\left(\sum_j W_j g\left(\sum_k w_{jk}x_k\right)\right)$$

Training the MLP: The Backpropagation Algorithm

$$\text{Out(x)} = g\left(\sum_j W_j g\left(\sum_k w_{jk} x_k\right)\right)$$

- Goal: Given a set of input/output training data (x_i, y_i), find a set of weights $\{W_j\}, \{w_{jk}\}$ to minimize $\sum_i \left(y_i - \text{Out}(x_i)\right)^2$ using the gradient descent method
- The sigmoid perceptron update rule is:

$$w_j \leftarrow w_j + \eta \sum_{i=1}^{R} \delta_i g_i (1 - g_i) x_{ij}$$

where $g_i = g\left(\sum_{j=1}^{m} w_j x_{ij}\right)$

$$\delta_i = y_i - g_i$$

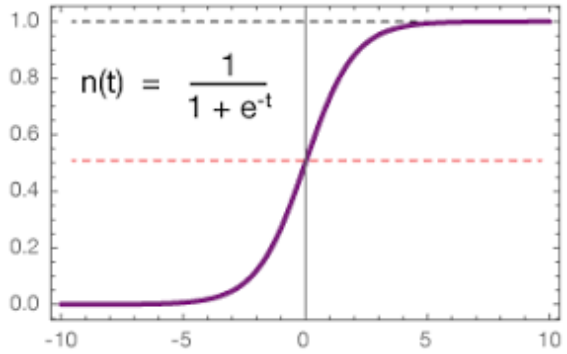
η is the Learning rate. It controls how fast the model converge. Typical values: [0.01,0.1]

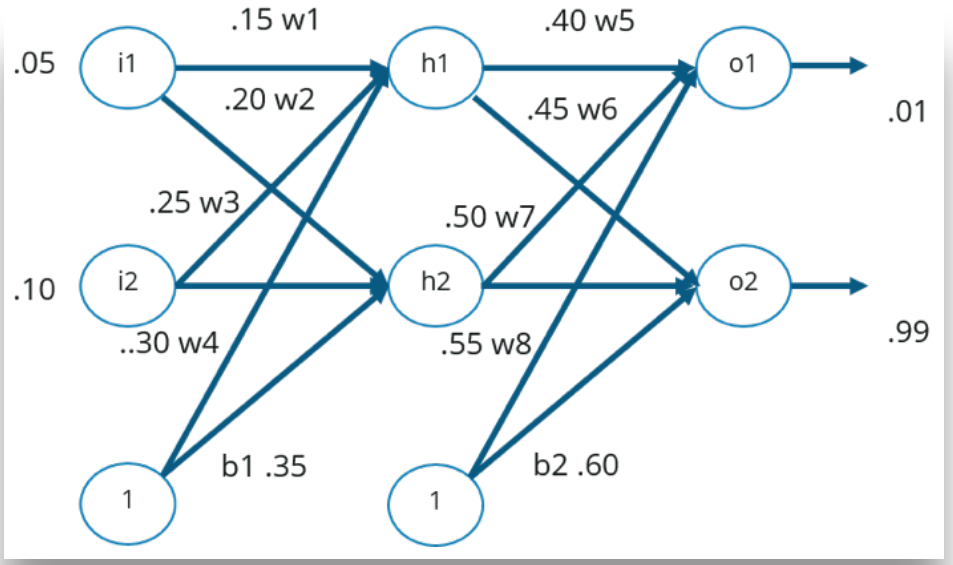
Training the MLP: The Backpropagation Algorithm (II)

1. Initialization
2. For each training example:
   1. Take input and process it (forward propagation)
   2. Result is compared with the desired output ⇒ the error is obtained
   3. Try to minimize the error, by changing some elements in the network (backward propagation)
      - Update each weight, so that the actual output becomes closer to the target output (minimizing the error). A learning rate is used (η).
3. If error $\geqq \varepsilon$, or if error is still decreasing at a reasonable rate, go to 2.

Step-by-Step Training Example

- 2 inputs: $i_1 = 0.05$; $i_2 = 0.1$
- 2 outputs: $o_1 = 0.01$; $o_2 = 0.99$
- Initial weights and bias: random
- Activation function: logistic

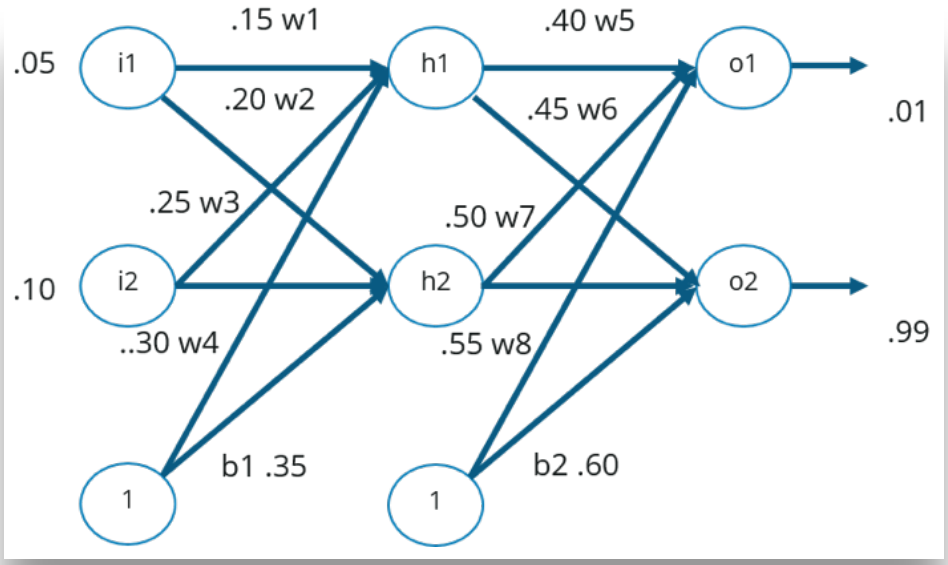




Example from:
https://www.edureka.co/blog/backpropagation/

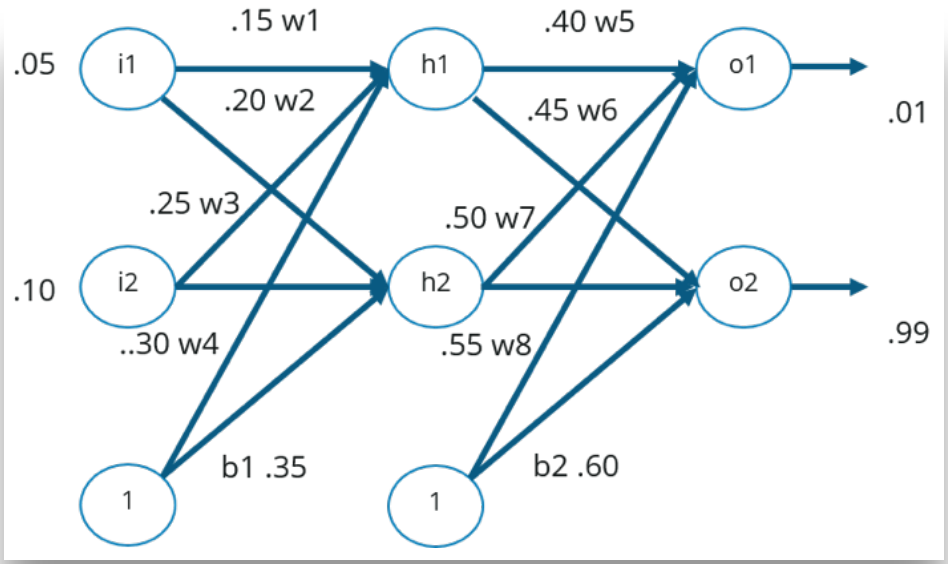
Step-by-Step Training Example (II)

- $net_{h1} = w_1 \times i_1 + w_2 \times i_2 + b_1 \times 1 = 0.15 \times 0.05 + 0.2 \times 0.1 + 0.35 \times 1 = 0.3775$
- After net_{h1} activation: $out_{h1} = 0.5933$
- Calculate: net_{h1}, out_{h2}, net_{o1}, out_{o1}, net_{o2}, out_{o2}



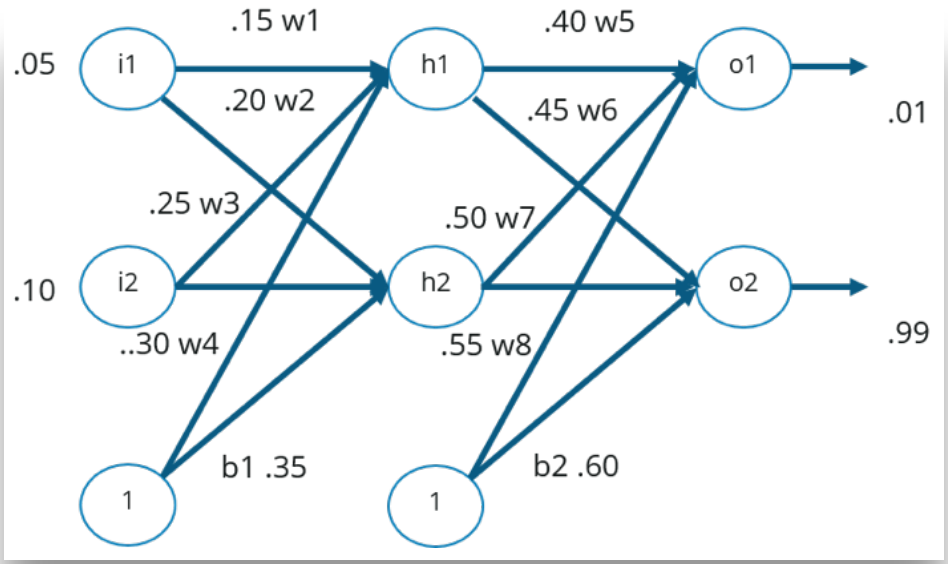
Step-by-Step Training Example (Error)

- $E_{o1} = \frac{1}{2} (target_{o1} - out_{o1})^2 = 0.2748$; Eo2 = 0.0236
- $E_{total} = E_{o1} + E_{o2} = 0.2984$



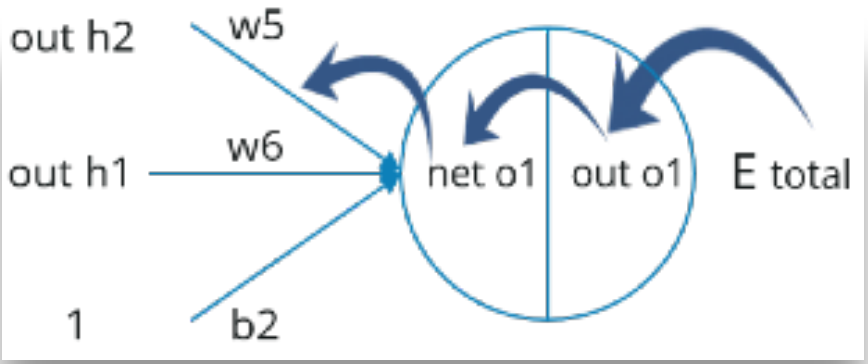
Step-by-Step Training Example (Backpropagation)

- How much a change in $w5$ affects the total error?
  - partial derivative of E_{total} with respect to w_5 = the gradient with respect to $w_5 = \frac{\partial E_{total}}{\partial w5}$



Step-by-Step Training Example (Backpropagation II)

- Chain rule: $\frac{\partial E_{total}}{\partial w5} = \frac{\partial E_{total}}{\partial Out_{01}} \times \frac{\partial Out_{01}}{\partial net_{01}} \times \frac{\partial net_{01}}{\partial w_5}$



Step-by-Step Training Example (Backpropagation III)

- Chain rule: $\frac{\partial E_{total}}{\partial w5} = \frac{\partial E_{total}}{\partial Out_{o1}} \times \frac{\partial Out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial w_5}$

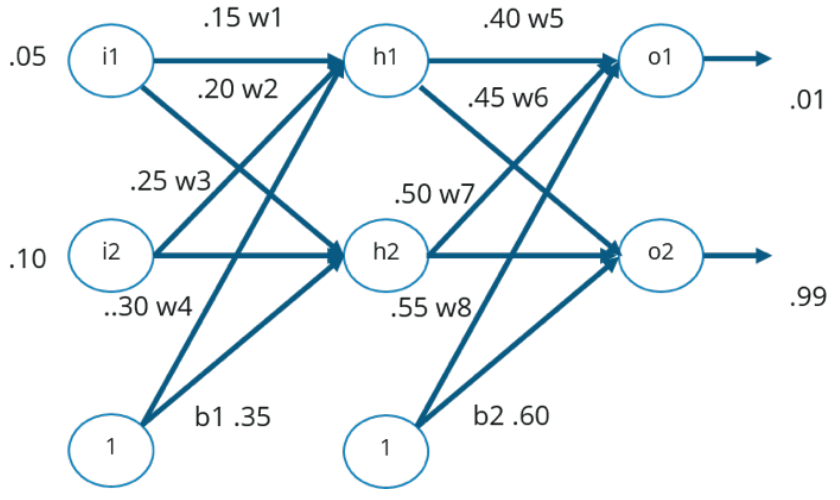
- $E_{total} = ½ (target_{o1} - Out_{o1})^2 + ½ (target_{o2} - Out_{o2})^2$, therefore:
  $\frac{\partial E_{total}}{\partial Out_{o1}} = -(target_{o1} - Out_{o1}) = -(0.01 - 0.7514) = 0.7414$
- $out_{o1} = 1/(1+e^{-neto1})$, therefore: $\frac{\partial Out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.1868$
- $net_{o1} = w_5 \times out_{h1} + w_6 \times out_{h2} + b_2 \times 1$, therefore: $\frac{\partial net_{o1}}{\partial w_5} = out_{h1} = 0.5933$
- $\frac{\partial E_{Total}}{\partial w_5} = 0.7414 \times 0.1868 \times 0.5933 = 0.0821$

Step-by-Step Training Example (Backpropagation III)

- Being $\eta=0.1$ the Learning rate, we update $w5$ accordingly:

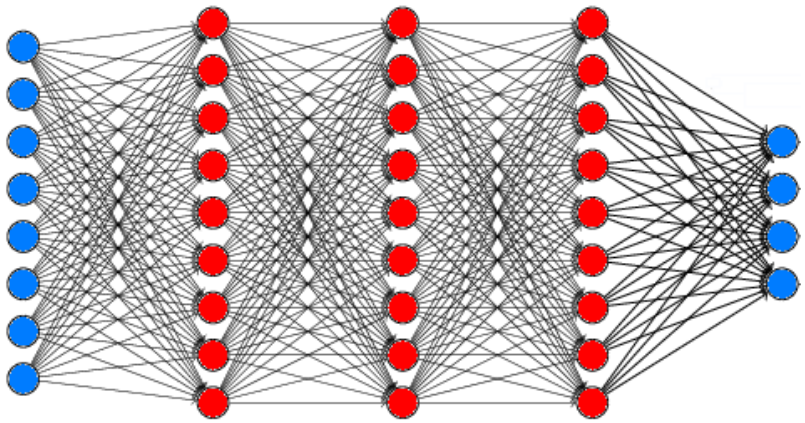
$$w_5 = w_5 - \eta \frac{\partial E_{total}}{\partial w5} = 0.4 - 0.1 \times 0.0821 = 0.3918$$

- The previous steps must be repeated with all weights and applied to all training data pairs



Hyperparameters: Inputs and Outputs

- Number of Inputs and Outputs: Can't be changed, they are associated with the problem one is trying to solve
  - The higher the dimensionality of the input, the (exponentially) higher is the needed size of training data
  - The dimensionality of the output increases (exponentially) the BP computing time due to the chain rule



Hyperparameters: Number of Hidden Layers

- Theoretically, a single hidden layer is enough to solve any problem
  - Universal approximator [Cybenko89, Hornik91]
- However making a single hidden layer network learn a complex problem might be tricky and highly dependent on the quality of the training set
- Hence, using additional hidden layers is advised for complex problems, such as those involving time-series and computer vision
- ☹ More layers ⇒ better learning algorithms; slower learning process

Hyperparameters: Number of Hidden Layers (II)

Number of Hidden Layers	MLP capabilities
0	Only capable of representing linear separable functions or decisions
1	Can approximate any function that contains a continuous mapping from one finite space to another
2	Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy
>2	Additional layers can learn complex representations (sort of automatic feature engineering for layers)

Hyperparameters: Number of Neurons in Hidden Layers

- Too few neurons ⇒ Underfitting (not enough neurons to capture the problem intricacies)
- Too much neurons ⇒ Overfitting (the information in the training set is not enough to train all the neurons in the hidden layers); Exponential increase in training time
- The ideal size will depend on the problem and the available dataset. Trial and error is needed. Some (slightly contradictory) rules of thumb:
  - Size of Input layer > Size of Hidden layer > Size of Output layer
  - Size of Hidden layer = 2/3 Size of Input layer + Size of Output layer
  - Size of Hidden layer < 2x Size of Input layer

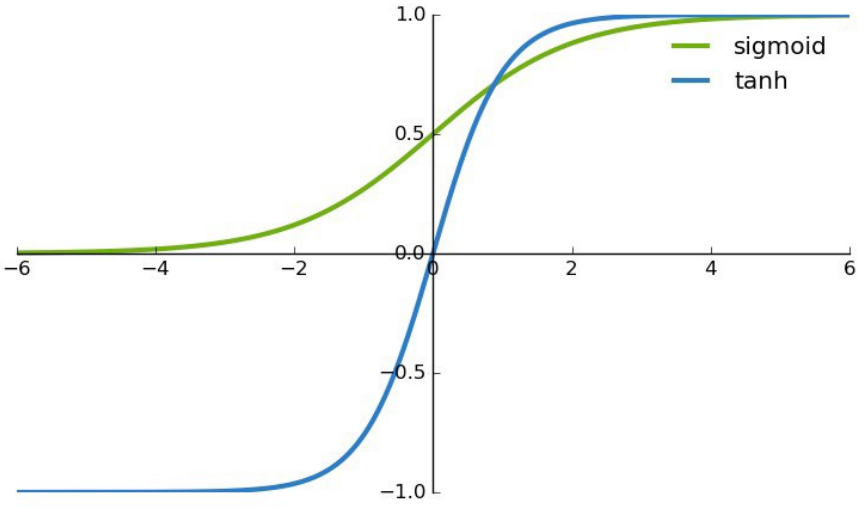
Hyperparameters: Activation function

- We need the activation function to introduce nonlinear real-world properties to artificial neural networks
- In MLP, the most commonly used are:
  - Logistic function (aka sigmoid)

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

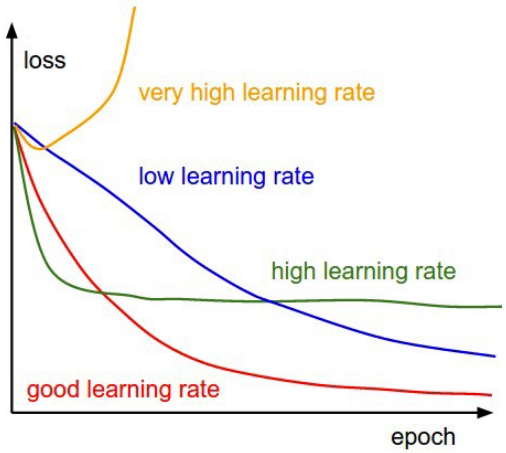
  - Tanh

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



Hyperparameters: Learning Rate η

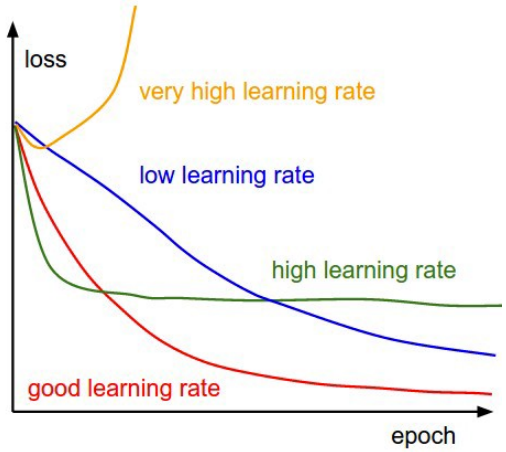
- η may be the most important NN hyperparameter
  - Large learning rates result in unstable training and non-optimal result
  - Tiny rates lengthen the training process and might result in a failure to train



- Typical values: $\eta \in [0.01, 0.1]$
- Several methods exist to improve the convergence process:
  - Momentum can accelerate training, and learning rate schedules can help to converge the optimization process
  - Adaptive learning rates can accelerate training and alleviate some of the pressure of choosing a learning rate and learning rate schedule

Epochs

- Epoch: One run over the whole training set
- Number of Epochs: Number of times the dataset is trained
  - Fixed number, or
  - Until reaching an acceptable error



MLP Made Easy...

- There are many Python libraries that allow for an easy implementation of MLP (as long as you know what you are doing)
- A good place to start (that should be familiar by now) is scikit-learn:
  - https://scikit-learn.org/stable/modules/neural_networks_supervised.html

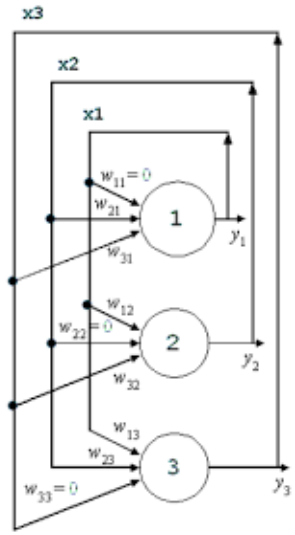
```
>>> from sklearn.neural_network import MLPClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                     hidden_layer_sizes=(5, 2), random_state=1)
...
>>> clf.fit(X, y)
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(5, 2), random_state=1,
              solver='lbfgs')
```

Recurrent NN

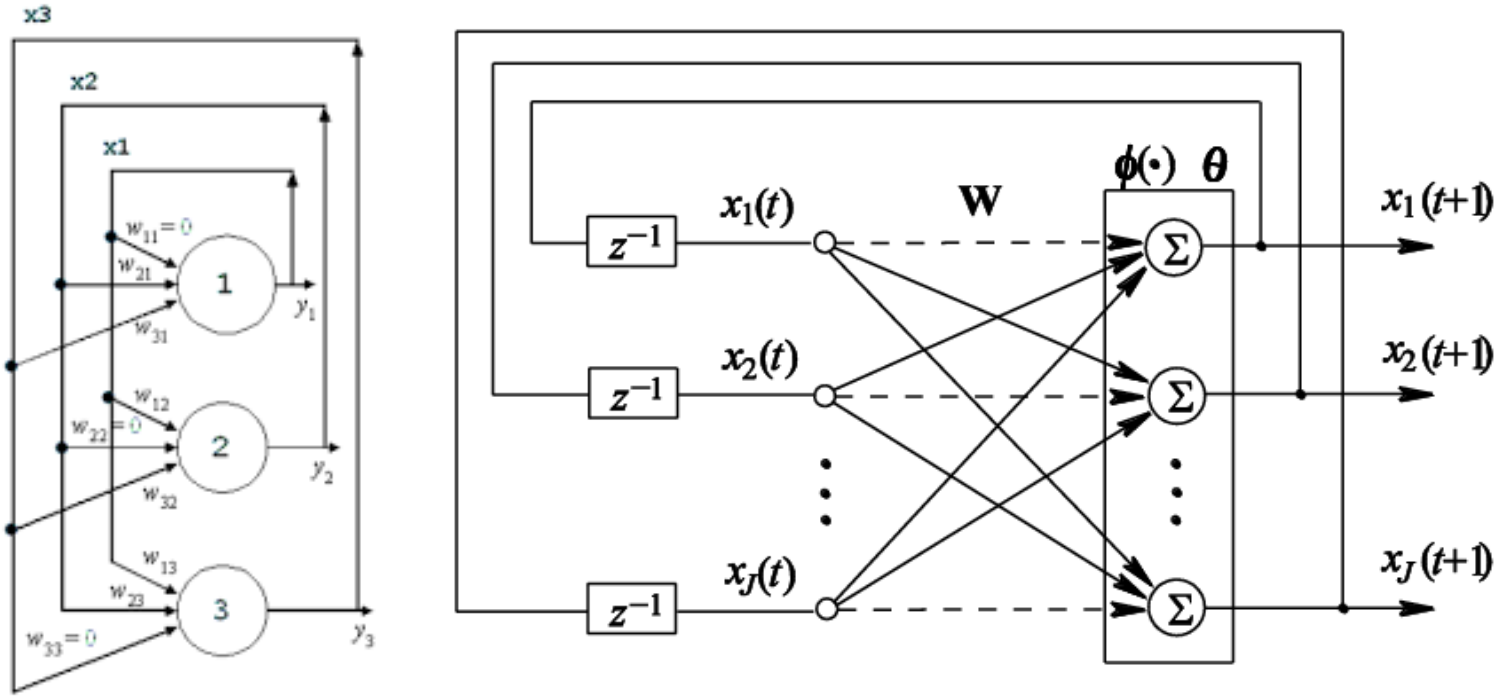
Hopfield Nets

Recurrent NN

- Recurrent NN: NN with feedback links (remember Flip-Flops?)
  - Like FF, recurrent NN exhibit a “memory behaviour”
- Hopfield Networks [1982]
  - Neurons are binary threshold units
    - Behave as associative memories: the net recovers memories on the basis of similarity to an input state (pattern)
    - The network converges to a "remembered" pattern if it is given only part of the pattern
      - ☹ sometimes converge to a wrong pattern – local minimum



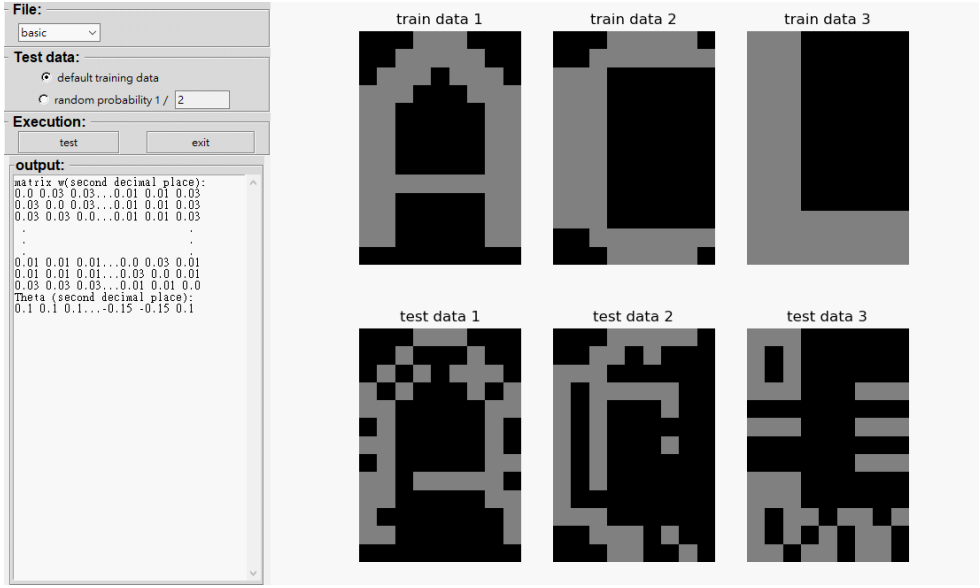
Hopfield Networks

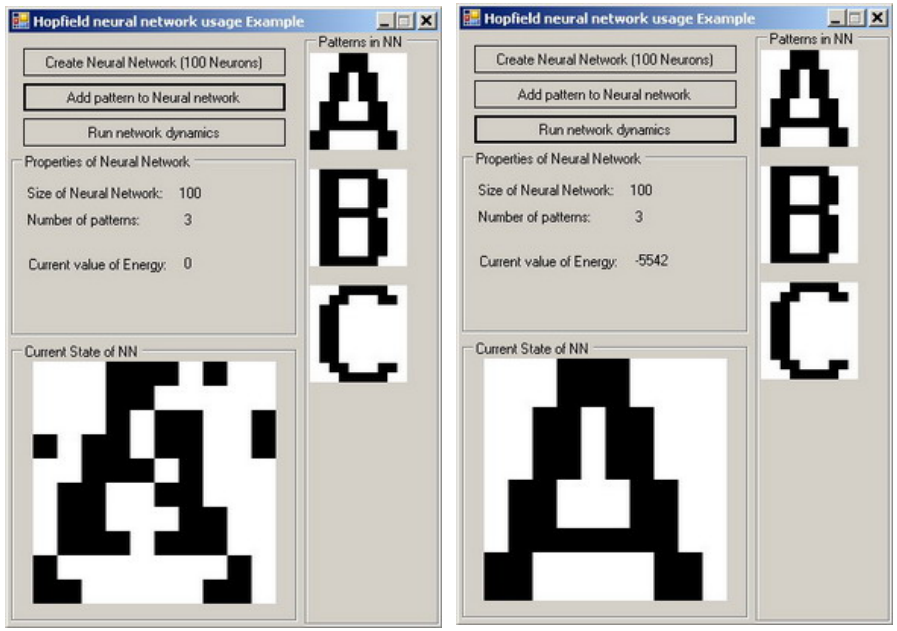


- Usually $w_{ii} = 0$, i.e., there is no self-realimentation

Hopfield Networks: Example

- https://github.com/daniel4lee/Hopfield
- https://www.codeproject.com/Articles/15949/Hopfield-model-of-neural-network-for-pattern-recog





Hopfield Networks: Inference and Learning

- Inference – updating one unit in the Hopfield network is performed using the following rule:

$$s_i \leftarrow \begin{cases} +1 & \text{if } \sum_j w_{ij} s_j \geq \theta_i, \\ -1 & \text{otherwise.} \end{cases}$$

  where s is the state (pattern) and θ the threshold

- Hebbian learning rule:

$$w_{ij} = \frac{1}{n} \sum_{\mu=1}^{n} \epsilon_i^\mu \epsilon_j^\mu$$

  where ε_i^μ represents node i from pattern μ.

  - Capacity: 138 patterns can be recalled from storage (i.e. learnt) for every 1000 nodes [Hertz 1991]

Hopfield Networks – Applications and Limitations

- Applications:
  - Recalling or reconstructing corrupted patterns (enhancement of X-Ray images, medical image restoration, etc.)
  - Image detection (hand writing recognition software)
- Limitations:
  - Practical applications are limited since the number of training patterns can be at most aprox 14% of the number of nodes in the network
  - If the network is overloaded, i.e., trained with more than the maximum accepted number of attractors, then it won't converge to clearly defined attractors
  - **Require stationary inputs** (feed it 1 input, and let the net converge)

Recurrent NN

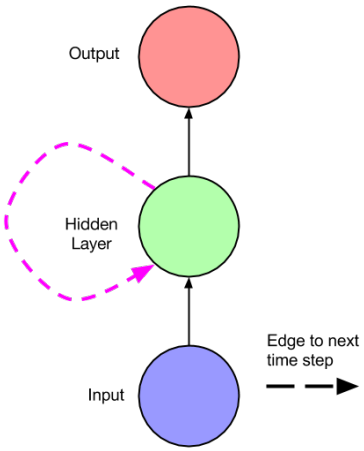
with non-stationary inputs: Elman RNN

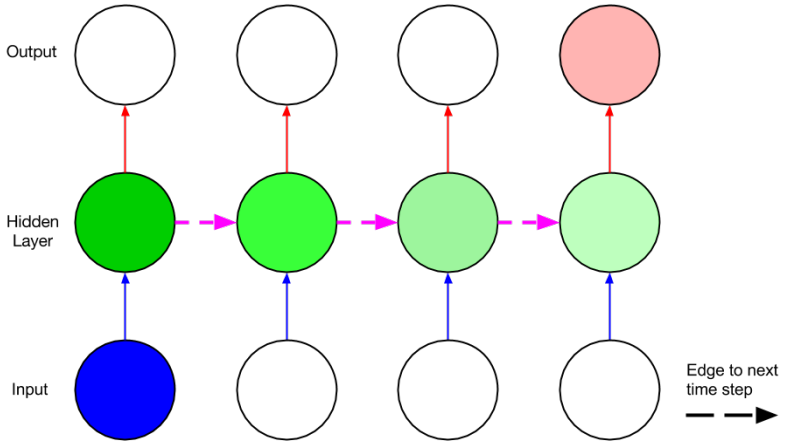
Recurrent Networks with Non-Stationary Inputs

- In some problems it is important to remember the past in order to better predict what is coming next. Examples:
  - Predicting the next word when typing a sentence;
  - Anomaly detection in time-series;
  - Intrusion detection in computer networks;
  - Human action recognition.
- Recurrent networks that accept non-stationary inputs (RNN) help dealing with these problems: the output from previous step is fed as input to the current step
  - A hidden state (context) is used to remember some information about the input sequence

RNN

- A simple recurrent net with one input unit, one output unit, and one recurrent hidden unit
  - The net can be visualized by “unfolding” it along the time axis

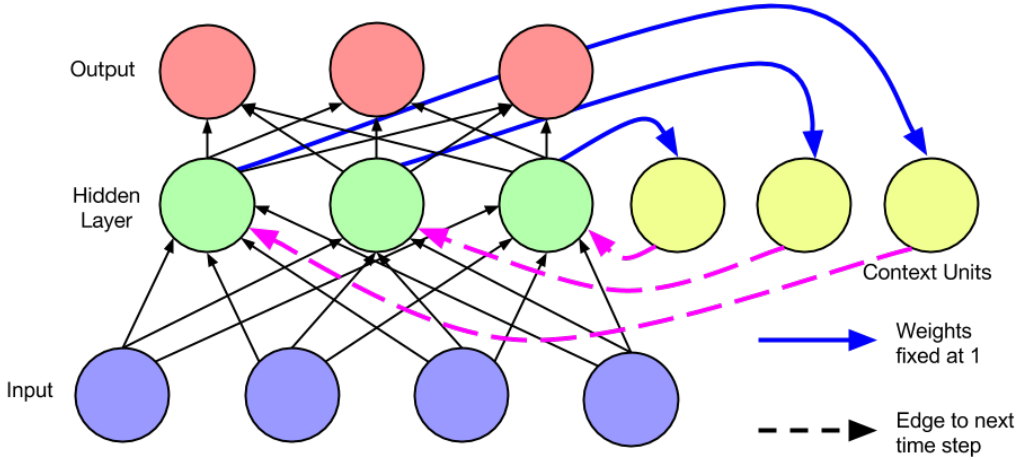




RNN – Elman and Jordan Networks

- Elman Networks

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$



x_t: input vector

h_t: hidden layer vector

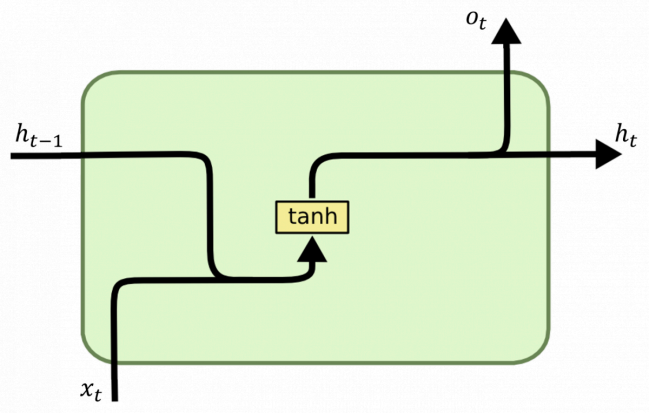
y_t: output vector

$W, U \; and \; b$: parameter Matrices (trainable) and bias vector

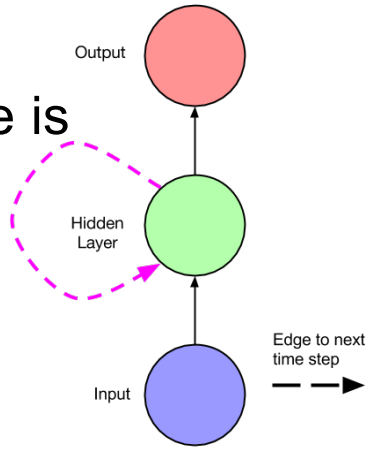
$\sigma_h \; and \; \sigma_y$: activation functions

- Jordan Networks
  - Context layer gets input from y instead of h, and has self-feedback

RNN – Training

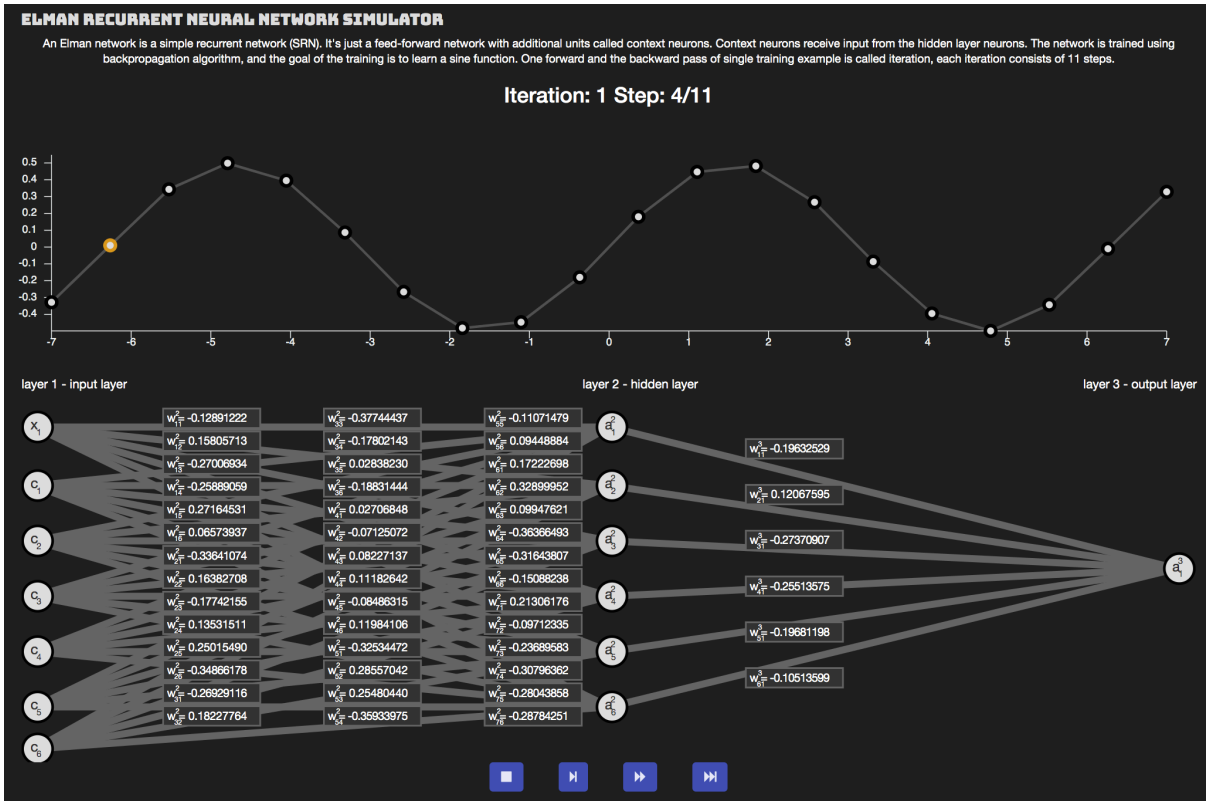


1. Feed the RNN a single time step of the input x
2. Calculate its current state using the current input x and the previous state h_{t-1}: $h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$
   - The current h_t becomes h_{t-1} for the next time step
3. Repeat for as many time steps as needed by the problem
4. Once all the time steps are completed, the final current state is used to calculate the output: $y_t = \sigma_y(W_y h_t + b_y)$
5. Compare the output with the target output and generate the error
6. Back-propagate the error to update the weights



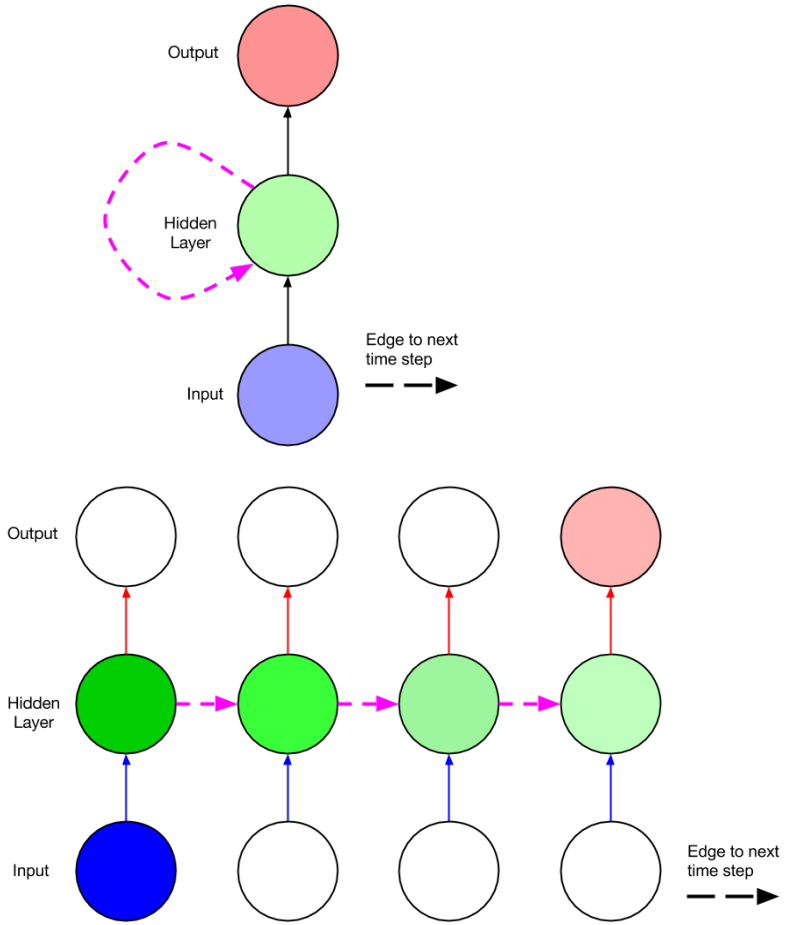
Elman RNN – Training Simulator

- https://www.mladdict.com/elman-recurrent-neural-network-simulator



Elman RNN – Problems

- Training an RNN is a very difficult task
  - It cannot process very long sequences if using tanh or ReLU as an activation function
- Gradient vanishing and exploding problems
  - Having a fixed U_h for many iterations will cause the contribution of the input to the output to either approach 0 or explode very fast
  - Hence, the derivative of the error with respect to the input will either vanish or explode



Recurrent NN

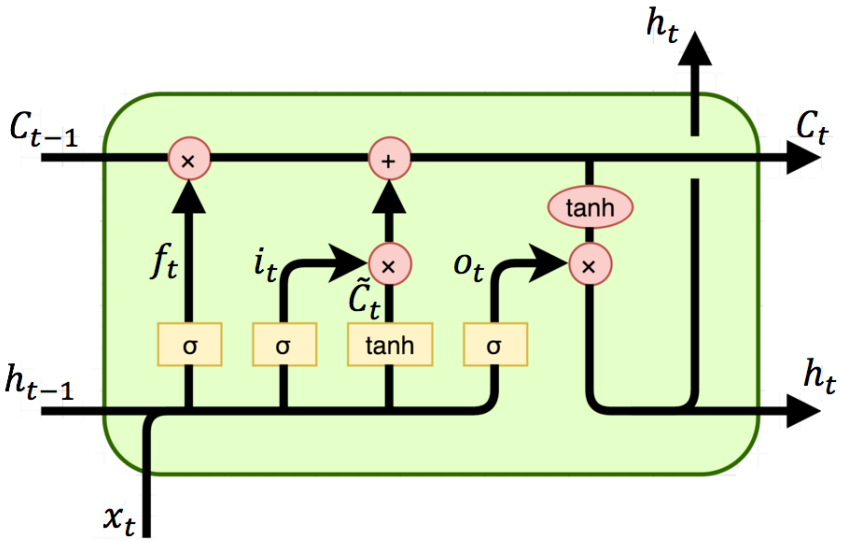
LSTM – Long Short-Term Memory

GRU – Gated Recurrent Units

LSTM – Long-short Term Memory RNN

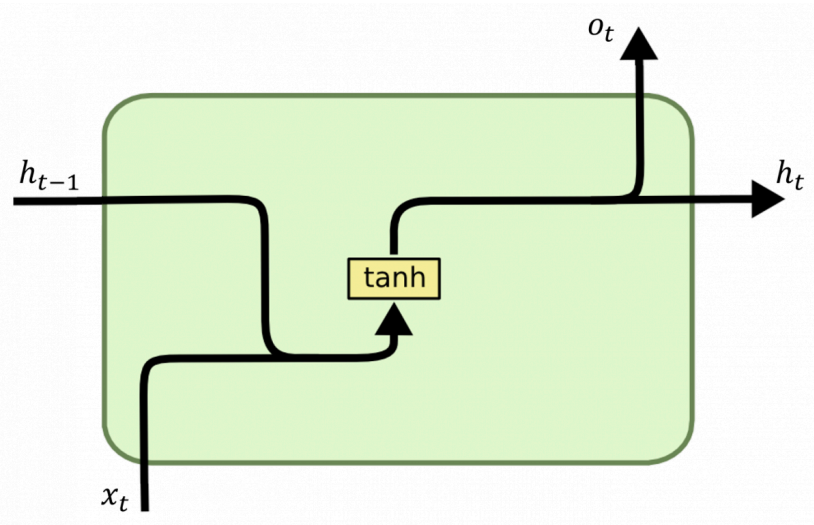
- LSTM [Hochreiter and Schmidhuber, 1997] were developed to deal with the RNN vanishing gradient problem, allowing them to learn long term dependencies by “remembering” longer sequences
- A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate

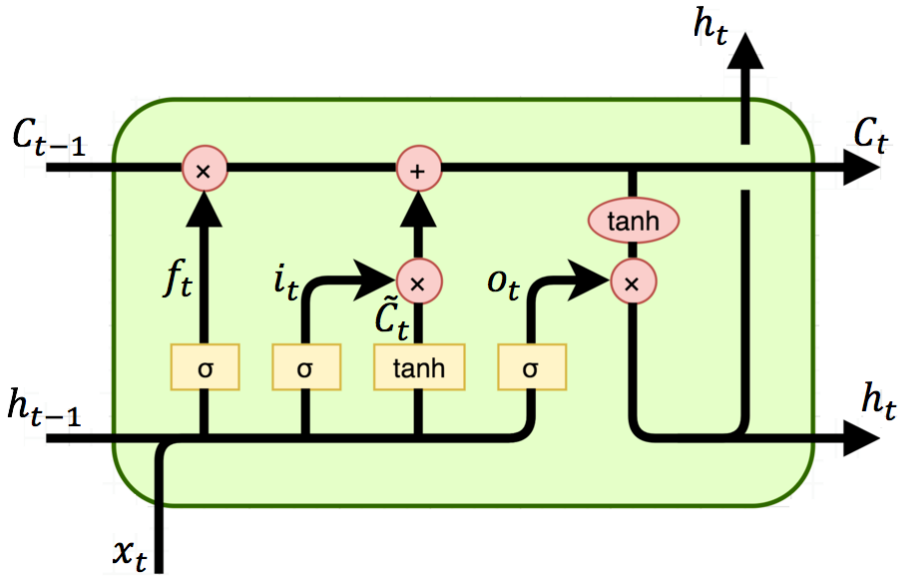
The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell



Simple RNN (Elman) vs. LSTM

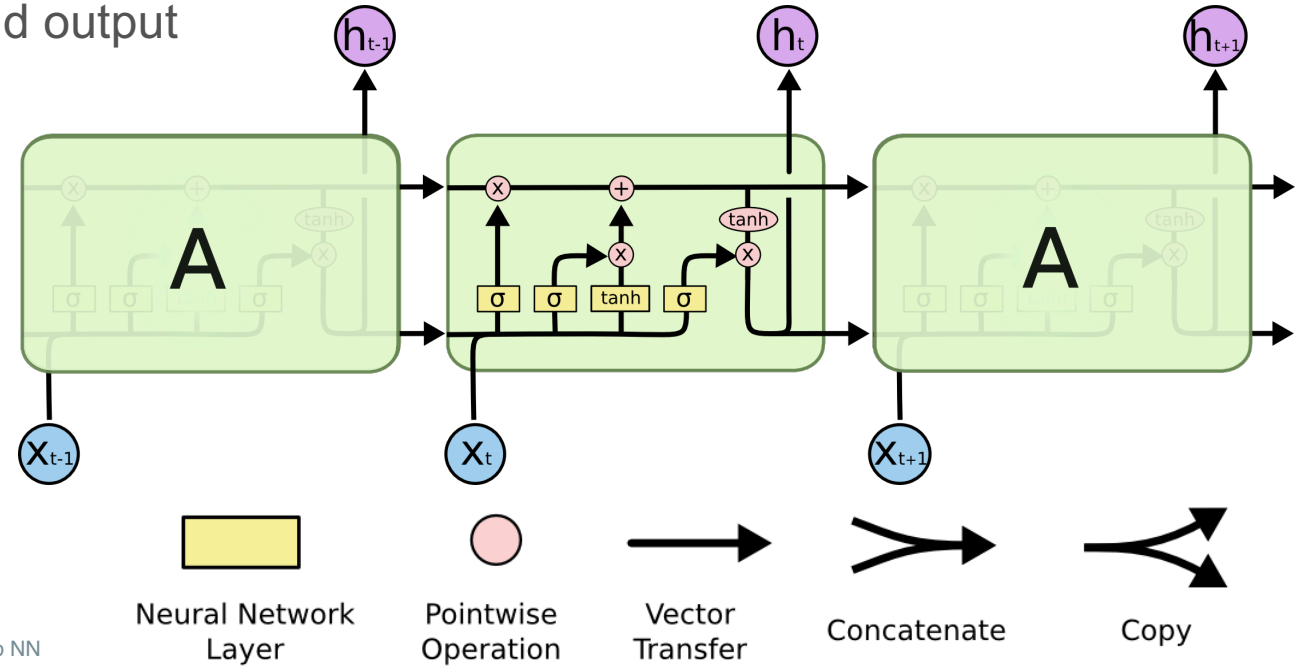
- Comparison between the structure of a simple RNN vs. a LSTM





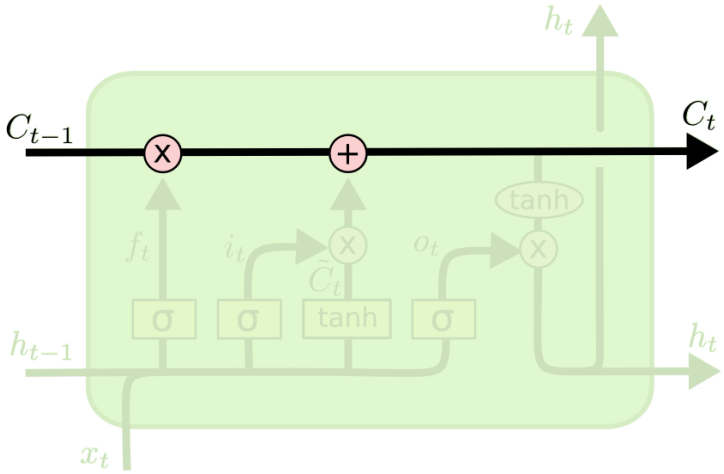
LSTM Chain

- LSTM can be unfolded to look as a chain of repeating cells with four layers interacting within:
  - Each LSTM cell receives inputs from an Input sequence and the previous cell's state and output



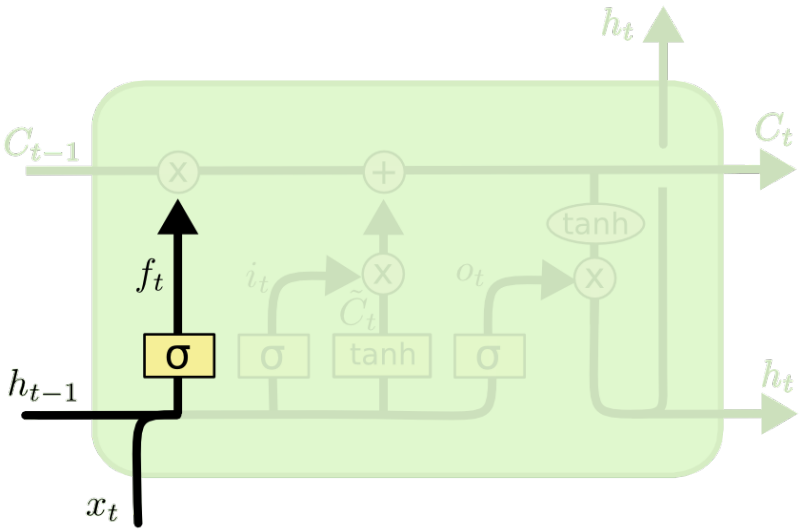
LSTM – Cell State

- Cell State allows easy flow of unchanged information through the subsequent LSTM cell thereby helping preserve context
  - Cell state play an important role for LSTM to learn long-term dependencies
- Cell states are controlled by gates on how much information to allow to subsequent LSTM cells



LSTM – Forget Gate

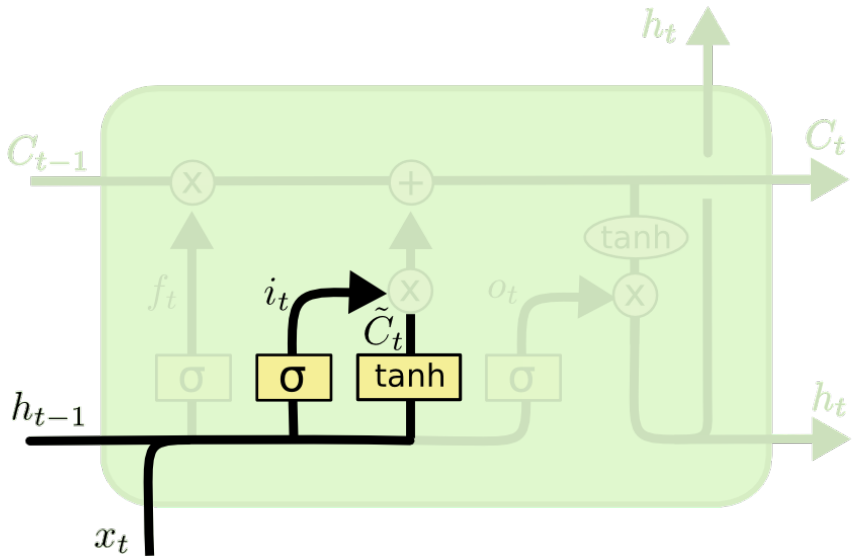
- "What information can be thrown away?"
- The Forget gate alters cell state based on the current input (x_t) and the output (h_{t-1}) from the "previous" LSTM cell



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

LSTM – Input Gate

- "What new information should be stored?"
- The Input gate decides and computes values to be updated in the cell state

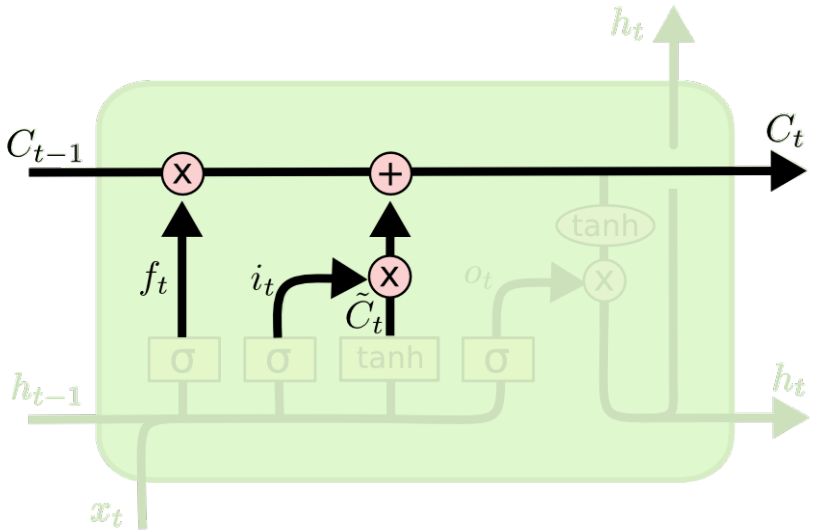


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

LSTM – Update Current State

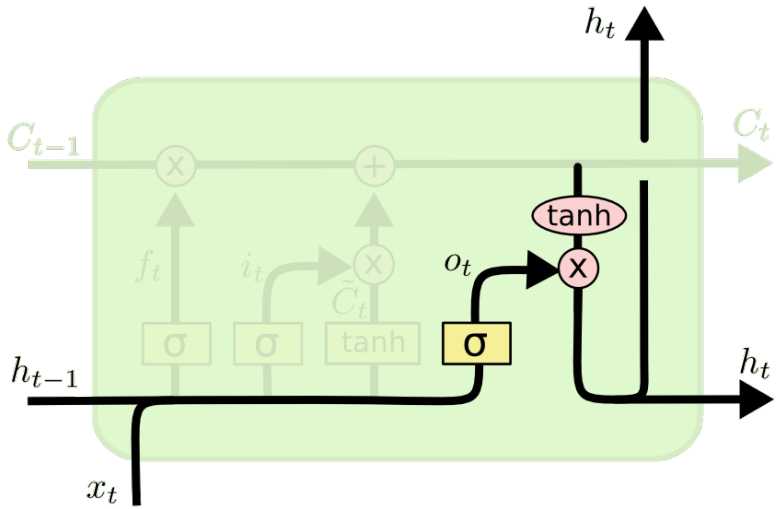
- Multiply the old state by f_t, forgetting the things decided to forget earlier
- Then add the new candidate values, scaled by how much it is decided to update each state value



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

LSTM – Output Gate

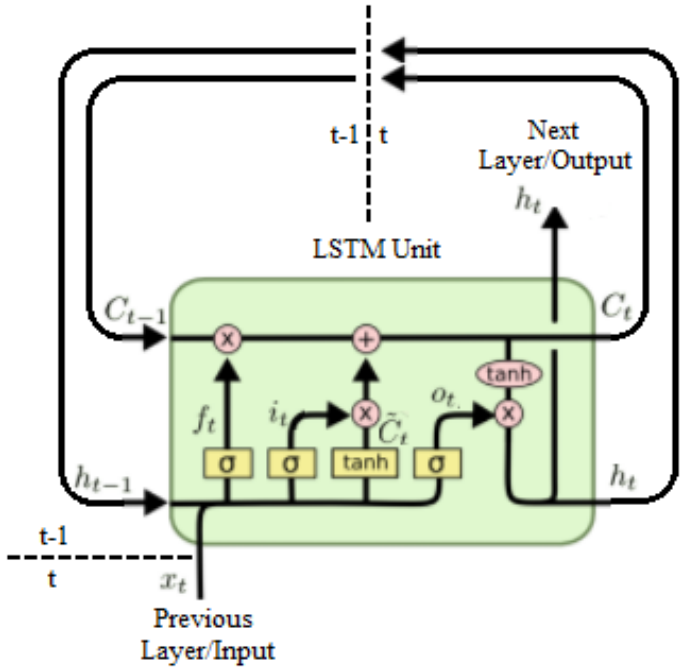
- "What will be output?"
- The Output gate decides and computes values to be updated in the hidden state
  - Filtered version of the current Cell state



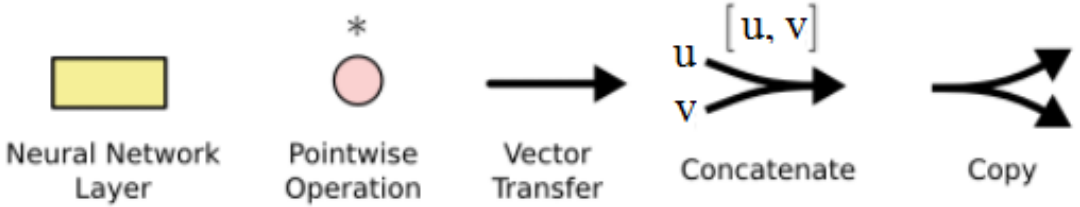
$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM – Inference Overview



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



LSTM – Training

1. Feed the LSTM a single time step of the input x
2. Calculate Cell state (C_t) and h_t using the inference steps
   - The current C_t and h_t will become C_{t-1} and h_{t-1} for the next time step
3. Repeat for as many time steps as needed by the problem
4. Once all the time steps are completed, the final h_t is used to calculate the output
5. Compare the output with the target output and generate the error
6. Back-propagate the error to update the weights

LSTM – Hyperparameters

- Layers : Number of units in hidden layers
- Sequence Length or Timestamps: Number of previous sequence / timestamps to look
- Batch size: Frequency of weight update
- Activation functions: Introduces non-linearity in the model
  - Sigmoid, Tanh, ReLU, SoftMax (SoftArgMax)
- Loss function: Computes the error (MSE, RMSE, Cross Entropy)
- Optimizer: Minimizes loss function

LSTM – Regularization

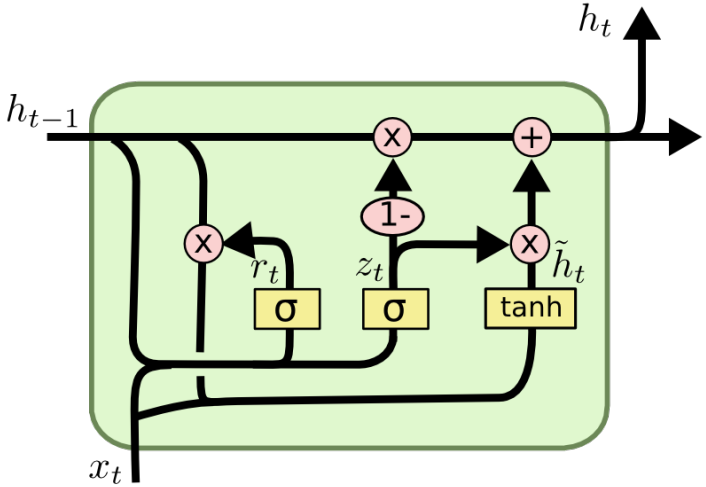
- Techniques that modifies the learning algorithm to generalize better (improve model performance on the unseen data)
- Regularization techniques:
  - Weight (L1 & L2) Regularization: Slows down learning and overfitting of the network by penalising the weights of the nodes
  - Dropout: Probabilistic removal of some input and recurrent connections in order to prevent some activations and weight updates during the learning process (slowing the learning process)
  - Early stopping (Callbacks): Cross-validation strategy which stops training when performance on the validation set is not improving after a certain iteration

LSTM Applications and Achievements

- LSTM achieved record results in natural language text compression and unsegmented connected handwriting recognition
- LSTM networks were a major component of a network that achieved a record 17.7% phoneme error rate on the classic TIMIT natural speech dataset (2013)
- Google used LSTM for speech recognition on Android, for the smart assistant Allo and for Google Translate
- Apple uses LSTM for the "Quicktype" function on the iPhone and for Siri
- Amazon uses LSTM for Amazon Alexa
- In 2017 Facebook performed some 4.5 billion automatic translations every day using LSTM
  - **Note:** all above are Deep LSTM based architectures (not simple LSTM)

GRU – Gated Recurring Unit

- GRU – Gated Recurring Unit [Cho et. al 2014]
  - Simplified LSTM (fewer parameters, no output gate)
  - Not as powerful as LSTM but able to outperform them on certain smaller datasets



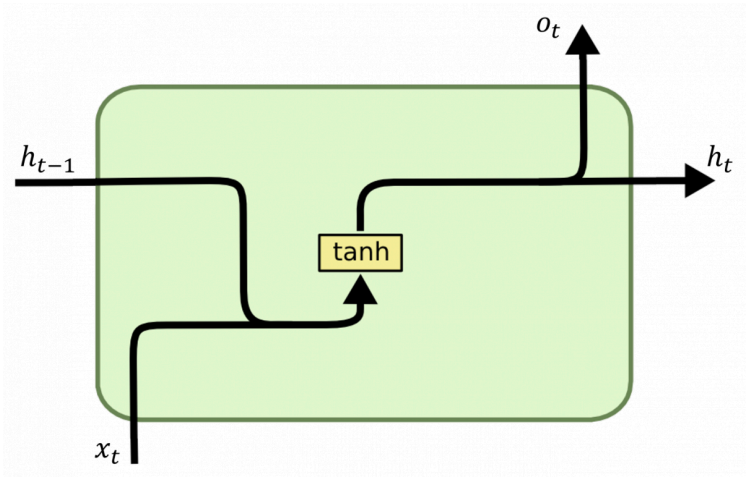
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

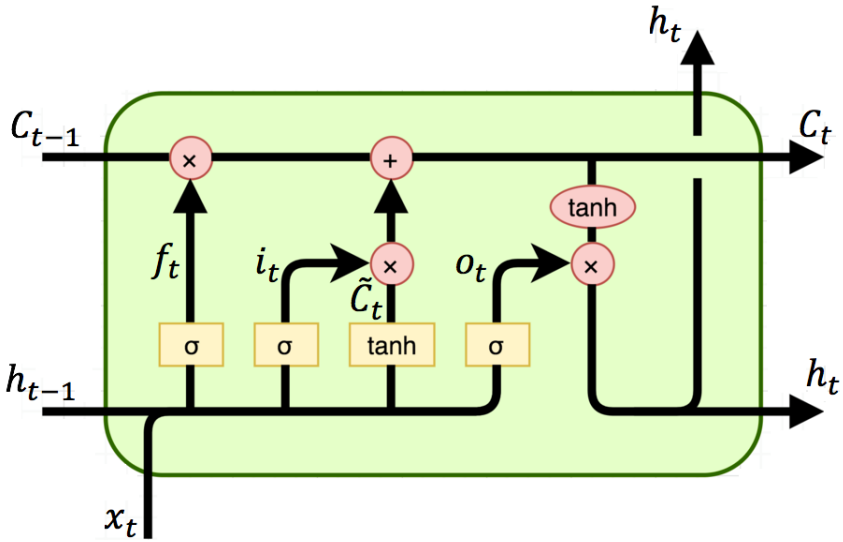
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

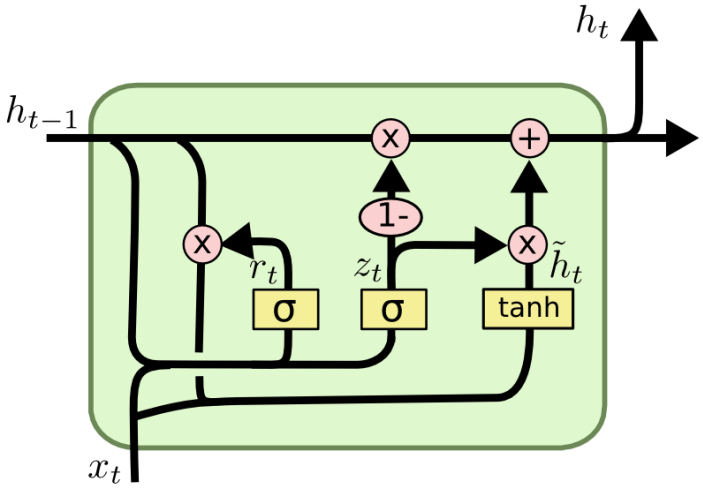
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

RNN vs. LSTM vs. GRU







Other RNN Architectures

- There are many other RNN architectures:
  - BAMs (Bidirectional Associative Memories)
  - IndRNN (Independently RNN)
  - Recursive NN
  - Bi-directional RNN (including bi-directional LSTM)
  - Memristive Networks
  - Etc.
- Each has its advantages and applications, but none are as successfull as LSTM as building blocks for deep learning architectures

Recurrent NN Made (not so) Easy

- Keras is a good way to start using RNN, LSTM, GRU, etc.
- https://keras.io/layers/recurrent/
  - keras.layers.LSTM(units, activation='tanh', recurrent_activation='sigmoid', use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None, bias_constraint=None, dropout=0.0, recurrent_dropout=0.0, implementation=2, return_sequences=False, return_state=False, go_backwards=False, stateful=False, unroll=False)
- Other: PyTorch, Tensor Flow (more on these later on)

Recurrent NN Made (not so) Easy (II)

- Using RNN isn't as easy as using MLP
  - Too many hard to control and understand hyper-parameters
  - Best results are obtained when using RNN as building blocks to complex deep learning architectures
- The best results have been obtained in Image Processing and in Speech and Natural Language Processing
  - Time series forecasting are an obvious application area, but obtaining the best results using off-the-shelf LSTM it is not that straight
    - https://stackabuse.com/time-series-prediction-using-lstm-with-pytorch-in-python/
- For a simple tutorial on using LSTM for time-series:
  - https://machinelearningmastery.com/how-to-develop-lstm-models-for-time-series-forecasting/

Conclusions

Where Are We?

- Deep Mind Artificial Intelligence (AlphaStar) wins 10 out of 11 StarCraft II games versus Pro human players

Where Are We?

- ThisPersonDoesNotExist.com

Where Are We?

System Prompt (human-written)

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Model Completion (machine-written, 10 tries)

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

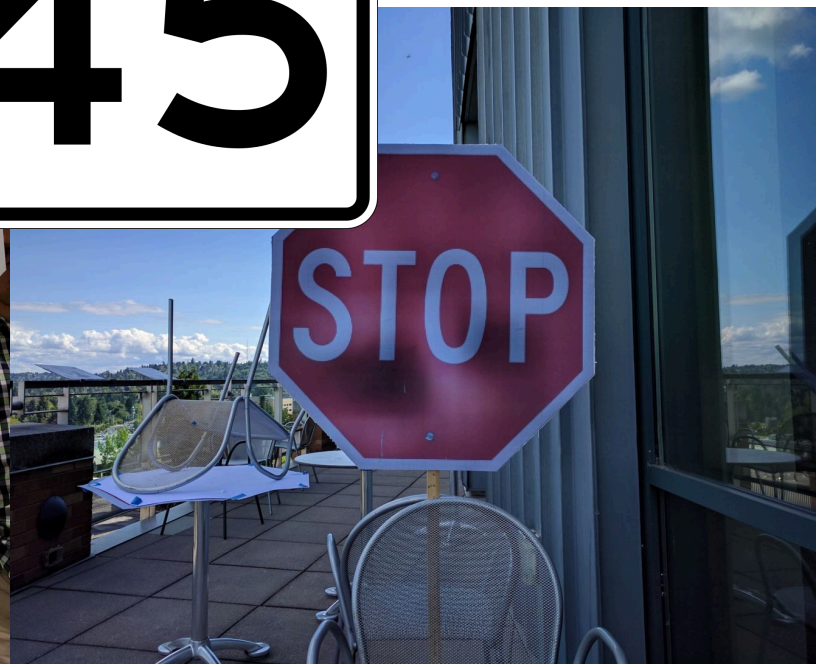
Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez.

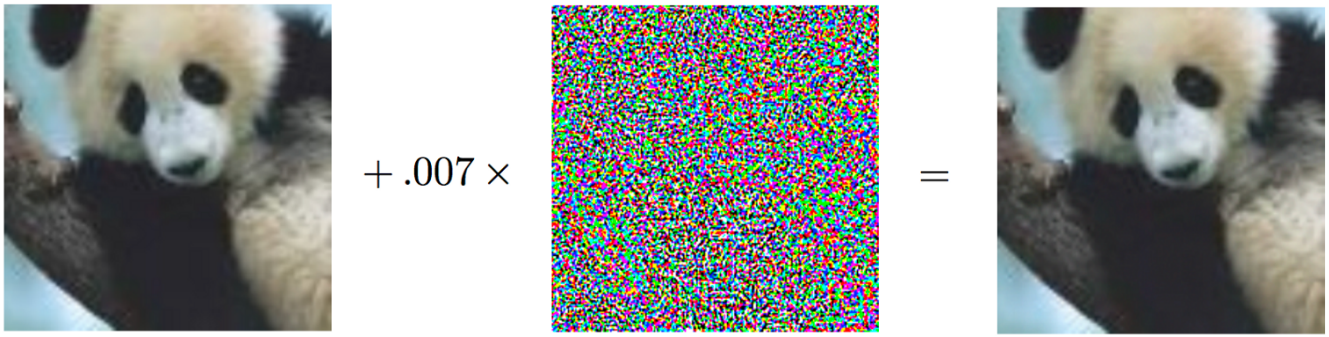
- Note: after 10 tries....

Where Are We?

- Is this a STOP sign?



Where Are We?



- Explaining and Harnessing Adversarial Examples [Goodfellow et al, 2015]

Where Are We?

- Chihuahua or Muffin?

Where Are We?

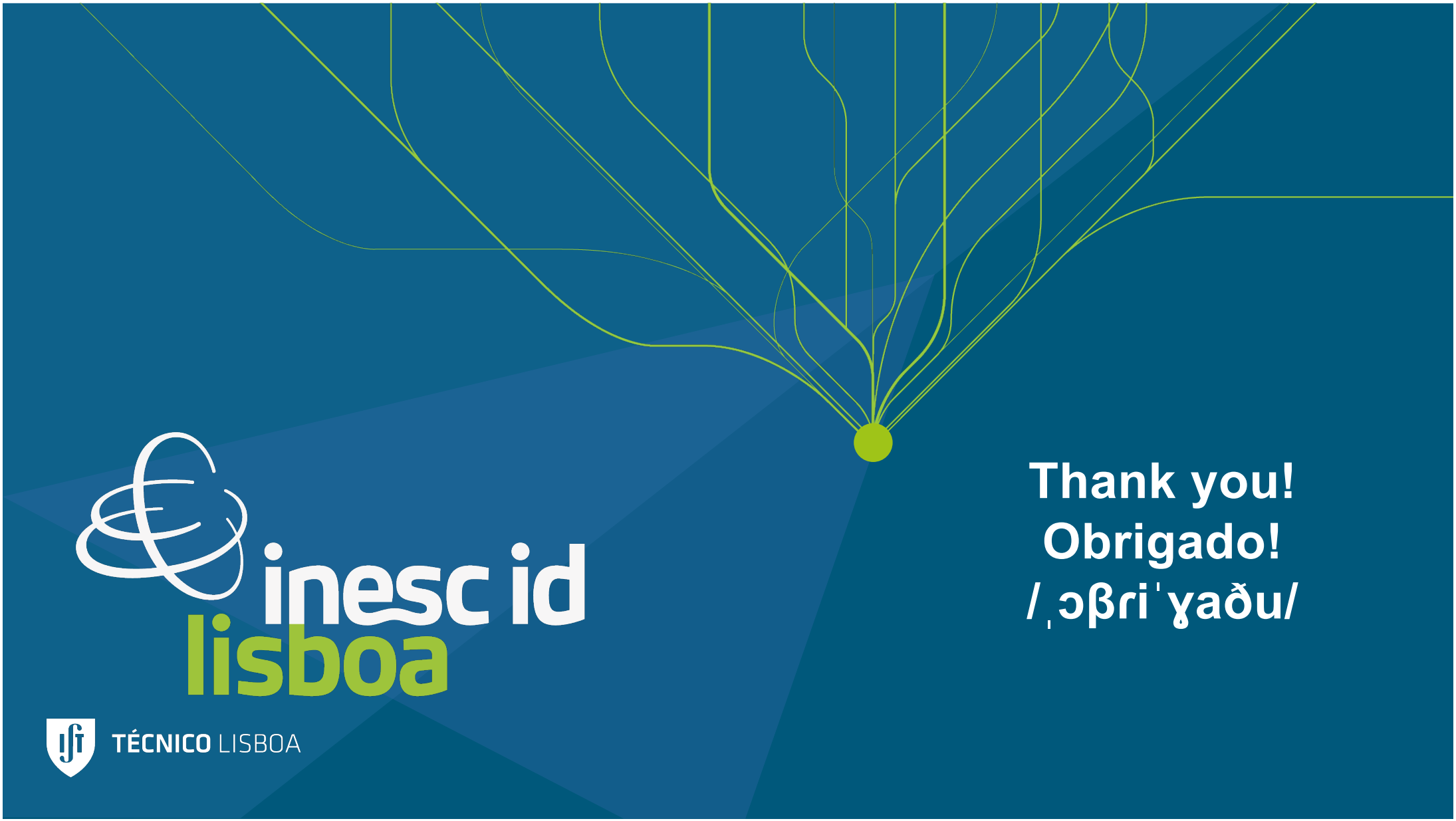
Original Text Prediction = **Positive**. (Confidence = 78%)
The promise of Martin Donovan playing Jesus was, **quite honestly**, enough to get me to see the film. Definitely worthwhile; clever and funny without overdoing it. The low quality filming was **probably** an appropriate effect but ended up being a little too jarring, and the ending sounded more like a PBS program than Hartley. Still, too many memorable lines and great moments for me to judge it harshly.
Adversarial Text Prediction = **Negative**. (Confidence = 59.9%)
The promise of Martin Donovan playing Jesus was, **utterly frankly**, enough to get me to see the film. Definitely worthwhile; clever and funny without overdoing it. The low quality filming was **presumably** an appropriate effect but ended up being a little too jarring, and the ending sounded more like a PBS program than Hartley. Still, too many memorable lines and great moments for me to judge it harshly.

NN Have Issues...

- NN can give amazing results in many real-world tasks
- But NN are black boxes
  - It is almost impossible to understand why they produce their results
  - This can lead to some serious issues

Conclusions

- When should a NN be used?
  - When a large amount of examples (input/output quantitative data) is available and one cannot formulate an algorithmic solution;
  - When continuous learning from previous results is important;
  - ☹ Only if there is no need to extract knowledge from the resulting NN, i.e., understand “why the NN produces a certain result”.


inesc id
lisboa
TÉCNICO LISBOA
Thank you!
Obrigado!
/ˌɔβriˈɣaðu/