- **Adaptive Exponential Integrate&Fire**

$$C_m \frac{\mathrm{d}}{\mathrm{d}t} V_m = g_L(E_L - V_m) + g_L \Delta_T \exp\left(\frac{V - V_T}{\Delta T}\right) + I_{syn} - w$$
$$\tau_w \frac{\mathrm{d}}{\mathrm{d}t} w = a(V_m - E_L) - w$$

Brette and Gerstner, *Journal of Neurophysiology*, 2005

  + Reproduces many neuro-computational properties of spiking and bursting models
  + Biophysical parameters
  - Mathematically untractable

- **Izhikevich**

$$\frac{\mathrm{d}}{\mathrm{d}t} V_m = 0.04 V_m^2 + 5 V_m + 140 + I_{syn} - w$$
$$\frac{\mathrm{d}}{\mathrm{d}t} w = a(b V_m - w)$$

  + Reproduces many neuro-computational properties of spiking and bursting models
  - Non-biophysical parameters
  + Mathematically tractable

Izhikevich, *IEEE Transactions on Neural Networks*, 2003

# Brain-Inspired Learning Machines

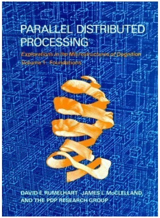
## From Biological Neural Networks to Artificial Neural Networks

Emre Neftci

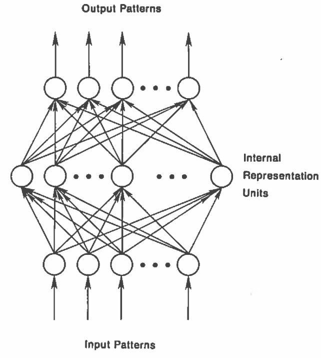
Department of Cognitive Sciences, UC Irvine,

October 7, 2016

"A set of approaches that models artificial intelligence using networks of simple (neuron-like) units."



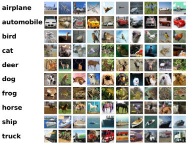

Rumelhart, McClelland, and Group,, 1988




"Deep" artificial neural networks are state-of-the-art in many problems.

A machine learning algorithm is a program that can learn to solve a given task based on **data**


MNIST



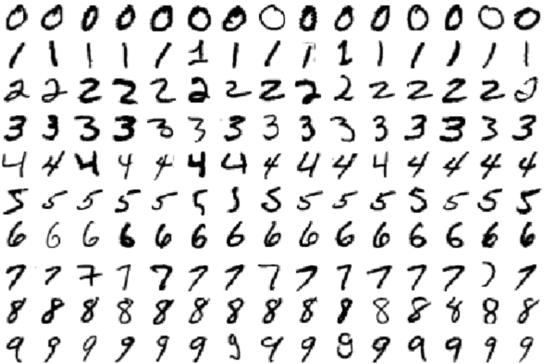
CIFAR-10


ImageNet


DARPA Neovision2 Tower benchmark

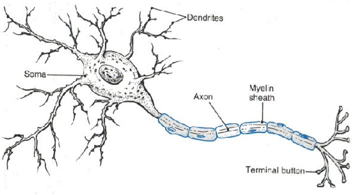
Machine learning models for supervised learning:

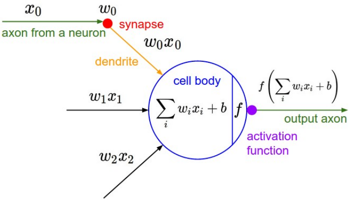
- **Discriminant function**
- Discriminative model
- Generative model



## Pattern Recognition: MNIST hand-written digits

Neural networks use a large dataset to recognize patterns.



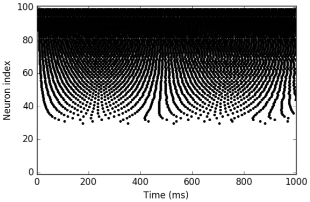



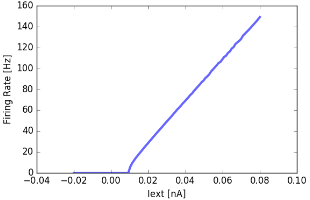

Firing rate neurons are the building blocks of artificial neural networks

# Average Firing Rate - Neuron Activation Function

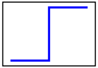
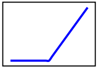
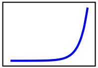
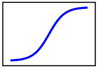
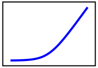
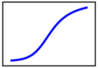
code/brian2_activation_function.py

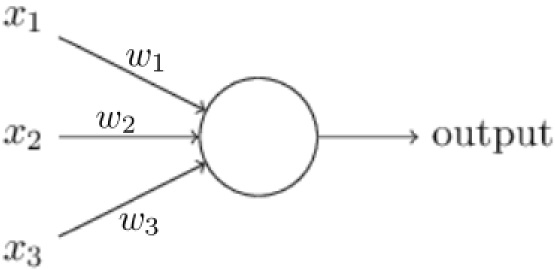
```
from brian2 import *

Cm = 50*pF; gl = 1e-9*siemens; taus = 20*ms
Vt = 10*mV; Vr = 0*mV;
sigma = 0./sqrt(ms)
eqs = '''
dv/dt = - gl*v/Cm
        + sigma*xi*mV
        + iext/Cm : volt (unless refractory)
iext : amp
'''

P = NeuronGroup(100, eqs, threshold='v>Vt', reset='v = Vr',
                refractory=0*ms, method='milstein')

P.v = Vr #Set initial V to reset voltage
P.iext = np.linspace(-.2, .8, 100)*.1*nA

s_mon = SpikeMonitor(P)

run(5.0 * second)
```



## Firing Rate of a Neuron: Commonly Used Activation Functions

| Name | | |
|---|---|---|
| Threshold | $\theta(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$ |  |
| Rectified Linear | $[x]_+ = \max(x, 0)$ |  |
| Exponential | $f(x) = \exp(x)$ |  |
| Sigmoid | $\sigma(x) = \frac{1}{1+e^{-x}}$ |  |
| Soft plus | $S(x) = \log(1 + e^x)$ |  |
| Soft plus with ARP | $S_{RP}(x) = \frac{S(x)}{S(x)+1}$ |  |

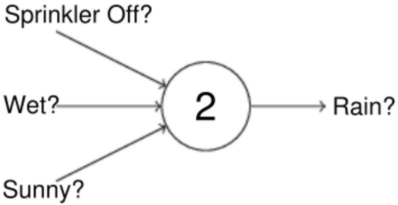



- Three inputs $x_1$, $x_2$, $x_3$ with weights $w_1$, $w_2$, $w_3$, and bias $b$

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j + b \leq 0 \\ 1 & \text{if } \sum_j w_j x_j + b > 0 \end{cases} \quad (1)$$

# Example Perceptron

Rain detector with three sensors

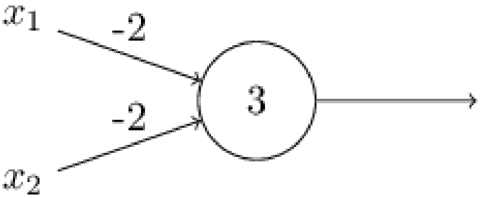



| Sprinkler off | Wet | Sun | $\sum_j w_j x_j + b$ | Rain |
|---|---|---|---|---|
| | | | | |

The Perceptron weigh the sensors inputs (evidence) to come up with a reasonable decision

## Perceptrons as logical gates

Originally, Perceptrons were originally thought of logical gates like `AND`, `OR`, `NOT`, etc.

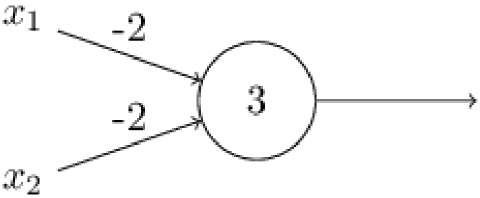



| $x_1$ | $x_2$ | $\sum_j w_j x_j + 3$ | Rain |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 0 | 0 | 3 | 1 |
| 1 | 1 | -1 | 0 |
| 1 | 0 | 1 | 1 |

- This is a NOT AND = NAND gate!

## Perceptrons as logical gates

Originally, Perceptrons were originally thought of logical gates like `AND`, `OR`, `NOT`, etc.




| $x_1$ | $x_2$ | $\sum_j w_j x_j + 3$ | Rain |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 0 | 0 | 3 | 1 |
| 1 | 1 | -1 | 0 |
| 1 | 0 | 1 | 1 |

- This is a NOT AND = NAND gate!
- The NAND gate is universal for computation, that is, we can build any computation out of NAND gates

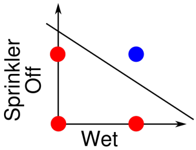
Big deal: Neuron-inspired units are capable of universal computation!

## XOR (exclusive OR)

| $x_1$ | $x_2$ | $\sum_j w_j x_j + b$ | XOR |
|---|---|---|---|
| 0 | 1 | | 0 |
| 0 | 0 | | 1 |
| 1 | 1 | | 0 |
| 1 | 0 | | 1 |

A perceptron is equivalent to a decision boundary.

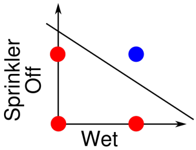
- A straight line can separate blue vs. red



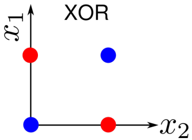

- There is no straight line that can separate blue vs. red

A perceptron is equivalent to a decision boundary.

- A straight line can separate blue vs. red

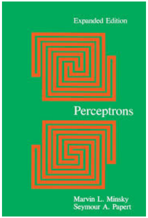



- There is no straight line that can separate blue vs. red



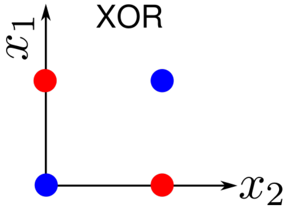

Problems where a straight line can separate two classes are called LINEARLY SEPARABLE

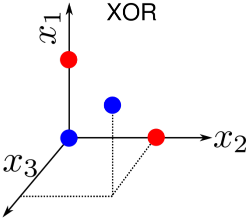
The limitation of a Perceptron to linearly separable problems caused its downfall:




Minsky and Papert,, 1969

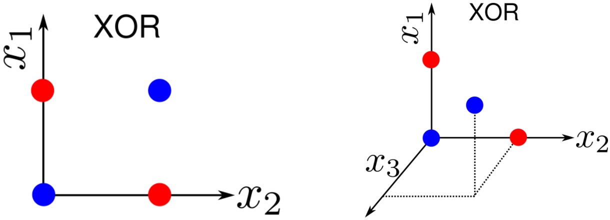
# XOR can be solved with an intermediate perceptron



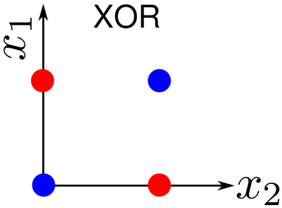



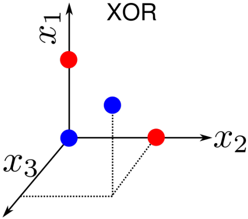

- We need an intermediate unit that is on only when $x_1$ and $x_2$ are both on.

## XOR can be solved with an intermediate perceptron

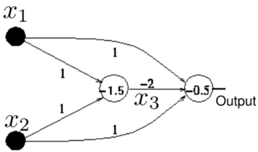



- We need an intermediate unit that is on only when $x_1$ and $x_2$ are both on.
- XOR gate with two perceptrons



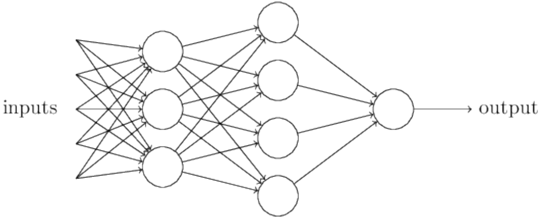




- We need an intermediate unit that is on only when $x_1$ and $x_2$ are both on.
- XOR gate with two perceptrons



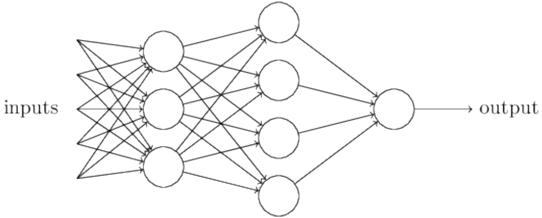

The strategy of extending networks with intermediate units is the key to neural networks' success

Systematically building intermediate layers.




- Each layer makes a decision based on previous inputs
- Subsequent layers make decisions based on more and more abstract information

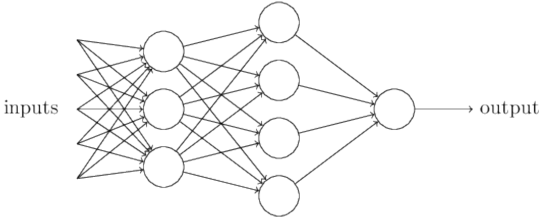
Systematically building intermediate layers.




- Each layer makes a decision based on previous inputs
- Subsequent layers make decisions based on more and more abstract information

But how should we choose the weights and biases? Can we automatically learn them?

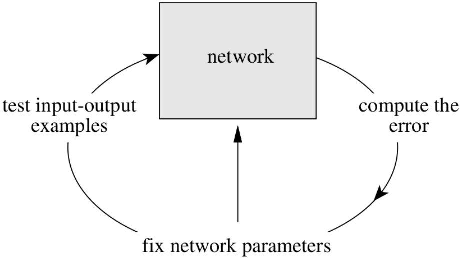
Systematically building intermediate layers.




- Each layer makes a decision based on previous inputs
- Subsequent layers make decisions based on more and more abstract information

But how should we choose the weights and biases? Can we automatically learn them?

Can we automatically learn them?




Error = Number of Misclassified Samples

Learning: Iteratively modify perceptron weights until Error is minimized

To minimize error, repeat for every data sample:

$$\text{new } w_i = w_i + \eta(\text{target} - \text{output})x_i \quad \text{for every i,}$$
$$\text{new } b = b + \eta(\text{target} - \text{output}),$$

where $\eta$ is a “learning rate”.

# The Perceptron Learning Rule

To minimize error, repeat for every data sample:

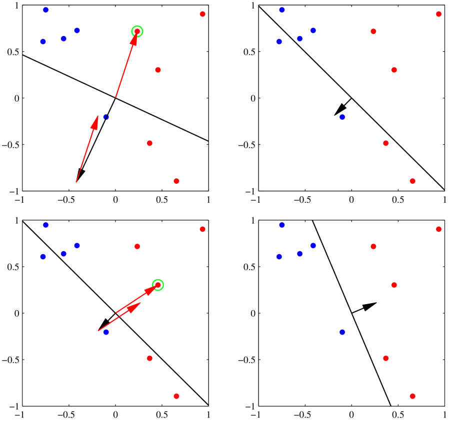
$$\text{new } w_i = w_i + \eta(\text{target} - \text{output})x_i \quad \text{for every i,}$$
$$\text{new } b = b + \eta(\text{target} - \text{output}),$$

where $\eta$ is a “learning rate”.

- If target $=$ output no change
- If target $= 1$ and output $= 0$: add inputs $x_i$ to weights
- If target $= 0$ and output $= 1$: subtract inputs $x_i$ from weights

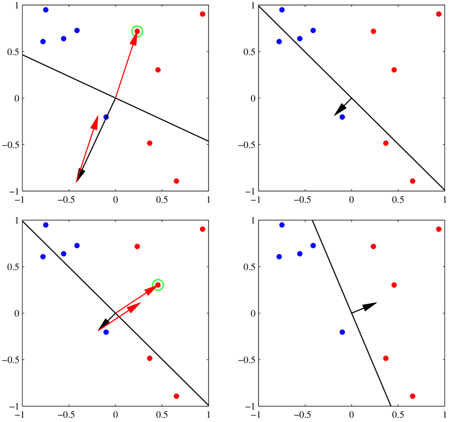
The Perceptron learning rule is a form of supervised learning

# The Perceptron Learning Rule




Bishop,, 2006

# The Perceptron Learning Rule




Bishop,, 2006

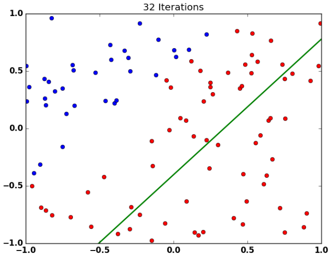
The **Perceptron convergence theorem**: if the training dataset is linearly separable, then the perceptron learning rule is guaranteed to find an exact solution

Rosenblatt, Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms, 1962

# Perceptron Learning Rule in Action

code/ann_demo.py

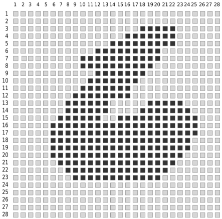
```
import npamlib
npamlib.ann_demo_pla_2D(N=100)
```



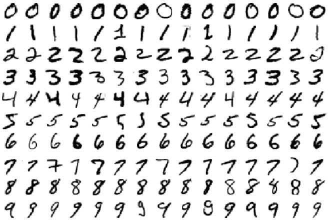
## Neural networks use a large dataset to learn to recognize patterns

Training sample:

$S_1 =$ (value pixel 1, value pixel 2, ..., value pixel 784), label of sample 1




Training set: $\{S_1, \ldots, S_N\}$



# Training perceptrons on mnist

code/ann_train_perceptron.py

```
from npamlib import *

#Load digits 3 and 8 only
data, labels = data_load_mnist([3,8])

#convert labels to True / False
labelsTF = (labels==labels[0])
#Train a data sample with trained perceptron:
w, res = ann_train_perceptron(data[:100], labelsTF[:100], n = 1000, eta = .1)

wbias = w[0]
wdata = w[1:]

#Test a data sample with trained perceptron:
print(ann_perceptron(data[0],w))

#Show stimulus
stim_show(data)
```

code/brian2_perceptron_learn.py

```
data, labels = ann_createDataSet(n_samples) #create 20 2d data samples
blabels = (labels+1)//2 #labels -1,1 to 0,1
data = (1+data)/2 #inputs in the range 0,1

#bias and weights (modify these)
wbias = 0.
wdata = [.1,.1]

## Spiking Network
#Following 2 lines for time-dependent inputs
rate = TimedArray(data*100*Hz, dt = duration)
Pdata = NeuronGroup(data.shape[1], 'rates = rate(t,i) : Hz', threshold='rand()<rates*dt')

#Input bias
Pbias = PoissonGroup(1, rates = 100*Hz)
P = NeuronGroup(1, eqs, threshold='v>Vt', reset='v = Vr',
                refractory=20*ms, method='milstein')

Sdata = Synapses(Pdata, P, 'w : amp', on_pre='isyn += w')
Sdata.connect() #Connect all-to-all
Sdata.w = wdata*nA

Sbias = Synapses(Pbias, P, 'w : amp', on_pre='isyn += w')
Sbias.connect() #Connect all-to-all
Sbias.w = wbias*nA

s_mon = SpikeMonitor(P)
r_mon = PopulationRateMonitor(P) #Monitor spike rates
s_mon_data = SpikeMonitor(Pdata)

run(n_samples*duration)

output_rate = bin_rate(r_mon, duration)
```

- The activation function is the firing rate of the neuron
- Artificial neurons are typically thought to represent the firing rate of spiking neuron
- The Perceptron is a simple decision making and logic circuits
- A Perceptron is equivalent to a linear decision boundary
- A perceptron can only classify linearly separable problems
- Networks of Perceptrons can solve linearly inseparable problems: XOR
- Neural networks can learn from data on how to recognize patterns
- Perceptron learning rule converges if data is linearly separable