

# Brain-Inspired Learning Machines

## Pattern Recognition II: Deep Artificial Neural Networks

Emre Neftci

Department of Cognitive Sciences, UC Irvine,

October 14, 2016

## Gradient-Descent Learning in Neural Networks

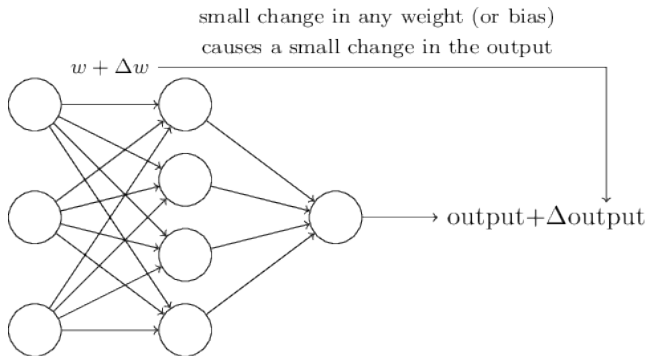
Error = Number of Misclassified Samples

To minimize error, repeat for every data sample:

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

where  $\eta$  is a “learning rate”.

## Continuous Activation Function

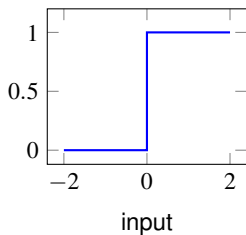


Problem with threshold units: A tiny  $\Delta w$  can induce a flip (large  $\Delta$  output)

## Perceptrons with a “smooth” activation function

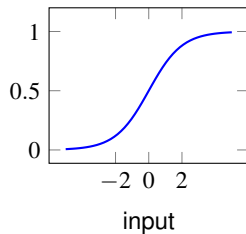
Threshold unit

$$\text{output} = \begin{cases} 0 & \text{if } \text{input} \leq 0 \\ 1 & \text{if } \text{input} > 0 \end{cases}$$



Sigmoid unit

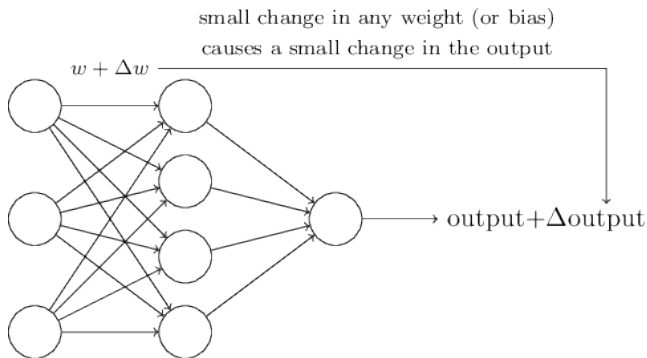
$$\text{output} = \sigma(\text{input}) = \frac{1}{1 + e^{-\text{input}}}.$$



$$\text{input} = \sum_j w_j x_j + b$$

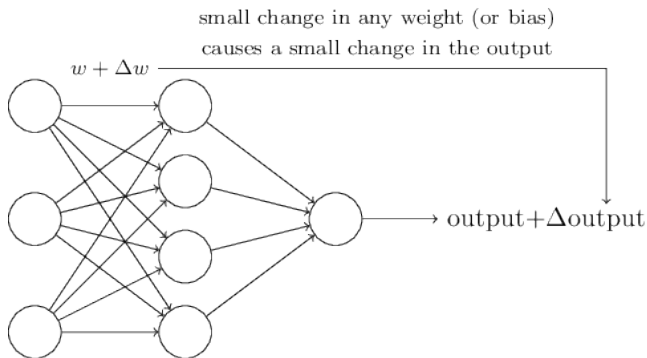
The sigmoid unit is smoother (its derivative is continuous)

## Smooth Activation Function



$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j$$

## Smooth Activation Function



$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j$$

Derivative of Sigmoid:

$$\frac{\partial \text{output}}{\partial w_j}$$

Cost (Error) function: a number representing how the Neural Network performed.

- Perceptrons: Cost function = Number of Misclassified Samples
- Sigmoid Units: Cost function = Mean Squared Error (MSE)

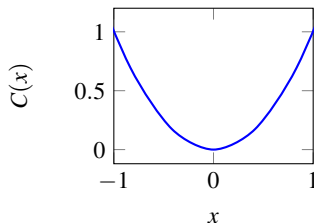
$$C_{\text{MSE}} = \sum_{\text{training set}} \sum_i (\text{output}_i - \text{target}_i)^2.$$

Objective: Minimize the cost function.



## Minimizing Arbitrary Functions by Gradient Descent

Example: Find  $x$  that minimizes  $C(x) = x^2$



Incremental change in  $\Delta x$ :

$$\Delta C \approx \underbrace{\frac{\partial C}{\partial x}}_{=\text{Slope of } C(x)} \Delta x \quad (1)$$

With  $\Delta x = -\eta \frac{\partial C}{\partial x}$ ,  $\Delta C \approx -\eta \left( \frac{\partial C}{\partial x} \right)^2$

Gradient Descent for finding the optimal  $x$

$$\text{new } x = \text{old } x - \eta \frac{\partial C}{\partial x} \quad (2)$$

## Gradient Descent

$$\Delta w_{ij} = -\eta \frac{\partial C_{\text{MSE}}}{\partial w_{ij}}$$

$$\Delta b_i = -\eta \frac{\partial C_{\text{MSE}}}{\partial b_i}$$

Cost function  $C_{\text{MSE}}(w, b)$ :

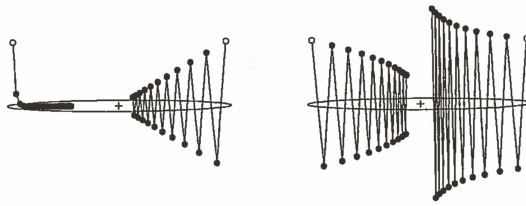
$$C_{\text{MSE}}(w, b) = \sum_{\text{training set}} \sum_i (\text{output}_i - \text{target}_i)^2.$$

$$\frac{\partial C_{\text{MSE}}}{\partial w_{ij}} = 2 \sum_{\text{training set}} \sum_i (\text{output}_i - \text{target}_i) \frac{\partial \text{output}_i}{\partial w_{ij}}.$$

For the Sigmoid neuron:

$$\frac{\partial \text{output}_i}{\partial w_{ij}} = \text{output}_i(1 - \text{output}_i)\text{input}_j$$

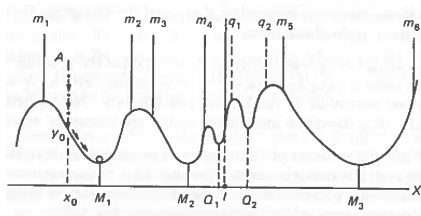
- Adjusting the Learning rate  $\eta$ :



$\eta$  increasing from left to right

$\eta$  too small = slow convergence,  $\eta$  too large = no convergence

- Gradient Descent can get stuck in local minima:



In practice, not a big problem, but it can slow down learning.

Stochastic can escape local minima

# Multinomial Classification with “One-Hot” Representation

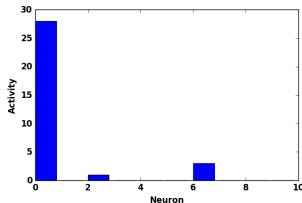
In many datasets, targets are discrete classes, but neural networks units output numbers in the range  $[0, 1]$

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2  
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3  
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4  
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6  
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7  
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8  
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

e.g. MNIST: 10 classes

Representing classes with an output layer:

- Output Layer: one unit per label
- Transform label= $k$  to target vector= $(0, \dots, \underbrace{1}_{\text{position } k}, \dots, 0)$
- Predicted class is defined as the position of output neuron with the highest activity

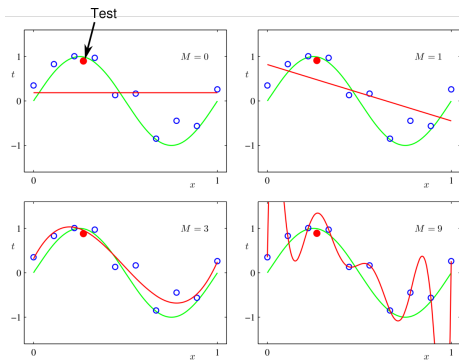


Dataset		
Train 80%	Validation 10%	Test 10%

- Training set: Apply learning rule to sampling in dataset
- Testing set: Set that is never used during training to test the classifier
- Validation set: Set for monitoring overfitting and testing algorithm with different learning parameters

# Underfitting and Overfitting

$M$  is the degree of a fitting polynomial: The higher  $M$ , the more parameters there are.



Too few parameters: Underfitting, Too many parameters: Overfitting

Regularization is a technique used to constrain the complexity of the neural network by introducing *a priori* knowledge

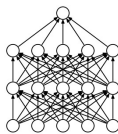
There are many regularization techniques. Most common:

- **$L^p$  norm regularization:** punish large weights ( $p \in \mathbb{N}$ )

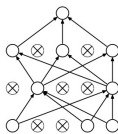
$$\text{New cost function} = C_{MSE} + \lambda \sum_{ij} w_{ij}^p$$

$\lambda$  is the regularization parameter.

- **DropOut:** During training, randomly drop 50% of the outputs



(a) Standard Neural Net



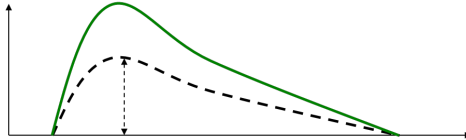
(b) After applying dropout.

- **Data Augmentation:**



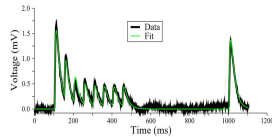
## Synaptic Plasticity: Learning in Spiking Neural Networks

## Long-Term Plasticity



- Induced over seconds, persistence over >10 hours
- Many mechanisms: Change in number of Receptors, Release Probability, ...

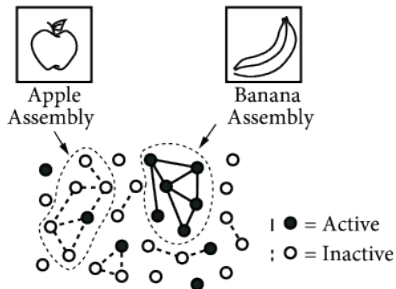
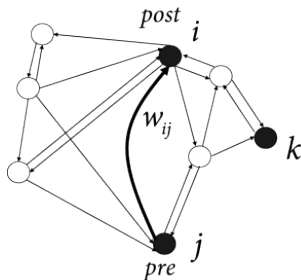
## Short-Term Plasticity



Tsodyks and Markram, *Proceedings of the National Academy of Sciences of the USA*, 1997

- Induced over fractions of a second
- Recovery over seconds
- Change in probability of vesicle release, ...

More on synaptic plasticity Mechanisms: Feldman, *Annual review of neuroscience*, 2009



When an axon of cell  $j$  repeatedly or persistently takes part in activating cell  $i$ , then  $j$ 's efficiency as one of the cells activating  $i$  is increased

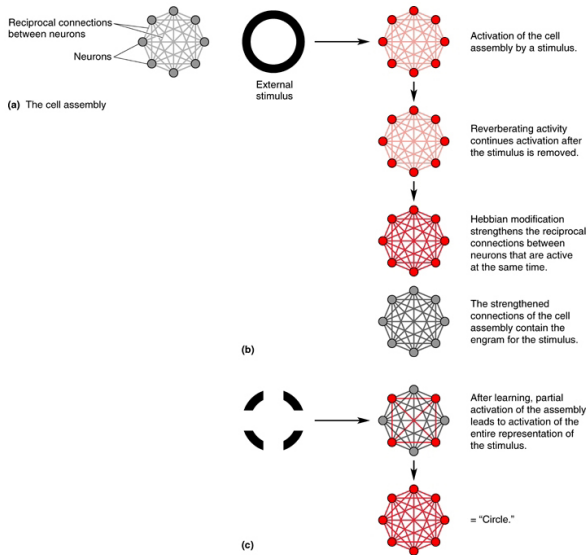
Hebb,, 1949

$$\frac{d}{dt}w_{ij}(t) = \eta \nu_i \nu_j$$

- Plasticity rule operating on local information
- Captures correlations in activity
- Unsupervised

"Neurons that fire together wire together"

# Hebb's Cell Assembly



**Generalized Hebbian learning:** Introduce dependence on pre-synaptic and post-synaptic activities, and the weight itself:

$$\frac{d}{dt} w_{ij}(t) = F(w_{ij}, \nu_i, \nu_j) \quad (3)$$

$$\frac{d}{dt} w_{ij}(t) = a_0(w_{ij}) + a_1^{pre}(w_{ij})\nu_j + a_1^{post}(w_{ij})\nu_i + a_2(w_{ij})\nu_i\nu_j + \dots$$

Pre	On	Off	On	Off
Post	On	On	Off	Off

**Generalized Hebbian learning:** Introduce dependence on pre-synaptic and post-synaptic activities, and the weight itself:

$$\begin{aligned}\frac{d}{dt}w_{ij}(t) &= F(w_{ij}, \nu_i, \nu_j) \\ \frac{d}{dt}w_{ij}(t) &= a_0(w_{ij}) + a_1^{pre}(w_{ij})\nu_j + a_1^{post}(w_{ij})\nu_i + a_2(w_{ij})\nu_i\nu_j + \dots\end{aligned}\tag{3}$$

Pre	On	Off	On	Off
Post	On	On	Off	Off
$\frac{d}{dt}w_{ij}(t) \propto \nu_i\nu_j$	+	0	0	0

**Generalized Hebbian learning:** Introduce dependence on pre-synaptic and post-synaptic activities, and the weight itself:

$$\begin{aligned}\frac{d}{dt}w_{ij}(t) &= F(w_{ij}, \nu_i, \nu_j) \\ \frac{d}{dt}w_{ij}(t) &= a_0(w_{ij}) + a_1^{pre}(w_{ij})\nu_j + a_1^{post}(w_{ij})\nu_i + a_2(w_{ij})\nu_i\nu_j + \dots\end{aligned}\tag{3}$$

Pre	On	Off	On	Off
Post	On	On	Off	Off
$\frac{d}{dt}w_{ij}(t) \propto \nu_i\nu_j$	+	0	0	0
$\frac{d}{dt}w_{ij}(t) \propto \nu_i\nu_j - c$	+	-	-	-

**Generalized Hebbian learning:** Introduce dependence on pre-synaptic and post-synaptic activities, and the weight itself:

$$\begin{aligned}\frac{d}{dt}w_{ij}(t) &= F(w_{ij}, \nu_i, \nu_j) \\ \frac{d}{dt}w_{ij}(t) &= a_0(w_{ij}) + a_1^{pre}(w_{ij})\nu_j + a_1^{post}(w_{ij})\nu_i + a_2(w_{ij})\nu_i\nu_j + \dots\end{aligned}\tag{3}$$

Pre Post	On On	Off On	On Off	Off Off
$\frac{d}{dt}w_{ij}(t) \propto \nu_i\nu_j$	+	0	0	0
$\frac{d}{dt}w_{ij}(t) \propto \nu_i\nu_j - c$	+	-	-	-
$\frac{d}{dt}w_{ij}(t) \propto (\nu_i - c)\nu_j$	(+)	0	-	0



**Generalized Hebbian learning:** Introduce dependence on pre-synaptic and post-synaptic activities, and the weight itself:

$$\begin{aligned}\frac{d}{dt}w_{ij}(t) &= F(w_{ij}, \nu_i, \nu_j) \\ \frac{d}{dt}w_{ij}(t) &= a_0(w_{ij}) + a_1^{pre}(w_{ij})\nu_j + a_1^{post}(w_{ij})\nu_i + a_2(w_{ij})\nu_i\nu_j + \dots\end{aligned}\tag{3}$$

Pre Post	On On	Off On	On Off	Off Off
$\frac{d}{dt}w_{ij}(t) \propto \nu_i\nu_j$	+	0	0	0
$\frac{d}{dt}w_{ij}(t) \propto \nu_i\nu_j - c$	+	-	-	-
$\frac{d}{dt}w_{ij}(t) \propto (\nu_i - c)\nu_j$	(+)	0	-	0
$\frac{d}{dt}w_{ij}(t) \propto (\nu_i - \langle \nu_i \rangle)(\nu_j - \langle \nu_j \rangle)$	+	-	-	+

## Modulated Hebb rule: Neuromodulators + Hebbian Learning

$$\frac{d}{dt}w_{ij}(t) = F(w_{ij}, \nu_i, \nu_j, mod(t)) \quad (4)$$

Example causes of neuromodulation can be rewards, error, attention, novelty.

### Examples:

Reinforcement learning:

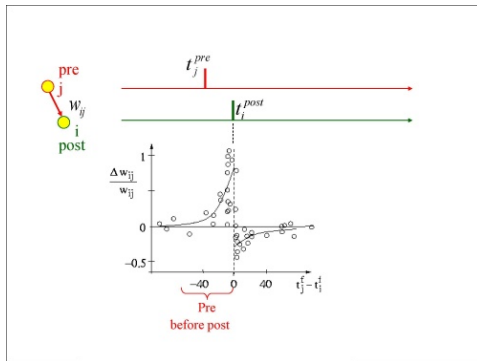
$$\frac{d}{dt}w_{ij}(t) \propto Reward(t)\nu_i\nu_j \quad (5)$$

Florian, *Neural Computation*, 2007

Supervised Learning:

$$\frac{d}{dt}w_{ij}(t) = Error_i(t)a_1^{pre}\nu_j \quad (6)$$

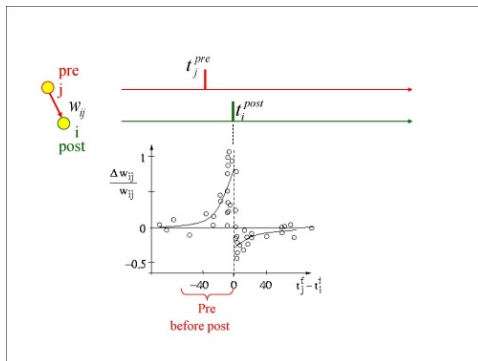
# Spike-Timing Dependent Plasticity



Bi and Poo, *J. Neurosci.*, 1998

Jesper Sjöström and Wulfram Gerstner (2010), *Scholarpedia*, 5(2):1362.

# Spike-Timing Dependent Plasticity (STDP)



Gerstner and Kistler,, 2002

Spike-Time Dependent Plasticity Rule:

$$\Delta w_j = \sum_{f=1}^N \sum_{n=1}^N W(t_i^n - t_j^f) \quad (7)$$

$W$ : Learning Window

$t_i^n$ :  $n$ th spike time of post-synaptic neuron  $i$

$t_j^f$ :  $f$ th spike time of pre-synaptic neuron  $i$

On-line Implementation of the Spike-Time Dependent Plasticity Rule:

$$\begin{aligned}\tau_+ \frac{d}{dt} x_j &= -x_j + a_+ \sum_f \delta(t - t_j^f) \\ \tau_- \frac{d}{dt} y &= -y + a_- \sum_n \delta(t - t^n) \\ \frac{d}{dt} w_j &= x(t) \sum_n \delta(t - t^n) + y(t) \sum_f \delta(t - t_j^f)\end{aligned}\tag{8}$$

$\delta(t)$ : Delta Dirac function (= spike at time  $t$ )

$a_+$ : Amplitude of LTP

$a_-$ : Amplitude of LTD

$\tau_+$ : Temporal window of LTP

$\tau_-$ : Temporal window of LTD

On-line Implementation of the Spike-Time Dependent Plasticity Rule:

$$\begin{aligned}\tau_+ \frac{d}{dt} x_j &= -x_j + a_+ \sum_f \delta(t - t_j^f) \\ \tau_- \frac{d}{dt} y &= -y + a_- \sum_n \delta(t - t^n) \\ \frac{d}{dt} w_j &= x(t) \sum_n \delta(t - t^n) + y(t) \sum_f \delta(t - t_j^f)\end{aligned}\tag{8}$$

$\delta(t)$ : Delta Dirac function (= spike at time  $t$ )

$a_+$ : Amplitude of LTP

$a_-$ : Amplitude of LTD

$\tau_+$ : Temporal window of LTP

$\tau_-$ : Temporal window of LTD

[More on white board](#)

# STDP implementation with Brian2

code/brian2\_stdp.py

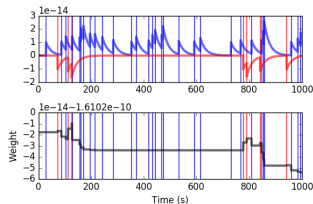
```
from brian2 import *
#Neuron parameters
Cm = 50*pF; gl = 1e-9*siemens; taus = 5*ms
sigma = 3/sqrt(ms)*mV; Vt = 10*mV; Vr = 0*mV;
#STDP Parameters
taupre = 20*ms; taupost = taupre
apre = .01e-12; apost = -apre * taupre / taupost * 1.05

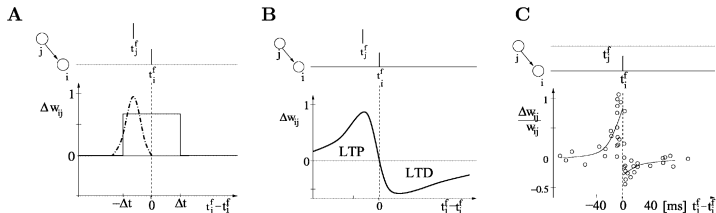
eqs = '''
dv/dt = -gl*v/Cm + isyn/Cm + sigma*x: volt (unless refractory)
disyn/dt = -isyn/taus : amp
'''

Pin = PoissonGroup(10, rates = 30*Hz)
P = NeuronGroup(1, eqs, threshold='v>Vt', reset='v = Vr',
               method='euler', refractory=5*ms)
S = Synapses(Pin, P, '''w : 1
                        dx/dt = -x / taupre : 1
                        dy/dt = -y / taupost : 1''',
             on_pre="''isyn += w*amp
                      x += apre
                      w += y''",
             on_post="''y += apost
                     w += x''")

S.connect()
S.w = '(rand()-.5)*1e-9'
mon = StateMonitor(S, variables=['w','x','y'], record=range(5))
s_mon = SpikeMonitor(P)
p_mon = SpikeMonitor(Pin)

run(1*second, report='text')
```





**Fig. 3A–C.** Learning window. The change  $\Delta w_{ij}$  of the synaptic efficacy depends on the timing of pre- and postsynaptic spikes. **A** The *solid line* indicates a rectangular time window as it is often used in standard Hebbian learning. The synapse is increased if the pre- and the postsynaptic neuron fire simultaneously with a temporal resolution  $\Delta t$ . The *dashed-dotted line* shows an asymmetric learning window useful for sequence learning (Herz et al. 1989; Gerstner and van Hemmen 1993). The synapse is strengthened if the presynaptic spike arrives slightly before the postsynaptic one, and is therefore

partially 'causal' in firing it. **B** An asymmetric biphasic learning window as introduced in model studies of delay selection (Gerstner et al. 1996). A synapse is strengthened (long-term potentiation, *LTP*) if the presynaptic spike arrives slightly before the postsynaptic one, but is decreased (long-term depression, *LTD*) if the timing is reversed. The biphasic learning window is sensitive to the temporal contrast in the input. **C** Experimental results have confirmed the existence of biphasic learning windows. *Data points* redrawn after the experiments of Bi and Poo (1998)

If the pre- and post-synaptic neuron spike times are independent:

$$\left\langle \frac{d}{dt} w_{ij} \right\rangle \cong \nu_i \nu_j \underbrace{\int W(s) ds}_{\text{Area under learning window}} \quad (9)$$



## A More General Spike-Time Dependent Plasticity Rule

$$\begin{aligned}
\frac{d}{dt}w_j = & a_0(w_{ij}) \\
& + \alpha_1^{pre}(w_{ij}) \sum_f \delta(t - t_j^f) \\
& + \alpha_1^{post}(w_{ij}) \sum_n \delta(t - t_i^n) \\
& + x(t) \sum_n \delta(t - t^n) + y(t) \sum_f \delta(t - t_j^f)
\end{aligned} \tag{10}$$

Implements the generalized Hebb rule:

$$\left\langle \frac{d}{dt}w_{ij} \right\rangle \cong a_0(w_{ij}) + \alpha_1^{pre}(w_{ij})\nu_j + \alpha_1^{post}(w_{ij})\nu_i + \nu_i\nu_j \int W(s)ds \tag{11}$$

- Start with `code/brian2_activation_function.py`
- Find a parameter regime in which the activation function is continuous
- Find a function that fits the activation function (e.g. see Sigmoid, Softplus with ARP)
- Starting from least squares, compute the weight update dynamics  $\frac{d}{dt} w$
- Write this rule in the form of generalized STDP
- Propose a spiking network diagram that would implement this rule (hand-in or upload a picture)
- Is your rule “local”? If not how many *different* non-local inputs to you need per neuron?

Optional (Hard):

- Start with `code/brian2_perceptron_learn.py`
- Create targets using one-hot representation
- Implement this rule in Brian2 using (generalized) STDP
- Train, Validate & Test