Unit 1:

Simulating Neurons

To simulate neural networks, we have to simulate neurons. What computational models do people use for neurons?

By the end of this unit, you will be able to...

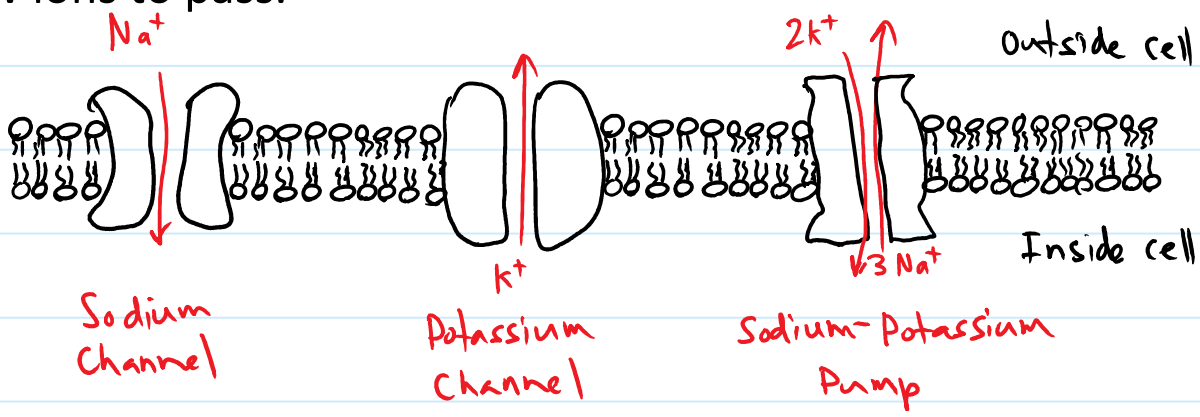
- model the nonlinear nature of how a neuron works, in the form of a famous neuron model.
- describe other, less complicated neuron models, and explain how they differ from other models.
- mathematically formulate the connections between populations of neurons.
- describe how the activity of a spiking neuron can often be summarized using its firing rate.

Hodgkin-Huxley Model

Goal: To see the nonlinear nature of how a neuron works, in the form of a famous neuron model.

Neuron Membrane Potential

Ions are molecules or atoms in which the number of electrons (-) does not match the number of protons (+), resulting in a net charge. Many ions float around in your cells. The cell's membrane, a lipid bi-layer, stops most ions from crossing. However, ion channels embedded in the cell membrane can allow ions to pass.



Sodium-Potassium Pump exchanges 3 Na+ ions inside the cell for 2 K+ ions outside the cell.

- Causes a higher concentration of Na+ outside the cell, and higher concentration of K+ inside the cell.
- It also creates a net positive charge outside, and thus a net negative charge inside the cell.

This difference in charge across the membrane induces a voltage difference, and is called the **membrane potential.**

Action Potential

Neurons have a peculiar behaviour: they can produce a spike of electrical activity called an action potential.
This electrical burst travels along the neuron's axon to its

electrical activity called an action potential.
This electrical burst travels along the neuron's axon to its synapses, where it passes signals to other neurons.

Hodgkin-Huxley Model

Alan Lloyd Hodgkin and Andrew Fielding Huxley received the Nobel Prize in Physiology or Medicine in 1963 for their model of an action potential (spike). Their model is based on the nonlinear interaction between membrane potential (voltage) and the opening and closing of Na+ and K+ ion channels.

sodium potassium

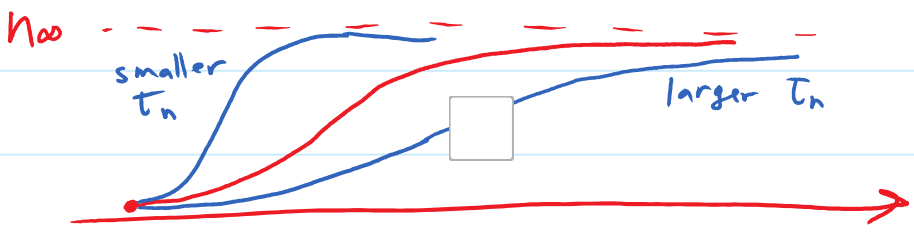
Both Na+ and K+ ion channels are voltage-dependent, so their opening and closing changes with the membrane potential.

Let V be the membrane potential. A neuron usually keeps a membrane potential of around **-70mV.**

The fraction of K+ channels that are open is n^4, where

$$\frac{dn}{dt} = \frac{1}{\tau_n(V)}\left(n_\infty(V) - n\right)$$

time constant; equilibrium solution



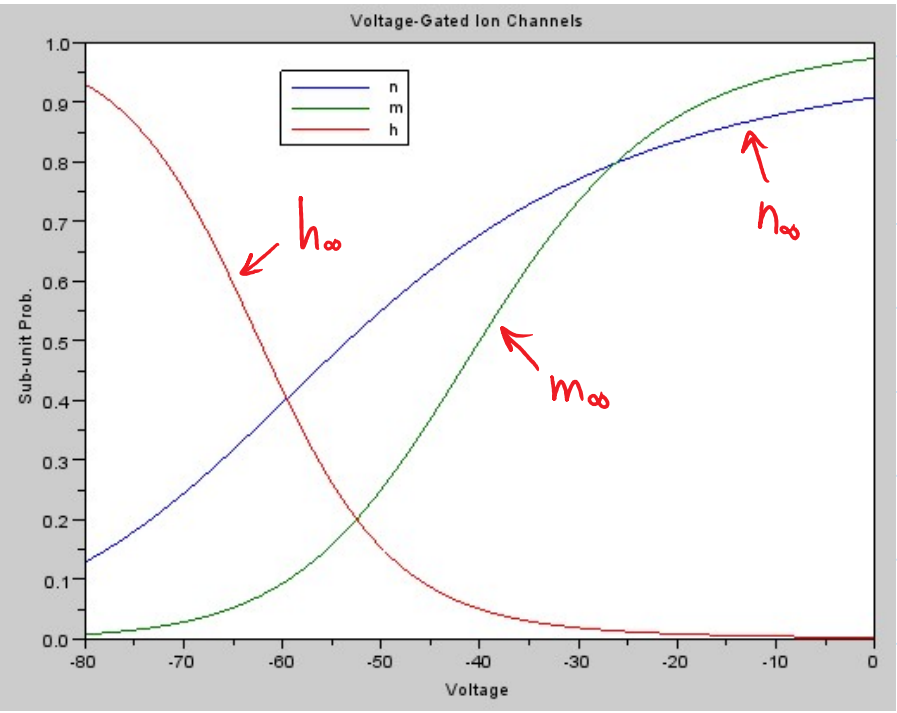
The fraction of Na+ ion channels open is m^3h, where

$$\frac{dm}{dt} = \frac{1}{\tau_m(V)}\left(m_\infty(V) - m\right)$$

$$\frac{dm}{dt} = \frac{1}{\tau_m(V)}\left(m_\infty(V) - m\right)$$

$$\frac{dh}{dt} = \frac{1}{\tau_h(V)}\left(h_\infty(V) - h\right)$$

These curves where measured empirically.



These two channels allow ions to flow into/out of the cell, inducing a current… which affects the membrane potential, V.

"Leak" Current — Current from Na^+ — Current from k^+

$$c\frac{dV}{dt} = J_{in} - g_L(V - V_L) - g_{Na}m^3h(V - V_{Na}) - g_k n^4(V - V_k)$$

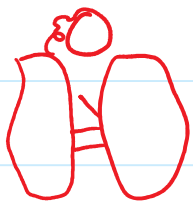
Capacitance

Input current (from other neurons)

max Na^+ conductance

max k^+ conductance

Zero-current potentials (the voltage that would result from equal concentration on both sides of membrane).



This system of four differential equations (DEs) governs the dynamics of the membrane potential.

Notice what happens when the input current is:

- negative
- zero
- slightly positive
- very positive

End of L1

(demo HH)

Simpler Neuron Models

Goal: To look at other, less complicated neuron models.

The HH model is already greatly simplified:

- a neuron is treated as a point in space
- conductances are approximated with formulas
- only considers K+, Na+ and generic leak currents
- etc.

But to model a single action potential (spike) takes many time steps of this 4-D system. However, spikes are fairly generic, and it is thought that the *presence* of a spike is more important than its specific shape.

Leaky Integrate-and-Fire (LIF) Model

The leaky integrate-and-fire (LIF) model only considers the sub-threshold membrane potential (voltage), but does NOT model the spike itself. Instead, it simply records when a spike occurs (ie. when the voltage reached the threshold).

$$C\frac{dV}{dt} = J_{in} - g_L(V - V_L)$$

C: capacitance

g_L: conductance, $g_L = \frac{1}{R}$

$$RC\frac{dV}{dt} = RJ_{in} - (V - V_L)$$

$RC = \tau_m$: time const.

Ohm's Law: Resistance × Current = Voltage

Let $V_{in} = RJ_{in}$

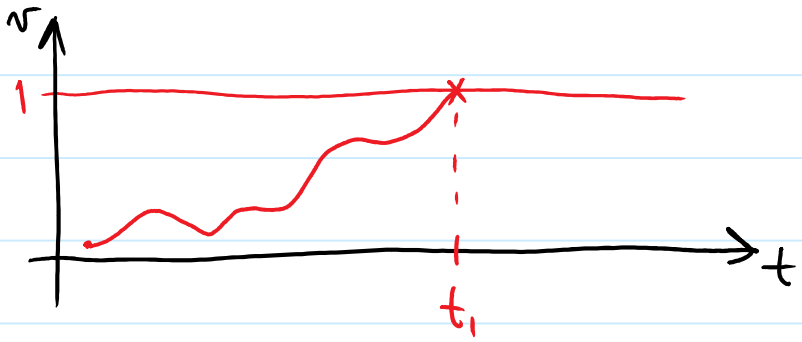
Thus, the voltage can be modelled as

$$\tau_m\frac{dV}{dt} = V_{in} - (V - V_L) \quad \text{for } V < V_{th}$$

Change of variables: $v = \frac{V - V_L}{V_{th} - V_L}$ Then $v < 1$

$$\Rightarrow \tau_m \frac{dv}{dt} = v_{in} - v$$

We integrate the DE for a given input current (or voltage) until v reaches the threshold value of 1.



Then we record a spike at time t_1

After it spikes, it remains dormant during its refractory period, τ_{ref}. (often just a few milliseconds). Then it can start integrating again.

(demo simple_LIF)

What's orange and sounds like a parrot?
A carrot.

LIF Firing Rate

Suppose we hold the input, v_{in}, constant. We can solve the DE analytically between spikes.

Claim: $v(t) = v_{in}\left(1 - e^{-\frac{t}{\tau}}\right)$ is a solution of

$$\tau \frac{dv}{dt} = v_{in} - v, \quad v(0) = 0$$

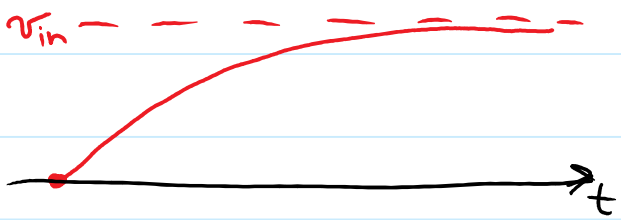
Proof: Plug the proposed solution into the DE and show LHS = RHS.

LHS = $\tau \frac{dv}{dt} = \tau \cdot \frac{1}{\tau}\left(e^{-\frac{t}{\tau}}\right) = \cdots e^{-\frac{t}{\tau}}$

$$LHS = \tau \frac{dv}{dt} = \tau R J_m \frac{-1}{\tau}\left(-e^{-\frac{t}{\tau}}\right) = R J_m e^{-\frac{t}{\tau}}$$

$$RHS = R J_m - v = R J_m - R J_m\left(1 - e^{-\frac{t}{\tau}}\right) = R J_m e^{-\frac{t}{\tau}}$$

What does the solution look like?



If $v_{in} > 1$, then our LIF neuron will spike. At what time will the spike occur (as a function of v_{in})?

$$t_{isi} = \tau_{ref} + t^* \quad \text{where} \quad v(t^*) = 1$$

$$v(t^*) = 1 = v_{in}\left(1 - e^{-\frac{t^*}{\tau}}\right)$$

$$1 - e^{-\frac{t^*}{\tau}} = \frac{1}{v_{in}}$$

$$e^{-\frac{t^*}{\tau}} = 1 - \frac{1}{v_{in}}$$

$$\ln \rightarrow -\frac{t^*}{\tau} = \ln\left(1 - \frac{1}{v_{in}}\right)$$

$$t^* = -\tau \ln\left(1 - \frac{1}{v_{in}}\right)$$

isi = "inter-spike interval"

$$\therefore t_{isi} = \tau_{ref} - \tau \ln\left(1 - \frac{1}{v_{in}}\right) \quad \text{for} \quad v_{in} > 1$$

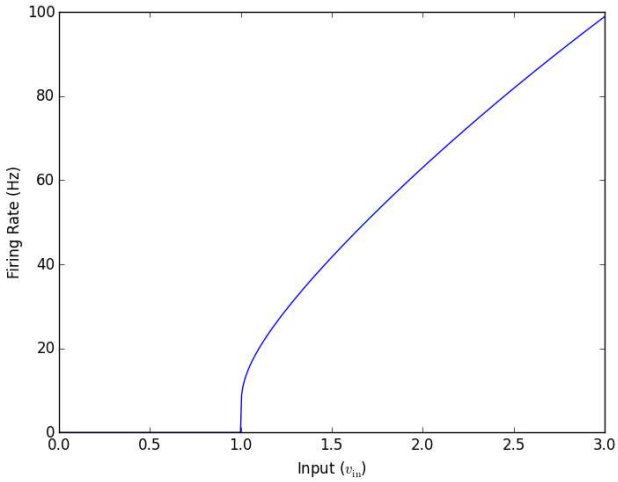
Thus, the steady-state firing rate for a constant input v_{in} is $\frac{1}{t_{isi}}$

$$G(v_{in}) = \begin{cases} \dfrac{1}{\tau_{ref} - \tau_m \ln\left(1 - \frac{1}{v_{in}}\right)} & \text{for } v_{in} > 1 \\ 0 & \text{otherwise} \end{cases}$$

Typical values for cortical neurons:

$$\tau_{ref} = 0.002 \text{ s} \quad (2 \text{ ms})$$

$$\tau_m = 0.02 \text{ s} \quad (20 \text{ ms})$$

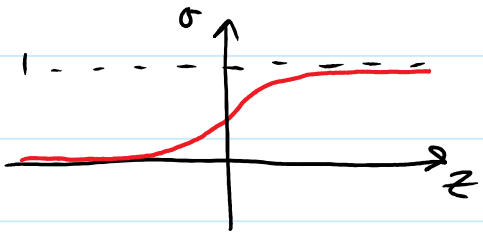


Sigmoid Neuron

As we've seen, the activity of a neuron is very low, or zero, when the input is low, and the activity goes up and approaches some maximum as the input increases. This general behaviour can be represented by a number of different *activation* functions.

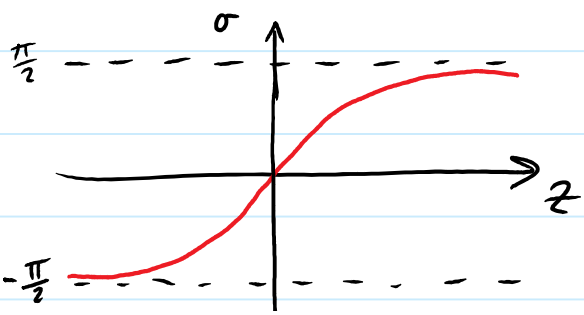
Logistic Curve

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



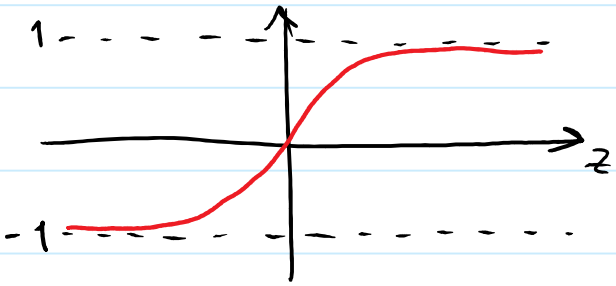
Arctan

$$\sigma(z) = \arctan(z)$$



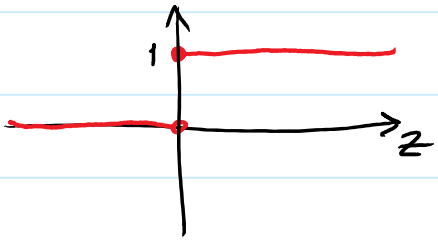
Hyperbolic Tangent

$$\sigma(z) = \tanh(z)$$



Threshold

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



Rectified Linear Unit (ReLU)

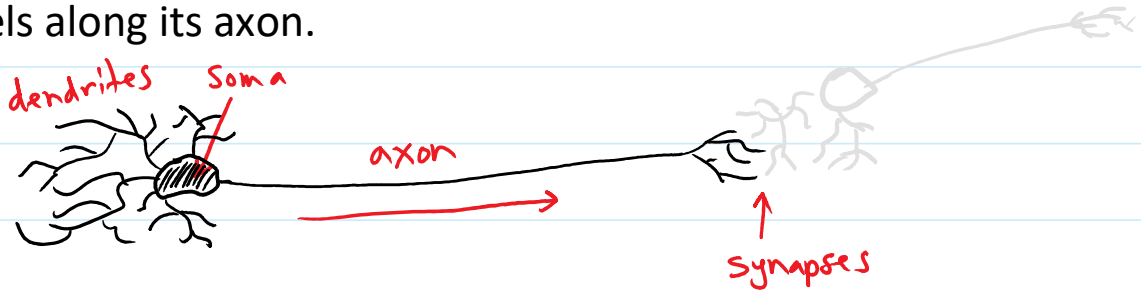
This is just a line that gets capped below at zero.

$$\text{ReLU}(z) = \max(0, z)$$

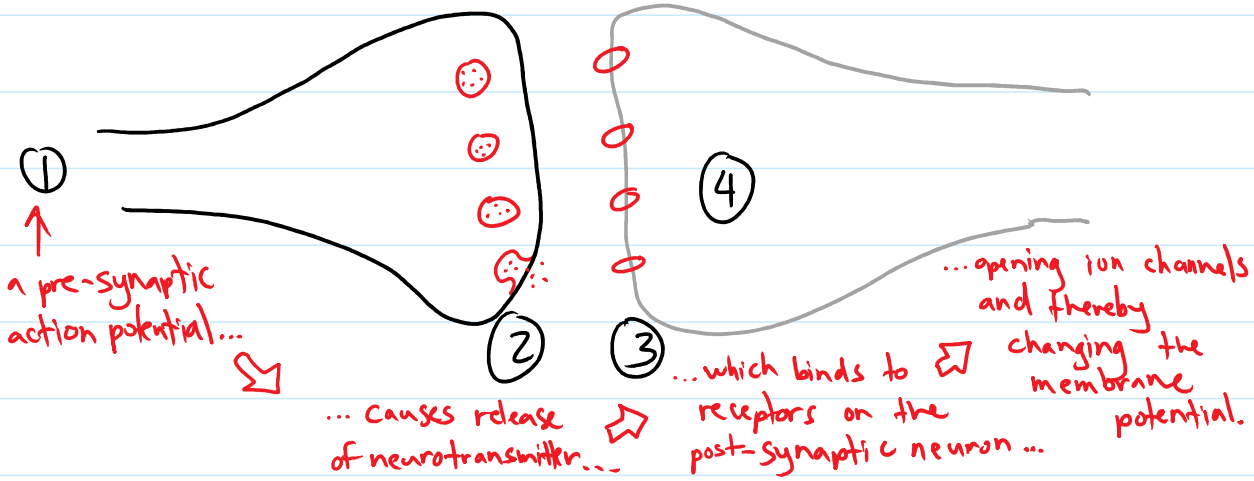
Synapses

Goal: To get an overview of how neurons pass information between them, and how we can model those communication channels.

So far, we've just looked at individual neurons, and how they react to their input. But that input usually comes from other neurons. When a neuron fires an action potential, the wave of electrical activity travels along its axon.



The junction where one neuron communicates with the next neuron is called a **synapse**.

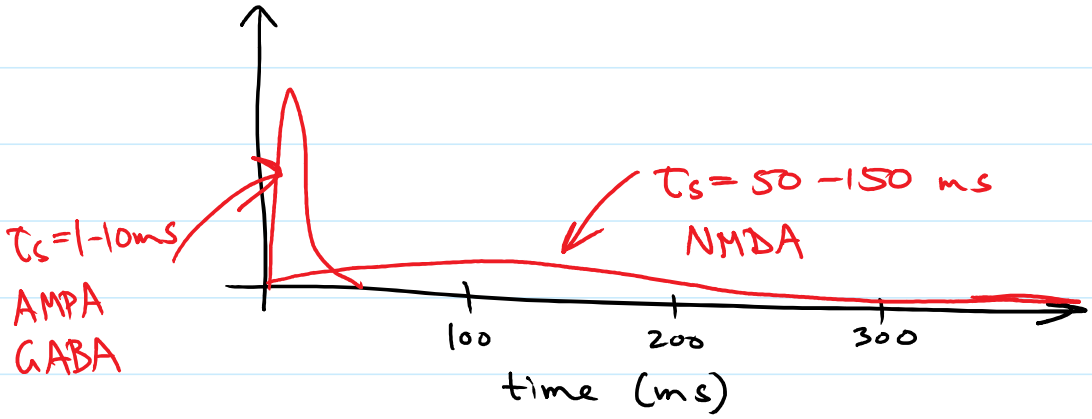


End of L2

Even though an action potential is very fast, the synaptic processes by which it affects the next neuron takes time. Some synapses are fast (taking just about 10 ms), and some are quite slow (taking over 300 ms). If we represent that time constant using τ_s, then the current entering the post-synaptic neuron can be written

$$s(t) = k t^n e^{-\frac{t}{\tau_s}} \quad \text{for some } n$$

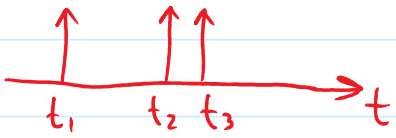
where k is chosen so that $\int_0^\infty s(t)\,dt = 1 \Rightarrow k = \frac{1}{n!\,\tau_s^{n+1}}$



(see Fig. 5.15 in D&A)

The function $s(t)$ is called a Post-Synaptic Current (PSC), or (in keeping with the ambiguity between current and voltage) Post-Synaptic Potential (PSP).

Multiple spikes form what we call a "spike train", and can be modelled as a sum of Dirac delta functions,



$$a(t) = \sum_{p=1}^{3} \delta(t - t_p)$$

Dirac Delta Function

$$\delta(t) = \begin{cases} \infty & \text{if } t = 0 \\ 0 & \text{otherwise} \end{cases}$$

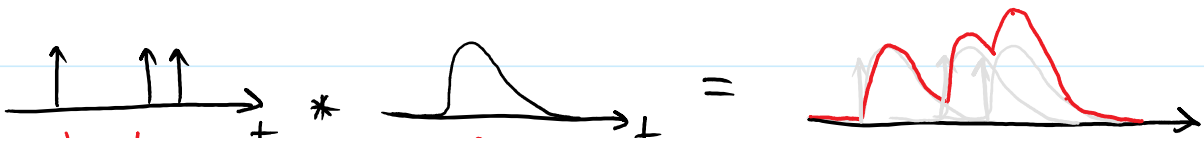
And

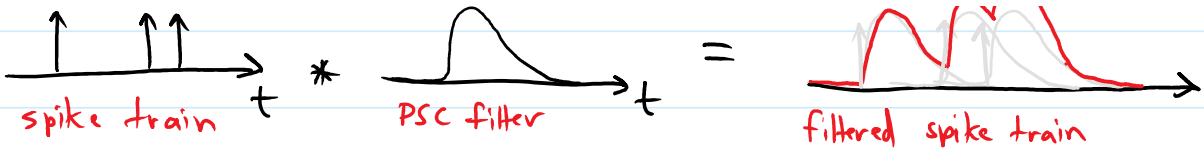
$$\int_{-\infty}^{\infty} \delta(t)\, dt = 1$$

And

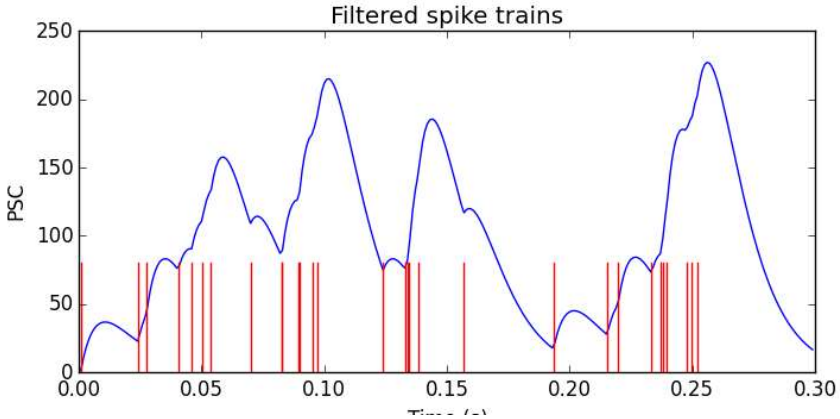
$$\int_{-\infty}^{\infty} f(t)\, \delta(s-t)\, dt = f(s)$$

How does a spike train influence the post-synaptic neuron? Answer: You simply add together all the PSCs, one for each spike. This is actually convolving the spike train with the PSC.





(demo psc)

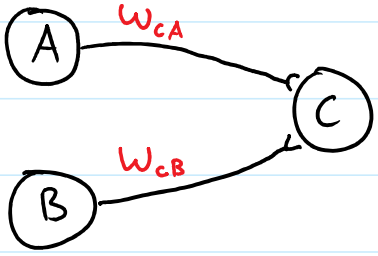


Connection Weight

The total current induced by an action potential onto a particular post-synaptic neuron can vary widely, depending on:

- the number and sizes of the synapses,
- the amount and type of neurotransmitter,
- the number and type of receptors,
- etc.

We can combine all those factors into a single number, the **connection weight**. Thus, the total input to a neuron is a weighted sum of filtered spike-trains.



(demo simple_synapse)

Weight Matrices

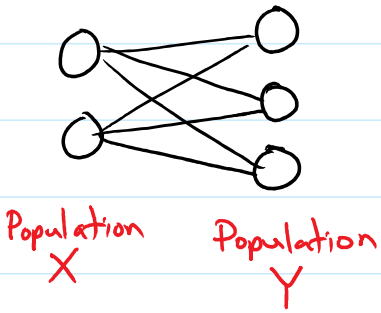
When we have many pre-synaptic neurons, it is more convenient to

use matrix-vector notation to represent the weights and activities.

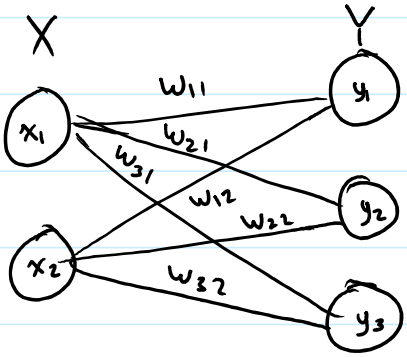
Suppose we have 2 populations, X and Y.

X has N nodes

Y has M nodes



If every node in X sends its output to every node in Y, then we will have a total of $N \times M$ connections, each with its own weight.



$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} = W \in \mathbb{R}^{M \times N}$$

Storing the neuron activities in vectors,

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

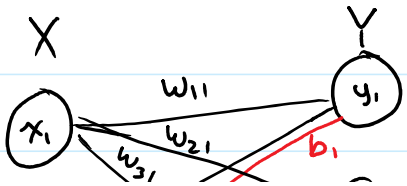
we can compute the input to the nodes in Y using

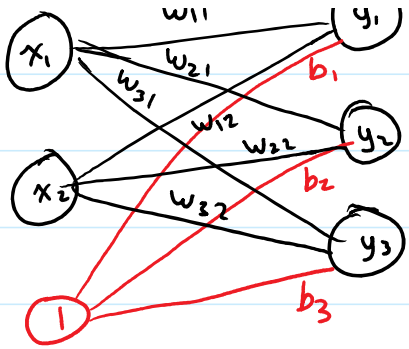
$\vec{z} = W\vec{x} + \vec{b}$ where $\vec{b}$ holds the biases for the nodes in Y.

Thus, $\vec{y} = \sigma(\vec{z}) = \sigma(W\vec{x} + \vec{b})$.

End of L3

Another way to represent the biases, $\vec{b}$.





So $\hat{W} = \left[\begin{array}{c|c} W & \vec{b} \end{array}\right] \left[\begin{array}{c} \vec{x} \\ \hline 1 \end{array}\right] = W\vec{x} + \vec{b}$

Unit 2:

Supervised Learning

To simulate neural networks, we have to simulate neurons. What computational models do people use for neurons'?

By the end of this unit, you will be able to...

- Formulate the problem of supervised learning as an optimization problem.
- List and explain some of the most common loss/cost functions.
- Describe a perceptron, and its limitations.
- Use gradient descent to optimize connection weights.
- Derive and implement the error backpropagation algorithm.
- Use labelled data wisely to train that are generalizable.
- Explain the problem of vanishing/exploding gradients.
- Employ some methods to improve the convergence of our learning.

Neural Learning

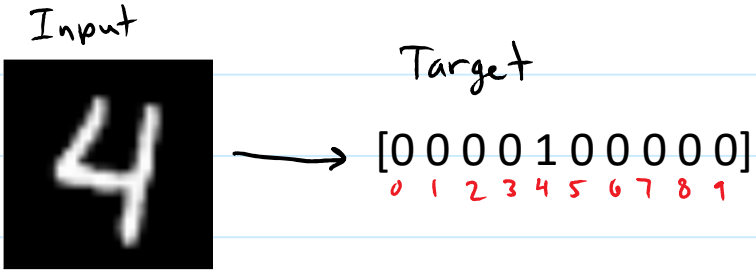
Goal: To formulate the problem of supervised learning as an optimization problem.

Getting a neural network to do what you want usually means finding a set of connection weights that yield the desired behaviour. That is, neural learning is all about adjusting connection weights.

There are three basic categories of learning problems:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

In **supervised learning**, the desired output is known so we can compute the error and use that error to adjust our network.
Example: Given an image of a digit, identify which digit it is.



In **unsupervised learning**, the output is not known (or not supplied), so cannot be used to generate an error signal. Instead, this form of learning is all about finding efficient representations for the statistical structure in the input.
Example: Given spoken English words, transform them into a more efficient representation such as phonemes, and then syllables.

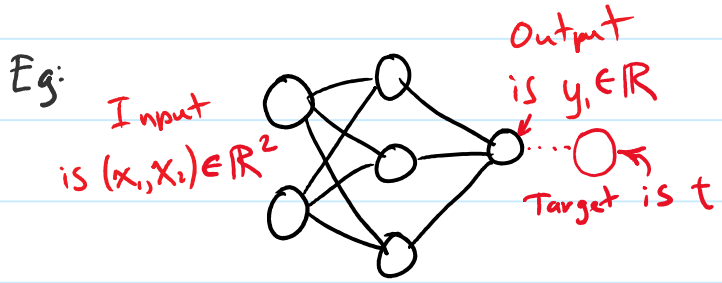
In **reinforcement learning**, feedback is given, but usually less often, and the error signal is usually less specific.
Example: When playing a game, a person knows their play was good

if they win the game. They can try to learn from the moves they made.

In this course, we will mostly focus on supervised learning. But we will look at some examples of unsupervised learning, and possibly some reinforcement learning.

Supervised Learning

Our neural network performs some mapping from an input space to an output space.



We are given training data, with many MANY examples of input/target pairs. This data is (presumably) the result of some consistent mapping process. For example, handwritten digits map to numbers. Or,

A	B	XOR(A,B)
1	1	0
1	0	1
0	1	1
0	0	0

Input $(A, B) \in \{0,1\}^2$

Output/Target $y, t \in \{0,1\}$

We are given inputs and their corresponding targets, one pair (or a few pairs) at a time. Our task is to alter the connection weights in our network so that our network mimics this mapping.

Our goal is to bring the output as close as possible to the target. But what, exactly, do we mean by "close"? For now, we will use the scalar function $E(y,t)$ as an error function, which returns a smaller value as our outputs are closer to the target.

Two common types of mappings encountered in supervised learning are **regression** and **classification**.

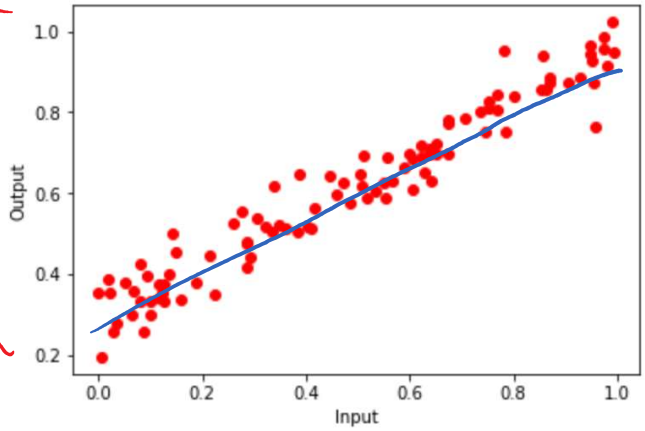
Regression

Output values are a continuous-valued function of the inputs. The outputs can take on a range of values.

Example:

Linear Regression

outputs fall in a range of values.



Classification

Outputs fall into a number of distinct categories.

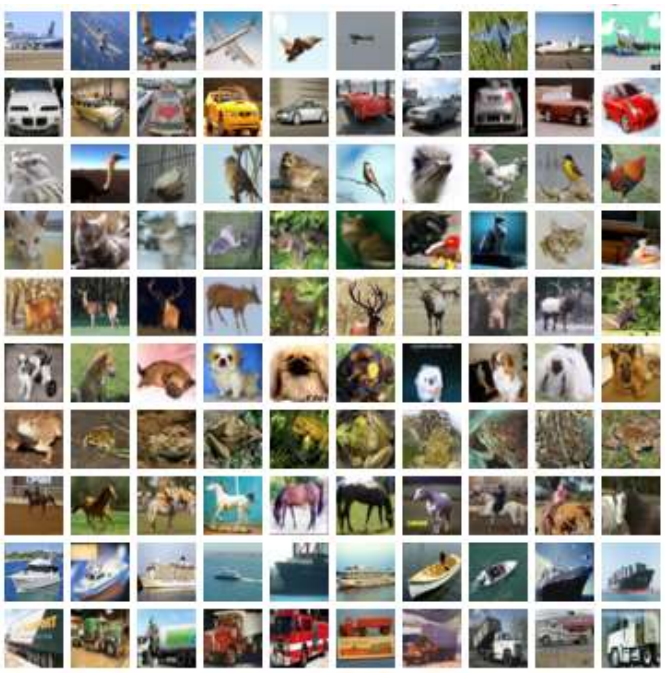
Example:

MNIST

Inputs	Outputs	Inputs	Outputs
	[0 0 0 0 0 0 0 1 0 0]		[0 0 0 0 0 1 0 0 0 0]
	[1 0 0 0 0 0 0 0 0 0]		[0 0 0 0 1 0 0 0 0 0]
	[0 0 0 0 0 0 1 0 0 0]		[0 0 0 0 0 0 0 0 0 1]

CIFAR-10

Inputs



Outputs

airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

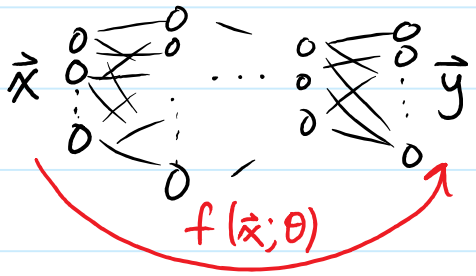
Optimization

Once we have a cost function, our neural-network learning problem can be formulated as an optimization problem.

Let our network be represented by the mapping f, so that

$$\vec{y} = f(\vec{x}; \theta)$$

where θ represents all the weights and biases.



$$\min_{\theta} \mathbb{E}\left[E\left(f(\vec{x};\theta), \vec{t}(\vec{x})\right) \right]_{\vec{x} \in \text{data}}$$

In other words, find the weights and biases that minimize the expected cost between the outputs and the targets.

End of L4

Loss Functions

Goal: To become familiar with some of the most common ways to measure error.

We have to choose a way to quantify how close our output is to the target. For this, we use a "cost function", also known as an "objective function". There are many choices, but here are two commonly-used ones.

For input $\vec{x}$, our target is $\vec{t}(\vec{x})$, and the output of our network is $\vec{y}(\vec{x})$.

Mean Squared Error (MSE)

$$E(\vec{y}, \vec{t}) = \frac{1}{N} \| \vec{y} - \vec{t} \|_2^2 \qquad N \text{ is the \# of samples}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \| \vec{y}_i - \vec{t}_i \|^2$$

The use of MSE as a cost function is often associated with linear activation functions, or ReLU. This is because these activation functions afford a larger output range.

Cross Entropy

Consider a function (or network) with a single output that is either 0 or 1. The task of mapping inputs to the correct output (0 or 1) is a classification problem.

$$x \rightarrow \boxed{f(x;\theta)} \rightarrow y \in [0,1]$$

Suppose we are given a training set,

$$\{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\}$$

Suppose we are given a training set,

$$\{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\}$$

where the true class is expressed in the target, $t_i \in \{0, 1\}$

If we suppose that y_i isk the probability that $x_i \to t_i$,

$$y_i = P(x_i \to 1 \mid \theta) = f(x_i, \theta)$$

then we can treat it as a Bernoulli distribution.

i.e. $P(x_i \to 1 \mid \theta) = y_i$ i.e. $t_i = 1$

Thus $P(x_i \to 0 \mid \theta) = 1 - y_i$ i.e. $t_i = 0$

Or

$$P(x_i \to t_i \mid \theta) = y_i^{t_i}(1-y_i)^{1-t_i}$$ works for both cases

Based on those probabilities, the likelihood of observing our training dataset is

$$P(x_1, \ldots, x_N, t_1, \ldots, t_N) = \prod_{i=1}^{N} P(x_i \to t_i)$$

$$= \prod_{i=1}^{N} y_i^{t_i}(1-y_i)^{1-t_i}$$

As is common practice in probability, it is more convenient to look at the **negative log-likelihood,**

$$-\ln P(x_1, \ldots, t_N) = -\ln \prod_{i=1}^{N} y_i^{t_i}(1-y_i)^{1-t_i}$$

$$= -\sum_{i=1}^{N} \ln y_i^{t_i} + \ln(1-y_i)^{1-t_i}$$

$$\therefore E(y, t) = -\sum_{i=1}^{N} t_i \ln y_i + (1-t_i)\ln(1-y_i) = E_t[-\ln y]$$

This log-likelihood formula is the basis of the **cross-entropy** cost function,

Cross entropy assumes that the output values are in the range [0, 1]. Hence, it works nicely with the logistic activation function.

SoftMax

SoftMax is like a probability distribution (or probability vector), so its elements add to 1. If z is the drive (input) to the output layer, then

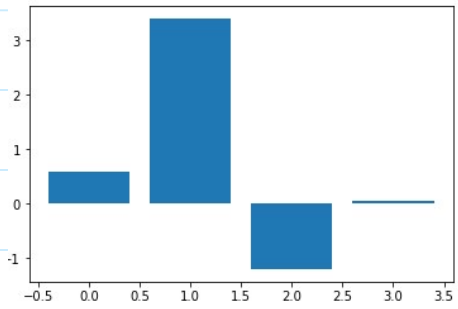
$$\text{Softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

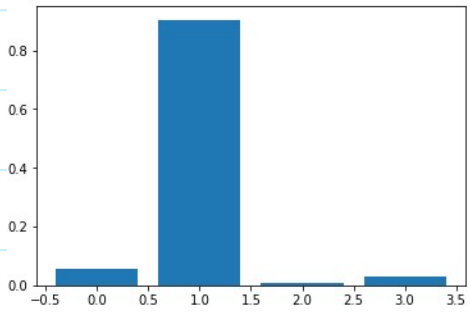
Then, by definition,

$$\sum_i \text{softmax}(z)_i = 1$$

Example:

$z = [0.6, 3.4, -1.2, 0.05] \xrightarrow{\text{softmax}} y = [0.06,\ 0.90,\ 0.009,\ 0.031]$





One-Hot

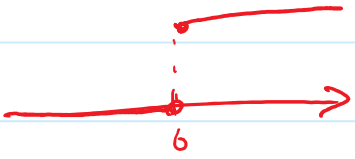
One-Hot is the extreme of the softmax, where only the largest element remains nonzero, while the others are set to zero.

$z = [0.6, 3.4, -1.2, 0.05] \xrightarrow{\text{one hot}} y = [0, 1, 0, 0]$

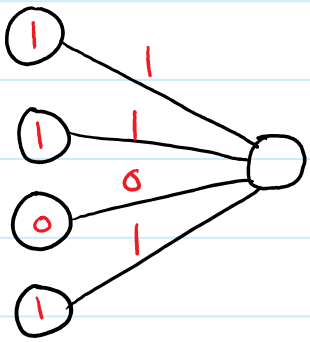
Perceptrons

Goal: To see a simple neural learning algorithm, and understand its limitations.

Let's look at a simple neural network that aims to recognize certain input patterns.



For example, let the output node be a simple threshold neuron, and suppose we want the output node to be 1 when the input is [1, 1, 0, 1], and zero otherwise.



Notice that if we set the weights to [1, 1, 0, 1] (matching the input), then we maximize the input to the output node.

i) $[1\ 1\ 0\ 1] \cdot [1\ 1\ 0\ 1] = 3 \rightarrow \sigma(3) = 1$ (weights · input)

But what about other inputs?

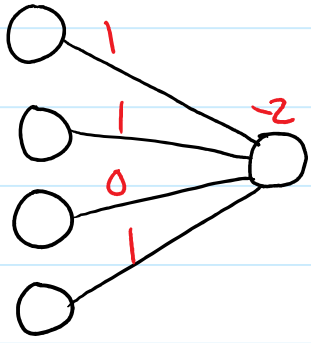
ii) $[1\ 1\ 0\ 1] \cdot [0\ 1\ 1\ 0] = 1 \rightarrow \sigma(1) = 1$

We need the un-matching input to give us a negative value so that the output node returns zero.

Solution: a negative bias

i) $[1\ 1\ 0\ 1] \cdot [1\ 1\ 0\ 1] - 2 = 1 \rightarrow \sigma(1) = 1$

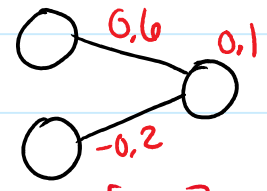
ii) $[1\ 1\ 0\ 1] \cdot [0\ 1\ 1\ 0] - 2 = -1 \rightarrow \sigma(-1) = 0$



Can we find the weights and bias automatically so that our perceptron produces the correct output for a variety of inputs?

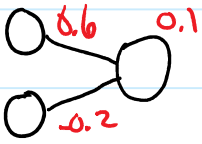
To see an approach, let's look at a 2-D case.

To see an approach, let's look at a 2-D case.
Suppose the 4 different inputs are:
[0, 0], [0, 1], [1, 0], and [1, 1]
And their corresponding outputs are
0, 1, 1, 1 (an "OR" gate)
Also, we will use the L1 error: $E(y,t) = t - y$

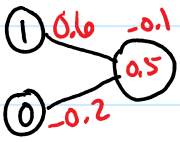


$w = \begin{bmatrix} 0.6 \\ -0.2 \end{bmatrix}$ $b = 0.1$

Start with random weights

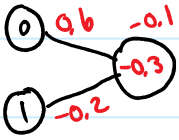


Input [1, 0] target 1



→ 1 $E = 0$ ✓

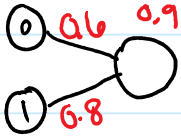
Input [0, 1] target 1



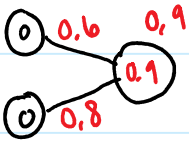
→ 0 $E = 1$

$w \leftarrow w + Ex$

$b \leftarrow b + Et$



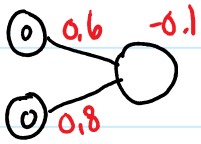
Input [0, 0] target 0



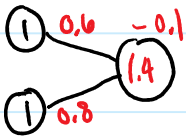
→ 1 $E = -1$

$w \leftarrow w + Ex$

$b \leftarrow b + Et$

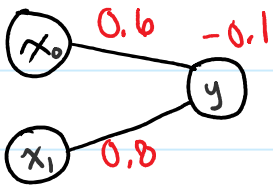


Input [1, 1] target 1



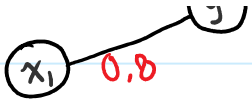
→ 1 $E = 0$

Graphical Interpretation

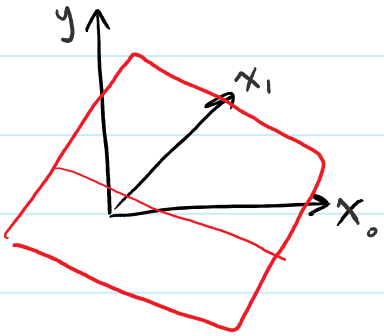


$y = 0.6x_0 + 0.8x_1 - 0.1$

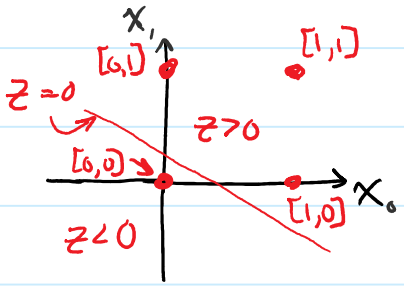
this is a linear equation... the equation of



this is a linear equation... the equation of a plane in 3-D.



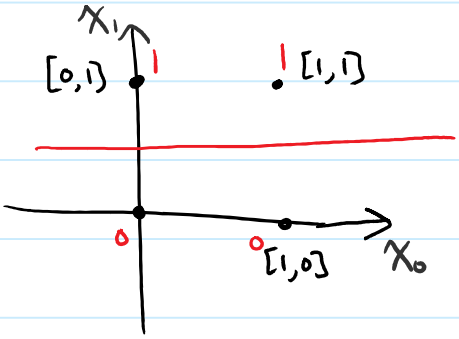
Looking down on the x_0-x_1 plane,

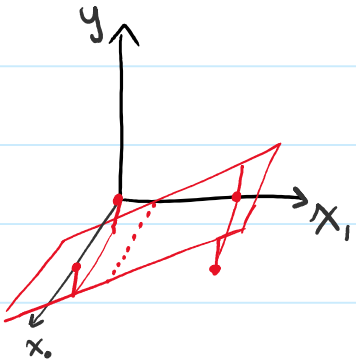


Finding the weights and bias is the same as finding a linear classifier… a linear function that returns a positive value for the inputs that should yield a 1, and a negative value for the inputs that should yield a 0.

Another example:

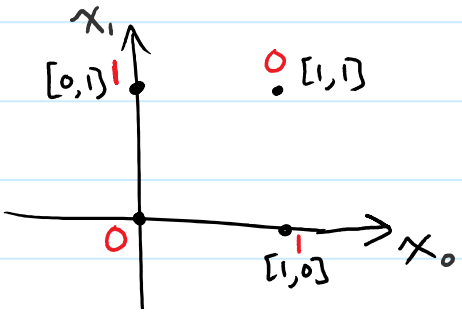
[0, 0] -> 0 [0, 1] -> 1 [1, 0] -> 0 [1, 1] -> 1 Echo x_1-value



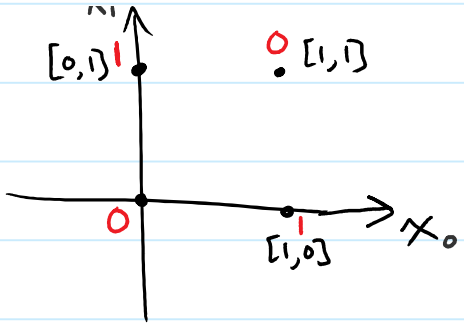


A final example:

[0, 0] -> 0 [0, 1] -> 1 [1, 0] -> 1 [1, 1] -> 0 "XOR"



there is NO linear classifier that will work for the XOR



there is NO linear classifier that will work for the XOR

Perceptrons are simple, two-layer neural networks, so only work for linearly separable data.
If you want to handle non-linearly-separable data, your neural network is going to need more layers.
But they give us our first glimpse at a learning algorithm.

(demo perceptron)

Gradient Descent Learning

Goal: To see how we can use a simple optimization method to tune our network weights.

The operation of our network can be written

$$\vec{y} = f(\vec{x}; \theta)$$

So, if our cost function is $E(\vec{y}, \vec{t})$, where $\vec{t}$ is the target, then neural learning becomes the optimization problem

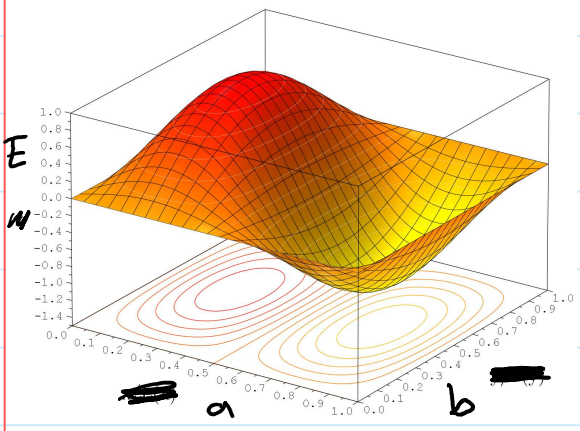
$$\min_{\theta} \mathbb{E}\left[E(f(\vec{x};\theta), \vec{t}(\vec{x})) \right]_{\vec{x} \in \text{data}}$$

We can apply gradient descent to E, using the gradient

$$\nabla_\theta E = \left[\frac{\partial E}{\partial \theta_0} \quad \frac{\partial E}{\partial \theta_1} \quad \cdots \quad \frac{\partial E}{\partial \theta_p} \right]^T$$

Gradient-Based Optimization

If you want to find a local maximum of a function, you can simply start somewhere, and keep walking uphill.
For example, suppose you have a function with two inputs, $E(a, b)$.
You wish to find a and b to maximize E.



https://commons.wikimedia.org/wiki/File:2D_Wavefunction_(2,1)_Surface_Plot.png

We are trying to find the parameters $(\bar{a}, \bar{b})$ that yield the maximum value of E.

i.e. $(\bar{a}, \bar{b}) = \underset{(a,b)}{\text{argmax}}\ E(a,b)$

No matter where you are, "uphill" is in the direction of the gradient vector,

$$\nabla E(a,b) = \left[\frac{\partial E}{\partial a}, \frac{\partial E}{\partial b} \right]^T$$

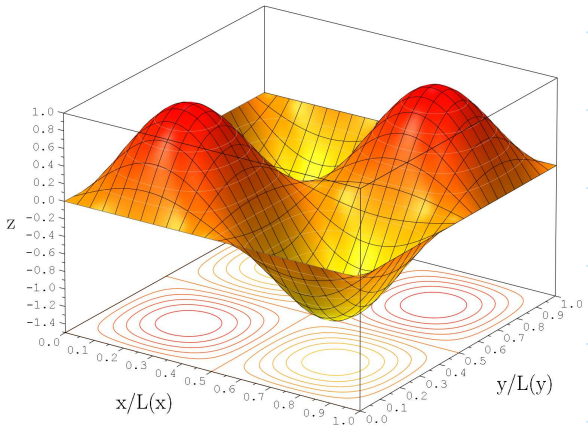
Gradient ascent is an optimization method where you step in the direction of your gradient vector.
If your current position is (a_n, b_n), then

$$(a_{n+1}, b_{n+1}) = (a_n, b_n) + k \nabla E(a_n, b_n)$$

where k is your step multiplier.

Gradient **DESCENT** aims to **minimize** your objective function. So, you walk downhill, stepping in the direction **opposite** the gradient vector.

Note that there is no guarantee that you will actually find the global optimum. In general, you will find a local optimum that may or may not be the global optimum.

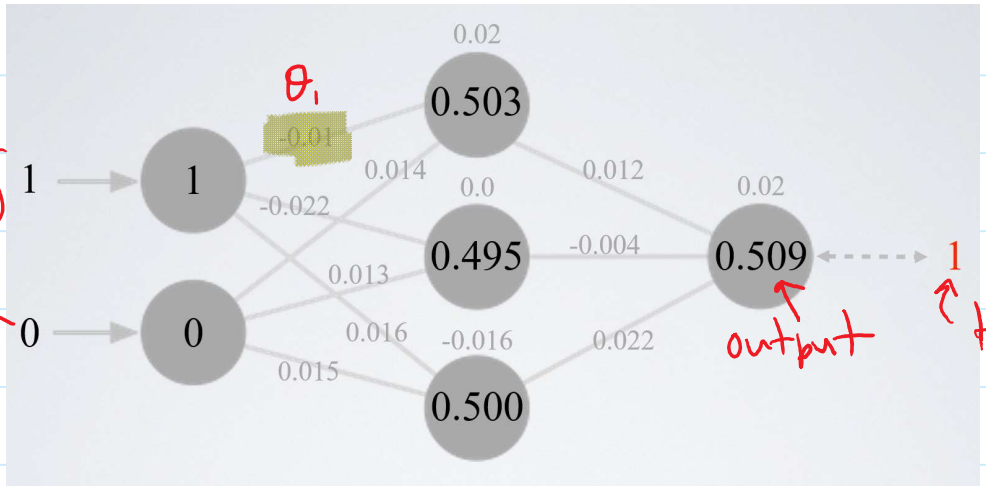


Approximating the Gradient Numerically
We can estimate the partial derivatives in the gradient using finite-differencing.

Finite-Difference Approximation
For a function $f(\theta)$, we can approximate $\frac{df}{d\theta}$ using

$$\frac{df}{d\theta} \approx \frac{f(\theta+\Delta\theta) - f(\theta-\Delta\theta)}{2\Delta\theta}$$

As an example, consider this network:



It's a neural network, with connection weights and biases shown.
We can model the action of the entire network using

$$\vec{y} = f(\vec{x}; \theta)$$

And we can formulate the problem as

$$\min_{\theta} E\left(f(\vec{x}, \theta), \vec{t}(\vec{x})\right)$$

Or, more compactly, $\min_{\theta} \bar{E}(\theta)$ where $\bar{E}(x) = E\left(f(\vec{x}, \theta), \vec{t}(\vec{x})\right)$

Consider θ_1 on its own. With $\theta_1 = -0.01$, our network output is

$y = 0.509$

This gives

$\bar{E}(0.01) = 0.24113$

What if we perturb θ_1, so that $\theta_1 = -0.01 + \mathbf{0.5} = 0.49$.
Then our output is

$y = 0.5093$

This yields

$\bar{E}(0.49) = 0.240761$

If, instead, we perturb θ_1 so that $\theta_1 = -0.01 - \mathbf{0.5} = -0.51$,
then our output is

$y = 0.5086$

which gives

$\bar{E}(x+1) = 0.241509$

$y = 0.5006$

which gives $\bar{E}(0.51) = 0.241509$

In summary,

Parameters	MSE
$\theta_1 + \Delta\theta$	0.240761
θ_1	0.24113
$\theta_1 - \Delta\theta$	0.241509

We can estimate $\frac{\partial \bar{E}}{\partial \theta_1}$ using

$$\frac{\partial \bar{E}}{\partial \theta_1} \approx \frac{\bar{E}(0.51) - \bar{E}(-0.49)}{1}$$

$$= -0.0007475$$

Obviously, increasing θ_1 seems to be the right thing to do.

$$\theta_1 \leftarrow 0.01 - k(-0.0007475)$$

$$= 0.01 + k\,0.0007475$$

End of L6

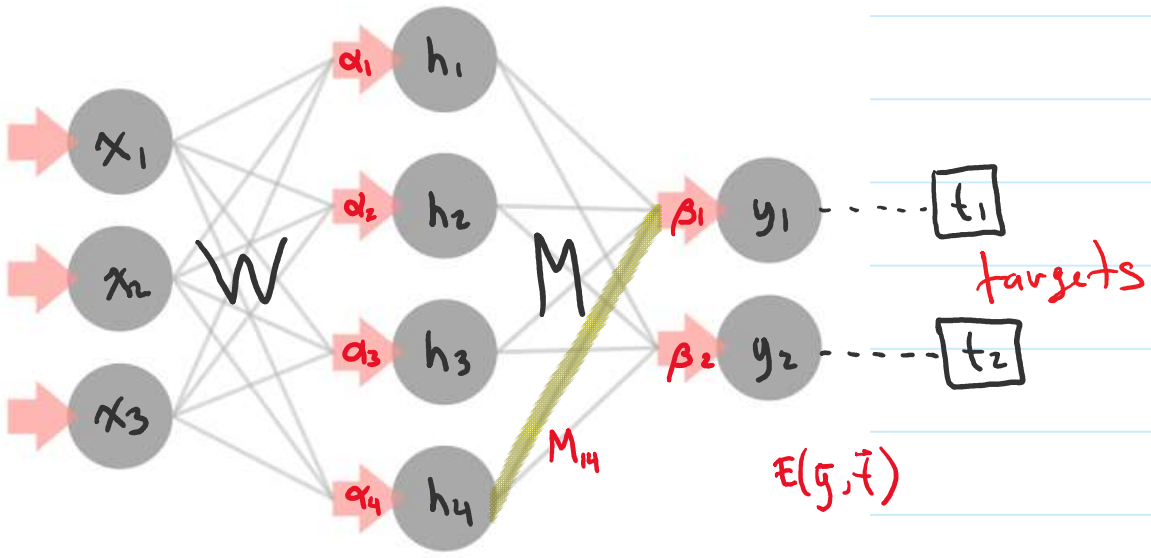
(demo: XOR example)

Error Backpropagation

Goal: To find an efficient method to compute the gradients for gradient-descent optimization.

We can apply gradient descent on a multi-layer network, again using chain rule to calculate the gradients of the error with respect to deeper connection weights and biases.

Consider the network



α_i is the input current to hidden node i.

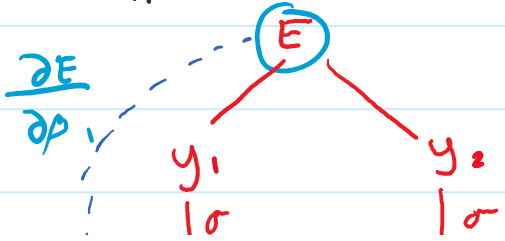
β_j is the input current to output node j.

For our cost (loss) function, we will use $E(\vec{y}, \vec{t})$

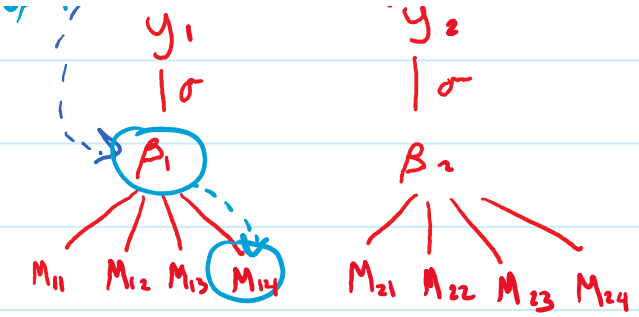
For learning, suppose we want to know, $\frac{\partial E}{\partial M_{14}}$

eg. M_{14}

$$\frac{\partial E}{\partial M_{14}} = \frac{\partial E}{\partial \beta_1} \frac{\partial \beta_1}{\partial M_{14}}$$



$$\frac{\partial E}{\partial M_{14}} = \frac{\partial E}{\partial \beta_1} \frac{\partial \beta_1}{\partial M_{14}}$$



Recall, $E(\vec{y}, \vec{t})$

$$= E\left(\sigma(\underbrace{M\vec{h} + \vec{b}}_{\vec{\beta}}), \vec{t}\right)$$

$$\therefore \frac{\partial E}{\partial \beta_1} = \frac{\partial E}{\partial y_1} \frac{d y_1}{d \beta_1}$$ ← We'll revisit this later

Thus, $\frac{\partial E}{\partial M_{14}} = \frac{\partial E}{\partial y_1} \frac{d y_1}{d \beta_1} \frac{\partial \beta_1}{\partial M_{14}}$

↑

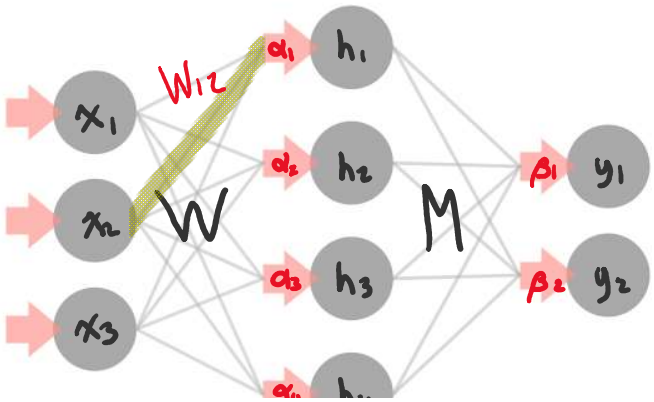
Recall $\beta_1 = \sum_{i=1}^{4} M_{1i} h_i + b_i$

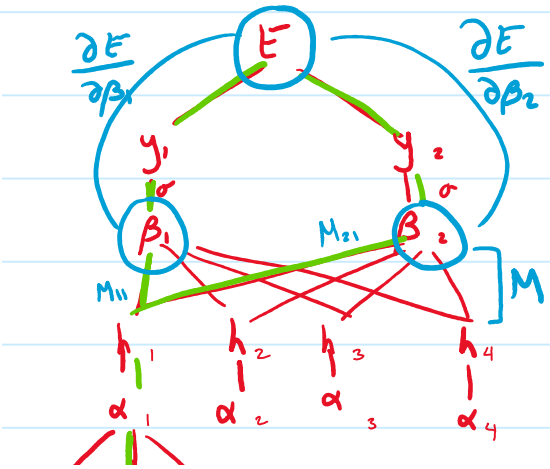
$$\therefore \frac{\partial E}{\partial M_{14}} = h_4$$

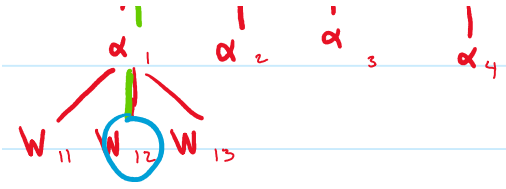
$$\therefore \frac{\partial E}{\partial M_{14}} = \frac{\partial E}{\partial y_1} \frac{d y_1}{d \beta_1} h_4$$

OK, that works for the connection weights between the top two layers. What about the connection weights between layers deeper in the network?

eg. W_{12}







$$\frac{\partial E}{\partial W_{12}} = \frac{\partial \alpha_1}{\partial W_{12}} \frac{\partial E}{\partial \alpha_1}$$

$$\alpha_1 = \sum_{j=1}^{3} W_{1j} x_j + a_j$$

$$\therefore \frac{\partial \alpha_1}{\partial W_{12}} = x_2$$

$$\frac{\partial E}{\partial \alpha_1} = \frac{dh_1}{d\alpha_1} \frac{\partial E}{\partial h_1}$$

$$= \frac{dh_1}{d\alpha_1} \left(\frac{\partial \beta_1}{\partial h_1} \frac{\partial E}{\partial \beta_1} + \frac{\partial \beta_2}{\partial h_1} \frac{\partial E}{\partial \beta_2} \right)$$

$$= \frac{dh_1}{d\alpha_1} \left(M_{11} \frac{\partial E}{\partial \beta_1} + M_{21} \frac{\partial E}{\partial \beta_2} \right)$$

We computed these already when we were learning M.

$$= \frac{dh_1}{d\alpha_1} \left(M_{11}, M_{21} \right) \cdot \left(\frac{\partial E}{\partial \beta_1}, \frac{\partial E}{\partial \beta_2} \right)$$

More generally, $\vec{x} \in \mathbb{R}^X$, $\vec{h} \in \mathbb{R}^H$, $\vec{y}, \vec{t} \in \mathbb{R}^Y$

$$\frac{\partial E}{\partial \alpha_i} = \frac{dh_i}{d\alpha_i} \left[M_{1i} \cdots M_{Yi} \right] \cdot \left[\frac{\partial E}{\partial \beta_1} \cdots \frac{\partial E}{\partial \beta_Y} \right]$$

This is the i^{th} column of M

$$= \frac{dh_i}{d\alpha_i} \left[M_{1i} \cdots M_{Yi} \right] \begin{bmatrix} \frac{\partial E}{\partial \beta_1} \\ \vdots \\ \frac{\partial E}{\partial \beta_Y} \end{bmatrix}$$

For more elements

$$\begin{bmatrix} \frac{\partial E}{\partial \alpha_i} \\ \frac{\partial E}{\partial \alpha_k} \end{bmatrix} = \begin{bmatrix} \frac{dh_i}{\alpha_i} \\ \frac{dh_k}{d\alpha_k} \end{bmatrix} \odot \begin{bmatrix} M_{1i} \cdots M_{Yi} \\ M_{1k} \cdots M_{Yk} \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial \beta_1} \\ \vdots \\ \frac{\partial E}{\partial \beta_Y} \end{bmatrix}$$

Hadamard product

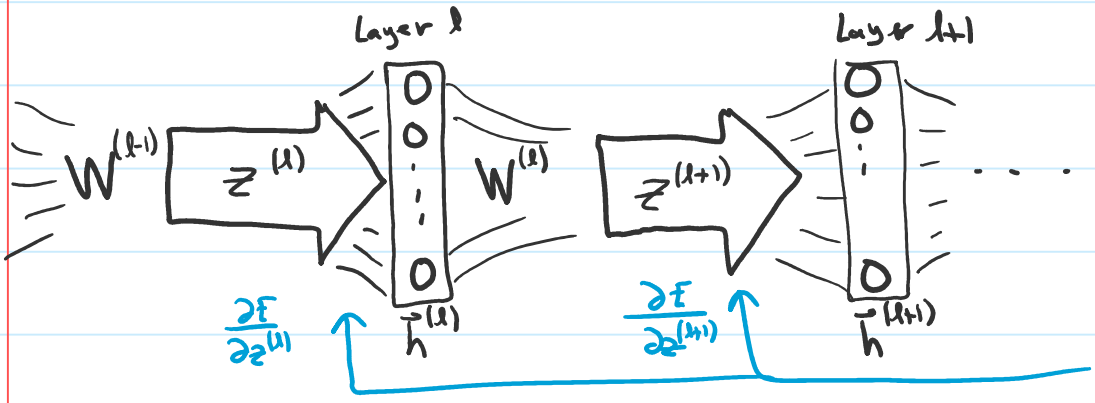
$[a\ b] \odot [c\ d] = [ab\ \ cd]$

For all elements,

$$\begin{bmatrix} \frac{\partial E}{\partial \alpha_1} \\ \vdots \\ \frac{\partial E}{\partial \alpha_H} \end{bmatrix} = \begin{bmatrix} \frac{dh_1}{d\alpha_1} \\ \vdots \\ \frac{dh_H}{d\alpha_H} \end{bmatrix} \odot \begin{bmatrix} M_{11} \cdots M_{Y1} \\ \vdots \\ M_{1H} \cdots M_{YH} \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial \beta_1} \\ \vdots \\ \frac{\partial E}{\partial \beta_Y} \end{bmatrix}$$

$$\frac{\partial E}{\partial \vec{\alpha}} = \frac{d\vec{h}}{d\vec{\alpha}} \odot M^T \frac{\partial E}{\partial \vec{\beta}}$$

The most general, in going down a layer



Suppose we have $\frac{\partial E}{\partial \vec{z}^{(l+1)}} = \nabla_{l+1} E$

Let $\vec{h}^{(l+1)} = \sigma\left(\vec{z}^{(l+1)}\right) = \sigma\left(W^{(l)} \vec{h}^{(l)} + b^{(l+1)}\right)$

$$\frac{\partial E}{\partial \vec{z}^{(l)}} = \frac{d\vec{h}^{(l)}}{d\vec{z}^{(l)}} \odot \left(W^{(l)}\right)^T \frac{\partial E}{\partial \vec{z}^{(l+1)}}$$

Then, to compute $\frac{\partial E}{\partial w_{ij}^{(l)}}$ ...

$$\frac{\partial E}{\partial \left[w^{(l)}\right]} = \frac{\partial z_i^{(l+1)}}{\partial \left[w^{(l)}\right]} \frac{\partial E}{\partial z^{(l+1)}}$$

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial z_i^{(l+1)}}{\partial w_{ij}^{(l)}} \frac{\partial E}{\partial z_i^{(l+1)}}$$

$$= h_j^{(l)} \frac{\partial E}{\partial z_i^{(l+1)}}$$

$$\frac{\partial E}{\partial W^{(l)}}^T = \begin{bmatrix} | \\ \vec{h}^{(l)} \\ | \end{bmatrix} \begin{bmatrix} - & \frac{\partial E}{\partial \vec{z}^{(l+1)}} & - \end{bmatrix} = \boxed{\text{same size as } W^T}$$

outer product

Went over solutions to A1

Training and Testing

Goal: Develop a process to use our labelled data to generate models that can predict future, unseen samples.

We have seen how to adjust a neural network to get it to learn our training data. Of course, the purpose of training a model is so that we can use it on other samples that aren't in our training set. For this reason, we usually break our data into two pieces:

1. Training set: Use most of your labeled data to train your model.
2. Test set: Once your model is trained, use the remaining labeled samples to evaluate your model.

Why? Suppose after some trial-and-error, adjusting the hyperparameters:

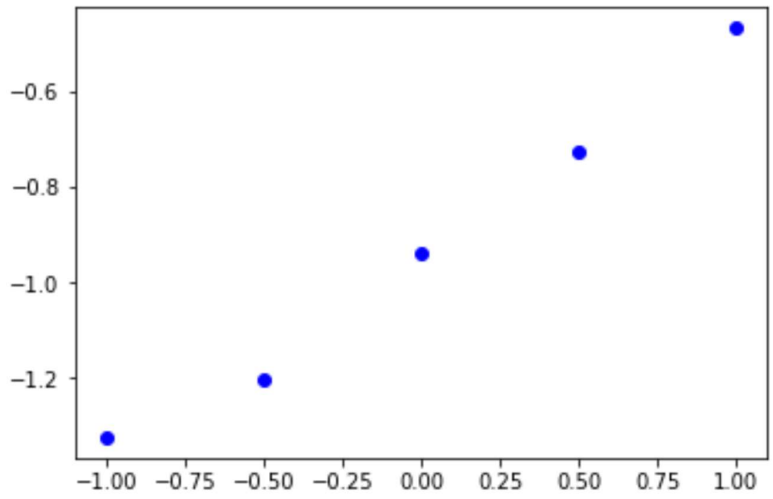
- number of neurons in each layer
- learning rate
- number of epochs
- initial weights

you finally get a low error on the training data. Does that accomplish what you want? Consider an example.

As an experiment, consider noisy samples coming from the ideal mapping,

$$y = 0.4x - 0.9$$

We can only get noisey samples from that mapping.



Our training dataset has 5 samples. And since this is a regression problem, we will use a **linear** activation function on the output, and **MSE** as a loss function.

Training usually entails going through the training data repeatedly, updating the network weights as we go. Each pass through the data is called an epoch.

Let's create a neural network to learn this mapping.

```
net = Network([1, 1000, 1])
```

inputs, hidden, outputs

Before training...

```
Training MSE = 0.955758517256
```

Call the learning function for many epochs.

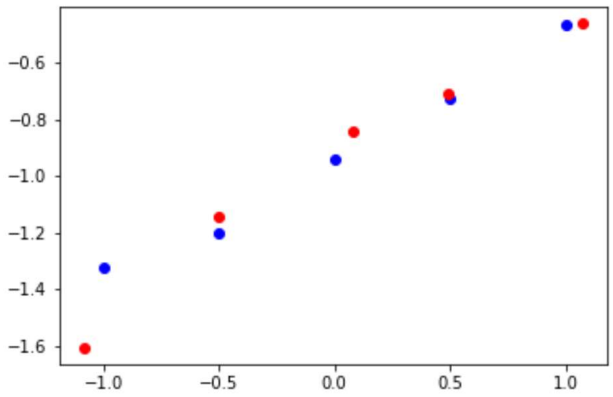
```
net.Learn([training_input, training_output], epochs=40000, lrate=0.01)
```

After going through the dataset 500 times, our average loss is

```
Training MSE = 0.000692658082161
```

SUCCESS!

Let's bask in the glory of our brilliance, and demonstrate how great we are on another sampling of data.



Red ones
Sample of 5

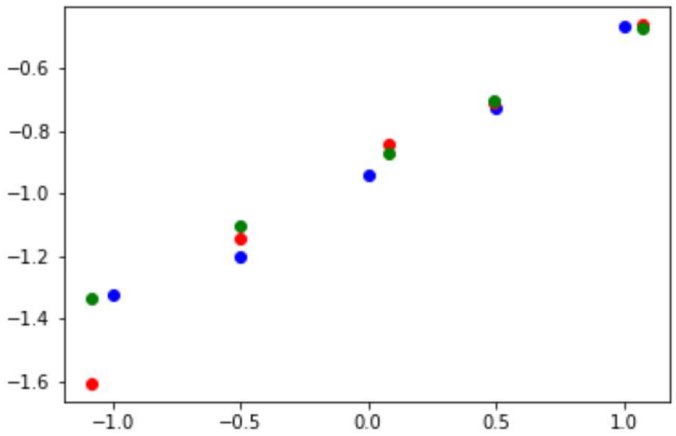
Our average loss on this new set of samples is

```
Test MSE = 0.0156456136501
```

Uh-oh!

This is not as good as our training error.

In fact, what happens when we give it a perfect dataset, without noise?



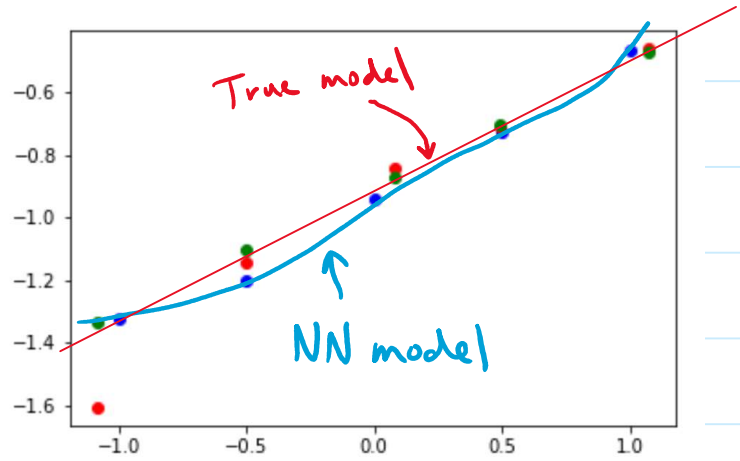

```
Perfect MSE = 0.00161413205859
```

Even perfect data has a higher error than the test data. ???

Recall that our sole purpose was to create a model to predict the output for samples it hasn't seen. How can we

avoid overfitting?

The false sense of success we get from the results on our training dataset is known as overfitting.



The model starts to fit the noise specific to the training set, rather than just to the underlying model.

Validation

If we want to estimate how well our model will generalize to samples it hasn't trained on, we can withhold part of the training set and try our model on that "validation set". Once our model does reasonably well on the validation set, then we have more confidence that it will perform reasonably well on the test set.

It's common to use a random subset of the training set as a validation set.

Overfitting

Goal: See some tricks for how to mitigate against overtraining.

We saw that if a model has enough degrees of freedom, it can become hyper-adapted to the training set, and start to fit the noise in the dataset.

Training error is very small

This is a problem because the model does not generalize well to new samples.

Test error is much bigger than training error

There are some strategies to try to stop our network from trying to fit the noise.

Regularization

Weight Decay

We can limit overfitting by creating a preference for solutions with smaller weights, achieved by adding a term to the loss function that penalizes for the magnitude of the weights.

$$\tilde{E}(\vec{y}, \vec{t}; \theta) = E(\vec{y}, \vec{t}; \theta) - \lambda \|\theta\|_F^2$$

original loss function

$$\|\theta\|_F^2 = \sum_j \theta_j^2$$

"Frobenius Norm"

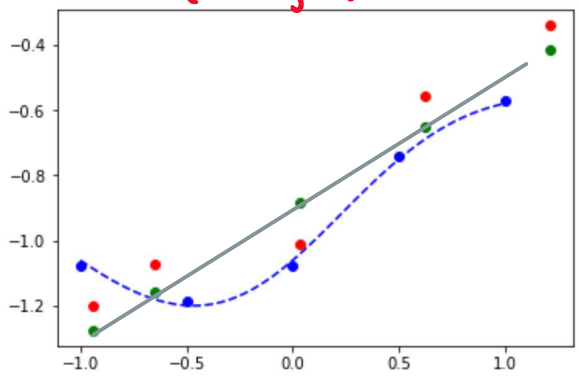
(sum over all weights)

How does this change our gradients, and thus our update rule?

How does this change our gradients, and thus our update rule?

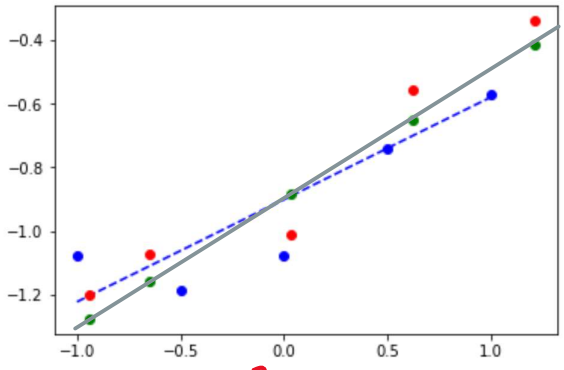
$$\frac{\partial \tilde{E}}{\partial \theta_i} = \frac{\partial E}{\partial \theta_i} - 2\lambda\theta_i$$

$\lambda = 0$ (no regn)



$\|\theta\|_F^2 = 62$

$\lambda = \frac{K}{200}$



$\|\theta\|_F^2 = 4.3$

```
Training MSE = 0.00596061335994
    Test MSE = 0.00574788521961
 Perfect MSE = 0.00178339752734
```

λ controls the weight of the regularization term.

One can also use different norms. For example, it is common to use the L1 norm,

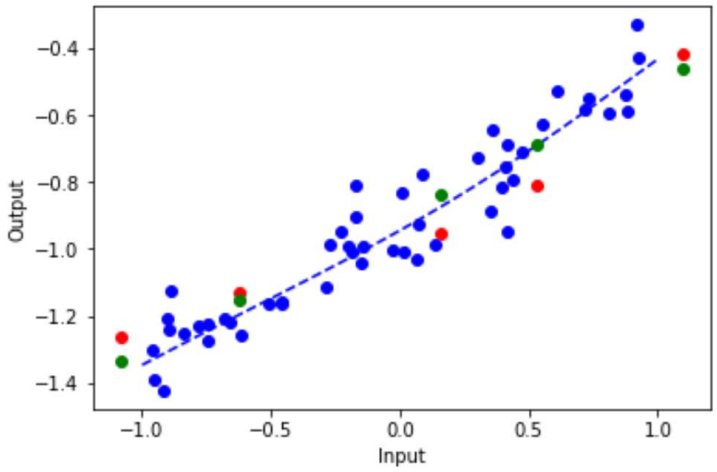
$$L_1(\theta) = \sum_i |\theta|$$

The L1 norm tends to favour scarcity (most weights are close to zero, with only a small number of non-zero weights).

Data Augmentation

Another approach is to include a wider variety of samples in your training set, so that the model is less likely to focus its efforts on

the noise of a few.

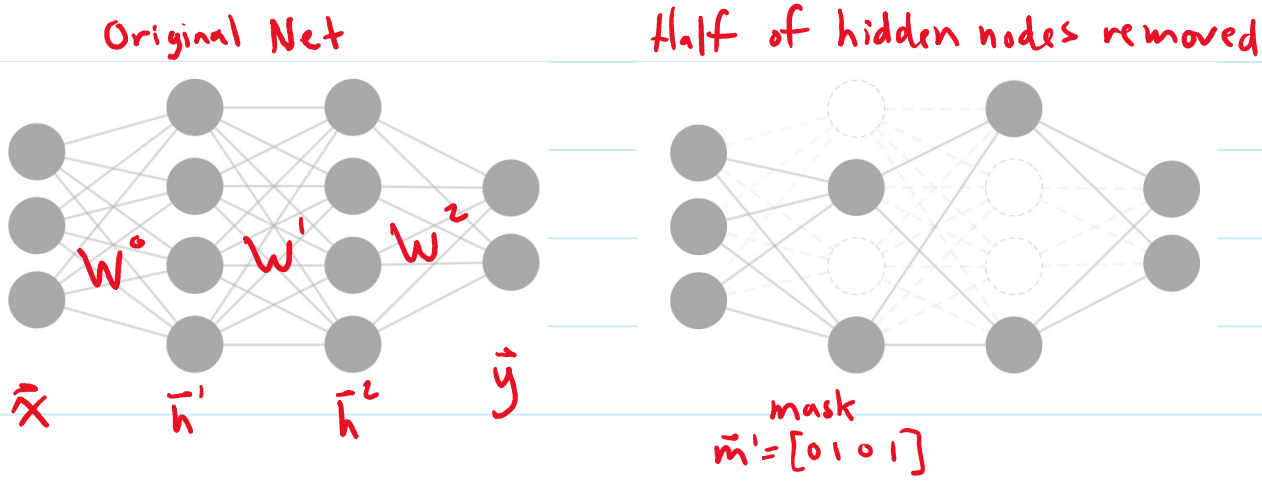



```
Training MSE = 0.00694267279424
    Test MSE = 0.00685380408996
 Perfect MSE = 0.00627912229049
```

Where does this extra data come from?
For image-recognition datasets, one can generate more samples by **shifting** or **rotating** the images. Those transformations presumably do not change the labelling.

Dropout

The last method we will talk about is the most bizarre.
While training using the dropout method, you systematically ignore a large fraction (typically half) of the hidden nodes for each sample. That is, you randomly choose about half of your hidden nodes to be temporarily taken off-line and set to zero.



$\vec{x}$ $\vec{h}^1$ $\vec{h}^2$ y

mask $\vec{m}^1 = [0\ 1\ 0\ 1]$

Do both a feedforward and backprop pass with this diminished network.

$\vec{z}^1 = W^0 \vec{x} + b^1 \Rightarrow \hat{\vec{h}}^1 = \sigma(\vec{z}^1) \in \mathbb{R}^{N^1}$ (hat)

Then apply mask $\vec{m}^1 = \{0,1\}^{N^1}$

$$\vec{h}^1 = \hat{h}^1 \odot \vec{m}^1$$

We need to double the remaining current to $\vec{z}^2$.

$$\vec{z}^2 = (2W^2)\vec{h}^1 + b^2 = W^2(2\vec{h}^1) + b^2$$

Let $\vec{h}^1 = 2\vec{h}^1 \Rightarrow \vec{z}^2 = W^1\vec{h}^1 + b^2$

Likewise, we get $\vec{h}^2$.

During backprop, suppose we have $\frac{\partial E}{\partial \vec{z}^2}$.

Then,

$$\frac{\partial E}{\partial W^1} = \frac{\partial \vec{z}^2}{\partial W^1}\frac{\partial E}{\partial \vec{z}^2} = \vec{h}^1 \frac{\partial E}{\partial \vec{z}^2} = 2\vec{h}^1 \frac{\partial E}{\partial \vec{z}^2}$$

And

$$\frac{\partial E}{\partial \vec{z}^1} = \frac{d\vec{h}^1}{d\vec{z}^1} \odot (W^1)^T \frac{\partial E}{\partial \vec{z}^2}$$

Recall, $\vec{h}^1 = \sigma(\vec{z}^1)$ & $\vec{h}^1 = 2\vec{h}^1 = 2\sigma(\vec{z}^1)$

Thus, $\frac{d\vec{h}^1}{d\vec{z}^1} = 2\frac{d\vec{h}^1}{d\vec{z}^1}$ as long as $\frac{d\vec{h}^1}{d\vec{z}^1} = 0$ if $\vec{h}^1 = 0$

This is true for the logistic function $\frac{d\vec{h}^1}{d\vec{z}^1} = \vec{h}^1(1-\vec{h}^1)$

But we have $\vec{h}^1$, not $\vec{h}^1$.

$$\vec{h}^1(1-\vec{h}^1) = 2\vec{h}^1(1-2\vec{h}^1) \neq 2\left[\vec{h}^1(1-\vec{h}^1)\right].$$

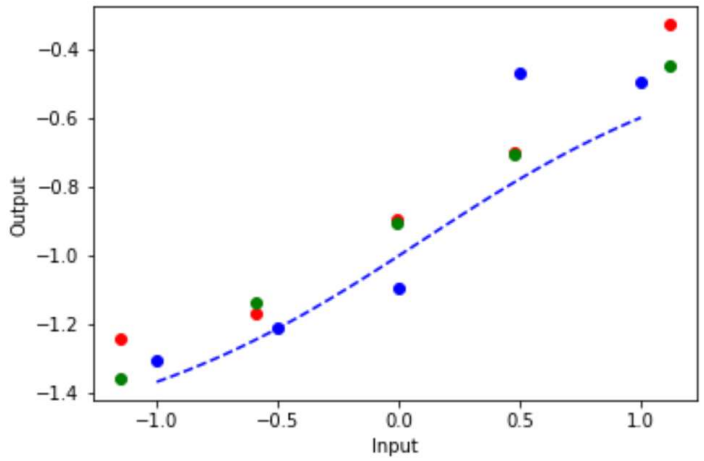
To compute $\sigma'(\vec{z}^1)$, you need to store $\vec{h}^1$ in addition to $\vec{h}^1$.

To compute $\sigma'(\vec{z}')$, you need to store $\vec{h}'$ in addition to $\vec{h}'$.
(or you could store $\vec{z}'$).
So

$$\frac{\partial E}{\partial \vec{z}'} = \frac{d\vec{h}'}{d\vec{z}'} \odot (2W')^T \frac{\partial E}{\partial \vec{z}^2}$$

Important caveat:

You have to **double** the connection weights projecting from a diminished layer in order to give **reasonable** inputs to the next layer.




```
Training MSE = 0.0234801932225
    Test MSE = 0.0212572143563
 Perfect MSE = 0.00874342134987
```

Why does dropout work?

- It's akin to training a bunch of different networks and combining their answers. Each diminished network is like a contributor to this consensus strategy.
- Dropout disallows sensitivity to particular combinations of nodes. Instead, the network has to seek a solution that is robust to loss of nodes.

End of L12

Deep Neural Networks

Goal: To see the advantages and disadvantages deep neural networks: representational power vs. vanishing or exploding gradients.

How many layers should our neural network have?

Universal Approximation Theorem

Let $\varphi(\cdot)$ be a nonconstant, bounded and monotonically-increasing continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, integer N, real constants $v_i, b_i \in \mathbb{R}$, and real vectors $w_i \in \mathbb{R}^m$, $i = 1, \ldots, N$ such that we may define

$$F(x) = \sum_{i=1}^{N} v_i \varphi(w_i^T x + b_i)$$

as an approximation of f where f is independent of φ.

That is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$.

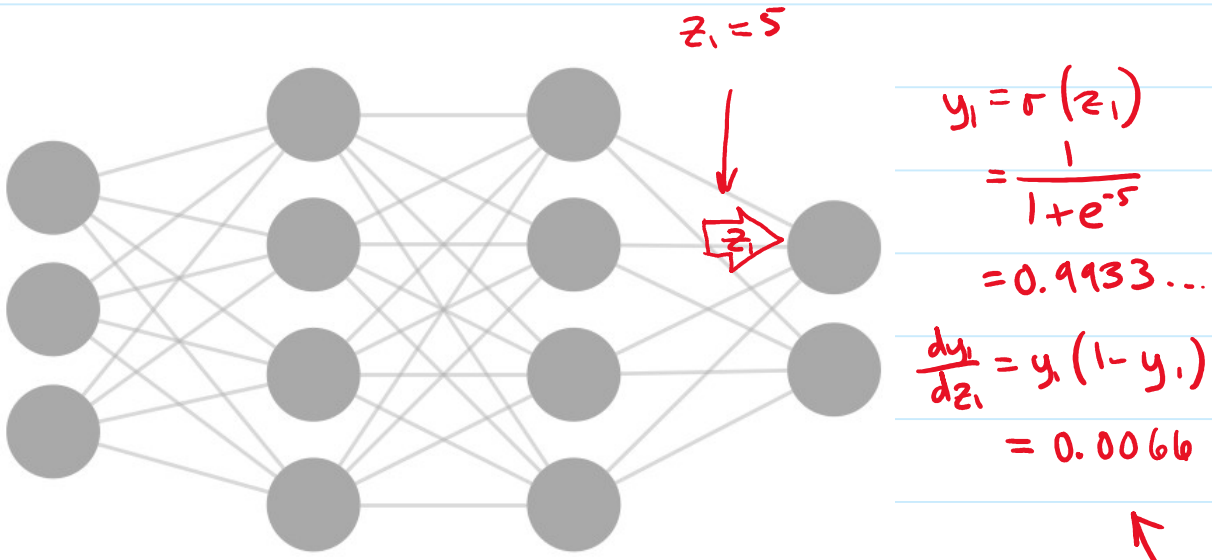
Thus, we really only ever need one hidden layer. But is that the best approach, either in number of nodes, or learning efficiency? No, it can be shown that such a shallow network would require an exponentially large number of nodes (ie. A really big N) to work.

So, a deeper network is preferred in many cases.
(see Python example)

So, why don't we always use really deep networks?

Vanishing Gradients

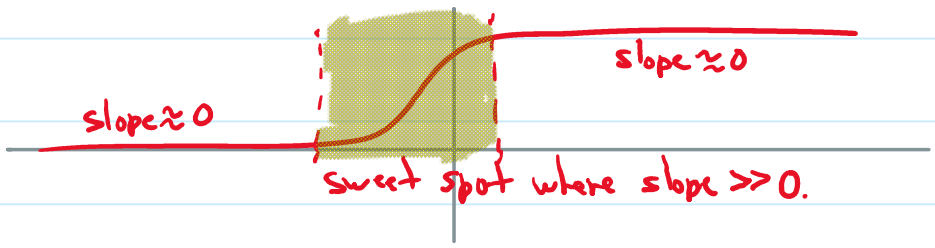
Suppose the initial weights and biases were large enough that the input current to many of the nodes was not too close to zero.
As an example, consider one of the output nodes.



Compare that to if the input current was 0.1.

$y_1 = \sigma(0.1) = 0.525$

$\frac{dy_1}{dz_1} = y_1(1-y_1) = 0.249$ Almost 40x larger than



Hence, the updates to the weights will be smaller when the input currents are large in magnitude.

What about the next layer down?

Suppose $\frac{\partial E}{\partial \vec{z}^{(4)}} \approx 0.01$

What if the inputs to the penultimate layer were around 4 in magnitude?
Then the corresponding slopes of their sigmoid functions will also be small.

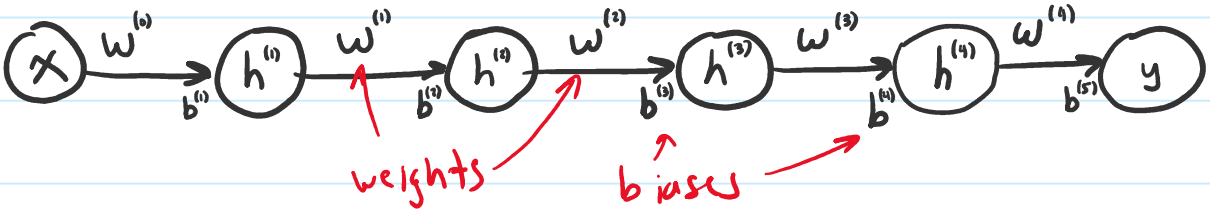
$$\sigma(4) = 0.982$$
$$\sigma'(4) = 0.0177$$

Recall that

$$\frac{\partial E}{\partial \vec{z}^{(3)}} = \frac{d\vec{h}^{(3)}}{d\vec{z}^{(3)}} \odot \left(W^{(3)}\right)^T \frac{\partial E}{\partial \vec{z}^{(4)}}$$
$$= (\text{approx. } 0.0177) \odot \left(W^{(3)}\right)^T (\text{approx. } 0.01)$$
$$= (\text{approx. } 0.000177) \left(W^{(3)}\right)^T$$

And it gets smaller and smaller as you go deeper.
When this happens, learning comes to a halt, especially in the deep layers. This is often called the
vanishing gradients problem.

Another way to look at it. Consider this simple, but deep network.



Start with the loss on the output side: $E(y,t)$
The gradient w.r.t. the input current of the output node is

The gradient w.r.t. the input current of the output node is

$$\frac{\partial E}{\partial z^{(5)}} = y - t$$

Then, using backprop, we can compute a single formula for

$$\frac{\partial E}{\partial z^{(4)}} = (y-t)\, w^{(4)} \sigma'(z^{(4)})$$

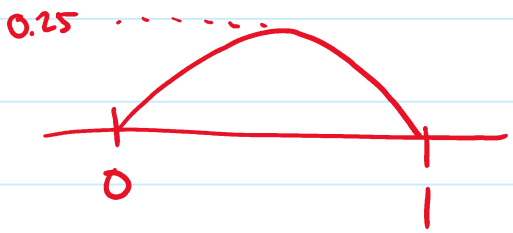
Going deeper...

$$\frac{\partial E}{\partial z^{(1)}} = (y-t) w^{(4)} \sigma'(z^{(4)}) w^{(3)} \sigma'(z^{(3)}) w^{(2)} \sigma'(z^{(2)}) w^{(1)} \sigma'(z^{(1)})$$

What is the steepest slope that $\sigma(z)$ attains?

$$\sigma(z) = \sigma(z)(1 - \sigma(z))$$

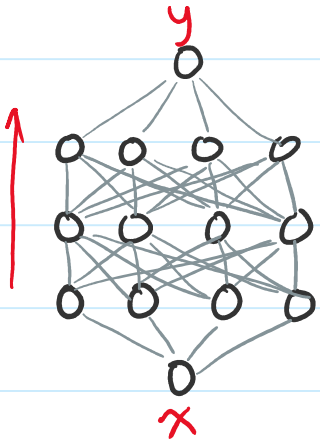
for $0 \le \sigma(z) \le 1$



All else being equal, the gradient goes down by a factor of at least 4 each layer.

We can see this if we look at the norm of the gradients at each layer.

I.e. $\left\| \frac{\partial E}{\partial z^{(i)}} \right\|^2 = \sum_j \left(\frac{\partial E}{\partial z_j^{(i)}} \right)^2$



5 layers
layer 4 norm=0.7598 [0.76]

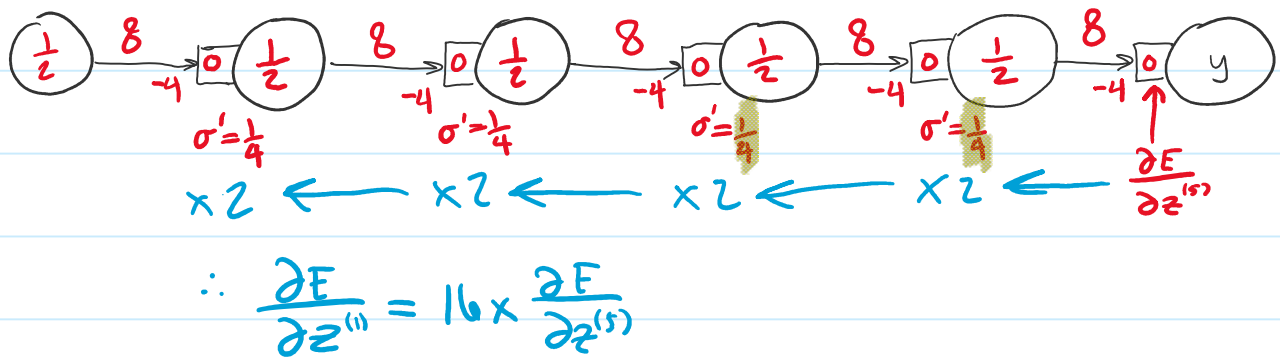
layer 3 norm=0.2266 [-0.038 -0.217 -0.039 -0.039]

layer 2 norm=0.0316 [0.018 -0.018 -0.007 0.018]

layer 1 norm=0.0094 [-0.004 0.002 0.001 0.008]

Exploding Gradients

A similar, though less frequent phenomenon can result in very large gradients.



$$\therefore \frac{\partial E}{\partial z^{(1)}} = 16 \times \frac{\partial E}{\partial z^{(5)}}$$

This situation is more rare since it only occurs when the weights are high and the biases compensate so that the input current lands in the sweet spot of the logistic curve.

Enhancing Optimization

Goal: To learn some methods that help learning go faster.

Suppose out training set is

$$\{(\vec{x}_1, \vec{t}_1) \dots (\vec{x}_P, \vec{t}_P)\} \quad \text{where } \vec{x} \in \mathbb{R}^X, \vec{t} \in \mathbb{R}^Y$$

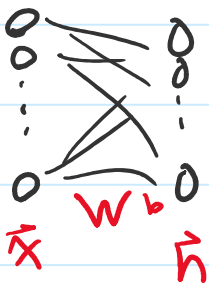
Notice that we can put all of our inputs into a single matrix

$$X = [\vec{x}_1 | \cdots | \vec{x}_P]$$

As well as our targets

$$T = [\vec{t}_1 | \cdots | \vec{t}_P]$$

Does this help us? Yes!
Consider the 1st hidden layer...



$$\vec{z}^{(1)} = W\vec{x} + \vec{b}$$
$$\text{Then } \vec{h}^{(1)} = \sigma(\vec{z}^{(1)})$$

But we can process all P inputs at once!

$$Z^{(1)} = WX + \vec{b}\underbrace{[1 \cdots 1]}_{P} \quad \cdots \quad \underbrace{[1 \cdots 1]}_{\vec{b}} = [\vec{b} | \cdots | \vec{b}]$$

Then, $H^{(1)} = \sigma(Z^{(1)})$.
At the top layer, we get Y.

$$E(Y, T) = \sum_{p=1}^{P} E(\vec{y}_p, \vec{t}_p)$$

$p=1$

Now, working our way back down,

$$\frac{\partial E}{\partial Z^{(l+1)}} = (Y - T) \qquad (Y \times P) \qquad Y \; \square \; P$$

And going down one layer

$$\underset{H \times P}{\frac{\partial E}{\partial Z^{(l)}}} = \underset{H \times P}{\frac{dH}{dZ^{(l)}}} \odot \underset{H \times Y}{\left(W^{(l)}\right)^T} \underset{Y \times P}{\frac{\partial E}{\partial Z^{(l+1)}}}$$

Then,

$$\underset{H \times P}{\frac{\partial E}{\partial W^{(l)}}^T} = \underset{H \times P}{H^{(l)}} \underset{P \times Y}{\left(\frac{\partial E}{\partial Z^{(l+1)}}\right)^T}$$

$$= \begin{bmatrix} 1 \\ \vec{h}_1^{(l)} \\ 1 \end{bmatrix} \left[- \frac{\partial E}{\partial \vec{z}_1^{(l+1)}} - \right] + \cdots + \begin{bmatrix} 1 \\ \vec{h}_P^{(l)} \\ 1 \end{bmatrix} \left[- \frac{\partial E}{\partial \vec{z}_P^{(l+1)}} - \right]$$

Hence, we can use the same formulas to process a whole batch of samples.

One problem with processing the entire dataset for a single update to the weights can be slow. Instead, there is an intermediate approach.

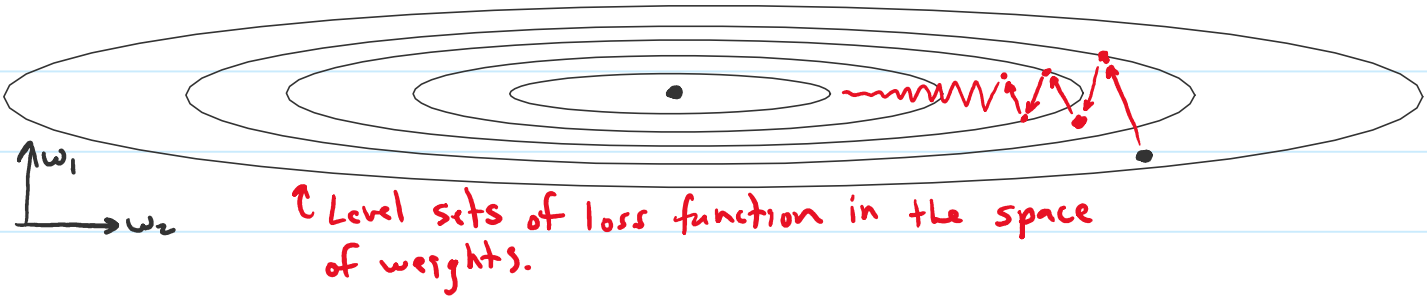
Stochastic Gradient Descent

Rather than processing the entire dataset, or just one sample, we can process chunks of our samples, randomly chosen, to determine our weight updates. We call these chunks "mini-batches". This approach is more stable than single-sample updates, but faster than full-dataset

updates.

Momentum

Consider gradient descent optimization in this situation...



↑ Level sets of loss function in the space of weights.

The shape of the loss function causes oscillation, and this back-and-forth action can be inefficient.

Instead, we can smooth out our trajectory using momentum.

Recall from physics,

$$\text{Velocity } V = \frac{dD}{dt} \qquad \text{Acceleration } A = \frac{dV}{dt}$$

So, solving numerically using Euler's method...

$$\begin{cases} D_{n+1} = D_n + \Delta t\, V_n & (1) \\ V_{n+1} = (1-r)V_n + \Delta t\, A_n \end{cases}$$

r is resistance from friction

When driving a car, this is what you control. - gas pedal, brakes (even steering)

In our previous optimization method, we used our error gradients like V, and D represented our weights. Then we updated our weights using (1)

$$\text{Distance } D_{n+1} = D_n + \Delta t\, V_n$$

$$\text{Velocity } V_{n+1} = V_n + \frac{\partial E}{\partial w}$$

Distance $D_{n+1} = D_n + \Delta t V_n$

Weights $W_{n+1} = W_n - k \frac{\partial E}{\partial W_n}$

But we can instead treat our error gradients as A, and integrate to get an accumulated weight update, akin to V.
In fact, let's call it V.

For each weight, W_{ij}, we also calculate V_{ij}.
Or, in matrix form, for each $W^{(l)}$, we have $V^{(l)}$.

$$V^{(l)} \leftarrow (1-r)V^{(l)} + \frac{\partial E}{\partial W^{(l)}}$$

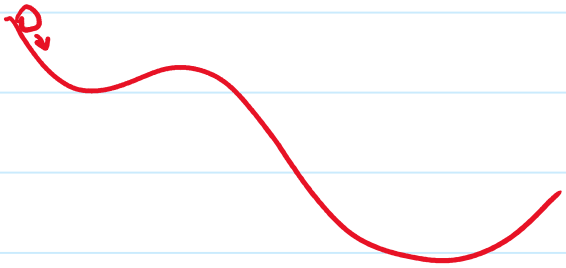
Or, as is commonly used,

$$V^{(l)} \leftarrow \beta V^{(l)} + (1-\beta) \frac{\partial E}{\partial W^{(l)}}$$

Then, update our weights using

$$W^{(l)} \leftarrow W^{(l)} - \kappa V^{(l)}$$

Not only does this smooth out oscillations, but can also help to avoid getting stuck in local minima.



Unit 3:

Unsupervised Learning

Can we get neural networks to extract knowledge from unlabelled data?

In this unit, we will look at some of the standard approaches, including...

- Hopfield networks
- Autoencoders
- Restricted Boltzmann Machines
- Self Organizing Maps
- Variational Autoencoders

Hopfield Networks

Content-Addressable Memory

Fill in the blanks:

intel_ _gent

irreplaceab_ _

1 _ 3 4 _ _ 7 8 9

Because these are patterns you have in memory, you can fill in the missing pieces. In fact, you can also detect errors...

1 2 3 8 5 6 7 2 9

nueroscience

Vaterloo

A *content-addressable memory* (CAM) is a system that can take part of a pattern, and produce the most likely match from memory.

In 1982, John Hopfield published a famous paper

Proc. Natl. Acad. Sci. USA
Vol. 79, pp. 2554–2558, April 1982
Biophysics

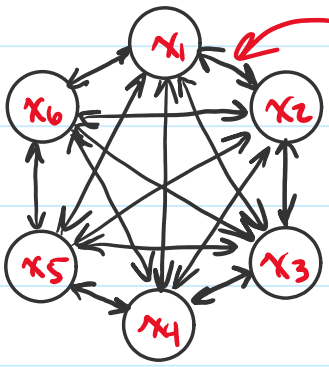
Neural networks and physical systems with emergent collective computational abilities

(associative memory/parallel processing/categorization/content-addressable memory/fail-soft devices)

J. J. HOPFIELD

Division of Chemistry and Biology, California Institute of Technology, Pasadena, California 91125; and Bell Laboratories, Murray Hill, New Jersey 07974

In it, Hopfield proposed a method for using a neural network as a CAM. The network learns the patterns, and converges to the closest pattern when shown a partial pattern.



W_{ij} is the connection weight to node i from node j

or -1

Each node in the network can be a 0 or a 1, or -1

$x_i \in \{-1, 1\} \quad i = 1, \ldots, N$

Or we can think of it as a binary string of length N.
Suppose each node wants to change its state so that

$$x_i = \begin{cases} -1 & \text{if } \sum_{j \neq i} w_{ij} x_j < b_i \\ 1 & \text{if } \sum_{j \neq i} w_{ij} x_j \geq b_i \end{cases}$$

no

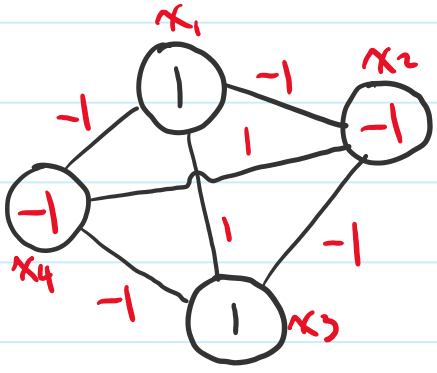
We will assume $b_i = 0$ for now

If we have a pattern that we would like the network to recall, we could set the weights such that:

$w_{ij} > 0$ between any 2 nodes in the same state

$w_{ij} < 0$ between any 2 nodes in different states

For example:



$$W = \begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$$

How can we find the connection matrix W that works for the set of "memories" we want to encode?
Hopfield's answer:

Given n states $\{x^1, x^2, \ldots, x^n\}$

$$w_{ij} = \frac{1}{n} \sum_{s=1}^{n} x_i^s x_j^s \quad , \; i \neq j$$

w_{ij} is the avg. co-activation between nodes i & j

$$w_{ii} = 0 = \frac{1}{n} \sum_{s=1}^{n} x_i^s x_i^s - 1$$

$x^1 = [1 \; 1 \; 1 \; 1]$
$x^2 = [1 \; -1 \; -1 \; 1]$

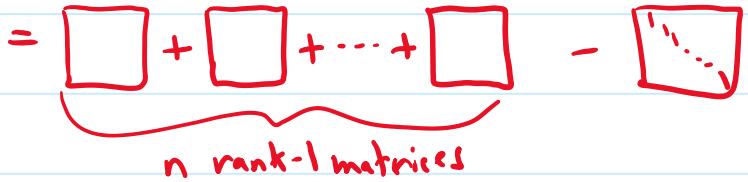
Notice,

Notice,

$x^1 = [1\ -1\ -1\ 1]$

$x^2 = [-1\ 1\ -1\ 1]$

$$W = \frac{1}{n}\sum_{s=1}^{n} x^s (x^s)^T - I$$



Why is this a good choice? Let $x^q \in \{x^1, \ldots, x^n\}$.

$$Wx^q = \frac{1}{n}\sum_{s=1}^{n} x^s (x^s)^T x^q - I x^q$$

Inner product

$$= \frac{1}{n}\sum_{s=1}^{n} x^s \left[(x^s)^T x^q\right] - x^q$$

if $(x^s)^T x^q = 0$ for $s \neq q$.

Note: if $x^s \perp x^q$ for $s \neq q$, then...

$$= \frac{1}{n} x^q \|x^q\|^2 - x^q$$

$$= \frac{N-n}{n} x^q$$

This method works best if the network states, $\{x^1, \ldots, x^n\}$, are all mutually orthogonal.

Examples:

(Python demo)

Hopfield recognized a link between these network states and the Ising model in physics. It has to do with a lattice of interacting magnetic dipoles. Each dipole can be "up" or "down". Which state it's in depends

on its neighbours. Just like the Ising model, you can write the energy of the system using a Hamiltonian function.
For our neural network, assuming W is symmetrical,

$$E = -\frac{1}{2}\sum_{ij} w_{ij} x_i x_j + \sum_i b_i x_i \qquad \text{Hopfield Energy}$$

$$= -\frac{1}{2} x^T W x + b^T x$$

Notice what gradient descent yields.

$$\frac{\partial E}{\partial x_i} = -\sum_{j \neq i} w_{ij} x_j + b_i$$

$$x_i \leftarrow x_i + \Sigma w_{ij} x_j - b_i$$

$$\therefore \frac{dx_i}{dt} = \sum_{i \neq j} w_{ij} x_j - b_i$$

If $\Sigma w_{ij} x_j - b_i > 0 \rightarrow \Sigma w_{ij} x_j > b_i$

→ Thus increase x_i

If $\Sigma w_{ij} x_j - b_i < 0$ . . .

→ Thus decrease x_i

$$\frac{\partial E}{\partial w_{ij}} = -x_i x_j \qquad i \neq j$$

$$\therefore \frac{dw_{ij}}{dt} = x_i x_j \qquad i \neq j$$

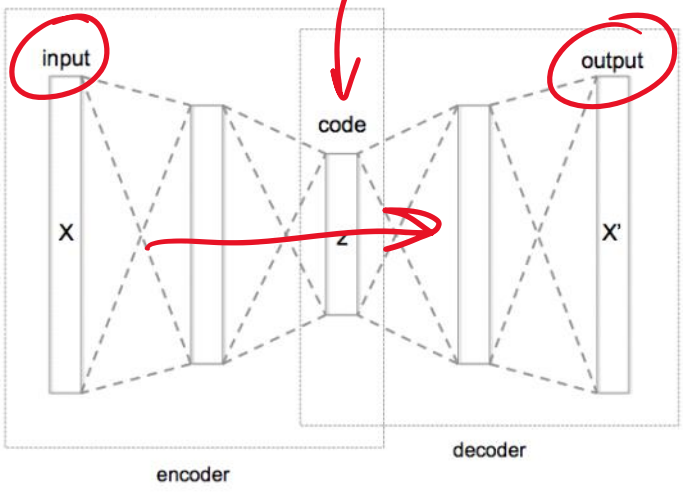
"Hebbian" update rule

END

Autoencoders

An autoencoder is a neural network that learns to encode (and decode) a set of inputs.

Notice that the encoding layer is smaller than input/output layers



(credit Chervinskii)

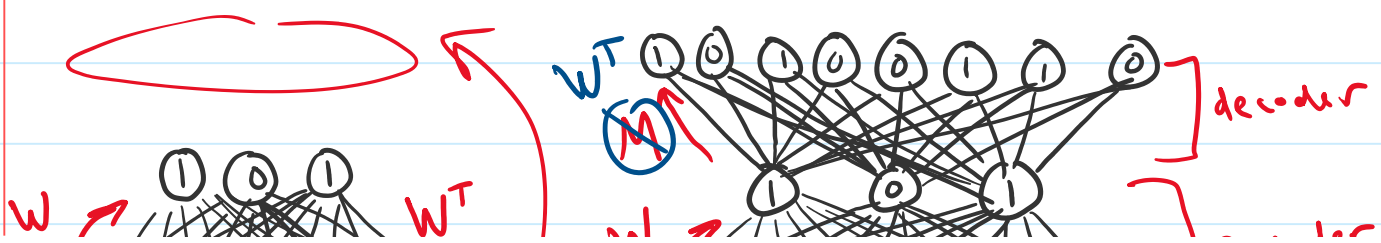
They can be used to find efficient codes for high-dimensional data. For example, suppose I have the following dataset:

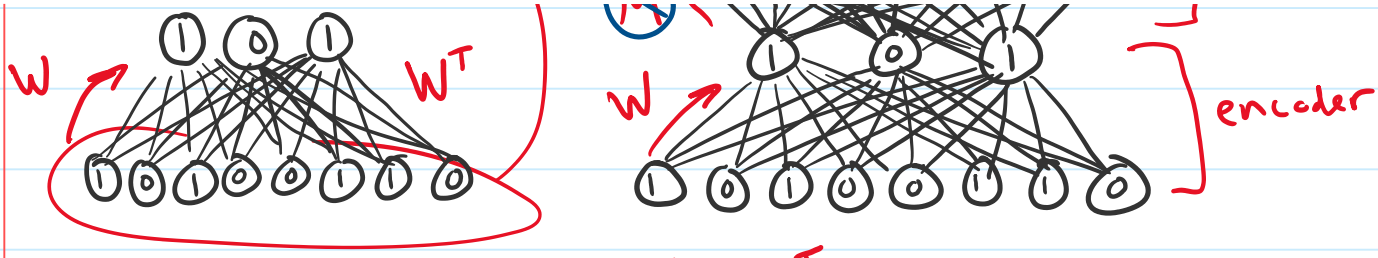
[1 0 1 0 0 1 1 0]
[0 1 0 1 0 1 0 1]
[0 1 1 0 1 0 0 1]
[1 0 0 0 1 0 1 1]
[1 0 0 1 0 1 0 1]

Even though the vectors are 8-D (so could take on 256 different inputs), the actual dataset has only 5 patterns. We can -- in principle -- encode each of them with a unique 3-bit code. But we can choose the dimension of the encoding layer.

Even though our autoencoder ~~layer~~ net is just 2 layers, we "unfold" it (or "unroll" it) to 3 layers, where the input layer and output layer are the same size, and have the same state.

Instead of... ... we use...





We may, or may not, insist that $M = W^T$

If we allow W and M to be different, then it's just a 3-layer network.

How do we enforce that they are the same?

Backprop will give us both $\frac{\partial E}{\partial W}$ & $\frac{\partial E}{\partial M}$

We simply set W and M to be the same to start out, and then update each using

$$\frac{\widetilde{\partial E}}{\partial W} = \frac{\widetilde{\partial E}}{\partial M^T} = \frac{1}{2}\left(\frac{\partial E}{\partial W} + \frac{\partial E}{\partial M^T}\right)$$

This is called "tied weights"

(demo AutoEnc_3L_noise_batch)

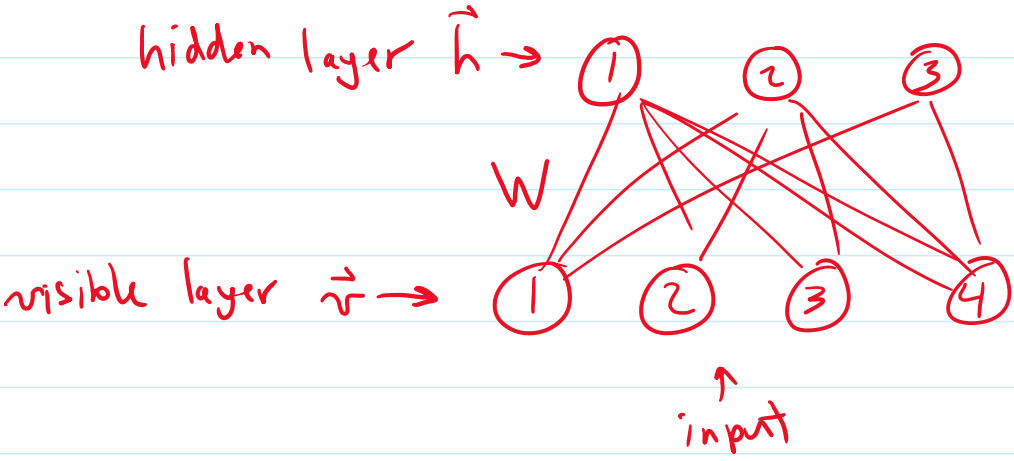
After training, we get the following 3-bit code.

[1 0 1 0 0 1 1 0]	->	[0.00 1.00 1.00]
[0 1 0 1 0 1 0 1]	->	[1.00 0.00 0.00]
[0 1 1 0 1 0 0 1]	->	[0.00 0.00 0.00]
[1 0 0 0 1 0 1 1]	->	[0.00 1.00 0.00]
[1 0 0 1 0 1 0 1]	->	[1.00 1.00 0.00]

Suppose you encounter the input [1 0 1 1 0 1 1 0].
Can you figure out which 3-bit encoding it should have?
Can our learned network handle such cases?

Restricted Boltzmann Machine (RBM)

Similar to a Hopfield network, but it is split into two layers of nodes. The two layers are fully connected, forming a bipartite graph.



Each node is binary, so is either "on" (1) or "off" (0). The probability that a node is on depends on the states of the nodes feeding it, and the connection weights.

$\vec{v} = [v_1 \; v_2 \; \dots \;]$ the visible state

$\vec{h} = [h_1 \; h_2 \; \dots \;]$ the hidden state

Together, they represent the network state.

eg. $\vec{v} = [1 \; 0 \; 0 \; 1]$, $\vec{h} = [0 \; 1 \; 1]$ } is a single network state

We define the energy of the network as

$$E(\vec{v}, \vec{h}) = -\sum_i \sum_j w_{ij} v_i h_j + \sum_i b_i v_i + \sum_j c_j h_j$$

$$= -\vec{v} W \vec{h}^T + \vec{v} \vec{b}^T + \vec{h} \vec{c}^T$$

discount when nodes i & j are both on simultaneously

energy incurred by turning a node on

Like many processes in nature, we want to find the minimum energy state. How?

Consider the "energy gap", the difference in energy when we flip node v_k from off to on.

Consider the energy gap, the difference in energy when we flip node v_k from off to on.

$$\Delta E_k = E(v_k \text{ off}) - E(v_k \text{ on})$$
$$= \sum_j w_{kj} h_j - b_k$$

- If $\Delta E_k > 0$, then $E(v_k \text{ off}) > E(v_k \text{ on})$
  → "on" is lower energy, so set $v_k = 1$
- If $\Delta E_k < 0$, then $E(v_k \text{ off}) < E(v_k \text{ on})$
  → "off" is lower energy, so set $v_k = 0$

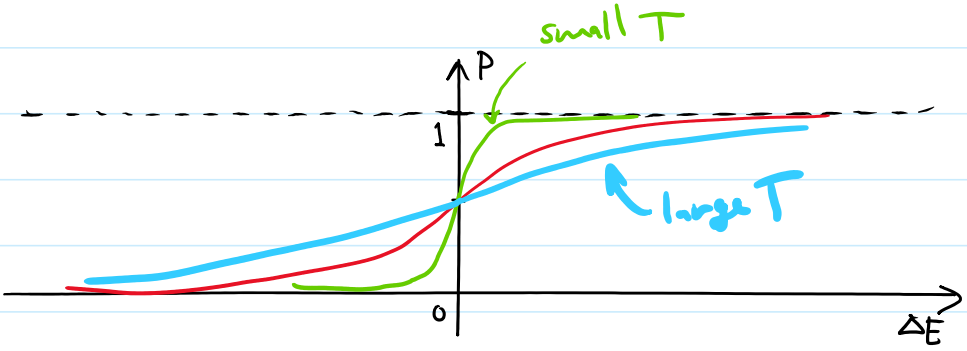
The energy gap of each node depends on the states of other nodes, so finding the min-energy state requires some doing.
One strategy is to visit and update the nodes in random order (like Hopfield).
We can do better since our network is bipartite (hence the "Restricted" in RBM). The visible units only depend on the hidden units, and vice versa. So we can update one whole layer at a time.

However, this is another example of a local optimization method... can get stuck in local min ≠ global min.

To avoid (reduce) getting stuck, we can add varying degrees of randomness

$$P(v_k = 1) = \frac{1}{1 + e^{-\frac{\Delta E_k}{T}}}$$ ← "temperature"

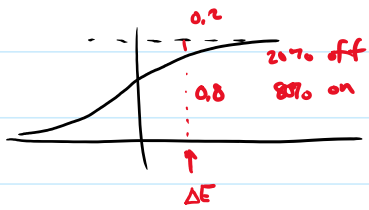
This is known as the Boltzmann distribution



Note: Comes from statistical mechanics. Suppose the particles can be in one of 2 states (0 or 1), and that the energy difference between the states is ΔE. P is the fraction of particles in state 1. As temperature increases, there is more movement back and forth between the states.

is ΔE. P is the fraction of particles in state 1. As temperature increases, there is more movement back and forth between the states.

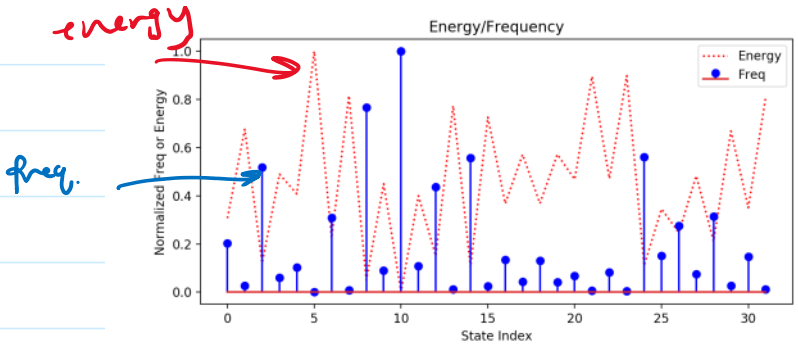
T large: hot, particles are dynamic
T small: cold, particles are frozen



To update node k:

1) compute ΔE_k
2) evaluate $p = P(v_k = 1) \in (0, 1)$
3) choose a uniform random # $r \in [0, 1]$
4) if $p > r$, then set node to "on" $v_k = 1$
   if $p \leq r$, set node "off" $v_k = 0$

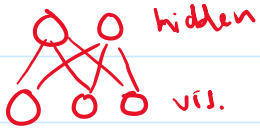
If you let the RBM run freely, it will jump around from state to state, but will occupy lower-energy states for frequently.



2^3 visible $\times$ 2^2 hidden $= 32$ states

Nonetheless, our goal is to change the connection weights (W) and biases (b and c) so that our network states tend to be relevant...
I'll explain what I mean with an example.
Example:

Suppose our net has 3 visible and 2 hidden nodes. Thus, input to the visible nodes takes the form of binary 3-vectors:



[0 0 0] [0 0 1] [0 1 0] [0 1 1] [1 0 0] [1 0 1] [1 1 0] [1 1 1]

Notice the hidden layer does not have enough nodes to encode all these objects.
However, perhaps the "universe" that the network lives

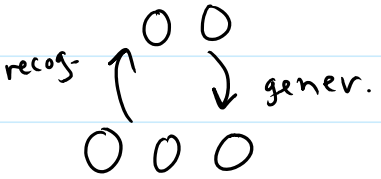
in frequently includes

[1 0 0] [0 1 0] and [0 0 1]

and almost never includes the others.
In this case, the hidden nodes have more than enough encoding capability, as long as the statistically relevant patterns are encoded appropriately. But how?

We want to encourage our network to hold the desired patterns, and discourage it from wandering away.
We call the up pass (from visible to hidden) "recognition".
We call the down pass (hidden to visible) "generative".

Given a visible pattern, we want...



... to generate visible states close to what we started with.

ie. The visible input $\vec{x}$ gets encoded in the hidden nodes, and we want that hidden representation to be able to generate only one visible pattern, $\vec{x}$.

In this way, the hidden nodes offer a new, more efficient representation of the statistically salient patterns in the "universe".

RBM Learning Using Contrastive Divergence

The algorithm is based on a comparison between the original input, and how well it can be reconstructed from the resulting hidden-layer state.

Given an input pattern $\vec{v}$

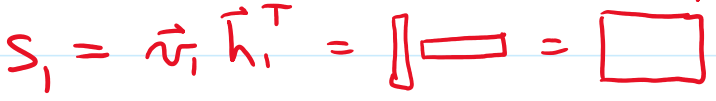
① Recognition Pass 1 $\vec{h}_1$ 0 0

$\vec{v}_1$ 0 0 0

$\Delta\vec{E} = \vec{v}_1 W - \vec{c}$ (row vector)

$P(\vec{h}_1 = 1) = \frac{1}{1+e^{-\frac{\Delta E}{T}}}$

② Get co-occurrence statistics

$S_1 = \vec{v}_1 \vec{h}_1^T$ =  same size as W

③ Generative Pass $\vec{h}_1$ 0 0 $\Delta\vec{E} = W\vec{h}_1 - \vec{b}$ (3-vec.)

③ Generative Pass $\vec{h}_1$ 0 0 $\Delta\vec{E} = W\vec{h}_1 - \vec{b}$ (3-vec.)

$\vec{v}_2$ 0 0 0 $P(\vec{v}_2 = 1) = \frac{1}{1 + e^{-\frac{\Delta E}{T}}}$

④ Recognition Pass 2 $\vec{h}_2$ 0 0 $\Delta E = \cdots$

$\vec{v}_2$ 0 0 0 $P(\vec{h}_2 = 1) = \cdots$

⑤ Get co-activation statistics again

$S_2 = \vec{v}_2 \vec{h}_2^T =$ [ ]

⑥ Update weights using

$$W_{new} = W_{old} + K(S_1 - S_2)$$

Similarly for biases

$$b_{new} = b_{old} - \gamma(\vec{v}_1 - \vec{v}_2)$$ note minus sign

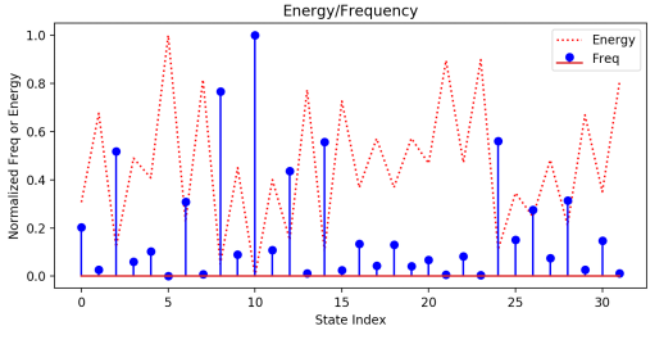
$$c_{new} = c_{old} - \gamma(\vec{h}_1 - \vec{h}_2)$$

Typically, one would process many (~100) input patterns and collect the statistics for all of them, and use that for the weight and bias updates.

Example:

Universe consists of [1 0 0] [0 1 0] [0 0 1]

Starting with random weights,



Training algorithm:

for each temperature $T = 20, 10, 5, 2, 1$

for each temperature $T = 20, 10, 5, 2, 1$

for each of 400 epochs

for each sample in a batch

choose visible pattern

add a little noise

project up → V_1, H_1, S_1

project down & back up → V_2, H_2, S_2

increment statistics

next sample

update weights & biases

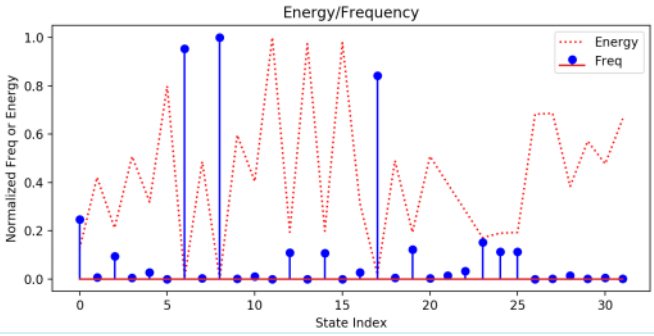
next epoch

next temp.

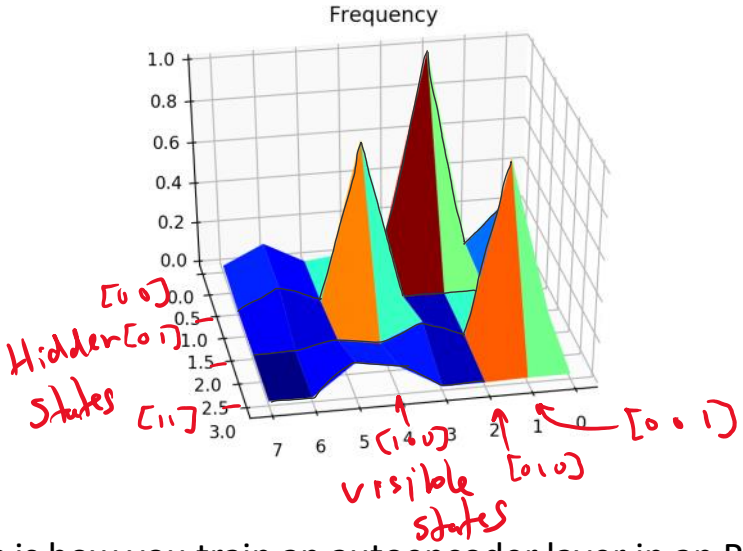
This could all be done using matrix formulas

Example continued:

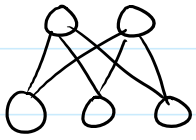
After all that, the energy profile (and state distribution) look different, with only a small number of states dominating the behaviour.



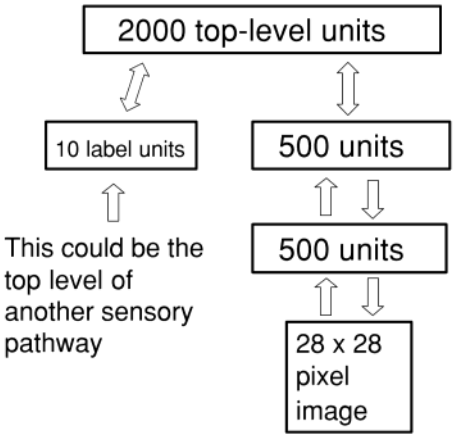
Let's look at all the visitation frequencies as a 2D surface.



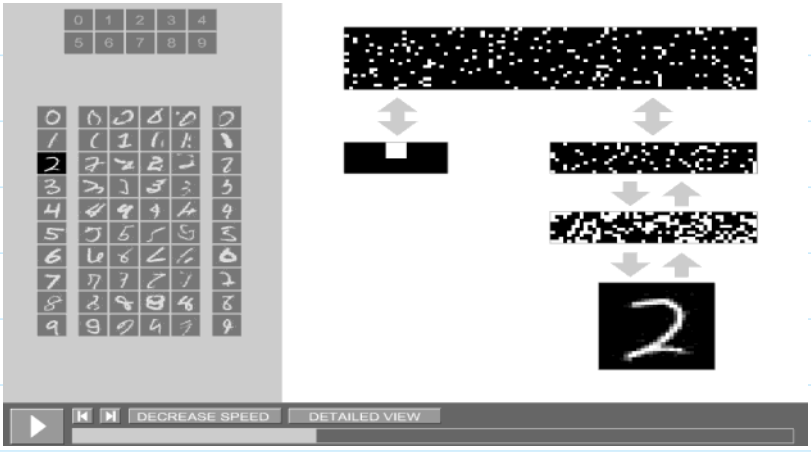
This is how you train an autoencoder layer in an RBM.
After you've trained one layer, you can then train another layer on top of that.
This is called a



Finally, a classification layer can be put on the top, and trained using backpropagation.



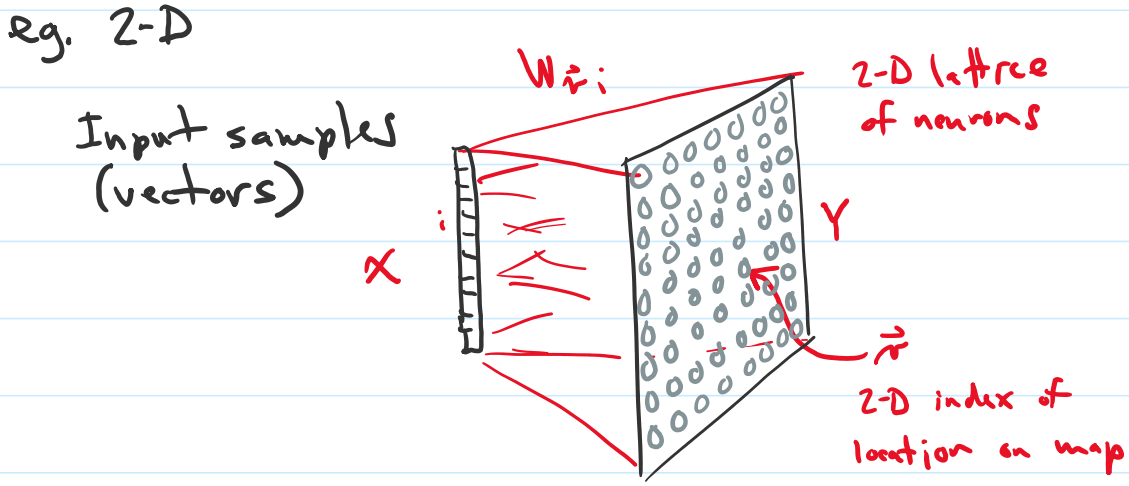
From:
Hinton, G. E., Osindero, S., Teh, Y.-W. "A Fast Learning Algorithm for Deep Belief Nets", *Neural Computation*, 18, 1527–1554, 2006.



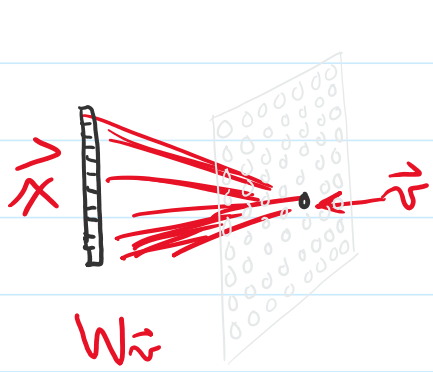
http://www.cs.toronto.edu/~hinton/adi/index.htm

Self Organizing Maps (SOM)

This is another method for unsupervised learning. The term "map" comes from the fact that the underlying neural network has its neurons arranged in a topological space. Usually, that space is 2-dimensional (but could be any dimension).



Each input node projects to each neuron in the lattice. Hence, you can picture the set of connections arriving at a neuron as a vector in the input space, similar to how we worked with perceptrons. And, like perceptrons, the weight increments involve the input samples.



eg. 11 inputs
→ 11 weights

$$W_{\vec{r}}^{t+1} = W_{\vec{r}}^{t} + \boxed{?}\left(\vec{x} - W_{\vec{r}}^{t}\right)$$

↑ nonlinearities

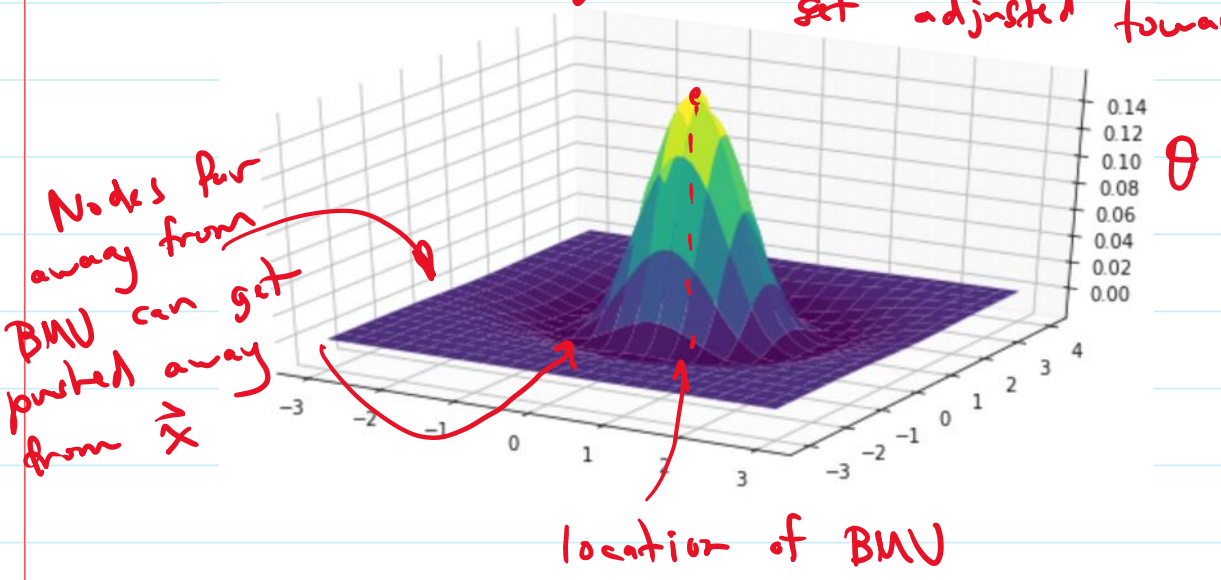
However, the neurons compete against each other to encode the input. The neuron that shows the largest response is called the

Best Matching Unit (BMU)

Its weights are influenced the most. The weights of other neurons are influenced according their distance from the BMU

Its weights are influenced the most. The weights of other neurons are influenced according their distance from the BMU
This spatial function is called the neighbourhood function.

Weights of nearby nodes also get adjusted toward $\vec{x}$.



$$\theta(\vec{v}) = N(\vec{v}; \mu = \vec{u}_{BMU}, \sigma_1) - N(\vec{v}; \mu = \vec{u}_{BMU}, \sigma_2)$$ Difference of Gaussians (DoG)

$$W_{\vec{v}} = W_{\vec{v}} + \theta(\vec{v}; \vec{u}_{BMU})(\vec{x} - W_{\vec{v}})$$

This is a common theme in neuroscience, called

Local excitation, lateral inhibition

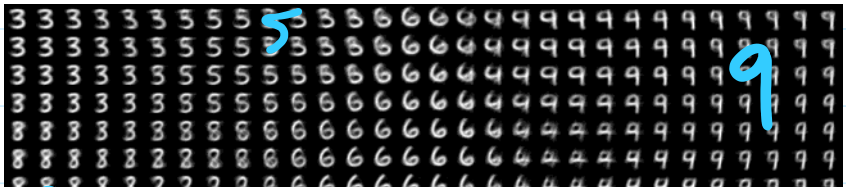
Over time, this encourages neurons in the same area to have similar sets of weights.

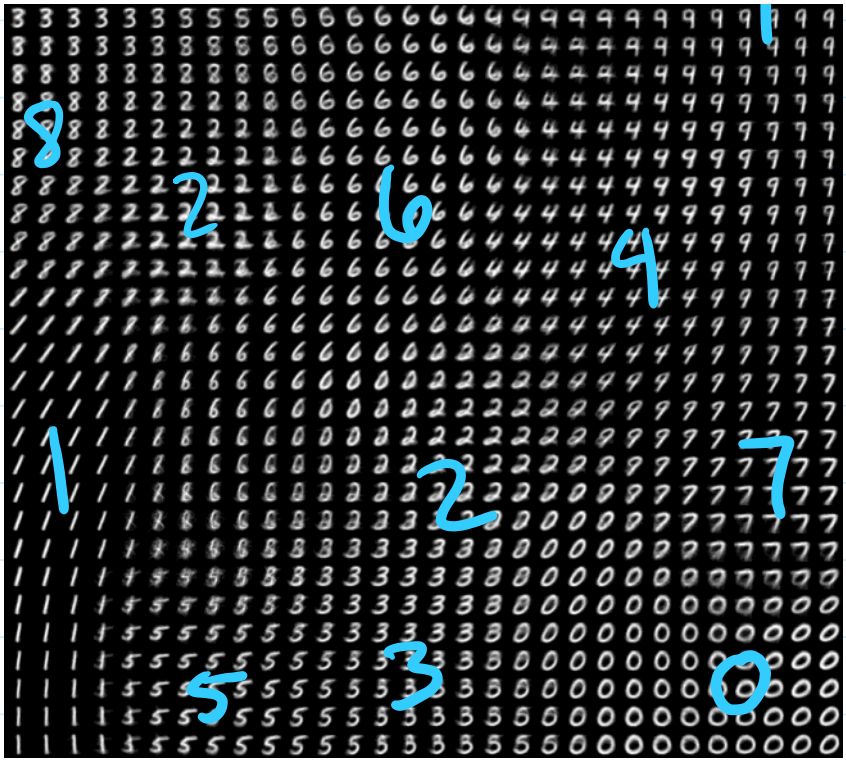
Finally, it's also common to add a time-dependent decay

$$W_{\vec{v}} = W_{\vec{v}} + \alpha(t)\,\theta(\vec{v}; \vec{u}_{BMU})(\vec{x} - W_{\vec{v}})$$

SOM on MNIST

30×30 lattice





The images show the sets of weights for each neuron.
This unsupervised method automatically clusters similar inputs, and embeds them into a lower-dimensional topological space, so that

Unit 4:

Recurrent Neural Networks

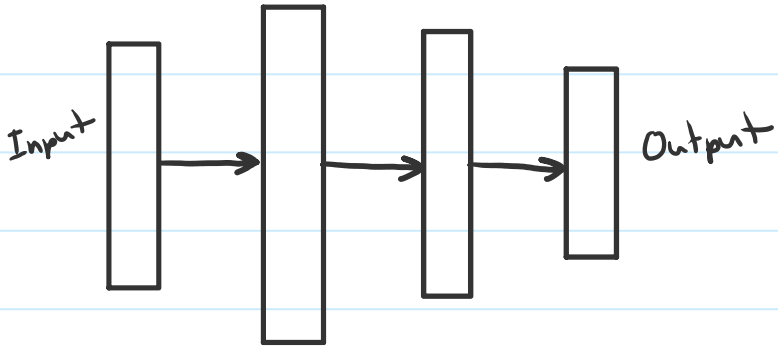
What happens when we allow connections between neurons in a layer?

In this unit, we will look at some of the standard approaches, including...

- Backpropagation Through Time (BPTT)
- Long Short-Term Memory (LSTM)

Recurrent Neural Networks

Thus far, we have mostly focussed on feedforward neural networks.



There are no loops in a feedforward network. But there are reasons that we might want to allow feedback connections that create loops.

Eg. 1) We recently looked at recurrent continuous-time networks to implement dynamics.

Eg. 2) If we want to build a running memory into our network behaviour.

Consider the task of predicting the next word in a sentence.

Emma's cat was sick, so she took her to the _____ .

I'll work it out with pencil and _____ .

She picked up the object, studied it, then put it _____ .

0, 2, 4, 6, _____ .

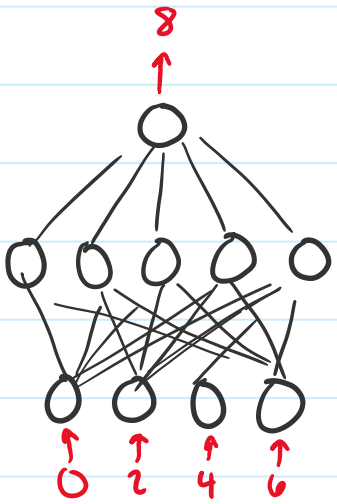
1, 2, 4, 8, _____ .

In each case, the word you predict depends - in very complex ways - on the words that precede it.
Thus, a network will probably have to encode an ordered sequence of words to solve this problem.

Solution 1:
Design the network so that the entire sequence is input all at once.

Design the network so that the entire sequence is input all at once.

eg.



Problems:

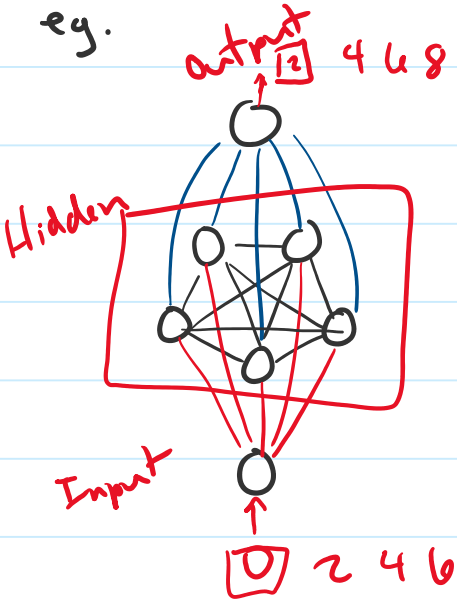
1) The net can only consider fixed-length sequences.
2) No processing can occur until the entire seq is given.

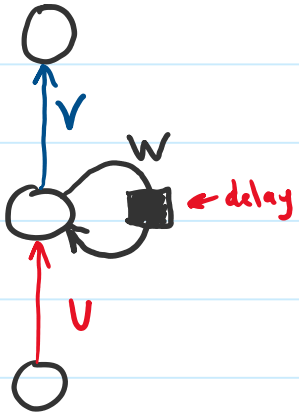
Solution 2:

Allow the state of the network to depend on new input, as well as its previous state.

Let the network be recurrent.

eg.



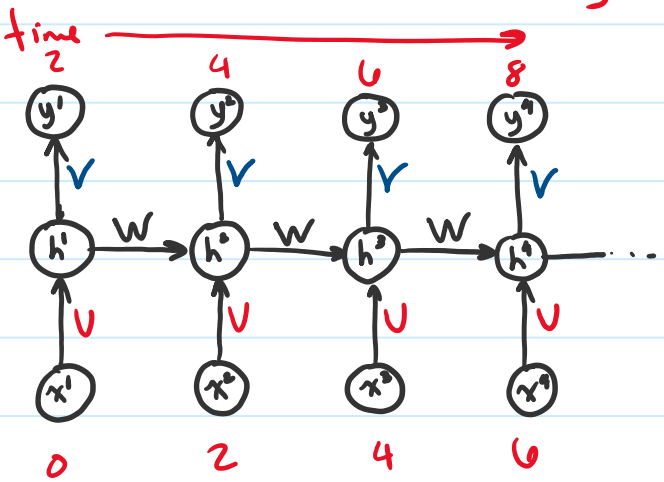


This is a recurrent Neural Network (RNN).

In an RNN, the state of the hidden layer can encode the input sequence, and thus have the information it needs to determine the

proper output.

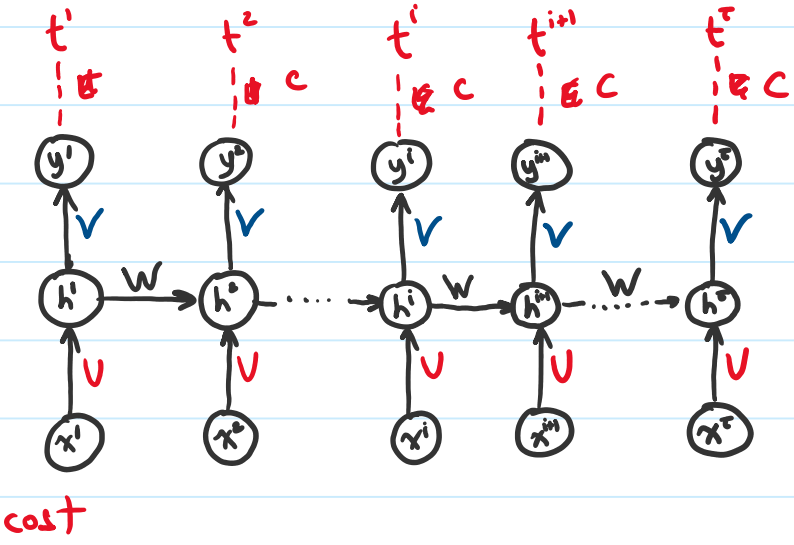
Neat idea, but how can we train such a network?
First, we unroll the network, similar to what we did for autoencoders. But this is unrolling through time.



Note:

$$s^i = Ux^i + Wh^{i-1} + b$$
$$h^i = \sigma(s^i)$$
$$z^i = Vh^i + c$$
$$y^i = \sigma(z^i)$$

Like before, we will have targets, t^i, and a cost function C.



Here, we go for τ steps.

The ~~loss~~ cost, or error, is the sum of all the loss functions,

$$E(y^1, \dots, y^\tau, t^1, \dots, t^\tau) = \sum_{i=1}^{\tau} C(y^i, t^i)\alpha_i$$

And as usual, we aim to minimize the expected cost over our dataset with respect to the connection weights and biases: $\theta = \{U, V, W, b, c\}$

$$\min_\theta \left\langle \sum_{i=1}^{\tau} C(y^i, t^i) \right\rangle_{y,t} = \min_\theta \left\langle E(\dots) \right\rangle_{y,t}$$

Then we can start at the last output, and work our way back through the network $x \in \mathbb{R}^X \quad h \in \mathbb{R}^H \quad u \in \mathbb{R}^Y$

Then we can start at the last output, and work our way back through the network $x \in \mathbb{R}^X \quad h \in \mathbb{R}^H \quad y \in \mathbb{R}^Y$

$$\frac{\partial E}{\partial z^\tau} = \frac{\partial E}{\partial y^\tau} \odot \frac{dy^\tau}{dz^\tau} = \frac{\partial E}{\partial y^\tau} \odot \sigma'(z^\tau) \quad \text{where } y^\tau = \sigma(z^\tau)$$

$\sigma' \in \mathbb{R}^Y$

$$\underset{H}{\frac{\partial E}{\partial h^\tau}} = \underset{H \times Y}{\frac{\partial z^\tau}{\partial h^\tau}} \underset{Y}{\frac{\partial E}{\partial z^\tau}} = V^T \frac{\partial E}{\partial z^\tau}$$

$$z^\tau = \underset{Y \times H}{V} h^\tau + c$$

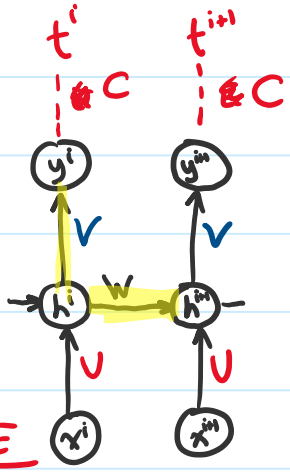
Suppose we have already computed $\frac{\partial E}{\partial h^{i+1}}$

We would like to compute

$$\frac{\partial E}{\partial h^i} = \frac{\partial}{\partial h_i}\left(\sum_{j=1}^{I} C(y^j, t^j)\right)$$

Proof by induction $= \frac{\partial C(y^i, t^i)}{\partial h^i} + \frac{\partial E}{\partial h^{i+1}} \frac{\partial h^{i+1}}{\partial h^i}$

$$= \underset{H \times Y}{V^T} \underset{Y}{\frac{\partial C}{\partial y^i}} \odot \underset{Y}{\sigma'(z^i)} + \underset{H \times H}{W^T} \underset{H}{\sigma'(s^{i+1})} \odot \underset{H}{\frac{\partial E}{\partial h^{i+1}}}$$



$$h^{i+1} = \sigma(\underbrace{Wh^i + Ux^{i+1} + b}_{s^{i+1}})$$

Once you have $\frac{\partial E}{\partial h^i}$, you can compute the gradient of the cost with respect to the weights and biases.

$$\frac{\partial E}{\partial V}, \frac{\partial E}{\partial W}, \frac{\partial E}{\partial U}, \frac{\partial E}{\partial b}, \frac{\partial E}{\partial c}$$

But that's left as an exercise for the reader. :)

Long Short-Term Memory (LSTM)

The benefit of an RNN is that it can accumulate a hidden state that encodes input over time. For example, a properly trained RNN could complete the sentence,

"To ride a bicycle, put your feet on the _____."

However, one difficulty with an RNN is that it can be very difficult to train it to maintain information in its hidden state for a long time. It would have trouble completing this paragraph,

"A bicycle is an efficient mode of transportation.
... (blah blah blah) ...
To get started, pick one up and put your feet on the _____."

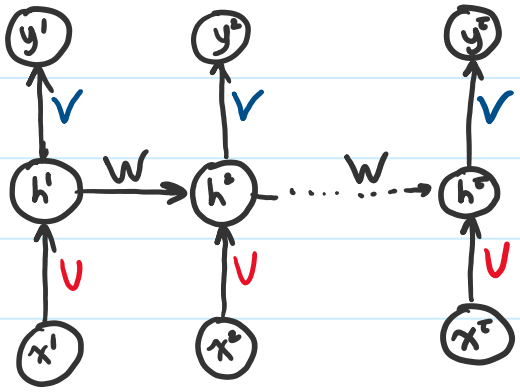
This instability was first pointed out by Bengio et al in 1994.

IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 5, NO. 2, MARCH 1994

Learning Long-Term Dependencies with Gradient Descent is Difficult

Yoshua Bengio, Patrice Simard, and Paolo Frasconi, *Student Member, IEEE*

They show that the longer you need to keep information in memory, the harder it is to train. The BPTT algorithm becomes unstable because of exploding or vanishing gradients. The problem stems from the fact that the gradients get multiplied by the recurrent connection weights for each time step.



$$\frac{\partial E}{\partial h^i} = W^T \sigma'(s^{i+1}) \frac{\partial E}{\partial h^{i+1}} + V^T \sigma'(z^i) \frac{\partial E}{\partial y^i}$$

$$\vdots \quad \frac{\partial E}{\partial h^{i+1}} = W^T \sigma'(s^{i+2}) \frac{\partial E}{\partial h^{i+2}} + V^T \sigma'(z^{i+1}) \frac{\partial E}{\partial y^{i+1}}$$

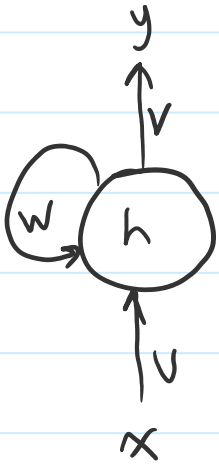
$$\frac{\partial E}{\partial h^i} = W^T \sigma'(s^{i+1}) W^T \sigma'(s^{i+2}) \cdots W^T \sigma'(s^T) \frac{\partial E}{\partial h^T} + \cdots$$

causes vanishing/exploding gradients

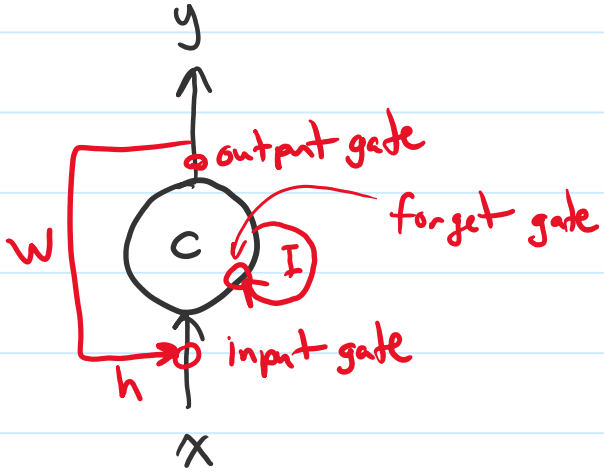
Long Short-TermMemory (LSTM)

To combat this state decay, Hochreiter & Schmidhuber (1997) proposed an additional hidden state that persists from step to step. It does not get multiplied by the connection weights at each time step.

Standard RNN



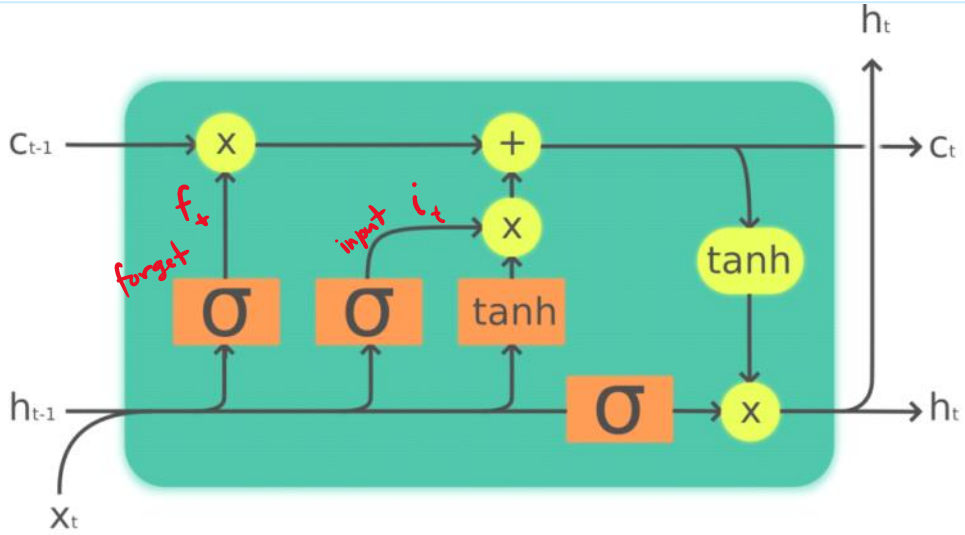
LSTM



There are various ways h and c can interact.

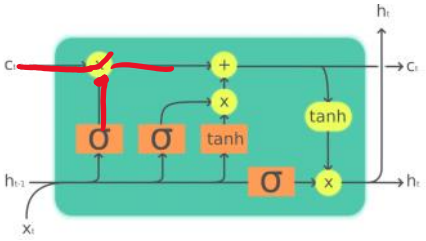
- h and x can increment c (input)
- h and x can erase c (forget)
- h and x control output of c (output)

Putting it together...



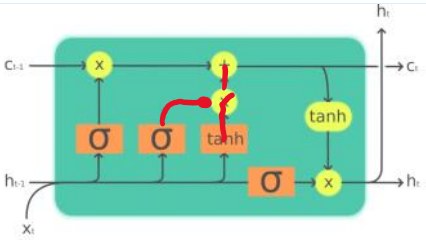
Forget Gate

$$f_t = \sigma\left(W_f\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix} + b_f\right)$$



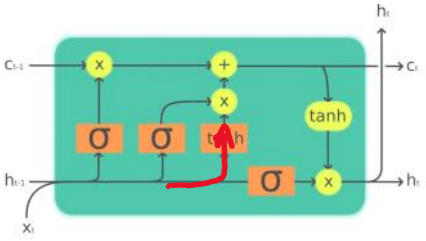
Input Gate

$$i_t = \sigma\left(W_i\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix} + b_i\right)$$



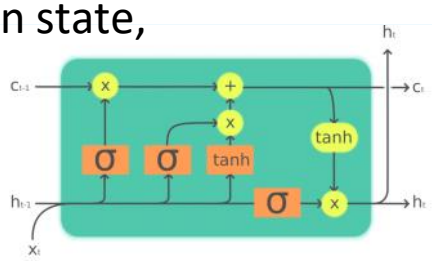
What is the input?

$$\tilde{c}_t = \tanh\left(W_c\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix} + b_c\right)$$



Combining the input with the non-forgotten state,

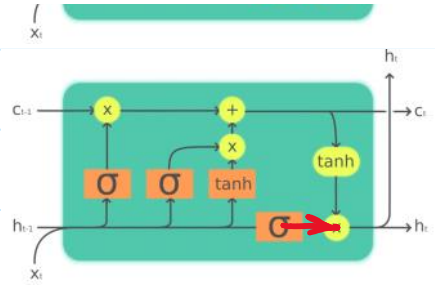
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$



Output Gate

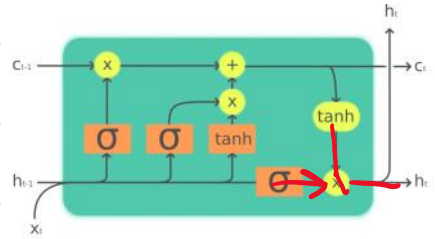
Output Gate

$$o_t = \sigma\left(W_o\begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_o\right)$$



Output

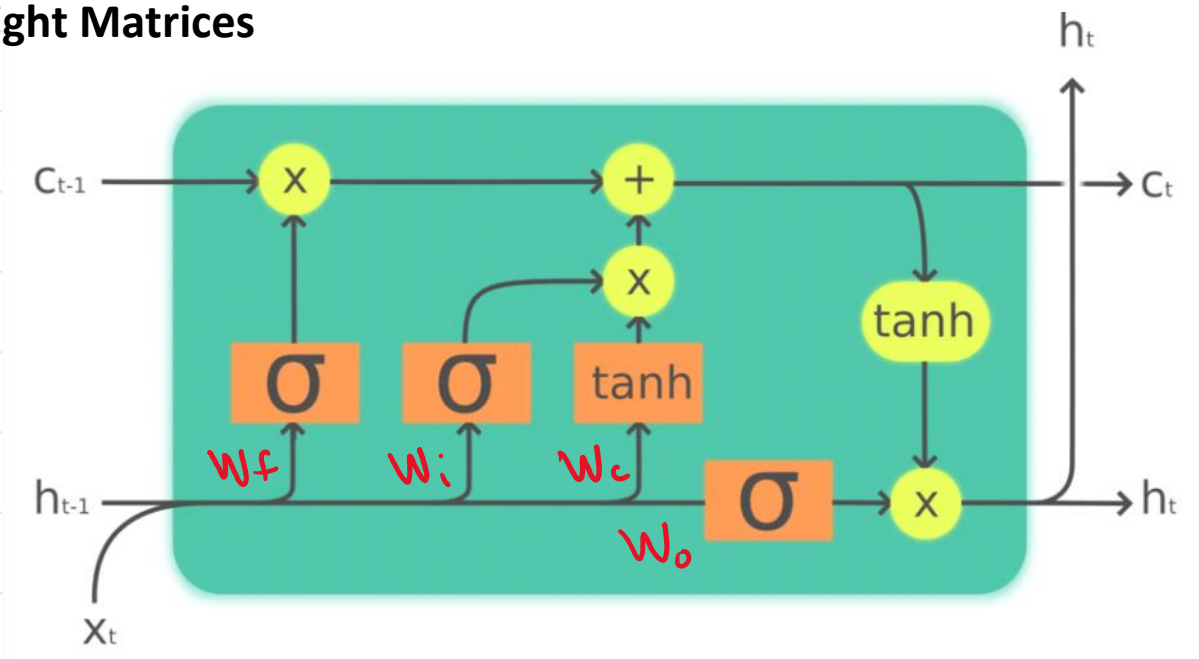
$$h_t = o_t \odot \tanh(c_t)$$



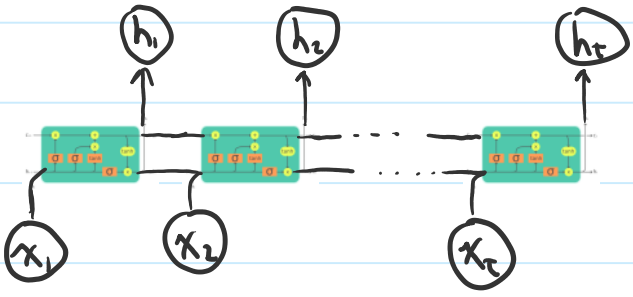
$$\begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}$$

$$C_t = \sigma\left(W_f X_t + b_f\right)C_{t-1} + \sigma\left(W_i X_t + b_i\right)\tanh\left(W_c X_t + b_c\right)$$

Weight Matrices



How does this make a difference?



$$\frac{\partial E}{\partial c_t} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial c_t} + \frac{\partial E}{\partial c_{t+1}}\frac{\partial c_{t+1}}{\partial c_t}$$

$$\frac{\partial c_{t+1}}{\partial c_t} = f_{t+1}$$

$$\frac{\partial E}{\partial c_t} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial c_t} + \frac{\partial E}{\partial c_{t+1}}\frac{\partial c_{t+1}}{\partial c_t} \qquad \frac{\partial c_{t+1}}{\partial c_t} = f_{t+1}$$

$$\frac{\partial E}{\partial c_{t+1}} = \frac{\partial E}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial c_{t+1}} + \frac{\partial E}{\partial c_{t+2}}\frac{\partial c_{t+2}}{\partial c_{t+1}}$$

$$\frac{\partial E}{\partial c_t} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial c_t} + \left(\frac{\partial E}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial c_{t+1}} + \frac{\partial E}{\partial c_{t+2}} f_{t+2}\right) f_{t+1}$$

$$= \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial c_t} + \frac{\partial E}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial c_{t+1}} f_{t+1} + \square\, f_{t+2} f_{t+1} + \cdots$$

Suppose that the forget gate is close to 1, and the input gate is 0. Then,

$$\frac{\partial E}{\partial c_t} = \sum_{j=t}^{T} \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial c_j}$$

Unit 5:

Population Coding

If we know what computation we want our neural network to do, how do we design the network to perform the task?

In this unit, we will essentially derive a compiler for a neural machine.
Topics include...

- Population coding
- Transformations through connections
- Recurrent network dynamics
- Learning decoders

Neural Coding 1

Recall the firing rate of an LIF neuron,

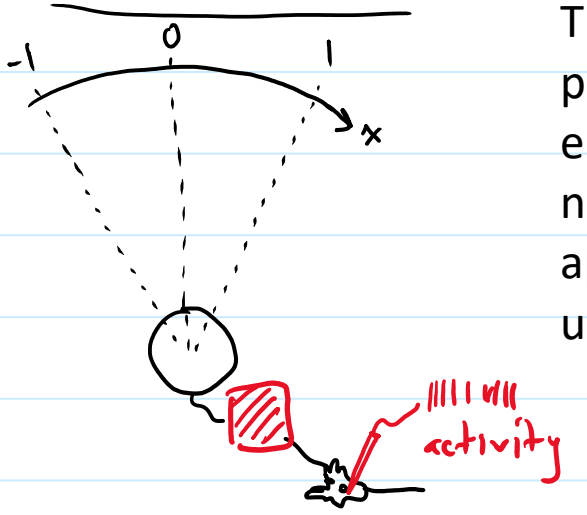
$$G(J) = \begin{cases} \dfrac{1}{\tau_{ref} - \tau \ln\left(1 - \frac{J_{th}}{J}\right)} & \text{for } J > J_{th} \\ 0 & \text{otherwise} \end{cases}$$

Given samples of firing rate over a variety of different values, we can characterize the activity of a neuron with respect to an environmental quantity. (Hubel & Wiesel video)

If a neuron's activity changes as the value changes, then we say that the neuron that quantity.

(cat electrophysiology demonstration)

eg. Gaze direction



There are many intermediate processes between the environmental value and the neurons that encode it. We will approximate all those processes using a linear remapping,

$$J(x) = \alpha e x + \beta$$

$\alpha \in \mathbb{R}_{\geq 0}$ "gain" $e \in \{-1, 1\}$ "encoder" $\beta \in \mathbb{R}$ "bias"

$J(x)$ is the current entering the neuron for stimulus x.
Hence, a LIF neuron is characterized by 6 parameters:

1) J_{th} - threshold (always = 1 for)
2) τ_m - membrane time const
3) τ_{ref} - refractory period

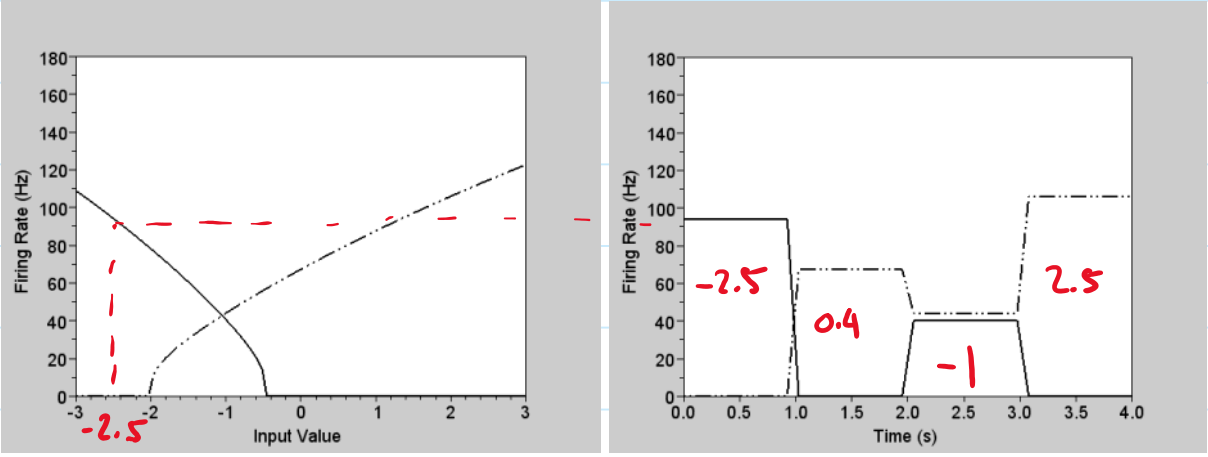
3) τ_{ref} - refractory period

4) α - gain

5) β - bias

6) e - encoder

Exercise: Given the tuning curves for the two neurons on the left, estimate the sequence of encoded values from 0 to 4 seconds in the plot on the right.



This is called decoding.

Ideally, you could decode with a small number of neurons (like 2, in the example). But neuronal firing rates appear to have a stochastic (random) component (at least, with respect to our variable of interest).

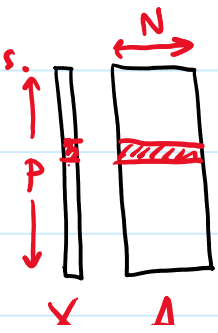
=> More observations improve the statistics of our inference.

Optimal Linear Decoding

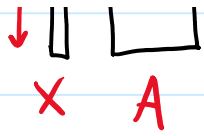
Suppose we have an array of neural activities over a range of x-values:

i.e. $X \in \mathbb{R}^{P \times 1}$ is a column vector containing x-vals.

$A \in \mathbb{R}^{P \times N}$ each row contains the activities (firing rates) of N neurons for the corresponding x-value

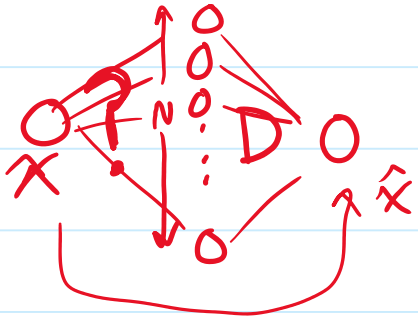


N neurons for the corresponding x-value

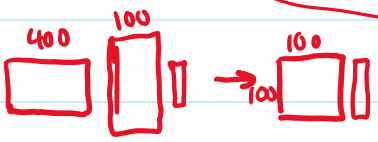


We want to find a set of decoding weights $D \in \mathbb{R}^{N \times 1}$ s.t.

$$AD \approx X \quad \text{or} \quad \min_D \|AD - X\|_2^2$$



Least Squares Solution



Method 1: Normal Equations

$$A^T(AD - X) = 0$$

$$A^TAD = A^TX \quad \Leftarrow \text{Normal Equations}$$

A^TA: $N \times N$

$$D = (A^TA)^{-1}A^TX = A^+X$$

A^+: Matrix pseudo inverse of A

This is an efficient method, but can be unstable if A is ill-conditioned.

Method 2: SVD

Let $U\Sigma V^T = A$ (Σ: diagonal; U, V: orthogonal)

$AD \approx X \Rightarrow$

$$Ax = b$$

$$U^TU\Sigma V^Tx = U^Tb$$

$$\Sigma V^Tx = U^Tb$$

$$V^Tx = \Sigma^{-1}U^Tb$$

$$x = V\Sigma^{-1}U^Tb$$

This method is more expensive, but more numerically stable.

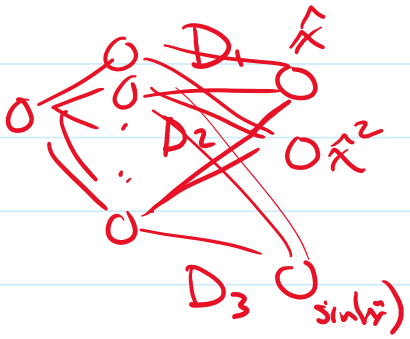
Python demo

Nengo.ca

Neural Coding 2

Behavioural Sampling

1. Choose a sampling of stimulus values, store them in $X \in \mathbb{R}^{P \times 1}$
2. Present each stimulus to the animal & measure neural responses, store them in $A \in \mathbb{R}^{P \times N}$
3. Solve for decoders $D = \underset{D}{\operatorname{argmin}} \|AD - X\|_2^2$
4. Now you can predict the stimulus from only the neural activities



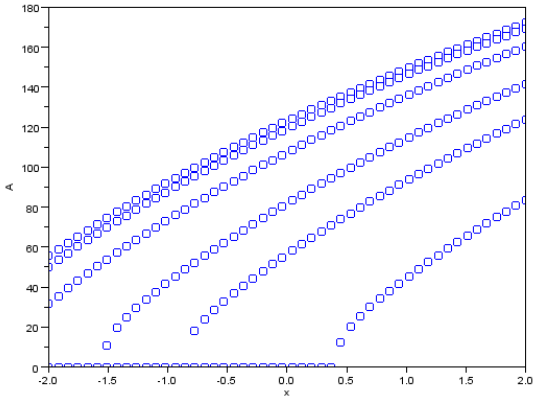
Moreover, we can decode other functions of x too.
To decode $f(x)$, simply solve

$$\min_D \|AD - f(x)\|_2^2$$

Python demo

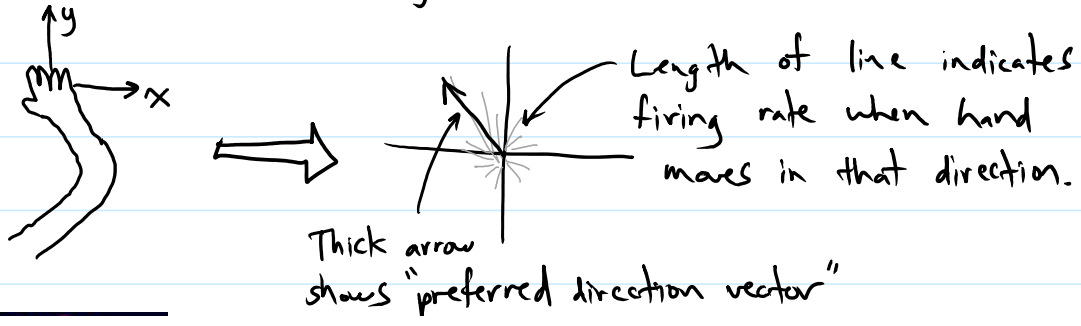
But there's more to it...

(demo)

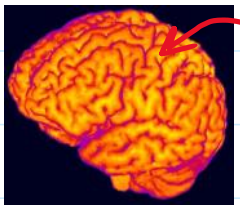


What's going on here? Why does this neuron's tuning curve keep changing?

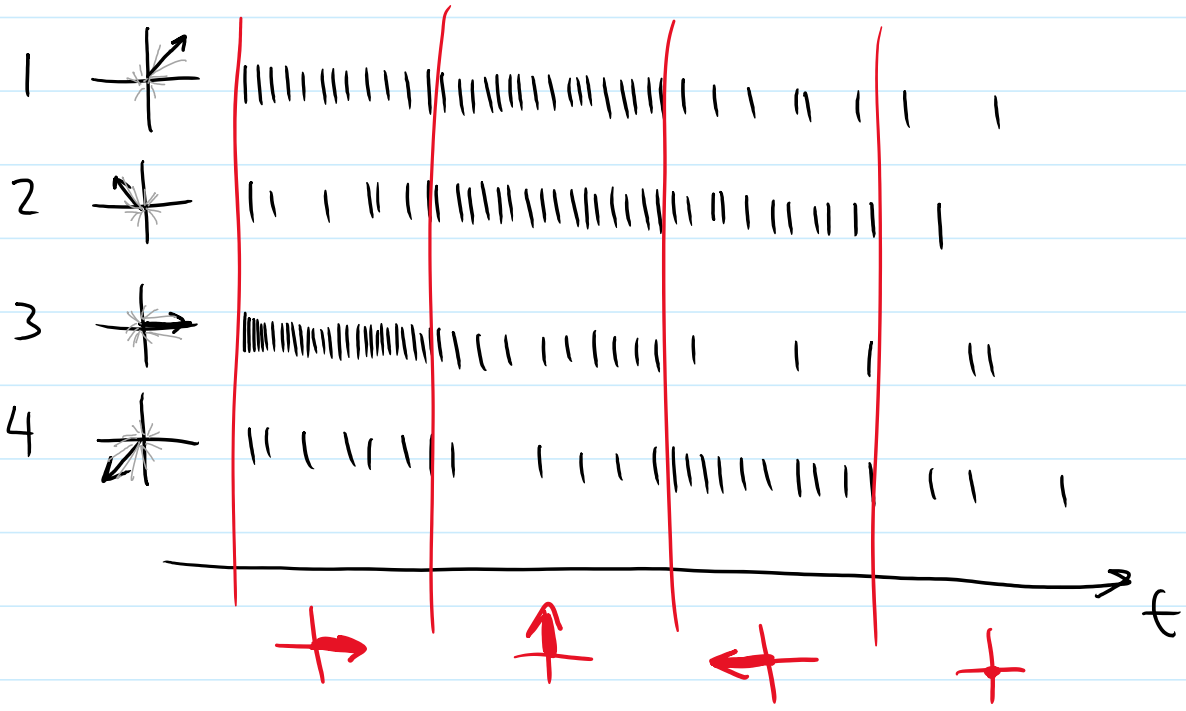
CASE: Georgopoulos, Science 233, pp 1416, 1986
"Neural Population Coding of Movement Direction"



Thick arrow!
shows "preferred direction vector"



Probed neurons in monkey's motor cortex



One approach:

Hand movement can be predicted as a weighted sum of the preferred direction vectors, weighted by each neuron's firing rate.

$$\text{hand movement} = a_1 \vec{x}_1 + a_2 \vec{x}_2 + a_3 \vec{x}_3 + a_4 \vec{x}_4$$

a_1: firing rate; $\vec{x}_1$: pref. vector

$$= \sum_n a_n \vec{x}_n$$

This works in some cases, when the neurons are uniformly distributed over the directions.

For other cases, we need a more statistically

For other cases, we need a more statistically principled approach.

BMI video

Vector Encoding

If a neuron's activity is influenced by 2 variables, then we could formulate that relationship using a 2D encoding vector.

Recall: The injected current for 1D encoding was computed

$$J(x) = \alpha e x + \beta$$

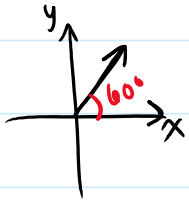
≥ 0 $\quad \{-1, 1\}$ $\quad \mathbb{R}$ $\quad \mathbb{R}$

For 2 external variables, (x, y)

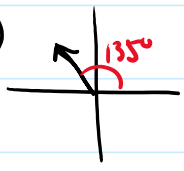
$$J(x,y) = \alpha (e_1, e_2) \cdot (x, y) + \beta$$

or $J(\vec{x}) = \alpha \vec{e} \cdot \vec{x} + \beta$ where $\|\vec{e}\| = 1$.

In the case of the monkey's arm movements,

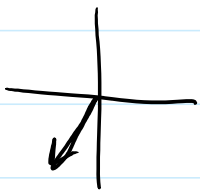
1) 

$\vec{e}_1 = \begin{bmatrix} \cos 60° \\ \sin 60° \end{bmatrix}$

2) 

$\vec{e}_2 = \begin{bmatrix} \cos 135° \\ \sin 135° \end{bmatrix}$

3) 

4) 

In fact, our neurons could depend on more variables. Let's say K of them: $\vec{x} \in \mathbb{R}^K$

In fact, our neurons could depend on more variables, let's say k of them i.e. $\vec{x} \in \mathbb{R}^k$

The same formula still holds:

$$J_n(\vec{x}) = \alpha_n \vec{e}_n \cdot \vec{x} + \beta_n$$

but with encoders $\vec{e}_n \in \mathbb{R}^k$

Suppose we have measured the neuron activities for a bunch of $\vec{x}$-values.

i.e. $X \in \mathbb{R}^{P \times k}$ is a $P \times k$ matrix containing one set of k variables in each row.

$A \in \mathbb{R}^{P \times N}$ is as before

And, as before, we seek linear decoding weights D that yield the least-squares solution,

$$\min_D \|AD - X\|_2^2 \Rightarrow D = (A^T A)^{-1} A^T X$$

But in this case, $D \in \mathbb{R}^{N \times k}$.

And, as before, we can decode functions of $\vec{x}$,

$$\min_D \|AD - f(X)\|_2^2 \quad \text{where } f: \mathbb{R}^k \to \mathbb{R}^L \text{ & } D \in \mathbb{R}^{N \times L}$$

Example: Given (x_p, y_p) stimulus and corresponding neural activities $(a_{p1}, a_{p2}, \dots, a_{pn})$

$\rightarrow X \in \mathbb{R}^{P \times k}$

$\rightarrow A \in \mathbb{R}^{P \times N}$

Decode $f(x,y) = xy$

Compute $Q \in \mathbb{R}^{P \times 1}$ where $q_p = x_p y_p$

Compute $Q \in \mathbb{R}^{T \times 1}$ where $q_p = x_p y_p$

Solve $\min_D \| AD - Q \|_2^2$ (demo 2D product)

Neural Decoding Overview

1) Behavioural Sampling

Biology:

Simulation:
- choose X, use Simulation to get A

2) Compute Decoders

Use data from behavioural sampling to solve

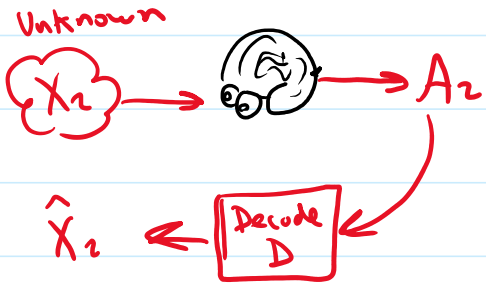
$$D = \underset{D}{\operatorname{argmin}} \| AD - f(x) \|_2^2$$

3) Decode Neural Activity

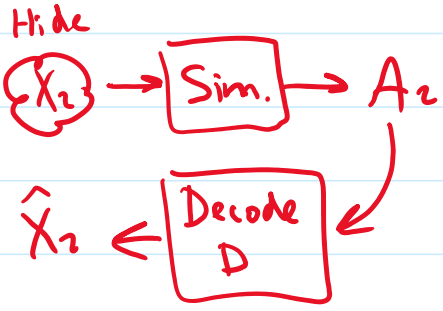
Now you can decode further neural activity

i.e. given A_2, you can estimate X_2

Biology:

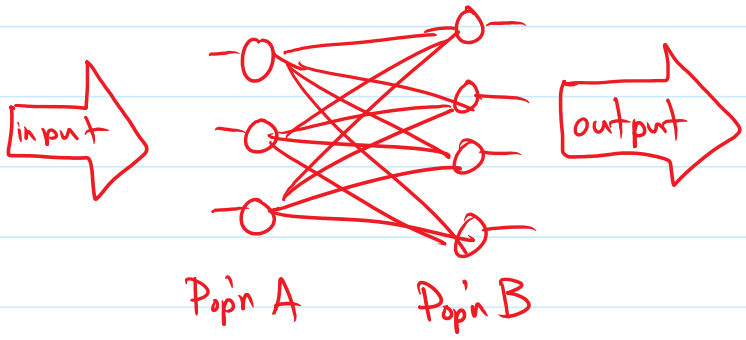


Simulation:



Neural Transformations

So far, we're only looked at interpretting the activity of a population of neurons. But neurons send their output to other neurons.

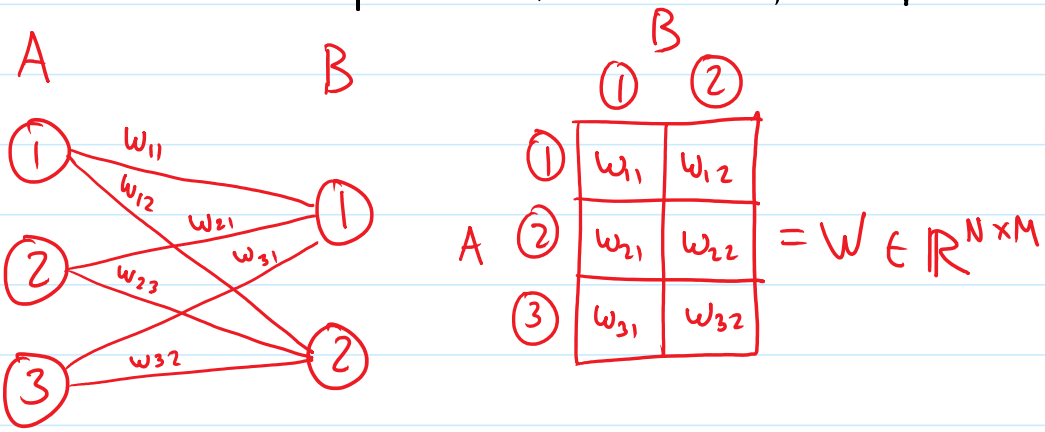


Suppose we have 2 populations A and B.

A has N neurons

B has M neurons

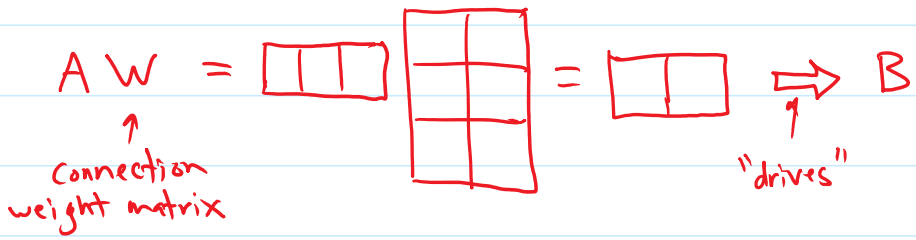
If every neuron in A sends its output to every neuron in B, how many connections would there be? How should we represent those connections?



This matrix notation is convenient... here's why. Let's store the activities of a population of neurons as a vector.

$$A = [a_1 \; a_2 \; a_3] \qquad B = [b_1 \; b_2] \qquad \text{(row vectors)}$$

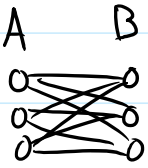
Then, to find out how A influences B, we use



> Note: We can equivalently store A & B as column vectors, giving us $W^T A^T \Rightarrow B^T$. But in this course, we will tend to use row vectors.

Encoder, Decoders and W (Oh my!)

Let's start with a simple example: we want B to encode the same value that is encoded in A.



Question: Can we simply copy the neural activities from A to B?

ie. $b_1 = a_1$

$b_2 = a_2$

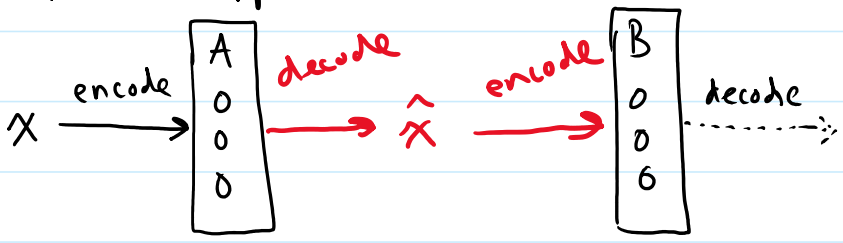
$\vdots$

Answer: No

1) Encoders are probably different . . .

2) A & B might have a different # of neurons

A different approach:



Decode from A, re-encode in B.

Step-by-step...

1) Encode x into A

$$A = G(J(x)) = G(xE_A\alpha_A + \beta_A) \in \mathbb{R}^{1\times N}$$

$1\times k$ $k\times N$ $N\times N$ diagonal $1\times N$

2) Decode the value from A

$$\hat{x} = AD_A \quad \text{(identity decoders)}$$

3) Re-encode into B

$$B = G(J(\hat{x})) = G((AD_A)E_B\alpha_B + \beta_B)$$

$M\times M$ diag.

$1\times N$ $N\times k$ $k\times M$ $1\times M$

$$= G(A(D_AE_B\alpha_B) + \beta_B) = G(AW + \beta_B)$$

$W \in \mathbb{R}^{N\times M}$

4) Decode from B (if desired)

$$\hat{\hat{x}} = BD_B \approx x$$

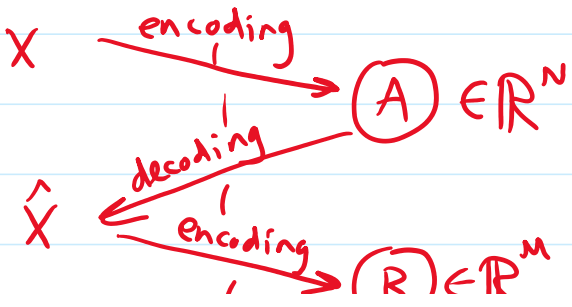
$1\times M$ $M\times k$

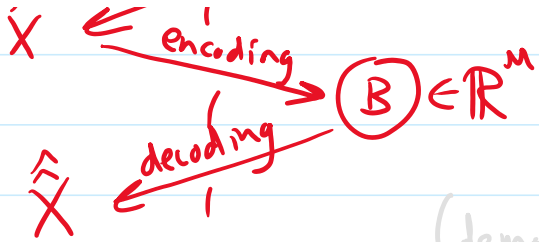
Pictorial Schematic of Mapping Between Spaces

It can help to think of the above operations as mapping back and forth between 2 spaces.

Representational $\longleftrightarrow$ Neural Activity

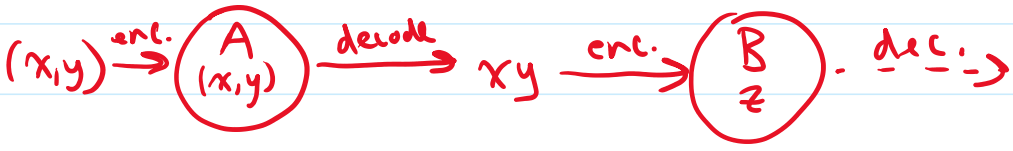
$\mathbb{R}^k$





Of course, the reason we connect populations of neurons is so that we can perform transformations on the data.

Example: Given input (x,y), we want to build a network that encodes xy (the product).



Step-by-step

1) Encode (x,y) into A

$$A = G\left(\underset{1\times 2}{X} E_A \alpha_A + \beta_B\right) \in \mathbb{R}^{1\times N}$$

(E_A: $2\times N$)

2) Decode xy from A

$$\hat{z} = AD_A \quad \text{where} \quad D_A = \underset{D}{\operatorname{argmin}} \left\| AD - z \right\|_2^2$$

(z: $X_{p,1}\ X_{p,2} \in \mathbb{R}^{P\times 1}$)

3) Re-encode $\hat{z}$ into B

$$B = G\left(AD_A E_B \alpha_B + \beta_B\right)$$

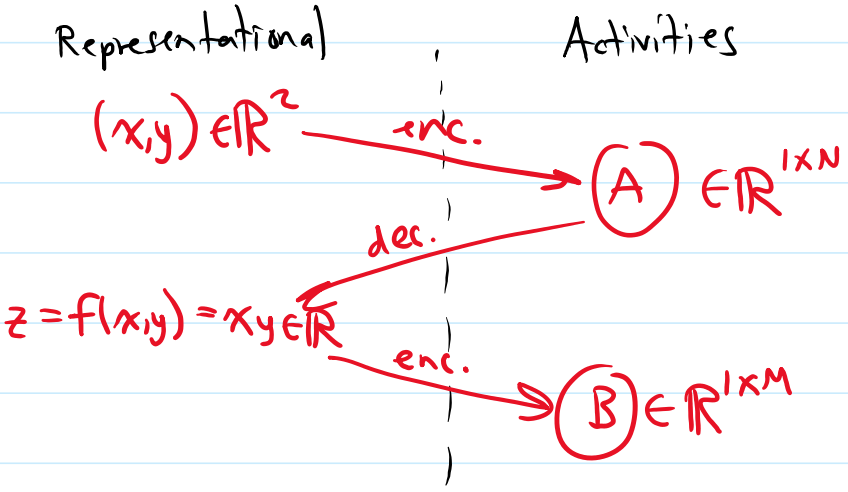
$$= G\left(AW + \beta_B\right)$$

4) Decode from B (if desired)

$$\hat{\hat{z}} = BD_B$$

$\epsilon - \delta L_B$

Pictorial View



Linear Transformations

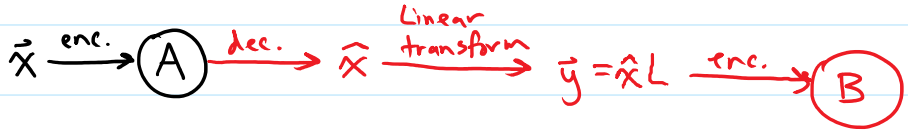
Consider the case where A encodes a vector $\vec{x} \in \mathbb{R}^{k_1}$, and we want B to encode a linear transformation of $\vec{x}$.

i.e. if B encodes $\vec{y} \in \mathbb{R}^{k_2}$, then we want

$$\vec{y} = \vec{x} L$$

$L \uparrow \mathbb{R}^{k_1 \times k_2}$

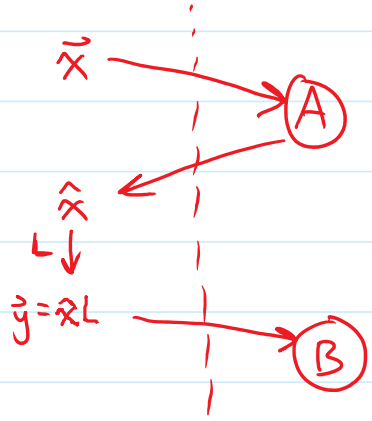
Here's how.



As before, we can combine the encoders and decoders...

$$B = G\left(\underbrace{A D_A L E_B}_{W} \alpha_B + \beta_B\right)$$

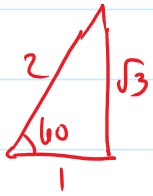
We can compute these connection weights using the identity decoders, by "sandwitching" L between the decoders & encoders.



Challenge: 2 populations, A & B, both 2D
Given their encoders & decoders, write the weight matrix that will rotate the vector in A by 60° and encode that value in B.

Answer: Rotation is a linear operation

$$R_\theta = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \quad R_{60^\circ} = \begin{bmatrix} \frac{1}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix}$$



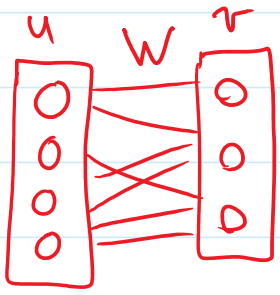
$$W = D_A R_{60^\circ} E_B \alpha_B$$

Network Dynamics

The dynamic model of a leaky integrate-and-fire (LIF) neuron contains two time-dependent processes:

Current: $\tau_s \frac{dJ}{dt} = -J + uW + \beta$ (syn. time const.)

Neural Activity: $\tau_m \frac{dr}{dt} = -r + G(J)$ (τ_m: Membr. time const.; G: Activation fcn)



Variants:

$\tau_m \ll \tau_s$: If the neuron membranes react quickly (ie. reach steady-state firing rate quickly) compared to synaptic dynamics...

$$\begin{cases} \tau_s \frac{dJ}{dt} = -J + uW + \beta \\ r = G(J) \end{cases}$$

$\tau_s \ll \tau_m$: If the PS current changes quickly compared to firing rate...

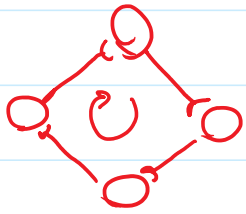
$$\begin{cases} J = uW + \beta \\ \tau_m \frac{dr}{dt} = -r + G(J) = -r + G(uW + \beta) \end{cases}$$

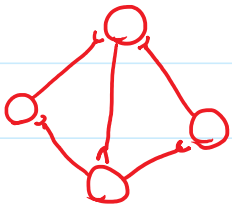
In steady state

$$r = G(uW + \beta)$$

Recurrent Networks

If the neurons of a population are connected to other neurons in the same population, then we say the network is "recurrent". In particular, recurrent networks can have circuits. eg.



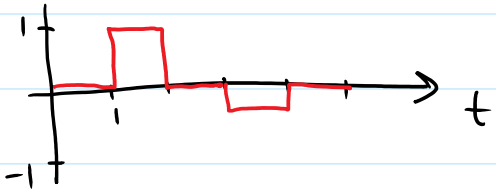


etc.

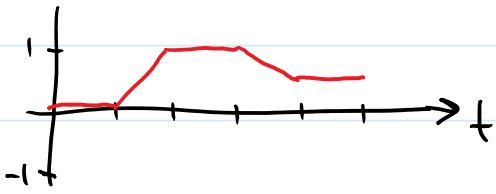
this feedback can lead to interesting dynamics.

Neural Integrator

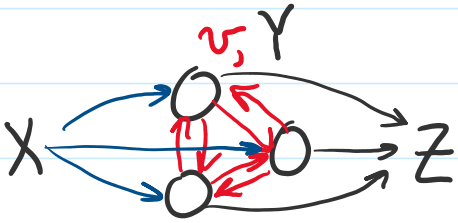
eg. Input value

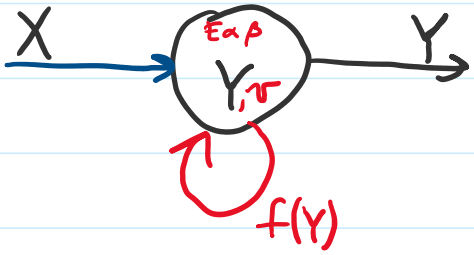


Decoded value



Consider a fully-connected population of N neurons.





Encoding $E\alpha, \beta$

Recurrent connections $W = D_f E\alpha$

Identity decoding $D, Z = vD$

For some initial state of the neurons, r, suppose $Y = rD$.
If there is no input (ie. $X=0$), then we want our population to maintain its value.

$$r = G(rW+\beta)$$
$$r = G(rD_f E\alpha+\beta)$$
$$rD = G(rD_f E\alpha+\beta)D = Y = G(YE\alpha+\beta)D$$
$$\therefore D_f = D \quad \text{identity decoders}$$

What about integrating the input?
We let the time dynamics be dictated by the synaptic time constant.
So we set $\tau_m = 0$.
Then

$$r = G(J)$$

and

$$\tau_s \frac{dJ}{dt} = -J + G(J)W + \beta + J_i$$

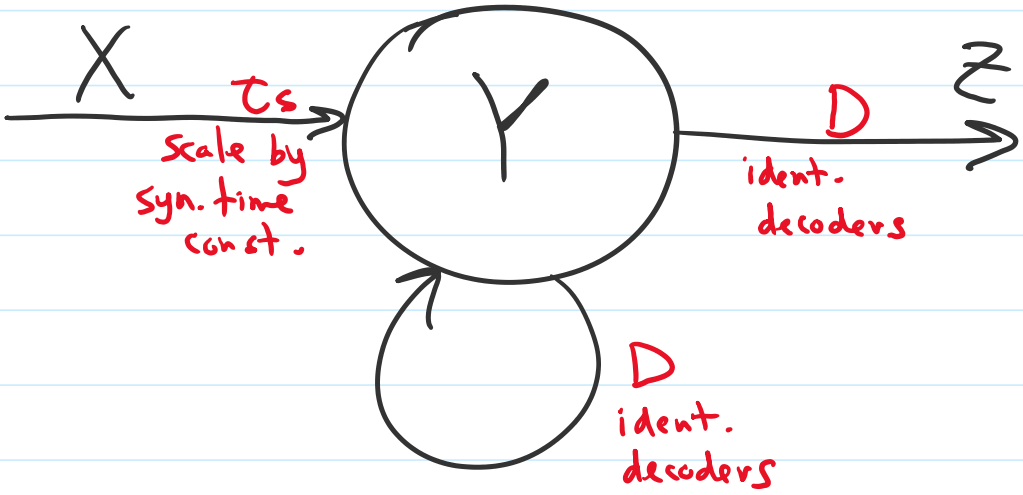
To get the input current, J_i, we need to encode the input value, X.

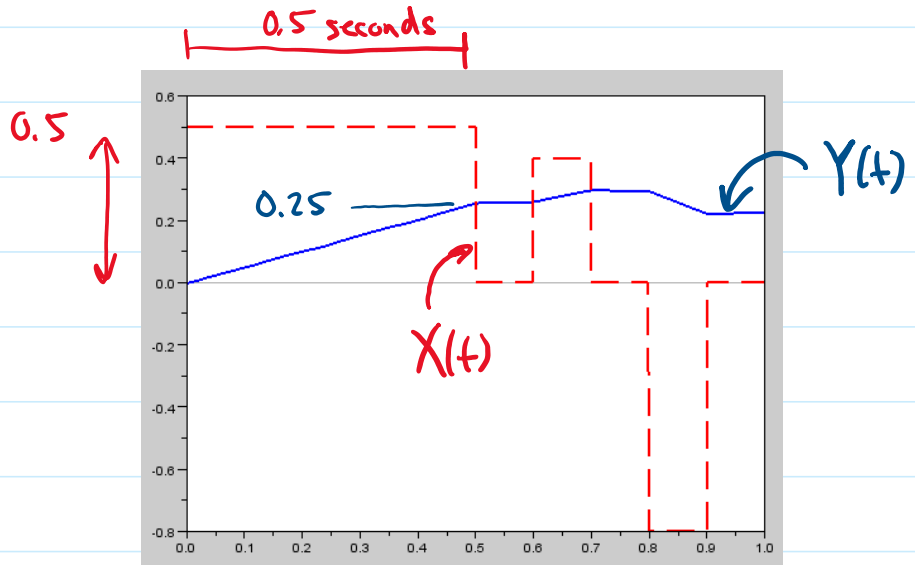
$$\Rightarrow XE\alpha$$

$$\frac{dJ}{dt} = \frac{rW+\beta-J}{\tau_s} + \tilde{J}_i \qquad \tilde{J}_i = XE\alpha$$

$$\tau_s \frac{dJ}{dt} = rW+\beta-J+\tau_s XE\alpha$$
$$= (rD+\tau_s X)E\alpha+\beta-J$$
$$= (Y+\tau_s X)E\alpha+\beta-J$$

Y: identity feedback
$\tau_s X$: input scaled by syn. time const.





Let's test how well our dynamic network can maintain its value. To do it, we'll run Dynamics, passing it an initial state that we want it to hold.

```
V = Dynamics(x, dt, lif, w, tau_s, tau_m, V0=None)
```

input x(t)

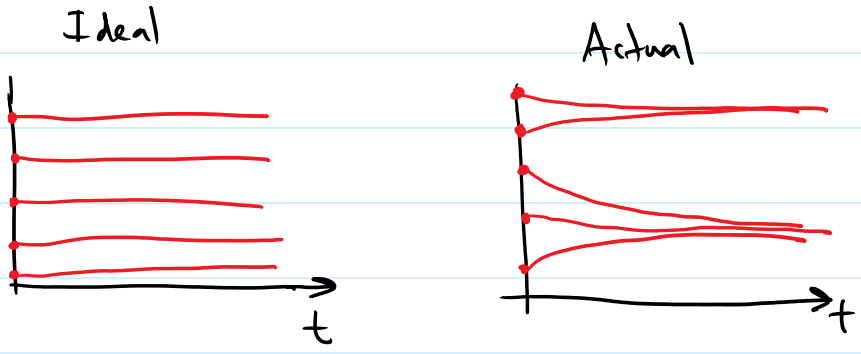
initial v (optional)

1) Choose an x-value to hold
2) Use x to find steady-state firing rates

$$v = G(J(x)) \rightarrow V0$$

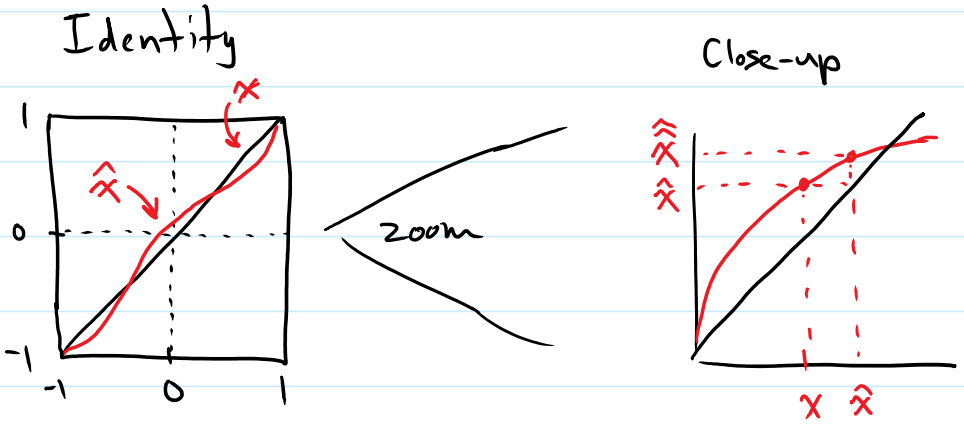
$r = G(J(x)) \rightarrow V\theta$

3) Call the Dynamics function. (demo steady-state)

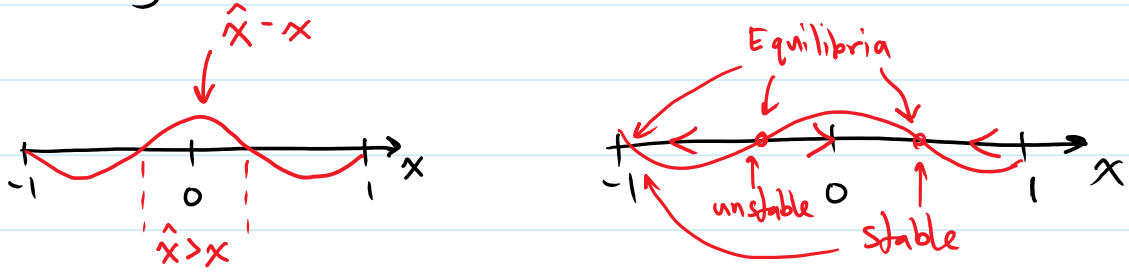


What went wrong?

A: Since the encoding is not perfect, the state of the network starts to drift until it reaches an equilibrium state.



Encoding Error ($\hat{x}-x$)



Will cause the next iteration to yield an even larger estimate $\hat{\hat{x}}$. (demo attractor)

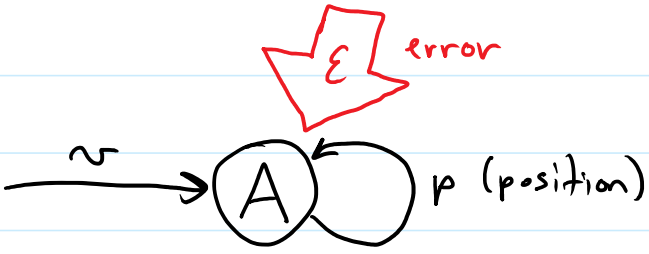
Learning Decoders

Thus far in this unit, we've built neural networks by prescribing connection weights. We decide what computation we want, and then choose the decoders to implement the required transformations.

As we previously discussed, most brains don't have an oracle to lay down a set of connection weights.

However, brains DO have error signals. For example, the oculomotor system receives corrective saccades from parts of the visual system. As it turns out, the oculomotor system (the part of the brain that tenses the muscles around the eyes to rotate the eyeballs and direct gaze) is an integrator. That is, it receives velocity input, and integrates that input to monitor gaze direction.
(see section 5.3 in Eliasmith & Anderson, 2003)

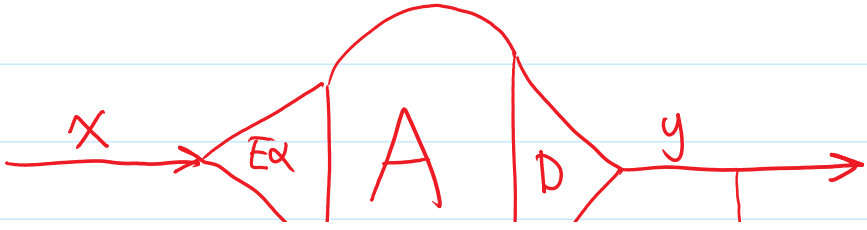
As we saw, integrators can drift, so when they do, the visual system detects "retinal slip" and generates a small, corrective saccade. This correction can be thought of as error feedback to the integrator.



How can that error be used to update the connection weights?

Learning a Decoder

Let's start with a simpler model.



Consider the decoding error,

$$\varepsilon = \|y - f(x)\|^2 = \|AD - f(x)\|^2 \qquad A = G(J(x))$$

To minimize this error iteratively, we can use gradient descent.

$$\frac{\partial \varepsilon}{\partial D} = 2A^T(AD - f(x))$$

This is the gradient vector that points "uphill" for the error function, or in the direction of greatest rate of increase. To move to a position (D) with lower error, you move in the direction opposite the gradient vector.

$$\Delta D = -\tilde{k}\, 2A^T(AD - f(x)) = k A^T(f(x) - AD)$$

More generally,

$$\Delta D = k A^T(\text{error})$$

learning rate; neural activity; error

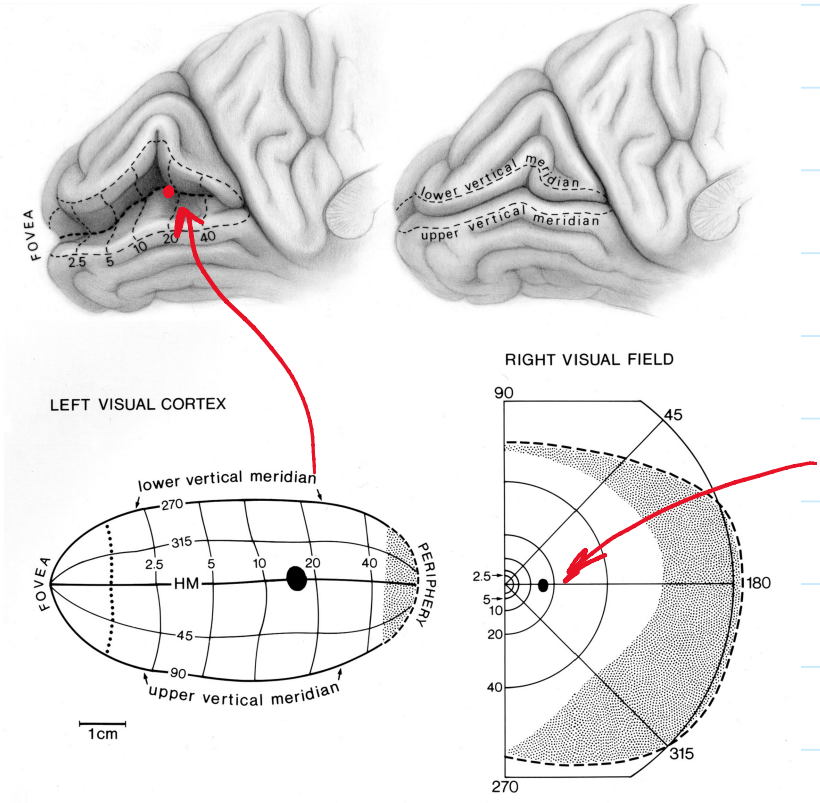
Convolutional Neural Networks

Most of the networks we have looked at assume an all-to-all connectivity between populations of neurons, like between layers in a network. But that's not the way our brains are wired, thankfully.

If every one of your 86 billion neurons was connected to every other neuron, your head would have to be MUCH bigger.

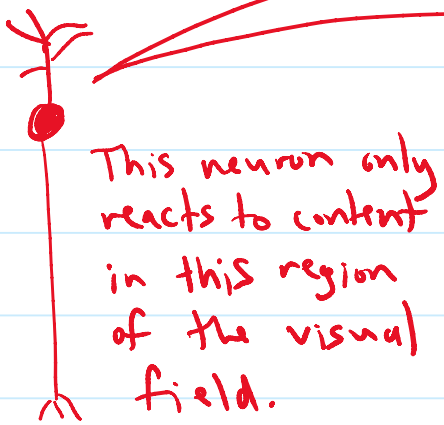
Instead, the neurons in a real brain tend to be sparsely connected. Example: your visual system

http://vision.ucsf.edu/hortonlab/ResearchProgram%20Pics/retinotopicMap.jpg



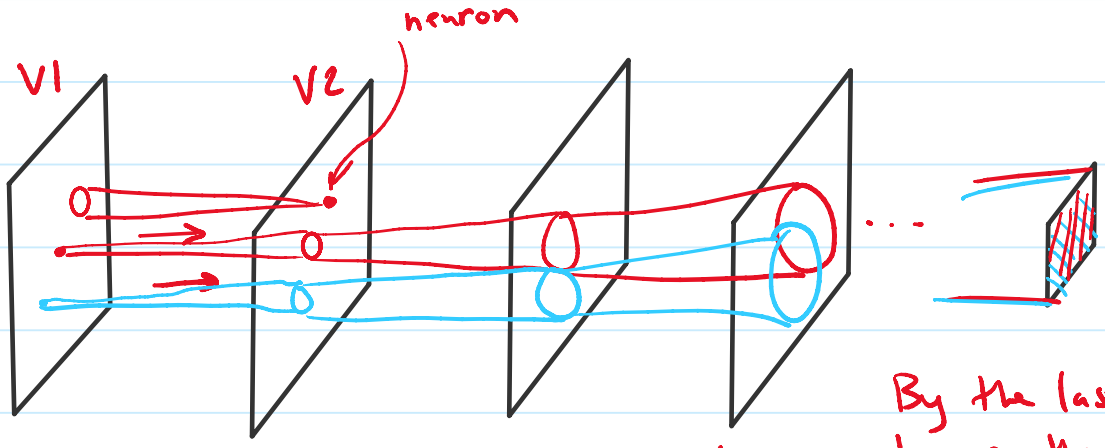
This dot in the visual field only excites a small patch of neurons in V1.

Conversely, each neuron in V1 is only activated by a small patch in the visual field.



This topological mapping between the visual field and the surface of the cortex is called a retinotopic mapping.

Moreover, neurons in V1 project to the next layer, V2, and again, the connections are retinotopically local.



The "footprint" of a region in the visual field gets larger as the data progresses up the layers.

By the last layer, the neurons are influenced by the entire visual field.

Inspired by this topological local connectivity, and in an effort to reduce the number of connection weights that need to be learned, scientists devised the Convolutional Neural Network (CNN).

Convolution:

Continuous domain $f, g : \mathbb{R} \to \mathbb{R}$

$$(f * g)(x) = \int_{-\infty}^{\infty} f(s) \cdot g(x - s)\, ds$$

Discrete domain $f, g \in \mathbb{R}^N$

$$(f * g)_m = \sum_{n=0}^{N-1} f_n \, g_{m-n}$$

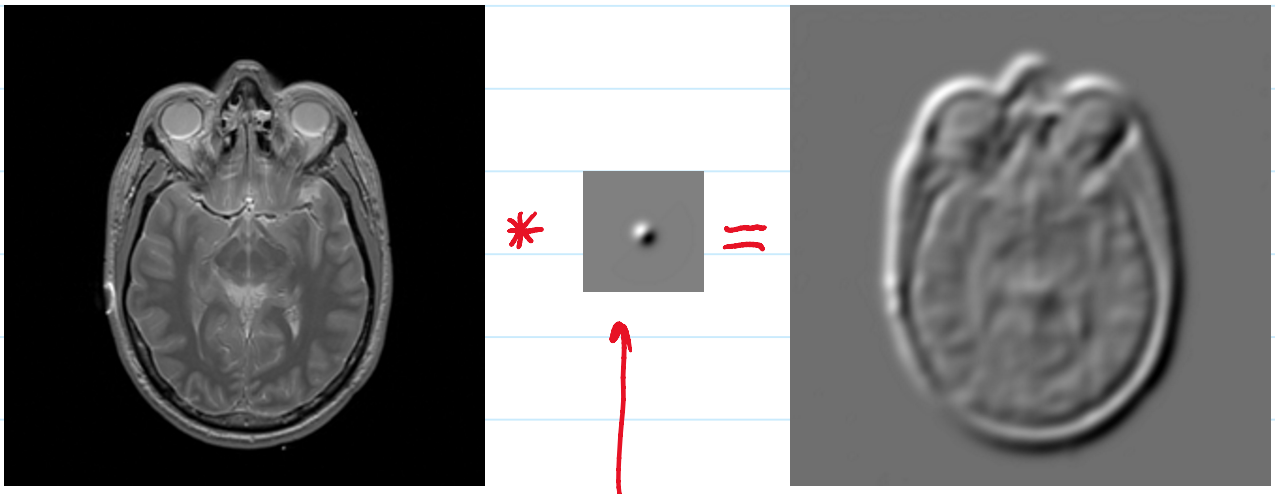
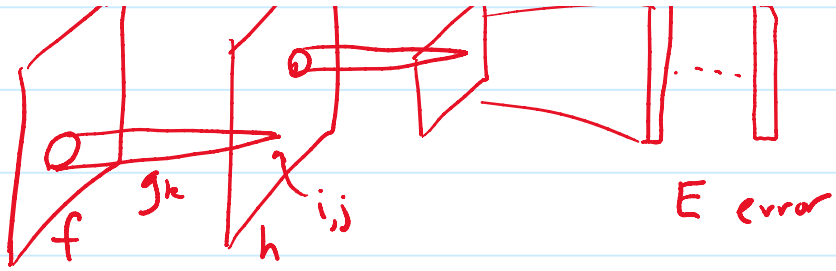


Image f | conv. kernel g "feature" | "Feature map" or "Activation map" h

in 2D $(f * g)_{ij} = \sum_{m,n} f_{ij} \, g_{m-i,n-j}$

Derivatives



$$\frac{\partial E}{\partial g_k} = \sum_{ij} \cdots = \text{essentially a conv. b/w } \frac{\partial E}{\partial z} \text{ and } f$$

$$h = \sigma(z)$$

$$\frac{\partial E}{\partial z} = \sum \cdots = \text{conv. b/w kernel from layer above and } h$$

Adversarial Examples in Machine Learning CS489/698

Rey Reza Wiyatno
Email: reywiyatno@gmail.com

Agenda

Intro

Adversarial Attacks

Adversarial Defenses

Takeaways

Agenda

Intro

Adversarial Attacks

Adversarial Defenses

Takeaways

Adversarial Examples

Inputs specifically designed to cause a model to make mistakes in its prediction, although they look like valid inputs to a human.

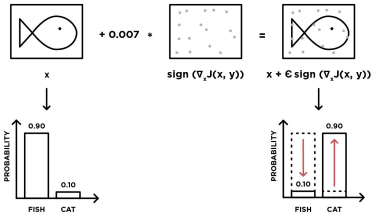


Illustration of adversarial example.

Source: https://medium.com/element-ai-research-lab/tricking-a-machine-into-thinking-youre-milla-jovovich-b19bf322d55c

Standard Definition in Classification Task

Generate adversarial example ***x'*** that looks like an original example ***x***, but is mispredicted by a model. Formally:

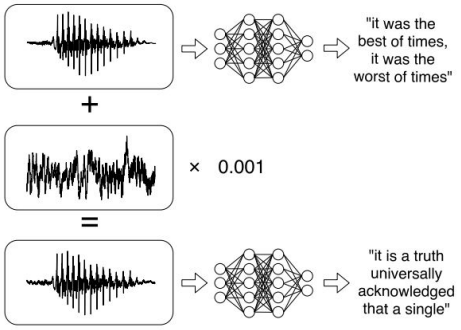
Given a model ***f*** and an input ***x***, find ***x'*** where $\|x - x'\|_p < \epsilon$, such that $f(x) \neq f(x')$

Note: other perceptual similarity metrics other than the L_p norm can also be used.

Adversarial Examples: Adversarial Patch

Adversarial Patch (Brown et al., 2017)

Adversarial Examples: Adversarial Audio



Demo: https://nicholas.carlini.com/code/audio_adversarial_examples/
Audio Adversarial Example: Targeted Attacks on Speech-to-Text (Carlini & Wagner, 2018)

Common Terms

Adversarial attacks: methods to generate adversarial examples.

Adversarial defenses: methods to defend against adversarial examples.

Adversarial robustness: property to resist misclassification of adversarial examples.

Adversarial detection: methods to detect adversarial examples.

Transferability: adversarial examples generated to fool a specific model can also be used to fool other models.

Common Terms

Adversarial perturbation: difference between original example and its adversarial counterpart

Whitebox attack: when attacker has full access to the victim model

Blackbox attack: when attacker only has access to victim's output

Targeted attack: when attacker wants an adversary to be mispredicted in a specific way

Non-targeted attack: when attacker does not care if an example is mispredicted in a specific way

Agenda

Intro

Adversarial Attacks

Adversarial Defenses

Takeaways

Goal:
Minimally modify inputs to confuse the victim model

We can re-formulate as:
Minimally modify inputs to maximize some loss function

Any ideas how?

This categorization is only based on methods covered in this presentation and does not encompasses every single attack or defense method that exist as of today.

Attack & Defense Methods

Attack strategies:

Whitebox:
- Direct gradient step(s): FGSM, BIM, R+FGSM, DAG
- Gradient-based greedy algorithm: JSMA
- Iterative optimization: L-BFGS, C&W, UAP, Adversarial Eyeglasses, RP_2, EOT
- Parameterized optimization: ATN

Blackbox:
- Decision boundary approximation: SBA
- Evolutionary algorithm: One Pixel Attack
- Finite difference method: ZOO

Defense strategies:

Data augmentation:
- Adversarial training
- Ensemble adversarial training
- PGD adversarial training

Gradient regularization:
- DCN
- Defensive distillation

Input manipulation:
- PixelDefend

Detection methods:
- Adversary detector networks
- Feature squeezing

Fast Gradient Sign Method (FGSM)

- Take **gradient w.r.t. input**
- Then do **single-step gradient ascent**

$$x' = x + \epsilon \cdot \text{sign}(\nabla_x \mathcal{L}(x, y))$$

- Is above equation for targeted/non-targeted attack?
- If targeted, how to make it untargeted (and vice-versa)?
- Why use "sign" operator?

Note: ***L(x,y)*** and ***J(x,y)*** denote the training loss function, ***x*** is an input, ***y*** is the true label of ***x***, ϵ is a small constant where $\epsilon > 0$

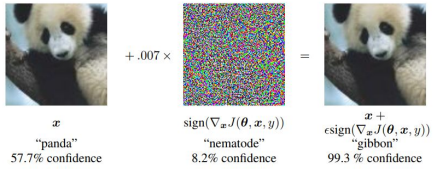


Illustration of FGSM. Adversarial example (right) misclassified as "Gibbon" with high confidence (Goodfellow et al., 2015).

Explaining and Harnessing Adversarial Examples (Goodfellow et al., 2015)

Implementing FGSM (see notebook)

Link: https://github.com/rrwiyatn/deep-learning/blob/master/fast_gradient_sign_attack/fgsm.ipynb

Basic Iterative Method (BIM)

- Iterative variant of FGSM

$$\boldsymbol{X}_0^{adv} = \boldsymbol{X}, \quad \boldsymbol{X}_{N+1}^{adv} = Clip_{X,\epsilon}\left\{\boldsymbol{X}_N^{adv} + \alpha\, \text{sign}\left(\nabla_X J(\boldsymbol{X}_N^{adv}, y_{true})\right)\right\}$$

- Targeted attack variant of BIM called Iterative Least-Likely Class Method (ILLCM)

$$\boldsymbol{X}_0^{adv} = \boldsymbol{X}, \quad \boldsymbol{X}_{N+1}^{adv} = Clip_{X,\epsilon}\left\{\boldsymbol{X}_N^{adv} - \alpha\, \text{sign}\left(\nabla_X J(\boldsymbol{X}_N^{adv}, y_{LL})\right)\right\}$$

$$y_{LL} = \arg\min_y \{p(y|\boldsymbol{X})\}$$

$$Clip_{X,\epsilon}\{\boldsymbol{X}'\}(x,y,z) = \min\left\{255, \boldsymbol{X}(x,y,z) + \epsilon, \max\left\{0, \boldsymbol{X}(x,y,z) - \epsilon, \boldsymbol{X}'(x,y,z)\right\}\right\}$$

- When to use ILLCM?

Adversarial Examples in the Physical World (Kurakin et al., 2017)
Adversarial Machine Learning at Scale (Kurakin et al., 2017)

This categorization is only based on methods covered in this presentation and does not encompasses every single attack or defense method that exist as of today.

Random FGSM (R+FGSM)

- FGSM variant with a random starting point

$$x^{\text{adv}} = x' + (\varepsilon - \alpha) \cdot \texttt{sign}\left(\nabla_{x'} J(x', y_{\text{true}})\right), \quad \text{where} \quad x' = x + \alpha \cdot \texttt{sign}(\mathcal{N}(\mathbf{0}^d, \mathbf{I}^d))$$

- Designed to circumvent a defense method called adversarial training (Goodfellow et al., 2015) in its naive implementation
- We will come back to the motivation later in defense section

Ensemble Adversarial Training: Attacks and Defenses (Tramèr et al., 2018)
Explaining and Harnessing Adversarial Examples (Goodfellow et al., 2015)

Attack & Defense Methods

Attack strategies:

Whitebox:
- Direct gradient step(s): FGSM, BIM, R+FGSM, DAG
- Gradient-based greedy algorithm: JSMA
- Iterative optimization: L-BFGS, UAP, C&W, Adversarial Eyeglasses, RP_2, EOT
- Parameterized optimization: ATN

Blackbox:
- Decision boundary approximation: SBA
- Evolutionary algorithm: One Pixel Attack
- Finite difference method: ZOO

Defense strategies:

Data augmentation:
- Adversarial training
- Ensemble adversarial training
- PGD adversarial training

Gradient regularization:
- DCN
- Defensive distillation

Input manipulation:
- PixelDefend

Detection methods:
- Adversary detector networks
- Feature squeezing

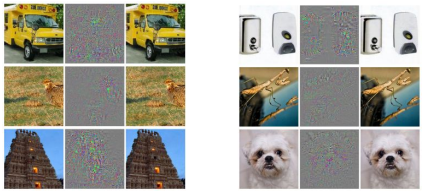
Details will be skipped since background knowledge in optimization (e.g., L-BFGS) is needed.

L-BFGS Attack

- Model adversarial example generation as optimization problem
- Use L-BFGS as optimizer
- Given a victim model ***f***, find ***r*** that minimizes:

$$c|r| + \text{loss}_f(x + r, l)$$
$$\text{subject to } x + r \in [0, 1]^m$$

- Note that the loss function does not have to be the same with the training loss of the victim



Adversarial examples generated using this method (Szegedy et al., 2014).

Intriguing Properties of Neural Networks (Szegedy et al., 2014)

Details will be skipped, this is just to provide an example on how to get creative with existing methods.

Universal Adversarial Perturbation (UAP)

Universal perturbations:

Single perturbation that can be added to multiple inputs to make them adversarial

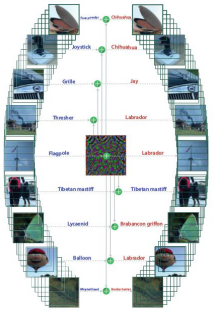


Illustration of UAP (Moosavi-Dezfooli et al., 2016).

Algorithm 1 Computation of universal perturbations.

input: Data points X, classifier $\hat{k}$, desired ℓ_p norm of the perturbation ξ, desired accuracy on perturbed samples δ.
output: Universal perturbation vector v.
Initialize $v \leftarrow 0$.
while $\text{Err}(X_v) \leq 1 - \delta$ **do**
  for each datapoint $x_i \in X$ **do**
    if $\hat{k}(x_i + v) = \hat{k}(x_i)$ **then**
      Compute the *minimal* perturbation that sends $x_i + v$ to the decision boundary:

$$\Delta v_i \leftarrow \arg\min_r \|r\|_2 \text{ s.t. } \hat{k}(x_i + v + r) \neq \hat{k}(x_i).$$

      Update the perturbation:

$$v \leftarrow \mathcal{P}_{p,\xi}(v + \Delta v_i).$$

  end if
end for
end while

$$\mathcal{P}_{p,\xi}(v) = \arg\min_{v'} \|v - v'\|_2 \text{ subject to } \|v'\|_p \leq \xi$$

Universal Adversarial Perturbations (Moosavi-Dezfooli et al., 2016)

This categorization is only based on methods covered in this presentation and does not encompasses every single attack or defense method that exist as of today.

Attack & Defense Methods

Attack strategies:

Whitebox:
- Direct gradient step(s): FGSM, BIM, R+FGSM, DAG
- Gradient-based greedy algorithm: JSMA
- Iterative optimization: L-BFGS, C&W, UAP, Adversarial Eyeglasses, RP_2, EOT
- Parameterized optimization: ATN

Blackbox:
- Decision boundary approximation: SBA
- Evolutionary algorithm: One Pixel Attack
- Finite difference method: ZOO

Defense strategies:

Data augmentation:
- Adversarial training
- Ensemble adversarial training
- PGD adversarial training

Gradient regularization:
- DCN
- Defensive distillation

Input manipulation:
- PixelDefend

Detection methods:
- Adversary detector networks
- Feature squeezing

Adversarial Transformation Network (ATN)

- Train a neural network to generate adversaries
- 2 variants: Adversarial Autoencoder (AAE) and Perturbation ATN (P-ATN)
- AAE: Given a victim model ***f***, train a generator network G_t on dataset ***X*** to output adversarial examples ***X'*** such that ***f(X') = t***, where ***t*** is a target misclassification class
- Every G_t can only be used generate adversarial examples that are misclassified as class ***t***

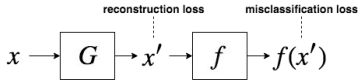
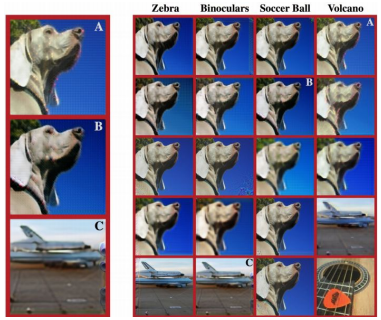


Illustration of AAE.

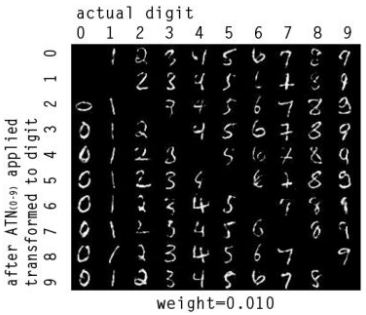
Adversarial Transformation Networks: Learning to Generate Adversarial Examples (Baluja & Fischer, 2017)

This categorization is only based on methods covered in this presentation and does not encompasses every single attack or defense method that exist as of today.

Adversarial Transformation Network (ATN)



Generated adversaries that fool Inception Resnet V2 (Baluja & Fischer, 2017)



Generated adversarial MNIST (Baluja & Fischer, 2017)

Adversarial Transformation Networks: Learning to Generate Adversarial Examples (Baluja & Fischer, 2017)

This categorization is only based on methods covered in this presentation and does not encompasses every single attack or defense method that exist as of today.

Attack & Defense Methods

Attack strategies:

Whitebox:
- Direct gradient step(s): FGSM, BIM, R+FGSM, DAG
- Gradient-based greedy algorithm: JSMA
- Iterative optimization: L-BFGS, C&W, UAP, Adversarial Eyeglasses, RP_2, EOT
- Parameterized optimization: ATN

Blackbox:
- Decision boundary approximation: SBA
- Evolutionary algorithm: One Pixel Attack
- Finite difference method: ZOO

Defense strategies:

Data augmentation:
- Adversarial training
- Ensemble adversarial training
- PGD adversarial training

Gradient regularization:
- DCN
- Defensive distillation

Input manipulation:
- PixelDefend

Detection methods:
- Adversary detector networks
- Feature squeezing

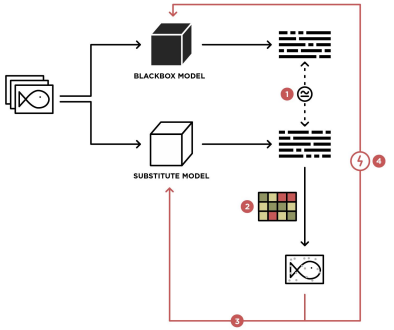
Substitute Blackbox Attack (SBA)

- Blackbox attack by approximating decision boundaries of victim
- Assumption: attacker has access to the softmax probability
- Procedures (see next slide):
  - Train a substitute model on a dataset **labelled by the victim**
  - Attack the substitute model using any whitebox methods
  - The generated adversaries should be transferable to the victim model (thanks to transferability property)
- Successfully attacked Google, Amazon, and MetaMind image recognition models

Practical Black-Box Attacks against Machine Learning (Papernot et al., 2016)

Substitute Blackbox Attack (SBA)

1. Train a substitute model on a dataset **labelled by the victim (i.e., blackbox model)**
2. Attack the substitute model using any whitebox methods
3. Validate that the adversarial examples fool the substitute model
4. Use the adversarial examples to fool the blackbox model



Source: https://medium.com/element-ai-research-lab/tricking-a-machine-into-thinking-youre-milla-jovovich-b19bf322d55c

This categorization is only based on methods covered in this presentation and does not encompasses every single attack or defense method that exist as of today.

Attack & Defense Methods

Attack strategies:

Whitebox:
- Direct gradient step(s): FGSM, BIM, R+FGSM, DAG
- Gradient-based greedy algorithm: JSMA
- Iterative optimization: L-BFGS, C&W, UAP, Adversarial Eyeglasses, RP_2, EOT
- Parameterized optimization: ATN

Blackbox:
- Decision boundary approximation: SBA
- Evolutionary algorithm: One Pixel Attack
- Finite difference method: ZOO

Defense strategies:

Data augmentation:
- Adversarial training
- Ensemble adversarial training
- PGD adversarial training

Gradient regularization:
- DCN
- Defensive distillation

Input manipulation:
- PixelDefend

Detection methods:
- Adversary detector networks
- Feature squeezing

Details will be skipped since knowledge in optimization is needed (e.g., ADAM and coordinate descent). This is just to provide an example on how to get creative with existing methods.

Zeroth Order Optimization (ZOO)

- Blackbox attack via **finite difference approximation**

$$\hat{g}_i := \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h}$$

- ***f*** is victim model, $\boldsymbol{e}_i$ is a vector where only the ***i***-th element is 1, $\boldsymbol{x}_i$ is the ***i***-th element of input ***x***
- What is the main disadvantage of using finite difference?

Algorithm 2 ZOO-ADAM: Zeroth Order Stochastic Coordinate Descent with Coordinate-wise ADAM

Require: Step size η, ADAM states $M \in \mathbb{R}^p, v \in \mathbb{R}^p, T \in \mathbb{Z}^p$, ADAM hyper-parameters $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

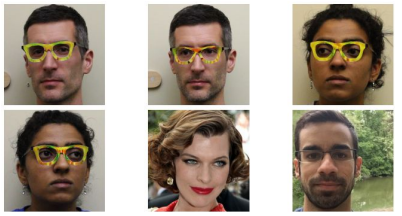
$M \leftarrow \mathbf{0}, v \leftarrow \mathbf{0}, T \leftarrow \mathbf{0}$
while not converged **do**
  Randomly pick a coordinate $i \in \{1, \cdots, p\}$
  Estimate $\hat{g}_i$ using (6)
  $T_i \leftarrow T_i + 1$
  $M_i \leftarrow \beta_1 M_i + (1 - \beta_1)\hat{g}_i, \quad v_i \leftarrow \beta_2 v_i + (1 - \beta_2)\hat{g}_i^2$
  $\hat{M}_i = M_i / (1 - \beta_1^{T_i}), \quad \hat{v}_i = v_i / (1 - \beta_2^{T_i})$
  $\delta^* = -\eta \frac{\hat{M}_i}{\sqrt{\hat{v}_i + \epsilon}}$
  Update $\mathbf{x}_i \leftarrow \mathbf{x}_i + \delta^*$
end while

ZOO: Zeroth Order Optimization based Black-box Attacks to Deep Neural Networks without Training Substitute Models (Chen et al., 2017)

Some real world examples

Details will be skipped, this is just to provide an example on how to get creative with existing methods.

Adversarial Eyeglasses



Example of adversarial eyeglasses (Sharif et al., 2016). Top row depicts an input given to a model, while bottom row depicts the person from the target misclassification class.

Accessorize to a Crime: Real and Stealthy Attacks on State-of-the-Art Face Recognition (Sharif et al., 2016)

Details will be skipped, this is just to provide an example on how to get creative with existing methods.

Adversarial Road Signs

- Similar attack method to the adversarial eyeglasses with different masks
- Perceptually different, but **inconspicuous**

"STOP" signs misclassified as a "speed limit" sign (Eykholt et al., 2017).

Robust Physical-World Attacks on Deep Learning Models (Eykholt et al., 2017)

Details will be skipped, this is just to provide an example on how to get creative with existing methods.

Adversarial Turtle

3D printed adversarial turtle misclassified as rifle (Athalye et al., 2018).

Synthesizing Robust Adversarial Examples (Athalye et al., 2018)

Agenda

Intro

Adversarial Attacks

Adversarial Defenses

Takeaways

This categorization is only based on methods covered in this presentation and does not encompasses every single attack or defense method that exist as of today.

Attack & Defense Methods

Attack strategies:

Whitebox:
- Direct gradient step(s): FGSM, BIM, R+FGSM, DAG
- Gradient-based greedy algorithm: JSMA
- Iterative optimization: L-BFGS, C&W, UAP, Adversarial Eyeglasses, RP_2, EOT
- Parameterized optimization: ATN

Blackbox:
- Decision boundary approximation: SBA
- Evolutionary algorithm: One Pixel Attack
- Finite difference method: ZOO

Defense strategies:

Data augmentation:
- Adversarial training
- Ensemble adversarial training
- PGD adversarial training

Gradient regularization:
- DCN
- Defensive distillation

Input manipulation:
- PixelDefend

Detection methods:
- Adversary detector networks
- Feature squeezing

Adversarial Training

- Include adversarial examples as part of the training set
- If using FGSM adversaries, training loss becomes:

$$\tilde{J}(\boldsymbol{\theta}, \boldsymbol{x}, y) = \alpha J(\boldsymbol{\theta}, \boldsymbol{x}, y) + (1 - \alpha) J(\boldsymbol{\theta}, \boldsymbol{x} + \epsilon \text{sign}\left(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y)\right))$$

- In other words, new adversarial examples are generated **per training iteration** based on the state of the model at that iteration
- **NOT** the same as the Generative Adversarial Nets (GAN)
- Robust to adversaries included during adversarial training

Note: $J(\cdot)$ denotes the classification loss (e.g. cross-entropy), $\boldsymbol{\theta}$ denotes the model's parameter, α denotes a constant that weigh the importance on classifying normal versus adversarial examples where $\alpha \in [0, 1]$.

Explaining and Harnessing Adversarial Examples (Goodfellow et al., 2015)

Adversarial Training

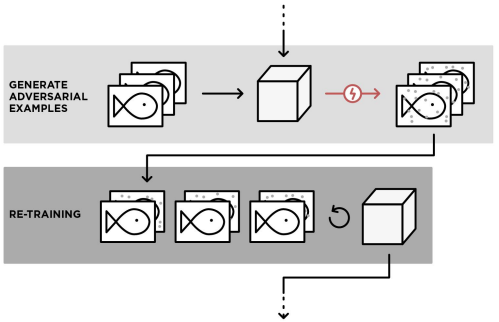


Illustration of adversarial training.

Source: https://medium.com/element-ai-research-lab/securing-machine-learning-models-against-adversarial-attacks-b6cd5d2be8e2

BEWARE: Gradient Masking

- When a defense method prevents a model to reveal meaningful gradients
- At the origin, local gradient towards ϵ_1 is larger compared to ϵ_2 direction
- But loss actually higher in ϵ_2 direction for higher ϵ values
- Many directions orthogonal to ϵ_1 with higher loss at larger ϵ
- Often **unintentional**

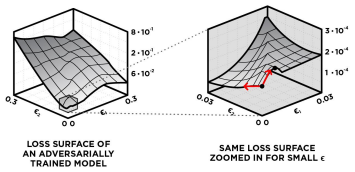


Illustration of loss surface of a model trained with FGSM adversarial training (adapted from Tramèr et al., 2018). Here, ϵ_1 is the direction given by calculating ***dL/dx***, and ϵ_2 is direction orthogonal to ϵ_1.

Ensemble Adversarial Training: Attacks and Defenses (Tramèr et al., 2018)
Source: https://medium.com/element-ai-research-lab/securing-machine-learning-models-against-adversarial-attacks-b6cd5d2be8e2

REVISIT: R+FGSM

$$x^{\text{adv}} = x' + (\varepsilon - \alpha) \cdot \text{sign}\left(\nabla_{x'} J(x', y_{\text{true}})\right), \quad \text{where} \quad x' = x + \alpha \cdot \text{sign}(\mathcal{N}(\mathbf{0}^d, \mathbf{I}^d))$$

(2) (1)

- R+FGSM: add small random perturbation **(1) before** calculating the gradient **(2)** (i.e. random start)
- Can circumvent naive implementation of FGSM adversarial training

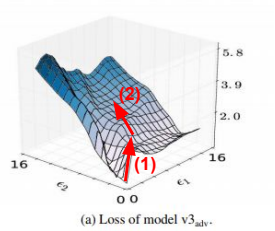


Illustration of how R+FGSM circumvents FGSM adversarial training.

Ensemble Adversarial Training: Attacks and Defenses (Tramèr et al., 2018)

PGD Adversarial Training

- Adversarial training from robust optimization perspective

$$\min_{\theta} \rho(\theta), \quad \text{where} \quad \rho(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\max_{\delta\in\mathcal{S}} L(\theta, x+\delta, y)\right]$$

- In other words: find a set of parameter θ that **minimizes** the loss in the **worst-case** scenario (Minimax formulation)
- Empirically showed that adversaries generated using R+BIM, which they called Projected Gradient Descent (PGD) method, are the worst-case adversaries
- In practice: perform adversarial training **only** on PGD adversaries

Towards Deep Learning Models Resistant to Adversarial Attacks (Madry et al., 2018)

Agenda

This categorization is only based on methods covered in this presentation and does not encompasses every single attack or defense method that exist as of today.

Attack & Defense Methods

Attack strategies:

Whitebox:
- Direct gradient step(s): FGSM, BIM, R+FGSM, DAG
- Gradient-based greedy algorithm: JSMA
- Iterative optimization: L-BFGS, UAP, C&W, Adversarial Eyeglasses, RP_2, EOT
- Parameterized optimization: ATN

Blackbox:
- Decision boundary approximation: SBA
- Evolutionary algorithm: One Pixel Attack
- Finite difference method: ZOO

Defense strategies:

Data augmentation:
- Adversarial training
- Ensemble adversarial training
- PGD adversarial training

Gradient regularization:
- DCN
- Defensive distillation

Input manipulation:
- PixelDefend

Detection methods:
- Adversary detector networks
- Feature squeezing

Recall our goal:

Minimally modify inputs to maximize some loss function

Gradient-based Attack Methods

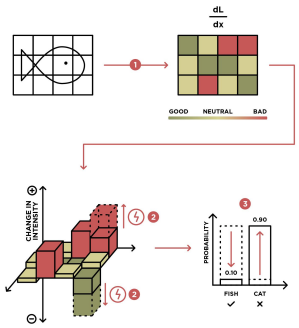


Illustration of gradient-based attacks.

Source: https://medium.com/element-ai-research-lab/tricking-a-machine-into-thinking-youre-milla-jovovich-b19bf322d55c

Takeaway Messages

- Arms race between adversarial attacks and defenses: attackers are winning
- Beware of **gradient masking**, often it is unintentional and may give false robustness
  - 7 out of 9 defenses accepted to ICLR 2018 were successfully attacked just few days after acceptance decision date (Athalye et al., 2018)
- Although most attacks focus on virtual world adversaries, there are works that aim to generate adversarial examples in the physical world (e.g. the adversarial eyeglasses, adversarial turtle, etc.)
- The field is very empirical, need more works that can provide guarantee on adversarial robustness (e.g., by providing upper bound of a proposed defense method)

Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples (Athalye et al., 2018)

Thank You. Questions?

My blog on adversarial examples:

1. Tricking a Machine Learning into Thinking You're Milla Jovovich (https://medium.com/element-ai-research-lab/tricking-a-machine-into-thinking-youre-milla-jovovich-b19bf322d55c)

2. Securing Machine Learning Models Against Adversarial Attacks (https://medium.com/element-ai-research-lab/securing-machine-learning-models-against-adversarial-attacks-b6cd5d2be8e2)

Intrinsic Plasticity and Batch Normalisation

Nolan Peter Shaw

2019-03-15

Agenda

- Introduce Intrinsic Plasticity (IP)
- Discuss the biological and computational benefits of IP

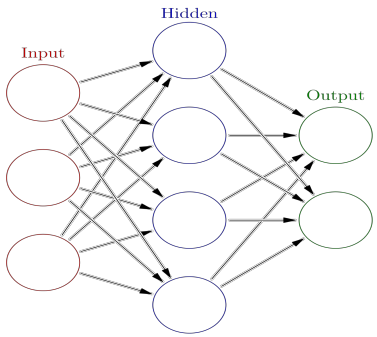
- Introduce batch normalisation (BN)
- Outline BN implementation

- Demonstrate the relation between IP and BN

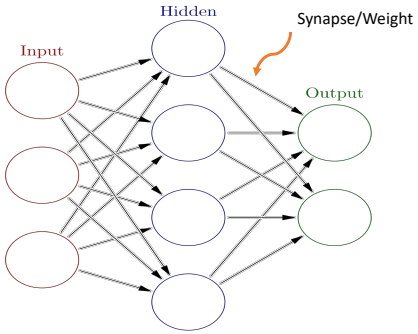
Section I: Intrinsic Plasticity

Computational Neuroscience

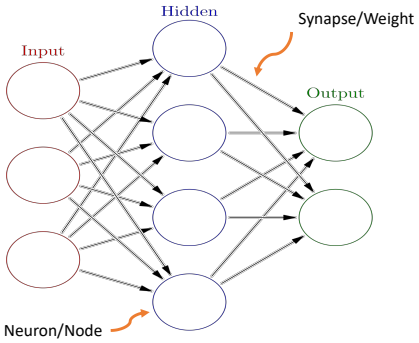
Synaptic vs Intrinsic Plasticity in the Brain



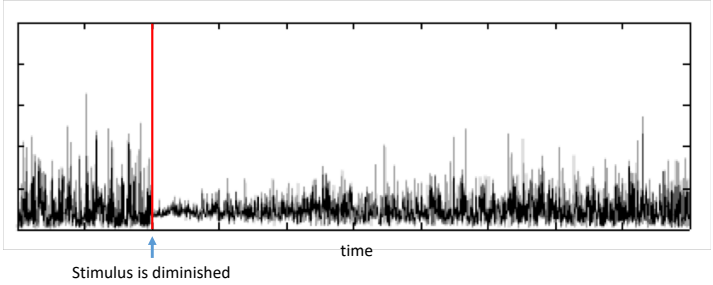
Synaptic vs Intrinsic Plasticity in the Brain



Synaptic vs Intrinsic Plasticity in the Brain

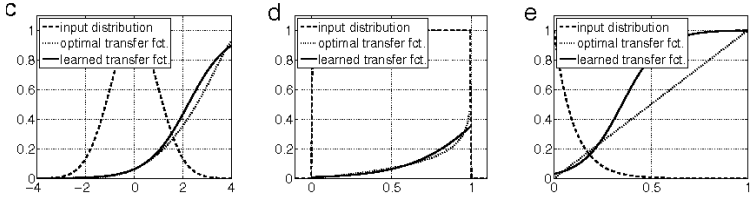


Intrinsic Plasticity



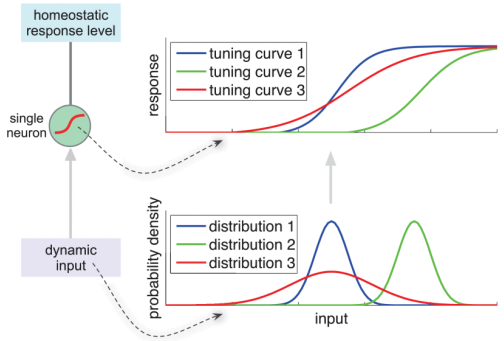
Biological Benefits

- Human brain consumes calories
- Cost of a 0/1 in an ANN is identical



Triesch, Jochen. (2005). A Gradient Rule for the Plasticity of a Neuron's Intrinsic Excitability. Artificial Neural Networks: Biological Inspirations ICANN 2005.

Computational Benefits



Bell AJ, Sejnowski TJ (November 1995). An information-maximization approach to blind separation and blind deconvolution. Neural Computation

Computational Benefits

Synaptic Plasticity

- Learn weights w.r.t. an error signal
- Minimise loss on some task

Intrinsic Plasticity

- Learn gains and biases w.r.t. local statistics
- Maximise information potential

Implementation in ANNs

Biology		**Artificial**
• Sensitivity	⟷	• Gain (Horizontal stretch)
• Threshold	⟷	• Bias (Horizontal translation)

Implementation in ANNs

Original Activation Function **Becomes**

$$y = \theta(x) \quad \rightarrow \quad y = \theta(\alpha * x + k)$$

Implementation in ANNs

Original Activation Function **Becomes**

$$y = \theta(x) \longrightarrow y = \theta(\alpha * x + k)$$

Super simple (YAY!)

Implementation in ANNs

Update rules

Gain:

$$\Delta\alpha = \frac{1}{\alpha} - 2 * \mathbf{E}[\boldsymbol{xy}]$$

Bias:

$$\Delta k = -2 * \mathbf{E}[\boldsymbol{y}]$$

- Note that these update rules are still being studied and that there may be update rules that are better suited to learning

Issues

- Unstable
- $\mathbf{E}[\boldsymbol{uy}]$ may be ill-suited for adjusting the sensitivity/gain
- May homogenise inputs too much
- Competes with error-based learning of synaptic weights

Section II: Batch Normalisation

Machine Learning

The Problem

- The “shape” of inputs to a layer may be radically different from one input to the next
- This slows down learning as hidden layers are required to learn representations and distributions as well as perform computation

The Problem

- The “shape” of inputs to a layer may be radically different from one input to the next
- This slows down learning as hidden layers are required to learn representations and distributions as well as perform computation

The Solution

- Normalise all inputs w.r.t. their distribution
  - (infeasible to do for an entire dataset so treat each batch as a sample of the population)
- De-normalise w.r.t. error
  - (prevents homogeneity and preserves computational properties of neurons)

Visualising Batch Normalisation

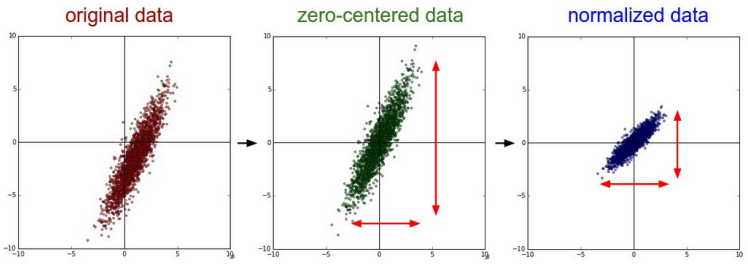


Image courtesy of: https://zaffnet.github.io/batch-normalization

Implementing Batch Normalisation

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

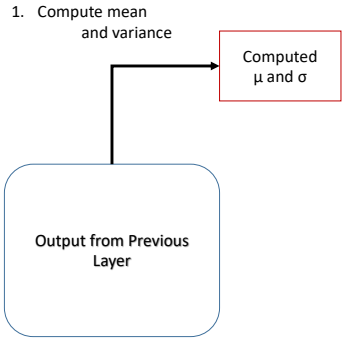
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

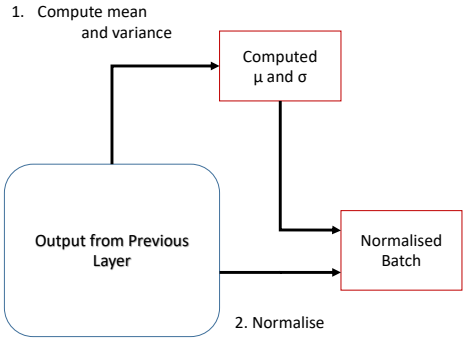
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

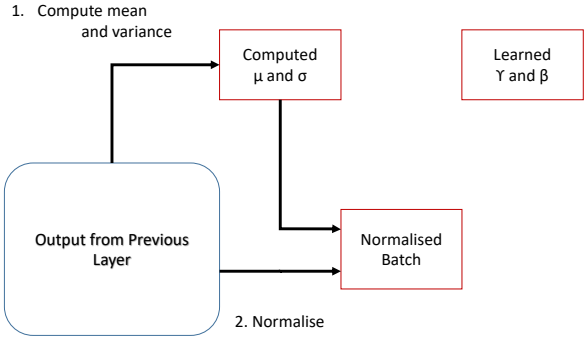
Step-by-step Walkthrough



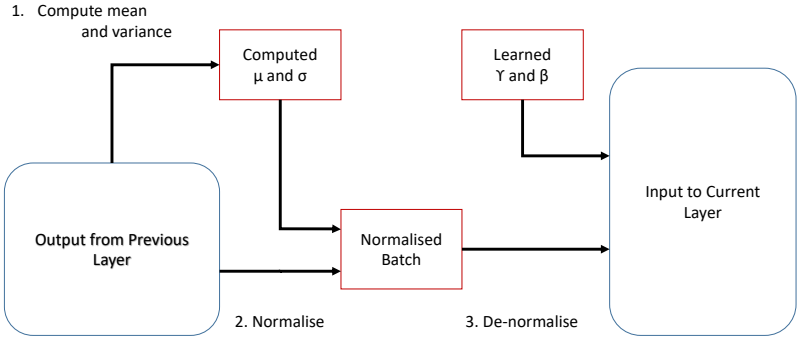
Step-by-step Walkthrough



Step-by-step Walkthrough



Step-by-step Walkthrough



Section III: Unifying IP and BN

(or: How I Stopped Caring About Big Data and Learned to Love the Brain)

Equivalence of the two models

Intrinsic Plasticity:

$$y = \theta(\alpha * x + k)$$

Equivalence of the two models

Intrinsic Plasticity:

$$y = \theta(\gamma\ * (\alpha * x + k) + \beta)$$

Equivalence of the two models

Intrinsic Plasticity:

$$y = \theta(\gamma\ * (\alpha * x + k) + \beta)$$

Batch Normalisation:

$$y = \theta(\gamma\ * \left(\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\right) + \beta)$$

Equivalence of the two models

Intrinsic Plasticity:

$$y = \theta(\gamma * (\alpha * x + k) + \beta)$$

Batch Normalisation:

$$y = \theta(\gamma * \left(\frac{1}{\sqrt{\sigma^2 + \epsilon}} * x + \frac{-\mu}{\sqrt{\sigma^2 + \epsilon}}\right) + \beta)$$

Equivalence of the two models

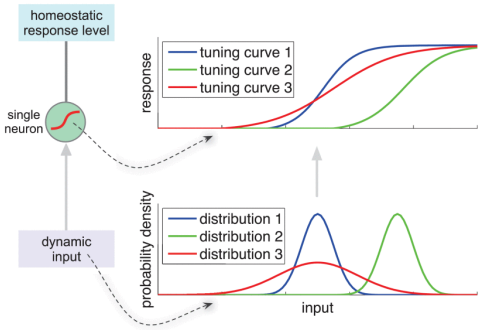
Intrinsic Plasticity:

$$y = \theta(\gamma * (\alpha * x + k) + \beta)$$

Batch Normalisation:

$$y = \theta(\gamma * \left(\frac{1}{\sqrt{\sigma^2 + \epsilon}} * x + \frac{-\mu}{\sqrt{\sigma^2 + \epsilon}}\right) + \beta)$$

Relation to the Vanishing Gradient Problem



Thank you!

Vector Embeddings

We have been using vectors to represent inputs and outputs.

eg. "2" = [0 0 1 0 ⋯] $\in \{0,1\}^{10}$

"e" = [0 0 0 0 1 0 ⋯] $\in \{0,1\}^{26}$

What about words?

Consider the set of all words encountered in a dataset. We will call this our vocabulary.

Let's order and index our vocabulary and represent words using one-hot vectors, like above. Let $word_i$ = i^{th} word in vocab.

ie. "cat" ~ $v \in W$

$W \subset \{0,1\}^{N_v} \subset \mathbb{R}^{N_v}$ where N_v is the # of words in our vocab (eg. 70 000)

Then $v_i = \begin{cases} 0 & \text{if } word_i \neq \text{"cat"} \\ 1 & \text{if } word_i = \text{"cat"} \end{cases}$

This is nice, but when we are doing Natural Language Processing (NLP), how do we handle the common situation in which different words can be used to form a similar meaning?

Example:

"CS 489 is interesting"

"CS 489 is fascinating"

We could form synonym groups, but where do we draw the line when words have similar, but not identical, meanings?

eg. content, happy, elated, ecstatic

These issues reflect the semantic relationships between words. We would like to find a different representation for each word, but one that also incorporates their semantics.

Predicting Word Pairs

We can get a lot of information from the simple fact that some words often occur together (or nearby) in sentences.

Example:

"Trump returned to Washington Sunday night, though his wife Melania Trump stayed behind in Florida."

From <http://www.cbc.ca/news/world/stormy-daniels-trump-threat-1.4594060>

"Human activity is degrading the landscape, driving species to extinction and worsening the effects of climate change"

From <http://www.cbc.ca/news/thenational/national-today-newsletter-russia-diplomats-biodiversity-1.4592950>

For the purposes of this topic, we will consider "nearby" to be within words.

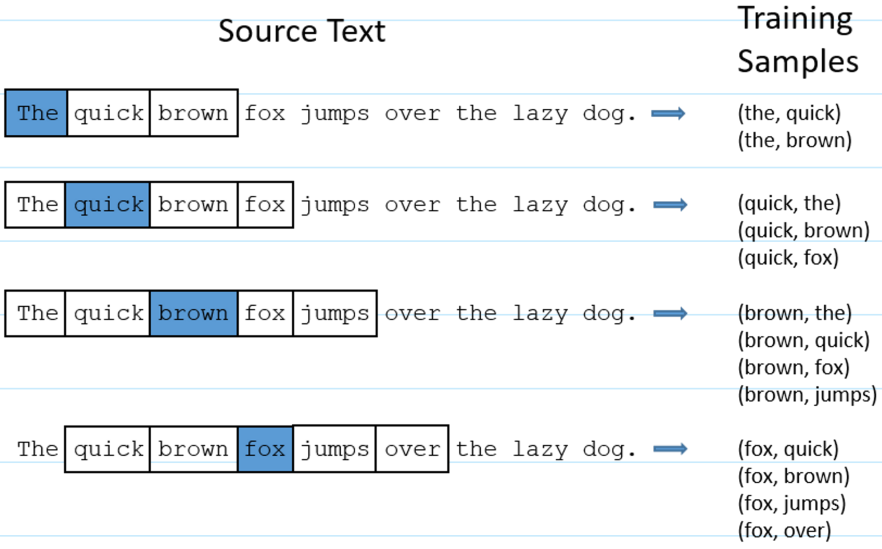
Example: d=2

"Trump returned to Washington Sunday night, though his wife Melania Trump stayed behind in Florida."

This gives us the word pairings:

(night, Washington), (night, Sunday), (night, though), (night, his)

Example:



Our approach is to try to predict these word co-occurrences using a 3-layer neural network.

- its input is a one-hot word vector, and
- its output is the probability of each word's co-occurrence.

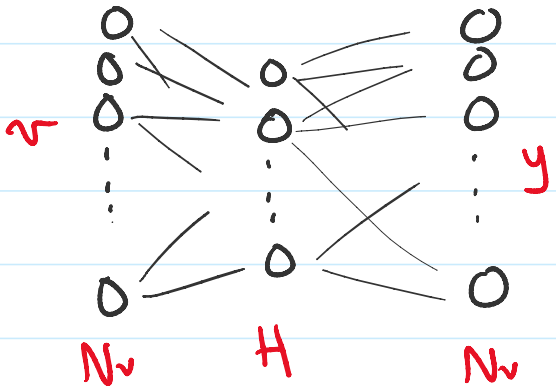
Our neural network performs

$y = f(v, \theta)$ where $v \in \mathbb{W}$

and $y = \mathbb{P}^{N_v} = \{p \in \mathbb{R}^{N_v} \mid p \text{ is a probability vector}\}$

i.e. $\sum p_i = 1$, $p_i \geq 0 \ \forall i$

Then, y_i equals the probability that word$_i$ is nearby v.



output layer uses

The hidden layer is much smaller.

This hidden-layer squeezing forces a compressed representation, requiring similar words to take on similar representations.
This is called an embedding.

word2vec

Word2vec is a popular embedding strategy for words (or phrases, or sentences). It uses additional tricks to speed up the learning.

1) Treats common phrases as new words. eg. "New York" is one word
2) Randomly ignores very common words
   eg. "the car hit the post on the curb"

Of the 56 possible word pairs, only 20 don't involve "the"

3) Negative Sampling
   Backprops only some of the negative cases

The embedding space is a relatively low-dimensional space where similar words are mapped to similar locations.

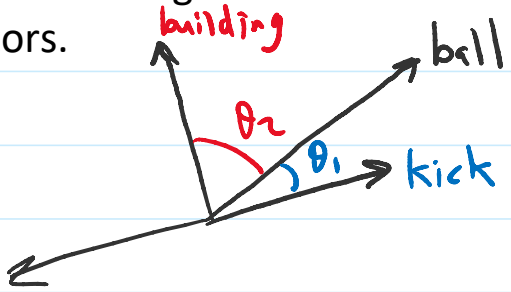
Where have we seen this before?

SOM

Why does this work?

Words with similar meaning likely co-occur with the same set of words, so the network should produce similar outputs.
∴ similar hidden-layer activation

The cosine angle is often used to measure the "distance" between two vectors.

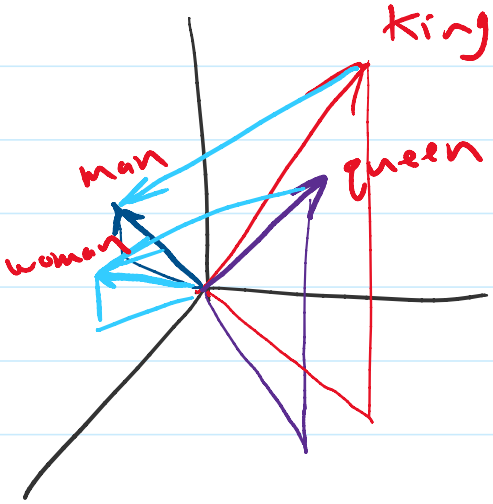


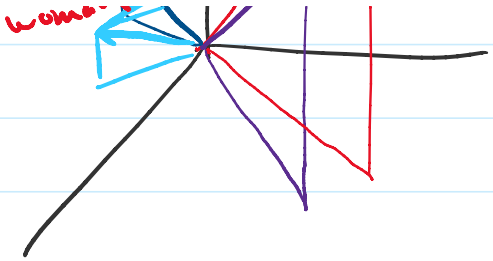
$\cos\theta_1 = 0.81$

$\cos\theta_2 = -0.05$

To some extent, you can do a sort of vector addition on these representations.

eg. king − man + woman = queen





Predictive-Coding Neural Networks

Jeff Orchard

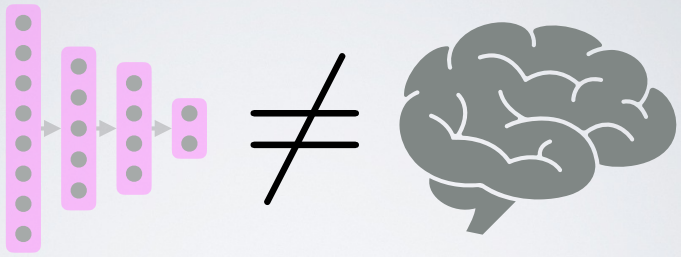
NEUROCOGNITIVE COMPUTING LAB

UNIVERSITY OF WATERLOO
FACULTY OF MATHEMATICS
DAVID R. CHERITON SCHOOL OF COMPUTER SCIENCE

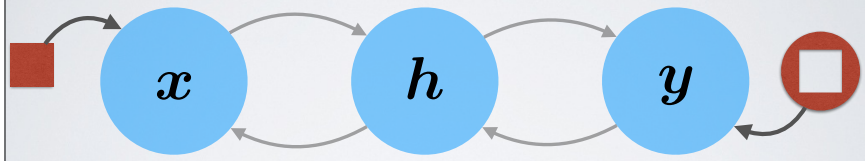
CTN
CENTRE FOR THEORETICAL NEUROSCIENCE
ctn.uwaterloo.ca

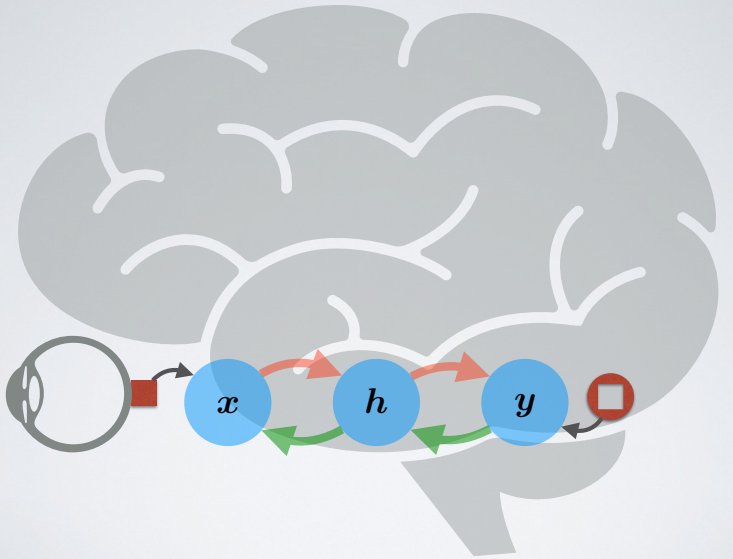
Bogacz, R.
"A tutorial on the free-energy framework for modelling perception and learning"
Journal of Mathematical Psychology, 76, 198–211, 2017.

Whittington, J. C. R., & Bogacz, R.
"An Approximation of the Error Backpropagation Algorithm in a Predictive Coding Network with Local Hebbian Synaptic Plasticity"
Neural Computation, 29(5), 1229–1262, 2017.

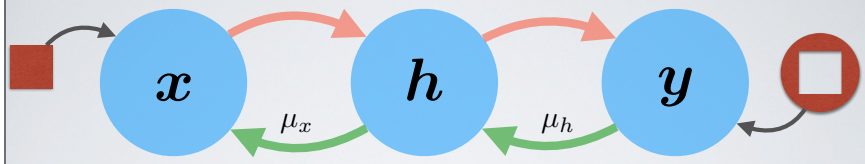


Predictive Estimator Network





Shipp, S. "Neural Elements for Predictive Coding". *Frontiers in Psychology*, 7, article 1792, 2016

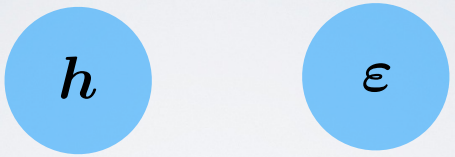


$$P(y|x) = P(y|h)\,P(h|x)$$

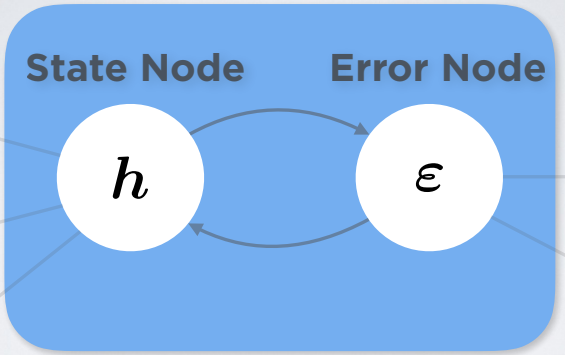
$$F = \ln\Big[P(y|h)P(h|x)\Big]$$

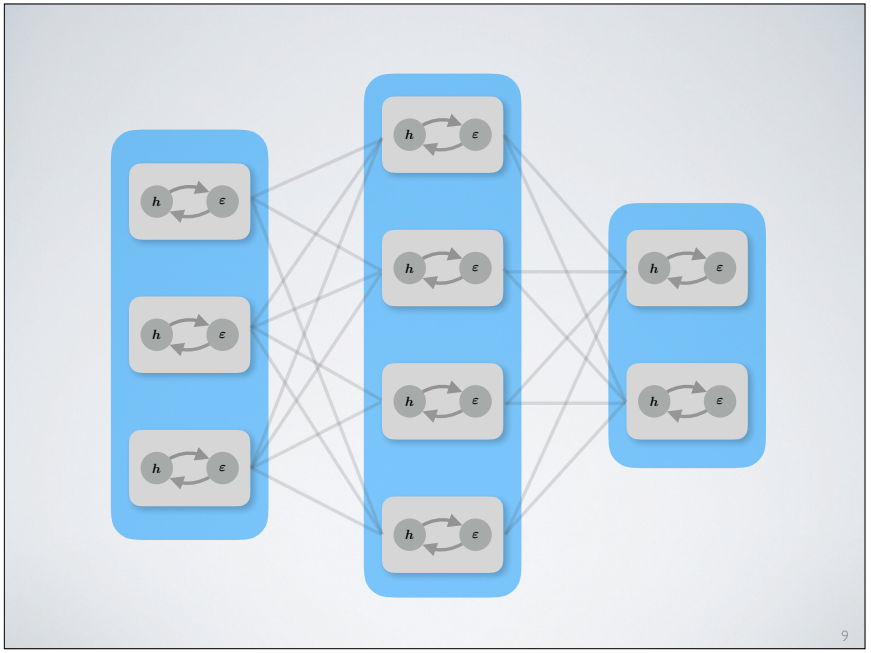
$$F = \ln P(y|h) + \ln P(h|x)$$

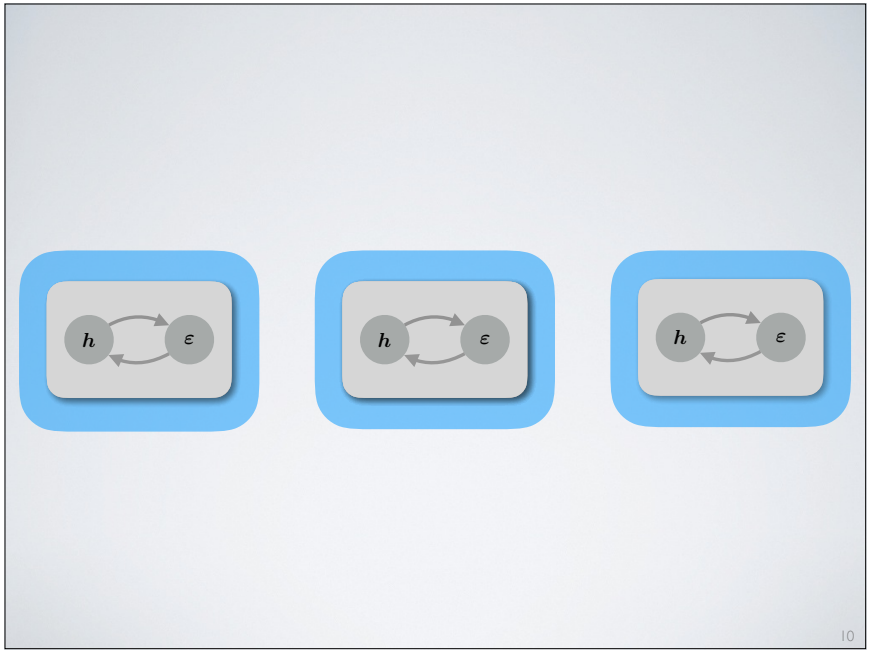
$$F \approx -\frac{(h-\mu_h)^2}{2\Sigma_h} - \frac{(x-\mu_x)^2}{2\Sigma_x}$$

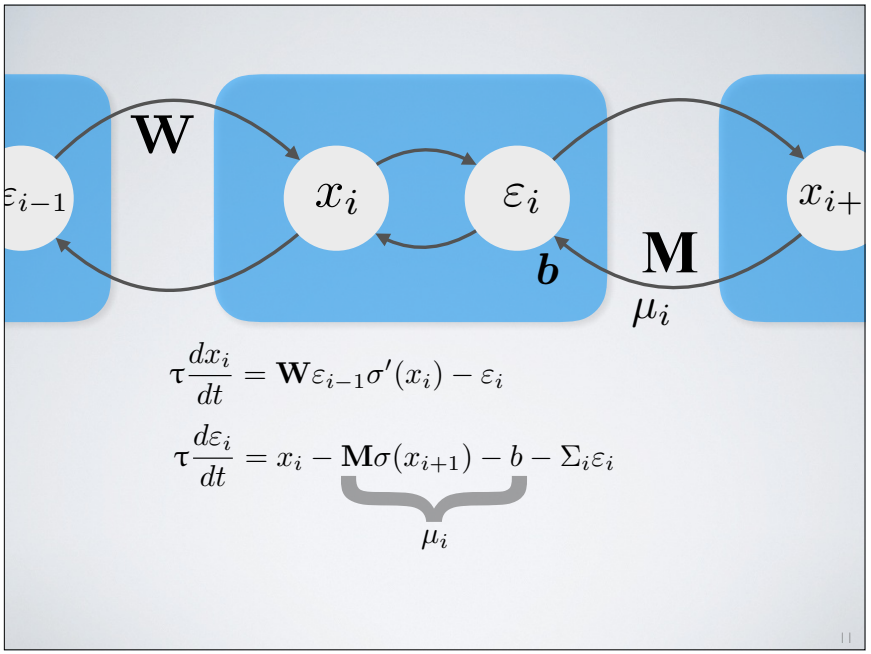


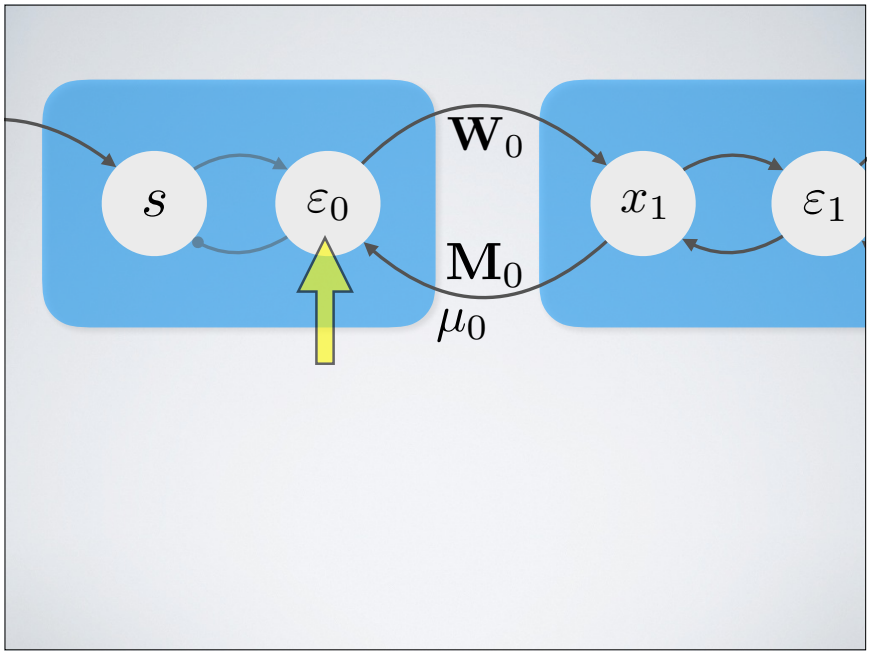
Predictive Esimator (PE)

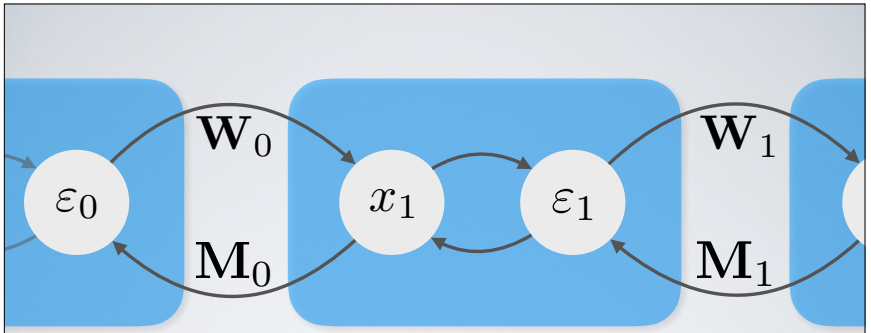



h
ε
9


h
ε
10


W
ε_{i-1}
x_i
ε_i
x_{i+}
b
M
μ_i
$\tau \frac{dx_i}{dt} = \mathbf{W}\varepsilon_{i-1}\sigma'(x_i) - \varepsilon_i$
$\tau \frac{d\varepsilon_i}{dt} = x_i - \mathbf{M}\sigma(x_{i+1}) - b - \Sigma_i \varepsilon_i$
μ_i
11


s
ε_0
$\mathbf{W}_0$
$\mathbf{M}_0$
μ_0
x_1
ε_1

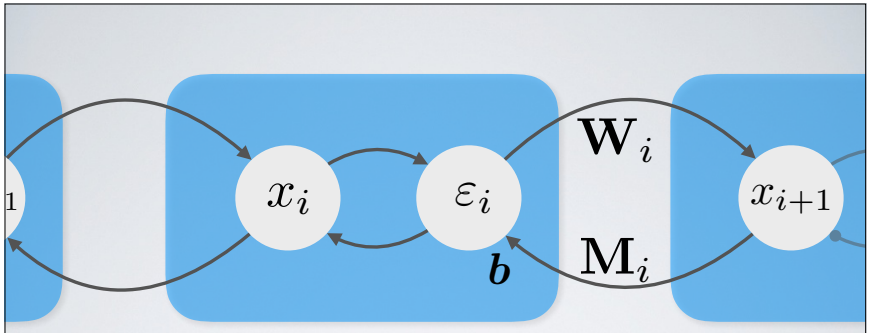


$$\tau \frac{dx_1}{dt} = \mathbf{W}_0 \varepsilon_0 \sigma'(x_1) - \varepsilon_1$$

$$\varepsilon_1 = \mathbf{W}_0 \varepsilon_0 \sigma'(x_1)$$ Steady State

$$\frac{\partial E}{\partial \vec{\alpha}} = \mathrm{M}^{\mathrm{T}} \frac{\partial E}{\partial \vec{\beta}} \sigma'(\vec{\alpha})$$

Yazdani, M. "Linear Backprop in non-linear networks". *NeurIPS, 2018*.

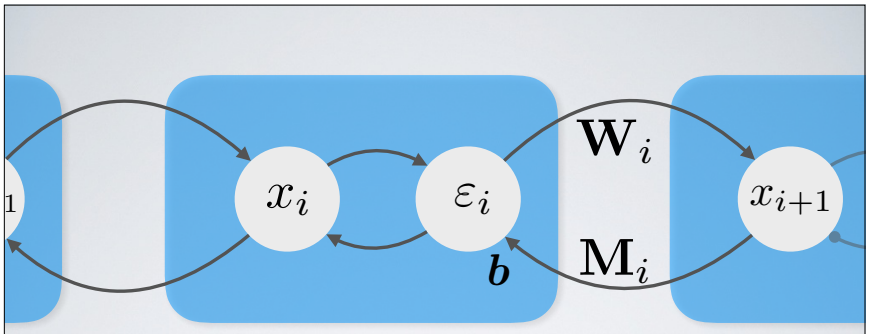


Learning Rule

$$\frac{\partial F}{\partial \mathbf{M}_i} = \sigma(x_{i+1}) \otimes \varepsilon_i$$

$$\frac{\partial F}{\partial \mathbf{W}_i} = \varepsilon_i \otimes \sigma(x_{i+1})$$

14



$$\tau \frac{dx_i}{dt} = \mathbf{W}_{i-1} \varepsilon_{i-1} - \varepsilon_i$$

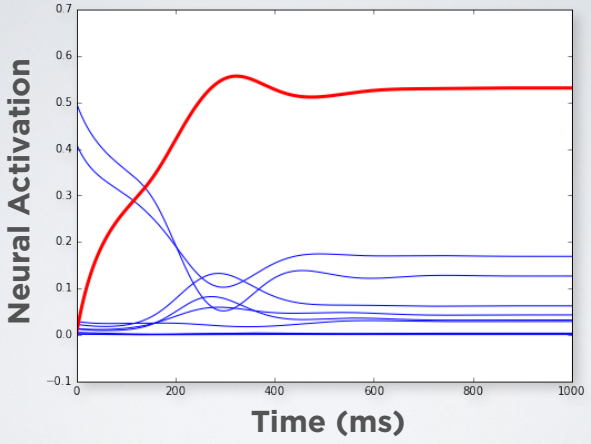
$$\tau \frac{d\varepsilon_i}{dt} = x_i - \mathbf{M}_i \sigma(x_{i+1}) - b_i - \Sigma_i \varepsilon_i$$

$$\gamma \frac{d\mathbf{M}_i}{\partial t} = \sigma(x_{i+1}) \otimes \varepsilon_i$$

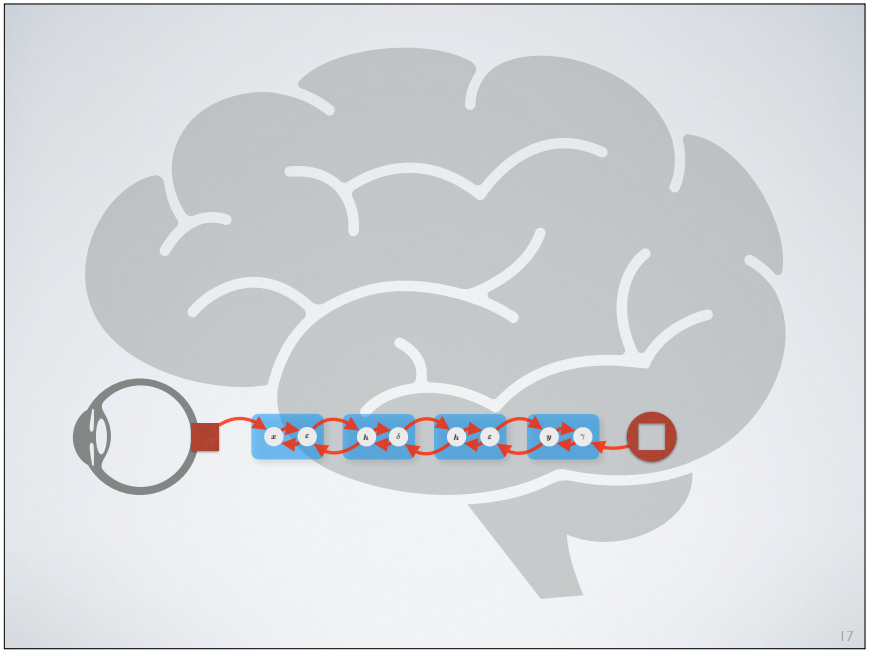
$$\gamma \frac{d\mathbf{W}_i}{dt} = \varepsilon_i \otimes \sigma(x_{i+1})$$

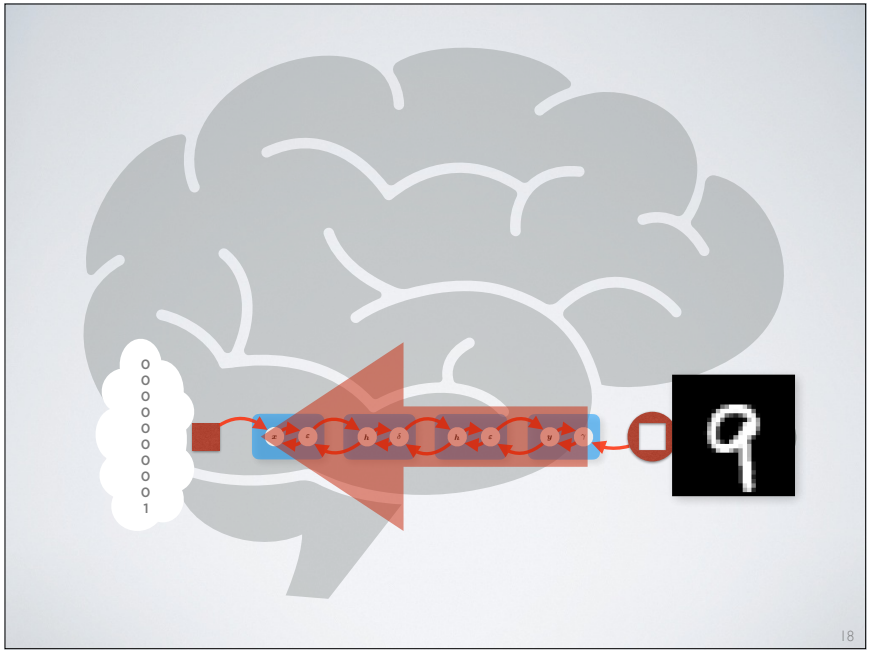
15

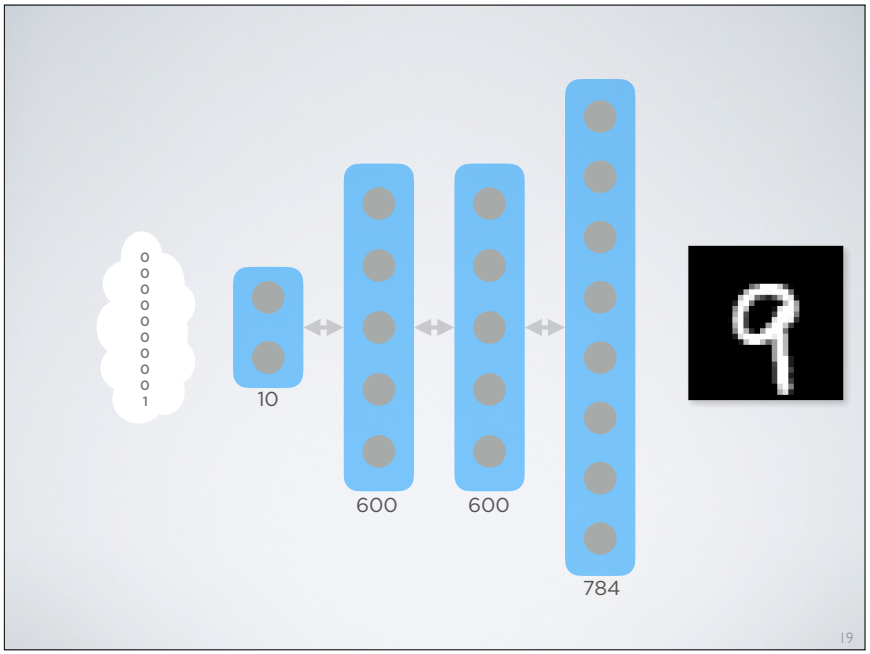
Solve numerically with time steps of 1 ms



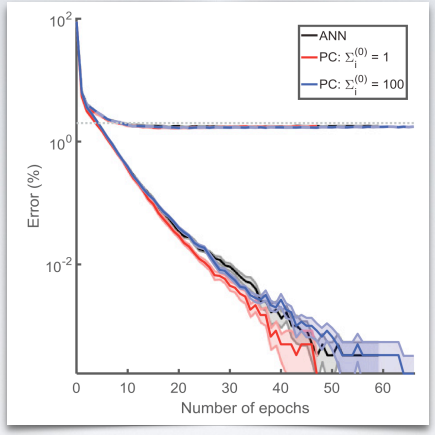
16



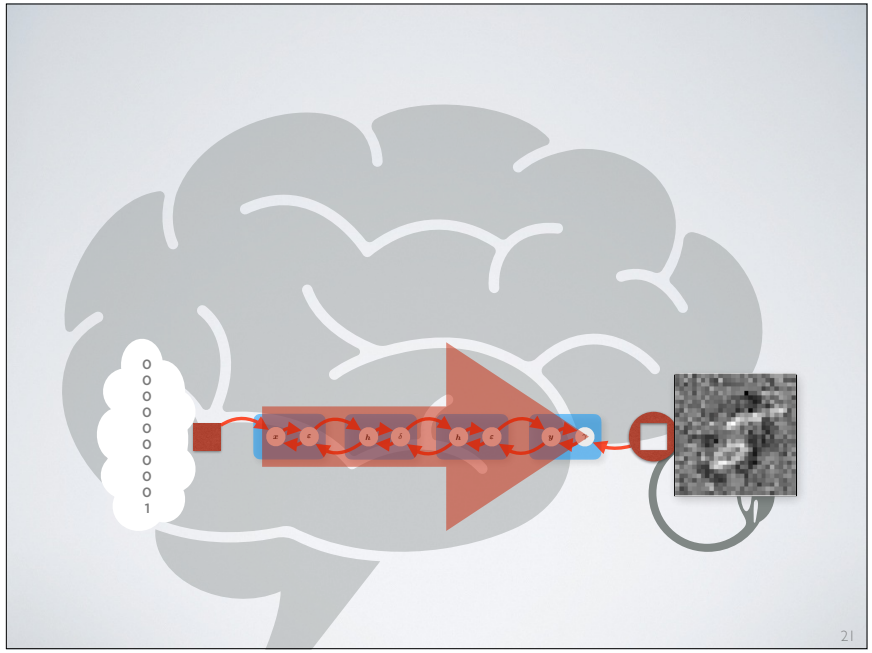




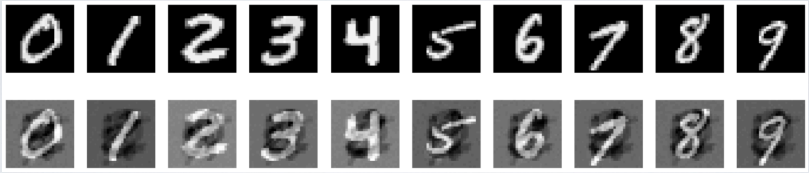
MNIST Classification



Whittington, J. C. R., & Bogacz, R. "An Approximation of the Error Backpropagation Algorithm in a Predictive Coding Network with Local Hebbian Synaptic Plasticity", *Neural Computation*, 29(5), 1229–1262, 2017.



10 MNIST Digits



1000 MNIST Digits

MNIST is not invertible