richardwu.ca

CS 489/689 Course Notes

Advanced Topics in CS (Neural Networks)

Jeff Orchard • Winter 2019 • University of Waterloo

Last Revision: April 15, 2019

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. These notes are my interpretation and transcription of the content covered in lectures. The instructor has not verified or confirmed the accuracy of these notes, and any discrepancies, misunderstandings, typos, etc. as these notes relate to course's content is not the responsibility of the instructor. If you spot any errors or would like to contribute, please contact me directly.

1 January 7, 2019

1.1 Simulating neurons and the Hodgkin-Huxley model

To construct neural networks we must first simulate how a biological neuron works.
Ions (positively and negatively charged molecules with excess protons and electrons, respectively) exists outside and inside of a cell and may be moved across the cell membrane.
There exists sodium and potassium **channels** which permits Na+ and K+ ions to move across the cell membrane, respectively. K+ channels move K+ ions out of the cell whereas Na+ channels move Na+ ions into the cell.
Sodium-potassium **pumps** exchange 3 Na+ inside the cell for 2 K+ ions outside the cell. This in effect creates a negative charge inside the cell.

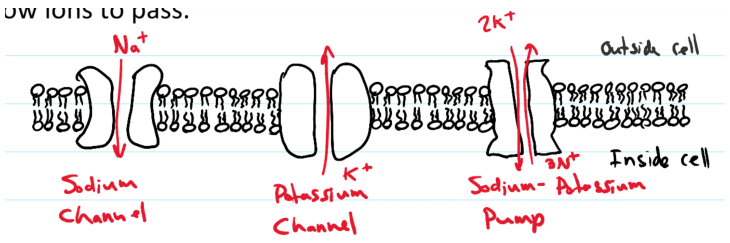


Figure 1.1: Cell membrane with Na+/K+ channels and a sodium-potassium pump. Ions move across the membrane via the channels and pump.

The difference in charge across the membrane induces a voltage difference called the **membrane potential**.
The **action potential** is a spike of electrical activity in neurons. This electrical burst travels along the neuron's **axon** to its **synapse** where it passes signals ot other neurons.
The **Hodgkin-Huxley** model describes how the action potential is effected. Note that both Na+ and K+ ion channels are voltage-dependent: Na+ and K+ move according to the membrane potential as the channels open and close 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

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

where $t_n(V)$ is the time constant and $n_\infty(V)$ is the equilibrium solution constant, which are empirically calculated (they're both functions of V however).
Note that each K+ channel is controlled by four gates wherein the probability of one gate being open is n, hence the probability of all gates being open is n^4.

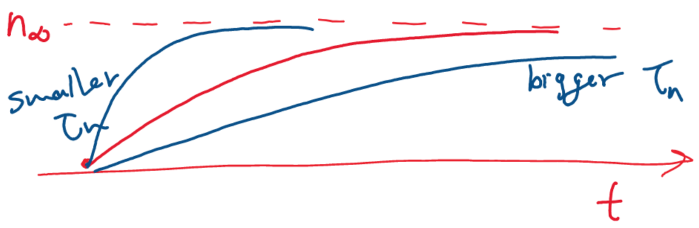
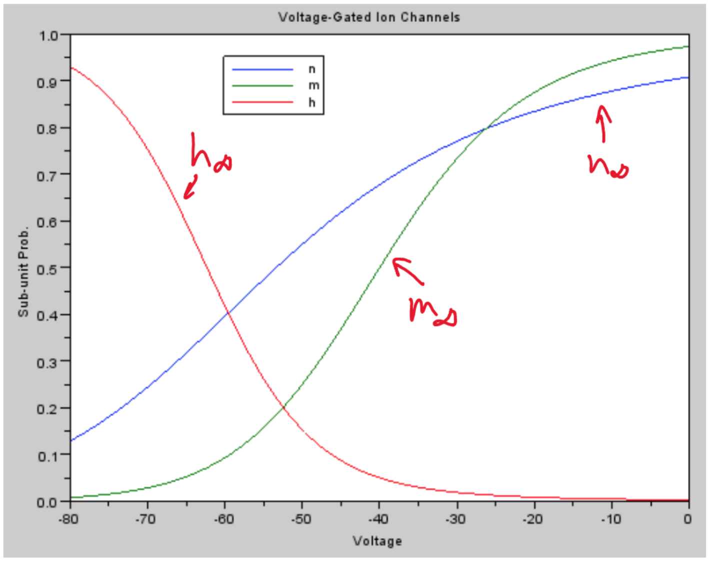


Figure 1.2: n in fraction of K+ channels open over time (for a fixed V). The blue graphs correspond to larger and smaller time constants t_n.

The fraction of Na+ channels that are open is m^3h, where (similar to above)

$$\frac{dm}{dt} = \frac{1}{t_m(V)}(m_\infty(V) - m)$$
$$\frac{dh}{dt} = \frac{1}{t_h(V)}(h_\infty(V) - h)$$

similar to above, we can interpret this as the Na+ channel is controlled by three gates with probability m being open and one gate with probability h being open.
If we measure empirically the equlibrium solutions for each of n, m, h over various voltage, we get logistic-like curves



We can thus express the membrane potential as a differential equation in terms of the fraction of K+ and Na+ channels open:

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

where each term corresponds to:

J_{in} input current (from other neurons)

$g_L(V - V_L)$ current from "leakiness"

$g_{Na}m^3h(V - V_{Na})$ current from Na+ channels

$g_K n^4(V - V_K)$ current from K+ channels

each g_X term corresponds to the max conductance for each of the sources, and each V_X term corresponds to the zero-current potential for each source. C corresponds to the capacitance of the neuron.
If we solve the above DE for V with various input potential J_{in} over time, we can see that increasing the input potential will cause the voltage to spike rapidly and successively which is the **action potential**.

2 January 9, 2019

2.1 Leaky Integrate-and-Fire (LIF) model

While the HH model already simplifies a neuron to a 4-D nonlinear system, we can further simplify it. We note that the presence of the spike is the most important takeaway and the shape (due to the K+ and Na+ channels) are less important.
The **leaky integrate-and-fire (LIF) model** models only the sub-threshold membrane potential but not the spike itself. We express it as

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

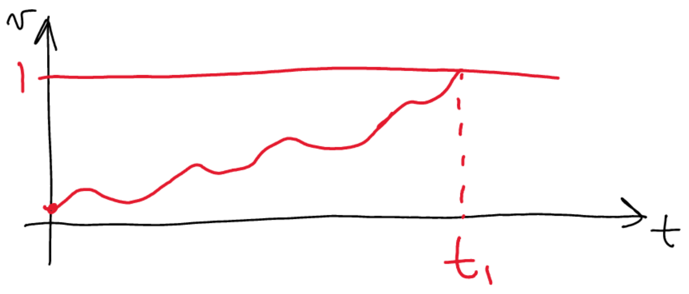
Note that $g_L = \frac{1}{R}$ where R is the resistance, thus we have

$$\begin{aligned} RC\frac{dV}{dt} &= RJ_{in} - (V - V_L) \\ \tau_m\frac{dV}{dt} &= V_{in} - (V - V_L) && \tau_m = RC \quad RJ_{in} = V(\text{Ohm's law}) \end{aligned}$$

if $V < V_{th}$ (threshold potential). If we let $v = \frac{V - V_L}{V_{th} - V_L}$ for $v < 1$, then we have

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

(note that unlike HH, our simplified time constant τ_m is not a function of v).
If we integrate the DE for a given **varying** input voltage until v reaches 1 i.e. the threshold voltage of the cell is reached at time t_1, we see the membrane potential climbs in an irregular pattern until time t_1

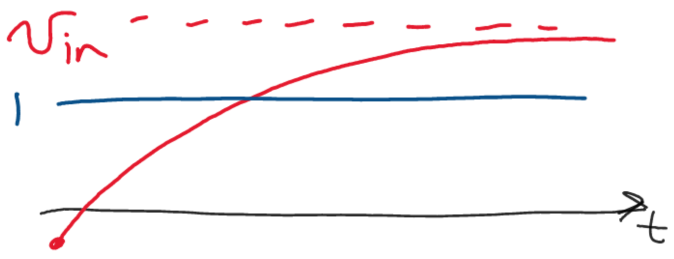


after which a spike is recorded and we reset the voltage to 0 again (after which we solve the DE for the next spike). There is a refractory period before it can spike again.
What is the firing rate if we held v_{in} constant? We need to solve for the DE analytically

Claim. We claim $v(t) = v_{in}(1 - e^{\frac{-t}{\tau_m}})$ is a solution to $\tau_m\frac{dv}{dt} = v_{in} - v$ where $v(0) = 0$.

Proof. Substitute and show that LHS = RHS. □

The graph of $v(t)$ looks like:



If $v_{in} > 1$ (our threshold for firing), then our LIF neuron will spike. To solve for the firing rate, we need to solve for the time the spike occurs (as a function of v_{in}).
The firing time t_{isi} is

$$t_{isi} = \tau_{ref} + t^*$$

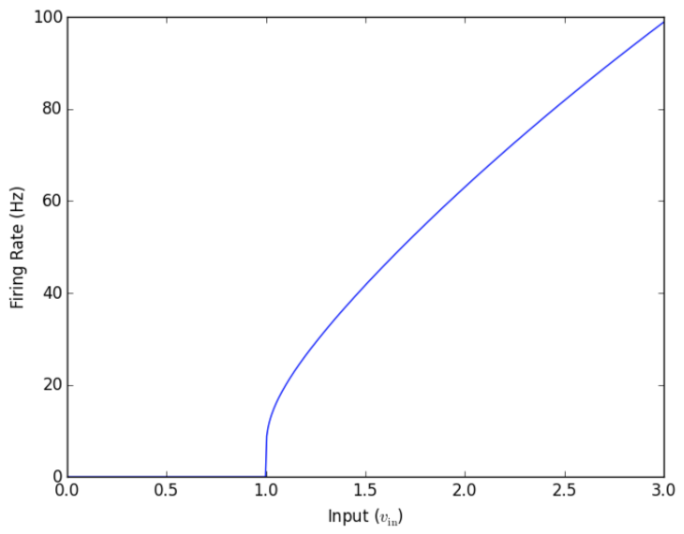
where τ_{ref} is the refractory time constant and t^* is the time for v to reach 1.
We need to find t^* where $v(t^*) = 1$. From our above solution

$$\begin{aligned} v(t^*) &= 1 = v_{in}(1 - e^{\frac{-t}{\tau_m}}) \\ \Rightarrow t^* &= -\tau_m \ln(1 - \frac{1}{v_{in}}) \qquad v_{in} > 1 \end{aligned}$$

So $t_{isi} = \tau_{ref} - \tau_m \ln(1 - \frac{1}{v_{in}})$ for $v_{in} > 1$.
Thus the steady-state firing rate for constant v_{in} is $\frac{1}{t_{isi}}$ or

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

Typical values for *cortical neurons* are $\tau_{ref} = 0.002s$ (2ms) and $\tau_m = 0.02s$ (20ms) which has the following firing rates as a function of v_{in}



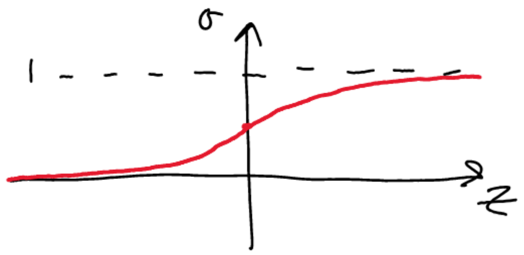
3 January 11, 2019

3.1 Sigmoid neurons

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

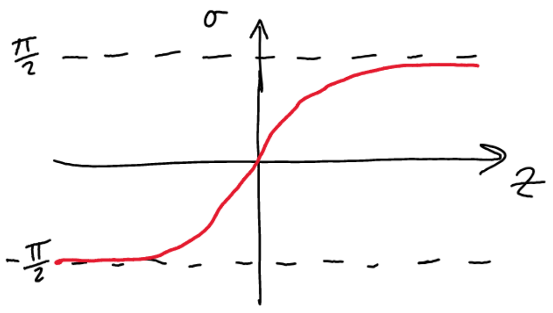
Logistic curve

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



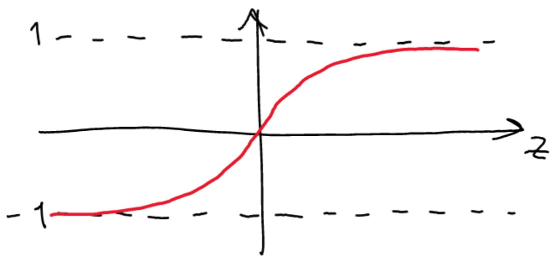
Arctan

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



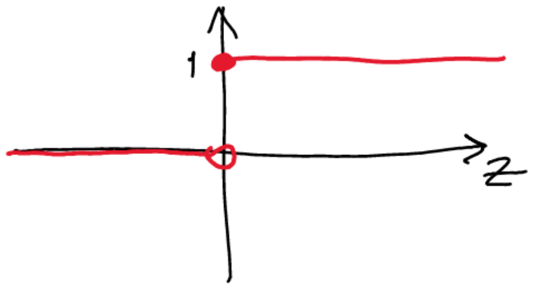
Hyperbolic tangent

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



Threshold

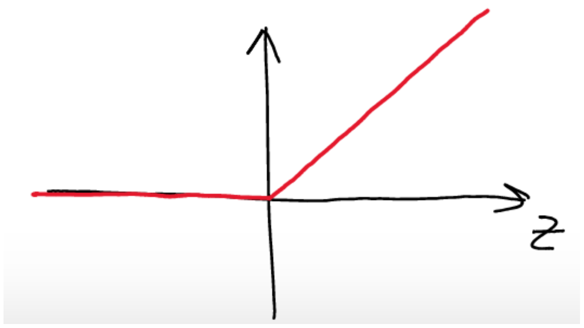
$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



3.2 Rectified Linear Unit (ReLU)

The ReLU function is simply a line that gets capped at zero below zero.

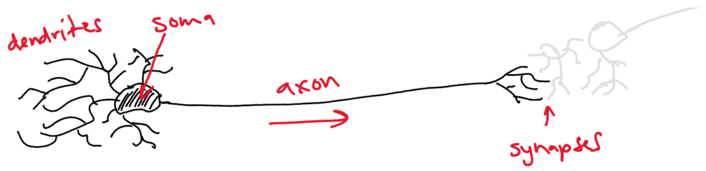
$$\text{ReLU}(z) = \max(0, z)$$



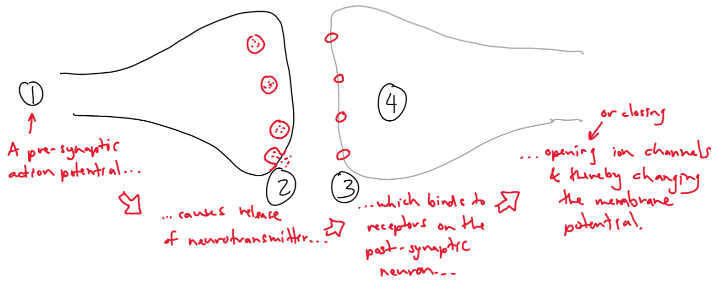
4 January 14, 2019

4.1 Synapses

We want to understand how neurons pass on information and how to model the communication channels.
The input of a neuron comes from multiple other neurons. When a neuron fires an action potential, the wave of electrical activity travels along its axon.



The junction between the axon and dendrites of two communicating neuron is called a **synapse**.
A **pre-synpatic** action potential causes the release of **neurotransmitters** into adjacent synapses which bind to receptors on the **post-synaptic** neuron. This in turn opens or closes ion channels in the post-synaptic neuron thereby changing membrane potential and causing the action potential to propagate.

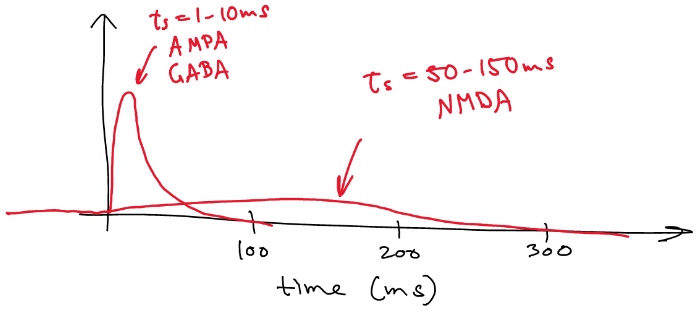

1
A pre-synaptic action potential...
2
... causes release of neurotransmitter...
3
...which binds to receptors on the post-synaptic neuron...
4
... opening ion channels & thereby changing the membrane potential.
or closing

While the action potential is very fast, the synapse process can take from 10ms to over 300ms.
If we represent the time with constant τ_s then the **post-synpatic current (PSC)** (or post-synaptic potential (PSP)) entering the post-synpatic neuron is

$$h(t) = \begin{cases} kt^n e^{\frac{-t}{\tau_s}} & \text{if } t \geq 0 \text{ (for some } n \in \mathbb{Z}^+) \\ 0 & \text{if } t < 0 \end{cases}$$

where k is a normalization constant such that $\int_0^\infty h(t)\,\mathrm{d}t = 1$ i.e. $k = \frac{1}{n!\tau_s^{n+1}}$ (we will later scale this to the appropriate current levels).

Note that when $n = 0$ we have exponential decay from time $t = 0$. When $n = 1$ (which is more realistic) we have a gamma-like distribution with $\alpha > 1$. The τ_s constant also influences the shape: as τ_s increases, the more "drawn out" the post-synpatic current



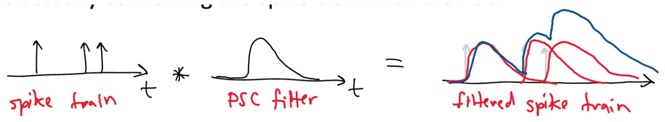
Multiple spikes (from multiple action potentials) form a **spike train** which is modelled as a sum of Dirac delta functions $a(t) = \sum_p \delta(t - t_p)$ where the Dirac delta function is defined as

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

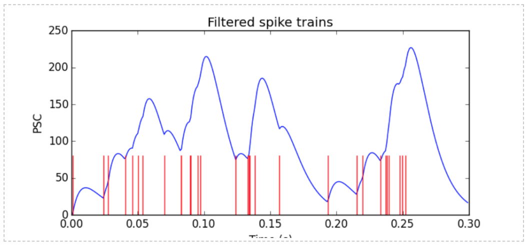
and

$$\int_{-\infty}^{\infty} \delta(t)\,\mathrm{d}t = 1$$
$$\int_{-\infty}^{\infty} f(t)\delta(s-t)\,\mathrm{d}t = f(s)$$

To combine our PSC filter/function with a spike train, we can simply take the convolution of the spike train (sum of Dirac deltas) and the PSC to form a **filtered spike train**



(demo psc)



5 January 16, 2019

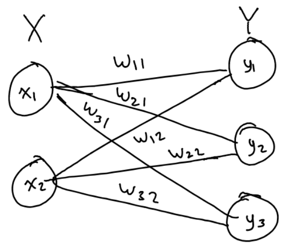
5.1 Connection weights

The total current induced on a *particular* post-synpatic neuron varies widely depending on:

- number and sizes of synapses (there may be multiple synapses between with multiple post-synpatic neurons)

- amount and type of neurotransmitter
- number and type of receptors
- etc.

We combine all these factors into a single number: the **connection weight** (which could be negative or inhibitory rather than excitatory). The total input is thus a *weighted sum* of filtered spike trains from pre-synpatic neurons. The weight from neuron A to C is denoted as w_{CA}. In general, for N pre-synpatic neurons (X) and M post-synpatic neurons (Y) we can represent the weights as an $M \times N$ **weight matrix**



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

If we represent the neuron activities in neurons X and neurons Y as vectors

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

then we can compute the linear input $\vec{z}$ to the nodes in Y

$$\vec{z} = W\vec{x} + \vec{b}$$

where $\vec{b}$ are the biases for the nodes in Y.

Aside. Biases can represent approximately constant noise from other neurons that is not a function of the upstream activities. It could also represent a baseline where some types of neurons have a propensity of firing even with little activity.

Finally after activation (spike) we have

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

Another way to introduce the bias is using $\hat{W}$ where

$$\hat{W} = \begin{bmatrix} W & \vec{b} \end{bmatrix}$$

which we can re-write

$$W\vec{x} + \vec{b} \rightarrow \hat{W} \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix}$$

5.2 Euler's method

Recall that $h(t) = kt^n e^{-\frac{t}{\tau_s}}$. For $n = 0$ we have $h(t) = \frac{1}{\tau_s} e^{-\frac{t}{\tau_s}}$ which is simply exponential decay with rate τ_s.

Claim. This also happens to be the solution to the DE

$$\tau_s \frac{dh}{dt} = -h$$

Proof. Substitute and show $LHS = RHS$. □

We essentially have an initial value problem (IVP) where $\frac{ds}{dt} = \frac{-s}{\tau_s}$ and $s(0) = \frac{1}{\tau_s}$. We can solve this DE or any first order DE numerically with **Euler's method**:

Algorithm 1 Euler's method for PSC $n = 0$

```
1: $s_0 \leftarrow s(0)$
2: $\Delta t$ is time step size
3: $t \leftarrow 0$
4: for $i = 1, 2, \ldots$ do
5:     $\frac{ds}{dt} \leftarrow \frac{-s_{i-1}}{\tau_s}$                          ▷ (slope)
6:     $s_i \leftarrow s_{i-1} + \Delta t \frac{ds}{dt} = s_{i-1}(1 - \frac{\Delta t}{\tau})$    ▷ (step)
7:     for each pre-synaptic neuron $n$ do
8:         if a spike arrived from neuron $n$ at current time $t$ then
9:             $s_i \leftarrow s_i + \frac{1}{\tau_s} w_n$          ▷ each spike contributes $s(0) \cdot w_n$
10:    $t \leftarrow t + \Delta t$
```

where after $m = \frac{T}{\Delta t}$ steps s_m represents the total post-synaptic current after time T from the spike trains of all pre-synaptic neurons.

5.3 Neural Learning

If we have a network with connection weights, how do we *adjust* the network to output what we want?
Neural learning is to formulate the problem of supervised learning as an optimization problem i.e. adjusting connection weights.
In **supervised learning** the desired output is known and we compute and the use the error to train our network.
In **unsupervised learning** the output is not supplied/known so no error signal can be generated. We aim to derive efficient representations for the statistical structure in the input. An example is transforming English words into efficient representations such as phonemes and then syllables.
In **reinforcement learning** feedback is given, but usually less often, and the error signal is less specific. An example occurs when playing chess, a player understands a play was good if they end up winning the game. They can then learn from the moves they made.

5.4 Supervised learning

Our neural network performs some mapping from an input space to an output space.
We are given training data with many examples of input/target pairs. The data is presumably from some *consistent* mapping process (the true labelling function). For example we may map handwritten digits to numbers or the XOR function:

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

where our input $(A, B) \in \{0, 1\}^2$ and our output/target is $y, t \in \{0, 1\}$.
Our goal is to adjust our connection weights to mimic this mapping and bring our output as close as possible to the target. We define the scalar function $E(y, t)$ as our **error function** which returns a smaller value as our outputs are closer to the target.
There are two common types of mappings in supervised learning:

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

An example is the simple linear regression.

Classification Outputs fall into distinct categories e.g. classifying handwritten digits into 10 digits (MNIST), or classifying images into 10 objects (CIFAR-10).

Once we have our cost function, our neural-network learning can be formulated as an **optimization problem**. Let our network be represented as $\vec{y} = f(\vec{x}; \theta)$ where θ represents the weights and biases. Then we optimize

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

That is: we find weights and biases that minimizes the expected cost between outputs and targets.

5.5 Loss functions

Given input $\vec{x}$, let $\vec{t}(\vec{x})$ be the target and $\vec{y}(\vec{x})$ be the output of our network.
There are many choices for cost functions. Here are some commonly-used ones:

Mean Squared Error (MSE)

$$\begin{aligned} E(\vec{y}, \vec{t}) &= \frac{1}{N} \|\vec{y} - \vec{t}\|_2^2 \\ &= \frac{1}{N} \sum_{i=1}^{n} \|\vec{y}_i - \vec{t}_i\|_2^2 \end{aligned}$$

where N is the number of samples (note the output of one sample can be n-dimensional).

The use of MSE as a cost function is often associated with *linear activation functions* such as ReLU since these functions have a larger output range ($[0, \infty)$).

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

Given training set $\{(x_1, t_1), \ldots, (x_N, t_N)\}$ where the true class is expressed in the target $t_i \in \{0, 1\}$. For example, if we suppose y_i is the probability that $x_i \to 1$ then

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

we can treat this as a **Bernoulli distribution** that is

$$\begin{aligned} P(x_i \to 1 \mid \theta) &= y_i && \text{i.e. } t_i = 1 \\ P(x_i \to 0 \mid \theta) &= 1 - y_i && \text{i.e. } t_i = 0 \end{aligned}$$

or

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

Therefore the likelihood of observing our dataset is

$$\begin{aligned} P(x_1, \ldots, x_n, t_1, \ldots, t_n) &= \prod_{i=1}^{N} P(x_i \to t_i) \\ &= \prod_{i=1}^{N} y_i^{t_i}(1-y_i)^{1-t_i} \end{aligned}$$

Taking the negative log-likelihood

$$\begin{aligned} -\ln P(x_1, \ldots, x_n, t_1, \ldots, t_n) &= -\ln \prod_{i=1}^{N} y_i^{t_i}(1-y_i)^{1-t_i} \\ \Rightarrow E(y,t) &= -\sum_{i=1}^{n} t_i \ln y_i + (1-t_i)\ln(1-y_i) = \mathbb{E}_t\big[-\ln y\big] \end{aligned}$$

This log-likelihood derivation is the basis for the cross-entropy cost function.

Note: cross entropy assumes output values are in the range $[0,1]$ so it works well with the *logistic activation function*.

Softmax The **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

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

so by definition $\sum_i \text{softmax}(z)_i = 1$.

Example 5.1. Suppose $z = (0.6, 3.4, -1.2, 0.05)$, then after softmax we have $y = (0.06, 0.9, 0.009, 0.031)$.

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

Example 5.2. Suppose $z = (0.6, 3.4, -1.2, 0.05)$, then after one-hot we have $y = (0, 1, 0, 0)$.

6 January 21, 2019

6.1 Perceptrons

Suppose we want to create a simple neural network to recognize certain input patterns. For example, let the output node be a simple threshold neuron ($\sigma(z) = 1$ iff $z \geq 0$, 0 otherwise), and suppose we want the output node to output 1 iff the input is $[1, 1, 0, 1]$.

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

$$[1, 1, 0, 1] \cdot [1, 1, 0, 1] = 3 \rightarrow \sigma(3) = 1$$

However other inputs like $[0, 1, 1, 0]$ results in

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

We need the *non-matching* inputs to give us a negative value so that the output node outputs 0. We could use a negative bias:

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

Question 6.1. Can we find weights and biases automatically so that our perceptron produces the correct output for a variety of inputs?

To see an approach, let's look at a 2D case. Suppose the 4 different inputs are $[0, 0], [0, 1], [1, 0]$ and $[1, 1]$, and their corresponding output should be $0, 1, 1$ and 1, respectively (OR gate).
Also we will use the "L1 error" where $E(y, t) = t - y$.
Suppose we start with random weights where $w = [0.6, -0.2]$ and $b = -0.1$.

Input [1,0], target 1 We have $[0.6, -0.2] \cdot [1, 0] - 0.1 = 0.5$ so $\sigma(0.5) = 1$ thus $E = 0$ (good).

Input [0,1], target 1 We have $[0.6, -0.2] \cdot [0, 1] - 0.1 = -0.3$ so $\sigma(-0.3) = 0$ thus $E = 1$. Let us update our weight based on the following rules:

$$w = w + Ex$$
$$b = b + Et$$

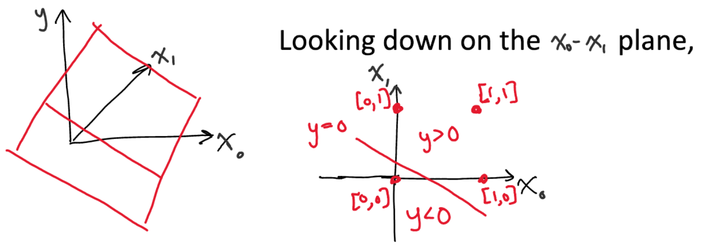
Thus $w = [0.6, -0.2] + 1[0, 1] = [0.6, 0.8]$ and $b = -0.1 + 1(1) = 0.9$.

Input [0,0], target 0 We have $[0.6, 0.8] \cdot [0, 0] + 0.9 = 0.9$ so $\sigma(0.9) = 1$ and $E = -1$.

Note that for this specific input/target pair no updates occur with our update rules.

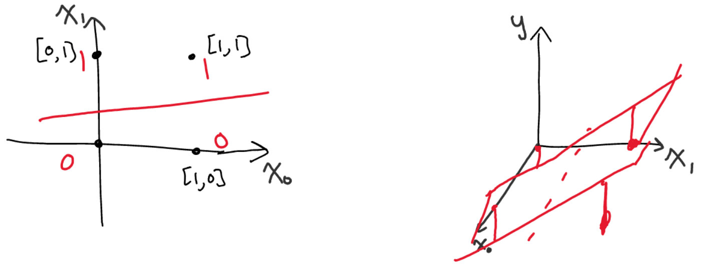
Input [1,1], target 1 We have $0.6 + 0.8 + 0.9 = 2.3$ so $\sigma(0.9) = 1$ and $E = 0$.

Eventully we note that when $w = [0.6, 0.8]$ and $b = -0.1$ this neuron satisfies our desired behaviour.
There is in fact a geometric intepretation suppose x_1 and x_2 are free and we have $y = 0.6x_1 + 0.8x_2 - 0.1$ a linear equation. In $\mathbb{R}^3$ we end up with a plane that separates $y < 0$ and $y \geq 0$

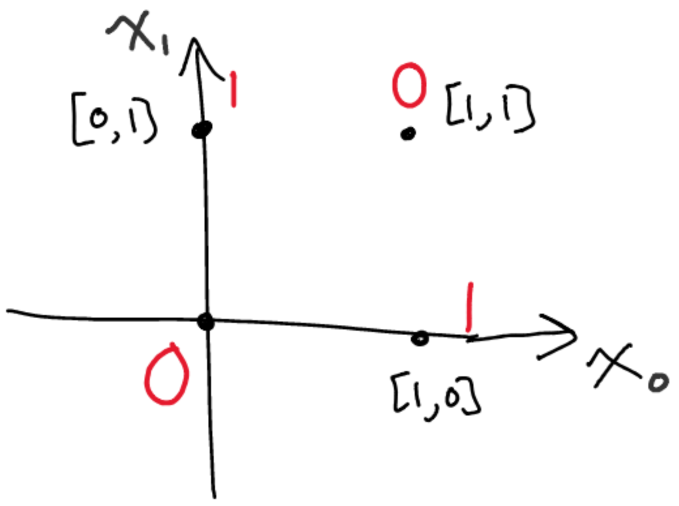


That is: find the weights and biases for our perceptron is the same as finding a **linear classifier**, a linear function that returns a positive value for the inputs that should yield a 1 and a negative value for the inputs that should yield a 0.

Another example is to produce a perceptron such that $[0,0] \to 0, [0,1] \to 1, [1,0] \to 0, [1,1] \to 1$. A possible solution is



Finally, what about XOR i.e. $[0,0] \to 1, [0,1] \to 1, [1,0] \to 1, [1,1] \to 0$?



Note that for the XOR function there is **no linear classifier** that will work.

Remark 6.1. Perceptrons are simple, two-layer neural networks and only work for **linearly separable data**.

7 January 23, 2019

7.1 Gradient descent learning

Note that the operation of our network can be written as

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

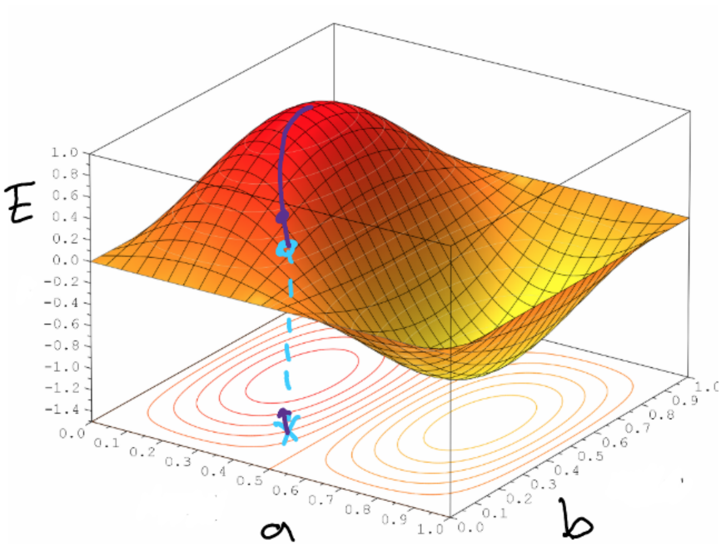
If our cost function is $E(\vec{y}, \vec{t})$ where $\vec{t}$ is the target, then the neural learning becomes an optimization problem where

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

(note that we minimize the expectation over our *entire* data distribution). We can apply gradient descent to E using the gradient

$$\nabla_\theta E = \begin{pmatrix} \frac{\partial E}{\partial \theta_0} & \frac{\partial E}{\partial \theta_1} & \cdots & \frac{\partial E}{\partial \theta_p} \end{pmatrix}^T$$

If we want to find a local maximum of a function, one can simply start somwhere and keep walking "uphill" using **gradient ascent**. Suppose we have a function with two inputs and error $E(a,b)$. We wish to find a,b to maximize E



We are thus trying to find parameters that yield the maximum value i.e.

$$(\bar{a},\bar{b}) = \text{argmax}_{(a,b)} E(a,b)$$

"Uphill" regardless of the current a,b is *in the direction* of the gradient

$$\nabla E(a,b) = \begin{pmatrix} \frac{\partial E}{\partial a} & \frac{\partial E}{\partial b} \end{pmatrix}^T$$

that is given current position (a_n, b_n) we perform the update that steps in the direction of the gradient

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

where k is the step multiplier or **learning rate**.
Gradient descent is similar to gradient ascent but instead aims to minimize the objective function: that is one walks downhill in the direction **opposite** of the gradient vector.

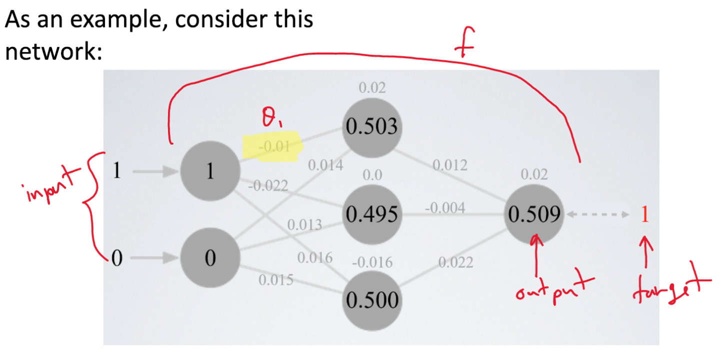
Remark 7.1. There is no guarantee one will actually find the *global optimum*. In general gradient ascent/descent will find a local optimum that may or may not be the global optimum.

We can **approximate the gradient numerically** using finite-differencing. For a function $f(\theta)$ we can approximate $\frac{df}{d\theta}$ by

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

for a small $\Delta\theta$.

Example 7.1. Consider the following network



We can model the action of the entire network as $\vec{y} = f(\vec{x};\theta)$.
We can formulate the optimization problem as

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

or more compactly as $\min_\theta \bar{E}(\theta)$ where $\bar{E}(\theta) = E(f(\vec{x},\theta),\vec{t}(\vec{x}))$.
Consider θ_1 in the diagram on its own. With $\theta_1 = -0.01$ our network output is $y = 0.509$ where our target is $t = 1$. This gives us an MSE $(y-t)^2$ for this single input $\bar{E}(-0.01) = 0.24113$.
What if we perturb θ_1 so that $\theta_1 = -0.01 + 0.5 = 0.49$? Then our output is $y = 0.5093$ with $\bar{E}(0.49) = 0.240761$.
Siilarly perturbing $\theta_1 = -0.01 - 0.5 = -0.51$ our output is $y = 0.5086$ giving us $\bar{E}(-0.51) = 0.24150$.
Note that error goes down if we perturb θ_1 up and error goes up if we perturb θ_1 down. Using finite differences we have

$$\frac{\partial \bar{E}}{\partial \theta_1} = \frac{\bar{E}(0.49) - \bar{E}(-0.51)}{2 \cdot 0.5} = -0.0007475$$

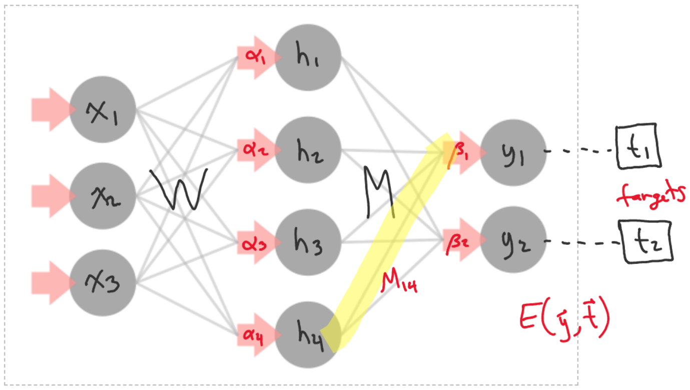
Obviously *increasing* θ_1 seems to be the right thing to do to minimize $\bar{E}$, thus

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

8 January 25, 2019

8.1 Error backpropagation

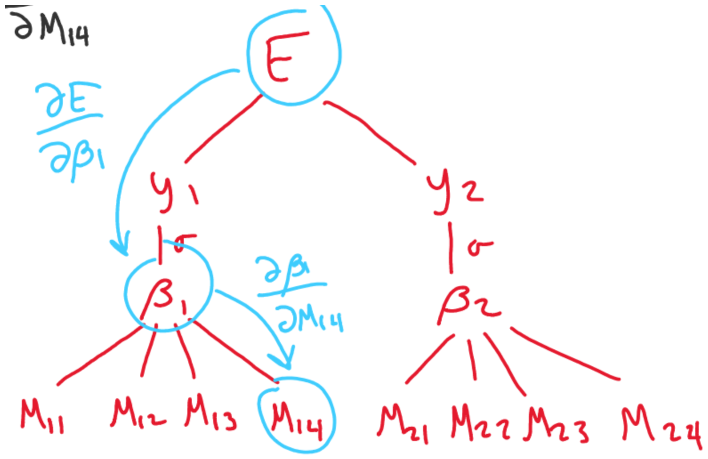
Instead of approximating the gradient using finite differencing, we can instead compute the actual gradient of our entire multi-layer network then run gradient descent appropriately. We can use the chain rule to compute the gradients from the loss at the end to any arbitrarily layer in the network and the parameters in that layer.
Suppose we have the following network:



Let α_i be the input current to hidden node h_i and β_j be the input current to output node y_j.
For our cost (loss) function we denote $E(\vec{y}, \vec{t})$.
For learning, suppose we want to compute $\frac{\partial E}{\partial M_{14}}$ (gradient of error wrt to the $(1, 4)$th entry of M). Note that our final loss of the network is a composition of functions:

$$E\Big(\sigma\big(M\sigma(W\vec{x} + \vec{a}) + \vec{b}\big), \vec{t}\Big)$$

If we draw out the "dependency" graph of the functions:



To compute $\frac{\partial E}{\partial M_{14}}$ (to adjust the parameter M_{14}), we can apply **chain rule** backwards from E to M_{14}. For example, to express it in terms of β_1

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

but note that

$$\frac{\partial E}{\partial \beta_1} = \frac{\partial E}{\partial y_1} \cdot \frac{\partial y_1}{\partial \beta_1}$$

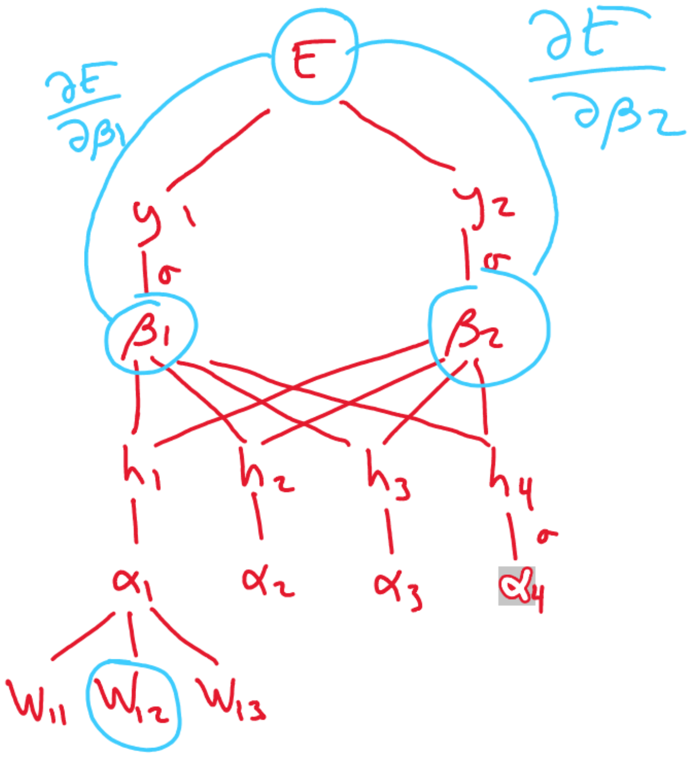
therefore

$$\frac{\partial E}{\partial M_{14}} = \frac{\partial E}{\partial y_1} \cdot \frac{\partial y_1}{\partial \beta_1} \cdot \frac{\partial \beta_1}{\partial M_{14}}$$

Recall $\beta_1 = \sum_{i=1}^4 M_{1i}h_i + b_i$ so $\frac{\partial \beta_1}{\partial M_{14}} = h_4$ i.e.

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

Let's go one layer deeper: suppose we'd like to find $\frac{\partial E}{\partial W_{12}}$. The dependency graph is



Note that if we first hop from E to α_1 then to W_{12}

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

Note that since $\alpha_1 = \sum_{i=1}^3 W_{1j}x_j + a_j$ then $\frac{\partial \alpha_1}{\partial W_{12}} = x_2$. Also

$$\begin{aligned}
\frac{\partial E}{\partial \alpha_1} &= \frac{\partial h_1}{\partial \alpha_1} \cdot \frac{\partial E}{\partial h_1} \\
&= \frac{\partial h_1}{\partial \alpha_1} \cdot \left[\frac{\partial \beta_1}{\partial h_1} \cdot \frac{\partial E}{\partial \beta_1} + \frac{\partial \beta_2}{\partial h_1} \cdot \frac{\partial E}{\partial \beta_2}\right] \\
&= \frac{\partial h_1}{\partial \alpha_1} \cdot \left[M_{11} \cdot \frac{\partial E}{\partial \beta_1} + M_{21} \cdot \frac{\partial E}{\partial \beta_2}\right] \qquad * \\
&= \frac{\partial h_1}{\partial \alpha_1} \begin{pmatrix} M_{11} & M_{12} \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial E}{\partial \beta_1} & \frac{\partial E}{\partial \beta_2} \end{pmatrix}
\end{aligned}$$

(*): we learned $\frac{\partial E}{\partial \beta_i}$ already when we were learning gradients for M.

Remark 8.1. To do backprop on W_{12}, we must take the derivative through **both** β_1 and β_2. Intuitively a change in W_{12} influences both the change of β_1 and β_2 so we need to account for both. This is evident in chain rule.

9 January 28, 2019

9.1 Backpropagation in matrix notation

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

$$\begin{aligned} \frac{\partial E}{\partial \alpha_i} &= \frac{\partial h_i}{\partial \alpha_i} \begin{pmatrix} M_{1i} & \ldots & M_{Yi} \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial E}{\partial \beta_1} & \ldots & \frac{\partial E}{\partial \beta_Y} \end{pmatrix} \\ &= \frac{\partial h_i}{\partial \alpha_i} \begin{pmatrix} M_{1i} & \ldots & M_{Yi} \end{pmatrix} \begin{pmatrix} \frac{\partial E}{\partial \beta_1} \\ \vdots \\ \frac{\partial E}{\partial \beta_Y} \end{pmatrix} \end{aligned}$$

For expressing the gradient for all α_i's:

$$\begin{pmatrix} \frac{\partial E}{\partial \alpha_1} \\ \vdots \\ \frac{\partial E}{\partial \alpha_H} \end{pmatrix} = \begin{pmatrix} \frac{\partial h_1}{\partial \alpha_1} \\ \vdots \\ \frac{\partial h_H}{\partial \alpha_H} \end{pmatrix} \odot \begin{pmatrix} M_{11} & \ldots & M_{Y1} \\ \vdots & \ddots & \vdots \\ M_{1H} & \ldots & M_{YH} \end{pmatrix} \begin{pmatrix} \frac{\partial E}{\partial \beta_1} \\ \vdots \\ \frac{\partial E}{\partial \beta_Y} \end{pmatrix}$$

where $\odot$ is the **Hadamard product** or element-wise matrix multiplication, that is

$$\begin{pmatrix} a & b \end{pmatrix} \odot \begin{pmatrix} c & d \end{pmatrix} = \begin{pmatrix} ac & bd \end{pmatrix}$$

Even more compactly:

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

Remark 9.1. Oftentimes we denote $\frac{\partial E}{\partial \vec{z}^{(l+1)}} = \nabla_{l+1} E$.

9.2 Backpropagation to adjust connection weights

More generally, when we advance more layer down during backpropagation:

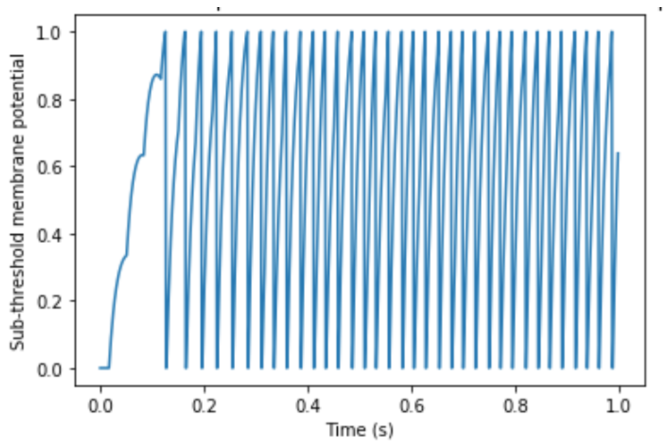


Figure 9.1: Suppose we needed to advance one layer futher down during backpropagation from layer $l+1$ to l.

let

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

we know for the activation $\vec{z}^{(l)}$ at layer l we have

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

thus to compute the *gradient for our connection weights*

$$\begin{aligned}\frac{\partial E}{\partial W_{ij}^{(l)}} &= \frac{\partial E}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial W_{ij}^{(l)}} \\ &= \frac{\partial E}{\partial z_i^{(l+1)}} h_j^{(l)}\end{aligned}$$

or for all connection weights

$$\frac{\partial E}{\partial W^{(l)}} = \begin{pmatrix} | \\ \frac{\partial E}{\partial \vec{z}^{(l+1)}} \\ | \end{pmatrix} \begin{pmatrix} - & \vec{h}^{(l)} & - \end{pmatrix}$$

which produces a matrix the same size as W (each (i,j)-th element corresponds to the error gradient of W_{ij}).

10 January 30, 2019

10.1 Training and testing

We want to develop a systematic way of training our neural network from training data. Note that our *ultimate goal is to train on unseen data* not in our training set. For this reason we usually break our data into two pieces:

Training set use more of our labelled data to train our model

Test set once our model is trained, we use the remaining labelled samples to evaluate our model

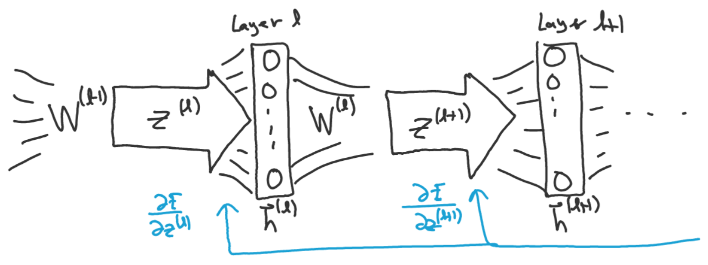
Definition 10.1 (Epoch)**.** Training usually involves going through the training data repeatedly and updating the network weights as we go. Each pass through the entire training data set is called an **epoch**.

Why do we need this split? Suppose after trial and error we find the optimal set of hyperparameters (e.g. number of neurons per layer, learning rate, number of epochs, initial weight) and we get a low error.
Does this accomplish what we want?
Consider the following example:

Example 10.1. Consider noisy samples coming from the ideal mapping

$$y = 0.4x - 0.9$$

and suppose our training dataset has the 5 samples



Since this is a regression problem we will use a **linear activation function** (basically the identity activation function) on the output and **MSE** as the loss function.
Suppose we create a neural network with 1 input neuron, 1000 hidden neurons, and 1 output neuron.
Suppose before training we get an MSE of 0.956 and after many epochs (e.g. 500) our average loss is 0.00069 (**training error**).
Suppose we receive a new set of samples and apply our model to it. Our average loss on this new set is 0.01565 which is not as good as our training error.

Recall that our sole purpose was to create a model to *predict the output for samples it has not seen.*

Definition 10.2 (Overfitting)**.** The false sense of success we get from results on our training dataset is known as **overfitting** or **overtraining**.
That is, the model starts to fit the noise of the training set rather than just to the underlying data distribution.

In we want to estimate how well our model will generalize to samples it has not trained on we can withhold further a part of our training set and tune our model on a **validation set**.

10.2 Overfitting

We saw that if a model has many degrees of freedom it becomes "hyper-adapted" to the training set and start to fit the noise in the dataset (training error is very small).
This is a problem since the model would not generalize to new samples (test error becomes much bigger than the training error).
There are some strategies (called **regularization**) to prevent our network from overfitting to the noise:

Weight decay We can limit overfitting by preferring solutions with smaller connection weights which can be achieved by adding a term to the loss function that penalizes the magnitude of the weights e.g.

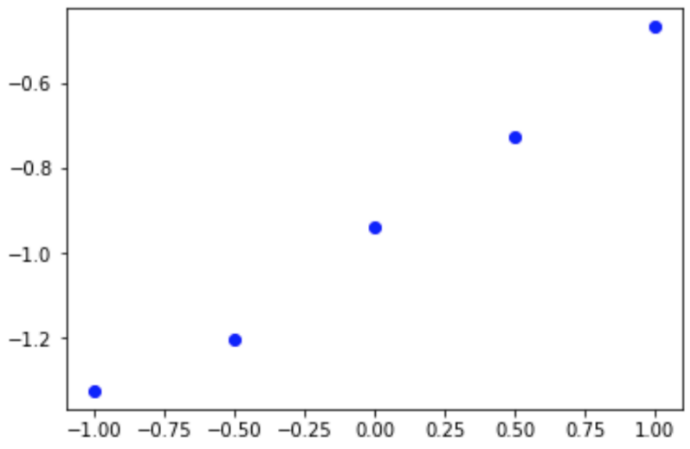
$$\tilde{E}(\vec{y}, \vec{t}; \theta) = E(\vec{y}, \vec{t}) + \lambda \|\theta\|_F^2$$

where $\|\theta\|_F^2 = \sum_i \theta_i^2$ the **Frobenius norm** or the **L2 norm** (L2 penalty).

How does this change our gradient and thus update rule?

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

for example in our previous data set, we see weight decay helps with our generalization error:



where λ controls the weight of the regularization term.

One can also use different norms, for example the **L1 norm**

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

the L1 norm favours sparsity (most weights are pushed down to zero with only a small number of non-zero weights).

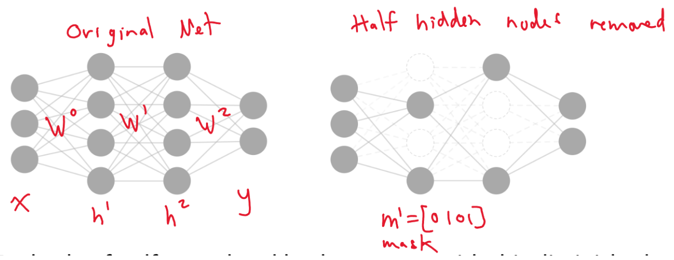
Data augmentation Another approach is to include a wider variety of samples in the training set so model is more robust to the noise of a few.

For image-recognition datasets one can generate more valid samples by shifting or rotating images. Note: data augmentation transformations should presumably not change the labelling.

11 February 1, 2019

11.1 Dropout

The **dropout method** systematically ignores a large fraction (typically 50%) of the hidden nodes for each sample. That is: we randomly choose half of the hidden nodes to be temporarily zero'ed out:



During *training*, we do a feedforward and backprop pass with the diminished network.
During *feedforward*, suppose our activation at the first hidden layer is

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

then we apply our **dropout mask**

$$h^{(1)} = \hat{h}^{(1)} \odot m^1$$

Note that to compensate we need to double (or multiply by $\frac{1}{1-rate}$) the remaining current $h^{(1)}$ going into hidden layer 2, which is equivalent to doubling the weights

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

Let $\mathfrak{h}^{(1)} = 2h^{(1)}$ so $z^{(2)} = W^{(1)}\mathfrak{h}^{(1)} + b^{(2)}$. Likewise we get $\mathfrak{h}^{(2)}$ from hidden layer 2.
During *backprop*, suppose we have $\frac{\partial E}{\partial z^{(2)}}$ we can easily compute

$$\frac{\partial E}{\partial W^{(1)}} = \frac{\partial z^{(2)}}{\partial W^{(1)}}\frac{\partial E}{\partial z^{(2)}} = \mathfrak{h}^{(1)}\frac{\partial E}{\partial z^{(2)}} = 2h^{(1)}\frac{\partial E}{\partial z^{(2)}}$$

to compute the gradient for the next hidden layer down

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

Recall that $h^{(1)} = \sigma(z^{(1)})$ and $\mathfrak{h}^{(1)} = 2h^{(1)} = 2\sigma(z^{(1)})$, thus

$$\frac{\partial \mathfrak{h}^{(1)}}{\partial z^{(1)}} = 2\frac{\partial h^{(1)}}{\partial z^{(1)}}$$

as long as $\frac{\partial h^{(1)}}{\partial z^{(1)}} = 0$ when $h^{(1)} = 0$ (which holds for the logistic function where $\frac{\partial h^{(1)}}{\partial z^{(1)}} = h^{(1)}(1 - h^{(1)})$).

Remark 11.1. In general we need to keep a reference of the input current $z^{(1)}$ in order to compute $\frac{\partial h^{(1)}}{\partial z^{(1)}}$. While the logistic function's derivative can be computed in terms of h (the activation), some activation functions and their derivatives cannot e.g. $\arctan(z)$ where $\arctan'(z) = (1 + z^2)^{-1}$.

Therefore we have

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

Remark 11.2. We must **double** (or multiply by $\frac{1}{1-rate}$) the connection weights projecting from a diminished layer in order to give *reasonable* inputs to the next layer (as a result of our dropout).
In practice we usually scale by the number of actual nodes remaining after the mask is applied per iteration rather than the expected number of nodes remaining.

Question 11.1. Why does dropout work?

1. It's akin to training a bunch of different networks and combining or ensembling their answers. Each diminished network is a contributor to this ensemble or consensus strategy.

2. Dropout disallows sensitivity to particular combination of nodes. Instead the network seeks a solution that is robust to loss of nodes.

11.2 Deep neural networks

Question 11.2. How many layers should our neural network have?

We first look at the following theorem:

Theorem 11.1 (Universal Approximation Theorem)**.** Let $\phi(\cdot)$ be a non-constant bounded and monotonically increasing continuous function.
Let I_m denote the m-dimension unit hypercube $[0,1]^m$.
The space of continuous real-valued functions on I_m is denoted as $C(I_m)$.
Then given any $\epsilon > 0$ and any function $f \in C(I_m)$, $\exists N \in \mathbb{N}$, real constants $v_i, b_i \in \mathbb{R}$, and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$ where we define

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

then an approximation for f is F that is independent of ϕ, that is

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

If we let x be our input vector, the N w_i's form our $N \times M$ weight matrix W, and our N v_i's as our final output connection weights, then the approximation theorem essentially says any real-valued m-dimensional function can be approximated by a **single-layer neural network**.
However the number of hidden nodes N grows exponentially for certain functions, thus a *deeper* network is preferred.
Why don't we always use really deep networks?

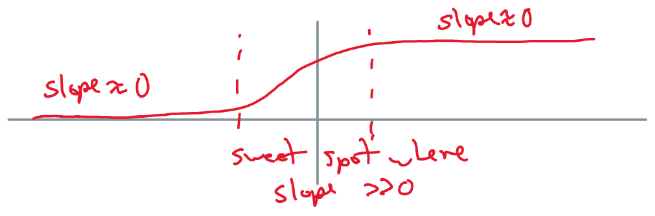
12 February 4, 2019

12.1 Vanishing gradients

Suppose the initial weights and biases were large enough such that the input current to many of the nodes was not close to zero. For example, consider the output node $y_1 = \sigma(z_1)$ with input current z_1.
Suppose $z = 5$ thus $y_1 = \frac{1}{1+e^{-5}} = 0.9933$. However note that

$$\frac{dy_1}{dz_1} = y_1(1 - y_1) = 0.0066$$

We compare the gradient if the input current was 0.1: $y_1 = \sigma(0.1) = 0.525$ and $\frac{dy_1}{dz_1} = 0.249$ which is almost 40 times larger than the gradient before!
We take a look at the logistic function and how its slope varies across its domain:

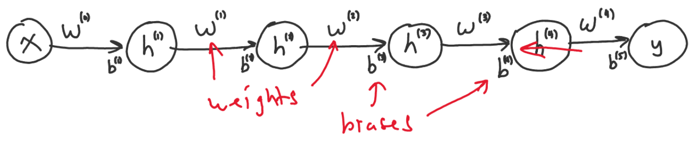


note that as $z \to -\infty$ or $z \to \infty$ then $\frac{dy}{dz} \to 0$, hence the updates to the weights will be smaller when th input currents are large in magnitude.
What about the next layer down? Suppose that $\frac{\partial E}{\partial \vec{z}^{(4)}} \approx 0.01$ (error gradient in the last layer) and what if the inputs to penultimate layer were about 4 in magnitude?
Then the corresponding slopes of their sigmoid functions will also be small where $\sigma(4) = 0.982$ and $\sigma'(4) = 0.0177$. Thus if we wanted to calculate the error gradient for the penultimate layer:

$$\begin{aligned}\frac{\partial E}{\partial \vec{z}^{(3)}} &= \frac{\partial \vec{h}}{\partial \vec{z}^{(3)}} \odot (W^{(3)})^T \frac{\partial E}{\partial \vec{z}^{(4)}} \\ &= (0.0177) \odot (W^{(3)})^T (0.01) \\ &= (0.000177) \odot (W^{(3)})^T\end{aligned}$$

The gradient becomes smaller and smaller the deeper we go. When this happens learning comes to a halt especially in deep layers. This is often called the **vanishing gradients problem**.

Example 12.1. Consider the following simple but deep network:



We start the loss on the output side $E(y,t)$. The gradient wrt the input current of the output node is

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

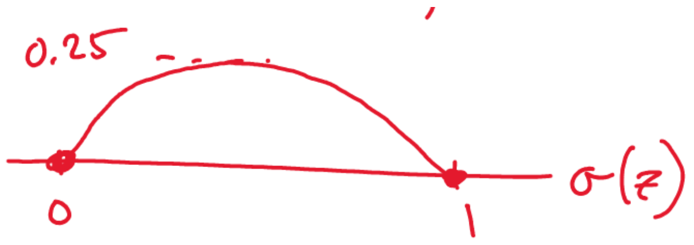
Using backprop we can compute a single formula for

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

Going deeper we have

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

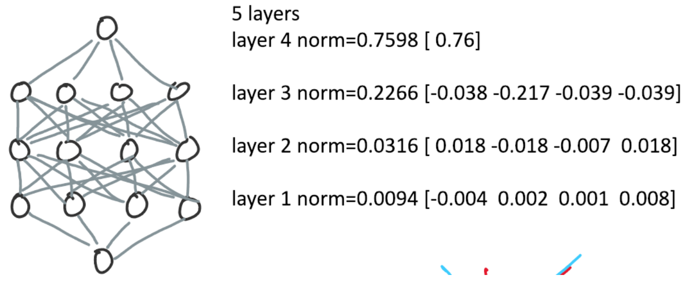
What is the steepest slope that $\sigma(z)$ attains? We note that $\sigma'(z) = \sigma(z)(1-\sigma(z))$ for $0 \le \sigma(z) \le 1$ which looks like:



we see it attains a maximum at $\sigma(z) = 0$ which corresponds to the inflection point in the logistic curve.
All else being equal the gradient (*at best*) goes down by a factor of at least 4 each layer (and worse goes down by a larger factor).
We can observe this if we look at the norms of the gradients at each layer in a model

$$\left\|\frac{\partial E}{\partial z^{(1)}}\right\|^2 = \sum_j \left(\frac{\partial E}{\partial z_j^{(i)}}\right)^2$$

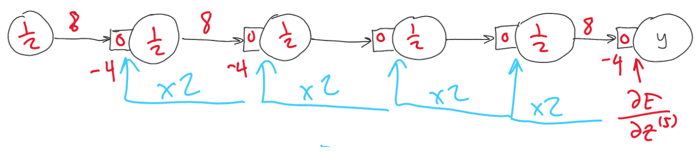
for example in a 5 layer network we may have


```
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]
```

where the gradients vanish to 0 the deeper we backprop.

12.2 Exploding gradients

A similar but less frequently observed phenomenon can result in very large gradients. Consider the following network with logistic activations, an initial input current of $\frac{1}{2}$, weights of 8, and biases of -4:



We note that for the error gradient for the first layer

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

The situation is more rare since this 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.

13 February 8, 2019

13.1 Batch gradient descent

We would like to optimize our training by using larger batches to take advantage of fast matrix processing in CPUs and GPUs.
Suppose our training set is

$$\{(\vec{x}_1, \vec{t}_1), \dots, (\vec{x}_1, \vec{t}_{\mathbb{P}})\}$$

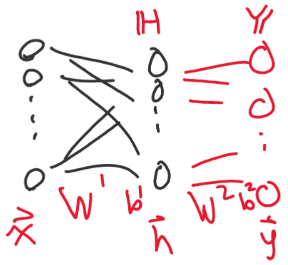
where $\vec{x} \in \mathbb{R}^{\mathbb{X}}, \vec{t} \in \mathbb{R}^{\mathbb{Y}}$, and $\mathbb{P}$ is the number of samples in our training set.
Notice we can put all our $\mathbb{P}$ inputs into a single matrix

$$X = \begin{bmatrix}\vec{x}_1 & \dots & \vec{x}_{\mathbb{P}}\end{bmatrix} \in \mathbb{R}^{\mathbb{X}\times\mathbb{P}}$$

similarly with our targets

$$T = \begin{bmatrix}\vec{t}_1 & \dots & \vec{t}_{\mathbb{P}}\end{bmatrix} \in \mathbb{R}^{\mathbb{Y}\times\mathbb{P}}$$

Does this help us? Yes. Consider the given network and the 1st hidden layer:



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

We can process all $\mathbb{P}$ inputs at once:

$$Z^{(1)} = W^1X + b'\begin{bmatrix}1 & \dots & 1\end{bmatrix}$$

whereby we broadcast our biases outwards by $\mathbb{P}$ dimensions:

$$\underset{b}{\begin{bmatrix} \ \end{bmatrix}}\begin{bmatrix}1 & \cdots & 1\end{bmatrix} = \begin{bmatrix} b & \cdots & b\end{bmatrix}$$

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

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

Now working our way back via backprop we get

$$\frac{\partial E}{\partial Z^{(L)}} = Y - T$$

which has dimensions $\mathbb{Y} \times \mathbb{P}$. Going one more layer down

$$\frac{\partial E}{\partial Z^{(L-1)}} = \frac{\partial H}{\partial Z^{(L-1)}} \odot \left(W^{(L-1)}\right)^T \frac{\partial E}{\partial Z^{(L)}}$$

where on the RHS we have dimensions $\mathbb{H} \times \mathbb{P}$, $\mathbb{H} \times \mathbb{Y}$ and $\mathbb{Y} \times \mathbb{P}$, respectively, resulting in a LHS of dimension $\mathbb{H} \times \mathbb{P}$.
Finally for the gradient of our weights at any layer l

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

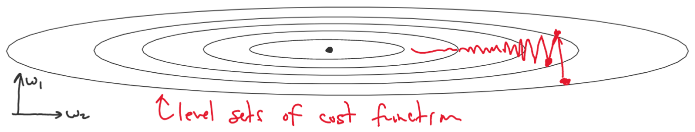
where on the RHS we have dimensions $\mathbb{Y} \times \mathbb{P}$ and $\mathbb{P} \times \mathbb{H}$, respectively, resulting in a LHS of dimension $\mathbb{Y} \times \mathbb{H}$.
Note that we can use the same backprop formulas to process a whole batch of samples: this is called **batch gradient descent**.

13.2 Stochastic gradient descent

One issue with batch gradient descent is that processing the entire dataset for a single update to weights can be slow. Instead we use an intermediary approach.
Rather than processing the entire dataset or just one sample, we can process chunks of our samples or **mini-batches** randomly chosen to determine our weight updates. This approach is more stable than single sample updates but faster than full-dataset updates.

13.3 Momentum-based gradients

Consider the following gradient descent trajectory:



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 that velocity $V = \frac{dD}{dt}$ and acceleration $A = \frac{dV}{dt}$. Solving for both using Euler's method:

$$D_{n+1} = D_n + \Delta t V_n$$
$$V_{n+1} = (1 - v)V_n + \Delta t A_n$$

where $v \in (0, 1)$ is the resistance (e.g. friction). Notice that our update of distance is similar to our update of our weights:

$$\text{Distance: } D_{n+1} = D_n + \Delta t V_n$$
$$\text{Weights: } W_{n+1} = W_n - \kappa \frac{\partial E}{\partial W_n}$$

where we treat V_n as our error gradient at step n.
Instead we let our instataneous gradient $\frac{\partial E}{\partial W_n}$ as acceleration A which we integrate to get an accumulated weight update akin to V. We call this new integrated acceleration V'.

For each weight W_{ij} we also calculate V'_{ij}. In matrix form for each $W^{(l)}$ we have a $V'^{(l)}$ such that:

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

or more commonly we have the convex equation

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

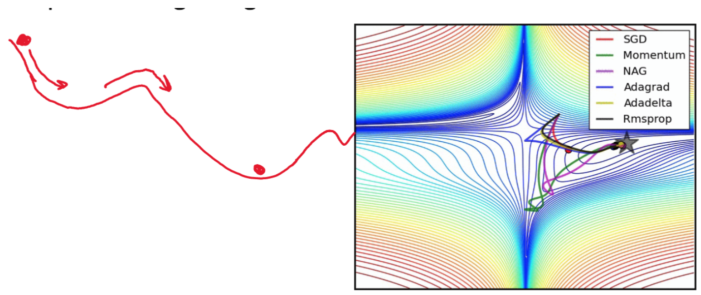
then our update to our weights would be

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

Remark 13.1. Our error gradient $V'^{(l)}$ is thus an exponential weighted average rather than a point-in-time gradient.

Remark 13.2. Not only does momentum-based gradient descent methods smooth out oscillations, but they can help us *avoid local minima.*

Various momentum-based gradient descent methods travel the cost landscape in various paths:



14 February 11, 2019

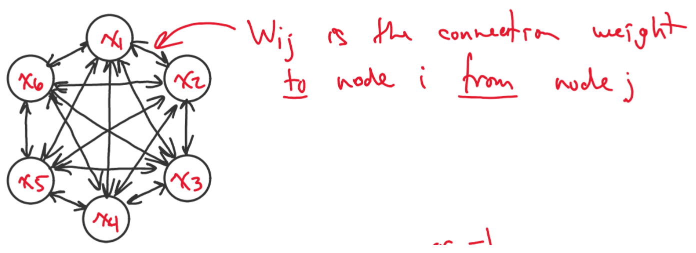
We look at **unsupervised learning techniques** next. Specifically we see how neural networks can be used to extract knowledge from *unlabelled data.*

14.1 Hopfield networks

Note that given an incomplete pattern such as "intel__igent", "irreplaceab__", or "1_34__789", we as humans can fill in the missing pieces since we've memorized these patterns. We can also detect errors e.g. "123856729" or "nueroscience".

Definition 14.1 (Content-addressable memory (CAM))**.** A **content-addressable memory (CAM)** is a system that can take part of a pattern and produce the most likely match from memory.

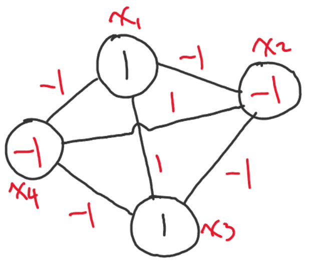
John Hopfield (1982) published a paper that proposes a method for using a neural network as a CAM. The network can learn patterns then converges to the closest pattern when shown a partial pattern.
The network is a complete bi-directional neural network:



Note that each node in the network can be a 0 (or alternatively -1) or a 1 i.e. $x_i \in \{-1, 1\}$ for $i = 1, \dots, N$.
We can also think of the network as a binary string of length N.
Suppose each node wants to change its state so that

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

(we will assume $b_i = 0$ (threshold) for now). That is node i is activated if its input current $W_{ij}x_j$ from all input nodes j exceeds the threshold.
If we have a pattern (*state* of the network) we would like to the network to recall, we could set the weights such that $W_{ij} > 0$ between 2 nodes in the *same state* and $W_{ij} < 0$ between 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/learn the connection matrix W that works for the set of "memories" we want to encode? Hopfield proposes that given n states (patterns) $\{x^{(1)}, \dots, x^{(n)}\}$ and N nodes

$$W_{ij} = \frac{1}{n} \sum_{s=1}^{n} x_i^{(s)} x_j^{(s)} \qquad i \neq j$$

$$W_{ii} = \frac{1}{n} \sum_{s=1}^{n} x_i^{(s)} x_i^{(s)} - 1 = 0$$

where $x_i^{(s)}$ is the ith node/bit of the sth state/pattern (we iterate through all training patterns). Note that W_{ij} is the **average co-activation between nodes i and j**. We can write this more compactly as

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

Why are these good update rules for W? Consider some state $x^{(q)} \in \{x^{(1)}, \ldots, x^{(n)}\}$:

$$\begin{aligned} Wx^{(q)} &= \frac{1}{n}\sum_{s=1}^{n} x^{(s)}(x^{(s)})^T x^{(q)} - Ix^{(q)} \\ &= \frac{1}{n}\sum_{s=1}^{n} x^{(s)}\left[(x^{(s)})^T x^{(q)}\right] - x^{(q)} \end{aligned}$$

Notice that if $x^{(s)} \perp x^{(q)}$ for $s \neq q$, then

$$\begin{aligned} Wx^{(q)} &= \frac{1}{n}x^{(q)}\|x^{(q)}\|^2 - x^{(q)} \\ &= \frac{N-n}{n}x^{(q)} \end{aligned}$$

that is for any state $x^{(q)}$ our weights converge within a (positive) constant multiple (as long as $N > n$).
Note that this method works best if the network states are all *mutually orthogonal.* In practice arbitrary vectors in very high dimensional space look orthogonal. Also in practice the number of states n is almost always kept much smaller than N the number of nodes.

15 February 13, 2019

15.1 Training hopfield networks

W_{ij} are set based on the patterns we give the Hopfield network (see rule for W_{ij} above).
Note that our update rule for node x_i depends on other nodes x_j for $j \neq 1$. Therefore we have circular dependency between nodes and their neighbours.
Synchronous update updates all the nodes at once using each node's current values. This is repeated for a number of epochs.
Asynchronous update updates only one node at a time which is picked randomly or in some pre-defined order.

Remark 15.1. Only the node values x_i are updated stochastically. The weights W_{ij} only depend on the memory patterns given.

15.2 Hopfield energy and Ising models

Hopfield recognized a link between Hopfield network states and Ising models in physics. The Ising model models a lattice of interacting magnetic dipoles (where each dipole can be "up" or "down"), and the state of each dipole depends on its neighbours.
Similar to the Ising model we can write the "energy" of our system using a Hamiltonian function. For our Hopfield neural network assuming W is symmetrical, we minimize the **Hopfield energy**

$$\begin{aligned} E &= \frac{-1}{2}\sum_{i,j} W_{ij}x_i x_j + \sum_i b_i x_i \\ &= \frac{-1}{2}x^T W x + b^T x \end{aligned}$$

Note that if $x_i = x_j$ (same sign), then we want W_{ij} to be positive as well. If $x_i \neq x_j$ (opposite signs), then we want W_{ij} to be negative. This aligns with what we want.

If we treat E as our cost function notice what gradient descent does:

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

Therefore we want to update $x_i \leftarrow x_i - t\big(-(\sum_{j\neq i} W_{ij}x_j) + b_i\big)$ or in DE form

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

Therefore if $\sum_{j\neq i} W_{ij}x_j - b_i > 0 \Rightarrow \sum_{j\neq i} W_{ij}x_j > b_i$ then we increase x_i, and vice versa. This is what we saw with x_i before.
For our weights we have

$$\frac{\partial E}{\partial W_{ij}} = -x_ix_j \qquad i \neq j$$

thus we have

$$\frac{\partial W_{ij}}{\partial t} = x_ix_j \qquad i \neq j$$

which is similar to our previous update rule for W. This is called the **Hebbian update rule**.

16 February 25, 2019

16.1 Autoencoders

An **autoencoder** is a neural network that learns to encode (and decode) a set of outputs, usually to a smaller dimensional space. An autoencoder looks like:

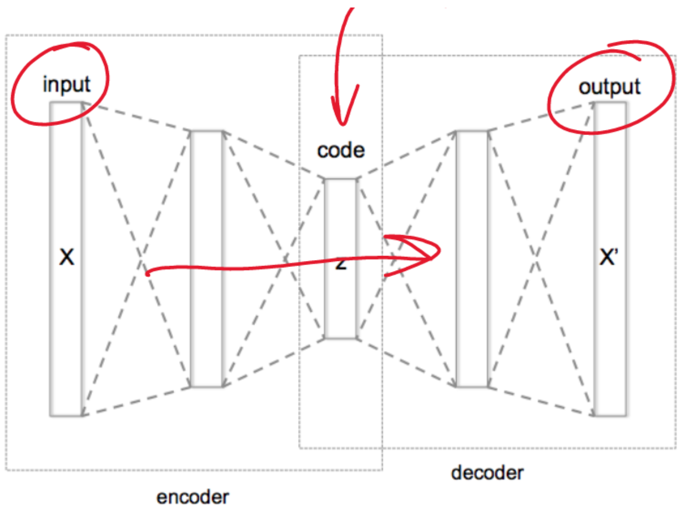


Figure 16.1: The "code" or encoding layer is smaller than the input and output layers.

We can use autoencoders to find efficient codes for high-dimensional data.
For example suppose we had 8-dimensional vectors (which has 256 different inputs) but the actual dataset only has 5 patterns. We can in principle encode each of the vectors into a unique 3-bit code. We can in general choose the dimension of the encoding layer.

Even though our autoencoder network is really just two layers (an input and an encoding layer), we can "unfold"/"unroll" into into three layers where the input and output layers are the same size and have the same state:

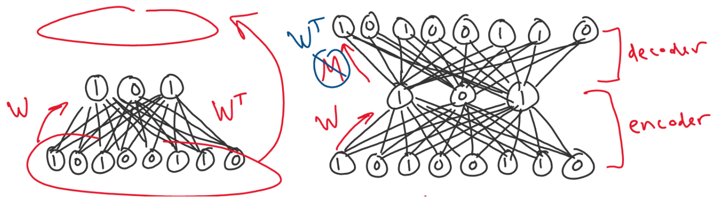


Figure 16.2: Left is a 2-layer representation of an autoencoder where we encode with weight matrix W and decode with W^T. Right is the autoencoder unrolled into three layers where we encode with W and decode with M.

We may or may not insist that the decoding matrix $M = W^T$. If we allow W and M to be different then it just becomes a 3-layer neural network.

Question 16.1. How do we enforce that they are the same?
Note that backprop will give us both $\frac{\partial E}{\partial W}$ and $\frac{\partial E}{\partial M}$. We simply set W and M to have the same initial weights and then update both using

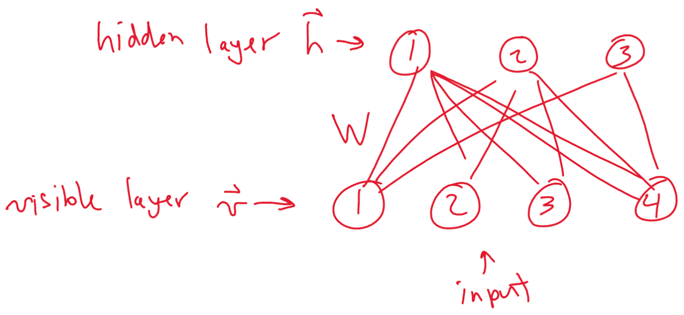
$$\frac{\partial \tilde{E}}{\partial W} = \frac{\partial \tilde{E}}{\partial M^T} = \frac{1}{2}(\frac{\partial E}{\partial W} + \frac{\partial E}{\partial W^T})$$

this is called **tied weights**.

17 February 27, 2019

17.1 Restricted Boltzmann Machines (RBM)

A **restricted Boltzmann machine (RBM)** is similar to a Hopfield network but it is split into two layers of nodes. The two layers of nodes are fully connected forming a *bipartite graph* (hence the "restricted" classification in RBM):



Each node is binary (either 1 "on" or 0 "off"). The probability that a node is on depends on the states of nodes feeding into it and the connection weights. Given $\vec{v} = [v_1, v_2, \ldots]$ as the visible state and $\vec{h} = [h_1, h_2, \ldots]$ as the hidden state, together they represent the network state.

Remark 17.1. We can think of an RBM as a rolled autoencoder where we only have one weight matrix W and we project the hidden layer back to the visible layer with $M = W^T$.

We define the **energy** of the network as

$$\begin{aligned} E(\vec{v},\vec{h}) &= -\sum_i \sum_j w_{ji} h_i h_j + \sum_i b_i v_i + \sum_j c_j h_j \\ &= -\vec{h}^T W \vec{v} + \vec{v}^T \vec{b} + \vec{h}^T \vec{c} \end{aligned}$$

Note that the RHS terms correspond to:

$-\vec{h}^T W \vec{v}$ discount when both nodes i and j are both on simultaneously

$\vec{v}^T \vec{b}$ energy incurred by turning on a visible node

$\vec{h}^T \vec{c}$ energy incurred by turning on a hidden node

We wish to find the minimum energy state. Consider the "energy gap": the difference in energy when we flip node v_k from off to on:

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

Therefore, if $\Delta E_k > 0$ then $E(v_k \text{ off}) > E(v_k \text{ on})$ (i.e. "on" is lower energy) so we set $v_k = 1$. Similarly if $\Delta E_k < 0$ then $E(v_k \text{ off}) < E(v_k \text{ on})$ (i.e. "off" is lower energy) so we set $v_k = 0$.
Sine the energy gap of each node depends on the state of other notes, finding the minimum energy state requires some stochasticity. One strategy is to visit and update nodes in random order (similar to Hopfield updates). We can do better since our network is bipartite. The visible units only depend on the hidden units and vice versa, so we can update one whole layer at a time.
This is another exaple of a local optimization method which can get stuck in a local minimum (that is not the global minimum).

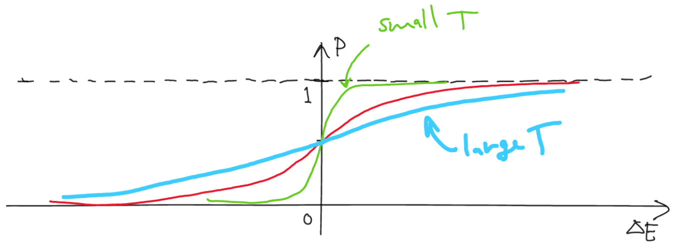
18 February 28, 2019

18.1 Updates for RBM

To avoid/reduce getting stuck, we add varying degrees of randomness, that is:

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

where T is the "temperature". This is called the **Boltzmann distribution**:

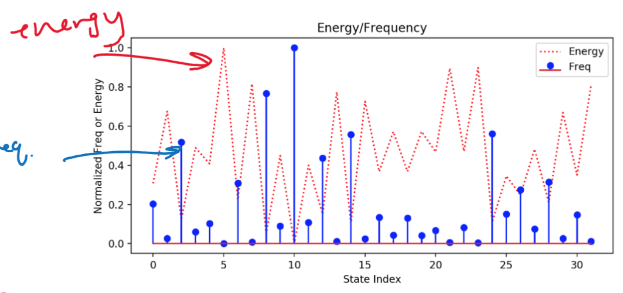


Remark 18.1. The Boltzmann energy and distributions come from **statistical mechanics**. Suppose the particles can be in one of 2 states (0 or 1), and that the energy difference between the states is ΔE.
P is thus the fraction of particles in state 1. As temperature increases there is more movement back and forth between the states hence the more flat distribution around 0.

Let us look at how to update node k first:

1. Compute ΔE_k
2. Evaluate $p = P(v_k = 1) \in (0, 1)$
3. Choose a uniform random number $r \in [0, 1]$
4. If $p > r$ then set node to "on" ($v_k = 1$), otherwise "off" ($v_k = 0$)

If we let the RBM run freely it will hop between states but will occupy low-energy states more frequently:



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

Our goal is to change the connection weights (W) and biases (b and c) so that our network states tend to be "relevant". For example:

Example 18.1. Suppose our network has 3 visible nodes and 2 hidden nodes.
Inputs to the visible nodes take the form of binary 3-vectors:

$$[0, 0, 0], [0, 0, 1], [0, 1, 0], \ldots, [1, 1, 1]$$

where we have 2^3 different vectors. Obviously our hidden layer does not have enough nodes to encode all objects; however perhaps our distribution frequently includes $[1, 0, 0], [0, 1, 0]$ and $[0, 0, 1]$ and almost never includes the others.
In this case the nodes have more than enough encoding capability as long as the statistically relevant patterns are encoded appropriately.

19 March 4, 2019

19.1 RBM learning via Contrastive Divergence

We want to encourage our network from holding the desired patterns and discourage it from wandering away.
We call the up pass (from visible to hidden) **recognition**.
We call the down pass (from hidden to visible) **generative**.
Given a visible pattern we want to generate visible states close to what we started after an up and down pass with the learned weights. That is: the visible input x gets encoded in the hidden nodes and we want that hidden representation to be able to only generate the same visible pattern x.

The hidden node offers a *new, more efficient representation* of the statistically salient patterns.
The **contrastive divergence** 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: compute $\Delta\vec{E} = W\vec{v}_1 - \vec{c}$ by projecting $\vec{v}_1$ up and we set each node of $\vec{h}_1$ with probability $P(\vec{h}_1 = 1) = \frac{1}{1+e^{-\Delta E/T}}$ (element-wise logistic).

2. Get co-occurrence/co-activation statistics: number of times node v_i occurs with node h_j that is
$$S_1 = \vec{v}_1\vec{h}_1^T$$
where S_1 has dimension $(m \times 1) \times (1 \times n) = m \times n$ which is the same size as W.

3. Generative pass: compute $\Delta\vec{E} = W^T\vec{h}_1 - \vec{b}$ by projecting $\vec{h}_1$ back down and we set each node of $\vec{v}_2$ with probability $P(\vec{v}_2 = 1) = \frac{1}{1+e^{-\Delta E/T}}$ (element-wise logistic).

4. Recognition pass 2: same as above, producing $\vec{h}_2$ with $\vec{v}_2$.

5. Get co-occurrence/co-activation statistics again ($S_2 = \vec{v}_2\vec{h}_2^T$).

6. Update weights using
$$W_{new} = W_{old} + \kappa(S_1 - S_2)$$
and similarly for biases
$$b_{new} = b_{old} - \gamma(\vec{v}_1 - \vec{v}_2)$$
$$c_{new} = c_{old} - \gamma(\vec{h}_1 - \vec{h}_2)$$
Note we have +'ve signs for weights and −'ve signs for biases: this comes from the flipped signage of each corresponding term in the energy function.

Remark 19.1. Typically one would process many (≈ 100) input patterns and collect statistics for all of them before making one update to weights/biases.

20 March 6, 2019

20.1 RBM training algorithm

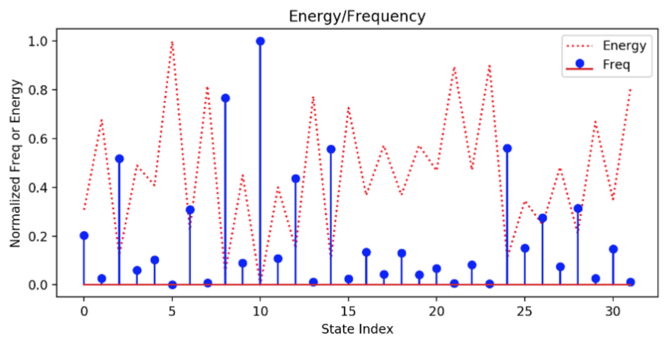
We outline the algorithm for training an RBM:

Algorithm 2 RBM training

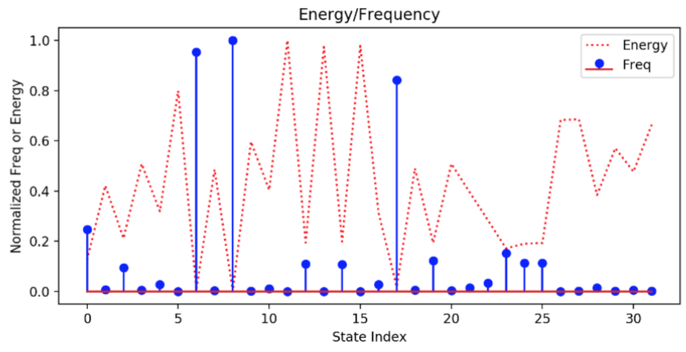
```
1: for temperature $T = 20, 10, 5, 2, 1$ do
2:     for 400 epochs do
3:         for sample in a batch do
4:             Choose visible pattern
5:             Add a little noise
6:             Project up: $V_1, H_1, S_1$
7:             Project down and back up: $V_2, H_2, S_2$
8:             Increment statistics
9:         Update weights and biases
```

Note that the innermost for loop can be done using matrix formulas.

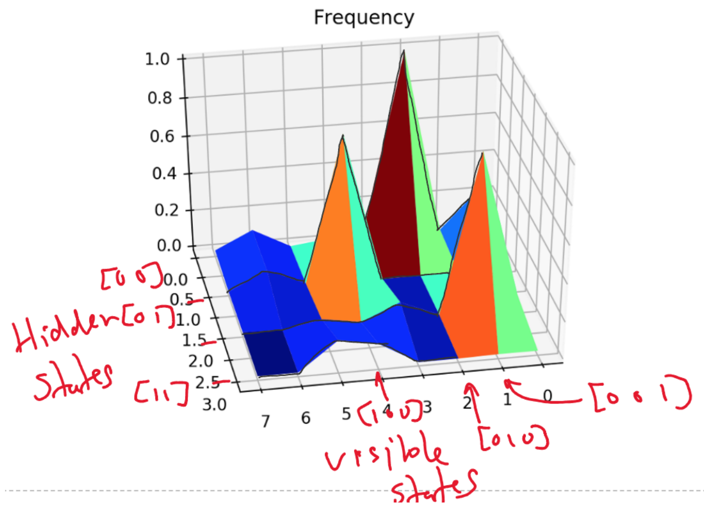
Example 20.1. Consider an RBM with 3 visible nodes and 2 hidden nodes ($2^5 = 32$ states).
Suppose the universe consists of mostly $[1, 0, 0], [0, 1, 0], [0, 0, 1]$.
When we initialize the network with random weights we may get the following energy levels and state distribution:



After training for a certain number of epochs, both the energy and state distribution changes, with only a small number of states dominating. These three high frequency states correspond to our three visible patterns and their learned hidden patterns:



On a 2D surface with hidden nodes and visible nodes on the two axes:



The above trains one autoencoder layer in an RBM. After we've trained one layer, we can train another layer on top of it. This is called a **belief network**. For example a belief network for MNIST may look like:

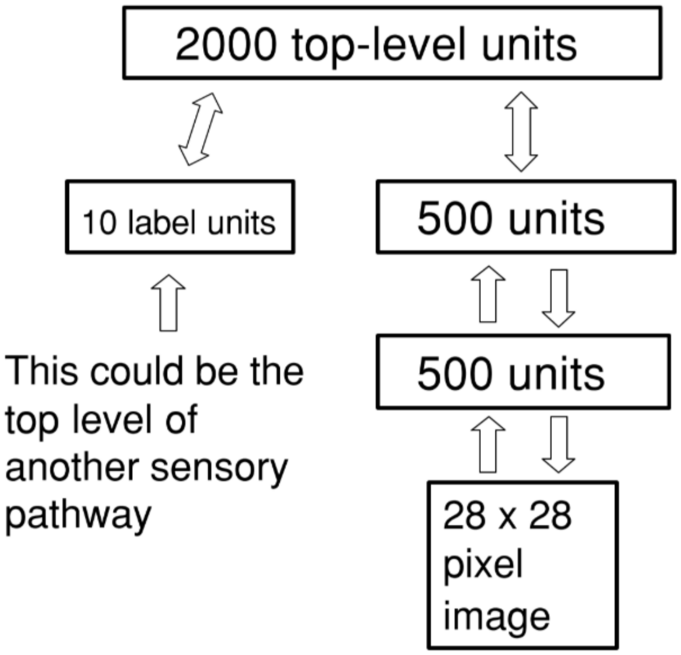
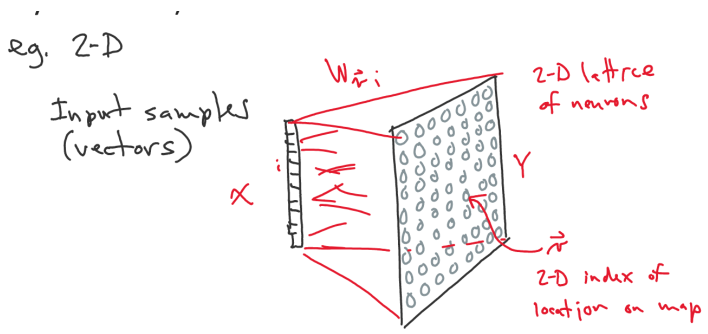


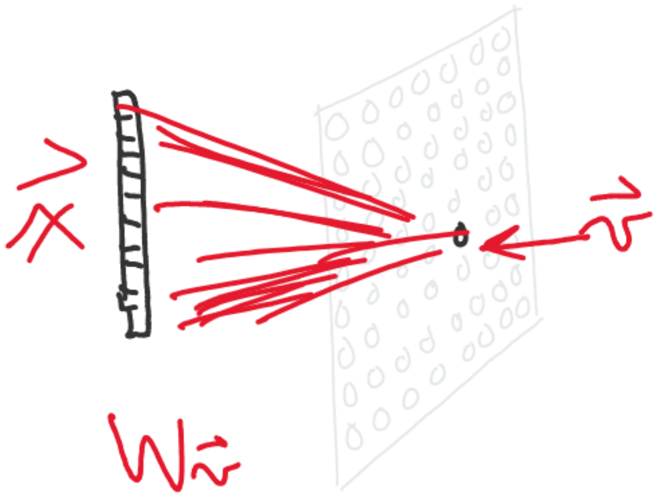
Figure 20.1: Hinton et al. "A Fast Learning Algorithm for Deep Belief Nets".

20.2 Self Organizing Maps (SOM)

SOMs are another method for unsupervised learning. The underlying network has its neurons arranged in a topological space, which is usually 2-dimensional but could be any n-dimensional.
For example we may project a 1D input vector $\vec{x}$ onto a 2D lattice of neurons Y:



For a given node in Y indexed at $\vec{v} = (v_1, v_2)$, we associate with it a weight vector $W_{\vec{v}}$ of the same dimension as $\vec{x}$:



In terms of updating $W_{\vec{v}}$ we have

$$W_{\vec{v}}^{t+1} = W_{\vec{v}}^{t} + \theta(\vec{v}; \vec{u}_{BMU})(\vec{x} - W_{\vec{v}}^{t})$$

where $\theta(\vec{v}; \vec{u}_{BMU})$ is some nonlinearity (see below). That is: we tend to update $W_{\vec{v}}$ towards $\vec{x}$.
Each of the neurons in Y compete against each other to encode the input. The neuron $\vec{v}$ with the closest weight vector $W_{\vec{v}}$ to $\vec{x}$ is called the **Best Matching Unit (BMU)**: $\vec{u}_{BMU}$.
The BMU's weights are influenced the most in the update rule. The weights of other neurons are influenced according to their spatial distance from the BMU. The spatial function used is called the **neighbourhood function**. A typical neighbourhood function might look like:

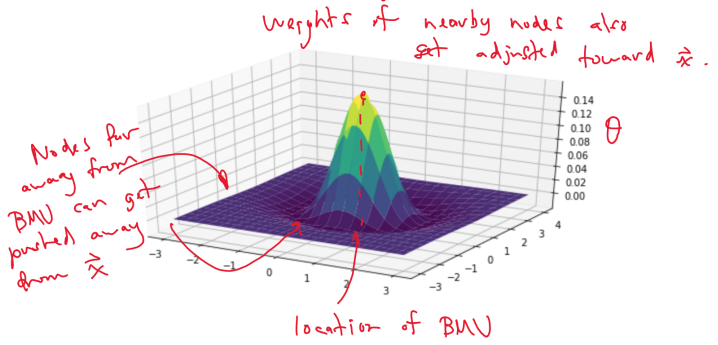


Figure 20.2: Neighbourhood function centred at $\mu = \vec{u}_{BMU}$. Nodes far away from BMU can actually get pushed away from $\vec{x}$ (since the neighbourhood function dips below 0).

For example the neighbourhood function θ could be the **difference of Gaussians (DOG)**:

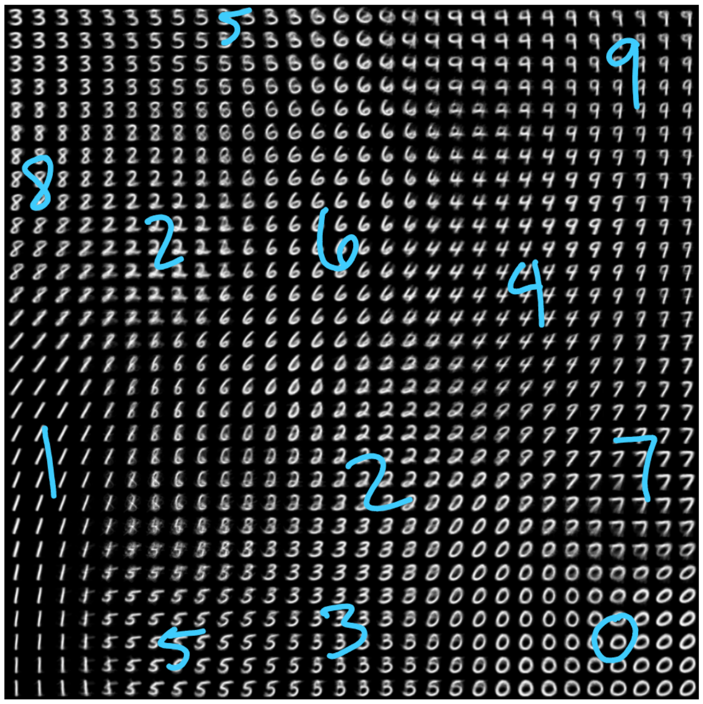
$$\theta(\vec{v}; \vec{u}_{BMU}) = N(\vec{v}; \mu = \vec{u}_{BMU}, \sigma_1) - N(\vec{v}; \mu = \vec{u}_{BMU}, \sigma_2)$$

The paradigm where neurons further away are pushed away from the goal is called in neuron science **"local excitation, lateral inhibition"**.
Over time, this encourages neurons in the same area to have similar weights.

Finally adding some time-dependent decay/learning rate:

$$W_{\vec{v}}^{t+1} = W_{\vec{v}}^{t} + \alpha(t)\theta(\vec{v}; \vec{u}_{BMU})(\vec{x} - W_{\vec{v}}^{t})$$

For example, a 30×30 SOM trained on MNIST gives us:



Note that the SOM clusters similar inputs and in essence embeds them into a lower-dimensional (2-dimensional) topological space.

21 March 8, 2019

21.1 Recurrent neural networks (RNN)

So far we have mostly been focusing on feedforward networks with no loops, but there are reasons we might want to allow feedback connections that create loops.

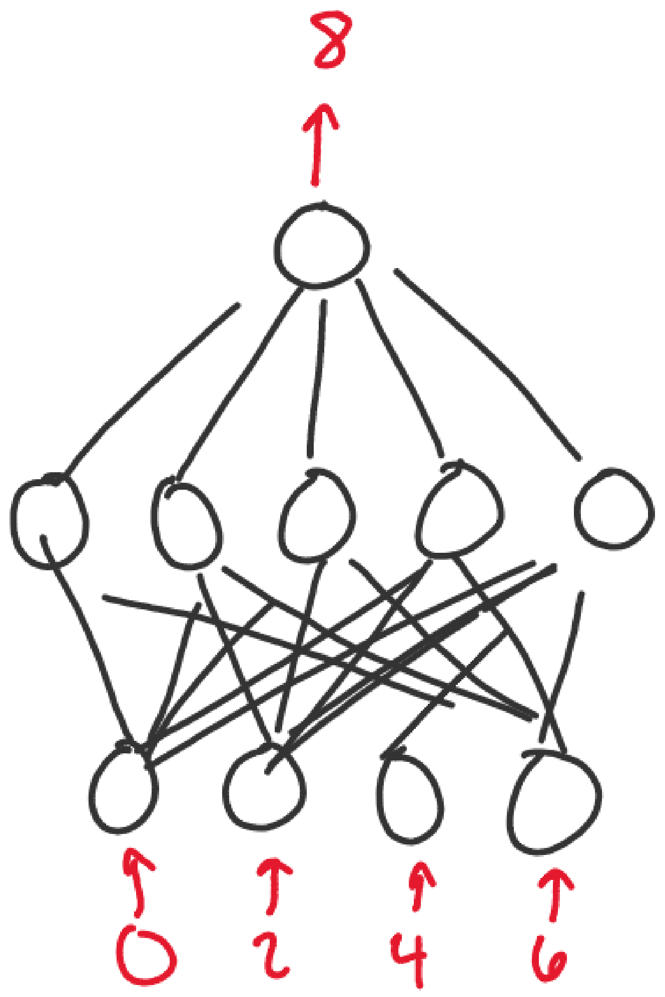
Example 21.1. We recently looked at recurrent continuous-time networks to implement dynamics.
We may also want to build a running memory into our network behaviour.

Consider the following tasks of predicting the next word in a sentence:

1. Emma's cat aws sick, so she took her to the . . .
2. She picked up the object, studied it, then put it . . .
3. $0, 2, 4, 6, \ldots$
4. $1, 2, 4, 8, \ldots$

In each case, the word you predict depends, sometimes in very complex ways, on preceding words. Thus the network will probably need to encode an ordered sequence of words to solve this problem.

Solution 1 We could design a network that takes the entire sequence as the input at once:



However:

1. The network can only consider fixed-length sequences
2. No processing can occur until the entire sequence is given

Solution 2 Allow the state of the network to depend on the new input as well as its previous state.

We let the network be **recurrent**. A **recurent neural network (RNN)** may look something like:

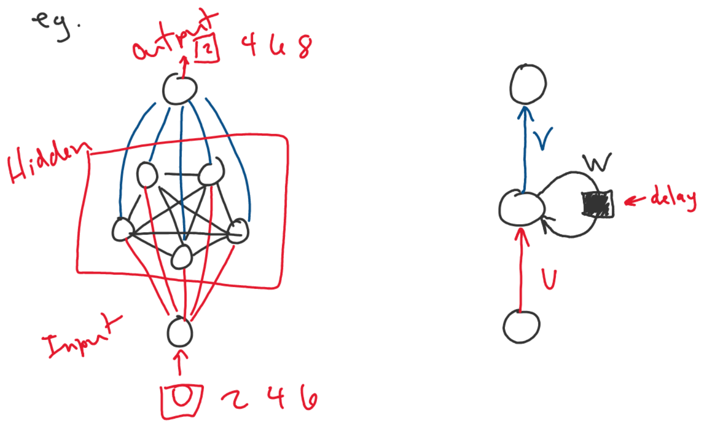
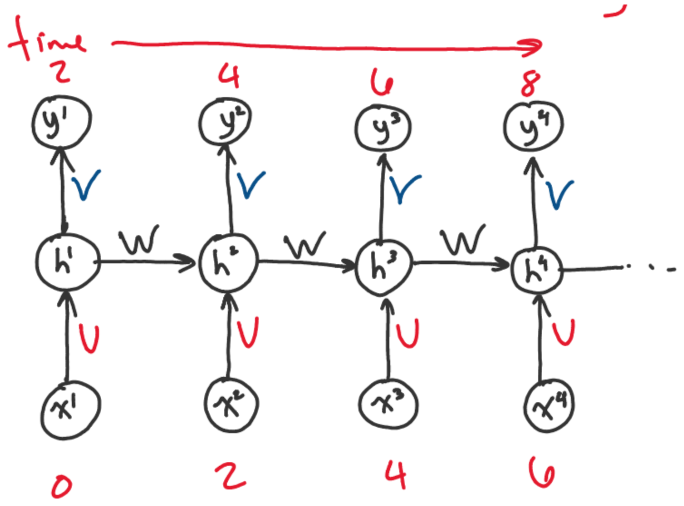


Figure 21.1: Left: hidden layers consists of several nodes. The output will depend on the current input as well as the previous state of the hiddnen layer. Right: common notation/representation of an RNN layer.

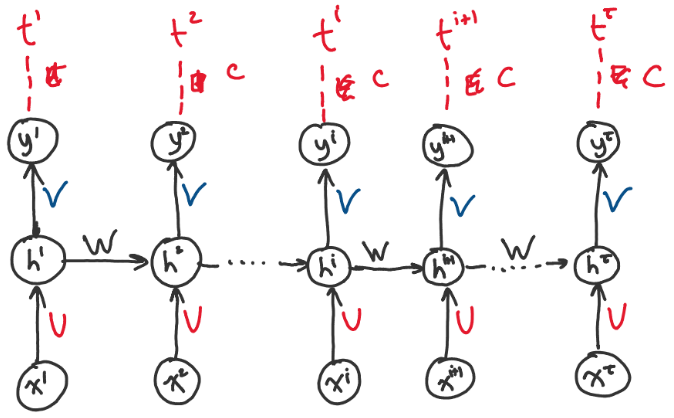
In an RNN, the state of the hidden layer can encode the input sequence and thus have the information it needs to determine the proper output.
How exactly do we train such a network? First we **unroll** the network, similar to what we did with autoencoders, but this is **unrolling through time**:



We note that in the above diagram:

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

Like before, we will have targets $t^{(i)}$ and a cost function C:



The cost or error is the sum from all the loss functions:

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

where τ is the number of timesteps we want to perform a backprop pass with (chosen during training) and α_i is a weight given to the cost at time step i.
As usual we aim to minimize the expected cost over the dataset with respect to the connection weights and biases, that is for $\theta = \{U, V, W, b, c\}$:

$$\min_\theta \Big\{ \sum_{i=1}^{\tau} C(y^{(i)}, t^{(i)}) \Big\}_{y,t} = \min_\theta \{E(\dots)\}_{y,t}$$

To compute the gradients flowing to our weights, we start at the last output and work our way back through the network ($x \in \mathbb{R}^X, h \in \mathbb{R}^H, y \in \mathbb{R}^Y$):

$$\frac{\partial E}{\partial z^{(\tau)}} = \frac{\partial E}{\partial y^{(\tau)}} \odot \frac{\partial y^{(\tau)}}{\partial z^{(\tau)}} = \frac{\partial E}{\partial y^{(\tau)}} \odot \sigma'(z^{(\tau)}) \qquad y^{(\tau)} = \sigma(z^{(\tau)})$$

where the LHS has dimensions $Y \times 1$ and both RHS vectors have dimensions $Y \times 1$. We also have:

$$\frac{\partial E}{\partial h^{(\tau)}} = \frac{\partial z^{(\tau)}}{\partial h^{(\tau)}} \frac{\partial E}{\partial z^{(\tau)}} = V^T \frac{\partial E}{\partial z^{(\tau)}}$$

where the LHS has dimensions $H \times 1$ and the RHS vectors have dimensions $H \times Y$ and $Y \times 1$ since $z^T = V h^T + c$.

22 March 11, 2019

22.1 Gradient of hidden layer in RNN

Suppose we have already computed $\frac{\partial E}{\partial h^{(i+1)}}$ then we would like to compute

$$\begin{aligned}
\frac{\partial E}{\partial h^{(i)}} &= \frac{\partial}{\partial h^{(i)}}\big(\sum_{j=1}^{\tau} C(y^{(i)}, t^{(i)})\big) \\
&= \frac{\partial C(y^{(i)}, t^{(i)})}{\partial h^{(i)}} + \frac{\partial E}{\partial h^{(i+1)}}\frac{\partial h^{(i+1)}}{\partial h^{(i)}} \\
&= V^T \frac{\partial C}{\partial y^{(i)}} \odot \sigma'(z^{(i)}) + W^T \sigma'(s^{(i+1)}) \odot \frac{\partial E}{\partial h^{(i+1)}}
\end{aligned}$$

where on the RHS we dimensions $V^T \in \mathbb{R}^{H \times Y}$, $\frac{\partial C}{\partial y^{(i)}} \in \mathbb{R}^{Y \times 1}$, $\sigma'(z^{(i)}) \in \mathbb{R}^{Y \times 1}$, $W^T \in \mathbb{R}^{H \times H}$, $\sigma'(s^{(i+1)}) \in \mathbb{R}^{H \times 1}$, and $\frac{\partial E}{\partial h^{(i+1)}} \in \mathbb{R}^{H \times 1}$.
We can show inductively this holds for any i.
Once we have $\frac{\partial E}{\partial h^{(i)}}$ we can comptue the gradient of the cost with respect to the weights and biases i.e.

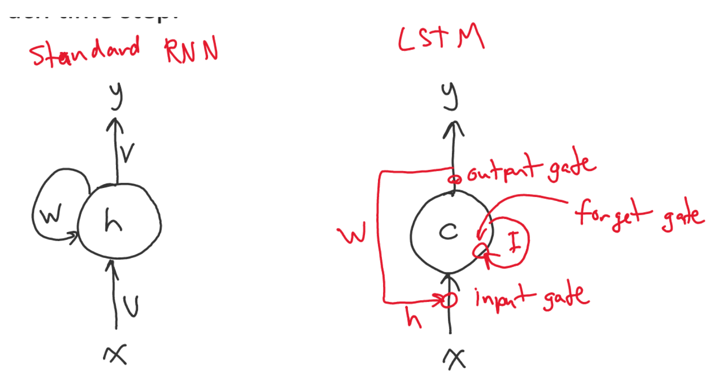
$$\frac{\partial E}{\partial V}, \frac{\partial E}{\partial W}, \frac{\partial E}{\partial U}, \frac{\partial E}{\partial b}, \frac{\partial E}{\partial c}$$

22.2 Long Short-Term Memory (LSTM)

The benefit of an RNN is that it can accumulate some hidden state that encodes input over time.
However one difficulty with an RNN is maintaining information in its hidden state for a long time. It would have troubling complete a long paragraph e.g. "A bicycle is an efficient mode of transportation, ... (other sentences) ... To get started, pick one up and put your feet on the ...."
This instability was first noted by Bengio et al. in 1994.
They showed the longer one needs to keep information in memory, the harder it is to train. The BPTT algorithm becomes unstable due to exploding or vanishing gradients since gradients are multiplied by recurrent connection weights at each time step.
Recall that

$$\begin{aligned}
\frac{\partial E}{\partial h^{(i)}} &= W^T \sigma'(s^{(i+1)}) \odot \frac{\partial E}{\partial h^{(i+1)}} + V^T \frac{\partial C}{\partial y^{(i)}} \odot \sigma'(z^{(i)}) \\
&\vdots \\
&= W^T \sigma'(s^{(i+1)}) W^T \sigma'(s^{(i+2)}) \ldots W^T \sigma'(s^{(\tau)}) \frac{\partial H}{\partial h^{(\tau)}}
\end{aligned}$$

whereby the repeated multiplication of W^T can cause vanishing or exploding gradients.
To combat this state decay, hochreiter and Schmidhuber (1997) proposed an additional hidden state that persists from step to step and does not get multiplied by the connection weights at each time step. A comparison between a standard RNN and LSTM:



There are various ways h and c can interact:

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

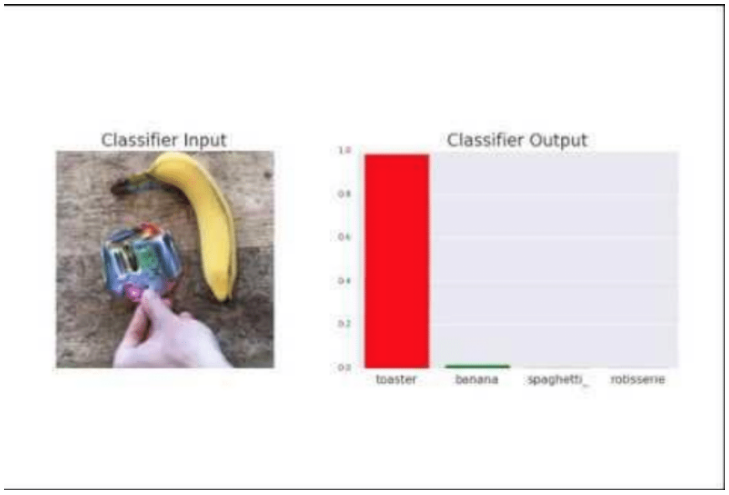
23 March 13, 2019

23.1 Adversarial examples

Adversarial examples are inputs specifically designed to cause a model to make mistakes in its prediction, although they look like valid inputs to a human. Formally: given a model f and an input x, find x' where $\|x - x'\|_p < \epsilon$ such that $f(x) \neq f(x')$.

Remark 23.1. Other perceptual similarity metrics other than L_p norm can be used.

Example 23.1. A physical **adversarial patch** (Brown et al., 2017) can cause a classifier to output something entirely incorrect with high probability:



Some terminology:

Adversarial attacks methods to generate adversarial examples

Adversarial defenses methods to defend against adversarial examples

Adversarial robustness property to resist misclassifcation 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

Adversarial pertubation difference between original example and its adversarial counterpart

Whitebox attack attacker has full access to victim model

Blackbox attack attacker only has access to victim's output

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

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

23.2 Adversarial attacks

The goal is to **minimally modify inputs to confuse the victim model**. Reformulated: we minimally modify inputs to *maximize some loss function.*
Whitebox adversarial attacks:

Fast Gradient Sign Method (FGSM) (Goodfellow et al., 2015) We take the gradient with respect to the input and do a **single-step gradient ascent**:

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

Basic Iterative Method (BIM) (Kurakin et al., 2017) BIM is the iterative variant of FGSM where

$$\begin{aligned} X'_0 &= X \\ X'_{n+1} &= Clip_{X,\epsilon}\Big\{X'_n + \alpha \text{sign}(\Delta_X L(X'_n, y))\Big\} \end{aligned}$$

A target attack variant of BIM called Iterative Least-Likely Class Method (ILLCM) uses the least-likely label y_{LL} instead of the true label y:

$$X'_{n+1} = Clip_{X,\epsilon}\Big\{X'_n - \alpha \text{sign}(\Delta_x L(X'_n, y_{LL}))\Big\}$$

Note that

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

Random FGSM (R+FGSM) (Tramer et al., 2018) R+FGSM is an FGSM variant but with a random starting point:

$$x^{adv} = x' + (\epsilon - \alpha)\text{sign}\big(\Delta_{x'} J(x', y_{true})\big)$$

where $x' = x + \alpha \text{sign}(N(0^d, I^d))$.

By having an additional "random step" in x' this circumvents a defense method called **adversarial training** (Goodfellow et al., 2015) in its naive implementation.

L-BFGS attack (Szegedy et al., 2014) L-BFGS models the adversarial example generation process as an optimization problem that uses the L-BFGS as the optimizer.

Given a victim model f, find r that minimizes

$$\min \quad c|r| + \text{loss}_f(x+r, y)$$

subject to $x + r \in [0,1]^m$.

Remark 23.2. The loss function in L-BFGS attack need to be the same as the training loss of the victim.

Universal Adversarial Perturbation (UAP) (Moosavi-Dezfooli et al., 2016) A single perturbation that can be added to *multiple inputs* to make them adversarial.

The pseudo-algorithm is as follows:

Algorithm 3 Universal Adversarial Perturbation

```
input Data points $X$, classifier $k$, desired $l_p$ norm of perturbation $\epsilon$, desired accuracy on perturbed sample $\delta$
output Universal perturbation vector $v$
1: Initialize $v \leftarrow 0$
2: while $Err(X_v) \leq 1 - \delta$ do
3:     for each datapoint $x_i \in X$ do
4:         if $k(x_i + v) = k(x_i)$ then
5:             Compute minimal perturbation that sends $x_i + v$ to decision boundary:
6:             $\Delta v_i \leftarrow \text{argmin}_r \|r\|_2$ s.t. $k(x_i + v + r) \neq k(x_i)$
7:             Update perturbation:
8:             $v \leftarrow P_{p,\epsilon}(v + \Delta v_i)$
```

where $P_{p,\epsilon}(v) = \text{argmin}_{v'} \|v - v'\|_2$ subject to $\|v'\|_p \leq \epsilon$.

Adversarial Tranformation Network (ATN) (Baluja & Fischer, 2017) ATN trains a neural network to generate adversaries. Two variates exist: **Adversarial Autoencoder (AAE)** and **Perturbation ATN (P-ATN)**.

For 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.

Remark 23.3. Every G_t can only be used to general adversarial examples that are misclassified as class t.

FGSM, BIM, and R+FGSM are all **direct gradient step** methods.
L-BFGS and UAP are both **iterative optimization** methods.
ATN is a **parameterized optimization** method.
Blackbox adversarial attacks:

Substitute Blackbox Attack (SBA) (Papernot et al., 2016) SBA approximates the decision boundaries of the vcitim. This assumes that the attacker has access to the softmax probabilities of the classifcation.

The procedure is as follows:

1. Train substitute model on dataset **labelled by victim**
2. Attack substitute model using any whitebox methods
3. Validate that adversarial examples fool substitute model

4. Use adversarial examples to fool blackbox model

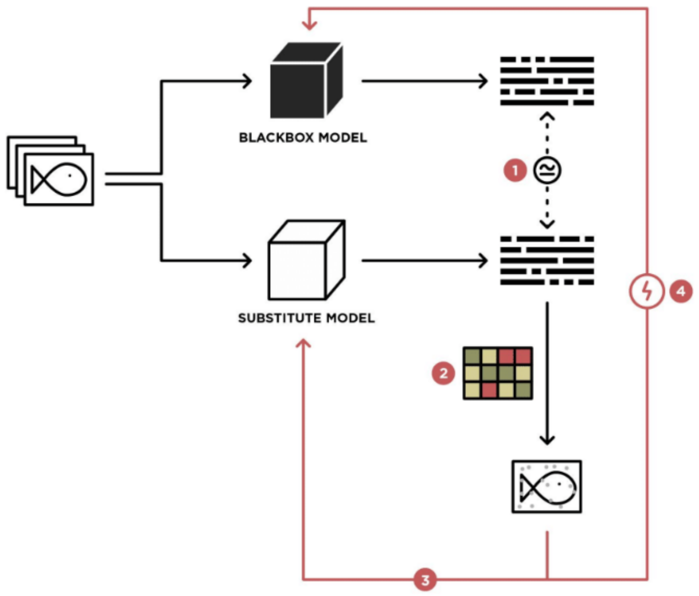


Figure 23.1: Substitute Blackbox Attack (SBA).

SBA is an example of a **decision boundary approximation** attack.

Zeroth Order Optimization (ZOO) (Chen et al., 2017) ZOO attacks via **finite difference approximation**:

$$\hat{g}_i = \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h}$$

where f is the victim model, e_i is a unit vector in coordinate i, and x_i is the ith coordinate of the input x.

ZOO uses coordinate-wise ADAM to perform stochastic coordinate descent since in general, evaluating $f(x + he_i)$ for every parameter and coordinate is quite expensive.

23.3 Adversarial defenses (Goodfellow et al., 2015)

Some methods of adversarial defenses:

1. Adversarial training: generate adversarial examples during training by also minimizing loss on inputs within ϵ-ball of input.

   For example, when using FGSM we may have the new loss function:

$$\tilde{J}(\theta, x, y) = \alpha J(\alpha, x, y) + (1 - \alpha) J(\theta, x + \epsilon \text{sign}(\Delta_x J(\theta, x, y)), y)$$

   that is: adversarial examples are generated per training iteration based on the current state.

   Note this is *not* the same as **Generative Adversarial Nets (GANs)**.

   The process of incorporating adversarial examples look something like:

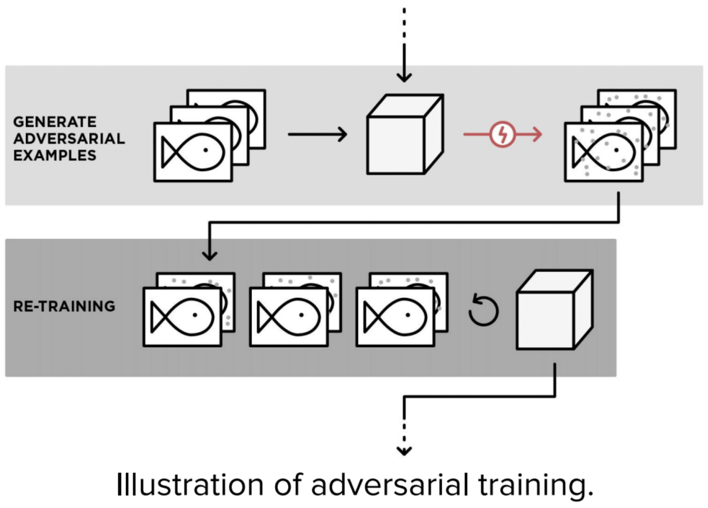
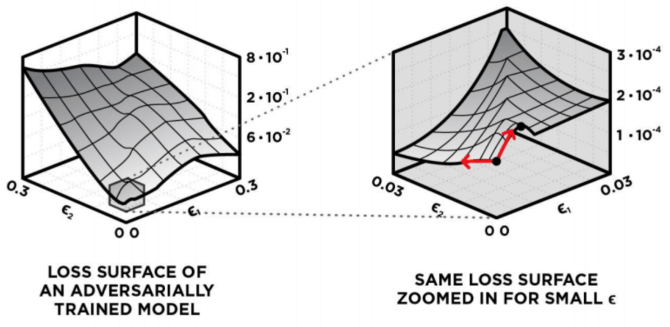

GENERATE ADVERSARIAL EXAMPLES
RE-TRAINING
Illustration of adversarial training.

Figure 23.2: Substitute Blackbox Attack (SBA).

et al., 2018) Gradient masking happens when the local gradient for a given ϵ in direction ϵ_1 appears to be larger compared to direction ϵ_2, but the loss/gradient is actually larger in the ϵ_2 direction for larger ϵ values.

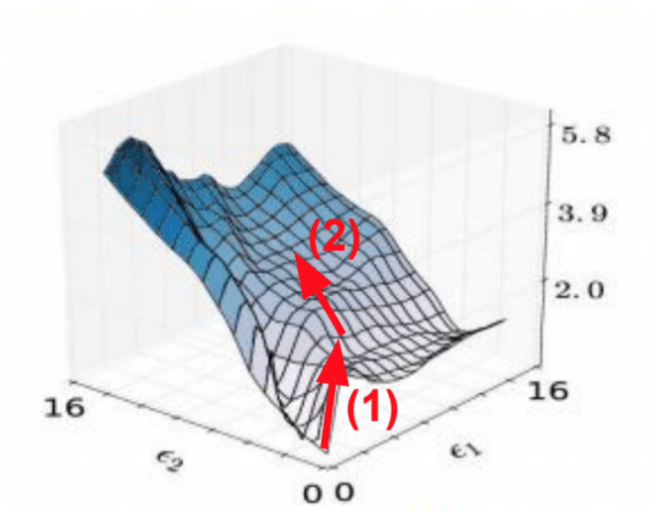
This is often unintentional but prevents models to reveal meaningful gradients:



R+FGSM Revisiting R+FGSM, it actually in essence takes a two step gradient where the first step is in a random direction. Recall (1) $x' = x + \alpha\text{sign}(N(0^d, I^d))$ and (2)

$$x^{adv} = x' + (\epsilon - \alpha)\text{sign}\big(\Delta_{x'} J(x', y_{true})\big)$$

where (1) adds small random perturbation and (2) takes the step that maximizes the loss:



(a) Loss of model v3$_{\text{adv}}$.

This can circumvent naive implementations of FGSM adversarial training.

et al., 2018) PGD adversarial training takes a robust optimization persepctive where our objective function becomes:

$$\min_{\theta} \rho(\theta)$$

where

$$\rho(\theta) = \mathbb{E}_{(x,y)\sim D}\left[\max_{\delta\in S} L(\theta, x+\delta, y)\right]$$

that is we find a set of parameter θ that **minimizes** the loss in the **worst-case scenario** (**minimax** formulation).

Authors empirically showed adversaries generated via R+BIM (which they called **Projected Gradient Descent (PGD)**) are the worst-case adversaries.

So in practice, one should perform adversarial training **only** on PGD adversaries.

Takeaways:

1. Gradient-based attacks uses the loss landscape and gradients as a cheat sheet to find adversarial (close) examples that increase loss.

2. Gradient masking may be unintentional and give false robustness

3. The field is currently very empirical, needs more work to provide guarantees on adversarial robustness (e.g. upper bound of proposed defense model)

24 March 15, 2019

24.1 Intrinsic plasticity

Instead of training weights and neurons with an error/loss, we train networks to maximize information potential of each neuron based on **local statistics**.

Whereas before in normal neural networks we have the activation $y = \theta(x)$, we instead now have the activation $y = \theta(\alpha x + k)$ where α is the sensitivity/gain parameter and k represents the bias (threshold term). x in this case was our $Wy^{(i-1)} + b^{(i-1)}$.
The update rules for the gain α and bias k are

$$\begin{aligned}\Delta\alpha &= \frac{1}{\alpha} - 2\mathbb{E}[xy] \\ \Delta k &= -2\mathbb{E}[y]\end{aligned}$$

(these rules are still being studied and may not be the most optimal).
Some issues with intrinsic plasticity:

- It is unstable
- $E[uy]$ may be ill-suited for adjusting sensitivity/gain
- May homogenise inputs too much: the input x can arbitrarily be scaled and shifted along $(-\infty, \infty)$ with α and k. What is the point of learning weights and biases?
- Competes with error-based learning of synpatic weights

24.2 Batch normalization

The shape of inputs to a layer may be radically different from one input to the next. This slows down learning since hidden layers are required to learn representations and distributions.
Batch normalization normalizes the distribution of each batch of input to each layer to 0 mean and unit variance. The steps are:

$$\begin{aligned}\mu_B &= \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma_B^2 &= \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \\ \bar{x}_i &= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ y_i &= \gamma \bar{x}_i + \beta\end{aligned}$$

where γ, β are learned parameters and y_i is the new normalized input.

24.3 Batch norm and intrinsic plasticity

Note that if we applied a similar de-normalization to the intrinsic plasticity activation, we get an equivalent model:

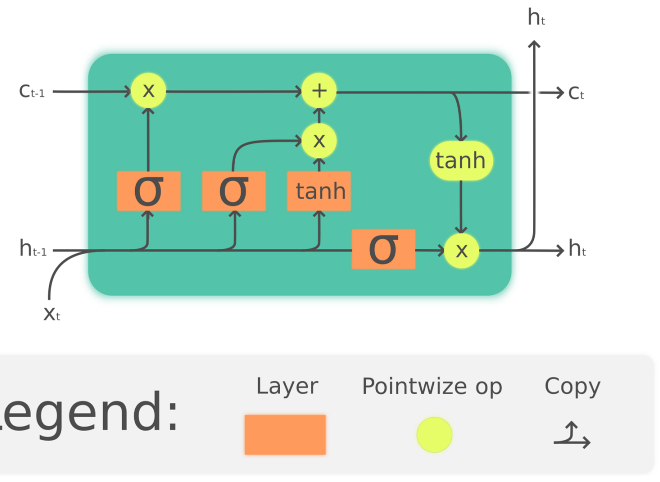
$$\begin{aligned}y &= \theta(\gamma(\alpha x + k) + \beta) \\ y &= \theta(\gamma(\frac{1}{\sqrt{\sigma^2 + \epsilon}}x + \frac{-\mu}{\sqrt{\sigma^2 + \epsilon}}) + \beta)\end{aligned}$$

The benefits of both batch normalization and intrinsic plasticity is that it brings the input close to mean 0 which remedies the vanishing gradient problem for sigmoid activation functions.

25 March 18, 2019

25.1 More on LSTM

An LSTM cell looks something like:



The **forget gate** modulates how much of the hidden cell state c_{t-1} gets propagated:

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

The current input to be introduced $\tilde{c}_t$ into the hidden state c_t is constructed from x_t:

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

then the **input gate** modulates how much of the current input $\tilde{c}_t$ gets introduced into the hidden state c_t:

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

combining the input with the non-forgotten state we end up with the new hidden state:

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ &= \sigma(W_f X_t + b_f) \odot c_{t-1} + \sigma(W_i X_t + b_i) \odot \tanh(W_c X_t + b_c) \end{aligned}$$

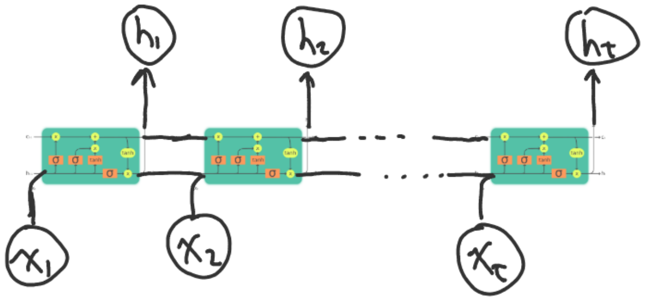
where $X_t = (h_{t-1}, x_t)^T$.
Finally the **output gate** modulates how much of the hidden state c_t is outputted as h_t:

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

where the output is

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

How does the LSTM cell help with error gradient backpropagation? Consider the LSTM unrolled through time:



Note that the gradient for any hidden state at time step t is dependent on the immediate output state h_t and the next hidden state c_{t+1}:

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

Note that the forget gate is precisely $f_{t+1} = \frac{\partial c_{t+1}}{\partial c_t}$, so we have

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

where we only have at worst a product of a chain of forget gates per term. Suppose the forget gate is close to 1 and the input gate is 0. Then

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

26 March 20, 2019

26.1 Population and neural coding

Recall the firing rate of an LIF neuron:

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

If a neuron's activity changes as the value changes, then we say that neuron **encodes** that quantity.
There are many intermediate processes between the environment value and the neurons that encode it. We will approximate all those processes using a linear remapping:

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

where:

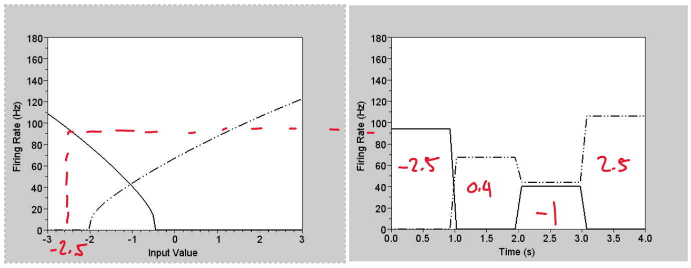
- $\alpha \in \mathbb{R}_{\geq 0}$: "gain"

- $e \in \{-1, 1\}$: "encoder"
- $\beta \in \mathbb{R}$: "bias"

$J(x)$ is the current entering the neuron for stimulus x, hence an LIF neuron is characterized by 6 parameters:

1. J_{th}: threshold current (always $= 1$)
2. τ_m: membrane time constant
3. τ_{ref}: refractory period
4. α: gain
5. β: bias
6. e: encoder

From a set of **tuning curves** (maps firing rate vs input value), we can inversely map firing rates back to their values given a time series of firing rates, e.g.:



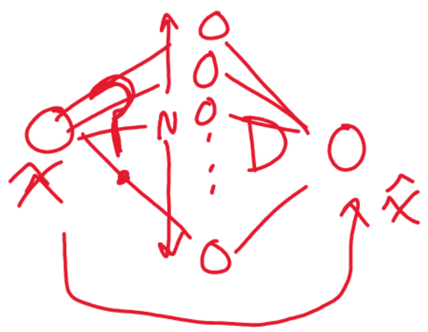
This process of mapping firing rates back to their input values is called **decoding**.
Ideally we can decode with only a small number of neurons, but neuronal firing rates appear to have a *stochastic component* (at least with respect to the variable of interest). Thus more observations will improve the statistics of our inference.
One way to decode neural activity in general is called **optimal linear decoding**. Suppose we have an array of neural activities over a range of input values:

- $X \in \mathbb{R}^{P \times 1}$ is a column vector containing our input x values
- $A \in \mathbb{R}^{P \times N}$ where each row contains activities (firing rates) for N observed neurons for the corresponding x value

We want to find a linear transformation i.e. set of decoding weights $D \in \mathbb{R}^{N \times 1}$ such that

$$AD \approx X$$

i.e. we are essentially find the decoder in an autoencoder where X was essentially encoded as A by some unknown weights:



thus our objecitve function is

$$\min_D \|AD - X\|_2^2$$

To solve this, there are 2 methods:

Least squares By solving for the normal equation for D by taking the derivative of $(AD-X)^T(AD-X)$ with respect to D:

$$\begin{aligned} &A^T(AD - X) = 0 \\ \Rightarrow &A^T AD = A^T X \\ \Rightarrow &D = (A^T A)^{-1} A^T X = A^\dagger X \end{aligned}$$

where $A^\dagger$ is the matrix pseudoinverse of A.

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

SVD We let $A = U\Sigma V^T$ where U, V are orthogonal and Σ is diagonal.

Then

$$\begin{aligned} &Ax = b \\ \Rightarrow &U\Sigma V^T x = b \\ \Rightarrow &x = V\Sigma^{-1}U^T b \end{aligned}$$

so

$$D = V\Sigma^{-1}U^T X$$

This method is more expensive but more numerically stable.

27 March 22 and March 25, 2019

27.1 Behaviourial sampling and applying neural coding

Suppose we wanted to figure out what stimulus an animal is receiving based just on its neural activity:

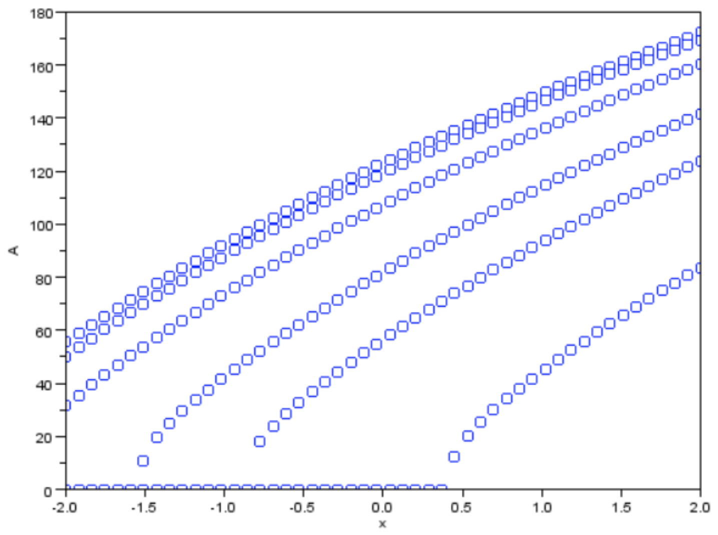
1. Choose a sampling of stimulus and store in $X \in \mathbb{R}^{P\times 1}$
2. Present each stimulus to the animal and measure neural responses and store in $A \in \mathbb{R}^{P\times N}$
3. Solve for the decoder
$$D = \text{argmin}_D \|AD - X\|_2^2$$

4. Now we can predict the stimulus from only the neural activities

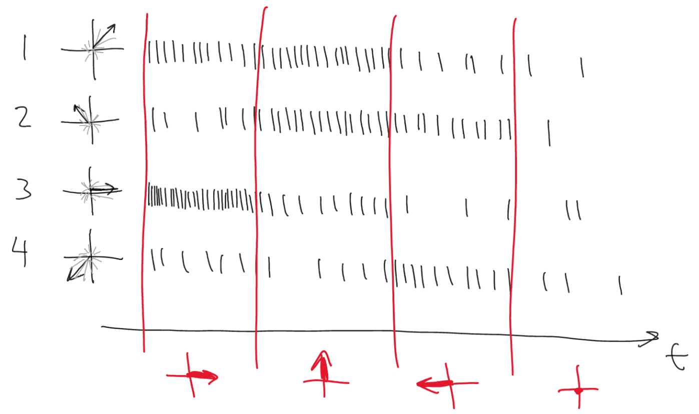
Note that we can decode other functions of X too: to decode $f(X)$ we simply solve

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

However, there is some noise we have to deal with: sometimes we may get changing tuning curves:



Example 27.1. Georgopoulos published "Neural Population Coding of Movement Direction" in Science which showed spike rasters for probed neuron's in a monkey's motor cortex in terms of the direction the monkey's hand moved:



The length of the lines in the Cartesian plot indicates firing rates when hand moves in that direction. We can thus decode the firing rates in terms of the directions a monkey's hand moves in.
One approach to decoding this is: hand movement can be predicted as a **weighted sum of the preferred direction vectors**, weighted by each neuron's firing rate. That is

$$\begin{aligned}\text{hand movement} &= a_1\vec{x}_1 + a_2\vec{x}_2 + a_3\vec{x}_3 + a_4\vec{x}_4 \\ &= \sum_n a_n\vec{x}_n\end{aligned}$$

where a_n are the firing rates and $\vec{x}_n$ is the preferred direction vector.
This works in some cases when the neurons are **uniformly distributed over the directions**. For other cases we need a more statistically principled approach.

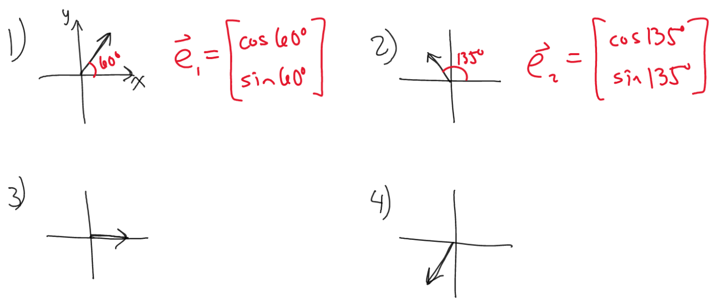
If a neuron's activity is influenced by 2 variables, then we can formulate that relationship using a **2D encoding vector**.
Recall: the injected current for 1D encoding was computed as:

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

where $\alpha \geq 0$, $e \in \{-1, 1\}$, $x \in \mathbb{R}$, and $\beta \in \mathbb{R}$.
For 2 external variables (x, y)

$$J(x,y) = \alpha(e_1, e_2) \cdot (x, y) + \beta$$
$$\Rightarrow \alpha \vec{e} \cdot \vec{x} + \beta$$

where $\|\vec{e}\| = 1$.
In the case of the monkey's arm movement:


1)
y
x
60°
$\vec{e}_1 = \begin{bmatrix} \cos 60° \\ \sin 60° \end{bmatrix}$
2)
135°
$\vec{e}_2 = \begin{bmatrix} \cos 135° \\ \sin 135° \end{bmatrix}$
3)
4)

In fact our neurons could depend on K variables i.e. $\vec{x} \in \mathbb{R}^K$. The same formula still holds:

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

but with encoders $\vec{e}_n \in \mathbb{R}^K$.
Suppose we have measured the neuron activities for a bunch of $\vec{x}$-values: $X \in \mathbb{R}^{P \times K}$ is a $P \times K$ matrix containing one set of of K variables in each row; $A \in \mathbb{R}P \times N$ as before. And as before we seek linear decoding weights D that yield the last-squares solution:

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

where in this case $D \in \mathbb{R}^{N \times K}$. As before we can decode functions of $\vec{x}$:

$$\min_D \|AD - f(X)\|_2^2 \qquad f : \mathbb{R}^K \to \mathbb{R}^L, D \in \mathbb{R}^{N \times L}$$

Example 27.2. Given (x_P, y_P) sitmulus and corresponding neural activities $(a_{P_1}, a_{P_2}, \ldots, a_{P_N})$ for N neurons we have $X \in \mathbb{R}^{P \times 2}$ and $A \in \mathbb{R}^{P \times N}$.
We want to decode $f(x, y) = xy$, so we compute $Q \in \mathbb{R}^{P \times 1}$ where $q_P = x_P y_P$ and solve:

$$\min_D \|AD - Q\|_2^2$$

27.2 Neural decoding summary

To perform neural decoding, we follow the steps:

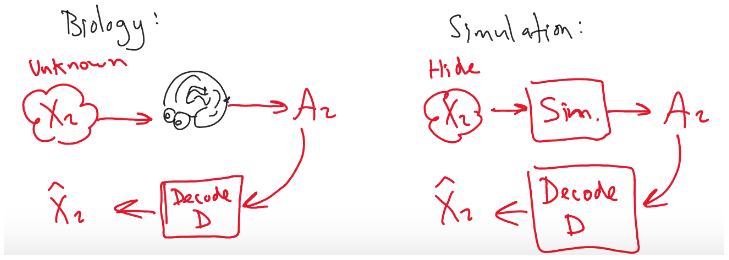
1. **Behaviourial sampling**: in **Biology**, we provide the brain with **stimulus** X and observe **neural activitiies** A.

   In a **simulation**, we choose X and we simulate to get A.

2. **Compute decoders**: use data from behaviourial sampling to solve

$$D = \text{argmin}_D \|AD - f(X)\|_2^2$$

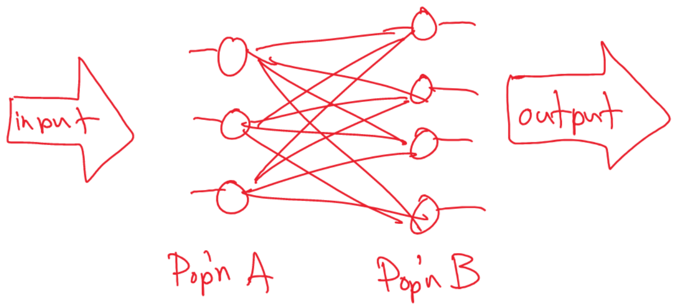
3. **Decode neural activity**: now we can decode further neural activity i.e. given A_2 we can estimate X_2:



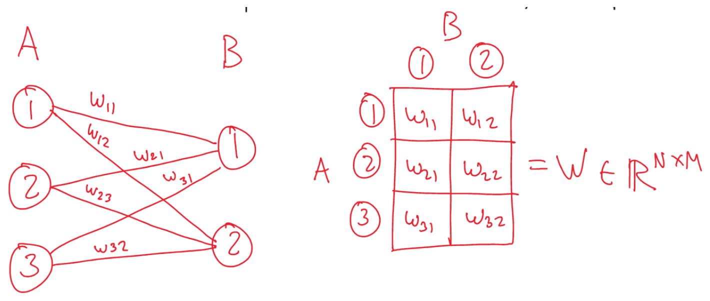
28 March 27, 2019

28.1 Neural transformations

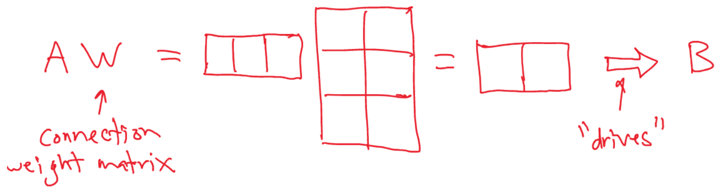
So far we've only looked at interpreting the activity of a population of neurons, but we know neurons send their outputs to other neurons:



Suppose we have 2 populations A and B (as above) where A has N neurons and 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 these connections?



This matrix notation is convenient for a reason. Let's store the activities of a population of neurons as vectors $A = (a_1, a_2, a_3)$, $B = (b_1, b_2)$ (row vectors).
Then to find out how A influences B we use:



Remark 28.1. We can equivalently store A, B as column vectors where we have $W^T A^T \Rightarrow B^T$ instead. We will stick to the convention of row vectors.

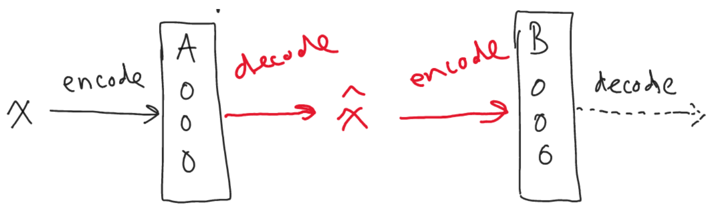
Let's start with a simple example: we want B to encode the same value that is encoded in A.

Question 28.1. Can we simply copy the neural activities from A to B i.e. $b_1 = a_1, b_2 = a_2, \ldots$?

Answer. No:

1. Encoders are probably different
2. A and B have different number of neurons

A different approach: decode from A then re-encode from B:



That is:

1. Encode x into A:
$$A = G(J(x)) = G(xE_A\alpha_A + \beta_A) \in \mathbb{R}^{1 \times N}$$
where $x \in \mathbb{R}^{1 \times k}$, $E_A \in \mathbb{R}^{k \times N}$, $\alpha_A \in \mathbb{R}^{N \times N}$ (diagonal) and $\beta_A \in \mathbb{R}^{1 \times N}$.

2. Decode the value from A (identity decoders)
$$\hat{X} = AD_A$$

3. Re-encode into B:

$$\begin{aligned} B = G(J(\hat{X})) &= G((AD_A)E_B\alpha_B + \beta_B) \\ &= G(A(D_AE_B\alpha_B) + \beta_B) \\ &= G(AW + \beta_B) \end{aligned}$$

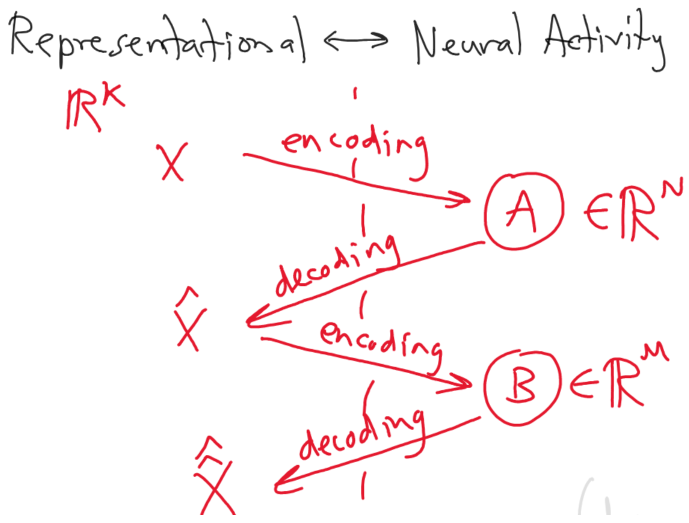
where $A \in \mathbb{R}^{1\times N}$, $D_A \in \mathbb{R}^{N\times k}$, $E_B \in \mathbb{R}^{k\times M}$, $\alpha_B \in \mathbb{R}^{M\times M}$ (diagonal), $\beta_B \in \mathbb{R}^{1\times M}$ and $W \in \mathbb{R}^{N\times M}$.

4. Decode from B (if desired):

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

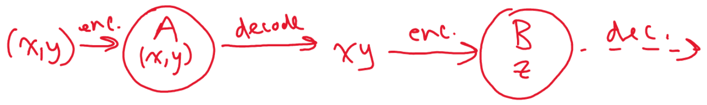
where $B \in \mathbb{R}^{1\times M}, D_B \in \mathbb{R}^{M\times k}$.

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



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

Example 28.1. Given input (x, y) we want to build a network that encodes xy (the product):



1. Encode (x, y) into A where

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

and $x \in \mathbb{R}^{1\times 2}$.

2. Decode xy from A:

$$\hat{Z} = AD_A$$

where $D_A = \text{argmin}_D \|AD - Z\|_2^2$ and $Z = X_{P,1}X_{P,2} \in \mathbb{R}^{P\times 1}$.

3. Re-encode $\hat{Z}$ into B:

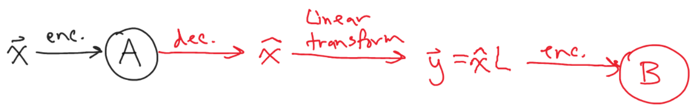
$$B = G(AD_AE_B\alpha_B + \beta_B) = G(AW + \beta_B)$$

4. Decode from B (if desired):
$$\hat{\hat{Z}} = BD_B$$

Linear transformations: consider the case when 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

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

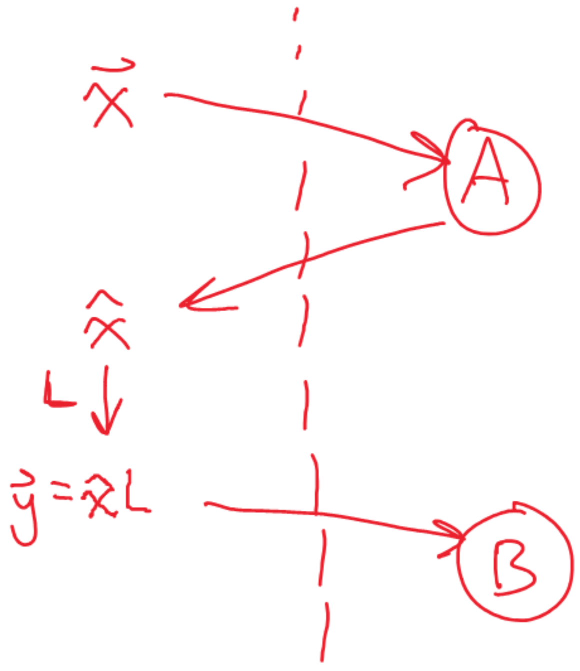
where $L \in \mathbb{R}^{k_1 \times k_2}$:



As before we can combine encoders and decoders:

$$B = G(AD_A LE_B \alpha_B + \beta_B) = G(AW\alpha_B + \beta_B)$$

We can compute these connection weights using the identity decoders, by "sandwiching" L between the decoders and encoders:



Example 28.2. Given populations A, B both 2-dimensional, given their encoder and decoders, write the weight matrix that will rotate the vector in A by 60^o and encode that value in B.
Rotation is a linear operation where:

$$R_\theta = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \Rightarrow R_{60^o} = \begin{bmatrix} 1/2 & \sqrt{3}/2 \\ -\sqrt{3}/2 & 1/2 \end{bmatrix}$$

thus we have $W = D_A R_{60^\circ} E_B \alpha_B$.

29 March 29, 2019

29.1 Network dynamics

How does an LIF network behave?
The dynamic model of a leaky integrate-and-fire (LIF) neuron contains two time-dependent processes:

$$\begin{aligned} \tau_s \frac{dJ}{dt} &= -J + uW + \beta && \text{current} \\ \tau_m \frac{dv}{dt} &= -v + G(J) && \text{neural activity} \end{aligned}$$

where τ_s, τ_m and G are the synapse time constant, membrane time constant, and activation function, respectively.
Depending on the time constant:

$\tau_m << \tau_s$ Neuron membranes react quickly (reach steady-state firing rate quickly) compared to synpatic dynamics:

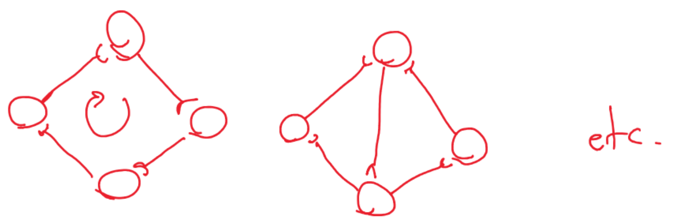
$$\begin{aligned} \tau_s \frac{\partial J}{\partial t} &= -J + uW + \beta \\ v &= G(J) \end{aligned}$$

$\tau_s << \tau_m$ If pre-synpatic current changes quickly compared to firing rate:

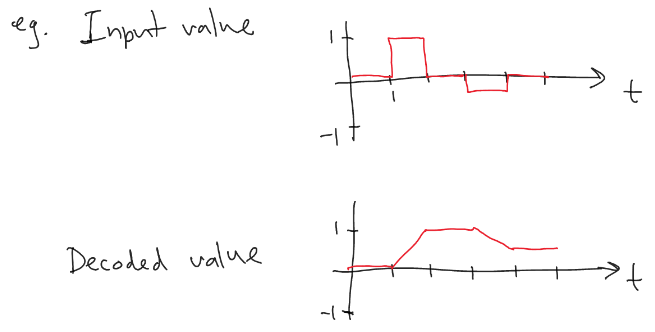
$$\begin{aligned} J &= uW + \beta \\ \tau_m \frac{\partial v}{\partial t} &= -v + G(J) = -v + G(uW + \beta) \end{aligned}$$

Steady state We have $v = G(uW + \beta)$ in steady state.

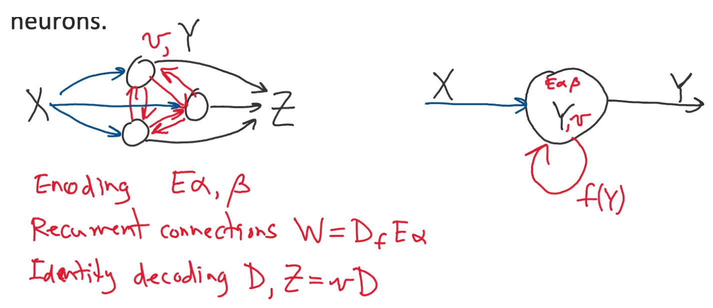
If the neurons of a popuation are connected to other neurons in the *same population* we say the network is **recurrent**.
In particular recurrent networks can have circuits:



This feedback leads to interesting dynamics.
Recall the LIF integrates over the input current to decode its value:



Now consider a fully-connected population of N neurons:



For some initial state v, suppose $Y = vD$. If there is no input (i.e. $X = 0$) then we want our population to **maintain its value** due to its recurrence, so:

$$\begin{aligned}
v &= G(vW + \beta) \\
v &= G(vD_f E_\alpha + \beta) \\
vD &= G(vD_f E_\alpha + \beta)D = Y = G(YE\alpha + \beta)D
\end{aligned}$$

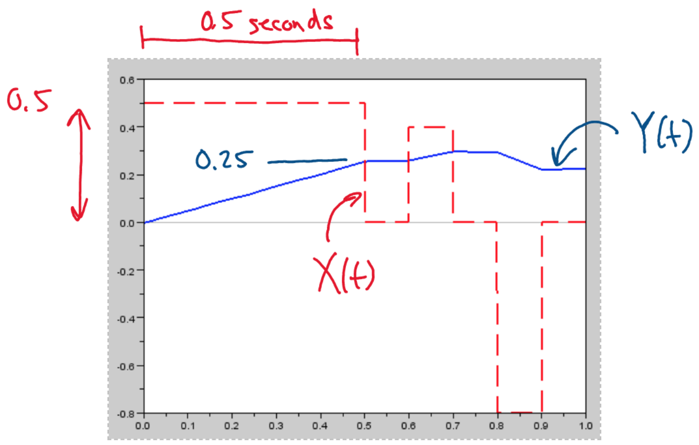
thus $D_f = D$ are identity decoders.
What about integrating the input? We let the time dynamic be dictated by the synaptic time constant. Let $\tau_m = 0$ so we set $v = G(J)$ and thus

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

To get the input current J_i we need to encode the input value X: i.e. $\tilde{J}_i = XE\alpha$ so

$$\begin{aligned}
\frac{dJ}{dt} &= \frac{vW + B - J}{\tau_s} + \tilde{J}_i \\
\tau_s \frac{dJ}{dt} &= vW + \beta - J + \tau_s X E_\alpha \\
&= (vD + \tau_s X)E_\alpha + \beta - J \\
&= (Y + \tau_s X)E_\alpha + \beta - J
\end{aligned}$$

where Y is the identity feedback and $\tau_s X$ is the input scaled by synaptic time constant. We thus have for a **recurrent LIF** the output:



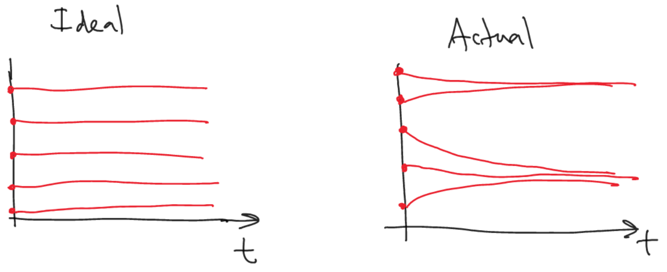
Let's test how well our dynamic network maintain its value. To do this we'll run the following funciton, passing it an initial state that we want it to hold:

$$V = \text{Dynamics}(x, \text{d}t, lif, w, \tau_s, \tau_m, V_0 = None)$$

where x is the input $x(t)$ and V_0 is the initial input (optional).

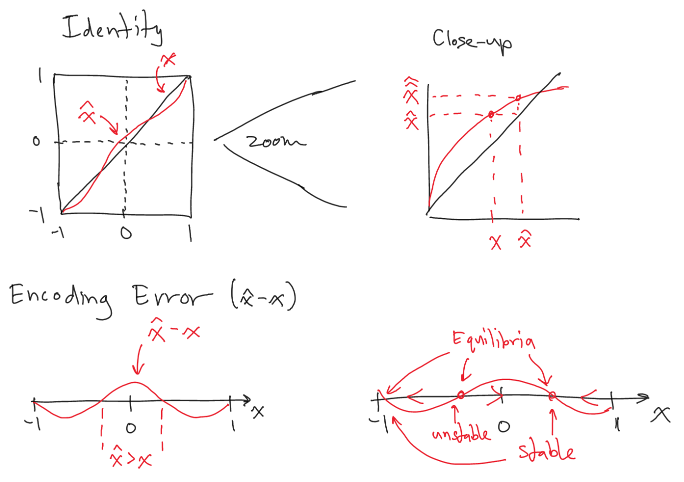
1. Choose an x-value to hold
2. Use x to find steady-state firing rates i.e. $v = G(J(x)) \to V_0$
3. Call the Dynamic function

Surprisingly, we get something very noisy:



Question 29.1. What went wrong?

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



30 April 1, 2019

30.1 Convolutional neural networks

Most of the networks we have looked at assumes an all-to-all connectivity between populations of neurons, like between layers in a network; however this is not exactly how our brains are wired. Neurons in a real brain tend to be sparsely connected.
For example, a neuron in the visual vortex V1 is only activated by a small patch in the visual field. This topological mapping between 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:

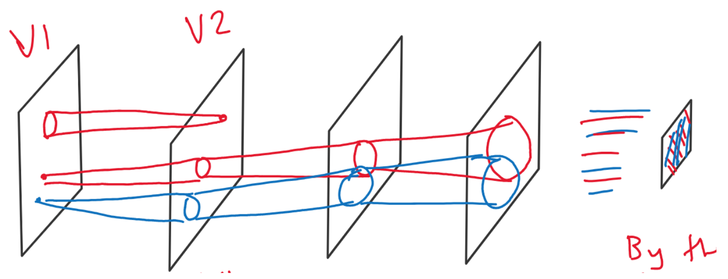


Figure 30.1: The "footprint" of a region in the visual field becomes larger as you progress up through the layers. By the last layer then neurons are influenced by the whole visual field.

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)**.

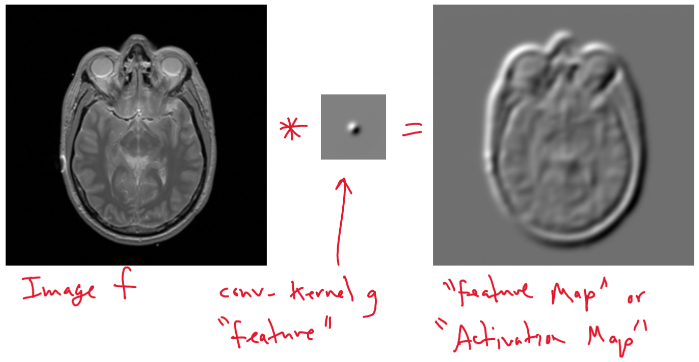
Definition 30.1 (Convolution)**.** Given $f, g : \mathbb{R} \to \mathbb{R}$ (continuous domain), then we define a mathematical **convolution** is

$$(f \star g)(x) = \int_{-\infty}^{\infty} f(s)g(x-s)\,\mathrm{d}s$$

And for $f, g : \mathbb{R}^N \to \mathbb{R}$ (discrete domain) we have

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

Given an image (f) and a convolutional kernel g, the resulting convolution maintains its topological properties:



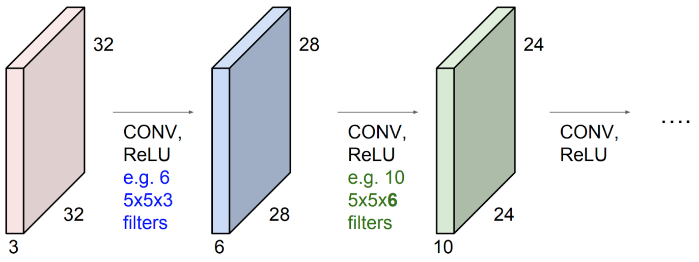
Remark 30.1. Note that in 2D (i.e. $f, g : D \to \mathbb{R}^2$) we have

$$(f \star g)_{m,n} = \sum_i \sum_j f_{mn} g_{m-i,n-j}$$

Convolution Layer Given a $W \times H \times C$ input (width, height, and # of channels e.g. RGB), which could be some image, a convolution layer consists of N kernels K_i of size $F \times F \times C$. For a **stride of 1**, we take the dot product betwen K_i and every $F \times F \times C$ partition of the input starting from the top left corner. Each dot product produces a real value. We then shift this kernel one to the right until we reach the rightmost edge, then begin the convolution again by shift K_i one row down. With a stride of s, we shift the kernels by F units every time (both horizontally and vertically). Thus each kernel produces one output channel and so N kernels produces N output channels. Thus the output size of the layer is:

$$\lfloor \lceil (W - F)/s + 1 \rceil \rfloor \times \lfloor \lceil (H - F)/s + 1 \rceil \rfloor \times N$$

Note that $\lceil (W - F)/s + 1 \rceil$ may not be intergral depending on W, F, s. In practice it is common to zero padding the image with an edge of size $(F - 1)/2$. With a stride of 1 this exactly preserves W and H.



31 April 3, 2019

31.1 Vector embeddings

We have been using vectors to represnt inputs and outputs, where “2” might be encoded as $[0, 0, 1, 0, \ldots] \in \{0, 1\}^{10}$ and “e” might be encoded as $[0, 0, 0, 0, 1, \ldots] \in \{0, 1\}^{26}$ (these are one-hot encodings).
What about words? Consider the set of all words encountered in the dataset. We call this our **vocabulary**. Let’s order and index our vocabulary and represent words using one-hot vectors like above. Let $word_i$ be the ith word in

the vocab i.e. "cat" $\approx v \in \mathbb{W}$ where $\mathbb{W} \subseteq \{0,1\}^{Nv} \subseteq \mathbb{R}^{Nv}$ where Nv is the number of words in our vocabulary (e.g. 70000). Then $v_i = 0$ if $word_i \neq$ "cat" and $v_i = 1$ if $word_i =$ "cat".
This works, 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? For example in "CS489 is *interesting*" and "CS489 is *fascinating*", *interesting* is similar in meaning to *fascinating*.
We could form **synonym groups**, but where do we draw the line when words have similar but not identical meanings e.g. content, happy, elated, ecstatic?
These issues reflect the semantic relationship between words. We would like to find a different representation for each word, but that also incorporates their semantics.
We can get a lot of information from teh simple fact that some words often **co-occur together** (or nearby) in sentences.

Example 31.1. "Trump returned to Washington Sunday night though his wife Melanie Trump stayed behind in Florida."
"Trump" occurs in general with "Washington" often. Similary "Sunday" and "night" and "Melanie" and "Trump".

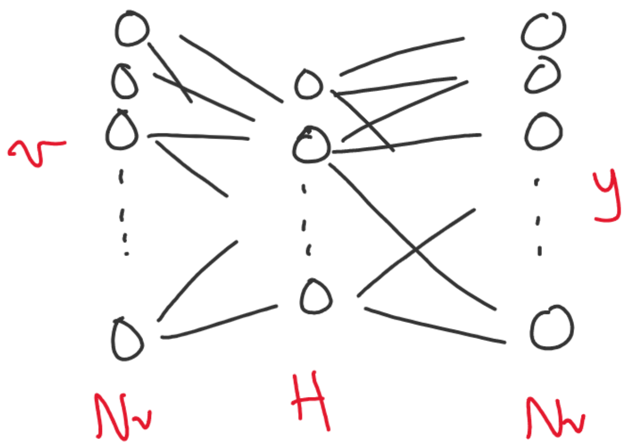
This is the idea behind **distributional representation of words**.
For the purpose of this topic, we consider "nearby" to be within words.

Example 31.2. If we let $d = 2$ and our current word is "night" in the sentence: "Trump returned to Washington Sunday night though his wife Melanie Trump stayed behind in Florida", then this gives us the word pairings with "night": (night, Washington), (night, Sunday), (night, though), (night, his).

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. That is our neural network performs

$$\begin{aligned} &y = f(v, \theta) \qquad v \in \mathbb{W} \\ &y \in \mathbb{P}^{Nv} = \{p \in \mathbb{R}^{Nv} \mid p \text{ is a probability vector}\} \end{aligned}$$

i.e. $\sum_i p_i = 1,\ p_i \geq 0$.
Then y_i equals the probability that $word_i$ is nearby our current word v. The neural network looks like:

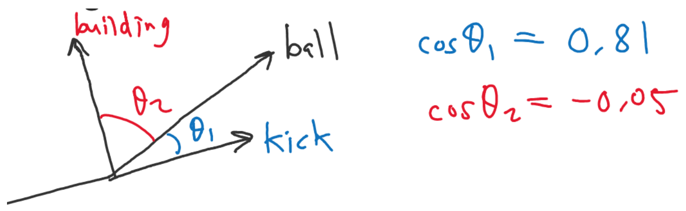


The hidden-layer being smaller forces a compressed representation (e.g. $k = 100$), requiring similar words to take on similar representations. This is called an **embedding**.

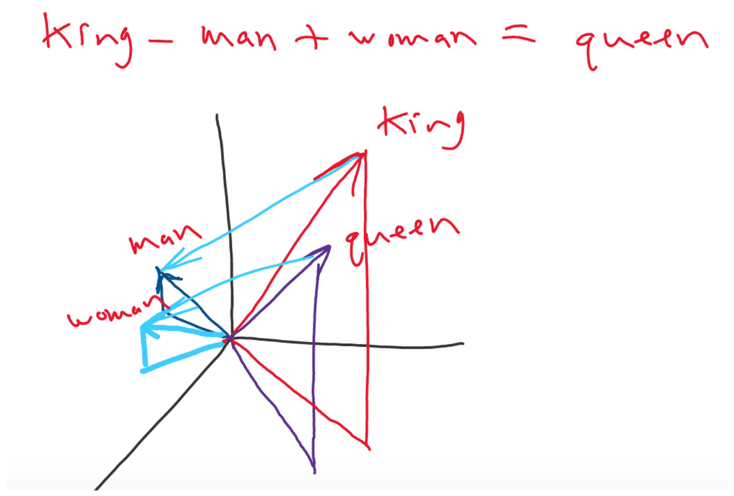
Word2vec Word2vec is a popular embedding strategy for words (or phrases or sentences). It uses additional tricks to speed up learning. Also:

1. Treats common phrases as new words e.g. "New York" is one word
2. Randomly ignores very common words e.g. "the car hit the post on the curb": of the 56 possible word pairs only 20 do not 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. We have seen this before in **Self Organizing Maps (SOM)**. This words because words with similar meaning likely co-occur with the same set of words, so the network should produce similar outputs since they have similar hidden-layer activations.
The **cosine angle** is often used to measure the "distance" between two vectors:



To some extent one can perform vector arithmetic on these operations:


king − man + woman = queen
king
queen
man
woman