richardwu.ca

CS 489/689 Course Notes

Advanced Topics in CS (Neural Networks)

Jeff Orchard • Winter 2019 • University of Waterloo

Last Revision: February 27, 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.

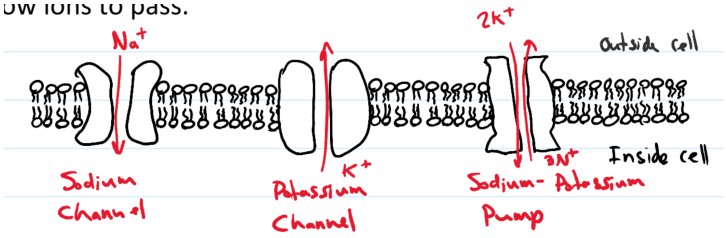


Figure 1.1: Cell membrane with Na+/K+ channels and a sodium-potassium pump. Ions move across the membrane via the channels and pump.

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

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

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.

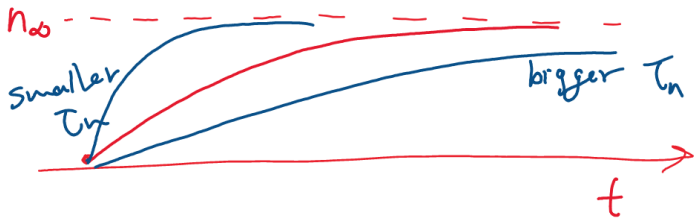
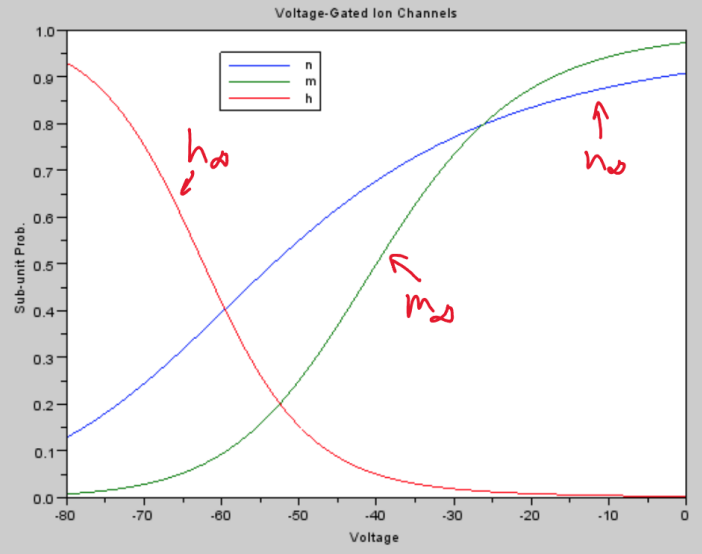


Figure 1.2: n in fraction of K+ channels open over time (for a fixed V). The blue graphs correspond to larger and smaller time constants t_n.

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

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

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 equilbrium 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

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

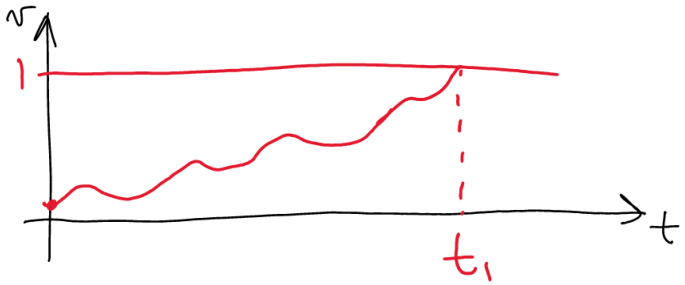
Note that $g_L = \frac{1}{R}$ where R is the resistance, thus we have

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

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

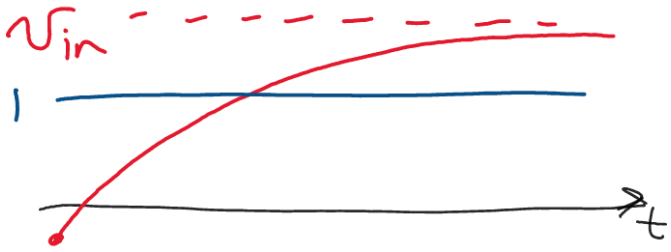


after which a spike is recorded and we reset the voltage to 0 again (after which we solve the DE for the next spike). There is a refractory period before it can spike again.
What is the firing rate if we held v_{in} constant? We need to solve for the DE analytically

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

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

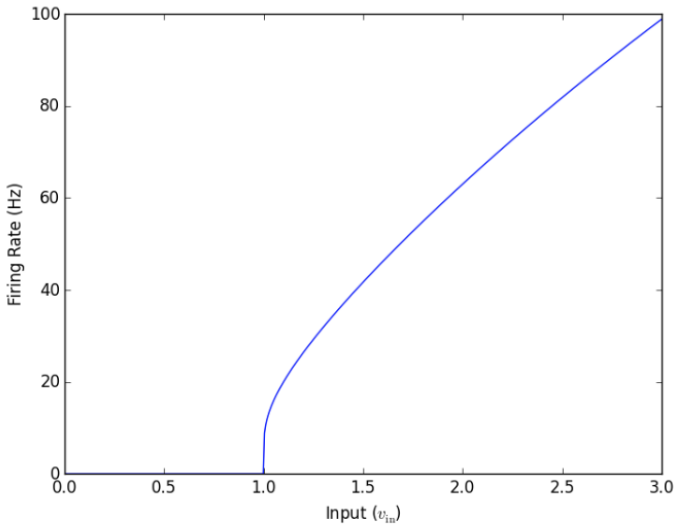
where τ_{ref} is the refractory time constant and t^* is the time for v to reach 1.
We need to find t^* where $v(t^*) = 1$. From our above solution

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

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}



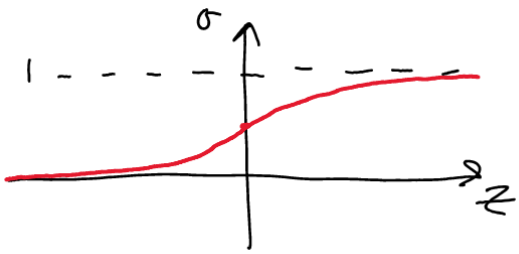
3 January 11, 2019

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**:

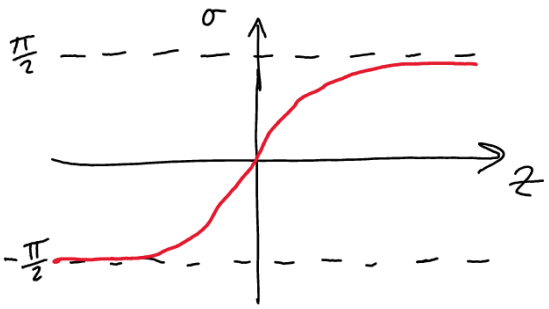
Logistic curve

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



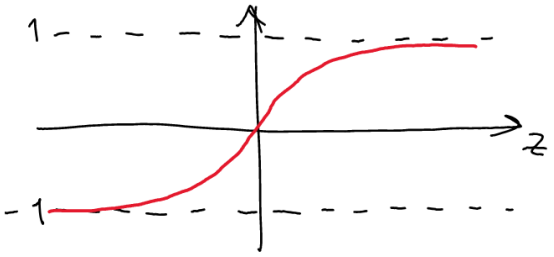
Arctan

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



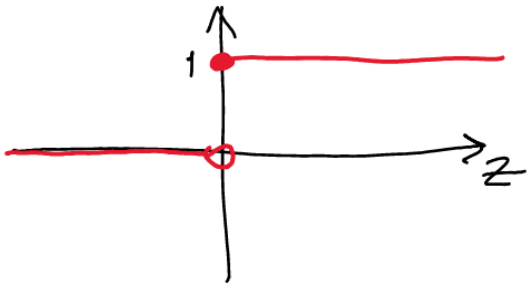
Hyperbolic tangent

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



Threshold

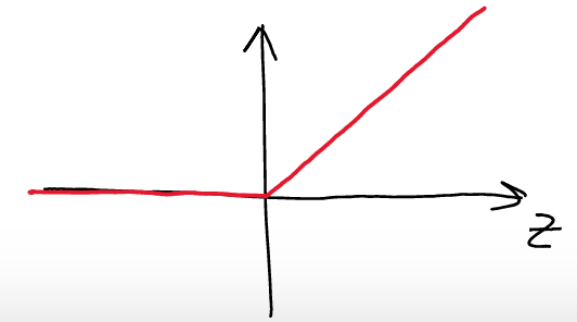
$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



3.2 Rectified Linear Unit (ReLU)

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

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

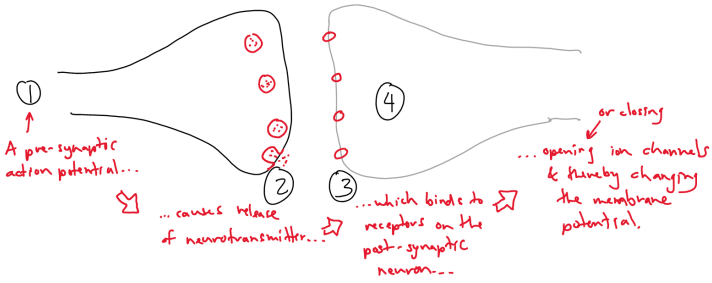


4 January 14, 2019

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.

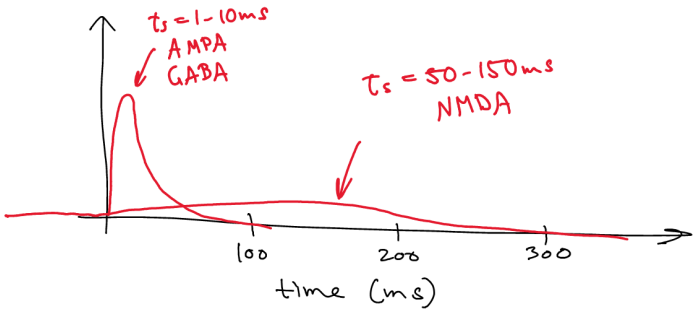


While the action potential is very fast, the synapse process can take from 10ms to over 300ms.
If we represent the time with constant τ_s then the **post-synpatic current (PSC)** (or post-synaptic potential (PSP)) entering the post-synpatic neuron is

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

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



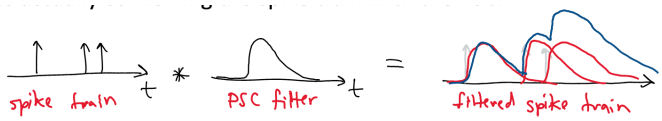
Multiple spikes (from multiple action potentials) form a **spike train** which is modelled as a sum of Dirac delta functions $a(t) = \sum_p \delta(t - t_p)$ where the Dirac delta function is defined as

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

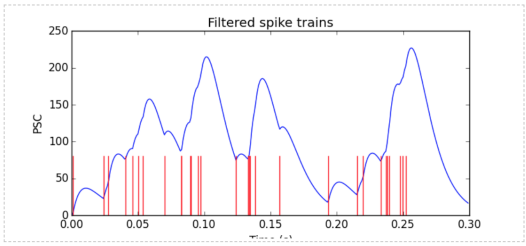
and

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

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

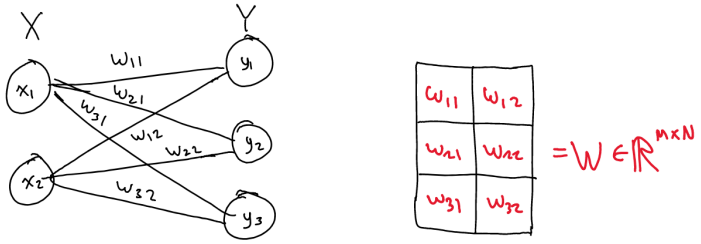
5.1 Connection weights

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**



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

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

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 $n = 0$

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

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

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

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

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

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

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

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

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

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:

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

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:

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

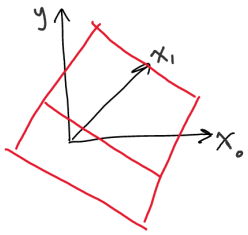
Thus $w=[0.6,-0.2]+1[0,1]=[0.6,0.8]$ and $b=-0.1+1(1)=0.9$.

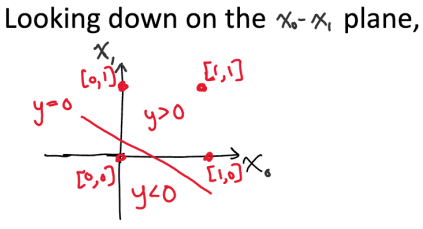
Input [0,0], target 0 We have $[0.6,0.8]\cdot[0,0]+0.9=0.9$ so $\sigma(0.9)=1$ and $E=-1$.

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 $E=0$.

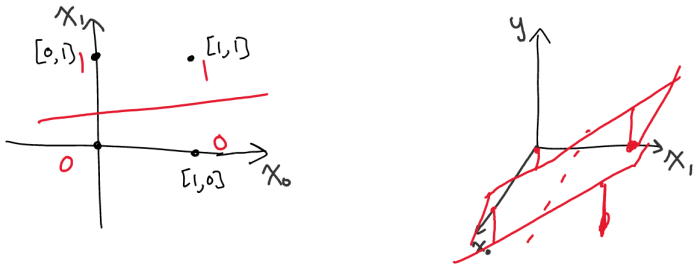
Eventully we note that when $w=[0.6,0.8]$ and $b=-0.1$ this neuron satisfies our desired behaviour.
There is in fact a geometric intepretation suppose x_1 and x_2 are free and we have $y=0.6x_1+0.8x_2-0.1$ a linear equation. In $\mathbb{R}^3$ we end up with a plane that separates $y<0$ and $y\geq 0$



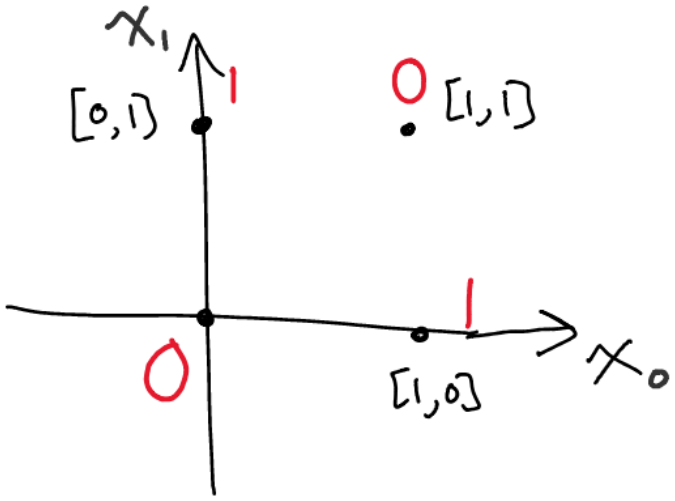


That is: find the weights and biases for our perceptron is the same as finding a **linear classifier**, a linear function that returns a positive value for the inputs that should yield a 1 and a negative value for the inputs that should yield a 0.

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

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

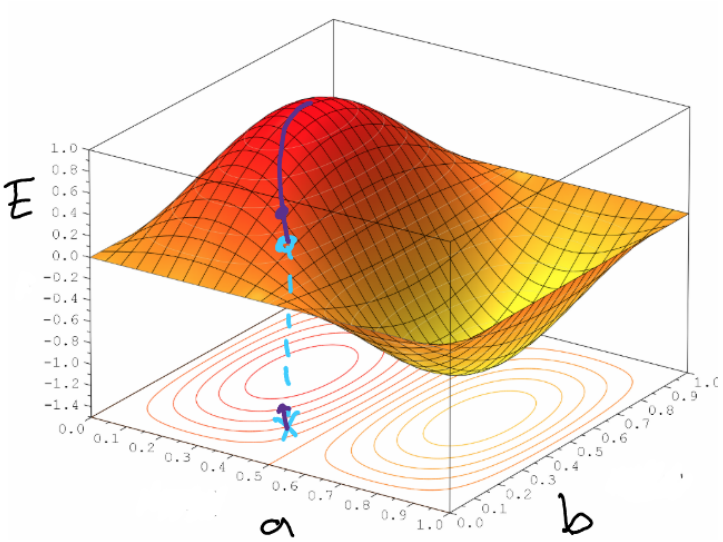
If our cost function is $E(\vec{y}, \vec{t})$ where $\vec{t}$ is the target, then the neural learning becomes an optimization problem where

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

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

$$\nabla_\theta E = \left(\frac{\partial E}{\partial \theta_0} \quad \frac{\partial E}{\partial \theta_1} \quad \dots \quad \frac{\partial E}{\partial \theta_p} \right)^T$$

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.

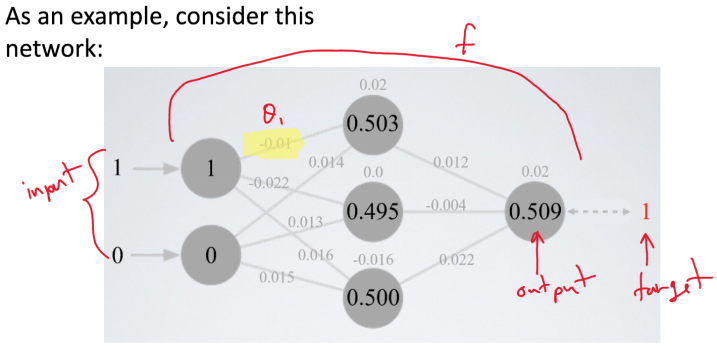
Remark 7.1. There is no guarantee one will actually find the *global optimum*. In general gradient ascent/descent will find a local optimum that may or may not be the global optimum.

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

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

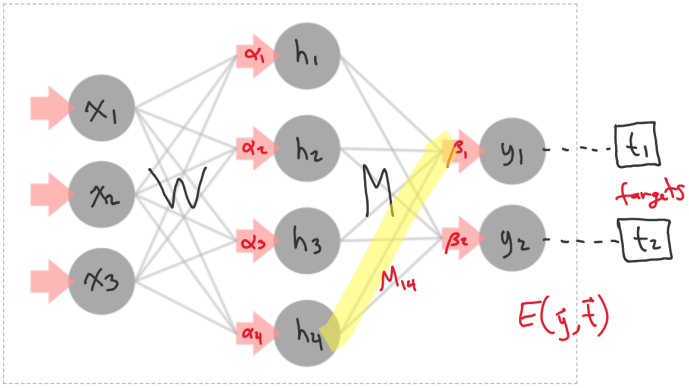
Obviously *increasing* θ_1 seems to be the right thing to do to minimize $\bar{E}$, thus

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

8 January 25, 2019

8.1 Error backpropagation

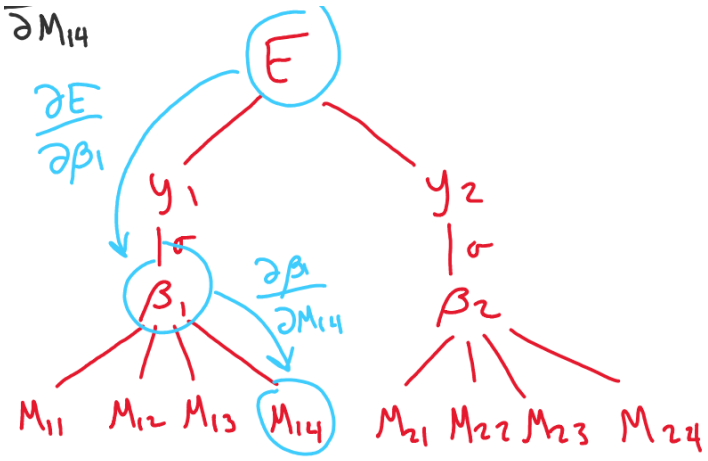
Instead of approximating the gradient using finite differencing, we can instead compute the actual gradient of our entire multi-layer network then run gradient descent appropriately. We can use the chain rule to compute the gradients from the loss at the end to any arbitrarily layer in the network and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

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

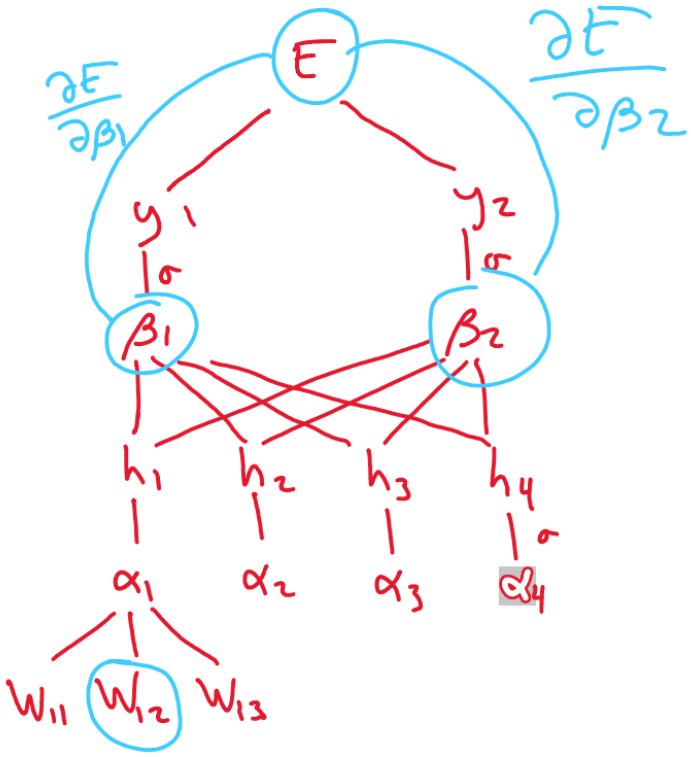
therefore

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

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

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

For expressing the gradient for all α_i's:

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

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:

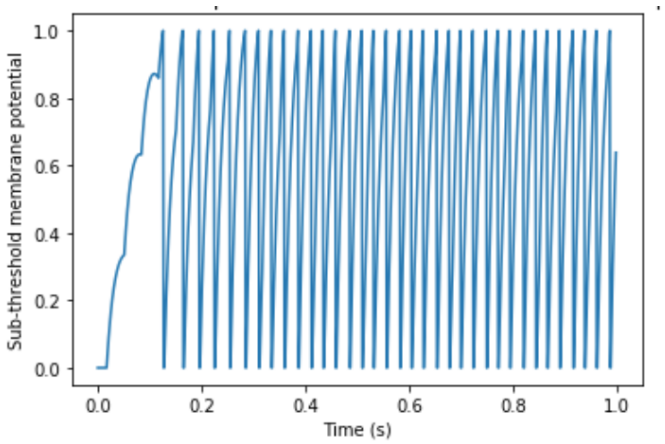


Figure 9.1: Suppose we needed to advance one layer futher down during backpropagation from layer $l+1$ to l.

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

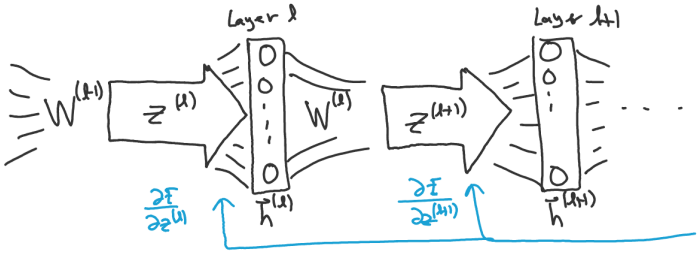
Definition 10.1 (Epoch)**.** Training usually involves going through the training data repeatedly and updating the network weights as we go. Each pass through the entire training data set is called an **epoch**.

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.

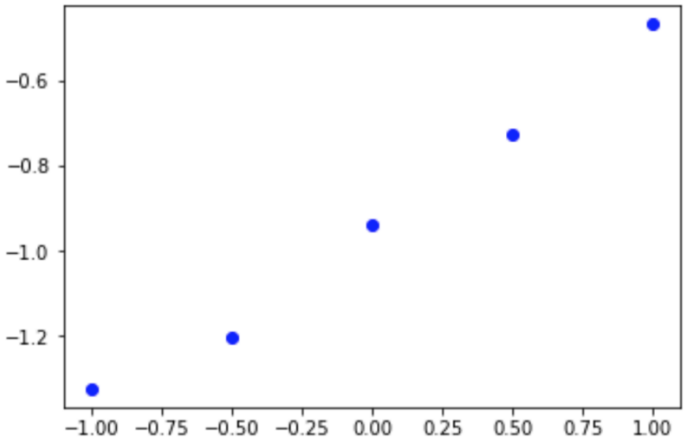
$$\tilde{E}(\vec{y}, \vec{t}; \theta) = E(\vec{y}, \vec{t}) + \lambda \|\theta\|_F^2$$

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).

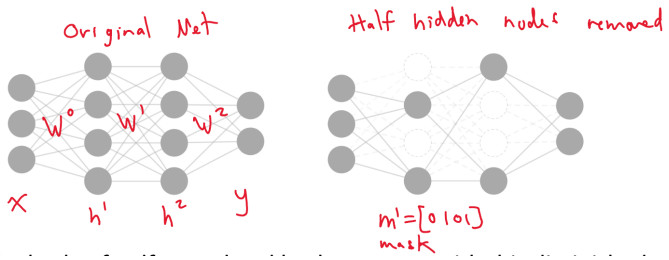
Data augmentation Another approach is to include a wider variety of samples in the training set so model is more robust to the noise of a few.

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).

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?

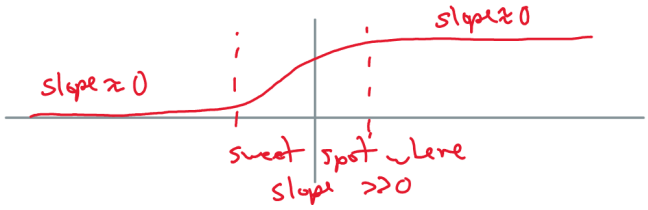
12 February 4, 2019

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:

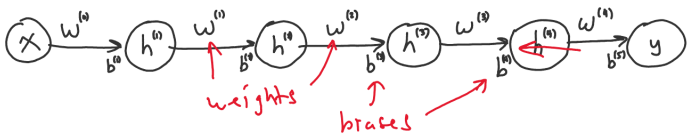


note that as $z \to -\infty$ or $z \to \infty$ then $\frac{dy}{dz} \to 0$, hence the updates to the weights will be smaller when th input currents are large in magnitude.
What about the next layer down? Suppose that $\frac{\partial E}{\partial \vec{z}^{(4)}} \approx 0.01$ (error gradient in the last 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

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

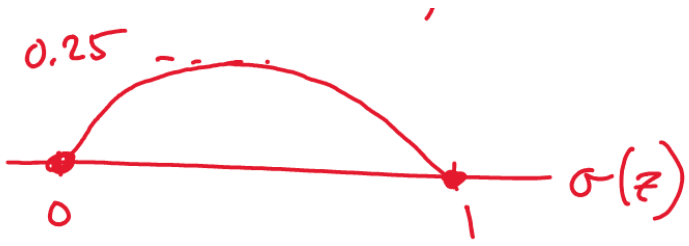
Using backprop we can compute a single formula for

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

Going deeper we have

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

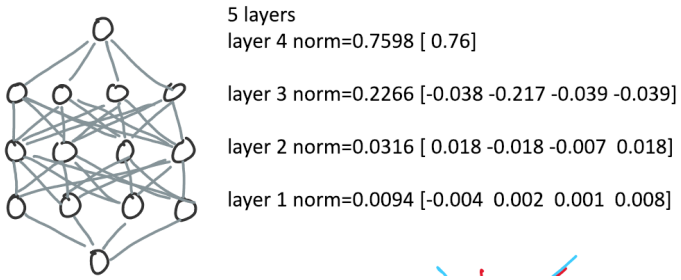
What is the steepest slope that $\sigma(z)$ attains? We note that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ for $0 \le \sigma(z) \le 1$ which looks like:



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

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

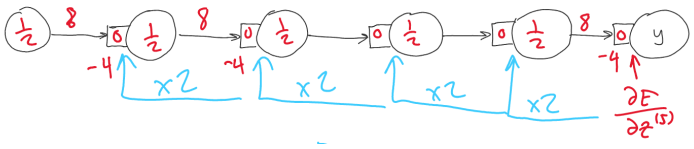
for example in a 5 layer network we may have



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

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

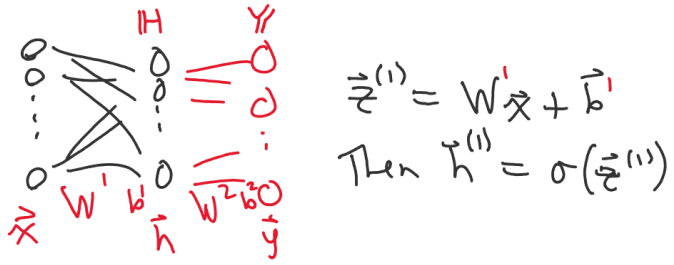
where $\vec{x} \in \mathbb{R}^{\mathbb{X}}, \vec{t} \in \mathbb{R}^{\mathbb{Y}}$, and $\mathbb{P}$ is the number of samples in our training set.
Notice we can put all our $\mathbb{P}$ inputs into a single matrix

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

similarly with our targets

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

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



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

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

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

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

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

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

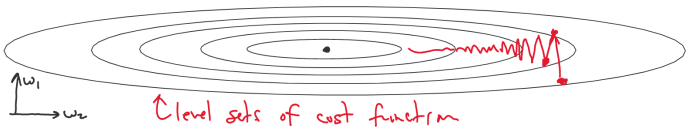
where on the RHS we have dimensions $\mathbb{Y} \times \mathbb{P}$ and $\mathbb{P} \times \mathbb{H}$, respectively, resulting in a LHS of dimension $\mathbb{Y} \times \mathbb{H}$. Note that we can use the same backprop formulas to process a whole batch of samples: this is called **batch gradient descent**.

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

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

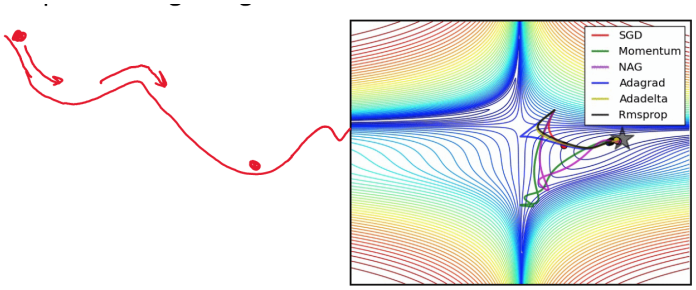
then our update to our weights would be

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

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

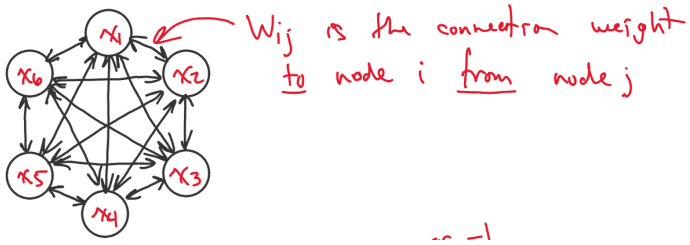
We look at **unsupervised learning techniques** next. Specifically we see how neural networks can be used to extract knowledge from *unlabelled data*.

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.

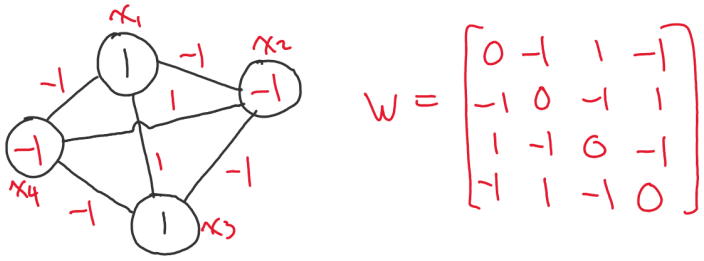
John Hopfield (1982) published a paper that proposes a method for using a neural network as a CAM. The network can learn patterns then converges to the closest pattern when shown a partial pattern.
The network is a complete bi-directional neural network:



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 $i = 1, \ldots, N$.
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:



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) $\{x^{(1)}, \ldots, x^{(n)}\}$ 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

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

thus we have

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

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:

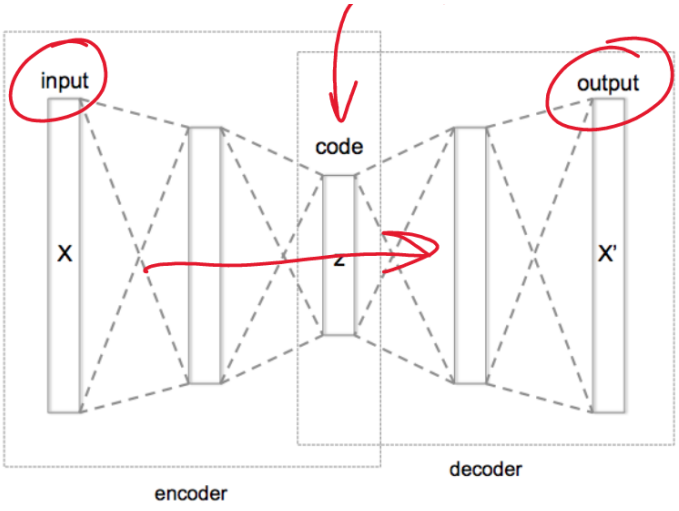


Figure 16.1: The "code" or encoding layer is smaller than the input and output layers.

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:

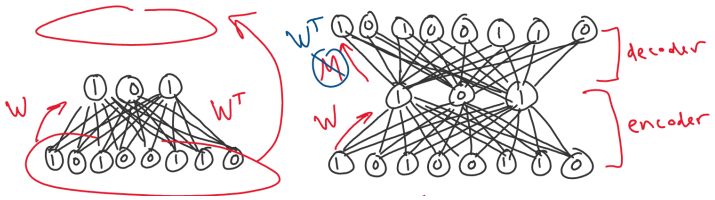


Figure 16.2: Left is a 2-layer representation of an autoencoder where we encode with weight matrix W and decode with W^T. Right is the autoencoder unrolled into three layers where we encode with W and decode with M.

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

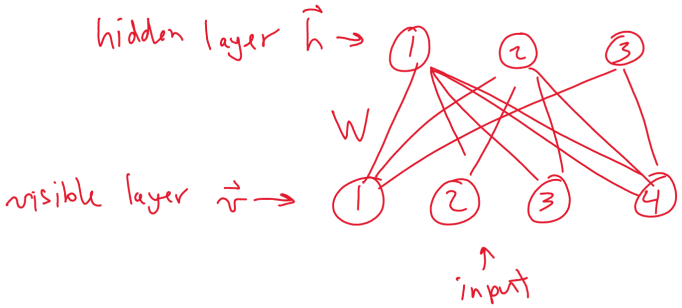
$$\frac{\partial \tilde{E}}{\partial W} = \frac{\partial \tilde{E}}{\partial M^T} = \frac{1}{2}\left(\frac{\partial E}{\partial W} + \frac{\partial E}{\partial W^T}\right)$$

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

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

Note that the RHS terms correspond to:

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

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

$\vec{h}\vec{c}^T$ 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).
To avoid/reduce getting stuck, we add varying degrees of randomness, that is:

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