Neural Networks

Jeff Orchard

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CS 479 during Winter 2021 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

Note that the course currently is CS 489, but in winter 2022, this course will become CS 479.

For simplicity, I will use the same chapter naming as the instructor did: one name for one week...

For any questions, send me an email via https://notes.sibeliusp.com/contact/.

You can find my notes for other courses on https://notes.sibeliusp.com/.

Sibelius Peng

Contents

Neuron Models

1.1 Neurons

A neuron is a special cell that can send and receive signals from other neurons.

A neuron can be quite long, sending its signal over a long distance; up to 50m long! But most are much shorter.
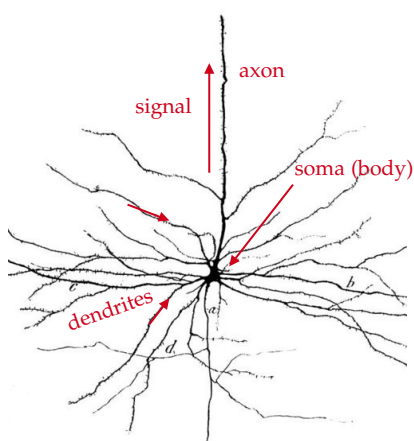
- Soma: 体细胞.

- Axon: 轴突. The electrical signal generated by soma travels along the axon.

- Dendrites: 树突. The electrical excitation is collected in the dendrites.

- Synapse: 突触. Structure that permits a neuron (or nerve cell) to pass an electrical or chemical signal to another neuron or to the target effector cell (wiki).

1.2 Neuron Membrane Potential

Ions(离子) are molecules or atoms in which the number of electrons (-) does not match the number of protons (+), resulting in a net charge. Many ions float around in your cells. The cell's membrane, a lipid bi-layer, stops most ions from crossing. However, ion channels embedded in the cell membrane can allow ions to pass.
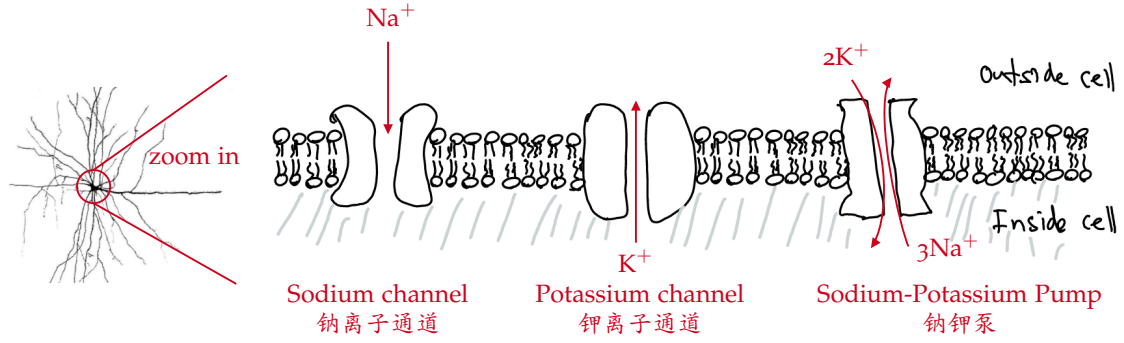
Sodium-Potassium Pump exchanges 3 Na$^+$ ions inside the cell for 2 K$^+$ ions outside the cell.

- Causes a higher concentration of Na$^+$ outside the cell, and higher concentration of K$^+$ inside the cell.

- It also creates a net positive charge outside, and thus a net negative charge inside the cell.

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

Neurons have a peculiar behaviour: they can produce a spike of electrical activity called an **action potential**(动作电位). This electrical burst travels along the neuron's *axon* to its *synapses*, where it passes signals to other neurons.

1.3 Hodgkin-Huxley Model

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

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

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

The fraction of K$^+$ channels that are open is $n(t)^4$, where $\frac{dn}{dt} = \frac{1}{\tau_n(V)}(n_\infty(V) - n)$. Here n is a dynamic variable, and $n_\infty(V)$ is the equilibrium solution constant.

The fraction of Na$^+$ ion channels is $(m(t))^3 h(t)$, where m and h are each themselves dynamic variables that also depend on the voltage.

$$\frac{dm}{dt} = \frac{1}{\tau_m(V)}(m_\infty(V) - m)$$
$$\frac{dh}{dt} = \frac{1}{\tau_h(V)}(h_\infty(V) - h)$$
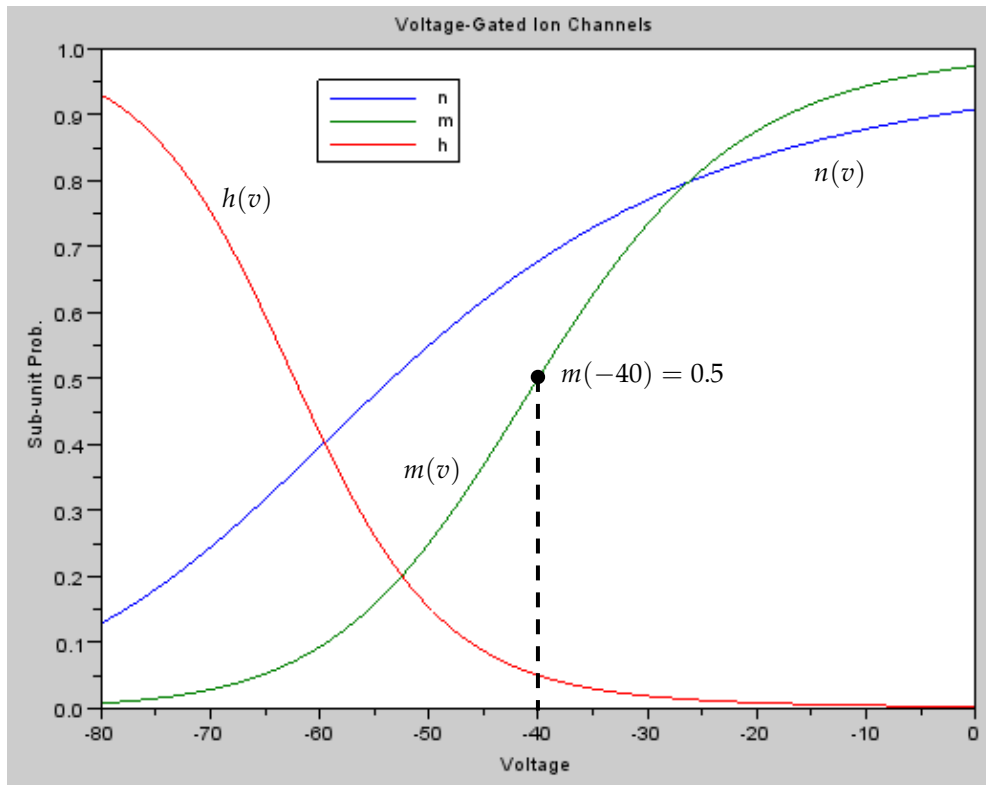
These two channels allow ions to flow into/out of the cell, inducing a current... which affects the membrane potential, V. Here is a differential equation which governs the membrane potential.

$$C\frac{dV}{dt} = J_{in} - \underbrace{g_L\left(V - V_L\right)}_{\text{leak current}} - \underbrace{g_{\text{Na}}m^3h\left(V - V_{\text{Na}}\right)}_{\text{sodium current}} - \underbrace{g_{\text{K}}n^4\left(V - V_{\text{K}}\right)}_{\text{potassium current}}$$

- C: capacitance.
- $\frac{dV}{dt}$: rate of change in voltage, or current.
- J_{in}: input current, usually from other neurons.
- $V_L, V_{\text{Na}}, V_{\text{K}}$: zero-current potentials.
- $g_L, g_{\text{Na}}, g_{\text{K}}$: max conductance.

This system of four differential equations (DEs) governs the dynamics of the membrane potential. Notice what happens when the input current is: negative, zero, slightly positive, very positive.

Here we can model this model in python. We have already seen these as functions of voltage.
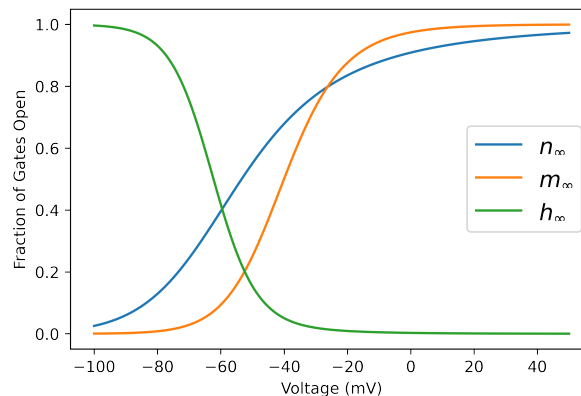
These are the τ's in case you are interested.
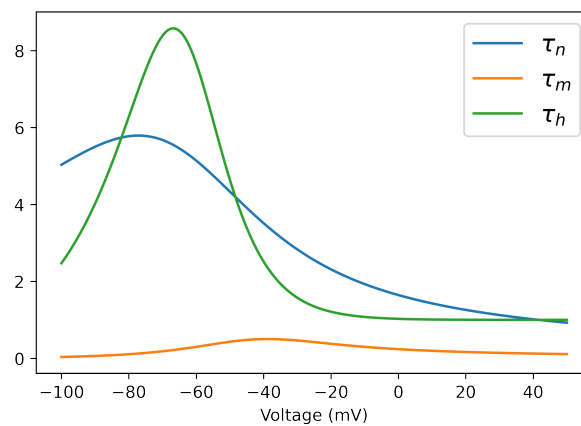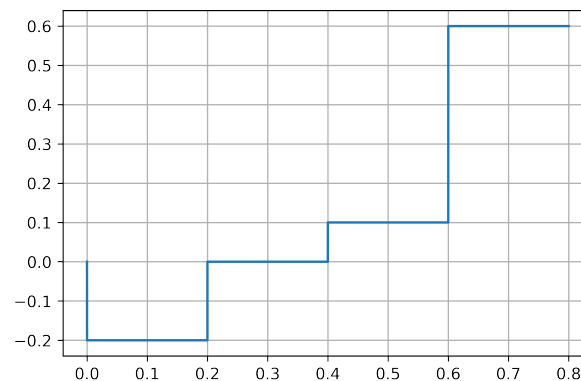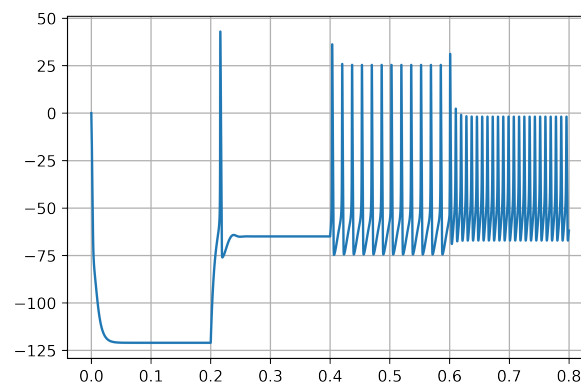

Here is the input current. At the beginning, we have negative current, then way to 0.6, which is fed in to the model.

Then this is how neuron behaves.

At the beginning, membrane potential goes to around -120. As we increase the input current, the membrane potential kinda goes higher. At 0.1, it's high enough that causes regular action potentials. As we increase input current even more, the action potentials continue to occur even faster. The firing rate of neurons goes up, the number of spikes per second goes up.

The HH model is already greatly simplified:

- a neuron is treated as a point in space

- conductances are approximated with formulas

- only considers K^+, Na^+ and generic leak currents

- etc.

But to model a single action potential (spike) takes many time steps of this 4-D system. However, spikes are fairly generic, and it is thought that the *presence* of a spike is more important than its specific shape. So instead of modelling spikes themselves, we are going to offload that to some generic spike phenomenon and look at the sub-threshold membrane potential model that.

1.4 Leaky Integrate-and-Fire Model

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

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

- C: capacitance.

- g_L: conductance and $g_L = \frac{1}{R}$ where R is resistance.

- J_{in}: input current.

If we multiply both sides by R, we get

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

- τ_m: time constant which dictates how quick things happen.

- RJ_{in}: by Ohm's Law, let $V_{in} = RJ_{in}$.

Thus, the voltage can be modelled as

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

So this is the dynamics of the sub-threshold membrane potential. Change of variables: $v = \frac{V - V_L}{V_{th} - V_L}$, then $v \to 0$ if $v_{in} = 0$ and $v = 1$ is the threshold. Then we end up with a DE:

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

We integrate the DE for a given input current (or voltage) until v reaches the threshold value of 1. Then we record a spike at time t_1. After it spikes, we wait a little bit, τ_{ref}, refractory time. It remains dormant during its refractory period, τ_{ref} (often just a few milliseconds). After that time, we integrate again from zero.
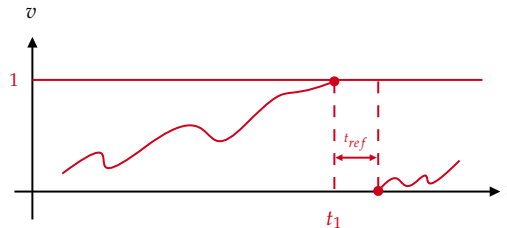
Let's put this in the context of the Hodgkin-Huxley model. If we zoom in on some little spikes here (between $0.72, 0.73$). We can see as follows:
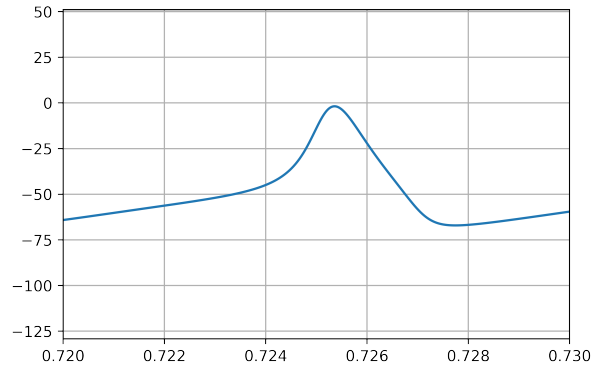
So the Hodgkin-Huxley model does model the spike itself.

LIF Firing Rate

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

Claim

$v(t) = v_{in}\left(1 - e^{-\frac{t}{\tau}}\right)$ is a solution of the IVP: $\tau \frac{dv}{dt} = v_{in} - v, \quad v(0) = 0.$

Proof:
Plug in the solution to the DE and show LHS = RHS. □

What does the solution look like? It will approach v_{in} asymptotically.

Importantly, for the neuron to fire an potential, v_{in} has to bigger than 1.

where t_{isi} stands for interspike interval.

It can be shown that the steady-state firing rate for a constant input v_{in} is

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

The graph plots the function above. It is called Tunning curve, because it tells us about how the neuron reacts to different input currents. In fact, eventually it would go asymptotic at a certain value.

Typical values for cortical neurons(神经元):

- $\tau_{ref} = 0.002$s

- $\tau_m = 0.02$s
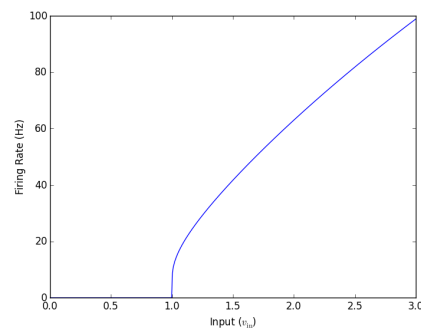
Let's take a look at even simpler neurons.

1.5 Activation functions

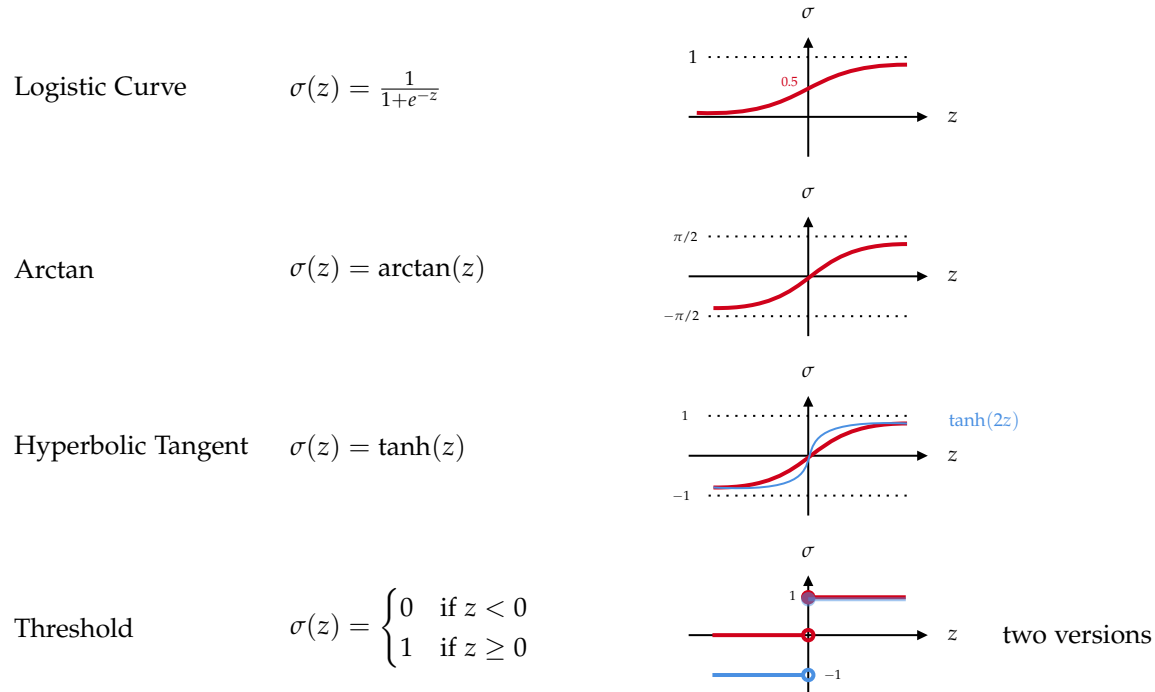As we've seen, the activity of a neuron is very low, or zero, when the input is low, and the activity goes up and approaches some maximum as the input increases. This general behaviour can be represented by a number of different activation functions. In general, we call these sigmoidal shape.

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

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

Hyperbolic Tangent $\quad \sigma(z) = \tanh(z)$

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

Rectified Linear Unit (ReLU): This is just a line that gets clipped below at zero. Leaky ReLU (LeReLU). Another version in green, which changes the slope when negative/at the origin.

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

LeReLU
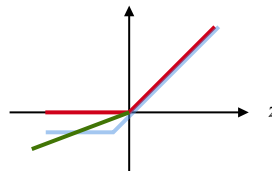
Multi-Neuron Activation Functions: Some activation functions depend on multiple neurons. Here are two examples.

SoftMax

SoftMax is ike a probability distribution (or probability vector), so its elements add to 1. If $\vec{z}$ is the drive (input) to a set of neurons, then

$$\text{SoftMax}(\vec{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$
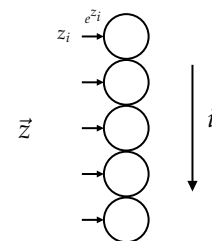
Then by definition, $\sum_i \text{SoftMax}(\vec{z})_i = 1$.

For example, $\vec{z} = [0.6, 3.4, -1.2, 0.05] \xrightarrow{\text{softmax}} \vec{y} = [0.06, 0.9, 0.009, 0.031]$

One-Hot

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

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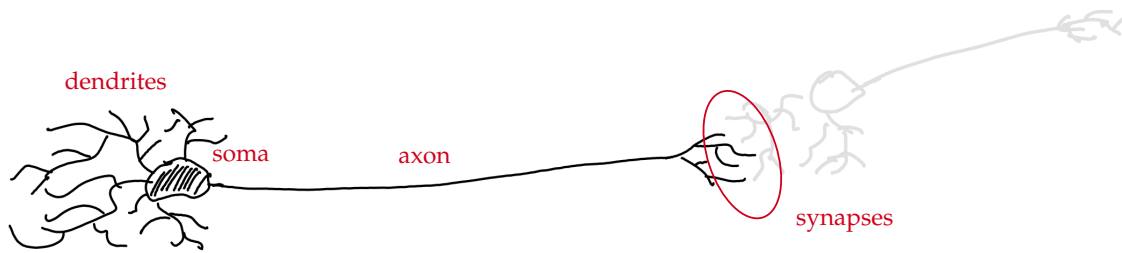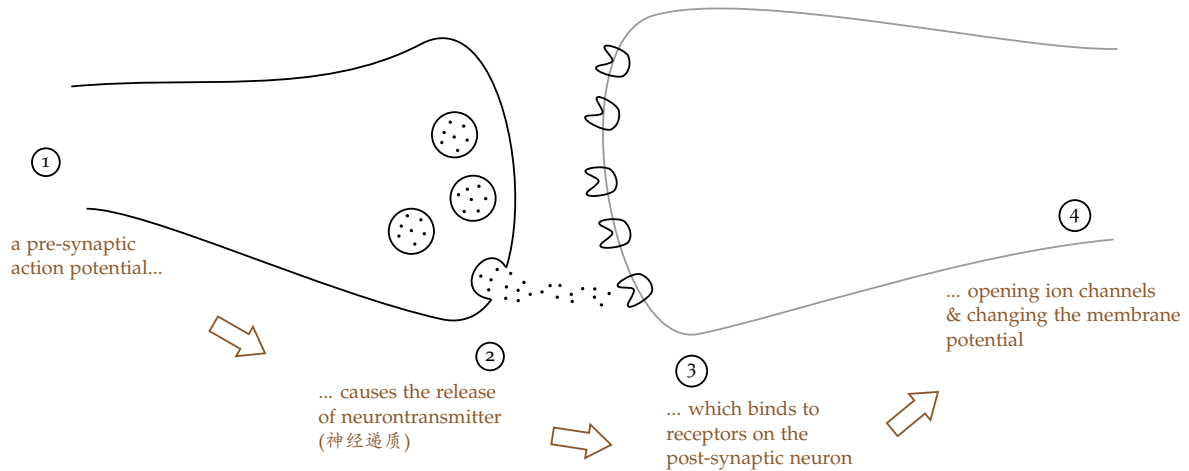
1.6 Synapses

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

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

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

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

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

where k is chosen so that $\int_0^\infty h(t)dt = 1 \implies k = \dfrac{1}{n!\tau_s^{n+1}}$.

The reason we have a split at zero is because the spike arrives at the synapse at time $t = 0$, and then we are looking what's happening after that.
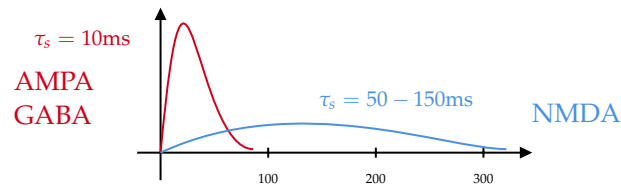
Some neurontransmitters are fast, like AMPA. Some are slow, like NMDA. The area under these curves are 1.

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

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

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

if we have three spikes at t_1, t_2, t_3.

Dirac Delta Function

Dirac Delta Function is defined as

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

and $\int_{-\infty}^{\infty} \delta(t)dt = 1$ and $\int_{-\infty}^{\infty} f(t)\delta(T - t)dt = f(T)$.

How does a spike train influence the post-synaptic neuron?

Answer: You simply add together all the PSC filters, one for each spike. This is actually convolving the spike train with the PSC filter.
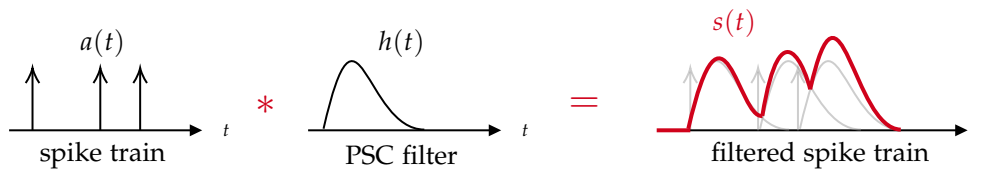
That is,

$$s(t) = (a * h)(t) = \sum_{p} h(t - t_p) = \text{sum of PSC filters, one for each spike}$$

1.7 Connection Weight

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

- the number and sizes of the synapses,

- the amount and type of neurotransmitter,

- the number and type of receptors,

- etc.

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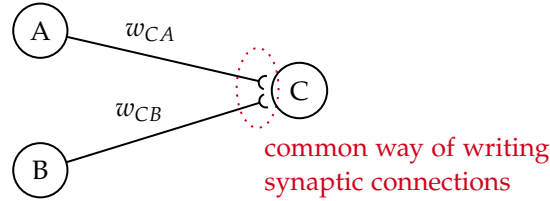

common way of writing
synaptic connections

Weight Matrices

When we have many pre-synaptic neurons, it is more convenient to use matrix-vector notation to represent the weights and activities.

Suppose we have 2 populations, X and Y, X has N nodes, Y has M nodes (neurons). If every node in X sends its output to every node in Y, then we will have a total of $N \times M$ connections, each with its own weight.
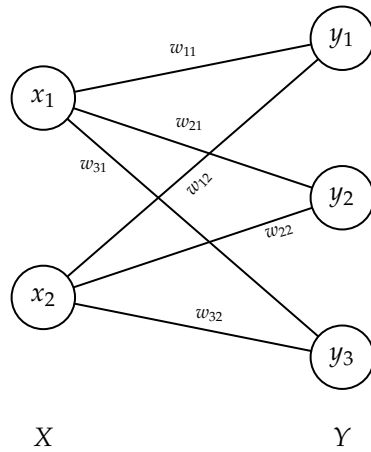
$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \in \mathbb{R}^{M \times N}$$

Storing the neuron activities in vectors,

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}.$$
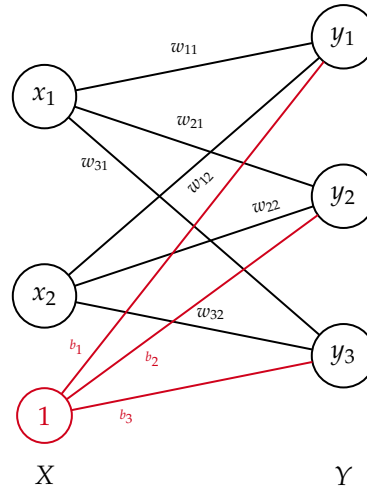
We can compute the input to the nodes in Y using

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

where $\vec{b}$ holds the biases for the nodes (neurons) in Y. Bias is sort of a catch-all for influences on the neuron that are not accounted for the connections that we are modelling.

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

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

So $W\vec{x} + \vec{b} = \begin{bmatrix} W|\vec{b} \end{bmatrix} \begin{bmatrix} \vec{x} \\ \overline{1} \end{bmatrix} = \hat{W} \begin{bmatrix} \vec{x} \\ \overline{1} \end{bmatrix}$.

Implementing Connections between Spiking Neurons

For simplicity, let $n = 0$: $h(t) = \frac{1}{\tau_s} e^{-\frac{t}{\tau_s}}$, which happens to be the solution of the IVP:

$$\tau_s \frac{ds}{dt} = -s, \quad s(0) = \frac{1}{\tau_s}.$$

Full LIF Neuron Model
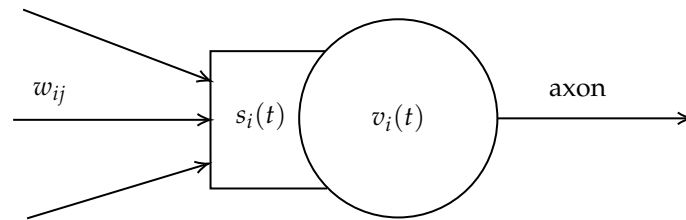
Differential equations:

$$\begin{cases} \tau_m \dfrac{dv_i}{dt} = s_i - v_i & \text{if not refracting} \\ \tau_s \dfrac{ds_i}{dt} = -s_i \end{cases}$$

If v_i reaches 1 (threshold)...

1. start refractory period,

2. send spike along axon,

3. reset membrane potential v to 0.

If a spike arrives from neuron j, increase s_i: $s_i \leftarrow s_i + \frac{w_{ij}}{\tau_s}$. The amount of current that it injects into the post-synaptic neuron is proportional to the weight, and we divide it by τ_s, which is the normalizing factor so that the total amount of current that eventually gets injected is the weight.

2

Formulation of Learning

2.1 Neural Learning

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

There are three basic categories of learning problems:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

In **supervised learning**, the desired output is known so we can compute the error and use that error to adjust our network.

> Example:
>
> Given an image of a digit, identify which digit it is.
>
>
>
> Input   Target
>
> $[0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0]$
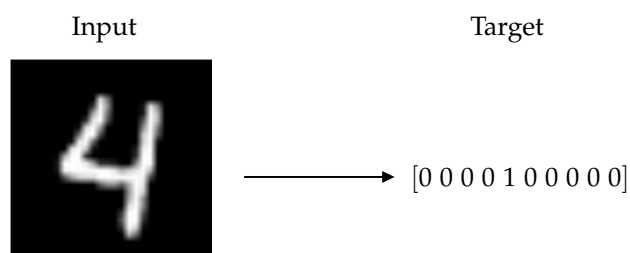
In **unsupervised learning**, the output is not known (or not supplied), so cannot be used to generate an error signal. Instead, this form of learning is all about finding efficient representations for the statistical structure in the input.

> Example:
>
> Given spoken English words, transform them into a more efficient representation such as phonemes, and then syllables.
>
> Or, cluster points into categories.

In **reinforcement learning**, feedback is given, but usually less often, and the error signal is usually less specific.

> Example:
>
> When playing a game of chess, a person knows their play was good if they win the game. They can try to learn from the moves they made.

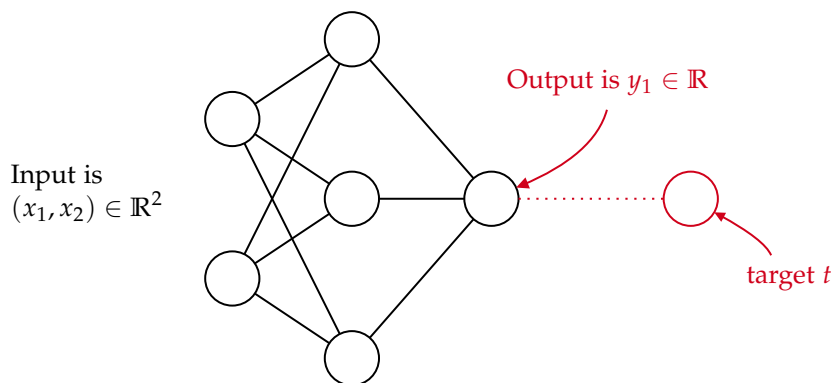In this course, we will mostly focus on supervised learning. But we will also look at some examples of unsupervised learning.

Supervised Learning

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

> Example:
>
>
>
> Input is $(x_1, x_2) \in \mathbb{R}^2$
>
> Output is $y_1 \in \mathbb{R}$
>
> target t

We are given training data, with many MANY examples of input/target pairs. This data is (presumably) the result of some consistent mapping process. For example, handwritten digits map to numbers. Or, XOR dataset. On the left, we have the inputs $(A, B) \in \{0, 1\}^2$, and the output $y \in [0, 1]$ and the target $t \in \{0, 1\}$.

A	B	$\mathrm{XOR}(A, B)$
1	1	0
1	0	1
0	1	1
0	0	0

Our task is to alter the connection weights in our network so that our network mimics this mapping. Our goal is to bring the output as close as possible to the target. But what, exactly, do we mean by "close"? For now, we will use the scalar function $L(y, t)$ as an error (or "loss") function, which returns a smaller value as our outputs are closer to the target.

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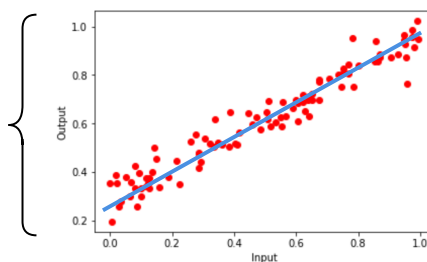
Regression

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

> Example: Linear regression
>
> $y, t \in \mathbb{R}$
>
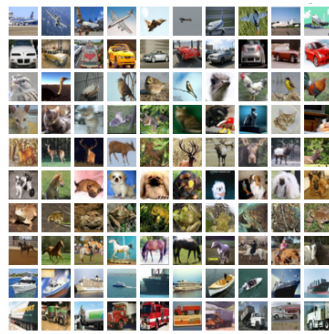> Outputs fall in a range of values
>
>

Classification

Outputs fall into a number of distinct categories.

> **Example: MNIST**
>
> MNIST stands for Modified National Institute of Standards and Technology database.

Inputs	Targets	Inputs	Targets
7	[0 0 0 0 0 0 0 1 0 0]	5	[0 0 0 0 0 1 0 0 0 0]
0	[1 0 0 0 0 0 0 0 0 0]	4	[0 0 0 0 1 0 0 0 0 0]
6	[0 0 0 0 0 0 1 0 0 0]	9	[0 0 0 0 0 0 0 0 0 1]

> **Example: CIFAR-10**

Inputs	Targets
	airplane
	automobile
	bird
	cat
	deer
	dog
	frog
	horse
	ship
	truck

Optimization

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

Let our network be represented by the mapping f so that $y = f(x; \theta)$ where θ represents all the weights and biases. Neural learning seeks

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

In other words, find the weights and biases that minimize the expected cost (or error, or loss) between the outputs and the targets, over the dataset.
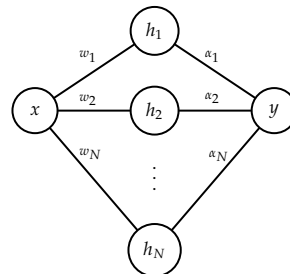
2.2 Universal Approximation Theorem

Can we approximate any function using a neural network?

Given a function $f(x)$, can we find weights $\omega_j, \alpha_j,$ and biases $\theta_j, j = 1, \dots, N$ such that

$$f(x) = \sum_{j=1}^{N} \alpha_j \underbrace{\sigma(w_j x + \theta_j)}_{h_j}$$

to arbitrary precision?

> **Theorem 2.1: Universal Approximation Theorem**
>
> Let σ be any continuous sigmoidal function. Then finite sums of the form
>
> $$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j x + \theta_j)$$
>
> are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(x)$, of the above form, for which
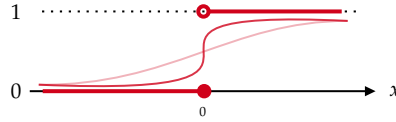> $$|G(x) - f(x)| < \epsilon \qquad \forall x \in I_n.$$

Here $C(I_n)$ denotes the continuous functions on I_n, and I_n can be $I_n = [0,1]^n$.

A function σ is "sigmoidal" if $\sigma(x) = \begin{cases} 1 & \text{as } x \to \infty \\ 0 & \text{as } x \to -\infty \end{cases}$
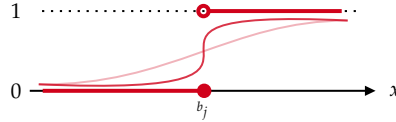
The theorem states that $\exists N$, and $\exists w_j, \theta_j, \alpha_j$ for $j = 1, \ldots, N$ such that $|G(x) - f(x)| < \epsilon$.

Proof:

Suppose we let $w_j \to \infty$ for $j = 1, \ldots, N$, then $\sigma(w_j x) \xrightarrow{w_j \to \infty} \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$. We can visualize it by, for example, logistic function, and crank up that w:

Or let's look at a shifted version of it: $\sigma(w_j(x - b_j)) \xrightarrow{w_j \to \infty} \begin{cases} 0 & \text{for } x \leq b_j \\ 1 & \text{for } x > b_j \end{cases}$.

This is the same as the Heaviside step function, $H(x) = \lim_{w \to \infty} \sigma(wx)$.

Define $H(x; b) := \lim_{w \to \infty} \sigma(w(x - b))$ which has two inputs: x and the shift b.

We can use two such functions to create a piece, $P(x; b, \delta) := H(x; b) - H(x; b + \delta)$
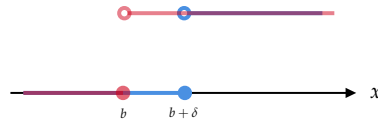
Then,

$$P(x; b, \delta) =$$

Since $f(x)$ is continuous, $\lim_{x \to a} f(x) = f(a)$, $\forall a \in I_n$. Then there exists an interval, $(a_j, a_j + \Delta x)$ such that $|f(x) - f(a_j)| < \epsilon$ $\forall x \in (a_j, a_j + \Delta x)$.

Choose $b_j = a_j$, $\delta_j = \Delta x$, and $\alpha_j = f(a_j)$. Therefore,

$$|f(x) - f(a_j)| < \epsilon \quad \text{for } a_j \leq x \leq a_j + \delta_j$$

$$|f(x) - \alpha_j P(x; b_j, \delta_j)| < \epsilon \quad \text{for } a_j \leq x \leq a_j + \delta_j$$

Max error in $[a_j, a_j + \delta_j]$ is less than ϵ.
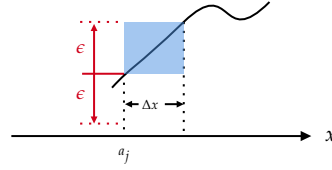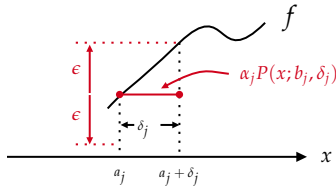
Repeat this process for $x = a_{j+1} = b_j + \delta_j$. Construct

$$G(x) = \sum_{j=1}^{N} \alpha_j P(x; b_j, \delta_j)$$

as desired.   □

This theorem shows that with a single hidden layer you can get arbitrarily close to modeling any functions you want. So, why would we ever need a neural network with more than one hidden layer? The theorem guarantees existence, but makes no claims about N, the number of hidden neurons N might grow exponentially as ϵ gets smaller.

2.3   Loss Functions

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

Suppose we are given a dataset $\{x_i, t_i\}_{i=1}^{N}$. For input x_i, the network's output is $y_i = f(x_i; \theta)$.

2.3.1   (Mean) Squared Error

$$L(y, t) = \frac{1}{2} \|y - t\|_2^2$$

Taking the expectation (mean) over the entire dataset,

$$E = \frac{1}{N} \sum_{i=1}^{N} L(y_i, t_i).$$

The use of MSE as a cost function is often associated with linear activation functions, or ReLU. This loss-function/activation-function pair is often used for regression problems.

2.3.2   Cross Entropy (Bernoulli Cross Entropy)

Consider the task of classifying inputs into two categories, labelled 0 and 1. Our neural-network model for this task will output a single value between 0 and 1.

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

where the true class is expressed in the target, t, is either 0 or 1.

If we suppose that y is the probability that $x \to 1$ (is of class 1), $y = P(x \to 1|\theta) = f(x;\theta)$, then we can treat it as a Bernoulli distribution:

$$\begin{aligned} P(x \to 1|\theta) &= y & \text{i.e., } t = 1 \\ P(x \to 0|\theta) &= 1 - y & \text{i.e., } t = 0 \end{aligned}$$

The likelihood of our data sample given our model is

$$P(x \to t|\theta) = y^t (1-y)^{1-t},$$

which works for both classes.

The task of "learning" would be finding a model (θ) that maximizes this likelihood. Or, we could equivalently minimize the negative log-likelihood

$$L(y,t) = -\big(t \log y + (1-t) \log(1-y)\big),$$

and this log-likelihood formula is the basis of the cross-entropy loss function.

The expected cross entropy over the entire dataset is

$$\begin{aligned} E &= -\mathbb{E}\big[t_i \log y_i + (1-t_i) \log(1-y_i)\big]_{\text{over the dataset}} \\ &= -\frac{1}{N} \sum_{i=1}^{N} t_i \ln y_i + (1-t_i) \ln(1-y_i) \end{aligned}$$

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

2.3.3   Categorical Cross-Entropy (Multinoulli Cross-Entropy)

Consider a classification problem that has K classes ($K > 2$). Given an input, the task of our model is to output the class of the input. For example, given an image of a digit, determine the digit class.

Suppose our model is given the input x, then the the network's output is $y = f(x;\theta) \in [0,1]^k$. For example, $y = [0.2, 0.1, 0.4, 0.3]$. We interpret y_k as the probability of x being from class k. That is, y is the distribution of x's membership over the K classes. Note that $\sum_{k=1}^{K} y_k = 1$.

Under that distribution, suppose we observed a sample from class $\bar{k}$, the likelihood of that observation is $P(x \in C_{\bar{k}}|\theta) = y_{\bar{k}}$ where $C_{\bar{k}} = \{x | x \text{ is from class } \bar{k}\}$.

Note that y is a function of the input x, and the model parameters θ (the prof put this statement in there for a reason that is not relevant now).

If we represent the target class using the one-hot vector

$$t = [0,0,\ldots, \underset{\underset{\bar{k}}{\uparrow}}{1}, 0, \ldots, 0],$$

then we can write the likelihood as

$$P(x \in C_{\bar{k}}|\theta) = \prod_{k=1}^{K} y_k^{t_k}.$$

Thus, the negative log-likelihood of x is

$$-\log P(x \in C_{\bar{k}}|\theta) = -\sum_{k=1}^{K} t_k \log y_k.$$

This loss function is known as **categorical cross-entropy**:

$$L(y, t) = - \sum_{k=1}^{K} t_k \log y_k.$$

The expected categorical cross-entropy for a dataset of N samples is

$$E = -\mathbb{E}\left[L(y_i, t_i)\right]_{\text{dataset}} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} t_k^{(i)} \log y_k^{(i)}$$

where the superscript (i) is for the sample i.

Since $\sum_k y_k = 1$, this cost function works well with SoftMax which outputs discrete distributions.

3

Error Backpropagation

3.1 Gradient Descent Learning

The operation of our network can be written $y = f(x; \theta)$ where θ are connection weights and biases. So, if our loss function is $L(y, t)$, where t is the target, then neural learning becomes the optimization problem $\min_\theta E(\theta)$ where $E(\theta) = \mathbb{E}\left[L\big(f(x; \theta), t(x)\big) \right]_{x \in data}$. We can apply gradient descent to E, using the gradient $\nabla_\theta E = \begin{bmatrix} \frac{\partial E}{\partial \theta_0} & \frac{\partial E}{\partial \theta_1} & \cdots & \frac{\partial E}{\partial \theta_p} \end{bmatrix}^T$.

3.1.1 Gradient-Based Optimization

If you want to find a local maximum of a function, you can simply start somewhere, and keep walking uphill. For example, suppose you have a function with two inputs, $E(a, b)$. You wish to find a and b to maximize E. We are trying to find the parameters $(\bar{a}, \bar{b})$ that yield the maximum value of E, i.e., $(\bar{a}, \bar{b}) = \text{argmax}_{(a,b)} E(a, b)$. No matter where you are, "uphill" is in the direction of the gradient vector,

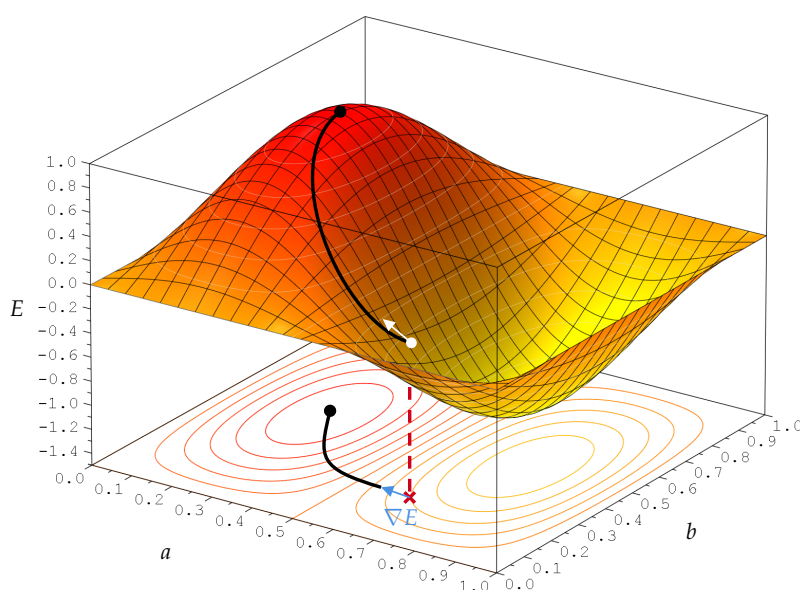$$\nabla E(a, b) = \begin{bmatrix} \frac{\partial E}{\partial a} & \frac{\partial E}{\partial b} \end{bmatrix}^T.$$

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

Gradient ascent is an optimization method where you step in the direction of your gradient vector. If your current position is (a_n, b_n), then $(a_{n+1}, b_{n+1}) = (a_n, b_n) + k\nabla E(a_n, b_n)$ where k is your step multiplier.

Gradient *descent* aims to *minimize* your objective function. So, you walk downhill, stepping in the direction opposite the gradient vector. Note that there is no guarantee that you will actually find the global optimum. In general, you will find a local optimum that may or may not be the global optimum.
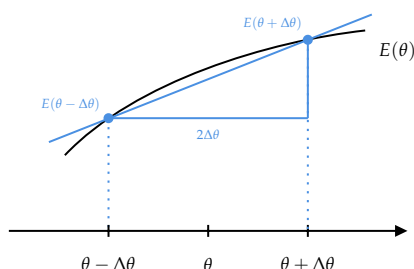
3.1.2 Approximating the Gradient Numerically

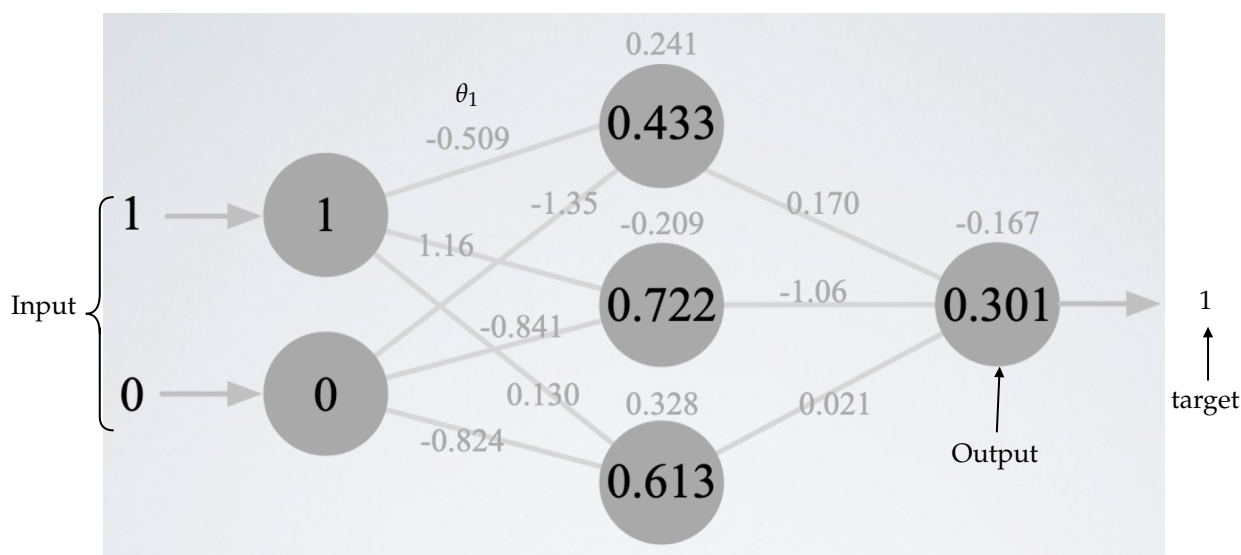We can estimate the partial derivatives in the gradient using finite-differencing.

> **Finite-Difference Approximation**
>
> For a function $E(\theta)$, we can approximate $\frac{dE}{d\theta}$ using
>
> $$\frac{dE}{d\theta} \approx \frac{E(\theta + \Delta\theta) - E(\theta - \Delta\theta)}{2\Delta\theta}$$

As an example, consider this network (assume logistic activation function):

It's a neural network, with connection weights and biases shown. Recall we seek $\min_\theta E(\theta)$. We will use cross entropy.

Consider θ_1 on its own. With $\theta_1 = -0.509$, our network output is $y = 0.301$. This gives $E(-0.509) = 1.201$. What if we perturb θ_1, so that $\theta_1 = -0.509 + 0.1 = -0.409$. The our output is $y = 0.302$. This yields $E(-0.409) = 1.198$.
$$\uparrow$$
$$\Delta\theta$$

If, instead, we perturb θ_1 so that $\theta_1 = -0.509 - 0.1 = -0.609$, then our output is $y = 0.302$, which gives $E(-0.609) = 1.204$.
$$\uparrow$$
$$-\Delta\theta$$

Then we can estimate $\frac{\partial E}{\partial \theta_1}$ using

$$\frac{\partial E}{\partial \theta_1} \approx \frac{E(-0.409) - E(-0.609)}{2 \times 0.1} = -0.0292$$

Obviously, increasing θ_1 seems to be the right thing to do. Then $\theta_1 \leftarrow \theta_1 - \underset{\substack{\uparrow \\ \text{positive constant}}}{k} \cdot (-0.0292)$.

3.2 Error Backpropagation

The goal here is to find an efficient method to compute the gradients for gradient-descent optimization. We can apply gradient descent on a multi-layer network, using chain rule to calculate the gradients of the error with respect to deeper connection weights and biases.

Consider the network:
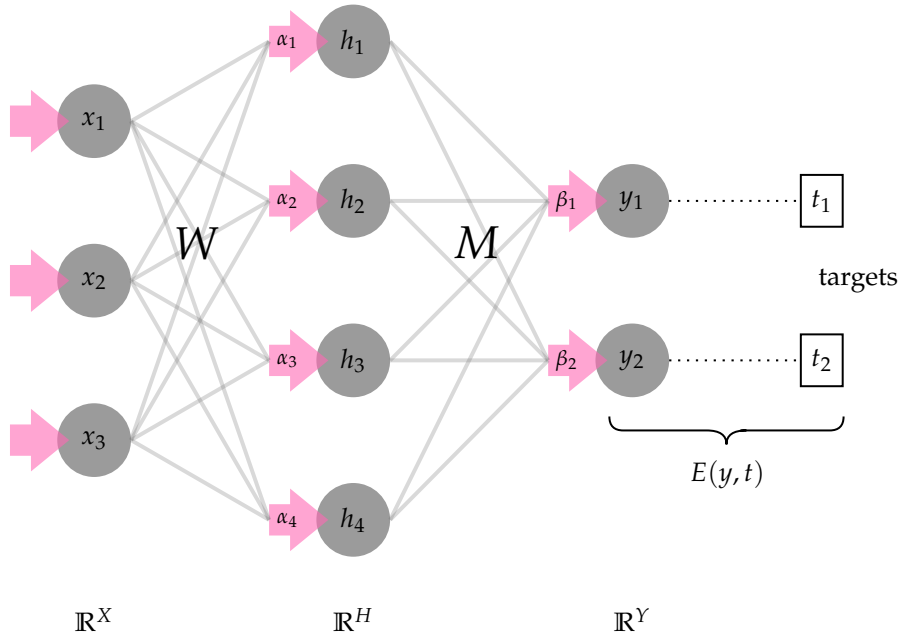
α_i is the input current to hidden node i. β_j is the input current to the output node j. For our cost (loss) function, we will use $E(y, t)$. For learning, suppose we want to know $\frac{\partial E}{\partial M_{41}}$, where M_{41} is going from h_4 to β_1. We can represent this by a computation/dependency graph.
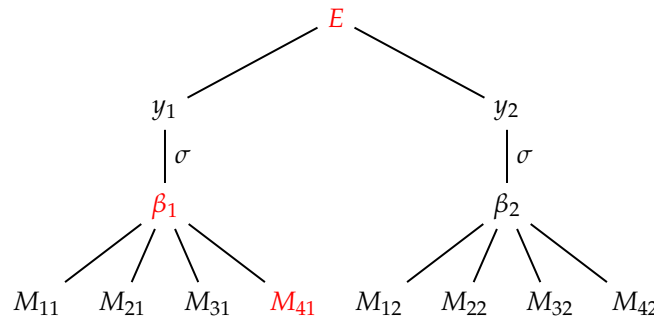
Recall, $E(y, t) = E(\sigma(\underbrace{hM + b}_{\beta_1}), t)$. Therefore, $\frac{\partial E}{\partial \beta_1} = \frac{\partial E}{\partial y_1} \frac{\partial y_1}{\partial \beta_1}$. Thus, $\frac{\partial E}{\partial M_{41}} = \frac{\partial E}{\partial \beta_1} \frac{\partial \beta_1}{\partial M_{41}}$.

Recall, $\beta_1 = \sum\limits_{i=1}^{4} h_i M_{i1} + b_1$, then $\frac{\partial \beta_1}{\partial M_{41}} = h_4$. Therefore, $\frac{\partial E}{\partial M_{41}} = \frac{\partial E}{\partial \beta_1} h_4$.

OK, that works for the connection weights between the top two layers. What about the connection weights between layers deeper in the network? Say if we want to find $\frac{\partial E}{\partial W_{21}}$. First, we draw a dependency graph.
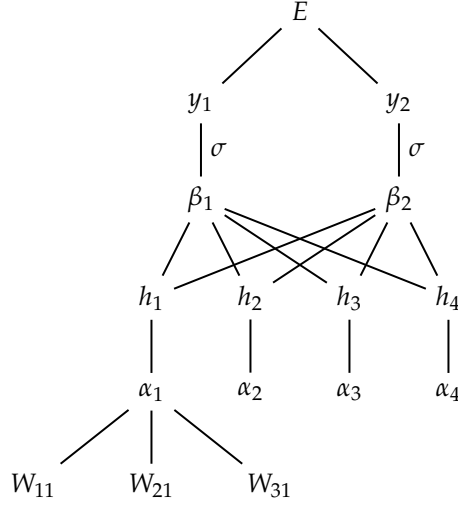
First note that $\alpha_1 = \sum\limits_{j=1}^{3} x_j W_{j1} + a_1$. Therefore, $\frac{\partial \alpha_1}{\partial W_{21}} = x_2$. And

$$
\begin{aligned}
\frac{\partial E}{\partial \alpha_1} &= \frac{\partial E}{\partial h_1} \frac{\mathrm{d} h_1}{\mathrm{d} \alpha_1} \\
&= \left(\frac{\partial E}{\partial \beta_1} \frac{\partial \beta_1}{\partial h_1} + \frac{\partial E}{\partial \beta_2} \frac{\partial \beta_2}{\partial h_1} \right) \frac{\mathrm{d} h_1}{\mathrm{d} \alpha_1} \\
&= \left(\frac{\partial E}{\partial \beta_1} M_{11} + \frac{\partial E}{\partial \beta_2} M_{12} \right) \frac{\mathrm{d} h_1}{\mathrm{d} \alpha_1} \\
&= \left(\frac{\partial E}{\partial \beta_1}, \frac{\partial E}{\partial \beta_2} \right) \cdot (M_{11}, M_{12}) \frac{\mathrm{d} h_1}{\mathrm{d} \alpha_1}.
\end{aligned}
\tag{$*$}
$$

($*$): assume $\frac{\partial E}{\partial \beta_1}, \frac{\partial E}{\partial \beta_2}$ are known because these gradients were used to compute the loss with respect to the weights in the top layer already and we are doing backpropagation.

Then $\frac{\partial E}{\partial W_{21}} = \frac{\partial E}{\partial \alpha_1} \frac{\partial \alpha_1}{\partial W_{21}} = \cdots$ using the results above.

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

$$
\frac{\partial E}{\partial \alpha_i} = \frac{\mathrm{d} h_i}{\mathrm{d} \alpha_i} \begin{bmatrix} \frac{\partial E}{\partial \beta_1} & \cdots & \frac{\partial E}{\partial \beta_Y} \end{bmatrix} \cdot \begin{bmatrix} M_{i1} & \cdots & M_{iY} \end{bmatrix} = \frac{\mathrm{d} h_i}{\mathrm{d} \alpha_i} \begin{bmatrix} \frac{\partial E}{\partial \beta_1} & \cdots & \frac{\partial E}{\partial \beta_Y} \end{bmatrix} \cdot \begin{bmatrix} M_{i1} \\ \vdots \\ M_{iY} \end{bmatrix}^T
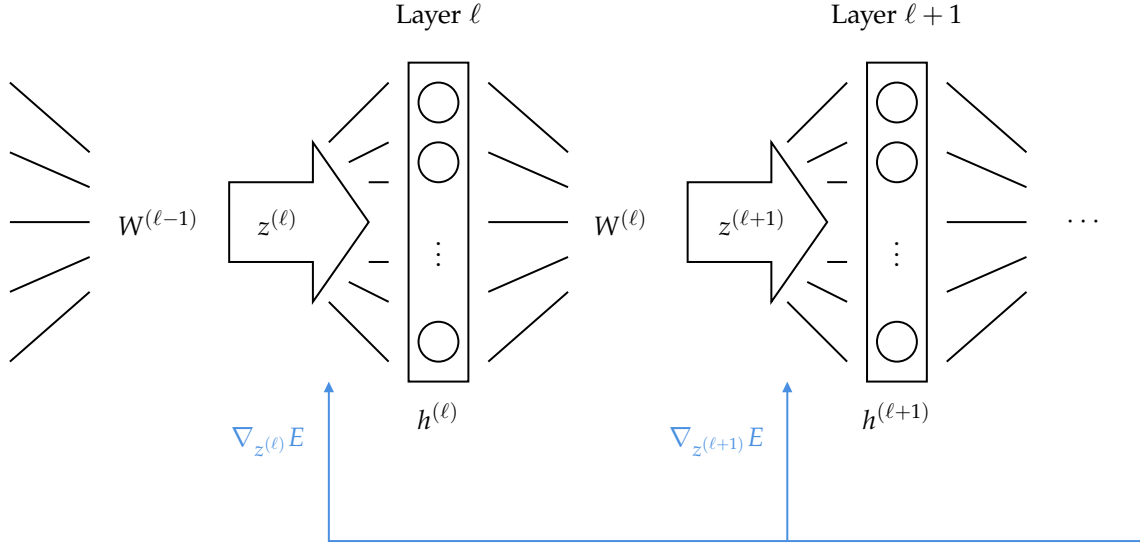$$

For all elements,

$$
\begin{bmatrix} \frac{\partial E}{\partial \alpha_1} & \cdots & \frac{\partial E}{\partial \alpha_H} \end{bmatrix} = \begin{bmatrix} \frac{\mathrm{d} h_1}{\mathrm{d} \alpha_1} & \cdots & \frac{\mathrm{d} h_H}{\mathrm{d} \alpha_H} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial E}{\partial \beta_1} & \cdots & \frac{\partial E}{\partial \beta_Y} \end{bmatrix} \begin{bmatrix} M_{11} & \cdots & M_{H1} \\ \vdots & & \vdots \\ M_{1Y} & \cdots & M_{HY} \end{bmatrix}
$$

where $\odot$ is the Hadamard product: $(A \odot B)_{ij} = (A)_{ij} (B)_{ij}$. Then

$$
\nabla_\alpha E = \frac{\mathrm{d} h}{\mathrm{d} \alpha} \odot \left(\nabla_\beta E \cdot M^T \right).
$$

The most general, in going down a layer, from layer $\ell + 1$ down to ℓ.

Note that in the network below, superscripts denote the layer.

Layer ℓ Layer $\ell + 1$

Suppose we have $\nabla_{z^{(\ell+1)}} E = \frac{\partial E}{\partial z^{(\ell+1)}}$. Let $h^{(l+1)} = \sigma(z^{(\ell+1)}) = \sigma(h^{(\ell)} W^{(\ell)} + b^{(\ell+1)})$. Then in our context,

$$\nabla_{z^{(\ell)}} = \frac{\mathrm{d}h^{(\ell)}}{\mathrm{d}z^{(\ell)}} \odot \left[\nabla_{z^{(\ell+1)}} E \cdot \left(W^{(\ell)} \right)^T \right]$$

Then, to compute $\frac{\partial E}{\partial W_{ij}^{(\ell)}}$,

$$\frac{\partial E}{\partial W_{ij}^{(\ell)}} = \frac{\partial E}{\partial z_j^{(\ell+1)}} \frac{\partial z_j^{(\ell+1)}}{\partial W_{ij}^{(\ell)}} = \frac{\partial E}{\partial z_j^{(\ell+1)}} h_i^{(\ell)} = h_i^{(\ell)} \frac{\partial E}{\partial z_j^{(\ell+1)}}$$

Note that, one term depends on i, the other depends on j, and there's no entity having both i and j. Then this can be written simply for all elements by picking h_i and z_j that we want.

$$\frac{\partial E}{\partial W^{(\ell)}} = \begin{bmatrix} \uparrow \\ h^{(\ell)} \\ \downarrow \end{bmatrix} \begin{bmatrix} \leftarrow & \nabla_{z^{(\ell+1)}E} & \rightarrow \end{bmatrix}$$

Note that this is an outer product between two vectors. The result is a matrix, same size as $W^{(\ell)}$.

Summary

Suppose we have $\nabla_{z^{(\ell+1)}} E$, we want to calculate $\nabla_{z^{(\ell+1)}} E$ and $\nabla_{W^{(\ell)}} E$. Here σ is the activation function between $z^{(\ell)}$ and $h^{(\ell)}$.

$$\nabla_{z^{(\ell)}} E = \sigma'(z^{(\ell)}) \odot \left[\nabla_{z^{(\ell+1)}} E \cdot \left(W^{(\ell)} \right)^T \right]$$
$$\nabla_{W^{(\ell)}} E = [h^{(\ell)}]^T \nabla_{z^{(\ell+1)}} E$$

Note that by default $h^{(\ell)}$ is a row vector.