Neural Networks

CS 479

Jeff Orchard

LaTeXed by Sibelius Peng

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

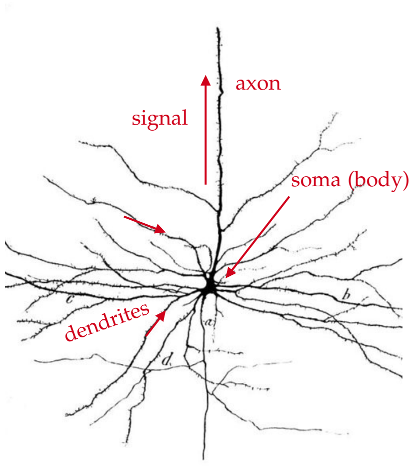
1

Neuron Models

1.1 Neurons

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

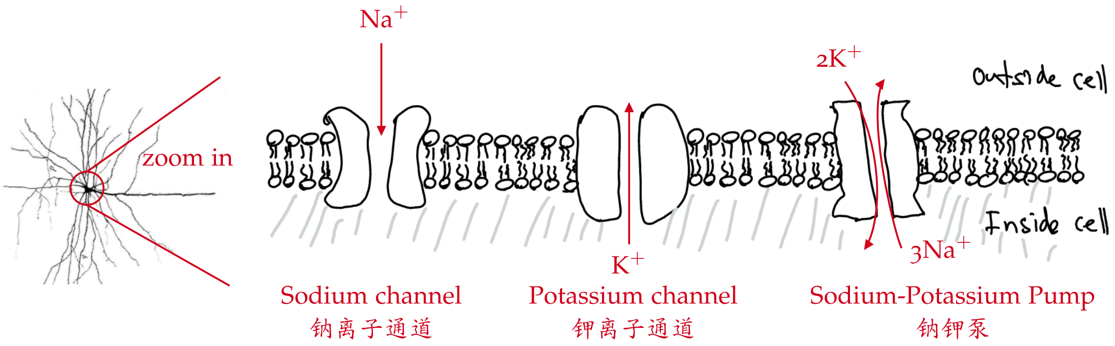
A neuron can be quite long, sending its signal over a long distance; up to 50m long! But most are much shorter.



- 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.

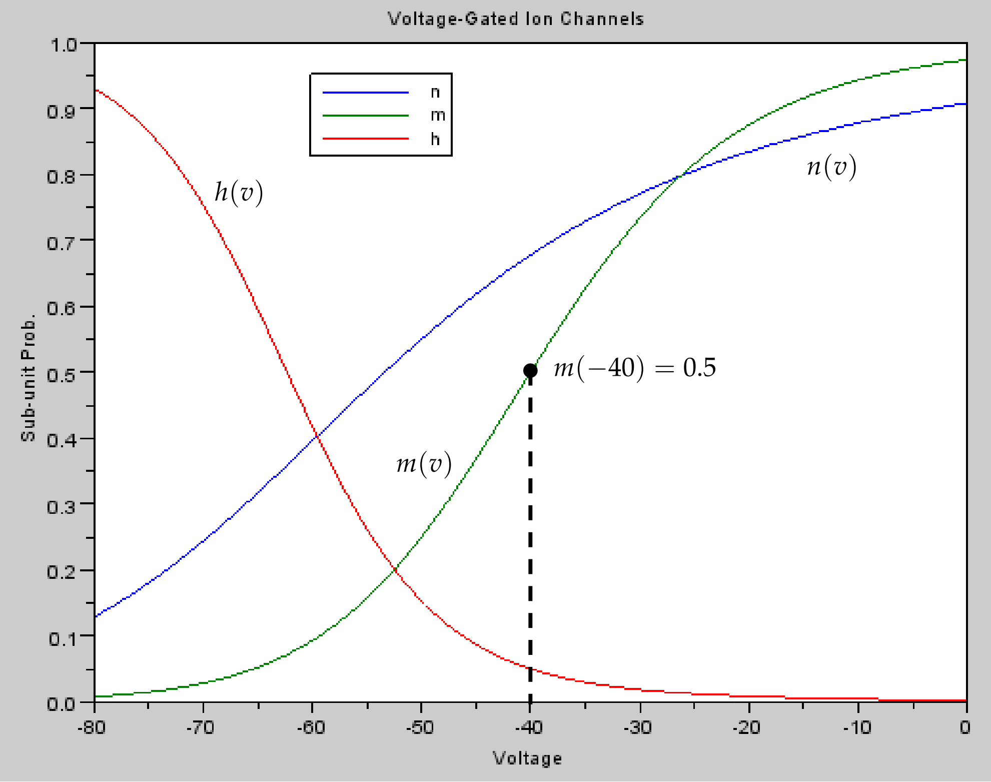
Both Na$^+$ and K$^+$ ion channels are voltage-dependent, so their opening and closing changes with the membrane potential.

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.

$$\begin{aligned}\frac{dm}{dt} &= \frac{1}{\tau_m(V)}(m_\infty(V) - m) \\ \frac{dh}{dt} &= \frac{1}{\tau_h(V)}(h_\infty(V) - h)\end{aligned}$$



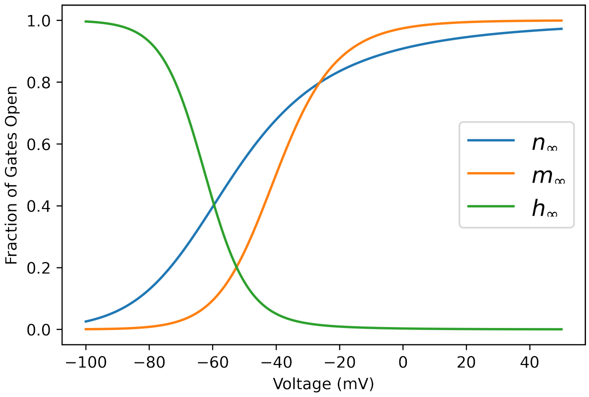
These two channels allow ions to flow into/out of the cell, inducing a current. . . which affects the membrane potential, V. Here is a differential equation which governs the membrane potential.

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

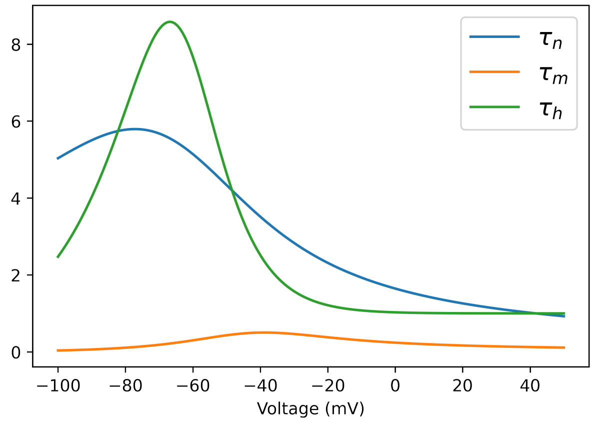
- C: capacitance.
- $\frac{dV}{dt}$: rate of change in voltage, or current.
- J_{in}: input current, usually from other neurons.
- $V_L, V_{\text{Na}}, V_{\text{K}}$: zero-current potentials.
- $g_L, g_{\text{Na}}, g_{\text{K}}$: max conductance.

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.

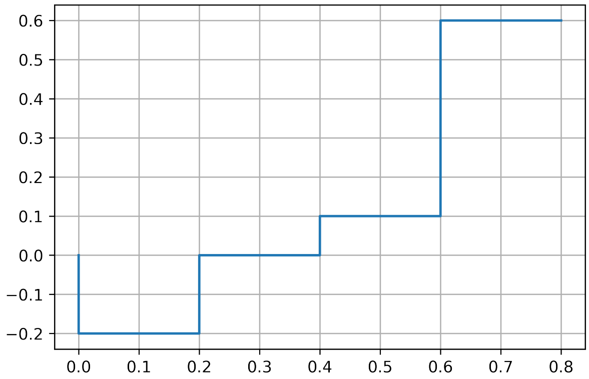
Here we can model this model in python. We have already seen these as functions of voltage.



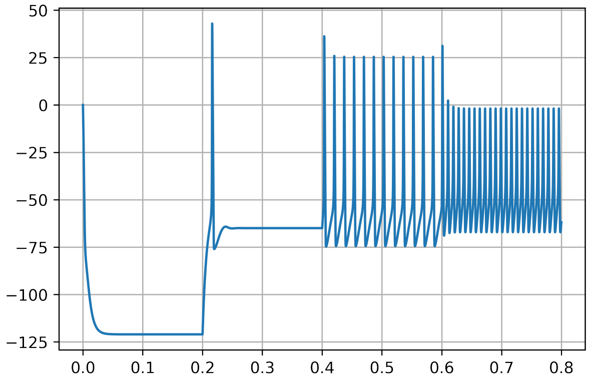
These are the τ's in case you are interested.



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 $V_{in} = RJ_{in}$.

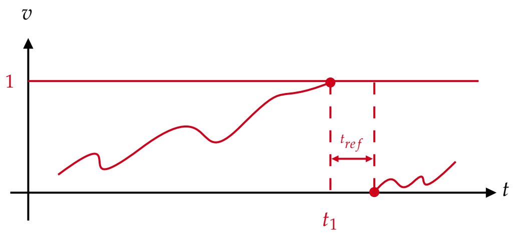
Thus, the voltage can be modelled as

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

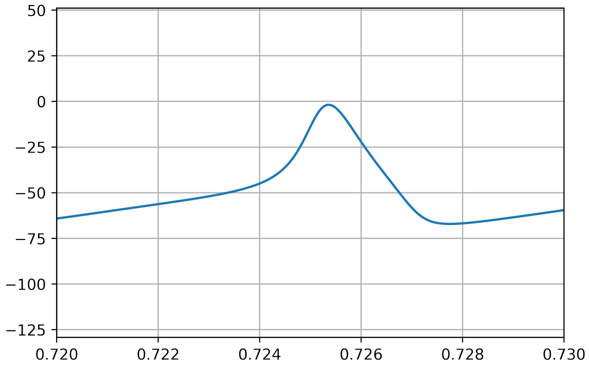
So this is the dynamics of the sub-threshold membrane potential. Change of variables: $v = \frac{V-V_L}{V_{th}-V_L}$, then $v \to 0$ if $v_{in} = 0$ and $v = 1$ is the threshold. Then we end up with a DE:

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

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.

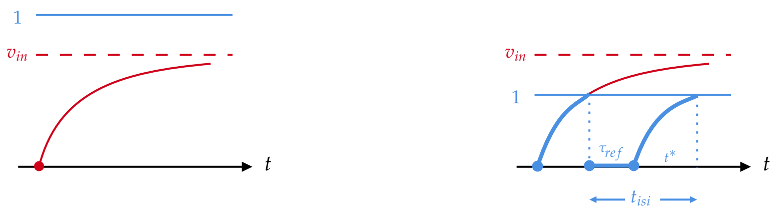
Claim

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

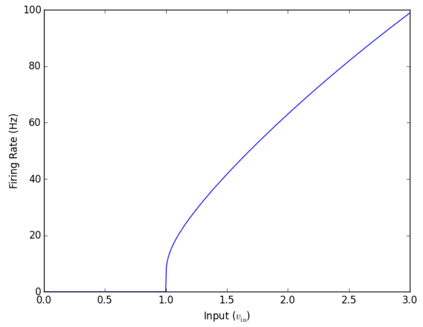


where t_{isi} stands for interspike interval.

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

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

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):

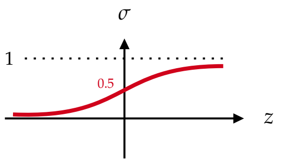
- $\tau_{ref} = 0.002$s
- $\tau_m = 0.02$s

Let's take a look at even simpler neurons.

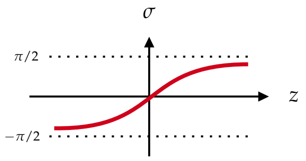
1.5 Activation functions

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.

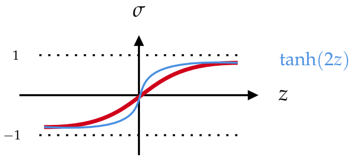
Logistic Curve $\sigma(z) = \frac{1}{1+e^{-z}}$



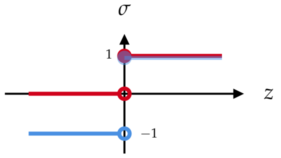
Arctan $\sigma(z) = \arctan(z)$



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



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

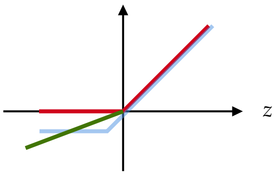


two versions

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

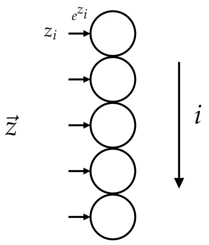


Multi-Neuron Activation Functions: Some activation functions depend on multiple neurons. Here are two examples.

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

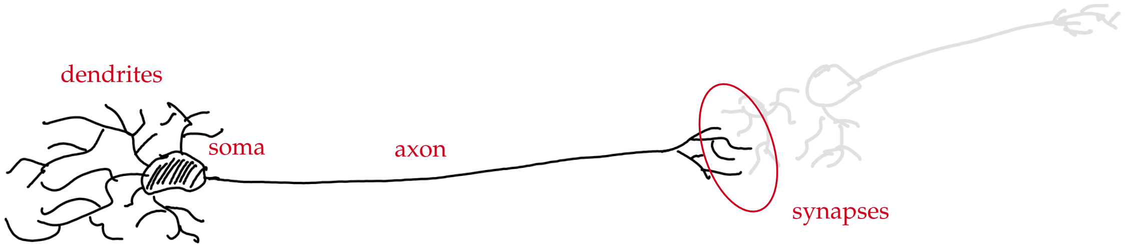
One-Hot is the extreme of the softmax, where only the largest element remains nonzero, while the others are set to zero.

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

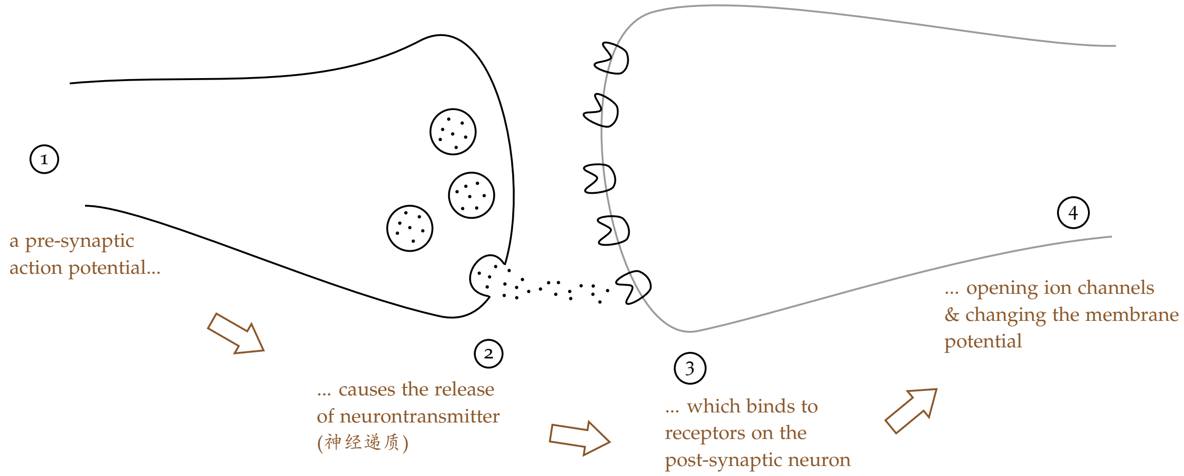
1.6 Synapses

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.

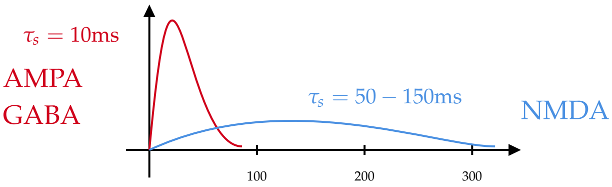


Even though an action potential is very fast, the synaptic processes by which it affects the next neuron takes time. Some synapses are fast (taking just about 10 ms), and some are quite slow (taking over 300 ms). If we represent that time constant using τ_s, then the current entering the post-synaptic 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 $\displaystyle\int_0^\infty h(t)dt = 1 \implies k = \frac{1}{n!\tau_s^{n+1}}$.

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

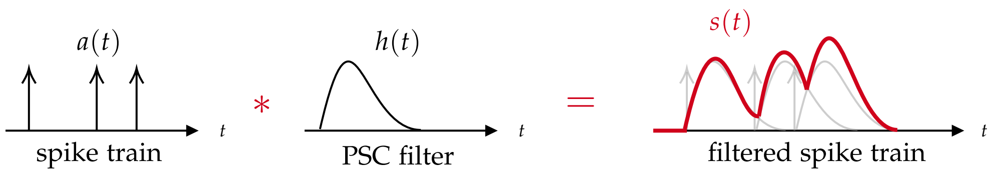
Dirac Delta Function is defined as

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

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,

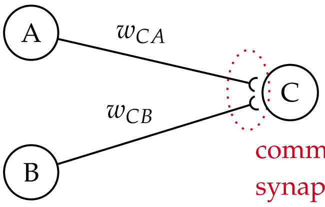
$$s(t) = (a * h)(t) = \sum_p h(t - t_p) = \text{sum of PSC filters, one for each spike}$$

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.

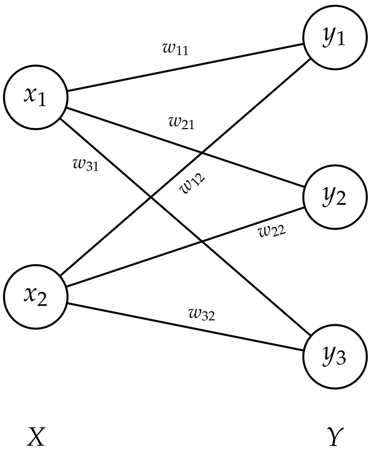


common way of writing synaptic connections

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,

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

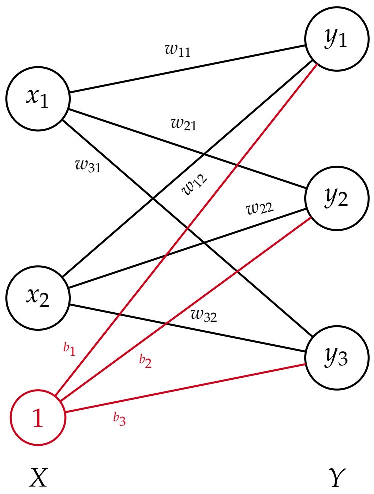
We can compute the input to the nodes in Y using

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

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, $\vec{b}$,



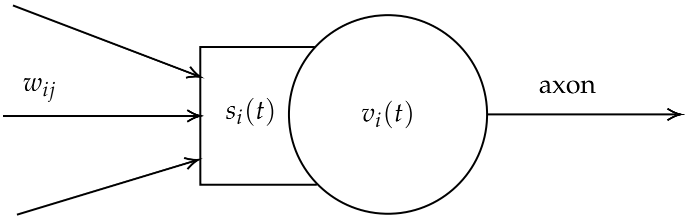
So $W\vec{x}+\vec{b} = \left[\, W|\vec{b}\,\right]\left[\frac{\vec{x}}{1}\right] = \hat{W}\left[\frac{\vec{x}}{1}\right]$.

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

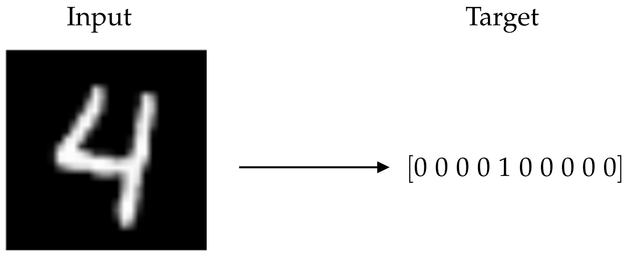
Getting a neural network to do what you want usually means finding a set of connection weights that yield the desired behaviour. That is, neural learning is all about adjusting connection weights.

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.



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:

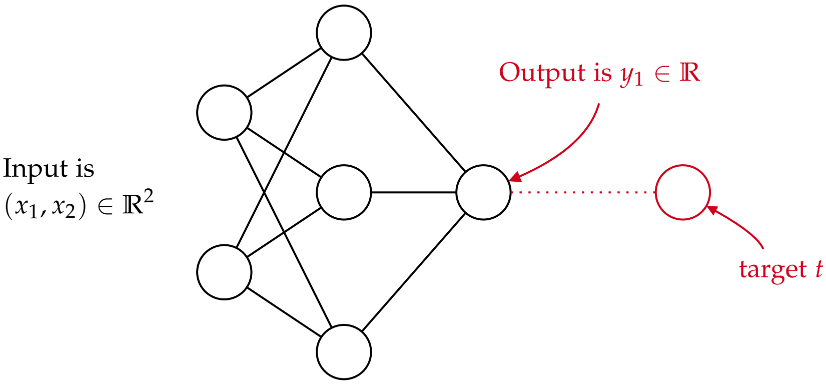
When playing a game of chess, a person knows their play was good if they win the game. They can try to learn from the moves they made.

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:



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	$\text{XOR}(A, B)$
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.

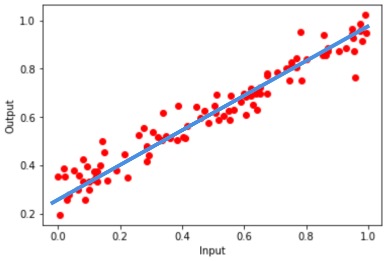
Regression

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

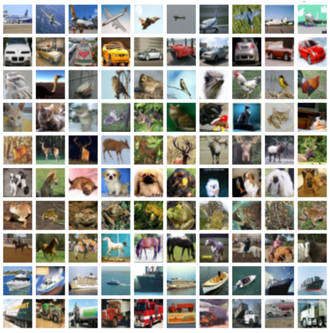
Outputs fall into a number of distinct categories.

Example: MNIST

MNIST stands for Modified National Institute of Standards and Technology database.

Inputs	Targets	Inputs	Targets
	[0 0 0 0 0 0 0 1 0 0]		[0 0 0 0 0 1 0 0 0 0]
	[1 0 0 0 0 0 0 0 0 0]		[0 0 0 0 1 0 0 0 0 0]
	[0 0 0 0 0 0 1 0 0 0]		[0 0 0 0 0 0 0 0 0 1]

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

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

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.

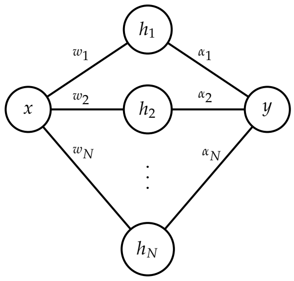
2.2 Universal Approximation Theorem

Can we approximate any function using a neural network?

Given a function $f(x)$, can we find weights ω_j, α_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

$$|G(x) - f(x)| < \epsilon \qquad \forall x \in I_n.$$

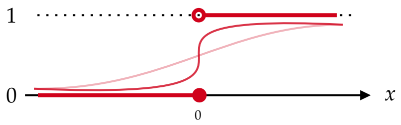
Here $C(I_n)$ denotes the continuous functions on I_n, and I_n can be $I_n = [0,1]^n$.

A function σ is "sigmoidal" if $\sigma(x) = \begin{cases} 1 & \text{as } x \to \infty \\ 0 & \text{as } x \to -\infty \end{cases}$

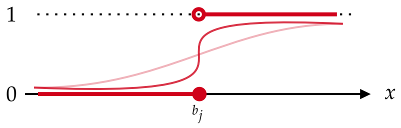
The theorem states that $\exists N$, and $\exists w_j, \theta_j, \alpha_j$ for $j = 1, \dots, N$ such that $|G(x) - f(x)| < \epsilon$.

Proof:

Suppose we let $w_j \to \infty$ for $j = 1, \dots, N$, 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:



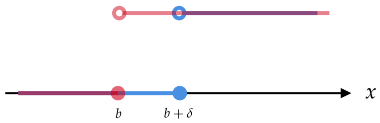
Or let's look at a shifted version of it: $\sigma(w_j(x - b_j)) \xrightarrow{w_j \to \infty} \begin{cases} 0 & \text{for } x \leq b_j \\ 1 & \text{for } x > b_j \end{cases}$.



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

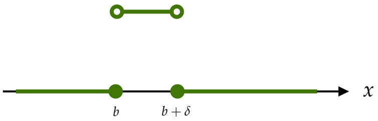
Define $H(x;b) := \lim_{w \to \infty} \sigma(w(x-b))$ which has two inputs: x and the shift b.

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

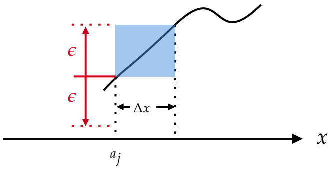


Then,

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



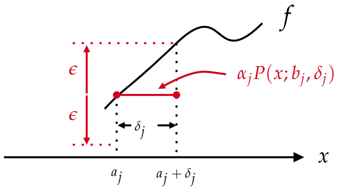
Since $f(x)$ is continuous, $\lim_{x\to a} f(x) = f(a)$, $\forall a \in I_n$. Then there exists an interval, $(a_j, a_j + \Delta x)$ such that $|f(x) - f(a_j)| < \epsilon \ \forall x \in (a_j, a_j + \Delta x)$.



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

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

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



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

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

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 $E(\theta) = \mathbb{E}\Big[L\big(f(x;\theta),t(x)\big)\Big]_{x\in data}$. 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., $(\bar{a},\bar{b}) = \operatorname{argmax}_{(a,b)} E(a,b)$. No matter where you are, "uphill" is in the direction of the gradient vector,

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

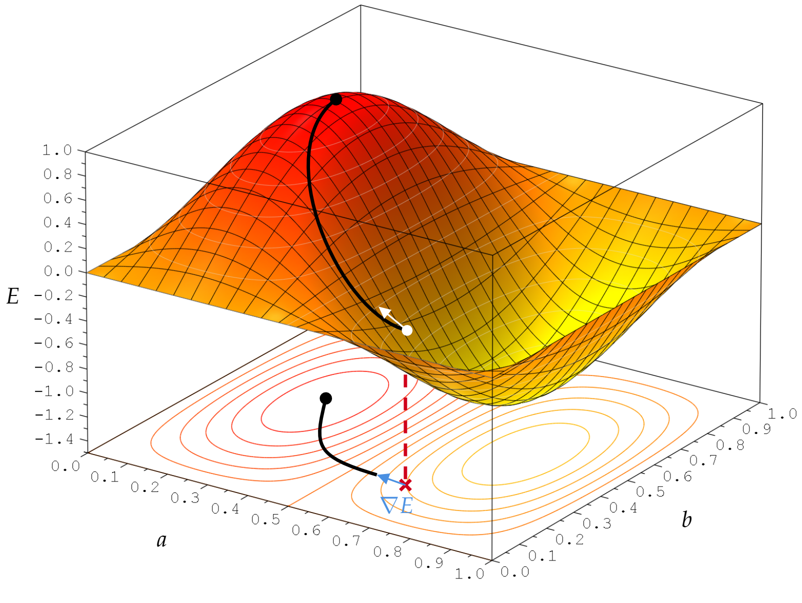


Image from https://commons.wikimedia.org/wiki/File:2D_Wavefunction_(2,1)_Surface_Plot.png.

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.

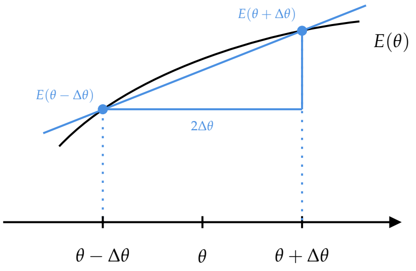
3.1.2 Approximating the Gradient Numerically

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

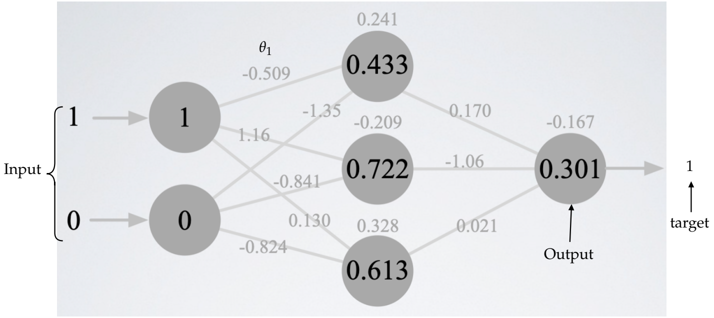
Finite-Difference Approximation

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 $\theta_1 = -0.509 + \underset{\substack{\uparrow \\ \Delta\theta}}{0.1} = -0.409$. The our output is $y = 0.302$. This yields $E(-0.409) = 1.198$.

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

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

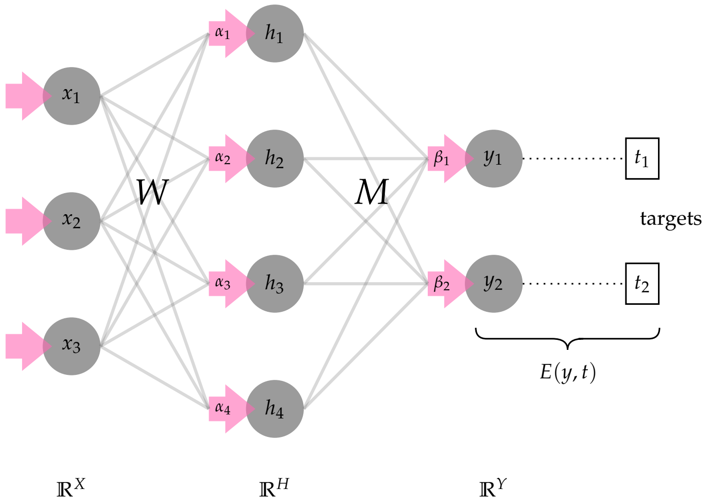
$$\frac{\partial E}{\partial \theta_1} \approx \frac{E(-0.409) - E(-0.609)}{2 \times 0.1} = -0.0292$$

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

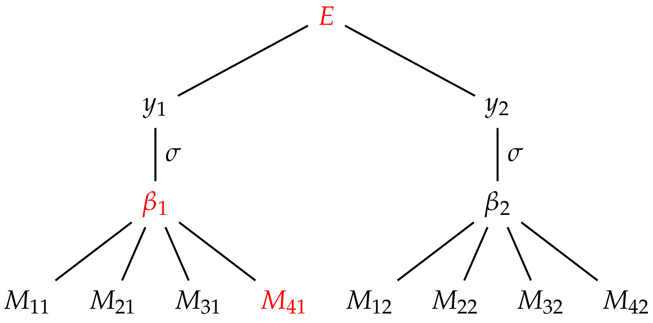
3.2 Error Backpropagation

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:



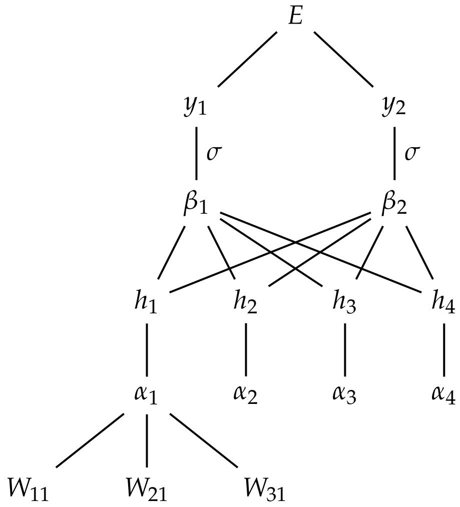
α_i is the input current to hidden node i. β_j is the input current to the output node j. For our cost (loss) function, we will use $E(y,t)$. For learning, suppose we want to know $\frac{\partial E}{\partial M_{41}}$, where M_{41} is going from h_4 to β_1. We can represent this by a computation/dependency 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, $\beta_1 = \sum_{i=1}^{4} h_i M_{i1} + b_1$, 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 $\alpha_1 = \sum_{j=1}^{3} x_j W_{j1} + a_1$. Therefore, $\frac{\partial \alpha_1}{\partial W_{21}} = x_2$. And

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

$(*)$: 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,

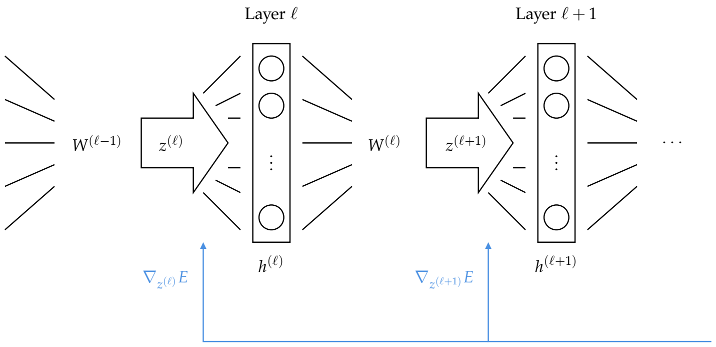
$$\begin{bmatrix}\frac{\partial E}{\partial \alpha_1} & \cdots & \frac{\partial E}{\partial \alpha_H}\end{bmatrix} = \begin{bmatrix}\frac{\mathrm{d}h_1}{\mathrm{d}\alpha_1} & \cdots & \frac{\mathrm{d}h_H}{\mathrm{d}\alpha_H}\end{bmatrix} \odot \begin{bmatrix}\frac{\partial E}{\partial \beta_1} & \cdots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

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.



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.

4

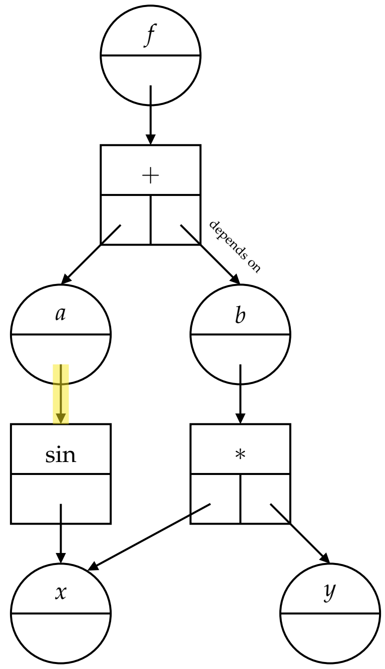
Automatic Differentiation

4.1 Theory

Consider $f = \underbrace{\sin(x)}_{a} + \underbrace{xy}_{b}$.

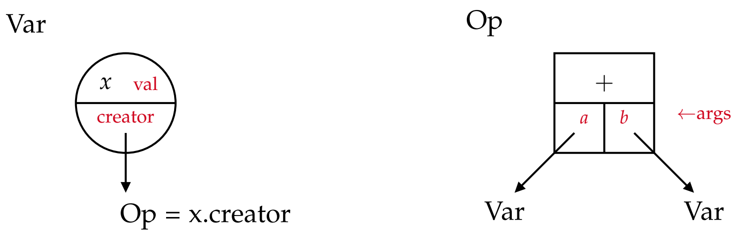
```
x = var
y = var
a = sin(x)
b = x * y
f = a + b
```

Let's draw the computation graph/expression graph.

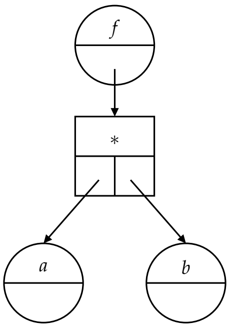


We refer the yellow line by "creator" pointer/reference: a was created from the sine function.

We will build a data structure to represent the expression graph using two different types of objects: Variables & Operations



Let's do another example: $f = a * b$. Given Var objects a and b, then



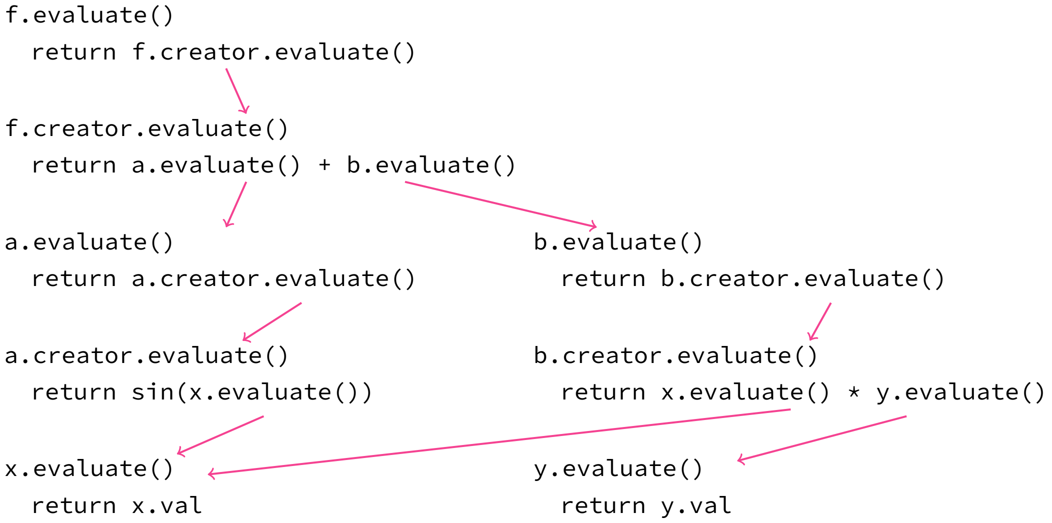
1. Create the Op object
2. Save references to the args (a, b)
3. Create a variable for the output (f)
4. Sent `f.creator` to this Op

4.1.1 Evaluate

We can use the expression graph to evaluate the expression. Each type of object has an `evaluate` function.

```
# Var.evaluate
if creator is None:
    return val
else:
    return creator.evaluate()

# Op.evaluate
call evaluate on all the args.
compute & return the value
```

Here is how we do evaluate on the previous example: $f = \sin(x) + xy$



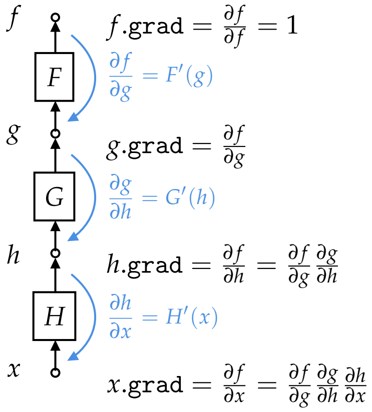
```
f.evaluate()
  return f.creator.evaluate()

f.creator.evaluate()
  return a.evaluate() + b.evaluate()

a.evaluate()                        b.evaluate()
  return a.creator.evaluate()         return b.creator.evaluate()

a.creator.evaluate()                b.creator.evaluate()
  return sin(x.evaluate())            return x.evaluate() * y.evaluate()

x.evaluate()                        y.evaluate()
  return x.val                        return y.val
```

4.1.2 Differentiate

The expression graph can also be used to compute the derivatives. Each Var stores the derivative of the expression w.r.t. itself. It stores it in its member `grad`.

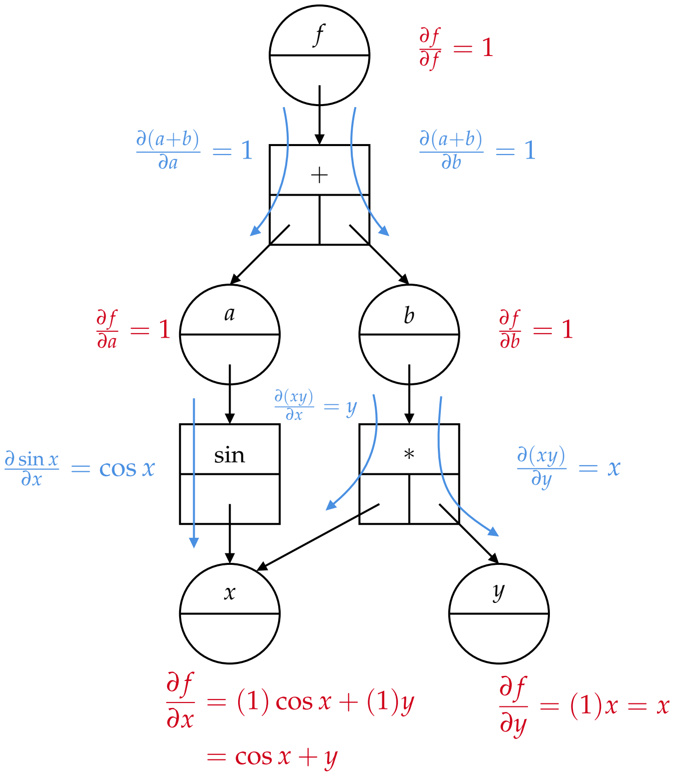
Consider $f = F(G(H(x)))$. For simplicity, denote $h = H(x), g = G(h), f = G(g)$. We want to find $x.\texttt{grad} = \frac{\partial f}{\partial x}$, which is partial derivative of the full expression with respect to variable x.

$$
\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{\partial F}{\partial g}\frac{\partial G(H(x))}{\partial x} \\
&= \frac{\partial F}{\partial g}\frac{\partial G(h)}{\partial h}\frac{\partial H(x)}{\partial x} \\
&= \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}
\end{aligned}
$$


f
F
g
G
h
H
x
$f.\texttt{grad} = \frac{\partial f}{\partial f} = 1$
$\frac{\partial f}{\partial g} = F'(g)$
$g.\texttt{grad} = \frac{\partial f}{\partial g}$
$\frac{\partial g}{\partial h} = G'(h)$
$h.\texttt{grad} = \frac{\partial f}{\partial h} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}$
$\frac{\partial h}{\partial x} = H'(x)$
$x.\texttt{grad} = \frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$

Starting with a value of 1 at the top, we work our way down through the graph, and increment `grad` of each Var as we go. Here "increment" does not necessarily mean "add"; in chain rule, it means multiplying. Each Op contributes its factor (according to chain rule), and passes the updated derivative down the graph.

Let's revisit the example above: $f = \sin(x) + xy$.


f
$\frac{\partial f}{\partial f} = 1$
$\frac{\partial(a+b)}{\partial a} = 1$
$+$
$\frac{\partial(a+b)}{\partial b} = 1$
$\frac{\partial f}{\partial a} = 1$
a
b
$\frac{\partial f}{\partial b} = 1$
$\frac{\partial(xy)}{\partial x} = y$
$\frac{\partial \sin x}{\partial x} = \cos x$
sin
$*$
$\frac{\partial(xy)}{\partial y} = x$
x
y
$\frac{\partial f}{\partial x} = (1)\cos x + (1)y$
$= \cos x + y$
$\frac{\partial f}{\partial y} = (1)x = x$

Each object has a `backward()` method that processes the derivative and passes it down the graph.

```
class Var:
    # self.var, self.grad, and s all have to be the same shape
    def backward(s):
        self.grad += s
        self.creator.backward(s)

class Op:
    # s must match the shape of the operator’s output
    def backward(s):
        for x in self.args:
            x.backward(s · ∂Op/∂x)
```

Here, s is the accumulated derivative of the part of the expression above the Var/Op.

4.2 Neural Networks with Auto-Diff

4.2.1 Optimization

Consider a scalar function E that depends (possibly remotely) on some variable v. Suppose we want to minimize E with respect to v, i.e., $\min_v E(v)$. We can use gradient descent: $v \leftarrow v - k\nabla_v E(v)$.

Algorithm 1: Gradient Descent (using AD)

```
initialize $v, k$
construct an expression graph for $E$
while not converged do
    evalaute $E$ at $v$
    set gradients to zero (i.e., $\nabla_v E = v.\texttt{grad} = 0$)
    propagate derivatives down (increment $v.\texttt{grad}$)
    $v \leftarrow v - k \cdot v.\texttt{grad}$
```

4.2.2 Neural Learning

We use the same process to implement error backpropagation for neural networks, and we optimize w.r.t. the connection weights and biases.

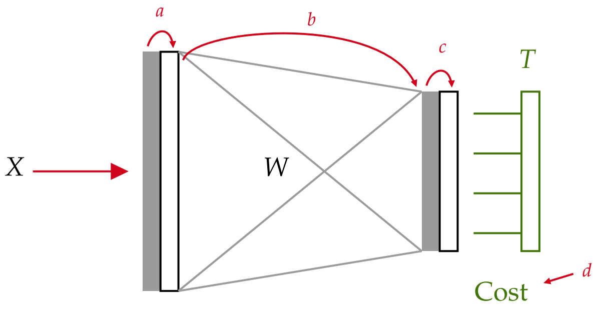
To accomplish this, our network will be composed of a series of layers, each layer transforming the data from the layer below it, culminating in a scalar-valued cost function.

Two types of operations in the network:

1. multiply by connection weights (including add bias)
2. apply activation function

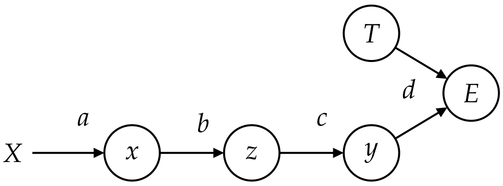
Finally, a cost function takes the output of the network, as well as the targets, and returns a scalar.

Consider this (very) small network:



X: input. Gray region represents the input current for that given layer. From the gray area to the outlined box, is the activation function, which we called a. Then we go through the connection weights, and we call it b. Then similarly c is the activation function. Lastly, we call the cost function d. Given dataset (X, T),

$$
\begin{aligned}
a &= \text{identity} \\
b &= (\lambda z : z \cdot W) \qquad \text{(multiply by } W\text{)} \\
c &= \text{logistic} \\
d &= \text{Cost}
\end{aligned}
$$



Each layer can be called like a function:

$$
\begin{aligned}
x &= a(X) && \text{e.g. } a(X) = X \\
z &= b(x) && \text{e.g. } b(x) = x \cdot W \\
y &= c(z) && \text{e.g. } c(z) = \sigma(z) \\
E &= d(y, T) && \text{e.g. } d(y, T) = \mathbb{E}\left[\tfrac{1}{2}\|y - T\|_2^2\right]
\end{aligned}
$$

Each layer, including the cost function, is just a function in a nested mathematical expression.

$$E = d\Big(c\Big(b\big(a(X)\big)\Big), T\Big)$$

Given that, neural learning is

$$W \leftarrow W - \kappa \nabla_W E$$

and in this case, W is a part of b function.

We construct our network using objects from our AD classes (Variables and Operations) so that we can take advantage of their `backward()` methods to compute the gradients. Net is basically a sequence of operations. For example, $net \equiv (a, b, c)$. And

$$
\begin{aligned}
y &= net(x) = c(b(a(x))) \\
E &= d(y, T)
\end{aligned}
$$

These two is called the forward pass, which sets the state of the network. State of the network means all the activations and input currents take on particular values. Given an input, and feed through the network, then all these input currents and neuron activations have actual values, which corresponds to that input, and corresponds to the output and the error.

Then we take gradient steps:

- set the gradients to zero: `E.zero_grad()`,
- then call `E.backward()`.

These two is called backward pass which sets all the gradients.

Algorithm 2: Neural learning using AD

```
Given dataset $(X, T)$, and network model $net$, with parameters $\theta$, and cost function "$Cost$"
for epochs... do
    // these two is the feedforward pass
    $y = net(X)$
    loss $= Cost(y, T)$
    // Backprop
    loss.zero_grad()
    loss.backward()
    // Gradient descent
    $\theta \leftarrow \theta - \kappa \cdot \theta$.grad
```

4.2.3 Matrix AD

To work with neural networks, our AD library will have to deal with matrix operations. For example, matrix addition. Suppose our scalar function involved a matrix addition. We have the cost function $L(\dots, A, B, \dots)$ where $A, B \in \mathbb{R}^{M\times N}$. What is $\nabla_A L$ and $\nabla_B L$?

Let $y = A + B \in \mathbb{R}^{M\times N}$, then

$$\nabla_A L = \underbrace{\nabla_y L}_{s} \odot \nabla_A y = s \odot 1_{M\times N}$$

which is of the same shape as A. L is a scalar function, and we take the gradient of the gradient with respect to every element in A,thus its shape is the same as A. Similarly,

$$\nabla_B L = \nabla_y L \odot \nabla_B y = s \odot 1_{M\times N}$$

which is of the same shape as B.

So the implementation:

```
+.backward(s)
   $A$.backward(s$\odot 1_{M\times N}$)
   $B$.backward(s$\odot 1_{M\times N}$)
```

Note that s is the same shape as y, the output of the operation.

Now let's talk about the matrix multiplication. Suppose we have the cost function $L(\dots, A, B, \dots)$ where $A \in \mathbb{R}^{M\times N}, B \in \mathbb{R}^{N\times K}$. Let $y = A \cdot B \in \mathbb{R}^{M\times K}$. What is $\nabla_A L$ and $\nabla_B L$?

$$\underset{M\times N}{\nabla_A L} = \underset{M\times K}{\nabla_y L} \cdot \underset{K\times N}{\nabla_A y} = s \cdot B^T$$
$$\underset{N\times K}{\nabla_B L} = \underset{N\times M}{\nabla_B y} \cdot \underset{M\times K}{\nabla_y L} = A^T \cdot s$$

The implementation would be

```
·.backward(s)
   $A$.backward(...)
   $B$.backward(...)
```

This is basically what you need now to apply a matrix type of library of automatic differentiation routines or classes, and apply them to neural networks. So the neural network is neural learning, or essentially backprop by constructing your network of a whole bunch of matrix variables and matrix operations, each of which has a backward function and knows how to contribute its derivative to a chain. So if you build your network out of these functions, it constructs the computation graph for you, and you can just call backward and it'll go down through the graph and populate all the gradients, and then you can pull out and use those gradients to do gradient descent.

5 Generalizability

5.1 PyTorch

PyTorch is an open source, free programming library that's particularly useful for implementing neural networks. So here we will demonstrate some of the features.

First, we are going to have some libraries.

```
import numpy as np
import torch
import matplotlib.pyplot as plt
import copy
```

5.1.1 PyTorch Tensors

Instead `Mat` type we created for AD, PyTorch has its own variable type, called a `Tensor`.

```
>>> x = torch.tensor([[1,2],[3,4.]], dtype=torch.float)
>>> print(x)
tensor([[1., 2.],
        [3., 4.]])
```

PyTorch has a whole library of operations to apply to tensors. Let's try to feed it to the log:

```
>>> y = torch.log(x)
>>> print(y)
tensor([[0.0000, 0.6931],
        [1.0986, 1.3863]])
```

PyTorch has auto-differentiation functionality.

Below, `z` is a node in the expression graph that yields `y`.

```
>>> z = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
>>> y = torch.prod(z)   # y = z[0]*z[1] = 1*2
>>> y.backward()
>>> z.grad      # dy/dz = [2,1]
tensor([2., 1.])
```

Note that if we don't have `requires_grad=True`, it won't offer `backward` function.

We can also temporarily suspend autograd.

```
z = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
y1 = torch.prod(z)      # dy1/dz = [2,1]
with torch.no_grad():
    y2 = torch.sum(z)   # dy2/dz = [1,1]
y = y1 + y2
y.backward()
print(z.grad)
```

This gives us `tensor([2., 1.])`. Without `no_grad`, this gives us

```
z = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
y1 = torch.prod(z)  # dy1/dz = [2,1]
y2 = torch.sum(z)   # dy2/dz = [1,1]
y = y1 + y2
y.backward()

>>> print(z.grad)
tensor([3., 2.])
```

It's pretty easy to convert back and forth between `numpy` arrays and `torch` tensors. For example, let's convert an array to a tensor:

```
>>> zn = np.array([1,2], dtype=float)
>>> zt = torch.tensor(zn)            # makes a copy
>>> print(f’torch version: {zt}’)
torch version: tensor([1., 2.], dtype=torch.float64)
```

It's also simple to convert a tensor to an array:

```
>>> zt = torch.tensor([1,2], dtype=torch.float)
>>> zn = zt.numpy()                  # does NOT make a copy
>>> print(f’numpy version: {zn}’)
>>> zn[0] = -1000.
>>> print(zt)
numpy version: [1. 2.]
tensor([-1000.,     2.])
```

Another thing to keep in mind is when we want to turn a tensor into a `numpy` array, we might run into trouble:

```
>>> zt = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
>>> zn = zt.numpy()
>>> print(f’numpy version: {zn}’)
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-8-dfae4b9d9e05> in <module>
      1 zt = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
----> 2 zn = zt.numpy()
      3 print(f’numpy version: {zn}’)
RuntimeError: Can’t call numpy() on Tensor that requires grad. Use tensor.detach().
    numpy() instead.
```

This is because numpy can't handle all these gradient information. Instead, we have to detach the tensor from the expression graph:

```
>>> zt = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
>>> zn = zt.detach().numpy()    # detaches zt from the expression graph, so grad is
     gone
>>> print(f'numpy version: {zn}')
numpy version: [1. 2.]
```

When a tensor is used in multiple calculations, its gradients are be *added* together (when you specify retain_graph=True).

```
>>> z = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
>>> y = torch.prod(z)  # dy/dz = [2,1]
>>> y.backward(retain_graph=True)
>>> z.grad
tensor([2., 1.])

>>> b = torch.sum(z)  # db/dz = [1,1]
>>> b.backward()
>>> z.grad
tensor([3., 2.])
```

5.1.2 Classification Dataset

We will use PyTorch's Dataset class.

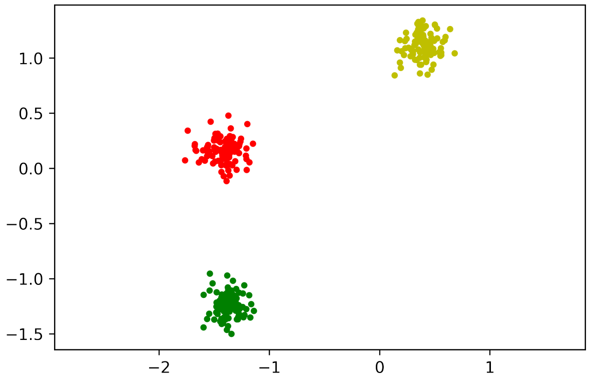
```
from torch.utils.data import Dataset, DataLoader
```

```
>>> B = torch.eye(3)
>>> A = 3.*(torch.rand((3, 6)) - 0.5)
>>> print('Input vectors')
>>> print(A)
>>> print('Target vectors')
>>> print(B)
Input vectors
tensor([[-1.3455,  0.6580, -0.2041, -0.0052,  0.2841, -0.6806],
        [-0.3096,  0.4767,  0.0416,  0.8221,  0.3553, -0.9552],
        [ 0.0603,  1.0587,  0.6078, -0.5603,  0.9886, -1.1840]])
Target vectors
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

```
class DiscreteMapping(Dataset):
    def __init__(self, A, B, n=300, noise=0.1):
        self.samples = []
        self.n_classes = len(A)
        self.input_dim = len(A[0])
        for i in range(n):
            r = np.random.randint(self.n_classes)
            t = B[r]
            sample = [A[r]+noise*torch.randn_like(A[r]), t]
            self.samples.append(sample)

    def __getitem__(self, idx):
        return self.samples[idx]

    def __len__(self):
        return len(self.samples)

    def inputs(self):
        x = []
        for s in self.samples:
            x.append(s[0])
        return torch.stack(x)

    def targets(self):
        t = []
        for s in self.samples:
            t.append(s[1])
        return torch.stack(t)

    def classes(self):
        c = []
        for s in self.samples:
            k = torch.argmax(s[1])
            c.append(k)
        return torch.tensor(c, dtype=torch.long)

    def plot(self, labels=[], idx=(0,1), equal=True):
        X = self.inputs()
        if len(labels)==0:
            labels = self.classes()
        colour_options = [’y’, ’r’, ’g’, ’b’, ’k’]
        cidx = self.classes()
        colours = [colour_options[k] for k in cidx]
        plt.scatter(X[:,idx[0]], X[:,idx[1]], color=colours, marker=’.’)

        if equal:
            plt.axis(’equal’);
```

Let’s now create a dataset and plot it:

```
ds = DiscreteMapping(A, B)
ds.plot
```



We can look at the inputs, targets and classes:

```
>>> print(ds.inputs()[:5])
tensor([[ 0.0648,  1.0591,  0.3801, -0.4864,  0.9960, -1.1367],
        [-0.4479,  0.5487,  0.1092,  0.7356,  0.3301, -0.9817],
        [ 0.0445,  1.0273,  0.5690, -0.5552,  1.1006, -1.1681],
        [ 0.0491,  1.0013,  0.7913, -0.5124,  1.0008, -1.1813],
        [-0.0058,  1.0477,  0.7242, -0.7067,  0.8411, -1.2293]])

>>> print(ds.targets()[:5])
>>> print(ds.classes()[:5])
tensor([[0., 0., 1.],
        [0., 1., 0.],
        [0., 0., 1.],
        [0., 0., 1.],
        [0., 0., 1.]])
tensor([2, 1, 2, 2, 2])
```

5.1.3 Neural Networks with PyTorch

Now let's make some neural networks. We can use the `torch.nn` module. The simplest version is using `Sequential`. Here we assume that each layer is connected to the previous layer.

```
net = torch.nn.Sequential(
    torch.nn.Linear(6,10),
    torch.nn.ReLU(),
    torch.nn.Linear(10,5),
    torch.nn.ReLU(),
    torch.nn.Linear(5,3),
    torch.nn.LogSoftmax(dim=1))
```

We have more flexibility with `Module`, since we get to specify the `forward` function.

```
class mynet(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.lyr = torch.nn.ModuleList()    # self.lyr = []  DOES NOT WORK
        self.lyr.append(torch.nn.Linear(6,10, bias=False))
        self.lyr.append(torch.nn.Sigmoid())
        self.lyr.append(torch.nn.Linear(10,5))
        self.lyr.append(torch.nn.Sigmoid())
        self.lyr.append(torch.nn.Linear(5,3))
        self.lyr.append(torch.nn.LogSoftmax(dim=1))

    def forward(self, x):
        # Here is where you can be creative.
        y = x
        for l in self.lyr:
            y = l(y)
        return y
```

Then we can create the network model:

```
net = mynet()
```

Now we need a loss funciton.

```
# loss = torch.nn.CrossEntropyLoss(reduction='mean')
loss_fcn = torch.nn.NLLLoss(reduction='mean')  # <=== Choose a cost function
```

Then we can push our datasets input through the network and compute the loss. Notice that `NLLLoss` wants the class index/label indices.

```
>>> y = net(ds.inputs())
>>> loss = loss_fcn(y, ds.classes())
>>> print(loss)
tensor(1.1283, grad_fn=<NllLossBackward>)
```

Then we can look at the output of the network:

```
>>> y[:5]
tensor([[-0.9371, -0.9805, -1.4561],
        [-0.9358, -0.9831, -1.4544],
        [-0.9361, -0.9867, -1.4481],
        [-0.9358, -0.9827, -1.4548],
        [-0.9355, -0.9875, -1.4477]], grad_fn=<SliceBackward>)
```

which will be the log of `SoftMax`, and the corresponding classes:

```
>>> ds.classes()[:5]
tensor([1, 1, 0, 1, 0])
```

Currently, the network hasn't been trained yet.

Network weights and biases are accessible through `net.parameters()`.

```
>>> for p in net.parameters():
>>>     print(p.shape)
torch.Size([10, 6])
torch.Size([5, 10])
torch.Size([5])
torch.Size([3, 5])
torch.Size([3])
```

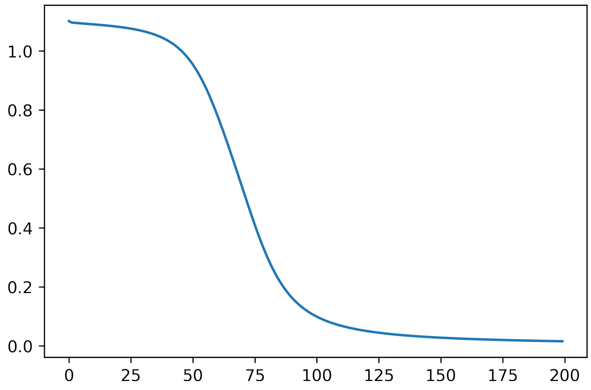
Another way to look at it:

```
>>> params = list(net.parameters())
>>> print(params[3])
Parameter containing:
tensor([[ 0.2359, -0.2637, -0.2381,  0.3370, -0.0659],
        [-0.1568,  0.1390, -0.0219, -0.1441,  0.4145],
        [ 0.1928, -0.4072, -0.2311,  0.2990,  0.1112]], requires_grad=True)
```

Now let's train the classification model myself. We can access the parameter gradients and implement gradient descent ourselves.

```
x = ds.inputs()
targets = ds.targets()
classes = ds.classes()
```

```
net = mynet()
lrate = 1.
n_epochs = 200
losses = []
for epoch in range(n_epochs):
    y = net(x)
    err = loss_fcn(y, classes) # for CE
    losses.append(err.item())
    net.zero_grad()
    err.backward()
    with torch.no_grad():
        for p in net.parameters():
            p -= lrate * p.grad
plt.plot(losses);
```



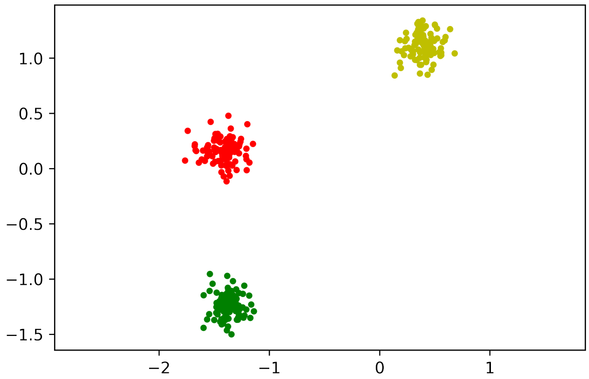
After we trained the network,

```
>>> y = net(x)
>>> print(y[:5])
>>> print(torch.exp(y[:5]))
tensor([[-5.1984, -0.0158, -4.5924],
        [-5.1438, -0.0130, -4.9449],
        [-0.0168, -4.6187, -4.9900],
        [-5.2278, -0.0139, -4.7713],
        [-0.0177, -4.5456, -4.9715]], grad_fn=<SliceBackward>)
tensor([[0.0055, 0.9843, 0.0101],
        [0.0058, 0.9870, 0.0071],
        [0.9833, 0.0099, 0.0068],
        [0.0054, 0.9862, 0.0085],
        [0.9825, 0.0106, 0.0069]], grad_fn=<ExpBackward>)
```

which matches the targets:

```
>>> targets[:5]
tensor([[0., 1., 0.],
        [0., 1., 0.],
        [1., 0., 0.],
        [0., 1., 0.],
        [1., 0., 0.]])
```

If we plot it using the output as the labels, `ds.plot(labels=y)`:

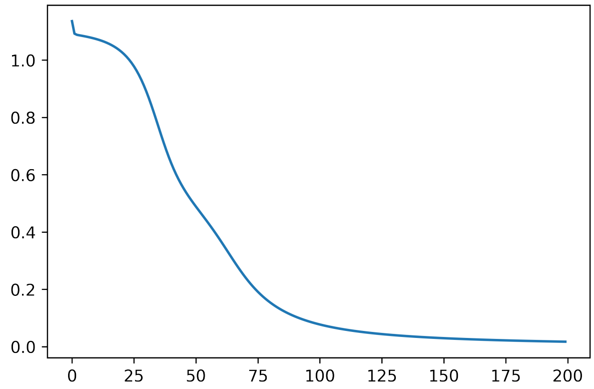


There are other methods that do these gradient descent stuff for us, which are in `optim` module.

```
net = mynet()
loss_fcn = torch.nn.NLLLoss(reduction='mean')
optim = torch.optim.SGD(net.parameters(), lr=1)
```

Then create the loop over epochs:

```
n_epochs = 200
losses = []
for epoch in range(n_epochs):
    y = net(x)
    err = loss_fcn(y, classes) # for CE
    losses.append(err.item())
    optim.zero_grad()
    err.backward()
    optim.step()
plt.plot(losses);
```

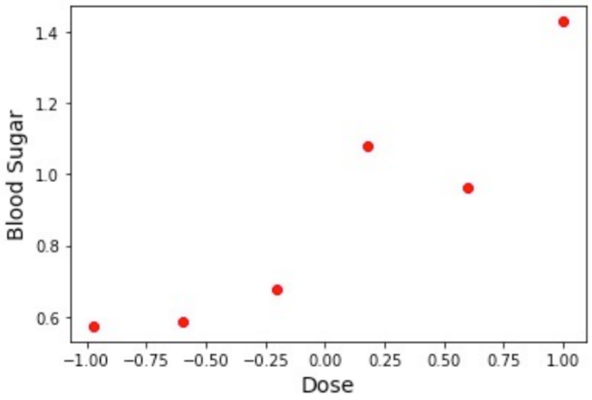


Then it works:

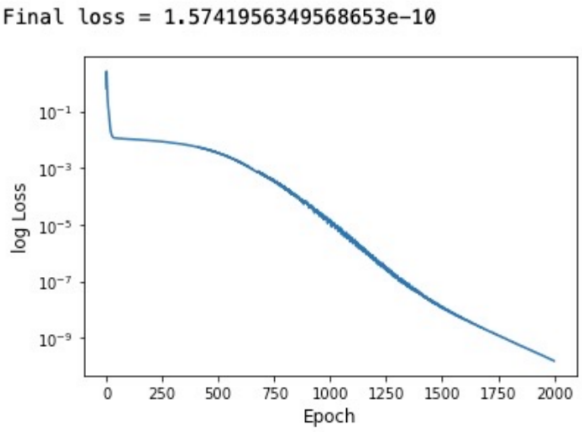
```python
y = net(x)
print(torch.exp(y[:5]))
print(classes[:5])
```

5.2 Overfitting

Suppose you have a dataset of drug dosage vs. your blood-sugar level. Your doctor would like to train a neural network so that, given a dose, she can predict your blood sugar. That dataset has 6 samples. And since this is a regression problem, we will use a linear activation function on the output, and MSE as a loss function.



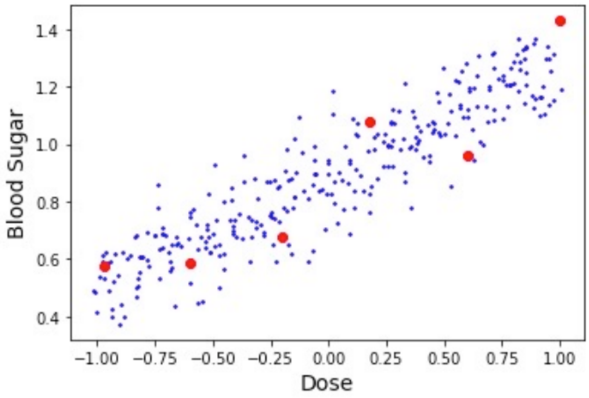
Your doctor creates a neural network with 1 input node, two hidden layers, each with 250 ReLU nodes, and 1 output node, and trains it on your dataset for 2000 epochs.



```
Final loss = 1.5741956349568653e-10
```

The doctor want to give you a dose of 0.65, so she uses the network to estimate what your blood sugar will be. The model says the blood sugar is 1.0043. Does this seem reasonable? If we take a look at the plot, this point is too close to the fifth point. So we don't necessarily want to fit all those points specifically, instead, we want to model the underlying phenomenon.

Suppose the doctor takes 300 more blood samples from you, at a variety of different doses. Once you're drained of blood, she runs the dataset through her model to see what the MSE loss is. This time, MSE loss is 0.018.

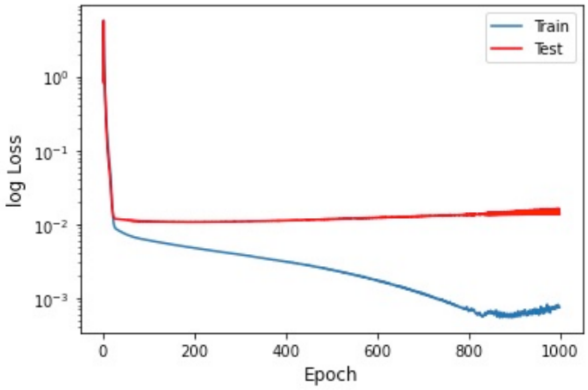


That's orders of magnitude worse than the 1.5×10^{-10} on the training dataset. The false sense of success we get from the results on our training dataset is known as *overfitting* or *overtraining*.

If your model has enough flexibility and you train it long enough for enough epochs, it will start to fit the specific points in your training dataset, rather than fit the underlying phenomenon that produced the noisy data.

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

A common practice is to keep some of your data as test data which your model does not train on.

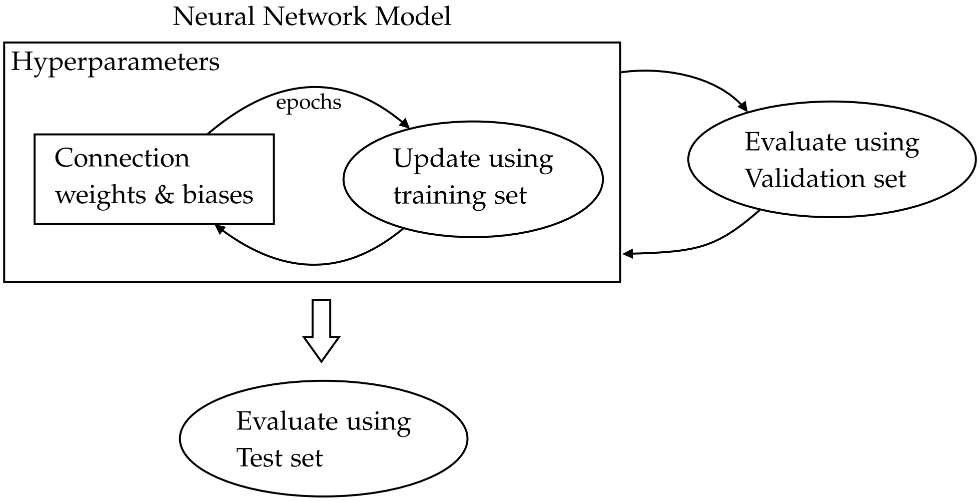


So we see the train data gets down as epochs go up. But in terms of the test data, we see it goes down first and then creeps up again. That's because the model is working so hard to fit the training data, including the noise of the training data, at the expense of fitting the test data. Thus divergence like this is a sign of overfitting.

5.3 Validation

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

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


Neural Network Model
Hyperparameters
epochs
Connection weights & biases
Update using training set
Evaluate using Validation set
Evaluate using Test set

5.4 Combatting Overfitting

We saw that if a model has enough degrees of freedom, it can become hyper-adapted to the training set, and start to fit the noise in the dataset. In that case, we see the training error is very small. This is a problem because the model does not generalize well to new samples: test error is much larger than training error. There are some strategies to try to stop our network from trying to fit the noise.

5.4.1 Regularization

Weight Decay

We can limit overfitting by creating a preference for solutions with smaller weights, achieved by adding a term to the loss function that penalizes for the magnitude of the weights. So let's our loss function before was E, now we are going to create a different one:

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

where

$$\|\theta\|_F = \sqrt{\sum_j \theta_j^2}$$

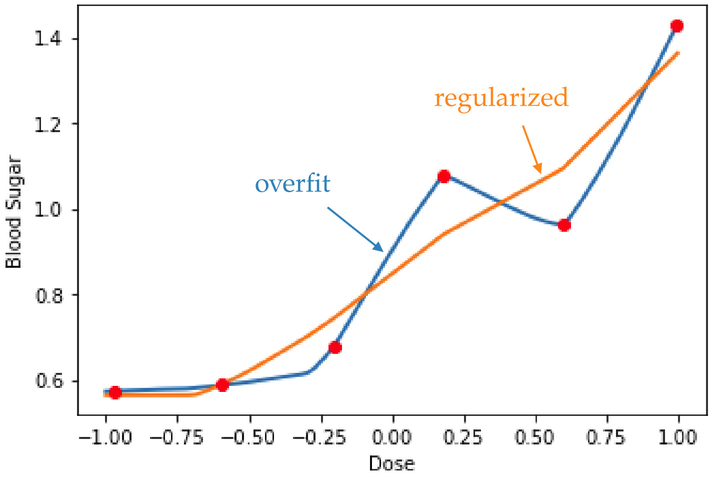
is the Frobenius norm.

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

$$\nabla_{\theta_i}\tilde{E} = \nabla_{\theta_i}E + \bar{\lambda}\theta_i \quad \longrightarrow \quad \theta_i \leftarrow \theta_i - \kappa\nabla_{\theta_i}E - \underbrace{(\bar{\lambda}\kappa)}_{\lambda}\theta_i$$

where the last term is decay term, which pulls it a bit closer down towards zero.

Now when apply this updated weight decay regularized update rule, we can see that our fit of small blood sugar data set is far less precisely hitting those points.



We don't really care about the training loss. Instead, we care about the test loss. After we run this two models with the test data, we get the original test loss is 0.01728 and weight decay test loss is 0.01219, which is smaller. If we take a look at weights themselves, the unregularized version: $||\theta||_F^2 = 254.4$ and the regularized version: $||\theta||_F^2 = 7.1$. We are limiting the solution we can achieve by preferring the smaller connection weights and biases.

Note that λ controls the weight of the regularization term.

One can also use different norms. For example, it is common to use the L_1 norm,

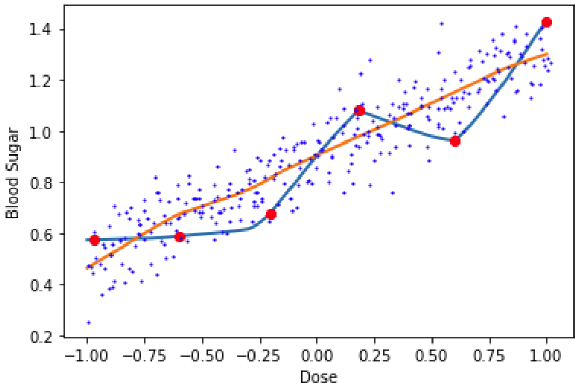
$$L_1(\theta) = \sum_i |\theta_i|,$$

The L_1 norm tends to favour sparsity (most weights are close to zero, with only a small number of non-zero weights).

5.4.2 Data Augmentation

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

For example, in our blood sugar/dose dataset, 6 points obviously is not a very comprehensive view of the underlying phenomenon. Thus we would want to have more points. If we train the model with 300 training samples, we get a more robust model.

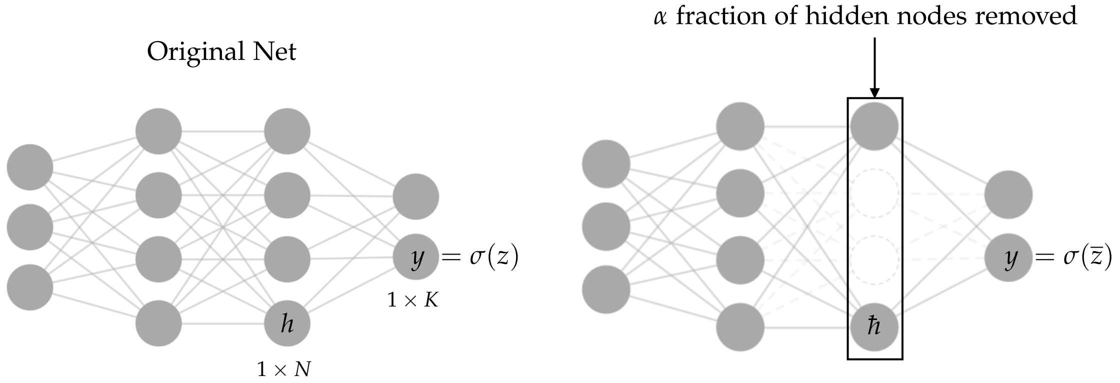


Original test loss is 0.01728 while augmented test loss is 0.00934.

Where does this extra data come from? For image-recognition datasets, one can generate more samples by shifting or rotating the images. Those transformations presumably do not change the labelling. More generally, we can make aby changes to whatever it is we want to make our model invariant to.

5.4.3 Dropout

The last method we will talk about is the most bizarre. While training using the dropout method, you systematically ignore a large fraction (typically at least half) of the hidden nodes for each sample. That is, given a dropout probability, α, each hidden node will be dropped with probability α. A dropped node is temporarily taken off-line and set to zero.



$$\hbar = \begin{cases} 0 & \text{with prob. } \alpha \\ h_n & \text{with prob. } 1-\alpha \end{cases}$$

where $\bar{z}$ is the new input current after we perform dropouts.

We do both a feedforward and backprop pass with this diminished network.

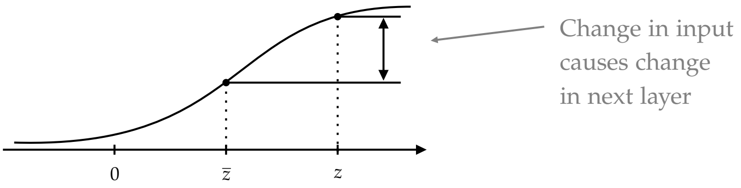
Consider the absolute input to the nodes in the output layer. Without dropout,

$$\mathbb{Z} = \sum_{k=1}^{K} |z_k| = \sum_{k=1}^{K} \big|[hW]_k\big|$$

With dropout, the *expected* absolute input

$$\sum_{k=1}^{K} |\bar{z}_k| = \sum_{k=1}^{K} \big|[\hbar W]_k\big| = \sum_{k=1}^{K} \big|[(1-\alpha)hW]_k\big| = (1-\alpha)\mathbb{Z}$$

Thus, a dropout rate α of reduces the expected input to the next layer by a factor of $1-\alpha$, which could affect the behaviour of the next layer.



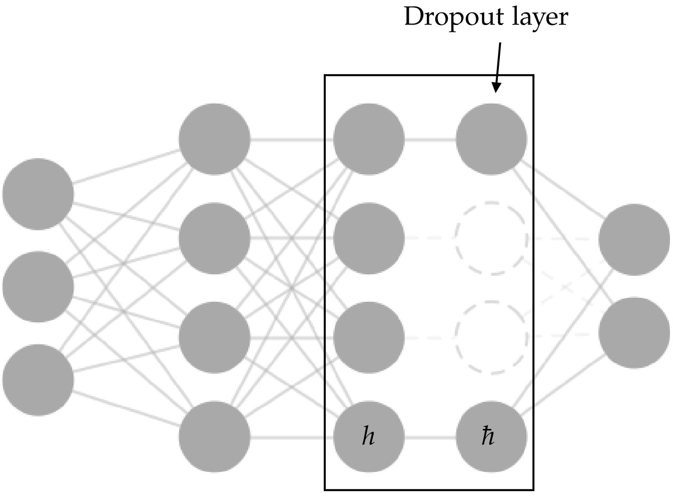
Change in input causes change in next layer

After training, we want the network to work without dropped nodes. The weights learned with dropout will not work properly in the full network.

Solution: We scale the output of the dropout layer up by a factor of $\frac{1}{1-\alpha}$. For example, if $\alpha = 0.8$, then $\frac{1}{1-\alpha} = 5$, then

$$\bar{z} = \frac{1}{1-\alpha}\hbar W \implies \sum_{k=1}^{K} |\bar{z}_k| = \mathbb{Z}.$$

We can accomplish all of this by adding another layer to the network.



The activation function of the dropout layer is

$$d(h) = \frac{1}{1-\alpha}\{h\}_\alpha \quad \text{where } \{h\}_\alpha = \begin{cases} 0 & \text{with prob. } \alpha \\ h & \text{with prob. } 1-\alpha \end{cases}$$

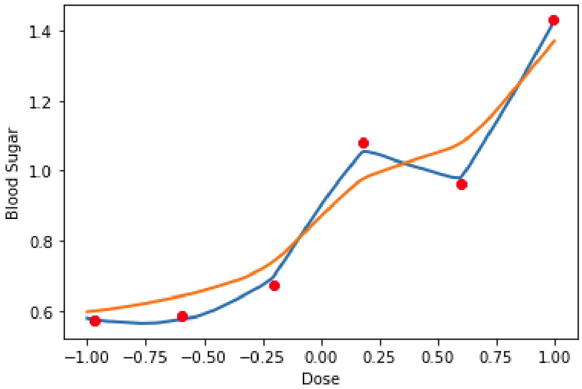
During backprop, the gradients have to go back through this layer.

$$\hbar = d(h), \qquad \nabla_h L = \nabla_\hbar L \frac{\mathrm{d}\hbar}{\mathrm{d}h}$$

where

$$\frac{\mathrm{d}\hbar}{\mathrm{d}h} = \begin{cases} 0 & \text{if } h \text{ was dropped} \\ \frac{1}{1-\alpha} & \text{if } h \text{ was not dropped} \end{cases}$$

Now after we have implemented this, we obtain



Original test loss is 0.01543 and dropout test loss is 0.01147.

Why does dropout work?

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

See `https://arxiv.org/abs/1207.0580`

6

Optimization Considerations

Now in this chapter, or in this week, we are going to discuss deep neural networks. We've looked at neural networks with a hidden layer, input layer, hidden layer, output layer, and now we're going to ask the question: what about adding more hidden layers? What are the pros and cons? So the goal is to see the advantages and disadvantages of deep neural networks and it's basically weighing representational power versus vanishing or exploding gradients. *How many layers should our neural network have?* Recall

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

$$|G(x) - f(x)| < \epsilon \qquad \forall x \in I_n.$$

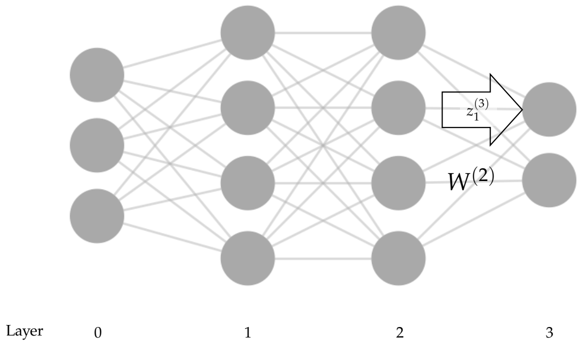
Thus, we really only ever need one hidden layer. But is that the best approach, in terms of number of nodes, or learning efficiency? No, it can be shown that such a shallow network could require an exponentially large number of nodes (i.e., A really big N) to work.

So, a deeper network is preferred in many cases. So, why don't we always use really deep networks?

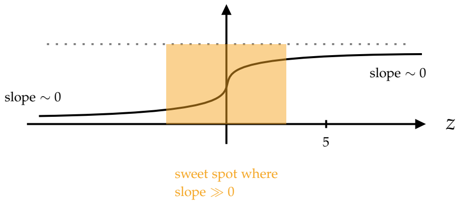
Let us visit one problem: **vanishing gradients**.

6.1 Vanishing Gradients

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



Let's say $z_1^{(3)} = 5$, and this node is logistic node, then $y_1 = \sigma\left(z_1^{(3)}\right) = \frac{1}{1+e^{-5}} = 0.9933$. Therefore, $\frac{\mathrm{d}y_1}{\mathrm{d}z_1^{(3)}} = y_1(1-y_1) = 0.0066$, which is small. Because the input current is high, then the derivative is small. Let's demonstrate it in a plot.



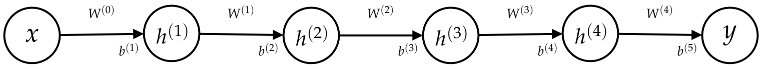
Compare that to if the input current was 0.1, then $y_1 = \sigma(0.1) = 0.525$, and $\frac{\mathrm{d}y_1}{\mathrm{d}z_1^{(3)}} = y_1(1-y_1) = 0.249$, which is almost 40 times bigger than before. Hence, the updates to the weights will be smaller when the input currents are large in magnitude.

What about the next layer down? Suppose $\nabla_{z^{(3)}} E \sim 0.01$. What if the inputs to the penultimate layer were around 4 in magnitude? Then the corresponding slopes of their sigmoid functions will also be small. In particular, $\sigma(4) = 0.982$, $\sigma'(4) = 0.0177$. Recall that

$$\nabla_{z^{(2)}} E = \sigma'\left(z^{(2)}\right) \odot \left(\nabla_{z^{(3)}} E \cdot \left(W^{(2)}\right)^T\right) \approx 0.0177 \odot \left(0.01 \cdot \left(W^{(2)}\right)^T\right) = 0.000177\left(W^{(2)}\right)^T$$

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

Here is another way to look at it. Consider this simple, but deep network.



where $W^{(i)}$ are weights and $b^{(i)}$ are biases. Let's start with the loss on the output side: $E(y,t)$. The gradient with respect to the input current of the output node is

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

Then using backprop, we can compute a single formula for

$$\frac{\partial E}{\partial z^{(4)}} = (y-t)W^{(4)}\sigma'\left(z^{(4)}\right).$$

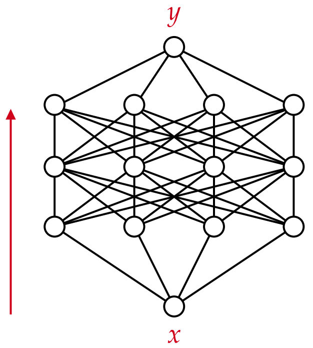
Going deeper... we have

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

What is the steepest slop that $\sigma(z)$ attains? $\sigma'(z) = \sigma(z)(1-\sigma(z))$ for $0 < \sigma(z) < 1$. All else being equal, the gradient goes down by a factor of at least 4 each layer. We can see this if we look at the norm of the gradients at each layer, i.e.,

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

Consider this particular example below:



5 layers (include input & output)

layer 4 norm $= 0.7598$ $[0.76]$

layer 3 norm $= 0.7598$ $[-0.038, -0.217, -0.039, -0.039]$

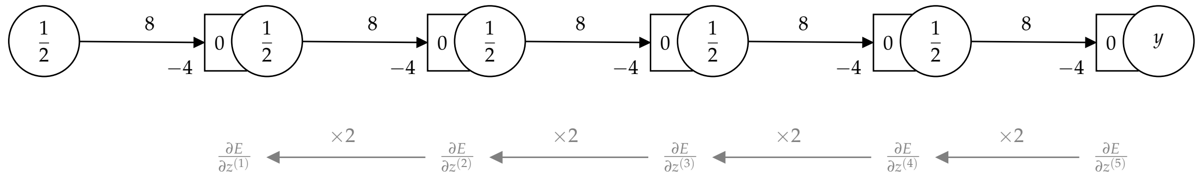
layer 2 norm $= 0.7598$ $[0.018, -0.018, -0.007, 0.018]$

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

We can see they start reasonably sized ,but they get smaller and smaller and smaller as we go down. We have the same number of nodes four at each layer, but they're getting smaller in value. This is just because the connection weights aren't big enough to compensate for the factor of $1/4$ that σ' lays on each layer.

6.2 Exploding Gradients

This is a similar, though less frequent phenomenon can result in very large gradients.


$\frac{1}{2}$
8
−4
0
$\frac{1}{2}$
8
−4
0
$\frac{1}{2}$
8
−4
0
$\frac{1}{2}$
8
−4
0
$\frac{1}{2}$
8
−4
0
y
$\frac{\partial E}{\partial z^{(1)}}$
×2
$\frac{\partial E}{\partial z^{(2)}}$
×2
$\frac{\partial E}{\partial z^{(3)}}$
×2
$\frac{\partial E}{\partial z^{(4)}}$
×2
$\frac{\partial E}{\partial z^{(5)}}$

$\sigma' = \frac{1}{4}$ because the input current is zero, which is the steepest part of logistic function. Then we multiply the weight, which is 8. Thus finally, we get a factor of 2. Therefore, we have

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

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

6.3 Enhancing Optimization

6.3.1 Stochastic Gradient Descent

Computing the gradient of the cost function can be very expensive and time-consuming, especially if you have a huge dataset. Remember the cost is the expected loss over the whole dataset. Assume we have D training samples, then the loss is

$$E(Y,T) = \frac{1}{D} \sum_{d=1}^{D} L(y_d, t_d).$$

Rather than compute the full gradient, we can try to get a cheaper estimate by computing the gradient from a random sampling. Let γ be a random sampling of B elements from $\{1, 2, \ldots, D\}$. Then we estimate $E(Y, T)$ using

$$E(Y, T) \approx E(\tilde{Y}, \tilde{T}) = \frac{1}{B} \sum_{b=1}^{B} L(y_{\gamma_b}, t_{\gamma_b}).$$

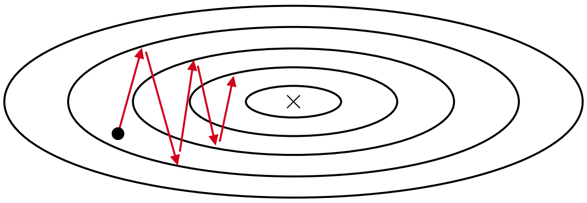
We refer to $\{(y_{\gamma_1}, t_{\gamma_1}), \ldots, (y_{\gamma_B}, t_{\gamma_B})\}$ as a *batch*. We use the estimate from this batch to update our weights, and then choose subsequent batches from the remaining samples. This method is called **Stochastic Gradient Descent**.

6.3.2 Momentum

Consider gradient descent optimization in these situations...

Case 1

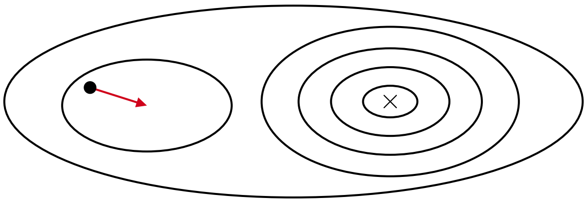
First let's consider the level curves of the objective function. Here x is the minimum we want to achieve, and we are starting at the $\bullet$.

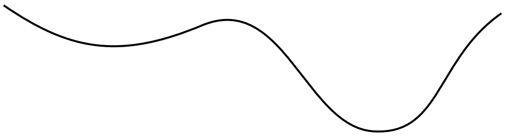


We can see it does a lot of oscillations and it will eventually work its way down. It's doing a lot of jumping back and forth across the valley, making hesitant progress along the bottom, but what we really really need to go is to go down the valley. This type of oscillation can be inefficient.

Case 2

In this case, we have a higher local minimum on the left. Using the conventional gradient descent, we would just move down to that higher local minimum and say "yay done". However, if we take a look at the side profile, what we really want to do is to get down in the lower value down on the right. The optimization stops in the shallow local minimum, but we would prefer to get into the deeper minimum.





A technique to improve our prospects in both situations is momentum. Thus far, we have been moving through parameter space by stepping in the direction of the gradient:[1]

$$\theta_{n+1} \leftarrow \theta_n - \kappa \nabla_{\theta_n} E.$$

[1] Note that here prof used a quite confusing notation: θ_n is the old set of weights and biases, while θ_{n+1} is the new set of weights and biases. This is different from θ_i before, which is a particular weight/bias.

But what if we thought of the gradient as a force that pushes us? Recall from physics, θ is our position, then,

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = V \quad (\text{velocity}), \qquad \frac{\mathrm{d}v}{\mathrm{d}t} = A \quad (\text{acceleration}).$$

So we have these two differential equations, and we can solve them numerically using Euler's method:

$$\theta_{n+1} = \theta_n + \Delta t V_n \tag{6.1}$$

$$V_{n+1} = (1-r)V_n + \Delta t A_n \tag{6.2}$$

where r is the resistance from friction.

But we treat our error gradients as A, and integrate and gain velocity, V, and thus momentum. It's like our weights are dictated by our location in parameter space, and we move around weight space, accelerated by the error gradients. We build speed if we get a lot of acceleration in the same direction. We gain momentum.

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

$$V^{(\ell)} \leftarrow (1-r)V^{(\ell)} + \eta \nabla_{W^{(\ell)}} E.$$

Or, as is commonly used,

$$V^{(\ell)} \leftarrow \beta V^{(\ell)} + \nabla_{W^{(\ell)}} E.$$

Then, update out weights using

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

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

Check `https://ruder.io/optimizing-gradient-descent/` for the performance of different gradient descent optimization algorithms.