Notes on CS-489/CS-698: Neural Networks

Unversity of Waterloo

David Duan

Last Updated: May 2, 2021 (Incomplete V1.0)

Contents

1 Topic 1. **1**
- 1 The Hodgkin-Huxley Neuron Model . . . 2
- 2 Leaky Integrate-and-Fire Model . . . 6
- 3 Synapses . . . 10

2 Foundation of Learning **16**
- 4 Neural Learning . . . 16
- 5 Universal Approximation Theorem . . . 17
- 6 Loss Functions . . . 19
- 7 Gradient Descent . . . 21
- 8 Error Backpropagation . . . 22
- 9 Auto-Differentiation . . . 26
- 10 Neural networks with AutoDiff . . . 29
- 11 Concrete Example . . . 31
- 12 Concrete Example: Backpropagation . . . 33
- 13 Overfitting . . . 36
- 14 Enhancing Optimization . . . 38

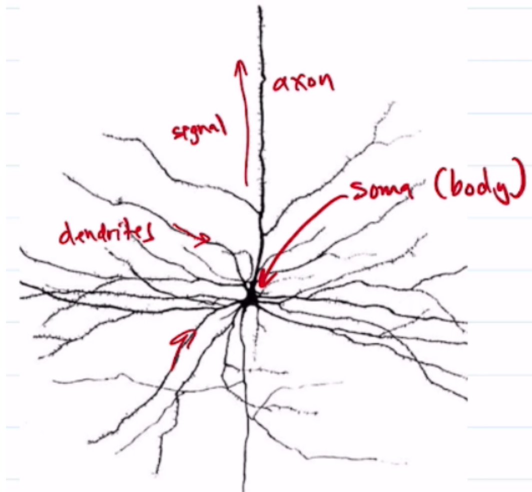
3 PyTorch **40**
- 15 Introduction to PyTorch . . . 40

Chapter 1. Topic 1.

Section 1. The Hodgkin-Huxley Neuron Model

1.1 Neurons

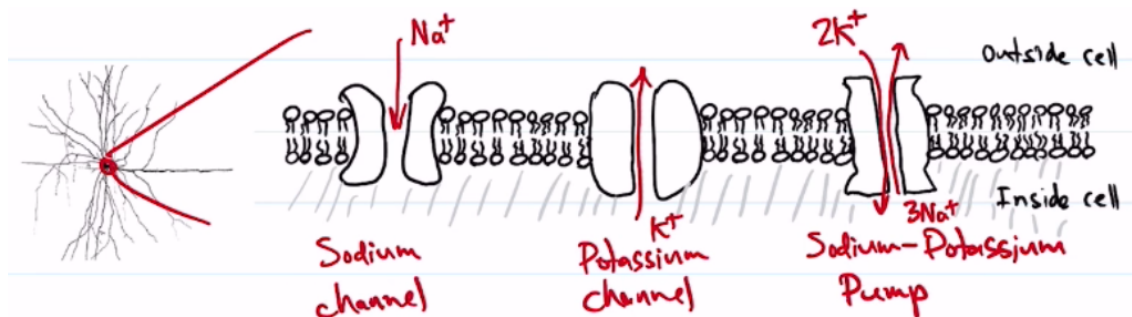
A **neuron** is a special cell that can send and receive signals from other neurons.



- **Soma**: generate electrical signals.
- **Axon**: transmit electrical signals.
- **Dendrites**: receive electrical signals.
- **Synapses**: send electrical signals.

1.2 Neuron Membrane Potential

Ions are molecules or atoms in which the number of electrons (-) does not match the number of protons (+), resulting in a net charge. Many ions float around your cells. The cell's **membrane**, a lipid bi-layer, stops most ions from crossing. However, ion channels embedded in the cell membrane allow ions to pass. There exist **sodium** and **potassium channels** which permits Na^+ and K^+ ions to move across the cell membrane, respectively.



The Na^+ channel moves Na^+ ions into the cell while the K^+ channel moves K^+ ions out of the cell. The **sodium-potassium pump** exchanges 3 Na^+ inside the cell for 2 K^+ ions outside the cell. This causes a higher concentration of Na^+ outside the cell and a higher concentration of K^+ inside the cell. It also creates a net positive charge outside and a net negative charge inside the cell. This difference in charge across the membrane induces a voltage difference and is called the **membrane potential**.

1.3 Action Potential

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

1.4 The Hodgkin-Huxley Model

The **Hodgkin-Huxley models** describes how action potentials in neurons are initiated and propagated. Their model is based on the non-linear interaction between membrane potential (aka **voltage**) and the opening/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 denote the membrane potential. A neuron usually keeps a membrane potential of around -70mV. We now wish to model the opening/closing of the channels.

Potassium Channels

The fraction of K^+ channels that are open is $n^4(t)$,[1] where

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

Here n is the dynamic variable and $n_\infty(V)$ is the equilibrium solution constant. Both $\tau_n(V)$ and $n_\infty(V)$ depend on voltage. Thus, the dynamics of the K^+ channel depends on the voltage and varies over time. As a remark, the DE converges to level $n_\infty(V)$; the rate of convergence is inversely proportional to τ, i.e., it converges faster if τ is smaller.

Sodium Channels

The fraction of Na^+ ion channels open is $(m(t))^3h(t)$, [2] where

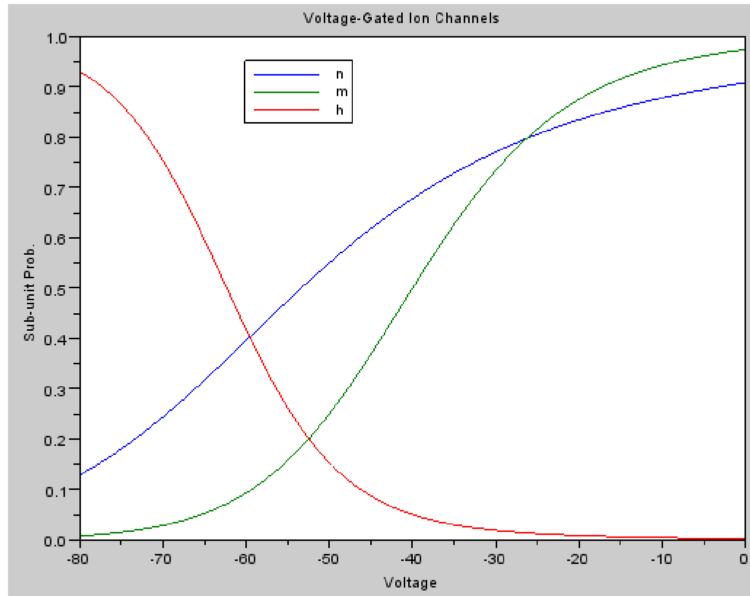
$$\boxed{\begin{aligned} \frac{dm}{dt} &= \frac{1}{\tau_m(V)}(m_\infty(v) - m) \\ \frac{dh}{dt} &= \frac{1}{\tau_h(V)}(h_\infty(v) - h) \end{aligned}}$$

Note all quantities like τ_m, τ_h, τ_n, etc., are measured empirically.

[1] The intuition is that each K^+ channel is controlled by four gates wherein the probability of one gate being open is n, hence the probability of all gates being open is n^4.

[2] Similar to above, we can interpret this as the Na^+ channel is controlled by three gates with probability m being open and one gate with probability h being open.

Below is a graph showing how $h(V), m(V), n(V)$ change as functions of voltage. As we can see, as voltage increases (move rightward) the n-gates and m-gates tend to open while the h-gate tend to close. To see how the DEs work, fix membrane potential at $V = -40$. Then we have $m(-40) \approx 0.5$ and $h(-40) \approx 0.05$. With this, you can compute the number (fraction) of sodium channels that are open as $(m(t))^3 h(t)$.



Channels and Membrane Potential

Now these two types of channels allow ions to flow into and out of the cell, inducing a current, which affects the membrane potential V. We can thus describe the membrane potential as a DE in terms of the fraction of K^+ and Na^+ channels that are open:

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

- C: **capacitance**.
- $\frac{dV}{dt}$: time 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**.
- $g_L, g_{\text{Na}}, g_{\text{K}}$: **maximum conductance**.
- $g_L(V - V_L)$: **leak current**.
- $g_{\text{Na}}m^3h(V - V_{\text{Na}})$: **sodium current**.
- $g_{\text{Na}}N^4(V - V_{\text{K}})$: **potassium current**.

This system of four DEs governs the dynamics of the membrane potential.

1.5 Python Simulations

Remark. The following two graphs come from my friend Sibelius's notes.

The following two graphs, where x-axis is time, can hopefully give you more intuition.

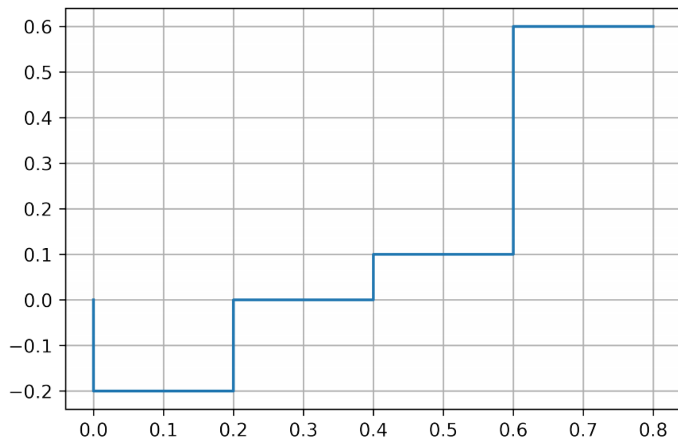


Figure 1.1: Amount of current.

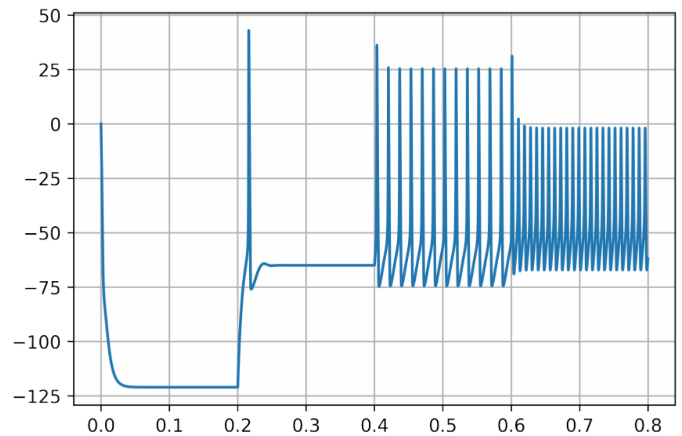


Figure 1.2: How neuron behaves.

The initial membrane potential is around -120 (right) and goes up as we increase the input current. At $J = 0.1$, it's high enough that causes regular action potentials. As we increase input current even more (to $J = 0.6$), the neuron spikes faster.

1.6 Summary and Motivation for LIF

The HH model is already greatly simplified:

- A neuron is treated as a point in space.
- Conductances are approximated with formulas.
- Only considers $\mathrm{K}^+, \mathrm{Na}^+$, and generic leak currents.

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.

Section 2. Leaky Integrate-and-Fire Model

2.1 The Leaky Integrate-and-Fire Model

The **leak 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). We express it as

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

- C: capacitance.
- g_L: conductance.
- J_{in}: input current.

Note that $g_L = 1/R$ where R is the resistance. Multiply both sides by R, we get

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

Let $\tau_m := RC$ be the time constant and define $V_{in} = RJ_{in}$ (Ohm's Law). Then the voltage can be modelled as

$$\tau_m\frac{dV}{dt} = V_{in} - (V - V_L).$$

Note this model is valid only when $V < V_{th}$ (later). 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. With this change of variable, we get

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

We integrate the DE for a given input current (or voltage) until v reaches the threshold of 1. Then we record a spike at that time. After it spikes, it remains dormant during its refractory period (denoted by τ_{ref}, often just a few ms). Then we start integrating again from zero (more on this below).

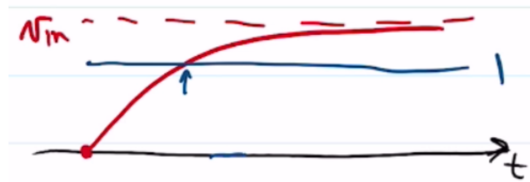
2.2 LIF Firing Rate

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

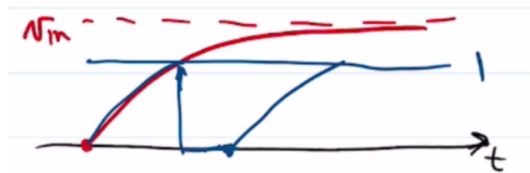
> **1.1. Lemma:** $v(t) := v_{in}(1 - e^{-t/\tau})$ *is a solution for*
> $$\tau \frac{dv}{dt} = v_{in} - v$$
> *with initial value* $v(0) = 0$.

In words, starting at a voltage of 0, follow the dynamics described by the DE, you get $v(t)$ as specified. To prove the claim, plug in the solution to the DE and show that LHS = RHS.

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

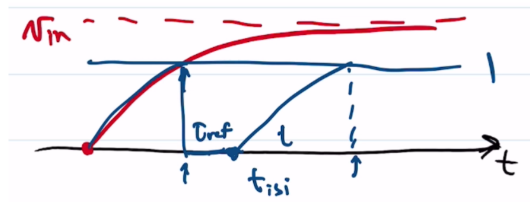


Note v_{in} must be above 1 or the neuron never spikes. The blue arrow points to the intersection of $v(t)$ and 1; at this time the neuron spikes. It then enters the refractory period and reinitiate the curve:



Solving for the Firing Rate

To solve for the firing rate, we need to solve for the time the spike occurs (as a function of v_{in}). Let t_{isi} denote the **inter-spike interval**. Note this value is the reciprocal of the fire rate. Now t_{isi} has two components, the refractory time constant τ_{ref}, plus the time it takes to go from $v = 0$ to $v = 1$, call it t^*.



We need to find t^* where $v(t^*) = 1$. From our above solution,

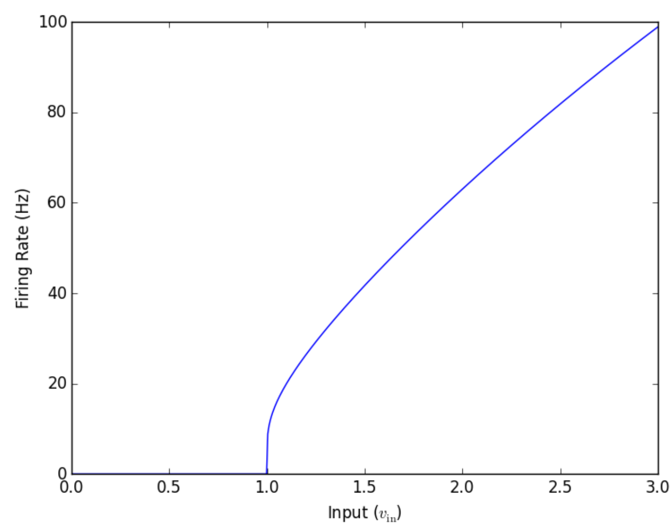
$$v(t^*) = 1 = v_{in}(1 - e^{-t/\tau}) \implies t^* = -\tau \ln\left(1 - \frac{1}{v_{in}}\right), \qquad v_{in} > 1.$$

Therefore, the firing rate is given by

$$\boxed{G(v_{in}) = \begin{cases} \dfrac{1}{\tau_{\text{ref}} - \tau_n \ln(1 - 1/v_{in})} & v_{in} > 1 \\ 0 & v_{in} \le 1. \end{cases}}$$

Cortical Neurons

Typical values for *cortical neurons* are $\tau_{ref} = 0.002$s or 2ms and $\tau_m = 0.02$s or 20ms. Below is the graph of the firing rate as a function of v_{in}.



Let us now look at even simpler neurons.

2.3 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 behavior can be represented by a number of different **activation functions**.[3]

Single Neuron Activation Functions

- Logistic

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

- Arctan

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

- Hyperbolic tangent

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

- Threshold

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

- Rectified Linear Unit (ReLU)

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

Multi-Neuron Activation Functions

Some activation functions depend on multiple neurons. Here are two examples.

SoftMax is like a probability distribution (or probability vector) as its elements add to 1. Given input $\vec{z} = (z_1, z_2, \ldots, z_N)$,

$$\text{SoftMax}(\vec{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

One-Hot is the extreme of the SoftMax, where only the largest element remains non-zero while the others are set to 0.

$$\text{One-Hot}(\vec{z})_i = \begin{cases} 1 & z_i = \max(\vec{z}) \\ 0 & \text{otherwise} \end{cases}$$

[3] Page 10 of Sibelius's notes has nice graphs for these functions.

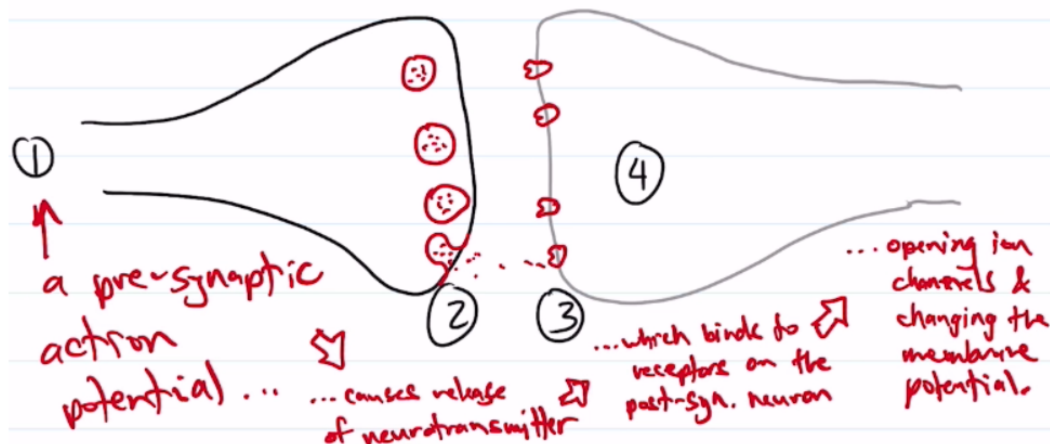
Section 3. Synapses

So far, we've just looked at individual neurons and how they react to their input. But that input usually comes from other neurons. In this lecture, we look at how neurons pass information between them and how we can model these communication channels.

3.1 Synapses

The action potential (wave of electrical activity) fired by a neuron travels along its **axon**. The junction where one neuron communicates with the next neuron is called a **synapse**.

A **pre-synaptic** action potential causes the release of **neurotransmitters** into adjacent synapses which bind to receptors on the **post-synaptic** neuron. This in turn opens or closes ion channels in the post-synaptic neuron, thereby changing membrane potential and causing the action potential to propagate.



3.2 Post-Synaptic Potential Filter

Even though an action potential is very fast, the synaptic processes by which it affects the next neuron takes time. Some synapses are fast (∼10ms) while some are slow (∼300ms). If we represent that time constant using τ_s, then the current entering the post-synaptic neuron can be written as a function of t:

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

where k is chosen so that

$$\int_0^\infty h(t)\,dt = 1.$$

Solving for k, we get

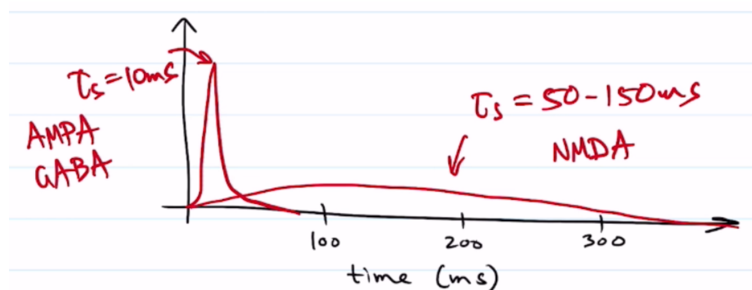
$$k = \frac{1}{n!\tau_s^{n+1}}.$$

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

Note we have a split at zero because the spike arrives at the synapse at time $t = 0$ and then we are looking at what's happening after that.

Effect of τ_s

Some neurotransmitters are fast (e.g., AMPA) while others are slow (e.g., NMDA). The area under these curves are 1 (by construction).



3.3 Spike Train

The Dirac delta function is defined as

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

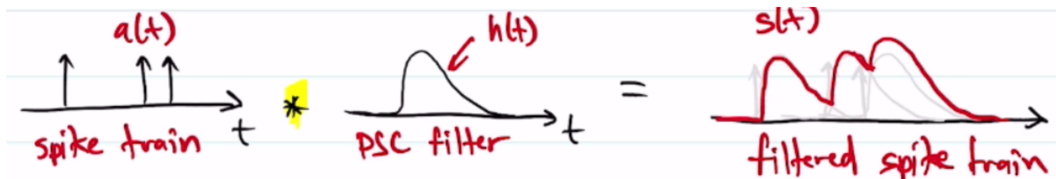
with the following two properties:

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

Multiple spikes form what we call a **spike train** and can be modelled as a sum of Dirac delta functions. Suppose we have a set of spikes indexed by p occurring at time t_p, then we can describe the activity of a neuron based on its spikes:

$$a(t) = \sum_p \delta(t - t_p).$$

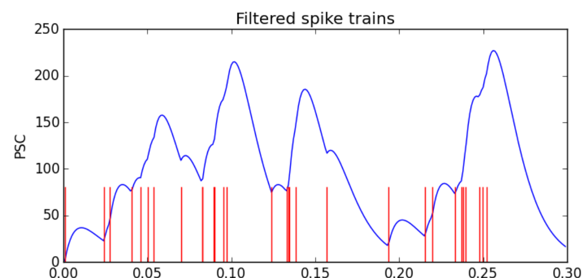
How does a spike train influence the post-synaptic neuron? You simply add together all the PSC filters, one for each spike. This is actually equivalent to **convolving** ($*$ operator) the spike train with the PSC filter.



That is,

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

Here's a real demo picture (with horizontal axis = time), where red lines are spikes and blue curve is the PSC induced by this spike train. Observe that (toward the right end) multiple spikes together cause a large increase in PSC.

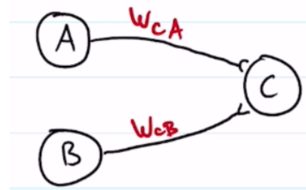


3.4 Connection Weight

How much does a spike at the pre-synaptic neuron influence the input current for the post-synaptic neuron? The total current induced by an action potential onto a particular post-synaptic neuron can vary widely, depending on:

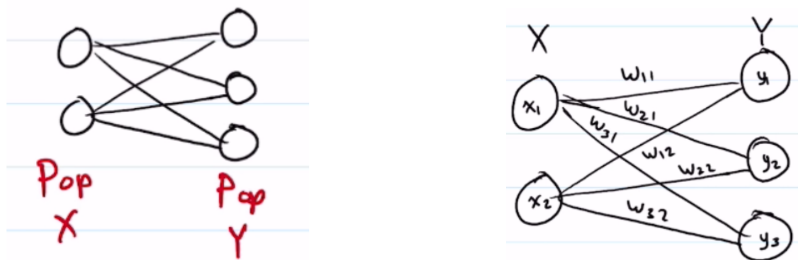
- the number and sizes of the synapses;
- the amount and type of neurotransmitter;
- the number and type of receptors;
- etc.

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 filter spike trains. In the following graph with three neurons A, B, C, we use w_{CA} and w_{CB} to denote the connection weight from A to C and from B to C, respectively.



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



The corresponding weight matrix is given by

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

Computing Input Current

Storing the neuron activities in vectors, i.e.,

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

we can compute the input to the nodes in Y using

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

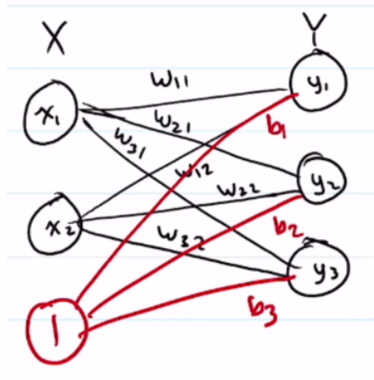
where $\vec{b}$ holds the biases for nodes (neurons) in Y. Thus,

$$\boxed{\vec{y} = \sigma(\vec{z})}$$

where $\sigma(\cdot)$ is an activation function.

Alternating Representation for Bias

Another way to represent the biases $\vec{b}$ is to add an auxiliary node in the input layer X with constant value 1.



With this modification, we can write the same thing but with an augmented matrix:

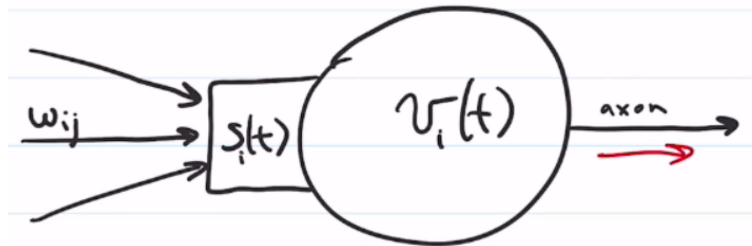
$$W\vec{x} + \vec{b} = [W \mid \vec{b}] \begin{bmatrix} \vec{x} \\ \hline 1 \end{bmatrix} =: \hat{W} \begin{bmatrix} \vec{x} \\ \hline 1 \end{bmatrix}$$

Implementing Connections Between Spiking Neurons

For simplicity, let $n = 0$ so $h(t) = \dfrac{1}{\tau_s} e^{-t/\tau_s}$. This happens to be the solution of the IVP

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

3.5 Full LIF Neuron Model



Relevant differential equations:

$$\begin{cases} \tau_m \dfrac{dv_i}{dt} = s_i - v_i & \text{if not refracting, i.e., integrating input current right now} \\ \tau_s \dfrac{ds_i}{dt} = -s_i \end{cases}$$

Sending Signal

If v_i reaches 1 (the threshold v_{th}),

1. start refractory period,
2. send spike along axon,
3. reset v to 0.

Receiving Signal

If a spike arrives from neuron j, increase s_i with

$$s_i \leftarrow s_i + \frac{W_{ij}}{\tau_s}.$$

In words, the amount of current that it injects into the post-synaptic neuron is proportional to the weight, and we divide it by τ_s (the normalizing factor) so the total amount of current that eventually gets injected is its weight.

Chapter 2. Foundation of Learning

Section 4. Neural Learning

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

There are three basic categories of learning problems:

1. In **supervised learning**, the desired output is known so we can compute the error and use that error to adjust our network.
2. In **unsupervised learning**, the output is unknown (or not supplied), so it 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.
3. In **reinforcement learning**, feedback is given, but usually less often and the error signal is usually less specific.

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

Supervised Learning

Our NN performs some mapping from an input space to an output space. We are given training data (input-target pairs), which is (presumably) the result of some consistent mapping process. Our task is to alter the connection weights in our network so that the network mimics this mapping.

Our goal is to bring the output as close as possible to the target. For now, we will use the scalar function $L(y, t)$ as an error/loss function, which returns a smaller value as our outputs are closer to the target.

There are two common types of mappings encountered in supervised learning:

- In **regression** problems, the outputs can take on a range of values.
- In **classification** problems, the outputs fall into a number of distinct categories.

Optimization

Once we have a cost function, our NN learning problem can be formulated as an optimization problem. Let our network be represented by the mapping f such that $y = f(x; \theta)$, where θ represents all the weights and biases. Neural learning seeks

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

In other words, find the weights are biases that minimize the expected cost (or error, or loss) between the outputs $f(x; \theta)$ and the targets $t(x)$.

Section 5. Universal Approximation Theorem

Can we approximate *any* function using a NN? That is, given a function $f(x)$, can we find the weights w_j, α_j and biases θ_j for $j = 1, \ldots, N$ such that

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

to arbitrary precision? The answer is yes.

2.1. Definition: A function σ is **sigmoidal** if $\sigma(x) = \begin{cases} 1 & x \to \infty \\ 0 & x \to -\infty. \end{cases}$

2.2. Theorem: *Let σ be any continuous sigmoidal function. Finite sums of the form*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j x + \theta_j)$$

are dense in $C(I_n)$, the set of continuous functions on $I_n = [0,1]^n$. In other words, given any $f \in C(I_n)$ are $\varepsilon > 0$, there is a sum, $G(x)$, of the above form, for which

$$|G(x) - f(x)| < \varepsilon \qquad \text{for all } x \in I_n.$$

Proof. For simplicity, let $w_j \to \infty$ for $j = 1, \ldots, N$. Then

$$\sigma(w_j x) \overset{w_j \to \infty}{\longrightarrow} \begin{cases} 0 & x \leq 0 \\ 1 & x > 0. \end{cases}$$

Shifting the curve by b_j (i.e., the jump occurs at $x = b_j$), we get

$$\sigma(w_j(x - b_j)) \to \begin{cases} 0 & x \leq 0 \\ 1 & x > 0. \end{cases}$$

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

$$H(x; b) = \lim_{w \to \infty} \sigma(w(x - b)).$$

We can use two such functions to create a "piece":

$$P(x; b, \delta) = H(x; b) - H(x; b + \delta).$$

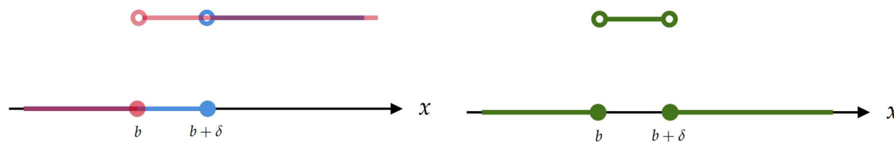


Figure 2.1: Credit: Sibelius.

Note this P is constructed with sigmoidal functions. Since $f(x)$ is continuous,

$$\forall a \in I_n : \lim_{x \to a} f(x) = f(a).$$

Therefore, there exists an interval $(a_j, a_j + \delta x)$ such that

$$\forall x \in (a_j, a_j + \Delta x) : |f(x) - f(a_j)| < \varepsilon.$$

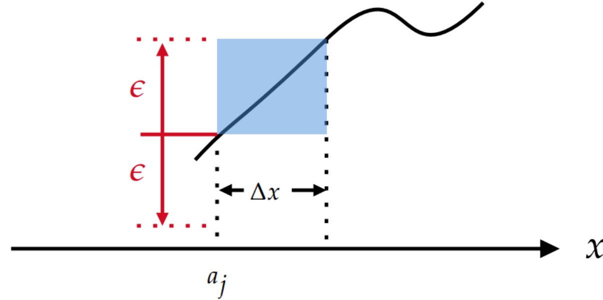


Figure 2.2: Credit: Sibelius.

Choose $b_j = a_j$ and $\delta_j = \Delta x$, and $\alpha_j = f(a_j)$. Then for all $a_j \leq x \leq a_j + \delta_j$, $|f(x) - f(a_j)| < \varepsilon$. Putting our piece in, we see that $|f(x) - \alpha_j P(x; b_j, \delta_j)| < \varepsilon$.

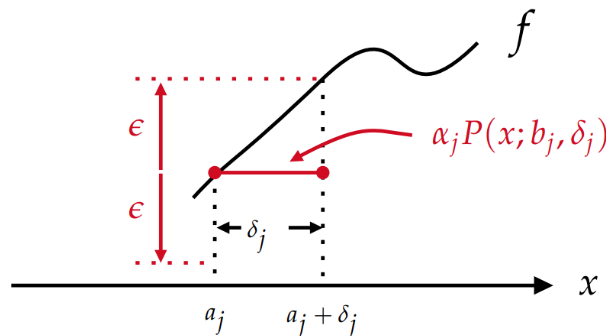


Figure 2.3: Credit: Sibelius.

In particular, the max error in $[b_j, b_j + \delta_j]$ is less than ε.

Repeat this process for $x = a_{j+1} = b_j + \delta_j$. Construct

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

Again, the choice of δ's depends on the ε's given. □

This theorem shows that with one layer of NN, you can approximate any function to any arbitrary precision. So why would we ever need a NN with more than 1 hidden layer? The short answer is that the theorem makes no guarantees about N, the number of hidden neurons. In fact, N might grow exponentially as ε gets smaller.

Section 6. Loss Functions

We need to choose a way to quantify how close our output is to the target. For this, we use a **cost function** (or **objective function**, or **loss function**, or **error function**). There are many choices, but here are three commonly-used ones.

Suppose we are given a dataset $\{x_i, t_i\}_{i=1,\dots,N}$. For input x_i, the NN outputs $y_i = f(x_i; \theta)$.

Mean Squared Error (MSE)

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

Taking the expectation 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 ReLU. This loss-function/activation-function pair is often used for regression problems.

Cross Entropy

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

$$y = f(x; \theta) \in (0, 1).$$

If we view y as the probability that $x = 1$, i.e.,

$$y = \Pr(x = 1 \mid \theta) = f(x; \theta),$$

then we can treat it as a Bernoulli distribution:

$$\begin{matrix} \Pr(x = 1 \mid \theta) = y \\ \Pr(x = 0 \mid \theta) = 1 - y \end{matrix} \implies \Pr(x = t \mid \theta) = y^t (1 - y)^{1-t}.$$

The task of learning would be finding a model (θ) that maximizes this likelihood. Equivalently, we are minimizing the negative log-likelihood:

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

This log-likelihood formula is the basis of the cross-entropy loss function. The expected cross entropy over the entire dataset is

$$E = -\mathbb{E}[t_i \log y_i + (1 - t_i)\log(1 - y_i)] = -\frac{1}{N}\sum_{i=1}^{N} t_i \ln y_i + (1 - t_i)\ln(1 - y_i).$$

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

Categorical Cross Entropy

Consider a classification problem that has $K > 2$ class. Given an input, the task of our model is to output the class of the input.

Given input x, suppose our model outputs $y = f(x; \theta) \in [0, 1]^k$ with

$$\sum_{k=1}^{Y} y_k = 1.$$

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.

Under that distribution, suppose we observe a sample from class i. The likelihood of that observation is

$$\Pr(x \in C_i \mid \theta) = y_i$$

where C_i contains the inputs of class i. Note that y is a function of the input x and the model parameters θ. If we represent the target class using an one-hot vector, we get $t_i = 1$ and $t_j = 0$ for other j. Then we can write the likelihood as

$$\Pr(x \in C_i \mid \theta) = \prod_{k=1}^{K} y_k^{t_k}.$$

Thus, the negative log-likelihood of x is

$$-\log \Pr(x \in C_i \mid \theta) = -\sum_{k=1}^{K} t_k \log y_k.$$

This loss function is known as the **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

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

Since $\sum_k y_k = 1$, categorical cross-entropy works well with SoftMax.

Section 7. Gradient Descent

Recall the operation of our vector can be written as

$$y = f(x; \theta)$$

- x: input data.
- θ: connection weights and biases.
- y: output by our network.

Denote the target by t and loss function by $L(y, t)$, we can view neural learning as the optimization problem

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

In words, we are looking for the set of parameters (weights and biases) θ that minimizes the expected value of the loss function, where the loss is calculated based on the targets $t(x)$ and our output $f(x; \theta)$. We can apply **gradient descent** to E using the gradient

$$\nabla_{\theta} E = \left[\frac{\partial E}{\partial \theta_0}, \frac{\partial E}{\partial \theta_1}, \cdots, \frac{\partial E}{\partial \theta_p}\right].$$

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, $E(a, b)$, and you wish to find a^* and b^* to maximize E:

$$(a^*, b^*) = \arg\max_{a,b} E(a, b).$$

The gradient of E points in the direction of steepest ascent:

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

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

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

where k is your **step multiplier** (more on this later).

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

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

Note 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.

Section 8. Error Backpropagation

2.3. We can apply gradient descent on a multi-layer network, using the **chain rule** from Calculus to calculate the gradients of the error wrt deeper connection weights and biases.

2.4. Example: Let (α_i, h_i) be the input and output of the hidden node i and (β_j, y_j) denote the input and output of the output node j. Denote the loss function by $E(y, t)$.

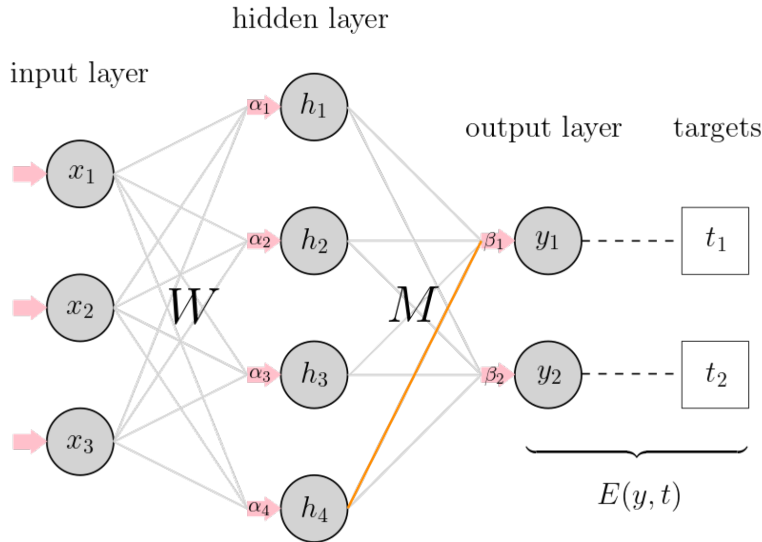


Figure 2.4: A toy network.

Suppose we want to compute $\frac{\partial E}{\partial M_{41}}$, the gradient of error wrt to M_{41}, which corresponds to the connection weight of edge $h_4 \to y_1$. This can help us adjust the parameter M_{41}.

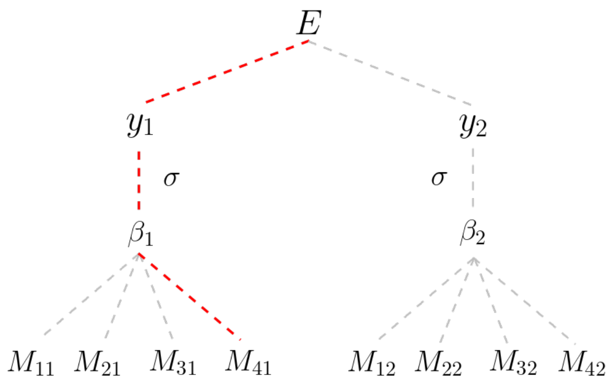


Figure 2.5: Dependency graph.

Look at the dependency graph, we see that

$$E(y,t) = E(\overbrace{\sigma(\underbrace{hM + b}_{\beta_1})}^{y}, t)$$

So by the chain rule,

$$\frac{\partial E}{\partial M_{41}} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial \beta_1}\frac{\partial \beta_1}{\partial M_{41}}.$$

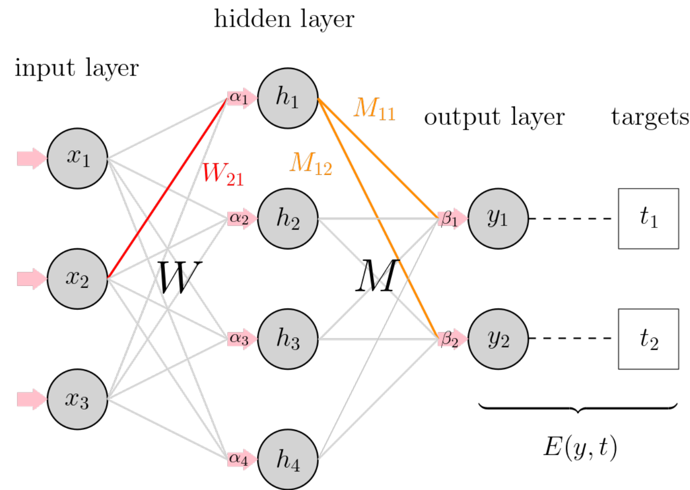
Recall $\beta_1 = \sum_{i=1}^{4} h_i M_{i1} + b_1$. Then

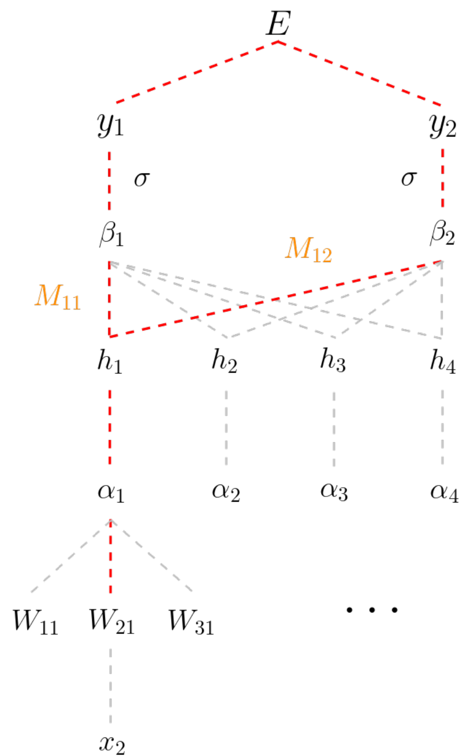
$$\frac{\partial \beta_1}{\partial M_{41}} = h_4.$$

Therefore,

$$\frac{\partial E}{\partial M_{41}} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial \beta_1} h_4.$$

2.5. Example: Let's go one layer deeper: suppose we'd like to find $\frac{\partial E}{\partial W_{21}}$, the gradient of error with respect to the connection weights of edge $x_2 \to h_1$.





Let's take α_1 as the intermediary:

$$\frac{\partial E}{\partial W_{21}} = \frac{\partial E}{\partial \alpha_1}\frac{\partial \alpha_1}{\partial W_{21}}$$

The second part is easy:

$$\alpha_1 = \sum_{j=1}^{3} x_j W_{j1} + a_1 \implies \frac{\partial \alpha_1}{\partial W_{21}} = x_2.$$

For the first part, we use h_1 as the intermediary. Note there are two paths from E to h_1, so we need to take the derivative through **both** β_1 and β_2 (line 2):

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

Finally, we've learned $\frac{\partial E}{\partial \beta_i}$ already when we were learning gradients for M (in the previous layer), so we can write

$$\begin{aligned}\frac{\partial E}{\partial \alpha_1} &= \frac{\partial h_1}{\partial \alpha_1}\begin{bmatrix}\frac{\partial E}{\partial \beta_1} & \frac{\partial E}{\partial \beta_2}\end{bmatrix} \cdot \begin{bmatrix}M_{11} & M_{12}\end{bmatrix}\\ &= \frac{\partial h_1}{\partial \alpha_1}\begin{bmatrix}\frac{\partial E}{\partial \beta_1} & \frac{\partial E}{\partial \beta_2}\end{bmatrix}\begin{bmatrix}M_{11}\\ M_{12}\end{bmatrix}.\end{aligned}$$

2.6. Note: More generally, for $x \in \mathbb{R}^X$, $h \in \mathbb{R}^H$, and $y, t \in \mathbb{R}^Y$. Denote the connection weights from the hidden layer to the output layer by $M \in \mathbb{R}^{H \times Y}$.

$$\begin{aligned}
\frac{\partial E}{\partial \alpha_i} &= \frac{\partial h_i}{\partial \alpha_i}\frac{\partial E}{\partial h_i} \\
&= \frac{\partial h_i}{\partial \alpha_i}\left(\frac{\partial E}{\partial \beta_1}\frac{\partial \beta_1}{\partial h_i} + \frac{\partial E}{\partial \beta_2}\frac{\partial \beta_2}{\partial h_i} + \cdots + \frac{\partial E}{\partial \beta_Y}\frac{\partial \beta_Y}{\partial h_i}\right) \\
&= \frac{\partial h_i}{\alpha_i}\left(\frac{\partial E}{\partial \beta_1}M_{i1} + \frac{\partial E}{\partial \beta_2}M_{i2} + \cdots + \frac{\partial E}{\partial \beta_Y}M_{iY}\right) \\
&= \frac{\partial h_i}{\partial \alpha_i}\begin{bmatrix}\frac{\partial E}{\partial \beta_1} & \cdots & \frac{\partial E}{\partial \beta_Y}\end{bmatrix} \cdot \begin{bmatrix}M_{i1} & \cdots & M_{iY}\end{bmatrix} && i\text{th row of } M \\
&= \frac{\partial h_i}{\partial \alpha_i}\begin{bmatrix}\frac{\partial E}{\partial \beta_1} & \cdots & \frac{\partial E}{\partial \beta_Y}\end{bmatrix}\begin{bmatrix}M_{i1} \\ \vdots \\ M_{iY}\end{bmatrix} && i\text{th col of } M^T
\end{aligned}$$

Now for all elements $\alpha_1, \ldots, \alpha_H$, we have

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

where $\odot$ denotes the Hadamard (element-wise) product. Even more compactly:

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

2.7. Note: We are ready for the most general version of backprop. Let us use superscript to indicate the layer information, i.e.,

- let $h^{(\ell)}$ denote the (outputs of) nodes in layer ℓ;
- let $z^{(\ell)}$ denote the input current to nodes in layer ℓ;
- let $W^{(\ell+1)}$ denote the connection weights between layer ℓ and layer $\ell + 1$;
- let $b^{(\ell+1)}$ denote the biases of (into, see blow) layer ℓ;
- let σ denote the activation function E denote the error function.

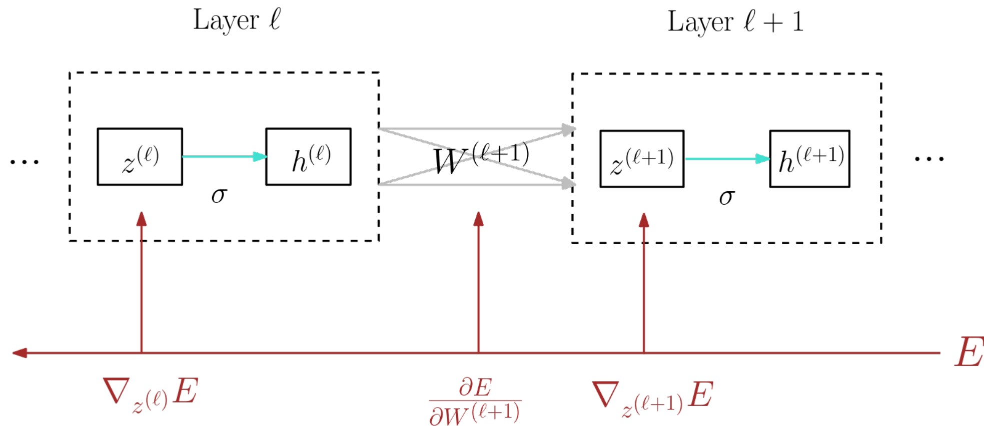
Suppose we have computed

$$\nabla_{z^{(\ell+1)}}E = \frac{\partial E}{\partial z^{(\ell+1)}}$$

in the previous iteration. We have (make sure you understand what this is saying!)

$$h^{(\ell+1)} = \sigma(z^{(\ell+1)}) = \sigma(h^{(\ell)}W^{(\ell+1)} + b^{(\ell+1)}).$$

In words, the output of layer $\ell + 1$, $h^{(\ell+1)}$, is obtained by applying the activation function σ to the input current $z^{(\ell+1)}$ of this layer, which is given by $h^{(\ell)}W^{(\ell+1)} + b^{(\ell+1)}$, i.e., the output from layer ℓ weighted by the connection weights $W^{(\ell+1)}$ and shifted by the bias $b^{(\ell+1)}$.



Apply what we derived above, we get

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

For the gradient of error wrt connection weights, recall that

$$z_j^{(\ell+1)} = h_j^{(\ell)} W_{ij}^{(\ell+1)} + b^{(\ell+1)} \implies \frac{\partial z_j^{(\ell+1)}}{\partial W_{ij}^{(\ell+1)}} = h_i^{(\ell)}.$$

Thus, by the chain rule, we have

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

Thus, $\frac{\partial E}{\partial W^{(\ell+1)}}$ is the outer product of $h^{(\ell)}$ and $\nabla_{z^{(\ell+1)}} E$:

$$\frac{\partial E}{\partial W^{(\ell+1)}} = \begin{bmatrix} h_1^{(\ell)} \\ \vdots \\ h_i^{(\ell)} \\ \vdots \\ h_H^{(\ell)} \end{bmatrix} \begin{bmatrix} \nabla_{z_1^{(\ell+1)}} E & \cdots & \nabla_{z_j^{(\ell+1)}} E & \cdots & \nabla_{z_Z^{(\ell+1)}} E \end{bmatrix} = \begin{bmatrix} \quad \text{Same size as } W^{(\ell+1)} \in \mathbb{R}^{H \times Z} \quad \end{bmatrix}$$

We summarize these results into the following theorem.

2.8. Theorem (Error Backpropagation)**:** *Suppose we have computed $\nabla_{z^{(\ell+1)}} E$. The gradient of error wrt the our connection weights $W^{(\ell+1)}$ is given by*

$$\boxed{\nabla_{W^{(\ell+1)}} E = [h^{(\ell)}]^T \nabla_{z^{(\ell+1)}} E.}$$

The gradient of error wrt the output of current layer, which is used for calculating the gradient of error wrt the connection weights of the next layer, is given by

$$\boxed{\nabla_{z^{(\ell)}} E = \sigma'(z^{(\ell)}) \odot (\nabla_{z^{(\ell+1)}} E \cdot (W^{(\ell+1)})^T).}$$

Section 9. Auto-Differentiation

Expression Graphs

A **computational** or **expression graph** is a digraph where each node represents an **operation** or a **variable**.

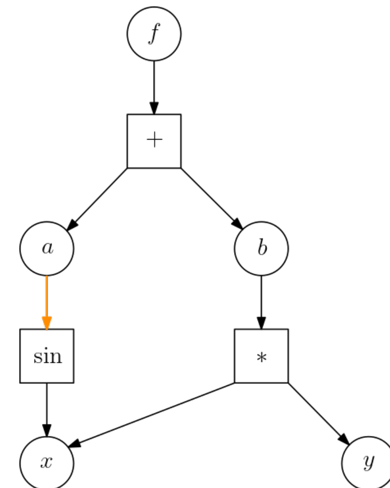
- Each variable has a *value* and a pointer to its *creator* (an operation).
- Each operation has pointers to its *arguments* (variables).

Consider the function

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

Starting from the bottom, we define

- $x = \text{Var}$; $y = \text{Var}$
- $a = \sin(x)$
- $b = x * y$
- $f = a + b$



Note the orange arrow from a to sin denotes the **creator reference**, as a is created from the sin function.

Constructing Computational Graphs

Consider $f = a * b$, where (variables) nodes a and b exist already. We can construct a computational graph for this expression following these steps:

0. (Given Var objects a and b).
1. Create the Op object for $*$.
2. Save references to the argument fields of Op $*$.
3. Create a variable for f, the output of this operation.
4. Set Op $*$ as the creator of Var f.

Evaluating Expression Graphs

We can evaluate the expression by calling its `evaluate` function.

Algorithm. Var`.evaluate`

```
1: if self.creator is None then
2:     return self.val
3: else
4:     return self.creator.evaluate()
```

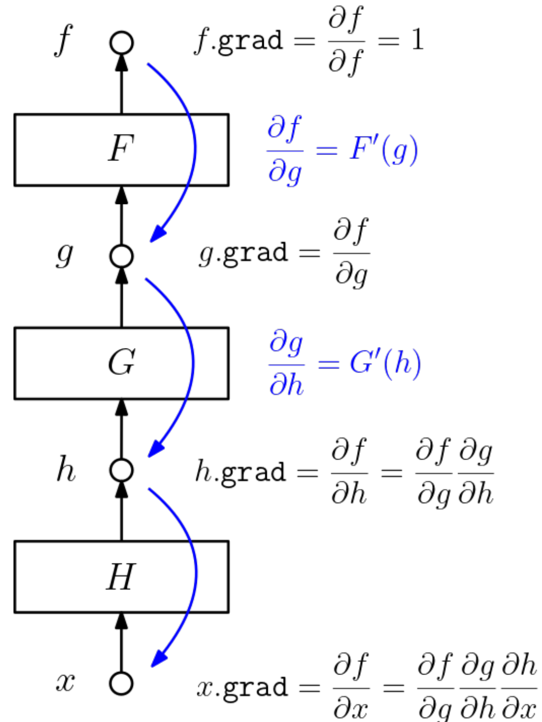
Algorithm. Op`.evaluate`

```
1: Evaluate each argument
2: Compute and return the value
```

Computing Derivatives

The expression graph can also be used to compute the derivatives. Each Var stores the derivative of the expression wrt itself in its field `self.grad`.

Consider $f = F(G(H(x)))$. Define $h = H(x)$ and $g = G(h)$. Then



Starting with a value of 1 at the top, work our way down through the graph and increment (multiplying, according to chain rule) grad of each Var as we go. Each Op contributes its factor according to chain rule and passes the updated derivative down the graph. More specifically, each object has a `backward(s)` method that processes the derivative s from above and passes it down the graph.

Algorithm. Var`.backward(s)`

input: s - Derivative inherited from above. Note that `self.val`, `self.grad`, and `s` must have the same shape.

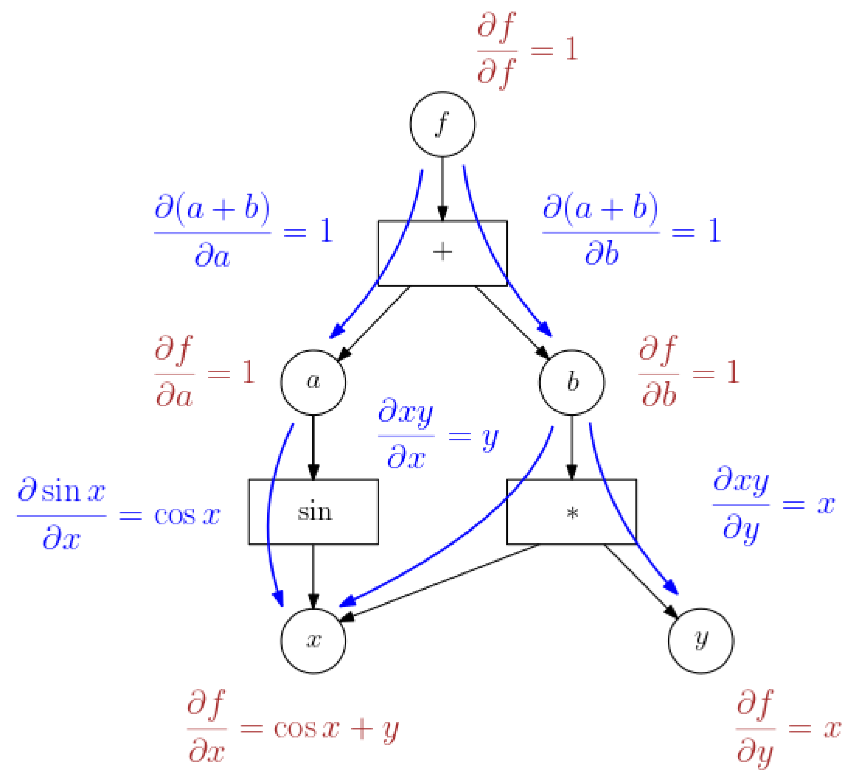
1: `self.grad += s` ▷ Update the derivative
2: `self.creator.backward(s)` ▷ Pass the derivative down

Algorithm. Op`.backward(s)`

input: s - Derivative inherited from above. Note that `s` must match the shape of the operation's output.

1: **for** x in `self.args` **do**
2: `x.backward(s` $\cdot \frac{\partial \text{Op}}{\partial x}$`)` ▷ Derivative of the operation wrt argument x

Here is a more detailed example:



Section 10. Neural networks with AutoDiff

Optimization using AutoDiff

Suppose we want to minimize some scalar function E wrt v. We can use gradient descent.

$$v = v - \kappa \nabla_v E(v).$$

Algorithm 1 Gradient Descent with AutoDiff

1: Initialize variable v and learning rate κ.
2: Construct an expression graph for E.
3: **while** not converged **do**
4: Evaluate E at v.
5: Set gradient $v.\texttt{grad} = \nabla_v E = 0$ to zero.
6: Propagation derivatives down (increment $v.\texttt{grad}$)
7: $v \leftarrow v - \kappa * v.\texttt{grad}$.

Neural Learning

We can use the same process to implement error backprop for NN. Our NN will be composed of a series of layers, each transforming the data from the layer below it, culminating in a scalar-valued cost function. The cost function takes the NN output as well as the targets and returns a scalar.

Consider the following simple network:

$$X \xrightarrow{a} z \xrightarrow{b} zW \xrightarrow{c} \sigma(zW) \xrightarrow{d} E(\sigma(zW), T).$$

Define the four common operations as follows. Given a dataset (X, T),

- a: identity I.
- b: multiply by W, i.e., $(\lambda z : zW)$,
- c: logistic σ.
- d: cost E.

The output from one layer becomes the input to the next layer. Each layer, including the cost function, is just a function in a nested mathematical expression. The cost function can thus be expressed as

$$E = d(c(b(a(x))), T)$$

and our update rule is given by

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

We construct our network using Var and Op objects so that we can take advantage of their `backward()` methods to compute the gradients.

Algorithm 2 Neural Learning with AutoDiff

```
1: Given dataset (X, T) and network model net with params θ, cost function cost.
2: In each iteration,
3: while not done do

   feed-forward
4:     y = net(x)                                    ▷ Feed x into our network
5:     loss = cost(y, T)

   backprop
6:     loss.zero_grad()
7:     loss.backward()

   gradient descent
8:     θ = θ - k θ.grad
```

Matrix Operations

To work with NN, our AD library will have to deal with matrix operations. ***For reference, any time you have $\partial L/\partial X$ where L is a scalar and X is a matrix or vector, $\partial L/\partial X$ must have the same shape as X.*** Let L be our cost function which depends on matrices A and B.

Matrix Addition. Let $A, B \in \mathbb{R}^{m\times n}$ and $y = A + B \in \mathbb{R}^{m\times n}$. Define $s := \nabla_y L$. Then the gradients of L wrt A and B are given by

$$\nabla_A L = \nabla_y L \odot \nabla_A y = s \odot \mathbf{1}_{m\times n} \in \mathbb{R}^{m\times n}$$
$$\nabla_B L = \nabla_y L \odot \nabla_B y = s \odot \mathbf{1}_{m\times n} \in \mathbb{R}^{m\times n}$$

Thus, in $+$.`backward(s)`, we would have

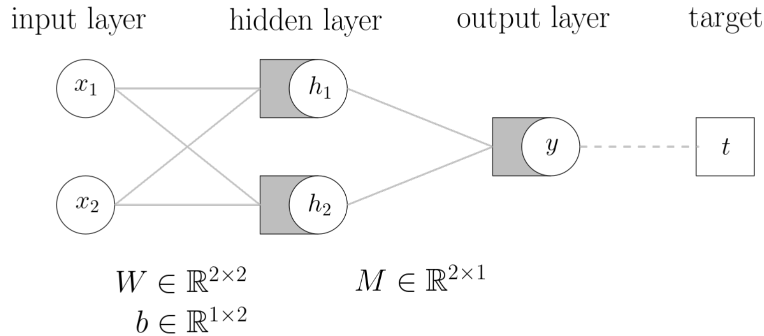
- A.`backward`$(s \odot \mathbf{1}_{m\times n})$.
- B.`backward`$(s \odot \mathbf{1}_{m\times n})$.

In words, the $+$ operation inherits a gradient s from above, then calls its arguments' `backward` function with updated gradients; nodes below A and B in the computational graph will inherit these new gradients.

Matrix Multiplication. Let $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times k}$, and $y = A * B \in \mathbb{R}^{m\times k}$ ($*$ denotes matrix multiplication). By following the rule of dimension, we get

$$\nabla_A L = \underbrace{\nabla_y L}_{m\times k} * \underbrace{\nabla_A y}_{k\times n} = s * B^T \in \mathbb{R}^{m\times n}$$
$$\nabla_B L = \underbrace{\nabla_B y}_{n\times m} * \underbrace{\nabla_y L}_{m\times k} = A^T * s \in \mathbb{R}^{n\times k}$$

Section 11. Concrete Example



Suppose we are given input data

$$x = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{bmatrix} \qquad \begin{matrix} \text{sample 1} \\ \text{sample 2} \\ \text{sample 3} \end{matrix}$$

- 3 samples;
- 2 features per sample;
- x_{ij} denotes the jth feature of the ith input.

Hidden Layer

Let W, b be the weight matrix and bias between the input layer and the hidden layer. Then the input of this layer is given by $xW + b$ and the output is given by

$$h = \sigma_1(xW + b),$$

where σ_1 denotes the activation function. The connection weights are given by

$$W = \begin{matrix} & h_1 & h_2 \\ x_1 & W_{11} & W_{12} \\ x_2 & W_{21} & W_{22} \end{matrix} \in \mathbb{R}^{2\times 2}.$$

- 2 inputs: x_1, x_2.
- 2 outputs: h_1, h_2.
- W_{ij} denotes the weight of the edge between x_i and h_j.

Note that $xW \in \mathbb{R}^{3\times 2}$, so we know that $b \in \mathbb{R}^{3\times 2}$. However, the biases are the same for each sample, so we only need to define one row and then broadcast it to the appropriate size:

$$b = \begin{bmatrix} b_{i1} & b_{i2} \end{bmatrix} \in \mathbb{R}^{1\times 2} \quad (\text{broadcast to } \mathbb{R}^{3\times 2}).$$

The output has dimension $h \in \mathbb{R}^{3\times 2}$. The first column of h corresponds to the output from the first node and the second column corresponds to the output from the second node. There are three row because we have one row of (h_1, h_2) for each of the three samples.

Output Layer

Let M denote the weight matrix between the hidden layer and the output layer and assume there is no bias term here. The input of this layer is hM and the output of this layer (and hence this network) is given by $y = \sigma(hM)$. Note in practice we might not need any activation function σ in the output layer. The connection weights are given by

$$M = \begin{matrix} & y \\ h_1 & \begin{bmatrix} M_1 \\ M_2 \end{bmatrix} \\ h_2 & \end{matrix} \in \mathbb{R}^{2\times 1}.$$

- 2 inputs: h_1, h_2.
- 1 output: y.
- M_i denotes the weight of the edge between h_i and y.

Since $h \in \mathbb{R}^{3\times 2}$ and $M \in \mathbb{R}^{2\times 1}$, the output y has dimension $y \in \mathbb{R}^{3\times 1}$ where y_i denotes the output of out network for the ith sample.

Concrete Implementation

Let us first initialize all the variables. Dimensions: $W \in \mathbb{R}^{2\times 2}, b \in \mathbb{R}^{1\times 2}, M \in \mathbb{R}^{2,1}$.

```
1  W = np.random.normal(scale=0.1, size=(2,2))  # weights
2  b = np.random.normal(scale=0.1, size=(1,2))  # biases
3  sigma1 = Logistic()                          # activation function
4  M = np.random.normal(scale=0.1, size=(2,1))  # weights
5  sigma2 = Logistic()                          # activation function
6  E = CrossEntropy()                           # cost function
7
8  x = ds.inputs()                              # dataset inputs, shape = (3, 2)
9  t = ds.targets()                             # dataset targets, shape = (3, 1)
```

Now the feedforward phase. Let $z^{(2)}$ denote the input current to y and $z^{(1)} = (z_1^{(1)}, z_2^{(1)})$ be the input current the hidden layer. Dimensions: $z^{(1)} \in \mathbb{R}^{3\times 2}, h \in \mathbb{R}^{3\times 2}, z^{(2)} \in \mathbb{R}^{3\times 2}, y \in \mathbb{R}^{3\times 1}$.

```
1  # From input layer to hidden layer
2  z1 = x @ W + a
3  h = sigma1(z1)  # Note here sigma1 stores dz1dh
4
5  # From hidden layer to output layer
6  z2 = h @ M
7  y = sigma2(z2)  # Note here sigma2 stores dz2dy
8
9  # Compute loss
10 loss = E(y, t)  # Note here E stores dEdy
```

Section 12. Concrete Example: Backpropagation

Error Backpropagation

Given network output $y \in \mathbb{R}^{3\times 1}$, we compare it with the true labels $t \in \mathbb{R}^{3\times 1}$. More precisely, we feed them to an error function $E(y, t)$, which quantifies the prediction error between y and t. We then compute each of

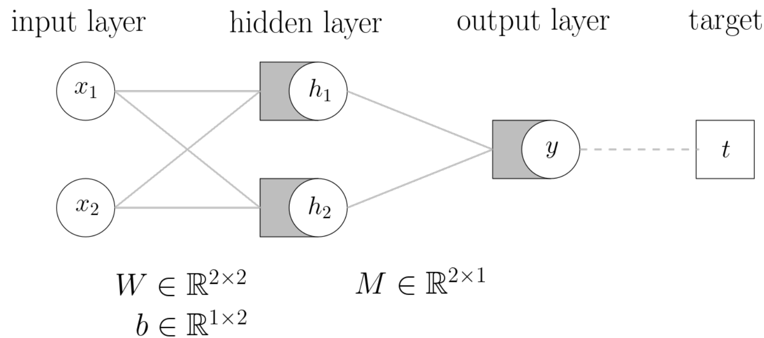
$$\frac{\partial E}{\partial M_i}, i \in \{1, 2\}, \qquad \frac{\partial E}{\partial W_{ij}}, i, j \in \{1, 2\}, \qquad \frac{\partial E}{\partial b_i}, i \in \{1, 2\}.$$

to adjust the connection weights and biases. Since the gradient points to the direction of steepest ascent, we want to move in the opposite direction of the gradient. Hence, we update the network parameters (simultaneously) by

$$\begin{aligned} W &= W - \alpha \cdot \frac{\partial E}{\partial W} \\ b &= b - \alpha \cdot \frac{\partial E}{\partial b} \\ M &= M - \alpha \cdot \frac{\partial E}{\partial M} \end{aligned}$$

where α denotes the learning rate. We are now ready for the next iteration of learning.

Previous Example



Dimension graph for feedforward phase (note sample size $m = 3$ in our example):

$$x \in \mathbb{R}^{m\times 2} \xrightarrow[b \in \mathbb{R}^{m\times 2},\ W \in \mathbb{R}^{2\times 2}]{z^{(1)} = xW + b} z^{(1)} \in \mathbb{R}^{m\times 2} \xrightarrow[\sigma^{(1)} : \mathbb{R}^{(p\times q)} \to \mathbb{R}^{(p\times q)}]{h = \sigma^{(1)}(z^{(1)})} h \in \mathbb{R}^{m\times 2} \xrightarrow[M \in \mathbb{R}^{2\times 1}]{z^{(2)} = hM} z^{(2)} \in \mathbb{R}^{m\times 1} \xrightarrow[\sigma^{(2)} : \mathbb{R}^{(p\times q)} \to \mathbb{R}^{(p\times q)}]{y = \sigma^{(2)}(z^{(2)})} y \in \mathbb{R}^{m\times 1}$$

More General Example

Consider the following `Layer` objects:

- A `Population` layer represents a layer of nodes; its fields and methods include
  - `self.z`: input.
  - `self.h`: output.
  - `self.act`: activation function.
  - `self.__call__(self, x=None)`: apply activation function if input is given:
    - if `x` is not `None`, set `self.z` to `x`
    - otherwise, `self.h = self.act(x)`
- A `Connection` layer represents a layer of all-to-all connections; its fields and methods include
  - `self.W` and `self.b`: weights and biases.
  - `self.__call__(self, x)`: apply the weights and biases to the input:
    - if `len(x) > 1`, `return x@self.W + np.outer(np.ones(P), self.b)`;
    - otherwise, `return x@self.W + self.b`.
- A `Dense` layer consists of two layers:
  - L1: a `Connection` layer of connection weights, and
  - L2: a `Population` layer, consisting of nodes that receives current from the `Connection` layer, and apply the activation function

Consider the following (sub)network:

Layer $\ell-1$, Population	Layer ℓ, Connection	Layer ℓ, Population	(Known)
$z^{(\ell-1)}, h^{(\ell-1)}, \sigma^{(\ell-1)}$	$W^{(\ell)}, b^{(\ell)}$	$z^{(\ell)}, h^{(\ell)}, \sigma^{(\ell)}$	$\frac{\partial E}{\partial h^{(\ell)}}$

Suppose we are interested in updating $W^{(\ell)}$ and $b^{(\ell)}$. Moreover, suppose we have computed $\frac{\partial E}{\partial h^{(\ell)}}$ already. Let `pre` denote Layer-$(\ell-1)$ and `post` Layer-ℓ. Backprop can be implemented as follows.

```
1  dEdz = post.L2.act.derivative() * dEdh
2
3  dEdW = pre.L2.h.T @ dEdz
4  dEdb = np.sum(dEdz, axis=0)
5
6  dEdh = dEdz @ post.L1.W.T
7
8  post.L1.W -= lrate * dEdW   # Update connection weights
9  post.L1.b -= lrate * dEdb   # Update connection weights
```

Recall the following result:

2.9. Theorem (Error Backpropagation)**:** *Suppose we have computed* $\nabla_{z^{(\ell+1)}} E$*. The gradient of error wrt the our connection weights* $W^{(\ell+1)}$ *is given by*

$$\boxed{\nabla_{W^{(\ell+1)}} E = [h^{(\ell)}]^T \nabla_{z^{(\ell+1)}} E.}$$

The gradient of error wrt the output of current layer, which is used for calculating the gradient of error wrt the connection weights of the next layer, is given by

$$\boxed{\nabla_{z^{(\ell)}} E = \sigma'(z^{(\ell)}) \odot (\nabla_{z^{(\ell+1)}} E \cdot (W^{(\ell+1)})^T).}$$

We now justify each line (except the last two, which are straightforward).

- `dEdz = post.L2.act.derivative() * dEdh`

$$\nabla_{z^{(\ell)}} E = \frac{\partial E}{\partial z^{(\ell)}} = \frac{\partial E}{h^{(\ell)}} \frac{\partial h^{(\ell)}}{\partial z^{(\ell)}} = \frac{\partial E}{h^{(\ell)}} \frac{\partial \sigma^{(\ell)}(z^{(\ell)})}{\partial z^{(\ell)}} = (\sigma^{(\ell)})'(z^{(\ell)}) \odot \frac{\partial E}{h^{(\ell)}}$$

- `dEdW = pre.L2.h.T @ dEdz`

$$\nabla_{W^{(\ell)}} E = [h^{(\ell-1)}]^T \nabla_{z^{(\ell)}} E.$$

- `dEdb = np.sum(dEdz, axis=0)`[1]

$$\nabla_{b^{(\ell)}} E = \mathbf{1}_{1\times m}^T \nabla_{z^{(\ell)}} E.$$

- `dEdh = dEdz @ post.L1.W.T`

$$\frac{\partial E}{\partial h^{(\ell-1)}} = (\nabla_{z^{(\ell)}} E \cdot (W^{(\ell)})^T).$$

It remains to justify our assumption, i.e., how to find the first $\frac{\partial E}{\partial h}$. Suppose we are at the last layer of the network, where the loss $E(y, t)$ has just been calculated. Note in this case, the output of the last layer, h, is exactly the output of the network, y. Thus, we can simply call `self.loss.derivative()` to find $\frac{\partial E}{\partial y}$. This concludes our procedure.

```
1  self.loss(y, t)                               # need to call this first
2  dEdh = self.loss.derivative()                 # compute the first dEdh
3  for i in range(len(self.lyr)-1, 0, -1):       # wrap in a for-loop
4      pre, post = self.lyr[i-1], self.lyr[i]    # layers
5      dEdz = post.L2.act.derivative() * dEdh    # dEdz
6      dEdW = pre.L2.h.T @ dEdz                  # dEdW
7      dEdb = np.sum(dEdz, axis=0)               # dEdb
8      dEdh = dEdz @ post.L1.W.T                 # dEdh
9      post.L1.W -= lrate * dEdW                 # update
10     post.L1.b -= lrate * dEdb                 # update
```

[1]Note we sum "vertically" because we need `dEdb` to have the same dimension as `b`.

Section 13. Overfitting

Overfitting

The false sense of success we get from the results on our training dataset is known as **overfitting**. If your model has enough flexibility and you train it long enough, it will start to fit the specific points in your training dataset rather than the underlying phenomenon that produced the noisy data. A common practice is to keep some of your data as **test data** which your model does not train on. If you see the test set loss has stopped decreasing (or even started increasing) while your training set loss continues decreasing, then it is likely that your model starts to overfit.

Validation

If we want to estimate how well our model will generalize to samples it hasn't trained on, we can without 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.

Regularization

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. Let us define a new loss function by adding an extra term to the original loss:

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

How does this change our gradients (and thus our update rule)? Observe that

$$\begin{aligned} \nabla_{\theta_i}\tilde{E} = \nabla_{\theta_i}E + \bar{\lambda}\theta_i \implies & \theta_i \leftarrow \theta_i - \kappa\nabla_{\theta_i}E - (\bar{\lambda}\kappa)\theta_i \\ & \theta_i \leftarrow \theta_i - \kappa\nabla_{\theta_i}E - \lambda\theta_i \qquad \lambda := \bar{\lambda}\kappa \end{aligned}$$

The hyperparameter λ controls the weight of the regularization term. Note one can also use other norms, e.g., the L_1 norm tends to favour sparcity (most weights are close to zero, with only a few of non-zero weights).

Data Augmentation

Another approach is to include a wide variety of samples in your training set, so that the model is less likely to focus on its efforts on the noise of a few. For image-recognition tasks, one can generate more samples by *shifting* or *rotating* the images. Those transformations presumably do not change the labelling.

Dropout

When 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. Do both a feedforward and backprop pass with the diminished network. More explicitly, for each node h, define

$$\tilde{h} = \begin{cases} 0 & \text{with probability } \alpha \\ h & \text{with probability } 1-\alpha \end{cases}$$

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

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

With dropout, the *expected* absolute input is

$$\tilde{Z} = \sum_{k=1}^{k} |\tilde{z}_k| = \sum_{k=1}^{K} |[\tilde{h}W]_k| = \sum_{k=1}^{K} |[(1-\alpha)hW]_k| = (1-\alpha)Z.$$

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

After training, we want to the network to work without dropped nodes. The weights learned with dropout will not work properly in the full network. The solution is to *scale* the output of the dropout layer by a factor of $\frac{1}{1-\alpha}$.

We can accomplish all of this by adding another layer. The activation function of the (newly-added) dropout layer is

$$d(h) = \frac{1}{1-\alpha}\tilde{h}$$

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

$$\tilde{h} = d(h)$$

$$\nabla_h L = \nabla_{\tilde{h}} L \frac{d\tilde{h}}{dh} \quad \text{where} \quad \frac{d\tilde{h}}{dh} = \begin{cases} 0 & \text{if } h \text{ was dropped} \\ \frac{1}{1-\alpha} & \text{if } h \text{ was not dropped} \end{cases}$$

So 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.

Section 14. Enhancing Optimization

2.10. Note (SGD)**:** Computing the gradient of the cost function can be very expensive and time-consuming, especially if you have a huge dataset. Rather than computing the full gradient, we can try to get a cheaper estimate by computing the gradient from a random sample. More explicitly, let γ be a random sampling of B elements from $\{1, 2, \dots, D\}$. Then we can estimate $\mathbb{E}(Y, T)$ using

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

We refer to $\{(y_{\gamma_1}, t_{\gamma_1}), \dots, (y_{\gamma_B}, t_{\gamma_B})\}$ 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**.

2.11. Note (Momentum)**:** Thus far, we have been moving through parameter space by stepping in the direction of the gradient. What if we thought of the gradient as a *force* that pushes us? Let θ denote our position. From physics,

$$\frac{d\theta}{dt} = V, \quad \frac{dV}{dt} = A.$$

Solving numeracally using Euler's method,

$$\begin{aligned} \theta_{n+1} &= \theta_n + \Delta t \cdot V_n \\ V_{n+1} &= (1 - r)V_n + \Delta t \cdot A_n. \end{aligned}$$

Here r denotes the resistance from friction. We can treat our error gradients as acceleration A, integrate and get velocity V and thus momentum. It's like our weights are dictated by our location in the 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 notation, for each $W^{(\ell)}$, we have

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

or as is commonly used,

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

Thus, update our weight 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.

2.12. For the rest of this course, you can read my friend Sibelius's notes here: CS 479 - Neural Networks. I was too lazy to compile a detailed set of notes as the professor was going over them at a very high level.

Chapter 3. PyTorch

Section 15. Introduction to PyTorch

AutoDiff

The fundamental variable type in PyTorch is called a `tensor`. It's like a NumPy `array` but with extra auto-differentiation functionality:

```
1    z = torch.tensor([1, 2], dtype=torch.float, requires_grad=True)
2    y = torch.prod(z)        # y = z[0] * z[1] = 1 * 2
3
4    y.backward()
5    print(z.grad)            # dy/dz = [2, 1]
```

You can also temporarily suspend autograd by using a `torch.no_grad()` block. Statement inside this block will not impact the gradients.

```
1    z = torch.tensor([1, 2], dtype=torch.float, requires_grad=True)
2    y1 = torch.prod(z)       # y1 = z[0] * z[1] = 1 * 2
3
4    with torch.no_grad():
5        y2 = torch.sum(z)    # dy2/dz = [1, 1]
6
7    y = y1 + y2
8    y.backward()
9    print(z.grad)            # dy/dz = [2, 1]
```

Without this call to `torch.no_grad()`, `dy/dz = [3, 2]` as `dy2/dz` will be added to `dy/dz`.

When a tensor is used in multiply calculations, its gradients can be added together (by specifying `retain_graph=True`.

```
1    z = torch.tensor([1, 2], dtype=torch.float, requires_grad=True)
2    y = torch.prod(z)        # y = z[0] * z[1] = 1 * 2
3
4    y.backward(retain_graph=True)
5    print(z.grad)            # z.grad = dy/dz = [2, 1]
6
7    b = torch.sum(z)         # db/dz = [1, 1]
8    b.backward()
9    print(z.grad)            # z.grad = dy/dz + db/dz = [3, 2]
```

Note how `z.grad` comes from both `dy/dz` and `db/dz`.

Tensors and Arrays

To *create* a `tensor` from an `array`, call `torch.tensor(arr)`. This returns a new tensor (fresh memory) with the same value as `arr`.

To *view* a tensor `t` as a NumPy array, call `t.numpy()`. Note this returns a reference to the tensor, i.e., changing the array will affect the tensor since they share the same memory.

To *view* a tensor `t` with grad information, call `t.detach().numpy()`. The `detach()` function detaches `t` from the expression graph.

NN with PyTorch

First, a sequential NN:

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

`Linear` corresponds to the `Connection` layer we've seen before, which multiplies the input by a weight matrix then adds the biases. `ReLU` is like the `Population` layer, which applies the specified application function to the input.

Alternatively, you can create a subclass of `torch.nn.Module` that represents your NN. This approach gives you more flexibility, since you get to specify the `forward` function.

```
1  class mynet(torch.nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.lyr = torch.nn.ModuleList()    # Note: self.lyr = [] DOES NOT WORK
5          self.lyr.append(torch.nn.Linear(6, 10, bias=False))
6          self.lyr.append(torch.nn.Sigmoid())
7          self.lyr.append(torch.nn.Linear(10, 5))
8          self.lyr.append(torch.nn.Sigmoid())
9          self.lyr.append(torch.nn.Linear(5, 3))
10         self.lyr.append(torch.nn.LogSoftmax(dim=1))
11
12     def forward(self, x):
13         # Here is where you can be creative
14         y = x
15         for l in self.lyr:
16             y = l(y)
17         return y
```

Here's a full example:

```
1    net = mynet()                                   # instantiate the model
2    loss_func = torch.nn.NLLLoss(reduction='mean')  # loss function
3    y = net(ds.inputs())                            # network output
4    loss = loss_func(y, ds.classes())               # compute loss
5    print(loss)
```

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

```
1    for p in net.parameters():
2        print(p.shape)
3
4    params = list(net.parameters())
5    print(params[3])
```

We can access the parameter gradients and implement gradient descent ourselves.

```
1    net = mynet()
2    lrate = 1
3    n_epochs = 200
4    losses = []
5    for epoch in range(n_epochs):
6        y = net(x)
7        err = loss_fcn(y, classes)
8        losses.append(err.item())   # for plotting purposes
9        net.zero_grad()
10       err.backward()
11       with torch.no_grad():
12           for p in net.parameters():
13               p -= lrate * p.grad
14
15   plt.plot(losses)
```

Note the call to `no_grad()`: we do not want to track the gradients as we are updating them.

Now instead of doing Lines 11 to 13 (simple gradient descent), we can use built-in functions to train the model automatically. We need to define an optimizer (pick one from `torch.optim`), then call its `step()` function.

```
1    # "Optimize net.parameters() with a learning rate of 1, using SGD"
2    # Note that different optimizers could ask for different parameters.
3    optim = torch.optim.SGD(net.parameters(), lr=1)
4
5    optim.step() # inside the for-loop, replace Line 11 to 13.
```