



# *Neural Networks*

CS 479

Jeff Orchard

# Preface

---

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

Note that the course currently is CS 489, but in winter 2022, this course will become [CS 479](#).

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

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

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

---

Sibelius Peng

# Contents

---

<b>Preface</b>	<b>1</b>
<b>1 Neuron Models</b>	<b>4</b>
1.1 Neurons . . . . .	4
1.2 Neuron Membrane Potential . . . . .	4
1.3 Hodgkin-Huxley Model . . . . .	5
1.4 Leaky Integrate-and-Fire Model . . . . .	8
1.5 Activation functions . . . . .	10
1.6 Synapses . . . . .	11
1.7 Connection Weight . . . . .	12
<b>2 Formulation of Learning</b>	<b>15</b>
2.1 Neural Learning . . . . .	15
2.2 Universal Approximation Theorem . . . . .	17
2.3 Loss Functions . . . . .	19
2.3.1 (Mean) Squared Error . . . . .	19
2.3.2 Cross Entropy (Bernoulli Cross Entropy) . . . . .	20
2.3.3 Categorical Cross-Entropy (Multinoulli Cross-Entropy) . . . . .	20
<b>3 Error Backpropagation</b>	<b>22</b>
3.1 Gradient Descent Learning . . . . .	22
3.1.1 Gradient-Based Optimization . . . . .	22
3.1.2 Approximating the Gradient Numerically . . . . .	23
3.2 Error Backpropagation . . . . .	24
<b>4 Automatic Differentiation</b>	<b>27</b>
4.1 Theory . . . . .	27
4.1.1 Evaluate . . . . .	28
4.1.2 Differentiate . . . . .	29
4.2 Neural Networks with Auto-Diff . . . . .	30
4.2.1 Optimization . . . . .	30
4.2.2 Neural Learning . . . . .	30
4.2.3 Matrix AD . . . . .	32
<b>5 Generalizability</b>	<b>34</b>
5.1 Overfitting . . . . .	34
5.2 Validation . . . . .	35
5.3 Combatting Overfitting . . . . .	36
5.3.1 Regularization . . . . .	36
5.3.2 Data Augmentation . . . . .	37

5.3.3	Dropout . . . . .	38
<b>6</b>	<b>Optimization Considerations</b>	<b>40</b>
6.1	Vanishing Gradients . . . . .	40
6.2	Exploding Gradients . . . . .	42
6.3	Enhancing Optimization . . . . .	42
6.3.1	Stochastic Gradient Descent . . . . .	42
6.3.2	Momentum . . . . .	43
<b>7</b>	<b>Special Architectures</b>	<b>45</b>
7.1	Autoencoders . . . . .	45
7.2	Your Visual System . . . . .	52
7.3	Convolutional Neural Networks . . . . .	54
<b>8</b>	<b>Hopfield Networks</b>	<b>56</b>
8.1	Batch Normalization . . . . .	56
8.2	Hopfield Networks . . . . .	58
8.2.1	Content-Addressable Memory . . . . .	58
8.3	Hopfield Demonstration . . . . .	60
8.3.1	Toy Dataset . . . . .	62
8.3.2	Letters Example . . . . .	64
<b>9</b>	<b>VAE</b>	<b>67</b>
9.1	Helpful Resources . . . . .	67
9.2	Variational Autoencoders . . . . .	67
<b>10</b>	<b>RNN</b>	<b>72</b>
10.1	Recurrent Neural Networks . . . . .	72
10.1.1	Backprop through time (BPTT) . . . . .	73
10.1.2	Long Short-Term Memory (LSTM) . . . . .	75
10.2	Our memory systems . . . . .	77
<b>11</b>	<b>Adversarial Attack and Defense</b>	<b>79</b>
11.1	Adversarial attack . . . . .	79
11.2	Adversarial defence . . . . .	82
11.2.1	Adversarial training . . . . .	82
11.2.2	TRADES . . . . .	83
<b>12</b>	<b>GANs</b>	<b>85</b>
12.1	Introduction . . . . .	85
12.2	Loss Function for a GAN . . . . .	86
12.3	Training the Discriminative Network . . . . .	86
12.4	Training the Generative Network . . . . .	86
12.5	Examples . . . . .	86
<b>13</b>	<b>Vector Embeddings</b>	<b>87</b>
<b>A</b>	<b>Implementation of Neural Network</b>	<b>88</b>
<b>B</b>	<b>PyTorch</b>	<b>89</b>
B.1	PyTorch Tensors . . . . .	89
B.2	Classification Dataset . . . . .	91
B.3	Neural Networks with PyTorch . . . . .	93

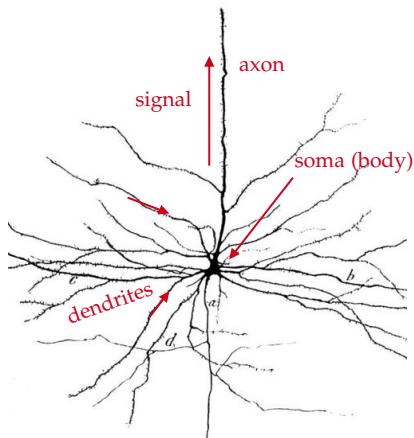
# Neuron Models

---

## 1.1 Neurons

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

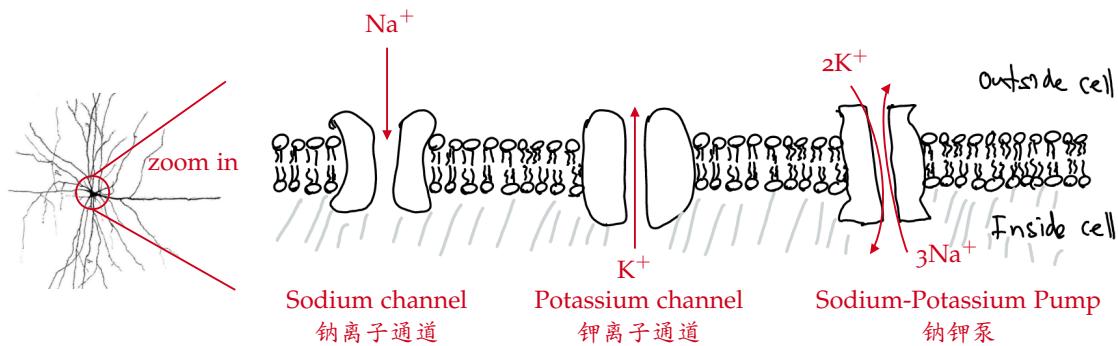
A neuron can be quite long, sending its signal over a long distance; up to **50m long!** But most are much shorter.



- 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  $\text{Na}^+$  ions inside the cell for 2  $\text{K}^+$  ions outside the cell.

- Causes a higher concentration of  $\text{Na}^+$  outside the cell, and higher concentration of  $\text{K}^+$  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  $\text{Na}^+$  and  $\text{K}^+$  ion channels.

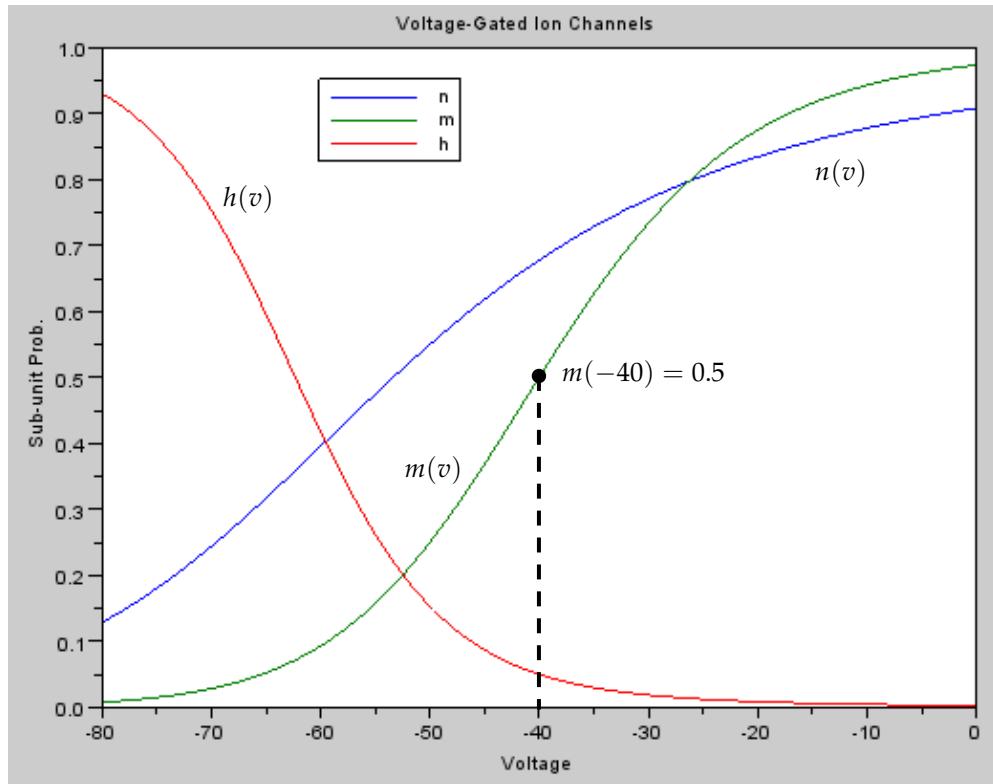
Both  $\text{Na}^+$  and  $\text{K}^+$  ion channels are voltage-dependent, so their opening and closing changes with the membrane potential.

Let  $V$  be the membrane potential. A neuron usually keeps a membrane potential of around  $-70\text{mV}$ .

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

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



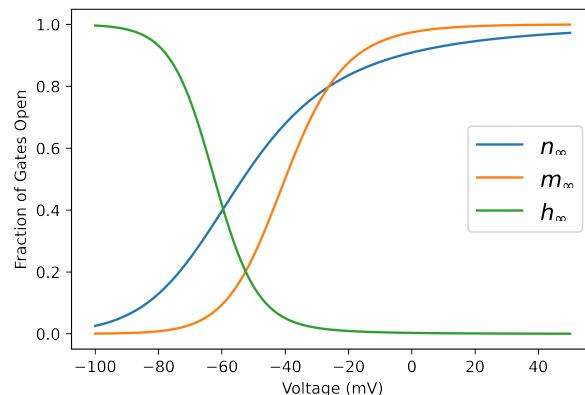
These two channels allow ions to flow into/out of the cell, inducing a current... which affects the membrane potential,  $V$ . Here is a differential equation which governs the membrane potential.

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

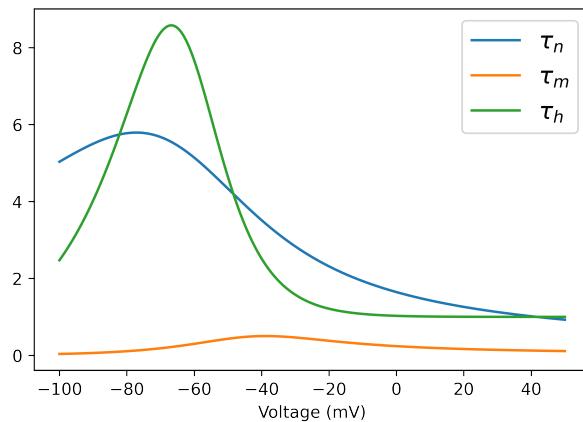
- $C$ : capacitance.
- $\frac{dV}{dt}$ : rate of change in voltage, or current.
- $J_{in}$ : input current, usually from other neurons.
- $V_L, V_{Na}, V_K$ : zero-current potentials.
- $g_L, g_{Na}, g_K$ : max conductance.

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.

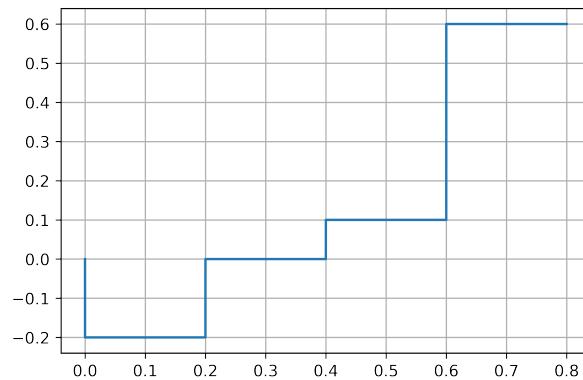
Here we can model this model in python. We have already seen these as functions of voltage.



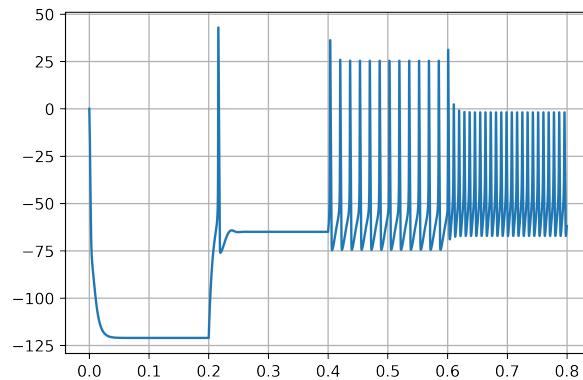
These are the  $\tau$ 's in case you are interested.



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

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

- $\tau_m$ : time constant which dictates how quick things happen.
- $RJ_{in}$ : by Ohm's Law, let  $V_{in} = RJ_{in}$ .

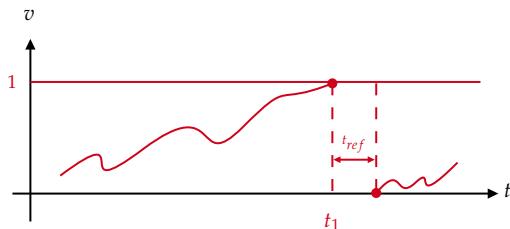
Thus, the voltage can be modelled as

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

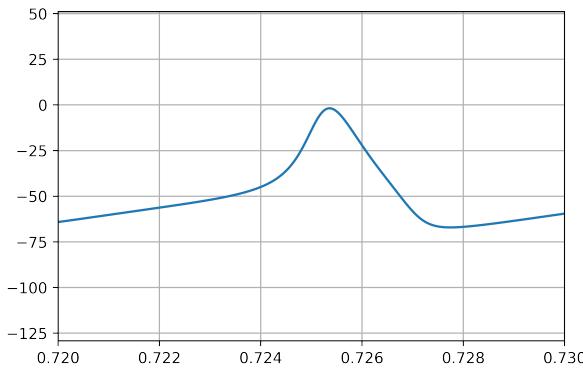
So this is the dynamics of the sub-threshold membrane potential. Change of variables:  $v = \frac{V - V_L}{V_{th} - V_L}$ , then  $v \rightarrow 0$  if  $V_{in} = 0$  and  $v = 1$  is the threshold. Then we end up with a DE:

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

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,  $\tau_{ref}$ , refractory time. It remains dormant during its refractory period,  $\tau_{ref}$  (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:  $\tau \frac{dv}{dt} = v_{in} - v$ ,  $v(0) = 0$ .

#### Proof:

Plug in the solution to the DE and show LHS = RHS.  $\square$

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.

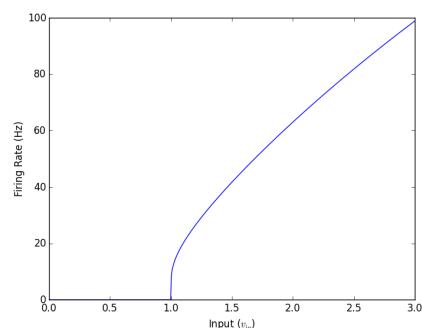
It can be shown that the steady-state firing rate for a constant input  $v_{in}$  is

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

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):

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

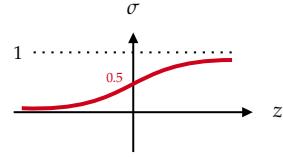


Let's take a look at even simpler neurons.

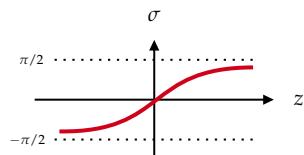
## 1.5 Activation functions

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.

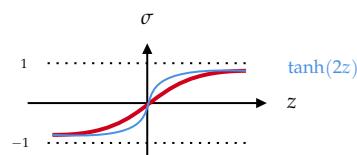
Logistic Curve  $\sigma(z) = \frac{1}{1+e^{-z}}$



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

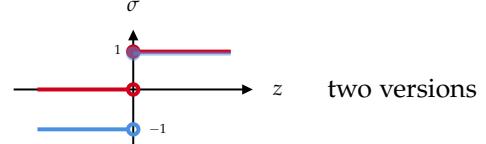


Hyperbolic Tangent  $\sigma(z) = \tanh(z)$



Threshold

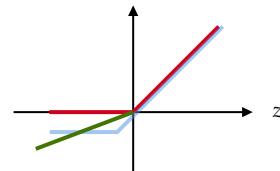
$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



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.

ReLU( $z$ ) =  $\max(0, z)$

LeReLU

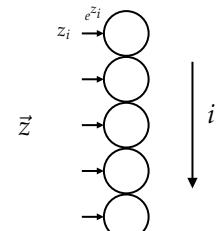


**Multi-Neuron Activation Functions:** Some activation functions depend on multiple neurons. Here are two examples.

### SoftMax

SoftMax is like 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

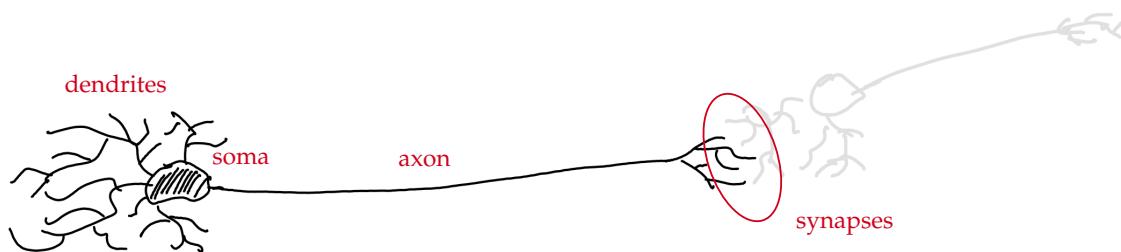
One-Hot is the extreme of the softmax, where only the largest element remains nonzero, while the others are set to zero.

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

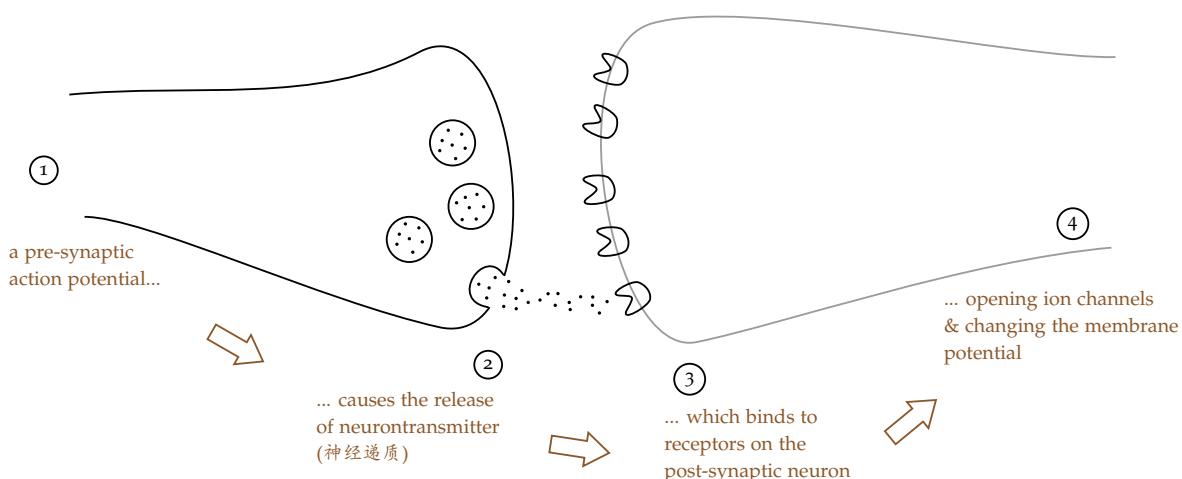
## 1.6 Synapses

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.

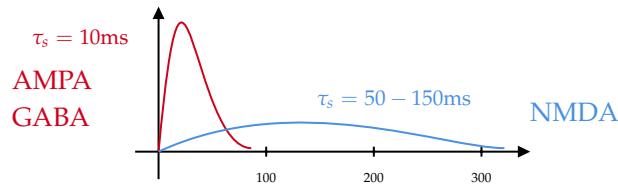


Even though an action potential is very fast, the synaptic processes by which it affects the next neuron takes time. Some synapses are fast (taking just about 10 ms), and some are quite slow (taking over 300 ms). If we represent that time constant using  $\tau_s$ , then the current entering the post-synaptic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

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

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

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 neurotransmitters 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

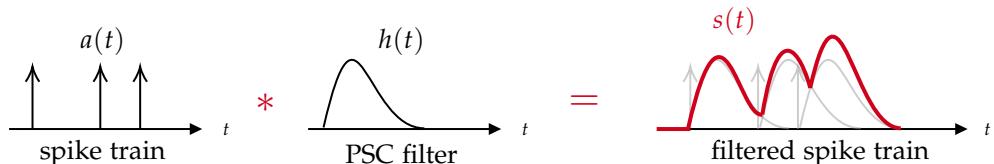
Dirac Delta Function is defined as

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

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,

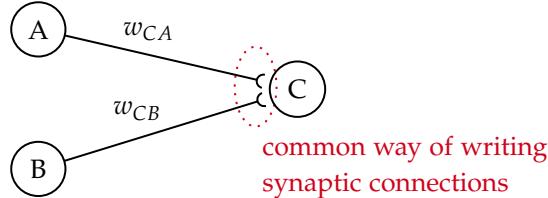
$$s(t) = (a * h)(t) = \sum_p h(t - t_p) = \text{sum of PSC filters, one for each spike}$$

## 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.

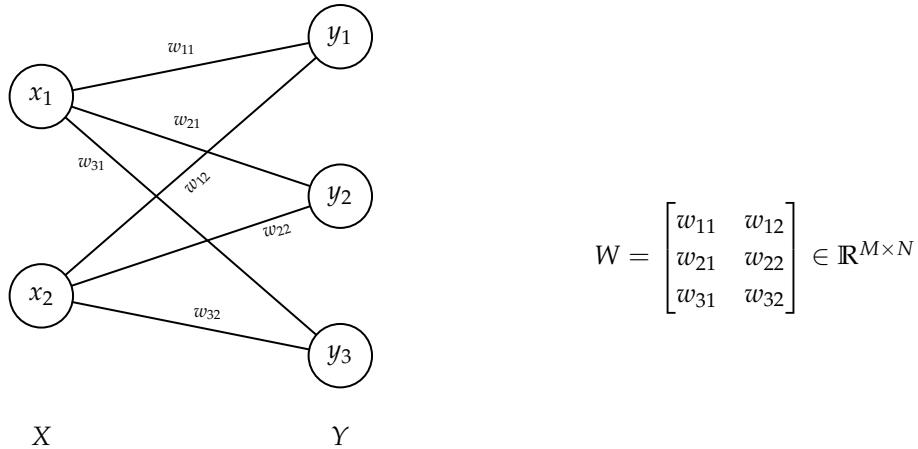
We can combine all those factors into a single number, the **connection weight**. Thus, the total input to a neuron is a weighted sum of filtered spike-trains.



## 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,

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

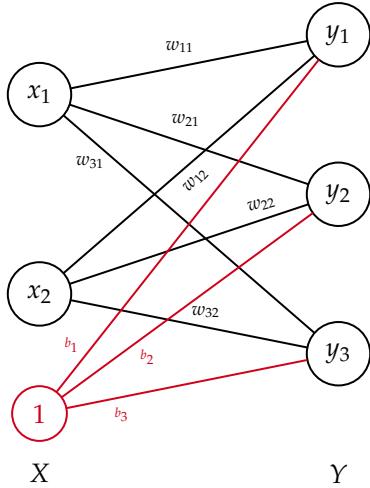
We can compute the input to the nodes in Y using

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

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,  $\vec{b}$ ,



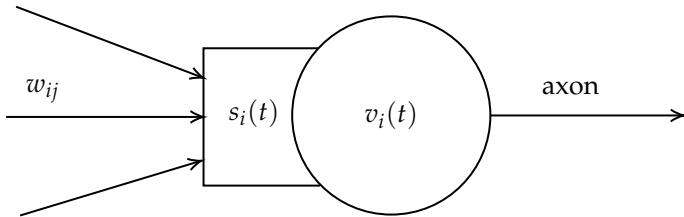
$$\text{So } W\vec{x} + \vec{b} = [W|\vec{b}] \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix} = \hat{W} \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix}.$$

### Implementing Connections between Spiking Neurons

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

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

### Full LIF Neuron Model



Differential equations:

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

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  $\tau_s$ , 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.

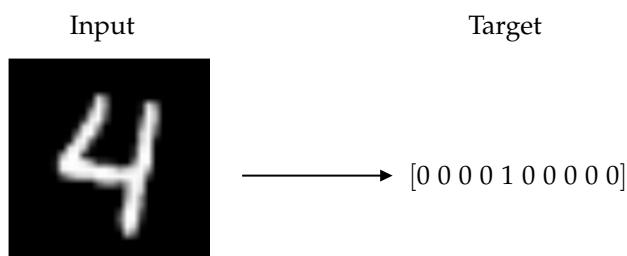
There are three basic categories of learning problems:

- 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.

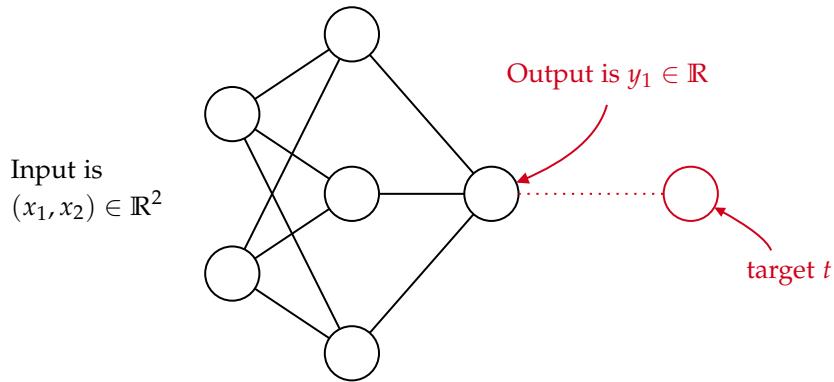
**Example:**

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

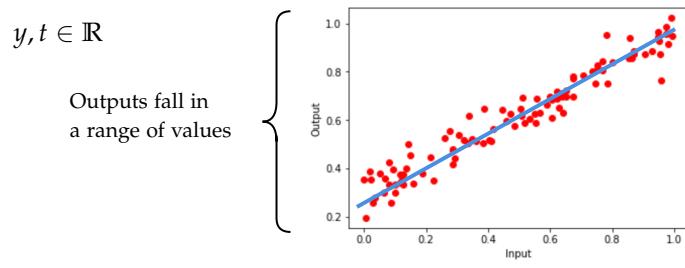
Our task is to alter the connection weights in our network so that our network mimics this mapping. Our goal is to bring the output as close as possible to the target. But what, exactly, do we mean by "close"? For now, we will use the scalar function  $L(y, t)$  as an error (or "loss") function, which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



## Classification

Outputs fall into a number of distinct categories.

### Example: MNIST

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

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

### Example: CIFAR-10

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

## Optimization

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

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

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

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

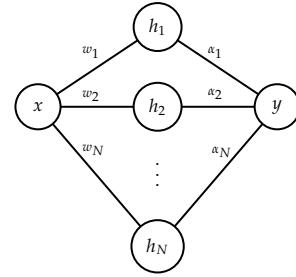
## 2.2 Universal Approximation Theorem

Can we approximate any function using a neural network?

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

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

to arbitrary precision?



### Theorem 2.1: Universal Approximation Theorem

Let  $\sigma$  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

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

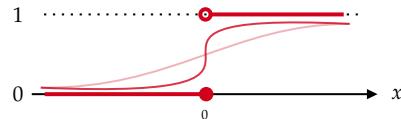
Here  $C(I_n)$  denotes the continuous functions on  $I_n$ , and  $I_n$  can be  $I_n = [0, 1]^n$ .

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

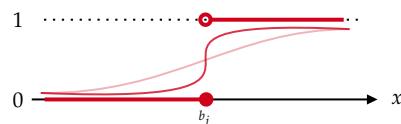
The theorem states that  $\exists N$ , and  $\exists w_j, \theta_j, \alpha_j$  for  $j = 1, \dots, N$  such that  $|G(x) - f(x)| < \epsilon$ .

**Proof:**

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



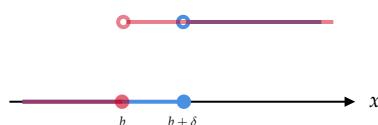
Or let's look at a shifted version of it:  $\sigma(w_j(x - b_j)) \xrightarrow{w_j \rightarrow \infty} \begin{cases} 0 & \text{for } x \leq b_j \\ 1 & \text{for } x > b_j \end{cases}$ .



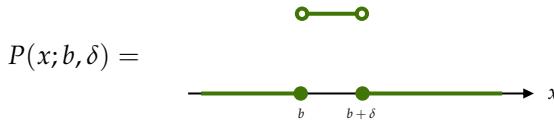
This is the same as the Heaviside step function,  $H(x) = \lim_{w \rightarrow \infty} \sigma(wx)$ .

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

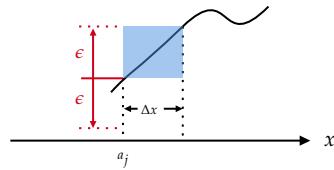
We can use two such functions to create a piece,  $P(x; b, \delta) := H(x; b) - H(x; b + \delta)$



Then,



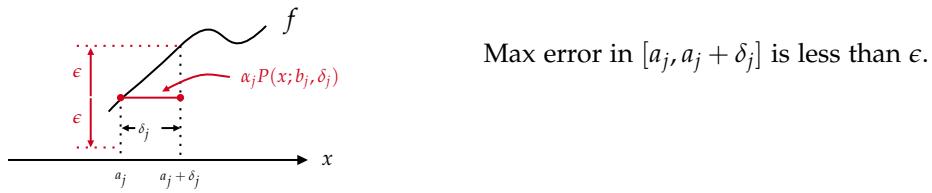
Since  $f(x)$  is continuous,  $\lim_{x \rightarrow a} f(x) = f(a)$ ,  $\forall a \in I_n$ . Then there exists an interval,  $(a_j, a_j + \Delta x)$  such that  $|f(x) - f(a_j)| < \epsilon \ \forall x \in (a_j, a_j + \Delta x)$ .



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

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

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



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.  $\square$

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  $\epsilon$  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

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

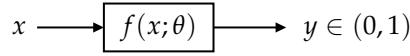
Taking the expectation (mean) over the entire dataset,

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

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.



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

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

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

The likelihood of our data sample given our model is

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

which works for both classes.

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

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

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

The expected cross entropy over the entire dataset is

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

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 (Multinomial 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 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  $\theta$  (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

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

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  $\theta$  are connection weights and biases. So, if our loss function is  $L(y, t)$ , where  $t$  is the target, then neural learning becomes the optimization problem  $\min_{\theta} E(\theta)$  where  $E(\theta) = \mathbb{E}\left[L(f(x; \theta), t(x))\right]_{x \in \text{data}}$ . 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,

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

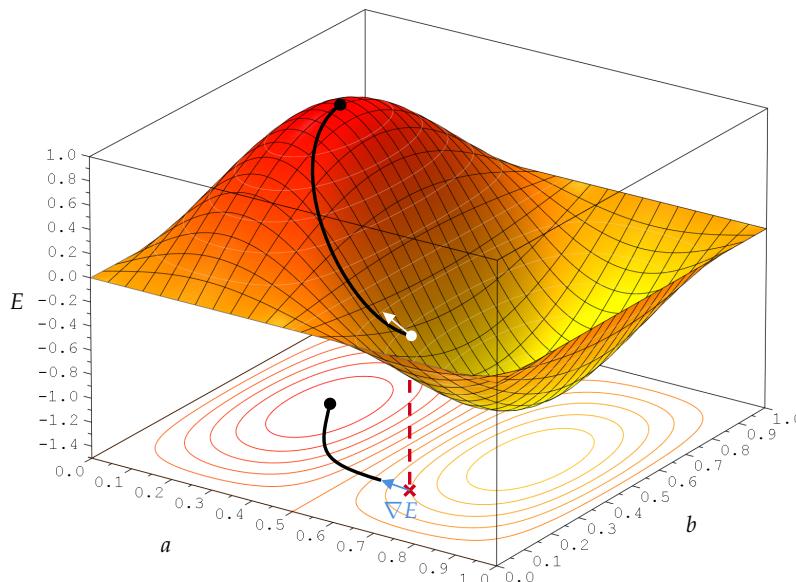


Image from [https://commons.wikimedia.org/wiki/File:2D\\_Wavefunction\\_\(2,1\)\\_Surface\\_Plot.png](https://commons.wikimedia.org/wiki/File:2D_Wavefunction_(2,1)_Surface_Plot.png).

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.

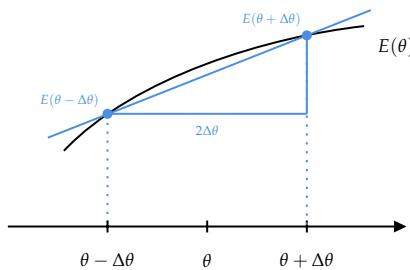
### 3.1.2 Approximating the Gradient Numerically

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

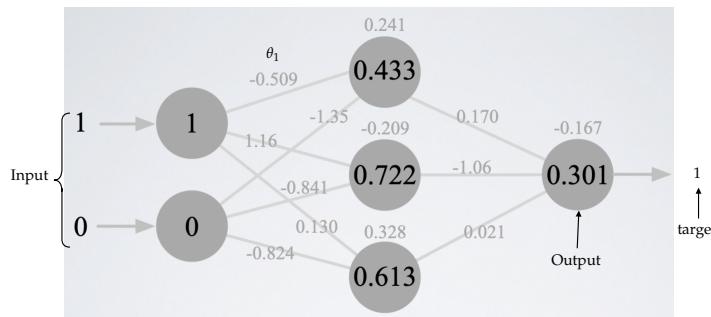
#### Finite-Difference Approximation

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  $\theta_1$  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  $\theta_1$ , so that  $\theta_1 = -0.509 + 0.1 = -0.409$ . Then our output is  $y = 0.302$ . This yields  $E(-0.409) = 1.198$ .

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

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

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

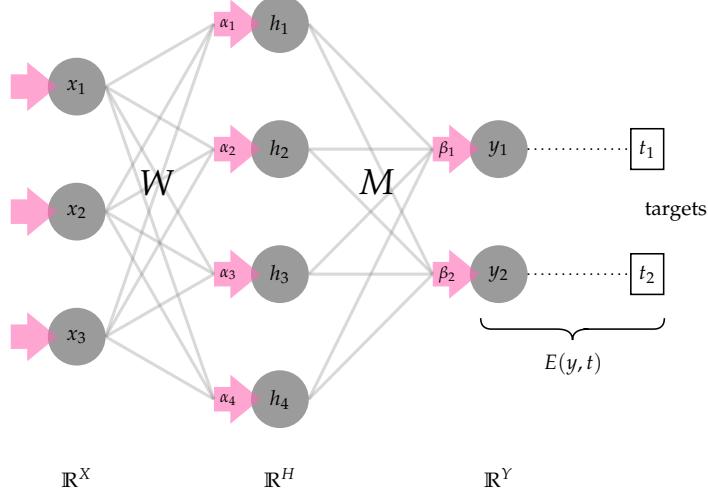
Obviously, increasing  $\theta_1$  seems to be the right thing to do. Then  $\theta_1 \leftarrow \theta_1 + k \cdot (-0.0292)$ .

$\uparrow$   
positive constant

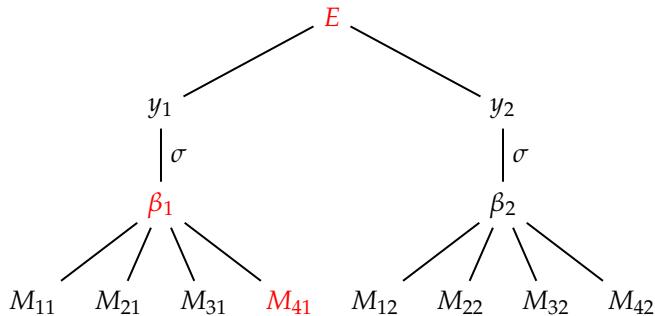
## 3.2 Error Backpropagation

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:



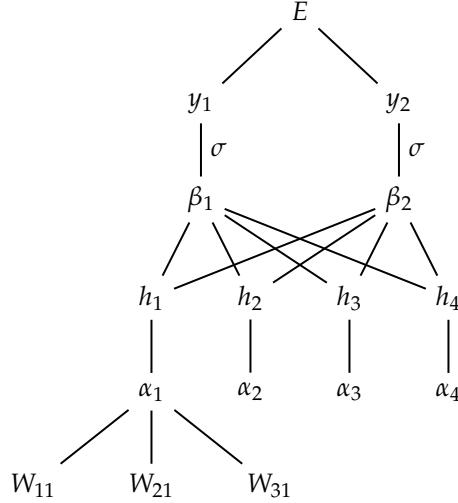
$\alpha_i$  is the input current to hidden node  $i$ .  $\beta_j$  is the input current to the output node  $j$ . For our cost (loss) function, we will use  $E(y, t)$ . For learning, suppose we want to know  $\frac{\partial E}{\partial M_{41}}$ , where  $M_{41}$  is going from  $h_4$  to  $\beta_1$ . We can represent this by a computation/dependency graph.



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

$$\begin{aligned}
 \frac{\partial E}{\partial \alpha_1} &= \frac{\partial E}{\partial h_1} \frac{dh_1}{d\alpha_1} \\
 &= \left( \frac{\partial E}{\partial \beta_1} \frac{\partial \beta_1}{\partial h_1} + \frac{\partial E}{\partial \beta_2} \frac{\partial \beta_2}{\partial h_1} \right) \frac{dh_1}{d\alpha_1} \\
 &= \left( \frac{\partial E}{\partial \beta_1} M_{11} + \frac{\partial E}{\partial \beta_2} M_{12} \right) \frac{dh_1}{d\alpha_1} \\
 &= \left( \frac{\partial E}{\partial \beta_1}, \frac{\partial E}{\partial \beta_2} \right) \cdot (M_{11}, M_{12}) \frac{dh_1}{d\alpha_1}.
 \end{aligned} \tag{*}$$

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

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

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

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

For all elements,

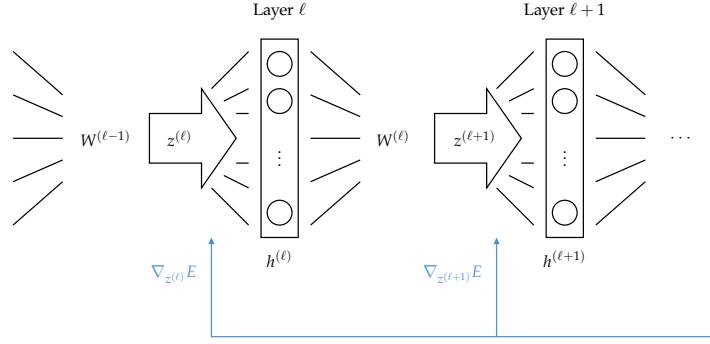
$$\begin{bmatrix} \frac{\partial E}{\partial \alpha_1} & \dots & \frac{\partial E}{\partial \alpha_H} \end{bmatrix} = \begin{bmatrix} \frac{dh_1}{d\alpha_1} & \dots & \frac{dh_H}{d\alpha_H} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial E}{\partial \beta_1} & \dots & \frac{\partial E}{\partial \beta_Y} \end{bmatrix} \begin{bmatrix} M_{11} & \dots & M_{H1} \\ \vdots & & \vdots \\ M_{1Y} & \dots & M_{HY} \end{bmatrix}$$

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

$$\nabla_\alpha E = \frac{dh}{d\alpha} \odot (\nabla_\beta E \cdot M^T).$$

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

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  $h^{(\ell+1)} = \sigma(z^{(\ell+1)}) = \sigma(h^{(\ell)} W^{(\ell)} + b^{(\ell+1)})$ . Then in our context,

$$\nabla_{z^{(\ell)}} = \frac{dh^{(\ell)}}{dz^{(\ell)}} \odot [\nabla_{z^{(\ell+1)}} E \cdot (W^{(\ell)})^T]$$

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.

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

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  $\sigma$  is the activation function between  $z^{(\ell)}$  and  $h^{(\ell)}$ .

$$\begin{aligned} \nabla_{z^{(\ell)}} E &= \sigma'(z^{(\ell)}) \odot [\nabla_{z^{(\ell+1)}} E \cdot (W^{(\ell)})^T] \\ \nabla_{W^{(\ell)}} E &= [h^{(\ell)}]^T \nabla_{z^{(\ell+1)}} E \end{aligned}$$

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

# 4

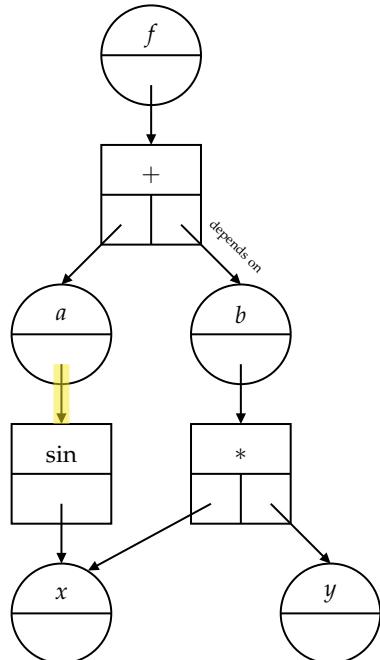
## Automatic Differentiation

### 4.1 Theory

Consider  $f = \underbrace{\sin(x)}_a + \underbrace{xy}_b$ .

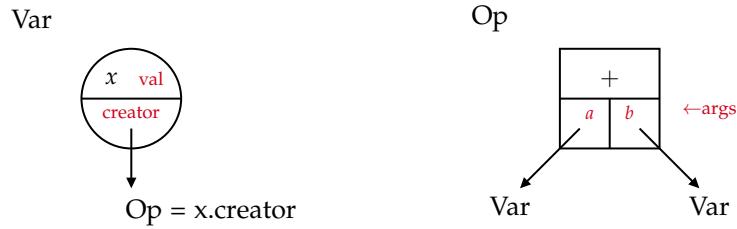
```
1 x = var
2 y = var
3 a = sin(x)
4 b = x * y
5 f = a + b
```

Let's draw the computation graph/expression graph.

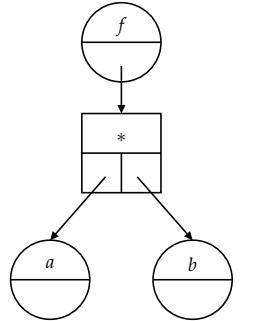


We refer the yellow line by “creator” pointer/reference:  $a$  was created from the sine function.

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. Send  $f.creator$  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.

```

1 # Var.evaluate
2 if creator is None:
3     return val
4 else:
5     return creator.evaluate()
6
7 # Op.evaluate
8 call evaluate on all the args.
9 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()
            return a.creator.evaluate()

            a.creator.evaluate()
                return sin(x.evaluate())

                x.evaluate()
                    return x.val

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

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

                y.evaluate()
                    return y.val

```

The code defines the `evaluate` method for both **Var** and **Op** objects. Red arrows highlight the flow of data from the input variables  $x$  and  $y$  through their creators to the final result  $f$ .

### 4.1.2 Differentiate

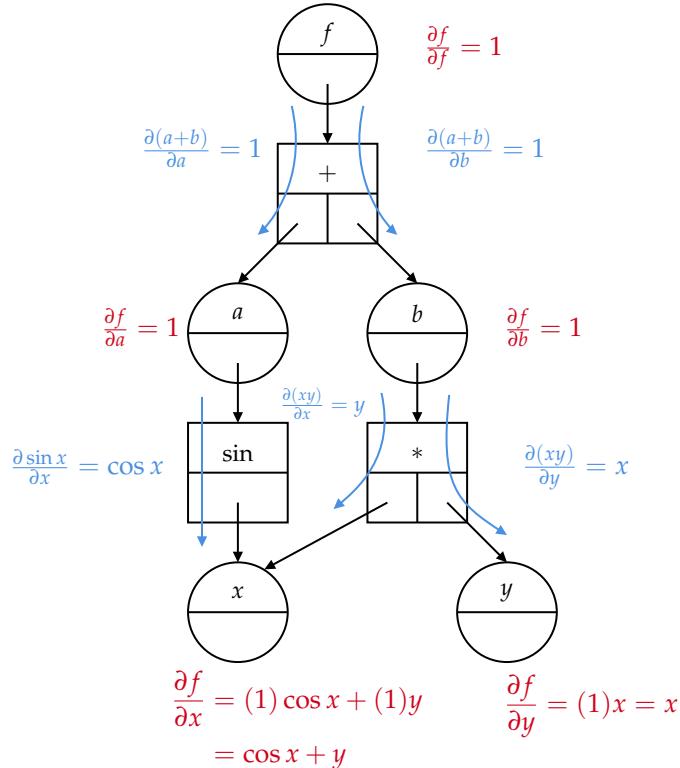
The expression graph can also be used to compute the derivatives. Each Var stores the derivative of the expression w.r.t. itself. It stores it in its member `grad`.

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

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

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:  $f = \sin(x) + xy$ .



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

```

1 class Var:
2     # self.var, self.grad, and s all have to be the same shape

```

```

3     def backward(s):
4         self.grad += s
5         self.creator.backward(s)
6
7 class Op:
8     # s must match the shape of the operator's output
9     def backward(s):
10        for x in self.args:
11            x.backward(s + ∂Op/∂x)

```

Here,  $s$  is the accumulated derivative of the part of the expression above the  $\text{Var}/\text{Op}$ .

## 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)

---

- 1 initialize  $v, k$
  - 2 construct an expression graph for  $E$
  - 3 **while** not converged **do**
  - 4 evalaute  $E$  at  $v$
  - 5 set gradients to zero (i.e.,  $\nabla_v E = v.\text{grad} = 0$ )
  - 6 propagate derivatives down (increment  $v.\text{grad}$ )
  - 7  $v \leftarrow v - k \cdot v.\text{grad}$
- 

### 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.

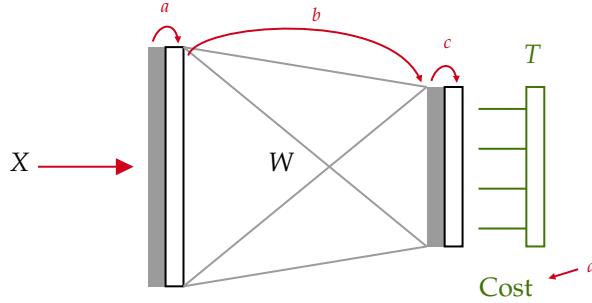
To accomplish this, our network will be composed of a series of layers, each layer transforming the data from the layer below it, culminating in a scalar-valued cost function.

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:



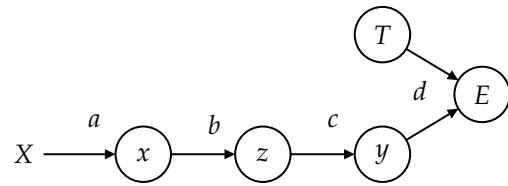
$X$ : input. Gray region represents the input current for that given layer. From the gray area to the outlined box, is the activation function, which we called  $a$ . Then we go through the connection weights, and we call it  $b$ . Then similarly  $c$  is the activation function. Lastly, we call the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

$$a = \text{identity}$$

$$b = (\lambda z : z \cdot W) \quad (\text{multiply by } W)$$

$$c = \text{logistic}$$

$$d = \text{Cost}$$



Each layer can be called like a function:

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

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

$$E = d\left(c\left(b(a(X))\right), T\right)$$

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,  $\text{net} \equiv (a, b, c)$ . And

$$y = \text{net}(x) = c(b(a(x)))$$

$$E = d(y, T)$$

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

---

```

1 Given dataset  $(X, T)$ , and network model  $net$ , with parameters  $\theta$ , and cost function “ $Cost$ ”
2 for  $epochs \dots$  do
    // these two is the feedforward pass
3      $y = net(X)$ 
4      $loss = Cost(y, T)$ 
    // Backprop
5      $loss.zero\_grad()$ 
6      $loss.backward()$ 
    // Gradient descent
7      $\theta \leftarrow \theta - \kappa \cdot \theta.grad$ 

```

---

### 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 ⊙ 1_{M × N})
B.backward(s ⊙ 1_{M × 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  $\nabla_B L$ ?

$$\begin{aligned} \nabla_A L &= \nabla_y L \cdot \nabla_A y = s \cdot B^T \\ \nabla_B L &= \nabla_y L \cdot \nabla_B y = A^T \cdot s \end{aligned}$$

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.

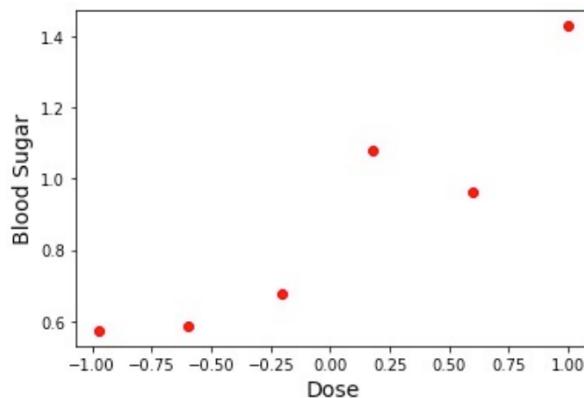
# 5

## Generalizability

---

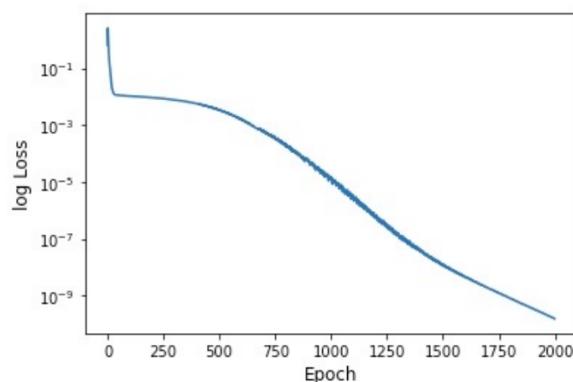
### 5.1 Overfitting

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.

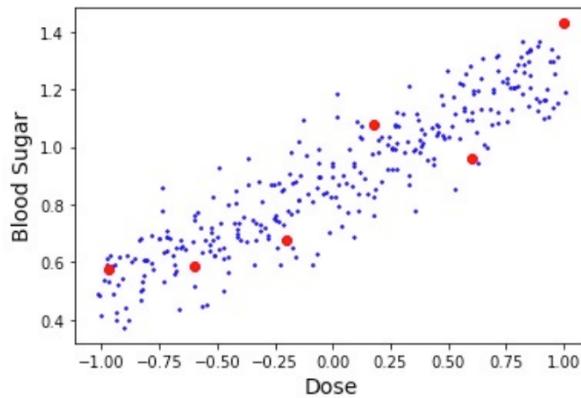
Final loss =  $1.5741956349568653e-10$



The doctor wants 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.

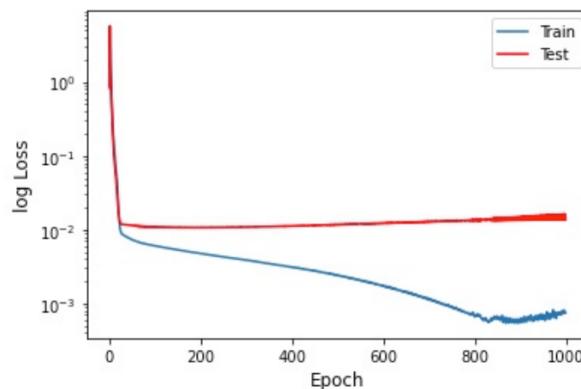


That's orders of magnitude worse than the  $1.5 \times 10^{-10}$  on the training dataset. The false sense of success we get from the results on our training dataset is known as *overfitting* or *overtraining*.

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.



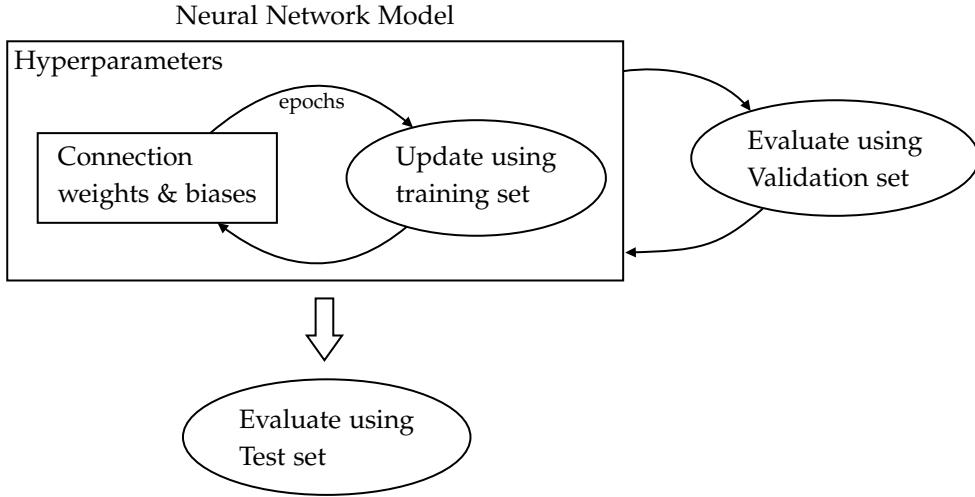
So we see the train data gets down as epochs go up. But in terms of the test data, we see it goes down first and then creeps up again. That's because the model is working so hard to fit the training data, including the noise of the training data, at the expense of fitting the test data. Thus divergence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

## 5.2 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.



## 5.3 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.

### 5.3.1 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

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

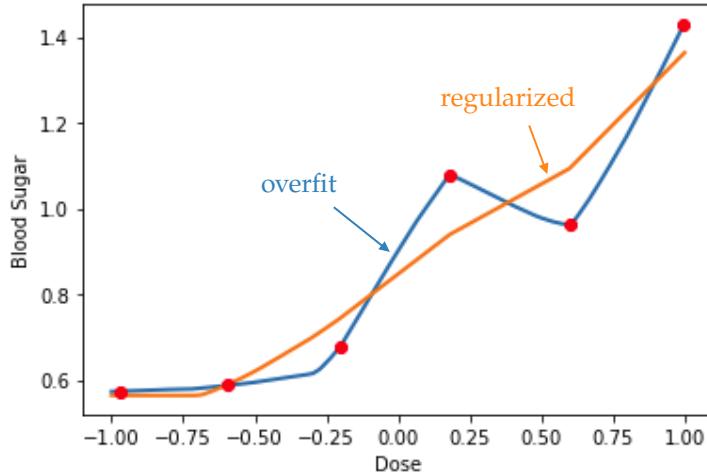
is the Frobenius norm.

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  $\lambda$  controls the weight of the regularization term.

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

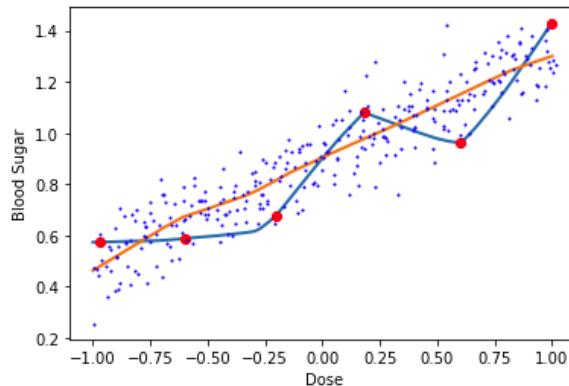
$$L_1(\theta) = \sum_i |\theta_i|,$$

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

### 5.3.2 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.



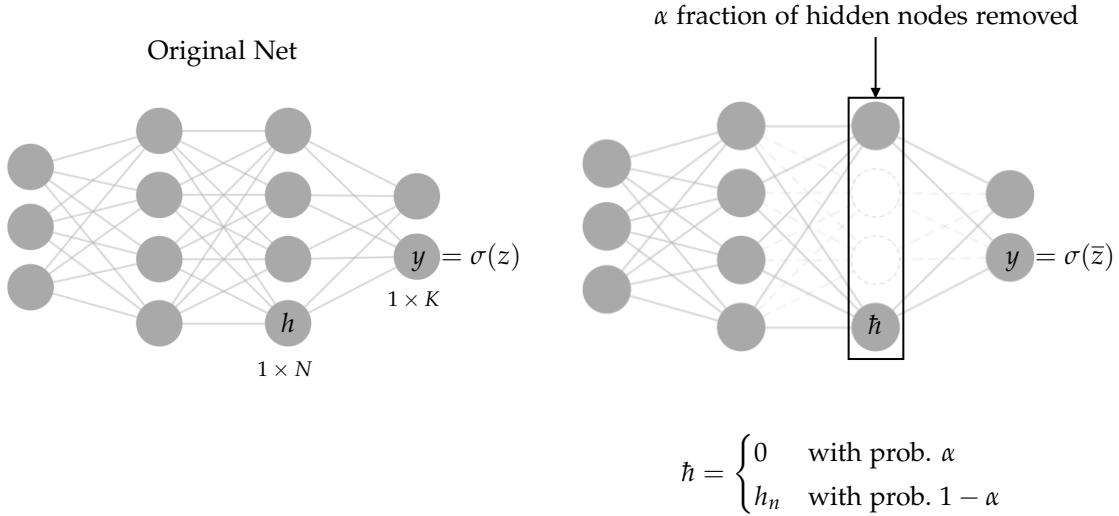
Original test loss is 0.01728 while augmented test loss is 0.00934.

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.

### 5.3.3 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,  $\alpha$ , each hidden node will be dropped with probability  $\alpha$ . A dropped node is temporarily taken off-line and set to zero.



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

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

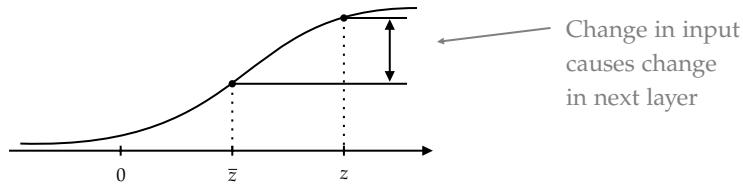
Consider the absolute input to the nodes in the output layer. Without dropout,

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

With dropout, the *expected* absolute input

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

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

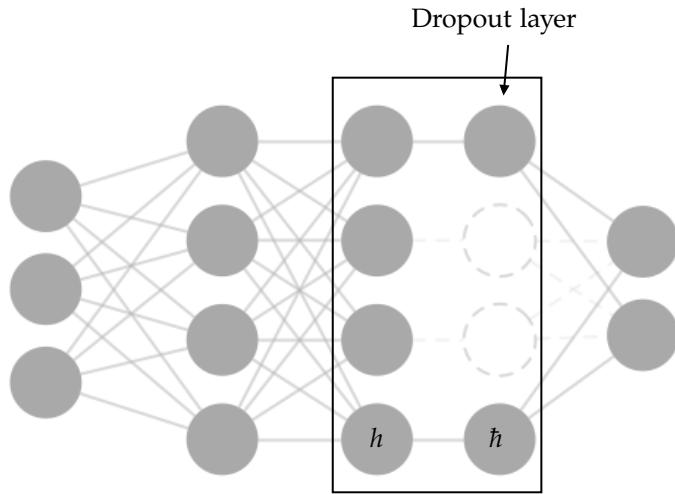


After training, we want the network to work without dropped nodes. The weights learned with dropout will not work properly in the full network.

*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

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

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



The activation function of the dropout layer is

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

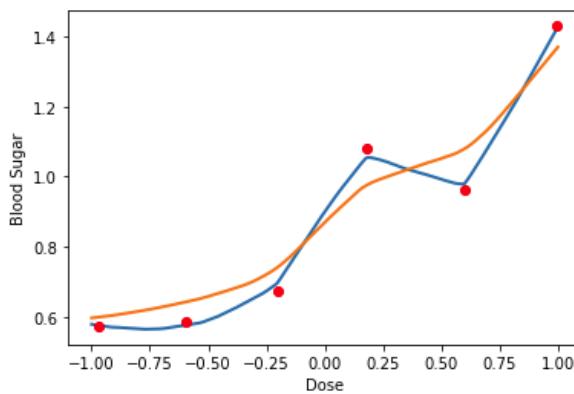
During backprop, the gradients have to go back through this layer.

$$\tilde{h} = d(h), \quad \nabla_h L = \nabla_{\tilde{h}} L \frac{d\tilde{h}}{dh}$$

where

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

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  $\sigma$  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

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

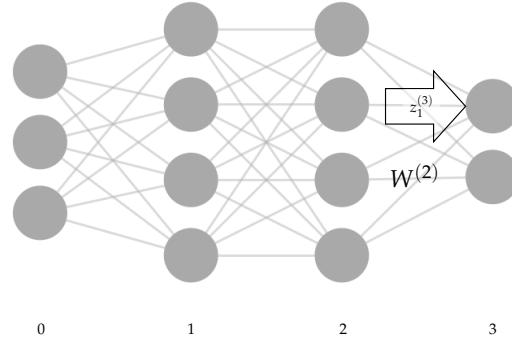
Thus, we really only ever need one hidden layer. But is that the best approach, in terms of number of nodes, or learning efficiency? No, it can be shown that such a shallow network could require an exponentially large number of nodes (i.e., A really big  $N$ ) to work.

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

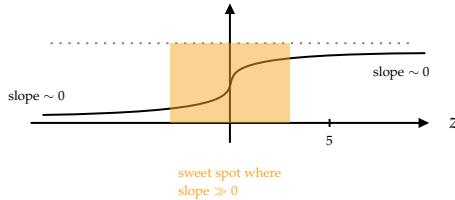
Let us visit one problem: **vanishing gradients**.

### 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  $y_1 = \sigma(z_1^{(3)}) = \frac{1}{1+e^{-5}} = 0.9933$ . Therefore,  $\frac{dy_1}{dz_1^{(3)}} = y_1(1 - y_1) = 0.0066$ , which is small. Because the input current is high, then the derivative is small. Let's demonstrate it in a plot.



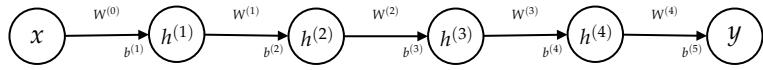
Compare that if the input current was 0.1, then  $y_1 = \sigma(0.1) = 0.525$ , and  $\frac{dy_1}{dz_1^{(3)}} = y_1(1 - y_1) = 0.249$ , which is almost 40 times bigger than before. Hence, the updates to the weights will be smaller when the input currents are large in magnitude.

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

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

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

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

Going deeper... we have

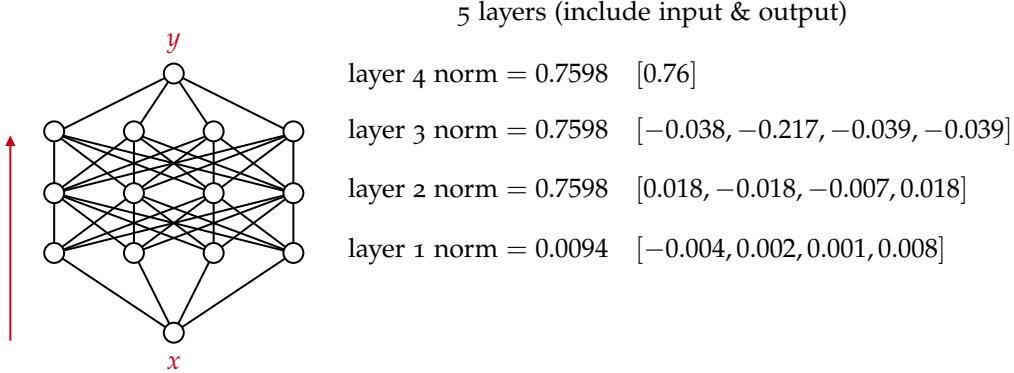
$$\frac{\partial E}{\partial z^{(1)}} = (y - t) W^{(4)} \sigma'(z^{(4)}) W^{(3)} \sigma'(z^{(3)}) W^{(2)} \sigma'(z^{(2)}) W^{(1)} \sigma'(z^{(1)}).$$

What is the steepest slope 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.,

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

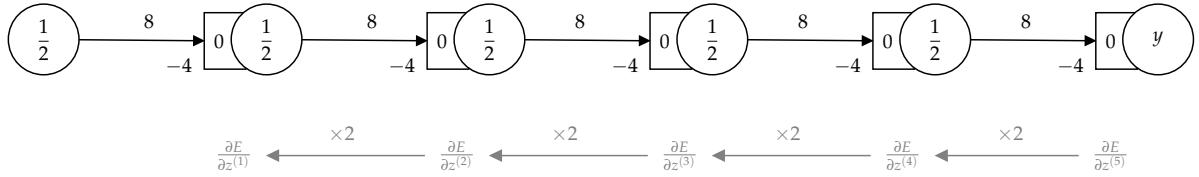
Consider this particular example below:



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  $\sigma'$  lays on each layer.

## 6.2 Exploding Gradients

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



$\sigma' = \frac{1}{4}$  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  $\gamma$  be a random sampling of  $B$  elements from  $\{1, 2, \dots, D\}$ . Then we estimate  $E(Y, T)$  using

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

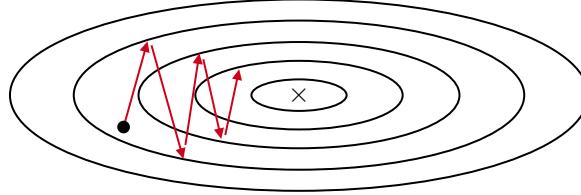
We refer to  $\{(y_{\gamma_1}, t_{\gamma_1}), \dots, (y_{\gamma_B}, t_{\gamma_B})\}$  as a *batch*. We use the estimate from this batch to update our weights, and then choose subsequent batches from the remaining samples. This method is called **Stochastic Gradient Descent**.

### 6.3.2 Momentum

Consider gradient descent optimization in these situations...

#### Case 1

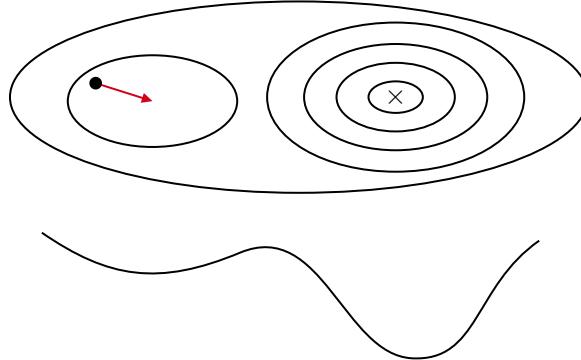
First let's consider the level curves of the objective function. Here  $x$  is the minimum we want to achieve, and we are starting at the  $\bullet$ .



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:<sup>1</sup>

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

---

<sup>1</sup>Note that here prof used a quite confusing notation:  $\theta_n$  is the old set of weights and biases, while  $\theta_{n+1}$  is the new set of weights and biases. This is different from  $\theta_i$  before, which is a particular weight/bias.

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

$$\frac{d\theta}{dt} = V \quad (\text{velocity}), \quad \frac{dv}{dt} = A \quad (\text{acceleration}).$$

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<sup>2</sup>,

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

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.

---

<sup>2</sup>another variation is  $V^{(\ell)} \leftarrow \beta V^{(\ell)} + (1 - \beta) \nabla_{W^{(\ell)}} E$

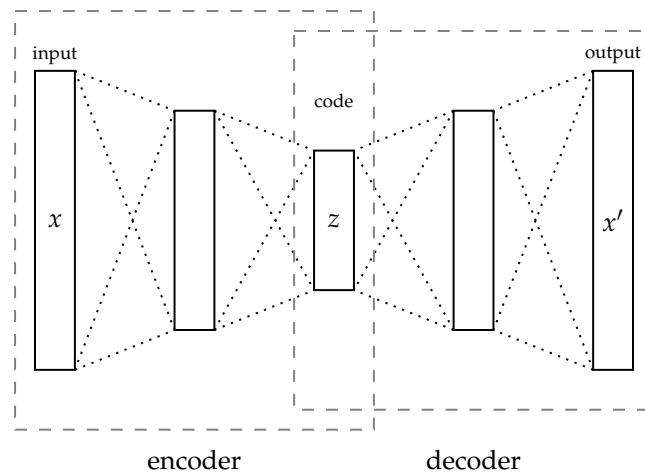
## 7

# Special Architectures

---

## 7.1 Autoencoders

An **autoencoder** is a neural network that learns to encode (and decode) a set of inputs. It's called an autoencoder because it learns the encoding automatically.



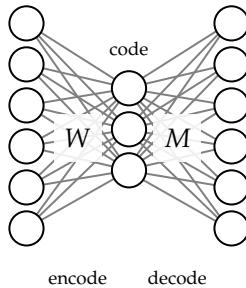
"Code" layer in the middle is often called *latent representation* or *embedding space*. Notice input and output are basically the same size. In the training process, we compare the output to the decoder and the input to the encoder. So our loss will be  $L(x', x)$ . Also note that the "code" layer is smaller than the input/output layers. Input and output are high dimension spaces, so there's a lot of information contained in the dataset. However, presumably, some of them are redundant. The "actual" information is actually lower dimensional. So the objective of an autoencoder is to squeeze the input and force the autoencoder to come up with a more efficient representation/representation. Therefore, they can be used to find efficient codes for high-dimensional data. For example, suppose I have the following dataset:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

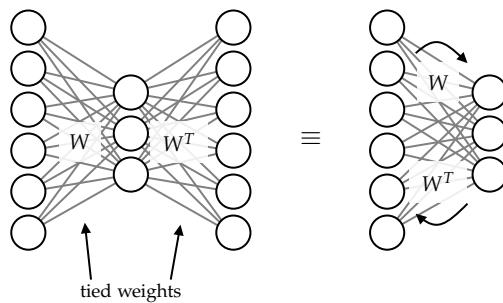
Here we have a bunch of binary strings, and we want to encode these strings more efficiently. Even though the vectors are 8-D (so could take on 256 different inputs), the actual dataset has only 5

patterns. We can, in principle, encode each of them with a unique 3-bit code. But we can choose the dimension of the encoding layer.

We can also think of our autoencoder is just 2 layers, and we can “unfold” it (or “unroll” it) to 3 layers, where the input layer and output layer are the same size, and have the same state. Instead of



We use



If we allow  $W$  and  $M$  to be different, then it's just a 3-layer (or more) network. However, if we enforce that  $M = W^T$ , then we say the weights are “tied”.

After training, we get the following 3-bit code.

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0.00 & 1.00 & 1.00 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1.00 & 0.00 & 0.00 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.00 & 0.00 & 0.00 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.00 & 1.00 & 0.00 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1.00 & 1.00 & 0.00 \end{bmatrix}$$

Suppose we encounter the input  $\begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$ . Can we figure out which 3-bit encoding it should have? Can our learned network handle such cases? Now let's switch to a demonstration of Python.

We are going to use PyTorch as usual.

```

1 import numpy as np
2 import torch
3 import torch.nn as nn
4 import matplotlib.pyplot as plt
5 from tqdm import tqdm

```

And we got this dataset:

```

1 A = torch.tensor([[1,0,1,0,0,1,1,0],
2                   [0,1,0,1,0,1,0,1],
3                   [0,1,1,0,1,0,0,1],
4                   [1,0,0,0,1,0,1,1],
5                   [1,0,0,1,0,1,0,1]], dtype=torch.float32)

```

```

1  class BinaryWorld(torch.utils.data.Dataset):
2      def __init__(self, A, noise=0.):
3          super().__init__()
4          self.x = A.clone() + torch.normal(torch.zeros_like(A))*noise
5          self.y = A.clone()
6
7      def __len__():
8          return len(self.x)
9
10     def __getitem__(self, idx):
11         return self.x[idx], self.y[idx]
12
13     def inputs(self):
14         return self.x
15
16     def targets(self):
17         return self.y

```

```

1 >>> train = BinaryWorld(A)
2 >>> train.targets()
3 tensor([[1., 0., 1., 0., 0., 1., 1., 0.],
4         [0., 1., 0., 1., 0., 1., 0., 1.],
5         [0., 1., 1., 0., 1., 0., 0., 1.],
6         [1., 0., 0., 0., 1., 0., 1., 1.],
7         [1., 0., 0., 1., 0., 1., 0., 1.]])

```

Now let's create a neural network model. Here is our autoencoder.

```

1  class AE(nn.Module):
2      def __init__(self, input_dim=8, latent_dim=3):
3          super().__init__()
4          self.encoder = nn.Sequential(
5              nn.Linear(input_dim, latent_dim),
6              nn.Sigmoid(),
7          )
8
9          self.decoder = nn.Sequential(
10             nn.Linear(latent_dim, input_dim),
11             nn.Sigmoid(),
12         )
13
14      def forward(self, x):
15          self.h = self.encoder(x)
16          return self.decoder(self.h)

```

Then we are going to use the standard learning functionality that we have been using all along:

```

1 # Let's wrap up the learning loop in a function
2 def learn(net, ds, epochs=5000):
3     x = ds.inputs()
4     t = ds.targets()
5     for epoch in tqdm(range(epochs)):

```

```

6     y = net(x)
7     loss = loss_fn(y, t)
8     optim.zero_grad()
9     loss.backward()
10    optim.step()
11    losses.append(loss.item())
12    plt.figure(figsize=(4,4))
13    plt.plot(losses);

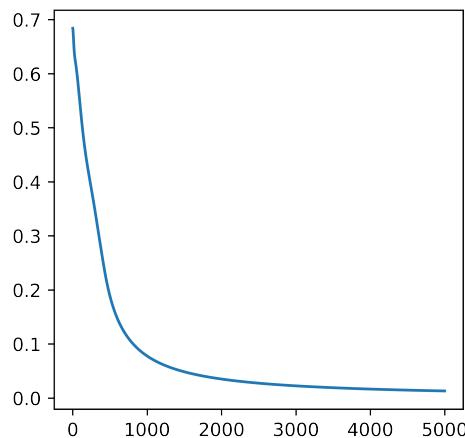
```

Then we train it:

```

1 net = AE(input_dim=8, latent_dim=3)
2 loss_fn = nn.BCELoss(reduction='mean')
3 optim = torch.optim.SGD(net.parameters(), lr=0.1, momentum=0.9)
4 losses = []
5
6 learn(net, train)

```



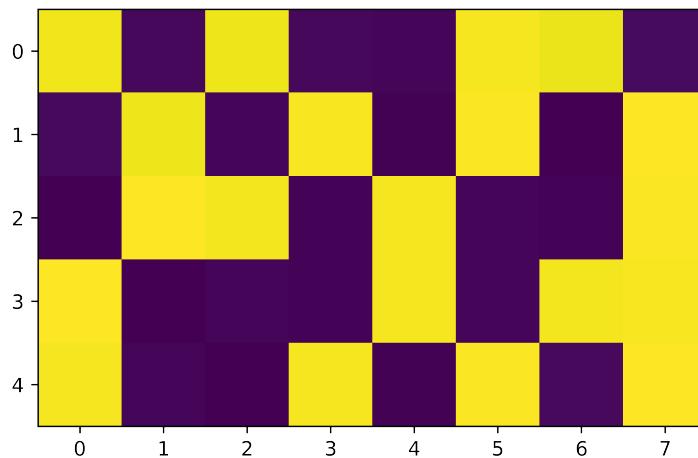
Now let's see how well it worked.

```

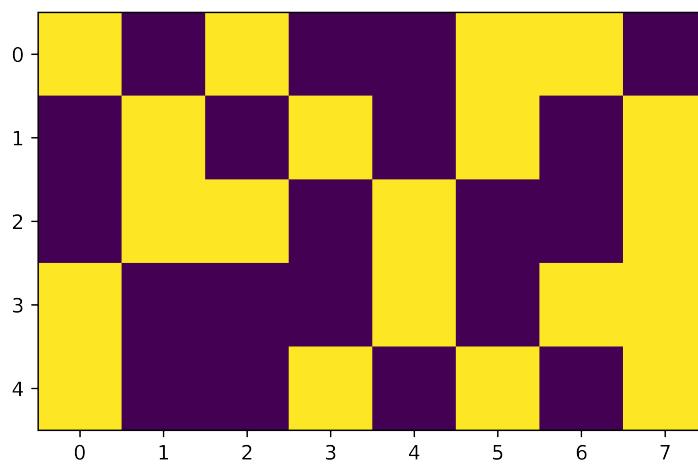
1 >>> y = net(train.inputs())
2 >>> print(torch.round(y*100.)/100)
3 tensor([[0.9800, 0.0200, 0.9700, 0.0200, 0.0200, 0.9900, 0.9700, 0.0300],
4         [0.0200, 0.9800, 0.0200, 0.9900, 0.0100, 0.9900, 0.0000, 1.0000],
5         [0.0000, 1.0000, 0.9800, 0.0100, 0.9900, 0.0100, 0.0100, 0.9900],
6         [1.0000, 0.0000, 0.0100, 0.0100, 0.9800, 0.0100, 0.9800, 0.9900],
7         [0.9900, 0.0100, 0.0000, 0.9900, 0.0100, 0.9900, 0.0200, 1.0000]],
8         grad_fn=<DivBackward0>)

```

This is running the inputs through my dataset and getting my outputs. Instead, we can take a look at its binary picture with `plt.imshow(y)`:



```
and plt.imshow(train.targets()):
```

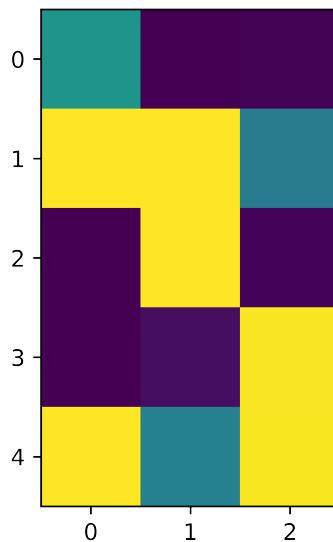


Each row represents one of the inputs, the binary strings. Autoencoder works because we see these two images are very similar.

What is the latent representation for each of the inputs?

```
1 >>> print(net.h)
2 tensor([[0.5172, 0.0010, 0.0061],
3         [0.9963, 0.9976, 0.4119],
4         [0.0024, 0.9959, 0.0119],
5         [0.0032, 0.0385, 0.9929],
6         [0.9971, 0.4381, 0.9867]], grad_fn=<SigmoidBackward>)
```

```
and plt.imshow(net.h):
```



Now we can see that the weights are not tied in this case:

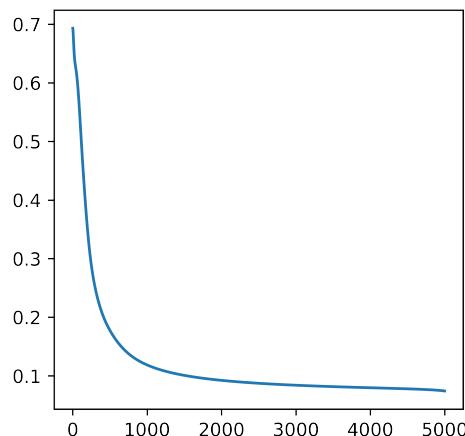
```

1 >>> net.encoder[0].weight
2 Parameter containing:
3 tensor([[ 0.1523, -0.0825, -1.5311,  3.1758, -3.3803,  3.5465, -1.4689, -0.4140],
4         [-3.0331,  3.2471, -0.3690,  1.5603,  0.7227, -0.6746, -2.8058,  1.9095],
5         [ 1.9931, -2.6727, -4.8877,  1.5643,  1.1234, -1.2633, -0.1961,  2.7582]], 
6         requires_grad=True)
7 >>> net.decoder[0].weight
8 Parameter containing:
9 tensor([[ 0.3821, -9.5563,   4.5215],
10        [-0.4445,   9.5668,  -4.4650],
11        [-3.9848,  -1.5246, -10.0008],
12        [ 7.8282,   3.3358,   3.0833],
13        [-10.4939,   2.9761,   2.8876],
14        [ 10.5077,  -3.0260,  -2.9358],
15        [ -3.8019,  -9.9677,  -1.0282],
16        [ -2.0190,   7.3370,   6.9832]], requires_grad=True)
```

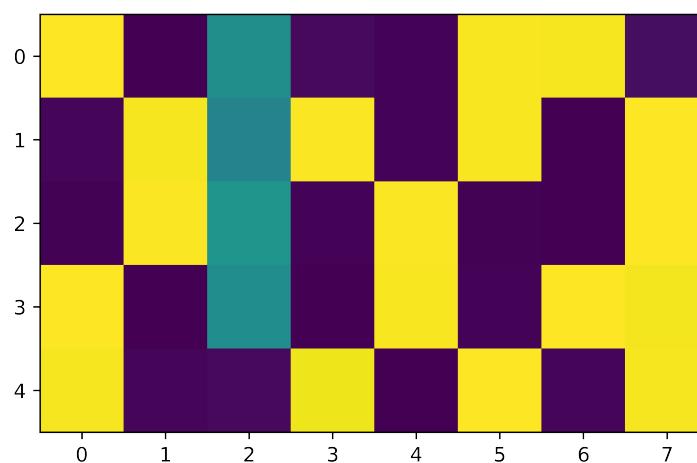
Now let's try with tied weights and train.

```

1 net = AE(input_dim=8, latent_dim=3)
2 loss_fcn = nn.BCELoss(reduction='mean')
3 optim = torch.optim.SGD(net.parameters(), lr=0.1, momentum=0.9)
4 losses = []
5 # Make both weight matrices point to the same tensor.
6 net.encoder[0].weight = nn.Parameter(net.decoder[0].weight.transpose(1,0))
7 learn(net, train)
```



```
and plt.imshow(net(train.inputs())):
```



Now if we take a look at the weights of encoder and decoder, we can see they are transpose of each other:

```

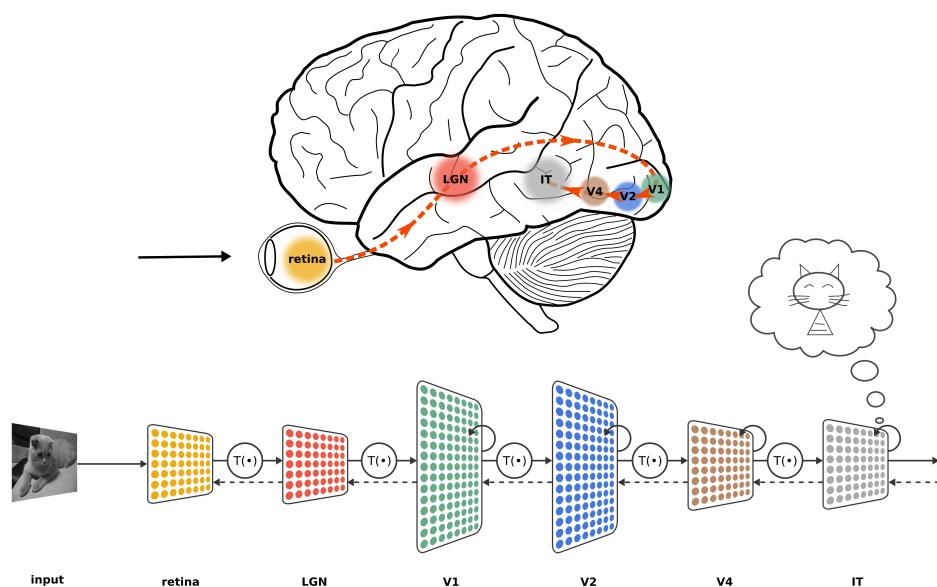
1 >>> net.encoder[0].weight
2 Parameter containing:
3 tensor([[ 8.5663, -8.6749, -3.4743, -1.1455, -2.3520,  2.0711,  3.9378, -2.5786],
4         [-3.6461,  3.3503, -3.6631,  7.3721, -2.3771,  2.0927, -8.4669,  7.4328],
5         [-1.3521,  1.2097, -0.0176, -8.7309,  9.3475, -9.3840,  1.7481,  7.2931]], 
6         requires_grad=True)
7 >>> net.decoder[0].weight
8 Parameter containing:
9 tensor([[ 8.5663, -3.6461, -1.3521],
10        [-8.6749,  3.3503,  1.2097],
11        [-3.4743, -3.6631, -0.0176],
12        [-1.1455,  7.3721, -8.7309],
13        [-2.3520, -2.3771,  9.3475],
14        [ 2.0711,  2.0927, -9.3840],
15        [ 3.9378, -8.4669,  1.7481],
16        [-2.5786,  7.4328,  7.2931]], requires_grad=True)
```

## 7.2 Your Visual System

Mammalian visual system, which is the basis for many neural-network vision systems. Most of the networks we have looked at assume an all-to-all connectivity between populations of neurons, like between layers in a network. But that's not the way our brains are wired, thankfully. If every one of your 86 billion neurons was connected to every other neuron, your head would have to be MUCH bigger. As an example of the wiring in your brain, here are some fascinating features of your visual system:

### It's Layered

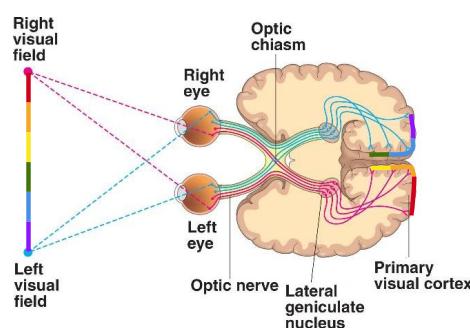
Although the details are more complicated, your visual system is roughly arranged into a hierarchy of layers.



Pic from Kubilius, Jonas (2017): Ventral visual stream. figshare. Figure. <https://doi.org/10.6084/m9.figshare.106794.v3>

### It's Topological

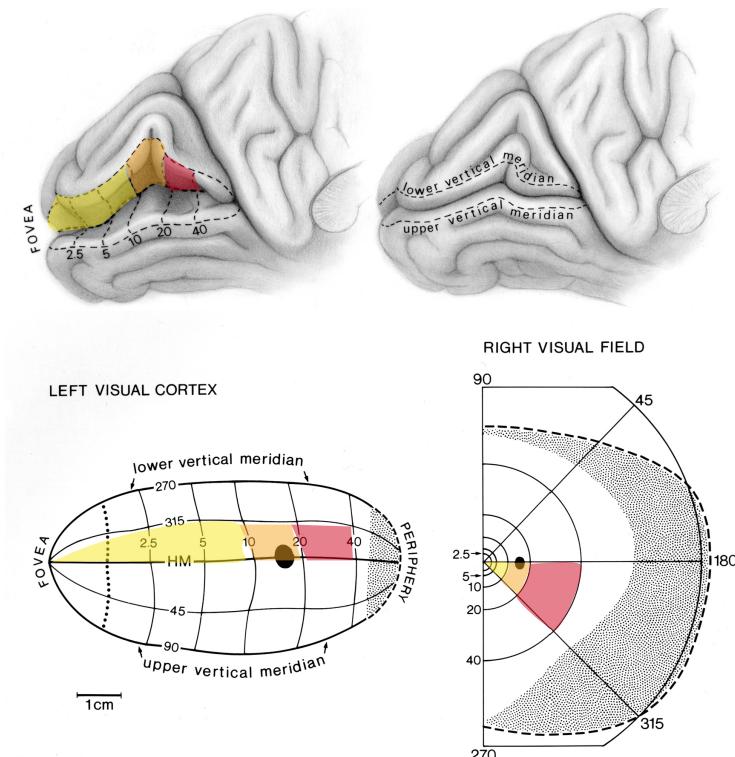
Topology, in general, means there's a continuous spatial mapping between two different spaces. Neurons close to each other in the primary visual cortex process parts of the visual scene that are close to each other. Let's discuss in a number of different ways.



<http://vision.ucsf.edu/hortonlab/ResearchProgram%20Pics/retinotopicMap.jpg>

This shows how the visual scene is spatially arranged. It is also spatially arranged on our cortex(大脑皮层). Half of each eyeball goes to one hemisphere and the other half goes to the opposite hemisphere. Both eyeballs project to both hemisphere.

Now let's look into its detail. The top left picture shows a part of the visual cortex, opened up. We can see the two black dots. This dot in the visual field only excites a small patch of neurons in V1. We can see that more resources concentrated in the center of the vision, and fewer on periphery.



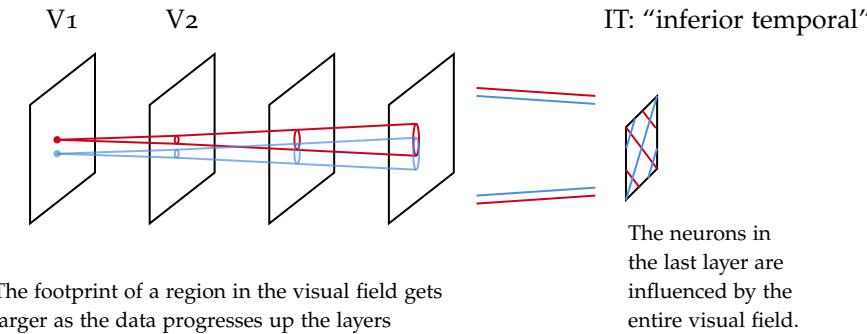
<http://vision.ucsf.edu/hortonlab/ResearchProgram%20Pics/retinotopicMap.jpg>

Here is a visual field. Each neuron in V1 is only activated by a small patch in the visual field.

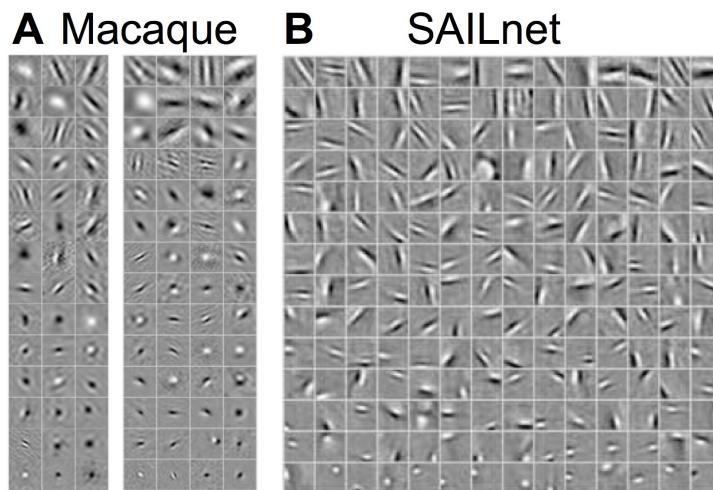


Conversely, each patch in the visual field excites only a small neighbourhood of neurons in V1. This topological mapping between the visual field and the surface of the cortex is called a retinotopic mapping. Moreover, neurons in V1 project to the next layer, V2, and again, the connections are

retinotopically local.



Finally, the visual system is a bit like a filter bank. In the lower levels of the hierarchy, the neurons seem to respond to standard patterns of input. Each little square corresponds to one V1 neuron, and shows the pattern that most activates that neuron. The picture below is a bunch of receptive fields: a configuration for the image that excites a particular neuron.



Derived from: Zylberberg, Joel; Timothy Murphy, Jason; Robert DeWeese, Michael (2015): SAILnet learns receptive fields (RFs) with the same diversity of shapes as those of simple cells in macaque primary visual cortex (V1).. PLOS Computational Biology. Figure. <https://doi.org/10.1371/journal.pcbi.1002250.g003>

### 7.3 Convolutional Neural Networks

In CNN, we can take advantage of some of our visual system's features in artificial neural networks. Inspired by the brain's topological (retinotopic) connectivity, and in an effort to reduce the number of connection weights that need to be learned, scientists devised the Convolutional Neural Network.

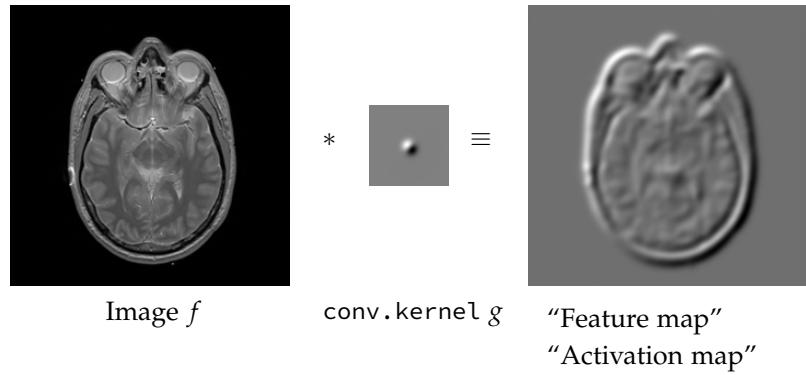
Let's first review convolution. In a continuous domain,  $f, g : \mathbb{R} \rightarrow \mathbb{R}$  convolution is

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x) \cdot g(x-s) ds$$

In a discrete domain,  $f, g \in \mathbb{R}^N$ , then

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

How does it look like in images?



In 2D, discrete convolution is

$$(f * g)_{m,n} = \sum_{i,j} f_{ij} \cdot g_{m-i,n-j},$$

or

$$(f \circledast g)_{m,n} = \sum_{i,j} f_{ij} \cdot g_{i-m,j-n}.$$

For the rest of this chapter, please check [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture5.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf). The original lecture video is available at <https://youtu.be/bNb2fEVKeEo>. The detailed notes is available at <https://cs231n.github.io/convolutional-networks/>.

# 8

## Hopfield Networks

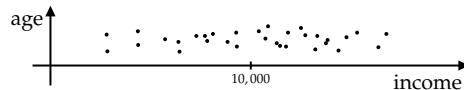
---

### 8.1 Batch Normalization

Consider a dataset in which you are trying to estimate if someone is a vegetarian from their age and income.

Age (years)	Income (\$)	Veg?
27	31,000	yes
52	126,000	no
16	10,000	no
	:	

Then we can plot it:

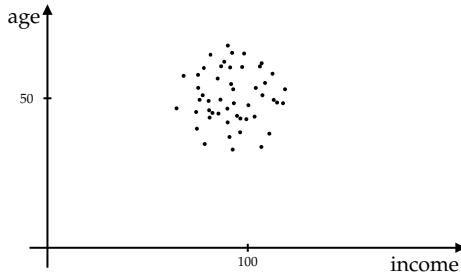


We can see from the plot that vast majority of variance is along the income axis. Then if we have these two inputs, age and income, to neural network, and connection weights  $W$ . Of course, the connection weights can accommodate the differences in scale: the weights coming from the income are small, and weights coming from the age are bigger. But, to do so will result in weights with vastly different scales. These weights are spanning 3 orders of magnitude, in our example. This forces us to use a small learning rate.

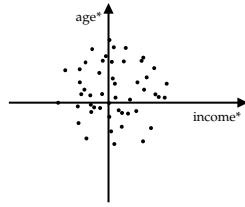
Alternatively, we can rescale our data. Before feeding this data into our network, we might be inclined to rescale the inputs:

Age (years)	Income (\$1k)	Veg?
27	31	yes
52	126	no
16	10	no
	:	

and the plot

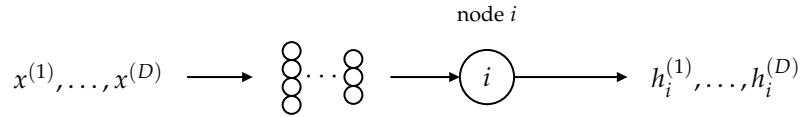


Why don't we rescale our feature inputs so that they are centered at zero? In the plot below, \* means rescaled. In particular, we aim for zero mean and unit variance along each axis.



This is the idea behind a method called **Batch Normalization**.

Consider a mini-batch of  $D$  samples,  $\{(x^{(1)}, t^{(1)}), \dots, (x^{(D)}, t^{(D)})\}$ , and the corresponding outputs of a layer,  $\{h^{(1)}, \dots, h^{(D)}\}$  where  $h^{(d)} = [h_1^{(d)} \dots h_N^{(d)}]$ . Now consider a single node  $i$  in that layer,



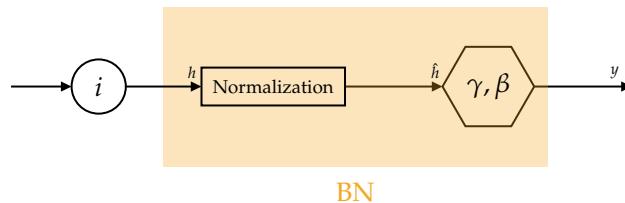
We want to normalize that batch of outputs so that its mean is 0 and its variance (and hence standard deviation) is 1:

$$\mu_i = \frac{1}{D} \sum_{d=1}^D h_i^{(d)}, \quad \sigma_i^2 = \frac{1}{D} \sum_{d=1}^D (h_i^{(d)} - \mu_i)^2.$$

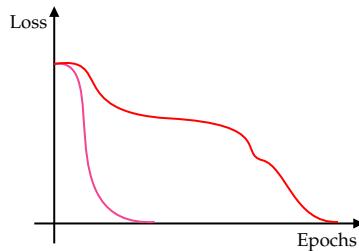
Then we rescale those inputs to normalize the batch,

$$\hat{h}_i^{(d)} = \frac{h_i^{(d)} - \mu_i}{\sigma_i} \quad \text{or} \quad \hat{h}_i^{(d)} = \frac{h_i^{(d)} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \text{ for some small } \epsilon > 0.$$

Finally, we rescale that output with learnable parameters  $\gamma_i, \beta_i$ :  $y_i^{(d)} = \gamma_i \hat{h}_i^{(d)} + \beta_i$ . We can depict as follows:



Batch normalization has a profound affect on the rate of convergence for learning. In the picture below, we compare **normal learning** and **batch normalization**:



Why does it work? It's debated, but here are some ideas. It helps mitigate vanishing/exploding gradients: If we rescale our data, it will help avoid bizarrely large output currents coming from a particular neuron.

It helps guard against internal covariate shift. What does this mean? Imagine we have a deep neural networks, where we have two kinds of layers: deep layers (close to the input layer) and shallow layers (or top layers, close to the output). Gradients tends to be bigger towards the output size of the network, thus shallow layers tend to learn more quickly. However, we have a problem. Shallow layers come up with a model after learning quickly, but over time, deep layers start to change or shift, which invalidates the solution established by the shallow layers. Then batch normalization would stop the deep layers from having large shifts.

## 8.2 Hopfield Networks

### 8.2.1 Content-Addressable Memory

First let's fill in the blanks:

intelligent      irreplaceable      123456789

Here we took a partial pattern and we filled in with the most likely completion with the help of our memory. Because these are patterns you have in memory, you can fill in the missing pieces. In fact, you can also detect errors:

1 2 3 8 5 6 7 8 9	nueroscience	Vaterloo
↑	↑	↑
4	eu	W

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

In 1982, John Hopfield published a famous paper:

*Proc. Natl. Acad. Sci. USA*  
Vol. 79, pp. 2554–2558, April 1982  
Biophysics

### Neural networks and physical systems with emergent collective computational abilities

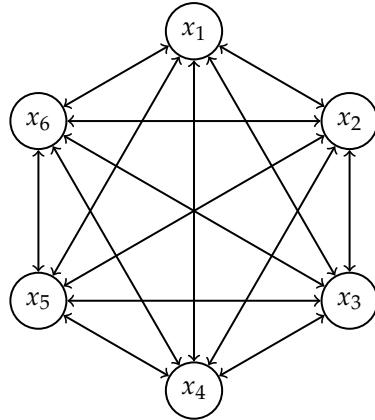
(associative memory/parallel processing/categorization/content-addressable memory/fail-soft devices)

J. J. HOPFIELD

Division of Chemistry and Biology, California Institute of Technology, Pasadena, California 91125, and Bell Laboratories, Murray Hill, New Jersey 07974

Contributed by John J. Hopfield, January 15, 1982

In it, Hopfield proposed a method for using a neural network as a CAM. The network learns the patterns, and converges to the closest pattern when shown a partial pattern.



$W_{ij}$  is the connection strength from node  $i$  to node  $j$ . Assume  $W_{ij} = W_{ji}$ . Each node in the network can be a  $-1$  or a  $1$ , <sup>1</sup>  $x_j \in \{-1, 1\}, j = 1, \dots, N$ . Or we can think of it as a binary string of length  $N$ . Suppose each node wants to change its state so that

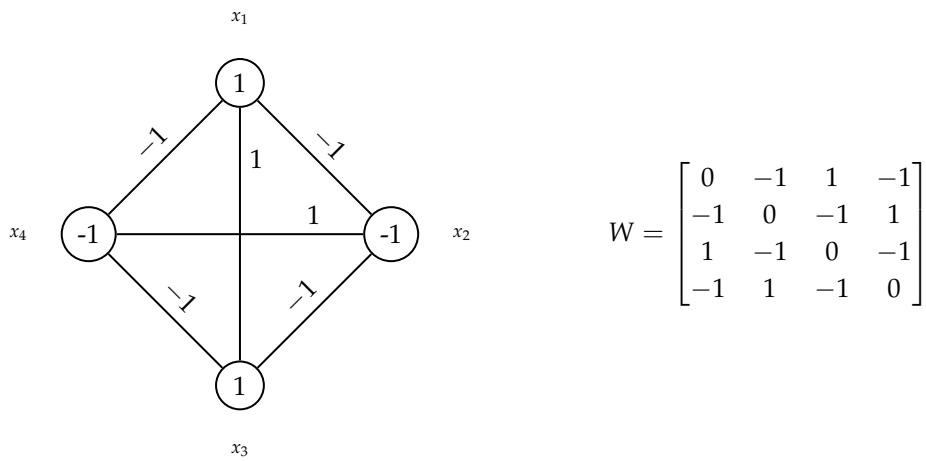
$$x_j = \begin{cases} -1 & \text{if } \sum_{i \neq j} x_i W_{ij} < -b_j \\ 1 & \text{otherwise.} \end{cases}$$

For the rest of this section, we assume  $b_j = 0$  for now.

If we have a pattern that we would like the network to recall, we could set the weights such that:

$$W_{ij} \begin{cases} > 0 & \text{between any 2 nodes in same state,} \\ < 0 & \text{between any 2 nodes that differ.} \end{cases}$$

For example,



How can we find the connection matrix  $W$  that works for the set of “memories” we want to encode? Hopfield’s answer is as follows: Given  $M$  target network states  $\{x^{(1)}, \dots, x^{(M)}\}$ , then we choose connection weights

$$W_{ij} = \frac{1}{M} \sum_{s=1}^M x_i^{(s)} x_j^{(s)} \quad \text{for } i \neq j,$$

Here  $W_{ij}$  is the average co-activation between nodes  $i$  and  $j$ . In particular,

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

---

<sup>1</sup>there’s a equivalent derivation for  $x_j \in \{0, 1\}$

Notice,

$$\begin{aligned} W &= \frac{1}{M} \sum_{s=1}^M x^{(s)} (x^{(s)})^T - I \\ &= \frac{1}{M} \sum_s \boxed{\quad} - I \\ &= \frac{1}{M} \left[ \underbrace{\boxed{\quad} + \boxed{\quad} + \cdots + \boxed{\quad}}_{M \text{ rank 1 matrices}} \right] - I \end{aligned}$$

This method works best if the network states,  $\{x^{(1)}, \dots, x^{(M)}\}$ , are all mutually orthogonal.

Hopfield recognized a link between these network states and the Ising model in physics. It has to do with a lattice of interacting magnetic dipoles. Each dipole can be “up” or “down”. Which state it’s in depends on its neighbours. Just like the Ising model, we can write the energy of the system using a Hamiltonian function.

For our neural network, assuming  $W$  is symmetrical, and  $W_{ii} = 0$  for all  $i$ , then

$$\begin{aligned} E &= -\frac{1}{2} \sum_{j \neq i} x_i W_{ij} x_j - \sum_j b_j x_j \\ &= -\frac{1}{2} x^T W x - b^T \end{aligned}$$

is called **Hopfield Energy**.

The gradient is

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

Using it for gradient descent,<sup>2</sup>

$$\frac{dx_j}{dt} = \kappa \left( \sum_{j \neq i} x_i W_{ij} + b_j \right),$$

which agrees with

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

Similarly, we have

$$\frac{\partial E}{\partial W_{ij}} = -\frac{1}{2} x_i x_j , \quad \frac{\partial E}{\partial b_j} = -x_j ,$$

and

$$\frac{dW_{ij}}{dt} = \kappa x_i x_j , \quad i \neq j , \quad \frac{db_j}{dt} = \kappa x_j .$$

Hopfield energy drives a lot of the theory and the practice behind unsupervised learning. We don’t have an output so we can’t compute a loss and therefore we can’t compute gradients to update our connection weights and biases. Instead, we come up with this idea of energy and we want to minimize energy, giving us something to optimize. Now we can start changing our network state, our connection weights and biases to minimize energy.

### 8.3 Hopfield Demonstration

First we have preamble:

---

<sup>2</sup>This is another representation of update rule:  $x_j \leftarrow x_j - t(\dots)$ .

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 plt.rcParams['figure.dpi'] = 600 # this ensures the plot is in high resolution
4 import copy

```

and some helper functions:

```

# Useful functions
def IsScalar(x):
    if type(x) in (list, np.ndarray,):
        return False
    else:
        return True

def Thresh(x):
    if IsScalar(x):
        val = 1 if x>0 else -1
    else:
        val = np.ones_like(x)
        val[x<0] = -1.
    return val

def Hamming(x, y):
    """
        d = Hamming(x,y)

        Hamming distance between two binary vectors x and y.
        It's the number of digits that differ.

    Inputs:
        x and y are arrays of binary vectors, and can be either {0,1} or {-1,1}

    Output:
        d is the number of places where the inputs differ
    """
    d = []
    for xx, yy in zip(x,y):
        dd = 0.
        for xxx,yyy in zip(xx,yy):
            if xxx==1 and yyy!=1:
                dd += 1.
            elif yyy==1 and xxx!=1:
                dd += 1.
        d.append(dd)
    return d

def Perturb(x, p=0.1):
    """
        y = Perturb(x, p=0.1)

        Apply binary noise to x. With probability p, each bit will be randomly
    """

```

```

45     set to -1 or 1.
46
47     Inputs:
48         x is an array of binary vectors of {-1,1}
49         p is the probability of each bit being randomly flipped
50
51     Output:
52         y is an array of binary vectors of {-1,1}
53         ...
54
55     y = copy.deepcopy(x)
56     for yy in y:
57         for k in range(len(yy)):
58             if np.random.rand()<p:
59                 yy[k] = Thresh(np.random.randint(2)*2-1)
60
61     return y

```

### 8.3.1 Toy Dataset

Here is a toy dataset:

```

1 # 4 Orthogonal Vectors
2 X = np.array([[ 1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1, -1, -1, -1, -1],
3               [ 1,  1,  1,  1, -1, -1, -1,  1,  1,  1, -1, -1, -1, -1, -1, -1],
4               [ 1, -1,  1, -1,  1, -1,  1, -1,  1, -1,  1, -1,  1, -1,  1, -1],
5               [ 1,  1, -1, -1,  1,  1, -1, -1,  1,  1, -1, -1,  1,  1, -1, -1],
6               ]])
7 N = np.shape(X)[1]
8 n = 4

```

and  $X.shape$  is  $(4, 16)$ . Or we can use random dataset (not used in the demo):

```

1 # Random
2 N = 50
3 n = 6
4 X = Thresh(np.random.normal(size=(n,N)))

```

Then we compute connection weights and biases:

```

1 b = np.zeros((1,N))
2 b = np.sum(X, axis=0) / n    # the avg activity throughout those different memories
3 W = ( X.T @ X ) / n - np.eye(N)
4 W0 = copy.deepcopy(W)

```

There's another way to compute  $W$ :

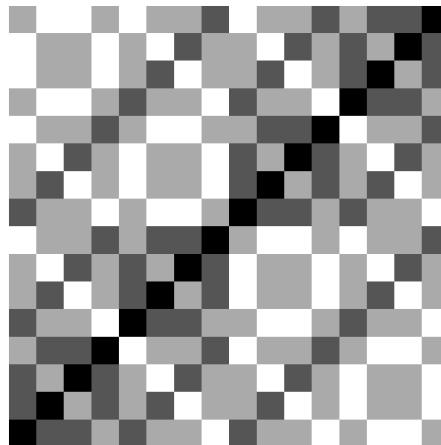
```

1 W1 = np.zeros_like(W)
2 for x in X:
3     W1 += np.outer(x, x)/n
4 W1 -= np.identity(N)
5
6 print(np.max(W.flatten()-W1.flatten())) # gives 0.0

```

Then we can take a look at the connection weights matrix via

```
1 plt.subplot(121); plt.imshow(W, cmap='gray'); plt.axis('off');
```



It's  $16 \times 16$ , and a bunch of zeros along diagonals.

Note that  $X @ W$  returns a multiple of  $X$ :

```
1 >>> print(X@W) # just change the length of vector, not the direction
2 [[ 3.  3.  3.  3.  3.  3.  3. -3. -3. -3. -3. -3. -3. -3. -3.]
3 [ 3.  3.  3.  3. -3. -3. -3.  3.  3.  3.  3. -3. -3. -3. -3.]
4 [ 3. -3.  3. -3.  3. -3.  3. -3.  3. -3.  3. -3.  3. -3.]
5 [ 3.  3. -3. -3.  3.  3. -3. -3.  3.  3. -3. -3.  3.  3. -3.]]
```

Then we define energy function and update function:

```
1 def Energy(W, b, X):
2     E = []
3     for xx in X:
4         blah = - 0.5 * (xx @ W)@xx.T + b@xx.T
5         E.append(blah)
6     return E
7
8 def Update(W, x, b):
9     xnew = x @ W - b
10    return Thresh(xnew)
```

Let's first generate a sample and set initial network state:

```
1 k = np.random.randint(len(X)) # randomness here
2 Y = Perturb( X , p=0.4 )
3 x = Y[k:k+1,:]
4 err = Hamming(x, X[k:k+1,:])
5 print('Class '+str(k)+ ' with '+str(err)+ ' errors') # Class 1 with [1.0] errors
6 x_orig = copy.deepcopy(x)
7 print(x_orig) # [[ 1  1  1  1 -1 -1 -1 -1  1 -1  1  1 -1 -1 -1 -1]]
8
9 # Notice how the inner product of the perturbed vector is almost
10 # orthogonal to all the other equilibria. i.e., relatively close to zero
11 print('Inner Products: '+str(x@X.T)) # Inner Products: [[ 2 14  2 -2]]
```

Now we iterate in our network: set network state to that corrupted sample and iterate our network until it settles down to some equilibrium.

```

1 if True:      # a version
2     x = copy.deepcopy(x_orig)
3     n_iters = 10
4     for k in range(n_iters):
5         #print(x)
6         x_new = Update(W, x, b)
7         print(Hamming(x, x_new))
8         x = x_new
9 else:          # another version
10    x = copy.deepcopy(x_orig)
11    n_iters = 40
12    for count in range(n_iters):
13        node_idx = list(range(N))
14        np.random.shuffle(node_idx)
15        for k in node_idx:
16            ic = x@W[:,k] - b[k]
17            x[0,k] = Thresh(ic)

```

This gives us [1.0] [0.0] ... [0.0]. We fix it during the iterations and it remains unchanged.

How close are we to recalling a memory?

```

1 for idx,t in enumerate(X):
2     d = Hamming(x, [t])[0]
3     print('Memory '+str(idx)+' has error '+str(d))
4 # Memory 1 has error 0.0

```

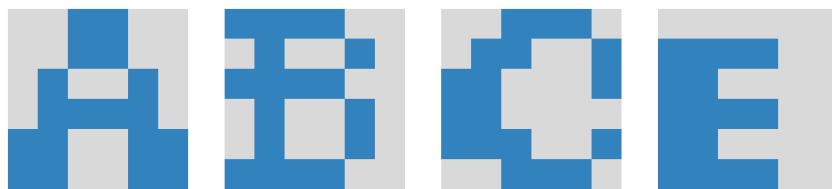
### 8.3.2 Letters Example

Let's look at a more interesting dataset.

```

1 from imageio import imread
2
3 letters = []
4 letters.append(imread('images/A.png'))
5 letters.append(imread('images/B.png'))
6 letters.append(imread('images/C.png'))
7 letters.append(imread('images/E.png'))
8 n = len(letters)
9 N = len(letters[0].flatten())
10 X = np.zeros((n, N))
11 for idx,img in enumerate(letters):
12     X[idx,:] = Thresh(np.array([img.flatten()-0.5]))
13
14 plt.figure(figsize=(16,4))
15 for k in range(n):
16     plt.subplot(1,n,k+1);
17     plt.imshow(np.reshape(X[k], (6,6)), cmap='tab20c_r'); plt.axis('off');

```



We can see that they are not totally orthogonal:

```

1 >>> X@X.T
2 array([[36.,  0.,  0.,  8.],
3        [ 0., 36.,  0.,  4.],
4        [ 0.,  0., 36.,  8.],
5        [ 8.,  4.,  8., 36.]])

```

We then compute connection weights and biases:

```

1 b = np.zeros((1,N))
2 b = np.sum(X, axis=0) / n
3 W = ( X.T @ X ) / n - np.eye(N)
4 W0 = copy.deepcopy(W)

```

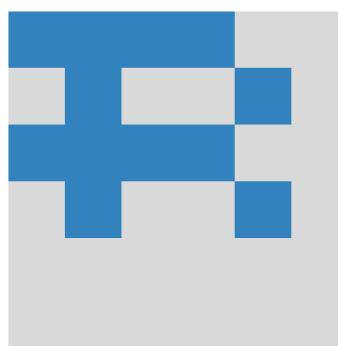
Now let's create a corrupted version of one of those images:

```

1 k = np.random.randint(n)
2 Y = Perturb( X , p=0.2)
3 x = Y[k:k+1,:]
4 x[0,24:] = -1.
5 err = Hamming(x, X[k:k+1,:])
6 print('Class '+str(k)+ ' with '+str(err)+' errors')
7 x_orig = copy.deepcopy(x)
8 #plt.imshow(np.reshape(x,[28,28]), cmap='gray'); plt.axis('off');
9 plt.imshow(np.reshape(x,[6,6]), cmap='tab20c_r'); plt.axis('off');

```

Class 1 with [6.0] errors



Let's see if we can reconstruct that. We present two methods:

```

1 xs = copy.deepcopy(x_orig)
2 xa = copy.deepcopy(x)
3 # Synchronous updating, described in previous sections
4 n_iters = 2
5 for idx in range(n_iters):
6     xs = Update(W, xs, b)

```

```

7 # Asynchronous updating: randomly select one neuron then update it
8 n_iters = 10
9 for count in range(n_iters):
10     node_idx = list(range(N))
11     np.random.shuffle(node_idx)
12     for idx in node_idx:
13         ic = xa@W[:,idx] - b[idx]
14         xa[0,idx] = Thresh(ic)

```

So these two methods both work in this case:

```

1 print('Correct class is '+str(k))
2 print('Synchronous updating')
3 for idx,t in enumerate(X):
4     ds = Hamming(xs, [t])[0]
5     print('Memory '+str(idx)+' has error '+str(ds))
6 print('Asynchronous updating')
7 for idx,t in enumerate(X):
8     da = Hamming(xa, [t])[0]
9     print('Memory '+str(idx)+' has error '+str(da))

```

```

Correct class is 1
Synchronous updating
Memory 0 has error 18.0
Memory 1 has error 0.0
Memory 2 has error 18.0
Memory 3 has error 16.0
Asynchronous updating
Memory 0 has error 18.0
Memory 1 has error 0.0
Memory 2 has error 18.0
Memory 3 has error 16.0

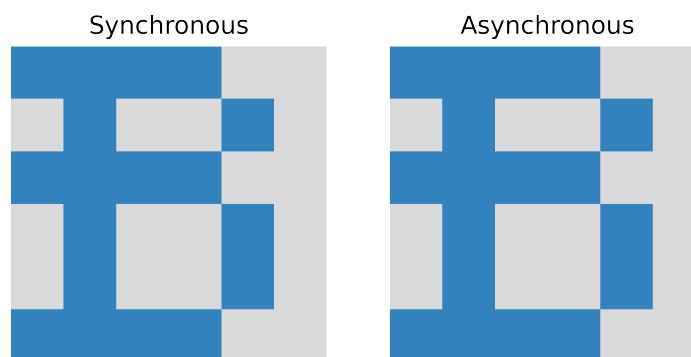
```

We can then plot:

```

1 plt.subplot(1,2,1); plt.imshow(np.reshape(xs,[6,6]), cmap='tab20c_r'); plt.title('
2   Synchronous'); plt.axis('off');
2 plt.subplot(1,2,2); plt.imshow(np.reshape(xa,[6,6]), cmap='tab20c_r'); plt.title('
3   Asynchronous'); plt.axis('off');

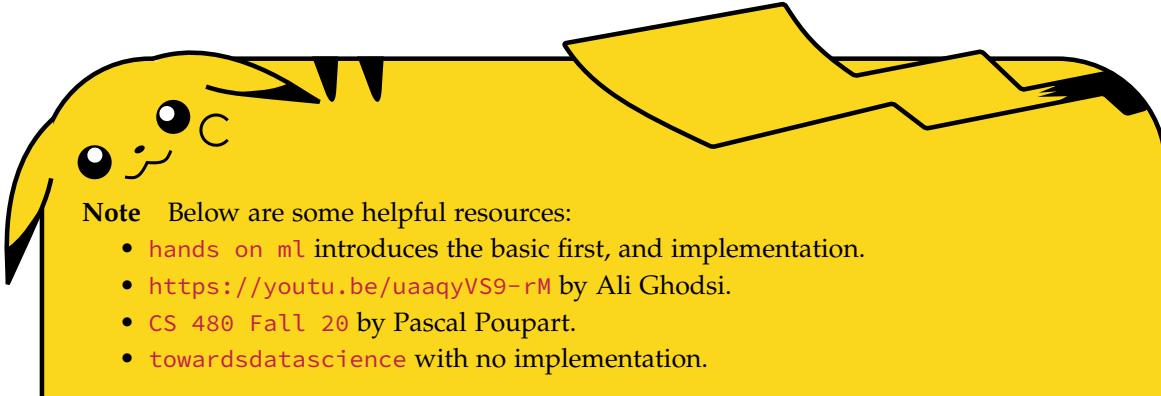
```



# 9

## VAE

### 9.1 Helpful Resources



**Note** Below are some helpful resources:

- [hands on ml](#) introduces the basic first, and implementation.
- <https://youtu.be/uuaqyVS9-rM> by Ali Ghodsi.
- [CS 480 Fall 20](#) by Pascal Poupart.
- [towardsdatascience](#) with no implementation.

### 9.2 Variational Autoencoders

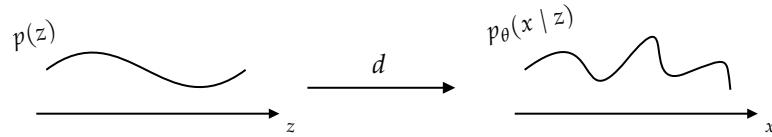
We would like to be able to reconstruct the samples in our dataset. In fact, we would like to be able to generate ANY valid sample. In essence, we would like to sample the distribution  $p(x)$ , the distribution of inputs.<sup>1</sup> We generate samples by choosing elements from some lower-dimensional latent space,  $z \sim p(z)$ . For example,  $z$  could represent digit class, line thickness, slant, etc... which are features specifying particular samples.

Then from those latent representations, we generate samples:

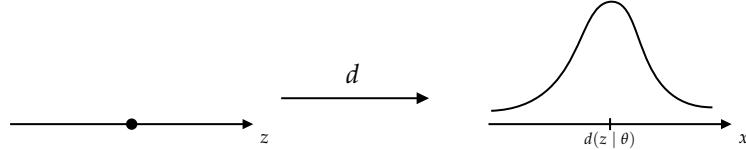
$$\text{Diagram: } z \xrightarrow[p_\theta(x|z)]{x \approx d(z, \theta)} x \quad p(x) = \int p_\theta(x|z)p(z) dz$$

Note that running the random variable  $z$  through  $d$  gives a distribution  $p(z|z)$ .

<sup>1</sup> $p(x)$  kinda has two purposes. First, given an  $x$ , it acts like a function which tells us the probability or probability density of  $x$ . Second, we can sample it: here is the distribution, please give us a sample that reflects the underlying distribution  $p(x)$ .



But, even for a fixed  $z$ , we assume  $p(x | z)$  is a distribution.



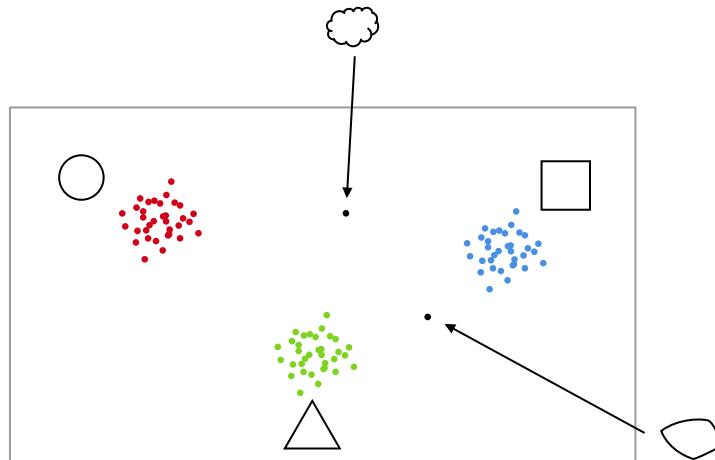
We have a dataset of samples,  $X$ , and we want to find  $\theta$  to maximize the likelihood of observing  $X$ .

If we assume  $p_\theta(x | z)$  is Gaussian, with  $\mu = d(z, \theta)$  and  $\sigma$ , then

$$-\ln p_\theta(X | z) = \frac{1}{2\sigma^2} \|X - d(z, \theta)\|^2 + C$$

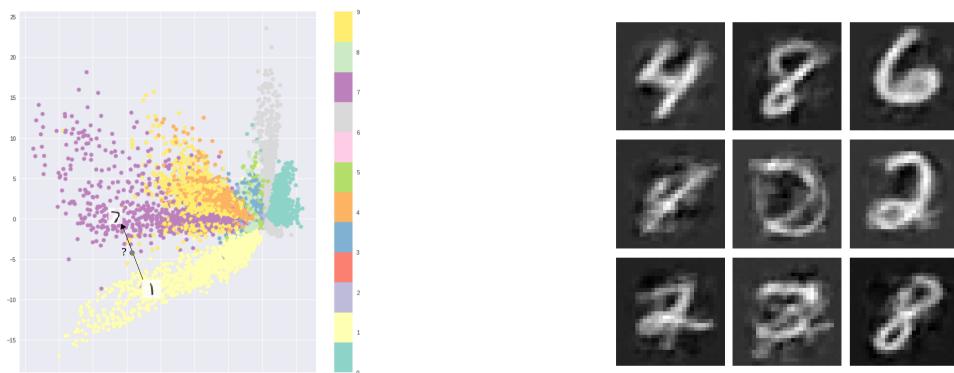
So, given samples  $z$ , we have a way to learn  $d(z, \theta)$  to maximize  $\mathbb{E}_{z \sim p(z)}[p_\theta(x | z)]$ .  $d(z, \theta)$  is our decoder, producing  $\hat{x}$ . The problem is that we don't know how to  $z \sim p(z)$ .

Let's try to illustrate this. Suppose we train an AE on a dataset of simple shapes:  $\circlearrowleft$ ,  $\triangle$ ,  $\square$ .



The latent space is 2D, and the clusters are well separated. However, latent vectors between the clusters generate samples that don't look like our training samples.

Another example would be MNIST. Below is MNIST 2D latent space, picture taken from [towardsdsds](#). The picture on the right is the result after we apply decoder to those “in-between” points.



What's happening? Why is our generator so bad? This is because we are choosing improbable  $z$  samples, where  $p(z) \approx 0$ .

We would like to sample only  $z$ 's that yield reasonable samples with high probability. We are now placing requirements on the distribution in our latent space. Can we get away with this?

Let's assume that we can choose the distribution of  $z$ 's in the latent space; call it  $q(z)$ . Then

$$\begin{aligned} p(x) &= \mathbb{E}_{z \sim p}[p(x | z)] \\ &= \sum_{z \sim p} p(x | z)p(z) \\ &\stackrel{\sim}{=} \sum_{z \sim q} p(x | z) \frac{p(z)}{q(z)} q(z) \\ &= \mathbb{E}_{z \sim q} \left[ p(x | z) \frac{p(z)}{q(z)} \right] \end{aligned}$$

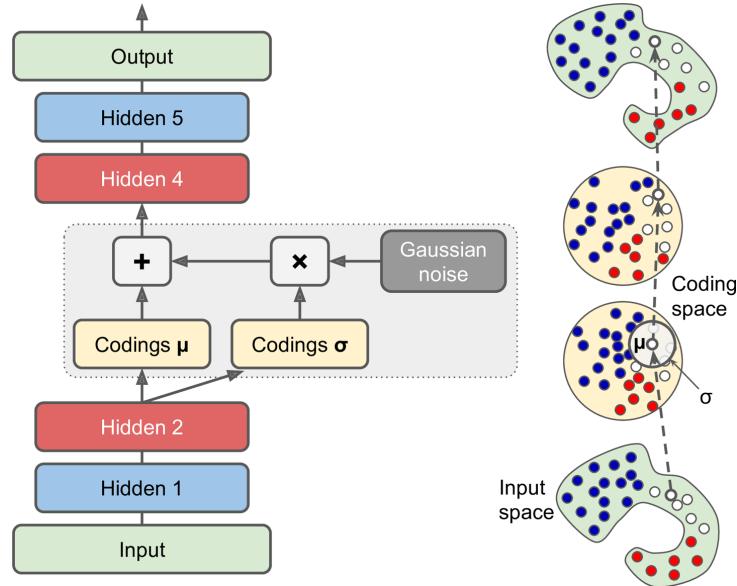
Then negative log likelihood is

$$\begin{aligned} -\ln p(x) &= -\mathbb{E}_q \left[ \ln p(x | z) + \ln \frac{p(z)}{q(z)} \right] \\ &= -\underbrace{\mathbb{E}_q [\ln p(x | z)]}_{(2)} + \underbrace{KL(q(z) \| p(z))}_{(1)} \end{aligned}$$

where  $KL(P \| Q) = -\sum_i P(i) \ln \frac{Q(i)}{P(i)}$  is Kullback–Leibler divergence between two distributions.

First consider (1). Let's choose a latent distribution that is convenient for us:  $p(z) \sim \mathcal{N}(0, I)$ . Then our aim is to design  $q(z)$  so that it is close to  $\mathcal{N}(0, I)$ , i.e.,  $\min KL(q(z) \| \mathcal{N}(0, I))$ .

How do we design our latent representations to achieve this? The answer is we design an encoder, and ask its outputs to be  $\mathcal{N}(0, I)$ . And we keep pressuring the encoder to give us  $\mu = 0$  and  $\sigma = I$ .



Picture taken from [hands-on-ml](#).

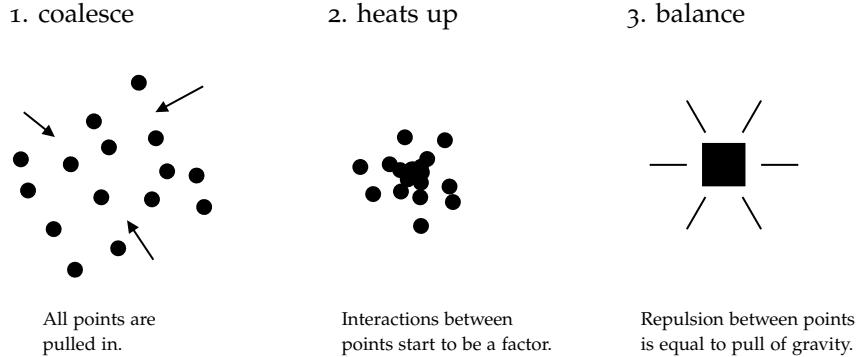
These Gaussians are convenient because there is a closed-form expression for

$$KL(\mathcal{N}(\mu, \sigma) \| \mathcal{N}(0, I)) = \frac{1}{2}(\sigma^2 + \mu^2 - \ln \sigma^2 - 1).$$

We want to minimize this, but there are other forces at play.

Now consider (2). The other term in the objective,  $\mathbb{E}_q[\ln p(x | z)]$ , is our reconstruction loss, which can be written as  $\mathbb{E}_q[\ln p(x | \hat{x})]$ , where  $z = \mu(x, \theta) + \epsilon\sigma(x, \theta)$ ,  $\epsilon \sim \mathcal{N}(0, I)$  and  $\hat{x} = d(\mu(x, \theta) + \epsilon\sigma(x, \theta), \theta)$ .  $\mu(x, \theta)$  and  $\sigma(x, \theta)$  is the encoder part as shown in the picture. This is a deterministic decoder.

Let's have some intuition. Think of a cloud of matter floating in space, but collapsing in by its own gravity,<sup>2</sup> but eventually forming a star.

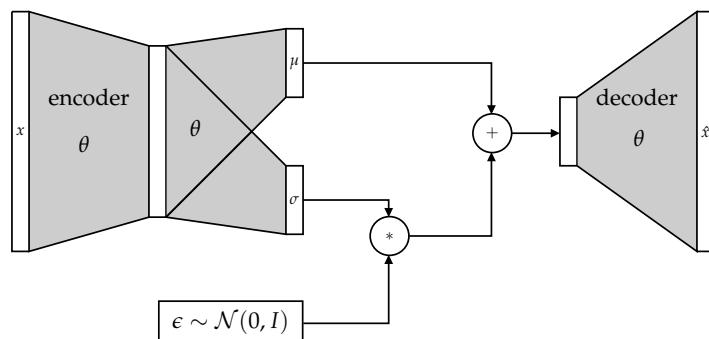


Here is the process for VAEs: start with  $\theta$ ,

- encode  $x$  by computing  $\mu(x, \theta)$  and  $\sigma(x, \theta)$  using neural networks.
  - sample  $z = \mu + \epsilon\sigma$ ,  $\epsilon \sim \mathcal{N}(0, I)$ .
  - calculate KL loss  $= \frac{1}{2}(\sigma^2 + \mu^2 - \ln \sigma^2 - 1)$ .
  - decode  $\hat{x}$  using another neural network  $\hat{x} = f(x, \theta) = d(\mu(x, \theta) + \epsilon\sigma(x, \theta))$ .
  - calculate reconstruction loss  $L(x, \hat{x})$ .
- E.g.,  $\frac{1}{2}\|x - \hat{x}\|^2$  for Gaussian  $p(x | \hat{x})$ . Or  $\sum x \ln \hat{x}$  for Bernoulli  $p(x | \hat{x})$ .
- both terms of our objective function are differentiable with respect to  $\theta$ :

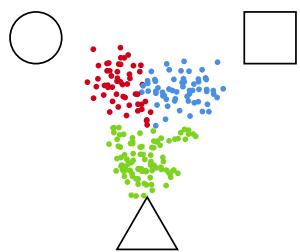
$$E = \mathbb{E}_x[L(x, \hat{x})] + \beta(\sigma^2 + \mu^2 - \ln \sigma^2 - 1)$$

where  $\hat{x}, \sigma, \mu$  all depend on  $\theta$ ,  $\beta$  adjusts the relative importance of reconstruction loss vs. KLD loss. So we can do gradient descent on  $\theta$ .



Because the VAE seeks a distribution in the latent space that is close to  $\mathcal{N}(0, I)$ , there are fewer holes. Take the previous simple shape example, we have

<sup>2</sup>Check [this link](#) on Is There Gravity in Space?



So now there are no (or fewer) gaps in the latent space.

# 10

## RNN

---

### 10.1 Recurrent Neural Networks

We have already seen one in the form of Hopfield networks. Thus far, we have mostly focussed on feedforward neural networks. There are no loops in a feedforward network. But there are reasons that we might want to allow feedback connections that create loops. For example, we recently looked at recurrent continuous-time networks to implement dynamics. Another example is that we want to build a running memory into our network behaviour.

Consider the task of predicting the next word in a sentence.

1. Emma's cat was sick, so she took her to the vet.
2. I'll work it out with pencil and paper.
3. She picked up the object, studied it, then put it down.
4. 0, 2, 4, 6.
5. 1, 2, 4, 8, 16.

In each case, the word you predict depends - in very complex ways - on the words that precede it. Thus, a network will probably have to encode an ordered sequence of words to solve this problem. How do we deal with a sequence of input?

#### Solution 1

Design the network so that the entire sequence is input all at once.

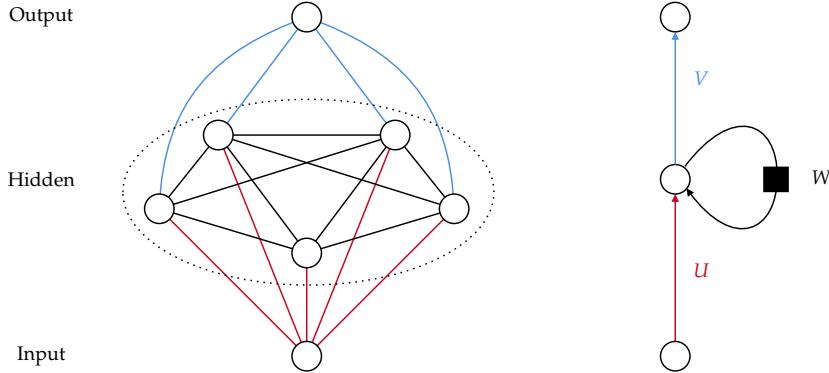
Consider the 4th example above, we design a network with 4 input neurons, accepting inputs [0 2 4 6] and one output neuron, which will give us 8. So this would work and there are times when this is the perfectly legitimate thing to do.

However, there are several problems:

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

## Solution 2

Allow the state of the network to depend on new input, as well as its previous state, namely, let the network be recurrent.



The picture on the left shows a possible structure of recurrent neural network. At the bottom, we have an input neuron, to which we can give 0, 2, 4, 6, in a sequence. In the middle, we have hidden layers. The black edges indicate the neurons are connected to each other, forming recurrent part of the network. On the top, it is an output neuron. Given an input, we are expecting it to output the next number in the sequence. For example, input 0, output 2; input 2, output 4 and so on.

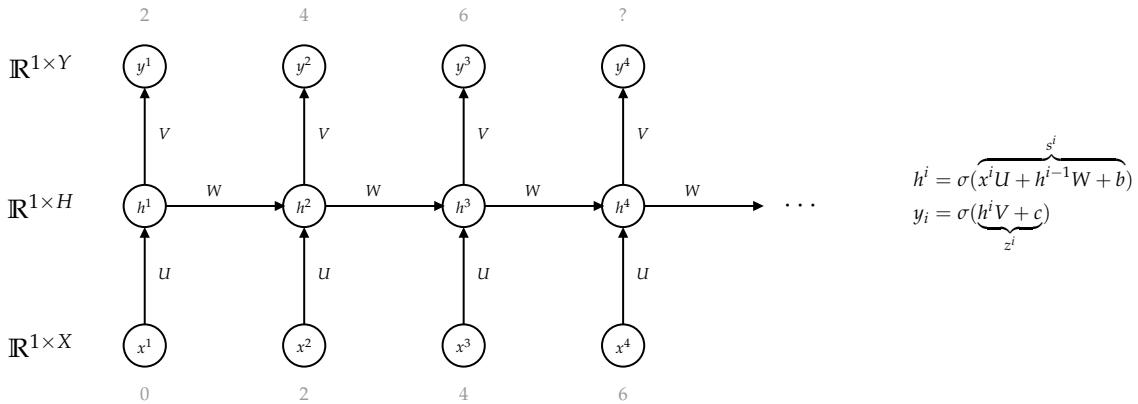
The picture on the right is a simpler version.  $U$  is the connection weight matrix between input and hidden,  $V$  is the connection weight matrix between hidden and output and  $W$  is the recurrent connection weight matrix back into the hidden layer. The black square indicates that it takes one time step for the feedback to happen.

This is a recurrent neural network (RNN). In an RNN, the state of the hidden layer can encode the input sequence, and thus have the information it needs to determine the proper output.

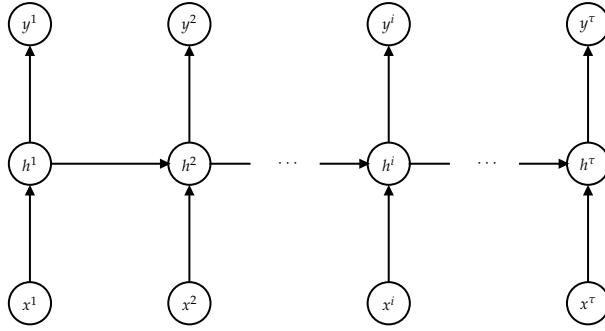
How do we train such a network?

### 10.1.1 Backprop through time (BPTT)

First, we unroll the network, similar to what we did for autoencoders. But this is unrolling through time.



Like before, we will have targets,  $t^i$ , and a loss function  $L$ . This unrolled network is a feedforward network. The expression graph is a DAG. Therefore, we can use Backprop.



Let  $\tau$  be the number of time steps. Then the loss, or error, is the sum of all the loss functions,

$$E(y^1, \dots, y^\tau, t^1, \dots, t^\tau) = \sum_{i=1}^{\tau} L(y^i, t^i) \underset{\substack{\uparrow \\ \text{we can put different weights} \\ \text{on parts of the sequence}}}{\alpha_i}$$

And as usual, we aim to minimize the expected cost over our dataset with respect to the connection weights and biases:  $\theta = \{U, V, W, b, c\}$ ,  $\min_{\theta} \mathbb{E}[E(y^1, \dots, y^\tau, t^1, \dots, t^\tau)]_{y, t}$ .

Just like in Backprop, we start with a forward pass. Then we can start at the last output, and propagate the error gradients down through the network. Gradients can be calculated as follows:

$$\begin{aligned} \nabla_{z^k} E &= \nabla_{z^k} \left( \sum_{i=1}^{\tau} L(y^i, t^i) \right) \\ &= \sum_{i=1}^{\tau} \nabla_{z^k} L(y^i, t^i) \\ &= \nabla_{z^k} L(y^k, t^k) \quad \text{only } y^k \text{ depends on } z^k \\ &= \nabla_{y^k} L(y^k, t^k) \odot \sigma'(z^k) \end{aligned}$$

where  $y^k = \sigma(z^k)$ .

Once we descend down to the hidden layer, it gets more interesting because each unrolled hidden layer depends on the one before it. So, we start at  $\tau$  and work out way back in time. We first define

$$E^k = \sum_{i=k}^{\tau} L(y^i, t^i).$$

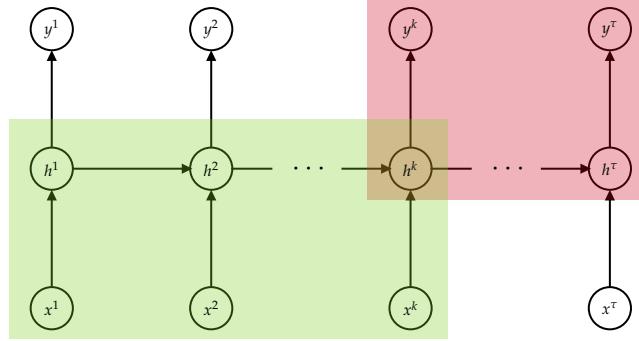
Notice that variables before time step  $k$  do not depend on variables after time step  $k$ , i.e.,  $h^k$  depends on  $h^{k-1}$ , not on  $h^{k+1}$ . By dependencies, we have

$$\nabla_{h^k} E = \nabla_{h^k} \sum_{i=k}^{\tau} L(y^i, t^i) = \nabla_{h^k} E^k.$$

Thus  $\nabla_{h^\tau} E = \nabla_{h^\tau} E^\tau$ . By chain rule, we have

$$\nabla_{h^\tau} E = \nabla_{z^\tau} E^\tau V^T$$

Note that all paths between  $E^k$  and variables  $x^i, s^i, h^i, i \leq k$ , must pass through  $h^k$ . This makes sense because in RNN, the only link to the future is the hidden stage.



Suppose we have already computed  $\nabla_{h^{i+1}} E$ . We can then compute

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

Once we have  $\nabla_{h^i} E$ ,  $i = 1, \dots, \tau$ , we can compute the gradient of the cost with respect to the weights and biases:  $\nabla_W E$ ,  $\nabla_V E$ ,  $\nabla_U E$ ,  $\nabla_b E$ ,  $\nabla_c E$ .

### 10.1.2 Long Short-Term Memory (LSTM)

Check CS 885 for more details. Two main takeaways from there: mitigate gradient vanishing and facilitate long term memory.

The benefit of an RNN is that it can accumulate a hidden state that encodes input over time. For example, a properly trained RNN could complete the sentence,

To ride a bicycle, put your feet on the pedals.

However, one difficulty with an RNN is that it can be very difficult to train it to maintain information in its hidden state for a long time. It would have trouble completing this paragraph,

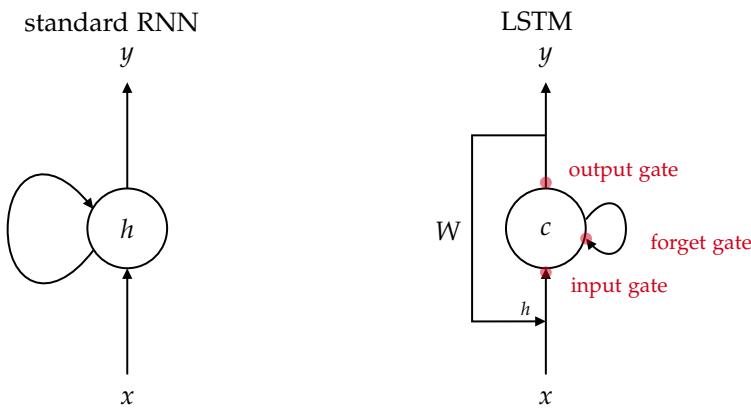
A bicycle is an efficient mode of transportation. ... To get started, pick one up and put your feet on the \_\_\_\_\_.

This instability was first pointed out by Bengio et al. in 1994. They show that the longer you need to keep information in memory, the harder it is to train. The BPTT algorithm becomes unstable because of exploding or vanishing gradients. Recall the formula for  $\nabla_{h^i} E$ , we then can expand the gradient inside:

$$\begin{aligned}\nabla_{h^i} E &= (\nabla_{h^{i+1}} E \odot \sigma'(s^{i+1})) W^T + (\nabla_{y^i} L(y^i, t^i) \odot \sigma'(z^i)) V^T \\ &= ((\nabla_{h^{i+2}} E \odot \sigma'(s^{i+2})) W^T \odot \sigma'(s^{i+1})) W^T + (\text{other stuff}) \\ &= \nabla_{h^{i+1}} E \odot \sigma'(s^{i+1}) W^T \cdots \sigma'(s^{\tau-1}) W^T \sigma'(s^\tau) W^T + \cdots\end{aligned}$$

The first term shown contributes to vanishing/exploding gradients.

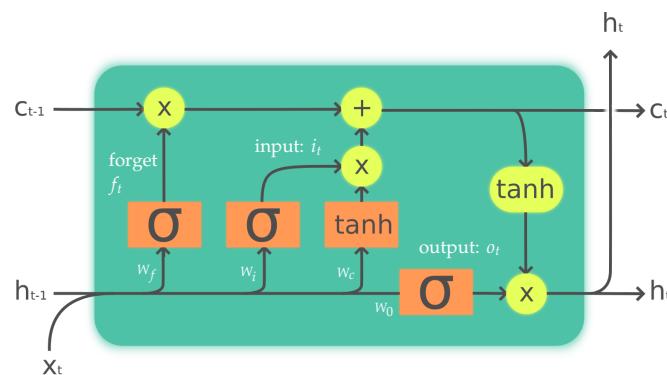
To combat this gradient decay, Hochreiter & Schmidhuber (1997) proposed an additional hidden state that persists from step to step. It does not pass through an activation function or get multiplied by the connection weights at each time step.



There are various ways  $h$  and  $c$  can interact:

- $h$  and  $x$  can increment  $c$  (input),
- $h$  and  $x$  can erase  $c$  (forget),
- $h$  and  $x$  control the output of  $c$  (output).

Putting it together, we have <sup>1</sup>



Legend:

Layer	Pointwise op	Copy

where  $W$ 's are weight matrices.

### Forget Gate

or unforgotten state, or “persistence gate”

$$f_t = \sigma([h_{t-1} \quad x_t]W_f + b_f)$$

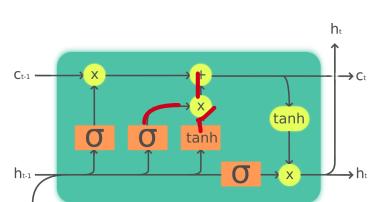
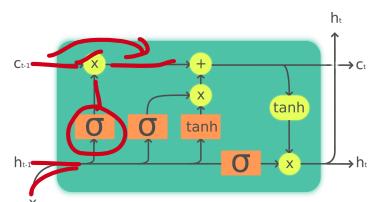
If this gate equals 1, persist; equals 0, forget.

### Input Gate

$$i_t = \sigma([h_{t-1} \quad x_t]W_i + b_i)$$

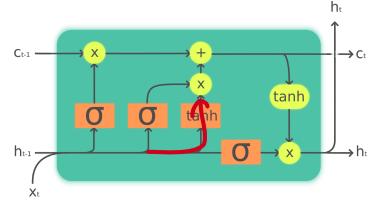
If equals 1, input; equals 0, don't input.

<sup>1</sup>picture from [https://commons.wikimedia.org/wiki/File:The\\_LSTM\\_cell.png](https://commons.wikimedia.org/wiki/File:The_LSTM_cell.png)



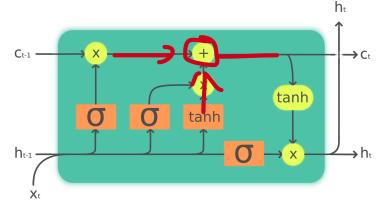
The input is

$$\tilde{c}_t = \tanh([h_{t-1} \ x_t]W_c + b_c)$$



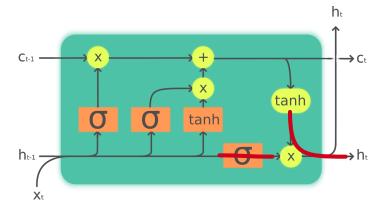
Combining the input with the non-forgetten state,

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$



### Output Gate

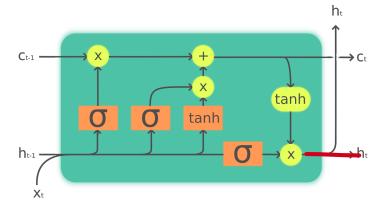
$$o_t = \sigma([h_{t-1} \ x_t]W_o + b_o)$$



If equals 1, output; 0, no output.

The actual output is

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



Denote  $x_t = [h_{t-1} \ x_t]$ . Then after all that, we have

$$c_t = \sigma(x_t W_f + b_f) c_{t-1} + \sigma(x_t W_i + b_i) \tanh(x_t W_c + b_c)$$

How does this make a difference?

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_{t-1}$$

How does  $\nabla_{c_{t-1}} E$  depend on  $\nabla_{c_t} E$ ?

$$\nabla_{c_{t-1}} E = \nabla_{c_t} E \odot f_t + \text{other stuff}$$

As long as  $f_t$  is close to 1, there is little decay of the gradient.

## 10.2 Our memory systems

There are three types of memories or memory systems that you have or that all mammals have. We basically have three different memory systems.

1. The shortest will is called *working memory*. It's like we're remembering a phone number and we keep repeating it to ourselves, then it sort of stays in our mind. However, as soon as we get distracted, we forget that number and it's gone. That seems to be implemented in the collective activity of populations of neurons, as long as they're maintaining their firing pattern.

2. The next level is *short-term memory*. It's actually longer than working memory. It's like where we can remember what we ate for breakfast this morning. That seems to be implemented in our hippocampus, which is a medial temporal lobe. They're stored basically by changing the efficacy of the synapses and the neurons. It's not the activity of the neurons that's storing the information. Instead, it's the short-term changes in the excitability of the synapses in the neurons. Those can maintain those memories for a 24-48 hour period.

It seems that what happens when we're sleeping or dreaming is that some of those memories get encoded into long-term memory.

3. *Long-term memory* is things like remembering our parents names. Those types of memories get encoded in the actual wiring, or the connections between our neurons in our cortex. The connections between those neurons seem to encode our long-term memories. That's like changing the connection weights and biases in our cortical neural network.

# Adversarial Attack and Defense

## 11.1 Adversarial attack

Suppose we are given a dataset,  $D = \{(x, t) \mid x \in X, t = 1, \dots, k\}$ , where  $X \subseteq \mathcal{X}$ ,  $\mathcal{X}$  is the whole space, and  $t$  the class of  $x$ , i.e., index.

Suppose we are given a classifier network for that dataset,  $f : \mathcal{X} \rightarrow P^k$ , where  $P^k$  is a space of probability vectors:  $P^k := \{y \in \mathbb{R}^k \mid 0 \leq y_i \leq 1, \sum_i y_i = 1\}$ . For example,  $y = \text{Softmax}(z)$ .

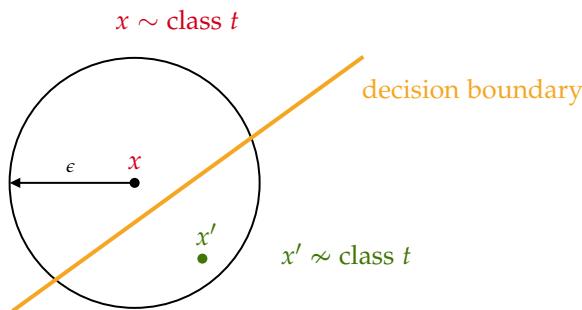
We can measure its classification errors using the loss function:

$$R(f) := \mathbb{E}_D \left[ \underset{\text{counts the # of events}}{\mathbb{1}} \{ \text{argmax}_i(y_i) \neq t \mid (x, t) \in D, y \in f(x) \} \right]$$

Let's define the  $\epsilon$ -ball (neighbourhood) of an input  $x$  as

$$\mathbb{B}(x, \epsilon) = \{x' \in \mathcal{X} \mid \|x' - x\| \leq \epsilon\}.$$

We ask ourselves, given an  $(x, t) \in D$ , is there  $x' \in \mathbb{B}(x, \epsilon)$  such that  $\text{argmax}_i(y') \neq t$ ,  $y = f(x')$ . In words, is there a very nearby input that would fool the network and yield an incorrect classification?

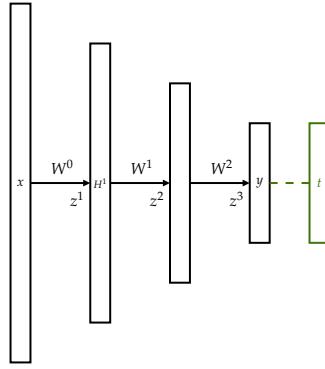


This situation happens remarkably often. These adversarial examples can be found quite easily. This is called an **adversarial attack**.

How do we generate examples to fool our neural network model? There're two basic classes of attacks:

- **whitebox attack**: the attacker has access to the parameters of the neural network (weights and biases, etc.)
- **blackbox attack**: the attacker only has access to the inputs and outputs of the network.

We will describe a common whitebox attack method. Recall, learning is done by gradient descent,  $\theta \leftarrow \theta - k\nabla_\theta E$  where  $E$  is our loss function.



Using backpropagation, we propagate the gradient of the cost function down through the layers of the network. We start at the top and work our way down down down, for example, down to  $z^1$ , we get the gradient  $\nabla_{z^1} E$ . What about  $\nabla_x E$ ? where  $x$  is the input. In order to calculate  $\nabla_x E$ , we use  $\nabla_{z^1} E$ . We have  $z^1 = xW^0 + b^1$ , therefore,

$$\nabla_x E = \nabla_{z^1} E \frac{\partial z^1}{\partial x} = \nabla_{z^1} E \cdot (W^0)^T.$$

This gives us the gradient of the loss with respect to the input. This gradient tells us how to adjust our input in order to decrease/increase our loss:

$$x = x + \kappa \nabla_x E(f(x, \theta), t(x)),$$

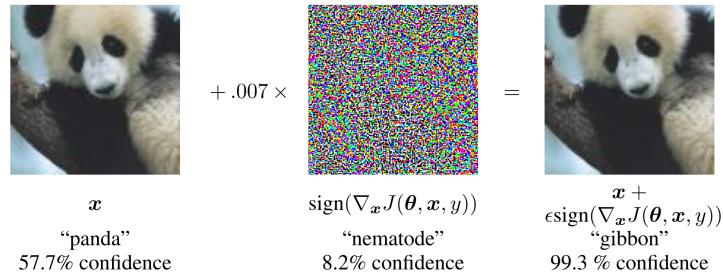
which is gradient *ascent*: pushing input in direction to *increase* loss. Or

$$x = x - \kappa \nabla_x E(f(x, \theta), \ell) \quad \text{for } \ell \neq t(x),$$

which is gradient *descent* for incorrect target.

These two different adversarial attacks are called *untargeted* and *targeted*. This first one is untargeted, because we are just pushing it away from something and we don't have a target. The second one is targeted because we would like to have our input to like a specific target.

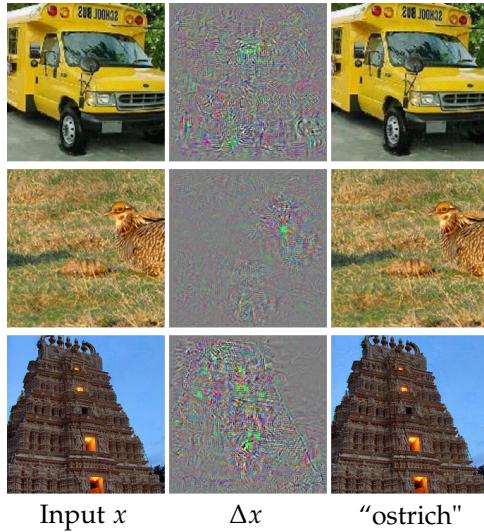
For example, a change in pixel intensity of 1 in an 8-bit image is imperceptible to the human eye. If we want to perturb our image by 1 for each pixel, then we let the perturbation be  $\Delta x = \text{sign}(\nabla_x E)$ , i.e.,  $\Delta x = \pm 1$ , so  $\|\Delta x\|_\infty = 1$ .



Goodfellow, I. J., Shlens, J., & Szegedy, C. (2015). Explaining and Harnessing Adversarial Examples. In *Proc. of ICLR*.

The image on the right here is an adversarial image.

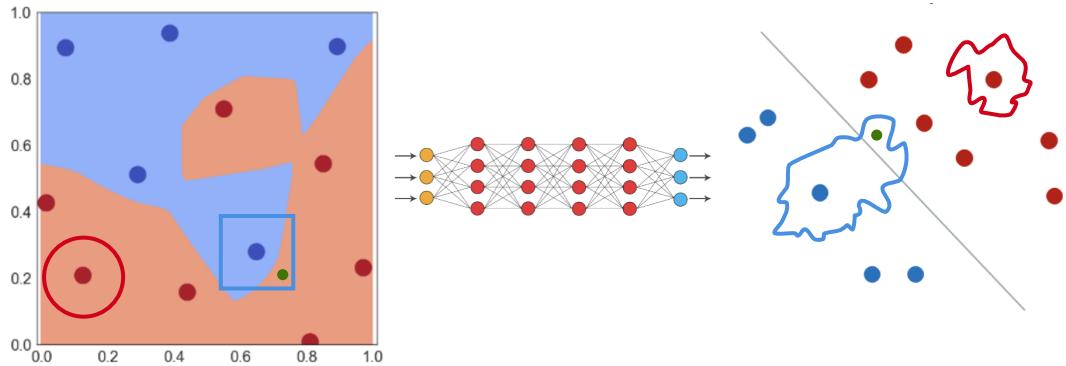
One can even look for the smallest image perturbation that will result in a particular class.



Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., & Fergus, R. (2014). Intriguing properties of neural networks. *arXiv*, 1312.6199v4.

Overcoming this problem is a very active field of research.

*Why are classification networks “easily fooled”?* Consider the input space; for  $28 \times 28$  for MNIST, dimension of input is 784. That’s a lot of axes, and a lot of space. The classification network partitions this high-dimensional space into regions, for example, 10 regions for MNIST. It turns out that most points are not too far away from a decision boundary.



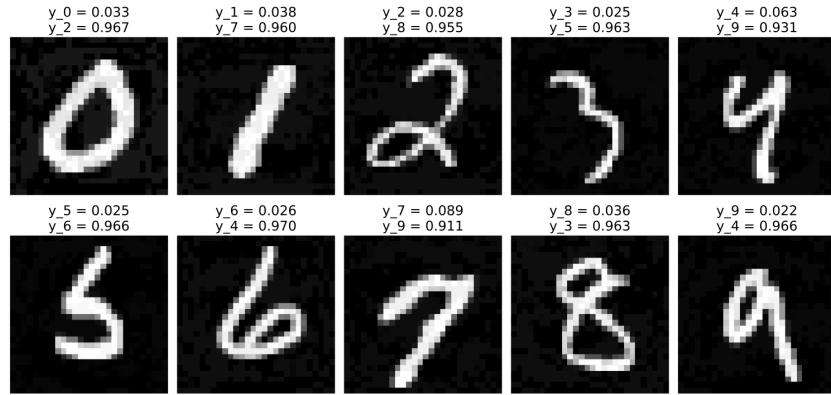
Picture from [Hongyang Zhang](#).

We can see the network has managed to separate these two classes. Now let’s take a look at the neighbourhood of points in the input space and feature space correspondingly. Red ball looks ok, but note that blue ball (using infinity norm) crosses the decision boundary. So we end up getting the green point is on the wrong side of decision boundary.

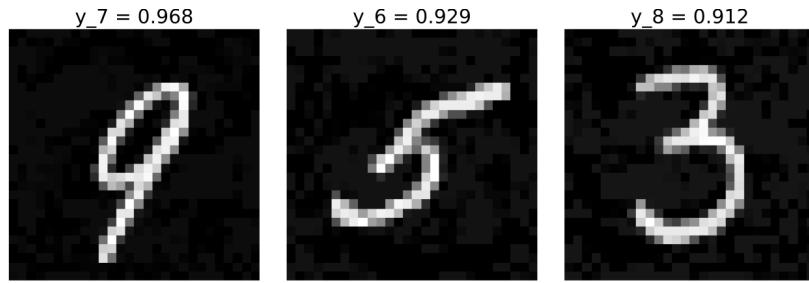
In summary, learning is  $\min_{\theta} \mathbb{E}_D [L(f(x), t)]$ , while adversarial attack is

$$\max_{x' \in \mathbb{B}(x, \epsilon)} L(f(x'), t), \quad \text{or} \quad \min_{x' \in \mathbb{B}(x, \epsilon)} L(f(x'), \ell) \text{ for } \ell \neq t.$$

Here are some examples from MNIST. First let’s see some untargeted.



Then consider some targeted attacks.

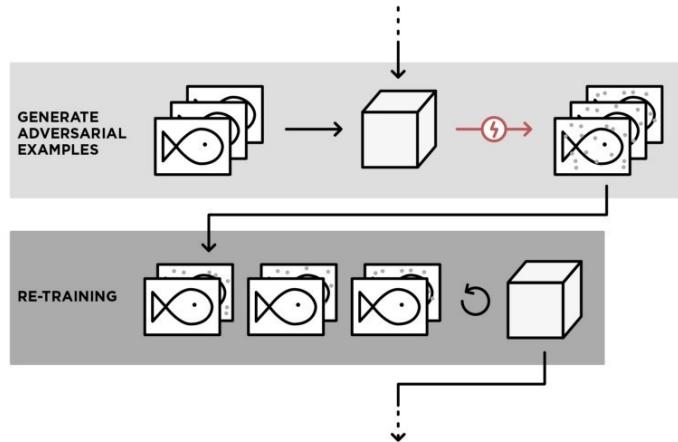


For example, in terms of the first image, it's a picture of 9, but we change it so that it's more like a 7. Similar for the other two images.

## 11.2 Adversarial defence

To combat adversarial attacks, there are several ways to train our neural networks so that they are not so easily fooled.

### 11.2.1 Adversarial training



Picture from [medium.com](https://medium.com/@mikemcdonald007/adversarial-training-explained-101-10f3a2a2a2d). Also from this article, which the instructor didn't focus on:

**Adversarial training** (Goodfellow et al., 2014) is a defence method used to increase adversarial robustness by retraining a model on adversarial examples. In adversarial training, adversarial examples are generated at each iteration based on current state of the model,

and are used to retrain the model.

**Gradient masking** (Papernot et al., 2016a; Papernot et al., 2016b; Tramèr et al., 2017): the practice of altering a model to hide its original gradients from an attacker. In other words, the defence methods mask the gradients of the model’s output with respect to its inputs.

Consider a different idea: What if we incorporate a miniature adversarial attack into every gradient step while training?

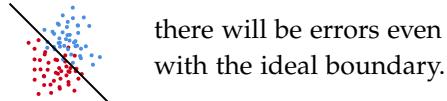
### 11.2.2 TRADES

which stands for TRadeoff-inspired Adversarial DEfense via Surrogate-loss minimization. The paper is [Theoretically Principled Trade-off between Robustness and Accuracy](#), which has quite a lot theory.<sup>1</sup>

Consider a model (NN)  $f : \mathcal{X} \rightarrow \mathbb{R}$ , data set  $D = (\mathcal{X}, T)$ , where inputs  $\mathcal{X} \subseteq \mathcal{X}$ , targets  $T \in \{-1, 1\}$ . Then  $\text{sign}(f(x))$  is the class of  $x$ . Therefore, classification is correct if  $f(\mathcal{X})T > 0$ . The classification loss (natural classification error) can be written

$$\mathcal{R}_{\text{nat}} := \mathbb{E}_D [\mathbb{1}\{f(\mathcal{X})T \leq 0\}] .$$

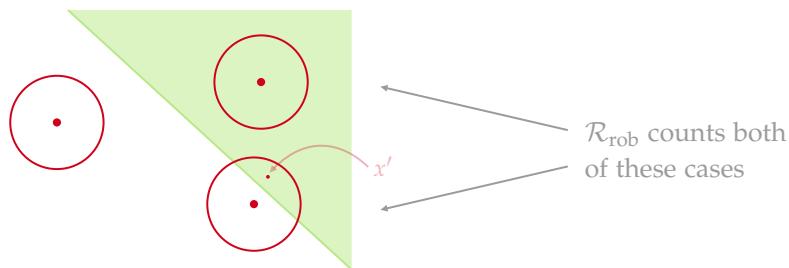
We can picture it as follows: even if the decision boundary seems reasonable,



If we want to consider how our model will perform under adversarial attack, we consider the robust loss:

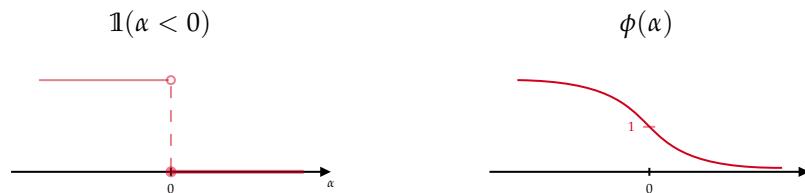
$$\mathcal{R}_{\text{rob}}(f) := \mathbb{E}_D [\mathbb{1}\{\exists X' \in \mathbb{B}(X, \epsilon) \mid f(X')T \leq 0\}] .$$

It has a built-in pessimism, that looks for the worst-case in the neighbourhood of  $X$ .



From paper, we note that the two errors satisfy  $\mathcal{R}_{\text{rob}}(f) \geq \mathcal{R}_{\text{nat}}(f)$  for all  $f$ ; the robust error is equal to the natural error when  $\epsilon = 0$ .

For theoretical reasons that I won’t go into (read, “I don’t understand”), we use a surrogate loss function,  $\phi$ , instead of using the binary classifier criterion.



and convert our learning task to  $\min_f \mathbb{E}_D [\phi(f(\mathcal{X})T)]$ . Then we try to train a robust model using the

<sup>1</sup>The instructor will elude to it but he won’t go into it. As he jokingly says in his notes later on, he doesn’t go into it because he doesn’t understand it. So later he will outline the part that he understands.

combined loss:

$$\min_f \mathbb{E}_D \left[ \underbrace{\phi(f(X)T)}_{(1)} + \underbrace{\max_{X' \in \mathbb{B}(X, \epsilon)} \phi(f(X)f(X')) / \lambda}_{(2)} \right]$$

- (1) *For accuracy.* Try to ensure that each  $X$  is correctly classified. In other words, from paper, this term encourages the natural error to be optimized by minimizing the “difference” between  $f(X)$  and  $T$ .
- (2) *Regularization for robustness.* Add a penalty for models  $f$  that put  $X$  within  $\epsilon$  of the decision boundary. In paper’s words, this term encourages the output to be smooth, that is, it pushes the decision boundary of classifier away from the sample instances via minimizing the “difference” between the prediction of natural example  $f(X)$  and that of adversarial example  $f(X')$ .

Here  $\lambda > 0$  is a regularization parameter, which the instructor has left out, but later added on in the implementation.

The implementation is as follows:

For each gradient-descent step:

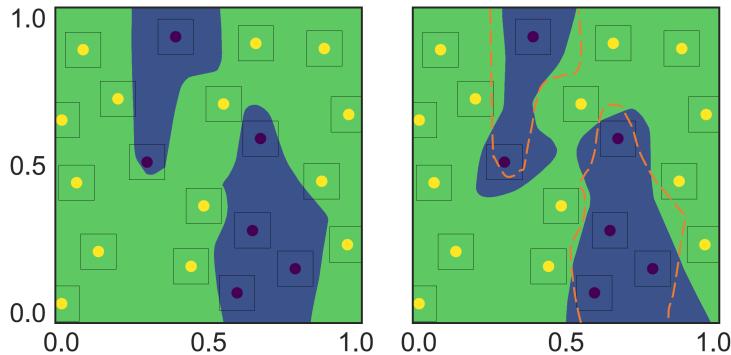
- run several steps of gradient ascent to find  $X'$  then fix  $X'$
- evaluate the joint loss:

$$\text{loss} = \phi(f(X)T) + \beta \cdot \phi(f(X)f(X'))$$

where  $\beta$  is a regularization parameter.

- use gradient of  $\text{loss}$  for gradient step.

The result is as follows (picture from paper):



**Left figure:** decision boundary learned by natural training method. **Right figure:** decision boundary learned by our adversarial training method, where the orange dotted line represents the decision boundary in the left figure. It shows that both methods achieve zero natural training error, while our adversarial training method achieves better robust training error than the natural training method.

The code is available at <https://github.com/yaodongyu/TRADES>.

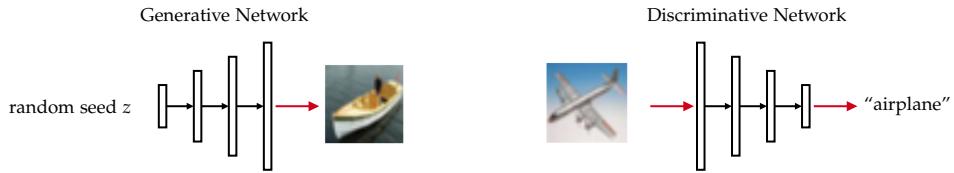
The TRADES method won the NeurIPS’18 Adversarial Vision Challenge. As of March 2021, in [RobustBench](#), 5 of the top 5, and 9 of the top 10 methods are all based on TRADES.

# 12

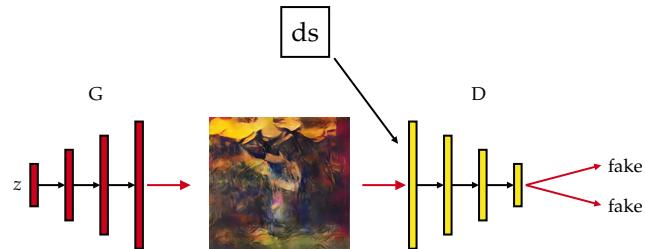
## GANs

### 12.1 Introduction

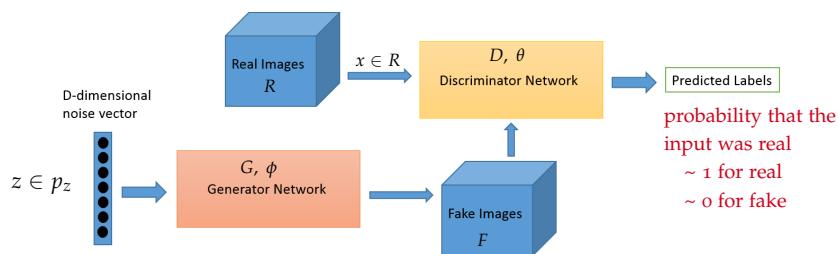
A Generative Adversarial Network (GAN) is comprised of two networks that are competing against each other.



A GAN links them together:



The Generative network is trying to fool the Discriminative network into misclassifying "fake" (generated) inputs. Meanwhile, the Discriminative network is also learning how to classify inputs as "fake" or "real", i.e., does it come from a dataset, or from the Generative network?



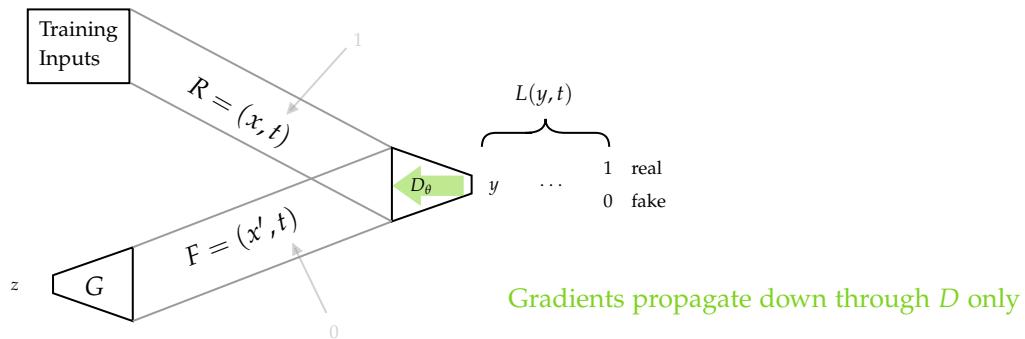
$D(x; \theta)$  outputs the probability that  $x$  is from the real data, i.e.,  $x \in R$ , and we set  $t = 1$ .  $G(z; \phi)$  creates an input sample from random noise  $z$  with distribution  $p_z$ , i.e.,  $x' = G(z; \phi) \in F$ , and we let  $t = 0$ .

## 12.2 Loss Function for a GAN

The loss for the GAN has two parts:

The first term is to minimize with respect to  $\theta$  to make  $D$  better at detecting real vs. fake inputs. The second term is to minimize with respect to  $\phi$  to make  $G$  better at producing fake inputs that fool  $D$ .

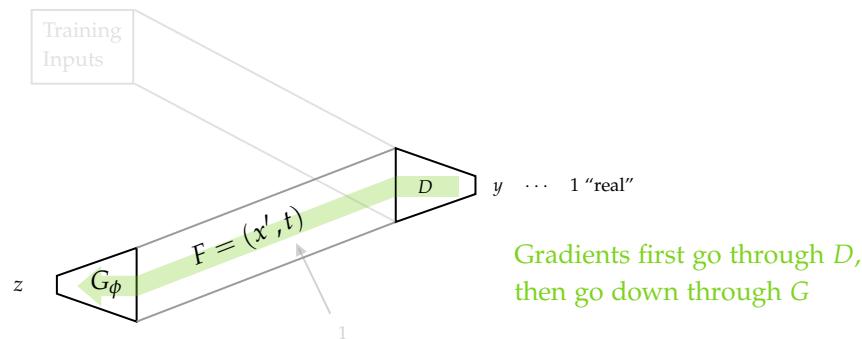
### 12.3 Training the Discriminative Network



We aim to  $\min_{\theta} \mathbb{E}_{R,F}[L(y, t)]$ .

Train  $D$  to distinguish real from fake inputs, and  $\theta \leftarrow \theta - \kappa \nabla_{\theta} L(y, t)$ .

## 12.4 Training the Generative Network



We aim to  $\min_{\phi} \mathbb{E}_F[L(y, 1)]$ .

This is like a targetted Adversarial attack with target set to 1. Then  $\phi \leftarrow \phi - \kappa \nabla_\phi L(y, 1)$ .

## 12.5 Examples

See example from <https://thispersondoesnotexist.com/>.

Also see examples from [freecodecamp.org](https://freecodecamp.org).

# 13

## Vector Embeddings

---

# A

## Implementation of Neural Network

---

The content here is based on the exercise from week 3.

To be added.

# B

## 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.

```
1 import numpy as np
2 import torch
3 import matplotlib.pyplot as plt
4 import copy
```

### B.1 PyTorch Tensors

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

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

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

```
1 >>> y = torch.log(x)
2 >>> print(y)
3 tensor([[0.0000, 0.6931],
4         [1.0986, 1.3863]])
```

PyTorch has auto-differentiation functionality.

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

```
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 >>> y.backward()
4 >>> z.grad      # dy/dz = [2,1]
5 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.

```

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

```

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

```

1 z = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
2 y1 = torch.prod(z)  # dy1/dz = [2,1]
3 y2 = torch.sum(z)  # dy2/dz = [1,1]
4 y = y1 + y2
5 y.backward()
6
7 >>> 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:

```

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

```

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

```

1 >>> zt = torch.tensor([1,2], dtype=torch.float)
2 >>> zn = zt.numpy()               # does NOT make a copy
3 >>> print(f'numpy version: {zn}')
4 >>> zn[0] = -1000.
5 >>> print(zt)
6 numpy version: [1. 2.]
7 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:

```

1 >>> zt = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
2 >>> zn = zt.numpy()
3 >>> print(f'numpy version: {zn}')
4 -----
5 RuntimeError                                         Traceback (most recent call last)
6 <ipython-input-8-dfae4b9d9e05> in <module>
7     1 zt = torch.tensor([1,2], dtype=torch.float, requires_grad=True)
8 ----> 2 zn = zt.numpy()
9     3 print(f'numpy version: {zn}')
10 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:

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

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

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

## B.2 Classification Dataset

We will use PyTorch's Dataset class.

```
 1 from torch.utils.data import Dataset, DataLoader  
  
 1 >>> B = torch.eye(3)  
 2 >>> A = 3.*torch.rand((3, 6)) - 0.5)  
 3 >>> print('Input vectors')  
 4 >>> print(A)  
 5 >>> print('Target vectors')  
 6 >>> print(B)  
 7 Input vectors  
 8 tensor([[-1.3455,  0.6580, -0.2041, -0.0052,  0.2841, -0.6806],  
 9          [-0.3096,  0.4767,  0.0416,  0.8221,  0.3553, -0.9552],  
10          [ 0.0603,  1.0587,  0.6078, -0.5603,  0.9886, -1.1840]])  
11 Target vectors  
12 tensor([[1., 0., 0.],  
13          [0., 1., 0.],  
14          [0., 0., 1.]])
```

```
1 class DiscreteMapping(Dataset):
2     def __init__(self, A, B, n=300, noise=0.1):
3         self.samples = []
4         self.n_classes = len(A)
5         self.input_dim = len(A[0])
6         for i in range(n):
7             r = np.random.randint(self.n_classes)
8             t = B[r]
```

```

 9     sample = [A[r]+noise*torch.randn_like(A[r]), t]
10     self.samples.append(sample)
11
12     def __getitem__(self, idx):
13         return self.samples[idx]
14
15     def __len__(self):
16         return len(self.samples)
17
18     def inputs(self):
19         x = []
20         for s in self.samples:
21             x.append(s[0])
22         return torch.stack(x)
23
24     def targets(self):
25         t = []
26         for s in self.samples:
27             t.append(s[1])
28         return torch.stack(t)
29
30     def classes(self):
31         c = []
32         for s in self.samples:
33             k = torch.argmax(s[1])
34             c.append(k)
35         return torch.tensor(c, dtype=torch.long)
36
37     def plot(self, labels=[], idx=(0,1), equal=True):
38         X = self.inputs()
39         if len(labels)==0:
40             labels = self.classes()
41         colour_options = ['y', 'r', 'g', 'b', 'k']
42         cidx = self.classes()
43         colours = [colour_options[k] for k in cidx]
44         plt.scatter(X[:,idx[0]], X[:,idx[1]], color=colours, marker='.')
45
46         if equal:
47             plt.axis('equal');

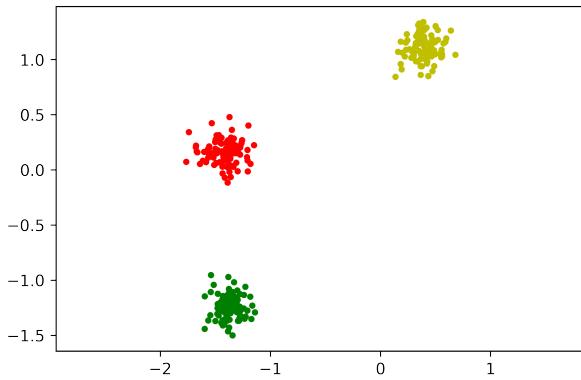
```

Let's now create a dataset and plot it:

```

1 ds = DiscreteMapping(A, B)
2 ds.plot

```



We can look at the inputs, targets and classes:

```

1 >>> print(ds.inputs()[:5])
2 tensor([[ 0.0648,   1.0591,   0.3801,  -0.4864,   0.9960,  -1.1367],
3         [-0.4479,   0.5487,   0.1092,   0.7356,   0.3301,  -0.9817],
4         [ 0.0445,   1.0273,   0.5690,  -0.5552,   1.1006,  -1.1681],
5         [ 0.0491,   1.0013,   0.7913,  -0.5124,   1.0008,  -1.1813],
6         [-0.0058,   1.0477,   0.7242,  -0.7067,   0.8411,  -1.2293]])
7
8 >>> print(ds.targets()[:5])
9 >>> print(ds.classes()[:5])
10 tensor([[0., 0., 1.],
11         [0., 1., 0.],
12         [0., 0., 1.],
13         [0., 0., 1.],
14         [0., 0., 1.]])
15 tensor([2, 1, 2, 2, 2])

```

## B.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.

```

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))

```

We have more flexibility with `Module`, since we 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()    # 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

```

Then we can create the network model:

```
1 net = mynet()
```

Now we need a **loss funciton**.

```

1 # loss = torch.nn.CrossEntropyLoss(reduction='mean')
2 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.

```

1 >>> y = net(ds.inputs())
2 >>> loss = loss_fcn(y, ds.classes())
3 >>> print(loss)
4 tensor(1.1283, grad_fn=<NllLossBackward>)

```

Then we can look at the output of the network:

```

1 >>> y[:5]
2 tensor([[-0.9371, -0.9805, -1.4561],
3         [-0.9358, -0.9831, -1.4544],
4         [-0.9361, -0.9867, -1.4481],
5         [-0.9358, -0.9827, -1.4548],
6         [-0.9355, -0.9875, -1.4477]], grad_fn=<SliceBackward>)

```

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

```

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

```

Currently, the network hasn't been trained yet.

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

```

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

```

Another way to look at it:

```

1 >>> params = list(net.parameters())
2 >>> print(params[3])
3 Parameter containing:
4 tensor([[ 0.2359, -0.2637, -0.2381,  0.3370, -0.0659],
5         [-0.1568,  0.1390, -0.0219, -0.1441,  0.4145],
6         [ 0.1928, -0.4072, -0.2311,  0.2990,  0.1112]], requires_grad=True)

```

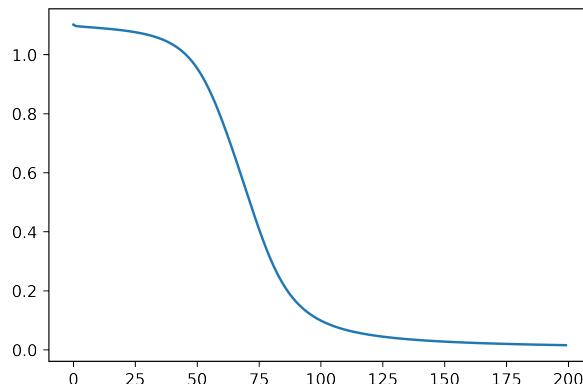
Now let's train the classification model myself. We can access the parameter gradients and implement gradient descent ourselves.

```

1 x = ds.inputs()
2 targets = ds.targets()
3 classes = ds.classes()

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) # for CE
8     losses.append(err.item())
9     net.zero_grad()
10    err.backward()
11    with torch.no_grad():
12        for p in net.parameters():
13            p -= lrate * p.grad
14 plt.plot(losses);

```



After we trained the network,

```

1 >>> y = net(x)
2 >>> print(y[:5])
3 >>> print(torch.exp(y[:5]))
4 tensor([[-5.1984, -0.0158, -4.5924],
5         [-5.1438, -0.0130, -4.9449],
6         [-0.0168, -4.6187, -4.9900],
7         [-5.2278, -0.0139, -4.7713],
8         [-0.0177, -4.5456, -4.9715]], grad_fn=<SliceBackward>)
9 tensor([[0.0055, 0.9843, 0.0101],
10        [0.0058, 0.9870, 0.0071],
11        [0.0061, 0.9897, 0.0048],
12        [0.0064, 0.9924, 0.0025],
13        [0.0067, 0.9951, 0.0002]])

```

```

11     [0.9833, 0.0099, 0.0068],
12     [0.0054, 0.9862, 0.0085],
13     [0.9825, 0.0106, 0.0069]], grad_fn=<ExpBackward>

```

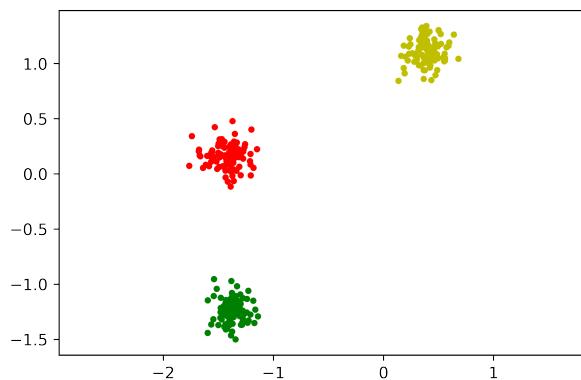
which matches the targets:

```

1 >>> targets[:5]
2 tensor([[0., 1., 0.],
3         [0., 1., 0.],
4         [1., 0., 0.],
5         [0., 1., 0.],
6         [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.

```

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

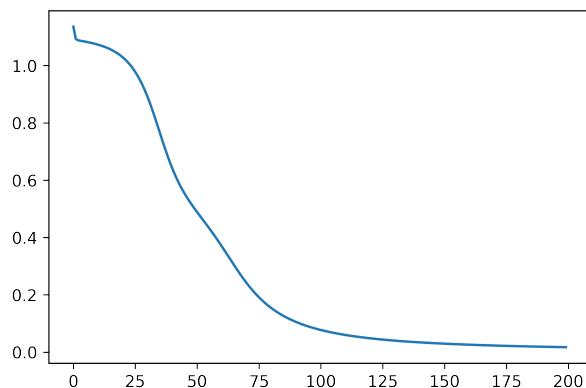
```

Then create the loop over epochs:

```

1 n_epochs = 200
2 losses = []
3 for epoch in range(n_epochs):
4     y = net(x)
5     err = loss_fcn(y, classes) # for CE
6     losses.append(err.item())
7     optim.zero_grad()
8     err.backward()
9     optim.step()
10 plt.plot(losses);

```



Then it works:

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