ECE/CS 559 Lecture 1 8/27

Mesrob Ohannessian M 1-2pm (Zoom)
T 11-12noon (SEO 1050)

TAs:

Alperen Gormez Q&A M 10-12 pm
Kenya Andrews Tutorials Th 1-3 pm
Runxuan Miao Q&A T 3-5 pm

Homeworks: due on Tuesdays 9pm Gradescope
next day noon - no penalty
after that → grade = 0
plagiarism → grade = -100
Homework 1 is out! Due T Sept.

Grading
Homework 20%
Midterm } min 30%
Final } max 40%
Participation / Attendance 10%
repeat plagiarism → 0

Material
- Mostly: notes (handwritten)
recorded lectures
recorded tutorials

Reference books: in syllabus.

Topics Covered:
- Models of artificial / mathematical neurons
- Single and multilayer neural networks.
- Perceptron
- Supervise learning (intro)
- Backpropagation algorithm
- Convolutional Neural Networks (CNNs)
- Unsupervised Learning (intro)
- k-Means, Hebbian, contrastive, GANs
- Reinforcement Learning (intro)
- RNNs, Attention and Transformers
- Diffusion models

Timeline of NN Research

- 1943 McCulloch and Pitts (at UIC!)
Linear threshold "neuron"
- 1962 Rosenblatt "Hebbian" learning
→ strengthen used connections
Novikoff: "perceptron" algorithm converges
- 1969. Minsky & Papert publish "Perceptrons"
↓
discourages work in artificial neural networks

But... some people continued working and "machine learning" (with other methods) was born:

- 1980 Fukushima : CNNs
- 1984 Valiant : PAC Learning
- 1985 Hinton & Sejnowski : Boltzmann machines
- 1986 same lab : success with back propagation (existed since 1961)
- 1997 LSTMs
- 1998 LeCunn reforms on CNNs

2003 Bengio neural language models

Modern Era : Deep Learning

2012 AlexNet
speech recognition

2014 VAEs, GANs, GNNs
computer vision

2015 Diffusion Models

2016 AlphaGo

2017 Transformers

2018 Pretraining, Contrastive

2021 GPT-3, DALL-E

2023 GPT-4, ChatGPT
RLHF

ECE/CS 559 - Neural Networks Lecture Notes #1
Neural networks: Definitions, motivation, properties

Erdem Koyuncu

1 Neural networks: Definitions and motivation

- Neural: Of, or related to nerves, neurons.
- Neuron: A highly-specialized cell that can transmit and receive information through electrical/chemical signaling. The fundamental building block of the brain.
- Nerve: A bundle of several neurons (more precisely, bundle of axons of several neurons).
- Network: A set of nodes and the connections between them.
- Biological neural network: The structure formed by many neurons and the connections between them.
- Artificial neural network (or just "neural networks" from now on): Mathematical models of biological neural networks.
- Why study neural networks?
  - Human brain is very successful at certain tasks that prove to be very difficult to accomplish on a classical computer: Learning, pattern recognition,... For example, think about what we call a generic classification example: When I show you the picture of an apple, you immediately recognize it as an apple. Or, as I speak, you recognize what I say. Computers usually are terrible at such tasks. E.g. even with todays' technology, most commercial speech recognizers still cannot get all of what you say because of e.g. your accent. So, it makes sense to look at how brain or the human neural network works.
  - So, why would human brain be much better at certain tasks compared to computers? Can it be about speed? No, a transistor can switch in nanoseconds, but if you put a needle to your leg, it takes a few milliseconds for your brain to realize what is going on. So, it is not necessarily the speed. What brain lacks in speed, it compensates with the number of processing elements, and their massive interconnected nature and everything running in parallel as opposed to the serial operation of the computer. Another important issue is that the human brain is constantly evolving in the sense that the way it operates can be modified through a learning process, while the physical structure of the CPU of a computer is fixed and does not change over time.
- Thus, to summarize: Classical computer: Almost serial operation (think about running a program for example and ignore possible pipelining), each operation can however be performed extremely fast ($<$ nanoseconds per operation.).
- Neural network: Massively parallel operation, may be more powerful than the classical computer despite the fact that a given neuron is relatively "slow." (A neuron can fire at most hundreds of times per second $\rightarrow$ milliseconds per "operation").
- Motivation: Perhaps, by imitating the brain's operation (by building artificial neural networks), we will be able to solve complex tasks that are not easily doable by classical computers.
  - Arguably the most important motivating factor for the developments in the field.

  - Other motivations: Understand how the brain and the nervous system work.

- Hence, to summarize, neural networks:

  1. Are massively parallel distributed processors that are made up of simple processing units (mathematical neurons) and the connections between these processing units.
  2. Can store and utilize experiential knowledge through a learning process.

- An artificial neural network resembles the brain in two respects:

  - Knowledge is acquired by the network from its environment through a learning process.
  - Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

2 Purposes

Neural networks can be used for a variety of purposes, e.g. classification (identifying the label/class of an input among a given set of labels/classes), clustering (finding different classes among unlabeled data), prediction (estimating the future values of a process given past values), strategic decision making, so on. But in general, mathematically, we have an input, the network, and an output...

3 Some properties of artificial neural networks:

- Non-linearity: The relationship between the input and the output of the network is not a linear relationship.
- Learning.
  - Supervised learning: Learning with a teacher. Involves modification of the parameters of the network through a set of training samples, with each sample consisting of an input pattern and associated desired response.
  - Unsupervised learning.
- Adaptivity: Adaptation of the free parameters to the changes in the environment.
- Evidential Response: The network can also provide a confidence level to its decision regarding the input pattern.
- Fault tolerance: When you get hit in the head, many of your neurons die, but you may still continue your daily routine. On the other hand, kill a transistor in a CPU, and the entire CPU is dead too. Thus, in principle, a neural network exhibits a graceful degradation in performance rather than catastrophic failure.
- Neurobiological analogy: Neurobiologists look to (artificial) neural networks as a research tool for the interpretation of neurobiological phenomena. On the other hand, engineers look to neurobiology for new ideas to solve problems more complex than those based on conventional hardwired design.

4 Our approach in this course:

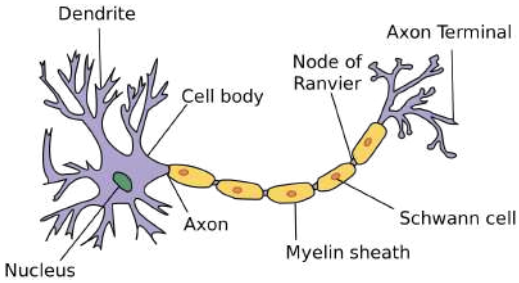
- Find mathematical models describing how the neuron (the basic building element of the brain) works.
- Find an appropriate network of such mathematical neurons that can perform the given task.

ECE/CS 559 - Neural Networks Lecture Notes #2
Mathematical models for the neuron, Neural network architectures

Erdem Koyuncu

1 Mathematical models for the neuron

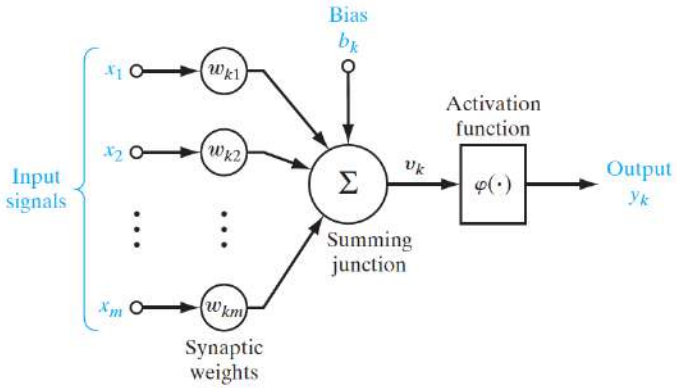
1.1 The biology of the neuron



(Image taken from Wikipedia)

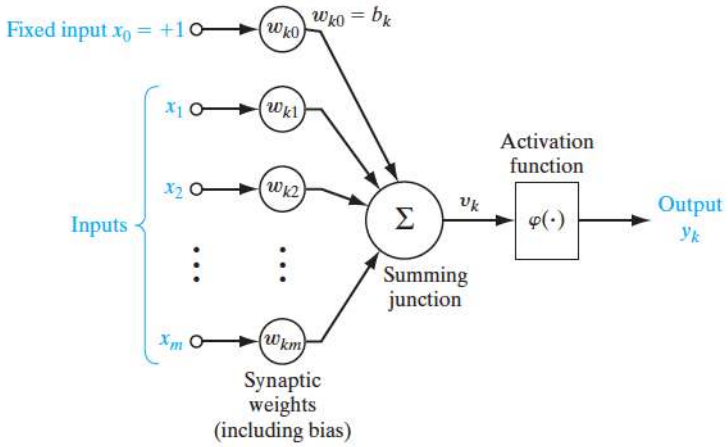
- Neuron: A highly-specialized cell that can transmit and receive information through electrical/chemical signaling. The fundamental building block of the brain.
- Synapse: The structure that connects two given neurons, i.e. the functional unit that mediates the interactions between neurons.
- A chemical synapse: Electrical signal through axon of one neuron $\rightarrow$ chemical signal / neurotransmitter $\rightarrow$ postsynaptic electrical signal at the dendrite of another neuron.
- Information flows through dendrites to axons: Dendrite (of neuron A) $\rightarrow$ Cell body (Soma) $\rightarrow$ Axon $\rightarrow$ Synapses $\rightarrow$ Dendrite (we are now at neuron B) $\rightarrow$ Cell body (Soma) $\rightarrow$ Axon $\rightarrow$ Synapses $\rightarrow \cdots$.
- A neuron can receive thousands of synaptic contacts and it can project onto thousands of target neurons.
- A synapse can impose excitation or inhibition (but not both) on the target neuron. The "strength" of excitation/inhibition may vary from one synapse to another.
- There are thousands of different types of neurons. There are, however, 3 main categories.
  - Sensory neurons: Conveys sensory (light, sound, touch) information.
  - Interneurons: Convey information between different types of neurons.
  - Motor neurons: Transmit muscle/gland control information from the brain/spinal cord.
- The soma is a few to a hundred micrometers in diameter. Axons are generally much longer, and can can in fact be up to a meter long! (e.g. neurons of the sciatic nerve).
- Speed of neural impulses: A few to a hundred meters per second.
- Around 10^{11} neurons in the brain, 10^{12} neurons in the entire nervous system. Also, an average a neuron makes thousands of connections to other neurons, resulting in around 10^{14} synapses in the brain.

1.2 A mathematical model for the neuron



(Image taken from our coursebook: S. Haykin, "Neural Networks and Learning Machines," 3rd ed.)

- Input signals are from dendrites of other neurons.
- The synaptic weights correspond to the synaptic strengths: positivity/negativity $\rightarrow$ excitation/inhibition.
- The summing unit models the operation of the cell body (soma).
- The nonlinearity $\varphi(\cdot)$ (activation function) models the axon operation.
- The output may be connected to dendrite of another neuron through another synapse.
- $v_k = \sum_{j=1}^{n} w_{kj} x_j + b_k$ is called the induced local field of neuron k.
- $y_k = \varphi(v_k) = \varphi\left(\sum_{j=1}^{n} w_{kj} x_j + b_k\right)$.
- Alternatively, we may consider a fixed input $x_0 = 1$ with weight $w_{k0} = b_k$:



(Image taken from our coursebook: S. Haykin, "Neural Networks and Learning Machines," 3rd ed.)

- $y_k = \varphi\left(\sum_{j=0}^{n} w_{kj} x_j\right)$. Note that now the summation starts from index 0.

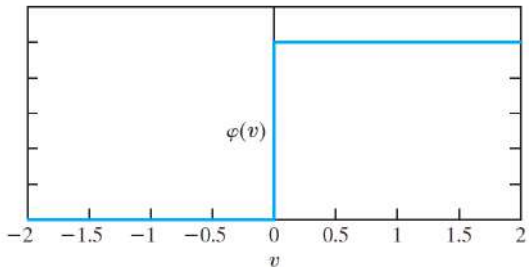
1.3 Types of activation function

Typically, $\varphi(\cdot)$ has bounded image (e.g. $[0, 1]$ or $[-1, 1]$), and thus is also called a squashing function. It limits the amplitude range of the neuron output.

1.3.1 Step function

- Threshold function (or the Heaviside/step function):

$$\varphi(v) = \begin{cases} 1, & v \geq 0 \\ 0, & v < 0 \end{cases}.$$



(Image taken from our coursebook: S. Haykin, "Neural Networks and Learning Machines," 3rd ed.)

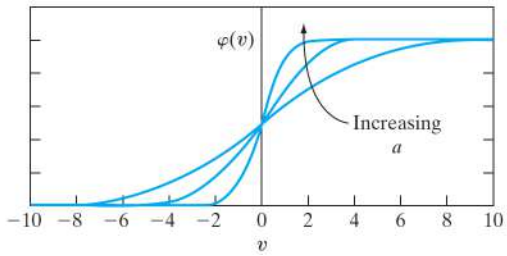
Such a neuron is referred to as the McCulloch-Pitts model (1943).

1.3.2 Sigmoid function

- The sigmoid function is defined as

$$\varphi(v) = \frac{1}{1 + \exp(-av)},$$

where a is called the slope parameter.



(Image taken from our coursebook: S. Haykin, "Neural Networks and Learning Machines," 3rd ed.)

- As $a \to \infty$ the sigmoid function approaches the step function.
- Unlike the step function, the sigmoid function is continuous and differentiable. Differentiability turns out to be a desirable property of an activation function, as we shall see later.

1.3.3 Signum function

$$\varphi(v) = \operatorname{sgn}(v) = \begin{cases} 1, & v \geq 0 \\ 0, & v = 0 \\ -1, & v < 0 \end{cases}.$$

1.3.4 Hyperbolic tangent

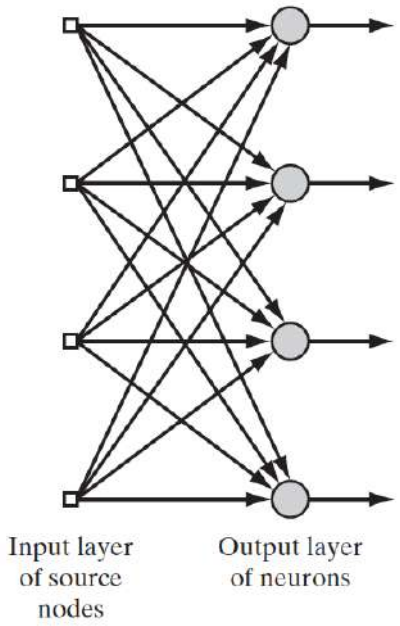
$$\varphi(v) = \tanh(av) = \frac{e^{av} - e^{-av}}{e^{av} + e^{-av}}.$$

for some parameter $a > 0$. Approaches the signum function as $a \to \infty$.

2 Neural network architectures

Having introduced our basic model of a (mathematical) neuron, we now introduce the different neural network architectures that we will keep revisiting throughout the course.

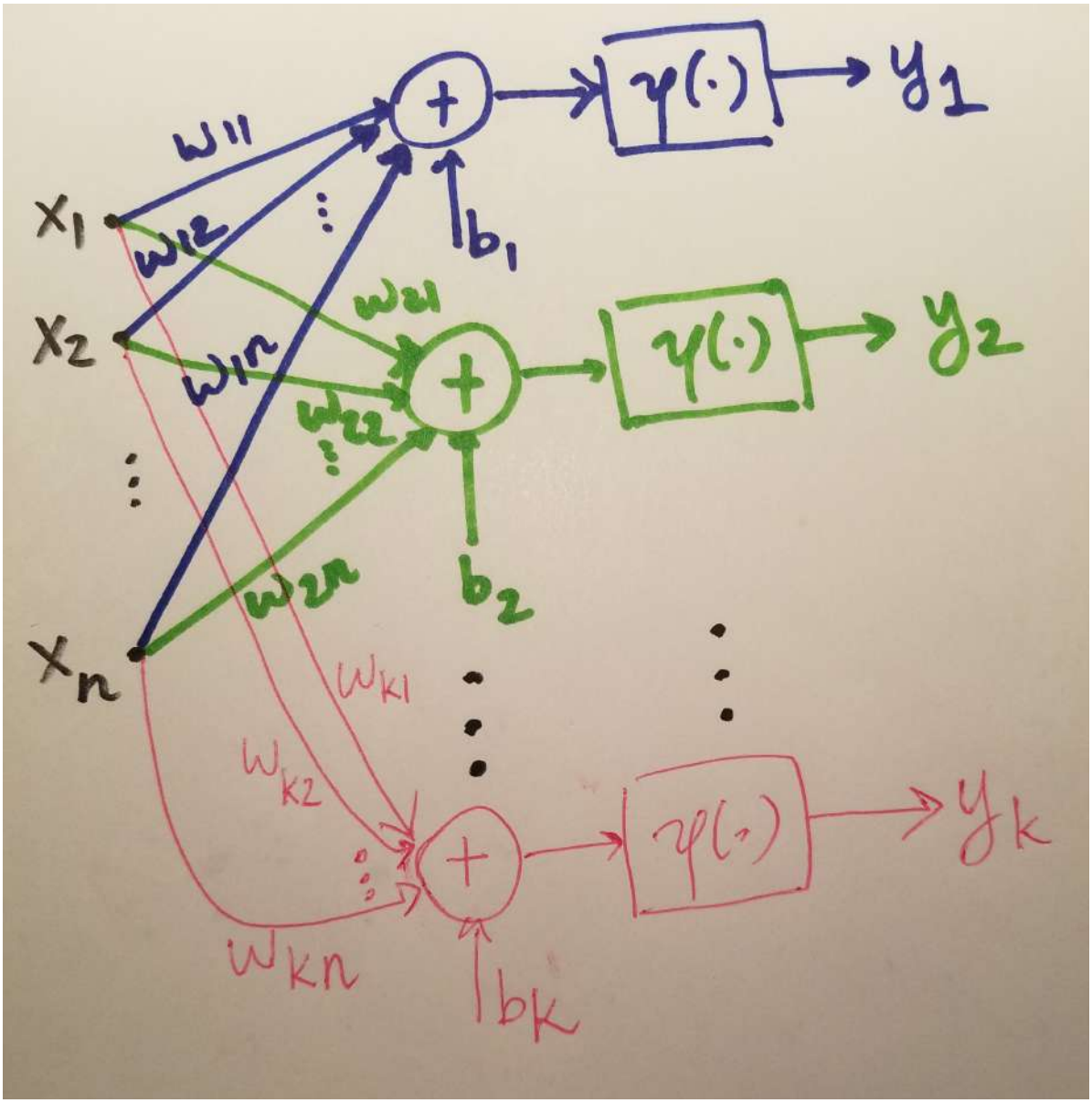
2.1 Single-layer feedforward networks



(Image taken from our coursebook: S. Haykin, "Neural Networks and Learning Machines," 3rd ed.)

Figure 1: Feedforward network with a single layer of neurons

- We just count the number of layers consisting of neurons (not the layer of source nodes as no computation is performed there). Thus, the network in Fig. 1 is called a single-layer network.
- Also, note that the information flow is only over one direction, i.e. the input layer of source nodes project directly onto the output layer of neurons (according to the non-linear transformations as specified by the neurons.). There is no **feedback** of the network's output to network's input. We thus say that the network in Fig. 1 is of feedforward type.
- Let us try to formulate the input-output relationship of the network. Putting in the symbols, we have the following diagram:



We have

$$
\begin{aligned}
y_1 &= \phi\left(b_1 + \sum_{i=1}^{n} w_{1i}x_i\right) \\
y_2 &= \phi\left(b_2 + \sum_{i=1}^{n} w_{2i}x_i\right) \\
&\vdots \\
y_k &= \phi\left(b_k + \sum_{i=1}^{n} w_{ki}x_i\right)
\end{aligned}
$$

We can rewrite all k equations via a single equation:

$$y_j = \phi\left(b_j + \sum_{i=1}^{n} w_{ji}x_i\right),\ j = 1, \ldots, k$$

Here w_{ji} is the weight from input i to Neuron j. We can further rewrite everything in a simple matrix form. Define

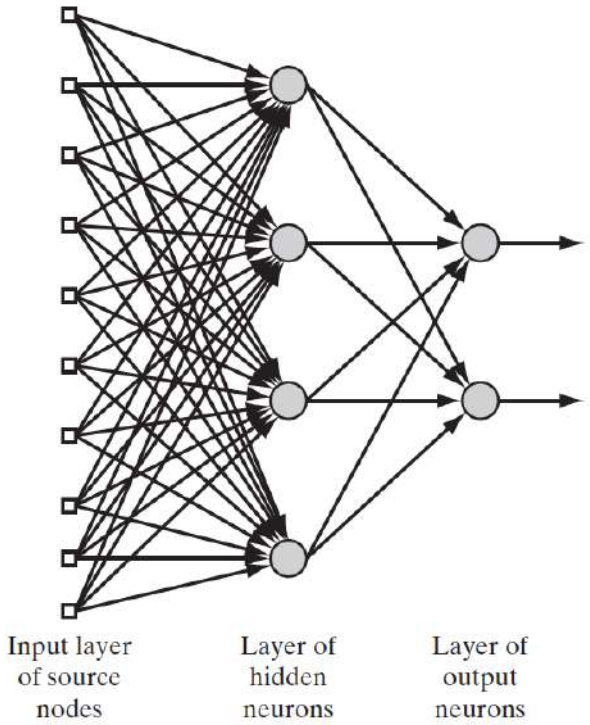
$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix}, \mathbf{W}' = \begin{bmatrix} b_1 & w_{11} & \cdots & w_{1n} \\ b_2 & w_{21} & \cdots & w_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_k & w_{k1} & \cdots & w_{kn} \end{bmatrix}, \mathbf{x}' = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

Then, the above input output relationship can just be written as $\mathbf{y} = \phi(\mathbf{W}'\mathbf{x}')$ in the sense that ϕ is applied component-wise. Sometimes biases are treated separately. Defining

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}, \mathbf{W} = \begin{bmatrix} w_{11} & \cdots & w_{1n} \\ w_{21} & \cdots & w_{2n} \\ \vdots & \vdots & \vdots \\ w_{k1} & \cdots & w_{kn} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

we can write $\mathbf{y} = \phi(\mathbf{W}'\mathbf{x}') = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$.

2.2 Multilayer feedforward networks



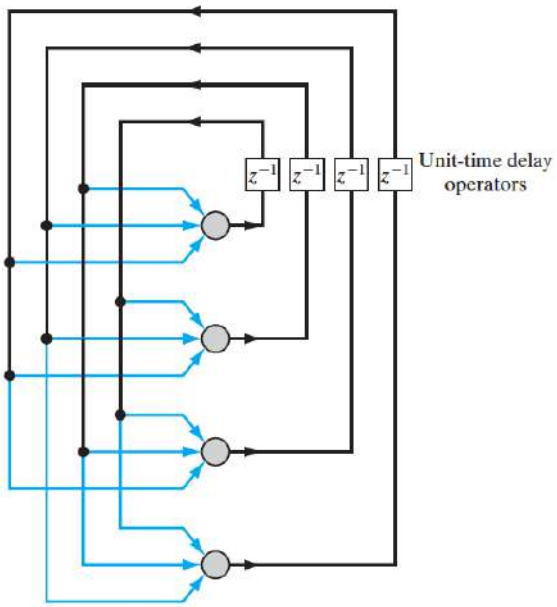
(Image taken from our coursebook: S. Haykin, "Neural Networks and Learning Machines," 3rd ed.)

Figure 2: Fully-connected 2-layer feedforward network with one hidden layer and one output layer.

- We can add more layers to the feedforward network in Fig. 1.
- The end-result is a multilayer network with one or more hidden layers.
- Hidden layers refer to those layers that are not seen directly from either the input or the output of the network.

- We call a network with m source nodes in its input layer, h_1 hidden nodes in its first hidden layer, h_2 nodes in its second hidden layer, ..., h_K nodes in its Kth hidden layer, and finally q nodes in its output layer a m-h_1-h_2-$\cdots$-h_K-q network. For example, the network in Fig. 2 is called a 10-4-2 network as it has 10 source nodes in its input layer, 4 nodes in the hidden layer, and 2 nodes in its output layer.
- Fully-connected network: Every node in each layer of the network is connected to every other node in the adjacent layer. Example: The network is Fig. 2 is fully connected. Otherwise, the network is partially connected.
- Deep network: Many (usually assumed to be > 1) hidden layers. Shallow network: The opposite.
- As will be made more precise later on, theoretically, only one hidden node is sufficient for almost any application provided that one can afford a large number of neurons. On the other hand, a deep network can perform the same tasks as a shallow network with the extra advantage of possibly a fewer number of neurons. Hence, deep networks, provided that they can be properly designed, may be better suited for complex practical applications.
- The input-output relationships may be formulated in a similar manner as the single-layer network discussed previously. For example, consider a two-layer network with n inputs, L_1 neurons in the first layer, and L_2 neurons in the second (output) layer. Let $\mathbf{x} \in \mathbb{R}^{n\times 1}$ be the vector of inputs, $\mathbf{W}_1 \in \mathbb{R}^{L_1\times n}$ be the matrix of weights connecting the input layer to the first layer of neurons (where the ith row jth column of $\mathbf{W}_1$ corresponds to the weight between input node j and neuron i of the first layer), $\mathbf{b}_1 \in \mathbb{R}^{L_1\times 1}$ be the vector of biases for the first layer of neurons, $\mathbf{W}_2 \in \mathbb{R}^{L_2\times L_1}$ be the matrix of weights connecting the first layer of neurons to the second layer of neurons (where the ith row jth column of $\mathbf{W}_2$ corresponds to the weight between neuron j of the first layer and neuron i of the second layer), and $\mathbf{b}_2 \in \mathbb{R}^{L_2\times 1}$ be the vector of biases for the second layer of neurons, and $\mathbf{y} \in \mathbb{R}^{L_2\times 1}$ be the vector of outputs. Then, we have $\mathbf{y} = \phi(\mathbf{W}_2\phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$.

2.3 Recurrent networks



(Image taken from our coursebook: S. Haykin, "Neural Networks and Learning Machines," 3rd ed.)

Figure 3: Recurrent network with no self-feedback loops and no hidden neurons.

- What distinguishes recurrent networks from feedforward networks is that they incorporate **feedback**.
- In Fig. 3, the boxes with z^{-1} represent the unit discrete time delays.
- No self-feedback: The output of a given neuron is not fed back to its own input.
- One may have variations of the structure in Fig. 3. We shall discuss these variations and the details of recurrent networks later on.

ECE/CS 559 - Neural Networks Lecture Notes #3
Some example neural networks

Erdem Koyuncu

1 A concrete example

- We begin with a concrete example.
- We are given the points

$$\{-1,1\}^3 = \{(-1,-1,-1),(-1,-1,1),(-1,1,-1),(-1,1,1),\\ (1,-1,-1),(1,-1,1),(1,1,-1),(1,1,1)\}.$$

- We wish to design the following classifier. Given the vector $(x_1,x_2,x_3) \in \{-1,1\}^3$, we wish to have the output $y=-1$ is the number of -1s in (x_1,x_2,x_3) is greater than the number of 1s. Otherwise, we want to have the output $y=1$.
- We consider the signum function as our activation function:

$$\text{sgn}(x) = \begin{cases} 1, & x>0, \\ 0, & x=0, \\ -1, & x<0. \end{cases}$$

- The following one-neuron "network" accomplishes this task:

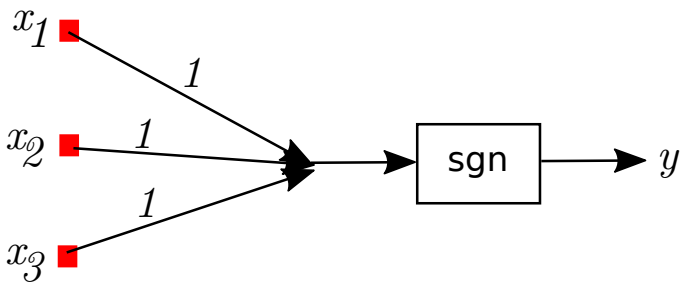


Figure 1: A one-neuron network for finding the component with the highest number occurrences.

- This can be generalized to any dimension of input vectors with components in $\{-1,1\}$ via $y=\text{sgn}(x_1+\cdots+x_n)$. For even dimensions ($n$ even), the network will output a 0 when the number of occurrences of 1 equals the number of occurrences of -1. Otherwise, the network will output the component with the highest number of occurrences.

2 A neural network for pattern classification - Digit classifier

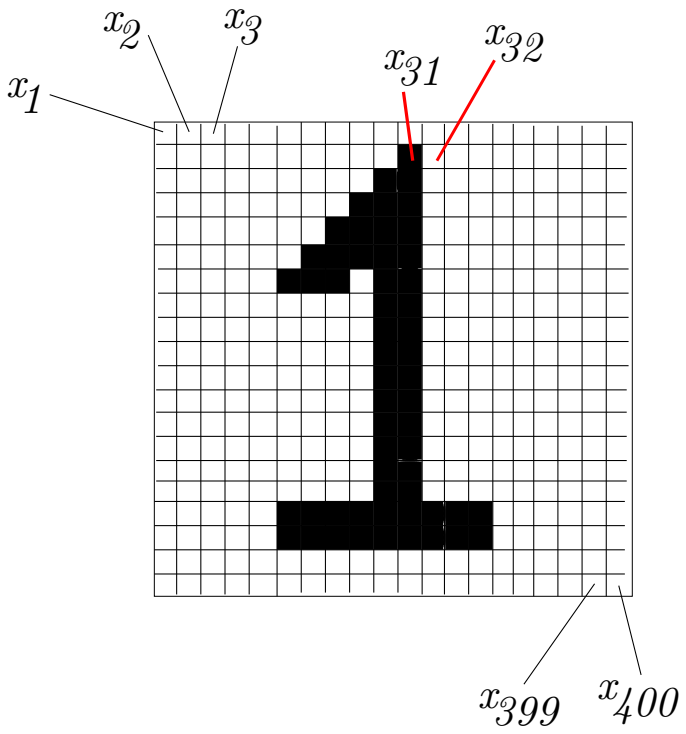


Figure 2: An image of a digit. Each pixel corresponds to one input neuron

- One may design a feedforward neural network for digit classification.
- We are given 20 pixel by 20 pixel images of handwritten digits. Our job is to find, for each given image, the actual digit corresponding to the image. Suppose that each pixel assumes binary values, i.e. the images are black and white only with no grayscale. One of the images may look like in Fig. 2.
- We can then consider a feedforward neural network with 400 nodes in the input layer (labeled as $x_1, x_2, \ldots, x_{400}$), some large number of nodes (say 1000 nodes) in a hidden layer, and 10 nodes in the output layer giving outputs $y_0, \ldots, y_9$. See Fig 3. Ignoring the biases, this fully-connected network has $400 \times 1000 + 1000 \times 10 = 410000$ parameters (synaptic weights) that we can optimize.

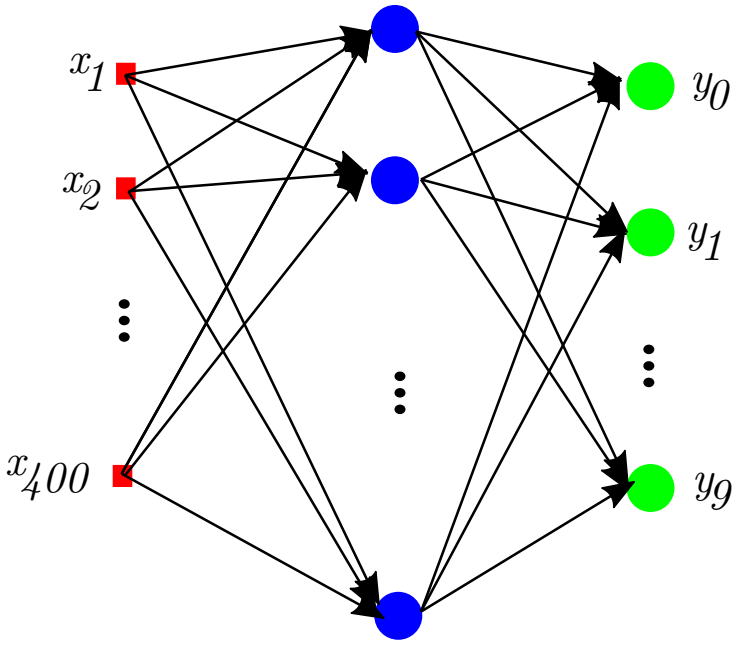


Figure 3: The 2-layer feedforward neural network for digit classification

- Each node in the input layer will correspond to one pixel in the image, as illustrated in Fig. 2. For example, if the image in Fig. 2 is to be fed to the network as input, the inputs would be $x_0 = 0$, $x_1 = 0, \ldots, x_{30} = 0, x_{31} = 1, x_{32} = 0, \ldots, x_{399} = 0, x_{400} = 0$, i.e. we use 0 to represent a white pixel, and we use 1 to represent a black square.
- It is possible to design the network (choose the weights) such that when some image of digit 0 is fed to the network, the output will be $y_0 = 1, y_1 = \ldots, y_9 = 0$. Likewise, for any $i \in \{0, \ldots, 9\}$, when some image of digit i is fed to the network the output will be $y_i = 1$ and $y_j = 0$, $\forall j \neq i$.
- This is usually done via supervised learning. We prepare (or usually acquire!) a large number of 20×20 training images each of which looks like Fig. 2 but of course with different digits, and differently written versions of the same digit. See Fig. 4 for some sample portion of such a training set. Each training image has a label that specifies the actual digit corresponding to the image. For example, in Fig. 4, the images in the first row will be labeled as 0, the second row as 1, and so on.

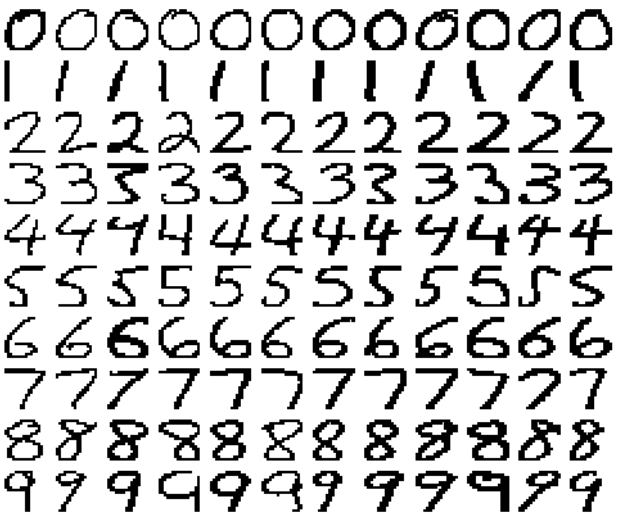


Figure 4: Part of a training set for digit classification

The supervised learning algorithm is then as follows. Usually, we begin with a random initialization of weights. We feed the first training image to the network with label, say, i (i.e. the actual digit that the image corresponds to). We then observe the outputs $y_0, \ldots, y_9$. Note that the desired response for our training image is $y_i^{\text{desired}} = 1$ and $y_j^{\text{desired}} = 0, \forall j \neq i$. We update the weights according to how the outputs $y_0, \ldots, y_9$ differ from the desired outputs that we wanted to have (There are different methods to update the weights, we will discuss these different methods later in the course). We repeat this process many times over all training images. After some point, the weights will converge and the network will be able to classify most of the digit images (even arbitrary images that are not in the set of training images) correctly.

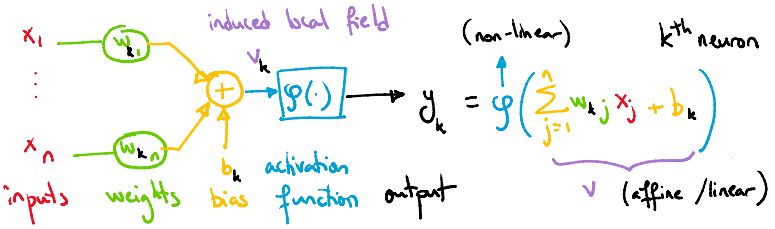
3 Approximation of arbitrary functions

- The key idea of the previous section is that any kind of practical input (image, voice, etc) can be represented digitally as some point in a multidimensional space. For example, each 20×20 image is a point in a 400-dimensional space.
- One can then imagine the existence of a function, the ultimate correct classifier, that maps each point of this 400-dimensional space to the corresponding correct classification, or in general, to some arbitrary point of another multidimensional space. In general, it is extremely hard to determine exactly what the optimal classifier function should be. Instead, we approximate the optimal function via samples of the values assumed by the optimal function (our training set and their labels). Using these samples, we design a neural network that can approximate the highly complicated non-linear optimal classifier function.
- In fact, a neural network with even a single hidden layer can approximate any continuous function to arbitrary accuracy (provided that one can afford as many neurons as needed). This statement will be made more precise later on.
- The same ideas apply to any other application. For example, a sample spoken letter can be modeled as a mere point in another multidimensional space (if it is in digitized audio format). A neural network can then be designed to classify spoken letters. The exact same network model and the learning rules as in digit image classification can be applied to this (seemingly fundamentally different) problem as well.

- Hence, one interesting fact about neural networks is as follows: Our approach and solution to the problem is independent of the "specific nature of the problem." For example, we can use the same network topology and the same learning algorithms to solve both the digit image and spoken letter classification problems. On the other hand, if we were looking for "classical algorithmic" solutions, then an algorithm for digit image classification would be fundamentally different from an algorithm for spoken letter classification; the solution for one problem would not be applicable to the other.

ECE/CS 559 Lecture 3 9/3

Last time: Biological Neuron → Mathematical Neuron



$\varphi(v) = \text{step}(v) = \begin{cases} 1 & v \geq 0 \\ 0 & v < 0 \end{cases}$ | $\text{sign}(v)$

$\varphi(v) = \sigma_a(v) = \frac{1}{1+e^{-av}}$ (sigmoid) | $\tanh(v)$

Also:

$\varphi(v) = \text{ReLU}(v) = \begin{cases} v & v \geq 0 \\ 0 & v < 0 \end{cases}$

Rectified Linear Unit

Linear Algebraic Representation

- All linear operations can be written as inner products!

$$\sum_{j=1}^{n} w_{1j} x_j + b_1 = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + b_1$$

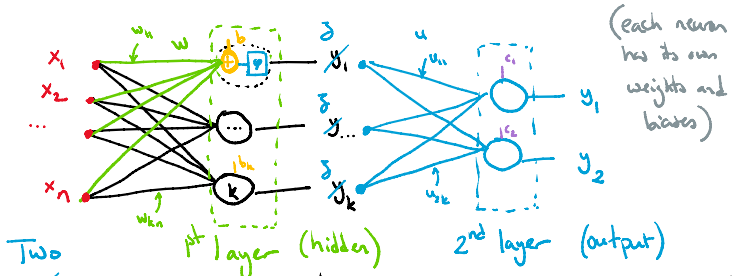
- Repeat this per neuron: $\begin{bmatrix} w_{21}, w_{22}, \cdots, w_{2n} \end{bmatrix} + b_2$; $\begin{bmatrix} w_{k1}, w_{k2}, \cdots, w_{kn} \end{bmatrix} + b_k$

- Single Layer: $y = \varphi(Wx + b)$ (applied per coordinate; matrix-vector multiplication)
- Two Layer:

$$y = \varphi(Uz + c) = \varphi(U\varphi(Wx + b) + c)$$

nested functions

① Feed-Forward Neural Networks:



Two ~~Single~~ Layer Feed Forward Fully Connected Neural Network

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & & w_{2n} \\ \vdots & & & \\ w_{k1} & w_{k2} & & w_{kn} \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_k \end{bmatrix} \quad U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1k} \\ u_{21} & u_{22} & \cdots & u_{2k} \end{bmatrix} \quad c = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

vector | matrix | vector | matrix | vector | vector

"parameters"

- It's important to bookkeep dimensions:

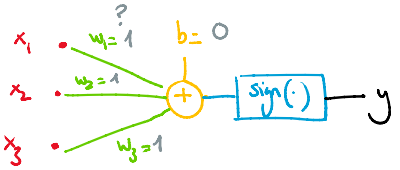
$x \in \mathbb{R}^n \quad W \in \mathbb{R}^{k \times n} \quad b \in \mathbb{R}^k \quad U \in \mathbb{R}^{2 \times k} \quad c \in \mathbb{R}^2 \quad y \in \mathbb{R}^2$

- Layer dimension: (input n), hidden k, output 2
- Naming: This is an n-k-2 fully connected FF network.
- If some weights are zero by design: not fully connected.
  Example: Convolutional Neural Networks:
- If some outputs feed back (after some delay) as inputs: not a feed forward neural network.
  Example: LSTMs (Long Short-Term Memory)

② Examples

(a) Majority operation:

- $n = 3$ inputs, assumed ± 1 $\quad x \in \{-1, +1\}^3$
- Activation function: sign $\quad \varphi(v) = \begin{cases} +1 & v > 0 \\ 0 & v = 0 \\ -1 & v < 0 \end{cases}$ (sign)
- Goal: Design a single neuron that outputs the value that most appears in the input.
  e.g. $x = (-1, -1, -1) \rightarrow y = -1$ $\quad x = (1, 1, -1) \rightarrow y = 1$.



$= \text{sign}(x_1 + x_2 + x_3)$

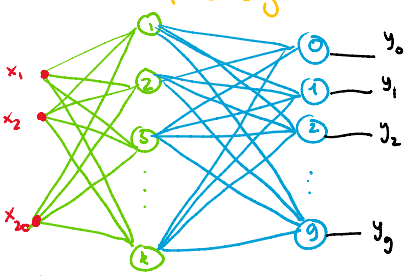
$= \text{majority}(x_1, x_2, x_3)$

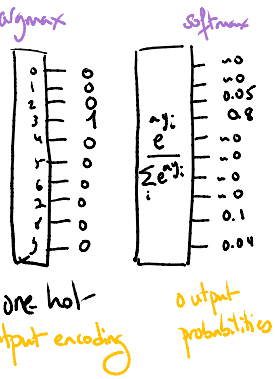
100% accurate.

(b) Logical Statements: Homework 1 + will revisit w/ Perceptrons

(c) Classification: $x =$  $\rightarrow y = 3$ $\quad x =$  $\rightarrow y = 1$

How? Vectorize $n=20$ input encoding





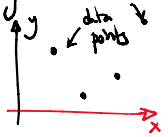
⑧ Approximation Theorems and Learning Algorithms

- Can we always find weights & biases (parameters)?
  - Given enough neurons & layers, almost any function can be approximated.
  - With logic (binary inputs/outputs) the DNF/CNF theorem shows that 2 layers are enough.

- But how do we set these parameters to achieve the desired behavior?
  - Manual adjustments → intractable except in simple cases
  - Instead, learn using guidance from data.

- This is not that different than fitting a polynomial.
- Data ( , 3) ( , 1) , etc.

  x y (target output)



$\varphi(u\varphi(Ax+b)+c) = \hat{y}(x)$ (neural network output)

Goal: minimize # of mistakes: $\sum_{\text{data}} \mathbb{1}\{\hat{y}(x) \neq y\}$

- Typical Algorithm processes data gradually:

epoch = 1 pass over data

1. Initialize weights randomly
2. For each (x, y) in data:
   - Feed x to NN and get $\hat{y}(x)$
   - Update weights to bring $\hat{y}(x)$ closer to y.
3. Repeat 2 until weights converge

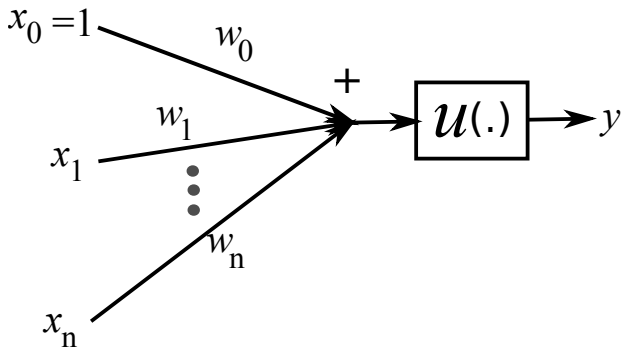
$(y, \hat{y})$ gap = loss

ECE/CS 559 - Neural Networks Lecture Notes #4
The perceptron and its training

Erdem Koyuncu

1 Rosenblatt's percepton

- Simplest case of a neural "network" consisting of one neuron with the McCulloch-Pitts model (step activation function).
- Can classify patterns that are linearly separable.
- Can learn to classify patterns. This is done by training the perceptron with labeled samples from each class.
- Recall that learning from labeled samples is called supervised learning.
- We typically use the symbol u for the step activation function. In other words, we define $u(x) = 1$ if $x \geq 0$, and otherwise, we define $u(x) = 0$ if $x < 0$.
- The block diagram of the perceptron is then

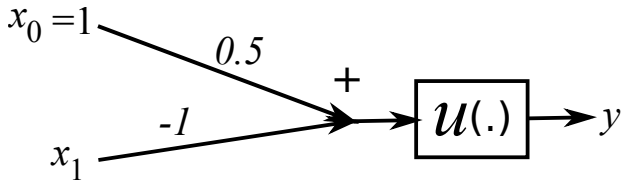


- The input output relationships are $y = u(\sum_{i=0}^{n} x_i w_i)$.

2 What can be done with the perceptron?

2.1 Logical NOT

- Input $x_1 \in \{0, 1\}$ with a 0 representing a FALSE, and 1 representing a TRUE as usual...
- We want $y = \text{NOT}(x_1)$ (sometimes also written $y = \overline{x_1}$). In other words, we want $y = \begin{cases} 1, & x_1 = 0 \\ 0, & x_1 = 1 \end{cases}$
- It turns out that the choices $w_0 = 0.5$ and $w_1 = -1$ work, i.e. we have the perceptron:



- $y = u(-x_1 + 0.5)$.
- Test: Feed $x_1 = 0$. We have $y = u(-0 + 0.5) = u(0.5) = 1$. OK.
- Test: Feed $x_1 = 1$. We have $y = u(-1+0.5) = u(-0.5) = 0$. OK. All desired input output relationships are satisfied. So, the perceptron works as intended.

2.2 Logical AND

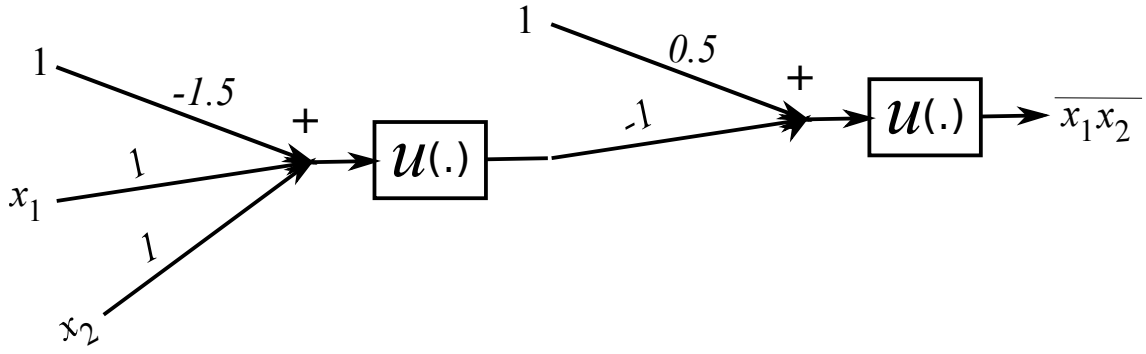
- We now have two inputs x_1 and x_2. We want $y = AND(x_1, x_2)$ or with a shorthand notation $y = x_1 x_2$. The following are the input output relationships we want:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

- Are there weights w_0, w_1, w_2 that satisfy the desired input output relationships?
- One option is again to guess to find appropriate weights.
- More systematically, one can determine the correct weights through solving linear inequalities. In our case,
- Given $x_1 = x_2 = 0$, we want the output $y = u(w_0)$ to be equal to 0. This yields the inequality $w_0 < 0$.
- Given $x_1 = 0$, $x_2 = 1$, we want $y = u(w_0 + w_2) = 0$. This yields $w_0 + w_2 < 0$.
- Given $x_1 = 1$, $x_2 = 0$, we want $y = u(w_0 + w_1) = 0$. This yields $w_0 + w_1 < 0$.
- Given $x_1 = x_2 = 1$, we want $y = u(w_0 + w_1 + w_2) = 1$. This yields $w_0 + w_1 + w_2 \geq 0$.
- One can formally solve this system of 4 inequalities. Actually, by inspection, it is not difficult to see that $w_0 = -\frac{3}{2}$, $w_1 = w_2 = 1$ is a solution.
- In general, for n inputs, setting $w_0 = -n + \frac{1}{2}$, $w_1 = \cdots = w_n = 1$ will implement the n-input AND gate, i.e. $y = x_1 x_2 \cdots x_n$.

2.3 Building a digital computer using perceptrons

- We have the NOT, we have the AND, which means we can get the NAND by cascading the NOT and the AND:



- But the NAND gate is universal in the sense that every possible logic operation over an arbitrary number of inputs can be implemented using NAND gates only.
- One can thus build a computer using a sufficiently large number of perceptrons.

2.4 Logical OR

- We now have two inputs x_1 and x_2. We want $y = OR(x_1, x_2)$ or with a shorthand notation $y = x_1 + x_2$. The following are the input output relationships we want:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

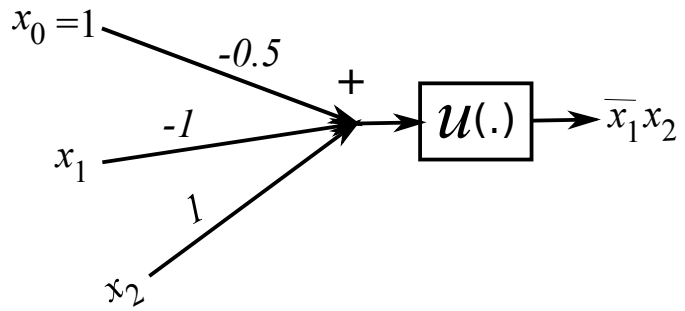
- We have the 4 inequalities corresponding to the 4 cases:
- $w_0 < 0$.
- $w_0 + w_2 \geq 0$.
- $w_0 + w_1 \geq 0$.
- $w_0 + w_1 + w_2 \geq 0$.
- By inspection, it is not difficult to see that $w_0 = -\frac{1}{2}$, $w_1 = w_2 = 1$ is a solution.
- In general, for n inputs, setting $w_0 = -\frac{1}{2}$, $w_1 = \cdots = w_n = 1$ will implement the n-input OR gate, i.e. $y = x_1 + x_2 + \cdots + x_n$.

2.5 Any logical product term

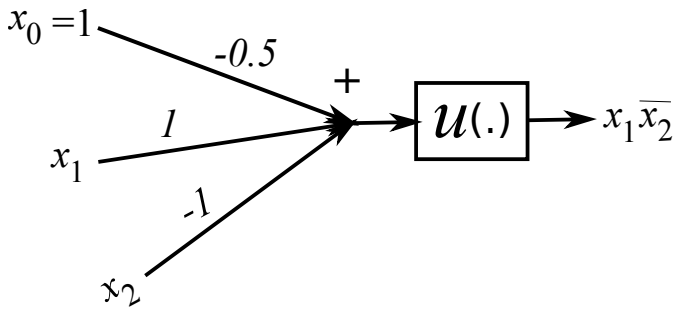
- Implementing the gate $y = \overline{x_1} \cdots \overline{x_n} x_{n+1} \cdots x_{n+m}$.
- By inspection, $w_0 = -m + \frac{1}{2}$, $w_1 = \cdots = w_n = -1$, and $w_{n+1} = \cdots = w_{n+m} = 1$ works.
- Actually this generalizes the NOT and AND gate examples earlier.
- Recall that any logic function can be implemented in a sum of products form. For example, suppose we want to implement the logic function of two variables with the following truth table (this is known as the XOR gate or the modulo-two addition):

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

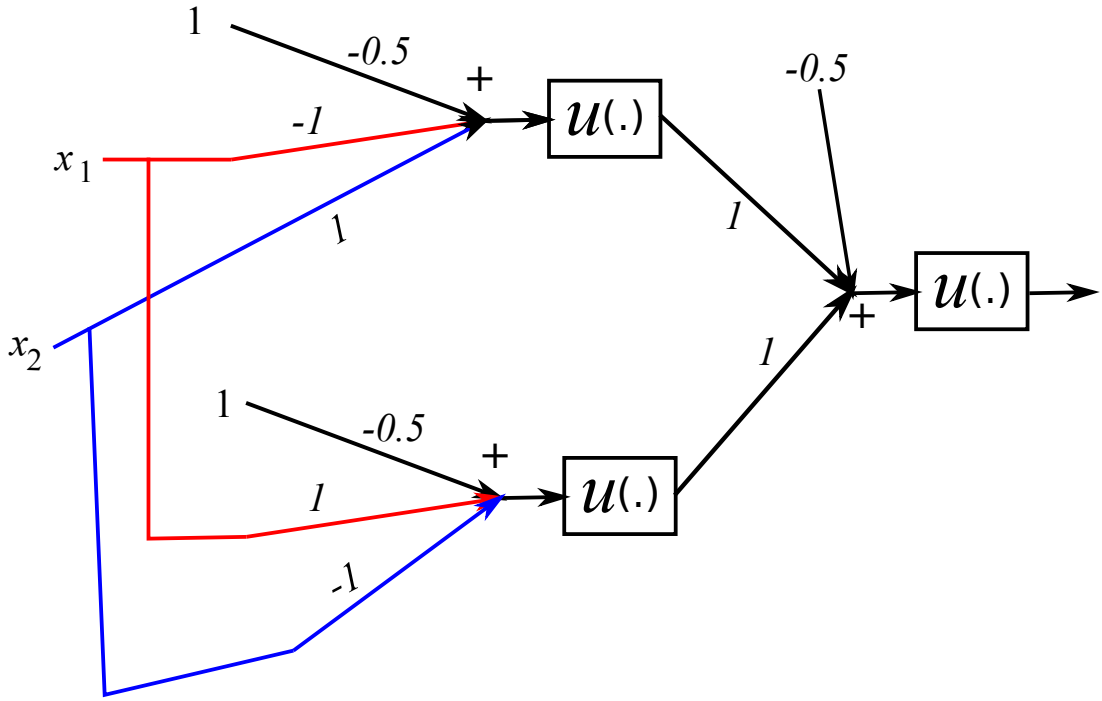
- We consider the inputs for which the output is equal to 1. The first set of inputs that provide an output of 1 are $x_1 = 0$ and $x_2 = 1$. The corresponding product term is $\overline{x_1} x_2$. The second set of inputs that provide an output of 1 are $x_1 = 1$ and $x_2 = 0$. The corresponding product term is $x_1 \overline{x_2}$. The function we are looking for is thus $y = \overline{x_1} x_2 + x_1 \overline{x_2}$ which is in the sum of products form.
- We know how to implement each product term, for example the following network provides the output $\overline{x_1} x_2$:



- On the other hand, the following network provides the output $x_1\overline{x_2}$:



- So put these two subnetworks in the first layer, and then OR them in the second layer to get the overall network that performs $y = \overline{x_1}x_2 + x_1\overline{x_2}$:



- This way, any arbitrary logic function over any number of inputs can be implemented using two layers only.

3 What cannot be done with a perceptron?

- Let us try implementing the XOR operation whose truth table was also provided above:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

(We know how to implement XOR using two layers of perceptrons with a total of 3 perceptrons. Here we want to implement the XOR using just a single perceptron).

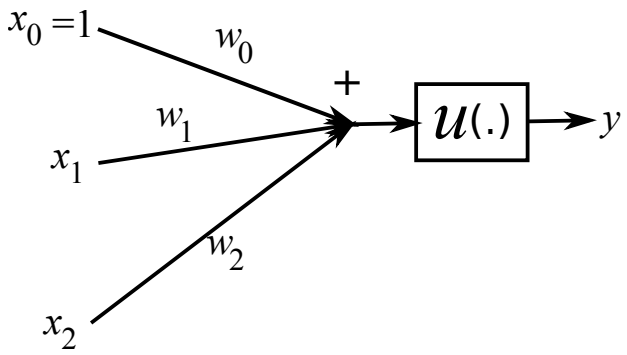
- The following inequalities should be satisfied:

$$w_0 < 0 \tag{1}$$
$$w_0 + w_2 \geq 0 \tag{2}$$
$$w_0 + w_1 \geq 0 \tag{3}$$
$$w_0 + w_1 + w_2 < 0 \tag{4}$$

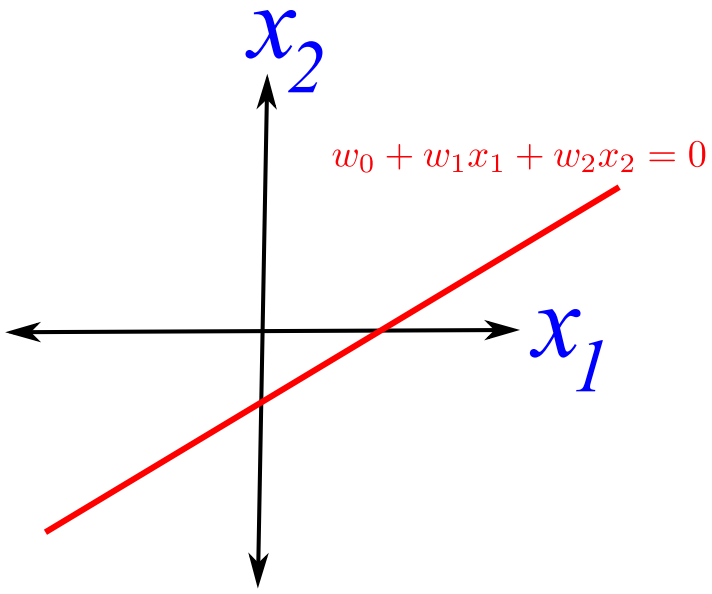
- $-(1) + (2) + (3) - (4)$, we get $0 > 0$, a contradiction! Which means, no choice of weights w_0, w_1, w_2 can satisfy the equations above. Or, a single perceptron, by itself, cannot implement the XOR gate.
- So, what is going on?

4 Geometric interpretation of the perceptron

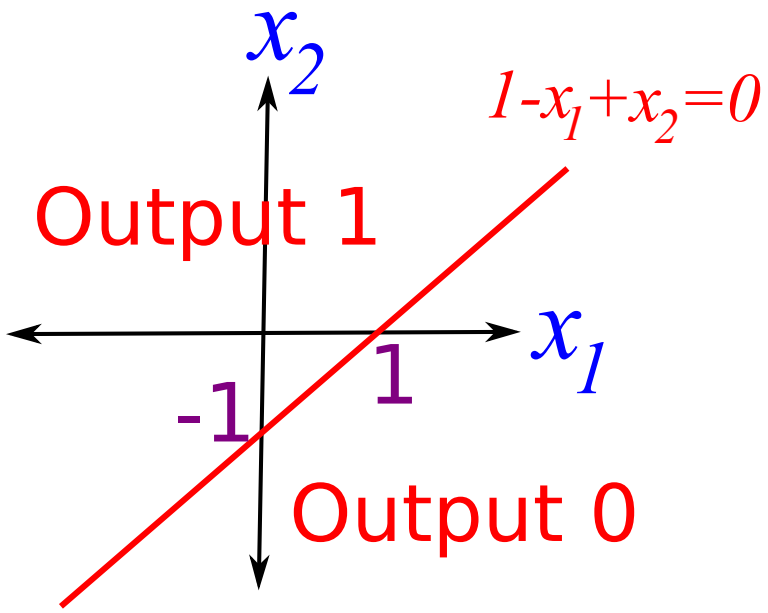
- Suppose we only have two inputs:



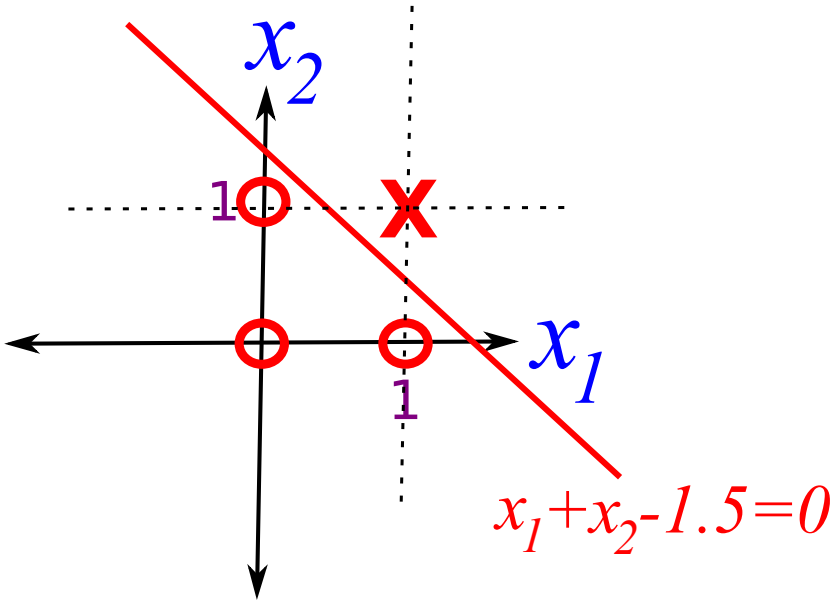
- We have $y = \begin{cases} 0, & w_0 + w_1 x_1 + w_2 x_2 < 0 \\ 1, & w_0 + w_1 x_1 + w_2 x_2 \geq 0 \end{cases}$
- Hence, the perceptron divides the input space $\mathbb{R}^2$ (the set of all x_1-x_2 pairs) to two disjoint subsets via the line $w_0 + w_1 x_1 + w_2 x_2 = 0$. It looks like as follows:



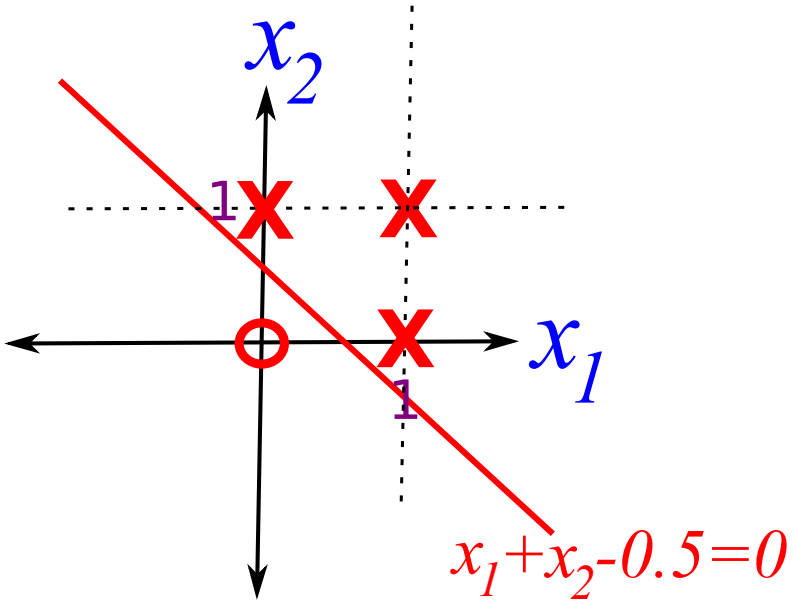
- Depending on the signs of w_1, w_2 for one side of the line, the output will be 0, and for the other side of the line, the output will be 1 (the separator always results in an output of 1 as $u(0) = 1$ by definition). For example, if $w_2 > 0$ the (x_1, x_2) pairs on the top of the line will result in an output of 1, and if $w_2 < 0$ the points on the bottom of the line will output 1.
- For example, if $w_0 = 1$, $w_1 = -1$, $w_2 = 1$, the separator is $1 - x_1 + x_2 = 0$.



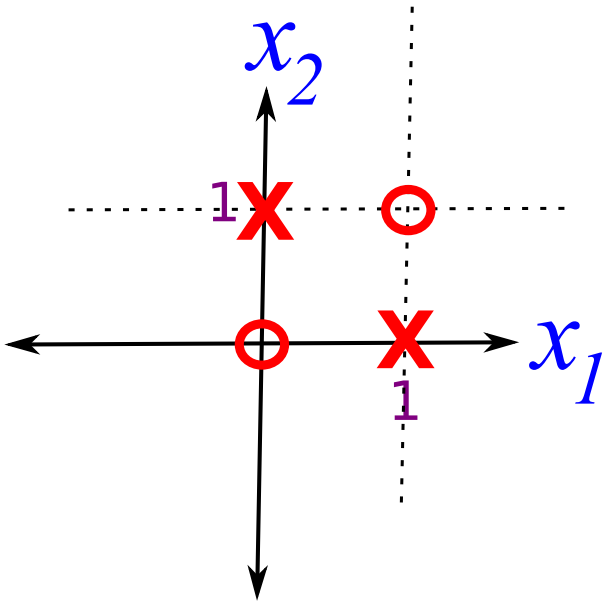
- If the signs of all weights are reversed, we would have the same separator, but now instead, the bottom of the separator would result in an output of 1.
- In general, for a perceptron with n inputs, the separator is a hyperplane, i.e. an $n-1$ dimensional subspace of $\mathbb{R}^n$. For example, for $n = 1$, you can easily verify that the separator is a mere point.
- In general, we observe that the classes should be linearly separable for the existence of a perceptron that can ultimately classify them. For the AND gate, the set of points and their desired respective classes, and the separator we found look as follows: We use an O to represent an output of 0, and an X to represent an output of 1.



For the OR gate example, the geometry looked like as follows:



- For the XOR gate, the classes we would like to separate look like as follows.



It should be clear that no line can separate the two classes. This serves as a "geometric proof" of the fact that the XOR gate cannot be implemented via a single perceptron.

5 Learning Algorithm for the Perceptron

- So, from now on, suppose we have two classes that are linearly separable. That is to say, let $\mathcal{C}_0$ and $\mathcal{C}_1$ be two subsets of $\mathbb{R}^{1+d}$ (column vectors) such that the first component of all vectors in $\mathcal{C}_0$ and $\mathcal{C}_1$ are equal to 1 (this is to introduce the bias term). We say that the classes $\mathcal{C}_0$ and $\mathcal{C}_1$ are linearly separable

if there exists a weight vector $\Omega \in \mathbb{R}^{1+d}$ such that

$$\Omega^T \mathbf{x} \geq 0 \text{ for every } \mathbf{x} \in \mathcal{C}_1, \text{ and}$$
$$\Omega^T \mathbf{x} < 0 \text{ for every } \mathbf{x} \in \mathcal{C}_0.$$

Here $(\cdot)^T$ represents the matrix transpose.

- Now suppose Ω is a weight vector that can separate the classes $\mathcal{C}_0$ and $\mathcal{C}_1$ as described above. Given $\Omega = (w_0 \cdots w_n)^T$, a d-input perceptron with the step activation function, weights $w_1, \ldots, w_d$, and bias w_0 will output a 1 if the input is any vector that belongs to class $\mathcal{C}_1$. Otherwise, for any input vector that belongs to class $\mathcal{C}_0$, the perceptron will output a 0.
- Given that $\mathcal{C}_0$ and $\mathcal{C}_1$ are linearly separable, how to find a weight vector $\mathbf{w}$ that can separate them?
- **Option 1:** Write all the $|\mathcal{C}_0| + |\mathcal{C}_1|$ inequalities and solve them: Not a very good idea if the classes are large.
- **Option 2:** Instead, we use the Perceptron Training Algorithm (PTA) that finds the weight for us. It is one special case of supervised learning.
- Suppose n training samples $x_1, \ldots, x_n \in \mathbb{R}^{1+d}$ are given. Again, the first component of these vectors are assumed to be equal to 1 to take into account the bias. Let $d_1, \ldots, d_n \in \{0, 1\}$ represent the desired output for each of the input vectors. With this notation, we have $\mathcal{C}_0 = \{\mathbf{x}_i : d_i = 0\}$, and $\mathcal{C}_1 = \{\mathbf{x}_i : d_i = 1\}$. Suppose $\mathcal{C}_0$ and $\mathcal{C}_1$ are linearly separable. Consider any learning parameter $\eta > 0$. Then, the following algorithm finds a weight vector $\Omega \in \mathbb{R}^{1+d}$ that can separate the two classes:

  1. Initialize Ω arbitrarily (e.g. randomly).
  2. `epochNumber` $\leftarrow 0$.
  3. While Ω cannot correctly classify all input patterns, i.e. while $u(\Omega^T \mathbf{x}_i) \neq d_i$ for some $i \in \{1, \ldots, n\}$,
     (a) `epochNumber` $\leftarrow$ `epochNumber` $+ 1$.
     (b) for $i = 1$ to n
        i. Calculate $y = u(\Omega^T x_i)$ (the output for the ith training sample with the current weights).
        ii. If $y = 1$ but $d_i = 0$,
           A. Update the weights as $\Omega \leftarrow \Omega - \eta \mathbf{x}_i$.
        iii. If $y = 0$ but $d_i = 1$,
           A. Update the weights as $\Omega \leftarrow \Omega + \eta \mathbf{x}_i$.

- Of course the variable epochNumber really does not do anything and it is just there for tracking purposes. We can also write the entire algorithm in a considerably simpler form:

  1. Initialize Ω arbitrarily (e.g. randomly).
  2. While Ω cannot correctly classify all input patterns, i.e. while $u(\Omega^T \mathbf{x}_i) \neq d_i$ for some $i \in \{1, \ldots, n\}$,
     (a) for $i = 1$ to n
        i. $\Omega \leftarrow \Omega + \eta \mathbf{x}_i (d_i - u(\Omega^T \mathbf{x}_i))$.

- Of course, the "while" conditioning can also be optimized - you can figure out those optimizations yourself.
- The parameter η is usually referred to as the learning rate parameter, or simply, the learning parameter.
  - Small η: The effects of the past inputs are more pronounced, fluctuations of the weights are lesser in magnitude, may take longer to converge.

– Large η: Faster adaptations to any possibly changes in the underlying classes. May also take longer to converge (this time because of large fluctuations).

- Nevertheless, if the input classes are linearly separable, then the PTA converges for any $\eta > 0$ (and thus results in a weight vector that can separate the two classes).

- **Proof:** I am providing a proof here as the proofs I found in different references were at places incomplete/flawed. We will provide a proof for the initialization $\Omega = \Omega_0 = 0$ (all zero vector) and $\eta = 1$. The proof can be generalized to different initializations and other η.

  Let Ω_{opt} be a vector of weights that can separate the two classes (existence is guaranteed as the classes are assumed to be linearly separable). We have

$$\mathbf{x}^T\Omega_{\text{opt}} \geq 0 \text{ for every } \mathbf{x} \in \mathcal{C}_1, \text{ and}$$
$$\mathbf{x}^T\Omega_{\text{opt}} < 0 \text{ for every } \mathbf{x} \in \mathcal{C}_0.$$

  Step-1: In fact, we can assume, without loss of generality, that

$$\mathbf{x}^T\Omega_{\text{opt}} > 0 \text{ for every } \mathbf{x} \in \mathcal{C}_1,$$

  i.e. we have strict inequality. To see this, let $\delta = \min_{\mathbf{x}\in\mathcal{C}_0} |\mathbf{x}^T\Omega_{\text{opt}}|$. Then, for

$$\Omega'_{\text{opt}} = \Omega_{\text{opt}} + \frac{\delta}{2}[1 \quad 0 \quad \cdots \quad 0]^T,$$

  we have, for every $\mathbf{x} \in \mathcal{C}_1$,

$$\mathbf{x}^T\Omega'_{\text{opt}} = \mathbf{x}^T\Omega_{\text{opt}} + \frac{\delta}{2}\mathbf{x}^T[1 \quad 0 \quad \cdots \quad 0]^T = \underbrace{\mathbf{x}^T\Omega_{\text{opt}}}_{\geq 0} + \underbrace{\frac{\delta}{2}}_{>0} > 0,$$

  and if $\mathbf{x} \in \mathcal{C}_0$, we have

$$\mathbf{x}^T\Omega'_{\text{opt}} = \underbrace{\mathbf{x}^T\Omega_{\text{opt}}}_{\leq -\delta} + \frac{\delta}{2} \leq -\frac{\delta}{2} < 0.$$

  Hence, Ω'_{opt} is also a correct classifier.

  Step-2: Recall that we show the samples in the order $\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{x}_1, \ldots, \mathbf{x}_n, \ldots$. Let $j_1, j_2, \ldots, \in \{1, \ldots, n\}$ represent the sequence of indices at which we update the neuron weights. Without of loss generality, we assume $j_i = i$. Let Ω_i denote the neuron weights after the ith update. Recall that the initial weights we choose are $\Omega_0 = 0$. We have

$$\Omega_1 = \begin{cases} \Omega_0 + \mathbf{x}_1, \ \Omega_{\text{opt}}^T\mathbf{x}_1 \geq 0, \ \Omega_0^T\mathbf{x}_1 < 0 \\ \Omega_0 - \mathbf{x}_1, \ \Omega_{\text{opt}}^T\mathbf{x}_1 < 0, \ \Omega_0^T\mathbf{x}_1 \geq 0 \end{cases}$$
$$\Omega_2 = \begin{cases} \Omega_1 + \mathbf{x}_2, \ \Omega_{\text{opt}}^T\mathbf{x}_2 \geq 0, \ \Omega_1^T\mathbf{x}_2 < 0 \\ \Omega_1 - \mathbf{x}_2, \ \Omega_{\text{opt}}^T\mathbf{x}_2 < 0, \ \Omega_1^T\mathbf{x}_2 \geq 0 \end{cases}$$
$$\vdots$$
$$\Omega_N = \begin{cases} \Omega_{N-1} + \mathbf{x}_N, \ \Omega_{\text{opt}}^T\mathbf{x}_N \geq 0, \ \Omega_{N-1}^T\mathbf{x}_N < 0 \\ \Omega_{N-1} - \mathbf{x}_N, \ \Omega_{\text{opt}}^T\mathbf{x}_N < 0, \ \Omega_{N-1}^T\mathbf{x}_N \geq 0 \end{cases}$$

  Why do you have these updates? For example, in order to have $\Omega_1 = \Omega_0 + \mathbf{x}_1$, the desired output given $\mathbf{x}_1$ should be 1 (this is provided by the condition $\Omega_{\text{opt}}^T\mathbf{x}_1 \geq 0$), but the actual output given $\mathbf{x}_1$ with the current weights Ω_0 should be 0 (this is provided by the condition $\Omega_0^T\mathbf{x}_1 < 0$). Similar with others.

  Take the first equality and multiply both sides by Ω_{opt}^T, we obtain:

$$\Omega_{\text{opt}}^T\Omega_1 = \begin{cases} \Omega_{\text{opt}}^T\Omega_0 + \Omega_{\text{opt}}^T\mathbf{x}_1, \ \Omega_{\text{opt}}^T\mathbf{x}_1 \geq 0, \ \Omega_0^T\mathbf{x}_1 < 0 \\ \Omega_{\text{opt}}^T\Omega_0 - \Omega_{\text{opt}}^T\mathbf{x}_1, \ \Omega_{\text{opt}}^T\mathbf{x}_1 < 0, \ \Omega_0^T\mathbf{x}_1 \geq 0 \end{cases}$$

The first conditions tell you that you can just write this in the considerably simpler form:

$$\Omega_{\text{opt}}^T \Omega_1 = \Omega_{\text{opt}}^T \Omega_0 + |\Omega_{\text{opt}}^T \mathbf{x}_1|.$$

Doing the same for all equalities, and using the fact that $\Omega_0 = 0$, we obtain

$$\begin{aligned}
\Omega_{\text{opt}}^T \Omega_1 &= \Omega_{\text{opt}}^T \Omega_0 + |\Omega_{\text{opt}}^T \mathbf{x}_1| = |\Omega_{\text{opt}}^T \mathbf{x}_1| \\
\Omega_{\text{opt}}^T \Omega_2 &= \Omega_{\text{opt}}^T \Omega_1 + |\Omega_{\text{opt}}^T \mathbf{x}_2| = |\Omega_{\text{opt}}^T \mathbf{x}_1| + |\Omega_{\text{opt}}^T \mathbf{x}_2| \\
&\vdots \\
\Omega_{\text{opt}}^T \Omega_N &= \sum_{i=1}^{N} |\Omega_{\text{opt}}^T \mathbf{x}_i|
\end{aligned}$$

From Step 1, recall that $|\Omega_{\text{opt}}^T \mathbf{x}_i| > 0$ for every $i \in \{1, \ldots, n\}$. Therefore, letting

$$\alpha = \min_{i \in \{1,\ldots,n\}} |\Omega_{\text{opt}}^T \mathbf{x}_i|,$$

we obtain

$$\Omega_{\text{opt}}^T \Omega_N \geq N\alpha$$

We now recall Cauchy-Schwarz inequality: For vectors $\mathbf{x}, \mathbf{y}$, we have $\mathbf{x}^T\mathbf{y} \leq \|\mathbf{x}\|\|\mathbf{y}\|$. In particular, $\Omega_{\text{opt}}^T \Omega_N \leq \|\Omega_{\text{opt}}\|\|\Omega_N\|$ so that

$$\|\Omega_N\|^2 \geq \frac{N^2\alpha^2}{\|\Omega_{\text{opt}}\|^2} \tag{5}$$

Step-3: Recall the first equality in the beginning of Step 2:

$$\Omega_1 = \begin{cases} \Omega_0 + \mathbf{x}_1,\ \Omega_{\text{opt}}^T \mathbf{x}_1 \geq 0,\ \Omega_0^T \mathbf{x}_1 < 0 \\ \Omega_0 - \mathbf{x}_1,\ \Omega_{\text{opt}}^T \mathbf{x}_1 < 0,\ \Omega_0^T \mathbf{x}_1 \geq 0 \end{cases}$$

Taking the squared Euclidean norms of both sides and expanding:

$$\|\Omega_1\|^2 = \begin{cases} \|\Omega_0\|^2 + \|\mathbf{x}_1\|^2 + 2\Omega_0^T \mathbf{x}_1,\ \Omega_{\text{opt}}^T \mathbf{x}_1 \geq 0,\ \Omega_0^T \mathbf{x}_1 < 0 \\ \|\Omega_0\|^2 + \|\mathbf{x}_1\|^2 - 2\Omega_0^T \mathbf{x}_1,\ \Omega_{\text{opt}}^T \mathbf{x}_1 < 0,\ \Omega_0^T \mathbf{x}_1 \geq 0 \end{cases}$$

Thus either way,

$$\|\Omega_1\|^2 = \|\Omega_0\|^2 + \|\mathbf{x}_1\|^2 - 2|\Omega_0^T \mathbf{x}_1| \leq \|\Omega_0\|^2 + \|\mathbf{x}_1\|^2$$

and likewise from the other equalities, we obtain

$$\begin{aligned}
\|\Omega_2\|^2 &\leq \|\Omega_1\|^2 + \|\mathbf{x}_2\|^2 \\
&\vdots \\
\|\Omega_N\|^2 &\leq \|\Omega_{N-1}\|^2 + \|\mathbf{x}_N\|^2
\end{aligned}$$

Summing all inequalities up, we obtain

$$\|\Omega_N\|^2 \leq \sum_{i=1}^{N} \|\mathbf{x}_i\|^2 \leq \beta N \tag{6}$$

where

$$\beta = \max_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i\|^2$$

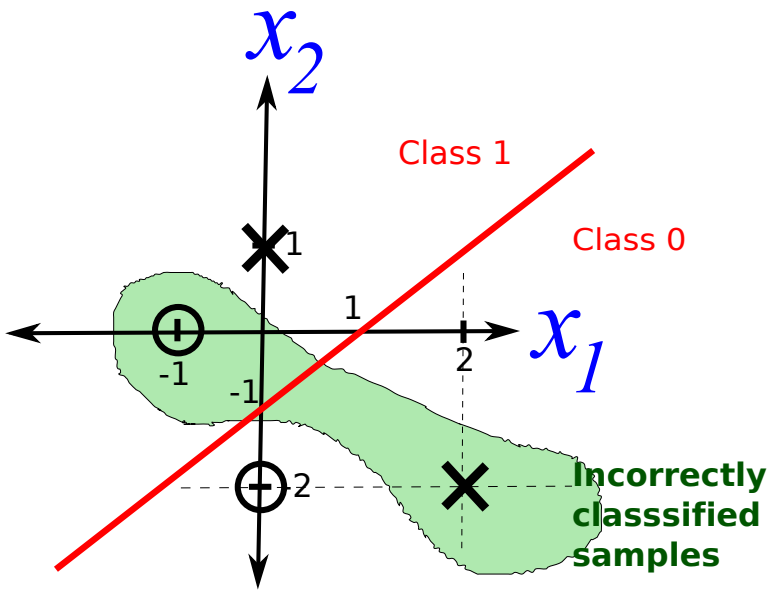
Step-4: Combining (5) and (6),

$$N \leq \frac{\beta \|\Omega_{\text{opt}}\|^2}{\alpha^2}$$

This means that the number of updates is finite, and thus the PTA converges. □

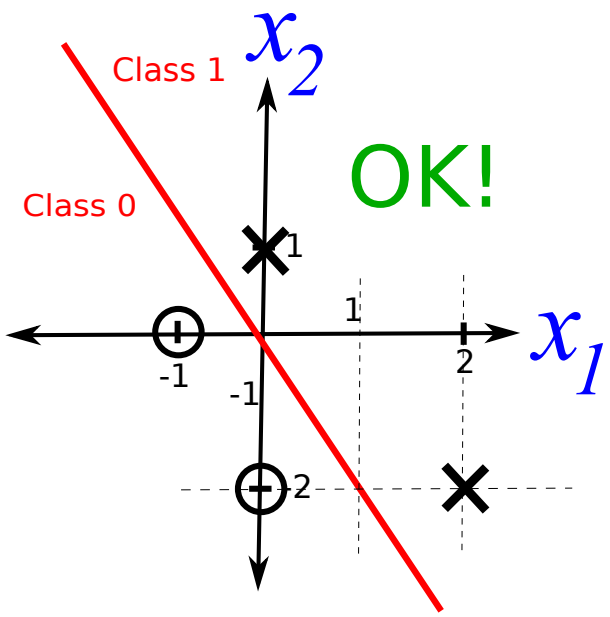
6 An Example

Given initial weights $\Omega_0 = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$, training set (without 1s for the biases) $\begin{bmatrix} 2 \\ -2 \end{bmatrix}, \begin{bmatrix} 0 \\ -2 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}$ with the desired outputs $1, 0, 1, 0$.



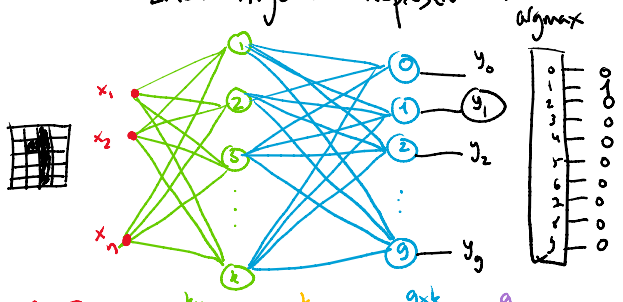
Initial weights cannot correctly classify the samples/patterns. So, let $\mathbf{x}_1 = \begin{bmatrix} 1 \\ 2 \\ -2 \end{bmatrix}, \mathbf{x}_2 = \begin{bmatrix} 1 \\ 0 \\ -2 \end{bmatrix}, \mathbf{x}_3 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \mathbf{x}_4 = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$ (I put the 1s for the biases).

- Epoch 1
  - We feed $\mathbf{x}_1$. $\Omega_0^T \mathbf{x}_1 = -3$, which implies an output of $u(-3) = 0$. But the desired output is 1. So, we update $\Omega_1 = \Omega_0 + \mathbf{x}_1 = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}$.
  - We feed $\mathbf{x}_2$. Now we use the updated weights: $\Omega_1^T \mathbf{x}_2 = 4$, which implies an output of $u(4) = 1$. But the desired output is 0. So, we update $\Omega_2 = \Omega_1 - \mathbf{x}_2 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.
  - We feed $\mathbf{x}_3$. We have $u(\Omega_2^T \mathbf{x}_3) = 1$. The desired output is also 1. So, no update on the weights, or $\Omega_3 = \Omega_2$.
  - We feed $\mathbf{x}_4$. We have $u(\Omega_3^T \mathbf{x}_4) = 0$. But the desired output is 0. So, we update $\Omega_4 = \Omega_3 - \mathbf{x}_4 = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$.
- Epoch 2: It turns out that there are no updates at this epoch, which means the PTA has converged. Let us now look at the final situation geometrically:


Class 1
x_2
OK!
Class 0
1
1
x_1
-1
2
-1
-2

ECE/CS 559 Lecture 4 9/5

Last time: Feed-Forward Neural Networks: Linear Algebraic Representation



$x \in \mathbb{R}^n \quad W \in \mathbb{R}^{k \times n} \quad b \in \mathbb{R}^k \quad U \in \mathbb{R}^{9 \times k} \quad c \in \mathbb{R}^9$

$$y = \varphi(U\varphi(Wx+b)+c)$$

- This is not that different than fitting a polynomial.
- Data ( , 3) (, 1), etc.

x y (target output)

$\varphi(U\varphi(Wx+b)+c) = y(x)$ (neural network output)

Goal: minimize # of mistakes: $\sum_{\text{data}} \mathbb{1}\{y(x) \neq y\}$

- Typical Algorithm process data gradually:
  1. Initialize weights randomly
  2. For each (x, y) in data:
     - Feed x to NN and get $y(x)$
     - Update weights to bring $y(x)$ closer to y.
  3. Repeat 2 until weights converge

epoch: 1-pass over data

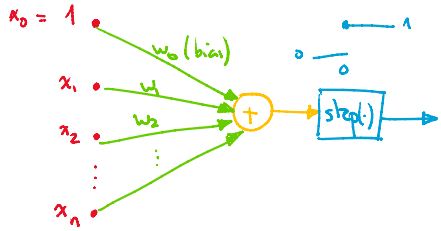
$(y, \hat{y})$ gap = loss

① Approximation Theorems and Learning Algorithms

- Can we always find weights & biases (parameters)?
  - Given enough neurons & layers, almost any function can be approximated.
  - With logic (binary inputs/outputs) the DNF/CNF theorem shows that 2 layers are enough.
- But how do we set these parameters to achieve the desired behavior?
  - Manual adjustments → intractable except in simple cases
  - Instead, learn with guidance from data.

② Perceptrons

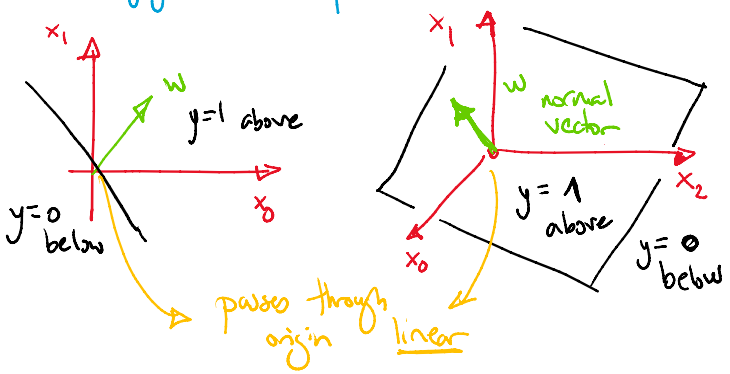
- Consider the McCulloch-Pitts (1943) neuron, with bias as weight:



$$y = \text{step}\left(\sum_{j=0}^{n} w_j x_j\right) = \begin{cases} 1, & [-w-]\begin{bmatrix} 1 \\ x \end{bmatrix} \geq 0 \\ 0, & < 0 \end{cases}$$

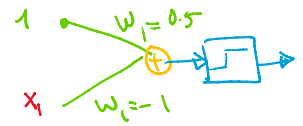
- What can it do?
  - Recall $[w_0, w_1]\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = 0$ line $[w_0, w_1, w_2]\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = 0$ plane
  - $[-w-]\begin{bmatrix} 1 \\ x \end{bmatrix} = 0$ is a hyperplane.
    ≥ 0 is a half-space.

- This single neuron was used later by Rosenblatt (1961) who referred to it as "perceptron".
- It could classify with a hyperplane as decision boundary:
  → jargon: linear separator

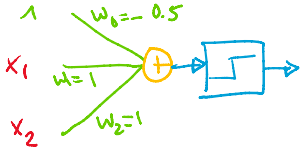


- Can model logic gate: 0 = False 1 = True

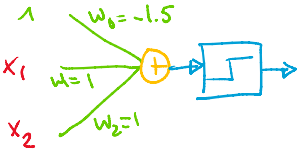
NOT:



OR:

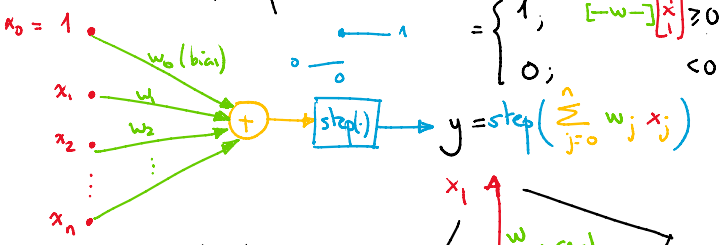


AND:

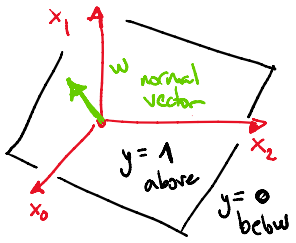


ECE/CS 559 Lecture 5 9/10

Last time: Approximation Theorems and Learning Algorithms
Perceptrons

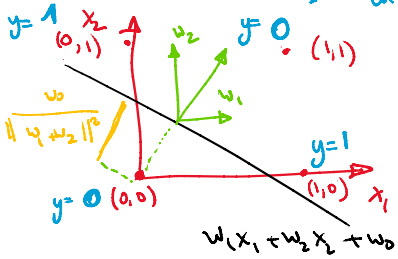


- What can it do?
  - linear separator
  - Can model logic gate



- How many layers are sufficient for any operation?
  Answer: Two. Encode the truth table (CNF/DNF Theorem)
  May not be the most efficient - Depth ⇒ fewer neurons
- Is 1 layer sufficient? Answer: No!

Example: XOR gate:



cannot create XOR pattern with a linear separator (without x_0, does not pass through origin) affine

① Example

$$\underbrace{\neg x_1 \wedge \neg x_2 \wedge \cdots \wedge \neg x_n}_{n \text{ negated}} \wedge \underbrace{x_{n+1} \wedge \cdots \wedge x_{n+m}}_{m \text{ direct}} \quad \text{(AND)}$$

$$\Leftrightarrow \quad (1-x_1) + \cdots + (1-x_n) + x_{n+1} + \cdots + x_{n+m} \geq n+m-\tfrac{1}{2}$$

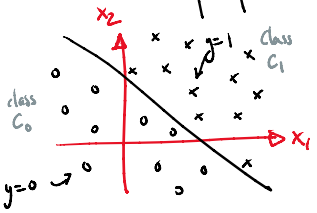
satisfied if all $= 1$

$$\Leftrightarrow \quad \underbrace{-n+\tfrac{1}{2}}_{w_0} \underbrace{-x_1}_{w_1} \underbrace{-x_2}_{w_2} - \cdots \underbrace{-x_n}_{w_n} + \underbrace{x_{n+1}}_{w_{n+1}} + \cdots + \underbrace{x_{n+m}}_{w_{n+m}} \geq 0$$

- Because NOT + either AND or OR are universal Multilayer Perceptrons can represent any logical operation.

② Perceptron Learning Algorithm

- Given a description of the desired behavior $y = f(x)$ how do we find parameters such that $y_{\text{params}}(x) \approx f(x)$? (in this case params $\equiv w$)
- Assume there exists w such that $y_w(x) = \text{step}(w^T x) = y$ exactly.
  - Jargon: Classes ($y=0$ $y=1$ regions) are called linearly separable since perceptron can realize this, the problem is realizable



- In supervised learning, the description takes the form of data: many pairs (x, y)

Algorithm: Input: Data $= \{(x,y), \cdots\}$ $x \in \mathbb{R}^n$ $y \in \{0,1\}$, η
Output: weights $w \in \mathbb{R}^n$

1. Initialize w arbitrarily.
2. While there exists $(x,y) \in$ Data such that $y_w(x) \neq y$:
3. For each $(x,y) \in$ Data:
   - If $y_w(x) = y$: do nothing
   - Else if $y_w(x) = 0$ and $y = 1$: $w \leftarrow w + \eta x$
   - Else if $y_w(x) = 1$ and $y = 0$: $w \leftarrow w - \eta x$

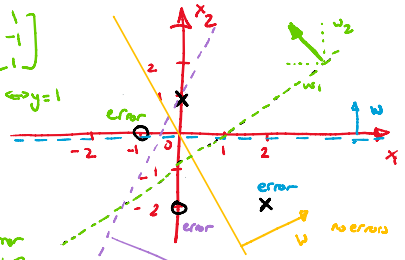
1 epoch = 1 pass over data

Simplifies to: $w \leftarrow w + \eta x (y - y(x))$

Theorem: If classes are linearly separable, will converge for any η.

Example: 1. Initialize $w = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$

$$y_w(x) = \begin{cases} 1; & 1 - 1x_1 + 1x_2 \geq 0 \\ 0; & 1 - 1x_1 + 1x_2 < 0 \end{cases} \qquad x_2 \geq x_1 - 1 \Leftrightarrow y = 1$$



2. Epoch 1: (here, $\eta = 1$)
3. Updates:
   $w = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \leftarrow w - \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$ (error) $\quad x_2 \geq 0 \Leftrightarrow y = 1$
   $w = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix} \leftarrow w + \begin{bmatrix} 1 \\ 2 \\ -2 \end{bmatrix}$ (error) $\quad 1 + 2x_1 - x_2 \geq 0$, $x_2 \leq 2x_1 + 1 \Leftrightarrow y = 1$
2. Epoch 2:
3. Updates: $w = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} \leftarrow w - \begin{bmatrix} 1 \\ 0 \\ -2 \end{bmatrix}$ (error) $\quad 0 + 2x_1 + x_2 \geq 0$, $x_2 \geq -2x_1 \Leftrightarrow y = 1$

No more errors, terminate & return w.

ECE/CS 559 - Introduction to Supervised Learning

Erdem Koyuncu

1 Introduction

The output of the neural network will depend on the weights $\mathbf{w}$ of the network and the input signals/vector $\mathbf{x}$ (in addition to the activator function, network topology, etc). Nevertheless, we may simply write $\mathbf{y} = f(\mathbf{w}, \mathbf{x})$ as the network input-output relationship, where f is a certain non-linear function that depends on the parameters discussed above.

Our general goal is to find the appropriate weights such that the network can accomplish a certain desired task. One example is the perceptron that we discussed in the previous lectures: We used the perceptron training algorithm to find the appropriate weights so that the perceptron could classify two linearly separable classes. This was one example of learning (actually, supervised learning).

In general, learning is a systematic method to change/update the weights so that the network can accomplish the desired task after a certain number of learning steps.

In a learning task, usually we have a training set $\mathcal{S} = \{\mathbf{x}_i : i = 1, \ldots, |\mathcal{S}|\}$, and we begin with an initial choice of weights (e.g. picked randomly) $\mathbf{w}_1$. If we are extremely lucky, our random initial choice of weights will perform the desired task, and we do not need to do any learning. This is almost always not the case (for any nontrivial application), and we proceed to the first epoch of learning.

At Epoch 1, We begin with training set example $\mathbf{x}_1$, feed it to the network, we get the output $\mathbf{y}_1 = f(\mathbf{x}_1, \mathbf{w}_1)$. Based on the output $\mathbf{y}_1$ and our specific learning method, we change/update the weights to $\mathbf{w}_2 = \mathbf{w}_1 + \Delta\mathbf{w}_1$. We then proceed to the second training set element $\mathbf{x}_2$, we get the output with the updated weights $\mathbf{y}_2 = f(\mathbf{x}_2, \mathbf{w}_2)$ and find the new weights as $\mathbf{w}_3 = \mathbf{w}_2 + \Delta\mathbf{w}_2$. We keep updating weights until all members of the training set are exhausted.

The weights we obtain at the end of epoch 1 may be able to correctly perform the desired task. If this is the case, the learning may end. Otherwise, we continue to the second epoch of learning, and keep updating the weights in the same manner as described in Epoch 1.

The expectation is that after a certain number of epochs, the weights will converge and they will be able to perform our desired task. In some scenarios, such as the perceptron training algorithm, the learning method provably converges under the assumption of linearly separable classes. In general, the learning process may not converge (In fact, the performance we obtain after each epoch of learning may degrade), in which case we need to stop the learning/training. One can then retry with a different set of initial weights, a different learning method, or change the parameters (e.g. learning rate) of the learning process.

There are basically three different types of learning:

1. Supervised learning: For each pattern (member of the training set), we know what the correct output should be, and we update the weights accordingly.

2. Unsupervised learning: We do not have any correct output information. The learning algorithm finds (by itself) similarities between different training samples, and classifies them accordingly (usually this classification is called "clustering" when talking about unsupervised learning). In certain contexts, the process is also referred to as self-organization.

3. Reinforcement learning: We may think of $\mathbf{y}_1, \mathbf{y}_2, \ldots$ (the outputs) as the actions we choose (e.g. moves of a chess game). Each action/output will incur a different reward (some outputs may be good/some may be bad, chess piece taken good, queen lost bad), and a new different observation of the environment (the new chess board configuration after the opponent also makes his move - note that this may thus be random). Depending on the rewards and the new observation, we update the weights.

In terms of how we update the weights, there are basically two different types:

- Stepwise learning: One input is given, output is observed and the weights are updated. Then, for the next input, the last updated weights are used (as in the description above or the perceptron training algorithm). In this case, there is no need to store the weight increments. This is also called on-line learning.
- Batch learning: One input is given, necessary weight increments are found, but the weights are not updated. When the next input is given, previous weights are used, and again the necessary weight increments are found. This process is repeated till the end of an epoch. At the end of epoch, the weight increments are added to find the total increment, and added to the weights used within the epoch. This is sometimes called as off-line learning.

  In other words, in offline learning, at Epoch 1, We begin with training set example $\mathbf{x}_1$, feed it to the network, we get the output $\mathbf{y}_1 = f(\mathbf{x}_1, \mathbf{w}_1)$. We then proceed to the second training set element $\mathbf{x}_2$, we get the output with the original weights $\mathbf{y}_2 = f(\mathbf{x}_2, \mathbf{w}_1)$ and so on until all members of the training set are exhausted, i.e. $\mathbf{y}_i = f(\mathbf{x}_i, \mathbf{w}_1)$, $i = 1, \ldots, |\mathcal{S}|$. We now update the weights $\mathbf{w}_1$ to $\mathbf{w}_2$ depending on our observations $\mathbf{y}_i$s and the inputs $\mathbf{x}_i$s. Usually, the update is of the form $\mathbf{w}_2 = \mathbf{w}_1 + \sum_{i=1}^{n} \delta(\mathbf{x}_i, \mathbf{y}_i)$, where δ is some function that depends on the specific learning method. We then proceed to a new Epoch, and so on.

First, we focus on supervised learning.

2 Supervised Learning

2.1 Formulation

As we have mentioned before in Section 3 of Lecture 3, we may think of learning as an approximation problem, where we wish to approximate the function $h : \mathbb{R}^n \to \mathbb{R}^m$. In supervised learning we have the training set $\mathcal{S} = \{\mathbf{x}_i : i = 1, \ldots, |\mathcal{S}|\}$, and the samples (desired outputs) of the function $\{h(\mathbf{x}_i) : i = 1, \ldots, |\mathcal{S}|\}$ are available. The learning problem is then to find the weights $\mathbf{w}$ such that $f(\mathbf{w}, \mathbf{x})$ (the output of our neural network) is as close to $h(\mathbf{x})$ as possible, at least over the training set.

The mathematical way to describe closeness is via the notion of a **metric**.

A metric ρ on a set X is a function $\rho : X \times X \to [0, \infty)$ that satisfies the following properties for every $x, y, z \in X$:

1. Non-negativity: $\rho(x, y) \geq 0$.
2. Identity of indiscernibles: $\rho(x, y) = 0 \implies x = y$.
3. Symmetry: $\rho(x, y) = \rho(y, x)$.
4. Subadditivity/Triangle inequality: $\rho(x, y) \leq \rho(x, z) + \rho(y, z)$.

A metric can be thought as a distance function. For $X = \mathbb{R}^n$, with $\mathbf{y} = [y_1 \cdots y_n]^T$ and $\mathbf{x} = [x_1 \cdots x_n]^T$, some examples include:

- Euclidean metric: $\rho(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$.
- The L^p-metric, where $p \geq 1$: $\rho(\mathbf{x}, \mathbf{y}) = (\sum_{i=1}^{n} |x_i - y_i|^p)^{\frac{1}{p}}$.
- The L^∞-metric: $\rho(\mathbf{x}, \mathbf{y}) = \max_{i \in \{1, \ldots, n\}} |x_i - y_i|$.

A popular choice is the Euclidean metric as it is often analytically tractable with nice properties. Note that then, $\rho^2(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} (x_i - y_i)^2 = (\mathbf{x} - \mathbf{y})^T (\mathbf{x} - \mathbf{y})$.

In any case, the learning problem is to find an optimal vector / matrix of weights $\mathbf{w}^\star$ such that $\rho(h(\mathbf{x}), f(\mathbf{x}, \mathbf{w}^\star)) \leq \rho(h(\mathbf{x}), f(\mathbf{x}, \mathbf{w}))$ for every $\mathbf{x} \in \mathcal{S}$ and $\mathbf{w}$.

Often, the errors are averaged out over the training samples so that defining (for the Euclidean metric)

$$E(\mathbf{w}) = \frac{1}{|\mathcal{S}|} \sum_{\mathbf{x} \in \mathcal{S}} \|f(\mathbf{x}, \mathbf{w}) - h(\mathbf{x})\|^2$$

the goal of learning is to find an optimal vector / matrix of weights $\mathbf{w}^\star$ such that $E(\mathbf{w}^\star) \leq E(\mathbf{w})$ for every $\mathbf{w}$.

2.2 An exact solution

The linear case can be solved exactly: Suppose we have m input neurons and n output neurons without any bias, and the activator function is identity. Let $\mathbf{W} \in \mathbb{R}^{n \times m}$ denote the weight matrix with the weight going to output neuron i from neuron j denoted by w_{ij}. Then, given input $\mathbf{x} \in \mathbb{R}^{m \times 1}$, the output is $\mathbf{y} = \mathbf{W}\mathbf{x}$. So, if the desired outputs are $\mathbf{d}_i$, $i = 1, \ldots, |\mathcal{S}|$ given patterns $\mathbf{x}_i$, $i = 1, \ldots, |\mathcal{S}|$, we have to minimize

$$E(\mathbf{W}) = \frac{1}{|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} \|\mathbf{d}_i - \mathbf{W}\mathbf{x}_i\|^2$$

or letting $\mathbf{D} = [\mathbf{d}_1 \cdots \mathbf{d}_{|\mathcal{S}|}]$, $\mathbf{X} = [\mathbf{x}_1 \cdots \mathbf{x}_{|\mathcal{S}|}]$, we have

$$E(\mathbf{W}) = \frac{1}{|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} \|\mathbf{d}_i - \mathbf{W}\mathbf{x}_i\|^2 = \frac{1}{|\mathcal{S}|} \|\mathbf{D} - \mathbf{W}\mathbf{X}\|^2$$

Definition: Let X be a real $m \times q$ matrix. There exists a unique $q \times m$ matrix X^+ with the following properties: (i) $XX^+X = X$, (ii) $X^+XX^+ = X^+$, and (iii) X^+X and XX^+ are (Hermitian) symmetric matrices. The matrix X^+ is called the pseudo-inverse, or the Moore-Penrose pseudo-inverse of X.

If X has linearly independent columns, then $A^+ = (A^TA)^{-1}A^T$.

If X has linearly independent rows, then $A^+ = A^T(AA^T)^{-1}$.

For complex case, replace all transposes with Hermitian transpose, and all symmetrixes with Hermitian symmetrixes.

Proposition: Let $X \in \mathbb{R}^{m \times q}$, $Y \in \mathbb{R}^{n \times q}$. The $n \times m$ minimizer of $\|Y - WX\|^2$ is YX^+.

Proof: For any matrix A, note that $\|A\|^2 = \mathrm{tr}(AA^T)$. We have

$$\begin{aligned}
\|YX^+X - WX\|^2 &= \mathrm{tr}\Big[(YX^+X - WX)(YX^+X - WX)^T\Big] \\
&= \mathrm{tr}\Big[YX^+X(X^+X)^TY^T \\
&\quad - YX^+XX^TW^T \\
&\quad - WX(X^+X)^TY^T \\
&\quad + WXX^TW^T\Big]
\end{aligned}$$

Use the fact that XX^+ is symmetric for the first term so that $(X^+X)^T = X^+X$, giving $X^+X(X^+X)^T = X^+XX^+X = X^+X$, the last equality is from (ii) of the properties of pseudoinverse. For the second term, use the fact that $X^+XX^T = (X^+X)^TX^T = (XX^+X)^T = X^T$, the last equality is from property (i) of pseudoinv. For the third term, we have $X(X^+X)^T = XX^+X = X$. Thus,

$$\begin{aligned}
&\|YX^+X - WX\|^2 \\
&= \mathrm{tr}\Big[YX^+XY^T - YX^TW^T - WXY^T + WXX^TW^T\Big].
\end{aligned}$$

On the other hand, again using $\|A\|^2 = \mathrm{tr}(AA^T)$, we obtain

$$\|Y - WX\|^2 = \mathrm{tr}\Big[YY^T - YX^TW^T - WXY^T + WXX^TW^T\Big]$$

The two equalities give

$$\|YX^+X - WX\|^2 = \|Y - WX\|^2 + \mu$$

where μ depends only on X and Y. Hence the optimal solution that minimizes $\|Y - WX\|^2$ should also minimize $\|YX^+X - WX\|^2 = \|(YX^+ - W)X\|^2$. One solution is clearly $W = YX^+$. ■

Hence, one minimizer of $E(\mathbf{W})$ in this case is given by $\mathbf{DX}^+$.

2.3 Nonlinear, nonconvex optimization

Unfortunately, most realistic problems require a non-linear non-convex approach, i.e. in general $E(\mathbf{W})$ is non-linear non-convex in $\mathbf{W}$ (e.g. when we utilize a non-linear activator function). Our general goal is thus to find the global minima of a certain energy function $E(\cdot)$ that may be non-linear, non-convex etc, say, for a single-layer network with a non-linear activation function

$$E(\mathbf{W}) = \frac{1}{|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} \|\mathbf{d}_i - \phi(\mathbf{W}\mathbf{x}_i)\|^2$$

with the understanding that $\phi(\cdot)$ is applied component-wise. We visit some well-known solution methods in this context.

2.4 Gradient descent

We ask ourselves, if we are currently at position $\mathbf{w}$ with energy $E(\mathbf{w})$, which position we have to move so that the energy $E(\cdot)$ is decreased. To answer this, define the gradient operator

$$\nabla \triangleq \left[\frac{\partial}{\partial w_1} \cdots \frac{\partial}{\partial w_n}\right]^T$$

so that letting $\mathbf{g} \triangleq \nabla E(\mathbf{w})$, we obtain via Taylor series

$$E(\mathbf{w} + \Delta\mathbf{w}) = E(\mathbf{w}) + \mathbf{g}^T\Delta\mathbf{w} + O(\|\Delta\mathbf{w}\|^2).$$

Hence, if we move just a little, i.e. if $\Delta\mathbf{w}$ is small the new energy is approximately,

$$E(\mathbf{w} + \Delta\mathbf{w}) \simeq E(\mathbf{w}) + \mathbf{g}^T\Delta\mathbf{w}.$$

Choosing $\Delta\mathbf{w} = -\eta\mathbf{g}$ for some constant $\eta > 0$, we obtain

$$E(\mathbf{w} + \Delta\mathbf{w}) \simeq E(\mathbf{w}) - \eta\|\mathbf{g}\|^2$$

so that the new position $\mathbf{w} + \delta\mathbf{w}$ indeed results in a lesser energy, provided that η is small. The update $\mathbf{w} \leftarrow \mathbf{w} - \eta\nabla E(\mathbf{w})$ is called the gradient descent.

Intuitively, the gradient $\mathbf{g}$ describes the direction where $E(\mathbf{w})$ increases the fastest. Hence, we move in the negative direction, which is the direction where $E(\mathbf{w})$ decreases the fastest.

The method of steepest descent converges to the optimal solution slowly, as the step size decreases as one approaches the optimal solution. Also, if η is small, the transient response of the algorithm is overdamped, in that the trajectory traced by $\mathbf{w}$ follows a smooth path (although if η is too small, then the convergence may take many iterations).

When η is large, the transient response of the algorithm is underdamped, in that the trajectory of $\mathbf{w}$ follows a zigzagging (oscillatory) path.

When η exceeds a certain critical value, the algorithm may become unstable, i.e. $\mathbf{w}$ diverges to infinity.

2.5 Newton's method

Define the Hessian

$$\nabla^2 \triangleq \begin{bmatrix} \frac{\partial^2 E(\mathbf{w})}{\partial w_1^2} & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_n} \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_2^2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_n \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_n^2} \end{bmatrix},$$

so that letting $\mathbf{g} \triangleq \nabla E(\mathbf{w})$ and $\mathbf{H} \triangleq \nabla^2 E(\mathbf{w})$, we obtain, again by Taylor series,

$$\begin{aligned} E(\mathbf{w} + \Delta\mathbf{w}) &= E(\mathbf{w}) + \mathbf{g}^T \Delta\mathbf{w} + \frac{1}{2}(\Delta\mathbf{w})^T \mathbf{H} \Delta\mathbf{w} + O(\|\Delta\mathbf{w}\|^3) \\ &\simeq E(\mathbf{w}) + \mathbf{g}^T \Delta\mathbf{w} + \frac{1}{2}(\Delta\mathbf{w})^T \mathbf{H} \Delta\mathbf{w}. \end{aligned}$$

We now minimize RHS over all possible choices of $\Delta\mathbf{w}$. The function is of the form $f(x) = x^T Q x + c^T x$. We have

$$\frac{1}{2} x^T Q x + c^T x = \sum_{i=1}^{n} \sum_{j=1}^{n} q_{ij} x_i x_j + \sum_{i=1}^{n} c_i x_i$$

so, taking the derivatives, we obtain

$$\frac{\partial}{\partial x_k} \left(\frac{1}{2} x^T Q x + c^T x \right) = q_{kk} x_k + \frac{1}{2} \sum_{i=1,\, i \neq k}^{n} q_{ik} x_i + \frac{1}{2} \sum_{i=1,\, i \neq k}^{n} q_{ki} x_i + c_k = \sum_{i=1}^{n} q_{ik} x_i$$

and thus $\frac{\partial}{\partial x} \left(\frac{1}{2} x^T Q x + c^T x \right) = Qx + c$. Letting $Qx + c = 0$, we obtain the critical point $x = -Q^{-1}c$, or $\Delta\mathbf{w} = -\mathbf{H}^{-1}\mathbf{g}$ for the original problem. Now, note that the Hessian of our objective function is Q. When the Hessian Q is positive definite, the critical points of the function are local minimum. The only critical point is $x = -Q^{-1}c$. A continuous, twice differentiable function of several variables is convex on a convex set if and only if its Hessian matrix is positive semidefinite on the interior of the convex set. Hence, our function is also convex. Since the function is convex (as shown below), the critical point, which is a local minimum, is also a global minimum.

Usually a learning parameter is also appended, i.e. $\Delta\mathbf{w} = -\eta\mathbf{H}^{-1}\mathbf{g}$. We then obtain

$$\begin{aligned} E(\mathbf{w} + \Delta\mathbf{w}) - E(\mathbf{w}) &\simeq -\eta \mathbf{g}^T \mathbf{H}^{-1} \mathbf{g} + \frac{\eta^2}{2} (\mathbf{H}^{-1}\mathbf{g})^T \mathbf{H} \mathbf{H}^{-1} \mathbf{g} \\ &= -\eta \mathbf{g}^T \mathbf{H}^{-1} \mathbf{g} + \frac{\eta^2}{2} \mathbf{g}^T (\mathbf{H}^{-1})^T \mathbf{g} \\ &= -\eta \mathbf{g}^T \mathbf{H}^{-1} \mathbf{g} + \frac{\eta^2}{2} \left(\mathbf{g}^T (\mathbf{H}^{-1})^T \mathbf{g} \right)^T \\ &= -\left(\eta + \frac{\eta^2}{2} \right) \mathbf{g}^T \mathbf{H}^{-1} \mathbf{g} \\ &\simeq -\eta \mathbf{g}^T \mathbf{H}^{-1} \mathbf{g} \end{aligned}$$

The last expression is greater than or equal to 0 if $\mathbf{H}^{-1}$ is positive definite, so that our method is indeed a descent, if η is also small.

If $\mathbf{H}$ is not positive-definite, usually the following modification is applied $\Delta\mathbf{w} = -\eta\beta(\beta\mathbf{I} + \mathbf{H})^{-1}\mathbf{g}$. As β gets large the "new Hessian" $\beta\mathbf{I} + \mathbf{H}$ is bound to be positive definite. In fact, as $\beta \to \infty$, this type of iteration becomes the same as Gradient descent.

The geometric interpretation of Newton's method is that at each iteration one approximates $E(w)$ by a quadratic function around w_n, and then takes a step towards the maximum/minimum of that quadratic function (in higher dimensions, this may also be a saddle point). Note that if $E(w)$ happens to be a quadratic

function, then the exact extremum is found in one step. Indeed if $E(w) = \frac{1}{2}w^T Q w + c^T w$, then gradient is $Qw + c$, and hessian is Q so that starting from $w = b$, we go to the point $b - H^{-1}(Hb + c) = -H^{-1}c$, but this new point is the global minimum.

Generally speaking, Newton's method converges quickly asymptotically and does not exhibit the zigzagging behavior that sometimes characterizes the method of steepest descent. In any case, one limitation of Newton's method is the computational complexity: To implement, the method, we need to calculate Hessian and take its inverse - both are difficult tasks when the dimensionality is large.

2.6 Gauss-Newton method

Used to solve non-linear least-squares problems, i.e. suppose $E(\mathbf{w})$ is of the form $E(\mathbf{w}) = \sum_{i=1}^{q} e_i^2(\mathbf{w})$ for some functions e_i. Suppose that the dimensionality of $\mathbf{w}$ is n. Then, we have

$$E(\mathbf{w}) = \left\| \begin{bmatrix} e_1(\mathbf{w}) \\ \vdots \\ e_q(\mathbf{w}) \end{bmatrix} \right\|^2$$

so that

$$E(\mathbf{w}+\Delta\mathbf{w}) = \left\| \begin{bmatrix} e_1(\mathbf{w}+\Delta\mathbf{w}) \\ \vdots \\ e_q(\mathbf{w}+\Delta\mathbf{w}) \end{bmatrix} \right\|^2 \simeq \left\| \begin{bmatrix} e_1(\mathbf{w}) + (\nabla e_1(\mathbf{w}))^T \Delta\mathbf{w} \\ \vdots \\ e_q(\mathbf{w}) + (\nabla e_q(\mathbf{w}))^T \Delta\mathbf{w} \end{bmatrix} \right\|^2 \triangleq \|\mathbf{e}(\mathbf{w}) + \mathbf{J}\Delta\mathbf{w}\|^2,$$

where $\mathbf{e}(\mathbf{w}) \triangleq \begin{bmatrix} e_1(\mathbf{w}) \\ \vdots \\ e_q(\mathbf{w}) \end{bmatrix}$, and $\mathbf{J} \triangleq \begin{bmatrix} (\nabla e_1(\mathbf{w}))^T \\ \vdots \\ (\nabla e_q(\mathbf{w}))^T \end{bmatrix}$ is the $q \times n$ Jacobian.

Now recall that the minimizer of $\|\mathbf{D} - \mathbf{W}\mathbf{X}\|^2$ was $\mathbf{D}\mathbf{X}^+$. Equivalently, the minimizer of $\|\mathbf{D}^T - \mathbf{X}^T\mathbf{W}^T\|^2$ is $\mathbf{D}\mathbf{X}^+$. Equivalently, the minimizer of $\|\mathbf{D} - \mathbf{X}\mathbf{W}^T\|^2$ is $\mathbf{D}^T(\mathbf{X}^T)^+$. One can easily show that $(\mathbf{X}^T)^+ = (\mathbf{X}^+)^T$. Hence, the minimizer of $\|\mathbf{D} - \mathbf{X}\mathbf{W}\|^2$ is $(\mathbf{D}^T(\mathbf{X}^T)^+)^T = \mathbf{X}^+\mathbf{D}$

Using this result, minimizing the linear approximation of $E(\mathbf{w})$ over all possible $\Delta\mathbf{w}$, we obtain the optimal descent $(\Delta\mathbf{w})_\star = (-\mathbf{J})^+\mathbf{e} = -\mathbf{J}^+\mathbf{e}(\mathbf{w})$. Note that usually q (the number of observations) is much larger than n, so that $\mathbf{J} \in \mathbb{R}^{q \times n}$ (often) has linearly independent columns. In such a scenario, we have $\mathbf{J}^+ = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T$, and the updates are given by $(\Delta\mathbf{w})_\star = -(\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\mathbf{e}$.

In certain cases, $\mathbf{J}$ may still not have linearly-independent columns, in which case one can use the update $(\Delta\mathbf{w})_\star = -(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})^{-1}\mathbf{J}^T\mathbf{e}(\mathbf{w})$, where δ is "regularization parameter." As $\delta \to \infty$, we obtain gradient descent over each component function $e_i(\mathbf{w})$.

Remark - 1: Note that the solution $(\Delta\mathbf{w})_\star = -(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})^{-1}\mathbf{J}^T\mathbf{e}(\mathbf{w})$ is the minimizer of the regularized mean squared problem

$$\|\mathbf{e}(\mathbf{w}) + \mathbf{J}\Delta\mathbf{w}\|^2 + \lambda\|\Delta\mathbf{w}\|^2.$$

To see this, we can write the objective function of the regularized mean-squares problem as

$$\left\| \begin{bmatrix} e(\mathbf{w})_{q\times 1} \\ \mathbf{0}_{n\times 1} \end{bmatrix} + \begin{bmatrix} \mathbf{J}_{q\times n} \\ \sqrt{\lambda}\mathbf{I}_{n\times n} \end{bmatrix} \Delta\mathbf{w}_{n\times 1} \right\|^2$$

The solution can be shown to be (using the Proposition in Section 3.2):

$$\left(\begin{bmatrix} \mathbf{J} \\ \sqrt{\lambda}\mathbf{I} \end{bmatrix}^T \begin{bmatrix} \mathbf{J} \\ \sqrt{\lambda}\mathbf{I} \end{bmatrix} \right)^{-1} \begin{bmatrix} \mathbf{J} \\ \sqrt{\lambda}\mathbf{I} \end{bmatrix}^T \begin{bmatrix} e(\mathbf{w}) \\ \mathbf{0} \end{bmatrix} = (\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})^{-1}\mathbf{J}^T e(\mathbf{w}).$$

The regularization ensures that the next step is not too far from the previous step (when λ is large).

Remark - 2: Derivation from Newton's method:

Recall that in Newton's method we had the iterations: $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{H}^{-1}\mathbf{g}$. In particular, for the function $E(\mathbf{w}) = \sum_{i=1}^{q} e_i^2(\mathbf{w})$, we can calculate

$$\mathbf{g} = \left[2\sum_{i=1}^{q} e_i \frac{\partial e_i}{\partial w_1} \quad \cdots \quad 2\sum_{i=1}^{q} e_i \frac{\partial e_i}{\partial w_n} \right]^T = 2\mathbf{J}^T\mathbf{e}(\mathbf{w}),$$

$$\mathbf{H}_{jk} = 2\sum_{i=1}^{q} \left(\frac{\partial e_i}{\partial w_k}\frac{\partial e_i}{\partial w_j} + e_i \frac{\partial e_i^2}{\partial w_j \partial w_k} \right) \simeq 2\sum_{i=1}^{q} \frac{\partial e_i}{\partial w_k}\frac{\partial e_i}{\partial w_j} = 2(\mathbf{J}^T\mathbf{J})_{j,k}$$

so the update rule is $\mathbf{w} \leftarrow \mathbf{w} - \left(2(\mathbf{J}^T\mathbf{J})\right)^{-1} 2\mathbf{J}^T\mathbf{e}(\mathbf{w})$ same as before.

The approximation "holds" if $\frac{\partial e_i}{\partial w_k}\frac{\partial e_i}{\partial w_j}$ is large relative to $e_i \frac{\partial e_i^2}{\partial w_j \partial w_k}$ in magnitude. Hence, the terms e_i should be small or/and the functions e_i should be close to being linear (When they are linear functions of $\mathbf{w}$, the approximation holds as the other terms are 0). In this context, if the function to be optimized is a quadratic function, that this rule will converge to the globally optimal solution in one iteration. Actually, then the $\mathbf{H}$ approximation becomes valid, the rule boils down to Newton's rule, which gives the optimal solution for quadratic functions in one step.

2.7 Supervised Learning Rules

We now have an arsenal of different optimization methods available at our disposal. Throughout the course, we shall mostly utilize however the simple gradient descent idea. Gradient descent applied to different scenarios are given different names including Delta Learning, Widrow-Hoff Rule, the LMS algorithm Back-propagation, etc. We begin with Widrow-Hoff learning.

2.8 Widrow-Hoff Learning/The LMS algorithm

Consider an identity activation function, one output neuron, m input neurons, and a training sequence of length $|\mathcal{S}|$ as usual. Given $\mathbf{w} \in \mathbb{R}^{m\times 1}$, We wish to minimize

$$E(\mathbf{w}) = \sum_{i=1}^{|\mathcal{S}|} (d_i - \mathbf{w}^T\mathbf{x}_i)^2$$

We already know the optimal $\mathbf{w}$ here. It is given by $\mathbf{w}_\star^T = \mathbf{d}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$, where $\mathbf{X} = [\mathbf{x}_1 \cdots \mathbf{x}_{|\mathcal{S}|}]$.

Alternatively, we may use gradient descent,

$$\frac{\partial E(\mathbf{w})}{\partial w_j} = -2\sum_{i=1}^{|\mathcal{S}|} (d_i - \mathbf{w}^T\mathbf{x}_i)x_{ij}$$

so that we shall utilize the update

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \sum_{i=1}^{|\mathcal{S}|} (d_i - \mathbf{w}^T\mathbf{x}_i)\mathbf{x}_i$$

In terms of learning, we thus begin with an initial weight. At epoch 1, We apply pattern $\mathbf{x}_1$, observe the output $\mathbf{w}^T\mathbf{x}_i$, do not change the weights, apply pattern $\mathbf{x}_2$, observe the outputs, do not change the weights, and so on until all training patterns are exhausted. We can then calculate the descent factor as above an update our weights accordingly at the end of the epoch (this has been called offline or batch learning remember).

For practical reasons, the update is also often done for each sample via

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(d_i - \mathbf{w}^T\mathbf{x}_i)\mathbf{x}_i$$

Note that this is shorthand notation for

$$\mathbf{w}(i+1) = \mathbf{w}(i) + \eta\left(d_i - [\mathbf{w}(i)]^T\mathbf{x}_i\right)\mathbf{x}_i,\ i = 1, \ldots, |\mathcal{S}|$$

with some initial condition $\mathbf{w}(1)$.

Note how this expression is very similar to the PTA. This is called the Widrow-Hoff learning rule, or the LMS algorithm. Note that this is not a true gradient descent, and may not converge to the globally optimal solution (unlike the true gradient descent does when η is taken sufficiently low). In fact, the solution will move randomly around the globally optimal solution (there is a nice theory that formalizes these ideas - but we will likely not discuss it in this course).

2.9 Delta Rule

This can be considered to be a generalization of Widrow-Hoff rule to an arbitrary (differentiable) activation function. Here, we wish to minimize

$$E(\mathbf{w}) = \sum_{i=1}^{|\mathcal{S}|}(d_i - \phi(\mathbf{w}^T\mathbf{x}_i))^2$$

Again, using the gradient descent idea, we obtain

$$\frac{\partial E(\mathbf{w})}{\partial w_j} = -2\sum_{i=1}^{|\mathcal{S}|}(d_i - \phi(\mathbf{w}^T\mathbf{x}_i))\phi'(\mathbf{w}^T\mathbf{x}_i)x_{ij}$$

so that the gradient descent relies on the update so that we shall utilize the update

$$\mathbf{w} \leftarrow \mathbf{w} + \eta\sum_{i=1}^{|\mathcal{S}|}(d_i - \phi(\mathbf{w}^T\mathbf{x}_i))\phi'(\mathbf{w}^T\mathbf{x}_i)\mathbf{x}_i$$

Again, for practical reasons, the update is usually done per-sample (online learning) via

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(d_i - \phi(\mathbf{w}^T\mathbf{x}_i))\phi'(\mathbf{w}^T\mathbf{x}_i)\mathbf{x}_i$$

This is called the Delta rule. For a less cumbersome formula, if $v_i = \mathbf{w}^T\mathbf{x}_i$ denotes the induced local field with input $\mathbf{x}_i$, and $y_i = \phi(v_i)$ is the corresponding output, we have

$$\mathbf{w}(i+1) = \mathbf{w}(i) + \eta(d_i - y_i)\phi'(v_i)\mathbf{x}_i,\ i = 1, \ldots, |\mathcal{S}|.$$

Calculation of ϕ' may be problematic in general. But, for some certain cases, the calculation is easier. For example,

- Recall
$$\tanh(\alpha x) = \frac{e^{\alpha x} - e^{-\alpha x}}{e^{\alpha x} + e^{-\alpha x}}$$
.
$$\begin{aligned}\frac{\partial \beta \tanh(\alpha x)}{\partial x} &= \beta\frac{\alpha(e^{\alpha x} + e^{-\alpha x})^2 - \alpha(e^{\alpha x} - e^{-\alpha x})^2}{(e^{\alpha x} + e^{-\alpha x})^2} \\ &= \beta\alpha(1 - \tanh^2(\alpha x)) = \frac{\alpha}{\beta}(\beta^2 - \beta^2\tanh^2(\alpha x))\end{aligned}$$
so in this case, the update is simply (since $y_i = \beta\tanh(\alpha v_i)$ by definition)
$$\mathbf{w}(i+1) = \mathbf{w}(i) + \eta\frac{\alpha}{\beta}(d_i - y_i)(\beta^2 - y_i^2)\mathbf{x}_i,\ i = 1, \ldots, |\mathcal{S}|.$$

- Or let $y = \phi(x) = \frac{\beta}{1+e^{-\alpha x}}$.
$$\frac{\partial\phi(x)}{\partial x} = \frac{\alpha\beta e^{-\alpha x}}{(1+e^{-\alpha x})^2} = \frac{\alpha}{\beta}e^{-\alpha x}y^2 = \frac{\alpha}{\beta}\left(\frac{\beta}{y} - 1\right)y^2 = \frac{\alpha}{\beta}y(\beta - y)$$
so the update is
$$\mathbf{w}(i+1) = \mathbf{w}(i) + \eta\frac{\alpha}{\beta}(d_i - y_i)y_i(\beta - y_i)\mathbf{x}_i,\ i = 1, \ldots, |\mathcal{S}|.$$

ECE/CS 559 Lecture 6 9/12

Last time: Perceptron Learning Algorithm

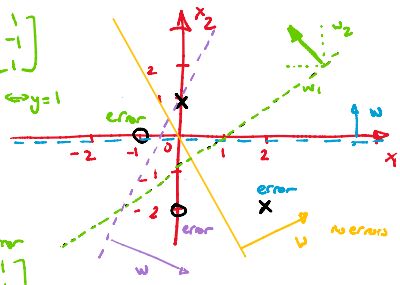
Algorithm: Input: Data $= \{(x,y), \dots\}$ $x \in \mathbb{R}^n$ $y \in \{0,1\}$, η
Output: weights $w \in \mathbb{R}^n$

1. Initialize w arbitrarily.
2. While there exists $(x,y) \in$ Data such that $y_w(x) \neq y$:
3. [1 epoch] For each $(x,y) \in$ Data:
   $w \leftarrow w + \eta x (y - y_w(x))$

Theorem: If classes are linearly separable, will converge for any η.

Example: 1. Initialize $w = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$

$y_w(x) = \begin{cases} 1 & ; 1 - 1x_1 + 1x_2 \geq 0 \quad x_2 \geq x_1 - 1 \Leftrightarrow y = 1 \\ 0 & ; 1 - 1x_1 + 1x_2 < 0 \end{cases}$



2. Epoch 1: (here, $\eta = 1$)
3. Updates:
   $w = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \leftarrow w - \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$ (error) $\quad x_2 \geq 0 \Leftrightarrow y = 1$
   $w = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix} \leftarrow w + \begin{bmatrix} 1 \\ 2 \\ -2 \end{bmatrix}$ (error) $\quad 1 + 2x_1 - x_2 \geq 0$, $x_2 \leq 2x_1 + 1 \Leftrightarrow y = 1$
2. Epoch 2:
3. Updates: $w = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} \leftarrow w - \begin{bmatrix} 1 \\ 0 \\ -2 \end{bmatrix}$ (error) $\quad 0 + 2x_1 + x_2 \geq 0$, $x_2 \geq -2x_1 \Leftrightarrow y = 1$

No more errors, terminate & return w.

① Supervised Learning

- Data (x, y), $x \in \mathcal{X}$ (feature space, also "context"), $y \in \mathcal{Y}$ (label space, also "target")
  - supervision
  - y discrete → classification, y continuous → regression
- Task: Predict/imitate y using only x.
- How: Use a neural network $y_w(x) = f(x; w)$
  This gives us a predictor, a function of x parametrized by w.
- Loss: Measures how far y and $f(x;w)$ are:
  0-1 loss: $\ell(y,f) = \mathbb{1}\{y \neq f\}$ Squared loss: $\ell(y,f) = \|y - f\|^2$
- Risk: Average loss over a data set: Data $= \{(x,y), \dots\}$
  $R(w) = \frac{1}{|\text{Data}|} \sum_{(x,y) \in \text{Data}} \ell(y, f(x,w))$ (empirical)

- Goal: Minimize (empirical) risk,
  make few mistakes/errors / stay close to y.
- Types of algorithms:
  - Online: data streams in, weights updated along the way
    (usually) don't revisit past data points
    - If we forgo epochs, perceptron = online.
  - Batch: data is available in whole, update weights using it all
    (usually) pass over data multiple times (epochs)

Online:
- See (x,y), update w
- See (x',y'), update w
- See (x'',y''), update w
⋮

Batch:
- Get Data $= \{(x,y), \dots\}$
- Epoch 1: use Data, update w
- Epoch 2: use Data, update w
⋮

- Mini batch: Process data gradually in small batches
  - Get Data - Divide it into small batches $\{\text{batch}_1, \text{batch}_2, \dots\}$ $|\text{batch}|$ = batch size
  - See batch_1, update w
  - See batch_2, update w
  ⋮
  } repeat (epochs)

ECE/CS 559 Lecture 7 9/17

Last time: Perceptron Learning Example — Supervised Learning

- Data: $\{(x,y),\dots\}$ $x \in \mathcal{X}$ (feature space), $y \in \mathcal{Y}$ (label space)
- Task: Predict/imitate y using only x.
- How: Use a predictor — neural network $y_w(x) \equiv f(x;w)$
- Loss: how far y and $f(x;w)$ are: $\ell(y,f) = \mathbb{1}\{y \neq f\}$ or $\|y-f\|^2$
- Risk: (empirical) $R(w) = \frac{1}{|\text{Data}|} \sum_{(x,y)\in\text{Data}} \ell(y, f(x,w))$
- Goal: Minimize (empirical) risk, make few mistakes/errors / stay close to y.

- Types of algorithms:
  - Online: - See (x,y), update w - See (x',y'), update w - See (x'',y''), update w ⋮
  - Batch: - Get Data $= \{(x,y),\dots\}$ • Epoch 1: use Data, update w • Epoch 2: use Data, update w ⋮

① Linear Regression with Squared Loss

- Data: $\{(x,y),\dots\}$ $x \in \mathbb{R}^n$ $y \in \mathbb{R}^m$
- Predictor: $f(x;w) = Wx$ $W \in \mathbb{R}^{m\times n}$ (this is a neuron with $g(v) = v$, identity activation)
- Goal: $\min_w R(w) = \sum_{(x,y)\in\text{Data}} \underbrace{\|y - Wx\|^2}_{\ell(y,Wx)}$

Analytic Solution: Optimality conditions, unconstrained $\Rightarrow \nabla_w R = 0$ (gradient)

- Single data point $\ell = \sum_{k=1}^{m} \left(y_k - \sum_{j=1}^{n} w_{kj} x_j\right)^2$

$$\frac{d\ell}{dw_{kj}} = -2x_j\left(y_k - \sum_{j'=1}^{n} w_{kj'}x_{j'}\right) = -2y_k x_j + \sum_{j'=1}^{n} w_{kj'}(x_j x_{j'})$$

$$\nabla_w \ell = -2yx^T + Wxx^T$$

- All data points: $R(w) = \sum_{(x,y)\in\text{Data}} \ell(y,Wx) \Rightarrow \nabla_w R = \sum_{(x,y)} \nabla_w \ell$

$$\nabla_w R = \sum_{(x,y)\in\text{Data}} -2yx^T + 2W \sum_{x\in\text{Data}} xx^T$$

$$= -2\underbrace{\begin{bmatrix} 1 & 1 & \cdots & 1 \\ y_1 & y_2 & \cdots & y_{|\text{Data}|} \end{bmatrix}}_{|\text{Data}|} \begin{bmatrix} -x_1- \\ -x_2- \\ \vdots \\ -x_{|\text{Data}|}- \end{bmatrix} + 2W \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_{|\text{Data}|} \end{bmatrix} \begin{bmatrix} -x_1- \\ -x_2- \\ \vdots \\ -x_{|\text{Data}|}- \end{bmatrix}$$

$$= -2YX^T + 2WXX^T$$

- $\nabla_w R = 0 \Rightarrow -2YX^T + 2WXX^T = 0$

$$\Rightarrow W = Y\underbrace{X^T(XX^T)^{-1}}_{\text{Moore-Penrose Pseudo inverse}}$$

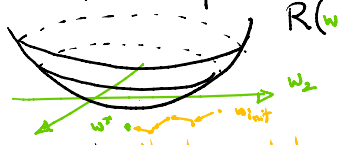
Compare to: $Wx \simeq Y$ not invertible if $|\text{Data}| > n$

$$\underbrace{\begin{bmatrix} w \end{bmatrix}}_{m\times n} \underbrace{\begin{bmatrix} x_1 & \cdots & x_{|\text{Data}|} \end{bmatrix}}_{n\times|\text{Data}|} \simeq \underbrace{\begin{bmatrix} y_1 & \cdots & y_{|\text{Data}|} \end{bmatrix}}_{m\times|\text{Data}|}$$

but it's "like" we inverted it above.

- Where are the updates? One single batch update!

Intuition: it's a quadratic equation



- Other shapes → solve by iterative updates

② Gradient Descent

Goal: $\min_w R(w) = \sum_{(x,y)\in\text{Data}} \underbrace{\|y - f(x;w)\|^2}_{\ell(y,f(x;w))}$

Idea: Set the derivative to zero! reduce step-by-step.

Gradient: $\nabla_w R = \left[\frac{\partial R}{\partial w_1}, \dots, \frac{\partial R}{\partial w_k}\right]$

Each step: Change w by Δw: $w' \leftarrow w + \Delta w$.

$$R(w') = R(w+\Delta w) = R(w) + \nabla_w R(w)^T \Delta w + O(\|\Delta w\|^2)$$

(Taylor expansion; ignore)

How should we choose Δw to get the best reduction?

- To minimize $\nabla_w R^T \Delta w$ over Δw, let $\Delta w \propto -\nabla_w R$: $w' \leftarrow w - \eta \nabla R(w)$ (proportional)

1-D Example: $R(w) = w^4$ $\frac{dR}{dw} = 4w^3$ $\eta = \frac{1}{8}$

$w(0) = 1$

$w(1) \leftarrow w(0) - \nabla R(w(0))\cdot\eta = 1 - 4\cdot 1^3 \cdot \frac{1}{8} = \frac{1}{2}$

$w(2) \leftarrow w(0) - \nabla R(w(0))\cdot\eta = \frac{1}{2} - 4\left(\frac{1}{2}\right)^3 \cdot \frac{1}{8} = \frac{3}{16}$

- What's happening?

$$w(t) \leftarrow w(t-1) - 4w(t-1)^3\eta = \underbrace{[1 - 4\eta w(t-1)^2]}_{<1} w(t-1)$$

- If η is small enough, always diminishes. (<1) Since bounded from below (by 0) will converge.
- But will it converge to 0? Yes, think about why.
- What if η is too big? Try $\eta = \frac{1}{2}$ with $w(0) = 1$.

ECE/CS 559 Lecture 8 Th 9/19

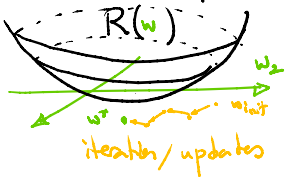
Last time: Linear Regression with Squared Loss

- Data: $\{(x,y),\dots\}$ $x \in \mathbb{R}^n$ $y \in \mathbb{R}^m$
- Predictor: $f(x;W) = Wx$ $W \in \mathbb{R}^{m\times n}$
- Goal: minimize $_W R(W) = \sum_{(x,y)\in Data} \underbrace{\| y - Wx \|^2}_{\ell(y, Wx) \equiv \ell((x,y),W)}$

$m\left[\hat{W}\right] \cdot \left[\overset{Data}{\begin{matrix}\vdots\\ X\\ \vdots\end{matrix}}\right] = m\left[\overset{Data}{\begin{matrix}\vdots\\ Y\\ \vdots\end{matrix}}\right]$

$W = Y\,[X^T(XX^T)^{-1}]$ Pseudo inverse

it's a quadratic equation



iterates / updates

① Gradient Descent

Goal: minimize $_W R(W) = \sum_{(x,y)\in Data} \underbrace{\| y - f(x;W) \|^2}_{\ell(y, f(x;W))}$

Idea: Set the derivative to zero! → reduce step by step.

Gradient: $\nabla_W R = \left[\frac{\partial R}{\partial w_1}, \dots, \frac{\partial R}{\partial w_k}\right]$

Each step: Change W by ΔW: $W' \leftarrow W + \Delta W$.

$R(W') = R(W+\Delta W) = R(W) + \nabla R(W)^T \Delta W + O(\|\Delta W\|^2)$

Taylor expansion; ignore

How should we choose ΔW to get the best reduction?

- To minimize $_{\Delta W} \nabla R^T \Delta W$, let $\Delta W \propto -\nabla_W R$ (proportional): $W' \leftarrow W - \eta \nabla R(W)$

1-D Example: $R(w) = w^4$ $\frac{dR}{dw} = 4w^3$ $\eta = \frac{1}{8}$

$w(0)=1$

$w(1) \leftarrow w(0) - \nabla R(w(0))\cdot\eta = 1 - 4\cdot 1^3 \cdot \frac{1}{8} = \frac{1}{2}$

$w(2) \leftarrow w(1) - \nabla R(w(1))\cdot\eta = \frac{1}{2} - 4\left(\frac{1}{2}\right)^3 \cdot \frac{1}{8} = \frac{7}{16}$

- What's happening?

$w(t) \leftarrow w(t-1) - 4\,w(t-1)^3\,\eta = [1 - 4\eta\, w(t-1)^2]\, w(t-1)$

- If η is small enough, always diminishes. (<1) Since bounded from below (by 0) will converge.
- But will it converge to 0? Yes, think about why.
- What if η is too big? Try $\eta = \frac{1}{2}$ with $w(0) = 1$.

② Widrow-Hoff LMS Algorithm

- Let's apply gradient descent to linear regression. Recall:
- $\nabla R(W) = -2 \sum_{(x,y)\in Data} (y - Wx)x^T$ same shape as W

$W \leftarrow W - \eta \nabla R(W) = W + 2\eta \sum_{(x,y)} (y - Wx)x^T$ absorb

- When $y \in \mathbb{R}^1$ then $W = w^T$ and $y - w^T x \in \mathbb{R}$

The updates become: $w \leftarrow w + \eta \sum_{(x,y)} (y - w^T x)\, x$

- We could do this 1 data point at a time:

For each $(x,y) \in$ Data: $w \leftarrow w + \eta (y - wx)\, x$

Similar to the perceptron learning algorithm!

③ Delta Rule

- Squared loss for single neuron with differentiable $\varphi(\cdot)$:

$R(w) = \sum_{(x,y)} (y - \varphi(w^T x))^2$

$\nabla R(w) = -2 \sum_{(x,y)} (y - \varphi(w^T x))\, \varphi'(w^T x)\, x^T$ chain rule

$w \leftarrow w + 2\eta \sum_{(x,y)} (y - \varphi(w^T x))\, \varphi'(w^T x)\, x$

- 1 data point at a time:

$w \leftarrow w + \eta (y - \varphi(w^T x))\, \varphi'(w^T x)\, x$

Example $\varphi(\cdot)$:

- Sigmoid: $\varphi(v) = \frac{1}{1+e^{-av}}$

$\varphi'(v) = \frac{0 - (-ae^{-av})}{(1+e^{-av})^2} = \frac{ae^{-av}}{(1+e^{-av})^2} = a\varphi(v)(1-\varphi(v))$

since $1 - \varphi(v) = \frac{e^{-av}}{1+e^{-av}}$

- ReLU: $\varphi(v) = \begin{cases} v & v \geq 0 \\ 0 & \text{otherwise} \end{cases}$

$\varphi'(v) = \begin{cases} 1 & v \geq 0 \\ 0 & \text{otherwise} \end{cases} = \text{step}(v)$

ECE/CS 559 Lecture 9 & 10 T,Th 9/24, 9/26

Last time: Gradient Descent; Widrow-Hoff LMS Algorithm

$$\text{minimize}_w \; R(w) = \sum_{(x,y)\in Data} \ell(y, f(x;w))$$

$$w' \leftarrow w - \eta \nabla R(w) \qquad \nabla_w R = \left[\frac{\partial R}{\partial w_1}, \dots, \frac{\partial R}{\partial w_k}\right]$$

- $\ell(y, f(x;w)) = \| y - f(x;w) \|^2$

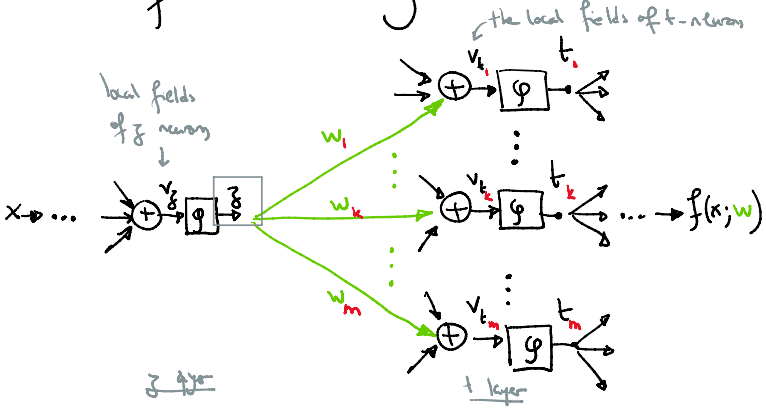
$f(x;w) = Wx$

$$W' \leftarrow W - \eta \nabla R(w) = W + 2\eta \sum_{(x,y)} (y - Wx) x^T$$

(absorb)

Back propagation

- So far we've been calculating gradients with 1 neuron.
- What if we connect many neurons?



- We will apply the chain rule multiple times:

$$\frac{\partial R}{\partial w_k} = \sum_{(x,y)} \left[\frac{\partial \ell}{\partial w_k}\right] \quad (?)$$

need [this]

(?) $\frac{\partial \ell}{\partial w_k} = \frac{\partial}{\partial w_k} \ell(t_k, \dots)\Big|_{t_k = \varphi(w_k z + \dots)}$ (v_{t_k})

have this from forward pass

(two chain rules) $= \left[\frac{\partial \ell}{\partial t_k}\Big|_{t_k=\varphi(v_{t_k})}\right] \cdot \varphi'(v_{t_k}) \cdot z$ (*)

need [this] at every neuron output

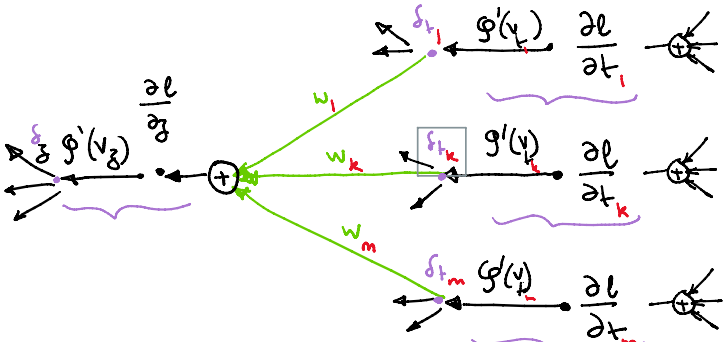
δ_{t_k}

(*) $\frac{\partial \ell}{\partial z}\Big|_{z=\varphi(u)} = \frac{\partial}{\partial z} \ell(t_1, \dots, t_k, \dots, t_m)\Big|_{t_k=\varphi(w_k z + \dots)}$ (v_{t_k})

(two chain rules) $= \sum_{k=1}^{m} \frac{\partial \ell}{\partial t_k}\Big|_{t_k=\varphi(v_{t_k})} \cdot \varphi'(v_{t_k}) \cdot w_k$

δ_{t_k}

- To calculate (?) and (*), we perform a forward pass to get all the outputs $(u, z, v_t, t, \dots)$, then we perform a backward pass:



Gradient of w = forward signal z × backward signal δ_t

- Of course, the layer of z has other (say n) neurons. Let's index the equations for these (blue j)

$$\frac{\partial \ell}{\partial w_{kj}} = \delta_{t_k} \cdot z_j \qquad \frac{\partial \ell}{\partial z_j} = \sum_{k=1}^{m} \delta_{t_k} w_{kj} \qquad j = 1, \dots, n$$

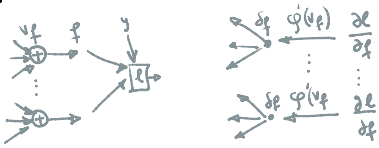
- If we think of δ_t as a m-dim vector
  of z as an n-dim vector
  of W as a $m \times n$-dim matrix
- We get the linear algebraic backward equation:

$$\nabla_W \ell = \delta_t z^T \qquad \nabla_z \ell = W^T \delta_t \qquad \delta_z = \nabla_z \ell \bullet \varphi'(v_z)$$

outer product; matrix multiplication; elementwise product

- To complete this, we need 2 things: how to start, how to handle bias
- Start at the last layer $\nabla_f \ell$ depends on the choice of ℓ

e.g. if $\ell = \|f - y\|^2$ then $\nabla_f \ell = 2(f - y)$, $\delta_f = \nabla_f \ell \bullet \varphi'(v_f)$

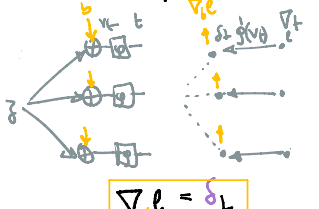


- Biases are like dead-ends on the backward path.

$$\frac{\partial \ell}{\partial b} = \frac{\partial}{\partial b} \ell(t_k, \dots)\Big|_{t_k=\varphi(\dots + b)} \quad (v_{t_k})$$

$$= \frac{\partial \ell}{\partial t}\Big|_{t_k=\varphi(v_t)} \cdot \varphi'(v_{t_k}) \cdot 1$$

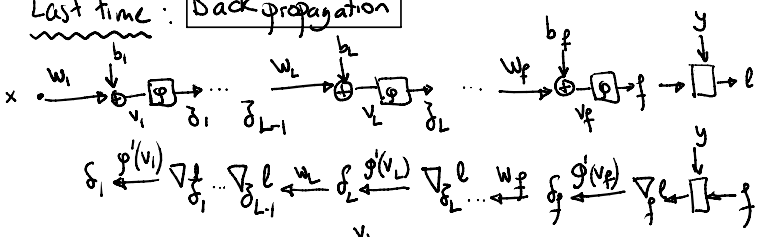
δ_{t_k}



Note: Can we forward eqs to get these: $\ell(t) = \ell(\underbrace{\varphi(w_j z + b)}_{t})$ $\Rightarrow$ $\nabla_w \ell = [\underbrace{\nabla_t \ell \bullet \varphi'(v_t)}_{\delta_t}] z^T$, $\nabla_b \ell = [\underbrace{\nabla_t \ell \bullet \varphi'(v_t)}_{\delta_t}] \cdot 1$, $\nabla_z \ell = W^T [\underbrace{\nabla_t \ell \bullet \varphi(v_t)}_{\delta_t}]$

ECE/CS 559 Lecture 11 T 10/1

Last time: **Backpropagation**



Forward: $\ell(z_L) = \ell\big(\varphi\big(\underbrace{W_L z_{L-1} + b_L}_{v_L}\big)\big)$

Backward: $\nabla_{z_{L-1}} \ell = W_L^T \big(\varphi'(v_L) \circ \nabla_{z_L} \ell\big) = W_L^T \delta_L$ (element wise)

Gradients: $\nabla_{W_L} \ell = \big(\varphi'(v_L) \circ \nabla_{z_L} \ell\big) z^T = \delta_L z^T$, $\nabla_{b_L} \ell = \delta_L$

- When learning neural networks, issue is that we train on the observed contexts x. However, we mostly care about performance on new unobserved contexts!
- Performing well in unobserved contexts = **generalization**
  Overfitting prevents generalization (unless it's mild)
- How do we quantify this?

  Training Data $= \{(x,y), \dots\}$ ↳ use this to build NN

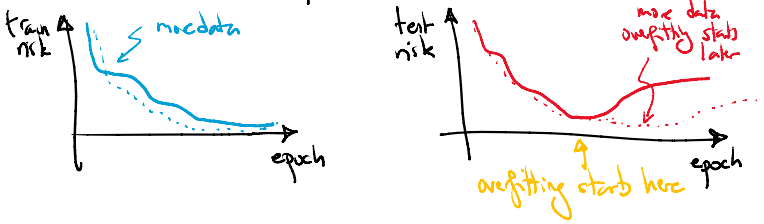
  Testing Data $= \{(x',y'), \dots\}$ ↳ use this to assess generalization
- We must be careful!
  We can't keep on using testing data to refine the NN, otherwise we may overfit it too and not generalize to newer data.

① Overfitting

- Tendency to overexplain the training data.
- Overexplain = when many explanations describe experience equally well, choose one that adds many complicated details and twists (typically what conspiracy theories do)
- Example: polynomial fitting

- Typical property leading to this: high sensitivity to changes in data

- Typical behavior of train vs. test risk:



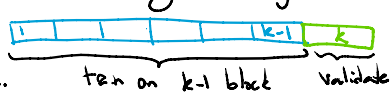
- Observations:
  - Stopping early could help alleviate overfitting
    Why? The NN will not fit the noise.
    How? By staying closer to its initialization (staying "simpler")
  - More training data reduces overfitting, delays its onset.

② Cross-Validation

- Since training risk is not a good enough proxy to testing risk, we could hold out some of the training data for validation:

Training Data $= \{(x,y), \dots\}$ (train on this often) Validation Data $= \{(x'',y''), \dots\}$ (check if okay only sometimes) Testing Data $= \{(x',y'), \dots\}$ (test on this only once)

- What is validation good for?
  - Choose what epoch to stop at.
  - Choose between possible architectures, learning rates, etc. ("hyper parameters")
  - These choices are "coarser" than choosing the weights & biases
- k-Fold Cross-validation:



Change validation block, repeat. Average.

③ Regularization

- Cross-validation helps see if we overfit. Regularization helps prevent it.
- The word comes from inverse problems in physics:
  - To decide between many possible explanations, we need to favor some.
  - We tend to favor "simple" ones (Occam's Razor) (e.g., low energy)
- E.g., For polynomials and NNs, favor small weights:

  $\min_w R(w)$ for $\|w\|^2 \le C$ $\iff$ $\min_w R(w) + \lambda \|w\|_2^2$ (regularized/penalized risk)
- What does it mean? Prefer smoother functions: (sparser)
- Why does it work? Can't overfit noise
  Regularized risk is a better proxy for test-risk
  $\|w\|_1 = |w_1| + \dots + |w_n|$ (other choice)
- How does it affect backpropagation? Simple!

  $\nabla_w \big(R(w) + \lambda \underbrace{\|w\|_2^2}_{\|w\|_2}\big) = \nabla_w R + 2\lambda w$ ($\operatorname{sign}(w)$)

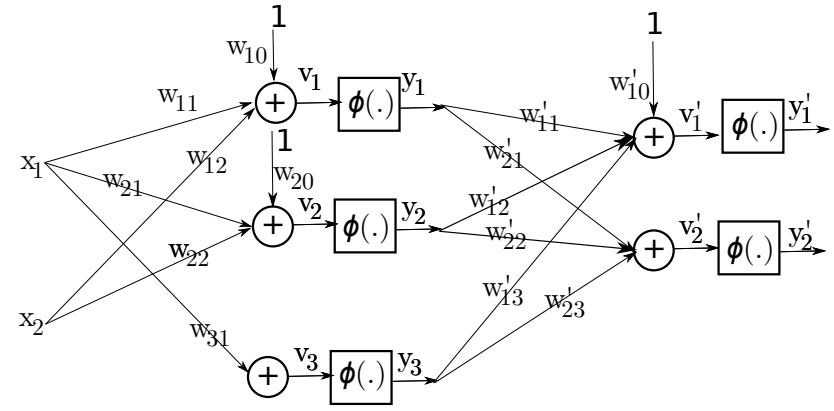
Note: Any change that helps prefer some w over others is regularization. This introduces what we call inductive bias, which (if right) improves generalization.

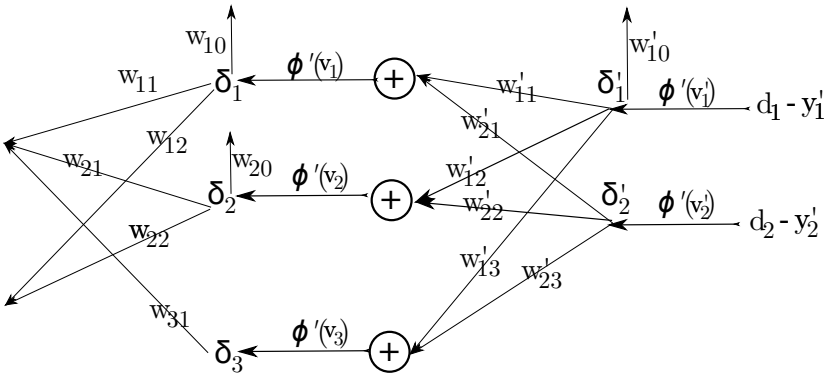
ECE/CS 559 - The Backpropagation Algorithm

Erdem Koyuncu

1 The algorithm

- Developed in the 1960s-1970s.
- Basically a method to calculate the partial derivatives (with respect to individual weights) of a multi-layer feedforward network.
- We begin with an example that shows how the algorithm works.

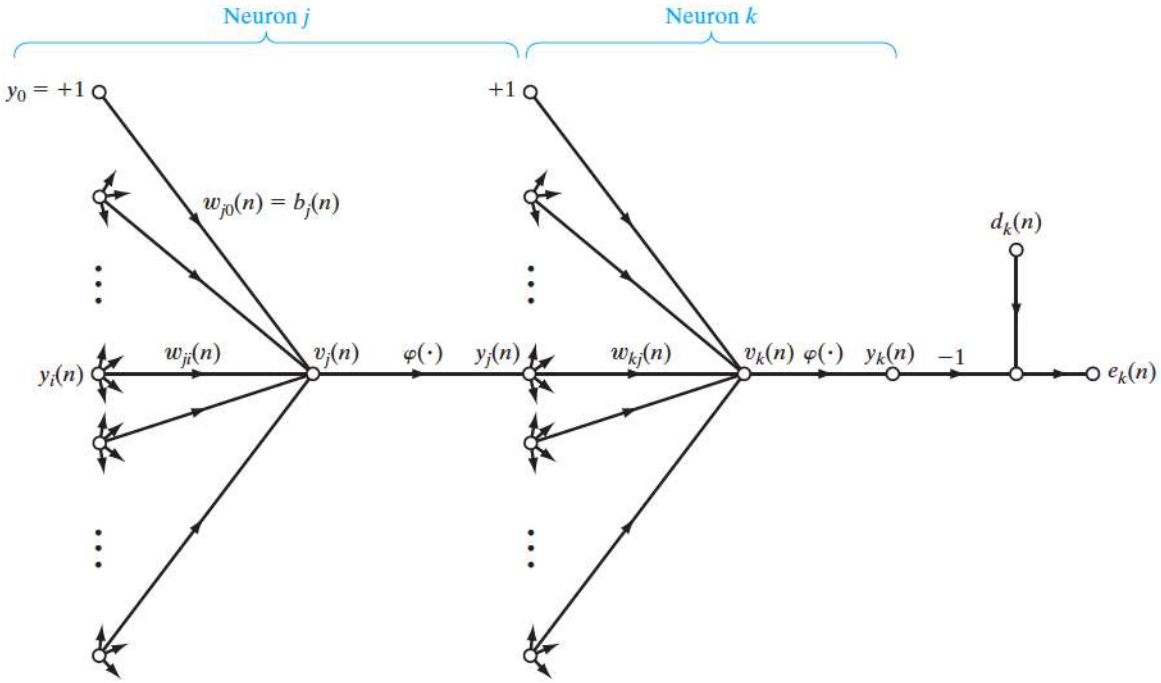




- Here we have input $\mathbf{x} = [x_1\ x_2]^T$ and the corresponding outputs $f(\mathbf{x}, \mathbf{w}) \triangleq \mathbf{y}' \triangleq [y'_1\ y'_2]^T$. We wish to calculate the derivatives of $E = \frac{1}{2}\|\mathbf{d} - \mathbf{y}'\|^2$ with respect to the weights.
- First, we feed pattern $\mathbf{x}$ to the network and obtain all the output information as shown in the first figure.
- We then do the backpropagation as shown in the second figure.
- $\frac{\partial E}{\partial w} = -$ (the signal before multiplication by w in the feedforward network) $\times$ (the signal before multiplication by w in the feedback network).
- Note that we would then do the update $w \leftarrow w - \eta \frac{\partial E}{\partial w}$ in the gradient descent.

2 Why does it work?

- We simply use the chain rule of calculus for the calculation. The algorithm just reveals itself.
- Chain Rule: $\frac{\partial f(g(x))}{\partial x} = f'(g(x))g'(x)$.
- Now consider the final two layers of the network (with a different notation):



(Image taken from the course book (Haykin))

- Let us calculate the partial derivative for any one of the weights connecting a neuron in the second last layer to the last layer. We have

$$\frac{1}{2}\frac{\partial E}{\partial w_{k'j'}} = \frac{1}{2}\frac{\partial \sum_{k\in K}(d_k - y_k)^2}{\partial w_{k'j'}} \tag{1}$$

$$= -\sum_{k\in K}(d_k - y_k)\frac{\partial y_k}{\partial w_{k'j'}} \tag{2}$$

$$= -\sum_{k\in K}(d_k - y_k)\frac{\partial \phi(\sum_{j\in J} w_{kj}y_j)}{\partial w_{k'j'}} \tag{3}$$

$$= -\sum_{k\in K}(d_k - y_k)\phi'(v_k)\frac{\partial \sum_{j\in J} w_{kj}y_j}{\partial w_{k'j'}} \tag{4}$$

$$= -\sum_{k\in K}(d_k - y_k)\phi'(v_k)y_{j'}\mathbf{1}(k = k',\, j = j') \tag{5}$$

$$= -(d_{k'} - y_{k'})\phi'(v_{k'})y_{j'} \tag{6}$$

$$\triangleq -\delta_{k'}y_{j'} \tag{7}$$

Hence,

$$\frac{1}{2}\frac{\partial E}{\partial w_{kj}} = -e_k\phi'(v_k)y_j = -y_j\delta_k \tag{8}$$

where $\delta_k \triangleq (d_k - y_k)\phi'(v_k)$.

- Similarly, we can calculate the partial derivative for any one of the weights connecting a neuron in the third last layer to the second layer. We have

$$\frac{1}{2}\frac{\partial E}{\partial w_{ji}} = \frac{1}{2}\frac{\partial \sum_{k\in K}(d_k - y_k)^2}{\partial w_{ji}} \tag{9}$$

$$= -\sum_{k\in K}(d_k - y_k)\frac{\partial y_k}{\partial w_{ji}} \tag{10}$$

$$= -\sum_{k\in K}(d_k - y_k)\frac{\partial \phi(\sum_{j\in J} w_{kj}y_j)}{\partial w_{ji}} \tag{11}$$

$$= -\sum_{k\in K} e_k\phi'(v_k)\sum_{j\in J} w_{kj}\frac{\partial y_j}{\partial w_{ji}} \tag{12}$$

$$= -\sum_{k\in K} e_k\phi'(v_k)\sum_{j\in J} w_{kj}\frac{\partial \phi(\sum_{i\in I} w_{ji}y_i)}{\partial w_{ji}} \tag{13}$$

$$= -\sum_{k\in K} e_k\phi'(v_k)w_{kj}\phi'(v_j)y_i \tag{14}$$

$$= -y_i\phi'(v_j)\sum_{k\in K}\delta_k w_{kj} \tag{15}$$

$$\triangleq -y_i\delta_j \tag{16}$$

- If there were another layer (say indexed via $z \in Z$), we would go

$$= -\sum_{k\in K} e_k\phi'(v_k)\sum_{j\in J} w_{kj}\phi'(v_j)w_{ji}\phi'(v_i)y_z \tag{17}$$

$$= -y_z\phi'(v_i)\sum_{j\in J}\delta_j w_{ji} \tag{18}$$

$$\triangleq -y_z\delta_i \tag{19}$$

  and so on.

3 Matrix representation of the algorithm

- The following matrix representation is useful for computation purposes.
- Suppose we have m_0 input nodes, m_1 nodes at layer 1, m_2 nodes at layer 2, and finally m_L nodes at layer L. Let $L \geq 1$.
- The input signal is $\mathbf{y}_0 \triangleq \mathbf{x} \in \mathbb{R}^{m_0\times 1}$.
- Neural network weight matrices $\mathbf{W}_\ell \in \mathbb{R}^{m_\ell \times (m_{\ell-1}+1)}$ including the biases.
- Then, the induced local fields are $\mathbf{v}_\ell \triangleq \mathbf{W}_\ell \begin{bmatrix} 1 \\ \mathbf{y}_{\ell-1} \end{bmatrix} \in \mathbb{R}^{m_\ell \times 1}$, $\ell = 1, \ldots, L$.
- And the outputs are $\mathbf{y}_\ell \triangleq \phi(\mathbf{v}_\ell)$, $\ell = 1, \ldots, L$.
- Define $E = \frac{1}{2}\|\mathbf{d} - \mathbf{y}_L\|^2$, where $\mathbf{d}$ is some certain desired output vector.
- The goal is to calculate $\frac{\partial E}{\partial [\mathbf{W}_\ell]_{i,j}}$ for the purpose of using it with gradient descent.

- Backpropagation equations:

$$\begin{aligned}\boldsymbol{\delta}_L &\triangleq (\mathbf{d} - \mathbf{y}_L) \cdot \phi'(\mathbf{v}_L) \\ \boldsymbol{\delta}_\ell &= \underline{\mathbf{W}_{\ell+1}^T \boldsymbol{\delta}_{\ell+1}} \cdot \phi'(\mathbf{v}_\ell),\ \ell = 1, \ldots, L-1 \\ \frac{\partial E}{\partial \mathbf{W}_\ell} &= -\boldsymbol{\delta}_\ell \begin{bmatrix} 1 \\ \mathbf{y}_{\ell-1} \end{bmatrix}^T,\ \ell = 1, \ldots, L.\end{aligned}$$

- In the above equations, $\cdot$ is the Kronecker/elementwise product, i.e., for vectors $a = [a_1 \cdots a_n]^T$ and $b = [b_1 \cdots b_n]^T$, we define $a \cdot b = [a_1 b_1 \cdots a_n b_n]^T$. Also, $\underline{a}$ is the "lose my first component operator", i.e. $\underline{a} = [a_2 \cdots a_n]$. The reason you need this operator is because of the biases: once you backpropagate signals for the biases, you no longer need these signals and have to destroy them...

- Hence, the weight updates for online learning would be

$$\mathbf{W}_\ell \leftarrow \mathbf{W}_\ell + \eta \boldsymbol{\delta}_\ell \begin{bmatrix} 1 \\ \mathbf{y}_{\ell-1} \end{bmatrix}^T,\ \ell = 1, \ldots, L$$

.

- Note that in the expression for E, usually we have a summation over a training sequence (over different input patterns, and their corresponding desired outputs, i.e. $E = \frac{1}{2|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} \|\mathbf{d}_i - \mathbf{y}_{i,L}\|^2$.)

- So, given η, for epochs = 1 to infinity, for $i = 1$ to training samples, do the forward and backward computations given the training example $\mathbf{x}_i$ as shown above, and perform the corresponding weight updates, loop until some stopping criterion is met. The criterions may be:
  - When the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.
  - When the absolute rate of change in the average squared error per epoch is sufficiently small.

4 Heuristics to make backpropagation perform better

- **Stochastic vs batch update:** Stochastic (online) mode of learning is computationally faster than batch (offline) learning. This is because, in stochastic learning, you make use of the redundancy in the training set. For example, suppose you have 1000 images, 10 copies of the same pattern, then online learning will approximately be 10x faster than offline learning. Even though real life samples do not contain identical images, a similar idea applies. So, the advantage of stochastic learning is much faster convergence on large data set that contain a lot of redundancy. Also, a stochastic trajectory of weight allows escape from local minima. The disadvantage of stochastic learning is that you typically keep bouncing around a local minima unless learning rate is reduced. Moreover, theoretical conditions for convergence are not as trivial. For batch learning, one can easily guarantee convergence to local minimum under simple conditions. But, batch learning is slow on large problems with large datasets.

- **Maximizing information content:** Use an example that results in the largest training error, or use an example that is radically different than all those previously used.

- Usually odd functions perform better than other types of activation functions. $\phi(-v) = -\phi(v)$. The tangent hyperbolic $\beta \tanh \alpha v$ satisfies this property. Some authors even propose suitable values for α, β for general purposes, e.g. $1.7159 \tanh \frac{2}{3} v$. This satisfies $\phi(1) = 1$ and $\phi(-1) = -1$. The second derivative of the function attains its maximum at 1.

- Target values. Obviously, the target values should be within range of the activation functions. In fact, there should be a certain guard interval to avoid saturation or divergence problems. In fact, if you set the target values on the asymptotes of the sigmoid, (1) the output weights will be driven to $-\infty$ or $+\infty$, (and thus saturate), (2), and therefore, if a training sample does not saturate the outputs (this can be an outlier, for example), it will produce enormous gradients due to large weights, resulting in

incorrect updates. and (3) outputs will tend to be binary even when they are wrong - a mistake that will be very difficult to correct. To avoid saturation, people sometimes pick activation functions like $\tanh x + \alpha x$.

- Normalizing the inputs: The training samples should be preprocessed so that their mean values are close to 0. E.g. if the training inputs are strictly positive, the weights of a neuron in the first hidden layer can only increase together or decrease together (not enough degree of freedom to move anywhere we like). For example, think about weights w_1, w_2 in the two dimensional plane. You can only move to the NE direction or the SW direction. If you then wanted to move towards SE or NW you can only do so by zigzagging.

  The input variables should be uncorrelated (to avoid correlations in the different weights of the neuron); this can be done via what is called principal component analysis (PCA) to be discussed later.

  The input variable should also have equal variances, so that the neurons will learn at approximately the same speed.

- Weight initialization: Large weights saturate the units, leading to small gradients and thus slow learning. Small weights correspond to a very flat are of the error surface and also lead to slow learning. For example, think about learning in a one input, one output, many neurons in a hidden layer network with all weights initialized to 0 with tanh activation function. The weights will remain as 0 except for the bias of the output neuron.

  For a heuristic for a good initialization of weights, suppose that the induced local field of neuron j is say $v_j = \sum_{i=1}^{m} w_{ji} y_i$. Suppose the inputs applied to each neuron in the net has 0 mean and variance 1, i.e. $E[y_i] = 0$, and $E[(y_i - E[y_i])^2] = E[y_i^2] = 1$. Also, assume y_i, y_k are uncorrelated, and suppose the synaptic weights are drawn from a distribution with 0 mean and variance σ^2. Then, the mean and variance of the induced local field is $E[v_j] = 0$ and $E[v_j^2] = E[\sum_i \sum_k w_{ji} w_{jk} y_i y_k] = \sum_i \sum_k E[w_{ji} w_{jk}] E[y_i y_k] = \sum_i \sigma^2 1 = m\sigma^2$. A good point to put $m\sigma^2$ is between the linear and saturation parts of the activation function. This point was point 1 for the special tanh function discussed above, so we obtain $\sigma^2 = \frac{1}{m}$.

- Learning rates. All neurons in the multilayer perceptron should ideally learn at the same rate. The last layers usually have larger local gradients than the layers at the front end of the network. Hence, the learning-rate parameter η should be assigned a smaller value in the last layers than in the front layers of the multilayer perceptron. Neurons with many inputs should have a smaller learning-rate parameter than neurons with few inputs so as to maintain a similar learning time for all neurons in the network. It is often suggested that for a given neuron, the learning rate should be inversely proportional to the square root of synaptic connections made to that neuron.

- The main problem with having too many problems is usually the vanishing or exploding gradient problems: Think about a line network with $y = f(w_1 f(w_2 f(w_3 f(w_4 x))))$ for example. To update w_4, we have $w_4 \leftarrow w_4 +$ many f's and $w_1 w_2 w_2$ product. f's are usually less than 1 and w_is are initialized to have mean less than 1: We have the vanishing gradient problem. Otherwise, we have exploding gradient.

  The real problem is the unstable gradients as a result of having many multiplications.

5 Overfitting

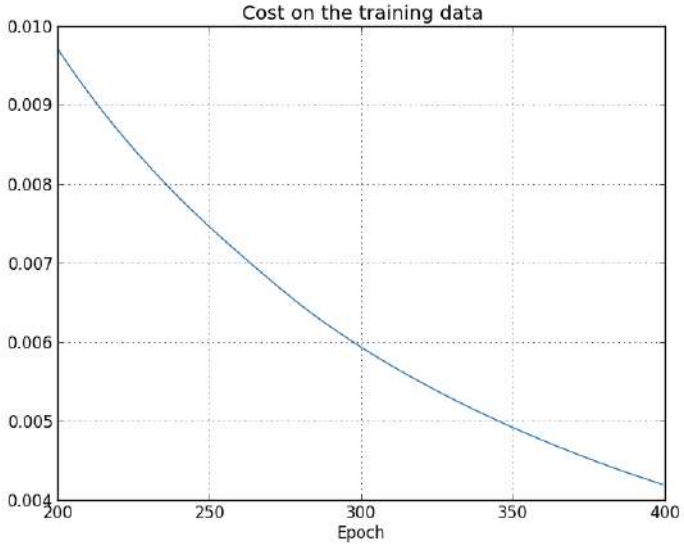
- A model with a large number of free parameters can describe a wide array of phenomena. Von Neumann: "With four parameters I can fit an elephant and with five I can make him wiggle his trunk."

- Just because the model fits so well with the available data does not make it a good model.

- For example, samples of a straight line; one intuitive way to fit it is via a straight line (which requires 2 params), while another way to fit is is via a crazy curve (if you have a sufficiently large number of

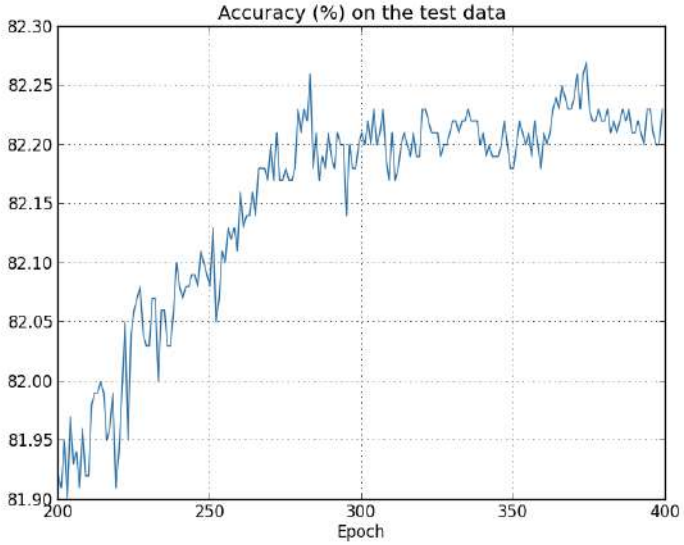
params). Occam's razor: Among competing hypotheses, the one with the fewest assumptions should be selected. It should be mentioned that sometimes a complex explanation is the correct one.

- If we for example train our neural network (with backpropagation) using a training set with a few samples (say 100s or thousand), we will see that the energy function (mean-squared error) will be smoothly decaying to zero as the epochs increase:



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

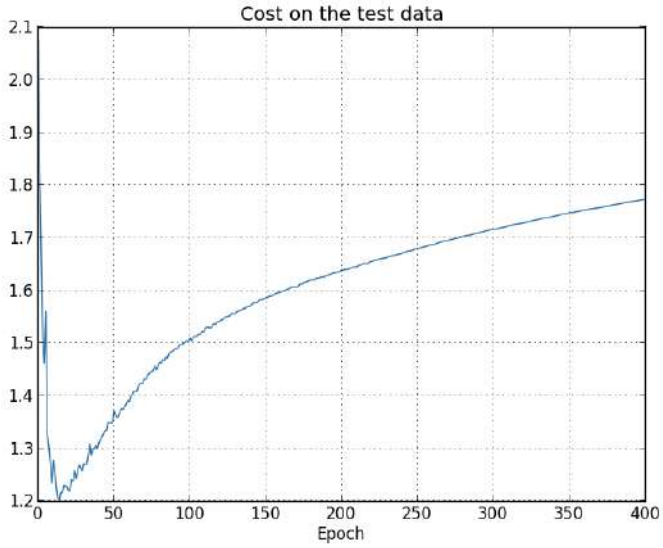
- When the resulting network is actually tested however, one observes a non-zero (and actually high) inaccuracy on the test data:



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

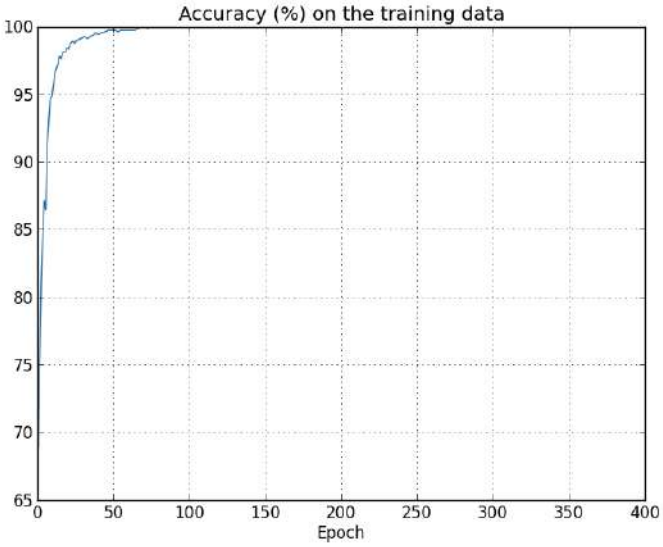
- From the training data, it appears that the network is getting better, although, on the test data the performance of the network saturates and fluctuates around a fixed value after around 280 epochs. We say that the network is overfitting after 280 epochs.

- You may think that the it is not fair to look at MSE on one graph and percent inaccuracy on another graph, but in fact, if we draw the cost on test data, we see a similar behavior:



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

- Now the points where things get worse are different if we take MSE or % inaccuracy as performance metric, but since we care about % inaccuracy at the end, it makes more sense to define the point where we start overtraining as 280.
- Another sign of overfitting is when we plot % inaccuracy on the test data:



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

- So our network really is learning about peculiarities of the training set (as if it is just memorizing the training set), without "understanding" the problem well-enough to solve/classify a previously-not-shown test example.
- **One of the simplest ways to prevent overfitting is to stop training when the precent errors (accuracy) in the test set stop improving.** In practice, one continues training a little more to be confident that the accuracy has in fact saturated.
- Now, say from one experiment, you got one set of weights (you made sure you prevented overfitting). You may want to play with different values of η, the initial condition, the network topology etc to further optimize the performance. These parameters are called the hyperparameters of the optimization

problem. For each hyperparameter, the achievable performance and the corresponding weights will be different. Obviously at the end, you will pick the weights that provide the lowest final classification error on the training set. On the other hand, by trying many different hyperparameters, we fall into a danger of this time overfitting with respect to the hyperparameters. That is why the available data set is usually broken down further to three parts called the training data, validation data and the test data. Usually the training, validation, test data follow a 5:1:1 scale ratio in terms of their size.

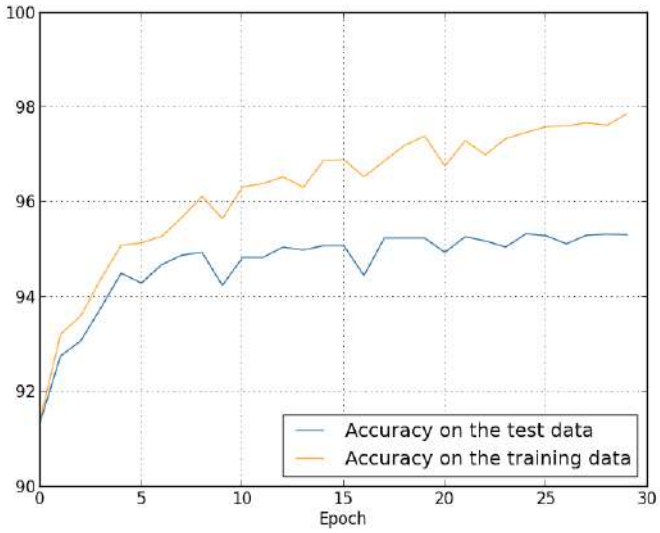
- **Cross-validation and the hold-out method:** Hence, we train the network using the training data as usual. We compute the classification accuracy on the validation data at the end of each epoch. Once the classification accuracy on the validation data has saturated/stops to get worse, we stop training. This strategy is called early stopping. We do this for different hyperparameters, and then pick the hyperparameters + weights that provide the best accuracy on the validation data. We can now use these parameters on the actual test data to see how well we perform (Now we are more confident that we do not do overfitting). **This called the hold-out method (the validation data is held out from the training data. This is thus another way to prevent overfitting**. The entire procedure is called cross-validation.

- Now what if the performance on the test data is unsatisfactory? We will go back to the drawing board, train using new hyperparameters and test on the data set. But then, at least technically, we can overfit on the test data! Fortunately, that seldom happens in practice, and the above 3-set procedure works fine. In fact, typically one does not even have enough data so that one just uses 2-set approach consisting of a training set and a test set.

- **Multi-fold cross validation:** Note that there are other variants of cross-validation. We may use multifold cross-validation by dividing the available set of N examples into K subsets, where $K > 1$; this procedure assumes that K is divisible into N. The model is trained on all the subsets except for one, and the validation error is measured by testing it on the subset that is left out. This procedure is repeated for a total of K trials, each time using a different subset for validation. The performance of the model is assessed by averaging the squared error under validation over all the trials of the experiment.There is a disadvantage to multifold cross-validation: It may require an excessive amount of computation, since the model has to be trained K times.

- **Leave-one-out method:** When the available number of labeled examples, N, is severely limited, we may use the extreme form of multifold cross-validation known as the leave-one-out method. In this case, N - 1 examples are used to train the model, and the model is validated by testing it on the example that is left out. The experiment is repeated for a total of N times, each time leaving out a different example for validation.The squared error under validation is then averaged over the N trials of the experiment.

- Now we were talking about ways to prevent overfitting. **Another way is obviously to use a larger test set:**



(Image taken from http://neuralnetworksanddeeplearning.com/chap3.html)

We see that there is still overfitting going on but it is on a much less serious scale: The performance with the test data and the training data are much closer. Unfortunately, training data is expensive so using a larger data set is not always an option.

- **Another way to prevent overfitting is regularization.**

6 Regularization

- We know how to determine the partial derivatives of $E(\mathbf{w})$ using the backpropagation algorithm. We then use gradient descent to update the weights.
- L_2 regularization: Consider the following function $E'(\mathbf{w}) \triangleq E(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$. The weight updates are $\mathbf{w} \leftarrow \mathbf{w} - \eta\nabla\mathbf{E}(\mathbf{w}) - \lambda\eta\mathbf{w} = (1-\eta\lambda)\mathbf{w} - \eta\nabla\mathbf{E}(\mathbf{w})$.
- Why does this work? Suppose our network mostly has small weights, as will tend to happen in a regularized network. The smallness of the weights means that the behaviour of the network won't change too much if we change a few random inputs here and there. That makes it difficult for a regularized network to learn the effects of local noise in the data.
- Also, small weights imply a usually lesser chance of saturation of neurons, implying better performance.
- L_1 **regularization**: $E'(\mathbf{w}) \triangleq E(\mathbf{w}) + \lambda\|\mathbf{w}\|_1$. We have $\mathbf{w} \leftarrow \mathbf{w} - \lambda\text{sgn}(\mathbf{w}) - \eta\nabla\mathbf{E}(\mathbf{w})$
- **Dropout:** Get first training sample, remove randomly half the neurons in the hidden layers (or some fraction of), forward and back propagate. Restore all connections. Get second training sample, do the same thing, again and again.

  Why does this work? We are trying to make the system robust to loss of neurons. Like brain damage.
- **Artificially expanding the training data:** Small rotations of images, elastic distortions (emulate random oscillations found in hand muscles). Or, if you are looking for a speech recognizer, you can add random noise to your voice samples to expand the training data.
- **Momentum method:** Initialize a velocity vector to 0, and then perform the updates $\mathbf{v} \leftarrow \mu\mathbf{v} - \eta\nabla E(\mathbf{w})$ with $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$. The factor $(1-\mu)$ can be thought of as the friction of the descent, you build speed as you roll down the valley, that may improve your speed of convergence (you avoid the slowness of gradient descent as you reach the global minimum), but you may overshoot if μ is too large.

ECE/CS 559 Lecture 13 T 10/8

Last time: Regularization

- Idea: Avoid overexplaining by limiting to reasonable explanations: prior knowledge / constraints on weights / biases
- $\min_w R(w)$ for $\|w\|_2 \le C$ $\Leftrightarrow$ $\min_w R(w) + \lambda \|w\|_2^2$ (L_2)
- During gradient descent simply add: $\nabla_w(R(w) + \lambda\|w\|_2^2) = \nabla_w R + 2\lambda w$ ($\|w\|_1$: L_1, $\text{sign}(w)$)
- Dropout: Randomly prune weights during training (with prob. p). Include all weights (scaled by p) at test time.

① Optimization

ⓐ Stochastic Gradient Descent

Empirical risk $R(w) = \frac{1}{n}\sum_{(x,y)\in Data} \ell(y, f(x;w))$

"True" risk $R'(w) = \mathbb{E}[\ell(Y, f(X;w))]$ — what we really want to minimize, behavior on future points.

Note: $R(w)$ is an unbiased estimate of $R'(w)$: $\mathbb{E}[R(w)] = R'(w)$

This means the gradients are unbiased: $\mathbb{E}[\nabla_w R(w)] = \nabla_w R'(w)$ (random/stochastic gradients)

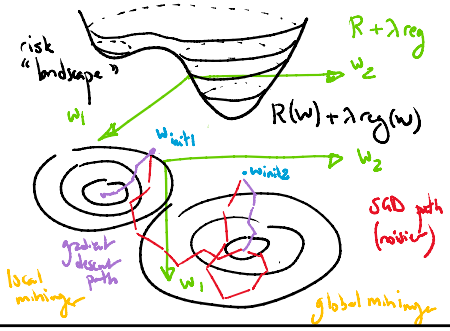
- But... this remains true for even a single point $\mathbb{E}[\nabla_w \ell(Y, f(x;w))] = \nabla_w R(w)$ or a mini batch of points $\mathbb{E}\left[\nabla_w \sum_{(x,y)\in B} \ell(y, f(x;w))\right] = \nabla_w R(w)$
- What changes? The variance. # points ↓ variance ↑ noisiness ↑

Then why don't we always use all the points?

- Reasons for minibatch SGD:
  - Computation: It's faster to get an update with small batches
  - Statistics: Homogeneous data ⇒ small batches give a glimpse of the whole
  - Optimization: The variance can be helpful to get out of local minima.

  (choose batch size to trade off this part with convergence speed.)
- General form of SGD:

```
for epoch in #of epochs:
    t=0 ; Σ∇ = 0
    For (x,y) in Data:
        Forward(x) ; Backward(y)   gives ∇R + λ∇reg
        Σ∇ += ∇R + λ∇reg ; t += 1
        if (t+1) % B == 0:
            W += -η Σ∇
            Σ∇ = 0
```



ⓑ Choice of η, the "learning rate"

- Because SGD introduces noise, η should be adjusted
  - Otherwise, performance can worsen/improve/worsen (oscillate)
- Typically: Reduce η by a factor (e.g. 90%) if sustained worsening.
- Advanced: Adapt to the landscape (e.g. flatness, steepness)
- Vary η by layer: Later layers ∇_w ↑, make η ↓
  by neuron: # Inputs ↑, make η ↓ $\propto 1/\sqrt{\#\text{ inputs}}$ (heuristic)

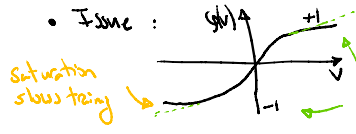
ⓒ Controlling instability:

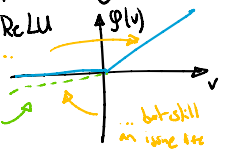
- Gradients are repeatedly multiplied during back propagation.
  - As a result, they could vanish ($\times 0.9 \times 0.9 \times \ldots$) or explode ($\times 1.1 \times 1.1 \times \ldots$)
  - Solution: normalize gradients (across a batch or a layer), clip gradients

② Heuristics

The "craft" of training neural networks.

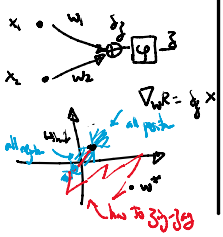
ⓐ Choosing the activation function:

- For the longest time: Sigmoid σ, hyperbolic tangent tanh
  - Issue: 

  - Instead: ReLU 

ⓑ Preprocessing inputs:

- Ideally, input coordinates are uncorrelated/centered
  - Otherwise, may slow down learning
  - Example: if x_i's are always + or − all together
- Preprocess to make them have 0 mean, unit variance, 0 covariance: "whitened"



- How? $\mu = \frac{1}{|Data|}\sum_{x\in Data} x$ (mean (vector)) $C = \frac{1}{|Data|}\sum_{x\in Data}(x-\mu)(x-\mu)^T$ (covariance (matrix))

C is positive semidefinite: can be diagonalized $C = UDU^T$, $UU^T = U^TU = I$

Let $x_{new} = D^{-\frac{1}{2}}U^T(x-\mu)$ ← has 0 mean, covariance = I

ⓒ Initializing the weights:

- Intuition: maintain similar statistics from layer to layer.
- Ignoring activation, assuming whitened inputs, covariance I, next layer's inputs is:

$$C_i = \frac{1}{|Data|}\sum_{x\in Data}(Wx)(Wx)^T = WW^T = n\left[\ \ \right]_n\left[\ \ \right]$$

Ideally, we want to make $\mathbb{E}[WW^T] = I$

- If w have 0 mean + uncorrelated, then off-diagonals are 0 ✓
- On the diagonal, we'd be adding n, $\mathbb{E}[w^2]$, so we get $n\,\text{var}(w)$.
- To make it identity, we just have to choose $\text{var}(w) = 1$
- E.g., choose w to be i.i.d. $\mathcal{N}\left(0, \frac{1}{\sqrt{\#\text{ inputs}}}\right)$

ECE/CS 559: Neural Networks **Fall 2024**

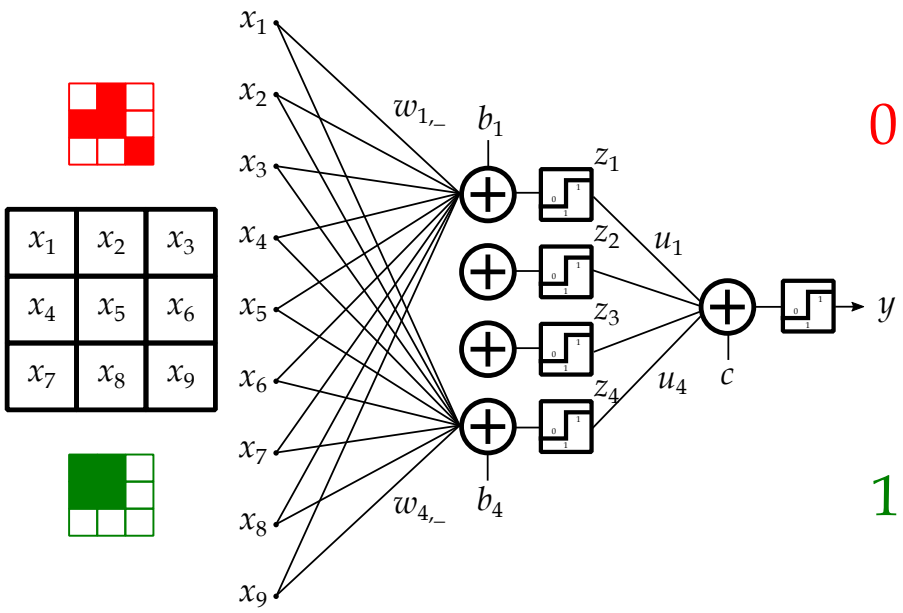
Homework 1

Due: **Tuesday** September 3, 2024 (by 9pm, on Gradescope)

Note: *You may discuss these problems in groups. However, you must write up your own solutions and mention the names of the people in your group. Also, please do mention any solution manuals, online material, or other sources you used in a major way.*

1. **[Logic and Neural Networks]**

   You have a 3×3 binary image (each pixel can be either 0 or 1). You want to build a detector that will output 1 if and only if there's a 2×2 square of 1's in this image. Consider the following *architecture*.



   Let the *inputs* of this neural network be the pixels of the image, labeled $x_1, \cdots, x_9$. These inputs are fed into 4 linear units, which are followed by a step nonlinearity. More precisely, for $j = 1, 2, 3, 4$ we get:

$$z_j = \text{step}_1\left(b_j + \sum_{i=1}^{9} w_{j,i} x_i\right), \text{ where } \text{step}_1(t) := \begin{cases} 0 & ; \quad t < 1 \\ 1 & ; \quad t \geq 1 \end{cases}$$

   These values, z_1, z_2, z_3 and z_4 are not output directly, which is why they are called *hidden*. The equations producing them is the first *layer* of this neural network.

Instead, these hidden values are fed into a similar linear unit followed by a step nonlinearity, to produce the *output*, as follows:

$$y = \mathsf{step}_1\left(c + \sum_{j=1}^{4} u_j z_j\right).$$

This is the second (and final) *layer*. Our goal is to adjust the *weights* $w_{j,i}$ (there are $4 \times 9 = 36$ of these numbers) and u_j (there are 4 of them), as well as the *biases* b_j (there are 4 of them) and c (just 1 number), such that the output detects 2×2 squares correctly ($y = 1$ if there's such a square, 0 if not.)

(a) Show that if, for some j, $w_{j,i}$ are all 1 and $b_j = 0$, then that z_j is the *logical OR* of all the x_i.

(b) Show that in (a), if for some i's you change $w_{j,i}$ to -1 and set b_j to the number of such changed i, then z_j is the logical OR of all the x_i, but *inverted* wherever i is changed.

(c) Use this analogy with logic to figure out how to adjust all 36 $w_{j,i}$, 4 b_j, 4 u_j, and 1 c, to produce the desired output correctly. Explain how you obtained these numbers. (Express $w_{j,i}$ in a 4×9 *matrix*.) [Hint: Break down the detection into four cases, then use DeMorgan's law to specialize each z_j in the first layer to one case, and then use the second layer to combine all four cases.]

2. **[Calculus and Neural Networks]** Let's say we have a complicated function $g(x)$. We would like to approximate it using another function from a family of functions $f(x, w)$, where by changing w we change which element in this family we pick. w is called a *parameter*. To judge how good the approximation is, we need to compare $g(x)$ and $f(x, w)$, using a loss function $\ell(f, g)$. Instead of making this comparison at every x, we typically summarize it by looking at the average loss across all x's, $L = \int \ell(f(x, w), g(x))\mathrm{d}x$. How do we find w to make L as small as possible? Doing this by hand (see Q1) could be tedious. As long as the functions are nice and differentiable, we can instead use the machinery of calculus and optimization to turn the 'knob' of w, until f nicely matches g.

Let's make this concrete. Let $g(x) = x^3$ and let $f(x, w) = wx$ be the linear family. That is, we are trying to approximate a cubic function with a linear function. Let's use the squared loss $\ell(f, g) = (f - g)^2$ to measure the loss. Let's say we only care about the approximation for $x \in [-1, 1]$, which we can take as the limits of L's integral:

$$L(w) = \int_{-1}^{1} \ell(f(x, w), g(x))\mathrm{d}x = \int_{-1}^{1} (xw - x^3)^2 \mathrm{d}x$$

By optimality, we want to find the value of w where $\frac{\mathrm{d}L}{\mathrm{d}w} = 0$. We need to calculate this derivative. By Fubini's theorem, we have:

$$\frac{\mathrm{d}L}{\mathrm{d}w} = \int_{-1}^{1} \frac{\partial}{\partial w} \ell(f(x, w), g(x)) \mathrm{d}x$$

All throughout, note that the inputs x and and target outputs $g(x)$ are just fixed constants that are not part of the differentiation.

(a) Write the chain rule, to see that to calculate $\frac{\partial}{\partial w}\ell$ we need to go *backward*: first calculate the derivative of $\ell(f, g)$ with respect to f, and then combine it with the derivative of $f(x, w)$ with respect to w. (The reason this is called *backward*, is because if the chain were longer, we would complete calculating the gradients of the parameters in each function from the last to the first.)

(b) Calculate those derivatives explicitly, combine them, and integrate them to get $\frac{\mathrm{d}L}{\mathrm{d}w}$. Set this to 0, to find the optimal parameter w.

(c) Plot $g(x)$ and $f(x, w)$ with the w from (b) on the same plot over the range $x \in [-1, 1]$, to see how well the w that you found works. (You can try other w's too, to qualitatively check that yours is better.)

Q1

(a) With this choice we get:

$$f = \begin{cases} 1 \; ; & \sum_i x_i \geq 1 \\ 0 \; ; & \text{otherwise} \end{cases}$$

Therefore f is 1 except when all x's are 0.
Thus the output is True unless all inputs are False.
This is exactly the definition of logical OR:

f represents $x_1 \vee x_2 \vee \cdots \vee x_g$ $(\vee = \text{OR})$

(b) Say we set the first m w's to -1 and the rest to 1, and we let $b = m$, we get:

$$f = \begin{cases} 1 \; ; & -x_1 - x_2 - \cdots - x_m + x_{m+1} + \cdots + x_g + m \geq 0 \\ 0 \; ; & \text{otherwise} \end{cases}$$

$$\Updownarrow$$

$$f = \begin{cases} 1 \; ; & (1-x_1) + (1-x_2) + \cdots + (1-x_m) + x_{m+1} + \cdots + x_m \geq 0 \\ 0 \; ; & \text{otherwise} \end{cases}$$

This shows that x_1 through x_m are now inverted

$$1-x = \begin{cases} 1 & \text{if } x=0 \\ 0 & \text{if } x=1 \end{cases} \iff 1-x \text{ represents } \neg x$$

arithmetic inversion — logical negation

Then, by (a), f now represents:

$$\neg x_1 \vee \neg x_2 \vee \dots \vee \neg x_m \vee x_{m+1} \vee \dots \vee x_9$$

(c) Let case 1 = square in top-left corner

case 2 = top right

case 3 = bottom left

case 4 = bottom right

Case 1 is the same as $x_1 \wedge x_2 \wedge \neg x_3$

AND statement $\rightarrow \wedge\, x_4 \wedge x_5 \wedge \neg x_6$

$\rightarrow \wedge \neg x_7 \wedge x_8 \wedge \neg x_9$

1	1	0
1	1	0
0	0	0

Since we know how to represent OR statements

Let's change this into an OR using De Morgan's.

$$\neg(\text{case 1}) = \neg x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee \neg x_5 \vee x_6 \vee x_7 \vee x_8 \vee x_9$$

$\neg$ (case 1) $= \neg x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee \neg x_5 \vee x_6 \vee x_7 \vee x_8 \vee x_9$

$z_1 = (1-x_1) + (1-x_2) + x_3 + (1-x_4) + (1-x_5) + x_6 + x_7 + x_8 + x_9$

$= -x_1 - x_2 + x_3 - x_4 - x_5 + x_6 + x_7 + x_8 + x_9 + 4$

$w_{1.} = [-1 \ -1 \ 1 \ -1 \ -1 \ 1 \ 1 \ 1 \ 1] \quad b_1 = 4$

(# of negation in (b))

Similarly: 

Case 2 $= \neg x_1 \wedge x_2 \wedge x_3 \wedge \neg x_4 \wedge x_5 \wedge x_6 \wedge \neg x_7 \wedge \neg x_8 \wedge \neg x_9$

$z_2 = \neg$(case 2) $= x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee \neg x_5 \vee \neg x_6 \vee x_7 \vee x_8 \vee x_9$

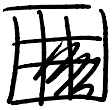
$w_{2.} = [1 \ -1 \ -1 \ 1 \ -1 \ -1 \ 1 \ 1 \ 1]$

$(b_2 = 4)$

Case 3: 

$z_3 = \neg$(case 3) $= x_1 \vee x_2 \vee x_3 \vee \neg x_4 \vee \neg x_5 \vee x_6 \vee \neg x_7 \vee \neg x_8 \vee x_9$

$w_3 = [1 \ 1 \ 1 \ -1 \ -1 \ 1 \ -1 \ -1 \ 1]$

Case 4: 

$(b_3 = 4)$

$z_4 = \neg$(case 4) $= x_1 \vee x_2 \vee x_3 \vee x_4 \vee \neg x_5 \vee \neg x_6 \vee x_7 \vee \neg x_8 \vee \neg x_9$

$w_4 = [1 \ 1 \ 1 \ 1 \ -1 \ -1 \ 1 \ -1 \ -1]$

$(b_4 = 4)$

Putting all together

We want $y =$ (case 1) $\vee$ (case 2) $\vee$ (case 3) $\vee$ (case 4)

(OR statement)

OR statement

since we want to detect if any of these cases occurs.

$$\Rightarrow y = \neg z_1 \vee \neg z_2 \vee \neg z_3 \vee \neg z_4$$

$$\Rightarrow u = [-1 \quad -1 \quad -1 \quad -1] \quad c = 4$$

Thus, we have:

Layer 1 : $$W = \begin{bmatrix} -1 & -1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & -1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 & -1 & 1 & -1 & -1 \end{bmatrix}$$

$$b = \begin{bmatrix} 4 \\ 4 \\ 4 \\ 4 \end{bmatrix}$$

Layer 2 : $u = [-1 \quad -1 \quad -1 \quad -1] \quad c = 4$

Note: This solution is not unique. It's simply the one that follows from the steps outlined in this question.

Q2

(a) Since $\ell(x,w) = \ell(f(x,w), g(x))$, the chain rule gives:

$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial f} \cdot \frac{\partial f}{\partial w} + \frac{\partial \ell}{\partial f} \underbrace{\frac{\partial g}{\partial w}}_{0} = \frac{\partial \ell}{\partial f} \cdot \frac{\partial f}{\partial w}$$

(b) We have $\ell(f,g) = (f-g)^2 \Rightarrow \frac{\partial \ell}{\partial f} = 2(f-g)$

$$f(x,w) = wx \qquad \frac{\partial f}{\partial w} = x$$

Thus, by the chain rule in (a):

$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial f} \cdot \frac{\partial f}{\partial w} = 2(f(x,w) - g(x)) \cdot x$$
$$= 2(wx - x^3)x$$
$$= 2wx^2 - 2x^4$$

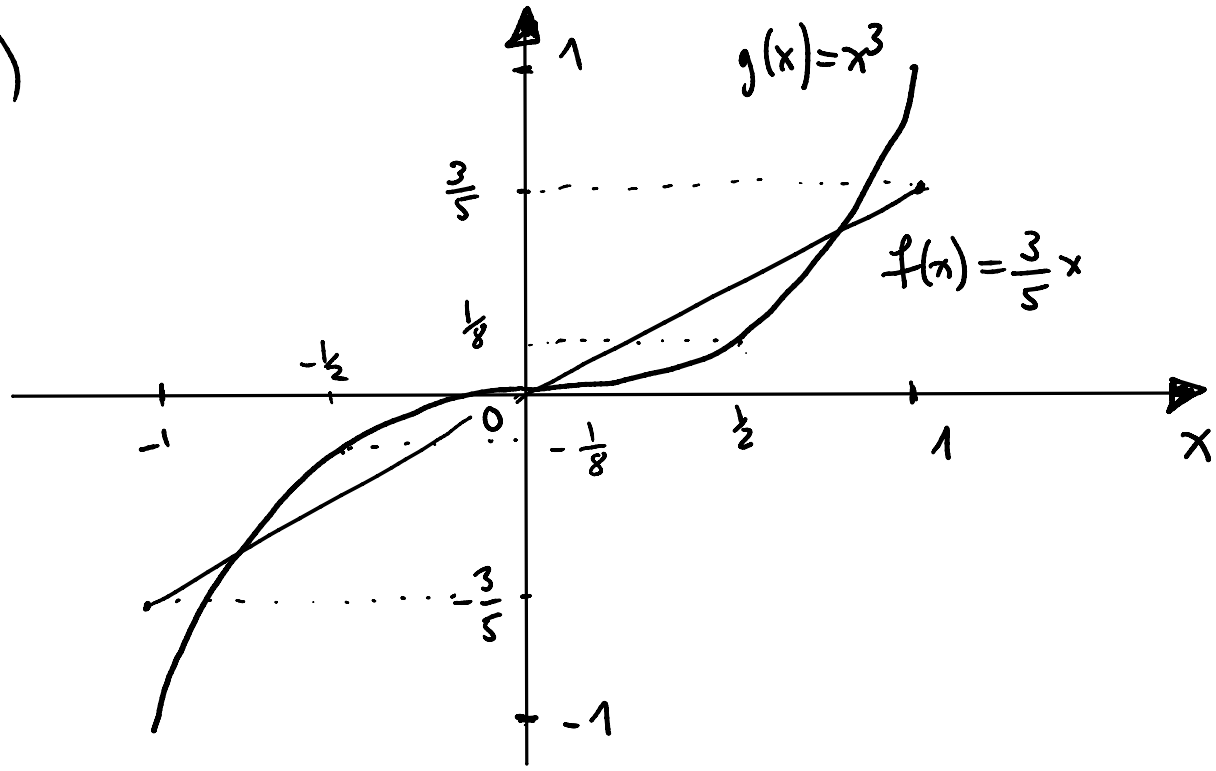
Thus: $$\frac{\partial L}{\partial w} = \int_{-1}^{1} (2wx^2 - 2x^4)\, dx$$

$$= \left. \frac{2wx^3}{3} - \frac{2x^5}{5} \right|_{-1}^{1} = \frac{4w}{3} - \frac{4}{5}$$

$$\frac{\partial L}{\partial w} = 0 \Rightarrow w = \frac{3}{5}$$

$$\frac{\partial L}{\partial w} = 0 \quad \Rightarrow \quad w = \frac{3}{5}$$

(c)



ECE/CS 559: Neural Networks **Fall 2024**

Homework 2

Due: **Tuesday** September 10, 2024 (by 9pm, on Gradescope)

Note: *You may discuss these problems in groups. However, you must write up your own solutions and mention the names of the people in your group. Also, please do mention any solution manuals, online material, or other sources you used in a major way.*

1. **[Designing a Feed-Forward NN]**

   Design a neural network that implements the following *logical* statement:

   $$(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_2 \wedge x_3).$$

   Use $+1$ to represent `True` and -1 to represent `False` (this is a bit of a change compared to HW1). Let your activation function be

   $$\varphi(v) = \mathsf{sign}(v) = \begin{cases} +1 & ; \quad v > 0, \\ 0 & ; \quad v = 0, \\ -1 & ; \quad v < 0. \end{cases}$$

   (a) Draw your chosen architecture with all the weights and biases explicitly, and specify what kind of feed forward neural network it is, e.g., 3-5-2-1 is a 3-layer network with 5 neurons in its first layer, 2 in its second, and 1 in its output later.

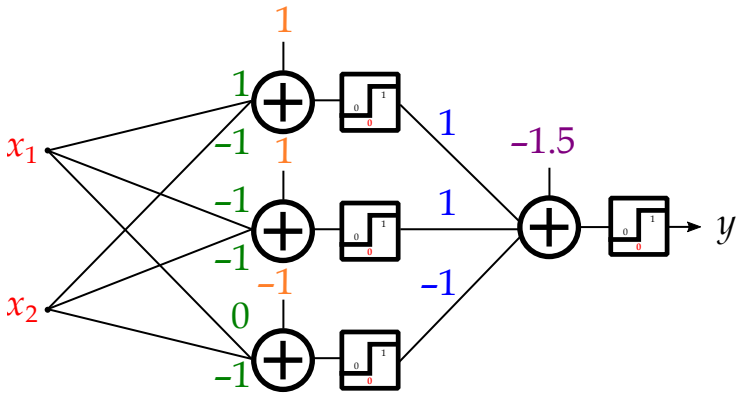
   (b) Write down the input-to-output *analytic* equation that your network represents.

   (c) Write code that produces the truth table (filled with `True/False`) of the logical statement and the input-output table (filled with $+1/-1$) of the analytic equation, and verify that they match. (In your main submission, include the outputted tables. In the code submission, include your code.)

2. **[Decision Boundary of a NN]**

   Consider the following feed-forward neural network that uses the step activation function:

   $$\varphi(v) = \mathsf{step}(v) = \begin{cases} 1 & ; \quad v \geq 0, \\ 0 & ; \quad v < 0. \end{cases}$$



(a) Name and list the parameter matrices and vectors with their dimensions.

(b) Write down the input-to-output *analytic* equation $y = f(x)$ that this network represents, in linear algebraic form.

(c) Write a Python program that does the following:

- Samples 1,000 random points x from a uniform distribution over the square $[-2, 2]^2$.
- Calculates the output of the neural network $y = f(x)$ for each point.
- Plots a scatter plot of all points x, where the points are blue when $y = 0$ and the points are red when $y = 1$.

Submit your code and report its output.

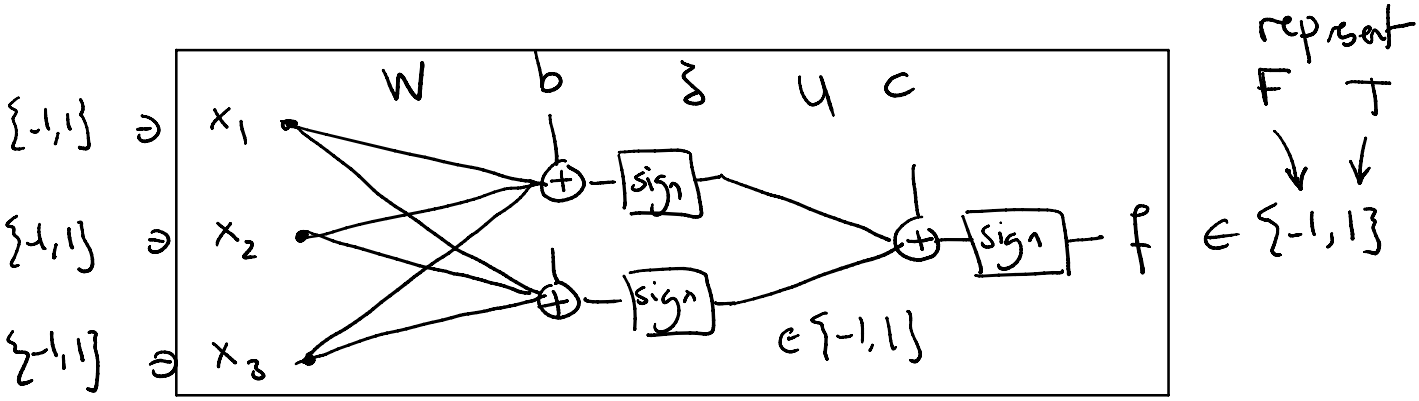
(d) Draw a rough sketch of the *decision boundary* of this neural network, i.e., the curve that separates the $y = 0$ and $y = 1$ regions.

(In your main submission, include the outputted plot and your sketch. In the code submission, include your code.)

Question 1

a) To make a network implement $(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_2 \wedge x_3)$, we need 2 layers, one for the ands and one for the or.

Since there are 2 and clauses, we can use a **3-2-1** NN:



- You can use the formulas we saw in lecture, but let's redo!
- To implement $x_1 \wedge x_2 \wedge x_3$ we can threshold $x_1 + x_2 + x_3$. This sum can be -3 -1 1 3

all -1 and is F here — one 1 — two 1s — all three 1s and is T here

threshold

We want thus to threshold between 1 and 3, say 2.

So, $x_1 + x_2 + x_3 > 2$ represents $x_1 \wedge x_2 \wedge x_3$.

- Since x_3 is negated for us, we just flip its sign:

$$x_1 + x_2 - x_3 > 2 \iff 1 \cdot x_1 + 1 \cdot x_2 - 1 \cdot x_3 - 2 > 0$$

$$x_1 + x_2 - x_3 > 2 \iff 1\cdot x_1 + 1\cdot x_2 - 1\cdot x_3 - 2 > 0$$

(w_{11}, w_{12}, w_{13}, b_1)

This way, z_1 will be the ±1 T/F representation of the clause $(x_1 \wedge x_2 \wedge \neg x_3)$.

- For the $(\neg x_2 \wedge x_3)$ clause, we eliminate x_1 by setting its weight to 0. To represent an and clause with two terms, $x_2 \wedge x_3$, look at the sum $x_2 + x_3$, can be $-2, 0, 2$
  (both −1 and F; ±1 threshold; both +1 and T)

  We need to threshold between 0 & 2, say 1.

  So, $x_2 + x_3 > 1$ represents $x_2 \wedge x_3$.

Since x_2 is negated, do: $-x_2 + x_3 > 1 \iff 0\cdot x_1 - 1\cdot x_2 + 1\cdot x_3 - 1 > 0$

(w_{21}, w_{22}, w_{23}, b_2)

- Finally, in the second layer, we need to implement $z_1 \vee z_2$, the sum $z_1 + z_2$ can be $-2, 0, 2$
  (both −1 or F; ±1 threshold; both 1 or T)

  We need to threshold between −2 & 0, say −1.

  So, $z_1 \vee z_2$ is represented by $z_1 + z_2 > -1 \iff +1\cdot z_1 + 1\cdot z_2 + 1 > 0$

(u_1, u_2, c)

- Thus: $\boxed{W = \begin{bmatrix} 1 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix} \quad b = \begin{bmatrix} -2 \\ -1 \end{bmatrix}}$

- hw: $W = \begin{bmatrix} 0 & -1 & 1 \end{bmatrix}$ $b = \begin{bmatrix} -1 \end{bmatrix}$

  $u = \begin{bmatrix} 1 & 1 \end{bmatrix}$ $c = 1$

b) The analytic expression is thus:

$$f(x) = \text{sign}\left(u \cdot \text{sign}(Wx + b) + c \right) \quad \text{or}$$

$$f(x) = \text{sign}\left(\text{sign}(x_1 + x_2 - x_1 - 2) + \text{sign}(-x_2 + x_3) + 1 \right)$$

c) Here's pseudo code that will generate the tables

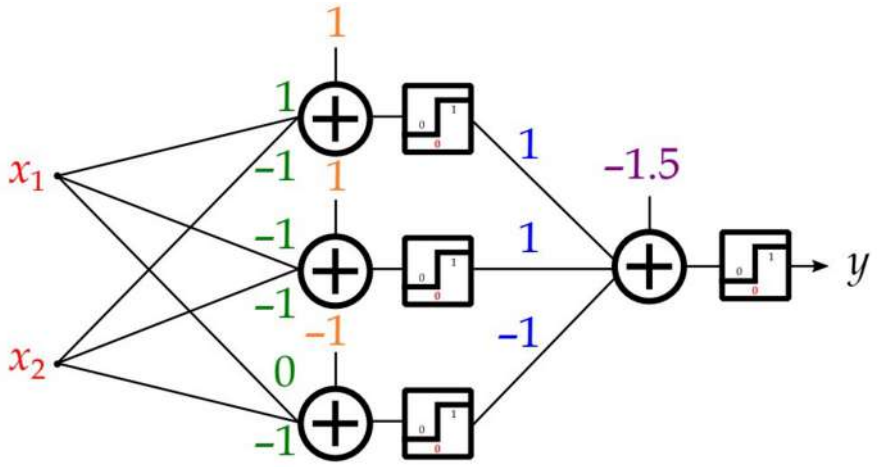
```
For x1 in {true, false}:
    For x2 in {true, false}:
        For x3 in {true, false}:
            y = (x1 and x2 and not x3) or (not x2 and x3)
            print(x1, x2, x3, y)
```

```
For i in range(8)                    # use binary digits
    x1 = 2*(i/4) % 2 - 1             # most significant
    x2 = 2*(i/2) % 2 - 1
    x3 = 2*(i/1) % 2 - 1             # least significant
    # The advantage of this is: no long nested for loops!
    y = sign( sign(x1 + x2 - x1 - 2) + sign(-x2 + x3) + 1 )
    print(x1, x2, x3, y)
```

print (x_1, x_2, x_3, y)

Question 2

a)



$W \in \mathbb{R}^{3\times 2}$ $b \in \mathbb{R}^3$ $u \in \mathbb{R}^{1\times 3}$ $c \in \mathbb{R}^1$

$$W = \begin{bmatrix} 1 & -1 \\ -1 & -1 \\ 0 & -1 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \quad u = [1\ 1\ -1] \quad c = -1.5$$

b) $y = \text{step}(u\,\text{step}(Wx+b) + c)$

← expected answer (linear algebraic) or this is okay

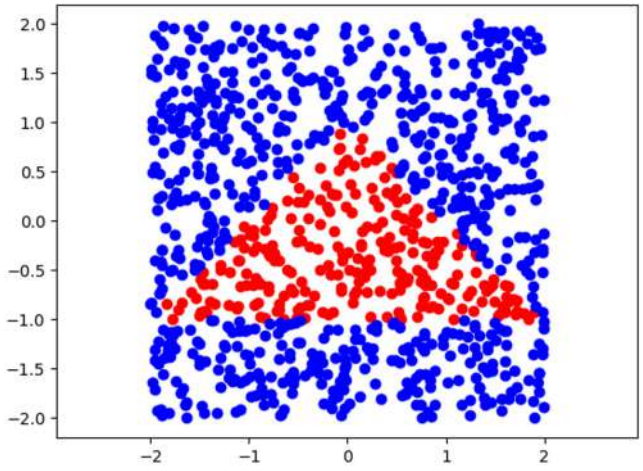
$$y = \text{step}(\text{step}(x_1 - x_2 + 1) + \text{step}(-x_1 - x_2 + 1) + \text{step}(-x_2 - 1) - 1.5)$$

c) Here's code to simulate the NN:

```
import numpy as np
import matplotlib.pyplot as plt
x1 = np.random.uniform(-2,2,1000)
x2 = np.random.uniform(-2,2,1000)
z1 = (x1-x2+1>=0)*1.
z2 = (-x1-x2+1>=0)*1.
```

```
z3 = (-x2-1>=0)*1.
y = np.heaviside(z1+z2-z3-1.5,1)
plt.scatter(x1[np.where(y==1)],x2[np.where(y==1)],c='red')
plt.scatter(x1[np.where(y==0)],x2[np.where(y==0)],c='blue')
plt.axis('equal')
plt.show()
```

0 1 | 2 3



1 + 1 + 1

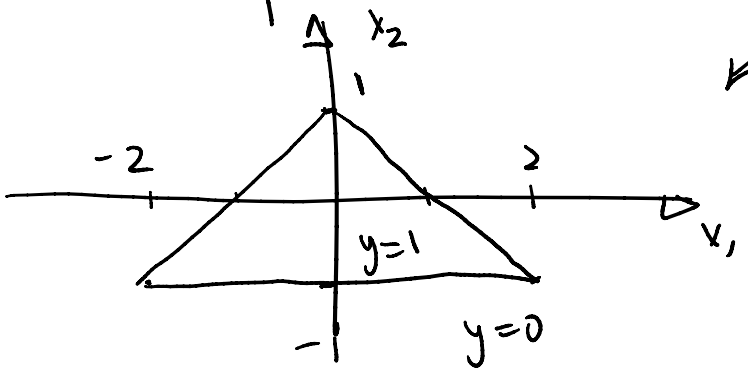
0 1 2 3

$x_2 < x_1 - 1$

$x_2 < -x_1 + 1$

and $x_2 < -1$

d) The boundary looks like a triangle. So, it's a reasonable guess to assume that's what the network represents:



acceptable answer

nerdy answer :)

- Is this all? What if we look outside of this range?

The second layer equation is $z_1 + z_2 + (1 - z_3) > 2.5$

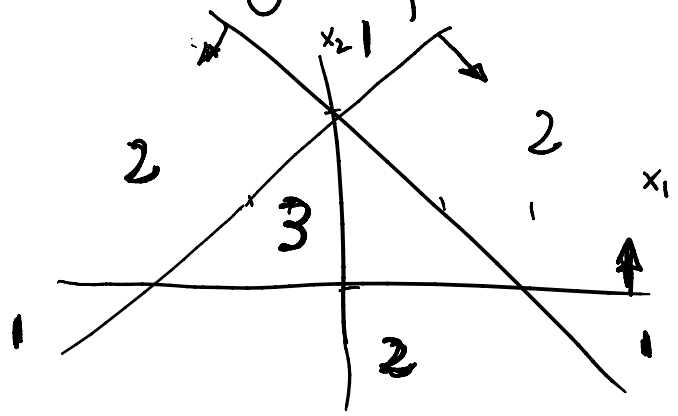
But $z_1 + z_2 + z_3$ can be 0, 1, 2, 3 ...

But $z_1 + z_2 + z_3$ can be 0, 1, 2, 3
(all 0, one 1, two 1s, all three 1s)

↑ threshold

Thus, the output is $y=1$ when <u>all three</u> of the half-spaces $x_2 < x_1 - 1$, $x_2 < -x_1 + 1$, and $x_2 < -1$ (negation of z_3) are active:

These define 7 regions, with active numbers:



So, we indeed have only the triangle giving $y=1$

ECE/CS 559: Neural Networks **Fall 2024**

Homework 5

Due: **Tuesday** October 1, 2024 (by 9pm, on Gradescope)

Note: *You may discuss these problems in groups. However, you must write up your own solutions and mention the names of the people in your group. Also, please do mention any solution manuals, online material, or other sources you used in a major way.*

Submit your code as a **two** `.py` **files**, one for each question, in Gradescope. Make sure that it **runs properly** and generates all the plots that you report on in the main submission.

1. **[Gradient Descent]**

   Let $\mathrm{w} \in \mathbb{R}^2$. Consider the following function:

   $$R(\mathrm{w}) = 13w_1^2 - 10w_1w_2 + 4w_1 + 2w_2^2 - 2w_2 + 1$$

   (a) Find the gradient of R and solve for the optimality condition exactly.

   (b) Implement gradient descent. At each iteration, calculate and save the distance between the iterate and the optimal solution. Run this with $\eta = 0.02, 0.05$, and 0.1 and 500 iterations each. Plot distance vs. iteration in each case. Report it.

   (c) Is even larger η beneficial? Why or why not?

2. **[Backpropagation]**

   Revisit the dataset you generated in HW2 Q2. Imagine that a friend gives you the $1,000$ points generated with that neural network, tells you what the architecture looks like (but not the weights), then asks you to train a neural network that imitates theirs.

   (a) You want to use gradient descent, but know that you can't use a step function to do the training. Choose instead to use a sigmoid function φ with parameter $a = 5$. Write down φ and φ' explicitly and implement Python functions for each, that operate on `numpy` arrays.

   (b) Call the first layer weights W and its biases b, the second layer weights U and its bias c, and the output f. Let the hidden layer output be z. Write down the dimensions of all weights, biases, and variables, then write down the forward equations from x to v_z, from v_z to z, from z to v_f, and from v_f to f. Translate these into Python code.

(c) Use the squared loss $\ell(y, f) = (f - y)^2$. Since we are targeting binary outputs and φ limits values to $[0, 1]$, this is not that different from 0-1 loss. Start writing down the backward equations for a single data point, by first writing the expression for $\nabla_f \ell$ followed by the expression for δ_f. Then, write the backward equation from δ_f to δ_z, then from δ_z to δ_x. Use these along with x and z to compute the gradients $\nabla_W \ell$, $\nabla_b \ell$, $\nabla_U \ell$, and $\nabla_c \ell$. (Remember that these gradients should have the same dimensions as the respective weights and biases.) Translate these into Python code.

(d) In Python, initialize all your weights and biases with i.i.d. Gaussian$(\mu = 0, \sigma = 0.1)$ samples. Then, create an outer `epoch` loop and inner `i` loop ranging over the $1,000$ data points. For each data point (x, y), run the forward pass code from (b), then the backward pass code from (c). Then, perform gradient descent with that single point (this is a form of *stochastic gradient descent, SGD*): $W \leftarrow W - \eta \nabla_W \ell$, etc. Use $\eta = 0.01$ and perform 100 epochs. At the end of each epoch, calculate and save the MSE. At the end, plot and report the MSE vs. epoch curve.

(e) In HW2 Q2, you visualized the decision boundary of your friend's neural network. Visualize the decision boundary of yours, by plotting a 3D scatter plot of $(x_1, x_2, f(x_1, x_2))$ using the weights at the end of your training. Report the plot. (You can also visualize this during the training, to see how the boundary evolves.) You may find the following snippet useful.

```
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, y_predicted, color = "green")
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```

(f) Try at least three of the following hacks individually or simultaneously. Say what you tried, then report on the MSE vs. epoch curve, the decision boundary, a description of what changed, and your hypothesis as to why.

- Change η.
- Start η high but multiply by 0.9 if the MSE increases (or if it increases for T epochs, where you choose T).
- Choose a different a in φ.
- Change the number of epochs that you perform.
- Do a proper gradient descent (accumulate gradients, update only in the end of an epoch, zero-out gradients, and move to next epoch).
- Use minibatches (accumulate B gradients, update, zero-out, and continue).
- Change the architecture by including more or fewer hidden neurons.
- Change how you initialize the weights and biases, e.g. by setting them to 0 or choosing σ very small or large.

Question 1

(a) $R(w) = 13w_1^2 - 10w_1w_2 + 4w_1 + 2w_2^2 - 2w_2 + 1$

$$\frac{\partial R}{\partial w_1} = 26w_1 - 10w_2 + 4 = 0 \quad (1)$$

$$\frac{\partial R}{\partial w_2} = -10w_1 + 4w_2 - 2 = 0 \quad (2)$$

(optimality)

$$(2) \Rightarrow -25w_1 + 10w_2 - 5 = 0 \quad (3)$$

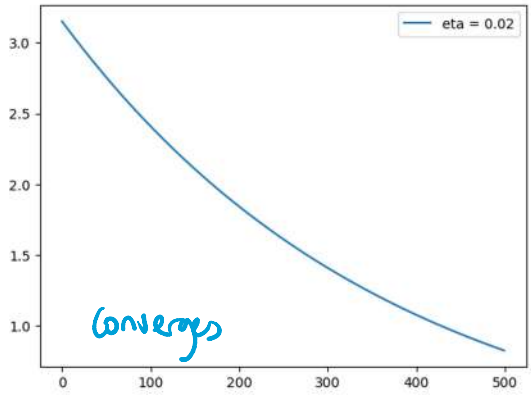
$$(1) + (3) \Rightarrow w_1 - 1 = 0 \Rightarrow \boxed{w_1 = 1} \quad (4)$$

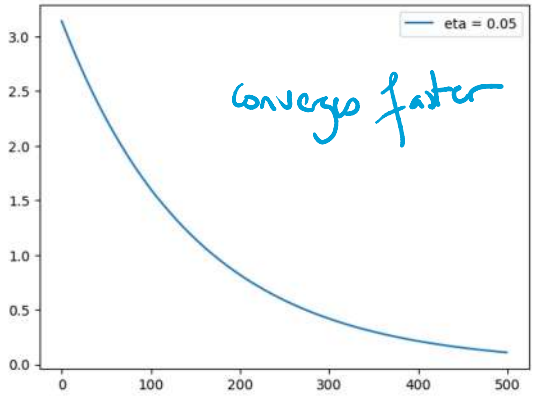
$$(2) + (4) \Rightarrow -10 + 4w_2 - 2 = 0 \Rightarrow \boxed{w_2 = 3}$$

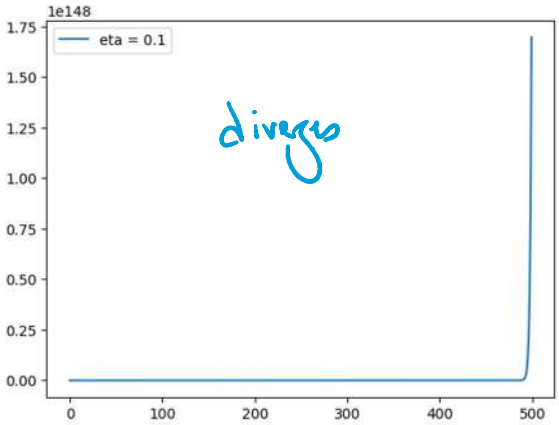
(b)

```
import numpy as np
import matplotlib.pyplot as plt
w_opt=[1,3]
number_of_epochs = 500
for eta in [0.02, 0.05, 0.1]:
  dist=np.zeros(number_of_epochs)
  w=np.zeros((2,))
  for i in range(number_of_epochs):
    grad = np.array([4+26*w[0]-10*w[1],-2-10*w[0]+4*w[1]])
    w = w - eta*grad
    dist[i]=np.linalg.norm(w-w_opt)
```

```
plt.plot(dist,label="eta = "+ str(eta))
plt.legend()
plt.show()
```







(c) Larger η is not always beneficial, because it would lead to instability (even if the gradients are exact, like here). This is because the 1st-order Taylor approximation is no longer very accurate.

Question 2

(a) $\varphi(v) = \frac{1}{1+e^{-5v}}$

$\varphi'(v) = 5\varphi(v)(1-\varphi(v))$

(b)

W	is	3×2	$(\dim(z) \times \dim(x))$
b	is	3	$(\dim(z))$
U	is	1×3	$(\dim(f) \times \dim(z))$
c	is	1	$(\dim(f))$

Forward equations!

$v_z = Wx + b \qquad z = \varphi(v_z)$

$v_f = Uz + c \qquad f = \varphi(v_f)$

```
v_z = np.matmul(W,x)+b
z = phi(v_z)
v_f = np.matmul(U,z)+c
f = phi(v_f)
```

(c) Backward equations for $\ell(y,f) = (y-f)^2$

$\nabla_f \ell = \frac{\partial \ell}{\partial f} = -2(y-f) \qquad \delta_f = \frac{\partial f}{\partial \ell} \cdot \varphi'(v_f)$

$$\nabla_f \ell = \frac{\partial \ell}{\partial f} = -2(y-f) \qquad \delta_f = \frac{\partial f}{\partial e} \cdot \varphi'(v_f)$$

$$\nabla_z \ell = U^T \delta_f \qquad \delta_z = \nabla_z \ell \cdot \varphi'(v_z)$$

$$\nabla_W \ell = \delta_z x^T \qquad \nabla_b = \delta_z$$

$$\nabla_U \ell = \delta_f z^T \qquad \nabla_c = \delta_f$$

```
dloss_df = -2*(y-f)
delta_f = dloss_df * phi_prime(v_f)
dloss_dz = np.matmul(np.transpose(U),delta_f)
delta_z = dloss_dz * phi_prime(v_z)
grad_W = np.matmul(delta_z, np.transpose(x))
grad_b = delta_z * 1
grad_U = np.matmul(delta_f, np.transpose(z))
grad_c = delta_f * 1
```

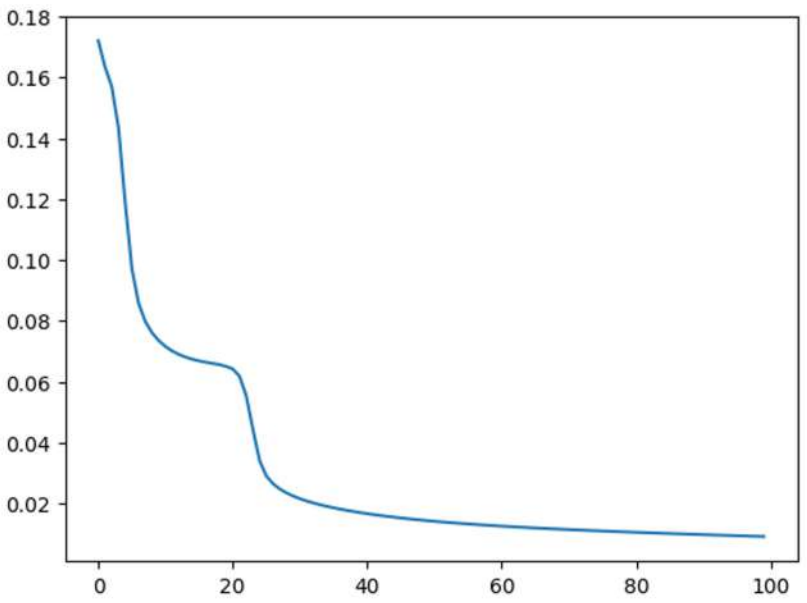
(d) Initialization:

```
sigma=0.1
W = np.random.randn(k,X.shape[0])*sigma
b = np.random.randn(k,1)*sigma
U = np.random.randn(1,k)*sigma
c = np.random.randn(1,1)*sigma
```

SGD:

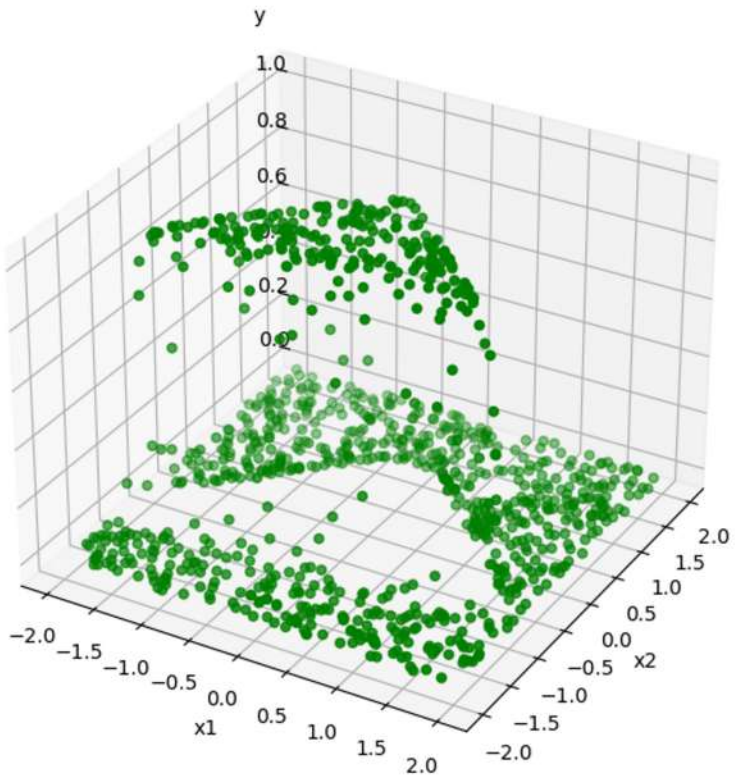
```
eta = 0.01
epochs = 100
risk = np.zeros(epochs)
```

```
for epoch in range(epochs):
  for i in range(N):
    x = X[:,i].reshape(x.shape)
    y = Y[0,i].reshape(f.shape)
    # Forward - Question 2(b)
    v_z = np.matmul(W,x)+b
    z = phi(v_z)
    v_f = np.matmul(U,z)+c
    f = phi(v_f)
    # Backward - Question 2(c)
    dloss_df = -2*(y-f)
    delta_f = dloss_df * phi_prime(v_f)
    dloss_dz = np.matmul(np.transpose(U),delta_f)
    delta_z = dloss_dz * phi_prime(v_z)
    grad_W = np.matmul(delta_z, np.transpose(x))
    grad_b = delta_z * 1
    grad_U = np.matmul(delta_f, np.transpose(z))
    grad_c = delta_f * 1
    # SGD
    W = W - eta * grad_W
    b = b - eta * grad_b
    U = U - eta * grad_U
    c = c - eta * grad_c
  Y_predicted = phi(np.matmul(U,phi(np.matmul(W,X)+b))+c)
  risk[epoch] = np.sum((Y-Y_predicted)**2)/N
plt.plot(risk)
plt.show()
```



(e) Visualization

```
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, Y_predicted, color = "green")
ax.set_xlabel('x1'); ax.set_ylabel('x2'); ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```



(e) This question is a bit open-ended.

- Changing η has its usual pitfalls. An adaptive η works remarkably well (many methods like Adam and Adagrad do this by monitoring the

Adam and Adagrad do this by monitoring the gradients themselves.)

- Minibatch is helpful to smooth out the gradients of SGD. However, we shouldn't simply accumulate the gradients, since then ther variance increases. Instead, we need to average across the minibatch.

- Initializing weights at 0 slows the convergence considerably, even halting it (by getting stuck in local minima.) Local minima are also an issue with very large σ for the random initialization.

- Making a big in φ makes it "less differentiable" do more unstable, because of both large (at the transition) and very small (at saturation) gradients. However,

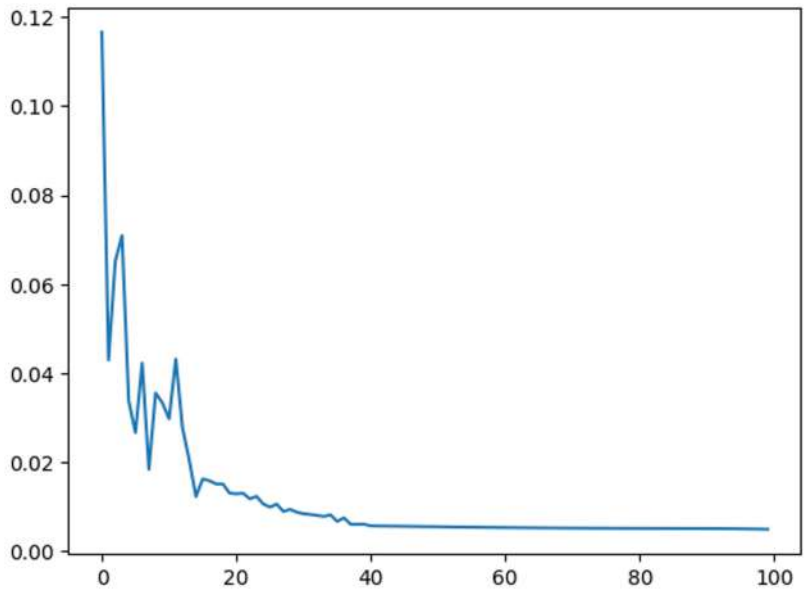
small (at saturation) gradients. However, the end results are crisper, as long as we can stabilize (adaptive η + minibatch help.)

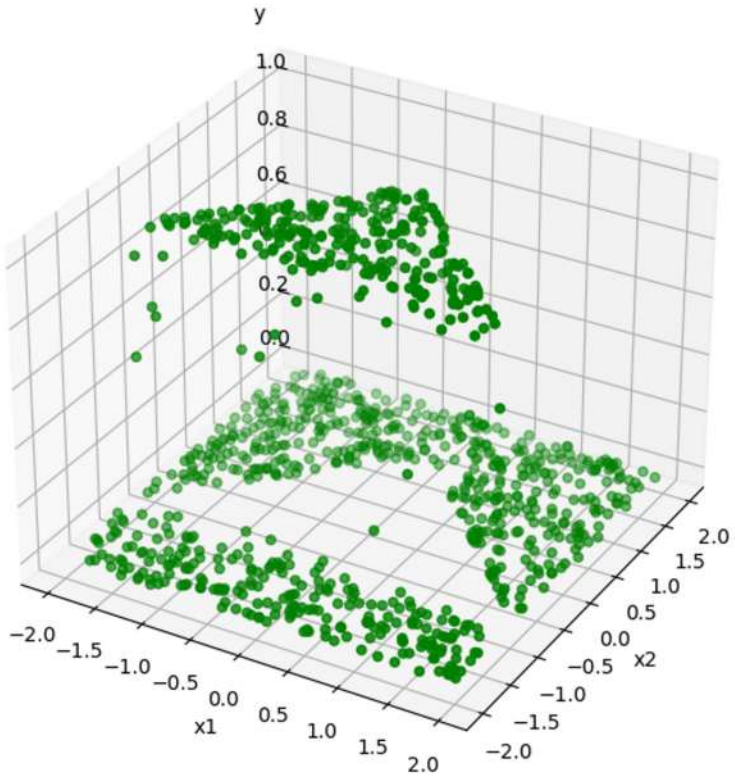
The following code does:

- Sets $\alpha = 10$ in φ.
- Adaptive η, starting at $\eta = 0.1$
- Minibatch with batch size 5, and with averaged gradients.

```
def phi(v):
  return 1./(1.+np.exp(-10*v))
def phi_prime(v):
  return 10*phi(v)*(1-phi(v))
sigma=0.1
# Initialization
W = np.random.randn(k,X.shape[0])*sigma
b = np.random.randn(k,1)*sigma
U = np.random.randn(1,k)*sigma
c = np.random.randn(1,1)*sigma
# W=W*0 # Zero initialization
# b=b*0
# U=U*0
# c=c*0
eta = 0.1
epochs = 100
batch_size = 5
risk = np.zeros(epochs)
grad_W = 0*W
grad_b = 0*b
grad_U = 0*U
grad_c = 0*c
for epoch in range(epochs):
  for i in range(N):
    x = X[:,i].reshape(x.shape)
    y = Y[0,i].reshape(f.shape)
    # Forward
    v_z = np.matmul(W,x)+b
    z = phi(v_z)
```

```
    v_f = np.matmul(U,z)+c
    f = phi(v_f)
    # Backward
    dloss_df = -2*(y-f)
    delta_f = dloss_df * phi_prime(v_f)
    dloss_dz = np.matmul(np.transpose(U),delta_f)
    delta_z = dloss_dz * phi_prime(v_z)
    grad_W += np.matmul(delta_z, np.transpose(x))
    grad_b += delta_z * 1
    grad_U += np.matmul(delta_f, np.transpose(z))
    grad_c += delta_f * 1
    # Minibatch SGD
    if (i+1)%batch_size==0:
      W = W - eta * grad_W/batch_size # Averaged
      b = b - eta * grad_b/batch_size # gradients
      U = U - eta * grad_U/batch_size
      c = c - eta * grad_c/batch_size
      grad_W = 0*W # Zeroing out the gradients
      grad_b = 0*b
      grad_U = 0*U
      grad_c = 0*c
  Y_predicted = phi(np.matmul(U,phi(np.matmul(W,X)+b))+c)
  risk[epoch] = np.sum((Y-Y_predicted)**2)/N
  if epoch > 1:
    if risk[epoch]>risk[epoch-1]:
      eta = eta*0.9
plt.plot(risk)
plt.show()
# Visualization
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, Y_predicted, color = "green")
ax.set_xlabel('x1'); ax.set_ylabel('x2'); ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```





Note the faster convergence and the crisper output. The convergence is noisier due to the large step sizes, but it's controlled thanks to the adaptation of η and the averaging inside the mini-batch.

(set batch_size=20 to see it get smoother but a bit slower.)

ECE/CS 559 - Fall 2016 - Midterm and Solutions.

Erdem Koyuncu

- **Q1 (24 pts):** In the following, we follow the convention that $x_0 = 1$.

  Recall that the step-activation function $u : \mathbb{R} \to \{0, 1\}$ is defined as $u(v) = 1$ if $v \geq 0$, and $u(v) = 0$, otherwise. Then, for n inputs $x_1, \ldots, x_n$, a **perceptron** can be defined via the input-output relationship

  $$y' = u\left(\sum_{i=0}^{n} w_i x_i\right) = u(w_0 + w_1 x_1 + \cdots + w_n x_n),$$

  where y' is the perceptron output, $w_1, \ldots, w_n$ are the synaptic weights, and w_0 is the bias term.

  We define a new type of neuron, namely, a **sauron**, via the input-output relationship

  $$y = u\left(\prod_{i=0}^{n} (w_i + x_i)\right) = u((w_0 + 1)(w_1 + x_1) \cdots (w_n + x_n)),$$

  where y is called the sauron output.

  Let the real number 1 represent a `TRUE`, and the real number 0 represent a `FALSE`.

  (a) **(8 pts):** Let $n = 1$. Does there exist w_0, w_1 such that $y = 1 - x_1$ for $x_1 \in \{0, 1\}$? In other words, can a single sauron implement the `NOT` gate? If your answer is "Yes," find specific w_0, w_1 such that the sauron implements the `NOT` gate. If your answer is "No," prove that no choice for w_0, w_1 can result in a sauron that implements the `NOT` gate.

  (b) **(8 pts):** Let $n = 2$. Does there exist w_0, w_1, w_2 such that $y = x_1 x_2$ for $x_1, x_2 \in \{0, 1\}$? In other words, can a single sauron implement the `AND` gate? Justify your answer as in (a).

  (c) **(8 pts):** Let $n = 2$. Does there exist w_0, w_1, w_2 such that $y = ((x_1 + x_2) \mod 2)$ for $x_1, x_2 \in \{0, 1\}$? In other words, can a single sauron implement the `XOR` gate? Justify your answer as in (a).

Solution:

(a) Yes. We need $u((1 + w_0)(w_1 + x_1)) = 1 - x_1$ for $x_1 \in \{0, 1\}$. In other words, we need $(1 + w_0)w_1 \geq 0$ and $(1 + w_0)(1 + w_1) < 0$. One solution is $w_0 = -2$ and $w_1 = -\frac{1}{2}$.

(b) No. The following inequalities need to be satisfied:

$$\begin{aligned} (1 + w_0)w_1 w_2 &< 0, \\ (1 + w_0)(1 + w_1)w_2 &< 0, \\ (1 + w_0)w_1(1 + w_2) &< 0, \\ (1 + w_0)(1 + w_1)(1 + w_2) &\geq 0. \end{aligned}$$

Dividing the third inequality by the first inequality, we obtain $\frac{1+w_2}{w_2} > 0$. Diving the fourth inequality by the second inequality, $\frac{1+w_2}{w_2} \leq 0$, a contradiction.

(c) Yes. We need

$$\begin{aligned} (1 + w_0)w_1 w_2 &< 0, \\ (1 + w_0)(1 + w_1)w_2 &\geq 0, \\ (1 + w_0)w_1(1 + w_2) &\geq 0, \\ (1 + w_0)(1 + w_1)(1 + w_2) &< 0. \end{aligned}$$

By inspection, $w_0 = -2$, $w_1 = w_2 = -\frac{1}{2}$ will work.

- **Q2 (26 pts):** In the following, consider only neurons with the step-activation function $u(\cdot)$.

  (a) **(13 pts):** Let $\mathcal{C}_0 = \{\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right], \left[\begin{smallmatrix}-1\\1\end{smallmatrix}\right], \left[\begin{smallmatrix}0\\4\end{smallmatrix}\right]\}$ and $\mathcal{C}_1 = \{\left[\begin{smallmatrix}0\\2\end{smallmatrix}\right], \left[\begin{smallmatrix}0\\-1\end{smallmatrix}\right], \left[\begin{smallmatrix}0\\-2\end{smallmatrix}\right]\}$ as illustrated in Figure (i). Members of classes $\mathcal{C}_0$ and $\mathcal{C}_1$ are represented by black disks and crosses, respectively.

  [I] **(7 pts):** We wish to design a perceptron $y = u(w_0 + w_1x_1 + w_2x_2)$ such that $y = 0$ if $\left[\begin{smallmatrix}x_1\\x_2\end{smallmatrix}\right] \in \mathcal{C}_0$ and $y = 1$ if $\left[\begin{smallmatrix}x_1\\x_2\end{smallmatrix}\right] \in \mathcal{C}_1$. Suppose that we use the perceptron training algorithm for this purpose with initial weights $w_0 = 1$, $w_1 = 0$, $w_2 = 1$ and learning rate $\eta = 1$. Either prove that the algorithm will converge, or prove that it will not converge. You may use the perceptron convergence theorem.

  [II] **(6 pts):** Design a neural network (single-layer or multi-layer) such that the network provides an output of 0 if $\left[\begin{smallmatrix}x_1\\x_2\end{smallmatrix}\right] \in \mathcal{C}_0$ and an output of 1 if $\left[\begin{smallmatrix}x_1\\x_2\end{smallmatrix}\right] \in \mathcal{C}_1$.

  (b) **(13 pts)** Repeat (a) for classes $\mathcal{C}_0 = \{\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right], \left[\begin{smallmatrix}-1\\1\end{smallmatrix}\right], \left[\begin{smallmatrix}0\\4\end{smallmatrix}\right]\}$ and $\mathcal{C}_1 = \{\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right], \left[\begin{smallmatrix}0\\-1\end{smallmatrix}\right], \left[\begin{smallmatrix}0\\-2\end{smallmatrix}\right]\}$ illustrated in Figure (ii). Note that the only difference is that now, instead of the point $\left[\begin{smallmatrix}0\\2\end{smallmatrix}\right]$, we have the point $\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$ in class $\mathcal{C}_1$.

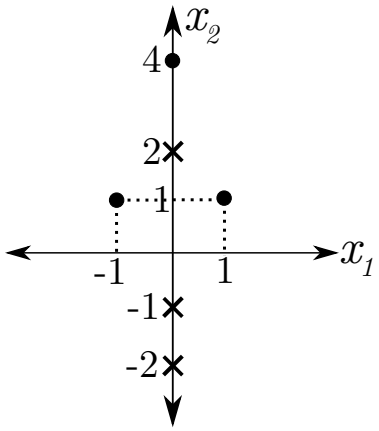


Figure (i)

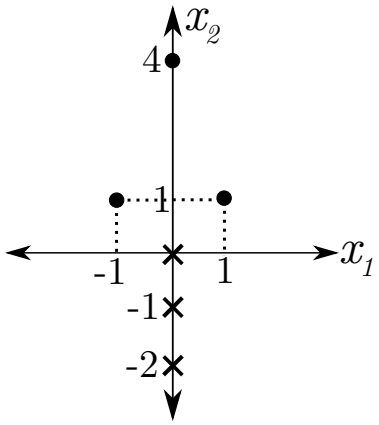


Figure (ii)

Solution:

(a) [I] PTA will not converge as the patterns are not linearly separable (the weights will always be updated).

[II] Let $y_1 = u(-x_2 - 3x_1 + 3)$ and $y_2 = u(-x_2 + 3x_1 + 3)$. The network $u(-\frac{3}{2} + y_1 + y_2)$ will work.

(b) [I] PTA will converge as the patterns are linearly separable.

[II] The same network in (a)-[II] will work! Or, a simpler network is $u(-x_2 + \frac{1}{2})$.

In the following, $\log(\cdot)$ is the natural logarithm, e is the base of the natural logarithm, $|\cdot|$ is the absolute value ($|x| = x$ if $x \geq 0$, and $|x| = -x$ if $x < 0$), and $\mathbb{R}$ is the set of real numbers. Recall $\frac{\partial e^x}{\partial x} = e^x$, and $e^{\log x} = x$, $x > 0$.

- **Q3 (25 pts):** Consider the activation function

$$f(v) = \begin{cases} 1 - e^{-v}, & v \geq 0, \\ -1 + e^{-|v|}, & v < 0. \end{cases}$$

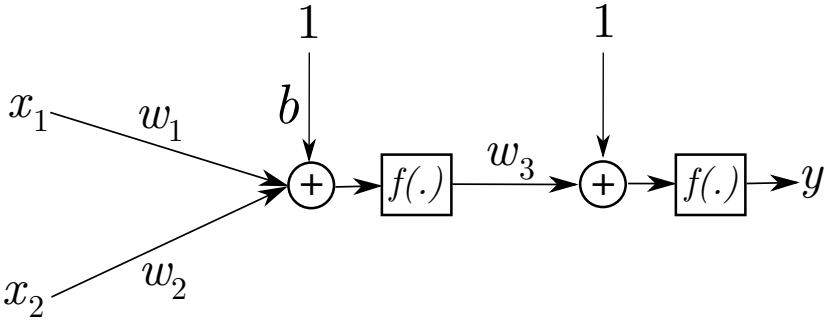
For (a)-(c), consider a single-neuron network with input-output relationship $y = f(b + \mathbf{w}^T\mathbf{x})$, where y is the network output, b is the bias term, $\mathbf{w} = \left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix}\right]$ are the synaptic weights, and $\mathbf{x} = \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right]$ is the network input.

(a) **(3 pts):** Calculate the derivative $f'(v)$ in closed form.

(b) **(7 pts):** Let $E = (d - y)^2$, where d is a constant (a generic desired output). Find the delta-learning rule (the gradient-descent update equations) for b, w_1, w_2 given learning parameter $\eta = \frac{1}{2}$. Remember that you can write a single (vectorized) update equation that can handle all variables. The final update expression(s) may however contain only the terms/functions $d, y, f', f, \mathbf{w}, w_1, w_2, b$.

(c) **(6 pts):** Consider the same delta-learning setup as in (b). Consider the training vectors $\mathbf{x}_1 = \left[\begin{smallmatrix} \log 2 \\ \log 3 \end{smallmatrix}\right]$, $\mathbf{x}_2 = \left[\begin{smallmatrix} 0 \\ \log 2 \end{smallmatrix}\right]$, with desired outputs $d_1 = \frac{2}{3}$, $d_2 = \frac{5}{2}$, respectively. Find the updated bias and the updated synaptic weights after one epoch of online learning given initial conditions $b = 0$, $\mathbf{w} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$.

(d) **(9 pts)** Consider now a multi-layer network as shown below.



Let $E = e^{(d-y)^4}$. Find the gradient-descent update equations for b, w_1, w_2, w_3 given learning parameter $\eta = \frac{1}{4}$. The final expression may only contain the terms/functions $d, y, f', f, \mathbf{w}, w_1, w_2, b, w_3$.
Hint: You do not have to apply some algorithm. Just write the input-output relationship of the network, and use chain rule. This is more of a calculus problem than a neural network problem!

Solution:

(a) By calculus, $f'(v) = e^{-|v|}$.

(b) The same delta-rule in class: $\left[\begin{smallmatrix} b \\ w_1 \\ w_2 \end{smallmatrix}\right] \leftarrow \left[\begin{smallmatrix} b \\ w_1 \\ w_2 \end{smallmatrix}\right] + (d - y)f'(v)\left[\begin{smallmatrix} 1 \\ x_1 \\ x_2 \end{smallmatrix}\right]$, where $v = b + \mathbf{w}^T\mathbf{x}$.

(c) For initial bias/weights $\left[\begin{smallmatrix} 0 \\ 0 \\ 1 \end{smallmatrix}\right]$, if the input is $\mathbf{x}_1$, we obtain $v = \log 3$ so that $y = 1 - e^{-\log 3} = \frac{2}{3}$. Since $d_1 = \frac{2}{3}$, there will be no update. For the input x_2, with same bias and weights we have started with, we obtain $v = \log 2$, so that $y = \frac{1}{2}$, and $f'(v) = \frac{1}{2}$. This gives $\left[\begin{smallmatrix} 0 \\ 0 \\ 1 \end{smallmatrix}\right] + (\frac{5}{2} - \frac{1}{2})\frac{1}{2}\left[\begin{smallmatrix} 1 \\ 0 \\ \log 2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 1 \\ 0 \\ 1 + \log 2 \end{smallmatrix}\right]$ as the final weights after epoch 1.

(d) We have $y = f(1 + w_3 f(b + \mathbf{w}^T\mathbf{x}))$. By calculus/chain rule, we obtain

$$w_3 \leftarrow w_3 - \eta\frac{\partial e^{(d-y)^4}}{\partial w_3} = w_3 + (d - y)^3 e^{(d-y)^4} f'(1 + w_3 f(b + \mathbf{w}^T\mathbf{x}))f(b + \mathbf{w}^T\mathbf{x})$$

$$\left[\begin{smallmatrix} b \\ w_1 \\ w_2 \end{smallmatrix}\right] \leftarrow \left[\begin{smallmatrix} b \\ w_1 \\ w_2 \end{smallmatrix}\right] + (d - y)^3 e^{(d-y)^4} f'(1 + w_3 f(b + \mathbf{w}^T\mathbf{x}))w_3 f'(b + \mathbf{w}^T\mathbf{x})\left[\begin{smallmatrix} 1 \\ x_1 \\ x_2 \end{smallmatrix}\right]$$

- **Q4 (25 pts):** Design a (possibly multi-layer) neural network with two inputs $x_1, x_2 \in \mathbb{R}$ and a single output y such that $y = 1$ if $x_1 = 3$ and $x_2 = 2$, and $y = 0$, otherwise. In other words, the network output should assume a value of 1 only at the single point $(x_1, x_2) = (3, 2)$ of the input space $\mathbb{R}^2$. Note that the inputs can be any real numbers. The neuron(s) of the network should use the step-activation function $u : \mathbb{R} \to \{0, 1\}$ given by $u(v) = 1$ if $v \geq 0$, and $u(v) = 0$, otherwise. The network should consist of **5 neurons at most**. Spoilers and their partial credits:

  (a) **(3 pts):** Design a network with input x_1 and output y such that $y = 1$ if $x_1 \geq 3$, and $y = 0$, otherwise.

  (b) **(4 pts):** Design a network with input x_1 and output y such that $y = 1$ if $x_1 \leq 3$, and $y = 0$, otherwise.

  (c) **(6 pts):** Design a network with input x_1 and output y such that $y = 1$ if $x_1 = 3$, and $y = 0$, otherwise.

  (d) **(6 pts):** Solve the original problem without any restrictions on the number of neurons.

  (e) **(6 pts):** Solve the original problem.

  You are not required to do (a)-(d) provided you can provide a correct solution to (e).

Solution:

(a) $y_1 = u(x_1 - 3)$.

(b) $y_2 = u(-x_1 + 3)$.

(c) $u(y_1 + y_2 - \frac{3}{2})$, i.e. AND the two previous networks.

(d) and (e) We do (a) and (b) for $x_2 = 2$ as well, i.e. let $z_1 = u(x_2 - 2)$ and $z_2 = u(-x_2 + 2)$. Now, AND y_1, y_2, z_1, z_2, i.e. let $y = u(y_1 + y_2 + z_1 + z_2 - \frac{7}{2})$.

ECE/CS 559 - Fall 2017 - Midterm Answers.

Full Name: **ID Number:**

Q1 (25 pts). Let $f(x) = x^2$.

(a) **(3 pts)** Let $x^\star$ be the global minimizer of f, i.e. $x^\star = \min_{x\in\mathbb{R}} f(x)$, where $\mathbb{R}$ is the set of real numbers. Find $x^\star$.

Solution: $x^\star = 0$.

(b) **(5 pts)** Recall that the gradient descent equations are given by $x_{n+1} = x_n - \eta f'(x_n)$, $n \in \{0, 1, 2, \ldots\}$, where $\eta > 0$ is the learning parameter, and $f'(x)$ is the derivative of $f(x)$ with respect to x. Find x_n for any $n \geq 1$ given $x_0 = 2$ and $\eta = \frac{1}{4}$.

Solution: $f'(x) = 2x$, so that the update equation is $x_{n+1} = x_n - 2\eta x_n = x_n(1 - 2\eta)$. In particular, for $\eta = \frac{1}{4}$, we obtain $x_{n+1} = \frac{x_n}{2}$. For $x_0 = 2$, this yields $x_1 = 1$, $x_2 = \frac{1}{2}$, or, in general, $x_n = 2^{-n+1}$.

(c) **(11 pts)** Describe the $n \to \infty$ asymptotic behavior of x_n and $f(x_n)$ for every possible initial condition $x_0 \in \mathbb{R}$ and learning parameter $\eta > 0$. For example, your answer should be able to describe where x_n and $f(x_n)$ go as $n \to \infty$ given initial conditions $x_0 = -2444$ and $\eta = 120$ (or any other x_0 and η that will be given to you).

Solution: From (b), we have $x_{n+1} = (1 - 2\eta)^{n+1} x_0$. First, if $x_0 = 0$, we have $x_n = 0$, $\forall n$ regardless of how η is chosen. Otherwise, we can observe that if $|1 - 2\eta| < 1$ (or, equivalently, if $0 < \eta < 1$), x_n converges to 0 regardless of what x_0 is. If $1 - 2\eta = 1$ (or, equivalently, if $\eta = 0$), we have $x_n = x_0$, $\forall n$. If $1 - 2\eta = -1$ (or, equivalently, if $\eta = 1$), x_n will oscillate between x_0 and $-x_0$. If $1 - 2\eta < -1$ (or, equivalently, if $\eta > 1$), x_n will oscillate between $+\infty$ and $-\infty$ as $n \to \infty$. If $1 - 2\eta > 1$ (or, equivalently, if $\eta < 0$), x_n will diverge to either $+\infty$ or $-\infty$, depending on the sign of x_0. This is a complete solution for any $\eta \in \mathbb{R}$; it is OK if you only consider $\eta > 0$.

Not Covered

(d) **(6 pts)** Let $k(y, z) = y^2 + z^4$. Let $g(y, z) \triangleq \begin{bmatrix} \frac{\partial k}{\partial y} \\ \frac{\partial k}{\partial z} \end{bmatrix}$ and $H(y, z) \triangleq \begin{bmatrix} \frac{\partial^2 k}{\partial y^2} & \frac{\partial^2 k}{\partial y \partial z} \\ \frac{\partial^2 k}{\partial z \partial y} & \frac{\partial^2 k}{\partial z^2} \end{bmatrix}$ denote the gradient and the Hessian of k, respectively. Recall that the update equations for Newton's method are given by $\begin{bmatrix} y_{n+1} \\ z_{n+1} \end{bmatrix} = \begin{bmatrix} y_n \\ z_n \end{bmatrix} - \eta \left(H(y_n, z_n)\right)^{-1} g(y_n, z_n)$, $n \in \{0, 1, 2, \ldots\}$. Given $y_0 = z_0 = \eta = 1$, calculate y_n, z_n for every $n \geq 1$.

Solution: By the definitions, we obtain $g(y, z) = \begin{bmatrix} 2y \\ 4z^3 \end{bmatrix}$ and $H(y, z) = \begin{bmatrix} 2 & 0 \\ 0 & 12z^2 \end{bmatrix}$ so that $(H(y, z))^{-1} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{12z^2} \end{bmatrix}$. For $\eta = 1$, this gives us the update equations $\begin{bmatrix} y_{n+1} \\ z_{n+1} \end{bmatrix} = \begin{bmatrix} y_n \\ z_n \end{bmatrix} - \begin{bmatrix} y_n \\ \frac{z_n}{3} \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{2}{3} z_n \end{bmatrix}$, $n \in \{0, 1, 2, \ldots\}$. Therefore, $y_n = 0$, and $z_n = (\frac{2}{3})^n$.

Q2 (25 pts). Consider a neuron with $n \geq 1$ inputs $x_1, \ldots, x_n$, and output $y = \theta(w_0 + w_1 x_1 + \cdots + w_n x_n)$, where $w_0, w_1, \ldots, w_n$ are the neuron bias and weights, and the activation function is given by $\theta(x) = 1$ if $x \in [0, 1]$, and $\theta(x) = 0$ if $x \notin [0, 1]$. Note that the activation function is different than the functions that we have encountered throughout the lectures.

(a) **(9 pts):** Let $n = 1$. Does there exist w_0, w_1 such that $y = 1 - x_1$ for $x_1 \in \{0, 1\}$? In other words, can a single neuron with activation function θ implement the `NOT` gate? If your answer is "Yes," find specific w_0, w_1 such that the neuron implements the `NOT` gate. If your answer is "No," prove that no choice for w_0, w_1 can result in a neuron that implements the `NOT` gate.

Solution: Yes. Geometrically, in the 2D plane, the activation function provides an output of 1 on a strip whose width and orientation you can adjust by adjusting the weights and the bias. With this observation, it is immediate that any gate of 2 inputs is, in fact, implementable. In particular, $w_0 = \frac{1}{2}$ and $w_1 = -1$ can implement the NOT gate. Note that these are the same weights that we have chosen in class for the step activation function.

(b) **(8 pts):** Let $n = 2$. Does there exist w_0, w_1, w_2 such that $y = x_1 x_2$ for $x_1, x_2 \in \{0, 1\}$? In other words, can a single neuron with activation function θ implement the `AND` gate? Justify your answer as in (a).
Solution: Yes. The choices $w_0 = -\frac{3}{2}$ and $w_1 = w_2 = 1$ can implement the AND gate. Note that these are the same weights that we have chosen in class for the step activation function.

(c) **(8 pts):** Let $n = 2$. Does there exist w_0, w_1, w_2 such that $y = ((x_1 + x_2) \mod 2)$ for $x_1, x_2 \in \{0, 1\}$? In other words, can a single neuron with activation function θ implement the `XOR` gate? Justify your answer as in (a).

Solution: Yes. One solution is $w_0 = -\frac{1}{2}$, $w_1 = w_2 = 1$.

Q3 (25 pts). Let u be the step activation function with $u(x) = 1$ if $x \geq 0$, and $u(x) = 0$, otherwise. Consider the perceptron $y = u(w_0 + w_1 x_1 + w_2 x_2)$, where w_1 and w_2 are the weights for inputs x_1 and x_2, respectively, w_0 is the perceptron bias, and y is the perceptron output. Let $\mathcal{C}_0 = \left\{ \begin{bmatrix} 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \end{bmatrix} \right\}$, and $\mathcal{C}_1 = \left\{ \begin{bmatrix} 1 & 1 \end{bmatrix} \right\}$. The desired output for class $\mathcal{C}_0$ is 0, and the desired output for class $\mathcal{C}_1$ is 1. Correspondingly, let $d(\mathbf{x}) = 0$ if $\mathbf{x} \in \mathcal{C}_0$, and otherwise, let $d(\mathbf{x}) = 1$ if $\mathbf{x} \in \mathcal{C}_1$.

(a) **(8 pts)** If possible, find w_0, w_1, w_2 that can separate $\mathcal{C}_0$ and $\mathcal{C}_1$ (i.e., provide the desired output for all 4 possible input vectors). Otherwise, prove that no choice of weights can separate the two classes.

Solution: This is equivalent to designing the AND gate. So, $w_0 = -\frac{3}{2}$, $w_1 = w_2 = 1$ will work.

(b) **(10 pts)** Recall that the perceptron training algorithm relies on the update $\mathbf{w} \leftarrow \mathbf{w} + \eta(d(\mathbf{x}) - y) \begin{bmatrix} 1 & \mathbf{x} \end{bmatrix}$, where $\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix}$ is the weight vector. Let $\eta = 1$ and the initial weight vector be given by $\mathbf{w} = \begin{bmatrix} -0.5 & 1 & 0 \end{bmatrix}$. Calculate the updated weights after two epochs of training.

Solution: Straightforward calculation. The final weights you get should be $\begin{bmatrix} -0.5 & 2 & 1 \end{bmatrix}$.

(c) **(7 pts)** Will the weights provided by the algorithm (as setup in (b)) eventually converge after a sufficiently larger number of epochs? Justify your answer.

Solution: Yes. The patterns are linearly separable so the PTA will converge.

Q4 (15 pts). Let

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \phi \left(\begin{bmatrix} w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \right), \text{ and } \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \phi \left(\begin{bmatrix} u_{10} & u_{11} & u_{12} \\ u_{20} & u_{21} & u_{22} \end{bmatrix} \begin{bmatrix} 1 \\ y_1 \\ y_2 \end{bmatrix} \right) \tag{1}$$

with the understanding that the activation function ϕ is applied component-wise. These equations define a two-layer neural network with 2 input nodes, 2 neurons in the hidden layer, and 2 output nodes.

(a) **(5 pts)** Draw the block diagram of the neural network with all inputs, outputs, weights labeled.

Solution: Left as exercise.

(b) **(10 pts)** Let $E = (d_1 - z_1)^2 + (d_2 - z_2)^4 + u_{22}^2$. Write down the expressions for $\frac{\partial E}{\partial w_{10}}$ and $\frac{\partial E}{\partial u_{22}}$. You may use the backpropagation algorithm. Your expressions may contain intermediate variables that you shall clearly define on the feedforward/feedback graphs.

Solution: Left as exercise.

Q5 (10 pts). Consider the activation function $\phi(v) = \frac{v}{1+|v|}$ defined for all real numbers.

(a) **(5 pts)** Find $\phi'(v) = \frac{\partial \phi}{\partial v}$.

Solution: For $v \geq 0$, we have $\phi(v) = \frac{v}{1+v}$ so that $\phi'(v) = \frac{1}{(1+v)^2}$. For $v \leq 0$, we have $\phi(v) = \frac{v}{1-v}$ so that $\phi'(v) = \frac{1}{(1-v)^2}$. Hence, for any v, we have $\phi'(v) = \frac{1}{(1+|v|)^2}$.

(b) **(5 pts)** Express $\phi'(v)$ in terms of $\phi(v)$ only.

Solution (by Aria Ameri, my solution was much less neater): We have $|\phi(v)| = \frac{|v|}{1+|v|}$. Negating both sides and adding 1, we get $1 - |\phi(v)| = \frac{1}{1+|v|}$. Therefore, $\phi'(v) = \frac{1}{(1+|v|)^2} = (1 - |\phi(v)|)^2$!

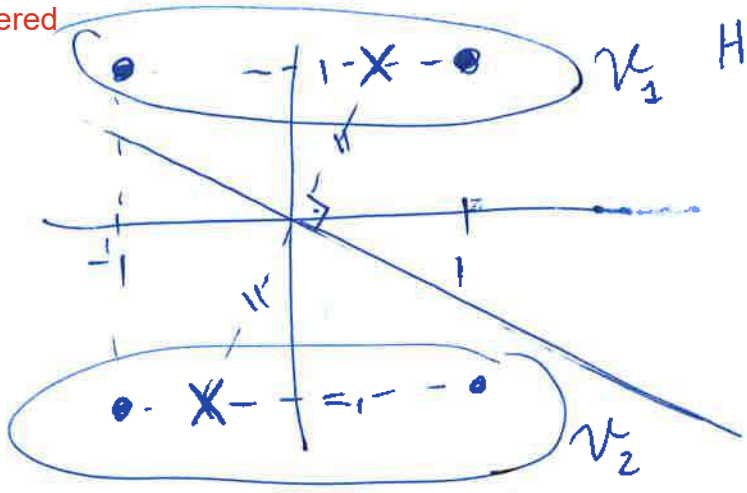
Ironic that this turned out to be toughest question of the exam with the worst overall performance. Only one person got this right. Some of you found essentially the same result, but you had conditioned on the negativity of positivity of v (Hence your expressions did not depend on $\phi(v)$ only).

ECE/CS 559 - Fall 2019 - Midterm #1.

Full Name: James Bond **ID Number:** 007

Q1 (10 pts). Consider applying the k-means algorithm to the set of vectors $\mathcal{C} = \{\begin{bmatrix}-1\\1\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}1\\-1\end{bmatrix}, \begin{bmatrix}-1\\-1\end{bmatrix}\}$ with initial centers $\begin{bmatrix}0.5\\1\end{bmatrix}, \begin{bmatrix}-0.5\\-1\end{bmatrix}$. What are the resulting centers after the algorithm converges? Recall that, given S is the input (training set), and c_i are the centers, the k-means algorithm relies on the update $c_i \leftarrow \frac{1}{|V_i|}\sum_{x \in V_i} x$, where $V_i = \{x \in S : \|x - c_i\| \leq \|x - c_j\|, \forall j\}$, and $|V_i|$ is the number of elements in V_i.

Not Covered



Hence, at iteration 1, the centers will be updated to $\begin{pmatrix}0\\1\end{pmatrix}$ and $\begin{pmatrix}0\\-1\end{pmatrix}$.

These centers will remain the same at subsequent iterations as V_1 & V_2 does not change.

Q2 (30 pts). Let u be the step activation function with $u(x) = 1$ if $x \geq 0$, and $u(x) = 0$, otherwise. Consider the perceptron $y = u(w_0 + w_1x_1 + w_2x_2)$, where w_1 and w_2 are the weights for inputs x_1 and x_2, respectively, w_0 is the perceptron bias, and y is the perceptron output. Let $\mathcal{C}_0 = \{\begin{bmatrix}0 & 2\end{bmatrix}, \begin{bmatrix}2 & 0\end{bmatrix}\}$, and $\mathcal{C}_1 = \{\begin{bmatrix}1 & 1\end{bmatrix}\}$. The desired output for class $\mathcal{C}_0$ is 0, and the desired output for class $\mathcal{C}_1$ is 1. Correspondingly, let $d(\mathbf{x}) = 0$ if $\mathbf{x} \in \mathcal{C}_0$, and otherwise, let $d(\mathbf{x}) = 1$ if $\mathbf{x} \in \mathcal{C}_1$.

(a) **(10 pts)** If possible, find w_0, w_1, w_2 that can separate $\mathcal{C}_0$ and $\mathcal{C}_1$ (i.e., provide the desired output for all 4 possible input vectors). Otherwise, prove that no choice of weights can separate the two classes.

We need

$w_0 + 2w_2 < 0$ (i)

$w_0 + 2w_1 < 0$ (ii)

$w_0 + w_1 + w_2 \geq 0$ (iii)

$-\frac{(i)-(ii)}{2} + (iii)$ leads to $0 > 0$

a contradiction.

So no choice of weights can separate the classes.

(b) **(10 pts)** Recall that the perceptron training algorithm relies on the update $\mathbf{w} \leftarrow \mathbf{w} + \eta(d(\mathbf{x}) - y)\begin{bmatrix}1 & \mathbf{x}\end{bmatrix}$, where $\mathbf{w} = \begin{bmatrix}w_0 & w_1 & w_2\end{bmatrix}$ is the weight vector. Let $\eta = 1$ and the initial weight vector be given by $\mathbf{w} = \begin{bmatrix}-1 & 0 & 0\end{bmatrix}$. Calculate the updated weights after one epoch of training.

With extended notation (biases)

$$C_0 = \left\{ \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} \right\}, \quad C_1 = \left\{ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\}$$

Epoch 1.

Feed $\begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$. $y = u\left(\begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}^T \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \right) = 0 =$ desired output

So, no update.

Feed $\begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}$, $y = u\left(\begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}^T \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} \right) = 0 =$ desired output

So, again, no update.

Feed $\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, $y = u\left(\begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}^T \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right) = 0 \neq$ desired output

So update $w \leftarrow \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} + 1 \cdot (1 - 0) \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$.

∴ Find weights after one epoch of training

$$\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}.$$

(c) **(10 pts)** Will the weights provided by the algorithm (as setup in (b)) eventually converge after a sufficiently larger number of epochs? Justify your answer.

No. If converges $\Rightarrow \exists w$ to separate the two classes. However, the classes are not linearly separable.

Q3 (40 pts): Consider a single-neuron network with input-output relationship $y = f(b + \mathbf{w}^T\mathbf{x})$, where y is the network output, f is some activation function, b is the bias term, $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ are the synaptic weights, and $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ is the network input.

(a) **(10 pts):** Let $E = (d - y)^2$, where d is a constant (a generic desired output). Write down the delta-learning rule (the gradient-descent update equations) for b, w_1, w_2 given learning parameter $\eta = \frac{1}{2}$.

Simple chain rule yields:

$$\begin{pmatrix} b \\ w_1 \\ w_2 \end{pmatrix} \leftarrow \begin{pmatrix} b \\ w_1 \\ w_2 \end{pmatrix} + (d - y) f'(b + w^T x) \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}.$$

(b) **(15 pts):** Consider the same delta-learning setup as in (a) with the activation function

$$f(v) = \begin{cases} v, & v \geq 0, \\ 0, & v < 0. \end{cases}$$

This is also known as the rectified linear unit (ReLU). Consider the training vectors $\mathbf{x}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, $\mathbf{x}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, with desired outputs $d_1 = -1$, $d_2 = 2$, respectively. Find the updated bias and the updated weights after 2019 epochs of online learning given initial conditions $b = 0$, $\mathbf{w} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

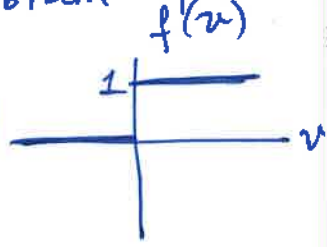
Epoch 1

Show $\begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$. Local field $v = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}^T \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} = -1$

$y = f(v) = 0$

According to the update rule in (a), we obtain

$$\begin{pmatrix} b \\ w_1 \\ w_2 \end{pmatrix} \leftarrow \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + (-1 - 0) f'(-1) \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$$



Since $f'(-1) = 0$, we have no update.

Show $\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$. Local field $v = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}^T \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = 1$ $y = f(1) = 1$

$$\begin{pmatrix} b \\ w_1 \\ w_2 \end{pmatrix} \leftarrow \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + (2 - 1) f'(1) \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}.$$

Epoch 2.

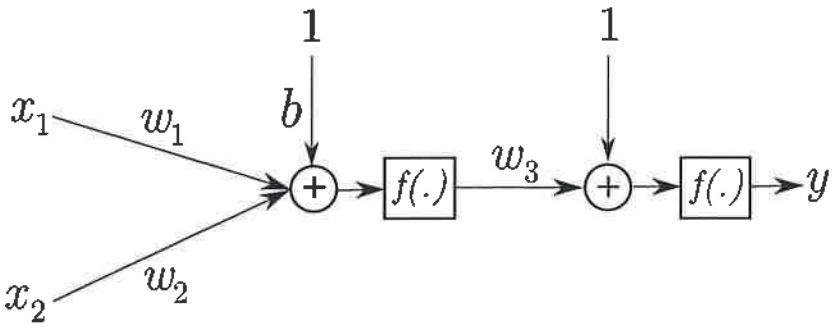
Show $\begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$. $v = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}^T \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} = -1 \Rightarrow$ no update.

Show $\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$. $v = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}^T \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = 3$, $y = f(v) = 3$

$$\begin{pmatrix} b \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} + (2 - 3) f'(3) \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

At the end of epoch 2, we end up with the same weights we began with. So after even numbered epochs, the weights are $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, and after odd-numbered epochs, the weights are $\begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$. In particular, after 2019 epochs, we get $\begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$.

(c) **(15 pts)** Consider now a multi-layer network as shown below.



Consider again a general f, as in (a). Let $E = (d - y)^4$. Find the gradient-descent update equations for b, w_1, w_2, w_3 given $\eta = \frac{1}{4}$.

$y = f(1 + w_3 f(b + w_1 x_1 + w_2 x_2))$. So, by chain rule.

$$w_3 \leftarrow w_3 + (d-y)^3 f'(1 + w_3 f(b + w_1 x_1 + w_2 x_2)) \times f(b + w_1 x_1 + w_2 x_2).$$

$$\begin{bmatrix} b \\ w_1 \\ w_2 \end{bmatrix} \leftarrow \begin{bmatrix} b \\ w_1 \\ w_2 \end{bmatrix} + (d-y)^3 f'(1 + w_3 f(b + w_1 x_1 + w_2 x_2)) \times w_3 f'(b + w_1 x_1 + w_2 x_2) \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}.$$

Q4 (20 pts). Let $f(x, y) = x^2 + xy + y^2$. Consider Newton's method given by the update rule $\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} x \\ y \end{bmatrix} - H^{-1}g$, where the gradient and the Hessian are given by $g = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$ and $H = \begin{bmatrix} \frac{\partial f^2}{\partial x \partial x} & \frac{\partial f^2}{\partial x \partial y} \\ \frac{\partial f^2}{\partial y \partial x} & \frac{\partial f^2}{\partial y \partial y} \end{bmatrix}$, respectively.

(a) **(10 pts)** Calculate g and H.

$$g = \begin{bmatrix} 2x + y \\ 2y + x \end{bmatrix} \qquad H = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}.$$

(b) **(10 pts)** Consider the initial point $x = 2019, y = 2020$. Find the next point after one update with Newton's method.

$$H^{-1} = \frac{1}{3}\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}. \quad \text{So, } H^{-1}g = \begin{bmatrix} x \\ y \end{bmatrix}$$

and any initial point $\begin{bmatrix} x \\ y \end{bmatrix}$ will end up at $\begin{bmatrix} x \\ y \end{bmatrix} - H^{-1}g = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ after one update with Newton's method. This is also the global minimum.