ECE499/ECE590
Machine Learning for Embedded Systems
(Fall 2021)

Lecture 2: Train Neural Networks

Weiwen Jiang, Ph.D.

Electrical and Computer Engineering

George Mason University

wjiang8@gmu.edu

Clarification

Clarification:
Mid-Terms

- No hand-writing mid-terms

$\neq$

- No mid-terms

Undergraduate (ECE 499)

- Homework & Labs 50%
  - **5 in total, including 2 mid-term labs**
- Paper Critiques 10%
- Project progress review 10%
- Project final review 30%

Graduate (ECE 590)

- Homework & Labs 50%
  - **5 in total, including 2 mid-term labs**
- Research paper presentation 20%
- Project progress review 10%
- Project final review/report 20%

Clarification:
“Repeatable” in Course Scheduling Systsem

You can select 499 in the future

Associated Term: Fall 2021
CRN: 84450
Campus: Fairfax
Schedule Type: Lec/Sem #1 (Repeatable)
Instructional Method: On-campus F2F 76-100%
Section Number: 001
Subject: Electrical & Computer Enginrg
Course Number: 499
Title: Mach Lrning Embedded Systems
Credit Hours: 3
Grade Mode: No Section specified grade mode, please see Catalog link below for more information.

Clarification:
Office Hours

Instructor	Dr. Weiwen Jiang
E-Mail	wjiang8@gmu.edu
Office Hour	Monday 14:00 am - 15:30 am
Office	Room 3247, Nguyen Engineering Building
Zoom	https://go.gmu.edu/zoom4weiwen

TA	Zhepeng Wang
E-Mail	zwang48@gmu.edu
Office Hour	Sep. 1st, 15:00 am - 17:00 am (this week only)
Location	Room 3202, Nguyen Engineering Building
Zoom	https://zoom.us/j/9935038408?pwd=QVpuQ3M1QW1LYXhoL3JyMk95RkxHQT09

https://go.gmu.edu/ml4emb

Clarification:
Readings

Schedule and Documents

[499 Syllabi] [590 Syllabi]

W	Date	Topic	Documents	Note
1	Aug 23	Course Information & Introduction to Machine Learning	[Slides] [Lab1]	Lab 1 releases
2	Aug 30	Train Neural Networks		**Lab 1 Due: 1 pm, Sep 3**
3	Sep 13	Deep Convolutional Neural Networks (CNN)		

Readings and Tutorial

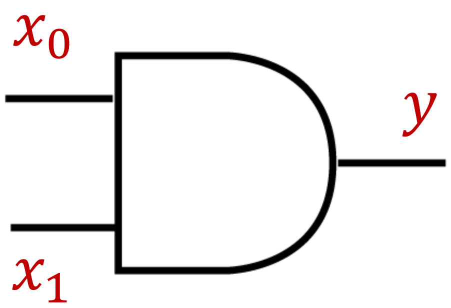
W	Date	Reading (R) & Tutorial (TT)
1	Aug 23	[R1]
2	Aug 30	[R2] [TT1] [TT1 Codes]
3	Sep 13	
4	Sep 20	

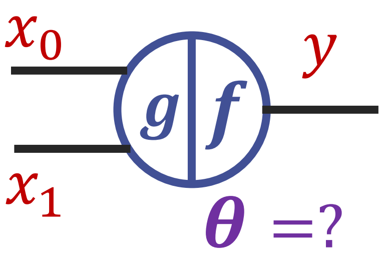
https://go.gmu.edu/ml4emb

Review of Previous Lecture

McCulloch-Pitts Neuron

Boolean function ‘AND’ can be implemented by using MP Neuron





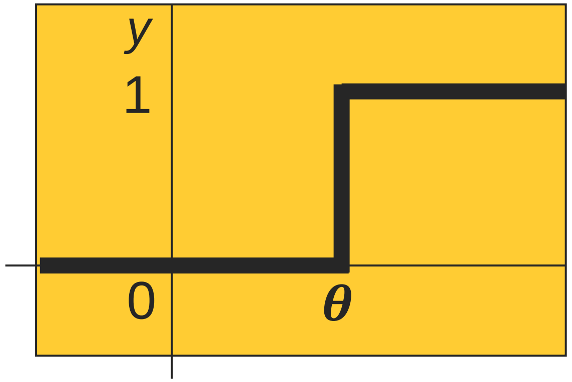
$$a = g(x_0, x_1) = w_0 x_0 + w_1 x_1 = x_0 + x_1$$

$$f(a) = \begin{cases} 1 & a > \theta \\ 0 & a \leq \theta \end{cases}$$

Given $w_0 = 1; w_1 = 1$; Determine θ

AND Gate

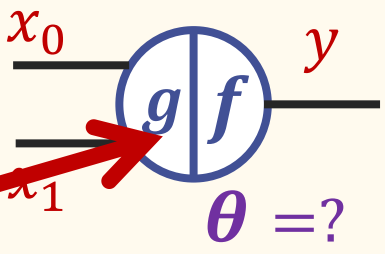
x_0	x_1	y
0	0	0
0	1	0
1	0	0
1	1	1



x_0	x_1	$g(x_0, x_1)$	Wanted $f(g(x_0, x_1))$
0	0	0+0=0	0
0	1	0+1=1	0
1	0	1+0=1	0
1	1	1+1=2	1

$$\therefore \theta \in [1, 2)$$

McCulloch-Pitts Neuron
Pytorch Implementation of AND




```
import torch
import torch.nn as nn

class AND_Preceptron(nn.Module):
    def __init__(self):
        """Initialize the layers of the model."""
        super(AND_Preceptron, self).__init__()
        self.layer1 = nn.Linear(2,1)
        self.nonlin_1 = torch.heaviside
    def forward(self,x):
        x = self.layer1(x)
        x = self.nonlin_1(x,torch.tensor(0.))
        return x

and_net = AND_Preceptron()
```

Structure

Data Flow

$$f(a) = \begin{cases} 1 & a > \theta \\ 0 & a \leq \theta \end{cases}$$

```
# Fixing weights and bias
w = torch.tensor([[+1.],[+1.]])
and_net.layer1.weight = nn.Parameter(w.t())
```

Weight Assignment
(nn.Linear is not for MP Neuron, but for general neuron)

```
theta = torch.tensor([1.5])
bias = -1*theta
and_net.layer1.bias = nn.Parameter(bias)
```

Using bias to set θ
(Bias should be -θ)

```
# Train_data
input_tensors = [torch.Tensor([0,0]), torch.Tensor([0,1]), torch.Tensor([1,0]), torch.Tensor([1,1])]
```

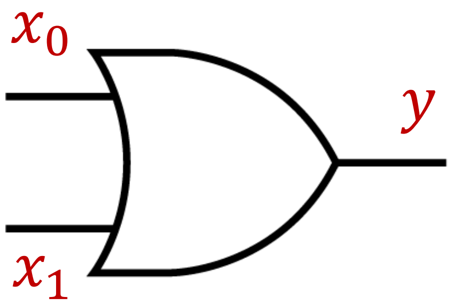
Inputs

```
for input in input_tensors:
  print(input, and_net(input).data)
```

Stream inputs to Structure (Neural Network)

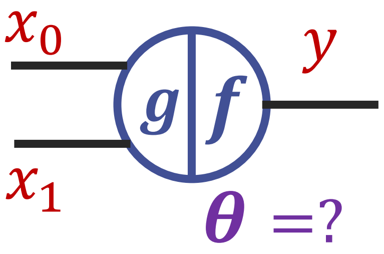
McCulloch-Pitts Neuron

Boolean function ‘OR’ can be implemented by using MP Neuron



OR Gate

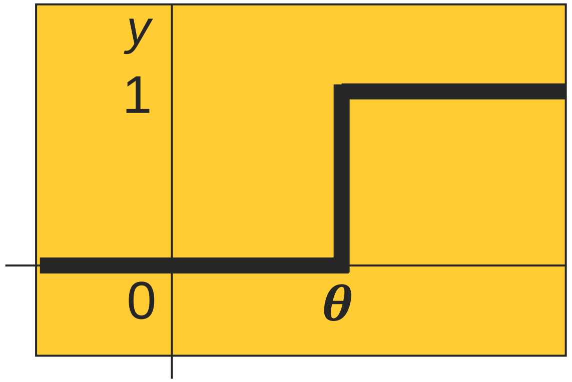
x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	1



$$a = g(x_0, x_1) = w_0 x_0 + w_1 x_1 = x_0 + x_1$$

$$f(a) = \begin{cases} 1 & a > \theta \\ 0 & a \leq \theta \end{cases}$$

Given $w_0 = 1; w_1 = 1$; Determine θ



x_0	x_1	$g(x_0, x_1)$	Wanted $f(g(x_0, x_1))$
0	0	0+0=0	0
0	1	0+1=1	1
1	0	1+0=1	1
1	1	1+1=2	1

$$\therefore \theta \in [0, 1)$$

McCulloch-Pitts Neuron
Pytorch Implementation of OR

Where to be modified?

```
import torch
import torch.nn as nn

class OR_Preceptron(nn.Module):
```

Structure

```
    def __init__(self):
        """Initialize the layers of the model."""
        super(OR_Preceptron, self).__init__()
        self.layer1 = nn.Linear(2,1)
        self.nonlin_1 = torch.heaviside
```

Data Flow

```
    def forward(self,x):
        x = self.layer1(x)
        x = self.nonlin_1(x,torch.tensor(0.))
        return x

or_net = OR_Preceptron()
```

```
# Fixing weights and bias
w = torch.tensor([[+1.],[+1.]])
or_net.layer1.weight = nn.Parameter(w.t())
```

Weight Assignment
(nn.Linear is not for MP Neuron, but for general neuron)

```
theta = torch.tensor([1.5])
bias = -1*theta
or_net.layer1.bias = nn.Parameter(bias)
```

Using bias to set θ
(Bias should be -θ)

Inputs

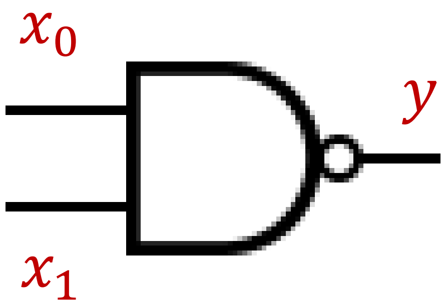
```
# Train_data
input_tensors = [torch.Tensor([0,0]), torch.Tensor([0,1]), torch.Tensor([1,0]), torch.Tensor([1,1])]
```

```
for input in input_tensors:
  print(input, or_net(input).data)
```

Stream inputs to Structure (Neural Network)

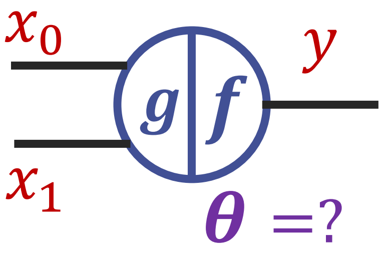
Perceptron

Boolean function 'NAND' can be implemented



NAND Gate

x_0	x_1	y
0	0	1
0	1	1
1	0	1
1	1	0



$$a = g(x_0, x_1) = w_0 x_0 + w_1 x_1 = x_0 + x_1$$

$$f(a) = \begin{cases} 1 & a > \theta \\ 0 & a \leq \theta \end{cases}$$

Determine $w_0; w_1; \theta$

x_0	x_1	$g(x_0, x_1)$	Wanted $f(g(x_0, x_1))$
0	0	$w_0 \cdot 0 + w_1 \cdot 0{=}0$	1
0	1	$w_0 \cdot 0 + w_1 \cdot 1 = w_1$	1
1	0	$w_0 \cdot 0 + w_1 \cdot 1 = w_0$	1
1	1	$w_0 \cdot 0 + w_1 \cdot 1 = w_0 + w_1$	0

$\therefore$
- $w_0 + w_1 < 0$
- $w_0 + w_1 < w_0$
- $w_0 + w_1 < w_1$
- $\theta \in [w_0 + w_1, min(0, w_0, w_1))$

$\Rightarrow$
- $w_0 < -w_1$
- $w_1 < 0$
- $w_0 < 0$
- $\theta \in [w_0 + w_1, min(w_0, w_1))$

$\Rightarrow$
- ✓ $w_1 = -1, w_2 = -1, \theta \in [-2, -1)$
- ✓ $w_1 = -2, w_2 = -2, \theta \in [-4, -2)$
- ✓ $w_1 = -0.1, w_2 = -2, \theta \in [-2.1, -2)$
- ✓ $w_1 = -0.1, w_2 = -0.3, \theta \in [-0.4, -0.3)$

Perceptron
Pytorch Implementation of NAND

Where to be modified?

```
import torch
import torch.nn as nn

class NAND_Preceptron(nn.Module):
    def __init__(self):
        """Initialize the layers of the model."""
        super(NAND_Preceptron, self).__init__()
        self.layer1 = nn.Linear(2,1)
        self.nonlin_1 = torch.heaviside
    def forward(self,x):
        x = self.layer1(x)
        x = self.nonlin_1(x,torch.tensor(0.))
        return x

nand_net = NAND_Preceptron()
```

Structure

Data Flow

```
# Fixing weights and bias
w = torch.tensor([[+1.],[+1.]])
nand_net.layer1.weight = nn.Parameter(w.t())
```

Weight Assignment
(nn.Linear is not for MP Neuron, but for general neuron)

```
theta = torch.tensor([1.5])
bias = -1*theta
nand_net.layer1.bias = nn.Parameter(bias)
```

Using bias to set θ
(Bias should be -θ)

```
# Train_data
input_tensors = [torch.Tensor([0,0]), torch.Tensor([0,1]), torch.Tensor([1,0]), torch.Tensor([1,1])]
```

Inputs

```
for input in input_tensors:
  print(input, nand_net(input).data)
```

Stream inputs to Structure (Neural Network)

Put Them Together: MLP
Pytorch Implementation of AND, OR, NAND

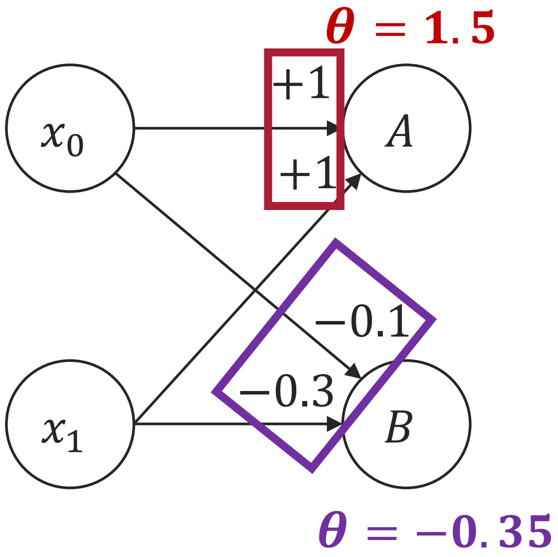
$A = AND(x_0, x_1)$

$B = NAND(x_0, x_1)$

$Y = OR(x_0, x_1)$

$$Y = (x_0\ AND\ x_1)\ OR\ \overline{(x_0\ AND\ x_1)}$$

x_0	x_1	A	B
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0



Layer 1

```
self.layer1 = nn.Linear(2,2)

Y_net = Y_Preceptron()

# Fixing weights and bias
w = torch.tensor([[+1.,-0.1],
                  [+1.,-0.3]])
Y_net.layer1.weight = nn.Parameter(w.t())

theta = torch.tensor([1.5, -0.35])
bias = -1*theta
Y_net.layer1.bias = nn.Parameter(bias)
```

Multi-Output Network

Pytorch Implementation of AND and NAND

```
import torch
import torch.nn as nn

class L1_Preceptron(nn.Module):
    def __init__(self):
        """Initialize the layers of the model."""
        super(L1_Preceptron, self).__init__()
        self.layer1 = nn.Linear(2,2)
        self.nonlin = torch.heaviside
    def forward(self,x):
        x = self.layer1(x)
        x = self.nonlin(x,torch.tensor(0.))
        return x

L1_net = L1_Preceptron()

# Fixing weights and bias
w = torch.tensor([[+1.,-0.1],
                  [+1.,-0.3]])
L1_net.layer1.weight = nn.Parameter(w.t())

theta = torch.tensor([1.5, -0.35])
bias = -1*theta
L1_net.layer1.bias = nn.Parameter(bias)

# Train_data
input_tensors = [torch.Tensor([0,0]), torch.Tensor([0,1]), torch
.Tensor([1,0]), torch.Tensor([1,1])]

for input in input_tensors:
  print(input, L1_net(input).data)
```

Put Them Together: MLP
Pytorch Implementation of AND, OR, NAND

$A = AND(x_0, x_1)$

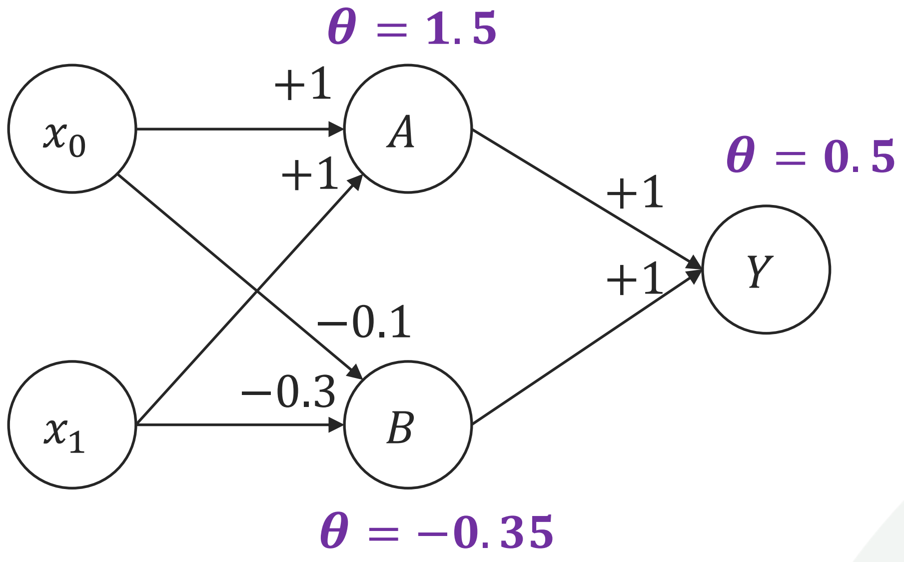
$B = NAND(x_0, x_1)$

$Y = OR(x_0, x_1)$

$$Y = (x_0\ AND\ x_1)\ OR\ \overline{(x_0\ AND\ x_1)}$$

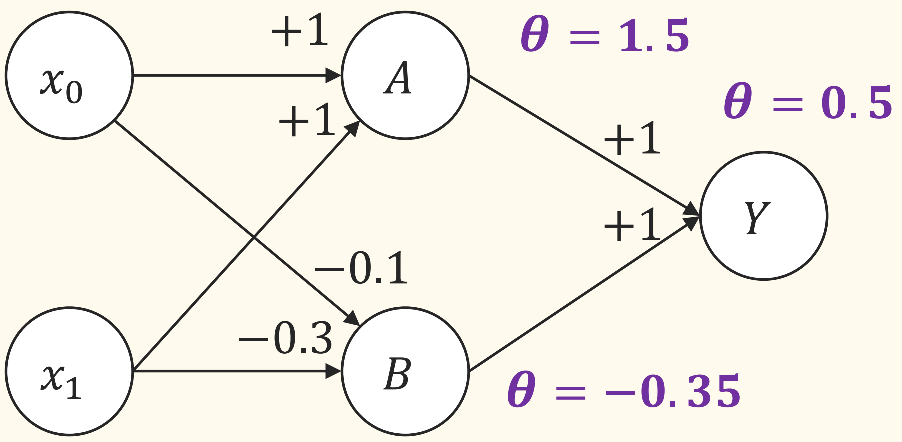
x_0	x_1	A	B	Y
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Layer 2



MLP Network

Pytorch Implementation of AND, OR and NAND

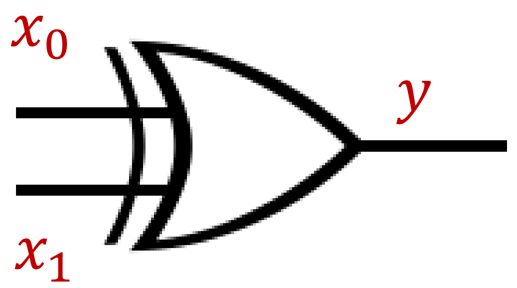



```python
import torch
import torch.nn as nn

class Y_Preceptron(nn.Module):
    def __init__(self):
        """Initialize the layers of the model."""
        super(Y_Preceptron, self).__init__()
        self.layer1 = nn.Linear(2,2)
        self.nonlin = torch.heaviside
        self.layer2 = nn.Linear(2,1)
    def forward(self,x):
        x = self.layer1(x)
        x = self.nonlin(x,torch.tensor(0.))
        x = self.layer2(x)
        x = self.nonlin(x,torch.tensor(0.))
        return x
```

```python
Y_net = Y_Preceptron()

# Fixing weights and bias
w = torch.tensor([[+1.,-0.1],
                  [+1.,-0.3]])
Y_net.layer1.weight = nn.Parameter(w.t())

theta = torch.tensor([1.5, -0.35])
bias = -1*theta
Y_net.layer1.bias = nn.Parameter(bias)

# Fixing weights and bias
w = torch.tensor([[+1.],[+1.]])
Y_net.layer2.weight = nn.Parameter(w.t())

theta = torch.tensor([0.5])
bias = -1*theta
Y_net.layer2.bias = nn.Parameter(bias)


# Train_data
input_tensors = [torch.Tensor([0,0]), torch.Tensor([0,1]),
torch.Tensor([1,0]), torch.Tensor([1,1])]

for input in input_tensors:
  print(input, Y_net(input).data)
```

Multi-Layer Perceptron (MLP)

Solving



$$= (x_0 \ or \ x_1) \ and \ (\overline{x_0 \ and \ x_1})$$

XOR Gate

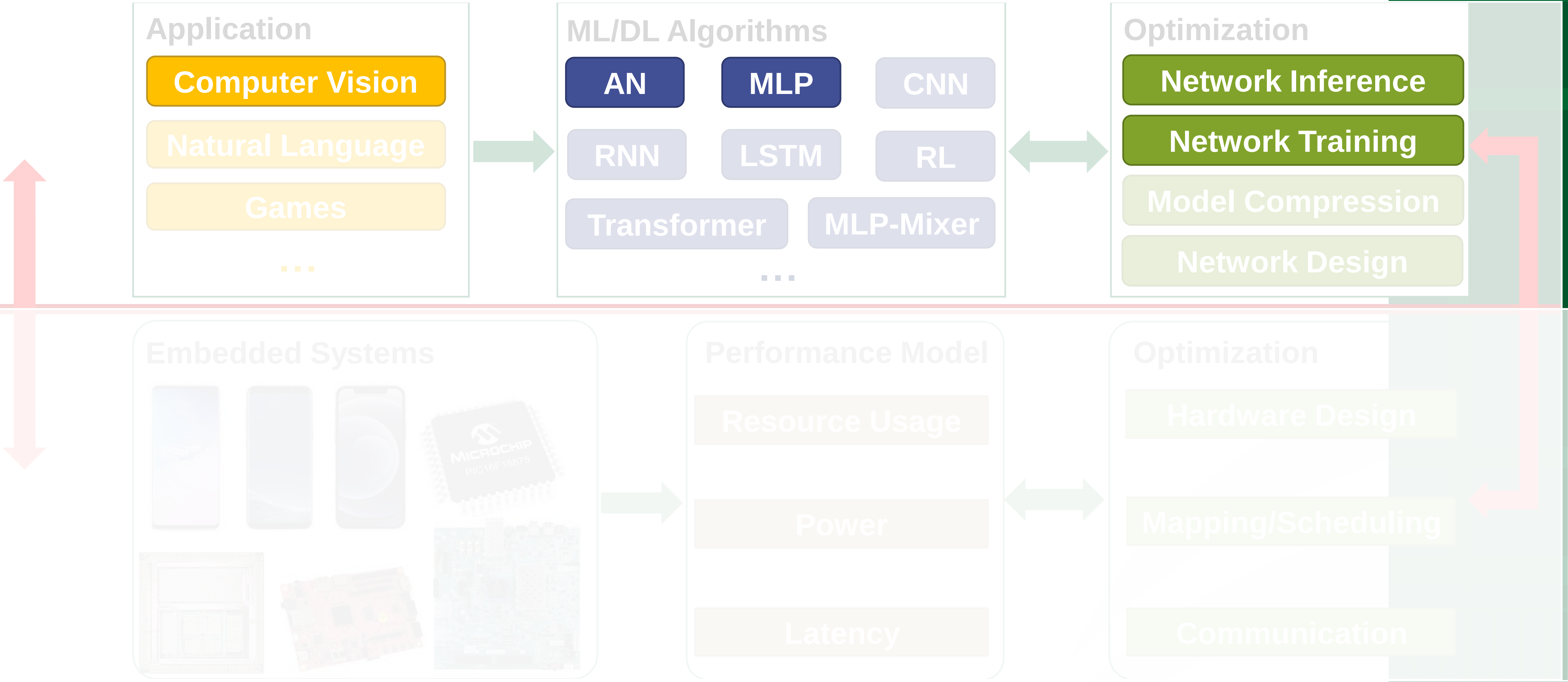
x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

Artificial Neuron Design

- **Idealized neuron models**
  - Idealization removes complicated details that are not essential for understanding the main principles.
  - It allows us to apply mathematics and to make analogies.
- **Break the limitations on MP Neuron**
  - What about non-boolean inputs (say, real number)? ✔
  - What if we want to assign more weight (importance) to some inputs? ✔
  - What about functions which are not linearly separable ? ✔
  - Do we always need to hand code the threshold? ? **=> Training**

Lecture 2

Week 2: From Inference to Training

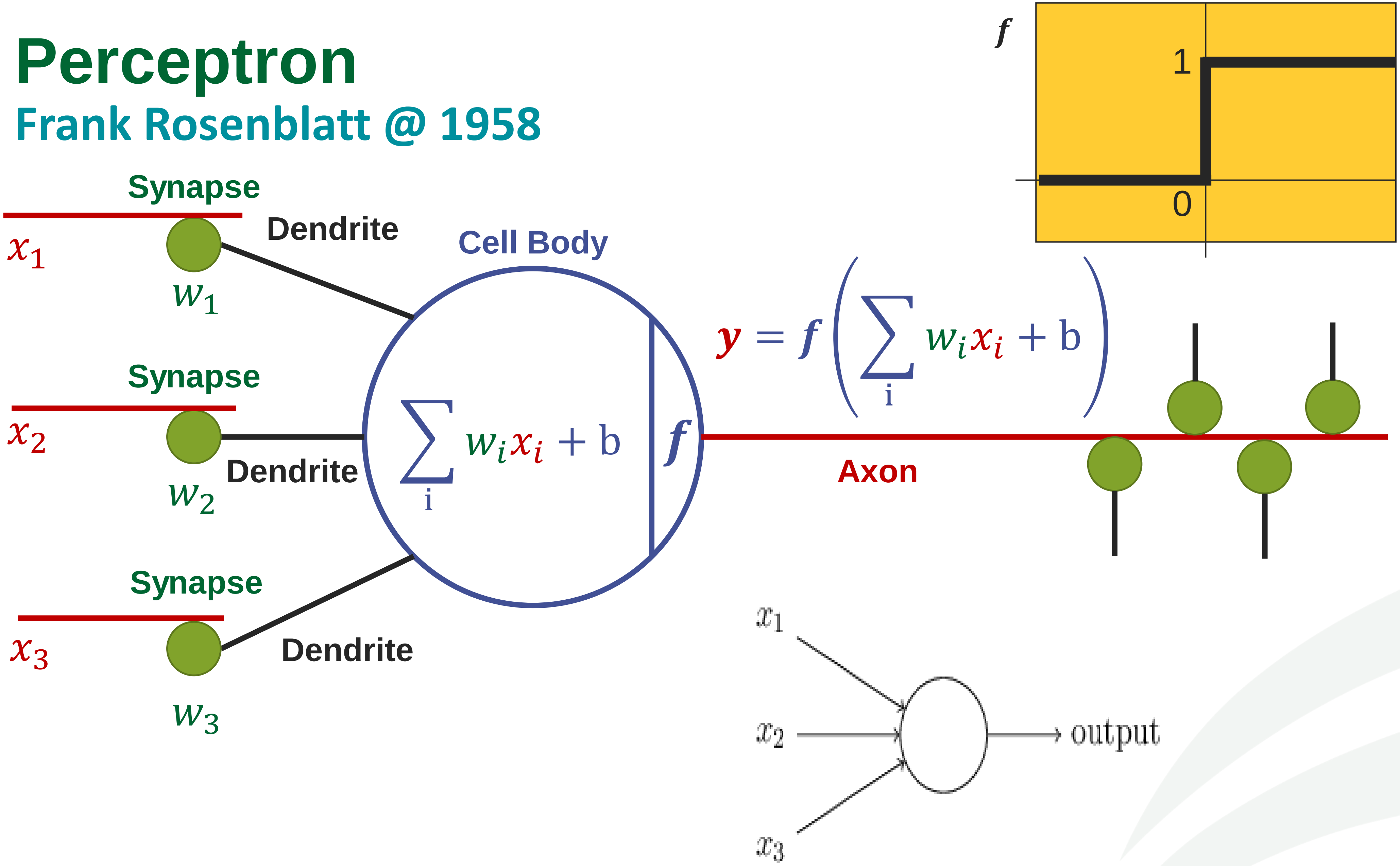


Agenda

- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

Perceptron

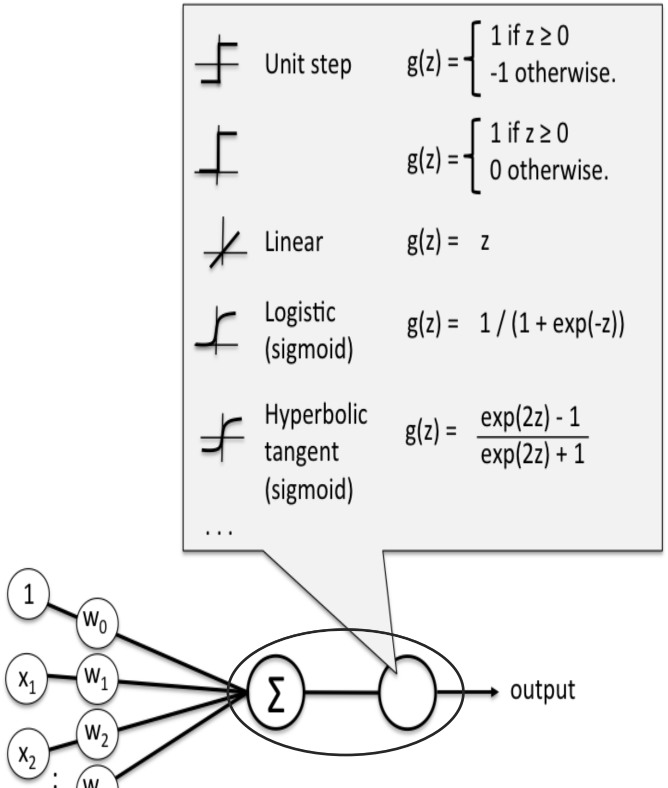
Frank Rosenblatt @ 1958



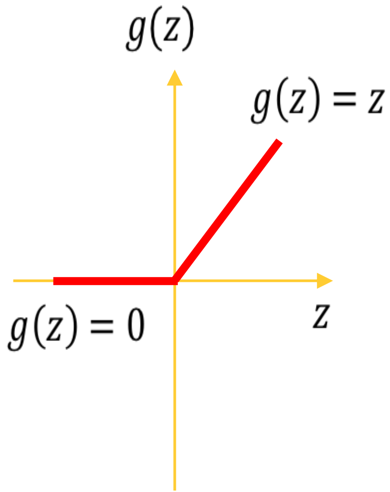
Various Activation Functions

- People invent new activation functions and publish papers

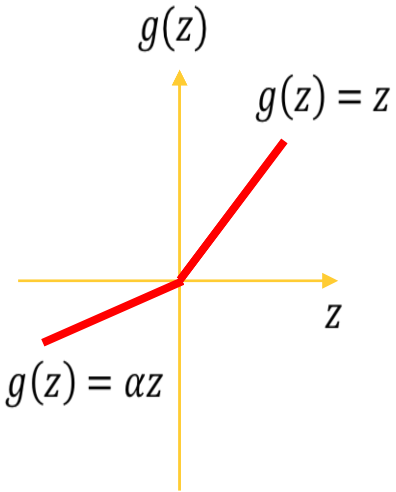
  "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," K. He et al., ICCV 2015



A selection of commonly used activation functions for artificial neurons.



Rectified Linear (ReLU) activation function

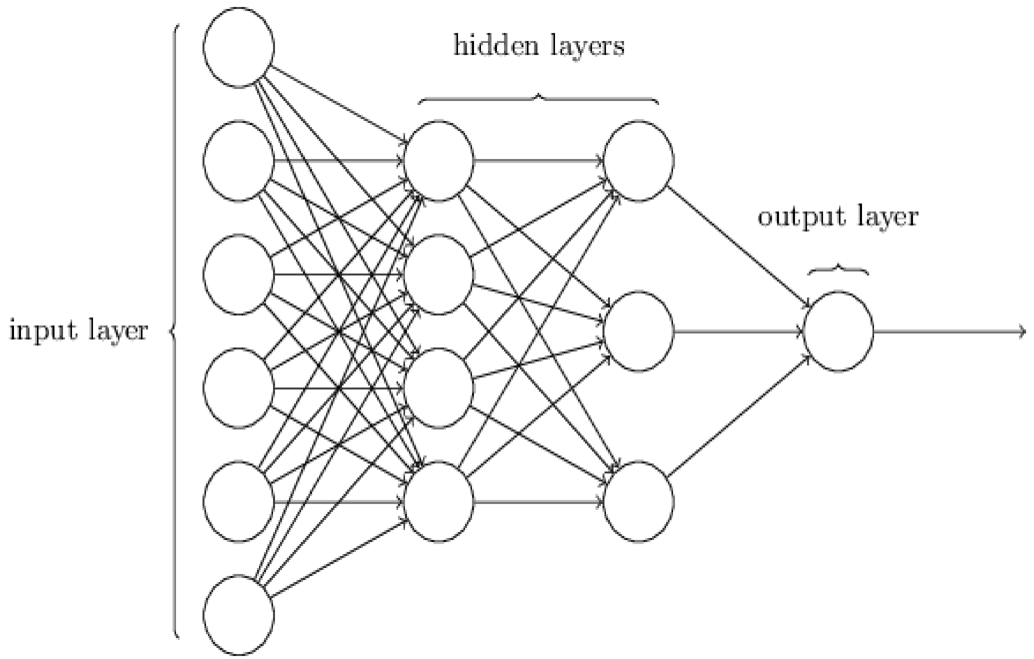


Parameterized Rectified Linear (PReLU) activation function

Latest research progress on defining new activation functions

Neural Network Terminologies

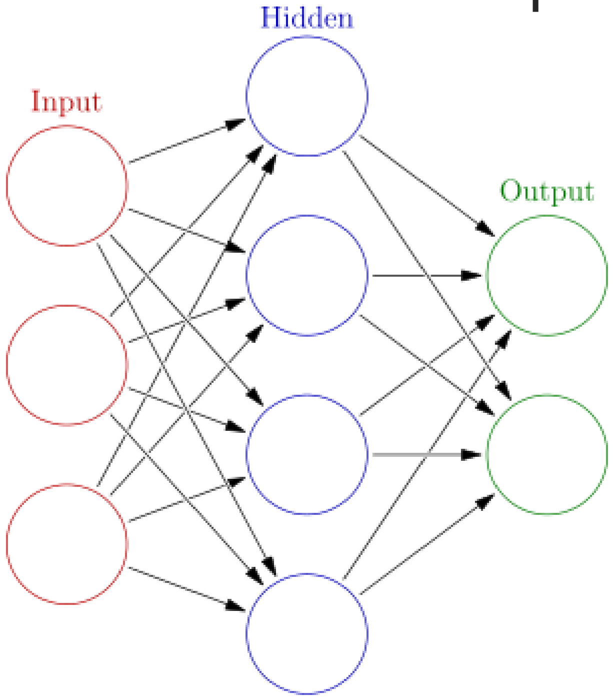
- Input layer, output layer and hidden layers



For historical reasons, such multiple layer networks are sometimes called **Multi-Layer Perceptrons** or **MLPs**, though they are mostly made up of **sigmoid neurons**, not HEAVISIDE

Neural Network for Multi-Class Classification

- Since our neuron (such as sigmoid neuron) is designed for (binary) classification problem, the so-built neuron network would be naturally suitable for solving (binary) classification problems
- Moreover, there is nothing to stop us from having multiple outputs, so it can be used for multi-class classification problems

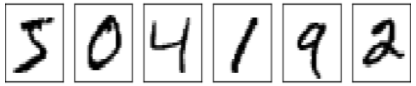


Example for Multi-Class Classification Formulation

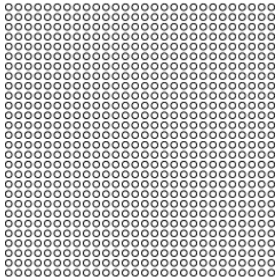
- Classify images of handwriting digits to 10 classes

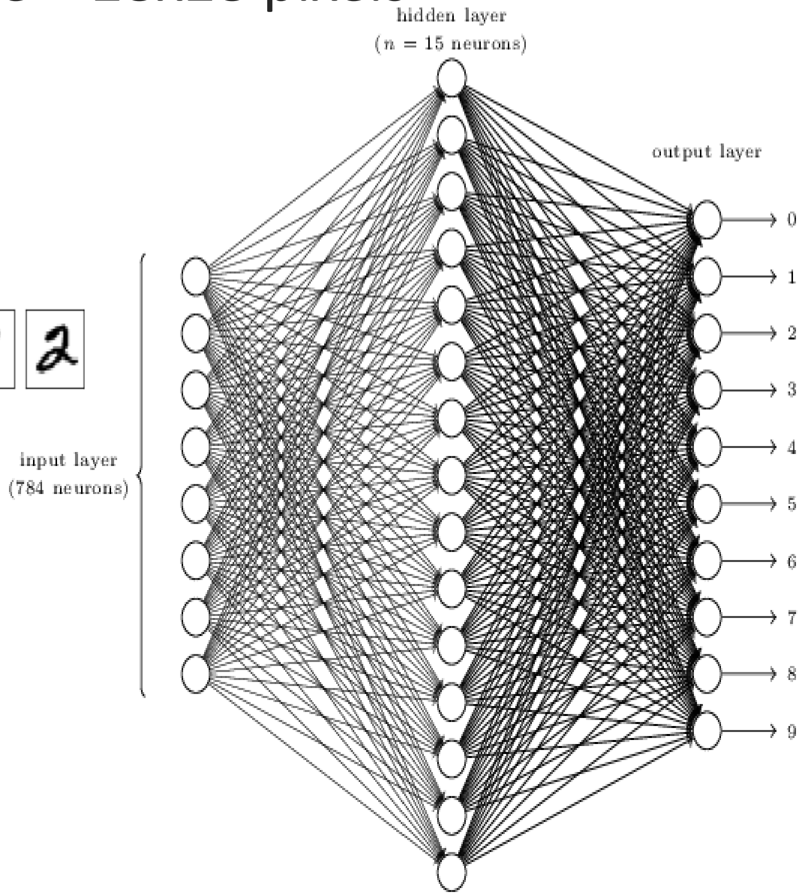
  Image size = 28x28 pixels

$x^{(i)}$



28x28 pixels





$y^{(i)}$

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

Agenda

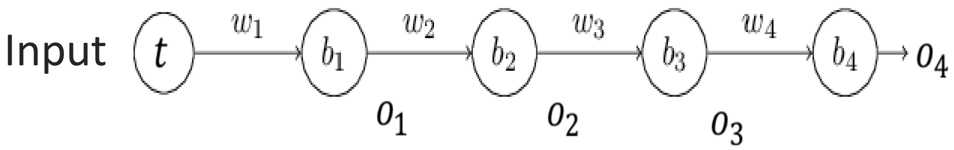
- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

Forward propagation

- Given a neural network model with weights and biases (parameters), forward propagation computes the outputs, starting from the inputs

  A simple example:

  Input to each neuron: $z = b + wx$ Neuron activation function: $g(z)$



$$o_1 = g(z_1) = g(b_1 + w_1 t)$$

$$o_2 = g(z_2) = g(b_2 + w_2 o_1)$$

$$o_3 = g(z_3) = g(b_3 + w_3 o_2)$$

$$o_4 = g(z_4) = g(b_4 + w_4 o_3)$$

  This is also called "inference"

- The same procedure holds for the more complicated neural networks too

  The output class is determined by the maximum value among all output neurons

Agenda

- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
  - Math
  - A quick review of unconstrained optimization
  - SGD and backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

Derivative of a Function

- The first order derivative of a function at point $(x, f(x))$ is defined as the instantaneous rate of change at that point, which is given by the limit of the average rate of change as

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- Some examples
  - Power rule $\frac{d}{dx}(x^p) = px^{p-1}$
  - Exponential rule $\frac{d}{dx}(b^x) = b^x \ln(b)$ $\frac{d}{dx}(e^x) = e^x$
  - Logarithm rule $\frac{d}{dx}(log_b(x)) = \frac{1}{x}\frac{1}{\ln(b)}$ $\frac{d}{dx}(ln(x)) = \frac{1}{x}$
  - Constant rule $\frac{d}{dx}(C) = 0$

Properties of Derivatives

- Constant

$$\frac{d}{dx}(cf(x)) = c\frac{d}{dx}(f(x))$$

- Sum and subtraction

$$\frac{d}{dx}(f(x) \pm g(x)) = \frac{d}{dx}(f(x)) \pm \frac{d}{dx}(g(x))$$

- Product

$$\frac{d}{dx}\big(f(x)g(x)\big) = \frac{d}{dx}\big(f(x)\big)g(x) + f(x)\,\frac{d}{dx}(g(x))$$

- Division

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{\frac{d}{dx}\big(f(x)\big)g(x) - f(x)\,\frac{d}{dx}(g(x))}{g(x)^2}$$

- Chain rule

$$\frac{d}{dx}(f(g(x))) = \frac{d}{dx}\big(f(g(x))\big)\frac{d}{dx}(g(x))$$

So Far, the Function and Derivatives are Defined for Scalars

- **Scalars**: 1 dimensional data
- Now let’s extend the same concept to

  Function of a **vector** of variables

  A vector function of a vector of variables

Gradient & Hessian

- Let f be a real-valued function of n variables

$$f(x) = f(x_1, x_2, \dots, x_n)$$

- **Gradient** is defined as a vector of first derivatives
- **Hessian** is defined as a matrix of second derivatives (which is also symmetric)

$$\nabla f(x) \equiv \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

An Example of Gradient & Hessian

$$f(x) = f(x_1, x_2, x_3) = \frac{1}{2}(4x_1^2 + 4x_1x_2 + 2x_1x_3 + 5x_2^2 + 6x_2x_3 + 7x_3^2) + 2x_1 - 8x_2 + 9x_3$$

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \frac{\partial f(x)}{\partial x_3} \end{bmatrix} = \begin{bmatrix} 4x_1 + 2x_2 + x_3 + 2 \\ 2x_1 + 5x_2 + 3x_3 - 8 \\ x_1 + 3x_2 + 7x_3 + 9 \end{bmatrix}$$

$$H = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 x_2} & \frac{\partial^2 f(x)}{\partial x_1 x_3} \\ \frac{\partial^2 f(x)}{\partial x_2 x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \frac{\partial^2 f(x)}{\partial x_2 x_3} \\ \frac{\partial^2 f(x)}{\partial x_3 x_1} & \frac{\partial^2 f(x)}{\partial x_3 x_2} & \frac{\partial^2 f(x)}{\partial x_3^2} \end{bmatrix} = \begin{bmatrix} 4 & 2 & 1 \\ 2 & 5 & 3 \\ 1 & 3 & 7 \end{bmatrix}$$

An Example of Gradient & Hessian (continued)

- You may have noticed the previous example is a quadratic form

$$
\begin{aligned}
f(x) = f(x_1, x_2, x_3) &= \frac{1}{2}(4x_1^2 + 4x_1x_2 + 2x_1x_3 + 5x_2^2 + 6x_2x_3 + 7x_3^2) + 2x_1 - 8x_2 + 9x_3 \\
&= \frac{1}{2}(4x_1^2 + 2x_1x_2 + x_1x_3 + 2x_1x_2 + 5x_2^2 + 3x_2x_3 + x_1x_3 + 3x_2x_3 + 7x_3^2) + 2x_1 - 8x_2 + 9x_3 \\
&= \frac{1}{2}x^TQx + b^Tx
\end{aligned}
$$

$$
\mathrm{Q}=\begin{bmatrix} 4 & 2 & 1 \\ 2 & 5 & 3 \\ 1 & 3 & 7 \end{bmatrix} \qquad b = \begin{bmatrix} 2 \\ -8 \\ 9 \end{bmatrix}
$$

$$
\nabla f(x) = Qx + b = \begin{bmatrix} 4x_1 + 2x_2 + x_3 + 2 \\ 2x_1 + 5x_2 + 3x_3 - 8 \\ x_1 + 3x_2 + 7x_3 + 9 \end{bmatrix}
$$

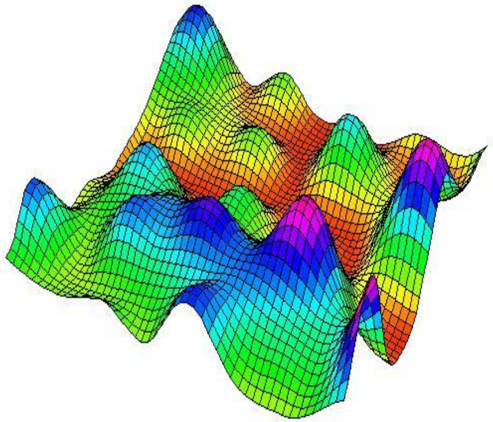
$$
H = Q = \begin{bmatrix} 4 & 2 & 1 \\ 2 & 5 & 3 \\ 1 & 3 & 7 \end{bmatrix}
$$

Agenda

- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
  - Math
  - A quick review of unconstrained optimization
  - SGD and backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

Unconstrained Optimization Problem

$$\text{Minimize}_{x \in S} \; f(x)$$



- No fundamental difference between minimization and maximization problems

$$\text{Maximize}_{x \in S} \; f(x) \equiv \text{Minimize}_{x \in S} \; -f(x)$$

- For optimization, a first-order necessary condition for a local minimizer is given by

$$\nabla f(x) = 0$$

General Optimization Algorithm

- Specify some initial guess of the solution x(0)
- For k=0,1,...
  - If x(k) is optimal, stop
  - Determine an improved estimate of the solution

$$x(k+1) = x(k) + \alpha(k) * p(k)$$

$\alpha(k)$: *a scalar for step length*
$p(k)$: *a search direction*

- Most practical optimization algorithms follow this paradigm
- It is an iterative algorithm
- The computed “solution” is only an approximation
  - Stop condition: typically when the value of f(k), x(k) and/or p(k) changes little or some approximation of the first-order condition ($\nabla f(x) = 0$) is satisfied

Gradient Descent Method

- The simplest method with a descent direction defined as

  $p(k) = -\nabla f\big(x(k)\big)$ $p(k)$ is the search direction: $x(k+1) - x(k)$

  – This is indeed a **descent direction**

  Taylor series approximation for $f(\boldsymbol{x})$ at expansion point $\boldsymbol{a} = \boldsymbol{x}(k)$

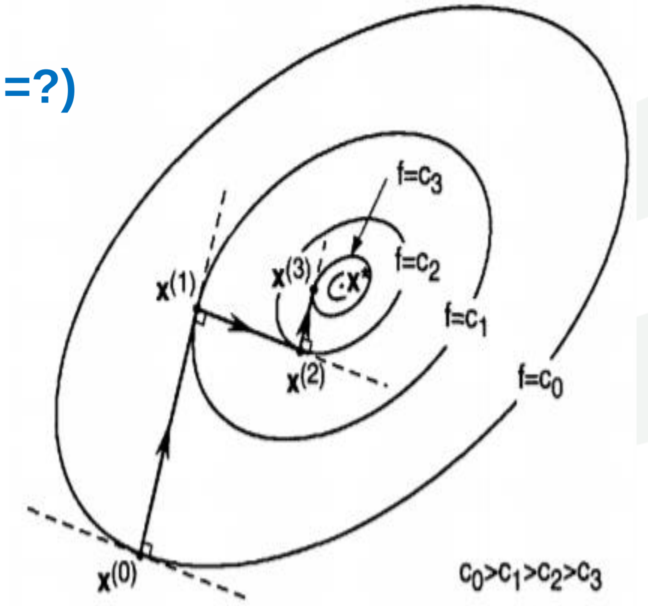
  $$f(\boldsymbol{x}) = f(\boldsymbol{a}) + (\boldsymbol{x}-\boldsymbol{a})^T \cdot \nabla f(\boldsymbol{a}) + (\boldsymbol{x}-\boldsymbol{a})^T \cdot \nabla^2 f(\boldsymbol{a}) \cdot (\boldsymbol{x}-\boldsymbol{a}) + \cdots$$
  $$\approx f(\boldsymbol{a}) + (\boldsymbol{x}-\boldsymbol{a})^T \cdot \nabla f(\boldsymbol{a})$$

  - In $f(\boldsymbol{x})$, let $\boldsymbol{x} = \boldsymbol{x}(k+1)$, then $(\boldsymbol{x}-\boldsymbol{a})^T = \boldsymbol{p}(\boldsymbol{k})$
  - If $\boldsymbol{p}(\boldsymbol{k}) = -\nabla f(\boldsymbol{a})$, we have $(\boldsymbol{x}-\boldsymbol{a})^T \cdot \nabla f(\boldsymbol{a}) < \boldsymbol{0}$ **(why not =?)**
  - We have $f(\boldsymbol{x}(k+1)) \approx f\big(\boldsymbol{x}(k)\big) - |\nabla f(\boldsymbol{a})|^{\boldsymbol{2}}$

- Step length to update search

  $$x(k+1) = x(k) + \underline{a(k)} \times p(k)$$

  Step, $a(k) > 0$



Objective Function in a Special Form of a Sum

- Most machine learning problems consider an objective in a form of a sum

$$f(w) = \frac{1}{N}\sum_{i=1}^{N} f_i(w)$$

where $f_i(w)$ is typically related to the i–th observation in the data set

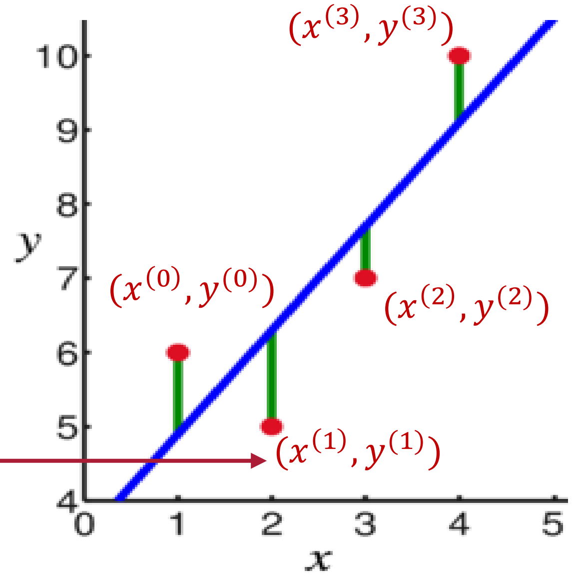
– E.g., regression objective

Slope in Linear regression

Mean Square Error (MSE)

$$\text{Minimize}_w \ f(w) = \frac{1}{2}\sum_{i=1}^{n}\left(\boxed{y^{(i)}} - H\left(\boxed{w}, \boxed{x^{(i)}}\right)\right)^2$$

where n is the number of total observations used for training



- The gradient computation depends on the size of data set N

$$\nabla f(w) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(w)$$

– When N is large (for Big Data application), the **computation cost** becomes too high

Agenda

- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
  - Math
  - A quick review of unconstrained optimization
  - SGD and backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

Stochastic Gradient Descent (SGD)

- Instead of using all data set N to compute the standard (or "batch") gradient descent, SGD uses **samples** of a subset of N summand functions at every iteration

$$\nabla f(w) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(w) \approx \frac{1}{N_m}\sum_{i=1}^{N_m} \nabla f_i(w) \quad \text{with } N_m \ll N$$

"Stochastic" means the algorithm "randomly" samples the data at every iteration

This is called **mini-batch SGD**

- On-line (confusedly, also called **stochastic**) **gradient descent** uses one sample at a time

$$\nabla f(w) \approx \nabla f_i(w)$$

- **Batch Gradient Descent**
  $$N_m = N$$
- **Mini-batch Stochastic GD**
  $$1 < N_m < N$$
- **Stochastic GD**
  $$N_m = 1$$

Training a Neural Network

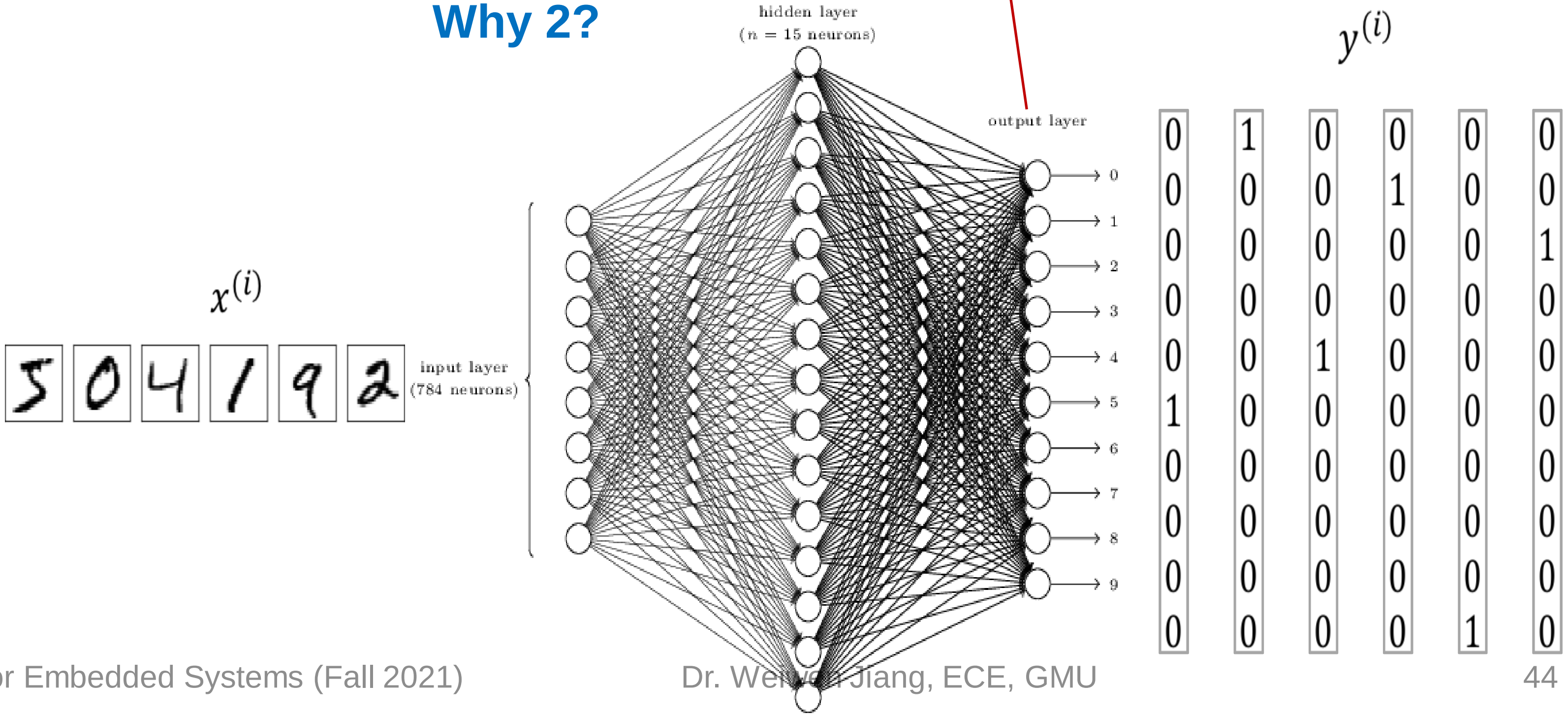
- The model parameters are the weights (and biases) of the model

$$\text{Input to each neuron: } z = w_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} w_i x_i = w^T x$$

- One objective can be to minimize the Sum of **Mean Squares Error (MSE)**

$$\min_w : f(w) = \frac{1}{\boxed{2}N} \sum_{i=1}^{N} \left(y^{(i)} - h\left(w, x^{(i)}\right) \right)^2$$

Why 2?



General Optimization Algorithm

- Specify some initial guess of the solution w(0)
- For k=0,1,…
  - If w(k) is optimal, stop
  - Determine an improved estimate of the solution

$$w(k+1) = w(k) + \alpha(k) * p(k)$$

$\alpha(k)$: *a scalar for step length*
$p(k)$: *a search direction*

- The simplest method with a descent direction defined as

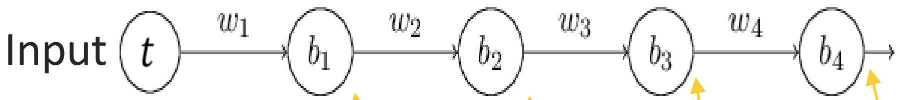
$$p(k) = -\nabla f\big(w(k)\big)$$

- For objective written as a sum of large N terms, SGD is typically used

$$\nabla f(w) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(w) \approx \frac{1}{N_m}\sum_{i=1}^{N_m} \nabla f_i(w)$$

Example of Writing Down the Neuron Network Model

- A simple neuron network with C as our objective (cost) function



$C = C(o_4)$

$o_4 = g(z_4) = g(b_4 + w_4 o_3)$

Activation function: $g(z)$

$o_3 = g(z_3) = g(b_3 + w_3 o_2)$

$o_2 = g(z_2) = g(b_2 + w_2 o_1)$

$o_1 = g(z_1) = g(b_1 + w_1 t)$

$$
\begin{aligned}
C &= C(o_4) \\
o_4 &= g(z_4) = g(b_4 + w_4 o_3) \\
o_3 &= g(z_3) = g(b_3 + w_3 o_2) \\
o_2 &= g(z_2) = g(b_2 + w_2 o_1) \\
o_1 &= g(z_1) = g(b_1 + w_1 t)
\end{aligned}
$$

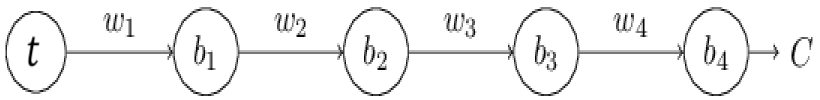
$$C = C(g(b_4 + w_4 g(b_3 + w_3 g(b_2 + w_2 g(b_1 + w_1 t)))))$$

$$C = h(w_1, b_1, w_2, b_2, w_3, b_3, w_4, b_4)$$

Example to Derive the Gradient

Make use of chain-rule

$$C = h(w_1, b_1, w_2, b_2, w_3, b_3, w_4, b_4)$$



$$C = C(o_4)$$
$$o_4 = g(z_4) = g(b_4 + w_4 o_3)$$
$$o_3 = g(z_3) = g(b_3 + w_3 o_2)$$
$$o_2 = g(z_2) = g(b_2 + w_2 o_1)$$
$$o_1 = g(z_1) = g(b_1 + w_1 t)$$

$$\frac{\partial C}{\partial b_4} = \frac{\partial z_4}{\partial b_4}\frac{\partial o_4}{\partial z_4}\frac{\partial C}{\partial o_4} = g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

$$\frac{\partial C}{\partial w_4} = \frac{\partial z_4}{\partial w_4}\frac{\partial o_4}{\partial z_4}\frac{\partial C}{\partial o_4} = g(z_3) g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

$$\frac{\partial C}{\partial b_3} = \frac{\partial z_3}{\partial b_3}\frac{\partial o_3}{\partial z_3}\frac{\partial z_4}{\partial o_3}\frac{\partial o_4}{\partial z_4}\frac{\partial C}{\partial o_4} = g'(z_3) w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

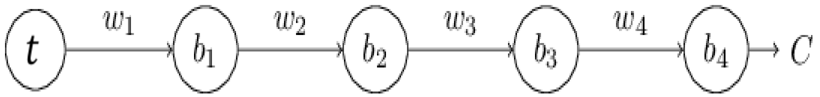
$$\frac{\partial C}{\partial b_2} = \frac{\partial z_2}{\partial b_2}\frac{\partial o_2}{\partial z_2}\frac{\partial z_3}{\partial o_2}\frac{\partial o_3}{\partial z_3}\frac{\partial z_4}{\partial o_3}\frac{\partial o_4}{\partial z_4}\frac{\partial C}{\partial o_4} = g'(z_2) w_3 g'(z_3) w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial z_1}{\partial b_1}\frac{\partial o_1}{\partial z_1}\frac{\partial z_2}{\partial o_1}\frac{\partial o_2}{\partial z_2}\frac{\partial z_3}{\partial o_2}\frac{\partial o_3}{\partial z_3}\frac{\partial z_4}{\partial o_3}\frac{\partial o_4}{\partial z_4}\frac{\partial C}{\partial o_4} = g'(z_1) w_2 g'(z_2) w_3 g'(z_3) w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

Why is it Beneficial to Have a Network Structure?

- A well-organized network structure helps us to "organize" our gradient computation much easier

$$\frac{\partial C}{\partial b_1} = g'(z_1)w_2 g'(z_2)w_3 g'(z_3)w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$



$$\frac{\partial C}{\partial b_4} = g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

$$\frac{\partial C}{\partial b_3} = g'(z_3)w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

$$\frac{\partial C}{\partial b_2} = g'(z_2)w_3 g'(z_3)w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

$$\frac{\partial C}{\partial b_1} = g'(z_1)w_2 g'(z_2)w_3 g'(z_3)w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

Backpropagation

- A fancy way to say "gradient computation for neural networks"
- Find out how changing the weights and bias (model parameters) changes the cost function

  We will skip the formal proof here

$$t \xrightarrow{w_1} b_1 \xrightarrow{w_2} b_2 \xrightarrow{w_3} b_3 \xrightarrow{w_4} b_4 \rightarrow C$$

$$\frac{\partial C}{\partial b_4} = g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

$$\frac{\partial C}{\partial b_3} = g'(z_3) w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

$$\frac{\partial C}{\partial b_2} = g'(z_2) w_3 g'(z_3) w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

$$\frac{\partial C}{\partial b_1} = g'(z_1) w_2 g'(z_2) w_3 g'(z_3) w_4 g'(z_4) \cdot \frac{\partial C}{\partial o_4}$$

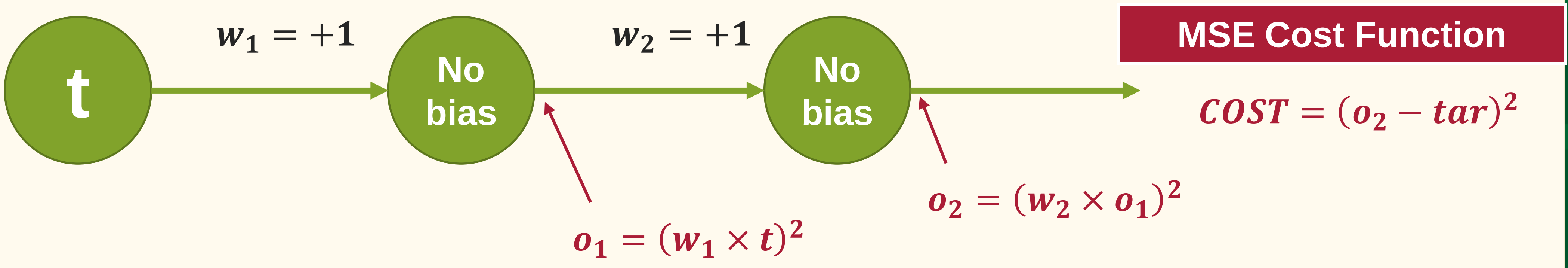
Properties of Backpropagation

- Requirements for **cost functions** for backpropagation
  - Cost can be written as a function of the outputs from the neural network
  - Cost is average cost over all individual training examples

$$\min_w: f(w) = \frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - h\left(w, x^{(i)}\right)\right)^2$$

- Backpropagation **simultaneously compute** all the **partial derivatives** using just **one forward pass** through the network, followed by **one backward pass** through the network.
  - Roughly speaking, the computational cost of the backward pass is about the same as the forward pass
- In other words, the backpropagation algorithm is a clever way of keeping track of **small perturbations** to the weights (and biases) as they propagate through the network, reach the output, and then affect the cost

Example
Pytorch Implementation of Backpropagation



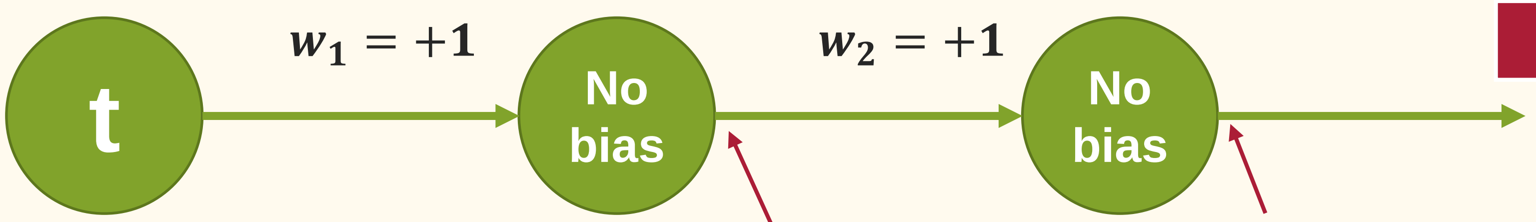
Given:

- Input $t = 0.5$
- Weight $w_1 = +1.0$
- Weight $w_2 = +1.0$
- Target $tar = +0.3125$
- Learning rate $lr = +0.1$

Calculate:

- Output o_1, o_2
- Loss/cost $COST$
- Gradient $grad_1$ of w_1
- Gradient $grad_2$ of w_2
- Updated weight w_1, w_2

Example
Pytorch Implementation of Backpropagation



MSE Cost Function

$COST = (o_2 - tar)^2$

$o_1 = (w_1 \times t)^2$

$o_2 = (w_2 \times o_1)^2$

Given:

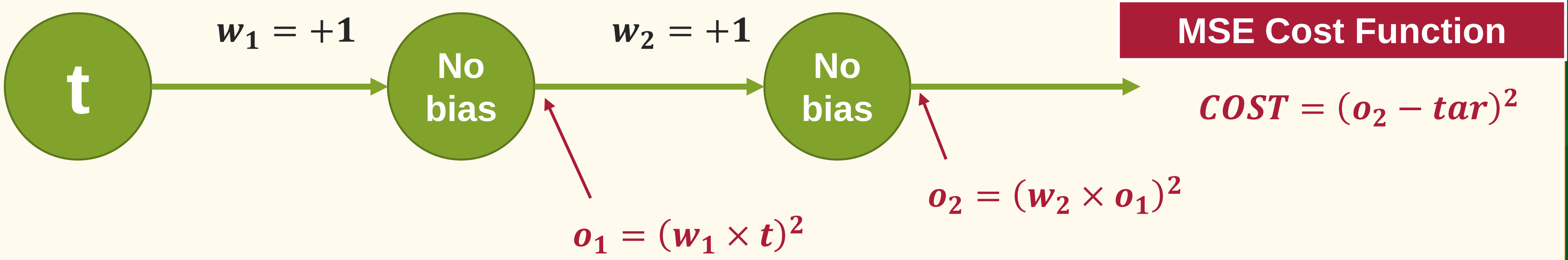
- Input $t = 0.5$
- Weight $w_1 = +1.0$
- Weight $w_2 = +1.0$
- Target $tar = +0.3125$
- Learning rate $lr = +0.1$

Calculate:

- Output o_1, o_2
- Loss/cost $COST$
- Gradient $grad_1$ of w_1
- Gradient $grad_2$ of w_2
- Updated weight w_1, w_2

Example

Pytorch Implementation of Backpropagation




```python
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self):
        """Initialize the layers of the model."""
        super(MLP, self).__init__()
        self.layer1 = nn.Linear(1,1,bias=False)
        self.layer2 = nn.Linear(1,1,bias=False)
    def forward(self,x):
        x = self.layer1(x)
        x = x.pow(2)
        x = self.layer2(x)
        x = x.pow(2)
        return x
```

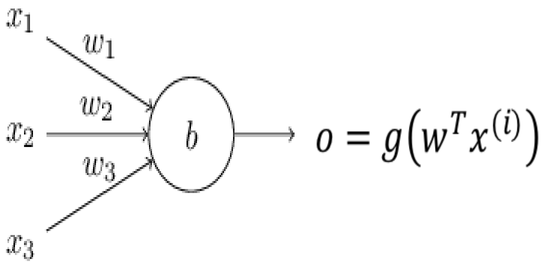
```python
chain_net = MLP()

w1 = +1.0; w2 = +1.0
nn.init.constant_(chain_net.layer1.weight, w1)
nn.init.constant_(chain_net.layer2.weight, w2)

input = torch.Tensor([0.5])
target = torch.Tensor([0.3125])
mse_loss = nn.MSELoss()
optimizer = torch.optim.SGD(chain_net.parameters
(), lr=0.1)

output =  chain_net(input)
loss = mse_loss(output, target)
loss.backward()
optimizer.step()
```

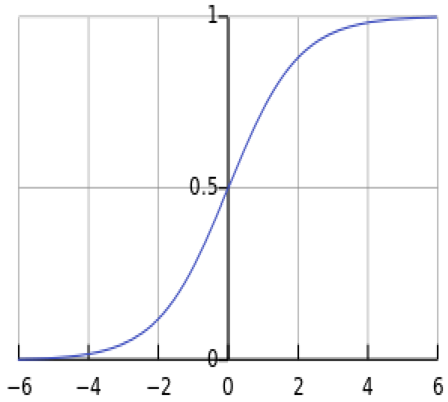
Agenda

- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
  - Math
  - A quick review of unconstrained optimization
  - SGD and backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

Cross-Entropy Cost Function for Single Neuron

- The cost function we used for logistic regression





$$\min_{w}: \ f(w) = \frac{1}{N}\sum_{i=1}^{N}\left[-y^{(i)} log\left(g\left(w^T x^{(i)}\right)\right) - \left(1 - y^{(i)}\right) log\left(1 - g\left(w^T x^{(i)}\right)\right)\right]$$

We derived this cost function based on maximum likelihood

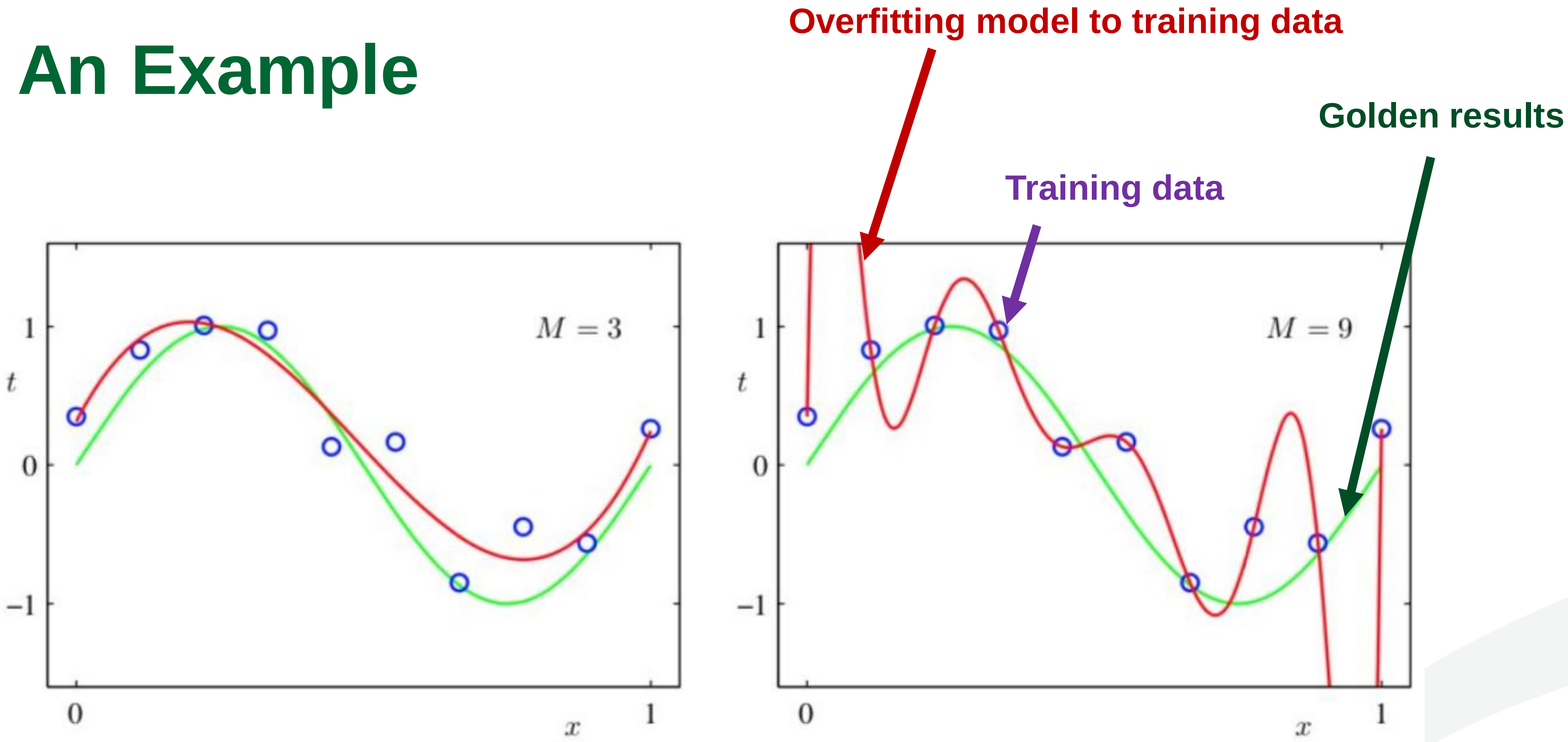
- This cost function is also called cross-entropy function

$$p(k)_j = -\frac{\partial f}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - g\left(w^T x^{(i)}\right)\right) x_j^{(i)}$$

Agenda

- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
  - Math
  - A quick review of unconstrained optimization
  - SGD and backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

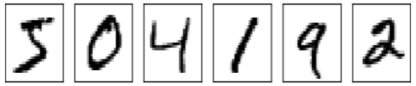
An Example


Overfitting model to training data
Golden results
Training data
M = 3
M = 9
t
x
1
0
−1
0
1

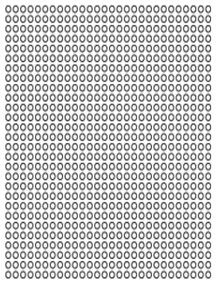
Another Example

- Neuron network with 30 hidden neurons for MINIST digital recognition

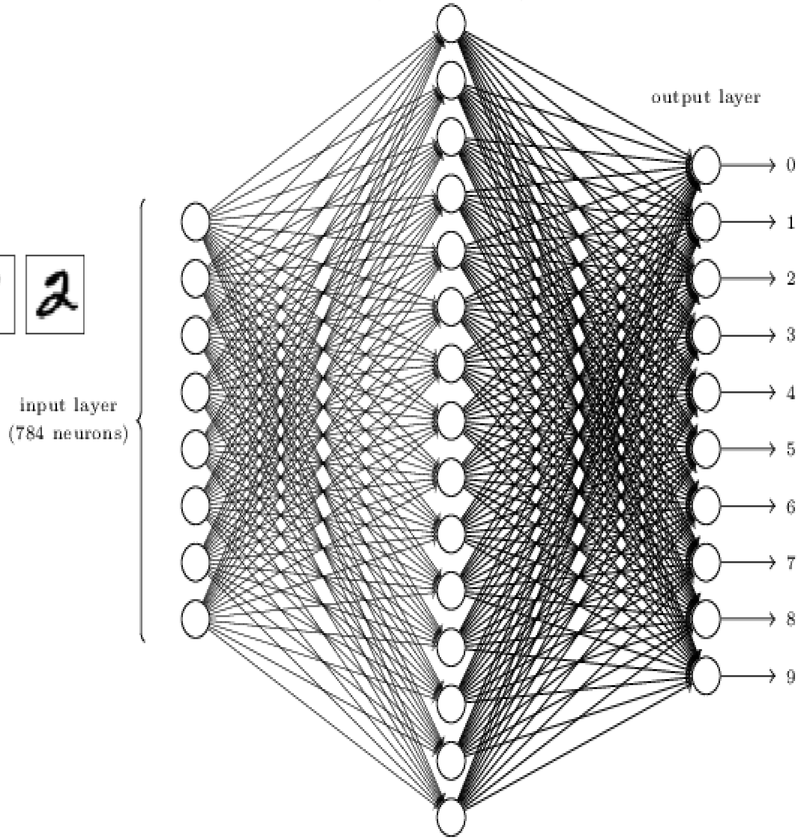
$x^{(i)}$



28x28 pixels



Hidden layer
(n = 30 neurons)



$y^{(i)}$

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

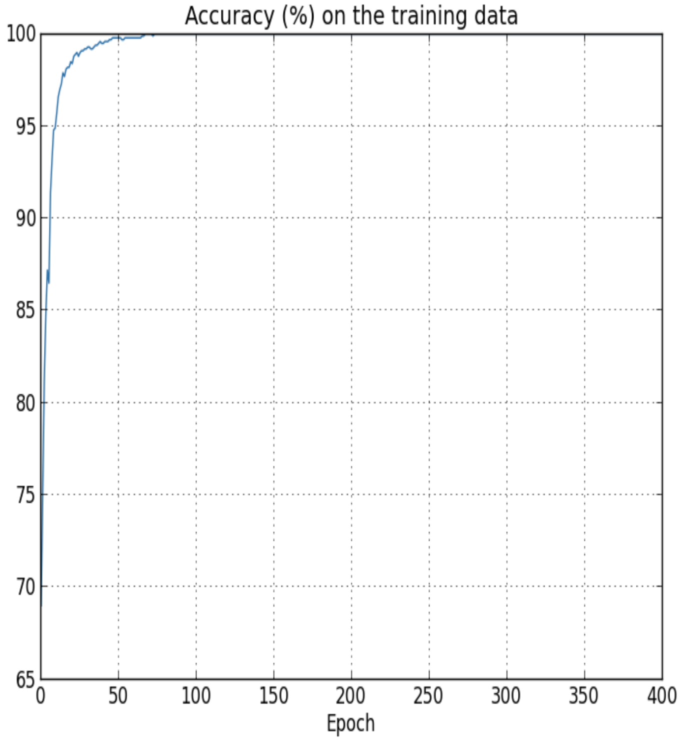
Issues of Overfitting

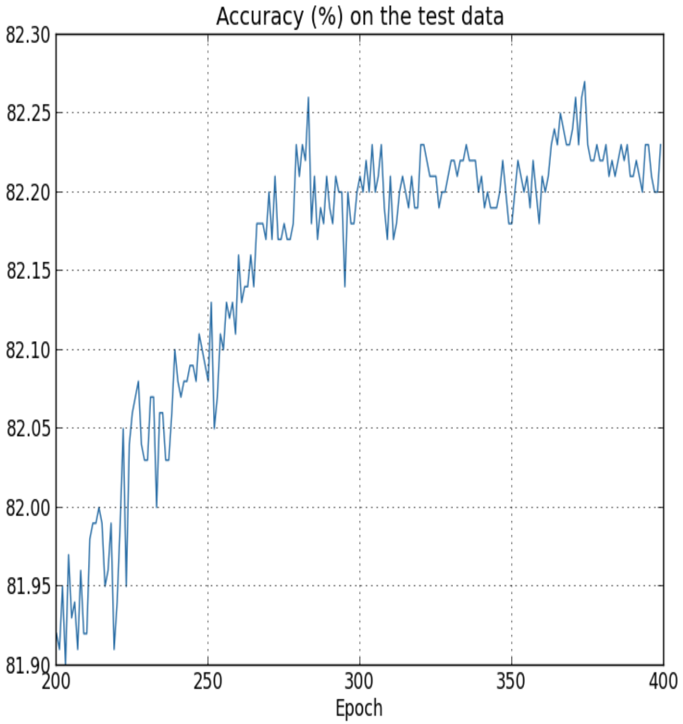
- Number of model parameters

  [(28*28)*30 + 30] + [30*10 + 10] = 23,860

- Train the network with 1,000 training images

  The # of parameters >> the # of training data





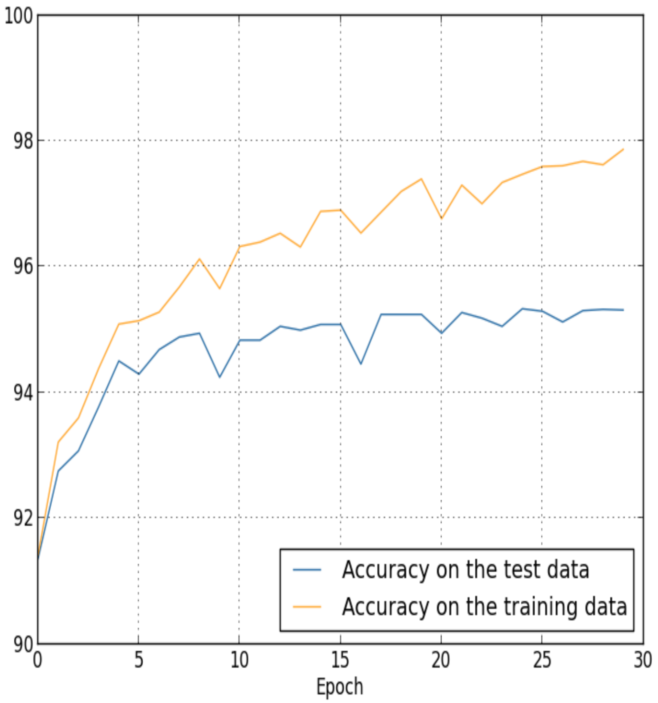
Issues of Overfitting

- Number of model parameters

  [(28*28)*30 + 30] + [30*10 + 10] = 23,860

- Train the network with 50,000 training images

  The # of parameters < the # of training data



Agenda

- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
  - Math
  - A quick review of unconstrained optimization
  - SGD and backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

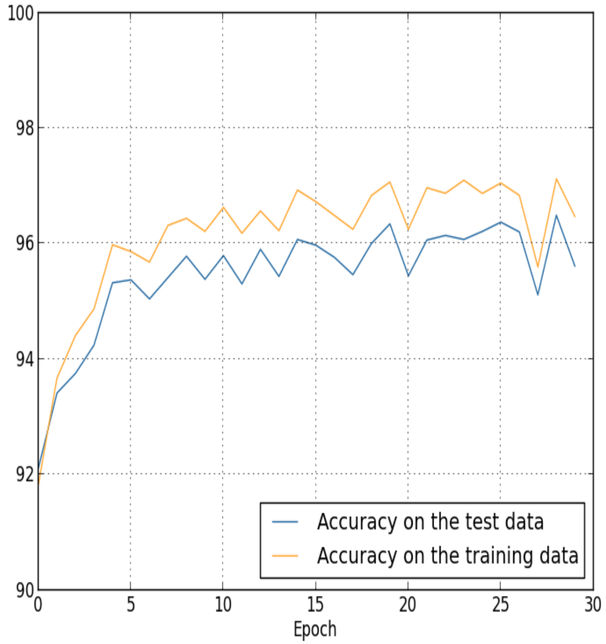
Overfitting and Regularization

- Overfitting is a major problem in neural networks

  This is especially true in modern networks, which often have **very large numbers of weights and biases**

- Regularization helps to overcome overfitting

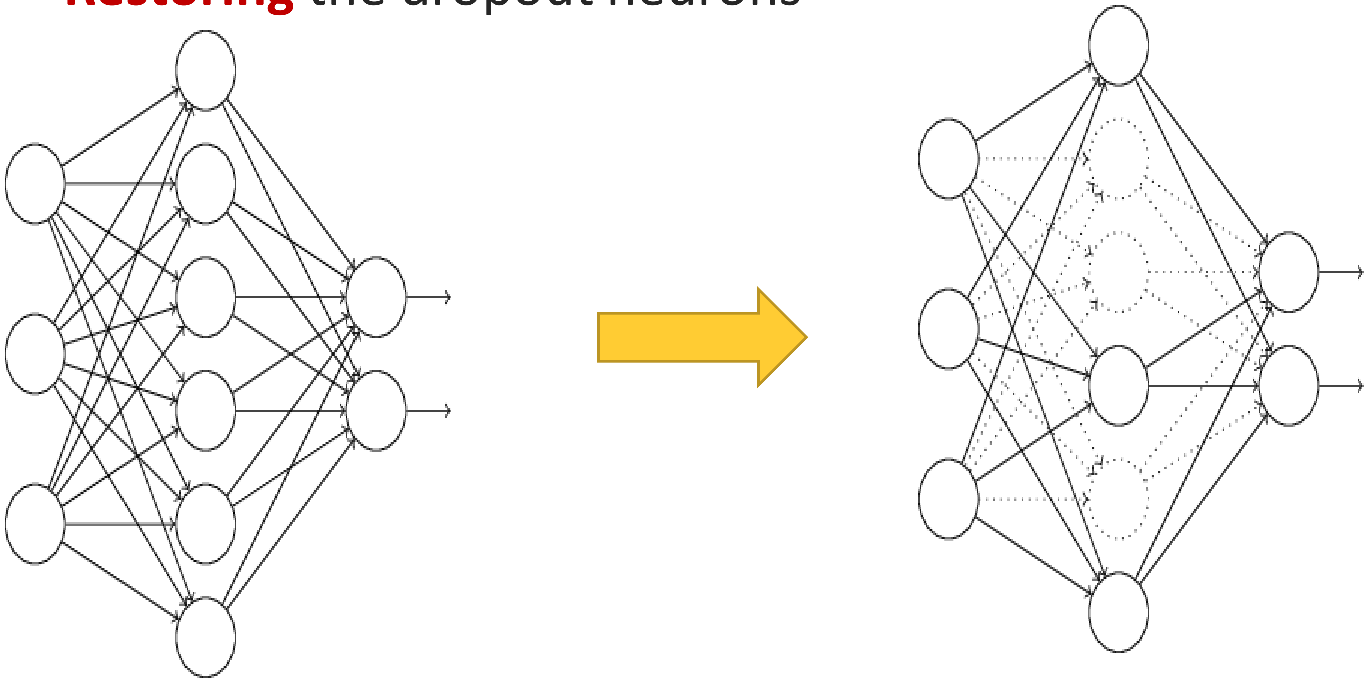
$$\min_{w}: \ f(w) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K}\left[-y^{(i)} log\left(g_j\left(w^T x^{(i)}\right)\right) - \left(1 - y^{(i)}\right) log\left(1 - g_j\left(w^T x^{(i)}\right)\right)\right] + \boxed{\frac{\lambda}{2N}\sum_{k} w_k^2}$$



Enforce weights to approach 0 to avoid overfitting

Dropout as Another Regularization Technique for Neural Networks

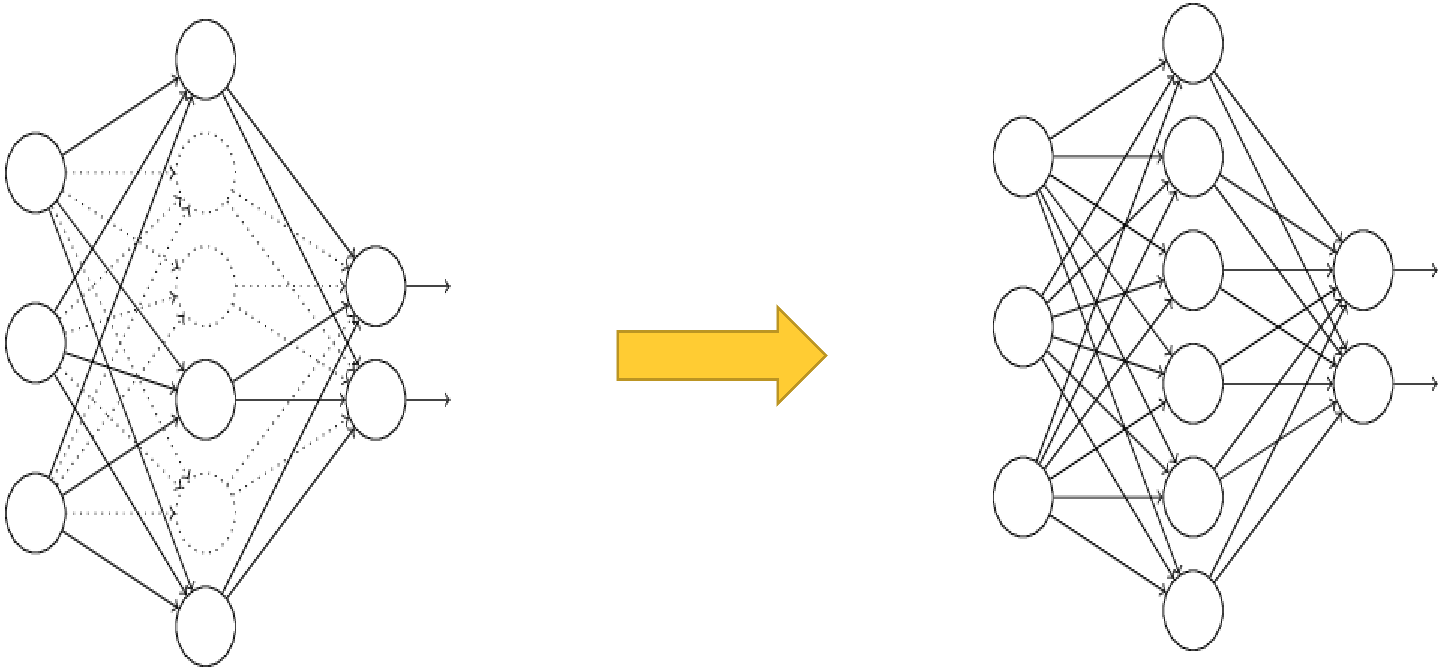
- Dropout does not modify the cost function, but modify the network itself
  - For each iteration over a mini-batch
    - We **randomly (but temporarily) delete or ignore half the hidden neurons** in the network, while leaving the input and output neurons untouched
    - Forward-propagate the input through the **modified network**, and then backpropagate the result, also through the **modified network**
    - **Update** the appropriate weights and biases.
    - **Restoring** the dropout neurons



Dropout as Another Regularization Technique for Neural Networks

- When we actually run the full network (inference), twice as many hidden neurons will be active

  To compensate for that, we halve the weights outgoing from the hidden neurons

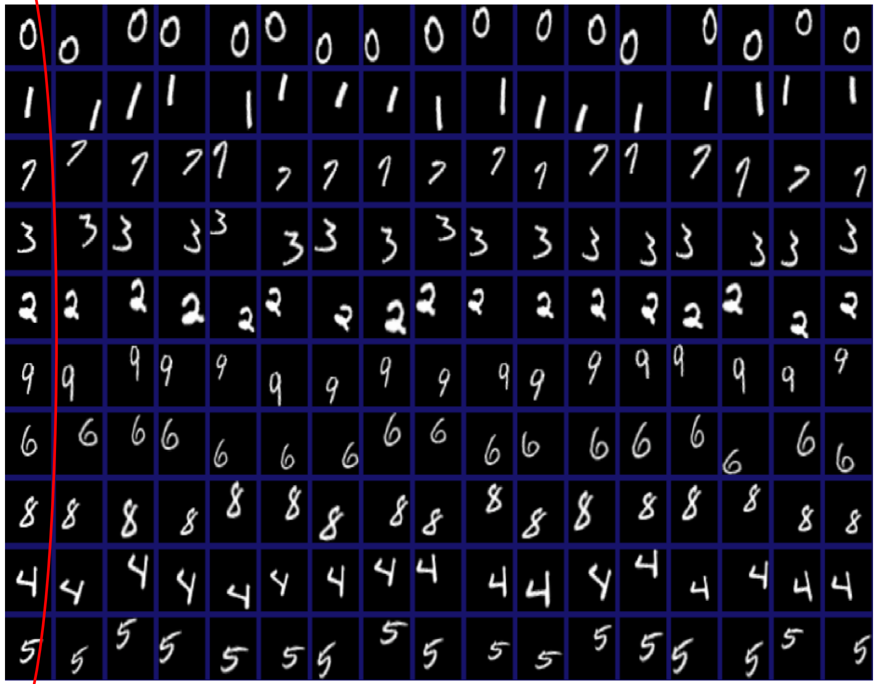


- Intuitively, why dropout may work?

  It's like training different networks (with reduced parameters) and then take an averaging from them

Artificially expanding the training data as another regularization technique

- One reason for overfitting is the **lack of training data** → so why not creating more training data
  - –**Obtaining labeled data** is more difficult (too costly)
  - –So **slightly modifying existing labeled data** to create new data with the same label



The original MNIST digits

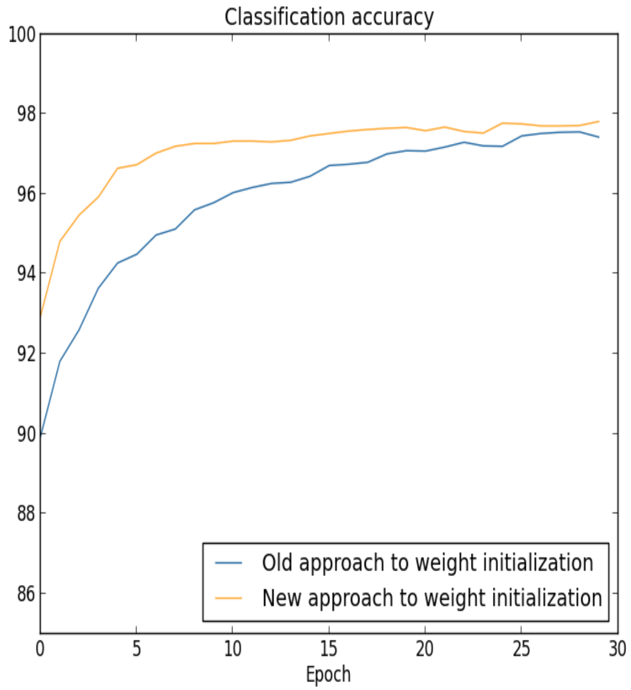
[http://www.cs.toronto.edu/~tijmen/affNIST/]

Agenda

- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
  - Math
  - A quick review of unconstrained optimization
  - SGD and backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

Parameter Initialization

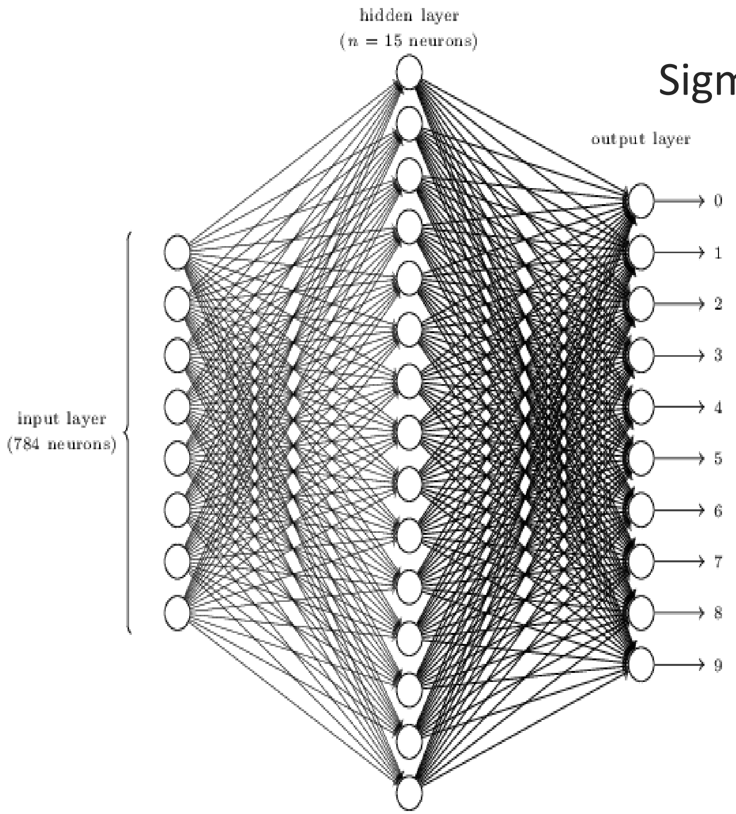
- Straightforward strategy: **randomly generate values** based on independent Gaussian random variables, normalized to have mean 0 and standard deviation 1
- It turns out different strategies for initialization can lead to different results with different quality



- **How to best define initialization strategy is still an open research problem**

Softmax Layer to Transform Outputs into a Probability Distribution

- For multi-class neural network, the **output class is determined by the maximum value among all output neurons** (after the forward propagation)
- Softmax layer transform the outputs into a **probability distribution**
  – The one with the **highest probability** determines the class

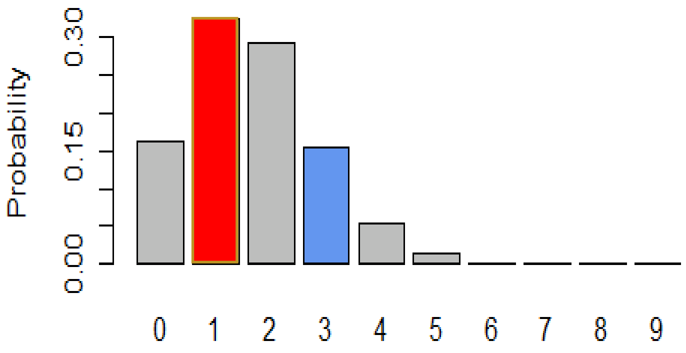


Sigmoid neuron output

$$o_1 = g(z_1)$$
$$o_2 = g(z_2)$$
$$o_3 = g(z_3)$$
$$o_{10} = g(z_{10})$$

Softmax neuron output

$$o_{j,new} = \frac{e^{z_j}}{\sum_{j=1}^{K} e^{z_j}}$$



Why SOFTMAX?

Because:

- Label is a "probability distribution"
- For better formulate cost in training

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

Agenda

- Artificial neuron network for multi-class classification
- Inference: forward propagation
- Training: backpropagation
  - Math
  - A quick review of unconstrained optimization
  - SGD and backpropagation
- Cross-entropy function as objective
- Overfitting issues of neural network
- Regularization techniques to overcome overfitting
- Initialization and softmax output layer
- Conclusions

Conclusions

- Artificial neural network is a machine learning model (in part) inspired by the neuroscience's understanding of biological neurons
  Multi-class classification can be easily formulated as an ANN problem

- **Forward propagation** and **backpropagation in SGD** are key algorithms for ANN

- **Overfitting is a major issues** for ANN training, and a number of regularization techniques are developed to address this issue

- ANN has been around for many years, and it only recently **gained popularity**

Lab 1: Introducing Yourself and Implementing XOR using MLP on Colab

Assignments and Related Documents:

- https://go.gmu.edu/ml4emb

Due Date: **This Friday** (09/03/2021) by 1 PM

- Please take this chance to evaluate the required programming background and the required bandwidth to decide whether keep or drop this course.

GMU.EDU

George Mason University

4400 University Drive
Fairfax, Virginia 22030

Tel: (703)993-1000