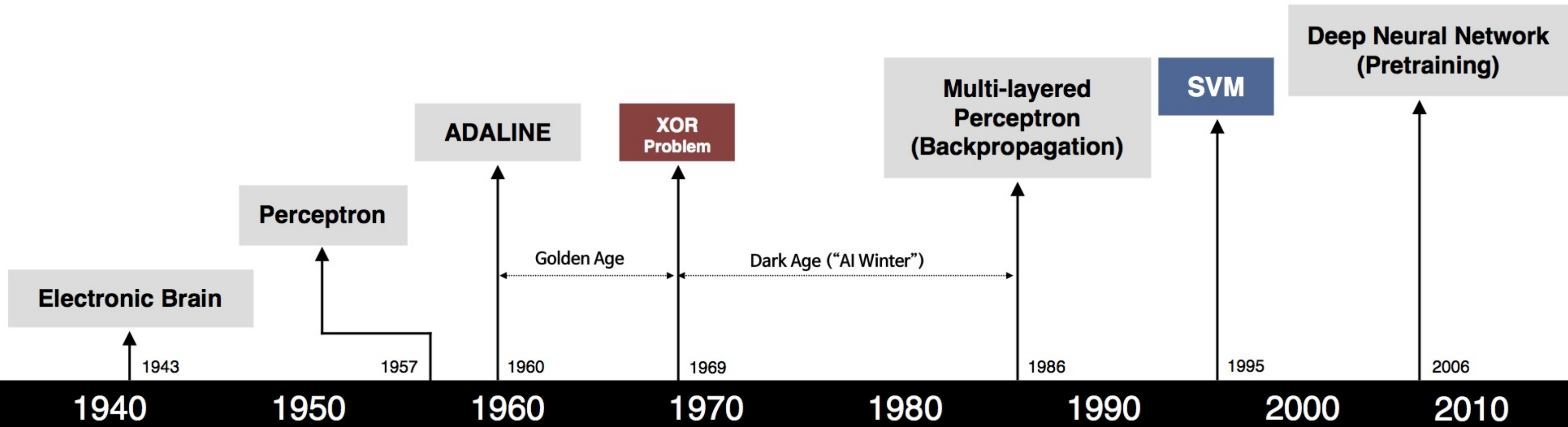
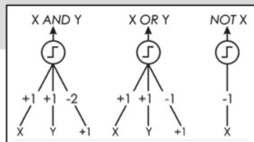
From Perceptrons to Deep Feed Forward Networks

ε.δε.μ^2

ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ & ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ



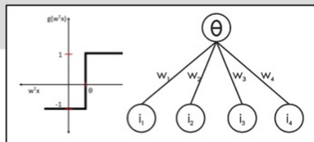
S. McCulloch – W. Pitts



- Adjustable Weights
- Weights are not Learned

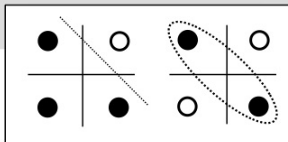
F. Rosenblatt

B. Widrow – M. Hoff



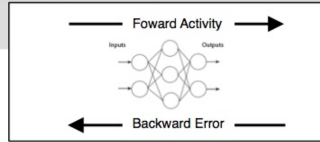
- Learnable Weights and Threshold

M. Minsky – S. Papert



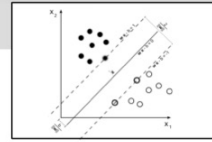
- XOR Problem

D. Rumelhart – G. Hinton – R. Wiliams



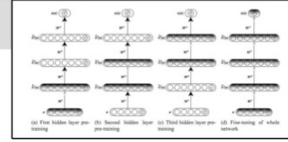
- Solution to nonlinearly separable problems
- Big computation, local optima and overfitting

V. Vapnik – C. Cortes



- Limitations of learning prior knowledge
- Kernel function: Human Intervention

G. Hinton – S. Ruslan



- Hierarchical feature Learning

NY Times 1960 article!!

NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI)—The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

In today's demonstration, the "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.

Training single-layer perceptron

How to determine MLP parameters

- MLPs can realize logical connectives
  - We crafted parameters (weights and biases) carefully to realize desired connectives

- However, crafting parameters is difficult
  - We are sometimes unsure of the internal logic associating input and output variables

- Find parameters automatically from data
  - We are interested in determining parameters from data describing pairs of inputs and outputs

Supervised learning (training)

- We have a supervision data
  - $D = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_N, y_N)\}$ (N instances)

- Find parameters such that they can predict training instances as correctly as possible

- We assume *generalization*
  - If the parameters predict training instances well, they will work for unseen instances

Supervised learning for single-layer NNs

- For simplicity, we include a bias term b in $\boldsymbol{w}$ hereafter
  - Redefine $\boldsymbol{x}^{(\mathrm{new})} = (x_1, x_2, \dots, x_d, 1)^\intercal$, $\boldsymbol{w}^{(\mathrm{new})} = (w_1, w_2, \dots, w_\mathrm{d}, b)^\intercal$
  - Then, $\boldsymbol{w}^{(\mathrm{new})} \cdot \boldsymbol{x}^{(\mathrm{new})} = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b$ (original form)

- We introduce a new notation to distinguish a computed output $\hat{y}$ from the gold output y in the supervision data
  - $D = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_N, y_N)\}$ (N instances)
  - We distinguish two kinds of outputs hereafter
    - $\hat{y}$: the output computed (predicted) by the model (perceptron) for the input
    - y: the true (gold) output for the input in the supervision data

- Training: find $\boldsymbol{w}$ such that,

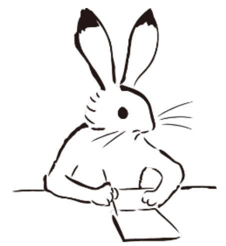
$$\forall n \in \{1, \dots, N\}: g(\boldsymbol{w} \cdot \boldsymbol{x}_n) = y_n$$

Perceptron algorithm (Rosenblatt, 1957)

1. $\boldsymbol{w} = 0$
2. Repeat:
3. $\quad (\boldsymbol{x}_n, y_n) \leftarrow$ a random sample from D
4. $\quad \hat{y} \leftarrow g(\boldsymbol{w} \cdot \boldsymbol{x_n})$
5. $\quad$ if $\hat{y} \neq y_n$ then:
6. $\quad\quad$ if $y_n = 1$ then:
7. $\quad\quad\quad \boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \boldsymbol{x}_n$
8. $\quad\quad$ else:
9. $\quad\quad\quad \boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{x}_n$
10. Until no instance updates $\boldsymbol{w}$

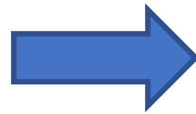
η $(0 < \eta)$ is the learning rate

Exercise: Train an SLP to realize OR



- Convert the truth table into training data

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



$$D = \left\{ \begin{array}{l} \left((0\ 0\ 1)^\intercal, 0\right), \\ \left((0\ 1\ 1)^\intercal, 1\right), \\ \left((1\ 0\ 1)^\intercal, 1\right), \\ \left((1\ 1\ 1)^\intercal, 1\right) \end{array} \right\}$$

- Initialize the weight vector $\boldsymbol{w} = 0$
- Apply the perceptron algorithm to find $\boldsymbol{w}$
  - Fix $\eta = 1$ in the exercise

Updating weights for OR

- Data: $D = \{((0\ 0\ 1)^\top, 0), ((0\ 1\ 1)^\top, 1), ((1\ 0\ 1)^\top, 1), ((1\ 1\ 1)^\top, 1)\}$
- Initialization: $\boldsymbol{w} = (0\ 0\ 0)^\top$
- Iteration #1: choose $(\boldsymbol{x}_4, y_4) = ((1\ 1\ 1)^\top, 1)$
  - Classification: $\hat{y} = g(\boldsymbol{w} \cdot \boldsymbol{x}_4) = g(0) = 0 \neq y_4$
  - Update: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{x}_4 = (1\ 1\ 1)^\top$
- Iteration #2: choose $(\boldsymbol{x}_1, y_1) = ((0\ 0\ 1)^\top, 0)$
  - Classification: $\hat{y} = g(\boldsymbol{w} \cdot \boldsymbol{x}_1) = g(1) = 1 \neq y_1$
  - Update: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \boldsymbol{x}_1 = (1\ 1\ 0)^\top$

We chose the instances in the order that minimizes the required number of updates

- Terminate (the weight $\boldsymbol{w}$ classifies all instances correctly)
  - $\boldsymbol{x} = (0\ 0\ 1)^\top$: $y = g((1\ 1\ 0)(0\ 0\ 1)^\top) = 0$
  - $\boldsymbol{x} = (0\ 1\ 1)^\top$: $y = g((1\ 1\ 0)(0\ 1\ 1)^\top) = 1$
  - $\boldsymbol{x} = (1\ 0\ 1)^\top$: $y = g((1\ 1\ 0)(1\ 0\ 1)^\top) = 1$
  - $\boldsymbol{x} = (1\ 1\ 1)^\top$: $y = g((1\ 1\ 0)(1\ 1\ 1)^\top) = 1$

Why perceptron algorithm learns

- Suppose the parameter $\boldsymbol{w}$ misclassifies $(\boldsymbol{x}_n, y_n)$
  - If $y_n = 1$ then:
    - Update the weight vector $\boldsymbol{w}' \leftarrow \boldsymbol{w} + \boldsymbol{x}_n$
    - If we classify $\boldsymbol{x}_n$ again with the updated weights $\boldsymbol{w}'$:
      - $\boldsymbol{w}' \cdot \boldsymbol{x}_n = (\boldsymbol{w} + \boldsymbol{x}_n) \cdot \boldsymbol{x}_n = \boldsymbol{w} \cdot \boldsymbol{x}_n + \boldsymbol{x}_n \cdot \boldsymbol{x}_n \geq \boldsymbol{w} \cdot \boldsymbol{x}_n$
      - The dot product was increased (more likely to be classified as 1)
  - If $y_n = 0$ then:
    - Update the weight vector $\boldsymbol{w}' \leftarrow \boldsymbol{w} - \boldsymbol{x}_n$
    - If we classify $\boldsymbol{x}_n$ again with the updated weights $\boldsymbol{w}'$:
      - $\boldsymbol{w}' \cdot \boldsymbol{x}_n = (\boldsymbol{w} - \boldsymbol{x}_n) \cdot \boldsymbol{x}_n = \boldsymbol{w} \cdot \boldsymbol{x}_n - \boldsymbol{x}_n \cdot \boldsymbol{x}_n \leq \boldsymbol{w} \cdot \boldsymbol{x}_n$
      - The dot product was decreased (more likely to be classified as 0)

- The algorithm updates the parameter $\boldsymbol{w}$ to the direction where it will classify $(\boldsymbol{x}_n, y_n)$ more correctly

Extending the algorithm to MLPs

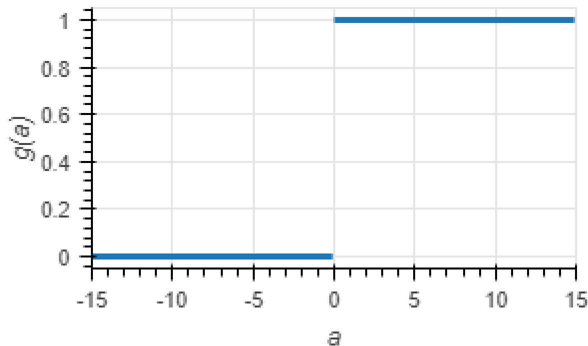
- The perceptron algorithm:
  - Can find SLP parameters for linearly-separable data
  - Does not terminate with linearly-inseparable data
    - This is because of the limitation of SLPs
    - We must force to terminate the algorithm with incomplete parameters

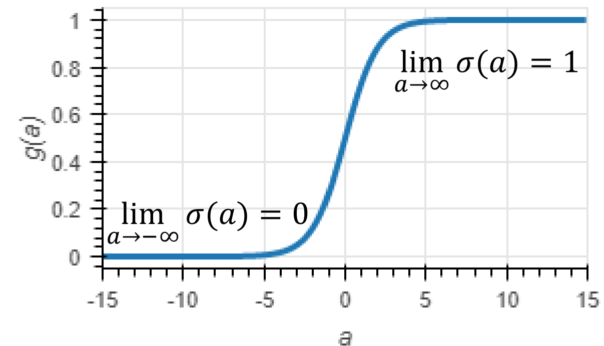
- Extending the algorithm to MLPs is non trivial
  - We have no training data for hidden states
  - The famous argument of Minsky and Papert (1969)

- Two new ideas: *sigmoid* and *backpropagation*

Single layer perceptron with sigmoid function

Activation function: step → sigmoid





Step function: $\mathbb{R} \to \{0,1\}$

$$g(a) = \begin{cases} 1 & (\text{if } a > 0) \\ 0 & (\text{otherwise}) \end{cases}$$

- Yields binary outputs
  - Unusable for multi-class classification
- Indifferentiable at zero
- With zero gradients

Sigmoid function: $\mathbb{R} \to (0,1)$

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

- Yields continuous scores
  - Usable for multi-class classification
- Differentiable at all points
- With mostly non-zero gradients
  - Useful for gradient descent

General form with sigmoid

- Single layer NN with sigmoid function

$$\hat{y} = \sigma(\boldsymbol{w} \cdot \boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w} \cdot \boldsymbol{x}}}$$

  - Given an input $\boldsymbol{x} \in \mathbb{R}^d$, it computes an output $\hat{y} \in (0,1)$ by using the parameter $\boldsymbol{w} \in \mathbb{R}^d$

- This is also known as *logistic regression*
  - We can interpret $\hat{y}$ as the conditional probability $P(1|\boldsymbol{x})$ where an input is classified to 1 (positive category)
  - Rule to classify an input to 1:

$$\hat{y} > 0.5 \iff \frac{1}{1 + e^{-\boldsymbol{w} \cdot \boldsymbol{x}}} > \frac{1}{2} \iff \boldsymbol{w} \cdot \boldsymbol{x} > 0$$

    - The classification rule is the same as the linear models

Example: logical AND

- The same parameter in the previous example

$$\hat{y} = \sigma(a), a = x_1 + x_2 - 1.5$$

x_1	x_2	$\boldsymbol{y} = x_1 \land x_2$	a	$\hat{y} = \sigma(a)$
0	0	0	-1.5	0.182
0	1	0	-0.5	0.378
1	0	0	-0.5	0.378
1	1	1	0.5	0.622

- The outputs are acceptable, but
  - $P(x_1 \land x_2 = 1 | x_1 = 1, x_2 = 1)$ is not so high (62.2%)
  - Room for improving $\boldsymbol{w}$ so that it yields $y \rightarrow 1$ (100%) for positives (true) and $y \rightarrow 1$ (0%) for negatives (false)

Instance-wise likelihood

- We introduce *instance-wise likelihood*, to measure how well the parameters reproduce $(\boldsymbol{x}_n, y_n)$

$$p_n = \begin{cases} \hat{y}_n & (\text{if } y_n = 1) \\ (1 - \hat{y}_n) & (\text{otherwise}) \end{cases}$$

x_1	x_2	$y = x_1 \wedge x_2$	a	$\hat{y} = \sigma(a)$	p	
0	0	0	-1.5	0.182	$1 - \hat{y} = 0.818$	⇨ 1
0	1	0	-0.5	0.378	$1 - \hat{y} = 0.622$	⇨ 1
1	0	0	-0.5	0.378	$1 - \hat{y} = 0.622$	⇨ 1
1	1	1	0.5	0.622	$\hat{y} = 0.622$	⇨ 1

Parameters of AND: $\hat{y} = \sigma(a), a = x_1 + x_2 - 1.5$

- Likelihood is a probability representing the 'fitness' of the parameters to the training data
  - We want to increase the likelihood by changing $\boldsymbol{w}$

Likelihood on the training data

- We assume that all instances in the training data are i.i.d. (independent and identically distributed)
- We define *likelihood* as a joint probability on data,

$$L_D(\boldsymbol{w}) = \prod_{n=1}^{N} p_n$$

  - When the training data $D = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_N, y_N)\}$ is fixed, likelihood is a function of the parameters $\boldsymbol{w}$

- Let us maximize $L_D(\boldsymbol{w})$ by changing $\boldsymbol{w}$
  - This is called *Maximum Likelihood Estimation (MLE)*
  - The maximizer $\boldsymbol{w}^*$ reproduces the training data well

Training as a minimization problem

- Products of $(0,1)$ values often cause underflow
- Use *log-likelihood*, the logarithm of the likelihood, instead

$$LL_D(\boldsymbol{w}) = \log L_D(\boldsymbol{w}) = \log \prod_{n=1}^{N} p_n = \sum_{n=1}^{N} \log p_n$$

- In mathematical optimization, we usually consider a minimization problem instead of maximization
- We define an objective function $E_D(\boldsymbol{w})$ by using the negative of the log-likelihood

$$E_D(\boldsymbol{w}) = -LL_D(\boldsymbol{w}) = -\sum_{n=1}^{N} \log p_n$$

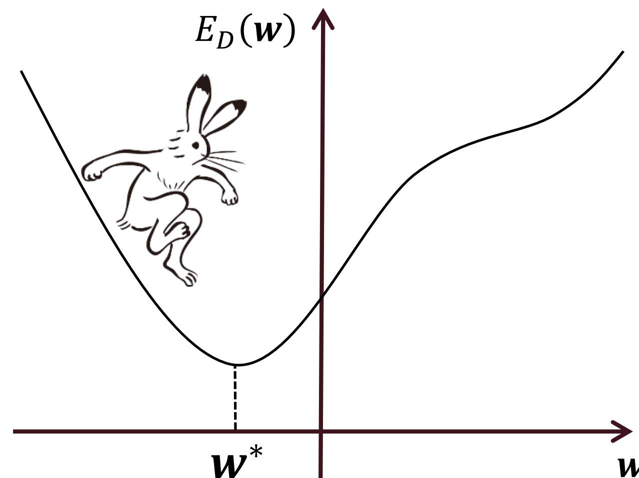
  - $E_D(\boldsymbol{w})$ is called a *loss function* or *error function*

Training as a minimization problem

- Given the training data $D = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_N, y_N)\}$, find $\boldsymbol{w}^*$ as the minimization problem,

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}}\, E_D(\boldsymbol{w}) = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{n=1}^{N} l_n\,,$$

$$l_n = -\log p_n = \begin{cases} -\log \hat{y}_n & (\text{if } y_n = 1) \\ -\log(1 - \hat{y}_n) & (\text{otherwise}) \end{cases} = -y_n \log \hat{y}_n - (1 - y_n) \log(1 - \hat{y}_n)$$



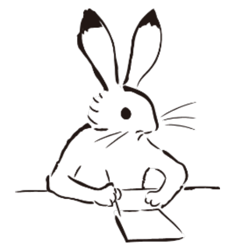
Stochastic Gradient Descent (SGD)

- The objective function is the sum of losses of instances,

$$E_D(\boldsymbol{w}) = \sum_{n=1}^{N} l_n$$

- We can use Stochastic Gradient Descent (SGD) and its variants (e.g., Adam) for minimizing $E_D(\boldsymbol{w})$
- SGD Algorithm (T is the number of updates)
  1. Initialize $\boldsymbol{w}$ with random values
  2. for $t \longleftarrow 1$ to T:
  3. $\quad \eta_t \longleftarrow 1/t$
  4. $\quad (\boldsymbol{x}_n, y_n) \longleftarrow$ a random sample from D
  5. $\quad \boldsymbol{w} \longleftarrow \boldsymbol{w} - \eta_t \frac{\partial l_n}{\partial \boldsymbol{w}}$

Exercise: compute the gradient



- Prove:

$$\frac{\partial l_n}{\partial \boldsymbol{w}} = \frac{\partial l_n}{\partial \hat{y}_n}\frac{\partial \hat{y}_n}{\partial a_n}\frac{\partial a_n}{\partial \boldsymbol{w}} = (\hat{y}_n - y_n)\boldsymbol{x}_n$$

by computing the gradients $\frac{\partial l_n}{\partial \hat{y}_n}, \frac{\partial \hat{y}_n}{\partial a_n}, \frac{\partial a_n}{\partial \boldsymbol{w}}$

- Here:
  - $l_n = -y_n \log \hat{y}_n - (1 - y_n) \log(1 - \hat{y}_n)$,
  - $\hat{y}_n = \sigma(a_n) = \frac{1}{1+e^{-a_n}}$,
  - $a_n = \boldsymbol{w} \cdot \boldsymbol{x}_n$

Answer: compute the gradients

$$\frac{\partial l_n}{\partial \hat{y}_n} = -\frac{y_n}{\hat{y}_n} - \frac{1 - y_n}{1 - \hat{y}_n} \cdot (-1) = \frac{-y_n(1 - \hat{y}_n) + \hat{y}_n(1 - y_n)}{\hat{y}_n(1 - \hat{y}_n)} = \frac{\hat{y}_n - y_n}{\hat{y}_n(1 - \hat{y}_n)},$$

$$\frac{\partial \hat{y}_n}{\partial a_n} = (-1) \cdot \frac{1}{\{1 + e^{-a_n}\}^2} \cdot e^{-a_n} \cdot (-1) = \frac{1}{1 + e^{-a_n}} \cdot \frac{e^{-a_n}}{1 + e^{-a_n}} = \hat{y}_n(1 - \hat{y}_n),$$

$$\frac{\partial a_n}{\partial \boldsymbol{w}} = \boldsymbol{x}_n$$

Therefore,

$$\frac{\partial l_n}{\partial \boldsymbol{w}} = \frac{\partial l_n}{\partial \hat{y}_n} \frac{\partial \hat{y}_n}{\partial a_n} \frac{\partial a_n}{\partial \boldsymbol{w}} = \frac{\hat{y}_n - y_n}{\hat{y}_n(1 - \hat{y}_n)} \cdot \hat{y}_n(1 - \hat{y}_n) \cdot \boldsymbol{x}_n = (\hat{y}_n - y_n)\boldsymbol{x}_n$$

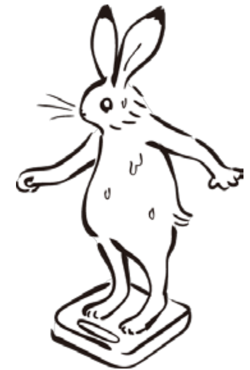
SGD for training SLP

1. Initialize $\boldsymbol{w}$ with random values
2. for $t \leftarrow 1$ to T:
3. $\quad \eta_t \leftarrow 1/t$
4. $\quad (\boldsymbol{x}_n, y_n) \leftarrow$ a random sample from D
5. $\quad \hat{y}_n \leftarrow \sigma(\boldsymbol{w} \cdot \boldsymbol{x}_n)$
6. $\quad \boldsymbol{w} \leftarrow \boldsymbol{w} - \eta_t \frac{\partial l_n}{\partial \boldsymbol{w}} = \boldsymbol{w} + \eta_t (y_n - \hat{y}_n) \boldsymbol{x}_n$

   # If $y_n = \hat{y}_n$, no need for updating $\boldsymbol{w}$
   # If $y_n = 1$ and $\hat{y}_n < 1$, add $\boldsymbol{x}_n$ scaled by $(1 - \hat{y}_n)$ to $\boldsymbol{w}$
   # If $y_n = 0$ and $0 < \hat{y}_n$, subtract $\boldsymbol{x}_n$ scaled by $\hat{y}_n$ to $\boldsymbol{w}$

The algorithm is the same as perceptron except for using the error $(y_n - \hat{y}_n)$ for weighting the amount of an update

Regularization



- MLE often causes over-fitting
  - When the training data is linearly separable

$$|\boldsymbol{w}| \rightarrow \infty \text{ as } \sum_{n=1}^{N} l_n \rightarrow 0$$

  - Subject to be affected by noises in the training data

- We use regularization (MAP estimation)
  - We introduce a penalty term when $\boldsymbol{w}$ becomes large
  - The loss function with an L2 regularization term:

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} l_n + C|\boldsymbol{w}|^2$$

    - C is the hyper parameter to control the trade-off between over/under fitting

Multi-class classification

- We extend binary classification to multi-class
  - Assign a weight vector $\boldsymbol{w}_y$ for every category y
  - Extend Perceptron algorithm to multi-class classification
  - Extend sigmoid function to softmax
  - Again, automatic differentiation is useful for SGD training
- Try ReLU as an activation function of internal layers
- Dropout realizes model averaging in a simple way

Handwritten recognition (MNIST; LeCun+ 1998)

4

1

0

5

6

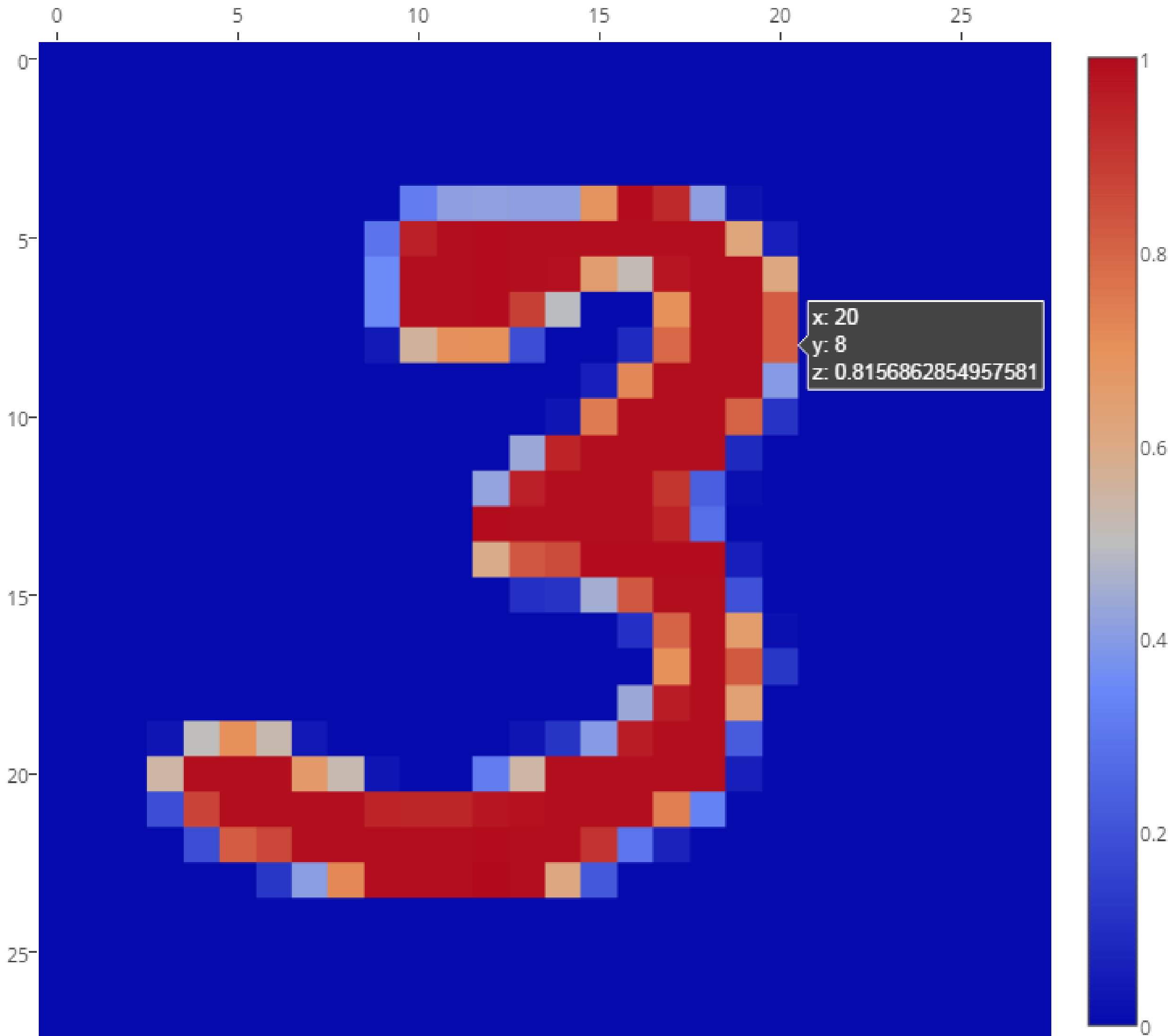
2

8

5

We want to classify an input image into 10 categories (digits)

Image representation



- An image (28 x 28 pixels, grayscale) is represented by a 28 x 28 matrix.
- The original dataset represents a brightness in an 8-bit integer ([0, 255]).
- In this lecture, a brightness is normalized within the range of [0, 1].

Multi-class classification and perceptron algorithm

General form: linear multi-class classification

Output: $\hat{y} \in \mathcal{Y}$

Input: $\boldsymbol{x} \in \mathbb{R}^d$

(d: number of dimension)

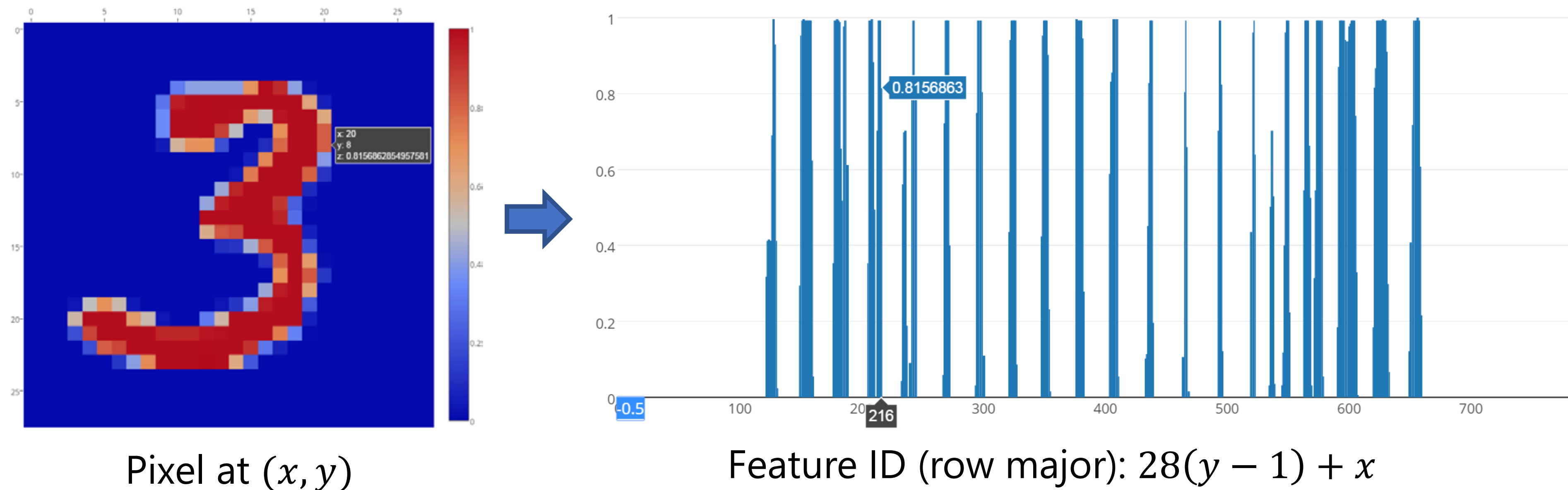
$$\hat{y} = \operatorname*{argmax}_{y \in \mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{x}$$

Set of possible categories
for the input

Parameter: weight $\boldsymbol{w}_y \in \mathbb{R}^d$
(prepared for every category)

Represent an image with a vector

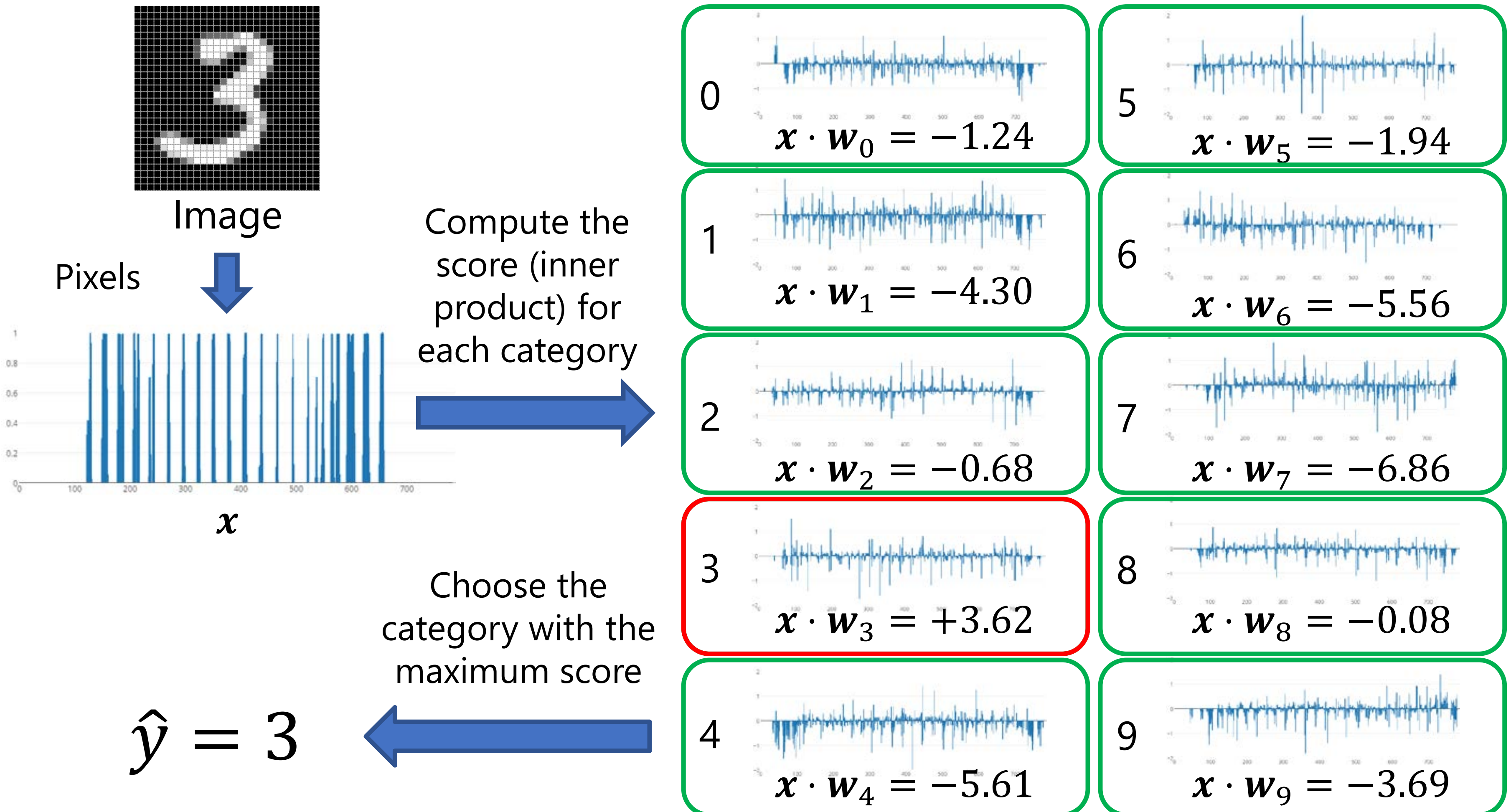
- We simply use the brightness of each pixel as an input vector by flattening a 2D matrix into a 1D vector
  - A 28×28 matrix into a vector of 784 ($= 28 \times 28$) dimension
  - A more sophisticated method (e.g., Convolutional Neural Network) will be presented later
  - Even this simple treatment surprisingly works well



Pixel at (x, y)

Feature ID (row major): $28(y-1)+x$

Linear multi-class classification



A model has a weight vector for every category

Training for multi-class classifier

- A training set consists of N instances:
  - $D = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_N, y_N)\}$

$\boldsymbol{x}_n$: the n-th instance in the training data
y_n: the category for the n-th instance

- We assume *generalization*: if weight vectors $\boldsymbol{w}_y$ predict training instances correctly, it will work for unseen instances

- Find the weight vectors $\boldsymbol{w}_y$ such that they can predict training instances as correctly as possible
  - Ideally, $\hat{y}_n = y_n$ for all $n \in [1, N]$ in the training data

Perceptron algorithm for multi-class
(Collins, 2002)

1. $\boldsymbol{w}_y = 0$ for all $y \in \mathcal{Y}$
2. Repeat:
3. $\quad (\boldsymbol{x}, y) \longleftarrow$ Random sample from the training data D
4. $\quad \hat{y} \longleftarrow \underset{y \in \mathcal{Y}}{\operatorname{argmax}} \, \boldsymbol{w}_y \cdot \boldsymbol{x}$
5. $\quad$ if $\hat{y} \neq y$ then: (*incorrect prediction*)
6. $\quad\quad \boldsymbol{w}_y \longleftarrow \boldsymbol{w}_y + \boldsymbol{x}$ ($\boldsymbol{w}_y \cdot \boldsymbol{x}$ will be larger)
7. $\quad\quad \boldsymbol{w}_{\hat{y}} \longleftarrow \boldsymbol{w}_{\hat{y}} - \boldsymbol{x}$ ($\boldsymbol{w}_{\hat{y}} \cdot \boldsymbol{x}$ will be smaller)
8. Until no instance updates $\boldsymbol{w}_y$

Summary and notes

- Given an input $\boldsymbol{x}$, a linear multi-class classifier compute a score for every category y as an inner product $\boldsymbol{w}_y \cdot \boldsymbol{x}$
  - $\boldsymbol{w}_y$ presents a weight vector for the category y

- It predicts a category $\hat{y}$ for the input yielding the highest score among the possible categories $\mathcal{Y}$

- Weight vectors $\boldsymbol{w}_y$ can be trained by an extension of Perceptron algorithm to multi-class (structured perceptron)
  - Again, we cannot use it for multi-layer neural networks

- Let’s consider SGD for training multi-class classifiers

Multi-class classification with softmax function

Training multi-class classifiers with SGD

- In order to train binary classifiers using SGD, we had to change the activation function from step to sigmoid

- What is the activation function for multi-class classification corresponding to sigmoid function?

- Answer: softmax function

Softmax function: Definition

- Given a vector $\boldsymbol{a} \in \mathbb{R}^m$, softmax $\sigma: \mathbb{R}^m \to \mathbb{R}^m$ yields,

$$\sigma(\boldsymbol{a})_i = \frac{\exp(a_i)}{\sum_{k=1}^{m} \exp(a_k)}$$

  - Here $\sigma(\boldsymbol{a})_i$ denotes the i-th element of the value of $\sigma(\boldsymbol{a})$
  - We use the same notation σ (do not confuse with sigmoid)

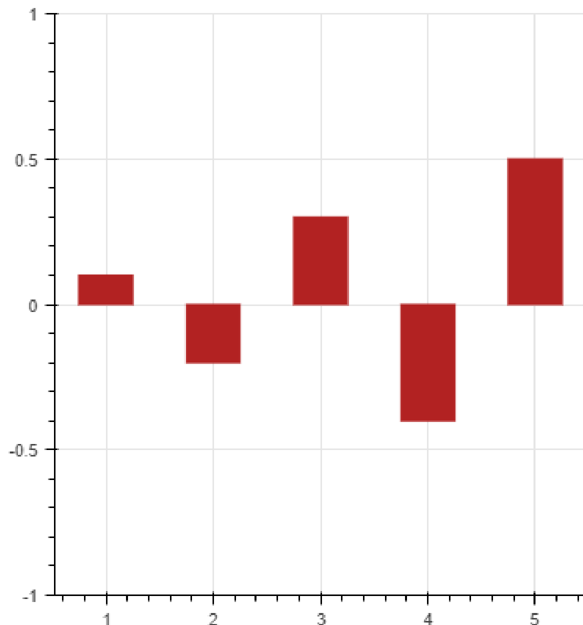
- A result of softmax function satisfies,

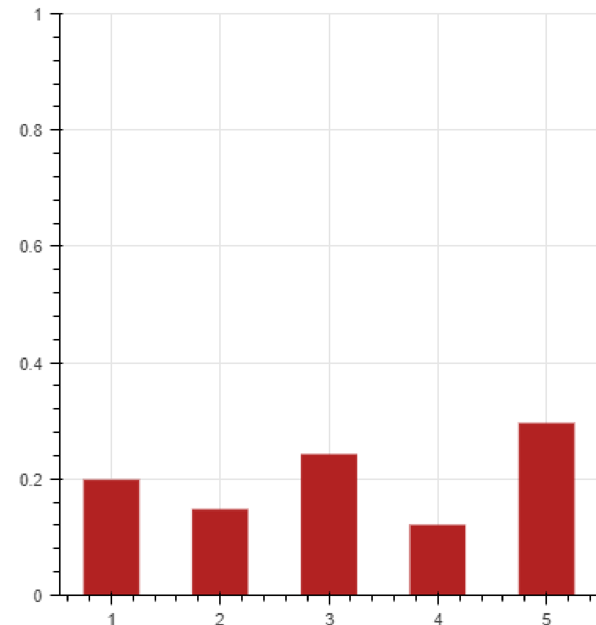
$$\forall k:\ \sigma(\boldsymbol{a})_k > 0,$$

$$\sum_{k=1}^{m} \sigma(\boldsymbol{a})_k = 1$$

Softmax function: Interpretation

- Intuitively, softmax function converts *scores* for m caetgories $\boldsymbol{a} \in \mathbb{R}^m$ into a *probability distribution*
  - In binary classification, sigmoid function converts a score to a *probability*





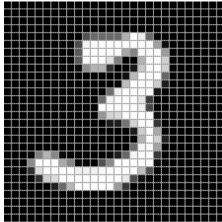
Single-layer NNs for multi-class classification

- Given an input $\boldsymbol{x} \in \mathbb{R}^d$, a single-layer NN for multi-class classification yields a probability distribution over K categories $\hat{\boldsymbol{y}} \in \mathbb{R}^K$,

$$\hat{\boldsymbol{y}} = \sigma(\boldsymbol{a}), \boldsymbol{a} = W\boldsymbol{x}$$

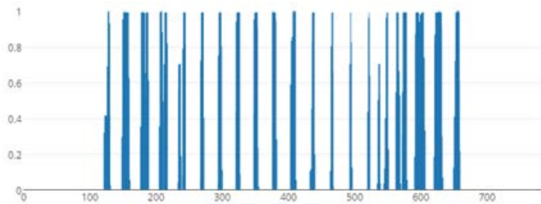
- Here, $W \in \mathbb{R}^{K \times d}$ is a weight matrix
  - W can be seen as a mapping: $\mathbb{R}^d \rightarrow \mathbb{R}^K$
- Let $\boldsymbol{w}_i$ denote the i-th row vector of the matrix W
  - The score for the category i is computed by $a_i = \boldsymbol{w}_i \cdot \boldsymbol{x}$

An example with softmax function



Image

Pixels



$\boldsymbol{x}$

$W\boldsymbol{x}$

$\hat{\boldsymbol{y}} = \sigma(W\boldsymbol{x})$

0 $\boldsymbol{x} \cdot \boldsymbol{w_0} = -1.24$ (0.74%)

1 $\boldsymbol{x} \cdot \boldsymbol{w_1} = -4.30$ (0.03%)

2 $\boldsymbol{x} \cdot \boldsymbol{w_2} = -0.68$ (1.29%)

3 $\boldsymbol{x} \cdot \boldsymbol{w_3} = +3.62$ (95.1%)

4 $\boldsymbol{x} \cdot \boldsymbol{w_4} = -5.61$ (0.01%)

5 $\boldsymbol{x} \cdot \boldsymbol{w_5} = -1.94$ (0.37%)

6 $\boldsymbol{x} \cdot \boldsymbol{w_6} = -5.56$ (0.01%)

7 $\boldsymbol{x} \cdot \boldsymbol{w_7} = -6.86$ (0.00%)

8 $\boldsymbol{x} \cdot \boldsymbol{w_8} = -0.08$ (2.35%)

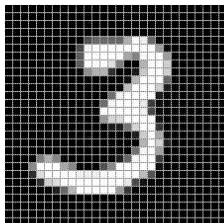
9 $\boldsymbol{x} \cdot \boldsymbol{w_9} = -3.69$ (0.06%)

Supervision data for multi-class

- We have a supervision data
  - $D = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \dots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$ (N instances)

- Input
  - $\boldsymbol{x}_n = (x_{n1}, x_{n2}, \dots, x_{nd})^\top \in \mathbb{R}^d$

- Output (*changed from the previous notation*)
  - $\boldsymbol{y}_n = (y_{n1}, y_{n2}, \dots, y_{nK})^\top \in \mathbb{R}^K$ (*one-hot* vector)



$$\boldsymbol{y} = (0,0,0,1,0,0,0,0,0,0)^\top$$

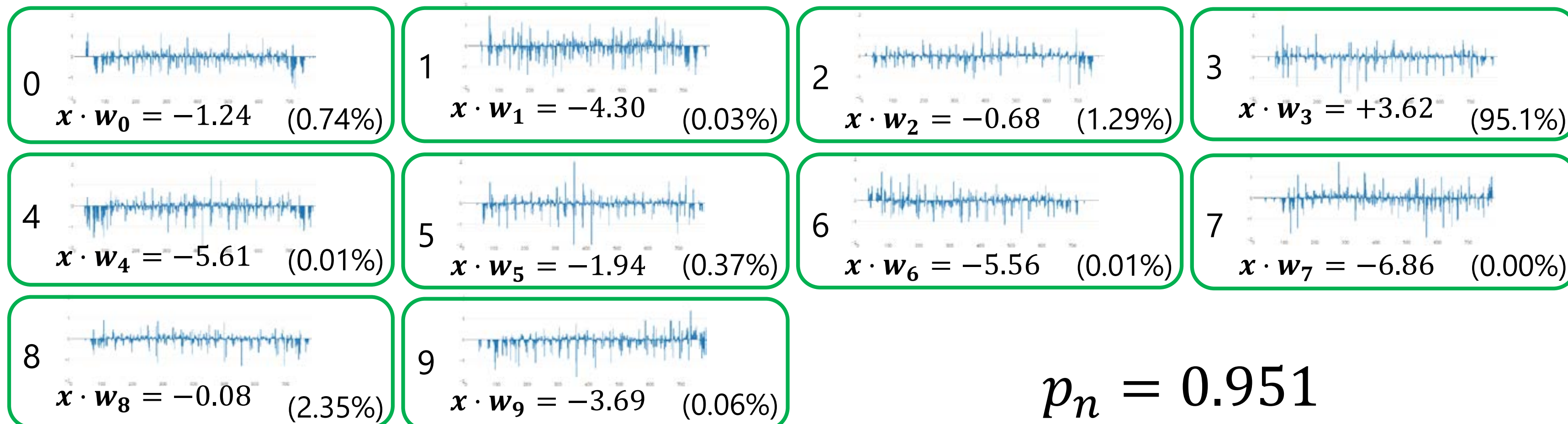
0 ⋯ 3 ⋯ 9

Instance-wise likelihood

- We introduce *instance-wise likelihood* to measure how well the parameters reproduce $(\boldsymbol{x}_n, \boldsymbol{y}_n)$

$$p_n = \prod_{k=1}^{K} \begin{cases} \hat{y}_{nk} & (\text{if } y_{nk} = 1) \\ 1 & (\text{if } y_{nk} = 0) \end{cases} = \prod_{k=1}^{K} \hat{y}_{nk}^{\,y_{nk}}$$

  - The probability of the true label *estimated* by the model



$p_n = 0.951$

Likelihood on the training data

- We assume that all instances in the training data are i.i.d. (independent and identically distributed)
- We define *likelihood* as a joint probability on data,

$$L_D(W) = \prod_{n=1}^{N} p_n$$

  - When the training data $D = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \dots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$ is fixed, likelihood is a function of the parameters W

- Let us maximize $L_D(W)$ by changing W
  - This is called *Maximum Likelihood Estimation (MLE)*
  - The maximizer W^* reproduces the training data well

Training as a minimization problem

- Products of $(0,1)$ values often cause underflow
- Use *log-likelihood*, the logarithm of the likelihood, instead

$$LL_D(W) = \log L_D(W) = \log \prod_{n=1}^{N} p_n = \sum_{n=1}^{N} \log p_n$$

- In mathematical optimization, we usually consider a minimization problem instead of maximization
- We define an objective function $E_D(W)$ by using the negative of the log-likelihood

$$E_D(W) = -LL_D(W) = -\sum_{n=1}^{N} \log p_n$$

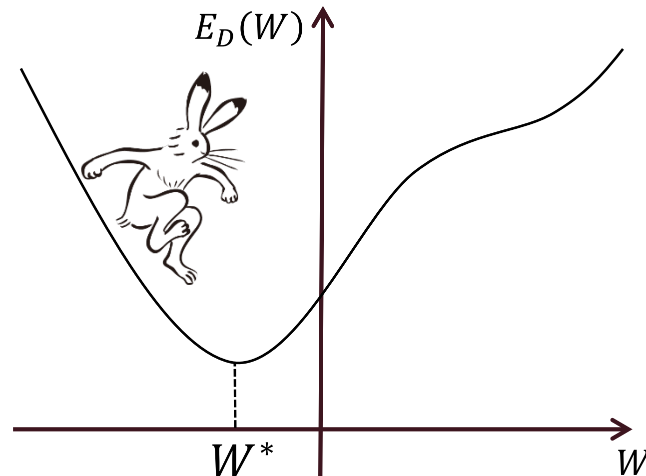
  - $E_D(W)$ is called a *loss function* or *error function*

Training as a minimization problem

- Given the training data $D = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \dots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$, find W^* as the minimization problem,

$$W^* = \operatorname*{argmin}_W E_D(W) = \operatorname*{argmin}_W \sum_{n=1}^{N} l_n \, ,$$

$$l_n = -\log p_n = -\log \prod_{k=1}^{K} \hat{y}_{nk}^{\,y_{nk}} = -\sum_{k=1}^{K} y_{nk} \log \hat{y}_{nk}$$



Stochastic Gradient Descent (SGD)

- The objective function is the sum of losses of instances,

$$E_D(W) = \sum_{n=1}^{N} l_n$$

- We can use Stochastic Gradient Descent (SGD) and its variants (e.g., Adam) for minimizing $E_D(W)$
- SGD Algorithm (T is the number of updates)
  1. For every k, initialize $\boldsymbol{w}_k$ with random values
  2. for $t \leftarrow 1$ to T:
  3. $\quad \eta_t \leftarrow 1/t$
  4. $\quad (\boldsymbol{x}_n, y_n) \leftarrow$ a random sample from D
  5. $\quad \forall k$: $\boldsymbol{w}_k \leftarrow \boldsymbol{w}_k - \eta_t \frac{\partial l_n}{\partial \boldsymbol{w}_k}$

Exercise: compute the gradient

- Prove (we omit the instance index n for simplicity):

$$\frac{\partial l}{\partial \boldsymbol{w}_i} = \frac{\partial l}{\partial a_i}\frac{\partial a_i}{\partial \boldsymbol{w}_i} = (\hat{y}_i - y_i)\boldsymbol{x}$$

  by computing the gradients $\frac{\partial l}{\partial a_i}$ and $\frac{\partial a_i}{\partial \boldsymbol{w}_i}$

- Here:

$$l = -\sum_{k=1}^{K} y_k \log \hat{y}_k \,,$$

$$\hat{y}_i = \sigma(a_i) = \frac{\exp(a_i)}{\sum_{k=1}^{K} \exp(a_k)},$$

$$a_i = \boldsymbol{w}_i \cdot \boldsymbol{x}$$

Answer: compute the gradients

- Because it is easy to find $\frac{\partial a_i}{\partial \boldsymbol{w}_i} = \boldsymbol{x}$, we concentrate on $\frac{\partial l}{\partial a_i}$,

$$\frac{\partial l}{\partial a_i} = -\sum_{k=1}^{K} y_k \frac{\partial \log \hat{y}_k}{\partial a_i} = -\sum_{k=1}^{K} y_k \frac{1}{\hat{y}_k}\frac{\partial \hat{y}_k}{\partial a_i} = -y_i \frac{1}{\hat{y}_i}\frac{\partial \hat{y}_i}{\partial a_i} - \sum_{k \neq i} y_k \frac{1}{\hat{y}_k}\frac{\partial \hat{y}_k}{\partial a_i}$$

- The first term is,

$$-y_i \frac{1}{\hat{y}_i}\frac{\partial \hat{y}_i}{\partial a_i} = -y_i \frac{1}{\hat{y}_i}\frac{\partial}{\partial a_i}\left(\frac{\exp(a_i)}{\sum_{k=1}^{K}\exp(a_k)}\right) = -y_i \frac{1}{\hat{y}_i}\frac{\exp(a_i)\Sigma - \exp(a_i)\exp(a_i)}{\Sigma^2}$$

$$= -y_i \frac{1}{\hat{y}_i}\frac{\exp(a_i)}{\Sigma}\frac{\Sigma - \exp(a_i)}{\Sigma} = -y_i \frac{1}{\hat{y}_i}\hat{y}_i(1-\hat{y}_i) = -y_i(1-\hat{y}_i) = -y_i + y_i\hat{y}_i$$

- The second term is,

$$-\sum_{k \neq i} y_k \frac{1}{\hat{y}_k}\frac{\partial \hat{y}_k}{\partial a_i} = -\sum_{k \neq i} y_k \frac{1}{\hat{y}_k}\frac{\partial}{\partial a_i}\left(\frac{\exp(a_k)}{\sum_{k'=1}^{K}\exp(a_{k'})}\right) = -\sum_{k \neq i} y_k \frac{1}{\hat{y}_k}\frac{0 - \exp(a_k)\exp(a_i)}{\Sigma^2}$$

$$= \sum_{k \neq i} y_k \frac{1}{\hat{y}_k}\hat{y}_k\hat{y}_i = \sum_{k \neq i} y_k\hat{y}_i$$

When $f(x) = \frac{g(x)}{h(x)}$, $f'(x) = \frac{g'(x)h(x) - g(x)h'(x)}{[h(x)]^2}$

$\Sigma = \sum_{k=1}^{K}\exp(a_k)$

- Therefore,

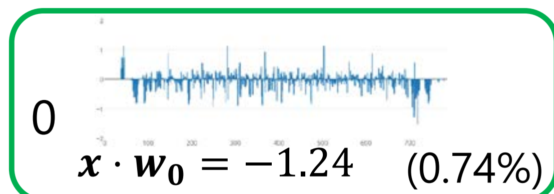
$$\frac{\partial l}{\partial a_i} = -y_i + y_i\hat{y}_i + \sum_{k \neq i} y_k\hat{y}_i = -y_i + \hat{y}_i\sum_{k=1}^{K} y_k = -y_i + \hat{y}_i$$

SGD for training SLP

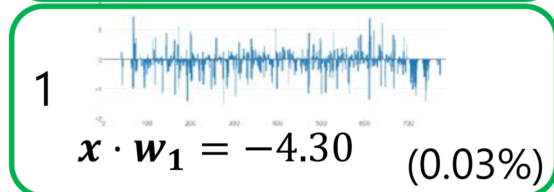
1. For every k, initialize $\boldsymbol{w}_k$ with random values
2. for $t \leftarrow 1$ to T:
3. $\quad \eta_t \leftarrow 1/t$
4. $\quad (\boldsymbol{x}_n, y_n) \leftarrow$ a random sample from D
5. $\quad \hat{y}_n \leftarrow \sigma(\boldsymbol{w} \cdot \boldsymbol{x}_n)$
6. $\quad \forall k\colon \boldsymbol{w}_k \leftarrow \boldsymbol{w}_k - \eta_t \frac{\partial l_n}{\partial \boldsymbol{w}_k} = \boldsymbol{w}_k + \eta_t (y_{nk} - \hat{y}_{nk}) \boldsymbol{x}_n$

- The algorithm is the same as that for binary classification
- For each category k, it updates a weight $\boldsymbol{w}_k$ by the amount of the error $(y_{nk} - \hat{y}_{nk})$ between the true probability y_{nk} and the estimated probability $\hat{y}_{nk}$

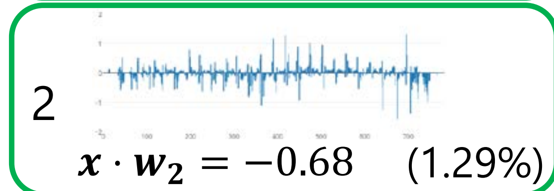
Intuitive example of SGD updates $(\eta_t = 1)$



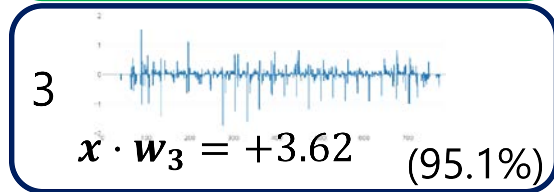
$-= \ 0.0074\boldsymbol{x}$



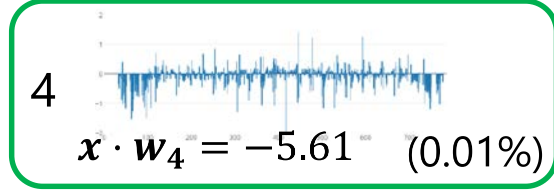
$-= \ 0.0003\boldsymbol{x}$



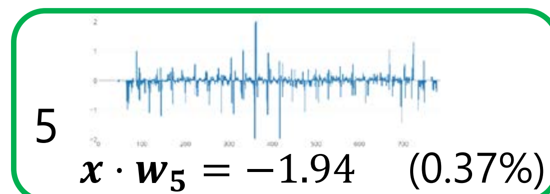
$-= \ 0.0129\boldsymbol{x}$



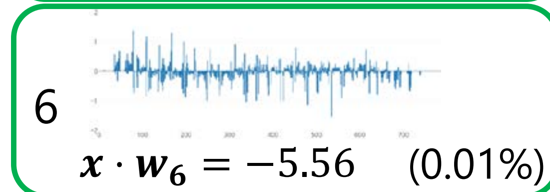
$+= \ 0.0490\boldsymbol{x}$



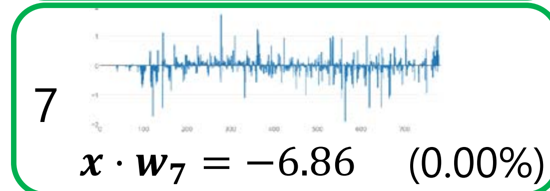
$-= \ 0.0001\boldsymbol{x}$



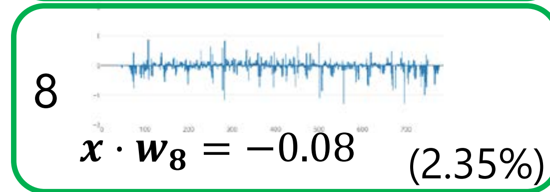
$-= \ 0.0037\boldsymbol{x}$



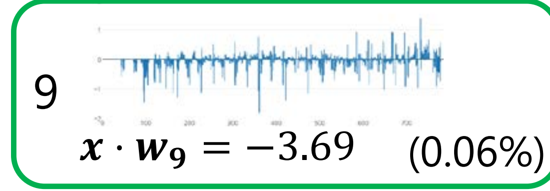
$-= \ 0.0001\boldsymbol{x}$



$-= \ 0.0000\boldsymbol{x}$



$-= \ 0.0235\boldsymbol{x}$



$-= \ 0.0006\boldsymbol{x}$

Computing the loss with mini-batch

- Single-batch

$$\sigma\left(\underset{d}{{}_{1}\boxed{\boldsymbol{x}}} \times {}_{d}\underset{K}{\boxed{W}} \right) = \underset{K}{\boxed{\hat{\boldsymbol{y}}}}{}_{1} \Leftrightarrow \underset{K}{\boxed{\boldsymbol{y}}}{}_{1}$$

$$l = -\boldsymbol{y} \cdot \log \hat{\boldsymbol{y}}$$

- Mini-batch (*parallelizable in GPU*)

$$\sigma\left(\underset{d}{{}_{m}\boxed{X}} \times {}_{d}\underset{K}{\boxed{W}} \right) = \underset{K}{\boxed{\hat{Y}}}{}_{m} \Leftrightarrow \underset{K}{\boxed{Y}}{}_{m}$$

$$l = -\frac{1}{m} \sum_{n=1}^{m} \boldsymbol{y}_n \cdot \log \hat{\boldsymbol{y}}_n$$

Mini-batch training

- Most DL frameworks implement mini-batch training by increasing the order of tensors:
  - For example, $(d) \rightarrow (\mathrm{m} \times d)$

- Increasing the batch size (m) may:
  - Speed up time required for an epoch with parallelization
  - Decrease the number of parameter updates $(1/m)$

- This paper (Goyal+ 2017) recommends:
  - When the minibatch size is multiplied by k, multiply the learning rate by k

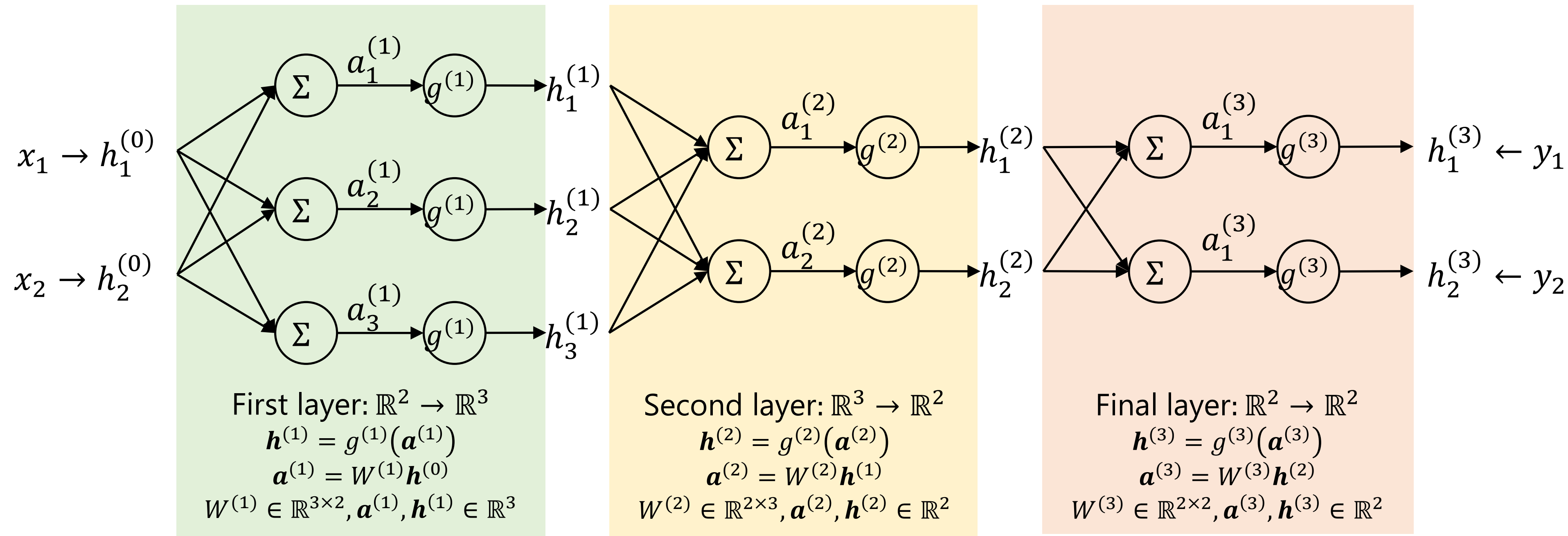
Summary and notes

- K-class classification is realized by changing the dimension of an output layer to K
- Softmax yields a probability distribution $\hat{\boldsymbol{y}} \in \mathbb{R}^K$
- The loss function compares a model output $\hat{\boldsymbol{y}}$ with an one-hot vector of a true category $\boldsymbol{y}$
- Again, automatic differentiation is also useful for training multi-class NNs

- A single-layer NN with softmax activation function is also known as *multi-class logistic regression* and *maximum entropy modeling*

Generic form of Feedforward Neural Networks

Designing feedforward neural networks



- The number of layers
- The numbers of dimensions of hidden layers
- An activation function for each layer
- A loss function

Cross entropy loss

- For binary classification

$$l(a, y) = -y \log \sigma(a) - (1 - y) \log\big(1 - \sigma(a)\big)$$

- For multi-class classification

$$l(\boldsymbol{a}, y) = -a_y + \log \sum_k \exp(a_k)$$

- Cross entropy

$$H(p, q) = -\sum_k p(k) \log q(k)$$

True probability distribution
(1 for true category; 0 otherwise)

Predicted probability distribution

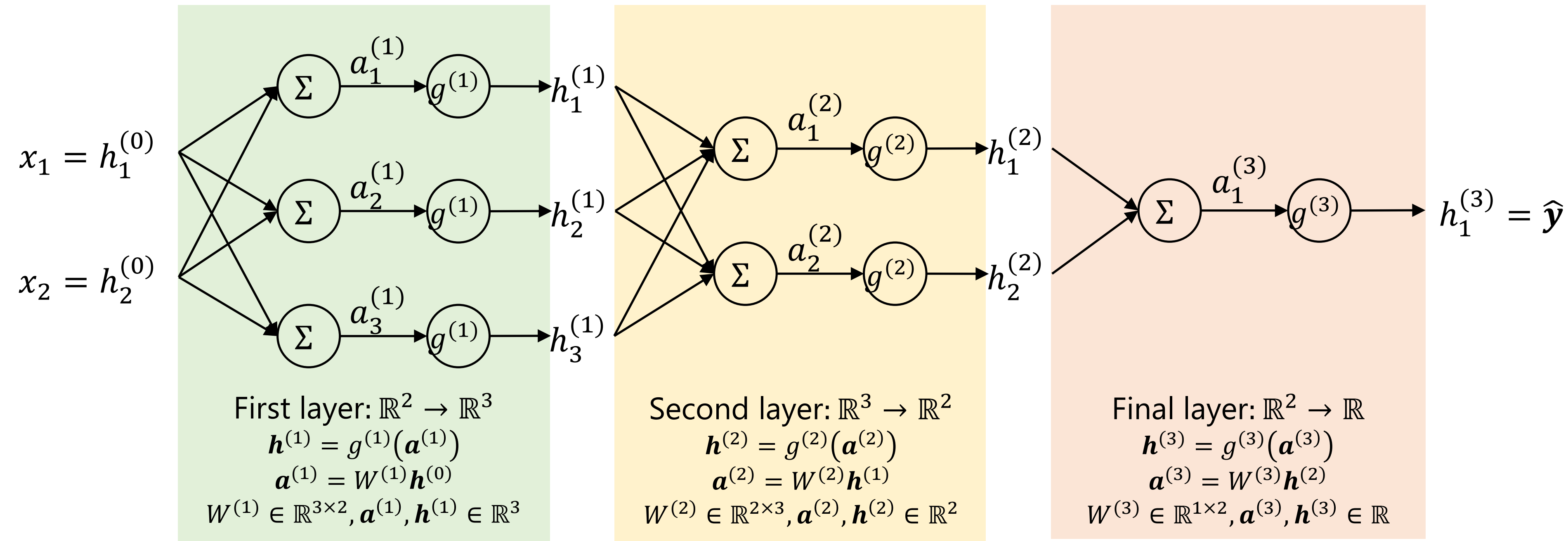
Mean Squared Error (MSE) loss

- Used for regression

$$l(\boldsymbol{a}, \boldsymbol{y}) = \frac{1}{2} \|\boldsymbol{y} - \boldsymbol{a}\|_2^2$$

Training multi-layer neural networks and back propagation

Generic notation for multi-layer NNs



- The l–th layer ($l \in \{1, \dots, L\}$) consists of:
  - Input: $\boldsymbol{h}^{(l-1)} \in \mathbb{R}^{d_{l-1}}$ $(\boldsymbol{h}^{(0)} = \boldsymbol{x})$
  - Output: $\boldsymbol{h}^{(l)} \in \mathbb{R}^{d_l}$ $(\boldsymbol{h}^{(L)} = \hat{\boldsymbol{y}})$
  - Weight: $W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$
  - Activation function: $g^{(l)}$
  - Activation: $\boldsymbol{a}^{(l)} \in \mathbb{R}^{d_l}$

$$\boldsymbol{h}^{(l)} = g^{(l)}(W^{(l)}\boldsymbol{h}^{(l-1)})$$

$$W^{(l)} = \left(w_{ij}^{(l)}\right)$$

$w_{ij}^{(l)}$: weight from the j-th neuron to the i-th neuron of the l-th layer

Please accept the notational conflict between an instance-wise loss l_n and a layer number l

How to train weights in MLPs

- We have no explicit supervision signals for the internal (hidden) inputs/outputs $\boldsymbol{h}^{(2)}, \dots, \boldsymbol{h}^{(L-1)}$

- Having said that, SGD only needs the value of gradient $\frac{\partial l_n}{\partial w_{ij}^{(l)}}$ for every weight $w_{ij}^{(l)}$ in MLPs

- Can we compute the value of $\frac{\partial l_n}{\partial w_{ij}^{(l)}}$ for every weight $w_{ij}^{(l)}$?

- Yes! *Backpropagation* can do that!!

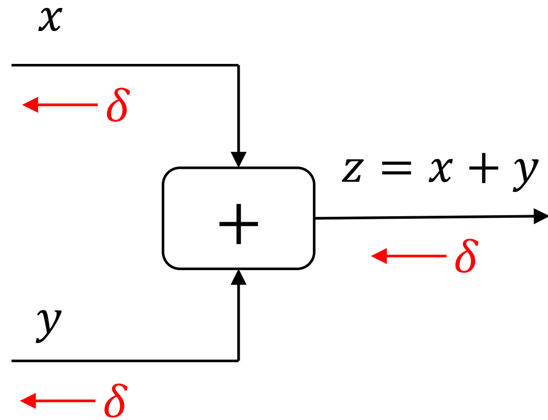
Backpropagation

- Commonly used in deep neural networks
- Formulas for backpropagation look complicated
- However:
  - We can understand backpropagation easily if we know the concept of *computation graph*
  - Most deep learning frameworks implement backpropagation by using *automatic differentiation*
- Let's see computation graph and automatic differentiation first

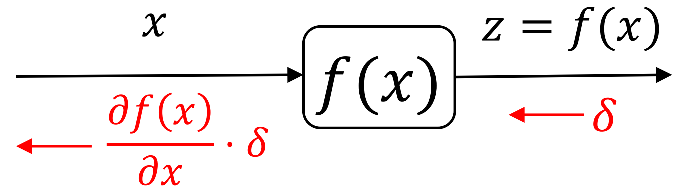
General Back-Propagation

The back-propagation algorithm is very simple. To compute the gradient of some scalar z with respect to one of its ancestors $\boldsymbol{x}$ in the graph, we begin by observing that the gradient with respect to z is given by $\frac{dz}{dz} = 1$. We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z. We continue multiplying by Jacobians traveling backwards through the graph in this way until we reach $\boldsymbol{x}$. For any node that may be reached by going backwards from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

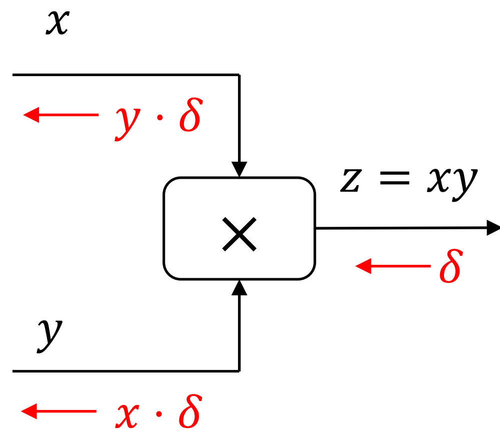
Rules for reverse-mode AD



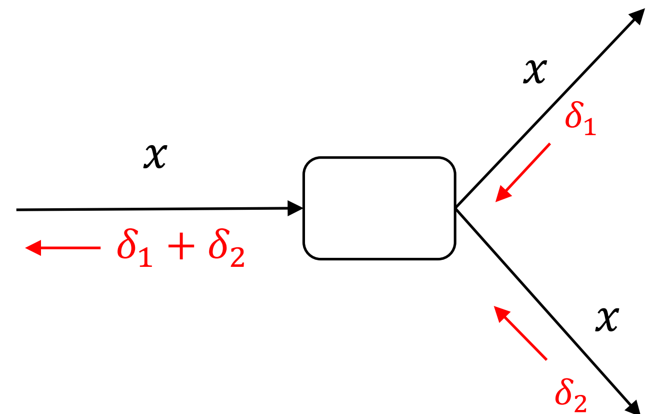
Add



Function application



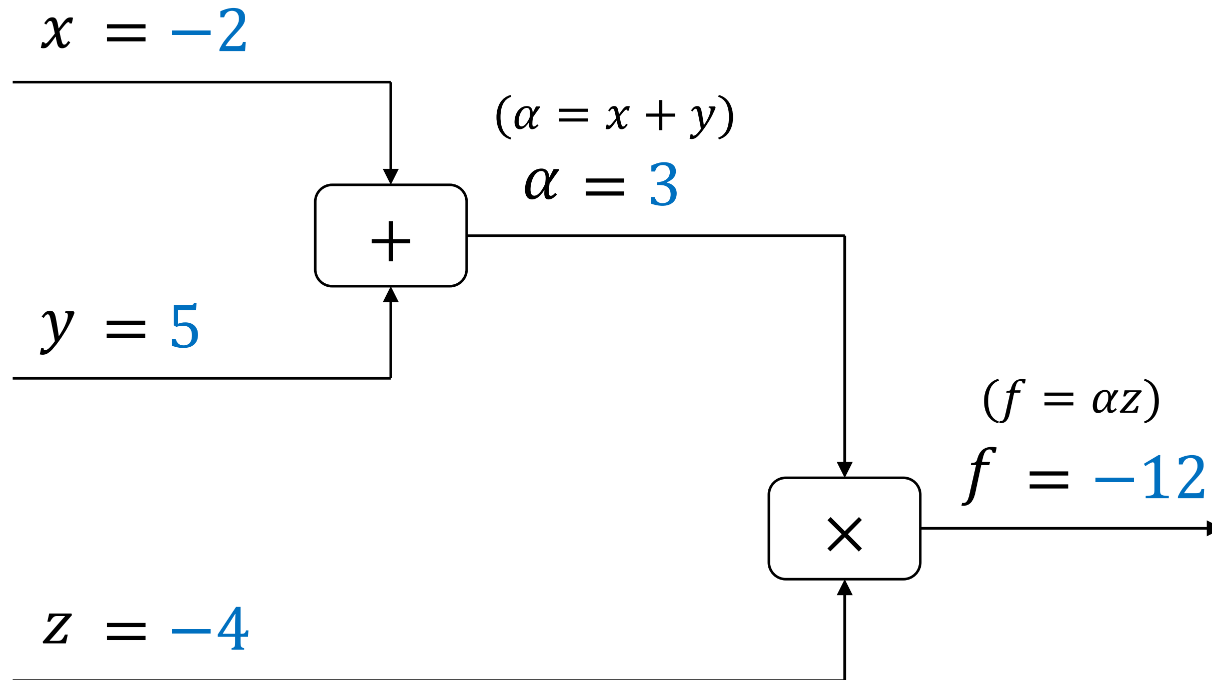
Multiply



Branch

Computation graph: $f(x, y, z) = (x + y)z$

http://cs231n.github.io/optimization-2/

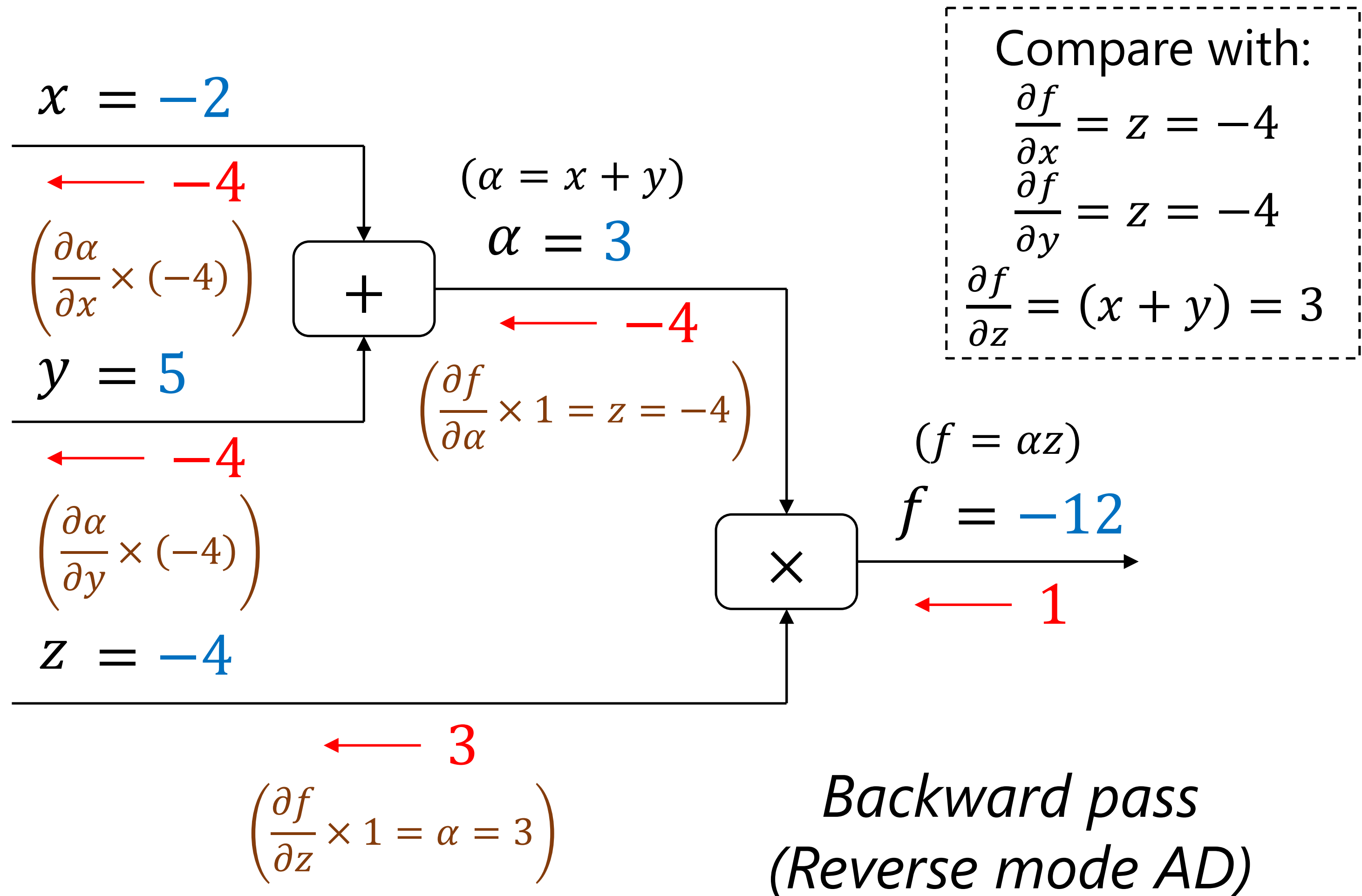


Forward pass

The value of a variable (above an arrow)

Automatic Differentiation (AD): $f(x, y, z) = (x + y)z$

http://cs231n.github.io/optimization-2/



The value of a variable (above an arrow)

The gradient of the output f with respect to the variable (below an arrow)

Automatic differentiation (Baydin+ 2018)

- AD computes derivations by using the *chain rule*
  - Function values computed in the forward pass
  - Derivations computed with respect to:
    - Every variable (in reverse-mode accumulation)
    - A specific variable (in forward-mode accumulation)

- Do not confuse with these:
  - Numerical differentiation: for example, $\frac{\partial f(x)}{\partial x} = \frac{f(x+\delta) - f(x)}{\delta}$
  - Symbolic differentiation: e.g., Mathematica, sympy

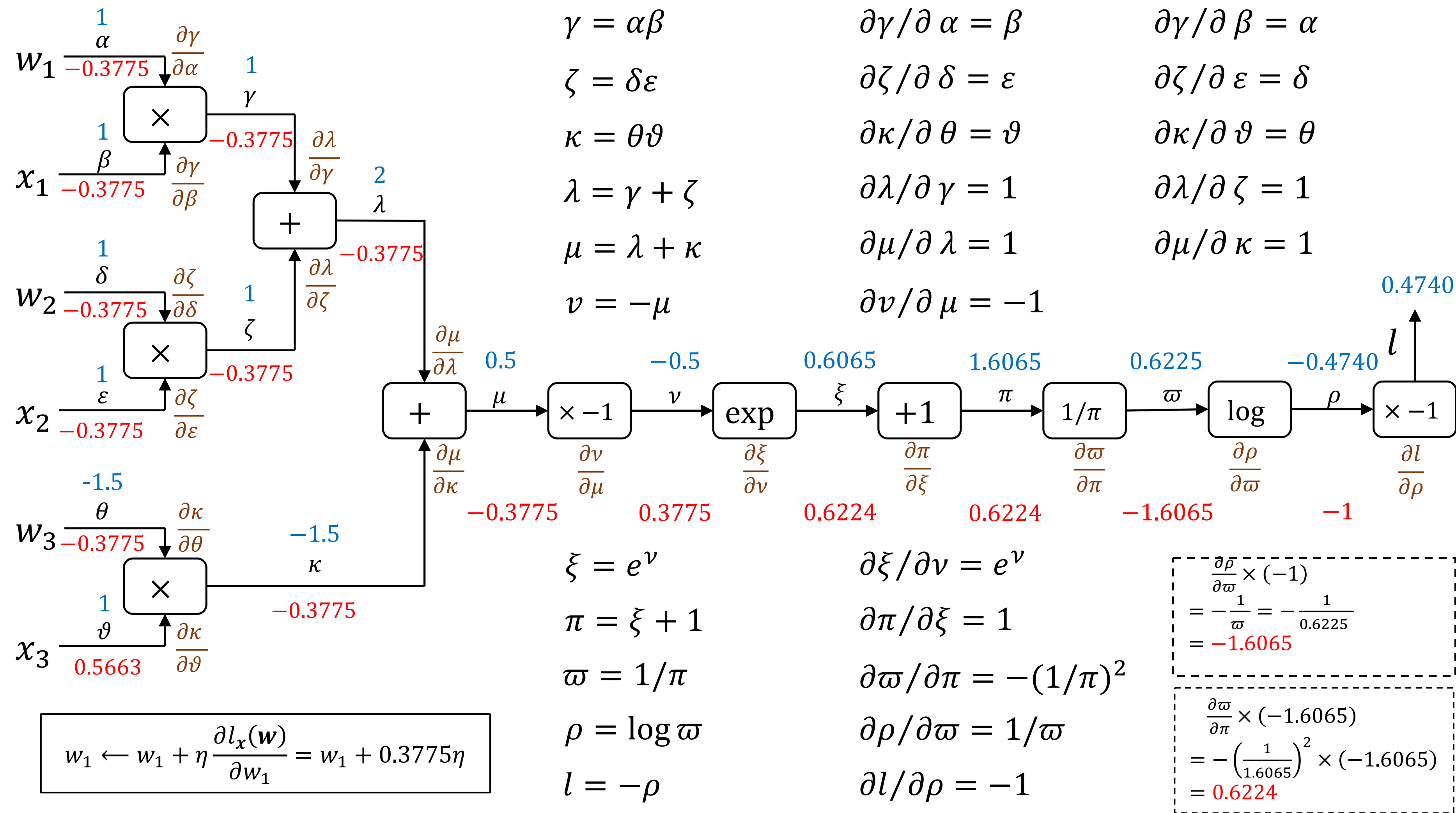
Exercise: AD on computation graph

- Write a computation graph for $l_{\boldsymbol{x}}(w)$,

$$l_{\boldsymbol{x}}(w) = -\log \sigma(\boldsymbol{w} \cdot \boldsymbol{x}) = -\log \frac{1}{1 + e^{-\boldsymbol{w} \cdot \boldsymbol{x}}}$$

- Consider $\boldsymbol{x} = (1,1,1)^\top$ and $\boldsymbol{w} = (1,1,-1.5)^\top$
  - Compute the value of $l_{\boldsymbol{x}}(\boldsymbol{w})$
  - Compute gradients $\frac{\partial l_{\boldsymbol{x}}(\boldsymbol{w})}{\partial \boldsymbol{w}}$

Computing $\frac{\partial l_x(\boldsymbol{w})}{\partial \boldsymbol{w}}$ using AD



No need to derive backpropagation

- Manual derivation of gradients is tedious and error-prone
  - Debugging a mistake in gradients is extremely difficult

- AD is employed in most deep learning frameworks
  - We only need implement an algorithm for a forward pass, i.e., how to compute an output from an input
  - We can concentrate on designing a structure of neural network
  - This boosted the speed of research and development
  - The idea of AD is not new (since 1959)

- Deriving a formula for backpropagation is legacy

Summary and notes

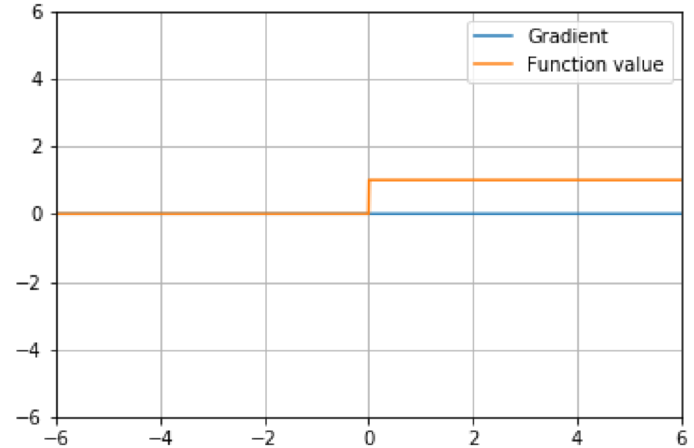
- We design:
  - A neural network model $f(\boldsymbol{x};\theta)$ (with parameters θ)
  - A loss function: $E_D(\theta) = \sum_{n=1}^{N} \mathcal{L}(f(\boldsymbol{x}_n;\theta), y_n)$
    - $\mathcal{L}$ is an instance-wise loss function
    - D presents a set of training data $D = \big((x_1, y_1), \dots, (x_N, y_N)\big)$
- We find a minimizer θ^* for $E_D(\theta)$ by using SGD
  - An update formula for every parameter $w \in \theta$ is derived in a generic manner based on automatic differentiation
- Step function is inappropriate for backpropagation
  - Gradients will not flow because $g'(a) = 0$ at $a \neq 0$

Activation functions

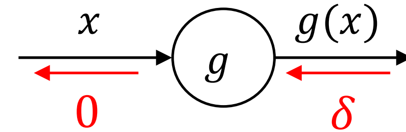
Step

Step function: $\mathbb{R} \rightarrow \{0,1\}$

$$g(x) = \begin{cases} 1 & (\text{if } x > 0) \\ 0 & (\text{otherwise}) \end{cases}$$



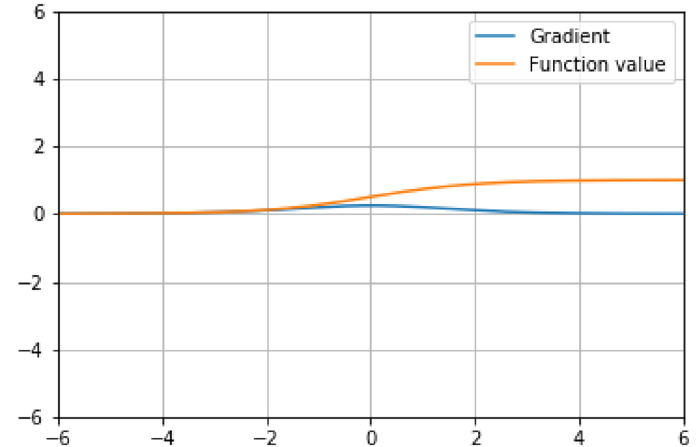
- Pros
  - Yields a binary output
- Cons (never use this)
  - Zero gradients
    - SGD cannot update parameters because $\frac{\partial l}{\partial w} = 0$



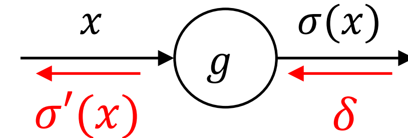
Sigmoid

Sigmoid: $\mathbb{R} \to (0,1)$

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

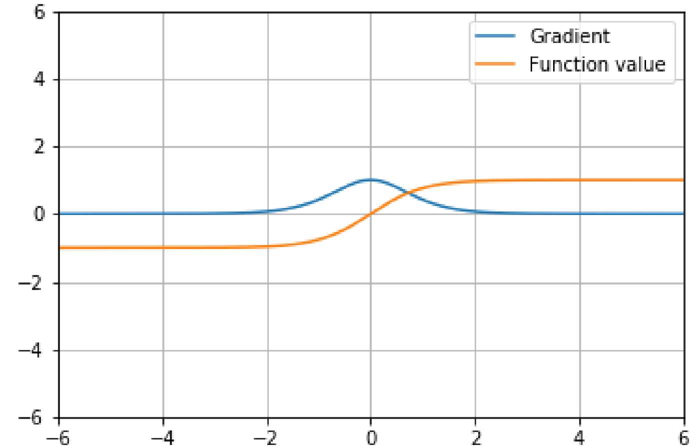


- Pros
  - Yields an output within $(0,1)$
- Cons
  - Not zero-centered
  - Zero (vanishing) gradients when $|x|$ is large



Hyperbolic tangent (tanh)

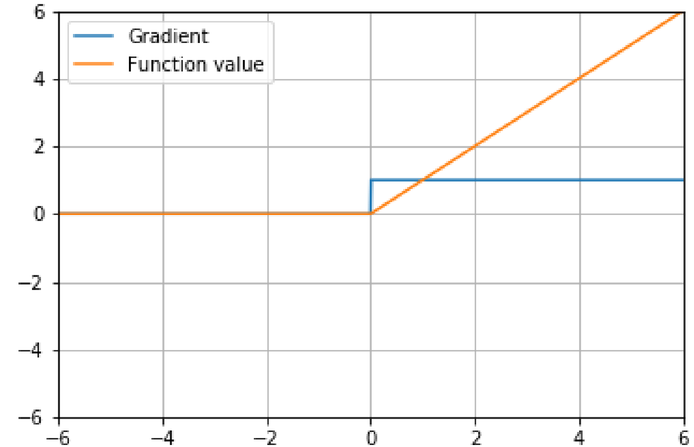
$$\tanh: \mathbb{R} \to (-1,1)$$
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$



- Pros
  - Yields an output within $(-1,1)$
  - Zero-centered
- Cons
  - Zero (vanishing) gradients when $|x|$ is large

Rectified Linear Unit (ReLU)

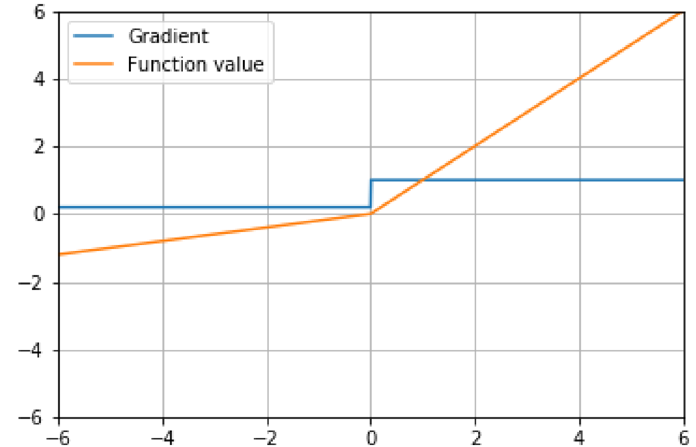
$$\text{ReLU}: \mathbb{R} \rightarrow \mathbb{R}_{\geq 0}$$
$$\text{ReLU}(x) = \max(0, x)$$



- Pros
  - Gradients do not vanish when $x > 0$
  - Light-weight (no e^x) computation
  - Faster convergence (e.g., 6x faster on CIFAR-10)
- Cons
  - Not zero centered
  - Dead neurons when $x \leq 0$

Leaky ReLU

$$\text{Leaky ReLU}: \mathbb{R} \to \mathbb{R}$$
$$\text{LeakyReLU}_\alpha(x) = \max(\alpha x, x)$$



- Pros
  - Gradients do not vanish
  - Light-weight (no e^x) computation
- Cons
  - Not zero centered
  - Not so much improvement over ReLU in practice

Typical definition of a DNN

Typical example of a (pseudo) definition for a Deep Neural Network of depth L (L-1 hidden layers and 1 output layer) for (almost) all modern DL frameworks and libraries:

```
initialize dnnmodel
```

dataset consists of D inputs x of dimensions d each

set hidden Layer 1 as a fully connected (fc) layer to inputs x containing n1 neurons (n1 x d connections):

```
dnnmodel.fc1(n1)
fc1.activation-function=relu
```

set hidden Layer 2 as a fully connected layer to Layer 1 containing n2 neurons (n2 x n1 connections):

```
dnnmodel.fc2(n2)
fc2.activation-function=relu
```

… more hidden layers ...

set hidden Layer L-1 as a fully connected layer to Layer L-2 containing nL neurons (nL-1 x nL-2 connections):

```
dnnmodel.fcL-1(nL-1)
fcL-1.activation-function=relu
```

set fcL (output layer) in a binary classification problem (1 x nL-1 connections)

```
dnnmodel.fcL(1)
fcL.activation-function=sigmoid
```

OR set fcL (output layer) in a K-class multiclass classification problem with one-hot encoding (K x nL-1 connections)

```
dnnmodel.fcL(K)
fcL.activation-function=softmax
```

Set Loss function to Cross Entropy (based on Maximum Likelihood Estimation). Use C for L2 weight regularization.

```
dnnmodel.loss=crossentropy(C)
```

Set the solver to mini batch SGD (or variations) with learning rate lr that by default will use automatic differentiation for the backprop gradient computation. Also set maximum number of epochs (e) and batch size (m) and use of dropout for further regularization. (one epoch equals D examples used for training)

```
dnnmodel.solver=minibatchSGD(lr,m,e,dropout=yes)
```

What made Deep Neural Networks possible and efficient?

Factors from the natural evolution of computation
- Better computers and software allow bigger networks with higher capacity to solve more difficult problems
- With bigger datasets available we must use stochastic methods like SGD

Algorithmic factors
- Cross-entropy is a better loss function than MSE for sigmoid, softmax
- ReLU in hidden layers is a better activation function than sigmoid and tanh for deeper networks
- Automatic differentiation is now a feature of all DL frameworks

What is the difference between a neural network and a deep neural network, and why do the deep ones work better?

Short answer: DNN simply seem to perform better! Read the first answer in the following link:

https://stats.stackexchange.com/questions/182734/what-is-the-difference-between-a-neural-network-and-a-deep-neural-network-and-w

Bibliography

- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6), 386.
- Minsky, M.L. and Papert, S.A. (1969) Perceptrons. MIT Press, Cambridge.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). Learning internal representations by error propagation (No. ICS-8506). California Univ San Diego La Jolla Inst for Cognitive Science.
- Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. science, 313(5786), 504-507.
- Jarrett, K., Kavukcuoglu, K., & LeCun, Y. (2009, September). What is the best multi-stage architecture for object recognition?. In 2009 IEEE 12th International Conference on Computer Vision (ICCV) (pp. 2146-2153). IEEE.
- Glorot, X., Bordes, A., & Bengio, Y. (2011, June). Deep sparse rectifier neural networks. In Proceedings of the fourteenth international conference on artificial intelligence and statistics (pp. 315-323).
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. nature, 521(7553), 436.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press. CHAPTER 6
- Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. Journal of Machine Learning Research, 18, 1-43.

All papers included in "DFFN bibliography" folder - Goodfellow book in mycourses documents root folder
DjVu viewer (if needed): https://sourceforge.net/projects/windjview/

Implementation: Comparison of MLP and DFFN with different activation functions (PyTorch, MNIST dataset)

- Model A: 1 Hidden Layer Feedforward Neural Network (Sigmoid Activation)
- Model B: 1 Hidden Layer Feedforward Neural Network (Tanh Activation)
- Model C: 1 Hidden Layer Feedforward Neural Network (ReLU Activation)
- Model D: 2 Hidden Layer Feedforward Neural Network (ReLU Activation)
- Model E: 3 Hidden Layer Feedforward Neural Network (ReLU Activation)

notebook: pytorch_feedforward_neuralnetwork.ipynb in mycourses folder

https://www.deeplearningwizard.com/deep_learning/practical_pytorch/pytorch_feedforward_neuralnetwork/