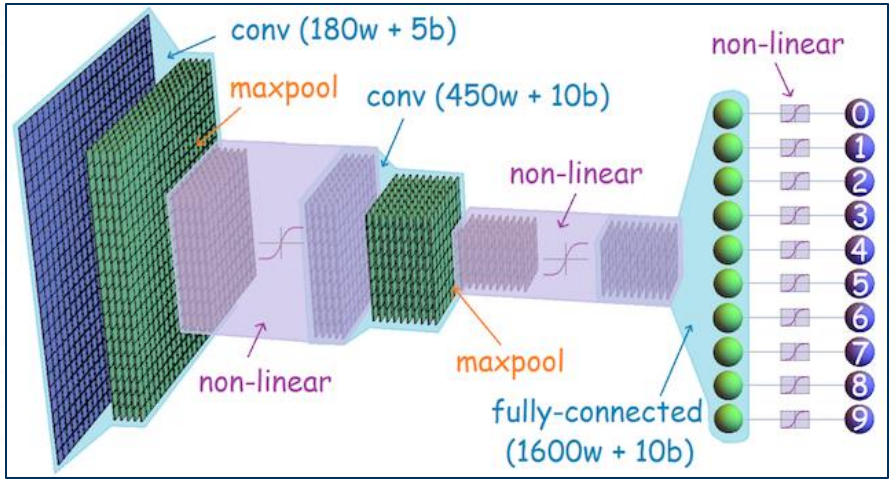
ECE 4803/8803: Fundamentals of Machine Learning (FunML)
Spring 2024

Lecture 13: Neural Networks



OLIVES @GeorgiaTech

Georgia Tech

Logistics

- Reminders
  - HW 4, Deadline: Oct 11
- Midterms
  - Avg: 86
  - Std. Dev: 12
  - Maximum: 100 (4 students)
  - Your scores are posted.
  - You have 1 week (Oct 14) to ask for any regrades. Post a private question on Piazza (preferable) or email me (mohit.p@gatech.edu) with ECE 4252/8803 in subject line.

Logistics for 8803 Section

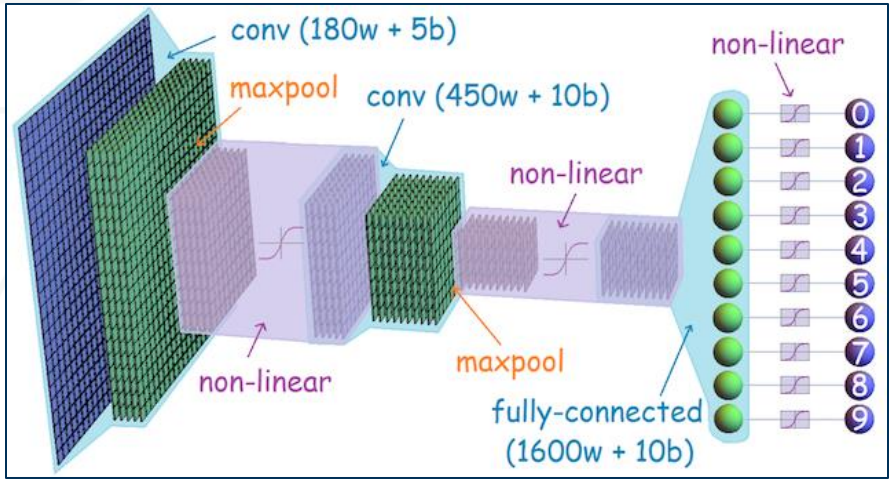
- Project: Biomarker Analysis on OLIVES dataset
- Logistics: Groups of 3
  - For a grad course, 20% project grade is roughly 40-50 hrs per person. Hence, project outputs must reflect 120-150 hrs of combined work
  - Grading is done on individual contributions
- Dataset download: https://zenodo.org/records/7105232 (also available on AI makerspace)
- Goal: Detect Biomarkers given OCT images, and clinical labels
  - You are free to choose your train/test splits
  - You are free to choose your methods/evaluation
- Challenges you will be evaluated on
  - How will you handle multi-modal data (images, text etc.)?
  - What methods will you employ for evaluation?
  - **What analysis will you perform on results?**
  - How good are your results?

Logistics for 8803 Section

- Some resources:
  - Original paper: https://arxiv.org/pdf/2209.11195
  - 2023 SPS VIP competition: https://alregib.ece.gatech.edu/competitions/2023-vip-cup/
  - Competition structured dataset: https://zenodo.org/records/8040573
  - Example paper from the competition: https://arxiv.org/pdf/2310.14005
- Group members
  - Form groups here: https://docs.google.com/spreadsheets/d/1LriuGQuiF4RDL8DbOlg-ZXMCrhokEjFPCT0AlLJfOPs/edit?usp=sharing
- Deliverables: (More instructions when we approach the deadline)
  - 25 Oct: “Progress” Report (PDF), needs to have the following
    - Team members (Representative team member uploads the PDF)
    - Problem description
    - Proposed approach (no results necessary)
    - No specific format
  - 29 Nov: Term Project Paper (PDF)
    - Max 4 pages of content + unlimited pages for references
    - IEEE conference style paper format
    - Code on Github/HuggingFace

Overview

In this Lecture..



Review: Artificial Neural Networks

- Activation Function
- Perceptron Network
- Multi-layer ANN

Backpropagation

- Feedforward and Backward Error Propagation
- Learning Algorithm
- Image Classification using ANNs
- Mini-batch Gradient descent
- Linear Classifier
- Logistic Regression

PyTorch

- Autograd
- Built-in Modules
- Examples

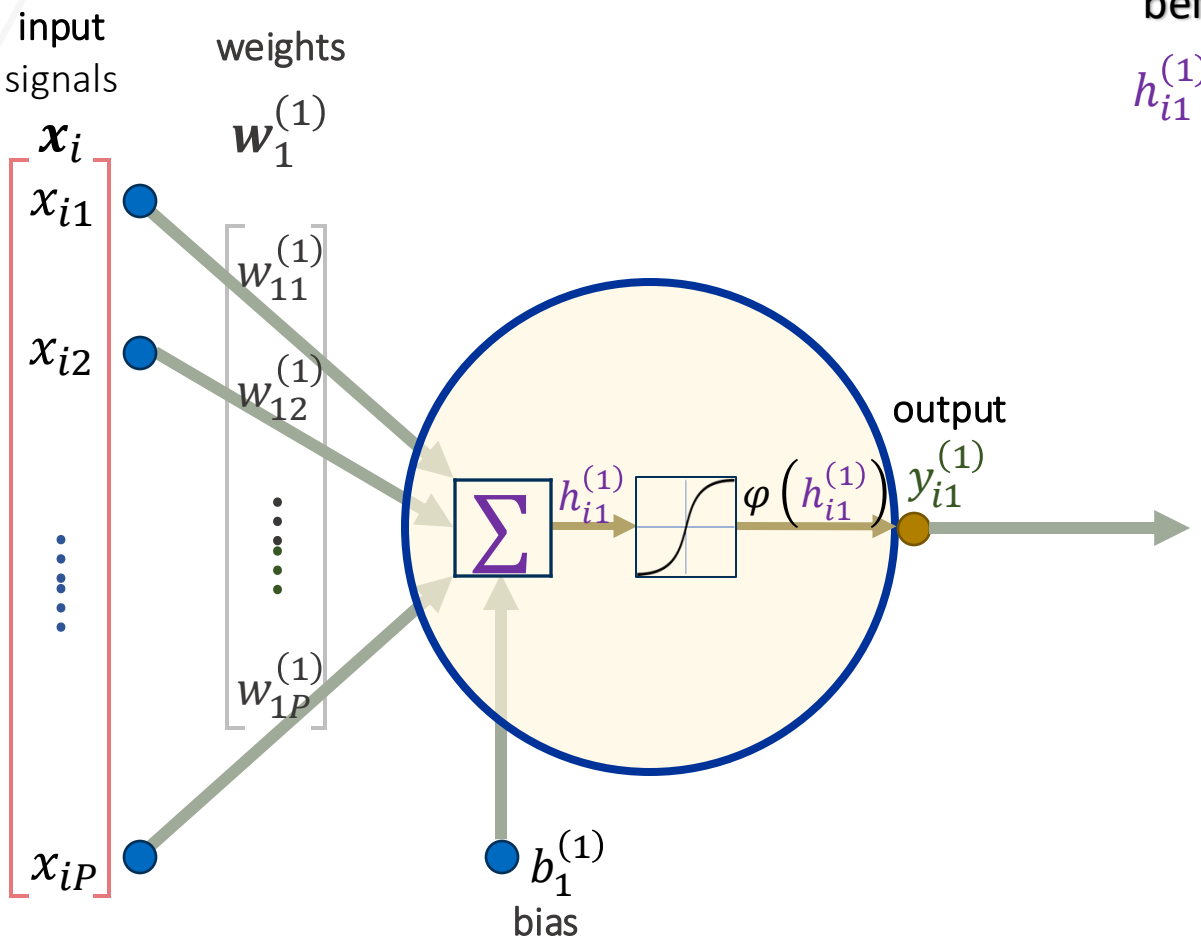
Classifier Comparison

Methods	Assumptions on Feat. Dist.	Feat. Normalization	Cost Function	Regularization	Linear Classifier	Prob. View of Prediction	Generative/Discriminative	Parametric/ Non-parametric	Overfitting
Logistic Regression	No	Required	BCE (convex)	Additional term	Linear	Yes	Discriminative	Parametric	Not often
K Nearest Neighbors	No	Required	N/A	N/A	Non-linear	N/A	Discriminative	Non-parametric	when k is too small
Decision Trees	No	Not Required	N/A	N/A	Non-linear	N/A	Discriminative	Non-parametric	with large depth
Support Vector Machines	No	Required	Hinge (convex)	C (control robustness)	Linear/ Non-linear(kernel)	N/A	Discriminative	Parametric	Not often
Naïve Bayes	Conditional independent	Not Required	N/A	N/A	Non-linear /Linear (Gaussian)	Yes	Generative	Parametric	Not often
Artificial Neural Networks	No	Required	Non-convex	Additional term	Non-linear	Yes	Discriminative/ Generative	Parametric	with many layers

Artificial Neural Networks

The Perceptron

- Single-layer Perceptron (SLP)



weighted input to the neuron before activation φ

$$h_{i1}^{(1)} = \left(\boldsymbol{w}_j^{(1)}\right)^T \boldsymbol{x}_i + b_1^{(1)}$$

output of the neuron after activation φ

$$\varphi\left(h_{i1}^{(1)}\right)$$

The simplest form of the perceptron uses linear activation $\varphi\left(h_{i1}^{(1)}\right) = h_{i1}^{(1)}$, the outputs are binary (i.e. $y_{i1}^{(1)} \in \{1, -1\}$):

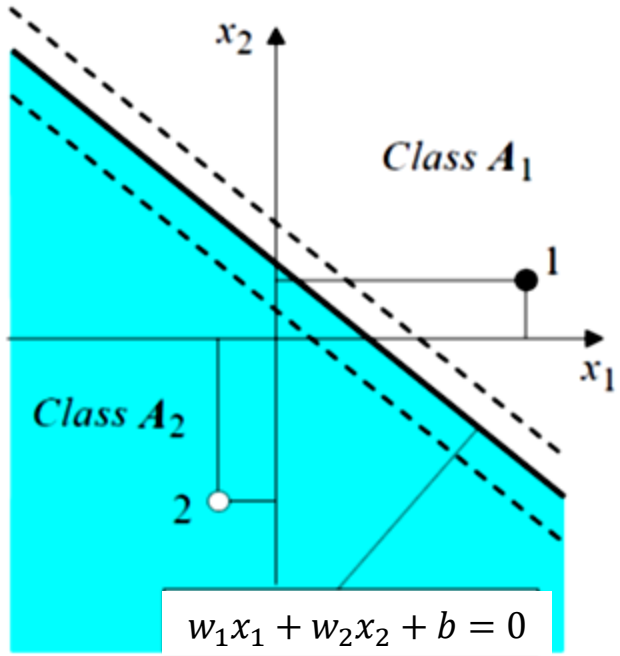
$$y_{i1}^{(1)} = \begin{cases} +1, if \left(\boldsymbol{w}_j^{(1)}\right)^T \boldsymbol{x}_i + b_1^{(1)} \geq 0 \\ -1, if \left(\boldsymbol{w}_j^{(1)}\right)^T \boldsymbol{x}_i + b_1^{(1)} < 0 \end{cases}$$

The above model applies to linearly separable data

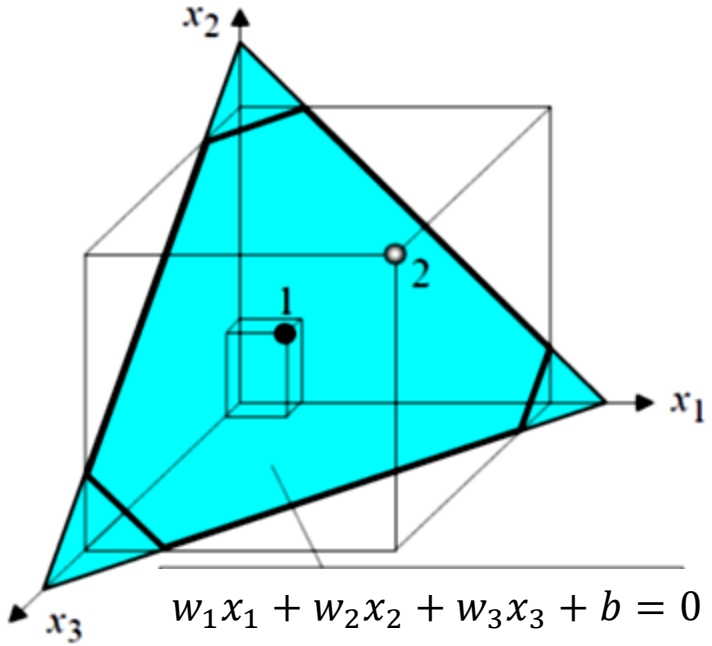
Artificial Neural Networks

The SLP Perceptron Model

- Linear separability in the perceptron



(a) Two-input perceptron.



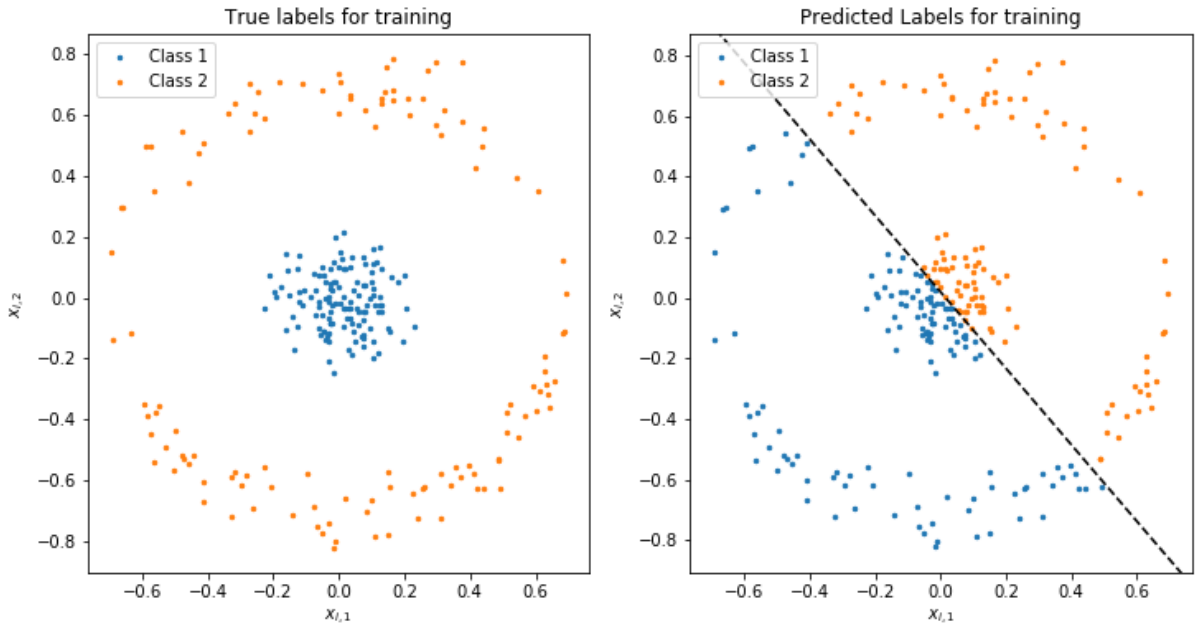
(b) Three-input perceptron.

Artificial Neural Networks

Hidden Layers

- Single neurons with linear activation fail to classify non-linearly separable dataset

- Non-linearly separable data requires
  - Non-linear Activation
  - Multi-layer networks of neurons (Multi-layer perceptron)

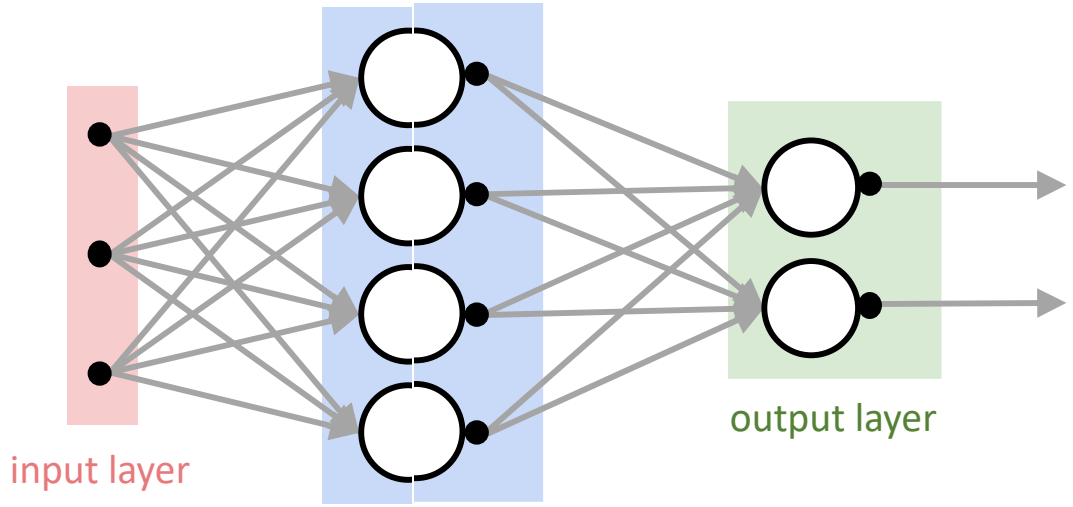


SLP with linear activation classifying non-linearly separable data

Artificial Neural Networks

Hidden Layers

- A NN with one hidden layer can represent:
  - any bounded continuous function (to some arbitrary ε) [Universal Approximation Theorem, Cybenko 1989]
  - any Boolean Function, but it requires 2^k hidden units for 1K inputs



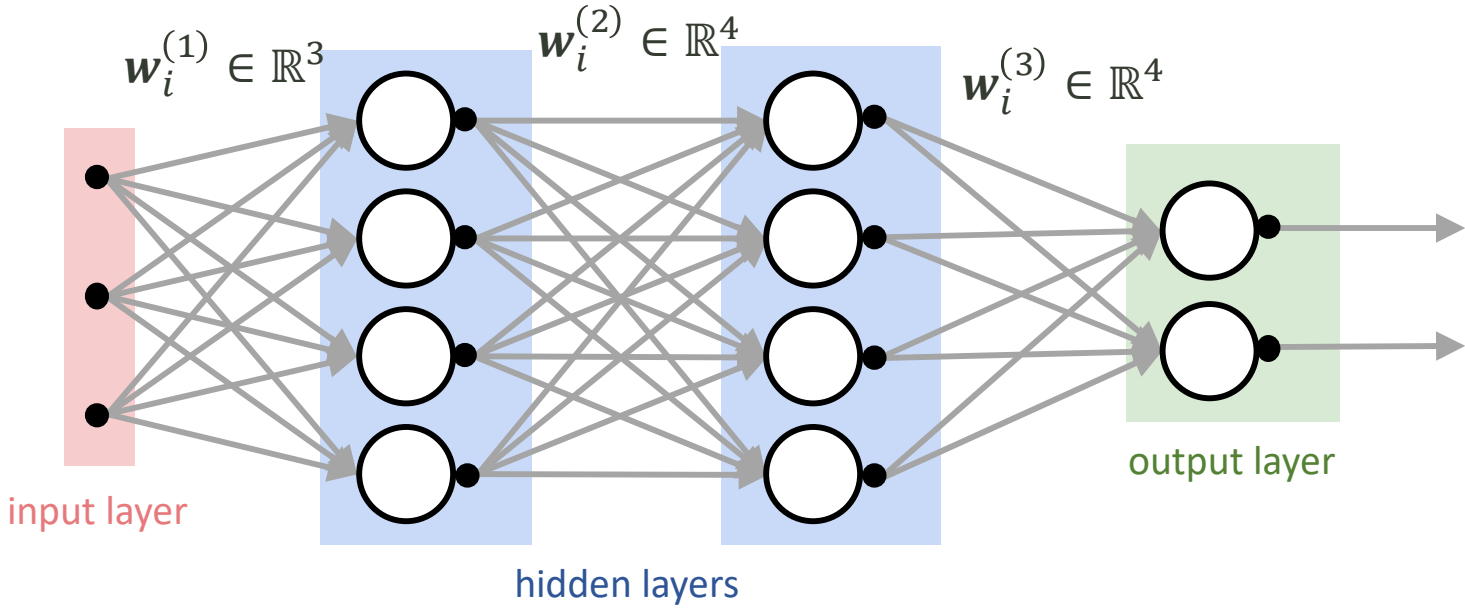
Artificial Neural Networks

Multiple Hidden Layers

Layer-wise organization: MLP consists of *fully-connected* layers in which neurons between two adjacent layers are fully pairwise connected, while neurons within a single layer share no connections.

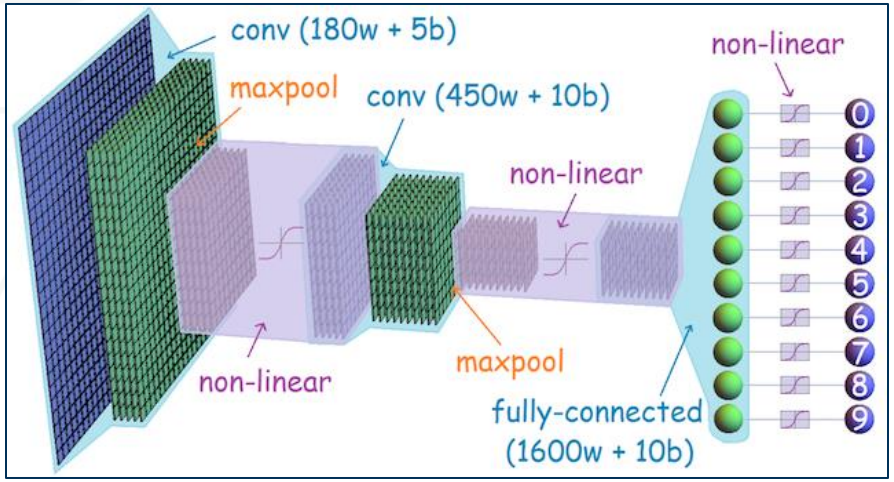
Sizing MLP: the number of neurons or the number of parameters (more commonly)

- The network has 4 + 4 + 2=10 neurons, [3 x 4] + [4 x 4] + [4 x 2] = 12 + 16 + 8 = 36 weights and 4 + 4 + 2 = 10 biases, for a total of 46 learnable parameters.



Overview

In this Lecture..



Review: Artificial Neural Networks

- Activation Function
- Perceptron Network
- Multi-layer ANN

Backpropagation

- Feedforward and Backward Error Propagation
- Learning Algorithm
- Image Classification using ANNs
- Mini-batch Gradient descent
- Linear Classifier
- Logistic Regression

PyTorch

- Autograd
- Built-in Modules
- Examples

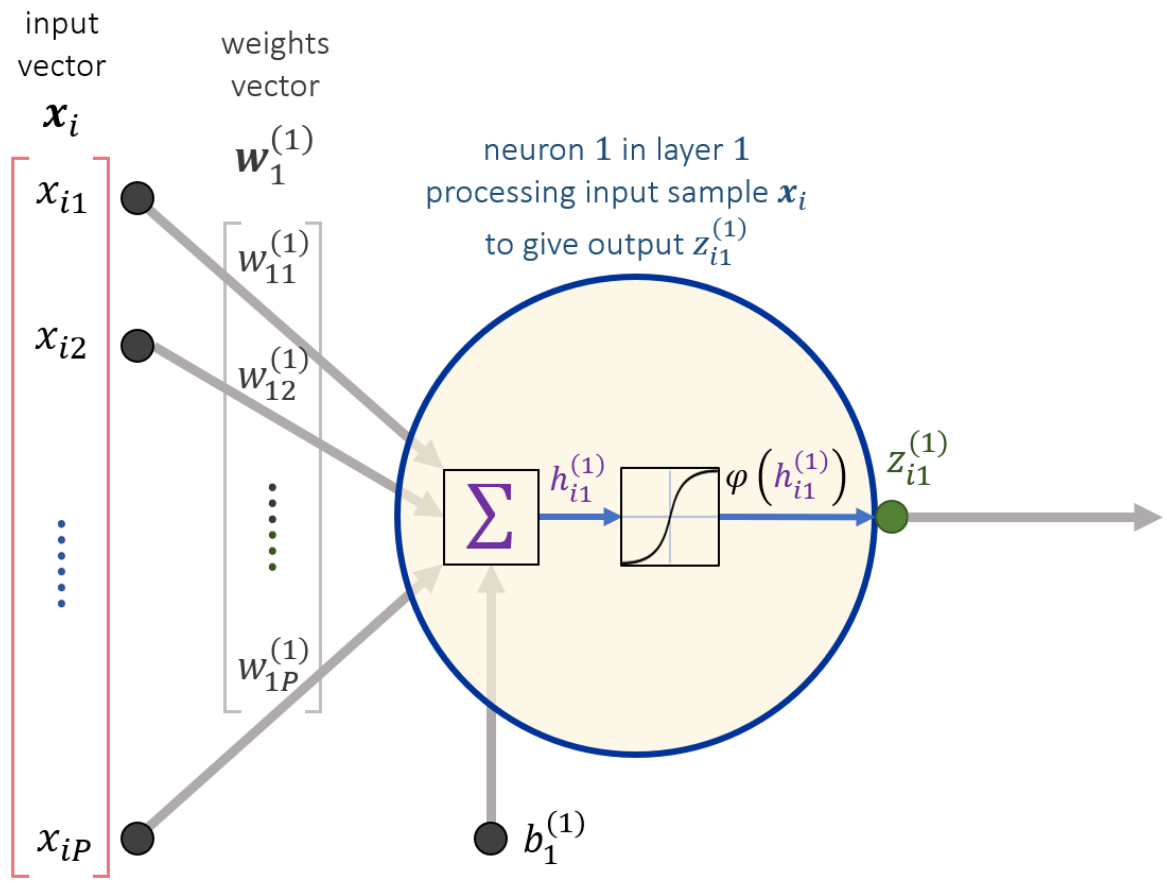
Backpropagation

Notations

Neuron 1 in layer 1:

- Let $\boldsymbol{w}_1^{(1)} = \begin{bmatrix} w_{11}^{(1)} \\ w_{12}^{(1)} \\ \vdots \\ w_{1P}^{(1)} \end{bmatrix}$ be the neuron's **weights** vector, where P is the number of features in the input layer
- Let $b_1^{(1)}$ be the neuron's **bias**
- Let $h_{i1}^{(1)} = \left(\boldsymbol{w}_j^{(1)}\right)^T \boldsymbol{x}_i + b_1^{(1)}$ be the weighted input to the neuron **before activation**
- Let $z_{i1}^{(1)} = \varphi\left(h_{i1}^{(1)}\right)$ be the **activation associated with** the neuron using any activation function φ such as the sigmoid function, σ :

$$z_{i1}^{(1)} = \varphi\left(h_{i1}^{(1)}\right) = \sigma\left(h_{i1}^{(1)}\right) = \frac{1}{1 + e^{-h_{i1}^{(1)}}}$$

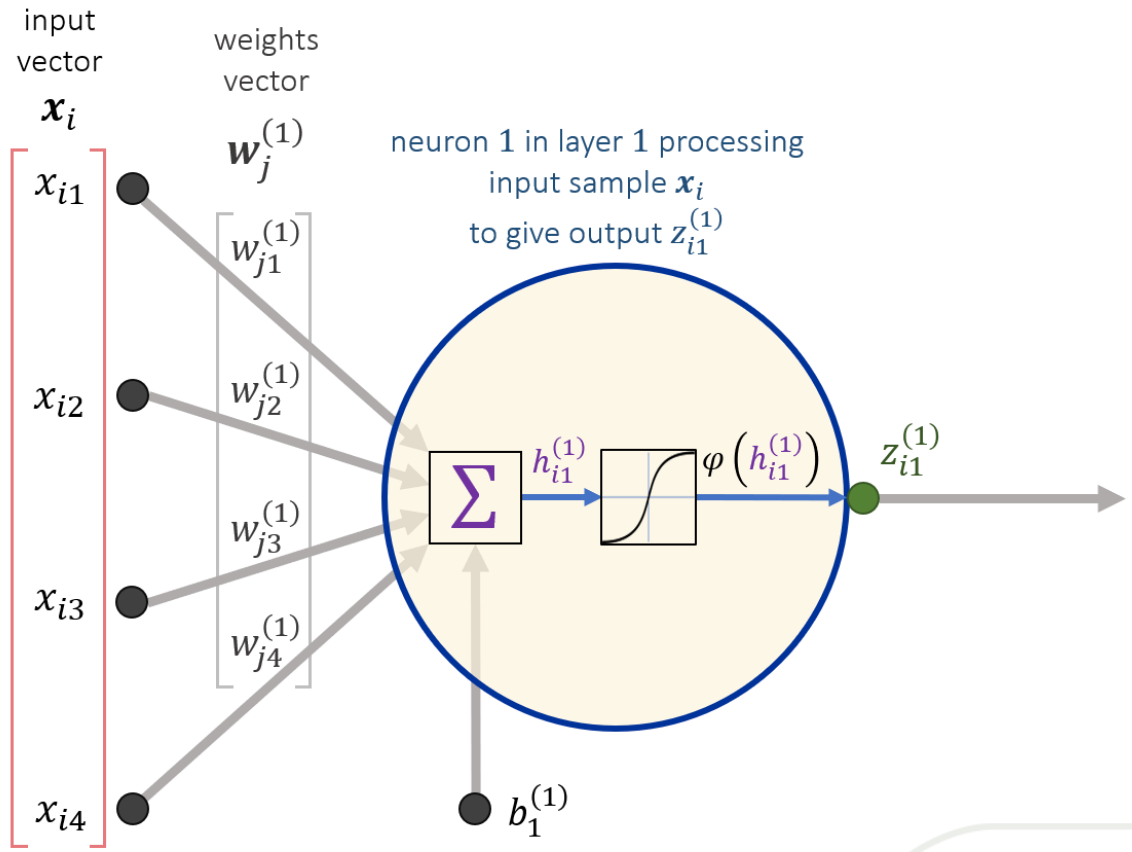


Backpropagation

Notations

Example: The 1^{st} neuron in layer 1, processing a 4-dimensional input

- Input: $\boldsymbol{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \\ x_{i4} \end{bmatrix}$, Weights: $\boldsymbol{w}_1^{(1)} = \begin{bmatrix} w_{11}^{(1)} \\ w_{12}^{(1)} \\ w_{13}^{(1)} \\ w_{14}^{(1)} \end{bmatrix}$
- Bias: $b_1^{(1)}$
- Weighted input: $h_{i1}^{(1)} = \left(\boldsymbol{w}_1^{(1)}\right)^T \boldsymbol{x}_i + b_1^{(1)}$
- Output: $z_{i1}^{(1)} = \varphi\left(h_{i1}^{(1)}\right) = \sigma\left(h_{i1}^{(1)}\right) = \frac{1}{1+e^{-h_{i1}^{(1)}}}$



Backpropagation

Notations

General notation, neuron j in layer k:

- Let $\boldsymbol{w}_j^{(k)} = \begin{bmatrix} w_{j1}^{(k)} \\ w_{j1}^{(k)} \\ \vdots \\ w_{jP^{(k-1)}}^{(k)} \end{bmatrix}$ be the neuron's weights vector, where $P^{(k-1)}$ is the number of neurons in the previous layer $k-1$

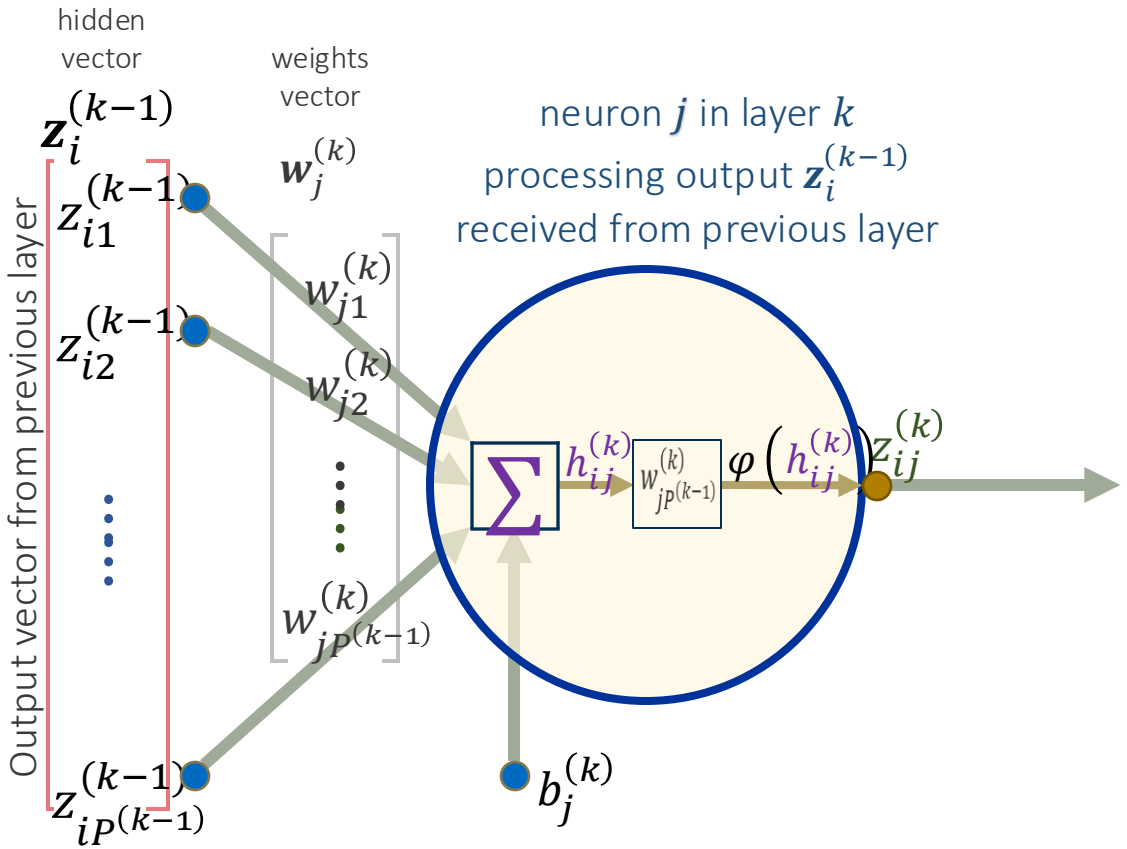
  ($P^{(0)} = P$, the number of features in the input vector)

- Let $b_j^{(k)}$ be the neuron's bias

- Let $h_{ij}^{(k)} = \left(\boldsymbol{w}_j^{(k)}\right)^T \boldsymbol{z}_i^{(k-1)} + b_j^{(k)}$ be the weighted input to the neuron before activation

- Let $z_{ij}^{(k)} = \varphi\left(h_{ij}^{(k)}\right)$ be the activation associated with the neuron using any activation function φ such as the sigmoid function, σ :

$$z_{ij}^{(k)} = \varphi\left(h_{ij}^{(k)}\right) = \sigma\left(h_{ij}^{(k)}\right) = \frac{1}{1 + e^{-h_{ij}^{(k)}}}$$

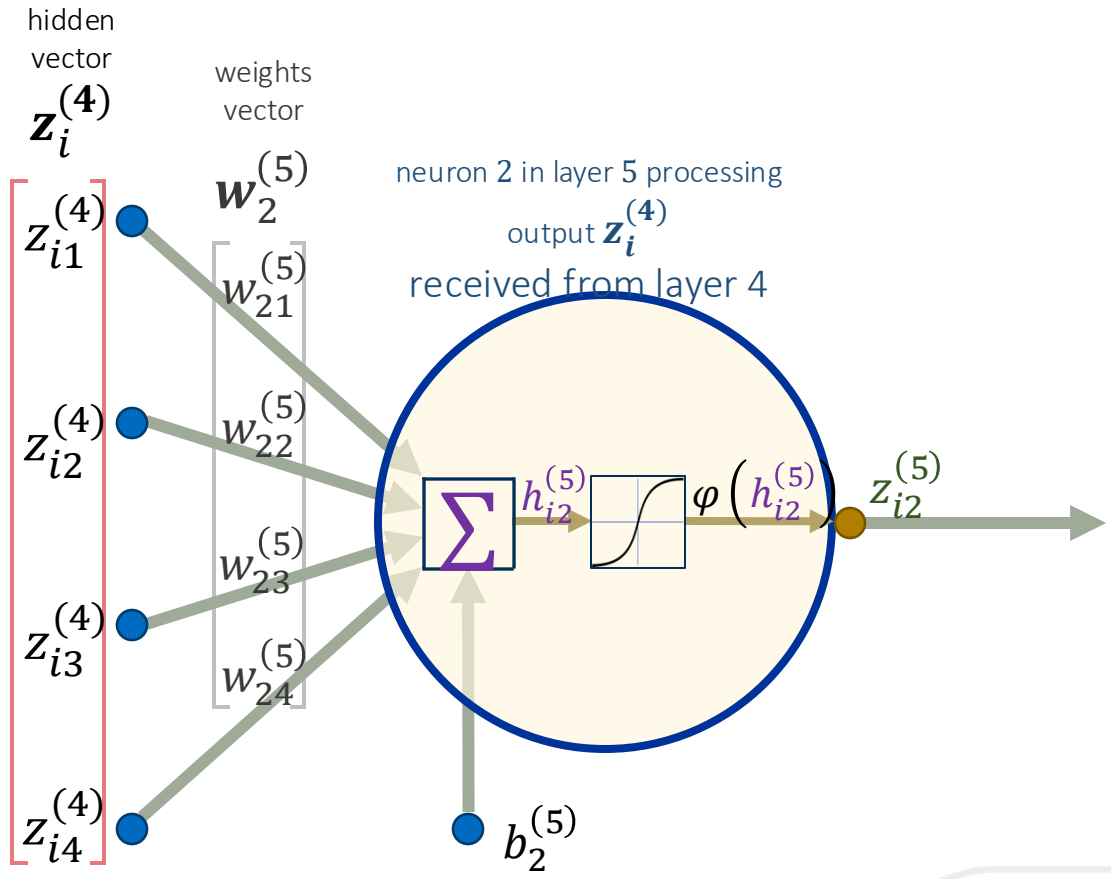


Backpropagation

Notations

Example: The 2^{nd} neuron in layer 5, processing a 4-dimensional input

- Input: $\boldsymbol{z}_i^{(4)} = \begin{bmatrix} z_{i1} \\ z_{i2} \\ z_{i3} \\ z_{i4} \end{bmatrix}$, Weights: $\boldsymbol{w}_2^{(5)} = \begin{bmatrix} w_{21}^{(5)} \\ w_{22}^{(5)} \\ w_{23}^{(5)} \\ w_{24}^{(5)} \end{bmatrix}$
- Bias: $b_2^{(5)}$
- Weighted input: $h_{i2}^{(5)} = \left(\boldsymbol{w}_2^{(5)}\right)^T \boldsymbol{z}_i^{(4)} + b_2^{(5)}$
- Output: $z_{i2}^{(5)} = \varphi\left(h_{i2}^{(5)}\right) = \sigma\left(h_{i2}^{(5)}\right) = \frac{1}{1+e^{-h_{i2}^{(5)}}}$



Backpropagation

Optimum Weights via Gradient Descent

- For a simple **linear** perceptron model, we can try to find the optimum $\boldsymbol{w}$ and b by minimizing a **Least Squares** cost function
- Let $L(\boldsymbol{\theta})$ be the MSE loss function defined over the entire dataset such that:

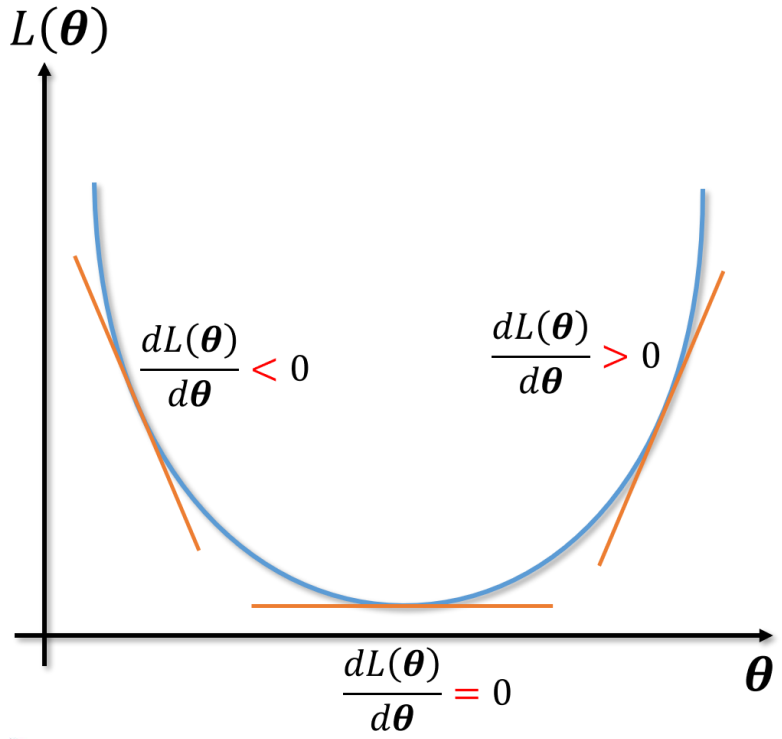
$$L(\boldsymbol{\theta}) = MSE(\boldsymbol{X}, \boldsymbol{\theta}) = \frac{1}{N}\left\|\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right\|_F^2$$

where $\boldsymbol{Y}$ is a matrix of all target outputs $\boldsymbol{y}$ and $\widehat{\boldsymbol{Y}}$ is a matrix of all predicted outputs $\hat{\boldsymbol{y}}$, and $\|\boldsymbol{X}\|_{\boldsymbol{F}}^{\boldsymbol{2}} = Tr(\boldsymbol{X}\boldsymbol{X}^T)$ is the Frobenius norm of a matrix which is the square root of the sum of the absolute squares of its elements.

- Let $\boldsymbol{\theta}^*$ be the set of optimum weights and biases such that:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, L(\boldsymbol{\theta})$$

- Considering the change in the loss function with respect to $\boldsymbol{\theta}$, the loss function is minimum when $\frac{dL(\boldsymbol{\theta})}{d\boldsymbol{\theta}} = 0$.
- Finding the $\boldsymbol{\theta}^*$ where the loss is minimum can be achieved either by:
  - Solving the Normal Equation
  - Training by Gradient Descent



In general, cost functions of NNs are non-convex with flat areas, and many saddle points.

Backpropagation

Optimum Weights for a Single Neuron

- Recall that $\boldsymbol{X} \in \mathcal{R}^{N\times P}$, $\mathbf{Y} \in \mathcal{R}^{N\times K}$, $\boldsymbol{W} \in \mathcal{R}^{P\times K}$, where we have *N* samples, *P* features, and *K* output classes per sample
- To simplify the math, let the bias $b_1^{(1)} = 0$, thus $\boldsymbol{\theta} = \boldsymbol{W}$ and $\widehat{\boldsymbol{Y}} = \boldsymbol{XW}$
- Assume that we use MSE as the loss function

$$L(\boldsymbol{\theta}) = MSE(\boldsymbol{X}, \boldsymbol{\theta}) = \frac{1}{N}\left\|\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right\|_F^2$$

where $\|\boldsymbol{X}\|_{\boldsymbol{F}}^{\boldsymbol{2}} = Tr(\boldsymbol{XX}^T)$ is the Frobenius norm of a matrix which is the square root of the sum of the absolute squares of its elements.

- Its derivative with respect to the weights $\boldsymbol{\theta}$ is:

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}} = \frac{1}{N}\frac{\partial}{\partial \boldsymbol{W}}\left(Tr\left\{\left(\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right)^{\boldsymbol{T}}\left(\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right)\right\}\right)$$

- The loss is minimum when $\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 0$. That is:

$$\boldsymbol{W} = (\boldsymbol{X}^T\boldsymbol{X})^{-\boldsymbol{1}}\boldsymbol{X}^T\boldsymbol{Y}$$

- The above is the Normal Form which gives a direct solution for the weight vector $\boldsymbol{\theta}$
- However, this method can be inefficient
- Gradient Descent is an iterative method that can find optimal $\boldsymbol{\theta}$ more efficiently.

Backpropagation

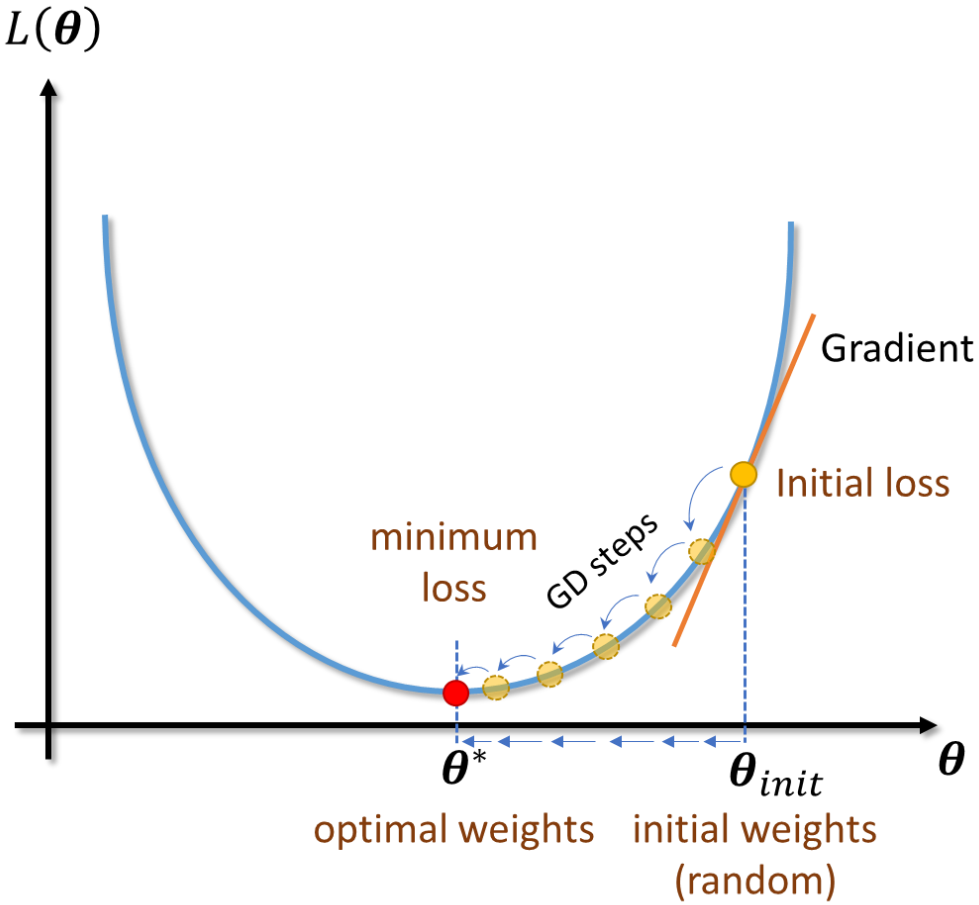
Optimum Weights for a Single Neuron

- An iterative process where the neuron learns iteratively from training samples to find the optimum weights at which the loss function reaches its minimum.
- The basic idea is to compute the gradient (derivative) of the loss function in a sequence of epochs (intake of the dataset) and update the weights in proportion to the gradient according to the update rule:

$$\boldsymbol{W}(t+1) = \boldsymbol{W}(t) - \alpha \frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}$$

$$= \boldsymbol{W}(t) - \alpha \frac{2}{N} \boldsymbol{X}^{\boldsymbol{T}} \left(\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right)$$

where $t+1$ and t indicate the new and current epochs, respectively, and α is a predetermined learning rate usually smaller than 0.5



Backpropagation

Optimum Weights for a Single Neuron

- Accounting for non-zero bias,

$$\widehat{\boldsymbol{Y}} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{o}\boldsymbol{b}^T$$

where $\boldsymbol{o} \in \mathbb{R}^{N\times 1} = \mathbf{1}_N$ is a vector of ones

- Again, using MSE as the loss function

$$L(\boldsymbol{\theta}) = MSE(\boldsymbol{X}, \boldsymbol{\theta}) = \frac{1}{N}\left\|\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right\|_F^2$$

- Its derivative with respect to the bias $\boldsymbol{b}$ is:

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{b}} = \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{b}} = \frac{1}{N}\frac{\partial}{\partial \boldsymbol{b}}\left(Tr\left\{\left(\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right)\left(\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right)^T\right\}\right) = \frac{2}{N}\boldsymbol{X}^{\boldsymbol{T}}\left(\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right)\boldsymbol{o}$$

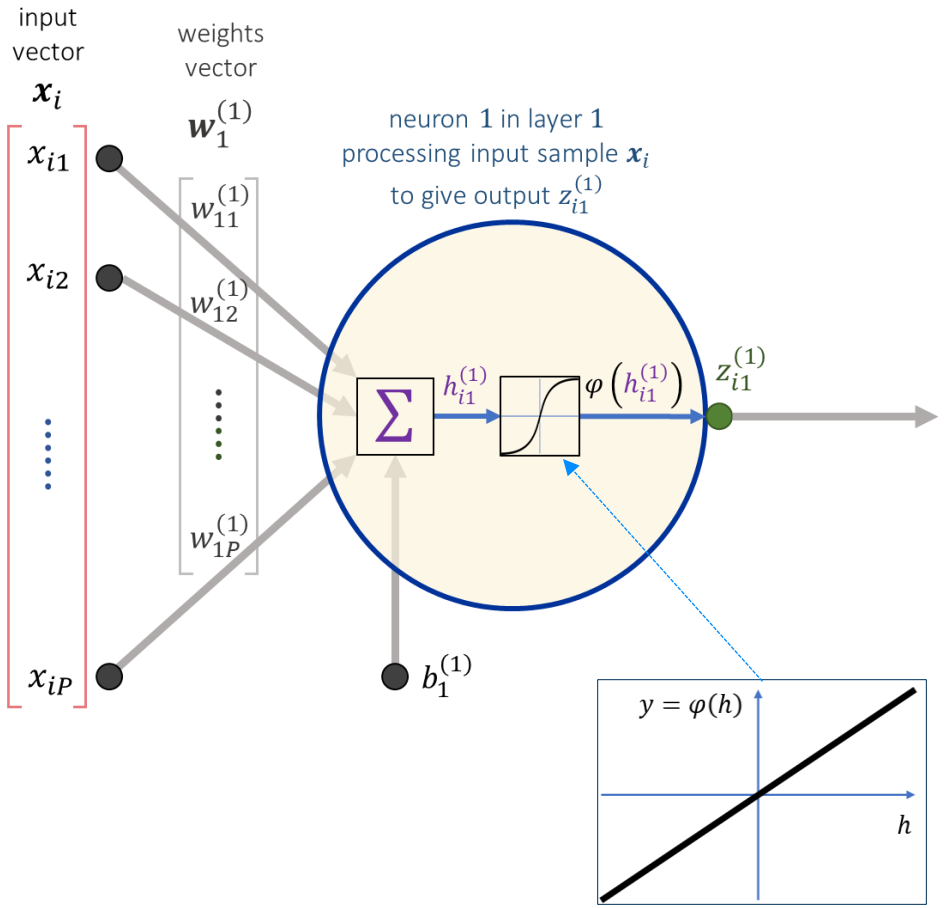
- Thus, the update rule for the Bias is:

$$\boldsymbol{b}(\mathrm{t}+1) = \boldsymbol{b}(\mathrm{t}) - \alpha\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{b}} = \boxed{\boldsymbol{b}(\mathrm{t}) - \alpha\frac{2}{N}\left(\widehat{\boldsymbol{Y}}^T - \boldsymbol{Y}^T\right)\boldsymbol{o}}$$

Backpropagation

Optimum Weights for a Single Neuron (GD)

- **Steps**:
  1. Initialize the weights and the bias
  2. While no convergence condition is met
     i. Calculate the predicted output matrix: $\widehat{\boldsymbol{Y}} = \boldsymbol{XW} + \boldsymbol{ob}^T$
     ii. Update weights: $\boldsymbol{W}(t+1) = \boldsymbol{W}(t) - \alpha \frac{2}{N}\left(\widehat{\boldsymbol{Y}}^T - \boldsymbol{Y}^T\right)\boldsymbol{X}$
     iii. Update bias: $\boldsymbol{b}(\mathrm{t}+1) = \boldsymbol{b}(\mathrm{t}) - \alpha \frac{2}{N}\left(\widehat{\boldsymbol{Y}}^T - \boldsymbol{Y}^T\right)\boldsymbol{o}$
- **Convergence**:
  - The epoch loss $\frac{1}{N}\left\|\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right\|_F^2$ is less than a threshold γ, or
  - A predetermined number of epochs have been completed.
- **Notes**:
  - bias $\boldsymbol{b}$ is updated similarly to the weights $\boldsymbol{W}$
  - This is *Batch GD*; two more variants are also used.



linear activation function

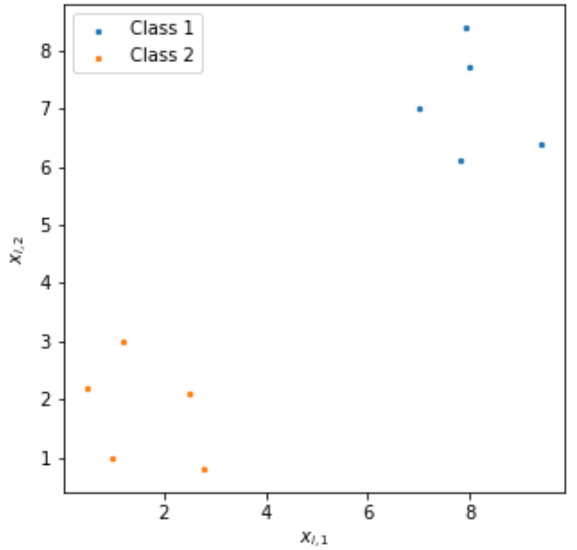
Backpropagation

Optimum Weights for a Single Neuron (GD)

Setup:

- Input: $\boldsymbol{X} = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \\ \vdots & \vdots \\ x_{10,1} & x_{10,2} \end{bmatrix}, \boldsymbol{Y} = \begin{bmatrix} y_{1,1} \\ y_{2,1} \\ \vdots \\ y_{10,1} \end{bmatrix}$
- Weights: $\boldsymbol{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \end{bmatrix}$
- Bias: $\boldsymbol{b}_1^{(1)} = \begin{bmatrix} b_1^{(k)} \end{bmatrix}$
- Output: $\widehat{\boldsymbol{Y}} = \boldsymbol{X}(\boldsymbol{W}^{(1)}) + \boldsymbol{o}(\boldsymbol{b}_1^{(1)})^{\boldsymbol{T}} = \begin{bmatrix} \hat{y}_{1,1} \\ \hat{y}_{2,1} \\ \vdots \\ \hat{y}_{10,1} \end{bmatrix}$
- MSE loss: $\frac{1}{N}\left\|\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right\|_F^2$

x_{i1}	x_{i2}	y_i
1.0	1.0	1
9.4	6.4	0
2.5	2.1	1
8.0	7.7	0
0.5	2.2	1
7.9	8.4	0
7.0	7.0	0
2.8	0.8	1
1.2	3.0	1
7.8	6.1	0



A dataset for perceptron classification

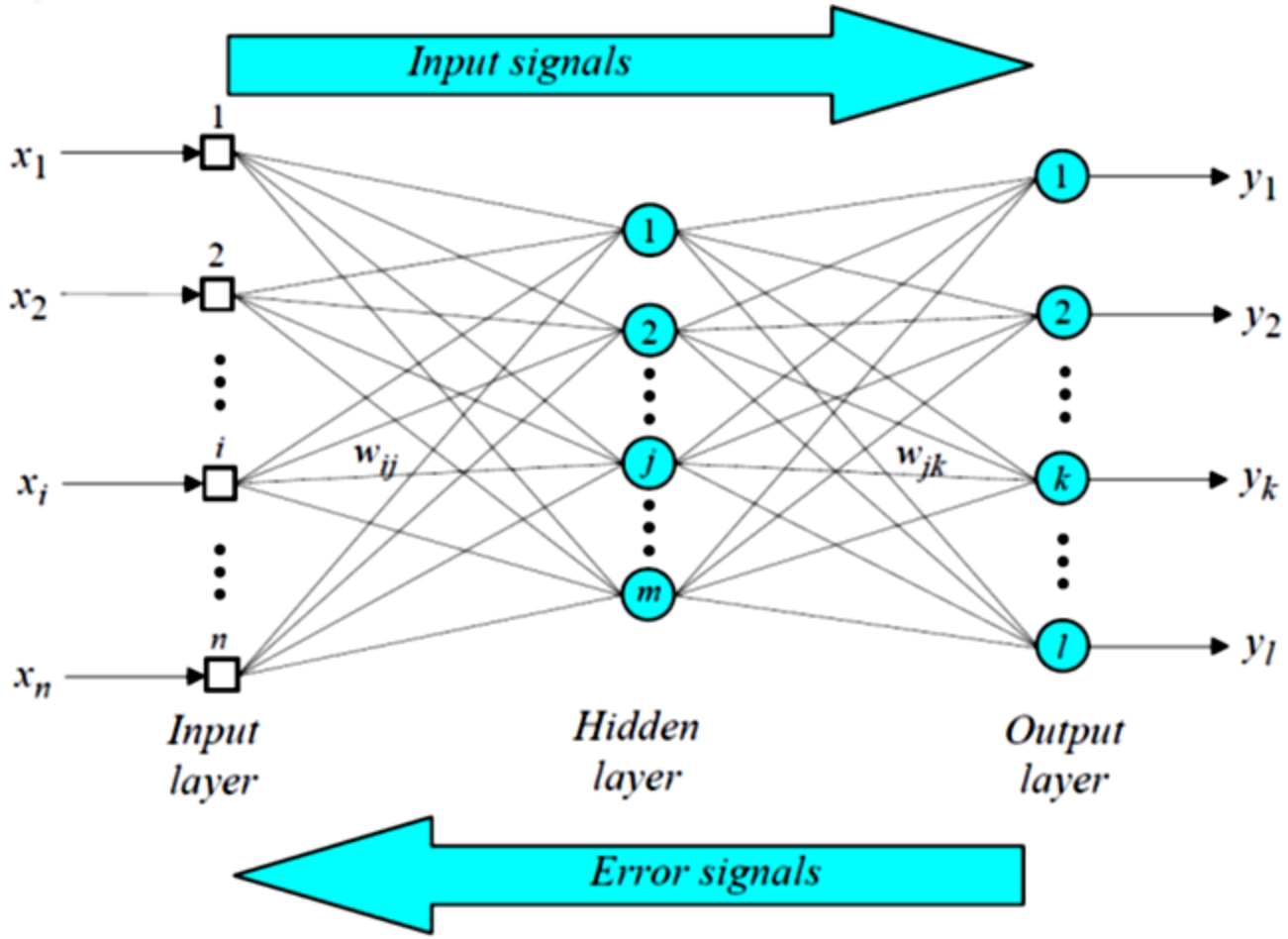
Gradient Descent:

Updating Weights: $\frac{\partial \boldsymbol{L}(\boldsymbol{\theta})}{\partial \boldsymbol{W}^{(1)}} = \frac{2}{N}\left(\widehat{\boldsymbol{Y}}^T - \boldsymbol{Y}^T\right)\boldsymbol{X}$

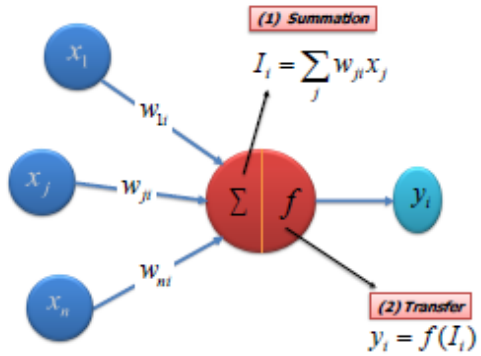
Updating Bias: $\frac{\partial \boldsymbol{L}(\boldsymbol{\theta})}{\partial \boldsymbol{b}_1^{(1)}} = \frac{2}{N}\left(\widehat{\boldsymbol{Y}}^T - \boldsymbol{Y}^T\right)\boldsymbol{o}$

Backpropagation

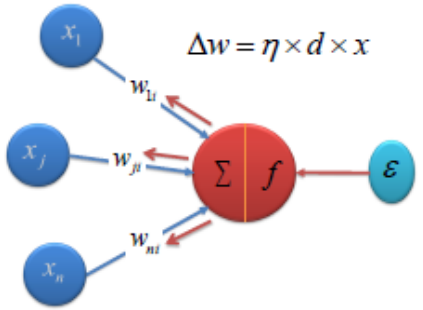
Terminologies



Feedforward Input Data

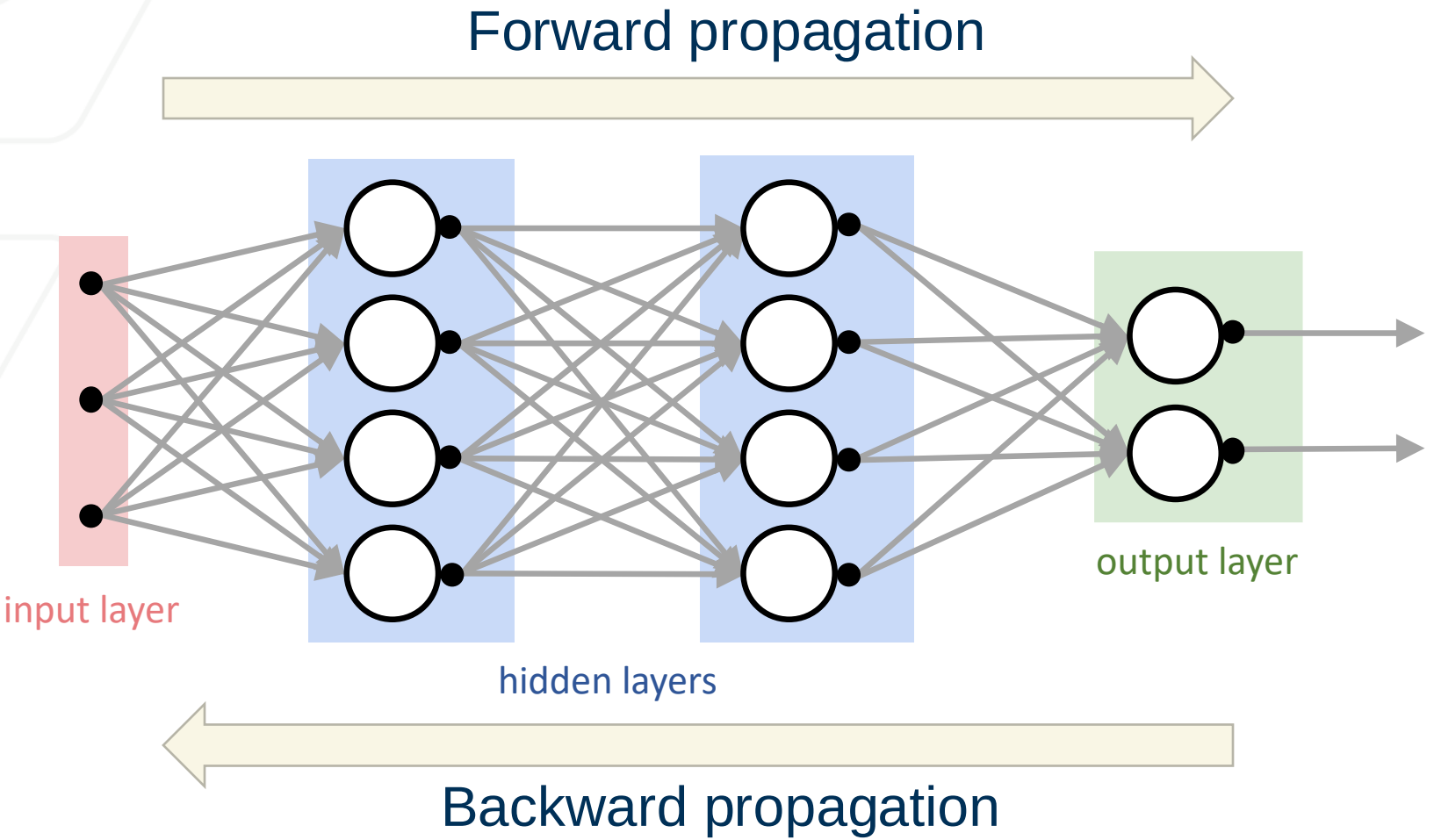


Backward Error Propagation

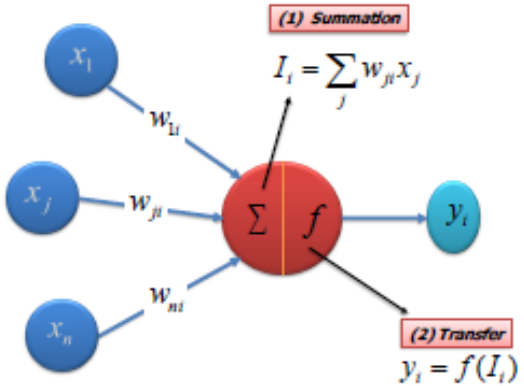


Backpropagation

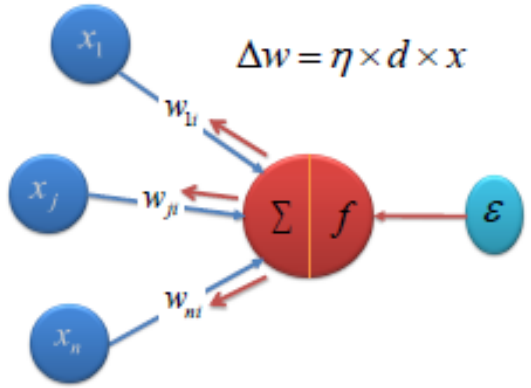
Terminologies



Feedforward Input Data



Backward Error Propagation

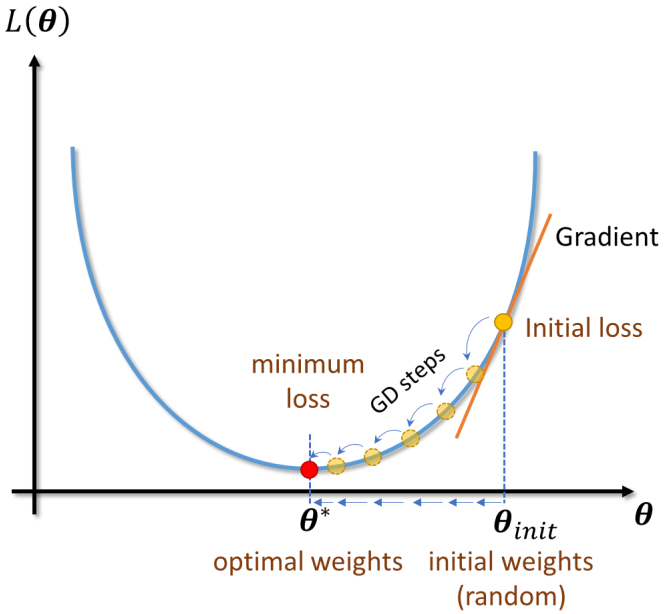


Neural Networks

Quick Review of Derivatives

Derivative of a function, $f(\boldsymbol{\theta})$:

- how fast *f* changes around $\boldsymbol{\theta}$
- will *f* increase or decrease if we increase $\boldsymbol{\theta}$
- is $\boldsymbol{\theta}$ higher or lower than it should be ($\boldsymbol{\theta}^*$); shall we make it bigger or smaller to be where it should be



$\frac{\partial}{\partial\theta} f(g(\theta)) = \frac{\partial}{\partial g} f(g(\theta)) \cdot \frac{\partial}{\partial\theta} g(\theta)$ chain rule

$\frac{\partial}{\partial\theta} \sum_i f_i(\theta) = \sum_i \frac{\partial}{\partial\theta} f_i(\theta)$ derivative of the sum is the sum of the derivatives

$\frac{\partial}{\partial\theta_k} \sum_i a_i f(\theta_i) = a_k \frac{\partial}{\partial\theta_k} f(\theta_k)$ derivative wrt one element of the sum collapses the sum

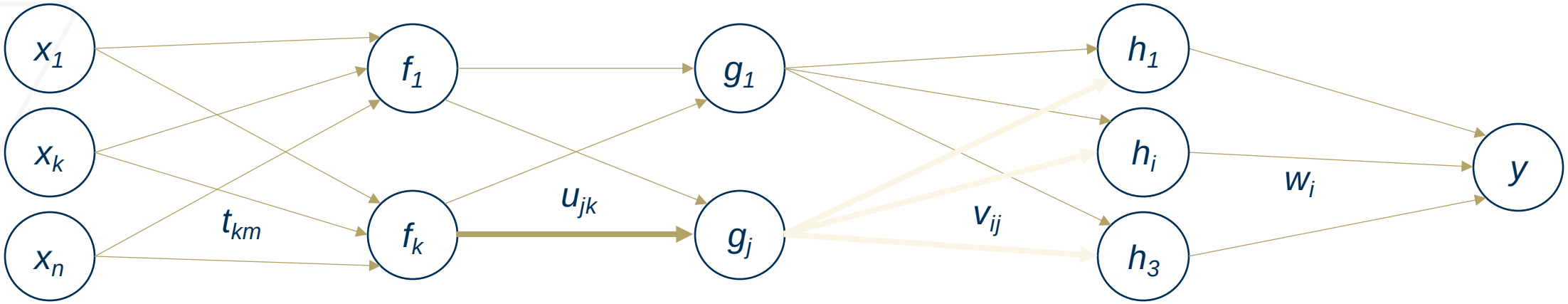
The sigmoid has a special property:

$$\sigma'(x) = \frac{\partial}{\partial x}(1+e^{-x})^{-1} = -(1+e^{-x})^{-2}(-e^{-x}) = \frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}} = \sigma(x)\big(1-\sigma(x)\big)$$

$$\frac{\partial}{\partial x}\sigma(f_x) = \sigma(f_x)\big(1-\sigma(f_x)\big)\frac{\partial}{\partial x} f_x$$

Neural Networks

Backpropagation

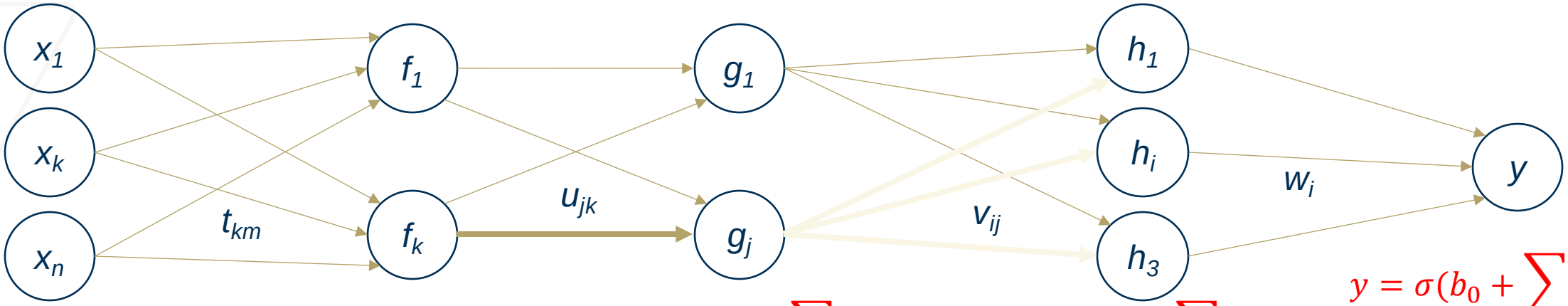


1. For a new sample $\boldsymbol{x} = [x_1, \ldots x_n]$
2. Feed forward: compute g_j based on units f_k *from the previous layer using* $g_j = \sigma(b_{j0} + \sum_k u_{jk} f_k)$
3. Predict y *as* y^*
4. Backpropagate error: $e = y - y^*$
    1. Determine whether we need g_j to be lower or higher (assuming we already determined the better values for h from a previous step) by $\frac{\partial e}{\partial g_j} = \sum_i \sigma'(h_i) v_{ij} \frac{\partial e}{\partial h_i}$, or in other words by determining how h_i will change as g_j changes AND by inspecting if h_i was too high or too low
    2. Update weights: update the weight u_{jk} that feeds g_j by $\frac{\partial e}{\partial u_{jk}} = \frac{\partial e}{\partial g_j} \sigma'(g_j) f_k$, or in other words by determining if we want g_j to be higher or lower AND by determining how g_j will change if u_{jk} becomes higher or lower; now update the weight with a learning rate multiplier of $\frac{\partial e}{\partial u_{jk}}$

Think of $\sigma'(h_i) v_{ij}$ as a scalar; if h is around 0, then changing g will impact h a lot; if h is close to 1 or 0, then this expression has no impact

Neural Networks

Backpropagation

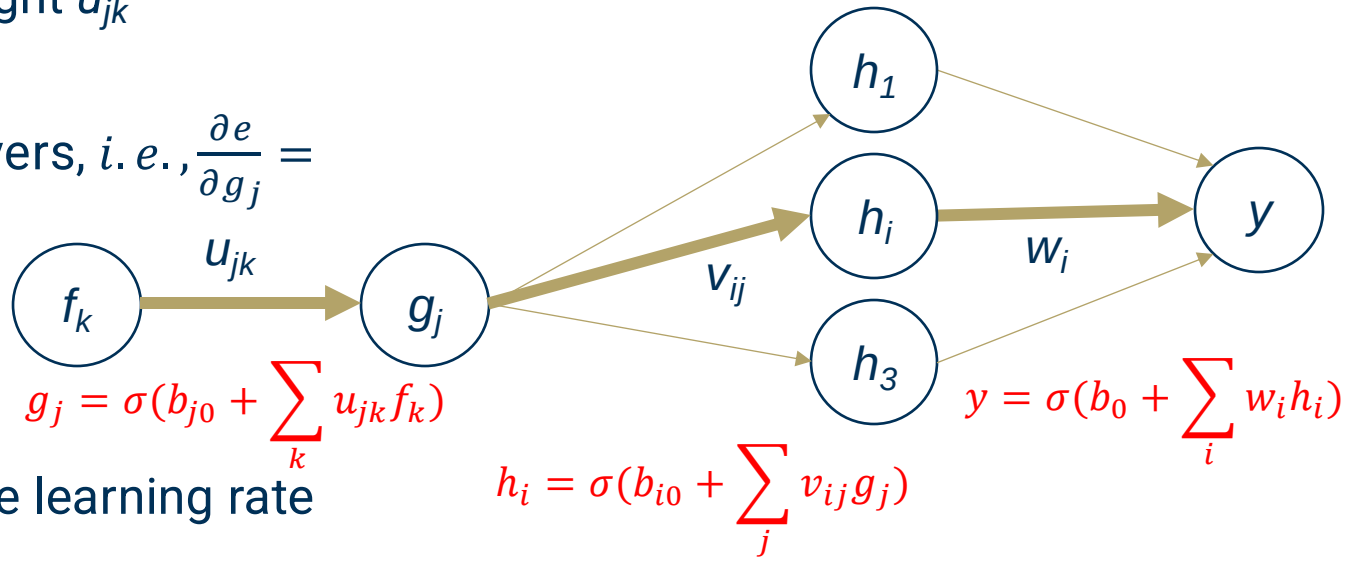


- $e = \frac{1}{2}(y - y^*)^2$
- $\frac{\partial e}{\partial h_i} = (y - y^*)\frac{\partial y}{\partial h_i} = (y - y^*)\, y(1-y) w_i$; the sum goes away because we are differentiating wrt to one of the elements
- $\frac{\partial e}{\partial g_j} = (y - y^*)\frac{\partial y}{\partial g_j} = (y - y^*)\, y(1-y) \sum_i w_i \frac{\partial h_i}{\partial g_j} = (y - y^*)\, y(1-y) \sum_i w_i h_i (1 - h_i) v_{ij}$
- By observing the nesting pattern, we can write $\frac{\partial e}{\partial g_j} = \sum_i h_i(1 - h_i) v_{ij} \frac{\partial e}{\partial h_i}$ and so on ; compare this equation with the one in 4.1 on the previous slide
- $\frac{\partial e}{\partial u} = (y - y^*)\frac{\partial y}{\partial u} = (y - y^*)\, y(1-y) \sum_i w_i h_i (1 - h_i) \sum_j \frac{\partial g_j}{\partial u} = (y - y^*)\, y(1-y) \sum_i w_i h_i (1 - h_i) v_{ij}\, g_j(1 - g_j) f_k = g_j(1 - g_j) f_k \frac{\partial e}{\partial g_j}$

Neural Networks

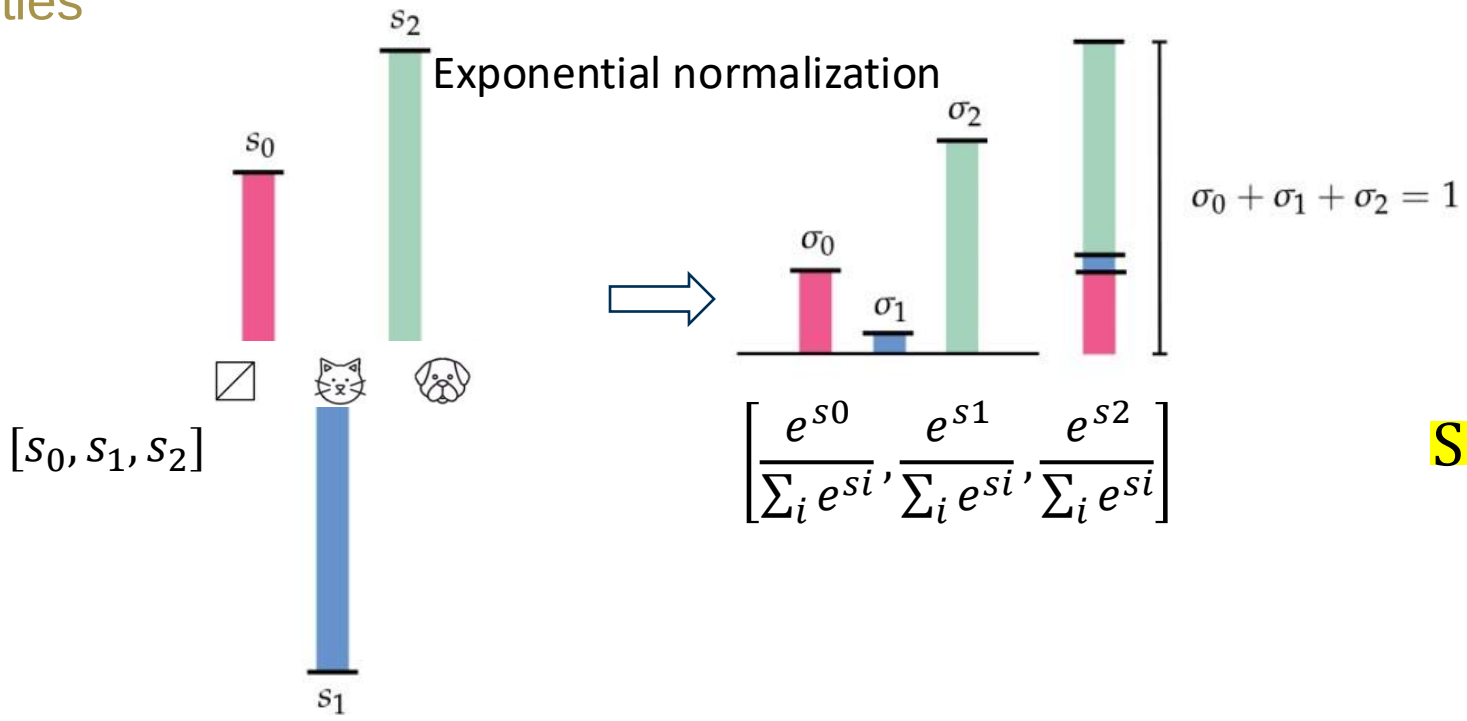
Backpropagation Summary

- **Feed forward propagate** the sample $\boldsymbol{x}$, compute activation outputs of neurons from previous layers and compute the predicted output y
- Compute the **backpropagation error**: $e = L(y, y^*)$, where L is the loss function
- Assume using sigmoid activation σ, for a weight u_{jk} compute $\frac{\partial e}{\partial u_{jk}}$ using **chain rule:**
  - Compute $\frac{\partial e}{\partial g_j}$ by carrying $\frac{\partial e}{\partial h_i}$ from later layers, $i.e., \frac{\partial e}{\partial g_j} = \sum_i h_i(1-h_i)v_{ij}\frac{\partial e}{\partial h_i}$
  - Compute $\frac{\partial e}{\partial u_{jk}} = g_j(1-g_j)f_k\frac{\partial e}{\partial g_j}$
- Update weights using **gradient descent**
  - $u_{jk}(t+1) = u_{jk}(t) - \alpha\frac{\partial e}{\partial u_{jk}}$, where α is the learning rate
- Iterate the above steps until the error e converges



Neural Networks

Softmax Probabilities



SoftMax activation:

$$f(s_i) = \frac{e^{s_i}}{\sum_i e^{s_i}}$$

- Suppose the raw outputs of a 3-class ANN is $\boldsymbol{s} = [s_0, s_1, s_2]$, where s_1<0. The raw outputs can not be interpreted as probability distribution.
- Exponentiating each element in $\boldsymbol{s}$ gives nonnegative values $[e^{s0}, e^{s1}, e^{s2}]$, maintaing the ordering of each element. If we normalize the the values by sum, we obtain the **discrete probability distributions**

$$\left[\frac{e^{s0}}{\sum_i e^{si}}, \frac{e^{s1}}{\sum_i e^{si}}, \frac{e^{s2}}{\sum_i e^{si}}\right]$$

of the model outputs. All entries now sum to 1

- This exponential normalization is referred to as SoftMax activation

- Suppose that we have numerical label values $y \in \{0,1,\ldots,C-1\}$. The One-hot encoded vectors for y takes the form:

$$y_p = 0 \longleftarrow \mathbf{y}_p = \begin{bmatrix} 1, 0, \cdots 0, 0 \end{bmatrix}$$

$$y_p = 1 \longleftarrow \mathbf{y}_p = \begin{bmatrix} 0, 1, \cdots 0, 0 \end{bmatrix}$$

$$\vdots$$

$$y_p = C-1 \longleftarrow \mathbf{y}_p = \begin{bmatrix} 0, 0, \cdots 0, 1 \end{bmatrix}.$$

- Each one-hot encoded vector is a *categorical label* with length C and contains all zeros except a 1 in the index equal to the true class (from 0 to C-1)

Neural Networks

Cross-entropy with Softmax

- Suppose for sample x_i with class p, a multi-class classifier predicts ***discrete probability distribution*** $\boldsymbol{f(s)} = [f(s_0), \dots, f(s_{c-1})]$ via exponential normalization.
- The desired prediction will be the one-hot vector $\boldsymbol{y}_i = [0, \dots 1, \dots 0]$

p-th entry

- Recall the cost function we used for Logistic Regression:
  - $\begin{cases} -\log(\sigma(\boldsymbol{x})), & if\ y = 1 \\ -\log\big(1 - \sigma(\boldsymbol{x})\big), & if\ y = 0 \end{cases}$
- Here the log cost can be written as $L = -\mathbf{y}_i^{\mathrm{T}} \log(f(\boldsymbol{s}))$

becomes binary cross entropy when $y_i \in \{[0,1], [1,0]\}$

Note that $\boldsymbol{y_i}$ is a one-hot vector, where $y_{ip} = 1$ and all other entries 0. The form simplifies:

$$L = -\log\big(f(s_p)\big)$$

- Plug in $f(s_j) = \frac{e^{s_j}}{\sum_j e^{s_j}}$, the form is precisely $L = -\log\left(\frac{e^{s_j}}{\sum_j e^{s_j}}\right)$

We then form a cost function by averaging the entire N data samples:

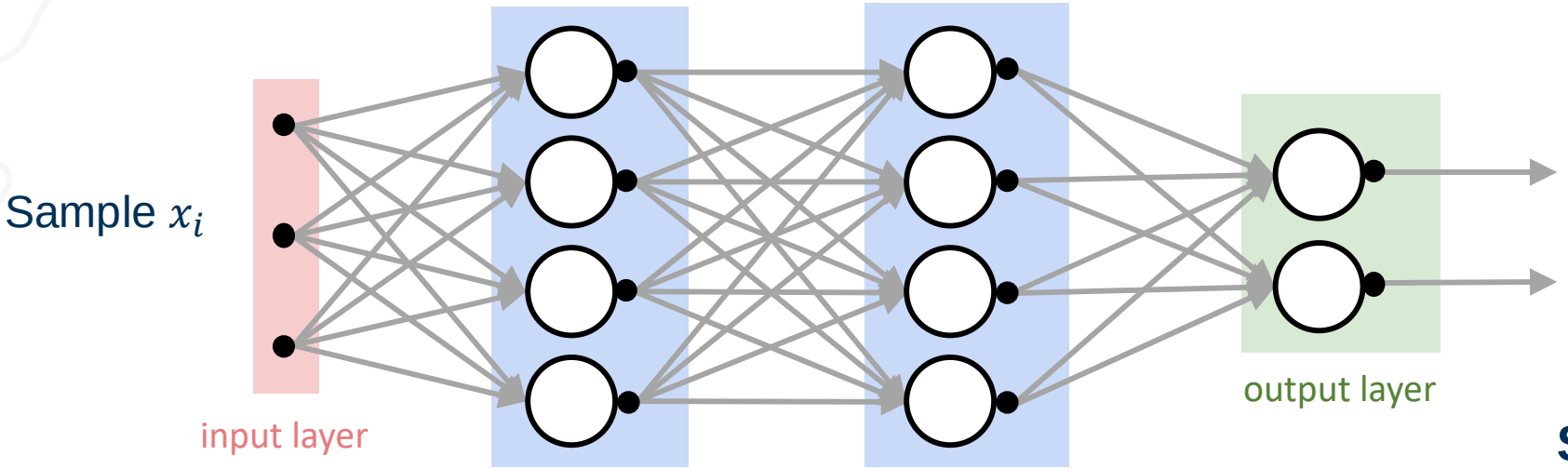
$$L = -\frac{1}{\mathrm{N}} \sum_i \log\left(\frac{e^{s_{ip}}}{\sum_j e^{s_{ij}}}\right)$$

Cross-entropy with SoftMax

Review: Artificial Neural Networks

Backpropagation Summary

During the inference, predict the class with the highest probability with SoftMax activation: $\text{argmax}_j\, f(y_{ij})$



Assume the raw outputs $y_i = [y_{i1}, y_{i2}, y_{i3}] = [3.2, 5.1, -1.7]$

Raw outputs

y_{i1}= 3.2
y_{i2}= 5.1
y_{i3}= -1.7

exp ⇨

unnormalized probabilities

24.5
164.0
0.18

normalize ⇨

probabilities

$f(y_{i1}) = 0.13$
$f(y_{i2}) = 0.87$
$f(y_{i3}) = 0.00$

SoftMax activation:

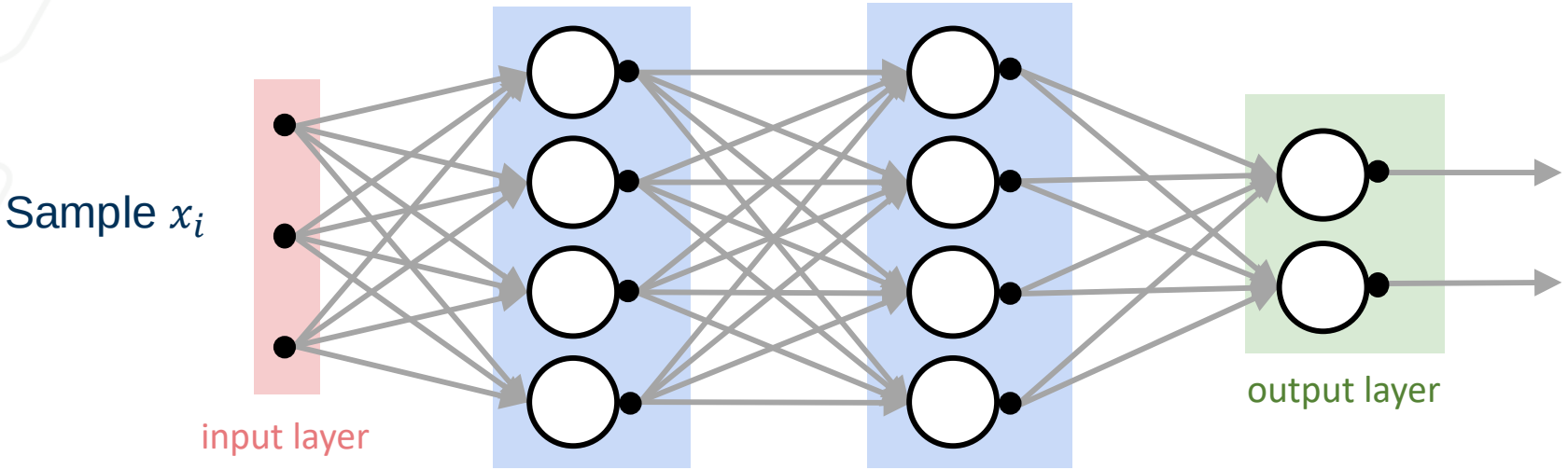
$$f(y_{ij}) = \frac{e^{y_{ij}}}{\sum_k e^{y_{ik}}}$$

predict class 2 with highest probability

Review: Artificial Neural Networks

Backpropagation Summary

During training, for every sample, we set the ground-truth label as a one-hot vector $[1, 0, \dots, 0]^{\mathrm{T}}$ with 1 for the correct class and 0 for every other class. Backpropagate the error and repeat for every sample.



Assume the raw outputs $y_i = [y_{i1}, y_{i2}, y_{i3}] = [3.2, 5.1, -1.7]$

Assume **ground-truth class is class 1**

Raw outputs	exp	unnormalized probabilities	normalize	probabilities
$y_{i1} = 3.2$	⇨	24.5	⇨	$f(y_{i1}) = 0.13$
$y_{i2} = 5.1$		164.0		$f(y_{i2}) = 0.87$
$y_{i3} = -1.7$		0.18		$f(y_{i3}) = 0.00$

One-hot ground-truth label

Loss function ⇦ **1.00** 0.00 0.00

⇨ to obtain error

Backpropagation

Linear Classifier

- Given the simple case of a NN having:
  - a single layer with
  - a single neuron with
  - a linear activation function.

  - The weight matrix is the single-column matrix $\boldsymbol{W} \in \mathbb{R}^{P\times 1}$

$$\boldsymbol{W}^{(1)} = \left[\boldsymbol{w}_1^{(1)}\right] = \boldsymbol{w}_1^{(1)}$$

  - The bias $\left[\boldsymbol{b}_1^{(1)}\right] = b_1^{(1)}$

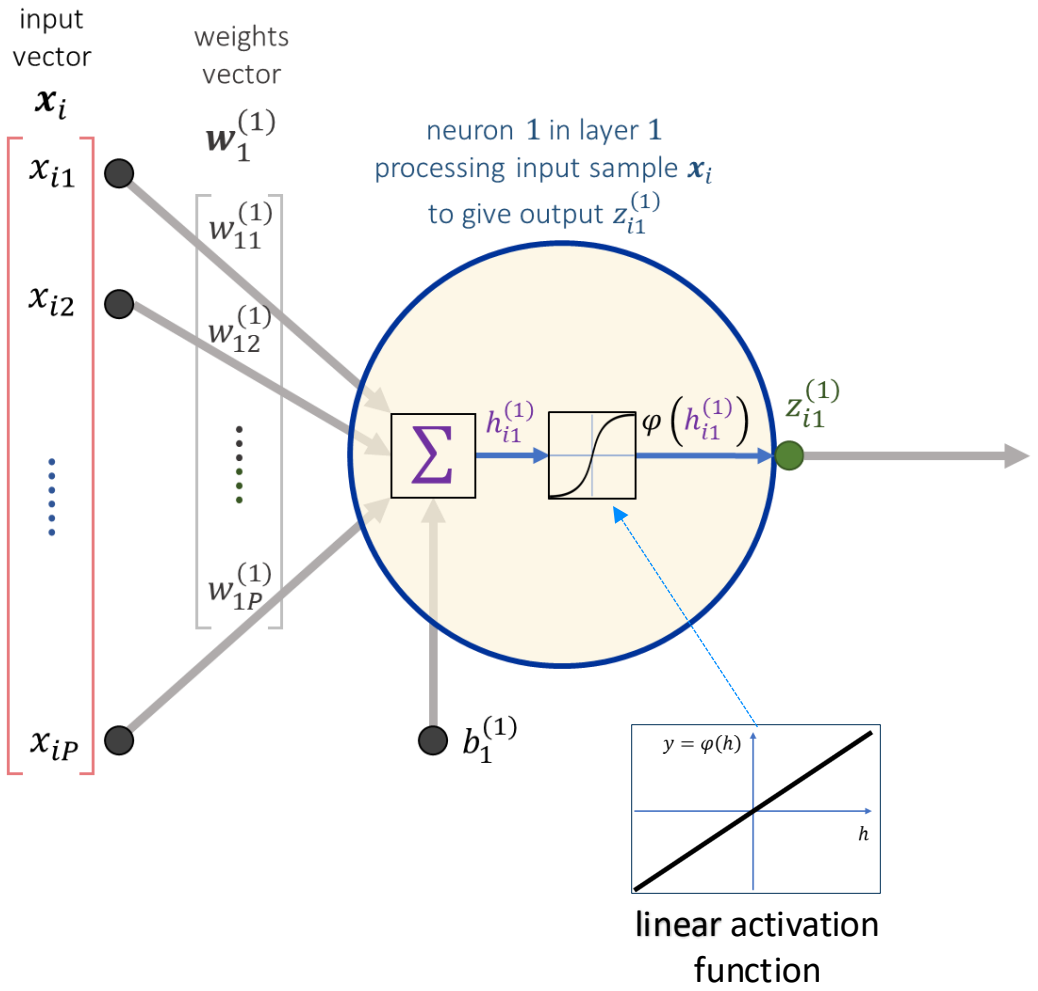
  - The output $\boldsymbol{Z}^{(1)} = \left[\left(\boldsymbol{z}_1^{(1)}\right)^T\right] = z_1^{(1)} = \varphi\left(h_{i1}^{(1)}\right) = h_{i1}^{(1)}$

- Given an input matrix $\boldsymbol{X} \in \mathbb{R}^{N\times P}$ as defined earlier, the neuron's output matrix $\hat{\boldsymbol{Y}} \in \mathbb{R}^{N\times 1}$ is:

$$\hat{\boldsymbol{Y}} = \varphi\left(\boldsymbol{X}(\boldsymbol{W}^{(1)})^T + \boldsymbol{o}(\boldsymbol{b}^{(1)})^T\right)$$

where $\boldsymbol{o} \in \mathbb{R}^{N\times 1}$ is a vector of ones $\boldsymbol{1}_N$

$$\boldsymbol{X}(\boldsymbol{W}^{(1)})^T + \boldsymbol{o}(\boldsymbol{b}^{(1)})^T = \boldsymbol{X}\boldsymbol{w}_1^{(1)} + b_1^{(1)}$$



Backpropagation

Single Layer MLPs

1. System setup:

$$\hat{\boldsymbol{Y}} = \boldsymbol{\sigma}\left(\boldsymbol{X}\left(\boldsymbol{W}^{(1)}\right)^{T} + \boldsymbol{o}\left(\boldsymbol{b}^{(1)}\right)^{T}\right)$$

For a single sample $\boldsymbol{x}_i \in \mathbb{R}^{p\times 1}$, the output of the ANN is given by:

$$\hat{y}_i = \sigma(h_i) = \frac{1}{1+e^{-h_i}}$$

$$h_i = \boldsymbol{w}^T\boldsymbol{x}_i + b = \boldsymbol{w}\boldsymbol{x}_i + b$$

where, $h_i, \hat{y}_i \in \mathbb{R}$, $\mathbf{w}^{(1)} \in \mathbb{R}^{1\times p}$ is the weights vector, and $b^{(1)} \in \mathbb{R}$ is the bias term

2. Derivation of the loss

Recall binary cross-entropy loss:

$$L(y_i, \hat{y}_i) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)$$

Backpropagation

Single Layer MLPs

2.1. derivative of the loss w.r.t. $\mathbf{w}$:

$$\frac{\partial L}{\partial \mathbf{w}} = -\frac{1}{N}\sum_{i=1}^{N} y_i \left(\frac{\partial}{\partial \mathbf{w}} \log \hat{y}_i\right) + (1-y_i)\left(\frac{\partial}{\partial \mathbf{w}} \log(1-\hat{y}_i)\right)$$

$$= -\frac{1}{N}\sum_{i=1}^{N} y_i \left(\frac{1}{\hat{y}_i}\frac{\partial \hat{y}_i}{\partial \mathbf{w}}\right) + (1-y_i)\left(\frac{-1}{1-\hat{y}_i}\frac{\partial \hat{y}_i}{\partial \mathbf{w}}\right)$$

using $\hat{y}_i = \sigma\,(h_i)$ and $\frac{\partial \sigma(h)}{\partial h} = \sigma(h)\,[1-\sigma(h)]$

$$\frac{\partial L}{\partial \mathbf{w}} = -\frac{1}{N}\sum_{i=1}^{N} y_i \left(\frac{1}{\hat{y}_i}\hat{y}_i[1-\hat{y}_i]\frac{\partial h_i}{\partial \mathbf{w}}\right) + (1-y_i)\left(\frac{-1}{1-\hat{y}_i}\hat{y}_i[1-\hat{y}_i]\frac{\partial h_i}{\partial \mathbf{w}}\right)$$

$$= -\frac{1}{N}\sum_{i=1}^{N} y_i \left([1-\hat{y}_i]\frac{\partial}{\partial \mathbf{w}}(\mathbf{w}_i + b)\right) + (1-y_i)\left(-\hat{y}_i\frac{\partial}{\partial \mathbf{w}}(\mathbf{w}_i + b)\right)$$

Therefore $\frac{\partial L}{\partial \mathbf{w}} = -\frac{1}{N}\sum_{i=1}^{N} y_i \left([1-\hat{y}_i]\mathbf{x}_i^T\right) + (1-y_i)\left(-\hat{y}_i\mathbf{x}_i^T\right)$

Backpropagation

Single Layer MLPs

2.1. derivative of the loss w.r.t. $\mathbf{w}$:

$$\frac{\partial L}{\partial \boldsymbol{w}} = -\frac{1}{N}\sum_{i=1}^{N} y_i\left([1-\hat{y}_i]\mathbf{x}_i^T\right) + (1-y_i)\left(-\hat{y}_i\mathbf{x}_i^T\right)$$

$$= -\frac{1}{N}\sum_{i=1}^{N}(y_i-\hat{y}_i)\mathbf{x}_i^T$$

$$= -\frac{1}{N}\left(\mathbf{Y}^T-\widehat{\mathbf{Y}}^T\right)\mathbf{X} = \frac{1}{N}\left(\widehat{\mathbf{Y}}^T-\mathbf{Y}^T\right)\mathbf{X}$$

2.2 derivate of the loss w.r.t. b:

Derivation of b is similar to $\mathbf{w}$, the final derivative:

$$\frac{\partial L}{\partial b} = -\frac{1}{N}\sum_{i=1}^{N} y_i\left([1-\hat{y}_i]\right) + (1-y_i)(-\hat{y}_i)$$

$$= -\frac{1}{N}\sum_{i=1}^{N}(y_i-\hat{y}_i)$$

$$= -\frac{1}{N}\left(\mathbf{Y}^T-\widehat{\mathbf{Y}}^T\right)\boldsymbol{o} = \frac{1}{N}\left(\widehat{\mathbf{Y}}^T-\mathbf{Y}^T\right)\boldsymbol{o}$$

Image Classification: mapping the image pixels to probabilities for each category

For simplicity, we take a linear function:

$$\widehat{\boldsymbol{Y}} = \boldsymbol{\phi}(\boldsymbol{X}\boldsymbol{W}^T + \boldsymbol{b}^T)$$

where

- $\boldsymbol{X} \in \mathbb{R}^{N \times P}$: dataset containing N vectorized images
- $P \in \mathbb{R}^{H \times W \times C}$: the number of pixels (*features*) of each image
- $\widehat{\boldsymbol{Y}} \in \mathbb{R}^{N \times P^{(k)}}$: the associated probabilities of each category $1,2,\ldots,P^{(k)}$
- $\boldsymbol{\phi}(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{b})$: the activation function
- $\boldsymbol{W} \in \mathbb{R}^{P \times P^{(k)}}$: the weight matrix
- $\boldsymbol{b} \in \mathbb{R}^{P^{(k)} \times 1}$: the bias vector

$P^{(k)} = C$ (Number of classes) for Classification

Single image binary classification (predicting dog/cat)

stretch pixels into single column vector

Input 32x32 RGB image

$$\phi\left(\begin{bmatrix} 0.2 & 2.1 \\ -0.5 & 0.0 \\ 0.1 & 0.25 \\ 2.0 & 0.2 \\ 1.5 & -0.3 \\ \vdots & \vdots \\ 1.3 & 1.2 \end{bmatrix}^{T} \begin{bmatrix} 56 \\ 231 \\ 24 \\ 188 \\ 75 \\ \vdots \\ 32 \end{bmatrix} + \begin{bmatrix} 0.2 \\ 2.4 \end{bmatrix}\right) = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix} \begin{matrix} \text{Cat score} \\ \text{Dog score} \end{matrix}$$

$w \in \mathbb{R}^{2\times(32)(32)(3)}$ $x_i \in \mathbb{R}^{(32)(32)(3)\times 1}$ $b \in \mathbb{R}^{2\times 1}$ $y_i \in \mathbb{R}^{2\times 1}$

Neural Networks

Training MLP for Image Classification

Example Datasets:

- MNIST (hand-written digits)
  - # total: 70,000 grayscale images
  - # classes: 10
  - # size: 28x28
  - # training samples: 60,000 images
  - # test samples: 10,000 images

https://www.researchgate.net/figure/Sample-images-of-MNIST-data_fig3_222834590

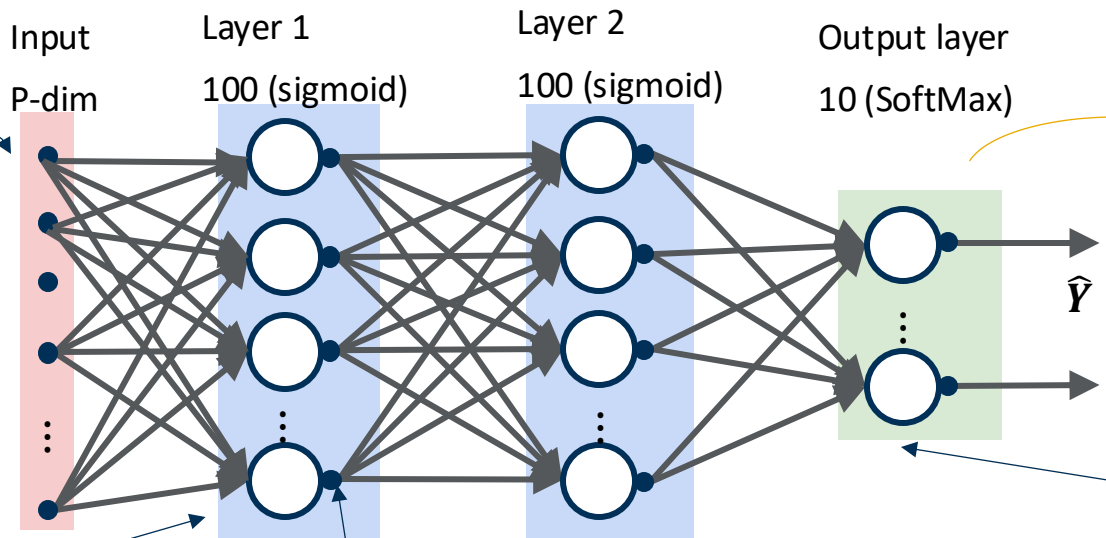
- CIFAR-10 (subsets of the 80 million tiny images)
  - # total: 60,000 color images
  - # classes: 10
  - # size: 32x32
  - # training samples: 50,000 images
  - # test samples: 10,000 images

https://www.kaggle.com/c/cifar-10

Neural Networks

Training MLP for Image Classification

- *P*-dimensional input vectors, 2 hidden layers with 100 neurons in each, and 10 neurons in output layer.



$$\hat{\boldsymbol{Y}} = \boldsymbol{Z}^{(3)}$$
$$\boldsymbol{Z}^{(3)} = \varphi(\boldsymbol{Z}^{(2)}(\boldsymbol{W}^{(3)})^T + \boldsymbol{o}(\boldsymbol{b}^{(3)})^T)$$
$$\boldsymbol{Z}^{(2)} = \varphi(\boldsymbol{Z}^{(1)}(\boldsymbol{W}^{(2)})^T + \boldsymbol{o}(\boldsymbol{b}^{(2)})^T)$$
$$\boldsymbol{Z}^{(1)} = \varphi(\boldsymbol{X}(\boldsymbol{W}^{(1)})^T + \boldsymbol{o}(\boldsymbol{b}^{(1)})^T)$$

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & \cdots & x_{1P} \\ x_{21} & x_{22} & \cdots & \cdots & x_{2P} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{N1} & x_{N2} & \cdots & \cdots & x_{NP} \end{bmatrix}$$

P = 784(28x28x1) for MNIST
P = 3072(32x32x3) for CIFAR-10

$$\hat{\boldsymbol{Y}} = \begin{bmatrix} \hat{y}_{11} & \cdots & \hat{y}_{1,10} \\ \hat{y}_{21} & \cdots & \hat{y}_{2,10} \\ \hat{y}_{N1} & \cdots & \hat{y}_{N,10} \end{bmatrix}$$

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & \cdots & \cdots & w_{1P}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & \cdots & \cdots & w_{2P}^{(1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{100,1}^{(1)} & w_{100,2}^{(1)} & \cdots & \cdots & w_{100,P}^{(1)} \end{bmatrix}$$

$$\boldsymbol{Z}^{(1)} = \begin{bmatrix} z_{11}^{(1)} & z_{12}^{(1)} & \cdots & z_{1,100}^{(1)} \\ z_{21}^{(1)} & z_{22}^{(1)} & \cdots & z_{2,100}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ z_{N,1}^{(1)} & z_{N,2}^{(1)} & \cdots & z_{N,100}^{(1)} \end{bmatrix}$$

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & \cdots & w_{1,100}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} & \cdots & w_{2,100}^{(2)} \\ w_{31}^{(2)} & w_{32}^{(2)} & \cdots & w_{3,100}^{(2)} \\ w_{100,1}^{(2)} & w_{100,2}^{(2)} & \cdots & w_{100,100}^{(2)} \end{bmatrix}$$

$$\boldsymbol{W}^{(3)} = \begin{bmatrix} w_{11}^{(3)} & w_{12}^{(3)} & w_{13}^{(3)} & w_{14}^{(3)} \\ \vdots & \vdots & \vdots & \vdots \\ w_{10,1}^{(3)} & w_{10,2}^{(3)} & w_{10,3}^{(3)} & w_{10,4}^{(3)} \end{bmatrix}$$

Neural Networks

Training MLP for Image Classification

https://www.researchgate.net/figure/Sample-images-of-MNIST-data_fig3_222834590

MNIST acc: ~72%

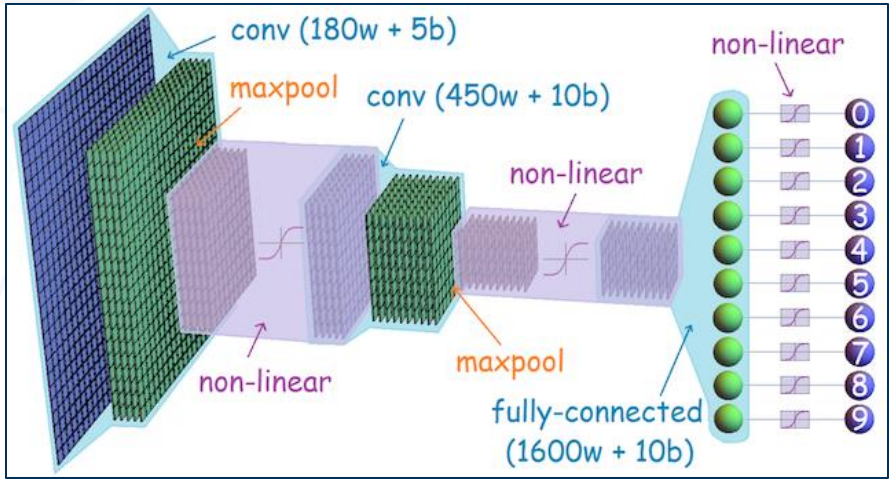
https://www.kaggle.com/c/cifar-10

CIFAR-10 acc:~53 %

- MLP tends to recognize better on the images in which objects are well-centered, with simple shape/texture information.
- need more powerful models to address complex image datasets.

Overview

In this Lecture..



Review: Artificial Neural Networks

- Activation Function
- Perceptron Network
- Multi-layer ANN

Backpropagation

- Feedforward and Backward Error Propagation
- Learning Algorithm
- Image Classification using ANNs
- Mini-batch Gradient descent
- Linear Classifier
- Logistic Regression

PyTorch

- Autograd
- Built-in Modules
- Examples

PyTorch
Basics

- What is PyTorch?
  - A Python-based scientific computing package for executing dynamic computational graphs over Tensor objects that behave similarly as Numpy arrays.
- Why PyTorch?
  - It comes with a powerful *automatic differentiation* engine that removes the need for manual back-propagation

PyTorch

AutoGrad: Automatic Differentiation

- Given a vector valued function $y = f(x), x \in \mathbb{R}^n, y \in \mathbb{R}^m$, the gradient of y w.r.t. x is a Jacobian matrix:

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

- Generally, torch.autograd is an engine for computing *vector-Jacobian* product. That is, given any vector $v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix}$, compute the product $v^T \boldsymbol{J}$. If v happens to be the gradient of a scalar function $l = g(y)$, that is, $v = \begin{bmatrix} \frac{\partial l}{\partial y_1} \\ \frac{\partial l}{\partial y_2} \\ \vdots \\ \frac{\partial l}{\partial m} \end{bmatrix}$, then by the chain rule, the *vector-Jacobian* product would be the gradient of l w.r.t. x.

PyTorch

AutoGrad: Toy Example

Given $\boldsymbol{y} = \|\boldsymbol{x_1} - \boldsymbol{x_2}\|_2^2$

Compare $2(\boldsymbol{x_1} - \boldsymbol{x_2})$ and $\frac{\partial y}{\partial x_1}$ using PyTorch *autograd*

```
import torch
x1 = torch.randn(2, 2).requires_grad_(True)
x2 = torch.randn(2, 2).requires_grad_(True)

y = torch.sum((x1-x2)**2)
y.backward()

print(2*(x1-x2))
print(x1.grad)
# ======= output ========
# tensor([[ 0.1411, 3.9669], [-1.2993, 2.1217]])

# tensor([[ 0.1411, 3.9669], [-1.2993, 2.1217]] ,
grad_fn=<MulBackward0>)
```

manually computed gradient $2(\boldsymbol{x_1} - \boldsymbol{x_2})$

$\frac{\partial y}{\partial x_1}$ using PyTorch autograd

PyTorch

Numpy vs PyTorch

Given example, we can compare the gradients computed manually with NumPy and PyTorch autograd (see highlighted lines)

```
for i in range(max_iter):
  Y_hat = X@W.T + o@b.T
  # MSE loss
  loss = np.sum((Y_hat-Y)**2)/num_samples
#FrobeniusNorm
  W_gradients = (2/num_samples)*(Y_hat.T - Y.T)@X
  b_gradients = (2/num_samples)*(Y_hat.T - Y.T)@o
  # follow the negative gradients
  W = W - alpha*W_gradients   # update the weight
matrix
  b = b - alpha*b_gradients   # update the biases
matrix
  Y_hat = X@W.T + o@b.T         # predicted
"probabilities
#W gradients: [ 0.06089246 -0.01254302]
#b gradients: -0.34077861476403615
Iteration:120 | Loss = 0.1104 | Train Acc = 0.80
```

```
for i in range(max_iter):
  y_hat = X_tensor@W_tensor.t() + o_tensor@b_tensor.t()
  # MSE loss
  loss = F.mse_loss(y_hat, y_tensor)
  loss.backward()
  W_gradient = W_tensor.grad
  b_gradient= b_tensor.grad
  # updating the weights
  with torch.no_grad():
    W_tensor.data = W_tensor.data - alpha*W_gradient
    b_tensor.data = b_tensor.data - alpha*b_gradient
  y_hat = X_tensor@W_tensor.t() + o_tensor@b_tensor.t()
# predicted "probabilities
#W gradients: [ 0.06089246 -0.01254302]
#b gradients: -0.34077861476403615
Iteration:120 | Loss = 0.1104 | Train Acc = 0.80
```

PyTorch

Numpy vs PyTorch: Sigmoid and BCE Loss

Given example, we can compare the gradients computed manually with NumPy and PyTorch autograd (see highlighted lines)

```
for i in range(max_iter):
  H = X@W.T + o@b.T
  Y_hat = 1. / (1. + np.exp(-H))
  # BCE loss
  loss = -1. * np.sum(Y * np.log(Y_hat) + (1-Y) * np.log(1-Y_hat)) / num_samples
  W_gradients = (1./num_samples)*(Y_hat.T - Y.T)@X
  b_gradients = (1./num_samples)*(Y_hat.T - Y.T)@o
  # follow the negative gradients
  W = W - alpha*W_gradients   # update the weight matrix
  b = b - alpha*b_gradients   # update the biases matrix
#W gradients: [0.07499208 -0.05085141]
#b gradients: -0.16729619094308953
Iteration:120 | Loss = 0.3512 | Train Acc = 0.90
```

```
for i in range(max_iter):
  h_hat = X_tensor@W_tensor.t() + o_tensor@b_tensor.t()
  y_hat = torch.sigmoid(h_hat)
  # BCE loss
  loss = F.binary_cross_entropy(y_hat, y_tensor)
  loss.backward()
  W_gradient = W_tensor.grad
  b_gradient = b_tensor.grad
  # updating the weights
  with torch.no_grad():
    W_tensor.data = W_tensor.data - alpha*W_gradient
    b_tensor.data = b_tensor.data - alpha*b_gradient
#W gradients: [0.07499208 -0.05085141]
#b gradients: -0.16729619095070608
Iteration:120 | Loss = 0.3512 | Train Acc = 0.90
```

Basic Modules in PyTorch:

nn.Module API: defines arbitrary network architectures, while automatically tracking every learnable parameters.

torch.optim package: implements all the common optimizers, such as SGD, RMSProp, Adagrad, and Adam, etc.

PyTorch

Basic Modules

To use the Module API, follow the steps below:

1. Subclass nn.Module

2. In the constructor __init__() define all the layers and components you need as class attributes

3. In the forward() method, define the connectivity of your network

```
# Give your network class an intuitive name like
`SNC(Sigle Neuron Classifier)`.
class SNC(nn.Module):
    def __init__(self):
        super(SNC, self).__init__()
        # define a single neuron
        self.single_neuron= nn.Linear(in_features=2,
out_features=1, bias=True) # the layer will learn an
additive bias
        self.non_linearity = nn.Sigmoid()

    def forward(self, x):
        x = self.single_neuron(x)
        x = self.non_linearity(x)
        return x
```

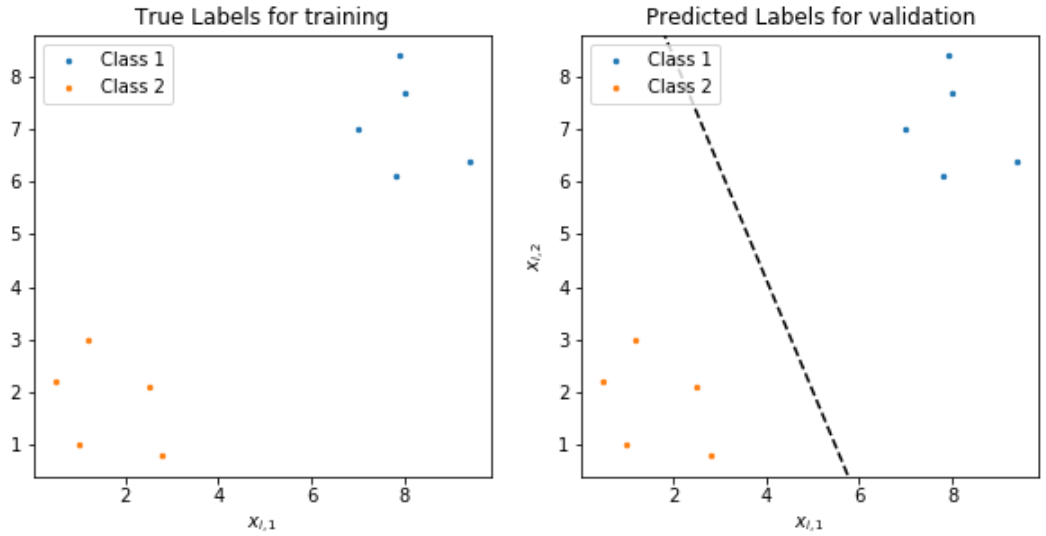
PyTorch example – single neuron classifier

PyTorch

Implementing Single Neuron Classifier

```
# Sigle Neuron Nonlinear Classifier

class SNC(nn.Module):

    def __init__(self): # define the layer(s) and network components

        super(SNC, self).__init__()

        self.single_neuron= nn.Linear(in_features=num_features,
out_features=1, bias=True) # the layer will learn an additive bias.

        self.non_linearity = nn.Sigmoid()

    def forward(self, x): # here we define the forward pass for network

        x = self.single_neuron(x)

        x = self.non_linearity(x)

        return x

network = SNC()

optimizer = optim.SGD(network.parameters(), lr=2) #Stochastic gradient
descend

criterion = nn.BCELoss() # using BCE loss for training

for i in range(max_iter):

    network.zero_grad() # clearning all gradients in the network

    y_hat_tensor = network(X_tensor) #forward pass

    loss = criterion(y_hat_tensor, y_tensor)

    loss.backward()

    optimizer.step()
```

example



⋮

Iteration 20#, training acc = 1.00

Loss: $L(\theta)$= 0.0321

W gradients:

$$\frac{\partial L(\theta)}{\partial W^{(1)}} = [\mathbf{0.0083}, \mathbf{0.0355}]^T$$

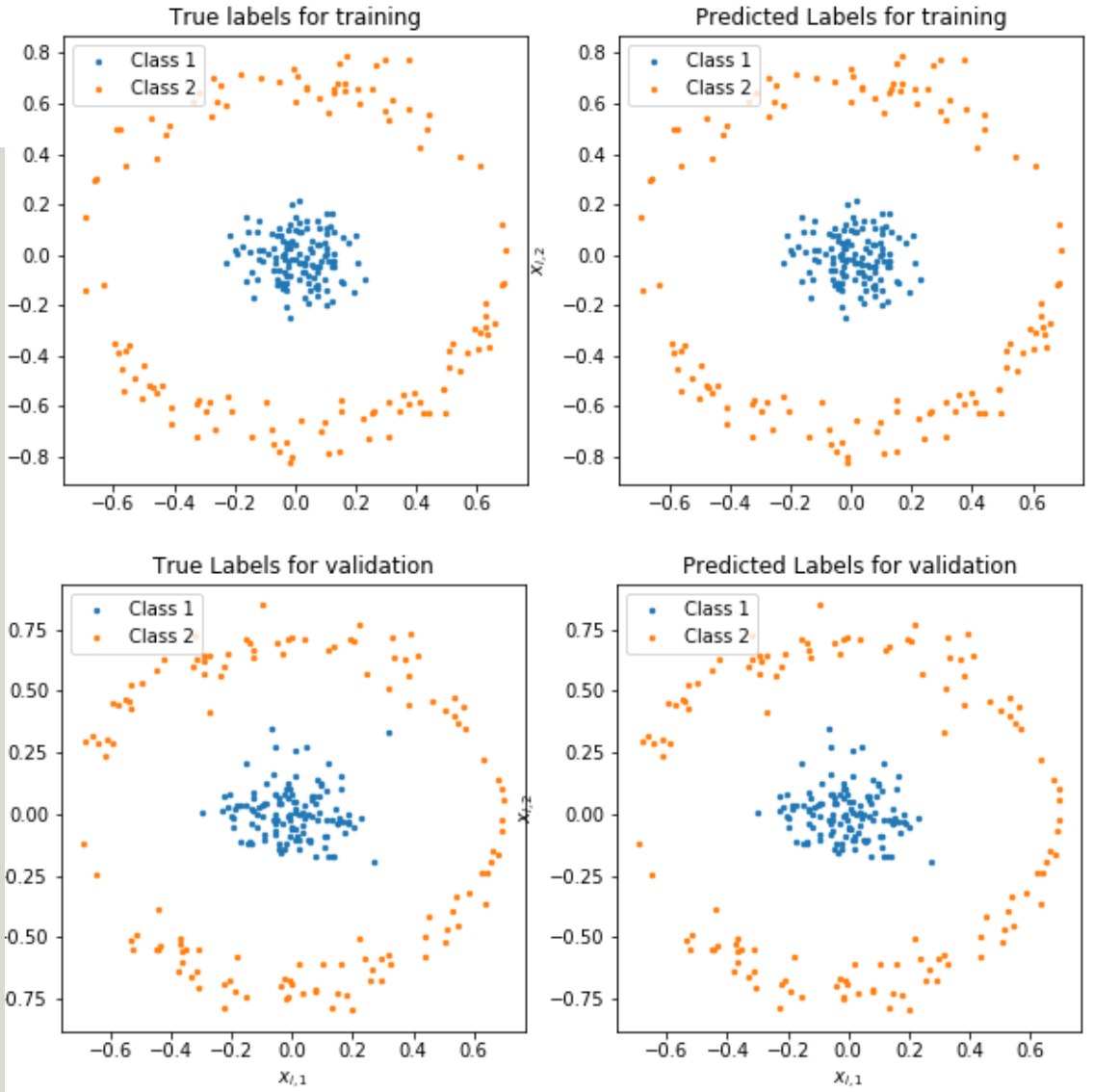
b gradients:

$$\frac{\partial L(\theta)}{\partial b^{(1)}} = \mathbf{-0.0133}$$

PyTorch

Implementing MLP

```
# Multi-Layer non-linear Classifier
class MLP(nn.Module):
    def __init__(self, num_classes): # define the layer(s) and network components
        super(MLP, self).__init__()
        self.layer1= nn.Linear(in_features=num_features, out_features=4, bias=True)
        self.out= nn.Linear(in_features=4, out_features=1, bias=True)
        self.non_linearity = nn.Sigmoid()
    def forward(self, x): # here we define the forward pass for network
        y = self.layer1(x)
        y = self.non_linearity(y)
        y = self.out(y)
        y = self.non_linearity(y)
        return y
network = MLP(num_classes=1)
optimizer = optim.SGD(network.parameters(), lr=1.2) criterion = nn.BCELoss()
for i in range(max_iter):
    network.zero_grad() # clearning all gradients in the network
    y_train_hat = network(X_train_tensor) #forward pass
    loss = criterion(y_train_hat, y_train_tensor)
    loss.backward()
    optimizer.step()
```



Appendix A: Notations

- x_i: a single feature
- $\boldsymbol{x}_i$: feature vector (a data sample)
- $\boldsymbol{x}_{:,i}$: feature vector of all data samples
- $\boldsymbol{X}$: matrix of feature vectors (dataset)
- N: number of data samples
- P: number of features in a feature vector
- $k^{(i)}$: number of neurons in the i^{th} layer
- α: learning rate
- Bold letter/symbol: vector
- Bold capital letters/symbol: matrix