Backpropagation

Applied Deep Learning
EE5438

Prof. Lai-Man Po
Department of Electrical Engineering
City University of Hong Kong

Message 1: Submission of Project Proposal

- **Just a friendly reminder:** The deadline to submit your group project proposal is **Feb 14, 2026, at 11pm**. Please submit a PDF file with the project title, list of group members, and other necessary details to the CANVAS group project proposal assignment.
- Only one proposal per group is required, and it should be submitted by the project's team leader.
- You can find more information about the group project on the course website:
  - https://www.ee.cityu.edu.hk/~lmpo/ee4016/projects.html
- Remember, **each group should assign a project leader** who will be responsible for submitting the proposal on CANVAS.
- The file name should follow this format:
  - Filename format : Proposal_GroupNumber_ProjectName.pdf
  - Filename example: Proposal_Group01_Audio_Classification.pdf

Message 2: Assignment 1

Image Classification with Multi-Layer Perceptron

- The assignment 1 is now available in the schedule webpage for download. The deadline for the assignment 1 is **Saturday of Week 5 (Feb 21, 2026).**
  - https://www.ee.cityu.edu.hk/~lmpo/ee4016/pdf/2026_EE4016_Ass01.pdf
  - **Colab:** https://colab.research.google.com/drive/1zSe-32cpojFYT2oxySvrAdSbMLr4vY9I#scrollTo=hjkFuokaRv3G
- **The answers of the section A must be handwritten** and then scan the answer sheets into a single pdf file.
- Submit the answer sheets and Colab notebook of the Assignment 1 as a zip file to this CANVAS assignment 1:
  - Filename format : Assignment01_StudentName_StudentID.zip
  - Filename example: Assignment01_Chen_Hoi_501234567.zip

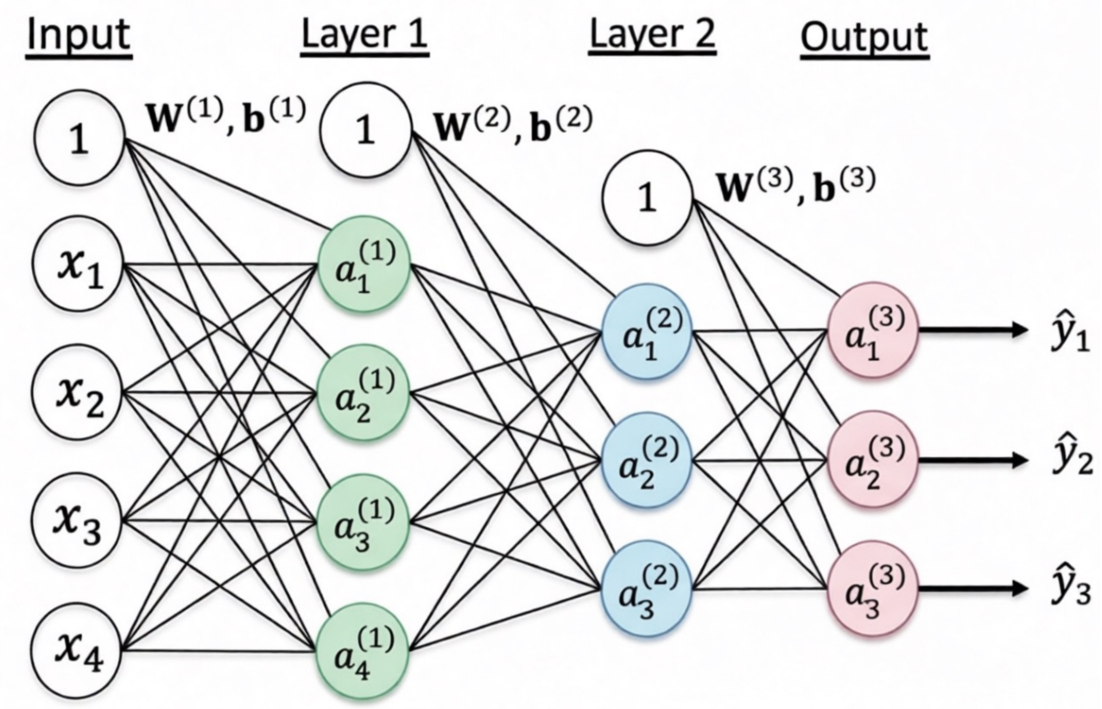
Anatomy of the MLP Architecture

Context: An MLP consists of an input layer, one or more hidden layers, and an output layer. Nodes are connected via weighted connections.

Notation:

- l : Layer index
- $\mathbf{W}^{(l)}$: Weight matrix for layer l
- $\mathbf{b}^{(l)}$: Bias vector for layer l
- $g^{(l)}$: Activation function

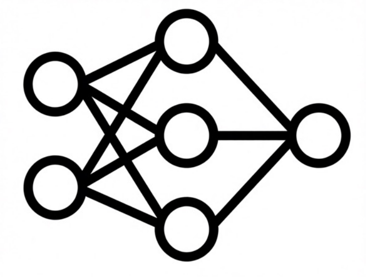
$$\boxed{\mathbf{a}^{(l)} = g^{(l)}\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right)}$$



$$\hat{\mathbf{y}} = f_\theta(\mathbf{x}) = \mathrm{Softmax}\left(\mathbf{W}^{(3)}\mathrm{ReLU}\left(\mathbf{W}^{(2)}\mathrm{ReLU}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) + \mathbf{b}^{(3)}\right)$$

MLP based Model Design

- Based on the problems to define the **Hyperparameters of the MLP architecture**:
  - Input dimension (d)
  - Network Depth (L)
  - Number of neurons of each layer ($n_l: l = 1,2, \ldots, L$)
  - Output dimension ($K = n_L$)
  - Activation functions ($\sigma, \text{tanh}, \text{ReLU}, \text{softmax}$)
  - **Cost function** $\mathcal{L}(\theta)$



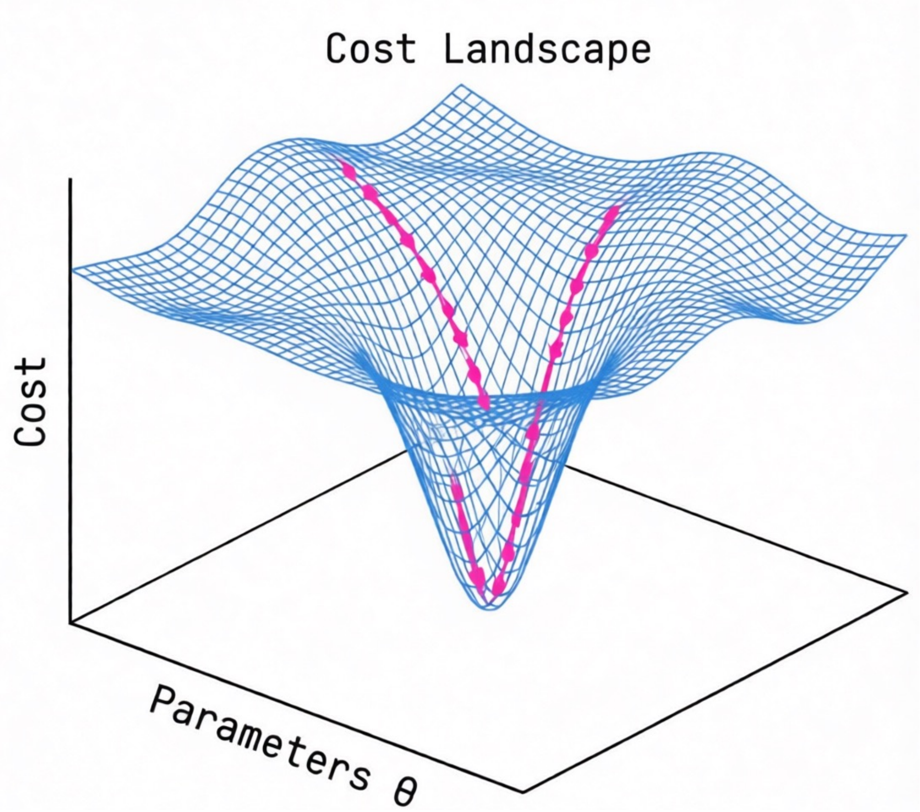
- The MLP model can be represented by **a set of weights and biases parameters** θ as
  - $\theta := \left\{\left(\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\right)\right\}_{l=1}^{L}$
  - $\hat{\mathbf{y}} = f_\theta(\mathbf{x}) = g\left(\mathbf{W}^{(L)} \cdots g\left(\mathbf{W}^{(2)} g\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) \cdots + \mathbf{b}^{(L)}\right)$
- For a given dataset $\mathcal{D}$, use **Gradient Descent** with **backpropagation** to find the **optimal model parameter set** θ^* that minimizes a cost function $\mathcal{L}(\theta)$

The Training Objective: Minimizing Cost

For a given dataset $\mathcal{D} := \left\{\left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)\right\}_{i=1}^{N}$, the goal of training is **to find the optimal set of parameters** θ (weights and biases) that minimizes the **cost function** $\mathcal{L}(\theta)$ between predictions $\hat{\mathbf{y}}^{(i)}$ and targets $\mathbf{y}^{(i)}$.

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta)$$

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} loss\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right)$$



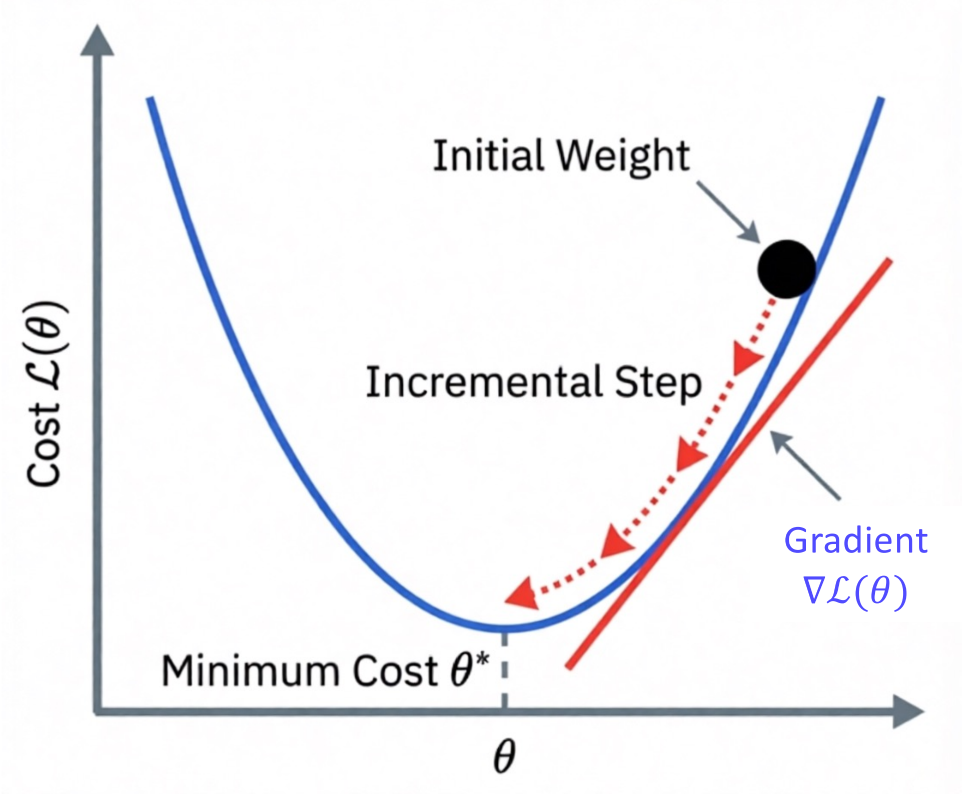
The Strategy: Gradient Descent

1. **Initialize**: Randomly set weights θ
2. **Compute Cost**: Measure performance $\mathcal{L}(\theta)$.
3. **Find Gradient**: Calculate $\nabla\mathcal{L}(\theta)$ (direction of steepest ascent).
4. **Update**: Step down the hill.

$$\theta_{new} = \theta_{old} - \eta \cdot \nabla\mathcal{L}(\theta)$$

η = Learning Rate (step size)

- **Repeat** steps 2 to 4, unit the cost is low enough or convergence.



The Computational Bottleneck

Why Backpropagation?

Gradient Descent Formula:

$$\theta_{new} = \theta_{old} - \eta \cdot \nabla\mathcal{L}(\theta)$$

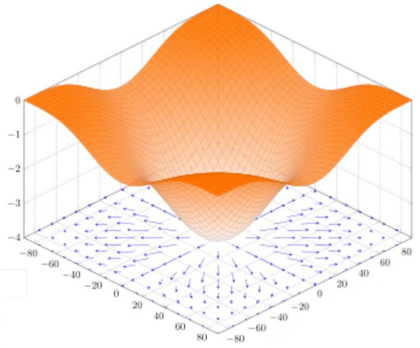
Naive Approach (Finite Differences):

- Perturb each parameter individually.
- Requires **O(*N*)** forward passes for *N* parameters.
- **Infeasible** for large models (e.g., 1M params → 1M forward passes per update!).

The Challenge:

- Modern networks have millions of parameters. Calculating the gradient this way has Exponential Complexity.

$$\nabla\mathcal{L}(\theta_t) = \begin{bmatrix} \vdots \\ \frac{\partial \mathcal{L}(\theta_t)}{\partial w_{i,j}^{(l)}} \\ \vdots \\ \frac{\partial \mathcal{L}(\theta_t)}{\partial b_i^{(l)}} \end{bmatrix}$$



This is what we need to calculate efficiently

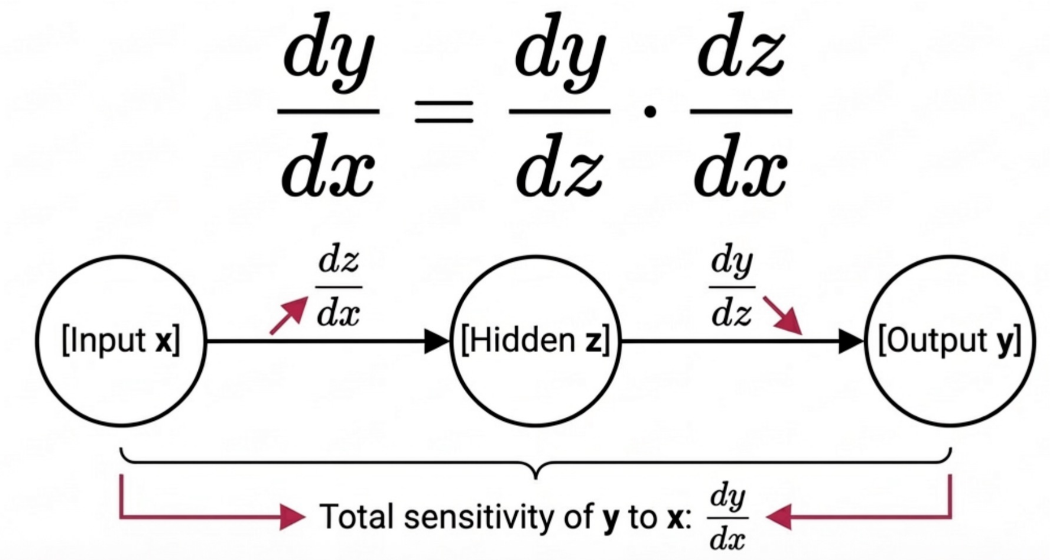
To efficiently compute the gradient when dealing with a large number of parameters, we employ a technique known as **backpropagation.**

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$$

The Engine: The Chain Rule

- Backpropagation leverages the Chain Rule to compute gradients for ALL parameters simultaneously in one backward sweep.



$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx}$$

Reduces complexity from exponential to linear, making Deep Learning feasible.

Overview of Backpropagation Algorithm

- The backpropagation algorithm uses the **Chain Rule** **to efficiently compute gradients** $\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}}$ in gradient descent–based network training.

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$$

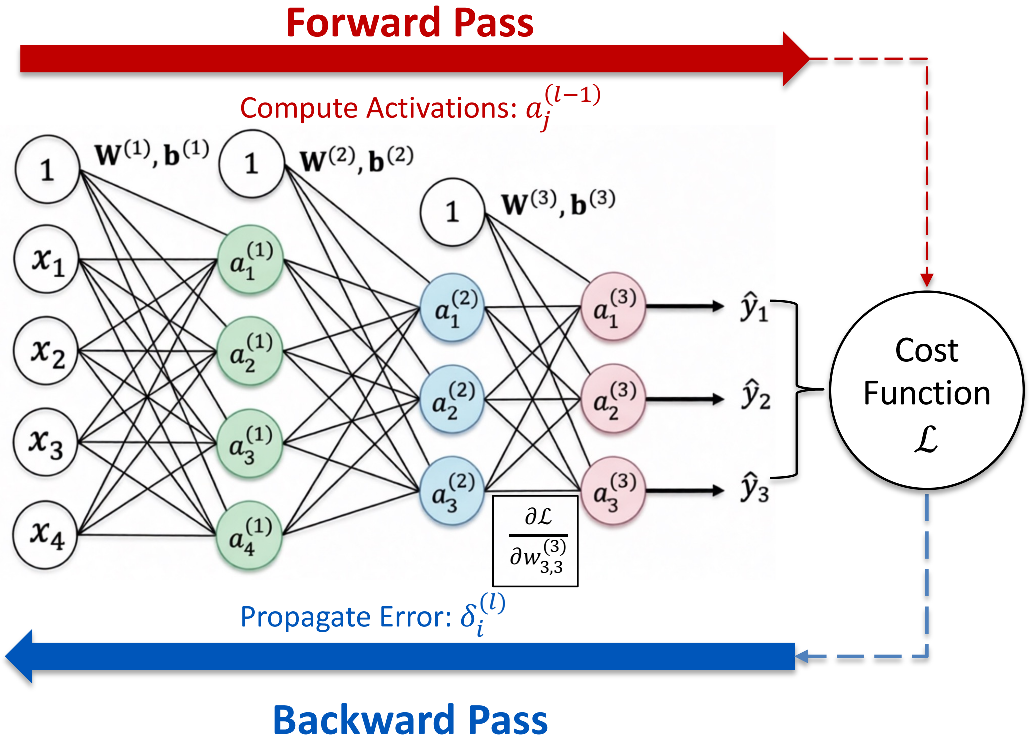
- **Forward Pass**: $a_j^{(l)}$

  Computer activation layer by layer. **Save these values.**

- **Backward Pass**:

$$\delta_i^{(l)} = g'^{(l)}\left(z_i^{(l)}\right)\sum_k \left(w_{k,i}^{(l+1)} \cdot \delta_k^{(l+1)}\right)$$

  Compute error signals (δ) in revers order. **Reuse cached values.**



Forward Pass

Compute Activations: $a_j^{(l-1)}$

Propagate Error: $\delta_i^{(l)}$

Backward Pass

Step: The Forward Pass

- Generating Predictions and Caching Activations

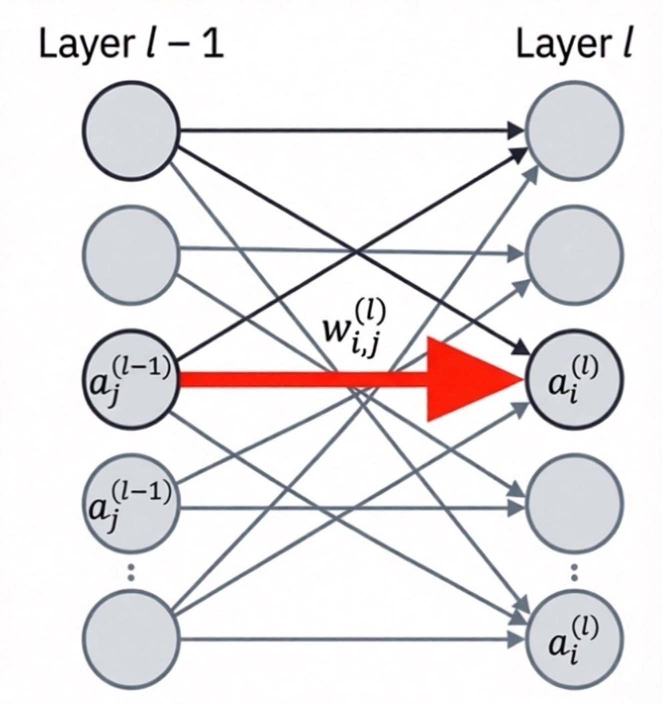
Net Input: $z_i^{(l)} = \sum_{j=1}^{n_{l-1}} w_{i,j}^{(l)} a_j^{(l-1)} + b_i^{(l)}$

Activation: $a_i^{(l)} = g^{(l)}\left(z_i^{(l)}\right)$

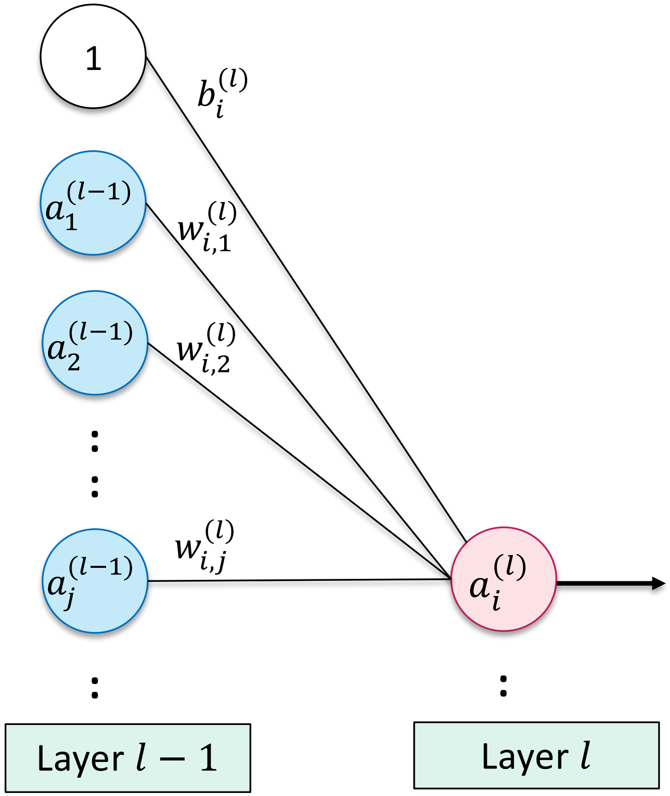
The Key Identity (Cached Value):

$$\frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} = a_j^{(l-1)}$$

The activation from the previous layer IS the partial derivative we need later. We store it.



$\frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}}$ and $\frac{\partial z_i^{(l)}}{\partial b_i^{(l)}} (l = 1,2, \dots, n_l)$



$$a_i^{(l)} = g^{(l)}\left(\sum_{j=1}^{n_{l-1}} w_{i,j}^{(l)} a_j^{(l-1)} + b_i^{(l)}\right)$$

$$a_i^{(l)} = g^{(l)}\left(z_i^{(l)}\right)$$

where $z_i^{(l)}$ is the net input and $g^{(l)}(\cdot)$ is the activation function

---

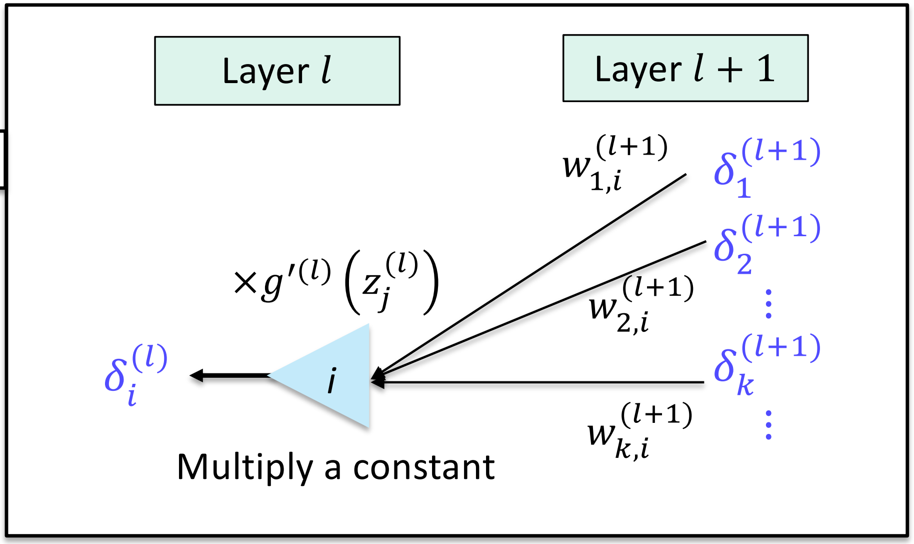
$$z_i^{(l)} = \sum_{j=1}^{n_{l-1}} w_{i,j}^{(l)} a_j^{(l-1)} + b_i^{(l)}$$

$$\boxed{\frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} = a_j^{(l-1)} \text{ and } \frac{\partial z_i^{(l)}}{\partial b_i^{(l)}} = 1}$$

$$\frac{\partial \mathcal{L}}{\partial z_i^{(l)}} = \delta_i^{(l)}$$: the propagated gradient corresponding to the l-th layer and i-th neuron

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}}{\partial z_i^{(l)}} = \sum_k \left(\frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \right)$$

$$= \underbrace{\frac{\partial a_i^{(l)}}{\partial z_i^{(l)}}} \sum_k \left(\underbrace{\frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}}}_{\delta_k^{(l+1)}} \cdot \underbrace{\frac{\partial z_k^{(l+1)}}{\partial a_i^{(l)}}} \right)$$

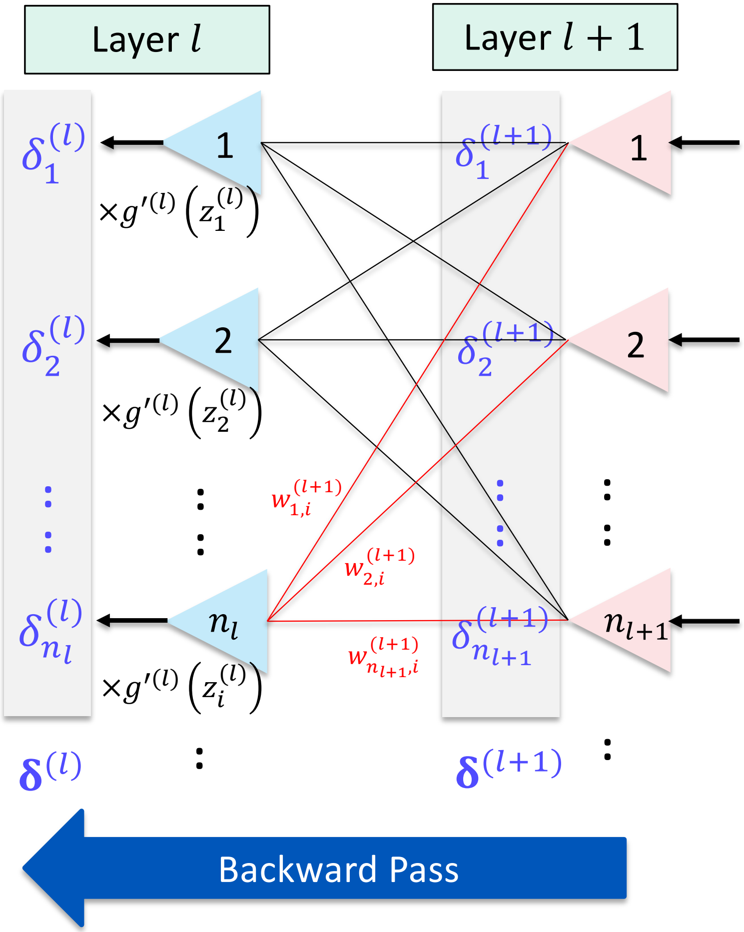


Gradient of the Activation function $$\frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = g'^{(l)}\left(z_i^{(l)}\right)$$

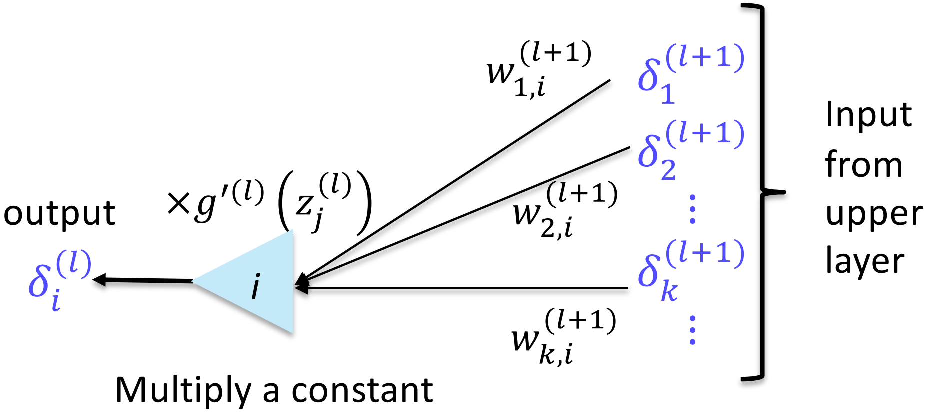
$$z_k^{(l+1)} = \sum_j w_{k,j}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} \Rightarrow \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} = w_{k,j}^{(l+1)}$$

$$\delta_i^{(l)} = g'^{(l)}\left(z_i^{(l)}\right) \sum_k \left(\delta_k^{(l+1)} w_{k,i}^{(l+1)} \right)$$

$\frac{\partial \mathcal{L}(\theta)}{\partial z_i^{(l)}} = \delta_i^{(l)}$ **is just a scaled weighted sum of** $\delta_k^{(l+1)}$ of the upper layer (**Backpropagation**)



$$=$$



$$\boxed{\delta_i^{(l)} = g'^{(l)}\left(z_i^{(l)}\right)\sum_k \left(w_{k,i}^{(l+1)}\delta_k^{(l+1)}\right)}$$

The backpropagation process begins by calculating the **delta terms** $\delta_k^{(L)}$ for the output layer *L*. It then systematically computes the delta terms $\delta_k^{(l)}$ for each preceding layer *l*, using the delta terms $\delta_k^{(l+1)}$ from the layer immediately above it.

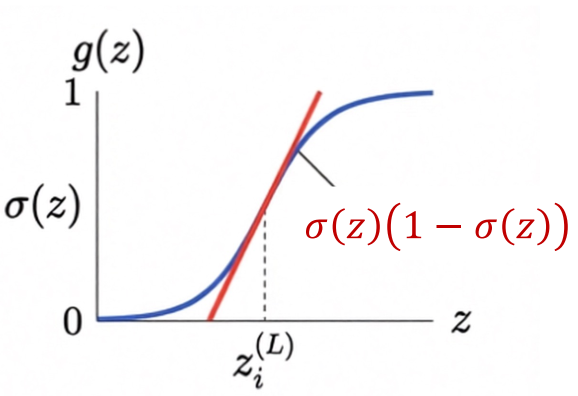
Step 2: Backward Pass (Output Layer)

- Calculating the Initial Error Signal

$$\delta_i^{(L)} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i^{(L)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \cdot g'^{(L)}\left(z_i^{(L)}\right)$$

Gradient of Cost
(How wrong was the prediction?)

Derivative of Activation
(How sensitive is the neuron?)



For Sigmoid Activation

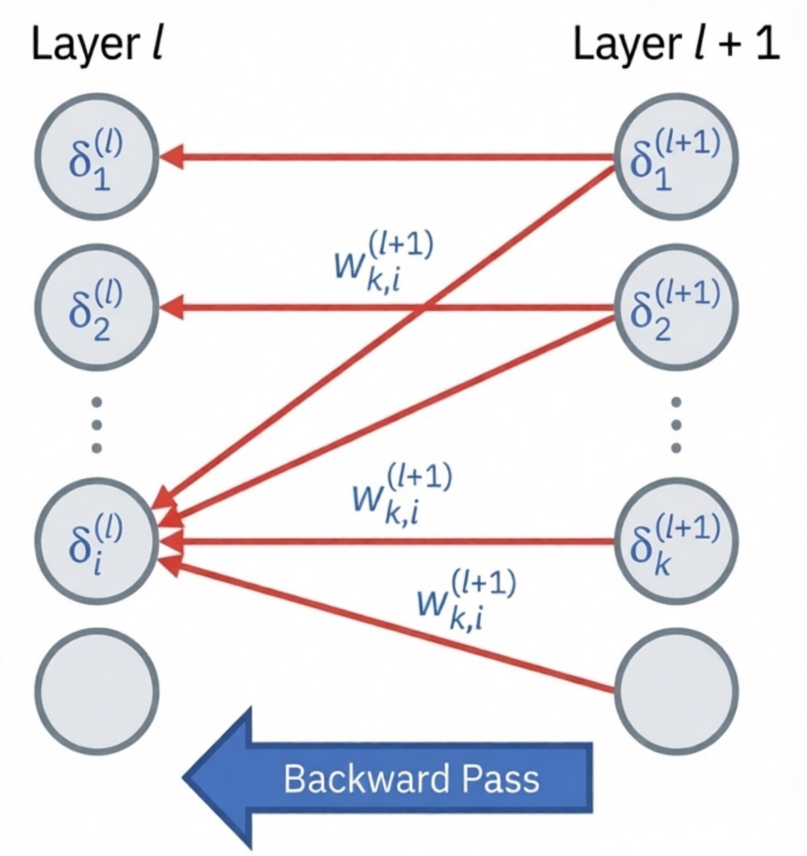
$$g'^{(L)}(z) = \sigma'(z) = \sigma(z)\left(1-\sigma(z)\right)$$

Step 3: Backward Pass (Hidden Layers)

Propagating the Error Recursively

$$\delta_i^{(l)} = g'^{(l)}\left(z_i^{(l)}\right)\sum_k \left(w_{k,i}^{(l+1)}\delta_k^{(l+1)}\right)$$

The error for the current layer is the **weighted sum of errors** from the layer ahead.



Overall Backward Pass

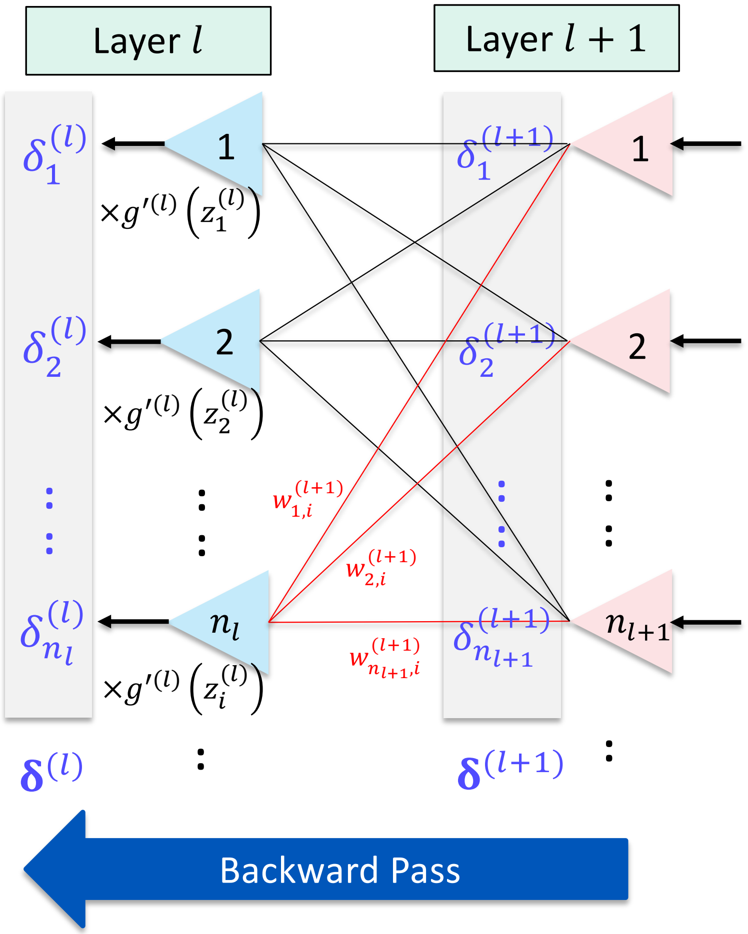
We begin at the end (last Layer L).

1. **Initialization: compute $\boldsymbol{\delta}^{(L)}$** based on $\nabla\mathcal{L}(\hat{\mathbf{y}})$
   - $\boldsymbol{\delta}^{(L)} = g'^{(L)}\left(\mathbf{z}^{(L)}\right) \odot \nabla\mathcal{L}(\hat{\mathbf{y}})$

2. **Compute $\boldsymbol{\delta}^{(l)}$ based on $\boldsymbol{\delta}^{(l+1)}$**
   - $\boldsymbol{\delta}^{(L-1)} = g'^{(L-1)}\left(\mathbf{z}^{(L-1)}\right) \odot \left(\mathbf{W}^{(L)}\right)^T \boldsymbol{\delta}^{(L)}$

     $\vdots$
   - $\boldsymbol{\delta}^{(l)} = g'^{(l)}\left(\mathbf{z}^{(l)}\right) \odot \left(\mathbf{W}^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)}$

     $\vdots$
   - $\boldsymbol{\delta}^{(1)} = g'^{(1)}\left(\mathbf{z}^{(1)}\right) \odot \left(\mathbf{W}^{(2)}\right)^T \boldsymbol{\delta}^{(2)}$



Synthesizing the Gradient

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$$

Backward Pass Term (δ)
Calculated from the error signal flowing back.

Forward Pass Term (a)
Calculated activation from the input flow.

The final gradient is dimply the product of the local error and the incoming activation.

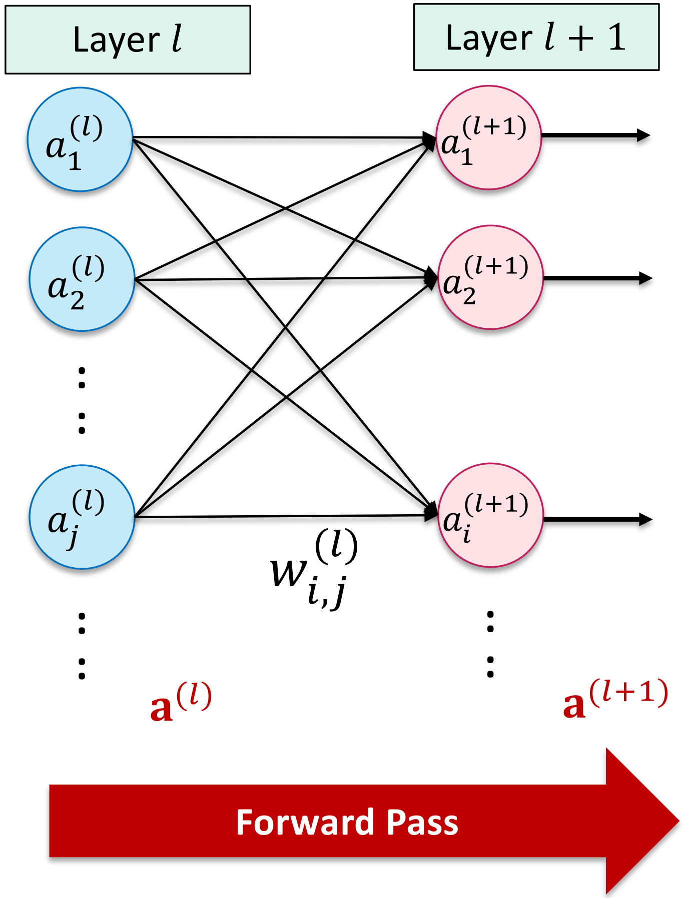
Forward Pass: Compute Activations

- $\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$

$$\frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} = a_j^{(l-1)} \quad \text{for } l = 1, a_j^{(0)} = x_j$$

Forward Pass

$$\mathbf{a}^{(0)} = \mathbf{x}$$
$$\mathbf{a}^{(1)} = g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^{(1)}\right)$$
$$\vdots$$
$$\mathbf{a}^{(l)} = g^{(l)}\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right)$$
$$\vdots$$
$$\mathbf{a}^{(L)} = g^{(L)}\left(\mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}\right)$$



Backward Pass: Compute Delta

- $$\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}.}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}}$$

$$\frac{\partial \mathcal{L}}{\partial z_i^{(l)}} = \delta_i^{(l)}$$

Backward Pass

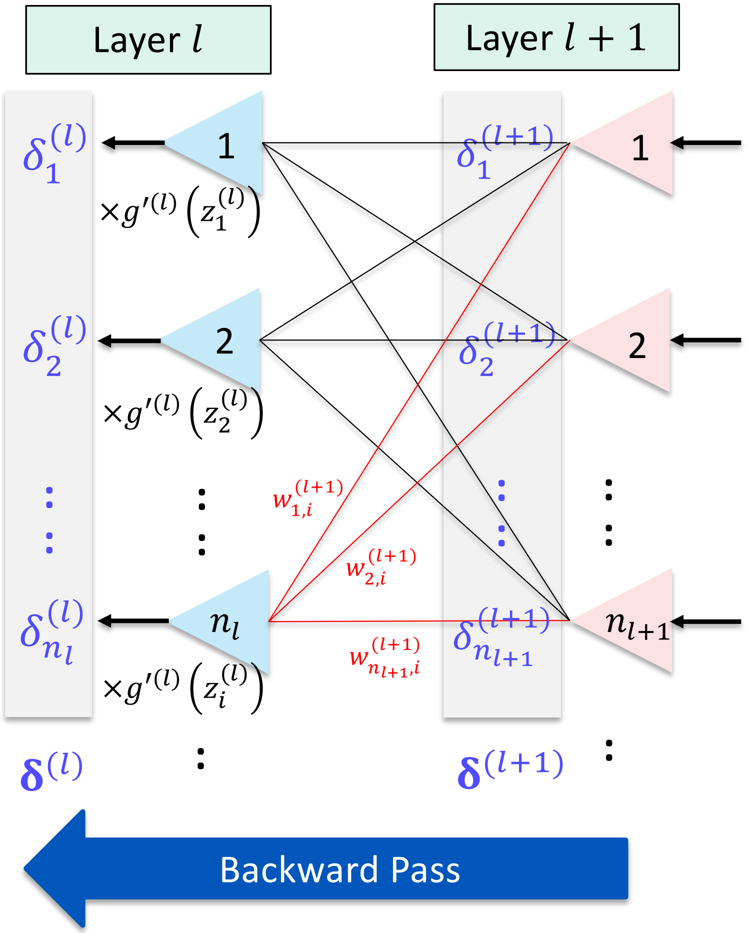
$$\boldsymbol{\delta}^{(L)} = g'^{(L)}\left(\mathbf{z}^{(L)}\right) \odot \nabla\mathcal{L}(\hat{\mathbf{y}})$$

$$\vdots$$

$$\boldsymbol{\delta}^{(l)} = g'^{(l)}\left(\mathbf{z}^{(l)}\right) \odot \left(\mathbf{W}^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)}$$

$$\vdots$$

$$\boldsymbol{\delta}^{(1)} = g'^{(1)}\left(\mathbf{z}^{(1)}\right) \odot \left(\mathbf{W}^{(2)}\right)^T \boldsymbol{\delta}^{(2)}$$



Synthesizing the Gradient

- $\theta_{new} = \theta_{old} - \eta \cdot \nabla\mathcal{L}(\theta_t)$
- $\nabla\mathcal{L} = \begin{bmatrix} \vdots \\ \frac{\partial\mathcal{L}}{\partial w_{i,j}^{(l)}} \\ \vdots \\ \frac{\partial\mathcal{L}}{\partial b_i^{(l)}} \end{bmatrix}$

$$\frac{\partial\mathcal{L}}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$$

Efficiently compute the gradient based on **two pre-computed terms** from forward $\mathbf{a}^{(l)}$ and $\boldsymbol{\delta}^{(l)}$ backward passes.

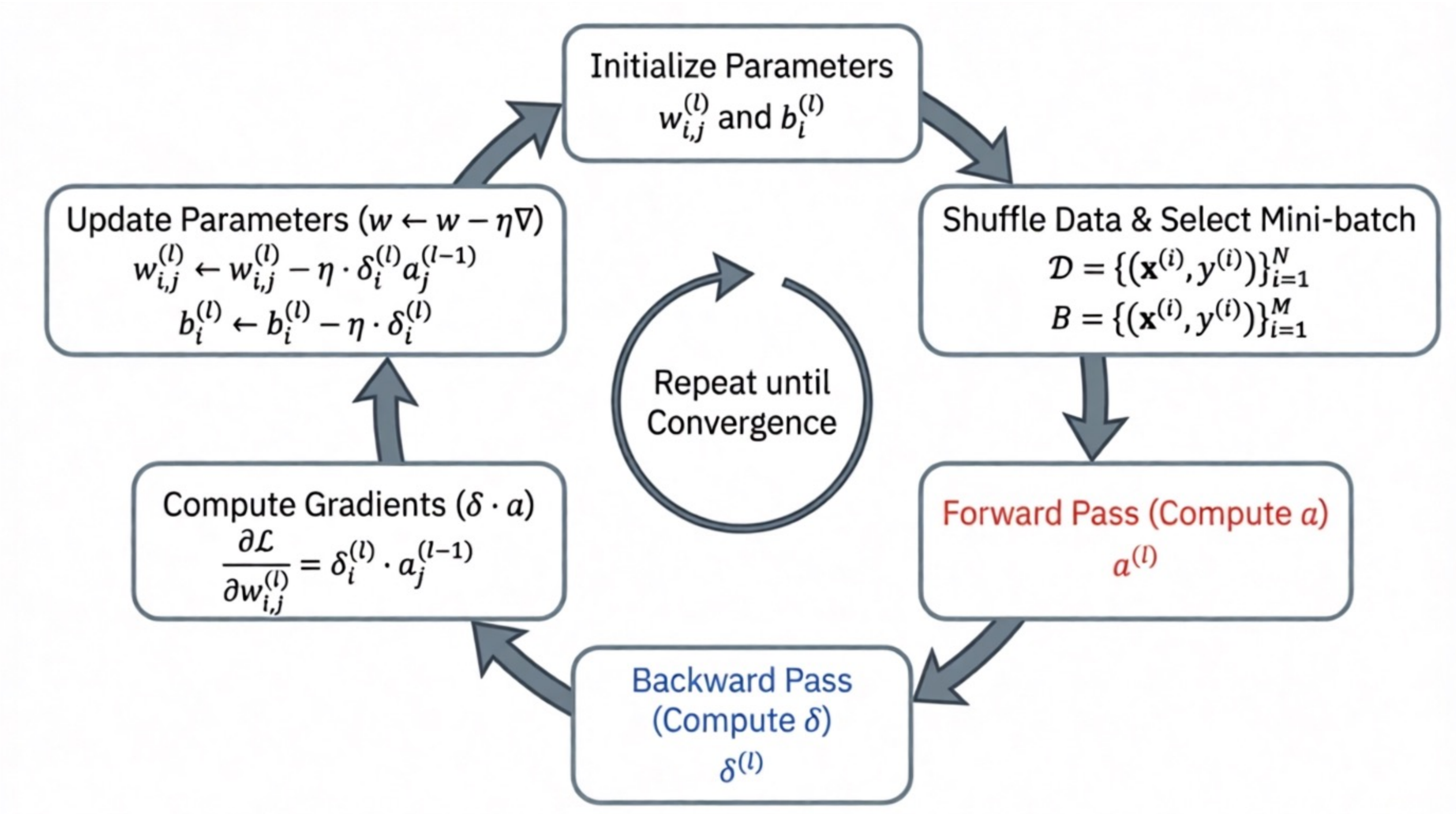
Backward Pass

$$\boldsymbol{\delta}^{(L)} = g'^{(L)}\left(\mathbf{z}^{(L)}\right) \odot \nabla\mathcal{L}(\hat{\mathbf{y}})$$
$$\vdots \qquad\qquad \vdots$$
$$\boldsymbol{\delta}^{(l)} = g'^{(l)}\left(\mathbf{z}^{(l)}\right) \odot \left(\mathbf{W}^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)}$$
$$\vdots \qquad\qquad \vdots$$
$$\boldsymbol{\delta}^{(1)} = g'^{(1)}\left(\mathbf{z}^{(1)}\right) \odot \left(\mathbf{W}^{(2)}\right)^T \boldsymbol{\delta}^{(2)}$$

Forward Pass

$$\mathbf{a}^{(0)} = \mathbf{x}$$
$$\mathbf{a}^{(1)} = g\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$
$$\vdots \qquad\qquad \vdots$$
$$\mathbf{a}^{(l)} = g\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right)$$
$$\vdots \qquad\qquad \vdots$$
$$\mathbf{a}^{(L)} = g\left(\mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}\right)$$

The Full Training Loop



Initialize Parameters

$w_{i,j}^{(l)}$ and $b_i^{(l)}$

Shuffle Data & Select Mini-batch

$\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$

$B = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{M}$

Forward Pass (Compute a)

$a^{(l)}$

Backward Pass (Compute δ)

$\delta^{(l)}$

Compute Gradients ($\delta \cdot a$)

$\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$

Update Parameters ($w \leftarrow w - \eta\nabla$)

$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \cdot \delta_i^{(l)} a_j^{(l-1)}$

$b_i^{(l)} \leftarrow b_i^{(l)} - \eta \cdot \delta_i^{(l)}$

Repeat until Convergence

Backpropagation Training Algorithm

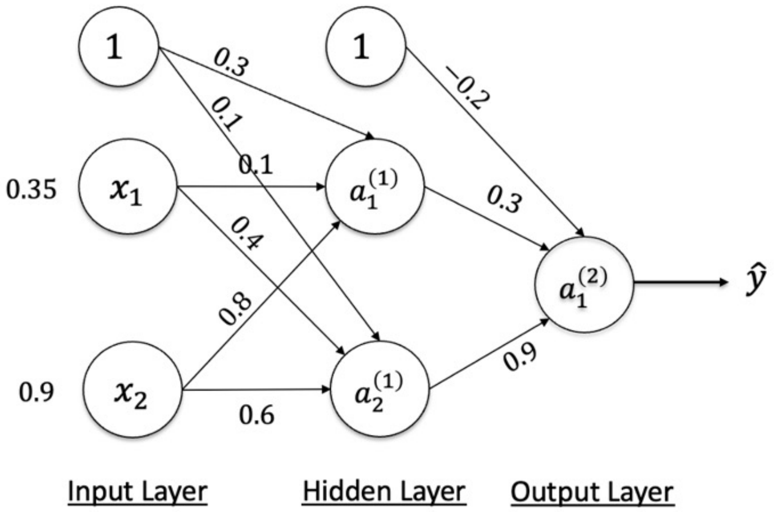
$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$$

$$\frac{\partial \mathcal{L}}{\partial b_i^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial b_i^{(l)}} = \delta_i^{(l)}$$

1. Initialize the model parameters $w_{i,j}^{(l)}$ and $b_i^{(l)}$.
2. Shuffle the training data $\mathcal{D} = \left\{\left(\mathbf{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}$.
3. Select a mini-batch from the shuffled data $B = \left\{\left(\mathbf{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{M}$.
4. Compute the gradients for the mini-batch.
   - Use Forward Pass to compute the activations $\mathbf{a}^{(1)}, \mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}$ of these samples
   - Use Backward Pass to compute the $\boldsymbol{\delta}^{(L)}, \boldsymbol{\delta}^{(L-1)}, \dots, \boldsymbol{\delta}^{(2)}, \boldsymbol{\delta}^{(1)}$ of these samples
   - Compute the gradients by $\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$ and $\frac{\partial \mathcal{L}}{\partial b_i^{(l)}} = \delta_i^{(l)}$
5. Update the parameters using the computed gradients.
   - $w_{i,j}^{(l)}(\text{new}) = w_{i,j}^{(l)}(\text{old}) - \eta \cdot \delta_i^{(l)} a_j^{(l-1)}$
   - $b_i^{(l)}(\text{new}) = b_i^{(l)}(\text{old}) - \eta \cdot \delta_i^{(l)}$
6. Repeat steps 3-5 for a specified no. of epoch or until a convergence criterion is met.

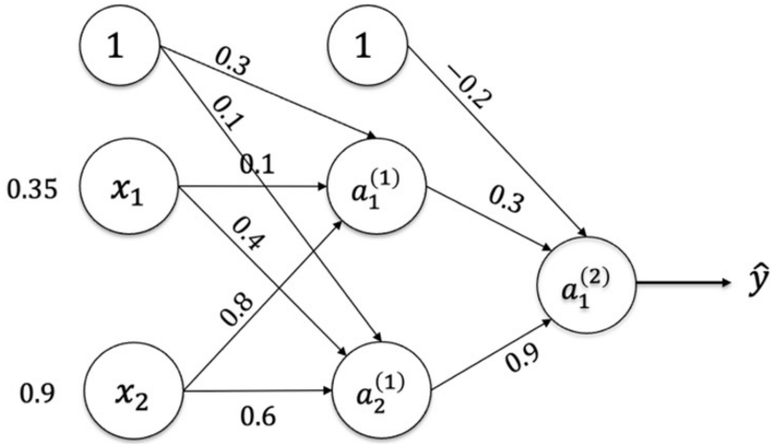
Backpropagation Exercise 1

- Given a two-layer feedforward neural network using **sigmoid activation** function for the hidden layer and the **identify activation function** for the output layer, determine the output $\hat{y}$ by representing the network in matrix form.
- Assume that actual output of the network y is 0.5, learning rate η is 0.5 and MSE cost function, perform the backpropagation to compute the new weights, new output and RMSE.



Solution Output of the Network

- $\hat{y} = \mathbf{W}^{(2)}\sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}$
- The net input vector $\mathbf{z}^{(1)}$ of the hidden layer is given by
  - $\mathbf{z}^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \end{bmatrix} = \mathbf{W}^{(1)}\ \mathbf{x} + \mathbf{b}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$
  - $\begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \end{bmatrix} = \begin{bmatrix} 0.1 & 0.8 \\ 0.4 & 0.6 \end{bmatrix}\begin{bmatrix} 0.35 \\ 0.9 \end{bmatrix} + \begin{bmatrix} 0.3 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 1.0550 \\ 0.7800 \end{bmatrix}$
- The activation vector $\mathbf{a}^{(1)}$ of the hidden layer is given by
  - $\mathbf{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \sigma\left(\mathbf{z}^{(1)}\right) = \sigma\left(\begin{bmatrix} 1.0550 \\ 0.7800 \end{bmatrix}\right) = \begin{bmatrix} 1/(1+e^{-1.055}) \\ 1/(1+e^{-0.7800}) \end{bmatrix} = \begin{bmatrix} 0.7417 \\ 0.6857 \end{bmatrix}$
- The net input vector $\mathbf{z}^{(2)}$ of the output layer is given by
  - $\mathbf{z}^{(2)} = \left[z_1^{(2)}\right] = \left[w_{1,1}^{(2)} \quad w_{1,2}^{(2)}\right]\begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} + \left[b_1^{(2)}\right] = [0.3 \quad 0.9]\begin{bmatrix} 0.7417 \\ 0.6857 \end{bmatrix} + [-0.2] = [0.6396]$
- The activation vector $\mathbf{a}^{(2)}$ of the output layer (output of the network $\hat{y}$) is given by
  - $\hat{y} = \mathbf{a}^{(2)} = \left[a_1^{(2)}\right] = g\left(\mathbf{z}^{(2)}\right) = \mathbf{z}^{(2)} = [0.6396]$



Output layer uses the identify function $g(z) = z$

Solution of the Backpropagation

- Assume that actual output y is 0.5, learning rate η is 1 and MSE cost function perform the backpropagation to find the updated weights.
  - $\mathcal{L}(\hat{y}) \ = \mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left\|\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}\right\|_2^2 = \frac{1}{2}(y - \hat{y})^2$
  - $\nabla\mathcal{L}(\hat{y}) = \frac{\partial\mathrm{MSE}}{\partial\hat{y}} = (\hat{y} - y) = (0.6396 - 0.5) = [0.1396]$

Backward Pass

$\boldsymbol{\delta}^{(2)} = g'\left(\mathbf{z}^{(2)}\right)\odot\nabla\mathcal{L}(\hat{\mathbf{y}})$

$\boldsymbol{\delta}^{(1)} = \sigma'\left(\mathbf{z}^{(1)}\right)\odot\left(\mathbf{W}^{(2)}\right)^{\mathrm{T}}\boldsymbol{\delta}^{(2)}$

Identify Activation Function

$g(z) = z$

$g'(z) = 1$

- Backward Pass: compute $\boldsymbol{\delta}^{(2)}$

$$\boldsymbol{\delta}^{(2)} = g'\left(\mathbf{z}^{(2)}\right)\odot\nabla\mathcal{L}(\hat{\mathbf{y}}) = 1\odot\nabla\mathcal{L}(\hat{y}) = (\hat{y} - y) = [0.1396]$$

Solution of the Backpropagation

- Backward Pass: compute $\boldsymbol{\delta}^{(1)}$

$$\boldsymbol{\delta}^{(1)} = \begin{bmatrix} \delta_1^{(1)} \\ \delta_2^{(1)} \end{bmatrix} = \sigma'\left(\mathbf{z}^{(1)}\right) \odot \left(\mathbf{W}^{(2)}\right)^T \boldsymbol{\delta}^{(2)}$$

$$= \sigma\left(\mathbf{z}_1^{(1)}\right)\left(1 - \sigma\left(\mathbf{z}_1^{(1)}\right)\right) \odot \left(\mathbf{W}^{(2)}\right)^T \boldsymbol{\delta}^{(2)}$$

$$= \sigma\left(\begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \end{bmatrix}\right)\left(1 - \sigma\left(\begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \end{bmatrix}\right)\right) \odot \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \end{bmatrix}^T \begin{bmatrix} \delta_1^{(2)} \end{bmatrix}$$

$$= \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix}\left(1 - \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix}\right) \odot \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \end{bmatrix}^T \begin{bmatrix} \delta_1^{(2)} \end{bmatrix}$$

$$= \begin{bmatrix} 0.7417 \\ 0.6857 \end{bmatrix}\left(1 - \begin{bmatrix} 0.7417 \\ 0.6857 \end{bmatrix}\right) \odot \begin{bmatrix} 0.3 \\ 0.9 \end{bmatrix} [0.1396] = \begin{bmatrix} 0.0080 \\ 0.0271 \end{bmatrix}$$

Backward Pass

$$\boldsymbol{\delta}^{(2)} = g'\left(\mathbf{z}^{(2)}\right) \odot \nabla J(\hat{\mathbf{y}})$$

$$\boldsymbol{\delta}^{(1)} = \sigma'\left(\mathbf{z}^{(1)}\right) \odot \left(\mathbf{W}^{(2)}\right)^{\mathrm{T}} \boldsymbol{\delta}^{(2)}$$

Sigmoid Activation Function

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

$$\sigma'(z) = \sigma(z)\left(1 - \sigma(z)\right)$$

Update the Parameters

$$w_{i,j}^{(l)}(\text{new}) = w_{i,j}^{(l)}(\text{old}) - \eta \cdot \delta_i^{(l)} a_j^{(l-1)}$$ and $\eta = 0.5$

- $w_{1,1}^{(1)}(\text{new}) = w_{1,1}^{(1)}(\text{old}) - 0.5 \cdot \delta_1^{(1)} x_1 = 0.0986$
- $w_{1,2}^{(1)}(\text{new}) = w_{1,2}^{(1)}(\text{old}) - 0.5 \cdot \delta_1^{(1)} x_2 = 0.7964$
- $w_{2,1}^{(1)}(\text{new}) = w_{2,1}^{(1)}(\text{old}) - 0.5 \cdot \delta_2^{(1)} x_1 = 0.3953$
- $w_{2,2}^{(1)}(\text{new}) = w_{2,2}^{(1)}(\text{old}) - 0.5 \cdot \delta_2^{(1)} x_2 = 0.5878$

- $b_1^{(1)}(\text{new}) = b_1^{(1)}(\text{old}) - 0.5 \cdot \delta_1^{(1)} = 0.2920$
- $b_2^{(1)}(\text{new}) = b_{2,}^{(1)}(\text{old}) - 0.5 \cdot \delta_2^{(1)} =0.0865$

- $w_{1,1}^{(2)}(\text{new}) = w_{1,1}^{(2)}(\text{old}) - 0.5 \cdot \delta_1^{(2)} a_1^{(2)} = 0.2482$
- $w_{1,2}^{(2)}(\text{new}) = w_{1,2}^{(2)}(\text{old}) - 0.5 \cdot \delta_1^{(2)} a_2^{(2)} = 0.8521$

- $b_1^{(2)}(\text{new}) = b_1^{(2)}(\text{old}) - 0.5 \cdot \delta_2^{(2)} = -0.2698$

Forward Pass to Compute the New Output

- Compute the new output for $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(2)}$ $(\hat{y})$ using the updated weights

$$\mathbf{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \sigma\left(\begin{bmatrix} 0.0986 & 0.7964 \\ 0.3953 & 0.5878 \end{bmatrix}\begin{bmatrix} 0.35 \\ 0.9 \end{bmatrix} + \begin{bmatrix} 0.2920 \\ 0.0865 \end{bmatrix}\right) = \begin{bmatrix} 0.7402 \\ 0.6800 \end{bmatrix}$$

$$\hat{y}(\text{new}) = \mathbf{a}^{(2)} = \left[a_1^{(2)}\right] = \left[z_1^{(2)}\right] = [0.2482 \quad 0.8521]\begin{bmatrix} 0.7402 \\ 0.6800 \end{bmatrix} - [0.2698] = 0.4934$$

- The new error
  - $\text{error} = y - \hat{y} = (0.5 - 0.4934) = -0.0066$
- RMSE (Root Mead Square Error):
  - $\text{RMSE} = \sqrt{(0.5 - 0.4934)^2} = 0.0066$

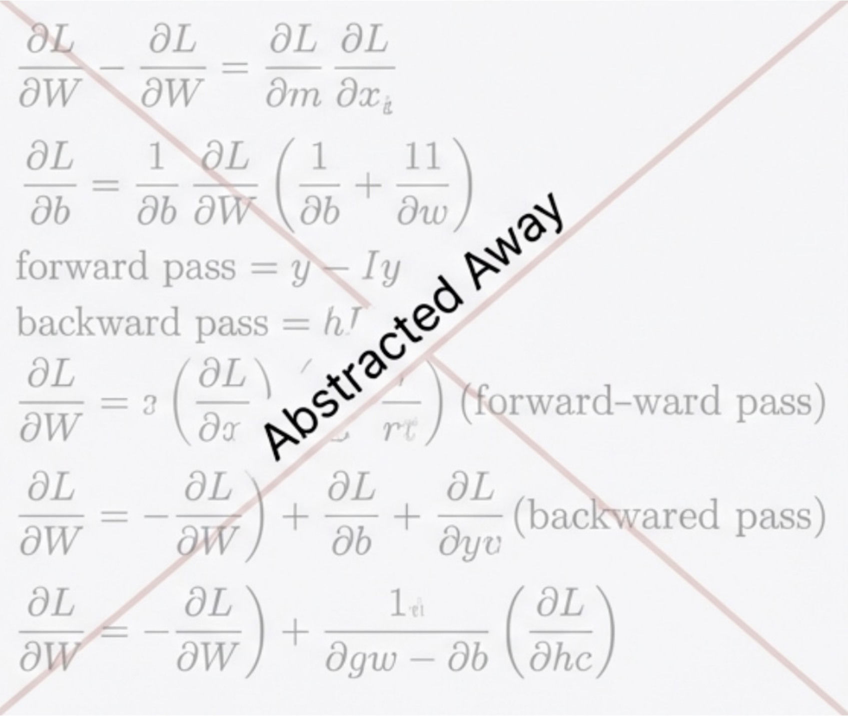
PyTorch
Automatic Differentiation

(Optional)

https://medium.com/@lmpo/pytorch-automatic-differentiation-autograd-772fba79e6ef

Modern Implementation: Automatic Differentiation

Manual Math



$$\frac{\partial L}{\partial W} - \frac{\partial L}{\partial W} = \frac{\partial L}{\partial m}\frac{\partial L}{\partial x_i}$$

$$\frac{\partial L}{\partial b} = \frac{1}{\partial b}\frac{\partial L}{\partial W}\left(\frac{1}{\partial b} + \frac{11}{\partial w}\right)$$

$$\text{forward pass} = y - Iy$$

$$\text{backward pass} = hJ$$

$$\frac{\partial L}{\partial W} = \partial \left(\frac{\partial L}{\partial x}\right) \ldots \frac{}{r\iota}\Big) \text{(forward-ward pass)}$$

$$\frac{\partial L}{\partial W} = -\frac{\partial L}{\partial W}\Big) + \frac{\partial L}{\partial b} + \frac{\partial L}{\partial yv} \text{(backwared pass)}$$

$$\frac{\partial L}{\partial W} = -\frac{\partial L}{\partial W}\Big) + \frac{1}{\partial gw - \partial b}\left(\frac{\partial L}{\partial hc}\right)$$

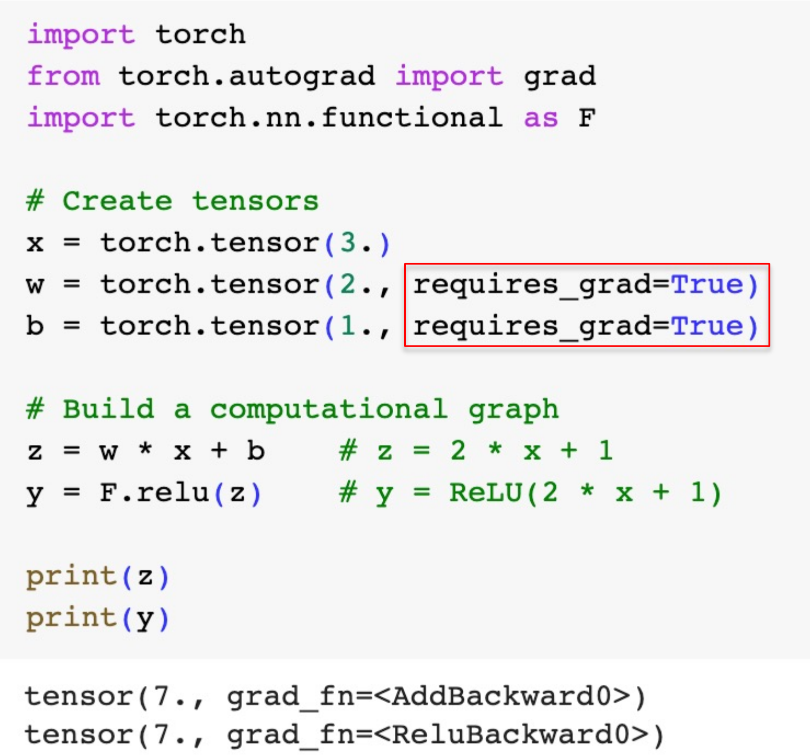
Modern Code

```
loss = criterion(y_pred, _target)
loss. backward()
optimizer.step()
```

AutoGrad: Modern frameworks like **PyTorch** automatically construct the computational graph and compute gradients, allowing researchers to focus on architecture rather than calculus

Colab: PyTorch Autogard Example

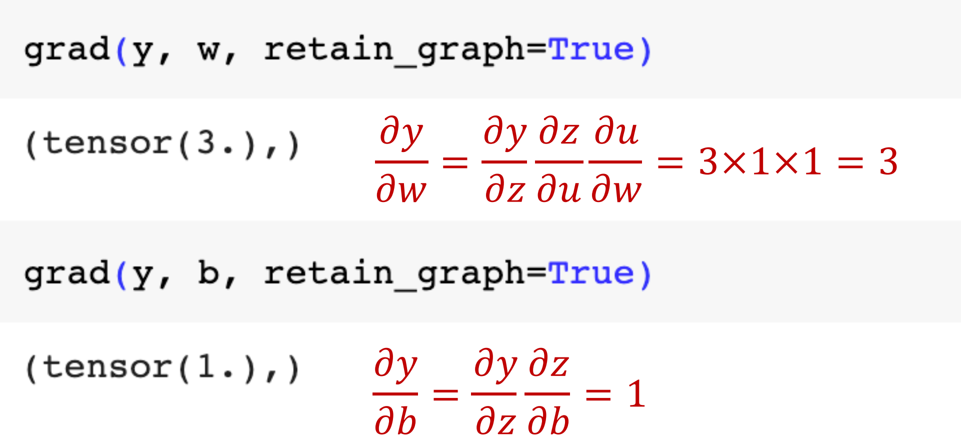
- https://colab.research.google.com/drive/1MvtZnvrS-1Npk8s_Fq8RCEoOATUT5jNq?usp=sharing



```
import torch
from torch.autograd import grad
import torch.nn.functional as F

# Create tensors
x = torch.tensor(3.)
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(1., requires_grad=True)

# Build a computational graph
z = w * x + b     # z = 2 * x + 1
y = F.relu(z)     # y = ReLU(2 * x + 1)

print(z)
print(y)
```

```
tensor(7., grad_fn=<AddBackward0>)
tensor(7., grad_fn=<ReluBackward0>)
```

Let's calculate the derivative of y with respect to w in the equation y = w * x + b. This will give us the gradient of y based on changes in w.



```
grad(y, w, retain_graph=True)
```

```
(tensor(3.),)
```

$$\frac{\partial y}{\partial w} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial u}\frac{\partial u}{\partial w} = 3\times1\times1 = 3$$

```
grad(y, b, retain_graph=True)
```

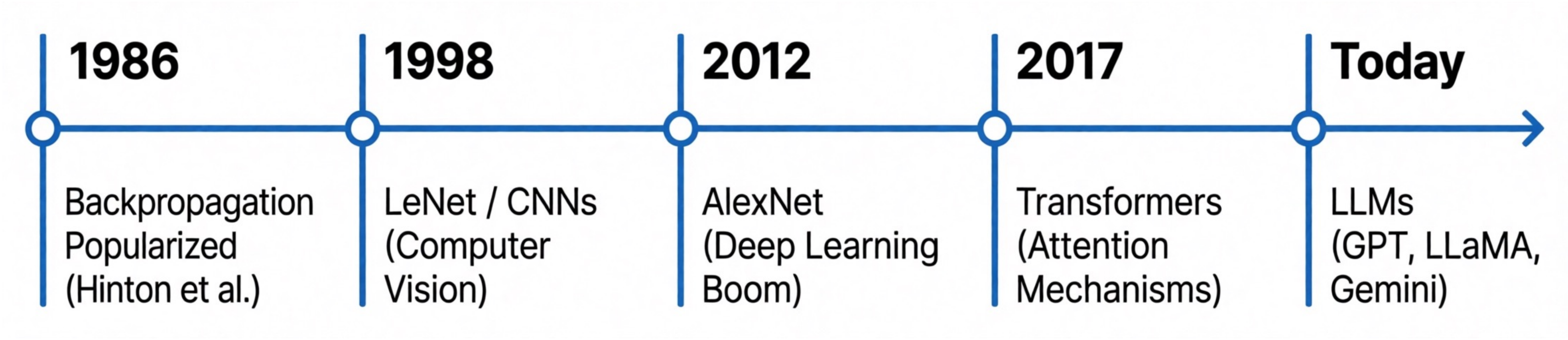
```
(tensor(1.),)
```

$$\frac{\partial y}{\partial b} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial b} = 1$$

https://medium.com/@thevnotebook/introduction-to-pytorch-4-7-a4fdcd6a497b

The Impact: From Perceptrons to Transformers

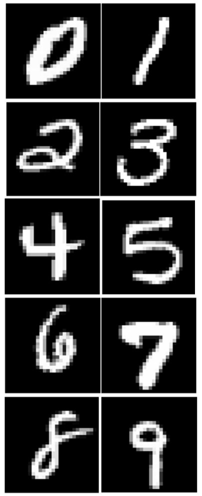
- Backpropagation remains the standard optimization engine for the massive models of today.



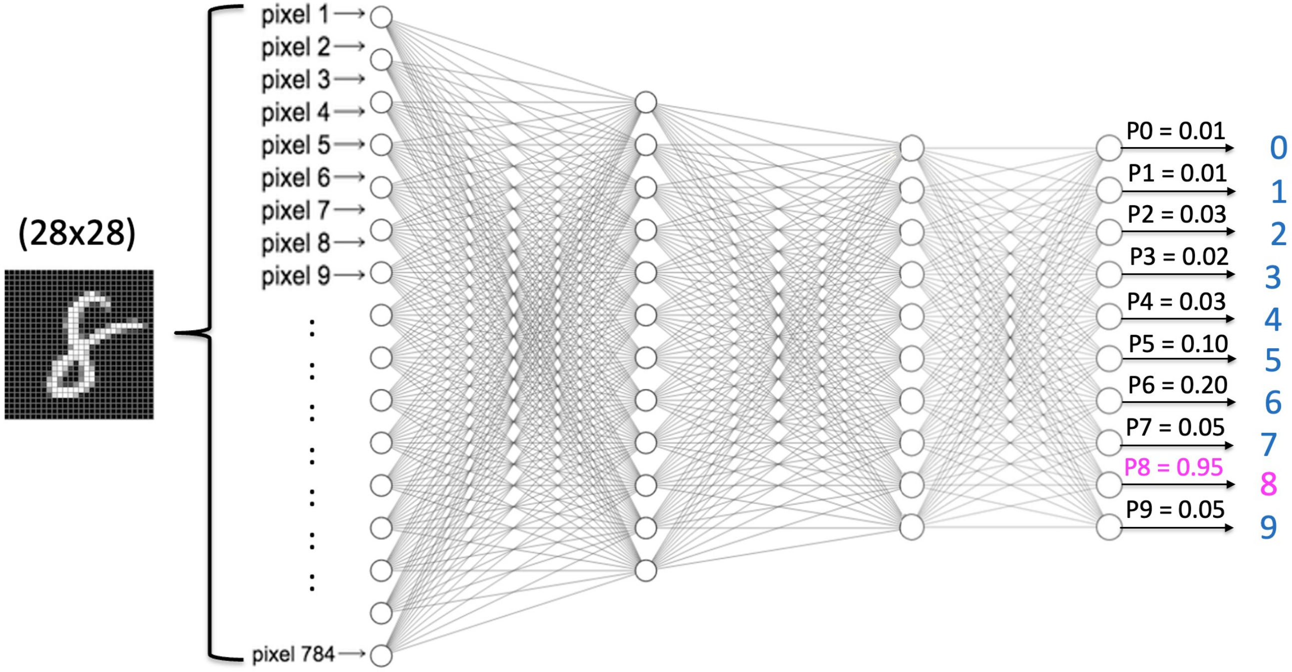
PyTorch Example for Image Classifications

Colab: MLP using NMIST Dataset

- In this example, we will use the PyTorch deep learning framework to create a MLP model that can recognize handwritten digits. We will train this model using a dataset called MNIST, which has 70,000 images of handwritten digits from 0 to 9.
- https://colab.research.google.com/drive/1roufrBO8BZfJA1HDgoi5uyZpgS_FCuu1?usp=sharing

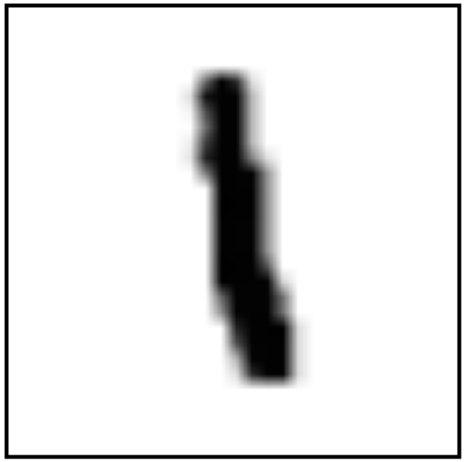


Handwritten digits from 0 to 9

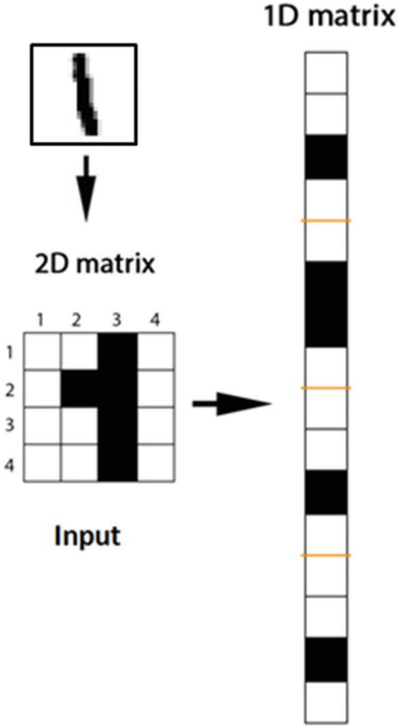


MNIST Image Format

- Each MNIST image is 28 pixels by 28 pixels. We can interpret this as a big array of numbers:



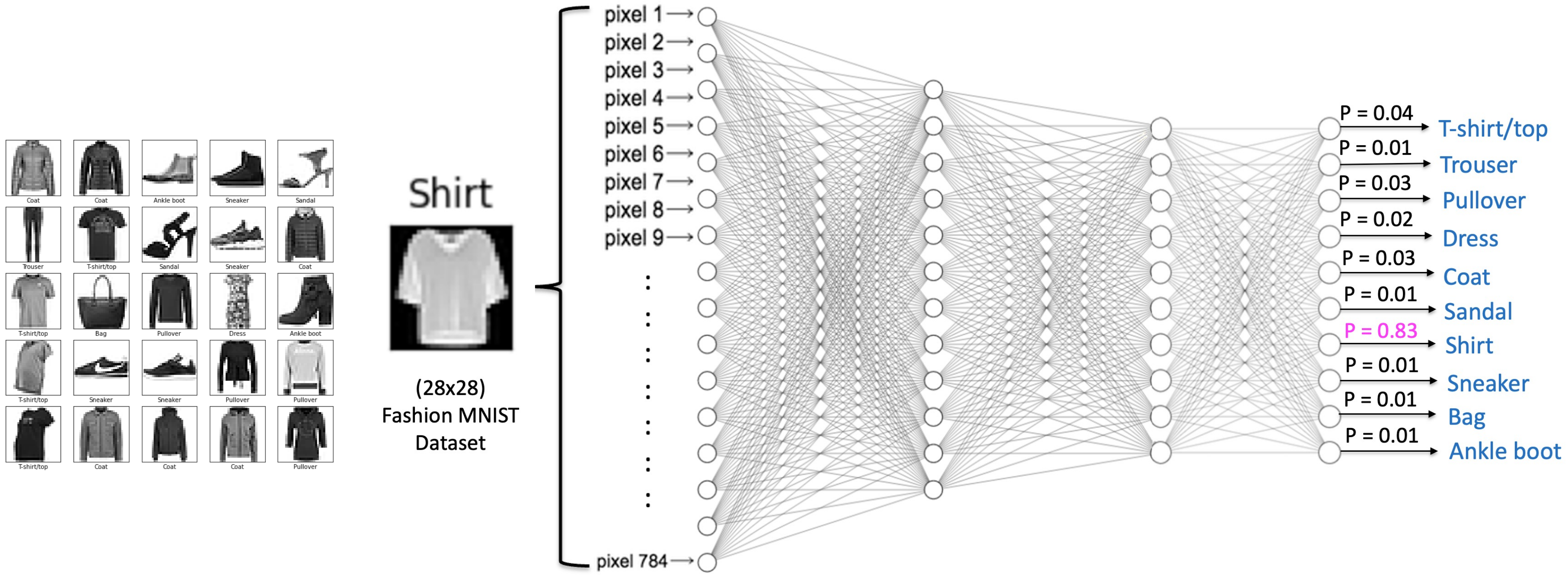
$$\cong \begin{bmatrix} 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&.6&.8&0&0&0&0&0&0 \\ 0&0&0&0&0&0&.7&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&.7&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&.5&1&.4&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&.4&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&.4&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&.7&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&.9&1&.1&0&0&0&0 \\ 0&0&0&0&0&0&0&.3&1&.1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \end{bmatrix}$$



- Thus, after flattening the image into vectors of 28*28=784, we obtain as mnist.train.images a tensor (an n-dimensional array) with a shape of [55000, 784].

Colab: MLP using Fashion NMIST Dataset

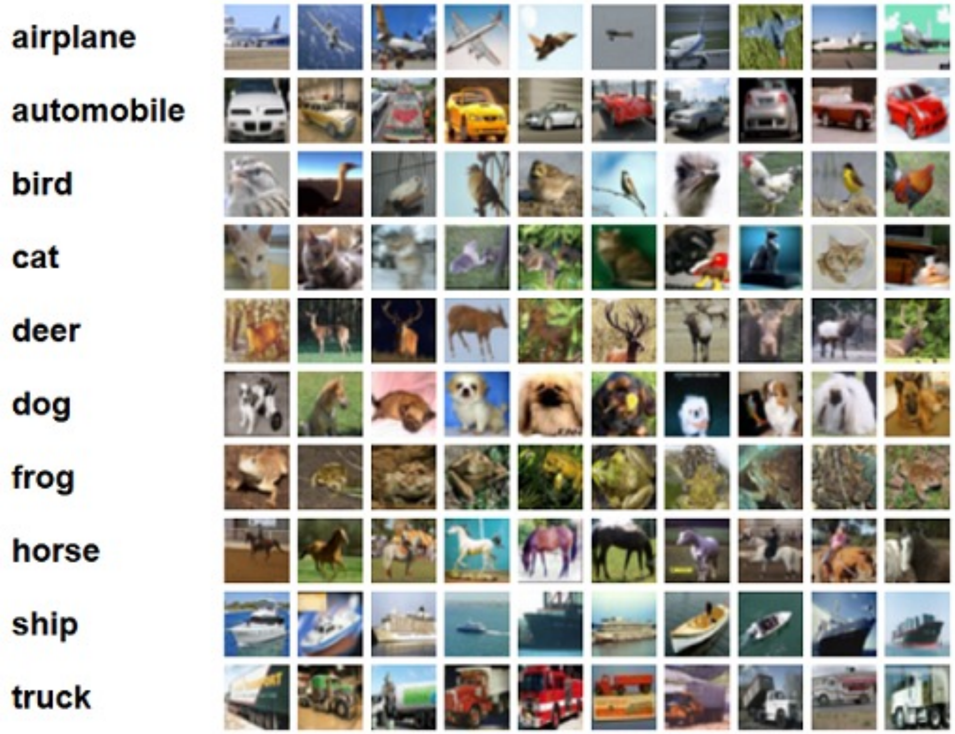
- We will train an MLP to classify images from the Fashion MNIST dataset, which consists of 70,000 grayscale fashion product images. Each image is 28x28 pixels in size.



https://colab.research.google.com/drive/15S3-F0wCA4o3Scs6rchqLR96zxkNDoHA?usp=sharing

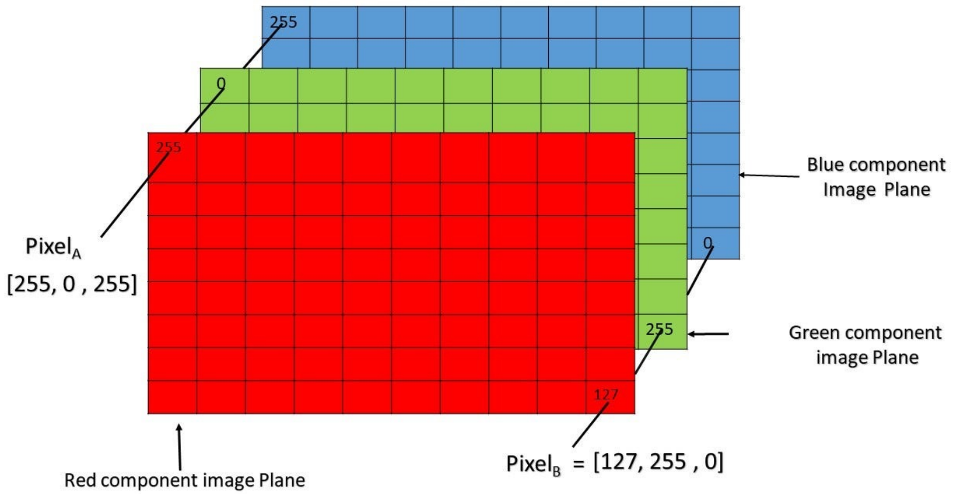
CIFAR-10 Color Image Dataset

- The **CIFAR-10 dataset** is a widely used collection of color images that is commonly used to train machine learning and computer vision algorithms
  - It consists of 60,000 32x32 color images in 10 different classes
  - Each class contains 6,000 images, with 5,000 images for training and 1,000 images for testing
  - The 10 different classes in the CIFAR-10 dataset represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks



CIFAR-10 Image Format

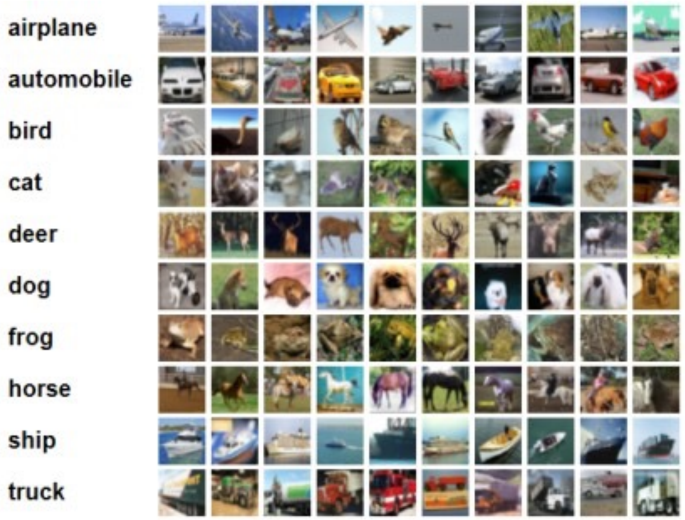
- The images in CIFAR-10 dataset are of (32 x 32) resolution and color images, which means they are in RGB format.
- Every image is of a shape (32,32,3) where 3 represent its number of channels-RGB, RED, GREEN and BLUE.
- Every image in this dataset is a mixture of these **3 color images**.
- All these images are in form of pixels, like in this particular data 32 x 32, means a matrix of 32 x 32 pixel values for 3 different channels.

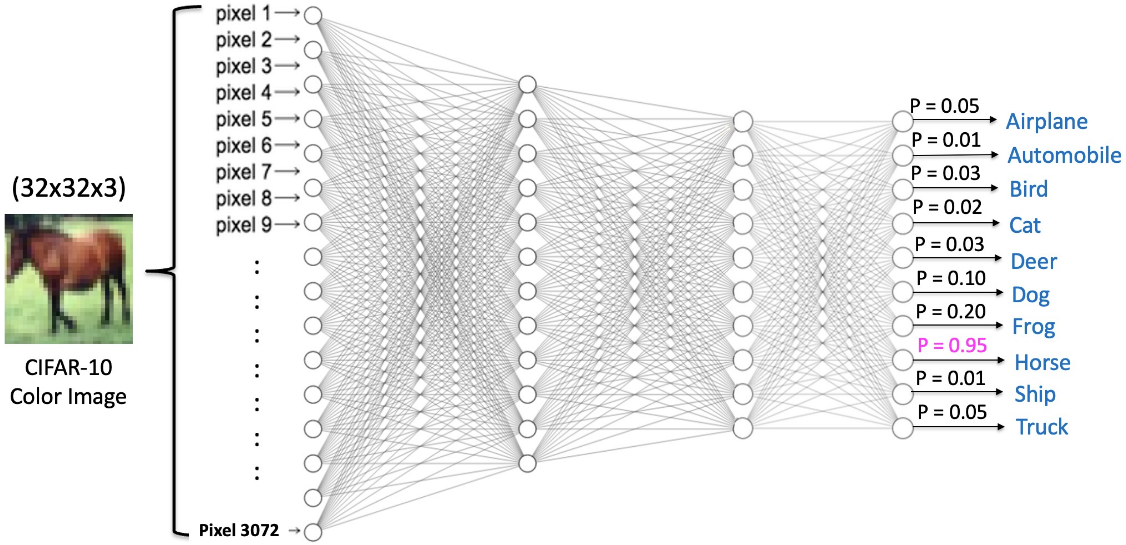


Pixel of an RGB image are formed from the corresponding pixel of the three component images

Colab: MLP using CIFAR-10 Dataset

- In this example, we demonstrate how to train a MLP model (or feedforward neural network) to classify images from the CIFAR-10 dataset. The images are be flattened into a **3072**-dimensional vector before being fed into the network.

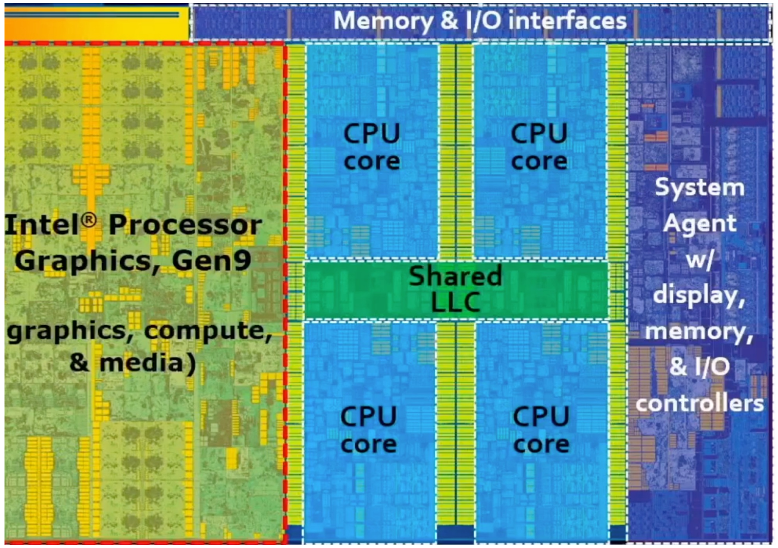




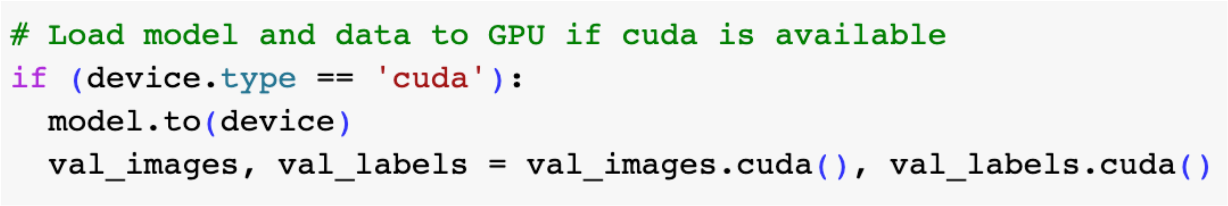
https://colab.research.google.com/drive/1vbFi4_6gZ_-bPhBFEdkoSOc3syjshoXP?usp=sharing

CPU vs GPU

- CPU: Small number of large cores
- GPU: Large number of small cores



Model & Data

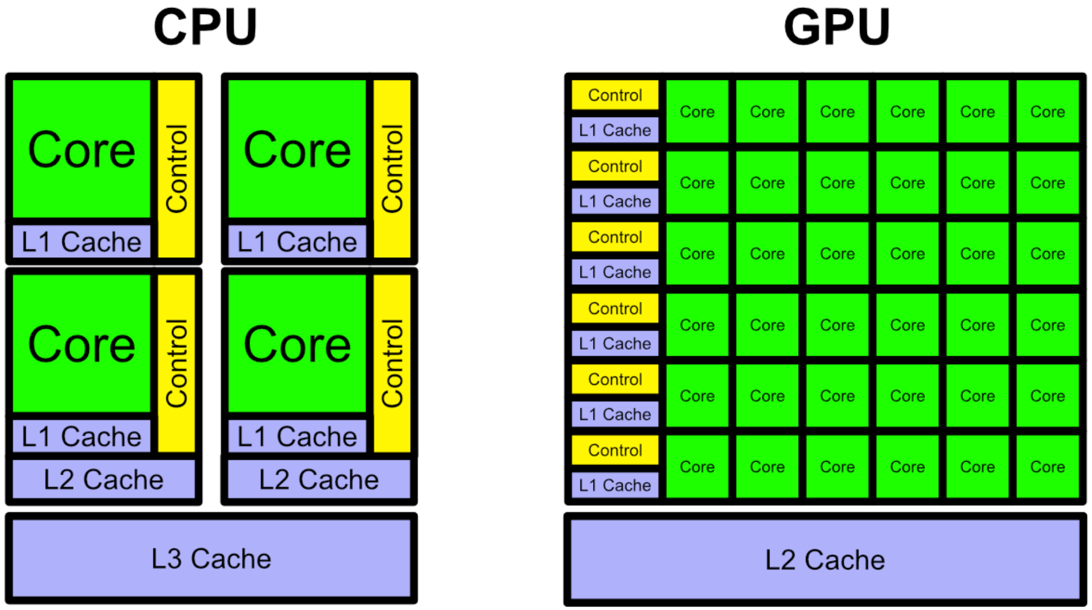


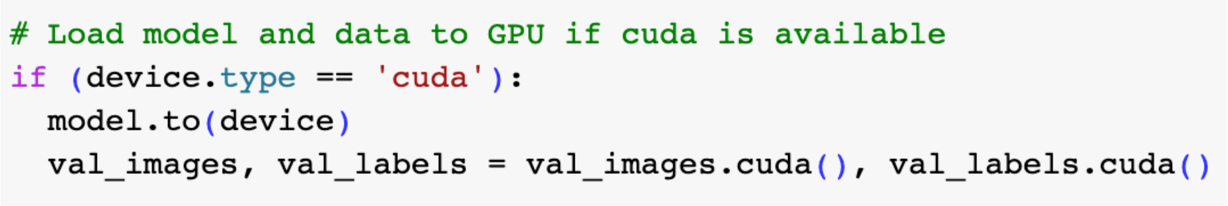
```
# Load model and data to GPU if cuda is available
if (device.type == 'cuda'):
  model.to(device)
  val_images, val_labels = val_images.cuda(), val_labels.cuda()
```

https://towardsdatascience.com/why-deep-learning-models-run-faster-on-gpus-a-brief-introduction-to-cuda-programming-035272906d66

CPU vs GPU

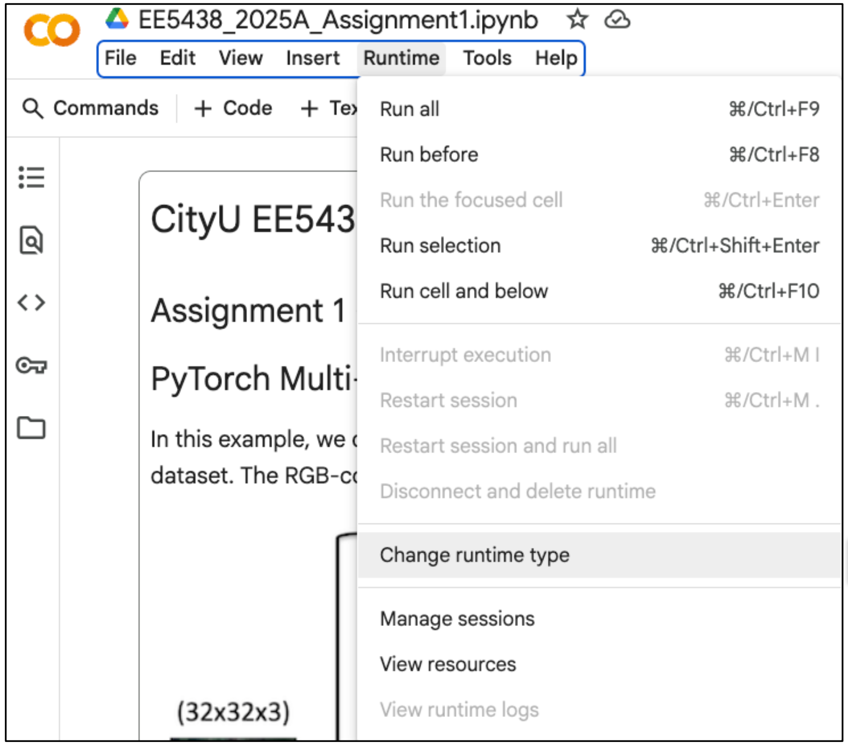
- There are many similarities between the CPU and GPU, but the focus on individual operation speed vs parallelism has major implications in terms of performance.

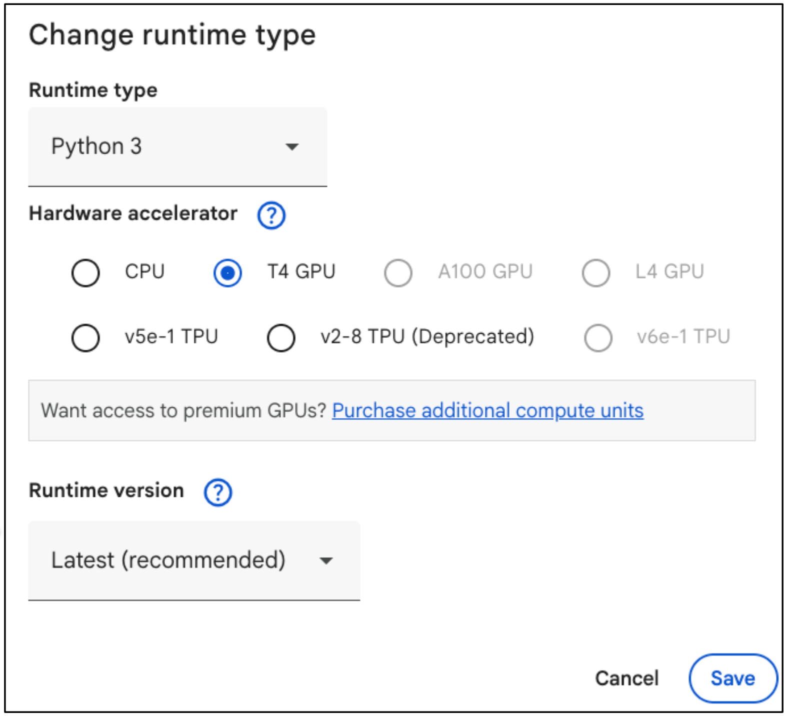




```
# Load model and data to GPU if cuda is available
if (device.type == 'cuda'):
  model.to(device)
  val_images, val_labels = val_images.cuda(), val_labels.cuda()
```

Colab: GPUs: T4, A100, L4, …





Change runtime type

Runtime type

Python 3

Hardware accelerator ?

○ CPU ● T4 GPU ○ A100 GPU ○ L4 GPU

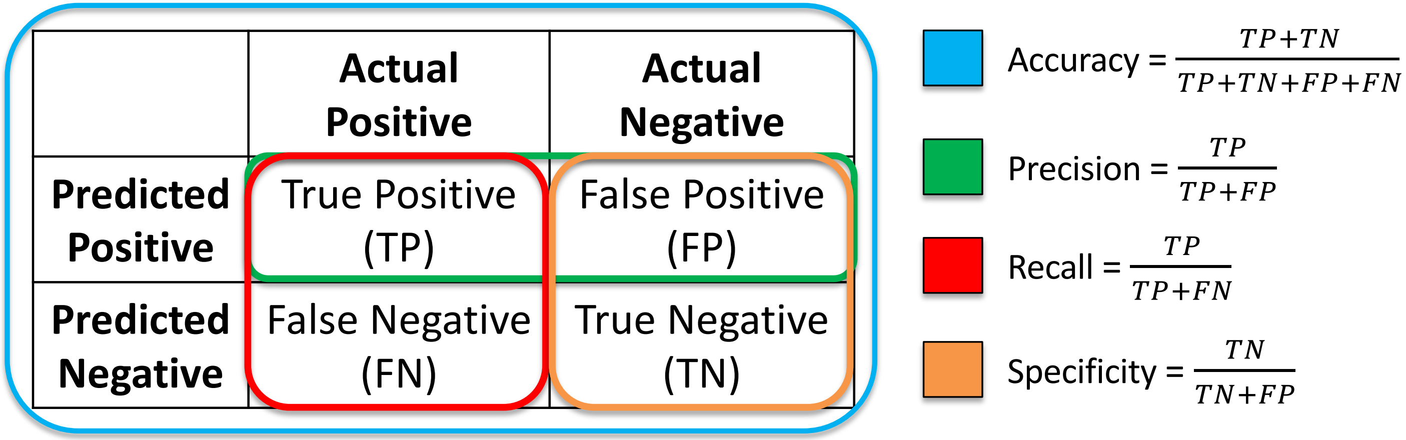
○ v5e-1 TPU ○ v2-8 TPU (Deprecated) ○ v6e-1 TPU

Want access to premium GPUs? Purchase additional compute units

Runtime version ?

Latest (recommended)

Classification Metrics



	Actual Positive	Actual Negative
Predicted Positive	True Positive (TP)	False Positive (FP)
Predicted Negative	False Negative (FN)	True Negative (TN)

🟦 Accuracy = $\frac{TP+TN}{TP+TN+FP+FN}$

🟩 Precision = $\frac{TP}{TP+FP}$

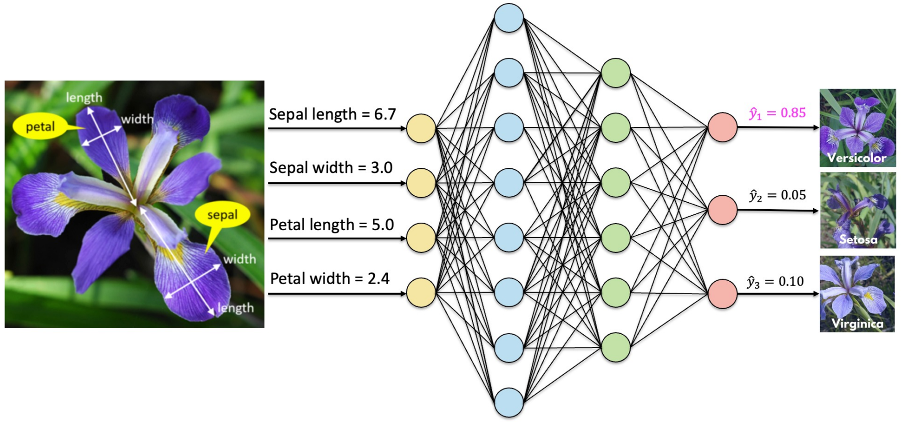
🟥 Recall = $\frac{TP}{TP+FN}$

🟧 Specificity = $\frac{TN}{TN+FP}$

https://medium.com/@lmpo/mastering-classification-metrics-a-deep-dive-into-accuracy-precision-recall-f1-score-and-f8caaf669bf0

Classification Metrics

- **Classification** is the problem of identifying to **which of a set of categories, a new observation** belongs to, based on **a training set** of data containing observations and **whose categories membership is known**.



- How to **measure the performance** of the trained classifier?

Terminologies of Classification Metrics

After a deep learning model is trained to detect a COVID-19 disease on patents. The output can either be positive (+ve) or negative (-ve)

There are only 4 cases any patient X could end up with:

1. **True positive (TP)**: Prediction is +ve and X is infected.
2. **True negative (TN)**: Prediction is –ve and X is healthy
3. **False positive (FP)**: Prediction is +ve and X is healthy.
4. **False negative (FN)**: Prediction is –ve and X is infected.

Detecting COVID-19 Disease

https://towardsdatascience.com/identifying-the-right-classification-metric-for-your-task-21727fa218a2

Confusion Matrix

- A **confusion matrix** specific table layout that allows visualization of the performance of supervised classification algorithm
- Typically row of the matrix represents the instances in a predicted class, while column represents the instances in an actual class.
- Its name stems from the fact that it makes it easy to see whether the **system is confusing two classes**.

Confusion Matrix

	Actual Positive	**Actual Negative**
Predicted Positive	True Positive (*TP*)	False Positive (*FP*)
Predicted Negative	False Negative (*FN*)	True Negative (*TN*)

https://towardsdatascience.com/identifying-the-right-classification-metric-for-your-task-21727fa218a2

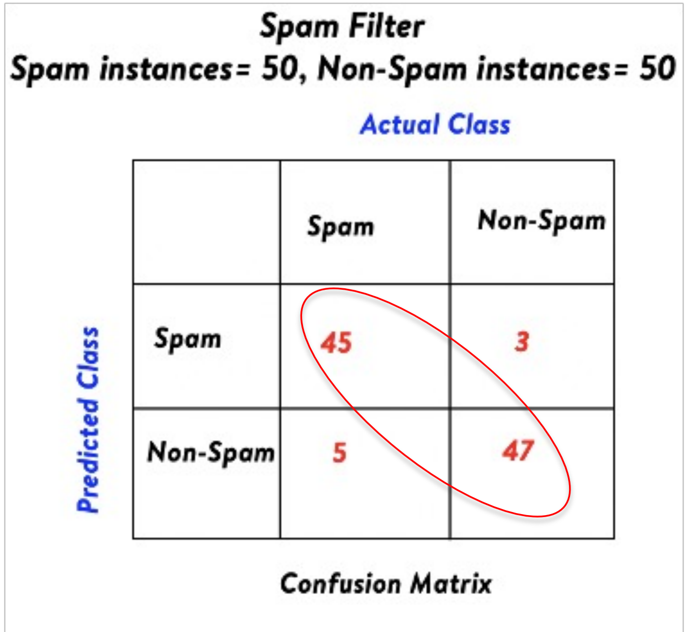
Classification Metric: Accuracy

- **Accuracy** is the ratio of the correctly labeled instances to the whole pool of instances.
- Accuracy is the most intuitive Classification metric.
- Accuracy answers the question that :
  - How many people were correctly labelled out of all the people?
- **Accuracy** = $\frac{TP+TN}{TP+FP+FN+TN}$
- Numerator: All correctly labeled people (TP+TN)
- Denominator: Al people (TP+FP+FN+TN)

	Actual Positive	**Actual Negative**
Predicted Positive	**True Positive (*TP*)**	False Positive (*FP*)
Predicted Negative	False Negative (*FN*)	**True Negative (*TN*)**

Weakness of Accuracy Metric

- Accuracy is a good metric only in the following cases.
  - The classes or categories have **evenly distributed instances i.e. It is a balanced dataset.**
  - The cost of false positives is the same as the cost of false negatives.

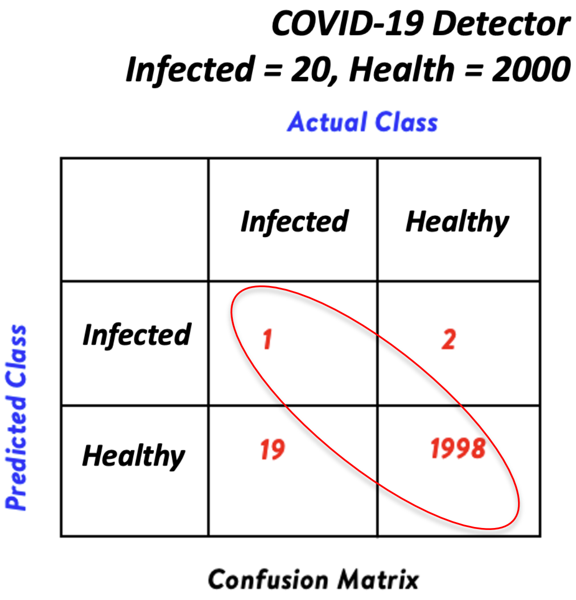


Spam Filter
Spam instances= 50, Non-Spam instances= 50

		Actual Class	
		Spam	Non-Spam
Predicted Class	Spam	45	3
	Non-Spam	5	47

Confusion Matrix

Accuracy = (45+47)/100 = 92%



COVID-19 Detector
Infected = 20, Health = 2000

		Actual Class	
		Infected	Healthy
Predicted Class	Infected	1	2
	Healthy	19	1998

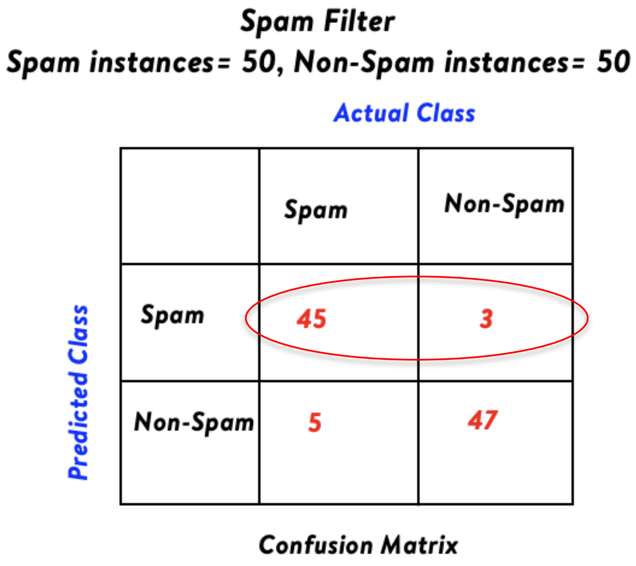
Confusion Matrix

Accuracy = (1998+1)/2020 = 99%

The model can only **identify 1 out of 20** obscene cases

Classification Metric: Precision

- **Precision** is the ratio of the **correctly positive labeled instances** by the model to **all positive labeled instances**.
- Precision answers the question: How many of those who we labeled as positive are actually positive?
- Precision = $\frac{TP}{TP+FP}$
- Choose precision if you want to be more confident of your True positive.



Spam Filter

Spam instances= 50, Non-Spam instances= 50

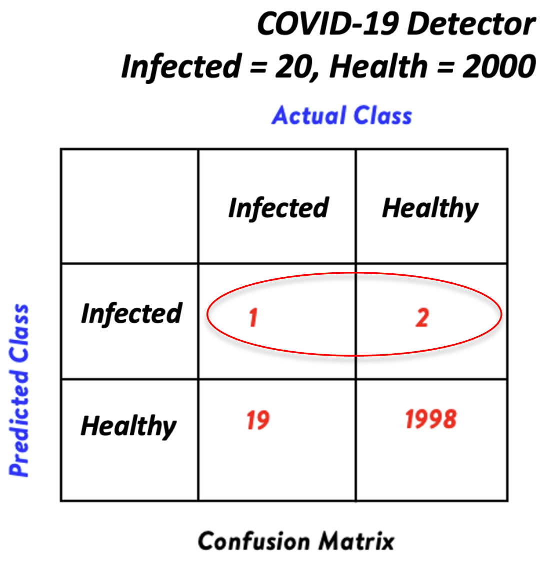
Actual Class

Predicted Class

	Spam	Non-Spam
Spam	45	3
Non-Spam	5	47

Confusion Matrix

Precision = 45/(45+3) = 93.75%



COVID-19 Detector

Infected = 20, Health = 2000

Actual Class

Predicted Class

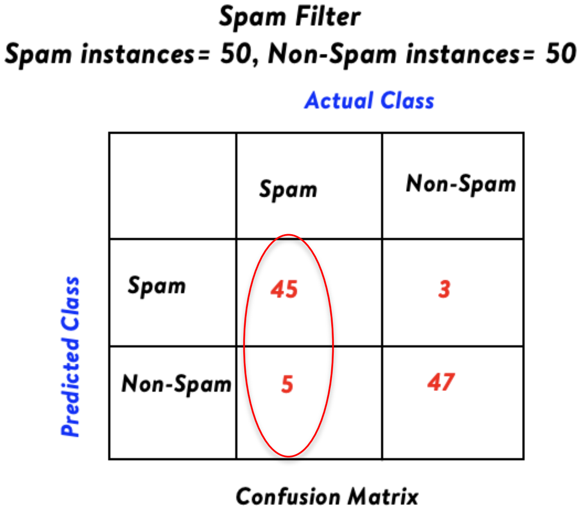
	Infected	Healthy
Infected	1	2
Healthy	19	1998

Confusion Matrix

Precision = 1/(1+2) = 33.33%

Classification Metric: Recall/Sensitivity

- **Recall/Sensitivity** is the **True Positive Rate** (TPR)
- Recall is the ratio of the correctly **positive** labeled instances by a model to all who are positive.
- Recall answers the question: Of all the instances who are positive, how many of these are correctly detected.
- Recall = $\frac{TP}{TP+FN}$
- Precision is how sure we are of True Positives, while Recall is how sure we are that we are not missing any positives

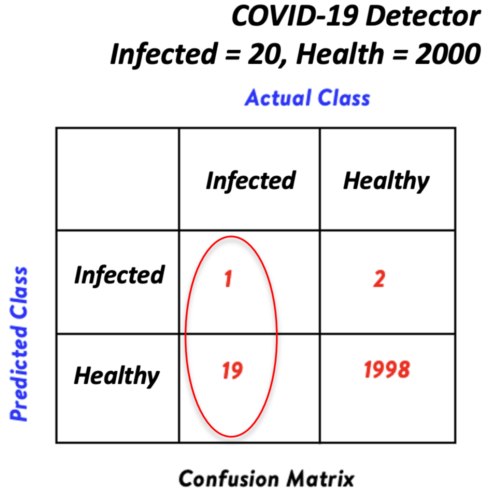


Spam Filter
Spam instances= 50, Non-Spam instances= 50

		Actual Class	
		Spam	Non-Spam
Predicted Class	Spam	45	3
	Non-Spam	5	47

Confusion Matrix

Recall = 45/(45+5) = 90%



COVID-19 Detector
Infected = 20, Health = 2000

		Actual Class	
		Infected	Healthy
Predicted Class	Infected	1	2
	Healthy	19	1998

Confusion Matrix

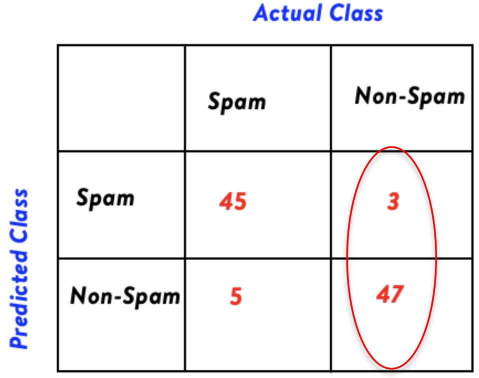
Recall = 1/(1+19) = 5%

Classification Metric: Specificity

- **Specificity** is the True **Negative** Rate (TNR)
- Specificity is the ratio of the correctly **negative** labeled instances by a model to all who are **actually negative**.
- Recall answers the question: Of all the instances who are positive, how many of these are correctly detected.
- Specificity = $\frac{TN}{TN+FP}$
- **Specificity is preferred** when we want to cover all the negatives, meaning we don't want any false alarms, we don't want any false positives.

Spam Filter

Spam instances= 50, Non-Spam instances= 50



Actual Class

Predicted Class

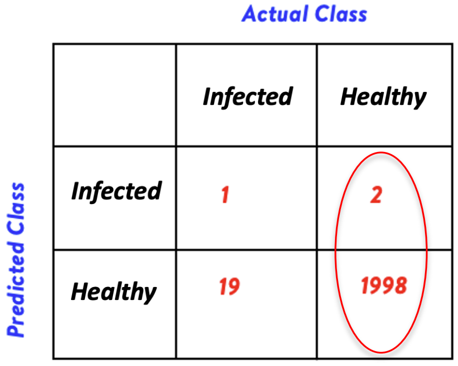
	Spam	Non-Spam
Spam	45	3
Non-Spam	5	47

Confusion Matrix

Specificity = 47/(3+47) = 94%

COVID-19 Detector

Infected = 20, Health = 2000



Actual Class

Predicted Class

	Infected	Healthy
Infected	1	2
Healthy	19	1998

Confusion Matrix

Specificity = 1998/(2+1998) = 99.9%

Classification Metric: F1-Score

- **F1-Score** considers both Precision and Recall
- F1-Score is the harmonic mean of the Precision and Recall.
- **F1 Score is the preferred metric in case of an imbalanced dataset.**
- F1-Score $= \dfrac{1}{\frac{1}{2}\cdot\frac{1}{\text{Precision}} + \frac{1}{2}\cdot\frac{1}{\text{Rcall}}} = 2\dfrac{(\text{Precisio} \times \text{Recall})}{(\text{Precisio} + \text{Recall})}$

Spam Filter
Spam instances= 50, Non-Spam instances= 50

Actual Class

		Spam	**Non-Spam**
Predicted Class	**Spam**	45	3
	Non-Spam	5	47

Confusion Matrix

Precision = 45/48 = 93.75%   Recall = 45/50 = 90%

F1-Score = 91.84%

COVID-19 Detector
Infected = 20, Health = 2000

Actual Class

		Infected	***Healthy***
Predicted Class	***Infected***	1	2
	Healthy	19	1998

Confusion Matrix

Precision = 1/3 = 33.33%   Recall = 1/20 = 5%

F1-Score = 8.68%

Classification Metric: Fβ-Score

- The **F1-Score** measure is obtained by taking **the harmonic mean of Precision and Recall**, namely the reciprocal of the average of the reciprocal of recall:

$$\text{F1-Score} = \frac{1}{\frac{1}{2}\frac{1}{\text{Precision}}+\frac{1}{2}\frac{1}{\text{Rcall}}} = 2\frac{(\text{Precision} \times\text{Recall})}{(\text{Precision}+\text{Recall})}$$

- Instead of giving precision and Recall equal weights that sums up to 1, we can instead assign that still sum to 1 but **weight on recall** is β times as large as the weight on precision.

$$\text{F}\beta\text{-Score} = \frac{1}{\frac{1}{\beta+1}\frac{1}{\text{Precision}}+\frac{\beta}{\beta+1}\frac{1}{\text{Rcall}}} = (1+\beta)\frac{(\text{Precision} \times \text{Recall})}{(\beta\cdot\text{Precision} + \text{Recall})}$$

- Commonly used β values are:
  - $\beta = 0.5$, weighs Recall lower than Precision.
  - $\beta = 1$, weights Recall equal to Precision.
  - $\beta = 2$, weights Recall higher than Precision.