Multi-Layer Perceptron: History and Foundations

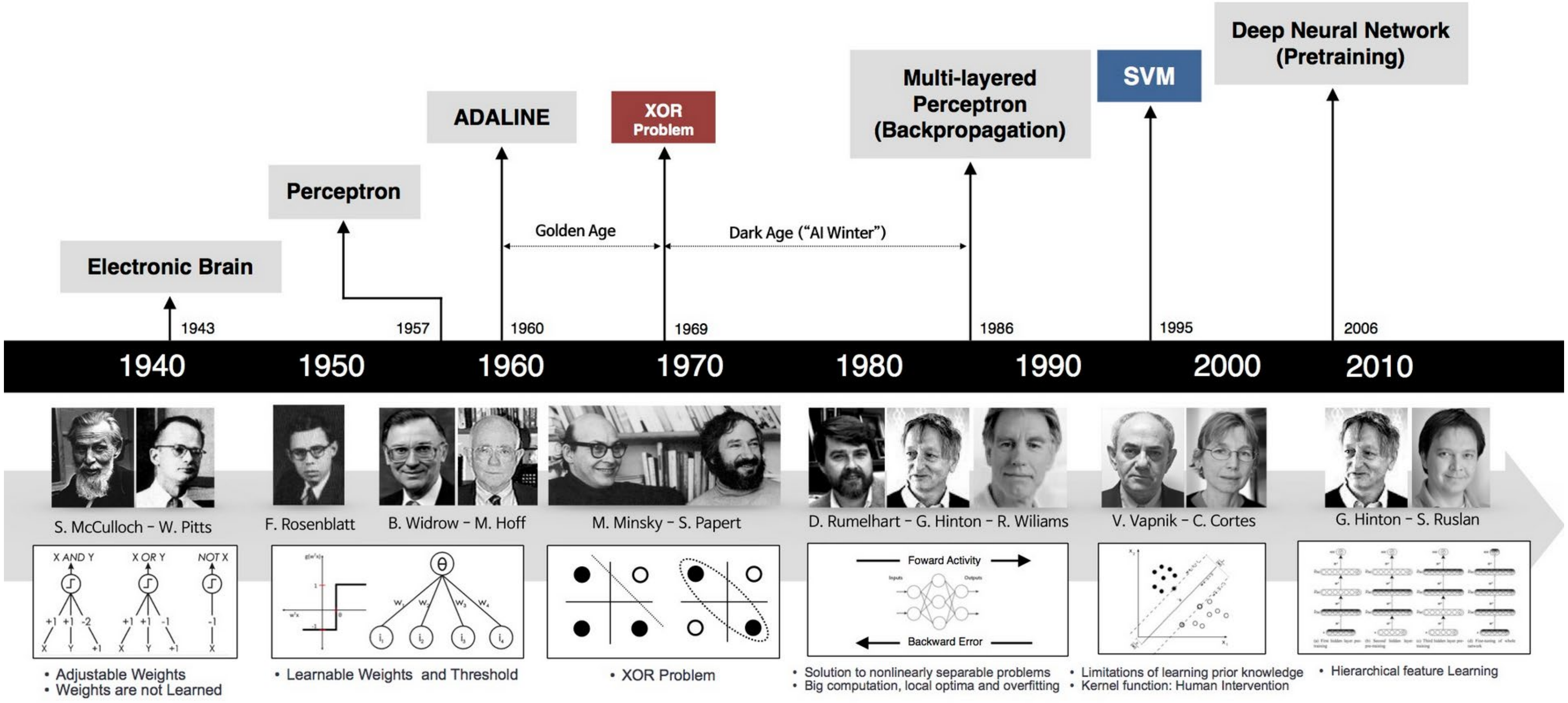
Paper presentation session
Principles of Machine Learning course

Instructor: SA Emami
Presented by: M Narimani

February 22, 2025

Part I: History

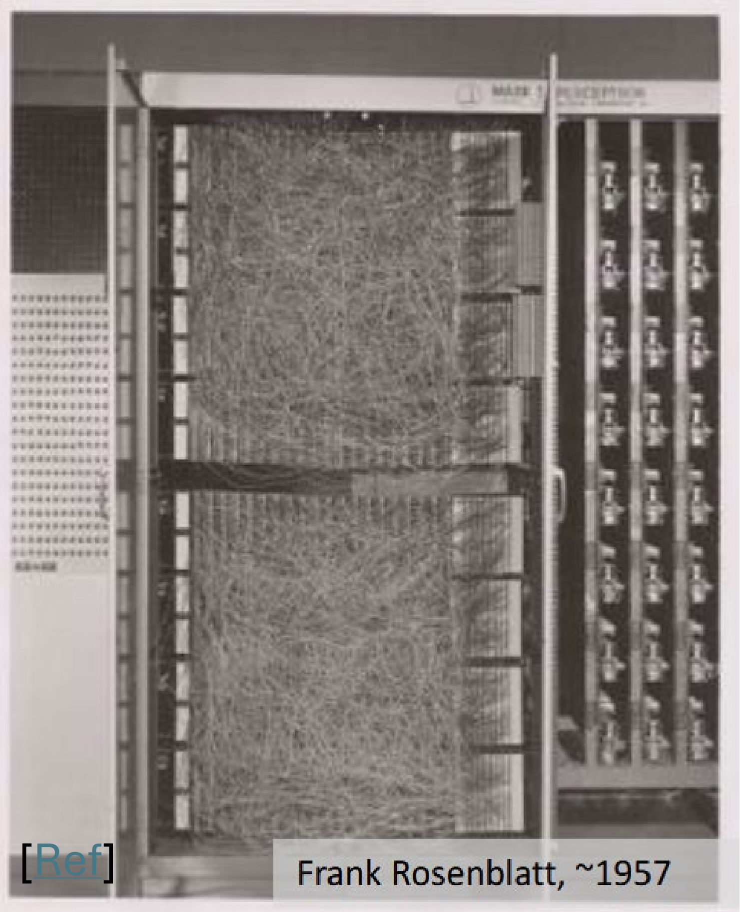
History of Neural Networks



[Image credit]

The Perceptron (1958)

- Frank Rosenblatt introduced the Perceptron
- First learning algorithm for supervised learning
- Could only solve linearly separable problems
- Rosenblatt envisioned multilayer networks but couldn't train them



[Ref]

Frank Rosenblatt, ~1957

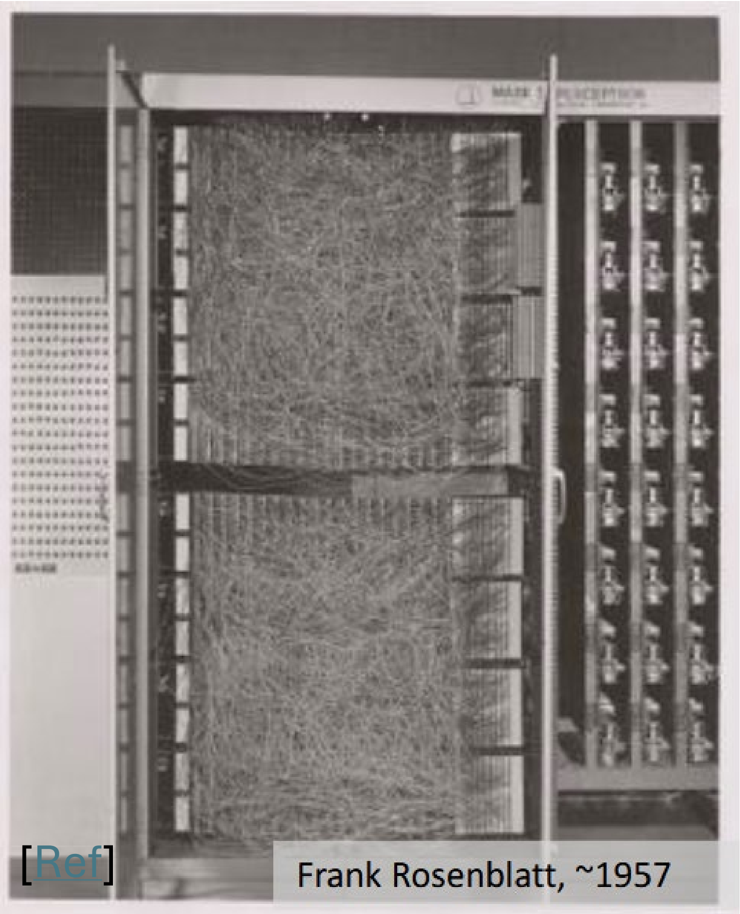
[Ref]

The Perceptron (1958)

- Learning Rule:
  - a procedure for modifying the weights and biases of a network
- Supervised Learning:
  - the learning rule is provided with a set of examples

Training Set $\{\mathbf{p}_1, \boxed{\mathbf{t}_1}\}, \{\mathbf{p}_2, \boxed{\mathbf{t}_2}\}, \dots, \{\mathbf{p}_Q, \boxed{\mathbf{t}_Q}\},$

Target



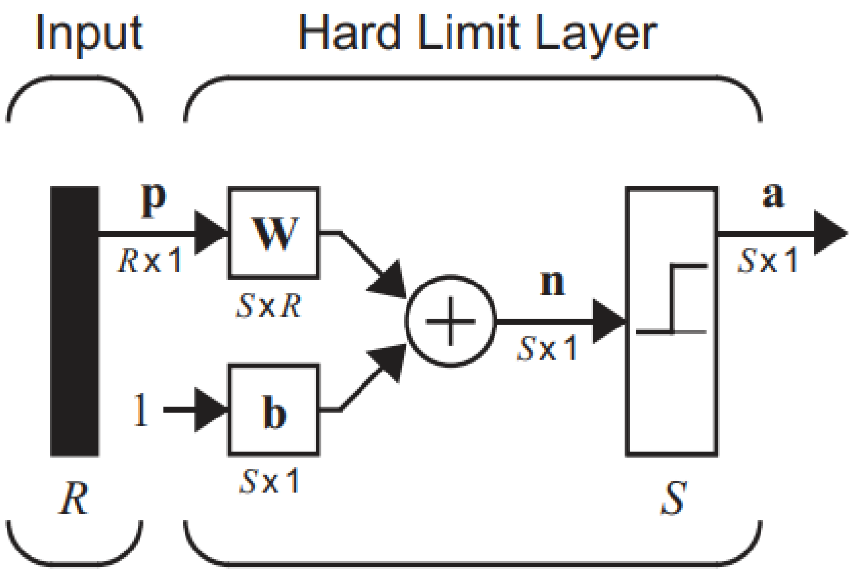
Frank Rosenblatt, ~1957

The Perceptron (1958)

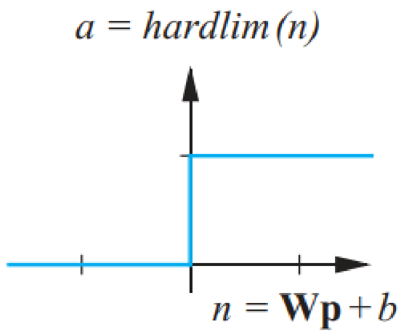
- Perceptron Architecture:

$$\mathbf{a} = \mathbf{hardlim}(\mathbf{W}\mathbf{p} + \mathbf{b}).$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}.$$



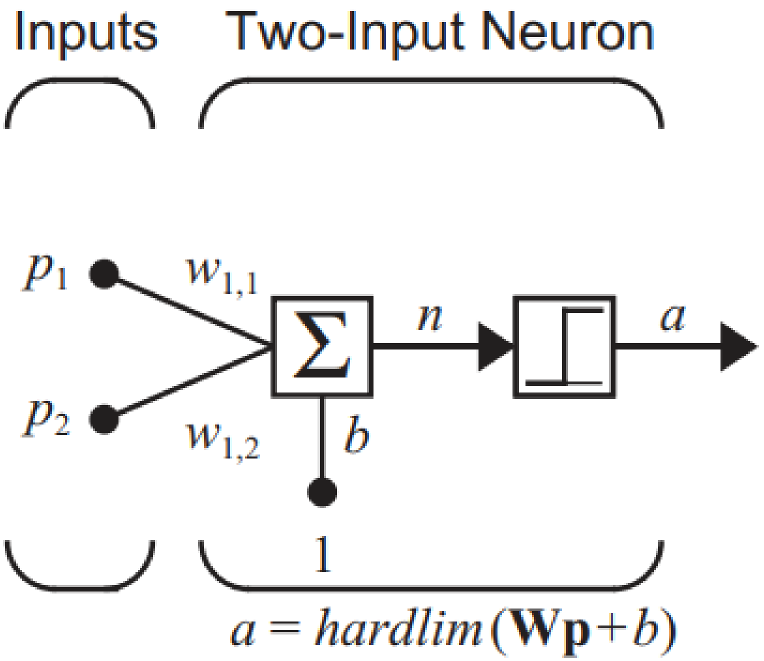
$\mathbf{a} = \mathbf{hardlim}(\mathbf{W}\mathbf{p}+\mathbf{b})$



[Ref]

The Perceptron (1958)

- Single-Neuron Perceptron:



$$a = hardlim(n) = hardlim(\mathbf{W}\mathbf{p} + b)$$

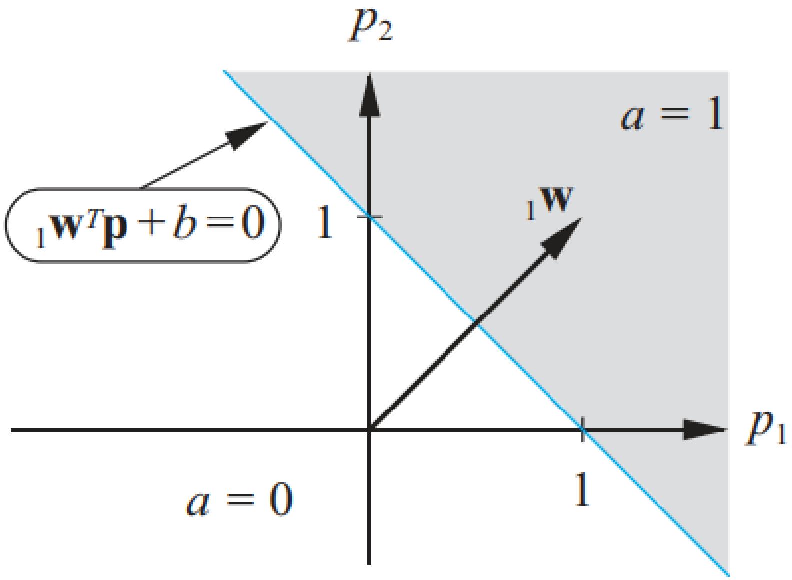
$$= hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b)$$

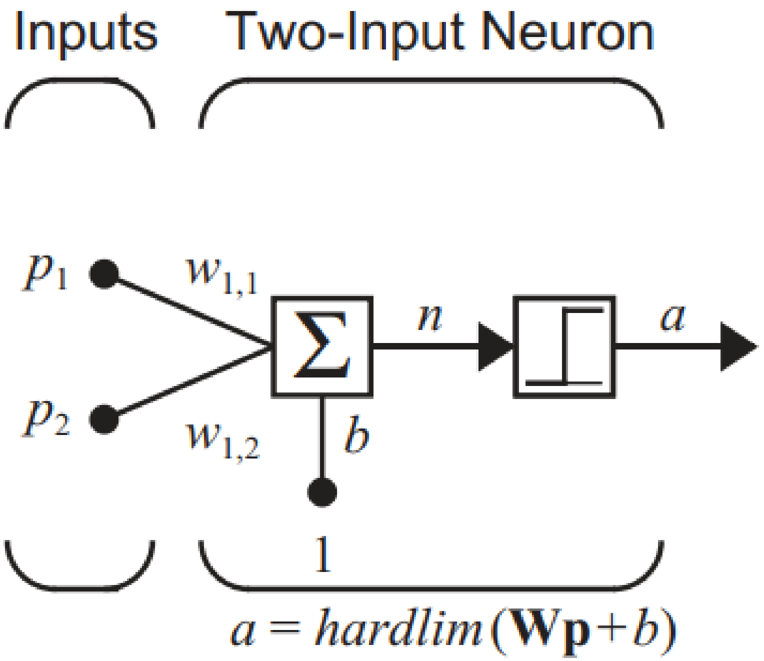
[Ref]

The Perceptron (1958)

- Single-Neuron Perceptron:
  - Decision Boundary:
    - NN input is zero
    - e.g., $w_{1,1} = 1, w_{1,2} = 1, b = -1.$

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = p_1 + p_2 - 1 = 0.$$

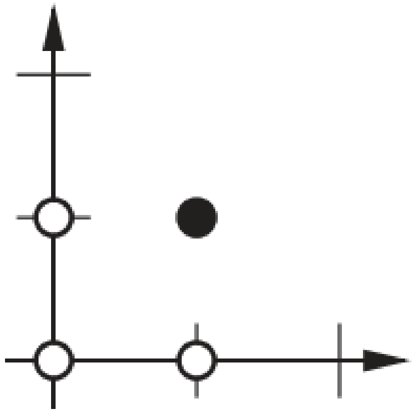


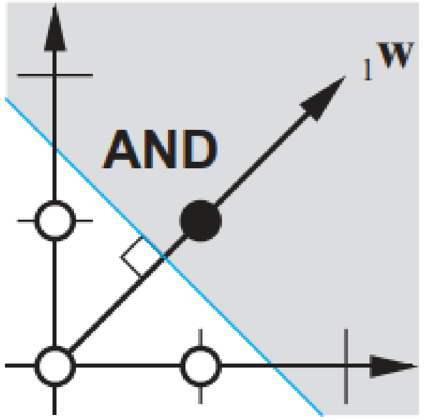


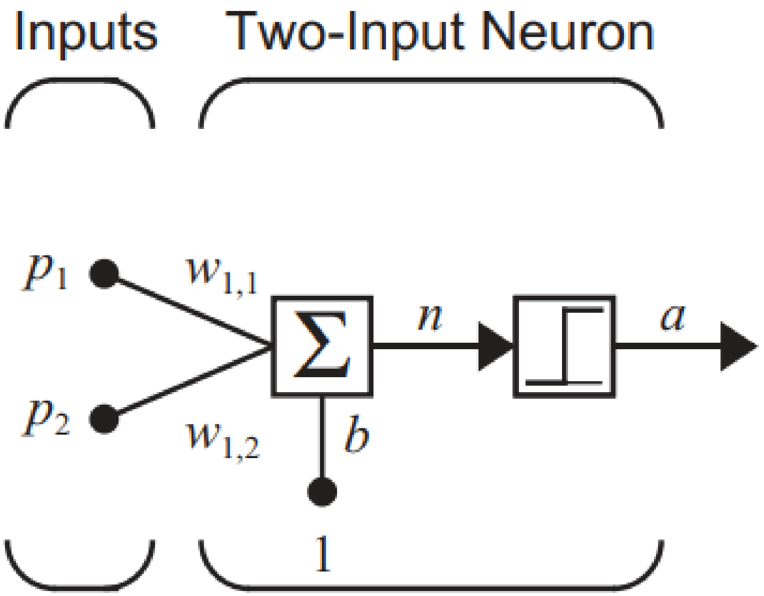
[Ref]

The Perceptron (1958)

- Single-Neuron Perceptron:
  - AND gate:

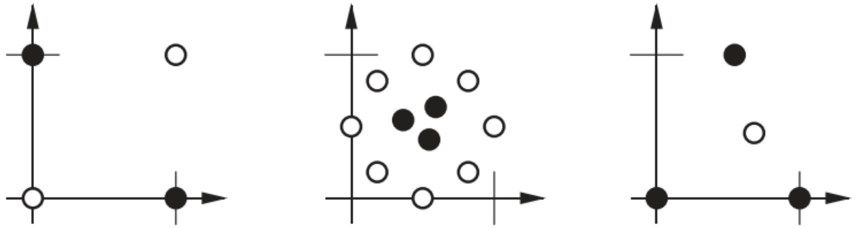






$$a = hardlim(\mathbf{W}\mathbf{p} + b)$$

  - Limitation: **Linear Separability**



[Ref]

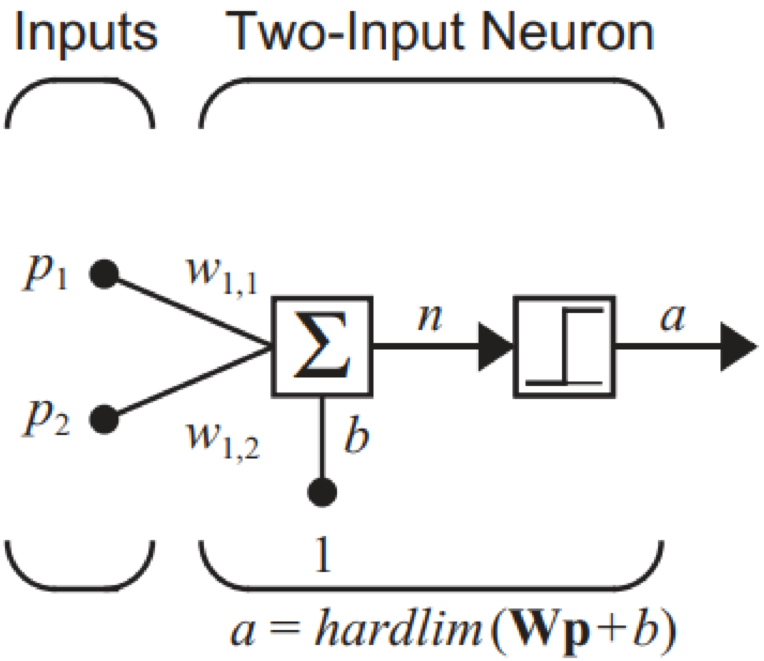
The Perceptron (1958)

- Perceptron Learning Rule:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

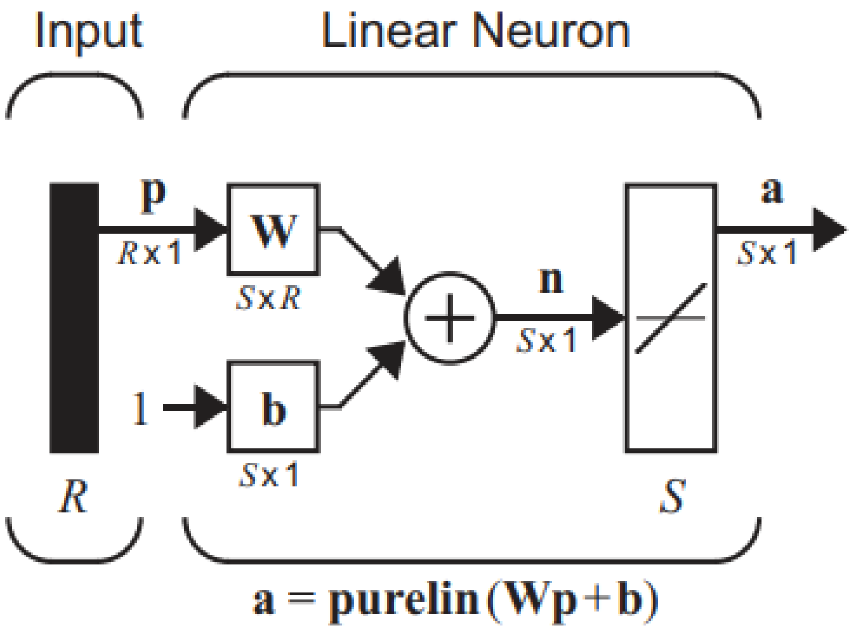
$$\text{where } \mathbf{e} = \mathbf{t} - \mathbf{a}\,.$$



[Ref]

Widrow-Hoff Learning (1960)

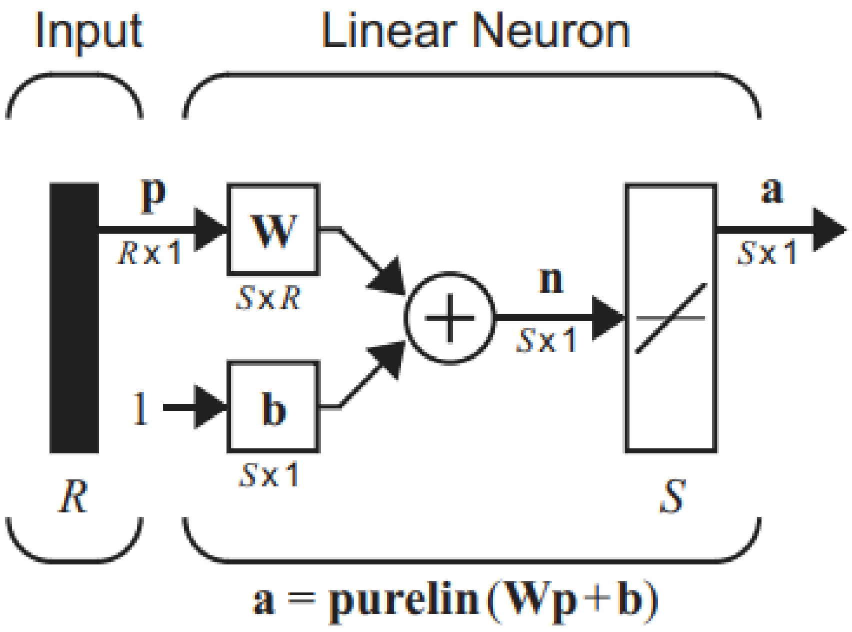
- Known as **ADALINE**
  - ADAptive LInear Neuron
- An approximate steepest descent algorithm



Widrow-Hoff - 1960

Widrow-Hoff Learning (1960)

- LMS Algorithm:
  - (Least Mean Square)
  - Supervised learning
  - Adjusts the weights and biases of the ADALINE
  - Minimizes mean square error
  - **MSE**: Difference between the target output and the network output
  - Algorithm:

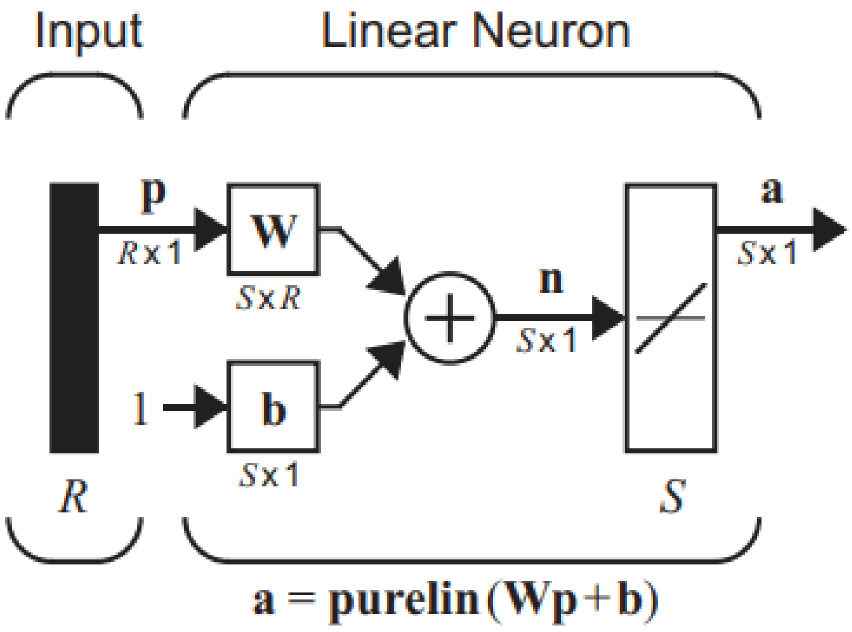


$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k),$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k).$$

Widrow-Hoff Learning (1960)

- **Discussion:**
  - More robust decision boundaries than Perceptron
  - Limited to linear separation (like the Perceptron)
  - Key difference: Uses continuous activation function

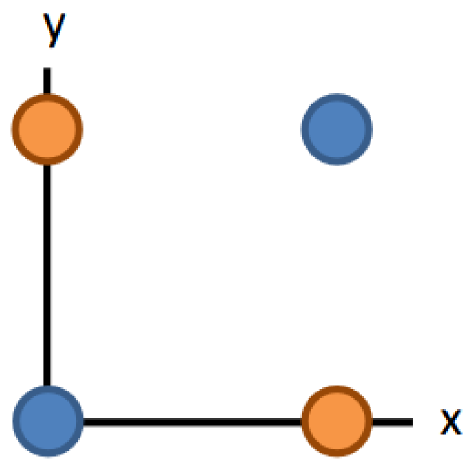


Widrow-Hoff - 1960

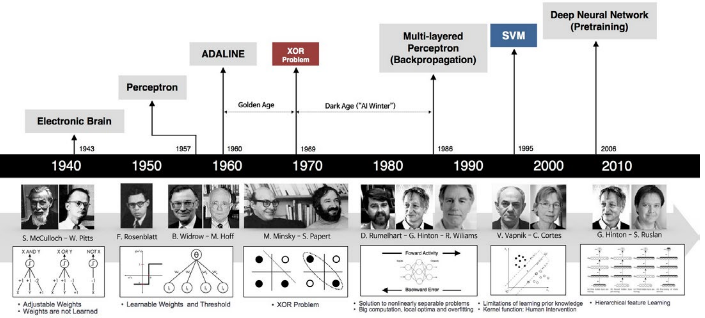
The XOR Problem (1969)

- Minsky and Papert's critique (1969)
- Showed that Perceptrons could not learn the XOR Function

X	Y	F(x,y)
0	0	0
0	1	1
1	0	1
1	1	0



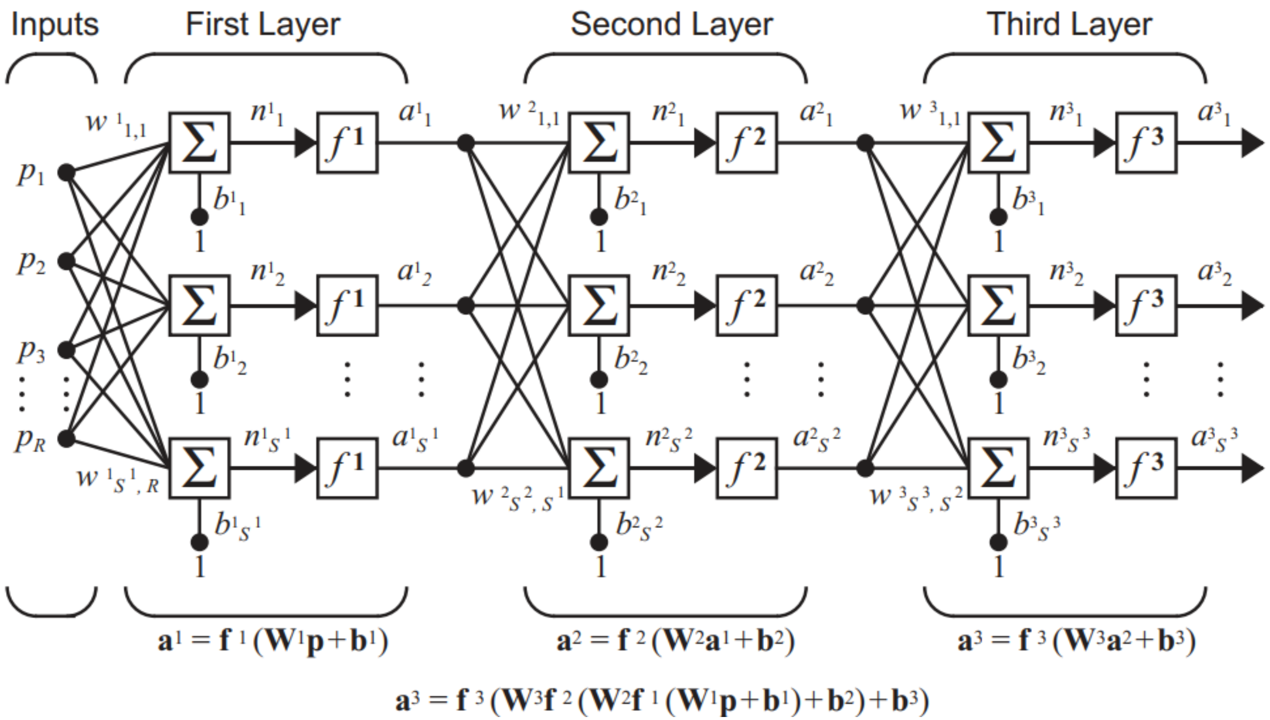
XOR: simplest non-linearly separable problem



Part II: MLP

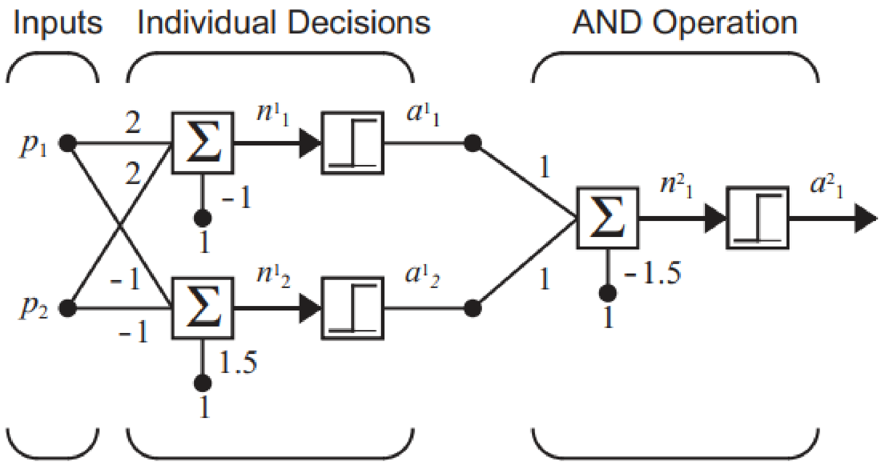
MLP Structure

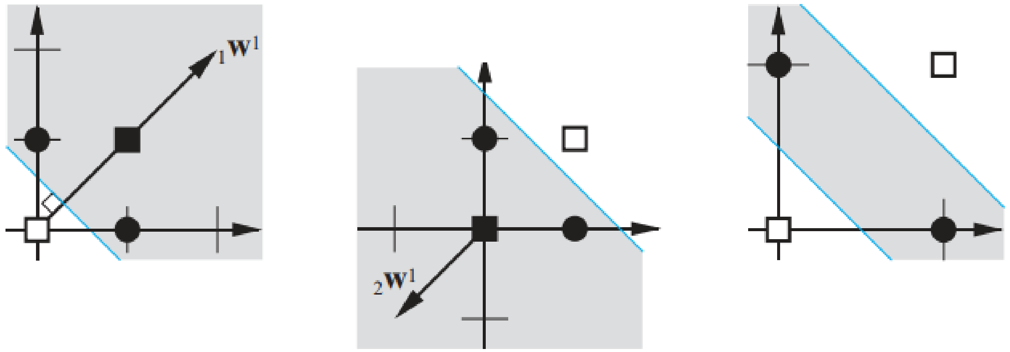
- Multiple layers of neurons
- Non-linear activation functions
- Universal function approximator
- Can solve XOR and other non-linear problems
- **Key challenge**: How to train multiple layers?



Pattern Classification with MLPs

- Can create complex decision boundaries
- Hidden layers learn feature representations
- Output layer performs final classification
- Capable of both binary and multi-class classification





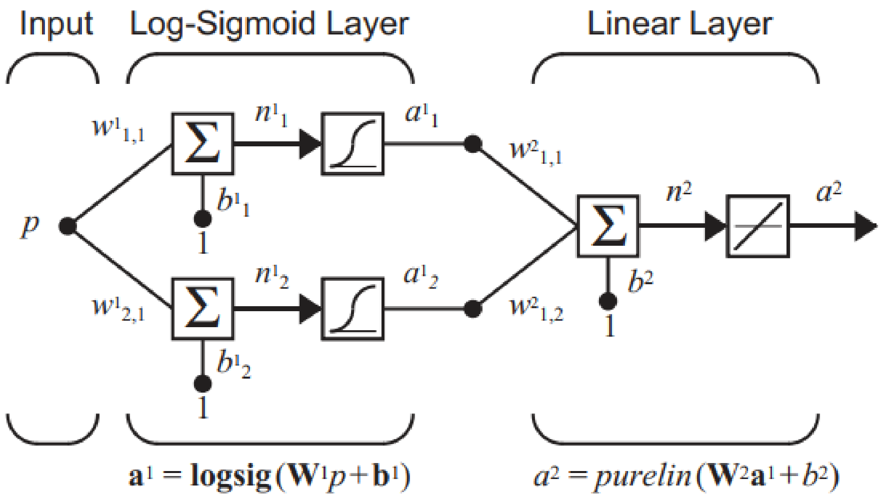
Layer 1/Neuron 1

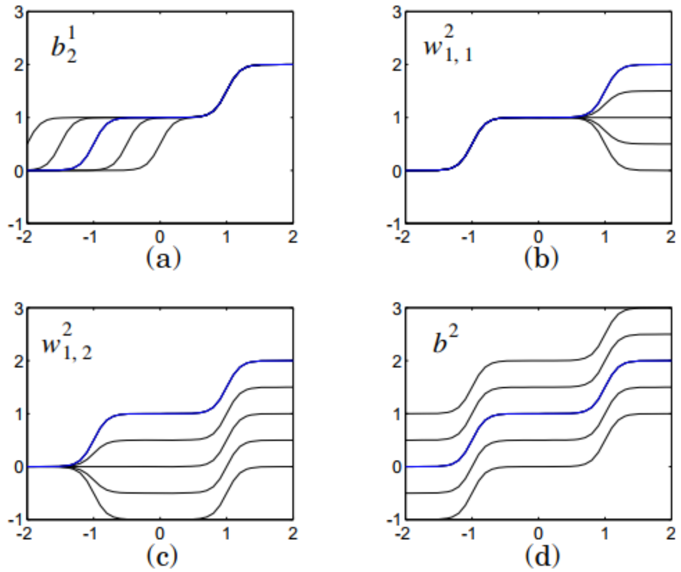
Layer 1/Neuron 2

Function Approximation with MLP

$$f^1(n) = \frac{1}{1+e^{-n}} \text{ and } f^2(n) = n\,.$$

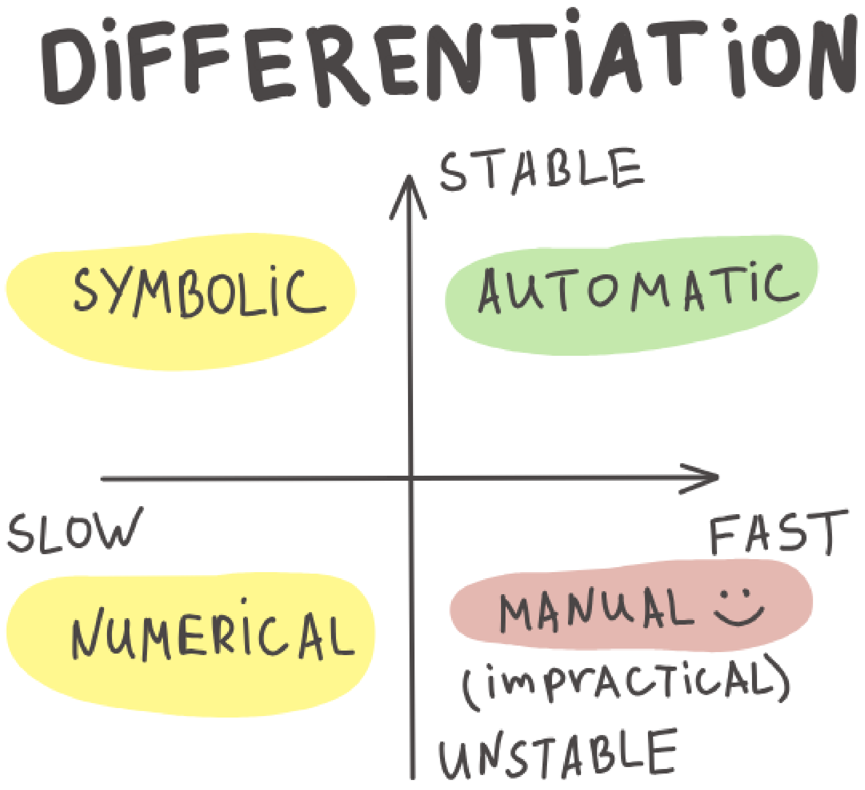
- Universal approximation theorem
- Can approximate any continuous function
- Trade-off between network size and accuracy
- Applications in regression problems





The Learning Problem

- How to compute gradients for multiple layers?
- Manual derivation is impractical
- Need efficient algorithm for gradient computation
- **Solution**: Reverse-mode automatic differentiation



[Ref]

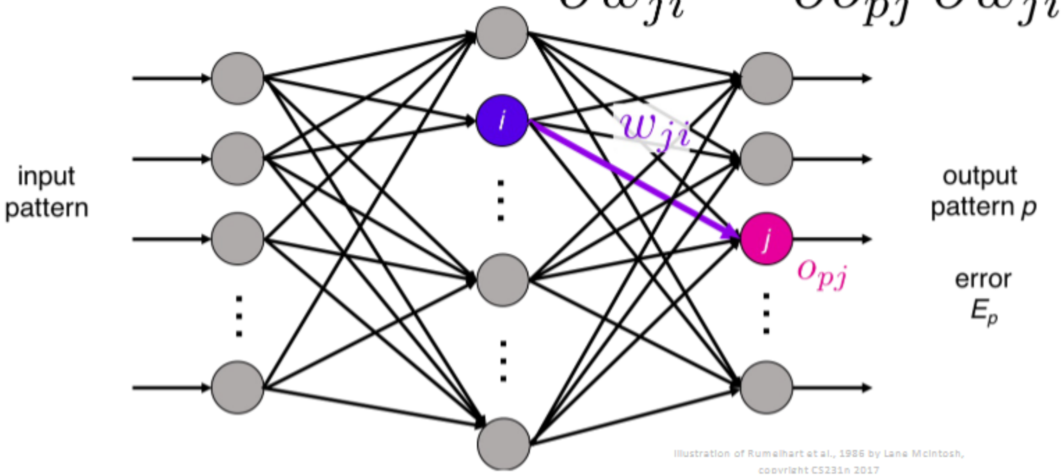
Part III: Backpropagation

History of Backpropagation

- **1970**: Linnainmaa introduces reverse-mode autodiff
- **1974**: Werbos describes training multilayer networks
- **1985**: Rumelhart, Hinton, Williams popularize backpropagation
  - Showed how networks learn internal representations

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}}$$

recognizable math



LMS vs Backpropagation

- **LMS**
  - Used for single-layer linear networks
  - Error is an explicit linear function of weights
  - Derivatives wrt weights are easily computed

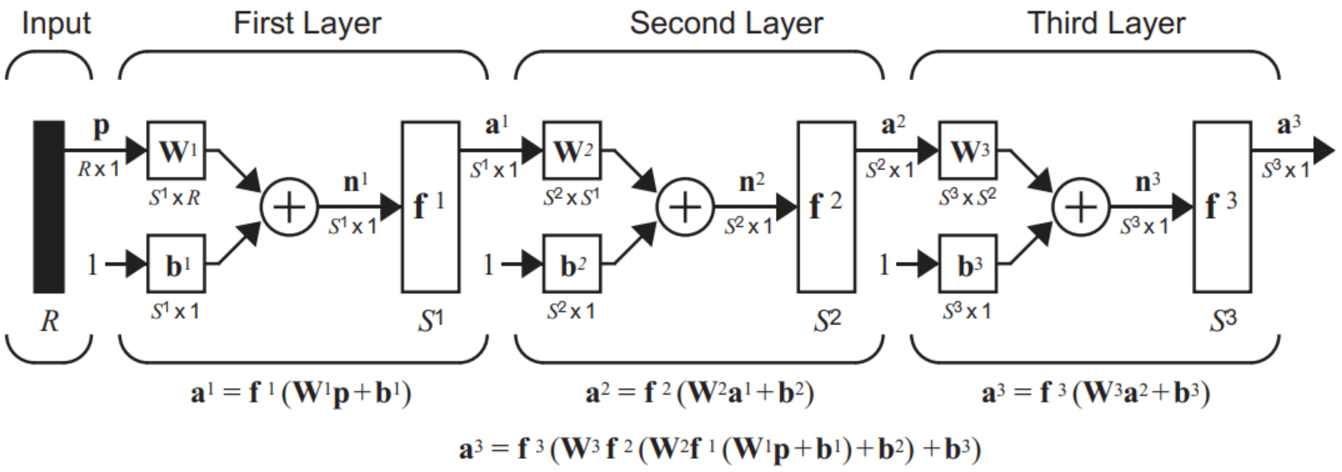
- **Backprop**
  - Generalization of the LMS algorithm
  - Used for training multilayer networks with nonlinear transfer functions
  - Error function is more complex; requires ***chain rule*** of calculus for derivatives.

Key Difference:

LMS: Linear relationship between error and weights.

Backprop: Nonlinear relationship requiring advanced calculus.

Backpropagation Algorithm



- Performance Index

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})]$$

- Weight update

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T,$$

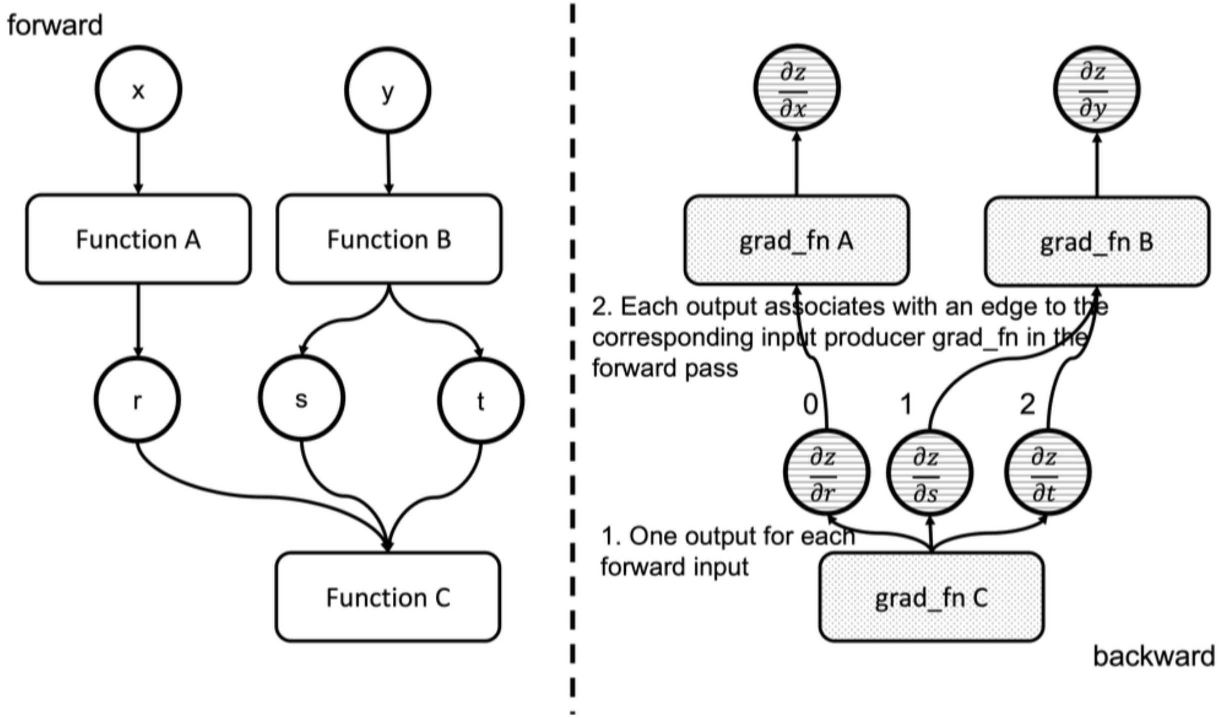
$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m.$$

- Where:

Sensitivity: $$\mathbf{s}^m \equiv \frac{\partial\hat{F}}{\partial\mathbf{n}^m} = \begin{bmatrix} \frac{\partial\hat{F}}{\partial n_1^m} \\ \frac{\partial\hat{F}}{\partial n_2^m} \\ \vdots \\ \frac{\partial\hat{F}}{\partial n_{S^m}^m} \end{bmatrix}$$

Computational Graphs

- Represent computation as directed graph
  - Nodes: Operations
  - Edges: Flow of data
  - Forward pass: Compute output
  - Backward pass: Compute gradients

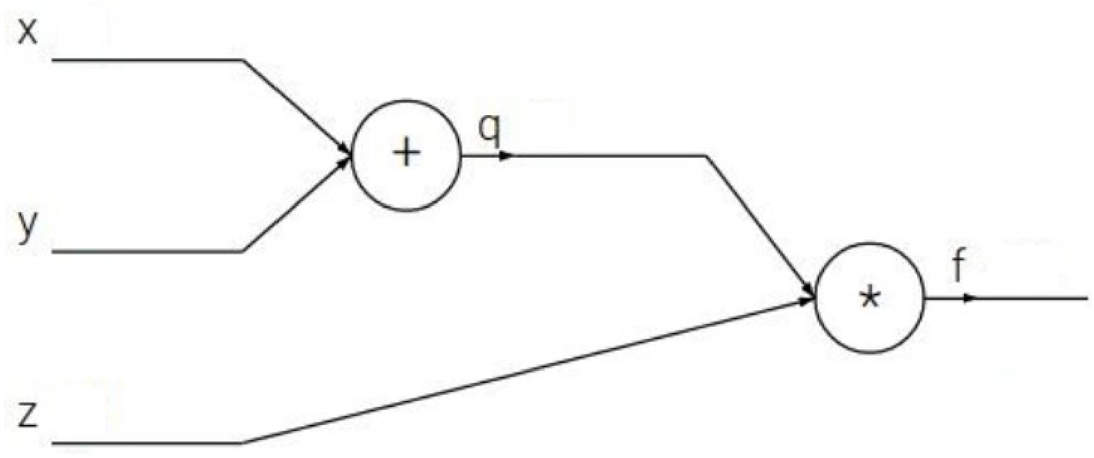


[Ref] PyTorch

Backpropagation Example

- Consider a simple network:

$$f(x, y, z) = (x + y)z$$



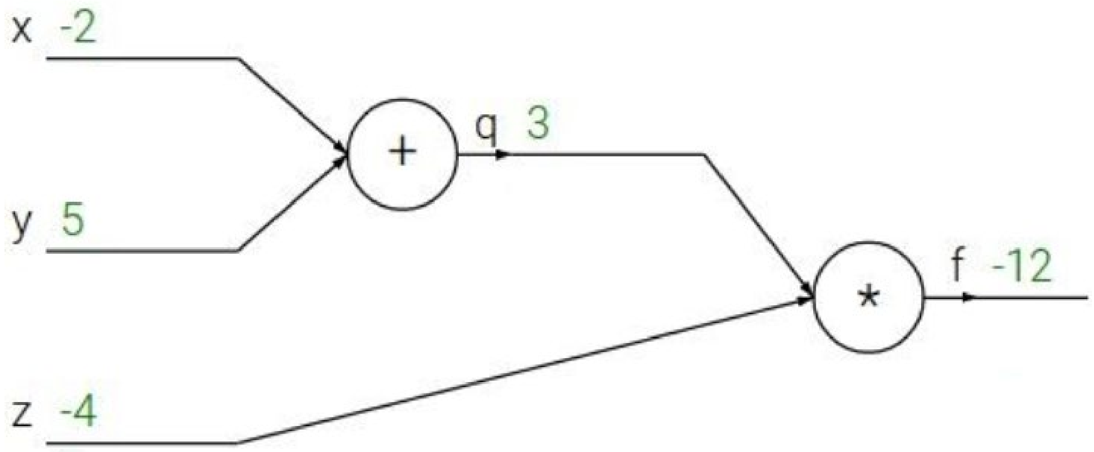
[Reference]

Backpropagation Example

- Consider a simple network:

$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4



- Forward pass:

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

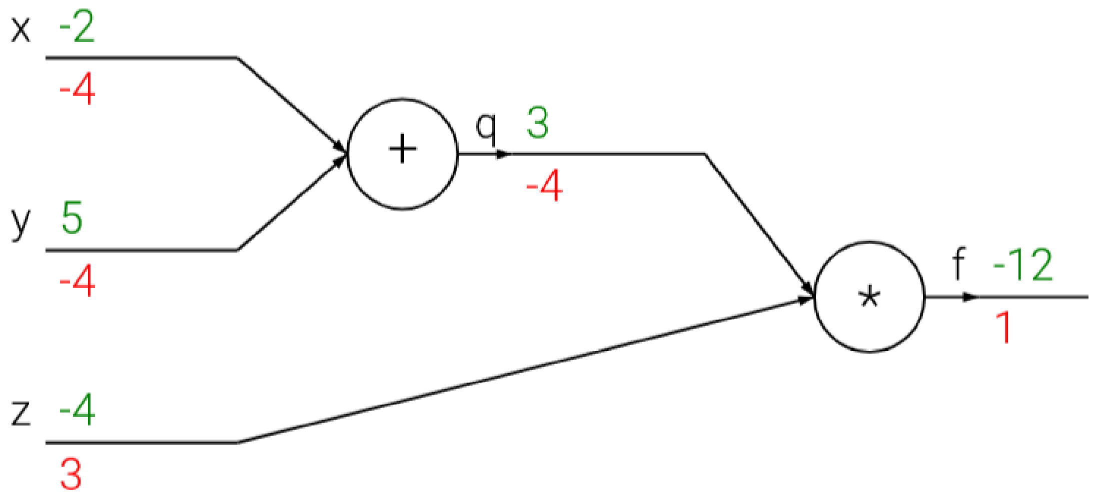
$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Backpropagation Example

- Consider a simple network:

$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4



- Forward pass:

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

- Backward pass:

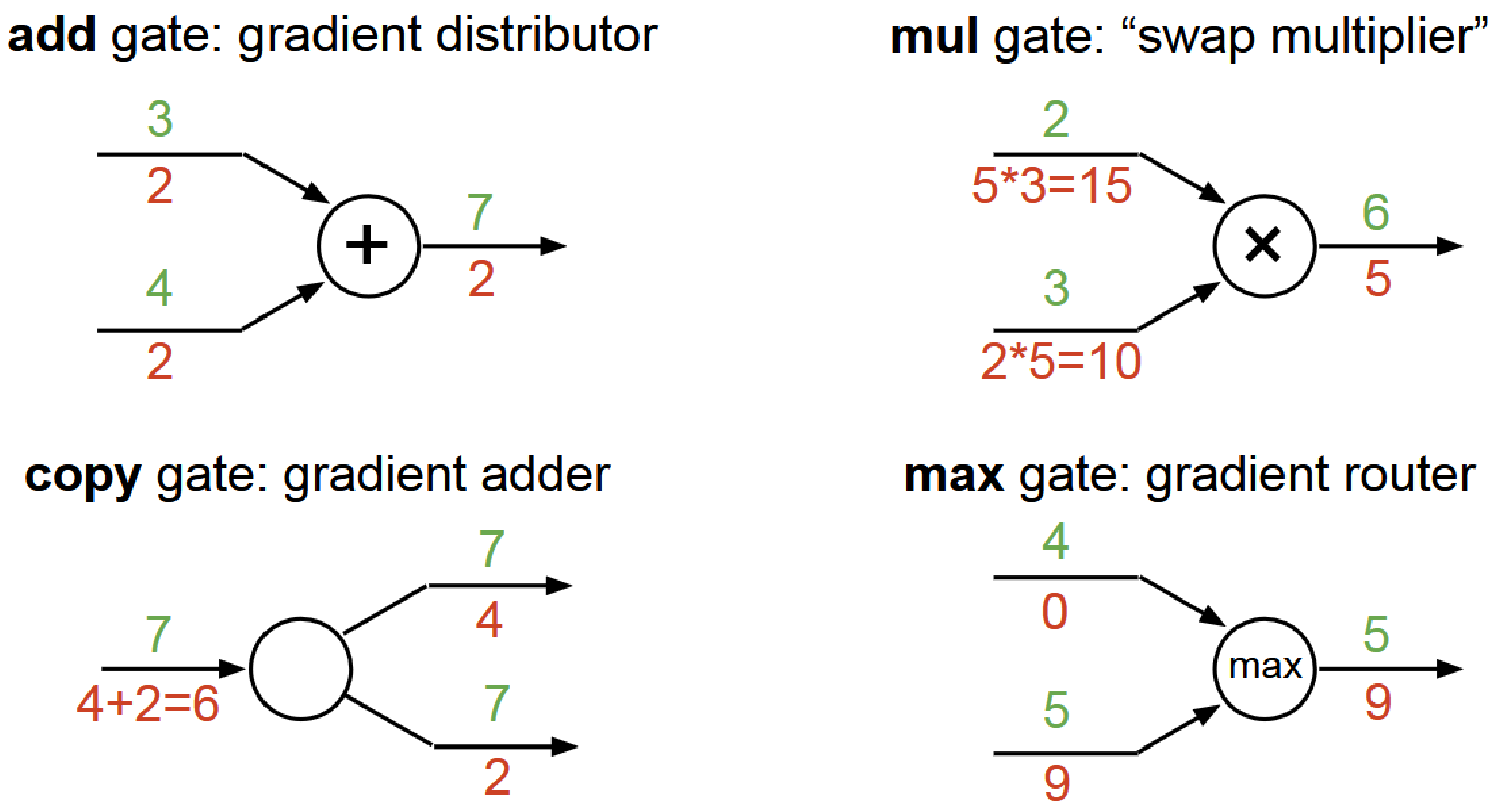
Chain rule:

$$\boxed{\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}}$$

Upstream gradient

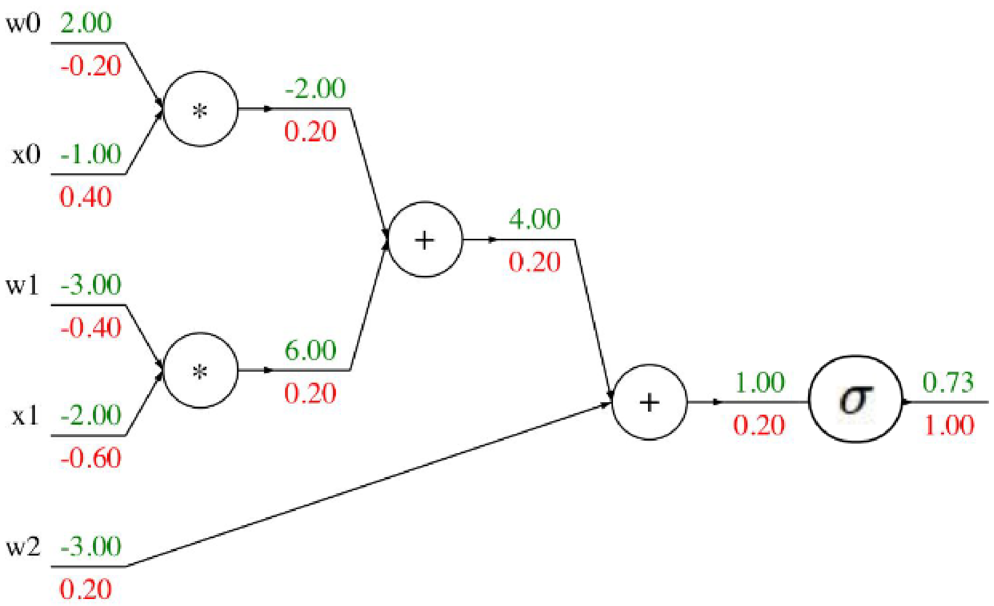
Local gradient

Patterns in gradient flow



[Reference]

Backprop Implementation



Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

Backward pass:
Compute grads

```
    grad_L = 1.0
    grad_s3 = grad_L * (1 - L) * L
    grad_w2 = grad_s3
    grad_s2 = grad_s3
    grad_s0 = grad_s2
    grad_s1 = grad_s2
    grad_w1 = grad_s1 * x1
    grad_x1 = grad_s1 * w1
    grad_w0 = grad_s0 * x0
    grad_x0 = grad_s0 * w0
```

[Reference]