# INTRODUCTION TO DEEP LEARNING & APPLICATIONS

Pablo Martínez Olmos, **pamartin@ing.uc3m.es**

## PART I

Neural Networks & Multilayer Perceptrons
BackPropagation
Regularization Techniques

## Overview

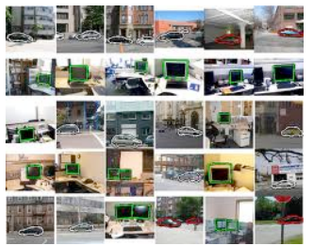
- 4 Sessions ( 1 Lecture + 1 Lab session):
  - **Part I: MLPs, BackPropagation, Regularization**
  - Part II: Convolutional Neural Networks for image processing
  - Part III: Sequential learning. Recurrent Neural Networks and 1-D convolutions.
  - Part IV: Natural Language Processing. Unsupervised Deep Learning
- Lab sessions:
  - Pytorch
  - Facebook DL Course in Udacity
  - 2 Homeworks

# Deep Learning Today

# Minig for Structure

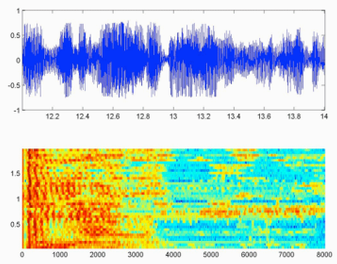
**Massive increase in both computational power and the amount of data** available from web, video cameras, laboratory measurements …

Images & Video

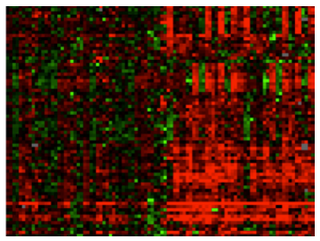


Text & Language

Speech & Audio

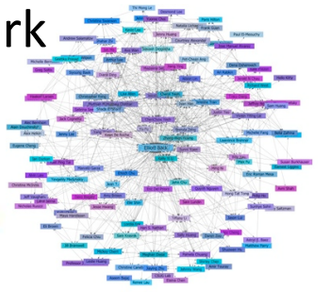


Gene Expression

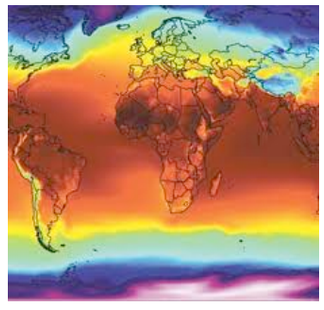


Product Recommendation

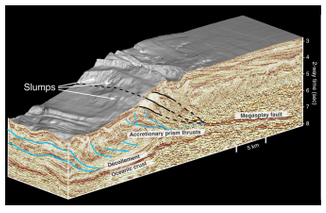
Relational Data/ Social Network



Climate Change



Geological Data



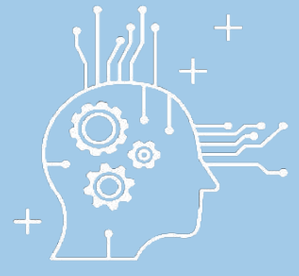

Develop **statistical models** that can discover **underlying structure**, semantic, relations, constraints, or invariances from data

**Robust, adaptive models** models that can deal with **missing measurements, nonstationary distributions, multimodal data** …

# What is Deep Learning?

## Artificial Intelligence

Any technique that enables computers to mimic human behavior

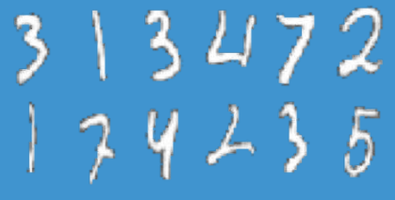


## Machine Learning

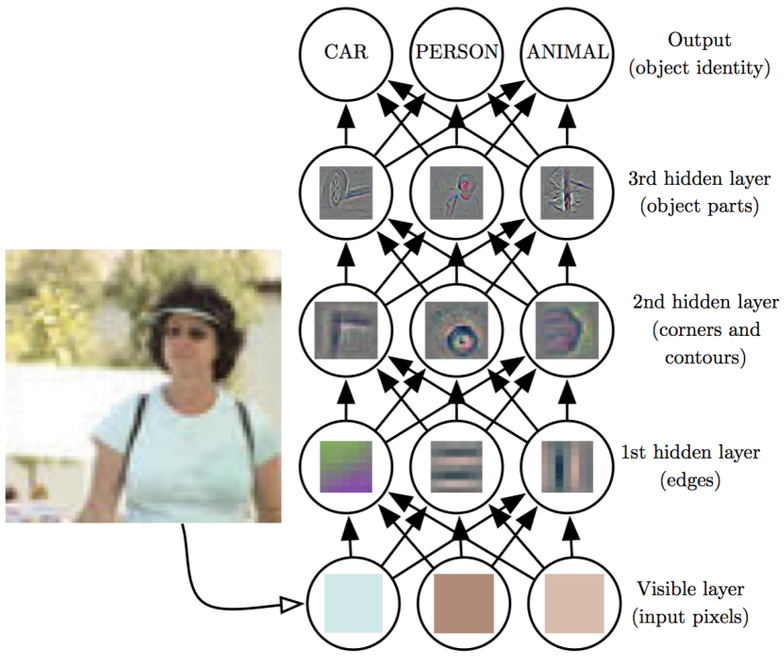
Ability to learn without explicitly being programmed

## Deep Learning

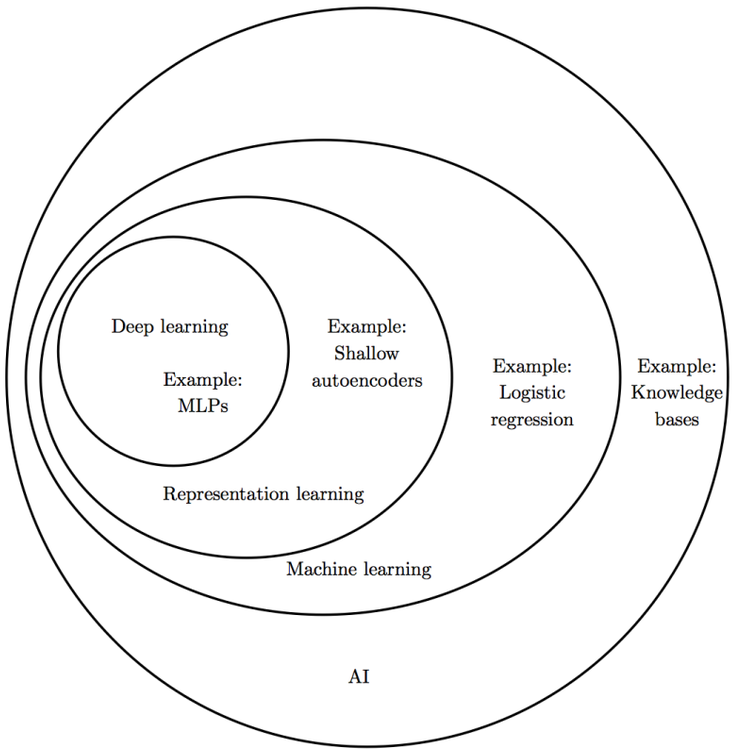
Extract patterns from data using neural networks





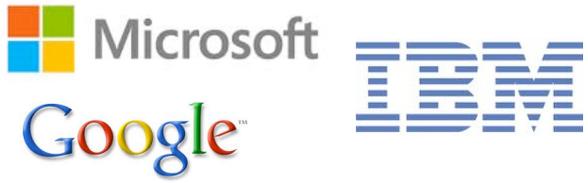



Source: Deep Learning, Good Fellow et al. 2017



# Impact of Deep Learning

- Speech Recognition
- Computer Vision
- Recommender Systems
- Language Understanding
- Autonomous systems





- Bioinformatics & Biotechnology
- Medical Imaging

**30 amazing applications of deep learning (link)**

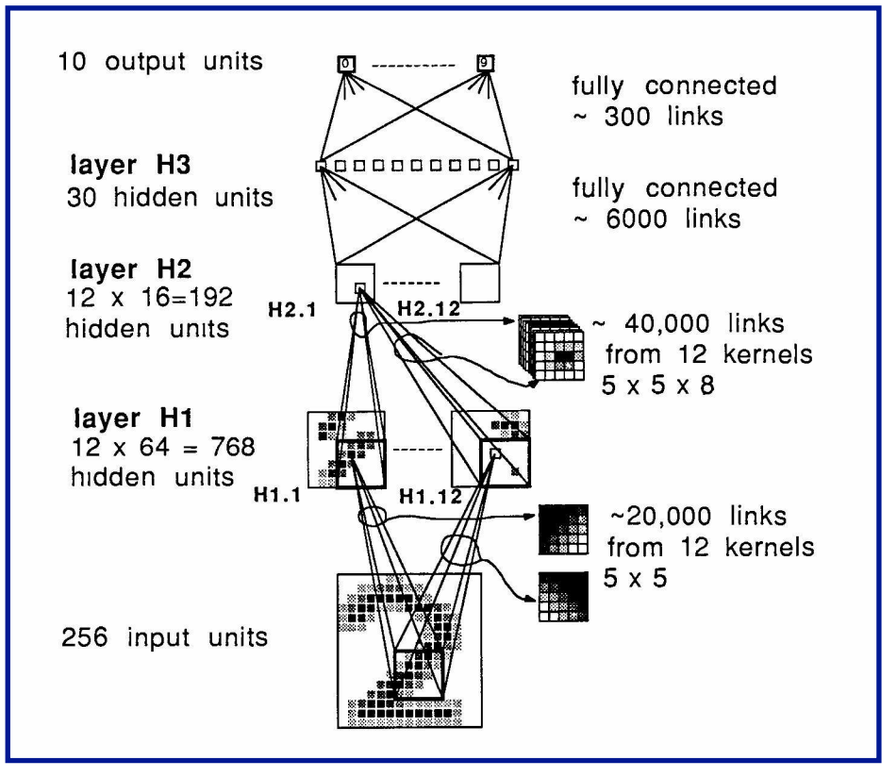
**10 Companies Using Machine Learning in Cool Ways**

**Deep Learning Weekly Newsletter**

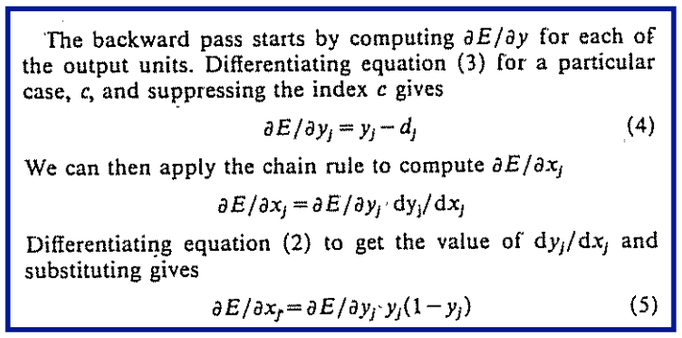
# Neural Networks Revisited

Most elements of network architecture employed as early as the mid-1980's

**A History of Deep Learning (link)**




Backpropagation applied to Handwritten Zip Code Recognition
LeCun et al. (1989)



The backward pass starts by computing $\partial E/\partial y$ for each of the output units. Differentiating equation (3) for a particular case, $c$, and suppressing the index $c$ gives

$$\partial E/\partial y_j = y_j - d_j \quad (4)$$

We can then apply the chain rule to compute $\partial E/\partial x_j$

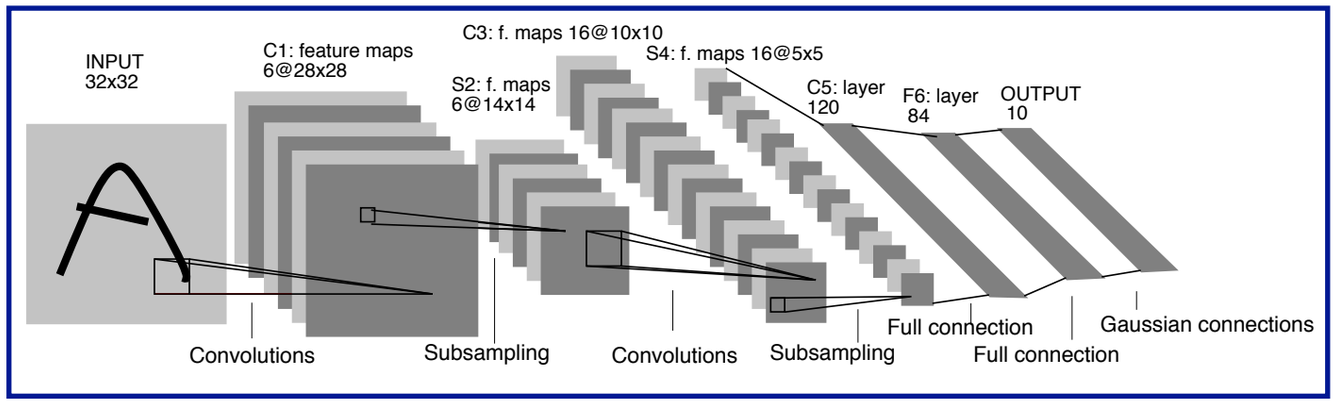
$$\partial E/\partial x_j = \partial E/\partial y_j \cdot \mathrm{d}y_j/\mathrm{d}x_j$$

Differentiating equation (2) to get the value of $\mathrm{d}y_j/\mathrm{d}x_j$ and substituting gives

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j(1-y_j) \quad (5)$$

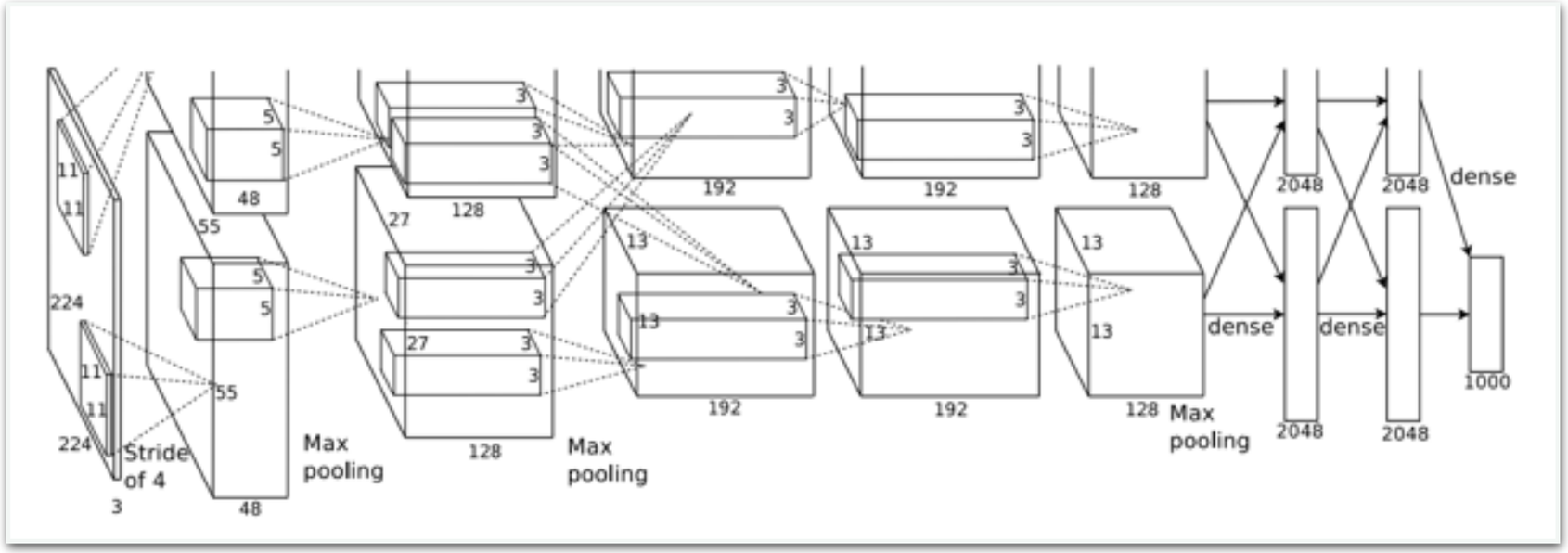
Learning representations
by back-propagating errors

Rumelhart, Hinton, Williams (1986)




Gradient-Based Learning applied to Document Recognition
LeCun, Bengio, Haffner (1998)

# Why now? Scale matters




**ImageNet Classification with Deep Convolutional Neural Networks**

A Krizhevsky I Sutskever, G Hinton (2012)

- Deep CNN (AlexNet) won 2012 ImageNet classification contest
- 60 Millions Parameters
- Achieving scale in compute and data is critical.
  - Large academic data sets. Easier collection and storage
  - SIMD hardware (e.g. GPU's). Massively Parallelization
  - Automatic Differentiation Toolboxes

IMAGENET

WIKIPEDIA
The Free Encyclopedia

TensorFlow

PyTorch

# Logistic Regression: the model

## Binary Logistic Regression

- Discriminative classifier:

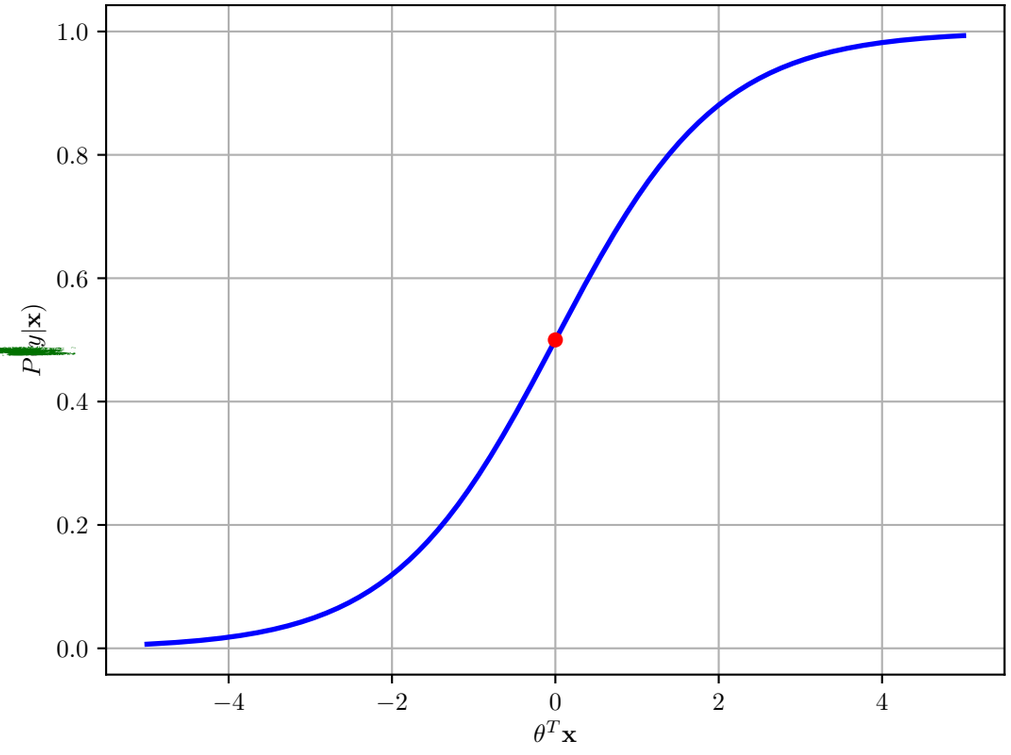
$$P(y=1|\boldsymbol{x}) = \frac{1}{1+\mathrm{e}^{-(\boldsymbol{w}^T\boldsymbol{x}+w_0)}} \doteq \sigma(\boldsymbol{w}^T\boldsymbol{x}+w_0)$$

Generalized linear model

- Decision boundary:

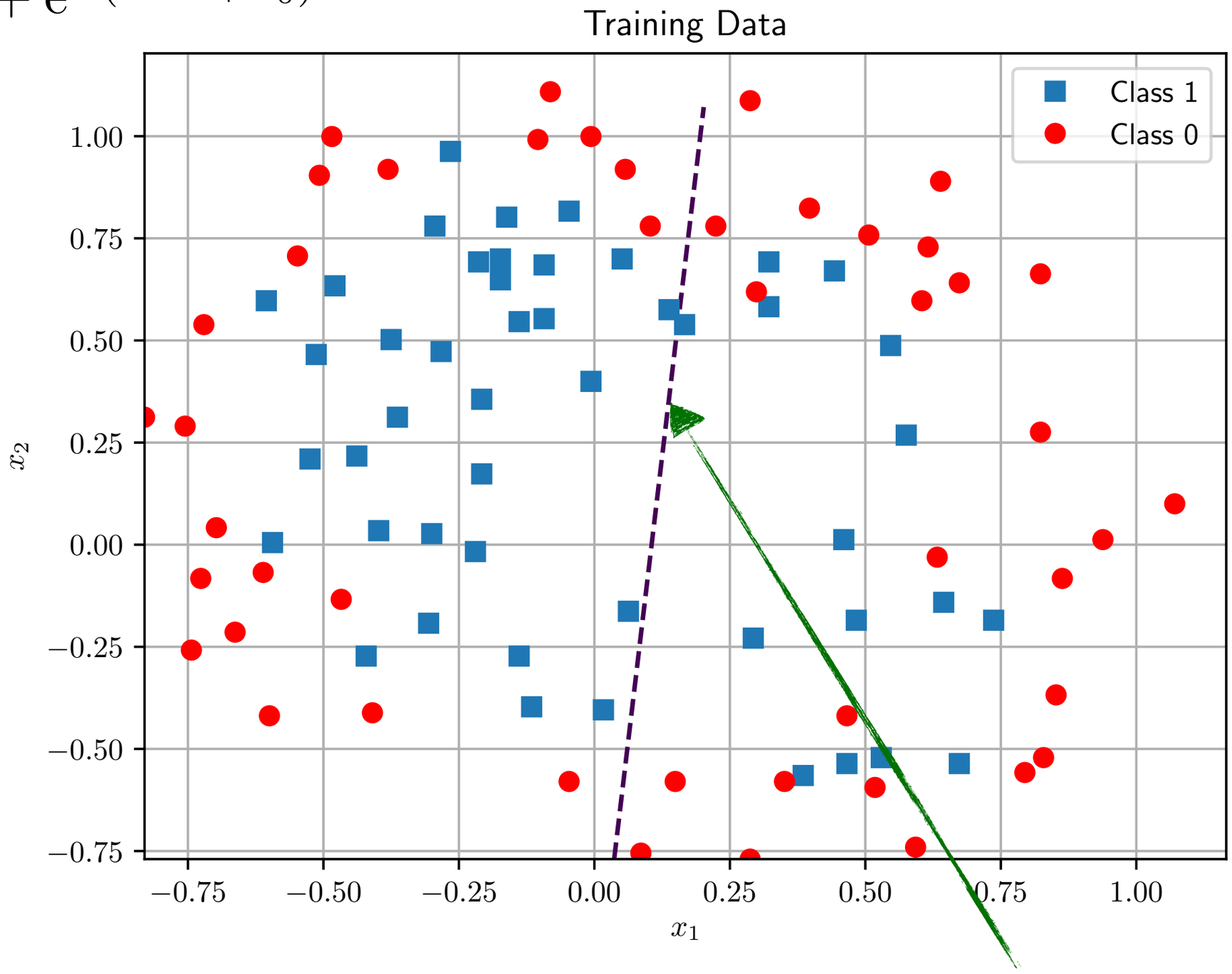
$$\{\boldsymbol{x} \in \mathbb{R}^m : \boldsymbol{w}^T\boldsymbol{x} + w_0 = \mathbf{0}\}$$

**Hyperplane! Linear classifier**



## Binary Logistic Regression

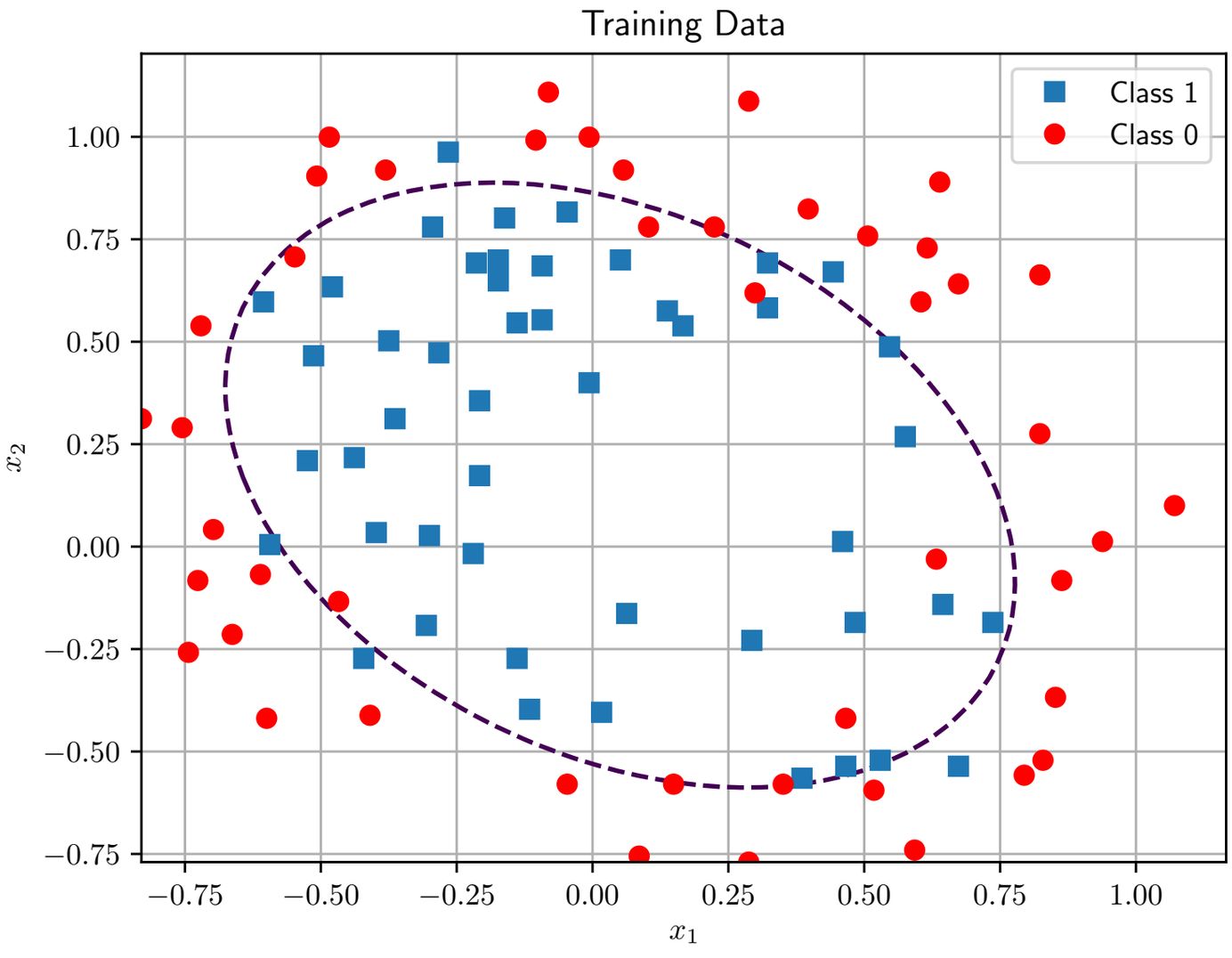
$$P(y=1|\boldsymbol{x}) = \frac{1}{1+\mathrm{e}^{-(\boldsymbol{w}^T\boldsymbol{x}+w_0)}} \doteq \sigma(\boldsymbol{w}^T\boldsymbol{x}+w_0)$$




LR decision boundary with linear features

## Binary Logistic Regression. Adding new features

- The problem is easy to visualize
- An ellipsoid clearly fits better as decision boundary than a straight line




- Expand input space with extra degrees of freedom
- Quadratic features

$$\boldsymbol{x} = [x_1,\ x_2]$$

$$\boldsymbol{\phi}(\boldsymbol{x}) = [x_1,\ x_2,\ x_1^2,\ x_2^2,\ x_1 x_2]$$

$$P(y = 1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) + w_0)$$

## Binary Logistic Regression. Binary Cross Entropy Function

$$P(y=1|\boldsymbol{x}) = \frac{1}{1+\mathrm{e}^{-(\boldsymbol{w}^T\boldsymbol{x}+w_0)}} \doteq \sigma(\boldsymbol{w}^T\boldsymbol{x}+w_0)$$

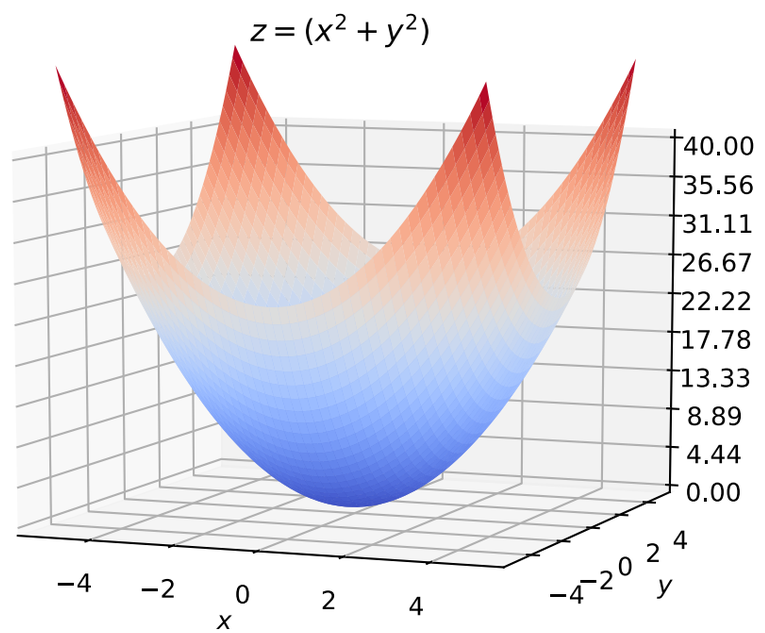
**Binary Cross Entropy Function**

$$\begin{aligned}\mathcal{L} &= -\log P(\boldsymbol{y}|\boldsymbol{X}) \\ &= -\sum_{i=1}^{N} \mathbb{1}[y^{(i)}=1]\log P(y^{(i)}=1|\boldsymbol{x}^{(i)}) + \mathbb{1}[y^{(i)}=0]\log P(y^{(i)}=0|\boldsymbol{x}^{(i)})\end{aligned}$$

**Logistic Regression Training**

$$\hat{\boldsymbol{w}}, \hat{w}_0 = \arg\min_{\boldsymbol{w}, w_0} \mathcal{L}$$

**$\mathcal{L}$ is convex!!**



## Multiclass Logistic Regression

- Training database: $\mathcal{D} \doteq (\boldsymbol{x}^{(i)}, y^{(i)})_{i=1}^{N} \qquad \boldsymbol{x}^{(i)} \in \mathbb{R}^m \quad y^{(i)} \in \{1, 2, \ldots, K\}$

- Discriminative classifier based on the **Softmax function**:

$$P(y = k|\boldsymbol{x}) = \frac{\mathrm{e}^{-(z_k)}}{\sum_{j=1}^{K} \mathrm{e}^{-(z_j)}}, \quad \boldsymbol{z} = \boldsymbol{W} \begin{bmatrix} 1.0 \\ \boldsymbol{x}^{(i)} \end{bmatrix}$$

$$K \times (m + 1)$$

- Loss function (Cross entropy):

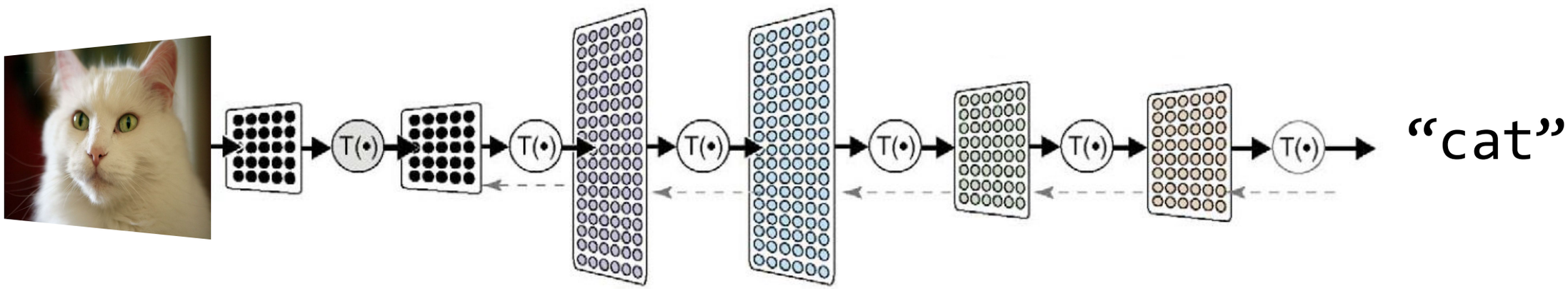
$$\mathcal{L} = -\log P(\boldsymbol{y}|\boldsymbol{X}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} \mathbb{1}[y^{(n)} == k] \log P(y = k|\boldsymbol{x})$$

- Similar gradient expression

# Fundaments of Neural Networks

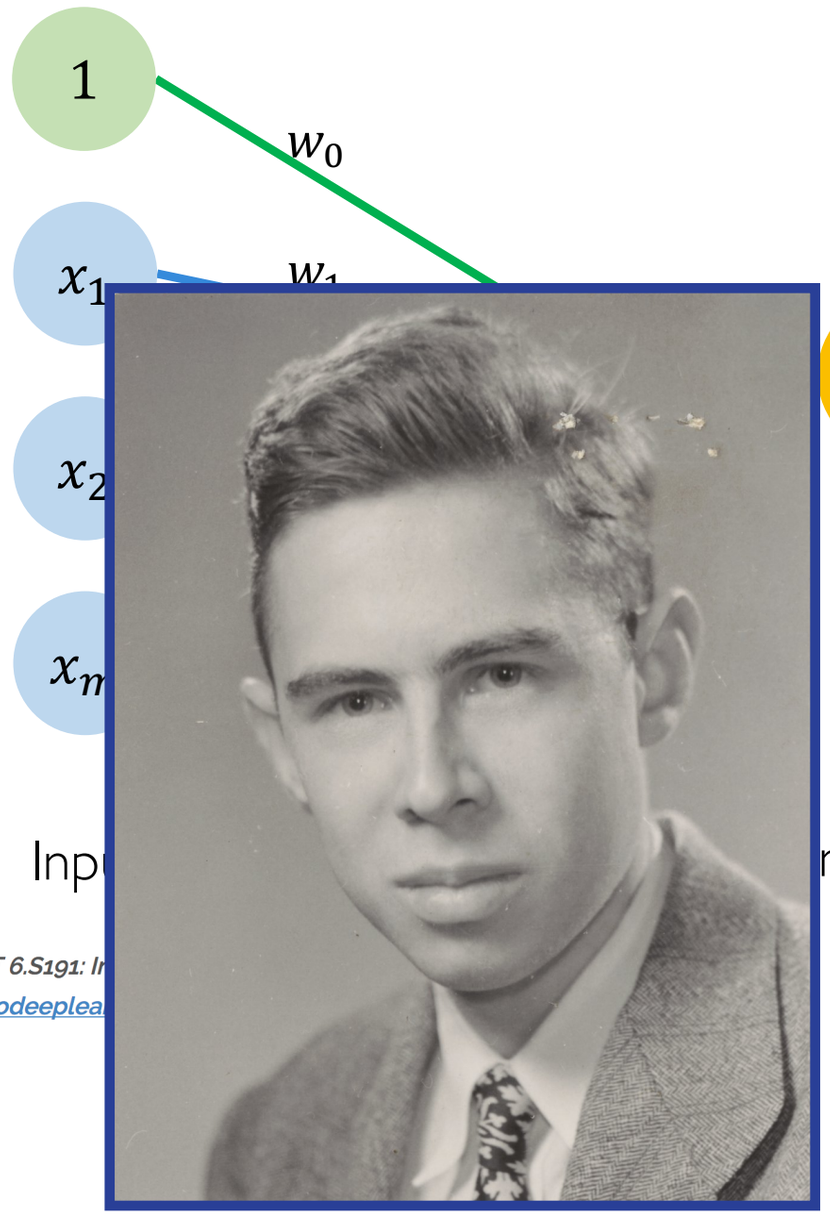
# Deep learning = artificial neural networks

**Hierarchical** composition of **simple mathematical** functions

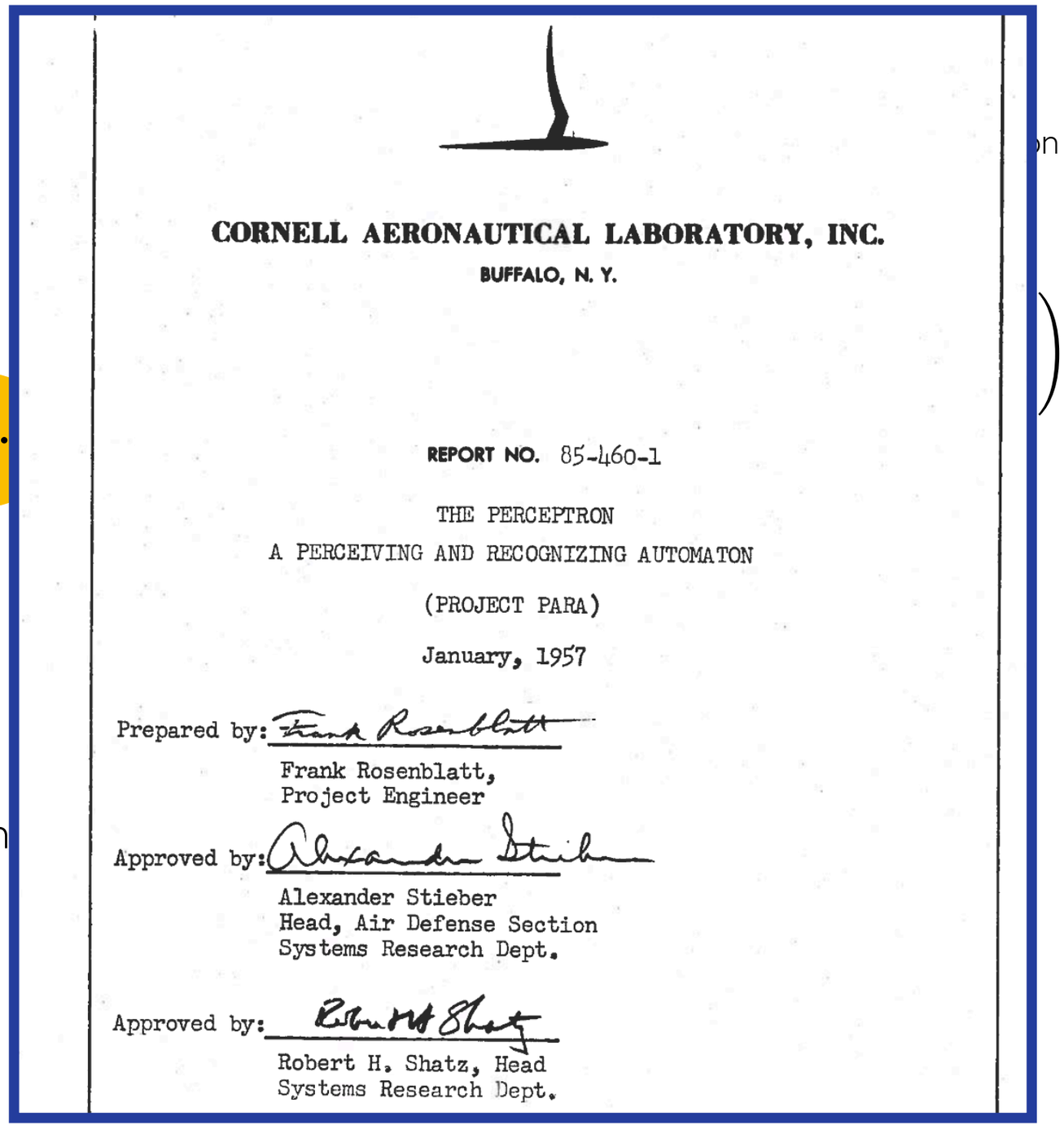



**Untangling invariant object recognition**
J DiCarlo and D Cox (2007)

# The Perceptron: The structural building block of deep learning





CORNELL AERONAUTICAL LABORATORY, INC.
BUFFALO, N. Y.

REPORT NO. 85-460-1

THE PERCEPTRON
A PERCEIVING AND RECOGNIZING AUTOMATON

(PROJECT PARA)

January, 1957

Prepared by: Frank Rosenblatt
Frank Rosenblatt,
Project Engineer

Approved by: Alexander Stieber
Alexander Stieber
Head, Air Defense Section
Systems Research Dept.

Approved by: Robert H. Shatz
Robert H. Shatz, Head
Systems Research Dept.

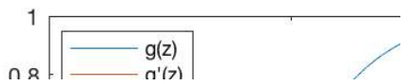


**Wikipedia Entry for Frank R.**

# Activation Function

The purpose of activation functions is to introduce non-linearities into the network

Sigmoid Functic

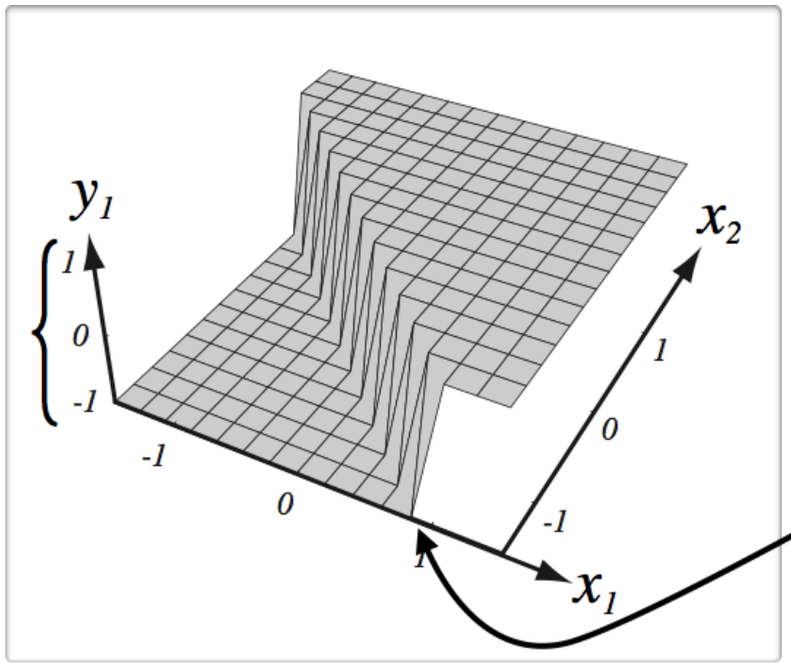



$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$ ed

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

$$g'(z) = g(z)(1 -$$

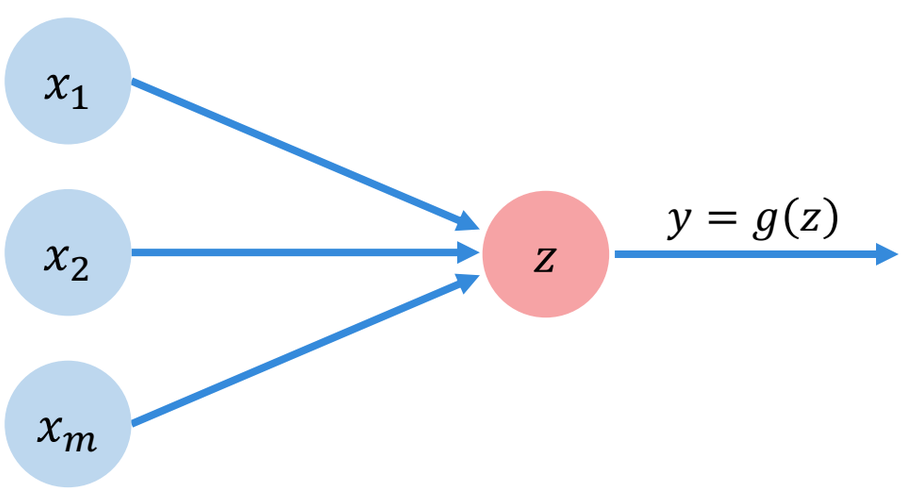



(from Pascal Vincent's slides)

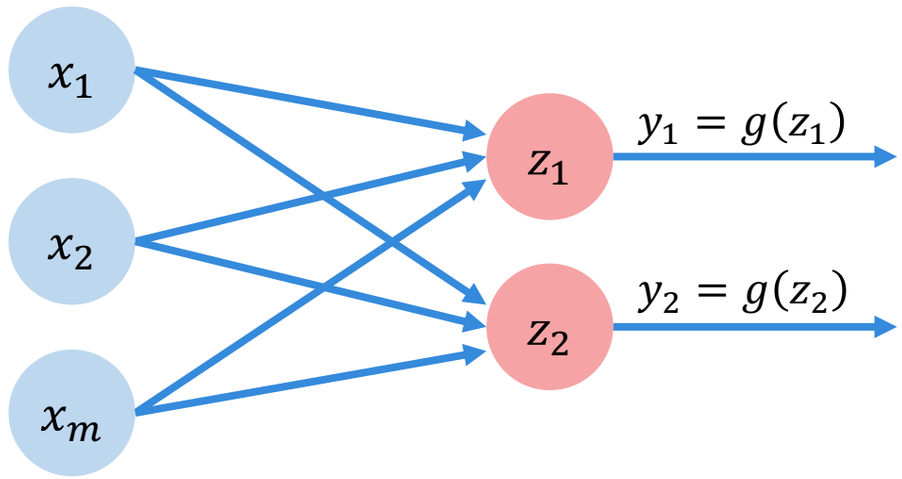
Bias only changes the position of the riff

# Multi-Output perceptron





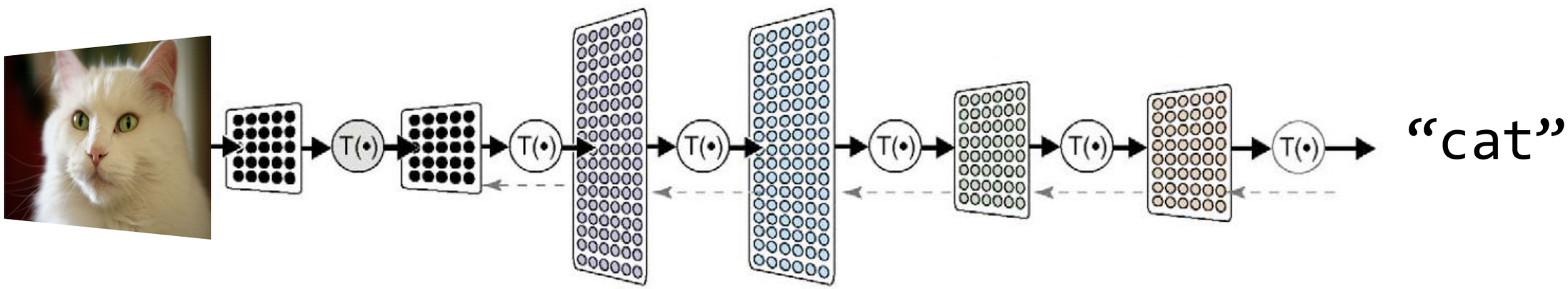

$$z = w_0 + \sum_{j=1}^{m} x_j \, w_j$$




$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

# Deep learning = artificial neural networks

**Hierarchical** composition of **simple mathematical** functions

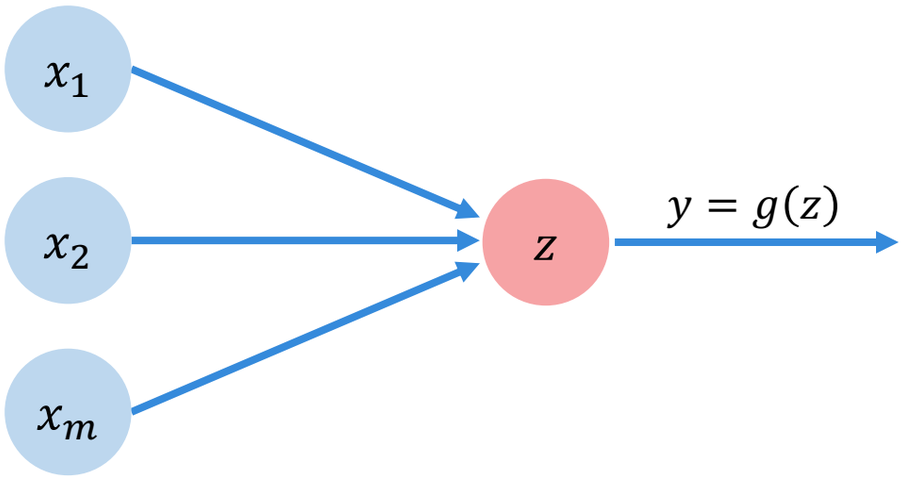



**Untangling invariant object recognition**
J DiCarlo and D Cox (2007)

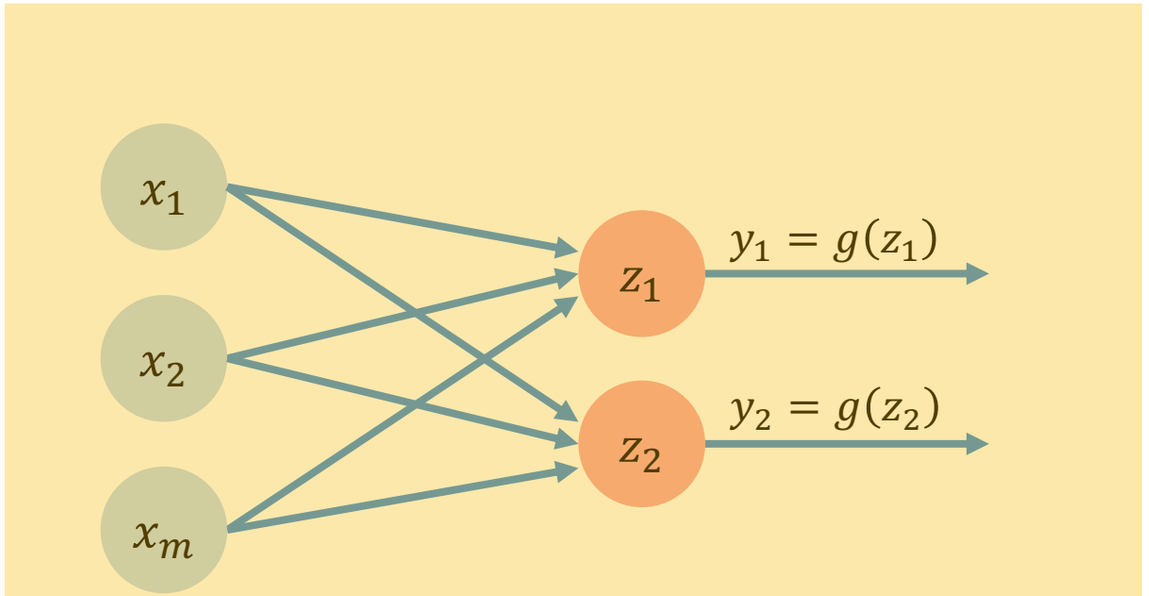
# The Multi-Layer Perceptron or Dense Neural Network

Hierarchical composition of nonlinear functions $g\left(g\left(g\left(\cdots\right)\right)\right)$





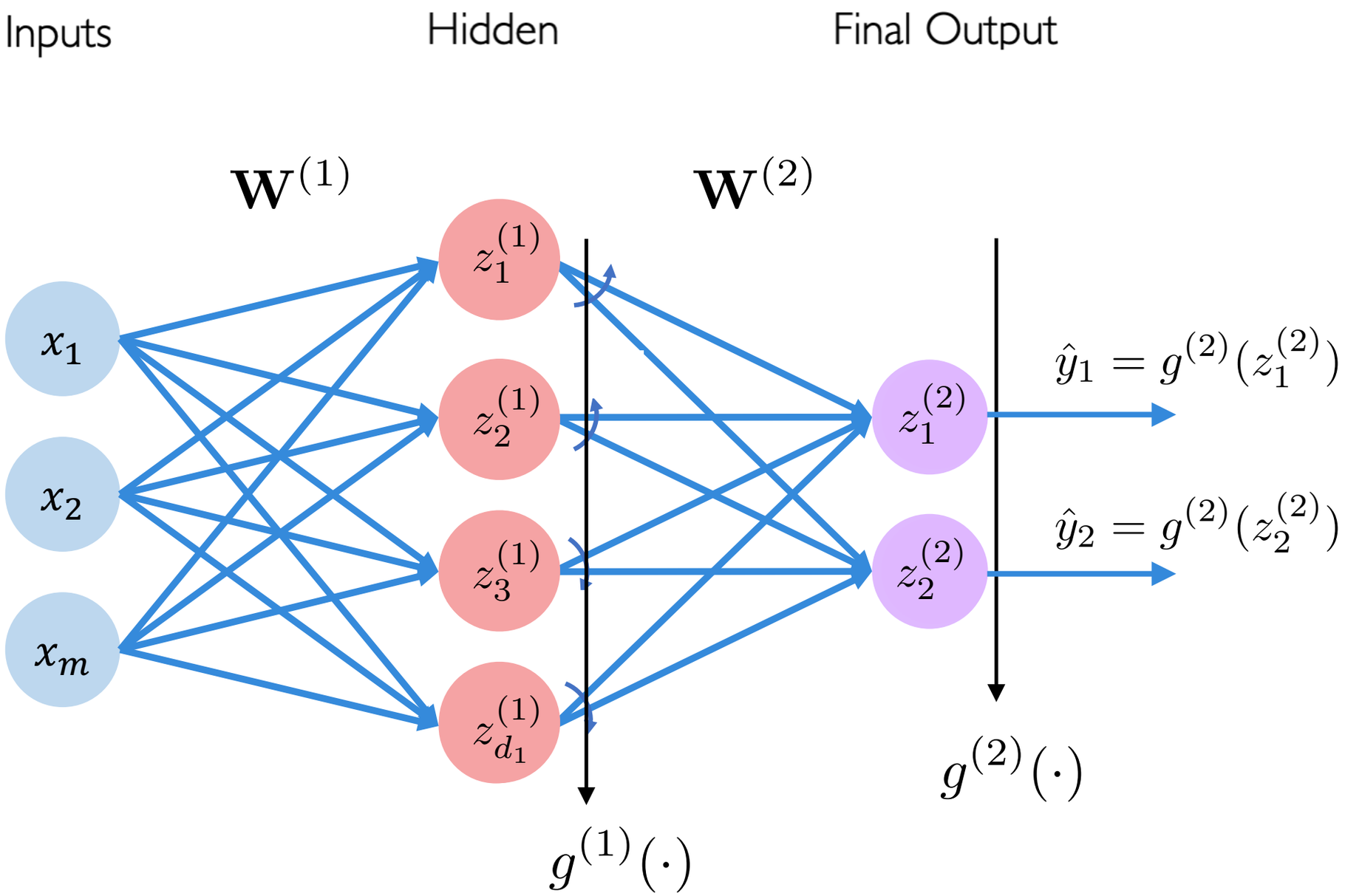

$$z = w_0 + \sum_{j=1}^{m} x_j \, w_j$$




$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

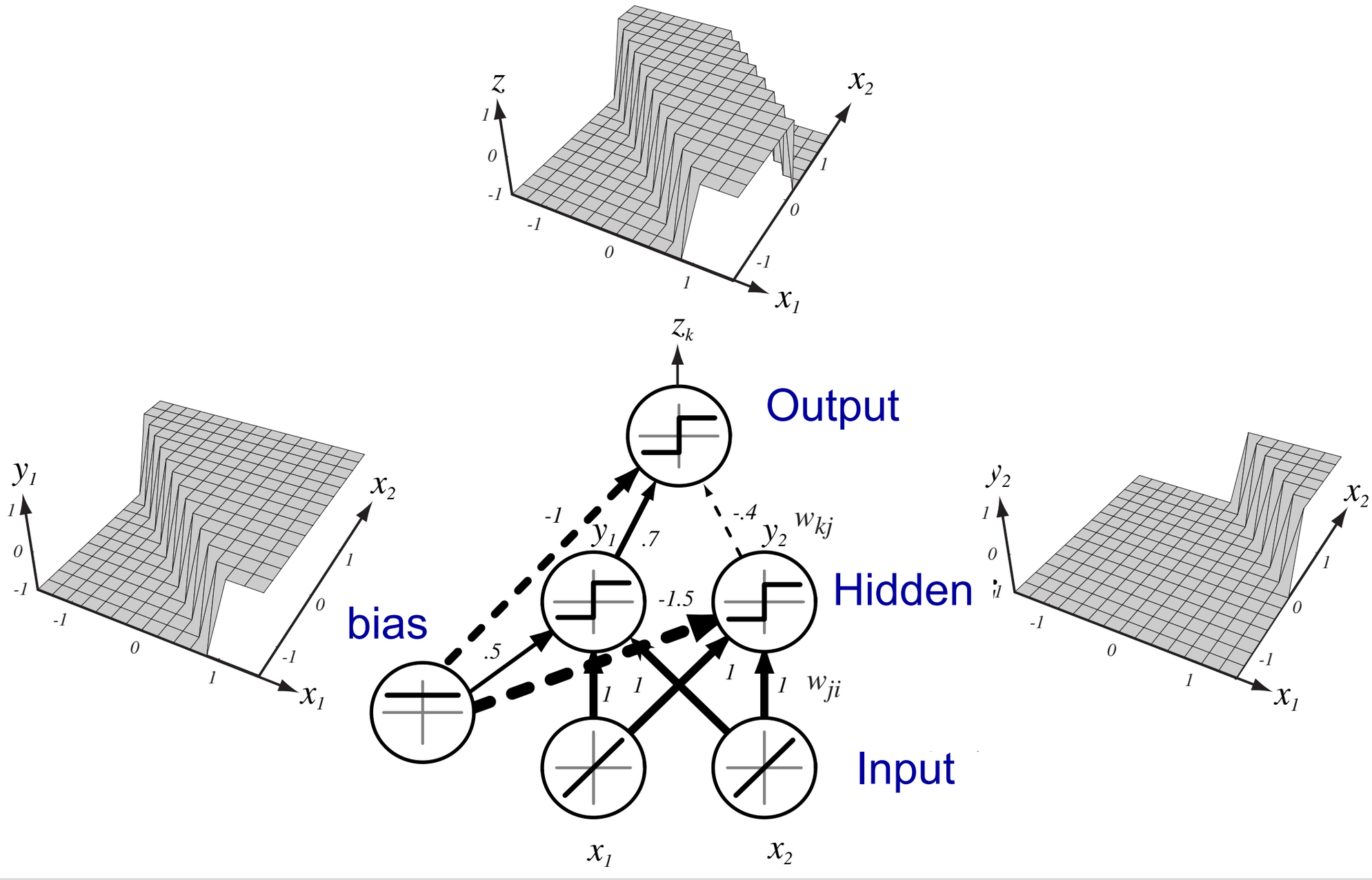
**1 layer = Linear Operation**
**+ Elementwise non-non-linear function**

# A Neural Network with two layers

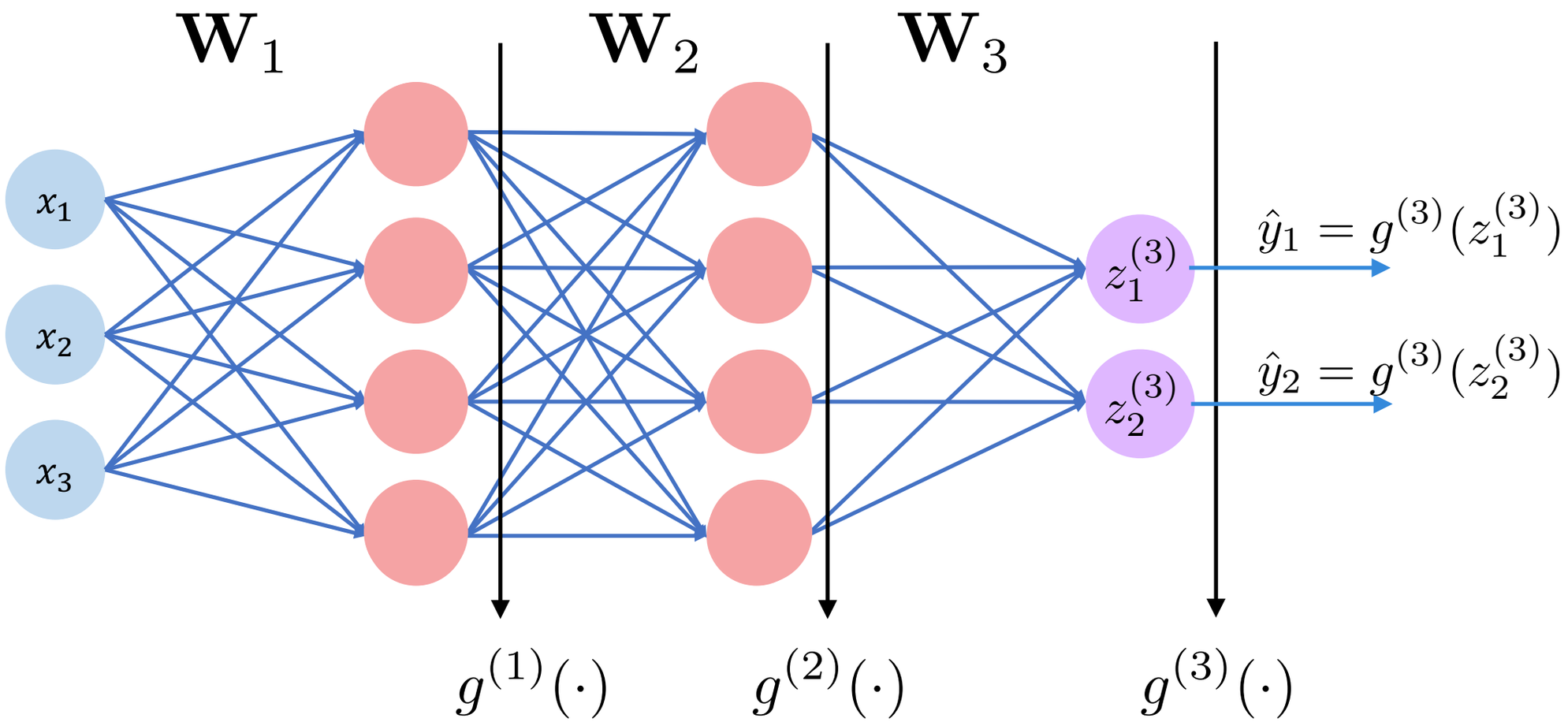



$$\hat{\mathbf{y}} = g^{(2)}\left(\mathbf{W}^{(2)}\ g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{w}_0^{(1)}\right) + \mathbf{w}_0^{(2)}\right)$$

# A Neural Network with two layers



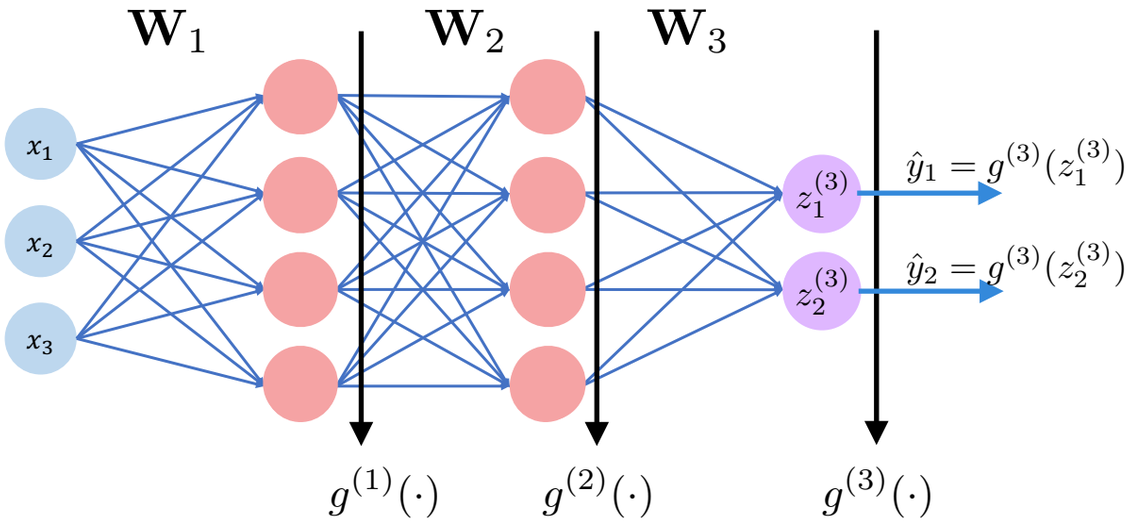
# A Neural Network with three layers






$$\hat{\mathbf{y}} = g^{(3)}\left(\mathbf{W}^{(3)}\ g^{(2)}\left(\mathbf{W}^{(2)}\ g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{w}_0^{(1)}\right) + \mathbf{w}_0^{(2)}\right) + \mathbf{w}_0^{(3)}\right)$$

# Training Neural Networks (Supervised task)




## Parameter Set

$$\mathbf{W} = \left\{ \mathbf{W}^{(i)}, \mathbf{w}_0^{(i)} \right\}_{i=1}^{3}$$

## Empirical Loss

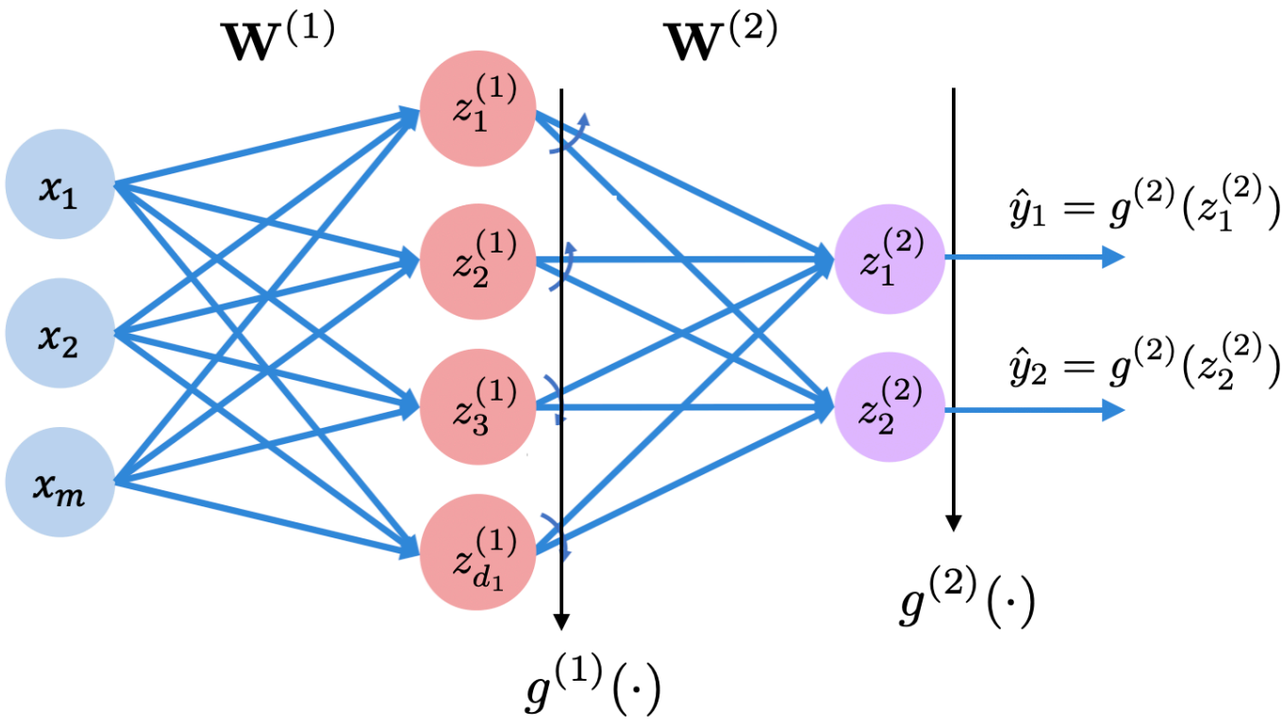
$$J(\mathbf{W}) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}\left( \hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)} \right)$$

## Model Training

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} J(\mathbf{W})$$

## Loss Function (Supervised task)

**Multi-output Regression** $\mathcal{D} = \left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)_{i=1}^{N} \quad \mathbf{x}^{(i)} \in \mathbb{R}^m \qquad \mathbf{y}^{(i)} \in \mathbb{R}^d$




**Squared loss**

$$\mathcal{L}\left(\hat{\mathbf{y}}, \mathbf{y}\right) = ||\mathbf{y} - \hat{\mathbf{y}}||^2$$

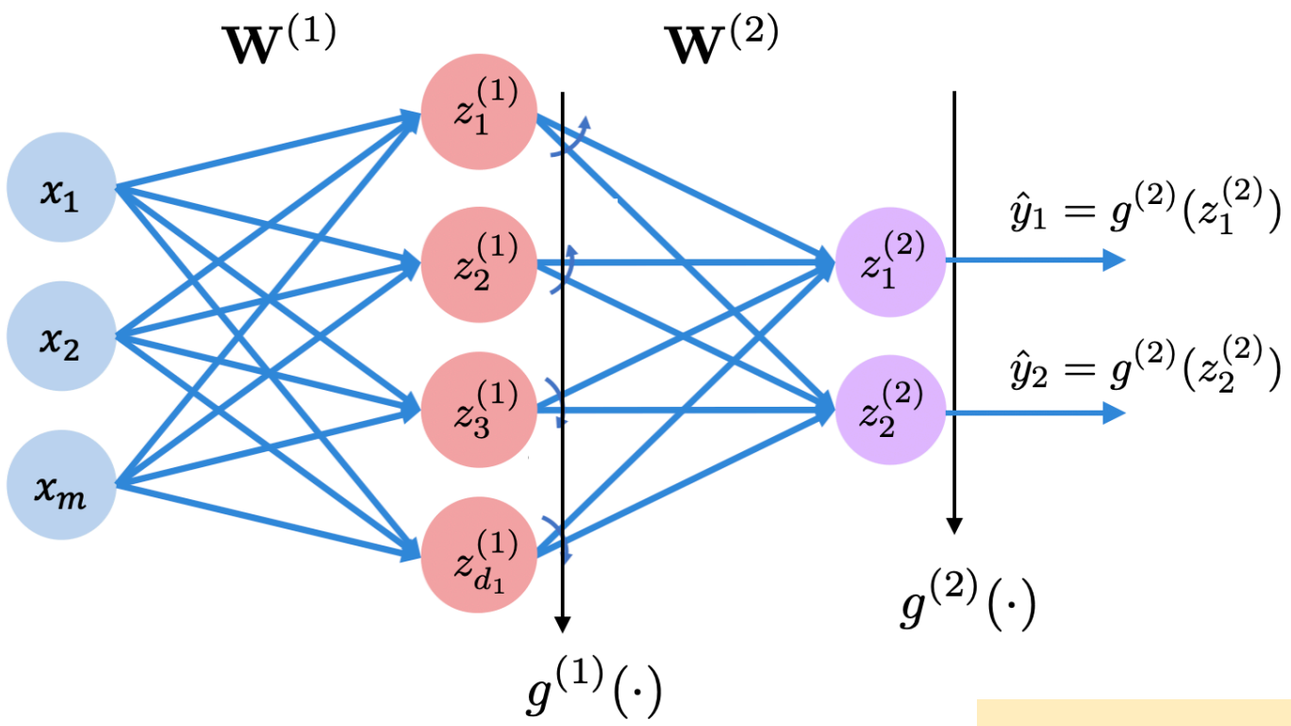
**Linear activation**

$$g^{(2)}(z) = z$$

## Loss Function (Supervised task)

**Classification** $\mathcal{D} \doteq (\boldsymbol{x}^{(i)}, y^{(i)})_{i=1}^{N} \qquad \boldsymbol{x}^{(i)} \in \mathbb{R}^m \quad y^{(i)} \in \{0, 1, \ldots, K\}$

Convert the network output to a probability distribution with the **softmax function**




### Softmax

$$\hat{y}_k = g^{(2)}(z_k^{(2)}) = P(y = k|\boldsymbol{x}) = \frac{\mathrm{e}^{z_k^{(2)}}}{\sum_{j=1}^{K} \mathrm{e}^{z_j^{(2)}}}$$

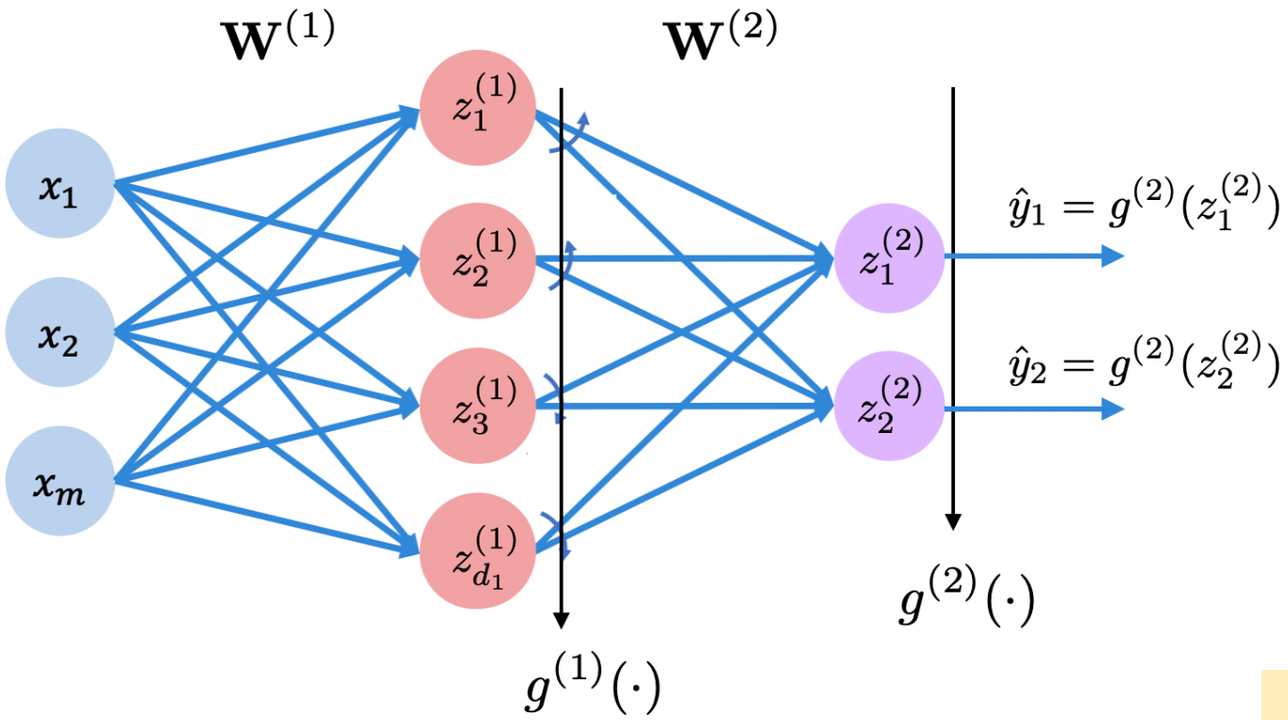
### Cross Entropy Loss Function

$$\mathcal{L}(\hat{\mathbf{y}}, y) = -\sum_{k=1}^{K} \mathbb{I}[y == k] \log P(y = k|\boldsymbol{x})$$

## Loss Function (Supervised task)

**Multilabel Classification** $\rightarrow \boldsymbol{y} = [010 \ldots 10 \ldots 1]$ is a K-dimensional binary vector

Convert the network output to a **binary probability per output** using the sigmoid activation



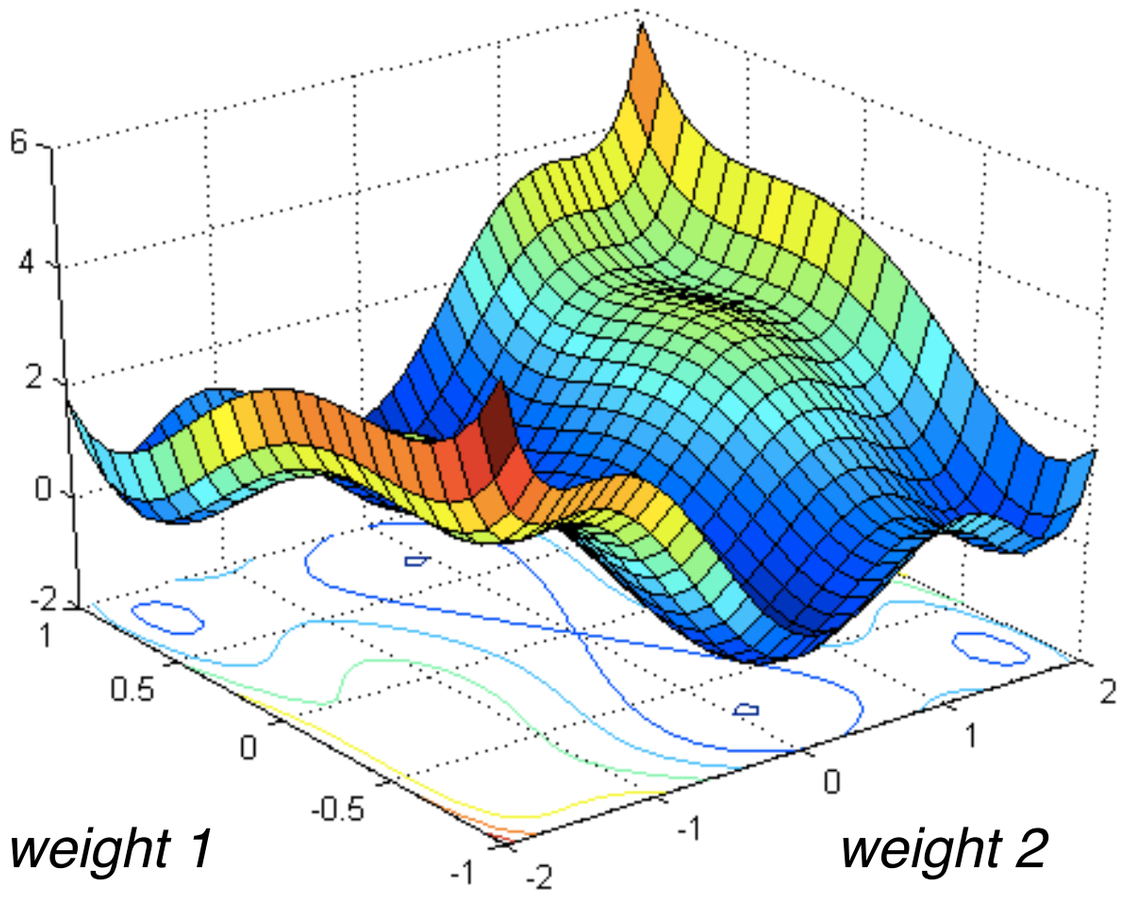

### Sigmoid Activation

$$\hat{y}_i = \sigma(z_i^{(2)}) = \frac{1}{1 + \exp(-z_i^{(2)})} = \text{Prob}(y_i = 1|\mathbf{x})$$

### Binary Cross Entropy Loss Function

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

# Optimization is highly non-convex …




$$\mathbf{W}^* = \arg\min_{\mathbf{W}} J(\mathbf{W})$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right)$$

Note that deep networks operation in O(1M) dimensions!

## … but training is brutally simple (we assume we never get to the global optima)

### Gradient Descent

Algorithm

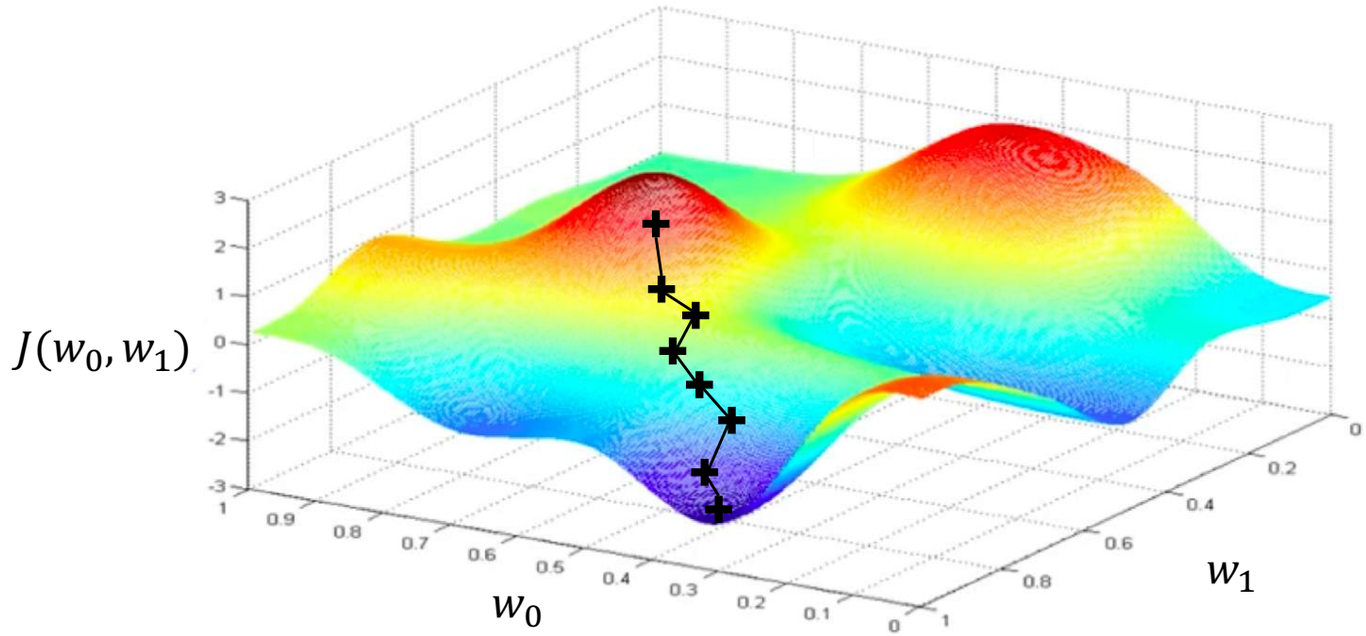
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:

   3. Compute gradient $\frac{\partial J}{\partial \mathbf{W}}$

   4. Update Weights $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J}{\partial \mathbf{W}}$

5. Return weights



## … but training is brutally simple (we assume we never get to the global optima)

### Gradient Descent

Algorithm

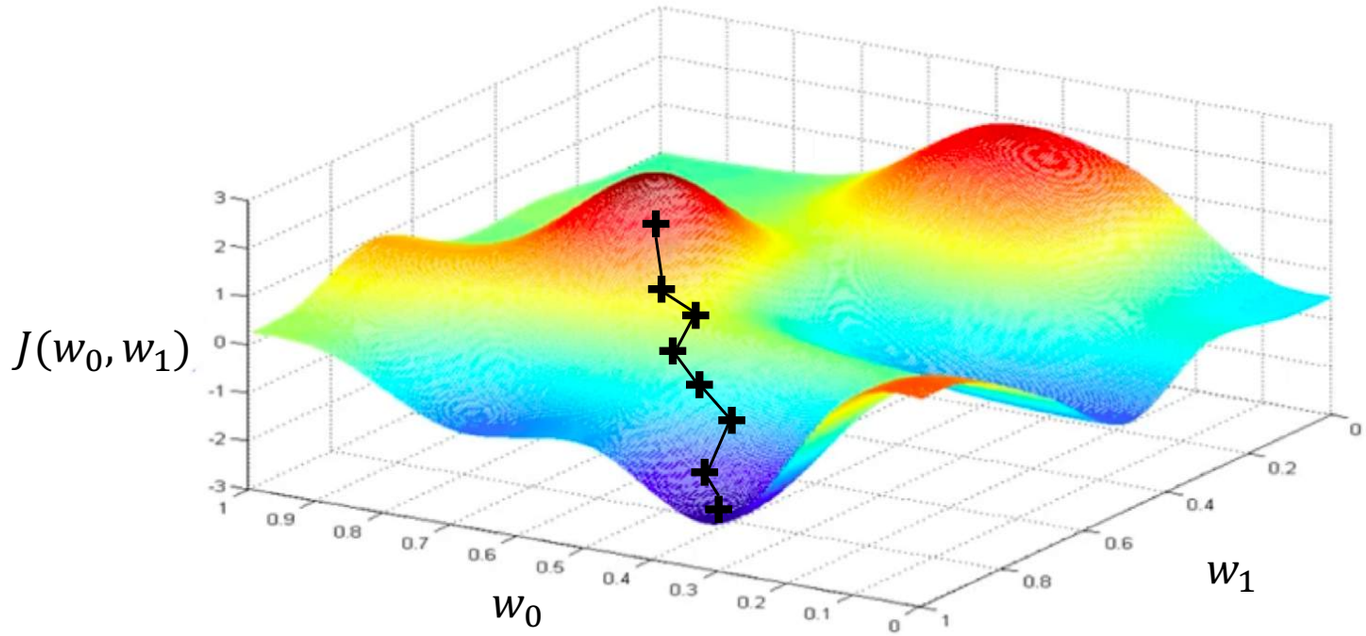
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:

   3. Compute gradient $\frac{\partial J}{\partial \mathbf{W}}$

   4. Update Weights $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J}{\partial \mathbf{W}}$

5. Return weights



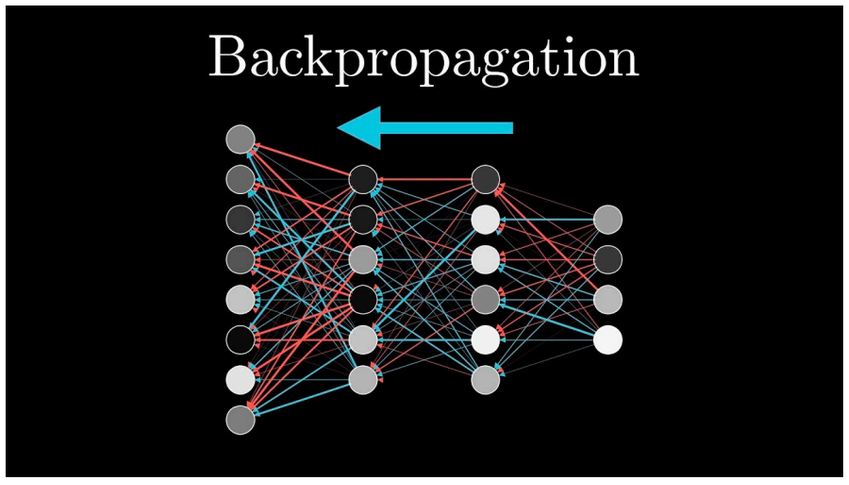



**Check out this excellent post about initialization!**

## Computing gradients efficiently: backpropagation

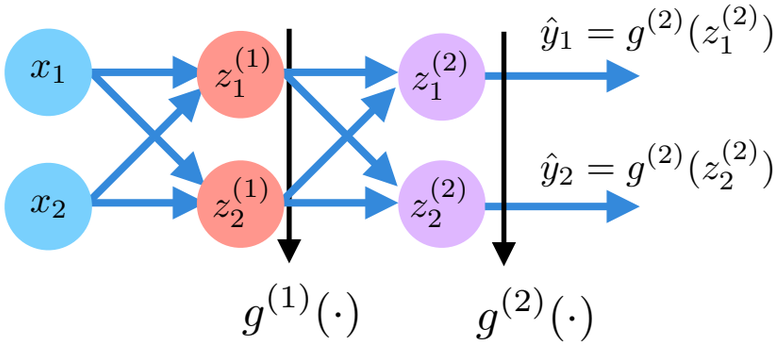
- A deep NN can contain millions of parameters
- Gradients cannot be computed individually for parameter, there must be some way of "reusing" computations
- **Backpropagation** is the key to understand why NNs have such a particular structure:
  - A multi-variate linear operator (efficiently implemented in HW)
  - Followed by element-wise non-linear functions

- Thanks to this structure, with **Backpropagation** we evaluate gradients from the top (output) of the network to the bottom (input) by reusing computations.
- Cost of computing gradients ~ cost of evaluating the NN output for a given input



# Example for a 2 layer NN with linear outputs (regression)

- $g^{(2)}(z) = z \qquad \mathcal{L}_n = ||\mathbf{y}_n - \hat{\mathbf{y}}_n||^2 \qquad \delta_i^{(2)} = \dfrac{\partial \mathcal{L}_n}{\partial z_i^{(2)}} = -2(y_1 - z_i^{(2)}), \quad i = 1, 2$




- $z_i^{(2)} = w_{i1}^{(2)} g_1(z_1^{(1)}) + w_{i2}^{(2)} g_1(z_2^{(1)}) + w_{0,i}^{(2)}$
- $\dfrac{\partial \mathcal{L}_n}{\partial w_{i1}^{(2)}} = \dfrac{\partial \mathcal{L}_n}{\partial z_i^{(2)}} \dfrac{z_i^{(2)}}{\partial w_{i1}^{(2)}} = \delta_i^{(2)} g_1(z_1^{(1)})$
- $\dfrac{\partial \mathcal{L}_n}{\partial w_{i2}^{(2)}} = \dfrac{\partial \mathcal{L}_n}{\partial z_i^{(2)}} \dfrac{z_i^{(2)}}{\partial w_{i2}^{(2)}} = \delta_i^{(2)} g_1(z_2^{(1)})$

- $\dfrac{\partial \mathcal{L}_n}{\partial z_i^{(1)}} = \delta_i^{(1)} = \dfrac{\partial \mathcal{L}_n}{\partial z_1^{(2)}} \dfrac{\partial z_1^{(2)}}{\partial z_i^{(1)}} + \dfrac{\partial \mathcal{L}_n}{\partial z_2^{(2)}} \dfrac{\partial z_2^{(2)}}{\partial z_i^{(1)}} = \delta_1^{(2)} w_{1i}^{(2)} g_1'(z_i^{(1)}) + \delta_2^{(2)} w_{2i}^{(2)} g_1'(z_i^{(1)}) = g_1'(z_i^{(1)}) \left( \delta_1^{(2)} w_{1i}^{(2)} + \delta_2^{(2)} w_{2i}^{(2)} \right)$

$$\delta_j^{(m-1)} = g'(z_j^{(m-1)}) \sum_{k=1}^{d_m} w_{k,j} \delta_k^{(m)}$$

# Stochastic Backpropagation

Assume an M-layer MLP

$$\mathbf{x} \longrightarrow \mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} \longrightarrow \mathbf{g}^{(1)} = g(\mathbf{z}^{(1)}) \longrightarrow \mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{g}^{(1)} \longrightarrow \cdots \longrightarrow \hat{\mathbf{y}} = g(\mathbf{z}^{(M)})$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) = \sum_{n=1}^{N} \mathcal{L}_n \longrightarrow \frac{\partial J(\mathbf{W})}{\partial w_{ji}^{(m)}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}}$$

Very costly for massive datasets!

**Stochastic Gradient Descent (SGD)**

Select at random a mini batch $\mathcal{B}$ of data at every SGD iteration

$$\frac{\partial J(\mathbf{W})}{\partial w_{ji}^{(m)}} \approx \sum_{n \in \mathcal{B}} \frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}}$$

**Parallelizable!**
**Very efficient in GPUs**

## Gradient Descent

Algorithm

1. Initialize weights randomly $\sim^{\text{iid}} \mathcal{N}(0, \sigma^2$

2. Loop until convergence:

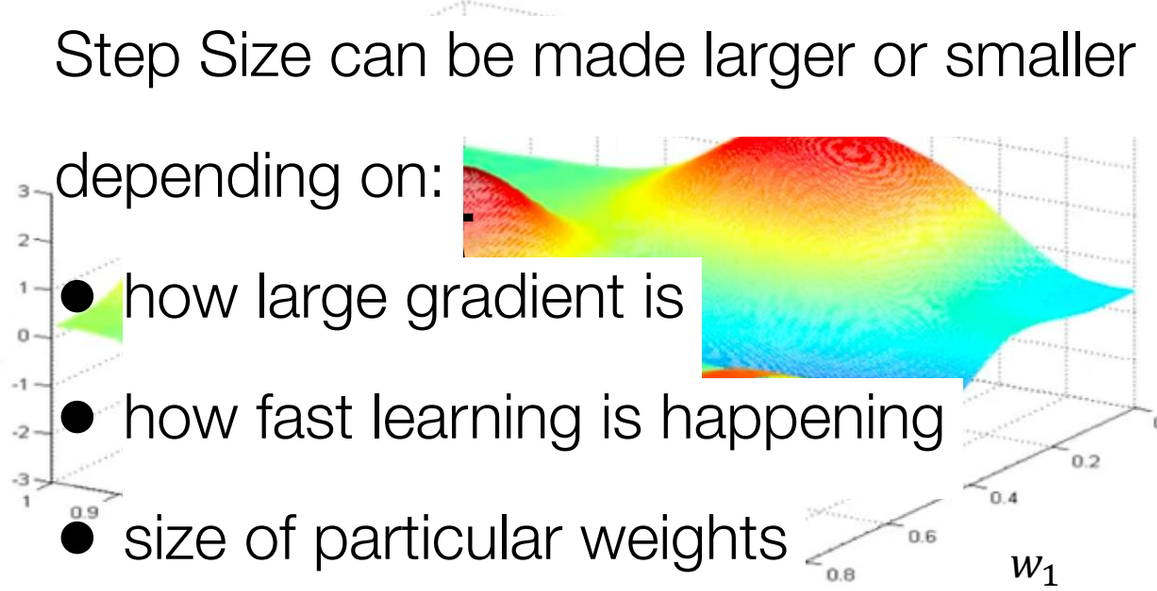
3. Compute (noisy) gradient $\frac{\partial J}{\partial \mathbf{W}}$

4. Update Weights $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J}{\partial \mathbf{W}}$

5. Return weights

## Adaptive Learning Rates

Step Size can be made larger or smaller depending on:




- how large gradient is
- how fast learning is happening
- size of particular weights
- etc...

**Momentum, Adagrad, Adam, RMSProp, … (Check out this post)**

## Model Selection

- Train your model on the **training set** $\mathcal{D}_{\text{train}}$

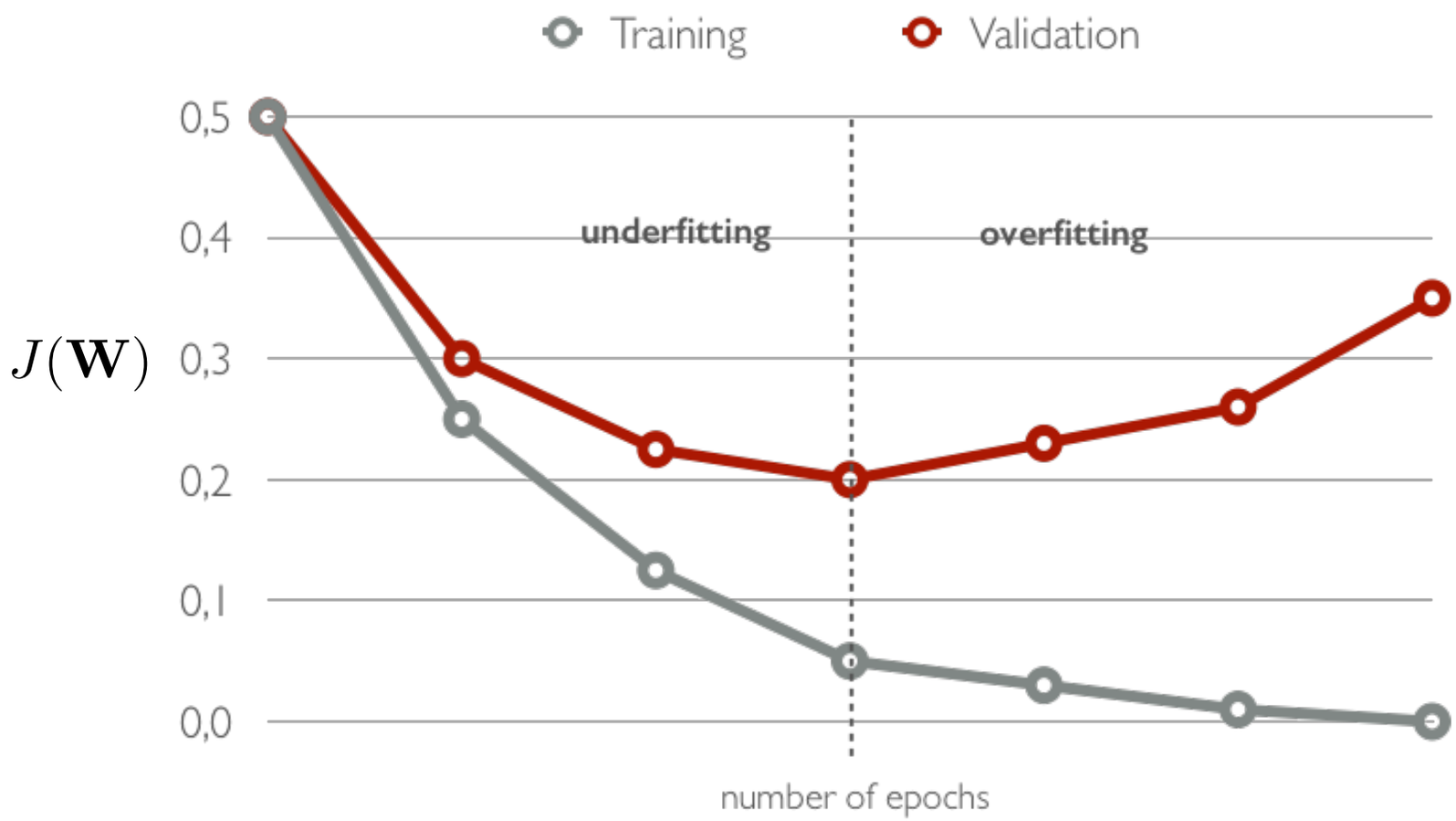
- For model selection, use a **validation set** $\mathcal{D}_{\text{val}}$

  ✓Hyper-parameter search: hidden layer sizes, number of layers, learning rate, number of iterations/epochs, …

- Estimate generalisation performance using a **test set** $\mathcal{D}_{\text{test}}$

## Regularization: Early Stopping

To select the number of epochs, stop training **when validation set error increases** (with some look ahead)

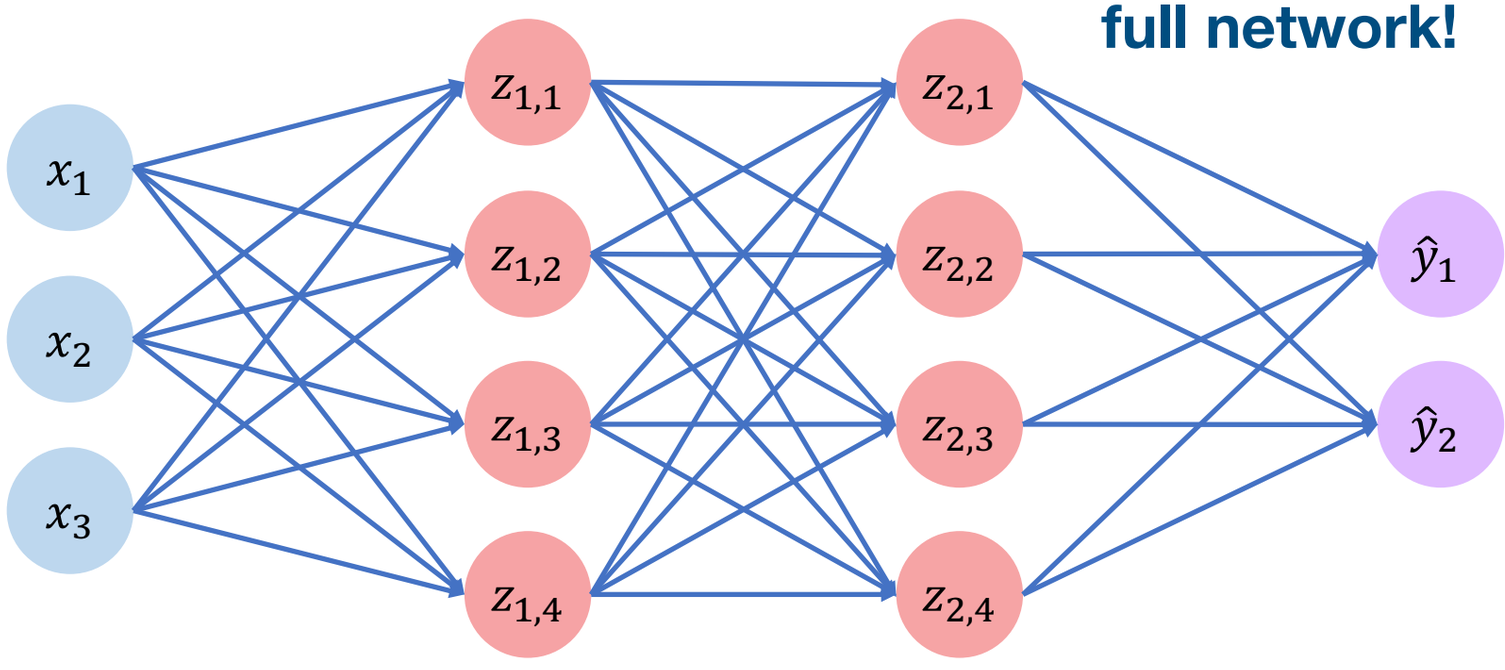


## Regularization: Dropout

**During training** cripple neural network by **removing hidden units stochastically**

- Randomly set some activations to 0
- Typically ‘drop’ 50% of activations in layer
- Forces network to not rely on any 1 node

**In validation & test we use the full network!**

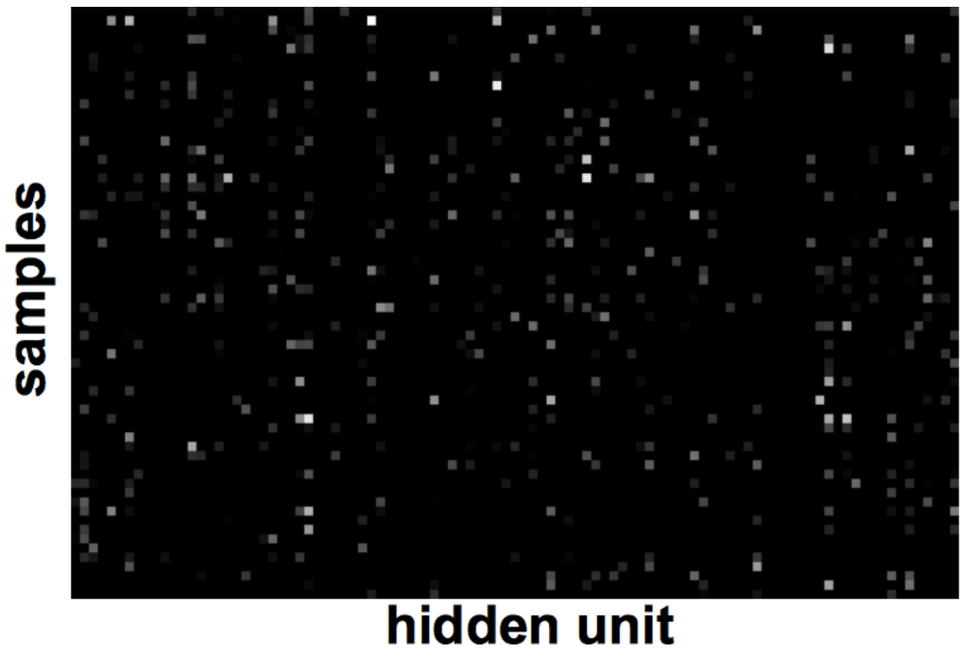


## Usual Practice

- Given a dataset $\mathcal{D}_{\text{train}}$, pick a model so that:
  - ✓ You can achieve 0 training error– **Overfit** on the training set
- **Regularize** the model
- SGD with Adam or Momentum, early stopping and Dropout
- Pick learning rate by running on a subset of the data
  - ✓Start with **large learning** rate & divide by 2 until loss does not diverge
- **Decay learning rate** by a factor of ~100 or more by the end of training
- Use **ReLU** nonlinearity
- Initialize parameters so that each feature across layers has similar variance. **Avoid units in saturation**.

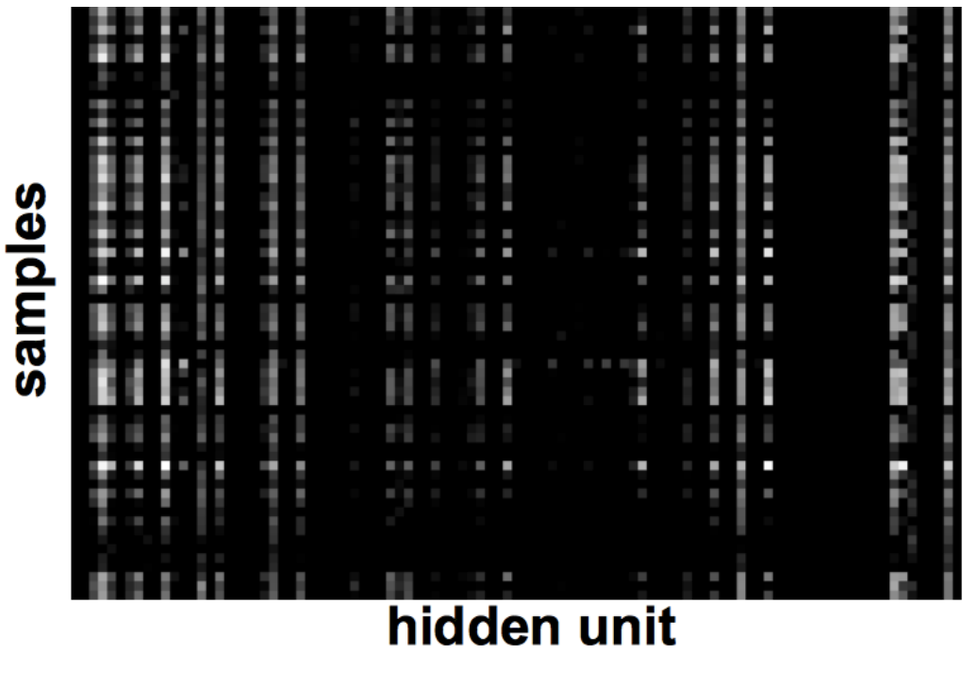
## Feature Visualization

Visualize features (features need to be **uncorrelated**) and have **high variance**




**Good Training**

Hidden units are sparse across samples




**BAD Training**

Many hidden units ignore the input and/or exhibit strong correlations

# Backpropagation formula: proof

# Computing Gradients: Backpropagation

Assume an M-layer MLP

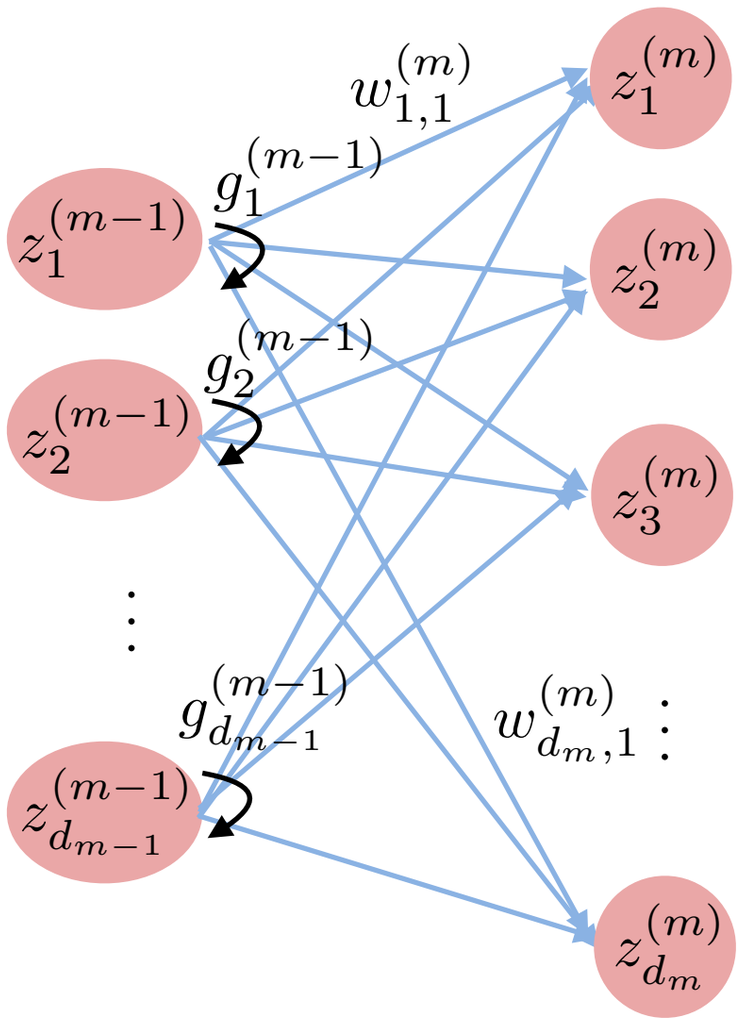
$$\mathbf{x} \longrightarrow \mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} \longrightarrow \mathbf{g}^{(1)} = g(\mathbf{z}^{(1)}) \longrightarrow \mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{g}^{(1)} \longrightarrow \cdots \longrightarrow \hat{\mathbf{y}} = g(\mathbf{z}^{(M)})$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) = \sum_{n=1}^{N} \mathcal{L}_n$$

$$\frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m)}} \frac{\partial z_j^{(m)}}{\partial w_{ji}^{(m)}} \longrightarrow \delta_j^{(m)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m)}}$$

$$\frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}} = \delta_j^{(m)} g_i^{(m-1)}$$

$$z_j^{(m)} = \sum_{i=1}^{d_{m-1}} w_{j,i}\ g_i^{(m-1)} \longrightarrow \frac{\partial z_j^{(m)}}{\partial w_{ji}^{(m)}} = g_i^{(m-1)}$$



# Computing Gradients: Error Backpropagation

$$\mathbf{x} \longrightarrow \mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} \longrightarrow \mathbf{g}^{(1)} = g(\mathbf{z}^{(1)}) \longrightarrow \mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{g}^{(1)} \longrightarrow \cdots \longrightarrow \hat{\mathbf{y}} = g(\mathbf{z}^{(M)})$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) = \sum_{n=1}^{N} \mathcal{L}_n \qquad \frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}} = \delta_j^{(m)} g_i^{(m-1)} \qquad \delta_j^{(m)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m)}}$$

## Linear output activation

$$\hat{\mathbf{y}} = \mathbf{z}^{(M)} \qquad \mathcal{L}_n = ||\mathbf{y} - \hat{\mathbf{y}}||^2$$

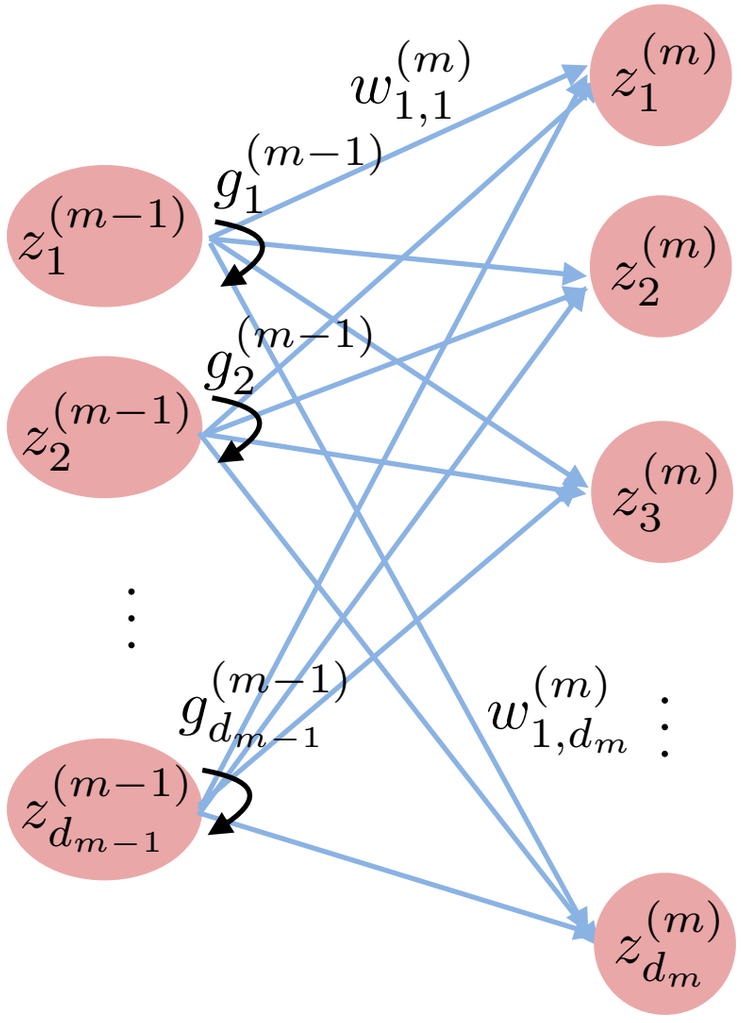
$$\delta_j^{(M)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(M)}} = \frac{\partial \mathcal{L}_n}{\partial \hat{y}_j} = -2(y_j - \hat{y}_j)$$

## Sigmoid output activation

$$\hat{y}_j = \frac{1}{1 + \exp(-z_j^{(M)})}$$

$$\mathcal{L}_n = \sum_{i=1}^{d} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

$$\delta_j^{(M)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(M)}} = -(y_j - \hat{y}_j)$$



# Computing Gradients: Error Backpropagation

$$\mathbf{x} \longrightarrow \mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} \longrightarrow \mathbf{g}^{(1)} = g(\mathbf{z}^{(1)}) \longrightarrow \mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{g}^{(1)} \longrightarrow \cdots \longrightarrow \hat{\mathbf{y}} = \mathbf{W}^{(M)}\mathbf{g}^{(M-1)}$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) = \sum_{n=1}^{N} \mathcal{L}_n \qquad \frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}} = \delta_j^{(m)} g_i^{(m-1)} \qquad \delta_j^{(m)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m)}}$$

$$\delta_j^{(m-1)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m-1)}} = \sum_{k=1}^{d_m} \frac{\partial \mathcal{L}_n}{\partial z_k^{(m)}} \frac{\partial z_k^{(m)}}{\partial z_j^{(m-1)}}$$

$$\frac{\partial z_k^{(m)}}{\partial z_j^{(m-1)}} = w_{k,j}\ g'(z_j^{(m-1)})$$

Learning representations
by back-propagating errors

Rumelhart, Hinton, Williams (1986)

$$\delta_j^{(m-1)} = g'(z_j^{(m-1)}) \sum_{k=1}^{d_m} w_{k,j}\delta_k^{(m)}$$