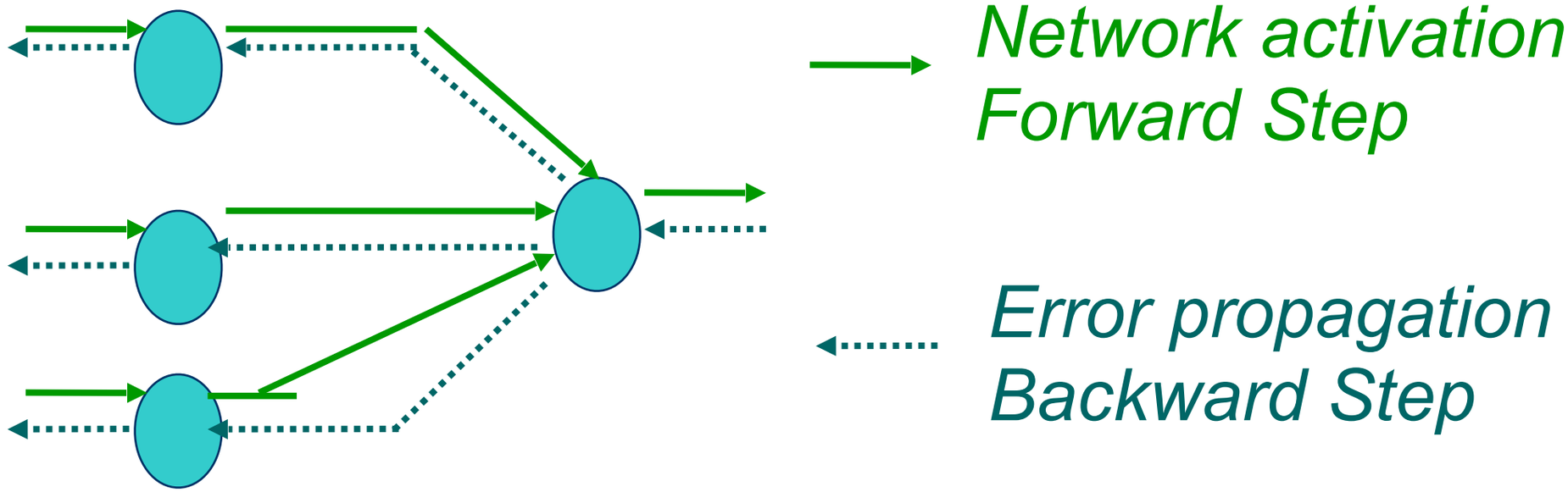
Back Propagation

Foundations of Data Analysis

April 13, 2023

Training: Back Propagation

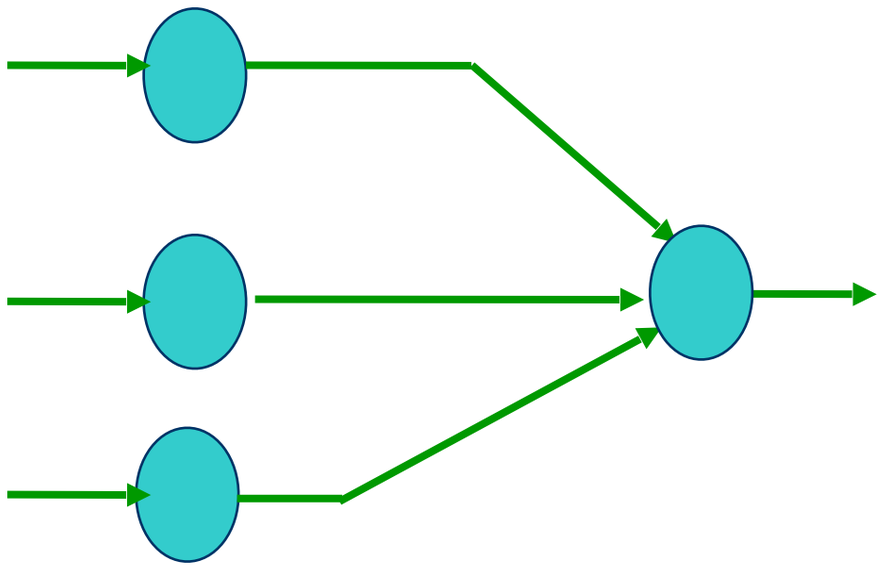
- Back-propagation training algorithm



- Back propagation adjusts the weights of the NN in order to minimize the network total mean squared error

Training: Back Propagation

- Back-propagation training algorithm

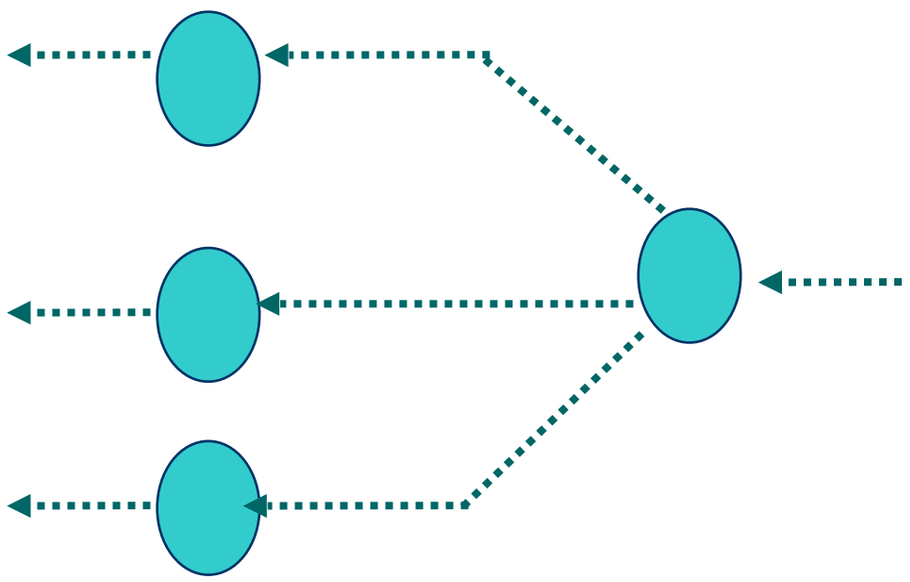


Network activation
Forward Step

  - **Forward pass**: in this step the network is activated on one example and the error of (each neuron of) the output layer is computed.

Training: Back Propagation

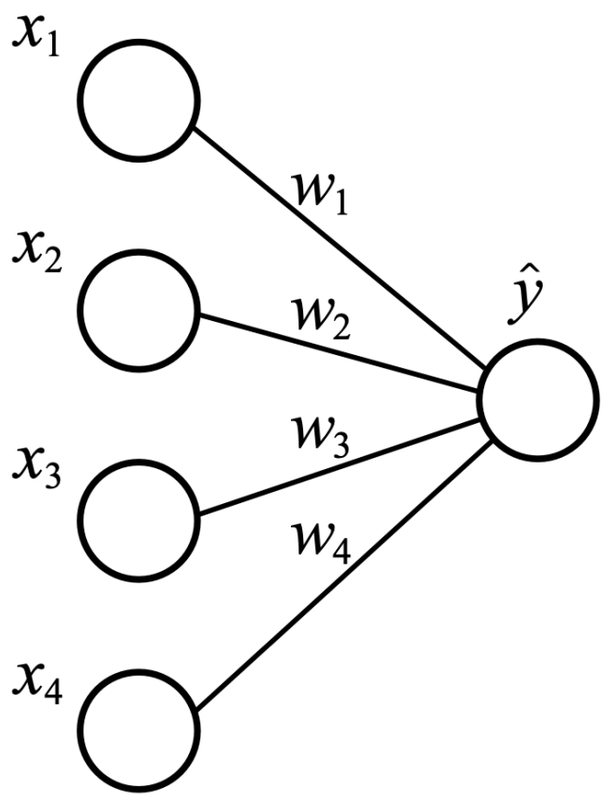
- Back-propagation training algorithm



Error propagation
Backward Step

  - **Backward pass**: in this step the network error is used for updating the weights. Starting at the output layer, the error is propagated backwards through the network, layer by layer. This is done by recursively computing the local gradient of each neuron.

Single Layer Model



Prediction:

$$\hat{y} = \phi(Wx)$$
$$= \phi(w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4)$$

Minimize loss between prediction, $\hat{y}$, and true value, *y*:

$$L(y, \hat{y})$$

This represents many different models we've seen!

Loss Functions for Regression

Dependent variable data: $y \in \mathbb{R}$

Model predictions: $\hat{y} \in \mathbb{R}$

Mean squared error (MSE): (Linear regression)

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Mean absolute error (MAE):

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} \left| y_i - \hat{y}_i \right|$$

Loss Functions for Classification

Binary labels: $y \in \{-1, 1\}$

Continuous score predictions: $\hat{y} \in \mathbb{R}$

Zero-One loss: (Perceptron)

$$L(y, \hat{y}) = \begin{cases} 1 & \text{if } y\hat{y} \leq 0 \\ 0 & \text{if } y\hat{y} > 0 \end{cases}$$

Loss Functions for Classification

Binary labels: $y \in \{0,1\}$
Probability predictions: $\hat{y} \in [0,1]$ predicts $p(y = 1 \mid x)$

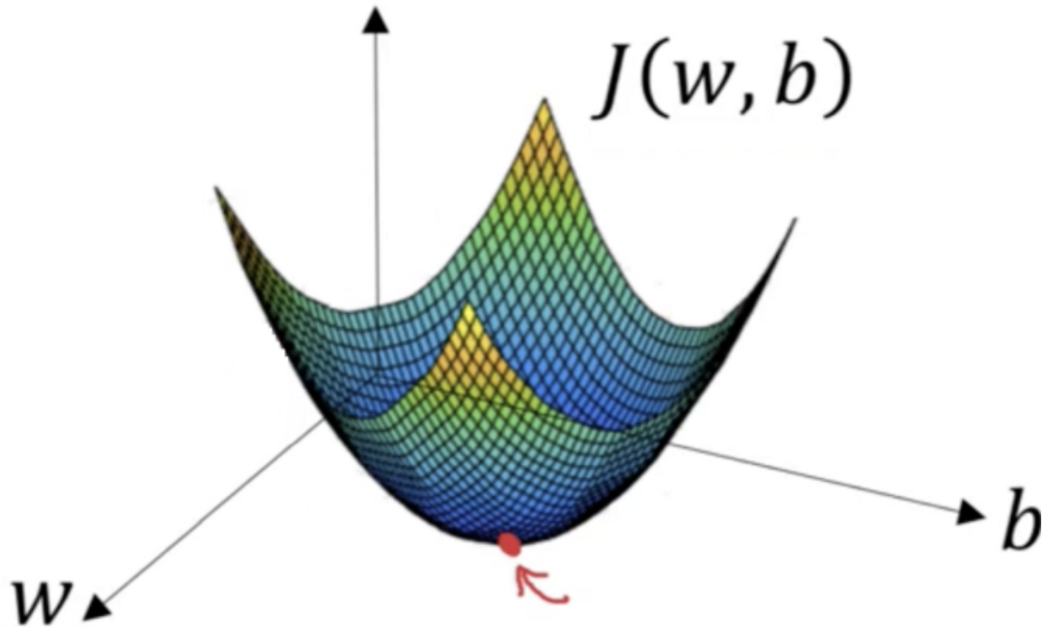
Cross entropy: (Logistic regression)

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

Gradient Decent To Optimize The Loss

Gradient: the derivative of a multi-variable function, e.g.,

$\nabla J(w, b)$ is the gradient of function $J(w, b)$



Gradient decent is to find the minimal value of a function.

Chain Rule

Given two differentiable functions,

$$f : \mathbb{R} \to \mathbb{R}, \quad g : \mathbb{R} \to \mathbb{R},$$

the derivative of their composition is:

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x)$$

Multivariate Mappings

Given a multivariate mapping,

$$g : \mathbb{R}^p \to \mathbb{R}^q,$$

we can write it as q multivariate functions:

$$g(x_1, \ldots, x_p) = (g_1(x_1, \ldots, x_p), \ldots, g_q(x_1, \ldots, x_p)) .$$

Jacobian Matrix

Partial derivatives: $D_j g_i = \dfrac{\partial g_i}{\partial x_j}$

The **Jacobian matrix (mathematical notation D)** is the $q \times p$ matrix of partial derivatives:

$$Dg = \begin{pmatrix} D_1 g_1 & D_2 g_1 & \dots & D_p g_1 \\ D_1 g_2 & D_2 g_2 & \dots & D_p g_2 \\ \vdots & \vdots & & \vdots \\ D_1 g_q & D_2 g_q & \dots & D_p g_q \end{pmatrix}$$

Multivariate Chain Rule

Given two differentiable functions,

$$f : \mathbb{R}^q \to \mathbb{R}^r, \qquad g : \mathbb{R}^p \to \mathbb{R}^q,$$

the derivative of their composition is:

$$D[f(g(x))] = Df(g(x))Dg(x)\,.$$

Note: This is a matrix multiplication on the right.

Gradient Chain Rule

Given a multivariate function,

$$f : \mathbb{R}^q \rightarrow \mathbb{R},$$

the Jacobian matrix is the same thing as the transposed gradient (a.k.a., derivative of a multi-variable function):

$$Df(x) = \left(\frac{\partial f}{\partial x_1} \cdots \frac{\partial f}{\partial x_q} \right) = \nabla f(x)^T$$

Single-Layer Neural Network

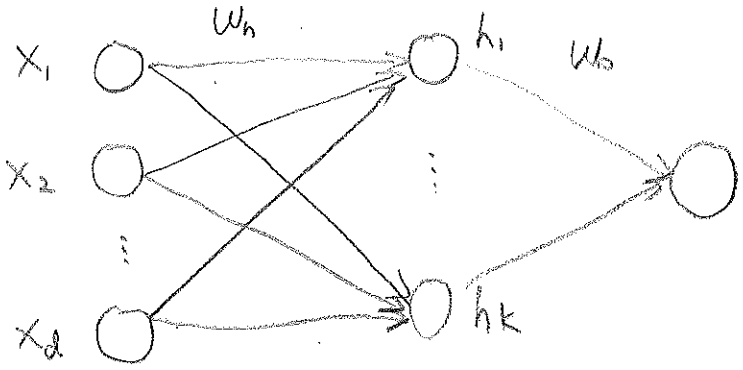
$z = \phi(Wx) = \phi(g(W, x))$, g is an activation function,

Gradient for weights:

$$\frac{\partial z_k}{\partial W_{ij}} = \phi'(g_k(W, x))D_{ij}g_k(W, x)$$

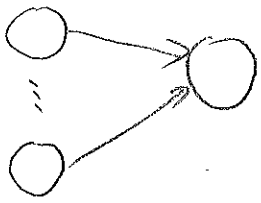
Note: ϕ' is the (univariate) derivative of ϕ.

Two-layer Neural Network



(Input layer) (hidden layer) (output layer)

traditional perceptron, Single layer NN



W_h : weights from hidden layer : $k * (d+1)$ (extra dimen is bias)

W_0 : weights from output : $k+1$ vector

Loss :

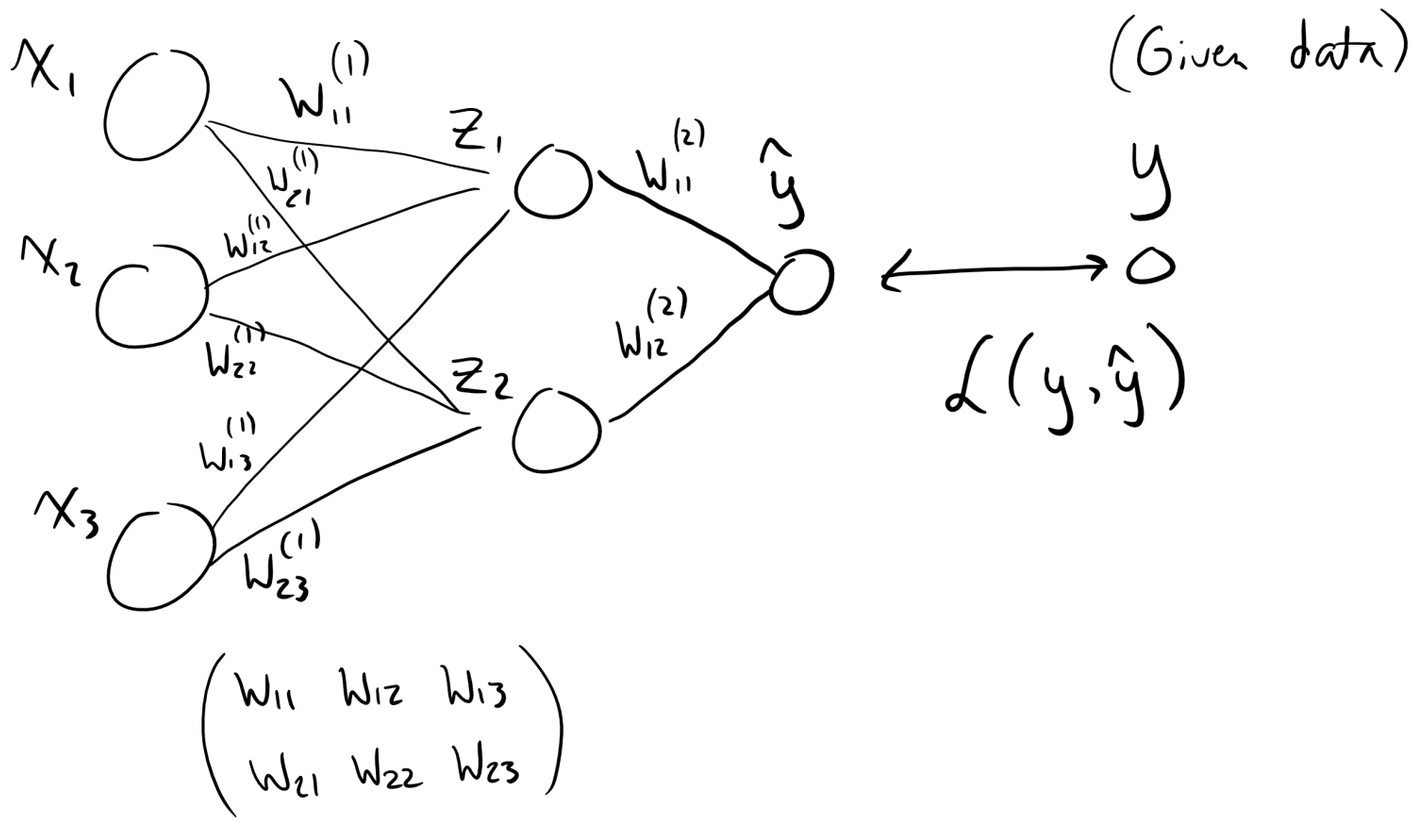
Another sigmoid ? Not necessary, but might reflect the accuracy. Feel free to try it out.

$$\sum_n \left\| y_n - \underline{W_0^T \cdot \text{sigmoid}\left[W_h^T x_n\right]} \right\|_2^2 + \underline{\lambda} \|W_0\|_2^2 + \underline{\lambda} \|W_h\|_2^2 ,$$

Could be different.

here again, x is a d-dimensional input vector, plus an extra dimension of bias

Two-Layer Neural Network



Layer 1: $z = \phi(W^{(1)}x)$

Layer 2: $\hat{y} = \phi(W^{(2)}z)$

Final Loss Function: $L(y, \hat{y})$

Second-Layer Derivative

Gradient of loss wrt $W^{(2)}$:

$$\frac{\partial L}{\partial W_{ij}^{(2)}} = \left(\frac{dL}{d\hat{y}}\right)\left\{\frac{\partial \hat{y}}{\partial W_{ij}^{(2)}}\right\}$$

$$= \left(\frac{dL}{d\hat{y}}\right)\left\{\phi'(g(W^{(2)}, z))D_{ij}g(W^{(2)}, z)\right\}$$

First-Layer Derivative

Gradient of loss wrt $W^{(1)}$:

$$\frac{\partial L}{\partial W_{ij}^{(1)}} = \left(\frac{dL}{d\hat{y}}\right) \{D_z \hat{y}\} \left\{\frac{\partial z}{\partial W_{ij}^{(1)}}\right\}$$

$$= \left(\frac{dL}{d\hat{y}}\right) \left\{\phi'(g(W^{(2)}, z)) D_{ij} g(W^{(2)}, z)\right\}$$

$$\times \left[D_{ij} g(W^{(1)}, x)\right]$$