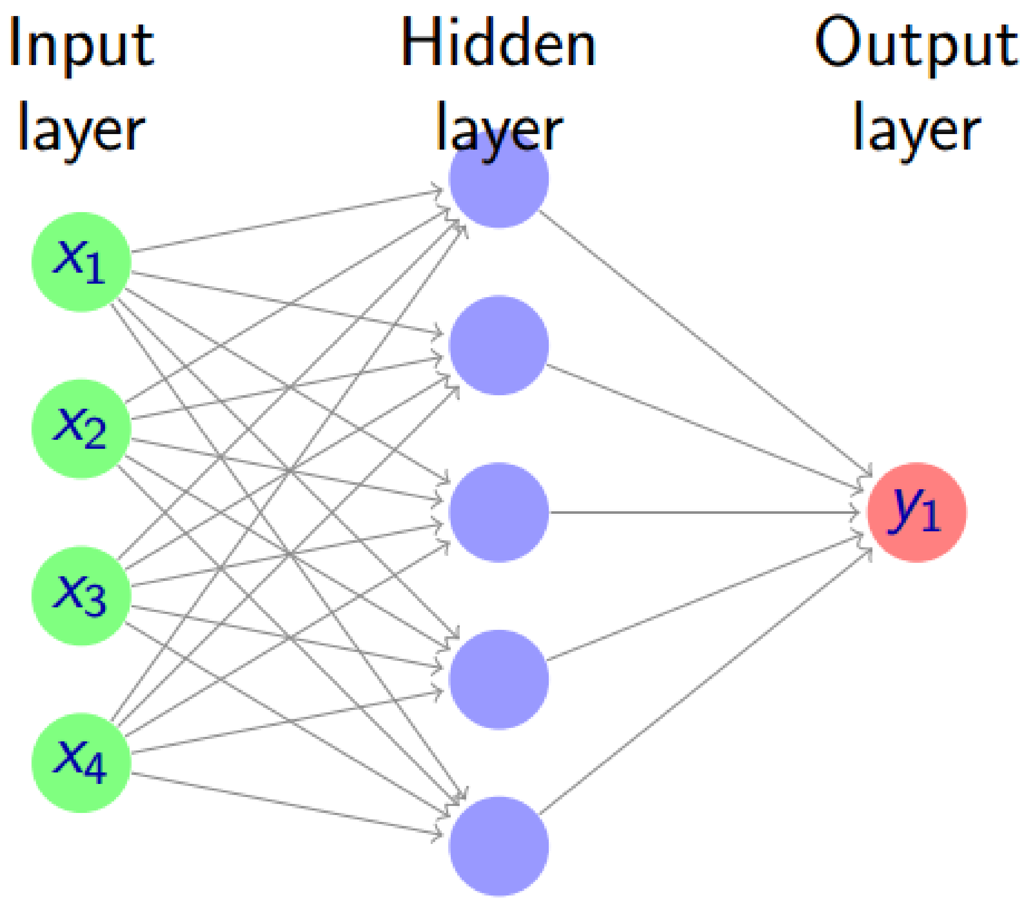
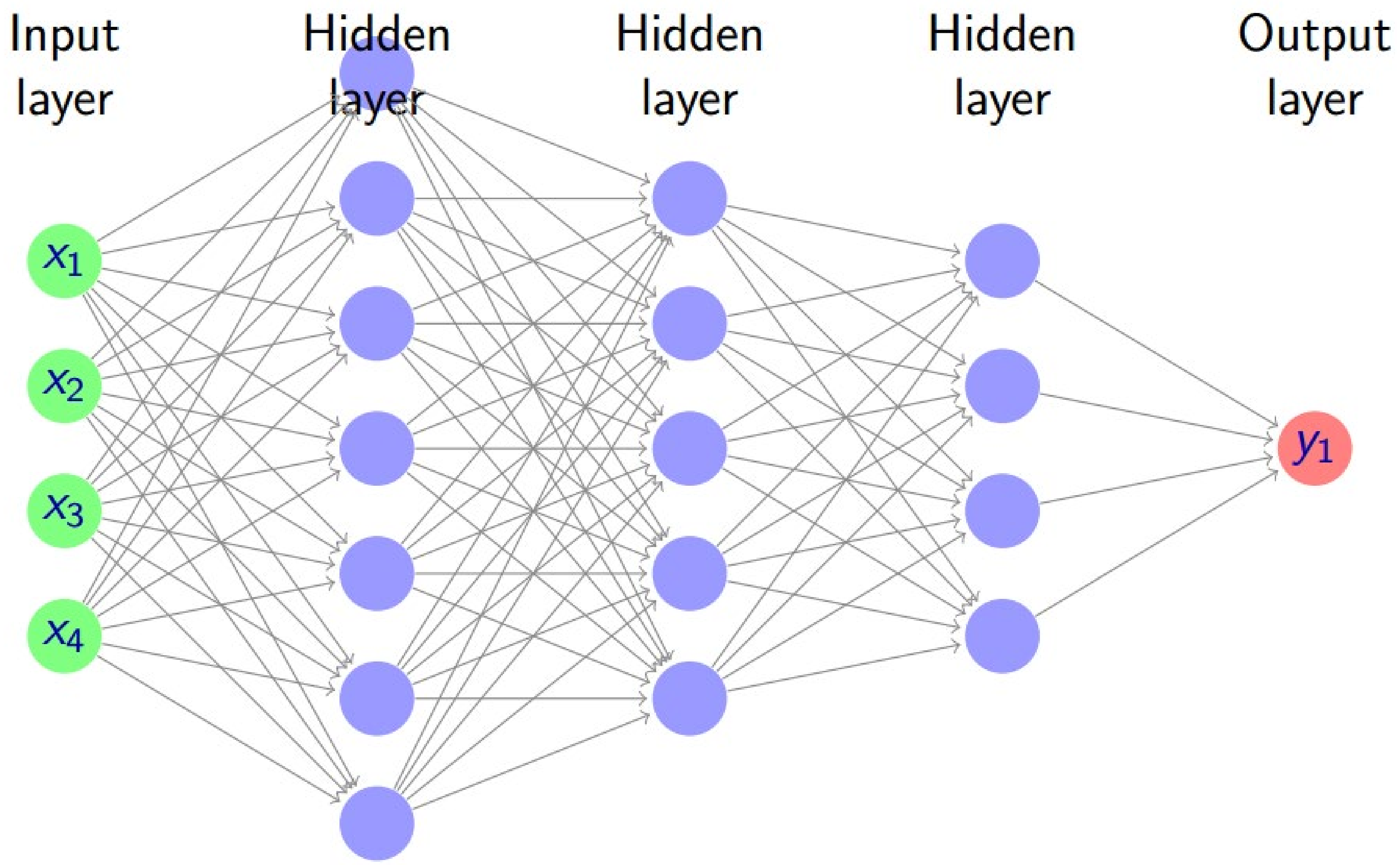
Lecture 2

Perceptron,Feedforward neural networks, Backpropagation

Feedforward Neural Network



Feedforward Deep Networks

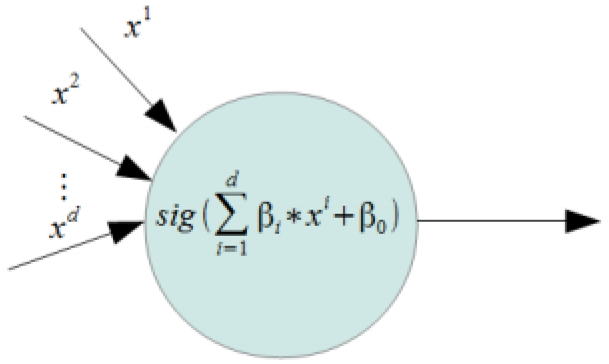


Feedforward Deep Networks

- Feedforward deep networks, a.k.a. multilayer perceptrons (MLPs), are parametric functions composed of several parametric functions.
- Each layer of the network defines one of these sub-functions.
- Each layer (sub-function) has multiple inputs and multiple outputs.
- Each layer composed of many units (scalar output of the layer).
- We sometimes refer to each unit as a feature.
- Each unit is usually a simple transformation of its input.
- The entire network can be very complex.

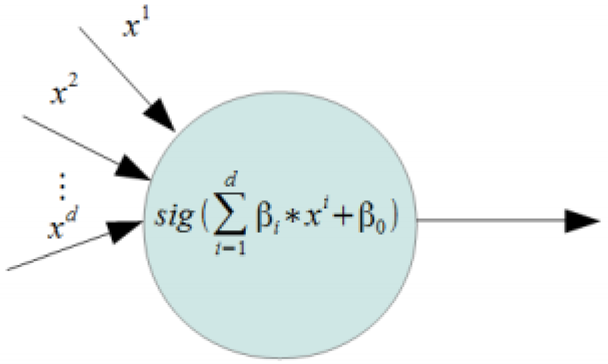
Perceptron

- The perceptron is the building block for neural networks.
- It was invented by Rosenblatt in 1957 at Cornell Labs, and first mentioned in the paper ‘The Perceptron – a perceiving and recognizing automaton’.
- Perceptron computes a linear combination of factor of input and returns the sign.



Simple perceptron

Perceptron



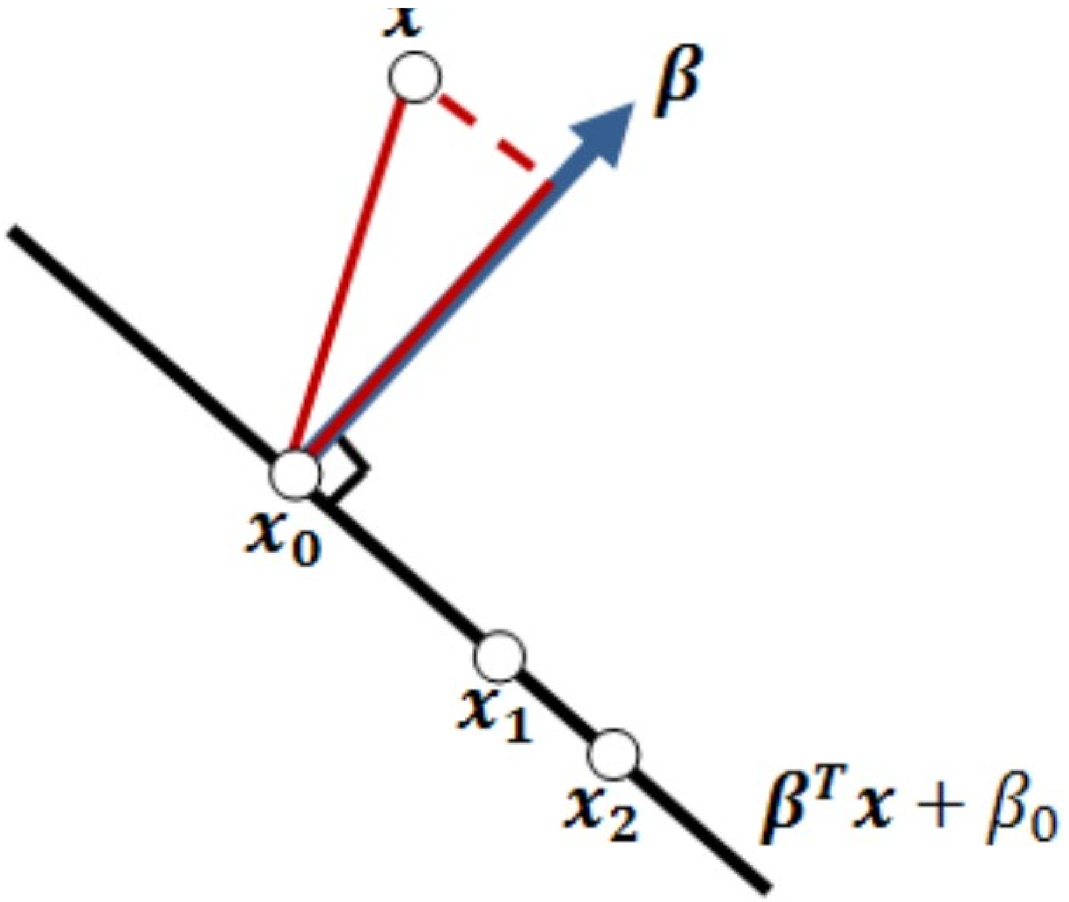
Simple perceptron

x^i is the *i*-th feature of a sample and β_i is the *i*-th weight. β_0 is defined as the bias. The bias alters the position of the decision boundary between the 2 classes. From a geometrical point of view, Perceptron assigns label "1" to elements on one side of $\beta^T x + \beta_0$ and label "-1" to elements on the other side

Perceptron

- define a cost function, $\phi(\boldsymbol{\beta}, \beta_0)$, as a summation of the distance between all misclassified points and the hyper-plane, or the decision boundary.

- To minimize this cost function, we need to estimate $\boldsymbol{\beta}, \boldsymbol{\beta_0}$.
  $\min_{\beta,\beta_0} \phi(\boldsymbol{\beta}, \beta_0) = \{$distance of all misclassified points$\}$



Distance between the point and the decision boundary hyperplane (black line).

1) A hyper-plane L can be defined as

$L = \{x : f(x) = \beta^T x + \beta_0 = 0\},$

For any two arbitrary points x_1 and x_2 on L , we have

$\beta^T x_1 + \beta_0 = 0$,

$\beta^T x_2 + \beta_0 = 0$,

such that

$\beta^T (x_1 - x_2) = 0$.

Therefore, β is orthogonal to the hyper-plane and it is the normal vector.

2) For any point x_0 in L,

$\beta^T x_0 + \beta_0 = 0$, which means $\beta^T x_0 = -\beta_0$.

3) We set $\beta^* = \frac{\beta}{||\beta||}$ as the unit normal vector of the hyper-plane L . For simplicity we call β^* norm vector. The distance of point x to L is given by

$$\beta^{*T}(x - x_0) = \beta^{*T}x - \beta^{*T}x_0 = \frac{\beta^T x}{||\beta||} + \frac{\beta_0}{||\beta||} = \frac{(\beta^T x + \beta_0)}{||\beta||}$$

Where x_0 is any point on L . Hence, $\beta^T x + \beta_0$ is proportional to the distance of the point x to the hyper-plane L .

4) The distance from a misclassified data point x_i to the hyper-plane L is

$$d_i = -y_i(\boldsymbol{\beta}^T x_i + \beta_0)$$

where y_i is a target value, such that $y_i = 1$ if $\boldsymbol{\beta}^T x_i + \beta_0 < 0$, $y_i = -1$ if $\boldsymbol{\beta}^T x_i + \beta_0 > 0$

Since we need to find the distance from the hyperplane to the *misclassified* data points, we need to add a negative sign in front. When the data point is misclassified, $\boldsymbol{\beta}^T x_i + \beta_0$ will produce an opposite sign of y_i . Since we need a positive sign for distance, we add a negative sign.

Learning Perceptron

The gradient descent is an optimization method that finds the minimum of an objective function by incrementally updating its parameters in the negative direction of the derivative of this function. That is, it finds the steepest slope in the D-dimensional space at a given point, and descends down in the direction of the negative slope. Note that unless the error function is convex, it is possible to get stuck in a local minima. In our case, the objective function to be minimized is classification error and the parameters of this function are the weights associated with the inputs, β

The gradient descent algorithm updates the weights as follows:

$\beta^{\text{new}} \leftarrow \beta^{\text{old}} - \rho \frac{\partial Err}{\partial \beta}$

ρ is called the *learning rate*.
The Learning Rate ρ is positively related to the step size of convergence of $\min \phi(\boldsymbol{\beta}, \beta_0)$.
i.e. the larger ρ is, the larger the step size is.
Typically, $\rho \in [0.1, 0.3]$.

The classification error is defined as the distance of misclassified observations to the decision boundary:

To minimize the cost function $\phi(\boldsymbol{\beta}, \beta_0) = -\sum_{i \in M} y_i(\boldsymbol{\beta}^T x_i + \beta_0)$ where

$M = \{\text{all points that are misclassified}\}$

$\dfrac{\partial \phi}{\partial \boldsymbol{\beta}} = -\sum_{i \in M} y_i x_i$ and $\dfrac{\partial \phi}{\partial \beta_0} = -\sum_{i \in M} y_i$

Therefore, the gradient is

$$\nabla D(\beta, \beta_0) = \begin{pmatrix} -\sum_{i \in M} y_i x_i \\ -\sum_{i \in M} y_i \end{pmatrix}$$

Using the gradient descent algorithm to solve these two equations, we have

$$\begin{pmatrix} \boldsymbol{\beta}^{\text{new}} \\ \beta_0^{\text{new}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\beta}^{\text{old}} \\ \beta_0^{\text{old}} \end{pmatrix} + \rho \begin{pmatrix} y_i x_i \\ y_i \end{pmatrix}$$

If the data is linearly-separable, the solution is theoretically guaranteed to converge to a separating hyperplane in a finite number of iterations.

In this situation the number of iterations depends on the learning rate and the margin. However, if the data is not linearly separable there is no guarantee that the algorithm converges.

Features

- A Perceptron can only discriminate between two classes at a time.
- When data is (linearly) separable, there are an infinite number of solutions depending on the starting point.
- Even though convergence to a solution is guaranteed if the solution exists, the finite number of steps until convergence can be very large.
- The smaller the gap between the two classes, the longer the time of convergence.

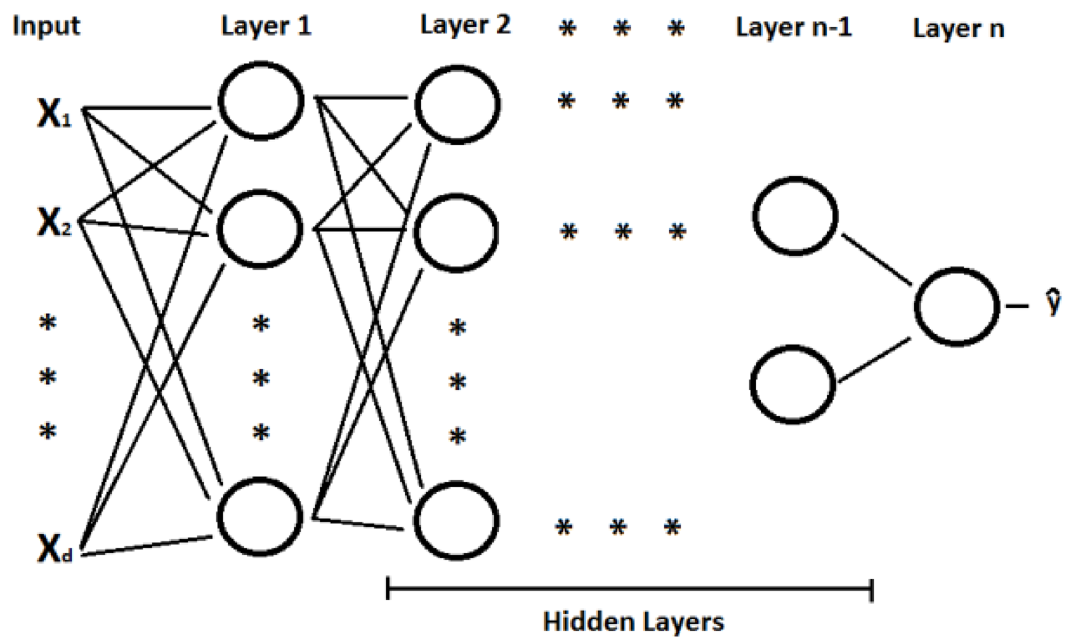
- When the data is not separable, the algorithm will not converge (it should be stopped after N steps).
- A learning rate that is too high will make the perceptron periodically oscillate around the solution unless additional steps are taken.
- The L.S compute a linear combination of feature of input and return the sign.
- This were called Perceptron in the engineering literate in late 1950.
- Learning rate affects the accuracy of the solution and the number of iterations directly.

Separability and convergence

The training set D is said to be linearly separable if there exists a positive constant γ and a weight vector β such that $(\beta^T x_i + \beta_0) y_i > \gamma$ for all $1 < i < n$. That is, if we say that β is the weight vector of Perceptron and y_i is the true label of x_i , then the signed distance of the x_i from β is greater than a positive constant γ for any $(x_i, y_i) \in D$.

Feedforward Neural Network

- A neural network is a multistate regression model which is typically represented by a network diagram.



Feed Forward Neural Network

Feedforward Neural Network

- For regression, typically k $=$ 1 (the number of nodes in the last layer), there is only one output unit y_1 at the end.
- For c-class classification, there are typically c units at the end with the c^{th} unit modelling the probability of class c, each y_c is coded as 0-1 variable for the cth class.

Feedforward Neural Network

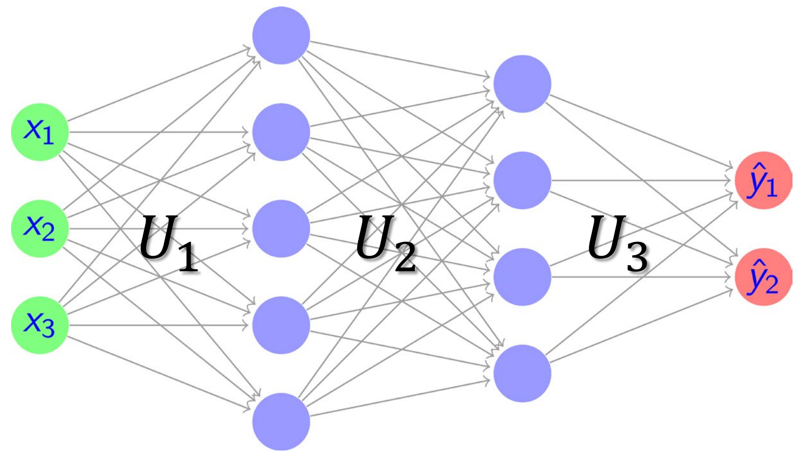
- Consider a feed forward neural network

$$U_1 = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ u_{21} & u_{22} & u_{23} & u_{24} & u_{25} \\ u_{31} & u_{32} & u_{33} & u_{34} & u_{35} \end{bmatrix}_{3\times 5}$$

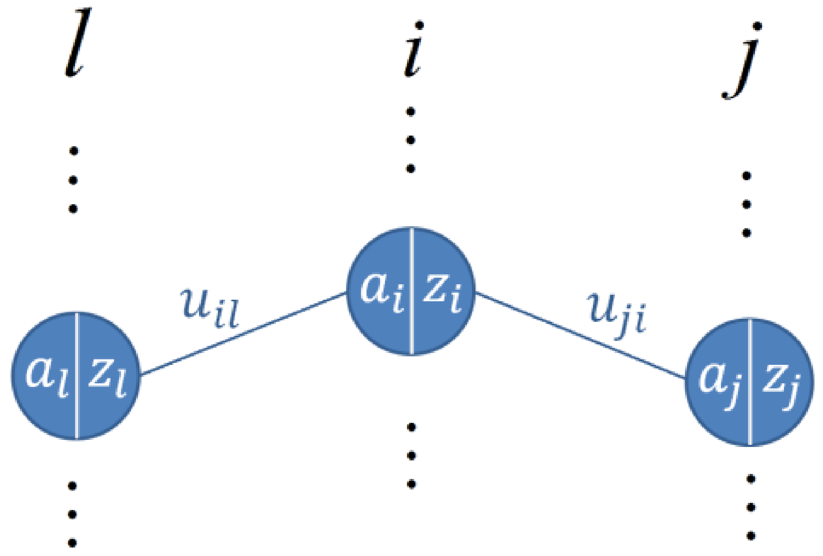
$$U_2 = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ u_{21} & u_{22} & u_{23} & u_{24} \\ u_{31} & u_{32} & u_{33} & u_{34} \\ u_{41} & u_{42} & u_{43} & u_{44} \\ u_{51} & u_{52} & u_{53} & u_{54} \end{bmatrix}_{5\times 4}$$

$$U_3 = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \\ u_{31} & u_{32} \end{bmatrix}_{4\times 2}$$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



Backpropagation

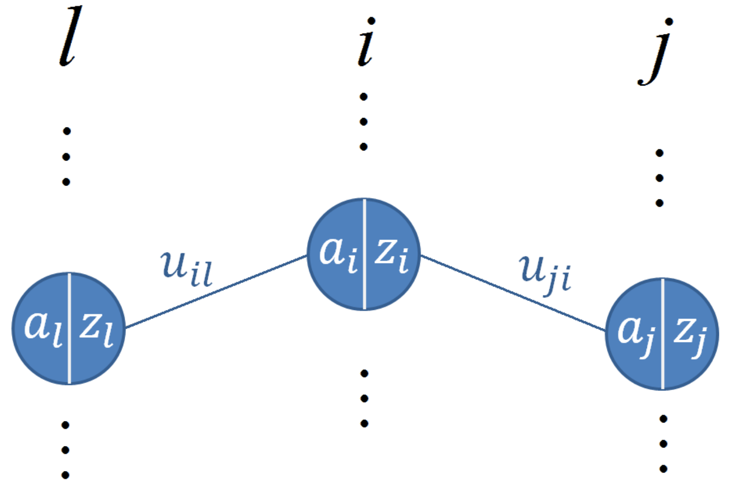


Nodes from three hidden layers within the neural network. Each node is divided into the weighted sum of the inputs and the output of the activation function.

$$a_i = \sum_l z_l u_{il}$$
$$z_i = \sigma(a_i)$$
$$\sigma(a) = \frac{1}{1+e^{-a}}$$

- Nodes from three hidden layers within the neural network are considered for the backpropagation algorithm.

- Each node has been divided into the weighted sum of the inputs a and the output of the activation function z. The weights between the nodes are denoted by u.

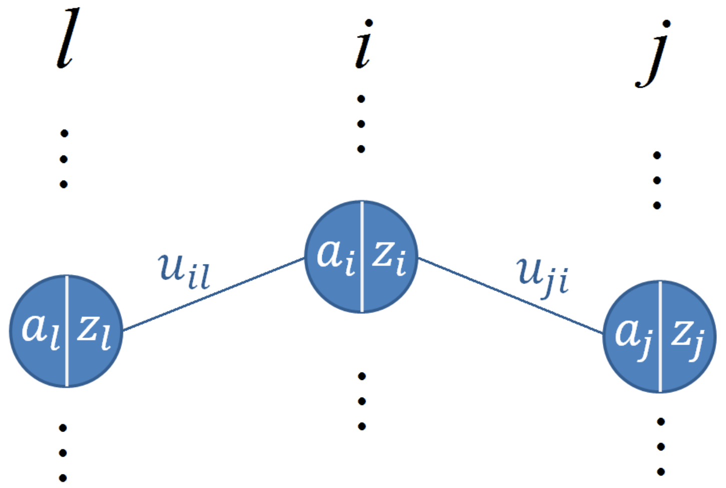


$$a_i = \sum_l z_l u_{il}$$

$$z_i = \sigma(a_i)$$

- Take the derivative w.r.t weight u_{il}

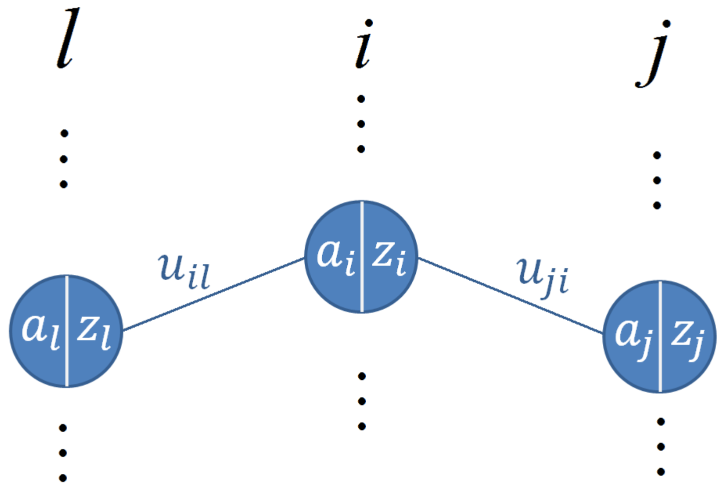
$$\frac{\partial \left|y-\hat{y}\right|^2}{\partial u_{il}} = \frac{\partial \left|y-\hat{y}\right|^2}{\partial a_i} \cdot \frac{\partial a_i}{\partial u_{il}}$$



- Take the derivative w.r.t weight u_{il}

Chain rule

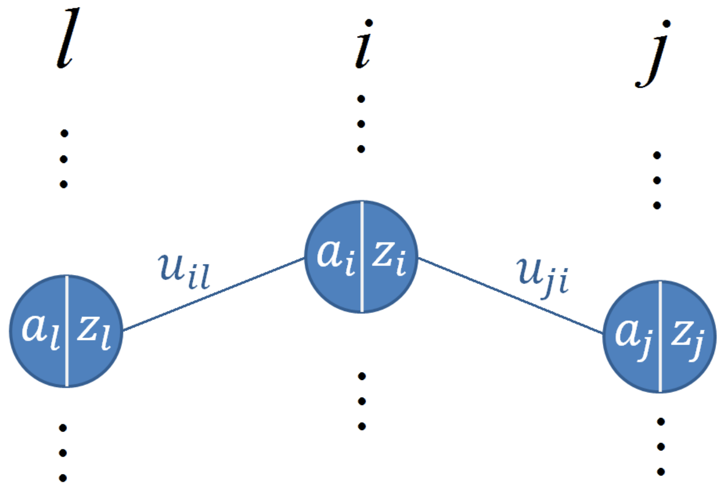
$$\frac{\partial |y-\hat{y}|^2}{\partial u_{il}} = \frac{\partial |y-\hat{y}|^2}{\partial a_i} \cdot \frac{\partial a_i}{\partial u_{il}}$$



- Take the derivative w.r.t weight u_{il}

Chain rule

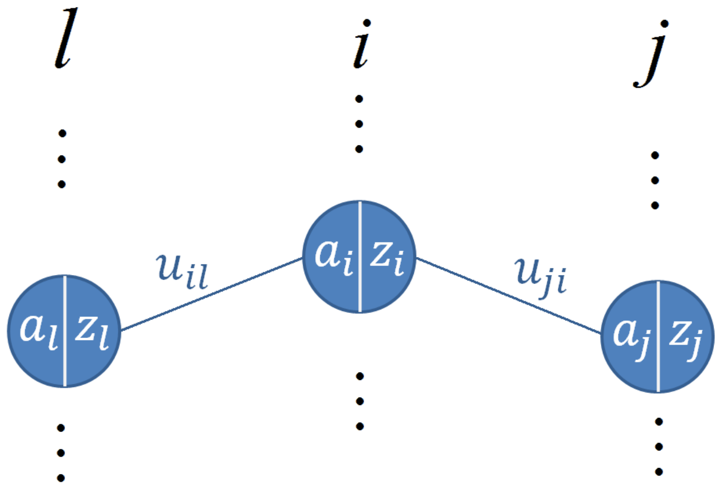
$$\frac{\partial |y-\hat{y}|^2}{\partial u_{il}} = \frac{\partial |y-\hat{y}|^2}{\partial a_i} \cdot \underbrace{\frac{\partial a_i}{\partial u_{il}}}_{z_l}$$



- Take the derivative w.r.t weight u_{il}

Chain rule

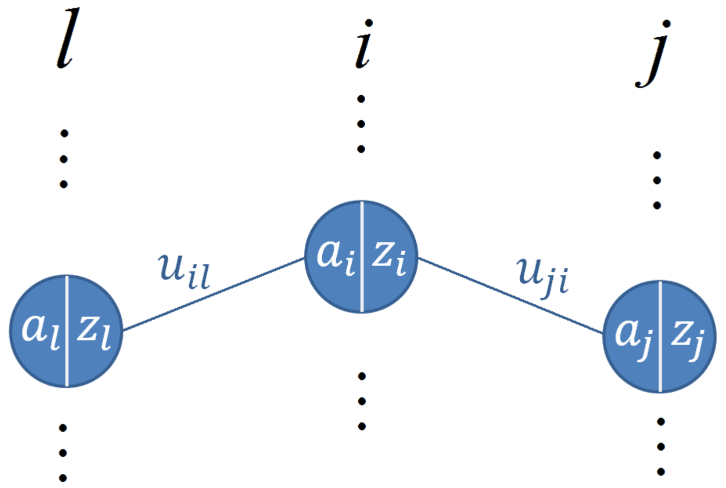
$$\frac{\partial |y-\hat{y}|^2}{\partial u_{il}} = \underbrace{\frac{\partial |y-\hat{y}|^2}{\partial a_i}}_{\delta_i} \cdot \underbrace{\frac{\partial a_i}{\partial u_{il}}}_{z_l}$$



- Take the derivative w.r.t weight u_{il}

Chain rule

$$\frac{\partial |y-\hat{y}|^2}{\partial u_{il}} = \underbrace{\frac{\partial |y-\hat{y}|^2}{\partial a_i}}_{\delta_i} \cdot \underbrace{\frac{\partial a_i}{\partial u_{il}}}_{z_l}$$

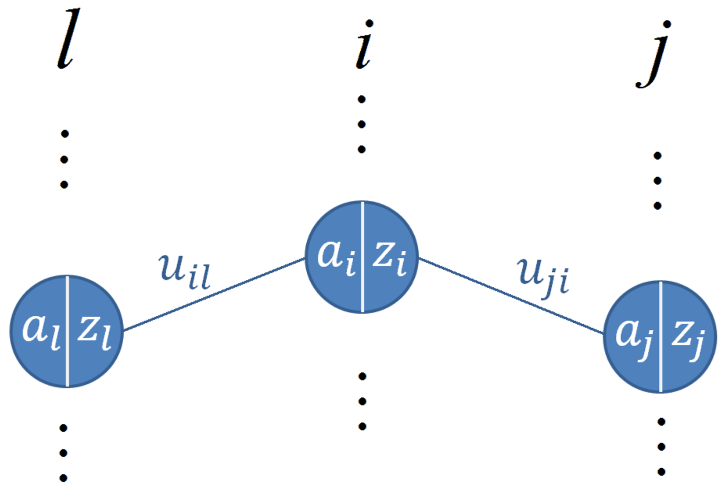


$$\frac{\partial |y-\hat{y}|^2}{\partial u_{il}} = \delta_i \cdot z_l$$

where $\delta_i = \dfrac{\partial |y-\hat{y}|^2}{\partial a_i}$

- whereas δ_i is unknown but can be expressed as a recursive definition in terms of δ_j.

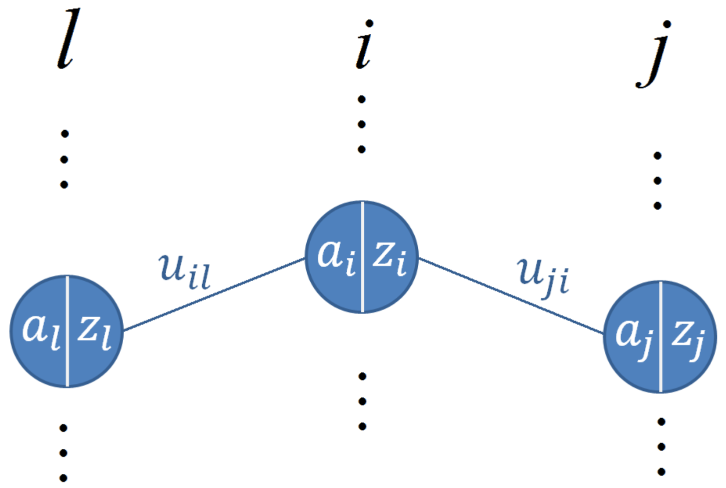
$$\delta_i = \frac{\partial\,|y - \hat{y}|^2}{\partial a_i} = \sum_j \frac{\partial\,|y - \hat{y}|^2}{\partial a_j} \cdot \frac{\partial a_j}{\partial a_i}$$



- whereas δ_i is unknown but can be expressed as a recursive definition in terms of δ_j.

Chain rule

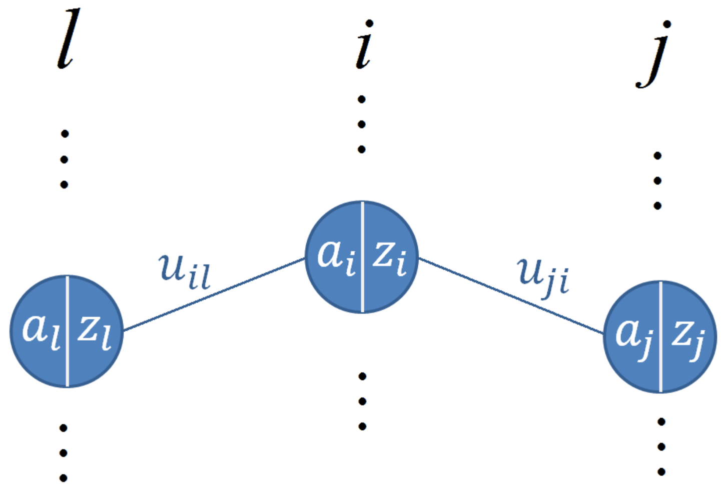
$$\delta_i = \frac{\partial\, |y - \hat{y}|^2}{\partial a_i} = \sum_j \frac{\partial\, |y - \hat{y}|^2}{\partial a_j} \cdot \frac{\partial a_j}{\partial a_i}$$



- whereas δ_i is unknown but can be expressed as a recursive definition in terms of δ_j.

Chain rule

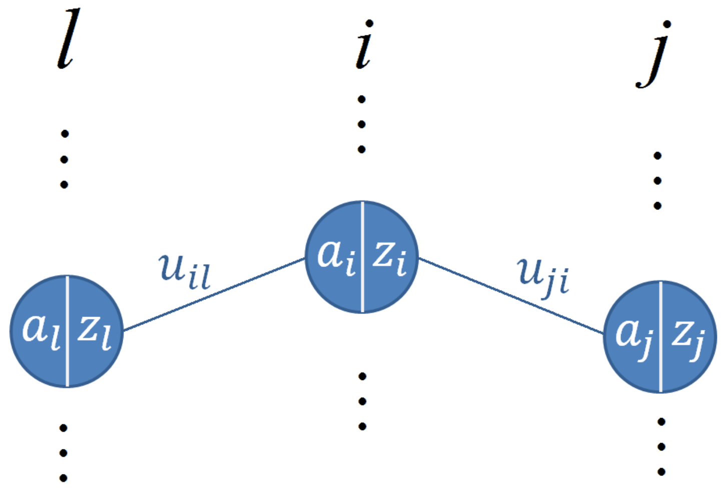
$$\delta_i = \frac{\partial\,|y-\hat{y}|^2}{\partial a_i} = \sum_j \underbrace{\frac{\partial\,|y-\hat{y}|^2}{\partial a_j}}_{\delta_j} \cdot \frac{\partial a_j}{\partial a_i}$$



- whereas δ_i is unknown but can be expressed as a recursive definition in terms of δ_j.

Chain rule

$$\delta_i = \frac{\partial |y-\hat{y}|^2}{\partial a_i} = \sum_j \underbrace{\frac{\partial |y-\hat{y}|^2}{\partial a_j}}_{\delta_j} \cdot \underbrace{\frac{\partial a_j}{\partial a_i}}_{\frac{\partial a_j}{\partial a_i} = \frac{\partial a_j}{\partial z_i} \cdot \frac{\partial z_i}{\partial a_i}}$$

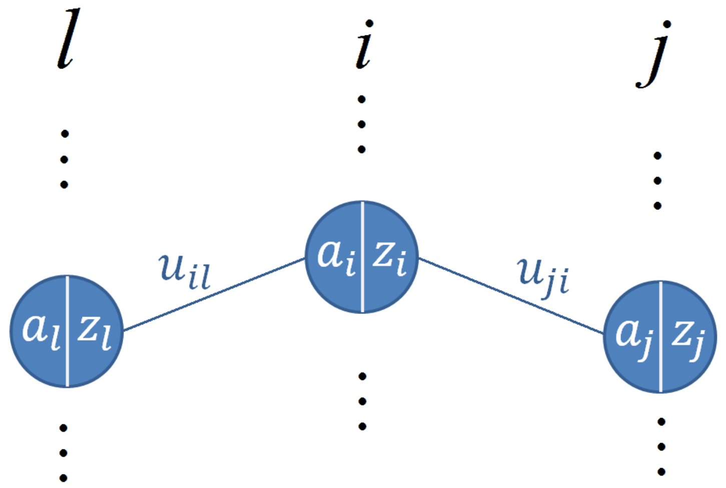


- whereas δ_i is unknown but can be expressed as a recursive definition in terms of δ_j.

Chain rule

$$\delta_i = \frac{\partial |y-\hat{y}|^2}{\partial a_i} = \sum_j \underbrace{\frac{\partial |y-\hat{y}|^2}{\partial a_j}}_{\delta_j} \cdot \underbrace{\frac{\partial a_j}{\partial a_i}}$$

$$\frac{\partial a_j}{\partial a_i} = \underbrace{\frac{\partial a_j}{\partial z_i}}_{u_{ji}} \cdot \frac{\partial z_i}{\partial a_i}$$

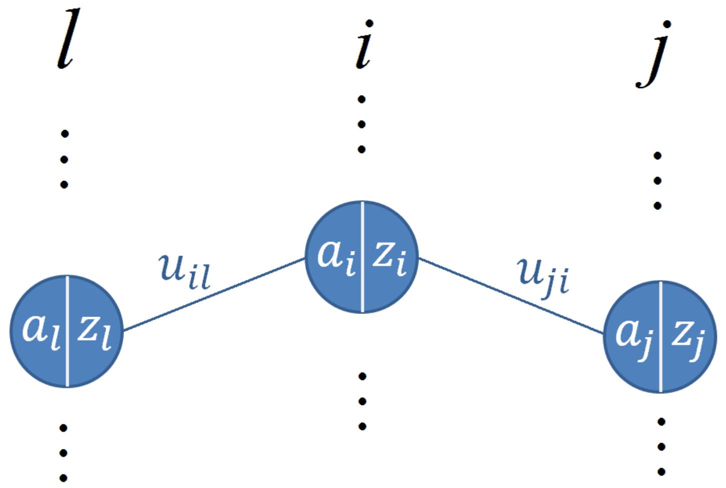


- whereas δ_i is unknown but can be expressed as a recursive definition in terms of δ_j.

Chain rule

$$\delta_i = \frac{\partial |y-\hat{y}|^2}{\partial a_i} = \sum_j \underbrace{\frac{\partial |y-\hat{y}|^2}{\partial a_j}}_{\delta_j} \cdot \underbrace{\frac{\partial a_j}{\partial a_i}}$$

$$\frac{\partial a_j}{\partial a_i} = \underbrace{\frac{\partial a_j}{\partial z_i}}_{u_{ji}} \cdot \underbrace{\frac{\partial z_i}{\partial a_i}}_{\sigma'(a_i)}$$



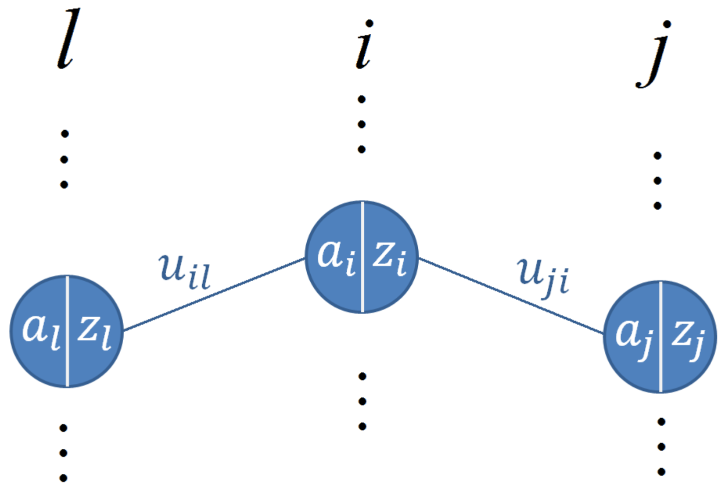
- whereas δ_i is unknown but can be expressed as a recursive definition in terms of δ_j.

Chain rule

$$\delta_i = \frac{\partial |y-\hat{y}|^2}{\partial a_i} = \sum_j \underbrace{\frac{\partial |y-\hat{y}|^2}{\partial a_j}}_{\delta_j} \cdot \underbrace{\frac{\partial a_j}{\partial a_i}}$$

$$\delta_i = \sum_j \delta_j \cdot \frac{\partial a_j}{\partial z_i} \cdot \frac{\partial z_i}{\partial a_i}$$

$$\frac{\partial a_j}{\partial a_i} = \underbrace{\frac{\partial a_j}{\partial z_i}}_{u_{ji}} \cdot \underbrace{\frac{\partial z_i}{\partial a_i}}_{\sigma'(a_i)}$$



- whereas δ_i is unknown but can be expressed as a recursive definition in terms of δ_j.

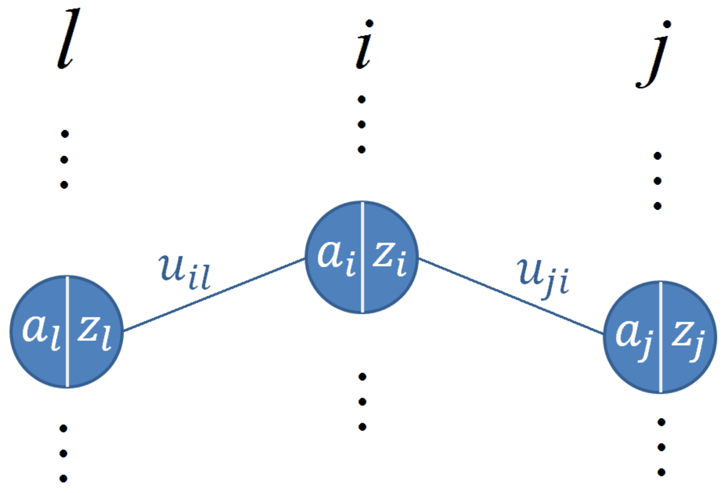
Chain rule

$$\delta_i = \frac{\partial |y-\hat{y}|^2}{\partial a_i} = \sum_j \underbrace{\frac{\partial |y-\hat{y}|^2}{\partial a_j}}_{\delta_j} \cdot \underbrace{\frac{\partial a_j}{\partial a_i}}$$

$$\frac{\partial a_j}{\partial a_i} = \underbrace{\frac{\partial a_j}{\partial z_i}}_{u_{ji}} \cdot \underbrace{\frac{\partial z_i}{\partial a_i}}_{\sigma'(a_i)}$$

$$\delta_i = \sum_j \delta_j \cdot \frac{\partial a_j}{\partial z_i} \cdot \frac{\partial z_i}{\partial a_i}$$

$$\delta_i = \sum_j \delta_j \cdot u_{ji} \cdot \sigma'(a_i)$$



- whereas δ_i is unknown but can be expressed as a recursive definition in terms of δ_j.

Chain rule

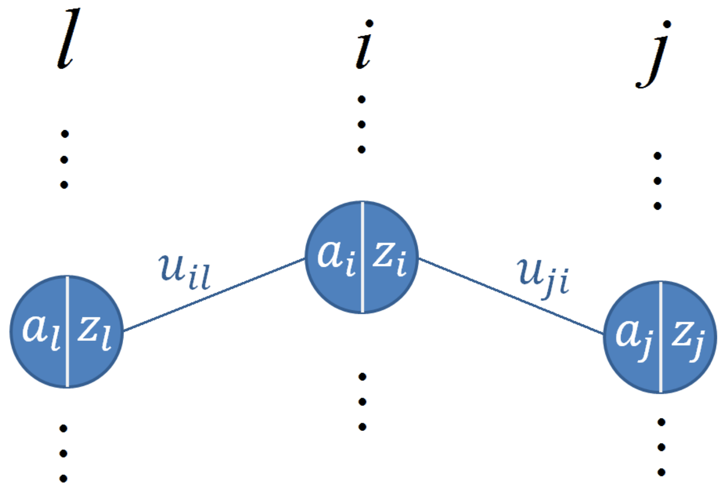
$$\delta_i = \frac{\partial |y - \hat{y}|^2}{\partial a_i} = \sum_j \underbrace{\frac{\partial |y - \hat{y}|^2}{\partial a_j}}_{\delta_j} \cdot \underbrace{\frac{\partial a_j}{\partial a_i}}$$

$$\frac{\partial a_j}{\partial a_i} = \underbrace{\frac{\partial a_j}{\partial z_i}}_{u_{ji}} \cdot \underbrace{\frac{\partial z_i}{\partial a_i}}_{\sigma'(a_i)}$$

$$\delta_i = \sum_j \delta_j \cdot \frac{\partial a_j}{\partial z_i} \cdot \frac{\partial z_i}{\partial a_i}$$

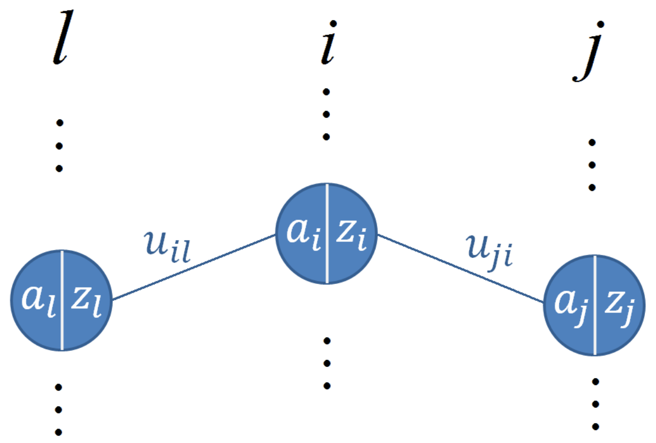
$$\delta_i = \sum_j \delta_j \cdot u_{ji} \cdot \sigma'(a_i)$$

$$\delta_i = \sigma'(a_i) \sum_j \delta_j \cdot u_{ji}$$



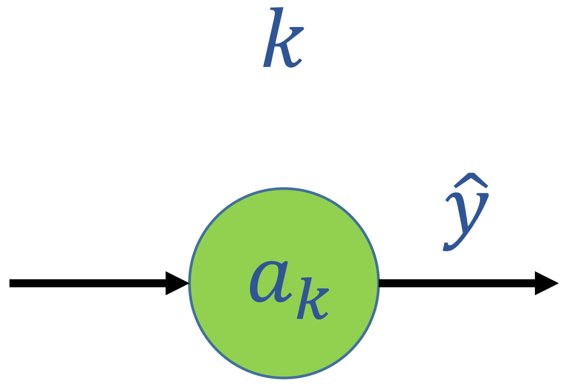
- The recursive definition of δ_i can be considered as a cost function at layer i for achieving the original goal of optimizing the weights to minimize $\|y - \hat{y}\|^2$

$$\delta_i = \sigma'(a_i) \sum_j \delta_j \cdot u_{ji}$$



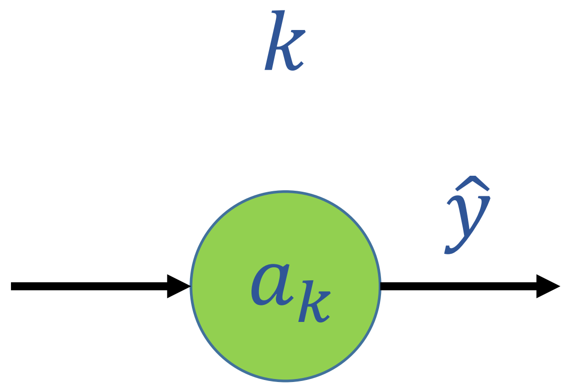
Now consider

$$\delta_k = \frac{\partial \left\| y - \hat{y} \right\|^2}{\partial a_k}$$



$$\delta_k = \frac{\partial \left\|y - \hat{y}\right\|^2}{\partial a_k}$$

- Where $a_k = \hat{y}$ because an activation function is not applied in the output layer



$$\delta_k = \frac{\partial \left\|y - \hat{y}\right\|^2}{\partial \hat{y}} = -2\left\|y - \hat{y}\right\|$$

- Since y is known and $\hat{y}$ can be computed for each data point.

- assuming small, random, initial values for the weights of the neural network in the beginning.

- Therefore the δ values for the layer before the output layer can be computed using δ_k and then the δ values for the layer before the layer before the output layer can be computed and so on.

$$\delta_i^{(k)} = -2|y_i - \hat{y}_i|$$

$$\Downarrow$$

$$\delta_i^{(k-1)} = \sigma'(a_i^{(k-1)}) \sum_j \delta_j^{(k)} \cdot u_{ij}^{(k-1)}$$

$$\Downarrow$$

$$\delta_i^{(k-2)} = \sigma'(a_i^{(k-2)}) \sum_j \delta_j^{(k-1)} \cdot u_{ij}^{(k-2)}$$

$$\Downarrow$$

$$\vdots$$

- Once all δ values are known, the errors due to each of the weights u will be known and techniques like gradient descent can be used to optimize the weights.

$$u_{il}^{new} \leftarrow u_{il}^{old} - \rho \frac{\partial \left\| y - \hat{y} \right\|^2}{\partial u_{il}}$$

Backpropagation

Backpropagation procedure is done using the following steps:

- First arbitrarily choose some random weights (preferably close to zero) for your network.

- Apply x to the FFNN's input layer, and calculate the outputs of all input neurons.

- Propagate the outputs of each hidden layer forward, one hidden layer at a time, and calculate the outputs of all hidden neurons.

- Once x reaches the output layer, calculate the output(s) of all output neuron(s) given the outputs of the previous hidden layer.

- At the output layer, compute $\delta_k = -2(y_k - \hat{y}_k)$ for each output neuron(s).

- Compute each δ_i , starting from $i = k - 1$ all the way to the first hidden layer, where $\delta_i = \sigma'(a_i) \sum_j \delta_j \cdot u_{ji}$.

- Compute $\dfrac{\partial \|y - \hat{y}\|^2}{\partial u_{il}} = \delta_i z_l$ for all weights u_{il} .

- Then update $u_{il}^{\mathrm{new}} \leftarrow u_{il}^{\mathrm{old}} - \rho \cdot \dfrac{\partial \|y - \hat{y}\|^2}{\partial u_{il}}$ for all weights u_{il} .

- Continue for next data points and iterate on the training set until weights converge.

Epochs

It is common to cycle through the all of the data points multiple times in order to reach convergence. An epoch represents one cycle in which you feed all of your datapoints through the neural network. It is good practice to randomized the order you feed the points to the neural network within each epoch; this can prevent your weights changing in cycles. The number of epochs required for convergence depends greatly on the learning rate & convergence requirements used.