Lecture 16

for any ~~point~~ mis-classified point x_i, with true label y_i, the distance

✓ $d_i = -y_i (\beta^T x_i + \beta_0)$

Say, a point x_i with true label is y_i

prediction of perceptron. $\beta^T x_i + \beta_0 > 0$ $+1$

< 0 -1

Let x_i is mis-classified.

$$d_i = -y_i(\beta^T x_i + \beta_0) > 0$$

To learn the parameters β & β_0, we will introduce an optimization method called gradient descent.

G.D

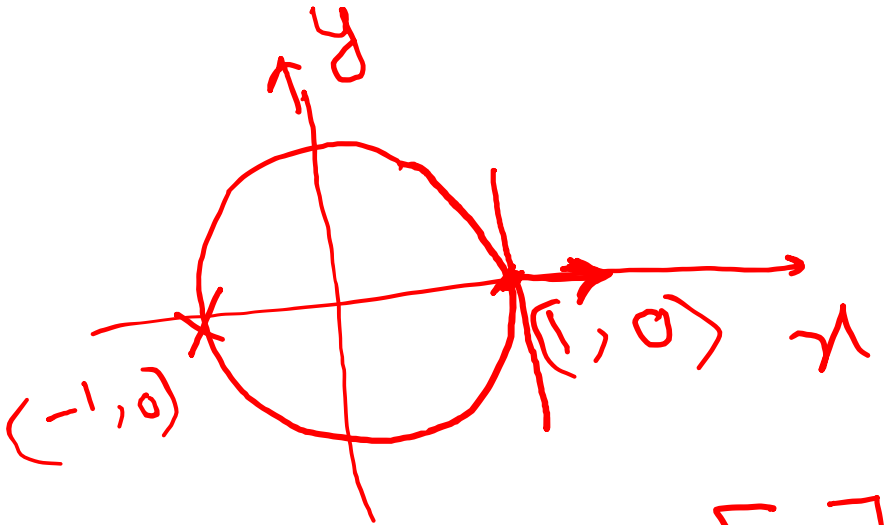
$$\min_{x,y} x^2 + y^2 = f(x,y)$$

$$\frac{\partial f(x,y)}{\partial x} = 2x, \quad \frac{\partial f}{\partial y} = 2y$$

$$\nabla f = \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix} = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

$x^{new} \leftarrow x_{old} - \eta \frac{\partial f}{\partial x}$ 'η' → learning rate or step-size

$y^{new} \leftarrow y_{old} - \eta \frac{\partial f}{\partial y}$



$\nabla f = 2\begin{bmatrix} x \\ y \end{bmatrix} = 2\begin{bmatrix} 1 \\ 0 \end{bmatrix}$

$\eta = 1$

$x^{new} \leftarrow 1 - (1) \cdot 2 \cdot 1 = -1$

$y^{new} \leftarrow 0 - (1) \cdot 2 \cdot (0) = 0$

$x^{new} \leftarrow -1 - (1) \cdot 2(-1) = 1$

$y^{new} \leftarrow 0$

large η, will oscillate & won't reach the desired solution.

$\eta \rightarrow$ small, 10^{-1}, 10^{-2}, 10^{-5}

$\eta \propto \frac{1}{\text{iterations}}$.

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

$$\beta^{new} \leftarrow \beta_{old} - \eta \frac{\partial d_i}{\partial \beta} = \beta_{old} - \eta(-y_i x_i)$$

$$\beta_0^{new} \leftarrow \beta_{0-old} - \eta(-y_i)$$

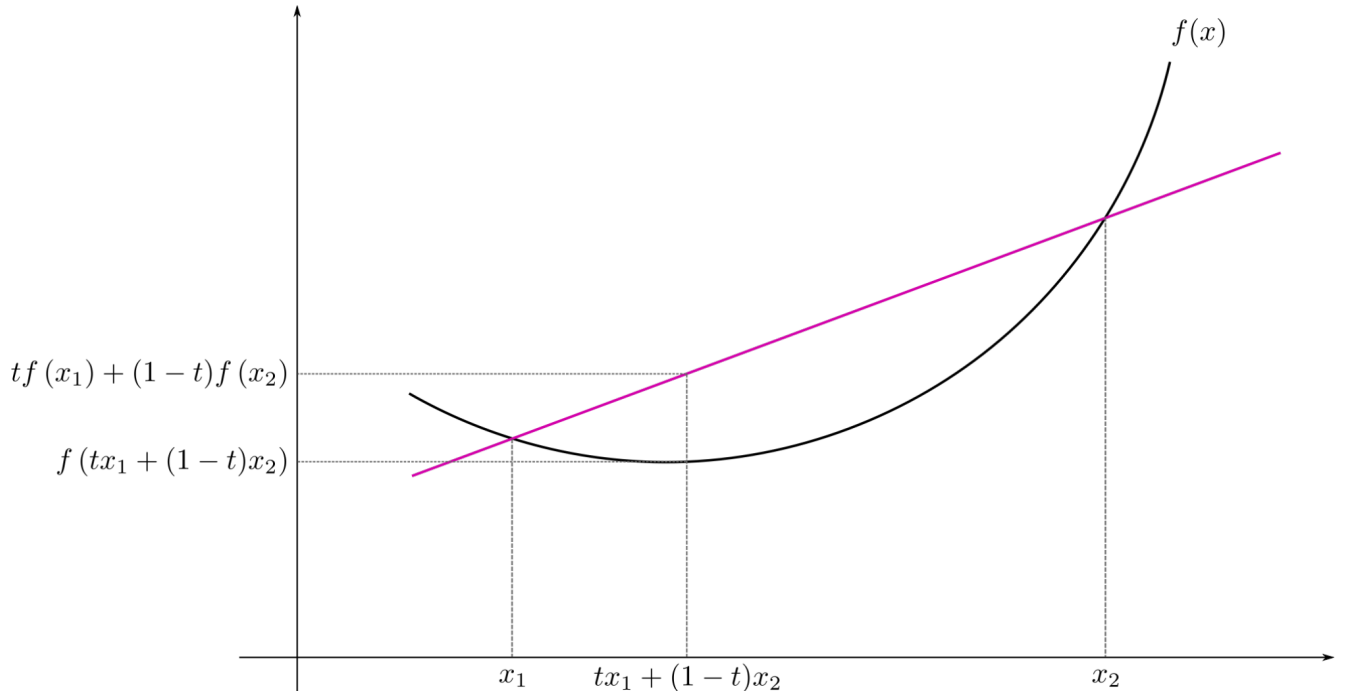
In the remainder of this section, assume $f : \mathbb{R}^d \to \mathbb{R}$ unless otherwise noted. We'll start with the definitions and then give some results.

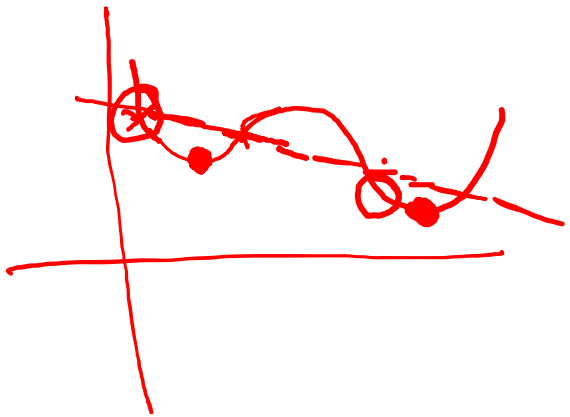
A function f is **convex** if

$$f(t\mathbf{x} + (1-t)\mathbf{y}) \leq tf(\mathbf{x}) + (1-t)f(\mathbf{y})$$

for all $\mathbf{x}, \mathbf{y} \in \text{dom}\, f$ and all $t \in [0,1]$.

If the inequality holds strictly (i.e. $<$ rather than $\leq$) for all $t \in (0,1)$ and $\mathbf{x} \neq \mathbf{y}$, then we say that f is **strictly convex**.





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

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

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

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.

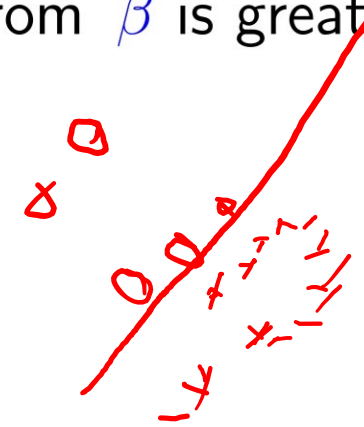
Features

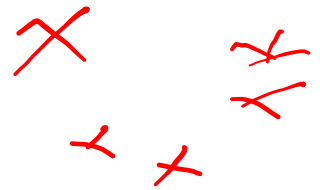
- A Perceptron can only discriminate between two classes at a time.
- When data is (linearly) separable, there are an infinite number of solutions depending on the starting point.
- Even though convergence to a solution is guaranteed if the solution exists, the finite number of steps until convergence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

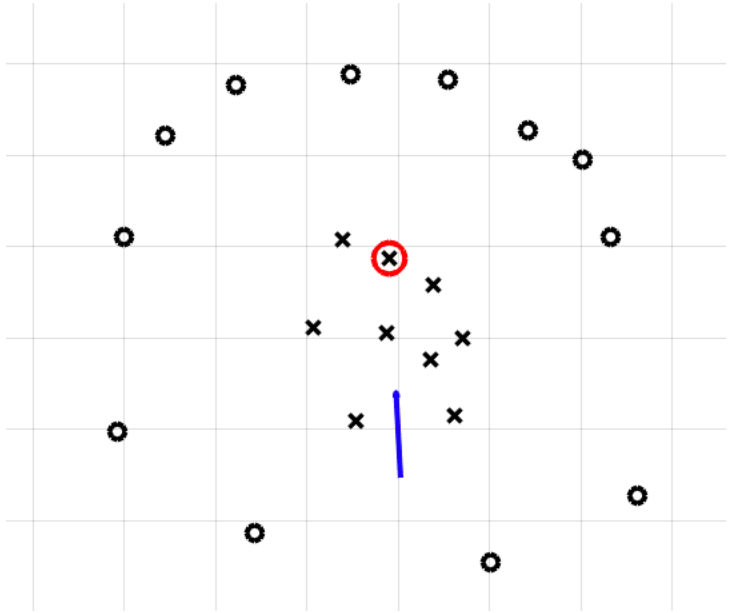
Separability and convergence

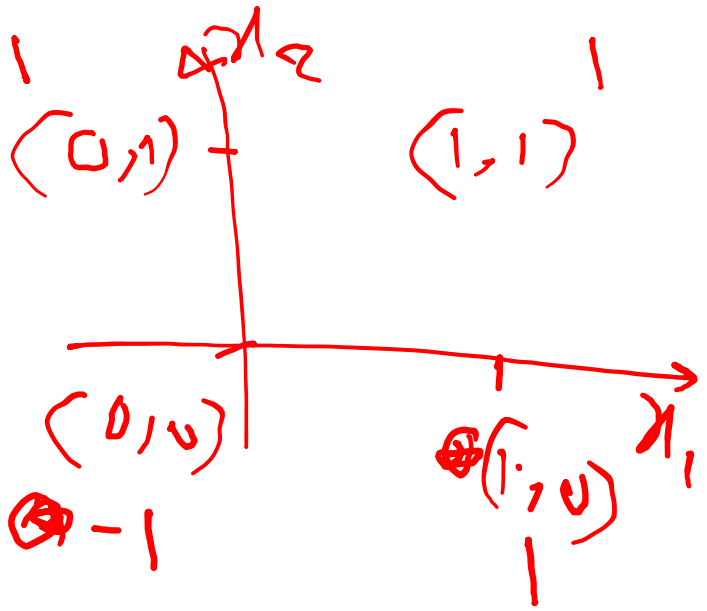
The training set D is said to be linearly separable if there exists a positive constant γ and a weight vector β such that $(\beta^T x_i + \beta_0) y_i > \gamma$ for all $1 < i < n$. That is, if we say that β is the weight vector of Perceptron and y_i is the true label of x_i, then the signed distance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





Will perceptron work for





OR/AND/XOR

Separability and convergence

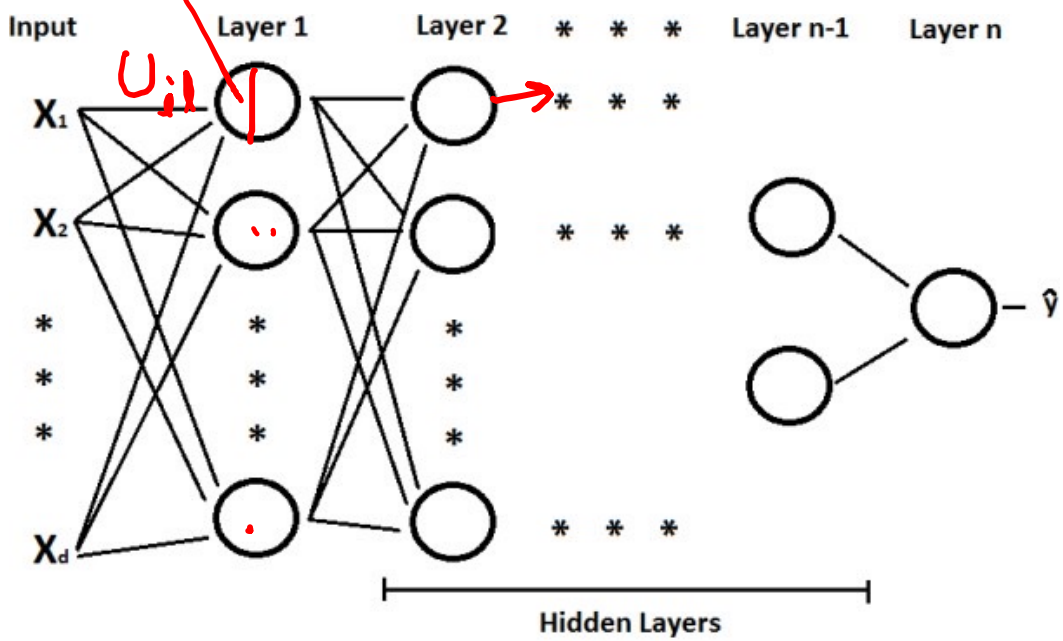
Novikoff (1962) proved that the perceptron algorithm converges after a finite number of iterations if the data set is linearly separable. The idea of the proof is that the weight vector is always adjusted by a bounded amount in a direction that it has a negative dot product with, and thus can be bounded above by $O(\sqrt{t})$ where t is the number of changes to the weight vector. But it can also be bounded below by $O(t)$ because if there exists an (unknown) satisfactory weight vector, then every change makes progress in this (unknown) direction by a positive amount that depends only on the input vector. This can be used to show that the number t of updates to the weight vector is bounded by $(\frac{2R}{\gamma})^2$, where R is the maximum norm of an input vector.

See http://en.wikipedia.org/wiki/Perceptron for details.

$\sigma\left(\sum x_i v_{il}\right)$

Neural Network

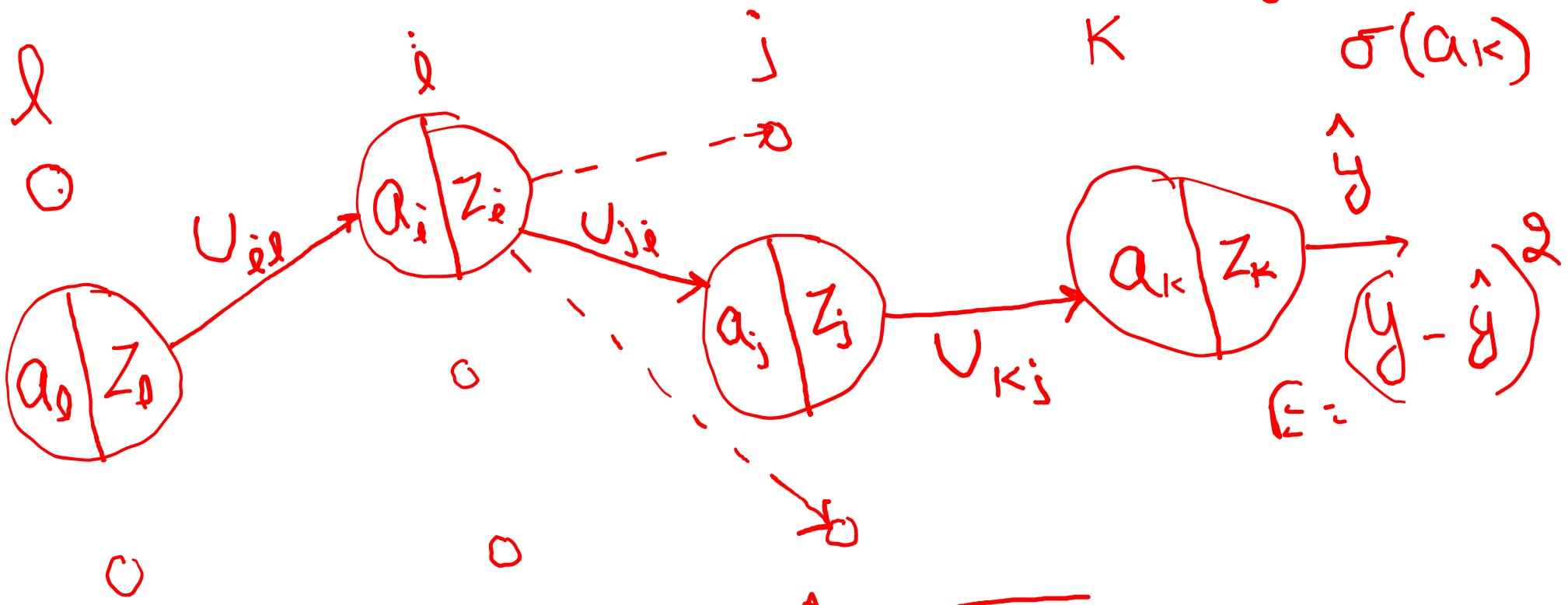
- A neural network is a multistate regression model which is typically represented by a network diagram.



Feed Forward 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.

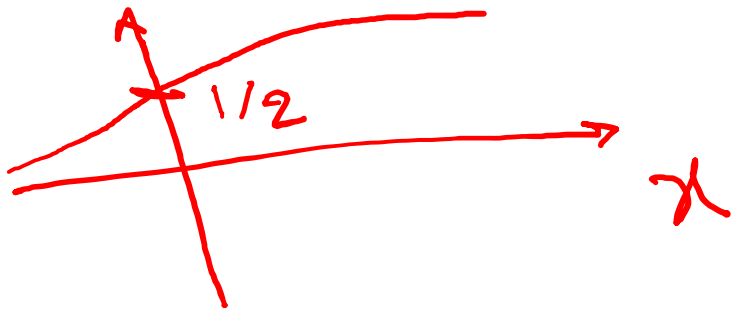
Backpropagation.



$Z_l = \sigma(a_l)$

σ - Sigmoid

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



$$v_{ij}^{new} \leftarrow v_{ij}^{old} - \eta \frac{\partial E}{\partial v_{ij}}$$

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

$$\frac{\partial E}{\partial v_{il}} = \underbrace{\frac{\partial E}{\partial a_i}}_{\delta_i} \cdot \frac{\partial a_i}{\partial v_{il}} = \delta_i \cdot z_l$$

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

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

$$a_j = \sum_i z_i v_{ji}$$

$$z_i = \sigma(a_i)$$

$$\delta_i = \sum_j \delta_j \cdot v_{ji} \cdot \sigma(a_i)[1 - \sigma(a_i)]$$

$$\delta_i = \sum_j \delta_j \cdot v_{ji} \cdot \sigma(a_i)\left[1 - \sigma(a_i)\right]$$

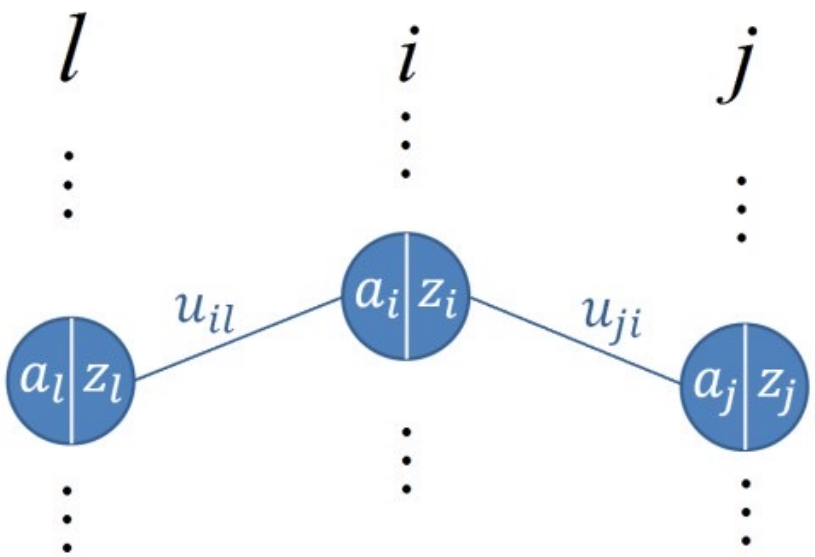
$$\delta_j = \sum_k \delta_k \cdot v_{kj} \cdot \sigma(a_j)\left[1 - \sigma(a_j)\right]$$

$$\delta_j = \delta_k \cdot v_{kj} \cdot \sigma(a_j)\left[1 - \sigma(a_j)\right]$$

$$\delta_k = \frac{\partial E}{\partial a_k} = \frac{\partial (y - \sigma(a_k))^2}{\partial a_k}$$

$$= -2\left(y - \sigma(a_k)\right)\sigma(a_k)\left(1 - \sigma(a_k)\right)$$

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.

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

Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. *Nature* **323,** 533–536 (1986). https://doi.org/10.1038/323533a0

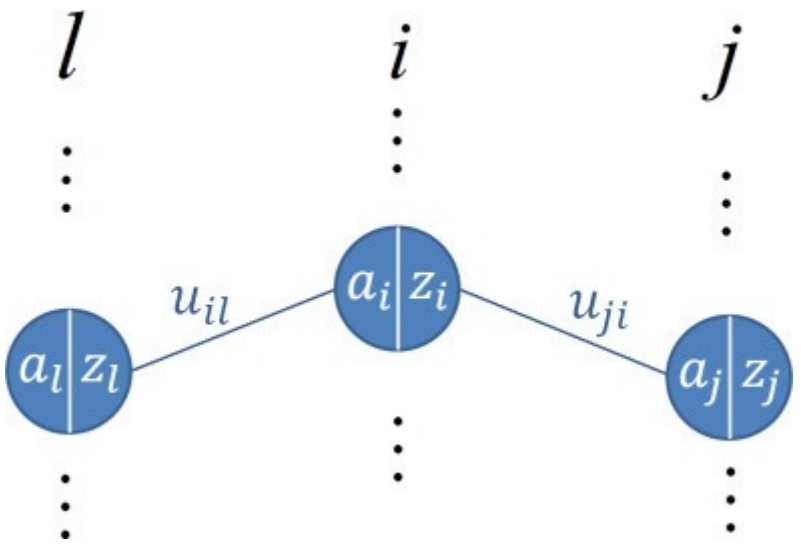
Backpropagation

Take the derivative with respect to weight u_{il}

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

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

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



Backpropagation

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

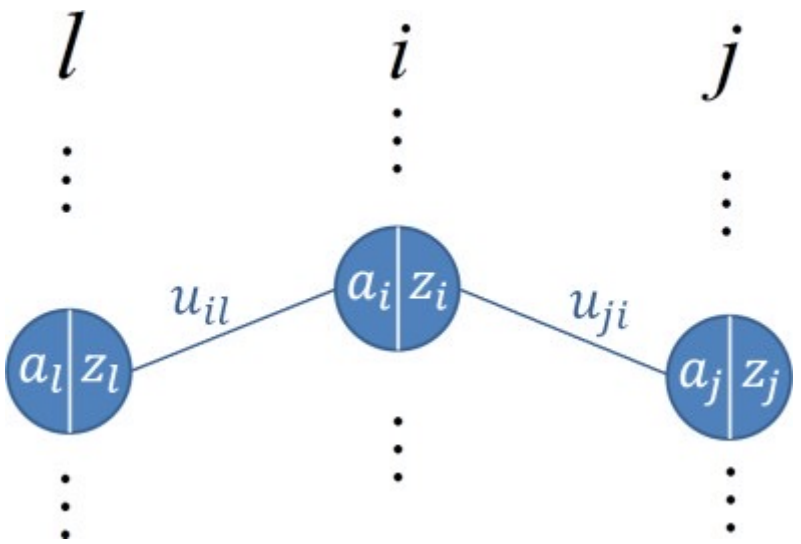
$$\sum_j \frac{\partial |y - \hat{y}|^2}{\partial a_j} \cdot \frac{\partial a_j}{\partial a_i}$$

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

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

where

$$\delta_j = \frac{\partial |y - \hat{y}|^2}{\partial a_j}$$



Backpropagation

Note that if $\sigma(x)$ is the sigmoid function, then
$\sigma'(x) = \sigma(x)(1 - \sigma(x))$

The recursive definition of δ_i
$\delta_i = \sigma'(a_i) \sum_j \delta_j \cdot u_{ji}$

Backpropagation

Now considering δ_k for the output layer:

$$\delta_k = \frac{\partial (y - \hat{y})^2}{\partial a_k} .$$

where $a_k = \hat{y}$

Assume an activation function is not applied in the output layer.

$$\delta_k = \frac{\partial (y - \hat{y})^2}{\partial \hat{y}}$$

$$\delta_k = -2(y - \hat{y})$$

Backpropagation

$$u_{il} \leftarrow u_{il} - \rho \frac{\partial (y - \hat{y})^2}{\partial u_{il}}$$

The network weights are updated using the backpropagation algorithm when each training data point x is fed into the feed forward neural network (FFNN).

all v's

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).

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 $u_{il}^{\text{new}} \leftarrow u_{il}^{\text{old}} - \rho \cdot \dfrac{\partial (y - \hat{y})^2}{\partial u_{il}}$ 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.

Dataset have N Points.

Stochastic gradient descent

Suppose that we want to minimize an objective function that is written as a sum of differentiable functions.

$$Q(w) = \sum_{i=1}^{n} Q_i(w) = \sum_{i=1}^{N} \left(y_i - \hat{y}_i\right)^2 = E$$

Each term Q_i is usually associated with the i_th data point.

Standard gradient descent (batch gradient descent):

$$w = w - \eta \nabla Q(w) = w - \eta \sum_{i=1}^{n} \nabla Q_i(w)$$

where η is the learning rate (step size).

Stochastic gradient descent

Stochastic gradient descent (SGD) considers only a subset of summand functions at every iteration.

This can be quite effective for large-scale problems.

Bottou, Leon; Bousquet, Olivier (2008). The Tradeoffs of Large Scale Learning. Advances in Neural Information Processing Systems 20. pp. 161168.

The gradient of $Q(w)$ is approximated by a gradient at a single example:
$w = w - \eta \nabla Q_i(w)$. $= W \leftarrow W - \eta \partial E / \partial W$

$\frac{\partial E}{\partial w} = \frac{\partial}{\partial w}(y_i - \hat{y}_i)^2$

This update needs to be done for each training example.

Several passes might be necessary over the training set until the algorithm converges.

η might be adaptive.

Stochastic gradient descent

- Choose an initial value for w and η.
- Repeat until converged
  - Randomly shuffle data points in the training set.
  - For $i = 1, 2, ..., n$, do:
  - $w = w - \eta \nabla Q_i(w)$.

Example

Suppose $y = w_1 + w_2 x$

The objective function is:

$$Q(w) = \sum_{i=1}^{n} Q_i(w) = \sum_{i=1}^{n} (w_1 + w_2 x_i - y_i)^2.$$

Update rule will become:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \eta \begin{bmatrix} 2(w_1 + w_2 x_i - y_i) \\ 2x_i(w_1 + w_2 x_i - y_i) \end{bmatrix}.$$

Example from Wikipedia

Mini-batches

Batch gradient decent uses all n data points in each iteration.

Stochastic gradient decent uses 1 data point in each iteration.

Mini-batch gradient decent uses b data points in each iteration.

b is a parameter called Mini-batch size.

Mini-batches

- Choose an initial value for w and η.
- Say $b = 10$
- Repeat until converged
  - Randomly shuffle data points in the training set.
  - For $i = 1, 11, 21, ..., n - 9$, do:
  - $w = w - \eta \sum_{k=i}^{i+9} \nabla Q_i(w)$.

$$w \leftarrow w - \eta \frac{\partial}{\partial w} \left\{ \sum_{i=1}^{10} (y_i - \hat{y}_i)^2 \right\}$$

Tuning η

If η is too high, the algorithm diverges.

If η is too low, makes the algorithm slow to converge.

A common practice is to make η_t a decreasing function of the iteration number t. e.g. $\eta_t = \frac{constant1}{t+constant2}$ $\approx \frac{1}{t}$: 1/epochs.

The first iterations cause large changes in the w, while the later ones do only fine-tuning.