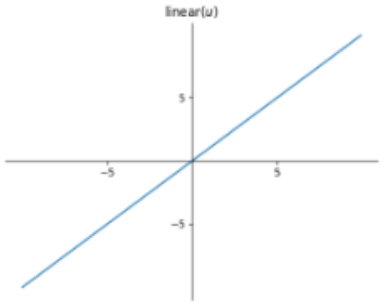
Linear Neuron



A linear neuron is given by

$$u = \boldsymbol{w}^T\boldsymbol{x} + b$$

The activation function is

$$y = f(u) = u$$

The output can be expanded to

$$y = w_1x_2 + w_2x_2 + \cdots + w_nx_n + b, \ \ with\ x_1 \ldots as\ inputs$$

The linear neuron performs linear regression and the weights and bias act as regression coefficients

Given a training dataset $\{(\boldsymbol{x}_p, \mathrm{d}_p)\}_{p=1}^{P}$ where input $\boldsymbol{x} \in \boldsymbol{R_n}$ and target $d_p \in \boldsymbol{R}$ training a linear neuron finds a regression function ϕ: $\boldsymbol{R}^n \rightarrow \boldsymbol{R}$, that is given by the linear mapping: $y = \boldsymbol{w}^T\boldsymbol{x} + b$

Cost function: $J = \frac{1}{2}(d-y)^2$

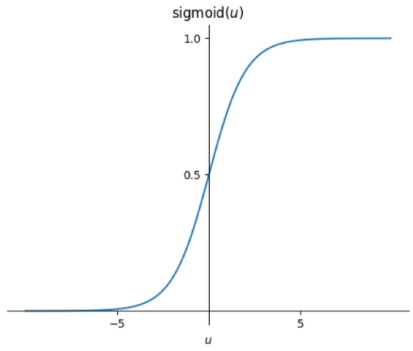
$$\nabla_{\mathrm{u}} J = \frac{\partial J}{\partial u} = \frac{\partial J}{\partial y} = -(d-y)$$

$$\nabla_{\boldsymbol{w}} J = \frac{\partial J}{\partial \boldsymbol{w}} = \frac{\partial J}{\partial u}\,\frac{\partial u}{\partial \boldsymbol{w}} = -(d-y)\cdot\boldsymbol{x}$$

$$\nabla_{b} J = \frac{\partial J}{\partial b} = \frac{\partial J}{\partial u}\,\frac{\partial u}{\partial b} = -(d-y)\cdot 1$$

GD	SGD
$(\boldsymbol{X}, \boldsymbol{d})$	$(\boldsymbol{x}_p, d_p)$
$J = \frac{1}{2}\sum_{p=1}^{P}(d_p - y_p)^2$	$J = \frac{1}{2}(d_p - y_p)^2$
$\boldsymbol{y} = \boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P$	$y_p = u_p = \boldsymbol{x}_p{}^T\boldsymbol{w} + b$
$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\boldsymbol{X}^T(\boldsymbol{d}-\boldsymbol{y})$	$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(d_p - y_p)\boldsymbol{x}_p$
$b \leftarrow b + \alpha\boldsymbol{1}_P{}^T(\boldsymbol{d}-\boldsymbol{y})$	$b \leftarrow b + \alpha(d_p - y_p)$

Perceptron- logistic function



A Perceptron neuron is given by

$$u = \boldsymbol{w}^T\boldsymbol{x} + b$$

The activation function is

$$y = f(u) = \frac{1}{1+e^u} = sigmoid(u)$$

The square error is used as cost function for learning

ϕ is a non-linear function and does non-linear regression

For the training date set $\{(\boldsymbol{x}_p, \mathrm{d}_p)\}_{p=1}^{P}$ the continuous perceptron finds a functional mapping ϕ: $\boldsymbol{R}^n \rightarrow \boldsymbol{R}$ by learning from training data.

Cost function: $J = \frac{1}{2}(d-y)^2$

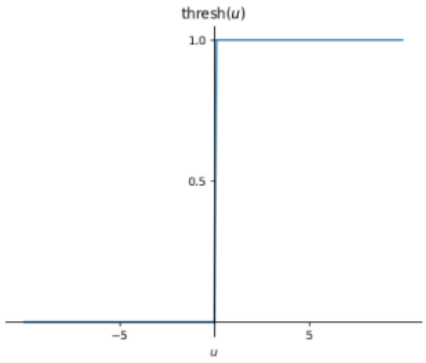
$$\nabla_{\mathrm{u}} J = \frac{\partial J}{\partial u} = \frac{\partial J}{\partial y} = -(d-y)\cdot f'(u)$$

$$\nabla_{\boldsymbol{w}} J = \frac{\partial J}{\partial \boldsymbol{w}} = \frac{\partial J}{\partial u}\,\frac{\partial u}{\partial \boldsymbol{w}} = -(d-y)\cdot f'(u)\cdot\boldsymbol{x}$$

$$\nabla_{b} J = \frac{\partial J}{\partial b} = \frac{\partial J}{\partial u}\,\frac{\partial u}{\partial b} = -(d-y)\cdot f'^{(u)}\cdot 1$$

GD	SGD
$(\boldsymbol{X}, \boldsymbol{d})$	$(\boldsymbol{x}_p, d_p)$
$J = \frac{1}{2}\sum_{p=1}^{P}(d_p - y_p)^2$	$J = \frac{1}{2}(d_p - y_p)^2$
$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P$	$u_p = \boldsymbol{x}_p{}^T\boldsymbol{w} + b$
$\boldsymbol{y} = f(\boldsymbol{u})$	$y_p = f(u_p)$
$\boldsymbol{w} = \boldsymbol{w} + \alpha\boldsymbol{X}^T(\boldsymbol{d}-\boldsymbol{y})\cdot f'(\boldsymbol{u})$	$\boldsymbol{w} = \boldsymbol{w} + \alpha(d_p - y_p)f'(u_p)\boldsymbol{x}_p$
$b = b + \alpha\boldsymbol{1}_P{}^T(\boldsymbol{d}-\boldsymbol{y})\cdot f'(\boldsymbol{u})$	$b = b + \alpha(d_p - y_p)f'(u_p)$

Discrete Perceptron



A Discrete Perceptron is given by

$$u = \boldsymbol{w}^T \boldsymbol{x} + b$$

The activation function is

$$y = f(u) = \begin{cases} 1, & u > 0 \\ 0, & u \leq 0 \end{cases} = thresh(u)$$

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$ where input $\boldsymbol{x} \in \boldsymbol{R_n}$ and target $d_p \in \{1,0\}$ training a Discrete Perceptron neuron finds a linear decision boundary in the feature space

Cost function: $\delta = (d - y) \in \{-1,0,1\}$

Batch learning	Stochastic learning
$(\boldsymbol{X}, \boldsymbol{d})$	$(\boldsymbol{x}_p, d_p)$
$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P$	$u_p = \boldsymbol{x}_p{}^T \boldsymbol{w} + b$
$\boldsymbol{y} = 1(\boldsymbol{u} > 0)$	$y_p = 1(u_p > 0)$
$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{y})$	$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(d_p - y_p)\boldsymbol{x}_p$
$b \leftarrow b + \alpha \boldsymbol{1}_P{}^T(\boldsymbol{d} - \boldsymbol{y})$	$b \leftarrow b + \alpha(d_p - y_p)$

Linear separator

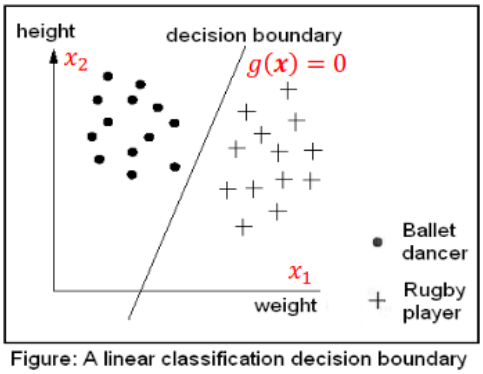


Figure: A linear classification decision boundary

$$G(\boldsymbol{x}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

$$\boldsymbol{w} = (w_1 \quad w_2 \quad \dots \quad w_n), \qquad b = w_0$$

The weight vector is orthogonal to the line.

$$\begin{pmatrix} 1 \\ 3 \end{pmatrix} \perp \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

Logistic Regression

A logistic regression neuron performs a binary classification of inputs. That is, it classifies inputs into two classes with labels 0 and 1'. The activation of a logistic regression neuron gives the probability of the neuron output belonging to class

Given an input **x**, the activation of the neuron gives P(y=1|**x**)

$$f(u) = P(y = 1 \mid \boldsymbol{x}) = \frac{1}{1 + e^{-u}}$$

Nb: the input gives the probability that the input belong to class

The output y of the neuron is given by:

$$y = 1(f(u) > 0.5)$$

Decision boundary is given by u=0.

Cost function:

$$J = -d \cdot \log(f(u)) - (1 - d) \cdot \log(1 - f(u))$$

$$\nabla_u J = \frac{\partial J}{\partial u} = \frac{\partial J}{\partial y} = -(d - f(u)) \cdot f'(u)$$

$$\nabla_{\boldsymbol{w}} J = \frac{\partial J}{\partial \boldsymbol{w}} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial \boldsymbol{w}} = -(d - f(u)) \cdot \boldsymbol{x}$$

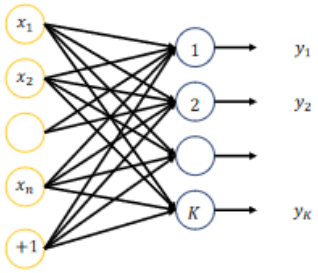
$$\nabla_b J = \frac{\partial J}{\partial b} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial b} = -(d - f(u)) \cdot 1$$

GD	SGD
$(\boldsymbol{X}, \boldsymbol{d})$	$(\boldsymbol{x}_p, d_p)$
$J(\boldsymbol{w}, b) = -\sum_{p=1}^{P} d_p \log(f(u_p)) + (1 - d_p)\log(1 - f(u_p))$	$J(\boldsymbol{w}, b) = -d_p \log(f(u_p)) - (1 - d_p)\log(1 - f(u_p))$
$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P$	$u_p = \boldsymbol{x}_p{}^T \boldsymbol{w} + b$
$f(\boldsymbol{u}) = \frac{1}{1 + e^{-\boldsymbol{u}}}$	$f(u_p) = \frac{1}{1 + e^{-u_p}}$
$\boldsymbol{y} = 1(f(\boldsymbol{u}) > 0.5)$	$y_p = 1(f(u_p) > 0.5)$
$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \boldsymbol{X}^T(\boldsymbol{d} - f(\boldsymbol{u}))$	$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(d_p - f(u_p))\boldsymbol{x}_p$
$b \leftarrow b + \alpha \boldsymbol{1}_P{}^T(\boldsymbol{d} - f(\boldsymbol{u}))$	$b \leftarrow b + \alpha(d_p - f(u_p))$

Derivatives

	$\boldsymbol{y}$	$\boldsymbol{y'}$
Linear	u	1
Perceptron "Sigmoid"	$\frac{1}{1 + e^{u}}$	$y(1 - y)$
Tanh "Bipolar Sigmoid"	$\frac{e^{u} - e^{-u}}{e^{u} + e^{-u}}$	$1 - y^2$
ReLu	$\max(0, u)$	NA
Threshold	$1(u > 0)$	NA

Layers – Single input



Let $\boldsymbol{w_k}$ and b_k denote the weight vector and bias of k-th neuron

Weights connected a neuron layers is represented by a weight **W** where columns are given by weight vectors connected to individual neurons. The dimension is (feature X labels)

$$\boldsymbol{W} = (\boldsymbol{w_1}\ \ \boldsymbol{w_2}\ \ \dots\ \ \boldsymbol{w_K}) = features \underbrace{\begin{Bmatrix} \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1K} \\ w_{21} & w_{22} & \dots & w_{2K} \\ \vdots & \vdots & \vdots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nK} \end{bmatrix} \end{Bmatrix}}_{labels}$$

And

$$\boldsymbol{b} = (b_1, b_2, \dots, k_K)^T$$

Given an input pattern $\boldsymbol{x} \in \boldsymbol{R^n}$ to a layer of k neurons

$$\phi: \boldsymbol{R^n} \rightarrow \boldsymbol{R^k}$$

Synaptic input to kth neuron u_k:

$$\boldsymbol{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_K \end{pmatrix} = \begin{pmatrix} \boldsymbol{w_1^T x + b_1} \\ \boldsymbol{w_2^T x + b_2} \\ \vdots \\ \boldsymbol{w_K^T x + b_K} \end{pmatrix} = \begin{pmatrix} \boldsymbol{w_1^T} \\ \boldsymbol{w_2^T} \\ \vdots \\ \boldsymbol{w_K^T} \end{pmatrix} \boldsymbol{x + b} = \boldsymbol{W^T x + b}$$

$$\boldsymbol{y} = f(\boldsymbol{u}) = \begin{pmatrix} f(u_1) \\ f(u_2) \\ \vdots \\ f(u_K) \end{pmatrix}$$

Layers – Batch

Given a training dataset $\{(\boldsymbol{x}_p)\}_{p=1}^{P}$ input patterns to a layer of K neurons where $\boldsymbol{x_p} \in \mathbf{R^n}$

Synaptic input $\boldsymbol{u}_p$ to the layer for an input pattern $\boldsymbol{x}_p$:

$$\boldsymbol{u}_p = \boldsymbol{W}^T \boldsymbol{x}_p + \boldsymbol{b}$$

The matrix **U** of synaptic inputs to the layer of P pattern

$$\boldsymbol{U} = \begin{pmatrix} \boldsymbol{u}_1^T \\ \boldsymbol{u}_2^T \\ \vdots \\ \boldsymbol{u}_P^T \end{pmatrix} = \begin{pmatrix} \boldsymbol{W}^T \boldsymbol{x}_1 + \boldsymbol{b} \\ \boldsymbol{W}^T \boldsymbol{x}_2 + \boldsymbol{b} \\ \vdots \\ \boldsymbol{W}^T \boldsymbol{x}_P + \boldsymbol{b} \end{pmatrix} = \boldsymbol{XW + B}$$

With bias matrix

$$B = \begin{pmatrix} \boldsymbol{b^T} \\ \boldsymbol{b^T} \\ \vdots \\ \boldsymbol{b^T} \end{pmatrix}$$

And input with dimension (pattern X features)

$$\boldsymbol{X} = patterns \underbrace{\begin{Bmatrix} \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ x_{P1} & x_{P2} & \dots & x_{Pn} \end{bmatrix} \end{Bmatrix}}_{Features}$$

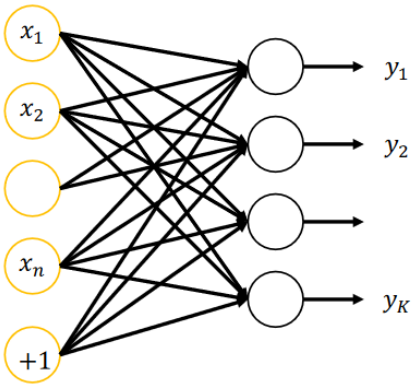
And activation function

$$\boldsymbol{Y} = F(\boldsymbol{U}) = \begin{pmatrix} f(\boldsymbol{u}_1)^T \\ f(\boldsymbol{u}_2)^T \\ \vdots \\ f(\boldsymbol{u}_P)^T \end{pmatrix} = \begin{pmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_P^T \end{pmatrix}$$

$$= P_{pattern} \underbrace{\begin{Bmatrix} \begin{bmatrix} y_{11} & y_{12} & \dots & y_{1K} \\ y_{21} & y_{22} & \dots & y_{2K} \\ \vdots & \vdots & \vdots & \vdots \\ y_{P1} & y_{P2} & \dots & y_{PK} \end{bmatrix} \end{Bmatrix}}_{K_{neurons}}$$

Learning of a layer		
SGD	$\nabla_W J = \boldsymbol{x}(\nabla_u J)^T$ $\nabla_b J = \nabla_u J$	$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{x}(\nabla_u J)^T$ $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_u J$
GD	$\nabla_W J = \boldsymbol{X}^T \nabla_U J$ $\nabla_b J = (\nabla_U J)^T \mathbf{1}_P$	$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_U J$ $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha (\nabla_U J)^T \mathbf{1}_P$

Perceptron layer



A layer that perform multidimensional regression. A Layer of K perceptron learns a multidimensional non-linear mapping:

$$\phi: R^n \rightarrow R^K$$

The learning is like single neuron learning.

Cost function: $J = \frac{1}{2}(d-y)^2$

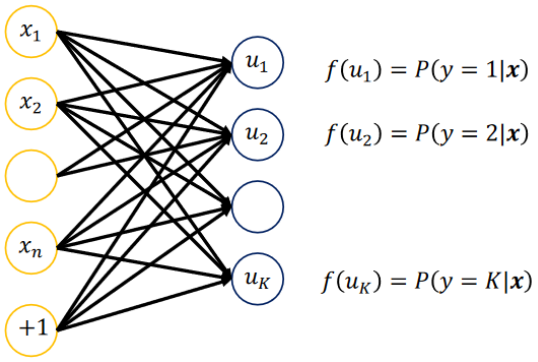
$$\nabla_u \boldsymbol{J} = \begin{pmatrix} \nabla_{u_1} J \\ \nabla_{u_2} J \\ \vdots \\ \nabla_{u_K} J \end{pmatrix} = -(\boldsymbol{d}-\boldsymbol{y}) \cdot f'(\boldsymbol{u})$$

GD	SGD
$(\boldsymbol{X}, \boldsymbol{D})$	$(\boldsymbol{x}, \boldsymbol{d})$
$\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B}$	$\boldsymbol{u} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$
$\boldsymbol{Y} = f(\boldsymbol{U})$	$\boldsymbol{y} = f(\boldsymbol{u})$
$\nabla_U J = -(\boldsymbol{D}-\boldsymbol{Y}) \cdot f'(\boldsymbol{U})$	$\nabla_u J = -(\boldsymbol{d}-\boldsymbol{y}) \cdot f'(\boldsymbol{u})$
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_U J$	$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{x} (\nabla_u J)^T$
$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha (\nabla_U J)^T \boldsymbol{1}_P$	$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_u J$

Summary for layers

Layers	$f(\boldsymbol{U}), \boldsymbol{Y}$	$\nabla_U J$
Linear	$\boldsymbol{Y} = f(\boldsymbol{U}) = \boldsymbol{U}$	$-(\boldsymbol{D}-\boldsymbol{Y})$
Perceptron	$Y = f\dfrac{1}{1+e^{-\boldsymbol{U}}}$	$-(\boldsymbol{D}-\boldsymbol{Y}) \cdot f'(\boldsymbol{U})$
Softmax	$F(\boldsymbol{U}) = \dfrac{e^U}{\sum_{k=1}^K e^{U_k}}$ $\boldsymbol{y} = \underset{k}{argmax}\, f(\boldsymbol{U})$	$-\big(K - f(\boldsymbol{U})\big)$ K= Onehot

Softmax Layer



Softmax layer is the extension of a logistic regression to multiclass classification problem, which is also know as multinomial logistic regression

The K neurons in the sofmax layer performs multinomial logistic regression and represent K classes.
The activation of each neuron k estimates the probability $P(y=k|x)$ that the input x belongs to class k:

$$P(y=k|\boldsymbol{x}) = f(u_k) = \frac{e^{u_k}}{\sum_{k'=1}^K e^{u_{k'}}}$$

Where $u_k = \boldsymbol{x}^T \boldsymbol{W}_k + b_k$ and $\boldsymbol{w}_k$is the weight vector and b_k is bias of neuron k
The above activation function f is known as softmax activation function

The output of y denotes the class label of the input pattern, which is given by

$$y = argmax_k\, P(y=k|\boldsymbol{x}) = argmax_k\, f(u_k)$$

That is, the class label is assigned to the class with the maximum activation.

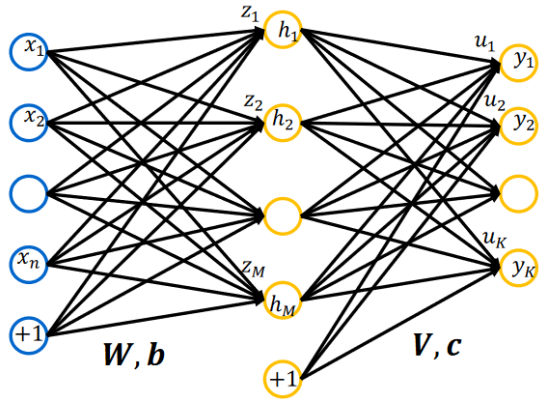
$$\boldsymbol{U} = \begin{pmatrix} 0.86 & 0.11 & 0.64 \\ -0.84 & -0.28 & -0.49 \\ -0.41 & -0.16 & -0.22 \\ -0.01 & -0.21 & 0.17 \\ -0.86 & -0.82 & -0.06 \\ -1.15 & -0.27 & -0.75 \\ 0.11 & -0.31 & 0.35 \\ -0.12 & -0.10 & -0.02 \end{pmatrix}$$

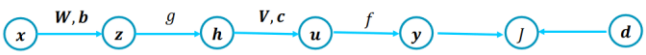
$$f(u_{12}) = \frac{e^{0.11}}{e^{0.86} + e^{0.11} + e^{0.64}}$$

$$f(\boldsymbol{U}) = \frac{e^{(\boldsymbol{U})}}{\sum_{k=1}^K e^{(\boldsymbol{U})}} = \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix}$$

GD	SGD
$(\boldsymbol{X}, \boldsymbol{D})$	$(\boldsymbol{x}, d)$
$\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B}$	$\boldsymbol{u} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$
$f(\boldsymbol{U}) = \frac{e^U}{\sum_{k'=1}^K e^{U_{k'}}}$	$f(\boldsymbol{u}) = \frac{e^{u_k}}{\sum_{k'=1}^K e^{u_{k'}}}$
$\boldsymbol{y} = \underset{k}{\text{argmax}}\, f(\boldsymbol{U})$	$y = \underset{k}{\text{argmax}}\, f(\boldsymbol{u})$
$\nabla_U J = -\big(\boldsymbol{K} - f(\boldsymbol{U})\big)$	$\nabla_u J = -\big(1(\boldsymbol{k}=d) - f(\boldsymbol{u})\big)$
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_U J$	$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{x} (\nabla_u J)^T$
$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha (\nabla_U J)^T \boldsymbol{1}_P$	$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_u J$

Deep FFN - SGD





Synaptic input **z** to hidden layer:

$$\boldsymbol{z} = \boldsymbol{W^T x} + \boldsymbol{b}$$

Output **h** of the hidden layers

$$\boldsymbol{h} = g(\boldsymbol{z})$$

Synaptic input **u** to output layer:

$$\boldsymbol{u} = \boldsymbol{V^T h} + \boldsymbol{c}$$

Output **y** of the output layer

$$\boldsymbol{y} = f(\boldsymbol{u})$$

Output layer:

$$\nabla_u J = \begin{cases} -(\boldsymbol{d} - \boldsymbol{y}), & for\ linear\ layers \\ -\big(1(\boldsymbol{k} = d) - f(\boldsymbol{u})\big), & for\ softmax\ layer \end{cases}$$

Update of weight:

$$\boldsymbol{V} \leftarrow \boldsymbol{V} - \alpha \cdot \boldsymbol{h} \cdot (\nabla_u \boldsymbol{J})^T$$

$$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha \cdot \nabla_u \boldsymbol{J}$$

Hidden layer:

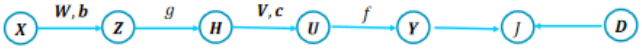
$$\nabla_z \boldsymbol{J} = \boldsymbol{V} \cdot \nabla_u \boldsymbol{J} \cdot g'(\boldsymbol{z})$$

Update of weight:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \cdot \boldsymbol{x} \cdot (\nabla_z \boldsymbol{J})^T$$

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \cdot \nabla_z \boldsymbol{J}$$

Deep FFN – GD



Synaptic input **Z** to hidden layer:

$$\boldsymbol{Z} = \boldsymbol{XW} + \boldsymbol{B}$$

Output **h** of the hidden layers

$$\boldsymbol{H} = g(\boldsymbol{Z})$$

Synaptic input **u** to output layer:

$$\boldsymbol{U} = \boldsymbol{HV} + \boldsymbol{C}$$

Output **y** of the output layer

$$\boldsymbol{Y} = f(\boldsymbol{U})$$

Output layer:

$$\nabla_u J = \begin{cases} -(\boldsymbol{D} - \boldsymbol{Y}), & for\ linear\ layers \\ -\big(\boldsymbol{K} - f(\boldsymbol{U})\big), & for\ softmax\ layer \end{cases}$$

Update of weight:

$$\boldsymbol{V} \leftarrow \boldsymbol{V} - \alpha \cdot \boldsymbol{H}^T \cdot (\nabla_U \boldsymbol{J})^T$$

$$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha \cdot (\nabla_u \boldsymbol{J})^T \cdot \boldsymbol{1_p}$$

Hidden layer:

$$\nabla_Z \boldsymbol{J} = \boldsymbol{V} \cdot \nabla_U \boldsymbol{J} \cdot g'(\boldsymbol{Z})$$

Update of weight:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \cdot \boldsymbol{X^T} \cdot \nabla_Z \boldsymbol{J}$$

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \cdot (\nabla_Z \boldsymbol{J})^T \cdot \boldsymbol{1_P}$$

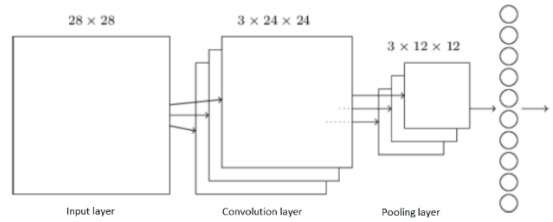
Definition

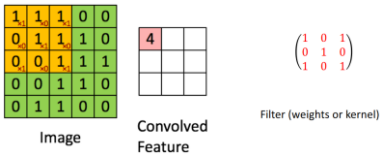
	SGD	GD
Single Neuron	$u_p = \boldsymbol{w}^T \boldsymbol{x}_p + b$ $y_p = f(u_p)$	$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b$ $\boldsymbol{y} = f(\boldsymbol{u})$
Layered	$\boldsymbol{u} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$ $\boldsymbol{y} = f(\boldsymbol{u})$	$\boldsymbol{U} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}$ $\boldsymbol{Y} = f(\boldsymbol{U})$

$\boldsymbol{x} = features \begin{Bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \end{Bmatrix}$	$\boldsymbol{X} = patterns \begin{Bmatrix} \begin{pmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_P^T \end{pmatrix} \end{Bmatrix} = patterns \begin{Bmatrix} \underbrace{\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ x_{P1} & x_{P2} & \dots & x_{Pn} \end{bmatrix}}_{Features} \end{Bmatrix}$
$\boldsymbol{w} = features \begin{Bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \end{Bmatrix}$	$\boldsymbol{W} = \underbrace{(\boldsymbol{w_1} \ \ \boldsymbol{w_2} \ \dots \ \boldsymbol{w_K})}_{labels} = features \begin{Bmatrix} \underbrace{\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1K} \\ w_{21} & w_{22} & \dots & w_{2K} \\ \vdots & \vdots & \vdots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nK} \end{bmatrix}}_{labels} \end{Bmatrix}$
$\boldsymbol{b} = labels \begin{Bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{bmatrix} \end{Bmatrix}$	$\boldsymbol{B} = patterns \begin{Bmatrix} \begin{bmatrix} \boldsymbol{b}^T \\ \boldsymbol{b}^T \\ \vdots \\ \boldsymbol{b}^T \end{bmatrix} \end{Bmatrix} = patterns \begin{Bmatrix} \underbrace{\begin{bmatrix} b_1 & b_2 & \dots & b_K \\ b_1 & b_2 & \dots & b_K \\ \vdots & \vdots & \vdots & \vdots \\ b_1 & b_2 & \dots & b_K \end{bmatrix}}_{labels} \end{Bmatrix}$
$\boldsymbol{y} = labels \begin{Bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_K \end{bmatrix} \end{Bmatrix}$	$\boldsymbol{Y} = \begin{pmatrix} \boldsymbol{y}_1^T \\ \boldsymbol{y}_2^T \\ \vdots \\ \boldsymbol{y}_P^T \end{pmatrix} = patterns \underbrace{\begin{Bmatrix} \begin{bmatrix} y_{11} & y_{12} & \dots & y_{1K} \\ y_{21} & y_{22} & \dots & y_{2K} \\ \vdots & \vdots & \vdots & \vdots \\ y_{P1} & y_{P2} & \dots & y_{PK} \end{bmatrix} \end{Bmatrix}}_{K_{neurons}}$

,

CNN





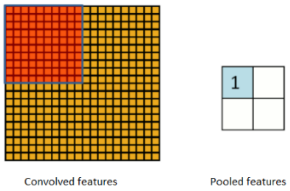
Let x_p denote the input window of the pixel, and **w** be the filter of the feature map, then the output of the neuron will be

$$u_p = \boldsymbol{x}_p^T \boldsymbol{w} + b$$

$$y_p = sigmoid(u_p)$$

3 filters give 3 feature maps.

When the convolution features is found the pooling is done by taking "max" or "avg" pooling.



Size of feature maps

IxJ = image_input size LxM = window size

1. VALID: apply the filter wherever it completely overlaps with the input.
   - $(I - L + 1)\ x\ (J - M + 1)$
2. SAME: apply the filter to make the output same size as input. The input is padded with zeros in obtaining the output.
   - IxJ

Stride:

The distance between adjacent centers of the kernel:

The convolution operation is intended to extract features of the input space as finely as possible. The **default** strides for convolution is 1 unit.

Pooling is intended to subsample the convolution layer. The **default** stride for pooling is equal the filter width

CNN – Example

Given an input pattern **X**:

$$\boldsymbol{X} = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.3 & -0.2 \\ 0.7 & 0.1 & 0.2 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}$$

The input pattern is received by an convolution layer consisting of one kernel

$$\boldsymbol{w} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}, \qquad b = 0.05$$

If convolution layer has a sigmoid activation function, find the outputs of the convolution layer if the padding is VALID at strides = 1.
If the pooling layer uses max pooling, has a pooling window size of 2x2, and strides =2, find the activations at the pooling layer for VALID and SAME padding.

Synaptic input to the convolution layer

$$f\big(u(i,j)\big) = \sum_l \sum_m x(i+l, j+m) \cdot w(l,m) + b$$

$$U = \begin{pmatrix} -0.35 & 1.35 & 0.75 \\ -0.55 & 0.85 & 0.05 \\ 0.75 & 0.05 & 1.15 \end{pmatrix}$$

Output of the convolution layer

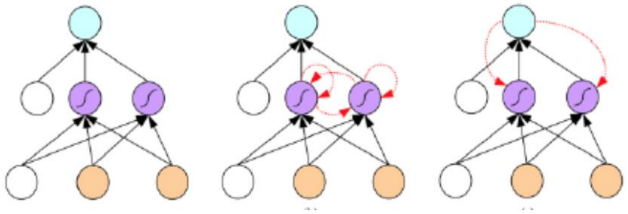
$$f(\boldsymbol{U}) = \frac{1}{1 + e^{-\boldsymbol{U}}} = \begin{pmatrix} 0.413 & 0.794 & 0.679 \\ 0.366 & 0.701 & 0.512 \\ 0.679 & 0.512 & 0.76 \end{pmatrix}$$

Output of the max pooling layer:

$$(0.794) : \text{VALID } pooling$$

$$\begin{pmatrix} 0.794 & 0.670 \\ 0.679 & 0.76 \end{pmatrix} : \text{SAME } pooling$$

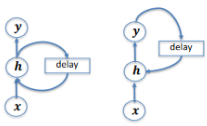
RNN

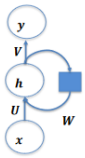


Feedforward NN

RNN with hidden recurrence (Elman-type)

RNN wiith top-down recurrence (Jordan-type)





$\boldsymbol{U}$: weight vector that transforms raw inputs to the hidden-layer
$\boldsymbol{W}$: recurrent weight vector connecting previous hidden-layer output to hidden input
$\boldsymbol{V}$: weight vector of the output layer
$\boldsymbol{b}$: bias connected to hidden layer
$\boldsymbol{c}$: bias connected to the output layer

ϕ: the $tanh$ hidden-layer activation function
σ: the $linear/softmax$ output-layer activation function.

SGD

Let **x**(t), **y**(t) and **h**(t) be the input, output and hidden output of the network at time t.

Activation of the Elman-type RNN with one hidden-layer is given by:

$$\boldsymbol{h}(t) = \Phi\left(\boldsymbol{U^T}x(\boldsymbol{t}) + \boldsymbol{W^T}\boldsymbol{h}(t-1) + \boldsymbol{b}\right)$$
$$\boldsymbol{y}(t) = \sigma\left(\boldsymbol{V^T}\boldsymbol{h}(t) + \boldsymbol{c}\right)$$

GD

Given P patterns $\{\boldsymbol{x}_p\}_{p=1}^{P}$ where

$$\boldsymbol{x}_p = \left(\boldsymbol{x}_p(t)\right)_{t=1}^{T}$$

denoting time from 1 to T

$$\boldsymbol{X}(t) = \begin{pmatrix} \boldsymbol{x}_1(t)^T \\ \boldsymbol{x}_2(t)^T \\ \vdots \\ \boldsymbol{x}_p(t)^T \end{pmatrix}$$

Let **X**(t), **Y**(t) and **H**(t) be batch input, output and hidden output of the network at time t.

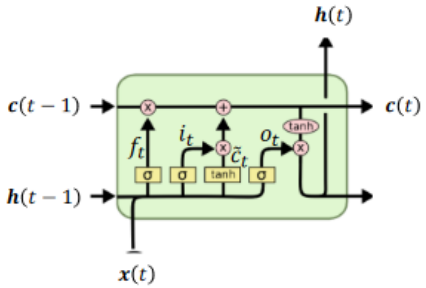
Activation of the three-layer Elman-type RNN is given by:

$$\boldsymbol{H}(t) = \phi\left(\underbrace{\boldsymbol{X}(t)\boldsymbol{U}}_{input\cdot W_1} + \underbrace{\boldsymbol{H}(t-1)\boldsymbol{W}}_{Hidded\ out\cdot W_2} + \boldsymbol{B}\right)$$
$$\boldsymbol{Y}(t) = \sigma(\boldsymbol{H}(t)\boldsymbol{V} + \boldsymbol{C})$$

Matrix sizes

RNN	LSTM				GRU		
$\boldsymbol{w}$	$\{\boldsymbol{w}_i$	$\boldsymbol{w}_f$	$\boldsymbol{w}_o$	$\boldsymbol{w}_c\}$	$\{\boldsymbol{w}_r$	$\boldsymbol{w}_z$	$\boldsymbol{w}_h\}$
$\boldsymbol{b}$	$\{\boldsymbol{b}_i$	$\boldsymbol{b}_f$	$\boldsymbol{b}_o$	$\boldsymbol{b}_c\}$	$\{\boldsymbol{b}_r$	$\boldsymbol{b}_z$	$\boldsymbol{b}_h\}$
$\boldsymbol{U}$	$\{\boldsymbol{U}_i$	$\boldsymbol{U}_f$	$\boldsymbol{U}_o$	$\boldsymbol{U}_c\}$	$\{\boldsymbol{U}_r$	$\boldsymbol{U}_z$	$\boldsymbol{U}_h\}$

Long Short Term Memory



F is the forget gate:

$$\boldsymbol{f}(t) = \sigma\left(\boldsymbol{U}_f^T x(t) + \boldsymbol{W}_f^T \boldsymbol{h}(t-1) + \boldsymbol{b}_f\right)$$

The forget gate can modulate the memory cell's self-recurrent connection, allowing the cell to remember or forget its previous state as needed. The value of **f**(t) determines if c(t-1) is to be remembered or not.

I is the input gate, $\tilde{C}$ Is the internal cell state

$$\boldsymbol{i}(t) = \sigma\left(\boldsymbol{U}_i^T \boldsymbol{x}(t) + \boldsymbol{W}_i^T \mathrm{h}(t-1) + \boldsymbol{b}_i\right)$$
$$\tilde{\boldsymbol{c}}(t) = \phi\left(\boldsymbol{U}_c^T \boldsymbol{x}(t) + \boldsymbol{W}_c^T \mathrm{h}(t-1) + \boldsymbol{b}_c\right)$$

The input gate can allow incoming signal to alter the state of the memory cell or block it. It decides what new information to store in the cell stage

This has two parts: a sigmoid input fate layer decides which values to updates; a tanh layer creates a vector of new candidate values $\tilde{\boldsymbol{c}}$ that could be added to the state.

C is the cell state

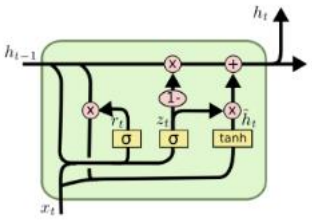
$$\boldsymbol{c}(\boldsymbol{t}) = \tilde{\boldsymbol{c}}(\boldsymbol{t}) \odot i(\boldsymbol{t}) + \boldsymbol{c}(t-1) \odot \boldsymbol{f}(t)$$

O is the output gate

$$\boldsymbol{o}(t) = \sigma\left(\boldsymbol{U}_o^T \boldsymbol{x}(t) + \boldsymbol{W}_o^T \mathrm{h}(t-1) + \boldsymbol{b}_o\right)$$
$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{c}(t)\right) \odot \boldsymbol{i}(t)$$

The output gate can allow the state of the memory cell to have an effect on other neurons or prevent it

Gated Recurrent Unit



$$\boldsymbol{r}(t) = \sigma(U_r^T \cdot \boldsymbol{x}(t) + \boldsymbol{W}_r^T \cdot \boldsymbol{h}(t-1) + \boldsymbol{b}_r)$$

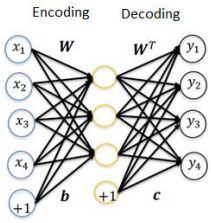
$$\boldsymbol{z}(t) = \sigma(U_z^T \cdot \boldsymbol{x}(t) + \boldsymbol{W}_z^T \cdot \boldsymbol{h}(t-1) + \boldsymbol{b}_z)$$

$$\tilde{\boldsymbol{h}}(t) = \phi\left(\boldsymbol{U}_h^T \cdot \boldsymbol{x}(t) + \boldsymbol{W}_h^T\left(r(t) \odot \boldsymbol{h}(t-1)\right) + \boldsymbol{b}_h\right)$$

$$\boldsymbol{h}(t) = \left(1 - \boldsymbol{z}(t)\right) \odot \boldsymbol{h}(t-1) + \boldsymbol{z}(t) \odot \tilde{\boldsymbol{h}}(t)$$

R is the reset ate and z update gate.

Autoencoders



The input signal is encoded into a code that represent the input, the output decodes the signal again.
The autoencoder goal is to reproduce the input data.
The network therefore learn the features of the input.

The network only performs optimal on trained data variation of trained inputs.

Given an input **x**, the hidden-layer performs the encoding function $\boldsymbol{h} = \phi(\boldsymbol{x})$ and the decoder φ produces the reconstruction $\boldsymbol{y} = \varphi(\boldsymbol{h})$.

If the autoencoder succeeds:

$$\boldsymbol{y} = \varphi\big(\phi(\boldsymbol{x})\big) = \boldsymbol{x}$$

Reverse mapping form the hidden layer to the output can be optionally constrained to be the same as the input to hidden layer. That is $\mathbf{W}' = \boldsymbol{W}^T$

Hidden layer and output layer activation can be written as

$$\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}) : \ g(\boldsymbol{XW} + \boldsymbol{B})$$
$$\boldsymbol{y} = f(\boldsymbol{Wh} + \boldsymbol{c}) : f\big(\boldsymbol{HW^T} + \boldsymbol{B'}\big)$$

f is usually sigmoid

The cost function of reconstruction can be measured by many ways, depending on the appropriate distributional assumptions on the inputs.

Learning of autoencoders is unsupervised as no specific targets are given.

Given a training set $\{\boldsymbol{x}_p\}_{p=1}^{P}$

The mean-square-error cost is usually used if the data is assumed to be continuous and Gaussian distributed:

$$J_{mse} = \frac{1}{P}\sum_{p=1}^{P}\left|\boldsymbol{y}_p - \boldsymbol{x}_p\right|^2$$

Where $\boldsymbol{y}_p$ is the output for input x_p and |·| denotes the magnitude of the vector.

If the inputs are interpreted as bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used

$$J_{cross} = -\sum_{p=1}^{P}\big(\boldsymbol{x}_p \cdot \log y_p + \big(1 - \boldsymbol{x}_p\big) \cdot \log\big(1 - \boldsymbol{y}_p\big)\big)$$

Denoising Autoencoders (DAE)

This encoder is used to removed noise from the data

The denoising autoencoders (DAE) receives corrupted data points as inputs and is trained to predict the original uncorrupted data points as its output.

The idea of DAE is that in order to force the hidden layer to discover more robust features and prevent it from simply learning the identity. We train the autoencoder to reconstruct the input from a corrupted version of it.

In other words, DAE attempts to encode the input (preserve the information about input) and attempts to undo the effect of corruption process applied to the input of the autoencoder.

To obtain corrupted version of input data, each input x_i of input data is added with additive or multiplicative noise.

Additive noise:

$$\tilde{x}_i = x_i + \epsilon$$

where noise ϵ is Gaussian distributed:

$$\epsilon \sim N(0, \sigma^2)$$

And σ is the standard deviation that determines the ration S/N ratio. Usually used for continuous data.

Multiplication noise:

$$\tilde{x}_i = \epsilon \cdot x_i$$

where noise ϵ could be Binomially distributed:

$$\epsilon \sim Binomial(p)$$

And p is the probability of ones and $1 - p$ is the probability of zeros (noise). Usually, used for binary data

Reconstruction error

$$Error = \boldsymbol{X} - \widehat{\boldsymbol{X}} = \begin{pmatrix} e_1^T \\ e_2^T \end{pmatrix} = n\begin{Bmatrix} e_{11} & e_{12} & \dots & e_{1K} \\ e_{21} & e_{22} & \dots & e_{2K} \end{Bmatrix}$$

Under and overcomplete

There are two types of autoencoders, undercomplete and overcomplete
Input dimension n and hidden dimension M:

$$if\ M < n,\ undercomplete$$
$$if\ M > n,\ overcomplete$$

Undercomplete reduce the information given, and it therefore need to learn the most important features of the data.
The overcomplete need to be forced to learn the important features by a regulator

Sparse Autoencoders (SAE)

A SAE is simply an autoencoder whose training criterion involves the sparsity penalty $\Omega_{sparsity}(\boldsymbol{h})$ at the hidden layer

$$J_1 = J + \beta_{sparsity}\,(\boldsymbol{h})$$

The sparsity penalty term makes the features (weights) learnt by the hidden-layer to be sparse

With sparsity constraint, one would constraint the neurons at the hidden layers to be inactive for most of the time. We say that the neuron is "active" when its output is close to 1 and the neuron is "inactive" when its output is close to 0

For a set $\{\boldsymbol{x}_p\}_{p=1}^{P}$ of input patterns, the average activation ρ_j of neuron j at the hidden-layer is given by

$$\rho_j = \frac{1}{P}\sum_{p=1}^{P} h_{pj} = \frac{1}{P}\sum_{p=1}^{P} f\left(\boldsymbol{x}_p^T \cdot \boldsymbol{w}_j + b\right)$$

Where h_{pj} is the activation of hidden neuron j for p th pattern, and $\boldsymbol{w}_j$ and b_j are the weight and biases of the hidden neuron j.

We would like to enforce the constrain $\rho_j = \rho$ such that the sparsity parameter ρ is set to a small value close to zero.

That is, most of the time, hidden neuron activations are maintained at ρ on average. By choosing a smaller value for ρ, the neurons are activated selectively to patterns and thereby learn sparse features.

To achieve sparse activations at the hidden-layer, the Kullback-Leibler (KL) divergence is used as the sparsity constraint:

$$D(\boldsymbol{h}) = \sum_{j=1}^{M} \rho \cdot \log\frac{\rho}{\rho_j} + (1-\rho)\cdot\log\frac{1-\rho}{1-\rho_j}$$

Where M is the number of hidden neurons and ρ is the sparsity parameter.

KL divergence measures the deviation of the distribution $\{\rho_j\}$ of activations at the hidden-layer from the uniform distribution ρ.

The KL divergence is minimum when $\rho_j = \rho$ for all j. That is, when the average activations are uniform and equal to very low value ρ.

The penalty is added to the cost function

$$J_1 = J + \beta \cdot D(\boldsymbol{h})$$

The gradients for can now be found

$$\nabla_{\boldsymbol{W}} D(\boldsymbol{h}) = (\nabla_{\boldsymbol{w_1}} D(\boldsymbol{h}) \quad \nabla_{\boldsymbol{w_2}} D(\boldsymbol{h}) \quad \dots \quad \nabla_{\boldsymbol{w_M}} D(\boldsymbol{h}))$$

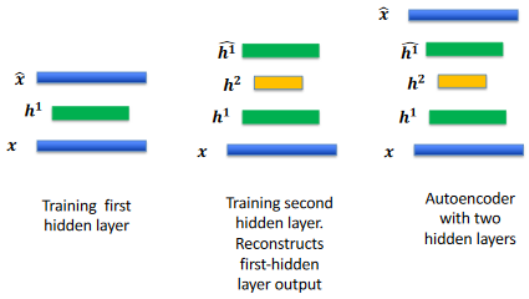
$$\nabla_{\boldsymbol{b}} D(\boldsymbol{h}) = (\nabla_{\boldsymbol{b_1}} D(\boldsymbol{h}) \quad \nabla_{\boldsymbol{b_2}} D(\boldsymbol{h}) \quad \dots \quad \nabla_{\boldsymbol{b_M}} D(\boldsymbol{h}))$$

And we update the learning algorithm

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha\left(\nabla_{\boldsymbol{W}} J + \beta \cdot \nabla_{\boldsymbol{W}} D(\boldsymbol{h})\right)$$

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha\left(\nabla_{\boldsymbol{b}} J + \beta \cdot \nabla_{\boldsymbol{b}} D(\boldsymbol{h})\right)$$

Deep stacked Autoencoders



Deep autoencoders can be built by stacking autoencoders one after the other. Training of deep autoencoders is done in a step-by-step fashion on layer at time.

After training the first level of denoising autoencoder, the resulting hidden representation is used to train a second level of the denoising encoder. The second level hidden representation can be used to train the third level of the encoders. This process is repeated and deep stacked autoencoder can be realized.

After training a stacked autoencoder, an output layer may be added on the top of the stacked hidden layers for classification/regression. The final layer is a supervised layer such as a Softmax for classification or a linear layer for regression.
The parameters of the feedforward network can be fine-tuned to minimize the error in regressing/classifying the targets by supervised gradient descent learning.