Multi-Layer Perceptron

Mohammad Emtiyaz Khan
EPFL

Dec. 3, 2015

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Multi-Layer Perceptron (MLP)

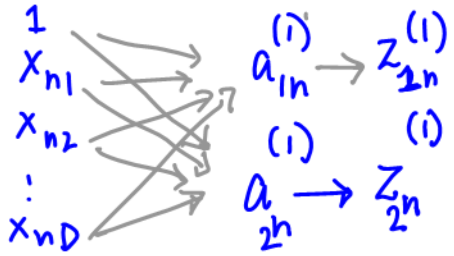
$(y_n, \underline{x}_n)$

This is also known as feed-forward neural network and can be represented graphically as follows:

K'th layer

deep →

$$\mathbf{x}_n \to \mathbf{a}_n^{(1)} \to \mathbf{z}_n^{(1)} \to \mathbf{a}_n^{(2)} \to \mathbf{z}_n^{(2)} \to \ldots \to \mathbf{z}_n^{(K-1)} \to \mathbf{y}_n$$

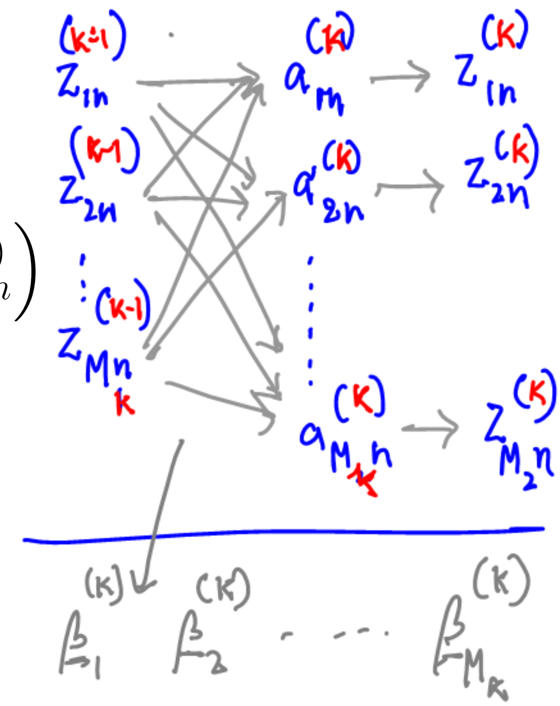


$a^{(1)}_{mn} = \underline{\beta}_m^T \tilde{\underline{x}}_n$

$z^{(1)}_{mn} = h(a^{(1)}_{mn})$

where $\{y_n, \mathbf{x}_n\}$ is the n'th input-output pair, $\mathbf{z}_n^{(k)}$ is the k'th hidden vector, $\mathbf{a}_n^{(k)}$ is the corresponding activation. There are a total of K layers.

For the k'th layer, we obtain the m'th activation $a_{mn}^{(k)}$ and the corresponding hidden variable $z_{mn}^{(k)}$, as shown below:



$$a_{mn}^{(k)} = \left(\boldsymbol{\beta}_m^{(k)}\right)^T \mathbf{z}_n^{(k-1)}, \quad z_{mn}^{(k)} = h\left(a_{mn}^{(k)}\right)$$

where $\mathbf{z}_n^{(k-1)}$ is the hidden vector for the previous layer. For the first layer, we set $\mathbf{z}_n^{(0)} = \mathbf{x}_n$. For the last layer, we use a link function to map $\mathbf{z}_n^{(K-1)}$ to the output $\mathbf{y}_n$.

Note that a 1-Layer MLP is simply a generalization of linear/logistic regression.

$z_n^{(K-1)} \longrightarrow y_n$

link function

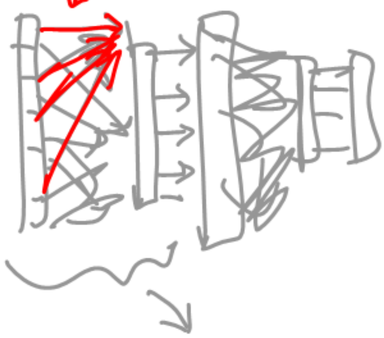
e.g. GLM

Defining $\mathbf{B}^{(k)}$ as a matrix with rows $(\boldsymbol{\beta}_m^{(k)})^T$, we can express the computation of activation and hidden vectors as follows:

$$\begin{bmatrix} \leftarrow & \beta_1^{(k)T} & \rightarrow \\ & \vdots & \\ \leftarrow & \beta_{M_k}^{(k)T} & \rightarrow \end{bmatrix} \triangleq B^{(k)}$$

$[\beta_1^{(1)}]^T$

$$\mathbf{a}_n^{(k)} = \mathbf{B}^{(k)}\mathbf{z}_n^{(k-1)}, \quad \mathbf{z}_n^{(k)} = h\left(\mathbf{a}_n^{(k)}\right)$$



In a more compact notation, we can express the input-output relationship as follows:

$$\hat{y}_n = g((\boldsymbol{\beta}^{(K-1)})^T * h(\mathbf{B}^{(K-2)} * h(* \ldots * h(\mathbf{B}^{(1)} * \mathbf{x}_n))),$$

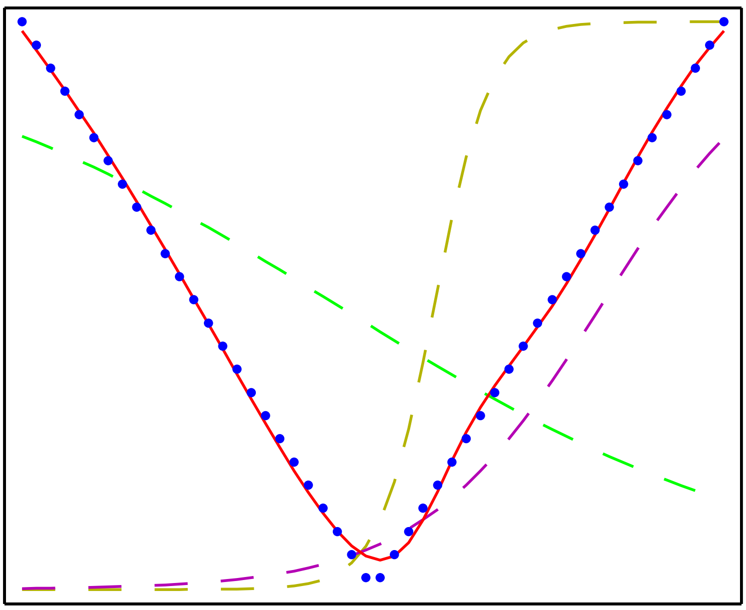
where g is an appropriate link function to match the output.

$$\sum_{n=1}^{N} (y_n - \hat{y}_n)^2$$

$$f(B^{(1)}, \ldots, {}^{(k-1)})$$

$$\frac{\partial f}{\partial B^{(k)}}$$

An illustration below shows reconstruction of the function $|x|$ at $N = 50$ data points sampled at the blue dots. The trained network has 2 layers and 3-hidden variables with $tanh()$ activation function.



Optimization and Back-propagation

We can learn parameters $\mathbf{B}$ using stochastic gradient-descent.

$$\mathcal{L} = \sum_{n=1}^{N} \left(y_n - f(\underline{x}_n)\right)^2$$

$$\sum_{n=1}^{N} \left(y_n - g\left(B^K * h\left(B^{K-1} * \cdots (\underline{x}_n)\right)\right)\right)^2$$

1,00,0,000

$K \times H \times 256 \times 256$

10,000

$$B = \{B^{(1)}, B^{(2)}, \ldots B^{(K)}\}$$

Gradient computation can be complicated due to the *deep* structure of the network. We can use back-propagation to simplify the computation. The key-idea is to express the derivatives in terms of activations $\mathbf{a}_n^{(k)}$ and hidden variables $\mathbf{z}_n^{(k)}$ using the chain rule. Below is the outline of the algorithm:

$$\frac{\partial \mathcal{L}}{\partial B} = \sum_{n=1}^{N} \left[10,000 \right]$$

Use the chain-rule → $\widehat{\dfrac{\partial \mathcal{L}}{\partial B}} = \left[10,000 \right]$

$$B_{i+1} = B_i - \delta_k \left.\widehat{\frac{\partial \mathcal{L}}{\partial B}}\right|_{B=B_i}$$

$$\left.\frac{\partial \mathcal{L}}{\partial B^{(K-1)}}\right|_{B^{(K-1)}=B_i^{(K-1)}}$$

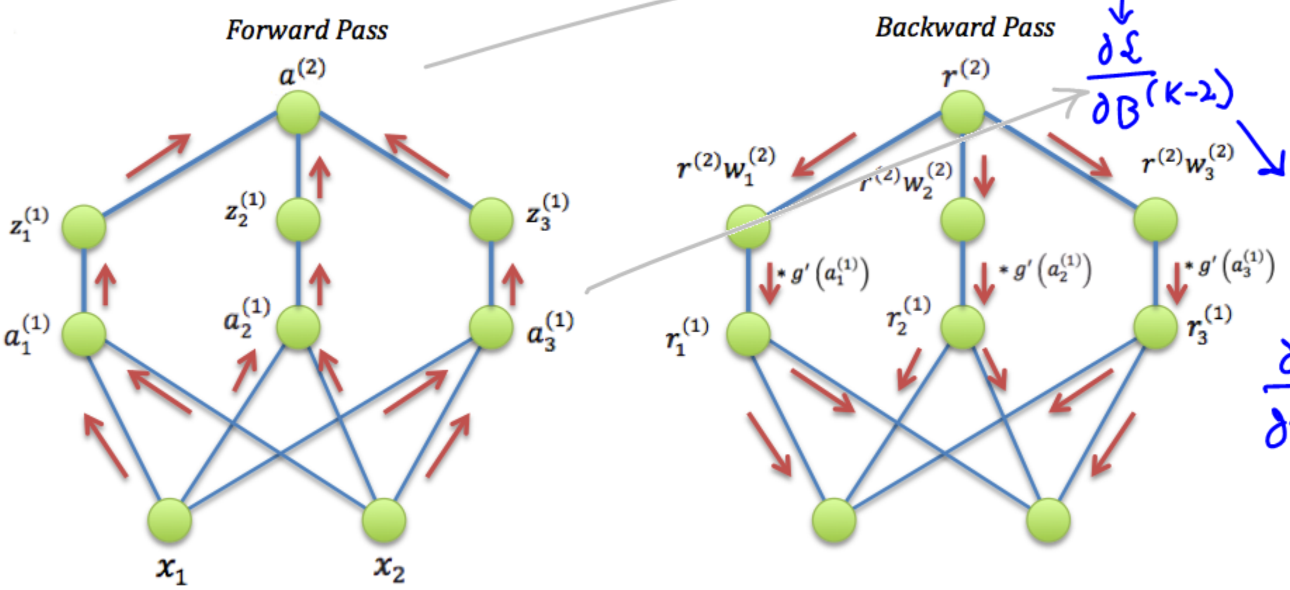
$$\frac{\partial \mathcal{L}}{\partial B^{(K-2)}}$$

$$\frac{\partial \mathcal{L}}{\partial B^{(1)}}$$

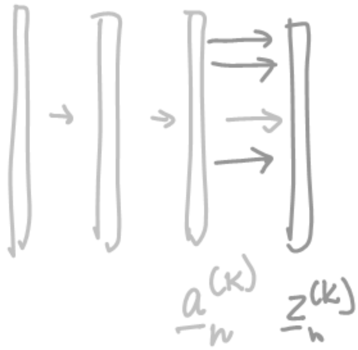
$f(\underline{x}_{n_i})$

B_i

x_{n_i}

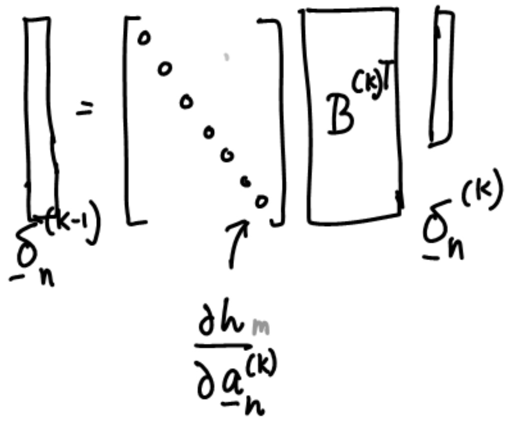


Step 1: Compute $\mathbf{a}_n^{(k)}$ and $\mathbf{z}_n^{(k)}$ using forward propagation.



Step 2: Compute $\boldsymbol{\delta}_n^{(k)} := \partial\mathcal{L}/\partial\mathbf{a}_n^{(k)}$ using backward propagation:

$$\boldsymbol{\delta}_n^{(k-1)} = \text{diag}\left[\mathbf{h}'(\mathbf{a}_n^{(k)})\right] \left(\mathbf{B}^{(k)}\right)^T \boldsymbol{\delta}_n^{(k)}$$



Step 3: Compute $\partial\mathcal{L}/\partial\mathbf{B}^{(k)}$ using the above derivatives.

$$\frac{\partial\mathcal{L}}{\partial\mathbf{B}^{(k)}} = \sum_n \boldsymbol{\delta}_n^{(k)} \left(\mathbf{z}_n^{(k)}\right)^T$$

Tricks

Obtaining a good generalization error with neural networks and avoiding overfitting requires a lot of hacks and tricks. A good summary of these are given in Bottou's paper "Stochastic gradient tricks". In addition, initialization seems to play a huge role in improving the performance. See the following paper "On the importance of initialization and momentum in deep learning" by Ilya Sutskever et. al.