Multilayer Perceptrons

Oliver Zhao

1 Deep Feedfoward Networks

Deep feedfoward networks, often called *multilayer perceptrons* (MLPs), are classic neural networks. The goal of the feedforward network is to approximate some ideal function f^*. This function $y = f^*(\mathbf{x})$ maps an input $\mathbf{x}$ to some output y. A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.

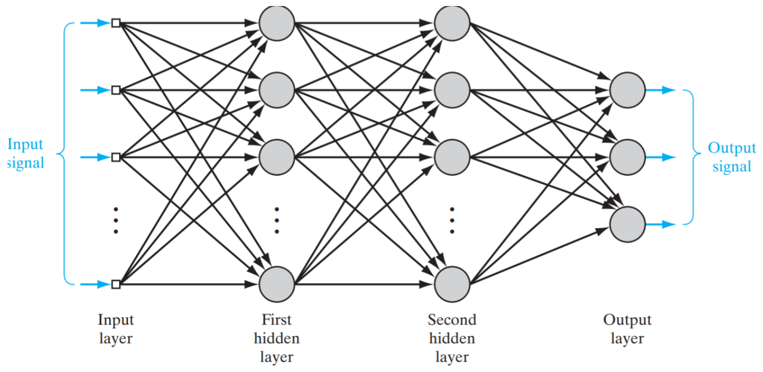


Figure 1: Generic MLP architecture.

Feedfoward neural networks are usually represented together by many different functions that describe individual layers. For example, a three-layer MLP can be described as

$$f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))), \tag{1.1}$$

Where $f^{(n)}$ is the function describing the nth layer of the feedforward network. When training this model, we drive $f(\mathbf{x})$ to match $f^*(\mathbf{x})$ with training data that give us noisy, approximate examples of $f^*(\mathbf{x})$ evaluated at different training points. Each example $\mathbf{x}$ has a corresponding label $y \approx f^*(\mathbf{x})$.

The learning algorithm must decide how to use the layers to produce the desired output y, but the training data only shows the desired output to the output layer. Consequently, the layers in between the input and output layers are called *hidden layers*.

Recall that an individual perceptron without a mapping function results in a linear model. Similarly, a deep feedforward network with only linear mappings can always be compressed into a single linear layer. To extend linear models to represent nonlinear functions of $\mathbf{x}$, we can apply the linear model (such as the output layer in a feedforward model) to a transformed input $\phi(\mathbf{x})$, where ϕ is a nonlinear transformation.

The strategy of deep learning is to learn ϕ, where we have a model $y = f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{w}) = \phi(\mathbf{x}; \boldsymbol{\theta})^T \mathbf{w}$. We now have the parameters $\boldsymbol{\theta}$ that we use to learn ϕ from a broad class of function, and parameters $\mathbf{w}$ that map $\phi(\mathbf{x})$ to the desired output. In a deep feedforward network, this mapping function ϕ defines a hidden layer.

2 Batch Learning and On-Line Learning

Consider a MLP with an input layer of source nodes, one or more hiddne layers, and an output layer consisting of one or more neurons. Let

$$\mathscr{T} = \{\mathbf{x}(n), \mathbf{d}(n)\}_{n=1}^{N} \tag{2.1}$$

Denote the *training sample* used to train the network. Let $y_j(n)$ denote the function signal produced at the output of neuron j in the output layer by the stimulus $\mathbf{x}(n)$ applied to the input layer. The *error signal* produced at the output of neuron j is defined by

$$e_j(n) = d_j(n) - y_j(n) \tag{2.2}$$

Where $d_j(n)$ is the ith element of the desired-response vector $\mathbf{d}(n)$. The *instantaneous error energy* of neuron j is defined by

$$\mathscr{E}_j(n) = \frac{1}{2} e_j^2(n) \tag{2.3}$$

Summing the error-energies of all the neurons in the output layer, we can define the *total instantaneous energy error* of the whole network as

$$\begin{aligned}\mathscr{E}(n) &= \sum_{j \epsilon C} \mathscr{E}_j(n) \\ &= \frac{1}{2} \sum_{j \epsilon C} e_j^2(n)\end{aligned} \tag{2.4}$$

Where the set C includes all the neurons in the output layer. With the training sample containing N samples, the *error energy averaged over the training sample* or *empirical risk* is defined as

$$\begin{aligned}\mathscr{E}_{\text{avg}}(N) &= \frac{1}{N} \sum_{n=1}^{N} \mathscr{E}(n) \\ &= \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \epsilon C} e_j^2(n)\end{aligned} \tag{2.5}$$

These error is used to guide the free parameters (i.e. synpatic weights) of the MLP when training the model.

2.1 Batch Learning

In batch learning, adjustments to the synaptic weights of the multilayer perceptron are performed after the presentation of all N examples in the training sample $\mathscr{T}$ that constitute one *epoch* of training. In other words, the average energy error $\mathscr{E}_{\text{avg}}$ is used.

The main benefit of batch learning during gradient descent is that we achieve an accurate estimation of the gradient vector and parallelization of the learning process. However, large batch sizes require significant storage requirements.

2.2 On-line Learning

In the on-line method, synaptic weights of the MLP are performed by an *example-by-example* basis. In other words, the cost function to be minimized is the total instantaneous error energy $\mathscr{E}(n)$. For training data that is highly redundant, on-line learning can take advantage of this redundancy by making adjustments for each example.

3 Back-Propagation Algorithm

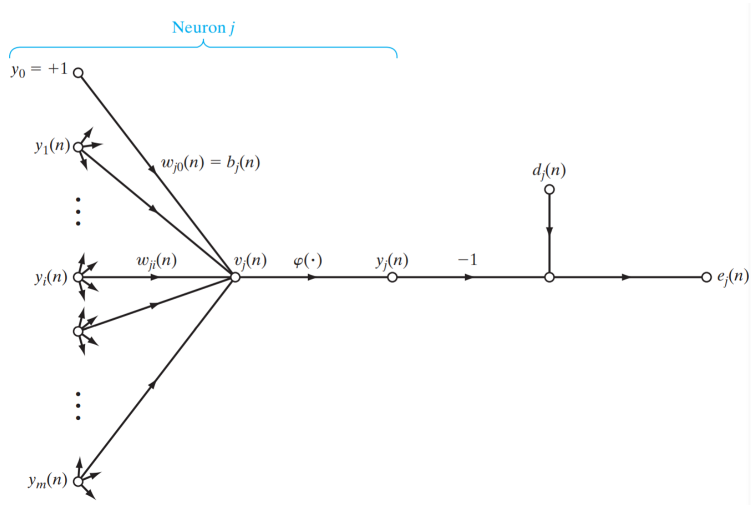


Figure 2: Signal flow for output neuron j.

Consider a neuron j being fed by a set of function signals produced by a preceding layer of neurons. This induced local field $v_j(n)$ is the input of the activation function associated with neuron j, or

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n)y_i(n) \tag{3.1}$$

Where m is the total number of inputs (excluding the bias) applied to neuron j. The synaptic weight w_{j0} equals the bias b_j applied to neuron j. Hence, the function signal $y_j(n)$ appearing at the output of neuron j at iteration n is

$$y_j(n) = \phi_j(v_j(n)). \tag{3.2}$$

The back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the partial derivative $\delta\mathscr{E}(n)/\delta w_{ji}(n)$. According to the chain rule, this gradient can be expressed as

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{3.3}$$

The partial derivative $\delta\mathscr{E}(n)/\delta w_{ji}(n)$ represents a *sensitivity factor*, determining the direction of search in weight space for the synaptic weight w_{ji}. Differentiating both sides of Equation (2.4) with respect to $e_j(n)$, we get

$$\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} = e_j(n) \tag{3.4}$$

Differentiating both sides of Equation (2.2) with respect to $y_j(n)$, we get

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{3.5}$$

Differentiating Equation (2.2) with respect to $v_j(n)$, we get

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \phi_j'(v_j(n)) \tag{3.6}$$

Where the prime signifies differentiation with respect to the argument. Finally, differentiating Equation (3.1) with respect to $w_{ji}(n)$ yields

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{3.7}$$

If we substitution Equation (3.4) and Equation (3.7) into Equation (3.3) yields

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\phi_j'(v_j(n))y_i(n) \tag{3.8}$$

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the *delta rule*, or

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} \tag{3.9}$$

Where η is the *learning-rate parameter* of the back-propagation algorithm. Accordingly, we can say that

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \tag{3.10}$$

Where the *local gradient* $\delta_j(n)$ is defined by

$$\begin{aligned}\delta_j(n) &= \frac{\partial \mathscr{E}(n)}{\partial v_j(n)} \\ &= \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n)\phi_j'(v_j(n))\end{aligned} \tag{3.11}$$

The local gradient points to the required changes in synaptic weights. According to Equation (3.11), the local gradient $\delta_j(n)$ for output neuron j is equal to the product of the corresponding error signal $e_j(n)$ for that neuron and the derivative $\phi_j'(v_j(n))$ of the associated activation function. We see that there are two potential cases for which Equation (3.11) is used. The neuron j may be either an output node or hidden node.

3.1 Case 1: Neuron j is an Output Node

In this case, it is straightforward to find $e_j(n)$ through Equation (2.2) and we can easily subsequently use Equation (3.11).

3.2 Case 2: Neuron j is a Hidden Node

When neuron j is located in a hidden layer, there is no specific desired response to that neuron. This, the error signal for a hidden neuron would have to be determined recursively and working backgrounds in terms of error signals of all neurons to which the hidden neuron is directly connected.

According to Equation (3.11), the local gradient $\delta_j(n)$ for a hidden neuron j can be redefined as

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} \phi_j(v_j(n))\end{aligned} \tag{3.12}$$

Where in the second line we have used Equation (3.6). To calculate the partial derivative $\partial \mathscr{E}(n)/\partial y_j(n)$, we first note that

$$\mathscr{E}(n) = \frac{1}{2} \sum_{k \epsilon C} e_k^2(n) \tag{3.13}$$

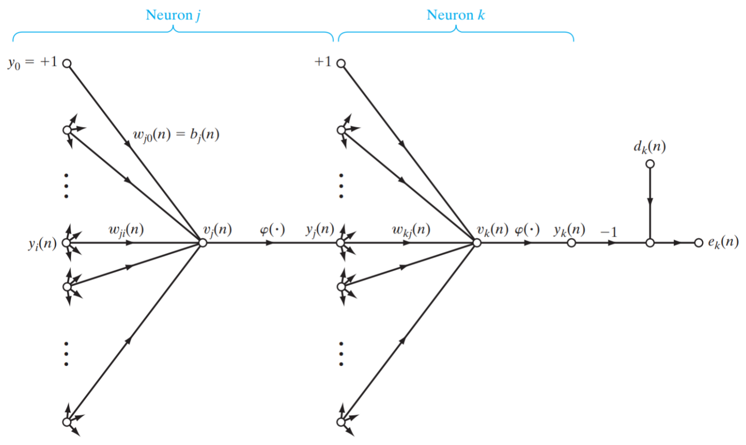


Figure 3: Signal flow of output neuron k connected to hidden neuron j.

Where we use the index k instead of index j from Equation (2.4) to avoid confusion with the use of index j that refers to a hidden neuron under case 2. Differentiating Equation (3.13) with respect to the function signal $y_j(n)$, we get

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \tag{3.14}$$

Next we use the chain rule for the partial derivative $\partial e_k(n)/\partial y_j(n)$ and rewrite Equation (3.14) as

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \tag{3.15}$$

However, note that

$$\begin{aligned} e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \phi_k(v_k(n)) \end{aligned} \tag{3.16}$$

Hence,

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\phi_k'(v_k(n)) \tag{3.17}$$

We also note that for neuron k, the induced local field is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{3.18}$$

Where m is the total number of inputs (excluding the bias) applied to neuron k. Here again, the synaptic weight $w_{k0}(n)$ is equal to the bias $b_k(n)$ applied to neuron k and the corresponding put is fixed at the value $+1$. Differentiating Equation (3.18) with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{3.19}$$

By substituting Equation (3.18) and Equation (3.19) into Equation (3.15), we get the desired partial derivative

$$\begin{aligned} \frac{\partial \mathscr{E}(n)}{\partial y_j(n)} &= -\sum_k e_k(n) \phi_k'(v_k(n)) w_{kj}(n) \\ &= -\sum_k \delta_k(n) w_{kj}(n) \end{aligned} \tag{3.20}$$

Finally, by using Equation (3.20) in Equation (3.12), we get the back-propagation formula for the local gradient $\delta_j(n)$, described by

$$\delta_j(n) = \phi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \tag{3.21}$$

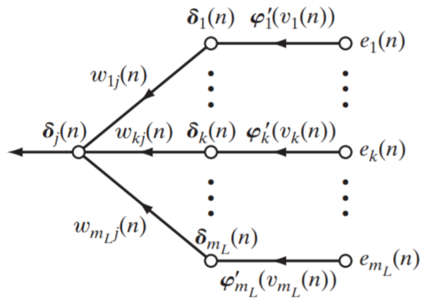


Figure 4: Signal flow for back-propagation of errors.

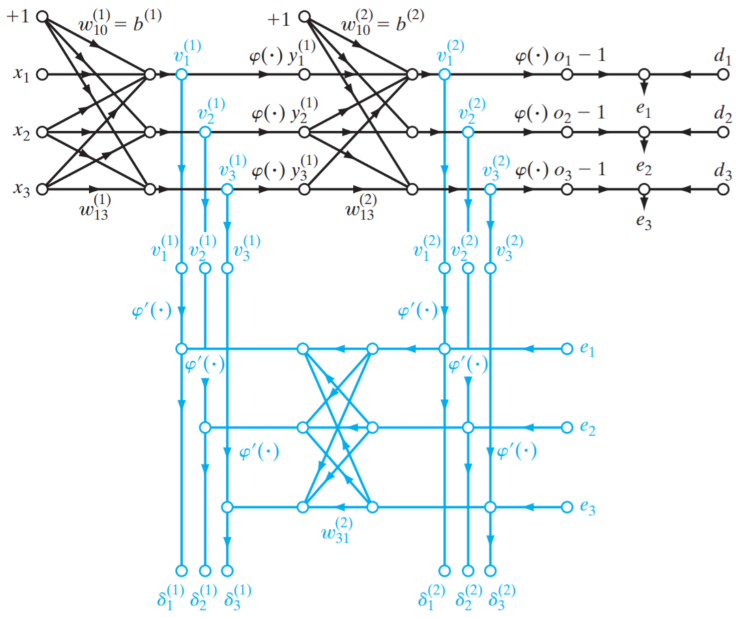
3.3 Summary

Consequently, we see that the weight correction $\Delta w_{ji}(n)$ is defined by the delta rule as

$$\begin{pmatrix}\text{Weight}\\ \text{Correction}\\ \Delta w_{ji}(n)\end{pmatrix} = \begin{pmatrix}\text{Learning-}\\ \text{Rate Parameter}\\ \eta\end{pmatrix} \times \begin{pmatrix}\text{Local}\\ \text{Gradient}\\ \delta_j(n)\end{pmatrix} \times \begin{pmatrix}\text{Input Signal}\\ \text{of Neuron } j\\ y_i(n)\end{pmatrix} \quad (3.22)$$

Under two conditions

1. If neuron j is an output node, $\delta_j(n)$ equals the product of the derivative $\phi_j'(v_j(n))$ and the error signal $e_j(n)$, both of which are associated with neuron j.
2. If neuron j is a hidden node, $\delta_j(n)$ equals the product of the associated derivative $\phi_j'(v_j(n))$ and the weighted sum of the δs computed for the neurons in the next hidden or output layer that are connected to neuron j.



$+1$
$w_{10}^{(1)} = b^{(1)}$
x_1
x_2
x_3
$w_{13}^{(1)}$
$v_1^{(1)}$
$v_2^{(1)}$
$v_3^{(1)}$
$\varphi(\cdot)\ y_1^{(1)}$
$\varphi(\cdot)\ y_2^{(1)}$
$\varphi(\cdot)\ y_3^{(1)}$
$+1$
$w_{10}^{(2)} = b^{(2)}$
$w_{13}^{(2)}$
$v_1^{(2)}$
$v_2^{(2)}$
$v_3^{(2)}$
$\varphi(\cdot)\ o_1 \quad -1$
$\varphi(\cdot)\ o_2 \quad -1$
$\varphi(\cdot)\ o_3 \quad -1$
d_1
d_2
d_3
e_1
e_2
e_3
$\varphi'(\cdot)$
$w_{31}^{(2)}$
$\delta_1^{(1)}$ $\delta_2^{(1)}$ $\delta_3^{(1)}$
$\delta_1^{(2)}$ $\delta_2^{(2)}$ $\delta_3^{(2)}$

Figure 5: Signal flow of back-propagation learning. Black = forward pass. Blue = backward pass.

4 Regularization

4.1 Dropout

Dropout layers are typically implemented in the multilayer perceptron portion of the convolutional neural network, rather than immediately after the convolution layers. Dropout is when a set fraction of nodes are randomly removed from the neural network during each training iteration. By reducing the number of nodes, we reduce the number of weights (parameters), serving the dual purpose of making the model train faster and reducing overfitting. Intuitively, this builds up redundancy in the neural network to prevent the model from becoming overly reliant on a particular set of nodes. However, it is unclear if this is actually the case. An example of a neural network before and after applying the dropout method is shown in the figure below.

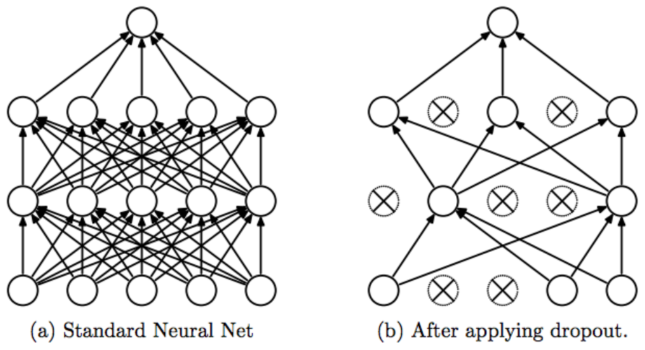


Figure 6: Max pooling and average pooling result in different outputs.

4.2 L1 and L2 Regularization

Analogous to lasso (L1) regression and ridge (L2) regression, additional penalty terms can be added to the cost function, where

$$\begin{aligned} &\text{Loss Function} + \lambda \sum_{j=1}^{p} |\beta_j| \qquad \text{(L1 Regularization)} \\ &\text{Loss Function} + \lambda \sum_{j=1}^{p} \beta_j^2 \qquad \text{(L2 Regularization)} \end{aligned} \tag{4.1}$$

Note that λ specifies how strong the regularization is. Meanwhile, the terms

β_j typically refer to the parameters of the neural network (i.e. the weights of each node). By reducing the values of the weights in the network, the impact of individual hidden nodes tends to decrease, thereby decreasing model complexity and overfitting. Less commonly, they may be represent the output values of each individual node.

4.3 Batch Normalization

Batch normalization almost always improves model training speed and stability, although this is discovered through empirical means rather than any rigorous theoretical derivation. One common proposal is that batch normalization improves model training and stability by reducing internal covariate shift, although this has been subject to intense scrutiny.

Batch normalization introduces two new sets of parameters: standard deviation (γ) and mean (β). This regularization method is quite simple, and fixes and rescales the means and variances of each layer's inputs. Note that older neural network architectures such as VGG16 do not have batch normalization because it did not exist at the time.

For values of x over a mini-batch $\mathscr{B} = \{x_1, \ldots, x_m\}$, we find the mini-batch mean $\mu_{\mathscr{B}}$ and mini-batch variance $\sigma^2_{\mathscr{B}}$, where

$$\begin{aligned} \mu_{\mathscr{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma^2_{\mathscr{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathscr{B}})^2 \end{aligned} \tag{4.2}$$

Then we just normalize the values of x_i to generate $\hat{x_i}$, where

$$\hat{x_i} \leftarrow \frac{x_i - \mu_{\mathscr{B}}}{\sqrt{\sigma^2_{\mathscr{B}} + \epsilon}} \tag{4.3}$$

ϵ is some small value to prevent division by zero. Finally, we scale and shift the values where

$$y_i \leftarrow \gamma \hat{x_i} + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i). \tag{4.4}$$

4.4 Noise Robustness

Adding a moderate amount of (often Gaussian) noise into the model can also reduce overfitting. Noise may be added to various components of the model and data. For example noise may be added to the input data (i.e. tabular data, images, audio, etc) or the final outputs of the model. Noise may also be added to the weights of each parameter or the gradients when updating weights. Intuitively, adding noise to the training process prevents overfitting by disallowing the model from focusing on potentially misleading features or details.

5 Residual Connections

In traditional neural networks, each layer feeds directly into the next layer. In residual blocks, the outputs of one layer feeds into another layer that is typically at least 2 layers forward. One motivation for these skip-connections is to reduce the issue of vanishing gradients, where the gradients become too small for meaningful weight updates. The backpropagation algorithm is nearly identical, except that some weights are fixed and not updated due to the skip-connections.

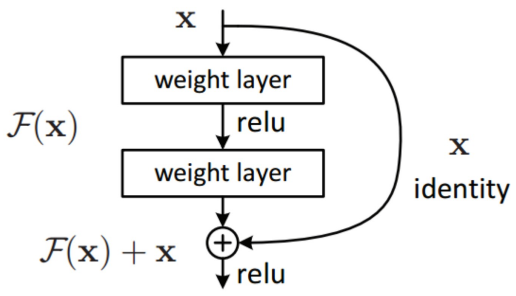


Figure 7: Residual block.