7 The Multilayer Perceptron

7.1 The multilayer perceptron – general

The "multilayer perceptron" (MLP) is a design that overcomes the shortcomings of the simple perceptron. The MLP can solve general nonlinear classification problems. A MLP is a hierarchical structure of several "simple" perceptrons (with smooth transfer functions). For instance, a "one hidden layer" MLP with a logistic output unit looks like this, see figures in e.g. (Bishop 1995) or (Haykin 1999),

$$\hat{y}(\mathbf{x}) = \frac{1}{1+\exp[-a(\mathbf{x})]} \tag{7.1}$$

$$a(\mathbf{x}) = \sum_{j=0}^{M} v_j h_j(\mathbf{x}) = \mathbf{v}^T \mathbf{h}(\mathbf{x}) \tag{7.2}$$

$$h_j(\mathbf{x}) = \sum_{k=0}^{D} \phi(w_{jk} x_k) = \phi(\mathbf{w}_j^T \mathbf{x}) \tag{7.3}$$

where the transfer function, or activation function, $\phi(z)$ typically is a sigmoid of the form

$$\phi(z) = \tanh(z), \tag{7.4}$$

$$\phi(z) = \frac{1}{1+e^{-z}}. \tag{7.5}$$

The former type, the hyperbolic tangent, is the more common one and it makes the training a little easier than if you use a logistic function.

The logistic output unit (7.1) is the correct one to use for a classification problem.

If the idea is to model a function (i.e. nonlinear regression) then it is common to use a linear output unit

$$\hat{y}(\mathbf{x}) = a(\mathbf{x}). \tag{7.6}$$

7.2 Training an MLP – Backpropagation

The perhaps most straightforward way to design a training algorithm for the MLP is to use the gradient descent algorithm. What we need is for the model output $\hat{y}$ to be differentiable with respect to all the parameters

w_{jk} and v_j. We have a training data set $\mathcal{X} = \{\mathbf{x}(n), y(n)\}_{n=1,\ldots,N}$ with N observations, and we denote all the weights in the network by $\mathbf{W} = \{\mathbf{w}_j, \mathbf{v}\}$. The batch form of gradient descent then goes as follows:

1. Initialize $\mathbf{W}$ with e.g. small random values.
2. Repeat until convergence (either when the error E is below some preset value or until the gradient $\nabla_W E$ is smaller than a preset value), t is the iteration number
   - 2.1 Compute the update $\Delta\mathbf{W}(t) = -\eta\nabla_W E(t) = \eta\sum_{n=1}^{N} e(n,t)\nabla_W \hat{y}(n,t)$ where $e(n,t) = (y(n) - \hat{y}(n,t))$
   - 2.2 Update the weights $\mathbf{W}(t+1) = \mathbf{W}(t) + \Delta\mathbf{W}(t)$
   - 2.3 Compute the error $E(t+1)$

As an example, we compute the weight updates for the special case of a multilayer perceptron with one hidden layer, using the transfer function $\phi(z)$ (e.g. $\tanh(z)$), and one output unit with the transfer function $\theta(z)$ (e.g. logistic or linear). We use half the mean square error

$$E = \frac{1}{2N}\sum_{n=1}^{N}\left[y(n) - \hat{y}(n)\right]^2 = \frac{1}{2N}\sum_{n=1}^{N} e^2(n), \tag{7.7}$$

and the following notation

$$\begin{aligned}
\hat{y}(\mathbf{x}) &= \theta\left[a(\mathbf{x})\right], && (7.8)\\
a(\mathbf{x}) &= v_0 + \sum_{j=1}^{M} v_j h_j(\mathbf{x}), && (7.9)\\
h_j(\mathbf{x}) &= \phi\left[b_j(\mathbf{x})\right], && (7.10)\\
b_j(\mathbf{x}) &= w_{j0} + \sum_{k=1}^{D} w_{jk} x_k. && (7.11)
\end{aligned}$$

Here, v_j are the weights between the hidden layer and the output layer, and w_{jk} are the weights between the input and the hidden layer.

For weight v_i we get

$$\begin{aligned}
\frac{\partial E}{\partial v_i} &= -\frac{1}{N}\sum_{n=1}^{N} e(n)\frac{\partial \hat{y}(n)}{\partial v_i}\\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right]\frac{\partial a(n)}{\partial v_i}
\end{aligned}$$

$$= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] h_i(n) \tag{7.12}$$

$$\Rightarrow \Delta v_i = -\eta\frac{\partial E}{\partial v_i} = \eta\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] h_i(n) \tag{7.13}$$

with the definition $h_0(n) \equiv 1$. If the output transfer function is linear, i.e. $\theta(z) = z$, then $\theta'(z) = 1$. If the output function is logistic, i.e. $\theta(z) = [1 + \exp(-z)]^{-1}$, then $\theta'(z) = \theta(z)[1 - \theta(z)]$.

For weight w_{il} we get

$$\begin{aligned}
\frac{\partial E}{\partial w_{il}} &= -\frac{1}{N}\sum_{n=1}^{N} e(n)\frac{\partial \hat{y}(n)}{\partial w_{il}} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right]\frac{\partial a(n)}{\partial w_{il}} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] v_i\frac{\partial h_i(n)}{\partial w_{il}} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] v_i\phi'\left[b_i(n)\right]\frac{\partial b_i(n)}{\partial w_{il}} \\
&= -\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] v_i\phi'\left[b_i(n)\right] x_l \qquad (7.14) \\
\Rightarrow \Delta w_{il} &= -\eta\frac{\partial E}{\partial w_{il}} = \eta\frac{1}{N}\sum_{n=1}^{N} e(n)\theta'\left[a(n)\right] v_i\phi'\left[b_i(n)\right] x_l(n) \qquad (7.15)
\end{aligned}$$

with the definition $x_0(n) \equiv 1$. If the hidden unit transfer function is the hyperbolic tangent function, i.e. $\phi(z) = \tanh(z)$, then $\phi'(z) = 1 - \phi^2(z)$.

This gradient descent method for updating the weights has become known as the "backpropagation" training algorithm. The motivation for the name becomes clear if we introduce the notation

$$\delta(n) = e(n)\theta'\left[a(n)\right], \tag{7.16}$$

$$\delta_i(n) = \delta(n)v_i\phi'\left[b_i(n)\right], \tag{7.17}$$

which enables us to write

$$\Delta v_i = \eta\frac{1}{N}\sum_{n=1}^{N}\delta(n)h_i(n), \tag{7.18}$$

$$\Delta w_{il} = \eta\frac{1}{N}\sum_{n=1}^{N}\delta_i(n)x_l(n), \tag{7.19}$$

which is very similar to the good old LMS algorithm. Expression (7.17) corresponds to a propagation of $\delta(n)$ backwards through the network.

The gradient descent learning algorithm corresponds to backprop in its batch form, where the update is computed using all the available training data. There is also an "on-line" version where the updates are done after each pattern $\mathbf{x}(n)$ without averaging over all patterns.

Bishop (1995) and Haykin (1999) discuss variants of backprop, e.g. using a "momentum" term and adaptive learning rate η. Backprop with "momentum" amounts to using an update equal to

$$\Delta \mathbf{W}(t) = -\eta \nabla_W E(t) + \alpha \Delta \mathbf{W}(t-1), \tag{7.20}$$

where $0 < \alpha < 1$. This enables the learning to gain momentum (hence the name). If there are several updates in the same direction, then the effective learning rate is increased. If there are several updates in opposite directions then the effective learning rate is decreased.

Backpropagation is, in general, a very slow learning algorithm – even with momentum – and there are many better algorithms which we discuss below. However, backpropagation was very important in the beginning of the 1980:ies because is was used to demonstrate that multilayer perceptrons can learn things.

7.3 RPROP

A very useful gradient based learning algorithm, that is not discussed in Bishop (Bishop 1995), is the "resilient backropagation" (RPROP) algorithm (Riedmiller & Braun 1993). It uses individual adaptive learning rates combined with the so-called "Manhattan" update step.

The standard backpropagation updates the weights according to

$$\Delta w_{il} = -\eta \frac{\partial E}{\partial w_{il}}. \tag{7.21}$$

The "Manhattan" update step, on the other hand, uses only the sign of the derivative (the reason for the name should be obvious to anyone who has seen a map of Manhattan), i.e.

$$\Delta w_{il} = -\eta \text{sign} \left[\frac{\partial E}{\partial w_{il}} \right]. \tag{7.22}$$

The RPROP algorithm combines this Manhattan step with individual learning rates for each weight, and the algorithm goes as follows

$$\Delta w_{il}(t) = -\eta_{il}(t)\text{sign}\left[\frac{\partial E}{\partial w_{il}}\right], \tag{7.23}$$

where w_{il} denotes any weight in the network (e.g. also hidden to output weights).

The learning rate $\eta_{il}(t)$ is adjusted according to

$$\eta_{il}(t) = \begin{cases} \gamma^+\eta_{il}(t-1) & \text{if } \partial_{il}E(t)\cdot\partial_{il}E(t-1) > 0 \\ \gamma^-\eta_{il}(t-1) & \text{if } \partial_{il}E(t)\cdot\partial_{il}E(t-1) < 0 \end{cases} \tag{7.24}$$

where γ^+ and γ^- are different growth/shrinking factors ($0 < \gamma^- < 1 < \gamma^+$). Values that have worked well for me are $\gamma^- = 0.5$ and $\gamma^+ = 1.2$, with limits such that $10^{-6} \leq \eta_{ij}(t) \leq 50$. I have used the short notation $\partial_{il}E(t) \equiv \frac{\partial E(t)}{\partial w_{il}}$. The RPROP algorithm is a batch algorithm, since the learning rate update becomes noisy and uncertain if the error E is evaluated over only a single pattern.

The RPROP algorithm is implemented in the MATLAB neural network toolbox.

7.4 Second order learning algorithms

Backpropagation, i.e. gradient descent, is a *first order* learning algorithm. This means that it only uses information about the first order derivative when it minimizes the error. The idea behind a first order algorithm can be illustrated by expanding the error E in a Taylor series around the current weight position $\mathbf{W}$

$$E(\mathbf{W}+\Delta\mathbf{W}) = E(\mathbf{W}) + \nabla_W E(\mathbf{W})^T\Delta\mathbf{W} + \mathcal{O}\left(\|\Delta\mathbf{W}\|^2\right). \tag{7.25}$$

The vector $\mathbf{W}$ contains all the weights w_{jk} and v_j (and others if we are considering other network architectures) and we require that $\Delta\mathbf{W}$ is small. The notation $\mathcal{O}\left(\|\Delta\mathbf{W}\|^2\right)$ denotes all the terms that contains the small weight step $\Delta\mathbf{W}$ multiplied by itself at least once, and by "small" we mean that $\Delta\mathbf{W}$ is so small that the gradient term $\nabla_W E(\mathbf{W})\Delta\mathbf{W}$ is larger than the sum of the higher order terms. In that case we can ignore the higher order terms and write

$$E(\mathbf{W}+\Delta\mathbf{W}) \approx E(\mathbf{W}) + \nabla_W E(\mathbf{W})^T\Delta\mathbf{W}. \tag{7.26}$$

Now, we want to change the weights so that the new error $E(\mathbf{W}+\Delta\mathbf{W})$ is smaller than the current error $E(\mathbf{W})$. One way to guarantee this is to set the weight update $\Delta\mathbf{W}$ proportional to the negative gradient, i.e. $\Delta\mathbf{W} = -\eta\nabla_W E(\mathbf{W})$, in which case we have

$$E(\mathbf{W}+\Delta\mathbf{W}) \approx E(\mathbf{W}) - \eta\|\nabla_W E(\mathbf{W})\|^2 \leq E(\mathbf{W}). \tag{7.27}$$

However, this of course requires that $\Delta\mathbf{W}$ is so small that we can motivate (7.26).

We can extend this and also consider the second order term in the Taylor expansion. That is

$$E(\mathbf{W}+\Delta\mathbf{W}) = E(\mathbf{W})+\nabla_W E(\mathbf{W})^T\Delta\mathbf{W}+\frac{1}{2}\Delta\mathbf{W}^T H(\mathbf{W})\Delta\mathbf{W}+\mathcal{O}\left(\|\Delta\mathbf{W}\|^3\right), \tag{7.28}$$

where

$$H(\mathbf{W}) = \nabla_W\nabla_W^T E(\mathbf{W}) \tag{7.29}$$

is the Hessian matrix with elements $H_{ij}(\mathbf{W}) = \frac{\partial^2 E(\mathbf{W})}{\partial w_i \partial w_j}$. The Hessian is symmetric (all eigenvalues are consequently real and we can diagonalize H with an orthogonal transformation).

If we can ignore the higher order terms in (7.28) then we have

$$E(\mathbf{W}+\Delta\mathbf{W}) \approx E(\mathbf{W}) + \nabla_W E(\mathbf{W})^T\Delta\mathbf{W} + \frac{1}{2}\Delta\mathbf{W}^T H(\mathbf{W})\Delta\mathbf{W}. \tag{7.30}$$

We want to change the weights so that the new error $E(\mathbf{W}+\Delta\mathbf{W})$ is smaller than the current error $E(\mathbf{W})$. Furthermore, we want it to be as small as possible. That is, we want to minimize $E(\mathbf{W}+\Delta\mathbf{W})$ by choosing $\Delta\mathbf{W}$ appropriately. The requirement that we end up at an extremum point is

$$\begin{aligned} \nabla_{\Delta W} E(\mathbf{W}+\Delta\mathbf{W}) &= \mathbf{0} \\ \Rightarrow \nabla_W E(\mathbf{W}) + H(\mathbf{W})\Delta\mathbf{W} &= \mathbf{0}, \end{aligned} \tag{7.31}$$

which yields the optimum weight update as

$$\Delta\mathbf{W} = H^{-1}(\mathbf{W})\nabla_W E(\mathbf{W}). \tag{7.32}$$

To guarantee that this is a mininum point we must also require that the Hessian matrix is positive definite. This means that all the eigenvalues of the Hessian matrix must pe positive. If any of the eigenvalues are zero then we have a saddle point and $H(\mathbf{W})$ is not invertible. If any of the eigenvalues of $H(\mathbf{W})$ are negative then we have a maximum point for at least one of the weights w_j and (7.32) will actually move away from the minimum!

The update step (7.32) is usually referred to as a "Newton-step", and the minimization method that uses this update step is the Newton algorithm.

Some problems with "vanilla" Newton learning (7.32) are:

- The Hessian matrix may not be invertible, i.e. some of the eigenvalues are zero.
- The Hessian matrix may have negative eigenvalues.
- The Hessian matrix is expensive to compute and also expensive to invert. The learning may therefore be slower than a first order method.

The first two problems are handled by regularizing the Hessian, i.e. by replacing $H(\mathbf{W})$ by $H(\mathbf{W})+\lambda\mathbf{I}$. This effectively filters out all eigenvalues that are smaller than λ. The third problem is handled by "Quasi-Newton" methods that iteratively try to estimate the inverse Hessian using expressions of the form $H^{-1}(\mathbf{W}+\Delta\mathbf{W}) \approx H^{-1}(\mathbf{W}) + \text{correction}$.

7.4.1 The Levenberg-Marquardt algorithm

The Levenberg-Marquardt, see e.g. (Bishop 1995), is a very efficient second order learning algorithm that builds on the assumption that the error E is a quadratic error (which it usually is), like half the mean square error. In this case we have

$$H_{ij} = \frac{1}{2}\frac{\partial^2 \text{MSE}}{\partial w_i \partial w_j} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial \hat{y}(n)}{\partial w_i}\frac{\partial \hat{y}(n)}{\partial w_j} + \frac{1}{N}\sum_{n=1}^{N} e(n)\frac{\partial^2 \hat{y}(n)}{\partial w_i \partial w_j}. \qquad (7.33)$$

If the residual $e(n)$ is symmetrically distributed around zero and small then we can assume that the second term in (7.33) is very small compared to the first term. If so, then we can approximate

$$\begin{aligned} H_{ij} &\approx \frac{1}{N}\sum_{n=1}^{N}\frac{\partial \hat{y}(n)}{\partial w_i}\frac{\partial \hat{y}(n)}{\partial w_j} \\ H(\mathbf{W}) &\approx \frac{1}{N}\sum_{n=1}^{N}\mathbf{J}(n)\mathbf{J}^T(n), \end{aligned} \qquad (7.34)$$

where we have used the notation

$$\mathbf{J}(n) = \nabla_W \hat{y}(n) \qquad (7.35)$$

and we will refer to $\mathbf{J}$ as the "Jacobian". This approximation is not as costly to compute as the exact Hessian, since no second order derivatives are needed.

The fact that the Hessian is approximated by a sum of outer products $\mathbf{J}\mathbf{J}^T$ means that the rank of H is at most N. That is, there must be at least as many observations as there are weights in the network (which intuitively makes sense).

This approximation of the Hessian is used in a Newton step together with a regularization term, so that the Levenberg-Marquardt update is

$$\Delta \mathbf{W} = \left[\frac{1}{N} \sum_{n=1}^{N} \mathbf{J}(n)\mathbf{J}^T(n) + \lambda \mathbf{I} \right]^{-1} \nabla_W E(\mathbf{W}). \tag{7.36}$$

The Levenberg-Marquardt update is a very useful learning algorithm, and it represents a combination of gradient descent and a Newton step search. We have that

$$\Delta \mathbf{W} \rightarrow \begin{cases} \frac{1}{\lambda} \nabla E(\mathbf{W}) & \text{when } \lambda \rightarrow \infty \\ \left[\frac{1}{N} \sum_n \mathbf{J}(n)\mathbf{J}^T(n) \right]^{-1} \nabla E(\mathbf{W}) & \text{when } \lambda \rightarrow 0 \end{cases}, \tag{7.37}$$

which corresponds to gradient descent, with $\eta = 1/\lambda$, when λ is large, and to Newton learning when λ is small.

7.5 Rules of thumb when choosing a learning algorithm

These rules of thumb are based on my own personal experience.

If the problem is small, with few observations and with few weights, then the Levenberg-Marquardt learning is the preferred learning algorithm. This is because it is very fast if the Hessian is small. Furthermore, the MATLAB implementation of Levenberg-Marquardt stores a matrix which is (number of weights) $\times$ (number of observations). Thus, if either of these numbers is big then the machine you are working on may run out of memory and start swapping – and your learning will take "forever".

If the problem is large, i.e. many observations and few/many weights, then RPROP is a fast and reliable algorithm.

If you have a real-time problem, where you need to adapt the network on-line, then you should use on-line backpropagation.

7.6 Interpretation of the MLP

7.6.1 Classification

If the multilayer perceptron will be used for classification then it should have a logistic output (in the two-class case, in multi-class cases we would use a generalization of the logistic function). If we have a single hidden layer MLP, the output is (c.f. equations (7.8) – (7.11))

$$\hat{y}(\mathbf{x}) = \left\{ 1 + \exp\left[v_0 + \sum_j v_j h_j(\mathbf{w}_j, \mathbf{x}) \right] \right\}^{-1} \tag{7.38}$$

which is of the general form

$$\hat{y}(\mathbf{x}) = \frac{1}{1 + \exp\left[f(\mathbf{x}) \right]} \tag{7.39}$$

where $f(\mathbf{x})$ is a nonlinear function of $\mathbf{x}$ (actually, it is a function of projections of $\mathbf{x}$ onto directions $\mathbf{w}_j$). If we compare this to the classical classification methods, we see that we are dealing with a generalization of the logistic regression, a nonlinear logistic regression model. That is, we are modeling the a posteriori probability $p(c|\mathbf{x})$, but using a nonlinear decision boundary.

We know from the logistic regression model that the proper error measure is the negative log-likelihood

$$\begin{aligned} \mathcal{L} &= \prod_n \hat{y}(n)^{y(n)} \left[1 - \hat{y}(n) \right]^{[1-y(n)]} \\ \Rightarrow E = -\ln \mathcal{L} &= -\sum_n \left\{ y(n) \ln\left[\hat{y}(n) \right] + \left[1 - y(n) \right] \ln\left[1 - \hat{y}(n) \right] \right\} \end{aligned} \tag{7.40}$$

with $\hat{y}(n) = \hat{y}[\mathbf{x}(n)] = \hat{p}(c_1|\mathbf{x}(n))$ and the target output is coded as

$$y(n) = \begin{cases} 1 & \text{if } \mathbf{x}(n) \in c_1 \\ 0 & \text{if } \mathbf{x}(n) \in c_2 \end{cases} \tag{7.41}$$

The error measure (7.40) is sometimes referred to as *cross entropy* and is discussed in e.g. (Bishop 1995).

Note: If we have probabilities $y(n) = p(c_1|\mathbf{x}(n))$ as targets, instead of the $\{0, 1\}$ coding, then we would use what is called the *Kullback-Leibler* distance, which leads to an expression very similar to (7.40). However, this is not included in the course.

Unfortunately, however, the error function (7.40) is not available in the MATLAB neural network toolbox NNET (neither in the NETLAB Toolbox from Aston). Only the summed square error (with possible regularization) is implemented. Fortunately, it also works to use the summed square error, although it isn't the perfectly correct error measure. The argument for this goes as follows: We assume that we have many training patterns ($N \to \infty$) so that we can express the mean square error (MSE) as an integral

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}[y(n)-\hat{y}(n)]^2 \to \sum_k \int [y(\mathbf{x})-\hat{y}(\mathbf{x})]^2 p(\mathbf{x}|c_k)p(c_k)d^D x. \quad (7.42)$$

Using (7.41) and expanding the quadratic expression yields

$$\begin{aligned} \text{MSE} &= \int [1-\hat{y}(\mathbf{x})]^2 p(\mathbf{x}|c_1)p(c_1)d^D x + \int \hat{y}(\mathbf{x})^2 p(\mathbf{x}|c_2)p(c_2)d^D x \\ &= \int \hat{y}(\mathbf{x})^2 p(\mathbf{x})d^D x - 2\int \hat{y}(\mathbf{x})p(c_1|\mathbf{x})p(\mathbf{x})d^D x + p(c_1) \end{aligned} \quad (7.43)$$

where we have used the facts that

$$p(\mathbf{x}|c_1)p(c_1) = p(\mathbf{x})p(c_1|\mathbf{x}), \quad (7.44)$$

$$p(\mathbf{x}|c_1)p(c_1) + p(\mathbf{x}|c_2)p(c_2) = p(\mathbf{x}), \quad (7.45)$$

$$\int p(\mathbf{x}|c_1)d^D x = 1. \quad (7.46)$$

Expression (7.43) can be written in the form

$$\text{MSE} = \int [\hat{y}(\mathbf{x}) - p(c_1|\mathbf{x})]^2 p(\mathbf{x})d^D x - \int p^2(c_1|\mathbf{x})p(\mathbf{x})d^D x + p(c_1) \quad (7.47)$$

where the last two terms are independent of the model $\hat{y}(\mathbf{x})$. This shows that $\hat{y}(\mathbf{x}) = p(c_1|\mathbf{x})$ if we minimize the summed square error (or mean square error), provided that we can trust the asymptotic results.

The result is contingent on the following things:

- We have a large training data set ($N \to \infty$) so that overfitting is no problem. (This is the most serious caveat.)
- The model family $\mathcal{F}$ that we use to fit the model is rich enough to contain $p(c_1|\mathbf{x})$, i.e. $p(c_1|\mathbf{x}) \in \mathcal{F}$. (This is not so serious as long as it can be well approximated by a function in $\mathcal{F}$.)

When we have a limited data set (i.e. always) then it is always more correct to use the negative log-likelihood (7.40). It is also quite simple to modify

a code that minimizes the summed square error so that it minimizes (7.40) instead. It is sufficient to replace $e(n) = y(n) - \hat{y}(n)$ with

$$e(n) \to \frac{e(n)}{\hat{y}(n)\left[1 - \hat{y}(n)\right]} \tag{7.48}$$

in all places where $e(n)$ occurs. Of course, this requires that the output is bounded to $]0, 1[$, which is most easily fixed by requiring a logistic output.

7.6.2 Regression

We use a linear output in the regression case. The MLP function, using a single hidden layer, is then (generally speaking)

$$\hat{y}(\mathbf{x}) = v_0 + \sum_{j=1}^{M} v_j h_j(\mathbf{w}_j^T \mathbf{x}) \tag{7.49}$$

where $h_j(\mathbf{w}^T\mathbf{x})$ are nonlinear functions of the projections $\mathbf{w}_j^T\mathbf{x}$. Models of this form are often referred to as *projection pursuit regression* (PPR) models in statistics, since projections $\mathbf{w}_j^T\mathbf{x}$ are used as arguments. (To be exact, PPR refers to a specific method of minimizing the error and choosing projections but there are strong similarities between the MLP and PPR.)

Just as in the case for classification, we can explore which function $\hat{y}$ that minimizes the summed square error. Again, we assume the limit $N \to \infty$

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}[y(n) - \hat{y}(n)]^2 \to \int\int [y(\mathbf{x}) - \hat{y}(\mathbf{x})]^2 p(\mathbf{x})p(\varepsilon)d^D x d\varepsilon. \tag{7.50}$$

We assume the standard process equation

$$y(n) = y[\mathbf{x}(n)] = g[\mathbf{x}(n)] + \varepsilon, \tag{7.51}$$

where the noise process is zero mean and with a fixed standard deviation, i.e.

$$\langle \varepsilon \rangle = \int \varepsilon p(\varepsilon) d\varepsilon = 0, \tag{7.52}$$

$$\langle \varepsilon^2 \rangle - \langle \varepsilon \rangle^2 = \langle \varepsilon^2 \rangle = \int \varepsilon^2 p(\varepsilon) d\varepsilon = \sigma_\varepsilon^2. \tag{7.53}$$

Using (7.51), expanding the quadratic expression, and using the fact that $\int p(\varepsilon)d\varepsilon = 1$ and $\int p(\mathbf{x})d^D x = 1$, yields

$$\text{MSE} = \int g^2(\mathbf{x})p(\mathbf{x})d^D x + \int \varepsilon^2 p(\varepsilon)d\varepsilon + \int \hat{y}^2(\mathbf{x})p(\mathbf{x})d^D x$$

$$
\begin{aligned}
&+ \quad 2\int [g(\mathbf{x}) - \hat{y}(\mathbf{x})]p(\mathbf{x})d^D x \int \varepsilon p(\varepsilon) d\varepsilon \\
&- \quad 2\int g(\mathbf{x})\hat{y}(\mathbf{x})p(\mathbf{x})d^D x. \qquad (7.54)
\end{aligned}
$$

We can simplify this further using (7.52) and (7.53), which results in

$$
\begin{aligned}
\text{MSE} &= \int g^2(\mathbf{x})p(\mathbf{x})d^D x + \sigma_\varepsilon^2 + \int \hat{y}^2(\mathbf{x})p(\mathbf{x})d^D x \\
&- 2\int g(\mathbf{x})\hat{y}(\mathbf{x})p(\mathbf{x})d^D x \\
&= \sigma_\varepsilon^2 + \int [g(\mathbf{x}) - \hat{y}(\mathbf{x})]^2 p(\mathbf{x})d^D x. \qquad (7.55)
\end{aligned}
$$

Thus, if we truly minimize the training error then the output $\hat{y}(\mathbf{x}) = g(\mathbf{x})$.

However, this is contingent upon two things:

- We have a large training data set ($N \to \infty$) so that overfitting is no problem. (This is the most serious caveat.)
- The model family $\mathcal{F}$ that we use to fit the model is rich enough to contain $g(\mathbf{x})$, i.e. $g \in \mathcal{F}$. (This is not so serious as long as g can be well approximated by a function in $\mathcal{F}$.)

The MSE value and the noise variance: We assume that the noise process ε has zero mean when we use the summed square error. If this is true, which we can check by looking at the residuals $e(n)$, then we have the relation

$$
\sigma_\varepsilon^2 = \frac{1}{N-1}\sum_{n=1}^{N} \varepsilon^2(n) \approx \frac{1}{N}\sum_{n=1}^{N} \varepsilon^2(n) = \frac{1}{N}\sum_{n=1}^{N} e^2(n) = \text{MSE}. \qquad (7.56)
$$

The distribution of the true value: From a maximum likelihood perspective, using the summed square error is equivalent to assuming a Gaussian noise distribution. Thus, if the residuals are normally distributed with zero mean (which we want them to be) then we also have an idea about the distribution of the true value y given the prediction $\hat{y}$.

$$
\begin{aligned}
p(e) &= \frac{1}{\text{RMS}\sqrt{2\pi}} \exp\left[\frac{-e^2}{2\text{MSE}}\right] \\
\Rightarrow p(y|\hat{y}) &= \frac{1}{\text{RMS}\sqrt{2\pi}} \exp\left[\frac{-(y-\hat{y})^2}{2\text{MSE}}\right] \qquad (7.57)
\end{aligned}
$$

where RMS $= \sqrt{\text{MSE}}$. That is, if we were able to collect a large quantity of experimental data and compared the prediction to the true value, then we would expect the true value to be distributed normally around the prediction value.

7.7 How to estimate the generalization error

The real goal when modeling is to generalize to new data, not just perform well on the training data set that is presented during training.

In general, the error on the training data will be a biased estimate of the generalization error. To be specific, it will tend to be smaller than the generalization error if we select our model such that it minimizes the training error. We can therefore not use the training error as our selection criteria.

One way to estimate the generalization error is to do cross-validation. This means using a test data set, which is a subset of the available data (typically 25-35%) that is removed before any training is done, and which is not used again until all training is done. The performance on this test data will be an unbiased estimate of the generalization error, provided that the data has not been used in any way during the modeling process. If it has been used, e.g. for model validation when selecting hyperparameter values, then it will be a biased estimate.

If there is lots of data available then it may be sufficient to use one test set for estimating the generalization error. However, if data is scarce then it is necessary to use more data-efficient methods. One such method is the K-fold cross-validation method.

The central idea in K-fold cross-validation is to repeat the cross-validation test K times. That is, divide the available data into K subsets, here denoted by $\mathcal{D}_k$, where each subset contains a sample of the data that reflects the data distribution (i.e. you must make sure that one subset does not contain e.g. only one category in a classification task). The procedure then goes like this:

1. Repeat K times, i.e. until all data subsets have been used for testing once.

    1.1 Set aside one of the subsets, $\mathcal{D}_k$, for testing, and use the remaining data subsets $\mathcal{D}_{i\neq k}$ for training.

1.2 Train your model using the training data.

1.3 Test your model on the data subset $\mathcal{D}_k$. This gives you a test data error $E_{test,k}$.

2. The estimate of the generalization error is the mean of the K individual test errors: $E_{gen.} = \frac{1}{K}\sum_k E_{test,k}$.

One benefit with K-fold cross-validation is that you can estimate an error bar for the generalization error by computing the standard deviation of the $E_{test,k}$ values.

Note: The errors $E_{test,k}$ are often approximately log-normally distributed. At least, $\log E_{test,k}$ tends to be more normally distributed than $E_{test,k}$. It is therefore more appropriate to use the mean of the logs as an estimate for the log generalization error. That is

$$\log E_{gen.} = \frac{1}{K}\sum_{k=1}^{K} \log E_{test,k} \tag{7.58}$$

$$\Delta \log E_{gen.} = 1.96\sqrt{\frac{1}{K-1}\sum_{k=1}^{K}[\log E_{test,k} - \log E_{gen.}]^2} \tag{7.59}$$

where the lower row is a 95% confidence band for the log generalization error.

An error bar from cross-validation includes the model variation due to both different training sets and different initial conditions.

References

Bishop, C. M. (1995), *Neural Networks for Pattern Recognition*, Oxford University Press, Oxford.

Haykin, S. (1999), *Neural Networks – A Comprehensive Foundation*, second edn, Prentice Hall Inc., Upper Saddle River, New Jersey.

Riedmiller, M. & Braun, H. (1993), A direct adaptive method for faster backpropagation learning: The RPROP algorithm, *in* H. Ruspini, ed., 'Proc. of the IEEE Intl. Conference on Neural Networks', San Fransisco, California, pp. 586–591.