Chapter 4
Deep Neural Networks (DNN)

Neural networks are a family of machine learning models that consist of connected function units called neurons. They are built as powerful function approximators that accurately map input data $\boldsymbol{x}$ to output y (i.e., to learn a function $f(\boldsymbol{x}) \approx y$) through multiple layers of nonlinear transformations. Such a design enables neural networks to perform tasks like classification and regression.

The neurons in a neural network are organized into three interconnected layers: input, hidden, and output layers. When a neural network has one or more layers of hidden units, it can be regarded as a deep neural network (DNN). Based on different types of input data, multiple variants of DNN are invented. For example, the convolutional neural networks (CNN or ConvNet) are suitable for modeling grid data such as images or continuous time series. The recurrent neural networks (RNN) are appropriate for modeling sequential data such as text and clinical event sequences. Moreover, these variants have demonstrated great performance in solving real-world healthcare problems, for instance, CNN for automatic classification of skin lesions from image data [45], and RNN for clinical event prediction from patients longitudinal electronic health data [25, 30]. We will describe different DNN variants in later chapters. In this chapter, our focus is the basic deep neural networks (DNN). We summarize notations used throughout this chapter in Table 4.1. We first describe the structure and the training method of a single neuron and then move on to a general DNN, where neurons are organized in multiple interconnected layers.

4.1 A Single Neuron

To describe a deep neural network, we start with a simplest one that comprises of a single neuron. The neuron is a computational unit that takes n input values $\boldsymbol{x} = [x_1, \cdots, x_n]$ and their associated weights $\boldsymbol{w} = [w_1, \cdots, w_n]$. Then, a neuron takes

C. Xiao, J. Sun, *Introduction to Deep Learning for Healthcare*,
https://doi.org/10.1007/978-3-030-82184-5_4

Table 4.1 Notations for deep neural networks

Notation	Definition
$\boldsymbol{x}$	Input feature vector
x_i	ith input feature
$\boldsymbol{y}$	Output layer
y_i	ith output feature
L	Number of layers
l	Index of hidden layers
$\boldsymbol{W}^{(l)}$	Weight matrix of the lth layer
$w_{ji}^{(l)}$	Weight associated with unit i of the lth layer and unit j of the $(l+1)$th layer
$\boldsymbol{b}^{(l)}$	Bias vector for the lth layer
$b_j^{(l)}$	Weight of lth layer bias associated with jth unit of the $(l+1)$th layer
$g^{(l)}(\cdot)$	Activation function for the lth layer
$\boldsymbol{z}^{(l)}$	Output vectors of the lth layer
$z_j^{(l)}$	Output value of the jth unit of the lth layer
$\boldsymbol{a}^{(l)}$	Output vectors of the activation of the lth layer
$a_j^{(l)}$	Activated output value of the jth unit of the lth layer

a linear transformation of the inputs as $z = \boldsymbol{w}^T\boldsymbol{x} = \sum_{i=1}^{n} w_i x_i$, and follows with a nonlinear activation function $g(z)$. The output $g(z)$ can be used in machine learning tasks such as classification, with additional evaluation using loss functions. Thus two key components in this computation are activation functions and loss functions.

4.1.1 Activation Function

The activation functions, denoted as $g(\cdot)$, are the main components that perform a nonlinear transformation. Although we can use simple linear activation functions such as $g(x) = x$, the neural network's power often comes from the nonlinear activation functions such as Sigmoid, Tanh, and Rectified Linear Unit (ReLU). They are plotted on Fig. 4.1 for comparison.

Sigmoid as given by Eq. (4.1) outputs real values bounded in the range of (0, 1), making it a popular choice for binary classification due to the results can naturally be interpreted as the probability of an event, for example, the probability of having heart diseases.

$$g(x) = \frac{1}{1+e^{-x}} \tag{4.1}$$

However, since the gradient around either end of the function $g(x)$ (i.e., when x is very large or very small) is very close to zero, the output values toward either end

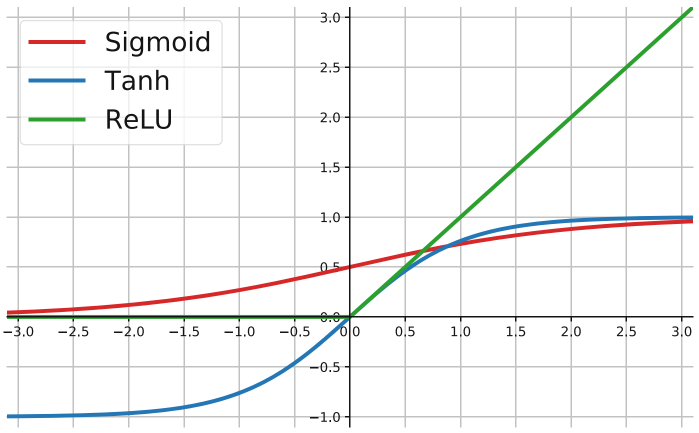


Fig. 4.1 The activation functions: Sigmoid, Tanh, and ReLU

tend to have less response to changes in input x, which is referred to as the *vanishing gradient* problem. The vanishing gradient causes gradient descent optimization to slow down dramatically, which is one of the most challenging problems in deep learning. In particular, the vanishing gradient becomes a major concern when using sigmoid activation. The gradient for the logistic activation function can be derived as below.

$$\begin{aligned}\frac{\partial}{\partial x}\left(\frac{1}{1+e^{-x}}\right) &= \frac{e^{-x}}{\left(1+e^{-x}\right)^2}\\ &= \frac{1+e^{-x}-1}{\left(1+e^{-x}\right)^2}\\ &= \frac{1+e^{-x}}{\left(1+e^{-x}\right)^2}-\left(\frac{1}{1+e^{-x}}\right)^2\\ &= g(x)(1-g(x))\end{aligned}$$

Tanh as given by Eq. (4.2) is an alternative to the Sigmoid function.

$$g(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}}=\frac{2}{1+e^{(-2x)}}-1=2\text{sigmoid}(2x)-1. \tag{4.2}$$

It is easy to see Tanh is a scaled and shifted Sigmoid function: with the scaling, the derivatives are much steeper, while the shift makes Tanh center around zero and has a range $(-1, 1)$. Due to scaling, the gradient of Tanh is stronger and has a wider range than that of the sigmoid (see Fig. 4.2), thus alleviate the vanishing gradient issue that occurs to a single unit. Nevertheless, Tanh still has the vanishing gradient

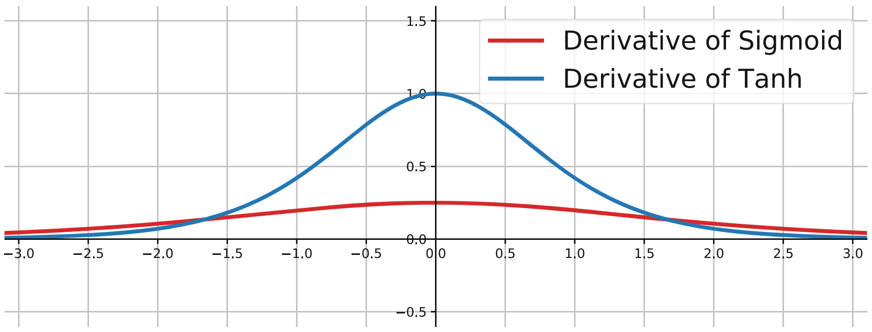


Fig. 4.2 The derivative of Sigmoid (red curve) and the derivative of Tanh (in blue curve) functions

problem at both ends. The gradient of Tanh is given by

$$\begin{aligned}\frac{\partial}{\partial x}\left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right) &= \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2} \\ &= 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\ &= 1 - g^2(x)\end{aligned}$$

using the quotient rule of derivative and knowing $\frac{\partial}{\partial x}e^x = e^x$ and $\frac{\partial}{\partial x}e^{-x} = -e^{-x}$

Softmax extends Sigmoid to handle multi-class classification. Assuming we have K classes, the goal is to estimate the probability of the class label taking on each of the K possible categories, $P(y = k)$ for $k = 1 \cdots, K$. Thus, we can normalize any K-dimensional vector $\boldsymbol{x}$ to a K-dimensional probability vector $\sigma(\boldsymbol{x})$:

$$\sigma(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{k=1}^K e^{x_k}}$$

Note that $\sigma(\boldsymbol{x})_i$ indicates the i-th element of K-dimensional probability vector.

The Gradient of the Softmax Function is derived as follows. For the $i = j$ case, we have

$$\begin{aligned}\frac{\partial(\sigma(\boldsymbol{x})_i)}{\partial x_j} &= \frac{\partial}{\partial x_j}\left(\frac{e^{x_i}}{\sum_k e^{x_k}}\right) \\ &= \frac{e^{x_i}(\sum_k e^{x_k}) - e^{x_j}e^{x_i}}{(\sum_k e^{x_k})^2}\end{aligned}$$

$$= \frac{e^{x_i}}{\sum_k e^{x_k}} \frac{\sum_k e^{x_k} - e^{x_j}}{\sum_k e^{x_k}}$$
$$= \sigma(\boldsymbol{x})_i(1 - \sigma(\boldsymbol{x})_j)$$

For the $i \neq j$ case, we have

$$\frac{\partial(\sigma(\boldsymbol{x})_i)}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{e^{x_i}}{\sum_k e^{x_k}}\right)$$
$$= \frac{-e^{x_j} e^{x_i}}{(\sum_k e^{x_k})^2}$$
$$= -\sigma(\boldsymbol{x})_j \sigma(\boldsymbol{x})_i$$

Note that the quotient rule is applied in the above derivation, namely given $f(x) = \frac{g(x)}{h(x)}$, $f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$.

In summary, the gradient of the softmax function is

$$\frac{\partial(\sigma(\boldsymbol{x})_i)}{\partial x_j} = \begin{cases} \sigma(\boldsymbol{x})_i(1 - \sigma(\boldsymbol{x})_j), & \text{if } i = j \\ -\sigma(\boldsymbol{x})_j \sigma(\boldsymbol{x})_i, & \text{if } i \neq j \end{cases} \tag{4.3}$$

ReLU The Rectified Linear Unit (ReLU) defined as $g(x) = \max(0, x)$ is half-rectified in the sense that its activation is thresholded at 0. When the input is smaller than 0, the output is always 0. Otherwise, ReLU is the identity function. The major advantages of ReLU are computational efficiency and less prone to vanishing gradient. The gradient is either 1 or 0. It might be surprising why ReLU can lead to a powerful function as it seems to simply zero out negative values. However, despite its simplicity, ReLU is still a nonlinear function which is essential for neural network learning. When multiple ReLU functions stack up, they can produce a complex nonlinear function.

The regular ReLU function may be stuck at zero when x are negative, limiting the learning capacity of the neuron. To solve this issue, the **leaky ReLU** was proposed to introduce a small slope for the negative part instead of strict zero:

$$f(x) = \left\{\begin{matrix} ax & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{matrix}\right\}$$

where a is a small constant, e.g., $a = 0.01$.

4.1.2 Loss Function

In contrast to activation functions that perform the transformation of the input data at each unit, a loss function encodes the optimization objective of a neural network and quantifies that gap ("loss") between the predicted output of the neural network as $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{W})$ and the ground truth $\boldsymbol{y}$. The choice of loss function depends on data types and tasks (e.g., regression or classification). Different loss functions will yield different loss value for the same prediction and will have a considerable effect on the resulting neural networks. Below are several commonly used loss functions.

Regression For regression tasks, one can use L2 loss such as the mean squared error (MSE) as given by Eq. (3.10). The MSE computes the square of the difference between the actual value and predicted value. Another choice for regression tasks is to use the L1 loss, such as the least absolute deviations as given by

$$L = \frac{1}{N}\sum_{i=1}^{N} |f(\boldsymbol{x}_i) - y_i| \tag{4.4}$$

Compared to L2 loss, the L1 loss is more robust in that it is more resistant to outliers in the data. However, L1 loss is harder to optimize than L2 loss due to the non-smoothness of its gradient.

Binary Classification The negative log-likelihood is a common loss function for binary classification as given by Eq. (4.5).

$$L = -\sum_{i=1}^{N} y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i), \tag{4.5}$$

where y_i is ground truth binary label for observation i and $\hat{y}_i = P(y_i|\boldsymbol{x}_i)$ is the probability estimate of y_i based on data $\boldsymbol{x}_i$.

Multi-Class Classification The **cross-entropy loss** is a common loss function for multi-class classification. Then we can use cross-entropy loss as given by Eq. (4.6) to calculate average performance over all N training data points over all K classes.

$$L = -\sum_{i=1}^{N}\sum_{k=1}^{K} I(y_i = k)\log(P(y_i = k)) \tag{4.6}$$

where $I(y_i = k)$ is binary indicator (0 or 1) if class label k is the correct classification for observation i. And $P(y_i = k)$ is the predicted probability that observation i is of class k. Note that cross-entropy loss and negative log-likelihood loss are essentially equivalent in the binary classification setting.

The Cross-Entropy Loss with Softmax is one of the most common configurations of the output layer of neural networks. Its gradient is fairly simple, but the derivation is a bit involved.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial o_i} &= \frac{\partial \mathcal{L}}{\partial L_i}\frac{\partial \mathcal{L}_i}{\partial o_i}\\ &= 1 \cdot \frac{\partial \mathcal{L}_i}{\partial o_i}\end{aligned}$$

where o_i is the softmax output and $L_i = -\sum_{k=1}^{K} I(y_i = k)\log(P(y_i = k))$. Then

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial o_i} &= -\sum_k I(y_i = k)\frac{\partial \log(P(y_i = k))}{\partial P(y_i = k)}\frac{\partial P(y_i = k)}{\partial o_i}\\ &= -\sum_k I(y_i = k)\frac{1}{P(y_i = k)}\frac{\partial P(y_i = k)}{\partial o_i}\\ &= -\frac{I(y_i = i)}{P(y_i = i)}\frac{\partial P(y_i = i)}{\partial o_i} - \sum_{k \neq i}\frac{I(y_i = k)}{P(y_i = k)}\frac{\partial P(y_i = k)}{\partial o_i}\end{aligned}$$

Now we can plug in the gradient of softmax function from Eq. (4.3) to derive the final gradient:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial o_i} &= -I(y_i = i)(1 - P(y_i = i)) + \sum_{k \neq i} I(y_i = k)P(y_i = i)\\ &= -I(y_i = i) + \sum_k I(y_i = k)P(y_i = i)\\ &= P(y_i = i) - I(y_i = i)\\ &= \hat{y}_i - y_i \qquad (4.7)\end{aligned}$$

where $\hat{y}_i$ is the probability estimate of class i from the model and y_i is the ground truth binary indicator for class i.

4.1.3 Train a Single Neuron

To train a neural network, we use gradient descent to adjust the weights of the network. Figure 4.3 illustrates the setup of training a single neuron with the sigmoid activation and squared loss function. In particular, we derive the role of each weight

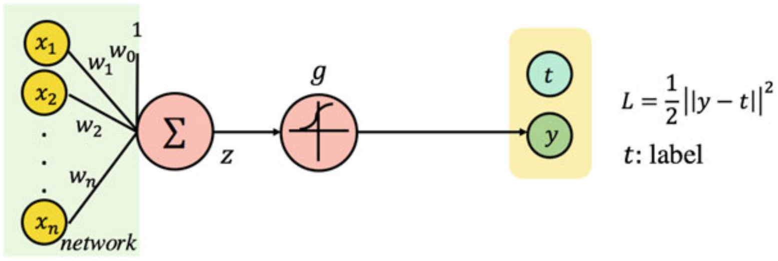


Fig. 4.3 Train a simplest neural network with one neuron. In this case we have sigmoid activation function and squared loss

in the network's overall loss as error gradient $\frac{\partial \mathcal{L}}{\partial w_i}$. Given a data point $(\boldsymbol{x}, t)$ where t is the target label and neural network output $y = f(\boldsymbol{x})$, the error gradient is the following:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_i} &= \frac{1}{2}\frac{\partial}{\partial w_i}||y - t||^2 \\ &= (y - t)\,\frac{\partial y}{\partial w_i} \\ &= (y - t)\,\frac{\partial y}{\partial \sum_i w_i x_i}\frac{\partial \sum_i w_i x_i}{\partial w_i} \\ &= (y - t)\,y(1 - y)x_i. \end{aligned} \tag{4.8}$$

Then we compute $\nabla \mathcal{L} = [\frac{\partial \mathcal{L}}{\partial w_0}, \cdots, \frac{\partial \mathcal{L}}{\partial w_n}]$. Finally, we can update the weight $\boldsymbol{w}$ in the opposite direction of $\nabla \mathcal{L}$ as shown in Algorithm 2.

Algorithm 2 Stochastic gradient descent

```
Input: training data $(x, t)$, learning rate $\eta$
Initialize each $w_i$ to some small random value.
Until convergence
DO
    Initialize each $\nabla w_i$ to 0.
    For each $(x, t)$ in training data, DO
        Pass $x$ through the neuron to compute output $y$;
        Compute $\mathcal{L}$ for this observation $(x, t)$;
        Compute $\nabla \mathcal{L}$ using Eq. (4.8);
        Update weight $w \leftarrow w - \eta \nabla \mathcal{L}$.
```

4.2 Multilayer Neural Network

After understanding how to learn a single neuron, next we present the algorithm for learning a general neural network.

4.2.1 Network Representation

Deep neural networks, also called multilayer perceptrons (MLP), feed-forward networks, or fully connected networks, are the most standard neural networks model. The networks contain many neurons organized by multiple layers, and the neurons between consecutive layers are connected.

- The feedforward neural networks define a mapping function $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{W})$ where $\boldsymbol{x}$ are the input feature vectors and $\boldsymbol{W}$ are the parameters of the neural network.
- The parameter set $\boldsymbol{W} = \{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)} | 1 \leq l \leq L\}$ where $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ are the weight matrix and bias vector of the l-th layer, respectively.
- The goal is to determine the values of $\boldsymbol{W}$ such that the neural network can predict correct value $\boldsymbol{y}$ based on input $\boldsymbol{x}$.
- Computation flows from the input vector $\boldsymbol{x}$ to the output vector $\boldsymbol{y}$ via hidden and output layers $\boldsymbol{h}^{(l)} | 1 \leq l \leq L$. The hidden layers are called "hidden" because their values are not observed in the training data.
- For a L-layer neural network, the L is the total number of hidden layers plus the output layer (as a convention, the input layer is not counted). In particular, $\boldsymbol{h}^{(L)}$ is the output layer also denoted as $\boldsymbol{y}$.

The deep neural networks can be leveraged in many predictive tasks such as disease risk prediction using the electronic health records (EHR) data. A common approach is to convert raw EHR data into feature vectors, consisting of various medical codes (e.g., diagnosis and procedure codes). For example, we can transform the raw EHR vectors into binary vectors using multi-hot encoding methods, as illustrated in Fig. 4.4. A **one-hot encoding** vector contains only one "1" at the dimension corresponding to the medical code and "0" otherwise. A **multi-hot encoding** vector has the same number of dimensions as one-hot encoding but can have multiple "1"s, e.g., the "1"s represent multiple medical codes in a single clinical visit. We denote these input vectors as $\boldsymbol{x}$.

For example, a feedforward neural network model can map $\boldsymbol{x}$ to a specific disease prediction such as heart failure onset. Note that the last hidden layer ($(L-1)$-th layer) produces higher-level patient representation from raw EHR data. Some previous studies utilize such learned representation in finding disease subtypes [4, 15, 177].

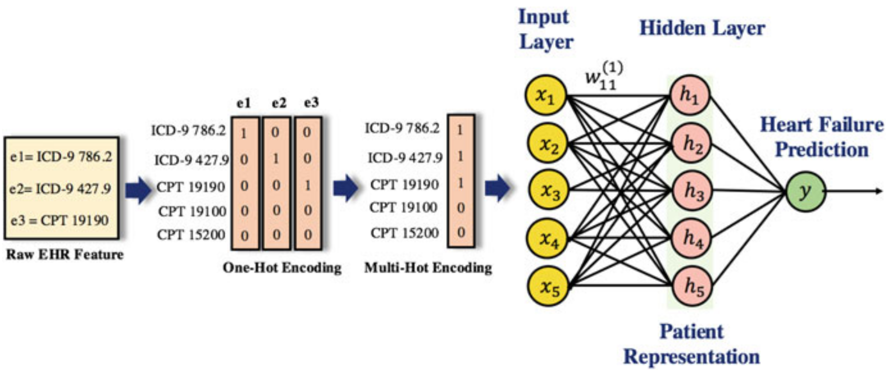


Fig. 4.4 We transform the raw EHR vectors that include multiple medical codes into binary vectors where a dimension with "1" indicates the presence of particular medical codes and "0"s mean the absence of the medical codes. The encoded features are then passed into a Deep Neural Network (DNN) for heart failure prediction

4.2.2 Train a Multilayer Neural Network

We will next walk through the example given by Fig. 4.5 to describe how to train a multilayer neural network. Similar to training a single neuron, we still use gradient descent to adjust the weights $w_{ji}^{(l)}$ and $b_j^{(l)}$ of the network to move its output in the opposite direction of the gradient. The only difference is the expression of the updating rule. The new rule is given by Eq. (4.10).

$$w_{ji}^{(l)} = w_{ji}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} \tag{4.9}$$

$$b_j^{(l)} = b_j^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b_j^{(l)}} \tag{4.10}$$

In this example, we still use sigmoid activations on training data (x, t) to minimize squared error.

Forward Computation

As the name suggests, forward computation is to perform the layer-by-layer computation in a forward manner against an input data point. For better illustration of this process we further define the following variables. We denote $z_j^{(l)}$ as the pre-activation value at the jth node of the lth layer, $g^{(l)}$ as the activation function of the lth layer, and $a_j^{(l)}$ as the output value of the jth node of the lth layer. The $a_j^{(l)}$ is the

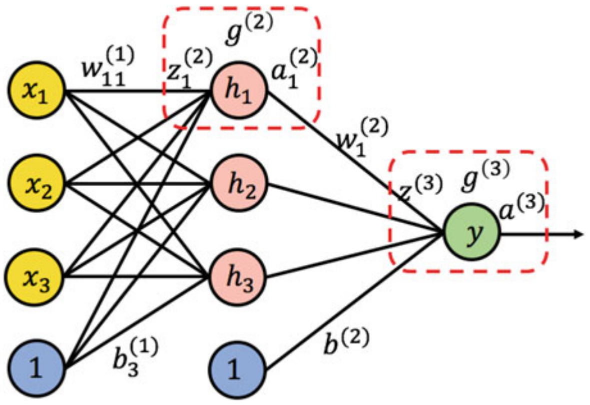


Fig. 4.5 Train a multilayer perceptron. In this case we have sigmoid activation function and squared loss

activation of $z_j^{(l)}$ via $g^{(l)}$. As the first step of the forward computation, each unit in the hidden layer performs a linear combination of the inputs and then transform the linear combination using an activation function:

$$\begin{aligned}
\overbrace{z_1^{(2)}}^{pre\text{-}activation} &= \textstyle\sum_i w_{1i}^{(1)} x_i + b_1^{(1)}, & \overbrace{a_1^{(2)}}^{output} &= \overbrace{g^{(2)}(z_1^{(2)})}^{activation} \\
z_2^{(2)} &= \textstyle\sum_i w_{2i}^{(1)} x_i + b_2^{(1)}, & a_2^{(2)} &= g^{(2)}(z_2^{(2)}) \\
z_3^{(2)} &= \textstyle\sum_i w_{3i}^{(1)} x_i + b_3^{(1)}, & a_3^{(2)} &= g^{(2)}(z_3^{(2)})
\end{aligned}$$

Similarly each unit in the output layer will take these activation $a_j^{(2)}$ as inputs and perform pre-activation linear combination and activation, respectively:

$$z^{(3)} = \sum_j w_j^{(2)} a_j^{(2)} + b^{(2)}, \quad a^{(3)} = g^{(3)}(z^{(3)}).$$

Note that these computations can be represented via compact vector notations:

$$\begin{aligned}
\boldsymbol{z}^{(2)} &= \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \\
\boldsymbol{a}^{(2)} &= g^{(2)}(\boldsymbol{z}^{(2)}) \\
\boldsymbol{z}^{(3)} &= \boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)} + \boldsymbol{b}^{(2)} \\
a^{(3)} &= g^{(3)}(\boldsymbol{z}^{(3)})
\end{aligned}$$

For a more common case, the forward computation from the lth layer to the $(l+1)$th layer can be expressed as:

$$\boldsymbol{z}^{(l+1)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)}$$

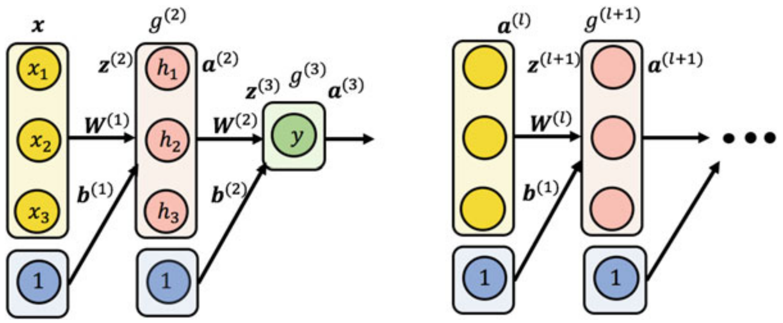


Fig. 4.6 Vector forms of forward computation

$$\boldsymbol{a}^{(l+1)} = g^{(l+1)}(\boldsymbol{z}^{(l+1)}).$$

The vector forms of forward computation for a deep neural network are illustrated in Fig. 4.6.

Backward Propagation

The backpropagation is designed to efficiently compute the partial derivatives $\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}}$ and $\frac{\partial \mathcal{L}}{\partial b_j^{(l)}}$ that are needed in the gradient descent optimization procedure. Those derivatives can be computed for neural networks through a backward traversal over the network from the output layer to the input layer. The procedure of backpropagation is described as follows: Given a training data point (x, t), we will first run a forward computation to compute all the activations throughout the network to produce the output value. For example, in Fig. 4.7 the output value is y or $a^{(3)}$. The network output will be used as input $z^{(4)}$ in the computation of loss against target t. Next, we perform the backward error propagation as follows. For each node j in the lth layer, we compute an error term $\delta_j^{(l)}$ that measures how much that node was responsible for the error in the output. The backpropagation update follows the reverse order of the forward computation: $a^{(3)}$,$\boldsymbol{z}^{(3)}$, $\boldsymbol{a}^{(2)}$, and $\boldsymbol{z}^{(2)}$.

First we derive the desired partial derivatives of the weights $\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}}$ based on the chain rule.

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial w_{ji}^{(l)}}$$

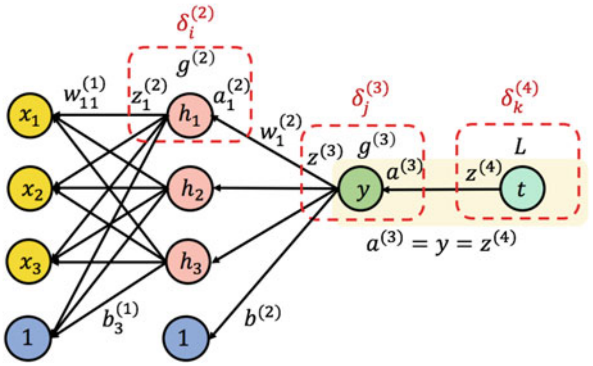


Fig. 4.7 Backpropagation on neural networks

$$= \delta_j^{(l+1)} \frac{\partial(\sum_{i'} w_{ji'}^{(l)} a_{i'}^{(l)} + b_{i'}^{(l)})}{\partial w_{ji}^{(l)}}$$

$$= \delta_j^{(l+1)} a_i^{(l)}$$

where $\delta_j^{(l+1)} = \frac{\partial \mathcal{L}}{\partial z_j^{(l+1)}}$ defines the error component that measures how much node j of the $l+1$th layer was responsible for the overall error of the output, and $a_i^{(l)}$ is the i-th input from the (l)-th layer. Note that here for $\partial(\sum_{i'} w_{ji'}^{(l)} a_{i'}^{(l)} + b_{i'}^{(l)})$ in the numerator we temporally use i' as the general index, to differentiate it from specific i of $\partial w_{ji}^{(l)}$ in the denominator. There is only one i' that matches i in $\partial w_{ji}^{(l)}$, and other items in the summation will be treated as constant with 0 derivative.

Similarly, we have the partial derivative of the bias weights $\frac{\partial \mathcal{L}}{\partial b_j^{(l)}}$:

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial b_j^{(l)}}$$

$$= \delta_j^{(l+1)} \frac{\partial(\sum_{i'} w_{ji'}^{(l)} a_{i'}^{(l)} + b_{i'}^{(l)})}{\partial b_j^{(l)}}$$

$$= \delta_j^{(l+1)}$$

where again $\delta_j^{(l+1)}$ is the error component that measures how much nodes j of the $l+1$th layer was responsible for the overall error of the output.

To compute the partial derivatives, we will need all $\delta_j^{(l)}$. They can be computed from the last layer to the first layer. To start the backpropagation, we first initialize

$\delta^{(4)}$ for the loss computation, which is given by:

$$\delta^{(4)} = \frac{\partial \mathcal{L}}{\partial z_k^{(4)}} = \frac{\partial \frac{1}{2}||y - t||^2}{\partial y} = -(t - y). \tag{4.11}$$

Then we proceed recursively to use $\delta_j^{(l+1)}$ to compute $\delta_j^{(l)}$. In the following, we use the index j to denote both the general index or used as the index for the layers between layers indexed by i and k. The $\delta_j^{(3)}$ is given by:

$$\begin{aligned}
\delta_j^{(3)} &= \frac{\partial \mathcal{L}}{\partial z_j^{(3)}} \\
&= \frac{\partial \mathcal{L}}{\partial z_k^{(4)}} \frac{\partial z_k^{(4)}}{\partial z_j^{(3)}} \\
&= \delta^{(4)} \frac{\partial g^{(3)}(z_j^{(3)})}{\partial z_j^{(3)}} \\
&= \delta^{(4)} (g^{(3)})'(z_j^{(3)}) \\
&= -(t - y)(g^{(3)})'(z^{(3)})
\end{aligned}$$

Next we have $\delta_i^{(2)}$ given by Eq. (4.12).

$$\begin{aligned}
\delta_i^{(2)} &= \frac{\partial \mathcal{L}}{\partial z_i^{(2)}} \\
&= \frac{\partial \mathcal{L}}{\partial z_j^{(3)}} \frac{\partial z_j^{(3)}}{\partial z_i^{(2)}} \\
&= \delta_j^{(3)} \frac{\partial \sum_{i=1}^{d^{(2)}=3} w_{ji}^{(2)} a_i^{(2)}}{\partial z_i^{(2)}} \\
&= \delta_j^{(3)} \frac{\partial \sum_{i=1}^{d^{(2)}=3} w_i^{(2)} a_i^{(2)}}{\partial a_i^{(2)}} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \\
&= \delta_j^{(3)} w_i^{(2)} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \\
&= \delta_j^{(3)} w_i^{(2)} (g^{(2)})'(z_i^{(2)})
\end{aligned}$$

Note that $(g^{(3)})'(\cdot)$ and $(g^{(2)})'(\cdot)$ are the gradients of the activation function $g^{(3)}(\cdot)$ and $g^{(2)}(\cdot)$, respectively. In this specific example, all the $g^{(l)}(\cdot)$ are sigmoid. Thus in computation we need to substitute $g^{(l)}(\cdot)$ and $(g^{(l)}(\cdot))'$ with sigmoid and its derivative. Also in general, we have the following relation between $\delta_j^{(l)}$ and $\delta_k^{(l+1)}$

$$\delta_j^{(l)} = \left[\sum_{k=1}^{d^{(l+1)}} w_{kj}^{(l)} \delta_k^{(l+1)} \right] (g^{(l)})'(z_j^{(l)}). \tag{4.12}$$

Here we summarize the backpropagation algorithm for efficient gradient computation.

Algorithm 3 Backpropagation algorithm

```
The forward pass
Starting with the input $\boldsymbol{x}$, go forward to output layer, compute and store the variables $\boldsymbol{z}^{(2)}, \boldsymbol{a}^{(2)}, \boldsymbol{z}^{(3)}, \boldsymbol{a}^{(3)}, \cdots, \boldsymbol{z}^{(L)}, \boldsymbol{a}^{(L)}, \boldsymbol{z}^{(L+1)}$.

The backward pass
Initialize $\delta^{(L+1)} = -(t-y) = -(t - z^{(L+1)})$ and compute the derivatives at the output layer as $\frac{\partial L}{\partial w_k^{(L+1)}} = -(t-y)a_j^{(L)}$.
DO
    Compute $\delta_j^{(l)} = \left[\sum_{k=1}^{d^{(l+1)}} w_{kj}^{(l)} \delta_k^{(l+1)}\right] (g^{(l)})'(z_j^{(l)})$
    compute the derivatives at layer $l$ as $\frac{\partial L}{\partial w_{ji}^{(l)}} = a_i^{(l)} \delta_j^{(l+1)}$ and $\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l+1)}$
```

4.2.3 *Parameters and Hyper-Parameters*

There are several variables in the model for deep neural networks that need to be learned from the data. These variables are called parameters such as the weight matrix $\boldsymbol{W}$ of the neural network. Learning the model parameters is done through a process known as model training. In other words, by training a neural network model with some existing data, we can fit the model parameters. Besides parameters, other parameters cannot be directly learned from the regular training process. These parameters, known as hyperparameters, express higher-level properties of the model and are usually fixed before the actual training process begins. For deep neural networks, hyperparameters include the number of nodes and the number of hidden layers.

To choose hyper-parameters, we can use a randomized search or Bayesian optimization. For randomized search, we can use knowledge of the problem to pre-identify a range of hyperparameters. We randomly select hyperparameters from this region and repeat this process until we find parameters that work well. In [6], the authors showed a randomized search performed better than other methods in their case. For Bayesian optimization, we can use existing experiment information to decide how to adjust the hyper-parameters for the next experiment. An example of this approach can be found in [137].

4.3 Case Study: Readmission Prediction from EHR Data with DNN

Problem Hospital readmissions are a huge problem that affects hospital quality and patient outcomes. A study has shown that 17.6% of hospital admissions resulted in readmissions within 30 days of discharge, with 76% of those are avoidable [158]. Medicare, the national insurance program in the US, has introduced payment penalties for various hospital readmissions within 30 days of discharge. These readmissions used to account for \$15 billion in Medicare spending. This kind of penalty has impacted over 2000 hospitals, with \$280 million in penalties. To avoid such expensive readmission, the analytic problem is to identify the high-risk individuals who will likely have readmissions before discharge. Therefore, more hospital resources can be used to avoid expensive readmissions such as more proactive follow-up after discharge.

Data The study used the New Zealand National Minimum Dataset, obtained from the New Zealand Ministry of Health. It consists of nearly 3.3 million hospital admissions in the New Zealand hospital system between 2006 and 2012. They have background information on the patient's race, sex, age, and length of stay for each visit. Additionally, they also know the type of facility (public or private), and whether the patient was a transfer. They also have a single Diagnosis Related Group (DRG) code for each visit, selected from a set of 815 unique DRGs that break down admissions into broader diagnoses classes than the particular ICD codes.

Method DNN models are trained to predict readmissions on five patient cohorts for five different phenotypes, respectively [49]. They are pneumonia (PN), chronic obstructive pulmonary disease (COPD), heart failure (HF), acute myocardial infarction (AMI), and total hip arthroplasty or total knee arthroplasty ((THA/TKA). The problem was formalized as five different readmission prediction problems, each is a binary classification task. The features are also multi-hot vectors of 12,045 diagnosis codes (1 indicating the diagnosis and 0 absence). Each data point corresponds to one admission. There are 3,295,775 admissions. The label is 1 if the patient is readmitted within 30 days and 0 otherwise. The entire $3{,}295{,}775 \times 12{,}045$ binary feature matrix is partitioned into 5 matrices, one for each phenotype. The architecture of DNN has an input layer with a binary feature vector corresponding to the diagnosis codes, three hidden layers, and one output, as shown in Fig. 4.8. As a baseline, penalized logistic regression (PLR) is used which a logistic regression with the elastic net regularization (i.e., regularized with $\alpha\|\beta\|_1 + (1-\alpha)\|\beta\|_2$ where α is a hyperparameter between 0 and 1).

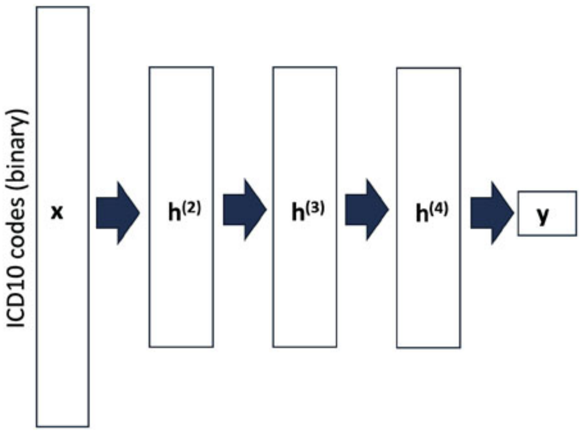
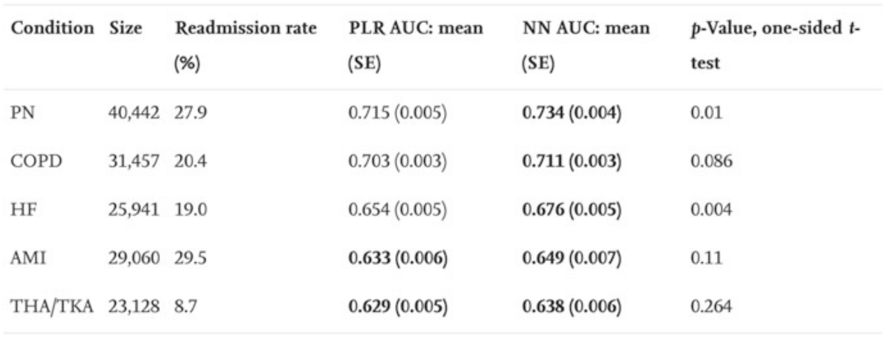


Fig. 4.8 DNN architecture for readmission prediction



Condition	Size	Readmission rate (%)	PLR AUC: mean (SE)	NN AUC: mean (SE)	*p*-Value, one-sided *t*-test
PN	40,442	27.9	0.715 (0.005)	**0.734 (0.004)**	0.01
COPD	31,457	20.4	0.703 (0.003)	**0.711 (0.003)**	0.086
HF	25,941	19.0	0.654 (0.005)	**0.676 (0.005)**	0.004
AMI	29,060	29.5	**0.633 (0.006)**	**0.649 (0.007)**	0.11
THA/TKA	23,128	8.7	**0.629 (0.005)**	**0.638 (0.006)**	0.264

Fig. 4.9 Results from 10-fold cross validation. A penalized logistic regression (PLR) is compared with a deep neural network (NN). *p*-Values for a one-sided t-test showed that DNN has significantly higher AUC than PLR

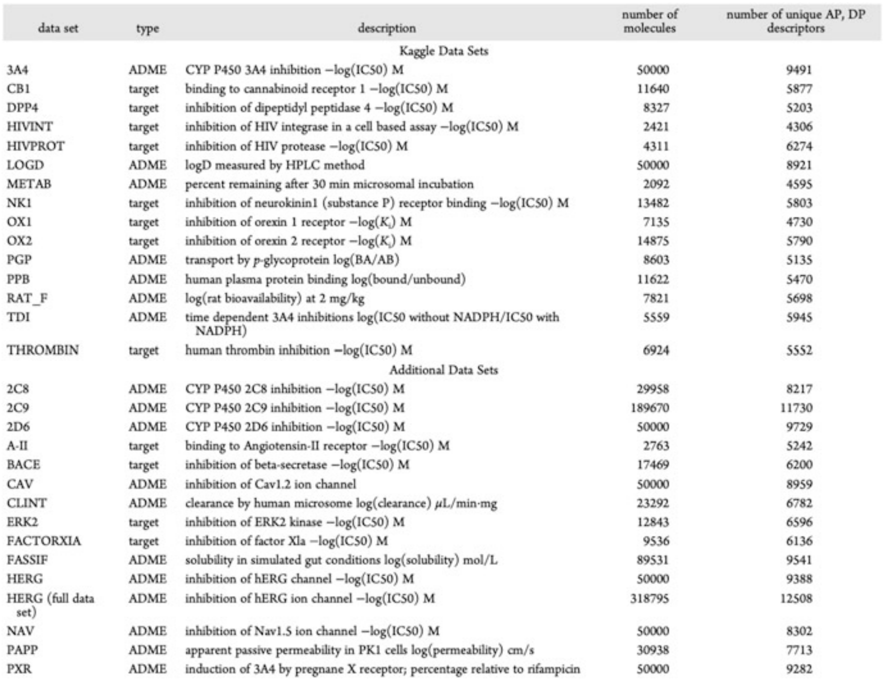
Results Figure 4.9 shows the resulting AUCs per group for both methods, along with the sample sizes and readmission rates. The deep neural networks consistently had better AUC (3 out of 5 times they were significantly better). But DNN models have a large number of parameters, requiring significantly more computational time to train, which has its limitation.

4.4 Case Study: DNN for Drug Property Prediction

Problem The quantitative structure-activity relationships (QSAR) is a classical problem in the pharmaceutical industry for predicting various activities of a chemical compound. Such predictions help prioritize the experiments during the drug discovery process and reduce the experimental work that needs to be done. However, the QSAR methods are particularly computationally expensive or require the adjustment of many sensitive parameters to achieve good prediction for an individual QSAR data set. In [105], the authors systematically evaluate the performance of DNN models for predicting quantitative structure-activity relationships.

Method This paper compares the performance of a DNN model with ReLU layer with a random forest (RF) classifier. First, the authors want to find out how well DNNs can perform relative to RF. Therefore, over 50 DNNs were trained using different parameter settings. These parameter settings were arbitrarily selected, but they attempted to cover a sufficient range of values of each adjustable parameter. The specific parameter choices can be found in the paper [105]. Then the authors tried the unsupervised pretraining to initialize the DNN parameters. For each data set, the authors trained five different DNN configurations with and without pretraining.

Data The data used in this study was from a 2012 Kaggle competition hosted by Merck. These are in-house Merck data sets, including on-target and ADME (absorption, distribution, metabolism, and excretion) activities. The competition aims to examine how well the state of the art machine learning methods can perform QSAR tasks. In this paper, 15 datasets of various sizes (2000–50,000 molecules), as shown in Fig. 4.10.



data set	type	description	number of molecules	number of unique AP, DP descriptors
		Kaggle Data Sets		
3A4	ADME	CYP P450 3A4 inhibition −log(IC50) M	50000	9491
CB1	target	binding to cannabinoid receptor 1 −log(IC50) M	11640	5877
DPP4	target	inhibition of dipeptidyl peptidase 4 −log(IC50) M	8327	5203
HIVINT	target	inhibition of HIV integrase in a cell based assay −log(IC50) M	2421	4306
HIVPROT	target	inhibition of HIV protease −log(IC50) M	4311	6274
LOGD	ADME	logD measured by HPLC method	50000	8921
METAB	ADME	percent remaining after 30 min microsomal incubation	2092	4595
NK1	target	inhibition of neurokinin1 (substance P) receptor binding −log(IC50) M	13482	5803
OX1	target	inhibition of orexin 1 receptor −log(K_i) M	7135	4730
OX2	target	inhibition of orexin 2 receptor −log(K_i) M	14875	5790
PGP	ADME	transport by *p*-glycoprotein log(BA/AB)	8603	5135
PPB	ADME	human plasma protein binding log(bound/unbound)	11622	5470
RAT_F	ADME	log(rat bioavailability) at 2 mg/kg	7821	5698
TDI	ADME	time dependent 3A4 inhibitions log(IC50 without NADPH/IC50 with NADPH)	5559	5945
THROMBIN	target	human thrombin inhibition −log(IC50) M	6924	5552
		Additional Data Sets		
2C8	ADME	CYP P450 2C8 inhibition −log(IC50) M	29958	8217
2C9	ADME	CYP P450 2C9 inhibition −log(IC50) M	189670	11730
2D6	ADME	CYP P450 2D6 inhibition −log(IC50) M	50000	9729
A-II	target	binding to Angiotensin-II receptor −log(IC50) M	2763	5242
BACE	target	inhibition of beta-secretase −log(IC50) M	17469	6200
CAV	ADME	inhibition of Cav1.2 ion channel	50000	8959
CLINT	ADME	clearance by human microsome log(clearance) μL/min·mg	23292	6782
ERK2	target	inhibition of ERK2 kinase −log(IC50) M	12843	6596
FACTORXIA	target	inhibition of factor XIa −log(IC50) M	9536	6136
FASSIF	ADME	solubility in simulated gut conditions log(solubility) mol/L	89531	9541
HERG	ADME	inhibition of hERG channel −log(IC50) M	50000	9388
HERG (full data set)	ADME	inhibition of hERG ion channel −log(IC50) M	318795	12508
NAV	ADME	inhibition of Nav1.5 ion channel −log(IC50) M	50000	8302
PAPP	ADME	apparent passive permeability in PK1 cells log(permeability) cm/s	30938	7713
PXR	ADME	induction of 3A4 by pregnane X receptor; percentage relative to rifampicin	50000	9282

Fig. 4.10 Datasets used in drug property prediction

To evaluate QSAR methods, each of these data sets was split into two non-overlapping subsets: a training set and a test set. Although a usual way of doing the split is by random selection. QSAR models are applied prospectively. Predictions are made for compounds not yet tested in the appropriate assay, and these compounds may or may not have analogs in the training set. The best way of simulating this is to generate training and test sets by time-split. For each data set, the first 75% of the molecules assayed for the particular activity form the training set, while the remaining 25% of the compounds assayed later form the test set.

As for the features (i.e., molecular descriptors), each molecule is represented by a list of features, that is, the union of the original atom pair descriptor—atom pairs (AP) [11], and donor-acceptor pair descriptors (DP) [85]. Both AP and DP descriptors are of the following form: atom types i-distance in bonds-atom type j. For AP, atom type includes the element, number of nonhydrogen neighbors, and number of pi electrons; it is very specific. For DP, atom type is one of seven (cation, anion, neutral donor, neutral acceptor, polar, hydrophobe, and others).

Various DNN model architectures have been tried with

- different data preprocessing strategies (original value, logarithm transform, binary transform),
- the number of hidden layers (1–4),
- the number of neurons in the hidden layers (varying from 100 to 4500),
- activation functions (Sigmoid vs ReLU),
- dropout rate,
- initialization (random vs. unsupervised pretraining),

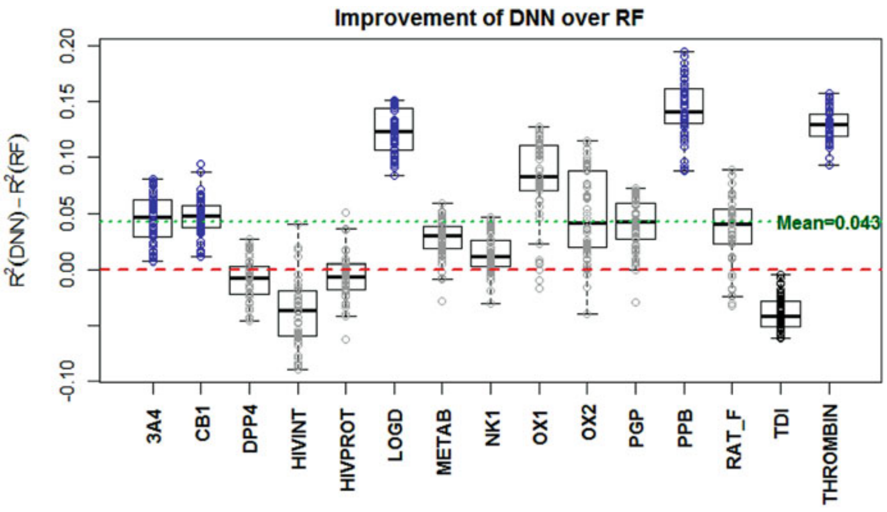


Fig. 4.11 Performance QSAR prediction results (R^2) in comparison to RF model. Red dash line is the average performance of RF model, green dash line is the average performance of DNN models. DNN outperformed RF on 12 out of 15 datasets

- mini-batch sizes and
- the number of epochs.

Results Figure 4.11 demonstrates that DNNs on average outperform RF in 11 out of the 15 data sets. It shows the difference in R^2 between DNNs and RF for each dataset. Each column represents a QSAR dataset, and each circle represents the improvement in R^2 of one DNN model over RF. A positive value means that the corresponding DNN outperforms RF. The average R^2 over all DNNs and all 15 data sets are 0.043 higher than that of RF, or a 10% improvement.

4.5 Exercises

1. Which types of health data do you think can be benefited most by DNN methods? Why?
2. Which types of health data do you think can be benefited least by DNN methods? Why?
3. Which of the following is NOT true about activation functions?
   (a) Activation functions describe non-linear transformation.
   (b) Activation functions are specified by the user when setting up the neural network architectures
   (c) Activation functions are learned directly from the data by neural network models.
   (d) ReLU is able to cope with vanishing gradient problems better than Sigmoid and Tanh.
4. What is NOT true about gradient descent?
   (a) Log-likelihood and likelihood function has the same optimal but log-likelihood is often easier to manipulate.
   (b) Gradient descent is an optimization method for optimizing model parameters.
   (c) Gradient descent is a specific design method for neural network optimization.
   (d) Stochastic gradient descent is a variant of the gradient descent method that is popular for neural networks training.
5. In forward computation, what is the weight $W_{12}^{(1)}$ used for?
   (a) Connect neuron $x1$ from the input layer to output neuron $h2$ in the second layer
   (b) Connect neuron $x2$ from the input layer to output neuron $h1$ in the second layer

   (c) Connect neuron $h1$ from the second layer to output neuron $h2$ in the third layer
   (d) Connect neuron $h2$ from the second layer to output neuron $h1$ in the second layer

6. In the general form of forward computation, the weight matrix $W^{(l)}$ and bias vector $b^{(l)}$ are used to connect?
7. What is true about back propagation?

   (a) Back propagation is an efficient way to compute derivatives on parameters on a neural network.
   (b) Back propagation does not require any form of forward pass of the neural network.
   (c) Most deep learning packages require users to specify the derivatives of each layer in order to perform back propagation.
   (d) Back propagation is a new algorithm invented specifically for training deep learning models.

8. Which is NOT true about Multilayer Neural Networks?

   (a) Multilayer neural networks are computed more efficiently on GPU.
   (b) There is no bias term in the input layer.
   (c) The linear combination of layer 2 is computed as $z_i^{(2)} = \sum_j (w_{ji}^{(1)} x_j + b_j)$

9. Which is NOT true in the readmission study using DNN?

   (a) Multiple layers of DNN can help construct better features before the final classification layer.
   (b) Separate DNNs are trained for the five different disease cohorts.
   (c) DNN models achieved better accuracy than logistic regression models in this study.
   (d) DNN can provide a clear interpretation of its prediction.

10. Why do you think DNN is a good model for QSAR applications?