Artificial neural networks and backpropagation

Dr. Yee Wei Law ⟨yeewei.law@unisa.edu.au⟩

May 3, 2023

Contents

List of acronyms

ANN	artificial neural network	1
DNN	deep neural network	4
MLP	multilayer perceptron	4
MP	McCulloch-Pitts	2
XOR	exclusive OR	4

1 Introduction

An *artificial neural network* (ANN) is a computing scheme represented as a network of artificial neurons, that models the brain's processing functions.

Alternative names for ANNs include parallel distributed processing models, connectionist models, self-organising systems, neurocomputers, neurocomputing systems, and neuromorphic systems [LL96, p. 207].

An *artificial neuron* is a simplistic mathematical model of a biological neuron, which consists of three main parts (see Figure 1): 1️⃣ soma, 2️⃣ dendrites and 3️⃣ axon.

- The axon terminates in strands called *axon terminals*, and each axon terminal terminates in a bulb-like organ called a *synapse*.
- A neuron signals other neurons by sending electrical pulses through its synapses.

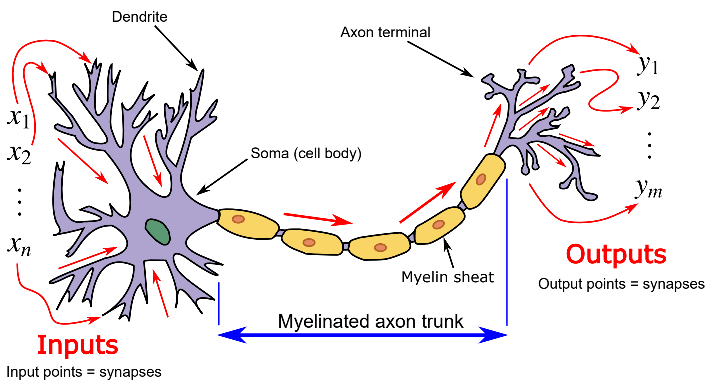

Figure 1: Structure of a typical neuron. Figure from Wikipedia.

- A receiving neuron fires if its electrical potential reaches a *threshold*, and an action potential of fixed strength and duration is transmitted through the axon to the synaptic junctions to other neurons.
- Synapses are *excitatory* if they let passing pulses cause the firing of the receiving neuron, or *inhibitory* if otherwise.

The McCulloch-Pitts (MP) neuron in Figure 2 is the earliest example of an artificial neuron.

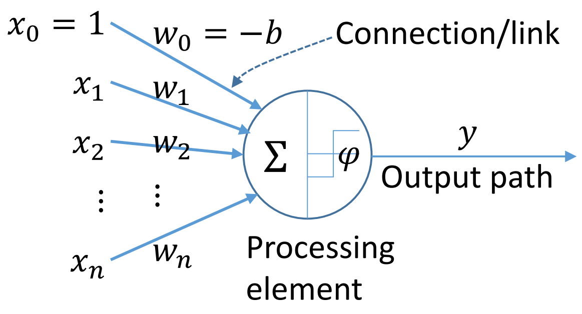

Figure 2: An MP neuron. The weight w_0 is usually negative so b is usually positive.

The MP model captures the firing characteristics of the biological neuron through the *activation function* $\varphi(\cdot)$, which is applied to the weighted sum of the input signals:

$$y = \varphi\left(\sum_{i=1}^{n} w_i x_i - b\right) = \varphi\left(\sum_{i=0}^{n} w_i x_i\right), \qquad (1)$$

where $w_1, \dots, w_n$ are the weights, $x_i, \dots, x_n$ are the inputs, and b is the *bias* or *firing threshold*.

In Figure 2,

- The bias (in negative) is treated as a weight corresponding to a unit input.
- A processing element, also called a *node* or *unit*, comprises a summing junction $\sum$ and an activation function φ.

φ plays a crucial role in the learning capabilities of a neuron.

Traditional options for φ

- are either bipolar (ranges from -1 to $+1$) or unipolar (ranges from 0 to 1);
- include, as listed in [LL96, Sec. 9.2.1] and [Hay09, pp. 13-14], the Heaviside step function, the signum (also called *hard limiter*) function, and sigmoid functions (see Definition 1).

Definition 1: Sigmoid function [HM95]

A bounded differentiable real function that is defined for all real input values and that has a positive derivative everywhere.

In simpler terms, a sigmoid function is a strictly increasing but bounded function that is shaped like an 'S'.

The standard sigmoid functions are:

- The *logistic* function $\varphi(x) = (1 + e^{-x})^{-1}$, which is often considered to be *the* sigmoid function, rather than a type of sigmoid function.
- The *hyperbolic tangent* function $\varphi(x) = \tanh(x)$.

When a processing element

- uses either the Heaviside step function or the signum function as the activation function, it is called a *linear threshold unit* or *threshold logic unit* [LL96, G22];
- uses a sigmoid function, it is called a *linear graded unit* [LL96, p. 209].

Over the years, many more options for the activation function φ have been proposed [DSC22].

Further discussion of φ, including the role of φ in learning and contemporary options for φ, is deferred to the Cyber Engineering Knowledge Base.

Subsequently,

- Sec. 2 covers the network structures/architectures, i.e., how neurons are connected to each other.
- Sec. 3 covers the mechanism of learning called backpropagation.

2 Network architectures

Neurons perform computation in conjunction, which involves information passing from one layer of neurons to another.

By design, there are no connections between neurons of the same layer.

Most ANNs use either the feedforward or the recurrent configuration.

- ***Feedforward network***: Information is fed in the forward direction only, from the input layer through the *hidden layers* (if any) to the output layer.

  Hidden layers are so called because they are hidden from the input and output [Hay09, p. 22].

  A *perceptron*, as shown in Figure 3, is a single-layer feedforward neural network [LL96, Ch. 10], and thus has no hidden layers.

  The input layer is not counted because it does not involve any computation, and it is often referred to as layer 0.

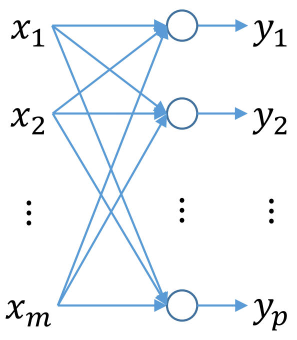

Figure 3: A perceptron.

  The original perceptron by Rosenblatt [Ros57, Ros58] was built around a single MP neuron [Hay09, Sec. 1.2].

  Perceptrons were designed to solve *linearly separable* classification problems [Hay09, Sec. 1.2].

Figure 4 shows an example of a linearly separable problem, where members of two different classes can be separated by a line (equivalently, plane in a three-dimensional space, hyperplane a higher-dimensional space).

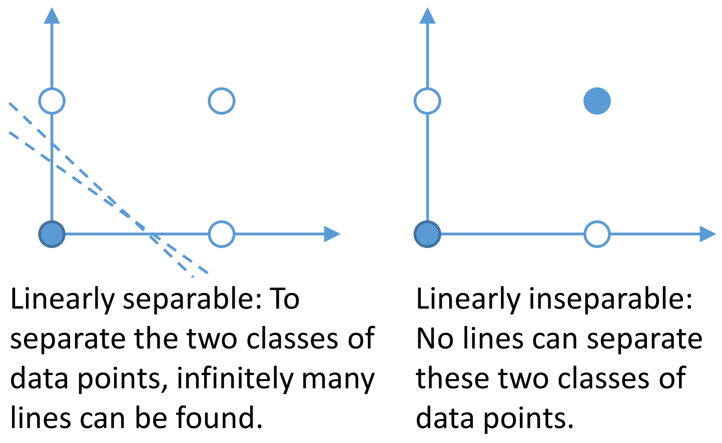
Linearly separable: To separate the two classes of data points, infinitely many lines can be found.
Linearly inseparable: No lines can separate these two classes of data points.

Figure 4: An example of a linearly separable problem and an example of a linearly inseparable problem.

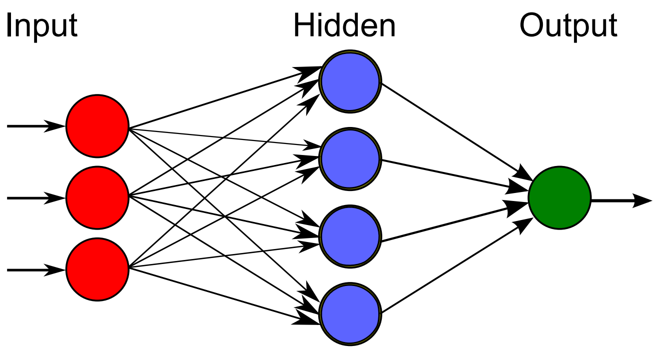

Figure 5: An example of a multilayer perceptron: a fully connected 3-4-1 network.

A simple but *not* linearly separable problem is the computation of exclusive OR (XOR), the solution of which requires a multilayer approach.

A *multilayer perceptron* (MLP), as shown in Figure 5, is a feedforward neural network with two or more layers of neurons (i.e., one or more hidden layers) and differentiable nonlinear activation functions [Hay09, Sec. 4.1].

Naming convention: An MLP with m source nodes, h_i nodes in the ith hidden layer, and p nodes in the output layer is referred to as an m-h_1-$h_2 \cdots$-p network.

When every node in a layer is connected to every other node in the preceding layer, that layer is *fully connected* or *dense*.

A network whose layers (not counting the input layer) are all dense is *fully connected* [Hay09, p. 23].

The MLP is the most widely used feedforward configuration.

An MLP with more than three hidden layers is called a *deep neural network* (DNN) [SCYE17].

- ***Recurrent network***: When outputs are redirected as inputs to the same or preceding layers, we get a feedback configuration.

  Feedback networks with closed loops are called *recurrent networks*.

Many machine learning problems can readily be viewed as *function approximation* problems.

Leshno et al. [LLPS93] proved that a multilayer feedforward network with a locally bounded piecewise continuous activation function (which an MLP is a special case of) can approximate any continuous function to any degree of accuracy if and only if the activation function is not a polynomial.

More encouragingly, Barron [Bar93] proved that an MLP with only one hidden layer can achieve an integrated squared error of order $O(1/n)$, where n is the number of hidden neurons, independent of the dimension of the input vector space.

The *universal approximation* property described above makes MLPs useful for control, because it enables a controller to construct a model of a plant, and determine control actions based on the model.

However, Leshno et al.'s and Barron's theorems are existential rather than constructive, i.e., they do not specify how to construct the learning algorithms.

3 Learning by error backpropagation

The central role of ANNs is *learning*, which is the building of a mathematical model relating inputs to outputs.

A learning process has three phases:

1. In the **training** phase, the learner derives a model based on samples from a *training set*, i.e., derives f given x and $y = f(x)$.

   Training adjusts the weights of an ANN but not the *hyperparameters*, which are settings for controlling the behaviour of the learning algorithm [GBC16, Sec. 5.3].

   Capacity hyperparameters are those that improve the accuracy of the learning algorithm, e.g., polynomial degree in polynomial regression (see Example 1).

   If learnt on the training set, capacity hyperparameters are always maximised, causing *overfitting*.

Example 1

When performing polynomial regression (i.e., fitting a polynomial to an input-output dataset), suppose training data was generated by sampling a quadratic function. Figure 6 shows that

- When we fit a linear function to the data, we get an underfit.
- A quadratic function fit to the data generalises well to unseen points.
- When we fit a 9th-degree polynomial to the data, we get an overfit.

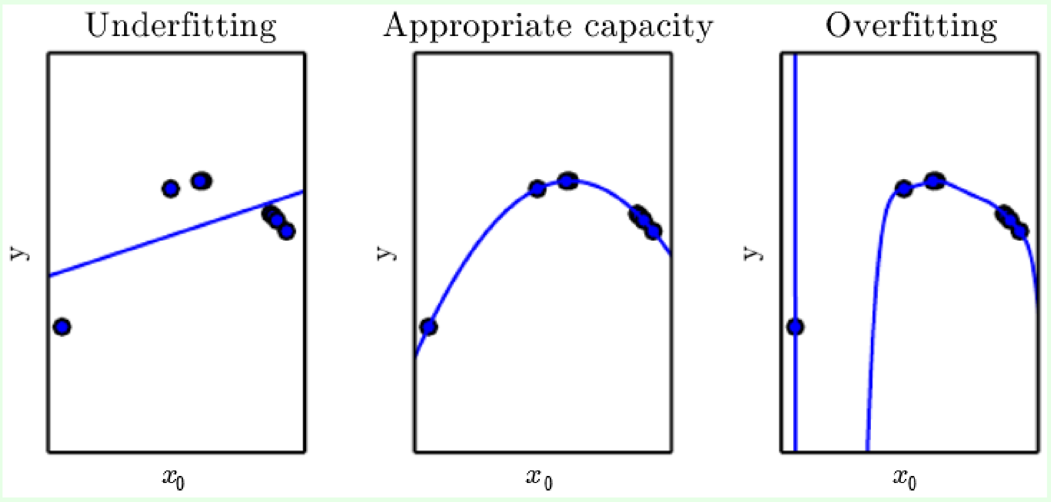

Figure 6: Examples of underfitting, appropriate capacity and overfitting [GBC16, Figure 5.2].

Thus, the degree of the fitting polynomial is an example of a capacity hyperparameter [GBC16, Sec. 5.3].

A higher-degree polynomial tends to fit the training data better than a lower-degree polynomial, but does not necessarily generalise well.

2. In the **validation** phase, the learner uses a *validation set* to estimate the *generalisation error* during or after training to determine how the hyperparameters should be updated. In other words, the subset of data used to guide the selection of hyperparameter values is called a validation set [GBC16, p. 119].

   Typically 80% of training data is used for training while the remainder is used for validation [GBC16, p. 119].

3. In the **testing phase**, the learner applies that model to samples from a *test set*, i.e., compares $f_{\text{learnt}}(x)$ with y.

   Whereas validation is an intermediate evaluation that determines the final model (e.g., by choosing the best out of an array of candidate classifiers) to be used, testing is an evaluation of the performance of the final model [Cic15].

Types of learning include:

- **Supervised learning**: For every input vector, an ANN receives the desired output vector.

  In training, the ANN's weights are adjusted to minimise the difference between its output and the desired output.

  Training is repeated, in the sense that previously used training samples maybe reapplied in a different order, until the weights converge to steady-state values.

  Supervised learning is also called *learning with a teacher*.

- **Unsupervised learning**: Without additional information, an ANN has to discover patterns such as clusters in the input data.

  In training, the ANN's weights are adjusted to reflect the distribution of the input samples.

The procedure used to perform the learning process is called a learning algorithm or learning rule [LL96, p. 212]:

- *Parameter learning* involves adjusting the synaptic weights.
- *Structure learning* involves modifying the network structure, including the number of neurons and their connection types.

Most existing learning algorithms are parameter learning, and among them, the *error backpropagation* (backpropagation or backprop for short) algorithm is the most influential and serves as the basis for many later algorithms.

⚠ Backprop and feedback are different concepts. In fact, backprop was designed for MLPs rather than feedback networks.

Historically, the backprop algorithm existed before Rumelhart et al. [RHW86] popularised it and in effect rejuvenated interests in ANNs in the 1980s.

Unlike the perceptron algorithm [MR88], backprop is applicable to both linearly separable and linearly inseparable problems. Furthermore, it can be used for both supervised and unsupervised learning, but only supervised learning is considered here.

The algorithm is comprehensively discussed in [Hay99, Sec. 4.3]. Here, a summary of the algorithm is given, for which the symbols in Table 1 and Figure 7 are used.

Table 1: Symbols for discussing backprop

N	Total number of time steps	φ	Activation function
$d_k[n]$	Desired output of node k at time step n	$\varphi'(v)$	$\frac{\partial\varphi(v)}{\partial v}$
$y_k[n]$	Output of node k at time step n	$w_{kj}[n]$	Weights for neuron k at time step n
$e_k[n]$	$d_k[n] - y_k[n]$	$\delta_k[n]$	Local gradient for neuron k at time step n
$v_k[n]$	Output of summing junction of node k at time step n	η	Learning rate

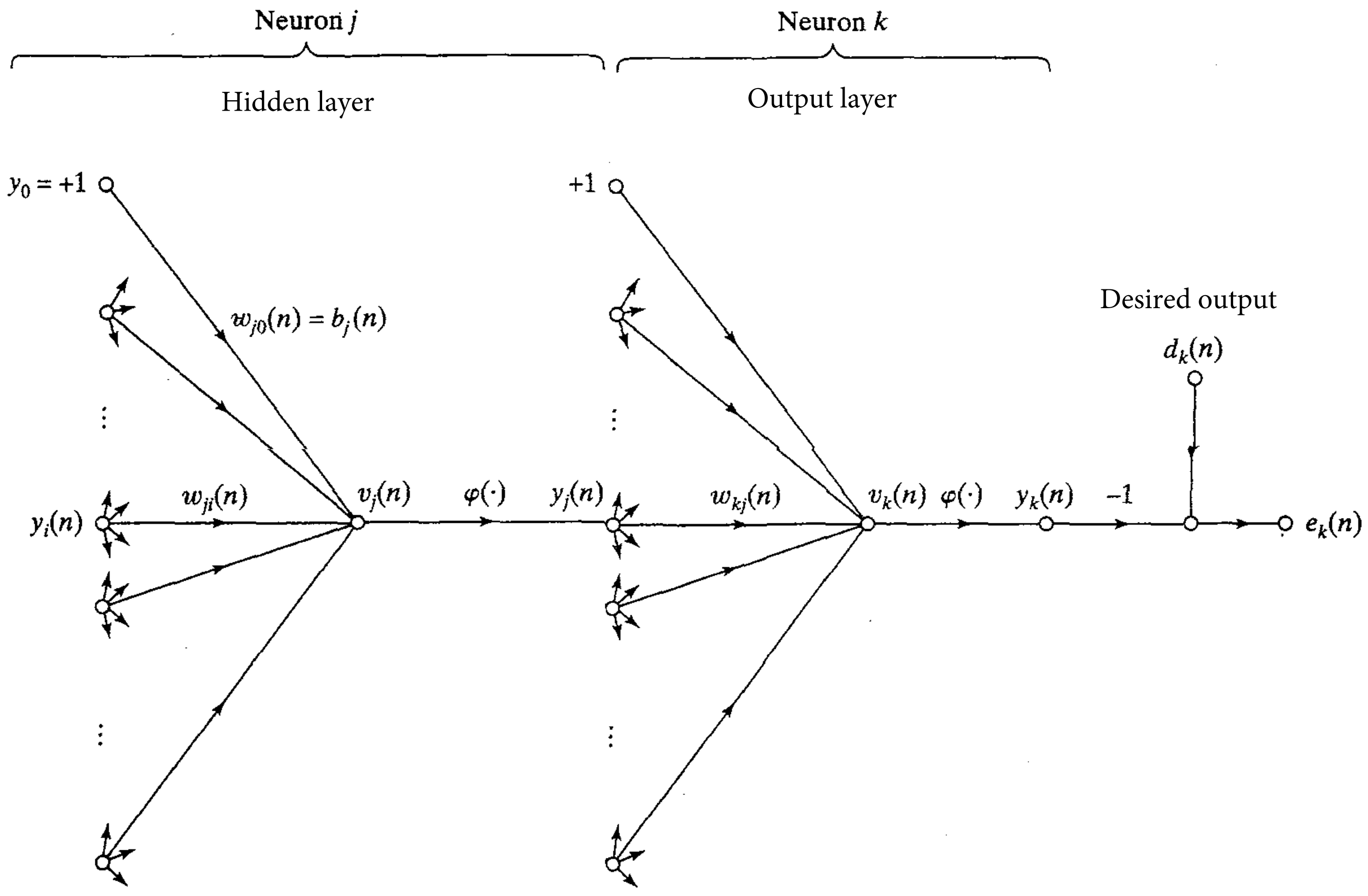

Figure 7: Signal-flow graph for backprop [Hay99, FIGURE 4.4].

The main idea of the algorithm is to minimise the cost/loss function, expressed as the *average squared error energy*:

$$\mathscr{E}_{\text{av}} \triangleq \frac{1}{N}\sum_{n=1}^{N}\mathscr{E}[n], \qquad \mathscr{E}[n] \triangleq \frac{1}{2}\sum_{k} e_k^2[n], \tag{2}$$

where k indexes all output neurons, and $e_k[n]$ is as defined in Table 1.

Detail: Logit, softmax and cross-entropy loss [Cha19, pp. 11-14]

In the contemporary formulation, the outputs of an ANN for a classification task serve as probability values for all possible classes; the class associated with the highest probability value is chosen as the predicted class.

For the outputs to serve as probability values, the outputs must be nonnegative and add up to 1; the mathematical trick to ensure the outputs satisfy these conditions is using the *softmax* function as the activation function of the output layer:

$$\sigma(\vec{y}[n])_k = \frac{e^{y_k[n]}}{\sum_k e^{y_k[n]}}. \tag{3}$$

The non-normalised values leaving the output neurons and entering the softmax function (acting as a normalisation function) are called *logits*.

The contemporary alternative to the loss function in (2) is the *cross-entropy loss function*, defined as the negative log function of the output of the softmax function corresponding to the ground-truth class.

The backprop algorithm runs in two passes:

- In the ***forward pass***, the output of each neuron is computed starting from the first hidden layer to the output layer.

  All synaptic weights remain unchanged.

  At the output layer, the error for each neuron is computed, i.e., for neuron k, the error is $e_k[n]$ as defined in Table 1.
- In the ***backward pass***, the first step is to compute the local gradient for each output neuron, i.e., for neuron k in the output layer, the *local gradient* $\delta_k[n]$ is computed (using the chain rule in the process) as

$$\delta_k[n] \triangleq -\frac{\partial \mathscr{E}[n]}{\partial v_k[n]} = e_k[n]\,\varphi'(v_k[n]). \tag{4}$$

Detail: Derivation of (4)

By the chain rule,

$$\frac{\partial \mathscr{E}[n]}{\partial v_k[n]} = \frac{\partial \mathscr{E}[n]}{\partial e_k[n]}\frac{\partial e_k[n]}{\partial y_k[n]}\frac{\partial y_k[n]}{\partial v_k[n]} = e_k[n]\cdot(-1)\cdot\varphi'(v_k[n]).$$

The weight $w_{kj}[n]$ is updated as

$$w_{kj}[n+1] = w_{kj}[n] + \Delta w_{kj}[n], \tag{5}$$

where $\Delta w_{kj}[n]$ is defined by the *delta rule*, based on the method of gradient descent, as

$$\Delta w_{kj}[n] \triangleq -\eta\frac{\partial \mathscr{E}[n]}{\partial w_{kj}[n]} = \eta\delta_k[n]y_j[n]. \tag{6}$$

η is the *learning rate*. The negative sign in (6) effects gradient descent, i.e., seeks direction of change in w that reduces $\mathscr{E}$.

Detail: Derivation of (6)

By the chain rule,

$$\begin{aligned}\frac{\partial \mathcal{E}[n]}{\partial w_{kj}[n]} &= \frac{\partial \mathcal{E}[n]}{\partial e_k[n]} \frac{\partial e_k[n]}{\partial y_k[n]} \frac{\partial y_k[n]}{\partial v_k[n]} \frac{\partial v_k[n]}{\partial w_{kj}[n]} \\ &= e_k[n] \cdot (-1) \cdot \varphi'(v_k[n]) \cdot y_j[n] = -e_k[n]\, \varphi'(v_k[n]) y_j[n].\end{aligned}$$

Substituting (4) into the preceding equation gives us

$$\frac{\partial \mathcal{E}[n]}{\partial w_{kj}[n]} = -\delta_k[n] y_j[n].$$

The error terms are propagated backward to the last hidden layer in the following manner.
For neuron j in this hidden layer, the local gradient $\delta_j[n]$ is defined as

$$\delta_j[n] \triangleq -\frac{\partial \mathcal{E}[n]}{\partial v_j[n]} = \left(\sum_k \delta_k[n] w_{kj}[n] \right) \varphi'(v_j[n]), \tag{7}$$

where k indexes all neurons connected to neuron j on the right side neuron j.

By comparing equations (4) and (7), notice that the error term is e_k for the output neuron k, but $\sum_k \delta_k[n] w_{kj}[n]$ for the hidden neuron j.

Detail: Derivation of (7)

By the chain rule,

$$\begin{aligned}\frac{\partial \mathcal{E}[n]}{\partial v_j[n]} &= \frac{\partial \mathcal{E}[n]}{\partial y_j[n]} \frac{\partial y_j[n]}{\partial v_j[n]} \\ &= \left(\sum_k \frac{\partial \mathcal{E}[n]}{\partial e_k[n]} \frac{\partial e_k[n]}{\partial y_j[n]} \right) \varphi'(v_j[n]) \\ &= \left(\sum_k e_k[n] \frac{\partial e_k[n]}{\partial v_k[n]} \frac{\partial v_k[n]}{\partial y_j[n]} \right) \varphi'(v_j[n]) \\ &= \left(\sum_k e_k[n] \cdot -\varphi'(v_k[n]) \cdot w_{kj}[n] \right) \varphi'(v_j[n]).\end{aligned}$$

Substituting (4) into the above gives us

$$\frac{\partial \mathcal{E}[n]}{\partial v_j[n]} = -\left(\sum_k \delta_k[n] w_{kj}[n] \right) \varphi'(v_j[n]).$$

The weight $w_{ji}[n]$ is updated as

$$w_{ji}[n+1] = w_{ji}[n] + \Delta w_{ji}[n] = w_{ji}[n] - \eta \frac{\partial \mathcal{E}[n]}{\partial w_{ji}[n]} = w_{ji}[n] + \eta \delta_j[n] y_i[n]. \tag{8}$$

Compare (8) with (5)–(6).

The procedure is repeated for the layer to the left of the current layer, i.e., (7) and (8) are invoked with the appropriate subscripts, until the first hidden layer is reached.

Backpropagation with the delta rule has several limitations [Hay99, Sec. 4.16]:

- Mostly notably, convergence is not guaranteed and is slow. In fact, empirical observations indicate that local convergence rates are linear.
- Secondly, it might be trapped in a local minimum, where a small change in the synaptic weights increases the cost function. This is undesirable especially if the global minimum is significantly lower than the local minimum.
- Furthermore, backpropagation does not scale for certain problems, in the sense that the computation time increases exponentially with the number of inputs.

The equations (4)–(8) are based on the definition of the loss function in (2). For any loss function, the general method of *stochastic gradient descent* is applicable.

Adam is a variation of the stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments [KB15].

- Adam is implemented in major machine learning libraries, including PyTorch, Tensorflow and scikit-learn.

4 Brief historical notes

Research on the ANNs dates back to the 1940s, when McCulloch and Pitts proposed their neuron model [SVdM96, pp. 3-4].

Rosenblatt invented perceptrons in the 1950s [Ros57, Ros58].

Minsky and Papert showed the theoretical limits of perceptrons in the 1960s, and following that progress stalled for 20 odd years.

It is not until Hopfield revived the field in the 1980s that self-organising map and backpropagation were proposed.

Today, ANNs continue to blossom, especially in the form of DNNs.

Algorithmically, ANNs can be used for pattern matching and classification, clustering, function approximation, prediction/forecasting, optimisation, content-addressable memory, vector quantisation, control and more [JMM96].

Due to this rich set of features, ANNs have enabled a broad range of applications in engineering, information technology, health sciences, natural sciences, finance, security, and many other areas [JMM96].

5 References

[Bar93] A. Barron, Universal approximation bounds for superpositions of a sigmoidal function, *IEEE Transactions on Information Theory* **39** no. 3 (1993), 930–945. https://doi.org/10.1109/18.256500.

[Cha19] E. Charniak, *Introduction to Deep Learning*, MIT Press, 2019. Available at https://ebookcentral.proquest.com/lib/unisa/reader.action?docID=6331506.

[Cic15] P. Cichosz, *Data Mining Algorithms: Explained Using R*, Wiley, 2015. https://doi.org/10.1002/9781118950951.

[DSC22] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, Activation functions in deep learning: A comprehensive survey and benchmark, *Neurocomputing* **503** (2022), 92–108, code at https://github.com/shivram1987/ActivationFunctions. https://doi.org/10.1016/j.neucom.2022.06.111.

[G22] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 3rd ed., O'Reilly Media, Inc., 2022. Available at https://learning.oreilly.com/library/view/hands-on-machine-learning/9781098125967/.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016. Available at http://www.deeplearningbook.org.

[HM95] J. Han and C. Moraga, The influence of the sigmoid function parameters on the speed of backpropagation learning, in *From Natural to Artificial Neural Computation* (J. Mira and F. Sandoval, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 195–201.

[Hay99] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., Pearson Education, Inc., 1999.

[Hay09] S. Haykin, *Neural Networks and Learning Machines*, 3rd ed., Pearson Education, Inc., 2009.

[JMM96] A. Jain, J. Mao, and K. M. Mohiuddin, Artificial neural networks: a tutorial, *Computer* **29** no. 3 (1996), 31–44. https://doi.org/10.1109/2.485891.

[KB15] D. P. Kingma and J. L. Ba, Adam: A method for stochastic optimization, in *ICLR*, 2015. Available at https://arxiv.org/abs/1412.6980.

[LLPS93] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, Multilayer feedforward networks with a nonpolynomial activation function can approximate any function, *Neural Networks* **6** no. 6 (1993), 861–867. https://doi.org/10.1016/S0893-6080(05)80131-5.

[LL96] C. Lin and C. Lee, *Neural Fuzzy Systems: A Neuro-Fuzzy Synergism to Intelligent Systems*, Prentice Hall, 1996.

[MR88] J. McClelland and D. Rumelhart, *Explorations in Parallel Distributed Processing*, MIT Press, 1988.

[Ros57] F. Rosenblatt, *The Perceptron: A Perceiving and Recognizing Automaton (Project Para)*, Tech. Report 85-460-1, Cornell Aeronautical Laboratory, Inc., January 1957.

[Ros58] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, *Psychological Review* **65** no. 6 (1958), 386–408. Available at https://oce.ovid.com/article/00006832-195811000-00007.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by back-propagating errors, *Nature* **323** (1986), 533–536. https://doi.org/10.1038/323533a0.

[SVdM96] J. A. Suykens, J. P. Vandewalle, and B. de Moor, *Artificial Neural Networks for Modelling and Control of Non-Linear Systems*, Springer New York, NY, 1996. https://doi.org/10.1007/978-1-4757-2493-6.

[SCYE17] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, Efficient processing of deep neural networks: A tutorial and survey, *Proceedings of the IEEE* **105** no. 12 (2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740.