Neural Network Essential

Ke Chen

Department of Computer Science, The University of Manchester

Ke.Chen@manchester.ac.uk

Outline

Introduction

History and application

Neuron model

Neuron, artificial neuron model, activation (transfer) functions

Neural Architecture

Feedforward versus recurrent, fully connected versus partially connected, homogeneous components versus heterogeneous components

Learning

Loss functions, stochastic gradient optimisation, back-propagation (BP) algorithm, practical issues

Introduction

- In **1943**, Neural networks (NNs) were originated by W. McCulloch & W. Pitts who created computational models to simulate neurons in human brain.
- In **1957**, F. Rosenblatt proposed the first ever learning algorithm named perceptron to train artificial NNs for binary classification tasks. (1st wave of neural networks)
- In **1969**, M. Minsky & S. Pappert wrote a book entitled perceptrons that pointed out the limitation due to a lack of learning algorithm for multilayered NNs.
- In **1986**, Parallel Distributed Processing (PDP) Project led to several seminal works in neural computation where the most influential one is back-propagation learning developed by D. Rumelhart, G. Hinton & R. Williams. (2nd wave of neural networks)
- From **end of 1990s** to **2006**, difficulties in training deep NNs diverted ML research to simple yet theoretically justified learning models, e.g. SVM and Adaboost.
- In **2006**, G. Hinton & his students proposed a new learning strategy to train deep NNs and further coined the term deep learning to replace neural networks.
- Since **2006**, ML has shifted its focus to deep learning, which lifts AI to a new era. In **2018**, G. Hinton, Y. Bengio & Y. LeCun received ACM Alan Turing Award for their contributions in deep learning. (3rd wave of neural networks)

Introduction

As an underpinning technology, deep learning has been applied to many AI domains.

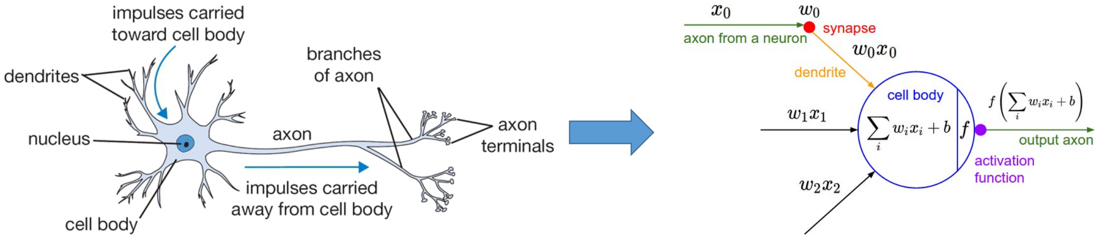
- **Computer vision**: deep learning has become a pre-dominated techniques and led to super-human performance in some visual recognition tasks.
- **Speech recognition**: as one of the most important technical components, deep learning dramatically improves recognition accuracy, e.g. Google voice recogniser.
- **Natural language processing**: as one of the most important technical components, deep learning has substantially improve information retrieval and machine translation, e.g., Google translate.
- **Game agent**: as one of the most important technical components, deep learning has created game agents outperforming human beings, e.g., Alpha Go and Atari agents.
- **Miscellaneous**: deep learning has played a crucial role in many real applications such as industry 4.0 manufacture automation and medical diagnosis and treatment.

Neuron Model

- Biological neuron has been well studied in biology and neuroscience.
- Computational neuron model may be biologically plausible or artificial.
- Biologically plausible model: modelling all biological mechanisms and functions via differential equation system, e.g., Hodgkin–Huxley model for spiking neuron
- Artificial model: modelling main functions abstractly by ignoring biological meaning

$$a = \mathbf{w}^T \mathbf{x} + b = \sum_i w_i x_i + b, \quad \text{output} = f(a),$$

w_i: weights, b: bias and x_i: input, a: action potential, $f(\cdot)$: activation function

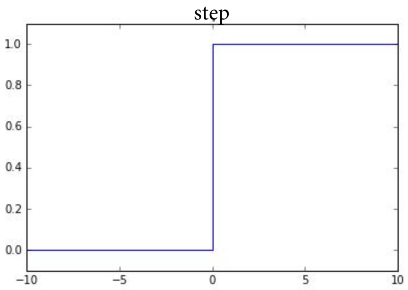


Activation Function

- Step (perceptron) function

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

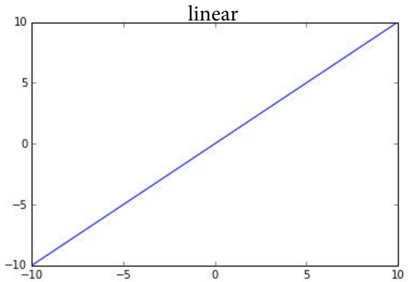
Step function is discontinuous hence has no gradient.



- Linear (identity) function

$$f(x) = x$$

$$\frac{df(x)}{dx} = 1.$$

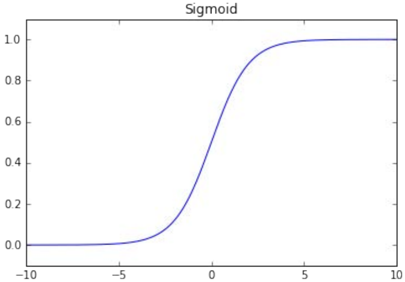


Activation Function

- Sigmoid (logistic) function

$$f(x) = \frac{1}{1 + e^{-x}}.$$

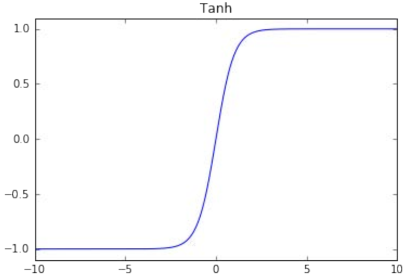
$$\frac{df(x)}{dx} = f(x)(1 - f(x)).$$



- Hyperbolic tangent function

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}.$$

$$\frac{df(x)}{dx} = 1 - f^2(x).$$

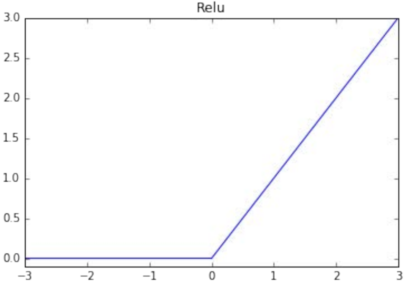


Activation Function

- Rectified linear unit (ReLU) function

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

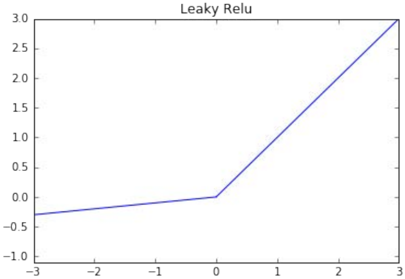
$$\frac{df(x)}{dx} = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$



- Leaky ReLU function

$$f(x) = \begin{cases} 0.1x & x < 0 \\ x & x \geq 0 \end{cases}$$

$$\frac{df(x)}{dx} = \begin{cases} 0.1 & x < 0 \\ 1 & x \geq 0 \end{cases}$$



Activation Function

- Scaled-exponential linear unit (SeLU) function

$$f(\alpha, x) = \begin{cases} \alpha\left(e^{x} - 1\right) & x < 0 \\ x & x \geq 0 \end{cases}, \qquad \frac{df(x)}{dx} = \begin{cases} f(\alpha, x) + \alpha & x < 0 \\ 1 & x \geq 0 \end{cases}$$

- Maxout function

$$f(\boldsymbol{x}) = \max_{i=1}^{|\boldsymbol{x}|} x_i, \qquad \frac{df(\boldsymbol{x})}{dx_j} = \begin{cases} 1 & j = \mathrm{argmax}_{i=1}^{|\boldsymbol{x}|} x_i \\ 0 & j \neq \mathrm{argmax}_{i=1}^{|\boldsymbol{x}|} x_i \end{cases}$$

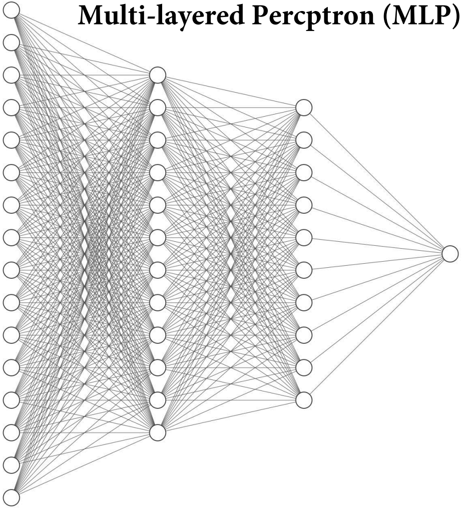
- Softmax function

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{|\boldsymbol{x}|} e^{x_j}}, \qquad \frac{df(x_i)}{dx_j} = f(x_i)\Big(\delta(i, j) - f(x_j)\Big) \quad i = 1, 2, \cdots, |\boldsymbol{x}|.$$

Feedforward Neural Networks

- Fully-connected: all neurons in one layer connected to all in its successive layers
- Homogeneous: all neurons apart from those in input/output layer are the same.

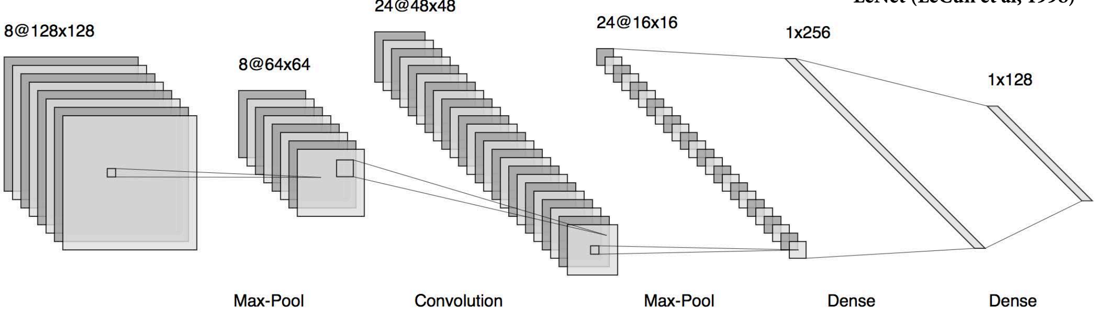
Multi-layered Percptron (MLP)



Feedforward Neural Networks

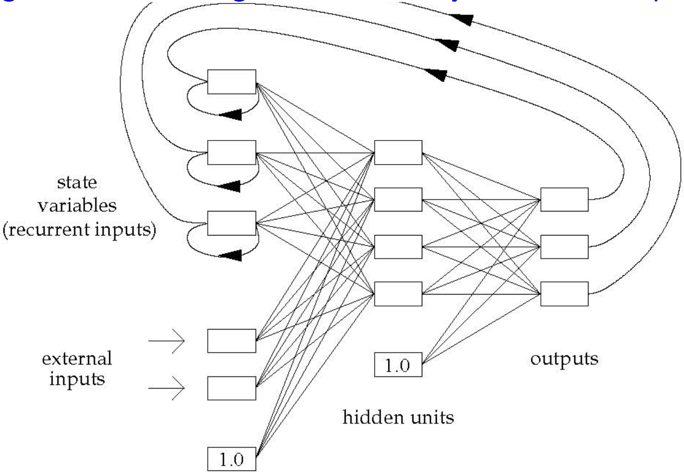
- Partially-connected: neurons in a layer only connected to some in its successive layer
- Heterogeneous: neurons in different layers are various for different purposes
- For example, Convolutional Neural Networks (CNNs)

LeNet (LeCun et al, 1998)



Recurrent Neural Networks

- Recurrent-connected: neurons with feedback connections to neurons in previous and/or the same layers (lateral connection)
- May be homogeneous or heterogeneous, and fully-connected or partially-connected



Loss Function

Given a training dataset, $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{|\mathcal{D}|}$, loss functions are used to train NNs with parameters (weights and bias), Θ (a collective notation of all parameters).

- **Mean squared error (MSE) loss** for regression

$$\mathcal{L}(\Theta; \mathcal{D}) = \frac{1}{2|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|}||\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i||^2.$$

$\hat{\boldsymbol{y}}_i$: output of NNs for input $\boldsymbol{x}_i$ and linear activation function used in output layer

- **Cross-entropy loss** for binary classification

$$\mathcal{L}(\Theta; \mathcal{D}) = -\frac{1}{|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|}\Big(y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\Big), \quad y_i \in \{0, 1\}.$$

$\hat{y}_i$: output of NNs for input $\boldsymbol{x}_i$ and sigmoid activation function used in output layer

Loss Function

Given a training dataset, $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{|\mathcal{D}|}$, loss functions are used to train NNs with parameters (weights and bias), Θ (a collective notation of all parameters).

- **Categorical cross-entropy loss** for C-class classification ($C > 2$)

$$\mathcal{L}(\Theta; \mathcal{D}) = -\frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \sum_{j=1}^{C} y_{ij} \log \hat{y}_{ij}, \quad \boldsymbol{y}_i = \{y_{i1}, y_{i2}, \cdots, y_{iC}\}, \; \hat{\boldsymbol{y}}_i = \{\hat{y}_{i1}, \hat{y}_{i2}, \cdots, \hat{y}_{iC}\}.$$

  $\hat{\boldsymbol{y}}_i$: output of NNs for input $\boldsymbol{x}_i$ and softmax activation function used in output layer

- **Regularised loss** for generalisation and exclusion of ill-posed solution

$$\mathcal{L}_R(\Theta; \mathcal{D}) = \mathcal{L}(\Theta; \mathcal{D}) + \lambda \mathcal{R}(\Theta),$$

  where $\mathcal{R}(\Theta)$ is a regularisation penalty and λ is a trade-off coefficient. For instance, weight decay is often used for regularisation, $\mathcal{R}(\Theta) = \frac{1}{2} \sum_{\text{all } \boldsymbol{w}} ||\boldsymbol{w}||^2$.

- **Stochastic Gradient Optimisation**
  1. Randomly initialise all the parameters, Θ_0.
  2. In each epoch, randomly split a training dataset, $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{|\mathcal{D}|}$, into K mini-batches, $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_K$, with equal size $|\mathcal{B}|$, e.g., $|\mathcal{B}| = 16$.
  3. Update all parameters via gradient descent on a mini-batch basis

$$\Theta_{k+1} \leftarrow \Theta_k - \frac{\eta}{|\mathcal{B}|} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{B}_{k+1}} \nabla_{\Theta} \mathcal{L}(\Theta; \boldsymbol{x}, \boldsymbol{y})|_{\Theta=\Theta_k}, \;\; k = 0, 1, \cdots, K$$

     where $0 < \eta < 1$ is a learning rate. Θ_0 refers to the one after last epoch.
  4. Repeat steps 2 and 3 until a stopping condition is satisfied.
- Add momentum, e.g., $\Delta\Theta_k = \Theta_k - \Theta_{k-1}$, to update rule to speed up training

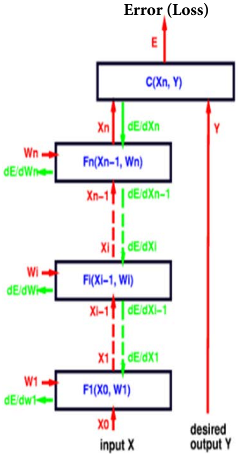
$$\Theta_{k+1} \leftarrow \Theta_k - \frac{\eta_1}{|\mathcal{B}|} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{B}_{k+1}} \nabla_{\Theta} \mathcal{L}(\Theta; \boldsymbol{x}, \boldsymbol{y})|_{\Theta=\Theta_k} + \eta_2 \Delta\Theta_k, \;\; k = 0, 1, \cdots, K$$

  where $0 < \eta_1 < 1$ and $0 < \eta_2 < 1$ are different learning rates.

Fact: Optimisation package, Adam, can compute gradient for any loss of DNNs automatically.

Back-propagation Procedure

- Main stages: forward propagation, backward gradient propagation, parameter update
- Compute gradients of a cost (loss) function with respect to parameters, weights and biases, associated with different layers by applying chain rule recursively



LeCun et al (1998)

- $\frac{\partial E}{\partial X_n} = \frac{\partial C(X_n, Y)}{\partial X_n}$
- $\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial X_{n-1}}$
- $\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial W_n}$
- $\frac{\partial E}{\partial X_{n-2}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial X_{n-2}}$
- $\frac{\partial E}{\partial W_{n-1}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial W_{n-1}}$
- ....etc, until we reach the first module.
- we now have all the $\frac{\partial E}{\partial W_i}$ for $i \in [1, n]$.

Back-propagation Algorithm

- **Forward propagation**
  An MLP has L hidden layers with neurons of activation function, $f(\cdot)$ and output layer of activation function, $g(\cdot)$. For training example, $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{B}_{k+1}$ (mini-batch),
  1. Input: $\boldsymbol{h}^{(0)} \leftarrow \boldsymbol{x}$; (input layer viewed as layer 0)
  2. Compute activation of neurons in hidden layers: for $l = 1, 2, \cdots, L$, compute
  $$\boldsymbol{a}^{(l)}(\boldsymbol{x}) = W_k^{(l)} \boldsymbol{h}^{(l-1)}(\boldsymbol{x}) + \boldsymbol{b}_k^{(l)}, \quad \boldsymbol{h}^{(l)} \leftarrow \boldsymbol{f}\left(\boldsymbol{a}^{(l)}(\boldsymbol{x})\right).$$
  3. Compute activation of output units: for output layer viewed as layer $L+1$,
  $$\boldsymbol{a}^{(L+1)}(\boldsymbol{x}) = W_k^{(L+1)} \boldsymbol{h}^{(L)}(\boldsymbol{x}) + \boldsymbol{b}_k^{(L+1)}, \quad \hat{\boldsymbol{y}} = \boldsymbol{h}^{(L+1)} \leftarrow \boldsymbol{g}\left(\boldsymbol{a}^{(L+1)}(\boldsymbol{x})\right).$$
  4. Compute loss of this training example: $\mathcal{L}(\Theta_k; \boldsymbol{x}, \boldsymbol{y})$, where $\Theta_k = \left\{W_k^{(l)}, \boldsymbol{b}_k^{(l)}\right\}_{l=1}^{L}$.
  (Step 4 is optional but required by an early stop.)

Back-propagation Algorithm

- **Backward gradient propagation**
  1. Compute gradient at output layer:

$$\boldsymbol{\delta}^{(L+1)}(\boldsymbol{x},\boldsymbol{y}) \leftarrow \frac{\partial \mathcal{L}(\Theta_k;\boldsymbol{x},\boldsymbol{y})}{\partial \boldsymbol{a}^{(L+1)}(\boldsymbol{x})} = \frac{\partial \mathcal{L}(\Theta_k;\boldsymbol{x},\boldsymbol{y})}{\partial \boldsymbol{h}^{(L+1)}(\boldsymbol{x})} \odot \boldsymbol{g}'\left(\boldsymbol{a}^{(L+1)}(\boldsymbol{x})\right).$$

  2. Compute gradient at different hidden layers: for $l = L, L-1, \cdots, 1$, compute

$$\frac{\partial \mathcal{L}(\Theta_k;\boldsymbol{x},\boldsymbol{y})}{\partial \boldsymbol{h}^{(l)}(\boldsymbol{x})} \leftarrow \left(W_k^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)}(\boldsymbol{x},\boldsymbol{y}),\;\; \boldsymbol{\delta}^{(l)}(\boldsymbol{x},\boldsymbol{y}) \leftarrow \frac{\partial \mathcal{L}(\Theta_k;\boldsymbol{x},\boldsymbol{y})}{\partial \boldsymbol{h}^{(l)}(\boldsymbol{x})} \odot \boldsymbol{f}'\left(\boldsymbol{a}^{(l)}(\boldsymbol{x})\right).$$

- **Update parameters on mini-batch**: for $l = L, L-1, \cdots, 0$,

$$\boldsymbol{b}_{k+1}^{(l+1)}(\boldsymbol{x}) \leftarrow \boldsymbol{b}_k^{(l+1)}(\boldsymbol{x}) - \frac{\eta}{|\mathcal{B}|} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{B}_{k+1}} \boldsymbol{\delta}^{(l+1)}(\boldsymbol{x},\boldsymbol{y}),$$

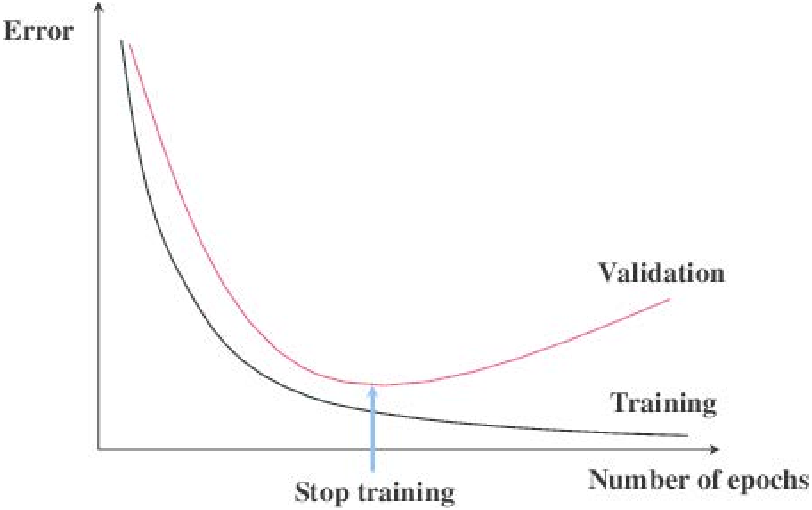
$$W_{k+1}^{(l+1)} \leftarrow W_k^{(l+1)} - \frac{\eta}{|\mathcal{B}|} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{B}_{k+1}} \boldsymbol{\delta}^{(l+1)}(\boldsymbol{x},\boldsymbol{y}) \left(\boldsymbol{h}^{(l)}(\boldsymbol{x})\right)^T.$$

Practical Issues

- Initialisation: randomly initialise weights with small values, e.g., $U[-\frac{1}{\sqrt{|\mathbf{x}|}}, \frac{1}{\sqrt{|\mathbf{x}|}}]$
- Hyper-parameter issues: many hyper-parameters to be set properly before training
  - Architectural/structral: number of hidden layers, number of hidden neurons/layer, other parameters in chosen activation functions
  - learning-related: learning rate(s), trade-off coefficient for regularisation, mini-batch size and so on
- Hyper-parameter tuning: grid search or random search from a range of values
- Model selection and evaluation
  - Main method: held-out validation and K-fold cross validation
  - During learning, use the performance on validation sets to find out optimal hyper-parameter values and decide the early stopping to avoid overfitting

Practical Issues

- Early stopping: effective measure to avoid over-fitting



Reference

If you want to deepen your understanding and learn something beyond this lecture, you can self-study the optional references below.

[Goodfellow et al., 2016] Goodfellow I., Bengio Y., and Courville A. (2016): *Deep Learning*, MIT Press. (Chapter 6 & Sections 11.1-11.5)

[Schmidhuber, 2015] Schmidhuber J. (2015): Deep learning in neural networks: An overview. *Neural Networks*, Vol. 61, pp. 85-117.