Introduction to Algorithms for Data Mining and Machine Learning

Chapter 8: Artificial Neural Networks and Deep Learning

Xin-She Yang

For details, please read the book:

Xin-She Yang, **Introduction to Algorithms for Data Mining and Machine Learning**, Academic Press/Elsevier, (2019).

Learning

Machine learning is a class of sophisticated algorithms, including supervised learning, unsupervised learning, reinforced learning, semi-supervised learning, etc.

Supervised Learning

Supervised learning uses data with known labels and classes. Regression is an example of supervised learning. Classification with support vector machine is also supervised learning.

Unsupervised Learning

Unsupervised learning does not use any labels. Clustering is an example is unsupervised learning.

Neural Networks can be supervised or unsupervised. If training a neural network uses labeled data, it is supervised learning. If no labeled data is used (such as in the auto-encoder), it becomes unsupervised.

Reinforced learning uses some reward objectives to be maximized, whereas semi-supervised learning is the learning of largely using a lot of unlabeled data with a few labeled data. Other new forms of learning will be outlined later (see Trends in Deep Learning in this chapter).

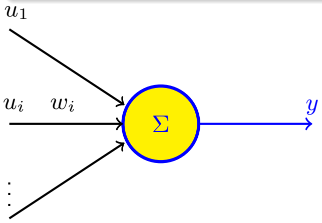
Neuron Model

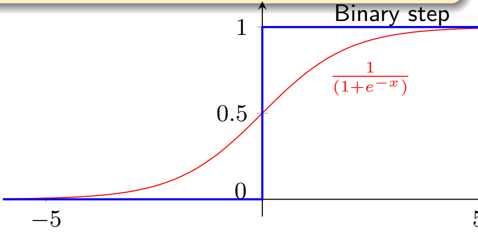
The basic mathematical model of an artificial neuron was first proposed by W. McCulloch and W. Pitts in 1943, now called the McCulloch-Pitts model.

An artificial neuron with n inputs or impulses and an output y will be activated if the signal strength reaches a certain threshold θ. Each input has a corresponding weight w_i. The output of this neuron is given by

$$y = f(x), \quad x = \sum_{i=1}^{n} w_i u_i,$$

where the weighted sum x is the total signal strength, and $f(x) = f(\sum_i u_i)$ is the so-called activation function, often with a threshold parameter θ. For example, the binary activation is $f(x) = 1$ if $x \geq \theta$; otherwise, $f(x) = 0$ if $x < \theta$.





A smooth activation function is the sigmoid function $S(x) = 1/[1 + \exp(-x)]$.

Activation Model

The sigmoid function is a smooth function and its derivative can be calculated by

$$S'(x) = \left(\frac{1}{1+e^{-x}}\right)' = S(x)[1 - S(x)].$$

There are many different activation models with different activation behaviour.

- Rectified linear model or rectified linear unit (ReLU), widely used in deep learning:

  $$f(x) = x^{+1} = \max\{0, x\} = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{otherwise.} \end{cases}$$

  A desired property of ReLU is that its derivative is 1 (constant) when $x > 0$.

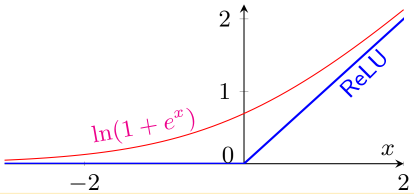
- The ReLU function can be approximated by a smoother function, called SoftPlus,

  $$f(x) = \log(1 + e^x),$$

  whose derivative is $f'(x) = e^x/(1 + e^x) = S(x)$.

- Parametric rectified linear unit (PReLU), also called leaky rectified linear unit (Leaky ReLU), with a parameter α (typically $\alpha = 0.01$):

  $$f(x) = \begin{cases} \alpha x, & \text{if } x < 0, \\ x, & \text{if } x \geq 0. \end{cases}$$



- The exponential linear unit (ELU) is defined by

$$f(x) = \begin{cases} \alpha(e^x - 1), & \text{if } x < 0, \\ x, & \text{if } x \geq 0. \end{cases}$$

- The hyperbolic tangent activation can be written as

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}} = \frac{2}{1 + e^{-2x}} - 1,$$

which approaches to $+1$ as $x \to \infty$ and -1 as $x \to -\infty$. Another related activation function is the arctan function $f(x) = \tan^{-1}(x)$.

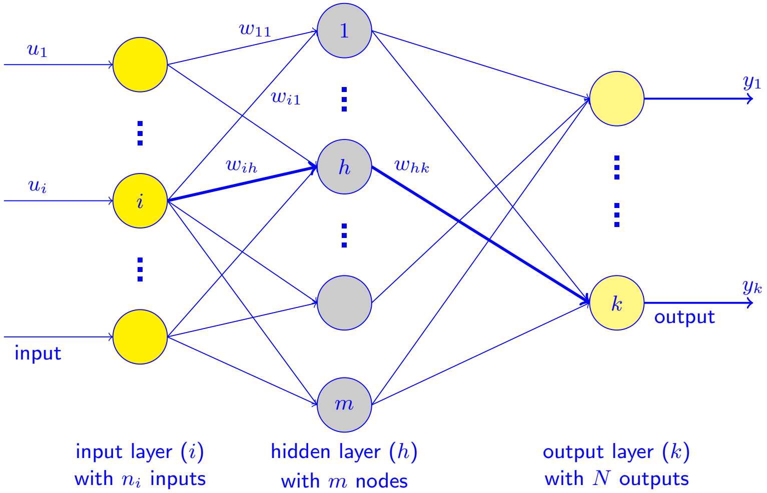
- The so-called softmax activation is often used to convert to probabilities of m classes. The probability for class i is given by

$$P_i = \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}}, \quad i = 1, 2, ..., m,$$

which is essentially the same as the softmax regression.

A single neuron can only perform a simple task – on or off. Complex functions can be designed and performed using a network of interconnecting neurons or perceptrons.

Different network architectures exist, though one of the most widely used is a layered structure, with an input layer, an output layer, and one or more hidden layers. The connection strength between two neurons is represented by its corresponding weight w.



Fitting/training Error

The errors can be defined as the difference between the calculated (or predicated) output v_k and real output y_k for all N output neurons in the least-square sense

$$E = \frac{1}{2}\sum_{k=1}^{N}(v_k - y_k)^2 = \frac{1}{2}||\boldsymbol{v} - \boldsymbol{y}||_2^2,$$

where the output v_k is a function of inputs, activation, and weights. $\boldsymbol{v} = [v_1, ..., v_N]$ is the output vector and $\boldsymbol{y} = [y_1, ..., y_N]$ is the real/desired output vector.

This is an optimization problem that can in principle be solved use any proper optimization techniques to find the weights. However, special techniques such as gradient-based methods tend to be more efficient.

Weight Increments

For any initial random weights, the weight increment for w_{hk} as an example is

$$\Delta w_{hk} = -\eta\frac{\partial E}{\partial w_{hk}} = -\eta\frac{\partial E}{\partial v_k}\frac{\partial v_k}{\partial w_{hk}}. \tag{1}$$

Putting all the weights as a vector, we have

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} - \eta\nabla E(\boldsymbol{w}^{t}),$$

Here, $0 < \eta \leq 1$ is the learning rate. In a special case, we can use $\eta = 1$ for discussions.

BPNN

As the ANN we discussed has a layered structure, we can estimate the error from the output layer and back propagate to estimate the errors in hidden layers and then to the input layer, leading to the well-known back propagation neural network (BPNN).

The inputs of the output layer are the nodes of the final hidden layer. That is

$$S_k = \sum_{h=1}^{m} w_{hk} v_h, (k = 1, 2, ..., N), \quad v_k = f(S_k) = \frac{1}{1 + e^{-S_k}}.$$

Since the derivative of the sigmoid function $f(x)$ can be calculated by multiplication $f'(x) = f(x)[1 - f(x)]$, we have

$$\frac{\partial v_k}{\partial w_{hk}} = \frac{\partial v_k}{\partial S_k} \frac{\partial S_k}{\partial w_{hk}} = v_k(1 - v_k) v_h, \quad \frac{\partial E}{\partial v_k} = (v_k - y_k).$$

Therefore, we have [after substituting the above into Eq. (1)]

$$\Delta w_{hk} = -\eta \delta_k v_h, \quad \delta_k = v_k(1 - v_k)(v_k - y_k),$$

which can be written compactly as an iterative formula

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t + \Delta \boldsymbol{w}^t = \boldsymbol{w}^t - \eta \nabla E(\boldsymbol{w}^t), \quad E = \frac{1}{2} \sum_{k=1}^{N} \Big[v_k(\boldsymbol{w}) - y_k \Big]^2.$$

Once we calculate the errors due to the inputs from the hidden layers, we can calculate (propagate) the errors due to inputs from the previously layer, until the actual inputs u_i.

Loss Function

The loss (or error) function we have discussed so far is mainly the residual errors in L_2-norm. Many other loss functions are used in the literature.

- If $\boldsymbol{y} = (y_1, y_2, ..., y_N)^T$ is a vector of the predicted values and $\bar{\boldsymbol{y}} = (\bar{y}_1, \bar{y}_2, ..., \bar{y}_N)$ is the vector of the true values, the L_2 loss function is usually defined as

$$E = ||\boldsymbol{y} - \bar{\boldsymbol{y}}||_2 = ||\bar{\boldsymbol{y}} - \boldsymbol{y}||_2 = \sum_{i=1}^{N} (\bar{y}_i - y_i)^2,$$

  which is essentially the same as before, except for a convenient factor of $1/2$.

- In stead of the smooth L_2-norm loss, we can use L_1-norm based loss function

$$E_1 = ||\bar{\boldsymbol{y}} - \boldsymbol{y}||_1 = \sum_{i=1}^{N} |\bar{y}_i - y_i|.$$

- The Kullback-Leibler (KL) divergence loss function can be defined by

$$E_{KL} = \frac{1}{N} \sum_{i=1}^{N} \bar{y}_i \log(\bar{y}_i) - \frac{1}{N} \sum_{i=1}^{N} \bar{y}_i \log y_i,$$

  which combines the entropy (the first sum) and the cross entropy (the second term).

- For other applications such the classification using the support vector machines, the loss function can be defined as a hinge loss

$$E_h = \sum_{i=1}^{N} \max\{0, 1 - \bar{y}_i \cdot y_i\},$$

  or with a factor 1/2

$$E_h = \sum_{i=1}^{N} \max\{0, \frac{1}{2} - \bar{y}_i \cdot y_i\}.$$

- Another related hinge loss function is the so-called squared hinge loss function

$$E_{h^2} = \sum_{i=1}^{N} \Big(\max\{0, 1 - \bar{y}_i \cdot y_i\} \Big)^2.$$

There are over a dozen other loss functions, including the Tanimoto loss, Chebyshev loss, and Cauchy-Schwarz divergence. Each can serve a different purpose with (slightly) different emphasis.

Since the minimization of $||E||$ is the same as minimization of $\frac{1}{2}||E||$ or $||E||/N$ where N is a positive integer, different formulations of loss functions may include some arbitrary factors for convenience.

Optimizers for ANN

Neural networks, specially multilayered networks, require the calculation of the gradient vectors iteratively. This can become very expensive if the numbers of weights and outputs are huge, which is true for deep learning. Thus, care should be taken in choosing the right algorithms.

- Stochastic Gradient Descent (SGD): A single data example (or a few subset) is used to estimate the true gradient ∇E, which gives

$$\boldsymbol{w}^{t+1} = \boldsymbol{w} - \eta \nabla E_i,$$

  which can reduce the number of calculations by a factor of n (typically $n \gg 1$).
- More appropriate optimizers are moment-based optimizers such as the Adam and RMSprop introduced in Chapter 3.

There are a vast variety of software packages (either free or commercial), and they have implemented almost all the commonly used optimizers and some popular network structures for ANNs, such as BPNN.

For example, Google's TensorFlow is a very powerful engine for deep learning. TensorFlow can be combined with front-end interfaces such as `Keras` and `TFLearn`, which makes their use much easier.

Network Structure

There are many other different structures for arranging neurons, and they can have different advantages for different applications.

- Feed-forward networks: The simplicity of the feed-forward neural networks allows the BPNN algorithm to work well. If the input layer and output layer have identical structure so as to train outputs to match the inputs, it is an autoencoder (AE).
- RBF networks: When the activation is carried out using a neighbor (in the similar manner to the k-nearest neighbor method), the activation function depends on the distance where radial basis functions (RBF) are used for activating each neuron, then the feed-forward network becomes an RBF network.
- Recurrent neural network (RNN): This network architecture, which uses a directed connection between every pair of neurons, and the connections are not just from the previous layer to the current layer, but also within the current layer itself. The connection strengths are modelled as time-varying, real-valued weights.
- Hopfield network: Each neuron is fully connected symmetrically with every other neuron in the network. It has a guarantee in terms of convergence.
- Self-organized map: A self-organized map (SOM), or a Kohonen network maps inputs onto outputs by activating most appropriate neurons and their neighbour neurons that can closely fit the inputs.

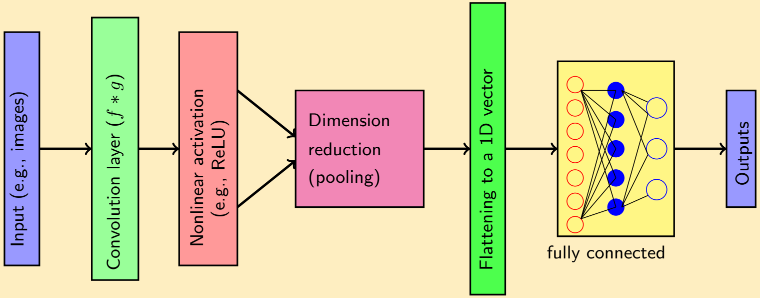
- Boltzmann machines: Boltzmann machines (BM) has certain similarities to Kohonen networks. A special class of BMs is the restricted Boltzmann machine (RBM) with feed-forward structures for connections.
- Deep belief networks: A deep belief network (DBN) is a deep multi-layer neural network that has many levels of nonlinearities with a deep network architecture using restricted Boltzmann machines or variational autoencoders. The top-level prior is a restricted Boltzmann machine between layer $j-1$ and layer j, and some greedy layer-wise training algorithm is often used to train one layer at a time. Then, some fine-tuning of parameters of all layers is carried out.
- Modular neural networks: A modular neural network (MNN) uses a combination of a set of multiple neural networks, and each network is considered as a module. Each module can only carry out a specific task and connection to other modules is designed on the system level.
- Spiking neural networks: The spiking neural network (SNN) aims to provide a more realistic model for neural networks that are based on the synaptic structures and states. It considers the time-varying information where signal impulses (a spiking train) rise for a short period and then gradually decay. The activation depends on the spiking impulse time interval, strength and frequency.

Others network architectures include the neural Turing machine, capsule networks, generative adversarial nets (GAN), and others such as extreme learning machines. Interested readers can refer to more advanced literature and textbooks.

Convolutional Neural Network

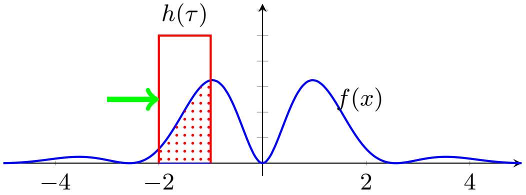
Deep learning (DL) involves neural networks with multiple hidden layers, and it usually has a modular structure of combing multiple convolutional neural networks (CNN) in a series of operations. In contrast with a fully connected ANN, a CNN has sparse connections and computation is done by convolution.

Deep Nets


Input (e.g., images)
Convolution layer ($f * g$)
Nonlinear activation (e.g., ReLU)
Dimension reduction (pooling)
Flattening to a 1D vector
fully connected
Outputs

A deep net consists of many layers of CNNs, and each CNN first converts inputs (e.g., images) into a set of convoluted features by going through a filter or kernel of a fixed size. The convoluted features go through activation and are sub-sampled by pooling. Then, pooled features are flattened into a 1D feature vector to be fed into a densely connected NN for classification.

Convolution



The basic idea of convolution between a function $f(x)$ and a kernel h is defined as

$$(f * h)(x) = \int_{-\infty}^{+\infty} f(x-\tau)h(\tau)d\tau = \int_{-\infty}^{+\infty} f(\tau)h(x-\tau)d\tau,$$

which is the integral of the overlapped area between two functions. This is equivalent to sliding one function over the other function.

Convolution has been used extensively in signal and image processing. For 2D inputs such as images $f(x,y)$, the 2D convolution with a kernel $h(x,y)$ can be written as

$$(f * h)(x,y) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} h(p,q)f(x-p,y-q)dpdq = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(p,q)h(x-p,y-q)dpdq,$$

which usually becomes a double sum over a region of an image.

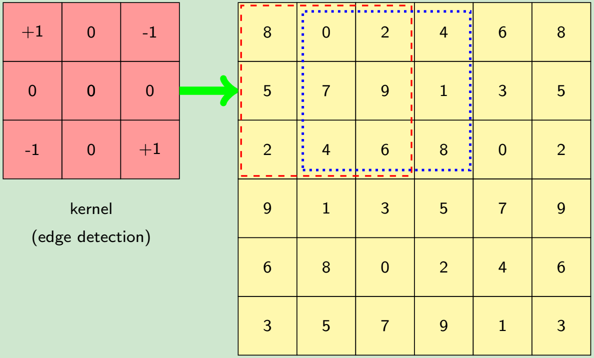
Example: 3×3 kernel

If we apply a simple edge detection 3×3 kernel

$$h = \begin{pmatrix} +1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & +1 \end{pmatrix},$$

to a 6×6 image, we have (top left corner, dashed)

$$W(1,1) = (f * h) = +1 \times 8 + 0 \times 0 + (-1) \times 2 + 0 \times 5 + 0 \times 7 + 0 \times 9 + (-1) \times 2 + 0 \times 4 + 1 \times 6 = 10.$$



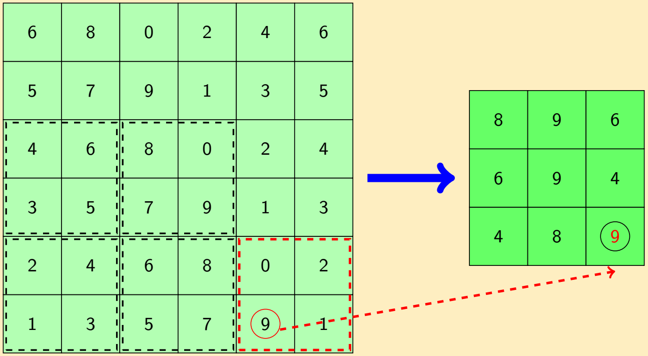
kernel
(edge detection)

The rest of W entries can be calculated in the similar manner.

Pooling

Pooling is a dimensional reduction technique by focusing a region of size $k \times k$ as a pooling filter. Only one value is calculated from a region of fixed size, either the maximum value or their average. If the maximum value is used, it is called max pooling.

For example, in the simple pooling shown in the figure, a 2×2 pooling filter is applied to every non-overlapping 2×2 subregions, and the bottom right corner after pooling becomes 9 from $\begin{vmatrix} 0 & \vdots & 2 \\ \cdots & \cdots & \cdots \\ 9 & \vdots & 1 \end{vmatrix}$.



Flattening

The outputs from convolution and pooling operations are either 2D arrays for grayscale images or 3D arrays for colour images. They are converted (flattened) into a 1D feature vector to be fed into the fully connected NN. This is done by stacking each row or column in a sequential order, and then at different depths.

For example, the 2×2 max pooling shown in the previous figure, the 2D array (on the right) can be flattened to

$$\begin{pmatrix} 8 & 9 & 6 \\ 6 & 9 & 4 \\ 4 & 8 & 9 \end{pmatrix} \Longrightarrow \begin{pmatrix} 8 & 9 & 6 & 6 & 9 & 4 & 4 & 8 & 9 \end{pmatrix}^T.$$

Fully Connected NN

The flattened feature vector (from the CNN) is now fed into a full connected NN for classification.

To make the network more efficient (and sparse), some dropout operations are carried out. The main idea of dropping out is that neurons in an ANN can be ignored and removed with a probability p, which can reduce the number of parameters and thus reduce the complexity of the network. This can also potentially prevent overfitting. Typically, the probability $p = 0.25$ is often used.

There are quite a few good packages for deep learning, including AlexNet/ImageNet, Google's TensorFlow and others.

Boltzmann Machines

Another very useful tool for deep learning applications is the restricted Boltzmann machine (RBM), which is a two-layer (or two-group) Boltzman machine with m visible units $v_i (i = 1, 2, ..., m)$ and n hidden units $h_j (j = 1, 2, ..., n)$ where both v_i and h_j are binary states.

The visible unit i has a bias α_i, and the hidden unit j has a bias β_j, while the weight connecting them is denoted by w_{ij}. The restriction is that their neuron units connecting visible and hidden units form a biparte graph, while no connection with the same group (visible or hidden) is allowed.

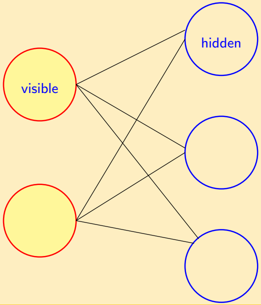
RBM

A restricted Boltzmann machine can be represented by a pair $(\boldsymbol{v}, \boldsymbol{h})$

$$\boldsymbol{v} = (v_1, v_2, ..., v_m)^T, \quad \boldsymbol{h} = (h_1, h_2, ..., h_n)^T.$$

The system energy can be calculated by

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{i \in \text{visible}} \alpha_i v_i - \sum_{j \in \text{hidden}} \beta_j h_j - \sum_{i,j} w_{ij} v_i h_j.$$



The probability of a network associated with every possible pair of $\boldsymbol{v}$ and $\boldsymbol{h}$ is assumed to obey the Boltzmann distribution

$$p(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z} e^{-E(\boldsymbol{v}, \boldsymbol{h})},$$

where Z is a normalization constant, also called partition function, which is essentially the summation over all possible configurations. That is $Z = \sum_{\boldsymbol{v}, \boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})}$.

The training of an RBM using data (e.g., images) is to adjust the weights and bias values so that a training image can maximize its associated network probability. That is

$$\text{Maximize} \quad p(\boldsymbol{v}) = \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})}.$$

The maximization of the joint probability $p(\boldsymbol{v})$ is equivalent to the maximization of the expected log probability [i.e., $\log p(\boldsymbol{v})$]. Since

$$\frac{\partial \log p(\boldsymbol{v})}{\partial w_{ij}} = < v_i h_j >_{\text{data}} - < v_i h_j >_{\text{RBMmodel}},$$

we can calculate the adjustments in weights by using the stochastic gradient method

$$\Delta w_{ij} = \eta\Big(< v_i h_j >_{\text{data}} - < v_i h_j >_{\text{RBMmodel}} \Big),$$

where $< v_i h_j >$ means the expectation over the associated distributions.

There are many good software packages for artificial neural networks and RBMs, and there are dozens of good books fully dedicated to theory and implementations.

Trends in Deep Learning

Deep learning is a very active research area and new progress is being made every year or even every week. Here, we only briefly outline some recent developments and trends.

- Self-taught learning is a two-stage learning approach where learning is carried out first on the unlabelled data to learn a representation, and then this learned representation is applied to labelled data for classification tasks. The data can be in different classes and from different distributions.
- Transfer learning transfers knowledge from one supervised learning task to another, which requires additional labelled data from a different (but related) task. The requirement of such extra labelled may be expensive.
- One-shot learning is an efficient learning technique where learning is carried out with only one example or a handful examples. Instead of learning from scratch, training is improved from previously learned categories.
- Capsule networks are a new network architecture with a hierarchical spatial relationship where a capsule is a group of neurons whose activity vector represents certain entity parameters. The entity can be an object or part of an image. Active capsules at one-level make predictions by using transformation matrices, and a higher-level capsule becomes active if multiple predictions agree.

- Generative adversarial networks (GAN) use one network to generate candidates such as realistic-looking images and other network to learn and detect if the generated candidates are real or not. The generated samples can be sufficient realistic to the real things. Bayesian GANs, as a hybrid model, can implicitly learn a rich distribution over data such as images and audios, and such data may be difficult to model with an explicit likelihood probability.
- Hybrid learning can combine different approaches to gain certain advantages. For example, the combination of recurrent neural networks (RNN) with CNN, long short-term memory (LSTM) networks, the RNN with long short-term memory, gated recurrent neural networks, deep reinforcement learning, and others.

Hyperparameter Tuning

In many algorithms and techniques of data mining and machine learning, there are some hyperparameters, including the k in k-means, λ in regularization, the learning rate η and the number of layers in deep nets. These parameters need to be tuned properly; however, there are no agreed effective methods or guidelines.

Currently, tuning of such hyperparameters is largely empirical. Thus, care should be taken when setting such parameters.

In most software packages, there are some recommended (default) values, and simulations should start with these values and fine-tuning can be carried out if needed.

Thank you :)

References

- Xin-She Yang, **Introduction to Algorithms for Data Mining and Machine Learning**, Academic Press/Elsevier, (2019).
- Xin-She Yang, **Optimization Techniques and Applications with Examples**, John Wiley & Sons, (2018).

Notes on Software

There are many software packages on ANN, CNN and deep learning.

- Matlab has the `nntool`, `alexnet` and `Googlenet`.
- R has toolboxes such as `elasticnet`, deep learning `deepnet` and Restricted Boltzmann machine `RcppDL` and interface to TensorFlow.
- Python has machine learning packages such as `scikit-learn`, including support vector machines and neural networks.
- Google's TensorFlow is a very powerful engine for deep learning. There are many frontend tools to interact with TensorFlow. For example, both `Keras` are `TFLearn` are among the most widely used and they are easy to start with many examples on the Internet.
- Other tools include `Theano`, `Torch`, `Caffe` and others. In addition, Microsoft, IBM, Amazon and many other companies have deep learning tools. The sources are diverse and the literature is vast.