Neural network

Authored by **SONG DAIWEI** 44161588-3 Written by Markdown and Latex.

You can see its source code in GitHub of SONG DAIWEI

1. Biological Neural Network

The structure of neural is composed of cell body, dendrite, axon and synapse.

- cell body

  The cell body is the main body of the neuron, which is composed of 3 parts, the cell nucleus, the cytoplasm and the cell membrane. The outer part of the cell body is the cell membrane, which separates the inside from the outside of the membrane. Because the cell membrane has different permeability to the different ions in the cell, this makes the difference of the ion concentration. This potential difference is called membrane potential.

- dendrite

  The dendrite is a nerve fiber that extends out of the cell body, which is in charge of receiving input signals from other neurons, equivalent to the input side of the cell body.

- axon

  The longest of the nerve fibers is called an axon, which is long and thin. The end of a lot of fine branches are called nerve endings, and each nerve endings can be heard in all directions. It is equivalent to the output of the cell body.

- synapse

  A neuron communicates with the cell body or dendrites of another neuron through its nerve endings, which is equivalent to the input / output interface (I/O) of the neuron. It is called the synapse.

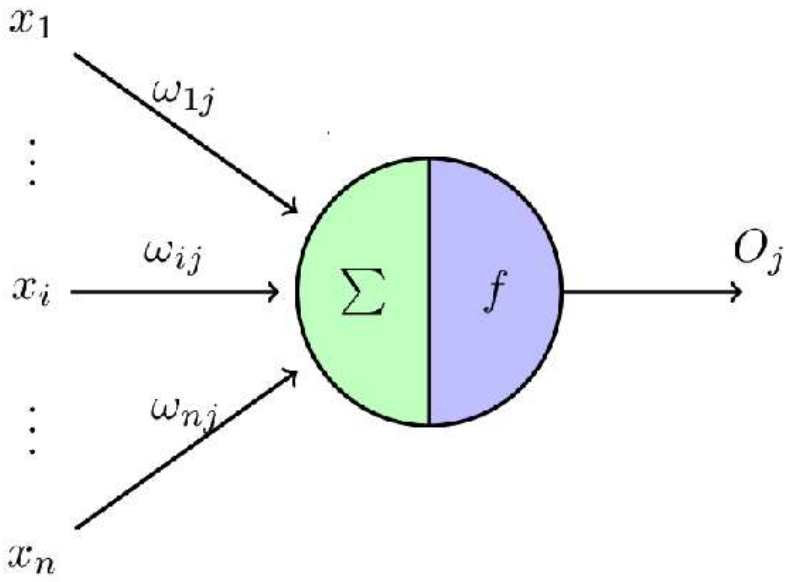
And the synapse leads to the changes of membrane potential, and the changes of potential can be cumulative. Thousands of neuron can communicate with a single neuron through the synapses, and the input positions are different, so the weights of the impact of neuron are also different.

Of course, the synapses can be divided into two types of excitatory and inhibitory synapses. Excitatory synapses may cause the next nerve cell excitatory, besides the other kind of synapses leads to inhibition.

Pulse propagation is one way and does not allow reverse propagation. In addition, synaptic transmission requires a certain delay.

In summary, we can conclude the features of the model of neuron network.

1. Each neuron is a information processing unit with multi inputs and a single output.
2. The pulses of neuron are composed of excitatory and inhibitory pulse.
3. Neurons have the characteristics of spatial integration and threshold.
4. There is a fixed delay between the input and the output of neurons, which depends on the delay of synaptic.



2. McCulloch-Pitts Neuron Model

According to the model of biological neuron, we establish the McCulloch-Pitts Neuron Model. In order to make the model simpler, and facilitate the formal expression, we regard complex factors as constants such as the time integration and the refractory period.

We can understand the model by analogy with the features of biological neuron.

Biological Neural and McCulloch-Pitts Neuron Model

biological neuron	neuron	input signal	weight	output signal	summary	membrane potential	threshold
M-P Neuron Model	j	χ_i	w_{ij}	o_j	Σ	$\sum_{i=1}^{n} w_{ij}\chi_i(t)$	T_j

For some one neuron j (some special c), it can accept many input signals in terms of χ_i, and because of the different properties of synapses, we add weight to its effect in terms of w_{ij}, whose signs simulate the excitatory and inhibitory signals. Also, because of the accumulation, we use the sum of all signals, so the value of membrane potential is as follow:

$$net'_j(t) = \sum_{i=1}^{n} w_{ij}\chi_i(t)$$

$$net'_j = W_j^T X$$

The activation of neurons depends on a certain threshold level, which means that only the sum of inputs goes beyond the threshold, T_j, the neuron will be active to generate the pulse.

$$o_j(t+1) = f\{[\sum_{i=1}^{n} w_{ij}\chi_i(t)] - T_j\}$$

$$o_j = f(net_j) = f(W_j^T X)$$

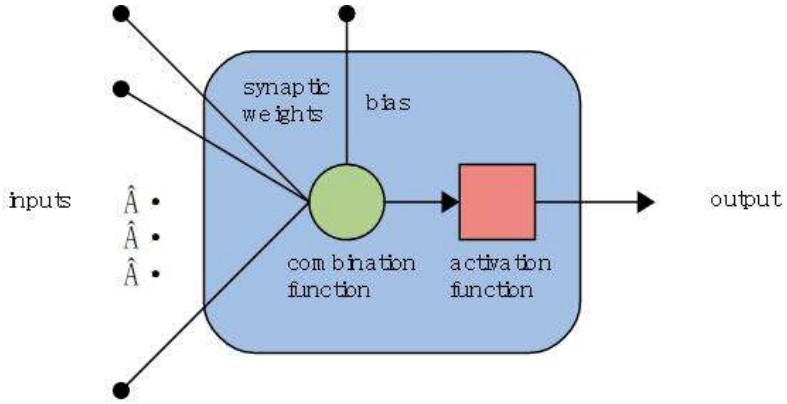
In summary, we can conclude the six features of M-P Model:

1. Each neuron is a information processing unit with multi inputs and a single output.
2. The pulses of neuron are composed of excitatory and inhibitory pulse.
3. Neurons have the characteristics of spatial integration and threshold.
4. There is a fixed delay between the input and the output of neurons, which depends on the delay of synaptic.
5. Ignore the time integration and the refractory period. $\chi_0 = -1$, $w_{0j} = T_j$, so $-T_j = \chi_0 w_{0j}$
6. Neurons themselves are time invariant, and their synaptic delay and synaptic strength are constant.

The McCulloch-Pitts model of a neuron is simple yet has substantial computing potential. It also has a precise mathematical definition. However, this model is so simplistic that it only generates a binary output and also the weight and threshold values are fixed. The neural computing algorithm has diverse features for various applications. Thus, we need to obtain the neural model with more flexible computational features.

3. One-neuron Model: Perceptron

A neuron model is the basic information processing unit in a neural network. They are inspired by the nervous cells, and somehow mimic their behavior. The perceptron is the characteristic neuron model in the multilayer perceptron. Following current practice, the term perceptron is here applied in a more general way than by Rosenblatt, and covers the types of units that were later derived from the original perceptron. The following figure is a graphical representation of a perceptron.



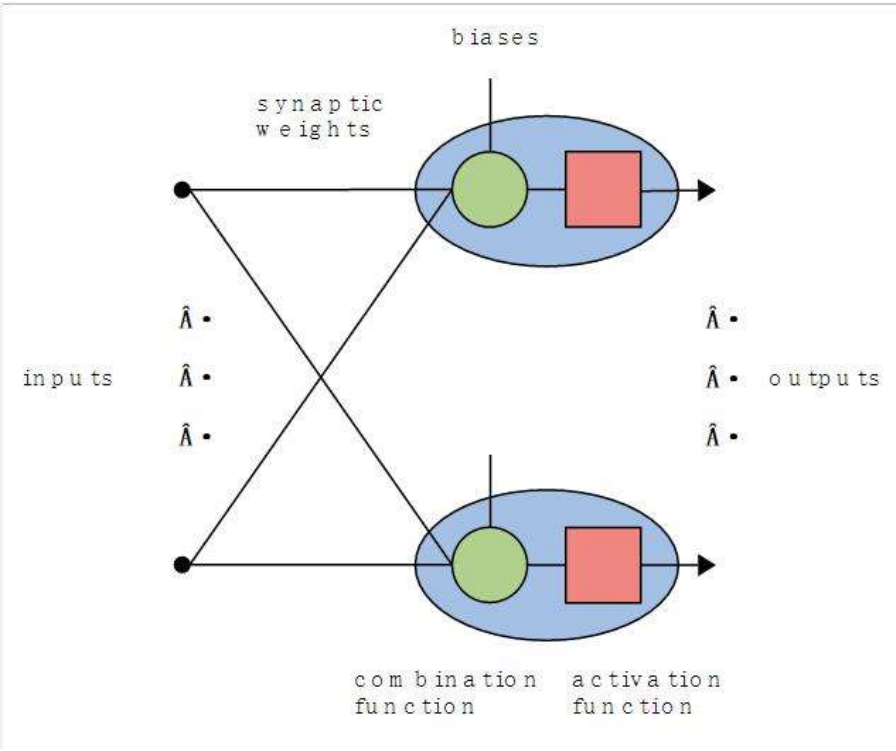
Here we identify three basic elements, which transform a vector of inputs into a single output:

- A set of parameters consisting of a bias and a vector of synaptic weights.
- A combination function.

- An activation function or transfer function.

4. Perceptron layer

Most neural networks, even biological neural networks, exhibit a layered structure. In this work layers are the basis to determine the architecture of a neural network. A layer of perceptron is composed by a set of perceptron sharing the same inputs. The architecture of a layer is characterized by the number of inputs and the number of perceptron. The next figure shows a general layer of perceptron.



Here we identify three basic elements, which transform a vector of inputs into a vector of outputs:

- A set of layer parameters.
- A layer combination function.
- A layer activation function.

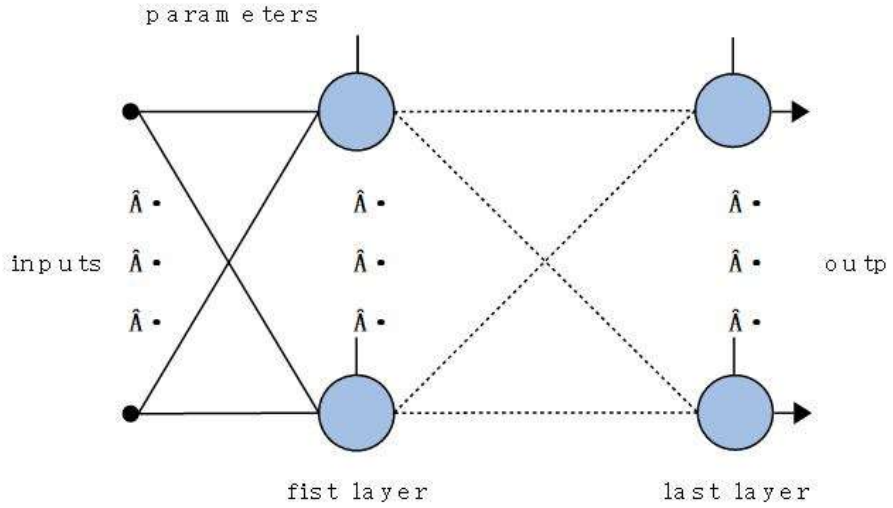
5. Multilayer Neural Network: Multilayer Perceptron

Layers of perceptron can be composed to form a multilayer perceptron. Most neural networks, even biological ones, exhibit a layered structure. Here layers and forward propagation are the basis to determine the architecture of a multilayer perceptron. This neural network represents an explicit function which can be used for a variety of purposes.

The architecture of a multilayer perceptron refers to the number of neurons, their arrangement and connectivity. Any architecture can be symbolized as a directed and labelled graph, where nodes represent neurons and edges represent connectivity among neurons. An edge label represents the parameter of the neuron for which the flow goes in. Thus, a neural network typically consists on a set of sensorial nodes which constitute the input layer, one or more hidden layers of neurons and a set of neurons which constitute the output layer.

There are two main categories of network architectures: acyclic or feed-forward networks and cyclic or recurrent networks. A feed-forward network represents a function of its current input; on the contrary, a recurrent neural network feeds outputs back into its own inputs. As it was said above, the characteristic neuron model of the multilayer perceptron is the perceptron. On the other hand, the multilayer perceptron has a feed-forward network architecture.

Hence, neurons in a feed-forward neural network are grouped into a sequence of layers of neurons, so that neurons in any layer are connected only to neurons in the next layer. The input layer consists of external inputs and is not a layer of neurons; the hidden layers contain neurons; and the output layer is also composed of output neurons. The following figure shows the network architecture of a multilayer perceptron.



A multilayer perceptron is characterized by:

- A network architecture.
- A set of parameters.
- The layers' activation functions.

Communication proceeds layer by layer from the input layer via the hidden layers up to the output layer. The states of the output neurons represent the result of the computation.

In this way, in a feed-forward neural network, the output of each neuron is a function of the inputs. Thus, given an input to such a neural network, the activations of all neurons in the output layer can be computed in a deterministic pass.

6. Gradient Decent method

Gradient descent is a first-order iterative optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point. If instead one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent.

Gradient descent is based on the observation that if the multi-variable function $F(x)$ is defined and differentiable in a neighborhood of a point a , then $F(x)$ decreases fastest if one goes from a in the direction of the negative gradient of F at a, $-\ nablaF(a)$. It follows that, if

$b = a - \gamma\ nablaF(a)$

for γ small enough, then $F(a) \geq F(b)$. In other words, the term $\gamma\ nablaF(a)$ is subtracted from a because we want to move against the gradient, namely down toward the minimum. With this observation in mind, one starts with a guess x_0 for a local minimum of F, and considers the sequence $x_0, x_1, x_2, \cdots$ such that

$x_{n+1} - \gamma_n \nabla F(x_n), n \geq 0.$

We have

$F(x_0) \geq F(x_1) \geq F(x_2) \geq \cdots,$

so hopefully the sequence (x_n) converges to the desired local minimum. Note that the value of the step size γ is allowed to change at every iteration. With certain assumptions on the function F (for example, F convex and ∇F Lipschitz) and particular choices of γ (e.g., chosen via a line search that satisfies the Wolfe conditions), convergence to a local minimum can be guaranteed. When the function F is convex, all local minima are also global minima, so in this case gradient descent can converge to the global solution.

This process is illustrated in the picture to the right. Here F is assumed to be defined on the plane, and that its graph has a bowl shape. The blue curves are the contour lines, that is, the regions on which the value of F is constant. A red arrow originating at a point shows the direction of the negative gradient at that point. Note that the (negative) gradient at a point is orthogonal to the contour line going through that point. We see that gradient descent leads us to the bottom of the bowl, that is, to the point where the value of the function F is minimal.

Gradient descent is an iterative minimization method. The gradient of the error function always shows in the direction of the steepest ascent of the error function. Thus, we can start with a random weight vector and subsequently follow the negative gradient (using a learning rate η).

$w_{t+1} = w_t + \Delta w_t, \Delta w_t = \eta \frac{\partial E(w)}{\partial w}^T |_{w_t}$

Impulse term:

$w_t = \alpha \Delta w_{t-1} - (1-\alpha)\eta \frac{\partial E(w)}{\partial w}^T|_{w_t}$

for $1 \leq \alpha \leq 1$.

Adaptive learning rate: Increase η if error has decreased, decrease η otherwise.

6.1. Limitations

For some of the above examples, gradient descent is relatively slow close to the minimum: technically, its asymptotic rate of convergence is inferior to many other methods. For poorly conditioned convex problems, gradient descent increasingly 'zigzags' as the gradients point nearly orthogonally to the shortest direction to a minimum point. For more details, see the comments below.

For non-differentiable functions, gradient methods are ill-defined. For locally Lipschitz problems and especially for convex minimization problems, bundle methods of descent are well-defined. Non-descent methods, like subgradient projection methods, may also be used. These methods are typically slower than gradient descent. Another alternative for non-differentiable functions is to "smooth" the function, or bound the function by a smooth function. In this approach, the smooth problem is solved in the hope that the answer is close to the answer for the non-smooth problem (occasionally, this can be made rigorous).

How about gradient decent method in neuron network? Consider a two-layer neural network with the following structure (blackboard):

Hidden Layer:

$a_j^{(1)} = \sum_i w_{ji}^{(1)} x_i, a^{(1)} = [a_1^{(1)}, a_2^{(1)}, \cdots, a_m^{(1)}]^T = W^{(1)} x,$

with $W_{ji}^{(1)} = w_{ji}^{(1)}$ being the weight matrix of the hidden layer, the jth row contains all weights of neuron j.

$z_j = h_{(1)}(a_j^{(1)}), z = [h_{(1)}a_1^{(1)}, h_{(1)}a_2^{(1)}, \cdots, h_{(1)}a_m^{(1)}]^T, \frac{\partial z}{\partial a^{(1)}} = diag([h'_{(1)}(a_j^{(1)})]) = H_{(1)}$

Output Layer:

$a_k^{(2)} = \sum_j w_{kj}^{(2)} z, a^{(2)} = [a_1^{(2)}, a_2^{(2)}, \cdots, a_n^{(2)}]^T = W^{(2)} z$

$o_k = h_{(2)}(a_k^{(2)}), \frac{\partial o_k}{\partial a_k^{(2)}} = h'_{(2)}(a_k^{(2)}), o = [h_{(2)}a_1^{(2)}, h_{(2)}a_2^{(2)}, \cdots, h_{(2)}a_n^{(2)}]^T, \frac{\partial o}{\partial a^{(2)}} = diag([h'_{(2)}(a_k^{(2)})]) = H_{(2)}$

The network has d inputs, m hidden neurons and n output neurons. The transfer function of the network is therefore given by

$o_k = h_{(2)}(\sum_{j=1}^m w_{kj}^{(2)} h_{(1)}(\sum_{i=1}^d w_{(ji)}^{(1)} x_i))$

or in matrix form

$o = h_{(2)}(W^{(2)} h_{(1)}(W^{(1)} x))$

Calculating the outputs o as a function of the inputs x is also denoted as forward sweep in the backpropagation algorithm.

7. BP Algorithm

We can now formulate the complete backpropagation algorithm and prove by induction that it works in arbitrary feed-forward networks with differentiable activation functions at the nodes. We assume that we are dealing with a network with a single input and a single output unit.

Consider a network with a single real input x and network function F. The derivative F'(x) is computed in two phases:

Feed-forward:	the input x is fed into the network. The primitive functions at the nodes and their derivatives are evaluated at each node. The derivatives are stored.
Backpropagation:	the constant 1 is fed into the output unit and the network is run backwards. Incoming information to a node is added and the result is multiplied by the value stored in the left part of the unit. The result is transmitted to the left of the unit. The result collected at the input unit is the derivative of the network function with respect to x.

Since we want to minimize the error function E, which depends on the network weights, we have to deal with all weights in the network one at a time. The feed-forward step is computed in the usual way, but now we also store the output of each unit in its right side. We perform the backpropagation step in the extended network that computes the error function and we then fix our attention on one of the weights, say w_{ij} whose associated edge points from the i-th to the j-th node in the network. This weight can be treated as an input channel into the subnetwork made of all paths starting at w_{ij} and ending in the single output unit of the network. The information fed into the subnetwork in the feed-forward step was $o_i w_{ij}$, where o_i is the stored output of unit i. The backpropagation step computes the gradient of E with respect to this input, i.e., $\partial E / \partial o_i w_{ij}$. Since in the backpropagation step o_i is treated as a constant, we finally have

$\frac{E}{w_{ij}} = o_i \frac{\partial E}{\partial o_i w_{ij}}$

Summarizing, the backpropagation step is performed in the usual way. All subnetworks defined by each weight of the network can be handled simultaneously, but we now store additionally at each node i:

- The output o_i of the node in the feed-forward step.
- The cumulative result of the backward computation in the backpropagation step up to this node. We call this quantity the back propagated error.

If we denote the back propagated error at the j-th node by δ_j , we can then express the partial derivative of E with respect to w_{ij} as:

$\frac{E}{w_{ij}} = o_i \delta_j$

Once all partial derivatives have been computed, we can perform gradient descent by adding to each weight w_{ij} the increment

$\Delta w_{ij} = -\gamma o_i \delta_j$

This correction step is needed to transform the backpropagation algorithm into a learning method for neural networks.

This graphical proof of the backpropagation algorithm applies to arbitrary feed-forward topologies. The graphical approach also immediately suggests hardware implementation techniques for backpropagation.

8. Deep Neural Network

A deep neural network (DNN) is an artificial neural network (ANN) with multiple hidden layers of units between the input and output layers. Similar to shallow ANNs, DNNs can model complex non-linear relationships. DNN architectures, e.g., for object detection and parsing, generate compositional models where the object is expressed as a layered composition of image primitives. The extra layers enable composition of features from lower layers, giving the potential of modeling complex data with fewer units than a similarly performing shallow network.

DNNs are typically designed as feedforward networks, but research has very successfully applied recurrent neural networks, especially LSTM, for applications such as language modeling. Convolutional deep neural networks (CNNs) are used in computer vision where their success is well-documented. CNNs also have been applied to acoustic modeling for automatic speech recognition (ASR), where they have shown success over previous models. For simplicity, a look at training DNNs is given here.

8.1. Backpropagation

A DNN can be discriminatively trained with the standard backpropagation algorithm. According to various sources, basics of continuous backpropagation were derived in the context of control theory by Henry J. Kelley in 1960 and by Arthur E. Bryson in 1961, using principles of dynamic programming. In 1962, Stuart Dreyfus published a simpler derivation based only on the chain rule. Vapnik cites reference in his book on Support Vector Machines. Arthur E. Bryson and Yu-Chi Ho described it as a multi-stage dynamic system optimization method in 1969. In 1970, Seppo Linnainmaa finally published the general method for automatic differentiation (AD) of discrete connected networks of nested differentiable functions. This corresponds to the modern version of backpropagation which is efficient even when the networks are sparse. In 1973, Stuart Dreyfus used backpropagation to adapt parameters of controllers in proportion to error gradients. In 1974, Paul Werbos mentioned the possibility of applying this principle to artificial neural networks, and in 1982, he applied Linnainmaa's AD method to neural networks in the way that is widely used today. In 1986, David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams showed through computer experiments that this method can generate useful internal representations of incoming data in hidden layers of neural networks. In 1993, Eric A. Wan was the first[5] to win an international pattern recognition contest through backpropagation.

The weight updates of backpropagation can be done via stochastic gradient descent using the following equation:

$w_{ij}(t+1) = w_{ij}(t) + \eta \frac{\partial C}{\partial w_{ij}} + \xi(t)$

Here, η is the learning rate, C is the cost function and $\xi(t)$ a stochastic term. The choice of the cost function depends on factors such as the learning type (supervised, unsupervised, reinforcement, etc.) and the activation function. For example, when performing supervised learning on a multiclass classification problem, common choices for the activation function and cost function are the softmax function and cross entropy function, respectively. The softmax function is defined as $p_j = \frac{\exp(x_j)}{\sum_k \exp(x_k)}$ where p_j represents the class probability (output of the unit j) and x_j and x_k represent the total input to units j and k of the same level respectively. Cross entropy is defined as $C = -\sum_j d_j \log(p_j)$ where d_j represents the target probability for output unit j and p_j is the probability output for j after applying the activation function. These can be used to output object bounding boxes in the form of a binary mask. They are also used for multi-scale regression to increase localization precision. DNN-based regression can learn features that capture geometric information in addition to being a good classifier. They remove the limitation of designing a model which will capture parts and their relations explicitly. This

helps to learn a wide variety of objects. The model consists of multiple layers, each of which has a rectified linear unit for non-linear transformation. Some layers are convolutional, while others are fully connected. Every convolutional layer has an additional max pooling. The network is trained to minimize L2 error for predicting the mask ranging over the entire training set containing bounding boxes represented as masks.

9. Problems with deep neural networks

As with ANNs, many issues can arise with DNNs if they are naively trained. Two common issues are overfitting and computation time.

DNNs are prone to overfitting because of the added layers of abstraction, which allow them to model rare dependencies in the training data. Regularization methods such as Ivakhnenko's unit pruning or weight decay (ℓ_2-regularization) or sparsity (ℓ_1-regularization) can be applied during training to help combat overfitting. A more recent regularization method applied to DNNs is dropout regularization. In dropout, some number of units are randomly omitted from the hidden layers during training. This helps to break the rare dependencies that can occur in the training data.

The dominant method for training these structures has been error-correction training (such as backpropagation with gradient descent) due to its ease of implementation and its tendency to converge to better local optima than other training methods. However, these methods can be computationally expensive, especially for DNNs. There are many training parameters to be considered with a DNN, such as the size (number of layers and number of units per layer), the learning rate and initial weights. Sweeping through the parameter space for optimal parameters may not be feasible due to the cost in time and computational resources. Various 'tricks' such as using mini-batching (computing the gradient on several training examples at once rather than individual examples) have been shown to speed up computation. The large processing throughput of GPUs has produced significant speedups in training, due to the matrix and vector computations required being well suited for GPUs. Radical alternatives to backprop such as Extreme Learning Machines, "No-prop" networks, training without backtracking, "weightless" networks, and non-connectionist neural networks are gaining attention.

10. Reference

https://www.spsc.tugraz.at/sites/default/files/lecturenotes_3.pdf

https://www.spsc.tugraz.at/sites/default/files/lecturenotes_3.pdf

https://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf

https://en.wikipedia.org/wiki/Gradient_descent

https://en.wikipedia.org/wiki/Deep_learning