Simple Neural Network

Before you walk through this post, you should have some knowledge of gradient descent and Perceptron. It's highly recommended to check out these two posts Gradient Descent Example Code[1] and Single-layer Neural Networks (Perceptrons)[2] before go through the following content.

As a beginner of Machine Learning, I recommend you to start with some tiny examples rather than some abstract concepts. Since you have learned some basics of Neural Network, read some influential papers of this filed. You will be amazed by the charm of Neural Network.

1 What is Neural Network?

1.1 Neural Network

Who care what is a neural network? Someone said neural network[3] is a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs. You can hardly understand what it means until you understand what is neural network. I would never ask to make sense of such obscure description. I promise you that after reading through this post you will have a vivid picture of neural network.

1.2 Perceptron and Neural Network

We suppose you have know perceptron. Ability of single perceptron is limited, whereas combination of perceptrons will be very powerful. Let me show you some examples:

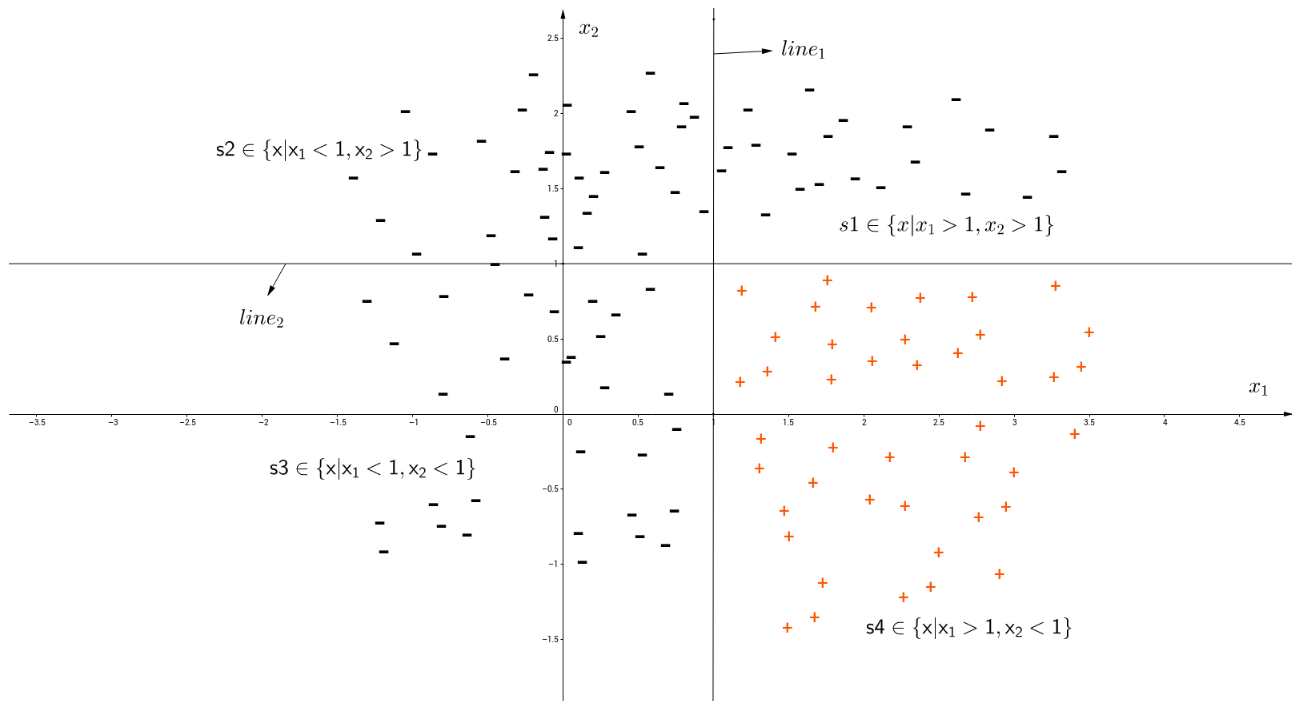

Figure 1: Label the samples as negative(-1) or positive(+1)

In Figure 1 to separate those samples with certain method, firstly we need to give the activation function, used to squashing the output:

$$sign(x) = \begin{cases} 1, & x > 0; \\ -1, & x \leq 0; \end{cases} \tag{1.2.1}$$

We define two perceptrons $p_1(x) = sign(x_1 - 1)$, $p_2(x) = sign(1 - x_2)$, you will find $line_1$ and $line_2$ are respectively the hyper-plane of perceptron p_1 and p_2. For perceptron p_1, samples in space s_2 and s_3 are labeled -1(negative), otherwise 1(positive). For perceptron p_2, samples in space s_1, s_2 are classified as negative, and others classified as positive.

What if combining these two perceptrons, it may work. I mean given a sample x, applying conjunction operation $p_1(x) \wedge p_2(x)$ will output the correct prediction. Then how can we achieve such operations in neural network way(actually feed-forward neural network[4])? We only needs some matrices.

$$\begin{bmatrix} x_1 & x_2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 1 & -1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} x_1 - 1 & 1 - x_2 \end{bmatrix}$$

then squash the output with function $sign(x)$:

$$\begin{bmatrix} sign(x_1 - 1) & sign(1 - x_2) \end{bmatrix} = \begin{bmatrix} p_1(x) & p_2(x) \end{bmatrix}$$

$$\begin{bmatrix} p_1(x) & p_2(x) & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = p_1(x) + p_2(x)$$

squash the output again, the finally result will be: $f(x) = sign(p_1(x) + p_2(x))$.

$$f(s_1) = sign(1 + (-1)) = sign(0) = -1$$
$$f(s_2) = sign(-1 + (-1)) = sign(-2) = -1$$
$$f(s_3) = sign(-1 + 1) = sign(0) = -1$$
$$f(s_4) = sign(1 + 1) = sign(2) = 1$$

You will find that $f(x)$ is what we are looking for to separate these samples. If you have make sense of this example, I'm glad to tell you that you are one step away from writing your own neural network since you have known how this network propagates input and produce output.

Here we just combined perceptrons p_1 and p_2, they can handle something that single perceptron never can do. With more perceptrons, it can be more powerful and actually that's the way feed-forward neural network works. According to universal approximation theorem[5] a feed-forward network with a single hidden layer containing a finite number of neurons (i.e., a multilayer perceptron), can approximate continuous functions on compact subsets of R^n

Anyway, in this example, I give the transform matrices directly, however, in neural network they can be computed with certain optimization algorithms.

Perceptron considered as the simplest neural network, has just 2 layers of nodes (input nodes and output nodes). Often called a single-layer network on account of having 1 layer of links, between input and output. In the above example, logistic operation of conjunction is applied, how about achieving disjunction or exclusive disjunction? I hope you can find it out yourself.

2 Talk is cheap, show me the code.

The following example is the so-called neural network(actually, feed-forward neural network). Do not consider it too difficult, in essence, I ensure you there are only some matrix multiplications done by Python code like the perceptron example. Our code is to train a model that outputs 1 when the first and second unit of a sample is the same value, otherwise outputs 0. You are encouraged to update the iteration count and step_size, change samples and anything as you wish to get direct perception.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""simple-neural-network.py: Train a model that outputs 1 when the first
and second unit of a sample is the same value, otherwise outputs 0. """

__author__    = "Cai-Jun Sun"
__copyright__ = "Copyright 2016, IoVision"
__license__   = "GPL"
__version__   = "1.0.0"
__maintainer__= "Cai-Jun Sun"
__email__   = "sun(dot)caijun(at)bupt(dot)edu(dot)cn"
__status__   = "Production"

import numpy as np

def sigmod(x):
    return 1/(1+np.exp(-x))

def test(x):
    l1 = sigmod(np.dot(x,syn0))
    return sigmod(np.dot(l1,syn1))

step_size = 1 # step size for gradient descent
iteration_count = 1000

X = np.array([[0,0,0], [0,0,1], [0,1,0], [0,1,1], [1,0,0], [1,0,1], [1,1,0]])
Y = np.array([[1,1,0,0,0,0,1]]).T

np.random.seed(1)

# random initial weights, ranging from -0.5 ~ 0.5
syn0 = 2 * np.random.random((3,15)) - 1
syn1 = 2 * np.random.random((15,1)) - 1

for j in xrange(iteration_count):
    Samples = X # input layer
    Hidden = sigmod(np.dot(Samples,syn0)) # hidden layer
    Output = sigmod(np.dot(Hidden,syn1)) # output layer
```

```
40
41     syn1 += (-1) * step_size * Hidden.T.dot((Output - Y) * Output * (1 - Output))
42     syn0 += (-1) * step_size * Samples.T.dot(((Output - Y) * Output * (1 - Output))\
43                                     .dot(syn1.T) * Hidden * (1-Hidden))
44
45 print test([1,1,1]) # output 0.48776692
46 print test([2,2,2]) # output 0.80574699
47 print test([3,3,3]) # output 0.86762845
```

3 Input Forward Propagation and Error Back Propagation

The above example is a feed-forward neural network with a single hidden layer optimized by algorithm of BP, an abbreviation for Backward Propagation of Errors[6]. BP is a common method for training artificial neural networks, usually implemented by Gradient Descent. We take *sigmoid* here as the activation function(aka squashing function[7], used to compress the outputs of the "neurons" in multi-layer neural network), however, there are still other alternatives, tanh will do in this experiment. In our code we even haven't take biases into consideration, here we just want to make the neural network dead simple.

Here I will explain in a general way how our simple neural network propagates information. Then after you have a better understanding of such kind of network, we move to the next step of optimizing the network.

3.1 Input Forward Propagation

Let's get started, we have m samples S in n dimensions space and corresponding labels Y.

$$samples : S_{m*n} = \begin{bmatrix} S_{11} & \dots & S_{1n} \\ \multicolumn{3}{c}{\dots\dots\dots\dots} \\ S_{m1} & \dots & S_{mn} \end{bmatrix} ; targets : Y_{m*1} = \begin{bmatrix} Y_1 \\ \dots \\ Y_m \end{bmatrix}$$

At beginning we define a first synapse layer of weights $syn0$, which connects input layer with hidden layer. All the variables in this matrix will be optimized during the progress of error back propagation:

$$syn0_{n*p} = \begin{bmatrix} syn0_{11} & \dots & syn0_{1p} \\ \multicolumn{3}{c}{\dots\dots\dots\dots} \\ syn0_{n1} & \dots & syn0_{np} \end{bmatrix}$$

We got temporary matrix Hr by $S_{m*n} \cdot syn0_{n*p}$, or say it's a layer just like hidden layer. But it's not a real hidden layer before squashing, I mean passing it to the activation function:

$$Hr_{m*p} = \begin{bmatrix} Hr_{11} & \dots & Hr_{1p} \\ \multicolumn{3}{c}{\dots\dots\dots\dots} \\ Hr_{m1} & \dots & Hr_{mp} \end{bmatrix}, \ where \ Hr_{xy} = \sum_{i=1}^{i=n} S_{xi} * syn0_{iy},$$

Then we squash the above layer with *sigmoid* function, we derive the hidden layer:

$$H_{m*p} = \begin{bmatrix} H_{11} & \dots & H_{1p} \\ \multicolumn{3}{c}{\dots\dots\dots\dots} \\ H_{m1} & \dots & H_{mp} \end{bmatrix}, \ where \ H_{xy} = \frac{1}{1 + e^{-Hr_{xy}}}$$

Before projecting the hidden layer to the output layer we have to define a connection layer as same as $syn0$ connecting the hidden layer and output layer:

$$syn1_{p*1} = \begin{bmatrix} syn1_1 \\ \dots\dots \\ syn1_p \end{bmatrix}$$

With the $syn1$ we get a temporary matrix Or by $H_{m*p} \cdot syn1$. It's just one step far from the output layer.

$$Or_{m*1} = \begin{bmatrix} Or_1 \\ \dots\dots \\ Or_m \end{bmatrix},\ where\ Or_x = \sum_{i=1}^{i=p} H_{xi} * syn1_i$$

Finally we get the output layer O by squashing the above matrix with *sigmoid* function:

$$O_{m*1} = \begin{bmatrix} O_1 \\ \dots \\ O_m \end{bmatrix},\ where\ Ox = \frac{1}{1 + e^{-Or_x}}$$

So far, we described how our simple neural network propagates information with some matrices. You can see that it's all matrix multiplication, nothing complicated.

3.2 Error Back Propagation

To evaluate the network, the cost or loss function can be expressed like this:

$$L = \frac{1}{2} \sum_{i=1}^{i=m} (O_i - Y_i)^2 \tag{3.2.1}$$

$\frac{1}{2}$ before $\sum$ is convenient for calculating the derivative. Since the loss function defined, it's time to tune the parameters in $syn0$ and $syn1$. The following method is the so-called BP algorithm implemented by Batch Gradient Descent.

Firstly, we update the second weight matrix $syn1$ which connecting the hidden and output layer. Consider sample S_i, its label is Y_i and prediction is O_i, we can calculate the partial derivative of loss function L with respect to parameter $syn1_x$ by applying the chain rule:

$$\frac{\partial L}{\partial syn1_x} = \frac{\partial L}{\partial O_i} * \frac{\partial O_i}{\partial Or_i} * \frac{\partial Or_i}{\partial syn1_x} \tag{3.2.2}$$

We can repeat this process to get all the partial derivatives. After all the partial derivatives derived, the gradient

can be written in this way:

$$grad1_i = \begin{bmatrix} \frac{\partial L}{\partial syn1_1} \\ \ldots\ldots \\ \frac{\partial L}{\partial syn1_p} \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial O_i} * \frac{\partial O_i}{\partial Or_i} * \frac{\partial Or_i}{\partial syn1_1} \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ \frac{\partial L}{\partial O_i} * \frac{\partial O_i}{\partial Or_i} * \frac{\partial Or_i}{\partial syn1_p} \end{bmatrix} \tag{3.2.3}$$

$$= \begin{bmatrix} (O_i - Y_i) * O_i * (1 - O_i) * H_{i1} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ (O_i - Y_i) * O_i * (1 - O_i) * H_{ip} \end{bmatrix} \tag{3.2.4}$$

$$= (O_i - Y_i) * O_i * (1 - O_i) * H_{i*}. \tag{3.2.5}$$

H_{i*} is the ith row of hidden layer H, then we update connection matrix $syn1$ through the gradient, I mean the negative direction of gradient:

$$syn1' = syn1 + (-1) * grad1_i * alpha$$

in this equation $alpha$ is the step size and i is the sample index in train set. Anyway this technique is call Stochastic Gradient Descent. In this experiment we can update $syn1$ in a batch way, as I said at the very beginning using Batch Gradient Descent. You will find that we simply accumulated the gradients with regard to each sample as following calculation.

$$syn1' = syn1 + (-1) * alpha * \sum_{i=1}^{m} grad1_i \tag{3.2.6}$$

$$= syn1 + (-1) * alpha * \sum_{i=1}^{m} (O_i - Y_i) * O_i * (1 - O_i) * H_{i*} \tag{3.2.7}$$

$$= syn1 + (-1) * alpha * H^T \cdot (O - Y) * O * (1 - O) \tag{3.2.8}$$

You may ask how get we get calculation from 3.2.7 to 3.2.8. Before providing you the proof, you should notice here operators $\cdot$ and $*$ are different in matrix multiplication of which the former is dot production and latter is element-wise multiplication(also known as Hadamard Product).

$$\sum_{i=1}^{m} (O_i - Y_i) * O_i * (1 - O_i) * H_{i*} \tag{3.2.9}$$

$$= \begin{bmatrix} (O_1 - Y_1) * O_1 * (1 - O_1) * H_{11} + \ldots + (O_m - Y_m) * O_m * (1 - O_m) * H_{m1} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ (O_1 - Y_1) * O_1 * (1 - O_1) * H_{1p} + \ldots + (O_m - Y_m) * O_m * (1 - O_m) * H_{mp} \end{bmatrix} \tag{3.2.10}$$

$$= \begin{bmatrix} H_{11} \ldots H_{m1} \\ \ldots\ldots\ldots \\ H_{1p} \ldots H_{mp} \end{bmatrix} \cdot \begin{bmatrix} (O_1 - Y_1) * O_1 * (1 - O_1) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ (O_m - Y_m) * O_m * (1 - O_m) \end{bmatrix} \tag{3.2.11}$$

$$= H^T \cdot (O - Y) * O * (1 - O) \tag{3.2.12}$$

We now update syn0 with sample S_i in a same manner above by applying the rule chain, even a little bit longer than previous one:

$$\frac{\partial L}{\partial syn0_{xy}} = \frac{\partial L}{\partial O_i} * \frac{\partial O_i}{\partial Or_i} * \frac{\partial Or_i}{\partial H_{iy}} * \frac{\partial H_{iy}}{\partial Hr_{iy}} * \frac{\partial Hr_{iy}}{\partial syn0_{xy}}$$

since $syn0_{11}$ only exists in Hr_{x1}, x the sample id, $1 \le x \le m$. For sample S_i,

$$Hr_{iy} = S_{ix} \cdot syn0_{xy}, \quad H_{iy} = \frac{1}{1 + e^{-Hr_{iy}}}, \quad Or_i = \sum_{j=1}^{p} H_{ij} * syn1_j$$

,

We can easily derive the partial derivative of loss function with respect to parameter $syn0_{xy}$:

$$\frac{\partial L}{\partial syn0_{xy}} = (O_i - Y_i) * O_i * (1 - O_i) * syn1_y * H_{iy} * (1 - H_{iy}) * S_{ix} \tag{3.2.13}$$

Continue such process, we get derivatives about all the parameters in connection matrix $syn0$, the gradient can be expressed like this:

$$\begin{aligned}
grad0_i &= \begin{bmatrix} \frac{\partial L}{\partial syn0_{11}} & \cdots & \frac{\partial L}{\partial syn0_{1p}} \\ \cdots & \cdots & \cdots \\ \frac{\partial L}{\partial syn0_{n1}} & \cdots & \frac{\partial L}{\partial syn0_{np}} \end{bmatrix} && (3.2.14)\\
&= (O_i - Y_i) * O_i * (1 - O_i) * (\begin{bmatrix} S_{i1} \\ \cdots \\ S_{in} \end{bmatrix} \cdot \begin{bmatrix} syn1_1 * H_{i1} * (1 - H_{i1}) & \ldots & syn1_p * H_{ip} * (1 - H_{ip}) \end{bmatrix}) && (3.2.15)\\
&= (O_i - Y_i) * O_i * (1 - O_i) * ((S_{i*})^T \cdot (syn1^T * H_{i*} * (1 - H_{i*})) && (3.2.16)\\
&= (S_{i*})^T \cdot ((O_i - Y_i) * O_i * (1 - O_i) * syn1^T * H_{i*} * (1 - H_{i*})) && (3.2.17)
\end{aligned}$$

Note that $(S_{i*})^T)$ is the ith sample of train data, which is transposed to a column space. H_{i*} is the ith row of hidden layer. It's a little complicated by apply Batch Gradient Descent since there are so many parameters, also you can imaging that it's a disaster once hidden layer doubles. Let's get back to updating work.

$$\begin{aligned}
syn0' &= syn0 + (-1) * alpha * \sum_{i=1}^{m} \cdot ((O_i - Y_i) * O_i * (1 - O_i) * syn1^T * H_{i*} * (1 - H_{i*})) && (3.2.18)\\
&= syn0 + (-1) * alpha * S^T \cdot (\begin{bmatrix} (O_1 - Y_1) * O_1 * (1 - O_1) * syn1^T * H_{1*} * (1 - H_{1*}) \\ \cdots \\ (O_m - Y_m) * O_m * (1 - O_m) * syn1^T * H_{m*} * (1 - H_{m*}) \end{bmatrix}) && (3.2.19)\\
&= syn0 + (-1) * alpha * S^T \cdot (\begin{bmatrix} (O_1 - Y_1) * O_1 * (1 - O_1) \\ \cdots \\ (O_m - Y_m) * O_m * (1 - O_m) \end{bmatrix} * \begin{bmatrix} syn1^T \\ \cdots \\ syn1^T \end{bmatrix} * \begin{bmatrix} H_1 * (1 - H_{1*}) \\ \cdots \\ H_m * (1 - H_{m*}) \end{bmatrix}) && (3.2.20)\\
&= syn0 + (-1) * alpha * S^T \cdot (((O - Y) * O * (1 - O)) \cdot syn1^T * H * (1 - H)) && (3.2.21)
\end{aligned}$$

You may be confused about equation 3.2.21, here we will explain in detail:

$$
\begin{aligned}
&\begin{bmatrix} (O_1 - Y_1) * O_1 * (1 - O_1) \\ \dots\dots\dots\dots\dots\dots \\ (O_m - Y_m) * O_m * (1 - O_m) \end{bmatrix} * \begin{bmatrix} syn1^T \\ \dots\dots \\ syn1^T \end{bmatrix} \\
&= \begin{bmatrix} (O_1 - Y_1) * O_1 * (1 - O_1) * syn1_1 & \dots & (O_1 - Y_1) * O_1 * (1 - O_1) * syn1_m \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ (O_m - Y_m) * O_m * (1 - O_m) * syn1_1 & \dots & (O_m - Y_m) * O_m * (1 - O_m) * syn1_m \end{bmatrix} \\
&= \begin{bmatrix} (O_1 - Y_1) * O_1 * (1 - O_1) \\ \dots\dots\dots\dots\dots\dots \\ (O_m - Y_m) * O_m * (1 - O_m) \end{bmatrix} \cdot \begin{bmatrix} syn1_1 \dots syn1_p \end{bmatrix} \\
&= ((O - Y) * O * (1 - O)) \cdot syn1^T
\end{aligned}
\tag{3.2.22}
$$

Finally the first round of back propagation is done, we get the formulas of tuning the connection matrix.

$$syn1' = syn1 + (-1) * alpha * H^T \cdot (O - Y) * O * (1 - O) \tag{3.2.23}$$

$$syn0' = syn0 + (-1) * alpha * S^T \cdot (((O - Y) * O * (1 - O)) \cdot syn1^T * H * (1 - H)) \tag{3.2.24}$$

Anyway, if you have heard of deep learning, you may know the problem of vanishing gradient[8]. Since $H_{ii} \in [0, 1]$, $H_{ii} * (1 - H_{ii}) \leq 0.25$. With more hidden layers, the gradient will decrease exponentially which means at least 75% dropped from the front layer. Gradient Descent cannot be directly applied to train deep neural network. What can we do to optimize weights of synapse which connecting two successive layers? Do pre-training[9] work well? Should we still use BP to fine tune the weights? Latter we will talk about Deep Learning, another interesting filed.

Some methods proposed to solve the problem of vanishing gradient: Use ReLU, maxout instead of sigmod, Highway network and deep residual learning.

4 Future Work

In this post, we talked about how to use perceptron to achieve conjunction operation, the relationship between perceptron and neural network, the way neural network works, and most importantly how to implement a single layer neural network optimized by BP algorithm. We also mentioned the problem of vanishing gradient at the end and provided some solutions. You are expected to have a basic idea of what is neural network and how neural network works and optimized. Anyway, there is still a bunch of problems to be solved, I mean it's not that simple in real world when we use neural network to solve problems.

The example neural network is just a feed-forward neural network, there are still lots of neural networks haven't covered, such as Radial Basis Function Neural Network, ART(Adaptive Resonance Theory[10]) Neural Network, Self-Organizing Map Neural Network[11][12], Cascade-Correlation Neural Network[13], Recurrent Neural Network[14], Boltzmann Machine[15], to name a few. You are expected to take a deep learning.

Here the Back Propagation algorithm relies on Batch Gradient Descent. To be honest, the performance sucks. How about switching to Stochastic Gradient Descent, will it out-perform the former one? And I can say that the train samples also matter the train result. You are strongly recommended to investigate it and dig deep.

In complicated networks with limited data, how can we prevent neural network from overfitting? What and how to achieve regularization[16] and early stopping[17]? G Hinton present a novel method called dropout[18], will that help?

The real world data contains plenty noisy samples, have you ever think about denoising[19] or utilize these noises?

References

[1] Cai-Jun Sun. *Gradient Descent Example Code*. 3 March 2016. URL: `https://www.iovi.com/post/2016-03-19-gradient-descent-example-code.html`.

[2] Mark Humphrys. *Single-layer Neural Networks (Perceptrons)*. 3 March 2011. URL: `http://computing.dcu.ie/~humphrys/Notes/Neural/single.neural.html`.

[3] Maureen Caudill. "Neural networks primer, part I". In: *AI expert* 2.12 (1987), pp. 46–52.

[4] George Bebis and Michael Georgiopoulos. "Feed-forward neural networks". In: *Potentials, IEEE* 13.4 (1994), pp. 27–31.

[5] Balázs Csanád Csáji. "Approximation with artificial neural networks". In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24 (2001), p. 48.

[6] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3 (1988), p. 1.

[7] Tom M Mitchell. "Artificial neural networks". In: *Machine learning* (1997), pp. 81–127.

[8] Sepp Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[9] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 807–814.

[10] Stephen Grossberg. *Adaptive resonance theory*. Wiley Online Library, 2003.

[11] Teuvo Kohonen. "The self-organizing map". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480.

[12] Teuvo Kohonen and Panu Somervuo. "Self-organizing maps of symbol strings". In: *Neurocomputing* 21.1 (1998), pp. 19–30.

[13] Scott E Fahlman and Christian Lebiere. "The cascade-correlation learning architecture". In: (1989).

[14] Christoph Goller and Andreas Kuchler. "Learning task-dependent distributed representations by backpropagation through structure". In: *Neural Networks, 1996., IEEE International Conference on*. Vol. 1. IEEE. 1996, pp. 347–352.

[15] Emile Aarts and Jan Korst. "Simulated annealing and Boltzmann machines". In: (1988).

[16] Yaochu Jin, Tatsuya Okabe, and Bemhard Sendhoff. "Neural network regularization and ensembling using multi-objective evolutionary algorithms". In: *Evolutionary Computation, 2004. CEC2004. Congress on*. Vol. 1. IEEE. 2004, pp. 1–8.

[17] Rich Caruana Steve Lawrence Lee Giles. "Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping". In: *Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*. Vol. 13. MIT Press. 2001, p. 402.

[18] Nitish Srivastava et al. "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[19] Pascal Vincent et al. "Extracting and composing robust features with denoising autoencoders". In: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, pp. 1096–1103.