How Neural Networks (NN) Can (Hopefully) Learn Faster by Taking Into Account Known Constraints

Chitta Baral[1], Martine Ceberio[2], and Vladik Kreinovich[2]

[1] Department of Computer Science, Arizona State University
Tempe, AZ 85287-5406, USA, chitta@asu.edu
[2] Department of Computer Science, University of Texas at El Paso
El Paso, TX 79968, USA, mceberio@utep.edu, vladik@utep.edu

Abstract. Neural networks are a very successful machine learning technique. At present, deep (multi-layer) neural networks are the most successful among the known machine learning techniques. However, they still have some limitations, One of their main limitations is that their learning process still too slow. The major reason why learning in neural networks is slow is that neural networks are currently unable to take prior knowledge into account. As a result, they simple ignore this knowledge and simulate learning "from scratch". In this paper, we show how neural networks can take prior knowledge into account and thus, hopefully, learn faster.

1 Formulation of the Problem

Need for machine learning. In many practical situations, we know that the quantities $y_1, \ldots, y_L$ depend on the quantities $x_1, \ldots, x_n$, but we do not know the exact formula for this dependence. To get this formula, we measure the values of all these quantities in different situations $m = 1, \ldots, M$, and then use the corresponding measurement results $x_i^{(m)}$ and $y_\ell^{(m)}$ to find the corresponding dependence. Algorithms that "learn" the dependence from the measurement results are known as *machine learning* algorithms.

Neural networks (NN): main idea and successes. One of the most widely used machine learning techniques is the technique of *neural networks* (NN) – which is based on a (simplified) simulation of how actual neurons works in the human brain (a brief technical description of this technique is given in Section 2). This technique has many useful applications; see, e.g., [1, 2].

At present (2016) multi-layer ("deep") neural networks are, empirically, the most efficient of the known machine learning techniques.

Neural networks: limitations. One of the main limitations of neural networks is that their learning very slow: they need many thousand iterations just to learn a simple dependence.

This slowness is easy to explain: the current neural networks always start "from scratch", from zero knowledge. In terms of simulating human brain, they do not simulate how we learn the corresponding dependence – they simulate how a newborn child will eventually learn to recognize this dependence. Of course, this inability to take any prior knowledge into account drastically slows down the learning process.

What is prior knowledge. Prior knowledge means that we know some relations ("constraints") between the desired values $y_1, \ldots, y_L$ and the observed values $x_1, \ldots, x_n$, i.e., we know several relations of the type $f_c(x_1, \ldots, x_n, y_1, \ldots, y_L) = 0$, $1 \le c \le C$.

Prior knowledge helps humans learn faster. Prior knowledge helps us learn. Yes, it takes some time to learn this prior knowledge, but this has been done *before* we have samples of x_i and y_ℓ. As a result, the time from gathering the samples to generating the desired dependence decreases.

This is not simply a matter of accounting: the same prior knowledge can be used (and usually is used) in learning several different dependencies. For example, our knowledge of sines, logarithms, of calculus helps in finding the proper dependence in many different situations. So, when we learn the prior knowledge first, we decrease the overall time needed to learn all these dependencies.

How to speed up artificial neural networks: a natural idea. In view of the above explanation, a natural idea is to enable neural networks to take prior knowledge into account. In other words, instead of learning all the data "from scratch", we should first learn the constraints. Then, when it is time to use the data, we should be able to use these constraints to "guide" the neural network in the right direction.

What we do in this paper. In this paper, we show how to implement this idea and thus, how to (hopefully) achieve the corresponding speed-up.

To describe this idea, we first, in Section 2, recall how the usual NN works. Then, in Section 3, we show how we can perform a preliminary training of a NN, so that it can learn to satisfy the given constraints. Finally, in Section 4, we show how to train the resulting pre-trained NN in such a way that the constraints remain satisfied.

2 Neural Networks: A Brief Reminder

Signals in a biological neural network. In a biological neural network, a signal is represented by a sequence of spikes. All these spikes are largely the same, what is different is how frequently the spikes come.

Several sensor cells generate such sequences: e.g., there are cells that translate the optical signal into spikes, there are cells that translate the acoustic signal into spikes. For all such cells, the more intense the original physical signal, the more spikes per unit time it generates. Thus, the frequency of the spikes can serve as a measure of the strength of the original signal.

From this viewpoint, at each point in a biological neural network, at each moment of time, the signal can be described by a single number: namely, by the frequency of the corresponding spikes.

What is a biological neuron: a brief description. A biological neuron has several inputs and one output. Usually, spikes from different inputs simply get together – probably after some filtering. Filtering means that we suppress a certain proportion of spikes. If we start with an input signal x_i, then, after such a filtering, we get a decreased signal $w_i \cdot x_i$. Once all the inputs signals are combined, we have the resulting signal $\sum\limits_{i=1}^{n} w_i \cdot x_i$.

A biological neuron usually has some excitation level w_0, so that if the overall input signal is below w_0, there is practically no output. The intensity of the output signal thus depends on the difference $d \stackrel{\text{def}}{=} \sum\limits_{i=1}^{n} w_i \cdot x_i - w_0$. Some neurons are linear, their output is proportional to this difference. Other neurons are non-linear, they output is equal to $s_0(d)$ for some non-linear function $s_0(z)$. Empirically, it was found that the corresponding non-linear transformation takes the form $s_0(z) = 1/(1 + \exp(-z))$.

Comment. It should be mentioned that this is a simplified description of a biological neuron: the actual neuron is a complex *dynamical* system, in the sense that its output at a given moment of time depends not only on the current inputs, but also on the previous input values.

Artificial neural networks and how they learn. If we need to predict the values of several outputs $y_1, \ldots, y_\ell, \ldots, y_L$, then for each output y_ℓ, we train a separate neural network.

In an artificial neural networks, input signals $x_1, \ldots, x_n$ first go to the neurons of the first layer, then the results go to neurons of the second layer, etc.

In the simplest (and most widely used) arrangement, the second layer has linear neurons. In this arrangement, the neurons from the first layer produce the signals $y_{\ell,k} = s_0\left(\sum\limits_{i=1}^{n} w_{\ell,ki} \cdot x_i - w_{\ell,k0}\right)$, $1 \le k \le K_\ell$, which are then combined into an output $y_\ell = \sum\limits_{k=1}^{K} W_{\ell,k} \cdot y_k - W_{\ell,0}$. This is called *forward propagation.* (In this paper, we will only describe formulas for this arrangement, since formulas for the multi-layer neural networks can be obtained by using the same idea.)

How a NN learns: derivation of the formulas. Once we have an observation $(x_1^{(m)}, \ldots, x_n^{(m)}, y_\ell^{(m)})$, we first input the values $x_1^{(m)}, \ldots, x_n^{(m)}$ into the current NN; the network generates some output $y_{\ell,NN}$. In general, this output is different from the observed output $y_\ell^{(m)}$. We therefore want to modify the weights $W_{\ell,k}$ and $w_{\ell,ki}$ so as to minimize the squared difference $J \stackrel{\text{def}}{=} (\Delta y_\ell)^2$, where $\Delta y_\ell \stackrel{\text{def}}{=} y_{\ell,NN} - y_\ell^{(m)}$. This minimization is done by using gradient descent, where each of the unknown values is updated as $W_{\ell,k} \to W_{\ell,k} - \lambda \cdot \dfrac{\partial J}{\partial W_{\ell,k}}$ and $w_{\ell,ki} \to$

$w_{\ell,ki} - \lambda \cdot \dfrac{\partial J}{\partial w_{\ell,ki}}$. The resulting algorithm for updating the weights is known as *backpropagation*. This algorithm is based on the following idea.

First, one can easily check that $\dfrac{\partial J}{\partial W_{\ell,0}} = -2\Delta y$, so $\Delta W_{\ell,0} = -\lambda \cdot \dfrac{\partial J}{\partial W_{\ell,0}} = \alpha \cdot \Delta y_\ell$, where $\alpha \stackrel{\text{def}}{=} 2\lambda$. Similarly, $\dfrac{\partial J}{\partial W_{\ell,k}} = 2\Delta y_\ell \cdot y_{\ell,k}$, so $\Delta W_{\ell,k} = -\lambda \cdot \dfrac{\partial J}{\partial W_{\ell,k}} = 2\lambda \cdot \Delta y_\ell \cdot y_{\ell,k}$, i.e., $\Delta W_{\ell,k} = -\Delta W_{\ell,0} \cdot y_{\ell,k}$.

The only dependence of y_ℓ on $w_{\ell,ki}$ is via the dependence of $y_{\ell,k}$ on $w_{\ell,ki}$. So, for $w_{\ell,k0}$, we can use the chain rule and get $\dfrac{\partial J}{\partial w_{\ell,k0}} = \dfrac{\partial J}{\partial y_{\ell,k}} \cdot \dfrac{\partial y_{\ell,k}}{\partial w_{\ell,k0}}$, hence:

$$\frac{\partial J}{\partial w_{\ell,k0}} = 2\Delta y_\ell \cdot W_{\ell,k} \cdot s_0'\left(\sum_{i=1}^{n} w_{\ell,ki} \cdot x_i - w_{\ell,k0}\right) \cdot (-1).$$

For $s_0(z) = 1/(1+\exp(-z))$, we have $s_0'(z) = \exp(-z)/(1+\exp(-z))^2$, i.e.,

$$s_0'(z) = \frac{\exp(-z)}{1+\exp(-z)} \cdot \frac{1}{1+\exp(-z)} = s_0(z) \cdot (1 - s_0(z)).$$

Thus, in the above formula, where $s_0(z) = y_{\ell,k}$, we get $s_0'(z) = y_{\ell,k} \cdot (1 - y_{\ell,k})$, $\dfrac{\partial J}{\partial w_{\ell,k0}} = -2\Delta y_\ell \cdot W_{\ell,k} \cdot y_{\ell,k} \cdot (1 - y_{\ell,k})$, and

$$\Delta w_{\ell,k0} = -\lambda \cdot \frac{\partial J}{\partial w_{\ell,k0}} = \lambda \cdot 2\Delta y_\ell \cdot W_{\ell,k} \cdot y_{\ell,k} \cdot (1 - y_{\ell,k}).$$

So, we have $\Delta w_{\ell,k0} = -\Delta W_{\ell,k} \cdot W_{\ell,k} \cdot (1 - y_{\ell,k})$. For $w_{\ell,ki}$, we have

$$\frac{\partial J}{\partial w_{\ell,ki}} = 2\Delta y_\ell \cdot W_{\ell,k} \cdot y_{\ell,k} \cdot (1 - y_{\ell,k}) \cdot x_i = -\frac{\partial J}{\partial w_{\ell,k0}} \cdot x_i,$$

hence $\Delta w_{\ell,ki} = -x_i \cdot \Delta w_{\ell,k0}$. Thus, we arrive at the following algorithm:

Resulting algorithm. We pick some value α, and cycle through observations $(x_1, \ldots, x_n)$ with the desired outputs y_ℓ. For each observation, we first apply the forward propagation to compute the network's prediction $y_{\ell,NN}$, then we compute $\Delta y_\ell = y_{\ell,NN} - y_\ell$, $\Delta W_{\ell,0} = \alpha \cdot \Delta y_\ell$, $\Delta W_{\ell,k} = -\Delta W_{\ell,0} \cdot y_{\ell,k}$, $\Delta w_{\ell,k0} = -\Delta W_{\ell,k} \cdot W_{\ell,k} \cdot (1 - y_{\ell,k})$, and $\Delta w_{\ell,ki} = -\Delta w_{\ell,k0} \cdot x_i$, and update each weight w to $w_{\text{new}} = w + \Delta w$. We repeat this procedure until the process converges.

3 How to Pre-Train a NN to Satisfies Given Constraints

To train the network, we can use any observations $(x_1^{(m)}, \ldots, x_n^{(m)}, y_1^{(m)}, \ldots, y_L^{(m)})$ that satisfy all the known constraints.

To satisfy the constraints $f_c(x_1, \ldots, x_n, y_1, \ldots, y_L) = 0$, $1 \le c \le C$, means to minimize the distance from the vector of values $(f_1, \ldots, f_C)$ to

the ideal point $(0, \ldots, 0)$, i.e., equivalently, to minimize the sum $F \stackrel{\text{def}}{=} \sum_{c=1}^{C} (f_c(x_1, \ldots, x_n, y_1, \ldots, y_L))^2$. To minimize this sum, we can use a similar gradient descent idea. From the mathematical viewpoint, the only difference from the usual backpropagation is the first step: here,

$$\frac{\partial F}{\partial W_{\ell,0}} = 2 \cdot \sum_{c=1}^{C} f_c \cdot \frac{\partial f_c}{\partial y_\ell}, \quad \text{hence} \quad \Delta W_{\ell,0} = -\alpha \cdot \sum_{c=1}^{C} f_c \cdot \frac{\partial f_c}{\partial y_\ell}.$$

Once we have computed $\Delta W_{\ell,0}$, all the other changes $\Delta W_{\ell,k}$ and $\Delta w_{\ell,ki}$ are computed based on the same formulas as above.

The consequence of this algorithm modification is that instead of L independent neural networks used to train each of the L outputs y_ℓ, now we have L dependent ones. The dependence comes from the fact that, to start a new cycle for each ℓ, we need to know the values $y_1, \ldots, y_L$ corresponding to all the outputs.

4 How to Retain Constraints When Training Neural Networks on Real Data

Once the networks is pre-trained so that the constraints are all satisfied, we need to train it on the real data. In this real-data training, we need to make sure that not only all the given data points fit, but that also all C constraints remain satisfied. In other words, on each step, we need to make sure not only that Δy_ℓ is close to 0, but also that $f_c(x_1, \ldots, x_n, y_1, \ldots, y_L)$ is close to 0 for all ℓ. So, similar to the previous section, instead of minimizing $J = (\Delta y_\ell)^2$, we should minimize a combined objective function $G \stackrel{\text{def}}{=} J + N \cdot F$, where N is an appropriate constant, and $F = \sum_{c=1}^{C} f_c^2$.

Similarly to pre-training, the only difference from the usual backpropagation algorithm is that we compute the values $\Delta W_{\ell,0}$ differently:

$$\Delta W_{\ell,0} = \alpha \cdot \left(\Delta y_\ell - N \cdot \sum_{c=1}^{C} f_c \cdot \frac{\partial f_c}{\partial y_\ell} \right).$$

Acknowledgments. This work was supported in part by NSF grants HRD-0734825, HRD-1242122, and DUE-0926721, and by an award from Prudential Foundation.

References

1. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, N.Y., 2006.
2. G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets", *Neural Computation*, 2006, Vol. 18, pp. 1527–1554.