Learning with Artifical Neural Networks

Daniel Khashabi

Fall 2016
Last Update: October 5, 2016

1 Introduction

The discussion of Neural Networks started off by creating good approximators for real-valued, discrete-valued and vector-valued target functions. Neural Networks are inpired by biological systems, which are the best examples we have of robust learning-based systems. In most of the neural structures, a big set of computational units (neurons) are connected to each other with a specific toplogy and via an algebraic transformation. Another feature that exist in natural neural systems, is their massive parallelism that may allow for computational efficiency.

Another issue that came along with supporters of drawing inspirations from nature is the distributed representation which exists in mind. For example, all the concepts and knowledge in someone's mind are result of a speicific wiring of neurons in brain, and the intelligent behavior "emerges" from large number of simple units rather than from explicit symbolically encoded rules. It was particulatly interesting that their application to different problems with various input-output architectures needed minimal amount of changes.

In practice NNs are over-simplification of the actual neural structure of the brain How close/similar the current models are to the brain neurons, is a highly debate. (See ??? for more discussion). Human brain has approximately 10^{10} neurons, each connected to other 10^4 neurons. Natural neuron "switching time" is O(milliseconds), compared to nanosecond for transistors. However, biological systems can perform significant cognitive tasks (vision, language understanding) in fractions of a second as a result of their massive parralelism.

NNs have been shown to be successful in many practicalproblems such as handwritten character recognition, spoken words recognition, face recognition, etc. The algorithm which helped NNs survived, and even come back stronger is the Back-Propagation algorithm.

Here I will summarize some basic and important algorithms relevant to Neural Networks, and some theory their properties.

2 Multi-layer feedforward networks

Multi-layer network were designed to overcome the computational (expressivity) limitation of a single threshold element. The idea is to stack several layers of threshold elements, each layer using the output of the previous layer as input. Multi-layer networks can represent arbitrary functions, but building effective learning methods for such network was [thought to be] difficult.

NNs are meant to represent intricate nonlinear behaviors. This can be achieved only if there exists a nonlinearity at each connection. Here are some common nonliearities:

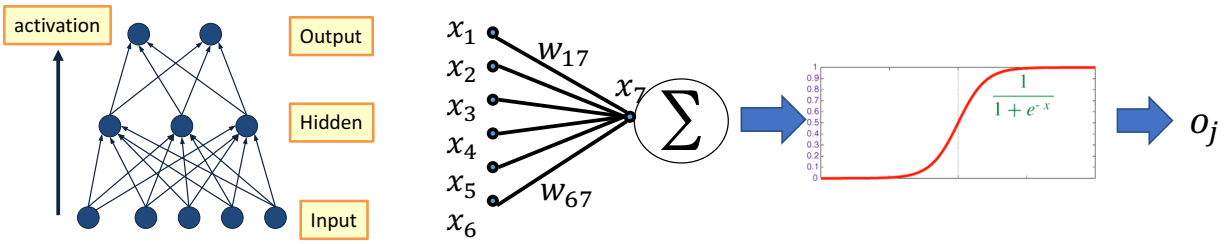


Figure 1: A feedforward network, with a single hidden layer.

- Threshold unit: This is a simpe, and popular choice, and seem to be closer to the natural chioce; but this is not differentiable, hence unsuitable for gradient based techniques for optimizing it.

$$o_j = \mathbf{sign}(\mathbf{w}.\mathbf{x} - T)$$

- Sigmoid unit: Is defined as:

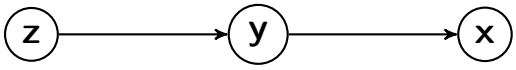
$$o_j = \sigma(\mathbf{w}.\mathbf{x} - T), \text{ where } \sigma(a) = \frac{1}{1+e^{-a}}$$

3 BackPropagatopn Algorithm

The core part of the backpropagation algorithm is using chain rules for calculation of gradients. Here we remind you of these rules:

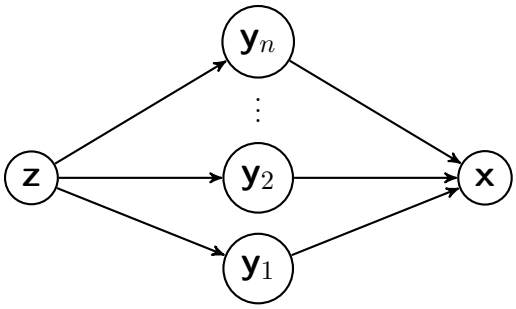
- If z is a function of y, and y is a function of x, then z is a function of x as well. This is how we can write $\frac{\partial z}{\partial x}$:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$



- Multiple path chanin rule

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$



Next we show the derivation of the Backpropagation algorithm. Here we will mostly follow the derivation for the simplified netwrok shown in Figure 2; but extending the ideas to a general netwrok is trivial.

The starting point is defining the error function for the output of the network:

$$\text{Err}(\mathbf{w}) = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$$

where $\mathbf{w}$ the vector which contains all the weights in the network. In our notation o_k is the predicted value for the kth output unit (Figure 2), and t_k is the corresponding gold output. Here is how the output value for the kth unit can be predicted:

$$o_k = \sigma(\text{net}_k - T_k), \quad \text{net}_k = \sum_j w_{jk}.x_j$$

Note that the same rule is used to predict intermediate units. The value of each intermediate unit x_j is predicted as:

$$x_j = \sigma(\text{net}_j - T_j), \quad \text{net}_j = \sum_i w_{ij}.x_i$$

The starting point is finding the update rules for the outer-most layer:

$$\frac{\partial \text{Err}(\mathbf{w})}{\partial w_{jk}} = \frac{\partial \text{Err}(\mathbf{w})}{\partial o_k}\frac{\partial o_k}{\partial w_{jk}} = \frac{\partial \text{Err}(\mathbf{w})}{\partial o_k}\frac{\partial o_k}{\partial \text{net}_k}\frac{\partial \text{net}_k}{\partial w_{jk}}$$

Here is how each of the derivatives simplified:

$$\begin{cases} \frac{\partial \text{Err}(\mathbf{w})}{\partial o_k} = -(t_k - o_k) \\ \frac{\partial o_k}{\partial \text{net}_k} = o_k(1 - o_k) \\ \frac{\partial \text{net}_k}{\partial w_{jk}} = x_j \end{cases}$$

Hence:

$$\frac{\partial \text{Err}(\mathbf{w})}{\partial w_{jk}} = -(t_k - o_k)o_k(1 - o_k)x_j$$

Just for simplicity of notation in future steps, define:

$$\delta_k = (t_k - o_k)o_k(1 - o_k) \tag{1}$$

and the update rules the outermost weights can be written as

$$w_{jk} \leftarrow w_{jk} - \delta_k x_j, \quad \forall j, k$$

Now we go one level lower, and derive the update rule for $\frac{\partial \text{Err}(\mathbf{w})}{\partial w_{jij}}$:

$$\frac{\partial \text{Err}(\mathbf{w})}{\partial w_{ij}} = \sum_{k \in \text{downstream}(j)} \frac{\partial \text{Err}(\mathbf{w})}{\partial o_k}\frac{\partial o_k}{\partial w_{ij}} = \left[\sum_{k \in \text{downstream}(j)} \frac{\partial \text{Err}(\mathbf{w})}{\partial o_k}\frac{\partial o_k}{\partial \text{net}_k}\frac{\partial \text{net}_k}{\partial x_j}\right]\frac{\partial x_j}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}}$$

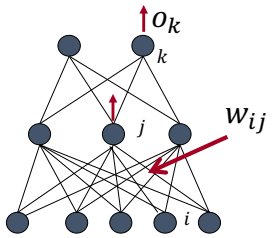


Figure 2: A sample two-layer network: o_k is the output unit, x_j and x_i are intermediate and input units, respectively. The weight connecting these two units is $\mathbf{w}_{ij}$.

Here is what we know about the derivatives:

$$\begin{cases} \frac{\partial \text{Err}(\mathbf{w})}{\partial o_k} = -(t_k - o_k) \\ \frac{\partial o_k}{\partial \text{net}_k} = o_k(1 - o_k) \\ \frac{\partial \text{net}_k}{\partial x_j} = w_{jk} \\ \frac{\partial x_j}{\partial \text{net}_j} = x_j(1 - x_j) \\ \frac{\partial \text{net}_j}{\partial w_{ij}} = x_i \end{cases}$$

which can be simplified as

$$\frac{\partial \text{Err}(\mathbf{w})}{\partial w_{ij}} = -\left[\sum_{k \in \text{downstream}(j)} \delta_k w_{jk}\right] x_j(1 - x_j)x_i$$

And for simplicity of notation in future, define:

$$\delta_j = x_j(1 - x_j) \sum_{k \in \text{downstream}(j)} \delta_k w_{jk}$$

and the update rules the outermost weights can be written as

$$w_{ij} \leftarrow w_{ij} - \delta_j x_i, \quad \forall i, k$$

Note that in the calculation of δ_j, we need only the downstream δ_k's. I can verify that this is the update rule for all the intermediate layers.

Algorithm 1: The backpropagation algorithm

Data: A multi-layer network, with weights randomly initialized.
Result: Weights of the network trained
while *stopping criterion not met* **do**
  for *each training instance* **do**
    Forward propagation (prediction):
    Compute the network output for this example;
    Backward propagation (weight updates):
    Copute the errror between th eoutput and target value:

$$\delta_k = (t_k - o_k)\, o_k(1 - o_k)$$

    For each output unit k, compute the error term:

$$\delta_j = x_j(1 - x_j) \sum_{k \in \text{downstream}(j)} -\delta_k w_{jk}$$

    For each weight w_{ij} update the weight using the calculated δ_j's:

$$w_{ij} \leftarrow w_{ij} - \delta_j x_i$$

    Update the network weights.

4 Expressive power of neural networks

Define the space of family of neural networks $\mathcal{H}_{V,E,\sigma}$:

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \text{ is a mapping from } E \text{ to } \mathbb{R}\}$$

where V and E are nodes and edges, respectively, and denote the wirings in the network. $\sigma : \mathbb{R} \to \mathbb{R}$ is the nonlinearity function, e.g. the sigmoid function in the previous setting. $w : E \to \mathbb{R}$ denote the weight of the edges.

We first start by showing a relatively weak claim on representability of all the binary functions, by a two layer network.

Theorem 4.1. *For every n there exists a graph $G(V, E)$ such that $\mathcal{H}_{V,E,sign}$ contains all the functions from $\{\pm 1\}^n$ to $\{\pm 1\}$.*

Proof. We will give a constructive way on how to construct a member from the family of functions $\mathcal{H}_{V,E,sign}$ which simulates our desired boolean function. First note that if two input vectors $\mathbf{x}$ and $\mathbf{y}$ are not identical their dot product $\langle \mathbf{x}, \mathbf{y} \rangle \leq n - 2$. Define $\mathcal{P}$ to be the set of positive instances, i.e. the set of inputs vectors that have output value of 1. For any $\mathbf{u} \in \mathcal{P}$ define $g(\mathbf{x}; \mathbf{u}) = \text{sign}\left(\langle \mathbf{x}, \mathbf{u} \rangle - (n - 1)\right)$. This function is one, if $\mathbf{x} = \mathbf{u}$, and zero otherwise. To create a conjunction of such functions for any $\mathbf{u} \in \mathcal{P}$:

$$f(\mathbf{x}) = \text{sign}\left(\sum_{\mathbf{u} \in \mathcal{P}} g(\mathbf{x}; \mathbf{u}) + k - 1\right)$$

■

One important point to notice in the previous proof is that, the number of hidden units is exponential in the size of the Boolean function. This is not an artifact of the proof, but a fundamental limitation:

Theorem 4.2. *Suppose $G(V, E)$ is the graph with minimal number of nodes $|V|$ such that $\mathcal{H}_{V,E,sign}$ contains all the Boolean functions with n variables $\{\pm 1\}^n$ to $\{\pm 1\}$. Then $|V|$ is exponential in n.*

Before moving to proof, we note that similar results hold when using sigmoid functions $\mathcal{H}_{V,E,\sigma}$, and the proof is similar to the one below.

Proof. We will show that the VC dimention of neural netwrosk with sign nonlinearity is $O(|E| \log |E|)$ which is less than $O(|V|^3)$. Since $\mathcal{H}_{V,E,sign}$ approximates all Boolean functions with n variables, the VC-dimension of this class is at least $\Omega\left(2^n\right)$. Hence $|V| \in \Omega\left(2^{n/3}\right)$. ■

In a more general work, Bartlett [5] shows that one can derive a more general bound for VC-dimension (and hence for sample comlpexity) that holds for any nonlinearity, and the it depends on the size of the weights, and not the function itself.

Rather than asking what NNs can express *all* the functions, we ask the inverse. What functions can NNs of polynomial size can express? The following theorem says that any Boolean function calculable in time $O\left(\left(\right) T(n)\right)$ can be approximated with a network of size $O\left(\left(\right) T^2(n)\right)$.

Theorem 4.3. *Let $T : \mathbb{N} \to \mathbb{N}$ and $\mathcal{F}_n$ be a set of function that can be implemented with a Turing machine using a runtime of $T(n)$ for every n. Then there exsits contants $b, c\mathbb{R}_+$ such that for every n, there exists a graph $G(V_n, E_n)$ of size at most $cT(n)^2 + b$ such that $\mathcal{H}_{V_n,E_n,sign}$ contains $\mathcal{F}_n$.*

The followin

Theorem 4.4 (NNs as Universal Approximators)**.** *Let $f : [-1, 1]^n \to [-1, 1]$ be a ρ-Lipchitz function. For some fixed $\epsilon > 0$, we can construct a network $N : [-1, 1]^n \to [-1, 1]$, with sigmoid activatin function,*

$$|f(\mathbf{x}) - N(\mathbf{x})| \leq \epsilon, \quad \forall \mathbf{x} \in [-1, 1]^n$$

Proof. In the proof, we partition $[-1, 1]^n$ into small blocks and using the Lipchitzness of f we approximate it inside each block. ■

Although NNs can be universal approximators, but their size can be exponential in terms of input:

Theorem 4.5. *For every n, let $s(n)$ the minimal integer such that there exists a graph $G(V, E)$ with $|V| = s(n)$ such that the hypothesis class $\mathcal{H}_{V,E,\sigma}$ can approximate within an arbitraty $\epsilon \in (0, 1)$, for every 1-Lipchitz function: $f : [-1, 1]^n \to [-1, 1]$. Then $s(n)$ is exponential in n.*

Proof. TODO ■

5 Sample complexity of Neural Networks

One way of quantifying sample complexity is using VC-dimentions * (in additon to other ways, e.g. covering numbers, Rademacher complexity, etc). VC-dimension quantify the complexity of the function class, which are the functions of $|E|$ and $|V|$. Before jumping into the theorems, a couple of useful lemmas:

*More discussion: http://khashab2.web.engr.illinois.edu/learn/vc.pdf

Lemma 5.1 (Growth function of product). *Suppose $\mathcal{F}_1$ be the set of functions from $\mathcal{X}$ to $\mathcal{Y}_1$. And similarly $\mathcal{F}_2$ be the set of functions from $\mathcal{X}$ to $\mathcal{Y}_2$. Define the Cartesian product to be $\mathcal{H} = \mathcal{F}_1 \times \mathcal{F}_2$, that is for any $h \in \mathcal{H}$, there exists two functions $f_1 \in \mathcal{F}_1$ and $f_2 \in \mathcal{F}_2$ such that $h(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x})), \forall \mathbf{x}$. Then $\Pi_{\mathcal{H}}(n) \leq \Pi_{\mathcal{F}_1}(n) \times \Pi_{\mathcal{F}_2}(n)$.*

Lemma 5.2 (Growth function of composition). *Suppose $\mathcal{F}_1$ be the set of functions from $\mathcal{X}$ to $\mathcal{Y}$. And similarly $\mathcal{F}_2$ be the set of functions from $\mathcal{Y}$ to $\mathcal{Z}$. Define the Cartesian product to be $\mathcal{H} = \mathcal{F}_1 \circ \mathcal{F}_2$, that is for any $h \in \mathcal{H}$, there exists two functions $f_1 \in \mathcal{F}_1$ and $f_2 \in \mathcal{F}_2$ such that $h(\mathbf{x}) = f_1(f_2(\mathbf{x})), \forall \mathbf{x}$. Then $\Pi_{\mathcal{H}}(n) \leq \Pi_{\mathcal{F}_1}(n) \times \Pi_{\mathcal{F}_2}(n)$.*

Theorem 5.3. *VC dimention of $\mathcal{H}_{V,E,sign}$ is $O(|E| \log |E|)$.*

Proof. 1 ■

6 Learning neural networks

For most of the common learning models, ERM is NP-hard.

Theorem 6.1. *For every $G(V, E)$, with n input nodes, $k + 1$ nodes at the (single) hidden layer, where one of them is the constant neuron and a single-output node. Then it is NP-hard to perform ERM rule with respect to $\mathcal{H}_{V,E,\sigma} V, E, sign$.*

Proof. 1 ■

7 Bibliographical notes

The backpropagation is introduced in [1] [†] The expressive power of neural networks and the connection to the circuit complexity has been studied in [3, 4]. Analysis of sample complexity is given in [5].

8 Excercise problems

8.1 true of false

- Backpropagation algorithm always achieve the optimal solution
- Number of nodes in hidden layer can control generalization
- Weights in neural networks have intuitive meaning
- Neural networks can be used for interpolating a function.
- Convergance is guaranteed in Backpropagation algorithm.

[†] Debatable. See section 5.5 of [2] for discussion.

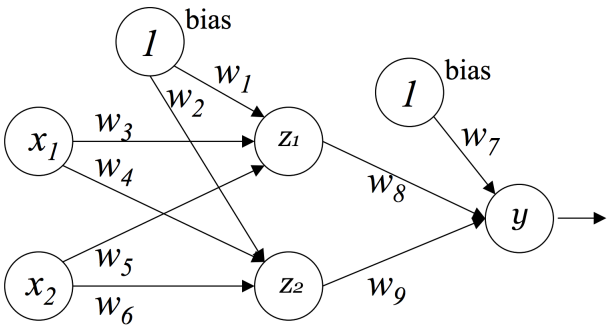


Figure 3: A two-layer neural network.

8.2 Backpropagation

Consider a neural net for a binary classification which has one hidden layer as shown in the Figure Figure 3. We use a linear activation function $h(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ at the hidden units, and a sigmoid activation function $g(\mathbf{z}) = \frac{1}{1+e^{\mathbf{w}^\top \mathbf{z}}}$ at the output unit.

- Find the value of the hidden variables z_1 and z_2, given x_1 and x_2.
- Find the value of the output variable y, given hidden variables y_1 and y_2.
- For fixed weights, under what input values, the network will predict $+1$ in the output?
- Define an error function E to be squared loss (the error between the predictions and target values). Find the gradient of the error with respect to the weights incoming to the output layer, i.e. $\frac{\partial E}{\partial w_i}$, for $w_i \in \{w_7, w_8, w_9\}$.
- Find the gradient of the error with respect to the weights incoming to the hidden layer, i.e. $\frac{\partial E}{\partial w_i}$, for $w_i \in \{w_1, w_2, w_3, w_4, w_5, w_6\}$.
- Given a training instance $((\hat{x}_1, \hat{x}_2), \hat{y})$, what are the Backpropagation update rules for one iteration using this training instance.

References

[1] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[2] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[3] John S Shawe-Taylor, Martin HG Anthony, and Walter Kern. Classes of feedforward neural networks and their circuit complexity. *Neural Networks*, 5(6):971–977, 1992.

[4] Ian Parberry. *Circuit complexity and neural networks*. MIT press, 1994.

[5] Peter L Bartlett. The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network. *IEEE transactions on Information Theory*, 44(2):525–536, 1998.