Using Different Cost Functions to Train Deep Networks with Supervision

Telmo Amaral

30th July 2013

NNIG Technical Report No. 3/2013

Project "Reusable Deep Neural Networks: Applications to Biomedical Data"
(PDTC/EIA-EIA/119004/2010)

Neural Networks Interest Group

Instituto de Engenharia Biomédica (INEB)
Rua Dr. Roberto Frias, 4200-465, Porto, Portugal

Contents

1 Introduction **3**

2 Generic case **3**
- 2.1 Output weight . . . 5
- 2.2 Hidden weight . . . 5
- 2.3 Summary . . . 7

3 Specific case: CE **7**
- 3.1 Output weight . . . 8
- 3.2 Hidden weight . . . 9

4 Specific case: EXP **10**

5 Specific case: SSE **11**

6 Paper overview: Using different cost functions to train stacked auto-encoders **12**

References **13**

1 Introduction

In a previous report [1] we presented detailed calculations for the derivatives of three error functions that can be used when pre-training, without supervision, each hidden layer of a deep neural network (by unfolding the layer as an auto-encoder). In this report we complement those calculations, by offering detailed calculations for the derivatives of three error functions that can be used when training, with supervision, an entire deep network (for example as a fine-tuning stage, following an unsupervised pre-training stage).

In Section 2 we obtain generic expressions for an error's partial derivative in order to a given network parameter, which can be an output weight or bias, or a hidden weight or bias. These expressions are generic in the sense that they are valid for any number of layers, cost function, hidden activation function, and output activation function. In Sections 3, 4 and 5 we refine the previously obtained expressions for three specific cost functions: cross-entropy, exponential, and sum of squared errors.

Section 6 presents an overview of a paper [2] presented at the Mexican International Conference on Artificial Intelligence (MICAI 2013). In this paper we describe experiments involving different combinations of pre-training and fine-tuning cost functions.

2 Generic case

Our goal is to obtain an expression for the partial derivative $\partial E_x / \partial w_{ji}$, where E_x is the error associated with a particular input vector and w_{ji} is the weight of the connection between unit i on one layer of a neural network and unit j on the next layer. So, this section presents calculations similar to those contained in Section 5.3.1 of Bishop [5], but here all intermediate steps are included and the calculations are kept entirely generic, that is, independent of E_x and independent of the activation functions used in the output layer and in the hidden layers.

The diagram in Fig. 1 illustrates the notation used, showing some of the units belonging to two consecutive layers of a network (which may have any number of layers). The values denoted as a and z are calculated via the forward propagation expressions, where $h(.)$ denotes an activation function:

$$a_j = \sum_i w_{ji} z_i \tag{1}$$

$$z_j = h(a_1, a_2, ..., a_j, ...) \tag{2}$$

Noting that E_x "depends on the weight w_{ji} only via the summed input a_j to unit j", we can express E_x as the function of a function:

$$E_x = E_x(a_j(w_{ji})) \tag{3}$$

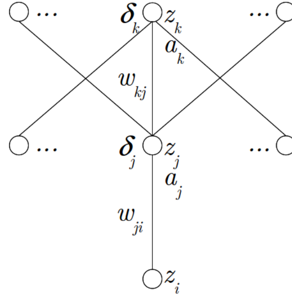

Figure 1: Illustration of the notation used, showing some of the units belonging to two consecutive layers of a network (Similar to Fig. 5.7 in Bishop [5]).

So,

$$\frac{\partial E_x}{\partial w_{ji}} = \frac{\partial E_x}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{4}$$

We define

$$\delta_j \equiv \frac{\partial E_x}{\partial a_j} \tag{5}$$

So we can write

$$\frac{\partial E_x}{\partial w_{ji}} = \delta_j \frac{\partial a_j}{\partial w_{ji}} \tag{6}$$

This expression is valid for any weight, including any output weight, in a network with any number of layers.

Using (1) we can write

$$\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_n w_{jn} z_n \tag{7}$$

$$= \sum_n \frac{\partial}{\partial w_{ji}} w_{jn} z_n \tag{8}$$

$$= \sum_n \left(w_{jn} \frac{\partial z_n}{\partial w_{ji}} + \frac{\partial w_{jn}}{\partial w_{ji}} z_n\right) \tag{9}$$

Noting that

$$\frac{\partial z_n}{\partial w_{ji}} = 0 \tag{10}$$

And

$$\frac{\partial w_{jn}}{\partial w_{ji}} = \begin{cases} 0 & \text{if } n \neq i \\ 1 & \text{if } n = i \end{cases} \tag{11}$$

We arrive at

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \tag{12}$$

And finally

$$\boxed{\frac{\partial E_x}{\partial w_{ji}} = \delta_j z_i} \tag{13}$$

We can obtain the partial derivative in order to a *bias* b_j (hidden or output) simply by setting z_i to 1:

$$\boxed{\frac{\partial E_x}{\partial b_j} = \delta_j} \tag{14}$$

2.1 Output weight

If w_{ji} is an output weight, we just use the definition:

$$\boxed{\delta_j \equiv \frac{\partial E_x}{\partial a_j}} \tag{15}$$

2.2 Hidden weight

If w_{ji} is a hidden weight, we note that "variations in a_j give rise to variations in the error only through variations in the variables a_k", so

$$E_x = \sum_k E_x(a_k(a_j)) \tag{16}$$

So we can write

$$\delta_j \equiv \frac{\partial E_x}{\partial a_j} \tag{17}$$

$$= \frac{\partial}{\partial a_j} \sum_k E_x(a_k(a_j)) \tag{18}$$

$$= \sum_k \frac{\partial}{\partial a_j} E_x(a_k(a_j)) \tag{19}$$

$$= \sum_k \frac{\partial E_x}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{20}$$

$$= \sum_k \delta_k \frac{\partial a_k}{\partial a_j} \tag{21}$$

From Fig. 1 and (1),

$$a_k = \sum_n w_{kn} z_n \tag{22}$$

So

$$\frac{\partial a_k}{\partial a_j} = \frac{\partial}{\partial a_j} \sum_n w_{kn} z_n \tag{23}$$

$$= \sum_n (w_{kn} \frac{\partial z_n}{\partial a_j} + \frac{\partial w_{kn}}{\partial a_j} z_n) \tag{24}$$

Noting that, when $n \neq j$,

$$\frac{\partial z_n}{\partial a_j} = 0 \tag{25}$$

And

$$\frac{\partial w_{kn}}{\partial a_j} = 0 \tag{26}$$

We arrive at

$$\frac{\partial a_k}{\partial a_j} = w_{kj} \frac{\partial z_j}{\partial a_j} \tag{27}$$

We can assume that the activation function used in a hidden layer takes only one parameter. That is, we can simplify expression (2):

$$\begin{aligned} z_j =& h(a_1, a_2, ..., a_j, ...) \\ =& h(a_j) \end{aligned} \tag{28}$$

So

$$\frac{\partial a_k}{\partial a_j} = w_{kj} \frac{\partial h(a_j)}{\partial a_k} \tag{29}$$

$$= w_{kj} h'(a_j) \tag{30}$$

And finally,

$$\delta_j = \sum_k \delta_k w_{kj} h'(a_j) \tag{31}$$

Or

$$\boxed{\delta_j = h'(a_j) \sum_k \delta_k w_{kj}} \tag{32}$$

Note that the above expression for δ_j depends on δ_k, which is associated with the layer above (see Fig. 1). In practice, the δ values are computed starting from the output layer, in back-propagation.

2.3 Summary

Given a specific case of error function E_x and hidden activation function $h(.)$, we only need to calculate two things:

1. $\partial E_x / \partial a_j$ in (15) (which includes a derivation of the *output* activation function, typically softmax); and
2. $h'(a_j)$ in (32).

3 Specific case: CE

Lets consider the specific case where the hidden activation function $h(.)$ is the logistic sigmoid, the output activation function is softmax, and the "multi-class" version of the cross-entropy (CE) error function is used:

$$E_x(\mathbf{y}, \mathbf{t}) = -\sum_k t_k \ln y_k \tag{33}$$

E_x denotes the error associated with a given input vector; $\mathbf{y}$ is the network's output vector and $\mathbf{t}$ is the ground-truth vector. y_k is the kth element of the output vector, that is, the posterior probability output for the kth class. t_k is the kth element of the ground-truth vector; it is 1 if the true class associated with the input vector is the kth class, and 0 otherwise.

3.1 Output weight

If w_{ji} is an output weight, we need to calculate

$$\delta_j \equiv \frac{\partial E_x}{\partial a_j} \tag{34}$$

$$= -\sum_k (t_k \frac{\partial}{\partial a_j} \ln y_k + \frac{\partial t_k}{\partial a_j} \ln y_k) \tag{35}$$

Noting that

$$\frac{\partial t_k}{\partial a_j} = 0 \tag{36}$$

We can write

$$\delta_j = -\sum_k t_k \frac{\partial}{\partial y_k} \ln y_k \frac{\partial y_k}{\partial a_j} \tag{37}$$

$$= -\sum_k t_k \frac{1}{y_k} \frac{\partial y_k}{\partial a_j} \tag{38}$$

Now we note that

$$y_k = \text{softmax}_k(\mathbf{a}) \tag{39}$$

So, using I_{kj} to denote element kj of an identity matrix,

$$\frac{\partial y_k}{\partial a_j} = y_k(I_{kj} - y_j) \tag{40}$$

$$= \begin{cases} y_k(1 - y_j) & \text{if } j = k \\ -y_k y_j & \text{if } j \neq k \end{cases}$$

And

$$\delta_j = -\sum_k t_k \frac{1}{y_k} y_k (I_{kj} - y_j) \tag{41}$$

$$= - t_j(1 - y_j) - \sum_{k \neq j} t_k(-y_j) \tag{42}$$

$$= - t_j + t_j y_j + y_j \sum_{k \neq j} t_k \tag{43}$$

$$= - t_j + y_j \sum_k t_k \tag{44}$$

But, since

$$\sum_k t_k = 1 \tag{45}$$

We arrive at

$$\boxed{\delta_j = y_j - t_j} \tag{46}$$

Note that, using the above expression of δ_j, we obtain this expression for the error derivative:

$$\frac{\partial E_x}{\partial w_{ji}} = \delta_j z_i \tag{47}$$

$$= (y_j - t_j) z_i \tag{48}$$

The above expression is the same as we would obtain for multi-class logistic regression, and corresponds to one term of the sum in Equation 4.109 in Bishop [5] (which shows a sum over input vectors).

3.2 Hidden weight

If w_{ji} is a hidden weight, we need to calculate

$$\delta_j = h'(a_j) \sum_k \delta_k w_{kj} \tag{49}$$

Noting that $h(.)$ is the logistic sigmoid and recalling (28), we can write

$$h'(a_j) = h(a_j)(1 - h(a_j)) \tag{50}$$
$$= z_j(1 - z_j) \tag{51}$$

So

$$\delta_j = z_j(1 - z_j) \sum_k \delta_k w_{kj} \tag{52}$$

We recall that this result is independent from error function E_x; it depends only on the choice of hidden activation function $h(.)$.

4 Specific case: EXP

Lets consider the specific case where the hidden activation function $h(.)$ is the logistic sigmoid, the output activation function is softmax, and the exponential (EXP) error function is used:

$$E_x(\mathbf{y}, \mathbf{t}) = \tau \exp\left(\frac{1}{\tau} \sum_k (y_k - t_k)^2\right) \tag{53}$$

If w_{ji} is an output weight, we need to calculate

$$\delta_j \equiv \frac{\partial E_x}{\partial a_j} \tag{54}$$
$$= \tau \frac{\partial}{\partial v} \exp(v) \frac{\partial}{\partial a_j} \underbrace{\frac{1}{\tau} \sum_k (y_k - t_k)^2}_{v} \tag{55}$$
$$= \tau \exp(v) \frac{1}{\tau} \sum_k \frac{\partial}{\partial a_j} (y_k - t_k)^2 \tag{56}$$
$$= \exp(v) \sum_k 2(y_k - t_k) \frac{\partial}{\partial a_j} (y_k - t_k) \tag{57}$$
$$= \exp(v) \sum_k 2(y_k - t_k)\left(\frac{\partial y_k}{\partial a_j} - \frac{\partial t_k}{\partial a_j}\right) \tag{58}$$

But

$$\frac{\partial t_k}{\partial a_j} = 0 \tag{59}$$

So, using also (40),

$$\delta_j = 2\exp(\frac{1}{\tau}\sum_k (y_k - t_k)^2)\sum_k (y_k - t_k) y_k (I_{kj} - y_j) \tag{60}$$

If w_{ji} is a hidden weight instead, we can just reuse the result obtained in Section 3.2.

5 Specific case: SSE

Lets consider the specific case where the hidden activation function $h(.)$ is the logistic sigmoid, the output activation function is softmax, and the sum of squared errors (SSE) is used as error function:

$$E_x(\mathbf{y}, \mathbf{t}) = \frac{1}{2}\sum_k (y_k - t_k)^2 \tag{61}$$

If w_{ji} is an output weight, we need to calculate

$$\delta_j \equiv \frac{\partial E_x}{\partial a_j} \tag{62}$$

$$= \frac{1}{2}\sum_k \frac{\partial}{\partial a_j}(y_k - t_k)^2 \tag{63}$$

$$= \frac{1}{2}\sum_k 2(y_k - t_k)\frac{\partial}{\partial a_j}(y_k - t_k) \tag{64}$$

$$= \sum_k (y_k - t_k)(\frac{\partial y_k}{\partial a_j} - \frac{\partial t_k}{\partial a_j}) \tag{65}$$

But

$$\frac{\partial t_k}{\partial a_j} = 0 \tag{66}$$

So, using also (40),

$$\delta_j = \sum_k (y_k - t_k) y_k (I_{kj} - y_j) \tag{67}$$

If w_{ji} is a hidden weight instead, we can just reuse the result obtained in Section 3.2.

6 Paper overview: Using different cost functions to train stacked auto-encoders

Deep architectures, such as neural networks with two or more hidden layers of units, are a class of machines that comprise several levels of non-linear operations, each expressed in terms of parameters that can be learned [3]. The organization of the mammal brain, as well as the apparent depth of cognitive processes, are among the main motivations for the use of such architectures. In spite of this, until 2006, attempts to train deep architectures resulted in poorer performance than that achieved by their shallow counterparts. The only exception to this difficulty was the convolutional neural network [9], a specialized architecture for image processing, modeled after the structure of the visual cortex.

A breakthrough took place with the introduction by Hinton *et al.* of the deep belief network [7], a learning approach where the hidden layers of a deep network are initially treated as restricted Boltzmann machines (RBMs) [11] and pre-trained, one at a time, in an unsupervised greedy approach. Given that auto-encoders [6] are easier to train than RBMs, this unsupervised greedy procedure was soon generalized into algorithms that pre-train the hidden levels of a deep network by treating them as a stack of auto-encoders [4, 8].

The auto-encoder (also called auto-associator or Diabolo network) is a type of neural network trained to output a reconstruction of its own input. Thus, in the training of auto-encoders, input vectors can themselves be interpreted as target vectors. This presents an opportunity for the comparison of various training criteria, namely different cost functions capable of reflecting the mismatch between inputs and targets.

The information theoretical concept of minimum error entropy has been recently applied by Marques de Sá *et al.* Marques de Sá et al. [10] to data classification machines, yielding evidence that risk functionals do not perform equally with respect to the attainment of solutions that approximate the minimum probability of error. In their work, the features of classic cost functions such as the sum of squared errors (SSE) and the so-called cross-entropy (CE) cost are discussed, and some approaches inspired by the error entropy concept are proposed. One such approach is a parameterized function called exponential (EXP) cost, sufficiently flexible to emulate the behavior of classic costs, namely SSE and CE, and to exhibit properties that are desirable in certain types of problems, such as good robustness to the presence of outliers.

In this work, we aimed to compare the performances of SSE, CE, and EXP costs when employed both in the unsupervised pre-training and in the supervised fine-tuning of deep networks whose hidden layers are regarded as a stack of auto-encoders. To the best of our knowledge, this type of comparison has not been done before in the context of deep learning. Using a number of artificial and real-world data sets, we first compared pre-training cost functions in terms of their impact on the reconstruction performance of hidden layers. Given that the output layer of our networks was designed for classification learning, we also compared various combinations of pre-training and fine-tuning costs in terms of their impact on classification performance.

In general, the best layer-wise reconstruction performance was achieved by SSE pre-training, though with binary data CE yielded the lowest errors for the first hidden layer. Classification performance was found to vary little with the combination of pre-training and fine-tuning costs. When pre-training with CE, fine-tuning via SSE was found not to be a good choice. In general, the choice of the same pre-training and fine-tuning costs yielded classification errors with lower variance.

With a heavily unbalanced artificial data set, fine-tuning failed except for the cost combination used to tune the model's hyper-parameters and for those cost combinations that involved EXP fine-tuning. This seeming robustness of EXP fine-tuning should be further investigated, using a wider variety data sets. Future work should also focus on improving both the pre-training early stopping mechanism and the stability of fine-tuning after large numbers of iterations. In future experiments we plan to adopt GPU-based processing, to allow the use of more computationally demanding data sets, such as variants of the popular MNIST character recognition set.

References

[1] T. Amaral, L. M. Silva, and L. A. Alexandre. Using different cost functions when pre-training stacked auto-encoders. Technical Report 1/2013, Instituto de Engenharia Biomédica / NNIG, April 2013. 3

[2] T. Amaral, L. M. Silva, L. A. Alexandre, C. Kandaswamy, J. M. Santos, and J. Marques de Sá. Using different cost functions to train stacked auto-encoders. In *Mexican International Conference on Artificial Intelligence (MICAI)*, pages 114–120, 2013. 3

[3] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009. 12

[4] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *Neural Information Processing Systems Conference*, volume 19, pages 153–160, 2007. 12

[5] C. Bishop. *Pattern recognition and machine learning*, volume 1. Springer, 2006. 3, 4, 9

[6] H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59(4-5):291–294, 1988. 12

[7] G. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006. 12

[8] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *International Conference on Machine Learning*, pages 473–480, 2007. 12

[9] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 12

[10] J. Marques de Sá, L. Silva, J. Santos, and L. Alexandre. *Minimum Error Entropy Classification*, volume 420 of *Studies in Computational Intelligence*. Springer, 2013. 12

[11] P. Smolensky. *Parallel distributed processing: explorations in the microstructure of cognition*, volume 1, chapter Information processing in dynamical systems: Foundations of harmony theory, pages 194–281. University of Colorado, 1986. 12