Derivation of the Back-propagation based learning algorithm

Alessio Micheli

Bibliography (examples)

1. Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Parallel Distributed Processing" *MIT Press, Cambridge, MA* (1986) **Chapter 8**.
2. Mitchell, Tom M., "Machine learning". *McGraw Hill* (1996) **Chapter 4**.
3. Haykin, S., "Neural Networks", 2nd Edition. *Prentice Hall* (1998).

Premise

Differential calculus for the gradient descent technique:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x} \quad \text{a.k.a. } \textbf{the Chain Rule} \tag{1}$$

$$D[f(g(x))] = D[f(g(x))] = f'(g(x)) \cdot g'(x) \tag{2}$$

$$D[f \cdot g] = f' \cdot g + g' \cdot f \tag{3}$$

where D stands for the derivative. The chain rule is useful because it decomposes and brings out what can be immediately computed, while keeping the main track.

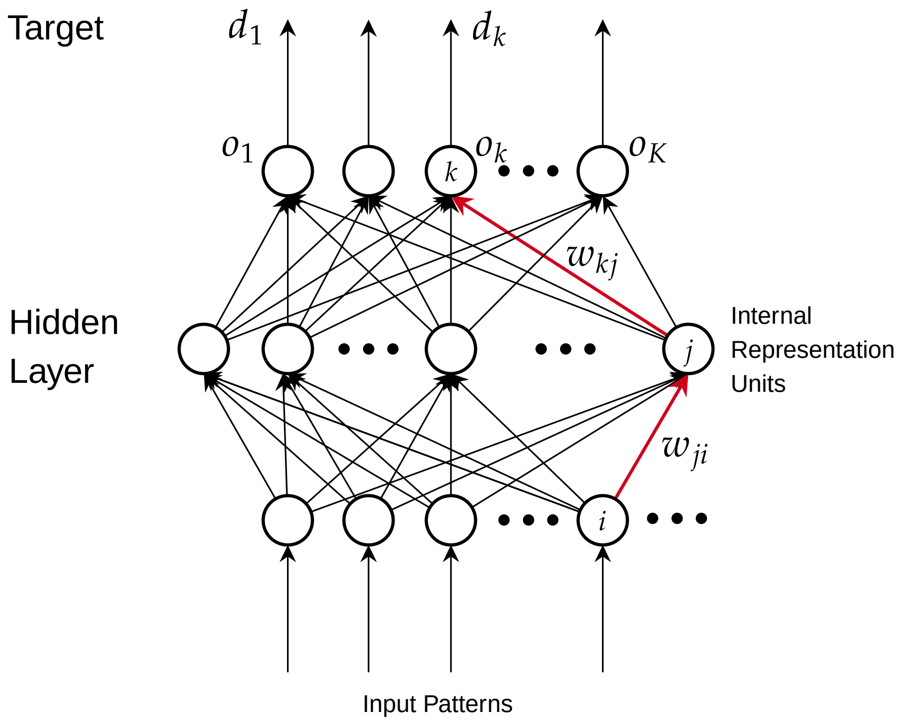
GENERAL SUGGESTIONS:

- Repeat the derivation by yourself (**don't just memorize**): you will have the means to be able to recompute it for different cases of loss/networks/etc. **This is our real full objective** (motivating the effort)
- Use the prj implementation has a further means to fully understand the algorithm, to explore different behaviours, and have fun with this algorithm!

By A. Micheli – Univ. of Pisa. First LaTeX version by F. Errica.

Back-propagation

The basic architecture is a feed-forward fully connected neural network (MLP). We use different indices, k, j and i, as generic indices for the different layers of the MLP.



The problem: to estimate the contribution of hidden units to the error at the output level. We do this by computing the Generalized Delta Rule.

Training Set (TR) $=$ $\{(\mathbf{x}^1, d^1), \ldots, (\mathbf{x}^l, d^l)\} \rightarrow$ Supervised Learning.
Each input unit i is loaded with the input x_i (component i of the input vector $\mathbf{x}$), hence $o_i = x_i$ per each pattern $p = 1 ... l$.

Total Error (of the network on the TR set): $E_{tot} = \sum_p E_p$, where $E_p = \frac{1}{2}\sum_{k=1}^{K}(d_k - o_k)^2$ is computed for the input pattern p for all the K units.
Note that, in order *to simplify*, p and x_p (or o_p) are and will be omitted in the following. E.g. o_k is for $o_k(\mathbf{x}_p)$ and d_k is for $d_{p,k}$. Above, for this reason, the patterns in the TR set were denoted in a special form as $(\mathbf{x}^p, d^p)$ without using the subscript form for $p = 1 ... l$.

Objective: find the values of parameters $\mathbf{w}$ that **minimize** E_{tot} (we first focus on the data term).

Least Mean Square (LMS): the minimization of the above Total Error.

Idea: Perform a descent on the surface of E_{tot} on the basis of the gradient (Figure 1).

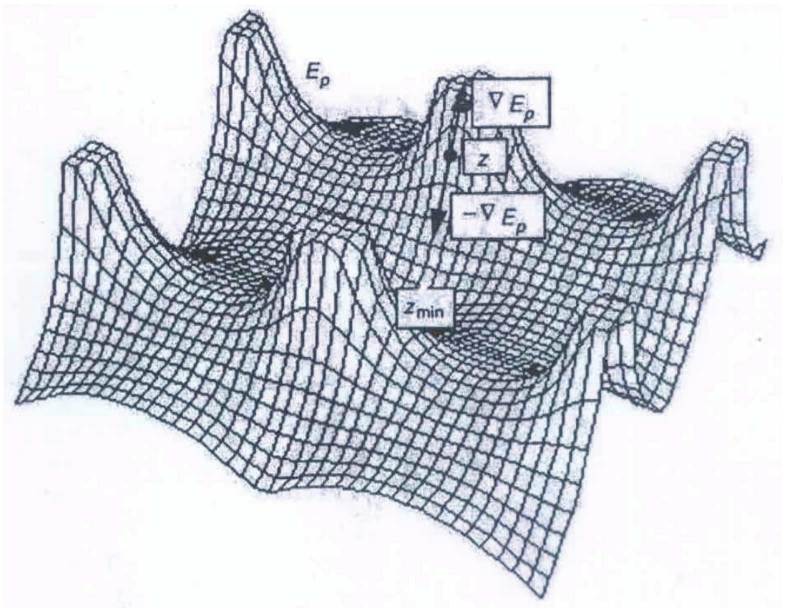


Figure 1: Example of E_{tot} landscape in the space of two weights (w_1 and w_2); the local gradient is shown in the point Z. Along the direction of ($-$gradient) we can reach the Z_{min} point.

Iterative gradient descent training algorithm

Description: Compute the loss (error) function (i.e. E_{tot}) $\rightarrow$ compute the gradient $\rightarrow$ update all the weights $\mathbf{w}$ in the network. Then repeat until convergence or until other stopping criteria.

Algorithm 1: Back-propagation (the same presented in general for the LMS)

```
Initialize all the weights $\mathbf{w}$ in the network and $\eta$;
Compute units outputs and $E_{tot}$;
while $E_{tot} > \epsilon$ (desired value or other criteria) do
    $\forall w \in \mathbf{w}$: $\Delta w = -\frac{\partial E_{tot}}{\partial w}$ (Step 1);
              $w^{new} \leftarrow w + \eta \Delta w + \dots$ (Step 2) ;
    Compute units outputs and $E_{tot}$;
end
```

Main Back-propagation Version: stochastic, batch

Distinguish between:

1. Gradient computation $\rightarrow \frac{\partial E_{tot}}{\partial \mathbf{w}}$ (we now compute this for each weight in the network)
2. Update rule for $\mathbf{w}$: Standard Back-prop, Quick-prop, R-prop, ...

We focus on (Step 1)

$$\Delta \mathbf{w} = -\frac{\partial E_{tot}}{\partial \mathbf{w}} = -\sum_p \frac{\partial E_p}{\partial \mathbf{w}} \overset{def.}{=} \sum_p \Delta_p \mathbf{w}$$

whereas p is p-th pattern to be fed as input to the neural network.

Then, for w_{tu} of a generic unit t, pattern p, and the input u from a generic unit u to the unit t, we have:

$$\Delta_p w_{tu} = -\frac{\partial E_p}{\partial w_{tu}} = \boxed{-\frac{\partial E_p}{\partial net_t}} \cdot \boxed{\frac{\partial net_t}{\partial w_{tu}}} = \boxed{\delta_t} \cdot \boxed{o_u}$$

(where w_{tu} connects t with the input from a generic unit u, either i or j), since

$$\begin{cases} net_t = \sum_s w_{ts} o_s \\ o_t = f_t(net_t) \end{cases} \implies \boxed{\frac{\partial net_t}{\partial w_{tu}}} = \frac{\partial \sum_s w_{ts} o_s}{\partial w_{tu}} = \boxed{o_u}$$ (all 0 but for $s = u$, note that o_s are inputs to the unit t)

Now we expand on $\boxed{\delta_t}$ (the delta of a unit t):

$$\boxed{\delta_t} = -\frac{\partial E_p}{\partial net_t} = \boxed{-\frac{\partial E_p}{\partial o_t}} \cdot \frac{\partial o_t}{\partial net_t} \overset{o_t = f_t(net_t)}{=} \boxed{-\frac{\partial E_p}{\partial o_t}} \cdot f_t'(net_t).$$

Now recall that $E_p = \frac{1}{2}\sum_{k=1}^{K}(d_k - o_k)^2$. We distinguish 2 cases, depending on whether o_t is an output unit k (Case 1) or an hidden unit j (Case 2).

Case 1: Output Unit ($t = k$, see the exercise of previous lecture)

$$\boxed{-\frac{\partial E_p}{\partial o_k}} = -\frac{\partial \frac{1}{2}\sum_{r=1}^{K}(d_r - o_r)^2}{\partial o_k} = (d_k - o_k) \implies \boxed{\delta_k} = -\frac{\partial E_p}{\partial net_k} = \boxed{(d_k - o_k)} \cdot f_k'(net_k)$$

Case 2: Hidden Unit ($t = j$)

$$\boxed{-\frac{\partial E_p}{\partial o_j}} = \sum_{k=1}^{K} -\frac{\partial E_p}{\partial net_k} \cdot \frac{\partial net_k}{\partial o_j} = \sum_{k=1}^{K} \delta_k \cdot w_{kj}$$

$$\text{since} \quad \frac{\partial net_k}{\partial o_j} = \frac{\partial \sum_s w_{ks} \cdot o_s}{\partial o_j} = w_{kj}$$

$$\implies \boxed{\delta_j} = \boxed{\left(\sum_{k=1}^{K} \delta_k \cdot w_{kj}\right)} \cdot f_j'(net_j)$$

Crucial: We have **already computed** $-\frac{\partial E_p}{\partial net_k}$ before! (see the def. of delta above with $t = k$). We can back-propagate the signals δ_k to the hidden unit j!

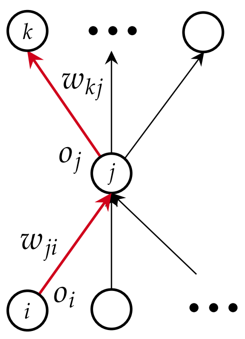


Figure 2: HINT: Use the network graph to localize the computation! (here by indices)

Notes on Case 2:

- It expresses the variation of E_p considering all the output units o_k
- Each o_k (and net_k) depends on o_j, hence we introduce a sum over k (see Fig. 2)
- This step can be obtained directly, using the def. of E_P (exercise), but in this form is more instructive (by computing the partial derivatives flow over the network, it shows the propagation over the provided network)

Summary

We derived, for two layers,

$$\Delta_p w_{tu} = \boxed{\delta_t} \cdot o_u$$

where δ_t is the error signal available to the unit t, and o_u is the input to t from a generic unit u, i.e. the input through the connection weighted by w_{tu}. In particular:

$$\begin{aligned} &\Delta_p w_{kj} = \delta_k \cdot o_j \quad \text{if } t = k \text{ (output unit) and the input is from unit } j \\ &\Delta_p w_{ji} = \delta_j \cdot o_i \quad \text{if } t = j \text{ (hidden unit), and input } i \end{aligned}$$

The other two equations specify the error signals:

$$\begin{aligned} \boxed{\delta_k} &= (d_k - o_k) \ \cdot \ f_k'(net_k) \qquad \text{if } t = k \text{ (output unit)} \\ \boxed{\delta_j} &= \Big(\sum_{k=1}^{K} \delta_k w_{kj}\Big) \cdot \ f_j'(net_j) \quad \text{if } t = j \text{ (hidden unit)} \end{aligned}$$

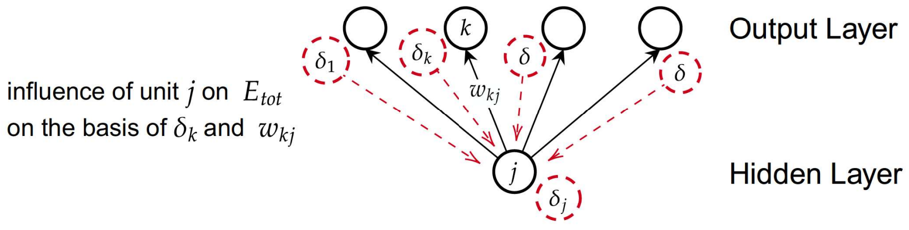


Figure 3: Retropropagation of delta (δ) values from the output layer to obtain the error signal for the hidden layers. This explains the name ”Backpropagation”.

This derivation can be applied (generalized) to m hidden layers! In fact, the deltas come not only from the output layer but also from any generic layer above the current one (a generic upper layer h). More in general, the deltas come from any units to which the current unit is connected to.

Hence, for each pattern p (again, omitting p):

$$w_{tu}^{new} = w_{tu} + \eta \cdot \delta_t \cdot o_u$$

where δ_t is the delta at **that** level for t and o_u is the input to the unit t from u through the connections w_{tu}. In this way we are using the $\Delta_p \mathbf{w}$ for each pattern p, and we have the on-line version; for the batch version we have to sum over the patterns (see $\Delta \mathbf{w}$ and Alg. 1.)

Further comments (and repetita):

- Summarizing the entire **training cycle** (as sketched in the Algorithm 1) we have (in order):
  - forward computation (computation of units outputs);
  - computation of errors and delta at output layer level;
  - propagation of the deltas backwards from the output layer to the hidden layers;
  - updating of the weights;
  - and we can restart again up to reach a criteria for stopping the training.
- The computation and the update must be applied also to all the **bias terms** of the units: We assumed the first input component of each unit =1, and the bias term present as w_{t0}.
- We compute only **local values**. We directly exploit only information around the unit (below or over), and not from far layers not connected to the unit, while the other information are received indirectly (deltas flow).
  Or in other words: Despite local computation, the changes of weight values have a global effect on the network. What is the influence of each w_{tu} on the total (global network) error? We have decomposed this complex effect in simpler components through the concept of the delta values local to each units.
- **Diffusion process** point of view: Backprogation as a local spatial temporal propagation from unit's parents to the unit and from the unit to unit's children (toward Hamiltonian view).
- **Efficiency**: "magic factorization" due the computation of deltas for each unit, which can be used for all the weights of that units. For all the w_{tu}, we use the same δ_t, computed just one time for all them (see the equations above) and not for each couple (t, u). Hence, we do not have a cost proportional to the square of number of weights, but to the number of the weights (check by yourself) .
  With models using bilions of weights this makes a huge difference: e.g. 10^9 versus $(10^9)^2 = 10^{18}$ operations (1 trillion, not feasible) !!!

General (didatics) Notes:

- Read the derivation at 3 levels: (i) singular math steps, (ii) interpretation of the (local) changes of a quantity with respect to the others (due to the partial derivatives) (iii) general framework of the meaning of the provided decomposition, i.e. to find delta values for each level decomposing the delta error ON THE NETWORK at hand.
- Don't say "backprop NN", NN is a model. Instead, this is a training algorithm based on the backpropagation of errors, related to the direct computing of the gradient through the network.

Suggestions:

- Repeat the derivation by yourself (don't just memorize).