# Training Convolutional Neural Nets

Carlo Tomasi

October 17, 2017

Consider a deep neural net with $L$ layers indexed by $\ell = 1, \ldots, L$ and a training set of input-output pairs

$$T = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\} \; .$$

The outputs $\mathbf{y}_n$ are scalars in a classification problem or possibly vectors, either for regression or when the deep net is used to generate a feature vector.

The supervised training of this net starts by setting the entries in the weight matrices $W^\ell$, which are collectively called the *parameters* of the net, to fit[1] the input-output pairs in $T$. *Fitting* in turn involves minimizing the training error

$$E(\mathbf{w}) = \overline{\mathrm{err}}(\mathbf{w}, T) = \frac{1}{N}\sum_{n=1}^{N} E_n(\mathbf{w}) \quad \text{where} \quad E_n(\mathbf{w}) = \mathcal{L}(\mathbf{y}_n, \mathbf{f}(\mathbf{x}_n, \mathbf{w})) \tag{1}$$

In this expression, the loss function $\mathcal{L}$ depends on the type of problem addressed, the (possibly vector-valued) function $\mathbf{f}$ represents the computation performed by the net, and the vector $\mathbf{w}$ collects all the weight-matrix entries. More specifically,

$$\mathbf{w} = \begin{bmatrix} \mathbf{w}^{(1)} \\ \vdots \\ \mathbf{w}^{(L)} \end{bmatrix}$$

where

$$\mathbf{w}^{(\ell)} = [W_{11}^{(\ell)}, W_{21}^{(\ell)}, \ldots, W_{D^{(\ell)}, D^{(\ell-1)}+1}^{(\ell)}]^T$$

denotes the entries of the matrix $W^{(\ell)}$ arranged into a column vector and has length

$$J^{(\ell)} = D^{(\ell)}(D^{(\ell-1)} + 1) \; .$$

The parameter vector $\mathbf{w}$ is fit to the training set by an iterative procedure that starts with some *initial values* $\mathbf{w}_0$ for $\mathbf{w}$, and then at step $k$

- computes the gradient of the error,

  $$\left.\frac{\partial E}{\partial \mathbf{w}}\right|_{\mathbf{w}=\mathbf{w}_{k-1}} ,$$

  and possibly[2] higher-order derivatives of it, and then

- takes a step that reduces the value of $E$ by one of many numerical optimization methods.

The gradient computation is called *error back-propagation* and is described next.

[1] Fitting is not the whole story, since we want to avoid over-fitting for good generalization. However, fitting is a good first step, and we will address generalization later.

[2] However, higher order derivatives are typically *not* computed in practice, because of the high cost involved.

# 1 Back-Propagation

The computation of the $n$-th error term $E_n(\mathbf{w})$ can be rewritten as follows:

$$\begin{array}{rcl}\mathbf{x}^{(0)} & = & \mathbf{x}_n \\ \mathbf{x}^{(\ell)} & = & h^{(\ell)}(W^{(\ell)}\tilde{\mathbf{x}}^{(\ell-1)}) \quad \text{for} \quad \ell = 1, \ldots, L \\ E_n & = & \mathcal{L}(\mathbf{y}_n, \mathbf{x}^{(L)})\end{array}$$

where $(\mathbf{x}_n, \mathbf{y}_n)$ is the $n$-th training sample and, if the first $L_c$ layers of the net are convolutional layers with max-pooling,

$$h^{(\ell)}(\mathbf{a}) = \begin{cases} \pi(h(\mathbf{a})) & \text{for } \ell = 1, \ldots, L_c \\ h(\mathbf{a}) & \text{for } \ell = L_c + 1, \ldots, L-1 \\ h_y(\mathbf{a}) & \text{for } \ell = L \end{cases}$$

(the last layer sometimes has a different activation function $h_y$, as discussed earlier).

Computation of the derivatives of the error term $E_n(\mathbf{w})$ can be understood with reference to Figure 1. The term $E_n$ depends on the parameter vector $\mathbf{w}^{(\ell)}$ for layer $\ell$ through the output $\mathbf{x}^{(\ell)}$ from that layer and nothing else, so that we can write

$$\frac{\partial E_n}{\partial \mathbf{w}^{(\ell)}} = \frac{\partial E_n}{\partial \mathbf{x}^{(\ell)}} \frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{w}^{(\ell)}} \quad \text{for} \quad \ell = L, \ldots, 1 \tag{2}$$

and the first gradient on the right-hand side satisfies the backward recursion

$$\frac{\partial E_n}{\partial \mathbf{x}^{(\ell-1)}} = \frac{\partial E_n}{\partial \mathbf{x}^{(\ell)}} \frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{x}^{(\ell-1)}} \quad \text{for} \quad \ell = L, \ldots, 2 \tag{3}$$

because $E_n$ depends on the output $\mathbf{x}^{(\ell-1)}$ from layer $\ell - 1$ only through the output $\mathbf{x}^{(\ell)}$ from layer $\ell$. The recursion (3) starts with

$$\frac{\partial E_n}{\partial \mathbf{x}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \tag{4}$$

where $\mathbf{y}$ is the second argument to the loss function $\mathcal{L}$.

All the derivatives of $E_n$ in the equations above are *row* vectors, and the two matrices

$$\frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{w}^{(\ell)}} = \begin{bmatrix} \frac{\partial x_1^{(\ell)}}{\partial w_1^{(\ell)}} & \cdots & \frac{\partial x_1^{(\ell)}}{\partial w_{J^{(\ell)}}^{(\ell)}} \\ \vdots & & \vdots \\ \frac{\partial x_{D^{(\ell)}}^{(\ell)}}{\partial w_1^{(\ell)}} & \cdots & \frac{\partial x_{D^{(\ell)}}^{(\ell)}}{\partial w_{J^{(\ell)}}^{(\ell)}} \end{bmatrix} \quad \text{and} \quad \frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{x}^{(\ell-1)}} = \begin{bmatrix} \frac{\partial x_1^{(\ell)}}{\partial x_1^{(\ell-1)}} & \cdots & \frac{\partial x_1^{(\ell)}}{\partial x_{D^{(\ell-1)}}^{(\ell-1)}} \\ \vdots & & \vdots \\ \frac{\partial x_{D^{(\ell)}}^{(\ell)}}{\partial x_1^{(\ell-1)}} & \cdots & \frac{\partial x_{D^{(\ell)}}^{(\ell)}}{\partial x_{D^{(\ell-1)}}^{(\ell-1)}} \end{bmatrix} \tag{5}$$

are the *Jacobian matrices* of the layer output $\mathbf{x}^{(\ell)}$ with respect to the layer parameters and inputs. Computation of the entries of these Jacobians is a simple exercise in differentiation, and is left to the Appendix.

The equations (2-4) are the basis for the *back-propagation* algorithm for the computation of the gradient of the training error $E(\mathbf{w})$ with respect to the parameter vector $\mathbf{w}$ of the neural net (Algorithm 1). The algorithm loops over the training samples. For each sample, it feeds the input $\mathbf{x}_n$ to the net to compute the layer outputs $\mathbf{x}^{(\ell)}$ for that sample and for all $\ell = 1, \ldots, L$, in this order. The algorithm temporarily stores all the values $\mathbf{x}^{(\ell)}$, because they are needed to compute the required derivatives. This initial volley of computation is called *forward propagation* (of the inputs). The algorithm then revisits the layers in

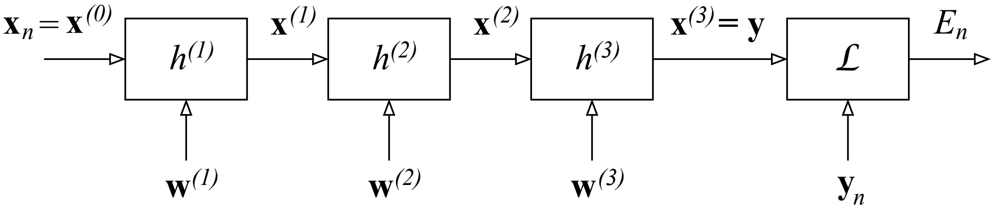

Figure 1: Example data flow for the computation of the error term $E_n$ for a neural net with $L = 3$ layers. When viewed from the error term $E_n$, the output $\mathbf{x}^{(\ell)}$ from layer $\ell$ (pick for instance $\ell = 2$) is a bottleneck of information for both the parameter vector $\mathbf{w}^{(\ell)}$ for that layer and the output $\mathbf{x}^{(\ell-1)}$ from the previous layer ($\ell - 1 = 1$ in the example). This observation justifies the use of the chain rule for differentiation to obtain equations (2) and (3).

reverse order while computing the derivatives in equations (2) and (3), and concatenates the resulting $L$ layer gradients into a single gradient $\frac{\partial E_n}{\partial \mathbf{w}}$. This computation is called *back-propagation* (of the derivatives). The gradient of $E(\mathbf{w})$ is the average (from equation (1)) of the gradients computed for each of the samples:

$$\frac{\partial E}{\partial \mathbf{w}} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E_n}{\partial \mathbf{w}} = \frac{1}{N} \sum_{n=1}^{N} \begin{bmatrix} \frac{\partial E_n}{\partial \mathbf{w}^{(1)}} \\ \vdots \\ \frac{\partial E_n}{\partial \mathbf{w}^{(L)}} \end{bmatrix} .$$

This average vector can be accumulated (see last assignment in Algorithm 1) as back-propagation progresses. For succinctness, operations are expressed as matrix-vector computations in Algorithm 1. In practice, the matrices would be very sparse, and convolutions and explicit loops over appropriate indices are used instead.

**Algorithm 1** Backpropagation

```
function $\nabla E \leftarrow \text{backprop}(T, \mathbf{w}, D^{(0)}, \ldots, D^{(L)})$
    $\nabla E = \text{zeros}(\text{size}(\mathbf{w}))$
    for $n = 1, \ldots, N$ do
        $\mathbf{x}^{(0)} = \mathbf{x}_n$
        for $\ell = 1, \ldots, L$ do                                    ▷ Forward propagation
            $W \leftarrow \text{reshape}(\mathbf{w}^{(\ell)}, D^{(\ell)}, D^{(\ell-1)} + 1)$          ▷ Turn vector $\mathbf{w}^{(\ell)}$ into a matrix
            $\mathbf{x}^{(\ell)} \leftarrow h^{(\ell)}(W\tilde{\mathbf{x}}^{(\ell-1)})$      ▷ Compute layer outputs to be used in back-propagation
        end for
        $\nabla E_n = [\,]$                ▷ Initially empty contribution of the $n$-th sample to the error gradient
        $\mathbf{g} = \frac{\partial \mathcal{L}(\mathbf{y}_n, \mathbf{x}^{(L)})}{\partial \mathbf{y}}$                                        ▷ $\mathbf{g}$ is $\frac{\partial E_n}{\partial \mathbf{x}^{(\ell)}}$
        for $\ell = L, \ldots, 2$ do                                    ▷ Back-propagation
            $\nabla E_n \leftarrow [\mathbf{g}\frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{w}^{(\ell)}}; \nabla E_n]$          ▷ Derivatives are evaluated at $\mathbf{w}^{(\ell)}$ and $\mathbf{x}^{(\ell)}$
            $\mathbf{g} \leftarrow \mathbf{g}\frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{x}^{(\ell-1)}}$                                            ▷ Ditto
        end for
        $\nabla E \leftarrow \frac{(n-1)\nabla E + \nabla E_n}{n}$                            ▷ Accumulate the average
    end for
end function
```

$$\mu_t = \min\left(\mu_{\max}, 1 - \frac{1}{2}\,\frac{1}{\left\lfloor \frac{t}{250} \right\rfloor + 1}\right)$$

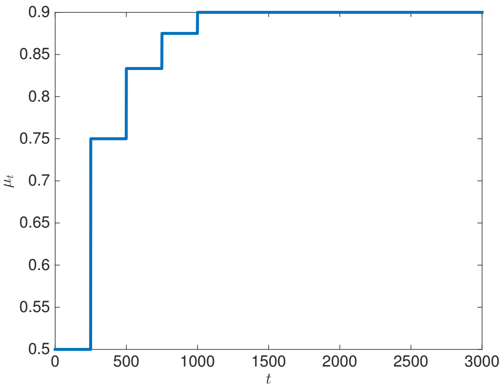

Figure 2: A possible schedule [7] for the momentum coefficient $\mu_t$.

## 2 Gradient Descent

In principle, a neural net can be trained by minimizing the training error $E(\mathbf{w})$ defined in equation (1) by any of a vast variety of numerical optimization methods [4, 2]. At one end of the spectrum, methods that make no use of gradient information take too many steps to converge. At the other end, methods that use second-order derivatives (Hessian) to determine high-quality steps tend to be too expensive in terms of both space and time at each iteration, although some researchers advocate these types of methods [3]. By far the most widely used methods employ gradient information, computed by back-propagation [1]. Line search is too expensive, and the step size is therefore chosen according to some heuristic instead.

The *momentum method* [5, 7] starts from an initial value $\mathbf{w}_0$ chosen at random and iterates as follows:

$$\begin{aligned} \mathbf{v}_{t+1} &= \mu_t \mathbf{v}_t - \alpha \nabla E(\mathbf{w}_t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \mathbf{v}_{t+1} \ . \end{aligned}$$

The vector $\mathbf{v}_{t+1}$ is the *step* or *velocity* that is added to the old value $\mathbf{w}_t$ to compute the new value $\mathbf{w}_{t+1}$. The scalar $\alpha > 0$ is the *learning rate* that determines how fast to move in the direction opposite to the error gradient $\nabla E(\mathbf{w})$, and the time-dependent scalar $\mu_t \in [0, 1]$ is the *momentum coefficient*. Gradient descent is obtained when $\mu_t = 0$. Greater values of $\mu_t$ encourage steps in a consistent direction (since the new velocity $\mathbf{v}_{t+1}$ has a greater component in the direction of the old velocity $\mathbf{v}_t$ than if no momentum were present), and these steps accelerate descent when the gradient of $E(\mathbf{w})$ is small, as is the case around shallow minima. The value of $\mu_t$ is often varied according to some schedule like the one in Figure 2. The rationale for the increasing values over time is that momentum is more useful in later stages, in which the gradient magnitude is very small as $\mathbf{w}_t$ approaches the minimum.

The learning rate $\alpha$ is often fixed, and is a parameter of critical importance [8]. A rate that is too large leads to large steps that often overshoot, and a rate that is too small leads to very slow progress. In practice, $\alpha$ is chosen by cross-validation to be some value much smaller than 1.

**Mini-Batches.** The gradient of the error $E(\mathbf{w})$ is expensive to compute, and one tends to use as large a learning rate as possible so as to minimize the number of steps taken. One way to prevent the resulting overshooting would be to do *online learning*, in which each step $\mu_t \mathbf{v}_t - \alpha \nabla E_n(\mathbf{w}_t)$ (there is one such step for each training sample) is taken right away, rather than accumulated into the step $\mu_t \mathbf{v}_t - \alpha \nabla E(\mathbf{w}_t)$ (no subscript $n$ here). In contrast, using the latter step is called *batch learning*. Computing $\nabla E_n$ is much less expensive (by a factor of $N$) than computing $\nabla E$. In addition—and most importantly for convergence behavior—online learning breaks a single batch step into $N$ small steps, after each of which the error function is re-evaluated. As a result, the online steps can follow very "curved" paths, whereas a single batch step can only move in a straight line. Because of this greater flexibility, online learning converges faster than batch learning for the same overall computational effort. The small online steps, however, have high variance, because each of them is taken based on minimal amounts of data. One can improve convergence further by processing *mini-batches* of training data: Accumulate $B$ gradients $\nabla E_n$ from the data in one mini-batch into a single gradient $\nabla E$, take the step, and move on to the next mini-batch. It turns out that small values of $B$ achieve the best compromise between reducing variance and keeping steps flexible. Values of $B$ around a few dozen are common.

**Termination.** When used outside learning, gradient descent is typically stopped when steps make little progress, as measured by step size $\|\mathbf{w}_t - \mathbf{w}_{t-1}\|$ and/or decrease in function value $|E(\mathbf{w}_t) - E(\mathbf{w}_{t-1})|$. When training a deep net, on the other hand, descent is often stopped earlier to improve generalization. Specifically, one monitors the error on a validation set, rather than on the training set, and stops when the validation-set error bottoms out, even if the training-set error would continue to decrease. A different way to improve generalization, sometimes used in combination with early termination, is discussed in Section 3.

## 3 Dropout

Since deep nets have a large number of parameters, they would need impractically large training sets to avoid overfitting if no special measures are taken during training. Early termination, described at the end of the previous section, is one such measure. In general, the best way to avoid overfitting in the presence of limited data would be to build one net for every possible setting of the parameters, compute the posterior probability of each setting given the training set, and then aggregate the nets into a single predictor that computes the average output weighted by the posterior probabilities. This approach, which is reminiscent of building a forest of trees, is obviously infeasible to implement for nontrivial nets.

One way to approximate this scheme in a computationally efficient way is called the *dropout* method [6]. Given a deep net to be trained, a *dropout net* is obtained by flipping a biased coin for each node of the original net and "dropping" that node if the flip turns out heads. Dropping the node means that all the weights and biases for that node are set to zero, so that the node becomes effectively inactive.

One then trains the net by using mini-batches of training data, and performs one iteration of training on each mini-batch after turning off neurons independently with probability $1 - p$. When training is done, all the weights in the network are multiplied by $p$, and this effectively averages the outputs of the nets with weights that depend on how often a unit participated in training. The value of $p$ is typically set to $1/2$.

Each dropout net can be viewed as a different net, and the dropout method effectively samples a large number of nets efficiently.

# Appendix: The Jacobians for Back-Propagation

If $h^{(\ell)}$ is a point function, that is, if it is $\mathbb{R} \to \mathbb{R}$, the individual entries of the Jacobian matrices (5) are easily found to be (reverting to matrix subscripts for the weights)

$$\frac{\partial x_i^{(\ell)}}{\partial W_{qj}^{(\ell)}} = \delta_{iq} \frac{dh^{(\ell)}}{da_i^{(\ell)}} \tilde{x}_j^{(\ell-1)} \quad \text{and} \quad \frac{\partial x_i^{(\ell)}}{\partial x_j^{(\ell-1)}} = \frac{dh^{(\ell)}}{da_i^{(\ell)}} W_{ij}^{(\ell)} \; .$$

The Kronecker delta

$$\delta_{iq} = \begin{cases} 1 & \text{if } i = q \\ 0 & \text{otherwise} \end{cases}$$

in the first of the two expressions above reflects the fact that $x_i^{(\ell)}$ depends only on the $i$-th activation, which is in turn the inner product of row $i$ of $W^{(\ell)}$ with $\tilde{\mathbf{x}}^{(\ell-1)}$. Because of this, the derivative of $x_i^{(\ell)}$ with respect to entry $W_{qj}^{(\ell)}$ is zero if this entry is not in that row, that is, when $i \neq q$. The expression

$$\frac{dh^{(\ell)}}{da_i^{(\ell)}} \quad \text{is shorthand for} \quad \left. \frac{dh^{(\ell)}}{da} \right|_{a=a_i^{(\ell)}} \; ,$$

the derivative of the activation function $h^{(\ell)}$ with respect to its only argument $a$, evaluated for $a = a_i^{(\ell)}$.

For the ReLU activation function $h^\ell = h$,

$$\frac{dh^{(\ell)}}{da} = \begin{cases} 1 & \text{for } a \geq 0 \\ 0 & \text{otherwise} \end{cases} \; .$$

For the ReLU activation function followed by max-pooling, $h^\ell(\cdot) = \pi(h(\cdot))$, on the other hand, the value of the output at index $i$ is computed from a window $P(i)$ of activations, and only one of the activations (the one with the highest value) in the window is relevant to the output[3]. Let then

$$p_i^{(\ell)} = \max_{q \in P(i)} (h(a_q^{(\ell)}))$$

be the value resulting from max-pooling over the window $P(i)$ associated with output $i$ of layer $\ell$. Furthermore, let

$$\hat{q} = \arg\max_{q \in P(i)} (h(a_q^{(\ell)}))$$

be the index of the activation where that maximum is achieved, where for brevity we leave the dependence of $\hat{q}$ on activation index $i$ and layer $\ell$ implicit. Then,

$$\frac{\partial x_i^{(\ell)}}{\partial W_{qj}^{(\ell)}} = \delta_{q\hat{q}} \frac{dh^{(\ell)}}{da_{\hat{q}}^{(\ell)}} \tilde{x}_j^{(\ell-1)} \quad \text{and} \quad \frac{\partial x_i^{(\ell)}}{\partial x_j^{(\ell-1)}} = \frac{dh^{(\ell)}}{da_{\hat{q}}^{(\ell)}} W_{\hat{q}j}^{(\ell)} \; .$$

[3] In case of a tie, we attribute the highest values in $P(i)$ to one of the highest inputs, say, chosen at random.

## References

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] S. Boyd and L. Vandeberghe. *Convex Optimization*. Cambdrige University Press, 2004.

[3] J. Martens. Learning recurrent neural networks with Hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning*, pages 735–742, 2011.

[4] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, NY, 1999.

[5] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[6] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[7] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1139–1147, 2013.

[8] D. R. Wilson and T. R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16:1429–1451, 2003.