Training Convolutional Neural Networks

Carlo Tomasi

August 17, 2021

1 The Soft-Max Simplex

Neural networks are typically designed to compute real-valued functions $\mathbf{y} = h(\mathbf{x}) \ : \ \mathbb{R}^d \to \mathbb{R}^e$ of their input $\mathbf{x}$. When a classifier is needed, a soft-max function is used as the last layer, with e entries in its output vector $\mathbf{p}$ if there are e classes in the label space Y. The class corresponding to input $\mathbf{x}$ is then is found as the arg max of $\mathbf{p}$. Thus, the network can be viewed as a function

$$\mathbf{p} = f(\mathbf{x}, \mathbf{w}) \ : \ X \to P$$

that transforms data space X into the *soft-max simplex* P, the set of all nonnegative real-valued vectors $\mathbf{p} \in \mathbb{R}^e$ whose entries add up to 1:

$$P \stackrel{\text{def}}{=} \{\mathbf{p} \in \mathbb{R}^e \ : \ \mathbf{p} \geq \mathbf{0} \ \text{ and } \ \sum_{i=1}^{e} p_i = 1\} \ .$$

This set has dimension $e - 1$, and is the convex hull of the e columns of the identity matrix in $\mathbb{R}^e$. Figure 1 shows the 1-simplex and the 2-simplex.[1]

The vector $\mathbf{w}$ in the expression above collects all the parameters of the neural network, that is, the gains and biases of all the neurons. More specifically, for a deep neural network with K layers indexed by $k = 1, \ldots, K$, we can write

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

where $\mathbf{w}^{(k)}$ is a vector collecting both gains and biases for layer k.

If the arg max rule is used to compute the class,

$$\hat{y} = h(\mathbf{x}) = \arg\max \mathbf{p} \ ,$$

then the network make a correct prediction for a data point with true class $c \in \{1, \ldots, e\}$ if $\mathbf{p}$ falls in the interior of the decision region

$$P_c = \{p_c \geq p_j \ \text{ for } \ j \neq c\} \ ,$$

because then $\arg\max \mathbf{p} = c$. Each of these regions is convex, because its boundaries are defined by linear inequalities in the entries of $\mathbf{p}$. Thus, when used for classification, the neural network can be viewed as learning a transformation of the original, possibly very complicated decision regions in X into the convex, very simple decision regions in the soft-max simplex.

[1] In geometry, the simplices are named by their dimension, which is one less than the number of classes.

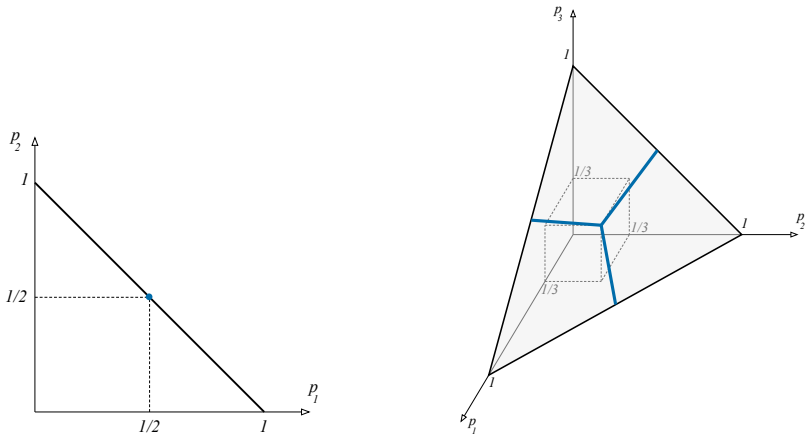

Figure 1: The 1-simplex for two classes (dark segment in the diagram on the left) and the 2-simplex for three classes (light triangle in the diagram on the right). The blue dot on the left and the blue line segments on the right are the boundaries of the decision regions. The boundaries meet at the *unit point* $\mathbf{1}/e$ in e dimensions.

2 Loss

The risk L_T to be minimized to train a neural network is the average loss on 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 categorical in a classification problem, and real-valued vectors in a regression problem.

For a regression problem, the loss function is typically the quadratic loss,

$$\ell(\mathbf{y}, \mathbf{y}') = \|\mathbf{y} - \mathbf{y}'\|^2 \; .$$

For classification, on the other hand, we would like the risk $L_T(h)$ to be differentiable, in order to be able to use gradient descent methods during training. However, the arg max is a piecewise-constant function, and its derivatives are either zero (almost everywhere) or undefined (where the arg max changes value), and gradient descent cannot be used. The zero-one loss function has similar properties.

To address these issue, a differentiable loss defined on f (the output from the soft-max) is used as a proxy for the zero-one loss defined on h (the output from the arg-max). Specifically, the multi-class cross-entropy loss is used, which we studied in the context of logistic-regression classifiers. Its definition is repeated here for convenience:

$$\ell(y, \mathbf{p}) = -\log p_y \; .$$

Equivalently, if $\mathbf{q}(y) = (q_1(y), \ldots, q_e(y))$ is the one-hot encoding of the true label y, the cross-entropy loss can also be written as follows:

$$\ell(y, \mathbf{p}) = -\sum_{c=1}^{e} q_c(y) \log p_c \; .$$

With these definitions, L_T is a piecewise-differentiable function, and one can use gradient or sub-gradient methods to compute the gradient of L_T with respect to the parameter vector $\mathbf{w}$. This is typically a very long vector, as it lists all the parameters (neuron weights or convolution kernel coefficients) in all of the network's K layers:

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

Exceptions to differentiability are due to the use of the ReLU (which has a cusp at the origin) as the nonlinearity in neurons, as well as to the possible use of max-pooling. These exceptions are pointwise, and are typically ignored in both the literature and the software packages used to minimize L_T. If desired, they could be addresses by either computing sub-gradients rather than gradients [3], or rounding out the cusps with differentiable joints.

As usual, once the loss has been settled on, the training risk is defined as the average loss over the training set, and expressed as a function of the parameters $\mathbf{w}$ of f:

$$L_T(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \ell_n(\mathbf{w}) \quad \text{where} \quad \ell_n(\mathbf{w}) = \ell(y_n, f(\mathbf{x}_n, \mathbf{w})) \; . \tag{1}$$

3 Back-Propagation

A local minimum for the risk $L_T(\mathbf{w})$ is found by an iterative procedure that starts with some *initial values* $\mathbf{w}_0$ for $\mathbf{w}$, and then at step t performs the following operations:

- Compute the gradient of the training risk,

$$\left. \frac{\partial L_T}{\partial \mathbf{w}} \right|_{\mathbf{w}=\mathbf{w}_{t-1}} .$$

- Take a step that reduces the value of L_T by moving in the general direction of the negative gradient by a variant of the steepest descent method called *Stochastic Gradient Descent* (SGD), discussed in Section 4, or other similar descent algorithms.

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

The computation of the n-th loss term $\ell_n(\mathbf{w})$ on the training data point $\mathbf{x}_n$ can be rewritten by passing $\mathbf{x}_n$ through the layers of a network from input to output and then through the loss function as follows:

$$\begin{array}{rrcl} \text{Rename the input:} & \mathbf{x}^{(0)} & = & \mathbf{x}_n \\ \text{Go through each layer:} & \mathbf{x}^{(k)} & = & f^{(k)}(\mathbf{x}^{(k-1)}) \quad \text{for} \quad k = 1, \ldots, K \\ \text{Rename the output from the soft-max layer:} & \mathbf{p} & = & \mathbf{x}^{(K)} \\ \text{Compute the loss:} & \ell_n & = & \ell(y_n, \mathbf{p}) \end{array}$$

where y_n is the true label for $\mathbf{x}_n$ and $f^{(k)}$ describes the function implemented by layer k. This chain of computations is called the *forward pass* or *forward propagation* through the network and loss function.

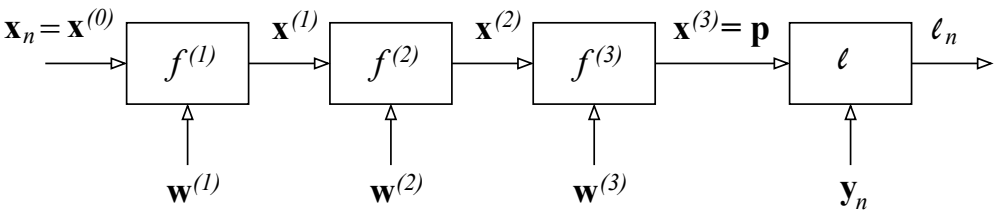

Figure 2: Example data flow for the computation of the loss term ℓ_n for a neural network with $K = 3$ layers. When viewed from the loss term ℓ_n, the output $\mathbf{x}^{(k)}$ from layer k (pick for instance $k = 2$) is a bottleneck of information for both the parameter vector $\mathbf{w}^{(k)}$ for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

Computation of the derivatives of the loss term $\ell_n(\mathbf{w})$ can be understood with reference to Figure 2, and is a straight-forward application of the chain rule for differentiation as follows.

The value of ℓ_n generally varies whenever any of the weights in $\mathbf{w}$ changes. More specifically, it depends on the parameter vector $\mathbf{w}^{(k)}$ for layer k only through the output $\mathbf{x}^{(k)}$ from that layer

$$\ell_n(\mathbf{w}^{(k)}) = \ell_n(\mathbf{x}^{(k)}(\mathbf{w}^{(k)}))$$

because the layers form a cascade. We can therefore use the chain rule of differentiation to write

$$\frac{\partial \ell_n}{\partial \mathbf{w}^{(k)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}} \quad \text{for} \quad k = K, \ldots, 1 \; . \tag{2}$$

For the same reason, ℓ_n depends on the output $\mathbf{x}^{(k-1)}$ from layer $k-1$ only through the output $\mathbf{x}^{(k)}$ from layer k:

$$\ell_n(\mathbf{x}^{(k-1)}) = \ell_n(\mathbf{x}^{(k)}(\mathbf{x}^{(k-1)}))$$

and we can therefore write the following backward recursion for the gradient on the right-hand side of equation (2):

$$\frac{\partial \ell_n}{\partial \mathbf{x}^{(k-1)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}} \quad \text{for} \quad k = K, \ldots, 2 \; . \tag{3}$$

The recursion (3) starts with

$$\frac{\partial \ell_n}{\partial \mathbf{x}^{(K)}} = \frac{\partial \ell}{\partial \mathbf{p}} \tag{4}$$

where $\mathbf{p}$ is the second argument to the loss function $\ell(y_n, \mathbf{p})$.

In the equations above, the derivative of a function with respect to a vector is to be interpreted as the *row* vector of all derivatives. Let d_k be the dimensionality (number of entries) of $\mathbf{x}^{(k)}$, and j_k be the dimensionality of $\mathbf{w}^{(k)}$. The two matrices

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

are the *Jacobian matrices* of the layer output $\mathbf{x}^{(k)}$ with respect to the layer parameters and inputs respectively. Computation of the entries of these Jacobians is a simple exercise in differentiation, and is left to the Appendix. All derivatives in these Jacobian matrices are *local* to each layer, in that they only require knowing the structure of that layer.

The equations (2) through (5) are the basis for the *back-propagation* algorithm for the computation of the gradient of the training risk $L_T(\mathbf{w})$ with respect to the parameter vector $\mathbf{w}$ of the neural network (Algorithm 1).

Specifically, the algorithm loops over the training samples. For each sample, it feeds the input $\mathbf{x}_n$ to the network to compute the layer outputs $\mathbf{x}^{(k)}$ for that sample and for all $k = 1, \ldots, K$, in this order (forward propagation). The algorithm temporarily stores all the values $\mathbf{x}^{(k)}$, because they are needed to compute the required derivatives for applying the chain rule.

The algorithm then revisits the layers in reverse order while computing the derivatives in equation (4) the first time around and then those in equations (2) and (3) for decreasing values of k. It finally concatenates the resulting K layer gradients into a single gradient $\frac{\partial \ell_n}{\partial \mathbf{w}}$. This computation is called *back-propagation* (of the derivatives).

The gradient of $L_T(\mathbf{w})$ is the average (from equation (1) and linearity of both summation and gradient) of the gradients computed for each of the samples:

$$\frac{\partial L_T}{\partial \mathbf{w}} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial \ell_n}{\partial \mathbf{w}} = \frac{1}{N} \sum_{n=1}^{N} \begin{bmatrix} \frac{\partial \ell_n}{\partial \mathbf{w}^{(1)}} \\ \vdots \\ \frac{\partial \ell_n}{\partial \mathbf{w}^{(K)}} \end{bmatrix}$$

(here, the derivatives with respect to $\mathbf{w}^{(k)}$ are read as column vectors of derivatives). This average vector can be accumulated over n (see last assignment statement 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.

4 Stochastic Gradient Descent

In principle, a neural network can be trained by minimizing the training risk $L_T(\mathbf{w})$ defined in equation (1) by any of a vast variety of numerical optimization methods [6, 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 [5]. By far the most widely used methods employ gradient information, computed by back-propagation [1]. Line search is too expensive, since it requires many function evaluations, each of which in turn requires a full forward pass through the network and loss.

The Learning Rate. Because of the high cost of line search, the step size α_t, which is also called the *learning rate*, is chosen according to some heuristic instead, and the standard (that is non-stochastic) version of gradient descent is then simply

$$\mathbf{w}_{t+1} \quad = \quad \mathbf{w}_t - \alpha_t \nabla L_T(\mathbf{w}_t) \tag{6}$$

Algorithm 1 Backpropagation

function $\nabla L_T \leftarrow \text{backprop}(T, \mathbf{w} = [\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(K)}], \ell)$
  $\nabla L_T = \text{zeros}(\text{size}(\mathbf{w}))$
  for $n = 1, \ldots, N$ **do**
    $\mathbf{x}^{(0)} = \mathbf{x}_n$
    for $k = 1, \ldots, K$ **do** ▷ Forward propagation
      $\mathbf{x}^{(k)} \leftarrow f^{(k)}(\mathbf{x}^{(k-1)}, \mathbf{w}^{(k)})$ ▷ Compute and store layer outputs to be used in back-propagation
    end for
    $\nabla \ell_n = [\,]$ ▷ Initially empty contribution of the n-th sample to the loss gradient
    $\mathbf{g} = \frac{\partial \ell(y_n, \mathbf{x}^{(K)})}{\partial \mathbf{p}}$ ▷ $\mathbf{g}$ is $\frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}}$
    for $k = K, \ldots, 2$ **do** ▷ Back-propagation
      $\nabla \ell_n \leftarrow [\mathbf{g} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}}; \nabla \ell_n]$ ▷ Derivatives are evaluated at $\mathbf{w}^{(k)}$ and $\mathbf{x}^{(k)}$
      $\mathbf{g} \leftarrow \mathbf{g} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}}$ ▷ Ditto
    end for
    $\nabla L_T \leftarrow \frac{(n-1)\nabla L_T + \nabla \ell_n}{n}$ ▷ Accumulate the average
  end for
end function

starting from an initial $\mathbf{w}_0$ chosen at random near the origin.

The learning rate α_t of critical importance [7]. A rate that is too large leads to large steps that often overshoot minima, and a rate that is too small leads to very slow progress. In practice, the learning rate is first set to some relatively large constant, say, 10^{-3}, to encourage rapid risk decrease in the early steps of optimization. Once the risk shows signs of flattening out, the step size is reduced gradually, so as not to miss narrow valleys that may contain a deep minimum.

One way to decrease the learning rate is to follow some fixed schedule. However, since convergence can take between hours and weeks for typical applications, the value of L_T is often monitored through some user interface. Every time progress starts to saturate, the learning rate can also be decreased manually (say, divided by 10).

Automatic methods for step size selection have been proposed as well [4]. Empirically, these show typically good results early on during training. As a minimum is approached, is has been found that switching to a fixed and small step size is often preferable.

Mini-Batches. The gradient of the risk $L_T(\mathbf{w})$ 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 $-\alpha \nabla \ell_n(\mathbf{w}_t)$ (there is one such step for each training sample) is taken right away, rather than accumulated into the step $-\alpha \nabla L_T(\mathbf{w}_t)$ (no subscript n here). In contrast, using the latter step as done in equation (6) is called *batch learning*.

Computing $\nabla \ell_n$ is much less expensive (by a factor of N) than computing ∇L_T. In addition—and most importantly for convergence behavior—online learning breaks a single batch step into N small steps, after each of which the value of the risk is re-evaluated. As a result, a sequence of N online steps can follow very "curved" paths, whereas a single batch step can only move in a fixed direction in parameter space for the same computational cost. 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 $\nabla \ell_n$ from the data in one mini-batch into a single gradient $\nabla L_T^{(B)}$, 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.

Early Termination. When used outside machine 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 $|L_T(\mathbf{w}_t) - L_T(\mathbf{w}_{t-1})|$. When training a deep network, on the other hand, descent is often stopped earlier to improve generalization. Specifically, one monitors the zero-one risk error of the classifier on a validation set, rather than the cross-entropy risk of the soft-max output on the training set, and stops when the validation-set error bottoms out, even if the training-set risk would continue to decrease.

Appendix

The Jacobians for Back-Propagation

If $f^{(k)}$ 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)

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

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

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

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

For the ReLU activation function $h^k = h$,

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

For the ReLU activation function followed by max-pooling, $h^k(\cdot) = \pi(h(\cdot))$, 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[2]. Let then

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

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

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

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 k implicit. Then,

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

[2]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. B. Hiriart-Urruty and C. Lemaréchal. *Convex analysis and minimization algorithms I: Fundamentals*, volume 305. Springer science & business media, 2013.

[4] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

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

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

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