Training Convolutional Neural Networks

Carlo Tomasi

February 13, 2019

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 has a low training risk if the transformed data points $\mathbf{p}$ fall in the decision regions

$$P_c = \{p_c \geq p_j \ \text{ for } \ j \neq c\} \quad \text{for} \quad c = 1, \ldots, e \ .$$

These regions are convex, because their 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 decision regions in X into the convex 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.

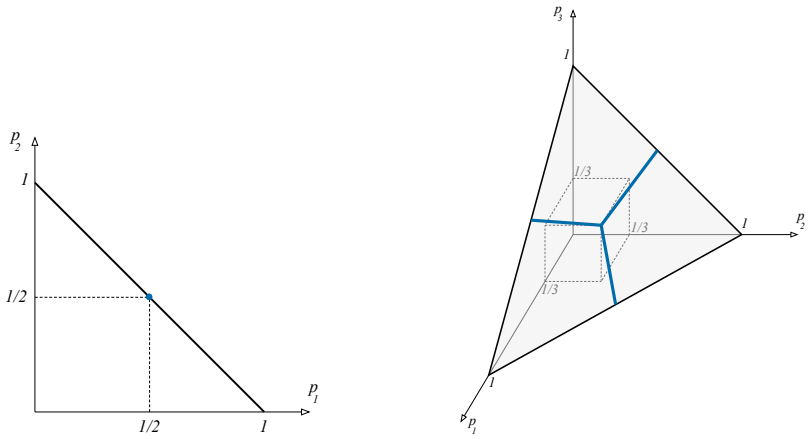

Figure 1: The 1-simplex for two classes (dark segment in the diagram on the left) and the 2-simplex for three classes (light triangle in the diagram on the right). The blue dot on the left and the blue line segments on the right are the boundaries of the decision regions. The boundaries meet at the *unit 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. However, the arg max is a piecewise-constant function, and its derivatives are either zero or undefined (where the arg max changes value). The zero-one loss function has similar properties. To address these issue, a differentiable loss defined on f is used as a proxy for the zero-one loss defined on h. Specifically, the multi-class cross-entropy loss, discussed next, is used. For pedagogical clarity, this loss is first introduced for a two-class classifier, and then generalized to the multi-class case.

2.1 The Two-Class Cross-Entropy Loss

The performance of a classifier is typically measured by how many mistakes it makes. If y is the true label of data point $\mathbf{x}$ and $\hat{y} = \hat{h}(\mathbf{x})$ is the class returned by the classifier $\hat{h}$, the zero-one loss is then

$$\ell_{0\text{-}1}(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{otherwise} \end{cases}$$

as we saw in an earlier note. However, we will see that neural networks are trained by steepest descent, for which the gradient of the risk relative to the network parameters is needed. If the loss is not differentiable, the gradient cannot be computed.

Our way to avoid this difficulty is to define a loss function ℓ that is a *differentiable* function of the softmax output $\mathbf{p}$ of the neural network, rather than a function of $\hat{y}$. If there are only two classes, $\mathbf{p} \in \mathbb{R}^2$, and because of normalization only one of the two values needs to be recorded, say, the score for the hypothesis that $y = 1$. Call that value p. Notice the shell game: The zero-one loss is really what we are after, but minimizing that would lead to a complex combinatorial problem, because $\hat{y}$ (and therefore $\ell_{0\text{-}1}$) is a discontinuous function of the network parameters. In order to find a solution efficiently, we then use the softmax score p as a proxy for $\hat{y}$, and define a loss function in terms of that:

$$\ell(y, p) \ .$$

This substitution is reasonable. After all, $\hat{y} \in \{0, 1\}$ and $p \in [0, 1]$, and a good classifier assigns softmax scores close to 1 to data points with label 1, and scores close to 0 to data points with label 0.

The rest of this Section introduces and describes a loss function with these properties, called the cross-entropy loss, and proves convexity of the resulting risk function. Appendix A establishes a connection between this measure of loss and probabilistic and information-theoretic considerations.

The *cross-entropy loss* of assigning score p to a data point with true label y is defined as follows:

$$\ell(y, p) \stackrel{\mathsf{def}}{=} \begin{cases} -\log p & \text{if } y = 1 \\ -\log(1 - p) & \text{if } y = 0. \end{cases}$$

Since y is binary, it can be used as a "switch" to rewrite this function like this:

$$\ell(y, p) = -y \log p - (1 - y) \log(1 - p) \ .$$

In these expressions, the base of the logarithm is unimportant, as long as the same base is used throughout, because logarithms in different bases differ by a multiplicative constant. In Appendix A, the logarithm base 2 is used, as is customary in information theory.

Figure 2 shows the function. The domain is the cross product

$$\{0, 1\} \times [0, 1] \ ,$$

that is, two segments on the plane (left in the Figure), and the right panel in the Figure shows the values of the loss on these two segments in corresponding colors. From either formula or Figure, we see that

$$\ell(1, p) = \ell(0, 1 - p) \ ,$$

reflecting the symmetry between the two labels and their scores. The two curves therefore meet when $p = 1/2$:

$$\ell(1, 1/2) = \ell(0, 1/2) = -\log(1/2) \ .$$

If base-2 logarithms are used, this value is 1.

The cross-entropy function makes at least partial sense as a measure of loss: When the true label y is one (blue), no cost is incurred when the score p is one as well, since p scores the hypothesis that $y = 1$. As p decreases for this value of the true label, a bigger and bigger cost is incurred, and

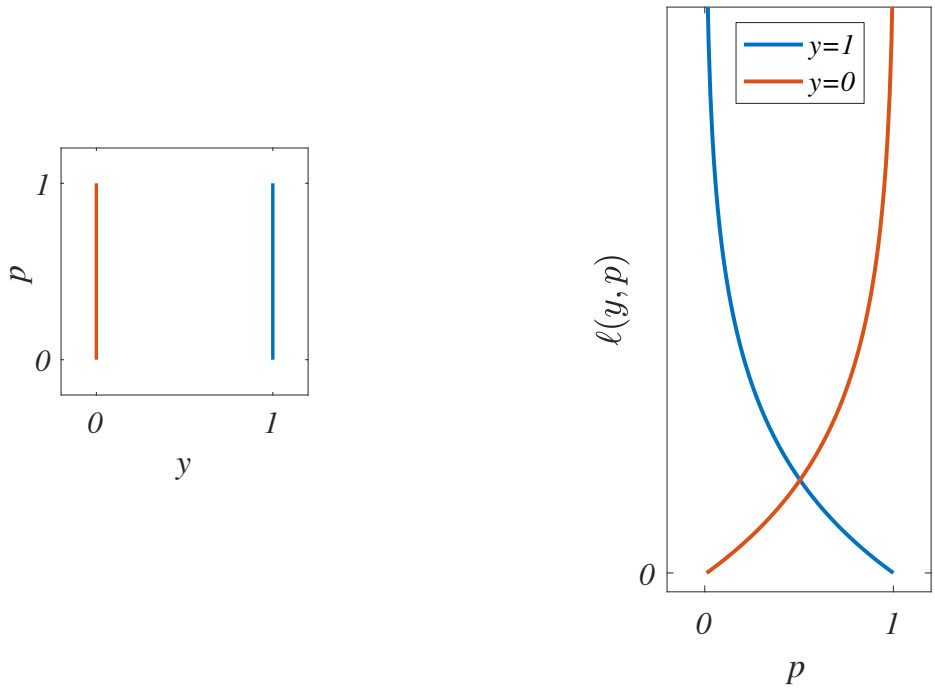

Figure 2: (Left) The two colored segments are the domain of any loss function for a binary classifier based on softmax scores. (Right) The cross-entropy loss. The two curves are defined on the segments of corresponding colors on the left. They asymptote to infinity for $p \to 0$ (when $y = 1$, blue) and for $p \to 1$ (when $y = 0$, orange).

as $p \to 0$ the cost grows to infinity. Why an unbounded loss makes sense in not entirely clear for neural networks.[2]

2.2 The Multi-Class Cross-Entropy Loss

The cross-entropy loss for the K-class case is the immediate extension of the definition we saw for $K = 2$:

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

and we can use the fact that $\mathbf{q}$ is an indicator function to rewrite this loss as follows:

$$\ell(y, \mathbf{p}) = -\sum_{k=1}^{K} q_k \log p_k$$

where $\mathbf{q}$ is the one-hot encoding of y. The two expressions are equivalent, but the latter is more convenient to use when computing derivatives.

The rest of the story is the same as in the case $K = 2$: The risk is still defined as the average loss over the training set:

$$L_T(b, \mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \ell(y_n \ ; \mathbf{w}) \ .$$

3 Back-Propagation

If the cross-entropy loss is used, 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 $\mathbf{w}$.

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 [4], or rounding out the cusps with differentiable joints.

As usual, 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:

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

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,

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

[2] In the literature, the cross-entropy loss was initially introduced for logistic-regression classifiers, for which an unbounded loss can be justified.

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

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

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

$$\begin{aligned} \mathbf{x}^{(0)} &= \mathbf{x}_n \\ \mathbf{x}^{(k)} &= f^{(k)}(W^{(k)}\tilde{\mathbf{x}}^{(k-1)}) \quad \text{for} \quad k = 1, \ldots, K \\ \mathbf{p} &= \mathbf{x}^{(K)} \\ \ell_n &= \ell(y_n, \mathbf{p}) \end{aligned}$$

where $(\mathbf{x}_n, y_n)$ is the n-th training sample and $f^{(k)}$ describes the function implemented by layer k.

Computation of the derivatives of the loss term $\ell_n(\mathbf{w})$ can be understood with reference to Figure 3. The term ℓ_n depends on the parameter vector $\mathbf{w}^{(k)}$ for layer k through the output $\mathbf{x}^{(k)}$ from that layer and nothing else, so that we can write

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

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

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

because ℓ_n depends on the output $\mathbf{x}^{(k-1)}$ from layer $k-1$ only through the output $\mathbf{x}^{(k)}$ from layer k. 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 ℓ.

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. Computation of the entries of these Jacobians is a simple exercise in differentiation, and is left to the Appendix.

The equations (2-5) 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). 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

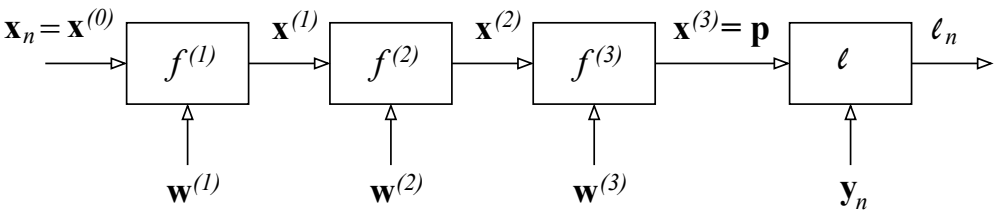

Figure 3: Example data flow for the computation of the loss term ℓ_n for a neural network with $K = 3$ layers. When viewed from the loss term ℓ_n, the output $\mathbf{x}^{(k)}$ from layer k (pick for instance $k = 2$) is a bottleneck of information for both the parameter vector $\mathbf{w}^{(k)}$ for 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).

order. The algorithm temporarily stores all the values $\mathbf{x}^{(k)}$, 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 reverse order while computing the derivatives in equation (4) first and then in equations (2) and (3), and 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)) 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 derivative with respect to $\mathbf{w}$ is read as a column vector of derivatives). 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 correlations 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 [8, 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 [6]. 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* [9, 11] starts from an initial value $\mathbf{w}_0$ chosen at random and iterates as follows:

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

Algorithm 1 Backpropagation

function $\nabla L_T \leftarrow \text{backprop}(T, \mathbf{w} = [\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(K)}], \ell)$
  $\nabla L_T = \text{zeros}(\text{size}(\mathbf{w}))$
  for $n = 1, \dots, N$ **do**
    $\mathbf{x}^{(0)} = \mathbf{x}_n$
    for $k = 1, \dots, K$ **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 $k = K, \dots, 2$ **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

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 risk gradient $\nabla L_T(\mathbf{w})$, 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 μ_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 $L_T(\mathbf{w})$ is small, as is the case around shallow minima. The value of μ_t is often varied according to some schedule like the one in Figure 4. 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 α is often fixed, and is a parameter of critical importance [12]. A rate that

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

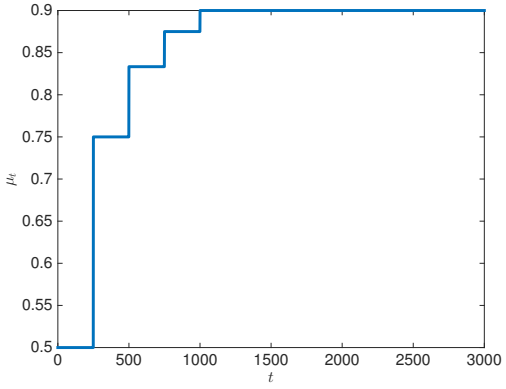

Figure 4: A possible schedule [11] for the momentum coefficient μ_t.

is too large leads to large steps that often overshoot, and a rate that is too small leads to very slow progress. In practice, an initial value of α is chosen by cross-validation to be some value much smaller than 1. Convergence can take between hours and weeks for typical applications, and the value of L_T is typically monitored through some user interface. When progress starts to saturate, the value of α is decreased (say, divided by 10).

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 $\mu_t \mathbf{v}_t - \alpha \nabla \ell_n(\mathbf{w}_t)$ (there is one such step for each training sample) is taken right away, rather than accumulated into the step $\mu_t \mathbf{v}_t - \alpha \nabla L_T(\mathbf{w}_t)$ (no subscript n here). In contrast, using the latter step 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, the online steps can follow very "curved" paths, whereas a single batch step can only move in a fixed direction in parameter space. 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 ∇L_T, 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 $|L_T(\mathbf{w}_t) - L_T(\mathbf{w}_{t-1})|$. 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. A different way to improve generalization, sometimes used in combination with early termination, is discussed in Section 5.

5 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 network 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 [10]. Given a deep network to be trained, a *dropout network* is obtained by flipping a biased coin for each node of the original network 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 network 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 network can be viewed as a different network, and the dropout method effectively samples a large number of nets efficiently.

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] T. M. Cover and J. A. Thomas. *Elements of Information Theory, 2nd Edition*. John Wiley and Sons, Inc., Hoboken, NJ, 2006.

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

[5] D. .J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, Cambridge, UK, 2003.

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

[7] R. J. McEliece. *The Theory of Information and Coding*. Addison-Wesley, Reading, MA, 1977.

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

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

[10] 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.

[11] 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.

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

A The Cross Entropy Loss and Information Theory

This Appendix discusses a connection between the cross entropy loss and concepts of probability and information theory. This link is a bit of a stretch for neural networks. However, at the very least, this connection explains the name of the loss function.

The discussion that follows is limited to binary and independent events, while the theory is much more general. In addition, our discussion is informal. Please refer to standard texts for more detail and proofs [3, 5, 7].

A.1 Coding for Efficiency

The concept of cross entropy has its roots in coding theory. Suppose that you repeatedly observe a binary experiment with possible outcomes 0 and 1. You want to transmit your observations of the outcomes to a friend over an expensive channel. The sequence of observations is assumed to be unbounded in length, so an initial cost of exchanging some communication protocol with your friend is acceptable, and is not part of the cost. However, you would like to define that protocol so that, once it is established, you then save on the number of bits sent to communicate the outcomes. Coding theory studies how to do this. The same considerations apply if instead of sending the information you want to store it on an expensive medium. Thus, coding theory is really at the heart of the digital revolution, as it is used extensively in both communications and computing (and more).

Assume that the *true* probability q that the outcome is 1 is known to both you and your friend. A naive transmission scheme would send a single bit, 1 or 0, for each outcome, so the cost of transmission would be one bit per outcome. It turns out that if $q = 1/2$ that's the best you can do.

However, if q has a different value, you can do better by *coding* the information before sending it, and your friend would then *decode* it at the other end of the channel.[3] This is obvious for extreme cases: If $q = 0$ or $q = 1$, you don't need to communicate anything.

Suppose now that q has some arbitrary value in the interval $[0, 1]$. There are very many ways of coding a stream of bits efficiently, and perhaps the simplest to think about is *Huffman coding*. Rather than sending one bit at a time, you send a sequence of *blocks* of m bits each.[4] There are $n = 2^m$ possible blocks $b_1, \ldots, b_n$ of m bits, and part of establishing the communication protocol is to prepare and share with your friend a table of n *codes* $c_1, \ldots, c_n$, one for each block. Each code is also a sequence of bits, but different codes can have different lengths, in contrast with blocks, which are all m bits long. Then, instead of sending block b_i, you look up its code c_i and send that. Your friend receives c_i and uses a reverse version of the same table to look up b_i. The whole magic is to build the tables so that *likely blocks get short codes*. As a result, you end up sending short blocks often and long blocks rarely, and on average you save.

Huffman codes are built with a very simple procedure, which is not described here.[5] Obviously, in order to do so, you need to know q, so you can figure out the probability of occurrence of each block.[6] Large blocks give you more flexibility, and the code becomes increasingly efficient (but harder to set up) as $m \to \infty$.

[3] In the storage application, you encode before you store, and you decode after you retrieve.

[4] This m has nothing to do with the number of parameters in a predictor!

[5] Huffman coding is fun and simple, and it would take you just a few minutes to understand, say, the Wikipedia entry on it.

[6] As stated earlier, we assume independent events, although the theory is more general than that.

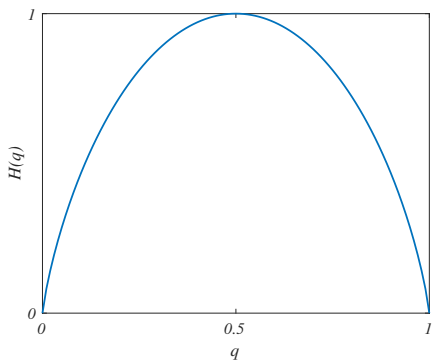

Figure 5: The entropy function.

A.2 Entropy

Claude Shannon, the inventor of information theory, proved that the best you can do, even using coding schemes different from Huffman's, is to use on average

$$H(q) \stackrel{\mathsf{def}}{=} \mathbb{E}[-\log_2 q] = -q \log_2 q - (1-q)\log_2(1-q)$$

code bits for each block bit you want to send, and called the quantity $-\log_2 q$ the *information* conveyed by a bit that has probability q of being 1 (and therefore probability $1-q$ of being 0). He called the statistical expectation $H(q)$ of the information the *entropy* of a source that emits a sequence of bits with that distribution.

The expression for entropy requires a small *caveat*: When $q = 0$ or $q = 1$, one of the two terms is undefined, as it is the product of zero and infinity. However, it is easy to use De l'Hospital rule to check that

$$\lim_{q \to 0} q \log_2 q = 0$$

so whenever we see that product with $q = 0$ we can safely replace it with 0. With this *caveat*, the entropy function looks as shown in Figure 5 when plotted as a function of q.

Note that $H(0) = H(1) = 0$, consistently with the fact that when both you and your friend know all the outcomes ahead of time (because q is either 0 or 1) there is nothing to send. Also

$$H(q) \leq 1 \ ,$$

meaning that you cannot do worse with a good coding scheme than without it, and $H(1/2) = 1$: When zeros and ones have the same probability of occurring, coding does not help. In addition, entropy is symmetric in q,

$$H(q) = H(1-q) \ ,$$

a reflection of the fact that zeros and ones are arbitrary values (so you can switch them with impunity, as long as everyone knows).

A.3 Cross Entropy

Cross entropy comes in when the table you share with your friend is based on the wrong probability. If the true probability is q but you think it's p, you pay a penalty, and Shannon proved that then the best you can do is to send on average

$$H(q,p) = -q\log_2 p - (1-q)\log_2(1-p)$$

code bits for every block bit. This quantity is the *cross entropy* between the two distributions: One is the true Bernoulli distribution $(1-q,q)$, the other is the estimated Bernoulli distribution $(1-p,p)$.

Cross entropy is obviously not symmetric,

$$H(q,p) \neq H(p,q) \quad \text{in general,}$$

as it is important to distinguish between true and estimated distribution. It should come as no surprise (but takes a little work with standard inequalities to show) that

$$H(q,p) \geq H(q) \quad \text{for all} \quad q,p \in [0,1]\,.$$

This reflects the fact that you need more bits if you use the wrong coding scheme. Therefore,

> We can view $H(q,p) - H(q)$ as the added average transmission cost of using probability distribution $(1-p,p)$ for coding, when the correct distribution is $(1-q,q)$.

This result holds more generally for arbitrary discrete distributions, not necessarily binary. If there are k possible outcomes rather than 2, the true probability distribution is $\mathbf{q} = (q_1,\ldots,q_k)$, and the estimated distribution is $\mathbf{p} = (p_1,\ldots,p_k)$, then the formulas for entropy and cross entropy generalize in the obvious way:

$$H(\mathbf{q}) = -\sum_{i=1}^{k} q_i \log_2 q_i \quad \text{and} \quad H(\mathbf{q},\mathbf{p}) = -\sum_{i=1}^{k} q_i \log_2 p_i\,.$$

A.4 The Cross Entropy Loss

In a binary classification with labels 0 and 1, given a data point $\mathbf{x}$, the corresponding true label y is either 0 or 1. This fact is certain, and we can express it probabilistically with an "extreme" distribution, depending on the value of y: If $y = 1$, then we can say that its "true Bernoulli distribution" is $(0,1)$, that is, $(1-q,q)$ with $q = 1$: The probability of y being 0 is 0, and the probability of y being 1 is 1. If $y = 0$, the situation is reversed: The probability of y being 0 is 1, and the probability of y being 1 is 0, so the true Bernoulli distribution is $(1,0)$. In either case, we have $q = y$, a fortunate consequence of our choice of names (0 and 1) for the two classes of the classification problem.

On the other hand, the value p returned by the softmax function can be interpreted as the Bernoulli distribution $(1-p,p)$: The score function's estimate of the probability that $y = 0$ is $1-p$, and the score function's estimate of the probability that $y = 1$ is p.

One way to quantify the cost of estimating that the distribution is $(1-p, p)$ while the true distribution is $(1-q, q)$ is to use the difference $H(q,p) - H(q)$. In light of our previous discussion, this effectively means that we reframe classification as a coding problem: Every time I make a mistake, I need to send an amendment, and that costs bits.

Of course, in classification we don't necessarily care about bits as a unit of measure, so we can use any base for the logarithm, not necessarily 2. In addition, since q is "extreme" (either 0 or 1) we can ignore $H(q)$, since, as we saw earlier, $H(0) = H(1) = 0$.

This argument therefore suggests using the cross entropy as a measure of loss. Since q and the true label y happen to have the same numerical value, we can replace q with y, which is a binary variable rather than a probability:

$$H(y,p) = -y \log p - (1-y)\log(1-p) = \begin{cases} \log p & \text{if } y = 1 \\ \log(1-p) & \text{if } y = 0. \end{cases}$$

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[7]. Let then

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

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

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,

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