Losses, Back-Propagation, and Convolutional Neural Networks

Carlo Tomasi

February 14, 2022

The zero-one loss was introduced in an earlier note as a desirable loss function for classification. Unfortunately, this loss cannot be used for training, as Section 1 will show, so that Section also introduces appropriate proxies for it. Section 2 then shows how to apply the chain rule for differentiation to compute the gradient of the empirical training risk with respect to the parameters of a neural network. Section 3 describes Convolutional Neural Networks (CNNs), a restricted form of deep network with a vastly reduced number of parameters. Finally, Section 4 sketches the state of the art in deep neural networks for image recognition.

1 A Loss for Classification

The empirical training risk $L_T(\mathbf{w})$ to be minimized to train a neural network is the average of the loss $\ell(y_n, \hat{y}_n)$ on a training set of input-output pairs

$$T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\} ,$$

where $\hat{y}_n$ is the output of the predictor when given input $\mathbf{x}_n$. The output y_n is categorical in a classification problem, and a real-valued vector in a regression problem.

In order to be able to use gradient descent methods for training, the risk $L_T(\mathbf{w})$ ought to be (i) differentiable and (ii) have a nonzero gradient almost everywhere.

The first requirement is technical and relatively minor, since the functions involved have discontinuities in their derivatives at most on a measure-zero subset of the domain $\mathbb{R}^m$. It can be proven that convergence is still possible even in the presence of these discontinuities.

The second requirement, on the other hand, is crucial. For a regression problem, the loss function is typically the quadratic loss,

$$\ell(y, \hat{y}) = \|y - \hat{y}\|^2 .$$

This loss is differentiable and has nonzero gradient $2(\hat{y} - y)$ with respect to its second argument $\hat{y}$, the output of the network.

The last stage of a K-class *classifier*, on the other hand, is the arg max function that picks the index of the highest soft-max score:

$$\hat{y} = \arg \max_{k \in \{1,\ldots,K\}} p_k$$

where

$$\mathbf{p} = (p_1, \ldots, p_K) = h(\mathbf{x}; \mathbf{w})$$

is the vector of K scores output by the network[1] for input $\mathbf{x}$.

Both the arg max and the zero-one loss

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

are piecewise-constant functions, and so is their composition

$$\ell_n(\mathbf{w}) \stackrel{\text{def}}{=} \ell_{0\text{-}1}(y_n, \hat{y}_n) = \begin{cases} 0 & \text{if } \arg\max_{k \in \{1,\dots,K\}} h(\mathbf{x}_n; \mathbf{w}) = y_n \\ 1 & \text{otherwise.} \end{cases}$$

As long as the changes in the weights $\mathbf{w}$ are small enough that the same output score remains the highest, the estimate $\hat{y}_n$ does not change and $\ell_n(\mathbf{w})$ remains constantly equal to either 0 or 1. A constant loss has zero gradient, and therefore provides no information on how to change the weights $\mathbf{w}$ to decrease the loss.

For example, suppose that the loss $\ell_n(\mathbf{w})$ is 1 for some n and for the current weight vector $\mathbf{w}$. This means that the highest score the network outputs for input $\mathbf{x}_n$ is not p_{y_n}, the score for the true label y_n, but rather some other score p_j. We would then need to know in what direction to change the weight vector $\mathbf{w}$ so as to flip the ordering of p_{y_n} and p_j: Either make p_{y_n} greater, or make p_j smaller, or both. However, an infinitesimal change of $\mathbf{w}$ will almost never change this ordering. Therefore, the gradient of $\hat{y}_n$ with respect to $\mathbf{w}$ is zero almost everywhere zero, and so is the gradient of $\ell_n(\mathbf{w})$: The optimization algorithm almost never receives useful information for training.

To address this issue, a differentiable loss defined directly on the output $h(\mathbf{x}; \mathbf{w})$ of the network, without the arg max, is used as a proxy for the zero-one loss defined on classification output $\hat{y} = \arg\max_k h(\mathbf{x}; \mathbf{w})$. Specifically, the multi-class cross-entropy loss is used, and is discussed next. For pedagogical clarity, this loss is first introduced for a two-class classifier, and then generalized to the multi-class case. Of course, the multi-class loss for $K = 2$ is equivalent to the two-class loss.

1.1 The Two-Class Cross-Entropy Loss

If there are only two classes, $K = 2$ and $Y = \{0, 1\}$, then the network h outputs a score $\mathbf{p} \in \mathbb{R}^2$. Because of the normalization performed by the soft-max function, which is the last stage of h when h is a classifier, only one of the two values needs to be recorded, say, the score for the hypothesis that $y = 1$. Call that value p. The score for the hypothesis that $y = 0$ is then $1 - p$, since the (positive) outputs of the soft-max function add up to one. Thus, we look for a differentiable function of y and p to stand for $\ell_{0\text{-}1}(y, \hat{y}(p))$.

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 $\hat{y}$ is a piecewise-constant function of p, and therefore 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 suitable loss function in terms of that:

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

The substitution of $\hat{y}$ with p 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

[1] We do not consider the arg max operator to be part of the network $h(\mathbf{x}; \mathbf{w})$ itself in this note: The network just outputs scores, one per class.

with label 0. We say that p is a *relaxation* of $\hat{y}$ and define the loss on the relaxed label estimate instead of on the label estimate itself.

The rest of this Section defines a suitable proxy loss function called the cross-entropy loss. Appendix A establishes a connection between this measure of loss and probabilistic and information-theoretic considerations. You may or may not be persuaded by this connection.

More pragmatically, the cross-entropy loss was introduced first for a type of classifier called a *linear classifier*. It can be proven that the composition of soft-max and cross-entropy is convex, and the problem of training a linear classifier based on this composition is a convex optimization problem: All solutions have a unique minimum risk. We will not use this fact here, since the output of a neural network is not a convex function of the network's parameters.

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 as follows:

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

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.[2]

Figure 1 shows the function. The domain is the cross product

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

that is, two segments on the plane (left panel 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 y, a bigger and bigger cost is incurred, and as $p \to 0$ the cost grows to infinity.

1.2 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 $K = 2$:

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

[2] In Appendix A, the logarithm base 2 is used, as is customary in information theory.

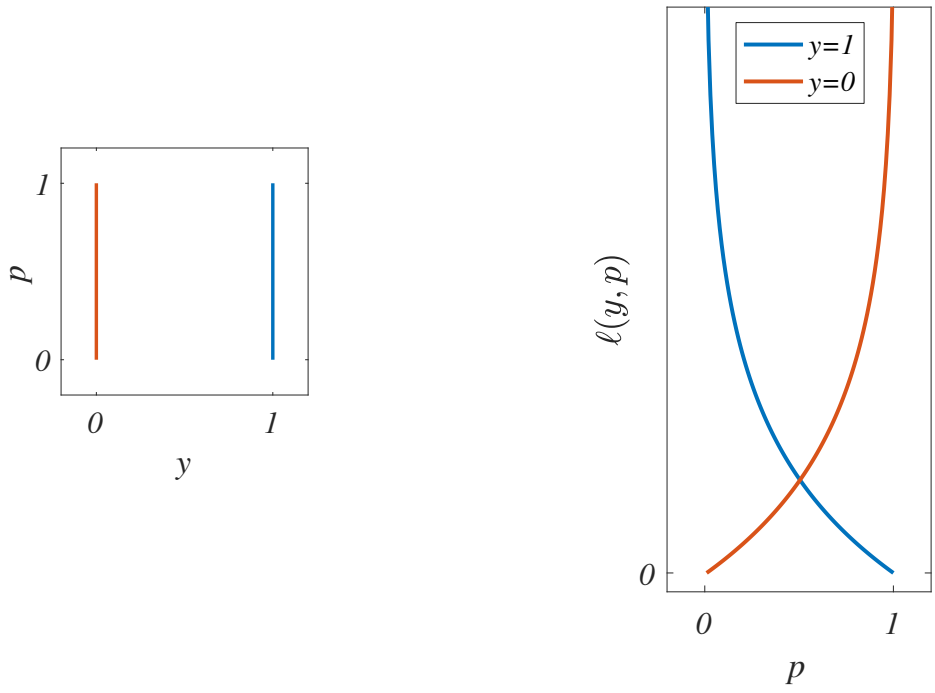

Figure 1: (Left) The two colored segments are the domain of any loss function for a binary classifier based on softmax scores. (Right) The cross-entropy loss. The two curves are defined on the segments of corresponding colors on the left. They asymptote to infinity for $p \to 0$ (when $y = 1$, blue)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

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 above 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 $K = 2$.

2 Back-Propagation

If the cross-entropy loss is used, the empirical risk L_T is a piecewise-differentiable function with nonzero gradient almost everywhere. Since the risk is the average of many loss terms, Stochastic Gradient Descent (SGD) can be used to find a weight vector $\mathbf{w}$ with low empirical training risk.

As usual, the training risk is defined as the average loss over the training set:

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

SGD starts with some *initial value* $\mathbf{w}_0$ for $\mathbf{w}$, and then at step $t = 1, 2, \ldots$ walks in the direction opposite to the gradient

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

computed on a mini-batch. The computation of this gradient is called *back-propagation* and is a straightforward implementation of the chain rule for differentiation. The related bookkeeping is described next.

The computation of the n-th loss term $\ell_n(\mathbf{w})$ from the input data sample $\mathbf{x}_n$ can be rewritten as follows for a network with J layers:

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

where $(\mathbf{x}_n, y_n)$ is the n-th training sample and $h^{(j)}$ describes the function implemented by layer j of the network h. Thus, the first three lines above describe the neural network (up to and including the computation of the score $\mathbf{p}$, and with a soft-max as the last layer), and the last one describes the loss computation.

Computation of the derivatives of the loss term $\ell_n(\mathbf{w})$ can be understood with reference to Figure 2. The term ℓ_n depends on the parameter vector $\mathbf{w}^{(j)}$ for layer j through the output $\mathbf{x}^{(j)}$ from that layer and nothing else, so that we can write

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

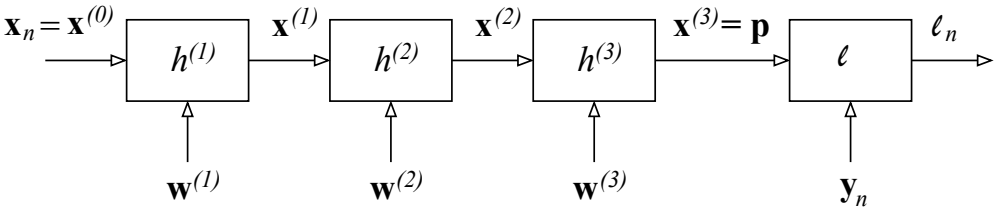

Figure 2: Example data flow for the computation of the loss term ℓ_n for a neural network with $J = 3$ layers. When viewed from the loss term ℓ_n, the output $\mathbf{x}^{(j)}$ from layer j (pick for instance $j = 2$) is a bottleneck of information for both the parameter vector $\mathbf{w}^{(j)}$ for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

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

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

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

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

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_j be the dimensionality (number of entries) of $\mathbf{x}^{(j)}$, and e_j be the dimensionality of $\mathbf{w}^{(j)}$. The two matrices

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

are the *Jacobian matrices* of the layer output $\mathbf{x}^{(j)}$ 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 Appendix B.

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). 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 $\mathbf{x}^{(j)}$ for that sample and for all $j = 1, \ldots, J$, in this order. The algorithm temporarily stores all the values $\mathbf{x}^{(j)}$, 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 J 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 the risk $L_T(\mathbf{w})$ is the average (from equation (1)) of the gradients computed for each of the samples:

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

(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 through the samples in the training set. 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.

Algorithm 1 Backpropagation

function $\nabla L_T \leftarrow \text{backprop}(T, \mathbf{w} = [\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(J)}], \ell)$
  $\nabla L_T = \text{zeros}(\text{size}(\mathbf{w}))$
  for $n = 1, \ldots, N$ **do**
    $\mathbf{x}^{(0)} = \mathbf{x}_n$
    for $j = 1, \ldots, J$ **do** ▷ Forward propagation
      $\mathbf{x}^{(j)} \leftarrow h^{(j)}(\mathbf{x}^{(j-1)}, \mathbf{w}^{(j)})$ ▷ 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
    $\mathbf{g} = \frac{\partial \ell(y_n, \mathbf{x}^{(J)})}{\partial \mathbf{p}}$ ▷ $\mathbf{g}$ is $\frac{\partial \ell_n}{\partial \mathbf{x}^{(j)}}$
    for $j = J, \ldots, 2$ **do** ▷ Back-propagation
      $\nabla \ell_n \leftarrow [\mathbf{g}\frac{\partial \mathbf{x}^{(j)}}{\partial \mathbf{w}^{(j)}}; \nabla \ell_n]$ ▷ Derivatives are evaluated at $\mathbf{w}^{(j)}$ and $\mathbf{x}^{(j)}$
      $\mathbf{g} \leftarrow \mathbf{g}\frac{\partial \mathbf{x}^{(j)}}{\partial \mathbf{x}^{(j-1)}}$ ▷ Ditto
    end for
    $\nabla L_T \leftarrow \frac{(n-1)\nabla L_T + \nabla \ell_n}{n}$ ▷ Accumulate the average
  end for
end function

3 Convolutional Neural Networks

A neuron matches the input $\mathbf{x}$ to a pattern $\mathbf{v}$ and matches it to a threshold ($-b$, the negative of the bias). What should the patterns in an image recognition system be? One could make $\mathbf{x}$ be the entire image, with its pixels strung into a vector, and then $\mathbf{v}$ could be an image (in vector form as well) of the object to be recognized. Except for normalizations, this is the idea of correlation, and may work for simple patterns that have consistent appearance in all images.

However, this network would not work well for, say, your grandmother's face, which could show up in images that look very different from $\mathbf{v}$ because of viewpoint, lighting, facial expression, other objects or people in the image, and other causes of discrepancy. In other words, it is unlikely that a single "grandmother neuron" could be made to work.

Instead, observe that faces typically have eyes, noses, ears, hair, and wrinkles—especially for an older person. These features can be analyzed in turn in terms of image edges, corners, curved

segments, small dark regions, and so forth. This suggest building a *hierarchy* of patterns, where higher-level ones are made of lower level ones, and only the lowest-level patterns are made directly out of pixels from the input image. At each level, each pattern should then take only a relatively small number of inputs in consideration, from a relatively small and compact part of the image: each neuron should have a *local input*.

In addition, many of the lower-level features appear multiple times in images and across objects, and this suggests that the same neuron could score patterns of its own type no matter where they appear in an image: the same detector could be *reused* over its domain.

These notions of hyerarchy, locality and reuse suggest the following structure for a neural-net layer [4, 5].

- Think of the input $\mathbf{x}$ as a two-dimensional array, one entry per pixel, rather than a vector, so that the notion of locality is more readily expressed.

- Group activations (layer outputs) into m *maps*: each map takes care of one type of pattern (reuse) through a separate, small (locality) convolution kernel.[3] The pattern encoded by the kernel coefficients is sometimes called a *feature*, so the m maps are called *feature maps*. The (common) activation function ρ is then applied to each feature map, entry by entry.

- To allow activation values to depend on large parts of the image, several layers of activations, separated by non-linearities, are chained, and the sizes of feature maps in subsequent layers are typically made increasingly smaller by sampling-like techniques such as *stride* or *max-pooling*, described later on. As will be explained in more detail below, this reduction in feature-map size increases what is called the *receptive field* of activations in later layers of the network. The receptive field is the size of the area of the input image that affects a particular activation value.

There are three consequences to this chaining of layers and increase of receptive field: (i) Activations in later layer capture the contents of larger areas of the image, so that different layers effectively analyze the image at different scales: The neural network is a *feature pyramid*, rather than an image pyramid. (ii) Activations in later layers are less sensitive to image detail and provide increasingly *abstract* representations of the input. (iii) Activation maps in later layers are smaller, a factor that reduces the *computational complexity* of the network beyond the reduction afforded by the convolutional structure of the layers.

Stride In a standard convolution, the kernel is slid over the entire input. However, any given output entry is not very different from neighboring output entries, because images often vary slowly as a function of image coordinates. To reduce the redundancy in the output, the values of convolutions or other local operators are often computed with a *stride* s greater than one. That is, only one value every s is computed in each dimension, thereby reducing the size of the output by a factor of about s in every dimension.

A strided convolution is equivalent to a standard convolution followed by retaining every s-th sample in each direction. However, it would be wasteful to first compute all values and then ignore a large fraction of them by sampling. Instead, strided convolutions are implemented by only computing the required outputs.

[3] As we know, convolutions and correlations are equivalent.

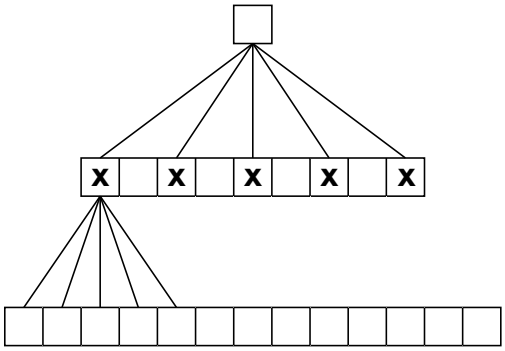

Figure 3: Receptive field of a single output from the cascade of a strided convolution (bottom) and a regular convolution (top) in one dimension. Each activation retained by striding (each cross) depends on $k_1 = 5$ input values (bottom) in each dimension. The output at the top depends on $k_2 = 5$ values (the crosses), but because of striding these values span a total of $s(k_2 - 1) + 1 = 2(5 - 1) + 1 = 9$ outputs from the first convolution. The overall receptive field of each output of the cascade is $(4 \cdot 2 + 5)^2 = 13^2 = 169$ pixels.

Max-Pooling If the input to a layer is divided into squares of $s \times s$ activations (or, for the first layer, image pixels), then a strided convolution would retain, say, the top-left value of the complete convolution from each square. In contrast, *max-pooling* following a convolution retains the *largest* output activation in each square. The rationale is that the largest activation may be the more salient, and provides richer information than the other activations.

A convolution followed by max-pooling is significantly more expensive than a convolution with the same stride. This is because max-pooling requires computing the entire (un-strided) convolution in order to determine which activation in each $s \times s$ square is the largest. It is not clear that the slight improvement in the saliency of the outputs justifies the larger cost of max-pooling over strided convolutions. This may be why max-pooling, while pervasive in the early days of deep learning, has become less popular in recent years.

Receptive Field Example Consider the following cascade:

- A convolution with a $k_1 \times k_1$ kernel and stride s.
- A convolution with a $k_2 \times k_2$ kernel.

Assume that both k_1 and k_2 are odd.

The calculation of the receptive field of an activation is most easily done in one dimension. The results are then squared (assuming square kernels and strides) to obtain the overall receptive field in pixels. Figure 3 illustrates the one-dimensional case with $k_1 = k_2 = 5$ and $s = 2$.

Each output from the second convolution depends on k_2 input activations in each dimension. These activations are spaced s apart, and therefore span $s(k_2 - 1) + 1$ input activations in each dimension. Each of those activations in turn depends on k_1 activations at the input of the first convolution, so the number of such activations increases by $k_1 - 1$ values.

Thus, the one-dimensional receptive field of the outputs of the second convolution is

$$r = (k_2 - 1)s + 1 + k_1 - 1 = (k_2 - 1)s + k_1$$

and the receptive field in two dimensions is r^2 pixels.

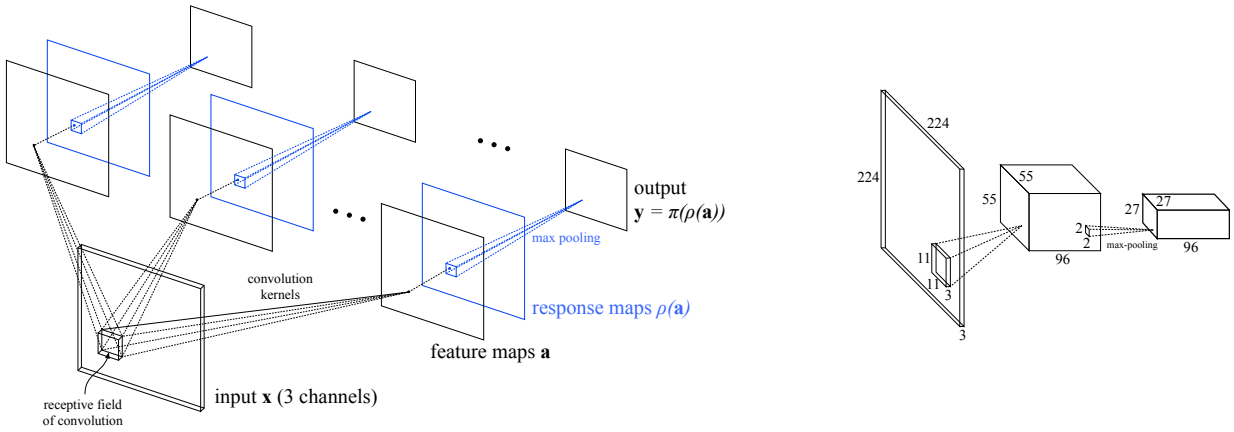

Figure 4: (Left) The structure of a convolutional neural-net layer for a color image with three channels (red, green, blue). (Right) In the literature, neural nets with many layers are drawn with each layer shown in a more compact way than on the left, although there is no standard format. Typically, the maps are stacked in a block, as shown here, rather than drawn side-to-side. Sometimes, max-pooling is only mentioned and not shown explicitly.

Convolutions Reduce the Number of Weights The *convolutional organization* described above is illustrated in Figure 4 for the input layer of a network from the literature [4]. This layer includes max-pooling. As a result of this structure, the number of distinct parameters in the layer drops dramatically when compared to that for fully connected layers. In a *fully connected* layer, each input scalar is connected through a trainable gain to each output scalar, as we have seen for generic layers so far. With a $n \times n \times 3$ array (a square color image) as input and m output feature maps of dimension $n \times n$ as output there would be $3mn^4$ gains and mn^2 bias terms (one bias per output scalar).

With m convolution kernels (m feature maps), on the other hand, the number of parameters is $(3k^2 + 1)m$ if all the kernels are $k \times k$, and if the single bias term added for each feature map is counted as well. For color $n \times n$ images and m feature maps of size $n \times n$, the count drops from $O(mn^4)$ weights for a fully connected layer to $m(3k^2 + 1)$ weights for a convolutional one.

For instance [4], for a $224 \times 224 \times 3$-pixel color image and 96 maps with an 11×11 receptive field, the drop is from more than 725 billion parameters to a mere $96 \times (3 \times 11^2 + 1) = 34{,}944$ parameters. More specifically, in that example, illustrated in Figure 4, the first layer of a convolutional net has a 224×224 color image as input, so that the input dimensionality is $d = 224^2 \times 3 = 150{,}528$. There are 96 kernels, feature maps, and response maps. The convolution kernels have receptive fields of size $11 \times 11 \times 3$ pixels (each pixel is a color), so each output scalar is computed from $3 \times 11^2 = 363$ input scalar values which are combined through 363 gains and one bias.

The original paper [4] does not describe the padding policy used for the convolutions. With the "same" style in the two image dimensions and a stride of 4 in each image dimension, each feature map should have width and height equal to $\lfloor 224/4 \rfloor = 56$. However, the paper reports feature maps that are 55×55 pixels each.[4] Max-pooling uses 3×3-pixel receptive fields and a stride of 2 pixels,

[4] A "valid" convolution with stride 4 would yield a feature map smaller than 55×55, and a "full" convolution would yield a feature map greater than 56×56. We cannot explain the discrepancy of one pixel between our calculation and the paper's.

and produces output maps of size 27×27 pixels.

On the other hand, each $11 \times 11 \times 3$ kernel must produce exactly one output feature map, so the convolution style in the channel dimension must be the "valid" style. Feature maps are also called *output channels* in the literature.

The set of activation maps computed by the 96 kernels is a $55 \times 55 \times 96$ block, and max pooling with stride 2 is applied to each of the 96 slices in this block to produce a $27 \times 27 \times 96$ output block from this layer. Thus, for the layer in the Figure (including max-pooling), the output dimensionality is $e = 27^2 \times 96 = 69{,}984$, a bit less than half of the input dimensionality $d = 224^2 \times 3 = 150{,}528$. On the other hand, the map *resolution* decreases more than eightfold, from 224 to 27 pixels on each side. The representation of the image has become more abstract, changing from a pure pixel-by-pixel list of its colors to a coarser map of how much each of 96 features is present at each (coarse) location in the image and in each color channel.

4 The State of the Art of Image Classification

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual competition that pits image recognition and object detection algorithms against each other on a standardized image database that contains 1.4 million images divided into 1,000 object classes. Designing the database and the various aspects of the competition is a significant research effort in its own right [8]. The competition includes three tasks:

Image Classification: The database maintainers label each image manually with the presence of *one* instance out of 1000 object categories, even if more objects are present in the image. These annotations are called the *ground truth*. Competing classification algorithms return a *list* of up to five objects for each test image, and the list is deemed correct if it includes the ground-truth label. Algorithms are trained on a subset of 1.2 million images for which the ground-truth annotations are made public, and about 50,000 images with annotations are made public for validation. The competition organizers withhold a set of 100,000 images to test the algorithms submitted by the contestants, and measure the *error rate* as the fraction of incorrect outputs.[5]

Single-Object Localization: In addition to image classification, prediction algorithms also provide an axis-aligned rectangular bounding box around *one* instance of the recognized object class. The box is correct if the area of the intersection of computed and true box is at least 1/2 of the area of their union. As above, a list of up to five object-box pairs is returned.

Object Detection: Same as single-object localization, but *every* instance of the object class is to be found exactly once. Missing, duplicate, and false detections are penalized.

Winners of the competition for each task are announced and invited to contribute insights at either the International Conference on Computer Vision (ICCV) or the European Conference on Computer Vision (ECCV) in alternate years.

Classification, detection, and localization go hand-in-hand: To classify an image one must first detect and localize an object that is deemed to be "of interest," and a correctly detected and

[5] Other measures of error have been proposed in the past and are given as secondary measures in current challenges as well.

well localized object is of course easier to classify correctly. Conversely, a detector often works by running a classifier on all or many windows in the image. This note only looks at performance figures in image classification, with the *caveat* that the best systems often perform well in more than one category, because of these interdependencies.

The classification task is very difficult for at least the following reasons:

- Images are "natural," that is, they are not contrived for the database but rather downloaded from photo sharing sites like Flickr. Because of this, the objects of interest are on arbitrary backgrounds. They may appear in very different sizes, viewpoints, and lighting conditions in different images, and may be only partially visible.
- There are 1,000 categories, and distinctions between categories are often very subtle. For instance, Siberian husky and Eskimo dog are different categories, but dogs of the two breeds look very similar.
- At the same time, variations of appearance within one category can be significant. (How many lamps can you think of?)
- What part of the image is of interest to the labeler may be based on very high-level semantic considerations that a machine learning algorithm may not have access to. For instance, a picture of a group of people examining a fishing rod was labeled as "reel" because everyone seemed intent on looking at the rod's line reel.

Because of these difficulties, the image classification error was 28.2 percent in 2010, even after many years of research and experimentation with smaller databases. The winner of the 2017 challenge, on the other hand, achieved a 2.3% error rate [3]. This dramatic, more than tenfold improvement was achieved through the use of a sophisticated deep convolutional network. As a reference point, the winning architecture returns the ensemble prediction of several networks, each with millions of parameters. The change from 17% to 2.3% from a 2012 system [4] was not just achieved by increasing the number of layers, but by several architectural insights developed over the five years between the two architectures. These include residual networks [2] and so-called squeeze-and-excitation networks [3]. We will look at the basics of deep convolutional neural networks, but there is no room in a one-semester course to explore all the nuances of these architectures. Please read the papers if you are interested.

References

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

[2] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[3] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *arXiv preprint arXiv:1709.01507*, 2017.

[4] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, pages 1106–1114, 2012.

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

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

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

[8] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, pages 1–42, April 2015.

Appendices

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 [1, 6, 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.[6] 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.[7] There are

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

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

$n = 2^m$ 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.[8] Obviously, in order to do so, you need to know q, so you can figure out the probability of occurrence of each block.[9] Large blocks give you more flexibility, and the code becomes increasingly efficient (but harder to set up) as $m \to \infty$.

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).

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

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

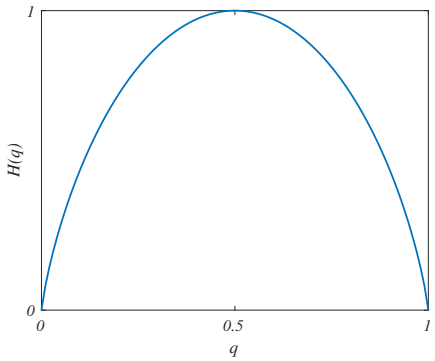

Figure 5: The entropy function.

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:

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

B The Local Jacobians for Back-Propagation

If $h^{(j)}$ 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)

$$\frac{\partial x_u^{(j)}}{\partial W_{qv}^{(j)}} = \delta_{uq} \frac{dh^{(j)}}{da_u^{(j)}} \tilde{x}_v^{(j-1)} \quad \text{and} \quad \frac{\partial x_u^{(j)}}{\partial x_v^{(j-1)}} = \frac{dh^{(j)}}{da_u^{(j)}} W_{uv}^{(j)} \, .$$

The Kronecker delta

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

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

$$\frac{dh^{(j)}}{da_u^{(j)}} \quad \text{is shorthand for} \quad \left. \frac{dh^{(j)}}{da} \right|_{a=a_u^{(j)}} ,$$

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

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

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

For the ReLU activation function followed by max-pooling, $h^j(\cdot) = \pi(h(\cdot))$, on the other hand, the value of the output at index u is computed from a window $P(u)$ of activations, and only one of the activations (the one with the highest value) in the window is relevant to the output[10]. Let then

$$p_u^{(j)} = \max_{q \in P(u)} (h(a_q^{(j)}))$$

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

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

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

$$\frac{\partial x_u^{(j)}}{\partial W_{qv}^{(j)}} = \delta_{q\hat{q}} \frac{dh^{(j)}}{da_{\hat{q}}^{(j)}} \tilde{x}_v^{(j-1)} \quad \text{and} \quad \frac{\partial x_u^{(j)}}{\partial x_v^{(j-1)}} = \frac{dh^{(j)}}{da_{\hat{q}}^{(j)}} W_{\hat{q}v}^{(j)} .$$

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