COMPSCI 527 — Homework 4

Due on October 29, 2015

Work on this assignment either alone or in pairs. You may work with different partners on different assignments, but you can only have up to one partner for any one assignment. You may not talk about this assignment with others until all of you have handed in their work. See Mechanics→Homework on the class web page for details on the homework policy.

Hand in your work as explained in the instructions for homework 1 (of course, change `hw1` to `hw4`).

1. A multi-layered neural net can be represented in MATLAB with an array `net` of data structures, where each element encodes a layer and has fields

- `gain`: a $D_{\text{out}} \times D_{\text{in}}$ matrix of gains (or weights), where D_{in} is the dimensionality of the input and D_{out} is the number of activations in the layer.
- `bias`: a $D_{\text{out}} \times 1$ column vector of biases.
- `h`: a handle to a function that computes the nonlinearity for that layer.

It is convenient to compute the outputs that the network returns on several inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$ simultaneously. In that case, the inputs are collected in a single $D \times N$ array `X`, where D is the dimensionality of the input. Because of this, the function whose handle is `h` should be prepared to take a matrix as input. The effect of calling that function on a matrix of activations should be to apply the function to each entry of the matrix and return the matrix of the results (same size as the input).

(a) Write a concise MATLAB function with header

```
function X = nn(X, net)
```

that takes such a matrix `X` of inputs and a multi-layered neural net `net` as described above and returns a matrix whose columns are the outputs of the net for the inputs in `X`. **Hand in your code. Here and elsewhere, do not cut-and-paste, but rather use the `VerbatimInput` command. For instance,**

```
\VerbatimInput[fontsize=\small,fontshape=n,xleftmargin=15mm]{nn.m}
```

(b) Write a MATLAB function with header

```
function net = approximator(f, T)
```

that takes a handle `f` of a function from the real interval $[0, 1]$ to the reals with header

```
function y = f(x)
```

and can take a vector `x` of inputs and simultaneously compute the vector `y` of corresponding outputs. The second argument to `approximator` is a sampling period $0 < \texttt{T} \leq 1$. The function `approximator` returns a two-layer network that computes a piecewise linear approximation of `f` using the ReLU nonlinearity in the first layer and the identity in the second. There are K linear pieces, defined on regularly spaced intervals of width at most `T`. The first sample should be at $x = 0$ and the last at $x = 1$. The commands

```
N = 101;
T = 0.05;
f = @(x) exp(x) .* sin(3* pi * x);
net = approximator(f, T);
x = linspace(0, 1, N);
y = nn(x, net);
clf
plot(x, f(x), 'b')
hold on
plot(x, y, 'r')
```

should display the actual function `f` in blue and its piecewise linear approximation in red. **Hand in your code for `approximator` and the diagram produced by the commands above.** Your diagram should have a legend that distinguishes the two plots and a title or caption that states how many weights (including both gains and biases, regardless of whether they are zero or not) are in your network.

(c) Write the simplest possible formula for the number n_w of weights as a function of the number K of linear pieces in the approximation of `f`.

2. Let us for now restrict our attention to *convolutional* neural nets. For inputs, we still only consider one-dimensional signals, that is, vectors $\mathbf{x} \in \mathbb{R}^m$, rather than images. Convolution kernels are then one-dimensional as well, but there can be several kernels in each layer. A feature map in such a net is obtained through the following computation:

$$\mathbf{y}(\mathbf{x}) = h(\mathbf{a}(\mathbf{x})) \quad \text{where} \quad \mathbf{a}(\mathbf{x}) = \mathbf{x} \ *_v \ \mathbf{k} + b$$

where the input $\mathbf{x}$ is a column vector of m real numbers, the convolution kernel $\mathbf{k}$ is a column vector of n real numbers, the symbol '$*_v$' (obtained in a LaTeX math environment with the command `\ast_v`) denotes convolution computed with the MATLAB `'valid'` option, the scalar b is the map's bias and is added to each component of $\mathbf{x} \ *_v \ \mathbf{k}$, and the activation function h is a nonlinear function from $\mathbb{R}$ to $\mathbb{R}$ applied to each element of the activation $\mathbf{a}(\mathbf{x})$. The activation map $\mathbf{a}(\mathbf{x})$ is a column vector of the same length p as the output feature map $\mathbf{y}(\mathbf{x})$.

We make one modification to the meaning of the `'valid'` option relative to its MATLAB meaning, to preserve commutativity of convolution. Specifically, if vector $\mathbf{c}$ is no longer than vector $\mathbf{d}$, then

$$\mathbf{c} \ *_v \ \mathbf{d} \quad \text{corresponds to} \quad \texttt{conv}(\mathbf{d}, \mathbf{c}, \texttt{'valid'}) \quad \text{in MATLAB.}$$

In words, the longer vector comes first when the `'valid'` option is evaluated in `conv` (or `convn`).

Given a training sample $(\mathbf{x}_n, \mathbf{y}_n)$ where $\mathbf{x}_n$ is an input and $\mathbf{y}_n$ is a feature, the error of the feature map is a nonnegative real number computed as

$$e = \mathcal{L}(\mathbf{y}_n, \mathbf{y}(\mathbf{x}_n))$$

where $\mathcal{L} : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}^+$ is a loss function. For simplicity in this and similar expressions in the future we drop the subscript n from e. Assume both $\mathcal{L}$ and h to be differentiable for the purpose of this exercise.

(a) What is p as a function of m and n? Here and elsewhere in this problem, assume that $m \geq n$.

(b) When computing gradients by back-propagation, we need the derivatives of the error e with respect to the input $\mathbf{x}$ and the parameters $\mathbf{k}$ and b of any given layer. Let

$$e_{\mathbf{a}} = \begin{bmatrix} \frac{\partial e}{\partial a_1} \\ \vdots \\ \frac{\partial e}{\partial a_p} \end{bmatrix}$$

be the column vector that gathers the derivatives of the error e with respect to the entries of the activation $\mathbf{a}$, so that the column vector that collects the derivatives of e with respect of $\mathbf{x}$,

$$e_{\mathbf{x}} = \begin{bmatrix} \frac{\partial e}{\partial x_1} \\ \vdots \\ \frac{\partial e}{\partial x_m} \end{bmatrix}$$

can be written as follows by applying the chain rule of differentiation:

$$e_{\mathbf{x}} = A_{\mathbf{x}}^T e_{\mathbf{a}} \ .$$

In this expression, the $p \times m$ matrix

$$A_{\mathbf{x}} = \begin{bmatrix} \frac{\partial a_1}{\partial x_1} & \cdots & \frac{\partial a_1}{\partial x_m} \\ \vdots & & \vdots \\ \frac{\partial a_p}{\partial x_1} & \cdots & \frac{\partial a_p}{\partial x_m} \end{bmatrix}$$

is called the *Jacobian* of the activation $\mathbf{a}$ with respect to the input $\mathbf{x}$.

Show by example with $m = 7$ and $n = 3$ that

$$e_{\mathbf{x}} = e_{\mathbf{a}} \ *_f \ \rho(\mathbf{k})$$

where the symbol '$*_f$' denotes convolution computed with the MATLAB `'full'` option and $\rho(\mathbf{k})$ denotes the convolution kernel $\mathbf{k}$ with its entries listed in reverse order (in MATLAB, this would be `k(end:-1:1)`).

Specifically, let

$$\mathbf{x} = [x_1, \ldots, x_7]^T \quad , \quad \mathbf{k} = [k_1, \ldots, k_3]^T \quad \text{and} \quad \mathbf{a} = [a_1, \ldots, a_p]^T$$

(for a suitable value of p) and write out all the entries of the '`valid`' convolution matrix $C_v(\mathbf{k})$ such that

$$C_v(\mathbf{k})\,\mathbf{x} = \mathbf{x}\ *_v\ \mathbf{k}$$

as you did in a previous homework assignment. Then write out all the entries of the '`full`' convolution matrix $C_f(\rho(\mathbf{k}))$ such that

$$C_f(\rho(\mathbf{k}))\,\mathbf{a} = \mathbf{a}\ *_f\ \rho(\mathbf{k})$$

and conclude by showing how these matrices tie together the two expressions for $e_{\mathbf{x}}$ given above. Leave zero entries blank in your matrices.

(c) For any matrix or column vector $\mathbf{v}$, let

$$\mathbf{s} = \sigma_r(\mathbf{v}, s) = \mathbf{v}(1 : s : \texttt{end}, :)$$

be a row-sampling operator. The integer argument $s \geq 1$ is called the *stride*, and the output vector $\mathbf{s}$ retains all entries that are in rows $1, 1+s, 1+2s, \ldots$ in $\mathbf{v}$. For instance,

$$\sigma_r(\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}, 3) = \begin{bmatrix} 1 \\ 4 \\ 7 \end{bmatrix}$$

A feature map with stride s is then equivalent to the following computation:

$$\mathbf{y}(\mathbf{x}) = h(\sigma_r(\mathbf{a}(\mathbf{x}), s))$$

where the meaning of h and $\mathbf{a}$ is the same as before.[1]

Continue the example from part (b) with the same values of m and n and with $s = 2$ to verify that if stride is included then $e_{\mathbf{a}}$ is now shorter, and

$$e_{\mathbf{x}} = \delta(e_{\mathbf{a}}, p)\ *_f\ \rho(\mathbf{k})$$

where $\mathbf{d} = \delta(\mathbf{u}, p)$ takes a vector $\mathbf{u}$ of length $q \leq p$ and *dilutes* it into a vector $\mathbf{d}$ of length p that has the entries $u_1, \ldots, u_q$ in positions $1, 1+s, 1+2s, \ldots, 1+(q-1)s$ where

$$s = \left\lceil \frac{p}{q} \right\rceil .$$

The vector $\mathbf{d}$ is elsewhere zero. For instance, if

$$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

then if $p = 8$ we have $s = \lceil 8/3 \rceil = 3$ and

$$\delta(\mathbf{u}, 8) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ 3 \\ 0 \end{bmatrix} .$$

3. Let us extend and consolidate what we learned in the previous problem, and relate it to back-propagation. Switching the role of $\mathbf{x}$ and $\mathbf{k}$ in the results you verified in the previous problem and invoking the commutative nature of convolution yields a similar relationship for the gradient with respect to $\mathbf{k}$ for stride s:

$$e_{\mathbf{k}} = \mu(\delta(e_{\mathbf{a}}, p)\ *_f\ \rho(\mathbf{x}), n)\ .$$

[1] It would be inefficient to compute $\mathbf{y}$ in this way, because one would first compute all values of $\mathbf{a}$ and then discard $s-1$ out of every s. Nonetheless, the mathematical expression above describes the input/output relationship correctly. Also, in this assignment we can afford this inefficiency.

The function $\mu(\mathbf{u}, n)$ takes a column vector $\mathbf{u}$ whose length q is an even number plus n and returns a column vector that contains the n elements in the middle of $\mathbf{u}$. The convolution that is input to this function has length

$$q = p + m - 1 = m - n + 1 + m - 1 = 2m - n = 2(m - n) + n$$

and therefore satisfies the requirement above. We are now ready to put everything together into the formulation of a back-propagation algorithm for convolutional nets.

Superscripts in the back-propagation equations simplify if we call $\mathbf{x}^{(\ell)}$ the input to layer ℓ and $\mathbf{y}^{(\ell)}$ the output to layer ℓ. In addition, layer ℓ can have $j^{(\ell)}$ kernels and biases, so that the feature maps $\mathbf{x}^{(\ell)}$ and $\mathbf{y}^{(\ell)}$ are in general matrices with $j^{(\ell)}$ columns (one column per feature map), and so are the activation maps $\mathbf{a}^{(\ell)}$. We then use subscript (ℓ, j) to denote the j-th kernel, bias, feature map, or activation map in layer ℓ. If we assume for simplicity that input $\mathbf{x}$ and output $\mathbf{y}$ of the net as a whole are vectors, we then have

$$\begin{aligned} \mathbf{x}^{(1,1)} &= \mathbf{x} \\ \mathbf{y}^{(\ell,j)} &= \mathbf{x}^{(\ell+1,j)} \quad \text{for} \quad \ell = 1, \ldots, L-1 \quad \text{and} \quad j = 1, \ldots, j^{(\ell)} \\ \mathbf{y}^{(L,1)} &= \mathbf{y} \end{aligned}$$

as illustrated in Figure 1 for $L = 3$ layers and $j^{(\ell)} = 1$ for all ℓ. With this notation, and taking into account the results in the previous problem, back-propagation can be rewritten as follows for a purely convolutional net where all the kernels in the same layer have the same size:

$$\begin{aligned} e_{\mathbf{y}^{(L,1)}} &= \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \\ \text{for } \ell = L, \ldots, 1 \quad \text{and} \quad j &= j^{(\ell)}, \ldots, 1 \ : \\ e_{\mathbf{a}^{(\ell,j)}} &= e_{\mathbf{y}^{(\ell,j)}} \odot \frac{dh}{\mathbf{a}^{(\ell,j)}} \\ e_{\mathbf{x}^{(\ell)}} &= \sum_{j'} \delta(e_{\mathbf{a}^{(\ell,j')}}, p^{(\ell)}) \ *_f \ \rho(\mathbf{k}^{(\ell,j')}) \\ e_{\mathbf{w}^{(\ell,j)}} &= \begin{bmatrix} e_{\mathbf{k}^{(\ell,j)}}(:) \\ e_{b^{(\ell,j)}} \end{bmatrix} \quad \text{where} \quad e_{\mathbf{k}^{(\ell,j)}} = \mu(\delta(e_{\mathbf{a}^{(\ell,j)}}, p^{(\ell)}) \ *_f \ \rho(\mathbf{x}^{(\ell)}), n) \quad \text{and} \quad e_{b^{(\ell,j)}} = \sum_r e_{\mathbf{a}^{(\ell,j)}}(r) \end{aligned}$$

The symbol '$\odot$' ('`\odot`' in LaTeX) denotes entry-by-entry multiplication ('`.*`' in MATLAB), and the notation

$$\frac{dh}{\mathbf{a}^{(\ell,j)}} = \begin{bmatrix} \frac{dh}{da}\big|_{a=a_1^{(\ell,j)}} \\ \vdots \\ \frac{dh}{da}\big|_{a=a_{D^{(\ell)}}^{(\ell,j)}} \end{bmatrix}$$

denotes a column vector with the values of the derivative of h with respect to its (scalar) argument computed at the activation j in layer ℓ. For instance, if h is the ReLU, this vector has a 1 where the activation is positive and a zero where the activation is zero or negative. The term $e_{b^{(\ell,j)}}$ is the derivative of the error e with respect to the bias $b^{(\ell)}$ in layer ℓ, obtained as follows: Dropping the superscripts ℓ and j for simplicity, we have

$$e_b = \frac{\partial e}{\partial b} = \sum_r \frac{\partial e}{\partial a_r} \frac{\partial a_r}{\partial b} = \sum_r \frac{\partial e}{\partial a_r} = \sum_r e_{\mathbf{a}^{(\ell)}}(r)$$

where r is a row index. A similar computation yields the expression for $e_{\mathbf{x}^{(\ell)}}$. The integers $p^{(\ell)}$ in the expressions above are the lengths of the convolution outputs, as you computed them in problem 2.

While you verified these expressions for one-dimensional kernels, they also hold for d-dimensional kernels. When $d > 1$, the 'ρ' operator is to be applied to each dimension, while the 'δ' and 'μ' functions operate on the first dimension of their first arguments only. When there are $j^{(\ell)} > 1$ kernels—and therefore $j^{(\ell)}$ activation maps—in a layer, the relationships above are to be applied to each of the kernels (and activation maps) separately.

You will now write code for the back-propagation algorithm for convolutional neural nets in a simple case in which no pooling occurs and the input, kernel, and output sizes are constrained as follows:[2]

- The input $\mathbf{x}$ is a column vector. Let $j^{(0)} = 1$, the number of columns in the input.
- Layer ℓ has $j^{(\ell)}$ 2-dimensional kernels of size $n^{(\ell)} \times j^{(\ell-1)}$ for $\ell = 1, \ldots L$ and stride $s^{(\ell)}$. There is one scalar bias per kernel.

[2]This is a simplification relative to real networks, which include pooling, and have images rather than vectors as inputs. However, this simplification makes the code manageable for this exercise.

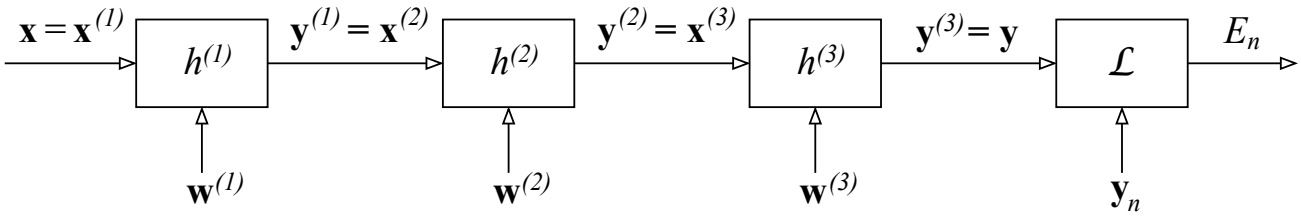

Figure 1: Notation used in this assignment for the computation of the error term E_n for a neural net with $L = 3$ layers and a single activation map per layer.

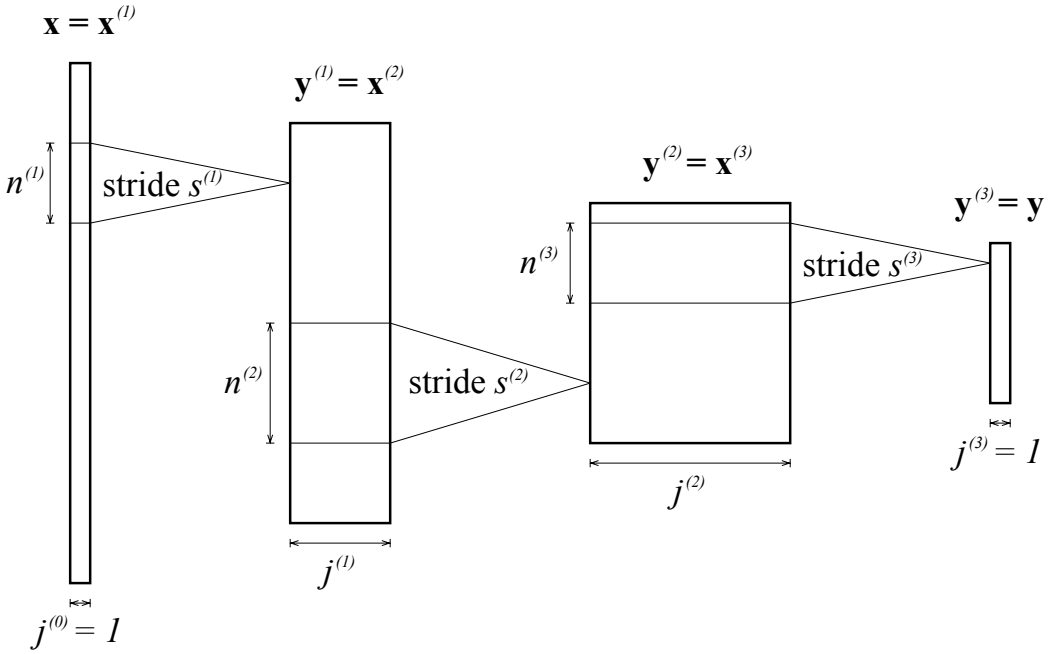

Figure 2: Kernel and feature map block sizes for a neural net with $L = 3$ layers. The second dimension of each kernel "fills" the second dimension of the feature map block it is applied to, so that convolution with the `'valid'` option results in a one-dimensional feature map for each kernel.

- For the last layer, $j^{(L)} = 1$ (a single, 2-dimensional kernel of size $n^{(L)} \times j^{(L-1)}$).
- As a consequence of the last constraint, the output $\mathbf{y}$ from the network is a vector.

All the forward convolutions in the net are computed with the `'valid'` option. Since the size $j^{(\ell-1)}$ of the second dimension of a kernel in layer ℓ is equal to the number of feature maps in the previous layer, each kernel computes a one-dimensional (column) feature map. Figure 2 illustrates for $L = 3$, using notation similar to that used in A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," *NIPS*, 25:1106–1114, 2012, but with a one-dimensional input.

(a) Write a MATLAB function

```
function [y, x, a] = cnn(x1, net)
```

that checks if the `net` satisfies the format requirements below and runs a convolutional neural net whose layers are as described in the previous problem. The argument `net` is a net whose format is similar to that in problem 1, but with the following fields for layer ℓ:

- `kernel` is an $n^{(\ell)} \times j^{(\ell-1)} \times j^{(\ell)}$ array that represents $j^{(\ell)}$ kernels of size $n^{(\ell)} \times j^{(\ell-1)}$.
- `bias` is a row vector with $j^{(\ell)}$ scalar entries, one per kernel.
- `stride` is a positive integer scalar.
- `h` is a handle to a function that computes the nonlinearity for that layer, and its derivative.

More specifically, the function with handle `h` is expected to have a header with the following format:

```
function [y, dhda] = h(a)
```

where `y` is the output corresponding to input `a` and `dhda` is the derivative of `h` at `a`. This function should behave like in problem 1 (and for both output arguments) as far as multiple inputs are concerned. The function `ReLU` provided with this assignment is an example of such a function.

The output from each layer is a matrix each of whose columns is a feature map. The argument `x1` is the input to the net—a single column vector—so the output from the first layer is a $p^{(1)} \times j^{(1)}$ array (see Figure 2) for a suitable value of $p^{(1)}$. The input to layers after the first is in general a matrix, and the output from that layer is the result of applying every kernel in the layer to the input array. Each application returns a single column of the output feature map. It is OK to loop explicitly over kernels (and therefore output feature-map columns) in your code.

The output argument `y` from `cnn` is the output from the last layer of the net, and is a vector because of the constraints imposed on the kernel sizes for this problem. The output arguments `x` and `a` from `cnn` are $L \times 1$ cell arrays for a net with L layers, and store intermediate quantities that will be needed for back-propagation. Specifically, cell `x{`ℓ`}` is the *input* matrix to layer ℓ, and cell `a{`ℓ`}` is the activation matrix in layer ℓ.

Hand in your code for the function `cnn`, as well as the value of the error `e` and a printout of the entries of the vector `y` that result from running the following code:

```
load test
y = cnn(xn, convnet);
e = norm(yn - y)^2;
```

The net `convnet` stored in the file `test.mat` provided with this assignment does not output the desired output `yn`, so you should not expect a zero (or even a small) error. *We* know what results to expect, so this test is a simple sanity check we will use for your code.

PROGRAMMING NOTES.

- If you say `load test` once you downloaded the assignment and placed the code in your working MATLAB directory, a sample net shows up in a variable called `convnet`, a sample input shows up in the variable `xn`, and a sample desired output shows up as `yn` (so `(xn, yn)` can be viewed as a training sample). The sample net satisfies all the size constraints.
- The function `ok` provided with this assignment takes a net and returns `true` if the net satisfies the format requirements listed above. Otherwise, it issues an error with an explanation.
- It is of course fine to use the function `ok` in your code.
- The function `square` is provided with this assignment.
- Pay close attention to the sizes of the matrix `x{`ℓ`}` and of the three-dimensional array `kernel` in each layer: The second dimension of `kernel` is equal to the number of feature maps in the *previous* layer, and therefore to the second dimension of the input `x{`ℓ`}` to the current layer. The third dimension of `kernel` is equal to the number of feature maps computed by the *current* layer and therefore to the second dimension of the output `x{`$\ell + 1$`}` from the current layer (for $\ell = L$, the output from the layer is `y`). Figure 2 may help clarify these relationships.
- It is fine to implement stride by first computing the convolution with stride 1, and then sampling the result.
- Use `convn` for multidimensional convolutions.

(b) Write a MATLAB function

```
function [g, e] = backprop(net, loss, xn, yn)
```

that uses back-propagation and the convolution results stated earlier to compute the gradient `g` and the value `e` of the error of a convolutional net with respect to all the parameters of the net. The argument `yn` is the desired output, taken from the training set, and `loss` is a handle to a function with header

```
function [e, ey] = loss(yn, y)
```

that returns the error `e` and the derivative of the loss function with respect to `y` for the training sample `yn`. The other input arguments to `backprop` are as in part (a). The network `net` may have any number of layers, but all the layers are of the format described in part (a).

Your output `g` is a column vector. For each layer ℓ, starting with $\ell = 1$, the vector `g` lists the gradient with respect to kernel 1, then to bias 1, then to kernel 2, then to bias 2, ..., then to kernel $j^{(\ell)}$, then to bias $j^{(\ell)}$. To clarify, you may want to look at the function `weights` provided with this assignment, which returns a column vector of all the weights in a convolutional net. **Hand in your code.**

PROGRAMMING NOTES.

- The goal of this exercise is to help you work through the details of writing a back-propagation routine. You will succeed if you patiently replicate the expressions given in the preamble to this problem.
- As a useful sanity check for your code, note that $e_{\mathbf{blah}}$ for any value of $\mathbf{blah}$ is the derivative of a scalar (the error $e = E_n$ for a single training sample) with respect to $\mathbf{blah}$, and therefore has the same shape and size as $\mathbf{blah}$. For instance, $e_{\mathbf{k}^{(\ell)}}(:,:,j)$ is an array of size $n^{(\ell)} \times j^{(\ell-1)}$, and $e_{\mathbf{w}^{(\ell)}}$ is a column vector.
- MATLAB functions `reverse`, `dilute`, and `middle`, which implement operators ρ, δ, and μ respectively, are provided with this assignment.

(c) Writing differentiation code is error-prone, so it is good to check your results against those obtained with a slower and less accurate but simpler method for computing derivatives, namely, numeric differentiation. Given a function $e(\mathbf{w})$, numeric differentiation is based on an approximation of the definition of derivative:

$$e_{\mathbf{w}}(i) = \frac{\partial e}{\partial w_i} \approx \frac{e(\mathbf{w} + \mathbf{d}_i) - e(\mathbf{w})}{d}$$

where d is a small positive number and $\mathbf{d}_i$ is a vector as long as $\mathbf{w}$ that has d in its i-th position and zero everywhere else. To compute the gradient, loop over all entries of $\mathbf{w}$ and compute the value in the right-hand side of the expression above.

Write a MATLAB function with header

```
function [g, e] = numeric(net, loss, xn, yn)
```

that uses numeric differentiation with $d = 10^{-6}$ to compute the gradient `g` and value `e` of the error E_n with the given convolutional `net` and `loss` function for the given training sample (`xn`, `yn`). The argument `loss` is the same as in part (b).
Hand in your code.

PROGRAMMING NOTE. You can change the weights in network `net` by saying

```
w = weights(net);
w = ...;
net = setWeights(w, net);
```

where `weights` and `setWeights` are functions provided with this assignment and the dots represent code you write.

(d) Evaluate the norm of the difference between the analytically and numerically computed gradients for our running example as follows:

```
load test
gb = backprop(convnet, @ee2, xn, yn);
gn = numeric(convnet, @ee2, xn, yn);
dg = gb - gn;
ndg = norm(dg) / mean([norm(gn), norm(gb)]);
```

where `ee2` is a function that computes value and gradient (relative to $\mathbf{y}$) of the loss function

$$\mathcal{L}(\mathbf{y}_n, \mathbf{y}) = \|\mathbf{y}_n - \mathbf{y}\|^2 .$$

This is the squared Euclidean norm of the vector difference between the output $\mathbf{y}$ from the whole net and the training sample output $\mathbf{y}_n$.

The network `convnet` comes initialized with parameter values, so your derivatives are computed at those values. The resulting discrepancy `ndg` will be greater than zero because of the numeric approximations. **Hand in the code for your `ee2` function, state the value of `ndg` you obtain and provide two separate diagrams.** The first diagram should show plots of `gb` and `gn` superimposed on each other, in different colors and with appropriate legends. The second should plot the difference vector `dg`.

PROGRAMMING NOTE. If `ndg` is large, there is a bug somewhere in your code. One way to debug it is to make a tiny net with perhaps two layers and very small kernels, so you can compute derivatives by hand and compare with what your code does.