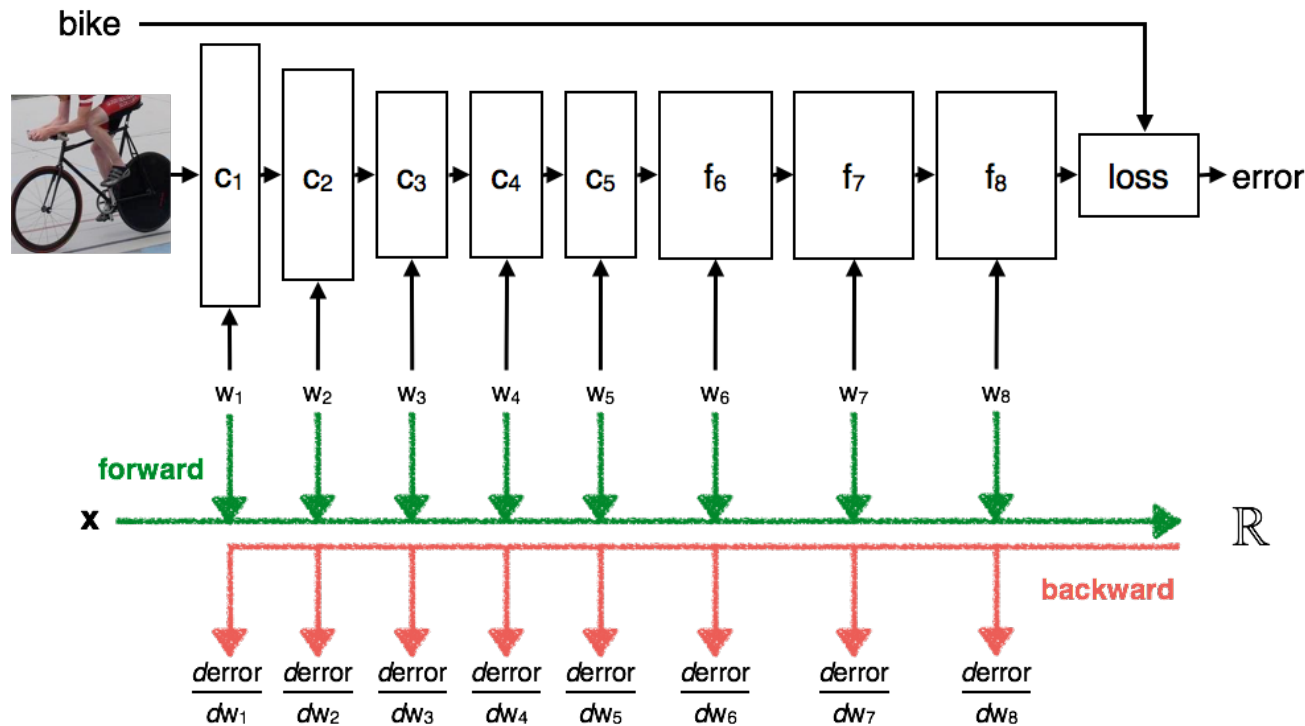


VGG Convolutional Neural Networks Practical

By Andrea Vedaldi and Andrew Zisserman

This is an [Oxford Visual Geometry Group](#) computer vision practical, authored by [Andrea Vedaldi](#) and Andrew Zisserman (Release 2015a).



Convolutional neural networks are an important class of learnable representations applicable, among others, to numerous computer vision problems. Deep CNNs, in particular, are composed of several layers of processing, each involving linear as well as non-linear operators, that are learned jointly, in an end-to-end manner, to solve a particular task. These methods are now the dominant approach for feature extraction from audiovisual and textual data.

Installation

Note that this practical requires compiling the (included) MatConvNet library (MatConvNet/matlab/vl_compilenn.m). This should happen automatically (see the setup.m script), but make sure that the compilation succeeds on the laboratory computers. The practical runs in MATLAB and uses MatConvNet and VLFeat. VLFeat comes with pre-built binaries, but MatConvNet does not.

After the installation is complete, open and edit the script `exercise1.m` in the MATLAB editor. The script contains commented code and a description for all steps of this exercise, for [Part I](#) of this document. You can cut and paste this code into the MATLAB window to run it, and will need to modify it as you go through the session. Other files `exercise2.m`, `exercise3.m`, and `exercise4.m` are given for [Part II](#), [III](#), and [IV](#).

This practical explores the basics of learning (deep) CNNs. The first part introduces typical CNN building blocks, such as ReLU units and linear filters, with a particular emphasis on understanding back-propagation. The second part looks at learning two basic CNNs. The first one is a simple non-linear filter capturing particular image structures, while the second one is a network that recognises typewritten characters (using a variety of different fonts). These examples illustrate the use of stochastic gradient descent with momentum, the definition of an objective function, the construction of mini-batches of data, and data jittering. The last part shows how powerful CNN models can be downloaded off-the-shelf and used directly in applications, bypassing the expensive training process.

Each part contains several **Questions** (that require pen and paper) and **Tasks** (that require experimentation or coding) to be answered/completed before proceeding further in the practical.

Part 1: CNN building blocks

Part 1.1: convolution

A feed-forward neural network can be thought of as the composition of number of functions

$$f(\mathbf{x}) = f_L(\dots f_2(f_1(\mathbf{x}; \mathbf{w}_1); \mathbf{w}_2) \dots), \mathbf{w}_L).$$

Each function f_l takes as input a datum \mathbf{x}_l and a parameter vector \mathbf{w}_l and produces as output a datum \mathbf{x}_{l+1} . While the type and sequence of functions is usually handcrafted, the parameters $\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_L)$ are *learned from data* in order to solve a target problem, for example classifying images or sounds.

In a *convolutional neural network* data and functions have additional structure. The data $\mathbf{x}_1, \dots, \mathbf{x}_n$ are images, sounds, or more in general maps from a lattice¹ to one or more real numbers. In particular, since the rest of the practical will focus on computer vision applications, data will be 2D arrays of pixels. Formally, each \mathbf{x}_i will be a $M \times N \times K$ real array of $M \times N$ pixels and K channels per pixel. Hence the first two dimensions of the array span space, while the last one spans channels. Note that only the input $\mathbf{x} = \mathbf{x}_1$ of the network is an actual image, while the remaining data are intermediate *feature maps*.

The second property of a CNN is that the functions f_l have a *convolutional structure*. This means that f_l applies to the input map \mathbf{x}_l an operator that is *local and translation invariant*. Examples of convolutional operators are applying a bank of linear filters to \mathbf{x}_l .

In this part we will familiarise ourselves with a number of such convolutional and non-linear operators. The first one is the regular *linear convolution* by a filter bank. We will start by focusing our attention on a single function relation as follows:

$$f : \mathbb{R}^{M \times N \times K} \rightarrow \mathbb{R}^{M' \times N' \times K'}, \quad \mathbf{x} \mapsto \mathbf{y}.$$

Open the `example1.m` file, select the following part of the code, and execute it in MATLAB (right button > Evaluate selection or Shift+F7).

```
% Read an example image
x = imread('peppers.png') ;

% Convert to single format
x = im2single(x) ;

% Visualize the input x
```

```
figure(1) ; clf ; imagesc(x)
```

This should display an image of bell peppers in Figure 1:



Use MATLAB `size` command to obtain the size of the array `x`. Note that the array `x` is converted to *single precision* format. This is because the underlying MatConvNet assumes that data is in single precision.

Question. The third dimension of `x` is 3. Why?

Now we will create a bank 10 of $5 \times 5 \times 3$ filters.

```
% Create a bank of linear filters
```

```
w = randn(5,5,3,10,'single') ;
```

The filters are in single precision as well. Note that w has four dimensions, packing 10 filters. Note also that each filter is not flat, but rather a volume with three layers. The next step is applying the filter to the image. This uses the `vl_nnconv` function from MatConvNet:

```
% Apply the convolution operator  
y = vl_nnconv(x, w, []) ;
```

Remark: You might have noticed that the third argument to the `vl_nnconv` function is the empty matrix `[]`. It can be otherwise used to pass a vector of bias terms to add to the output of each filter.

The variable y contains the output of the convolution. Note that the filters are three-dimensional, in the sense that it operates on a map \mathbf{x} with K channels. Furthermore, there are K' such filters, generating a K' dimensional map \mathbf{y} as follows

$$y_{i'j'k'} = \sum_{ijk} w_{ijkk'} x_{i+i', j+j', k}$$

Questions: Study carefully this expression and answer the following:

- Given that the input map \mathbf{x} has $M \times N \times K$ dimensions and that each of the K' filters has dimension $M_f \times N_f \times K$, what is the dimension of \mathbf{y} ?
- Note that x is indexed by $i + i'$ and $j + j'$, but that there is no plus sign between k and k' . Why?

Task: check that the size of the variable y matches your calculations.

We can now visualise the output y of the convolution. In order to do this, use the `vl_imarraysc` function to display an image for each feature channel in y :

```
% Visualize the output y  
figure(2) ; clf ; vl_imarraysc(y) ; colormap gray ;
```

Question: Study the feature channels obtained. Most will likely contain a strong response in correspondences of edges in the input image \mathbf{x} . Recall that w was obtained by

drawing random numbers from a Gaussian distribution. Can you explain this phenomenon?

So far filters preserve the resolution of the input feature map. However, it is often useful to *downsample the output*. This can be obtained by using the `stride` option in `vl_nnconv` :

```
% Try again, downsampling the output
y_ds = vl_nnconv(x, w, [], 'stride', 16) ;
figure(3) ; clf ; vl_imarrayshow(y_ds) ; colormap gray ;
```

As you should have noticed in a question above, applying a filter to an image or feature map interacts with the boundaries, making the output map smaller by an amount proportional to the size of the filters. If this is undesirable, then the input array can be padded with zeros by using the `pad` option:

```
% Try padding
y_pad = vl_nnconv(x, w, [], 'pad', 4) ;
figure(4) ; clf ; vl_imarrayshow(y_pad) ; colormap gray ;
```

Task: Convince yourself that the previous code's output has different boundaries compared to the code that does not use padding. Can you explain the result?

In order to consolidate what has been learned so far, we will now design a filter by hand:

```
w = [0  1  0 ;
      1 -4  1 ;
      0  1  0] ;
w = single repmat(w, [1, 1, 3]) ;
y_lap = vl_nnconv(x, w, []) ;

figure(5) ; clf ; colormap gray ;
subplot(1,2,1) ;
imagesc(y_lap) ; title('filter output') ;
subplot(1,2,2) ;
imagesc(-abs(y_lap)) ; title('- abs(filter output)') ;
```

Questions:

- What filter have we implemented?
- How are the RGB colour channels processed by this filter?
- What image structure are detected?

Part 1.2: non-linear gating

As we stated in the introduction, CNNs are obtained by composing several different functions. In addition to the linear filters shown in the [previous part](#), there are several non-linear operators as well.

Question: Some of the functions in a CNN *must* be non-linear. Why?

The simplest non-linearity is obtained by following a linear filter by a *non-linear gating function*, applied identically to each component (i.e. point-wise) of a feature map. The simplest such function is the *Rectified Linear Unit (ReLU)*

$$y_{ijk} = \max\{0, x_{ijk}\}.$$

This function is implemented by `vl_relu`; let's try this out:

```
w = single( repmat([1 0 -1], [1, 1, 3])) ;
w = cat(4, w, -w) ;
y = vl_nnconv(x, w, []) ;
z = vl_nnrelu(y) ;

figure(6) ; clf ; colormap gray ;
subplot(1,2,1) ; vl_imarrayesc(y) ;
```

```
subplot(1,2,2) ; vl_imarraysc(z) ;
```

Tasks:

- Run the code above and understand what the filter \mathbf{w} is doing.
- Explain the final result \mathbf{z} .

Part 1.3: pooling

There are several other important operators in a CNN. One of them is *pooling*. A pooling operator operates on individual feature channels, coalescing nearby feature values into one by the application of a suitable operator. Common choices include max-pooling (using the max operator) or sum-pooling (using summation). For example, *max-pooling* is defined as:

$$y_{ijk} = \max\{y_{i'j'k} : i \leq i' < i + p, j \leq j' < j + p\}$$

Max pooling is implemented by the `vl_nnpool` function. Try this now:

```
y = vl_nnpool(x, 15) ;  
figure(6) ; clf ; imagesc(y) ;
```

Question: look at the resulting image. Can you interpret the result?

The function `vl_nnpool` supports subsampling and padding just like `vl_nnconv`. However, for max-pooling feature maps are padded with the value $-\infty$ instead of 0. Why?

Part 1.4: normalisation

Another important CNN building block is channel-wise normalisation. This operator normalises the vector of feature channels at each spatial location in the input map \mathbf{x} . The form of the normalisation operator is actually rather curious:

$$y_{ijk'} = \frac{x_{ijk}}{\left(\kappa + \alpha \sum_{k \in G(k')} x_{ijk}^2\right)^\beta}$$

where $G(k) = \left[k - \lfloor \frac{\rho}{2} \rfloor, k + \lceil \frac{\rho}{2} \rceil\right] \cap \{1, 2, \dots, K\}$ is a group of ρ consecutive feature channels in the input map.

Task: Understand what this operator is doing. How would you set κ , α and β to achieve simple L^2 normalisation?

Now let's try this out:

```
rho = 5 ;  
kappa = 0 ;  
alpha = 1 ;  
beta = 0.5 ;  
y_nrm = vl_nnnormalize(x, [rho kappa alpha beta]) ;  
figure(6) ; clf ; imagesc(y_nrm) ;
```

Tasks:

- Inspect the figure just obtained. Can you interpret it?
- Compute the L^2 norm of the feature channels in the output map y_nrm . What do you notice?
- Explain this result in relation to the particular choice of the parameters ρ , κ , α and β .

Part 2: back-propagation and derivatives

The parameters of a CNN $\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_L)$ should be learned in such a manner that the overall CNN function $\mathbf{z} = f(\mathbf{x}; \mathbf{w})$ achieves a desired goal. In some cases, the goal is to model the distribution of the data, which leads to a *generative objective*. Here, however, we will use f as a *regressor* and obtain it by minimising a *discriminative objective*. In simple terms, we are given:

- examples of the desired input-output relations $(\mathbf{x}_1, \mathbf{z}_1), \dots, (\mathbf{x}_n, \mathbf{z}_n)$ where \mathbf{x}_i are input data and \mathbf{z}_i corresponding output values;
- and a loss $\ell(\mathbf{z}, \hat{\mathbf{z}})$ that expresses the penalty for predicting $\hat{\mathbf{z}}$ instead of \mathbf{z} .

We use those to write the empirical loss of the CNN f by averaging over the examples:

$$L(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \ell(\mathbf{z}_i, f(\mathbf{x}_i; \mathbf{w}))$$

Note that the composition of the function f with the loss ℓ can be thought of as a CNN with one more layer (called a *loss layer*). Hence, with a slight abuse of notation, in the rest of this part we incorporate the loss in the function f (which therefore is a map $\mathcal{X} \rightarrow \mathbb{R}$) and do not talk about it explicitly anymore.

The simplest algorithm to minimise L , and in fact one that is used in practice, is *gradient descent*. The idea is simple: compute the gradient of the objective L at a current solution \mathbf{w}^t and then update the latter along the direction of fastest descent of L :

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \frac{\partial f}{\partial \mathbf{w}}(\mathbf{w}^t)$$

where $\eta_t \in \mathbb{R}_+$ is the *learning rate*.

Part 2.1: the theory of back-propagation

The basic computational problem to solve is the calculation of the gradient of the function with respect to the parameter \mathbf{w} . Since f is the composition of several functions, the key ingredient is the *chain rule*:

$$\begin{aligned}\frac{\partial f}{\partial \mathbf{w}_l} &= \frac{\partial}{\partial \mathbf{w}_l} f_L(\dots f_2(f_1(\mathbf{x}; \mathbf{w}_1); \mathbf{w}_2) \dots), \mathbf{w}_L) \\ &= \frac{\partial \text{vec } f_L}{\partial \text{vec } \mathbf{x}_L^\top} \frac{\partial \text{vec } f_{L-1}}{\partial \text{vec } \mathbf{x}_{L-1}^\top} \dots \frac{\partial \text{vec } f_{l+1}}{\partial \text{vec } \mathbf{x}_{l+1}^\top} \frac{\partial \text{vec } f_l}{\partial \mathbf{w}_l^\top}\end{aligned}$$

The notation requires some explanation. Recall that each function f_l is a map from a $M \times N \times K$ array to a $M' \times N' \times K'$ array. The operator vec *vectorises* such arrays by stacking their elements in a column vector (the stacking order is arbitrary but conventionally column-major). The symbol $\partial \text{vec } f_l / \partial \text{vec } \mathbf{x}_l^\top$ then denotes the derivative of a column vector of output variables by a row vector of input variables. Note that \mathbf{w}_l is already assumed to be a column vector so it does not require explicit vectorisation.

Questions: Make sure you understand the structure of this formula and answer the following:

- $\partial \text{vec } f_l / \partial \text{vec } \mathbf{x}_l^\top$ is a matrix. What are its dimensions?
- The formula can be rewritten with a slightly different notation by replacing the symbols f_l with the symbols \mathbf{x}_{l+1} . If you do so, do you notice any formal cancellation?
- The formula only includes the derivative symbols. However, these derivatives must be computed at a well defined point. What is this point?

To apply the chain rule we must be able to compute, for each function f_l , its derivative with respect to the parameters \mathbf{w}_l as well as its input \mathbf{x}_l . While this could be done naively, a problem is the very high dimensionality of the matrices involved in this calculation as these are $M'N'K' \times MNK$ arrays. We will now introduce a “trick” that allows this to be reduced to working with MNK numbers only and which will yield the *back-propagation algorithm*.

The key observation is that we are not after $\partial \text{vec } f_l / \partial \mathbf{w}_l^\top$ but after $\partial f / \partial \mathbf{w}_l^\top$:

$$\frac{\partial f}{\partial \mathbf{w}_l^\top} = \frac{\partial g_{l+1}}{\partial \text{vec } \mathbf{x}_{l+1}^\top} \frac{\partial \text{vec } f_l}{\partial \mathbf{w}_l^\top}$$

where $g_{l+1} = f_L \circ \dots \circ f_{l+1}$ is the “tail” of the CNN.

Question: Explain why the dimensions of the vectors $\partial g_{l+1} / \partial \text{vec } \mathbf{x}_{l+1}$ and $\partial f / \partial \mathbf{w}_l$ equals the number of elements in \mathbf{x}_{l+1} and \mathbf{w}_l respectively. Hence, in particular, the symbol $\partial g_{l+1} / \partial \mathbf{x}_{l+1}$ (without vectorisation) denotes an array with the same size of \mathbf{x}_{l+1} .

Hint: recall that the last layer is the loss.

Hence the algorithm can focus on computing the derivatives of g_l instead of f_l which are far lower-dimensional. To see how this can be done iteratively, decompose g_l as:

$$\mathbf{x}_l \longrightarrow \boxed{f_l} \longrightarrow \mathbf{x}_{l+1} \longrightarrow \boxed{g_{l+1}} \longrightarrow \mathbf{x}_L$$

$$\quad \quad \quad \uparrow$$

$$\quad \quad \quad \mathbf{w}_l$$

Then the key of the iteration is obtaining the derivatives for layer l given the ones for layer $l + 1$:

- Input:
 - the derivative $\partial g_{l+1} / \partial \mathbf{x}_{l+1}$.
- Output:
 - the derivative $\partial g_l / \partial \mathbf{x}_l$
 - the derivative $\partial g_l / \partial \mathbf{w}_l$

Question: Suppose that f_l is the function $\mathbf{x}_{l+1} = A\mathbf{x}_l$ where \mathbf{x}_l and \mathbf{x}_{l+1} are column vectors. Suppose that $B = \partial g_{l+1} / \partial \mathbf{x}_{l+1}$ is given. Derive an expression for $C = \partial g_l / \partial \mathbf{x}_l$ and an expression for $D = \partial g_l / \partial \mathbf{w}_l$.

Part 2.1: using back-propagation in practice

A key feature of MatConvNet and similar neural network packages is the ability to support back-propagation. In order to do so, let's focus on a single computational block f , followed by a function g :

$$\mathbf{x} \longrightarrow \boxed{f} \longrightarrow \mathbf{y} \longrightarrow \boxed{g} \longrightarrow z$$

$$\quad \quad \quad \uparrow$$

$$\quad \quad \quad \mathbf{w}$$

where z is assumed to be a scalar. Then each computation block (for example `vl_nnconv` or `vl_nnpool`) can compute $\partial z / \partial \mathbf{x}$ and $\partial z / \partial \mathbf{w}$ given as input \mathbf{x} and $\partial z / \partial \mathbf{y}$. Let's put this into practice:

```

% Read an example image
x = im2single(imread('peppers.png')) ;

% Create a bank of linear filters and apply them to the image
w = randn(5,5,3,10,'single') ;
y = vl_nnconv(x, w, []) ;

% Create the derivative dz/dy
dzdy = randn(size(y), 'single') ;

% Back-propagation
[dzdx, dzdw] = vl_nnconv(x, w, [], dzdy) ;

```

Task: Run the code above and check the dimensions of `dzdx` and `dzdy`. Does this matches your expectations?

An advantage of this modular view is that new building blocks can be coded and added to the architecture in a simple manner. However, it is easy to make mistakes in the calculation of complex derivatives. Hence, it is a good idea to verify results numerically. Consider the following piece of code:

```

% Check the derivative numerically
ex = randn(size(x), 'single') ;
eta = 0.0001 ;
xp = x + eta * ex ;
yp = vl_nnconv(xp, w, []) ;

dzdx_empirical = sum(dzdy(:) .* (yp(:) - y(:)) / eta) ;
dzdx_computed = sum(dzdx(:) .* ex(:)) ;

fprintf(...
    'der: empirical: %f, computed: %f, error: %.2f %%\n', ...
    dzdx_empirical, dzdx_computed, ...
    abs(1 - dzdx_empirical/dzdx_computed)*100) ;

```

Questions:

- What is the meaning of `ex` in the code above?
- What are the derivatives `dzdx_empirical` and `dzdx_computed`?

Tasks:

- Run the code and convince yourself that `vl_nnconv` derivatives is (probably) correct.
- Create a new version of this code to test the derivative calculation with respect to `w`.

We are now ready to build our first elementary CNN, composed of just two layers, and to compute its derivatives:

```
% Parameters of the CNN
w1 = randn(5,5,3,10,'single') ;
rho2 = 10 ;

% Run the CNN forward
x1 = im2single(imread('peppers.png')) ;
x2 = vl_nnconv(x1, w1, []) ;
x3 = vl_nnpool(x2, rho2) ;

% Create the derivative dz/dx3
dzdx3 = randn(size(x3), 'single') ;

% Run the CNN backward
dzdx2 = vl_nnpool(x2, rho2, dzdx3) ;
[dzdx1, dzdw1] = vl_nnconv(x1, w1, [], dzdx2) ;
```

Question: Note that the last derivative in the CNN is `dzdx3`. Here, for the sake of the example, this derivative is initialised randomly. In a practical application, what would this derivative represent?

We can now use the same technique as before to check that the derivative computed through back-propagation are correct.

```
% Check the derivative numerically
ew1 = randn(size(w1), 'single') ;
eta = 0.0001 ;
w1p = w1 + eta * ew1 ;

x1p = x1 ;
x2p = vl_nnconv(x1p, w1p, []) ;
x3p = vl_nnpool(x2p, rho2) ;

dzdw1_empirical = sum(dzdx3(:) .* (x3p(:) - x3(:)) / eta) ;
dzdw1_computed = sum(dzdw1(:) .* ew1(:)) ;

fprintf(...
    'der: empirical: %f, computed: %f, error: %.2f %%\n', ...
    dzdw1_empirical, dzdw1_computed, ...
    abs(1 - dzdw1_empirical/dzdw1_computed)*100) ;
```

Part 3: learning a tiny CNN

In this part we will learn a very simple CNN. The CNN is composed of exactly two layers: a convolutional layer and a max-pooling layer:

$$\mathbf{x}_2 = W * \mathbf{x}_1 + b, \quad \mathbf{x}_3 = \text{maxpool}_\rho \mathbf{x}_2.$$

W contains a single 3×3 square filter, so that b is a scalar. and the input image $\mathbf{x} = \mathbf{x}_1$ has a single channel.

Task

- Open the file `tinycnn.m` and inspect the code. Convince yourself that the code computes the CNN just described.
- Look at the paddings used in the code. If the input image \mathbf{x}_1 has dimensions $M \times N$, what is the dimension of the output feature map \mathbf{x}_3 ?

In the rest of the section we will learn the CNN parameters in order to extract blob-like structures from images, such as the ones in the following image:



Part 3.1: training data and labels

The first step is to load the image `data/dots.jpg` and to use the supplied `extractBlackBlobs` function to extract all the black dots in the image.

```
% Load an image
im = rgb2gray(im2single(imread('data/dots.jpg')));

% Compute the location of black blobs in the image
```



```
[pos,neg] = extractBlackBlobs(im) ;
```

The arrays `pos` and `neg` contain now pixel labels and will be used as *annotations* for the supervised training of the CNN. These annotations can be visualised as follows:

```
figure(1) ; clf ;  
subplot(1,3,1) ; imagesc(im) ; axis equal ; title('image') ;  
subplot(1,3,2) ; imagesc(pos) ; axis equal ; title('positive points (blob centres)') ;  
subplot(1,3,3) ; imagesc(neg) ; axis equal ; title('negative points (not a blob)') ;  
colormap gray ;
```

Task: Inspect `pos` and `neg` and convince yourself that:

- `pos` contains a single `true` value in correspondence of each blob centre;
- `neg` contains a `true` value for each pixel sufficiently far away from a blob.

Are there pixels for which both `pos` and `neg` evaluate to false?

Part 3.2: image preprocessing

Before we attempt to train the CNN, the image is pre-processed to remove its mean value. It is also smoothed by applying a Gaussian kernel of standard deviation 3 pixels:

```
% Pre-smooth the image  
im = vl_imsmooth(im,3) ;  
  
% Subtract median value  
im = im - median(im(:)) ;
```

We will come back to this preprocessing steps later.

Part 3.3: learning with gradient descent

We will now setup a learning problem to learn W and b to detect black blobs in images. Recall that the CNN computes for each image pixel (u, v) a score $f(\mathbf{x}; \mathbf{w}, b)_{(u,v)}$. We would like this score to be:

- at least as large as 1 for any pixel that is marked as a blob centre (pos or $(u, v) \in \mathcal{P}$) and
- at most zero for any pixel that is marked as being far away from a blob (neg or $(u, v) \in \mathcal{N}$).

We do so by defining and then optimising the following objective function:

$$E(\mathbf{w}, b) = \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{|\mathcal{P}|} \sum_{(u,v) \in \mathcal{P}} \max\{0, 1 - f(\mathbf{x}; \mathbf{w}, b)_{(u,v)}\} + \frac{1}{|\mathcal{N}|} \sum_{(u,v) \in \mathcal{N}} \max\{0, f(\mathbf{x}; \mathbf{w}, b)_{(u,v)}\}.$$

Questions:

- What can you say about the score of each pixel if $\lambda = 0$ and $E(\mathbf{w}, b) = 0$?
- Note that the objective enforces a *margin* between the scores of the positive and negative pixels. How much is this margin?

We can now train the CNN by minimising the objective function with respect to \mathbf{w} and b . We do so by using an algorithm called *gradient descent with momentum*. Given the current solution (\mathbf{w}_t, b_t) and update it, this is updated to $(\mathbf{w}_{t+1}, b_{t+1})$ by following the direction of fastest descent as given by the negative gradient $-\nabla E(\mathbf{w}_t, b_t)$ of the objective. However, gradient updates are smoothed by considering a *momentum* term $(\bar{\mathbf{w}}_t, \bar{b}_t)$, yielding the update equations

$$\bar{\mathbf{w}}_{t+1} \leftarrow \mu \bar{\mathbf{w}}_t + \eta \frac{\partial E}{\partial \mathbf{w}_t}, \quad \mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \bar{\mathbf{w}}_t.$$

and similarly for the bias term. Here μ is the *momentum rate* and η the *learning rate*.

Questions:

- Explain why the momentum rate must be smaller than 1. What is the effect of having a momentum rate close to 1?
- The learning rate establishes how fast the algorithm will try to minimise the objective function. Can you see any problem with a large learning rate?

The parameters of the algorithm are set as follows:

```
numIterations = 500 ;  
rate = 5 ;  
momentum = 0.9 ;  
shrinkRate = 0.0001 ;  
plotPeriod = 10 ;
```

Tasks:

- Inspect the code in the file `exercise3.m`. Convince yourself that the code is implementing the algorithm described above. Pay particular attention at the forward and backward passes as well as at how the objective function and its derivatives are computed.
- Run the algorithm and observe the results. Then answer the following questions:
 - The learned filter should resemble the discretisation of a well-known differential operator. Which one?
 - What is the average of the filter values compared to the average of the absolute values?
- Run the algorithm again and observe the evolution of the histograms of the score of the positive and negative pixels in relation to the values 0 and 1. Answer the following:
 - Is the objective function minimised monotonically?
 - As the histograms evolve, can you identify at least two “phases” in the optimisation?
 - Once converged, do the score distribute in the manner that you would expect?

Hint: the `plotPeriod` option can be changed to plot the diagnostic figure with a higher or lower frequency; this can significantly affect the speed of the algorithm.

Part 3.4: experimenting with the tiny CNN

In this part we will experiment with several variants of the network just learned. First, we study the effect of the image smoothing:

Task: Train again the tiny CNN *without smoothing the input image in preprocessing*. Answer the following questions:

- Is the learned filter very different from the one learned before?
- If so, can you figure out what “went wrong”?
- Look carefully at the output of the first layer, magnifying with the loupe tool. Is the maximal filter response attained in the middle of each blob?

Hint: The Laplacian of Gaussian operator responds maximally at the centre of a blob only if the latter matches the blob size. Relate this fact to the combination of pre-smoothing the image and applying the learned 3×3 filter.

Now restore the smoothing but switch off subtracting the median from the input image.

Task: Train again the tiny CNN *without subtracting the median value in preprocessing*. Answer the following questions:

- Does the algorithm converge?
- Reduce a hundred-fold the learning rate and increase the maximum number of iterations by an equal amount. Does it get better?
- Explain why adding a constant to the input image can have such a dramatic effect on the performance of the optimisation.

Hint: What constraint should the filter \mathbf{w} satisfy if the filter output should be zero when (i) the input image is zero or (ii) the input image is a large constant? Do you think that it would be easy for gradient descent to enforce (ii) at all times?

What you have just witnessed is actually a fairly general principle: centring the data usually makes learning problems much better conditioned.

Now we will explore several parameters in the algorithms:

Task: Restore the preprocessing as given in `experiment4.m`. Try the following:

- Try increasing the learning rate `eta`. Can you achieve a better value of the energy in the 500 iterations?
- Disable momentum by setting `momentum = 0`. Now try to beat the result obtained above by choosing `eta`. Can you succeed?

Finally, consider the regularisation effect of shrinking:

Task: Restore the learning rate and momentum as given in `experiment4.m`. Then increase the shrinkage factor tenfold and a hundred-fold.

- What is the effect on the convergence speed?

- What is the effect on the final value of the total objective function and of the average loss part of it?

Part 4: learning a character CNN

In this part we will learn a CNN to recognise images of characters.

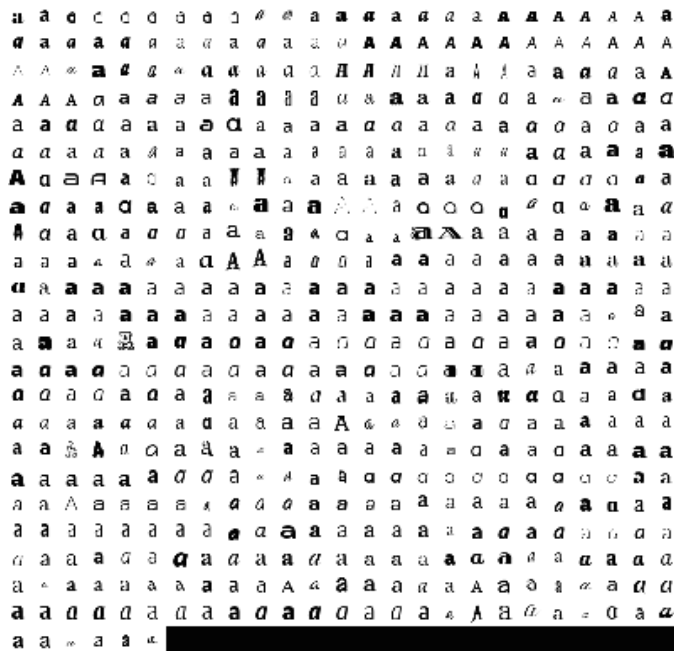
Part 4.1: prepare the data

Open up `exercise4.m` and execute Part 4.1. The code loads a structure `imdb` containing images of the characters *a*, *b*, ..., *z* rendered using approximately 931 fonts downloaded from the [Google Fonts Project](#). Look at the `imdb.images` substructure:

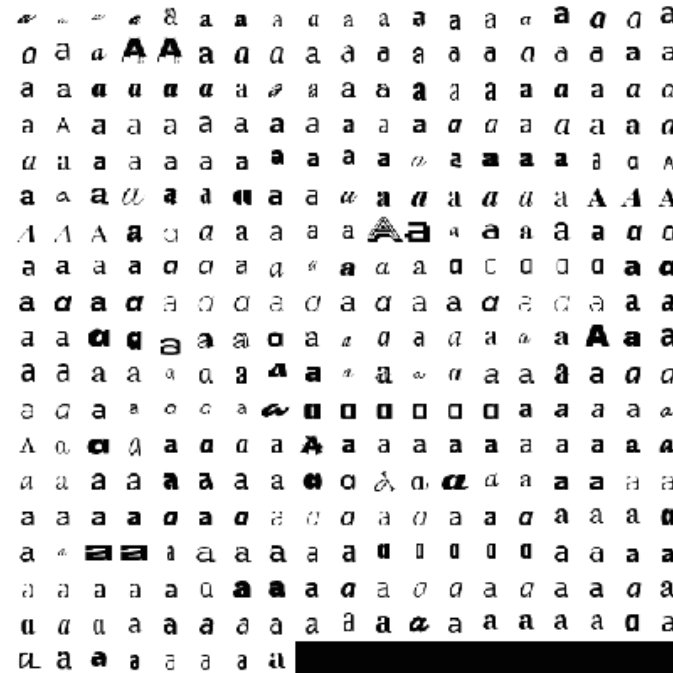
```
>> imdb.images
ans =
    id: [1x24206 double]
   data: [32x32x24206 single]
  label: [1x24206 double]
    set: [1x24206 double]
```

These are stored as the array `imdb.images.id` is a 24,206-dimensional vector of numeric IDs for each of the 24,206 character images in the dataset. `imdb.images.data` contains a 32×32 image for each character, stored as a slice of a $32 \times 32 \times 24,206$ -dimensional array. `imdb.images.label` is a vector of image labels, denoting which one of the 26 possible characters it is. `imdb.images.set` is equal to 1 for each image that should be used to train the CNN and to 2 for each image that should be used for validation.

training chars for 'a'



validation chars for 'a'



Task: look at the Figure 1 generated by the code and at the code itself and make sure that you understand what you are looking at.

Part 4.2: initialize a CNN architecture

The function `initializeCharacterCNN.m` creates a CNN initialised with random weights that will be trained to recognise character images.

Tasks:

1. By inspecting `initializeCharacterCNN.m` get a sense of the architecture that will be trained. How many layers are there? How big are the filters?
2. Use the function `vl_simplenn_display` to produce a table summarising the architecture.

Note that the *penultimate* layer has 26 output dimensions, one for each character. Character recognition looks at the maximal output to identify which character is processed by the network.

However, the last network layer is `vl_nnsoftmaxloss`, which in turn is a combination of the `vl_nnsoftmax` function and of the classification log-loss `vl_nnloss`. The *softmax* operator is given by

$$y_{ijk'} = \frac{e^{x_{ijk'}}}{\sum_k e^{x_{ijk}}}$$

whereas the *log-loss* is given by

$$y_{ij} = -\log x_{ijc_{ij}}$$

where c_{ij} is the index of the ground-truth class at spatial location (i, j) .

Remark: While in MatConvNet all operators are convolutional, in this case the network is configured such that the output of the classification layer is a $1 \times 1 \times 26$ -dimensional feature map, i.e. there remains only one spatial location.

Tasks:

1. Understand what the softmax operator does. **Hint:** to use the log-loss the data must be in the $(0, 1]$ interval.
2. Understand what is the effect of minimising the log-loss. Which neural response should become larger?
3. Why do you think MatConvNet provides a third function `vl_nnsoftmaxloss` combining both functions into a single layer?

Part 4.3: train and evaluate the CNN

We are now ready to train the CNN. To this end we use the example SGD implementation in MatConvNet (`examples/cnn_train.m`). This function requires some options:

```

trainOpts.batchSize = 100 ;
trainOpts.numEpochs = 100 ;
trainOpts.continue = true ;
trainOpts.useGpu = false ;
trainOpts.learningRate = 0.001 ;
trainOpts.numEpochs = 15 ;
trainOpts.expDir = 'data/chars-experiment' ;

```

This says that the function will operate on SGD mini-batches of 100 elements, it will run for 15 epochs (passes through the data), it will continue from the last epoch if interrupted, if will *not* use the GPU, it will use a learning rate of 0.001, and it will save any file in the `data/chars-experiment` subdirectory.

Before the training starts, the average image value is subtracted:

```

% Take the average image out
imageMean = mean(imdb.images.data(:)) ;
imdb.images.data = imdb.images.data - imageMean ;

```

This is similar to what we have done in Part 3.

The training code is called as follows:

```

% Call training function in MatConvNet
[net,info] = cnn_train(net, imdb, @getBatch, trainOpts) ;

```

Here the key, in addition to the `trainOpts` structure, is the `@getBatch` function handle. This is how `cnn_train` obtains a copy of the data to operate on. Examine this function (see the bottom of the `exercise4.m` file):

```

function [im, labels] = getBatch(imdb, batch)
im = imdb.images.data(:, :, batch) ;
im = 256 * reshape(im, 32, 32, 1, []) ;

```



```
labels = imdb.images.label(1,batch) ;
```

The function extracts the m images corresponding to the vector of indexes `batch`. It also reshapes them as a $32 \times 32 \times 1 \times m$ array (as this is the format expected by the MatConvNet functions) and multiplies the values by 256 (the resulting values match the network initialisation and learning parameters). Finally, it also returns a vector of labels, one for each image in the batch.

Task: Run the learning code and examine the plots that are produced. As training completes answer the following questions:

1. How many images per second can you process? (Look at the output in the MATLAB screen)
2. There are two sets of curves: energy and prediction error. What do you think is the difference? What is the “energy”?
3. Some curves are labelled “train” and some other “val”. Should they be equal? Which one should be lower than the other?
4. Both the top-1 and top-5 prediction errors are plotted. What do they mean? What is the difference?

Once training is finished, the model is saved back:

```
% Save the result for later use
net.layers(end) = [] ;
net.imageMean = imageMean ;
save('data/chars-experiment/charscnn.mat', '-struct', 'net') ;
```

Note that we remember the `imageMean` for later use. Note also that the softmaxloss layer is *removed* from the network before saving.

Part 4.4: visualise the learned filters

The next step is to glance at the filters that have been learned:

```
figure(2) ; clf ; colormap gray ;
vl_imarraysc(squeeze(net.layers{1}.filters), 'spacing', 2)
axis equal ;
```

```
title('filters in the first layer') ;
```

Task: what can you say about the filters?

Part 4.5: apply the model

We now apply the model to a whole sequence of characters. This is the image `data/sentence-lato.png` :

the rat the cat the dog chased killed ate the malt

```
% Load the CNN learned before
net = load('data/chars-experiment/charscnn.mat') ;

% Load the sentence
im = im2single(imread('data/sentence-lato.png')) ;
im = 256 * (im - net.imageMean) ;

% Apply the CNN to the larger image
res = vl_simplenn(net, im) ;
```

Question: The image is much wider than 32 pixels. Why can you apply to it the CNN learned before for 32×32 patches?

Task: examine the size of the CNN output using `size(res(end).x)` . Does this match your expectation?

Now use the `decodeCharacters()` function to visualise the results:

```
% Visualize the results
```

```
figure(3) ; clf ;  
decodeCharacters(net, imdb, im, res) ;
```

Tasks: inspect the output of the `decodeCharacters()` function and answer the following:

1. Is the quality of the recognition any good?
2. Does this match your expectation given the recognition rate in your validation set (as reported by `cnn_train` during training)?

Part 4.6: training with jitter

A key issue with the previous CNN is that it is not trained to recognise characters in the context of other characters. Furthermore, characters are perfectly centred in the patch. We can relax these assumptions by making the training data “more realistic”. In this part we will train a second network applying *data jittering* by:

1. Randomly adding a character to the left and to the right of the one recognised and
2. Randomly shifting the characters by up to ± 5 pixels horizontally and ± 2 pixels vertically.

This is implemented by the `getBatchWithJitter()` function (note that jittering is applied on the fly as it is so fast).

Tasks:

1. Train a second model, using the jittered data.
2. Look at the training and validation errors. Is their gap as wide as it was before?
3. Use the new model to recognise the characters in the sentence by repeating the previous part. Does it work better?
4. **Advanced.** What else can you change to make the performance even better?

Part 4.7: Training using the GPU

Skip this part if you do not wish to experiment training using GPU hardware.

A key challenge in deep learning is the sheer amount of computation required to train gigantic models from equally gigantic data collections. State-of-the-art vision models, for example, take weeks to train on specialised hardware such as GPUs, and they are essentially untrainable on CPU (unless you have access to a very large cluster). Thus it is practically important to learn how to use this hardware.

In MatConvNet this is almost trivial as it builds on the easy-to-use GPU support in MATLAB. You can follow this list of steps to try it out:

1. Clear the models generated and cached in the previous steps. To do this, rename or delete the directories `data/characters-experiment` and `data/characters-jit-experiment`.
2. Make sure that MatConvNet is compiled with GPU support. To do this, use

```
> setup('useGpu', true) ;
```
3. Try again training the model of `exercise4.m` switching to `true` the `useGpu` flag.

Task: Follow the steps above and note the speed of training. How many images per second can you process now?

For these small images, the GPU speedup is probably modest (perhaps 2-5 fold). However, for larger models it becomes really dramatic (>10 fold).

Part 5: using pretrained models

A characteristic of deep learning is that it constructs *representations* of the data. These representations tend to have a universal value, or at least to be applicable to an array of problems that transcends the particular task a model was trained for. This is fortunate as training complex models requires weeks of work on one or more GPUs or hundreds of CPUs; these models can then be frozen and reused for a number of additional applications, with no or minimal additional work.

In this part we will see how MatConvNet can be used to download and run high-performance CNN models for image classification. These models are trained from 1.2M images in the ImageNet datasets to discriminate 1,000 different object categories.

Several [pertained models](#) can be downloaded from the MatConvNet website, including several trained using other CNN implementations such as Caffe. One such models is included in

the practical `data/imagenet-vgg-verydeep-16.mat` file. This is one of the best models from the ImageNet ILSVCR Challenge 2014.

Part 5.1: load a pre-trained model

The first step is to load the model itself. This is in the format of the `vl_simplenn` CNN wrapper, and ships as a MATLAB `.mat` file:

```
net = load('data/imagenet-vgg-verydeep-16.mat') ;  
vl_simplenn_display(net) ;
```

Tasks:

1. Look at the output of `vl_simplenn_display` and understand the structure of the model. Can you understand why it is called “very deep”?
2. Look at the size of the file `data/imagenet-vgg-verydeep-16.mat` on disk. This is *just the model*.

Part 5.2: use the model to classify an image

We can now use the model to classify an image. We start from `peppers.png`, a MATLAB stock image:

```
% obtain and preprocess an image  
im = imread('peppers.png') ;  
im_ = single(im) ; % note: 255 range  
im_ = imresize(im_, net.normalization.imageSize(1:2)) ;  
im_ = im_ - net.normalization.averageImage ;
```

The code normalises the image in a format compatible with the model `net`. This amounts to: converting the image to `single` format (but with range 0,...,255 rather than [0, 1] as typical in MATLAB), resizing the image to a fixed size, and then subtracting an average image.

It is now possible to call the CNN:

```
% run the CNN
res = vl_simplenn(net, im_) ;
```

As usual, `res` contains the results of the computation, including all intermediate layers. The last one can be used to perform the classification:

```
% show the classification result
scores = squeeze(gather(res(end).x)) ;
[bestScore, best] = max(scores) ;

figure(1) ; clf ; imagesc(im) ;
title(sprintf('%s (%d), score %.3f',...
    net.classes.description{best}, best, bestScore)) ;
```

That completes this practical.

Links and further work

- The code for this practical is written using the software package [MatConvNet](#). This is a software library written in MATLAB, C++, and CUDA and is freely available as source code and binary.
- The ImageNet model is the *VGG very deep 16* of Karen Simonyan and Andrew Zisserman.

Acknowledgements

Beta testing by: Karel Lenc and Carlos Arteta.

Bugfixes/typos by: Sun Yushi.