Problem 1  Geometric verification

*Geometric verification* starts with $M$ putative matches $(x_1, x'_1), \ldots, (x_M, x'_M)$ between feature points extracted from two images and searches for the largest subset of them that is compatible with a simple overall image transformation (the subset of inliers).

For simplicity, in this exercise we will assume that image transformations are *similarities* of the type

$$x' = sRx + T$$

where $s \in \mathbb{R}_+$ is a scaling factor, $R$ is a $2 \times 2$ rotation matrix, and $T$ is a 2D translation. The goal of geometric verification is to find the similarity transformation that fits accurately the largest possible subset of feature matches:

$$(s^*, R^*, T^*) = \arg \max_{s, R, T} | \{ i : \| x'_i - sRx_i - T \| \leq \epsilon \} |$$

where $\epsilon$ is the tolerance for considering a match correct and $| \cdot |$ denotes the cardinality (number of elements) of a set.

This is a highly non-linear problem which is usually solved by using RANSAC. Recall that RANSAC repeats the following two steps:

1. It samples at random a minimal subset of matches, just enough to be able to estimate $(s, R, T)$ from them.

2. It scores the estimate $(s, R, T)$ by counting how many matches agree with it, in the sense specified above.

This is repeated a certain number of times $t$ in the hope that at least one of the randomly-sampled subsets of matches is composed only of inliers. Such a subset should result in the correct similarity transformation being estimated, which in turn should result in a large number of inliers.

Answer the following questions:

1. How many matches $m \ll M$ are needed to estimate a similarity transformation $(s, R, T)$?

2. Assuming that only a fraction $p = 1/5$ of matches are inliers, estimate the smallest number of RANSAC trials $t$ needed such that a minimal set of inliers is found with probability at least 99%. 

3. In practice, with covariant feature detectors we do not have just feature points but more complex features. For example, in the case of SIFT, each feature is a circle with a center $x$, a radius $\rho$, and an orientation $\theta$. Explain how RANSAC can be modified in order to take advantage of this information.

**Solution to problem 1**

1. Each point match provides 2 constraints ($x$ and $y$ translation) and a similarity transformation has four degrees of freedom. Hence the minimal set is composed of just 2 matches.

2. The probability of sampling at random a set of of $m$ inliers is approximately $p^m$. Thus the probability of failing to sample a set of inliers for $t$ consecutive iterations is $(1 - p^m)^t$. We would like to pick $t$ large enough such that this probability less than 1%:

   $$(1 - p^m)^t \leq 10^{-2} \quad \Rightarrow \quad t \geq \left\lceil \frac{\log 10^{-2}}{\log(1 - p^m)} \right\rceil = 113.$$  

3. If features have more geometric structure than 2D points, this structure can induce additional constraints for RANSAC. For instance, in the case of the SIFT detector, each feature has a centre $x$, a radius $\rho$, and an angle $\theta$. Given a pair of matching SIFT features $(x, \rho, \theta), (x', \rho', \theta')$ in two images related by a similarity transformation $(s, R, T)$, we can write the constraints:

   $$x' = sR x + T, \quad \rho' = s \rho, \quad R(\theta') = R \cdot R(\theta),$$

where $R(\theta)$ is a 2D rotation of $\theta$ radians. This effectively provides 4 constraints (instead of the 2 that points would provide) which are enough to determine the similarity completely.

We conclude that in this case the minimal set is composed of a single match. This is beneficial as now one requires only

   $$t \geq \left\lceil \frac{\log 10^{-2}}{\log(1 - p^m)} \right\rceil = 20$$

RANSAC trials to find the inliers with the required probability.

**Problem 2  Precision and recall**

Consider an image retrieval system for a database containing 7 images in total. Given a certain query, the system responds with a ranked list of results whose relevances to the query are, in order of decreasing confidence, $(+1, -1, +1, -1, -1, +1, -1)$ (where $+1$ means relevant and $-1$ means not relevant).

1. Plot the precision-recall curve and compute the average precision for this result.

2. How would the relevances, precision-recall curve, and average precision change for a query perfectly answered? And for the worst possible answer?

**Solution to problem 2**

1. Each ranked result corresponds to a certain point in the precision-recall curve, as follows:
<table>
<thead>
<tr>
<th>rank</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>relevance</td>
<td>+1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>precision</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>recall</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

The average precision can be computed as the area under this curve. In more detail, it is the average of the precision at each relevant result:

\[ AP = \frac{1}{3} \left( 1 + \frac{2}{3} + \frac{3}{6} \right) = 72\%. \]

2. The best possible ranked list has all the relevant results upfront (+1, +1, +1, -1, -1, -1, -1), and has AP equal to 1. The worst possible ranked list is the reverse, (-1, -1, -1, -1, +1, +1, +1), which has AP

\[ AP = \frac{1}{3} \left( \frac{1}{5} + \frac{2}{6} + \frac{3}{7} \right) = 32\%. \]

Note that in more realistic cases the worst possible AP is almost zero since there is a vast majority of irrelevant results for each query.

**Problem 3  Tensor derivatives**

1. The vec operator reshapes a tensor \( \mathbf{x} \) into a column vector \( \text{vec} \mathbf{x} \). The specific ordering of the tensor elements in the resulting vector is arbitrary, but fixed. In PyTorch, a tensor \( \mathbf{x} \in \mathbb{R}^{C \times H \times W} \) can be vectorized using the command \( \mathbf{x} \text{.view}(-1) \). Using PyTorch’s convention, consider an element \( x_{cvu} \) in \( \mathbf{x} \) and find out the index \( i \) of the corresponding element \( \text{[vec} \mathbf{x}]_i \) in \( \text{vec} \mathbf{x} \).

2. Consider a tensor function \( \mathbf{y} = f(\mathbf{x}) \) where \( \mathbf{x} \in \mathbb{R}^{U_1 \times U_2 \times U_3} \) and \( \mathbf{y} \in \mathbb{R}^{V_1 \times V_2 \times V_3} \). The vec operator can be used to define the Jacobian of the vectorized function:

\[ \frac{d \text{vec} f}{d \text{vec} \mathbf{x}}. \] (1)

Find out the dimensions of this matrix. Then, find out which derivative is stored at row \( i \) and column \( j \), also using PyTorch convention for vectorization.

3. Let \( g \circ f \) be the composition of two scalar functions \( z = g(y) \) and \( y = f(x) \) (hence \( x, y \) and \( z \) are all scalars). Show that

\[ \frac{d(g \circ f)}{dx} = \frac{dg}{dy} \cdot \frac{df}{dx}. \]

Now assume that \( \mathbf{z} = g(\mathbf{y}) \) and \( \mathbf{y} = f(\mathbf{x}) \) are vector functions (in the sense that \( \mathbf{x}, \mathbf{y} \) and \( \mathbf{z} \) are all vectors). Show that:

\[ \frac{d(g \circ f)}{dx} = \frac{dg}{dy} \cdot \frac{df}{dx}. \]

Finally, assume that \( g \) and \( f \) are tensor functions (in the sense that \( \mathbf{x}, \mathbf{y} \) and \( \mathbf{z} \) are all tensors). Show that

\[ \frac{d \text{vec}(g \circ f)}{d \text{vec} \mathbf{x}} = \frac{d \text{vec} g}{d \text{vec} \mathbf{y}} \cdot \frac{d \text{vec} f}{d \text{vec} \mathbf{x}}. \]
Solution to problem 3

1. PyTorch uses the row-major convention. Using zero-based indexing, since $0 \leq u < W$, $0 \leq v < H$ and $0 \leq c < C$:

$$i = \mu(c, v, u) = HW \cdot c + W \cdot v + u.$$  

Note that $0 \leq i < CHW$. This map is invertible, with inverse $\eta(i; C, H, W) = (c, v, u)$ where

$$u = i \mod W, \quad v = \left\lfloor \frac{i}{W} \right\rfloor \mod WH, \quad c = \left\lfloor \frac{i}{HW} \right\rfloor.$$

2. With reference to the previous question, we can interpret $C = U_1$ as the number of channels, $H = U_2$ as the height, and $W = U_3$ as the width. The Jacobian is a $(V_1 V_2 V_3) \times (U_1 U_2 U_3)$ matrix. Entry $(i, j)$ contains the derivative of the corresponding input and output elements of $f$, using the vectorization order:

$$\left[ \frac{df}{dx} \right]_{ij} = \frac{f \eta(i; V_1, V_2, V_3)}{dx \eta(j; U_1, U_2, U_3)}$$

where the index conversion function $\eta$ is given in the answer to question (1).

3. The first equation is a direct application of the chain rule for scalar functions. To prove the second equation, consider the partial derivative of each output element $z_i$ with respect to each input element $x_i$ and then use the total derivative rule:

$$\left[ \frac{d(g \circ f)}{dx} \right]_{ij} = \left[ \frac{dz}{dx} \right]_{ij} = \sum_k \frac{\partial z_i}{\partial y_k} \frac{\partial y_k}{\partial x_j} = \frac{dz_i}{dy} \frac{dy}{dx_j}.$$

Finally, the last equation is obtained immediately from the rule above by using the vec operator to convert tensor quantities into vectors.

Problem 4 Back-propagation algorithm

1. Given a deep neural network $h = f_n \circ f_{n-1} \circ \cdots \circ f_1$ where $x_i = f_i(x_{i-1})$ are all tensor functions show that:

$$\frac{d \text{vec} h}{d \text{vec} x_0} = \frac{d \text{vec} f_n}{d \text{vec} x_{n-1}} \frac{d \text{vec} f_{n-1}}{d \text{vec} x_{n-2}} \cdots \frac{d \text{vec} f_1}{d \text{vec} x_0}.$$

2. Assume that all tensors in the above expressions have the same size $C \times H \times W$, where $H = W = 16$ and $C = 256$. Find the size of the Jacobian matrices in eq. (2) and find out how much memory is required to store them in IEEE single precision format.

3. Find out the size of the first matrix $d \text{vec} f_n/d \text{vec} x$, but this time assuming that its output is a scalar. Compare it to the result above.
4. Now consider the penultimate layer \( f_{n-1} \). Let \( \mathbf{p}_{n-1} \) a fixed tensor that has the same size as \( \mathbf{x}_{n-1} \). Use this tensor to build the new projected” function \( \langle \mathbf{p}_{n-1}, f_{n-1}(\mathbf{x}_{n-2}) \rangle \). The derivative of this function is the backward” version of \( f_{n-1} \):

\[
\mathbf{p}_{n-2} = f_{n-1}^{\text{BP}}(\mathbf{x}_{n-2}, \mathbf{p}_{n-1}) = \frac{d\langle \mathbf{p}_{n-1}, f_{n-1}(\mathbf{x}_{n-2}) \rangle}{d\mathbf{x}_{n-2}}
\]

Find the size of the tensor \( \mathbf{p}_{n-2} \).

5. Suppose that \( f_n \) has a scalar output as before and that you are given an efficient implementation of the backward functions \( f_n^{\text{BP}}(\mathbf{x}_{n-1}, \mathbf{p}_n) \) and \( f_{n-1}^{\text{BP}}(\mathbf{x}_{n-2}, \mathbf{p}_{n-1}) \). Show how this can be used to compute efficiently the derivative

\[
\frac{d\text{vec}(f_n \circ f_{n-1})}{d\text{vec} \mathbf{x}_{n-2}}.
\]

Find out how much memory is required to store intermediate results.

6. Extend the reasoning above to show that, provided that all layers have an efficient backward function implementation, then one can compute efficiently the overall derivative of eq. (2). Under the assumptions above on the dimensions of the tensors, compute the amount of memory required to run this algorithm.

**Solution to problem 4**

1. The equation is obtained from a direct application of the results in the previous problem.

2. The matrices have size \( CHW \times CHW = 65,536 \times 65,536 \), which comprises 4 billion entries and requires 16 GB of memory if stored in IEEE single precision (four bytes per element).

3. If the last function has scalar output, then the corresponding Jacobian matrix is \( 1 \times CHW \). This matrix requires only stored in just 256 KB of memory.

4. The tensor \( \mathbf{p}_{n-2} \) is the derivative of a scalar function \( \langle \mathbf{p}_{n-1}, f_{n-1}(\mathbf{x}_{n-2}) \rangle \) with tensor argument \( \mathbf{x} \), or in other words a gradient. Therefore \( \mathbf{p}_{n-2} \) is a tensor that has the same size as \( \mathbf{x}_{n-2} \).

5. We need to compute the product of Jacobian matrices

\[
\frac{d\text{vec} \mathbf{h}}{d\text{vec} \mathbf{x}_0} = \frac{df_n}{d\text{vec} \mathbf{x}_{n-1}} \cdot \frac{df_{n-1}}{d\text{vec} \mathbf{x}_{n-2}}.
\]

The first factor can be computed as

\[
\frac{df_n}{d\text{vec} \mathbf{x}_{n-1}} = \left( \text{vec} \frac{df_n}{d\mathbf{x}_{n-1}} \right)^\top = \left( \text{vec} \frac{d\langle 1, f_n \rangle}{d\mathbf{x}_{n-1}} \right)^\top = \left( \text{vec} f_n^{\text{BP}}(\mathbf{x}_{n-1}, 1) \right)^\top = (\text{vec} \mathbf{p}_{n-1})^\top
\]

The product can now be computed as:

\[
(\text{vec} \mathbf{p}_{n-1})^\top \cdot \frac{df_{n-1}}{d\text{vec} \mathbf{x}_{n-1}} = \frac{d\langle \mathbf{p}_{n-1}, f_{n-1} \rangle}{d\text{vec} \mathbf{x}_{n-1}} = \left( \text{vec} \frac{d\langle \mathbf{p}_{n-1}, f_{n-1} \rangle}{d\mathbf{x}_{n-1}} \right)^\top = \left( f_{n-1}^{\text{BP}}(\mathbf{x}_{n-2}, \mathbf{p}_{n-1}) \right)^\top = (\text{vec} \mathbf{p}_{n-2})^\top.
\]

Up to reshaping the results, the two steps can be summarised as:

\[
\mathbf{p}_{n-1} = f_n^{\text{BP}}(\mathbf{x}_{n-1}, 1)
\]

\[
\mathbf{p}_{n-2} = f_{n-1}^{\text{BP}}(\mathbf{x}_{n-2}, \mathbf{p}_{n-1})
\]
6. In the forward pass, the network is evaluated starting from the input $x_0$, resulting in tensors:

\[
\begin{align*}
  x_0 & \quad 256 \text{ KB} \\
  x_1 &= f_1(x_0) \quad 256 \text{ KB} \\
  x_2 &= f_2(x_1) \quad 256 \text{ KB} \\
  &\quad \vdots \\
  x_{n-1} &= f_{n-1}(x_{n-2}) \quad 256 \text{ KB} \\
  x_n &= f_n(x_{n-1}) \quad 4B 
\end{align*}
\]

Note that intermediate results but but $x_n$ need to be remembered for backpropagation as these are needed in the backward version of each layer.

In the backward pass, BP moves backward from the last to the first layer in the network:

\[
\begin{align*}
  p_{n-1} &= f_{n-1}^{BP}(x_{n-1}, 1) \quad 256 \text{ KB} \\
  p_{n-2} &= f_{n-1}^{BP}(x_{n-2}, p_{n-1}) \quad 256 \text{ KB} \\
  &\quad \vdots \\
  p_1 &= f_2^{BP}(x_1, p_2) \quad 256 \text{ KB} \\
  p_0 &= f_1^{BP}(x_0, p_1) \quad 256 \text{ KB} 
\end{align*}
\]

As we are only interested in the final derivative $p_0$, as backpropagation progresses the intermediate tensors $p_i, x_i$ can now be forgotten.

Thus the maximum amount of memory is occupied when the first tensor $p_{n-1}$ is computed, as at that time BP has in memory $(x_0, \ldots, x_{n-1}, p_{n-1})$.

**Problem 5  Layers and backward mode**

This exercise looks at the problem of defining a new layer in a CNN and implementing its forward and backward modes. Consider in particular the layer $y = f(x)$ which subtracts from each feature channel of $x$ the corresponding mean:

\[
y_{cv} = x_{cv} - \frac{1}{HW} \sum_{u=0}^{(H,W)-1} x_{cu}.
\]

Recall that we use index base 0 and we use the multi-index $u = (u_1, u_2)$ for compactness. A related layer, called batch normalisation, is very commonly used in recent CNN architectures to improve their numerical conditioning.

1. Find an analytical expression for the backward function $f^{BP}$.
2. Write MATLAB or Python functions that implement $f$ and $f^{BP}$.
3. In writing the MATLAB or Python function that implements $f^{BP}$, did you need to explicitly compute the very large Jacobian matrix $d\text{vec } f/d\text{vec } x$? If so, can you rewrite your implementation to be memory efficient?
4. Computing derivatives is error-prone. Verify your implementation of \( f \) and \( f^{BP} \) numerically. In order to do so, recall that, given a scalar function \( g(x) \), its derivative can be approximated by finite differences:
\[
\frac{dg}{dx}(x) \approx \frac{g(x + \delta) - g(x)}{\delta}.
\]
Use this fact to write a MATLAB or Python program that computes an approximation of the output of \( f^{BP}(x, p) \) by taking the differences of forward evaluations of \( f(x) \) (you can use a random but fixed value of \( x \) and \( p \) for this experiment). Then, check that this approximation and \( f \) produce nearly the same answer. If this is not the case, then your code is buggy and should be fixed.

**Solution to problem 5**

1. Recall that the backward mode function computes the derivative of the projected function \( (p, f(x)) \). Next, we compute explicitly the entry \( cv \) of this matrix:

\[
\left[f^{BP}(x; p)\right]_{cu} = \frac{\partial (p, f(x))}{\partial x_{cu}}
= \frac{\partial}{\partial x_{cu}} \sum_{kv} p_{kv} \cdot y_{kv}
= \frac{\partial}{\partial x_{cu}} \sum_{kv} p_{kv} \left(x_{kv} - \frac{1}{HW} \sum_w x_{kw}\right)
= \sum_{kv} p_{kv} \left(\frac{\partial x_{kv}}{\partial x_{cu}} - \frac{1}{HW} \sum_w \frac{\partial x_{kw}}{\partial x_{cu}}\right)
= \sum_{kv} p_{kv} \left(\delta_{kv=cu} - \frac{1}{HW} \sum_w \delta_{kw=cu}\right)
= \sum_{kv} p_{kv} \left(\delta_{kv=cu} - \frac{1}{HW} \delta_{k=e}\right)
= p_{cu} - \frac{1}{HW} \sum_v p_{cv}.
\]

Here \( \delta_A \) is a function equal to 1 if the condition \( A \) is satisfied and to 0 otherwise (Kronecker delta).

Thus for this particular layer the backward mode function is the same function as the forward mode, but applied to \( p \) instead of \( x \):

\[f^{BP}(x; p) = f(p)\].

2. A MATLAB program implementing \( f \) and \( f^{BP} \) and verifying them numerically is given below:

```matlab
function bpver()
H = 4 ;
W = 4 ;
C = 3 ;
```
delta = 1e-4;
x = randn(H,W,C);
y = f(x);
dy = randn(size(y));
dx = df(x,dy);

dx_ = zeros(size(x));
for i = 1:numel(x)
x_ = x;
x_(i) = x_(i) + delta;
y_ = f(x_);
dx_(i) = dy(:)' *(y_(:) - y(:)) / delta;
end
fprintf('max rel error: %g
', max(abs((dx_(:)-dx(:))./dx(:))));

function y = f(x)
y = bsxfun(@minus, x, mean(mean(x,1),2));

function dx = df(x,dy)
dx = f(dy);

Note that the backward mode is extremely memory efficient (just as the forward mode), despite the fact that, in principle, this code is multiplying a vector by a very large Jacobian matrix. The reason is that the code implicitly exploits layer-specific simplifications, captured by the analytical expression for $f^{BP}$ found above.

**Problem 6  Convolutional networks**

This problem studies a simple but realistic CNN for classification of small images. The model is known as LeNet and is one of the earliest deep convolutional neural networks, designed for handwritten digit classification. The network takes as input a $28 \times 28$-dimensional image of a digit and produces a 10-dimensional vector of class probabilities, one for each of 10 possible digits.

LeNet uses four standard layer types: convolution, max pooling, ReLU and softmax cross-entropy loss. The following table shows how these layers are combined and configured:

<table>
<thead>
<tr>
<th>layer type</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>filt. shape</td>
<td>–</td>
<td>$1 \times 5 \times 5$</td>
<td>$2 \times 2$</td>
<td>$20 \times 5 \times 5$</td>
<td>$2 \times 2$</td>
<td>$50 \times 4 \times 4$</td>
<td>–</td>
<td>$500 \times 1 \times 1$</td>
<td>–</td>
</tr>
<tr>
<td>num. filt.</td>
<td>–</td>
<td>20</td>
<td>–</td>
<td>50</td>
<td>–</td>
<td>500</td>
<td>–</td>
<td>10</td>
<td>–</td>
</tr>
<tr>
<td>stride</td>
<td>–</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>–</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>pad</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td>0</td>
<td>–</td>
</tr>
</tbody>
</table>

1. Given that the input tensor $x_0$ has shape $1 \times 28 \times 28$, find out the shapes of all other tensors $x_1, \ldots, x_8$. Compute also the memory size of each tensor (in bytes) and the memory requirements for backpropagation.
2. The \textit{receptive field} of element \([x_t]_{cvu}\) of tensor \(x_t\) is the subset of the input image \(x_0\) that can affect \([x_t]_{cvu}\). Calculate the size of the receptive fields for the elements of tensors \(x_1, \ldots, x_8\) for LeNet.

\textbf{Solution to problem 6}

The full table is given by:

<table>
<thead>
<tr>
<th>layer type</th>
<th>0 input</th>
<th>1 conv</th>
<th>2 mpool</th>
<th>3 conv</th>
<th>4 mpool</th>
<th>5 conv</th>
<th>6 relu</th>
<th>7 conv</th>
<th>8 softmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>filt. shape</td>
<td>–</td>
<td>1 × 5 × 5</td>
<td>2 × 2</td>
<td>20 × 5 × 5</td>
<td>2 × 2</td>
<td>50 × 4 × 4</td>
<td>–</td>
<td>500 × 1 × 1</td>
<td>–</td>
</tr>
<tr>
<td>num. filt.</td>
<td>–</td>
<td>20</td>
<td>–</td>
<td>50</td>
<td>–</td>
<td>500</td>
<td>–</td>
<td>10</td>
<td>–</td>
</tr>
<tr>
<td>stride</td>
<td>–</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>–</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>pad</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>data shape</td>
<td>1 × 28 × 28</td>
<td>20 × 24 × 24</td>
<td>20 × 12 × 12</td>
<td>50 × 8 × 8</td>
<td>50 × 4 × 4</td>
<td>500 × 1 × 1</td>
<td>10 × 1 × 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>data size</td>
<td>3KB</td>
<td>45KB</td>
<td>11KB</td>
<td>12KB</td>
<td>3KB</td>
<td>2KB</td>
<td>40B</td>
<td>4B</td>
<td>4B</td>
</tr>
<tr>
<td>r.f. size</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>14</td>
<td>16</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
</tbody>
</table>

1. For the ReLU layer, the output shape is the same as the input shape. For the softmax log-loss layer, the output is scalar. For the convolutional layers, we can use the formula \(O = \lceil(I - F + 2P)/S\rceil + 1\) in the notes to compute the output spatial resolution \(O\) given the input spatial resolution \(F\), the padding \(P\) and the stride \(S\). The number of feature channels \(C = K\) of the output is the same as the number \(K\) of filters in the bank.

Given these formulas, it is easy to complete the data shape and data size rows of the table. The memory required for backpropagation is about the same as the memory required to store all such tensors, as seen in Problem 5.

2. From the notes, we can recover the receptive field size in layer 0 of a neuron in layer \(n\) using the formula \(R_{n-0} = 1 + \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} S_j\right) (F_i - 1)\) where \((F_n, S_n)\) are the filter size and stride for each layer. Hence:

\[
\begin{array}{c|c|c|c|c}
 n & F_n & S_n & \prod_{j=1}^{n-1} S_j & R_n \\
\hline
 0 & - & - & - & 1 \\
 1 & 5 & 1 & 1 & 5 \\
 2 & 2 & 2 & 1 & 6 \\
 3 & 5 & 1 & 2 & 14 \\
 4 & 2 & 2 & 2 & 16 \\
 5 & 4 & 1 & 4 & 28 \\
 6 & 1 & 1 & 4 & 28 \\
 7 & 1 & 1 & 4 & 28 \\
 8 & 1 & 1 & 4 & 28 \\
\end{array}
\]