Problem 1  Instantaneous image motion

1. A scene $X$ has instantaneous translational velocity $V$ and angular velocity $\Omega$ relative to a perspective camera with focal length $f$. Derive an expression for the projected image motion, assuming that the image plane lies in front of the optic centre.

2. Explain the origin of the depth-speed scaling ambiguity.

3. The scene now moves such that $\Omega = 0$. Explain how you could solve graphically for the focus of expansion using the position and motion of (at least) two image points.

4. The minimal problem involved intersecting two lines. State the condition under which the construction is (i) most ill-conditioned (ii) best conditioned.

5. Suppose now that the scene is also rotating, but the rotation is known. Show that the translation can be determined (up to the depth-speed scaling ambiguity) by measuring the motion field at 2 points.

Solution to problem 1

By definition of linear and angular velocity of the point $X$ relative to the camera centre one has

$$\dot{X} = V + \Omega \times X,$$

where

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \begin{bmatrix} \dot{V}_x \\ \dot{V}_y \\ \dot{V}_z \end{bmatrix} + \begin{bmatrix} \Omega_x \\ \Omega_y \\ \Omega_z \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} V_x + \Omega_y Z - \Omega_z Y \\ V_y + \Omega_z X - \Omega_x Z \\ V_z + \Omega_x Y - \Omega_y X \end{bmatrix}.$$

1. Let

$$\begin{bmatrix} x \\ y \end{bmatrix} = f \begin{bmatrix} \frac{X}{Z} \\ \frac{Y}{Z} \end{bmatrix}$$

be the point $X$ projected on the image plane. Then the time derivative $\dot{x}$ of the $x$ coordinate of the projected point is

$$\dot{x} = f \frac{\dot{X}}{Z} = f \frac{V_x + \Omega_y Z - \Omega_z Y}{Z} = f \frac{V_z + \Omega_x Y - \Omega_y X}{Z}.$$

A similar expression is obtained for $\dot{y}$. 

$$(1)$$
2. The previous expressions for $\dot{x}$ and $\dot{y}$ depend only on the projected coordinates $x, y$, the angular velocities $\Omega_x, \Omega_y, \Omega_z$, and the scaled linear velocities $V_x/Z, V_y/Z, V_z/Z$. Therefore the projected instantaneous motion does not change if the linear velocity and the depth are scaled by the same amount.

3. If $\Omega = 0$ then
   
   \[
   \dot{x} = f \frac{V_x}{Z} - x \frac{V_z}{Z}, \quad \dot{y} = f \frac{V_y}{Z} - y \frac{V_z}{Z}. \tag{2}
   \]

   The focus of expansion (FoE) is the image point with zero projected motion, i.e. $\dot{x} = 0$ and $\dot{y} = 0$. Substituting into (2) and solving for the pixel coordinates $x$ and $y$ yields
   
   \[
   x = f \frac{V_x}{V_z} \overset{\text{def}}{=} v_x, \quad y = f \frac{V_y}{V_z} \overset{\text{def}}{=} v_y.
   \]

   Thus the FoE is given by the projection $(v_x, v_y)$ of the linear velocity vector on the image plane. To find the point $(v_x, v_y)$ graphically, consider now an image point $(x, y)$ that is not the FoE and its velocity $(\dot{x}, \dot{y})$. Using (2) and rearranging the terms one obtains:
   
   \[
   v_x = f \frac{V_x}{V_z} = x + \lambda \dot{x}, \quad v_y = f \frac{V_y}{V_z} = y + \lambda \dot{y}, \quad \lambda = \frac{Z}{V_z}.
   \]

   While $\lambda$ is unknown, letting it vary traces a line that contains the FoE. Therefore, given a second point $(x', y')$, a second line can be found and their intersection gives the FoE.

4. The construction is best conditioned when $(x, y)$ and $(x', y')$ are at a right angle w.r.t. the FoE and most ill-conditioned when $(x, y)$ and $(x', y')$ form an angle of zero or 180 degrees w.r.t. the FoE.

5. Consider a projected point $x, y$ let $\dot{x}, \dot{y}$ be its measured velocity. Assume that the angular velocity $\Omega$ is known. Then in the equation
   
   \[
   \dot{x} = f \frac{V_x}{Z} + f \Omega_y - y \Omega_z - x \frac{V_z}{Z} - \frac{xy}{f} \Omega_x + \frac{x^2}{f} \Omega_y
   \]

   all the terms except $V_x/Z$ and $V_y/Z$ are known. Writing a similar equation for $y$ results in two linear equations in four unknowns $V_x, V_y, V_z$ and $Z$:
   
   \[
   f \frac{V_x}{Z} - x \frac{V_z}{Z} = \dot{x} - f \Omega_y + y \Omega_z + \frac{xy}{f} \Omega_x - \frac{x^2}{f} \Omega_y \overset{\text{def}}{=} a,
   \]
   \[
   f \frac{V_y}{Z} - y \frac{V_z}{Z} = b
   \]

   Due to the depth-scale ambiguity, the solution will be determined up to a scaling factor no matter how many more measures one takes; to fix the scaling factor, set without loss of generality $Z = 1$. This is still insufficient to solve the problem so we consider a second point.
yielding two additional equations, obtaining overall the system
\[
\begin{align*}
\frac{fV_x}{Z} - \frac{xV_z}{Z} &= a, \\
\frac{fV_y}{Z} - \frac{yV_z}{Z} &= b, \\
\frac{fV_x}{Z'} - \frac{x'V_z}{Z'} &= a', \\
\frac{fV_y}{Z'} - \frac{y'V_z}{Z'} &= b', \\
Z &= 1.
\end{align*}
\]

Using these five equations it is possible to solve for the five unknowns \(V_x, V_y, V_z, Z,\) and \(Z'.\)

**Problem 2  \(K\)-means**

\(K\)-means is one of the most popular *vector clustering algorithms*. The input to \(K\)-means is a set of \(n\) data vectors \(d_1, \ldots, d_n \in \mathbb{R}^d\) and the number \(K\) of desired clusters and the output is a set of \(K\) cluster means or centres \(\mu_1, \ldots, \mu_K \in \mathbb{R}^d\). The goal of the algorithm is to find the means such that the average squared distance from the data vectors to their closest means is minimised. Formally, this can be written as the following *energy minimisation problem*:

\[
(\mu_1^*, \ldots, \mu_K^*) = \arg\min_{\mu_1, \ldots, \mu_K} E(\mu_1, \ldots, \mu_K), \quad \text{where } E(\mu_1, \ldots, \mu_K) = \frac{1}{n} \sum_{i=1}^n \min_k \|d_i - \mu_k\|^2.
\]

Solving this problem exactly is very difficult due to its combinatorial nature. \(K\)-means finds an approximate but often acceptable solution as follows:

1. Initialise the means, for example by sampling \(\mu_k\) from a fixed probability distribution on \(\mathbb{R}^d\).
2. Repeat until convergence:
   
   (a) For each data vector \(d_i\), find the index \(\pi_i\) of the mean closer to it:
   
   \[
   \pi_i \leftarrow \arg\min_{k=1, \ldots, K} \|d_i - \mu_k\|^2, \quad i = 1, \ldots, N.
   \]
   
   The indices \(\pi\) are called *assignments* (of the data vectors to the means).

   (b) Update each mean to be equal to the average of the data vectors assigned to it:
   
   \[
   \mu_k \leftarrow \frac{1}{|\{i : \pi_i = k\}|} \sum_{i : \pi_i = k} d_i, \quad k = 1, \ldots, K.
   \]

Read and understand the algorithm, then answer the following questions:

1. Show that step (2.b) updates the means such that the energy \(E(\mu_1, \ldots, \mu_K)\) decreases or stays the same.
2. Use the previous result to show that, after finitely many \(K\)-means iterations, the \(K\)-means energy does not change anymore (i.e. the algorithm converges).
3. Suggest two alternative initialisation strategies for step (1) that are likely to be superior to sampling the initial means from a fixed probability distribution.

4. Suggest which step is likely to be the slowest and determine the order of complexity of a $K$-means iteration as a function of the number of visual words $K$, the vector dimension $D$, and the number of data vectors $N$. Recall from B16 Structured Programming that the order of complexity is the asymptotic order of the number of elementary operations performed by the algorithm.

5. $K$-means is often used to compute a dictionary of visual words by clustering a large number of example local feature descriptors. Assuming that $K$-means converges in $T = 100$ iterations, estimate (up to a constant $C$) the number of operations required to run the algorithm for $K = 10^5$ visual words, $N = 10^6$ data vectors and $D = 128$ dimensional descriptors. Assuming that your computer is able to perform $10^9$ operations in a second (which is roughly speaking the capability of a 1GHz CPU), estimate the time required.

Solution to problem 2

1. Right after updating the assignments $\pi$ in step (2.a), the $K$-means energy is, by definition, given by the distances to the assigned means:

$$E(\mu_1, \ldots, \mu_K) = \frac{1}{n} \sum_{i=1}^{n} \min_k \|d_i - \mu_k\|^2 = \frac{1}{n} \sum_{i=1}^{n} \|d_i - \mu_{\pi_i}\|^2.$$ 

Since each data vector is assigned to exactly one mean, we can break the last summation in a contribution for each mean:

$$E(\mu_1, \ldots, \mu_K) = \frac{1}{n} \sum_{k=1}^{K} \left( \sum_{i: \pi_i = k} \|d_i - \mu_k\|^2 \right).$$

Each term in brackets is the sum of the squared distances of a subset of the data vectors $d_i$ (a cluster) to a fixed vector $\mu_k$ (the mean or cluster centre). It is well known that the vector $\mu_k$ that minimises the total squared distance to vectors $d_i$ is their average. Hence, replacing the means with

$$\mu_k' = \frac{1}{|\{i : \pi_i = k\}|} \sum_{i: \pi_i = k} d_i$$

can only maintain or decrease the energy. We conclude that:

$$E(\mu_1, \ldots, \mu_K) = \frac{1}{n} \sum_{i=1}^{n} \|d_i - \mu_{\pi_i}\|^2 \geq \frac{1}{n} \sum_{i=1}^{n} \|d_i - \mu_{\pi_i}'\|^2 \geq \frac{1}{n} \sum_{k=1}^{K} \min_k \|d_i - \mu_k'\|^2 = E(\mu_1', \ldots, \mu_K').$$

2. Each assignment of the data vectors to the $K$ means results in a certain set of means and a corresponding energy level. Since there are only finitely many such assignments, there are only finitely many possible energy levels. Since at each iteration the algorithm either maintains the energy level or decreases it, after a finite number of iterations the energy level cannot change anymore.
3. The disadvantage of using a fixed distribution is that means may be sampled far away from the data vectors. Since the means should approximate the data (because they should have low average distance to the data vectors), this strategy may be suboptimal. A simple and much better method is to initialise the \( K \) means to \( K \) data vectors selected at random. An even better strategy is to select such data vectors iteratively, picking each time the one farthest away from the means selected so far. The intuition is that this should result in means that cover the data more uniformly than random selection.

4. The bottleneck is the computation of the distances of each of the \( K \) means with each of the \( N \) data vectors, each of which is \( D \) dimensional. An iteration has cost \( O(TKND) \) and \( T \) iterations \( O(TKND) \).

5. Based on the order of complexity computed above, the number of CPU operations is given \( CTKND \) where \( C \) is some constant (typically larger than one).

It is not unusual for image retrieval systems to use a large vocabulary of \( K = 10^5 \) words. As a rule of thumbs, one would like to have at least 10 example data vectors per word in order to estimate well the means. Hence in this case one should run the algorithm on \( N = 10^6 \) example descriptors. For such numbers, \( K \)-means will terminate in

\[
C \times \frac{10^5 \times 10^6 \times 128 \times 10^2}{10^9 \times 60 \times 60 \times 24} \approx C \times 15 \text{ days}.
\]

This is far too slow to be practical. This problem is usually solved by using an algorithm such as a KD-tree to accelerate step (2.a) of \( K \)-means. A KD-tree can quickly determine the nearest mean to a data point by only considering a tiny fraction of all \( K \) possible means.

**Problem 3  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 (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 3

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 \(m\) 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:

\[
x' = sRx + T, \quad \rho' = s\rho, \quad R(\theta') = R \cdot R(\theta),
\]

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 4  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 4

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>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>recall</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</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 5 Tensor derivatives

A convolutional neural network (CNN) is a learnable function that operates on image-like data. Formally, in a CNN a data tensor is an array \(x_i \in \mathbb{R}^{H \times W \times C}\) consisting of \(H\) rows, \(W\) columns, and \(C\) feature channels. A CNN is a function

\[
h(x_0) = (f_n \circ \cdots \circ f_2 \circ f_1)(x_0)
\]

obtained by composing \(n\) intermediate layers

\[
f_i : \mathbb{R}^{H_i-1 \times W_i-1 \times C_{i-1}} \rightarrow \mathbb{R}^{H_i \times W_i \times C_i}, \quad x_{i-1} \mapsto x_i = f_i(x_{i-1}), \quad i = 1, \ldots, n.
\]

Each layer \(f_i\) takes as input a tensor \(x_{i-1}\) and produces as output another tensor \(x_i\). Each layer depends also on a parameter vector \(w_i\), not shown here in order to keep the notation simple.

Learning a CNN usually requires computing the derivative of the function \(h\) with respect to each parameter vector \(w_i\). Here, we discuss the related problem of computing the derivative of \(h\) with respect to the network input \(x_0\) as the resulting algorithm can be used to compute all other required derivatives as well. In this exercise, in particular, we develop a notation for the tensor derivatives that will be needed later.
1. The *stacking* 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 is assumed to be always the same for tensors of a given shape. In MATLAB, a tensor $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ is naturally represented as a 3D array $\mathbf{x}$ with dimensions $H \times W \times C$ whose stacking $\text{vec} \mathbf{x}$ can then be computed as $\mathbf{x}( :)$. Using MATLAB’s convention, find out which tensor element $x_{vuc}$ corresponds to which element $[\text{vec} \mathbf{x}]_i$ in the stacked tensor.

2. We use the symbol $\text{vec} f$ to denote the same function as the tensor function $f$, but operating on stacked input and output.\(^1\) Stacking thus reduces a tensor function to a vector function whose derivative is the Jacobian matrix:

$$
\frac{d \text{vec} f}{d (\text{vec} \mathbf{x})^\top} \in \mathbb{R}^{M \times N}.
$$

This notation formally emphasises that the Jacobian is formed by taking the partial derivatives of each element of the output of $\text{vec} f$, a column vector, with respect to each element of the input $(\text{vec} \mathbf{x})^\top$, a row vector. For a tensor function $f : \mathbb{R}^{H \times W \times C} \rightarrow \mathbb{R}^{H' \times W' \times C'}$, what are the dimensions $M \times N$ of the Jacobian? What is the meaning of the $(i, j)$-th element of the Jacobian matrix?

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

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

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 column vectors). Show that:

$$
\frac{d (g \circ f)}{d \mathbf{x}^\top} = \frac{dg}{d \mathbf{y}^\top} \frac{df}{d \mathbf{x}^\top}.
$$

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

$$
\frac{d \text{vec} (g \circ f)}{d (\text{vec} \mathbf{x})^\top} = \frac{d \text{vec} g}{d (\text{vec} \mathbf{y})^\top} \frac{d \text{vec} f}{d (\text{vec} \mathbf{x})^\top}.
$$

**Solution to problem 5**

1. MATLAB stacks an array $x_{vuc}$ by by enumerating all indices $(v, u, c)$ in such a manner that $v$ varies the fastest and $c$ varies the slowest (i.e. $(1, 1, 1), (2, 1, 1), (3, 1, 1), \ldots, (1, 1, 2), (2, 1, 2), \ldots, (H, W, C)$). Hence

$$
x_{vuc} = [\text{vec} \mathbf{x}]_i \quad \text{where} \quad \begin{cases} 
  v & = ((i - 1) \mod H) + 1, \\
  u & = ([ (i - 1)/H ] \mod W) + 1, \\
  c & = ([ (i - 1)/(HW) ] + 1.
\end{cases}
$$

2. The Jacobian contains the derivatives of each output element with respect to each input element, such that $M = H'W'C'$ and $N = HWC$. The $(i, j)$-th element of the Jacobian is the derivative of the $i$-th output element with respect to the $j$-th input element. Denote by

\(^1\)In symbols: $\text{vec}[f(\mathbf{x})] = (\text{vec } f_i)(\text{vec } \mathbf{x})$. 
\( y = f(x) \) the tensor computed by the function. The \( i \)-th output element is the element \( y_{i^\prime} \) that comes in \( i \)-th position in the stacked tensor \( \text{vec}y \). Similarly, the \( j \)-th input element is the element \( x_{vuc} \) that comes in \( i \)-th position in the stacked tensor \( \text{vec}x \). In short:

\[
\left[ \frac{d\text{vec}f}{d(\text{vec}x)^\top} \right]_{ij} = \frac{\partial y_{i^\prime}x_{vuc}}{\partial x_{vuc}}, \quad y = f(x), \quad y_{i^\prime}x_{vuc} = [\text{vec}y]_i, \quad x_{vuc} = [\text{vec}x]_j.
\]

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:

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

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

**Problem 6 Back-propagation algorithm**

Backpropagation is a memory-efficient method for computing the derivative of function compositions. While the algorithm is fairly general, we discuss it here in the context of learning deep neural networks (DNNs).

For backpropagation with a DNN \( h = f_n \circ \cdots \circ f_1 \), we make the important assumption that the last layer \( f_n \) is scalar-valued, in the sense that \( x_n = x_n \in \mathbb{R} \) (i.e. \( H_n = W_n = C_n = 1 \)). This is usually the case when training a deep network, as the last layer is the loss function, which measures the prediction error (performance) of the network on the training data.

1. Given a deep neural network \( h = f_n \circ f_{n-1} \circ \cdots \circ f_1 \), show that:

\[
\frac{d\text{vec}h}{d(\text{vec}x_0)^\top} = \frac{d\text{vec}f_n}{d(\text{vec}x_{n-1})^\top} \frac{d\text{vec}f_{n-1}}{d(\text{vec}x_{n-2})^\top} \cdots \frac{d\text{vec}f_1}{d(\text{vec}x_0)^\top}.
\]

2. Find an expression for the size of matrices in eq. (4). Assume that the DNN is convolutional and that all tensors \( x_i \) have identical dimensions \( H_i = W_i = 16 \) and \( C = 256 \). Find the amount of memory required to store each factor \( d\text{vec}f_i/d(\text{vec}x_{i-1})^\top \) if scalars are stored in IEEE single precision format. Is it feasible to store such factors explicitly?

3. Repeat the calculation above for the last layer \( f_n \), this time assuming that \( x_n \) is a scalar quantity (whereas the size of the tensor \( x_{n-1} \) remains unchanged). What is the size of the matrix \( d\text{vec}f_n/d(\text{vec}x_{n-1})^\top \)? Can we store it?

4. Consider now computing the derivative of the penultimate layer \( f_n \) projected on a constant tensor \( p \):

\[
\frac{d\langle p,f_{n-1} \rangle}{dx_{n-2}}
\]

Here the symbol \( \langle \cdot, \cdot \rangle \) is the inner product of the tensors \( p \) and \( f_{n-1}(x_{n-2}) \). We use the symbol \( f_{n-1}^{\text{BP}}(x_{n-2}, p) \) to denote the function that computes eq. (5) and we call the latter the backward function of layer \( f_{n-1} \). Note that, in general, the backward function depends on both \( p \) and \( x_{n-2} \).

Find the size of matrix eq. (5). Note that we do not need to use stacking here; why?

\(^2\text{i.e. } \langle p, f_{n-1} \rangle = (\text{vec}p)^\top (\text{vec} f_{n-1}(x_{n-2})).\)
5. Suppose that we are given an algorithm that can compute the backward function \( f_{n-1}^{BP}(x_{n-2}, p) \) efficiently, using using an amount of memory proportional to the size of the function output. Under the assumption that the last layer is scalar-valued, suggest how this algorithm can be used to compute efficiently the derivative of the composition of the last two layers:

\[
\frac{d \, \text{vec}(f_n \circ f_{n-1})}{d(\text{vec} \, x_{n-2})^\top}.
\]

Compute the size of the resulting matrix.

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 eq. (4). Under the assumptions above on the dimensions of the tensors, compute the amount of memory required to run this algorithm.

**Solution to problem 6**

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

2. As seen in the previous problem, the factors are matrices of dimensions

\[
H_i W_i C_i \times H_{i-1} W_{i-1} C_{i-1} = 65,536 \times 65,536.
\]

Such a matrix has 4 billion entries and requires 16 GB of memory if stored in IEEE single precision. Given that there is one such matrix for each network layer, we cannot store all these factors explicitly.

3. For the last layer, \( H_n W_n C_n = 1 \) and the result can be store explicitly as it takes only 256 KB of memory.

4. The function \( g = \langle p, f_{n-1} \rangle \) is scalar-valued just like the last layer. In this case we can but do not need to use the stacking operator to express the derivative. Instead, it is customary to consider the tensor \( dg/dx_{n-2} \) of the partial derivatives of the scalar output with respect to each element of the input tensor \( x_{n-2} \). This tensor of partial derivatives has the same size as \( x_{n-2} \) and can be stored in 256 KB of memory.

5. As seen in the previous exercise, we have:

\[
\frac{d \, \text{vec}(f_n \circ f_{n-1})}{d(\text{vec} \, x_{n-2})^\top} = \frac{d \, \text{vec} \, f_n}{d(\text{vec} \, x_{n-1})^\top} \frac{d \, \text{vec} \, f_{n-1}}{d(\text{vec} \, x_{n-2})^\top}.
\]

Since \( f_n \) is scalar-valued, the first factor can be equivalently written as:

\[
\frac{d \, \text{vec} \, f_n}{d(\text{vec} \, x_{n-1})^\top} = \frac{d \langle 1, f_n \rangle}{d(\text{vec} \, x_{n-1})^\top} = \left( \text{vec} \, \frac{d \langle 1, f_n \rangle}{d x_{n-1}} \right)^\top = \left( \text{vec} \, f_{n-1}^{BP}(x_{n-1}, 1) \right)^\top.
\]

We introduce the short-hand notation

\[
dx_{n-1} = f_{n-1}^{BP}(x_{n-1}, 1).
\]

and write

\[
\frac{d \, \text{vec}(f_n \circ f_{n-1})}{d(\text{vec} \, x_{n-2})^\top} = (\text{vec} \, dx_{n-1})^\top \frac{d \, \text{vec} \, f_{n-1}}{d(\text{vec} \, x_{n-2})^\top} = \left( \text{vec} \, f_{n-1}^{BP}(x_{n-1}, dx_{n}) \right)^\top.
\]

Note that the intermediate results \( dx_{n-1} \) and \( dx_{n-2} \) have the same size as \( x_{n-1} \) and \( x_{n-2} \), respectively.
6. We are now ready to give the complete back-propagation (BP) algorithm. In the forward pass, BP computes all intermediate variables starting from the input $x_0$:

$$
\begin{align*}
x_1 &= f_1(x_0) \\
x_2 &= f_2(x_1) \\
\vdots &= \\
x_{n-1} &= f_{n-1}(x_{n-2}) \\
x_n &= f_n(x_{n-1})
\end{align*}
$$

where the storage requirements are computed based on the assumptions on the dimensions of the tensors $x_i$.

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

$$
\begin{align*}
dx_{n-1} &= f_{n}^{BP}(x_{n-1}, 1) \\
dx_{n-2} &= f_{n-1}^{BP}(x_{n-2}, dx_{n-1}) \\
\vdots &= \\
dx_1 &= f_{2}^{BP}(x_1, dx_2) \\
dx_0 &= f_{1}^{BP}(x_0, dx_1)
\end{align*}
$$

From the table above, back-propagation requires $256 \text{ KB} \times (2n - 1) + 4 \text{ B}$ of storage for the various tensor variables. However, on the way back one can delete the intermediate variables $x_i$ and $dx_i$ as they are not needed anymore, such that the memory required is about half of that: $256 \text{ KB} \times (n - 1) + 4 \text{ B}$.

**Problem 7  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:

$$
y_{v'u'c'} = x_{v'u'c'} - \frac{1}{HW} \sum_{v=1}^{H} \sum_{u=1}^{W} x_{v'u'c'}.
$$

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 functions that implement $f$ and $f^{BP}$.

3. In writing the MATLAB function that implements $f^{BP}$, did you need to explicitly compute the very large Jacobian matrix $d(\text{vec } f)/d(\text{vec } x)^T$? 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 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 7**

1. Recall that the backward mode function computes the derivative of the projected function $p^\top \text{vec} f(x)$. Next, we compute explicitly the entry $vuc$ of this matrix:

$$[f_{BP}(x; p)]_{vuc} = \frac{\partial \langle p, f(x) \rangle}{\partial x_{vuc}}$$

$$= \frac{\partial}{\partial x_{vuc}} \sum_{v'u'c'} p_{v'u'c'} y_{v'u'c'}$$

$$= \frac{\partial}{\partial x_{vuc}} \sum_{v'u'c'} p_{v'u'c'} \left( x_{v'u'c'} - \frac{1}{HW} \sum_{\bar{v}\bar{u}} x_{\bar{v}\bar{u}c'} \right)$$

$$= \sum_{v'u'c'} p_{v'u'c'} \left( \frac{\partial x_{v'u'c'}}{\partial x_{vuc}} - \frac{1}{HW} \sum_{\bar{v}\bar{u}} \frac{\partial x_{\bar{v}\bar{u}c'}}{\partial x_{vuc}} \right)$$

$$= \sum_{v'u'c'} p_{v'u'c'} \left( \delta_{v'=v, u'=u, c'=c} - \frac{1}{HW} \sum_{\bar{v}\bar{u}} \delta_{\bar{v}=v, \bar{u}=u, c'=c} \right)$$

$$= \sum_{v'u'c'} p_{v'u'c'} \left( \delta_{v'=v, u'=u, c'=c} - \frac{1}{HW} \delta_{c'=c} \right)$$

$$= p_{vuc} - \frac{1}{HW} \sum_{v'u'} p_{v'u'clock}.$$ 

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) ;
1
2
3
4
5
6
7```
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 8  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 output a 10-dimensional vector of class probabilities, one for each of 10 possible digits.

There are four layer types:

- The convolution layer $y = \phi_s(x; F, b)$ convolves an input tensor $x \in \mathbb{R}^{H \times W \times C}$ by a bank $F = (f_1, \ldots, f_K)$ of $K$ filters. Each filter has a spatial dimension $L \times L$ and $C$ feature channels (it is therefore a volume) and produces one feature channel of the output tensor $y$ as follows:

$$y_{\nu\nu'\sigma} = b_{\sigma} + \sum_{\bar{v}\bar{u}\sigma} f_{\bar{v}\bar{u}\sigma} x_{\bar{v}+\nu'-1, \bar{u}+\nu-1, \sigma}$$

Here in addition to the filters $F$ there is also a vector $b$ of channel-specific biases.

The output can be downsampled with a stride $s \geq 1$, which means that only one every other output pixels is kept, and the other discarded. Furthermore, the input can be padded with zeros by $P$ pixels at the top/bottom/left/right. Accounting for padding and downsampling, the formula is slightly more complicated:

$$y_{\nu\nu'\sigma} = b_{\sigma} + \sum_{\bar{v}\bar{u}\sigma} f_{\bar{v}\bar{u}\sigma} x_{\bar{v}+S(\nu'-1)-P, \bar{u}+S(\nu-1)-P, \sigma}$$
• The **max pooling** layer \( y = \phi_{\text{MP}}(x) \) computes the local maximum of each feature channel in \( L \times L \) neighbourhoods:

\[
y_{v'u'c'} = \max_{1 \leq v, u \leq L} x_{v+u'-1, u+u'-1, c}.
\]

Just like the convolutional layer, the output can be downsampled \( s \geq 1 \) times and the input can be padded (this time with \(-\infty\)) by an amount \( P \) on all side.

• The **ReLU** (rectified linear unit) \( y = \phi_{\text{ReLU}}(x) \) cuts-off negative values as follows:

\[
y_{v'u'c'} = \max\{0, x_{v'u'c'}\}.
\]

• The **softmax log-loss** layer \( y = \phi_{\text{softmax}}(x, c^*) \) takes a vector of \( C \) class scores \( x \in \mathbb{R}^C \), converts them into probabilities using the softmax operator, and then compares the result to the ground truth class label \( c^* \):

\[
y = -\log \frac{e^{x_{c^*}}}{e^{x_1} + \ldots e^{x_n}}.
\]

Note that, since this is a loss function, the output is a scalar quantity. For uniformity, however, we will treat both \( x \) and \( y \) as tensors, the first having dimension \( 1 \times 1 \times C \) and the second \( 1 \times 1 \times 1 \).

The log-loss is only used for training the network. Once the network is deployed (i.e. used to classify new digits), this layer is replaced with **softmax** alone:

\[
y_c = \frac{e^{x_c}}{e^{x_1} + \ldots e^{x_n}}.
\]

Note that this produces a vector of \( C = 10 \) class probabilities for our problem.

Given these definitions, the following table specifies the structure of the network:

<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>(5 \times 5 \times 1)</td>
<td>(2 \times 2)</td>
<td>(5 \times 5 \times 20)</td>
<td>(2 \times 2)</td>
<td>(4 \times 4 \times 50)</td>
<td>–</td>
<td>(1 \times 1 \times 500)</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 \( 28 \times 28 \times 1 \), 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 **receptive field** of element \( [x_t]_{vuc} \) of tensor \( x_t \) in the CNN is the subset of the input image \( x_0 \) that can affect \( [x_t]_{vuc} \). Calculate the size of the receptive fields for the elements of tensors \( x_1, \ldots, x_8 \) in LeNet.
Solution to problem 8

The full table is given by:

<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>$5 \times 5 \times 1$</td>
<td>$2 \times 2$</td>
<td>$5 \times 5 \times 20$</td>
<td>$2 \times 2$</td>
<td>$4 \times 4 \times 50$</td>
<td>–</td>
<td>$1 \times 1 \times 500$</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>$28 \times 28 \times 1$</td>
<td>$24 \times 24 \times 20$</td>
<td>$12 \times 12 \times 20$</td>
<td>$8 \times 8 \times 50$</td>
<td>$4 \times 4 \times 50$</td>
<td>$1 \times 1 \times 500$</td>
<td>$1 \times 1 \times 10$</td>
<td>$1 \times 1 \times 10$</td>
<td>1</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>2KB</td>
<td>40B</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 layer, we need to compute the output shape from the input and filter shapes. Let the shape of the input be $H \times W \times C$ and of each filter be $\bar{H} \times \bar{W} \times C$. We can workout $\bar{H}$ and the other dimension $\bar{W}$ is obtained in a similar manner.

In order to do so, consider only the vertical dimension $H$ of the input tensor $x$ and $\bar{H}$ of the filter $f$. Assume for now that there is no padding $P = 0$ and unit stride $S = 1$. The filter is applied starting at position $1, 2, \ldots, \bar{H}$ in the input tensor, where $\bar{H}$ is the largest possible position that allows the filter be fully contained in the input tensor. Hence $\bar{H} = H - \bar{H} + 1$. This can be illustrated as follows, where each triangle represents a filter application, generating one output pixel.

A padding $P > 0$ amounts to increasing the width of the of the input tensor by $2P$ pixels. Hence $\bar{H} = H - \bar{H} + 1 + 2P$.

A stride $S > 1$ amounts to retaining one every $S$ output pixels. For example:

In order to compute the size $H'$ after downsampling, note that pixel $H'$ correspond to $1 + S(H' - 1)$, before downsampling. We must have $1 + S(H' - 1) \leq H$, i.e. $H' \leq (H - 1)/S + 1$. 

A padding $P > 0$ amounts to increasing the width of the of the input tensor by $2P$ pixels. Hence $\bar{H} = H - \bar{H} + 1 + 2P$.

A stride $S > 1$ amounts to retaining one every $S$ output pixels. For example:
Since the latter has to be an integer number, \( H' = \lfloor (H - 1)/S \rfloor + 1 \), where \( \lfloor \cdot \rfloor \) means rounding down.

Putting everything together, and accounting for the fact that the horizontal direction works in the same manner, the spatial dimensions of the convolution output \( y \) are

\[
H' = \left\lfloor \frac{H - \bar{H} + 2P}{S} \right\rfloor + 1, \quad W' = \left\lfloor \frac{W - \bar{W} + 2P}{S} \right\rfloor + 1.
\]

The number of feature channels \( C' = K \) 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. Consider a neuron \( z \) in a convolutional neural network and consider an intermediate layer \( y \) below \( z \). The receptive field of \( z \) into \( y \) is the subset of the tensor \( y \) that affects the value of \( z \). Assume that the size of this receptive field is \( R' \). Suppose also that \( y = \phi(x; F, b) \) is generated by a convolutional layer with filter size \((\bar{H}, \bar{W})\) and stride \( S \) (padding is irrelevant here; why?). The situation can be illustrated as follows:

Given this information, we wish to find the size \( R \) of the receptive field of \( z \) in layer \( x \). In order to do so, note that the rightmost pixel of the receptive field of \( z \) in \( y \) has coordinate \( R' \). After undoing the effect downsampling, this pixel has coordinate \( S(R' - 1) + 1 \). The value of this pixel is computed by applying the filter to the tensor \( x \) at the specified shift. The rightmost pixel of \( x \) accessed by this filter application is

\[
R = S(R' - 1) + \bar{H}.
\]

Max pooling layers behave in the same way as convolutional ones. ReLU layers behave like convolutional layers with \( 1 \times 1 \) filters, no padding, and stride 1. Hence, if \( z \) is a neuron in the \( n \)-th tensor, a repeated application of the formula above allows to find its receptive field size in all tensors below it, until the input image \( (n = 0) \) is reached. It is easy to see that this repeated application of the formula above yields

\[
R_n = 1 + \sum_{i=1}^{n} \left( \prod_{j=1}^{i-1} S_j \right) (\bar{H}_i - 1).
\]
In our case, this formula can be evaluated $n = 8$ times to fill the table. More efficiently, by isolating the last term in the summation, we obtain the recursive formula

$$R_n = R_{n-1} + \left( \prod_{j=1}^{n-1} S_j \right) \left( \bar{H}_n - 1 \right)$$

The recursion start at $R_0 = 1$. We can now complete the table:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$S_n$</th>
<th>$\prod_{j=1}^{n-1} S_j$</th>
<th>$\bar{H}_n$</th>
<th>$R_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>$\bar{H}_0$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>28</td>
</tr>
</tbody>
</table>