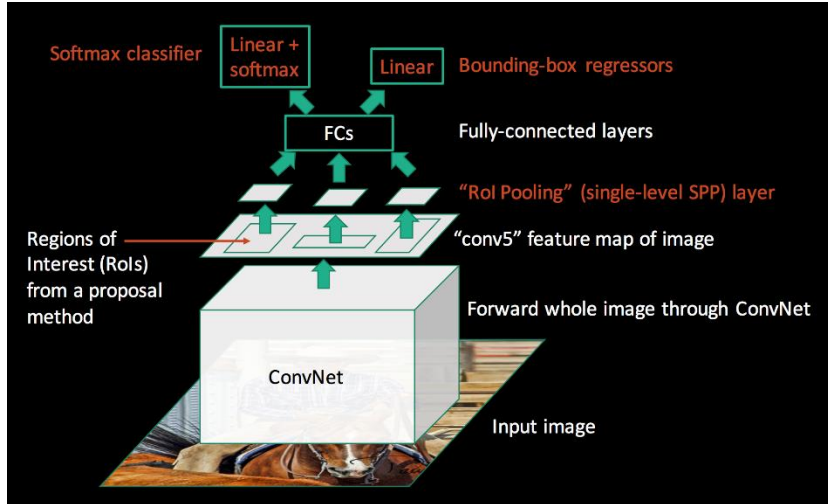


Q1 –

First, we proceed under the assumption that there is familiarity with the architecture and functionality of the RCNN family of models. Before discussing anything further, we will provide explanations regarding the loss function of Fast RCNN.

After applying the backbone network and obtaining the feature map, and then applying the selective search algorithm followed by RoI pooling, we need to output a label (class) and bounding box coordinates using a fully connected network for the output of this layer. Consequently, we will have a loss function for each. As a result, the total loss of the network consists of two parts.



Classification Loss (L_{cls}): This component is responsible for determining the class of the detected object. It is a standard categorical cross-entropy loss computed over the predicted class probabilities and the ground truth class labels. For N classes, the loss can be formulated as:

$$L_{cls}(p, u) = -\log(p_u)$$

where p is the predicted probability distribution over N classes (including the background class), and u is the true class label.

Regression Loss (L_{reg}): This component refines the bounding box coordinates for the detected objects. It uses a smooth L1 loss to regress the predicted bounding box coordinates (t^x, t^y, t^w, t^h) towards the ground truth coordinates $(t^x_*, t^y_*, t^w_*, t^h_*)$:

$$L_{reg}(t^u, v) = \sum_{i \in \{x, y, w, h\}} \text{smooth}_{L1}(t_i^u - v_i)$$

where t^u are the predicted bounding box coordinates for the true class u , and v are the ground truth bounding box coordinates. The smooth L1 loss is defined as:

$$smooth_{L1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

The total loss for Fast R-CNN is a weighted sum of the classification and regression losses:

$$L(p, u, t^u, v) = L_{cls}(p, u) + \gamma[u \geq 1]L_{reg}(t^u, v)$$

Here, gamma is a balancing parameter (typically set to 1), and $[u \geq 1]$ is an indicator function that is 1 if $u \geq 1$ (the class is not background).

Alright, now that we have discussed and examined both main components of the Fast RCNN model's loss function, we need to understand a few other things before analyzing the loss function of the Faster RCNN model.

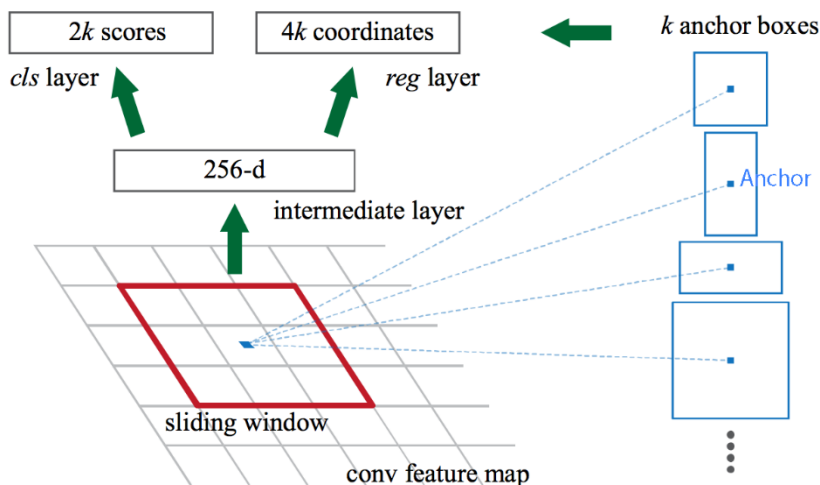
First, we need to know what the differences between these two models are and why it is called "Faster." Second, we need to determine whether this change affects the loss function and how it can lead to improvements and new benefits. Additionally, we need to identify which part of the new loss function relates to these changes and why.

Among the models of RCNN family, both the original RCNN and Fast RCNN use selective search or other algorithms to obtain region proposals. The primary difference between Fast RCNN and Faster RCNN lies in this aspect. Instead of using selective search, which is computationally intensive and time-consuming, Faster RCNN employs a Region Proposal Network (RPN) to extract region proposals using an entirely convolutional network. Assuming familiarity with the architectures, I will provide a brief overview of this model.

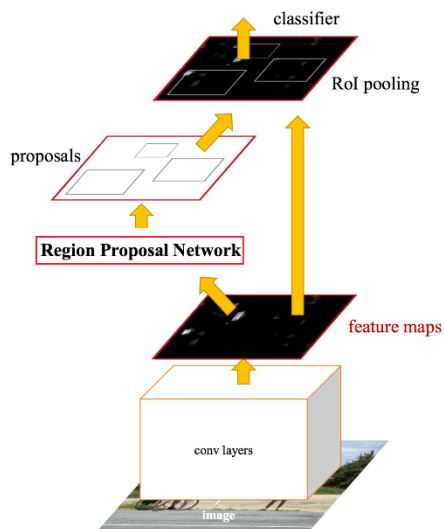
The Faster RCNN model takes the feature map obtained from the backbone model and uses sliding windows with a 3x3 kernel to slide over the feature map to a specified depth (number of channels). The output of each sliding window is a feature map with a depth of 256. The centers of these windows are called anchors, and k is the number of anchor boxes obtained for each anchor. To provide a comprehensive perspective with different scales and shapes, an anchor box is obtained for each configuration. Thus, for each anchor, we have k anchor boxes.

The resulting feature map is then fed into two fully connected networks to produce two outputs:

1. A vector of size $2k$ indicating whether each anchor box contains an object or not.
2. A vector of size $4k$ providing the coordinates for each anchor box.

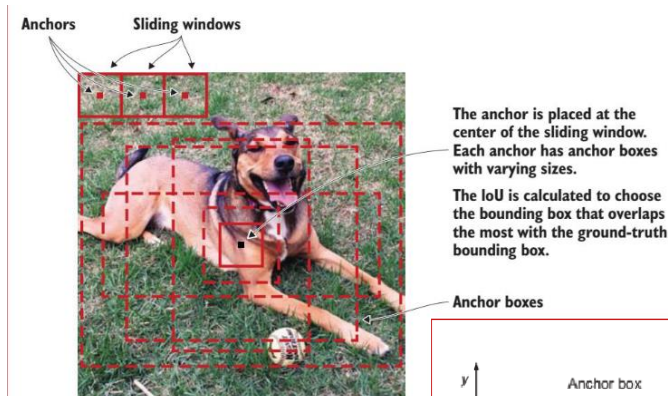


Finally, these outputs, along with the initial feature map (RPN input feature map), are fed into the ROI pooling layer. The remaining process is similar to Fast RCNN, with the only difference being the use of the RPN.

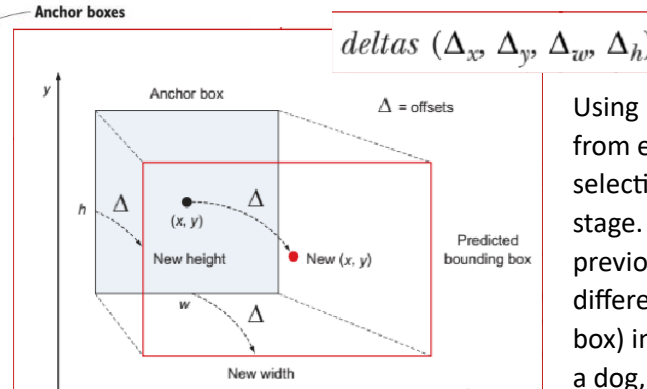


With this in mind, the introduction of the RPN in Faster RCNN not only brings changes to the loss function, which I will discuss further later on, but also leads to significant improvements. These advancements include increased training speed and the creation of an end-to-end model, unlike the previous models. Several notable improvements are listed below:

- **End-to-end training** allows for joint optimization of proposal generation and detection.
- **Increased efficiency and speed** by leveraging shared convolutional features.
- **Better proposal quality** with higher recall and precision.
- **Scalability and flexibility** to handle objects of different sizes and aspect ratios.
- **Reduced redundancy** through non-maximum suppression.
- **Improved detection performance** with better accuracy and speed.
- **Simplified architecture** for easier implementation and maintenance.



For the center of the window or the anchor coordinates, to have different perspectives, different ratios of objects with different scales, and similar issues, we consider different anchor boxes in different scales and shapes.



Now let's talk about the relationship between the RPN and the given image. As we mentioned, the RPN is responsible for obtaining region proposals, but it does this using sliding windows. These windows move with a 3x3 kernel. The center of each window is called an anchor. For each anchor, k anchor boxes must be obtained. So how does this happen?

Using IoU, we can choose the correct anchor box from each scale, shape, and size. Finally, by selecting k anchor boxes, we move to the next stage. The relationship between the new and previous coordinates, and these deltas (the difference in the displacement of the new anchor box) in different scales and sizes for the image of a dog, is shown in the form of a diagram.

Now it's time to discuss the RPN loss term after we have developed an end-to-end network, allowing us to have a unified optimization and define a single loss function for the entire network.

The overall loss of the model consists of two parts:

1. The loss of the Fast RCNN network, which has been thoroughly explained earlier.
2. The loss of the RPN network, which itself comprises two components.

It can be broken down into the following components:

1. **RPN Loss:**

- **RPN Classification Loss (L_{rpn_cls}):** This is a binary cross-entropy loss for determining whether an anchor is an object or not (foreground or background).
- **RPN Regression Loss (L_{rpn_reg}):** This is a smooth L1 loss for refining the coordinates of the anchor boxes to better fit the ground truth boxes.

The RPN loss is given by:

$$L_{rpn}(p, p^*, t, t^*) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \gamma \frac{1}{N_{reg}} \sum_i [p_i^* \geq 1] L_{reg}(t_i, t_i^*)$$

where p_i is the predicted probability of anchor i being an object, p_i^* is the ground truth label (1 for object, 0 for background), t_i are the predicted bounding box coordinates for anchor i , t_i^* are the ground truth coordinates, N_{cls} is the number of anchors, and N_{reg} is the number of anchors with objects.

2. **Fast R-CNN Loss:** Once the proposals are generated by the RPN, they are fed into the Fast R-CNN detector. The Fast R-CNN loss is applied to these proposals, consisting of:
 - **Fast R-CNN Classification Loss (L_{cls}):** Categorical cross-entropy loss for classifying the proposed regions.
 - **Fast R-CNN Regression Loss (L_{reg}):** Smooth L1 loss for refining the bounding box coordinates of the proposed regions.

The total loss for Faster R-CNN combines the RPN and Fast R-CNN losses:

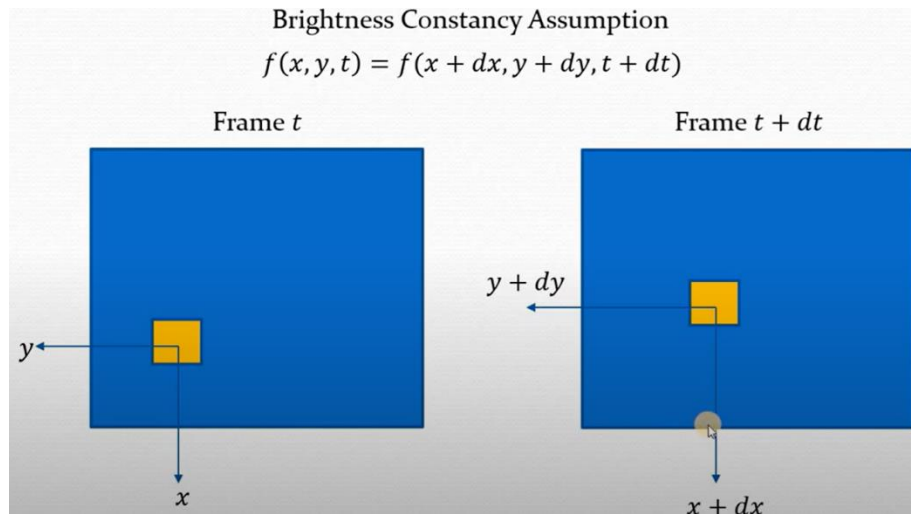
$$L = L_{cls} + L_{fast-rcnn}$$

Q2 – Part A –

For this question, we also assume that there is a relative familiarity with the solution and general method of these two algorithms.

To examine the ideas of each algorithm, we first need to state the assumptions of each one. Then, we will review the equations derived from these assumptions and their simplifications.

One of the constant assumptions that exists for both algorithms and needs to be explained only once is the assumption of brightness constancy. This assumption states that the brightness of pixels from frame t to frame $t+dt$, and the displacement in the y direction by $y+dy$ and in the x direction by $x+dx$, will not change and remains constant.



Now we will simplify the brightness equation obtained using the Taylor expansion. Finally, we divide the entire equation by dt to obtain the velocities in each direction. Velocity here refers to the rate of displacement changes over time (from frame to frame). We need to find the two unknowns, u and v , to determine the optical flow vectors for each pixel.

$$f(x, y, t) = f(x + dx, y + dy, t + dt)$$

Using Taylor Series

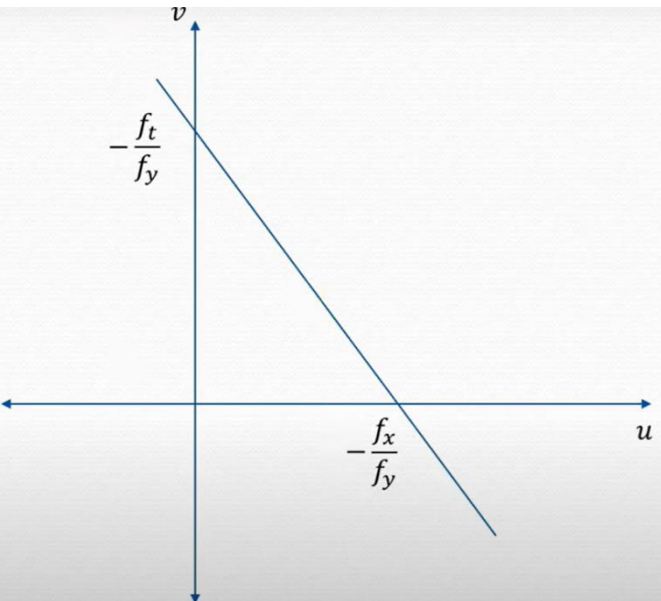
$$f(x, y, t) = f(x, y, t) + \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial y} dy + \frac{\partial f}{\partial t} dt + \dots \text{higher order terms}$$
$$0 = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial y} dy + \frac{\partial f}{\partial t} dt$$
$$f_x \frac{dx}{dt} + f_y \frac{dy}{dt} + f_t \frac{dt}{dt} = 0$$

So, for each pixel, we can obtain the changes in the x and y directions. However, the problem is that we need these displacements (velocities) in both directions for each frame, but we only have one equation with two unknowns. Thus, we have an underdetermined system, which can generally be represented as follows

$$f_x \frac{dx}{dt} + f_y \frac{dy}{dt} + f_t \frac{dt}{dt} = 0$$

$$f_x u + f_y v + f_t = 0$$

$$f_y v = -f_x u - f_t$$

$$v = -\frac{f_x}{f_y} u - \frac{f_t}{f_y}$$


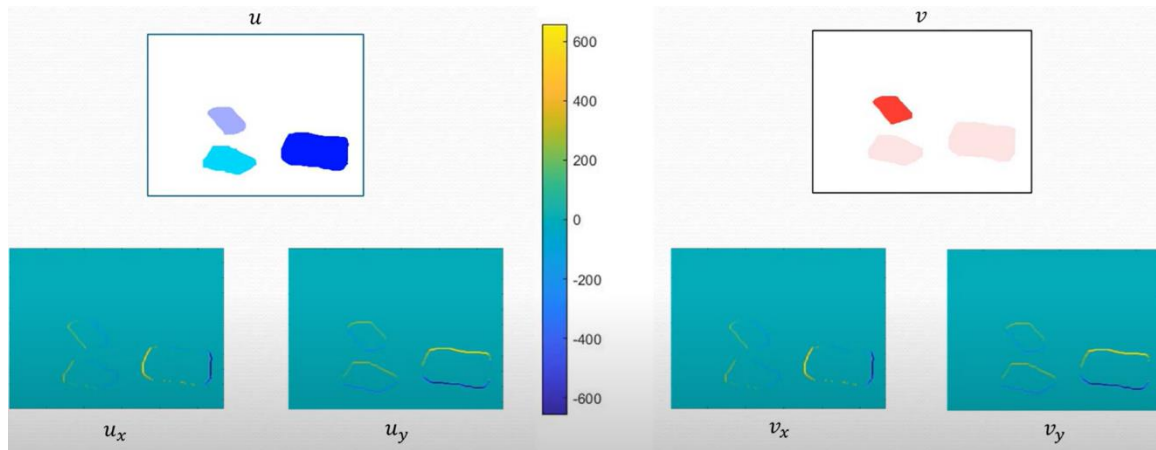
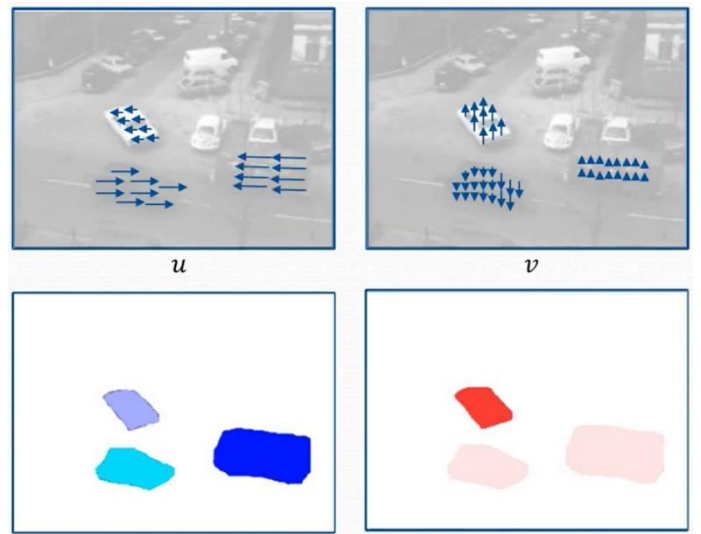
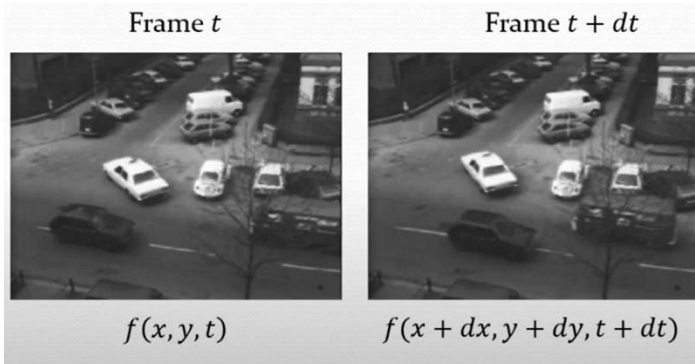
Now, the main difference between these two algorithms actually lies in the two assumptions that we will examine next.

The Horn and Schunck algorithm assumes that the optical flow field is smooth over the entire image. This leads to the smoothness constraint, which states that the flow vectors (velocities u and v) change smoothly across the image.

Now, if we assume that we obtain the displacement vectors in the x and y directions for two frames, meaning we calculate u and v , and based on that we plot heatmaps, we can derive the derivatives of these heatmaps with respect to x and y .

By doing so, we can observe that in most regions of the image, the value of this derivative will be zero, indicating no changes. However, in small sections, changes can occur in both directions. Specifically, the derivative of the heatmap corresponding to u in the x direction is positive, and in the y direction, it is negative. Similarly, for v , the same pattern holds.

The assumption in the Horn and Schunck algorithm is that the displacement changes from frame to frame are very smooth across the entire image. As a result, these changes are not rapid and are very small, and as mentioned, in most areas, they are zero. Therefore, in addition to the term enforcing the constancy of brightness intensity from frame to frame, another term is added to ensure smoothness.



After this point, as evident from the image, most of the image's derivatives and displacements are zero. However, in some areas where there is no background, there is displacement. Additionally, because we have negative values, we add the squares of these values. Now, the main task is to minimize our loss function, which consists of these two terms. The reason for the integral here is also because in 1980, when this algorithm was invented, images were considered continuous.

$$E = \iint_{x,y} (f_x u + f_y v + f_t)^2 + \lambda(u_x^2 + u_y^2 + v_x^2 + v_y^2) dx dy$$

Indeed, ultimately, we can simplify it using the Euler-Lagrange equation.

$$E = \iint_{x,y} (f_x u + f_y v + f_t)^2 + \lambda(u_x^2 + u_y^2 + v_x^2 + v_y^2) dx dy$$

Euler-Lagrange equations

$$E_u - \frac{du_x}{dx} - \frac{du_y}{dy} = 0 \qquad E_v - \frac{dv_x}{dx} - \frac{dv_y}{dy} = 0$$

$$2(f_x u + f_y v + f_t)f_x - 2\lambda(u_{xx} + u_{yy}) = 0 \qquad 2(f_x u + f_y v + f_t)f_y - 2\lambda(v_{xx} + v_{yy}) = 0$$

$$(f_x u + f_y v + f_t)f_x - \lambda(\Delta^2 u) = 0 \qquad (f_x u + f_y v + f_t)f_y - \lambda(\Delta^2 v) = 0$$

For calculating spatial and temporal changes, you can use the following masks:

$$\begin{matrix} \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} & \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} & \begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix} \\ \text{mask1} & \text{mask2} & \text{mask3} & \text{mask4} \end{matrix}$$

- Apply mask1 to both images, add the responses to get f_x ;
- Apply mask2 to both images, add the responses to get f_y ;
- Apply mask4 to image1, apply mask3 to image2, add the responses to get f_t .

Now, using the Laplacian filter, we can simplify the equation to the following form.

$$\begin{bmatrix} \frac{1}{12} & \frac{1}{6} & \frac{1}{12} \\ \frac{1}{6} & -1 & \frac{1}{6} \\ \frac{1}{12} & \frac{1}{6} & \frac{1}{12} \end{bmatrix} \quad \begin{aligned} (\Delta^2 u) &= (u_{avg} - u) \\ (\Delta^2 v) &= (v_{avg} - v) \end{aligned}$$

$$(f_x u + f_y v + f_t) f_x - \lambda(u_{avg} - u) = 0 \quad (f_x u + f_y v + f_t) f_y - \lambda(v_{avg} - v) = 0$$

So, after successive simplifications and substitutions, we arrive at the following equations. Repeat calculating the following values until convergence.

$$\begin{aligned} u &= u_{avg} - f_x \frac{P}{D} & v &= v_{avg} - f_y \frac{P}{D} \\ P &= (f_x u_{avg} + f_y v_{avg} + f_t) & D &= (f_y^2 + f_x^2 + \lambda) \end{aligned}$$

Now, if we want to have the general Horn-Schunck algorithm, it will be as follows (it's an iterative algorithm):

- Initialize u and v as 0 matrices $P = (f_x u_{avg} + f_y v_{avg} + f_t)$
- Compute f_x, f_y and f_t images using the masks $D = (f_y^2 + f_x^2 + \lambda)$
- Update $u = u_{avg} - f_x \frac{P}{D}, v = v_{avg} - f_y \frac{P}{D}$
- Repeat the above steps until some error measure is satisfied

For the Lucas-Kanade algorithm, in addition to assuming brightness constancy, another assumption is made. It assumes that for a pixel within a small local neighborhood, which is proportional to the size of the kernel and window, all displacement changes in two directions will be the same. As a result, we will have 9 equations:

$$eqs^n \left\{ \begin{array}{l} f_x(x, y)u + f_y(x, y)v = -f_t(x, y) \longleftarrow f_x u + f_y v + f_t = 0 \\ f_x(x + 1, y)u + f_y(x + 1, y)v = -f_t(x + 1, y) \\ f_x(x - 1, y)u + f_y(x - 1, y)v = -f_t(x - 1, y) \\ f_x(x, y + 1)u + f_y(x, y + 1)v = -f_t(x, y + 1) \\ \vdots \\ f_x(x + 1, y + 1)u + f_y(x + 1, y + 1)v = -f_t(x + 1, y + 1) \end{array} \right.$$

$$f_{xi}u + f_{yi}v = -f_{ti} \quad \longrightarrow \quad [f_{xi} \quad f_{yi}] \begin{bmatrix} u \\ v \end{bmatrix} = -f_{ti}$$

Now, if we want to write it in a general matrix form, considering a 3x3 window for optical flow estimation for all 9 pixels, it would be as follows:

$$\begin{bmatrix} I_x(x_1, y_1) & I_y(x_1, y_1) \\ I_x(x_2, y_2) & I_y(x_2, y_2) \\ \vdots & \vdots \\ I_x(x_9, y_9) & I_y(x_9, y_9) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -I_t(x_1, y_1) \\ -I_t(x_2, y_2) \\ \vdots \\ -I_t(x_9, y_9) \end{bmatrix}$$

Exactly, in this formulation, we have 9 equations and 2 unknowns (flow velocities u and v for each pixel within the 3x3 window). Therefore, to solve this overdetermined system, we typically use the least squares method. This method finds the flow velocities that minimize the sum of the squared differences between the left-hand side and right-hand side of the equations.

To minimize this error, we can take the derivatives with respect to u and v and set them equal to zero.

$$9 \text{ eqs}^n \longleftarrow f_{xi}u + f_{yi}v = -f_{ti}$$

$$C = \sum_i (f_{xi}u + f_{yi}v + f_{ti})^2$$

Minimizing the cost

$$\begin{aligned}
\sum_i 2 \times (f_{xi}u + f_{yi}v + f_{ti})f_{xi} &= 0 & \sum_i 2 \times (f_{xi}u + f_{yi}v + f_{ti})f_{yi} &= 0 \\
\sum_i (f_{xi}^2u + f_{yi}f_{xi}v + f_{ti}f_{xi}) &= 0 & \sum_i (f_{xi}f_{yi}u + f_{yi}^2v + f_{ti}f_{yi}) &= 0 \\
\sum_i f_{xi}^2u + \sum_i f_{yi}f_{xi}v &= \sum_i -f_{ti}f_{xi} & \sum_i f_{xi}f_{yi}u + \sum_i f_{yi}^2v &= \sum_i -f_{ti}f_{yi}
\end{aligned}$$

Now, applying the least squares method to these equations, we arrive at the final matrix, which closely resembles the Hessian matrix.

$$\begin{aligned}
&\sum_i f_{xi}^2u + \sum_i f_{yi}f_{xi}v = \sum_i -f_{ti}f_{xi} \\
&\sum_i f_{xi}f_{yi}u + \sum_i f_{yi}^2v = \sum_i -f_{ti}f_{yi} \\
&\begin{bmatrix} \sum_i f_{xi}^2 & \sum_i f_{yi}f_{xi} \\ \sum_i f_{xi}f_{yi} & \sum_i f_{yi}^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\sum_i f_{ti}f_{xi} \\ -\sum_i f_{ti}f_{yi} \end{bmatrix} \\
&\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{xi}^2 & \sum_i f_{yi}f_{xi} \\ \sum_i f_{xi}f_{yi} & \sum_i f_{yi}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{ti}f_{xi} \\ -\sum_i f_{ti}f_{yi} \end{bmatrix}
\end{aligned}$$

Now, if we want to write the Lucas-Kanade algorithm, it would be as follows:

- Compute f_x, f_y and f_t images using the masks
- For each pixel
- If the determinant of A is not zero,
- then calculate u and v for that pixel using least squares.

Lucas-Kanade Algorithm

1. Local Method:

- The Lucas-Kanade algorithm is a local method. It computes the optical flow in a small neighborhood around each pixel.

2. Assumptions:

- It assumes that the flow is essentially constant in the neighborhood of the pixel.

3. Computational Complexity:

- It involves solving a small system of linear equations for each pixel, which makes it computationally efficient for small windows.

4. Sparsity:

- Typically used in sparse optical flow applications where flow is calculated only at specific points or features (e.g., corners).

5. Window Size:

- The accuracy and reliability depend on the size of the window used. A larger window may improve accuracy but reduce the ability to capture small movements.

Horn-Schunck Algorithm

1. Global Method:

- The Horn-Schunck algorithm is a global method. It computes optical flow by considering the entire image at once.

2. Assumptions:

- It assumes that the optical flow field is smooth over the entire image.

3. Computational Complexity:

- It involves solving a large system of equations over the entire image, which can be computationally intensive.

4. Implementation:

- It includes a regularization term to enforce smoothness, which adds complexity to the solution.

5. Regularization Parameter:

- The accuracy and smoothness of the flow depend on the choice of the regularization parameter.

Noise Robustness

- **Lucas-Kanade:**
 - More robust to noise due to its local approach. The assumption of constant flow within a small neighborhood makes it less sensitive to noise in specific regions.
 - The use of smaller neighborhoods can make it less prone to large errors caused by global noise, but very small windows might be more affected by local noise.
- **Horn-Schunck:**
 - Less robust to noise because it considers the entire image, and noise can affect the smoothness term, leading to incorrect flow estimates.
 - The global smoothness constraint might spread the noise effect across the image, making the method more sensitive to noise.

Lucas-Kanade Assumptions

1. **Constant Flow in Local Neighborhood:**
 - Assumes the optical flow does not change significantly within a small window around each pixel.
2. **Brightness Constancy:**
 - Assumes that the intensity of the pixels remains constant between consecutive frames.

Horn-Schunck Assumptions

1. **Smoothness Constraint:**
 - Assumes that the optical flow field is smooth over the entire image.
2. **Brightness Constancy:**
 - Similar to Lucas-Kanade, it assumes that the intensity of the pixels remains constant between consecutive frames.

Q2 – Part B –

We provided a comprehensive explanation of the Horn-Schunck algorithm along with its error function, which consists of two terms. We mentioned the purpose of each term: the first term enforces intensity constancy across frames for pixels, and the second term relates to the smoothness constraint. Now, along with an example, let's elaborate on the effects of the regularization parameter (usually denoted by λ) associated with the second term.

The regularization parameter λ controls the relative importance of these two terms. A higher value of λ emphasizes smoothness, leading to smoother flow fields with less noise and fewer discontinuities. Conversely, a lower value of λ gives more weight to the data term, which can result in more accurate but potentially noisier flow estimates.

Let's illustrate the effects of varying the regularization parameter λ with a numeric example using synthetic data. For simplicity, let's consider a small image patch with known intensity values and compute the optical flow using the Horn-Schunck algorithm.

Suppose we have a small 3x3 image patch with the following intensity values for two consecutive frames:

$$\begin{aligned} \text{frame 1} &= \begin{bmatrix} 10 & 20 & 30 \\ 15 & 25 & 35 \\ 5 & 10 & 15 \end{bmatrix} \\ \text{frame 2} &= \begin{bmatrix} 12 & 22 & 32 \\ 17 & 27 & 37 \\ 7 & 12 & 17 \end{bmatrix} \end{aligned}$$

Let's calculate the optical flow for this patch using the Horn-Schunck algorithm with different values of λ .

Let's assume we have computed the gradients and initial flow field as follows:

$$I_x = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}, \quad I_y = \begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \end{bmatrix}, \quad I_t = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

Let's consider two extreme cases:

1. **High λ :** Assume $\lambda = 10$

Using the Horn-Schunck algorithm with a high λ , the smoothness term dominates, and the flow field is regularized strongly towards zero. The resulting flow might look like:

$$u = v = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

2. **Low λ :** Assume $\lambda = 0.1$

With a low λ , the data term is emphasized, leading to flow estimates that closely match the observed intensity changes. The resulting flow might look like:

$$u \approx v \approx \begin{bmatrix} 1.82 & 1.82 & 1.82 \\ 1.82 & 1.82 & 1.82 \\ 1.82 & 1.82 & 1.82 \end{bmatrix}$$

In this simplified example, a high λ results in a highly regularized flow field with small or zero flow values, while a low λ leads to flow estimates that closely follow the observed intensity changes.

Now, we can write in a concise and specific manner:

High λ (Strong Smoothness Prior):

- **Scenario:**
 - λ is set to a high value.
 - The algorithm heavily penalizes large spatial gradients in the flow field.
- **Effect:**
 - The resulting optical flow field appears very smooth and stable.
 - Noise and small variations in the image sequence are suppressed.
 - However, the flow field might not accurately capture rapid changes in the vehicle's motion, especially when it makes sudden turns or accelerations.
 - The flow might be overly smoothed, leading to inaccuracies in regions with sharp intensity changes, such as edges of the vehicle or its surroundings.

Low λ (Weak Smoothness Prior):

- **Scenario:**

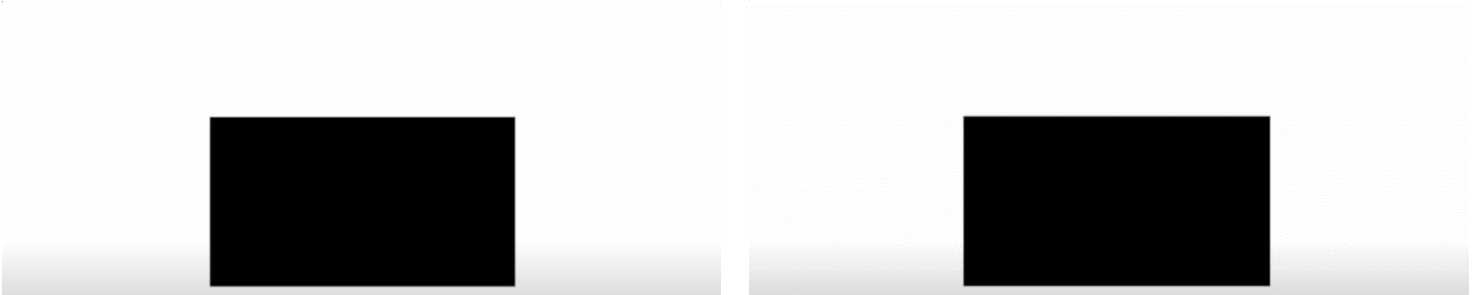
- λ is set to a low value.
- The algorithm gives less weight to the smoothness term, prioritizing data fidelity.

- **Effect:**

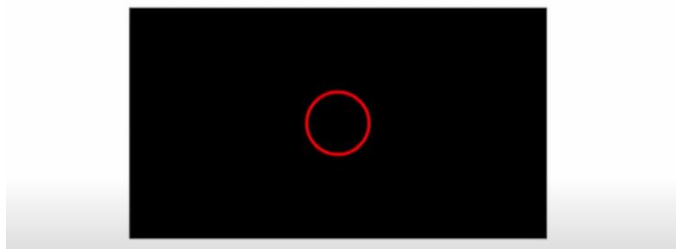
- The resulting optical flow field preserves finer details and captures rapid changes in motion more accurately.
- The flow might exhibit more noise and fluctuations, especially in regions with low contrast or texture.
- It can accurately capture sharp changes in motion, such as sudden accelerations or decelerations of the vehicle.
- However, the flow might also be less stable and coherent overall, with more discontinuities and artifacts.

Q2 – Part C –

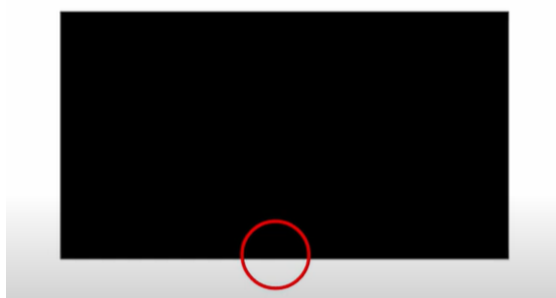
the aperture problem in computer vision we encountered this in problems like motion estimation now consider this black rectangle on a white background and it's moving back and forth in the horizontal direction .



if we consider only a small region in the middle of this rectangle we look at the pixel intensities to try to find the motion in that case we don't see any motion because it's completely flat in the center so this point is actually not good for estimating the motion of the rectangle



how about the horizontal edge well the motion is horizontal and the edge is horizontal so this point too doesn't see any change in pixel intensities so this too is very bad



on the other hand this is a vertical edge and the vertical edge clearly sees a horizontal motion so this is great but again we know the vertical edge will not see a vertical motion and horizontal edges will not see a horizontal motion



so the best kinds of features that help us see all kinds of motion both horizontal and vertical are these corner features and that's why while doing motion estimation we first try to find these corner because other parts of the image they suffer from this problem called aperture problem



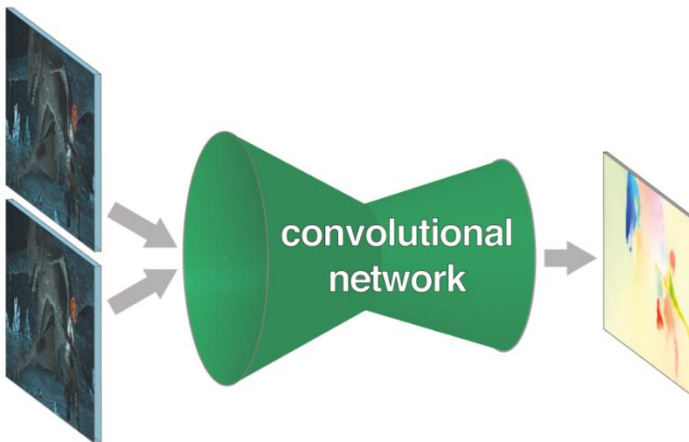
we are looking at the image through a small window in motion estimation typically for example in lucas-kanade a optical flow we first find features and around that region we try to estimate the motion we are not looking at the entire image even if you were looking at the entire image you could think about the entire image as an aperture you do not have access to the whole world you're looking at the world through this aperture and so you can estimate only certain kinds of motion if you're not looking at corners

Q3 – Part A & B –

For this question, I prefer to answer both the first and second parts at once and within one concept.

Since convolutional neural network (CNN) models were introduced, they have been used in almost all vision tasks, including image and video processing. Many tasks such as classification, object detection, and more have been addressed using these networks. Another application where convolutional networks can be utilized is obtaining optical flow. The initial models that proposed an architecture and model for this task using convolutional networks were presented in the paper "FlowNet: Learning Optical Flow with Convolutional Networks," which itself introduced two models with different architectures.

To obtain optical flow, we need two sequential images from a video (frame by frame) to capture the displacement changes in directions (velocity). Therefore, a general schematic can be envisioned as follows: two consecutive frames of an event are fed into a convolutional network, and the optical flow is obtained as the output.

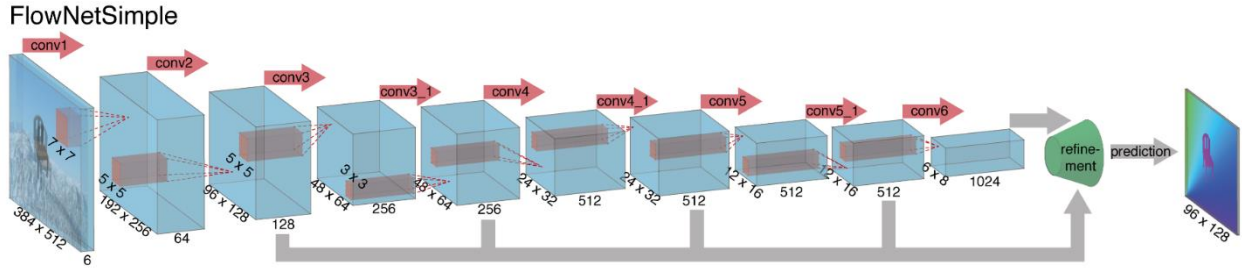


In general, for all end-to-end neural networks, it is necessary to train the network with a set of data. For this case, we have a set of image pairs and ground truth, and we train the model output, which is the x-y flow fields, using the Euclidean distance error. Now, before discussing the components of both FlowNet models, we need to address a topic mentioned in the paper:

Pooling in CNNs is necessary to make network training computationally feasible and, more fundamentally, to allow aggregation of information over large areas of the input images. But pooling results in reduced resolution, so to provide dense per-pixel predictions, we need to refine the coarse pooled representation. To this end, our networks contain an expanding part which intelligently refines the flow to high resolution.

Thus, in general, these models use an encoder part to obtain the feature maps and feature representations of the input, and to achieve per-pixel mapping, we need another part to expand and increase the resolution, which is the decoder's responsibility. We will become familiar with the details of each. Therefore, both FlowNet models contain a CNN-based autoencoder.

The first approach mentioned in the paper is to provide the image pairs in a stacked form as input to the model, so that the network learns how to process the image pair in a way that extracts the motion information from it. This fully convolutional model is named FlowNetSimple.

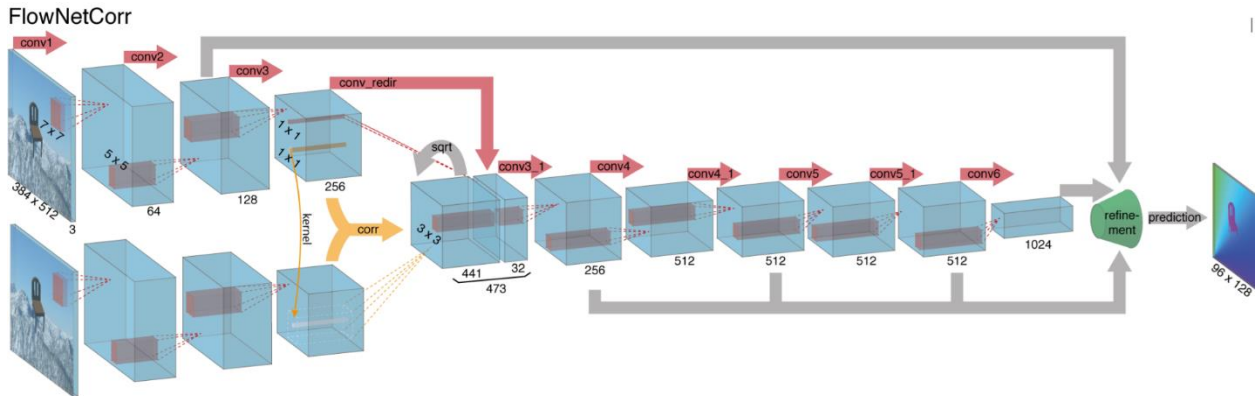


Of course, both models have similar components, including the encoder, decoder, and refinement parts. For convenience, we will explain these components only once, so that we can separately examine the different parts of each model if needed.

The second approach, which is more complex, involves designing a separate CNN-based encoder for each pair of consecutive images we have. Each image is processed separately by its own network, resulting in each network providing a meaningful representation of one image. These representations are then combined in a way that forms a higher-level representation, which is more suitable for this task.

Now, to obtain matching features between these two feature maps and find correspondences, a layer called the correlation layer is implemented. This layer performs multiplicative patch comparisons between two feature maps. Given two multi-channel feature maps $f_1, f_2 : \mathbb{R}^2 \rightarrow \mathbb{R}^c$ where w, h , and c are their width, height, and number of channels, the correlation layer allows the network to compare each patch from f_1 with each patch from f_2 .

The second architecture you see in the image is called FlowNetCorr.



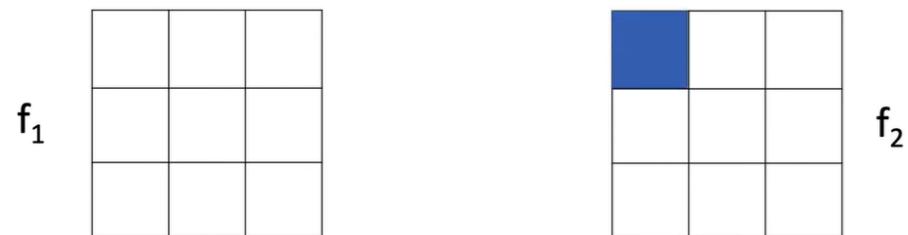
Now, for a better understanding of the correlation layer's functionality, let's provide an example correlation operation is defined in this equation this is basically a convolution operation between two feature maps.

$$c(\mathbf{x}_1, \mathbf{x}_2) = \sum_{\mathbf{o} \in [-k, k] \times [-k, k]} \langle \mathbf{f}_1(\mathbf{x}_1 + \mathbf{o}), \mathbf{f}_2(\mathbf{x}_2 + \mathbf{o}) \rangle$$

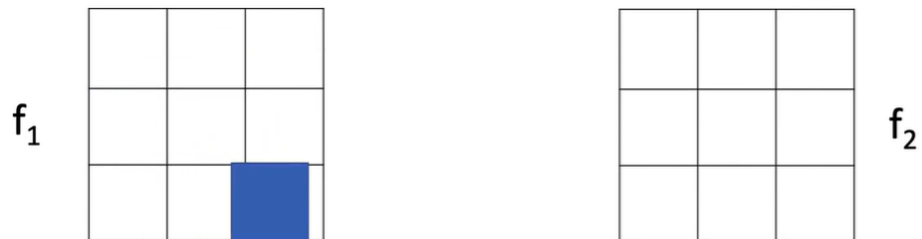
assume these are two three by three feature maps



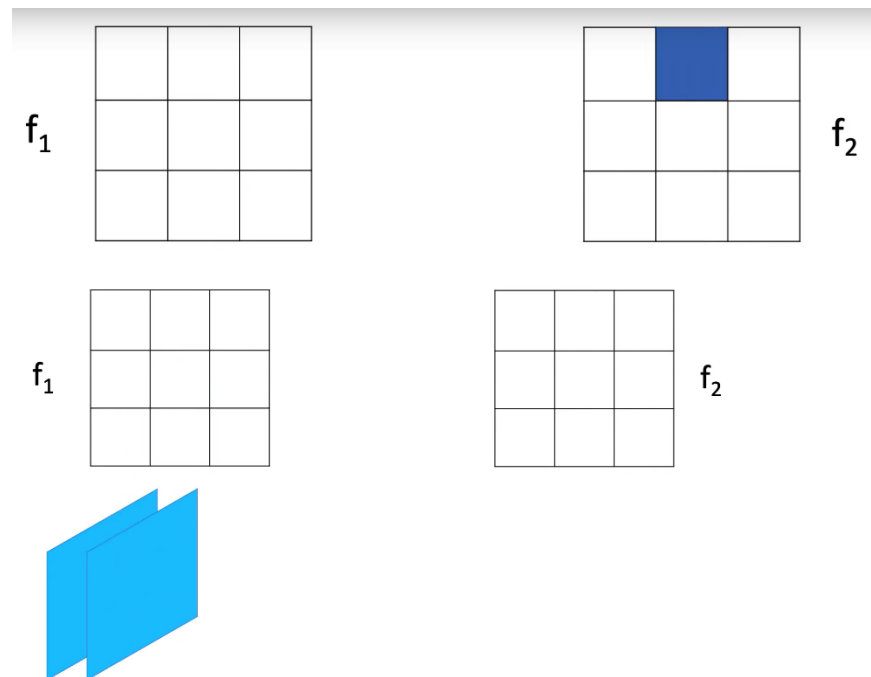
let kernel size be one for convolution operation and strides for both feature maps are one



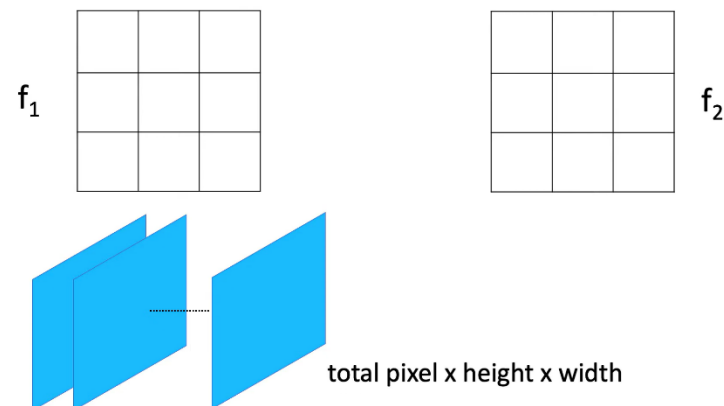
then the correlation operation can be stated like this one pixel from f_2 performs convolution operation with each of the pixels from f_1 and produces a single channel 3x3 feature map



second pixel from f_2 does the same operation and produces another 3x3 feature map

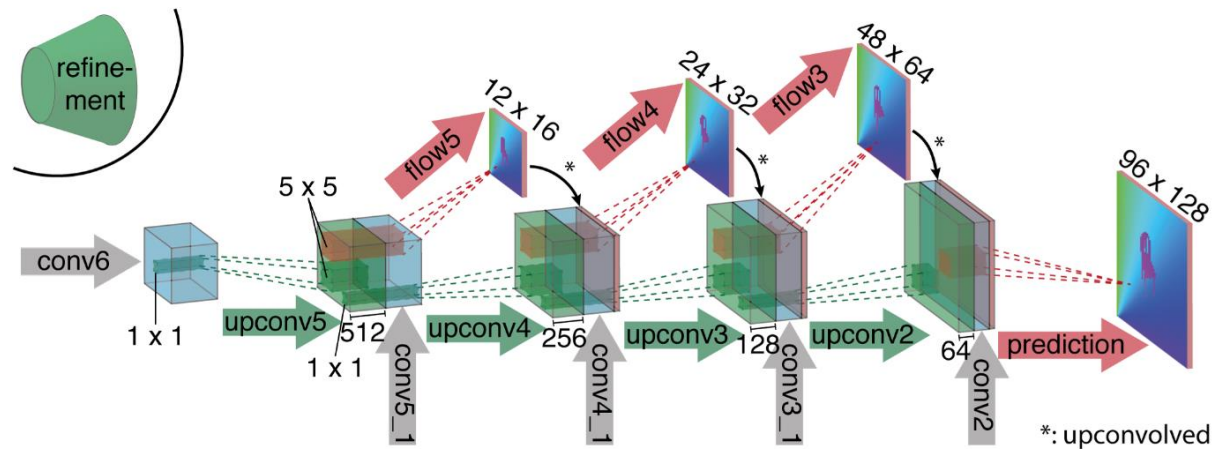


at the end when all of the pixels from f_2 have performed convolution with all of the pixels in f_1 then the feature map with matching information is produced the size of this feature map is total pixel by height by width and is passed to the next layers for motion prediction the features from contracting part are also skip connected to the refinement layers.



The main ingredient of the expanding part are 'upconvolutional' layers, consisting of unpooling and a convolution. Such layers have been used previously. To perform the refinement, applied the 'upconvolution' to feature maps, and concatenate it with corresponding feature maps from the 'contractive' part of the network (skip connections) and an upsampled coarser flow prediction (if available). This way we preserve both the high-level information passed from coarser feature maps and fine local information provided in lower layer feature maps. Each step increases the

resolution twice. repeat this 4 times, resulting in a predicted flow for which the resolution is still 4 times smaller than the input.



In the end, I provided all my writings to ChatGPT, along with additional information, to summarize these contents and present the challenges of the FlowNetCorr network.

FlowNetS

Architecture:

FlowNetS (Simple) employs a straightforward encoder-decoder architecture. It consists of:

1. **Encoder:** A series of convolutional layers that progressively downsample the input images to extract hierarchical features.
2. **Decoder:** A series of convolutional and upsampling layers that reconstruct the optical flow from the encoded features.
3. **Skip Connections:** Connections between corresponding encoder and decoder layers to preserve spatial information.

Advantages:

- **Simplicity:** The architecture is easy to understand and implement.
- **Feature Extraction:** The hierarchical feature extraction helps in capturing both fine and coarse details of the motion.
- **Efficiency:** FlowNetS can be trained relatively quickly due to its straightforward structure.

FlowNetCorr Architecture

1. Input Layer:

- **Two Input Images:** The model takes two consecutive video frames or images as input.

2. Feature Extraction (Encoders):

- **Separate Encoders:** Each image is processed through a series of convolutional layers independently, producing feature maps for each image.
- **Shared Weights:** Initially, the encoders can share weights to reduce the number of parameters and improve generalization.

3. Correlation Layer:

- **Cost Volume Calculation:** This layer computes the correlation between feature patches from the two encoded images. It essentially measures the similarity between patches, creating a cost volume that represents potential motion vectors.

4. Decoder:

- **Flow Estimation:** The decoder takes the cost volume as input and processes it through several convolutional and upsampling layers to estimate the dense optical flow.
- **Refinement:** Additional layers refine the flow estimates to improve accuracy.

5. Skip Connections:

- **Feature Fusion:** Similar to FlowNetS, skip connections are used to combine high-resolution features from the encoder with the decoder, aiding in preserving spatial details.

Challenges

1. Computational Complexity:

- **Cost Volume Calculation:** The correlation operation is computationally intensive, especially for high-resolution images, leading to increased memory usage and slower inference times.

2. Large Displacements:

- **Handling Large Motions:** Despite the cost volume, accurately estimating very large displacements remains challenging, as the correlation layer can only capture limited ranges of motion without multi-scale approaches.

3. Training Data:

- **Synthetic to Real Gap:** FlowNetCorr, like other deep learning models, is often trained on synthetic datasets (e.g., FlyingChairs) that may not perfectly represent real-world scenarios, leading to domain adaptation issues.

4. Occlusions and Ambiguities:

- **Occlusions:** Areas occluded in one image but visible in another pose significant challenges, as the model must infer motion without direct visual correspondence.
- **Textureless Regions:** Homogeneous or low-texture regions can confuse the correlation layer, making it difficult to determine the correct motion vectors.

FlowNetCorr represents a significant advancement in learning-based optical flow estimation by incorporating explicit feature matching. However, addressing the computational demands and improving robustness to various real-world challenges remain active areas of research.

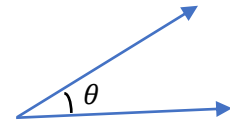
Q4 – To better understand self-attention in the Vision Transformer (ViT) model, it is advisable to first grasp the concept from the original Transformer paper.

One method for calculating the similarity between two vectors and matrices is cosine similarity.

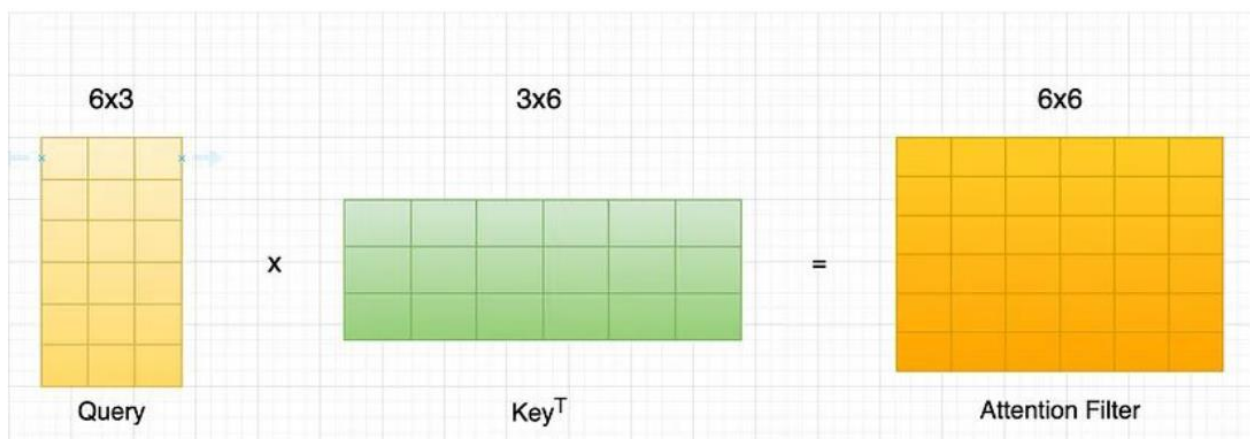
$$\text{sim}(A, B) = \cos(\theta) = \frac{A \cdot B}{|A||B|}$$

$$\text{sim}(A, B) = \frac{A \cdot B^T}{\text{scale}}$$

$$\text{in transformer} \rightarrow \text{sim}(Q, K) = \frac{Q \cdot k^T}{\text{scale}}$$



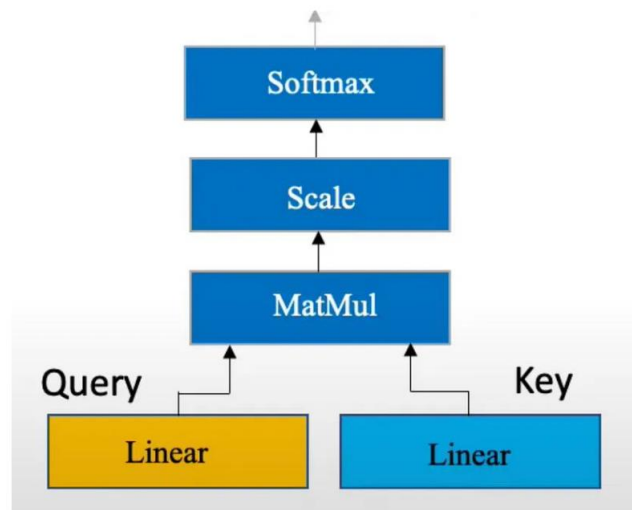
Suppose you want to search for something in a dataset. The term you are searching for is the query. What is displayed are keys, and the values inside those keys are our values. The closer the displayed key is to the query, the more relevant the content of that key is to us.



Hi	89	12	32	45	73	34
,	20	11	15	21	29	24
How	33	25	91	36	55	45
are	52	68	12	13	40	27
you	28	27	54	41	92	12
?	39	30	73	64	12	74
Hi	,	How	are	you	?	

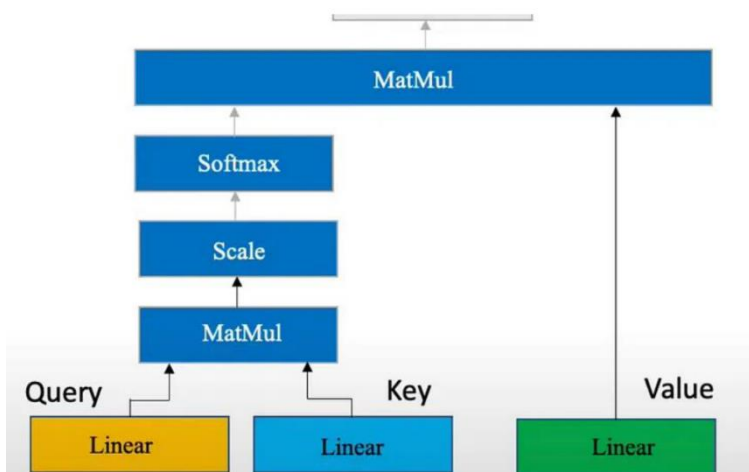
Here, we calculate the value and context of the key and query to determine the context for each key element in the query and how close and important it is. After applying softmax, which outputs a number between zero and one, we obtain our scores.

Now, we can say that those key elements with more important contexts have more significant values. The context calculation is performed element-wise using the cosine similarity formula.



In the attention function, after applying the links on the queries and keys, we pass it to a linear layer. This layer has 64 units. Then, we apply the matmul function to calculate the context of each key element in the query.

And we calculate that context using cosine similarity to determine the closeness of the key element to the query and to obtain its overall context within the query. After applying scaling by the square root of d_k the dimension of the keys (which is the square root of 64, equal to 8), we pass it to softmax to get scores between zero and one. This helps us determine which key elements are more important. As a result, by multiplying it with the value matrix, we focus more on the content whose key and context are closer to the query and are more important to us. This effectively computes the important values.



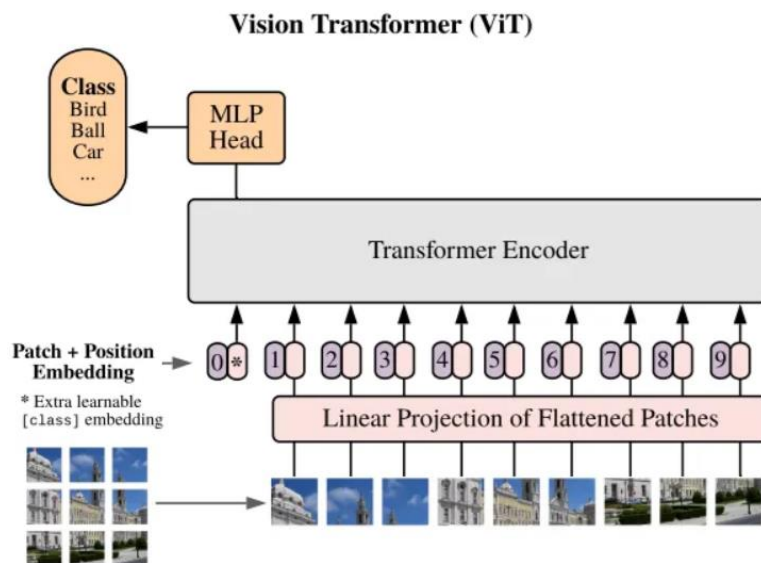
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Now, after understanding attention and self-attention, we can more easily grasp how this concept works in Vision Transformers (ViT), especially since it is used in exactly the same way.

The input image is divided into several square patches of the same size. Each patch is then linearly transformed into a vector by a learnable linear projection layer.

Ultimately, this process results in a sequential output of patch embeddings, which are considered as input tokens for the subsequent layers.

For positional encoding in this model, you can indeed use two-dimensional embedding algorithms or a learnable filter. The rest of the functions and the model's behavior remain the same as described earlier for the self-attention concept in the original Transformer paper.



Vision Transformers vs. CNN?

ViT differs from Convolutional Neural Networks (CNNs) in several key aspects:

- **Input Representation:** While CNNs process raw pixel values directly, ViT divides the input image into patches and transforms them into tokens.
- **Processing Mechanism:** CNNs use convolutional and pooling layers to hierarchically capture features at different spatial scales. ViT employs self-attention mechanisms to consider relationships among all patches.
- **Global Context:** ViT inherently captures global context through self-attention, which helps in recognizing relationships between distant patches. CNNs rely on pooling layers for coarse global information.
- **Data Efficiency:** CNNs often require large amounts of labeled data for training, whereas ViT can benefit from pre-training on large datasets and then fine-tuning on specific tasks.

Limitations of Vision Transformers

- **Large Datasets:** Training Vision Transformers effectively often requires large datasets, which might not be available for all domains.
- **Computational Demands:** Training ViT can be computationally intensive due to the self-attention mechanisms.
- **Spatial Information:** ViT's sequential processing might not capture fine-grained spatial patterns as effectively as CNNs for tasks like segmentation.

To compute the number of learnable parameters and the number of patches created from the input image for the Vision Transformer (ViT) with the given attributes, we need to follow these steps:

Number of Patches

1. **Input Size:** 224×224
2. **Patch Size:** 16×16

The number of patches N is given by:

$$N = (\text{Input Height} / \text{Patch Height}) \times (\text{Input Width} / \text{Patch Width})$$

Given:

- Input Height = 224
- Input Width = 224
- Patch Height = 16
- Patch Width = 16

$$N = (224/16) \times (224/16) = 14 \times 14 = 196$$

So, the number of patches is **196**.

Number of Learnable Parameters

1. **Embedding Dimension Size:** 512
2. **Number of Multi-Head Attentions:** 8
3. **Number of Layers:** 12

Patch Embedding Layer

Each patch is embedded into a vector of size 512. The embedding layer can be seen as a linear transformation from the patch size to the embedding dimension.

Patch Embedding Parameters= $(\text{Patch Size}^2 \times 3) \times \text{Embedding Dimension}$

Here, 3 represents the RGB channels.

Patch Embedding Parameters= $(16 \times 16 \times 3) \times 512 = 12288 \times 512 = 6,291,456$

Learnable Class Token and Position Embeddings

The class token and position embeddings will also have learnable parameters:

- Class token: one vector of size 512.
- Position embeddings: one vector of size 512 for each of the 197 positions (196 patches + 1 class token).

Class Token=512

Position Embeddings= $197 \times 512 = 100,864$

Total for class token and position embeddings:

$512 + 100,864 = 101,376$

Multi-Head Attention

In each layer, we have a multi-head attention mechanism with 8 heads. Each head has a query, key, and value matrix.

Query/Key/Value Parameters per Head = $(\text{Embedding Dimension} \times \text{Head Dimension})$

Head Dimension = $\text{Embedding Dimension} / \text{Number of Heads} = 512 / 8 = 64$

Query/Key/Value Parameters per Head = $512 \times 64 = 32,768$

Since we have 3 matrices (Query, Key, Value):

Total QKV Parameters per Head = $3 \times 32,768 = 98,304$

For 8 heads:

Total QKV Parameters for 8 Heads = $8 \times 98,304 = 786,432$

Additionally, there is an output projection matrix:

Output Projection Parameters = $512 \times 512 = 262,144$

Total Multi-Head Attention parameters:

$786,432 + 262,144 = 1,048,576$

Feed-Forward Network

Each layer contains a feed-forward network with two linear transformations. Assuming a hidden dimension of size 2048 (common in ViT):

FeedForward Network Parameters = $2 \times (\text{Embedding Dimension} \times \text{Hidden Dimension}) + (\text{Hidden Dimension} \times \text{Embedding Dimension})$

FeedForward Network Parameters= $2 \times (512 \times 2048) + (2048 \times 512) = 2,097,152 + 1,048,576 = 3,145,728$

Total Parameters for One Layer

Total Parameters per Layer=Multi-Head Attention Parameters+Feed-Forward Network Parameters

Total Parameters per Layer= $1,048,576 + 3,145,728 = 4,194,304$

Total Parameters for All Layers

Since there are 12 layers:

Total Parameters for 12 Layers= $12 \times 4,194,304 = 50,331,648$

Final Total Learnable Parameters

Combining all components : $50,331,648 + 101,376 + 6,291,456 = 56,724,480$

References

Q1 :

<https://medium.com/analytics-vidhya/introduction-to-object-detection-with-rcnn-family-models-310558ce2033>

<https://jhui.github.io/2017/03/15/Fast-R-CNN-and-Faster-R-CNN/>

<https://semazeynepbulut.medium.com/a-brief-overview-of-r-cnn-fast-r-cnn-and-faster-r-cnn-9c6843c9ffc0>

<https://towardsdatascience.com/review-faster-r-cnn-object-detection-f5685cb30202>

<https://medium.com/coinmonks/review-fast-r-cnn-object-detection-a82e172e87ba>

Q2 :

<https://www.youtube.com/watch?v=AViKP8y6n88>

<https://www.youtube.com/watch?v=6wMoHgpVUn8>

<https://www.youtube.com/watch?v=HdPnPLxjJ9c>

<https://www.youtube.com/watch?v=vVGorOxMh8w>

chat_gpt

Course slides

Q3 :

<https://www.youtube.com/watch?v=lUM8btvonYk>

<https://arxiv.org/abs/1504.06852>

Chat_gpt

Course slides

Q4 :

<https://medium.com/@hansahettiarachchi/unveiling-vision-transformers-revolutionizing-computer-vision-beyond-convolution-c410110ef061>

Prof. Dr. Farahani slides

<https://viso.ai/deep-learning/vision-transformer-vit>

chat_gpt