Nanonets

★ FEATURED OPTICAL FLOW OBJECT TRACKING DEEP LEARNING OPENCV COMPUTER VISION MACHINE LEARNING HUMAN ACTION RECOGNITION TUTORIAL PYTHON

Introduction to Motion Estimation with Optical Flow

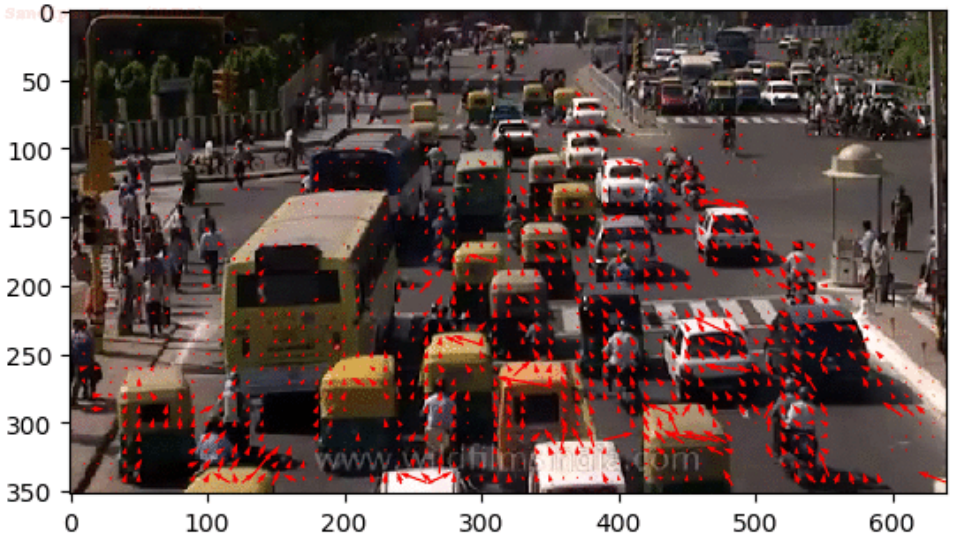
by **Chuan-en Lin** 9 months ago 12 MIN READ

Nanonets

> **In this tutorial, we dive into the fundamentals of Optical Flow, look at some of its applications and implement its two main variants (sparse and dense). We also briefly discuss more recent approaches using deep learning and promising future directions.**

Recent breakthroughs in computer vision research have allowed machines to perceive its surrounding world through techniques such as object detection for detecting instances of objects belonging to a certain class and semantic segmentation for pixel-wise classification.

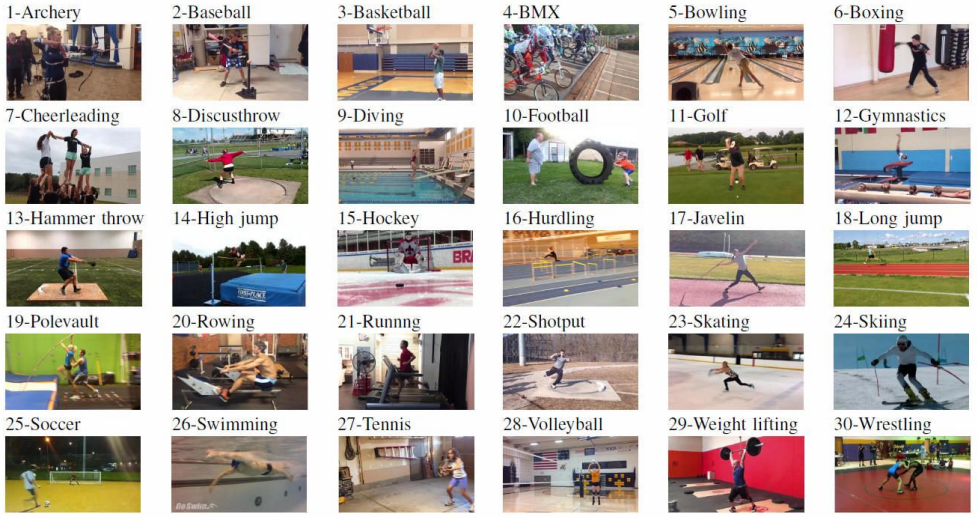
However, for processing real-time video input, most implementations of these techniques only address relationships of objects within the same frame (x, y) disregarding time information (t). In other words, they re-evaluate each frame independently, as if they are completely unrelated images, for each run. However, what if we do need the relationships between consecutive frames, for example, we want to **track the motion of vehicles across frames** to estimate its current velocity and predict its position in the next frame?



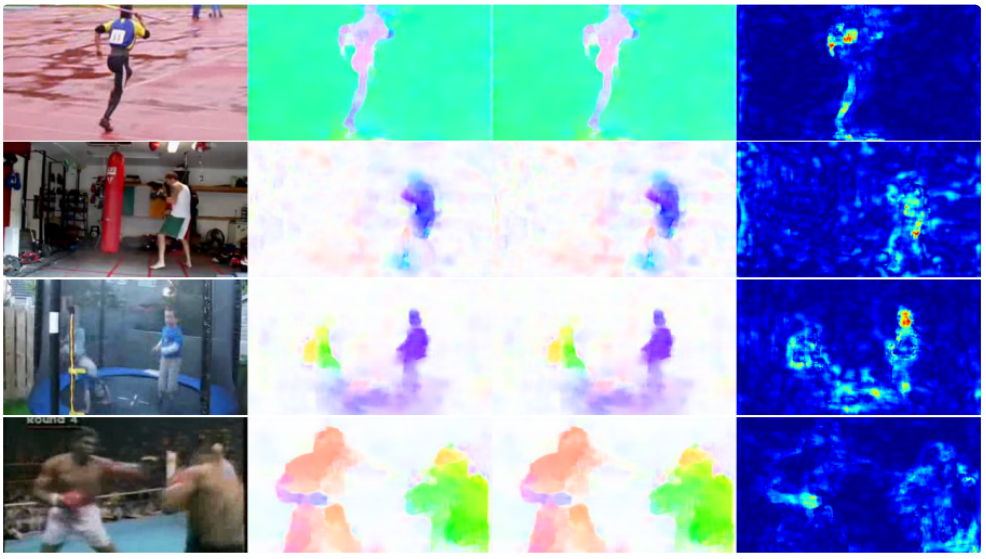
Sparse optical flow of traffic (Each arrow points in the direction of predicted flow of the corresponding pixel).

Nanonets

baseball, and basketball?



Various action classifications



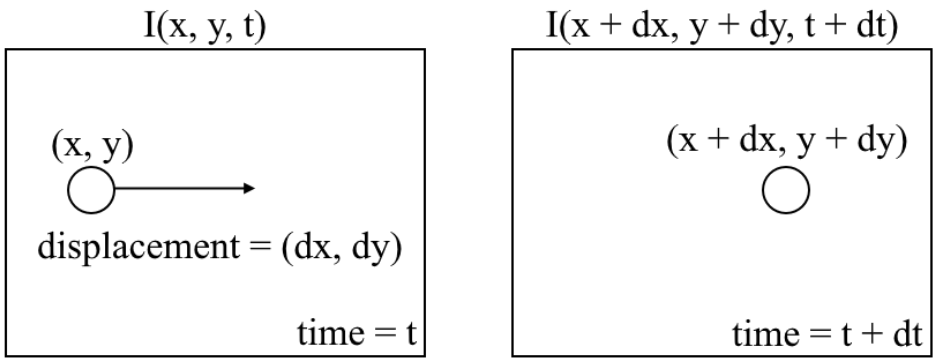
Classifying actions with optical flow

In this tutorial, we will learn what Optical Flow is, how to implement its two main variants (sparse and dense), and also get a big picture of more recent approaches involving deep learning and promising future directions.

Table of Contents

Nanonets

Let us begin with a high-level understanding of optical flow. Optical flow is the motion of objects between consecutive frames of sequence, caused by the relative movement between the object and camera. The problem of optical flow may be expressed as:



Optical flow problem

where between consecutive frames, we can express the image intensity (I) as a function of space (x, y) and time (t). In other words, if we take the first image $I(x, y, t)$ and move its pixels by (dx, dy) over t time, we obtain the new image $I(x + dx, y + dy, t + dt)$.

First, we assume that pixel intensities of an object are constant between consecutive frames.

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t)$$

Constant intensity assumption for optical flow

Second, we take the Taylor Series Approximation of the RHS and remove common terms.

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t + \dots$$

$$\Rightarrow \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t = 0$$

Taylor Series Approximation of pixel intensity

Third, we divide by dt to derive the optical flow equation:

Nanonets

$$\frac{\partial x}{\partial} u + \frac{\partial y}{\partial} v + \frac{\partial t}{\partial} = 0$$

Optical flow equation

where $u = dx/dt$ and $v = dy/dt$.

$dI/dx, dI/dy$, and dI/dt are the image gradients along the horizontal axis, the vertical axis, and time. Hence, we conclude with the problem of optical flow, that is, solving $u(dx/dt)$ and $v(dy/dt)$ to determine movement over time. You may notice that we cannot directly solve the optical flow equation for u and v since there is only one equation for two unknown variables. We will implement some methods such as the Lucas-Kanade method to address this issue.

Sparse vs Dense Optical Flow

Sparse optical flow gives the flow vectors of some "interesting features" (say few pixels depicting the edges or corners of an object) within the frame while *Dense optical flow*, which gives the flow vectors of the entire frame (all pixels) - up to one flow vector per pixel. As you would've guessed, *Dense optical flow* has higher accuracy at the cost of being slow/computationally expensive.

Left: Sparse Optical Flow - track a few "feature" pixels; Right: Dense Optical Flow - estimate the flow of all pixels in the image.

Implementing Sparse Optical Flow

Sparse optical flow selects a sparse feature set of pixels (e.g. interesting features such as edges and corners) to track its velocity vectors (motion). The extracted features are passed in the optical flow function from frame to frame to ensure that the same points are being tracked. There are various implementations of sparse optical flow, including the Lucas–Kanade method, the Horn–Schunck method, the Buxton–Buxton method, and more. We will be

Nanonets

1. Setting up your environment

If you do not already have OpenCV installed, open Terminal and run:

```
pip install opencv-python
```

Now, clone the tutorial repository by running:

```
git clone https://github.com/chuanenlin/optical-flow.git
```

Next, open sparse-starter.py with your text editor. We will be writing all of the code in this Python file.

2. Configuring OpenCV to read a video and setting up parameters

```python
import cv2 as cv
import numpy as np

# Parameters for Shi-Tomasi corner detection
feature_params = dict(maxCorners = 300, qualityLevel = 0.2, minDistance = 2, blockSize = 7)
# Parameters for Lucas-Kanade optical flow
lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITER
# The video feed is read in as a VideoCapture object
cap = cv.VideoCapture("shibuya.mp4")
# Variable for color to draw optical flow track
color = (0, 255, 0)
# ret = a boolean return value from getting the frame, first_frame = the first frame in the entire v
ret, first_frame = cap.read()

while(cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being projecte
    ret, frame = cap.read()
    # Frames are read by intervals of 10 milliseconds. The programs breaks out of the while loop whe
    if cv.waitKey(10) & 0xFF == ord('q'):
        break
# The following frees up resources and closes all windows
cap.release()
cv.destroyAllWindows()
```

sparse-checkpoint1.py hosted with ♡ by GitHub view raw

3. Grayscaling

```python
import cv2 as cv
import numpy as np

# Parameters for Shi-Tomasi corner detection
# feature_params = dict(maxCorners = 300, qualityLevel = 0.2, minDistance = 2, blockSize = 7)
# Parameters for Lucas-Kanade optical flow
# lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITI
# The video feed is read in as a VideoCapture object
```

Nanonets

```
# ret = a boolean return value from getting the frame, first_frame = the first frame in the entire v
# ret, first_frame = cap.read()
# Converts frame to grayscale because we only need the luminance channel for detecting edges - less
prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)

# while(cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being projecte
    # ret, frame = cap.read()
    # Converts each frame to grayscale - we previously only converted the first frame to grayscale
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    # Visualizes checkpoint 2
    cv.imshow("grayscale", gray)
    # Updates previous frame
    prev_gray = gray.copy()
    # Frames are read by intervals of 10 milliseconds. The programs breaks out of the while loop whe
    # if cv.waitKey(10) & 0xFF == ord('q'):
        # break
# The following frees up resources and closes all windows
# cap.release()
# cv.destroyAllWindows()
```

sparse-checkpoint2.py hosted with ♡ by GitHub view raw

4. Shi-Tomasi Corner Detector - selecting the pixels to track

For the implementation of sparse optical flow, we only track the motion of a feature set of pixels. Features in images are points of interest which present rich image content information. For example, such features may be points in the image that are invariant to translation, scale, rotation, and intensity changes such as corners.

The Shi-Tomasi Corner Detector is very similar to the popular Harris Corner Detector which can be implemented by the following three procedures:

1. Determine windows (small image patches) with large gradients (variations in image intensity) when translated in both x and y directions.
2. For each window, compute a score R.
3. Depending on the value of R, each window is classified as a flat, edge, or corner.

If you would like to know more on a step-by-step mathematical explanation of the Harris Corner Detector, feel free to go through these slides.

Shi and Tomasi later made a small but effective modification to the Harris Corner Detector in their paper Good Features to Track.

Nanonets

Shi-Tomasi performs better than Harris. Source

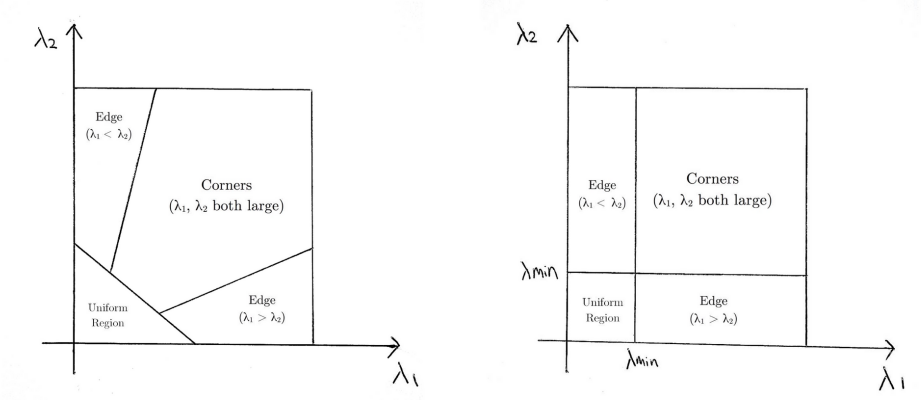
The modification is to the equation in which score R is calculated. In the Harris Corner Detector, the scoring function is given by:

$$R = \det M - k(\operatorname{trace} M)^2$$
$$\det M = \lambda_1 \lambda_2$$
$$\operatorname{trace} M = \lambda_1 + \lambda_2$$

Instead, Shi-Tomasi proposed the scoring function as:

$$R = \min(\lambda_1, \lambda_2)$$

which basically means if R is greater than a threshold, it is classified as a corner. The following compares the scoring functions of Harris (left) and Shi-Tomasi (right) in $\lambda 1 - \lambda 2$ space.



Comparison of Harris and Shi-Tomasi scoring functions on λ1-λ2 space. Source

Nanonets

The documentation of OpenCV's implementation of Shi-Tomasi via `goodFeaturesToTrack()` may be found here.

```python
import cv2 as cv
import numpy as np

# Parameters for Shi-Tomasi corner detection
# feature_params = dict(maxCorners = 300, qualityLevel = 0.2, minDistance = 2, blockSize = 7)
# Parameters for Lucas-Kanade optical flow
# lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITI
# The video feed is read in as a VideoCapture object
# cap = cv.VideoCapture("shibuya.mp4")
# Variable for color to draw optical flow track
# color = (0, 255, 0)
# ret = a boolean return value from getting the frame, first_frame = the first frame in the entire v
# ret, first_frame = cap.read()
# Converts frame to grayscale because we only need the luminance channel for detecting edges - less
# prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
# Finds the strongest corners in the first frame by Shi-Tomasi method - we will track the optical fl
# https://docs.opencv.org/3.0-beta/modules/imgproc/doc/feature_detection.html#goodfeaturestotrack
prev = cv.goodFeaturesToTrack(prev_gray, mask = None, **feature_params)

# while(cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being projecte
    # ret, frame = cap.read()
    # Converts each frame to grayscale - we previously only converted the first frame to grayscale
    # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    # Updates previous frame
    # prev_gray = gray.copy()
    # Frames are read by intervals of 10 milliseconds. The programs breaks out of the while loop whe
    # if cv.waitKey(10) & 0xFF == ord('q'):
        # break
# The following frees up resources and closes all windows
# cap.release()
# cv.destroyAllWindows()
```

sparse-checkpoint3.py hosted with ♡ by GitHub view raw

Tracking Specific Objects

There may be scenarios where you want to only track a specific object of interest (say tracking a certain person) or one category of objects (like all 2 wheeler-vehicles in traffic). You can easily modify the code to track the pixels of the object(s) you want by changing the `prev` variable.

You can also combine Object Detection with this method to only estimate the flow of pixels within the detected bounding boxes. This way you can track all objects of a particular type/category in the video.

Nanonets

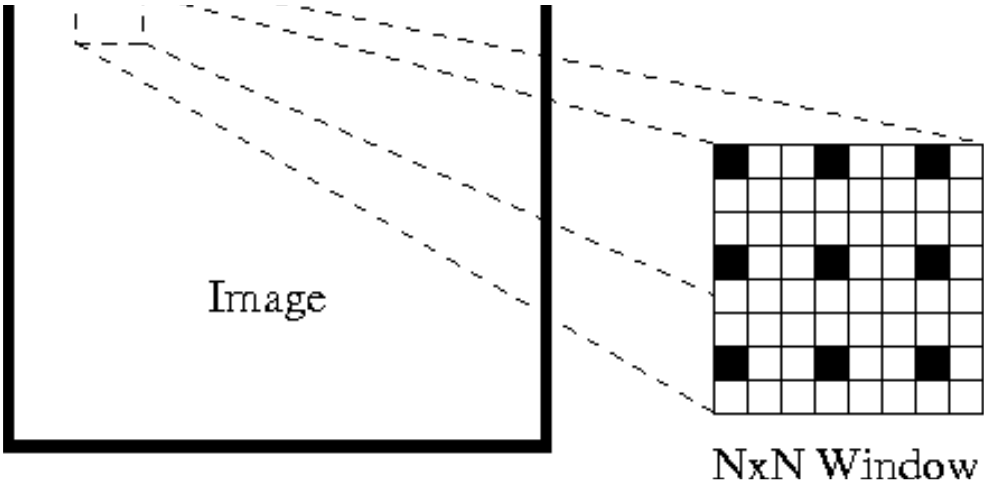
Tracking a single object using optical flow.

5. Lucas-Kanade: Sparse Optical Flow

Lucas and Kanade proposed an effective technique to estimate the motion of interesting features by comparing two consecutive frames in their paper An Iterative Image Registration Technique with an Application to Stereo Vision. The Lucas-Kanade method works under the following assumptions:

1. Two consecutive frames are separated by a small time increment (dt) such that objects are not displaced significantly (in other words, the method work best with slow-moving objects).
2. A frame portrays a “natural” scene with textured objects exhibiting shades of gray that change smoothly.

First, under these assumptions, we can take a small 3x3 window (neighborhood) around the features detected by Shi-Tomasi and assume that all nine points have the same motion.



Lucas-Kanade: Optical flow is estimated for the black pixels

This may be represented as

$$I_x(q_1)V_x + I_y(q_1)V_y = -I_t(q_1)$$
$$I_x(q_2)V_x + I_y(q_2)V_y = -I_t(q_2)$$
$$\vdots$$
$$I_x(q_n)V_x + I_y(q_n)V_y = -I_t(q_n)$$

Lucas-Kanade: 9 pixel intensities in equation form

where $q_1, q_2, \ldots, q_n$ denote the pixels inside the window (e.g. n = 9 for a 3x3 window) and $I_x(q_i)$, $I_y(q_i)$, and $I_t(q_i)$ denote the partial derivatives of image I with respect to position (x, y) and time t, for pixel q_i at the current time.

This is just the Optical Flow Equation (that we described earlier) for each of the *n* pixels.

The set of equations may be represented in the following matrix form where $Av = b$:

Nanonets

$$A = \begin{bmatrix} I_x(q_2) & I_y(q_2) \\ \vdots & \vdots \\ I_x(q_n) & I_y(q_n) \end{bmatrix} \qquad v = \begin{bmatrix} V_x \\ V_y \end{bmatrix} \qquad b = \begin{bmatrix} -I_t(q_2) \\ \vdots \\ -I_t(q_n) \end{bmatrix}$$

9 pixel intensities in matrix form

Take note that previously (see "What is optical flow?" section), we faced the issue of having to solve for two unknown variables with one equation. We now face having to solve for two unknowns (V_x and V_y) with nine equations, which is over-determined.

Second, to address the over-determined issue, we apply least squares fitting to obtain the following two-equation-two-unknown problem:

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i) I_y(q_i) \\ \sum_i I_y(q_i) I_x(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(q_i) I_t(q_i) \\ -\sum_i I_y(q_i) I_t(q_i) \end{bmatrix}$$

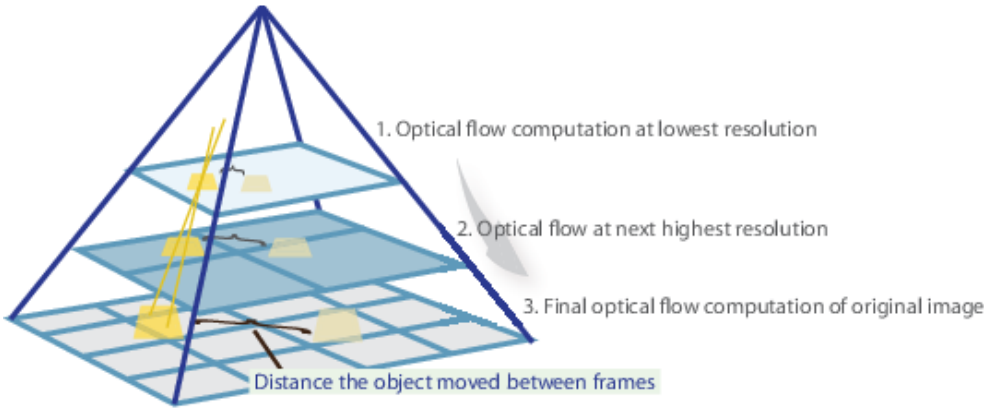
New optical flow equation in two-equation-two-unknown form

where $Vx = u = dx/dt$ denotes the movement of x over time and $Vy = v = dy/dt$ denotes the movement of y over time. Solving for the two variables completes the optical flow problem.

Sparse optical flow of horses on a beach. Source

Nanonets

Lucas-Kanade method only works for small movements (from our initial assumption) and fails when there is large motion. Therefore, the OpenCV implementation of the Lucas-Kanade method adopts pyramids.



Pyramid method computes optical flow at different resolutions. Source

In a high-level view, small motions are neglected as we go up the pyramid and large motions are reduced to small motions - we compute optical flow along with scale. A comprehensive mathematical explanation of OpenCV's implementation may be found in Bouguet's notes and the documentation of OpenCV's implementation of the Lucas-Kanade method via `calcOpticalFlowPyrLK()` may be found here.

```
import cv2 as cv
import numpy as np

# Parameters for Shi-Tomasi corner detection
# feature_params = dict(maxCorners = 300, qualityLevel = 0.2, minDistance = 2, blockSize = 7)
# Parameters for Lucas-Kanade optical flow
# lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITI
# The video feed is read in as a VideoCapture object
# cap = cv.VideoCapture("shibuya.mp4")
# Variable for color to draw optical flow track
# color = (0, 255, 0)
# ret = a boolean return value from getting the frame, first_frame = the first frame in the entire v
# ret, first_frame = cap.read()
# Converts frame to grayscale because we only need the luminance channel for detecting edges - less
# prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
# Finds the strongest corners in the first frame by Shi-Tomasi method - we will track the optical fl
# https://docs.opencv.org/3.0-beta/modules/imgproc/doc/feature_detection.html#goodfeaturestotrack
# prev = cv.goodFeaturesToTrack(prev_gray, mask = None, **feature_params)

# while(cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being projected
    # ret, frame = cap.read()
    # Converts each frame to grayscale - we previously only converted the first frame to grayscale
    # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    # Calculates sparse optical flow by Lucas-Kanade method
```

Nanonets

```
    good_old = prev[status == 1]
    # Selects good feature points for next position
    good_new = next[status == 1]
    # Updates previous frame
    # prev_gray = gray.copy()
    # Updates previous good feature points
    prev = good_new.reshape(-1, 1, 2)
    # Frames are read by intervals of 10 milliseconds. The programs breaks out of the while loop whe
    # if cv.waitKey(10) & 0xFF == ord('q'):
        # break
# The following frees up resources and closes all windows
# cap.release()
# cv.destroyAllWindows()
```

sparse-checkpoint4.py hosted with ♡ by GitHub view raw

6. Visualizing

```
import cv2 as cv
import numpy as np

# Parameters for Shi-Tomasi corner detection
# feature_params = dict(maxCorners = 300, qualityLevel = 0.2, minDistance = 2, blockSize = 7)
# Parameters for Lucas-Kanade optical flow
# lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITE
# The video feed is read in as a VideoCapture object
# cap = cv.VideoCapture("shibuya.mp4")
# Variable for color to draw optical flow track
# color = (0, 255, 0)
# ret = a boolean return value from getting the frame, first_frame = the first frame in the entire v
# ret, first_frame = cap.read()
# Converts frame to grayscale because we only need the luminance channel for detecting edges - less
# prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
# Finds the strongest corners in the first frame by Shi-Tomasi method - we will track the optical fl
# https://docs.opencv.org/3.0-beta/modules/imgproc/doc/feature_detection.html#goodfeaturestotrack
# prev = cv.goodFeaturesToTrack(prev_gray, mask = None, **feature_params)
# Creates an image filled with zero intensities with the same dimensions as the frame - for later dr
mask = np.zeros_like(first_frame)

# while(cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being projected
    # ret, frame = cap.read()
    # Converts each frame to grayscale - we previously only converted the first frame to grayscale
    # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    # Calculates sparse optical flow by Lucas-Kanade method
    # https://docs.opencv.org/3.0-beta/modules/video/doc/motion_analysis_and_object_tracking.html#ca
    # next, status, error = cv.calcOpticalFlowPyrLK(prev_gray, gray, prev, None, **lk_params)
    # Selects good feature points for previous position
    # good_old = prev[status == 1]
    # Selects good feature points for next position
    # good_new = next[status == 1]
    # Draws the optical flow tracks
    for i, (new, old) in enumerate(zip(good_new, good_old)):
        # Returns a contiguous flattened array as (x, y) coordinates for new point
        a, b = new.ravel()
        # Returns a contiguous flattened array as (x, y) coordinates for old point
```

Nanonets

```
        # Draws filled circle (thickness of -1) at new position with green color and radius of 3
        frame = cv.circle(frame, (a, b), 3, color, -1)
    # Overlays the optical flow tracks on the original frame
    output = cv.add(frame, mask)
    # Updates previous frame
    # prev_gray = gray.copy()
    # Updates previous good feature points
    # prev = good_new.reshape(-1, 1, 2)
    # Opens a new window and displays the output frame
    cv.imshow("sparse optical flow", output)
    # Frames are read by intervals of 10 milliseconds. The programs breaks out of the while loop wh
    # if cv.waitKey(10) & 0xFF == ord('q'):
        # break
# The following frees up resources and closes all windows
# cap.release()
# cv.destroyAllWindows()
```

sparse-checkpoint5.py hosted with ♡ by GitHub view raw

And that's it! Open Terminal and run

```
python sparse-starter.py
```

to test your sparse optical flow implementation. 👏

In case you have missed any code, the full code can be found in sparse-solution.py.

Implementing Dense Optical Flow

We've previously computed the optical flow for a sparse feature set of pixels. Dense optical flow attempts to compute the optical flow vector for every pixel of each frame. While such computation may be slower, it gives a more accurate result and a denser result suitable for applications such as learning structure from motion and video segmentation. There are various implementations of dense optical flow. We will be using the Farneback method, one of the most popular implementations, with using OpenCV, an open source library of computer vision algorithms, for implementation.

1. Setting up your environment

If you have not done so already, please follow Step 1 of implementing sparse optical flow to set up your environment.

Next, open dense-starter.py with your text editor. We will be writing all of the code in this Python file.

2. Configuring OpenCV to read a video

Nanonets

```
# The video feed is read in as a VideoCapture object
cap = cv.VideoCapture("shibuya.mp4")
# ret = a boolean return value from getting the frame, first_frame = the first frame in the entire v
ret, first_frame = cap.read()

while(cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being projecte
    ret, frame = cap.read()
    # Opens a new window and displays the input frame
    cv.imshow("input", frame)
    # Frames are read by intervals of 1 millisecond. The programs breaks out of the while loop when
    if cv.waitKey(1) & 0xFF == ord('q'):
        break
# The following frees up resources and closes all windows
cap.release()
cv.destroyAllWindows()
```

dense-checkpoint1.py hosted with ♡ by GitHub view raw

3. Grayscaling

```
import cv2 as cv
import numpy as np

# The video feed is read in as a VideoCapture object
# cap = cv.VideoCapture("shibuya.mp4")
# ret = a boolean return value from getting the frame, first_frame = the first frame in the entire v
# ret, first_frame = cap.read()
# Converts frame to grayscale because we only need the luminance channel for detecting edges - less
prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)

# while(cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being projecte
    # ret, frame = cap.read()
    # Opens a new window and displays the input frame
    # cv.imshow("input", frame)
    # Converts each frame to grayscale - we previously only converted the first frame to grayscale
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    # Visualizes checkpoint 2
    cv.imshow("grayscale", gray)
    # Updates previous frame
    prev_gray = gray
    # Frames are read by intervals of 1 millisecond. The programs breaks out of the while loop when
    # if cv.waitKey(1) & 0xFF == ord('q'):
        # break
# The following frees up resources and closes all windows
# cap.release()
# cv.destroyAllWindows()
```

dense-checkpoint2.py hosted with ♡ by GitHub view raw

4. Farneback Optical Flow

Nanonets

Frame Motion Estimation Based on Polynomial Expansion.

First, the method approximates the windows (see Lucas Kanade section of sparse optical flow implementation for more details) of image frames by quadratic polynomials through polynomial expansion transform. Second, by observing how the polynomial transforms under translation (motion), a method to estimate displacement fields from polynomial expansion coefficients is defined. After a series of refinements, dense optical flow is computed. Farneback's paper is fairly concise and straightforward to follow so I highly recommend going through the paper if you would like a greater understanding of its mathematical derivation.

Nanonets

Dense optical flow of three pedestrians walking in different directions. Source

For OpenCV's implementation, it computes the magnitude and direction of optical flow from a 2-channel array of flow vectors $(dx/dt, dy/dt)$, the optical flow problem. It then visualizes the angle (direction) of flow by hue and the distance (magnitude) of flow by value of HSV color representation. The strength of HSV is always set to a maximum of 255 for optimal visibility. The documentation of OpenCV's implementation of the Farneback method via `calcOpticalFlowFarneback()` may be found here.

```
1  import cv2 as cv
2  import numpy as np
3
4  # The video feed is read in as a VideoCapture object
```

Nanonets

```
# Converts frame to grayscale because we only need the luminance channel for detecting edges - less
# prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)

# while(cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being projecte
    # ret, frame = cap.read()
    # Opens a new window and displays the input frame
    # cv.imshow("input", frame)
    # Converts each frame to grayscale - we previously only converted the first frame to grayscale
    # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    # Calculates dense optical flow by Farneback method
    # https://docs.opencv.org/3.0-beta/modules/video/doc/motion_analysis_and_object_tracking.html#ca
    flow = cv.calcOpticalFlowFarneback(prev_gray, gray, None, 0.5, 3, 15, 3, 5, 1.2, 0)
    # Updates previous frame
    # prev_gray = gray
    # Frames are read by intervals of 1 millisecond. The programs breaks out of the while loop when
    # if cv.waitKey(1) & 0xFF == ord('q'):
        # break
# The following frees up resources and closes all windows
# cap.release()
# cv.destroyAllWindows()
```

dense-checkpoint3.py hosted with ♡ by GitHub view raw

5. Visualizing

```
import cv2 as cv
import numpy as np

# The video feed is read in as a VideoCapture object
# cap = cv.VideoCapture("shibuya.mp4")
# ret = a boolean return value from getting the frame, first_frame = the first frame in the entire v
# ret, first_frame = cap.read()
# Converts frame to grayscale because we only need the luminance channel for detecting edges - less
# prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
# Creates an image filled with zero intensities with the same dimensions as the frame
mask = np.zeros_like(first_frame)
# Sets image saturation to maximum
mask[..., 1] = 255

# while(cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being projecte
    # ret, frame = cap.read()
    # Opens a new window and displays the input frame
    # cv.imshow("input", frame)
    # Converts each frame to grayscale - we previously only converted the first frame to grayscale
    # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    # Calculates dense optical flow by Farneback method
    # https://docs.opencv.org/3.0-beta/modules/video/doc/motion_analysis_and_object_tracking.html#ca
    # flow = cv.calcOpticalFlowFarneback(prev_gray, gray, None, 0.5, 3, 15, 3, 5, 1.2, 0)
    # Computes the magnitude and angle of the 2D vectors
    magnitude, angle = cv.cartToPolar(flow[..., 0], flow[..., 1])
    # Sets image hue according to the optical flow direction
    mask[..., 0] = angle * 180 / np.pi / 2
    # Sets image value according to the optical flow magnitude (normalized)
    mask[..., 2] = cv.normalize(magnitude, None, 0, 255, cv.NORM_MINMAX)
```

Nanonets

```
    cv.imshow("dense optical flow", rgb)
    # Updates previous frame
    # prev_gray = gray
    # Frames are read by intervals of 1 millisecond. The programs breaks out of the while loop when
    # if cv.waitKey(1) & 0xFF == ord('q'):
        # break
# The following frees up resources and closes all windows
# cap.release()
# cv.destroyAllWindows()
```

dense-checkpoint4.py hosted with ♡ by GitHub view raw

And that's it! Open Terminal and run

```
python dense-starter.py
```

to test your dense optical flow implementation. 👏

In case you have missed any code, the full code can be found in dense-solution.py.

Optical Flow using Deep Learning

While the problem of optical flow has historically been an optimization problem, recent approaches by applying deep learning have shown impressive results. Generally, such approaches take **two video frames as input** to **output the optical flow (colour-coded image)**, which may be expressed as:

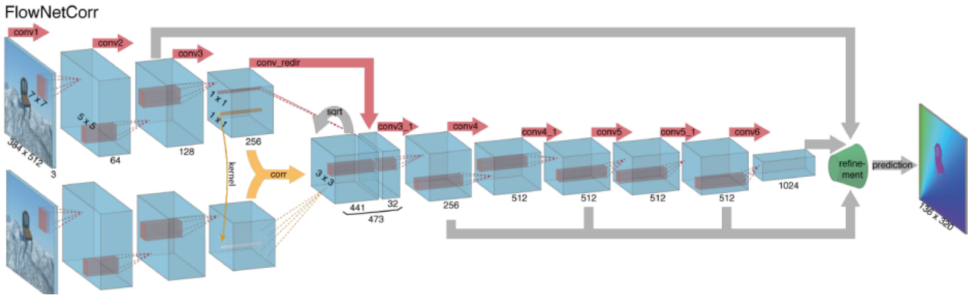
$$(u, v) = f(I_{t-1}, I_t)$$

Generation equation of optical flow computed with a deep learning approach.

Nanonets

Output of a deep learning model: colour-coded image; colour encodes the direction of pixel while intensity indicates their speed.

where u is the motion in the x direction, v is the motion in the y direction, and f is a neural network that takes in two consecutive frames I_{t-1} (frame at time = $t-1$) and I_t (frame at time = t) as input.



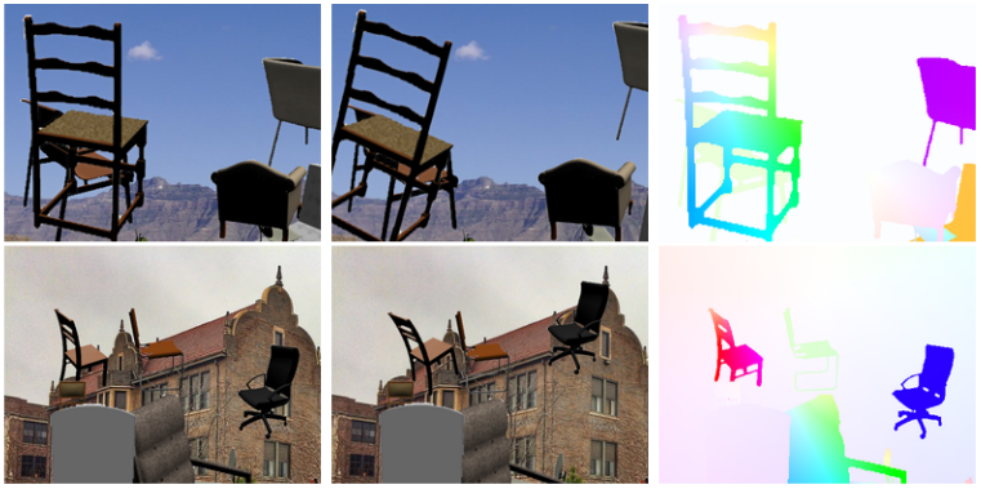
Architecture of FlowNetCorr, a convolutional neural network for end-to-end learning of optical flow. Source

Computing optical flow with deep neural networks requires large amounts of training data which is particularly hard to obtain. This is because labeling video footage for optical flow requires accurately figuring out the exact motion of each and every point of an image to subpixel accuracy. To address the issue of labeling training data, researchers used computer graphics to simulate massive realistic worlds. Since the worlds are generated by instruction, the motion of each and every point of an image in a video sequence is known. Some examples of such include MPI-Sintel, an open-

Nanonets

also with optical flow labeling.

Synthetically generated data for training Optical Flow Models – MPI-Sintel dataset. Source



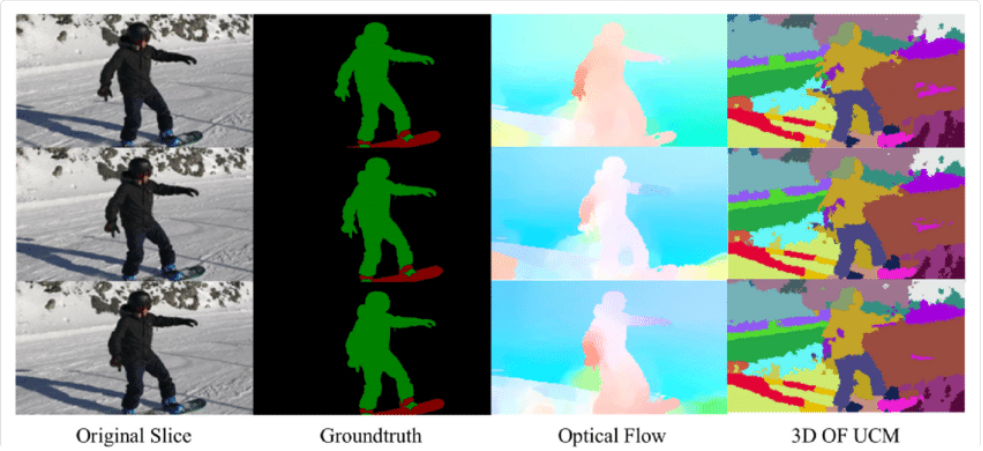
Synthetically generated data for training Optical Flow Models – Flying Chairs dataset. Source

Solving optical flow problems with deep learning is an extremely hot topic at the moment, with variants of FlowNet, SPyNet, PWC-Net, and more each outperforming one another on various benchmarks.

Optical Flow application: Semantic Segmentation

Nanonets

to see applications of optical flow in junction with several other fundamental computer visions tasks. For example, the task of semantic segmentation is to divide an image into series of regions corresponding to unique object classes yet closely placed objects with identical textures are often difficult for single frame segmentation techniques. If the objects are placed separately, however, the distinct motions of the objects may be highly helpful where discontinuity in the dense optical flow field correspond to boundaries between objects.



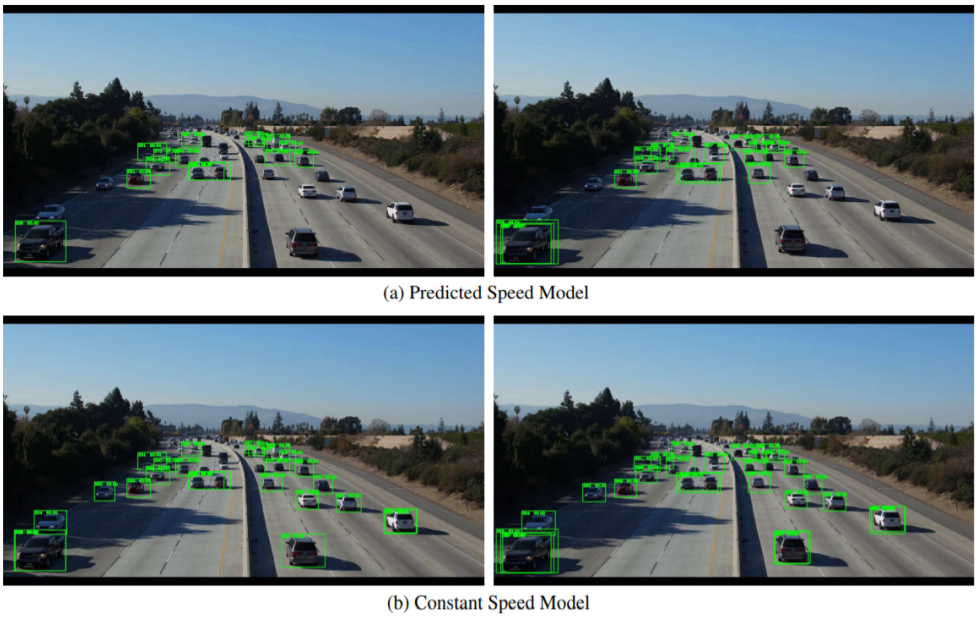
Semantic segmentation generated from optical flow. Source

Optical Flow application: Object Detection & Tracking

Another promising application of optical flow may be with object detection and tracking or, in a high-level form, towards building real-time vehicle tracking and traffic analysis systems. Since sparse optical flow utilizes tracking of points of interest, such real-time systems may be performed by feature-based optical flow techniques from either from a stationary camera or cameras attached to vehicles.

Nanonets

Real-time tracking of vehicles with optical flow. Source



Optical flow can be used to predict vehicle speeds. Source

Conclusion

Fundamentally, optical flow vectors function as input to a myriad of higher-level tasks requiring scene understanding of video sequences while these tasks may further act as building blocks to yet more complex systems such as facial expression analysis, autonomous vehicle navigation, and much more. Novel applications for optical flow yet to be discovered are limited only by the ingenuity of its designers.

Lazy to code, don't want to spend on GPUs? Head over to Nanonets and build computer vision models for free!

Nanonets

Login

Add a comment

MARKDOWN

ADD COMMENT

Powered by **Commento**

NEWER POST
A 2019 guide to 3D Human Pose Estimation

OLDER POST
A 2019 guide to Human Pose Estimation with Deep Learning

Nanonets

PRODUCTS

- Object Detection
- Image Classification

SOLUTIONS

- NSFW
- Drones
- E-commerce
- Inspection
- Multi Label Classification
- OCR API
- Hygiene & Safety Compliance
- Insurance

CASE STUDIES

- Counting Cars
- Solar Panel faults
- Windmill faults
- NSFW Content Moderation
- Furniture Research & Recommendation

COMPANY

- About Us
- Blog

CONTACT

156 2nd Street, San Francisco, CA 94105, USA
+1 650 382 8676
info@nanonets.com