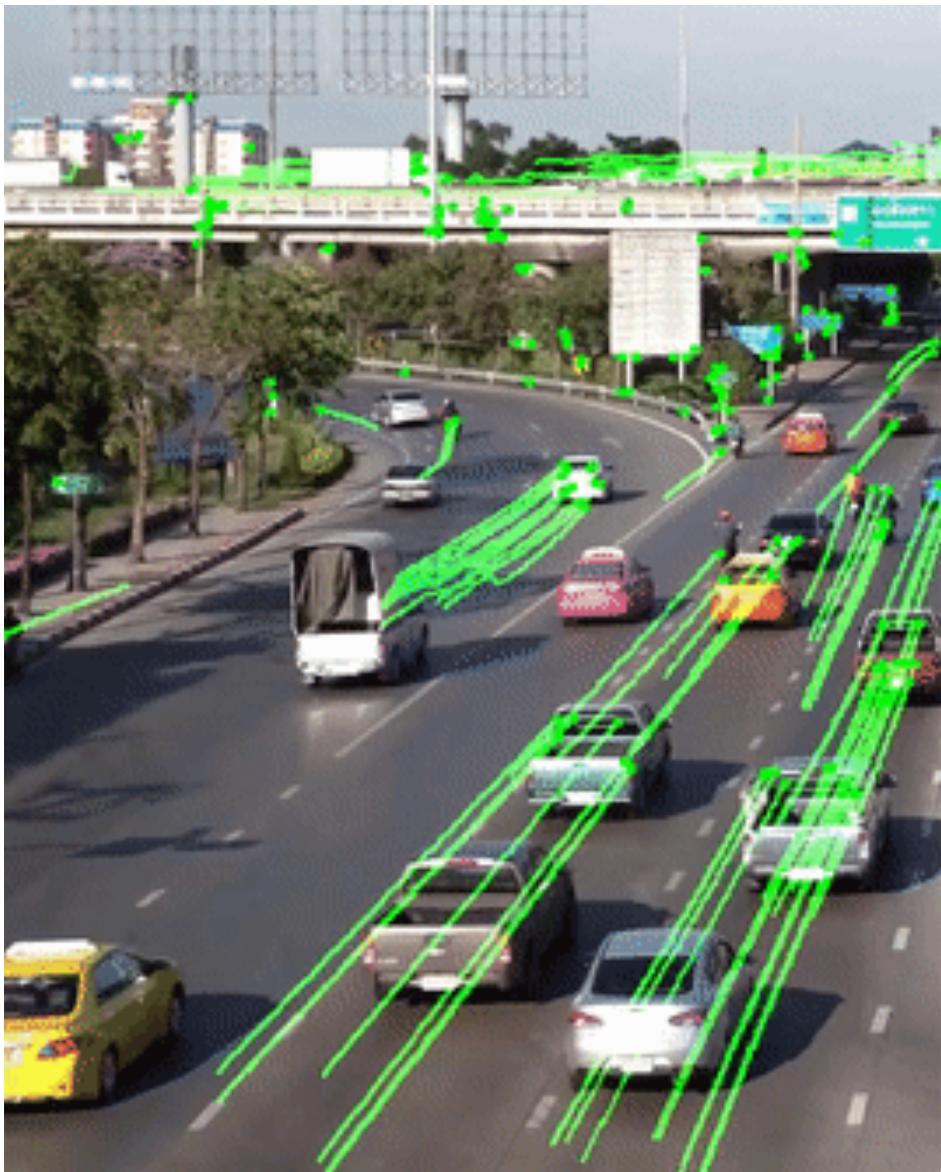


[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

[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** 3 years ago

⌚ 12 MIN READ

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

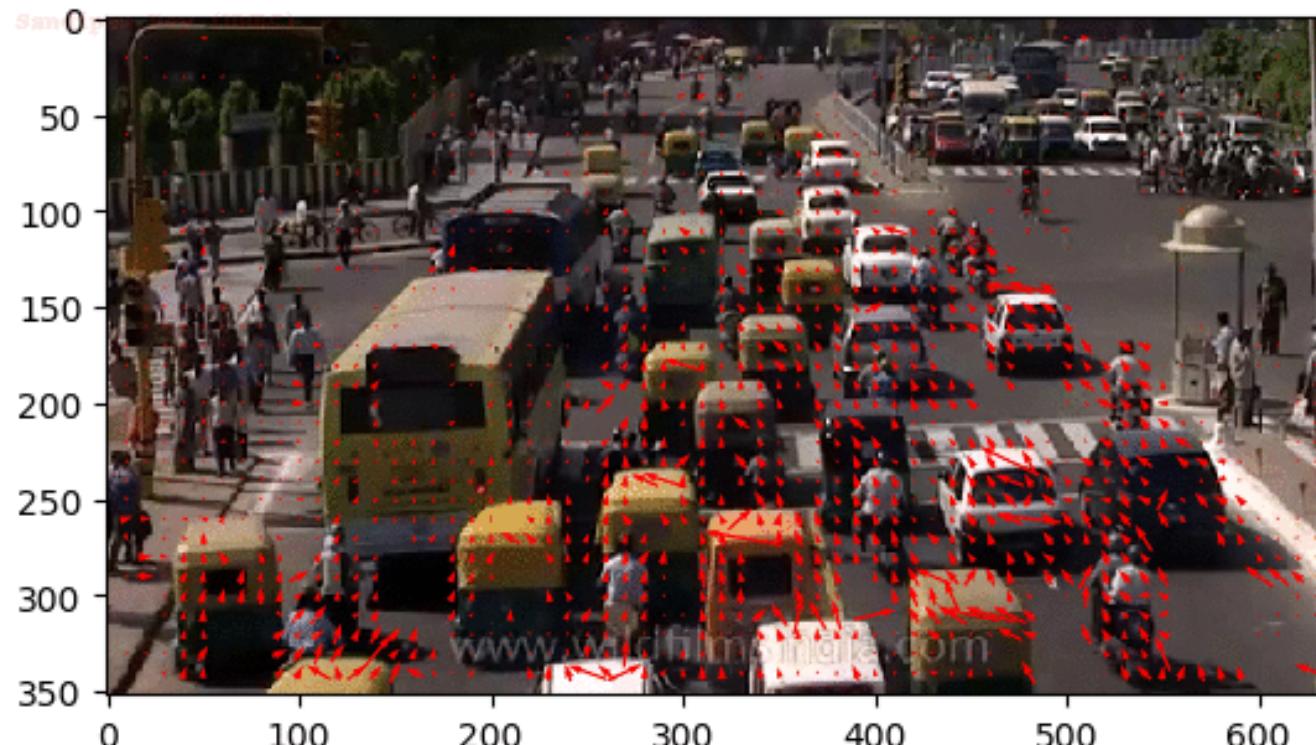
In this tutorial, we dive into the fundamentals of Optical Flow, look at some of its applications and implement its two main variants (sparse and dense). We also briefly discuss more recent approaches using deep learning and promising future directions.

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-

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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).



Products Solutions

TRY NANONETS OCR →

baseball, and basketball?

1-Archery



2-Baseball



3-Basketball



4-BMX



5-Bowling



6-Boxing



7-Cheerleading



8-Discus throw



9-Diving



10-Football



11-Golf



12-Gymnastics



13-Hammer throw



14-High jump



15-Hockey



16-Hurdling



17-Javelin



18-Long jump



19-Pole vault



20-Rowing



21-Running



22-Shotput



23-Skating



24-Skiing



25-Soccer



26-Swimming



27-Tennis



28-Volleyball



29-Weight lifting



30-Wrestling



Various action classifications

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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.

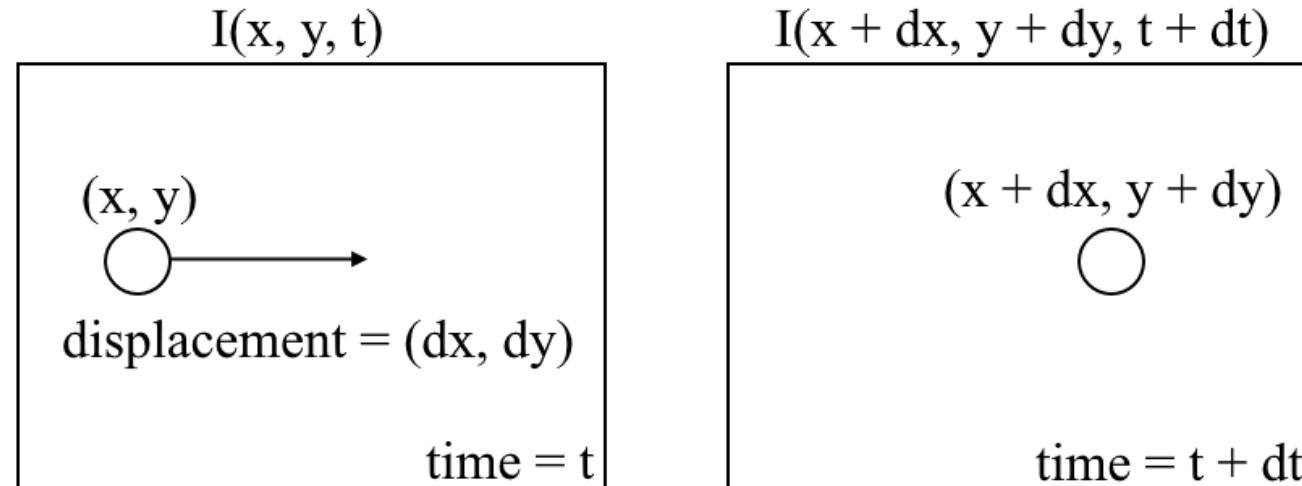
Table of Contents

What is Optical Flow?

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)Deep learning and beyond

What is optical flow?

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

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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.

$$\begin{aligned} I(x + \delta x, y + \delta y, t + \delta t) &= I(x, y, t) + \frac{\partial I}{\partial x} \delta x + \frac{\partial I}{\partial y} \delta y + \frac{\partial I}{\partial t} \delta t + \dots \\ &\Rightarrow \frac{\partial I}{\partial x} \delta x + \frac{\partial I}{\partial y} \delta y + \frac{\partial I}{\partial t} \delta t = 0 \end{aligned}$$

Taylor Series Approximation of pixel intensity

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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

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

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

implementations of sparse optical flow, including the Lucas-Kanade method, the Horn-Schunck method, the Buxton-Buxton method, and more. We will be using the Lucas-Kanade method with OpenCV, an open source library of computer vision algorithms, for implementation.

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

```
1 import cv2 as cv
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
5  feature_params = dict(maxCorners = 300, qualityLevel = 0.2, minDistance = 2, blockSize =
6  # Parameters for Lucas-Kanade optical flow
7  lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv.TERM_CRITERIA_EPS | cv.T
8  # The video feed is read in as a VideoCapture object
9  cap = cv.VideoCapture("shibuya.mp4")
10 # Variable for color to draw optical flow track
11 color = (0, 255, 0)
12 # ret = a boolean return value from getting the frame, first_frame = the first frame in t
13 ret, first_frame = cap.read()
14
15 while(cap.isOpened()):
16     # ret = a boolean return value from getting the frame, frame = the current frame bein
17     ret, frame = cap.read()
18     # Frames are read by intervals of 10 milliseconds. The programs breaks out of the whi
19     if cv.waitKey(10) & 0xFF == ord('q'):
20         break
21     # The following frees up resources and closes all windows
22     cap.release()
23     cv.destroyAllWindows()
```

[sparse-checkpoint1.py](#) hosted with ❤ by GitHub[view raw](#)

3. Grayscaleing

```
1 import cv2 as cv
2 import numpy as np
3
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
7  # lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv.TERM_CRITERIA_EPS | cv
8  # The video feed is read in as a VideoCapture object
9  # cap = cv.VideoCapture("shibuya.mp4")
10 # Variable for color to draw optical flow track
11 # color = (0, 255, 0)
12 # ret = a boolean return value from getting the frame, first_frame = the first frame in t
13 # ret, first_frame = cap.read()
14 # Converts frame to grayscale because we only need the luminance channel for detecting ed
15 prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
16
17 # while(cap.isOpened()):
18     # ret = a boolean return value from getting the frame, frame = the current frame bein
19     # ret, frame = cap.read()
20     # Converts each frame to grayscale - we previously only converted the first frame to
21     gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
22     # Visualizes checkpoint 2
23     cv.imshow("grayscale", gray)
24     # Updates previous frame
25     prev_gray = gray.copy()
26     # Frames are read by intervals of 10 milliseconds. The programs breaks out of the whi
27     # if cv.waitKey(10) & 0xFF == ord('q'):
28         # break
29     # The following frees up resources and closes all windows
30     # cap.release()
31     # cv.destroyAllWindows()
```

[sparse-checkpoint2.py](#) hosted with ❤ by GitHub[view raw](#)



pixels to track

Products Solutions

TRY NANONETS OCR →

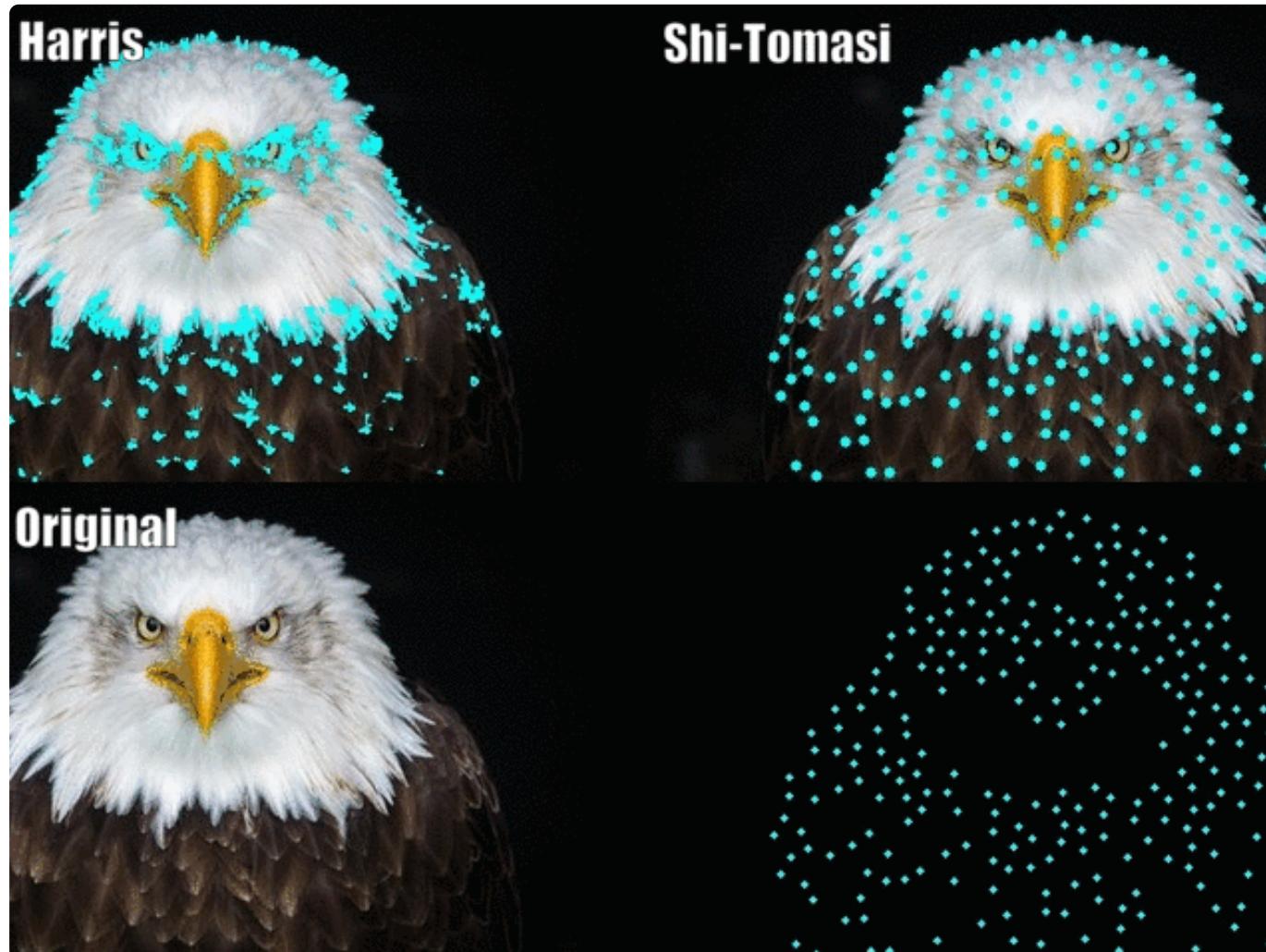
For the implementation of sparse optical flow, we only track the motion of a feature set of pixels. Features in images are points of interest which present rich image content information. For example, such features may be points in the image that are invariant to translation, scale, rotation, and intensity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

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

Shi-Tomasi performs better than Harris. [Source](#)

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

$$R = \det M - k(\text{trace } M)^2$$

$$\det M = \lambda_1 \lambda_2$$

$$\text{trace } M = \lambda_1 + \lambda_2$$

Instead, Shi-Tomasi proposed the scoring function as:

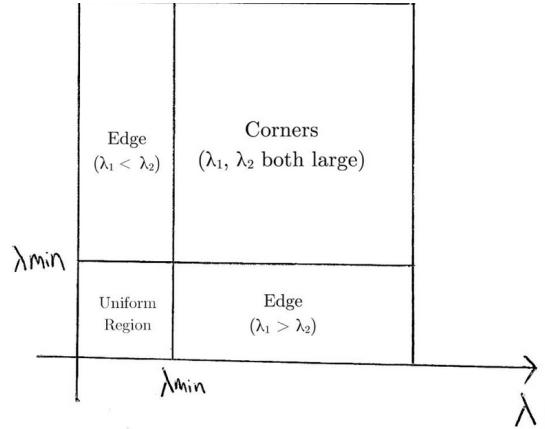
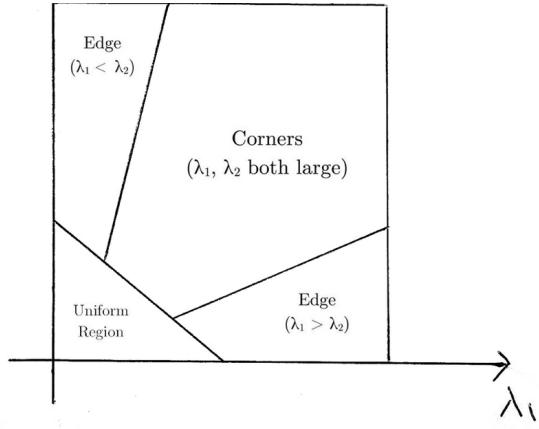
$$R = \min (\lambda_1, \lambda_2)$$

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.



Products Solutions

TRY NANONETS OCR →



Comparison of Harris and Shi-Tomasi scoring functions on λ_1 - λ_2 space. [Source](#)

For Shi-Tomasi, only when λ_1 and λ_2 are above a minimum threshold λ_{min} is the window classified as a corner.

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

```
1 import cv2 as cv
2 import numpy as np
3
4 # Parameters for Shi-Tomasi corner detection
5 # feature_params = dict(maxCorners = 300, qualityLevel = 0.2, minDistance = 2, blockSize
6 # Parameters for Lucas-Kanade optical flow
7 # lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv.TERM_CRITERIA_EPS | cv
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
11 # color = (0, 255, 0)
12 # ret = a boolean return value from getting the frame, first_frame = the first frame in t
13 # ret, first_frame = cap.read()
14 # Converts frame to grayscale because we only need the luminance channel for detecting ed
15 # prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
16 # Finds the strongest corners in the first frame by Shi-Tomasi method - we will track the
17 # https://docs.opencv.org/3.0-beta/modules/imgproc/doc/feature_detection.html#goodfeature
18 prev = cv.goodFeaturesToTrack(prev_gray, mask = None, **feature_params)
19
20 # while(cap.isOpened()):
21     # ret = a boolean return value from getting the frame, frame = the current frame bein
22     # ret, frame = cap.read()
23     # Converts each frame to grayscale - we previously only converted the first frame to
24     # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
25     # Updates previous frame
26     # prev_gray = gray.copy()
27     # Frames are read by intervals of 10 milliseconds. The programs breaks out of the whi
28     # if cv.waitKey(10) & 0xFF == ord('q'):
29         # break
30 # The following frees up resources and closes all windows
31 # cap.release()
32 # cv.destroyAllWindows()
```

[sparse-checkpoint3.py](#) hosted with ❤ by GitHub[view raw](#)

Tracking Specific Objects

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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.



[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

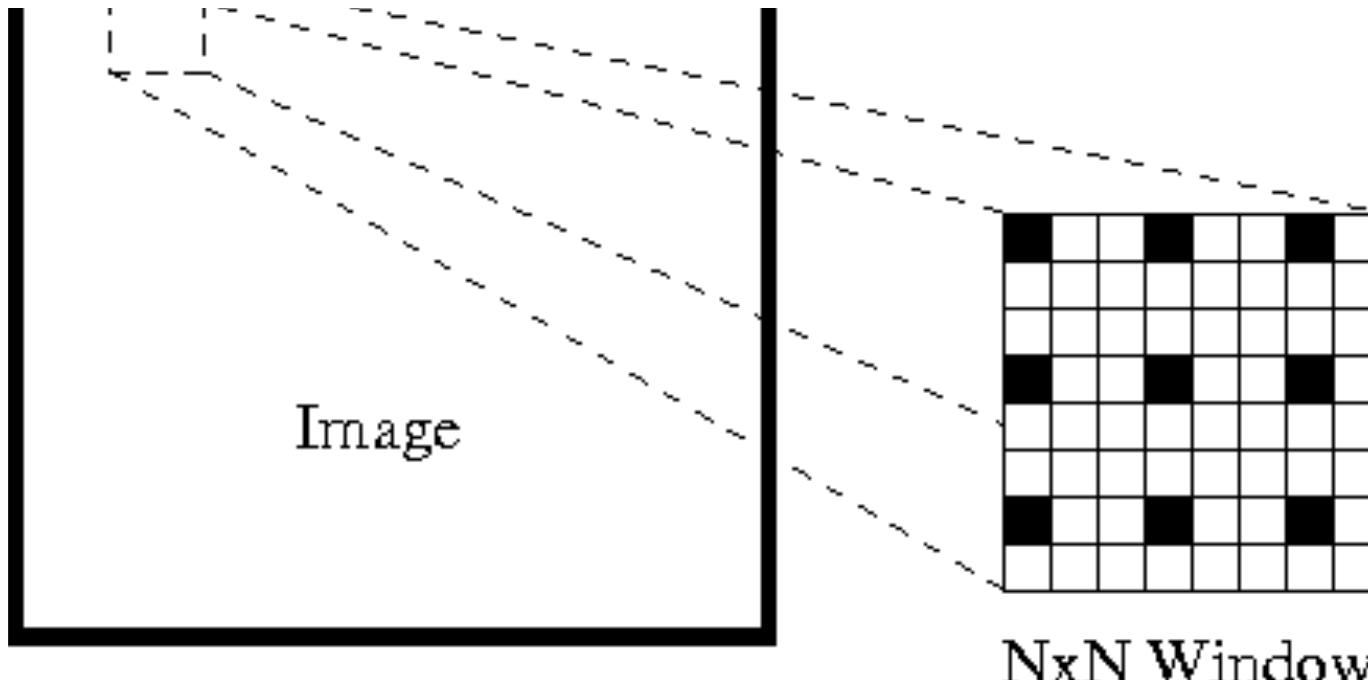
5. Lucas-Kanade: Sparse Optical Flow

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 3×3 window (neighborhood) around the features detected by Shi-Tomasi and assume that all nine points have the same motion.

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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

This may be represented as

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

$$I_x(q_2)V_x + I_y(q_2)V_y = -I_t(q_2)$$

⋮
⋮
⋮

$$I_x(q_n)V_x + I_y(q_n)V_y = -I_t(q_n)$$

Lucas-Kanade: 9 pixel intensities in equation form

where q_1, q_2, \dots, q_n denote the pixels inside the window (e.g. $n = 9$ for a 3×3 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.

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

$$A = \begin{bmatrix} I_x(q_1) & I_y(q_1) \\ I_x(q_2) & I_y(q_2) \\ \vdots & \vdots \\ I_x(q_n) & I_y(q_n) \end{bmatrix} \quad v = \begin{bmatrix} V_x \\ V_y \end{bmatrix} \quad b = \begin{bmatrix} -I_t(q_1) \\ -I_t(q_2) \\ \vdots \\ -I_t(q_n) \end{bmatrix}$$

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:

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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

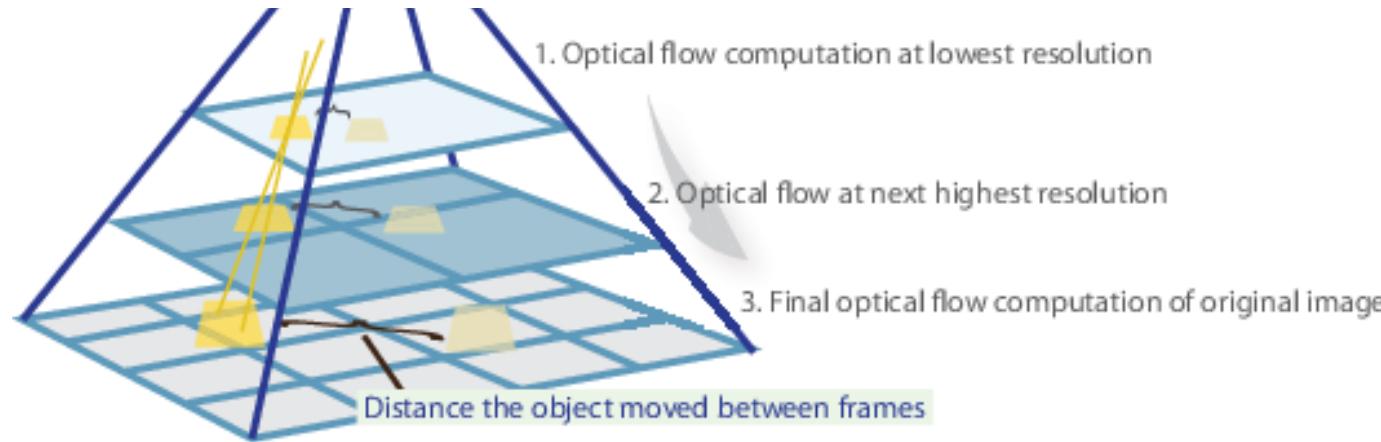
New optical flow equation in two-equation-two-unknown form

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.

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

Sparse optical flow of horses on a beach. [Source](#)

In a nutshell, we identify some interesting features to track and iteratively compute the optical flow vectors of these points. However, adopting the 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.

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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](#).

```
1 import cv2 as cv
2 import numpy as np
3
4 # Parameters for Shi-Tomasi corner detection
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
8 # The video feed is read in as a VideoCapture object
9 # cap = cv.VideoCapture("shibuya.mp4")
10 # Variable for color to draw optical flow track
11 # color = (0, 255, 0)
12 # ret = a boolean return value from getting the frame, first_frame = the first frame in t
13 # ret, first_frame = cap.read()
14 # Converts frame to grayscale because we only need the luminance channel for detecting ed
15 # prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
16 # Finds the strongest corners in the first frame by Shi-Tomasi method - we will track the
17 # https://docs.opencv.org/3.0-beta/modules/imgproc/doc/feature_detection.html#goodfeature
18 # prev = cv.goodFeaturesToTrack(prev_gray, mask = None, **feature_params)
19
20 # while(cap.isOpened()):
21     # ret = a boolean return value from getting the frame, frame = the current frame bein
22     # ret, frame = cap.read()
23     # Converts each frame to grayscale - we previously only converted the first frame to
24     # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
25     # Calculates sparse optical flow by Lucas-Kanade method
26     # https://docs.opencv.org/3.0-beta/modules/video/doc/motion_analysis_and_object_track
27     next, status, error = cv.calcOpticalFlowPyrLK(prev_gray, gray, prev, None, **lk_param
28     # Selects good feature points for previous position
29     good_old = prev[status == 1]
30     # Selects good feature points for next position
31     good_new = next[status == 1]
32     # Updates previous frame
33     # prev_gray = gray.copy()
34     # Updates previous good feature points
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
38         # break
39 # The following frees up resources and closes all windows
40 # cap.release()
41 # cv.destroyAllWindows()
```

sparse-checkpoint4.py hosted with ❤ by GitHub

[view raw](#)

6. Visualizing

```
1 import cv2 as cv
2 import numpy as np
3
4 # Parameters for Shi-Tomasi corner detection
5 # feature_params = dict(maxCorners = 300, qualityLevel = 0.2, minDistance = 2, blockSize
6 # Parameters for Lucas-Kanade optical flow
7 # lk_params = dict(winSize = (15,15), maxLevel = 2, criteria = (cv.TERM_CRITERIA_EPS | cv
8 # The video feed is read in as a VideoCapture object
9 # cap = cv.VideoCapture("shibuya.mp4")
10 # Variable for color to draw optical flow track
11 # color = (0, 255, 0)
12 # ret = a boolean return value from getting the frame, first_frame = the first frame in t
13 # ret, first_frame = cap.read()
14 # Converts frame to grayscale because we only need the luminance channel for detecting ed
15 # prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
16 # Finds the strongest corners in the first frame by Shi-Tomasi method - we will track the
17 # https://docs.opencv.org/3.0-beta/modules/imgproc/doc/feature_detection.html#goodfeature
18 # prev = cv.goodFeaturesToTrack(prev_gray, mask = None, **feature_params)
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
22 # while(cap.isOpened()):  
23     # ret = a boolean return value from getting the frame, frame = the current frame bein  
24     # ret, frame = cap.read()  
25     # Converts each frame to grayscale - we previously only converted the first frame to  
26     # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)  
27     # Calculates sparse optical flow by Lucas-Kanade method  
28     # https://docs.opencv.org/3.0-beta/modules/video/doc/motion_analysis_and_object_track  
29     # next, status, error = cv.calcOpticalFlowPyrLK(prev_gray, gray, prev, None, **lk_par  
30     # Selects good feature points for previous position  
31     # good_old = prev[status == 1]  
32     # Selects good feature points for next position  
33     # good_new = next[status == 1]  
34     # Draws the optical flow tracks  
35     for i, (new, old) in enumerate(zip(good_new, good_old)):  
36         # Returns a contiguous flattened array as (x, y) coordinates for new point  
37         a, b = new.ravel()  
38         # Returns a contiguous flattened array as (x, y) coordinates for old point  
39         c, d = old.ravel()  
40         # Draws line between new and old position with green color and 2 thickness  
41         mask = cv.line(mask, (a, b), (c, d), color, 2)  
42         # Draws filled circle (thickness of -1) at new position with green color and radi  
43         frame = cv.circle(frame, (a, b), 3, color, -1)  
44     # Overlays the optical flow tracks on the original frame  
45     output = cv.add(frame, mask)  
46     # Updates previous frame  
47     # prev_gray = gray.copy()  
48     # Updates previous good feature points
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
52     # Frames are read by intervals of 10 milliseconds. The programs breaks out of the whi
53     # if cv.waitKey(10) & 0xFF == ord('q'):
54         # break
55     # The following frees up resources and closes all windows
56     # cap.release()
57     # 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

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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

```
1 import cv2 as cv
2 import numpy as np
3
4 # The video feed is read in as a VideoCapture object
5 cap = cv.VideoCapture("shibuya.mp4")
6 # ret = a boolean return value from getting the frame, first_frame = the first frame in t
7 ret, first_frame = cap.read()
8
9 while(cap.isOpened()):
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
13     cv.imshow("input", frame)
14     # Frames are read by intervals of 1 millisecond. The programs breaks out of the while
15     if cv.waitKey(1) & 0xFF == ord('q'):
16         break
17     # The following frees up resources and closes all windows
18     cap.release()
19     cv.destroyAllWindows()
```

[dense-checkpoint1.py hosted with ❤ by GitHub](#)[view raw](#)

3. Grayscaleing

```
1 import cv2 as cv
2 import numpy as np
3
4 # The video feed is read in as a VideoCapture object
5 # cap = cv.VideoCapture("shibuya.mp4")
6 # ret = a boolean return value from getting the frame, first_frame = the first frame in t
7 # ret, first_frame = cap.read()
8 # Converts frame to grayscale because we only need the luminance channel for detecting ed
9 prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
10
11 # while(cap.isOpened()):
12     # ret = a boolean return value from getting the frame, frame = the current frame bein
13     # ret, frame = cap.read()
14     # Opens a new window and displays the input frame
15     # cv.imshow("input", frame)
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
19     cv.imshow("grayscale", gray)
20     # Updates previous frame
21     prev_gray = gray
22     # Frames are read by intervals of 1 millisecond. The programs breaks out of the while
23     # if cv.waitKey(1) & 0xFF == ord('q'):
24         # break
25     # The following frees up resources and closes all windows
26     # cap.release()
27     # cv.destroyAllWindows()
```

[dense-checkpoint2.py hosted with ❤ by GitHub](#)[view raw](#)

4. Farneback Optical Flow

Gunnar Farneback proposed an effective technique to estimate the motion of interesting features by comparing two consecutive frames in his paper [Two-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

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

understanding of its mathematical derivation.

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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
5 # cap = cv.VideoCapture("shibuya.mp4")
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
9  # prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
10
11 # while(cap.isOpened()):
12     # ret = a boolean return value from getting the frame, frame = the current frame being
13     # read
14     # ret, frame = cap.read()
15     # Opens a new window and displays the input frame
16     # cv.imshow("input", frame)
17     # Converts each frame to grayscale - we previously only converted the first frame to
18     # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
19     # Calculates dense optical flow by Farneback method
20     # https://docs.opencv.org/3.0-beta/modules/video/doc/motion_analysis_and_object_tracking.html#cv2.calcOpticalFlowFarneback
21     flow = cv.calcOpticalFlowFarneback(prev_gray, gray, None, 0.5, 3, 15, 3, 5, 1.2, 0)
22     # Updates previous frame
23     # prev_gray = gray
24     # Frames are read by intervals of 1 millisecond. The program breaks out of the while
25     # loop if q is pressed
26     # if cv.waitKey(1) & 0xFF == ord('q'):
27         # break
28     # The following frees up resources and closes all windows
29     # cap.release()
30     # cv.destroyAllWindows()
```

[dense-checkpoint3.py hosted with ❤ by GitHub](#)[view raw](#)

5. Visualizing

```
1 import cv2 as cv
2 import numpy as np
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
6  # ret = a boolean return value from getting the frame, first_frame = the first frame in t
7  # ret, first_frame = cap.read()
8  # Converts frame to grayscale because we only need the luminance channel for detecting ed
9  # prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
10 # Creates an image filled with zero intensities with the same dimensions as the frame
11 mask = np.zeros_like(first_frame)
12 # Sets image saturation to maximum
13 mask[..., 1] = 255
14
15 # while(cap.isOpened()):
16     # ret = a boolean return value from getting the frame, frame = the current frame bein
17     # ret, frame = cap.read()
18     # Opens a new window and displays the input frame
19     # cv.imshow("input", frame)
20     # Converts each frame to grayscale - we previously only converted the first frame to
21     # gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
22     # Calculates dense optical flow by Farneback method
23     # https://docs.opencv.org/3.0-beta/modules/video/doc/motion_analysis_and_object_track
24     # flow = cv.calcOpticalFlowFarneback(prev_gray, gray, None, 0.5, 3, 15, 3, 5, 1.2, 0)
25     # Computes the magnitude and angle of the 2D vectors
26     magnitude, angle = cv.cartToPolar(flow[..., 0], flow[..., 1])
27     # Sets image hue according to the optical flow direction
28     mask[..., 0] = angle * 180 / np.pi / 2
29     # Sets image value according to the optical flow magnitude (normalized)
30     mask[..., 2] = cv.normalize(magnitude, None, 0, 255, cv.NORM_MINMAX)
31     # Converts HSV to RGB (BGR) color representation
32     rgb = cv.cvtColor(mask, cv.COLOR_HSV2BGR)
```

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

```
36      # prev_gray = gray
37      # Frames are read by intervals of 1 millisecond. The programs breaks out of the while
38      # if cv.waitKey(1) & 0xFF == ord('q'):
39          # break
40      # The following frees up resources and closes all windows
41      # cap.release()
42      # 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

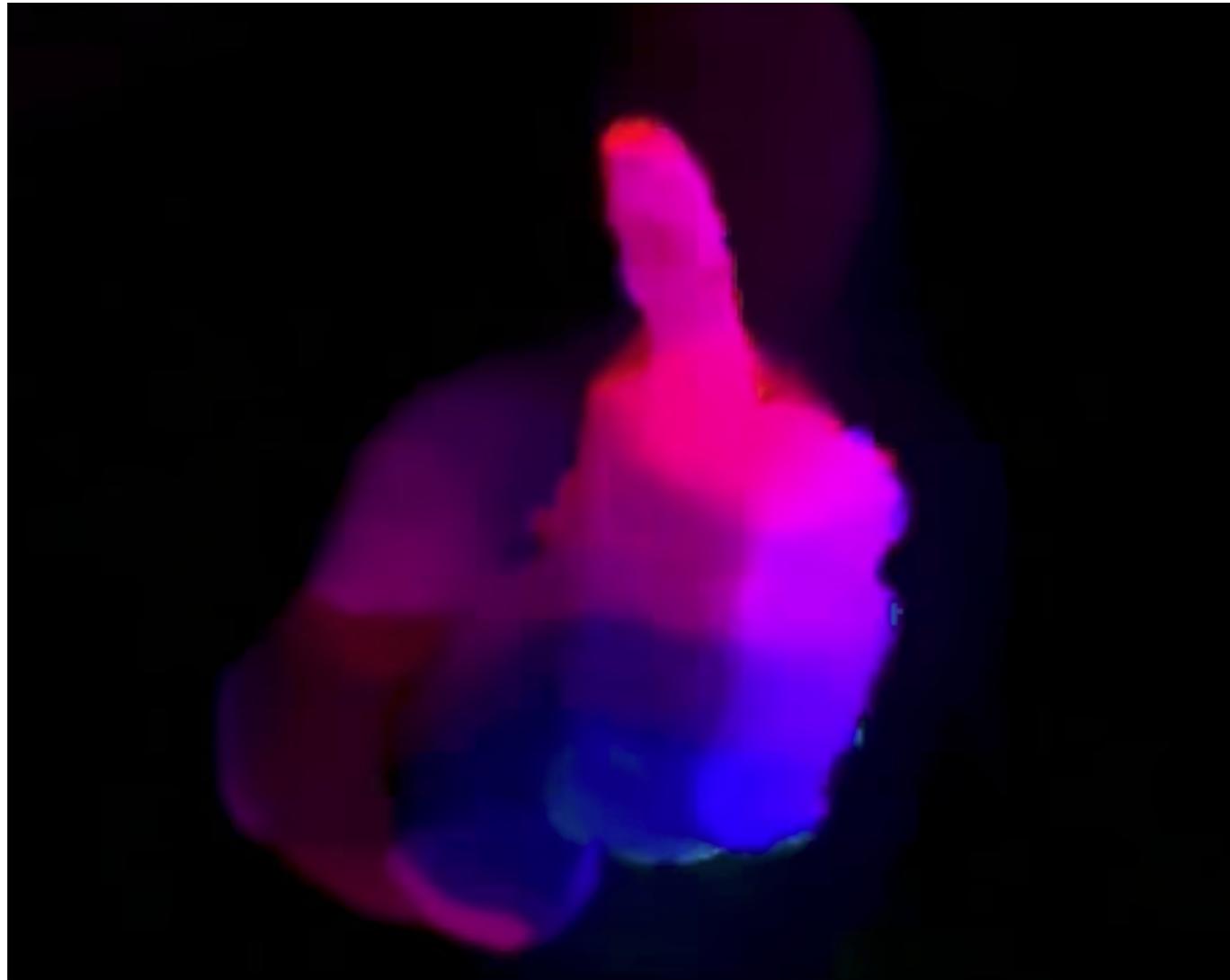
While the problem of optical flow has historically been an optimization

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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.

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

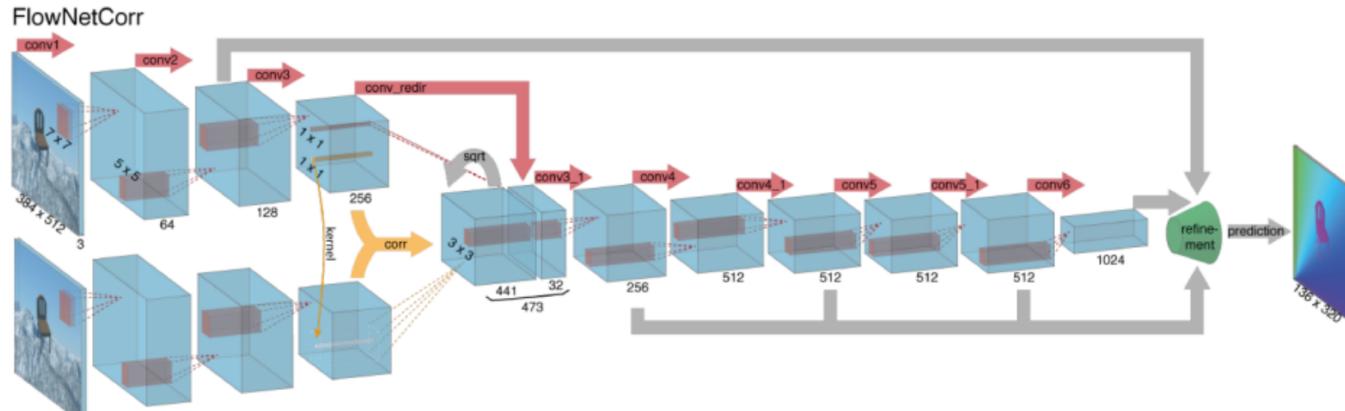
Output of a deep learning model: colour-coded image; colour encodes the direction of pixel while intensity indicates their speed.



Products Solutions

TRY NANONETS OCR →

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](#)

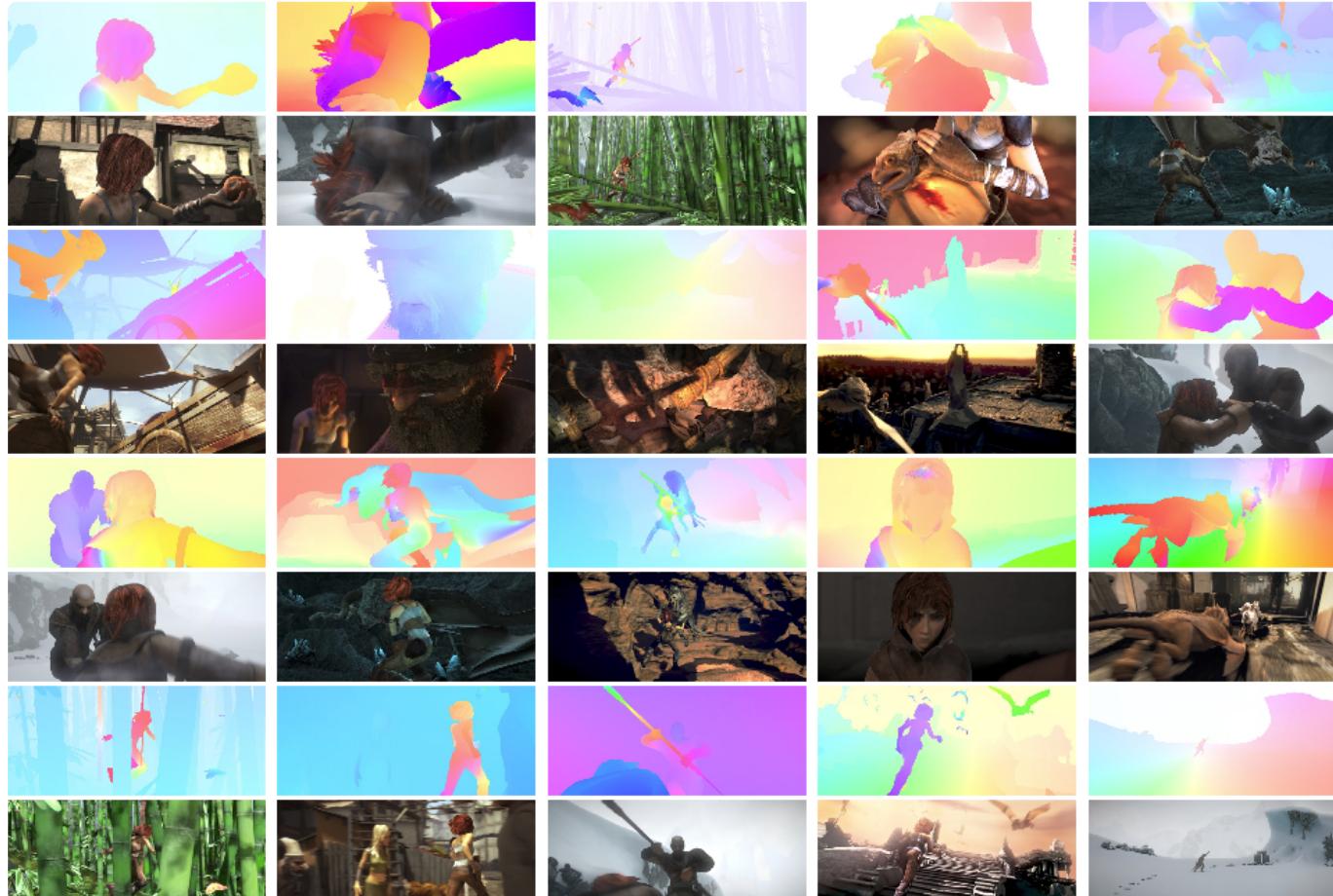
Computing optical flow with deep neural networks requires large amounts of training data which is particularly hard to obtain. This is because labeling video footage for optical flow requires accurately figuring out the exact motion of each and every point of an image to subpixel accuracy. To address 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-



Products Solutions

TRY NANONETS OCR →

backgrounds also with optical flow labeling.



Synthetically generated data for training Optical Flow Models – MPI-Sintel dataset. [Source](#)



Products Solutions

TRY NANONETS OCR →



Synthetically generated data for training Optical Flow Models – Flying Chairs dataset. [Source](#)

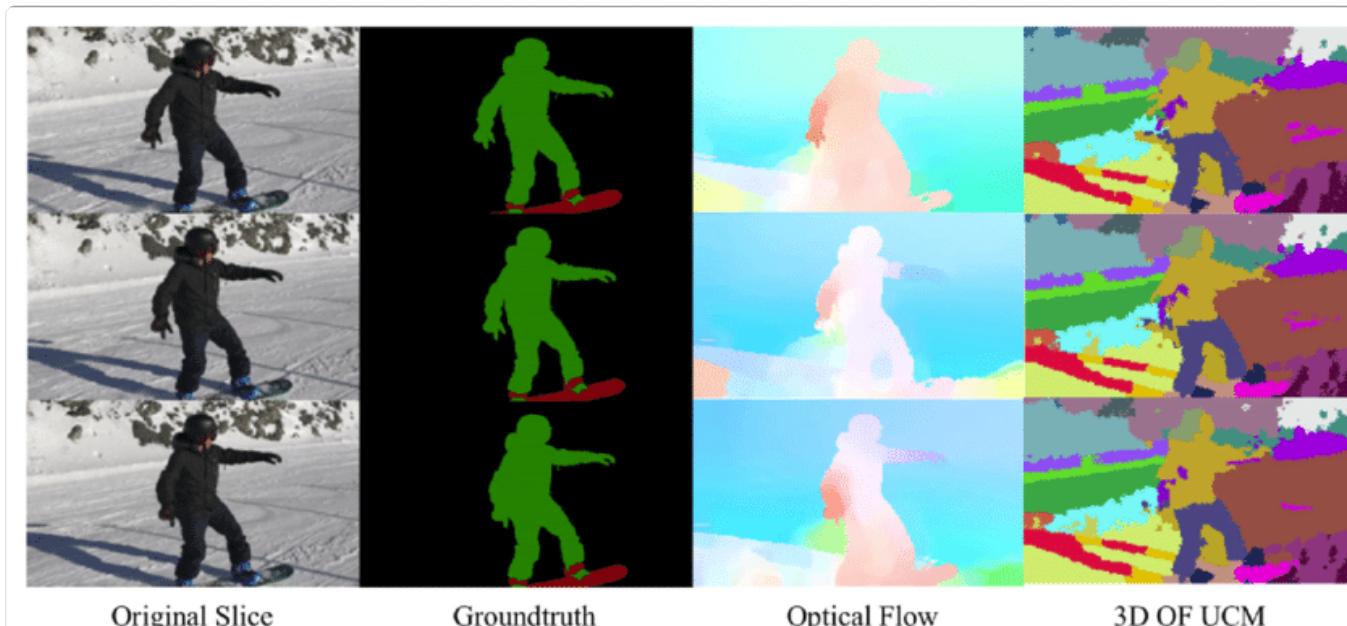
Solving optical flow problems with deep learning is an extremely hot topic at the moment, with variants of [FlowNet](#), [SPyNet](#), [PWC-Net](#), and more each outperforming one another on various [benchmarks](#).

Optical Flow application: Semantic Segmentation

The optical flow field is a vast mine of information for the observed scene. As

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

fundamental computer vision 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](#)

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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.

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

Loc 3



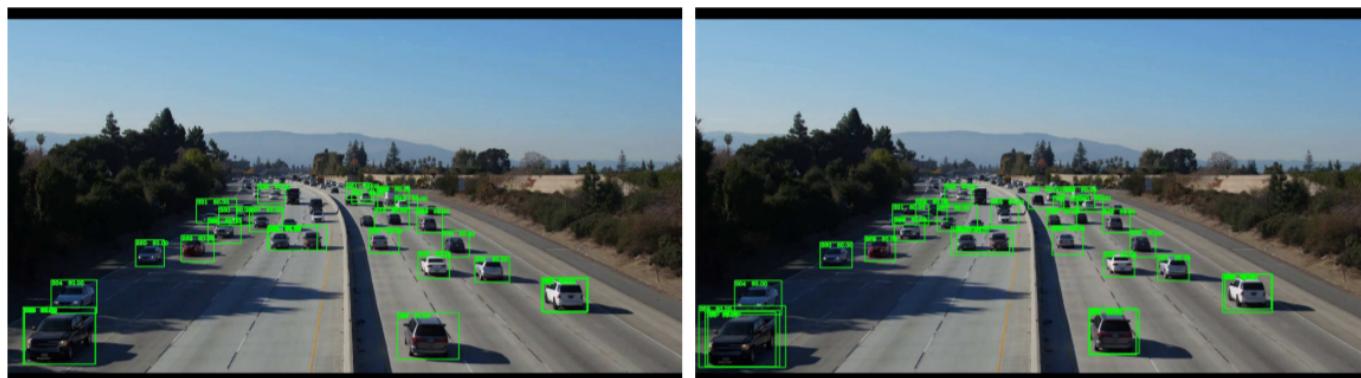
Loc 4



Real-time tracking of vehicles with optical flow. [Source](#)

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

(a) Predicted Speed Model



(b) Constant Speed Model

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

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)

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!

[Start now](#)[Login](#)

Add a comment

[M ↓ MARKDOWN](#)[ADD COMMENT](#)[Upvotes](#) [Newest](#) [Oldest](#)

[Products](#) [Solutions](#)[TRY NANONETS OCR →](#)