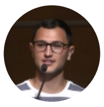
Human pose estimation using OpenPose with TensorFlow (Part 2)



Ale Solano
Nov 20, 2017 · 9 min read

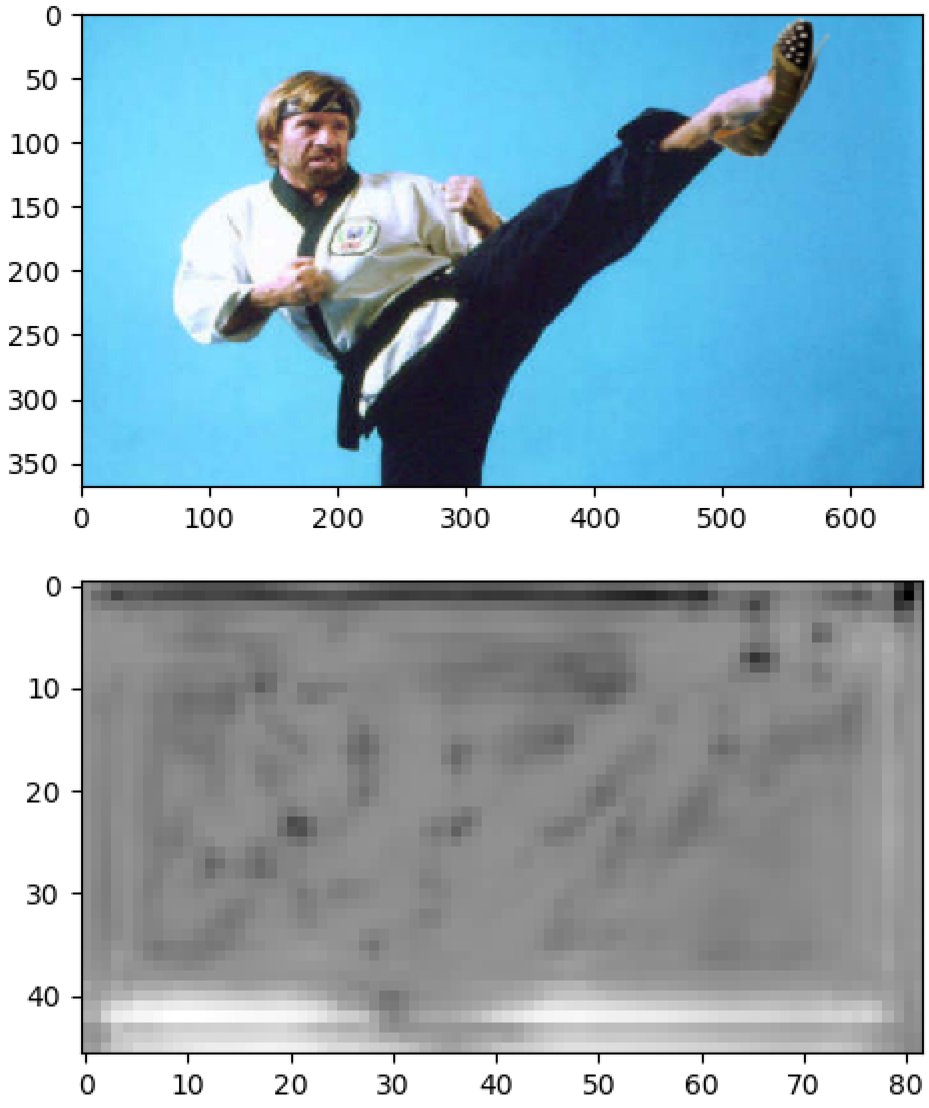
This is the second part of a series of blog articles. Part 1 here.

So, where were we?

Oh, yes. We were trying to revolutionize the world of robotics and virtual reality. Our goal was to extract the position of each of the body parts of every person appearing in an image with no more sensors than a digital camera.

OpenPose is a library that allow us to do so. The library consists of a neural network and some other functions that magically do the work. However, we discovered it ran on Caffe and we don't feel so comfortable with that. So we converted the neural network to a format that TensorFlow understands.

Quickly, we fed our OpenPose neural network with a picture of Chuck Norris trying to predict his movements and… we failed, obviously. (Yes, I still make jokes about Chuck Norris. I'm trapped in 2000s internet. I explain why in this video)



Our unsuccessful attempt to beat Chuck Norris in a Kung Fu battle

The output of the net was, surprisingly, an image of 57 layers of depth: 18 layers for body parts location, 1 for background and 38 for limbs information in both X and Y directions. We concluded **we need to make some post-processing** on this super tensor in order to get valuable information out of it.

And… that's what we're talking about today.

We're going to analyze and understand every stage of the pipeline that an image follows before we finally get the skeletons of the people shown in the picture.

Exploring the pipeline

Without further ado, say hello to the OpenPose pipeline.

Openpose Pipeline

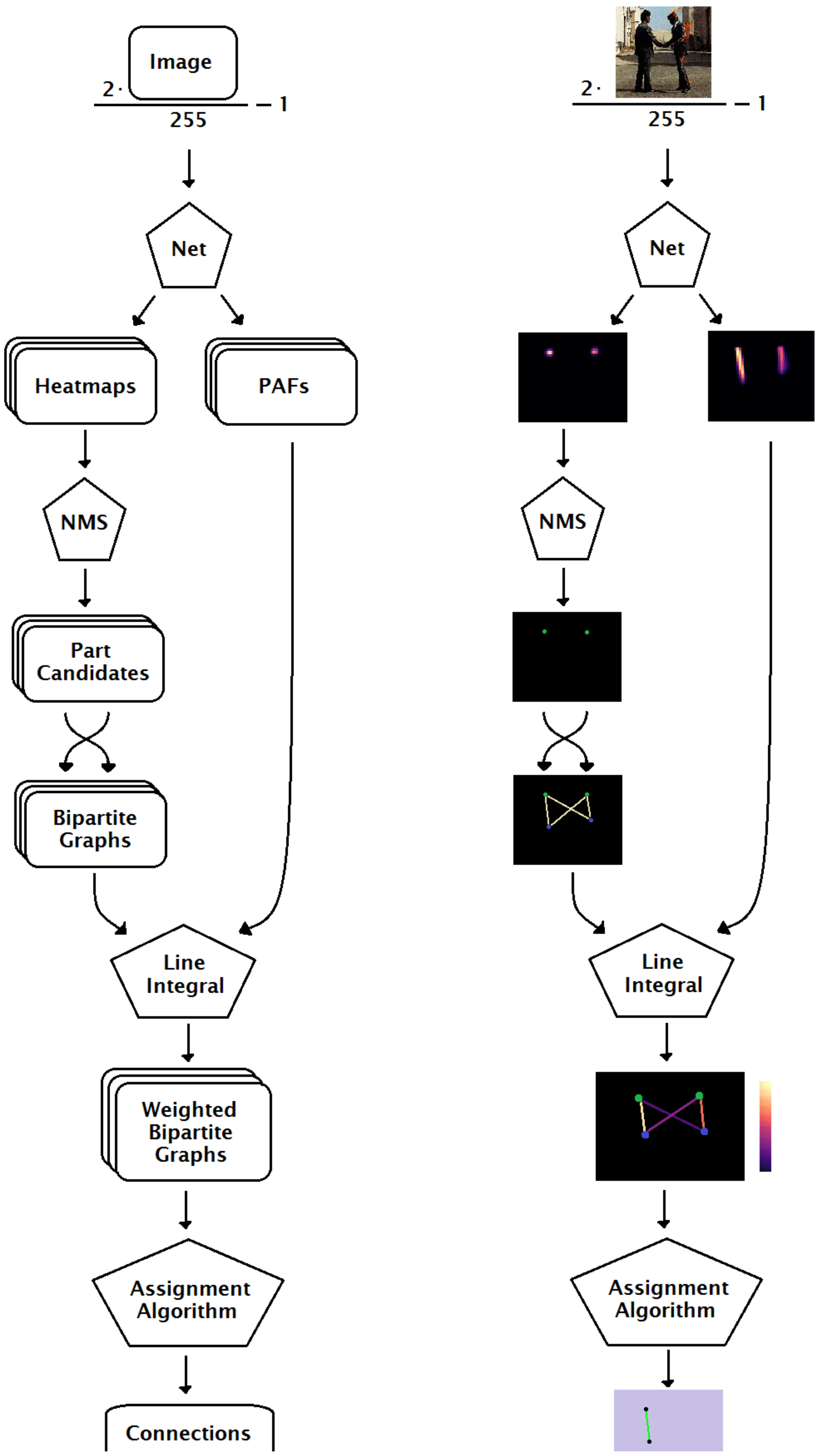
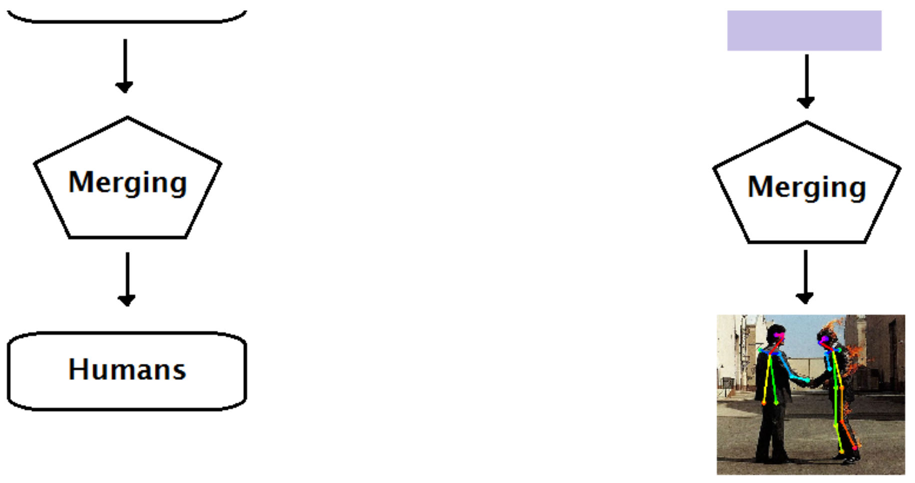


Image
2 ·
255
− 1
Net
Heatmaps
PAFs
NMS
Part Candidates
Bipartite Graphs
Line Integral
Weighted Bipartite Graphs
Assignment Algorithm
Connections
2 ·
255
− 1
Net
NMS
Line Integral
Assignment Algorithm



God bless MS Paint

Huge, right? It well deserves a blog post.

The algorithms are very rich in content. I love how the pipeline involves different fields like deep learning, calculus, graph theory and set theory. I'll try to explain each stage as clear as possible without making it boring.

I'll add a code snippet in Python for each algorithm presented, in order to make it clearer to understand. Also, I will provide you with a script to run OpenPose in Python easily. This is all thanks to the amazing work of Ildoo Kim, that translated most code of the OpenPose Library to Python. I've learned a lot about the OpenPose pipeline just looking at its code in the GitHub repository below:

ildoonet/tf-openpose
tf-openpose - Openpose from CMU implemented using Tensorflow with Custom Architecture for…
github.com

Parts and Pairs

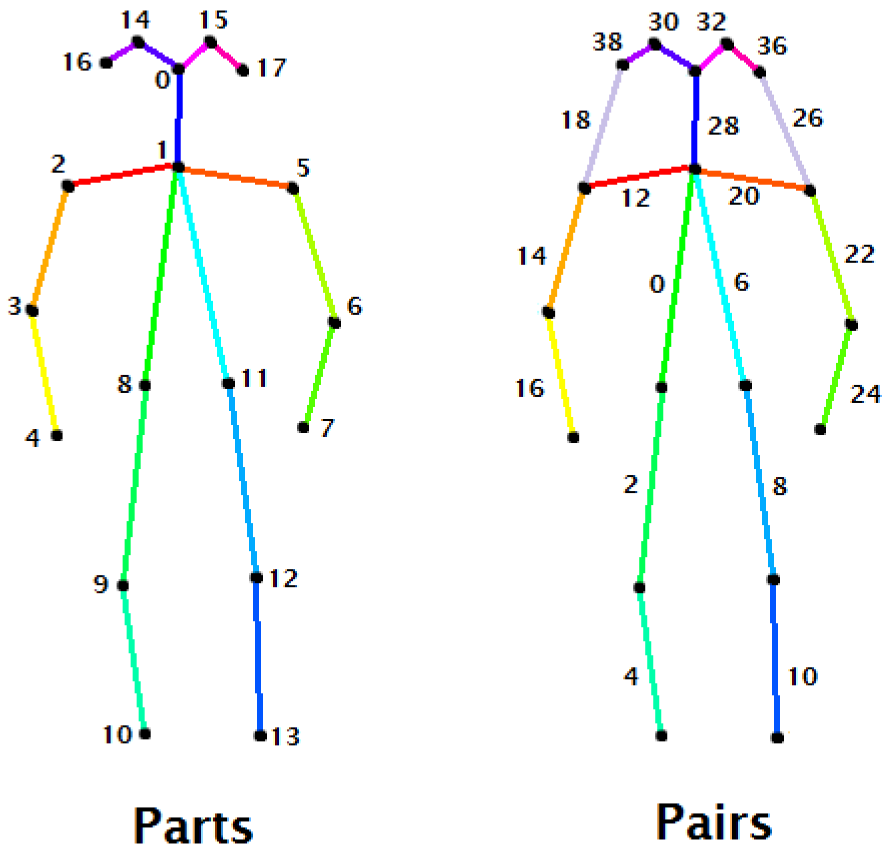
Parts. Pairs. Parts. Pairs. They may look like the same word, but they're not.

We will talk about parts, pairs, pairs of parts, Peter Piper picked a pair of pepper parts, and so on. So we better understand what we're talking about.

- A body **part** is an element of the body, like neck, left shoulder or right hip.

- A **pair** is a couple of parts. A *connection* between parts. We could say limb, but the connection between the nose and the left eye is definitely not a limb. Also, we are going to deal with connections between ears and shoulders that do not exist in real life.

These skeletons show the indeces of parts and paris on the COCO dataset. For the MPII dataset these skeletons vary slightly: there is one more body part corresponding to lower abs.



Parts and Pairs indexes for COCO dataset

Preprocessing

First but not least, we convert the image from [0, 255] to [-1, 1]. Though it's the easiest step of all, I failed miserably on this in my first post of these series.

```
img = img * (2.0 / 255.0) - 1.0
```

Neural Network

Here is where magic happens. It's the same dark force that makes planes fly and smartphones exist. We'd better treat it like a black box.

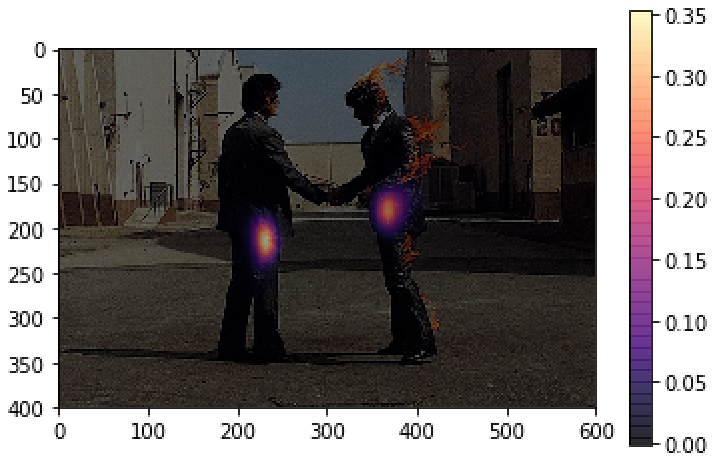
The last operation of the neural network returns a tensor consisting of 57 matrices. However, this last op is just a concatenation of two different tensors: heatmaps and PAFs. The understanding of these two tensors is essential.

$$HM: \quad \mathbb{R}^2 \quad \rightarrow \quad \mathbb{R} \qquad\qquad PAF: \quad \mathbb{R}^2 \quad \rightarrow \quad \mathbb{R}^2$$

$$[0,81] \times [0,45] \rightarrow [0,1] \qquad\qquad [0,81] \times [0,45] \rightarrow [-1,1] \times [-1,1]$$

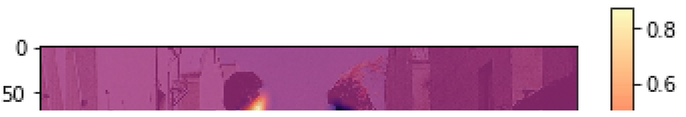
Heatmaps and PAFs as math functions

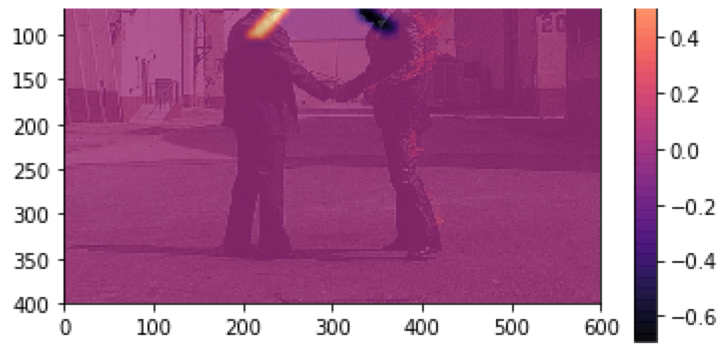
- A **heatmap** is a matrix that stores the confidence the network has that a certain pixel contains a certain part. There are 18 (+1) heatmaps associated with each one of the parts and indexed as we showed in the drawing of the skeletons. We will extract the **location of the body parts** out of these 18 matrices.



Heatmaps don't measure the hotness of a certain body part

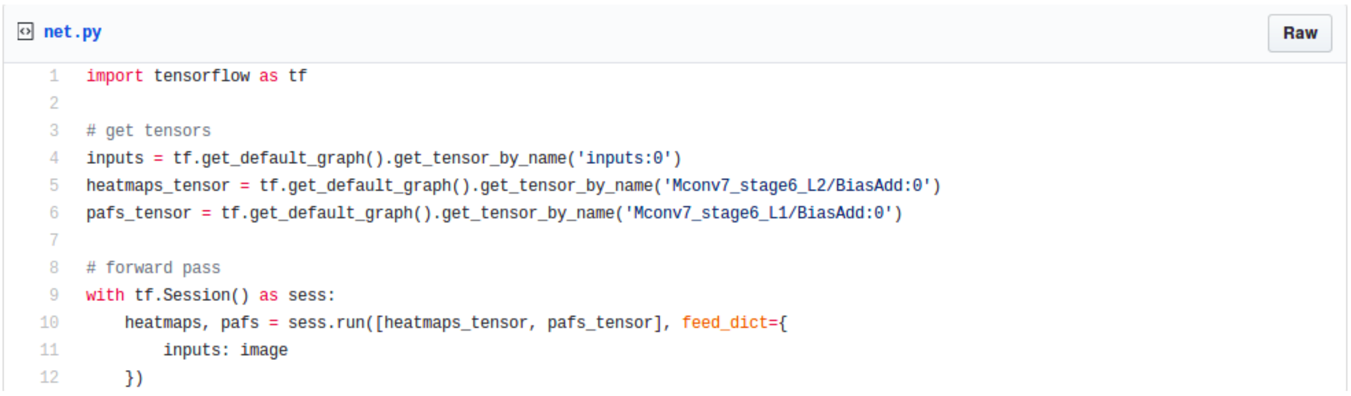
- **PAFs** (Part Affinity Fields) are matrices that give information about the position and orientation of pairs. They come in couples: for each part we have a PAF in the 'x' direction and a PAF in the 'y' direction. There are 38 PAFs associated with each one of the pairs and indexed as we showed in the drawing of the skeletons. We will **associate couples of parts into pairs** thanks to these 38 matrices.





Note how PAF sign is different for opposite senses

I hope someday Medium will allow to paste one single Gist file without including the others. Meanwhile, here's a screenshot of the code that runs this stage of the pipeline.

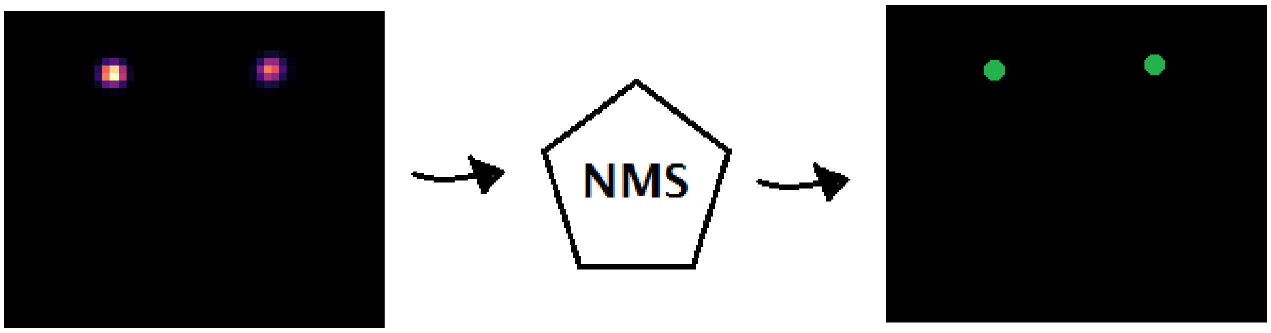


```python
import tensorflow as tf

# get tensors
inputs = tf.get_default_graph().get_tensor_by_name('inputs:0')
heatmaps_tensor = tf.get_default_graph().get_tensor_by_name('Mconv7_stage6_L2/BiasAdd:0')
pafs_tensor = tf.get_default_graph().get_tensor_by_name('Mconv7_stage6_L1/BiasAdd:0')

# forward pass
with tf.Session() as sess:
    heatmaps, pafs = sess.run([heatmaps_tensor, pafs_tensor], feed_dict={
        inputs: image
    })
```

Non Maximum Suppression

Next step is detecting the parts in the image. Heatmaps are *cool* (pun intended), but we should transform confidence into certainty if we want to move forward.

We need to extract parts locations out of a heatmap. Or, in other words, we need to extract points out of a function. What are these points? The local maximums.



Super-illustrative diagram

We apply a non-maximum suppression (NMS) algorithm to get those peaks.

1. Start in the first pixel of the heatmap.
2. Surround the pixel with a window of side 5 and find the maximum value in that area.
3. Substitute the value of the center pixel for that maximum
4. Slide the window one pixel and repeat these steps after we've covered the entire heatmap.
5. Compare the result with the original heatmap. Those pixels staying with same value are the peaks we are looking for. *Suppress* the other pixels setting them with a value of 0.

After all the process, the non-zero pixels denote the location of the part candidates.

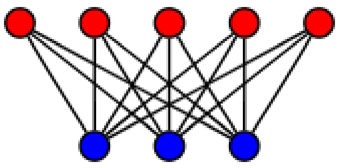
nms.py Raw

```python
from scipy.ndimage.filters import maximum_filter

part_candidates = heatmap*(heatmap == maximum_filter(heatmap, footprint=np.ones((window_size, window_size))))
```

Bipartite graph

Now that we have found the candidates for each one of the body parts, we need to connect them to form pairs. And this is where graph theory steps into.

Say that, for a given image, we have located a set of neck candidates and a set of right hip candidates. For each neck there is a possible association, or *connection candidate*, with each one of the right hips. So, what we have, is a complete bipartite graph, where the vertices are the part candidates, and the edges are the connection candidates.

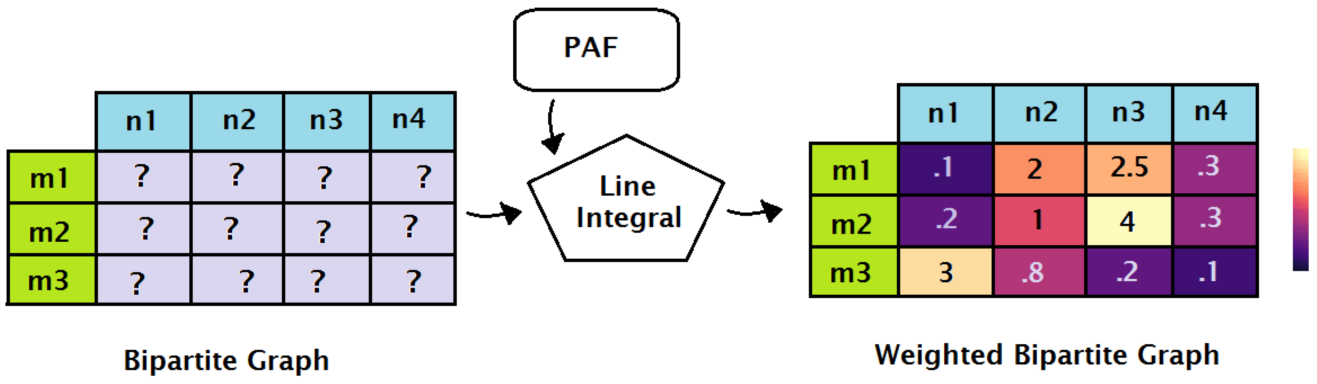


Which neck connects with which right hip?

How can we find the right connections? Finding the best matching between vertices of a bipartite graph is a well-known problem in graph theory known as the assignment problem. In order to solve it, each edge on the graph should have a weight.

Let's put some weights on those edges.

Line Integral



This is where PAFs enter the pipeline. We will compute the line integral along the segment connecting each couple of part candidates, over the corresponding PAFs (x and y) for that pair. As Wikipedia says, a line integral measures the effect of a given field (in our case, the Part Affinity Fields) along a given curve (in our case, the possible connections between part candidates). As usual, Khan Academy explains it very well with an easy example.

$$\int_{y_1}^{y_2}\int_{x_1}^{x_2}\begin{bmatrix}\mathbf{PAF}_{\mathbf{x}}(x,y)\\ \mathbf{PAF}_{\mathbf{y}}(x,y)\end{bmatrix}\cdot\begin{bmatrix}\mathbf{v}_{\mathbf{x}}\\ \mathbf{v}_{\mathbf{y}}\end{bmatrix}\,dx\,dy$$

Our beautiful line integral

The line integral will give each connection a score, that will be saved in a weighted bipartite graph and will allow us to solve the assignment problem.

Note: in the code, we approximate the line integral by taking samples of the dot product and sum them all.

lineintegral.py Raw

```python
import math
import numpy as np

# building the vectors
dx, dy = x2 - x1, y2 - y1
normVec = math.sqrt(dx ** 2 + dy ** 2)
vx, vy = dx/normVec, dy/normVec

# sampling
num_samples = 10
xs = np.arange(x1, x2, dx/num_samples).astype(np.int8)
ys = np.arange(y1, y2, dy/num_samples).astype(np.int8)

# evaluating on the field
pafXs = pafX[ys, xs]
```

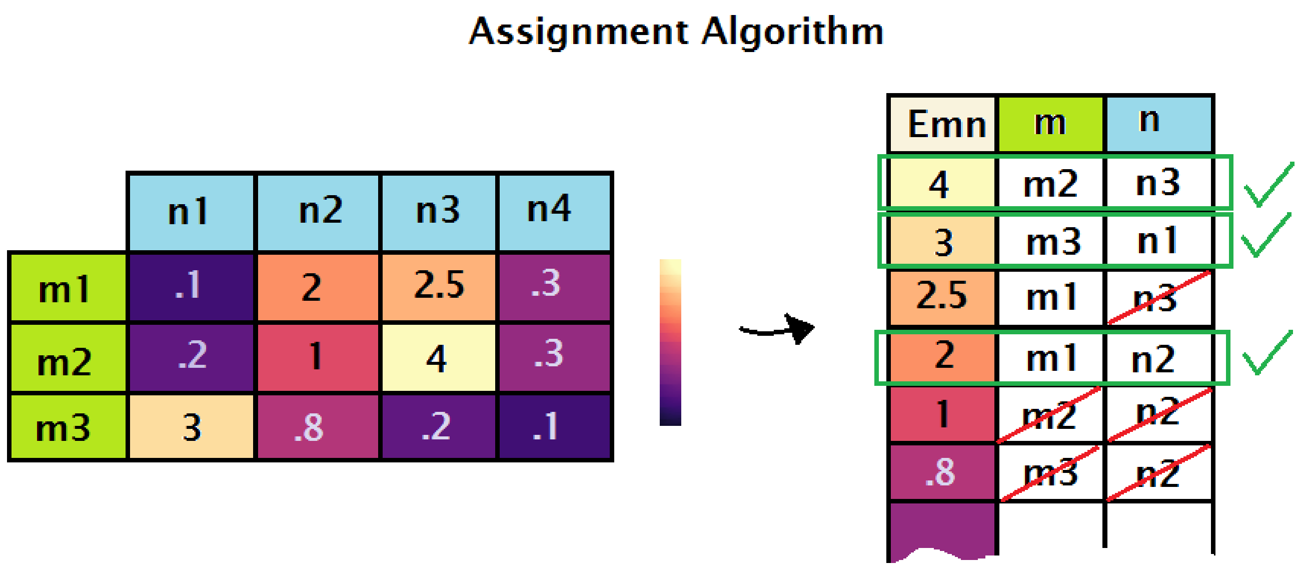
```
pafYs = pafY[ys, xs]

# integral
score = sum(pafXs * vx + pafYs * vy) / num_samples
```

Assignment

The weighted bipartite graph shows all possible connections between candidates of two parts, and holds a score for every connection. The mission now is to find the connections that maximize the total score, that is, solving the assignment problem.

There are plenty of good solutions to this problem, but we are going to pick the most intuitive one:

1. Sort each possible connection by its score.
2. The connection with the highest score is indeed a final connection.
3. Move to next possible connection. If no parts of this connection have been assigned to a final connection before, this is a final connection.
4. Repeat the step 3 until we are done.



	n1	n2	n3	n4
m1	.1	2	2.5	.3
m2	.2	1	4	.3
m3	3	.8	.2	.1

Emn	m	n	
4	m2	n3	✓
3	m3	n1	✓
2.5	m1	~~n3~~	
2	m1	n2	✓
1	~~m2~~	~~n2~~	
.8	~~m3~~	~~n2~~	

As you see, there may be part candidates that will finally not fit into a pair.

assignment.py Raw

```
connection = []
used_idx1, used_idx2 = [], []
# sort possible connections by score, from maximum to minimum
for conn_candidate in sorted(connection_temp, key=lambda x: x['score'], reverse=True):
    # check not connected
    if conn_candidate['idx'][0] in used_idx1 or conn_candidate['idx'][1] in used_idx2:
        continue
        continue
    connection.append(conn_candidate)
    used_idx1.append(conn_candidate['idx'][0])
    used_idx2.append(conn_candidate['idx'][1])
```

Merging

The final step is to transform these detected connections into the final skeletons. We will start with a naive assumption: at first, every connection belongs to a different human. This way, we have the same number of humans as connections we have detected.

Let *Humans* be a collection of sets {H1, H2, …, Hk}. Each one of these sets — that is, each human — contains, at first, two parts (a pair). And let's describe a part as a tuple of an index, a coordinate in the 'x' direction and a coordinate in the 'y' direction.

$$
\begin{aligned}
&\mathrm{Humans} = \{\mathrm{H_1}, \mathrm{H_2}, \ldots, \mathrm{H_k}\} \\
&\texttt{where} \\
&\qquad k := \text{number of final connections} \\
&\qquad \mathrm{H_i} = \{(\mathrm{m_{idx}}, \mathrm{m_x}, \mathrm{m_y}), (\mathrm{n_{idx}}, \mathrm{n_x}, \mathrm{n_y})\}
\end{aligned}
$$

At first, each human is a set of two parts

Here comes the merging: if humans H1 and H2 share a part index with the same coordinates, they are sharing the same part! H1 and H2 are, therefore, the same humans. So we merge both sets into H1 and remove H2.

```
if   $H_1 \cap H_2 \neq \emptyset$
then
    $H_1 = H_1 \cup H_2$
    delete$(H_2)$
```

Merging algorithm

We continue to do this for every couple of humans until no couple share a part.

Note: in the code, for some sensible reasons, we first define a human as a set of connections, not as set of parts. After all the merging is done, we finally describe a human as a set of parts.

merging.py Raw

```python
from collections import defaultdict
import itertools

no_merge_cache = defaultdict(list)
empty_set = set()

while True:
    is_merged = False

    for h1, h2 in itertools.combinations(connections_by_human.keys(), 2):
        for c1, c2 in itertools.product(connections_by_human[h1], connections_by_human[h2]):
            # if two humans share a part (same part idx and coordinates), merge those humans
            if set(c1['partCoordsAndIdx']) & set(c2['partCoordsAndIdx']) != empty_set:
                is_merged = True
                # extend human1 connections with human2 connections
                connections_by_human[h1].extend(connections_by_human[h2])
                connections_by_human.pop(h2) # delete human2
                break

    if not is_merged: # if no more mergings are possible, then break
        break

# describe humans as a set of parts, not as a set of connections
humans = [human_conns_to_human_parts(human_conns) for human_conns in connections_by_human.values()]
```

Output

Finally what you get is a collection of human sets, where each human is a set of parts, where each part contains its index, its relative coordinates and its score.

Try it

You can try OpenPose on your computer just downloading the code and the model. You should have Python 3 with TensorFlow, OpenCV, Numpy, Scipy and Matplotlib installed.

```
$ git clone
https://gist.github.com/alesolano/b073d8ec9603246f766f9f15d002f4f4
openpose_pipeline
$ cd openpose_pipeline
$ mkdir models
$ cd models
$ wget
https://www.dropbox.com/s/2dw1oz9l9hi9avg/optimized_openpose.pb
$ cd ..
$ python inference.py --imgpath /path/to/your/img
```

OpenPose for Robotics

I'm currently involved in a project at University of Málaga in which we try to make an assistant robot approach smoothly to a person, finally facing her. We need to know where the person is located and where she's looking. We are very interested in OpenPose as it could give us accurate position and orientation info, even if people are far from the camera.

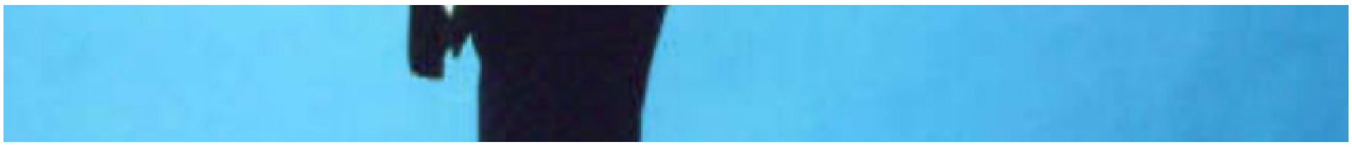
Time performance is crucial in robotics. And, so far, I have not been able to run OpenPose at a decent speed: an image lasts 10 seconds to pass through the neural network. I've been running this Python version of the pipeline on my laptop, though we have acquired a Jetson TX2 recently. I will be exploring all the possibilities to run OpenPose faster.

So, next posts on this series may be about TensorFlow C++, OpenPose performance in day-life environments (e.g., how near should a person be to the camera in order to be detected by OpenPose), or maybe about something totally different. We'll see.

And, of course, if you're also working with OpenPose or are interested in doing so, get in touch with me!

Thanks for reading!!

P.S.: I'm still unable to fully predict Chuck Norris' movements. Those kicks are just too much.



TensorFlow Openpose Deep Learning Robotics Virtual Reality

About Help Legal