1 Paper review

1.1 Paper Information

- **Title:** FlowNet3D: Learning Scene Flow in 3D Point Clouds
- **Authors:** Xingyu Liu, Charles R. Qi, Leonidas J. Guibas
- **Link:** https://arxiv.org/abs/1806.01411
- **Tags:** Neural Network, Performance, Covariate Shift, Regularization
- **Year:** 2018(v1), 2019(v3 - latest)

1.2 Summary

1.2.1 General information

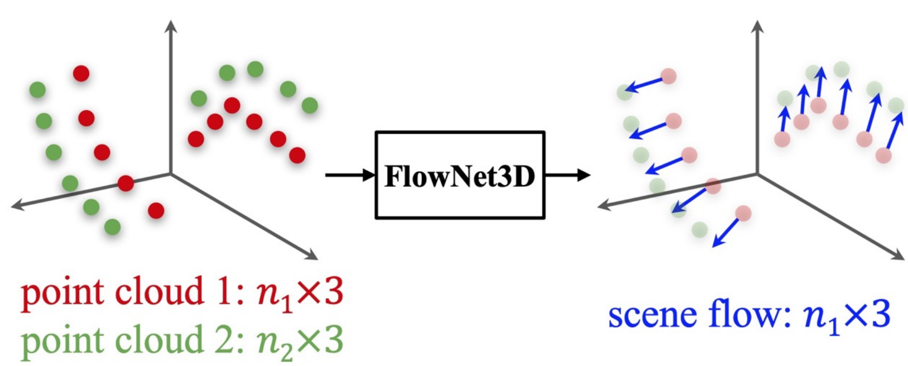
Scene flow is the dense or semi-dense 3D motion field of a scene that moves completely of partially with respect to a camera. If project it onto some 2D plane we would obtain optical flow.

In this work authors focus on learning scene flow directly from point clouds. That's interesting, as long as long as most previous methods focus on stereo and RGB-D images as input.

The potential applications of scene flow are numerous. In robotics, it can be used for autonomous navigation and/or manipulation in dynamic environments where the motion of the surrounding objects needs to be predicted. On the other hand, it could be employed for human-robot or human-computer interaction, as well as for virtual and augmented reality.

1.2.2 How it works

Authors are proposing a new deep neural network called *FlowNet3D* that learns scene flow in 3D point clouds end-to-end:

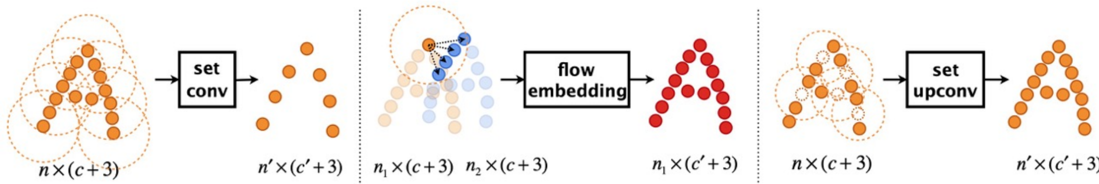


As we can see from the picture, this networks takes two consecutive frames (point clouds) as input and estimates a translational flow vector to show for each point it's motion from first frame to the second.

In this paper authors introduce two new learning layers on point clouds: a *flow embedding* layer and a *set upconv* layer.

So, whole network itself consists of three trainable key modules:

- *set conv* layer to learn deep point cloud features;
- *flow embedding* layer to learn geometric relations between two point clouds to infer motions;
- *set upconv* layer to up-sample and propagate point features in a learnable way.



Output of the model is $\mathbb{R}^3$ predicted scene flow which is produced by the final linear flow regression layer.

1.2.3 Results

The final model was trained and tested using synthetic dataset (FlyingThings3D). The 3D end point error (EPE) and flow estimation accuracy (ACC) were used as evaluation metrics. It showed quite good results in compare to baseline models:

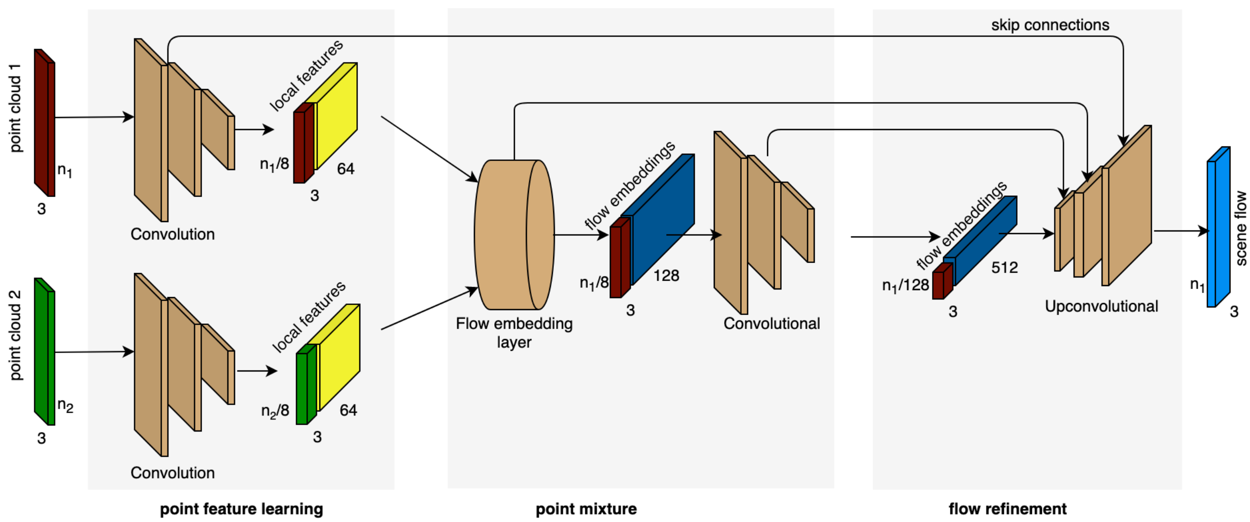
Method	Input	EPE	ACC (0.05)	ACC (0.1)
FlowNet-C [8]	depth	0.7887	0.20%	1.49%
	RGBD	0.7836	0.25%	1.74%
ICP [3]	points	0.5019	7.62%	21.98%
EM-baseline (ours)	points	0.5807	2.64%	12.21%
LM-baseline (ours)	points	0.7876	0.27%	1.83%
DM-baseline (ours)	points	0.3401	4.87%	21.01%
FlowNet3D (ours)	points	**0.1694**	**25.37%**	**57.85%**

However this model was trained on the synthetic dataset, it was also showed that this model can be directly applied to detect scene flow in point clouds from real data. To prove that LiDAR scans from the KITTI benchmark were used:

Method	Input	EPE (meters)	outliers (0.3m or 5%)	KITTI ranking
LDOF [4]	RGB-D	0.498	12.61%	21
OSF [16]	RGB-D	0.394	8.25%	9
PRSM [30]	RGB-D	0.327	6.06%	3
	RGB stereo	0.729	6.40%	
Dewan et al. [7]	points	0.587	71.74%	-
ICP (global)	points	0.385	42.38%	-
ICP (segmentation)	points	0.215	13.38%	-
FlowNet3D (ours)	points	**0.122**	**5.61%**	-

So, to sum up, results of this paper give us model, which showed its competitive or better results to various baselines and prior arts. It also can be used with real data datasets, even if was trained using only synthetic dataset.

2 Network visualization



3 Experimental results

3.1 Batch normalization

First changes I made - adding two batch normalization layers after each of convolution layer. So network looks in next way:

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
```

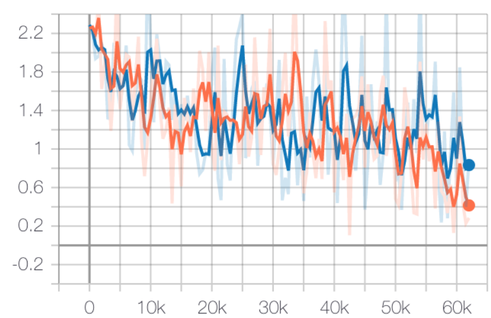
```
  (bn1): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

This trick increased accuracy by 2%(from 63 to 65). Here are results by each category:

```
Accuracy of plane : 66 %
Accuracy of   car : 72 %
Accuracy of  bird : 49 %
Accuracy of   cat : 41 %
Accuracy of  deer : 56 %
Accuracy of   dog : 53 %
Accuracy of  frog : 72 %
Accuracy of horse : 70 %
Accuracy of  ship : 76 %
Accuracy of truck : 72 %
```

Also it took 1 more minute to train it(blue): 6min 10sec with 5min 10sec for default model(orange). Running loss behaves almost same as for default model:

RunningLoss
tag: Train/RunningLoss



3.2 Filters amount

After that I increased amount of filters in conv layers. From 5 to 10 for first layer and from 16 to 20 for second. So model is:

```
Net(
  (conv1): Conv2d(3, 10, kernel_size=(5, 5), stride=(1, 1))
  (bn1): BatchNorm2d(10, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (bn2): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
  (fc1): Linear(in_features=500, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

This trick increased training time by 35 seconds (up to 6min 45sec). It also increased accuracy by 3% (from 65 to 68). Here are results by each category:

```
Accuracy of plane : 76 %
Accuracy of   car : 80 %
Accuracy of  bird : 56 %
Accuracy of   cat : 43 %
Accuracy of  deer : 62 %
Accuracy of   dog : 55 %
Accuracy of  frog : 83 %
Accuracy of horse : 73 %
Accuracy of  ship : 79 %
Accuracy of truck : 75 %
```

If compare with previous version, we got the best improvement for 'frog' category (+11%).