- End-to-end Lane Detection through Differentiable Least-Squares Fitting
- Towards End-to-End Lane Detection an Instance Segmentation Approach
- ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation
- Local Importance-based Pooling

周至公 11.11

End-to-end Lane Detection through Differentiable Least-Squares Fitting

- Pre-defined number of lanes
- update the weights *wi* while keeping the coordinates (*xi; yi*) fixed
- Why end-to-end?

  the problem with such a two-step approach (first get mask then fit line) is that the parameters of the network are not optimized for the true task of interest (estimating the lane curvature parameters) but for a proxy task (segmenting the lane markings), resulting in sub-optimal performance.

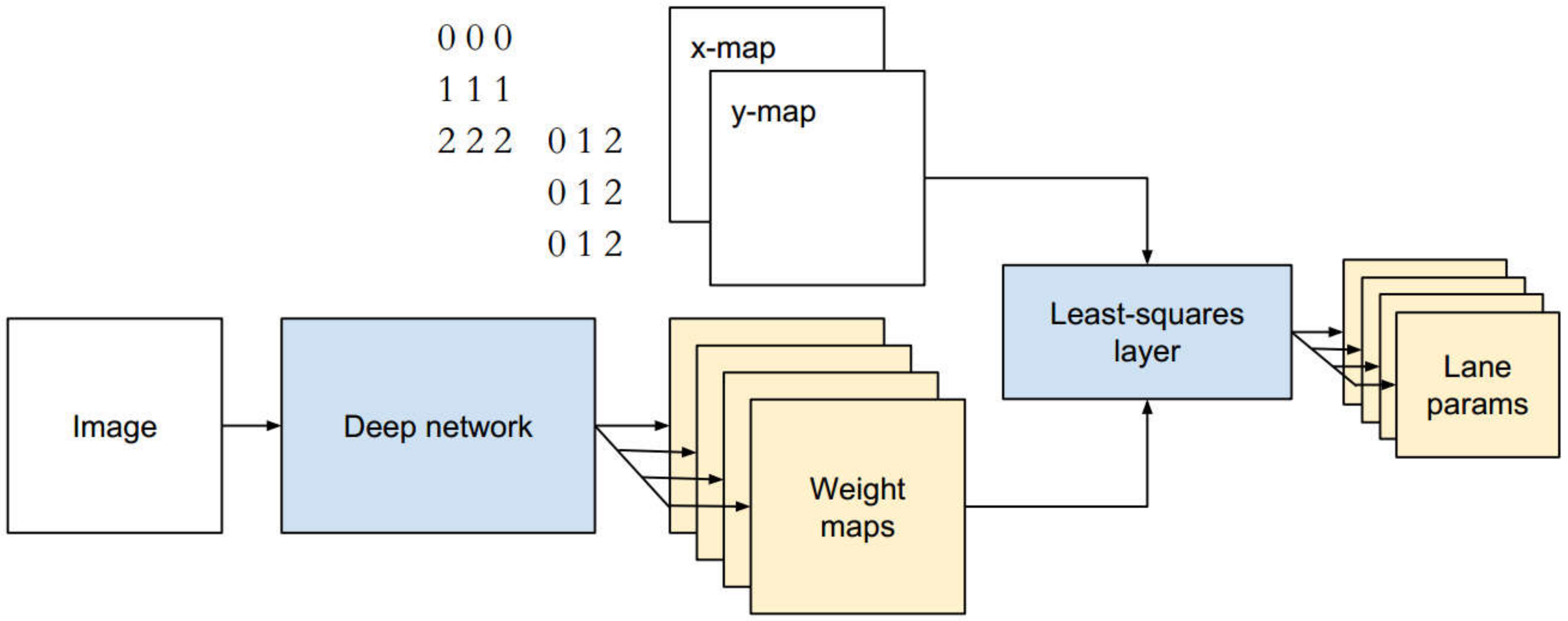


Figure 1. Overview of the architecture. In this example the network produces four weight maps, each corresponding to one of four lane lines for which the parameters are estimated.

- 视角转换：the list of coordinates(*xi; yi; wi*) * a homography matrix *H*
- in order to restrict the weight maps to be nonnegative, the output of the network is **squared**
- multiple weight maps: **one for each lane line** that needs to be detected.

Loss function

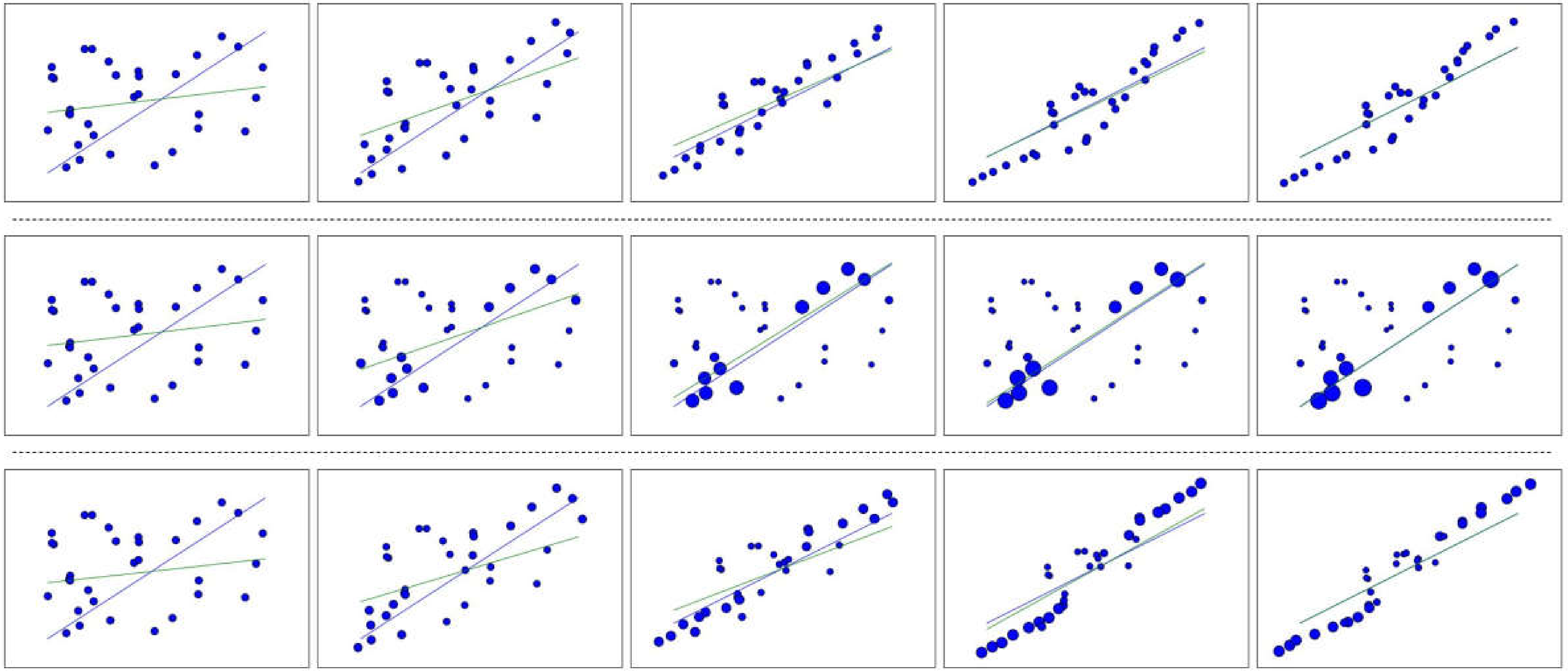


Figure 3. Illustration of differentiable weighted least-squares line fitting on a synthetic example, explained in Section 3.1.

Loss function

with $X \in \mathbb{R}^{m\times n}$, $\beta \in \mathbb{R}^{n\times 1}$, and $Y \in \mathbb{R}^{m\times 1}$:

$$X = \begin{pmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & \cdots & x_m^{n-1} \end{pmatrix}, \beta = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix}, Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

$$WX\beta = WY.$$

By defining $X' = WX$ and $Y' = WY$ with

$$W = \mathrm{diag}(\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}) = \begin{pmatrix} w_1 & 0 & \cdots & 0 \\ 0 & w_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & w_m \end{pmatrix}$$

$$L = \int_0^t (y_\beta(x) - y_{\hat{\beta}}(x))^2 dx$$

$$L = \int_0^t (y_\beta(x) - y_{\hat{\beta}}(x))^2 dx \tag{5}$$

For a straight line $y = \beta_0 + \beta_1 x$, this results in:

$$L = \Delta\beta_0^2 t + \Delta\beta_1 \Delta\beta_0 t^2 + \frac{\Delta\beta_1^2 t^3}{3}, \tag{6}$$

where $\Delta\beta_i = \beta_i - \hat{\beta}_i$. For a parabolic curve $y = \beta_0 + \beta_1 x + \beta_2 x^2$ it gives:

$$L = \frac{\Delta\beta_2^2 t^5}{5} + \frac{2\Delta\beta_2\Delta\beta_1 t^4}{4} + \frac{(\Delta\beta_1^2 + 2\Delta\beta_2\Delta\beta_0)t^3}{3} + \frac{2\Delta\beta_1\Delta\beta_0 t^2}{2} + \Delta\beta_0^2 t. \tag{7}$$

Loss function

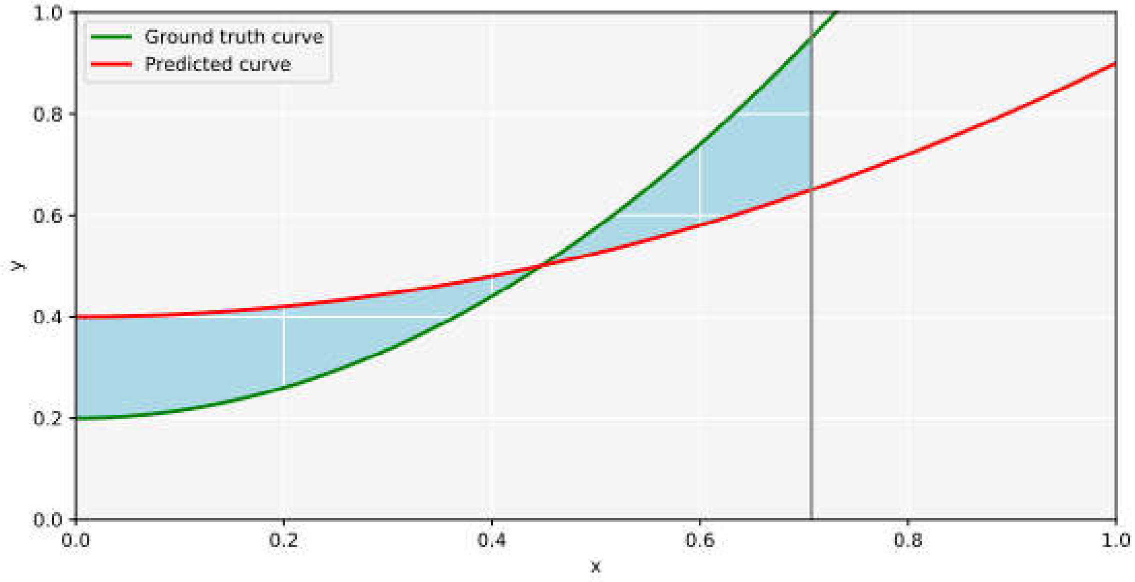


Figure 2. The geometric loss minimizes the (squared) area between the predicted curve and ground truth curve up to a point t.

Different sensitivities: a small error in one parameter value might have a larger effect on the curve shape than an error of the same magnitude in another parameter

The loss is thus only backpropagated to the coordinate *weights*,

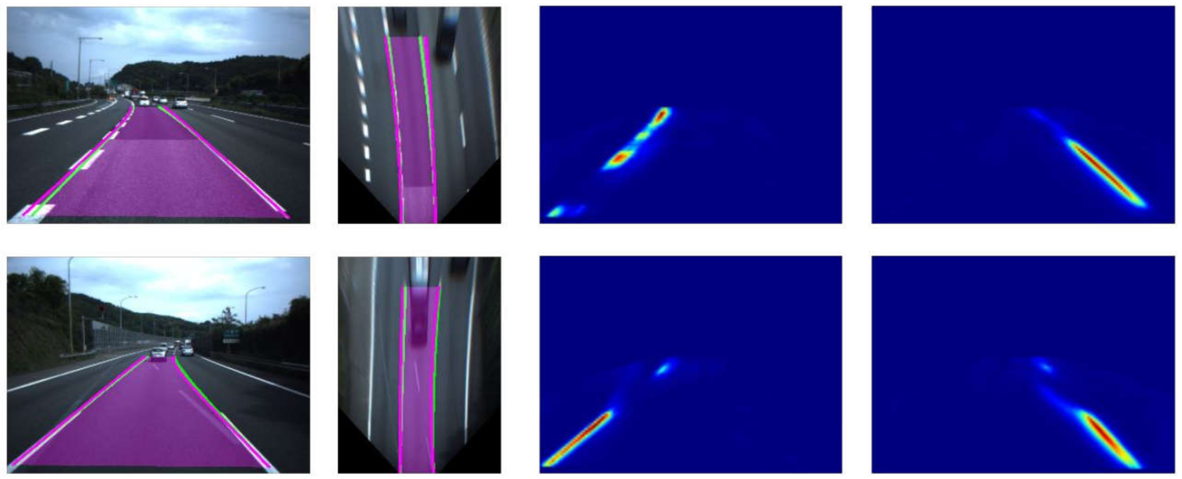


Figure 5. Qualitative results on the lane detection task. From left to right: input image with overlayed ground truth (green) and predicted lane lines (purple), ortho-view of the scene in which the loss is calculated (see text), coordinate weight maps corresponding to left and right lane lines. The network learns to handle the large variance in lane markings and challenging conditions like the faded markings in the bottom row that are correctly ignored.

End-to-end 比 two-step收敛慢：没有显示的像素监督信息（一种结合方式：先在two-step pre-train）

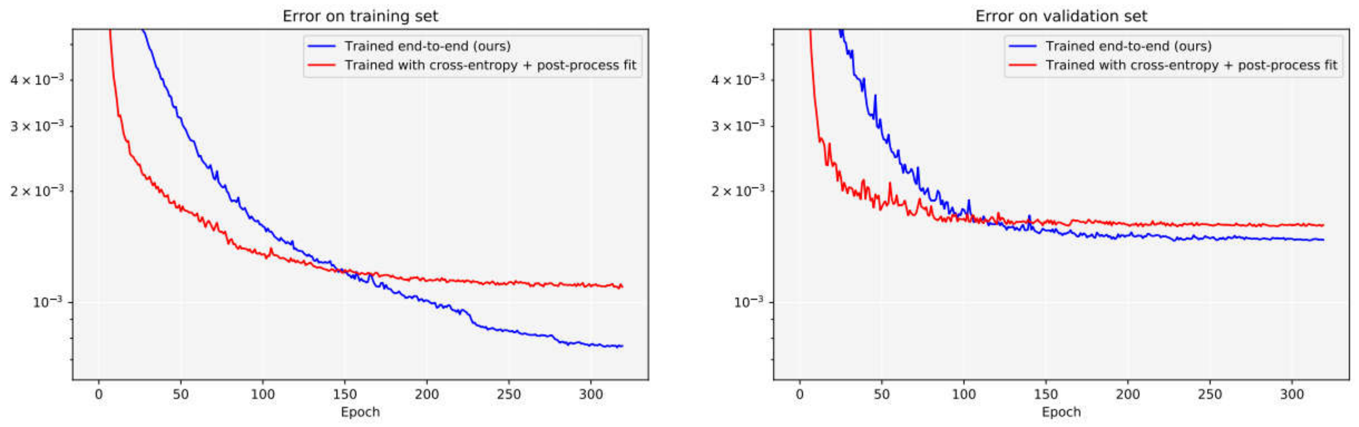


Figure 4. Convergence of the error during training (left) on the training set and (right) on the validation set. The error curve of our method is indicated in blue, the error curve of the method trained with a segmentation-like cross-entropy loss and fitting as a post-processing step is indicated in red.

Towards End-to-End Lane Detection an Instance Segmentation Approach

- cast the lane detection problem as an instance segmentation problem
- apply a learned perspective transformation

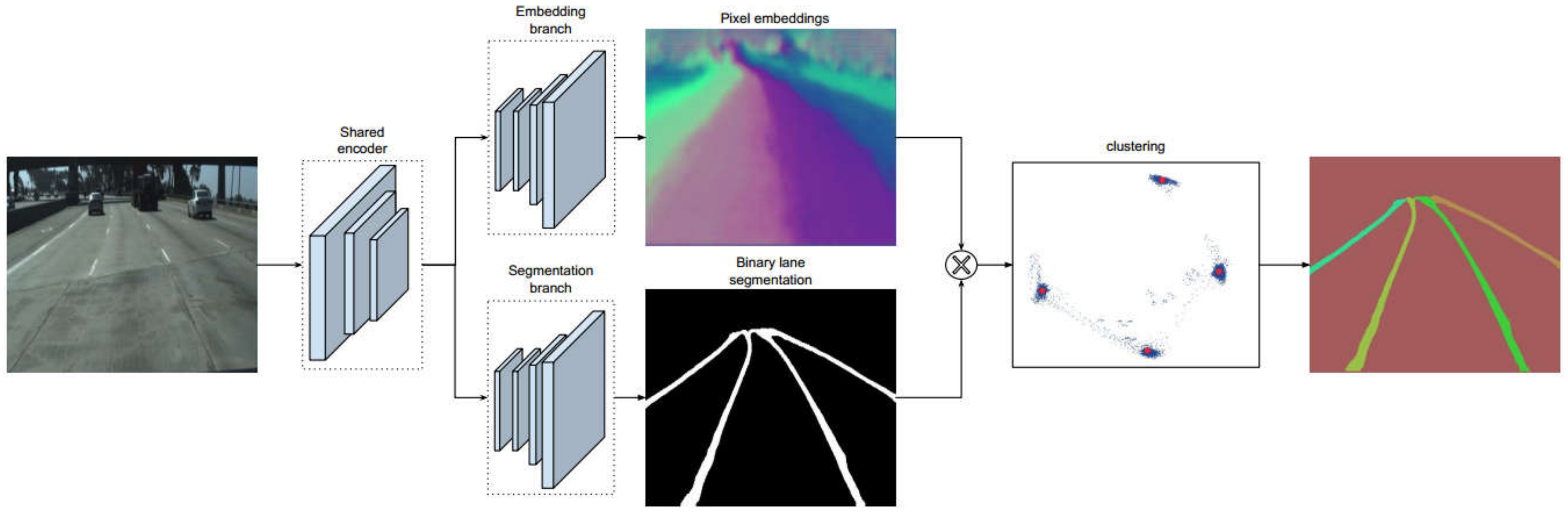


Fig. 2. LaneNet architecture. It consists of two branches. The segmentation branch (bottom) is trained to produce a binary lane mask. The embedding branch (top) generates an N-dimensional embedding per lane pixel, so that embeddings from the same lane are close together and those from different lanes are far in the manifold. For simplicity we show a 2-dimensional embedding per pixel, which is visualized both as a color map (all pixels) and as points (only lane pixels) in a xy grid. After masking out the background pixels using the binary segmentation map from the segmentation branch, the lane embeddings (blue dots) are clustered together and assigned to their cluster centers (red dots).

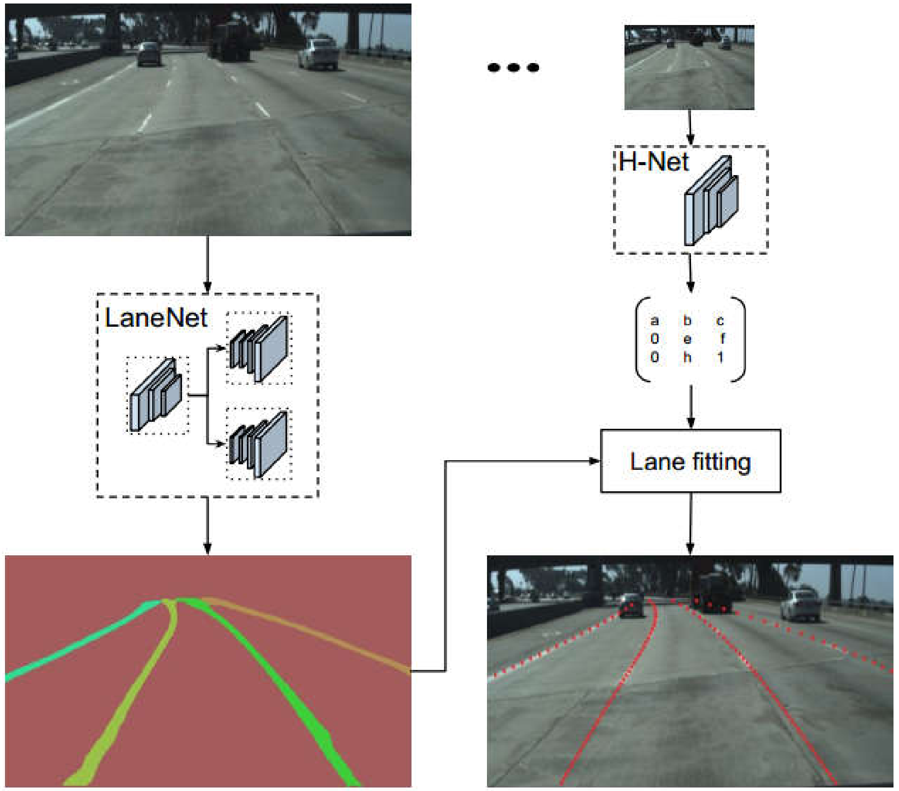


Fig. 1. System overview. Given an input image, LaneNet outputs a lane instance map, by labeling each lane pixel with a lane id. Next, the lane pixels are transformed using the transformation matrix, outputted by H-Net which learns a perspective transformation conditioned on the input image. For each lane a 3rd order polynomial is fitted and the lanes are reprojected onto the image.

Clustering

first use mean shift to shift closer to the cluster center and then do the thresholding $\delta d > 6\delta v$

$$\begin{cases} L_{var} = \frac{1}{C}\sum_{c=1}^{C}\frac{1}{N_c}\sum_{i=1}^{N_c}[\|\mu_c - x_i\| - \delta_{\mathrm{v}}]_+^2 \\ L_{dist} = \frac{1}{C(C-1)}\sum_{c_A=1}^{C}\sum_{c_B=1, c_A\neq c_B}^{C}[\delta_{\mathrm{d}} - \|\mu_{c_A} - \mu_{c_B}\|]_+^2 \end{cases}$$

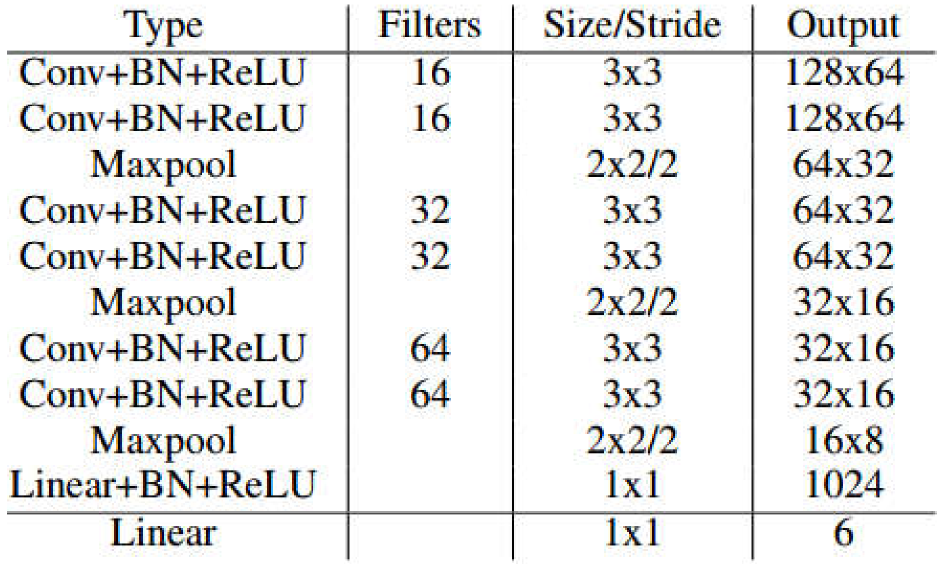
network architecture

lanenet only shares the first two stages (1 and 2) between the two branches, leaving stage 3 of the enet encoder and the full ENet decoder as the backbone of each separate branch

- segmentation branch outputs a one channel image (binary segmentation),
- embedding branch outputs a n-channel image,
- each branch’s loss term is equally weighted

H-Net

$$H = \begin{bmatrix} a & b & c \\ 0 & d & e \\ 0 & f & 1 \end{bmatrix}$$



Type	Filters	Size/Stride	Output
Conv+BN+ReLU	16	3x3	128x64
Conv+BN+ReLU	16	3x3	128x64
Maxpool		2x2/2	64x32
Conv+BN+ReLU	32	3x3	64x32
Conv+BN+ReLU	32	3x3	64x32
Maxpool		2x2/2	32x16
Conv+BN+ReLU	64	3x3	32x16
Conv+BN+ReLU	64	3x3	32x16
Maxpool		2x2/2	16x8
Linear+BN+ReLU		1x1	1024
Linear		1x1	6

自适应路面变化
H has 6 degrees of freedom
optimized end-to-end.
The zeros are placed to enforce the constraint that horizontal lines remain horizontal under the transformation. (only transform y-axis)

Loss

fit a polynomial *f(y0)* = *αy02* + *βy0* + *γ* using the least squares closed-form solution

only update x-axis

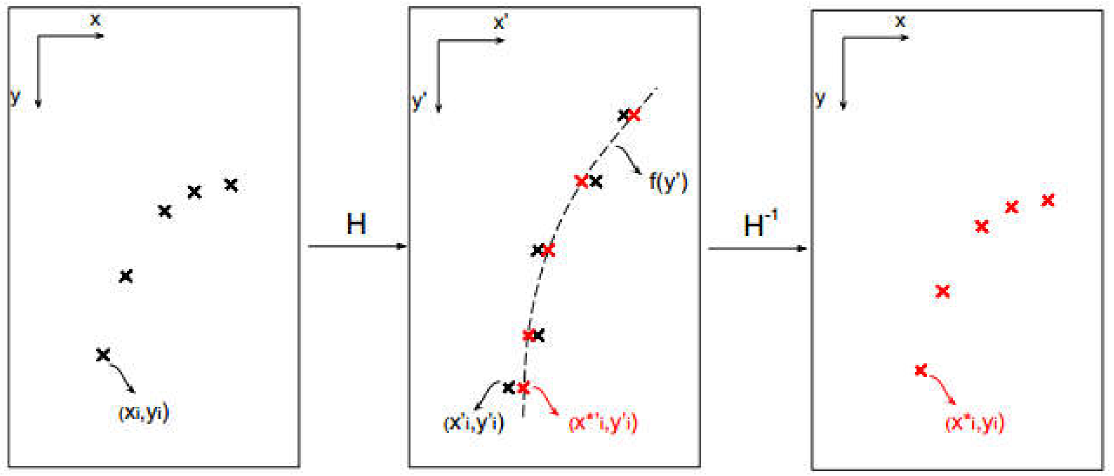
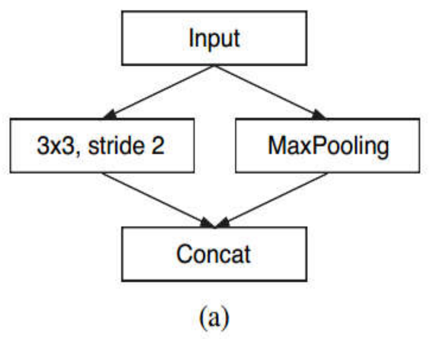


Fig. 3. Curve fitting. *Left:* The lane points are transformed using the matrix H generated by H-Net. *Mid:* A line is fitted through the transformed points and the curve is evaluated at different heights (red points). *Right:* The evaluated points are transformed back to the original image space.

$$\mathbf{w} = (\mathbf{Y}^T\mathbf{Y})^{-1}\mathbf{Y}^T\mathbf{x}'$$

$$Loss = \frac{1}{N}\sum_{i=1,N}(x_i^* - x_i)^2$$

ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation



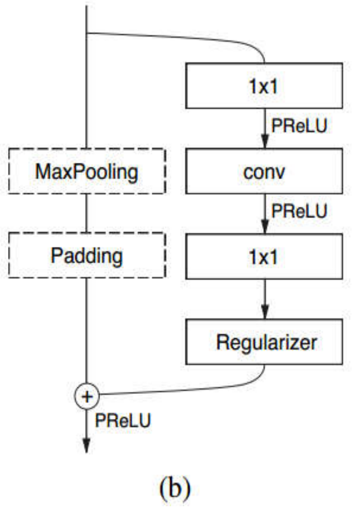


Figure 2: (a) ENet initial block. MaxPooling is performed with non-overlapping 2×2 windows, and the convolution has 13 filters, which sums up to 16 feature maps after concatenation. This is heavily inspired by [28]. (b) ENet bottleneck module. conv is either a regular, dilated, or full convolution (also known as deconvolution) with 3×3 filters, or a 5×5 convolution decomposed into two asymmetric ones.

Name	Type	Output size
initial		$16 \times 256 \times 256$
bottleneck1.0	downsampling	$64 \times 128 \times 128$
$4\times$ bottleneck1.x		$64 \times 128 \times 128$
bottleneck2.0	downsampling	$128 \times 64 \times 64$
bottleneck2.1		$128 \times 64 \times 64$
bottleneck2.2	dilated 2	$128 \times 64 \times 64$
bottleneck2.3	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.4	dilated 4	$128 \times 64 \times 64$
bottleneck2.5		$128 \times 64 \times 64$
bottleneck2.6	dilated 8	$128 \times 64 \times 64$
bottleneck2.7	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.8	dilated 16	$128 \times 64 \times 64$
Repeat section 2, without bottleneck2.0		
bottleneck4.0	upsampling	$64 \times 128 \times 128$
bottleneck4.1		$64 \times 128 \times 128$
bottleneck4.2		$64 \times 128 \times 128$
bottleneck5.0	upsampling	$16 \times 256 \times 256$
bottleneck5.1		$16 \times 256 \times 256$
fullconv		$C \times 512 \times 512$

- Feature map resolution
- Downsampling drawbacks:1、丢失空间信息 2、需要upsampling复原，增大model
- *FCN* adds the feature maps produced by encoder
- *SegNet* saves indices of elements chosen in max pooling layers, and using them to produce sparse upsampled maps in the decoder.
- dilated convolutions for a bigger receptive field
- Early downsampling
- our intuition is that the initial network layers should <u>not directly contribute </u>to classification
- <u>only preprocess </u>the input for later portions of the network
- increasing the number of feature maps from 16 to 32 did not improve accuracy on Cityscapes
- Decoder size
- a large encoder, and a small decoder
- encoder要提取feature, decoder仅仅upsample

- Nonlinear operations
- removing most ReLUs in the initial layers of the network improved the results
- our network uses only a couple of layers, and it needs to quickly filter out information

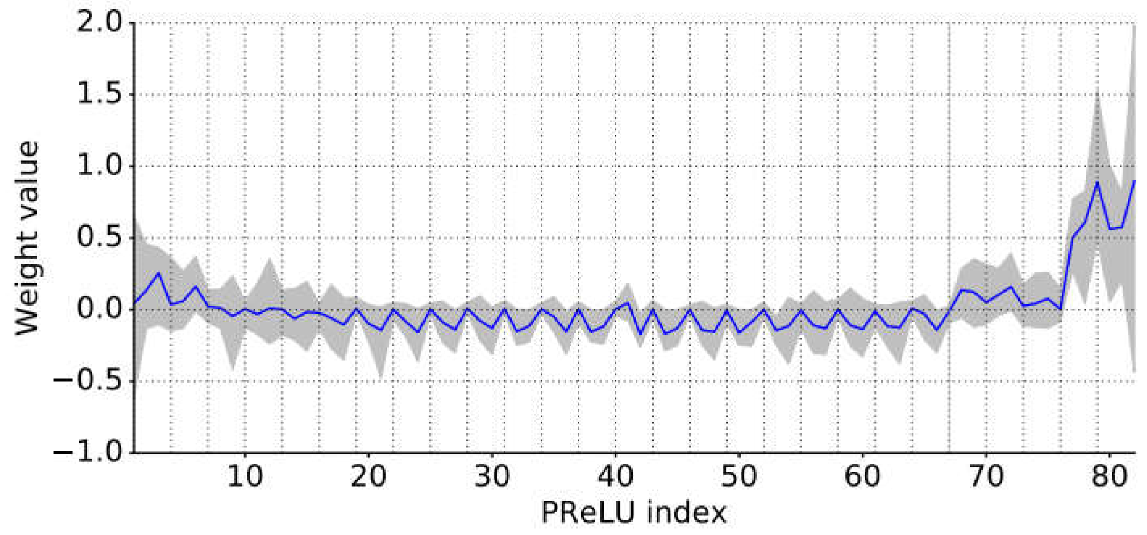


Figure 3: PReLU weight distribution vs network depth. Blue line is the weights mean, while an area between maximum and minimum weight is grayed out. Each vertical dotted line corresponds to a PReLU in the main branch and marks the boundary between each of bottleneck blocks. The gray vertical line at 67th module is placed at encoder-decoder border.

Table 2: Performance comparison.

Model	NVIDIA TX1						NVIDIA Titan X					
	480×320		640×360		1280×720		640×360		1280×720		1920×1080	
	ms	fps	ms	fps	ms	fps	ms	fps	ms	fps	ms	fps
SegNet	757	1.3	1251	0.8	-	-	69	14.6	289	3.5	637	1.6
ENet	47	21.1	69	14.6	262	3.8	7	135.4	21	46.8	46	21.6

Table 3: Hardware requirements. FLOPs are estimated for an input of $3 \times 640 \times 360$.

	GFLOPs	Parameters	Model size (fp16)
SegNet	286.03	29.46M	56.2 MB
ENet	3.83	0.37M	0.7 MB

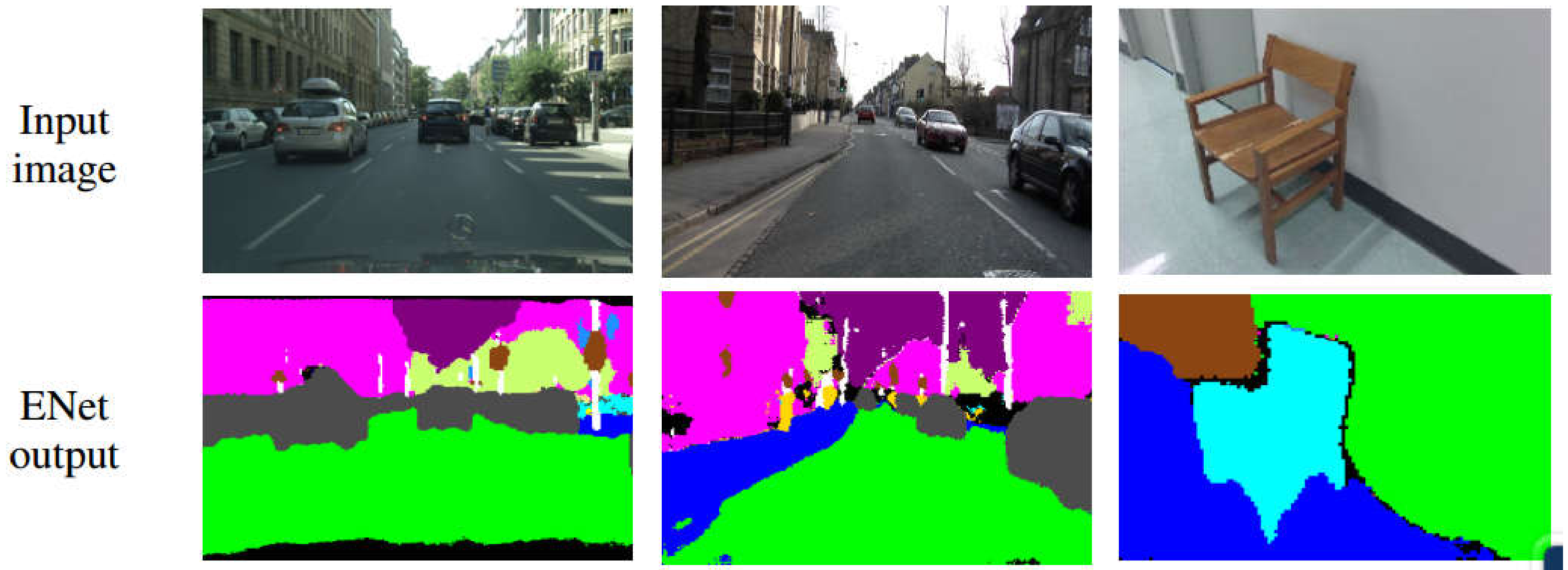
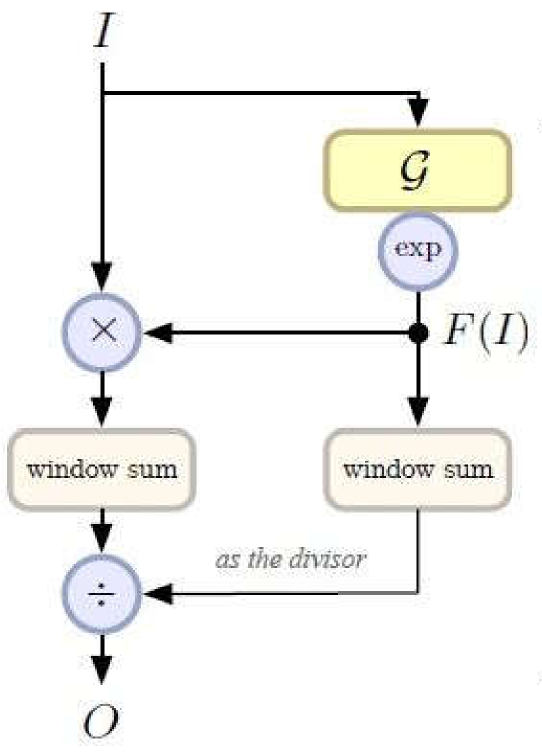


Figure 1: ENet predictions on different datasets (left to right Cityscapes, CamVid, and SUN).

Local Importance-based Pooling

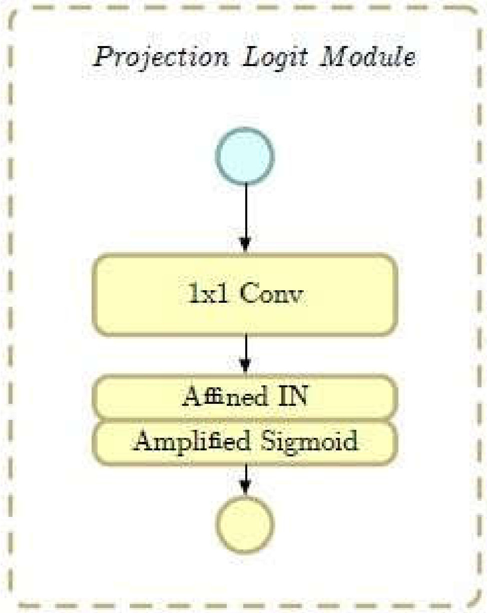


(a)

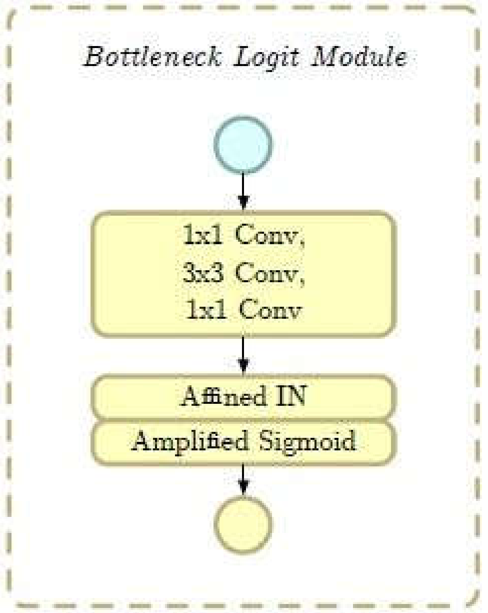
```
import torch
import torch.nn.functional as F

def lip2d(x, logit,
            kernel_size=3,
            stride=2,
            padding=1):
    weight = torch.exp(logit)
    return F.avg_pool2d(x*weight
          , kernel_size, stride,
          padding)/F.avg_pool2d(
          weight, kernel_size,
          stride, padding)
```

(b)



(d)



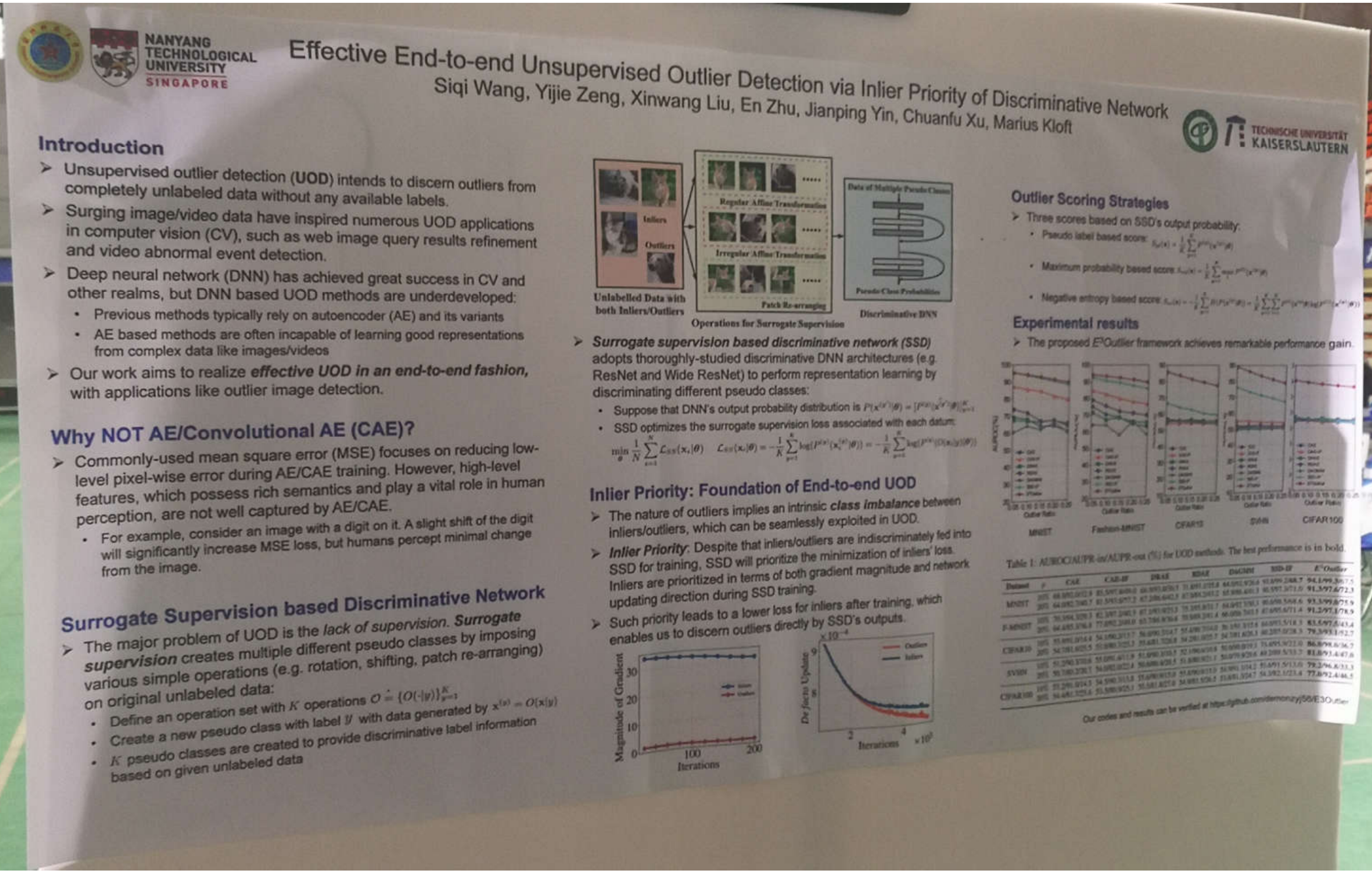
(e)

Layer	Combination of LIP substitutions				
	A	B	C	D	E
Max Pooling	✓				
Res_3	✓	✓			
Res_4	✓	✓	✓		
Res_5	✓	✓	✓	✓	
Top-1	78.19	77.87	77.78	76.92	76.40
Top-5	93.96	93.94	93.81	93.37	93.15
#Params	23.9M	23.8M	23.7M	23.9M	25.6M
FLOPs	5.33G	4.87G	4.26G	4.11G	4.12G

Table 3: Different LIP substitution locations. Combination A stands for the ResNet-50 with full 7 LIPs (LIP-ResNet w Bottleneck-128) and E stands for the vanilla ResNet.

Backbone	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
Faster R-CNN w FPN results						
ResNet-50	37.7	59.3	41.1	21.9	41.5	48.7
LIP-ResNet-50	39.2	61.2	42.5	24.0	43.1	50.3
ResNet-101	39.4	60.7	43.0	22.1	43.6	52.1
LIP-ResNet-101	41.7	63.6	45.6	25.2	45.8	54.0
ResNeXt-101	40.7	62.1	44.5	23.0	44.5	53.6
RetinaNet results						
ResNet-50	36.6	56.6	38.9	19.6	40.3	48.9
LIP-ResNet-50	38.0	58.8	40.5	22.6	41.5	49.9
ResNet-101	38.1	58.1	40.6	20.2	41.8	50.8

Table 5: Faster R-CNN with FPN and RetinaNet with different backbones results on COCO 2017 `val` set. ResNeXt-101 stands for ResNeXt-64x4d-101 backbone in [44].



NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Effective End-to-end Unsupervised Outlier Detection via Inlier Priority of Discriminative Network

Siqi Wang, Yijie Zeng, Xinwang Liu, En Zhu, Jianping Yin, Chuanfu Xu, Marius Kloft

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Introduction

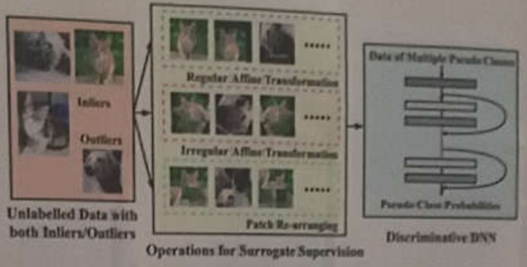
- Unsupervised outlier detection (**UOD**) intends to discern outliers from completely unlabeled data without any available labels.
- Surging image/video data have inspired numerous UOD applications in computer vision (CV), such as web image query results refinement and video abnormal event detection.
- Deep neural network (DNN) has achieved great success in CV and other realms, but DNN based UOD methods are underdeveloped:
  - Previous methods typically rely on autoencoder (AE) and its variants
  - AE based methods are often incapable of learning good representations from complex data like images/videos
- Our work aims to realize ***effective UOD in an end-to-end fashion,*** with applications like outlier image detection.

Why NOT AE/Convolutional AE (CAE)?

- Commonly-used mean square error (MSE) focuses on reducing low-level pixel-wise error during AE/CAE training. However, high-level features, which possess rich semantics and play a vital role in human perception, are not well captured by AE/CAE.
  - For example, consider an image with a digit on it. A slight shift of the digit will significantly increase MSE loss, but humans percept minimal change from the image.

Surrogate Supervision based Discriminative Network

- The major problem of UOD is the *lack of supervision*. ***Surrogate supervision*** creates multiple different pseudo classes by imposing various simple operations (e.g. rotation, shifting, patch re-arranging) on original unlabeled data:
  - Define an operation set with K operations $\mathcal{O} \doteq \{O(\cdot|y)\}_{y=1}^{K}$
  - Create a new pseudo class with label y with data generated by $\mathbf{x}^{(y)} = O(\mathbf{x}|y)$
  - K pseudo classes are created to provide discriminative label information based on given unlabeled data

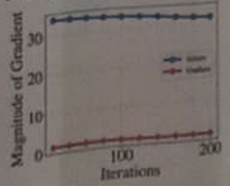

Inliers
Outliers
Unlabelled Data with both Inliers/Outliers
Regular Affine Transformation
Irregular Affine Transformation
Patch Re-arranging
Operations for Surrogate Supervision
Data of Multiple Pseudo Classes
Pseudo Class Probabilities
Discriminative DNN

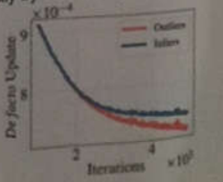
- ***Surrogate supervision based discriminative network (SSD)*** adopts thoroughly-studied discriminative DNN architectures (e.g. ResNet and Wide ResNet) to perform representation learning by discriminating different pseudo classes:
  - Suppose that DNN's output probability distribution is $P(\mathbf{x}^{(y')}|\theta) = [P^{(y)}(\mathbf{x}^{(y')}|\theta)]_{y=1}^{K}$
  - SSD optimizes the surrogate supervision loss associated with each datum:

$$\min_{\theta} \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_{SS}(\mathbf{x}_i|\theta) \quad \mathcal{L}_{SS}(\mathbf{x}_i|\theta) = -\frac{1}{K}\sum_{y=1}^{K}\log(P^{(y)}(\mathbf{x}_i^{(y)}|\theta)) = -\frac{1}{K}\sum_{y=1}^{K}\log(P^{(y)}(O(\mathbf{x}_i|y)|\theta))$$

Inlier Priority: Foundation of End-to-end UOD

- The nature of outliers implies an intrinsic ***class imbalance*** between inliers/outliers, which can be seamlessly exploited in UOD.
- ***Inlier Priority***: Despite that inliers/outliers are indiscriminately fed into SSD for training, SSD will prioritize the minimization of inliers' loss. Inliers are prioritized in terms of both gradient magnitude and network updating direction during SSD training.
- Such priority leads to a lower loss for inliers after training, which enables us to discern outliers directly by SSD's outputs.





Outlier Scoring Strategies

- Three scores based on SSD's output probability:
  - Pseudo label based score: $S_{pl}(\mathbf{x}) = \frac{1}{K}\sum_{y=1}^{K} P^{(y)}(\mathbf{x}^{(y)}|\theta)$
  - Maximum probability based score: $S_{mp}(\mathbf{x}) = \frac{1}{K}\sum_{y=1}^{K} \max_{t} P^{(t)}(\mathbf{x}^{(y)}|\theta)$
  - Negative entropy based score: $S_{ne}(\mathbf{x}) = -\frac{1}{K}\sum_{y=1}^{K} H(P(\mathbf{x}^{(y)}|\theta)) = \frac{1}{K}\sum_{y=1}^{K}\sum_{t=1}^{K} P^{(t)}(\mathbf{x}^{(y)}|\theta)\log(P^{(t)}(\mathbf{x}^{(y)}|\theta))$

Experimental results

- The proposed E³Outlier framework achieves remarkable performance gain.

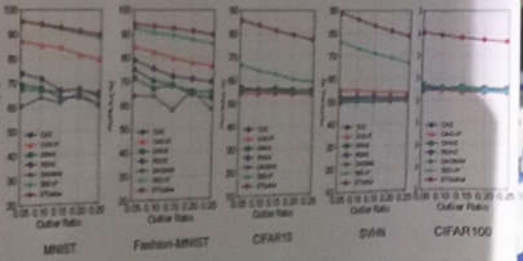


Table 1: AUROC/AUPR-in/AUPR-out (%) for UOD methods. The best performance is in bold.

Dataset	ρ	CAE	CAE-IF	DRAE	RDAE	DAGMM	SSD-IF	E³Outlier
MNIST	10%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**94.1/99.3/67.5**
	20%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**91.3/97.6/72.3**
F-MNIST	10%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**93.3/99.8/75.9**
	20%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**91.2/97.1/78.9**
CIFAR10	10%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**83.5/97.5/43.4**
	20%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**79.3/93.1/52.7**
SVHN	10%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**86.8/98.8/36.7**
	20%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**81.8/93.4/47.6**
CIFAR100	10%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**79.2/96.8/33.3**
	20%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**77.8/92.4/46.5**

Our codes and results can be verified at https://github.com/demonzyj56/E3Outlier