Rethinking Efficient Lane Detection via Curve Modeling
Supplementary Material

This supplementary material is organized as follows: Section 1 describes the FPS test protocol and environments; Section 2 introduces implementation details for each compared method (including ours in Section 2.8); Section 3 provides implementation details for Bézier curves, including sampling, ground truth generation and transforms; Section 4 formulates the IoU loss for Bézier curves and discusses why it failed; Section 5 explores matching priors other than the *centerness* prior; Section 6 presents qualitative results from our method, visualized on three datasets. Section 7 shows extra ablation studies on datasets other than CULane [10], to verify the generalization of feature flip fusion. Section 8 discusses limitations and recognizes new progress in the lane detection field.

1. FPS Test Protocol

Let one Frames-Per-Second (FPS) test trial be the average runtime of 100 consecutive model inference with its PyTorch [12] implementation, without calculating gradients. The input is a 3x360x640 random Tensor (some use all 1 [17], which does not have impact on speed). Note that all methods do **not** use optimization from packages like TensorRT. We wait for all CUDA kernels to finish before counting the whole runtime. We use Python *time.perf_counter()* since it is more precise than *time.time()*. For all methods, the FPS is reported as the best result from 3 trials.

Before each test trial, at least 10 forward pass is conducted as warm-up of the device. For each new method to be tested, we keep running warm-up trials of a recorded method until the recorded FPS is reached again, so we can guarantee a similar peak machine condition as before.

Evaluation Environment. The evaluation platform is a 2080 Ti GPU (standard frequency), on a Intel Xeon-E3 CPU server, with CUDA 10.2, CuDNN 7.6.5, PyTorch 1.6.0. FPS is a platform-sensitive metric, depending on GPU frequency, condition, bus bandwidth, software versions, *etc*. Also using 2080 Ti, Tabelini *et al*. [17] can achieve a better peak performance for all methods. Thus we use the same platform for all FPS tests, to provide fair comparisons.

Remark. Note that **FPS (image/s)** is different from **throughput (image/s)**. Since FPS restricts batch size to 1, which better simulates the real-time application scenario. While throughput considers a batch size more than 1. LSTR [9] reported a 420 FPS for its fastest model, which is actually throughput with batch size 16. Our re-tested FPS is 98.

2. Specifications for Compared Methods

2.1. Segmentation Baseline

The segmentation baseline is based on DeeplabV1 [3], originally proposed in SCNN [10]. It is essentially the original DeeplabV1 without CRF, lanes are considered as different classes, and a separate lane existence branch (a series of convolution, pooling and MLP) is used to facilitate lane post-processing. We optimized its training and testing scheme based on recent advances [19]. Re-implemented in our codebase, it attains higher performance than what recent papers usually report.

Post-processing. First, the existence of a lane is determined by the lane existence branch. Then, the predicted per-pixel probability map is interpolated to the input image size. After that, a 9×9 Gaussian blur is applied to smooth the predictions. Finally, for each existing lane class, the smoothed probability map is traversed by pre-defined Y coordinates (quantized), and corresponding X coordinates are recorded by the maximum probability position on the row (provided it passes a fixed threshold). Lanes with less than two qualified points are simply discarded.

Data Augmentation. We use a simple random rotation with small angles (3 degrees), then resize to input resolution.

2.2. SCNN

Our SCNN [10] is re-implemented from the Torch7 version of the official code. Advised by the authors, we added an initialization trick for the spatial CNN layers, and learning rate warm-up, to prevent gradient explosion caused by recurrent feature aggregation. Thus, we can safely adjust the learning rate. Our improved SCNN achieves significantly better performance than the original one.

Some may find reports of 96.53 accuracy of SCNN on TuSimple. However, that was a competition entry trained with external data. We report SCNN with ResNet backbones, trained with the same data as other re-implemented methods in our codebase.

Post-processing. Same as Section 2.1.
Data Augmentation. Same as Section 2.1.

2.3. RESA

Our RESA [19] is implemented based on its published paper. A main difference to the official code release is we do not cutout no-lane areas (in each dataset, there is a certain height range for lane annotation). Because that trick is dataset specific and not generalizable, we do not use that for all compared methods. Other differences are all validated to have better performance than the official code, at least on the CULane *val* set.
Post-processing. Same as Section 2.1.
Data Augmentation. Same as Section 2.1. The original RESA paper [19] also apply random horizontal flip, which was found ineffective in our re-implementation.

2.4. UFLD

Ultra Fast Lane Detection (UFLD) [13] is reported from their paper and open-source code. Since TuSimple FP and FN information is not in the paper, and training from source code leads to very high FP rate (almost 20%), we did not report their performance on this dataset. We adjusted its profiling scripts to calculate number of parameters and FPS in our standard.
Post-processing. Since this method uses gridding cells (each cell is equivalent to several pixels in a segmentation probability map), each point's X coordinate is calculated as the expectation of locations (cells from the same row), i.e. a weighted average by probability. Differently from segmentation post-processing, it is possible to be efficiently implemented.
Data Augmentation. Augmentations include random rotation and some form of random translation.

2.5. PolyLaneNet

PolyLaneNet [16] is reported from their paper and open-source code. We added a profiling script to calculate number of parameters and FPS in our standard, by help of the paper authors.
Post-processing. This method requires no post-processing.
Data Augmentation. Augmentations include large random rotation (10 degrees), random horizontal flip and random crop. They are applied with a probability of $\frac{10}{11}$.

2.6. LaneATT

LaneATT [17] is reported from their paper and open-source code. We adjusted its profiling scripts to calculate parameters and FPS in our standard, by help of the authors.
Post-processing. Non-Maximal Suppression (NMS) is implemented by a customized CUDA kernel. An extra interpolation of lanes by B-Spline is removed both in testing and profiling, since it is slowly executed on CPU and provides little improvement ($\sim 0.2\%$ on CULane).
Data Augmentation. LaneATT uses random affine transforms including scale, translation and rotation. While it also uses random horizontal flip.
Followup. We did not have time to validate the re-implementation of LaneATT in our codebase, prior the submission deadline. Therefore, the LaneATT performance is still reported from the official code. Our re-implementation indicates that all LaneATT results are reproducible except for the ResNet-34 backbone on CULane, which is slightly outside the standard deviation range, but still a reasonable number.

2.7. LSTR

LSTR [9] is re-implemented in our codebase. All ResNet backbone methods start from ImageNet [6] pre-training. While LSTR [9] use 256 channels ResNet-18 for CULane ($2\times$), 128 channels for other datasets ($1\times$), which makes it impossible to use off-the-shelf pre-trained ResNets. Although whether ImageNet pre-training helps lane detection is still an open question. Our reported performance of LSTR on CULane, is the first documented report of LSTR on this dataset. With tuning of hyper-parameters (learning rate, epochs, prediction threshold), bug fix (the original classification branch has 3 output channels, which should be 2), we achieve 4% better performance on CULane than the authors' trial. Specifically, we use learning rate 2.5×10^{-4} with batch size 20. 150 and 2000 epochs, 0.95 and 0.5 prediction thresholds, for CULane and TuSimple. The lower threshold in TuSimple is due to the official test metric, which significantly favors a high recall. However, for real-world applications, a high recall leads to high False Positive rate, which is undesired.

We divide the curve loss weighting by 10 with our LSTR-Beizer ablation, since there were 100 sample points with both X and Y coordinates to fit, that is a loss scale about 10 times the original loss (LSTR loss takes summation of point $L1$ distances instead of average). This modulation achieves a similar loss landscape to original LSTR.
Post-processing. This method requires no post-processing.
Data Augmentation. Data augmentation includes PolyLaneNet's (Section 2.5), then appends random color distortions (brightness, contrast, saturation, hue) and random lighting by a light source calculated from the COCO dataset [7]. That is by far the most complex data augmentation pipeline in this research field, we have validated that all components of this pipeline helps LSTR training.
Remark. The polynomial coefficients of LSTR are unbounded, which leads to numerical instability (while the bipartite matching requires precision), and high failure rate of training. The failure rate of fp32 training on CULane is $\sim 30\%$. This is circumvented in BézierLaneNet, since

our $L1$ loss can be bounded to $[0, 1]$ without influence on learning (control points easily converges to on-image).

2.8. BézierLaneNet

BézierLaneNet is implemented in the same code framework where we re-implemented other methods. Same as LSTR, the default prediction threshold is set to 0.95, while 0.5 is used for TuSimple [1].

Post-processing. This method requires no post-processing.

Data Augmentation. We use augmentations similar to LSTR (Section 2.7). Concretely, we remove the random lighting from LSTR (to strictly avoid using knowledge from external data), and replace the PolyLaneNet $\frac{10}{11}$ chance augmentations with random affine transforms and random horizontal flip, like LaneATT (Section 2.6). The random affine parameters are: rotation (10 degrees), translation (maximum 50 pixels on X, 20 on Y), scale (maximum 20%).

Polynomial Ablations. For the polynomial ablations (Table 7), we modified the network to predict 6 coefficients for 3rd order Polynomial (4 curve coefficients and start/end Y coordinates). Extra $L1$ losses are added for the start/end Y coordinates similar to LSTR [9]. With extensive tryouts (adjusting learning rate, loss weightings, number of epochs), even at the full BézierLaneNet setup, with 150 epochs on CULane, the models still can not converge to a good enough solution. In other word, not precise enough to pass the CULane metric. The sampling loss on polynomial curves can only get to 0.02, which means $0.02 \times 1640\text{pixels} = 32.8\text{pixels}$ average X coordinate error on training set. CULane requires a 0.5 IoU between curves, which are enlarged to 30 pixels wide, thus at least around 10 pixels average error is needed to get meaningful results. By loosen up the IoU requirement to 0.3, we can get F1 score 15.82 for "3rd Polynomial from BézierLaneNet". Although the reviewing committee suggested adding simple regularization for this ablation to converge, regretfully we failed to do this.

3. Bézier Curve Implementation Details

Fast Sampling. The sampling of Bézier curves may seem tiresome due to the complex Bernstein basis polynomials. To fast sample a Bézier curve by a series of fixed t values, one can simply pre-compute the results from Bernstein basis polynomials, then the sampling process becomes a simple matrix multiplication.

Remarks on GT Generation. The ground truth of Bézier curves are generated with least squares fitting, a common technique for polynomials. We use it for its simplicity and the fact that it already shows near-perfect lane line fitting ability (99.996 and 99.72 F1 score on CULane *test* and LLAMAS *val*, respectively). However, it is not an ideal algorithm for parametric curves. There is a whole research field for fitting Bézier curves better than the least squares method [11].

Bézier Curve Transform. Another implementation difficulty on Bézier curves is how to apply affine transform (for transforming ground truth curves in data augmentation). Mathematically, affine transform on the control points is equivalent to affine transform on the entire curve. However, translation or rotation can move control points out of the image. In this case, a cutting of Bézier curves is required. The classical De Casteljau's algorithm is used for cutting an on-image Bézier curve segment. Assume a continuous on-image segment, valid sample points with minimum boundary $t = t_0$, maximum boundary $t = t_1$. The formula to cut a cubic Bézier curve defined by control points $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ to its on-image segment $\mathcal{P}'_0, \mathcal{P}'_1, \mathcal{P}'_2, \mathcal{P}'_3$, is derived as:

$$
\begin{aligned}
\mathcal{P}'_0 &= u_0u_0u_0\mathcal{P}_0 + (t_0u_0u_0 + u_0t_0u_0 + u_0u_0t_0)\mathcal{P}_1 \\
&\quad + (t_0t_0u_0 + u_0t_0t_0 + t_0u_0t_0)\mathcal{P}_2 + t_0t_0t_0\mathcal{P}_3, \\
\mathcal{P}'_1 &= u_0u_0u_1\mathcal{P}_0 + (t_0u_0u_1 + u_0t_0u_1 + u_0u_0t_1)\mathcal{P}_1 \\
&\quad + (t_0t_0u_1 + u_0t_0t_1 + t_0u_0t_1)\mathcal{P}_2 + t_0t_0t_1\mathcal{P}_3, \\
\mathcal{P}'_2 &= u_0u_1u_1\mathcal{P}_0 + (t_0u_1u_1 + u_0t_1u_1 + u_0u_1t_1)\mathcal{P}_1 \\
&\quad + (t_0t_1u_1 + u_0t_1t_1 + t_0u_1t_1)\mathcal{P}_2 + t_0t_1t_1\mathcal{P}_3, \\
\mathcal{P}'_3 &= u_1u_1u_1\mathcal{P}_0 + (t_1u_1u_1 + u_1t_1u_1 + u_1u_1t_1)\mathcal{P}_1 \\
&\quad + (t_1t_1u_1 + u_1t_1t_1 + t_1u_1t_1)\mathcal{P}_2 + t_1t_1t_1\mathcal{P}_3,
\end{aligned}
\tag{1}
$$

where $u_0 = 1 - t_0$, $u_1 = 1 - t_1$. This formula can be efficiently implemented by matrix multiplication. The possibility of noncontinuous cubic Bézier segment on lane detection datasets is extremely low and thus ignored for simplicity. If it does happen, Equation (1) will not change the curve, while our network can also predict out-of-image control points, which still fit the on-image lane segments.

4. IoU Loss for Bézier Curves

Here we briefly introduce how we formulated the IoU loss between Bézier curves. Before diving into the algorithm, there are two preliminaries.

- Polar sort: By anchoring on an arbitrary point inside the N-sided polygon with vertices $c_i(x_i, y_i)_{i=1}^N$ (normally the mean coordinate between vertices $c' = (\frac{1}{N}\sum_{i=1}^N x_i, \frac{1}{N}\sum_{i=1}^N y_i)$), vertices are sorted by its *atan2* angles. This will return a clockwise or counterclockwise polygon.
- Convex polygon area: A sorted convex polygon can be efficiently cut into consecutive triangles by simple indexing operations. The convex polygon area is the sum of these triangles. The area S of triangle $((x_1, y_1), (x_2, y_2), (x_3, y_3))$ is: $S = \frac{1}{2}|x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)|$.

Assume we have two convex hulls from Bézier curves (there are a lot of convex hull algorithms). Now the IoU between Bézier curves are converted to IoU between convex polygons. Based on the simple fact that the intersec-

tion of convex polygons is still a convex polygon, after polar sorting all the convex hulls and determining the intersected polygon, we can easily formulate IoU calculations as a series of convex polygon area calculations. The difficulty lies in how to efficiently determine the intersection between convex polygon pairs.

Consider two intersected convex polygons, their intersection includes two types of vertices:

- Intersections: intersection points between edges.
- Insiders: vertices inside/on both polygons.

For Intersections, we first represent every polygon edge as the general line equation: $ax + by = c$.

Then, for line $a_1x + b_1y = c_1$ and line $a_2x + b_2y = c_2$, the intersection (x', y') is calculated by:

$$\begin{aligned} x' &= (b_2c_1 - b_1c_2)/det \\ y' &= (a_1c_2 - a_2c_1)/det, \end{aligned} \tag{2}$$

where $det = a_1b_2 - a_2b_1$. All (x', y') that is on the respective line segments are Intersections.

For Insiders, there is a certain definition:

Def. 1 *For a convex polygon, point $P(x, y)$ on the same side of each edge is inside the polygon.*

A sorted convex polygon is a series of edges (line segments defined by $P_0(x_0, y_0), P_1(x_1, y_1)$), the equation to decide which side a point is to a line segment is as follows:

$$sign = (y - y_0)(x_1 - x_0) - (x - x_0)(y_1 - y_0). \tag{3}$$

$sign > 0$ means P is on the right side, $sign < 0$ is the left side, and $sign = 0$ means P is on the line segment. Note that equality is not a stable operation for float computations. But there are simple ways to circumvent that in coding, which we will not elaborate here.

There are other ways to determine Intersections and Insiders, but the above formulas can be efficiently implemented with matrix operations and indexing, making it possible to quickly train networks with batched inputs.

Finally, after being able to compute convex polygon intersections and areas, the Generalized IoU loss (GIoU) is simply (as in [15]):

input : Two arbitrary convex shapes: $A, B \subseteq \mathbb{S} \in \mathbb{R}^n$
output: $GIoU$
1. For A and B, find the smallest enclosing convex object C, where $C \subseteq \mathbb{S} \in \mathbb{R}^n$
2. $IoU = \dfrac{|A \cap B|}{|A \cup B|}$
3. $GIoU = IoU - \dfrac{|C \backslash (A \cup B)|}{|C|}$

Union is computed as $A \cup B = A + B - A \cap B$. The enclosing convex object C can be computed as the convex hull of two convex polygons, or upper-bounded by a enclosing rectangle. We implement the IoU computation purely in PyTorch [12], the runtime for our implementation is only about $5\times$ the runtime of rectangle IoU loss computation.

However, lane lines are mostly straight based on road design regulations [4, 18]. This leads to extremely small convex hull area for Bézier curves, thus introduces numerical instabilities in optimization. Although succeeded in a toy polygon fitting experiment, we currently failed to observe the loss's convergence to help learning on lane datasets.

5. GT and Prediction Matching Prior

Figure 1. Logits activation statistics ($1 \times \frac{W}{16}$) on CULane [10].

Instead of the *centerness* prior, we explore a local maximum prior, *i.e.*, restricts matched prediction to have a local maximum classification logit. This prior can facilitate the model to understand the spatially sparse structure of lane lines. As shown in Figure 1, the learned feature activation for classification logits exhibits a similar structure as an actual driving scene.

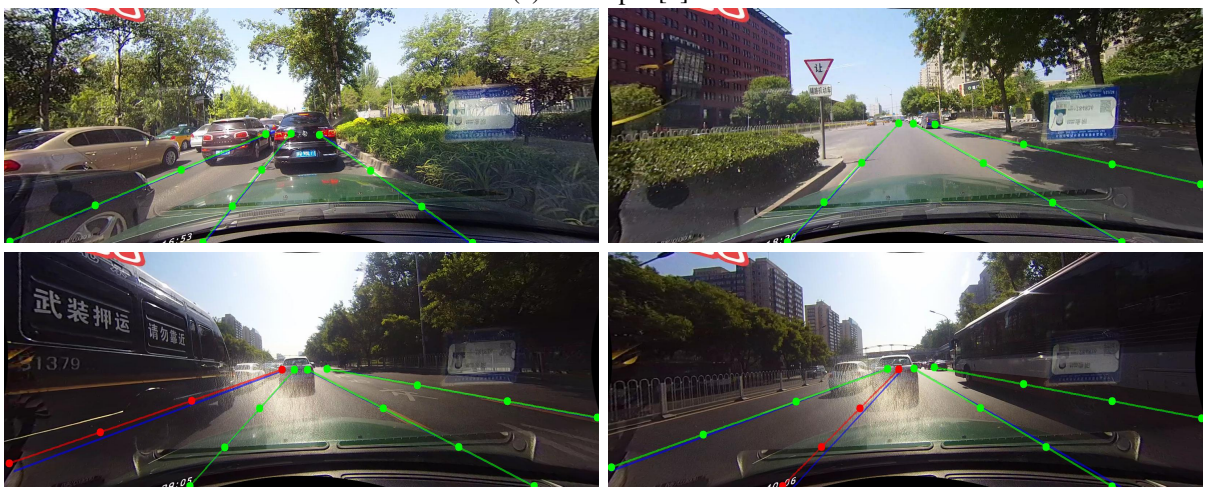
6. Qualitative Results

Qualitative results are shown in Figure 2, from our ResNet-34 backbone models. Bézier control points are highlighted with large circles. False Positives (FP) are marked by red, True Positives (TP) are marked by green, ground truth are drawn in blue. Some blue lines that are barely visible are quite precisely covered by green lines (a precise prediction). It is recommended to enlarge the manuscript, to better observe lane line details. For each dataset, 4 results are shown in two rows: first row shows qualitative successful predictions; second row shows typical failure cases.

TuSimple. As shown in Figure 2(a), our model fits highway curves well, only slight errors are seen on the far side where image details are destroyed by projection. Our typical failure case is a high FP rate, mostly attributed to the use of low threshold (Section 2.8). However, in the bottom-right wide road scene, our FP prediction is actually a meaningful lane line that is ignored in center line annotations.

CULane. As shown in Figure 2(b), most lanes in this dataset are straight. Our model can make accurate predictions under heavy congestion (top-left) and shadows (top-right, shadow cast by trees). A typical failure case is inaccurate prediction under occlusion (second row), in these cases one often cannot visually tell which one is better (ground truth or our FP prediction).

(a) TuSimple [1].



(b) CULane [10].

(c) LLAMAS [2].

Figure 2. Qualitative results from BézierLaneNet (ResNet-34) on *val* sets. False Positives (FP) are marked by red, True Positives (TP) are marked by green, ground truth are drawn in blue. Bézier curve control points are marked with solid circles. Images are slightly resized for alignment. Best viewed in color, in 2× scale.

LLAMAS. As shown in Figure 2(c), our method performs accurate for clear straight-lines (top-left), and also good for large curvatures in a challenging scene almost entirely covered by shadow. In bottom-left image, our model fails in a low-illumination, tainted road. While in the other low-illumination scene (bottom-right), the unsupervised annotation from LIDAR and HD-map is misled by the white arrow (see the zigzag shape of the right-most blue line).

7. Extra Results

	TuSimple [1]	**LLAMAS [2]**
Bézier Baseline	93.36	95.27
+ Feature Flip Fusion	95.26 (+1.90)	96.00 (+0.73)

Table 1. Ablation study on TuSimple (*test* set Accuracy) and LLAMAS (*val* set F1), before and after adding the Feature Flip Fusion module. Reported 3-times average with the ResNet-34 backbone, since ablations often are not stable enough on these datasets to exhibit a clear difference between methods.

8. Discussions

There exists a primitive application of lane detectors from lateral-mounted cameras [5] that contradicts the use of feature flip fusion, to estimate the distance to the border of the drivable area. In this case, possibly a lower order Bézier curve baseline (with row-wise instead of column-wise pooling) would suffice. This is out of the focus of this paper.

Recent Progress. Recently, others have explored alternative lane representation or formulation methods that do not fully fit in the three categories (segmentation, point detection, curve). Instead of the popular top-down regime, [14] propose a bottom-up approach that focus on local details. [8] achieve state-of-the-art performance, but the complex conditional decoding of lane lines results in unstable runtime depending on the input image, which is not desirable for a real-time system.

References

[1] TuSimple benchmark. https://github.com/TuSimple/tusimple-benchmark, 2017. 3, 5, 6

[2] Karsten Behrendt and Ryan Soussan. Unsupervised labeled lane markers using maps. In *ICCV*, 2019. 5, 6

[3] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Semantic image segmentation with deep convolutional nets and fully connected crfs. In *ICLR*, 2015. 1

[4] MOT Highway Department and Highway Engineering Committee under China Association for Engineering Construction Standardization. *Technical Standard of Highway Engineering*. 2004. 4

[5] Alexandru Gurghian, Tejaswi Koduri, Smita V Bailur, Kyle J Carey, and Vidya N Murali. Deeplanes: End-to-end lane position estimation using deep neural networksa. In *CVPR Workshops*, 2016. 6

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012. 2

[7] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *ECCV*. Springer, 2014. 2

[8] Lizhe Liu, Xiaohao Chen, Siyu Zhu, and Ping Tan. Condlanenet: a top-to-down lane detection framework based on conditional convolution. In *ICCV*, 2021. 6

[9] Ruijin Liu, Zejian Yuan, Tie Liu, and Zhiliang Xiong. End-to-end lane shape prediction with transformers. In *WACV*, 2021. 1, 2, 3

[10] Xingang Pan, Jianping Shi, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Spatial as deep: Spatial cnn for traffic scene understanding. In *AAAI*, 2018. 1, 4, 5

[11] Tim A Pastva. Bezier curve fitting. Technical report, NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 1998. 3

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019. 1, 4

[13] Zequn Qin, Huanyu Wang, and Xi Li. Ultra fast structure-aware deep lane detection. In *ECCV*, 2020. 2

[14] Zhan Qu, Huan Jin, Yang Zhou, Zhen Yang, and Wei Zhang. Focus on local: Detecting lane marker from bottom up via key point. In *CVPR*, 2021. 6

[15] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. In *CVPR*, 2019. 4

[16] Lucas Tabelini, Rodrigo Berriel, Thiago M Paixao, Claudine Badue, Alberto F De Souza, and Thiago Oliveira-Santos. Polylanenet: Lane estimation via deep polynomial regression. In *ICPR*, 2020. 2

[17] Lucas Tabelini, Rodrigo Berriel, Thiago M Paixao, Claudine Badue, Alberto F De Souza, and Thiago Oliveira-Santos. Keep your eyes on the lane: Real-time attention-guided lane detection. In *CVPR*, 2021. 1, 2

[18] Federal Highway Administration under United States Department of Transportation. *Standard Specifications for Construction of Roads and Bridges on Federal Highway Projects*. 2014. 4

[19] Tu Zheng, Hao Fang, Yi Zhang, Wenjian Tang, Zheng Yang, Haifeng Liu, and Deng Cai. Resa: Recurrent feature-shift aggregator for lane detection. In *AAAI*, 2021. 1, 2