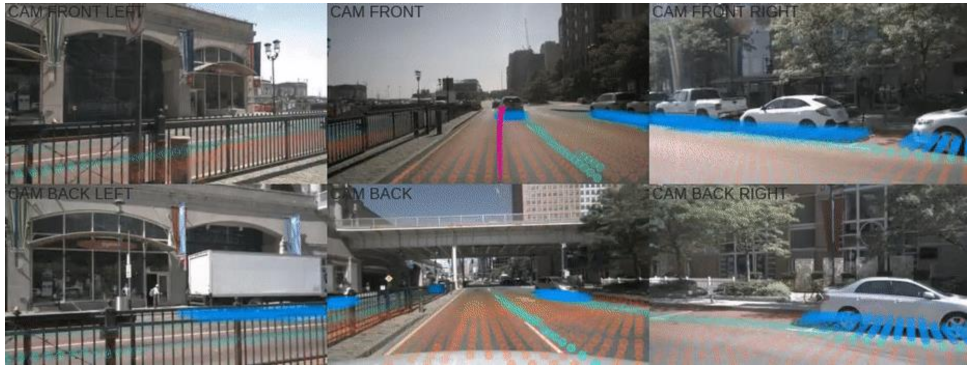
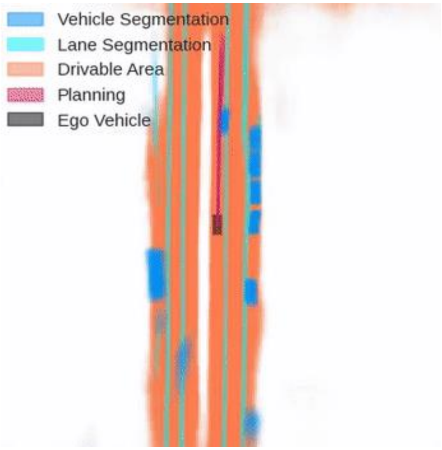
Lift, Splat, Shoot: Encoding Images from Arbitrary Camera Rigs by Implicitly Unprojecting to 3D

Hanyang University, Department of Automotive engineering
Chungwoo Lee
https://github.com/lcwoo
canwooj@gmail.com

Contents



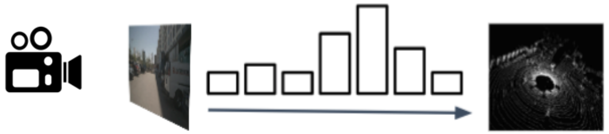


- **General Flow of BEV Perception**
- **Lift, Spat, Shoot (ECCV 2020)**
  - **Contributions**
  - **Method**
    - Lift
    - Splat
    - Shoot
  - **Results**
  - **Conclusion**

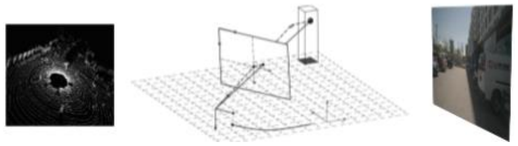
General Flow of BEV Perception

- **How to transfer 2D Image Information to 3D space (View transformation)**
  - **2D - 3D transformation by depth distribution**
  - **3D– 2D back projection**
  - **Pure network based**

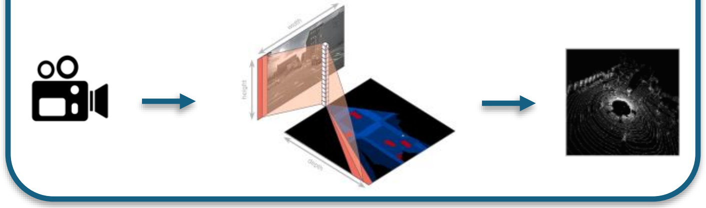
View Transformation 2D-3D



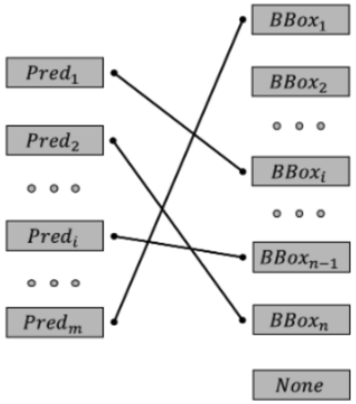
View Transformation 3D-2D



View Transformation Pure-network-based



3D Decoder



2D Features from Images (Backbone)

$$F^*_{2D}(u, v) = M_{feat}(I^*(u, v))$$

3D Features from 2D Features: View Transformation

$$F_{3D}(x, y, z) = M_{trans}(F^*_{2d}((\hat{u}, \hat{v}), [R\ T], K)$$

Lift, Splat, Shoot

Lift, Splat, Shoot: Encoding Images from Arbitrary Camera Rigs by Implicitly Unprojecting to 3D

Jonah Philion Sanja Fidler

NVIDIA University of Toronto Vector Institute

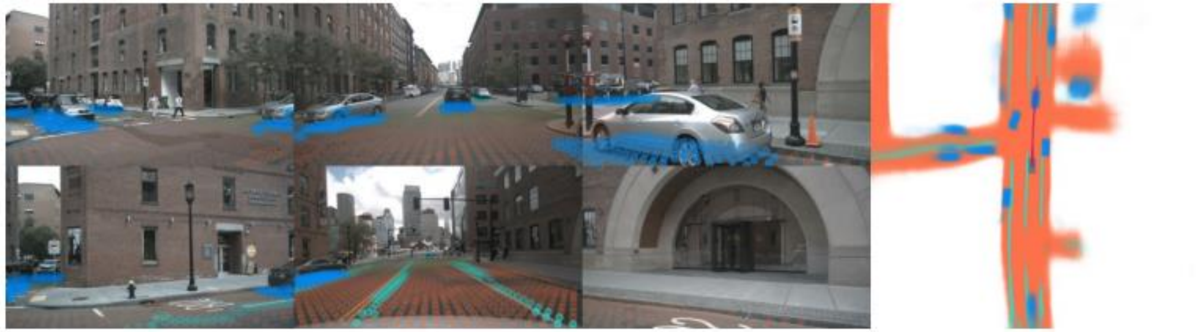


Fig. 1: We propose a model that, given multi-view camera data (left), infers semantics directly in the bird's-eye-view (BEV) coordinate frame (right). We show vehicle segmentation (blue), drivable area (orange), and lane segmentation (green). These BEV predictions are then projected back onto input images (dots on the left).

Lift, Splat, Shoot

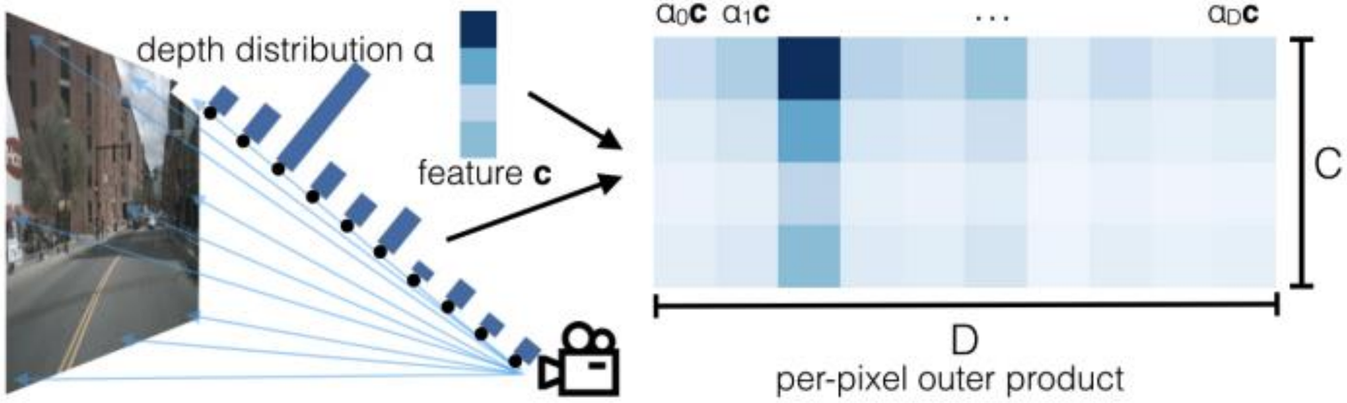
Contributions

- **Lift-Splat Architecture for Camera-Based BEV**
  - Predicts depth distributions from multiple cameras and projects them into a unified BEV grid
- **End-to-End Motion Planning**
  - Learns a BEV cost map for interpretable evaluation and selection of trajectory candidates
  - Integrates raw camera inputs to final motion plans in a single pipeline
- **Robust to Camera Setup & Calibration Errors**
  - Excels on BEV tasks and remains robust to camera dropout/noise
  - Zero-shot transfer to new camera rigs without fine-tuning

Lift: Latent Depth Distribution

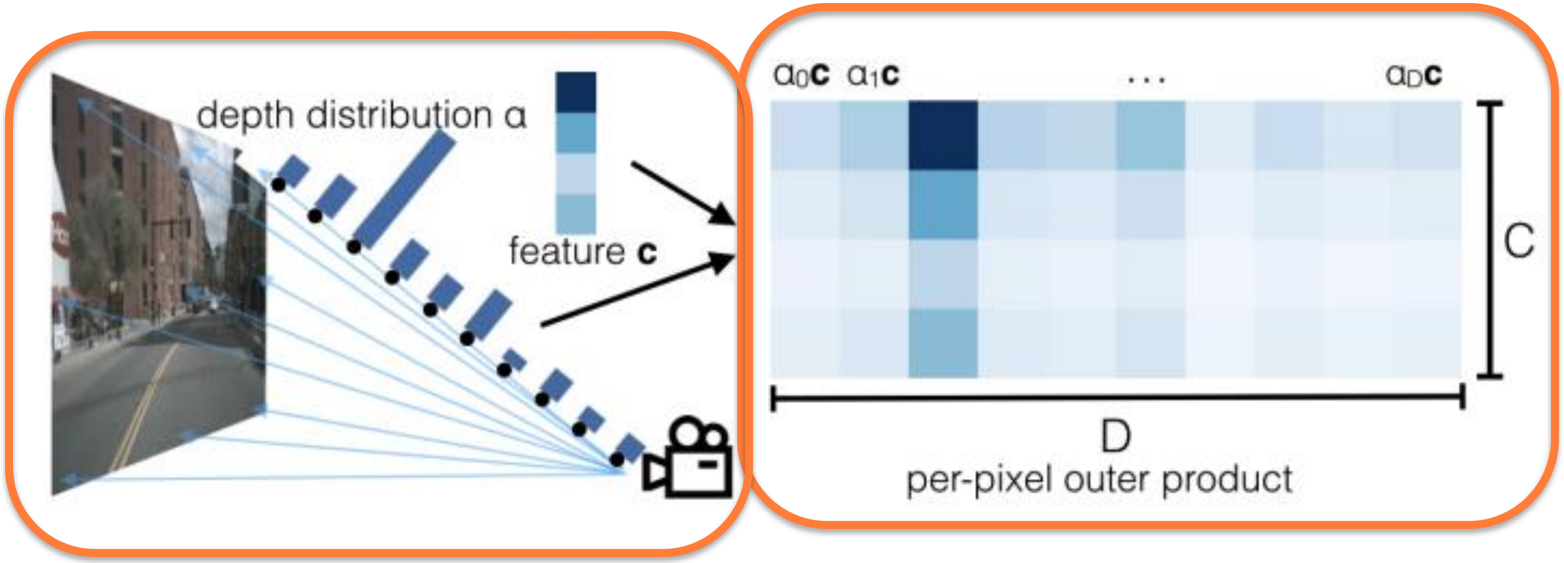
Considering depth candidates per pixel in a 2D image to extract 3D spatial features

- **Extracting 2D features from EfficientNet**
- **DepthNet**
  - **Depth probability distribution** → $\alpha \in \Delta^{|D|-1}$
    (41 depth candidates from 4m to 45m with 1m intervals)
  - Estimates **context vectors** for each pixel → $\boldsymbol{c} \in \mathbb{R}^{C}$
- **Generating 3D features** for each depth candidate d per pixel → $\boldsymbol{c}_d = \alpha_d \cdot \boldsymbol{c}$
- Obtains a 3D representation in the form of **[B , N , D , H , W , C]**



Lift: Latent Depth Distribution

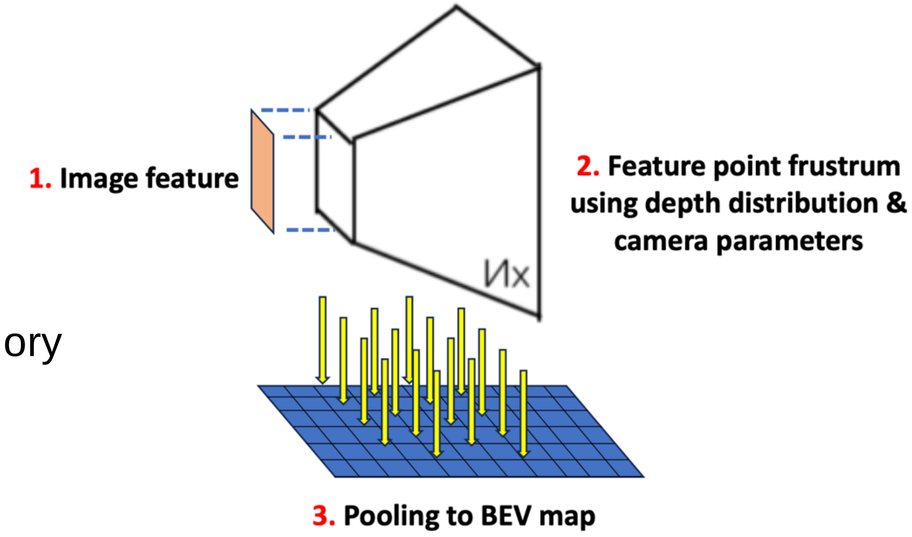
```
def get_depth_feat(self, x):
    x = self.get_eff_depth(x)
    # Depth
    x = self.depthnet(x)                                    # [B, D+C, H, W]

    depth = self.get_depth_dist(x[:, :self.D])
    new_x = depth.unsqueeze(1) * x[:, self.D:(self.D +  self.C)].unsqueeze(2)   # [B, C, D, H, W]

    return depth, new_x
```



Splat: Pillar Pooling

Converting 3D Features to a BEV Pseudo-Image

- **Flatten & Indexing**
  - Flatten the 3D points generated in the Lift phase
  - Convert each point's (x, y, z) coordinates into BEV grid indices
- **Filter & Sort**
  - Remove points that are outside the predefined BEV boundaries
  - Sort points so that those within the same voxel are contiguous in memory
- **Sum Pooling (Cumsum Trick)**
  - Aggregate all features in each voxel using a cumulative sum approach
  - Minimizes unnecessary padding and speeds up computation
- **BEV Pseudo-Image Generation**
  - Place the summed features into the corresponding voxel cells within the BEV grid
  - Collapse the Z-axis to yield the final 2D BEV pseudo-image



Splat: Pillar Pooling

```
# flatten indices
geom_feats = ((geom_feats - (self.bx - self.dx/2.)) / self.dx).long()
geom_feats = geom_feats.view(Nprime, 3)
batch_ix = torch.cat([torch.full([Nprime//B, 1], ix,
                                 device=x.device, dtype=torch.long) for ix in range(B)])
geom_feats = torch.cat((geom_feats, batch_ix), 1)
```

- Shape after flattening: (B*N*D*H*W,4)
- Each row: $(X\_idx, Y\_idx, Z\_idx, \mathrm{batch})$

```
# filter out points that are outside box
kept = (geom_feats[:, 0] >= 0) & (geom_feats[:, 0] < self.nx[0])\
       & (geom_feats[:, 1] >= 0) & (geom_feats[:, 1] < self.nx[1])\
       & (geom_feats[:, 2] >= 0) & (geom_feats[:, 2] < self.nx[2])
x = x[kept]
geom_feats = geom_feats[kept]
```

- Boundary Check
- Remove points outside voxel grid range

```
# get tensors from the same voxel next to each other
ranks = geom_feats[:, 0] * (self.nx[1] * self.nx[2] * B)\
       + geom_feats[:, 1] * (self.nx[2] * B)\
       + geom_feats[:, 2] * B\
       + geom_feats[:, 3]
sorts = ranks.argsort()
x, geom_feats, ranks = x[sorts], geom_feats[sorts], ranks[sorts]
```

- Compute voxel ranks
- Ensure neighboring features within the same voxel are sorted together

Splat: Pillar Pooling

```
class QuickCumsum(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, geom_feats, ranks):
        x = x.cumsum(0)
        kept = torch.ones(x.shape[0], device=x.device, dtype=torch.bool)
        kept[:-1] = (ranks[1:] != ranks[:-1])

        x, geom_feats = x[kept], geom_feats[kept]
        x = torch.cat((x[:1], x[1:] - x[:-1]))

        # save kept for backward
        ctx.save_for_backward(kept)

        # no gradient for geom_feats
        ctx.mark_non_differentiable(geom_feats)

        return x, geom_feats

    @staticmethod
    def backward(ctx, gradx, gradgeom):
        kept, = ctx.saved_tensors
        back = torch.cumsum(kept, 0)
        back[kept] -= 1

        val = gradx[back]

        return val, None, None
```

Efficient Feature Summation within Voxels using Cumulative Sum

Ex)

ranks = [1, 1, 1, 10, 10, 15]

kept = [False, False, True, False, True, True]

Ex)

Kept = [False, False, True, False, True, True]

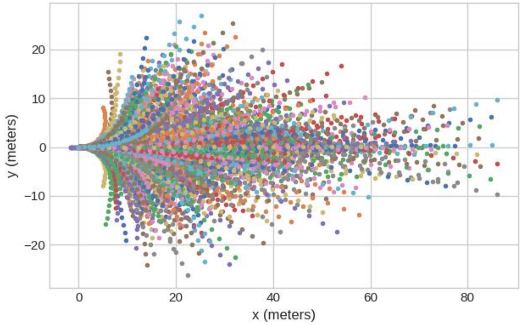
back = [0, 0, 1, 1, 2, 3]

back = [0, 0, 0, 1, 1, 2]

Adjusts gradients precisely to original voxel positions

Shoot: Motion Planning

- **Trajectory Templates (K-Means Clustering)**
  - **From a large dataset of expert trajectories, perform K-Means clustering with K = 1000**
  - **This yields 1,000 representative template trajectories $\{\tau_1, \ldots, \tau_K\}$, each capturing a typical motion pattern.**



- **Cost Computation**
  - **Given a Cost Map $C_o(x, y)$ predicted from sensor observations o,**

    each template trajectory τ_i is scored by summing the cost at every point (x_i, y_i) in that trajectory

$$p(\tau_i|o) = \frac{\exp\left(-\sum_{x_i, y_i \in \tau_i} c_o(x_i, y_i)\right)}{\sum_{\tau \in \mathcal{T}} \exp\left(-\sum_{x_i, y_i \in \tau} c_o(x_i, y_i)\right)}$$

    - $p(\tau_i | o)$: Probability of selecting trajectory τ_i given the observed information o
    - $c_o(x_i, y_i)$: Cost at location (x_i, y_i) in the cost map
  - **Trajectories with lower overall cost become more likely to be selected.**

Shoot: Motion Planning

- Expert Trajectory Labeling
  - In the training dataset, each expert path is matched to the closest template

- Training via Negative Log-Likelihood
  - Use Negative Log-Likelihood Loss to maximize the predicted probability of the expert's matching template

$$\mathcal{L} = -\boldsymbol{log}[\boldsymbol{p}(\boldsymbol{\tau}^{*} \mid \boldsymbol{o})]$$

  - By lowering the cost for the expert's template trajectory, the model learns an end-to-end Cost Map such that "expert-like" paths have lower costs and hence higher probabilities.

Results

Segmentation

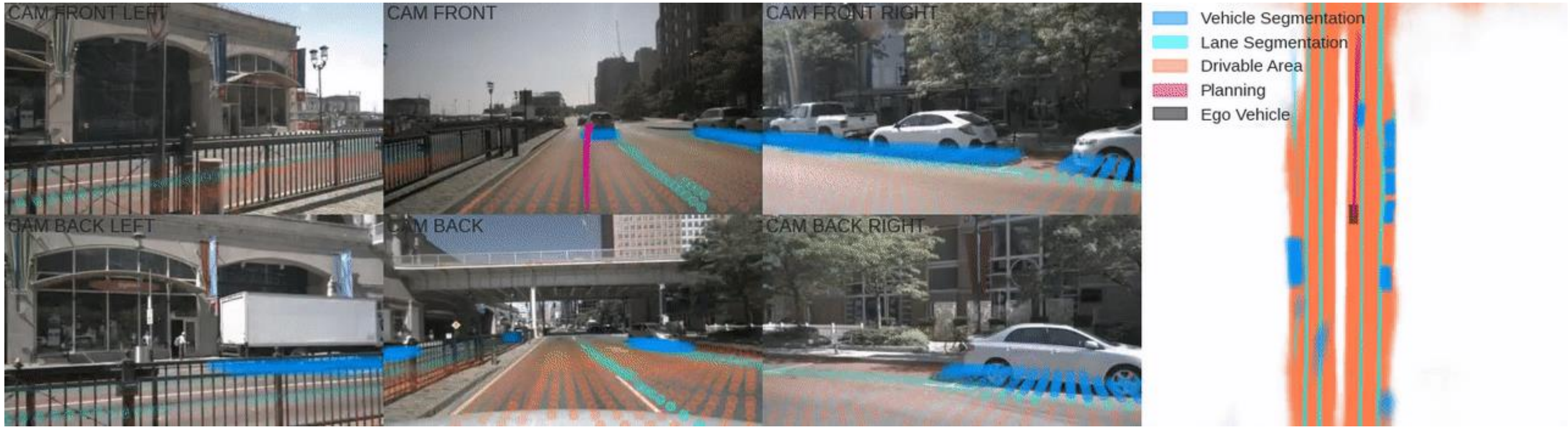
	nuScenes		Lyft	
	Car	Vehicles	Car	Vehicles
CNN	22.78	24.25	30.71	31.91
Frozen Encoder	25.51	26.83	35.28	32.42
OFT	29.72	30.05	39.48	40.43
Lift-Splat (Us)	**32.06**	**32.07**	**43.09**	**44.64**
PON* [28]	24.7	-	-	-
FISHING* [9]	-	30.0	-	56.0

Table 1: Segment. IOU in BEV frame

	Drivable Area	Lane Boundary
CNN	68.96	16.51
Frozen Encoder	61.62	16.95
OFT	71.69	18.07
Lift-Splat (Us)	**72.94**	**19.96**
PON* [28]	60.4	-

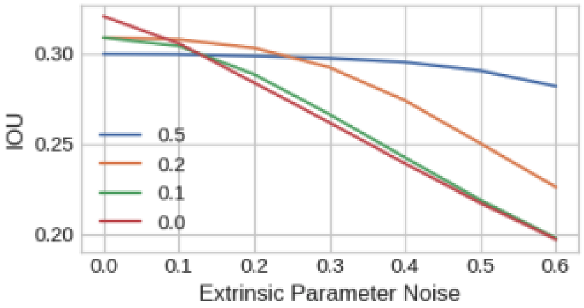
Table 2: Map IOU in BEV frame

Lift-Splat outperform baselines on bird's-eye-view segmentation

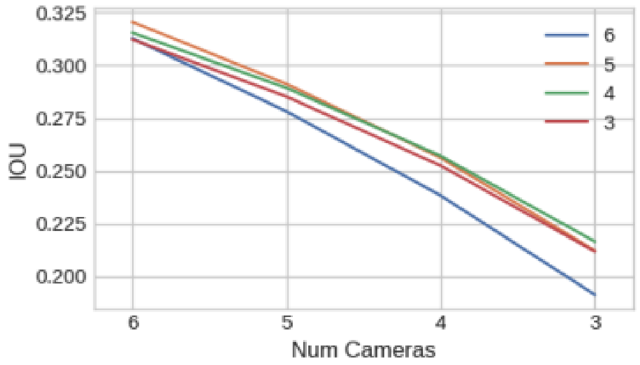


Results

Robustness



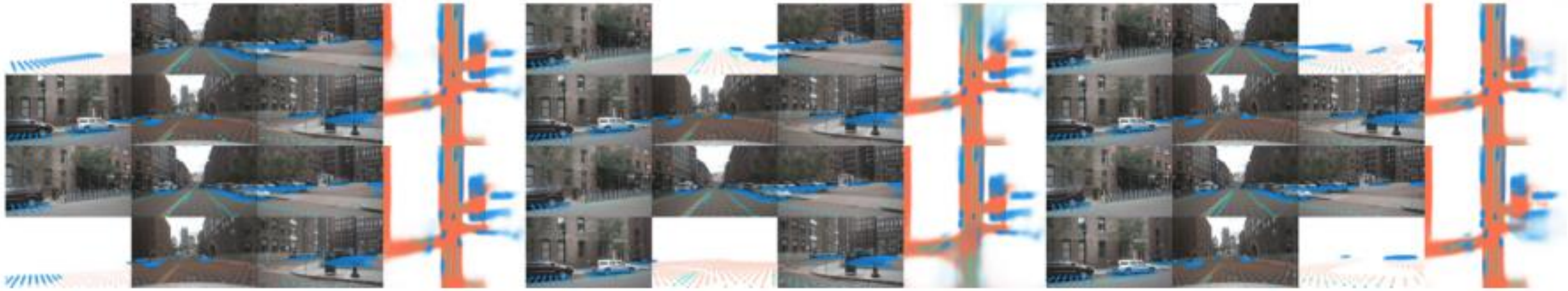
(a) Test Time Extrinsic Noise



(b) Test Time Camera Dropout

Models trained with higher extrinsic noise sustain better performance under test-time calibration errors (Fig. 6a)

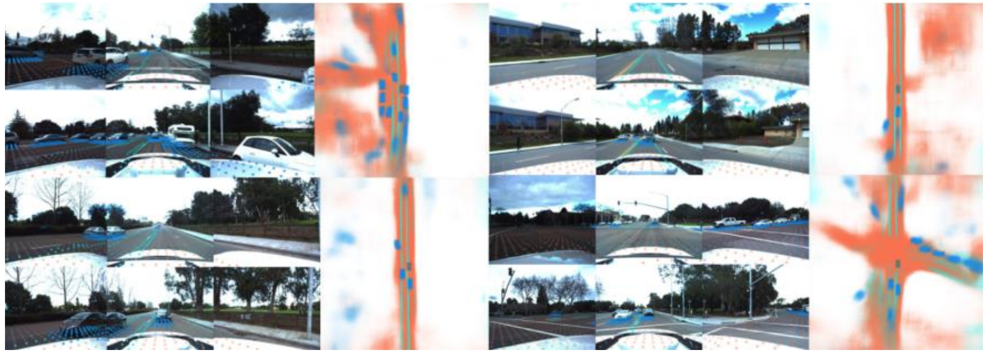
Training with random camera dropout improves resilience to missing cameras at test time (Fig. 6b)



Results

Camera Rig Transfer

	Lyft Car	Lyft Vehicle
CNN	7.00	8.06
Frozen Encoder	15.08	15.82
OFT	16.25	16.27
Lift-Splat (Us)	**21.35**	**22.59**



We take the above experiment a step farther and probe how well our model generalizes to the Lyft camera rig if it was only trained on nuScenes data.

Differs significantly from nuScenes in terms of position, resolution, and field of view (FoV)

Conclusions

- Proposed an end-to-end model for BEV representation from multi-camera images without depth supervision.
- Achieved significantly improved BEV segmentation accuracy compared to previous methods.
- Demonstrated robustness to various sensor errors, such as calibration issues and camera dropout.
- Generalizes well to new camera setups without additional training.
- Facilitates interpretable end-to-end motion planning through learned BEV representations.

Thank you