Targets and target types

Please refer to the simple_bev code.

At line 81, these are the data for one sample. (Note: not all of them are useful)

```
def run_model(model, loss_fn, d, device='cuda:0', sw=None):
    metrics = {}
    total_loss = torch.tensor(0.0, requires_grad=True).to(device)

    imgs, rots, trans, intrins, pts0, extra0, pts, extra, lrtlist_velo,
vislist, tidlist, scorelist, seg_bev_g, valid_bev_g, center_bev_g,
offset_bev_g, radar_data, egopose = d

# shapes and types
# ----------------
# imgs        : torch.Size([1, 1, 4, 3, 448, 672]) torch.float32
# rots        : torch.Size([1, 1, 4, 3, 3]) torch.float32
# trans       : torch.Size([1, 1, 4, 3]) torch.float32
# intrins     : torch.Size([1, 1, 4, 4, 4]) torch.float32
# pts0        : torch.Size([1, 1, 3, 24247]) torch.float32
# extra0      : torch.Size([1, 1, 3, 24247]) torch.float32
# pts         : torch.Size([1, 1, 3, 30000]) torch.float32
# extra       : torch.Size([1, 1, 3, 30000]) torch.float32
# lrtlist_velo: torch.Size([1, 1, 150, 19]) torch.float32
# vislist     : torch.Size([1, 1, 150]) torch.float32
# tidlist     : torch.Size([1, 1, 150]) torch.int64
# scorelist   : torch.Size([1, 1, 150]) torch.float32
# seg_bev_g   : torch.Size([1, 1, 1, 400, 400]) torch.float32
# valid_bev_g : torch.Size([1, 1, 1, 400, 400]) torch.float32
# center_bev_g: torch.Size([1, 1, 1, 400, 400]) torch.float32
# offset_bev_g: torch.Size([1, 1, 2, 400, 400]) torch.float32
# radar_data  : torch.Size([1, 1, 3, 700]) torch.float32
# egopose     : torch.Size([1, 1, 4, 4]) torch.float32
```

According to line 83, the idimensions of the `imgs` are `B0,T,S,C,H,W` where:

- `B`: batch
- `T`: time (useless)
- `S`: sequence (4 because there are four cameras)
- `C`: channel
- `H`: image height
- `W`: image width

At lines 179-184, these are the outputs from its model, which are supposed to be the inputs of the next model.

```
    _, feat_bev_e, seg_bev_e, center_bev_e, offset_bev_e = model(
            rgb_camXs=rgb_camXs,
            pix_T_cams=pix_T_cams,
            cam0_T_camXs=cam0_T_camXs,
            vox_util=vox_util,
            rad_occ_mem0=in_occ_mem0)

# shapes and types
# ----------------
# feat_bev_e  : torch.Size([1, 128, 400, 400]) torch.float32
# seg_bev_e   : torch.Size([1, 1, 400, 400]) torch.float32
# center_bev_e: torch.Size([1, 1, 400, 400]) torch.float32
# offset_bev_e: torch.Size([1, 2, 400, 400]) torch.float32
```

When you ignore the dimension that has size 1, the shapes of `seg_bev_e` and `seg_bev_g` match, as they are the prediction and ground truth, respectively. This is also true for `center_bev_e` and `offset_bev_e`.

The model is called `Segnet` and its heads that create those outputs can be found in the `Decoder`.

Setup and installation

Install requirements and download the repo on GrootCompute

```
conda create -n bev python=3.10
conda activate bev
conda install pytorch==1.12.1 torchvision==0.13.1 cudatoolkit=11.3 -c
pytorch
pip install -U pip
pip install fire efficientnet_pytorch tensorboardX scikit-image pandas
nuscenes-devkit
pip install laspy[laszip]

git clone git@github.com:GrootCompute/simple_bev.git
git checkout fork  # use this branch to process carla data
```

Training and visualization

To run, first put `postprocessed.zip` at the base directory (i.e. one level up from `simple_bev`) and unzip it there. Then, enter `simple_bev` and call `train_carla.py`:

```
cd simple_bev/
python train_carla.py \
       --exp_name='rgb_mine' \
       --max_iters=501 \
       --log_freq=100 \
       --save_freq=100 \
       --batch_size=1 \
       --nworkers=1 \
```

```
        --grad_acc=5 \
        --use_scheduler=True \
        --data_dir='../postprocessed' \
        --log_dir='logs_postprocessed' \
        --ckpt_dir='checkpoints' \
        --device_ids=[0]
```

Once training is done, the model can be found in the subdirectory `checkpoints/1x5_3e-4s_rgb_mine_<time>` (where `<time>` looks something like `17:34:03`). To run the visualization, enter `simple_bev` and call `vis_carla.py` and substitute in the subdirectory name at the line `--init_dir='checkpoints/...'`.

```
cd simple_bev/
python vis_carla.py \
        --max_iters=1 \
        --nworkers=1 \
        --data_dir='../postprocessed' \
        --log_dir='logs_postprocessed' \
        --init_dir='checkpoints/1x5_3e-4s_rgb_mine_17:34:03' \
        --device_ids=[0]
```

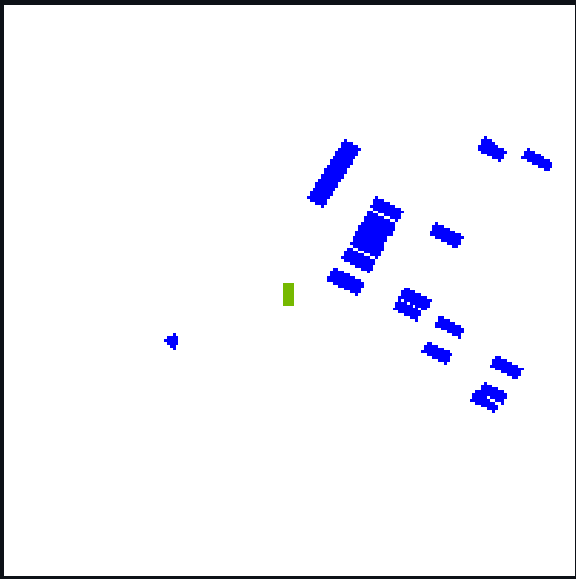
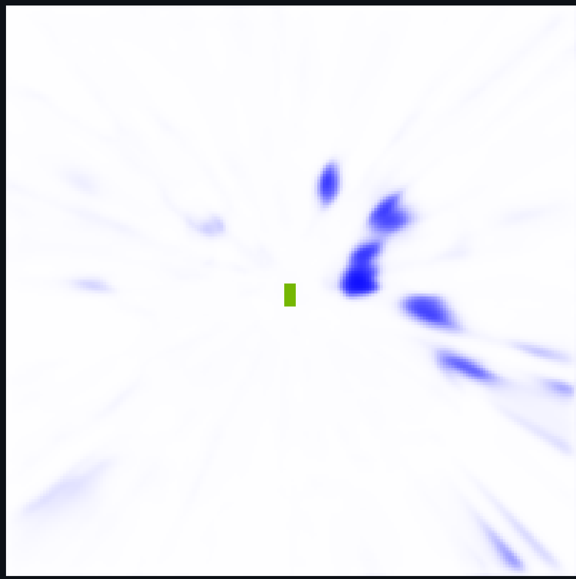
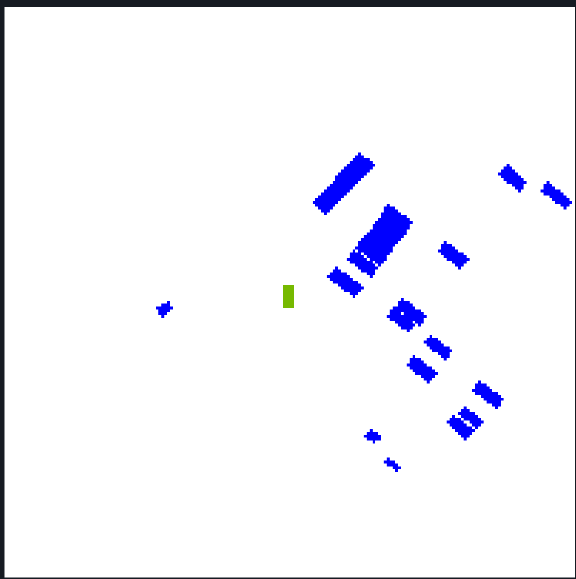
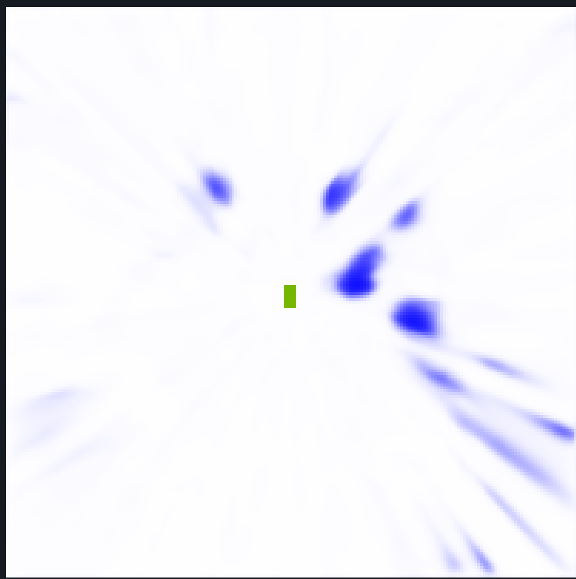
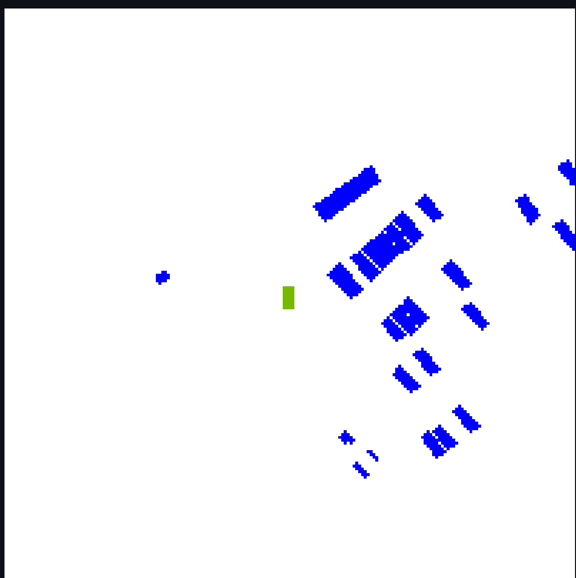
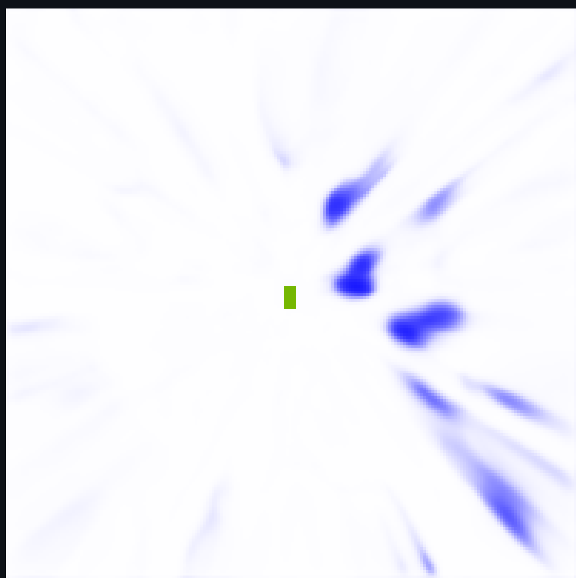
Other things

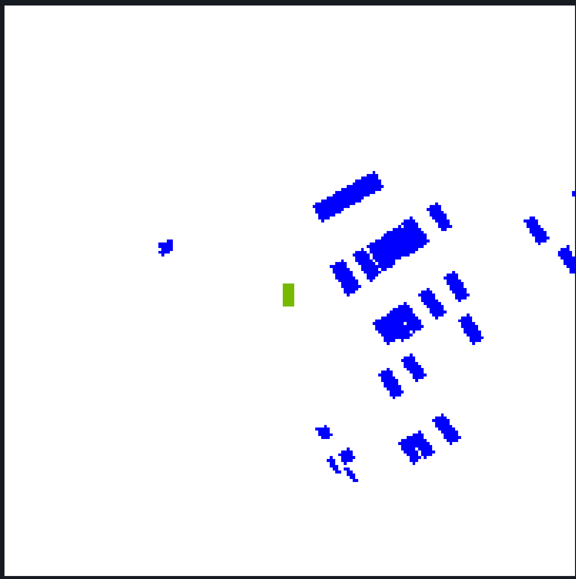
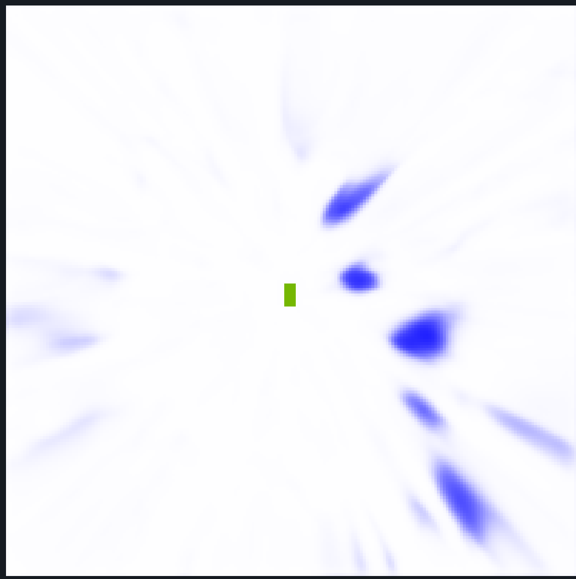
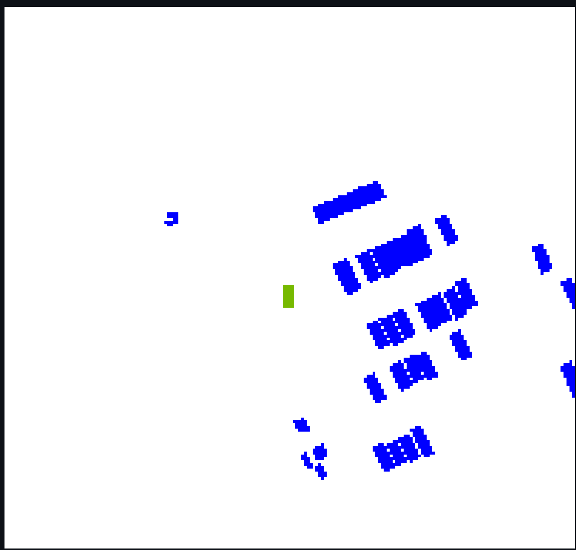
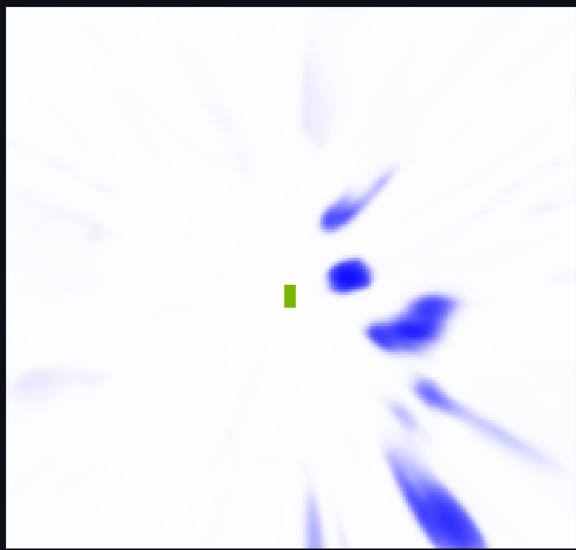
It seems like the segmentation maps generated by `simple_bev` are not usable. There are a lot of confusions with the coordinate systems and transformation matrices (in `nuscenesdataset_carla.py`). I don't think these can be fixed easily.

Targets and target types (nuScenes)

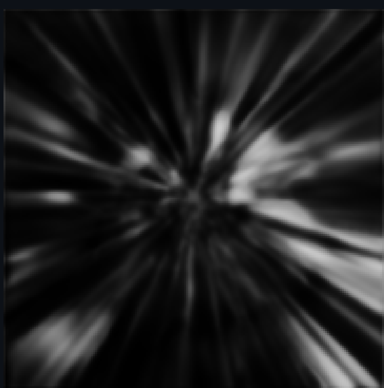
I ran `train_nuscenes.py` for 1000 iterations using **nuScenes data** to obtain a quick model. Then I ran `vis_nuscenes.py` with that model for 5 "timesteps" and these are the outputs. Only `seg_bev_e` (shape: `(1, 1, 200, 200)`) from the outputs of the model is being used for visualization in the script.

```
    _, feat_bev_e, seg_bev_e, center_bev_e, offset_bev_e = model(
            rgb_camXs=rgb_camXs,
            pix_T_cams=pix_T_cams,
            cam0_T_camXs=cam0_T_camXs,
            vox_util=vox_util,
            rad_occ_mem0=in_occ_mem0)
```

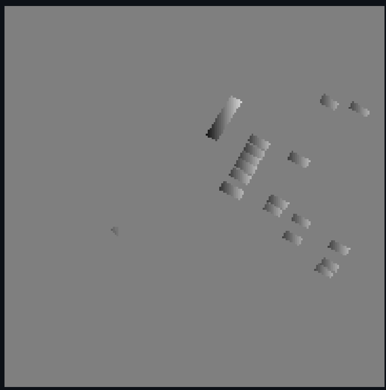
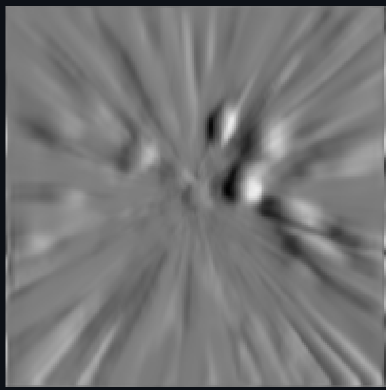
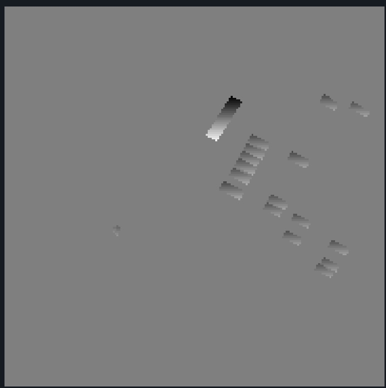
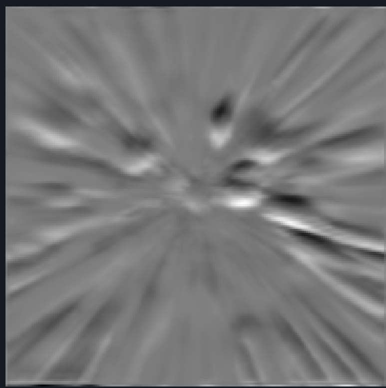
Ground truth (i.e. `seg_bev_g`)	Segnet output (i.e. `seg_bev_e`)
	
	
	

Ground truth (i.e. `seg_bev_g`)	Segnet output (i.e. `seg_bev_e`)
	
	

I dumped `center_bev_e` and `offset_bev_e` into a JSON file and inspectd the outputs. For `center_bev_e` (shape: `(1, 1, 200, 200)`), it looks like this. Only the first timestep is shown:

Ground truth (i.e. `center_bev_g`)	Segnet output (i.e. `center_bev_e`)
	

For `offset_bev_e` (shape: `(1, 2, 200, 200)`), it looks like this. Note: they are shown as x-offset and y-offset separately:

Ground truth (i.e. `offset_bev_g`)	Segnet output (i.e. `offset_bev_e`)
	
	

Mean, min, max of the arrays:

```
>>> center_bev_g.mean(), center_bev_g.min(), center_bev_g.max()
(0.021626132781710477, 0.0, 1.0)
>>> center_bev_e.mean(), center_bev_e.min(), center_bev_e.max()
(0.13331910948131118, 0.0012023173039779067, 0.8007904291152954)
>>> offset_bev_g.mean(), offset_bev_g.min(), offset_bev_g.max()
(-0.02825, -24.0, 22.0)
>>> offset_bev_e.mean(), offset_bev_e.min(), offset_bev_e.max()
(-0.4612532062917482, -13.117025375366211, 13.411355018615723)
```

Loss functions

The outputs of the model are named `seg_bev_e` (segmentation map), `center_bev_e` (centerness), and `offset_bev_e` (offset). Based on my understanding, `seg_bev_e` is the main output that can be used for downstream tasks; while `center_bev_e` and `offset_bev_e` are auxiliary outputs that are meant to optimize the training of the model. This is an excerpt from their paper:

acting as the segmentation task head. Following FIERY [21], we complement the segmentation head with auxiliary task heads for predicting centerness and offset, which serve to regularize the model. The offset head produces a vector field where, within each object mask, each vector points to the center of that object. We train the segmentation head with a cross-entropy loss, and supervise the centerness and offset fields with an L1 loss. We use an uncertainty-based learnable

At lines 186-189, these are the loss functions applied to `seg_bev_e`, `center_bev_e`, and `offset_bev_e`.

```
    ce_loss = loss_fn(seg_bev_e, seg_bev_g, valid_bev_g)  # this is like torch.nn.BCEWithLogitsLoss, but using a 'valid' mask to remove invalid elements and calculate the "masked mean". Note that they set pos_weight to 2.13.
    center_loss = balanced_mse_loss(center_bev_e, center_bev_g)  # this is like torch.nn.MSELoss, but can be supplied with a 'valid' mask, which is not supplied here
    offset_loss = torch.abs(offset_bev_e-offset_bev_g).sum(dim=1, keepdim=True)  # this is like torch.nn.L1Loss
    offset_loss = utils.basic.reduce_masked_mean(offset_loss, seg_bev_g*valid_bev_g)  # a 'valid' mask is used here to remove invalid elements and calculate the "masked mean"
```

Prompt: Try to understand the intentions behind the choices of loss functions.

- Generally speaking, for a feature map or segmentation map, where the output is a grid, and each cell contains a probability in the range of 0-1, the natural choice is binary cross-entropy (`torch.nn.BCEWithLogitsLoss`) or multi-class cross-entopy (`torch.nn.CrossEntropyLoss`). For deeper reasons, I refer you to Logistic regression.
- Meanwhile, if the output represents a measurement (with arbitrary range) that has linear response (e.g. distance), the natural choices are L1 loss and L2 loss. If the measurement is non-linear, typically you want to linearized it first (e.g. if the output is exponential, you'd use *log(x)* instead of *x*). See Linear regression.
  - L2 loss is very common but is sensitive to outliers (loss increases as *x^2* at large *x*); L1 loss avoids that, but its gradient has a discontinuity at 0.
- For area-based outputs, people use IoU-related loss functions.
- And there are many other loss functions, e.g. focal loss which is optimized for cases where the two classes are highly imbalanced.
- Anyway, in the case of `simple_bev`, their loss functions are basically binary cross-entropy and L1/L2 losses (see above).