Occlusion-Net:
2D/3D Occluded Keypoint Localization Using Graph Network

N. Dinesh Reddy, Minh Vo, and Srinivasa G. Narasimhan

CVPR 2019

Minji Kim (2020-28702)

Seoul National University

Graph Convolution Networks :: Final Presentation

Occlusion matters

Any scene has occlusions

self-occlusion

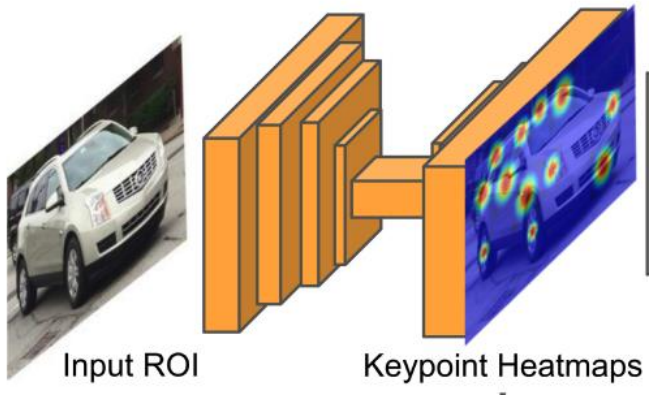
blocked by other objects

Performance ↓

~~detection~~
~~tracking~~
~~reconstruction~~
~~recognition~~
…

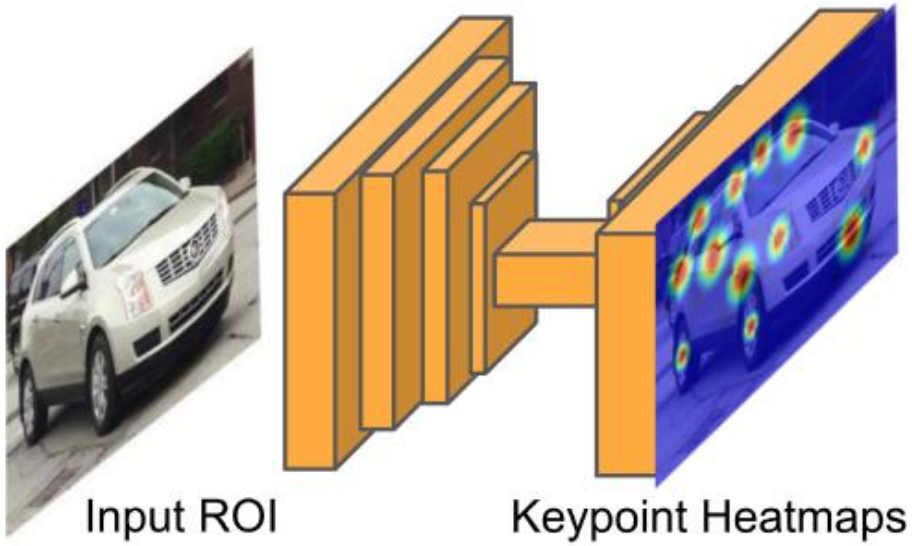
Introduction

- 2D/3D occluded keypoint localization using GNN
- in a largely self-supervised manner
  - Only supervision : visible keypoint annotations
- Off-the-shelf detector (e.g. MaskRCNN) is used as keypoint heatmap generator

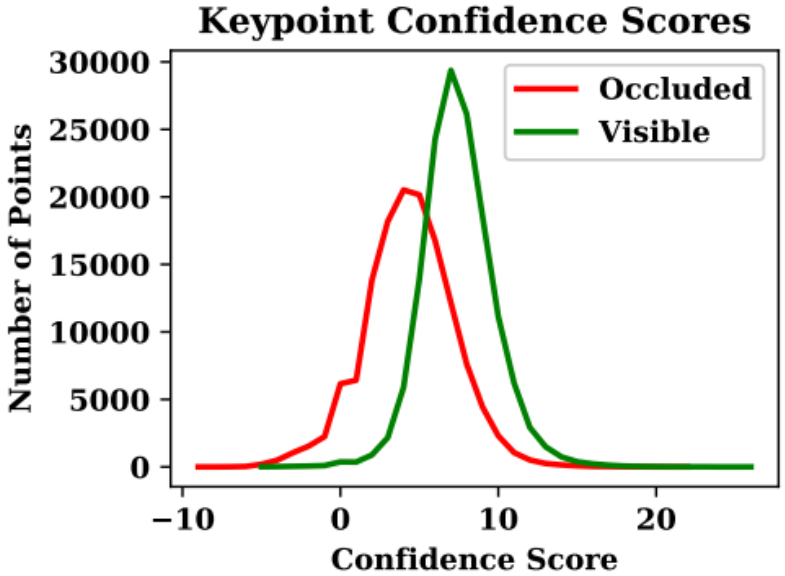


Problems

- How do we find which key points are occluded?
- How do we provide supervision for a hidden-point location?

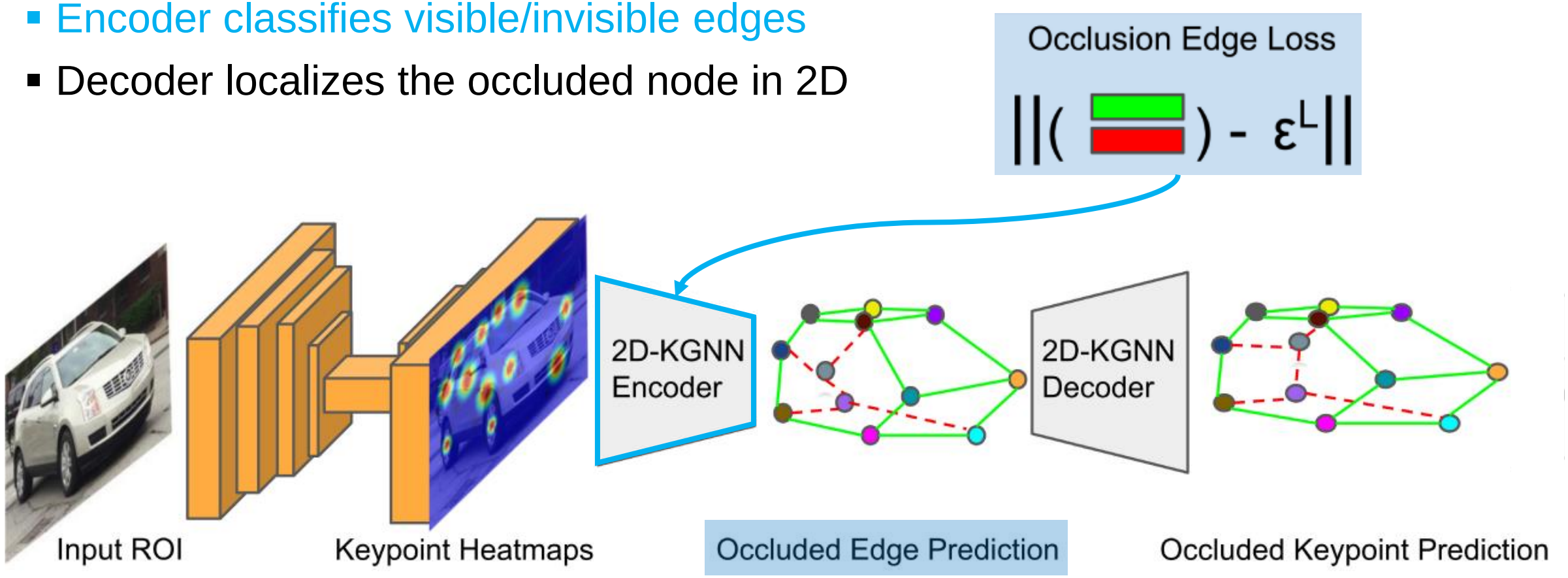


Similar scores!



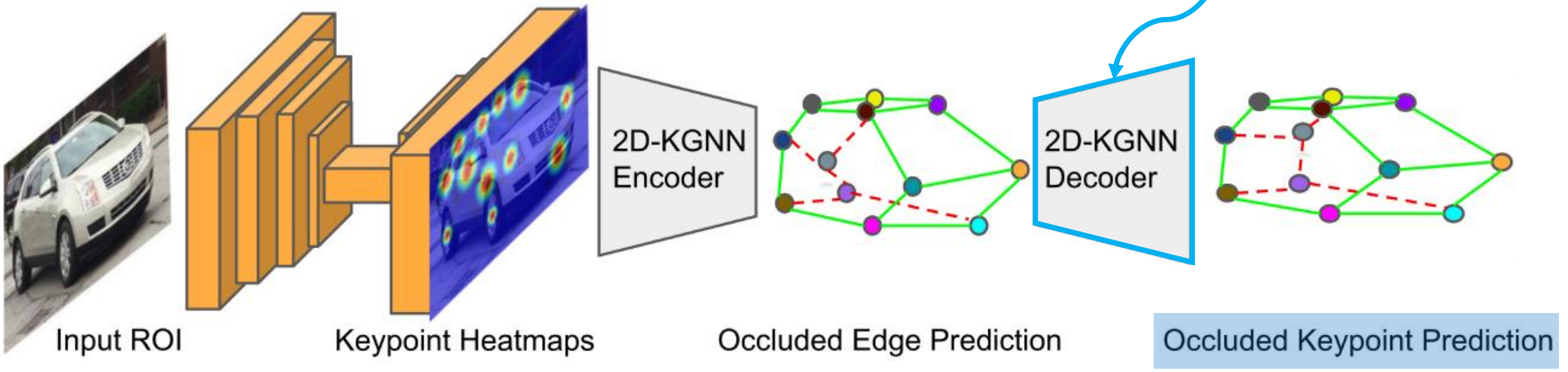
Encoder-decoder graph network

- Encoder classifies visible/invisible edges
- Decoder localizes the occluded node in 2D



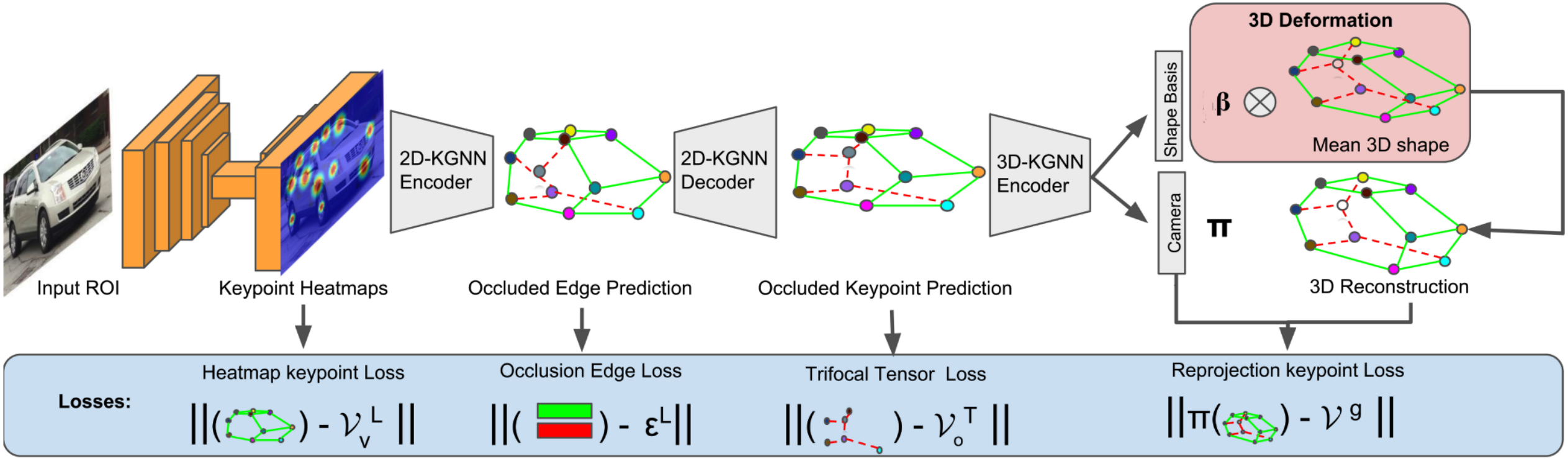
Encoder-decoder graph network

- Encoder classifies visible/invisible edges
- Decoder localizes the occluded node in 2D
  - Loss is computed using multi-views where a keypoint is seen



Occlusion-Net

- Finally, the 3D object shape and camera pose are estimated
- The entire pipeline is trained in end-to-end manner



2D-KGNN Encoder: Occluded Edge Predictor

Vertex (Keypoint)

$\mathcal{V} = (\mathcal{V}_1, \ldots, \mathcal{V}_k)$ for k keypoints

→ v : visible keypoints
→ o : invisible/occluded keypoints

Edge

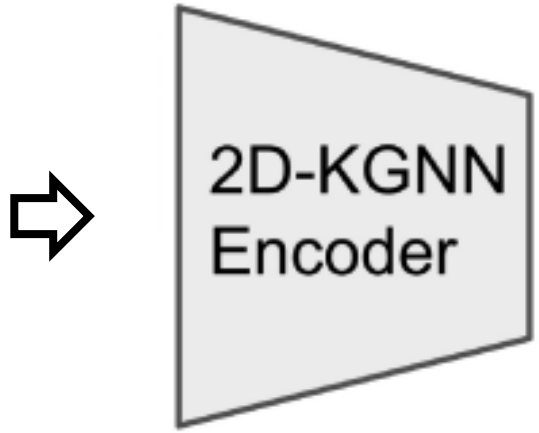
$$\mathcal{E}_{ij} = \{\mathcal{V}_i, \mathcal{V}_j\} = \begin{cases} 1, & if\ i \in v\ and\ j \in v \\ 0, & otherwise \end{cases}$$

2D-KGNN Encoder: Occluded Edge Predictor

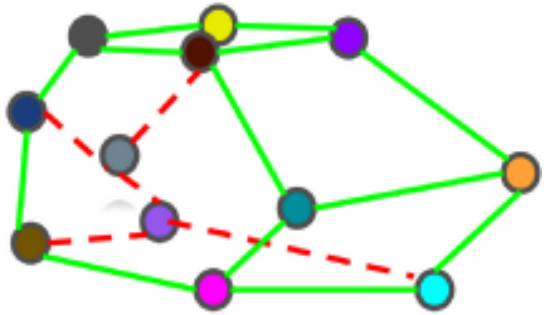
Initial keypoints
encoded from the heatmap

$$\mathcal{V}_i = \{x_i, y_i, c_i, t_i\}$$

(x_i, y_i) : location
c_i : confidence
t_i : type of the keypoint



Latent graph structure



$$q(\mathcal{E}_{ij}|\mathcal{V}) = softmax(f_{enc}(\mathcal{V}))$$

2D-KGNN Encoder: Occluded Edge Predictor

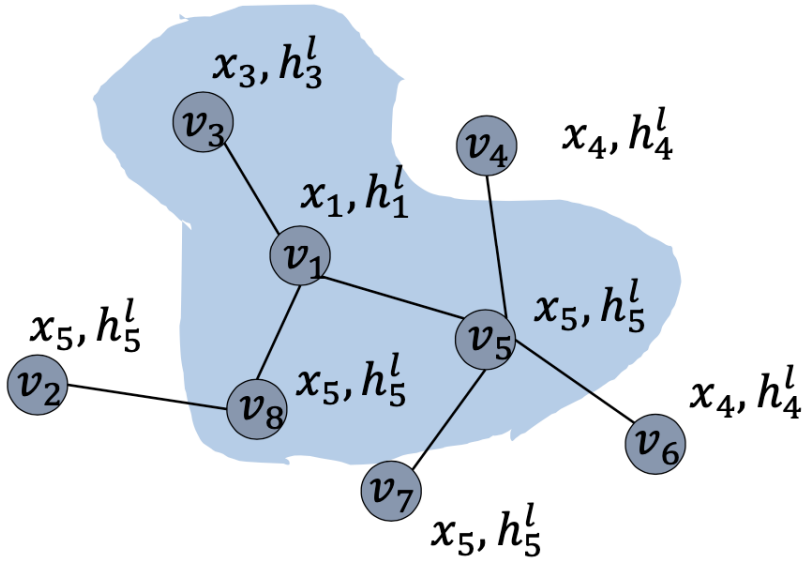
Message passing operations

$$h_j^1 = f_{enc}(\mathcal{V}_j)$$

$$v \rightarrow e: \quad h_{(i,j)}^1 = f_e^1\left([h_i^1, h_j^1]\right)$$

$$e \rightarrow v: \quad h_j^2 = f_v\left(\sum_{i \neq j} h_{(i,j)}^1\right)$$

$$v \rightarrow e: \quad h_{(i,j)}^2 = f_e^2\left([h_i^2, h_j^2]\right)$$

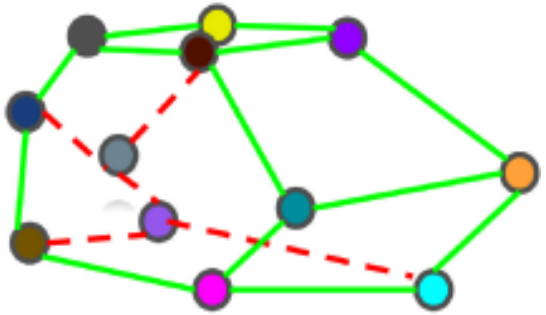


$$L_{Edge} = -\sum_{i,j \in k} \mathcal{E}_{ij} \log(\mathcal{E}_{ij}^l)$$

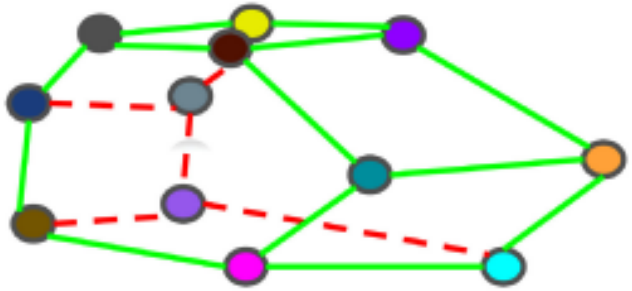
GT Prediction

2D-KGNN Decoder: Occluded Point Predictor

Initial keypoints
+ edges from the encoder



Accurate keypoint locations



$$P_\theta(\mathcal{V}^g|\mathcal{V},\mathcal{E})$$

2D-KGNN Decoder: Occluded Point Predictor

Message passing operations

$$v \rightarrow e: \quad h_{(i,j)} = \sum_{p} \mathcal{E}_{ij,p} f_e^p([\mathcal{V}_i, \mathcal{V}_j])$$

$$e \rightarrow v: \qquad \mu_j^g = \mathcal{V}_j + f_v\left(\sum_{i \neq j} h_{(i,j)}\right)$$

2D-KGNN Decoder: Occluded Point Predictor

Message passing operations

$$v \rightarrow e: \quad h_{(i,j)} = \sum_{p} \mathcal{E}_{ij,p} f_e^p([\mathcal{V}_i, \mathcal{V}_j])$$

$$e \rightarrow v: \quad \mu_j^g = \mathcal{V}_j + f_v\left(\sum_{i \neq j} h_{(i,j)}\right)$$

Current state is added
→Model is learning to deform the keypoint
i.e., predict the difference $\Delta\mathcal{V} = \mathcal{V}^g - \mathcal{V}$

2D-KGNN Decoder: Occluded Point Predictor

Message passing operations

$$v \rightarrow e: \; h_{(i,j)} = \sum_{p} \mathcal{E}_{ij,p} f_e^p([\mathcal{V}_i, \mathcal{V}_j])$$

$$e \rightarrow v: \quad \mu_j^g = \mathcal{V}_j + f_v\left(\sum_{i \neq j} h_{(i,j)}\right)$$

Current state is added
→Model is learning to deform the keypoint
i.e., predict the difference $\Delta\mathcal{V} = \mathcal{V}^g - \mathcal{V}$

➔ $P_\theta(\mathcal{V}^g|\mathcal{V}, \mathcal{E}) = \mathcal{N}(\mu_j^g, \rho^2 I)$

2D-KGNN Decoder: Occluded Point Predictor

Prediction

Trifocal tensor

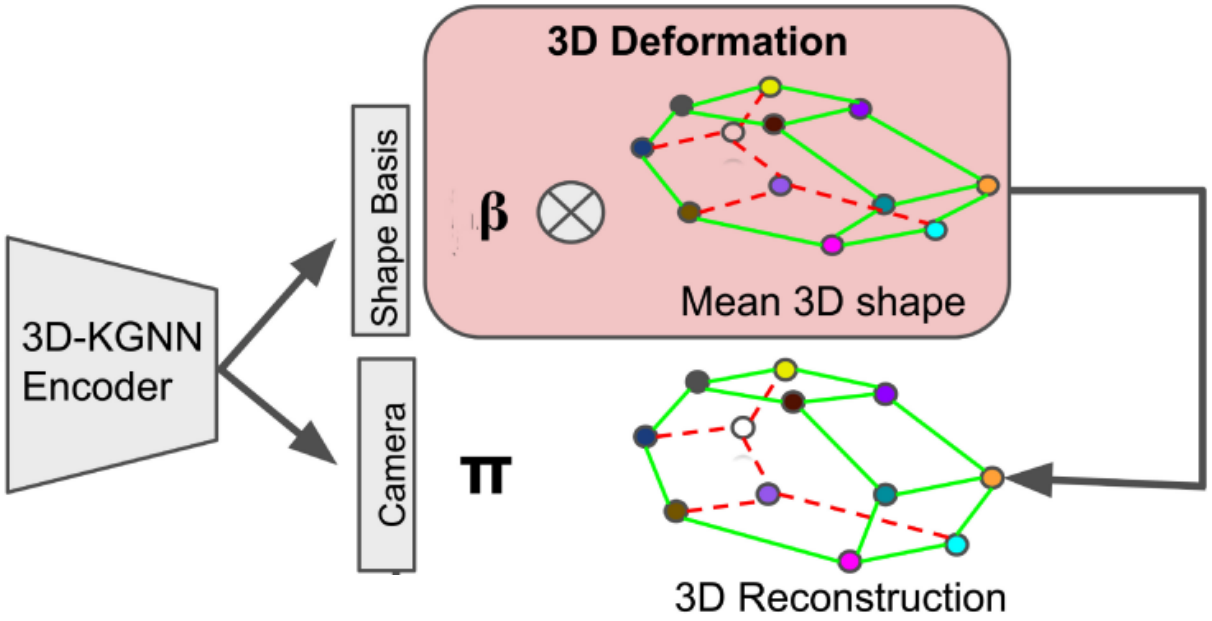
$$L_{Trifocal} = \sum_{j \in o} \left[\mathcal{V}_j^g \right]_\times \left(\sum_i (\mathcal{V}')_j^i T_i \right) \left[\mathcal{V}_j'' \right]_\times$$

GT (another views)

$$[\mathrm{x}']_\times \left(\sum_i x_i T_i \right) [\mathrm{x}'']_\times = 0_{3\times 3}$$

< Trifocal tensor property >

3D-Keypoint GNN



$$q(\beta, \pi | \mathcal{V}) = f_{enc}(\mathcal{V})$$

3D object shape W

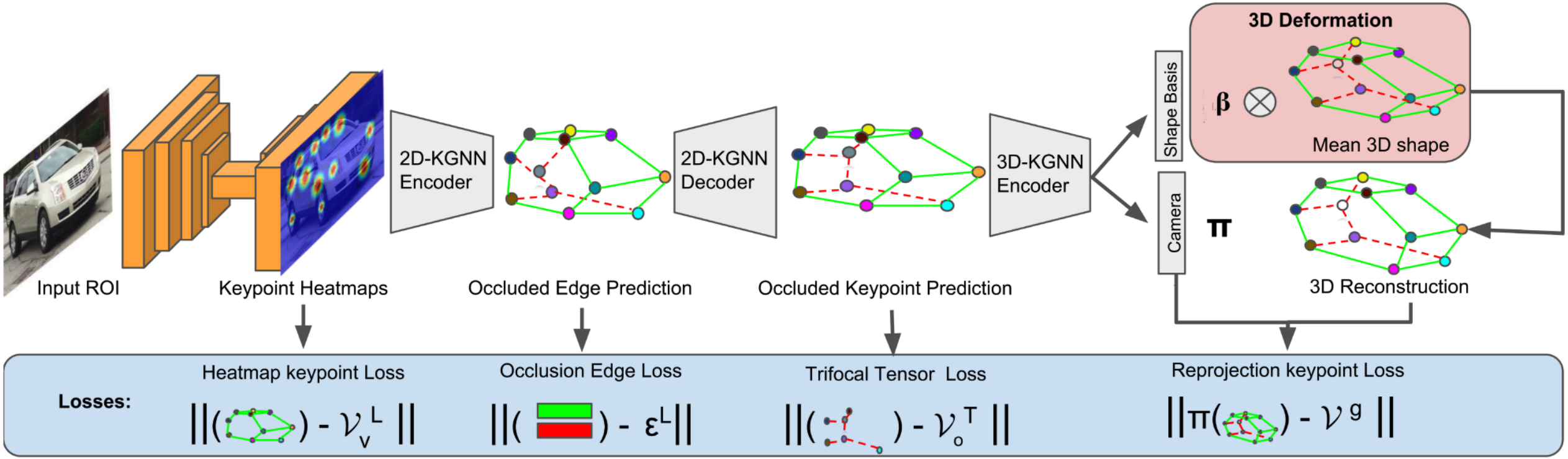
$$W = b_0 + \sum_{k=1}^{n} \beta_k * \sigma_k * b_k$$

b_0 : mean shape
b_j : principal shape components
σ_j : standard deviations
β_j : principal components

Camera projection matrix π

$$L_{Reproj} = \sum_{j \in k} \left\| \pi(W_j) - \mathcal{V}_j^g \right\|^2$$

Total Loss



$$L = L_{Keypoints} + L_{Edge} + L_{Trifocal} + L_{Reproj}$$

Experimental Results

- Datasets
  - Car-render self-occlusion dataset using CAD model
  - CarFusion dataset

- Evaluation
  - PCK metric
    → a keypoint is considered correct if it lies within the radius αL of the GT

  - To evaluate the 3D reconstruction
    → Project the reconstructed keypoints and compute the 2D PCK error

Experimental Results

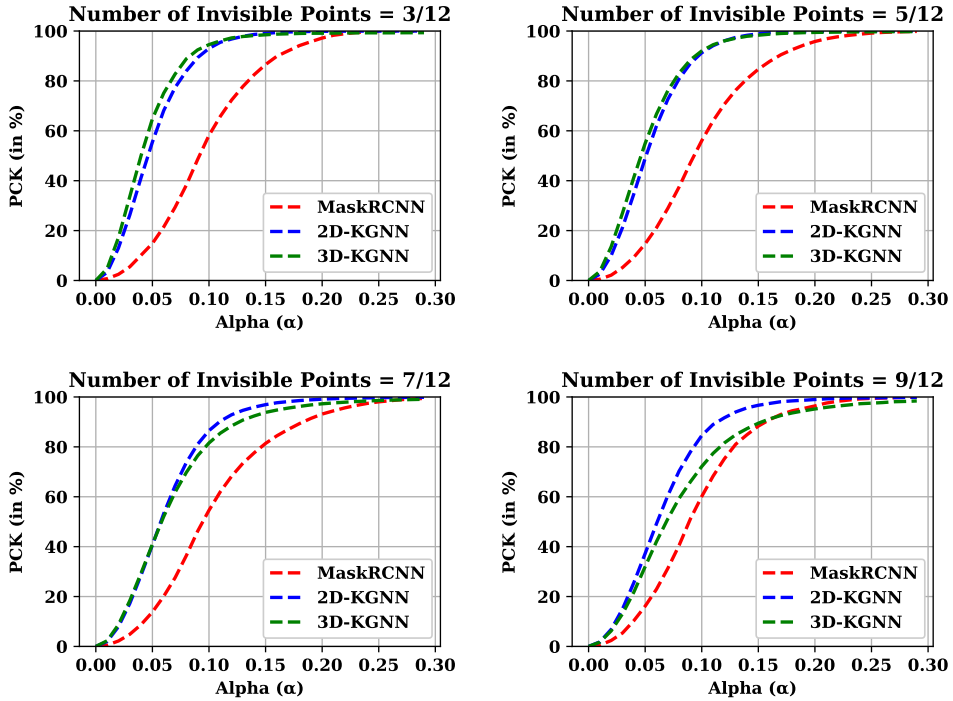


Figure 5: Accuracy with respect to different alpha values of PCK for the Car-render dataset. Graph based methods (2D/3D) outperform the MaskRCNN trained keypoints for all the occlusion types. Specifically at alpha=0.1 we observe an increase of 22% for cases with 3 invisible points and 10% in case of 9 invisible points (out of 12 keypoints).

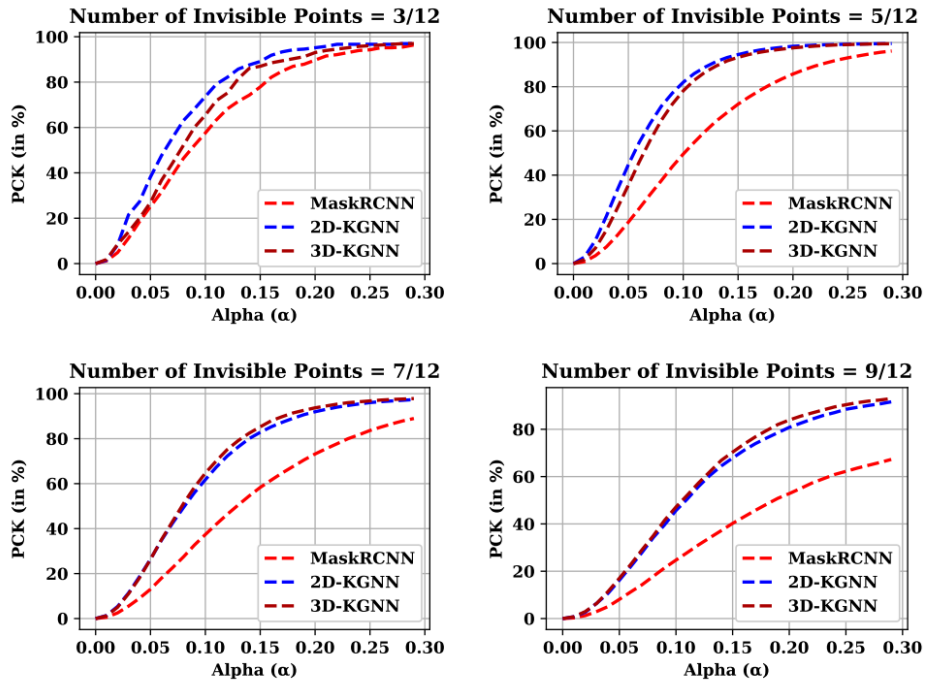


Figure 8: Accuracy vs Alpha on the CarFusion dataset. Focusing on Alpha=0.1 across the plots, graph based methods show an improvement of 6% for cases where only 3 (out of 12) points are occluded and nearly 10% or more improvement for more severe occlusion, justifying the usage of graph networks for occlusion modeling.

Experimental Results

Method	2D					3D	yaw(Error)
	Full	Truncation	Car-Occ	Oth-Occ	All	Full	Full
[18]	88.0	76.0	81.0	82.7	82.0	NA	
[52]	73.6	NA				73.5	7.3
[25]	**93.1**	78.5	**82.9**	85.3	85.0	**95.3**	2.2
Ours	89.73	**87.41**	81.68	**86.45**	**88.8**	93.2	**1.9**

Table 1: PCK Evaluation[α=0.1] and comparison of Occlusion-Net on 2D *visible* keypoints annotated in KITTI-3D. Full denotes unoccluded cars, Truncation denotes cars not fully contained in the image, Car-Occ denotes cars occluded by cars, and Oth-Occ denotes cars occluded by other objects. All represents combining the statistics for all the occlusion categories. Our method outperforms in most of the occlusion categories. The 3D keypoint localization (last two columns) in [25] is only evaluated on Full.

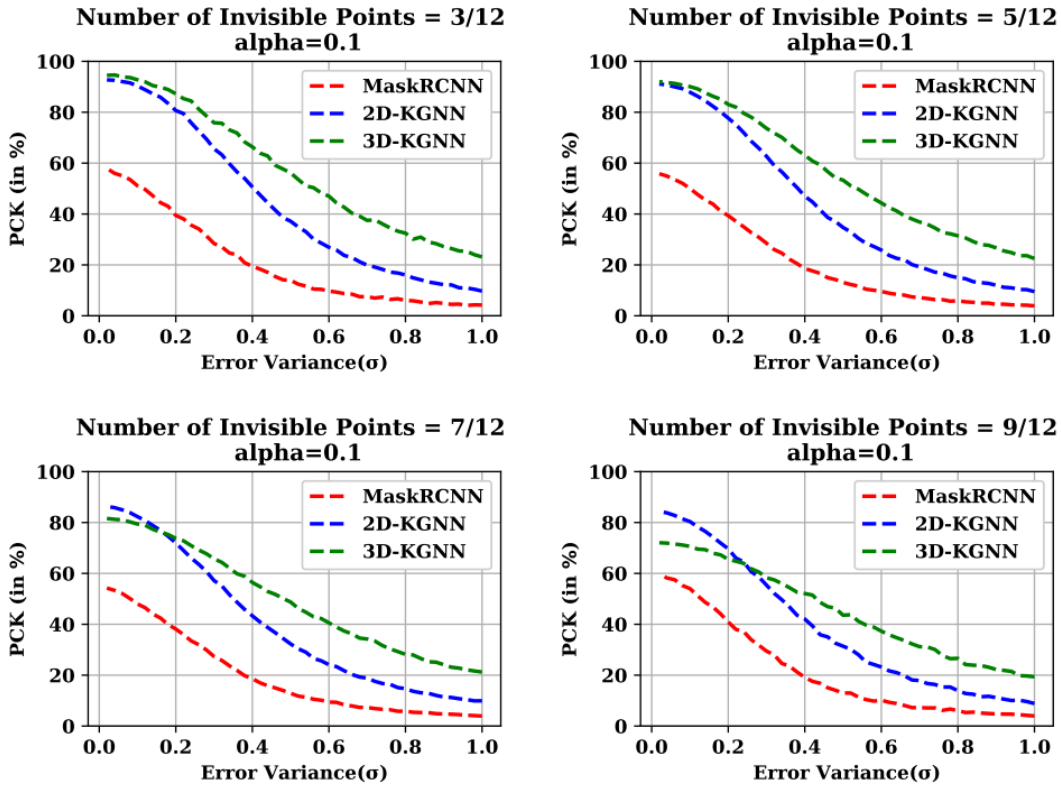
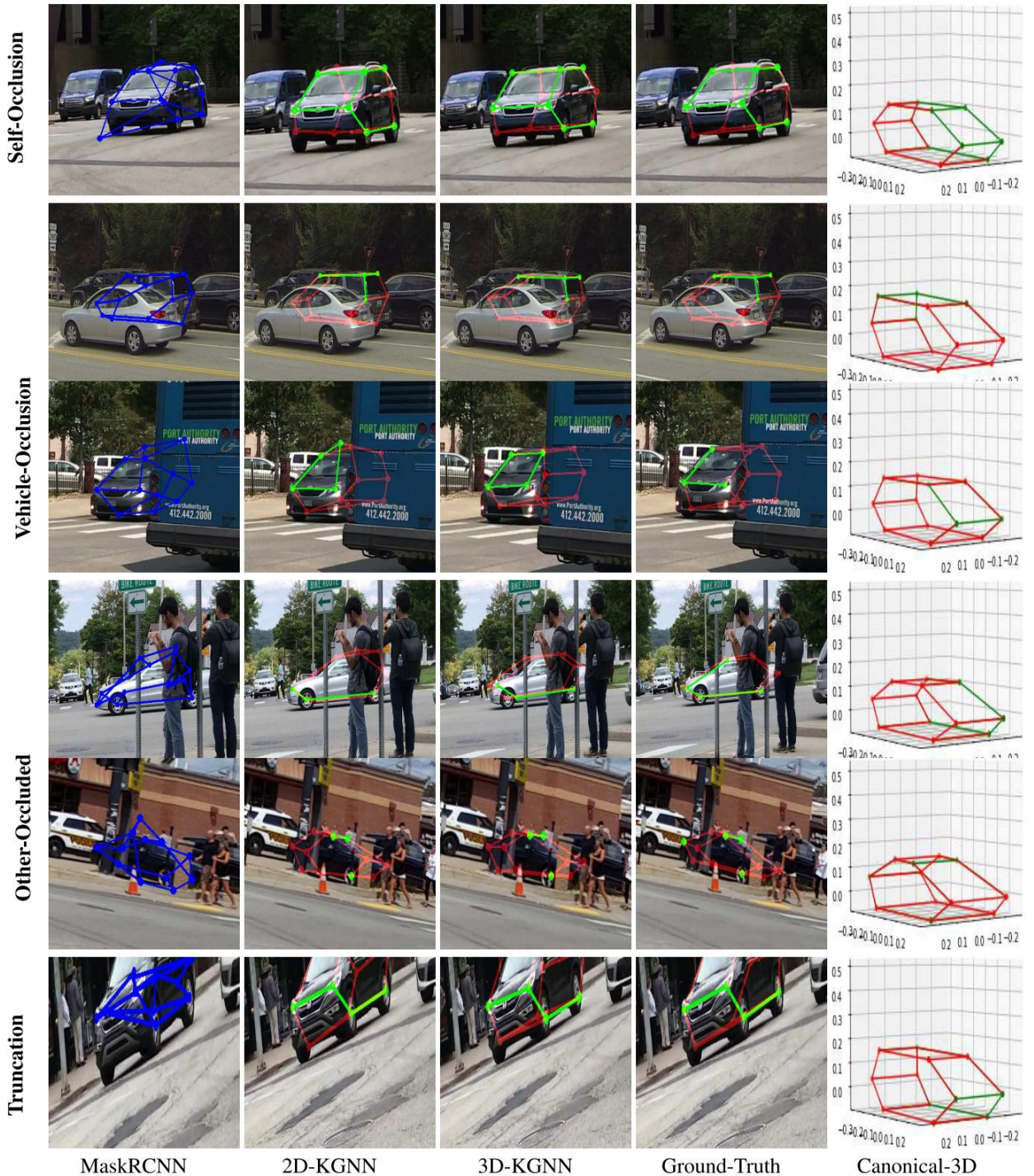


Figure 10: The plots depict the change in accuracy for the methods when Gaussian noise is added to the input keypoints. As expected, 3D-KGNN (green) performs much better in the presence of strong noise.

Experimental Results



구현 및 실험결과

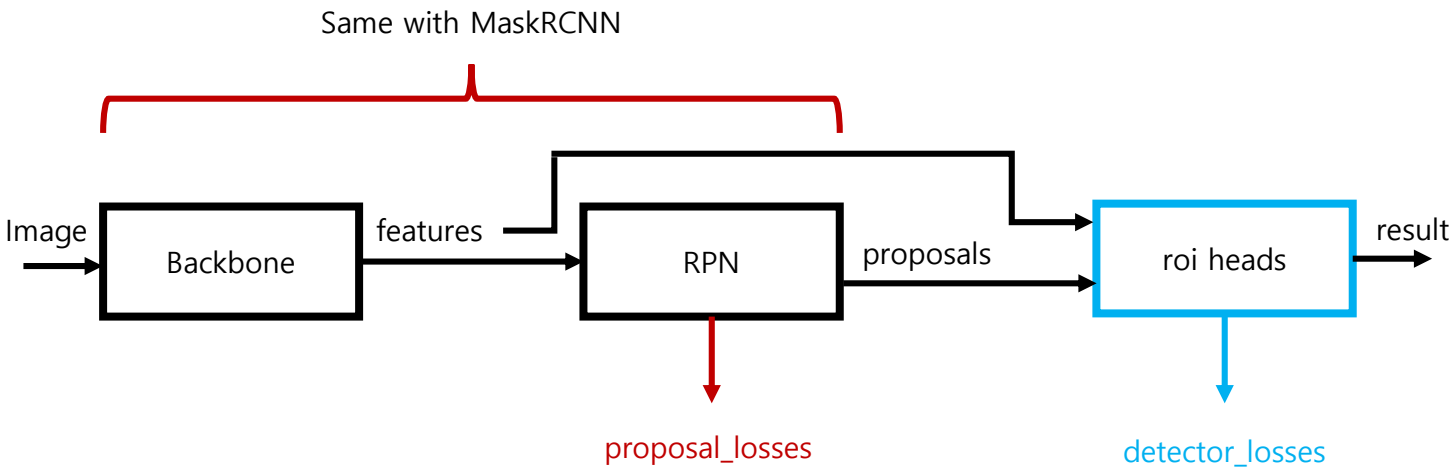
Reproducing

- 코드는 저자 github에 공개된 것을 활용하였으며, 실험을 위해 argument와 image parsing 등 일부만을 수정하였음

- 원저자 Github link
  - https://github.com/dineshreddy91/Occlusion_Net

- 본 project에서의 Reproducing 결과 Github link
  - https://github.com/minggii/Occlusion_Net

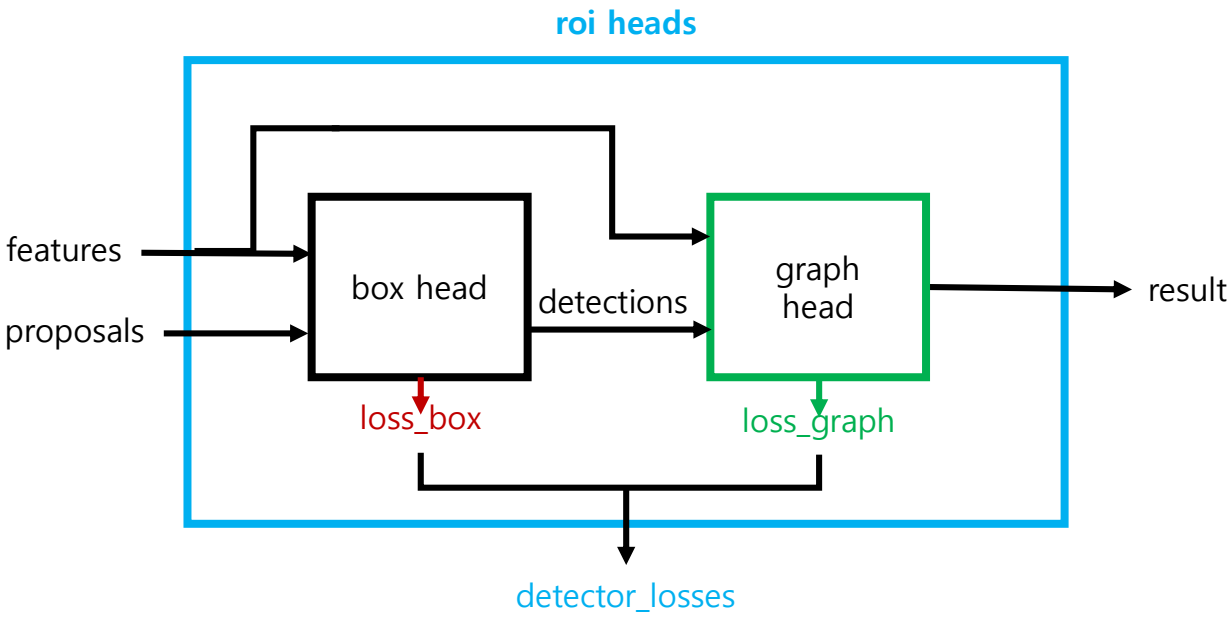
코드 구조




```python
class GeneralizedRCNN(nn.Module):
    """
    Main class for Generalized R-CNN. Currently supports boxes and masks.
    It consists of three main parts:
    - backbone
    - rpn
    - heads: takes the features + the proposals from the RPN and computes
        detections / masks from it.
    """

    def __init__(self, cfg):
        super(GeneralizedRCNN, self).__init__()

        self.backbone = build_backbone(cfg)
        self.rpn = build_rpn(cfg, self.backbone.out_channels)
        self.roi_heads = build_roi_heads(cfg, self.backbone.out_channels)

    def forward(self, images, targets=None):
        """
        Arguments:
            images (list[Tensor] or ImageList): images to be processed
            targets (list[BoxList]): ground-truth boxes present in the image (optional)

        Returns:
            result (list[BoxList] or dict[Tensor]): the output from the model.
                During training, it returns a dict[Tensor] which contains the losses.
                During testing, it returns list[BoxList] contains additional fields
                like `scores`, `labels` and `mask` (for Mask R-CNN models).

        """
        if self.training and targets is None:
            raise ValueError("In training mode, targets should be passed")
        images = to_image_list(images)
        features = self.backbone(images.tensors)
        proposals, proposal_losses = self.rpn(images, features, targets)
        if self.roi_heads:
            x, result, detector_losses = self.roi_heads(features, proposals, targets)
        else:
            # RPN-only models don't have roi_heads
            x = features
            result = proposals
            detector_losses = {}

        if self.training:
            losses = {}
            losses.update(detector_losses)
            losses.update(proposal_losses)
            return losses

        return result
```

- Implemented in lib/occlusion_net/detector/generalized_rcnn.py
- Backbone과 RPN은 MaskRCNN에서 사용하는 것과 동일함
- proposal_losses와 detector_losses를 각 모듈에서 계산하고, 이를 main문에서 backprop하여 학습하는 구조

RoI Heads

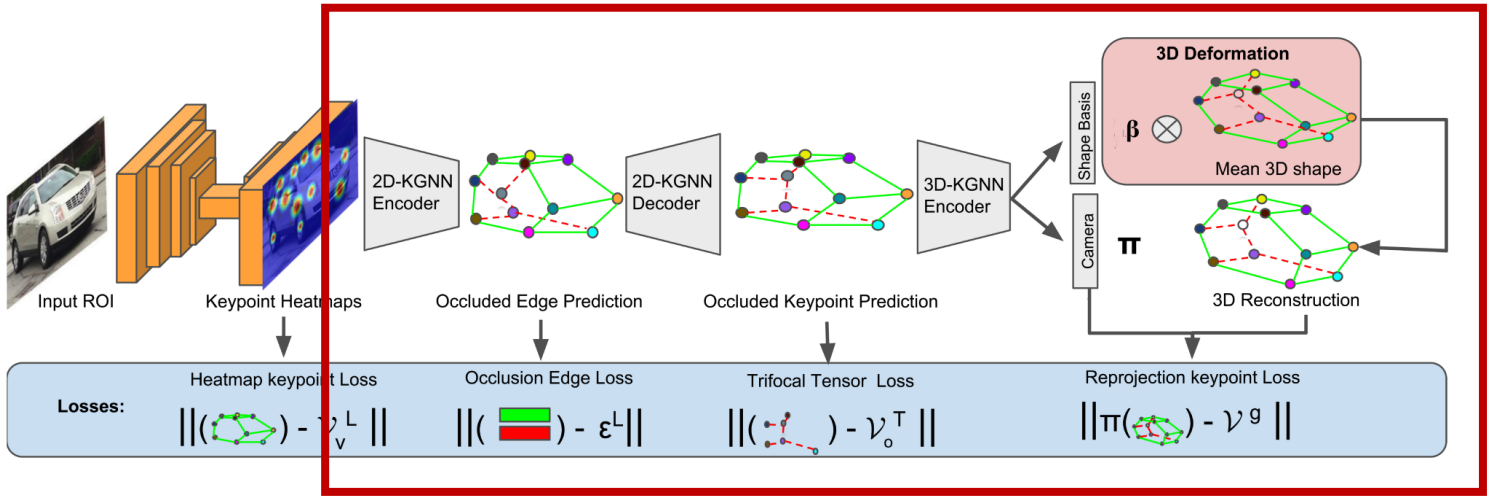



```python
def build_roi_heads(cfg, in_channels):
    # individually create the heads, that will be combined together
    # afterwards
    roi_heads = []
    if cfg.MODEL.RETINANET_ON:
        return []

    if not cfg.MODEL.RPN_ONLY:
        roi_heads.append(("box", build_roi_box_head(cfg, in_channels)))
    if cfg.MODEL.MASK_ON:
        roi_heads.append(("mask", build_roi_mask_head(cfg, in_channels)))
    if cfg.MODEL.KEYPOINT_ON:
        roi_heads.append(("keypoint", build_roi_keypoint_head(cfg, in_channels)))
    if cfg.MODEL.KEYPOINT_ON and cfg.MODEL.GRAPH_ON:
        roi_heads.append(("graph", build_roi_graph_head(cfg, in_channels)))

    # combine individual heads in a single module
    if roi_heads:
        roi_heads = CombinedROIHeads(cfg, roi_heads)

    return roi_heads
```

- Implemented in lib/occlusion_net/roi_heads/roi_heads.py
- box head는 maskRCNN과 동일하게 proposal로부터 (keypoint) detection 결과를 제공
- graph head에서는 keypoint detection 결과를 이용함. encoder-decoder graph network을 거쳐 최종적으로 occluded keypoint localization 결과를 제공
- 코드상 mask head, keypoint head는 기존 MaskRCNN framework과 동일하며 비교실험을 위해 구현되었음

Graph Head

```python
class ROIGraphHead(torch.nn.Module):
    def __init__(self, cfg, in_channels):
        super(ROIGraphHead, self).__init__()
        self.cfg = cfg.clone()
        self.predictor_heatmap = make_roi_keypoint_predictor(
            cfg, cfg.MODEL.ROI_KEYPOINT_HEAD.CONV_LAYERS[-1])
        self.feature_extractor = make_roi_graph_feature_extractor(cfg)
        self.feature_extractor_heatmap = make_roi_keypoint_feature_extractor(cfg, in_channels)
        self.post_processor = make_roi_graph_post_processor(cfg)
        self.post_processor_heatmap = make_roi_keypoint_post_processor(cfg)
        self.loss_evaluator_heatmap = make_roi_keypoint_loss_evaluator(cfg)
        self.loss_evaluator = make_roi_graph_loss_evaluator(cfg)
        self.edges = cfg.MODEL.ROI_GRAPH_HEAD.EDGES
        self.KGNN2D = cfg.MODEL.ROI_GRAPH_HEAD.KGNN2D
        self.KGNN3D = cfg.MODEL.ROI_GRAPH_HEAD.KGNN3D

```



- Implemented in lib/occlusion_net/roi_heads/graph_head/graph_head.py
- 2D-KNN encoder, 2D-KGNN decoder, 3D-KGNN encoder 등이 구현됨

lib/occlusion_net/roi_heads/graph_head/graph_head.py

```python
def forward(self, features, proposals, targets=None):
```

```python
    if self.training:
        with torch.no_grad():
            proposals = self.loss_evaluator_heatmap.subsample(proposals, targets)
    ## heatmaps computation
    # features, proposals -> keypoint heatmap
    x = self.feature_extractor_heatmap(features,proposals)
    kp_logits = self.predictor_heatmap(x)

    if x.shape[0] == 0:
        return torch.zeros((0,x.shape[2],3)),proposals,{}


    ## convert heatmap to graph features
    # heatmap -> keypoints V = (x, y, c, t)
    graph_features = heatmaps_to_graph(kp_logits)
    #x = self.feature_extractor(featu)

    for inc,proposals_per_image in enumerate(proposals):
        proposals_per_image = proposals_per_image.convert("xyxy")
        width = proposals_per_image.bbox[:, 2] - proposals_per_image.bbox[:, 0]
        height = proposals_per_image.bbox[:, 3] - proposals_per_image.bbox[:, 1]
        if inc == 0:
            ratio = width/height
        else:
            ratio = torch.cat((ratio,width/height))

    edge_logits,KGNN2D,KGNN3D = self.feature_extractor(graph_features,ratio)
```

backbone feature와 keypoint detection 결과를 이용하여 keypoint heatmap 생성

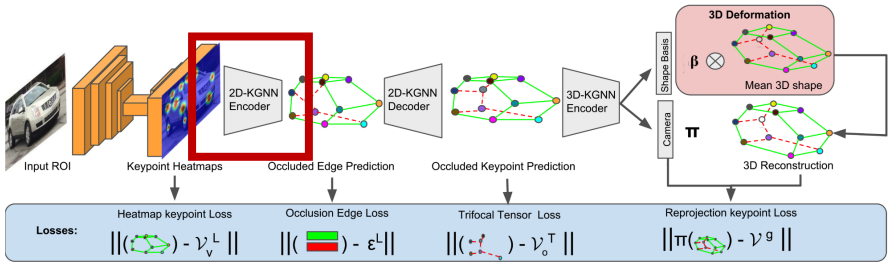
heatmap으로부터 graph feature 생성 (keypoint V = (x, y, c, t))

Encoder-decoder network을 거쳐 edge prediction & keypoint localization

```python
    ## loss computation
    loss_kp = self.loss_evaluator_heatmap(proposals, kp_logits)

    if self.edges == True:
        loss_edges = self.loss_evaluator.loss_edges(valid_vis_all, edge_logits)
        loss_dict_all = dict(loss_edges=loss_edges,loss_kp=loss_kp)
    if self.KGNN2D ==True:
        loss_trifocal = self.loss_evaluator.loss_kgnn2d(keypoints_gt,valid_invis_all, KGNN2D)
        loss_dict_all = dict(loss_edges=loss_edges,loss_kp=loss_kp,loss_trifocal=loss_trifocal)
    if self.KGNN3D ==True:
        valid_all = (valid_vis_all+valid_invis_all)*0 +1
        valid_all[:,-1] = valid_all[:,-1]*0 # dont compute loss in kgnn3d for center point
        valid_all[:,8] = valid_all[:,8]*0 # dont compute the loss for exhaust
        loss_kgnn3d = self.loss_evaluator.loss_kgnn3d(KGNN2D, valid_all, KGNN3D)
        loss_dict_all = dict(loss_edges=loss_edges,loss_kp=loss_kp,loss_trifocal=loss_trifocal,loss_kgnn3d=loss_kgnn3d

    return KGNN2D, proposals, loss_dict_all
```

Loss computation
- loss_edges : occlusion edge loss
- loss_trifocal : trifocal tensor loss
- loss_kgnn3d : reprojection keypoint loss

2D-KGNN Encoder




```python
class GraphEncoder(nn.Module):
    def __init__(self, n_in, n_hid, n_out, do_prob=0., factor=True):
        super(GraphEncoder, self).__init__()

        self.factor = factor

        self.mlp1 = MLP(n_in, n_hid, n_hid, do_prob)
        self.mlp2 = MLP(n_hid * 2, n_hid, n_hid, do_prob)
        self.mlp3 = MLP(n_hid, n_hid, n_hid, do_prob)
        if self.factor:
            self.mlp4 = MLP(n_hid * 3, n_hid, n_hid, do_prob)
            print("Using factor graph MLP encoder.")
        else:
            self.mlp4 = MLP(n_hid * 2, n_hid, n_hid, do_prob)
            print("Using MLP graph encoder.")
        self.fc_out = nn.Linear(n_hid, n_out)
        self.init_weights()

    def init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_normal_(m.weight.data)
                m.bias.data.fill_(0.1)

    def edge2node(self, x, rel_rec, rel_send):
        # NOTE: Assumes that we have the same graph across all samples.
        incoming = torch.matmul(rel_rec.t(), x)
        return incoming / incoming.size(1)

    def node2edge(self, x, rel_rec, rel_send):
        # NOTE: Assumes that we have the same graph across all samples.
        # ex) u -> v
        receivers = torch.matmul(rel_rec, x)# v
        senders = torch.matmul(rel_send, x)# u
        edges = torch.cat([receivers, senders], dim=2)# v, u
        return edges
```

MLP : 2-layer fully connected ELU network with batch norm

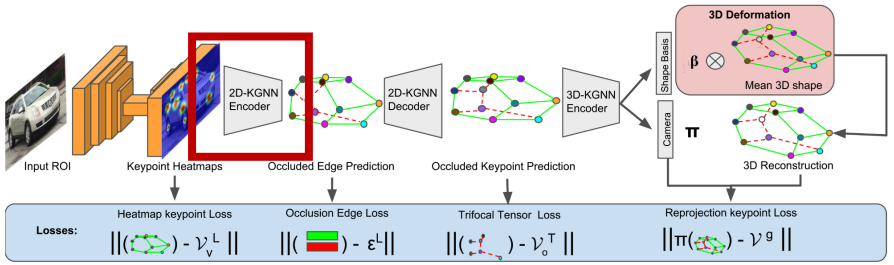
$$e \rightarrow v: \quad h_j^{t+1} = f_v\left(\sum_{i \neq j} h_{(i,j)}^t\right)$$

< edge to node >

$$v \rightarrow e: \quad h_{(i,j)}^t = f_e^t\left([h_i^t, h_j^t]\right)$$

< node to edge >

2D-KGNN Encoder




```python
    def forward(self, inputs, rel_rec, rel_send):
        # Input shape: [num_sims, num_atoms, num_timesteps, num_dims]
        x = inputs.view(inputs.size(0), inputs.size(1), -1)
        # New shape: [num_sims, num_atoms, num_timesteps*num_dims]
        x = self.mlp1(x)  # 2-layer ELU net per node

        x = self.node2edge(x, rel_rec, rel_send)
        x = self.mlp2(x)
        x_skip = x

        if self.factor:
            x = self.edge2node(x, rel_rec, rel_send)
            x = self.mlp3(x)
            x = self.node2edge(x, rel_rec, rel_send)
            x = torch.cat((x, x_skip), dim=2)  # Skip connection
            x = self.mlp4(x)
        else:
            x = self.mlp3(x)
            x = torch.cat((x, x_skip), dim=2)  # Skip connection
            x = self.mlp4(x)

        return self.fc_out(x)
```

$$h_j^1 = f_{enc}(\mathcal{V}_j)$$

$$v \to e: \quad h_{(i,j)}^1 = f_e^1\left([h_i^1, h_j^1]\right)$$

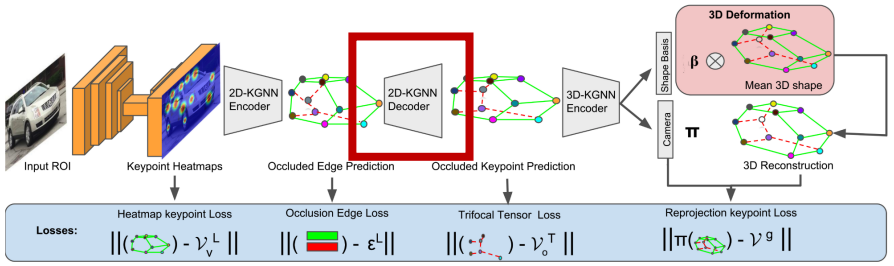
$$e \to v: \quad h_j^2 = f_v\left(\sum_{i \neq j} h_{(i,j)}^1\right)$$

$$v \to e: \quad h_{(i,j)}^2 = f_e^2\left([h_i^2, h_j^2]\right)$$

2D-KGNN Decoder

```python
class GraphDecoder(nn.Module):
```

```python
    def single_step_forward(self, single_timestep_inputs, rel_rec, rel_send,
                            single_timestep_rel_type):

        # single_timestep_inputs has shape
        # [batch_size, num_timesteps, num_atoms, num_dims]

        # single_timestep_rel_type has shape:
        # [batch_size, num_timesteps, num_atoms*(num_atoms-1), num_edge_types]

        # Node2edge
        receivers = torch.matmul(rel_rec, single_timestep_inputs)
        senders = torch.matmul(rel_send, single_timestep_inputs)
        pre_msg = torch.cat([receivers, senders], dim=-1)

        all_msgs = Variable(torch.zeros(pre_msg.size(0), pre_msg.size(1),self.msg_out_shape))
        if single_timestep_inputs.is_cuda:
            all_msgs = all_msgs.cuda()

        if self.skip_first_edge_type:
            start_idx = 1
        else:
            start_idx = 0

        # Run separate MLP for every edge type
        # NOTE: To exlude one edge type, simply offset range by 1
        for i in range(start_idx, len(self.msg_fc2)):
            msg = F.relu(self.msg_fc1[i](pre_msg))
            msg = F.dropout(msg, p=self.dropout_prob)
            msg = F.relu(self.msg_fc2[i](msg))
            msg = msg * single_timestep_rel_type[:, :, i:i + 1]
            all_msgs += msg

        # Aggregate all msgs to receiver
        agg_msgs = all_msgs.transpose(-2, -1).matmul(rel_rec).transpose(-2, -1)
        agg_msgs = agg_msgs.contiguous()

        # Skip connection
        aug_inputs = torch.cat([single_timestep_inputs, agg_msgs], dim=-1)

        # Output MLP
        pred = F.dropout(F.relu(self.out_fc1(aug_inputs)), p=self.dropout_prob)
        pred = F.dropout(F.relu(self.out_fc2(pred)), p=self.dropout_prob)
        pred = self.out_fc3(pred)
#        print(pred.shape,single_timestep_inputs.shape)

        # Predict position/velocity difference
        return single_timestep_inputs + pred
```



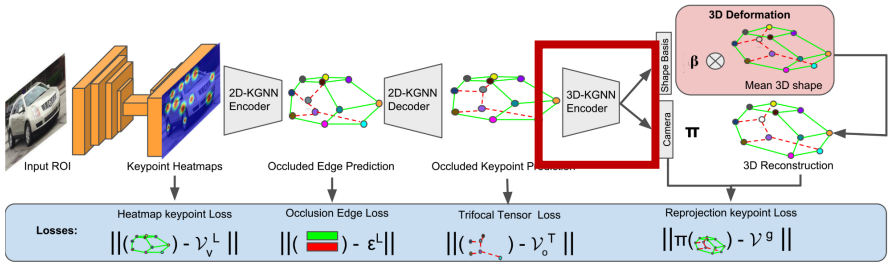
Message passing을 통해 state prediction

$$v \to e: \quad h_{(i,j)} = \sum_{p} \mathcal{E}_{ij,p} f_e^p([\mathcal{V}_i, \mathcal{V}_j])$$

$$e \to v: \quad \mu_j^g = \mathcal{V}_j + f_v\left(\sum_{i \neq j} h_{(i,j)}\right)$$

Skip connection

3D-KGNN Encoder




```python
class GraphEncoder3D(nn.Module):
    def __init__(self, n_in, n_hid, n_out, n_kps, do_prob=0., factor=True):
        super(GraphEncoder3D, self).__init__()

        self.factor = factor

        self.mlp1 = MLP(n_in, n_hid, n_hid, do_prob)
        self.mlp2 = MLP(n_hid * 2, n_hid, n_hid, do_prob)
        self.mlp3 = MLP(n_hid, n_hid, n_hid, do_prob)
        if self.factor:
            self.mlp4 = MLP(n_hid * 3, n_hid, n_hid, do_prob)
            print("Using factor graph MLP encoder.")
        else:
            self.mlp4 = MLP(n_hid * 2, n_hid, n_hid, do_prob)
            print("Using MLP graph encoder.")
        self.fc_out = nn.Linear(n_hid*n_kps, n_out)
        self.flatten = Flatten()
        self.init_weights()

    def init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_normal_(m.weight.data)
                m.bias.data.fill_(0.1)

    def edge2node(self, x, rel_rec, rel_send):
        # NOTE: Assumes that we have the same graph across all samples.
        incoming = torch.matmul(rel_rec.t(), x)
        return incoming / incoming.size(1)

    def node2edge(self, x, rel_rec, rel_send):
        # NOTE: Assumes that we have the same graph across all samples.
        receivers = torch.matmul(rel_rec, x)
        senders = torch.matmul(rel_send, x)
        edges = torch.cat([receivers, senders], dim=2)
        return edges
```

2D-KGNN과 동일한 구조이며 dimension만 다름

```python
    def forward(self, inputs, rel_rec, rel_send):
        # Input shape: [num_sims, num_atoms, num_timesteps, num_dims]
        x = inputs.view(inputs.size(0), inputs.size(1), -1)
        # New shape: [num_sims, num_atoms, num_timesteps*num_dims]

        x = self.mlp1(x)  # 2-layer ELU net per node

        x = self.node2edge(x, rel_rec, rel_send)
        x = self.mlp2(x)
        x_skip = x

        if self.factor:
            x = self.edge2node(x, rel_rec, rel_send)
            x = self.mlp3(x)
            x = self.node2edge(x, rel_rec, rel_send)
            x = torch.cat((x, x_skip), dim=2)  # Skip connection
            x = self.mlp4(x)
            x = self.edge2node(x, rel_rec, rel_send)
            x = self.flatten(x)
        else:
            x = self.mlp3(x)
            x = torch.cat((x, x_skip), dim=2)  # Skip connection
            x = self.mlp4(x)
            x = self.edge2node(x, rel_rec, rel_send)
            x = self.flatten(x)

        return self.fc_out(x)
```

실험 결과

- Training
  - Learning rate = 0.00025 with weight decay = 0.0001
  - Max iteration = 220000
  - Using CarFusion dataset – train set

```
loss_classifier: 0.0235 (0.0471)  loss_box_reg: 0.0089 (0.0282)  loss_edges: 0.0819 (0.1478)  loss_kp:
2.8448 (3.2825)  loss_trifocal: 0.0231 (0.0608)  loss_objectness: 0.0003 (0.0042)  loss_rpn_box_reg:
0.0007 (0.0037)  time: 0.1916 (0.4864)  data: 0.0043 (0.2811)  lr: 0.000250  max mem: 4406
2020-06-17 15:02:32,928 maskrcnn_benchmark.utils.checkpoint INFO: Saving checkpoint to ./log/
model_0220000.pth
2020-06-17 15:02:33,632 maskrcnn_benchmark.utils.checkpoint INFO: Saving checkpoint to ./log/
model_final.pth
2020-06-17 15:02:34,347 maskrcnn_benchmark.trainer INFO: Total training time: 1 day, 6:06:57.843496
(0.4928 s / it)
```

model_0200000.pth	581.1 MB	2020/06/17 9:19 PM
model_0205000.pth	581.1 MB	2020/06/17 10:00 PM
model_0210000.pth	581.1 MB	2020/06/17 10:41 PM
model_0215000.pth	581.1 MB	2020/06/17 11:21 PM
model_0220000.pth	581.1 MB	Yesterday 12:02 AM
model_final.pth	581.1 MB	Yesterday 12:02 AM

실험 결과

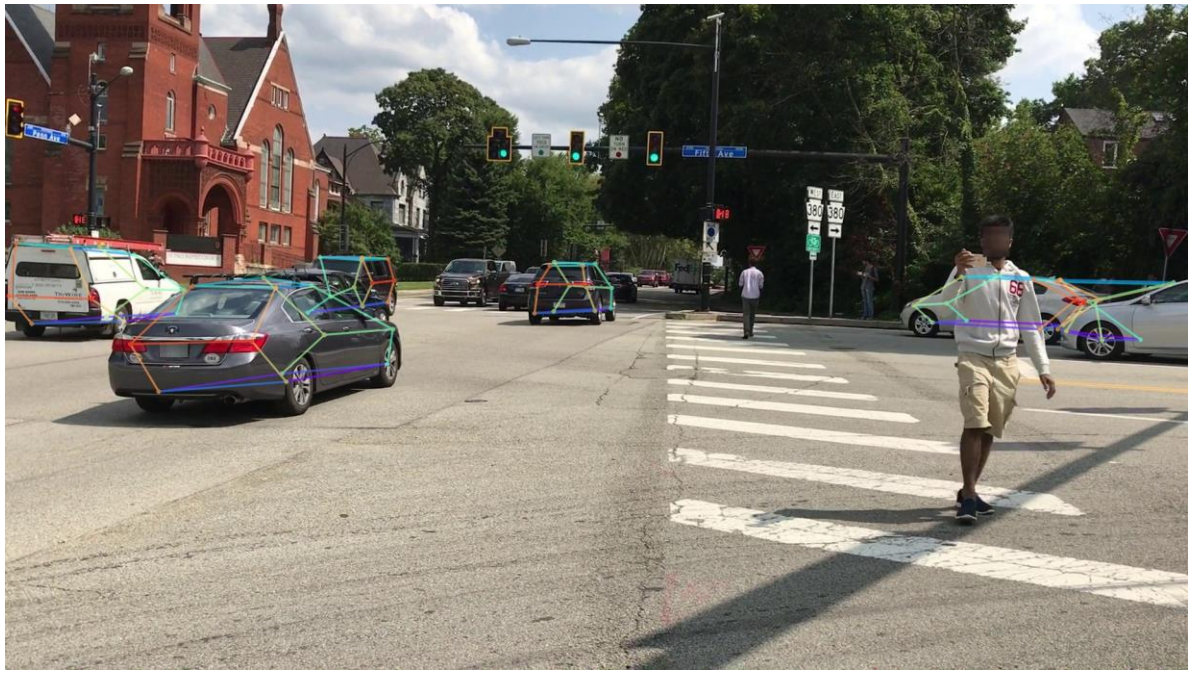
- Test
  - CarFusion dataset – test set (car_penn1, car_penn2) 사용
  - Evaluation은 따로 구현하지 못하여 2D image에서의 visualization 결과만 첨부

실험 결과

- Success case

실험 결과

- Success case



실험 결과

- Failure case

Detection에 실패한 경우

겹쳐진 객체의 keypoint를 검출한 경우

Discussion

- 대체로 정확한 keypoint를 찾았지만, image에 따라 차량이 오검출되거나 붙어 있는 차량의 boundary를 잘못 잡는 경우가 있었다.
- 이유는 여러가지가 있겠지만, 본 연구가 occluded keypoint localization을 GNN task로 삼은 첫 논문이기 때문에 개선할 여지가 많다고 생각한다.
- 그 중 한가지로, ResNet50을 사용하는 backbone network에 비해 encoder-decoder network으로 구성된 roi heads는 shallow하기 때문에 파라미터의 비율이 맞지 않는데, 이것이 학습에 안좋은 방향의 영향을 미칠 것 같다는 생각이 들었다.
- 또한 Human gait recognition 등의 다른 task에서도 활용 가능성이 높다고 생각된다.