Architecture detail for each model

The overall architecture can be broken down into two major stages based on the function they perform.

The first stage is the semantic segmentation of each known object in the color image. An encoder-decoder-based architecture is used for the semantic segmentation that first takes the color image as input then encodes the information into smaller dimension features then decodes them into ***(N + 1)*** channeled segmentation map, where ***N***=number of objects in the dataset and each channel is a binary mask and true pixels indicate the presence of that particular object. One extra channel is to denote the background or no object. This semantic segmentation map is used to generate a bounding box which in turn is used to extract cropped patches of the object. Then, this patch along with the masked cloud or masked depth pixel is forwarded to the second stage.
The encoder part of the model comprises five encoding layers, where each layer contains convolutional layers with different filter sizes and channel numbers. These layers are followed by batch normalization and ReLU activation functions to extract features from the input image at various scales. After each encoding layer, a max pooling layer is applied to reduce the spatial dimensions of the feature maps while preserving important features. The decoder part of the model comprises five decoding layers, where each layer contains upsampling layers followed by convolutional layers with different filter sizes and channel numbers. These layers are also followed by batch normalization and ReLU activation functions. The upsampling layers increase the spatial dimensions of the feature maps, while the convolutional layers recover the fine-grained details of the segmentation map. The segmentation map has the same size as the input image, and for each pixel, it shows the probability of belonging to each class.

Architecture detail of Semantic segmentation model

Input		[3,480,640]
Encoder	Layer 1	(conv11): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn11): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv12): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn12): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
	Layer 2	(conv21): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn21): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv22): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn22): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
	Layer 3	(conv31): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn31): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv32): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn32): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv33): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn33): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
	Layer 4	(conv41): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv42): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn42): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv43): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn43): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
	Layer 5	(conv51): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn51): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv52): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn52): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv53): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn53): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
Decoder	Layer 5	(conv53d): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn53d): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

		(conv52d): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn52d): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv51d): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn51d): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
	Layer 4	(conv43d): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn43d): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv42d): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn42d): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv41d): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn41d): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
	Layer 3	(conv33d): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn33d): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv32d): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn32d): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv31d): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn31d): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
	Layer 2	(conv22d): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn22d): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv21d): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn21d): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
	Layer 1	(conv12d): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn12d): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (conv11d): Conv2d(64, 22, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
Output layer		[N+1,480,640]

Output of Semantic segmentation model

Input layer	[3,480,640]	
Encoding Layer 1	Conv2D → BatchNorm → Relu	Maxpool
	[64,480,640]	[64,240,320]
Encoding Layer 2	[128,240,320]	[128,120,160]
Encoding Layer 3	[256,120,160]	[256,60,80]
Encoding Layer 4	[512,60,80]	[512,30,40]
Encoding Layer 5	[512,30,40]	[512,15,20]
Decoding Layer 5	Upsampling	Conv2D → BatchNorm → Relu
	[512,30,40]	[512,30,40]
Decoding Layer 4	[512,60,80]	[256,60,80]
Decoding Layer 3	[256,120,160]	[128,120,160]
Decoding Layer 2	[128,240,320]	[64,240,320]
Decoding Layer 1	[64,480,640]	[N+1,480,640]
Output layer	[N+1, 480,640]	

The second stage is the estimator or predictor stage where actual prediction takes place. It comprises four components that play a critical role in generating the final predictions. Details are as follows:

- CNN-based encoder-decoder architecture that operates on the cropped images generated by the semantic segmentation stage. This architecture maps the color information ***(H x W x 3)*** to a color feature embedding ***(H x W x*** d_{RGB} ***)***. For the encoder part, we use ResNet18, which comprises convolutional layers, pooling layers, batch normalization layers, ReLU, and four residual blocks with a feature map dimension of [64, 128, 256, 512] whereas the decoder consists of PSPModule, bottleneck layer, and four upsampling layers that are being added to increase the resolution of the feature maps.

Architecture detail of CNN model

Input	To CNN is cropped Image [3, H, W] output of segmentation model	
Encoder - Resnet18	(conv1)	Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
	(relu)	ReLU(inplace)
	(maxpool)	MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
	(layer1)	Sequential((0): BasicBlock((conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) (relu): ReLU(inplace) (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)) (1): BasicBlock((conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) (relu): ReLU(inplace) (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)))
	(layer2)	Sequential((0): BasicBlock((conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False) (relu): ReLU(inplace) (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) (downsample): Sequential((0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False))) (1): BasicBlock((conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) (relu): ReLU(inplace) (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)))
	(layer3)	Sequential((0): BasicBlock((conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) (relu): ReLU(inplace) (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) (downsample): Sequential((0): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1), bias=False))) (1): BasicBlock((conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2), bias=False) (relu): ReLU(inplace) (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2, 2), bias=False)))
	(layer4)	Sequential((0): BasicBlock((conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) (relu): ReLU(inplace) (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) (downsample): Sequential((0): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)))

		(1): BasicBlock((conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(4, 4), dilation=(4, 4), bias=False) (relu): ReLU(inplace) (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(4, 4), dilation=(4, 4), bias=False)))
Decoder	(psp): PSPModule ((stages): ModuleList((0): Sequential((0): AdaptiveAvgPool2d(output_size=(1, 1)) (1): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)) (1): Sequential((0): AdaptiveAvgPool2d(output_size=(2, 2)) (1): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)) (2): Sequential((0): AdaptiveAvgPool2d(output_size=(3, 3)) (1): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)) (3): Sequential((0): AdaptiveAvgPool2d(output_size=(6, 6)) (1): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False))) (bottleneck): Conv2d(2560, 1024, kernel_size=(1, 1), stride=(1, 1)) (relu): ReLU()) (drop_1): Dropout2d(p=0.3) (up_1): PSPUpsample((conv): Sequential((0): Upsample(scale_factor=2, mode=bilinear) (1): Conv2d(1024, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (2): PReLU(num_parameters=1))) (up_2): PSPUpsample((conv): Sequential((0): Upsample(scale_factor=2, mode=bilinear) (1): Conv2d(256, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (2): PReLU(num_parameters=1))) (up_3): PSPUpsample((conv): Sequential((0): Upsample(scale_factor=2, mode=bilinear) (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (2): PReLU(num_parameters=1))) (drop_2): Dropout2d(p=0.15) (final): Sequential((0): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1)) (1): LogSoftmax()) (classifier): Sequential((0): Linear(in_features=256, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=21, bias=True))	
Output	[32, H, W]	

Output of CNN model

Input: Cropped Image	[3, 120, 120]	
Encoder - Resnet18	Conv2D	[64, 60, 60]
	Maxpool	[64, 30, 30]
	Layer 1: BasicBlock 1 → BasicBlock 2	[64, 30, 30]
	Layer 2: BasicBlock 1 → BasicBlock 2	[128, 15, 15]
	Layer 3: BasicBlock 1 → BasicBlock 2	[256, 15, 15]
	Layer 4: BasicBlock 1 → BasicBlock 2	[512, 15, 15]
Decoder	PSPModule and Bottleneck	[1024, 15, 15]
	Upsampling Layer 1	[256, 30, 30]
	Upsampling Layer 2	[64, 60, 60]

	Upsampling Layer 3	[64, 120, 120]
	Upsampling Layer 4	[32, 120, 120]
Output	[32, 120, 120]	

- In next component, PointNet like structure is used as a geometric embedding network, that takes in the segmented 3D point cloud ***($P \times 3$)*** of the cropped image as input and gives us the geometric feature embedding by converting each of the segmented points into a feature space of ***($P \times d_{GEO}$)***, creating a dense per-point feature.

- After embedding the features from both the color and point cloud spaces, we combine them to create global features ***($P \times d_{GLOBAL}$)***. This is done by using an MLP network with a symmetric function (in this case average pooling) and then a fusion network performs pixel-wise combination to original per pixel features. So now we have for each pixel color features, geometric features, and global features. This whole process of forming per pixel features is named as Dense Fusion of features in DenseFusion
  Now after obtaining the features, the actual estimation is remaining, for each pixel feature there is a prediction of pose along with a confidence score. Then we predict the final pose in this that has the highest confidence score among all the per-pixel predictions. Unlike Dense Fusion, this approach uses separate networks for symmetric and non-symmetric object types, resulting in improved accuracy.

Architecture detail of PointNet , Dense Fusion and Pose estimator model

Input to PointNet	cropped image embeddings= Emb : [32, 500]	Masked point cloud for cropped imageX: [3, 500]
	(e_conv1): Conv1d(32, 64, kernel_size=(1,), stride=(1,))	(conv1): Conv1d(3, 64, kernel_size=(1,), stride=(1,))
	(e_conv2): Conv1d(64, 128, kernel_size=(1,), stride=(1,))	(conv2): Conv1d(64, 128, kernel_size=(1,), stride=(1,))
Pixel-wise Dense Fusion Network	(conv5): Conv1d(256, 512, kernel_size=(1,), stride=(1,))	
	(conv6): Conv1d(512, 1024, kernel_size=(1,), stride=(1,))	
	(ap1): AvgPool1d(kernel_size=(500,), stride=(500,), padding=(0,))	
Pose estimator Network	For non-symmetric object	For symmetric object
	(conv1_r): Conv1d(1408, 640, kernel_size=(1,), stride=(1,)) (conv1_t): Conv1d(1408, 640, kernel_size=(1,), stride=(1,)) (conv1_c): Conv1d(1408, 640, kernel_size=(1,), stride=(1,))	(conv1_r): Conv1d(1408, 640, kernel_size=(1,), stride=(1,)) (conv1_t): Conv1d(1408, 640, kernel_size=(1,), stride=(1,)) (conv1_c): Conv1d(1408, 640, kernel_size=(1,), stride=(1,))
	(conv2_r): Conv1d(640, 256, kernel_size=(1,), stride=(1,)) (conv2_t): Conv1d(640, 256, kernel_size=(1,), stride=(1,)) (conv2_c): Conv1d(640, 256, kernel_size=(1,), stride=(1,))	(conv2_r): Conv1d(640, 256, kernel_size=(1,), stride=(1,)) (conv2_t): Conv1d(640, 256, kernel_size=(1,), stride=(1,)) (conv2_c): Conv1d(640, 256, kernel_size=(1,), stride=(1,))
	(conv3_r): Conv1d(256, 128, kernel_size=(1,), stride=(1,)) (conv3_t): Conv1d(256, 128, kernel_size=(1,), stride=(1,)) (conv3_c): Conv1d(256, 128, kernel_size=(1,), stride=(1,))	(conv3_r): Conv1d(256, 128, kernel_size=(1,), stride=(1,)) (conv3_t): Conv1d(256, 128, kernel_size=(1,), stride=(1,)) (conv3_c): Conv1d(256, 128, kernel_size=(1,), stride=(1,))
	(conv4_r): Conv1d(128, 64, kernel_size=(1,), stride=(1,)) (conv4_t): Conv1d(128, 64, kernel_size=(1,), stride=(1,)) (conv4_c): Conv1d(128, 64, kernel_size=(1,), stride=(1,))	(conv4_r): Conv1d(128, 52, kernel_size=(1,), stride=(1,)) (conv4_t): Conv1d(128, 39, kernel_size=(1,), stride=(1,)) (conv4_c): Conv1d(128, 13, kernel_size=(1,), stride=(1,))
	(conv5_r): Conv1d(64, 52, kernel_size=(1,), stride=(1,)) (conv5_t): Conv1d(64, 39, kernel_size=(1,), stride=(1,)) (conv5_c): Conv1d(64, 13, kernel_size=(1,), stride=(1,))	

Output of PointNet , Dense Fusion and Pose estimator model

Input to PointNet	cropped image embeddings=Emb : [32, 500]	Masked point cloud for cropped image=X: [3, 500]
	e_conv1 → Emb :[64, 500]	conv1 →X: [64, 500]
	Concat [Emb, X] = [128,500] → pointfeat_1	
	e_conv2 → Emb: [128, 500]	conv2 →X: [128, 500]
	Concat [Emb, X]= [256,500] → pointfeat_2	
Pixel wise Dense Fusion Network	conv5 → pointfeat_2:[512,500]	
	conv6 → pointfeat_2:[1024,500]	
	average pooling → pointfeat_2:[1024,1]=pointfeat_ap:[1024,500]	
	Concat [pointfeat_1, pointfeat_2, pointfeat_ap]= { [128,500] ,[256,500], [1024,500] } =[1408,500] → f (pixel wise features)	
Pose estimator Network	For non-symmetric object	For symmetric object
	Conv1_r: [640, 500], Conv1_t: [640, 500], Conv1_c: [640, 500]	Conv1_r: [640, 500], Conv1_t: [640, 500] Conv1_c: [640, 500]
	Conv2_r: [256, 500], Conv2_t: [256, 500] Conv2_c: [256, 500]	Conv2_r: [256, 500], Conv2_t: [256, 500] Conv2_c: [256, 500]
	Conv3_r: [128, 500], Conv3_t: [128, 500] Conv3_c:[128, 500]	Conv3_r: [128, 500], Conv3_t: [128, 500] Conv3_c:[128, 500]
	Conv4_r: [64, 500], Conv4_t: [64, 500] Conv4_c:[64, 500]	Conv4_r: [52, 500] → Conv4_r:[13, 4, 500] Conv4_t: [39, 500] → Conv4_t:[13, 3, 500] Conv4_c:[13, 500] → Conv4_c:[13, 1, 500]
	Conv5_r: [52, 500] → Conv5_r:[13, 4, 500] Conv5_t: [39, 500] → Conv5_t:[13, 3, 500] Conv5_c:[13, 500] → Conv5_c:[13, 1, 500]	
Output	out_r: [4, 500] , out_t: [3, 500], out_c: [1, 500]	

- The pose prediction is then used as input to an iterative refinement network, which uses a curriculum learning approach. Separate iterative refinement networks are used for each type of object to support their respective estimators, such that each estimator network and iterative refinement network form a separate pair for each type of object.

Architecture detail of Pose Refiner model

Pose RefineNet Feat	cropped image embeddings=Emb : [32, 500]	Masked point cloud for cropped image=X: [3, 500]
	(e_conv1): Conv1d(32, 64, kernel_size=(1,), stride=(1,))	(conv1): Conv1d(3, 64, kernel_size=(1,), stride=(1,))
	(e_conv2): Conv1d(64, 128, kernel_size=(1,), stride=(1,))	(conv2): Conv1d(64, 128, kernel_size=(1,), stride=(1,))
	(conv5): Conv1d(384, 512, kernel_size=(1,), stride=(1,))	
	(conv6): Conv1d(512, 1024, kernel_size=(1,), stride=(1,))	
	(ap1): AvgPool1d(kernel_size=(500,), stride=(500,), padding=(0,))	
Pose RefineNet	For non-symmetric object	For symmetric object

	(conv1_r): Linear(in_features=1024, out_features=512, bias=True) (conv1_t): Linear(in_features=1024, out_features=512, bias=True)	(conv1_r): Linear(in_features=1024, out_features=512, bias=True) (conv1_t): Linear(in_features=1024, out_features=512, bias=True)
	(conv2_r): Linear(in_features=512, out_features=128, bias=True) (conv2_t): Linear(in_features=512, out_features=128, bias=True)	(conv2_r): Linear(in_features=512, out_features=128, bias=True) (conv2_t): Linear(in_features=512, out_features=128, bias=True)
	(conv3_r): Linear(in_features=128, out_features=64, bias=True) (conv3_t): Linear(in_features=128, out_features=64, bias=True)	(conv3_r): Linear(in_features=128, out_features=52, bias=True) (conv3_t): Linear(in_features=128, out_features=39, bias=True)
	(conv4_r): Linear(in_features=64, out_features=52, bias=True) (conv4_t): Linear(in_features=64, out_features=39, bias=True)	

Output of Pose Refiner model

Pose RefineNet Feat	cropped image embeddings=Emb : [32, 500]	Masked point cloud for cropped image=X: [3, 500]
	e_conv1 → Emb :[64, 500]	conv1 → X: [64, 500]
	Concat [Emb, X] = [128,500] → pointfeat_1	
	e_conv2 → Emb :[128, 500]	conv2 → X: [128, 500]
	Concat [Emb, X] = [256,500] → pointfeat_2	
	Concat [pointfeat_1, pointfeat_2] = [384,500] → pointfeat_3	
	conv5 → pointfeat_3:[512,500]	
	conv6 → pointfeat_3:[1024,500]	
	average pooling → pointfeat_3:[1024,1] = pointfeat_ap	
Pose RefineNet	For non-symmetric object	For symmetric object
	Conv1_r: [512, 1] , Conv1_t: [512, 1]	Conv1_r: [512, 1] , Conv1_t: [512, 1]
	Conv2_r: [128, 1] , Conv2_t: [128, 1]	Conv2_r: [128, 1] , Conv2_t: [128, 1]
	Conv3_r: [64, 1] , Conv3_t: [64, 1]	Conv3_r: [52, 1] → Conv3_r: [13, 4, 1], Conv3_t: [39, 1] → Conv3_t: [13, 3, 1]
	Conv4_r: [52, 1] → Conv4_r: [13, 4, 1] Conv4_t: [39, 1] → Conv4_t: [13, 3, 1]	
Output	out_r: [4, 1] , out_t: [3, 1]	