MP4-Q1 by Jongwon Lee (jongwon5)

1. **Implement BaseNet [5 pts].**

Implementation

Below is implementation for BaseNet, a vary simple deep-NN model for image classification.

```
import torch.nn as nn
import torch.nn.functional as F

class BaseNet(nn.Module):
    def __init__(self):
        super(BaseNet, self).__init__()
        # convolutional kernel
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # affine transformation y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        # layer 1 to layer 3
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # layer 4 to layer 6
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        # flatten the output
        x = x.view(-1, 16 * 5 * 5)
        # layer 7 and 8
        x = F.relu(self.fc1(x))
        # layer 9
        x = self.fc2(x)

        return x
```

Result

The above model can be visualized by using `print(net)` command as follows:

```
# Create an instance of the nn.module class defined above:
net = BaseNet()
print(net)
```

By doing so, we are able to obtain all the layers consisting of the model like below. Please note that other operations, such as activation layers or flattening, are not revealed using this function.

```
BaseNet(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=200, bias=True)
  (fc2): Linear(in_features=200, out_features=10, bias=True)
)
```

By following a sequence of training and validation cycle, I was able to observe a set of evaluation results on the validation dataset.

```
Accuracy of the final network on the val images: 60.0 %
Accuracy of airplane : 71.8 %
Accuracy of automobile : 76.8 %
Accuracy of  bird : 45.3 %
Accuracy of   cat : 35.5 %
Accuracy of  deer : 60.5 %
Accuracy of   dog : 43.5 %
Accuracy of  frog : 74.0 %
Accuracy of horse : 64.1 %
Accuracy of  ship : 58.5 %
Accuracy of truck : 70.2 %
```

2. **Improve BaseNet [20 pts].**

Implementation

I came up with VGGNet-like structure due to its simple structure but powerful performance. The detailed implementation is as follow:

```
cfg = {
    '1': [16, 'M', 32, 'M', 64, 'M', 128, 'M'], # 81.2%
    '2': [32, 'M', 64, 'M', 128, 'M', 256, 'M', 512, 'M'], # 86.2%
    '3': [32, 'M', 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M'], #
85.9%
}

class ImprovedNet(nn.Module):
    def __init__(self, model_name):
        super(ImprovedNet, self).__init__()
        # normalize input tensor
        self.normalize_input = transforms.Compose([
              transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                   std=[0.229, 0.224, 0.225])])
        # model's layers
        self.convnet = self._make_layers(cfg[model_name])
        self.fc = nn.Linear(in_features=512, out_features=10)

        # arbitrary variable to save the number of input channels for each
```

```
  layer
          self.in_channels = 3

      def _make_layers(self, cfg):
          layers = []
          for param in cfg:
              if param == 'A':
                  layers += [nn.AvgPool2d(kernel_size=2, stride=2)]
              elif param == 'M':
                  layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
              else:
                  layers += [nn.Conv2d(self.in_channels, param,
  kernel_size=3, padding=1),
                             nn.BatchNorm2d(param),
                             nn.ReLU(inplace=True)]
                  self.in_channels = param
          layers += [nn.AvgPool2d(kernel_size=1, stride=1)]

          return nn.Sequential(*layers)

      def forward(self, x):
          x = self.normalize_input(x)
          x = self.convnet(x)
          x = x.view(x.size(0), -1)
          x = self.fc(x)

          return x
```

Note that `conv2d + batchnorm + relu` and `maxpool2d` are stacked alternatively to reduce the tensor's size gradually. Also, there exists three options to create an ImprovedNet with different depth; ImprovedNet-V1, ImprovedNet-V2, and ImprovedNet-V3 from shallower to deeper.

Architecture for ImprovedNet-V1 (which can be rendered by `net = ImprovedNet('1')`) is provided below:

Layer No.	Layer Type	Kernel Size	Input Dim	Output Dim	Input Channels	Output Channels
1	conv2d + batchnorm + relu	3	32	32	3	16
2	maxpool2d	2	32	16	16	16
3	conv2d + batchnorm + relu	3	16	16	16	32
4	maxpool2d	2	16	8	32	32
5	conv2d + batchnorm + relu	3	8	8	32	64
6	maxpool2d	2	8	4	64	64

Layer No.	Layer Type	Kernel Size	Input Dim	Output Dim	Input Channels	Output Channels
7	conv2d + batchnorm + relu	3	4	4	64	128
8	maxpool2d	2	4	2	128	128
9	flatten	-	2	1	128	512
9	linear	-	1	1	512	10

Architecture for ImprovedNet-V2 (which can be rendered by `net = ImprovedNet('2')`) is provided below:

Layer No.	Layer Type	Kernel Size	Input Dim	Output Dim	Input Channels	Output Channels
1	conv2d + batchnorm + relu	3	32	32	3	32
2	maxpool2d	2	32	16	32	32
3	conv2d + batchnorm + relu	3	16	16	32	64
4	maxpool2d	2	16	8	64	64
5	conv2d + batchnorm + relu	3	8	8	64	128
6	maxpool2d	2	8	4	128	128
7	conv2d + batchnorm + relu	3	4	4	128	256
8	maxpool2d	2	4	2	256	256
9	conv2d + batchnorm + relu	3	2	2	256	512
10	maxpool2d	2	2	1	512	512
11	flatten	-	2	1	512	512
12	linear	-	1	1	512	10

Architecture for ImprovedNet-V2 (which can be rendered by `net = ImprovedNet('3')`) is provided below:

Layer No.	Layer Type	Kernel Size	Input Dim	Output Dim	Input Channels	Output Channels
1	conv2d + batchnorm + relu	3	32	32	3	32

Layer No.	Layer Type	Kernel Size	Input Dim	Output Dim	Input Channels	Output Channels
2	maxpool2d	2	32	16	32	32
3	conv2d + batchnorm + relu	3	16	16	32	64
4	maxpool2d	2	16	8	64	64
5	conv2d + batchnorm + relu	3	8	8	64	128
6	conv2d + batchnorm + relu	3	8	8	128	128
7	maxpool2d	2	8	4	128	128
8	conv2d + batchnorm + relu	3	4	4	128	256
9	conv2d + batchnorm + relu	3	4	4	256	256
9	maxpool2d	2	4	2	256	256
10	conv2d + batchnorm + relu	3	2	2	256	512
11	conv2d + batchnorm + relu	3	2	2	512	512
12	maxpool2d	2	2	1	512	512
13	flatten	-	2	1	512	512
14	linear	-	1	1	512	10

Result

Below is a table for comparing the aforementioned three models with different depth serving image classification task. Data augmentation with random cropping and horizontal flipping (`transforms.Compose([transforms.RandomResizedCrop(32), transforms.RandomHorizontalFlip()])`) were utilized to maximize the training performance.

model	accuracy [%]
ImprovedNet-V1	81.2
ImprovedNet-V2	86.2
ImprovedNet-V3	85.9

It could be observed that ImprovedNet-V2, a model with intermediate depth, showed the best validation accuracy among three options. The reason why the deepest model (ImprovedNet-V3) does not

demonstrate the best performance can be explained in various ways: deeper model typically takes more data and time to learn the task, too deeper model can encounter gradient vanishing, or etc.

3. **Secret test set [5 pts].**

I uploaded the results obtained from aforementioned ImprovedNet-V2 model and the results are:

```
Q1 evaluation results:
Accuracy: 74.6 %
Accuracy of airplane : 65.0 %
Accuracy of automobile : 74.0 %
Accuracy of  bird : 66.5 %
Accuracy of   cat : 60.0 %
Accuracy of  deer : 80.0 %
Accuracy of   dog : 73.0 %
Accuracy of  frog : 84.5 %
Accuracy of horse : 81.0 %
Accuracy of  ship : 77.0 %
Accuracy of truck : 85.0 %
```

MP4-Q2 by Jongwon Lee (jongwon5)

1. **Implement training cycle:**

Implementation

Below shows how my training cycle is organized.

```
# essential hyperparameters for training
learning_rate = 1e-3    # learning rate
loss_name = 'L1'        # choose one in ['L1', 'cos']
decoder_name = 'none'   # choose one in ['none', 'basic', 'unet']
num_epochs = 200        # number of epochs to go through
val_period = 5          # period of validation over epochs
num_early_stop = 10     # number of patiences over validation to quit
training
resume_path = None      # path to load a model to be resumed

# directories and file names to save or load model
model_dir = "part2_model/{0}_{1}_{2:.0e}/".format(decoder_name, loss_name,
learning_rate)
best_model_fname = "model_best.pth"
best_model_path = os.path.join(model_dir, best_model_fname)

# create model instance and load weights if it is in the case
model = MyModel(decoder_type=decoder_name).to(device)

if resume_path is not None and os.path.exists(resume_path):
    model.load_state_dict(torch.load(resume_path))
    print("Resuming best model from %s" % (resume_path))

# create a directory to save checkpoints
if not os.path.exists(model_dir):
    os.makedirs(model_dir)
    print("Create %s" % (model_dir))

# create data loader, loss function, optimizer, and scheduler
train_dataset = NormalDataset(split='train')
train_dataloader = data.DataLoader(train_dataset, batch_size=8,
                                   shuffle=True, num_workers=2,
                                   drop_last=True)
criterion = MyCriterion(loss_type=loss_name).to(device)
optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9,
weight_decay=0.01)
scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5,
verbose=True)

'''
start training
'''
```

```
train_loss_over_epochs = []
val_error_over_epochs = []

for epoch in range(num_epochs):
    # training
    training_loss = simple_train(model, criterion, optimizer,
train_dataloader, epoch)
    train_loss_over_epochs.append(training_loss)

    # validation
    if epoch % val_period == (val_period-1):
        val_loss, val_error = simple_validation(model, epoch,
val_metric='mean_error', visualize=False)
        val_error_over_epochs.append(val_error)

        # step over scheduler
        scheduler.step(np.mean(val_loss))

        # check whether the performance has been improved. if not, quit
training
        if val_error == min(val_error_over_epochs):
            torch.save(model.state_dict(), best_model_path)
            print("[%d] Best model saved at %s\n" % (epoch + 1,
best_model_path))
            num_no_improvement = 0
        else:
            num_no_improvement += 1
            print("[%d] No improvement (%d / %d)\n" % (epoch + 1,
num_no_improvement, num_early_stop))
            if num_no_improvement >= num_early_stop:
                break
```

Below describes how the loss function is designed. Here, I prepared two types of loss function: *cos* and *L1* loss. For the remaing parts of this report, we assume that *L1* loss were used unless mentioned.

```
class MyCriterion(nn.Module):
    def __init__(self, loss_type):
        super(MyCriterion, self).__init__()
        assert loss_type in ['L1', 'cos']

        self.loss_type = loss_type

    def forward(self, prediction, target, mask):
        if self.loss_type is 'L1':
            prediction_norm = F.normalize(prediction, p=2, dim=1)
            target_norm = F.normalize(target, p=2, dim=1)
            loss =  (F.l1_loss(prediction_norm, target_norm) * mask).mean()

        if self.loss_type is 'cos':
            loss = ((1.0 - nn.CosineSimilarity(dim=1, eps=1e-6)(prediction,
target)) * mask).mean()
```

```
        return loss
```

2. **Build on top of ImageNet pre-trained Model [15 pts]:**

Implementation

The overall implementations of mine, including both the response for question 2 and 3, are managed in a unified class named `MyModel`. Each instance having different decoder followed by the ResNet18 backbone model can be generated by specifying the type of decoder, which is passed as a parameter `decoder_type`. Especially, for this case (ResNet18 + 32X Upsample), the corresponding instance can be generated by `MyModel(decoder_type='none')`. For more detail, please refer to the following portion of code:

```
class MyModel(nn.Module):
    def __init__(self, decoder_type='none'):
        super(MyModel, self).__init__()
        assert decoder_type in ['none', 'basic', 'unet']

        self.decoder_type = decoder_type

        self.encoder_base = models.resnet18(pretrained=True)
        self.encoder_base_layers = list(self.encoder_base.children())

        self.conv0 = nn.Sequential(*self.encoder_base_layers[:3])     #
output: [N,  64, H/2, W/2]
        self.conv1 = nn.Sequential(*self.encoder_base_layers[3:5])    #
output: [N,  64, H/4, W/4]
        self.conv2 = self.encoder_base_layers[5]                      #
output: [N, 128, H/8, W/8]
        self.conv3 = self.encoder_base_layers[6]                      #
output: [N, 256, H/16, W/16]
        self.conv4 = self.encoder_base_layers[7]                      #
output: [N, 512, H/32, W/32]

        if self.decoder_type == 'none':
            self.upsample = nn.Upsample(scale_factor=32, mode='bilinear',
align_corners=False)  # upsample [N,C,16,16] to [N,C,512,512]
            self.conv_last = nn.Conv2d(in_channels=512, out_channels=3,
kernel_size=1)          # deconv [N,512,H,W] to [N,3,H,W]

        elif self.decoder_type == 'basic':
            self.upsample = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=False)  # upsample [N,C,H/2,W/2] to [N,C,H,W]
            self.conv_last = nn.Conv2d(in_channels=64, out_channels=3,
kernel_size=1)          # deconv [N,64,H,W] to [N,3,H,W]

            self.deconv4 = self.conv_bn_relu(512, 512, kernel_size=1,
padding=0)  # input: [N, 512, H/32, W/32], output: [N, 512, H/32, W/32]
            self.deconv3 = self.conv_bn_relu(512, 256, kernel_size=3,
padding=1)  # input: [N, 512, H/16, W/16], output: [N, 256, H/16, W/16]
            self.deconv2 = self.conv_bn_relu(256, 128, kernel_size=3,
```

```
padding=1)  # input: [N, 256, H/8, W/8], output: [N, 128, H/8, W/8]
            self.deconv1 = self.conv_bn_relu(128,  64, kernel_size=3,
padding=1)  # input: [N, 128, H/4, W/4], output: [N,  64, H/4, W/4]

        elif self.decoder_type == 'unet':
            self.upsample = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=False)   # upsample [N,C,H/2,W/2] to [N,C,H,W]
            self.conv_last = nn.Conv2d(in_channels=64, out_channels=3,
kernel_size=1)          # deconv [N,64,H,W] to [N,3,H,W]

            self.conv4_1x1 = self.conv_bn_relu(512, 256, kernel_size=1,
padding=0)

            self.deconv3 = self.conv_bn_relu(256 + 256, 128, kernel_size=3,
padding=1) # merge upconv & upsampled conv4_out [N, 256, H/16, W/16] and
conv3_out [N, 256, H/16, W/16] to be deconv3_out [N, 128, H/16, W/16]
            self.deconv2 = self.conv_bn_relu(128 + 128, 64, kernel_size=3,
padding=1) # merge upconv & upsampled deconv3_out [N, 128, H/8, W/8] and
conv2_out [N, 128, H/8, W/8]   to be deconv2_out [N, 64, H/8, W/8]
            self.deconv1 = self.conv_bn_relu(64 + 64, 64, kernel_size=3,
padding=1)  # merge upconv & upsampled deconv2_out [N, 64, H/4, W/4] and
conv1_out [N, 64, H/4, W/4]    to be deconv1_out [N, 64, H/4, W/4]

    def conv_bn_relu(self, in_planes, out_planes, kernel_size, padding):
        "convolution + BN + relu"
        return nn.Sequential(
                nn.Conv2d(in_planes, out_planes,
                          kernel_size=kernel_size, padding=padding,
bias=False),
                nn.BatchNorm2d(out_planes),
                nn.ReLU(inplace=True),
                )

    def _forward_encoder(self, x):
        conv0_out = self.conv0(x)
        conv1_out = self.conv1(conv0_out)
        conv2_out = self.conv2(conv1_out)
        conv3_out = self.conv3(conv2_out)
        conv4_out = self.conv4(conv3_out)

        return conv0_out, conv1_out, conv2_out, conv3_out, conv4_out

    def _forward_basic(self, conv4_out):
        deconv4_out = self.upsample(self.deconv4(conv4_out))
        deconv3_out = self.upsample(self.deconv3(deconv4_out))
        deconv2_out = self.upsample(self.deconv2(deconv3_out))
        deconv1_out = self.upsample(self.deconv1(deconv2_out))

        return deconv1_out

    def _forward_unet(self, conv0_out, conv1_out, conv2_out, conv3_out,
conv4_out):
        # merge upsampled deconv4_out [N, 256, H/16, W/16] and conv3_out
[N, 256, H/16, W/16] to be deconv3_out [N, 256, H/16, W/16]
```

```
        deconv4_out = self.upsample(self.conv4_1x1(conv4_out))
        deconv3_out = self.deconv3(torch.cat([conv3_out, deconv4_out],
dim=1))

        # merge upsampled deconv3_out [N, 128, H/8, W/8] and conv2_out [N,
128, H/8, W/8]   to be deconv2_out [N, 128, H/8, W/8]
        deconv3_out = self.upsample(deconv3_out)
        deconv2_out = self.deconv2(torch.cat([conv2_out, deconv3_out],
dim=1))

        # merge upsampled deconv2_out [N, 64, H/4, W/4] and conv1_out [N,
64, H/4, W/4]    to be deconv1_out [N, 64, H/4, W/4]
        deconv2_out = self.upsample(deconv2_out)
        deconv1_out = self.deconv1(torch.cat([conv1_out, deconv2_out],
dim=1))

        # upsample deconv1_out [N, 64, H/4, W/4] to be deconv1_out [N, 64,
H/2, W/2]
        deconv1_out = self.upsample(deconv1_out)

        return deconv1_out

    def forward(self, x):
        conv0_out, conv1_out, conv2_out, conv3_out, conv4_out =
self._forward_encoder(x)

        if self.decoder_type == 'none':
            deconv_out = conv4_out
            deconv_out = self.conv_last(deconv_out)
            pred = self.upsample(deconv_out)

        elif self.decoder_type == 'basic':
            deconv_out = self._forward_basic(conv4_out)
            deconv_out = self.upsample(deconv_out)
            pred = self.conv_last(deconv_out)

        elif self.decoder_type == 'unet':
            deconv_out = self._forward_unet(conv0_out, conv1_out,
conv2_out, conv3_out, conv4_out)
            deconv_out = self.upsample(deconv_out)
            pred = self.conv_last(deconv_out)

        pred = torch.squeeze(pred, dim=1)
        pred = F.normalize(pred, p=2, dim=1)
        return pred
```

To see the parts directly pertaining to what this question is asking for, please have a look on `if self.decoder_type == 'none': ...`, `_forward_encoder()`, and `forward()`. To be specific, in `forward()`, you may see that the output of ResNet18 backbone model `deconv_out` goes through 1x1 covolution layer named `self.conv_last` and be shrinked down to have threed output channels, followed by a upsampling operation `self.upsample`. In the end, the output `pred` is normalized to have a unit size.

Result

Below is the model structure obtained by using a command `print(model)`:

```
model = MyModel(decoder_type='none').to(device)
print(model)
```

```
MyModel(
  (encoder_base): ResNet(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (relu): ReLU(inplace=True)
    (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)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer2): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer3): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
```

```
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=512, out_features=1000, bias=True)
  )
  (conv0): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (conv1): Sequential(
    (0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
    (1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
```

```
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
  )
  (conv2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (conv3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
```

```
  track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
  (1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
  (1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
      )
    )
    (conv4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=
  (1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
  (1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
  bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
  (1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
  (1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
      )
    )
    (upsample): Upsample(scale_factor=32.0, mode=bilinear)
    (conv_last): Conv2d(512, 3, kernel_size=(1, 1), stride=(1, 1))
  )
```

Details on the training hyperparameters are as follow:

```
learning rate: 1e-3
optimizer: SGD (momentum: 0.9, weight decay: 0.01)
scheduler: plateau reducing scheduler (patience: 5)
batch size: 1
number of epochs: 200 (early stopping on, patience: 10)
validation period: 5 epochs / validation
loss function: L1 loss
```

The final performance on validation set is reported as below:

model	mean error [deg]	median error [deg]	acc @ 11.25 [%]	acc @ 22.5 [%]	acc @ 30 [%]
none	32.7	26.3	23.2	44.3	55.1

3. **Increase your model output resolution [15 pts]:**

Implementation

As responded in the previous question, the overall implementations of mine (the response for both question 2 and 3) are managed in a unified class named `MyModel`. Each instance having different decoder followed by the ResNet18 backbone model can be generated by specifying the type of decoder, which is passed as a parameter `decoder_type`. Especially, the instance having **UNet** structure can be generated by `MyModel(decoder_type='unet')`. As an ablation study, the vary basic **encoder-decoder** structure, whose instance can be readily generated by `MyModel(decoder_type='basic')`, has been implemented as well. Unlike **UNet** structure, **encoder-decoder** structure does not have any skip connection across convolution and deconvolution layers.

For those who have a look on the detailed implementation, please refer to the reponse for Question 2.

Result

Below is the **UNet** model structure obtained by using a command `print(model)`:

```
model = MyModel(decoder_type='unet').to(device)
print(model)
```

```
MyModel(
  (encoder_base): ResNet(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (relu): ReLU(inplace=True)
    (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)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer2): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer3): Sequential(
```

```
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=512, out_features=1000, bias=True)
  )
  (conv0): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (conv1): Sequential(
    (0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
    (1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
  )
  (conv2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (conv3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (conv4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (upsample): Upsample(scale_factor=2.0, mode=bilinear)
  (conv_last): Conv2d(64, 3, kernel_size=(1, 1), stride=(1, 1))
  (conv4_1x1): Sequential(
    (0): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (deconv3): Sequential(
    (0): Conv2d(512, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (deconv2): Sequential(
    (0): Conv2d(256, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (deconv1): Sequential(
    (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
)
```

For reader's information, below is the basic **encoder-decoder** model structure obtained by using a command `print(model)`:

```
model = MyModel(decoder_type='basic').to(device)
print(model)
```

```
MyModel(
  (encoder_base): ResNet(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (relu): ReLU(inplace=True)
    (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)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer2): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
```

```
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer3): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
```

```
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=512, out_features=1000, bias=True)
  )
  (conv0): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (conv1): Sequential(
    (0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
    (1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
```

```
(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
  )
  (conv2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (conv3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
```

```
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (conv4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (upsample): Upsample(scale_factor=2.0, mode=bilinear)
  (conv_last): Conv2d(64, 3, kernel_size=(1, 1), stride=(1, 1))
  (deconv4): Sequential(
    (0): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
```

```
  (deconv3): Sequential(
    (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (deconv2): Sequential(
    (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (deconv1): Sequential(
    (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
)
```

Details on the training hyperparameters are as follow:

```
learning rate: 1e-3
optimizer: SGD (momentum: 0.9, weight decay: 0.01)
scheduler: plateau reducing scheduler (patience: 5)
batch size: 1
number of epochs: 200 (early stopping on, patience: 10)
validation period: 5 epochs / validation
loss function: L1 loss
```

The final performance on validation set is reported as below:

model	mean error [deg]	median error [deg]	acc @ 11.25 [%]	acc @ 22.5 [%]	acc @ 30 [%]
UNet	33.3	25.2	22.9	46.1	56.3

Ablation Study

Here, I provide an ablation table across three different DNN architecture.

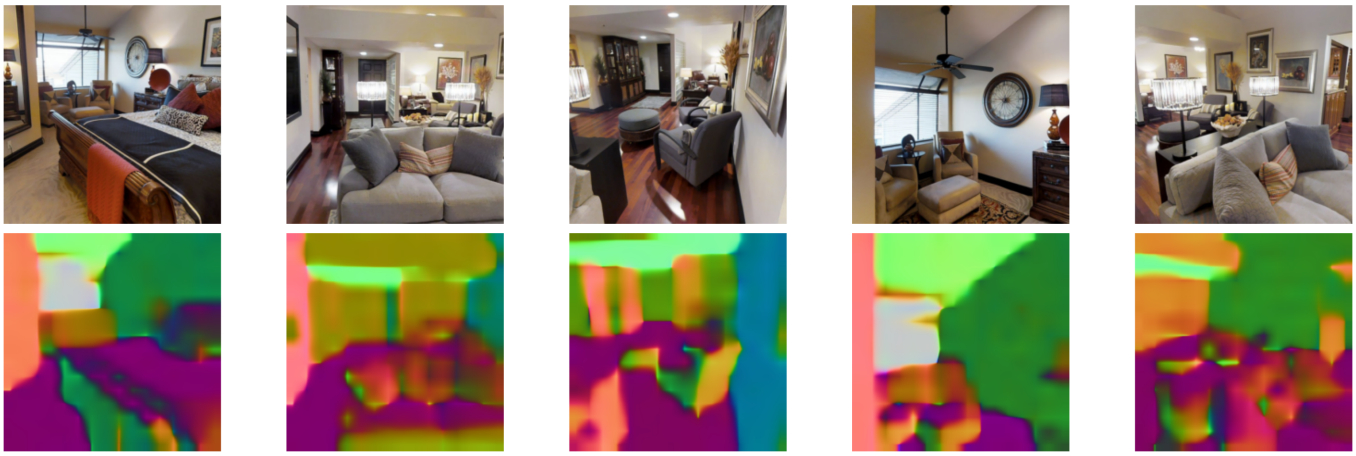
model	mean error [deg]	median error [deg]	acc @ 11.25 [%]	acc @ 22.5 [%]	acc @ 30 [%]
Upsampling	32.7	26.3	23.2	44.3	55.1

model	mean error [deg]	median error [deg]	acc @ 11.25 [%]	acc @ 22.5 [%]	acc @ 30 [%]
Encoder-Decoder	33.6	24.8	26.0	47.1	56.2
UNet	33.3	25.2	22.9	46.1	56.3

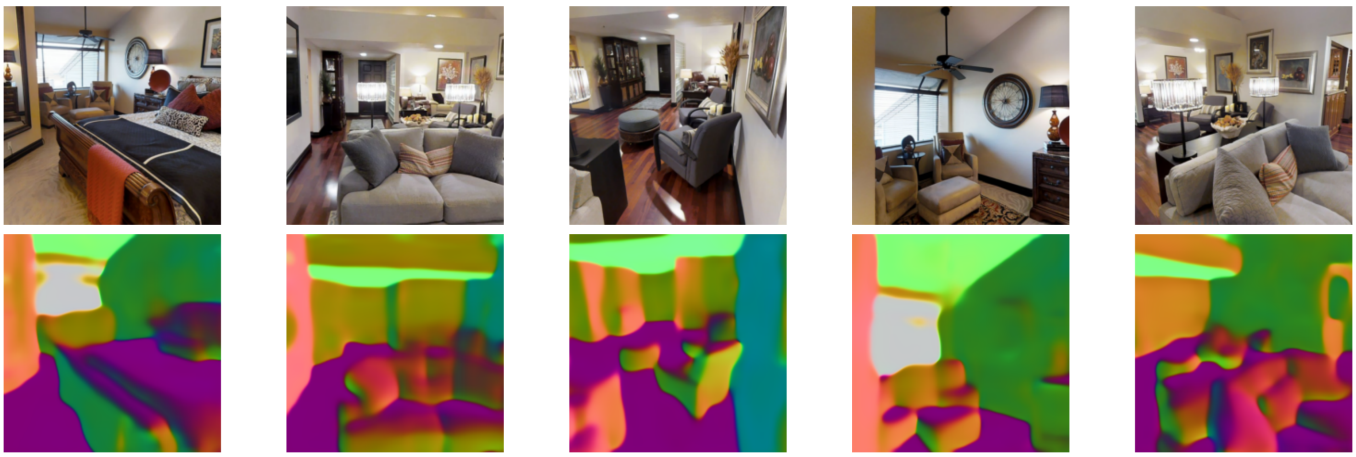
- Though the improved architecture (**Encoder-Decoder** and **UNet**) does not outperform the simple architecture (**Upsampling**), its visualization results shows drastic improvement compared to the latter. For details, please refer to the following section.
- Regarding the comparison between **Encoder-Decoder** and **UNet**, both of them show almost similar performance.
- The results obtained by using cosine similarity loss are not included as they showed no significant difference to utilizing L1 loss.

4. **Visualize your prediction [5 pts]:**

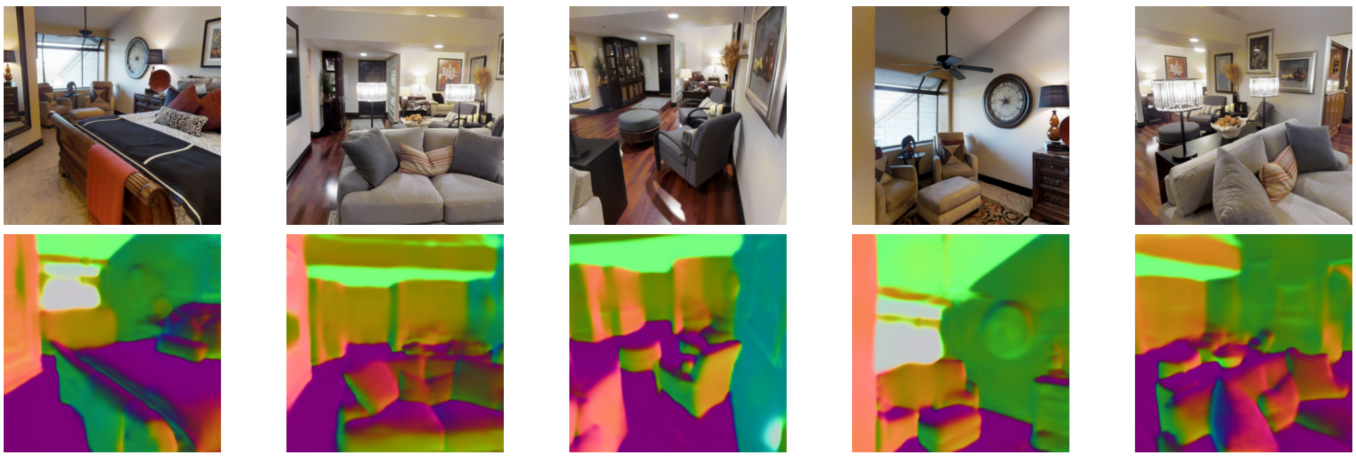
Upsampling



Encoder-decoder



UNet



Discussion

As can be seen above, the results from **Upsampling** shows plausible but imperior visualization output, with a noisy and blurred pattern. This artifact arises from upsampling small-sized outputs (16 x 16) to be a larger size (512 x 512). In contrast, however, the improved architecture (**Encoder-decoder** and **UNet**) demonstrates much concise and finer outputs. From this, we may deduce that the upconvolutional network and skip connection structure are able to obtain estimation output without unwanted information loss.

5. **Secret test set [5 pts].**

I uploaded the results obtained from aforementioned **UNet** model and the results are:

```
Q2 evaluation results:
Test metrics:
mean error 32.7
median error 23.8
accuracy at 11.25deg 23.9
accuracy at 22.5deg 48.0
accuracy at 30deg 58.2
```