GAN for Semantic Segmentation

GAN for Semantic Segmentation示例

- Generate Image from Segmentation Map Using Deep Learning
- Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data

Generate Image from Segmentation Map Using Deep Learning

This example shows how to generate a synthetic image of a scene from a semantic segmentation map using a pix2pixHD conditional generative adversarial network (CGAN).

Pix2pixHD consists of two networks that are trained simultaneously to maximize the performance of both.

- The generator is an encoder-decoder style neural network that generates a scene image from a semantic segmentation map. A CGAN network trains the generator to generate a scene image that the discriminator misclassifies as real.
- The discriminator is a fully convolutional neural network that compares a generated scene image and the corresponding real image and attempts to classify them as fake and real, respectively. A CGAN network trains the discriminator to correctly distinguish between generated and real image.

Download CamVid Data Set

This example uses the CamVid data set from the University of Cambridge for training. This data set is a collection of 701 images containing street-level views obtained while driving. The data set provides pixel labels for 32 semantic classes including car, pedestrian, and road.

```
imageURL =
'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_Still
sRaw_full.zip';
labelURL =
'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApp
roved_full.zip';

dataDir = fullfile(pwd);
downloadCamVidData(dataDir,imageURL,labelURL);
imgDir = fullfile(dataDir,"images","701_StillsRaw_full");
labelDir = fullfile(dataDir,'labels');
```

Preprocess Training Data

Create an ImageDatastore to store the images in the CamVid data set.

```
imds = imageDatastore(imgDir);
imageSize = [576 768];
```

Define the class names and pixel label IDs of the 32 classes in the CamVid data set using the helper function defineCamVid32ClassesAndPixelLabelIDs. Get a standard colormap for the CamVid data set using the helper function camvid32ColorMap.

```
numClasses = 32;
[classes,labelIDs] = defineCamVid32ClassesAndPixelLabelIDs;
cmap = camvid32ColorMap;
```

Preprocess Training Data

Create a PixelLabelDatastore to store the pixel label images.

```
pxds = pixelLabelDatastore(labelDir,classes,labelIDs);
```

Preview a pixel label image and the corresponding ground truth scene image. Convert the labels from categorical labels to RGB colors by using the label2rgb function, then display the pixel label image and ground truth image in a montage.

```
im = preview(imds);
px = preview(pxds);
px = label2rgb(px,cmap);
montage({px,im})
```

Preprocess Training Data

Partition the data into training and test sets using the helper function partitionCamVidForPix2PixHD. This function is attached to the example as a supporting file. The helper function splits the data into 648 training files and 53 test files.

```
[imdsTrain,imdsTest,pxdsTrain,pxdsTest] =
partitionCamVidForPix2PixHD(imds,pxds,classes,labelIDs);
```

Use the combine function to combine the pixel label images and ground truth scene images into a single datastore.

```
dsTrain = combine(pxdsTrain,imdsTrain);
```

Preprocess Training Data

Augment the training data by using the transform function with custom preprocessing operations specified by the helper function preprocessCamVidForPix2PixHD.

The preprocessCamVidForPix2PixHD function performs these operations:

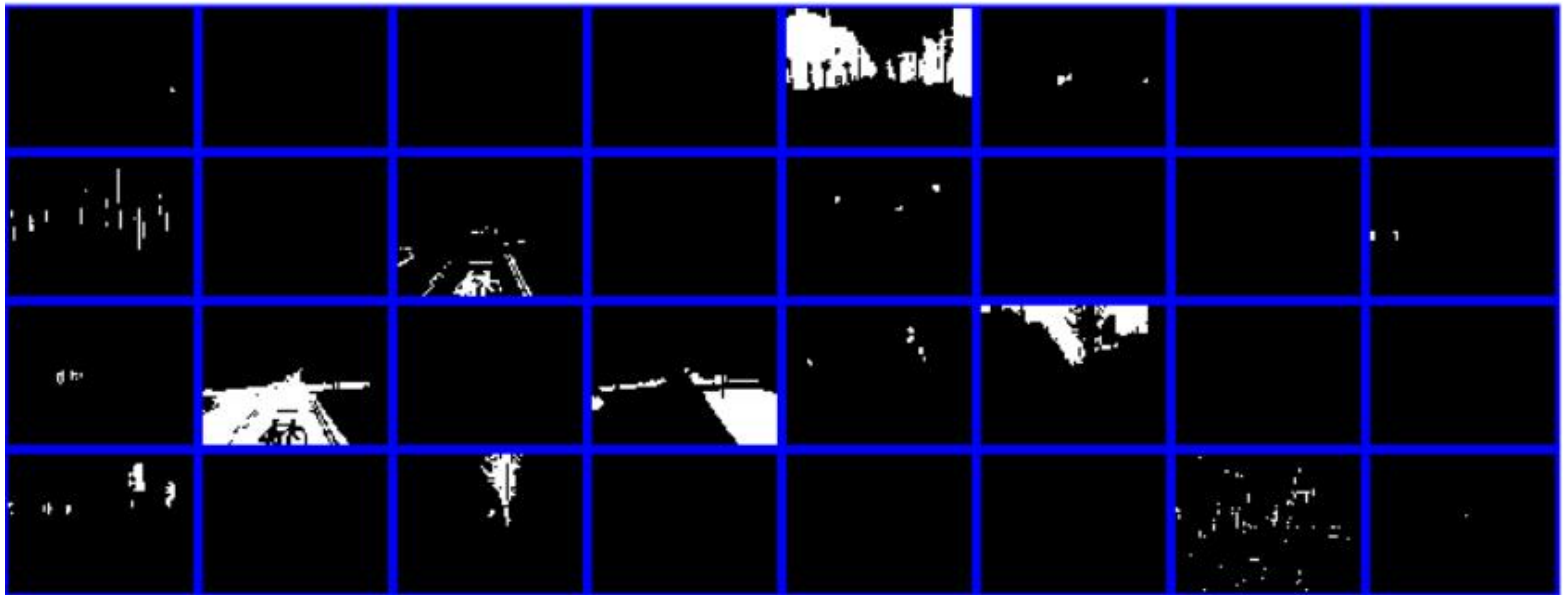
1. Scale the ground truth data to the range [-1, 1]. This range matches the range of the final tanhLayer in the generator network.
2. Resize the image and labels to the output size of the network, 576-by-768 pixels, using bicubic and nearest neighbor downsampling, respectively.
3. Convert the single channel segmentation map to a 32-channel one-hot encoded segmentation map using the onehotencode function.
4. Randomly flip image and pixel label pairs in the horizontal direction.

```
dsTrain = transform(dsTrain,@(x) preprocessCamVidForPix2PixHD(x,imageSize));
```

Preprocess Training Data

Preview the channels of a one-hot encoded segmentation map in a montage. Each channel represents a one-hot map corresponding to pixels of a unique class.

```
map = preview(dsTrain);
montage(map{1},'Size',[4 8],'Bordersize',5,'BackgroundColor','b')
```



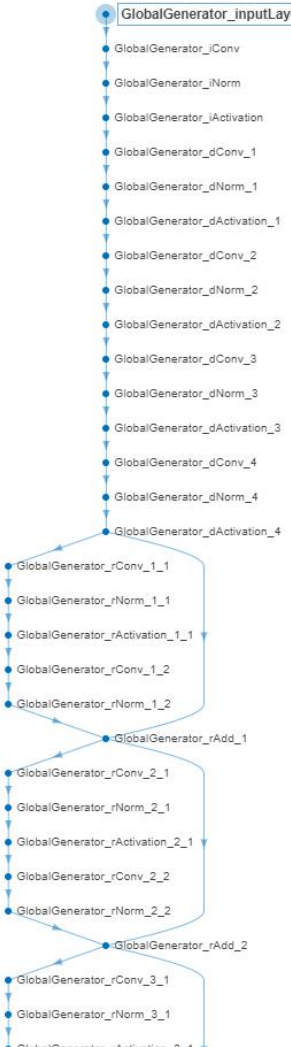
Create Generator Network

Define a pix2pixHD generator network that generates a scene image from a depth-wise one-hot encoded segmentation map. This input has same height and width as the original segmentation map and the same number of channels as classes.

```
generatorInputSize = [imageSize numClasses];
```

Create the pix2pixHD generator network using the pix2pixHDGlobalGenerator function.

```
dlnetGenerator = pix2pixHDGlobalGenerator(generatorInputSize);
Display the network architecture.
analyzeNetwork(dlnetGenerator)
```



ANALYSIS RESULT

	Name	Type	Activations	Learnable Prope...
1	GlobalGenerator_inputLayer 576×768×32 images	Image Input	576(S) × 768(S) × 32(C) × 1(B)	-
2	GlobalGenerator_iConv 64 7×7×32 convolutions with stride [1 1] ...	Convolution	576(S) × 768(S) × 64(C) × 1(B)	Weig… 7 × 7 × 32… Bias 1 × 1 × 64
3	GlobalGenerator_iNorm Instance normalization with 64 channels	Instance Normaliza...	576(S) × 768(S) × 64(C) × 1(B)	Offset 1 × 1 × 64 Scale 1 × 1 × 64
4	GlobalGenerator_iActivation ReLU	ReLU	576(S) × 768(S) × 64(C) × 1(B)	-
5	GlobalGenerator_dConv_1 128 3×3×64 convolutions with stride [2 2...	Convolution	288(S) × 384(S) × 128(C) × 1(B)	Weig… 3 × 3 × 64 … Bias 1 × 1 × 128
6	GlobalGenerator_dNorm_1 Instance normalization with 128 channels	Instance Normaliza...	288(S) × 384(S) × 128(C) × 1(B)	Offset 1 × 1 × 128 Scale 1 × 1 × 128
7	GlobalGenerator_dActivation_1 ReLU	ReLU	288(S) × 384(S) × 128(C) × 1(B)	-
8	GlobalGenerator_dConv_2 256 3×3×128 convolutions with stride [2 ...	Convolution	144(S) × 192(S) × 256(C) × 1(B)	Wei… 3 × 3 × 128… Bias 1 × 1 × 256
9	GlobalGenerator_dNorm_2 Instance normalization with 256 channels	Instance Normaliza...	144(S) × 192(S) × 256(C) × 1(B)	Offset 1 × 1 × 256 Scale 1 × 1 × 256
10	GlobalGenerator_dActivation_2 ReLU	ReLU	144(S) × 192(S) × 256(C) × 1(B)	-
11	GlobalGenerator_dConv_3 512 3×3×256 convolutions with stride [2 ...	Convolution	72(S) × 96(S) × 512(C) × 1(B)	Wei… 3 × 3 × 256… Bias 1 × 1 × 512
12	GlobalGenerator_dNorm_3 Instance normalization with 512 channels	Instance Normaliza...	72(S) × 96(S) × 512(C) × 1(B)	Offset 1 × 1 × 512 Scale 1 × 1 × 512
13	GlobalGenerator_dActivation_3 ReLU	ReLU	72(S) × 96(S) × 512(C) × 1(B)	-
14	GlobalGenerator_dConv_4 1024 3×3×512 convolutions with stride [...	Convolution	36(S) × 48(S) × 1024(C) × 1(B)	Wei… 3 × 3 × 512… Bias 1 × 1 × 1024
15	GlobalGenerator_dNorm_4 Instance normalization with 1024 channels	Instance Normaliza...	36(S) × 48(S) × 1024(C) × 1(B)	Offs… 1 × 1 × 10… Scale 1 × 1 × 10…
16	GlobalGenerator_dActivation_4 ReLU	ReLU	36(S) × 48(S) × 1024(C) × 1(B)	-
17	GlobalGenerator_rConv_1_1 1024 3×3×1024 convolutions with stride ...	Convolution	36(S) × 48(S) × 1024(C) × 1(B)	Wei… 3 × 3 × 102… Bias 1 × 1 × 1024
18	GlobalGenerator_rNorm_1_1 Instance normalization with 1024 channels	Instance Normaliza...	36(S) × 48(S) × 1024(C) × 1(B)	Offs… 1 × 1 × 10… Scale 1 × 1 × 10…
19	GlobalGenerator_rActivation_1_1 ReLU	ReLU	36(S) × 48(S) × 1024(C) × 1(B)	-
20	GlobalGenerator_rConv_1_2 1024 3×3×1024 convolutions with stride ...	Convolution	36(S) × 48(S) × 1024(C) × 1(B)	Wei… 3 × 3 × 102… Bias 1 × 1 × 1024
21	GlobalGenerator_rNorm_1_2 Instance normalization with 1024 channels	Instance Normaliza...	36(S) × 48(S) × 1024(C) × 1(B)	Offs… 1 × 1 × 10… Scale 1 × 1 × 10…
22	GlobalGenerator_rAdd_1 Element-wise addition of 2 inputs	Addition	36(S) × 48(S) × 1024(C) × 1(B)	-
23	GlobalGenerator_rConv_2_1 1024 3×3×1024 convolutions with stride ...	Convolution	36(S) × 48(S) × 1024(C) × 1(B)	Wei… 3 × 3 × 102… Bias 1 × 1 × 1024
24	GlobalGenerator_rNorm_2_1 Instance normalization with 1024 channels	Instance Normaliza...	36(S) × 48(S) × 1024(C) × 1(B)	Offs… 1 × 1 × 10… Scale 1 × 1 × 10…

Create Discriminator Network

Define the patch GAN discriminator networks that classifies an input image as either real (1) or fake (0). This example uses two discriminator networks at different input scales, also known as multiscale discriminators. The first scale is the same size as the image size, and the second scale is half the size of image size.

The input to the discriminator is the depth-wise concatenation of the one-hot encoded segmentation maps and the scene image to be classified. Specify the number of channels input to the discriminator as the total number of labeled classes and image color channels.

```
numImageChannels = 3;
numChannelsDiscriminator = numClasses + numImageChannels;
```

Create Discriminator Network

Specify the input size of the first discriminator. Create the patch GAN discriminator with instance normalization using the patchGANDiscriminator function.

```
discriminatorInputSizeScale1 = [imageSize numChannelsDiscriminator];
dlnetDiscriminatorScale1 =
patchGANDiscriminator(discriminatorInputSizeScale1,"NormalizationLayer","instance");
```

Specify the input size of the second discriminator as half the image size, then create the second patch GAN discriminator.

```
discriminatorInputSizeScale2 = [floor(imageSize)./2 numChannelsDiscriminator];
dlnetDiscriminatorScale2 =
patchGANDiscriminator(discriminatorInputSizeScale2,"NormalizationLayer","instance");
```

Create Discriminator Network

Visualize the networks.

```
analyzeNetwork(dlnetDiscriminatorScale1);
analyzeNetwork(dlnetDiscriminatorScale2);
```

Create Discriminator Network

ANALYSIS RESULT

	Name	Type	Activations	Learnable Prope...
1	input_top 576×768×35 images	Image Input	576(S) × 768(S) × 35(C) × 1(B)	-
2	conv2d_top 64 4×4×35 convolutions with stride [2 2] ...	Convolution	288(S) × 384(S) × 64(C) × 1(B)	Weig… 4 × 4 × 35… Bias 1 × 1 × 64
3	act_top Leaky ReLU with scale 0.2	Leaky ReLU	288(S) × 384(S) × 64(C) × 1(B)	-
4	conv2d_mid_1 128 4×4×64 convolutions with stride [2 2...	Convolution	144(S) × 192(S) × 128(C) × 1(B)	Weig… 4 × 4 × 64 … Bias 1 × 1 × 128
5	norm2d_mid_1 Group normalization with 128 channels ...	Group Normalization	144(S) × 192(S) × 128(C) × 1(B)	Offset 1 × 1 × 128 Scale 1 × 1 × 128
6	act_mid_1 Leaky ReLU with scale 0.2	Leaky ReLU	144(S) × 192(S) × 128(C) × 1(B)	-
7	conv2d_mid_2 256 4×4×128 convolutions with stride [2 ...	Convolution	72(S) × 96(S) × 256(C) × 1(B)	Wei… 4 × 4 × 128… Bias 1 × 1 × 256
8	norm2d_mid_2 Group normalization with 256 channels ...	Group Normalization	72(S) × 96(S) × 256(C) × 1(B)	Offset 1 × 1 × 256 Scale 1 × 1 × 256
9	act_mid_2 Leaky ReLU with scale 0.2	Leaky ReLU	72(S) × 96(S) × 256(C) × 1(B)	-
10	conv2d_tail 512 4×4×256 convolutions with stride [1 ...	Convolution	71(S) × 95(S) × 512(C) × 1(B)	Wei… 4 × 4 × 256… Bias 1 × 1 × 512
11	norm2d_tail Group normalization with 512 channels ...	Group Normalization	71(S) × 95(S) × 512(C) × 1(B)	Offset 1 × 1 × 512 Scale 1 × 1 × 512
12	act_tail Leaky ReLU with scale 0.2	Leaky ReLU	71(S) × 95(S) × 512(C) × 1(B)	-
13	conv2d_final 1 4×4×512 convolutions with stride [1 1] ...	Convolution	70(S) × 94(S) × 1(C) × 1(B)	Weigh… 4 × 4 × 5… Bias 1 × 1

Create Discriminator Network

ANALYSIS RESULT

	Name	Type	Activations	Learnable Prope...
1	input_top 288×384×35 images	Image Input	288(S) × 384(S) × 35(C) × 1(B)	-
2	conv2d_top 64 4×4×35 convolutions with stride [2 2] ...	Convolution	144(S) × 192(S) × 64(C) × 1(B)	Weig… 4 × 4 × 35… Bias 1 × 1 × 64
3	act_top Leaky ReLU with scale 0.2	Leaky ReLU	144(S) × 192(S) × 64(C) × 1(B)	-
4	conv2d_mid_1 128 4×4×64 convolutions with stride [2 2...	Convolution	72(S) × 96(S) × 128(C) × 1(B)	Weig… 4 × 4 × 64 … Bias 1 × 1 × 128
5	norm2d_mid_1 Group normalization with 128 channels ...	Group Normalization	72(S) × 96(S) × 128(C) × 1(B)	Offset 1 × 1 × 128 Scale 1 × 1 × 128
6	act_mid_1 Leaky ReLU with scale 0.2	Leaky ReLU	72(S) × 96(S) × 128(C) × 1(B)	-
7	conv2d_mid_2 256 4×4×128 convolutions with stride [2 ...	Convolution	36(S) × 48(S) × 256(C) × 1(B)	Wei… 4 × 4 × 128… Bias 1 × 1 × 256
8	norm2d_mid_2 Group normalization with 256 channels ...	Group Normalization	36(S) × 48(S) × 256(C) × 1(B)	Offset 1 × 1 × 256 Scale 1 × 1 × 256
9	act_mid_2 Leaky ReLU with scale 0.2	Leaky ReLU	36(S) × 48(S) × 256(C) × 1(B)	-
10	conv2d_tail 512 4×4×256 convolutions with stride [1 ...	Convolution	35(S) × 47(S) × 512(C) × 1(B)	Wei… 4 × 4 × 256… Bias 1 × 1 × 512
11	norm2d_tail Group normalization with 512 channels ...	Group Normalization	35(S) × 47(S) × 512(C) × 1(B)	Offset 1 × 1 × 512 Scale 1 × 1 × 512
12	act_tail Leaky ReLU with scale 0.2	Leaky ReLU	35(S) × 47(S) × 512(C) × 1(B)	-
13	conv2d_final 1 4×4×512 convolutions with stride [1 1] ...	Convolution	34(S) × 46(S) × 1(C) × 1(B)	Weigh… 4 × 4 × 5… Bias 1 × 1

Define Model Gradients and Loss Functions

The helper function modelGradients calculates the gradients and adversarial loss for the generator and discriminator. The function also calculates the feature matching loss and VGG loss for the generator.

Generator Loss

The objective of the generator is to generate images that the discriminator classifies as real (1). The generator loss consists of three losses.

- The adversarial loss is computed as the squared difference between a vector of ones and the discriminator predictions on the generated image. $\hat{Y}_{generated}$ are discriminator predictions on the image generated by the generator. This loss is implemented using part of the `pix2pixhdAdversarialLoss` helper function defined in the Supporting Functions section of this example.

$$lossAdversarialGenerator = (1 - \hat{Y}_{generated})^2$$

Generator Loss

The objective of the generator is to generate images that the discriminator classifies as real (1). The generator loss consists of three losses.

- The feature matching loss penalises the L^1 distance between the real and generated feature maps obtained as predictions from the discriminator network. T is total number of discriminator feature layers. Y_{real} and $\hat{Y}_{generated}$ are the ground truth images and generated images, respectively. This loss is implemented using the `pix2pixhdFeatureMatchingLoss` helper function defined in the Supporting Functions section of this example

$$lossFeatureMatching = \sum_{i=1}^{T} ||Y_{real} - \hat{Y}_{generated}||_1$$

Generator Loss

The objective of the generator is to generate images that the discriminator classifies as real (1). The generator loss consists of three losses.

- The perceptual loss penalises the L^1 distance between real and generated feature maps obtained as predictions from a feature extraction network. T is total number of feature layers. $Y_{VggReal}$ and $\hat{Y}_{VggGenerated}$ are network predictions for ground truth images and generated images, respectively. This loss is implemented using the `pix2pixhdVggLoss` helper function defined in the Supporting Functions section of this example. The feature extraction network is created in Load Feature Extraction Network.

$$lossVgg = \sum_{i=1}^{T} ||Y_{VggReal} - \hat{Y}_{VggGenerated}||_1$$

Generator Loss

The overall generator loss is a weighted sum of all three losses. λ_1, λ_2, and λ_3 are the weight factors for adversarial loss, feature matching loss, and perceptual loss, respectively.

$$lossGenerator = \lambda_1 * lossAdversarialGenerator + \lambda_2 * lossFeatureMatching + \lambda_3 * lossPerceptual$$

Note that the adversarial loss and feature matching loss for the generator are computed for two different scales.

Discriminator Loss

The objective of the discriminator is to correctly distinguish between ground truth images and generated images. The discriminator loss is a sum of two components:

- The squared difference between a vector of ones and the predictions of the discriminator on real images
- The squared difference between a vector of zeros and the predictions of the discriminator on generated images

$$lossDiscriminator = (1 - Y_{real})^2 + (0 - \hat{Y}_{generated})^2$$

The discriminator loss is implemented using part of the pix2pixhdAdversarialLoss helper function defined in the Supporting Functions section of this example. Note that adversarial loss for the discriminator is computed for two different discriminator scales.

Load Feature Extraction Network

This example modifies a pretrained VGG-19 deep neural network to extract the features of the real and generated images at various layers. These multilayer features are used to compute the perceptual loss of the generator.

```
netVGG = vgg19;
```

Load Feature Extraction Network

To make the VGG-19 network suitable for feature extraction, keep the layers up to 'pool5' and remove all of the fully connected layers from the network. The resulting network is a fully convolutional network.

```
netVGG = layerGraph(netVGG.Layers(1:38));
```

Create a new image input layer with no normalization. Replace the original image input layer with the new layer.

```
inp = imageInputLayer([imageSize 3],"Normalization","None","Name","Input");
netVGG = replaceLayer(netVGG,"input",inp);
netVGG = dlnetwork(netVGG);
```

Specify Training Options

Specify the options for Adam optimization. Train for 60 epochs. Specify identical options for the generator and discriminator networks.

- Specify an equal learning rate of 0.0002.
- Initialize the trailing average gradient and trailing average gradient-square decay rates with [].
- Use a gradient decay factor of 0.5 and a squared gradient decay factor of 0.999.
- Use a mini-batch size of 1 for training.

Specify Training Options

```
numEpochs = 60;
learningRate = 0.0002;
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminatorScale1 = [];
trailingAvgSqDiscriminatorScale1 = [];
trailingAvgDiscriminatorScale2 = [];
trailingAvgSqDiscriminatorScale2 = [];
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
miniBatchSize = 1;
```

Specify Training Options

Create a minibatchqueue object that manages the mini-batching of observations in a custom training loop. The minibatchqueue object also casts data to a dlarray object that enables auto differentiation in deep learning applications.

Specify the mini-batch data extraction format as SSCB (spatial, spatial, channel, batch). Set the DispatchInBackground name-value pair argument as the boolean returned by canUseGPU. If a supported GPU is available for computation, then the minibatchqueue object preprocesses mini-batches in the background in a parallel pool during training.

```
mbqTrain = minibatchqueue(dsTrain,"MiniBatchSize",miniBatchSize, ...
    "MiniBatchFormat","SSCB","DispatchInBackground",canUseGPU);
```

Train the Network

By default, the example downloads a pretrained version of the pix2pixHD generator network for the CamVid data set by using the helper function downloadTrainedPix2PixHDNet. The helper function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the doTraining variable in the following code to true. Train the model in a custom training loop. For each iteration:

- Read the data for current mini-batch using the next function.
- Evaluate the model gradients using the dlfeval function and the modelGradients helper function.
- Update the network parameters using the adamupdate function.
- Update the training progress plot for every iteration and display various computed losses.

Train the Network

Training takes about 22 hours on an NVIDIA™ Titan RTX and can take even longer depending on your GPU hardware. If your GPU device has less memory, try reducing the size of the input images by specifying the imageSize variable as [480 640] in the Preprocess Training Data section of the example.

```
doTraining = false;
if doTraining
    fig = figure;
    lossPlotter = configureTrainingProgressPlotter(fig);
    iteration = 0;
    for epoch = 1:numEpochs   % Loop over epochs
        reset(mbqTrain);      % Reset and shuffle the data
        shuffle(mbqTrain);
        while hasdata(mbqTrain)        % Loop over each image
            iteration = iteration + 1;
            % Read data from current mini-batch
            [dlInputSegMap,dlRealImage] = next(mbqTrain);
            % Evaluate the model gradients and the generator state
            [gradParamsG,gradParamsDScale1,gradParamsDScale2,...
               lossGGAN,lossGFM,lossGVGG,lossD] = dlfeval(   ...
               @modelGradients,dlInputSegMap,dlRealImage,dlnetGenerator,...
               dlnetDiscriminatorScale1,dlnetDiscriminatorScale2,netVGG);
```

```
% Update the generator parameters
[dlnetGenerator,trailingAvgGenerator,trailingAvgSqGenerator] = adamupdate( ...
  dlnetGenerator,gradParamsG, trailingAvgGenerator,trailingAvgSqGenerator,iteration, ...
  learningRate,gradientDecayFactor,squaredGradientDecayFactor);

% Update the discriminator scale1 parameters
[dlnetDiscriminatorScale1,trailingAvgDiscriminatorScale1,trailingAvgSqDiscriminatorScal
e1] = adamupdate( dlnetDiscriminatorScale1,gradParamsDScale1, ...
trailingAvgDiscriminatorScale1,trailingAvgSqDiscriminatorScale1,iteration, ...
                learningRate,gradientDecayFactor,squaredGradientDecayFactor);

% Update the discriminator scale2 parameters
[dlnetDiscriminatorScale2,trailingAvgDiscriminatorScale2,trailingAvgSqDiscriminatorScal
e2] = adamupdate( dlnetDiscriminatorScale2,gradParamsDScale2, ...
trailingAvgDiscriminatorScale2,trailingAvgSqDiscriminatorScale2,iteration, ...
                learningRate,gradientDecayFactor,squaredGradientDecayFactor);

% Plot and display various losses
lossPlotter = updateTrainingProgressPlotter(lossPlotter,iteration, ...
                epoch,numEpochs,lossD,lossGGAN,lossGFM,lossGVGG);
        end
    end
```

Evaluate Generated Images from Test Data

The performance of this trained Pix2PixHD network is limited because the number of CamVid training images is relatively small. Additionally, some images belong to an image sequence and therefore are correlated with other images in the training set. To improve the effectiveness of the Pix2PixHD network, train the network using a different data set that has a larger number of training images without correlation.

Evaluate Generated Images from Test Data

Because of the limitations, this Pix2PixHD network generates more realistic images for some test images than for others. To demonstrate the difference in results, compare the generated images for the first and third test image. The camera angle of the first test image has an uncommon vantage point that faces more perpendicular to the road than the typical training image. In contrast, the camera angle of the third test image has a typical vantage point that faces along the road and shows two lanes with lane markers. The network has significantly better performance generating a realistic image for the third test image than for the first test image.

Evaluate Generated Images from Test Data

Get the first ground truth scene image from the test data. Resize the image using bicubic interpolation.

```
idxToTest = 1;
gtImage = readimage(imdsTest,idxToTest);
gtImage = imresize(gtImage,imageSize,"bicubic");
```

Get the corresponding pixel label image from the test data. Resize the pixel label image using nearest neighbor interpolation.

```
segMap = readimage(pxdsTest,idxToTest);
segMap = imresize(segMap,imageSize,"nearest");
```

Evaluate Generated Images from Test Data

Convert the pixel label image to a multichannel one-hot segmentation map by using the onehotencode function.

```
segMapOneHot = onehotencode(segMap,3,'single');
```

Create dlarray objects that inputs data to the generator. If a supported GPU is available for computation, then perform inference on a GPU by converting the data to a gpuArray object.

```
dlSegMap = dlarray(segMapOneHot,'SSCB');
if canUseGPU
    dlSegMap = gpuArray(dlSegMap);
end
```

Evaluate Generated Images from Test Data

Generate a scene image from the generator and one-hot segmentation map using the predict function.

```
dlGeneratedImage = predict(dlnetGenerator,dlSegMap);
generatedImage = extractdata(gather(dlGeneratedImage));
```

The final layer of the generator network produces activations in the range [-1, 1]. For display, rescale the activations to the range [0, 1].

```
generatedImage = rescale(generatedImage);
```

For display, convert the labels from categorical labels to RGB colors by using the label2rgb function.

```
coloredSegMap = label2rgb(segMap,cmap);
```

Evaluate Generated Images from Test Data

Display the RGB pixel label image, generated scene image, and ground truth scene image in a montage.

```
figure
montage({coloredSegMap generatedImage gtImage},'Size',[1 3])
title(['Test Pixel Label Image ',num2str(idxToTest),' with Generated and
Ground Truth Scene Images'])
```

Evaluate Generated Images from Test Data

Get the third ground truth scene image from the test data. Resize the image using bicubic interpolation.

```
idxToTest = 3;
gtImage = readimage(imdsTest,idxToTest);
gtImage = imresize(gtImage,imageSize,"bicubic");
```

To get the third pixel label image from the test data and to generate the corresponding scene image, you can use the helper function evaluatePix2PixHD.

Evaluate Generated Images from Test Data

The evaluatePix2PixHD function performs the same operations as the evaluation of the first test image:

- Get a pixel label image from the test data. Resize the pixel label image using nearest neighbor interpolation.
- Convert the pixel label image to a multichannel one-hot segmentation map using the onehotencode function.
- Create a dlarray object to input data to the generator. For GPU inference, convert the data to a gpuArray object.
- Generate a scene image from the generator and one-hot segmentation map using the predict function.
- Rescale the activations to the range [0, 1].

Evaluate Generated Images from Test Data

```
[generatedImage,segMap] =
evaluatePix2PixHD(pxdsTest,idxToTest,imageSize,dlnetGenerator);
```

For display, convert the labels from categorical labels to RGB colors by using the label2rgb function.

```
coloredSegMap = label2rgb(segMap,cmap);
```

Evaluate Generated Images from Test Data

Display the RGB pixel label image, generated scene image, and ground truth scene image in a montage.

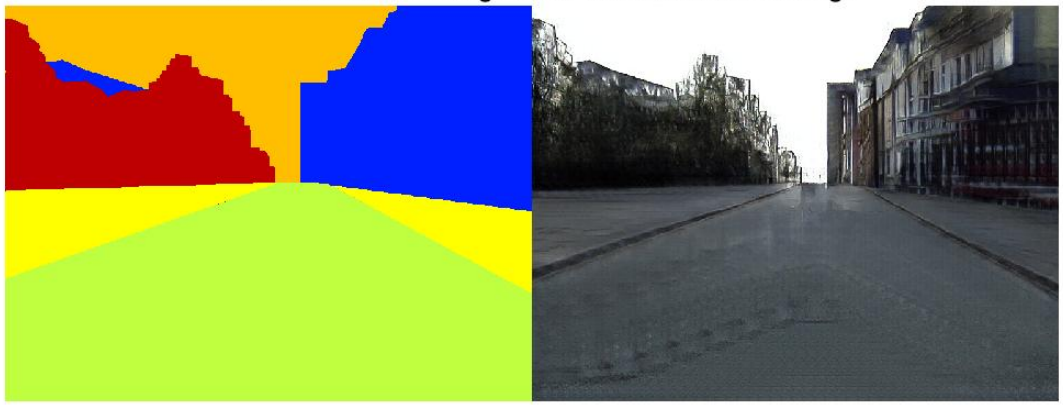
```
figure
montage({coloredSegMap generatedImage gtImage},'Size',[1 3])
title(['Test Pixel Label Image ',num2str(idxToTest),' with Generated and
Ground Truth Scene Images'])
```

Evaluate Generated Images from Custom Pixel Label Images

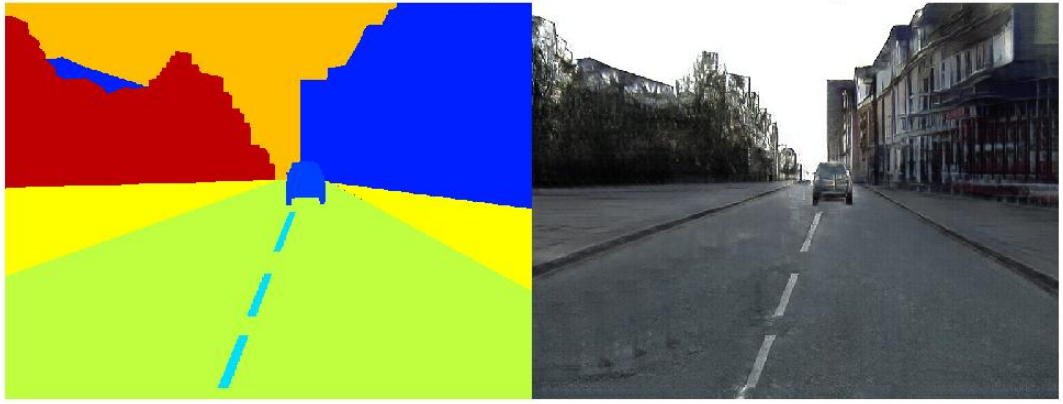
For each pixel label image in the datastore, generate a scene image using the helper function evaluatePix2PixHD.

```
for idx = 1:length(cpxds.Files)
    % Get the pixel label image and generated scene image
    [generatedImage,segMap] =
evaluatePix2PixHD(cpxds,idx,imageSize,dlnetGenerator);

    % For display, convert the labels from categorical labels to RGB colors
    coloredSegMap = label2rgb(segMap);

    % Display the pixel label image and generated scene image in a montage
    figure
    montage({coloredSegMap generatedImage})
    title(['Custom Pixel Label Image ',num2str(idx),' and Generated Scene
Image'])
end
```

Custom Pixel Label Image 1 and Generated Scene Image



Custom Pixel Label Image 2 and Generated Scene Image



Model Gradients Function

The modelGradients helper function calculates the gradients and adversarial loss for the generator and discriminator. The function also calculates the feature matching loss and VGG loss for the generator.

```
function [gradParamsG,gradParamsDScale1,gradParamsDScale2,lossGGAN,lossGFM,lossGVGG,lossD] = ...
    modelGradients(inputSegMap,realImage,generator,discriminatorScale1,discriminatorScale2,netVGG)

    % Compute the image generated by the generator given the input semantic map.
    generatedImage = forward(generator,inputSegMap);

    % Define the loss weights
    lambdaDiscriminator = 1;
    lambdaGenerator = 1;
    lambdaFeatureMatching = 5;
    lambdaVGG = 5;

    % Concatenate the image to be classified and the semantic map
    inpDiscriminatorReal = cat(3,inputSegMap,realImage);
    inpDiscriminatorGenerated = cat(3,inputSegMap,generatedImage);

    % Compute the adversarial loss for the discriminator and the generator for first scale.
    [DLossScale1,GLossScale1,realPredScale1D,fakePredScale1G] = pix2pixHDAdverserialLoss( ...
        inpDiscriminatorReal,inpDiscriminatorGenerated,discriminatorScale1);

    % Scale the generated image, the real image, and the input semantic map to half size
    resizedRealImage = dlresize(realImage, 'Scale',0.5, 'Method',"linear");
    resizedGeneratedImage = dlresize(generatedImage,'Scale',0.5,'Method',"linear");
    resizedinputSegMap = dlresize(inputSegMap,'Scale',0.5,'Method',"nearest");

    % Concatenate the image to be classified and the semantic map
    inpDiscriminatorReal = cat(3,resizedinputSegMap,resizedRealImage);
    inpDiscriminatorGenerated = cat(3,resizedinputSegMap,resizedGeneratedImage);

    % Compute the adversarial loss for the discriminator and the generator for second scale.
    [DLossScale2,GLossScale2,realPredScale2D,fakePredScale2G] = pix2pixHDAdverserialLoss( ...
        inpDiscriminatorReal,inpDiscriminatorGenerated,discriminatorScale2);
```

```
    % Compute the feature matching loss for first scale.
    FMLossScale1 = pix2pixHDFeatureMatchingLoss(realPredScale1D,fakePredScale1G);
    FMLossScale1 = FMLossScale1 * lambdaFeatureMatching;

    % Compute the feature matching loss for second scale.
    FMLossScale2 = pix2pixHDFeatureMatchingLoss(realPredScale2D,fakePredScale2G);
    FMLossScale2 = FMLossScale2 * lambdaFeatureMatching;

    % Compute the VGG loss
    VGGLoss = pix2pixHDVGGLoss(realImage,generatedImage,netVGG);
    VGGLoss = VGGLoss * lambdaVGG;

    % Compute the combined generator loss
    lossGCombined = GLossScale1 + GLossScale2 + FMLossScale1 + FMLossScale2 + VGGLoss;
    lossGCombined = lossGCombined * lambdaGenerator;

    % Compute gradients for the generator
    gradParamsG = dlgradient(lossGCombined,generator.Learnables,'RetainData',true);

    % Compute the combined discriminator loss
    lossDCombined = (DLossScale1 + DLossScale2)/2 * lambdaDiscriminator;

    % Compute gradients for the discriminator scale1
    gradParamsDScale1 = dlgradient(lossDCombined,discriminatorScale1.Learnables,'RetainData',true);

    % Compute gradients for the discriminator scale2
    gradParamsDScale2 = dlgradient(lossDCombined,discriminatorScale2.Learnables);

    % Log the values for displaying later
    lossD = gather(extractdata(lossDCombined));
    lossGGAN = gather(extractdata(GLossScale1 + GLossScale2));
    lossGFM  = gather(extractdata(FMLossScale1 + FMLossScale2));
    lossGVGG = gather(extractdata(VGGLoss));
end
```

Adversarial Loss Function

The helper function pix2pixHDAdverserialLoss computes the adversarial loss gradients for the generator and the discriminator. The function also returns feature maps of the real image and synthetic images.

```
function [DLoss,GLoss,realPredFtrsD,genPredFtrsD] = pix2pixHDAdverserialLoss(inpReal,inpGenerated,discriminator)

    % Discriminator layer names containing feature maps
    featureNames = {'act_top','act_mid_1','act_mid_2','act_tail','conv2d_final'};

    % Get the feature maps for the real image from the discriminator
    realPredFtrsD = cell(size(featureNames));
    [realPredFtrsD{:}] = forward(discriminator,inpReal,"Outputs",featureNames);

    % Get the feature maps for the generated image from the discriminator
    genPredFtrsD = cell(size(featureNames));
    [genPredFtrsD{:}] = forward(discriminator,inpGenerated,"Outputs",featureNames);

    % Get the feature map from the final layer to compute the loss
    realPredD = realPredFtrsD{end};
    genPredD = genPredFtrsD{end};

    % Compute the discriminator loss
    DLoss = (1 - realPredD).^2 + (genPredD).^2;
    DLoss = mean(DLoss,"all");

    % Compute the generator loss
    GLoss = (1 - genPredD).^2;
    GLoss = mean(GLoss,"all");
end
```

Feature Matching Loss Function

The helper function pix2pixHDFeatureMatchingLoss computes the feature matching loss between a real image and a synthetic image generated by the generator.

```
function featureMatchingLoss = pix2pixHDFeatureMatchingLoss(realPredFtrs,genPredFtrs)

    % Number of features
    numFtrsMaps = numel(realPredFtrs);

    % Initialize the feature matching loss
    featureMatchingLoss = 0;

    for i = 1:numFtrsMaps
        % Get the feature maps of the real image
        a = extractdata(realPredFtrs{i});
        % Get the feature maps of the synthetic image
        b = genPredFtrs{i};

        % Compute the feature matching loss
        featureMatchingLoss = featureMatchingLoss + mean(abs(a - b),"all");
    end
end
```

Perceptual VGG Loss Function

The helper function pix2pixHDVGGLoss computes the perceptual VGG loss between a real image and a synthetic image generated by the generator.

```
function vggLoss = pix2pixHDVGGLoss(realImage,generatedImage,netVGG)

    featureWeights = [1.0/32 1.0/16 1.0/8 1.0/4 1.0];

    % Initialize the VGG loss
    vggLoss = 0;

    % Specify the names of the layers with desired feature maps
    featureNames = ["relu1_1","relu2_1","relu3_1","relu4_1","relu5_1"];

    % Extract the feature maps for the real image
    activReal = cell(size(featureNames));
    [activReal{:}] = forward(netVGG,realImage,"Outputs",featureNames);

    % Extract the feature maps for the synthetic image
    activGenerated = cell(size(featureNames));
    [activGenerated{:}] = forward(netVGG,generatedImage,"Outputs",featureNames);

    % Compute the VGG loss
    for i = 1:numel(featureNames)
        vggLoss = vggLoss + featureWeights(i)*mean(abs(activReal{i} - activGenerated{i}),"all");
    end
end
```

Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data

Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data

This example shows how to use 3-D simulation data to train a semantic segmentation network and fine-tune it to real-world data using generative adversarial networks (GANs).

This example uses 3-D simulation data generated by Driving Scenario Designer and the Unreal Engine®.

The 3-D simulation environment generates the images and the corresponding ground truth pixel labels. Using the simulation data avoids the annotation process, which is both tedious and requires a large amount of human effort. However, domain shift models trained on only simulation data do not perform well on real-world data sets. To address this, you can use domain adaptation to fine-tune the trained model to work on a real-world data set.

Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data

This example uses AdaptSegNet, a network that adapts the structure of the output segmentation predictions, which look alike irrespective of the input domain. The AdaptSegNet network is based on the GAN model and consists of two networks that are trained simultaneously to maximize the performance of both:

- Generator — Network trained to generate high-quality segmentation results from real or simulated input images
- Discriminator — Network that compares and attempts to distinguish whether the segmentation predictions of the generator are from real or simulated data

To fine-tune the AdaptSegNet model for real-world data, this example uses a subset of the CamVid data and adapts the model to generate high-quality segmentation predictions on the CamVid data.

Download Pretrained Network

Download the pretrained network. The pretrained model allows you to run the entire example without having to wait for training to complete. If you want to train the network, set the doTraining variable to true.

```
doTraining = false;
if ~doTraining
    pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/data/trainedAdaptSegGANNet.mat';
    pretrainedFolder = fullfile(tempdir,'pretrainedNetwork');
    pretrainedNetwork = fullfile(pretrainedFolder,'trainedAdaptSegGANNet.mat');
    if ~exist(pretrainedNetwork,'file')
        mkdir(pretrainedFolder);
        disp('Downloading pretrained network (57 MB)...');
        websave(pretrainedNetwork,pretrainedURL);
    end
    pretrained = load(pretrainedNetwork);
    dlnetGenerator = pretrained.dlnetGenerator;
end
```

Download Data Sets

Download the simulation and real data sets by using the downloadDataset function, defined in the Supporting Functions section of this example. The downloadDataset function downloads the entire CamVid data set and partition the data into training and test sets.

The simulation data set was generated by Driving Scenario Designer. The generated scenarios, which consist of 553 photorealistic images with labels, were rendered by the Unreal Engine. You use this data set to train the model.

The real data set is a subset of the CamVid data set from the University of Cambridge. To adapt the model to real-world data, 69 CamVid images. To evaluate the trained model, you use 368 CamVid images.

The downloaded files include the pixel labels for the real domain, but note that you do not use these pixel labels in the training process. This example uses the real domain pixel labels only to calculate the mean intersection over union (IoU) value to evaluate the efficacy of the trained model.

Download Data Sets

```
simulationDataURL = 'https://ssd.mathworks.com/supportfiles/vision/data/SimulationDrivingDataset.zip';
realImageDataURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.zip';
realLabelDataURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.zip';

simulationDataLocation = fullfile(tempdir,'SimulationData');
realDataLocation = fullfile(tempdir,'RealData');
[simulationImagesFolder, simulationLabelsFolder, realImagesFolder, realLabelsFolder, ...
    realTestImagesFolder, realTestLabelsFolder] = ...
    downloadDataset(simulationDataLocation,simulationDataURL,realDataLocation,realImageDataURL,realLabelDataURL);
```

Load Simulation and Real Data

Use imageDatastore to load the simulation and real data sets for training. By using an image datastore, you can efficiently load a large collection of images on disk.

```
simData = imageDatastore(simulationImagesFolder);
realData = imageDatastore(realImagesFolder);
```

Preview images from the simulation data set and real data set.

```
simImage = preview(simData);
realImage = preview(realData);
montage({simImage,realImage})
```

Load Simulation and Real Data

The real and simulated images look very different. Consequently, models trained on simulated data and evaluated on real data perform poorly due to domain shift.

Load Pixel-Labeled Images for Simulation Data and Real Data

Load the simulation pixel label image data by using pixelLabelDatastore. A pixel label datastore encapsulates the pixel label data and the label ID to a class name mapping.

For this example, specify five classes useful for an automated driving application: road, background, pavement, sky, and car.

```
classes = [
    "Road"
    "Background"
    "Pavement"
    "Sky"
    "Car"
    ];
numClasses = numel(classes);
```

Load Pixel-Labeled Images for Simulation Data and Real Data

The simulation data set has eight classes. Reduce the number of classes from eight to five by grouping the building, tree, traffic signal, and light classes from the original data set into a single background class. Return the grouped label IDs by using the helper function simulationPixelLabelIDs. This helper function is attached to the example as a supporting file.

```
labelIDs = simulationPixelLabelIDs;
```

Use the classes and label IDs to create a pixel label datastore of the simulation data.

```
simLabels = pixelLabelDatastore(simulationLabelsFolder,classes,labelIDs);
```

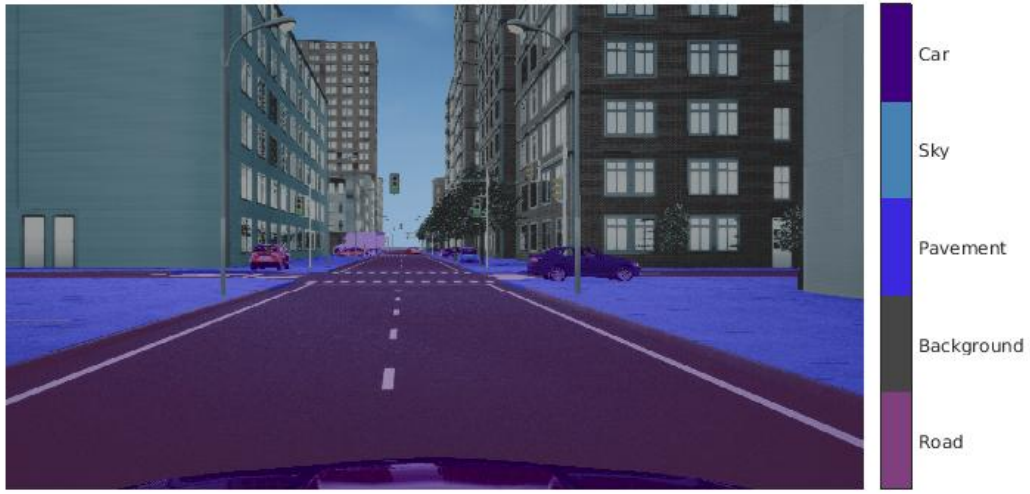
Initialize the colormap for the segmented images using the helper function domainAdaptationColorMap.

```
dmap = domainAdaptationColorMap;
```

Load Pixel-Labeled Images for Simulation Data and Real Data

Preview a pixel-labeled image by overlaying the label on top of the image using the labeloverlay function.

```
simImageLabel = preview(simLabels);
overlayImageSimulation = labeloverlay(simImage,simImageLabel,'ColorMap',dmap);
figure
imshow(overlayImageSimulation)
labelColorbar(dmap,classes);
```



Load Pixel-Labeled Images for Simulation Data and Real Data

Shift the simulation and real data used for training to zero center, to center the data around the origin, by using the transform function and the preprocessData helper function, defined in the Supporting Functions section.

```
preprocessedSimData = transform(simData, @(simdata)preprocessData(simdata));
preprocessedRealData = transform(realData, @(realdata)preprocessData(realdata));
```

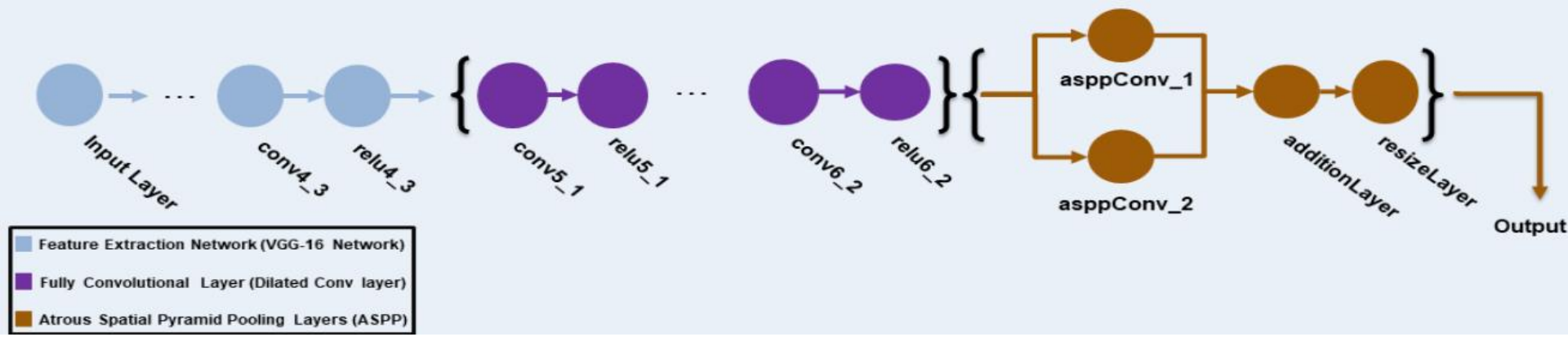
Use the combine function to combine the transformed image datastore and pixel label datastores of the simulation domain. The training process does not use the pixel labels of real data.

```
combinedSimData = combine(preprocessedSimData,simLabels);
```

Define AdaptSegNet Generator

This example modifies the VGG-16 network pretrained on ImageNet to a fully convolutional network. To enlarge the receptive fields, dilated convolutional layers with strides of 2 and 4 are added. This makes the output feature map resolution one-eighth of the input size. Atrous spatial pyramid pooling (ASPP) is used to provide multiscale information and is followed by a resize2dlayer with an upsampling factor of 8 to resize the output to the size of the input.

The AdaptSegNet generator network used in this example is illustrated in the following diagram.



Define AdaptSegNet Generator

To get a pretrained VGG-16 network, install the Deep Learning Toolbox™ Model for VGG-16 Network.

```
net = vgg16;
```

To make the VGG-16 network suitable for semantic segmentation, remove all VGG layers after 'relu4_3'.

```
vggLayers = net.Layers(2:24);
```

Create an image input layer of size 1280-by-720-by-3 for the generator.

```
inputSizeGenerator = [1280 720 3];
inputLayer =
imageInputLayer(inputSizeGenerator,'Normalization','None','Name','inputLayer');
```

Define AdaptSegNet Generator

Create fully convolutional network layers. Use dilation factors of 2 and 4 to enlarge the respective fields.

```
fcnlayers = [
    convolution2dLayer([3 3], 360,'DilationFactor',[2 2],'Padding',[2 2 2 2],'Name','conv5_1',
'WeightsInitializer','narrow-normal','BiasInitializer','zeros')
    reluLayer('Name','relu5_1')
    convolution2dLayer([3 3], 360,'DilationFactor',[2 2],'Padding',[2 2 2 2] ,'Name','conv5_2',
'WeightsInitializer','narrow-normal','BiasInitializer','zeros')
    reluLayer('Name','relu5_2')
    convolution2dLayer([3 3], 360,'DilationFactor',[2 2],'Padding',[2 2 2 2],'Name','conv5_3',
'WeightsInitializer','narrow-normal','BiasInitializer','zeros')
    reluLayer('Name','relu5_3')
    convolution2dLayer([3 3], 480,'DilationFactor',[4 4],'Padding',[4 4 4 4],'Name','conv6_1',
'WeightsInitializer','narrow-normal','BiasInitializer','zeros')
    reluLayer('Name','relu6_1')
    convolution2dLayer([3 3], 480,'DilationFactor',[4 4],'Padding',[4 4 4 4] ,'Name','conv6_2',
'WeightsInitializer','narrow-normal','BiasInitializer','zeros')
    reluLayer('Name','relu6_2')
    ];
```

Define AdaptSegNet Generator

Combine the layers and create the layer graph.

```
layers = [
    inputLayer
    vggLayers
    fcnlayers
    ];
lgraph = layerGraph(layers);
```

ASPP is used to provide multiscale information. Add the ASPP module to the layer graph with a filter size equal to the number of channels by using the addASPPToNetwork helper function, defined in the Supporting Functions section.

```
lgraph  = addASPPToNetwork(lgraph, numClasses);
```

ISSUES

ANALYSIS RESULT

	Name	Type	Activations	Learnable Prope…
1	inputLayer 1280×720×3 images	Image Input	1280(S) × 720(S) × 3(C) × 1(B)	-
2	conv1_1 64 3×3×3 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	1280(S) × 720(S) × 64(C) × 1(B)	Weig… 3 × 3 × 3 … Bias 1 × 1 × 64
3	relu1_1 ReLU	ReLU	1280(S) × 720(S) × 64(C) × 1(B)	-
4	conv1_2 64 3×3×64 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	1280(S) × 720(S) × 64(C) × 1(B)	Weig… 3 × 3 × 64… Bias 1 × 1 × 64
5	relu1_2 ReLU	ReLU	1280(S) × 720(S) × 64(C) × 1(B)	-
6	pool1 2×2 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	640(S) × 360(S) × 64(C) × 1(B)	-
7	conv2_1 128 3×3×64 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	640(S) × 360(S) × 128(C) × 1(B)	Weig… 3 × 3 × 64 … Bias 1 × 1 × 128
8	relu2_1 ReLU	ReLU	640(S) × 360(S) × 128(C) × 1(B)	-
9	conv2_2 128 3×3×128 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	640(S) × 360(S) × 128(C) × 1(B)	Wei… 3 × 3 × 128… Bias 1 × 1 × 128
10	relu2_2 ReLU	ReLU	640(S) × 360(S) × 128(C) × 1(B)	-
11	pool2 2×2 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	320(S) × 180(S) × 128(C) × 1(B)	-
12	conv3_1 256 3×3×128 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	320(S) × 180(S) × 256(C) × 1(B)	Wei… 3 × 3 × 128… Bias 1 × 1 × 256
13	relu3_1 ReLU	ReLU	320(S) × 180(S) × 256(C) × 1(B)	-
14	conv3_2 256 3×3×256 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	320(S) × 180(S) × 256(C) × 1(B)	Wei… 3 × 3 × 256… Bias 1 × 1 × 256
15	relu3_2 ReLU	ReLU	320(S) × 180(S) × 256(C) × 1(B)	-
16	conv3_3 256 3×3×256 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	320(S) × 180(S) × 256(C) × 1(B)	Wei… 3 × 3 × 256… Bias 1 × 1 × 256
17	relu3_3 ReLU	ReLU	320(S) × 180(S) × 256(C) × 1(B)	-
18	pool3 2×2 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	160(S) × 90(S) × 256(C) × 1(B)	-
19	conv4_1 512 3×3×256 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	160(S) × 90(S) × 512(C) × 1(B)	Wei… 3 × 3 × 256… Bias 1 × 1 × 512
20	relu4_1 ReLU	ReLU	160(S) × 90(S) × 512(C) × 1(B)	-
21	conv4_2 512 3×3×512 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	160(S) × 90(S) × 512(C) × 1(B)	Wei… 3 × 3 × 512… Bias 1 × 1 × 512
22	relu4_2 ReLU	ReLU	160(S) × 90(S) × 512(C) × 1(B)	-
23	conv4_3 512 3×3×512 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	160(S) × 90(S) × 512(C) × 1(B)	Wei… 3 × 3 × 512… Bias 1 × 1 × 512

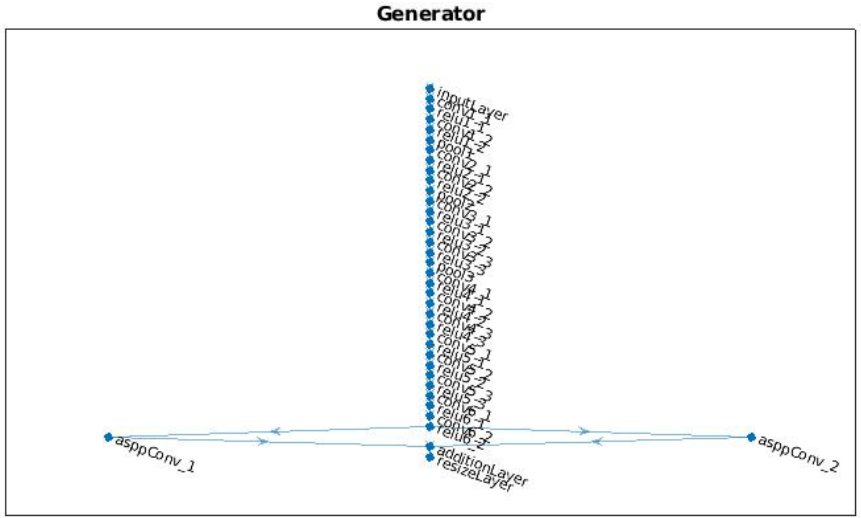
Define AdaptSegNet Generator

Apply resize2dLayer with an upsampling factor of 8 to make the output match the size of the input.

```
upSampleLayer =
resize2dLayer('Scale',8,'Method','bilinear','Name','resizeLayer');
lgraphGenerator = addLayers(lgraph,upSampleLayer);
lgraphGenerator =
connectLayers(lgraphGenerator,'additionLayer','resizeLayer');
```

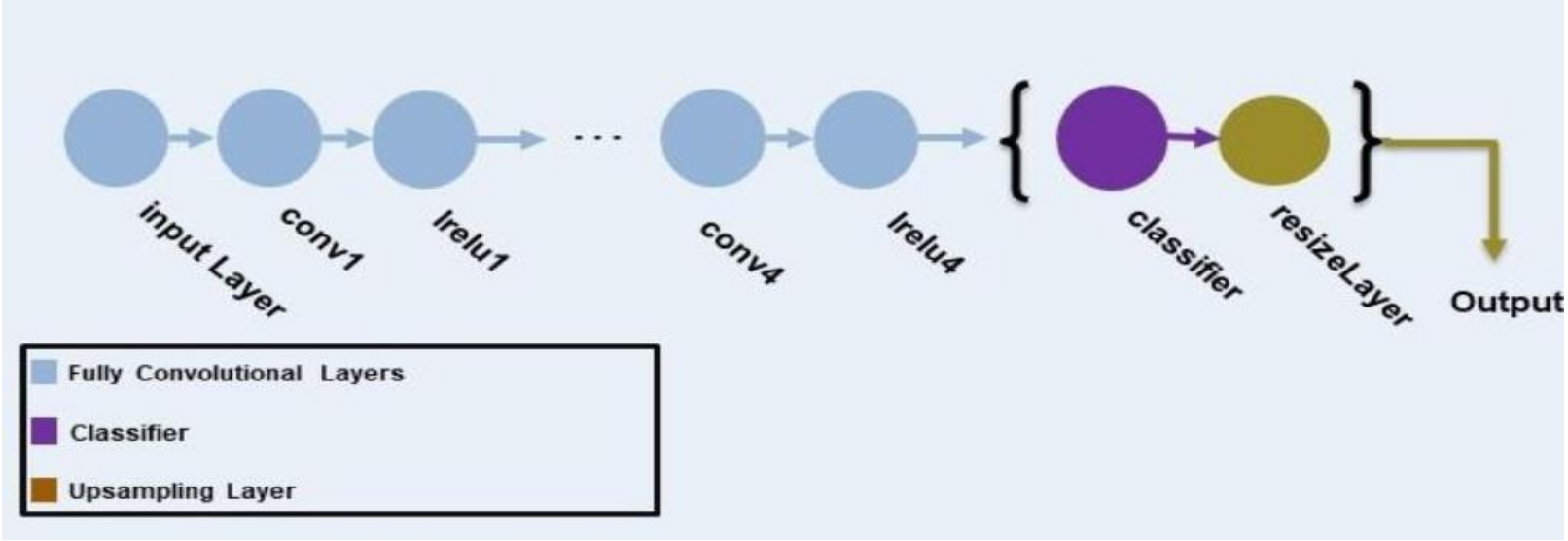
Visualize the generator network in a plot.

```
plot(lgraphGenerator)
title("Generator")
```



Define AdaptSeg Discriminator

The discriminator network consists of five convolutional layers with a kernel size of 3 and a stride of 2, where the number of channels is {64, 128, 256, 512, 1}. Each layer is followed by a leaky ReLU layer parameterized by a scale of 0.2, except for the last layer. resize2dLayer is used to resize the output of the discriminator. Note that this example does not use batch normalization, as the discriminator is jointly trained with the segmentation network using a small batch size.



Define AdaptSeg Discriminator

Create an image input layer of size 1280-by-720-by-numClasses that takes in the segmentation predictions of the simulation and real domains.

```
inputSizeDiscriminator = [1280 720 numClasses];
```

Create fully convolutional layers and generate the discriminator layer graph.

```
numChannelsFactor = 64;   % Factor for number of channels in convolution layer.
resizeScale = 64;   % Scale factor to resize the output of the discriminator.
leakyReLUScale = 0.2;% Scalar multiplier for leaky ReLU layers.
% Create the layers of the discriminator.
layers = [
imageInputLayer(inputSizeDiscriminator,'Normalization','none','Name','inputLayer')
convolution2dLayer(3,numChannelsFactor,'Stride',2,'Padding',1,'Name','conv1',
'WeightsInitializer','narrow-normal','BiasInitializer','narrow-normal')
leakyReluLayer(leakyReLUScale,'Name','lrelu1')
convolution2dLayer(3,numChannelsFactor*2,'Stride',2,'Padding',1,'Name','conv2',
'WeightsInitializer','narrow-normal','BiasInitializer','narrow-normal')
leakyReluLayer(leakyReLUScale,'Name','lrelu2')
convolution2dLayer(3,numChannelsFactor*4,'Stride',2,'Padding',1,'Name','conv3',
'WeightsInitializer','narrow-normal','BiasInitializer','narrow-normal')
leakyReluLayer(leakyReLUScale,'Name','lrelu3')
convolution2dLayer(3,numChannelsFactor*8,'Stride',2,'Padding',1,'Name','conv4',
'WeightsInitializer','narrow-normal','BiasInitializer','narrow-normal')
leakyReluLayer(leakyReLUScale,'Name','lrelu4')
convolution2dLayer(3,1,'Stride',2,'Padding',1,'Name','classifer',
'WeightsInitializer','narrow-normal','BiasInitializer','narrow-normal')
resize2dLayer('Scale', resizeScale,'Method','bilinear','Name','resizeLayer');  ];

% Create the layer graph of the discriminator.
lgraphDiscriminator  = layerGraph(layers);
```

Define AdaptSeg Discriminator

Visualize the discriminator network in a plot.

```
plot(lgraphDiscriminator)
title("Discriminator")
```

Specify Training Options

- Set the total number of iterations to 5000. By doing so, you train the network for around 10 epochs.
- Set the learning rate for the generator to 2.5e-4.
- Set the learning rate for the discriminator to 1e-4.
- Set the L2 regularization factor to 0.0005.
- The learning rate exponentially decreases based on the formula $learningrate \times \left[\frac{iteration}{total\ \ iterations}\right]^{power}$. This decrease helps to stabilize the gradients at higher iterations. Set the power to 0.9.
- Set the weight of the adversarial loss to 0.001.
- Initialize the velocity of the gradient as [ ]. This value is used by SGDM to store the velocity of the gradients.
- Initialize the moving average of the parameter gradients as [ ]. This value is used by Adam initializer to store the average of parameter gradients.
- Initialize the moving average of squared parameter gradients as [ ]. This value is used by Adam initializer to store the average of the squared parameter gradients.
- Set the mini-batch size to 1.

Specify Training Options

```
numIterations = 5000;
learnRateGenBase = 2.5e-4;
learnRateDisBase = 1e-4;
l2Regularization = 0.0005;
power = 0.9;
lamdaAdv = 0.001;
vel= [];
averageGrad = [];
averageSqGrad = [];
miniBatchSize = 1;
```

Specify Training Options

Train on a GPU, if one is available. To automatically detect if you have a GPU available, set executionEnvironment to "auto". If you do not have a GPU, or do not want to use one for training, set executionEnvironment to "cpu". To ensure the use of a GPU for training, set executionEnvironment to "gpu".

```
executionEnvironment = "auto";
```

Create the minibatchqueue object from the combined datastore of the simulation domain.

```
mbqTrainingDataSimulation =
minibatchqueue(combinedSimData,"MiniBatchSize",miniBatchSize, ...
    "MiniBatchFormat","SSCB","OutputEnvironment",executionEnvironment);
```

Create the minibatchqueue object from the input image datastore of the real domain.

```
mbqTrainingDataReal =
minibatchqueue(preprocessedRealData,"MiniBatchSize",miniBatchSize, ...
    "MiniBatchFormat","SSCB","OutputEnvironment",executionEnvironment);
```

Train Model

Train the model using a custom training loop. The helper function modelGradients calculate the gradients and losses for the generator and discriminator. Create the training progress plot using configureTrainingLossPlotter, attached to this example as a supporting file, and update the training progress using updateTrainingPlots. Loop over the training data and update the network parameters at each iteration. For each iteration:

- Read the image and label information from the minibatchqueue object of the simulation data using the next function.
- Read the image information from the minibatchqueue object of the real data using the next function.
- Evaluate the model gradients using dlfeval and the modelGradients helper function, defined in the Supporting Functions section. modelGradients returns the gradients of the loss with respect to the learnable parameters.
- Update the generator network parameters using the sgdmupdate function.
- Update the discriminator network parameters using the adamupdate function.
- Update the training progress plot for every iteration and display various computed losses.

```
if doTraining

    % Create the dlnetwork object of the generator.
    dlnetGenerator = dlnetwork(lgraphGenerator);

    % Create the dlnetwork object of the discriminator.
    dlnetDiscriminator = dlnetwork(lgraphDiscriminator);

    % Create the subplots for the generator and discriminator loss.
    fig = figure;
    [generatorLossPlotter, discriminatorLossPlotter] = configureTrainingLossPlotter(fig);

    % Loop through the data for the specified number of iterations.
    for iter = 1:numIterations

        % Reset the minibatchqueue of simulation data.
        if ~hasdata(mbqTrainingDataSimulation)
            reset(mbqTrainingDataSimulation);
        end

        % Retrieve the next mini-batch of simulation data and labels.
        [dlX,label] = next(mbqTrainingDataSimulation);

        % Reset the minibatchqueue of real data.
        if ~hasdata(mbqTrainingDataReal)
            reset(mbqTrainingDataReal);
        end
```

```
    % Retrieve the next mini-batch of real data.
    dlZ = next(mbqTrainingDataReal);

    % Evaluate the model gradients and loss using dlfeval and the modelGradients function.
    [gradientGenerator,gradientDiscriminator, lossSegValue, lossAdvValue, lossDisValue] = ...
        dlfeval(@modelGradients,dlnetGenerator,dlnetDiscriminator,dlX,dlZ,label,lamdaAdv);

    % Apply L2 regularization.
    gradientGenerator  = dlupdate(@(g,w) g + l2Regularization*w, gradientGenerator, dlnetGenerator.Learnables);

    % Adjust the learning rate.
    learnRateGen = piecewiseLearningRate(iter,learnRateGenBase,numIterations,power);
    learnRateDis = piecewiseLearningRate(iter,learnRateDisBase,numIterations,power);

     % Update the generator network learnable parameters using the SGDM optimizer.
    [dlnetGenerator.Learnables, vel] = ...
        sgdmupdate(dlnetGenerator.Learnables,gradientGenerator,vel,learnRateGen);

     % Update the discriminator network learnable parameters using the Adam optimizer.
    [dlnetDiscriminator.Learnables, averageGrad, averageSqGrad] = ...
        adamupdate(dlnetDiscriminator.Learnables,gradientDiscriminator,averageGrad,averageSqGrad,iter,learnRateDis) ;

    % Update the training plot with loss values.
    updateTrainingPlots(generatorLossPlotter,discriminatorLossPlotter,iter, ...
        double(gather(extractdata(lossSegValue + lamdaAdv * lossAdvValue))),double(gather(extractdata(lossDisValue))));

end
```

Evaluate Model on Real Test Data

Evaluate the performance of the trained AdaptSegNet network by computing the mean IoU for the test data predictions.

Load the test data using imageDatastore.

```
realTestData = imageDatastore(realTestImagesFolder);
```

The CamVid data set has 32 classes. Use the realpixelLabelIDs helper function to reduce the number of classes to five, as for the simulation data set. The realpixelLabelIDs helper function is attached to this example as a supporting file.

```
labelIDs = realPixelLabelIDs;
```

Use pixelLabelDatastore to load the ground truth label images for the test data.

```
realTestLabels = pixelLabelDatastore(realTestLabelsFolder,classes,labelIDs);
```

Evaluate Model on Real Test Data

Shift the data to zero center to center the data around the origin, as for the training data, by using the transform function and the preprocessData helper function, defined in the Supporting Functions section.

```
preprocessedRealTestData = transform(realTestData, @(realtestdata)preprocessData(realtestdata));
```

Use combine to combine the transformed image datastore and pixel label datastores of the real test data.

```
combinedRealTestData = combine(preprocessedRealTestData,realTestLabels);
```

Create the minibatchqueue object from the combined datastore of the test data. Set "MiniBatchSize" to 1 for ease of evaluating the metrics.

```
mbqimdsTest = minibatchqueue(combinedRealTestData,"MiniBatchSize",1,...
    "MiniBatchFormat","SSCB","OutputEnvironment",executionEnvironment);
```

To generate the confusion matrix cell array, use the helper function predictSegmentationLabelsOnTestSet on minibatchqueue object of test data.

```
imageSetConfusionMat = predictSegmentationLabelsOnTestSet(dlnetGenerator,mbqimdsTest);
```

Evaluate Model on Real Test Data

Use evaluateSemanticSegmentation to measure semantic segmentation metrics on the test set confusion matrix.

```
metrics = evaluateSemanticSegmentation(imageSetConfusionMat,classes,'Verbose',false);
```

To see the data set level metrics, inspect metrics.DataSetMetrics.

```
metrics.DataSetMetrics
```

The data set metrics provide a high-level overview of network performance. To see the impact each class has on the overall performance, inspect the per-class metrics using metrics.ClassMetrics.

```
metrics.ClassMetrics
```

Evaluate Model on Real Test Data

ans=5×2 table

	Accuracy	IoU
Road	0.9147	0.81301
Background	0.93418	0.85518
Pavement	0.33373	0.27105
Sky	0.82652	0.81109
Car	0.83586	0.47399

The data set performance is good, but the class metrics show that the car and pavement classes are not segmented well. Training the network using additional data can yield improved results.

Segment Image

Run the trained network on one test image to check the segmented output prediction.

```
% Read the image from the test data.
data = readimage(realTestData,350);

% Perform the preprocessing step of zero shift on the image.
processeddata = preprocessData(data);

% Convert the data to dlarray.
processeddata = dlarray(processeddata,'SSCB');

% Predict the output of the network.
[genPrediction, ~] = forward(dlnetGenerator,processeddata);

% Get the label, which is the index with the maximum value in the channel dimension.
[~, labels] = max(genPrediction,[],3);

% Overlay the predicted labels on the image.
segmentedImage = labeloverlay(data,uint8(gather(extractdata(labels))),'Colormap',dmap);
```

Segment Image

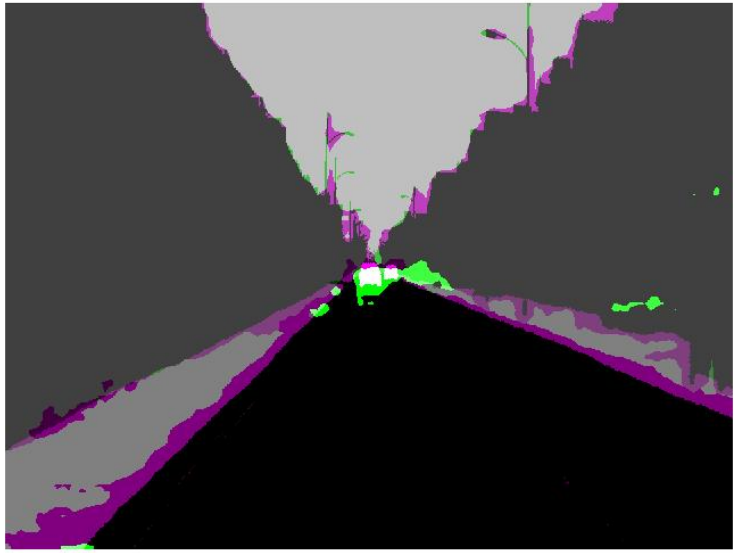
Display the results.

```
figure
imshow(segmentedImage);
labelColorbar(dmap,classes);
```

Segment Image

Compare the label results with the expected ground truth stored in realTestLabels. The green and magenta regions highlight areas where the segmentation results differ from the expected ground truth.

```
expectedResult = readimage(realTestLabels,350);
actual = uint8(gather(extractdata(labels)));
expected = uint8(expectedResult);
figure
imshowpair(actual,expected)
```



Model Gradients Function

The helper function modelGradients calculates the gradients and adversarial loss for the generator and discriminator. The function also calculates the segmentation loss for the generator and the cross-entropy loss for the discriminator. As no state information is required to be remembered between the iterations for both generator and discriminator networks, the states are not updated.

```
function [gradientGenerator, gradientDiscriminator, lossSegValue, lossAdvValue, lossDisValue] = ...
modelGradients(dlnetGenerator, dlnetDiscriminator, dlX, dlZ, label, lamdaAdv)

% Labels for adversarial training.
simulationLabel = 0;
realLabel = 1;

% Extract the predictions of the simulation from the generator.
[genPredictionSimulation, ~] = forward(dlnetGenerator,dlX);

% Compute the generator loss.
lossSegValue = segmentationLoss(genPredictionSimulation,label);

% Extract the predictions of the real data from the generator.
[genPredictionReal, ~] = forward(dlnetGenerator,dlZ);

% Extract the softmax predictions of the real data from the discriminator.
disPredictionReal = forward(dlnetDiscriminator,softmax(genPredictionReal));

% Create a matrix of simulation labels of real prediction size.
Y = simulationLabel * ones(size(disPredictionReal));

% Compute the adversarial loss to make the real distribution close to the simulation label.
lossAdvValue = mse(disPredictionReal,Y)/numel(Y(:));

% Compute the gradients of the generator with regard to loss.
gradientGenerator = dlgradient(lossSegValue + lamdaAdv*lossAdvValue,dlnetGenerator.Learnables);
```

```
% Extract the softmax predictions of the simulation from the discriminator.
disPredictionSimulation = forward(dlnetDiscriminator,softmax(genPredictionSimulation));

% Create a matrix of simulation labels of simulation prediction size.
Y = simulationLabel * ones(size(disPredictionSimulation));

% Compute the discriminator loss with regard to simulation class.
lossDisValueSimulation = mse(disPredictionSimulation,Y)/numel(Y(:));

% Extract the softmax predictions of the real data from the discriminator.
disPredictionReal = forward(dlnetDiscriminator,softmax(genPredictionReal));

% Create a matrix of real labels of real prediction size.
Y = realLabel * ones(size(disPredictionReal));

% Compute the discriminator loss with regard to real class.
lossDisValueReal = mse(disPredictionReal,Y)/numel(Y(:));

% Compute the total discriminator loss.
lossDisValue = lossDisValueSimulation + lossDisValueReal;

% Compute the gradients of the discriminator with regard to loss.
gradientDiscriminator = dlgradient(lossDisValue,dlnetDiscriminator.Learnables);

end
```

Segmentation Loss Function

The helper function segmentationLoss computes the feature segmentation loss, which is defined as the cross-entropy loss for the generator using the simulation data and its respective ground truth. The helper function computes the loss by using the crossentropy function.

```
function loss = segmentationLoss(predict, target)

% Generate the one-hot encodings of the ground truth.
oneHotTarget = onehotencode(categorical(extractdata(target)),4);

% Convert the one-hot encoded data to dlarray.
oneHotTarget = dlarray(oneHotTarget,'SSBC');

% Compute the softmax output of the predictions.
predictSoftmax = softmax(predict);

% Compute the cross-entropy loss.
loss =  crossentropy(predictSoftmax,oneHotTarget,'TargetCategories','exclusive')/(numel(oneHotTarget)/2);
end
```

addASPPToNetwork Function

The helper function addASPPToNetwork creates the atrous spatial pyramid pooling (ASPP) layers and adds them to the input layer graph. The function returns the layer graph with ASPP layers connected to it.

```
function lgraph  = addASPPToNetwork(lgraph, numClasses)

% Define the ASPP dilation factors.
asppDilationFactors = [6,12];

% Define the ASPP filter sizes.
asppFilterSizes = [3,3];

% Extract the last layer of the layer graph.
lastLayerName = lgraph.Layers(end).Name;

% Define the addition layer.
addLayer = additionLayer(numel(asppDilationFactors),'Name','additionLayer');

% Add the addition layer to the layer graph.
lgraph = addLayers(lgraph,addLayer);

% Create the ASPP layers connected to the addition layer
% and connect the layer graph.
for i = 1: numel(asppDilationFactors)
    asppConvName = "asppConv_" + string(i);
    branchFilterSize = asppFilterSizes(i);
    branchDilationFactor = asppDilationFactors(i);
    asspLayer  = convolution2dLayer(branchFilterSize, numClasses,'DilationFactor', branchDilationFactor,...
        'Padding','same','Name',asppConvName,'WeightsInitializer','narrow-normal','BiasInitializer','zeros');
    lgraph = addLayers(lgraph,asspLayer);
    lgraph = connectLayers(lgraph,lastLayerName,asppConvName);
    lgraph = connectLayers(lgraph,asppConvName,strcat(addLayer.Name,'/',addLayer.InputNames{i}));
end
end
```

predictSegmentationLabelsOnTestSet Function

The helper function predictSegmentationLabelsOnTestSet calculates the confusion matrix of the predicted and ground truth labels using the segmentationConfusionMatrix function.

```
function confusionMatrix =  predictSegmentationLabelsOnTestSet(net, minbatchTestData)

confusionMatrix = {};
i = 1;
while hasdata(minbatchTestData)

    % Use next to retrieve a mini-batch from the datastore.
    [dlX, gtlabels] = next(minbatchTestData);

    % Predict the output of the network.
    [genPrediction, ~] = forward(net,dlX);

    % Get the label, which is the index with maximum value in the channel dimension.
    [~, labels] = max(genPrediction,[],3);

    % Get the confusion matrix of each image.
    confusionMatrix{i}  = segmentationConfusionMatrix(double(gather(extractdata(labels))),double(gather(extractdata(gtlabels))));

    i = i+1;
end

confusionMatrix = confusionMatrix';

end
```

piecewiseLearningRate Function

The helper function piecewiseLearningRate computes the current learning rate based on the iteration number.

```
function lr = piecewiseLearningRate(i, baseLR, numIterations, power)

fraction = i/numIterations;
factor = (1 - fraction)^power * 1e1;
lr = baseLR * factor;

end
```

preprocessData Function

The helper function preprocessData performs a zero center shift by subtracting the number of the image channels by the respective mean.

```
function data = preprocessData(data)

% Extract respective channels.
rc = data(:,:,1);
gc = data(:,:,2);
bc = data(:,:,3);

% Compute the respective channel means.
r = mean(rc(:));
g = mean(gc(:));
b = mean(bc(:));

% Shift the data by the mean of respective channel.
data = single(data) - single(shiftdim([r g b],-1));
end
```

Example code

```
%Generate Image from Segmentation Map Using Deep Learning
openExample('deeplearning_shared/SynthesizeSegmentationMapUsingDeepLearningExample')

%Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data
openExample('deeplearning_shared/TrainADeepLearningSemanticSegmentationNetUsing3DSimDataExample')
```