Deep Learning Models – GAN (cont.)

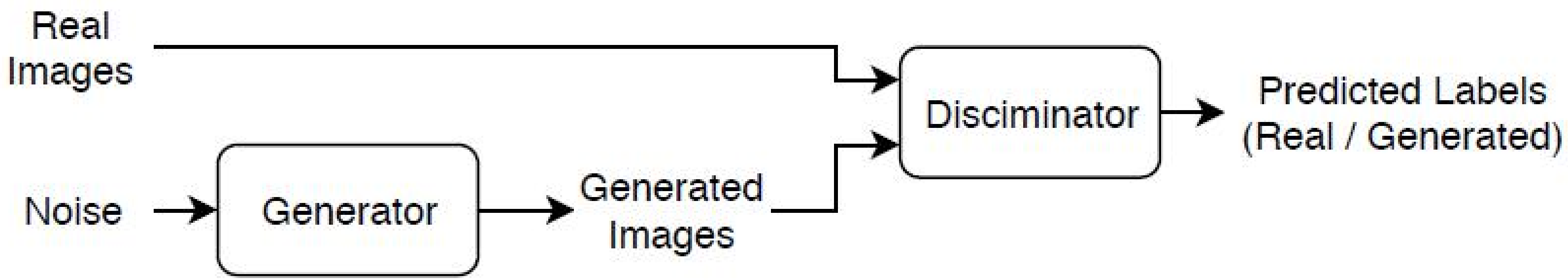
- Conditional generative adversarial network and more (cGAN, wGAN)

Outline

- Conditional Generative Adversarial Network (cGAN)
- Wasserstein Generative Adversarial Network (wGAN)

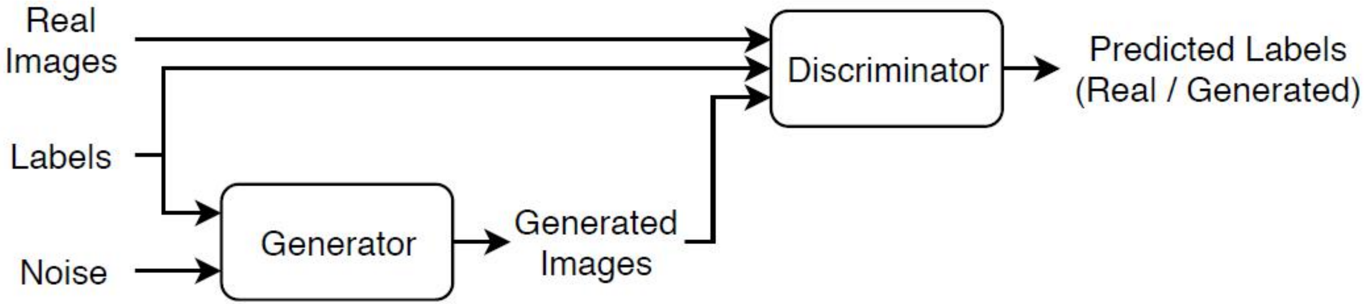
Generative Adversarial Network (GAN)

- A generative adversarial network (GAN) is a type of deep learning network that can generate data with similar characteristics as the input real data. A GAN consists of two networks that train together:
  - Generator — Given a vector of random values (latent inputs) as input, this network generates data with the same structure as the training data.
  - Discriminator — Given batches of data containing observations from both the training data, and generated data from the generator, this network attempts to classify the observations as "real" or "generated".



conditional generative adversarial network (CGAN)

- A conditional generative adversarial network (CGAN) is a type of GAN that also takes advantage of labels during the training process.
  - Generator — Given a label and random array as input, this network generates data with the same structure as the training data observations corresponding to the same label.
  - Discriminator — Given batches of labeled data containing observations from both the training data and generated data from the generator, this network attempts to classify the observations as "real" or "generated".



conditional generative adversarial network (CGAN)

- To train a conditional GAN, train both networks simultaneously to maximize the performance of both:
  - Train the generator to generate data that "fools" the discriminator.
  - Train the discriminator to distinguish between real and generated data.
- To maximize the performance of the generator, maximize the loss of the discriminator when given generated labeled data. That is, the objective of the generator is to generate labeled data that the discriminator classifies as "real".
- To maximize the performance of the discriminator, minimize the loss of the discriminator when given batches of both real and generated labeled data. That is, the objective of the discriminator is to not be "fooled" by the generator.
- Ideally, these strategies result in a generator that generates convincingly realistic data that corresponds to the input labels and a discriminator that has learned strong feature representations that are characteristic of the training data for each label.

conditional generative adversarial network (CGAN)

- Download and extract the Flowers data set.

```
url = 'http://download.tensorflow.org/example_images/flower_photos.tgz';
downloadFolder = tempdir;
filename = fullfile(downloadFolder,'flower_dataset.tgz');

imageFolder = fullfile(downloadFolder,'flower_photos');
if ~exist(imageFolder,'dir')
    disp('Downloading Flowers data set (218 MB)...')
    websave(filename,url);
    untar(filename,downloadFolder)
end
```

conditional generative adversarial network (CGAN)

- Create an image datastore containing the photos of the flowers.

```
datasetFolder = fullfile(imageFolder);
imds = imageDatastore(datasetFolder,IncludeSubfolders=true,LabelSource="foldernames");
```

- View the number of classes.

```
classes = categories(imds.Labels);
numClasses = numel(classes)
numClasses = 5
```

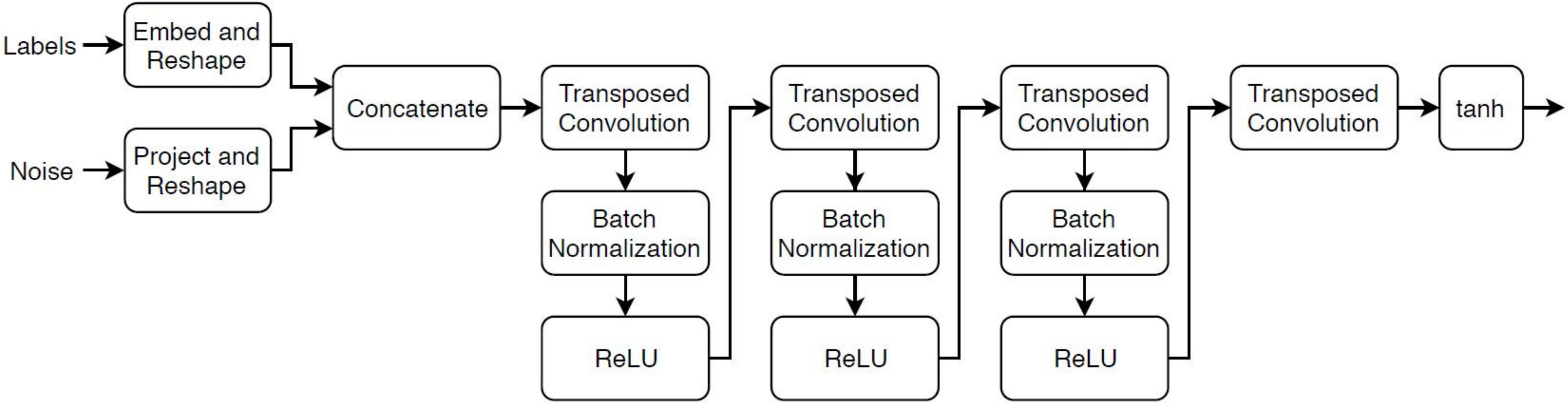
conditional generative adversarial network (CGAN)

- Augment the data to include random horizontal flipping and resize the images to have size 64-by-64.

```
augmenter = imageDataAugmenter(RandXReflection=true);
augimds = augmentedImageDatastore([64 64],imds,DataAugmentation=augmenter);
```

Define Generator Network

- Define the following two-input network, which generates images given random vectors of size 100 and corresponding labels.



Define Generator Network

This network:

- Converts the random vectors of size 100 to 4-by-4-by-1024 arrays using a fully connected layer followed by a reshape operation.
- Converts the categorical labels to embedding vectors and reshapes them to a 4-by-4 array.
- Concatenates the resulting images from the two inputs along the channel dimension. The output is a 4-by-4-by-1025 array.
- Upscales the resulting arrays to 64-by-64-by-3 arrays using a series of transposed convolution layers with batch normalization and ReLU layers.

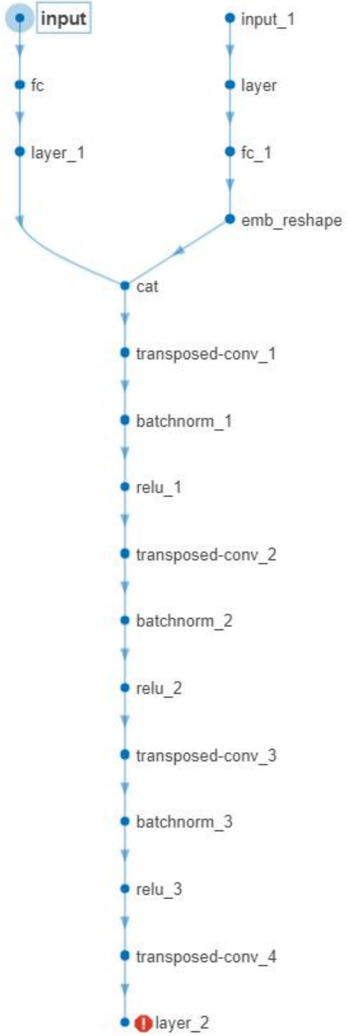
Define Generator Network

Define this network architecture as a layer graph and specify the following network properties.

- For the categorical inputs, use an embedding dimension of 50.
- For the transposed convolution layers, specify 5-by-5 filters with a decreasing number of filters for each layer, a stride of 2, and "same" cropping of the output.
- For the final transposed convolution layer, specify a three 5-by-5 filter corresponding to the three RGB channels of the generated images.
- At the end of the network, include a tanh layer.

```
numLatentInputs = 100;
embeddingDimension = 50;
numFilters = 64;

filterSize = 5;
projectionSize = [4 4 1024];

layersGenerator = [
    featureInputLayer(numLatentInputs)
    fullyConnectedLayer(prod(projectionSize))
    functionLayer(@(X) feature2image(X,projectionSize),Formattable=true)
    concatenationLayer(3,2,Name="cat");
    transposedConv2dLayer(filterSize,4*numFilters)
    batchNormalizationLayer
    reluLayer
    transposedConv2dLayer(filterSize,2*numFilters,Stride=2,Cropping="same")
    batchNormalizationLayer
    reluLayer
    transposedConv2dLayer(filterSize,numFilters,Stride=2,Cropping="same")
    batchNormalizationLayer
    reluLayer
    transposedConv2dLayer(filterSize,3,Stride=2,Cropping="same")
    tanhLayer];
```

```
lgraphGenerator = layerGraph(layersGenerator);

layers = [
    featureInputLayer(1)
    embeddingLayer(embeddingDimension,numClasses)
    fullyConnectedLayer(prod(projectionSize(1:2)))
    functionLayer(@(X) feature2image(X,
                        [projectionSize(1:2) 1]),
                        Formattable=true,
                        Name="emb_reshape")];

lgraphGenerator = addLayers(lgraphGenerator,layers);
lgraphGenerator = connectLayers(lgraphGenerator,
                              "emb_reshape","cat/in2");
```



	Name	Type	Activations	Learnables
1	input 100 features	Feature Input	100	-
2	input_1 1 features	Feature Input	1	-
3	fc 16384 fully connected layer	Fully Connected	16384	Weights 16384×100 Bias 16384×1
4	layer Embedding layer with dimension 50	embeddingLayer	50	Weights 50×5
5	fc_1 16 fully connected layer	Fully Connected	16	Weights 16×50 Bias 16×1
6	layer_1 @(X)feature2image(X,projectionSize)	Function	4×4×1024	-
7	emb_reshape @(X)feature2image(X,[projectionSize(1:2),1])	Function	4×4×1	-
8	cat Concatenation of 2 inputs along dimension 3	Concatenation	4×4×1025	-
9	transposed-conv_1 256 5×5 transposed convolutions with stride [1 1] and cropping [0 0 0 0]	Transposed Convolution	8×8×256	Weigh… 5×5×256×10… Bias 1×1×256
10	batchnorm_1 Batch normalization	Batch Normalization	8×8×256	Offset 1×1×256 Scale 1×1×256
11	relu_1 ReLU	ReLU	8×8×256	-
12	transposed-conv_2 128 5×5 transposed convolutions with stride [2 2] and cropping 'same'	Transposed Convolution	16×16×128	Weights 5×5×128×256 Bias 1×1×128
13	batchnorm_2 Batch normalization	Batch Normalization	16×16×128	Offset 1×1×128 Scale 1×1×128
14	relu_2 ReLU	ReLU	16×16×128	-
15	transposed-conv_3 64 5×5 transposed convolutions with stride [2 2] and cropping 'same'	Transposed Convolution	32×32×64	Weights 5×5×64×128 Bias 1×1×64
16	batchnorm_3 Batch normalization	Batch Normalization	32×32×64	Offset 1×1×64 Scale 1×1×64
17	relu_3 ReLU	ReLU	32×32×64	-
18	transposed-conv_4 3 5×5 transposed convolutions with stride [2 2] and cropping 'same'	Transposed Convolution	64×64×3	Weights 5×5×3×64 Bias 1×1×3
19	layer_2	Tanh	64×64×3	-

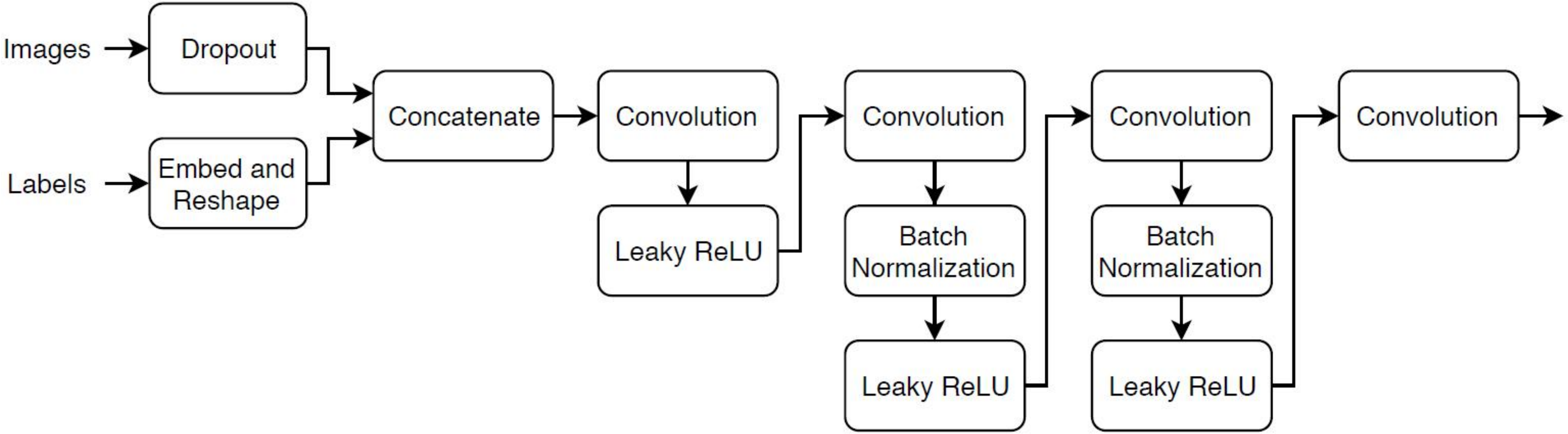
Define Generator Network

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a dlnetwork object.

```
dlnetGenerator = dlnetwork(lgraphGenerator)
dlnetGenerator =
  dlnetwork with properties:

        Layers:        [19×1 nnet.cnn.layer.Layer]
   Connections:        [18×2 table]
    Learnables:        [19×3 table]
         State:        [6×3 table]
    InputNames:        {'input'  'input_1'}
   OutputNames:        {'layer_2'}
   Initialized:        1
```

Define Discriminator Network

Define the following two-input network, which classifies real and generated 64-by-64 images given a set of images and the corresponding labels.



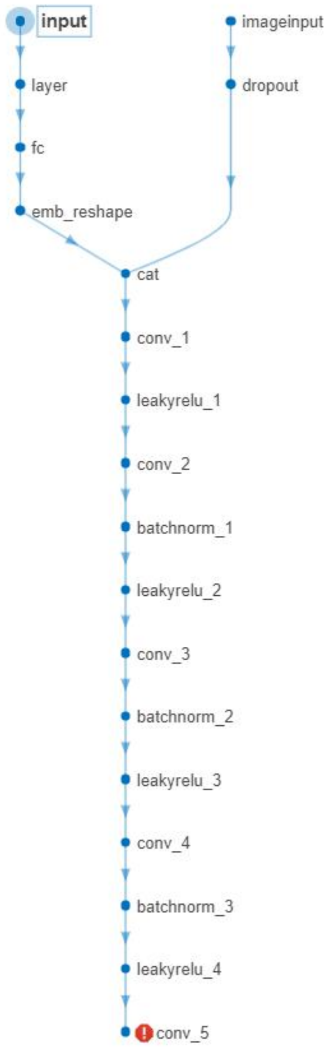
Define Discriminator Network

Create a network that takes as input 64-by-64-by-1 images and the corresponding labels and outputs a scalar prediction score using a series of convolution layers with batch normalization and leaky ReLU layers. Add noise to the input images using dropout.

- For the dropout layer, specify a dropout probability of 0.75.
- For the convolution layers, specify 5-by-5 filters with an increasing number of filters for each layer. Also specify a stride of 2 and padding of the output on each edge.
- For the leaky ReLU layers, specify a scale of 0.2.
- For the final layer, specify a convolution layer with one 4-by-4 filter.

```
dropoutProb = 0.75;
numFilters = 64;
scale = 0.2;

inputSize = [64 64 3];
filterSize = 5;

layersDiscriminator = [
    imageInputLayer(inputSize,Normalization="none")
    dropoutLayer(dropoutProb)
    concatenationLayer(3,2,Name="cat")
    convolution2dLayer(filterSize,numFilters,Stride=2,Padding="same")
    leakyReluLayer(scale)
    convolution2dLayer(filterSize,2*numFilters,Stride=2,Padding="same")
    batchNormalizationLayer
    leakyReluLayer(scale)
    convolution2dLayer(filterSize,4*numFilters,Stride=2,Padding="same")
    batchNormalizationLayer
    leakyReluLayer(scale)
    convolution2dLayer(filterSize,8*numFilters,Stride=2,Padding="same")
    batchNormalizationLayer
    leakyReluLayer(scale)
    convolution2dLayer(4,1)];
```

```
lgraphDiscriminator = layerGraph(layersDiscriminator);

layers = [
    featureInputLayer(1)
    embeddingLayer(embeddingDimension,numClasses)
    fullyConnectedLayer(prod(inputSize(1:2)))
    functionLayer(@(X) feature2image(X,
                                    [inputSize(1:2) ]),
                                    Formattable=true,
                                    Name="emb_reshape")];

lgraphDiscriminator = addLayers(lgraphDiscriminator,layers);
lgraphDiscriminator = connectLayers(lgraphDiscriminator,
                                        "emb_reshape","cat/in2");
```



	Name	Type	Activations	Learnables
1	input 1 features	Feature Input	1	-
2	layer Embedding layer with dimension 50	embeddingLayer	50	Weights 50×5
3	imageinput 64×64×3 images	Image Input	64×64×3	-
4	dropout 75% dropout	Dropout	64×64×3	-
5	fc 4096 fully connected layer	Fully Connected	4096	Weights 4096×50 Bias 4096×1
6	emb_reshape @(X)feature2image(X,[inputSize(1:2),1])	Function	64×64×1	-
7	cat Concatenation of 2 inputs along dimension 3	Concatenation	64×64×4	-
8	conv_1 64 5×5 convolutions with stride [2 2] and padding 'same'	Convolution	32×32×64	Weights 5×5×4×64 Bias 1×1×64
9	leakyrelu_1 Leaky ReLU with scale 0.2	Leaky ReLU	32×32×64	-
10	conv_2 128 5×5 convolutions with stride [2 2] and padding 'same'	Convolution	16×16×128	Weights 5×5×64×128 Bias 1×1×128
11	batchnorm_1 Batch normalization	Batch Normalization	16×16×128	Offset 1×1×128 Scale 1×1×128
12	leakyrelu_2 Leaky ReLU with scale 0.2	Leaky ReLU	16×16×128	-
13	conv_3 256 5×5 convolutions with stride [2 2] and padding 'same'	Convolution	8×8×256	Weights 5×5×128×256 Bias 1×1×256
14	batchnorm_2 Batch normalization	Batch Normalization	8×8×256	Offset 1×1×256 Scale 1×1×256
15	leakyrelu_3 Leaky ReLU with scale 0.2	Leaky ReLU	8×8×256	-
16	conv_4 512 5×5 convolutions with stride [2 2] and padding 'same'	Convolution	4×4×512	Weights 5×5×256×512 Bias 1×1×512
17	batchnorm_3 Batch normalization	Batch Normalization	4×4×512	Offset 1×1×512 Scale 1×1×512
18	leakyrelu_4 Leaky ReLU with scale 0.2	Leaky ReLU	4×4×512	-
19	conv_5	Convolution	1×1×1	Weights 4×4×512

Define Discriminator Network

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a dlnetwork object.

```
dlnetDiscriminator = dlnetwork(lgraphDiscriminator)
dlnetDiscriminator =
  dlnetwork with properties:

        Layers:      [19×1 nnet.cnn.layer.Layer]
   Connections:      [18×2 table]
    Learnables:      [19×3 table]
         State:      [6×3 table]
    InputNames:      {'imageinput'  'input'}
   OutputNames:      {'conv_5'}
   Initialized:      1
```

Define Model Gradients and Loss Functions

Create the function modelGradients, which takes as input the generator and discriminator networks, a mini-batch of input data, and an array of random values, and returns the gradients of the loss with respect to the learnable parameters in the networks and an array of generated images.

Specify Training Options

Train with a mini-batch size of 128 for 500 epochs.

numEpochs = 500;

miniBatchSize = 128;

Specify the options for Adam optimization. For both networks, use:

- A learning rate of 0.0002
- A gradient decay factor of 0.5
- A squared gradient decay factor of 0.999

learnRate = 0.0002;

gradientDecayFactor = 0.5;

squaredGradientDecayFactor = 0.999;

Specify Training Options

Update the training progress plots every 100 iterations.

validationFrequency = 100;

If the discriminator learns to discriminate between real and generated images too quickly, then the generator can fail to train. To better balance the learning of the discriminator and the generator, randomly flip the labels of a proportion of the real images. Specify a flip factor of 0.5.

flipFactor = 0.5;

Train Model

Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration. To monitor the training progress, display a batch of generated images using a held-out array of random values to input into the generator and the network scores.

Use minibatchqueue to process and manage the mini-batches of images during training. For each mini-batch:

- Use the custom mini-batch preprocessing function preprocessMiniBatch to rescale the images in the range [-1,1].
- Discard any partial mini-batches with less than 128 observations.
- Format the image data with the dimension labels "SSCB" (spatial, spatial, channel, batch).
- Format the label data with the dimension labels "BC" (batch, channel).
- Train on a GPU if one is available. When the OutputEnvironment option of minibatchqueue is "auto", minibatchqueue converts each output to a gpuArray if a GPU is available.

Train Model

The minibatchqueue object, by default, converts the data to dlarray objects with underlying type single.

```
augimds.MiniBatchSize = miniBatchSize;
executionEnvironment = "auto";

mbq = minibatchqueue(augimds, ...
    MiniBatchSize=miniBatchSize, ...
    PartialMiniBatch="discard", ...
    MiniBatchFcn=@preprocessData, ...
    MiniBatchFormat=["SSCB" "BC"], ...
    OutputEnvironment=executionEnvironment);
```

Train Model

Initialize the parameters for the Adam optimizer.

```
velocityDiscriminator = [];
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminator = [];
trailingAvgSqDiscriminator = [];
```

Train Model

Initialize the plot of the training progress. Create a figure and resize it to have twice the width.

```
f = figure;
f.Position(3) = 2*f.Position(3);
```

Create subplots of the generated images and of the scores plot.

```
imageAxes = subplot(1,2,1);
scoreAxes = subplot(1,2,2);
```

Train Model

Initialize animated lines for the scores plot.

```
lineScoreGenerator = animatedline(scoreAxes,Color=[0 0.447 0.741]);
lineScoreDiscriminator = animatedline(scoreAxes,Color=[0.85 0.325 0.098]);
```

Customize the appearance of the plots.

```
legend("Generator","Discriminator");
ylim([0 1])
xlabel("Iteration")
ylabel("Score")
grid on
```

Train Model

To monitor training progress, create a held-out batch of 25 random vectors and a corresponding set of labels 1 through 5 (corresponding to the classes) repeated five times.

```
numValidationImagesPerClass = 5;
ZValidation = randn(numLatentInputs,numValidationImagesPerClass*numClasses,"single");
TValidation = single(repmat(1:numClasses,[1 numValidationImagesPerClass]));
```

Convert the data to dlarray objects and specify the dimension labels "CB" (channel, batch).

```
dlZValidation = dlarray(ZValidation,"CB");
dlTValidation = dlarray(TValidation,"CB");
```

Train Model

Train the conditional GAN. For each epoch, shuffle the data and loop over mini-batches of data.

For each mini-batch:

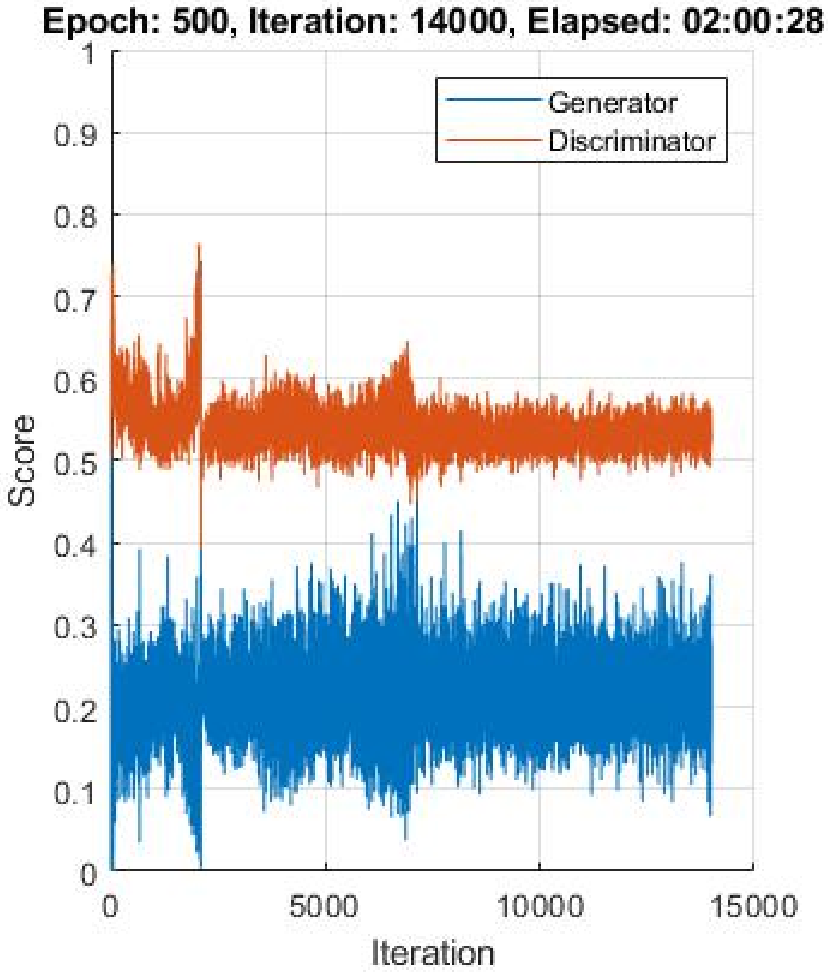
- Evaluate the model gradients using dlfeval and the modelGradients function.
- Update the network parameters using the adamupdate function.
- Plot the scores of the two networks.
- After every validationFrequency iterations, display a batch of generated images for a fixed held-out generator input.

Training can take some time to run.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Reset and shuffle data.
    shuffle(mbq);

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX,dlT] = next(mbq);
        Z = randn(numLatentInputs,miniBatchSize,"single");
        dlZ = dlarray(Z,"CB");

        [gradientsGenerator, gradientsDiscriminator, stateGenerator, scoreGenerator, scoreDiscriminator] = ...
            dlfeval(@modelGradients, dlnetGenerator, dlnetDiscriminator, dlX, dlT, dlZ, flipFactor);
        dlnetGenerator.State = stateGenerator;
```

```
[dlnetDiscriminator,trailingAvgDiscriminator,trailingAvgSqDiscriminator] = ...
    adamupdate(dlnetDiscriminator, gradientsDiscriminator, ...
    trailingAvgDiscriminator, trailingAvgSqDiscriminator, iteration, ...
    learnRate, gradientDecayFactor, squaredGradientDecayFactor);

[dlnetGenerator,trailingAvgGenerator,trailingAvgSqGenerator] = ...
    adamupdate(dlnetGenerator, gradientsGenerator, ...
    trailingAvgGenerator, trailingAvgSqGenerator, iteration, ...
    learnRate, gradientDecayFactor, squaredGradientDecayFactor);

% Every validationFrequency iterations, display batch of generated images.
if mod(iteration,validationFrequency) == 0 || iteration == 1
    dlXGeneratedValidation = predict(dlnetGenerator,dlZValidation,dlTValidation);
    I = imtile(extractdata(dlXGeneratedValidation), ...
        GridSize=[numValidationImagesPerClass numClasses]);
    I = rescale(I);
    % Display the images.
    subplot(1,2,1);
    image(imageAxes,I)
    xticklabels([]);
    yticklabels([]);
    title("Generated Images");
end

% Update the scores plot.
        subplot(1,2,2)
        addpoints(lineScoreGenerator,iteration,...
            double(gather(extractdata(scoreGenerator))));

        addpoints(lineScoreDiscriminator,iteration,...
            double(gather(extractdata(scoreDiscriminator))));

        % Update the title.
        D = duration(0,0,toc(start),Format="hh:mm:ss");
        title(...
            "Epoch: " + epoch + ", " + ...
            "Iteration: " + iteration + ", " + ...
            "Elapsed: " + string(D))

        drawnow
    end
end
```

Generated Images



Epoch: 500, Iteration: 14000, Elapsed: 02:00:28
Generator
Discriminator
Score
Iteration
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
5000
10000
15000

Generate New Images

To generate new images of a particular class, use the predict function on the generator with a dlarray object containing a batch of random vectors and an array of labels corresponding to the desired classes. Convert the data to dlarray objects and specify the dimension labels "CB" (channel, batch). For GPU prediction, convert the data to gpuArray objects. To display the images together, use the imtile function and rescale the images using the rescale function.

Create an array of 36 vectors of random values corresponding to the first class.

```
numObservationsNew = 36;
idxClass = 1;
Z = randn(numLatentInputs,numObservationsNew,"single");
T = repmat(single(idxClass),[1 numObservationsNew]);
```

Generate New Images

Convert the data to dlarray objects with the dimension labels "SSCB" (spatial, spatial, channels, batch).

```
dlZ = dlarray(Z,"CB");
dlT = dlarray(T,"CB");
```

To generate images using the GPU, also convert the data to gpuArray objects.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZ = gpuArray(dlZ);
    dlT = gpuArray(dlT);
end
```

Generate New Images

Generate images using the predict function with the generator network.

```
dlXGenerated = predict(dlnetGenerator,dlZ,dlT);
```

Display the generated images in a plot.

```
figure
I = imtile(extractdata(dlXGenerated));
I = rescale(I);
imshow(I)
title("Class: " + classes(idxClass))
```

Model Gradients Function

The function modelGradients takes as input the generator and discriminator dlnetwork objects dlnetGenerator and dlnetDiscriminator, a mini-batch of input data dlX, the corresponding labels dlT, and an array of random values dlZ, and returns the gradients of the loss with respect to the learnable parameters in the networks, the generator state, and the network scores.

If the discriminator learns to discriminate between real and generated images too quickly, then the generator can fail to train. To better balance the learning of the discriminator and the generator, randomly flip the labels of a proportion of the real images.

```
function [gradientsGenerator, gradientsDiscriminator, stateGenerator, scoreGenerator, scoreDiscriminator] = ...
    modelGradients(dlnetGenerator, dlnetDiscriminator, dlX, dlT, dlZ, flipFactor)

% Calculate the predictions for real data with the discriminator network.
dlYPred = forward(dlnetDiscriminator, dlX, dlT);

% Calculate the predictions for generated data with the discriminator network.
[dlXGenerated,stateGenerator] = forward(dlnetGenerator, dlZ, dlT);
dlYPredGenerated = forward(dlnetDiscriminator, dlXGenerated, dlT);

% Calculate probabilities.
probGenerated = sigmoid(dlYPredGenerated);
probReal = sigmoid(dlYPred);

% Calculate the generator and discriminator scores.
scoreGenerator = mean(probGenerated);
scoreDiscriminator = (mean(probReal) + mean(1-probGenerated)) / 2;

% Flip labels.
numObservations = size(dlYPred,4);
idx = randperm(numObservations,floor(flipFactor * numObservations));
probReal(:,:,:,idx) = 1 - probReal(:,:,:,idx);
```

```
% Calculate the GAN loss.
[lossGenerator, lossDiscriminator] = ganLoss(probReal, probGenerated);

% For each network, calculate the gradients with respect to the loss.
gradientsGenerator = dlgradient(lossGenerator, dlnetGenerator.Learnables,RetainData=true);
gradientsDiscriminator = dlgradient(lossDiscriminator, dlnetDiscriminator.Learnables);

end
```

GAN Loss Function

```
function [lossGenerator, lossDiscriminator] = ganLoss(scoresReal,scoresGenerated)

% Calculate losses for the discriminator network.
lossGenerated = -mean(log(1 - scoresGenerated));
lossReal = -mean(log(scoresReal));

% Combine the losses for the discriminator network.
lossDiscriminator = lossReal + lossGenerated;

% Calculate the loss for the generator network.
lossGenerator = -mean(log(scoresGenerated));

end
```

Mini-Batch Preprocessing Function

The preprocessMiniBatch function preprocesses the data using the following steps:

- Extract the image and label data from the input cell arrays and concatenate them into numeric arrays.
- Rescale the images to be in the range [-1,1].

```
function [X,T] = preprocessData(XCell,TCell)

% Extract image data from cell and concatenate
X = cat(4,XCell{:});

% Extract label data from cell and concatenate
T = cat(1,TCell{:});

% Rescale the images in the range [-1 1].
X = rescale(X,-1,1,InputMin=0,InputMax=255);
end
```

Wasserstein Generative Adversarial Network (wGAN)

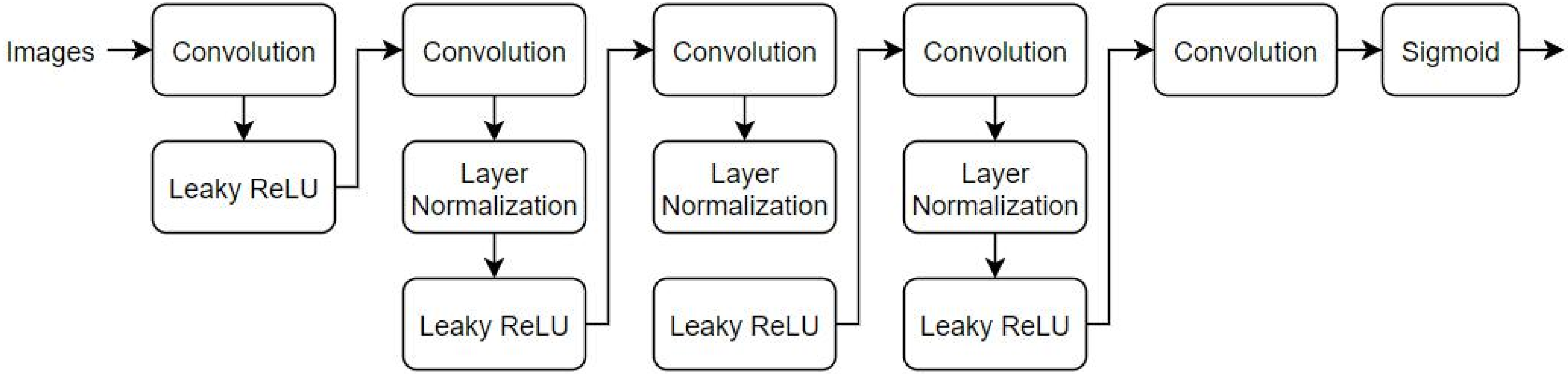
To train a GAN, train both networks simultaneously to :

- Train the generator to generate data that "fools" the discriminator.
- Train the discriminator to distinguish between real and generated data.

However, Arjovsky argues that the divergences which GANs typically minimize are potentially not continuous with respect to the generator's parameters, leading to training difficulty and introduces the Wasserstein GAN (WGAN) model that uses the Wasserstein loss to help stabilize training. A WGAN model can still produce poor samples or fail to converge because interactions between the weight constraint and the cost function can result in vanishing or exploding gradients. To address these issues, Gulrajani introduces a gradient penalty which improves stability by penalizing gradients with large norm values at the cost of longer computational time. This type of model is known as a WGAN-GP model.

Define Discriminator Network

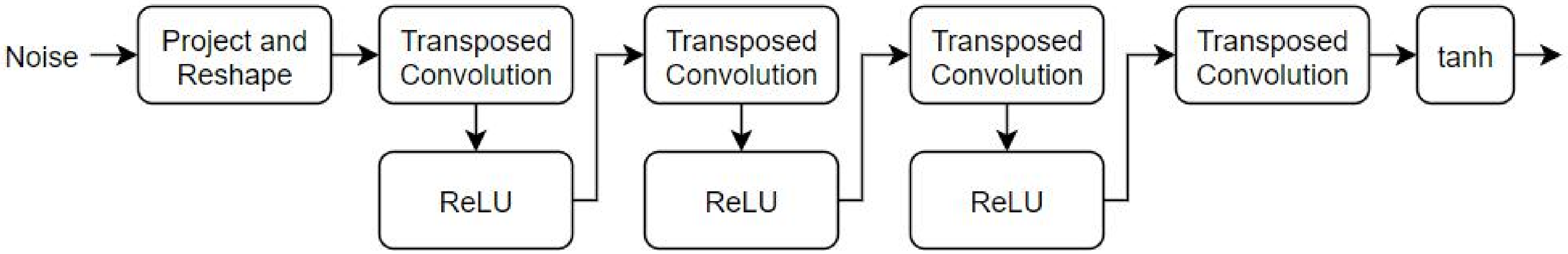
Define the following network, which classifies real and generated 64-by-64 images.



```
numFilters = 64;    scale = 0.2;
inputSize = [64 64 3];   filterSize = 5;

layersD = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
    convolution2dLayer(filterSize,numFilters,'Stride',2,'Padding','same','Name','conv1')
    leakyReluLayer(scale,'Name','lrelu1')
    convolution2dLayer(filterSize,2*numFilters,'Stride',2,'Padding','same','Name','conv2')
    layerNormalizationLayer('Name','bn2')
    leakyReluLayer(scale,'Name','lrelu2')
    convolution2dLayer(filterSize,4*numFilters,'Stride',2,'Padding','same','Name','conv3')
    layerNormalizationLayer('Name','bn3')
    leakyReluLayer(scale,'Name','lrelu3')
    convolution2dLayer(filterSize,8*numFilters,'Stride',2,'Padding','same','Name','conv4')
    layerNormalizationLayer('Name','bn4')
    leakyReluLayer(scale,'Name','lrelu4')
    convolution2dLayer(4,1,'Name','conv5')
    sigmoidLayer('Name','sigmoid')];
```

Define Generator Network

Define the following network architecture, which generates images from 1-by-1-by-100 arrays of random values:



Define Generator Network

This network:

- Converts the random vectors of size 100 to 7-by-7-by-128 arrays using a project and reshape layer.
- Upscales the resulting arrays to 64-by-64-by-3 arrays using a series of transposed convolution layers and ReLU layers.

Define this network architecture as a layer graph and specify the following network properties.

- For the transposed convolution layers, specify 5-by-5 filters with a decreasing number of filters for each layer, a stride of 2, and cropping of the output on each edge.
- For the final transposed convolution layer, specify three 5-by-5 filters corresponding to the three RGB channels of the generated images, and the output size of the previous layer.
- At the end of the network, include a tanh layer.

```
filterSize = 5;
numFilters = 64;
numLatentInputs = 100;

projectionSize = [4 4 512];

layersG = [
    featureInputLayer(numLatentInputs,'Normalization','none','Name','in')
    projectAndReshapeLayer(projectionSize,numLatentInputs,'Name','proj');
    transposedConv2dLayer(filterSize,4*numFilters,'Name','tconv1')
    reluLayer('Name','relu1')
    transposedConv2dLayer(filterSize,2*numFilters,'Stride',2,'Cropping','same','Name','tconv2')
    reluLayer('Name','relu2')
    transposedConv2dLayer(filterSize,numFilters,'Stride',2,'Cropping','same','Name','tconv3')
    reluLayer('Name','relu3')
    transposedConv2dLayer(filterSize,3,'Stride',2,'Cropping','same','Name','tconv4')
    tanhLayer('Name','tanh')];
```

Define Model Gradients Functions

Create the functions **modelGradientsD** and **modelGradientsG**, that calculate the gradients of the discriminator and generator loss with respect to the learnable parameters of the discriminator and generator networks, respectively.

The function **modelGradientsD** takes as input the generator and discriminator dlnetG and dlnetD, a mini-batch of input data dlX, an array of random values dlZ, and the lambda value used for the gradient penalty, and returns the gradients of the loss with respect to the learnable parameters in the discriminator, and the loss.

The function **modelGradientsG** takes as input the generator and discriminator dlnetG and dlnetD, and an array of random values dlZ, and returns the gradients of the loss with respect to the learnable parameters in the generator, and the loss.

Specify Training Options

To train a WGAN-GP model, you must train the discriminator for more iterations than the generator. In other words, for each generator iteration, you must train the discriminator for multiple iterations.

Train with a mini-batch size of 64 for 10,000 generator iterations.

miniBatchSize = 64;

numIterationsG = 10000;

For each generator iteration, train the discriminator for 5 iterations.

numIterationsDPerG = 5;

For the WGAN-GP loss, specify a lambda value of 10. The lambda value controls the magnitude of the gradient penalty added to the discriminator loss.

lambda = 10;

Specify Training Options

Specify the options for Adam optimization:

- For the discriminator network, specify a learning rate of 0.0002.
- For the generator network, specify a learning rate of 0.001.
- For both networks, specify a gradient decay factor of 0 and a squared gradient decay factor of 0.9.

```
learnRateD = 2e-4;
learnRateG = 1e-3;
gradientDecayFactor = 0;
squaredGradientDecayFactor = 0.9;
```

Display the generated validation images every 20 generator iterations.

```
validationFrequency = 20;
```

Train Model

Train the WGAN-GP model by looping over mini-batches of data.

For numIterationsDPerG iterations, train the discriminator only. For each mini-batch:

- Evaluate the discriminator model gradients using dlfeval and the modelGradientsD function.
- Update the discriminator network parameters using the adamupdate function.

After training the discriminator for numIterationsDPerG iterations, train the generator on a single mini-batch.

- Evaluate the generator model gradients using dlfeval and the modelGradientsG function.
- Update the generator network parameters using the adamupdate function.

After updating the generator network:

- Plot the losses of the two networks.
- After every validationFrequency generator iterations, display a batch of generated images for a fixed held-out generator input.

After passing through the data set, shuffle the mini-batch queue.

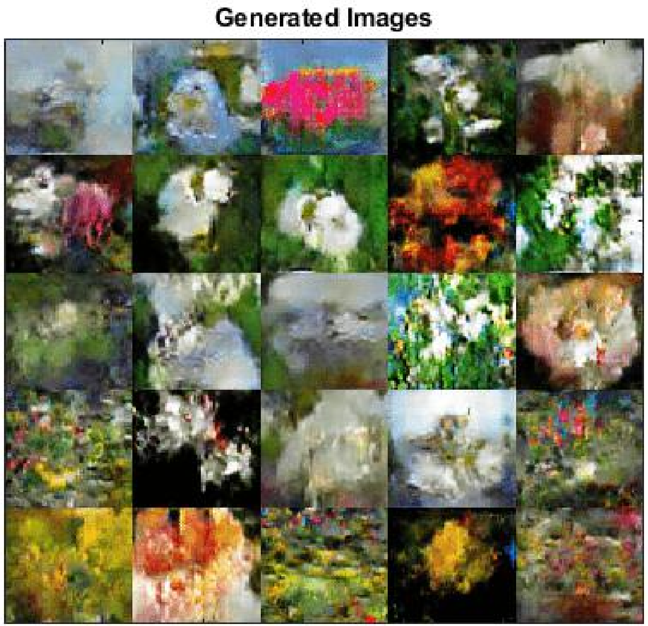
Training can take some time to run and may require many iterations to output good images.

```
iterationG = 0;
iterationD = 0;
start = tic;

% Loop over mini-batches
while iterationG < numIterationsG
   iterationG = iterationG + 1;

   % Train discriminator only
   for n = 1:numIterationsDPerG
      iterationD = iterationD + 1;

      % Reset and shuffle mini-batch queue when there is no more data.
      if ~hasdata(mbq)
         shuffle(mbq);
      end

      % Read mini-batch of data.
      dlX = next(mbq);

      % Generate latent inputs for the generator network. Convert to
      % dlarray and specify the dimension labels 'CB' (channel, batch).
      Z = randn([numLatentInputs size(dlX,4)],'like',dlX);
      dlZ = dlarray(Z,'CB');
```

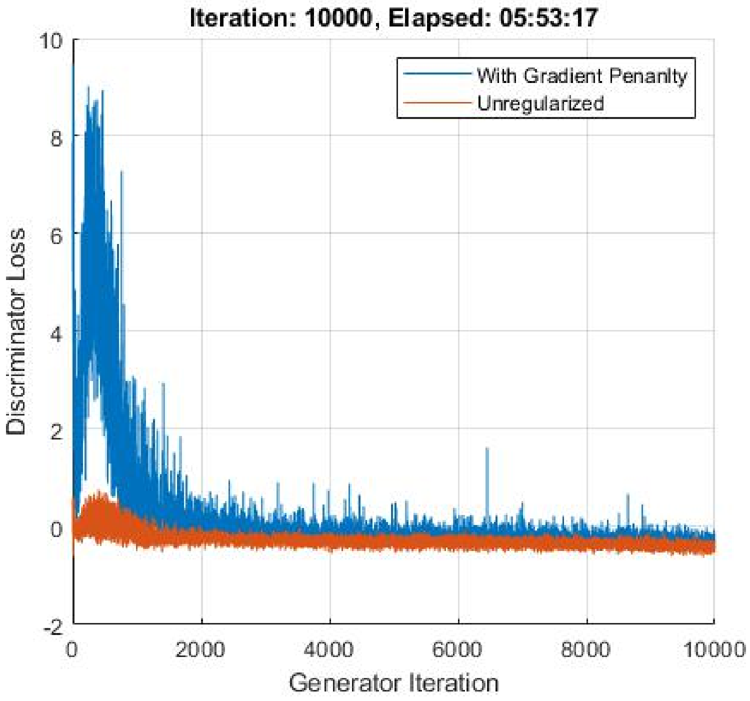
```
% Evaluate the discriminator model gradients using dlfeval and the
    % modelGradientsD function listed at the end of the example.
    [gradientsD, lossD, lossDUnregularized] = dlfeval(@modelGradientsD, dlnetD, dlnetG, dlX, dlZ, lambda);

    % Update the discriminator network parameters.
    [dlnetD,trailingAvgD,trailingAvgSqD] = adamupdate(dlnetD, gradientsD, ...
      trailingAvgD, trailingAvgSqD, iterationD, ...
      learnRateD, gradientDecayFactor, squaredGradientDecayFactor);
  end

  % Generate latent inputs for the generator network. Convert to dlarray
  % and specify the dimension labels 'CB' (channel, batch).
  Z = randn([numLatentInputs size(dlX,4)],'like',dlX);
  dlZ = dlarray(Z,'CB');

  % Evaluate the generator model gradients using dlfeval and the
  % modelGradientsG function listed at the end of the example.
  gradientsG = dlfeval(@modelGradientsG, dlnetG, dlnetD, dlZ);

  % Update the generator network parameters.
  [dlnetG,trailingAvgG,trailingAvgSqG] = adamupdate(dlnetG, gradientsG, ...
    trailingAvgG, trailingAvgSqG, iterationG, ...
    learnRateG, gradientDecayFactor, squaredGradientDecayFactor);
```

```
% Every validationFrequency generator iterations, display batch of generated
if mod(iterationG,validationFrequency) == 0 || iterationG == 1
    % Generate images using the held-out generator input.
    dlXGeneratedValidation = predict(dlnetG,dlZValidation);

    % Tile and rescale the images in the range [0 1].
    I = imtile(extractdata(dlXGeneratedValidation));
    I = rescale(I);

    % Display the images.
    subplot(1,2,1);
    image(imageAxes,I)
    xticklabels([]);
    yticklabels([]);
    title("Generated Images");
end
```

```
% Update the scores plot
    subplot(1,2,2)

    lossD = double(gather(extractdata(lossD)));
    lossDUnregularized = double(gather(extractdata(lossDUnregularized)));
    addpoints(lineLossD,iterationG,lossD);
    addpoints(lineLossDUnregularized,iterationG,lossDUnregularized);

    D = duration(0,0,toc(start),'Format','hh:mm:ss');
    title( ...
        "Iteration: " + iterationG + ", " + ...
        "Elapsed: " + string(D))
    drawnow
end
```



Generated Images



Iteration: 10000, Elapsed: 05:53:17
With Gradient Penanlty
Unregularized
Discriminator Loss
Generator Iteration
10
8
6
4
2
0
-2
0
2000
4000
6000
8000
10000

Discriminator Model Gradients Function

The function modelGradientsD takes as input the generator and discriminator dlnetwork objects dlnetG and dlnetD, a mini-batch of input data dlX, an array of random values dlZ, and the lambda value used for the gradient penalty, and returns the gradients of the loss with respect to the learnable parameters in the discriminator, and the loss.

Given an image X, a generated image $\widetilde{X}$, define $\widehat{X} = \epsilon X + (1-\epsilon)\widetilde{X}$ for some random $\epsilon \in U(0,1)$

For the WGAN-GP model, given the lambda value λ, the discriminator loss is given by

$$\mathrm{loss}_D = \widetilde{Y} - Y + \lambda\left(\left\|\nabla_{\widehat{X}}\widehat{Y}\right\|_2 - 1\right)^2,$$

```
function [gradientsD, lossD, lossDUnregularized] = modelGradientsD(dlnetD, dlnetG, dlX, dlZ, lambda)

% Calculate the predictions for real data with the discriminator network.
dlYPred = forward(dlnetD, dlX);

% Calculate the predictions for generated data with the discriminator
% network.
dlXGenerated = forward(dlnetG,dlZ);
dlYPredGenerated = forward(dlnetD, dlXGenerated);

% Calculate the loss.
lossDUnregularized = mean(dlYPredGenerated - dlYPred);

% Calculate and add the gradient penalty.
epsilon = rand([1 1 1 size(dlX,4)],'like',dlX);
dlXHat = epsilon.*dlX + (1-epsilon).*dlXGenerated;
dlYHat = forward(dlnetD, dlXHat);

% Calculate gradients. To enable computing higher-order derivatives, set
% 'EnableHigherDerivatives' to true.
gradientsHat = dlgradient(sum(dlYHat),dlXHat,'EnableHigherDerivatives',true);
gradientsHatNorm = sqrt(sum(gradientsHat.^2,1:3) + 1e-10);
gradientPenalty = lambda.*mean((gradientsHatNorm - 1).^2);

% Penalize loss.
lossD = lossDUnregularized + gradientPenalty;

% Calculate the gradients of the penalized loss
%  with respect to the learnable parameters.
gradientsD = dlgradient(lossD, dlnetD.Learnables);

end
```

Generator Model Gradients Function

The function modelGradientsG takes as input the generator and discriminator dlnetwork objects dlnetG and dlnetD, and an array of random values dlZ, and returns the gradients of the loss with respect to the learnable parameters in the generator.

Given a generated image $\widetilde{X}$, the generator loss is given by

$$\mathrm{loss}_G = -\widetilde{Y},$$

```
function gradientsG =  modelGradientsG(dlnetG, dlnetD, dlZ)

% Calculate the predictions for generated data with the discriminator
% network.
dlXGenerated = forward(dlnetG,dlZ);
dlYPredGenerated = forward(dlnetD, dlXGenerated);

% Calculate the loss.
lossG = -mean(dlYPredGenerated);

% Calculate the gradients of the loss with respect to the learnable
% parameters.
gradientsG = dlgradient(lossG, dlnetG.Learnables);

end
```

Mini-Batch Preprocessing Function

The preprocessMiniBatch function preprocesses the data using the following steps:

- Extract the image data from the input cell array and concatenate into a numeric array.
- Rescale the images to be in the range [-1,1].

```
function X = preprocessMiniBatch(data)

% Concatenate mini-batch
X = cat(4,data{:});

% Rescale the images in the range [-1 1].
X = rescale(X,-1,1,'InputMin',0,'InputMax',255);

end
```

Generate New Images

- To generate new images, use the predict function on the generator with a dlarray object containing a batch of random vectors. To display the images together, use the imtile function and rescale the images using the rescale function.
- Create a dlarray object containing a batch of 25 random vectors to input to the generator network.

```
ZNew = randn(numLatentInputs,25,'single');
dlZNew = dlarray(ZNew,'CB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment ==
"gpu"
    dlZNew = gpuArray(dlZNew);
end
```

Generate New Images

- Generate new images using the predict function with the generator and the input data.

```
dlXGeneratedNew = predict(dlnetG,dlZNew);
```

- Display the images.

```
I = imtile(extractdata(dlXGeneratedNew));
I = rescale(I);
figure
image(I)
axis off
title("Generated Images")
```