Deep Learning Models - GAN

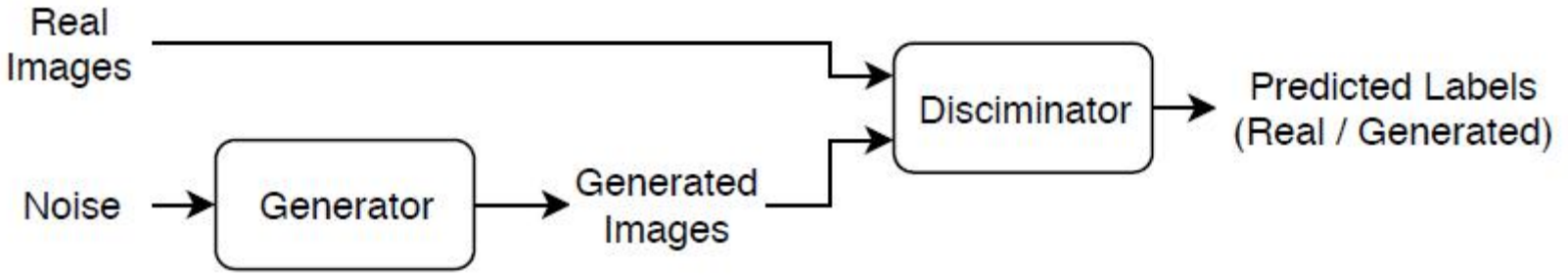
- generative adversarial network (GAN) to generate images and sounds

Outline

- About Generative Adversarial Network (GAN)
- Train Generative Adversarial Network (GAN) for Image Generation
- Train Generative Adversarial Network (GAN) for Sound Synthesis

About 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".



Train Generative Adversarial Network (GAN)

- To train a 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 optimize the performance of the generator, maximize the loss of the discriminator when given generated data. That is, the objective of the generator is to generate data that the discriminator classifies as "real".

Train Generative Adversarial Network (GAN)

- To optimize the performance of the discriminator, minimize the loss of the discriminator when given batches of both real and generated 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 and a discriminator that has learned strong feature representations that are characteristic of the training data.

Train Generative Adversarial Network (GAN) for Image Generation

- 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
```

Load Training Data

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

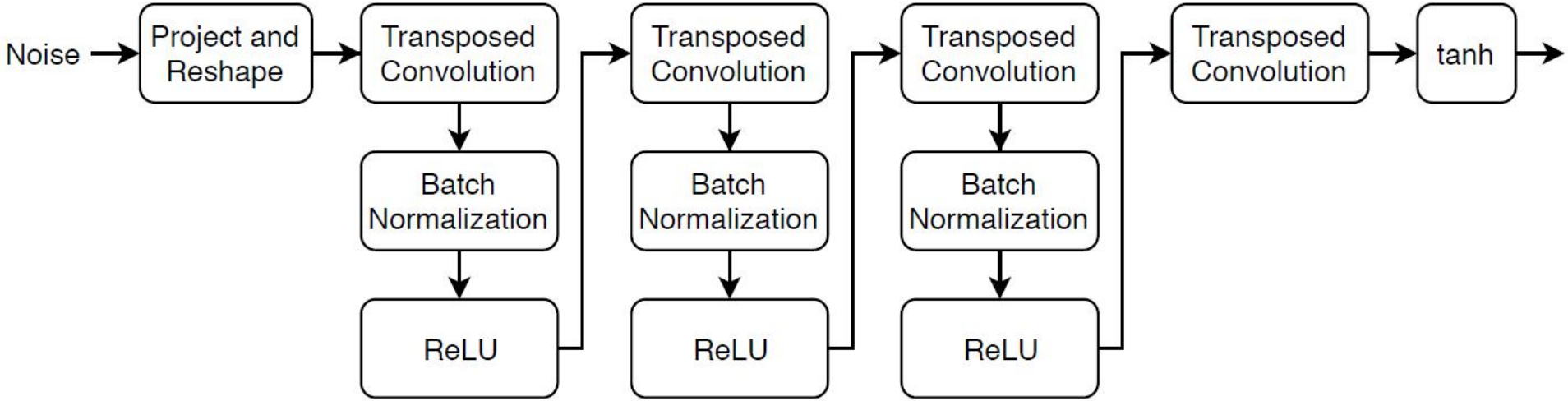
```
datasetFolder = fullfile(imageFolder);
imds = imageDatastore(datasetFolder, 'IncludeSubfolders',true);
```

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



Define Generator Network

- This network: Converts the 1-by-1-by-100 arrays of noise to 4-by-4-by-512 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 with batch normalization 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.

Define Generator Network

- To project and reshape the noise input, use the custom layer projectAndReshapeLayer, attached to this example as a supporting file. The projectAndReshapeLayer layer upscales the input using a fully connected operation and reshapes the output to the specified size.

```
filterSize = 5;
numFilters = 64;
numLatentInputs = 100;
projectionSize = [4 4 512];
layersGenerator = [
    imageInputLayer([1 1 numLatentInputs],'Normalization','none','Name','in')
    projectAndReshapeLayer(projectionSize,numLatentInputs,'proj'); % [4, 4, 512]
    transposedConv2dLayer(filterSize,4*numFilters,'Name','tconv1')
    batchNormalizationLayer('Name','bnorm1')
    reluLayer('Name','relu1') % [8, 8, 256]
    transposedConv2dLayer(filterSize,2*numFilters,'Stride',2,'Cropping','same','Name','tconv2')
    batchNormalizationLayer('Name','bnorm2')
    reluLayer('Name','relu2') % [16, 16, 128]
    transposedConv2dLayer(filterSize,numFilters,'Stride',2,'Cropping','same','Name','tconv3')
    batchNormalizationLayer('Name','bnorm3')
    reluLayer('Name','relu3') % [32, 32, 64]
    transposedConv2dLayer(filterSize,3,'Stride',2,'Cropping','same','Name','tconv4')
    tanhLayer('Name','tanh')]; % [64, 64, 3]
lgraphGenerator = layerGraph(layersGenerator);
```

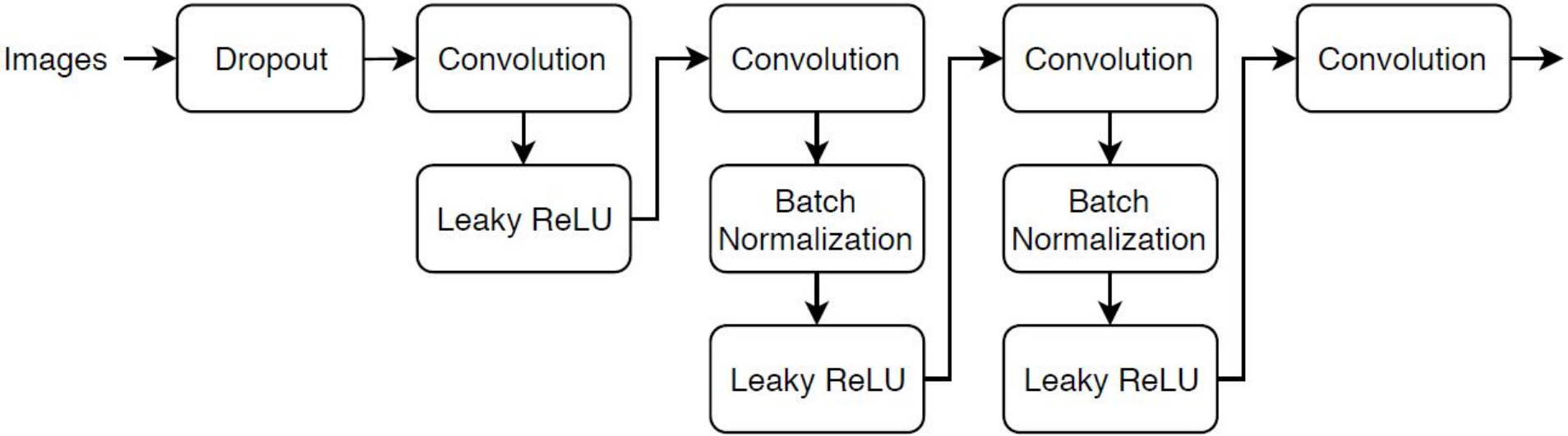
Define Generator Network

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

```
dlnetGenerator = dlnetwork(lgraphGenerator);
```

Define Discriminator Network

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



Define Discriminator Network

- Create a network that takes 64-by-64-by-3 images and returns 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 0.5.
  - 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 a padding of the output.
  - For the leaky ReLU layers, specify a scale of 0.2.
  - For the final layer, specify a convolutional layer with one 4-by-4 filter.

```
dropoutProb = 0.5;
numFilters = 64;     scale = 0.2;
inputSize = [64 64 3];     filterSize = 5;
layersDiscriminator = [
   imageInputLayer(inputSize,'Normalization','none','Name','in')
   dropoutLayer(0.5,'Name','dropout')
   convolution2dLayer(filterSize,numFilters,'Stride',2,'Padding','same','Name','conv1') % [32, 32, 64]
   leakyReluLayer(scale,'Name','lrelu1')
   convolution2dLayer(filterSize,2*numFilters,'Stride',2,'Padding','same','Name','conv2') % [16, 16, 128]
   batchNormalizationLayer('Name','bn2')
   leakyReluLayer(scale,'Name','lrelu2')
   convolution2dLayer(filterSize,4*numFilters,'Stride',2,'Padding','same','Name','conv3') % [8, 8, 256]
   batchNormalizationLayer('Name','bn3')
   leakyReluLayer(scale,'Name','lrelu3')
   convolution2dLayer(filterSize,8*numFilters,'Stride',2,'Padding','same','Name','conv4') % [4, 4, 512]
   batchNormalizationLayer('Name','bn4')
   leakyReluLayer(scale,'Name','lrelu4')
   convolution2dLayer(4,1,'Name','conv5')]; % [1, 1, 1]
lgraphDiscriminator = layerGraph(layersDiscriminator);
```

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);
```

Define Model Gradients, Loss Functions and Scores

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

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, an array of random values dlZ and the percentage of real labels to flip flipFactor, and returns the gradients of the loss with respect to the learnable parameters in the networks, the generator state, and the scores of the two networks. Because the discriminator output is not in the range [0,1], modelGradients applies the sigmoid function to convert it into probabilities.

```
function [gradientsGenerator, gradientsDiscriminator, stateGenerator, scoreGenerator,
scoreDiscriminator] = ...
    modelGradients(dlnetGenerator, dlnetDiscriminator, dlX, dlZ, flipFactor)
% Calculate the predictions for real data with the discriminator network.
dlYPred = forward(dlnetDiscriminator, dlX);
% Calculate the predictions for generated data with the discriminator network.
[dlXGenerated,stateGenerator] = forward(dlnetGenerator,dlZ);
dlYPredGenerated = forward(dlnetDiscriminator, dlXGenerated);
% Convert the discriminator outputs to probabilities.
probGenerated = sigmoid(dlYPredGenerated);
probReal = sigmoid(dlYPred);
% Calculate the score of the discriminator.
scoreDiscriminator = ((mean(probReal)+mean(1-probGenerated))/2);
% Calculate the score of the generator.
scoreGenerator = mean(probGenerated);
```

```
% Randomly flip a fraction of the labels of the real images.
numObservations = size(probReal,4);
idx = randperm(numObservations,floor(flipFactor * numObservations));
% Flip the labels
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
```

Specify Training Options

- Train with a mini-batch size of 128 for 500 epochs. For larger datasets, you might not need to train for as many epochs.

```
numEpochs = 500;
miniBatchSize = 128;
```

- Specify the options for Adam optimization. For both networks, specify
  - 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

- If the discriminator learns to discriminate between real and generated images too quickly, then the generator may fail to train. To better balance the learning of the discriminator and the generator, add noise to the real data by randomly flipping the labels.
- Specify to flip 30% of the real labels. This means that 15% of the total number of labels are flipped during training. Note that this does not impair the generator as all the generated images are still labelled correctly.

```
flipFactor = 0.3;
```

- Display the generated validation images every 100 iterations.

```
validationFrequency = 100;
```

Train Model

- Use minibatchqueue to process and manage the mini-batches of images. 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). By default, the minibatchqueue object converts the data to dlarray objects with underlying type single.

Train Model

```
augimds.MiniBatchSize = miniBatchSize;

executionEnvironment = "auto";

mbq = minibatchqueue(augimds,...
    'MiniBatchSize',miniBatchSize,...
    'PartialMiniBatch','discard',...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat','SSCB',...
    'OutputEnvironment',executionEnvironment);
```

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 as well as a plot of the scores.
- Initialize the parameters for Adam.

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

Train Model

- To monitor training progress, display a batch of generated images using a held-out batch of fixed arrays of random values fed into the generator and plot the network scores.
- Create an array of held-out random values.

```
numValidationImages = 25;
ZValidation = randn(1,1,numLatentInputs,numValidationImages,'single');
```

- Convert the data to dlarray objects and specify the dimension labels 'SSCB' (spatial, spatial, channel, batch).

```
dlZValidation = dlarray(ZValidation,'SSCB');
```

Train Model

- For GPU training, convert the data to gpuArray objects.

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

- Initialize the training progress plots. Create a figure and resize it to have twice the width.

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

- Create a subplot for the generated images and the network scores.

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

Train Model

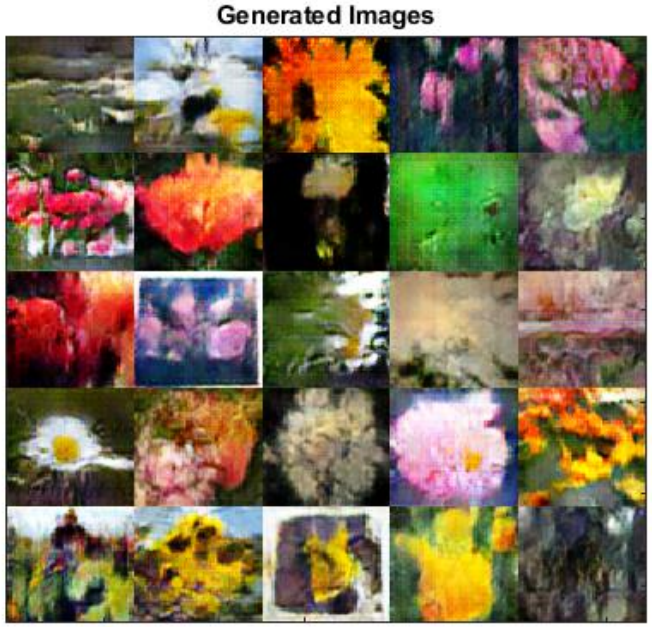
- Initialize the animated lines for the scores plot.

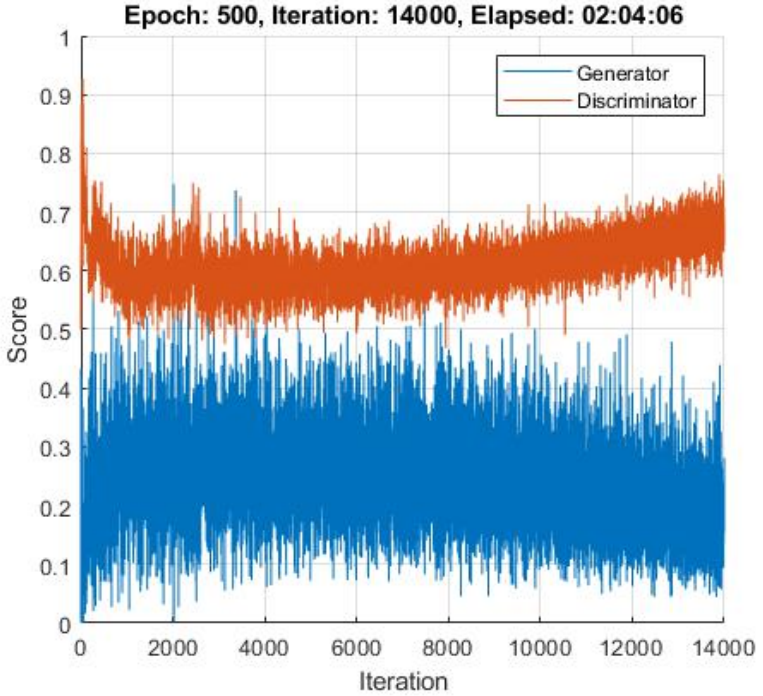
```
lineScoreGenerator = animatedline(scoreAxes,'Color',[0 0.447 0.741]);
lineScoreDiscriminator = animatedline(scoreAxes, 'Color', [0.85 0.325 0.098]);
legend('Generator','Discriminator');
ylim([0 1])
xlabel("Iteration")
ylabel("Score")
grid on
```

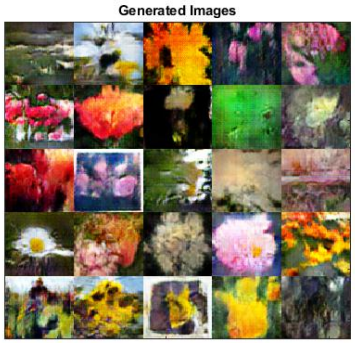
Train Model

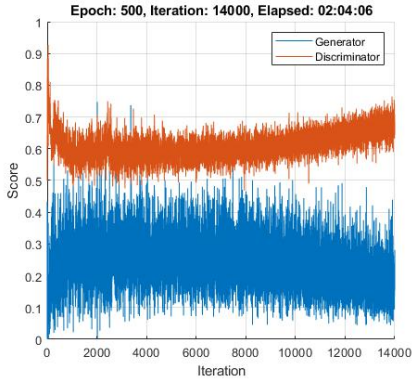
- Train the GAN. For each epoch, shuffle the datastore and loop over mini-batches of data.
- For each mini-batch:
  - Evaluate the model gradients using dlfeval and the modelGradients function.
  - Update the network parameters using the adamupdate function.
  - Plot the scores of the two networks.
  - After every validationFrequency iterations, display a batch of generated images for a fixed held-out generator input.
- Training can take some time to run.

Train Model


Generated Images


Epoch: 500, Iteration: 14000, Elapsed: 02:04:06
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
2000
4000
6000
8000
10000
12000
14000





- Here, the discriminator has learned a strong feature representation that identifies real images among generated images. In turn, the generator has learned a similarly strong feature representation that allows it to generate realistic looking data.
- The training plot shows the scores of the generator and discriminator networks.

Generate New Images

- To generate new images, use the predict function on the generator with a dlarray object containing a batch of 1-by-1-by-100 arrays of random values. 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 1-by-1-by-100 arrays of random values to input into the generator network.

```
ZNew = randn(1,1,numLatentInputs,25,'single');
dlZNew = dlarray(ZNew,'SSCB');
```

Generate New Images

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

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

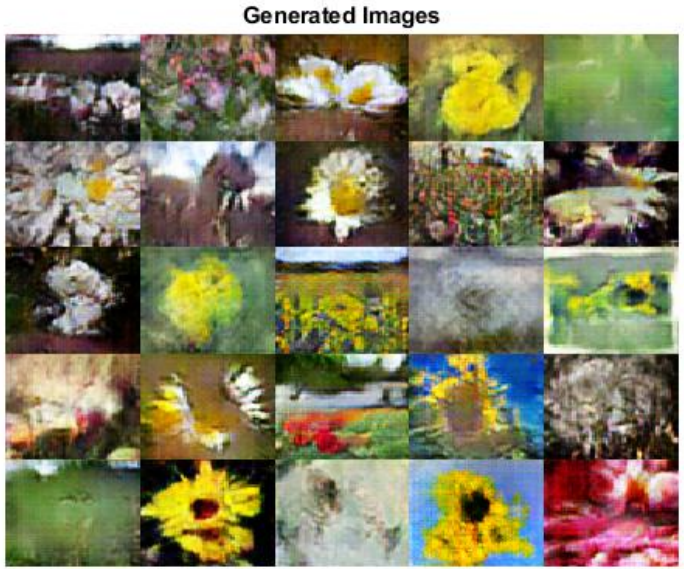
- Generate new images using the predict function with the generator and the input data.

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

Generate New Images

- Display the images.

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



Train Generative Adversarial Network (GAN) for Sound Synthesis

- This example shows how to train and use a generative adversarial network (GAN) to generate sounds.

Introduction

- In generative adversarial networks, a generator and a discriminator compete against each other to improve the generation quality.
- GANs have generated significant interest in the field of audio and speech processing. Applications include text-to-speech synthesis, voice conversion, and speech enhancement.
- This example trains a GAN for unsupervised synthesis of audio waveforms. The GAN in this example generates drumbeat sounds. The same approach can be followed to generate other types of sound, including speech.

Synthesize Audio with Pre-Trained GAN

- Before you train a GAN from scratch, you will use a pretrained GAN generator to synthesize drum beats. Download the pretrained generator.

```
matFileName = 'drumGeneratorWeights.mat';
if ~exist(matFileName,'file')
websave(matFileName,'https://www.mathworks.com/supportfiles/audio/GanAudioSynthesis/drumGeneratorWeights.mat');
end
```

Synthesize Audio with Pre-Trained GAN

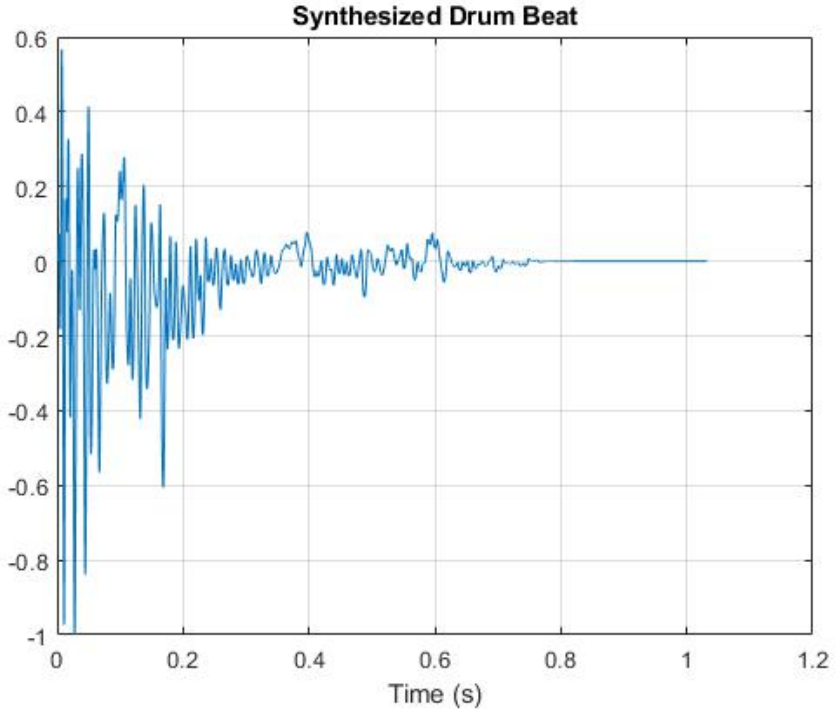
- The function synthesizeDrumBeat calls a pretrained network to synthesize a drumbeat sampled at 16 kHz. Synthesize a drumbeat and listen to it.

```
drum = synthesizeDrumBeat;
fs = 16e3;
sound(drum,fs)
```

Synthesize Audio with Pre-Trained GAN

- Plot the synthesized drumbeat.

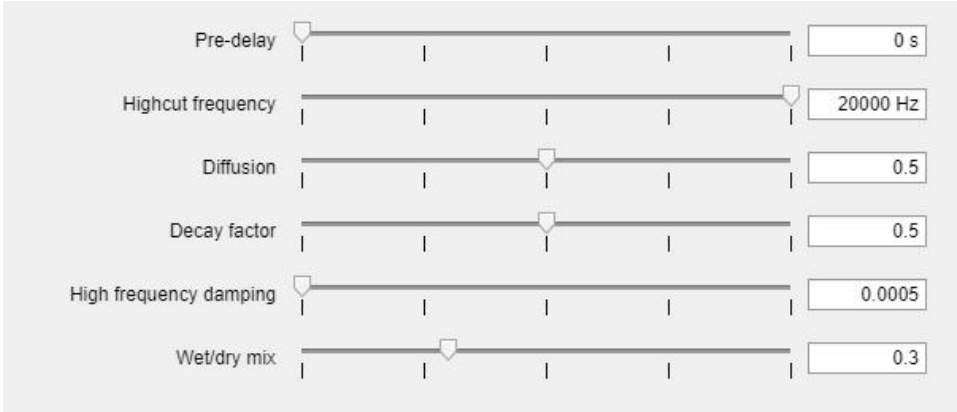
```
t = (0:length(drum)-1)/fs;
plot(t,drum)
grid on
xlabel('Time (s)')
title('Synthesized Drum Beat')
```



Synthesize Audio with Pre-Trained GAN

- You can use the drumbeat synthesizer with other audio effects to create more complex applications. For example, you can apply reverberation to the synthesized drum beats.
- Create a reverberator object and open its parameter tuner UI. This UI enables you to tune the reverberator parameters as the simulation runs.

```
reverb = reverberator('SampleRate',fs);
parameterTuner(reverb);
```



Synthesize Audio with Pre-Trained GAN

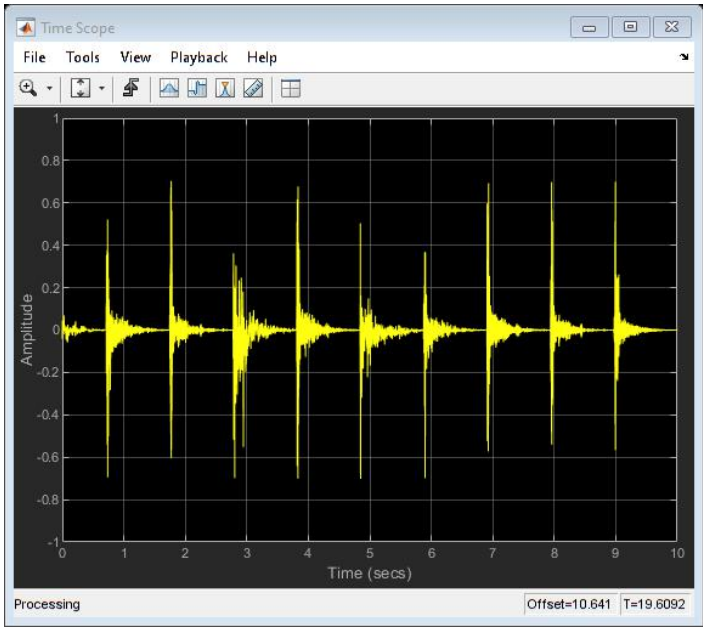
- Create a dsp.TimeScope object to visualize the drum beats.

```
ts = dsp.TimeScope('SampleRate',fs, ...
                   'TimeSpanOverrunAction','Scroll', ...
                   'TimeSpan',10, ...
                   'BufferLength',10*256*64, ...
                   'ShowGrid',true, ...
                   'YLimits',[-1 1]);
```

Synthesize Audio with Pre-Trained GAN

- In a loop, synthesize the drum beats and apply reverberation. Use the parameter tuner UI to tune reverberation. If you want to run the simulation for a longer time, increase the value of the loopCount parameter.

```
loopCount = 20;
for ii = 1:loopCount
    drum = synthesizeDrumBeat;
    drum = reverb(drum);
    ts(drum(:,1));
    soundsc(drum,fs)
    pause(0.5)
end
```



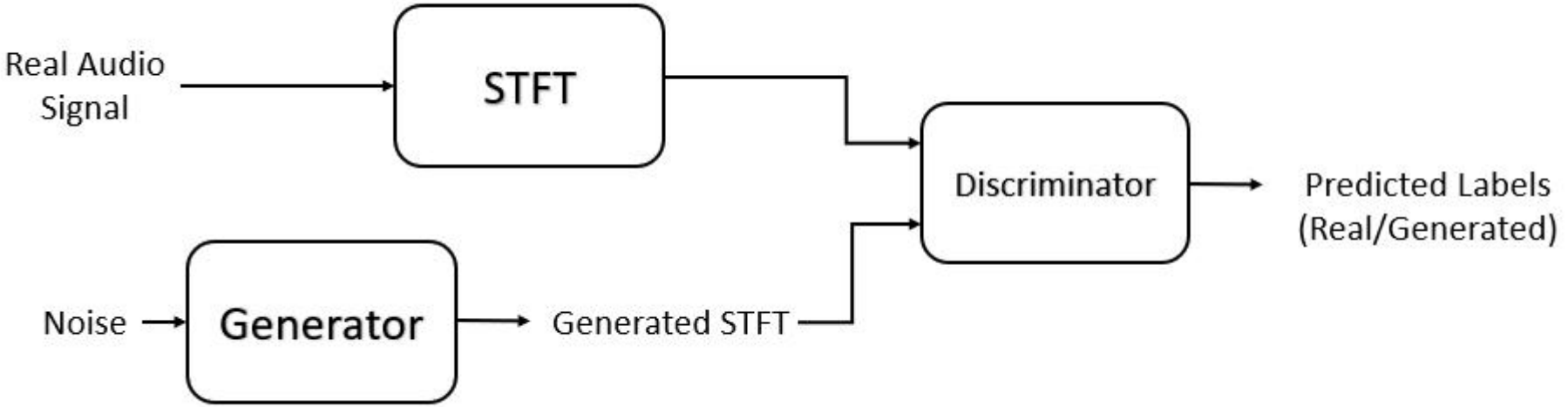
Train the GAN

- Now that you have seen the pretrained drumbeat generator in action, you can investigate the training process in detail.
- A GAN is a type of deep learning network that generates data with characteristics similar to the training data.
- A GAN consists of two networks that train together, a generator and a discriminator:
  - Generator - Given a vector or random values as input, this network generates data with the same structure as the training data. It is the generator's job to fool the discriminator.
  - Discriminator - Given batches of data containing observations from both the training data and the generated data, this network attempts to classify the observations as real or generated.

Train the GAN

- To maximize the performance of the generator, maximize the loss of the discriminator when given generated data. That is, the objective of the generator is to generate 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 data. Ideally, these strategies result in a generator that generates convincingly realistic data and a discriminator that has learned strong feature representations that are characteristic of the training data.
- In this example, you train the generator to create fake time-frequency short-time Fourier transform (STFT) representations of drum beats. You train the discriminator to identify real STFTs. You create the real STFTs by computing the STFT of short recordings of real drum beats.

Train the GAN



Load Training Data

- Train a GAN using the Drum Sound Effects dataset. Download and extract the dataset.

```
url = 'http://deepyeti.ucsd.edu/cdonahue/wavegan/data/drums.tar.gz';
downloadFolder = tempdir;
filename = fullfile(downloadFolder,'drums_dataset.tgz');
drumsFolder = fullfile(downloadFolder,'drums');
if ~exist(drumsFolder,'dir')
    disp('Downloading Drum Sound Effects Dataset (218 MB)...')
    websave(filename,url);
    untar(filename,downloadFolder)
end
```

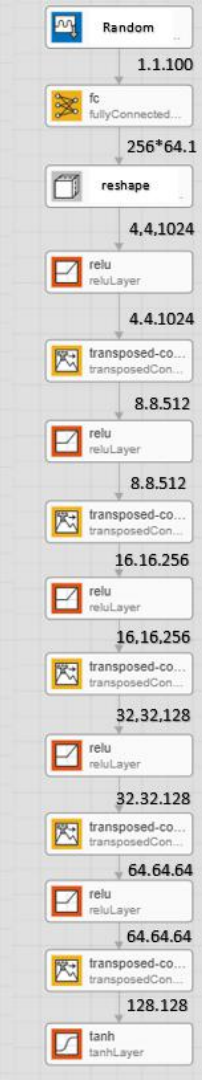
Load Training Data

- Create an audioDatastore object that points to the drums dataset.

```
ads = audioDatastore(drumsFolder,'IncludeSubfolders',true);
```

Define Generator Network

- Define a network that generates STFTs from 1-by-1-by-100 arrays of random values. Create a network that upscales 1-by-1-by-100 arrays to 128-by-128-by-1 arrays using a fully connected layer followed by a reshape layer and a series of transposed convolution layers with ReLU layers.
- This figure shows the dimensions of the signal as it travels through the generator.



Define Discriminator Network

- Define a network that classifies real and generated 128-by-128 STFTs.
- Create a network that takes 128-by-128 images and outputs a scalar prediction score using a series of convolution layers with leaky ReLU layers followed by a fully connected layer.
- This figure shows the dimensions of the signal as it travels through the discriminator.

Generate Real Drumbeat Training Data

- Generate STFT data from the drumbeat signals in the datastore.
- Define the STFT parameters.

```
fftLength = 256;
win = hann(fftLength,'periodic');
overlapLength = 128;
```

Generate Real Drumbeat Training Data

- To speed up processing, distribute the feature extraction across multiple workers using parfor.
- First, determine the number of partitions for the dataset. If you do not have Parallel Computing Toolbox™, use a single partition.

```
if ~isempty(ver('parallel'))
    pool = gcp;
    numPar = numpartitions(ads,pool);
else
    numPar = 1;
end
```

Generate Real Drumbeat Training Data

```
parfor ii = 1:numPar
    subds = partition(ads,numPar,ii);
    STrain = zeros(fftLength/2+1,128,1,numel(subds.Files));
    for idx = 1:numel(subds.Files)
        x = read(subds);
        if length(x) > fftLength*64
            % Lengthen the signal if it is too short
            x = x(1:fftLength*64);
        end
        % Convert from double-precision to single-precision
        x = single(x);
```

Generate Real Drumbeat Training Data

```
    % Scale the signal
    x = x ./ max(abs(x));
    % Zero-pad to ensure stft returns 128 windows.
    x = [x ; zeros(overlapLength,1,'like',x)];
    S0 = stft(x,'Window',win,'OverlapLength',overlapLength,'Centered',false);
    % Convert from two-sided to one-sided.
    S = S0(1:129,:);
    S = abs(S);
    STrain(:,:,:,idx) = S;
  end
  STrainC{ii} = STrain;
end
```

Generate Real Drumbeat Training Data

- Convert the output to a four-dimensional array with STFTs along the fourth dimension.

```
STrain = cat(4,STrainC{:});
```

- Convert the data to the log scale to better align with human perception.

```
STrain = log(STrain + 1e-6);
```

- Normalize training data to have zero mean and unit standard deviation.
- Compute the STFT mean and standard deviation of each frequency bin.

```
SMean = mean(STrain,[2 3 4]);
SStd = std(STrain,1,[2 3 4]);
```

- Normalize each frequency bin.

```
STrain = (STrain-SMean)./SStd;
```

Generate Real Drumbeat Training Data

- The computed STFTs have unbounded values. Following the approach in [1], make the data bounded by clipping the spectra to 3 standard deviations and rescaling to [-1 1].

```
STrain = STrain/3;
Y = reshape(STrain,numel(STrain),1);
Y(Y<-1) = -1;
Y(Y>1) = 1;
STrain = reshape(Y,size(STrain));
```

- Discard the last frequency bin to force the number of STFT bins to a power of two (which works well with convolutional layers).

```
STrain = STrain(1:end-1,:,:,:);
```

- Permute the dimensions in preparation for feeding to the discriminator.

```
STrain = permute(STrain,[2 1 3 4]);
```

Specify Training Options

- Train with a mini-batch size of 64 for 1000 epochs.

```
maxEpochs = 1000;
miniBatchSize = 64;
```

- Compute the number of iterations required to consume the data.

```
numIterationsPerEpoch = floor(size(STrain,4)/miniBatchSize);
```

Specify Training Options

- Specify the options for Adam optimization. Set the learn rate of the generator and discriminator to 0.0002. For both networks, use a gradient decay factor of 0.5 and a squared gradient decay factor of 0.999.

```
learnRateGenerator = 0.0002;
learnRateDiscriminator = 0.0002;
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
```

- Initialize the generator and discriminator weights. The initializeGeneratorWeights and initializeDiscriminatorWeights functions return random weights obtained using Glorot uniform initialization.

```
generatorParameters = initializeGeneratorWeights;
discriminatorParameters = initializeDiscriminatorWeights;
```

Train Model

- Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration.
- For each epoch, shuffle the training data and loop over mini-batches of data.
- For each mini-batch:
  - Generate a dlarray object containing an array of random values for the generator network.
  - For GPU training, convert the data to a gpuArray (Parallel Computing Toolbox) object.
  - Evaluate the model gradients using dlfeval (Deep Learning Toolbox) and the helper functions, modelDiscriminatorGradients and modelGeneratorGradients.
  - Update the network parameters using the adamupdate (Deep Learning Toolbox) function.

Train Model

- Initialize the parameters for Adam.

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

- You can set saveCheckpoints to true to save the updated weights and states to a MAT file every ten epochs. You can then use this MAT file to resume training if it is interrupted. For the purpose of this example, set saveCheckpoints to false.

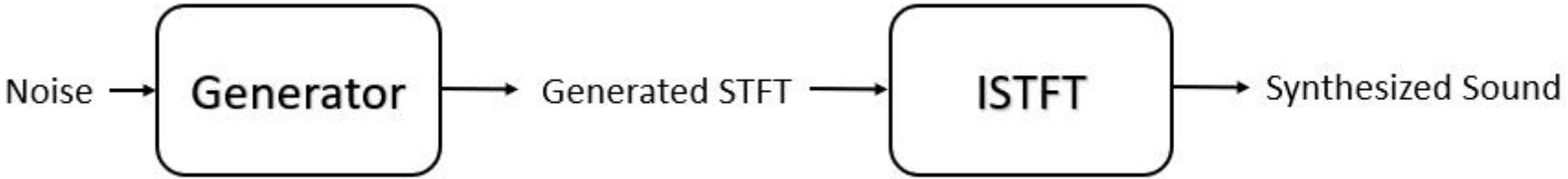
```
saveCheckpoints = false;
```

- Specify the length of the generator input.

```
numLatentInputs = 100;
```

Synthesize Sounds

- Now that you have trained the network, you can investigate the synthesis process in more detail.



Synthesize Sounds

- The trained drumbeat generator synthesizes short-time Fourier transform (STFT) matrices from input arrays of random values. An inverse STFT (ISTFT) operation converts the time-frequency STFT to a synthesized time-domain audio signal.
- Load the weights of a pretrained generator. These weights were obtained by running the training highlighted in the previous section for 1000 epochs.

```
load(matFileName,'generatorParameters','SMean','SStd');
```

- The generator takes 1-by-1-by-100 vectors of random values as an input. Generate a sample input vector.

```
numLatentInputs = 100;
dlZ = dlarray(2 * ( rand(1,1,numLatentInputs,1,'single') - 0.5 ));
```

Synthesize Sounds

- Pass the random vector to the generator to create an STFT image. generatorParameters is a structure containing the weights of the pretrained generator.

```
dlXGenerated = modelGenerator(dlZ,generatorParameters);
```

- Convert the STFT dlarray to a single-precision matrix.

```
S = dlXGenerated.extractdata;
```

- Transpose the STFT to align its dimensions with the istft function.

```
S = S.';
```

- The STFT is a 128-by-128 matrix, where the first dimension represents 128 frequency bins linearly spaced from 0 to 8 kHz. The generator was trained to generate a one-sided STFT from an FFT length of 256, with the last bin omitted. Reintroduce that bin by inserting a row of zeros into the STFT.

```
S = [S ; zeros(1,128)];
```

Synthesize Sounds

- Revert the normalization and scaling steps used when you generated the STFTs for training.

```
S = S * 3;
S = (S.*SStd) + SMean;
```

- Convert the STFT from the log domain to the linear domain.

```
S = exp(S);
```

- Convert the STFT from one-sided to two-sided.

```
S = [S; S(end-1:-1:2,:)];
```

- Pad with zeros to remove window edge-effects.

```
S = [zeros(256,100) S zeros(256,100)];
```

Synthesize Sounds

- The STFT matrix does not contain any phase information. Use a fast version of the Griffin-Lim algorithm with 20 iterations to estimate the signal phase and produce audio samples.

```
myAudio = stftmag2sig(S,256, ...
   'FrequencyRange','twosided', ...
   'Window',hann(256,'periodic'), ...
   'OverlapLength',128, ...
   'MaxIterations',20, ...
   'Method','fgla');
myAudio = myAudio./max(abs(myAudio),[],'all');
myAudio = myAudio(128*100:end-128*100);
```

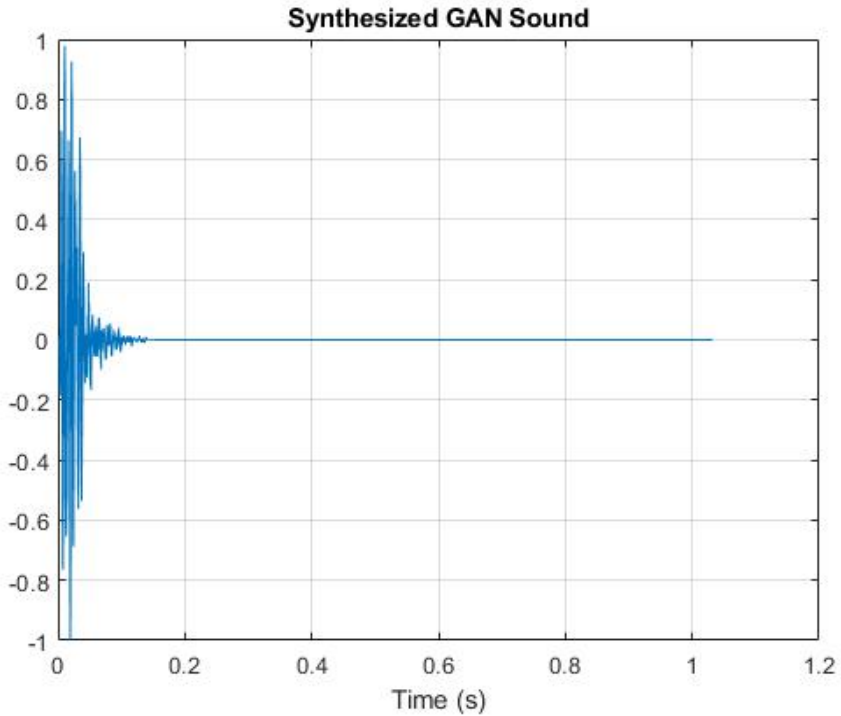
Synthesize Sounds

- Listen to the synthesized drumbeat.

```
sound(myAudio,fs)
```

- Plot the synthesized drumbeat.

```
t = (0:length(myAudio)-1)/fs;
plot(t,myAudio)
grid on
xlabel('Time (s)')
title('Synthesized GAN Sound')
```



Synthesize Sounds

- Plot the STFT of the synthesized drumbeat.

```
figure
stft(myAudio,fs,'Window',hann(256,'periodic'),'OverlapLength',128);
```