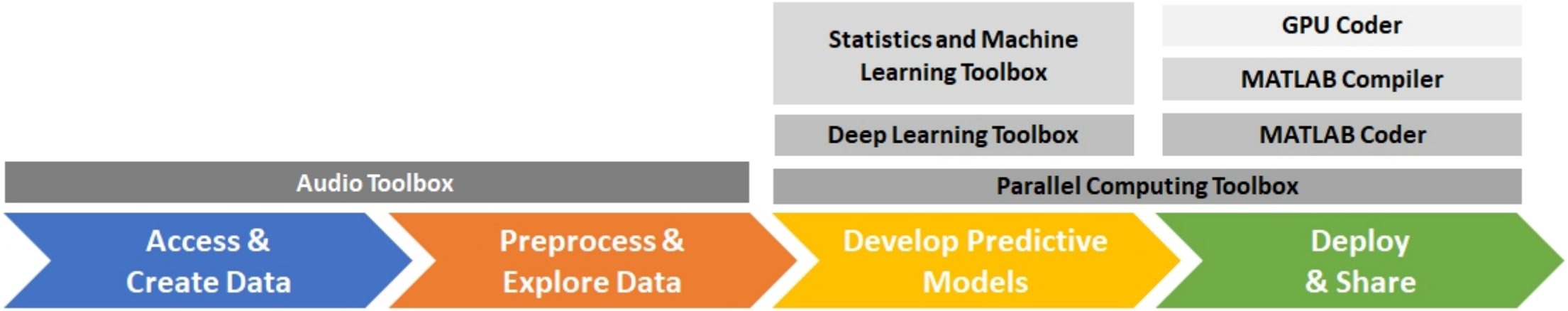
Deep Learning for Audio Applications

Outline

- Introduction to Deep Learning for Audio Applications
- Spoken Digit Recognition with Wavelet Scattering and Deep Learning

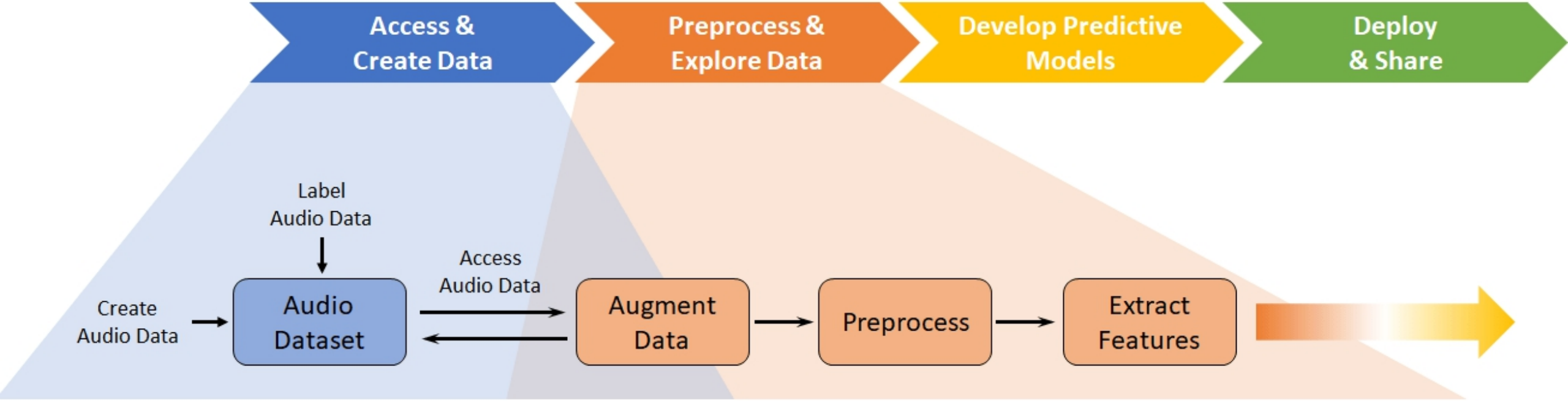
Introduction to Deep Learning for Audio Applications

Developing audio applications with deep learning typically includes creating and accessing data sets, preprocessing and exploring data, developing predictive models, and deploying and sharing applications. MATLAB provides toolboxes to support each stage of the development.



Introduction to Deep Learning for Audio Applications

While Audio Toolbox supports each stage of the deep learning workflow, its principal contributions are to Access and Create Data and Preprocess and Explore Data.



Access and Create Data

Deep learning networks perform best when you have access to large training data sets. However, the diversity of audio, speech, and acoustic signals, and a lack of large well-labeled data sets, makes accessing large training sets difficult. When using deep learning methods on audio files, you may need to develop new data sets or expand on existing ones. Audio Toolbox provides the Audio Labeler app to help you enlarge or create new labeled data sets.

Access and Create Data

Once you have an initial data set, you can enlarge it by applying augmentation techniques such as pitch shifting, time shifting, volume control, and noise addition. The type of augmentation you want to apply depends on the relevant characteristics for your audio, speech, or acoustic application. For example, pitch shifting (or vocal tract perturbation) and time stretching are typical augmentation techniques for automatic speech recognition (ASR). For far-field ASR, augmenting the training data by using artificial reverberation is common. Audio Toolbox provides audioDataAugmenter to help you apply augmentations deterministically or probabilistically.

Access and Create Data

The training data used in deep learning workflows is typically too large to fit in memory. Accessing data efficiently and performing common deep learning tasks (such as splitting a data set into train, validation, and test sets) can quickly become unmanageable. Audio Toolbox provides audioDatastore to help you manage and load large data sets.

Preprocess and Explore Data

Preprocessing audio data includes tasks like resampling audio files to a consistent sample rate, removing regions of silence, and trimming audio to a consistent duration. You can accomplish these tasks by using MATLAB, Signal Processing Toolbox and DSP System Toolbox. Audio Toolbox provides additional audio-specific tools to help you perform preprocessing, such as detectSpeech and voiceActivityDetector.

Preprocess and Explore Data

Audio is highly dimensional and contains redundant and often unnecessary information. Historically, mel-frequency cepstral coefficients (mfcc) and low-level features, such as the zero-crossing rate and spectral shape descriptors, have been the dominant features derived from audio signals for use in machine learning systems. Machine learning systems trained on these features are computationally efficient and typically require less training data. Audio Toolbox provides audioFeatureExtractor so that you can efficiently extract audio features.

Preprocess and Explore Data

Advances in deep learning architectures, increased access to computing power, and large and well-labeled data sets have decreased the reliance on hand-designed features. State-of-the-art results are often achieved using mel spectrograms (melSpectrogram), linear spectrograms, or raw audio waveforms. Audio Toolbox provides audioFeatureExtractor so that you can extract multiple auditory spectrograms.

Preprocess and Explore Data

Using audioFeatureExtractor enables you to systematically determine audio features for your deep learning model. Alternatively, you can use the melSpectrogram function to quickly extract just the mel spectrogram. Audio Toolbox also provides the modified discrete cosine transform (mdct), which returns a compact spectral representation without any loss of information.

Spoken Digit Recognition with Wavelet Scattering and Deep Learning

- This example shows how to classify spoken digits using both machine and deep learning techniques. In the example, you perform classification using wavelet time scattering with a support vector machine (SVM) and with a long short-term memory (LSTM) network. You also apply Bayesian optimization to determine suitable hyperparameters to improve the accuracy of the LSTM network. In addition, the example illustrates an approach using a deep convolutional neural network (CNN) and mel-frequency spectrograms.

Data

- Clone or download the Free Spoken Digit Dataset (FSDD), available at https://github.com/Jakobovski/free-spoken-digit-dataset. FSDD is an open data set, which means that it can grow over time. This example uses the version committed on January 29, 2019, which consists of 2000 recordings in English of the digits 0 through 9 obtained from four speakers. In this version, two of the speakers are native speakers of American English, one speaker is a nonnative speaker of English with a Belgian French accent, and one speaker is a nonnative speaker of English with a German accent. The data is sampled at 8000 Hz.

Data

- Use audioDatastore to manage data access and ensure the random division of the recordings into training and test sets. Set the location property to the location of the FSDD recordings folder on your computer, for example:

```
pathToRecordingsFolder = fullfile('free-spoken-digit-dataset','recordings');
location = pathToRecordingsFolder;
```

- Point audioDatastore to that location.

```
ads = audioDatastore(location);
```

Data

- The helper function helpergenLabels creates a categorical array of labels from the FSDD files. The source code for helpergenLabels is listed in the appendix. List the classes and the number of examples in each class.

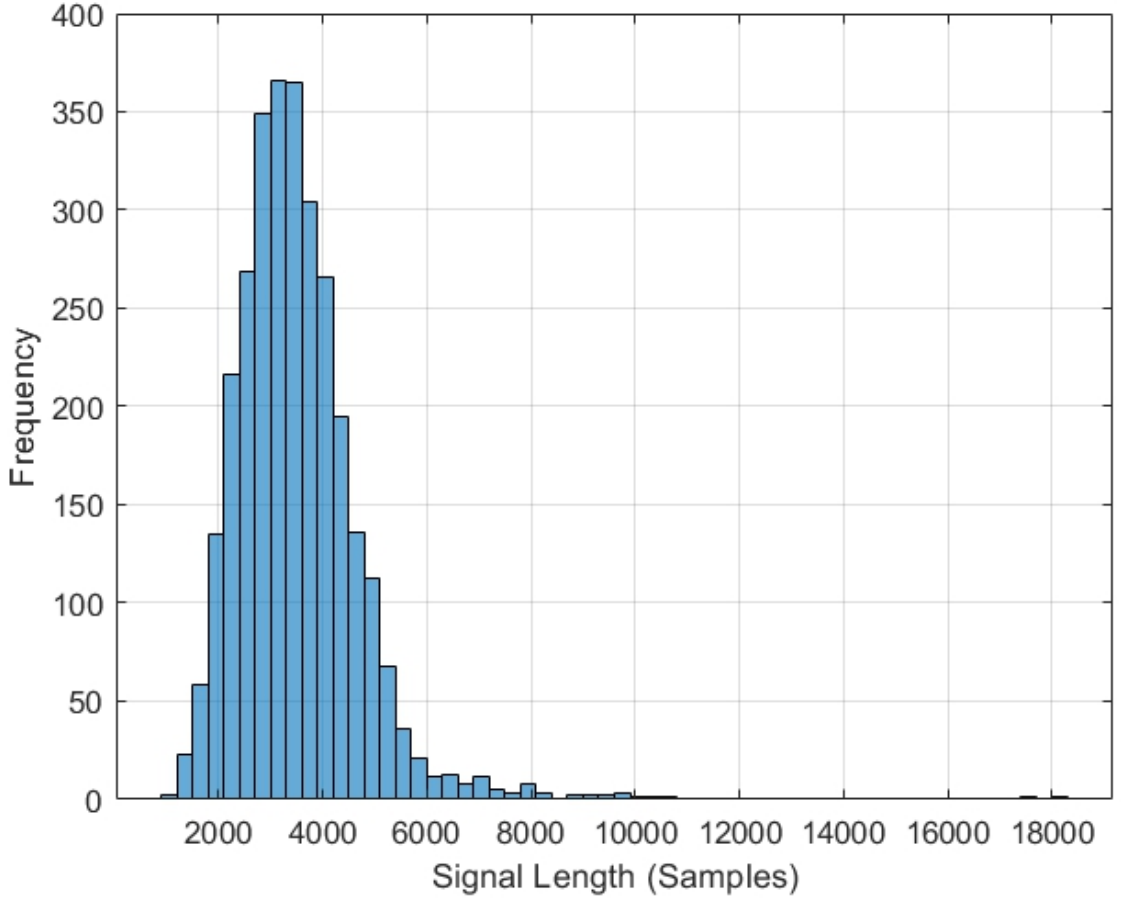
```
ads.Labels = helpergenLabels(ads);

summary(ads.Labels)
```

0 300
1 300
2 300
3 300
4 300
5 300
6 300
7 300
8 300
9 300

Data

- The FSDD data set consists of 10 balanced classes with 200 recordings each. The recordings in the FSDD are not of equal duration. The FSDD is not prohibitively large, so read through the FSDD files and construct a histogram of the signal lengths.

```
LenSig = zeros(numel(ads.Files),1);
nr = 1;
while hasdata(ads)
digit = read(ads);
LenSig(nr) = numel(digit);
nr = nr+1;
end
reset(ads)
histogram(LenSig)
grid on
xlabel('Signal Length (Samples)')
ylabel('Frequency')
```



Wavelet Time Scattering

- Use *waveletScattering* to create a wavelet time scattering framework using an invariant scale of 0.22 seconds. In this example, you create feature vectors by averaging the scattering transform over all time samples. To have a sufficient number of scattering coefficients per time window to average, set OversamplingFactor to 2 to produce a four-fold increase in the number of scattering coefficients for each path with respect to the critically downsampled value.

```
sf = waveletScattering('SignalLength',8192,'InvarianceScale',0.22,...
'SamplingFrequency',8000,'OversamplingFactor',2);
```

Wavelet Time Scattering

- Split the FSDD into training and test sets. Allocate 80% of the data to the training set and retain 20% for the test set. The training data is for training the classifier based on the scattering transform. The test data is for validating the model.

```
rng default;
ads = shuffle(ads);
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
countEachLabel(adsTrain)
```

	Label	Count
1	NaN	240
2	NaN	240
3	NaN	240
4	NaN	240
5	NaN	240
6	NaN	240
7	NaN	240
8	NaN	240
9	NaN	240
10	NaN	240

Wavelet Time Scattering

countEachLabel(adsTest)

	Label	Count
1	NaN	60
2	NaN	60
3	NaN	60
4	NaN	60
5	NaN	60
6	NaN	60
7	NaN	60
8	NaN	60
9	NaN	60
10	NaN	60

Wavelet Time Scattering

- The helper function helperReadSPData truncates or pads the data to a length of 8192 and normalizes each recording by its maximum value. The source code for helperReadSPData is listed in the appendix. Create an 8192-by-1600 matrix where each column is a spoken-digit recording.

```
Xtrain = [];

scatds_Train = transform(adsTrain,@(x)helperReadSPData(x));

while hasdata(scatds_Train)

smat = read(scatds_Train);

Xtrain = cat(2,Xtrain,smat);

end
```

Wavelet Time Scattering

- Apply the wavelet scattering transform to the training and test sets.

```
Strain = sf.featureMatrix(Xtrain);

Stest = sf.featureMatrix(Xtest);
```

- Obtain the mean scattering features for the training and test sets. Exclude the zeroth-order scattering coefficients.

```
TrainFeatures = Strain(2:end,:,:);

TrainFeatures = squeeze(mean(TrainFeatures,2))';

TestFeatures = Stest(2:end,:,:);

TestFeatures = squeeze(mean(TestFeatures,2))';
```

SVM Classifier

- Now that the data has been reduced to a feature vector for each recording, the next step is to use these features for classifying the recordings. Create an SVM learner template with a quadratic polynomial kernel. Fit the SVM to the training data.

```
template = templateSVM(...
'KernelFunction', 'polynomial', 'PolynomialOrder', 2, ...
'KernelScale', 'auto', 'BoxConstraint', 1, ...
'Standardize', true);


classificationSVM = fitcecoc(...
TrainFeatures, ...
adsTrain.Labels, ...
'Learners', template, ...
'Coding', 'onevsone', ...
'ClassNames', categorical({'0'; '1'; '2'; '3'; '4'; '5'; '6'; '7'; '8'; '9'}));
```

SVM Classifier

- Use k-fold cross-validation to predict the generalization accuracy of the model based on the training data. Split the training set into five groups.

```
partitionedModel = crossval(classificationSVM, 'KFold', 5);

[validationPredictions, validationScores] = kfoldPredict(partitionedModel);

validationAccuracy = (1 - kfoldLoss(partitionedModel, 'LossFun', 'ClassifError'))*100
```

```
validationAccuracy = 97.2500
```

SVM Classifier

- The estimated generalization accuracy is approximately 97%. Use the trained SVM to predict the spoken-digit classes in the test set.

```
predLabels = predict(classificationSVM,TestFeatures);

testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100
```

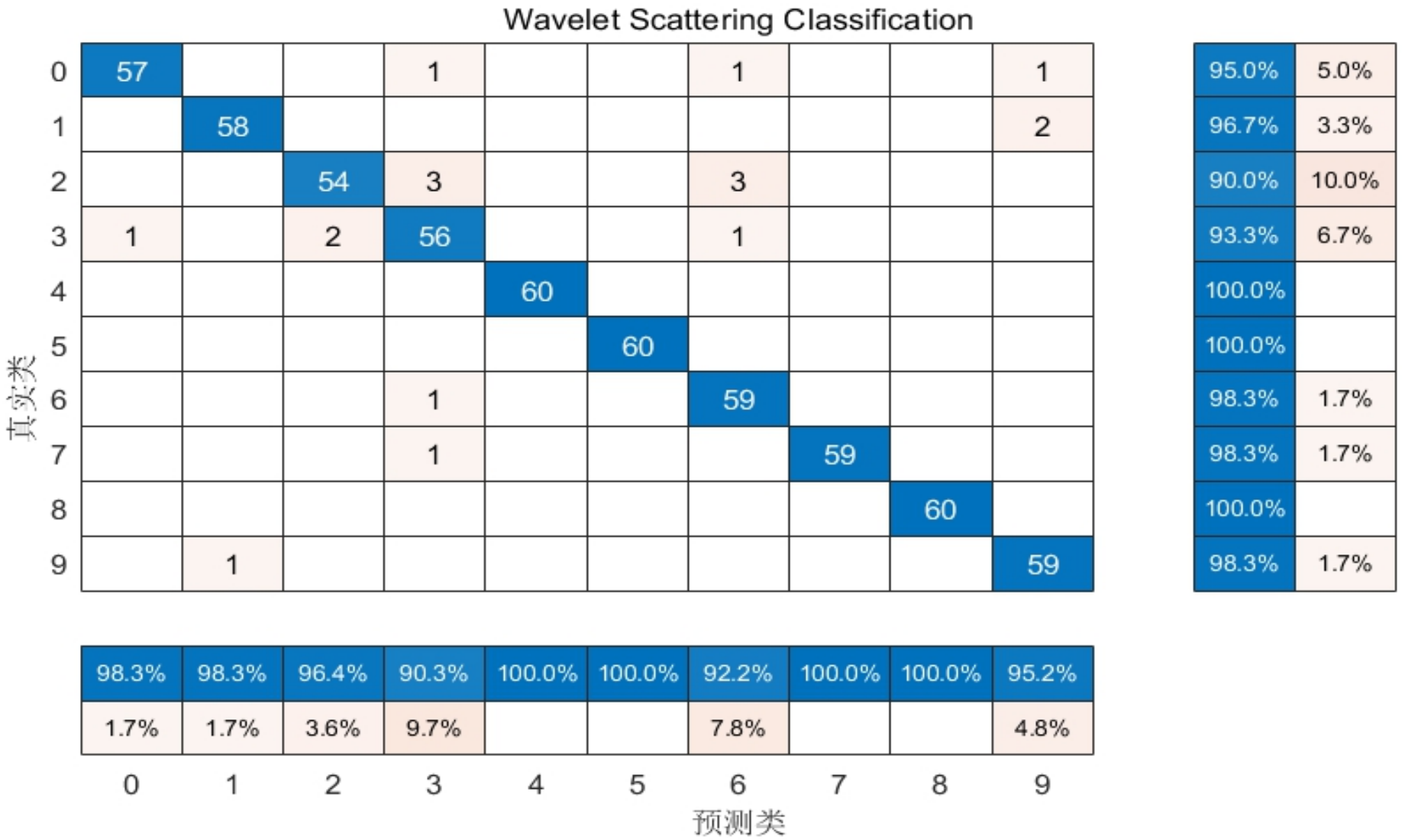
```
testAccuracy = 97
```

SVM Classifier

- Summarize the performance of the model on the test set with a confusion chart. Display the precision and recall for each class by using column and row summaries. The table at the bottom of the confusion chart shows the precision values for each class. The table to the right of the confusion chart shows the recall values.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
ccscat = confusionchart(adsTest.Labels,predLabels);
ccscat.Title = 'Wavelet Scattering Classification';
ccscat.ColumnSummary = 'column-normalized';
ccscat.RowSummary = 'row-normalized';
```

SVM Classifier



Long Short-Term Memory (LSTM) Networks

- An LSTM network is a type of recurrent neural network (RNN). RNNs are neural networks that are specialized for working with sequential or temporal data such as speech data. Because the wavelet scattering coefficients are sequences, they can be used as inputs to an LSTM. By using scattering features as opposed to the raw data, you can reduce the variability that your network needs to learn.

Long Short-Term Memory (LSTM) Networks

- Modify the training and testing scattering features to be used with the LSTM network. Exclude the zeroth-order scattering coefficients and convert the features to cell arrays.

```
TrainFeatures = Strain(2:end,:,:);

TrainFeatures = squeeze(num2cell(TrainFeatures,[1 2]));

TestFeatures = Stest(2:end,:,:);

TestFeatures = squeeze(num2cell(TestFeatures, [1 2]));
```

Long Short-Term Memory (LSTM) Networks

- Construct a simple LSTM network with 512 hidden layers.

```
[inputSize, ~] = size(TrainFeatures{1});
YTrain = adsTrain.Labels;

numHiddenUnits = 512;
numClasses = numel(unique(YTrain));

layers = [ ...
sequenceInputLayer(inputSize)
lstmLayer(numHiddenUnits,'OutputMode','last')
fullyConnectedLayer(numClasses)
softmaxLayer
classificationLayer];
```

Long Short-Term Memory (LSTM) Networks

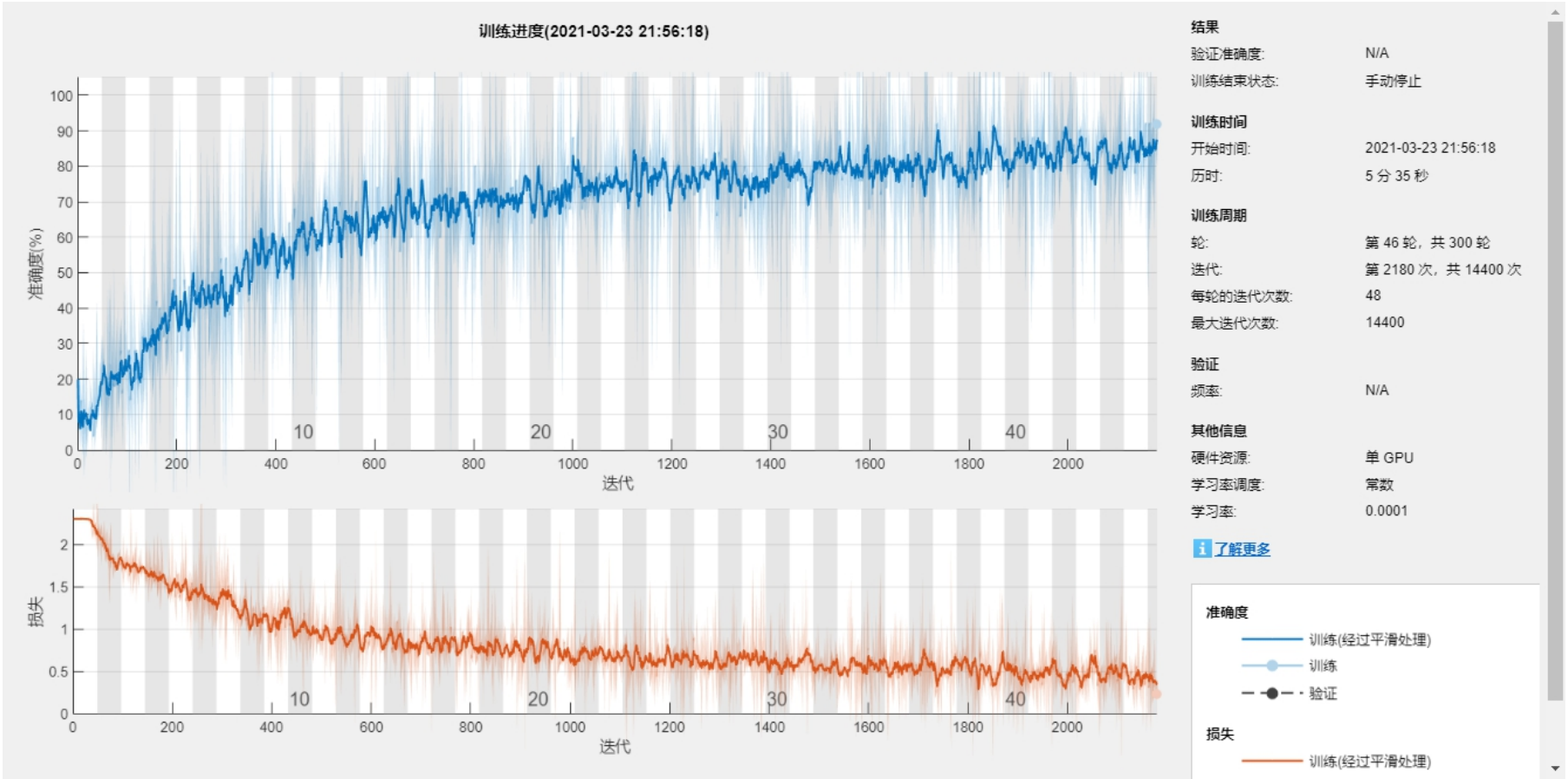
- Set the hyperparameters. Use Adam optimization and a mini-batch size of 50. Set the maximum number of epochs to 300. Use a learning rate of 1e-4. You can turn off the training progress plot if you do not want to track the progress using plots. The training uses a GPU by default if one is available. Otherwise, it uses a CPU.

```
maxEpochs = 300;

miniBatchSize = 50;


options = trainingOptions('adam', ...

'InitialLearnRate',0.0001,...

'MaxEpochs',maxEpochs, ...

'MiniBatchSize',miniBatchSize, ...

'SequenceLength','shortest', ...

'Shuffle','every-epoch',...

'Verbose', false, ...

'Plots','training-progress');
```

Long Short-Term Memory (LSTM) Networks

- Train the network.

net = trainNetwork(TrainFeatures,YTrain,layers,options);



Long Short-Term Memory (LSTM) Networks

```
predLabels = classify(net,TestFeatures);
testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100
```

```
testAccuracy = 85
```

Bayesian Optimization

- Determining suitable hyperparameter settings is often one of the most difficult parts of training a deep network. To mitigate this, you can use Bayesian optimization. In this example, you optimize the number of hidden layers and the initial learning rate by using Bayesian techniques. Create a new directory to store the MAT-files containing information about hyperparameter settings and the network along with the corresponding error rates.

```
YTrain = adsTrain.Labels;
YTest = adsTest.Labels;
if ~exist("results/",'dir')
mkdir results
end
```

Bayesian Optimization

- Initialize the variables to be optimized and their value ranges. Because the number of hidden layers must be an integer, set 'type' to 'integer'.

```
optVars = [
optimizableVariable('InitialLearnRate',[1e-5, 1e-1],'Transform','log')
optimizableVariable('NumHiddenUnits',[10, 1000],'Type','integer')
];
```

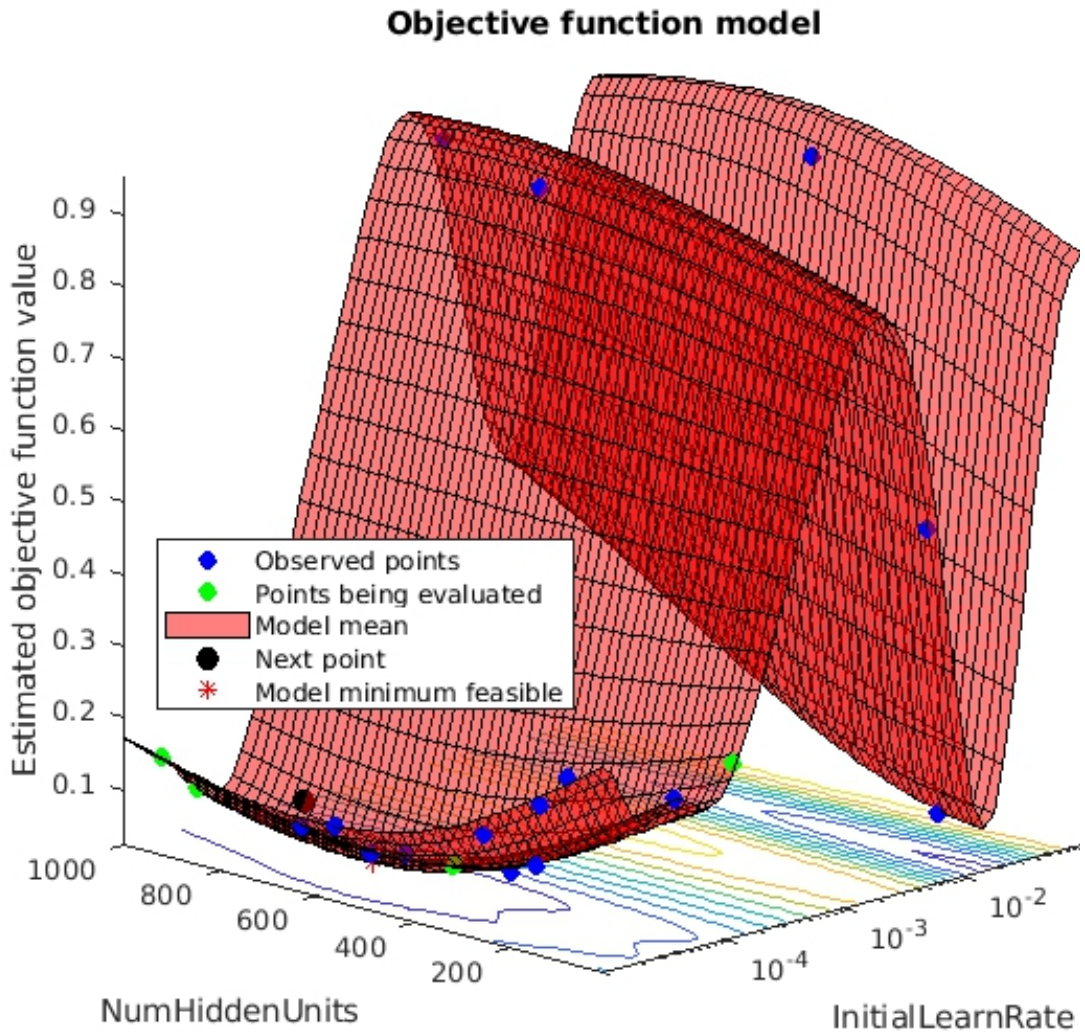
Bayesian Optimization

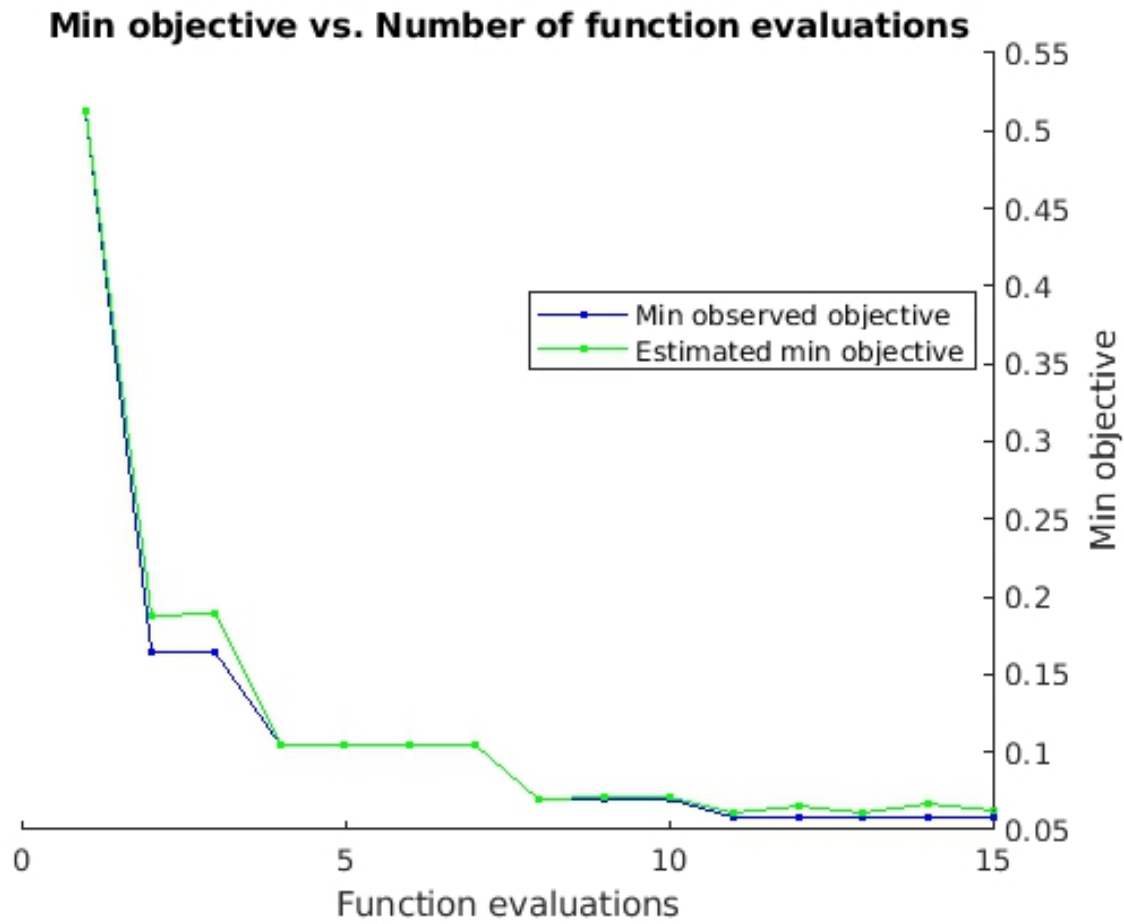
- Bayesian optimization is computationally intensive and can take several hours to finish. For the purposes of this example, set optimizeCondition to false to download and use predetermined optimized hyperparameter settings. If you set optimizeCondition to true, the objective function helperBayesOptLSTM is minimized using Bayesian optimization. The objective function, listed in the appendix, is the error rate of the network given specific hyperparameter settings. The loaded settings are for the objective function minimum of 0.02 (2% error rate).

Bayesian Optimization

```
ObjFcn = helperBayesOptLSTM(TrainFeatures,YTrain,TestFeatures,YTest);
optimizeCondition = true;
if optimizeCondition
BayesObject = bayesopt(ObjFcn,optVars,...
'MaxObjectiveEvaluations',15,...
'IsObjectiveDeterministic',false,...
'UseParallel',true);
end
```

Bayesian Optimization





Bayesian Optimization

- Use the optimized values for the number of hidden units and initial learning rate and retrain the network.

```
numHiddenUnits = 768;
numClasses = numel(unique(YTrain));
layers = [ ...
sequenceInputLayer(inputSize)
lstmLayer(numHiddenUnits,'OutputMode','last')
fullyConnectedLayer(numClasses)
softmaxLayer
classificationLayer];
maxEpochs = 300;
miniBatchSize = 50;
```

```
options = trainingOptions('adam', ...
'InitialLearnRate',2.198827960269379e-04,...
'MaxEpochs',maxEpochs, ...
'MiniBatchSize',miniBatchSize, ...
'SequenceLength','shortest', ...
'Shuffle','every-epoch',...
'Verbose', false, ...
'Plots','training-progress');
net = trainNetwork(TrainFeatures,YTrain,layers,options);
predLabels = classify(net,TestFeatures);
testAccuracy =
sum(predLabels==adsTest.Labels)/numel(predLabels)*100
```

Deep Convolutional Network Using Mel-Frequency Spectrograms

- As another approach to the task of spoken digit recognition, use a deep convolutional neural network (DCNN) based on mel-frequency spectrograms to classify the FSDD data set. Use the same signal truncation/padding procedure as in the scattering transform. Similarly, normalize each recording by dividing each signal sample by the maximum absolute value. For consistency, use the same training and test sets as for the scattering transform.

Deep Convolutional Network Using Mel-Frequency Spectrograms

- Set the parameters for the mel-frequency spectrograms. Use the same window, or frame, duration as in the scattering transform, 0.22 seconds. Set the hop between windows to 10 ms. Use 40 frequency bands.

```
segmentDuration = 8192*(1/8000);
frameDuration = 0.22;
hopDuration = 0.01;
numBands = 40;
```

- Reset the training and test datastores.

```
reset(adsTrain);
reset(adsTest);
```

Deep Convolutional Network Using Mel-Frequency Spectrograms

- The helper function helperspeechSpectrograms, defined at the end of this example, uses melSpectrogram to obtain the mel-frequency spectrogram after standardizing the recording length and normalizing the amplitude. Use the logarithm of the mel-frequency spectrograms as the inputs to the DCNN. To avoid taking the logarithm of zero, add a small epsilon to each element.

```
epsil = 1e-6;

XTrain = helperspeechSpectrograms(adsTrain,segmentDuration,frameDuration,hopDuration,numBands);

XTrain = log10(XTrain + epsil);

XTest = helperspeechSpectrograms(adsTest,segmentDuration,frameDuration,hopDuration,numBands);

XTest = log10(XTest + epsil);

YTrain = adsTrain.Labels;

YTest = adsTest.Labels;
```

Define DCNN Architecture

- Construct a small DCNN as an array of layers. Use convolutional and batch normalization layers, and downsample the feature maps using max pooling layers. To reduce the possibility of the network memorizing specific features of the training data, add a small amount of dropout to the input to the last fully connected layer.

```
sz = size(XTrain);

specSize = sz(1:2);

imageSize = [specSize 1];

numClasses = numel(categories(YTrain));

dropoutProb = 0.2;

numF = 12;
```

Define DCNN Architecture

```
layers = [
imageInputLayer(imageSize)
convolution2dLayer(5,numF,'Padding','same')
batchNormalizationLayer
reluLayer
maxPooling2dLayer(3,'Stride',2,'Padding','same')
convolution2dLayer(3,2*numF,'Padding','same')
batchNormalizationLayer
reluLayer
maxPooling2dLayer(3,'Stride',2,'Padding','same')
convolution2dLayer(3,4*numF,'Padding','same')
batchNormalizationLayer
reluLayer
maxPooling2dLayer(3,'Stride',2,'Padding','same')
convolution2dLayer(3,4*numF,'Padding','same')
batchNormalizationLayer
reluLayer
convolution2dLayer(3,4*numF,'Padding','same')
batchNormalizationLayer
reluLayer
maxPooling2dLayer(2)
dropoutLayer(dropoutProb)
fullyConnectedLayer(numClasses)
softmaxLayer
classificationLayer('Classes',categories(YTrain));
];
```

Define DCNN Architecture

- Set the hyperparameters to use in training the network. Use a mini-batch size of 50 and a learning rate of 1e-4. Specify Adam optimization. Because the amount of data in this example is relatively small, set the execution environment to 'cpu' for reproducibility. You can also train the network on an available GPU by setting the execution environment to either 'gpu' or 'auto'.

```
miniBatchSize = 50;

options = trainingOptions('adam', ...

'InitialLearnRate',1e-4, ...

'MaxEpochs',30, ...

'MiniBatchSize',miniBatchSize, ...

'Shuffle','every-epoch', ...

'Plots','training-progress', ...

'Verbose',false, ...

'ExecutionEnvironment','cpu');
```

Define DCNN Architecture

- Train the network.

```
trainedNet = trainNetwork(XTrain,YTrain,layers,options);
```

- Use the trained network to predict the digit labels for the test set.

```
[Ypredicted,probs] = classify(trainedNet,XTest,'ExecutionEnvironment','CPU');
cnnAccuracy = sum(Ypredicted==YTest)/numel(YTest)*100
```

Define DCNN Architecture

- Summarize the performance of the trained network on the test set with a confusion chart. Display the precision and recall for each class by using column and row summaries. The table at the bottom of the confusion chart shows the precision values. The table to the right of the confusion chart shows the recall values.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);

ccDCNN = confusionchart(YTest,Ypredicted);

ccDCNN.Title = 'Confusion Chart for DCNN';

ccDCNN.ColumnSummary = 'column-normalized';

ccDCNN.RowSummary = 'row-normalized';
```

- The DCNN using mel-frequency spectrograms as inputs classifies the spoken digits in the test set with an accuracy rate of approximately 98% as well.

Summary

- This example shows how to use different machine and deep learning approaches for classifying spoken digits in the FSDD. The example illustrated wavelet scattering paired with both an SVM and a LSTM. Bayesian techniques were used to optimize LSTM hyperparameters. Finally, the example shows how to use a CNN with mel-frequency spectrograms.
- The goal of the example is to demonstrate how to use MathWorks® tools to approach the problem in fundamentally different but complementary ways. All workflows use audioDatastore to manage flow of data from disk and ensure proper randomization.

Summary

- All approaches used in this example performed equally well on the test set. This example is not intended as a direct comparison between the various approaches. For example, you can also use Bayesian optimization for hyperparameter selection in the CNN. An additional strategy that is useful in deep learning with small training sets like this version of the FSDD is to use data augmentation. How manipulations affect class is not always known, so data augmentation is not always feasible. However, for speech, established data augmentation strategies are available through audioDataAugmenter.
- In the case of wavelet time scattering, there are also a number of modifications you can try. For example, you can change the invariant scale of the transform, vary the number of wavelet filters per filter bank, and try different classifiers.