This section provides step-by-step instructions on how to create and train a convolutional neural network for classification of this paper.

This section demonstrates how to:

* Load and explore image data.
* Define the network architecture.
* Specify training options.
* Train the network.
* Predict the labels of new data and calculate the classification accuracy.

### Load and Explore Image Data

Load the spectrograms as an image datastore. *imageDatastore* automatically labels the images based on folder names and stores the data as an *ImageDatastore* object. An image datastore supports large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

%% Specify the dataset directory

currentFolder = pwd;

imds = imageDatastore(fullfile(currentFolder, 'trainingData'), ...

'IncludeSubfolders',true,'LabelSource','foldernames');

Calculate the number of images in each category. *labelCount* is a table that contains the labels and the number of images having each label. The datastore contains 315 images for spectrogram type A and 200 images for spectrograms type B.

%% Getting number of labels

labelCount = countEachLabel(imds);

labelCount = 2×2 table

Label Count

\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_

spectrogramA 315

spectrogramB 200

Please note that the size of the images in the input layer of the network must be specified. It can be check by the following code:

img = readimage(imds,1);

imageSize = size(img);

### Specify Training and Validation Sets

The data are divided into training and validation data sets, in this case the training set contains 150 images for each category, and the validation set contains the remaining images. *splitEachLabel* splits the datastore into two new datastores, *imdsTrain* and *imdsValidation*.

%% Specify the number of files used for training

filesToTrain = 150;

[imdsTrain,imdsValidation] = splitEachLabel(imds,filesToTrain,'randomize');

### Define Network Architecture

Define the convolutional neural network architecture.

%% Declare the CNN network

layers = [

imageInputLayer(size(img))

convolution2dLayer(64,8,'Padding','same')

batchNormalizationLayer

reluLayer

maxPooling2dLayer(32,'Stride',8)

convolution2dLayer(32,16,'Padding','same')

batchNormalizationLayer

reluLayer

maxPooling2dLayer(16,'Stride',16)

convolution2dLayer(16,32,'Padding','same')

batchNormalizationLayer

reluLayer

fullyConnectedLayer(2)

softmaxLayer

classificationLayer];

**Image Input Layer** As the name suggests, the image size is specified here. Function *size* returns height, width, and the channel size for imageInputLayer. The digit data consists of grayscale images, so the channel size (color channel) is 1. By default, the network shuffles the data at the beginning of training.

**Convolutional Layer** In the convolutional layer, there are two main arguments. The first one is filterSize, which is the height and width of the filters the training function uses while scanning along the images. The second argument is numFilters, this indicates the number of neurons that connect to the same region of the input and thus determines the number of feature maps. Also, 'Padding' is used to add padding to the input feature map and 'same' padding are used to ensure that the spatial output size is the same as the input size.

**Batch Normalization Layer** Batch normalization layers normalize the activations and gradients propagating through a network, making network training an easier optimization problem. Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up network training and reduce the sensitivity to network initialization.

**ReLU Layer** One of the commonly used nonlinear activation function.

**Max Pooling Layer** Used to down-sampling to reduce the spatial size of the feature map and remove redundant spatial information. This reduces the amount of computational power required and make it possible to increase the number of filters in following layers. The first argument poolSize specifies the maximum values of rectangular regions of inputs that the max pooling layer returns. The 'Stride' name-value pair argument specifies the step size that the training function takes as it scans along the input.

**Fully Connected Layer** Following the convolutional and down-sampling layers, this is the layer where the neurons connect to all the neurons in the preceding layer. Therefore, all the features learned by previous layers can be combined to identify larger patterns. The OutputSize parameter in the last fully connected layer should be equal to the number of classes in the target data which is 2 in this case. Because the last fully connected layer combines the features to classify the images.

**Softmax Layer** Used to create classification probabilities by the classification layer. The softmax activation function normalizes the output of the fully connected layer and its output are positive numbers that sum to one.

**Classification Layer** The final layer. After the softmax activation function returns the softmax activation function, this layer assigns the input to one of the mutually exclusive classes and compute the loss.

### Specify Training Options

After defining the network structure, training options should be specified. In this case, the network uses stochastic gradient descent with momentum (SGDM) with an initial learning rate of 0.01. The number of epochs is 5. An epoch is a full training cycle on the entire training data set. validation data and validation frequency are used to monitor the network accuracy during training. Shuffle the data every epoch. The network trains on the training data and calculates the accuracy on the validation data at regular intervals during training. The validation data is only used for accuracy calculation and not for updating the network weights.

%% Specify Training Options for the model

options = trainingOptions('sgdm', ...

'InitialLearnRate',0.01, ...

'MaxEpochs',5, ...

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

'ValidationData',imdsValidation, ...

'ValidationFrequency',30, ...

'Verbose',false, ...

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

'MiniBatchSize',2, ...

'ExecutionEnvironment','gpu');

### Train Network Using Training Data

Training the network comes after defining the layers, the training data, and the training options. This project uses supported GPU device to train the network, but it is also possible to train it using CPU. By default, trainNetwork uses a GPU if one is available, otherwise, it uses a CPU. By default, the execution environment will use the GPU but it can also be specified by using the 'ExecutionEnvironment' name-value pair argument of trainingOptions.

The training progress plot shows the mini-batch loss and accuracy and the validation loss and accuracy. The loss is the cross-entropy loss. The accuracy is the percentage of images that the network classifies correctly.

%% Train The CNN Network

net = trainNetwork(imdsTrain,layers,options);

### Classify Validation Images and Compute Accuracy

YPred is the result after predicting the labels, while YValidation is the validation data. With these two datasets, accuracy, which is the fraction of labels that was correctly predicted, can be calculated.

A confusion matrix can be plotted using *plotconfusion.*

Some other statistics are also calculated in this project. These are True Positive (TP), True Negative (TN), False Positive (FP), False Negative (FN). Consequently, with the aforementioned values, Positive Predictive Value (PPV), False Discovery Rate (FDR), Negative Predictive Value (NPV), False Omission Rate (FOR), True Positive Rate (TPR), True Negative Rate (TNR), F-score (F1), False Positive Rate (FPR) can be calculated.

%% Checking Network Performance

YPred = classify(net,imdsValidation);

YValidation = imdsValidation.Labels;

accuracy = sum(YPred == YValidation)/numel(YValidation);

plotconfusion(YPred,YValidation)

TP = 0;

TN = 0;

FP = 0;

FN = 0;

for i=1:size(YPred)

if (YValidation(i)=="spectrogramA")&&(YPred(i)=="spectrogramA")

TP = TP+1;

end

if (YValidation(i)=="spectrogramB")&&(YPred(i)=="spectrogramB")

TN = TN+1;

end

if (YValidation(i)=="spectrogramB")&&(YPred(i)=="spectrogramA")

FP = FP+1;

end

if (YValidation(i)=="spectrogramA")&&(YPred(i)=="spectrogramB")

FN = FN+1;

end

end

PPV = TP/(FP+TP);

FDR = FP/(FP+TP);

NPV = TN/(TN+FN);

FOR = FN/(TN+FN);

TPR = TP/(TP+FN);

TNR = TN/(TN+FP);

F1 = 2\*TP/(2\*TP+FP+FN);

FPR = FP/(FP+TN);

FPRm = [0 FPR 1];

TPRm = [0 TPR 1];

figure

plot(FPRm, TPRm)

grid

axis([0 1 0 1])

### Save the network

Saving the state of the network.

%% Save the Network for future validation

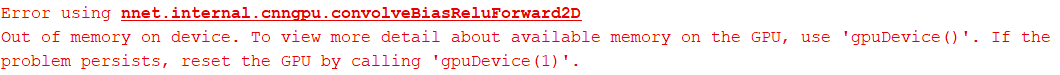
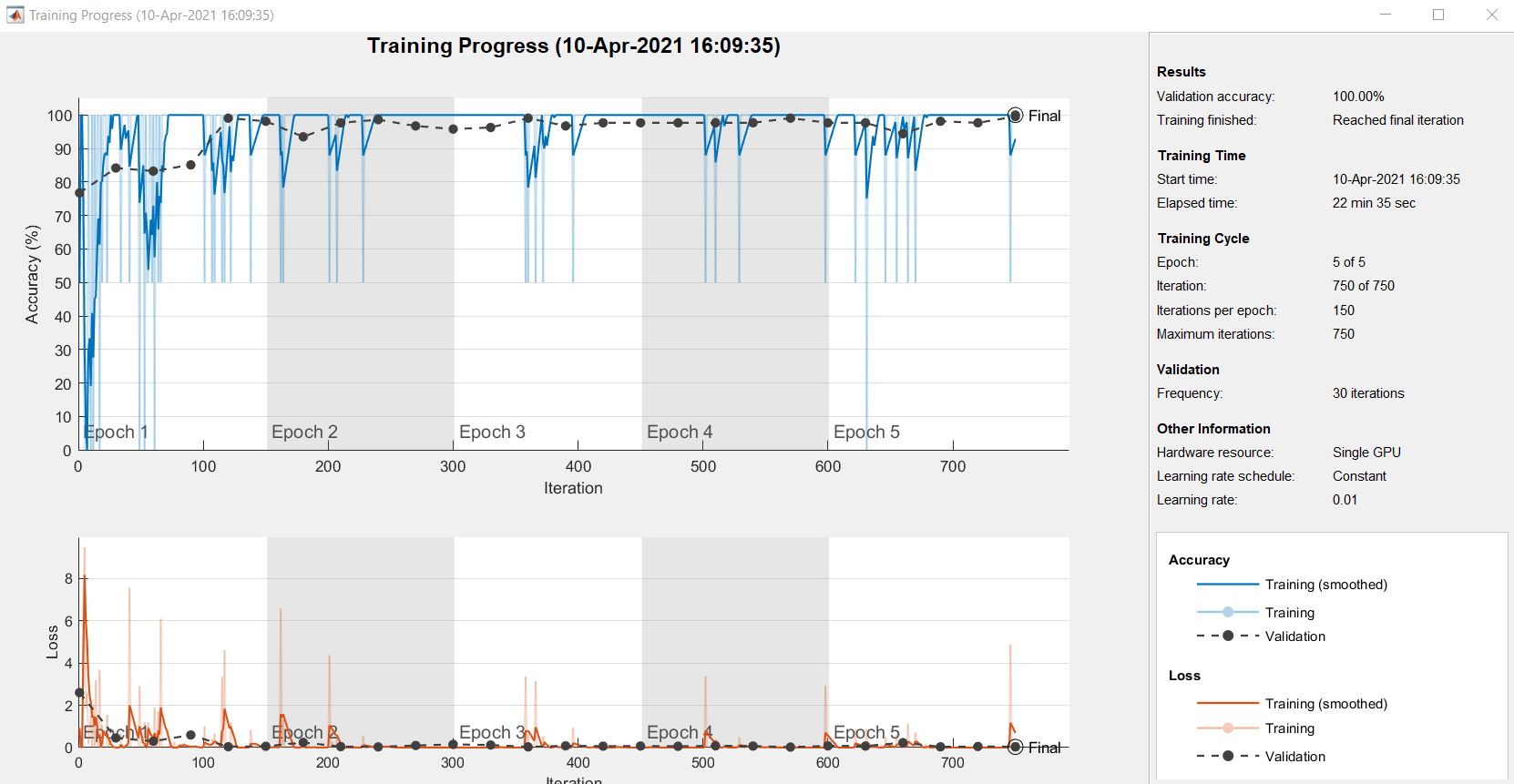
save net;

Performance Analysis

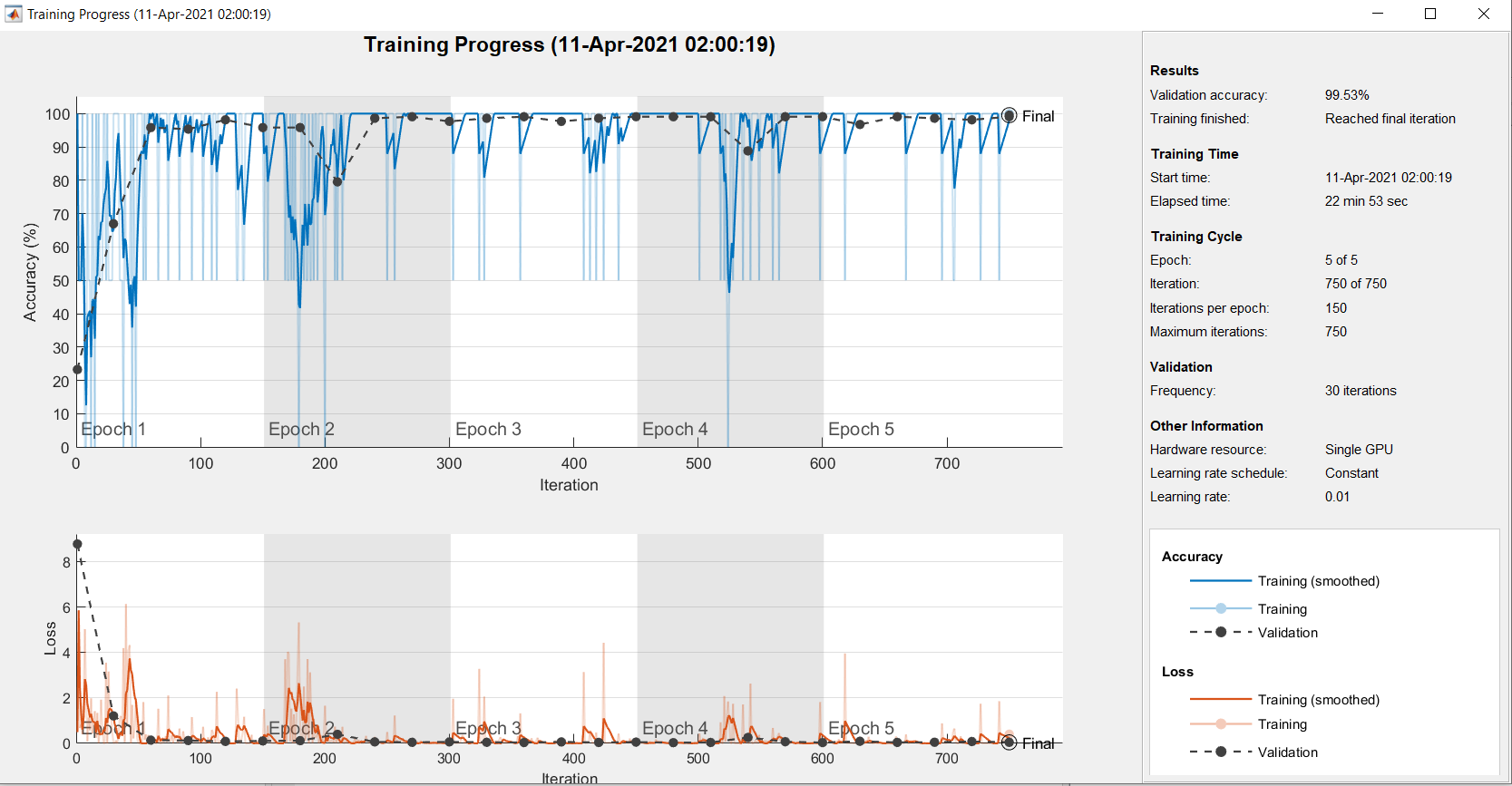
In this section, the performance of the input different spectrograms would be tested and evaluate to choose the most appropriate spectrograms. Recall that the spectrograms were created with RGB colors but later changed to gray scale. Also, the resolution of these spectrograms should be considered. Because these factors influence the speed, accuracy and the integrity of the training process. Of course, to keep the consistency of the results, the layers of the network are not changed, only the spectrograms are changed.

Please note that the following results may only apply to the current hardware and may be different on other hardware. Given that the network runs on GPU, for your information, the GPU that was used to run this experiment is a RTX 2060 with 6GB of VRAM.

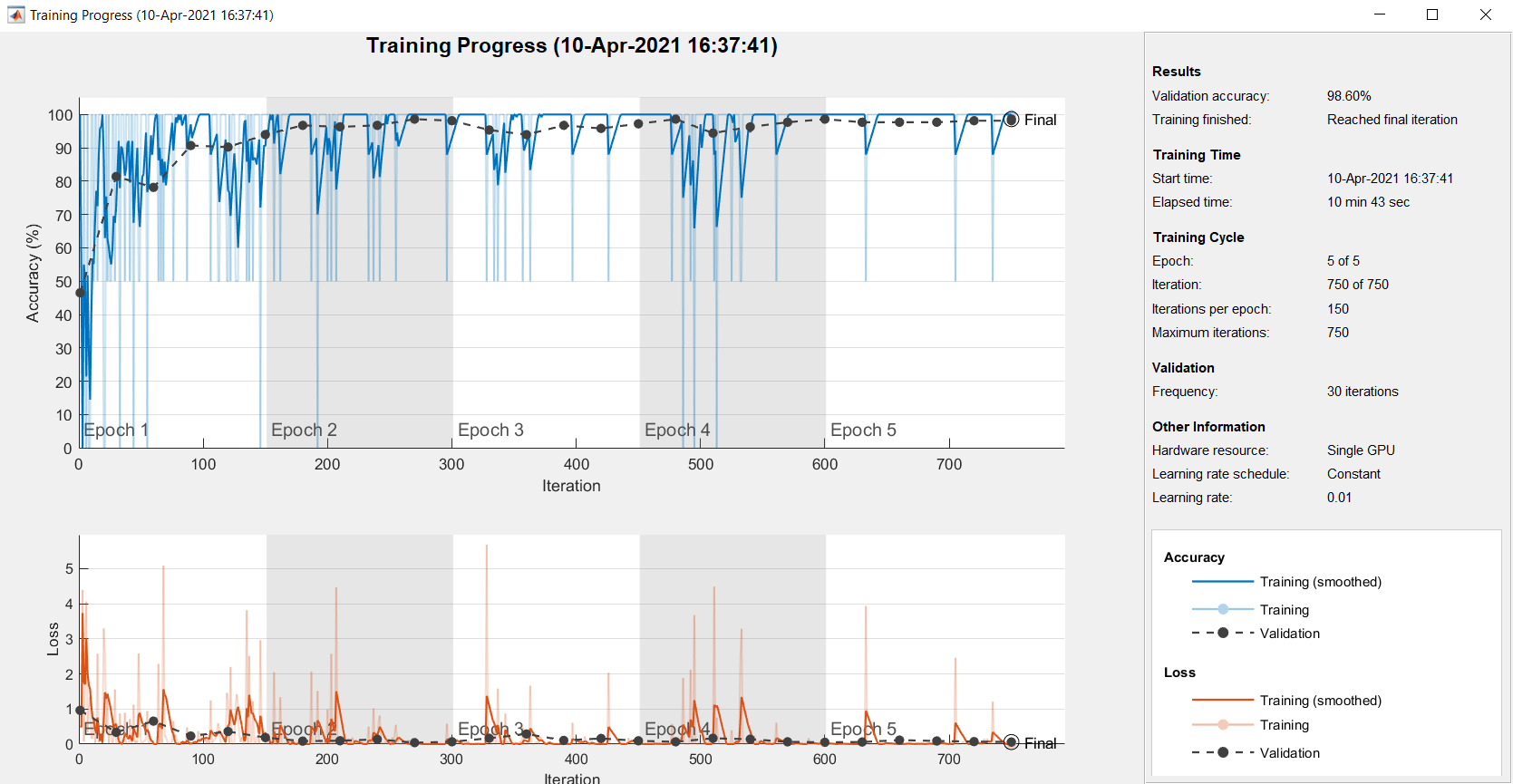
With the chosen spectrograms’ resolution of 682x540 RGB scale. The network achieves 100% accuracy with the total training and validation time of 22 minutes 53 seconds. However, running with this resolution repeatedly several times and it was noticed that sometimes there was error about GPU running out of memory. This prevented the network from further prediction and evaluation.



As such, the input data were changed to gray scale but with the same resolution as mentioned above. The result was 99.53% accuracy which is well within the margin of error and with a similar total running time of 22 minutes, 53 seconds. The error, however, did not reappear.



Taking another step, the input data were reduced to 456x361 gray scale. This results in around 98.60% accuracy which is, again, within the margin of error but with half the total running of only 10 minutes and 43 seconds. Again, the GPU error did not reappear in this case.



Of course, with the results that were represented so far, and with a stronger hardware, 682x540 RGB scale may be preferable with its 100% accuracy so far (assuming that no GPU error are presented). However, this experiment will use 456x361 gray scale resolution for further evaluation because it introduces no GPU error, has almost the same accuracy but with only half the running time ( which could increase drastically if the dataset is getting bigger).

Testing

...