EGU2020-9232

Supporting risk management in the Caribbean by application of Deep Learning for object classification of aerial imagery with MATLAB

Sebastian Bomberg (The MathWorks GmbH, Aachen, Germany)
Neha Goel (The MathWorks Inc., Natick (MA), USA)

This document was exported directly from MATLAB. The original file type, a so-called Live Script, is an executable *script which combines formatted text and equations with code, the computed results and visualizations. Live Scripts may be enriched with clickable control elements such as drop down menus and Live Editor tasks that serve a more complex purpose such as detrending and smoothing data. In that way, Live Scripts are ideal for both exploratory analyses and the presentation of findings in the form of an interactive narrative. Exporting Live Scripts to various formats (e.g. PDF) makes sharing easy also with non MATLAB users.*

1 Introduction

Many regions in the Caribbean often face earthquakes, hurricanes and floods. Especially in poor and informal settlements, where buildings are not up to modern construction standards, these natural hazards can have a devastating effect. One of the main risk factors of a building is the construction material of its roof. Light material may fly off during a hurricane and no longer shelter inhabitants from wind, rain and flying debris. Heavy material itself may pose a threat if roofs collapse during an earthquake.

In order to better prepare for disaster, buildings may be retrofit. However, identification of high-risk buildings requires time-consuming and costly onsite assessment of building conditions by engineers. To make retrofit decisions more quickly and less costly, the use of aerial imagery captured by drones together with artificial intelligence algorithms (AI) for image recognition is proposed to automatically narrow down the number of buildings that require onsite inspection.

To proof that image recognition techniques applied to drone imagery can speed up identifying relevant buildings, MathWorks sponsored the data science competition "Open AI Caribbean Challenge: Mapping Disaster Risk from Aerial Imagery" run by DrivenData in December 2019. The objective was to design a machine learning model that is able to most accurately predict roof construction material from drone imagery. A set of overhead imagery with labeled building footprints was provided by NPO WeRobotics and the Global Progam for Resilient Housing of the World Bank. MathWorks provided participants with complimentary software licenses as well as help and support developing their AI models. For instance, MATLAB code to design and train a benchmark model was published that is discussed in the following sections.

2 Image preprocessing

We are provided with a folder `stac` containing the dataset consisting of seven large, high-resolution Cloud Optimized GeoTiffs of aerial images of regions in Colombia, St. Lucia, and Guatemala captured by drones. The spatial resolution of the images is roughly 4cm. Additionally, the dataset contains GeoJSON files including the building footprint, unique building ID, and roof material labels (for the training data).

The size of the dataset is around 30GB in total. Within it there are three folders, one for each of the countries Colombia, St. Lucia and Guatemala. The folder of each country contains subfolders of areas/regions. For instance, within the folder "Colombia" we have two subfolders for the regions named "borde_rural" and "borde_soacha". The folder of each region contains:

- a BigTIFF image file of the region – for example, `borde_rural_ortho-cog.tif`
- GeoJSON files with metadata on the image extent in latitude/longitude, training data, and test data.

2.1 Accessing data as bigimage

We add the path of the top level data folder `stac` and an create a string array of two region names we are training on. This example is restricted to two regions only. To perform training on all the regions, add the names of all the regions to the array.

```
% Change to whichever folder on your machine contains the dataset.
addpath(genpath("stac"));
regionNames = ["borde_rural" "borde_soacha"];
```

`bigimage` is a function provided by Image Processing Toolbox introduced in MATLAB R2019b for processing very large images that may not fit in memory. Here, we create `bigimage` objects for the BigTIFF image of each region.

```
for idx = 1:numel(regionNames)
    bimg = bigimage(which(regionNames(idx) + "_ortho-cog.tif"));
```

2.2 Separating image channels into RGB and mask

Once we have the `bigimage` we notice that the image has four channels – three channels of RGB and a fourth mask channel of opacity. With the use of the helper function `separateChannels` we remove the opacity mask channel. For further training we will use the three RGB channels only.

NOTE: This will take some time to process. Set the `UseParallel` flag to `true` to speed things up by starting a parallel pool that uses multiple CPU cores.

```
    brgb(idx) = apply(bimg,1,@separateChannels,'UseParallel',true);
```

2.3 Setting spatial reference for bigimage

Since the image of each region spans a rectangular section with a particular latitude and longitude extent, we want to assign this as the spatial reference for the image. This will allow us to extract image regions by using the latitude and longitude values rather than the pixel values, which we will need to do later on. For more information, refer to the example "set spatial referencing for big images" in the documentation.

Parse the GeoJSON files providing the bounding box of the entire region.

```
    fid = fopen(regionNames(idx) + "-imagery.json");
    imageryStructs(idx) = jsondecode(fread(fid,inf,'*char')');
    fclose(fid);
```

Use the bounding boxes parsed out to set the X and Y world limits to the longitude and latitude extents.

```
    for k = 1:numel(brgb(idx).SpatialReferencing)
        % Longitude limits
        brgb(idx).SpatialReferencing(k).XWorldLimits = ...
            [imageryStructs(idx).bbox(1) imageryStructs(idx).bbox(3)];

        % Latitude limits
        brgb(idx).SpatialReferencing(k).YWorldLimits = ...
            [imageryStructs(idx).bbox(2) imageryStructs(idx).bbox(4)];
    end
end
clear bimg
```

3 Preparing image data and labels for training

3.1 Creating training data

The training set consists of three pieces of information that can be parsed from the GeoJSON files for each region:

1. the building ID
2. the building polygon coordinates (in latitude-longitude points)
3. the building material.

To extract the training set, we open the GeoJSON file of each region, read it and decode the files using the `jsondecode` function.

```
for idx = 1:numel(regionNames)
    fid = fopen("train-" + regionNames(idx) + ".geojson");
    trainingStructs(idx) = jsondecode(fread(fid,inf,'*char')');
    fclose(fid);
end
```

Once we have all the values in the `trainingStructs` array we will concatenate all the structures together and get a total number of training set elements.

```
numTrainRegions = arrayfun(@(x)sum(length(x.features)),trainingStructs);
numTrainRegionsCumulative = cumsum(numTrainRegions);
numTrain = sum(numTrainRegions);
trainingStruct = cat(1,trainingStructs.features);
```

Next, we create placeholder arrays for the ID, material, and coordinates.

```
trainID = cell(numTrain,1);          % Training data ID
trainMaterial = cell(numTrain,1);    % Training data material
trainCoords = cell(numTrain,1);      % Training data coordinates
```

Loop through all training data elements.

```
regionIdx = 1;
for k = 1:numTrain
```

Extract the ID, material, and coordinates of each ROI.

```
    trainID{k} = trainingStruct(k).id;
    trainMaterial{k} = trainingStruct(k).properties.roof_material;
    coords = trainingStruct(k).geometry.coordinates;
    if iscell(coords)
        coords = coords{1};
    end
    trainCoords{k} = squeeze(coords);
```

Increment the index of regions as we loop through the training set to ensure we are referring to the correct region.

```
    if k > numTrainRegionsCumulative(regionIdx)
        regionIdx = regionIdx+1;
    end
```

Correct for coordinate convention by flipping the Y image coordinates of the building region coordinates.

```
    trainCoords{k}(:,2) = ...
        brgb(regionIdx).SpatialReferencing(1).YWorldLimits(2) - ...
        (trainCoords{k}(:,2) -
brgb(regionIdx).SpatialReferencing(1).YWorldLimits(1));
 end
```

Convert the text array of materials to a categorical array for later classification.

```
trainMaterial = categorical(trainMaterial);
```

Clear the training data structures since they have now been parsed into individual arrays.

```
clear trainingStruct trainingStructs
```

3.2 Visualizing training data

First, we visualize a specific region and annotate all the training samples as regions of interest (ROIs). Generating the thousands of polygons and displaying them along with a large image is a graphics and computation intensive process, so it will take some time to run this section.

To execute the following steps, change the `display` flag value to `true`.

```
display = false;
```

For more information, refer to the example "extract training samples from big image" in the documentation.

```
displayRegion = "borde_rural";
displayRegionNum = find(regionNames==displayRegion);

if  display
    % Find the indices of the overall training structure that correspond to
    % the selected region
    if displayRegionNum == 1
        polyIndices = 1:numTrainRegions(displayRegionNum);
    else
        polyIndices = numTrainRegions(displayRegionNum-1) + ...
            1:numTrainRegions(displayRegionNum);
    end

    % Extract the ROI polygons
    polyFcn = @(position)images.roi.Polygon('Position',position);
    polys = cellfun(polyFcn,trainCoords(polyIndices));

    % Display the image with ROIs and label the plot
    figure
    bigimageshow(brgb(displayRegionNum))
    xlabel('Longitude')
    ylabel('Latitude')
    set(polys,'Visible','on')
    set(polys,'Parent',gca)
    set(polys,'Color','r')
```

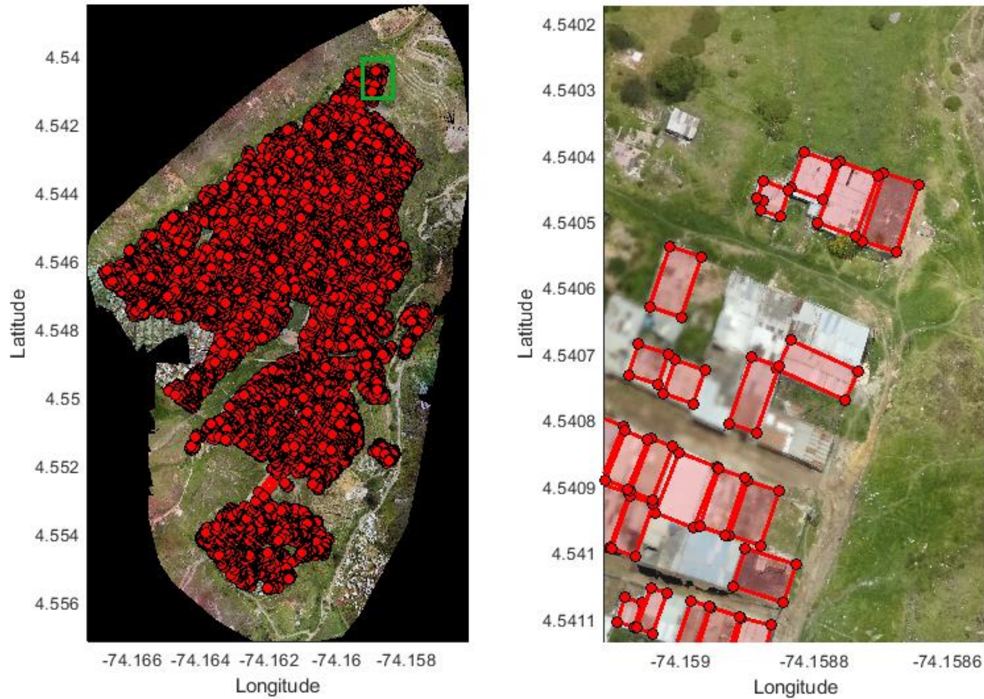

Fig.1: Image of one region with training set ROIs (left) and zoomed-in image region in the highlighted region on the left image (right).

In the following code, we extract a few ROIs from the training set at random to verify that the roof regions have been extracted correctly.

```
    figure
    displayIndices = randi(numTrainRegions(displayRegionNum),4,1);
    for k = 1:numel(displayIndices)
        coords = trainCoords{displayIndices(k) + polyIndices(1) - 1};
        regionImg = getRegion(brgb(displayRegionNum),1, ...
            [min(coords(:,1)) min(coords(:,2))],[max(coords(:,1))
max(coords(:,2))]);
        subplot(2,2,k)
        imshow(regionImg);
    end
end
```

Fig. 2: Sample ROIs extracted from the training set.

3.3 Storing training data

As all the training data is prepared properly, we extract each building to a separate small image file and place it in a `training_data` folder with subfolders for each material. This restructuring simplifies setting up the training process of a classification model.

There are also ways to use `datastore` in order not to save all the images separately to disk, thus reducing the overhead of generating the training set up front. Instead, a datastore can read chunks from the image file of each region only as needed during training. However, this approach will likely be slower during training time.

If the `training_data` folder already exists, skip this step and simply load the saved training data.

```
if exist("training_data","dir")

    load(fullfile("training_data","training_data"));
```

Else, create a new folder with all the training data, including subfolders for each material label.

NOTE: If changing the training set (e.g. changing the number of regions), we recommend deleting any existing `training_data` folder to force the folder to be recreated.

```
else
    mkdir("training_data")
    cd training_data

    materialCategories = categories(trainMaterial);

    for k = 1:numel(materialCategories)
        mkdir(materialCategories{k})
    end
    cd ..
```

Extract training images based on the ROIs and save each image to its corresponding material subfolder (*NOTE: This will take some time*).

```
    regionIdx = 1;

    for k = 1:numTrain
```

Increment the index of regions as we loop through the training set to ensure we are referring to the correct image file when extracting regions.

```
        if k > numTrainRegionsCumulative(regionIdx)
            regionIdx = regionIdx + 1;
        end
```

In this step, we simply use the lower left and upper right coordinates of the building polygon to segment a rectangular region from the image to extract individual buildings for training.

```
        coords = trainCoords{k};

        regionImg = getRegion(brgb(regionIdx),1, ...
            [min(coords(:,1)) min(coords(:,2))], ...
            [max(coords(:,1)) max(coords(:,2))]);

        imgFilename = fullfile("training_data", ...
            string(trainMaterial(k)), ...
            trainID{k} + ".png");

        imwrite(regionImg,imgFilename);
    end
```

Save the `trainID`, `trainMaterial`, and `trainCoords` variables to a MAT-file to refer to them later without regenerating all the training data.

```
    save(fullfile("training_data","training_data"), ...
        "trainID","trainMaterial","trainCoords")
end
```

3.4 Creating and storing test data

Just as we did with the training data, we now parse the test data GeoJSON files and save the data and images to a `test_data` folder.

The test set consists of two pieces of information that can be parsed from the GeoJSON files for each region:

1. The building ID
2. The building polygon coordinates (in latitude-longitude points)

If the folder already exists, skip this step and simply load the saved test data.

```
if exist("test_data","dir")
    load(fullfile("test_data","test_data"));
```

Else, create a new folder with all the test data.

NOTE: If changing the test set (e.g. changing the number of regions), we recommend deleting any existing `test_data` folder to force the folder to be recreated.

```
else
    mkdir("test_data")
```

First, we set up `bigimage` variables for all regions with test labels (which are all except "Castries" and "Gros Islet").

```
    regionNames = ["borde_rural" "borde_soacha" ...
        "mixco_1_and_ebenezer" "mixco_3" "dennery"];

    for idx = 1:numel(regionNames)

        bimg = bigimage(which(regionNames(idx) + "_ortho-cog.tif"));
        brgb(idx) = apply(bimg,1, @separateChannels,'UseParallel',true);

        fid = fopen(regionNames(idx) + "-imagery.json");
        imageryStructs(idx) = jsondecode(fread(fid,inf,'*char')');
        fclose(fid);

        for k = 1:numel(brgb(idx).SpatialReferencing)

            % Longitude limits
```

```
            brgb(idx).SpatialReferencing(k).XWorldLimits = ...
                [imageryStructs(idx).bbox(1) imageryStructs(idx).bbox(3)];

            % Latitude limits
            brgb(idx).SpatialReferencing(k).YWorldLimits = ...
                [imageryStructs(idx).bbox(2) imageryStructs(idx).bbox(4)];
        end
    end

    clear bimg
```

Next, we are open the GeoJSON file of each region with test labels, read it and decode the files using the `jsondecode` function.

```
    for idx = 1:numel(regionNames)

        fid = fopen("test-" + regionNames(idx) + ".geojson");
        testStructs(idx) = jsondecode(fread(fid,inf,'*char')');
        fclose(fid);

    end
```

Once we have all the values in the `testStructs` array, we concatenate all the structures together and get a total number of test set elements.

```
    numTestRegions = arrayfun(@(x)sum(length(x.features)),testStructs);
    numTestRegionsCumulative = cumsum(numTestRegions);
    numTest = sum(numTestRegions);
    testStruct = cat(1, testStructs.features);
```

Next, we create placeholder arrays for the ID and coordinates.

```
    testID = cell(numTest,1);          % Test data ID
    testCoords = cell(numTest,1);      % Test data coordinates
```

Loop through all test data elements.

```
    regionIdx = 1;
    for k = 1:numTest
```

Extract the ID and coordinates of each ROI.

```
        testID{k} = testStruct(k).id;
        coords = testStruct(k).geometry.coordinates;
        if iscell(coords)
            coords = coords{1};
```

```
        end
        testCoords{k} = squeeze(coords);
```

Increment the index of regions as we loop through the training set to ensure we refer to the correct region.

```
        if k > numTestRegionsCumulative(regionIdx)
            regionIdx = regionIdx + 1;
        end
```

Correct for coordinate convention by flipping the Y image coordinates of the building region coordinates.

```
        testCoords{k}(:,2) = 
brgb(regionIdx).SpatialReferencing(1).YWorldLimits(2) - ...
            (testCoords{k}(:,2)-
brgb(regionIdx).SpatialReferencing(1).YWorldLimits(1));
    end
```

Clear the test data structures since they have now been parsed into individual arrays.

```
    clear testStruct testStructs
```

Extract training images based on the ROIs and save each image to its corresponding material subfolder (*NOTE: This will take some time*).

```
    regionIdx = 1;
    for k = 1:numTest
```

Increment the index of regions as we loop through the test set to ensure we refer to the correct image file when extracting regions.

```
        if k > numTestRegionsCumulative(regionIdx)
            regionIdx = regionIdx + 1;
        end
```

In this step, we simply use the lower left and upper right coordinates of the building polygon to segment a rectangular region from the image to extract individual buildings for test.

```
        coords = testCoords{k};
        regionImg = getRegion(brgb(regionIdx),1,[min(coords(:,1))
min(coords(:,2))], ...
            [max(coords(:,1)) max(coords(:,2))]);
        imgFilename = fullfile("test_data",testID{k} + ".png");
        imwrite(regionImg,imgFilename);
    end
```

Save the testID and testCoords variables to a MAT-file to refer to them later without regenerating all the test data.

```
    save(fullfile("test_data","test_data"),"testID","testCoords")
end
```

3.5 Managing training data with datastore

First, we create an imageDatastore for the training_data folder. This datastore is used to manage a collection of image files, where each individual image fits into memory, but the entire collection of images does not necessarily fit.

Image augmentation helps to prevent overfitting of a neural network if only a small amount of labeld training data is available. To further augment and preprocess the data images we recommend looking at the following resources:

- Preprocess images for deep learning
- Augment images for deep learning workflows using Image Processing Toolbox

```
imds = imageDatastore("training_data","IncludeSubfolders",true, ...
    "FileExtensions",".png","LabelSource","foldernames")
```

```
imds = 
  ImageDatastore with properties:

                       Files: {
                              'C:\Users\scastro\Desktop\training_data\concrete_cement\7a1c66f6.png';
                              'C:\Users\scastro\Desktop\training_data\concrete_cement\7a1c6d7c.png';
                              'C:\Users\scastro\Desktop\training_data\concrete_cement\7a1c7646.png'
                               ... and 10353 more
                              }
                      Labels: [concrete_cement; concrete_cement; concrete_cement ... and 10353 more categorical]
    AlternateFileSystemRoots: {}
                    ReadSize: 1
                     ReadFcn: @readDatastoreImage
```

Explore the distribution of all five materials among sample buildings. Notice that the number of samples for each material can be quite different, which means the classes are not balanced. This could affect the performance of the model if not addressed properly, since this may bias the model to predict materials that are more frequent in the training set.

```
labelInfo = countEachLabel(imds)
```

```
labelInfo = 5×2 table
```

Table 1: Identified labels and the frequency of the respective roof material classes.

	Label	Count
1	concrete_cement	497
2	healthy_metal	5544
3	Incomplete	660
4	irregular_metal	3632
5	Other	23

4 Training a neural network using transfer learning

Transfer learning is commonly used in deep learning applications: Taking a pretrained network and using it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. Learned features can be quickly transferred to a new task using a smaller number of training images.

4.1 Configuring a pretrained network for transfer learning

In this example, we use the ResNet-18 neural network as a baseline for our classifier.

NOTE: On the first run, the Deep Learning Toolbox Model for ResNet-18 Network support package needs to be downloaded from

```
net = resnet18;
```

To retrain ResNet-18 to classify new images, we replace the last fully connected layer and the final classification layer of the network. In ResNet-18, these layers have the names `'fc1000'` and `'ClassificationLayer_predictions'`, respectively. We set the new fully connected layer to have the same size as the number of classes in the new data set. To learn faster in the new layers than in the transferred layers, we increase the learning rate factors of the fully connected layer using the `'WeightLearnRateFactor'` and `'BiasLearnRateFactor'` properties.

```
numClasses = numel(categories(imds.Labels));
lgraph = layerGraph(net);

newFCLayer = fullyConnectedLayer(numClasses,'Name','new_fc', ...
    'WeightLearnRateFactor',10,'BiasLearnRateFactor',10);

lgraph = replaceLayer(lgraph,'fc1000',newFCLayer);

newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassLayer);
```

We view the modified network using the `analyzeNetwork` function. To visialize and interactively modify the network architecture, we can also open it using the Deep Network Designer app.

```
analyzeNetwork(lgraph);
```

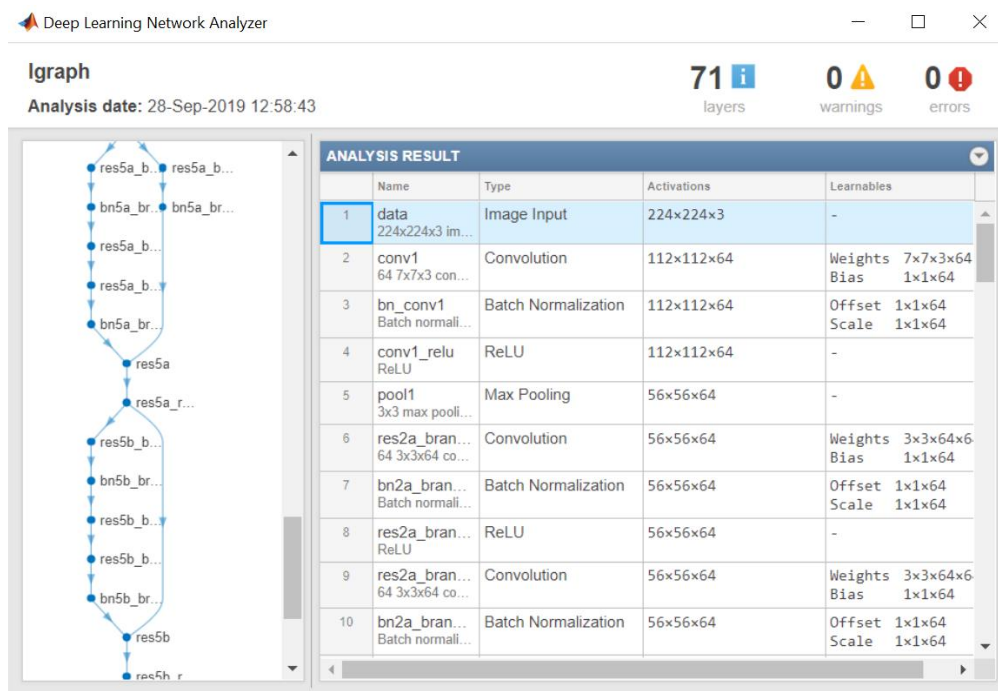

Fig. 3: Visualization of the architecture of ResNet-18 neural network.

4.2 Setting up training options

We configure the image datastore to use the input image size required by the neural network. To do this, we register a custom function called `readAndResize` (which can be found at the end of this script) and set it as the `ReadFcn` of the datastore.

```
inputSize = net.Layers(1).InputSize;

% Refers to a helper function at the end of this script.
imds.ReadFcn = @(im)readAndResize(im,inputSize);
```

We split the training data into training and validation sets. Note that this is randomly selecting a split. Further options to ensure that classes are balanced can be found in the function`splitEachLabel`.

```
[imdsTrain,imdsVal] = splitEachLabel(imds,0.7,"randomized");
```

Next, we specify the training options, including mini-batch size and validation data. Setting `InitialLearnRate` to a small value slows down learning in the transferred layers. In the previous section, we increased the learning rate factors for the fully connected layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers.

Refer to the documentation for `trainingOptions` for different options to improve the training.

```
% Validation frequency
fVal = floor(numel(imdsTrain.Files)/(32*2));

options = trainingOptions('sgdm', ...
    'MiniBatchSize',32, ...
    'MaxEpochs',5, ...
    'InitialLearnRate',1e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsVal, ...
    'ValidationFrequency',fVal, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

4.3 Training a neural network

Here, we use the image datastores, neural network layer graph, and training options to train your model. Note that training will take a long time using a CPU. However, MATLAB will automatically detect if a supported GPU is available to automatically accelerate training.

Set the `doTraining` flag below to `false` to load a presaved network instead for demonstration purposes.

```
doTraining = false;

if doTraining
    netTransfer = trainNetwork(imdsTrain,lgraph,options);
else
    load resnet_presaved.mat
end
```

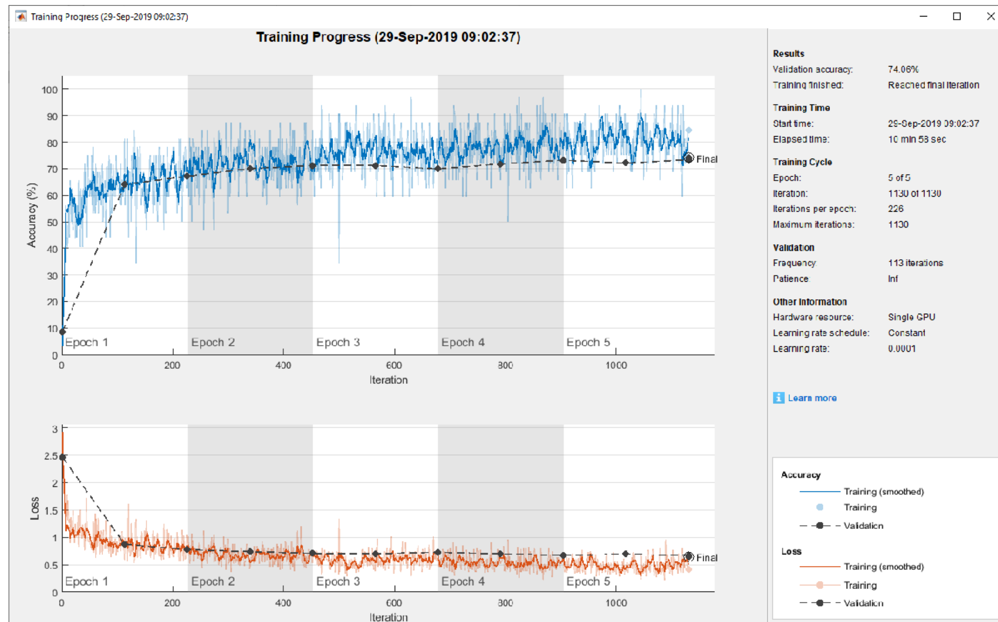

Fig. 4: Accuracy and loss during training process.

5 Predicting and sharing results

5.1 Making predictions on test set

Once we have our trained network, we can perform predictions on our test set. To do so, first, we create an image datastore for the test set.

```
imdsTest = imageDatastore("test_data","FileExtensions",".png");
imdsTest.ReadFcn = @(im)readAndResize(im,inputSize)
```

```
imdsTest = 
  ImageDatastore with properties:

                       Files: {
                              'C:\Users\scastro\Desktop\test_data\7a44da50.png';
                              'C:\Users\scastro\Desktop\test_data\7a44db72.png';
                              'C:\Users\scastro\Desktop\test_data\7a44dc08.png'
                               ... and 7322 more
                              }
    AlternateFileSystemRoots: {}
                    ReadSize: 1
                      Labels: {}
                     ReadFcn: @(im)readAndResize(im,inputSize)
```

Next, we predict labels (`testMaterial`) and scores (`testScores`) using the trained network.

NOTE: This will take some time, but just as with training the network, MATLAB will determine whether a supported GPU is available and significantly speed up this process automatically.

```
[testMaterial,testScores] = classify(netTransfer,imdsTest)
```

```
testMaterial = 7325×1 categorical
healthy_metal
healthy_metal
incomplete
irregular_metal
healthy_metal
incomplete
incomplete
irregular_metal
healthy_metal
incomplete
      ⋮
testScores = 7325×5 single matrix
    0.0044    0.8997    0.0012    0.0930    0.0017
    0.0001    0.9950    0.0000    0.0047    0.0001
    0.0565    0.0167    0.4824    0.4385    0.0058
    0.0007    0.3847    0.0025    0.6119    0.0001
    0.0001    0.8488    0.0017    0.1490    0.0003
    0.0029    0.0953    0.4688    0.4307    0.0024
    0.1648    0.0559    0.3960    0.3809    0.0024
    0.0055    0.0132    0.2002    0.7792    0.0018
    0.0015    0.7648    0.0099    0.2224    0.0015
    0.4453    0.0007    0.5483    0.0051    0.0006
      ⋮
```

The following code displays the predicted materials for a few test images.

```
figure
displayIndices = randi(numTest,4,1);
for k = 1:numel(displayIndices)
    testImg = readimage(imdsTest,displayIndices(k));
    subplot(2,2,k)
    imshow(testImg);
    title(string(testMaterial(displayIndices(k))),"Interpreter","none")
end
```

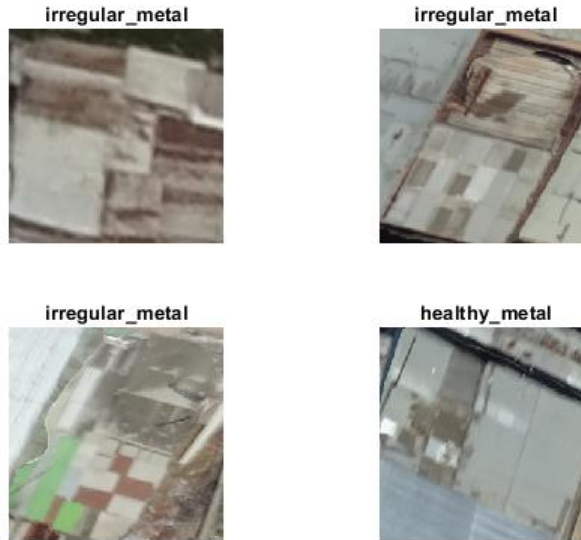

Fig. 5: Predicted roof construction material classes displayed along with images of the classified roof for a few test images.

5.2 Exporting predictions to a file

Create a table of the results based on the IDs and prediction scores. The desired format is:

`id, concrete_cement, healthy_metal, incomplete, irregular_metal, other`

We place all the test results in a MATLAB table, which simplifies visualization and export to the desired file format.

```
testResults = table(testID,testScores(:,1),testScores(:,2), ...
    testScores(:,3),testScores(:,4),testScores(:,5), ...
    'VariableNames',['id';categories(testMaterial)])
```

testResults = 7325×6 table

	id	concrete_cement	healthy_metal	incomplete	irregular_metal	...
1	'7a4d630a'	0.0044	0.8997	0.0012	0.0930	
2	'7a4bbbd6'	0.0001	0.9950	0.0000	0.0047	
3	'7a4ac744'	0.0565	0.0167	0.4824	0.4385	
4	'7a4881fa'	0.0007	0.3847	0.0025	0.6119	
5	'7a4aa4a8'	0.0001	0.8488	0.0017	0.1490	
6	'7a514434'	0.0029	0.0953	0.4688	0.4307	
7	'7a485f72'	0.1648	0.0559	0.3960	0.3809	
8	'7a4b8d32'	0.0055	0.0132	0.2002	0.7792	

	id	concrete_cement	healthy_metal	incomplete	irregular_metal	...
9	'7a47eb3c'	0.0015	0.7648	0.0099	0.2224	
10	'7a4be3ae'	0.4453	0.0007	0.5483	0.0051	
11	'7a46a330'	0.0000	0.9991	0.0000	0.0009	
12	'7a481620'	0.4060	0.1086	0.1759	0.3084	
13	'7a49c678'	0.0004	0.9917	0.0008	0.0064	
14	'7a4ea044'	0.1865	0.2137	0.4766	0.0862	
⋮						

Finally, we write the results to a CSV file.

```
writetable(testResults,'testResults.csv');
```

6 Resources on Deep Learning

The Deep Learning model discussed here served as a simple proof of concept – the baseline or benchmark model. Further adjustment to the model architecture and hyperparameters, such as the various training options, are considered to significantly improve the performance of the model in terms of accuracy. For further information on how MATLAB and Deep Learning are used in the field of geosciences, refer to the following resources:

Live webinar:

- Deep Learning for Geosciences with MATLAB made easy (13 May, 2020, 10:30 – 12:00 CEST)

Web resources:

- MATLAB for earth, ocean and atmospheric sciences
- MATLAB for Deep Learning

Free, browser-based, hands-on trainings:

- MATLAB Onramp
- Deep Learning Onramp

Appendix: Helper functions

Separates a bigimage into its RGB (first to third) and mask (fourth) channels.

```
function [rgb, m] = separateChannels(rgbm)
    rgb = rgbm(:,:,1:3);
    m = logical(rgbm(:,:,4));
end
```

Read and resize an image to a desired input size.

```
function im = readAndResize(filename,sz)
    im = imresize(imread(filename),sz(1:2));
end
```

Copyright 2020 The MathWorks, Inc.