Graph Convolutional Network

For Node Classification Application

Outline

- About Graph Convolutional Network
- Node Classification Using Graph Convolutional Network

About Graph Convolutional Network

Many important real-world datasets come in the form of graphs or networks: social networks, knowledge graphs, protein-interaction networks, the World Wide Web, etc. (just to name a few). Yet, until recently, very little attention has been devoted to the generalization of neural network models to such structured datasets.

About Graph Convolutional Network

Currently, most graph neural network models have a somewhat universal architecture in common. For these models, the goal is then to learn a function of signals/features on a graph G=(V,E) which takes as input:

- ✓A feature description x_i for every node i; summarized in a N×D feature matrix X (N: number of nodes, D: number of input features)
- ✓A representative description of the graph structure in matrix form; typically in the form of an adjacency matrix A

and produces a node-level output Z (an N×F feature matrix, where F is the number of output features per node).

About Graph Convolutional Network

Every neural network layer can then be written as a non-linear function

$$H^{(l+1)}=f(H^{(l)},A),$$

with $H^{(0)}=X$ and $H^{(L)}=Z$, L being the number of layers. The specific models then differ only in how $f(\cdot,\cdot)$ is chosen and parameterized.

About Graph Convolutional Network

As an example, let's consider the following very simple form of a layer-wise propagation rule:

$$f(H^{(l)},A)=\sigma(AH^{(l)}W^{(l)}),$$

where $W^{(l)}$ is a weight matrix for the l-th neural network layer and $\sigma(\cdot)$ is a non-linear activation function like the ReLU.

About Graph Convolutional Network

Let us address two limitations of this simple model:

- multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself (unless there are self-loops in the graph). We can "fix" this by enforcing self-loops in the graph: we simply add the identity matrix to A.

About Graph Convolutional Network

- The second major limitation is that A is typically not normalized and therefore the multiplication with A will completely change the scale of the feature vectors. Normalizing A such that all rows sum to one, i.e. $D^{-1}A$, where D is the diagonal node degree matrix, gets rid of this problem. Multiplying with $D^{-1}A$ now corresponds to taking the average of neighboring node features. In practice, dynamics get more interesting when we use a symmetric normalization, i.e. $D^{-1/2}AD^{-1/2}$ (as this no longer amounts to mere averaging of neighboring nodes).

About Graph Convolutional Network

Combining these two tricks, we essentially arrive at the propagation rule introduced in Kipf & Welling (ICLR 2017):

$$Z_{l+1} = \sigma\left(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}Z_l W_l\right)$$

with $\hat{A} = A + I_N$, where I_N is the identity matrix and $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$.

Node Classification Using Graph Convolutional Network

The node classification task is one where an algorithm, in this example, a GCN, has to predict the labels of unlabelled nodes in a graph. In this example, a graph is represented by a molecule. Atoms in the molecule represent nodes in the graph and the chemical bonds between atoms represent edges in the graph. Node labels are the types of atom, for example, Carbon. As such, input to the GCN are molecules and the outputs are predictions of the type of atom of each unlabelled atom in the molecule.

Node Classification Using Graph Convolutional Network

To assign a categorical label to each node of a graph, the GCN models a function $f(X, A)$ on a graph $G = (V, E)$, where *V* denotes the set of nodes and *E* denotes the set of edges, such that $f(X, A)$ takes as input:

•*X*: A feature matrix of dimension $N \times C$, where $N = |V|$ is the number of nodes in *G* and *C* is number of input channels/features per node.

•*A*: An adjacency matrix of dimension $N \times N$ representing *E* and describing the structure of *G*.

and returns an output:

•*Z*: An Embedding or feature matrix of dimension $N \times F$, where *F* is number of output features per node. In other words, *Z* is the predictions of the network and *F* is the number of classes.

Node Classification Using Graph Convolutional Network

The model $f(X, A)$ is based on spectral graph convolution, with weights/filter parameters shared over all locations in *G*. The model can be represented as a layer-wise propagation model, such that the output of layer $l + 1$ is expressed as $Z_{l+1} = \sigma\left(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}Z_l W_l\right)$, where

- σ is an activation function.
- Z_l is the activation matrix of layer *l*, with $Z_1 = X$.
- W_l is the weight matrix of layer *l*.
- $\hat{A} = A + I_N$ is the adjacency matrix of graph *G* with added self-connections. I_N is the identity matrix.
- $\hat{D}$ is the degree matrix of $\hat{A}$.

Node Classification Using Graph Convolutional Network

Expression $\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}$ can be referred to as the *normalized* adjacency matrix of the graph.

The GCN model in this example is a variant of the standard GCN model described above. The variant uses residual connections between layers. The residual connections enable the model to carry over information from previous layer's input. Therefore, the output of layer $l+1$ of the GCN model in this example is

$$Z_{l+1} = \sigma\left(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}Z_l W_l\right) + Z_l$$

QM7 Dataset

This example uses the QM7 dataset, which is a molecular dataset consisting of 7165 molecules composed of up to 23 atoms. That is, the molecule with the highest number of atoms has 23 atoms. Overall, the dataset consists of 5 unique atoms: Carbon, Hydrogen, Nitrogen, Oxygen, and Sulphur.

Download the QM7 dataset from the following URL:

```
dataURL = 'http://quantum-machine.org/data/qm7.mat';
outputFolder = fullfile(".",'qm7Data');
dataFile = fullfile(outputFolder,'qm7.mat');

if ~exist(dataFile, 'file')
    mkdir(outputFolder);
    fprintf('Downloading file ''%s'' ...\n', dataFile);
    websave(dataFile, dataURL);
end
```

Load QM7 data

```
data = load(dataFile)
data = struct with fields:
    X: [7165×23×23 single]
    R: [7165×23×3 single]
    Z: [7165×23 single]
    T: [1×7165 single]
    P: [5×1433 int64]
```

The data consists of five different arrays. This example uses the arrays in fields X and Z of struct data. The array in X represents the Coulomb matrix representation of each molecule, totalling 7165 molecules, and the array in Z represents the atomic charge/number of each atom in the molecules. The adjacency matrices of the graphs representing the molecules, and the feature matrices of the graphs, are extracted from the Coulomb matrices. The categorical array of labels is extracted from the array in Z. For any molecule that does not have up to 23 atoms, contains padded zeros.

Extract and Preprocess Graph Data

To extract graph data, get the Coulomb matrices and atomic numbers. Permute the data representing the Coulomb matrices and change the datatype to double. Sort the data representing the atomic charges so that it matches the data representing the Coulomb matrices.

```
coulombData = double(permute(data.X, [2 3 1]));
atomicNumber = sort(data.Z,2,'descend');
```

Reformat the Coulomb matrix representation of the molecules to binary adjacency matrices using the coloumb2Adjacency function attached to this example as a supporting file.

```
adjacencyData = coloumb2Adjacency(coulombData, atomicNumber);
```

Extract and Preprocess Graph Data

Note that the coloumb2Adjacency function does not remove padded zeros from the data. They are left intentionally to make it easier to split the data into separate molecules for training, validation and inference. Therefore, ignoring the padded zeros, the adjacency matrix of the graph representing the molecule at index 1, which consists of 5 atoms, is

```
adjacencyData(1:5,1:5,1)
ans = 5×5

     0     1     1     1     1
     1     0     0     0     0
     1     0     0     0     0
     1     0     0     0     0
     1     0     0     0     0
```

Extract and Preprocess Graph Data

Before preprocessing the data, use the splitData function to randomly select and split the data into training, validation and test data. The function uses the ratio 80:10:10 to split the data.

The adjacencyDataSplit output of the splitData function is the adjacencyData input data split into three different arrays. Likewise, the coulombDataSplit and atomicNumberSplit outputs are the coulombData and atomicNumber input data split into three different arrays respectively.

```
[adjacencyDataSplit, coulombDataSplit, atomicNumberSplit] =
splitData(adjacencyData, coulombData, atomicNumber);
```

```
function [adjacencyDataSplit, coulombDataSplit, atomicNumberSplit] =
splitData(adjacencyData, coulombData, atomicNumber)

adjacencyDataSplit = cell(1,3);
coulombDataSplit = cell(1,3);
atomicNumberSplit = cell(1,3);

numMolecules = size(adjacencyData, 3);

% Set initial random state for example reproducibility.
rng(0);

% Get training data
idx = randperm(size(adjacencyData, 3), floor(0.8*numMolecules));
adjacencyDataSplit{1} = adjacencyData(:,:,idx);
coulombDataSplit{1} = coulombData(:,:,idx);
atomicNumberSplit{1} = atomicNumber(idx,:);
adjacencyData(:,:,idx) = [];
coulombData(:,:,idx) = [];
atomicNumber(idx,:) = [];
```

```
% Get validation data
idx = randperm(size(adjacencyData, 3), floor(0.1*numMolecules));
adjacencyDataSplit{2} = adjacencyData(:,:,idx);
coulombDataSplit{2} = coulombData(:,:,idx);
atomicNumberSplit{2} = atomicNumber(idx,:);
adjacencyData(:,:,idx) = [];
coulombData(:,:,idx) = [];
atomicNumber(idx,:) = [];

% Get test data
adjacencyDataSplit{3} = adjacencyData;
coulombDataSplit{3} = coulombData;
atomicNumberSplit{3} = atomicNumber;

end
```

Extract and Preprocess Graph Data

Use the preprocessData function to process the adjacencyDataSplit, coulombDataSplit, and atomicNumberSplit and return the adjacency matrix adjacency, feature matrix features, and categorical array labels.

The preprocessData function builds a sparse block-diagonal matrix of the adjacency matrices of different graph instances, such that, each block in the matrix corresponds to the adjacency matrix of one graph instance. This preprocessing is required because GCN accepts a single adjacency matrix as input, whereas this example deals with multiple graph instances. The function takes the non-zero diagonal elements of the Coulomb matrices and assigns them as features. Therefore, the number of input features per node in the example is 1.

```
[adjacency, features, labels] = cellfun(@preprocessData, adjacencyDataSplit, coulombDataSplit, atomicNumberSplit, 'UniformOutput', false);
```

Extract and Preprocess Graph Data

The preprocessData function preprocesses the input data as follows:
For each graph/molecule

- Remove padded zeros from atomicNumber.
- Concatenate the atomic number data with the atomic number data of other graph instances. It is necessary to concatenate the data since the example deals with multiple graph instances.
- Remove padded zeros from adjacencyData.
- Build a sparse block-diagonal matrix of the adjacency matrices of different graph instances. Each block in the matrix corresponds to the adjacency matrix of one graph instance. This step is also necessary because there are multiple graph instances in the example.
- Extract feature array from coulombData. The feature array is the non-zero diagonal elements of the Coulomb matrix in coulombData.
- Concatenate the feature array with feature arrays of other graph instances.

The function then converts the atomic number data to categorical arrays.

```
function [adjacency, features, labels] = preprocessData(adjacencyData, coulombData,
atomicNumber)

adjacency = sparse([]);
features = [];
labels = [];
for i = 1:size(adjacencyData, 3)
    tmpLabels = nonzeros(atomicNumber(i,:)); % Remove padded zeros from atomicNumber
    labels = [labels; tmpLabels];

    validIdx = 1:numel(tmpLabels); % Get the indices of the un-padded data

    % Use the indices for un-padded data to remove padded zeros from the adjacency data
    tmpAdjacency = adjacencyData(validIdx, validIdx, i);

    % Build the adjacency matrix into a block diagonal matrix
    adjacency = blkdiag(adjacency, tmpAdjacency);

    % Remove padded zeros from coulombData and extract the feature array
    tmpFeatures = diag(coulombData(validIdx, validIdx, i));
    features = [features; tmpFeatures];
end
```

```
% Convert labels to categorical array
atomicNumbers = unique(labels);
atomNames = ["Hydrogen","Carbon","Nitrogen","Oxygen","Sulphur"];
labels = categorical(labels, atomicNumbers, atomNames);

end
```

Extract and Preprocess Graph Data

View the adjacency matrices of the training, validation, and test data.

```
adjacency
adjacency=1×3 cell array
    {88722×88722 double}    {10942×10942 double}    {10986×10986 double}
```

This shows that there are 88722 nodes in the training data, 10942 nodes in the validation data, and 10986 nodes in the test data.

Extract and Preprocess Graph Data

Normalize the feature array using the normalizeFeatures function.

```
features = normalizeFeatures(features);
```

Get the training and the validation data.

```
featureTrain = features{1};
adjacencyTrain = adjacency{1};
targetTrain = labels{1};

featureValidation = features{2};
adjacencyValidation = adjacency{2};
targetValidation = labels{2};
```

Normalize Features Function

```
function features = normalizeFeatures(features)

% Get the mean and variance from the training data
meanFeatures = mean(features{1});
varFeatures = var(features{1}, 1);

% Standardize training, validation and test data
for i = 1:3
    features{i} = (features{i} - meanFeatures)./sqrt(varFeatures);
end

end
```

Visualize Data and Data Statistics

Sample and specify indices of molecules to visualize.

For each specified index

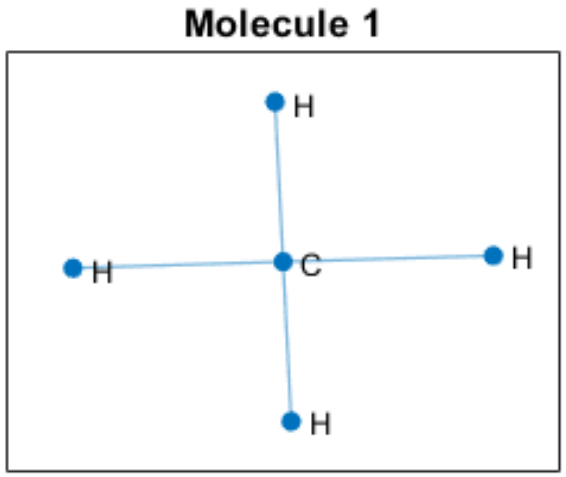
- Remove padded zeros from the data representing unprocessed atomic numbers atomicNumber and unprocessed adjacency matrix adjacencyData of the sampled molecule. The unprocessed data are used here for easy sampling.
- Convert the adjacency matrix to graph using the graph function.
- Convert the atomic numbers to symbols.
- Plot the graph using the atomic symbols as node labels.

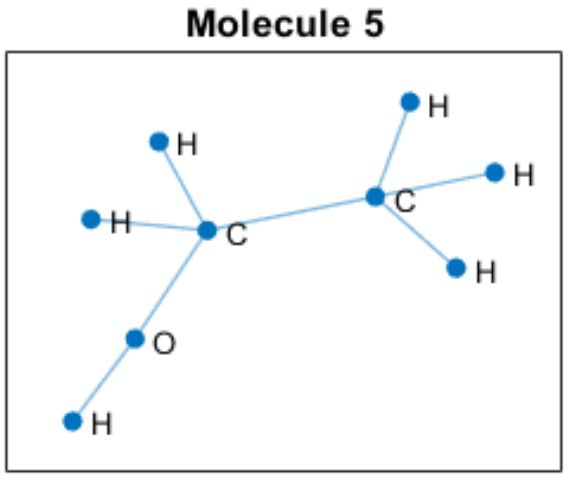
Visualize Data and Data Statistics

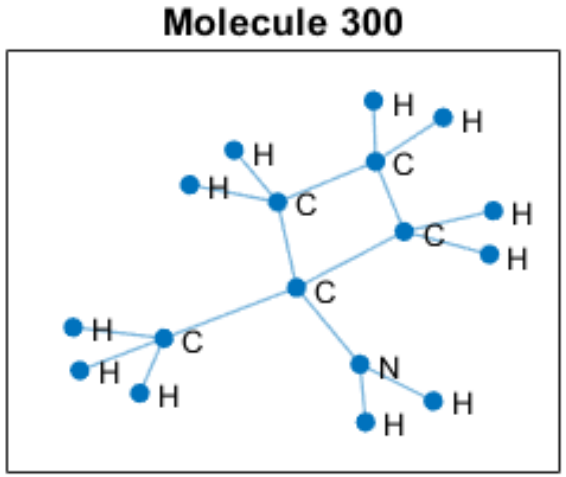
```
idx = [1 5 300 1159];
for j = 1:numel(idx)
    % Remove padded zeros from the data
    atomicNum = nonzeros(atomicNumber(idx(j),:));
    numOfNodes = numel(atomicNum);
    adj = adjacencyData(1:numOfNodes,1:numOfNodes,idx(j));

    % Convert adjacency matrix to graph
    compound = graph(adj);
```

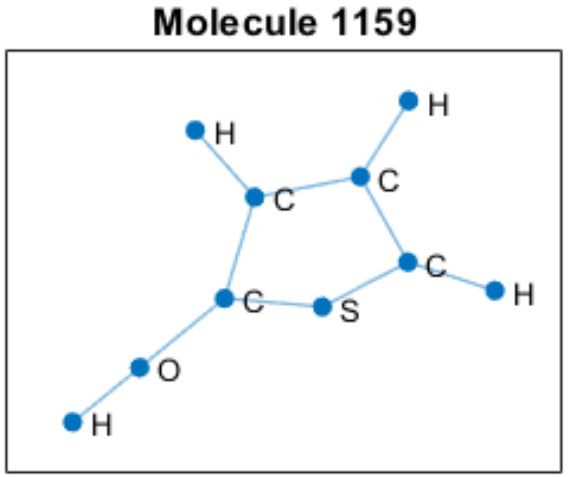
```
% Convert atomic numbers to symbols
    symbols = cell(numOfNodes, 1);
    for i = 1:numOfNodes
        if atomicNum(i) == 1
            symbols{i} = 'H';
        elseif atomicNum(i) == 6
            symbols{i} = 'C';
        elseif atomicNum(i) == 7
            symbols{i} = 'N';
        elseif atomicNum(i) == 8
            symbols{i} = 'O';
        else
            symbols{i} = 'S';
        end
    end

    % Plot graph
    subplot(2,2,j)
    plot(compound, 'NodeLabel', symbols, 'LineWidth', 0.75, ...
    'Layout', 'force')
    title("Molecule " + idx(j))
end
```

Visualize Data and Data Statistics







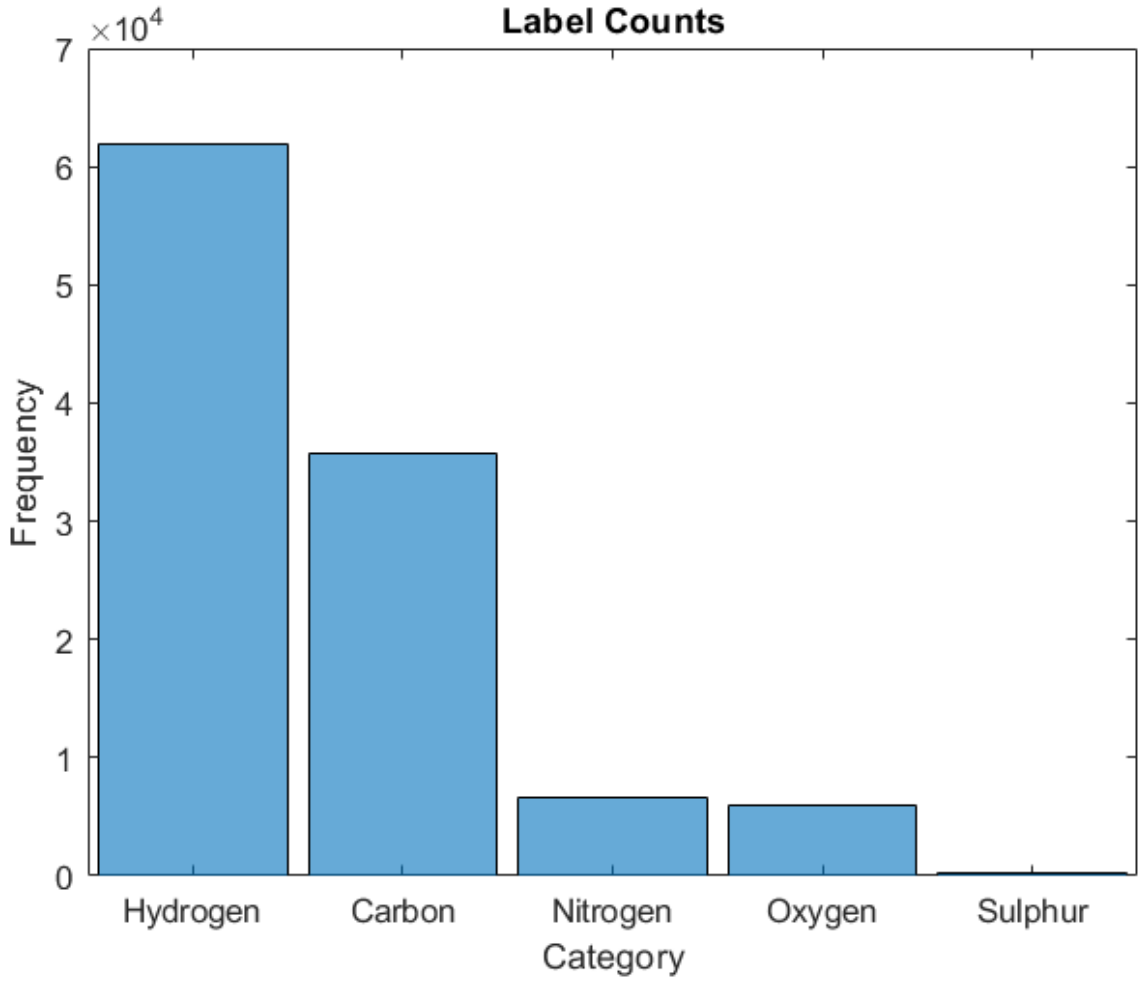


Get all the labels and the classes.

```
labelsAll = cat(1,labels{:});
classes = categories(labelsAll)
classes = 5×1 cell
    {'Hydrogen'}
    {'Carbon'  }
    {'Nitrogen'}
    {'Oxygen'  }
    {'Sulphur' }
```

Visualize frequency of each label category using a histogram.

```
figure
histogram(labelsAll)
xlabel('Category')
ylabel('Frequency')
title('Label Counts')
```



Define Model Function

Create the function model, that takes the feature data dlX, the adjacency matrix A, and the model parameters parameters as input and returns predictions for the label.

```
function dlY = model(dlX, A, parameters)

% Normalize adjacency matrix
L = normalizeAdjacency(A);

Z1 = dlX;

Z2 = L * Z1 * parameters.W1;
Z2 = relu(Z2) + Z1;

Z3 = L * Z2 * parameters.W2;
Z3 = relu(Z3) + Z2;

Z4 = L * Z3 * parameters.W3;
dlY = softmax(Z4, 'DataFormat', 'BC');

end
```

Normalize Ajacency Function

The normalizeAdjacency function calculates and returns the normalized adjacency matrix normAdjacency of the input adjacency matrix adjacency.

```
function normAdjacency = normalizeAdjacency(adjacency)

% Add self connections to adjacency matrix
adjacency = adjacency + speye(size(adjacency));

% Compute degree of nodes
degree = sum(adjacency, 2);

% Compute inverse square root of degree
degreeInvSqrt = sparse(sqrt(1./degree));

% Normalize adjacency matrix
normAdjacency = diag(degreeInvSqrt) * adjacency * diag(degreeInvSqrt);

end
```

Initialize Model Parameters

Set the number of input features per node. This is the column length of the feature matrix.

numInputFeatures = size(featureTrain,2)
numInputFeatures = 1

Set the number of feature maps for the hidden layers.

numHiddenFeatureMaps = 32;

Set the number of output features as the number of categories.

numOutputFeatures = numel(classes)
numOutputFeatures = 5

Initialize Model Parameters

```
sz = [numInputFeatures numHiddenFeatureMaps];
numOut = numHiddenFeatureMaps;
numIn = numInputFeatures;
parameters.W1 = initializeGlorot(sz,numOut,numIn,'double');

sz = [numHiddenFeatureMaps numHiddenFeatureMaps];
numOut = numHiddenFeatureMaps;
numIn = numHiddenFeatureMaps;
parameters.W2 = initializeGlorot(sz,numOut,numIn,'double');

sz = [numHiddenFeatureMaps numOutputFeatures];
numOut = numOutputFeatures;
numIn = numHiddenFeatureMaps;
parameters.W3 = initializeGlorot(sz,numOut,numIn,'double');
```

Define Model Gradients Function

Create the function modelGradients, that takes the feature data dlX, the adjacency matrix adjacencyTrain, the one-hot encoded targets T of the labels, and the model parameters parameters as input and returns the gradients of the loss with respect to the parameters, the corresponding loss, and the network predictions.

```
function [gradients, loss, dlYPred] = modelGradients(dlX, adjacencyTrain, T, parameters)

dlYPred = model(dlX, adjacencyTrain, parameters);

loss = crossentropy(dlYPred, T, 'DataFormat', 'BC');

gradients = dlgradient(loss, parameters);

end
```

Specify Training Options

Train for 1500 epochs and set the learn rate for Adam solver to 0.01.

numEpochs = 1500;
learnRate = 0.01;
Validate the network after every 300 epochs.

validationFrequency = 300;
Visualize the training progress in a plot.

plots = "training-progress";
To train on a GPU if one is available, specify the execution environment "auto".

executionEnvironment = "auto";

Train Model

```
if plots == "training-progress"
    figure

    % Accuracy.
    subplot(2,1,1)
    lineAccuracyTrain = animatedline('Color',[0 0.447 0.741]);
    lineAccuracyValidation = animatedline( ...
        'LineStyle','--', ...
        'Marker','o', ...
        'MarkerFaceColor','black');
    ylim([0 1])
    xlabel("Epoch")
    ylabel("Accuracy")
    grid on

    % Loss.
    subplot(2,1,2)
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    lineLossValidation = animatedline( ...
        'LineStyle','--', ...
        'Marker','o', ...
        'MarkerFaceColor','black');
    ylim([0 inf])
    xlabel("Epoch")
    ylabel("Loss")
    grid on
end
```

Train Model

Initialize parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

Convert training and validation feature data to dlarray.

```
dlX = dlarray(featureTrain);
dlXValidation = dlarray(featureValidation);
```

For GPU training, convert data to gpuArray objects.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlX = gpuArray(dlX);
End
```

Encode training and validation label data using onehotencode.

```
T = onehotencode(targetTrain, 2, 'ClassNames', classes);
TValidation = onehotencode(targetValidation, 2, 'ClassNames', classes);
```

Train Model

Train the model.

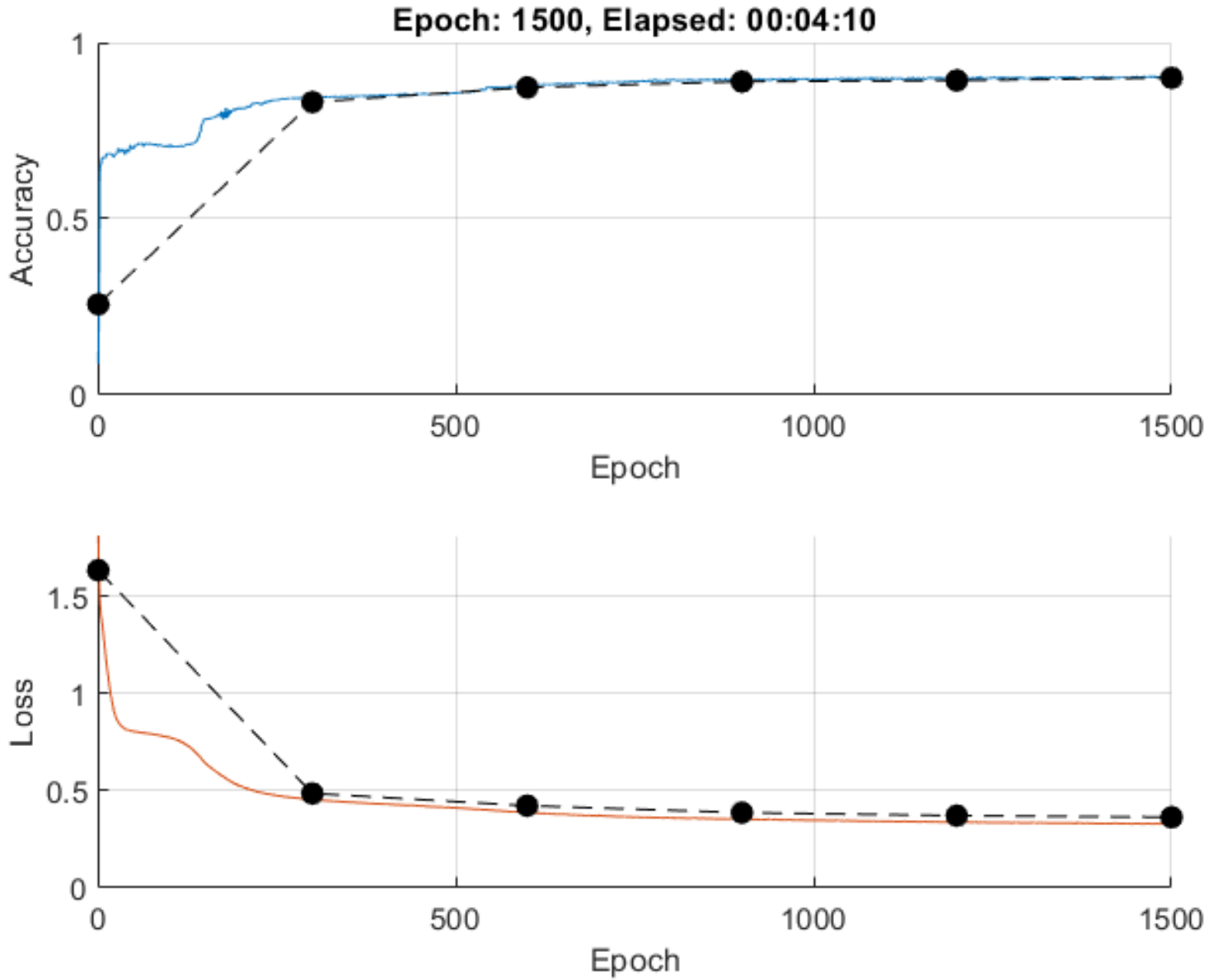
For each epoch

- Evaluate the model gradients and loss using dlfeval and the modelGradients function.
- Update the network parameters using adamupdate.
- Compute the training accuracy score using the accuracy function. The function takes the network predictions, the target containing the labels, and the categories classes as inputs and returns the accuracy score.
- If required, validate the network by making predictions using the model function and computing the validation loss and the validation accuracy score using crossentropy and the accuracy function.
- Update the training plot.

```
start = tic;
% Loop over epochs.
for epoch = 1:numEpochs

    % Evaluate the model gradients and loss using dlfeval and the
    % modelGradients function.
    [gradients, loss, dlYPred] = dlfeval(@modelGradients, dlX, adjacencyTrain, T, parameters);

    % Update the network parameters using the Adam optimizer.
    [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
        trailingAvg,trailingAvgSq,epoch,learnRate);

    % Display the training progress.
    if plots == "training-progress"
        subplot(2,1,1)
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        title("Epoch: " + epoch + ", Elapsed: " + string(D))

        % Loss.
        addpoints(lineLossTrain,epoch,double(gather(extractdata(loss))))
```

```
        % Accuracy score.
        score = accuracy(dlYPred, targetTrain, classes);
        addpoints(lineAccuracyTrain,epoch,double(gather(score)))

        drawnow

        % Display validation metrics.
        if epoch == 1 || mod(epoch,validationFrequency) == 0
            % Loss.
            dlYPredValidation = model(dlXValidation, adjacencyValidation, parameters);
            lossValidation = crossentropy(dlYPredValidation, TValidation, 'DataFormat', 'BC');
            addpoints(lineLossValidation,epoch,double(gather(extractdata(lossValidation))))

            % Accuracy score.
            scoreValidation = accuracy(dlYPredValidation, targetValidation, classes);
            addpoints(lineAccuracyValidation,epoch,double(gather(scoreValidation)))

            drawnow
        end
    end
end
```

Train Model


Epoch: 1500, Elapsed: 00:04:10
Accuracy
1
0.5
0
0
500
1000
1500
Epoch
Loss
1.5
1
0.5
0
0
500
1000
1500
Epoch

Test Model

```
featureTest = features{3};
adjacencyTest = adjacency{3};
targetTest = labels{3};
```

Convert the test feature data to dlarray.

```
dlXTest = dlarray(featureTest);
```

Make predictions on the data.

```
dlYPredTest = model(dlXTest, adjacencyTest, parameters);
```

Test Model

Calculate the accuracy score using the accuracy function. The accuracy function also returns a decoded network predictions predTest as class labels. The network predictions are decoded using onehotdecode.

```
[scoreTest, predTest] = accuracy(dlYPredTest, targetTest, classes);
```

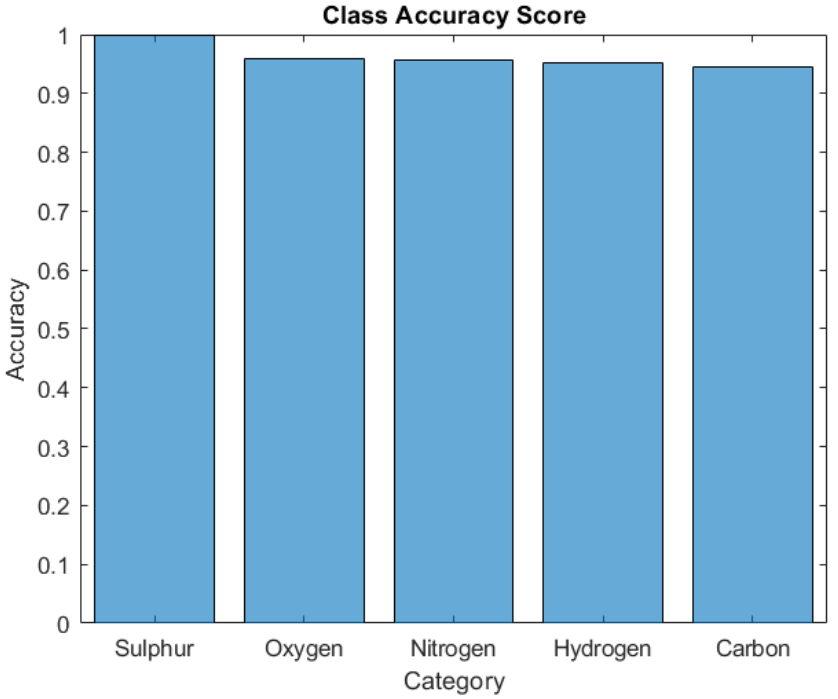
View the accuracy score.

```
scoreTest
```

```
scoreTest = 0.9053
```

Visualize Predictions

```
numOfSamples = numel(targetTest);
classTarget = zeros(numOfSamples, numOutputFeatures);
classPred = zeros(numOfSamples, numOutputFeatures);
for i = 1:numOutputFeatures
    classTarget(:,i) = targetTest==categorical(classes(i));
    classPred(:,i) = predTest==categorical(classes(i));
end

% Compute class-wise accuracy score
classAccuracy = sum(classPred == classTarget)./numOfSamples;

% Visualize class-wise accuracy score
figure
[~,idx] = sort(classAccuracy,'descend');
histogram('Categories',classes(idx), ...
    'BinCounts',classAccuracy(idx), ...
    'Barwidth',0.8)
xlabel("Category")
ylabel("Accuracy")
title("Class Accuracy Score")
```



Visualize Predictions

To visualize how the model makes incorrect predictions and evaluate the model based on class-wise precision and class-wise recall, calculate the confusion matrix using confusionmat and visualize the results using confusionchart.

```
[confusionMatrix, order] = confusionmat(targetTest, predTest);
figure
cm = confusionchart(confusionMatrix, classes, ...
    'ColumnSummary','column-normalized', ...
    'RowSummary','row-normalized', ...
    'Title', 'GCN QM7 Confusion Chart');
```