Deep Learning Models for Text Processing - CNN

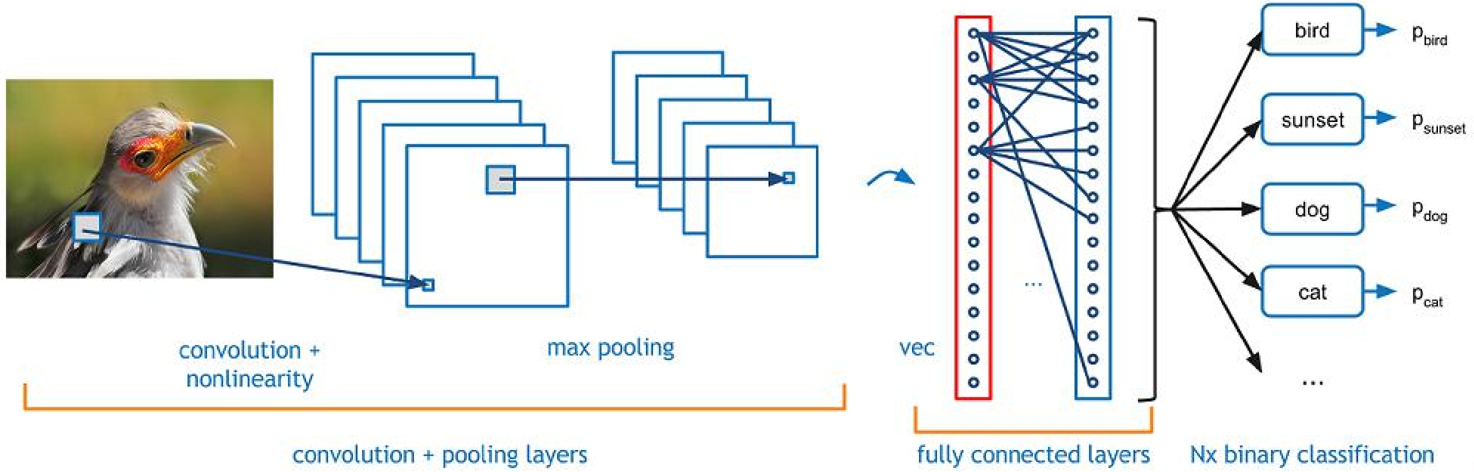
- Classify Text with CNN Models

Outline

- About CNN
- Learn About Convolutional Neural Networks
- Specify Layers of Convolutional Neural Network
- Classify Text Data Using Convolutional Neural Network

CNN

- A convolutional neural network (CNN or ConvNet) is one of the most popular algorithms for deep learning, a type of machine learning in which a model learns to perform classification tasks directly from images, video, text, or sound.



CNN

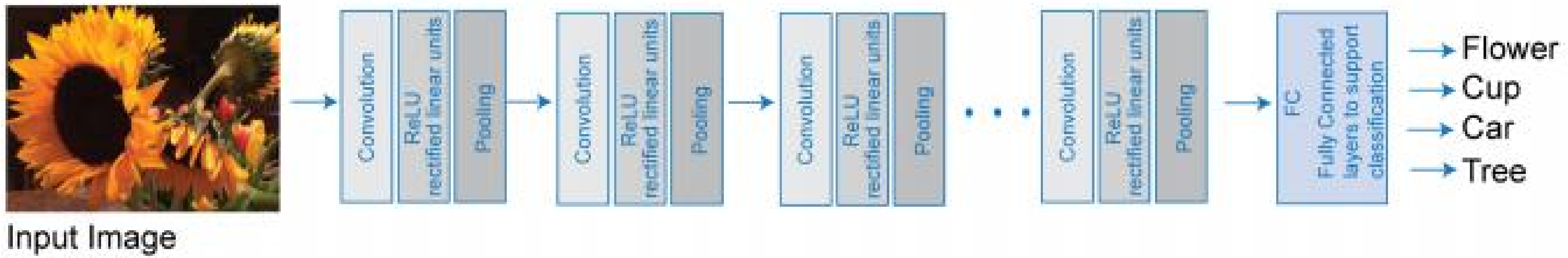
- Convolutional neural networks (ConvNets) are specifically suitable for images as inputs, although they are also used for other applications such as text, signals, and other continuous responses. They differ from other types of neural networks in a few ways:
- Convolutional neural networks are inspired from the biological structure of a visual cortex, which contains arrangements of simple and complex cells. These cells are found to activate based on the subregions of a visual field. These subregions are called receptive fields.

CNN

- Inspired from the findings of this study, the neurons in a convolutional layer connect to the subregions of the layers before that layer instead of being fully-connected as in other types of neural networks. The neurons are unresponsive to the areas outside of these subregions in the image.
- These subregions might overlap, hence the neurons of a ConvNet produce spatially-correlated outcomes, whereas in other types of neural networks, the neurons do not share any connections and produce independent outcomes.

CNN

- In addition, in a neural network with fully-connected neurons, the number of parameters (weights) can increase quickly as the size of the input increases. A convolutional neural network reduces the number of parameters with the reduced number of connections, shared weights, and downsampling.
- CNN consists of convolutional, pooling and fully-connected layers.


Input Image
Convolution
ReLU rectified linear units
Pooling
Convolution
ReLU rectified linear units
Pooling
Convolution
ReLU rectified linear units
Pooling
Convolution
ReLU rectified linear units
Pooling
FC
Fully Connected layers to support classification
Flower
Cup
Car
Tree

CNN

- The neurons in each layer of a ConvNet are arranged in a 3-D manner, transforming a 3-D input to a 3-D output. For example, for an image input, the first layer (input layer) holds the images as 3-D inputs, with the dimensions being height, width, and the color channels of the image. The neurons in the first convolutional layer connect to the regions of these images and transform them into a 3-D output.

CNN

- The hidden units (neurons) in each layer learn nonlinear combinations of the original inputs, which is called feature extraction. These learned features, also known as activations, from one layer become the inputs for the next layer. Finally, the learned features become the inputs to the classifier or the regression function at the end of the network.

CNN

- The architecture of a ConvNet can vary depending on the types and numbers of layers included. The types and number of layers included depends on the particular application or data. For example, if you have categorical responses, you must have a classification function and a classification layer, whereas if your response is continuous, you must have a regression layer at the end of the network.

CNN

- A smaller network with only one or two convolutional layers might be sufficient to learn a small number of gray scale image data. On the other hand, for more complex data with millions of colored images, you might need a more complicated network with multiple convolutional and fully connected layers.

CNN

- You can concatenate the layers of a convolutional neural network in MATLAB in the following way:

```
layers = [imageInputLayer([28 28 1])
 convolution2dLayer(5,20)
 reluLayer
 maxPooling2dLayer(2,'Stride',2)
 fullyConnectedLayer(10)
 softmaxLayer
 classificationLayer];
```

CNN

- After defining the layers of your network, you must specify the training options using the trainingOptions function. For example,

```
options = trainingOptions('sgdm');
```

- Then, you can train the network with your training data using the trainNetwork function. The data, layers, and training options become the inputs to the training function. For example,

```
convnet = trainNetwork(data,layers,options);
```

Specify Layers of Convolutional Neural Network

- The first step of creating and training a new convolutional neural network (ConvNet) is to define the network architecture. The network architecture can vary depending on the types and numbers of layers included. The types and number of layers included depends on the particular application or data. For example, if you have categorical responses, you must have a softmax layer and a classification layer, whereas if your response is continuous, you must have a regression layer at the end of the network.

Specify Layers of Convolutional Neural Network

- A smaller network with only one or two convolutional layers might be sufficient to learn on a small number of grayscale image data. On the other hand, for more complex data with millions of colored images, you might need a more complicated network with multiple convolutional and fully connected layers.
- To specify the architecture of a deep network with all layers connected sequentially, create an array of layers directly. For example, to create a deep network which classifies 28-by-28 grayscale images into 10 classes, specify the layer array

Specify Layers of Convolutional Neural Network

```
layers = [
 imageInputLayer([28 28 1])
 convolution2dLayer(3,16,'Padding',1)
 batchNormalizationLayer
 reluLayer
 maxPooling2dLayer(2,'Stride',2)
 convolution2dLayer(3,32,'Padding',1)
 batchNormalizationLayer
 reluLayer
 fullyConnectedLayer(10)
 softmaxLayer
 classificationLayer];
```

Specify Layers of Convolutional Neural Network

- layers is an array of **Layer** objects. You can then use layers as an input to the training function trainNetwork.
- To specify the architecture of a neural network with all layers connected sequentially, create an array of layers directly. To specify the architecture of a network where layers can have multiple inputs or outputs, use a **LayerGraph** object.

Image Input Layer

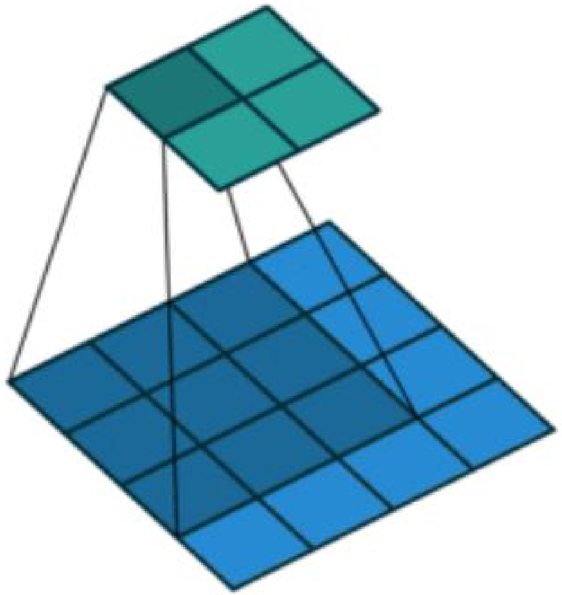
- Create an image input layer using *imageInputLayer*.
- An image input layer inputs images to a network and applies data normalization.
- Specify the image size using the *inputSize* argument. The size of an image corresponds to the height, width, and the number of color channels of that image. For example, for a grayscale image, the number of channels is 1, and for a color image it is 3.

Convolutional Layer

- A 2-D convolutional layer applies sliding convolutional filters to the input. Create a 2-D convolutional layer using *convolution2dLayer*. The convolutional layer consists of various components: **Filters and Stride**
- A convolutional layer consists of neurons that connect to subregions of the input images or the outputs of the previous layer. The layer learns the features localized by these regions while scanning through an image. When creating a layer using the *convolution2dLayer* function, you can specify the size of these regions using the filterSize input argument.

Convolutional Layer

- This image shows a 3-by-3 filter scanning through the input. The lower map represents the input and the upper map represents the output.

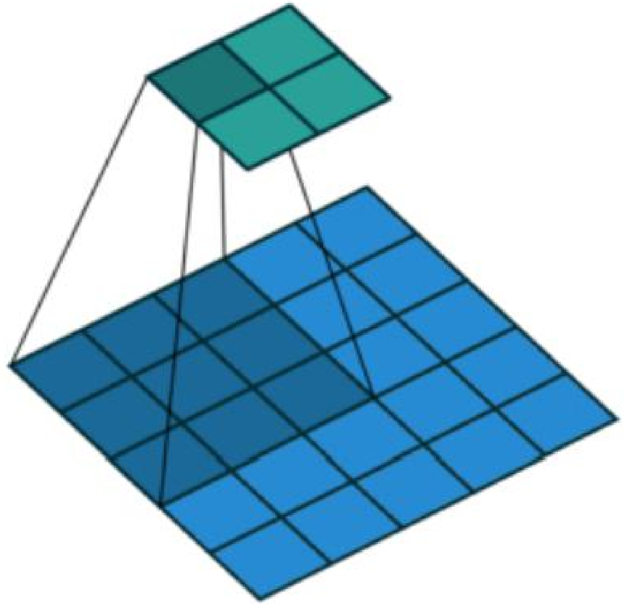


Convolutional Layer

- The step size with which the filter moves is called a *stride*. You can specify the step size with the *Stride* name-value pair argument. The local regions that the neurons connect to can overlap depending on the filterSize and 'Stride' values.

Convolutional Layer

- This image shows a 3-by-3 filter scanning through the input with a stride of 2. The lower map represents the input and the upper map represents the output.



Convolutional Layer

- The number of weights in a filter is h * w * c, where h is the height, and w is the width of the filter, respectively, and c is the number of channels in the input. For example, if the input is a color image, the number of color channels is 3. The number of filters determines the number of channels in the output of a convolutional layer. Specify the number of filters using the numFilters argument with the convolution2dLayer function.

Dilated Convolution

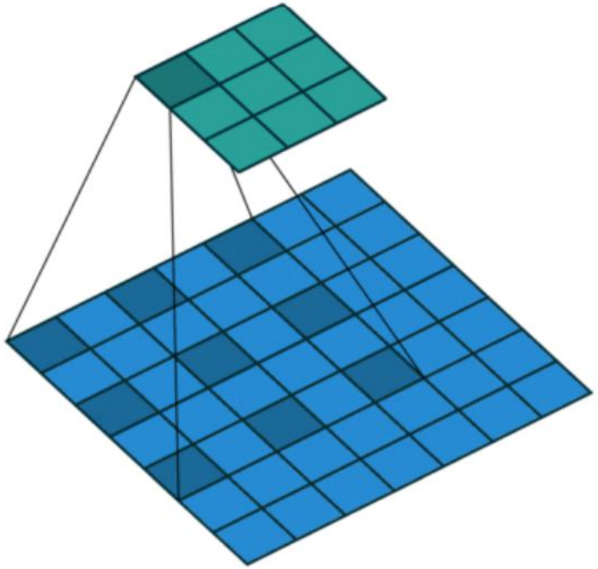
- A dilated convolution is a convolution in which the filters are expanded by spaces inserted between the elements of the filter. Specify the dilation factor using the 'DilationFactor' property.
- Use dilated convolutions to increase the receptive field (the area of the input which the layer can see) of the layer without increasing the number of parameters or computation.

Dilated Convolution

- The layer expands the filters by inserting zeros between each filter element. The dilation factor determines the step size for sampling the input or equivalently the upsampling factor of the filter. It corresponds to an effective filter size of (Filter Size – 1) .* Dilation Factor + 1. For example, a 3-by-3 filter with the dilation factor [2 2] is equivalent to a 5-by-5 filter with zeros between the elements.

Dilated Convolution

- This image shows a 3-by-3 filter dilated by a factor of two scanning through the input. The lower map represents the input and the upper map represents the output.



Feature Maps

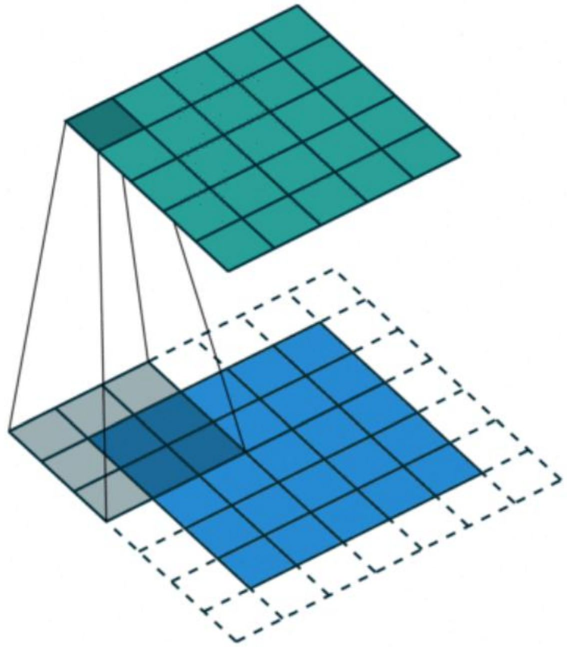
- As a filter moves along the input, it uses the same set of weights and the same bias for the convolution, forming a feature map. Each feature map is the result of a convolution using a different set of weights and a different bias. Hence, the number of feature maps is equal to the number of filters. The total number of parameters in a convolutional layer is ((h*w*c + 1)*Number of Filters), where 1 is the bias.

Zero Padding

- You can also apply zero padding to input image borders vertically and horizontally using the 'Padding' name-value pair argument. Padding is rows or columns of zeros added to the borders of an image input. By adjusting the padding, you can control the output size of the layer.

Zero Padding

- This image shows a 3-by-3 filter scanning through the input with padding of size 1. The lower map represents the input and the upper map represents the output.



Output Size

- The output height and width of a convolutional layer is (Input Size – ((Filter Size – 1)*Dilation Factor + 1) + 2*Padding)/Stride + 1. This value must be an integer for the whole image to be fully covered. If the combination of these options does not lead the image to be fully covered, the software by default ignores the remaining part of the image along the right and bottom edges in the convolution.

Number of Neurons

- The product of the output height and width gives the total number of neurons in a feature map, say Map Size. The total number of neurons (output size) in a convolutional layer is Map Size*Number of Filters.
- Usually, the results from these neurons pass through some form of nonlinearity, such as rectified linear units (ReLU).

Number of Neurons

- For example, suppose that the input image is a 32-by-32-by-3 color image. For a convolutional layer with eight filters and a filter size of 5-by-5, the number of weights per filter is 5 * 5 * 3 = 75, and the total number of parameters in the layer is (75 + 1) * 8 = 608. If the stride is 2 in each direction and padding of size 2 is specified, then each feature map is 16-by-16. This is because (32 – 5 + 2 * 2)/2 + 1 = 16.5, and some of the outermost zero padding to the right and bottom of the image is discarded. Finally, the total number of neurons in the layer is 16 * 16 * 8 = 2048.

Batch Normalization Layer

- Create a batch normalization layer using batchNormalizationLayer.
- A batch normalization layer normalizes each input channel across a mini-batch. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers.

Batch Normalization Layer

- The layer first normalizes the activations of each channel by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. Then, the layer shifts the input by a learnable offset β and scales it by a learnable scale factor γ. β and γ are themselves learnable parameters that are updated during network training.

Batch Normalization Layer

- Batch normalization layers normalize the activations and gradients propagating through a neural network, making network training an easier optimization problem. You can try increasing the learning rate. You can also try reducing the L2 and dropout regularization. With batch normalization layers, the activations of a specific image during training depend on which images happen to appear in the same mini-batch. To take full advantage of this regularizing effect, try shuffling the training data before every training epoch.

ReLU Layer

- Create a ReLU layer using reluLayer.
- A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Max and Average Pooling Layers

- A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region. Create a max pooling layer using maxPooling2dLayer.
- An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region. Create an average pooling layer using averagePooling2dLayer.

Max and Average Pooling Layers

- Pooling layers follow the convolutional layers for down-sampling, hence, reducing the number of connections to the following layers. They do not perform any learning themselves, but reduce the number of parameters to be learned in the following layers. They also help reduce overfitting.
- Pooling layers scan through the input horizontally and vertically in step sizes you can specify using the 'Stride' name-value pair argument. If the pool size is smaller than or equal to the stride, then the pooling regions do not overlap.

Dropout Layer

- Create a dropout layer using dropoutLayer. A dropout layer randomly sets input elements to zero with a given probability.
- At training time, the layer randomly sets input elements to zero given by the dropout mask rand(size(X))<Probability, where X is the layer input and then scales the remaining elements by 1/(1-Probability). This operation effectively changes the underlying network architecture between iterations and helps prevent the network from overfitting. A higher number results in more elements being dropped during training.
- At prediction time, the output of the layer is equal to its input.

Fully Connected Layer

- Create a fully connected layer using fullyConnectedLayer. A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.
- The convolutional (and down-sampling) layers are followed by one or more fully connected layers.

Fully Connected Layer

- As the name suggests, all neurons in a fully connected layer connect to all the neurons in the previous layer. This layer combines all of the features (local information) learned by the previous layers across the image to identify the larger patterns. For classification problems, the last fully connected layer combines the features to classify the images. This is the reason that the outputSize argument of the last fully connected layer of the network is equal to the number of classes of the data set. For regression problems, the output size must be equal to the number of response variables.

Softmax and Classification Layers

- A softmax layer applies a softmax function to the input. Create a softmax layer using softmaxLayer.
- A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes. Create a classification layer using classificationLayer.
- For classification problems, a softmax layer and then a classification layer must follow the final fully connected layer.

Regression Layer

- Create a regression layer using regressionLayer.
- A regression layer computes the half-mean-squared-error loss for regression problems. For typical regression problems, a regression layer must follow the final fully connected layer.

Classify Text Data Using Convolutional Neural Network

- This example shows how to classify text data using a convolutional neural network. To classify text data using convolutions, you must convert the text data into images. To do this, pad or truncate the observations to have constant length S and convert the documents into sequences of word vectors of length C using a word embedding. You can then represent a document as a 1-by-S-by-C image (an image with height 1, width S, and C channels).

Classify Text Data Using Convolutional Neural Network

- To convert text data from a CSV file to images, create a tabularTextDatastore object. Then convert the data read from the tabularTextDatastore object to images for deep learning by calling transform with a custom transformation function. The transformTextData function, takes data read from the datastore and a pretrained word embedding, and converts each observation to an array of word vectors.

Classify Text Data Using Convolutional Neural Network

- This example trains a network with 1-D convolutional filters of varying widths. The width of each filter corresponds the number of words the filter can see (the n-gram length). The network has multiple branches of convolutional layers, so it can use different n-gram lengths.

Load Pretrained Word Embedding and Data

- Load the pretrained fastText word embedding.

```
emb = fastTextWordEmbedding;
```

- Create a tabular text datastore from the data in factoryReports.csv. Read the data from the "Description" and "Category" columns only.

```
filenameTrain = "factoryReports.csv";
textName = "Description";
labelName = "Category";
ttdsTrain = tabularTextDatastore(filenameTrain,'SelectedVariableNames',[textName
labelName]);
```

tabularTextDatastore

- Use a TabularTextDatastore object to manage large collections of text files containing column-oriented or tabular data where the collection does not necessarily fit in memory.
- Tabular data is data that is arranged in a rectangular fashion with each row having the same number of entries.

Load Data

- Create a custom transform function that converts data read from the datastore to a table containing the predictors and the responses. The transformTextData function, takes the data read from a tabularTextDatastore object and returns a table of predictors and responses. The predictors are 1-by-sequenceLength-by-C arrays of word vectors given by the word embedding emb, where C is the embedding dimension. The responses are categorical labels over the classes in classNames.

Load Data

- Read the labels from the training data using the readLabels function, and find the unique class names.

```
labels = readLabels(ttdsTrain,labelName);
classNames = unique(labels);
numObservations = numel(labels);
```

readLabels

```
function labels = readLabels(ttds,labelName)
  ttdsNew = copy(ttds);
  ttdsNew.SelectedVariableNames = labelName;
  tbl = readall(ttdsNew);
  labels = tbl.(labelName);
end
```

Load Data

Transform the datastore using transformTextData function and specify a sequence length of 14.

```
sequenceLength = 14;
tdsTrain = transform(ttdsTrain, @(data)
transformTextData(data,sequenceLength,emb,classNames))
```

Transform Text Data Function

- The transformTextData function takes the data read from a tabularTextDatastore object and returns a table of predictors and responses. The predictors are 1-by-sequenceLength-by-C arrays of word vectors given by the word embedding emb, where C is the embedding dimension. The responses are categorical labels over the classes in classNames.

Transform Text Data Function

```
function dataTransformed = transformTextData(data,sequenceLength,emb,classNames)
  % Preprocess documents.
  textData = data{:,1};
  % Prepocess text
  dataTransformed = preprocessText(textData,sequenceLength,emb);
  % Read labels.
  labels = data{:,2};
  responses = categorical(labels,classNames);
  % Convert data to table.
  dataTransformed.Responses = responses;
end
```

Preprocess Text Function

```
function tbl = preprocessText(textData,sequenceLength,emb)
  documents = tokenizedDocument(textData);
  documents = lower(documents);
  % Convert documents to embeddingDimension-by-sequenceLength-by-1 images.
  predictors = doc2sequence(emb,documents,'Length',sequenceLength);
  % Reshape images to be of size 1-by-sequenceLength-embeddingDimension.
  predictors = cellfun(@(X) permute(X,[3 2 1]),predictors,'UniformOutput',false);
  tbl = table;
  tbl.Predictors = predictors;
end
```

Load Data

- Preview the transformed datastore. The predictors are 1-by-S-by-C arrays, where S is the sequence length and C is the number of features (the embedding dimension). The responses are the categorical labels.

```
preview(tdsTrain)
ans=8×2 table
 Predictors Responses
 {1×14×300 single} Mechanical Failure
 {1×14×300 single} Mechanical Failure
 {1×14×300 single} Electronic Failure ......
```

Define Network Architecture

The following steps describe the network architecture.

- Specify an input size of 1-by-S-by-C, where S is the sequence length and C is the number of features (the embedding dimension).
- For the n-gram lengths 2, 3, 4, and 5, create blocks of layers containing a convolutional layer, a batch normalization layer, a ReLU layer, a dropout layer, and a max pooling layer.
- For each block, specify 200 convolutional filters of size 1-by-N and pooling regions of size 1-by-S, where N is the n-gram length.

Define Network Architecture

- Connect the input layer to each block and concatenate the outputs of the blocks using a depth concatenation layer.
- To classify the outputs, include a fully connected layer with output size K, a softmax layer, and a classification layer, where K is the number of classes.

Define Network Architecture

- First, in a layer array, specify the input layer, the first block for unigrams, the depth concatenation layer, the fully connected layer, the softmax layer, and the classification layer.

```
numFeatures = emb.Dimension;
inputSize = [1 sequenceLength numFeatures];
numFilters = 200;
ngramLengths = [2 3 4 5];
numBlocks = numel(ngramLengths);
numClasses = numel(classNames);
```

Define Network Architecture

- Create a layer graph containing the input layer. Set the normalization option to 'none' and the layer name to 'input'.

```
layer = imageInputLayer(inputSize,'Normalization','none','Name','input');
lgraph = layerGraph(layer);
```

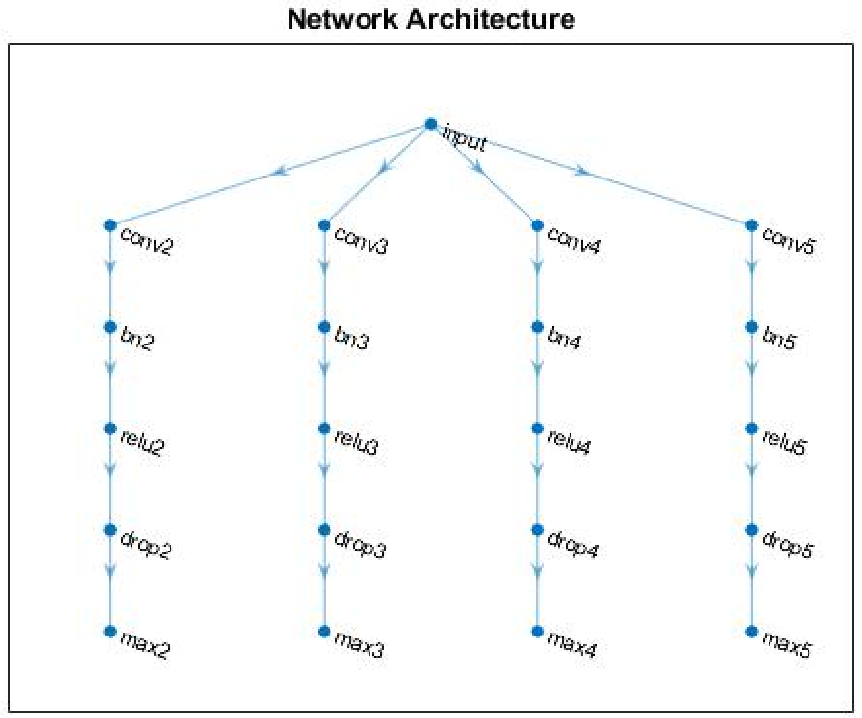
- For each of the n-gram lengths, create a block of convolution, batch normalization, ReLU, dropout, and max pooling layers. Connect each block to the input layer.

```
for j = 1:numBlocks
   N = ngramLengths(j);
   block = [
      convolution2dLayer([1 N],numFilters,'Name',"conv"+N,'Padding','same')
      batchNormalizationLayer('Name',"bn"+N)
      reluLayer('Name',"relu"+N)
      dropoutLayer(0.2,'Name',"drop"+N)
      maxPooling2dLayer([1 sequenceLength],'Name',"max"+N)];
   lgraph = addLayers(lgraph,block);
   lgraph = connectLayers(lgraph,'input',"conv"+N);
end
```

Define Network Architecture

- View the network architecture in a plot.

```
figure
plot(lgraph)
title("Network Architecture")
```



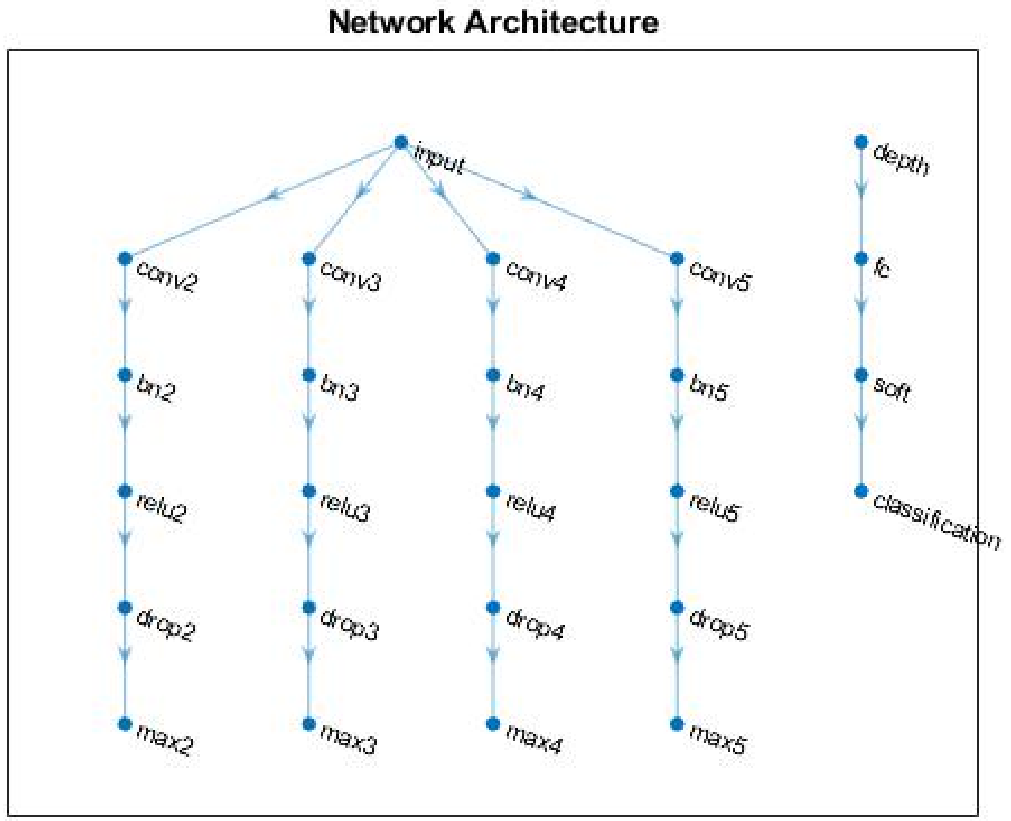
Define Network Architecture

- Add the depth concatenation layer, the fully connected layer, the softmax layer, and the classification layer.

```
layers = [
 depthConcatenationLayer(numBlocks,'Name','depth')
 fullyConnectedLayer(numClasses,'Name','fc')
 softmaxLayer('Name','soft')
 classificationLayer('Name','classification')];
lgraph = addLayers(lgraph,layers);
```

Define Network Architecture

```
figure
plot(lgraph)
title("Network Architecture")
```



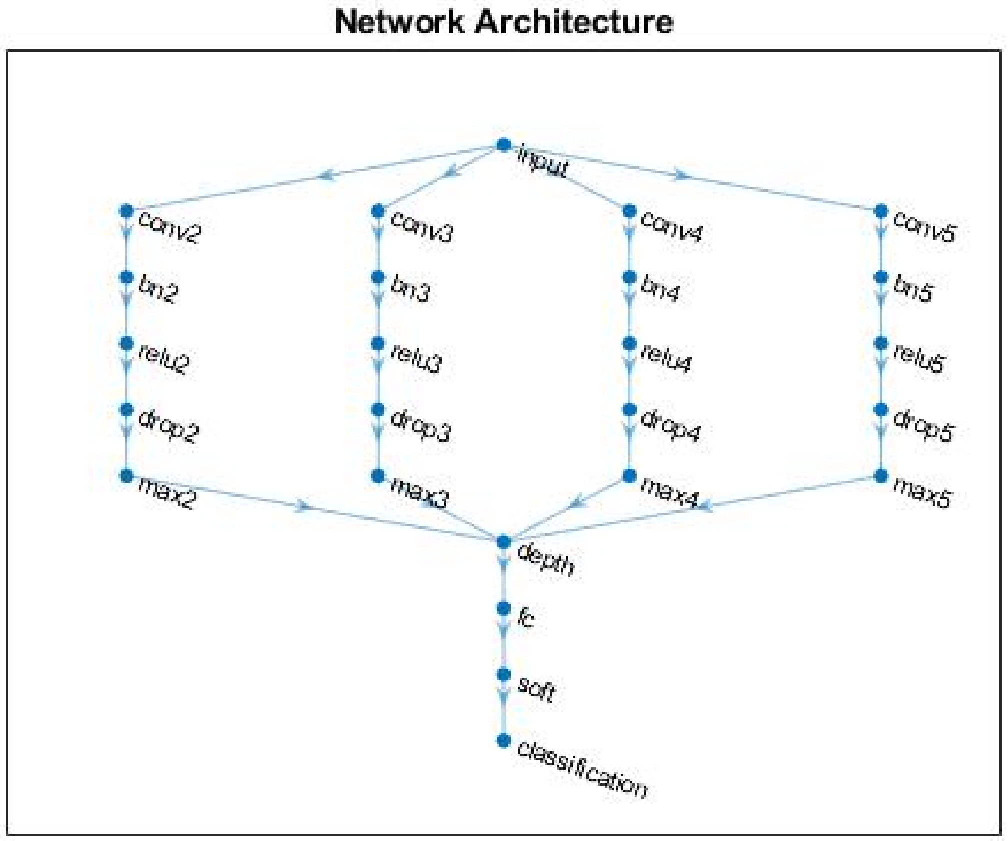
Define Network Architecture

- Connect the max pooling layers to the depth concatenation layer and view the final network architecture in a plot.

```
for j = 1:numBlocks
 N = ngramLengths(j);
 lgraph = connectLayers(lgraph,"max"+N,"depth/in"+j);
end
```

Define Network Architecture

```
figure
plot(lgraph)
title("Network Architecture")
```



Train Network

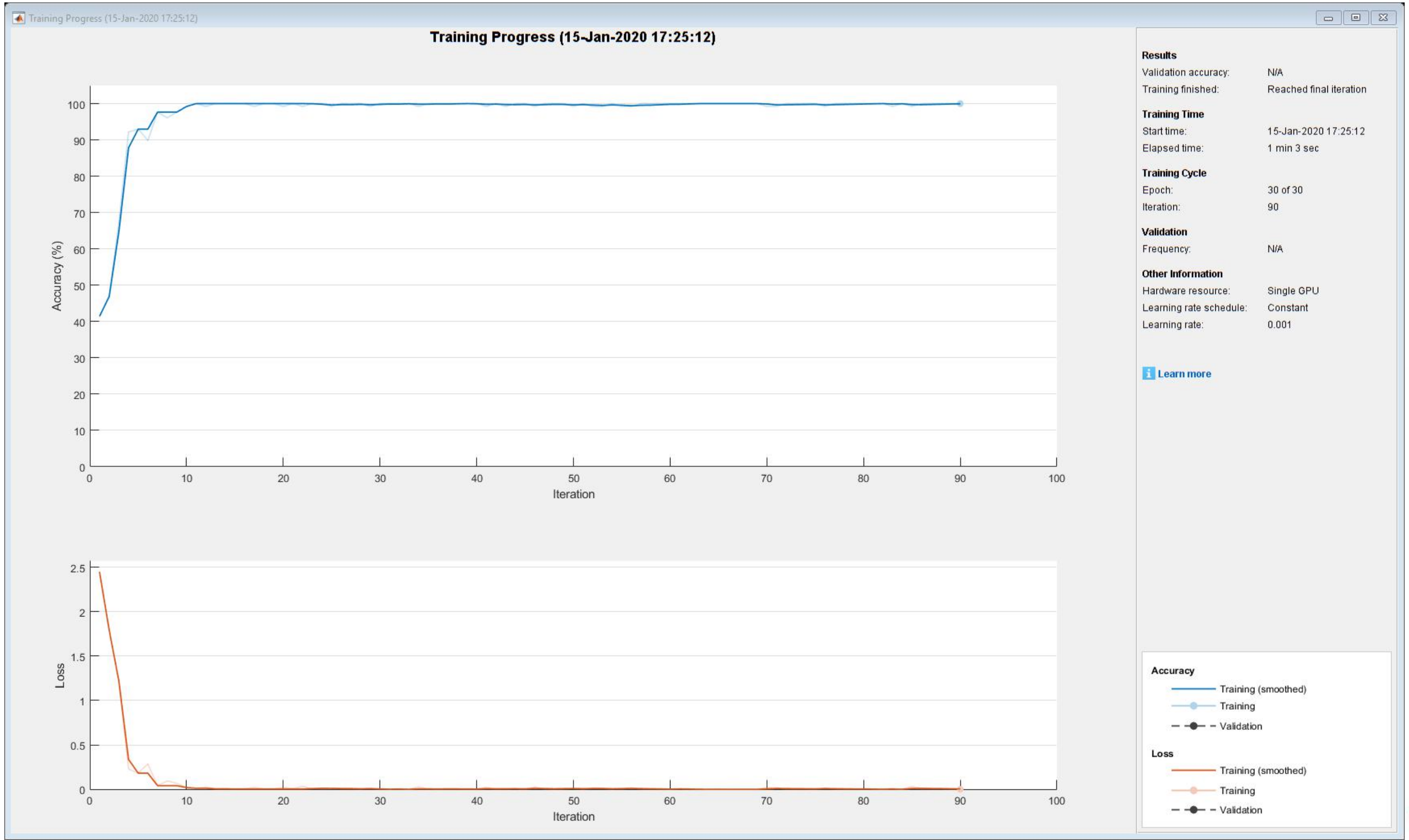
Specify the training options:

- Train with a mini-batch size of 128.
- Do not shuffle the data because the datastore is not shuffleable.
- Display the training progress plot and suppress the verbose output.

```
miniBatchSize = 128;
numIterationsPerEpoch = floor(numObservations/miniBatchSize);
options = trainingOptions('adam', ...
 'MiniBatchSize',miniBatchSize, ...
 'Shuffle','never', ...
 'Plots','training-progress', ...
 'Verbose',false);
```

Train the network using the trainNetwork function.

```
net = trainNetwork(tdsTrain,lgraph,options);
```



Training Progress (15-Jan-2020 17:25:12)

Results

Validation accuracy:	N/A
Training finished:	Reached final iteration

Training Time

Start time:	15-Jan-2020 17:25:12
Elapsed time:	1 min 3 sec

Training Cycle

Epoch:	30 of 30
Iteration:	90

Validation

Frequency:	N/A

Other Information

Hardware resource:	Single GPU
Learning rate schedule:	Constant
Learning rate:	0.001

Learn more

Accuracy
- Training (smoothed)
- Training
- Validation

Loss
- Training (smoothed)
- Training
- Validation

OF CHINA

Predict Using New Data

- Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [
 "Coolant is pooling underneath sorter."
 "Sorter blows fuses at start up."
 "There are some very loud rattling sounds coming from the assembler."];
```

- Preprocess the text data using the preprocessing steps as the training documents.

```
XNew = preprocessText(reportsNew,sequenceLength,emb);
```

Predict Using New Data

- Classify the new sequences using the trained LSTM network.

```
labelsNew = classify(net,XNew)
labelsNew = 3×1 categorical
 Leak
 Electronic Failure
 Mechanical Failure
```