Deep Learning Models for Text Processing - Autoencoder

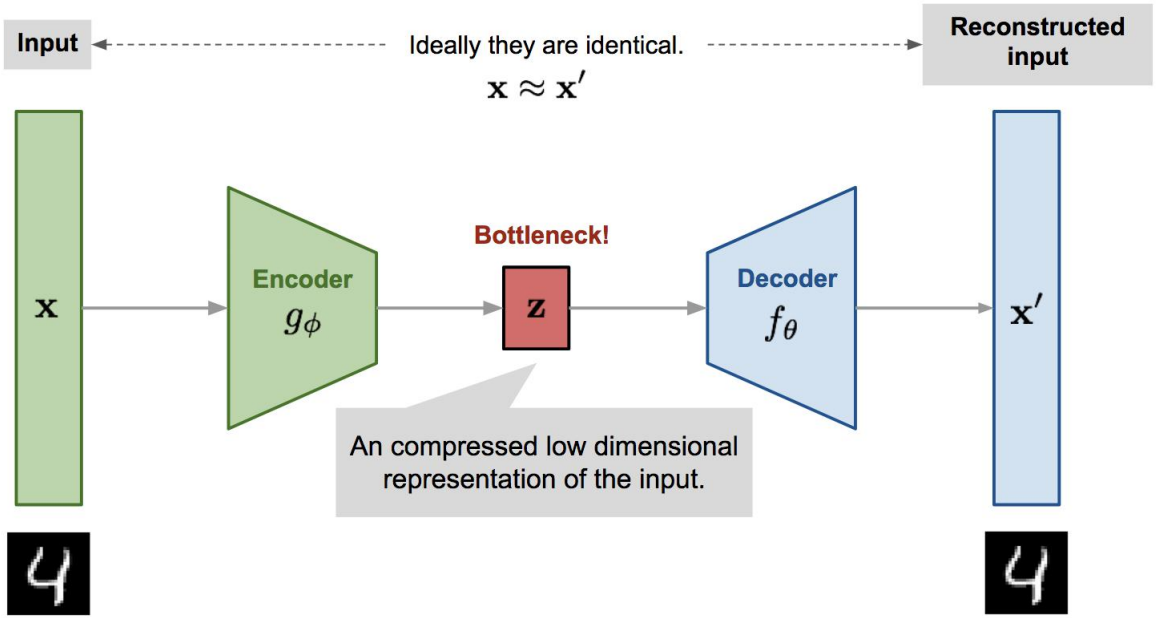
- generate text data using autoencoders

Outline

- About Autoencoders
- Generate Text Using Autoencoders

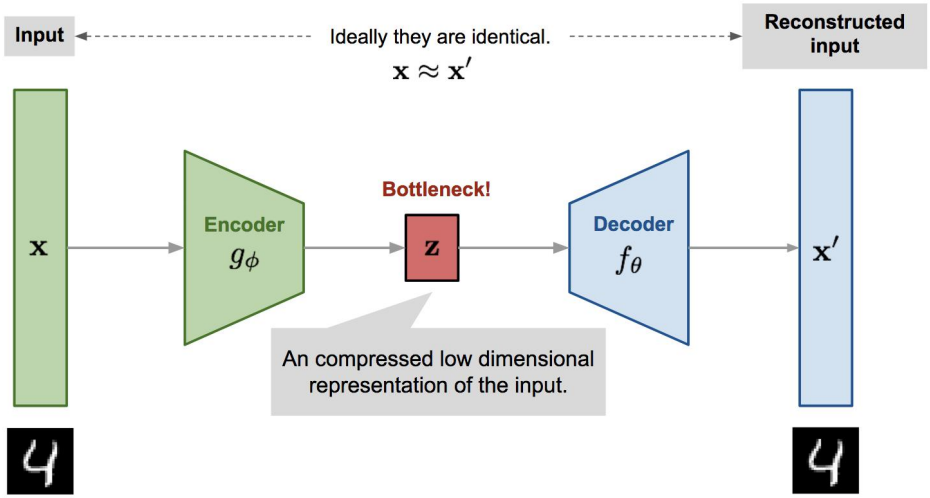
Autoencoders

- Autoencoder is a neural network designed to learn an identity function in an unsupervised way to reconstruct the original input while compressing the data in the process so as to discover a more efficient and compressed representation.



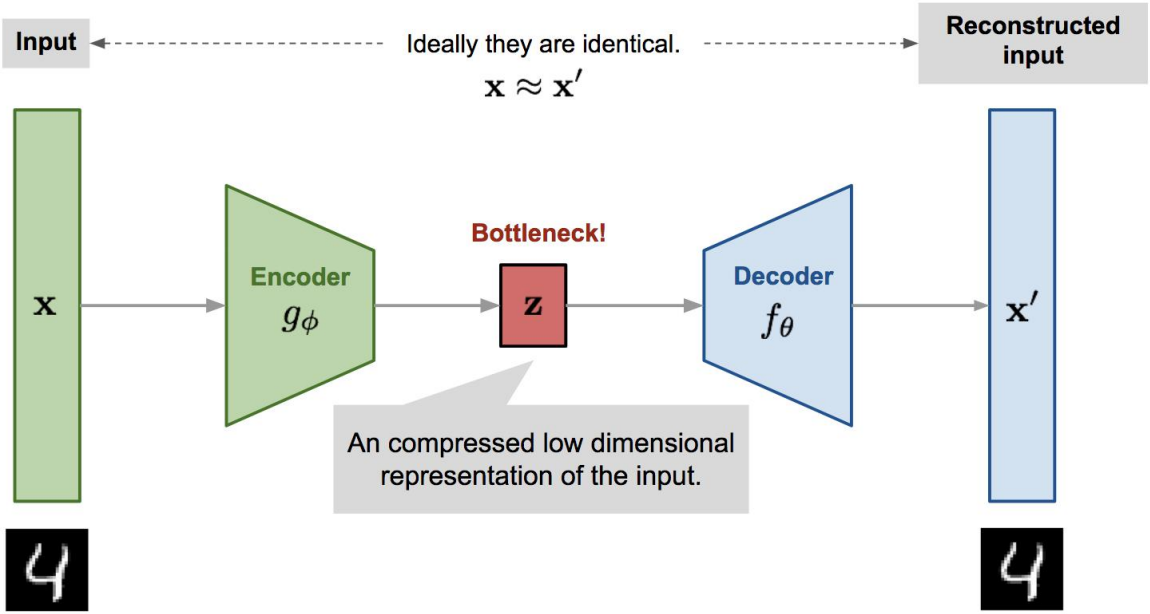
Autoencoders

- It consists of two networks:
  - Encoder network: It translates the original high-dimension input into the latent low-dimensional code. The input size is larger than the output size.
  - Decoder network: The decoder network recovers the data from the code, likely with larger and larger output layers.



Autoencoders

- We can perform unsupervised learning of features using autoencoder neural networks. If you have unlabeled data, perform unsupervised learning with autoencoder neural networks for feature extraction.



Generate Text Using Autoencoders

- This example shows how to generate text data using autoencoders.
- An autoencoder is a type of deep learning network that is trained to replicate its input. An autoencoder consists of two smaller networks: and encoder and a decoder. The encoder maps the input data to a feature vector in some latent space. The decoder reconstructs data using vectors in this latent space.

Generate Text Using Autoencoders

- The training process is unsupervised. In other words, the model does not require labeled data. To generate text, you can use the decoder to reconstruct text from arbitrary input.
- This example trains an autoencoder to generate text. The encoder uses a word embedding and an LSTM operation to map the input text into latent vectors. The decoder uses an LSTM operation and the same embedding to reconstruct the text from the latent vectors.

Load Data

- The file sonnets.txt contains all of Shakespeare's sonnets in a single text file.
- Read the Shakespeare's Sonnets data from the file "sonnets.txt".

```
filename = "sonnets.txt";
textData = fileread(filename);
```

sonnets.txt

```
THE SONNETS

by William Shakespeare




  I

  From fairest creatures we desire increase,
  That thereby beauty's rose might never die,
  But as the riper should by time decease,
  His tender heir might bear his memory:
  But thou, contracted to thine own bright eyes,
  Feed'st thy light's flame with self-substantial fuel,
  Making a famine where abundance lies,
  Thy self thy foe, to thy sweet self too cruel:
  Thou that art now the world's fresh ornament,
  And only herald to the gaudy spring,
  Within thine own bud buriest thy content,
  And tender churl mak'st waste in niggarding:
    Pity the world, or else this glutton be,
    To eat the world's due, by the grave and thee.

  II
```

textData

1x100266 char

```
val =

    'THE SONNETS

     by William Shakespeare




       I

       From fairest creatures we desire increase,
       That thereby beauty's rose might never die,
       But as the riper should by time decease,
       His tender heir might bear his memory:
       But thou, contracted to thine own bright eyes,
       Feed'st thy light's flame with self-substantial fuel,
       Making a famine where abundance lies,
       Thy self thy foe, to thy sweet self too cruel:
       Thou that art now the world's fresh ornament,
       And only herald to the gaudy spring,
       Within thine own bud buriest thy content,
       And tender churl mak'st waste in niggarding:
         Pity the world, or else this glutton be,
         To eat the world's due, by the grave and thee.

       II
```

Load Data

- The sonnets are indented by two whitespace characters. Remove the indentations using replace and split the text into separate lines using the split function. Remove the header from the first nine elements and the short sonnet titles.

```
textData = replace(textData,"  ","");
textData = split(textData,newline);
textData(1:9) = [];
textData(strlength(textData)<5) = [];
```

textData

2625x1 cell

	1	2	3
1	THE SONNETS		
2			
3	by William Shakespeare		
4			
5			
6			
7			
8	I		
9			
10	From fairest creatures we desire increase,		
11	That thereby beauty's rose might never die,		
12	But as the riper should by time decease,		
13	His tender heir might bear his memory:		
14	But thou, contracted to thine own bright eyes,		
15	Feed'st thy light's flame with self-substantial fuel,		
16	Making a famine where abundance lies,		
17	Thy self thy foe, to thy sweet self too cruel:		
18	Thou that art now the world's fresh ornament,		
19	And only herald to the gaudy spring,		
20	Within thine own bud buriest thy content,		
21	And tender churl mak'st waste in niggarding:		
22	Pity the world, or else this glutton be,		
23	To eat the world's due, by the grave and thee.		
24			
25	II		

textData

2220x1 cell

	1	2	3
1	From fairest creatures we desire increase,		
2	That thereby beauty's rose might never die,		
3	But as the riper should by time decease,		
4	His tender heir might bear his memory:		
5	But thou, contracted to thine own bright eyes,		
6	Feed'st thy light's flame with self-substantial fuel,		
7	Making a famine where abundance lies,		
8	Thy self thy foe, to thy sweet self too cruel:		
9	Thou that art now the world's fresh ornament,		
10	And only herald to the gaudy spring,		
11	Within thine own bud buriest thy content,		
12	And tender churl mak'st waste in niggarding:		
13	Pity the world, or else this glutton be,		
14	To eat the world's due, by the grave and thee.		
15	When forty winters shall besiege thy brow,		
16	And dig deep trenches in thy beauty's field,		
17	Thy youth's proud livery so gazed on now,		
18	Will be a tatter'd weed of small worth held:		
19	Then being asked, where all thy beauty lies,		
20	Where all the treasure of thy lusty days;		
21	To say, within thine own deep sunken eyes,		
22	Were an all-eating shame, and thriftless praise.		
23	How much more praise deserv'd thy beauty's use,		
24	If thou couldst answer 'This fair child of mine		
25	Shall sum my count, and make my old excuse,'		
26	Proving his beauty by succession thine!		

Prepare Data

- Create a function that tokenizes and preprocesses the text data. The function preprocessText performs these steps:
  - Prepends and appends each input string with the specified start and stop tokens, respectively.
  - Tokenize the text using tokenizedDocument.
  - Preprocess the text data and specify the start and stop tokens "<start>" and "<stop>", respectively.

```
startToken = "<start>";
stopToken = "<stop>";
documents = preprocessText(textData,startToken,stopToken);
```

Text Preprocessing Function

1. Prepends and appends each input string with the specified start and stop tokens, respectively.
2. Tokenize the text using tokenizedDocument.

```
function documents = preprocessText(textData,startToken,stopToken)
% Add start and stop tokens.
textData = startToken + textData + stopToken;
% Tokenize the text.
documents = tokenizedDocument(textData,'CustomTokens',[startToken stopToken]);
end
```

Prepare Data

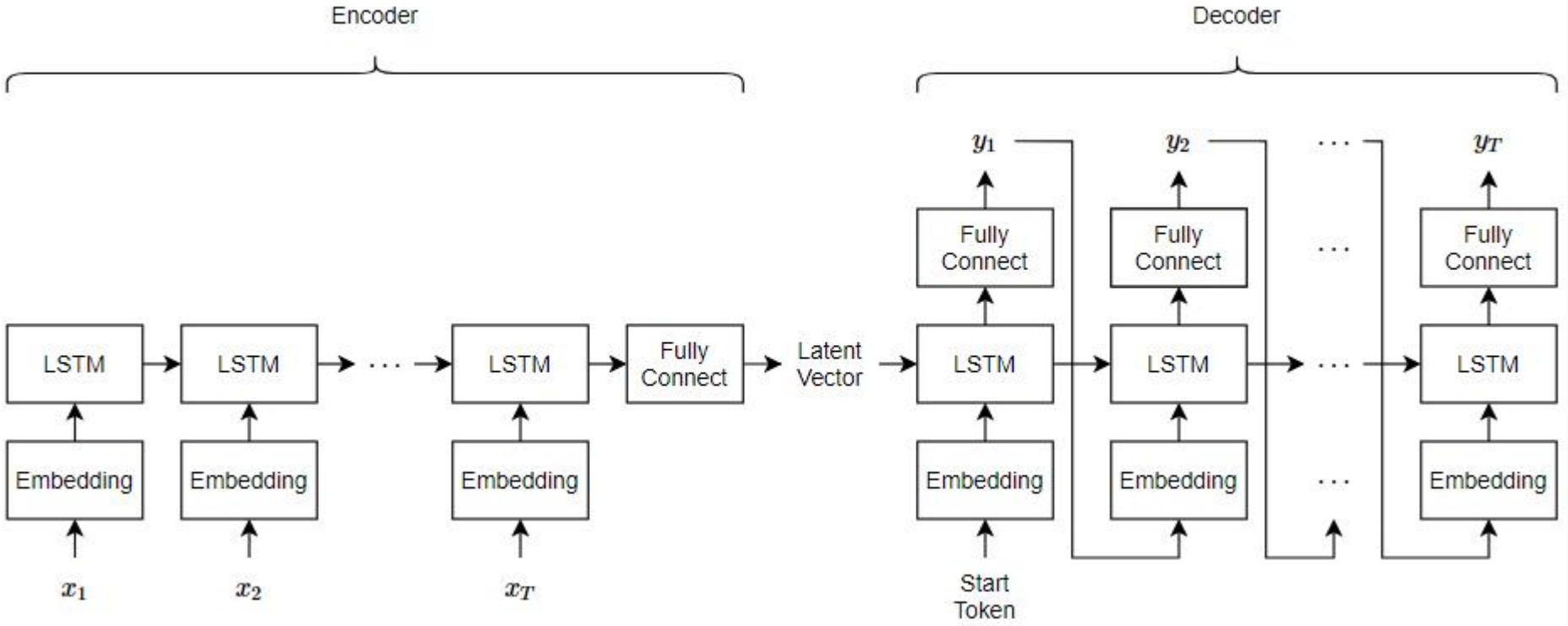
- Create a word encoding object from the tokenized documents.

```
enc = wordEncoding(documents);
```

- The input data must be a numeric array containing sequences of a fixed length. Because the documents have different lengths, you must pad the shorter sequences. Add a padding token and determine the index of that token.

```
paddingToken = "<pad>";
newVocabulary = [enc.Vocabulary paddingToken];
enc = wordEncoding(newVocabulary);
paddingIdx = word2ind(enc,paddingToken)
```

Initialize Model Parameters



Initialize Model Parameters

- The encoder maps sequences of word indices to a latent vector by converting the input to sequences of word vectors using an embedding, inputting the word vector sequences into an LSTM operation, and applying a fully connected operation to the last time step of the LSTM output. The decoder reconstructs the input using an LSTM initialized the encoder output. For each time step, the decoder predicts the next time step and uses the output for the next time-step predictions. Both the encoder and the decoder use the same embedding.

Initialize Model Parameters

- Specify the dimensions of the parameters.

```
embeddingDimension = 100;
numHiddenUnits = 150;
latentDimension = 75;
vocabularySize = enc.NumWords;
```

- Create a struct for the parameters.

```
parameters = struct;
```

Initialize Model Parameters

- Initialize the weights of the embedding using the Gaussian using the initializeGaussian function which is attached to this example as a supporting file. Specify a mean of 0 and a standard deviation of 0.01.

```
mu = 0;
sigma = 0.01;
parameters.emb.Weights = initializeGaussian([embeddingDimension vocabularySize],mu,sigma);
```

Initialize Model Parameters

- Initialize the learnable parameters for the encoder LSTM operation:
  - Initialize the input weights with the Glorot initializer using the initializeGlorot function.
  - Initialize the recurrent weights with the orthogonal initializer using the initializeOrthogonal function.
  - Initialize the bias with the unit forget gate initializer using the initializeUnitForgetGate function.

Initialize Model Parameters

```
sz = [4*numHiddenUnits embeddingDimension];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension;

parameters.lstmEncoder.InputWeights =
initializeGlorot(sz,numOut,numIn);
parameters.lstmEncoder.RecurrentWeights =
initializeOrthogonal([4*numHiddenUnits numHiddenUnits]);
parameters.lstmEncoder.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize Model Parameters

- Initialize the learnable parameters for the encoder fully connected operation:
  - Initialize the weights with the Glorot initializer.
  - Initialize the bias with zeros using the initializeZeros function.

```
sz = [latentDimension numHiddenUnits];
numOut = latentDimension;
numIn = numHiddenUnits;
parameters.fcEncoder.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fcEncoder.Bias = initializeZeros([latentDimension 1]);
```

Initialize Model Parameters

- Initialize the learnable parameters for the decoder LSTM operation:
  - Initialize the input weights with the Glorot initializer.
  - Initialize the recurrent weights with the orthogonal initializer.
  - Initialize the bias with the unit forget gate initializer.

```
sz = [4*latentDimension embeddingDimension];
numOut = 4*latentDimension;
numIn = embeddingDimension;
parameters.lstmDecoder.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.lstmDecoder.RecurrentWeights = initializeOrthogonal([4*latentDimension latentDimension]);
parameters.lstmDecoder.Bias = initializeZeros([4*latentDimension 1]);
```

Initialize Model Parameters

- Initialize the learnable parameters for the decoder fully connected operation:
  - Initialize the weights with the Glorot initializer.
  - Initialize the bias with zeros.

```
sz = [vocabularySize latentDimension];
numOut = vocabularySize;
numIn = latentDimension;
parameters.fcDecoder.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fcDecoder.Bias = initializeZeros([vocabularySize 1]);
```

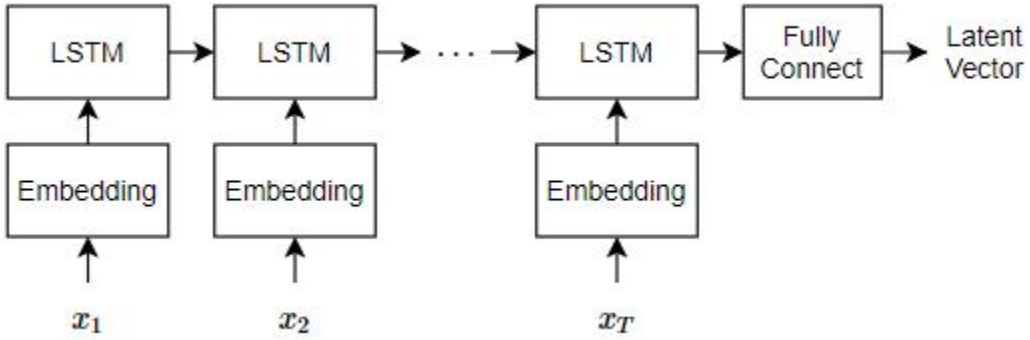
Initialize Model Parameters

- Initialize the learnable parameters for the decoder fully connected operation:
  - Initialize the weights with the Glorot initializer.
  - Initialize the bias with zeros.

```
sz = [vocabularySize latentDimension];
numOut = vocabularySize;
numIn = latentDimension;
parameters.fcDecoder.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fcDecoder.Bias = initializeZeros([vocabularySize 1]);
```

Define Model Encoder Function

- Create the function **modelEncoder**, that computes the output of the encoder model. The modelEncoder function, takes as input sequences of word indices, the model parameters, and the sequence lengths, and returns the corresponding latent feature vector.



Define Model Encoder Function

- Because the input data contains padded sequences of different lengths, the padding can have adverse effects on loss calculations. For the LSTM operation, instead of returning the output of the last time step of the sequence (which likely corresponds to the LSTM state after processing lots of padding values), determine the actual last time step given by the sequenceLengths input.

Define Model Encoder Function

```
function dlZ = modelEncoder(parameters,dlX,sequenceLengths)
    weights = parameters.emb.Weights;  % Embedding.
    dlZ = embedding(dlX,weights); % See next

    % LSTM.
    inputWeights = parameters.lstmEncoder.InputWeights; %[4*numHiddenUnits embeddingDimension]
    recurrentWeights = parameters.lstmEncoder.RecurrentWeights;%[4*numHiddenUnits numHiddenUnits]
    bias = parameters.lstmEncoder.Bias; %[4*numHiddenUnits,1]
    numHiddenUnits = size(recurrentWeights,2);
    hiddenState = zeros(numHiddenUnits,1,'like',dlX);
    cellState = zeros(numHiddenUnits,1,'like',dlX);
  dlZ1 = lstm(dlZ,hiddenState,cellState,inputWeights,recurrentWeights,bias,'DataFormat','CBT');
```

Component	Formula
Input gate	$i_t = \sigma_g(W_i\mathbf{x}_t + \mathrm{R}_i\mathbf{h}_{t-1} + b_i)$
Forget gate	$f_t = \sigma_g(W_f\mathbf{x}_t + \mathrm{R}_f\mathbf{h}_{t-1} + b_f)$
Cell candidate	$g_t = \sigma_c(W_g\mathbf{x}_t + \mathrm{R}_g\mathbf{h}_{t-1} + b_g)$
Output gate	$o_t = \sigma_g(W_o\mathbf{x}_t + \mathrm{R}_o\mathbf{h}_{t-1} + b_o)$

$$\mathbf{h}_t = o_t \odot \sigma_c(\mathbf{c}_t),$$

$$\mathbf{c}_t = f_t \odot \mathbf{c}_{t-1} + i_t \odot g_t,$$

Define Model Encoder Function

```
    % Output mode 'last' with masking.
    miniBatchSize = size(dlZ1,2);
    dlZ = zeros(numHiddenUnits,miniBatchSize,'like',dlZ1);
    for n = 1:miniBatchSize
        t = sequenceLengths(n);
        dlZ(:,n) = dlZ1(:,n,t); %output of the actual last time step
    end
    % Fully connect.
    weights = parameters.fcEncoder.Weights; % [latentDimension numHiddenUnits]
    bias = parameters.fcEncoder.Bias; % [latentDimension 1]
    dlZ = fullyconnect(dlZ,weights,bias,'DataFormat','CB');
end
```

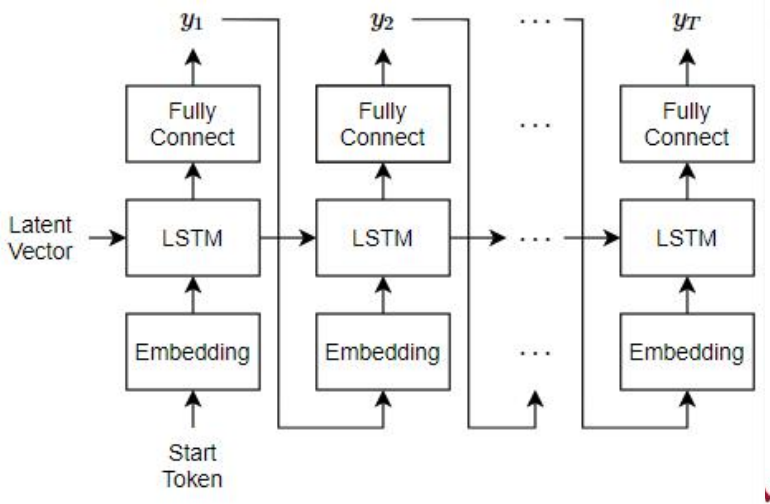
Embedding Function

- The embedding function maps sequences of indices to vectors using the given weights.

```
function Z = embedding(X, weights)
  % Reshape inputs into a vector.
  [N, T] = size(X, 2:3);
  X = reshape(X, N*T, 1);
  % Index into embedding matrix.
  Z = weights(:, X);
  % Reshape outputs by separating out batch and sequence dimensions.
  Z = reshape(Z, [], N, T);
end
```

Define Model Decoder Function

- Create the function **modelDecoder**, that computes the output of the decoder model. The modelDecoder function, takes as input the model parameters, sequences of word indices, and the network state, and returns the decoded sequences.



Define Model Decoder Function

- Because the lstm function is stateful (when given a time series as input, the function propagates and updates the state between each time step) and that the embedding and fullyconnect functions are time-distributed by default (when given a time series as input, the functions operate on each time step independently), the modelDecoder function supports both sequence and single time-step inputs.

Define Model Decoder Function

```
function [dlY,state] = modelDecoder(parameters,dlX,state)
    % Embedding.
    weights = parameters.emb.Weights;
    dlX = embedding(dlX,weights);
    inputWeights = parameters.lstmDecoder.InputWeights;
    % LSTM.
    recurrentWeights = parameters.lstmDecoder.RecurrentWeights;
    bias = parameters.lstmDecoder.Bias;
    hiddenState = state.HiddenState;
    cellState = state.CellState;
    [dlY,hiddenState,cellState] = lstm(dlX,hiddenState,cellState, inputWeights, ...
        recurrentWeights,bias,'DataFormat','CBT');
```

Define Model Decoder Function

```
    state.HiddenState = hiddenState;
    state.CellState = cellState;

    % Fully connect.
    weights = parameters.fcDecoder.Weights;
    bias = parameters.fcDecoder.Bias;
    dlY = fullyconnect(dlY,weights,bias,'DataFormat','CBT');

    % Softmax.
    dlY = softmax(dlY,'DataFormat','CBT');

end
```

Define Model Gradients Function

- The modelGradients function, takes as input the model learnable parameters, the input data dlX, and a vector of sequence lengths for masking, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

Define Model Gradients Function

- To calculate the masked loss, the model gradients function uses the maskedCrossEntropy loss function. To train the decoder to predict the next time-step of the sequence, specify the targets to be the input sequences shifted by one time-step.

Define Model Gradients Function

```
function [gradients, loss] = modelGradients(parameters,dlX,sequenceLengths)
    % Model encoder.
    dlZ = modelEncoder(parameters,dlX,sequenceLengths);
    % Initialize LSTM state.
    state = struct;
    state.HiddenState = dlZ;
    state.CellState = zeros(size(dlZ),'like',dlZ);
    % Teacher forcing.
    dlY = modelDecoder(parameters,dlX,state);
```

Define Model Gradients Function

```
    % Loss.
    dlYPred = dlY(:,:,1:end-1);
    dlT = dlX(:,:,2:end);
    loss = mean(maskedCrossEntropy(dlYPred,dlT,sequenceLengths));

    % Gradients.
    gradients = dlgradient(loss,parameters);

    % Normalize loss for plotting.
    sequenceLength = size(dlX,3);
    loss = loss / sequenceLength;

end
```

Masked Cross Entropy Loss Function

- The maskedCrossEntropy function calculates the loss between the specified input sequences and target sequences ignoring any time steps containing padding using the specified vector of sequence lengths.

Masked Cross Entropy Loss Function

```
function maskedLoss = maskedCrossEntropy(dlY,T,sequenceLengths)
  numClasses = size(dlY,1);
  miniBatchSize = size(dlY,2);
  sequenceLength = size(dlY,3);
  maskedLoss = zeros(sequenceLength,miniBatchSize,'like',dlY);
  for t = 1:sequenceLength
     T1 = single(oneHot(T(:,:,t),numClasses));
     mask = (t<=sequenceLengths)';
     maskedLoss(t,:) = mask .* crossentropy(dlY(:,:,t),T1,'DataFormat','CBT');
  end
  maskedLoss = sum(maskedLoss,1);
end
```

Specify Training Options

- Train for 100 epochs with a mini-batch size of 128.

```
miniBatchSize = 128;
numEpochs = 100;
```

- Train with a learning rate of 0.01.

```
learnRate = 0.01;
```

- Display the training progress in a plot.

```
plots = "training-progress";
```

Train Network

- Train the network using a custom training loop.
- Initialize the parameters for the Adam optimizer.

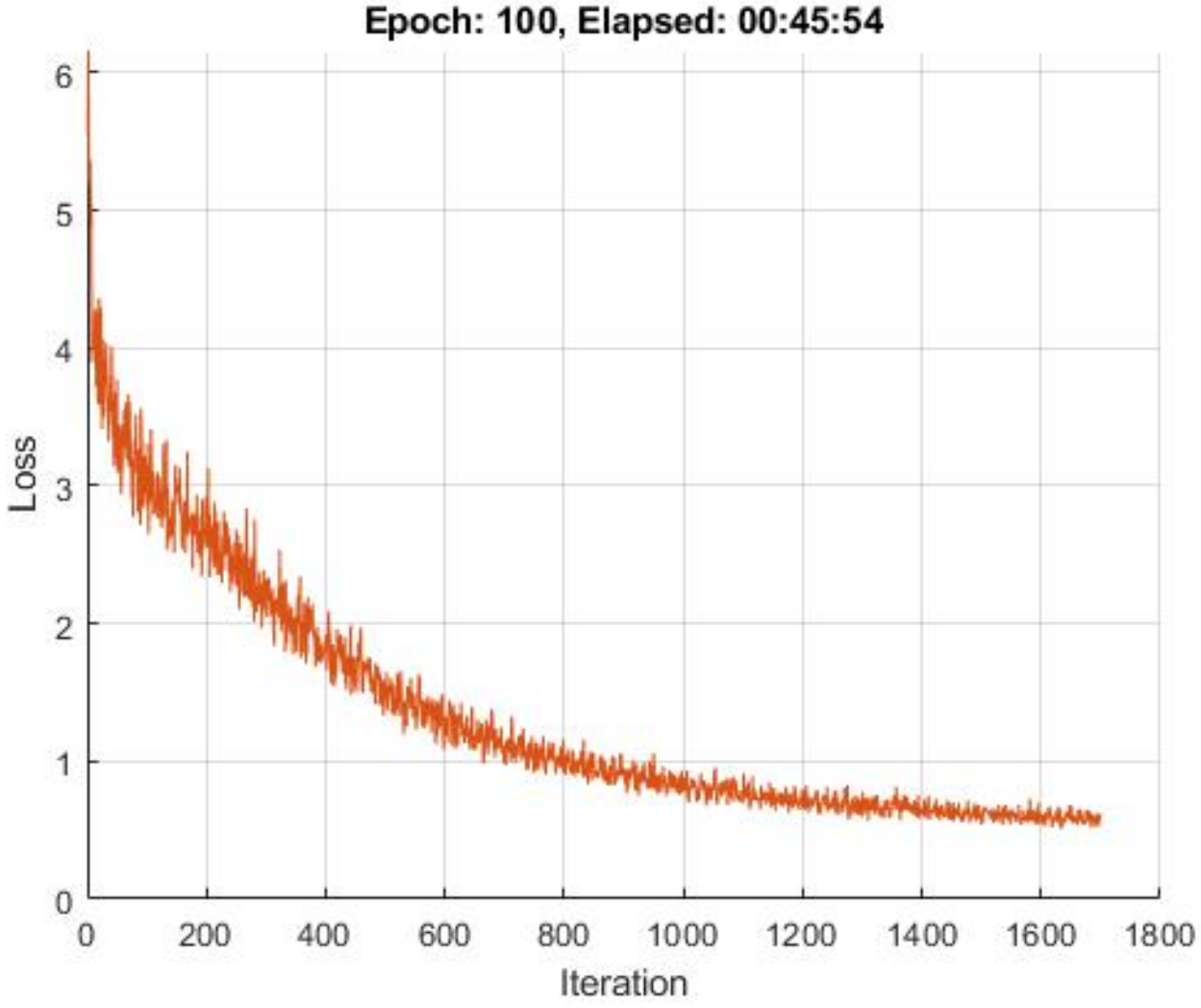
```
trailingAvg = [];
trailingAvgSq = [];
```

Train Network

- Train the model. For the first epoch, shuffle the data and loop over mini-batches of data.
- For each mini-batch:
  - Convert the text data to sequences of word indices.
  - Convert the data to dlarray.
  - For GPU training, convert the data to gpuArray objects.
  - Compute loss and gradients.
  - Update the learnable parameters using the adamupdate function.
  - Update the training progress plot.

Train Network

```
.....% Convert to sequences.
X = doc2sequence(enc,documentsBatch, 'PaddingDirection','right', ...
    'PaddingValue',paddingIdx);
X = cat(1,X{:});
% Convert to dlarray.
dlX = dlarray(X,'BTC');
% Calculate sequence lengths.
sequenceLengths = doclength(documentsBatch);
% Evaluate model gradients.
[gradients,loss] = dlfeval(@modelGradients, parameters, dlX, sequenceLengths);
% Update learnable parameters.
[parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
    trailingAvg,trailingAvgSq,iteration,learnRate); .....
```


Epoch: 100, Elapsed: 00:45:54
Loss
Iteration
0
1
2
3
4
5
6
0
200
400
600
800
1000
1200
1400
1600
1800

Generate Text

- Generate text using closed loop generation by initializing the decoder with different random states. Closed loop generation is when the model generates data one time-step at a time and uses the previous prediction as input for the next prediction.
- Specify to generate 3 sequences of length 16.

```
numGenerations = 3;
sequenceLength = 16;
```

Generate Text

- Create an array of random values to initialize the decoder state.

```
dlZ = dlarray(randn(latentDimension,numGenerations),'CB');
```

- If predicting on a GPU, then convert data to gpuArray.

```
if (executionEnvironment == "auto" && canUseGPU) ||
executionEnvironment == "gpu"
    dlZ = gpuArray(dlZ);
end
```

Generate Text

- Make predictions using the modelPredictions function. The modelPredictions function returns the output scores of the decoder given the model parameters, decoder initial state, maximum sequence length, word encoding, start token, and mini-batch size.

```
dlY = modelDecoderPredictions(parameters,dlZ,sequenceLength,enc,startToken,miniBatchSize);
```

- Find the word indices with the highest scores.

```
[~,idx] = max(dlY,[],1);
idx = squeeze(idx);
```

Model Predictions Function

- The modelPredictions function returns the output scores of the decoder given the model parameters, decoder initial state, maximum sequence length, word encoding, start token, and mini-batch size.

Model Predictions Function

```
function dlY = modelDecoderPredictions(parameters,dlZ,maxLength,enc,startToken,miniBatchSize)

  numObservations = size(dlZ,2);
  numIterations = ceil(numObservations / miniBatchSize);

  startTokenIdx = word2ind(enc,startToken);
  vocabularySize = enc.NumWords;

  dlY = zeros(vocabularySize,numObservations,maxLength,'like',dlZ);
```

```
    % Loop over mini-batches.
    for i = 1:numIterations
        idxMiniBatch = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);
        miniBatchSize = numel(idxMiniBatch);
        % Initialize state.
        state = struct;
        state.HiddenState = dlZ(:,idxMiniBatch);
        state.CellState = zeros(size(dlZ(:,idxMiniBatch)),'like',dlZ);
        % Initialize decoder input.
        decoderInput = dlarray(repmat(startTokenIdx,[1 miniBatchSize]),'CBT');
        % Loop over time steps.
        for t = 1:maxLength
            % Predict next time step.
            [dlY(:,idxMiniBatch,t), state] = modelDecoder(parameters,decoderInput,state);
            % Closed loop generation.
            [~,idx] = max(dlY(:,idxMiniBatch,t));
            decoderInput = idx;
        end
    end
end
```

Generate Text

- Convert the numeric indices to words and join them using the join function.

```
strGenerated = join(enc.Vocabulary(idx));
```

- Extract the text before the first stop token using the extractBefore function. To prevent the function from returning missing when there are no stop tokens, append a stop token to the end of each sequence.

```
strGenerated = extractBefore(strGenerated+stopToken,stopToken);
```

Generate Text

- Remove padding tokens.

```
strGenerated = erase(strGenerated,paddingToken);
```

- The generation process introduces whitespace characters between each prediction, which means that some punctuation characters appear with unnecessary spaces before and after. Reconstruct the generated text by removing the spaces before and after the appropriate punctuation characters.
- Remove the spaces before the specified punctuation characters.

```
punctuationCharacters = ["." "," "”" ")" ":" ";" "?" "!"];
strGenerated = replace(strGenerated," " +
punctuationCharacters,punctuationCharacters);
```

Generate Text

- Remove the spaces that appear after the specified punctuation characters.

```
punctuationCharacters = ["(" "‘"];
strGenerated = replace(strGenerated,punctuationCharacters + " ",punctuationCharacters);
```

- Remove leading and trailing white space using the strip function and view the generated text.

```
strGenerated = strip(strGenerated)
strGenerated = 3×1 string
   "love's thou rest light best ill mistake show seeing farther cross enough by me"
   "as before his bending sickle's compass come look find."
   "summer's lays? truth once lead mayst take,"
```