Deep Learning Models for Text Processing - Attentions

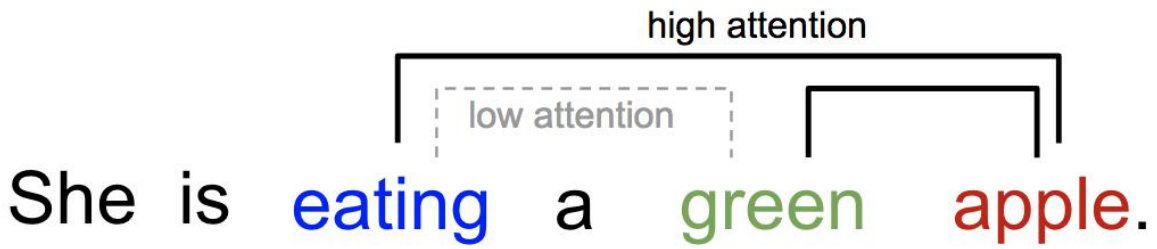
- building sequence-to-sequence model with attention

Outline

- About Attention
- Sequence-to-Sequence Translation Using Attention

Attention

- The attention mechanism has changed the way we work with deep learning algorithms. Fields like Natural Language Processing (NLP) and even Computer Vision have been revolutionized by the attention mechanism.

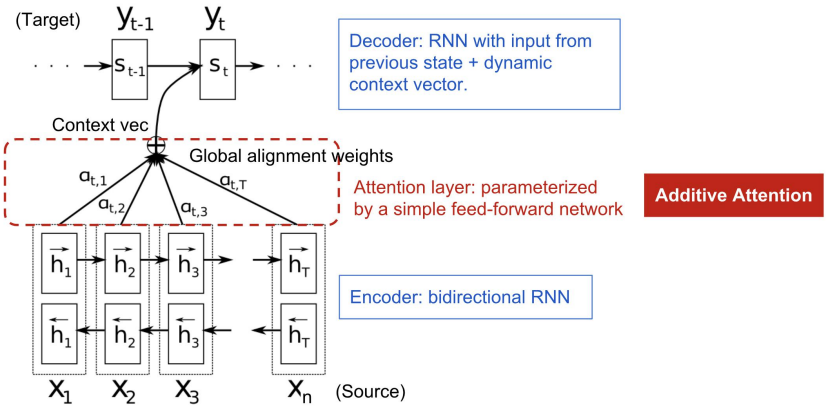


Attention

- In a nutshell, attention in deep learning can be broadly interpreted as a vector of importance weights: in order to predict or infer one element, such as a pixel in an image or a word in a sentence, we estimate using the attention vector how strongly it is correlated with (or “attends to” as you may have read in many papers) other elements and take the sum of their values weighted by the attention vector as the approximation of the target.

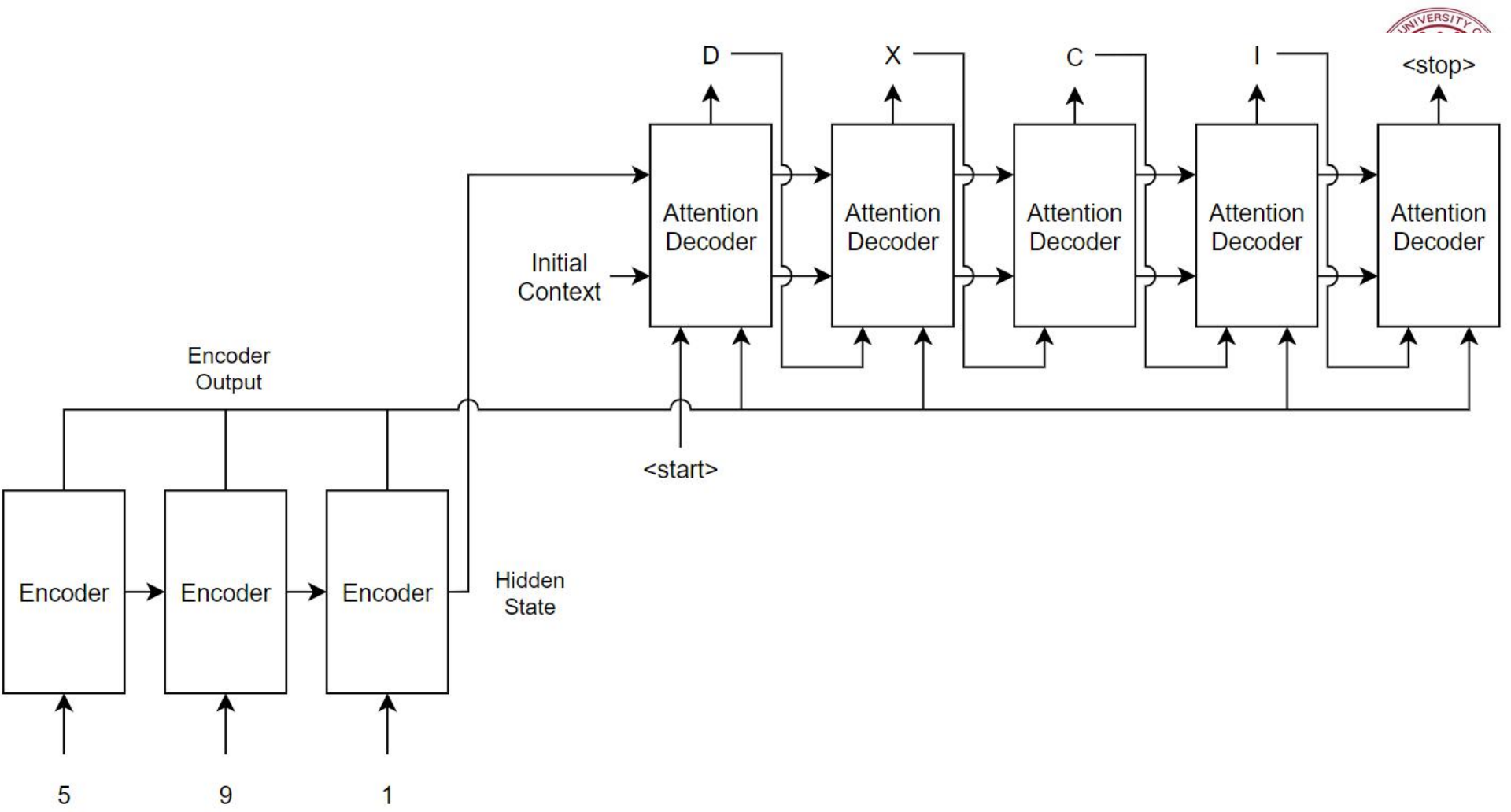
Attention

- The attention mechanism was born to help memorize long source sentences in neural machine translation (NMT). Rather than building a single context vector out of the encoder's last hidden state, the secret sauce invented by attention is to create shortcuts between the context vector and the entire source input. The weights of these shortcut connections are customizable for each output element.



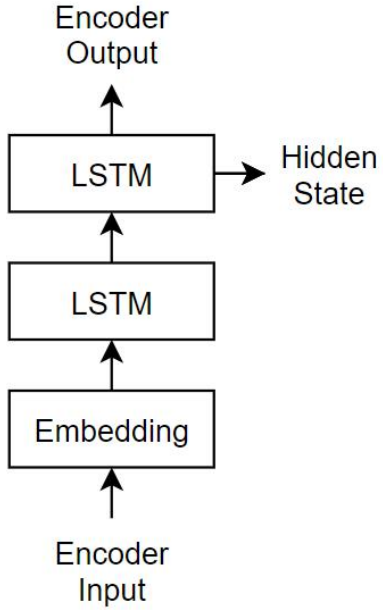
Sequence-to-Sequence Translation Using Attention

- This example shows how to convert decimal strings to Roman numerals using a recurrent sequence-to-sequence encoder-decoder model with attention.
- Recurrent encoder-decoder models have proven successful at tasks like abstractive text summarization and neural machine translation. The models consistent of an encoder which typically processes input data with a recurrent layer such as LSTM, and a decoder which maps the encoded input into the desired output, typically with a second recurrent layer. Models that incorporate attention mechanisms into the models allows the decoder to focus on parts of the encoded input while generating the translation.


D
X
C
I
<stop>
Attention Decoder
Attention Decoder
Attention Decoder
Attention Decoder
Attention Decoder
Initial Context
Encoder Output
<start>
Encoder
Encoder
Encoder
Hidden State
5
9
1

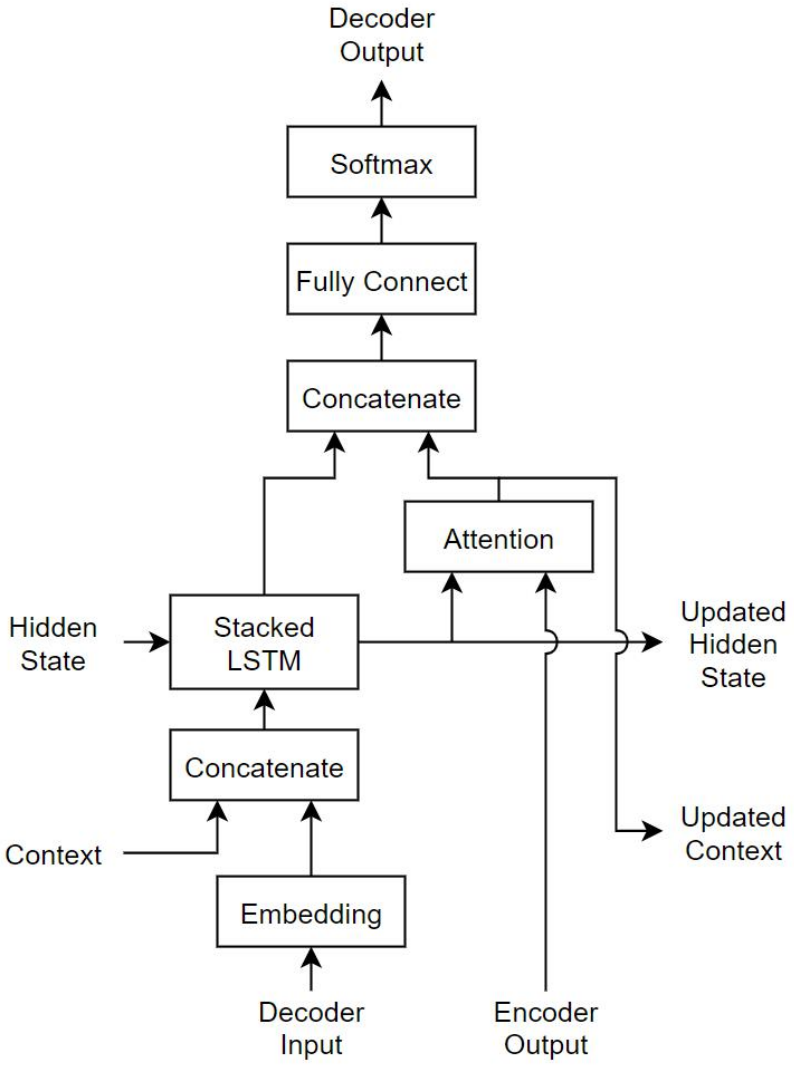
Sequence-to-Sequence Translation Using Attention

- For the encoder model, this example uses a simple network consisting of an embedding followed by two LSTM operations. Embedding is a method of converting categorical tokens into numeric vectors.



Sequence-to-Sequence Translation Using Attention

- For the decoder model, this example uses a network very similar to the encoder that contains two LSTMs. However, an important difference is that the decoder contains an attention mechanism. The attention mechanism allows the decoder to attend to specific parts of the encoder output.



Load Training Data

- Download the decimal-Roman numeral pairs from "romanNumerals.csv"

```
filename = fullfile("romanNumerals.csv");
options = detectImportOptions(filename, ...
    'TextType','string', ...
    'ReadVariableNames',false);
options.VariableNames = ["Source" "Target"];
options.VariableTypes = ["string" "string"];
data = readtable(filename,options);
```

Roman Number

Individual decimal places

	Thousands	Hundreds	Tens	Units
1	M	C	X	I
2	MM	CC	XX	II
3	MMM	CCC	XXX	III
4		CD	XL	IV
5		D	L	V
6		DC	LX	VI
7		DCC	LXX	VII
8		DCCC	LXXX	VIII
9		CM	XC	IX

Load Training Data

- Split the data into training and test partitions containing 50% of the data each.

```
idx = randperm(size(data,1),500);
dataTrain = data(idx,:);
dataTest = data;
dataTest(idx,:) = [];
```

Load Training Data

- View some of the decimal-Roman numeral pairs.

head(dataTrain)

ans=8×2 table

Source	Target
"228"	"CCXXVIII"
"267"	"CCLXVII"
"294"	"CCXCIV"
"179"	"CLXXIX"
"396"	"CCCXCVI"
"2"	"II"
"4"	"IV"
"270"	"CCLXX"

Preprocess Data

- Preprocess the text data using the transformText function. The transformText function preprocesses and tokenizes the input text for translation by splitting the text into characters and adding start and stop tokens. To translate text by splitting the text into words instead of characters, skip the first step.

```
startToken = "<start>";
stopToken = "<stop>";
strSource = dataTrain{:,1};
documentsSource = transformText(strSource,startToken,stopToken);
```

Preprocess Data

- Create a wordEncoding object that maps tokens to a numeric index and vice-versa using a vocabulary.

```
encSource = wordEncoding(documentsSource);
```

- Using the word encoding, convert the source text data to numeric sequences.

```
sequencesSource = doc2sequence(encSource, documentsSource,'PaddingDirection','none');
```

Preprocess Data

- Convert the target data to sequences using the same steps.

```
strTarget = dataTrain{:,2};
documentsTarget = transformText(strTarget,startToken,stopToken);
encTarget = wordEncoding(documentsTarget);
sequencesTarget = doc2sequence(encTarget, documentsTarget,'PaddingDirection','none');
```

Initialize Model Parameters

- Initialize the model parameters. for both the encoder and decoder, specify an embedding dimension of 128, two LSTM layers with 200 hidden units, and dropout layers with random dropout with probability 0.05.

```
embeddingDimension = 128;
numHiddenUnits = 200;
dropout = 0.05;
```

Initialize Encoder Model Parameters

- Initialize the weights of the encoding 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.

```
inputSize = encSource.NumWords + 1;
sz = [embeddingDimension inputSize];
mu = 0;
sigma = 0.01;
parameters.encoder.emb.Weights = initializeGaussian(sz,mu,sigma);
```

Initialize Encoder 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 Encoder Model Parameters

- Initialize the learnable parameters for the first encoder LSTM operation.

```
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 Encoder Model Parameters

- Initialize the learnable parameters for the second encoder LSTM operation.

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

parameters.encoder.lstm2.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.encoder.lstm2.RecurrentWeights =
initializeOrthogonal([4*numHiddenUnits numHiddenUnits]);
parameters.encoder.lstm2.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize Decoder Model Parameters

- Initialize the weights of the encoding embedding using the Gaussian using the initializeGaussian function. Specify a mean of 0 and a standard deviation of 0.01.

```
outputSize = encTarget.NumWords + 1;
sz = [embeddingDimension outputSize];
mu = 0;
sigma = 0.01;
parameters.decoder.emb.Weights = initializeGaussian(sz,mu,sigma);
```

Initialize Decoder Model Parameters

- Initialize the weights of the attention mechanism using the Glorot initializer using the initializeGlorot function.

```
sz = [numHiddenUnits numHiddenUnits];
numOut = numHiddenUnits;
numIn = numHiddenUnits;
parameters.decoder.attn.Weights =
initializeGlorot(sz,numOut,numIn);
```

Initialize Decoder Model Parameters

- Initialize the learnable parameters for the decoder LSTM operations:
  - 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 Decoder Model Parameters

- Initialize the learnable parameters for the first decoder LSTM operation.

```
sz = [4*numHiddenUnits embeddingDimension+numHiddenUnits];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension + numHiddenUnits;
parameters.decoder.lstm1.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.decoder.lstm1.RecurrentWeights =
initializeOrthogonal([4*numHiddenUnits numHiddenUnits]);
parameters.decoder.lstm1.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize Decoder Model Parameters

- Initialize the learnable parameters for the second decoder LSTM operation.

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

parameters.decoder.lstm2.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.decoder.lstm2.RecurrentWeights =
initializeOrthogonal([4*numHiddenUnits numHiddenUnits]);
parameters.decoder.lstm2.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize Decoder Model Parameters

- Initialize the learnable parameters for the decoder fully connected operation:
  - Initialize the weights with the Glorot initializer.
  - Initialize the bias with zeros using the initializeZeros function which is attached to this example as a supporting file. To learn more, see Zeros Initialization.

```
sz = [outputSize 2*numHiddenUnits];
numOut = outputSize;
numIn = 2*numHiddenUnits;
parameters.decoder.fc.Weights = initializeGlorot(sz,numOut,numIn);
parameters.decoder.fc.Bias = initializeZeros([outputSize 1]);
```

Define Model Functions

- Create the functions modelEncoder and modelDecoder, that compute the outputs of the encoder and decoder models, respectively.
- The modelEncoder function, takes the input data, the model parameters, the optional mask that is used to determine the correct outputs for training and returns the model outputs and the LSTM hidden state.
- The modelDecoder function, takes the input data, the model parameters, the context vector, the LSTM initial hidden state, the outputs of the encoder, and the dropout probability and outputs the decoder output, the updated context vector, the updated LSTM state, and the attention scores.

Define Model Encoder Function

- The function modelEncoder takes the input data, the model parameters, the optional mask that is used to determine the correct outputs for training and returns the model output and the LSTM hidden state.
- If sequenceLengths is empty, then the function does not mask the output. Specify and empty value for sequenceLengths when using the modelEncoder function for prediction.

Define Model Encoder Function

```
function [dlZ, hiddenState] = modelEncoder(parametersEncoder, dlX, sequenceLengths)
    % Embedding.
    weights = parametersEncoder.emb.Weights; # [128 13]
    dlZ = embed(dlX,weights,'DataFormat','CBT'); # [128 32 5]
    % LSTM 1.
    inputWeights = parametersEncoder.lstm1.InputWeights; # [800 128]
    recurrentWeights = parametersEncoder.lstm1.RecurrentWeights; # [800 200]
    bias = parametersEncoder.lstm1.Bias; # [800 1]
    numHiddenUnits = size(recurrentWeights, 2); # 200
    initialHiddenState = dlarray(zeros([numHiddenUnits 1])); # [200 1]
    initialCellState = dlarray(zeros([numHiddenUnits 1])); # [200 1]
    dlZ = lstm(dlZ, initialHiddenState, initialCellState, inputWeights, ... # [200 32 5]
        recurrentWeights, bias, 'DataFormat', 'CBT');
```

```
    % LSTM 2.
    inputWeights = parametersEncoder.lstm2.InputWeights; # [800 200]
    recurrentWeights = parametersEncoder.lstm2.RecurrentWeights; # [800 200]
    bias = parametersEncoder.lstm2.Bias; # [800 1]
    [dlZ, hiddenState] = lstm(dlZ,initialHiddenState, initialCellState, ...
        inputWeights, recurrentWeights, bias, 'DataFormat', 'CBT'); # [200 32 5] [200 32]
    % Masking for training.
    if ~isempty(sequenceLengths)
        miniBatchSize = size(dlZ,2);
        for n = 1:miniBatchSize
            hiddenState(:,n) = dlZ(:,n,sequenceLengths(n));
        end
    end
end
```

Define Model Decoder Function

- The function modelDecoder takes the input data, the model parameters, the context vector, the LSTM initial hidden state, the outputs of the encoder, and the dropout probability and outputs the decoder output, the updated context vector, the updated LSTM state, and the attention scores.

Define Model Decoder Function

```
function [dlY, context, hiddenState, attentionScores] =
            modelDecoder(parametersDecoder, dlX, context, hiddenState, dlZ, dropout)


  % Embedding.
  weights = parametersDecoder.emb.Weights; # [128 10]
  dlX = embed(dlX, weights,'DataFormat','CBT'); # [128 32 11]


  % RNN input.
  sequenceLength = size(dlX,3);
  dlY = cat(1, dlX, repmat(context, [1 1 sequenceLength])); # [328 32 11]
  # context [200 32 11]
```

```
% LSTM 1.
inputWeights = parametersDecoder.lstm1.InputWeights; # [800 328]
recurrentWeights = parametersDecoder.lstm1.RecurrentWeights; # [800 200]
bias = parametersDecoder.lstm1.Bias; # [800 1]
initialCellState = dlarray(zeros(size(hiddenState))); # [200 32]
dlY = lstm(dlY, hiddenState, initialCellState, inputWeights, recurrentWeights, bias,
                    'DataFormat', 'CBT'); # [200 32 11]
% Dropout.
mask = ( rand(size(dlY), 'like', dlY) > dropout );
dlY = dlY.*mask;
% LSTM 2.
inputWeights = parametersDecoder.lstm2.InputWeights; # [800 200]
recurrentWeights = parametersDecoder.lstm2.RecurrentWeights; # [800 200]
bias = parametersDecoder.lstm2.Bias; # [800 1]
[dlY, hiddenState] = lstm(dlY, hiddenState, initialCellState,inputWeights, recurrentWeights,
                                    bias, 'DataFormat', 'CBT');
```

Define Model Decoder Function

```
    % Attention.
    weights = parametersDecoder.attn.Weights; # [200 200]
    [attentionScores, context] = attention(hiddenState, dlZ, weights);# [5 1 32] [200 32]
    % Concatenate.
    dlY = cat(1, dlY, repmat(context, [1 1 sequenceLength])); # [200+200 32 11]
    % Fully connect.
    weights = parametersDecoder.fc.Weights; # [10 400]
    bias = parametersDecoder.fc.Bias; # [10 1]
    dlY = fullyconnect(dlY,weights,bias,'DataFormat','CBT'); # [10 32 11]
    % Softmax.
    dlY = softmax(dlY,'DataFormat','CBT'); # [10 32 11]
end
```

Define Attention Function

- The attention function returns the attention scores according to Luong "general" scoring and the updated context vector. The energy at each time step is the dot product of the hidden state and the learnable attention weights times the encoder output.

Define Attention Function

```
function [attentionScores, context] = attention(hiddenState, encoderOutputs, weights)
  % Initialize attention energies.
  [miniBatchSize, sequenceLength] = size(encoderOutputs, 2:3); # encoderOutputs [200 32 5]
  attentionEnergies = zeros([sequenceLength miniBatchSize],'like',hiddenState); # [5 32]
  % Attention energies. hiddenState [200 32]; weights [200 200]
  hWX = hiddenState .* pagemtimes(weights,encoderOutputs); # [200 32 5]
  for tt = 1:sequenceLength
     attentionEnergies(tt, :) = sum(hWX(:, :, tt), 1);
  end
  % Attention scores.
  attentionScores = softmax(attentionEnergies, 'DataFormat', 'CB'); # [5 32]
```

Define Attention Function

```
    % Context.
    encoderOutputs = permute(encoderOutputs, [1 3 2]); # [200 5 32]
    attentionScores = permute(attentionScores,[1 3 2]); # [5 1 32]
    context = pagemtimes(encoderOutputs,attentionScores); # [200 1 32]
    context = squeeze(context); # [200 32]
end
```

Define Model Gradients Function

- The modelGradients function takes the encoder and decoder model parameters, a mini-batch of input data and the padding masks corresponding to the input data, and the dropout probability and returns the gradients of the loss with respect to the learnable parameters in the models and the corresponding loss.

Define Model Gradients Function

```
function [gradients, loss] = modelGradients(parameters, dlX, T, ...
        sequenceLengthsSource, sequenceLengthsTarget, dropout)
    % Forward through encoder.
    [dlZ, hiddenState] = modelEncoder(parameters.encoder, dlX, sequenceLengthsSource);
    % Decoder Output.
    doTeacherForcing = rand < 0.5;
    sequenceLength = size(T,3);
    dlY = decoderPredictions(parameters.decoder,dlZ,T,hiddenState,dropout,...
            doTeacherForcing,sequenceLength);
```

Define Model Gradients Function

```
    % Masked loss.
    dlY = dlY(:,:,1:end-1);
    T = T(:,:,2:end);
    T = onehotencode(T,1,'ClassNames',1:size(dlY,1));
    loss = maskedCrossEntropy(dlY,T,sequenceLengthsTarget-1);
    % Update gradients.
    gradients = dlgradient(loss, parameters);
    % For plotting, return loss normalized by sequence length.
    loss = extractdata(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 loss = maskedCrossEntropy(dlY,T,sequenceLengths)
  % Initialize loss.
  loss = 0;
  % Loop over mini-batch.
  miniBatchSize = size(dlY,2);
  for n = 1:miniBatchSize
     idx = 1:sequenceLengths(n);
     loss = loss + crossentropy(dlY(:,n,idx), T(:,n,idx),'DataFormat','CBT');
  end
  % Normalize.
  loss = loss / miniBatchSize;
end
```

Specify Training Options

- Train with a mini-batch size of 32 for 75 epochs with a learning rate of 0.002.

```
miniBatchSize = 32;
numEpochs = 75;
learnRate = 0.002;
```

- Initialize the options from Adam.

```
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Train Network

- Train the model using a custom training loop.
- Train with the sequences sorted by increasing sequence length. This results in batches with sequences of approximately the same sequence length and ensures smaller sequence batches are used to update the model before longer sequence batches.
- Sort the sequences by length.

```
sequenceLengths = cellfun(@(sequence) size(sequence,2), sequencesSource);
[~,idx] = sort(sequenceLengths);
sequencesSource = sequencesSource(idx);
sequencesTarget = sequencesTarget(idx);
```

Train Network

- Initialize the training progress plot.

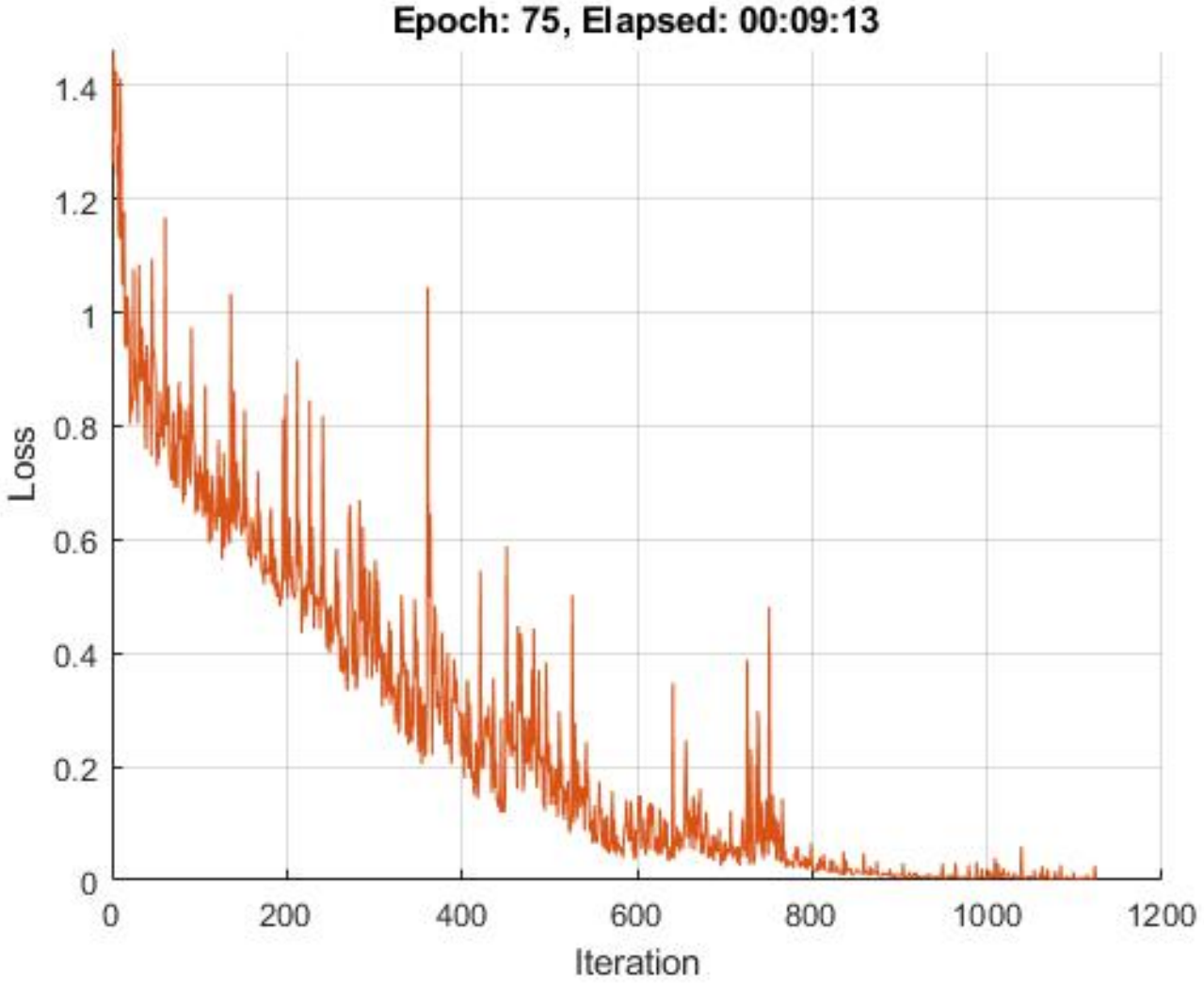
```
figure
lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

- Initialize the values for the adamupdate function.

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

Train Network

- Train the model. For each mini-batch:
  - Read a mini-batch of sequences and add padding.
  - Convert the data to dlarray.
  - Compute loss and gradients.
  - Update the encoder and decoder model parameters using the adamupdate function.
  - Update the training progress plot.


Epoch: 75, Elapsed: 00:09:13
Loss
Iteration
0
0.2
0.4
0.6
0.8
1
1.2
1.4
0
200
400
600
800
1000
1200

Generate Translations

- To generate translations for new data using the trained model, convert the text data to numeric sequences using the same steps as when training and input the sequences into the encoder-decoder model and convert the resulting sequences back into text using the token indices.
- Preprocess the text data using the same steps as when training. Use the transformText function, listed at the end of the example, to split the text into characters and add the start and stop tokens.

```
strSource = dataTest{:,1};
strTarget = dataTest{:,2};
```

Generate Translations

- Translate the text using the modelPredictions function.

```
maxSequenceLength = 10;
delimiter = "";
strTranslated =
translateText(parameters,strSource,maxSequenceLength,miniBatchSize, ...
    encSource,encTarget,startToken,stopToken,delimiter);
```

Generate Translations

- Create a table containing the test source text, target text, and translations.

```
tbl = table;
tbl.Source = strSource;
tbl.Target = strTarget;
tbl.Translated = strTranslated;
```

Generate Translations

- View a random selection of the translations.

```
idx = randperm(size(dataTest,1),miniBatchSize);
tbl(idx,:)
```

ans=32×3 table

Source	Target	Translated
"936"	"CMXXXVI"	"CMXXXVI"
"423"	"CDXXIII"	"CDXXIII"
"981"	"CMLXXXI"	"CMLXXXIX"
"200"	"CC"	"CC"
"224"	"CCXXIV"	"CCXXIV"
"56"	"LVI"	"DLVI" .....

Text Transformation Function

- The transformText function preprocesses and tokenizes the input text for translation by splitting the text into characters and adding start and stop tokens. To translate text by splitting the text into words instead of characters, skip the first step.

```
function documents = transformText(str,startToken,stopToken)
  str = strip(replace(str,"","" ""));
  str = startToken + str + stopToken;
  documents = tokenizedDocument(str,'CustomTokens',[startToken stopToken]);
end
```

Sequence Padding Function

- The padSequences function takes a mini-batch of sequences and a padding value and returns padded sequences with the corresponding padding masks.

```
function [X, sequenceLengths] = padSequences(sequences, paddingValue)
  % Initialize mini-batch with padding.
  numObservations = size(sequences,1);
  sequenceLengths = cellfun(@(x) size(x,2), sequences);
  maxLength = max(sequenceLengths);
  X = repmat(paddingValue, [1 numObservations maxLength]);
  % Insert sequences.
  for n = 1:numObservations
    X(1,n,1:sequenceLengths(n)) = sequences{n};
  end
end
```