ATTENTION IN NEURAL NETWORKS

Attention Technique in Seq2Seq:

Sequence to sequence using RNNs modelling is used in -

Neural Machine Translation

Speech Recognition

Text Summarization

Image Captioning

Chatbots and

Other sequence modelling tasks

Need for Attention:

In tradition sequence to sequence model If the sequence is large It can forgets the previous words it is not able to retain all the information and interaction between the entities resulting in poor accuracy. So comes the

Attention model :

It pay attention to the part of a input sentence while generating a translation where as the sequence to sequence model uses Encoder Decoder formulation for translation .

Attention initially developed for Machine translation later it extended for many areas as well.

Idea:

In sequence to sequence model we used to discard the intermediate states(hidden and cell states) and we took only the final state which comprised of all compressed vector (Final vector/Context Vector/Thought Vector) of all the words together which later fed into decoder system.

So instead of discarding all the intermediate state which contains the essence of till read text Attention model takes them into consideration which are the vectors used by the decoder to generate the output sequence.

Long Sequence Problem: Attention Intuition

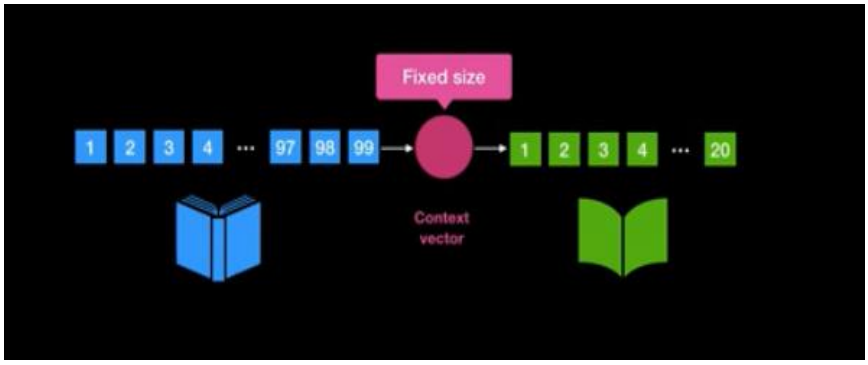
In Encoder Decoder model it first tries to learn the whole sequence and then start translating them which we can think as if a human tries to translate like this there is a maximum chance that it will lose some words or information.

But, Lets look at the case how human translator will actually do ,he will first see the part of sentence and analyse what are the important words in that and if the word is important then he may translate that part or ignore it .Such way of translation and reading of the next part and again translates them and so on.... is followed till the whole statement is traversed .

Because it is difficult to memorize whole long sentence.Attention model leverages this which ultimately outperforms the classical one.

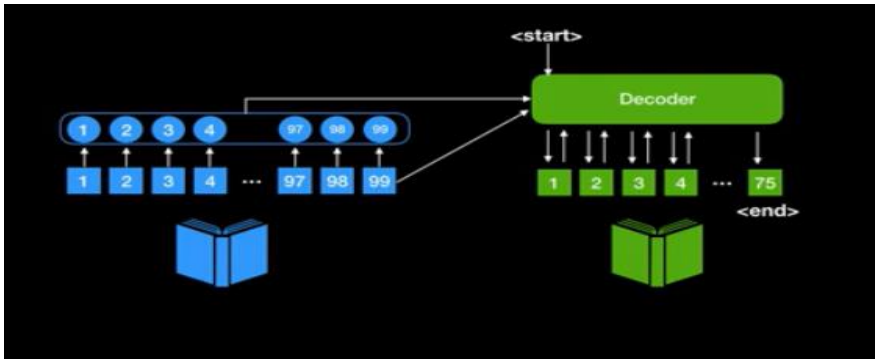
a) Traditional Seq2Seq

Context vector may not have all the necessary info if input with long sentence.



b) With Attention

Rather than using the fixed final context vector we can use output state of each encoder to generate context vector.



Mechanism:

Example : Translation from English to French

English : *I am a student*

French : *Je suis estudente*

I am a student -> After data preprocessing it has to be *<start>* I am a student*<end>* to know sentence is starting and ending.

After the input is passed through the encoder return the encoder hidden state and encoder output

That is passed to decoder input

Weighted Vector

[I am a student] + [<start>] -> [12,3,4,…….] -> Softmax ->

Context Vector

[**0.95**,0.0.01,0.02,0.001,……]

After getting <start> into decoder probability for next word **'Je'** is higher so the output will be **'Je'**

At next time step ,

Probability of word **'suis'** will be higher like -> [0.2, **0.75**,0.01,0.02,0.001,……]

So on

It will be passed into the decoder which is learning to predict the translated word.

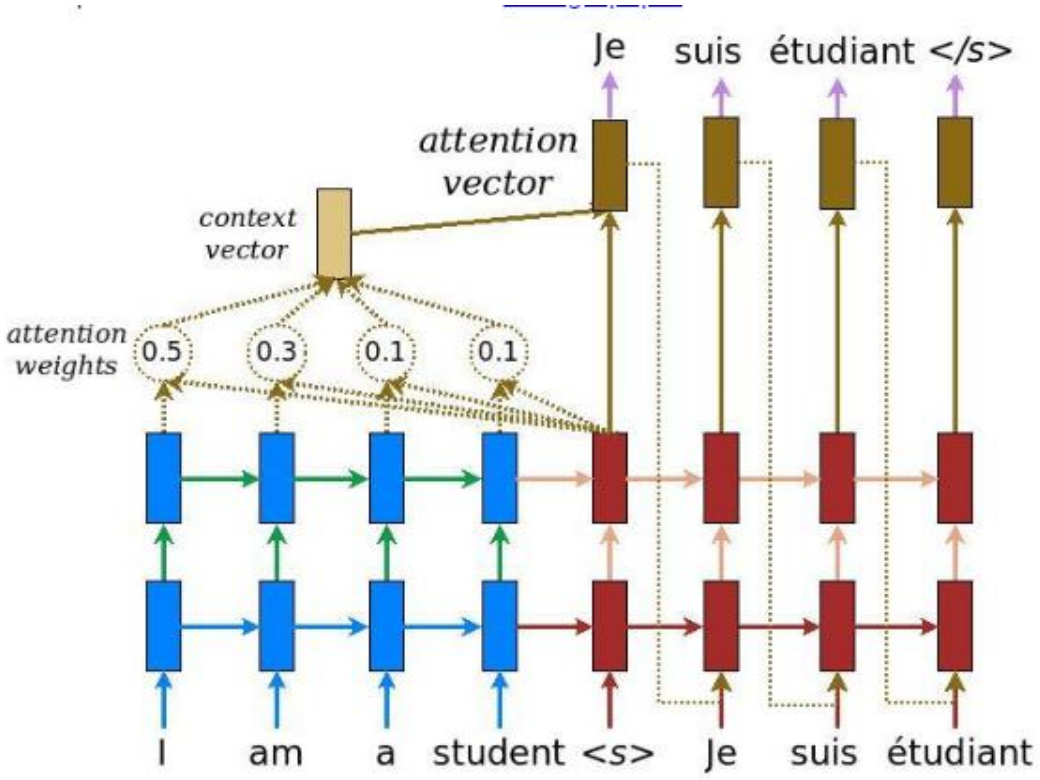
Implementation Guidelines:

1.Preprocess the data and after preprocessing and cleaning the data create each sample point of both English and Spanish in the below format .

<START> I am a student <END> Input tensor

<START> Je suis estudiant <END> Target Tensor

Create word Embedding for them and slot them into batches and store them into dataset



(Figure: Loung Attention)

Shapes:

Batch Size: 64 **Input Tensor** : 64*16 **Target Tensor** : 64*14 (14 may be maximum length of words in target)

Take GRU as Recurrent cell as it has only 1 hidden layer to avoid complexity (tf.keras.layers.GRU)

Here take no of GRU Units = 50

Here we are taking Banhadau Attention .There is one more which is called Luong Attention.

Both differs in only how Context Vector is created .

2. **Encoder :**

output, state = gru(Input_batch , hidden) //Initially Hidden =[0 0 0 0 ……. 50^{th}]

encoder(input_batch, hidden) // calling

```
class Encoder(tf.keras.Model):
  def call(self, x, hidden):
    x = self.embedding(x)
    output, state = self.gru(x, initial_state = hidden)
    return output, state
```

```
encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)
```

3. **Attention:**

Attention(query,value)

Score = V(tanh (W1 (values) + W2 (query)))

// Before passing values to W1 Change the dimensions

// W1 W2: Fully Connected Layer with n neurons V: Dense Layer : 1 neuron which gives

attention_weights = softmax(score)

context_vector = attention_weights * values

context_vector = tf.reduce_sum(context_vector, axis=1)

```
class Attention(tf.keras.Model):
    hidden_with_time_axis = tf.expand_dims(query, 1)
    score = self.V(tf.nn.tanh(
        self.W1(values) + self.W2(hidden_with_time_axis)))
    attention_weights = tf.nn.softmax(score, axis=1)
    context_vector = attention_weights * values
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector, attention_weights
```

Shapes:

Query : Hidden layer : 64* 50 (Units)

Values : Output layer of Encoder : 64 * 16 * 50(Units)

Then you might wondering how we are adding the output from Fully connect dense layer that is where we are increasing the dimension of query vector by 1.

Attention weights : 64*16*1

Context vector : 64*50

// We are using tf.reduce sum to reduce context vector from size 64*16*50 to 64*50

4. **Decoder**

Decoder(x , hidden, encoder_output)

GRU()

context_vector, attention_weights = Attention(hidden , encoder_output)

x=embedding(x)

x = tf.concat ([tf.expand_dims(context_vector, 1), x], axis=-1)

output, state = GRU(x)

output = tf.reshape(output, (-1, output.shape[2]))

x = FC (output)

return x

```
class Decoder(tf.keras.Model):
    context_vector, attention_weights = self.attention(hidden, enc_output)
    x = self.embedding(x)
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)
    output, state = self.gru(x)
    output = tf.reshape(output, (-1, output.shape[2]))
    x = self.fc(output)
    return x, state, attention_weights
```

Shape :

Encoder_output : 64*16*50

Output shape : 64, vocab_size

4 Driver :

Foreach in epoch:

encoder_hidden = [0 0 0 0 …… 50th] //Initialized with 0

Foreach (batch ,input,target) in dataset:

encoder_output, encoder_hidden = encoder(input,encoder_hidden)

decoder_hidden = encoder_hidden

decoder_input = target(<START>) * 64 *1

For i in target.shape[1]:

predictions, decoder_hidden, _ = decoder(decoder_input, decoder_hidden, enc_output)

dec_input = tf.expand_dims(target[:, i], 1)

// for each i it is taking it is increasing the count to predict the next one.

```
for epoch in range(EPOCHS):
  start = time.time()

  enc_hidden = encoder.initialize_hidden_state()
  total_loss = 0

  for (batch, (inp, targ)) in enumerate(dataset.take(steps_per_epoch)):
    batch_loss = train_step(inp, targ, enc_hidden)

    loss = 0
    with tf.GradientTape() as tape:
      enc_output, enc_hidden = encoder(inp, enc_hidden)
      dec_hidden = enc_hidden
      dec_input = tf.expand_dims([targ_lang.word_index['<start>']] * BATCH_SIZE, 1)
    # Teacher forcing - feeding the target as the next input
      for t in range(1, targ.shape[1]):
        # passing enc_output to the decoder
        predictions, dec_hidden, _ = decoder(dec_input, dec_hidden, enc_output)
        loss += loss_function(targ[:, t], predictions)
        # using teacher forcing
        dec_input = tf.expand_dims(targ[:, t], 1)

    batch_loss = (loss / int(targ.shape[1]))
    variables = encoder.trainable_variables + decoder.trainable_variables
    gradients = tape.gradient(loss, variables)
    optimizer.apply_gradients(zip(gradients, variables))

  total_loss += batch_loss
```

5.Validation

```
1 translate(u'I am  student.')
```

```
Input: <start> i am student . <end>
Predicted translation: je suis etudiant . <end>
```

Input : <start> I am a student. <end>

Input = word embedding vector , Hidden = Initialized with Zeros

enc_out, enc_hidden = encoder(inputs, hidden)

decoder_hidden = encoder_hidden

dec_input = <START> // Word index of START

for i (maximum_length of target) :

predictions, dec_hidden, attention_weights = decoder(dec_input, decoder_hidden, enc_out)

predicted_ID = tf.argmax(prediction[0]).numpy()

result + = Index_word(predicted_ID)

return result

```python
def translate(sentence):
    sentence = preprocess_sentence(sentence)

    inputs = [inp_lang.word_index[i] for i in sentence.split(' ')]
    inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs],maxlen=max_length_inp,padding='post')
    inputs = tf.convert_to_tensor(inputs)
    result = ''
    hidden = [tf.zeros((1, units))]
    enc_out, enc_hidden = encoder(inputs, hidden)
    dec_hidden = enc_hidden
    dec_input = tf.expand_dims([targ_lang.word_index['<start>']], 0)
    for t in range(max_length_targ):
        predictions, dec_hidden, attention_weights = decoder(dec_input,dec_hidden,enc_out)
        predicted_id = tf.argmax(predictions[0]).numpy()
        result += targ_lang.index_word[predicted_id] + ' '
        if targ_lang.index_word[predicted_id] == '<end>':
            return result, sentence, attention_plot
        # the predicted ID is fed back into the model
        dec_input = tf.expand_dims([predicted_id], 0)
    result, sentence, attention_plot = evaluate(sentence)
    print('Input: %s' % (sentence))
    print('Predicted translation: {}'.format(result))
```

```python
translate(u'I am a good boy')
```

```
Input: <start> i am a good boy <end>
Predicted translation: je suis bien bon . <end>
```

Steps once more:

Pass the input through encoder now we have encoder output and encoder hidden states .Now the encoder output encoder hidden states is passed through the decoder input

Now at decoder Input we have || <START> || It will try to predict **Je**

We have now prediction and decoder hidden state now this decoder hidden state is passed through next cell to predict next word i.e. **suis** . Instead of passing prediction vector we pass the true label so the model learn fast and less prone to error this is **Teacher Forcing**.

Attention Technique in CNN:

To explain use of attention technique in CNN we would take a example **Image Captioning**

While ConvNets are use for signal processing and Image Detection/Classification they are not good in pattern recognitions .

Recurrent Neural Networks is there to learn pattern through their feed forward network .

Image Captioning

Looking at the image we have to return a free form text what image is all about . We used to take the output from final layer(Fully connected) which contained the feature of the image which is fed into the RNN network as a hidden state in previous time step which will generate the first word and this output goes again to RNN to generate second word on the next time step and so on.... .Second part is same as Machine Translation part .

Problem:

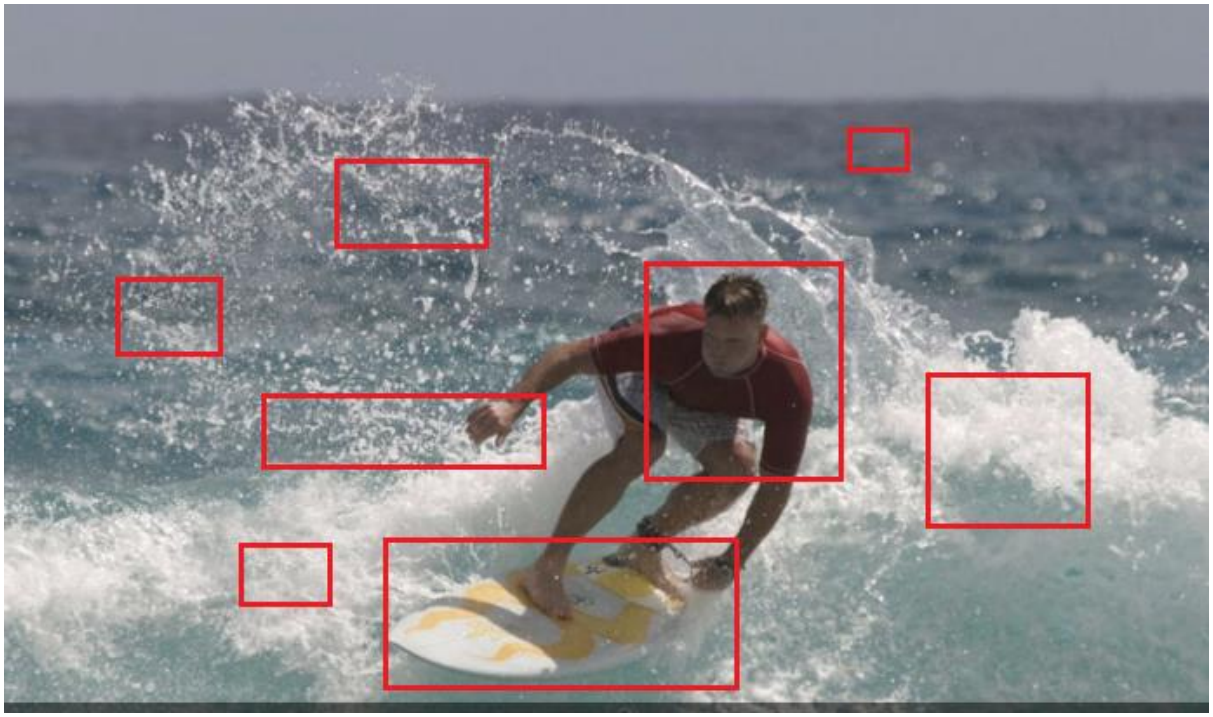
We used the output from final layer into RNN which contained the overall summary of the image in form of vector .Here the RNN does not see the whole image but only final feature generated.

Idea:

Instead of taking final feature output of convolution layer should be taken which will act as hidden state vector in

previous time step to generate the first word and rest all same.

Intutively, initially RNN was looking whole image at once(as a result from final Fully Connected Layer) and now RNN looks at different part of image at each time step.



The part to which more attention to be given to get a most accurate caption. Man,Surfboard, Water Splashes , Ocean , water etc can be inferred from the image.

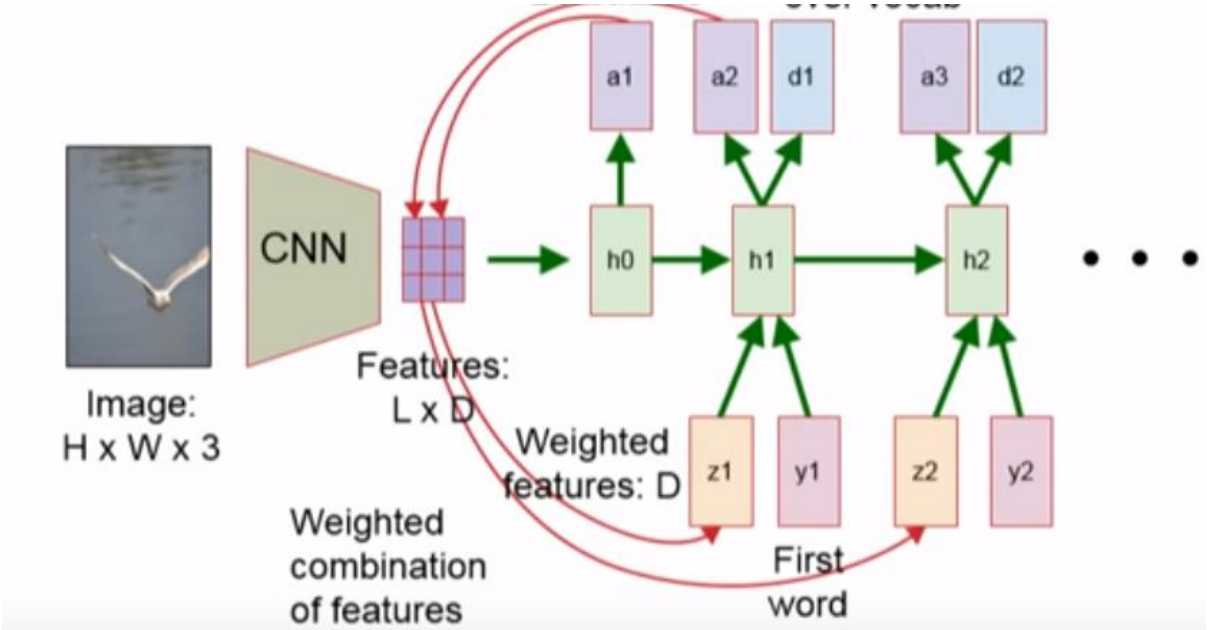


Illustration from Figure:

e.g. Output from CNN 14*14*256 (256 – channels/filters) : **L*D** is 196 * 256 – Feature Matrix is passed as hidden state into RNN which gives output in the form 14*14 (**a1**).This matrix shows where to look closely.

a1 matrix that we get from RNN contains the probabilities values where to look probabilistically or which part to give more focus , which is multiplied to each channel values from such that vector we get is the weighted valued vector (**z1**) of size 1*256 .

This **a1** multiplied with feature matrix to give weighted sum vector of size 1*256 **(z1)** .

y1(Word Embedding Vector**)** and **z1** is passed to next RNN to generate **a2** and **d1** (output word vector) . So the process followed the same way.

As here only one hidden state assume it as GRU

hi : Hidden state

yi : Initial Word (Embedding Vector)

zi : Weighted Sum / Context Vector (here, 256 dimensional vector –depth from convolved Layer

ai : Output generated by RNN (here, 14 * 14) such that each block contains a number between 0-1

Sum of all values = 1 , Cell which have more value gives more attention to that block.

Explanation through code

We are using InceptionV3 model which is pre trained on Imagenet and it is available in keras.

Input Shape: 299*299

We want output from last but new layer from the Inception model

```
image_model = tf.keras.applications.InceptionV3(include_top=False,
                                                 weights='imagenet')
new_input = image_model.input
hidden_layer = image_model.layers[-1].output

image_features_extract_model = tf.keras.Model(new_input, hidden_layer)
```

The output from the convolution layer is of Size = (8,8,2048) compress => 64 * 2048

For caption we used word embedding technique

Size for each image and caption pair I have:

Img_vector : 64 * 2048

Target _vector : 40

Create a batch of 32 items wrap it into tensor and store it into a **dataset**.

Batch 0 : Img_tensor : 32 * 64 *2048 target_tensor : 32*40

Driver :

for each epoch:

for each batch in dataset:

*hidden_val = zeros(batch_size) , decoder input = word_index['<start>']*batch_size,*

features = encoder(img_tensor)

for each value in target.shape[1]:

// Each time hidden layer is updated and goes into the model

prediction , hidden_val , _ = decoder(dec_input,features,hidden)

loss += loss_function(target[:, i], predictions)

//**Teacher Forcing** : Instead of giving predicted value as next input we are passing true value to //next RNN Cell this helps in faster learning and less mistake

decoder_input = tf.expand_dims(target[:, i], 1) // Each time we are expanding

Here comes the core of the whole algorithm:

Code part is mostly similar to Neural Machine Translation instead we have to give input the vector from convolution layer from Inception model.

Encoder

It takes the img_tensor apply Dense layer and Activation on top and pass it into Decoder .

```
class CNN_Encoder(tf.keras.Model):
    def __init__(self, embedding_dim):
        # shape after fc == (batch_size, 64, embedding_dim)
        self.fc = tf.keras.layers.Dense(embedding_dim)
    def call(self, x):
        x = self.fc(x)
        x = tf.nn.relu(x)
        return x
```

Decoder

Contect_vector ,attention_weigts = Attention(features,hidden)

//Bahandau :- softmax(tanh(features,hidden))

x = concat(x,contex_vector) //x is decoder input

Output , state = GRU(x)

x=Dense_layer(output)

return x,state,attention_weights

```
class RNN_Decoder(tf.keras.Model):
  def __init__(self, embedding_dim, units, vocab_size):
    context_vector, attention_weights = self.attention(features, hidden)
    x = self.embedding(x)
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)
    output, state = self.gru(x)
    x = self.fc1(output)
    x = tf.reshape(x, (-1, x.shape[2]))
    x = self.fc2(x)
    return x, state, attention_weights
```

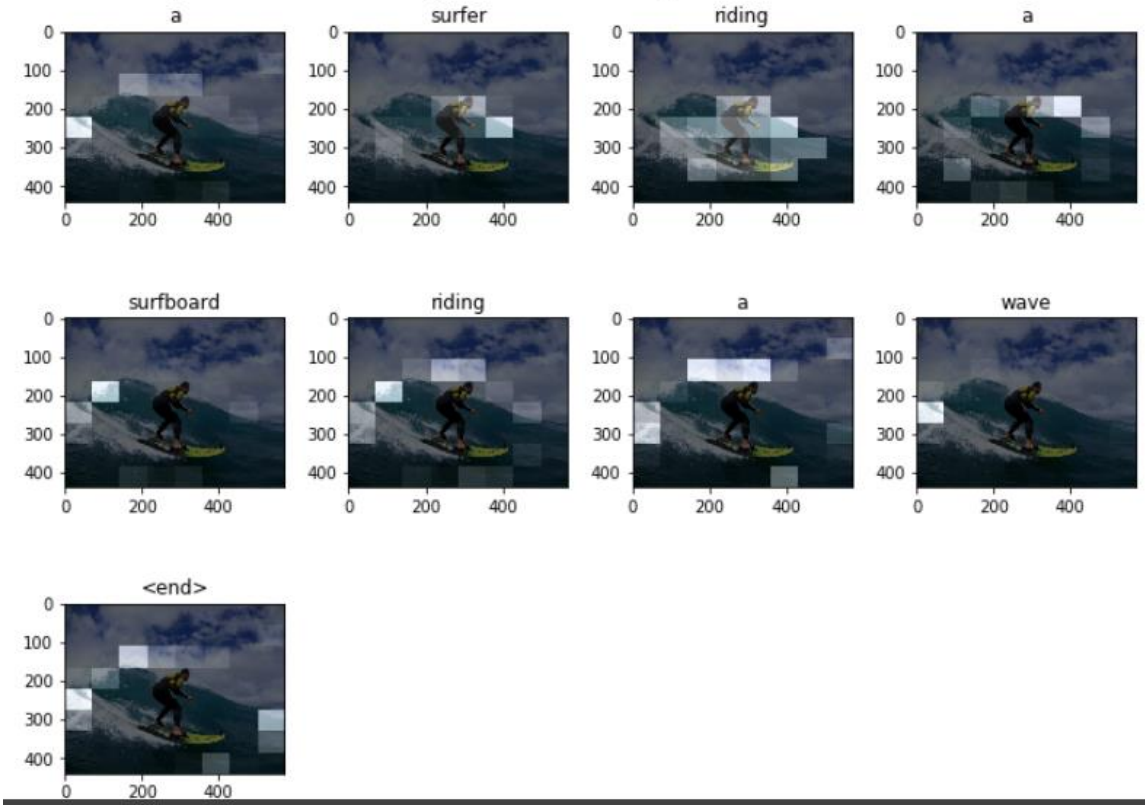
Validation

Given a image extract the tensor of required dimension and the model starts with <START> to predict the next word describing the picture.

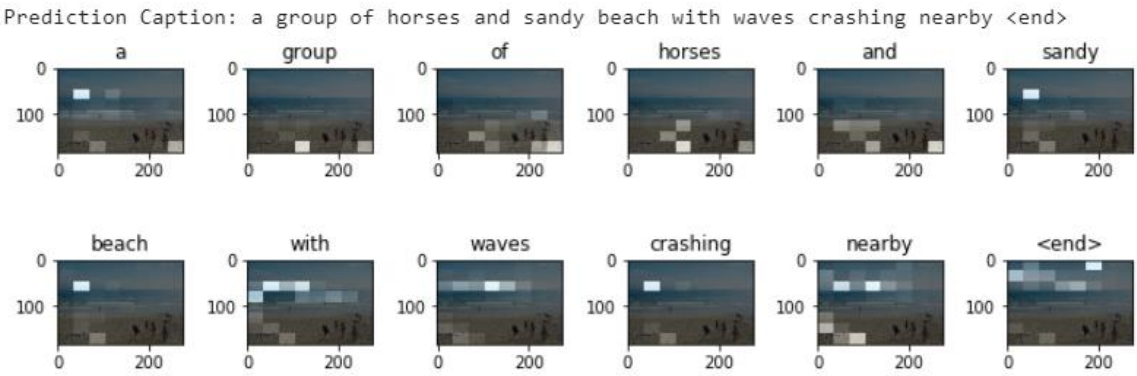
```
def evaluate(image):
    hidden = decoder.reset_state(batch_size=1)
    img_tensor_val = image_features_extract_model(image)   #Image tensor
    features = encoder(img_tensor_val)
    dec_input = tf.expand_dims([tokenizer.word_index['<start>']], 0)
    result = []
    for i in range(max_length):  #Max length Maximum Length of any caption
        predictions, hidden, attention_weights = decoder(dec_input, features, hidden)
        predicted_id = tf.argmax(predictions[0]).numpy()
        result.append(tokenizer.index_word[predicted_id])
        if tokenizer.index_word[predicted_id] == '<end>':
              break
        dec_input = tf.expand_dims([predicted_id], 0)
    print ('Real Caption:', real_caption)
    print ('Prediction Caption:', ' '.join(result))
```

Let’s take the below image for validation and see which part is more focused by using attention mechanism.

```
Prediction Caption: a surfer riding a surfboard riding a wave <end>
```



As the model was trained on relatively small dataset so it is not surprising to get weird result.



Shapes:

Context_vector : 32* hidden_layer_shape

To know more about Bahandau and Loung attention : https://github.com/spro/practical-pytorch/blob/master/seq2seq-translation/seq2seq-translation.ipynb

Sources:

https://www.tensorflow.org/beta/tutorials/sequences/nmt_with_attention

https://www.tensorflow.org/beta/tutorials/text/image_captioning

https://zhuanlan.zhihu.com/p/31804728

https://machinelearningmastery.com/encoder-decoder-attention-sequence-to-sequence-prediction-keras/

https://towardsdatascience.com/light-on-math-ml-attention-with-keras-dc8dbc1fad39