# Neural Machine Translation with Attention

January 31, 2024

[39]:
```
### v1.1
```

[40]:
```
from tensorflow.keras.layers import Bidirectional, Concatenate, Permute, Dot,␣
↪Input, LSTM, Multiply
from tensorflow.keras.layers import RepeatVector, Dense, Activation, Lambda
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import load_model, Model
import tensorflow.keras.backend as K
import tensorflow as tf
import numpy as np

from faker import Faker
import random
from tqdm import tqdm
from babel.dates import format_date
from nmt_utils import *
import matplotlib.pyplot as plt
%matplotlib inline
```

[41]:
```
# The model built here can be used to translate from one language to another.␣
↪But Language Translation requires massive datasets
# and takes days of training on GPUs.
# So here we will perform a simpler 'date translation' task where the
# 'the 29th of August 1958' will be translated to '1958-08-29'
# '03/30/1968' will be translated to '1968-03-30'
# '24 JUNE 1987' will be translated to '1987-06-24'
# human readable date will be translated to machine readable date

m = 10000
dataset, human_vocab, machine_vocab, inv_machine_vocab = load_dataset(m)

# dataset is a list of tuples and each tuple is a pair of (input, output)
# 'human_vocab' is a dictionary mapping each character in␣
↪human_readable_date(input) to an integer-valued index
# 'machine_vocab' is a dictionary mapping each character in␣
↪machine_readable_date(output) to an integer-valued index
```

```
# There are 37 different characters in the 'human_vocab'. len(human_vocab) = 37
# There are 11 different characters in the 'machine_vocab'. len(machine_vocab)␣
↪= 11
```

```
100%|      | 10000/10000 [00:00<00:00, 24544.15it/s]
```

[42]:
```
dataset[:10]
```

[42]:
```
[('27 november 1980', '1980-11-27'),
 ('friday september 13 2019', '2019-09-13'),
 ('tuesday july 17 2018', '2018-07-17'),
 ('4/10/19', '2019-04-10'),
 ('wednesday april 27 1977', '1977-04-27'),
 ('tuesday december 6 1977', '1977-12-06'),
 ('01 sep 1991', '1991-09-01'),
 ('3 10 22', '2022-10-03'),
 ('tuesday july 20 1999', '1999-07-20'),
 ('wednesday january 29 1992', '1992-01-29')]
```

[43]:
```
Tx = 30
Ty = 10
X, Y, Xoh, Yoh = preprocess_data(dataset, human_vocab, machine_vocab, Tx, Ty)

print("X.shape:", X.shape)
print("Y.shape:", Y.shape)
print("Xoh.shape:", Xoh.shape)
print("Yoh.shape:", Yoh.shape)

# We set Tx = 30 which is the maximum length of the human readable date in our␣
↪dataset
# We set Ty = 10 as the machine_readable_date is exactly 10 characters long

# The following preprocessing is done

# X: a processed version of the human readable dates in the training set.
#    - Each character in X is replaced by an index (integer) mapped to the␣
↪character using human_vocab.
#    - Each date is padded to ensure a length of    using a special character␣
↪(< pad >).
#    - X.shape = (m, Tx) where m is the number of training examples in a batch.

# Y: a processed version of the machine readable dates in the training set.
#    - Each character is replaced by an index (integer) mapped to the character␣
↪using machine_vocab.
#    - Y.shape = (m, Ty)
```

```
# Xoh: one-hot version of X

# Yoh: one-hot version of Y
```

```
X.shape: (10000, 30)
Y.shape: (10000, 10)
Xoh.shape: (10000, 30, 37)
Yoh.shape: (10000, 10, 11)
```

[44]:

```
# Let's look at an example of the preprocessed training example

index = 0
print("Source date:", dataset[index][0])
print("Target date:", dataset[index][1])
print()
print("Source after preprocessing (indices):", X[index])
print("Target after preprocessing (indices):", Y[index])
print()
print("Source after preprocessing (one-hot):", Xoh[index])
print("Target after preprocessing (one-hot):", Yoh[index])
```

```
Source date: 27 november 1980
Target date: 1980-11-27

Source after preprocessing (indices): [ 5 10  0 25 26 32 17 24 14 17 28  0  4 12
11  3 36 36 36 36 36 36 36 36
 36 36 36 36 36 36]
Target after preprocessing (indices): [ 2 10  9  1  0  2  2  0  3  8]

Source after preprocessing (one-hot): [[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [1. 0. 0. … 0. 0. 0.]
 …
 [0. 0. 0. … 0. 0. 1.]
 [0. 0. 0. … 0. 0. 1.]
 [0. 0. 0. … 0. 0. 1.]]
Target after preprocessing (one-hot): [[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]
```

[45]:

```python
# Defined shared layers as global variables
# A vector of shape (m,n_s) becomes (m,Tx,n_s) after repeating
repeator = RepeatVector(Tx)


concatenator = Concatenate(axis=-1)
# concatenator(a,b) where a is of dimension (x,y,z1) and b is of dimension (x,y,z2).
# All the dimensions need to match except the last dimension
# The resultant is of dimension (x,y,z1+z2)

# The 'Dense' layer only transforms the last dimension.
# When you pass a tensor of size (x,y,z) through the 'densor1', the resultant is of size (x,y,10).
densor1 = Dense(10, activation = "tanh")
densor2 = Dense(1, activation = "relu")
# When you pass a tensor through a Dense layer, it multiplies the values in the last dimension of the input tensor with
# the layer's weights and adds the bias, followed by applying the activation function, resulting in a new last dimension as
# specified by the number of units in the Dense layer.

activator = Activation(softmax, name='attention_weights')
# We are using a custom softmax(axis = 1) loaded in this notebook

dotor = Dot(axes = 1)


# Note

# Question - The advantage of attention model is that it doesn't have to wait till processing the entire sentence and can
# start the language translation even after few words of the input sentence are processed right? but don't we take
# attention on x<1>, x<2>,....x<tx> all into consideration? Aren't we still processing the complete input sentence and
# then only y<1> is being produced right?

# Answer
# Models that don't use attention put equal emphasis on every part of the input sentence whereas models that
# use attention puts more emphasis on relevant parts of the input sentence. This is the only difference.
# Otherwise both models with and without attention does depend on the entire input sequence.
```

[46]:

```
def one_step_attention(a, s_prev):
    """
    Performs one step of attention: Outputs a context vector computed as a dot␣
↪product of the attention weights
    "alphas" and the hidden states "a" of the Bi-LSTM.

    Arguments:
    a -- hidden state output of the Bi-LSTM, numpy-array of shape (m, Tx, 2*n_a)
    s_prev -- previous hidden state of the (post-attention) LSTM, numpy-array␣
↪of shape (m, n_s)

    Returns:
    context -- context vector, input of the next (post-attention) LSTM cell
    """

    # s_prev is of shape (m, n_s)
    # Use 'repeator' to repeat s_prev to be of shape (m, Tx, n_s) so that we␣
↪can concatenate it with all hidden states "a"
    s_prev = repeator(s_prev)

    # Use 'concatenator' to concatenate a and s_prev on the last axis
    concat = concatenator([a,s_prev])
    # 'concat' is of shape (m, Tx, 2*n_a + n_s)

    # Use 'densor1' to propagate concat through a small fully-connected neural␣
↪network to compute the
    # "intermediate energies" variable e.
    e = densor1(concat)
    # 'e' is of shape (m, Tx, 10)

    # When you pass a tensor through a Dense layer, it multiplies the values in␣
↪the last dimension of the input tensor with
    # the layer's weights and adds the bias, followed by applying the␣
↪activation function, resulting in a new last dimension as
    # specified by the number of units in the Dense layer.

    # Use 'densor2' to propagate e through a small fully-connected neural␣
↪network to compute the "energies" variable energies.
    energies = densor2(e)
    # 'energies' is of shape (m, Tx, 1)

    # Use "activator" on "energies" to compute the attention weights "alphas"
    alphas = activator(energies)
    # 'alphas' is of shape (m,Tx,1)
    # 'alphas' matrix values change when context<t> changes (as timestep t␣
↪changes).
```

```
    # Use dotor together with "alphas" and "a", to compute the context vector␣
↪to be given to the next (post-attention) LSTM-cell
    context = dotor([alphas,a])
    # For every 2*n_a vector in the 'a', there is one value in 'alphas' which␣
↪is along the T_x direction.
    # This value will be multiplied throughout all values of the 2*n_a vector.␣
↪Basically we are scaling the 2*n_a vector.
    # The scaling process is done for each time step and the vectors along the␣
↪T_x direction are added.
    # The shape of 'context' is (m, 2*n_a)

    # Another alternative method that we could have done
    # context = np.sum(alphas*a, axis = 1)

    return context
```

[47]:

```
# UNIT TEST
def one_step_attention_test(target):

    m = 10
    Tx = 30
    n_a = 32
    n_s = 64
    #np.random.seed(10)
    a = np.random.uniform(1, 0, (m, Tx, 2 * n_a)).astype(np.float32)
    s_prev =np.random.uniform(1, 0, (m, n_s)).astype(np.float32) * 1
    context = target(a, s_prev)

    assert type(context) == tf.python.framework.ops.EagerTensor, "Unexpected␣
↪type. It should be a Tensor"
    assert tuple(context.shape) == (m, 1, n_s), "Unexpected output shape"
    assert np.all(context.numpy() > 0), "All output values must be > 0 in this␣
↪example"
    assert np.all(context.numpy() < 1), "All output values must be < 1 in this␣
↪example"

    #assert np.allclose(context[0][0][0:5].numpy(), [0.50877404, 0.57160693, 0.
↪45448175, 0.50074816, 0.53651875]), "Unexpected values in the result"
    print("\033[92mAll tests passed!")

one_step_attention_test(one_step_attention)
```

```
All tests passed!
```

```
[48]: n_a = 32 # number of units for the pre-attention, bi-directional LSTM's hidden␣
      ↪state 'a'
n_s = 64 # number of units for the post-attention LSTM's hidden state "s"

# This is the post attention LSTM cell.
post_activation_LSTM_cell = LSTM(n_s, return_state = True) # Please do not␣
 ↪modify this global variable.
# In a standard LSTM, return_state=True will return the last hidden state␣
 ↪(a<Tx>) and the last cell state (c<Tx>), along with
# the output sequence (if return_sequences=True) ([a<1>,a<2>,....a<Tx>]) or the␣
 ↪last output (a<Tx>) (if return_sequences=False).

# In a standard LSTM, if return_state is False, then it will not return the␣
 ↪last cell state (c<Tx>) and will only return
# the last hidden state a<Tx>.

# In a Bidirectional LSTM, return_sequences=True ensures that you get the␣
 ↪output (hidden states) from both directions for
# each time step of the input sequence.

output_layer = Dense(len(machine_vocab), activation=softmax)
```

```
[49]: def modelf(Tx, Ty, n_a, n_s, human_vocab_size, machine_vocab_size):
    """
    Arguments:
    Tx -- length of the input sequence
    Ty -- length of the output sequence
    n_a -- hidden state size of the Bi-LSTM
    n_s -- hidden state size of the post-attention LSTM
    human_vocab_size -- size of the python dictionary "human_vocab"
    machine_vocab_size -- size of the python dictionary "machine_vocab"

    Returns:
    model -- Keras model instance
    """

    # Define the inputs of your model with a shape (Tx, human_vocab_size)
    X = Input(shape=(Tx, human_vocab_size))

    # Define s0 (initial hidden state) and c0 (initial cell state)
    # for the decoder LSTM with shape (n_s,)
    s0 = Input(shape=(n_s,), name='s0')
    c0 = Input(shape=(n_s,), name='c0')
    # Note the 'comma' is required in 'shape = (n_s,)' because in Keras when we␣
↪define an input shape, we have to pass a tuple
    # representing the dimensions of the input.
```

```python
    # 's0' and 'c0' are just for 1 timestep (the initial one)

    # hidden state
    s = s0
    # cell state
    c = c0

    # Initialize empty list of outputs
    outputs = []

    # Define the pre-attention Bi-LSTM
    a = Bidirectional(LSTM(n_a, return_sequences = True),␣
↪merge_mode='concat')(X)
    # In a Bidirectional LSTM, return_sequences=True ensures that you get the␣
↪output (hidden states) from both directions for
    # each time step of the input sequence.
    # LSTM, Keras automatically initializes the hidden state and the cell state␣
↪to zero vectors by default if we don't
    # specify them. And automatically c<1>,a<1> are used to calculate a<2>.

    # Iterate for Ty steps
    for t in range(Ty):

        # Perform one step of the attention mechanism to get back the context␣
↪vector at step t
        context = one_step_attention(a, s)

        # Apply the post-attention LSTM cell to the "context" vector.
        _, s, c = post_activation_LSTM_cell(context, initial_state = [s,c])

        # Note - The below line of code doesn't work because when we do it␣
↪iteratively in keras we have to explicitly mention
        # the inputs (_, s, c = post_activation_LSTM_cell(initial_state =␣
↪[s,c])(context))

        # Apply Dense layer to the hidden state output of the post-attention␣
↪LSTM
        out = output_layer(s)

        # Append "out" to the "outputs" list
        outputs.append(out)

    # Create model instance taking three inputs and returning the list of␣
↪outputs
    model = Model(inputs=[X,s0,c0], outputs=outputs)
```

```
    return model

# Note 1
# The pre-attention Bidirectional LSTM is called the 'encoder'. Usually the␣
↪'encoder' calculates the output all at once. This is
# because the output of one time step is not fed as input to calculate the␣
↪output of next time step.
# Whereas the 'post_activation_LSTM_cell' is called the 'decoder'. The␣
↪'context' of each time step is fed as input to calculate
# the output of next time step. Hence we calculate the outputs of the time␣
↪steps iteratively.

# Note 2
# We could have defined 's0' and 'c0' to be zero vectors of respective sizes␣
↪(n_s) and
# could have defined 'model = Model(inputs=[X], outputs=outputs)'.
# But if 's0' and 'c0' are hardcoded, it would decrease the flexibility and␣
↪they would always be zero vectors and
# can't be input.
```

[50]:
```
# UNIT TEST
from test_utils import *

def modelf_test(target):
    Tx = 30
    n_a = 32
    n_s = 64
    len_human_vocab = 37
    len_machine_vocab = 11


    model = target(Tx, Ty, n_a, n_s, len_human_vocab, len_machine_vocab)

    print(summary(model))


    expected_summary = [['InputLayer', [(None, 30, 37)], 0],
                        ['InputLayer', [(None, 64)], 0],
                        ['Bidirectional', (None, 30, 64), 17920],
                        ['RepeatVector', (None, 30, 64), 0, 30],
                        ['Concatenate', (None, 30, 128), 0],
                        ['Dense', (None, 30, 10), 1290, 'tanh'],
                        ['Dense', (None, 30, 1), 11, 'relu'],
                        ['Activation', (None, 30, 1), 0],
                        ['Dot', (None, 1, 64), 0],
                        ['InputLayer', [(None, 64)], 0],
```

```
                             ['LSTM',[(None, 64), (None, 64), (None, 64)],␣
↪33024,[(None, 1, 64), (None, 64), (None, 64)],'tanh'],
                             ['Dense', (None, 11), 715, 'softmax']]

    assert len(model.outputs) == 10, f"Wrong output shape. Expected 10 !=␣
↪{len(model.outputs)}"

    comparator(summary(model), expected_summary)


modelf_test(modelf)
```

```
[['InputLayer', [(None, 30, 37)], 0], ['InputLayer', [(None, 64)], 0],
['Bidirectional', (None, 30, 64), 17920], ['RepeatVector', (None, 30, 64), 0,
30], ['Concatenate', (None, 30, 128), 0], ['Dense', (None, 30, 10), 1290,
'tanh'], ['Dense', (None, 30, 1), 11, 'relu'], ['Activation', (None, 30, 1), 0],
['Dot', (None, 1, 64), 0], ['InputLayer', [(None, 64)], 0], ['LSTM', [(None,
64), (None, 64), (None, 64)], 33024, [(None, 1, 64), (None, 64), (None, 64)],
'tanh'], ['Dense', (None, 11), 715, 'softmax']]
All tests passed!
```

[51]:
```
model = modelf(Tx, Ty, n_a, n_s, len(human_vocab), len(machine_vocab))
```

[ ]:
```
model.summary()
```

[53]:
```
# Compiling the Model
```

[54]:
```
opt = Adam(lr=0.005, beta_1=0.9, beta_2=0.999, decay=0.01) # Adam(...)
model.compile(loss = 'categorical_crossentropy', optimizer = opt, metrics =␣
↪['accuracy'])
```

[55]:
```
# UNIT TESTS
assert opt.lr == 0.005, "Set the lr parameter to 0.005"
assert opt.beta_1 == 0.9, "Set the beta_1 parameter to 0.9"
assert opt.beta_2 == 0.999, "Set the beta_2 parameter to 0.999"
assert opt.decay == 0.01, "Set the decay parameter to 0.01"
assert model.loss == "categorical_crossentropy", "Wrong loss. Use␣
↪'categorical_crossentropy'"
assert model.optimizer == opt, "Use the optimizer that you have instantiated"
assert model.compiled_metrics._user_metrics[0] == 'accuracy', "set metrics to␣
↪['accuracy']"

print("\033[92mAll tests passed!")
```

```
All tests passed!
```

[56]:
```
# Model Fitting
```

[57]:
```
s0 = np.zeros((m, n_s))
c0 = np.zeros((m, n_s))

# The 'outputs' to be fed to the model is a list of Ty elements where each␣
↪element is of shape (m,len(machine_vocab)).
# But 'Yoh' is of shape (m,Ty,len(machine_vocab))
# So we swap the axes 0 and 1 to make 'Yoh' of the shape␣
↪(Ty,m,len(machine_vocab)).
# Then we do 'list(swapped_Yoh)' so that 'Yoh' becomes a list consisting of Ty␣
↪elements where
# each element of size (m,Ty,len(machine_vocab))

outputs = list(Yoh.swapaxes(0,1))
```

[58]:
```
model.fit([Xoh, s0, c0], outputs, epochs=1, batch_size=100)
```

```
100/100 [==============================] - 9s 86ms/step - loss: 16.5680 -
dense_5_loss: 1.1589 - dense_5_1_loss: 1.0174 - dense_5_2_loss: 1.8156 -
dense_5_3_loss: 2.6593 - dense_5_4_loss: 0.7361 - dense_5_5_loss: 1.2673 -
dense_5_6_loss: 2.6730 - dense_5_7_loss: 0.9684 - dense_5_8_loss: 1.7123 -
dense_5_9_loss: 2.5598 - dense_5_accuracy: 0.5120 - dense_5_1_accuracy: 0.6828 -
dense_5_2_accuracy: 0.2907 - dense_5_3_accuracy: 0.0836 - dense_5_4_accuracy:
0.9513 - dense_5_5_accuracy: 0.3205 - dense_5_6_accuracy: 0.0500 -
dense_5_7_accuracy: 0.9391 - dense_5_8_accuracy: 0.2542 - dense_5_9_accuracy:
0.1061
```

[58]:
```
<tensorflow.python.keras.callbacks.History at 0x7fa59e17cb50>
```

[59]:
```
# The below model has been run for longer time and the weights have been saved

model.load_weights('models/model.h5')
```

[60]:
```
# Results with the saved model weights

EXAMPLES = ['3 May 1979', '5 April 09', '21th of August 2016', 'Tue 10 Jul␣
↪2007', 'Saturday May 9 2018', 'March 3 2001', 'March 3rd 2001', '1 March␣
↪2001']
s00 = np.zeros((1, n_s))
c00 = np.zeros((1, n_s))
for example in EXAMPLES:
    source = string_to_int(example, Tx, human_vocab)
    #print(source)
    source = np.array(list(map(lambda x: to_categorical(x,␣
↪num_classes=len(human_vocab)), source))).swapaxes(0,1)
    source = np.swapaxes(source, 0, 1)
    source = np.expand_dims(source, axis=0)
    prediction = model.predict([source, s00, c00])
```

```python
    prediction = np.argmax(prediction, axis = -1)
    output = [inv_machine_vocab[int(i)] for i in prediction]
    print("source:", example)
    print("output:", ''.join(output),"\n")
```

```
source: 3 May 1979
output: 1979-05-33

source: 5 April 09
output: 2009-04-05

source: 21th of August 2016
output: 2016-08-20

source: Tue 10 Jul 2007
output: 2007-07-10

source: Saturday May 9 2018
output: 2018-05-09

source: March 3 2001
output: 2001-03-03

source: March 3rd 2001
output: 2001-03-03

source: 1 March 2001
output: 2001-03-01
```

[61]:
```python
# Note
# Machine translation models can be used to map from one sequence to another.
# They are useful not just for translating human languages (like␣
 ↪French->English) but also for tasks like
# date format translation.

# A network using an attention mechanism can translate from inputs of length Tx␣
 ↪to outputs of length Ty, where
# Tx and Ty can be different.

# The model that we built here can be used to translate from one language to␣
 ↪another, such as translating from English to Hindi.
# However, language translation requires massive datasets and usually takes␣
 ↪days of training on GPUs.
```