Deep Learning School

Физтех-Школа Прикладной математики и информатики (ФПМИ) МФТИ

Some parts of the notebook are almost the exact copy of https://github.com/yandexdataschool/nlp_course

We are going to implement the model from the (2014) NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE paper.

Background

Attention

Attention layer can take in the previous hidden state of the decoder s_{t-1}, and all of the stacked forward and backward hidden states H from the encoder. The layer will output an attention vector a_t, that is the length of the source sentence, each element is between 0 and 1 and the entire vector sums to 1.

Intuitively, this layer takes what we have decoded so far s_{t-1}, and all of what we have encoded H, to produce a vector a_t, that represents which words in the source sentence we should pay the most attention to in order to correctly predict the next word to decode $\hat{y}_{t+1}$. The decoder input word that has been embedded y_t.

You can use any type of the attention scores between previous hidden state of the encoder s_{t-1} and hidden state of the decoder $h \in H$, you prefer. We have met at least three of them:

$$\operatorname{score}(\boldsymbol{h}, \boldsymbol{s}_{t-1}) = \begin{cases} \boldsymbol{h}^\top \boldsymbol{s}_{t-1} & \text{dot} \\ \boldsymbol{h}^\top \boldsymbol{W_a} \boldsymbol{s}_{t-1} & \text{general} \\ \boldsymbol{v}_a^\top \tanh(\boldsymbol{W_a}[\boldsymbol{h}; \boldsymbol{s}_{t-1}]) & \text{concat} \end{cases}$$

We wil use "concat attention":

First, we calculate the *energy* between the previous decoder hidden state s_{t-1} and the encoder hidden states H. As our encoder hidden states H are a sequence of T tensors, and our previous decoder hidden state s_{t-1} is a single tensor, the first thing we do is `repeat` the previous decoder hidden state T times. $\Rightarrow$
We have:

$$H = \left[\boldsymbol{h}_0, \ldots, \boldsymbol{h}_{T-1}\right]$$
$$\left[\boldsymbol{s}_{t-1}, \ldots, \boldsymbol{s}_{t-1}\right]$$

The encoder hidden dim and the decoder hidden dim should be equal: **dec hid dim = enc hid dim**.
We then calculate the energy, E_t, between them by concatenating them together:

$$\left[[\boldsymbol{h}_0, \boldsymbol{s}_{t-1}], \ldots, [\boldsymbol{h}_{T-1}, \boldsymbol{s}_{t-1}]\right]$$

And passing them through a linear layer (`attn` = $\boldsymbol{W_a}$) and a $\tanh$ activation function:

$$E_t = \tanh(\operatorname{attn}(H, s_{t-1}))$$

This can be thought of as calculating how well each encoder hidden state "matches" the previous decoder hidden state.

We currently have a **[enc hid dim, src sent len]** tensor for each example in the batch. We want this to be **[src sent len]** for each example in the batch as the attention should be over the length of the source sentence. This is achieved by multiplying the `energy` by a **[1, enc hid dim]** tensor, v.

$$\hat{a}_t = vE_t$$

We can think of this as calculating a weighted sum of the "match" over all `enc_hid_dem` elements for each encoder hidden state, where the weights are learned (as we learn the parameters of v).

Finally, we ensure the attention vector fits the constraints of having all elements between 0 and 1 and the vector summing to 1 by passing it through a $\mathrm{softmax}$ layer.

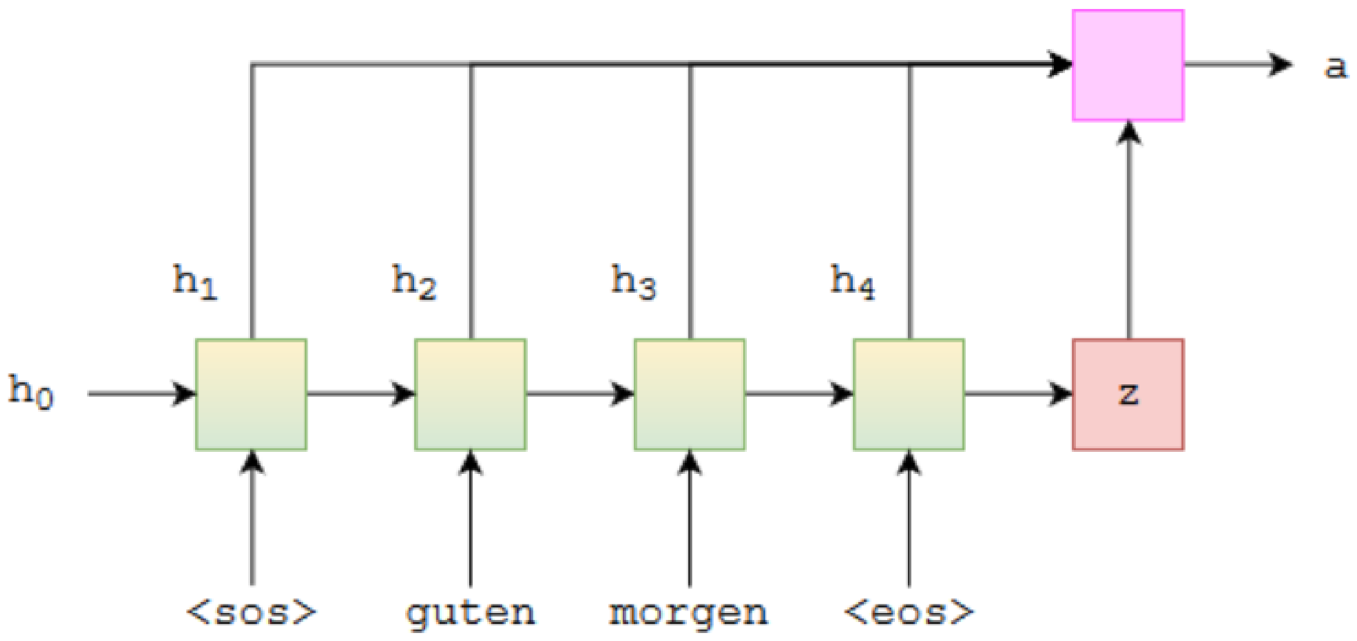
$$a_t = \mathrm{softmax}(\hat{a_t})$$

Temperature SoftMax

$$\mathrm{softmax}(x)_i = \frac{e^{\frac{y_i}{T}}}{\Sigma_j^N e^{\frac{y_j}{T}}}$$

This gives us the attention over the source sentence!

Graphically, this looks something like below. $z = s_{t-1}$. The green/yellow blocks represent the hidden states from both the forward and backward RNNs, and the attention computation is all done within the pink block.



Mount Drive

```
%ls
```

```
    sample_data/
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
%ls drive/MyDrive/colab_projects/nlp_from_dls/11_HW_Seq2Seq_with_Attention/
```

```
'Copy of [homework]NeuralMachineTranslation.ipynb'   modules.py
 data/                                               __pycache__/
'learning curves, epochs 8 - 27.png'                 weights/
 modules_original.py
```

```
%cd ./drive/MyDrive/colab_projects/nlp_from_dls/11_HW_Seq2Seq_with_Attention/
```

```
/content/drive/MyDrive/colab_projects/nlp_from_dls/11_HW_Seq2Seq_with_Attention
```

Neural Machine Translation

Write down some summary on your experiments and illustrate it with convergence plots/metrics and your thoughts. Just like you would approach a real problem.

```
import os
os.listdir()
```

```
['modules_original.py',
 'weights',
 'data',
 'modules.py',
 '__pycache__',
 'learning curves, epochs 8 - 27.png',
 'Copy of [homework]NeuralMachineTranslation.ipynb']
```

```
if 'data' not in os.listdir():
    os.mkdir('./data')
    print('Created an ampty folder `data` in current directory')
else:
    print('Current directory already has `./data/` subdirectory.')
```

```
Current directory already has `./data/` subdirectory.
```

```
if 'data.txt' not in os.listdir('./data'):
    ! wget https://drive.google.com/uc?id=1NWYqJgeG_4883LINdEjKUr6nLQPY6Yb_ -O ./da
    print('Downloaded the data into `./data/data.txt`.')
else:
    print('Data has already been downloaded.')

# Thanks to YSDA NLP course team for the data
# (who thanks tilda and deephack teams for the data in their turn)
```

```
Data has already been downloaded.
```

```
import torch
import torch.nn as nn
import torch.optim as optim

import torchtext
from torchtext.legacy.data import Field, BucketIterator

import spacy

import random
import math
import time
import tqdm
import numpy as np

import matplotlib
matplotlib.rcParams.update({'figure.figsize': (16, 12), 'font.size': 14})
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import clear_output

from nltk.tokenize import WordPunctTokenizer
```

We'll set the random seeds for deterministic results.

```
SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Preparing Data

Here comes the preprocessing

```
tokenizer_W = WordPunctTokenizer()

def tokenize_ru(x, tokenizer=tokenizer_W):
    return tokenizer.tokenize(x.lower())[::-1] # reversed the order for source

def tokenize_en(x, tokenizer=tokenizer_W):
    return tokenizer.tokenize(x.lower())
```

Note, we **reverse the source sentences**. It improves the performance when using only forward RNNs. When using bi-directional RNNs, reversing will not make any difference since we will feed the sentence from both sides anyway.

```
SRC = Field(tokenize=tokenize_ru,
            init_token = '<sos>',
            eos_token = '<eos>',
            lower = True)

TRG = Field(tokenize=tokenize_en,
            init_token = '<sos>',
            eos_token = '<eos>',
            lower = True)
```

```
dataset = torchtext.legacy.data.TabularDataset(
    path='./data/data.txt',
    format='tsv',
    fields=[('trg', TRG), ('src', SRC)]
)
```

```
print(len(dataset.examples))
print(dataset.examples[0].src[::-1]) # reversed back for readability
print(dataset.examples[0].trg)
```

```
50000
['отель', 'cordelia', 'расположен', 'в', 'тбилиси', ',', 'в', '3', 'минутах', '>
['cordelia', 'hotel', 'is', 'situated', 'in', 'tbilisi', ',', 'a', '3', '-', 'mi
```

There is something strange with train/val/test order (in the documentation too).

```
train_data, valid_data, test_data = dataset.split(split_ratio=[0.8, 0.15, 0.05])

print(f"Number of training examples: {len(train_data.examples)}")
print(f"Number of validation examples: {len(valid_data.examples)}")
print(f"Number of testing examples: {len(test_data.examples)}")

assert len(train_data.examples) >= len(test_data.examples) >= len(valid_data.exampl
    "Check the print statements, are those really the numbers you wanted to get?"
```

```
Number of training examples: 40000
Number of validation examples: 2500
Number of testing examples: 7500
```

Choosing `min_freq` >1 results in less words, which in turn results in

- less noise from misspelt words
- lower memmry and compute for the Embedding layers

Do not choose `min_freq` too high, otherwise, there will be a lot of unknown words in validation and test sets.

```
SRC.build_vocab(train_data, min_freq = 3)
TRG.build_vocab(train_data, min_freq = 3)


print(f"Unique tokens in source (ru) vocabulary: {len(SRC.vocab)}")
print(f"Unique tokens in target (en) vocabulary: {len(TRG.vocab)}")
```

```
   Unique tokens in source (ru) vocabulary: 9256
   Unique tokens in target (en) vocabulary: 6734
```

And here is example from train dataset:

```
print(vars(train_data.examples[9]))
```

```
   {'trg': ['other', 'facilities', 'offered', 'at', 'the', 'property', 'include', '
```

```
for key, item in vars(train_data.examples[9]).items():
    print(key, ':', item, '\n', '\t', item[::-1]) # also print the reversed order
```

```
   trg : ['other', 'facilities', 'offered', 'at', 'the', 'property', 'include', 'gr
            ['.', 'services', 'ironing', 'and', 'laundry', ',', 'deliveries', 'groc
   src : ['.', 'услуги', 'гладильные', 'и', 'прачечной', 'услуги', ',', 'продуктов'
            ['также', 'предлагается', 'доставка', 'продуктов', ',', 'услуги', 'прач
```

When we get a batch of examples using an iterator we need to make sure that all of the source sentences are padded to the same length, the same with the target sentences. Luckily, TorchText iterators handle this for us!

We use a `BucketIterator` instead of the standard `Iterator` as it creates batches in such a way that it minimizes the amount of padding in both the source and target sentences.

```
!nvidia-smi
```

```
   Thu Apr 15 14:46:40 2021
   +-----------------------------------------------------------------------------+
   | NVIDIA-SMI 460.67       Driver Version: 460.32.03    CUDA Version: 11.2     |
   |-------------------------------+----------------------+----------------------+
   | GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
   | Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
   |                               |                      |               MIG M. |
   |===============================+======================+======================|
   |   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
   | N/A   31C    P8    29W / 149W |      0MiB / 11441MiB |      0%      Default |
   |                               |                      |                  N/A |
   +-------------------------------+----------------------+----------------------+

   +-----------------------------------------------------------------------------+
   | Processes:                                                                  |
   |  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
   |        ID   ID                                                   Usage      |
   |=============================================================================|
   |  No running processes found                                                 |
   +-----------------------------------------------------------------------------+
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cuda')
```

```
def _len_sort_key(x):
    return len(x.src)

# ORIGINAL: 80
BATCH_SIZE = 128

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device,
    sort_key=_len_sort_key
)
```

Encoder

For a multi-layer RNN, the input sentence, X, goes into the first (bottom) layer of the RNN and hidden states, $H = \{h_1, h_2, \ldots, h_T\}$, output by this layer are used as inputs to the RNN in the layer above. Thus, representing each layer with a superscript, the hidden states in the first layer are given by:

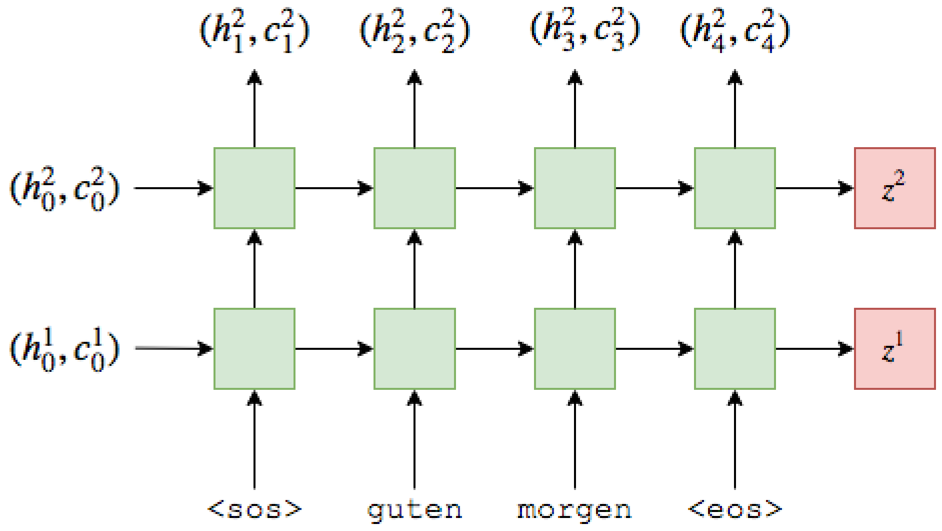
$$h_t^1 = \text{EncoderRNN}^1(x_t, h_{t-1}^1)$$

The hidden states in the second layer are given by:

$$h_t^2 = \text{EncoderRNN}^2(h_t^1, h_{t-1}^2)$$

Extending our multi-layer equations to LSTMs, we get:

$$(h_t^1, c_t^1) = \text{EncoderLSTM}^1(x_t, (h_{t-1}^1, c_{t-1}^1))$$
$$(h_t^2, c_t^2) = \text{EncoderLSTM}^2(h_t^1, (h_{t-1}^2, c_{t-1}^2))$$



Note, in this paper, "gated hidden units" are used both in the Encoder and in the Decoder (see torch.nn.GRU). However, LSTM cells can be used instead as was done in a similar context by Sutskever *et al.* (2014).

```
class Encoder
```

[ ] ↳ *1 cell hidden*

Attention

Attention layer can take in the previous hidden state of the decoder s_{t-1}, and all of the stacked forward and backward hidden states H from the encoder. The layer will output an attention vector a_t, that is the length of the source sentence, each element is between 0 and 1 and the entire vector sums to 1.

Intuitively, this layer takes what we have decoded so far s_{t-1}, and all of what we have encoded H, to produce a vector a_t, that represents which words in the source sentence we should pay the most attention to in order to correctly predict the next word to decode $\hat{y}_{t+1}$. The decoder input word that has been embedded y_t.

You can use any type of the attention scores between previous hidden state of the encoder s_{t-1} and hidden state of the decoder $h \in H$, you prefer. We have met at least three of them:

$$\Big(\boldsymbol{h}^\top \boldsymbol{s}_{t-1} \qquad\qquad\qquad\qquad \text{dot}$$

We will use "concat attention":

First, we calculate the *energy* between the previous decoder hidden state s_{t-1} (`hidden`) and the encoder hidden states (of the last encoder layer if more than 1) H (`encoder_outputs`). As our encoder hidden states H are a sequence of T (`src sent len`) tensors, and our previous decoder hidden state s_{t-1} (`hidden`) is a single tensor, the first thing we do is `repeat` the previous decoder hidden state T times. $\Rightarrow$
We have:

$$H = \left[\boldsymbol{h}_0, \ldots, \boldsymbol{h}_{T-1}\right]$$
$$\left[\boldsymbol{s}_{t-1}, \ldots, \boldsymbol{s}_{t-1}\right]$$

The encoder hidden dim and the decoder hidden dim should be equal: **dec hid dim = enc hid dim**.
We then calculate the energy, E_t, between them by concatenating them together:

$$\left[[\boldsymbol{h}_0, \boldsymbol{s}_{t-1}], \ldots, [\boldsymbol{h}_{T-1}, \boldsymbol{s}_{t-1}]\right]$$

And passing them through a linear layer (`attn` = $\boldsymbol{W_a}$) and a $\tanh$ activation function:

$$E_t = \tanh(\text{attn}(H, s_{t-1}))$$

This can be thought of as calculating how well each encoder hidden state "matches" the previous decoder hidden state.

We currently have a **[enc hid dim, src sent len]** tensor for each example in the batch. We want this to be **[src sent len]** for each example in the batch as the attention should be over the length of the source sentence. This is achieved by multiplying the `energy` by a **[1, enc hid dim]** tensor, v.

$$\hat{a}_t = vE_t$$

We can think of this as calculating a weighted sum of the "match" over all `enc_hid_dim` elements for each encoder hidden state, where the weights are learned (as we learn the parameters of v).

Finally, we ensure the attention vector fits the constraints of having all elements between 0 and 1 and the vector summing to 1 by passing it through a softmax layer.

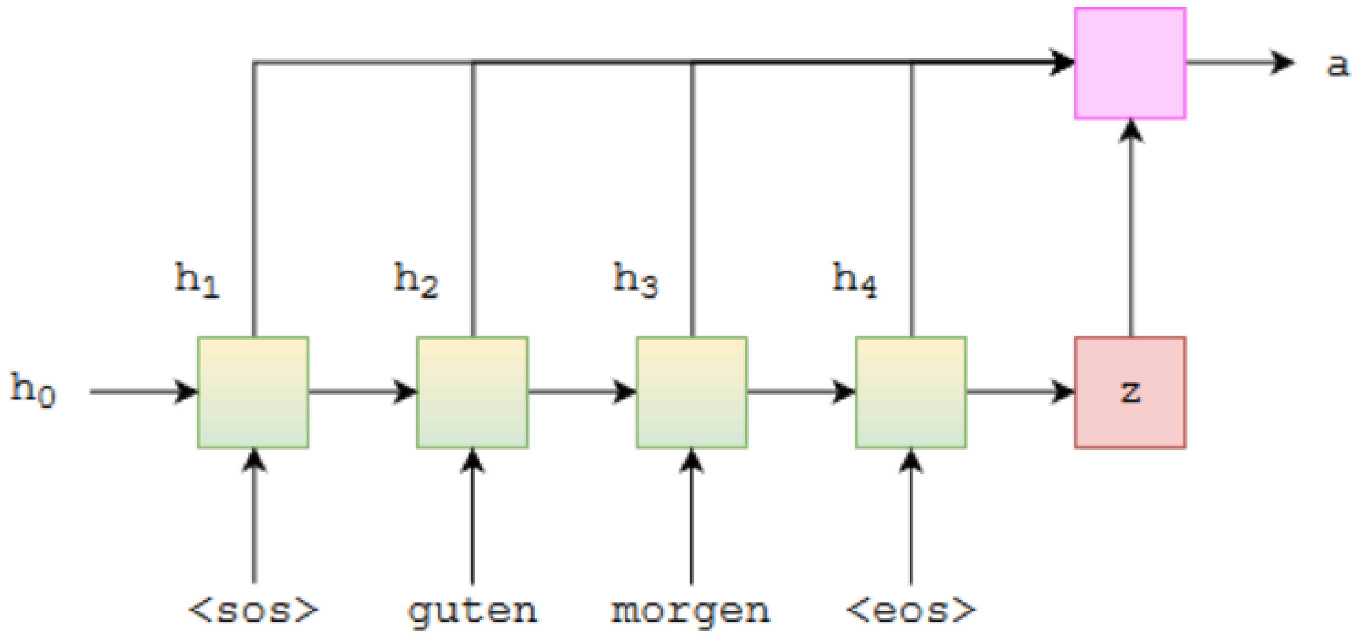
$$a_t = \text{softmax}(\hat{a_t})$$

Temperature SoftMax

$$\text{softmax}(x)_i = \frac{e^{\frac{y_i}{T}}}{\Sigma_j^N e^{\frac{y_j}{T}}}$$

This gives us the attention over the source sentence!

Graphically, this looks something like below. $z = s_{t-1}$ (`hidden`). The green/yellow blocks represent the hidden states from both the forward and backward RNNs, and the attention computation is all done within the pink block.



Attention Summary

$$\text{score}(\boldsymbol{h}, \boldsymbol{s}_{t-1}) = \boldsymbol{v}_a^\top \tanh(\boldsymbol{W_a} [\boldsymbol{h}; \boldsymbol{s}_{t-1}]) \text{ - concat attention}$$

```
# use your temperature
def softmax(x, temperature):
    e_x = torch.exp(x / temperature)
    return e_x / torch.sum(e_x, dim=0)
```

`class Attention`

[ ] ↳ *1 cell hidden*

Checking that how it all works on example tensors.

[ ] ↳ *3 cells hidden*

Decoder with Attention

To make it really work you should also change the `Decoder` class from the classwork in order to make it to use `Attention`. You may just copy-paste `Decoder` class and add several lines of code to it.

The decoder contains the attention layer `attention`, which takes the previous hidden state s_{t-1} (`hidden`), all of the encoder hidden states H (`encoder_outputs`), and returns the attention vector a_t.

We then use this attention vector to create a weighted source vector, w_t (`weighted`), which is a weighted sum of the encoder hidden states, H, using a_t as the weights.

$$w_t = a_t H$$

Without batches :

- $a_t \in \mathbb{R}^{(1, src-sent-len)}$
- $H \in \mathbb{R}^{(src-sent-len, enc-hid-dim)}$
- $w_t \in \mathbb{R}^{(1, enc-hid-dim)}$

Need to do something like that for every sentence in the batch.

The input word (`input`) that has been embedded (`embedded`) y_t, the weighted source vector w_t (`weighted`), and the previous decoder hidden state s_{t-1} (`hidden`), are then all passed into the decoder RNN, with y_t (`embedded`) and w_t (`weighted`) being concatenated together.

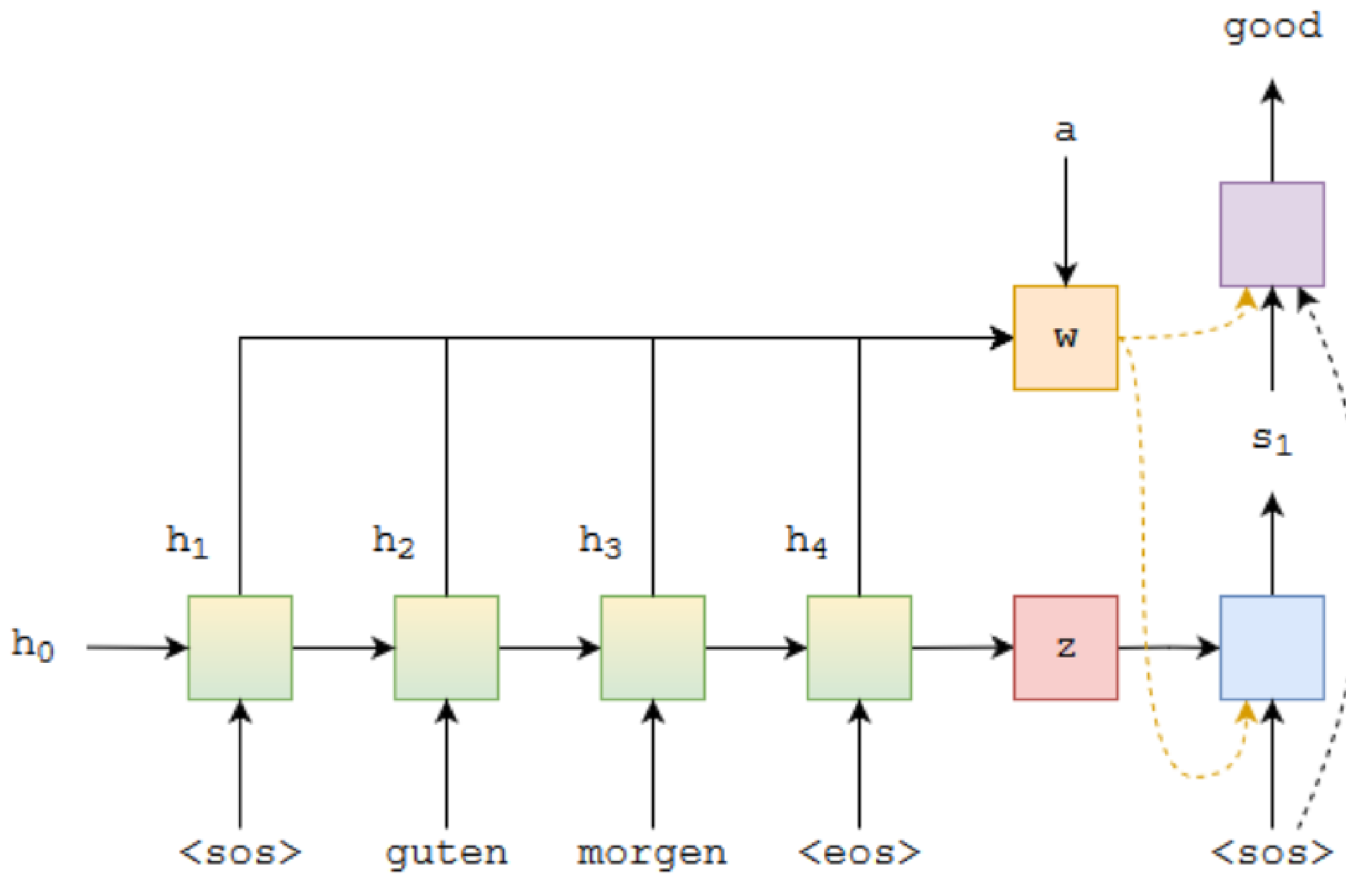
$$s_t = \text{DecoderGRU}([y_t, w_t], s_{t-1})$$

We then pass y_t (`embedded`), w_t (`weighted`) and s_t (**updated** `hidden`) through the linear layer, f, to make a prediction of the next word in the target sentence, $\hat{y}_{t+1}$. This is done by concatenating them all together.

$$\hat{y}_{t+1} = f(y_t, w_t, s_t)$$

The image below shows decoding the **first** word in an example translation.

The green/yellow blocks show the forward/backward encoder RNNs which output H, the red block is $z = s_{t-1} = s_0$, the blue block shows the decoder RNN which outputs $s_t = s_1$, the purple block shows the linear layer, f, which outputs $\hat{y}_{t+1}$ and the orange block shows the calculation of the weighted sum over H by a_t and outputs w_t. Not shown is the calculation of a_t.



`class DecoderWithAttention`

[ ] ↳ 1 cell hidden

Checking that how it all works on example tensors.

[ ] ↳ 5 cells hidden

Seq2Seq

Main idea:

- $w_t = a_t H$
- $s_t = \text{DecoderGRU}([y_t, w_t], s_{t-1})$
- $\hat{y}_{t+1} = f(y_t, w_t, s_t)$

Note: our decoder loop starts at 1, not 0. This means the 0th element of our `outputs` tensor remains all zeros. So our `trg` and `outputs` look something like:

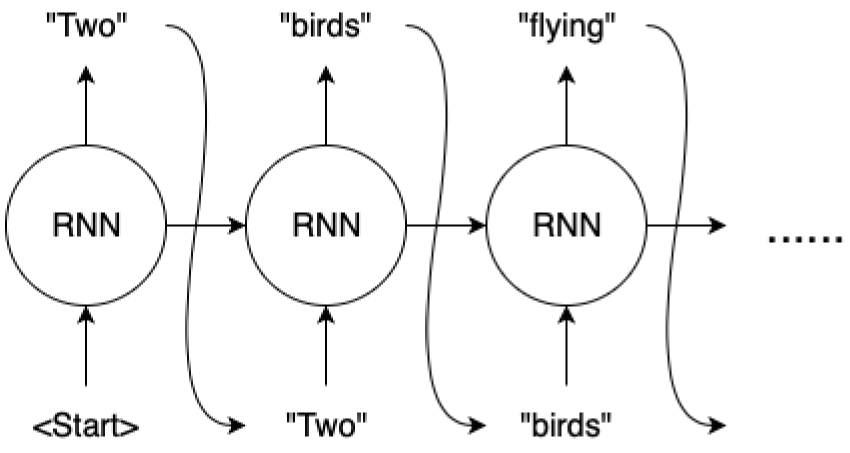
$$\text{trg} = [<sos>, y_1, y_2, y_3, <eos>]$$
$$\text{outputs} = [0, \hat{y}_1, \hat{y}_2, \hat{y}_3, <eos>]$$

Later on when we calculate the loss, we cut off the first element of each tensor to get:

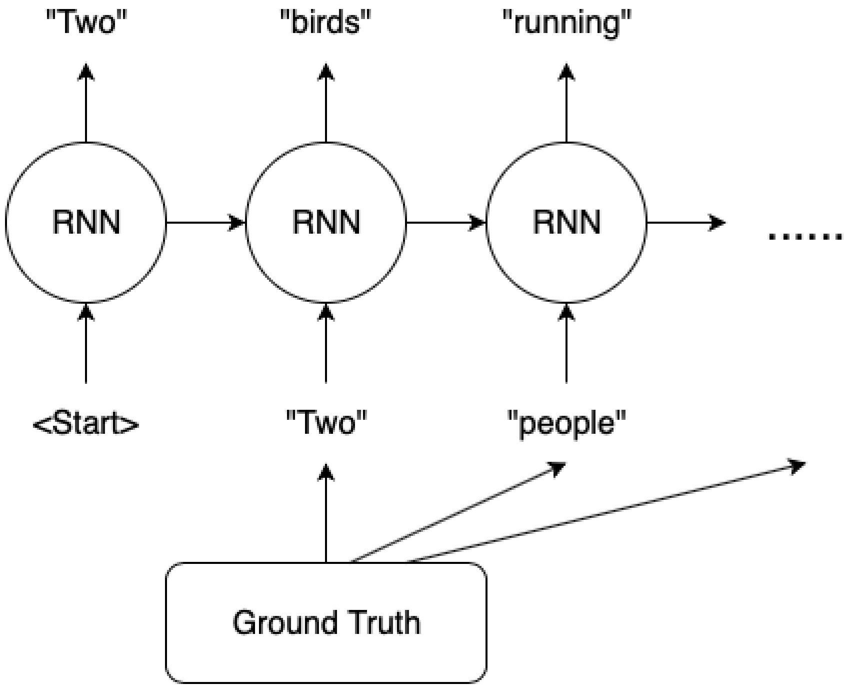
$$\text{trg} = [y_1, y_2, y_3, <eos>]$$
$$\text{outputs} = [\hat{y}_1, \hat{y}_2, \hat{y}_3, <eos>]$$

Teacher forcing

Teacher forcing is a method for quickly and efficiently training recurrent neural network models that use the ground truth from a prior time step as input.



Without Teacher Forcing



With Teacher Forcing

When training/testing our model, we always know how many words are in our target sentence, so we stop generating words once we hit that many. During inference (i.e. real world usage) it is common to keep generating words until the model outputs an `<eos>` token or after a certain amount of words have been generated.

Once we have our predicted target sentence, $\hat{Y} = \{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_T\}$, we compare it against our actual target sentence, $Y = \{y_1, y_2, \ldots, y_T\}$, to calculate our loss. We then use this loss to update all of the parameters in our model.

class Seq2Seq

[ ] ↳ 1 cell hidden

Training

```
# For reloading
import modules
import imp
imp.reload(modules)

Encoder = modules.Encoder
Attention = modules.Attention
Decoder = modules.DecoderWithAttention
Seq2Seq = modules.Seq2Seq
```

Initialize model

```
                                                     # ORIGINAL
INPUT_DIM = len(SRC.vocab)                           # 30,000
OUTPUT_DIM = len(TRG.vocab)                          # 30,000
ENC_EMB_DIM = 256                                    # 620
DEC_EMB_DIM = 256                                    # 620
DEC_HID_DIM = 512                                    # 1000
N_LAYERS = 2
ENC_DROPOUT = 0.5
DEC_DROPOUT = 0.5
BIDIRECTIONAL = True                                 # True
ENC_HID_DIM = DEC_HID_DIM // (1 + BIDIRECTIONAL)    # 1000 fwd + 1000 bwd = 2000

enc = Encoder(INPUT_DIM, ENC_EMB_DIM, ENC_HID_DIM, N_LAYERS, ENC_DROPOUT, BIDIRECTI

# we do not give encoder's bidirectional argument to attention and decoder, but nee
att = Attention(enc_hid_dim=ENC_HID_DIM*(1+BIDIRECTIONAL), dec_hid_dim=DEC_HID_DIM,
dec = DecoderWithAttention(OUTPUT_DIM, DEC_EMB_DIM, ENC_HID_DIM*(1+BIDIRECTIONAL),

# dont forget to put the model to the right device
model = Seq2Seq(enc, dec, device).to(device)
```

Train From Pretrained?

```
FROM_PRETRAINED = True
```

IMPORTANT: From the original paper (2014): We trained each model for approximately **5 days**. That is **why we use smaller model**.

```
def init_weights(m):
    for name, param in m.named_parameters():
        nn.init.uniform_(param, -0.08, 0.08)

#model.apply(init_weights)

if FROM_PRETRAINED:
    try:
        # Make sure the following line is in the detector.
        # prediction = self.out(torch.cat([embedded, weighted, output], dim=2).sque
        model.load_state_dict(torch.load('./weights/best-val-model.pt'))
        print('Loaded pre-trained model.')
    except FileNotFoundError as e:
        print(e)
        model.apply(init_weights)
        print('\nInitialized the weights randomly: Uniform distrubution (-0.08, 0.0

else:
    model.apply(init_weights)
    print('\nInitialized the weights randomly: Uniform distrubution (-0.08, 0.08).'
```

```
   Loaded pre-trained model.
```

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
   The model has 19,419,727 trainable parameters
```

Define training, validation, and BLEU-evaluation functions

Helper functions

```
def cut_on_eos(tokens_iter):
    for token in tokens_iter:
        if token == '<eos>':
            break
        yield token

def remove_tech_tokens(tokens_iter, tokens_to_remove=['<sos>', '<unk>', '<pad>']):
    return [x for x in tokens_iter if x not in tokens_to_remove]
```

```
def generate_translation(src, trg, model, TRG_vocab, SRC_vocab):
    model.eval()

    output = model(src, trg, 0) #turn off teacher forcing
    output = output[1:].argmax(-1)

    source = remove_tech_tokens(cut_on_eos([SRC_vocab.itos[x] for x in list(src[:,0
    original = remove_tech_tokens(cut_on_eos([TRG_vocab.itos[x] for x in list(trg[:
    generated = remove_tech_tokens(cut_on_eos([TRG_vocab.itos[x] for x in list(outp

    print('Source: {}'.format(' '.join(source[::-1])))
    print('Original: {}'.format(' '.join(original)))
    print('Generated: {}'.format(' '.join(generated)))
    print()

def get_text(x, vocab):
     generated = remove_tech_tokens(cut_on_eos([vocab.itos[elem] for elem in list(x
     return generated
```

Actual functions

```
from nltk.translate.bleu_score import corpus_bleu

def train(model, iterator, optimizer, criterion, clip, train_history=None, valid_hi
    model.train()

    epoch_loss = 0
    history = []
    for i, batch in enumerate(tqdm.notebook.tqdm(iterator)):

        src = batch.src
        trg = batch.trg

        optimizer.zero_grad()

        output = model(src, trg)

        #trg = [trg sent len, batch size]
        #output = [trg sent len, batch size, output dim]

        output = output[1:].view(-1, OUTPUT_DIM)
        trg = trg[1:].view(-1)

        #trg = [(trg sent len - 1) * batch size]
        #output = [(trg sent len - 1) * batch size, output dim]

        loss = criterion(output, trg)

        loss.backward()
```

```
            # Let's clip the gradient
            torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

            optimizer.step()

            epoch_loss += loss.item()

            history.append(loss.cpu().data.numpy())

        return history, (epoch_loss / len(iterator))

    def evaluate(model, iterator, criterion):

        model.eval()

        epoch_loss = 0

        history = []

        with torch.no_grad():
            for i, batch in enumerate(tqdm.notebook.tqdm(iterator)):

                src = batch.src
                trg = batch.trg

                output = model(src, trg, 0) #turn off teacher forcing

                #trg = [trg sent len, batch size]
                #output = [trg sent len, batch size, output dim]

                output = output[1:].view(-1, OUTPUT_DIM)
                trg = trg[1:].view(-1)

                #trg = [(trg sent len - 1) * batch size]
                #output = [(trg sent len - 1) * batch size, output dim]

                loss = criterion(output, trg)

                epoch_loss += loss.item()

        return epoch_loss / len(iterator)


    from nltk.translate.bleu_score import corpus_bleu
    def evaluate_bleu(model, iterator, TRG_vocab):
        original_text = []
        generated_text = []

        model.eval()
        with torch.no_grad():
            for i, batch in enumerate(tqdm.notebook.tqdm(iterator)):
```

```
            src = batch.src
            trg = batch.trg

            output = model(src, trg, 0) #turn off teacher forcing

            #trg = [trg sent len, batch size]
            #output = [trg sent len, batch size, output dim]

            output = output[1:].argmax(-1)

            original_text.extend([get_text(x, TRG_vocab) for x in trg.cpu().numpy()
            generated_text.extend([get_text(x, TRG_vocab) for x in output.detach().
            # original_text = flatten(original_text)
            # generated_text = flatten(generated_text)
        return corpus_bleu([[text] for text in original_text], generated_text) * 10
```

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Define optimizer, scheduller, criterion, etc.

```
PAD_IDX = TRG.vocab.stoi['<pad>']
optimizer = optim.Adam(model.parameters()) # ORIGINAL: SGD together with Adadelta
sheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min')
criterion = nn.CrossEntropyLoss(ignore_index = PAD_IDX)
```

```
import matplotlib
matplotlib.rcParams.update({'figure.figsize': (16, 12), 'font.size': 14})
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import clear_output
```

Train the model?

```
TRAIN = False
```

```
if TRAIN:
    train_history = []
    valid_history = []
```

```
        valid_bleu_history = []

        N_EPOCHS = 30
        PATIENCE = 5
        CLIP = 5

        # better to optimise for BLEU on validation set if that's what we care about on
        #   the test set
        # best_valid_loss = float('inf')

        # allowes to keep the best model saved and only substituted if better valid ble
        print('Calculating valid bleu of the best checkpoint.')
        best_valid_bleu = evaluate_bleu(model, valid_iterator, TRG.vocab)
        print(f'Best valid bleu achieved: {best_valid_bleu:.3}')
        print('\n', '-'*80, '\n')

        best_epoch = 0
        current_patience = 0

        for epoch in range(N_EPOCHS):
            print(f'Epoch: {epoch+1:02}')

            start_time = time.time()

            print('Calculating train_loss')
            epochs_history, train_loss = train(model, train_iterator, optimizer, criter
            print('Calculating valid_loss')
            valid_loss = evaluate(model, valid_iterator, criterion)
            print('Calculating valid_bleu')
            valid_bleu = evaluate_bleu(model, valid_iterator, TRG.vocab)

            # change learning rate
            sheduler.step(valid_loss)

            end_time = time.time()

            epoch_mins, epoch_secs = epoch_time(start_time, end_time)

            # REMEMBER TO CHECK THE SIGN
            if valid_bleu > best_valid_bleu:
                # record
                best_valid_bleu = valid_bleu
                current_patience = 0
                best_epoch = epoch
                # save
                torch.save(model.state_dict(), './weights/best-val-model.pt')
            else:
                current_patience += 1

            train_history.append(train_loss)
            valid_history.append(valid_loss)
            valid_bleu_history.append(valid_bleu)
            # plot once every epoch
            fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(13.5, 6))
            ax[0].plot(epochs_history, label='train loss')
            ax[0].set_xlabel('Batch')
            ax[0].set_title('Train loss')
            if train_history is not None:
                ax[1].plot(train_history, label='train loss')
                ax[1].set_xlabel('Epoch')
            if valid_history is not None:
                ax[1].plot(valid_history, label='valid loss')
            if valid_bleu_history is not None:
                ax[2].plot(valid_bleu_history, label='valid BLEU')
                ax[2].set_xlabel('Epoch')
            plt.legend()
            plt.show()
            # print once every epoch
            print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
            print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7
            print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7
            print(f'\t Val. BLEU: {valid_bleu:.3f}')

            # break if reached the patience
            if current_patience > PATIENCE:
                print(f"No improvement for {PATIENCE} epochs.")
                break

            print('\n', '-'*80, '\n')

        print(f'Best valid bleu = {best_valid_bleu:.3f} achieved at epoch {best_epoch+1
    else:
        print('Using a pre-trained model w/o further training.')
```

```
Using a pre-trained model w/o further training.
```

IMPORTANT: in the original implementation **beam-search** was used to improve results.

Model evaluation, change hyperparameters until satisfied

Load best model and generate 10 translations

```
model.load_state_dict(torch.load('./weights/best-val-model.pt'))

# # remember to change the predictions calculation in the decoder
# model.load_state_dict(torch.load('./weights/model-bleu-29-one-layer.pt'))
```

```
batch = next(iter(test_iterator))

for idx in range(10):
    src = batch.src[:, idx:idx+1]
    trg = batch.trg[:, idx:idx+1]
    generate_translation(src, trg, model, TRG.vocab, SRC.vocab)
```

```
Source: стойка регистрации открыта круглосуточно .
Original: there is a 24 - hour front desk at the property .
Generated: there is a 24 - hour front desk and a 24 - hour front desk .

Source: стойка регистрации работает круглосуточно .
Original: you will find a 24 - hour front desk at the property .
Generated: there is a 24 - hour front desk at the property .

Source: стойка регистрации работает круглосуточно .
Original: there is a 24 - hour front desk at the property .
Generated: there is a 24 - hour front desk at the property .

Source: имеется бесплатная частная парковка .
Original: free private parking is available .
Generated: free private parking is available on site .

Source: поблизости работает несколько ресторанов .
Original: there are several restaurants in the surrounding area .
Generated: there are several restaurants nearby restaurants nearby .

Source: гостям предоставляется бесплатная парковка .
Original: the property also offers free parking .
Generated: free parking is available .

Source: в доме имеется кухня .
Original: the unit is fitted with a kitchen .
Generated: the accommodation is equipped with a kitchen .

Source: ванная комната оборудована душем .
Original: the bathroom has a shower .
Generated: the bathroom comes with a shower .

Source: в гостиной установлен камин .
Original: there is also a fireplace in the living room .
Generated: living room is a fireplace .

Source: в распоряжении гостей кофемашина .
Original: you will find a coffee machine in the room .
Generated: you will find a coffee machine in the room .
```

Evaluate Validation BLEU

```
from nltk.translate.bleu_score import corpus_bleu

#     """ Estimates corpora-level BLEU score of model's translations given inp and
#     translations, _ = model.translate_lines(inp_lines, **flags)
#     # Note: if you experience out-of-memory error, split input lines into batches
#     return corpus_bleu([[ref] for ref in out_lines], translations) * 100


val_bleu = evaluate_bleu(model, valid_iterator, TRG.vocab)
print(f'Validation BLEU {val_bleu:.3}')
```

```
100%                                        20/20 [00:04<00:00, 4.28it/s]

Validation BLEU 32.2
```

Recommendations:

- use bidirectional RNN ✅
- you can use more than one layer ✅
- change learning rate from epoch to epoch ✅
- when classifying the word don't forget about embedding and sum of encoder's states (attention) ✅

Test BLEU

Only run the next cell when satisfied with the validation BLEU!

```
check = input('Are you satisfied with your Validation BLEU? (yes/no)')

if check == 'yes':
    test_bleu = evaluate_bleu(model, test_iterator, TRG.vocab)
    print(f'Test BLEU: {test_bleu:.3}\n')
else:
    print('Do not overfit on the test set!')
```

```
Are you satisfied with your Validation BLEU? (yes/no)yes
100%                                        59/59 [00:13<00:00, 4.52it/s]

Test BLEU: 30.3
```

Your Conclusion

Information about your the results obtained

See Scoring.

Difference between seminar and homework model

- Seminar: LSTM cells for encoder and decoder; HW: LSTM for encoder, GRU for decoder. In the paper - GRU for both.
- Different number of layers possible here, impossible in the seminar notebook.
- Attention mechanism to convey info from all the hidden units (in the last layer) of encoder to decoder
- Added here vs the seminar:
  - Learning-rate scheduler
  - Early Stopping
- Better BLEU score

Scoring

You will get:

- `2` points if `21 < bleu score < 23`
- `4` points if `23 < bleu score < 25`
- `7` points if `25 < bleu score < 27`
- `9` points if `27 < bleu score < 29`
- `10` points if `bleu score > 29` ✅
  - v1: my initial version
    - one-layer encoder (bidirectional) and decoder with attention
    - concat attention with `outputs` instead of `hidden[-1::].unsqueeze(0)` to the last prediction layer in decoder (see commented lines)
    - training for 20 epochs w/o scheduler, default optimizer
    - `min_freq` = 2
  - v2: closer to the paper
    - two-layer encoder (bidirectional) and decoder with attention
    - concat attention like in the paper
    - Scheduler `ReduceLROnPlateau(valid_bleu)`, default optimizer
    - 30 epochs + early stopping `patience` =5, stopped after 27 epochs - no need to abuse colab :)
    - `min_freq` = 3

When your result is checked, your 10 translations will be checked too

✓ 18s completed at 15:51