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 (https://github.com/yandexdataschool/nlp_course)

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:

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

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.

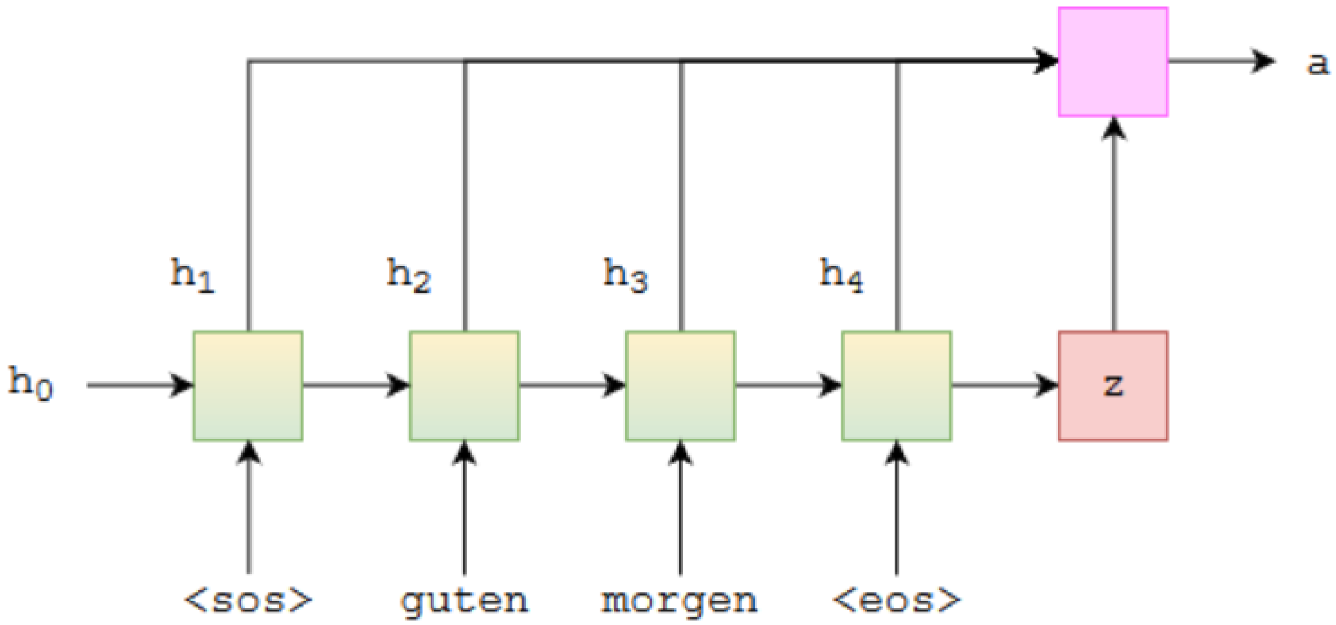
$$a_t = \mathrm{softmax}(\hat{a_t})$$

Temperature SoftMax

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

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.



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.

Ввод [ ]:

```
! wget https://drive.google.com/uc?id=1NWYqJgeG_4883LINdEjKUr6nLQPY6Yb_ -O data.txt

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

executed in 12ms, finished 12:16:10 2021-11-05

```
--2021-11-05 14:04:13--  https://drive.google.com/uc?id=1NWYqJgeG_4883LINdEjKUr6nLQPY6Yb_ (https://drive.google.com/uc?id=1NWYqJgeG_4883LINdEjKUr6nLQPY6Yb_)
Resolving drive.google.com (drive.google.com)... 173.194.194.101, 173.194.194.100, 173.194.194.138, ...
Connecting to drive.google.com (drive.google.com)|173.194.194.101|:443... connected.
HTTP request sent, awaiting response... 302 Moved Temporarily
Location: https://doc-14-00-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/p1tu90crt61qdjh90lagl1m5uv4dnj82/1636121025000/16549096980415837553/*/1NWYqJgeG_4883LINdEjKUr6nLQPY6Yb_ (https://doc-14-00-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/p1tu90crt61qdjh90lagl1m5uv4dnj82/1636121025000/16549096980415837553/*/1NWYqJgeG_4883LINdEjKUr6nLQPY6Yb_) [following]
Warning: wildcards not supported in HTTP.
--2021-11-05 14:04:14--  https://doc-14-00-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/p1tu90crt61qdjh90lagl1m5uv4dnj82/1636121025000/16549096980415837553/*/1NWYqJgeG_4883LINdEjKUr6nLQPY6Yb_ (https://doc-14-00-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/p1tu90crt61qdjh90lagl1m5uv4dnj82/1636121025000/16549096980415837553/*/1NWYqJgeG_4883LINdEjKUr6nLQPY6Yb_)
Resolving doc-14-00-docs.googleusercontent.com (doc-14-00-docs.googleusercontent.com)... 173.194.197.132, 2607:f8b0:4001:c1b::84
Connecting to doc-14-00-docs.googleusercontent.com (doc-14-00-docs.googleusercontent.com)|173.194.197.132|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12905334 (12M) [text/plain]
Saving to: ‘data.txt’

data.txt            100%[===================>]  12.31M  --.-KB/s    in 0.1s

2021-11-05 14:04:14 (117 MB/s) - ‘data.txt’ saved [12905334/12905334]
```

Ввод [ ]:

```
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 numpy as np

import warnings
warnings.filterwarnings(action='ignore')

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
```

executed in 26.4s, finished 12:16:36 2021-11-05

We'll set the random seeds for deterministic results.

Ввод [ ]:

```
SEED = 1234

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

executed in 29ms, finished 12:16:36 2021-11-05

Preparing Data

Here comes the preprocessing

Ввод [ ]:

```
tokenizer_W = WordPunctTokenizer()

def tokenize_ru(x, tokenizer=tokenizer_W):
    return tokenizer.tokenize(x.lower())[::-1]

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

executed in 13ms, finished 12:16:36 2021-11-05

Ввод [ ]:

```
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.txt',
    format='tsv',
    fields=[('trg', TRG), ('src', SRC)]
)
```

executed in 2.59s, finished 12:16:39 2021-11-05

Ввод [ ]:

```
print(len(dataset.examples))
print(dataset.examples[0].src)
print(dataset.examples[0].trg)
```

executed in 14ms, finished 12:16:39 2021-11-05

```
50000
['.', 'собора', 'троицкого', '-', 'свято', 'от', 'ходьбы', 'минутах', '3',
'в', ',', 'тбилиси', 'в', 'расположен', 'cordelia', 'отель']
['cordelia', 'hotel', 'is', 'situated', 'in', 'tbilisi', ',', 'a', '3', '-',
'minute', 'walk', 'away', 'from', 'saint', 'trinity', 'church', '.']
```

Ввод [ ]:

```
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)}")
```

executed in 62ms, finished 12:16:39 2021-11-05

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

Ввод [ ]:

```
SRC.build_vocab(train_data, min_freq = 2)
TRG.build_vocab(train_data, min_freq = 2)
```

executed in 548ms, finished 12:16:40 2021-11-05

Ввод [ ]:

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

executed in 11ms, finished 12:16:40 2021-11-05

```
Unique tokens in source (ru) vocabulary: 14129
Unique tokens in target (en) vocabulary: 10104
```

And here is example from train dataset:

Ввод [ ]:

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

executed in 13ms, finished 12:16:40 2021-11-05

```
{'trg': ['other', 'facilities', 'offered', 'at', 'the', 'property', 'includ
e', 'grocery', 'deliveries', ',', 'laundry', 'and', 'ironing', 'services',
'.'], 'src': ['.', 'услуги', 'гладильные', 'и', 'прачечной', 'услуги', ',',
'продуктов', 'доставка', 'предлагается', 'также']}
```

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.

Ввод [ ]:

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

executed in 12ms, finished 12:16:40 2021-11-05

Ввод [ ]:

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

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
)
```

executed in 14ms, finished 12:16:40 2021-11-05

Let's use modules.py

Ввод [ ]:

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

executed in 14ms, finished 12:16:40 2021-11-05

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

Ввод [ ]:

```
!ls drive/MyDrive/data
```

executed in 14ms, finished 12:16:40 2021-11-05

```
data.txt  modules.py  __pycache__  svdTfit.pkl  word2vec-google-news-300
```

Ввод [ ]:

```
%cd ./drive/MyDrive/data
```

executed in 14ms, finished 12:16:40 2021-11-05

```
/content/drive/MyDrive/data
```

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:

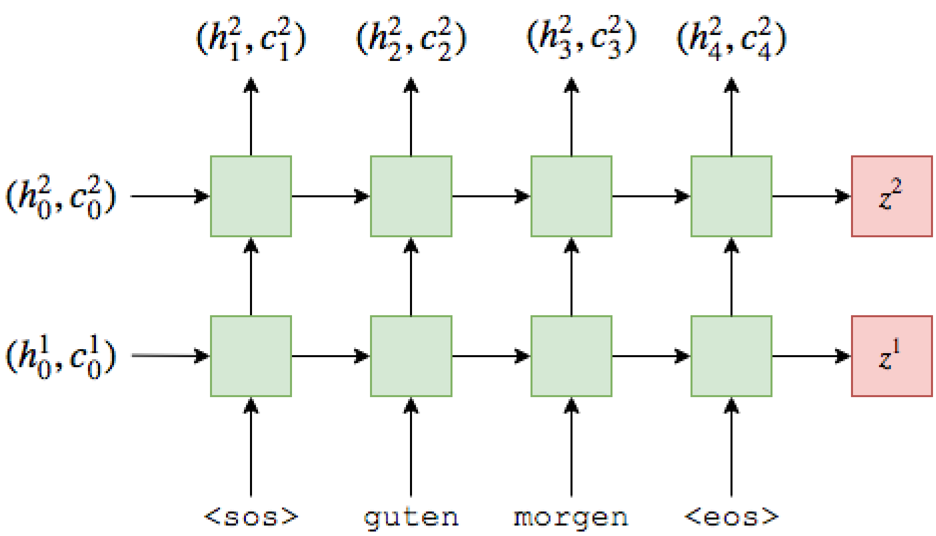
$$h_t^1 = \text{EncoderRNN}^1(x_t, h_{t-1}^1)$$

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:

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



Ввод [ ]:

```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, dropout, bidirectional):
        super().__init__()

        self.input_dim = input_dim
        self.emb_dim = emb_dim
        self.hid_dim = hid_dim
        self.n_layers = n_layers
        self.dropout = dropout
        self.bidirectional = bidirectional

        self.embedding = nn.Embedding(input_dim, emb_dim)

        self.rnn = nn.LSTM(emb_dim, hid_dim, num_layers=n_layers, dropout=dropout, bidirect

        self.dropout = nn.Dropout(p=dropout)

    def forward(self, src):

        #src = [src sent len, batch size]
        # Compute an embedding from the src data and apply dropout to it
        embedded = self.dropout(self.embedding(src))
        #embedded = [src sent len, batch size, emb dim]

        # Compute the RNN output values of the encoder RNN.
        # outputs, hidden and cell should be initialized here. Refer to nn.LSTM docs ;)

        outputs, (hidden, cell) = self.rnn(embedded)
        #outputs = [src sent len, batch size, hid dim * n directions]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]

        #outputs are always from the top hidden layer
        if self.bidirectional:
            hidden = hidden.reshape(self.n_layers, 2, -1, self.hid_dim) # n_layers, 2, batc
            hidden = hidden.transpose(1, 2).reshape(self.n_layers, -1, 2 * self.hid_dim) #
            cell = cell.reshape(self.n_layers, 2, -1, self.hid_dim) # n_layers, 2, batch si
            cell = cell.transpose(1, 2).reshape(self.n_layers, -1, 2 * self.hid_dim) # n_la

        return outputs, hidden, cell
```

executed in 14ms, finished 12:16:40 2021-11-05

Attention

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

Ввод [ ]:

```
def softmax(x, temperature=15): # use your temperature
    e_x = torch.exp(x / temperature)
    return e_x / torch.sum(e_x, dim=0)
```

executed in 14ms, finished 12:16:40 2021-11-05

Ввод [ ]:

```
class Attention(nn.Module):
    def __init__(self, enc_hid_dim, dec_hid_dim):
        super().__init__()

        self.enc_hid_dim = enc_hid_dim
        self.dec_hid_dim = dec_hid_dim

        self.attn = nn.Linear(enc_hid_dim + dec_hid_dim, enc_hid_dim)
        self.v = nn.Linear(enc_hid_dim, 1)

    def forward(self, hidden, encoder_outputs):

        # encoder_outputs = [src sent len, batch size, enc_hid_dim]
        # [src sent len, batch size, hid dim * enc_n directions]
        # hidden = [1, batch size, dec_hid_dim]
        #print('attn hidden', hidden.size())
        #print('attn encoder_outputs', encoder_outputs.size())
        # repeat hidden and concatenate it with encoder_outputs
        hidden = hidden.repeat(encoder_outputs.shape[0], 1, 1)
        #print('attn hidden_rep', hidden.size())
        hidden = torch.cat((hidden, encoder_outputs),2)
        #print('attn hidden_rep_cat', hidden.size())
        # calculate energy
        energy = torch.tanh(self.attn(hidden))
        #print('attn energy', energy.size())
        # get attention, use softmax function which is defined, can change temperature
        attn = softmax(self.v(energy))
        #print('attn softmax', attn.size())
        return attn
```

executed in 14ms, finished 12:16:40 2021-11-05

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}, all of the encoder hidden states H, and returns the attention vector a_t.

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

$$w_t = a_t H$$

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

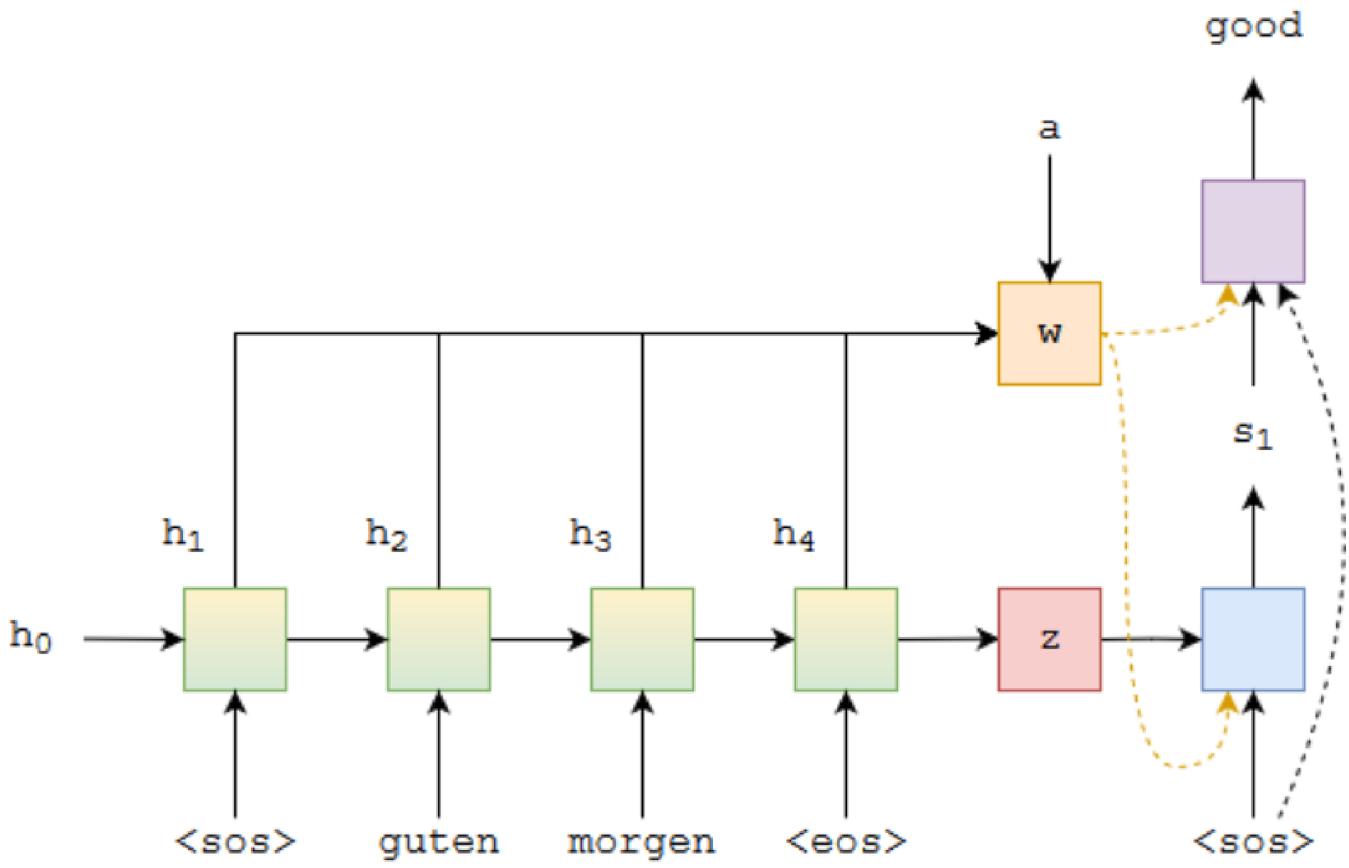
$$s_t = \text{DecoderGRU}([y_t, w_t], s_{t-1})$$

We then pass y_t, w_t and s_t 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.

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

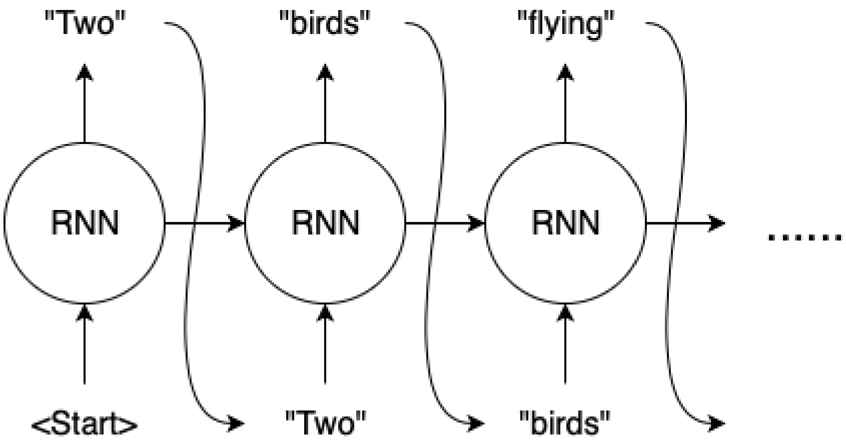
The image below shows decoding the **first** word in an example translation.

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.

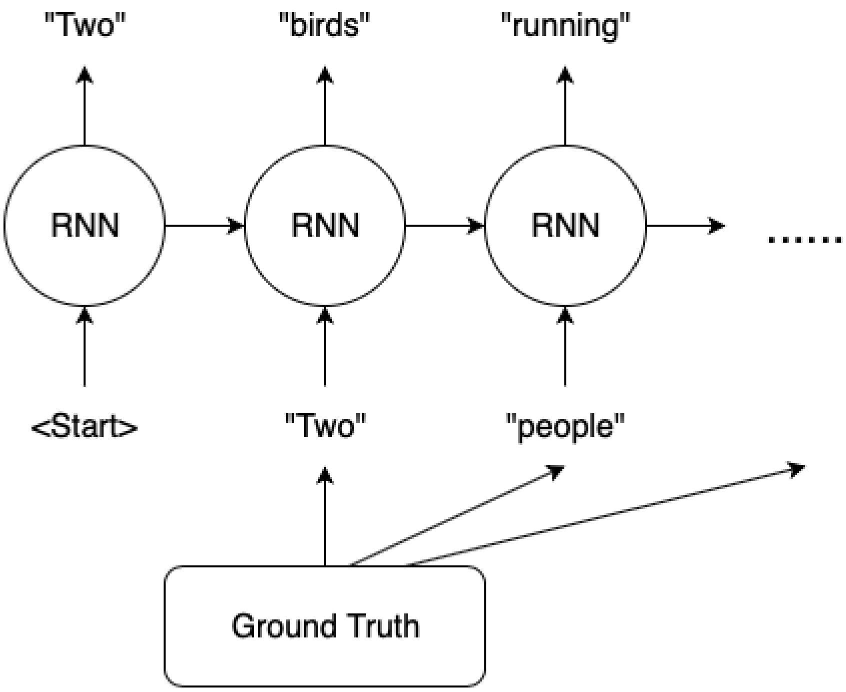


Teacher forcing

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.

Ввод [ ]:

```
class DecoderWithAttention(nn.Module):
    def __init__(self, output_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout, attention):
        super().__init__()

        self.emb_dim = emb_dim
        self.enc_hid_dim = enc_hid_dim
        self.dec_hid_dim = dec_hid_dim
        self.output_dim = output_dim
        self.attention = attention

        self.embedding = nn.Embedding(output_dim, emb_dim)

        self.rnn = nn.GRU(emb_dim+enc_hid_dim, dec_hid_dim,num_layers=1,dropout=dropout) #

        self.out = nn.Linear(dec_hid_dim*3, output_dim) # linear layer to get next word

        self.dropout = nn.Dropout(dropout)

    def forward(self, input_data, hidden, encoder_outputs):
        #hidden = [n layers * n directions, batch size, hid dim]
        #n directions in the decoder will both always be 1, therefore:
        #hidden = [n layers, batch size, hid dim]

        input_data = input_data.unsqueeze(0) # because only one word, no words sequence

        #input_data = [1, batch size]

        embedded = self.dropout(self.embedding(input_data))
        #embedded = [1, batch size, emb dim]

        attn = self.attention(hidden[-1],encoder_outputs)

        # get weighted sum of encoder_outputs
        weighted = torch.bmm(attn.permute(1,2,0),encoder_outputs.permute(1,0,2)) #'''your c
        # concatenate weighted sum and embedded, break through the GRU
        weighted = weighted.permute(1,0,2)

        output, hidden = self.rnn(torch.cat((embedded,weighted),2),hidden[-1].unsqueeze(0))

        # get predictions
        embedded = embedded.squeeze(0)
        output = output.squeeze(0)
        weighted = weighted.squeeze(0)
        prediction = self.out(torch.cat([output,weighted,hidden.squeeze(0)],1))
        #prediction = [batch size, output dim]
        return prediction, hidden
```

executed in 14ms, finished 12:16:40 2021-11-05

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:

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

Ввод [ ]:

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.device = device

        assert encoder.hid_dim*(encoder.bidirectional + 1) == decoder.dec_hid_dim, \
            "Hidden dimensions of encoder and decoder must be equal!"

    def forward(self, src, trg, teacher_forcing_ratio = 0.5):

        # src = [src sent len, batch size]
        # trg = [trg sent len, batch size]
        # teacher_forcing_ratio is probability to use teacher forcing
        # e.g. if teacher_forcing_ratio is 0.75 we use ground-truth inputs 75% of the time

        # Again, now batch is the first dimention instead of zero
        batch_size = trg.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        #tensor to store decoder outputs
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)

        #last hidden state of the encoder is used as the initial hidden state of the decode
        enc_states, hidden, cell = self.encoder(src)
        #first input to the decoder is the <sos> tokens
        input_data = trg[0,:]

        for t in range(1, trg_len):
            output, hidden = self.decoder(input_data, hidden, enc_states) #'''your code'''
            outputs[t] = output
            #decide if we are going to use teacher forcing or not
            teacher_force = random.random() < teacher_forcing_ratio
            #get the highest predicted token from our predictions
            top1 = output.argmax(-1)
            #if teacher forcing, use actual next token as next input
            #if not, use predicted token
            input_data = trg[t] if teacher_force else top1

        return outputs
```

executed in 14ms, finished 12:16:40 2021-11-05

Training

Ввод [ ]:

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

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

executed in 14ms, finished 12:16:40 2021-11-05

Ввод [ ]:

```
INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
ENC_EMB_DIM = 192
DEC_EMB_DIM = 192
HID_DIM = 384
N_LAYERS = 1 # simple model: n_layers=1
ENC_DROPOUT = 0.4
DEC_DROPOUT = 0.4
BIDIRECTIONAL = True

enc = Encoder(INPUT_DIM, ENC_EMB_DIM, HID_DIM//2, N_LAYERS, ENC_DROPOUT, BIDIRECTIONAL)
attention = Attention(HID_DIM,HID_DIM)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, HID_DIM, HID_DIM, DEC_DROPOUT, attention)

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

executed in 4.89s, finished 12:16:45 2021-11-05

Ввод [ ]:

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

model.apply(init_weights)
```

executed in 13ms, finished 12:16:45 2021-11-05

Out[24]:

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(14129, 192)
    (rnn): LSTM(192, 192, dropout=0.4, bidirectional=True)
    (dropout): Dropout(p=0.4, inplace=False)
  )
  (decoder): DecoderWithAttention(
    (attention): Attention(
      (attn): Linear(in_features=768, out_features=384, bias=True)
      (v): Linear(in_features=384, out_features=1, bias=True)
    )
    (embedding): Embedding(10104, 192)
    (rnn): GRU(576, 384, dropout=0.4)
    (out): Linear(in_features=1152, out_features=10104, bias=True)
    (dropout): Dropout(p=0.4, inplace=False)
  )
)
```

Ввод [ ]:

```
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')
```

executed in 14ms, finished 12:16:45 2021-11-05

```
The model has 18,299,449 trainable parameters
```

Ввод [ ]:

```
from torch.optim.lr_scheduler import StepLR
PAD_IDX = TRG.vocab.stoi['<pad>']
optimizer = optim.Adam(model.parameters(),lr=0.002)
scheduler = StepLR(optimizer, step_size=5, gamma=0.5)
criterion = nn.CrossEntropyLoss(ignore_index = PAD_IDX)

def train(model, iterator, optimizer, criterion, clip, train_history=None, valid_history=No
    model.train()

    epoch_loss = 0
    history = []
    for i, batch in enumerate(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())
        if (i+1)%10==0:
            fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 8))

            clear_output(True)
            ax[0].plot(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='general train history')
                ax[1].set_xlabel('Epoch')
            if valid_history is not None:
                ax[1].plot(valid_history, label='general valid history')
            plt.legend()

            plt.show()


    return epoch_loss / len(iterator)
```

```
def evaluate(model, iterator, criterion):

    model.eval()

    epoch_loss = 0

    history = []

    with torch.no_grad():

        for i, batch in enumerate(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)

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
```

executed in 19ms, finished 12:16:47 2021-11-05

Ввод [ ]:

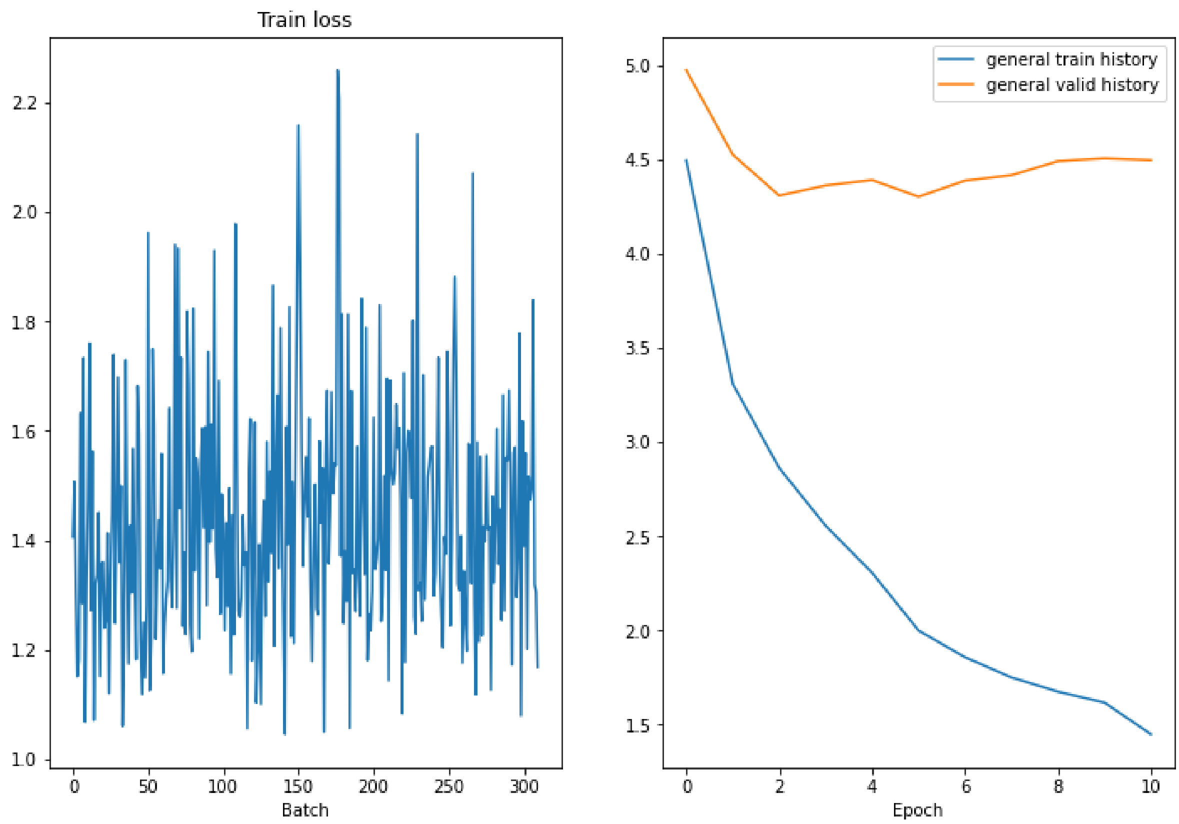
```
import matplotlib
matplotlib.rcParams.update({'figure.figsize': (16, 12), 'font.size': 14})
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import clear_output
```

executed in 7ms, finished 12:16:48 2021-11-05

Ввод [ ]:

```
train_history = []
valid_history = []

N_EPOCHS = 12
CLIP = 6

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP, train_history, va
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'best-val-model.pt')

    curr_lr = optimizer.param_groups[0]['lr']
    train_history.append(train_loss)
    valid_history.append(valid_loss)
    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):7.3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')
    print(f'Epoch {epoch}\tLR:{curr_lr}')
    scheduler.step()
```

executed in 34m 50s, finished 12:51:38 2021-11-05



```
Epoch: 12 | Time: 6m 30s
        Train Loss: 1.441 | Train PPL:   4.227
         Val. Loss: 4.550 |  Val. PPL:  94.672
Epoch 11        LR:0.0005
```

Let's take a look at our network quality:

Ввод [ ]:

```
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):
    model.eval()

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

    original = remove_tech_tokens(cut_on_eos([TRG_vocab.itos[x] for x in list(trg[:,0].cpu(
    generated = remove_tech_tokens(cut_on_eos([TRG_vocab.itos[x] for x in list(output[:, 0]

    print('Original: {}'.format(' '.join(original)))
    print('Generated: {}'.format(' '.join(generated)))
    print()

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

executed in 15ms, finished 12:54:20 2021-11-05

Ввод [ ]:

```
#model.load_state_dict(torch.load('best-val-model.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)
```

executed in 525ms, finished 12:54:21 2021-11-05

```
Original: there is a 24 - hour front desk at the property .
Generated: there is a 24 - hour front desk at the property .

Original: you will find a 24 - hour front desk at the property .
Generated: there is a 24 - hour front desk at the property .

Original: there is a 24 - hour front desk at the property .
Generated: there is a 24 - hour front desk at the property .

Original: free private parking is available .
Generated: free private parking is available on site .

Original: there are several restaurants in the surrounding area .
Generated: several restaurants restaurants can be found nearby .

Original: the property also offers free parking .
Generated: the property offers free parking .

Original: the unit is fitted with a kitchen .
Generated: the unit is equipped with a kitchen .

Original: the bathroom has a shower .
Generated: the bathroom comes with a shower .

Original: there is also a fireplace in the living room .
Generated: the living room is a fireplace .

Original: you will find a coffee machine in the room .
Generated: you will find a coffee machine in the room .
```

Bleu

[link (https://www.aclweb.org/anthology/P02-1040.pdf)](https://www.aclweb.org/anthology/P02-1040.pdf)

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}.$$

Then,

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right).$$

The ranking behavior is more immediately apparent in the log domain,

$$\log \text{BLEU} = \min(1 - \frac{r}{c}, 0) + \sum_{n=1}^{N} w_n \log p_n.$$

In our baseline, we use $N = 4$ and uniform weights $w_n = 1/N$.

Ввод [ ]:

```
from nltk.translate.bleu_score import corpus_bleu

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

executed in 14ms, finished 12:54:27 2021-11-05

Ввод [ ]:

```
import tqdm
original_text = []
generated_text = []
model.eval()
with torch.no_grad():

    for i, batch in tqdm.tqdm(enumerate(test_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().T])
        generated_text.extend([get_text(x, TRG.vocab) for x in output.detach().cpu().numpy(

# original_text = flatten(original_text)
# generated_text = flatten(generated_text)
```

executed in 7.68s, finished 12:54:36 2021-11-05

```
59it [00:12,  4.72it/s]
```

Ввод [ ]:

```
corpus_bleu([[text] for text in original_text], generated_text) * 100
```

executed in 1.16s, finished 12:54:37 2021-11-05

Out[33]:

```
31.368944788068156
```

Recommendations:

- use bidirectional RNN
- change learning rate from epoch to epoch
- when classifying the word don't forget about embedding and summa of encoders state
- you can use more than one layer

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`

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

Your Conclusion

- information about your the results obtained
- difference between seminar and homework model

1) BLEU: 31.37 (показатель колеблится, но не ниже 29)

За 12 эпох при последнем обучении Train Loss: 1.441 | Train PPL: 4.227; Val. Loss: 4.550 | Val. PPL: 94.672. Переводы модели получились практически идентичные правильным.

2) Основные различия и оссобенности: разница в размерностях скрытых слоёв (они были уменьшены 512->384 и 256->192) и dropout (0.5->0.4) (хотя это не должно влиять на однослойные RNN), bidirectional также используется; из энкодера теперь выводится output; в декодере теперь используется GRU и изменён forward, в соответствии с теорией (т.к. используется attention); в данной модели теперь используется concat-attention; forwdard Seq2Seq'a тоже изменён из-за attention (в декодер передаются все скрытые состояния энкодера) CLIP изменён 5->6 для более быстрого обучения; изменён lr Adam'a 0.001->0.002 и теперь используется шедулер, который делит lr на 2 каждые 5 эпох для достижения более низкого лосса. Повысил температуру softmax'a до 10->15.

Ввод [ ]: