Физтех-Школа Прикладной математики и информатики (ФПМИ) МФТИ

Some parts of the notebook are almost the exact copy of [ML-MIPT course](). Special thanks to ML-MIPT team for making them publicly available. [Original notebook]().

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:

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

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. ⇒
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.

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

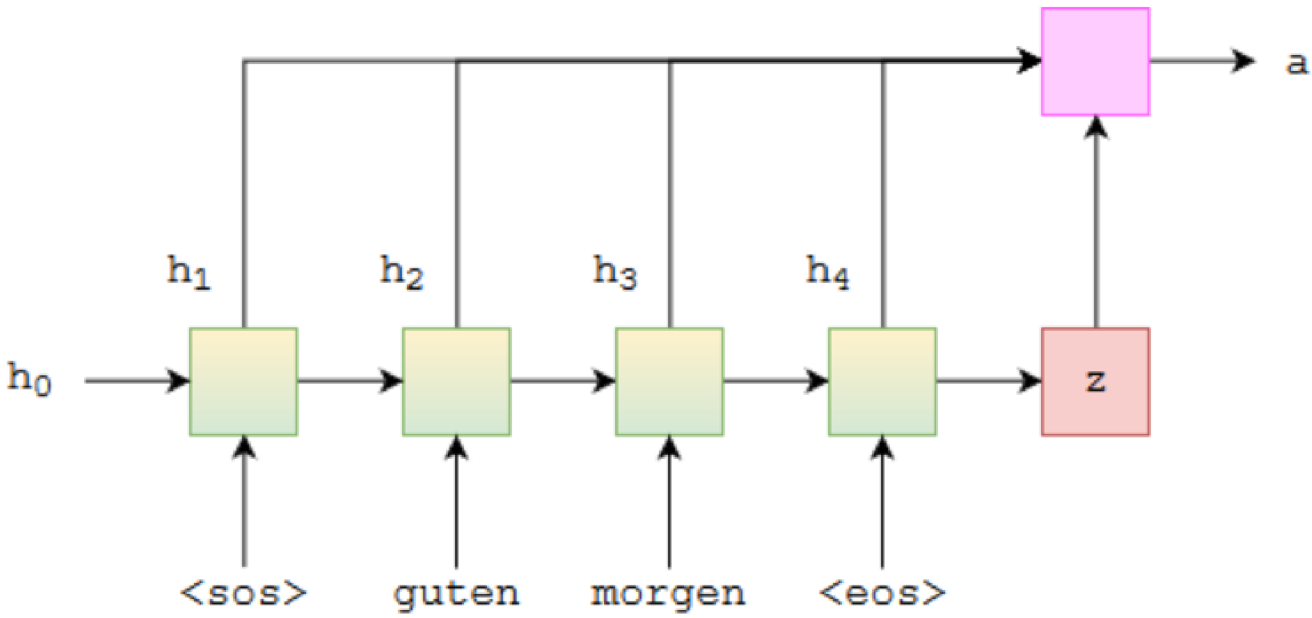
We can think of this as calculating a weighted sum of the "match" over all `enc_hid_dem` elements for each encoder hidden state, where the weights are learned (as we learn the parameters of v).

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 softmax layer.

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

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.



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.

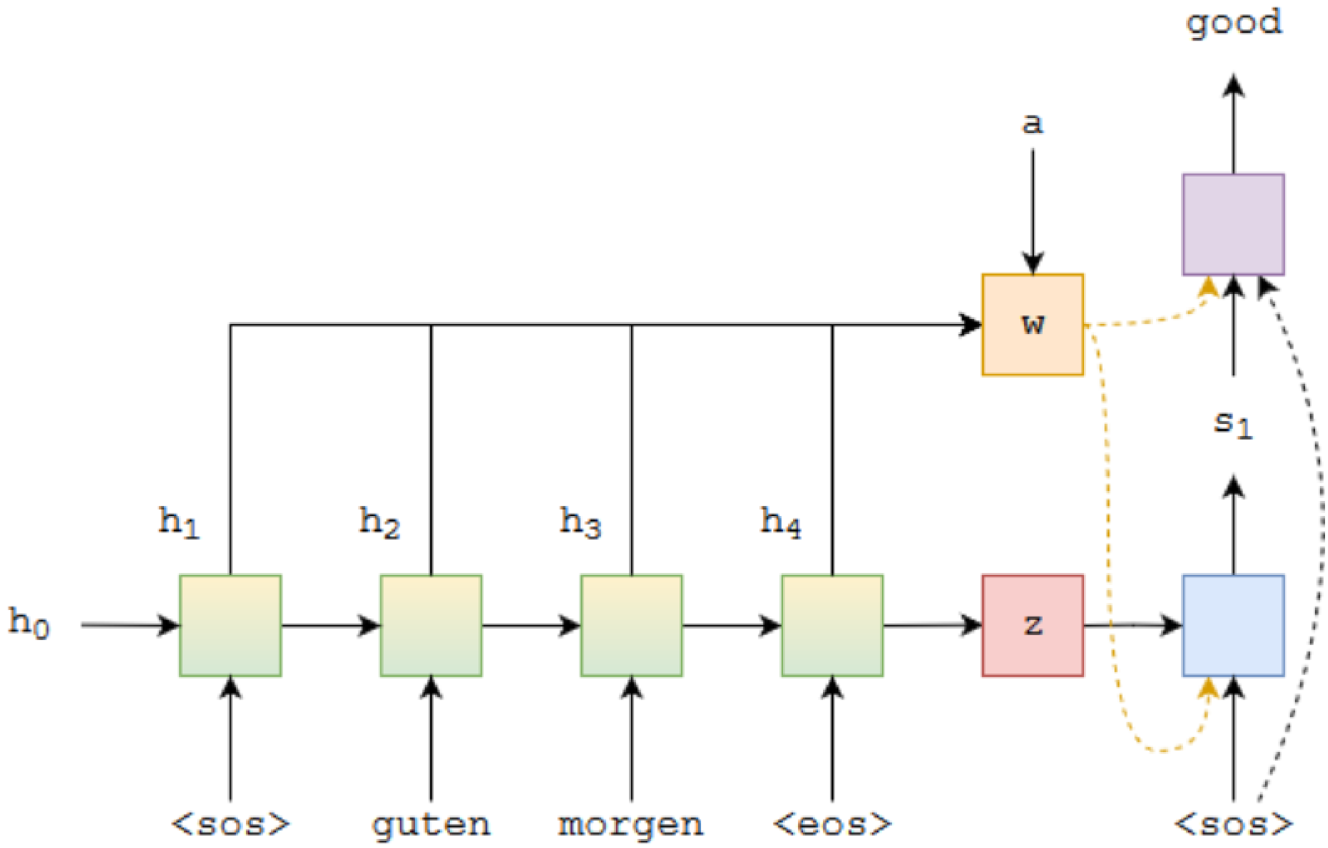
$$s_t = \text{DecoderGRU}([y_t, w_t], s_{t-1})$$

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.

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

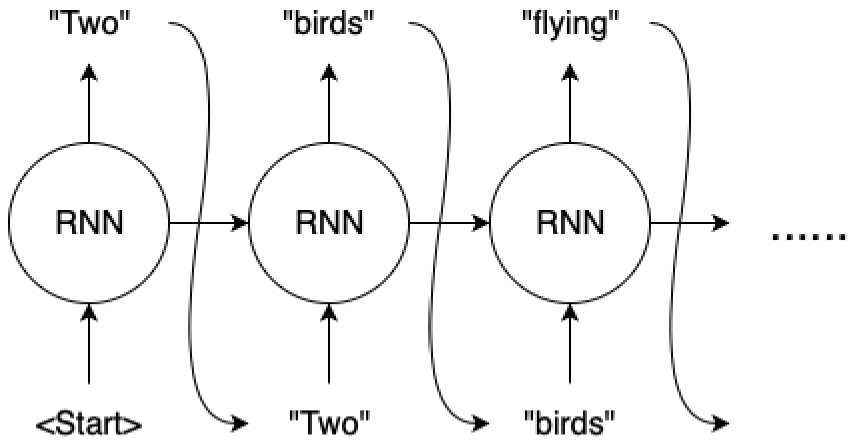
The image below shows decoding the **first** word in an example translation.

The green/yellow blocks show the forward/backward encoder RNNs which output H, the red block is $z = s_{t-1} = s_0$ in this moment and $s_0 = h_4$, 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.

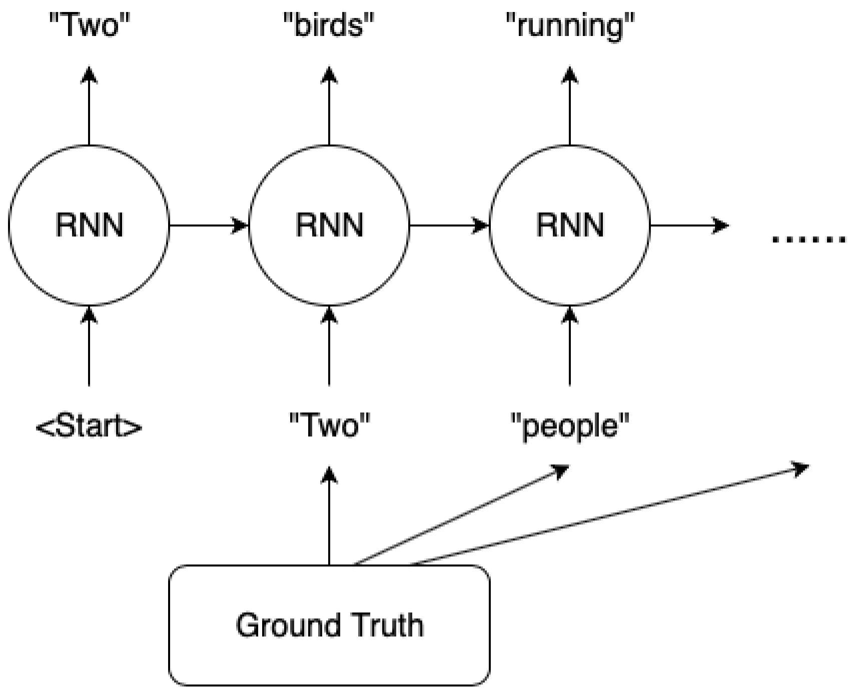


Teacher forcing

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

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.

```
! pip  install subword-nmt
! pip install nltk
! pip install torchtext
! wget https://raw.githubusercontent.com/girafe-ai/ml-mipt/advanced/homeworks/Lab1_

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

```
Collecting subword-nmt
  Downloading https://files.pythonhosted.org/packages/74/60/6600a7bc09e7ab38b
Installing collected packages: subword-nmt
Successfully installed subword-nmt-0.3.7
Requirement already satisfied: nltk in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: torchtext in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: torch in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: tqdm in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: future in /usr/local/lib/python3.6/dist-packag
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /us
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-
--2020-10-31 21:02:44--  https://raw.githubusercontent.com/girafe-ai/ml-mipt/
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.0.
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.0
HTTP request sent, awaiting response... 200 OK
Length: 12905334 (12M) [text/plain]
Saving to: 'data.txt'

data.txt            100%[===================>]  12.31M  22.1MB/s    in 0.6s

2020-10-31 21:02:45 (22.1 MB/s) - 'data.txt' saved [12905334/12905334]
```

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

import torchtext
from torchtext.datasets import TranslationDataset, Multi30k
from torchtext.data import Field, BucketIterator

import spacy

import random
import math
import time

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
from subword_nmt.learn_bpe import learn_bpe
from subword_nmt.apply_bpe import BPE
```

Main part

Here comes the preprocessing. Try to use BPE or more complex preprocessing :)

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


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

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


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


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


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

And here is example from train dataset:

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

```
{'trg': ['the', 'en', 'suite', 'bathroom', 'includes', 'a', 'bathrobe', ',',
```

Model side

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


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

BATCH_SIZE = 256 #'''your code'''

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
)
```

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

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

INPUT_DIM = len(SRC.vocab) #'''your code'''
OUTPUT_DIM = len(TRG.vocab) #'''your code'''
ENC_EMB_DIM = 512 #'''your code'''
DEC_EMB_DIM = 512 #'''your code'''
HID_DIM = 512    #'''your code'''
N_LAYERS = 1
ENC_DROPOUT = 0.5
DEC_DROPOUT = 0.5

enc = Encoder(INPUT_DIM, ENC_EMB_DIM, HID_DIM, N_LAYERS, ENC_DROPOUT)
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)
```

```
/usr/local/lib/python3.6/dist-packages/torch/nn/modules/rnn.py:60: UserWarnin
  "num_layers={}".format(dropout, num_layers))
```

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

model.apply(init_weights)
```

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(9321, 512)
    (rnn): LSTM(512, 512, dropout=0.5)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (decoder): DecoderWithAttention(
    (attention): Attention(
      (attn): Linear(in_features=1024, out_features=512, bias=True)
      (v): Linear(in_features=512, out_features=1, bias=True)
    )
    (embedding): Embedding(6715, 512)
    (rnn): GRU(512, 512, dropout=0.5)
    (out): Linear(in_features=512, out_features=6715, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
```

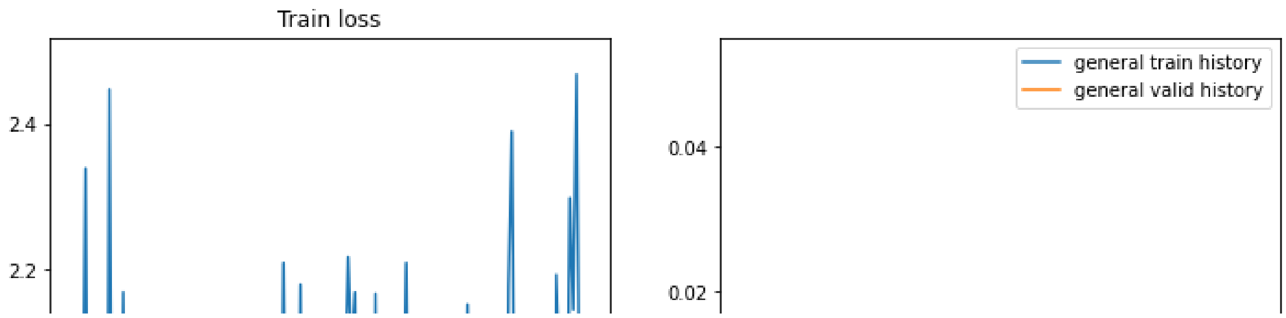
```
        )
    )


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 15,857,724 trainable parameters
```

```
PAD_IDX = TRG.vocab.stoi['<pad>']
optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss(ignore_index = PAD_IDX)

def train(model, iterator, optimizer, criterion, clip, train_history=None, valid_h
    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.shape[-1])
        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.shape[-1])
            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


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_history = []
valid_history = []

N_EPOCHS = 1
CLIP = 1

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_his
    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')

    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}
```



Let's take a look at our network quality:

```
import utils
import imp
imp.reload(utils)
generate_translation = utils.generate_translation
remove_tech_tokens = utils.remove_tech_tokens
get_text = utils.get_text
flatten = utils.flatten
```

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

for idx in [1,2]:
    src = batch.src[:, idx:idx+1]
    trg = batch.trg[:, idx:idx+1]
    generate_translation(src, trg, model, TRG.vocab)
```

```
Original: younger guests will enjoy a children ’ s playground .
Generated: vyšehrad children ' s playground . children ' s playground .

Original: rooms are bright and well - appointed .
Generated: vyšehrad bright rooms are all are all the .
```

Bleu

[link]()

```
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 batche:
#     return corpus_bleu([[ref] for ref in out_lines], translations) * 100
```

```
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.argmax(dim=-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

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

```
30it [00:04,  7.00it/s]
```

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

```
23.18564241375928
```

Baseline solution BLEU score is quite low. The checkpoints are:

- **20** - minimal score to submit - 10 эпох.
- **25** - good score to submit