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

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:

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

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:

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

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 $\operatorname{softmax}$ layer.

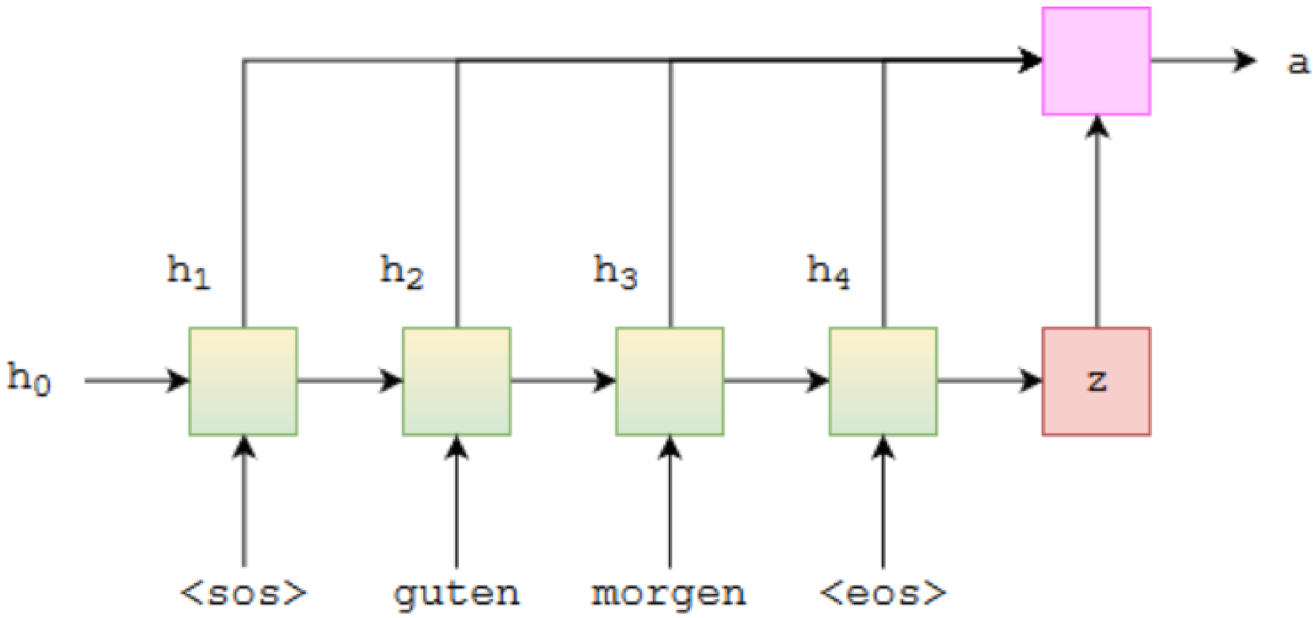
$$a_t = \operatorname{softmax}(\hat{a}_t)$$

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. $z = s_{t-1}$. 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.

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

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

```
--2021-11-12 19:47:09--  https://drive.google.com/uc?id=1NWYqJgeG_4883LINdEjK
Resolving drive.google.com (drive.google.com)... 173.194.216.102, 173.194.216
Connecting to drive.google.com (drive.google.com)|173.194.216.102|:443... con
HTTP request sent, awaiting response... 302 Moved Temporarily
Location: https://doc-14-00-docs.googleusercontent.com/docs/securesc/ha0ro937
Warning: wildcards not supported in HTTP.
--2021-11-12 19:47:11--  https://doc-14-00-docs.googleusercontent.com/docs/se
Resolving doc-14-00-docs.googleusercontent.com (doc-14-00-docs.googleusercont
Connecting to doc-14-00-docs.googleusercontent.com (doc-14-00-docs.googleuser
HTTP request sent, awaiting response... 200 OK
Length: 12905334 (12M) [text/plain]
```

```
Saving to: ‘data.txt’

data.txt            100%[===================>]  12.31M  --.-KB/s    in 0.08s

2021-11-12 19:47:11 (151 MB/s) - ‘data.txt’ saved [12905334/12905334]
```

```
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 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]

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

```
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)]
)


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

```
50000
['.', 'собора', 'троицкого', '-', 'свято', 'от', 'ходьбы', 'минутах', '3', 'в
['cordelia', 'hotel', 'is', 'situated', 'in', 'tbilisi', ',', 'a', '3', '-',
```

```
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)}")
```

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

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


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

And here is example from train dataset:

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

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

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.

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

```
!nvidia-smi
```

```
Fri Nov 12 19:47:31 2021
+-----------------------------------------------------------------------------
| NVIDIA-SMI 450.119.04   Driver Version: 450.119.04   CUDA Version: 11.0
|-------------------------------+----------------------+----------------------
| 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 P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0
| N/A   45C    P0    35W / 250W |  14977MiB / 16280MiB |      0%      Default
|                               |                      |                  N/A
+-------------------------------+----------------------+----------------------

+-----------------------------------------------------------------------------
| Processes:
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory
|        ID   ID                                                   Usage
|=============================================================================
+-----------------------------------------------------------------------------
```

```
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
)
```

Let's use modules.py

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

```
!cp best-val-model.pt drive/MyDrive/best-model.pt
```

```
!ls your_path_to_modules.py
```

```
%cd ./drive/MyDrive/your_path_to_modules.py
```

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:

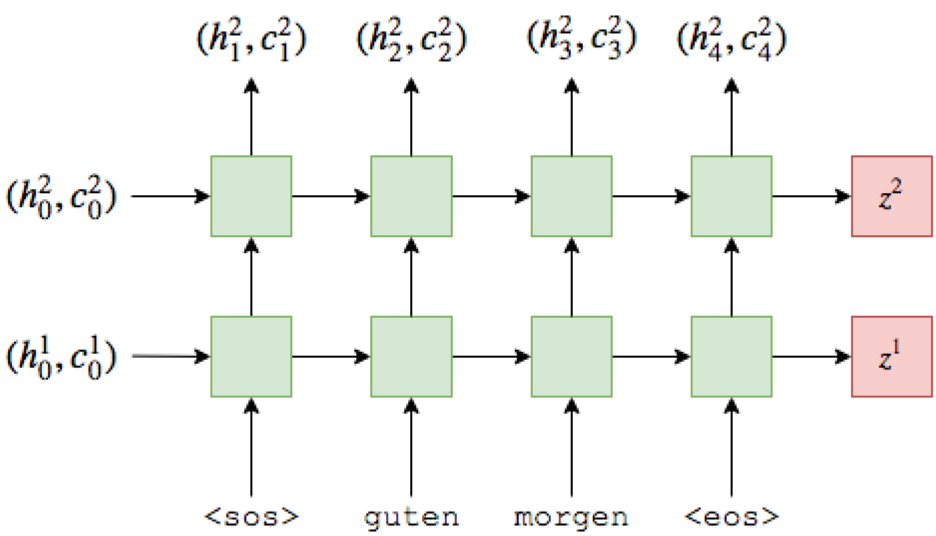
$$h_t^1 = \text{EncoderRNN}^1(x_t, h_{t-1}^1)$$

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:

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




```
# you can paste code of encoder from modules.py
# the encoder can be like seminar encoder but you have to return outputs
# and if you use bidirectional you won't make the same operation like with hidden
# because outputs = [src sent len, batch size, hid dim * n directions]

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, b

        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 = [n_layers * 2, batch_size, hid_dim]
          hidden = hidden.reshape(self.n_layers, 2, -1, self.hid_dim)
          # hidden = [n_layers, 2, batch_size, hid_dim]
          hidden = hidden.transpose(1, 2).reshape(self.n_layers, -1, 2 * self.hid_di
          #hidden = [n_layers, batch_size, 2 * hid_dim]


      # outputs = [src_sent_len, batch_size, hid_dim * (1 + bidir)]
      # in my option bidir = 1 ---> outputs = [src_sent_len, batch_size, 2 * hi_dim
      # hidden = [n_layers, batch_size, n_directions * hid_dim]

      return outputs, hidden, cell
```

Attention

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

```
# you can paste code of attention from modules.py
```

```
def softmax(x, temperature=10): # use your temperature
    e_x = torch.exp(x / temperature)
    return e_x / torch.sum(e_x, dim=0)

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

        self.enc_hid_dim = enc_hid_dim
        self.dec_hid_dim = dec_hid_dim

        self.attn = nn.Linear(2*enc_hid_dim + dec_hid_dim, enc_hid_dim)
        self.v = nn.Linear(enc_hid_dim, 1)
        self.temp = temp

    def forward(self, hidden, encoder_outputs):

        # encoder_outputs = [src sent len, batch size, enc_hid_dim]
        # hidden = [1, batch size, dec_hid_dim]

        # repeat hidden and concatenate it with encoder_outputs
        hidden_repeat = hidden.repeat(encoder_outputs.shape[0], 1, 1)
        encode_repeat = torch.cat((encoder_outputs, hidden_repeat), dim = -1)
        # encode_repeat = [src_sent_len, batch_size, enc_hid_dim + dec_hid_dim]
        # calculate energy
        E = torch.tanh(self.attn(encode_repeat))
        # E = [src_sent_len, batch_size, enc_hid_dim]
        E = self.v(E)
        # E = [src_sent_len, batch-size , 1]
        # get attention, use softmax function which is defined, can change tempera
        attention = softmax(E, self.temp)

        return attention.permute(1, 2, 0) # [batch_size, 1, src_sent_len]
```

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.

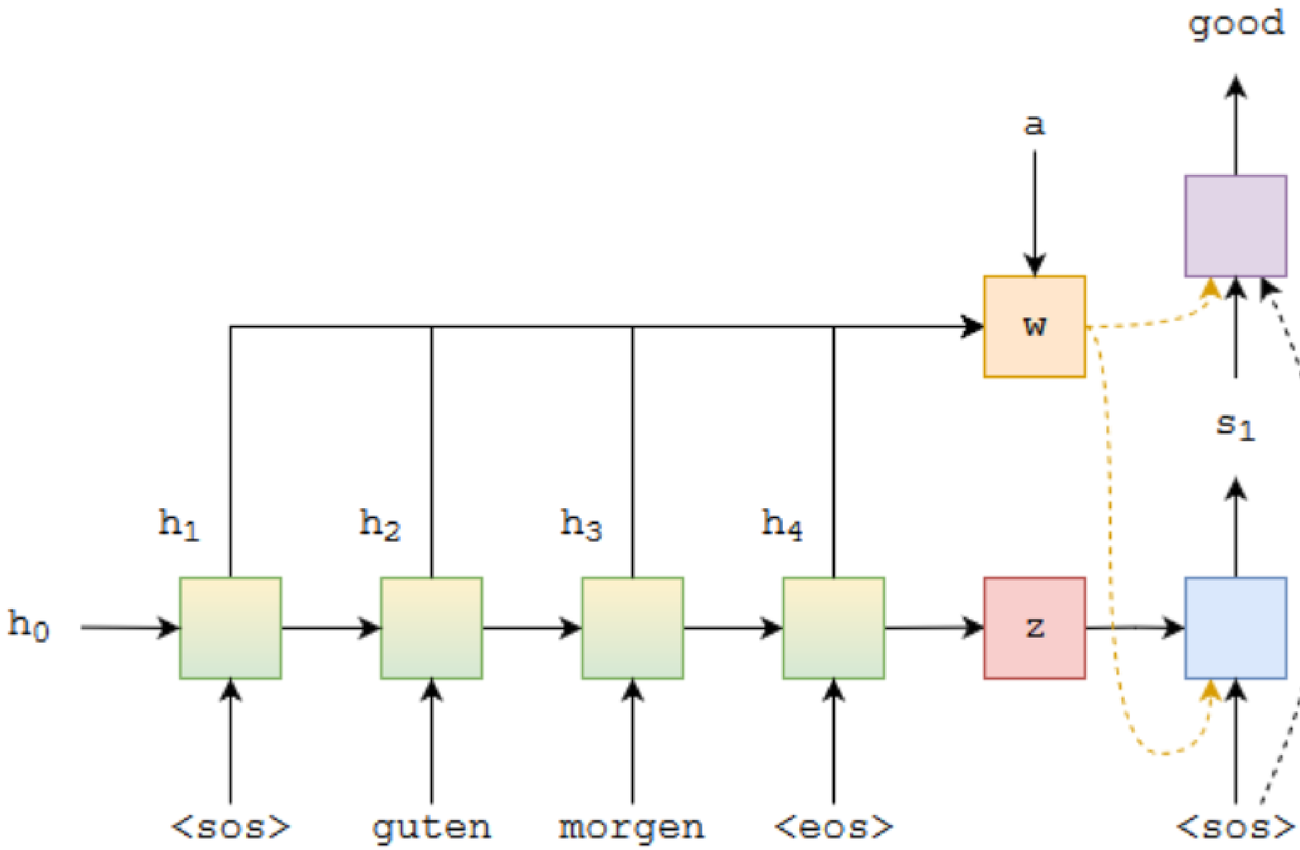
$$s_t = \text{DecoderGRU}([y_t, w_t], s_{t-1})$$

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.

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

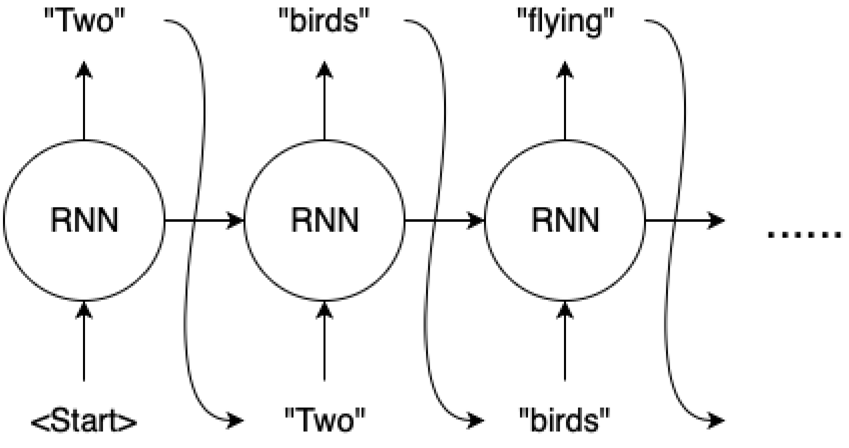
The image below shows decoding the **first** word in an example translation.

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.

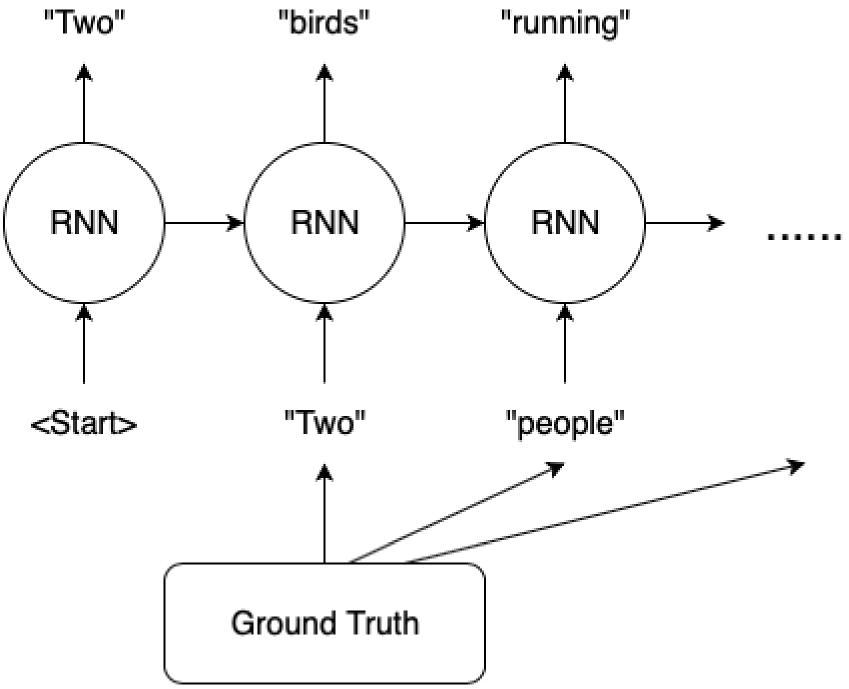


Teacher forcing

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.

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

```
        self.enc_hid_dim = enc_hid_dim
        if (bidir):
          self.dec_hid_dim = 2 * dec_hid_dim
          self.emb_dim = 2 * emb_dim
        else :
          self.dec_hid_dim = dec_hid_dim
          self.emb_dim = emb_dim
        self.output_dim = output_dim
        self.attention = attention

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

        self.rnn = nn.GRU(self.emb_dim + 2 * enc_hid_dim, self.dec_hid_dim) # use GR

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

        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, encoder_outputs):
        #input = [batch size]
        #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 = input.unsqueeze(0) # because only one word, no words sequence

        #input = [1, batch size]

        embedded = self.dropout(self.embedding(input)).squeeze(0)

        #embedded = [batch size, emb dim]

        # get weighted sum of encoder_outputs
        weights = self.attention(hidden, encoder_outputs) # [batch_size, 1, sent_len
        encoder_out = encoder_outputs.permute(1, 0, 2) # [batch_size, sent_len, enc_
        weighted_sum = torch.bmm(weights, encoder_out).squeeze(1) # [batch_size, enc_

        # concatenate weighted sum and embedded, break through the GRU
        weight_emb = torch.cat((weighted_sum, embedded), dim = -1).unsqueeze(0)
        output, hidden = self.rnn(weight_emb)
        # hidden = [1, batch_size, dec_hid]
        # get predictions
        all = torch.cat((embedded, weighted_sum, hidden.squeeze(0)), dim = -1)
        prediction = self.out(all.unsqueeze(0)).squeeze(0)

        #prediction = [batch size, output dim]

        return prediction, 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:

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

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

        self.encoder = encoder.to(device)
        self.decoder = decoder.to(device)
        self.device = device

        assert 2 * encoder.hid_dim == 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 th

       # 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
       enc_states, hidden, cell = self.encoder(src)

       #first input to the decoder is the <sos> tokens
       input = trg[0,:]

       for t in range(1, trg_len):

         output, hidden = self.decoder(input, hidden, enc_states)
```

```
            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 = trg[t] if teacher_force else top1

        return outputs
```

Training

```
INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
ENC_EMB_DIM = 350
DEC_EMB_DIM = 350
HID_DIM = 350
N_LAYERS = 1 # simple model: n_layers=1
ENC_DROPOUT = 0.4
DEC_DROPOUT = 0.4
BIDIRECTIONAL = True

enc = Encoder(INPUT_DIM, DEC_EMB_DIM, HID_DIM, N_LAYERS, ENC_DROPOUT, BIDIRECTIONA
attention = Attention(HID_DIM, 2 * HID_DIM, 5)
dec = DecoderWithAttention(OUTPUT_DIM, ENC_EMB_DIM, HID_DIM, HID_DIM,
                           DEC_DROPOUT, attention, BIDIRECTIONAL)

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


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

model.apply(init_weights)


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')


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_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

torch.save(model.state_dict(), 'best-val-model-end.pt')


train_history = []
valid_history = []

N_EPOCHS = 12
CLIP = 5

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}
```

```
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
```

Let's take a look at our network quality:

```
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[
    generated = remove_tech_tokens(cut_on_eos([TRG_vocab.itos[x] for x in list(out

    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 li
     return generated
```

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

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

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

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

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

Bleu

[link]

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

Then,

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

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

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

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

```
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[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

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

```
59it [00:11,  4.99it/s]
```

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

```
27.002692127922966
```

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

[ ] ↳ *Скрыта 1 ячейка.*