In [1]:
```
# set tf 1.x for colab
%tensorflow_version 1.x
```

```
UsageError: Line magic function `%tensorflow_version` not found.
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore', category=DeprecationWarning)
warnings.filterwarnings('ignore', category=FutureWarning)
```

Learn to calculate with seq2seq model

In this assignment, you will learn how to use neural networks to solve sequence-to-sequence prediction tasks. Seq2Seq models are very popular these days because they achieve great results in Machine Translation, Text Summarization, Conversational Modeling and more.

Using sequence-to-sequence modeling you are going to build a calculator for evaluating arithmetic expressions, by taking an equation as an input to the neural network and producing an answer as it's output.

The resulting solution for this problem will be based on state-of-the-art approaches for sequence-to-sequence learning and you should be able to easily adapt it to solve other tasks. However, if you want to train your own machine translation system or intellectual chat bot, it would be useful to have access to compute resources like GPU, and be patient, because training of such systems is usually time consuming.

Libraries

For this task you will need the following libraries:

- TensorFlow (https://www.tensorflow.org) — an open-source software library for Machine Intelligence.
- scikit-learn (http://scikit-learn.org/stable/index.html) — a tool for data mining and data analysis.

If you have never worked with TensorFlow, you will probably want to read some tutorials during your work on this assignment, e.g. Neural Machine Translation (https://www.tensorflow.org/tutorials/seq2seq) tutorial deals with very similar task and can explain some concepts to you.

Data

One benefit of this task is that you don't need to download any data — you will generate it on your own! We will use two operators (addition and subtraction) and work with positive integer numbers in some range. Here are examples of correct inputs and outputs:

```
Input: '1+2'
Output: '3'

Input: '0-99'
Output: '-99'
```

Note, that there are no spaces between operators and operands.

Now you need to implement the function *generate_equations*, which will be used to generate the data.

In [3]:
```
import random
```

In [4]:
```
def generate_equations(allowed_operators, dataset_size, min_value, max_value):
    """Generates pairs of equations and solutions to them.

       Each equation has a form of two integers with an operator in between.
       Each solution is an integer with the result of the operaion.

        allowed_operators: list of strings, allowed operators.
        dataset_size: an integer, number of equations to be generated.
        min_value: an integer, min value of each operand.
        max_value: an integer, max value of each operand.

        result: a list of tuples of strings (equation, solution).
    """
    sample = []
    for _ in range(dataset_size):
        ########################################
        ######### YOUR CODE HERE #############
        operand1 = random.randint(min_value, max_value)
        operand2 = random.randint(min_value, max_value)
        operator = random.choice(allowed_operators)

        equation = str(operand1) + operator + str(operand2)
        solution = str(eval(equation))
        sample.append((equation, solution))
        ########################################
    return sample
```

To check the correctness of your implementation, use *test_generate_equations* function:

In [5]:
```
def test_generate_equations():
    allowed_operators = ['+', '-']
    dataset_size = 10
    for (input_, output_) in generate_equations(allowed_operators, dataset_size, 0, 100):
        if not (type(input_) is str and type(output_) is str):
            return "Both parts should be strings."
        if eval(input_) != int(output_):
            return "The (equation: {!r}, solution: {!r}) pair is incorrect.".format(input_, output_)
    return "Tests passed."
```

In [6]:
```
print(test_generate_equations())
```

```
Tests passed.
```

Finally, we are ready to generate the train and test data for the neural network:

In [7]:
```
from sklearn.model_selection import train_test_split
```

In [8]:
```
allowed_operators = ['+', '-']
dataset_size = 100000
data = generate_equations(allowed_operators, dataset_size, min_value=0, max_value=9999)

train_set, test_set = train_test_split(data, test_size=0.2, random_state=42)
```

Prepare data for the neural network

The next stage of data preparation is creating mappings of the characters to their indices in some vocabulary. Since in our task we already know which symbols will appear in the inputs and outputs, generating the vocabulary is a simple step.

How to create dictionaries for other task

First of all, you need to understand what is the basic unit of the sequence in your task. In our case, we operate on symbols and the basic unit is a symbol. The number of symbols is small, so we don't need to think about filtering/normalization steps. However, in other tasks, the basic unit is often a word, and in this case the mapping would be *word* → *integer*. The number of words might be huge, so it would be reasonable to filter them, for example, by frequency and leave only the frequent ones. Other strategies that your should consider are: data normalization (lowercasing, tokenization, how to consider punctuation marks), separate vocabulary for input and for output (e.g. for machine translation), some specifics of the task.

In [9]:
```
word2id = {symbol:i for i, symbol in enumerate('#^$+-1234567890')}
id2word = {i:symbol for symbol, i in word2id.items()}
```

Special symbols

In [10]:
```
start_symbol = '^'
end_symbol = '$'
padding_symbol = '#'
```

You could notice that we have added 3 special symbols: '^', '$' and '#':

- '^' symbol will be passed to the network to indicate the beginning of the decoding procedure. We will discuss this one later in more details.
- '$' symbol will be used to indicate the *end of a string*, both for input and output sequences.
- '#' symbol will be used as a *padding* character to make lengths of all strings equal within one training batch.

People have a bit different habits when it comes to special symbols in encoder-decoder networks, so don't get too much confused if you come across other variants in tutorials you read.

Padding

When vocabularies are ready, we need to be able to convert a sentence to a list of vocabulary word indices and back. At the same time, let's care about padding. We are going to preprocess each sequence from the input (and output ground truth) in such a way that:

- it has a predefined length *padded_len*
- it is probably cut off or padded with the *padding symbol* '#'
- it *always* ends with the *end symbol* '$'

We will treat the original characters of the sequence **and the end symbol** as the valid part of the input. We will store *the actual length* of the sequence, which includes the end symbol, but does not include the padding symbols.

Now you need to implement the function *sentence_to_ids* that does the described job.

In [11]:
```
def sentence_to_ids(sentence, word2id, padded_len):
    """ Converts a sequence of symbols to a padded sequence of their ids.

      sentence: a string, input/output sequence of symbols.
      word2id: a dict, a mapping from original symbols to ids.
      padded_len: an integer, a desirable length of the sequence.

      result: a tuple of (a list of ids, an actual length of sentence).
    """

    ######### YOUR CODE HERE #############
    if len(sentence) + 1 <= padded_len:
        ended_sentence = sentence + '$'
    else:
        ended_sentence = sentence[:padded_len-1] + '$'
    padded_sentence = ended_sentence + '#' * (padded_len - len(ended_sentence))

    sent_ids = []
    for i in padded_sentence:
        sent_ids.append(word2id.get(i))

    ######### YOUR CODE HERE #############
    sent_len = len(ended_sentence)

    return sent_ids, sent_len
```

Check that your implementation is correct:

In [12]:
```
def test_sentence_to_ids():
    sentences = [("123+123", 7), ("123+123", 8), ("123+123", 10)]
    expected_output = [([5, 6, 7, 3, 5, 6, 2], 7),
                       ([5, 6, 7, 3, 5, 6, 7, 2], 8),
                       ([5, 6, 7, 3, 5, 6, 7, 2, 0, 0], 8)]
    for (sentence, padded_len), (sentence_ids, expected_length) in zip(sentences, expected_output):
        output, length = sentence_to_ids(sentence, word2id, padded_len)
        if output != sentence_ids:
            return("Convertion of '{}' for padded_len={} to {} is incorrect.".format(
                sentence, padded_len, output))
        if length != expected_length:
            return("Convertion of '{}' for padded_len={} has incorrect actual length {}.".format(
                sentence, padded_len, length))
    return("Tests passed.")
```

In [13]:
```
print(test_sentence_to_ids())
```

```
Tests passed.
```

We also need to be able to get back from indices to symbols:

In [14]:
```
def ids_to_sentence(ids, id2word):
    """ Converts a sequence of ids to a sequence of symbols.

          ids: a list, indices for the padded sequence.
          id2word:  a dict, a mapping from ids to original symbols.

          result: a list of symbols.
    """

    return [id2word[i] for i in ids]
```

Generating batches

The final step of data preparation is a function that transforms a batch of sentences to a list of lists of indices.

In [15]:
```
def batch_to_ids(sentences, word2id, max_len):
    """Prepares batches of indices.

       Sequences are padded to match the longest sequence in the batch,
       if it's longer than max_len, then max_len is used instead.

        sentences: a list of strings, original sequences.
        word2id: a dict, a mapping from original symbols to ids.
        max_len: an integer, max len of sequences allowed.

        result: a list of lists of ids, a list of actual lengths.
    """

    max_len_in_batch = min(max(len(s) for s in sentences) + 1, max_len)
    batch_ids, batch_ids_len = [], []
    for sentence in sentences:
        ids, ids_len = sentence_to_ids(sentence, word2id, max_len_in_batch)
        batch_ids.append(ids)
        batch_ids_len.append(ids_len)
    return batch_ids, batch_ids_len
```

The function *generate_batches* will help to generate batches with defined size from given samples.

In [16]:
```
def generate_batches(samples, batch_size=64):
    X, Y = [], []
    for i, (x, y) in enumerate(samples, 1):
        X.append(x)
        Y.append(y)
        if i % batch_size == 0:
            yield X, Y
            X, Y = [], []
    if X and Y:
        yield X, Y
```

To illustrate the result of the implemented functions, run the following cell:

In [17]:
```
sentences = train_set[0]
ids, sent_lens = batch_to_ids(sentences, word2id, max_len=10)
print('Input:', sentences)
print('Ids: {}\nSentences lengths: {}'.format(ids, sent_lens))
```

```
Input: ('9736+4167', '13903')
Ids: [[13, 11, 7, 10, 3, 8, 5, 10, 11, 2], [5, 7, 13, 14, 7, 2, 0, 0, 0, 0]]
Sentences lengths: [10, 6]
```

Encoder-Decoder architecture

Encoder-Decoder is a successful architecture for Seq2Seq tasks with different lengths of input and output sequences. The main idea is to use two recurrent neural networks, where the first neural network *encodes* the input sequence into a real-valued vector and then the second neural network *decodes* this vector into the output sequence. While building the neural network, we will specify some particular characteristics of this architecture.

In [18]:
```
import tensorflow as tf
```

Let us use TensorFlow building blocks to specify the network architecture.

In [19]:
```
class Seq2SeqModel(object):
    pass
```

First, we need to create placeholders (https://www.tensorflow.org/api_guides/python/io_ops#Placeholders) to specify what data we are going to feed into the network during the execution time. For this task we will need:

- *input_batch* — sequences of sentences (the shape will equal to [batch_size, max_sequence_len_in_batch]);
- *input_batch_lengths* — lengths of not padded sequences (the shape equals to [batch_size]);
- *ground_truth* — sequences of groundtruth (the shape will equal to [batch_size, max_sequence_len_in_batch]);
- *ground_truth_lengths* — lengths of not padded groundtruth sequences (the shape equals to [batch_size]);
- *dropout_ph* — dropout keep probability; this placeholder has a predifined value 1;
- *learning_rate_ph* — learning rate.

In [20]:
```python
def declare_placeholders(self):
    """Specifies placeholders for the model."""

    # Placeholders for input and its actual lengths.
    self.input_batch = tf.placeholder(shape=(None, None), dtype=tf.int32, name='input_batch')
    self.input_batch_lengths = tf.placeholder(shape=(None, ), dtype=tf.int32, name='input_batch_lengths')

    # Placeholders for groundtruth and its actual lengths.
    ######### YOUR CODE HERE #############
    self.ground_truth = tf.placeholder(shape=(None, None), dtype=tf.int32, name='ground_truth')
    ######### YOUR CODE HERE #############
    self.ground_truth_lengths = tf.placeholder(shape=(None, ), dtype=tf.int32, name='ground_truth_lengths')

    self.dropout_ph = tf.placeholder_with_default(tf.cast(1.0, tf.float32), shape=[])
    ######### YOUR CODE HERE #############
    self.learning_rate_ph = tf.placeholder(dtype=tf.float32, shape=[], name='learning_rate_ph')
```

In [21]:
```python
Seq2SeqModel.__declare_placeholders = classmethod(declare_placeholders)
```

Now, let us specify the layers of the neural network. First, we need to prepare an embedding matrix. Since we use the same vocabulary for input and output, we need only one such matrix. For tasks with different vocabularies there would be multiple embedding layers.

- Create embeddings matrix with tf.Variable (https://www.tensorflow.org/api_docs/python/tf/Variable). Specify its name, type (tf.float32), and initialize with random values.
- Perform embeddings lookup (https://www.tensorflow.org/api_docs/python/tf/nn/embedding_lookup) for a given input batch.

In [22]:
```python
def create_embeddings(self, vocab_size, embeddings_size):
    """Specifies embeddings layer and embeds an input batch."""

    random_initializer = tf.random_uniform((vocab_size, embeddings_size), -1.0, 1.0)
    ######### YOUR CODE HERE #############
    self.embeddings = tf.Variable(initial_value=random_initializer, dtype=tf.float32, name='embeddings')

    # Perform embeddings lookup for self.input_batch.
    ######### YOUR CODE HERE #############
    self.input_batch_embedded = tf.nn.embedding_lookup(self.embeddings, self.input_batch)
```

In [23]:
```python
Seq2SeqModel.__create_embeddings = classmethod(create_embeddings)
```

Encoder

The first RNN of the current architecture is called an *encoder* and serves for encoding an input sequence to a real-valued vector. Input of this RNN is an embedded input batch. Since sentences in the same batch could have different actual lengths, we also provide input lengths to avoid unnecessary computations. The final encoder state will be passed to the second RNN (decoder), which we will create soon.

- TensorFlow provides a number of RNN cells (https://www.tensorflow.org/api_guides/python/contrib.rnn#Core_RNN_Cells_for_use_with_TensorFlow_s_core_RNN_methods) ready for use. We suggest that you use GRU cell (https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/GRUCell), but you can also experiment with other types.
- Wrap your cells with DropoutWrapper (https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/DropoutWrapper). Dropout is an important regularization technique for neural networks. Specify input keep probability using the dropout placeholder that we created before.
- Combine the defined encoder cells with Dynamic RNN (https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn). Use the embedded input batches and their lengths here.
- Use *dtype=tf.float32* everywhere.

In [24]:
```python
def build_encoder(self, hidden_size):
    """Specifies encoder architecture and computes its output."""

    # Create GRUCell with dropout.
    ######### YOUR CODE HERE #############
    encoder_cell = tf.nn.rnn_cell.GRUCell(num_units=hidden_size)
    encoder_cell = tf.nn.rnn_cell.DropoutWrapper(encoder_cell, input_keep_prob=self.dropout_ph)

    # Create RNN with the predefined cell.
    ######### YOUR CODE HERE #############
    _, self.final_encoder_state = tf.nn.dynamic_rnn(cell=encoder_cell,
                                                    inputs=self.input_batch_embedded,
                                                    sequence_length=self.input_batch_lengths,
                                                    dtype=tf.float32)
```

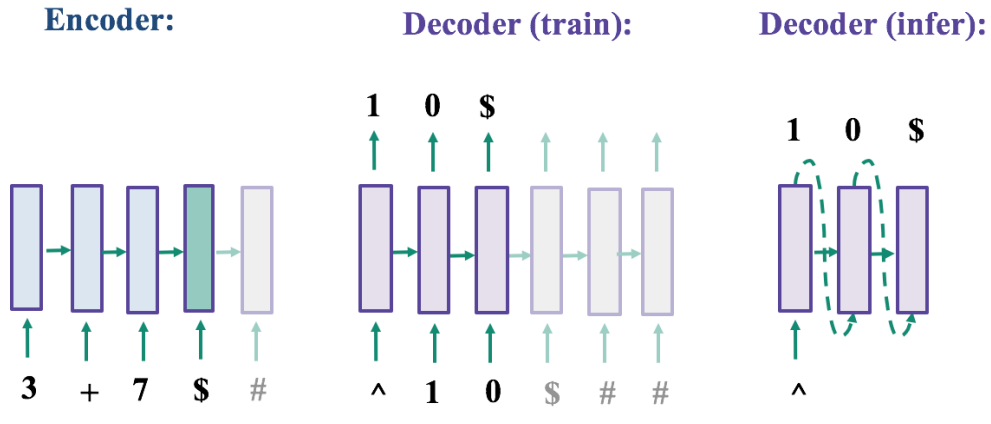
In [25]:
```python
Seq2SeqModel.__build_encoder = classmethod(build_encoder)
```

Decoder

The second RNN is called a *decoder* and serves for generating the output sequence. In the simple seq2seq arcitecture, the input sequence is provided to the decoder only as the final state of the encoder. Obviously, it is a bottleneck and Attention techniques (https://www.tensorflow.org/tutorials/seq2seq#background_on_the_attention_mechanism) can help to overcome it. So far, we do not need them to make our calculator work, but this would be a necessary ingredient for more advanced tasks.

During training, decoder also uses information about the true output. It is feeded in as input symbol by symbol. However, during the prediction stage (which is called *inference* in this architecture), the decoder can only use its own generated output from the previous step to feed it in at the next step. Because of this difference (*training* vs *inference*), we will create two distinct instances, which will serve for the described scenarios.

The picture below illustrates the point. It also shows our work with the special characters, e.g. look how the start symbol ^ is used. The transparent parts are ignored. In decoder, it is masked out in the loss computation. In encoder, the green state is considered as final and passed to the decoder.



Now, it's time to implement the decoder:

- First, we should create two helpers (https://www.tensorflow.org/api_guides/python/contrib.seq2seq#Dynamic_Decoding). These classes help to determine the behaviour of the decoder. During the training time, we will use TrainingHelper (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/TrainingHelper). For the inference we recommend to use GreedyEmbeddingHelper (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/GreedyEmbeddingHelper).
- To share all parameters during training and inference, we use one scope and set the flag 'reuse' to True at inference time. You might be interested to know more about how variable scopes (https://www.tensorflow.org/programmers_guide/variables) work in TF.
- To create the decoder itself, we will use BasicDecoder (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/BasicDecoder) class. As previously, you should choose some RNN cell, e.g. GRU cell. To turn hidden states into logits, we will need a projection layer. One of the simple solutions is using OutputProjectionWrapper (https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/OutputProjectionWrapper).
- For getting the predictions, it will be convinient to use dynamic_decode (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/dynamic_decode). This function uses the provided decoder to perform decoding.

In [26]:
```
def build_decoder(self, hidden_size, vocab_size, max_iter, start_symbol_id, end_symbol_id):
    """Specifies decoder architecture and computes the output.

        Uses different helpers:
          - for train: feeding ground truth
          - for inference: feeding generated output

        As a result, self.train_outputs and self.infer_outputs are created.
        Each of them contains two fields:
          rnn_output (predicted logits)
          sample_id (predictions).

    """

    # Use start symbols as the decoder inputs at the first time step.
    batch_size = tf.shape(self.input_batch)[0]
    start_tokens = tf.fill([batch_size], start_symbol_id)
    ground_truth_as_input = tf.concat([tf.expand_dims(start_tokens, 1), self.ground_truth], 1)

    # Use the embedding layer defined before to lookup embedings for ground_truth_as_input.
    ######### YOUR CODE HERE #############
    self.ground_truth_embedded = tf.nn.embedding_lookup(self.embeddings, ground_truth_as_input)

    # Create TrainingHelper for the train stage.
    train_helper = tf.contrib.seq2seq.TrainingHelper(self.ground_truth_embedded,
                                                     self.ground_truth_lengths)

    # Create GreedyEmbeddingHelper for the inference stage.
    # You should provide the embedding layer, start_tokens and index of the end symbol.
    ######### YOUR CODE HERE #############
    infer_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(self.embeddings,
                                                            start_tokens,
                                                            end_symbol_id)


    def decode(helper, scope, reuse=None):
        """Creates decoder and return the results of the decoding with a given helper."""

        with tf.variable_scope(scope, reuse=reuse):
            # Create GRUCell with dropout. Do not forget to set the reuse flag properly.
            ######### YOUR CODE HERE #############

            decoder_cell = tf.nn.rnn_cell.GRUCell(num_units=hidden_size, reuse=reuse)
            decoder_cell = tf.nn.rnn_cell.DropoutWrapper(decoder_cell, input_keep_prob=self.dropout_ph)

            # Create a projection wrapper.
            decoder_cell = tf.contrib.rnn.OutputProjectionWrapper(decoder_cell, vocab_size, reuse=reuse)

            # Create BasicDecoder, pass the defined cell, a helper, and initial state.
            # The initial state should be equal to the final state of the encoder!
            ######### YOUR CODE HERE #############
            decoder = tf.contrib.seq2seq.BasicDecoder(cell=decoder_cell,
                                                      helper=helper,
                                                      initial_state=self.final_encoder_state)

            # The first returning argument of dynamic_decode contains two fields:
            #   rnn_output (predicted logits)
            #   sample_id (predictions)
            outputs, _, _ = tf.contrib.seq2seq.dynamic_decode(decoder=decoder, maximum_iterations=max_iter,
                                                              output_time_major=False, impute_finished=True)

            return outputs

    self.train_outputs = decode(train_helper, 'decode')
    self.infer_outputs = decode(infer_helper, 'decode', reuse=True)
```

In [27]:
```
Seq2SeqModel.__build_decoder = classmethod(build_decoder)
```

In this task we will use sequence_loss (https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq/sequence_loss), which is a weighted cross-entropy loss for a sequence of logits. Take a moment to understand, what is your train logits and targets. Also note, that we do not want to take into account loss terms coming from padding symbols, so we will mask them out using weights.

In [28]:
```
def compute_loss(self):
    """Computes sequence loss (masked cross-entopy loss with logits)."""

    weights = tf.cast(tf.sequence_mask(self.ground_truth_lengths), dtype=tf.float32)

    ######### YOUR CODE HERE #############
    self.loss = tf.contrib.seq2seq.sequence_loss(logits=self.train_outputs.rnn_output,
                                                 targets=self.ground_truth,
                                                 weights=weights)
```

In [29]:
```
Seq2SeqModel.__compute_loss = classmethod(compute_loss)
```

The last thing to specify is the optimization of the defined loss. We suggest that you use optimize_loss (https://www.tensorflow.org/api_docs/python/tf/contrib/layers/optimize_loss) with Adam optimizer and a learning rate from the corresponding placeholder. You might also need to pass global step (e.g. as tf.train.get_global_step()) and clip gradients by 1.0.

In [30]:
```
def perform_optimization(self):
    """Specifies train_op that optimizes self.loss."""

    ######### YOUR CODE HERE #############
    self.train_op = tf.contrib.layers.optimize_loss(loss=self.loss,
                                                    global_step=tf.train.get_global_step(),
                                                    optimizer='Adam',
                                                    learning_rate=self.learning_rate_ph,
                                                    clip_gradients=1.0)
```

In [31]:
```
Seq2SeqModel.__perform_optimization = classmethod(perform_optimization)
```

Congratulations! You have specified all the parts of your network. You may have noticed, that we didn't deal with any real data yet, so what you have written is just recipies on how the network should function. Now we will put them to the constructor of our Seq2SeqModel class to use it in the next section.

In [32]:
```
def init_model(self, vocab_size, embeddings_size, hidden_size,
               max_iter, start_symbol_id, end_symbol_id, padding_symbol_id):

    self.__declare_placeholders()
    self.__create_embeddings(vocab_size, embeddings_size)
    self.__build_encoder(hidden_size)
    self.__build_decoder(hidden_size, vocab_size, max_iter, start_symbol_id, end_symbol_id)

    # Compute loss and back-propagate.
    self.__compute_loss()
    self.__perform_optimization()

    # Get predictions for evaluation.
    self.train_predictions = self.train_outputs.sample_id
    self.infer_predictions = self.infer_outputs.sample_id
```

In [33]:
```
Seq2SeqModel.__init__ = classmethod(init_model)
```

Train the network and predict output

Session.run (https://www.tensorflow.org/api_docs/python/tf/Session#run) is a point which initiates computations in the graph that we have defined. To train the network, we need to compute *self.train_op*. To predict output, we just need to compute *self.infer_predictions*. In any case, we need to feed actual data through the placeholders that we defined above.

In [34]:
```
def train_on_batch(self, session, X, X_seq_len, Y, Y_seq_len, learning_rate, dropout_keep_probability):
    feed_dict = {
            self.input_batch: X,
            self.input_batch_lengths: X_seq_len,
            self.ground_truth: Y,
            self.ground_truth_lengths: Y_seq_len,
            self.learning_rate_ph: learning_rate,
            self.dropout_ph: dropout_keep_probability
        }
    pred, loss, _ = session.run([
            self.train_predictions,
            self.loss,
            self.train_op], feed_dict=feed_dict)
    return pred, loss
```

In [35]:
```
Seq2SeqModel.train_on_batch = classmethod(train_on_batch)
```

We implemented two prediction functions: *predict_for_batch* and *predict_for_batch_with_loss*. The first one allows only to predict output for some input sequence, while the second one could compute loss because we provide also ground truth values. Both these functions might be useful since the first one could be used for predicting only, and the second one is helpful for validating results on not-training data during the training.

In [36]:
```
def predict_for_batch(self, session, X, X_seq_len):
    ######### YOUR CODE HERE #############
    feed_dict = {
            self.input_batch: X,
            self.input_batch_lengths: X_seq_len
    }

    pred = session.run([
            self.infer_predictions
        ], feed_dict=feed_dict)[0]
    return pred

def predict_for_batch_with_loss(self, session, X, X_seq_len, Y, Y_seq_len):
    ######### YOUR CODE HERE #############
    feed_dict = {
            self.input_batch: X,
            self.input_batch_lengths: X_seq_len,
            self.ground_truth: Y,
            self.ground_truth_lengths: Y_seq_len
    }

    pred, loss = session.run([
            self.infer_predictions,
            self.loss,
        ], feed_dict=feed_dict)
    return pred, loss
```

In [37]:
```
Seq2SeqModel.predict_for_batch = classmethod(predict_for_batch)
Seq2SeqModel.predict_for_batch_with_loss = classmethod(predict_for_batch_with_loss)
```

Run your experiment

Create *Seq2SeqModel* model with the following parameters:

- *vocab_size* — number of tokens;
- *embeddings_size* — dimension of embeddings, recommended value: 20;
- *max_iter* — maximum number of steps in decoder, recommended value: 7;
- *hidden_size* — size of hidden layers for RNN, recommended value: 512;
- *start_symbol_id* — an index of the start token (^).
- *end_symbol_id* — an index of the end token ($).
- *padding_symbol_id* — an index of the padding token (#).

Set hyperparameters. You might want to start with the following values and see how it works:

- *batch_size*: 128;
- at least 10 epochs;
- value of *learning_rate*: 0.001
- *dropout_keep_probability* equals to 0.5 for training (typical values for dropout probability are ranging from 0.1 to 1.0); larger values correspond smaler number of dropout units;
- *max_len*: 20.

In [38]:
```
tf.reset_default_graph()

######### YOUR CODE HERE #############
model = Seq2SeqModel(vocab_size=len(word2id),
                     embeddings_size=20,
                     max_iter=7,
                     hidden_size=512,
                     start_symbol_id=word2id['^'],
                     end_symbol_id=word2id['$'],
                     padding_symbol_id=word2id['#']
                    )

######### YOUR CODE HERE #############
batch_size = 128

######### YOUR CODE HERE #############
n_epochs = 10

######### YOUR CODE HERE #############
learning_rate = 0.001

######### YOUR CODE HERE #############
dropout_keep_probability = 0.5

######### YOUR CODE HERE #############
max_len = 20

n_step = int(len(train_set) / batch_size)
```

```
WARNING:tensorflow:From <ipython-input-24-314cb3255491>:6: GRUCell.__init__ (from tensorflow.python.ops.rnn_cell_impl) is deprecated and will be removed in a future version.
Instructions for updating:
This class is equivalent as tf.keras.layers.GRUCell, and will be replaced by that in Tensorflow 2.0.
WARNING:tensorflow:From <ipython-input-24-314cb3255491>:14: dynamic_rnn (from tensorflow.python.ops.rnn) is deprecated and will be removed in a future version.
Instructions for updating:
Please use `keras.layers.RNN(cell)`, which is equivalent to this API
WARNING:tensorflow:From C:\Users\Xiaowei\Anaconda3\envs\tfspark\lib\site-packages\tensorflow\python\ops\init_ops.py:1251: calling VarianceScaling.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From C:\Users\Xiaowei\Anaconda3\envs\tfspark\lib\site-packages\tensorflow\python\ops\rnn_cell_impl.py:564: calling Constant.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From C:\Users\Xiaowei\Anaconda3\envs\tfspark\lib\site-packages\tensorflow\python\ops\rnn_cell_impl.py:574: calling Zeros.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:Entity <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A1986748>> could not be transformed and will be executed as-is. Please report this to the AutgoGraph team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output. Cause: converting <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A1986748>>: AssertionError: Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A1986748>> could not be transformed and will be executed as-is. Please report this to the AutgoGraph team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output. Cause: converting <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A1986748>>: AssertionError: Bad argument number for Name: 3, expecting 4
WARNING:tensorflow:From C:\Users\Xiaowei\Anaconda3\envs\tfspark\lib\site-packages\tensorflow\python\ops\rnn.py:244: add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.array_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
WARNING:tensorflow:
The TensorFlow contrib module will not be included in TensorFlow 2.0.
For more information, please see:
  * https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-sunset.md (https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-sunset.md)
  * https://github.com/tensorflow/addons (https://github.com/tensorflow/addons)
  * https://github.com/tensorflow/io (https://github.com/tensorflow/io) (for I/O related ops)
If you depend on functionality not listed there, please file an issue.

WARNING:tensorflow:Entity <bound method OutputProjectionWrapper.call of <tensorflow.contrib.rnn.python.ops.core_rnn_cell.OutputProjectionWrapper object at 0x000001F6A3E2FDC8>> could not be transformed and will be executed as-is. Please report this to the AutgoGraph team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output. Cause: converting <bound method OutputProjectionWrapper.call of <tensorflow.contrib.rnn.python.ops.core_rnn_cell.OutputProjectionWrapper object at 0x000001F6A3E2FDC8>>: AssertionError: Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method OutputProjectionWrapper.call of <tensorflow.contrib.rnn.python.ops.core_rnn_cell.OutputProjectionWrapper object at 0x000001F6A3E2FDC8>> could not be transformed and will be executed as-is. Please report this to the AutgoGraph team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output. Cause: converting <bound method OutputProjectionWrapper.call of <tensorflow.contrib.rnn.python.ops.core_rnn_cell.OutputProjectionWrapper object at 0x000001F6A3E2FDC8>>: AssertionError: Bad argument number for Name:
```

```
 3, expecting 4
WARNING:tensorflow:Entity <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001
F6A3DBE808>> could not be transformed and will be executed as-is. Please report this to the AutgoGraph team. When filin
g the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output. Cause: conve
rting <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A3DBE808>>: Assert
ionError: Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A3DBE808
>> could not be transformed and will be executed as-is. Please report this to the AutgoGraph team. When filing the bug,
set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output. Cause: converting <boun
d method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A3DBE808>>: AssertionError: B
ad argument number for Name: 3, expecting 4
WARNING:tensorflow:Entity <bound method OutputProjectionWrapper.call of <tensorflow.contrib.rnn.python.ops.core_rnn_cel
l.OutputProjectionWrapper object at 0x000001F6A3F9F188>> could not be transformed and will be executed as-is. Please re
port this to the AutgoGraph team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=1
0`) and attach the full output. Cause: converting <bound method OutputProjectionWrapper.call of <tensorflow.contrib.rn
n.python.ops.core_rnn_cell.OutputProjectionWrapper object at 0x000001F6A3F9F188>>: AssertionError: Bad argument number
 for Name: 3, expecting 4
WARNING: Entity <bound method OutputProjectionWrapper.call of <tensorflow.contrib.rnn.python.ops.core_rnn_cell.OutputPr
ojectionWrapper object at 0x000001F6A3F9F188>> could not be transformed and will be executed as-is. Please report this
 to the AutgoGraph team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and a
ttach the full output. Cause: converting <bound method OutputProjectionWrapper.call of <tensorflow.contrib.rnn.python.o
ps.core_rnn_cell.OutputProjectionWrapper object at 0x000001F6A3F9F188>>: AssertionError: Bad argument number for Name:
 3, expecting 4
WARNING:tensorflow:Entity <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001
F6A34D9CC8>> could not be transformed and will be executed as-is. Please report this to the AutgoGraph team. When filin
g the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output. Cause: conve
rting <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A34D9CC8>>: Assert
ionError: Bad argument number for Name: 3, expecting 4
WARNING: Entity <bound method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A34D9CC8
>> could not be transformed and will be executed as-is. Please report this to the AutgoGraph team. When filing the bug,
set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output. Cause: converting <boun
d method GRUCell.call of <tensorflow.python.ops.rnn_cell_impl.GRUCell object at 0x000001F6A34D9CC8>>: AssertionError: B
ad argument number for Name: 3, expecting 4
```

Finally, we are ready to run the training! A good indicator that everything works fine is decreasing loss during the training. You should account on the loss value equal to approximately 2.7 at the beginning of the training and near 1 after the 10th epoch.

In [39]:

```
%%time
session = tf.Session()
session.run(tf.global_variables_initializer())


invalid_number_prediction_counts = []
all_model_predictions = []
all_ground_truth = []

print('Start training... \n')
for epoch in range(n_epochs):
    random.shuffle(train_set)
    random.shuffle(test_set)

    print('Train: epoch', epoch + 1)
    for n_iter, (X_batch, Y_batch) in enumerate(generate_batches(train_set, batch_size=batch_size)):
        ####################################
        ######### YOUR CODE HERE #############
        ####################################
        # prepare the data (X_batch and Y_batch) for training
        # using function batch_to_ids
        X, X_seq_len = batch_to_ids(X_batch, word2id, max_len)
        Y, Y_seq_len = batch_to_ids(Y_batch, word2id, max_len)

        ######### YOUR CODE HERE #############
        predictions, loss = model.train_on_batch(session, X, X_seq_len,
                                                 Y, Y_seq_len, learning_rate,
                                                 dropout_keep_probability)

        if n_iter % 200 == 0:
            print("Epoch: [%d/%d], step: [%d/%d], loss: %f" % (epoch + 1, n_epochs, n_iter + 1, n_step, loss))

    X_sent, Y_sent = next(generate_batches(test_set, batch_size=batch_size))
    ####################################
    ######### YOUR CODE HERE #############
    ####################################
    # prepare test data (X_sent and Y_sent) for predicting
    # quality and computing value of the loss function
    # using function batch_to_ids
    X, X_seq_len = batch_to_ids(X_sent, word2id, max_len)
    Y, Y_seq_len = batch_to_ids(Y_sent, word2id, max_len)

    ######### YOUR CODE HERE #############
    predictions, loss = model.predict_for_batch_with_loss(session, X, X_seq_len, Y, Y_seq_len)

    print('Test: epoch', epoch + 1, 'loss:', loss,)
    for x, y, p  in list(zip(X, Y, predictions))[:3]:
        print('X:',''.join(ids_to_sentence(x, id2word)))
        print('Y:',''.join(ids_to_sentence(y, id2word)))
        print('O:',''.join(ids_to_sentence(p, id2word)))
        print('')

    model_predictions = []
    ground_truth = []
    invalid_number_prediction_count = 0
    # For the whole test set calculate ground-truth values (as integer numbers)
    # and prediction values (also as integers) to calculate metrics.
    # If generated by model number is not correct (e.g. '1-1'),
    # increase invalid_number_prediction_count and don't append this and corresponding
    # ground-truth value to the arrays.
    for X_batch, Y_batch in generate_batches(test_set, batch_size=batch_size):
        ####################################
        ######### YOUR CODE HERE #############
        ####################################
        X, X_seq_len = batch_to_ids(X_batch, word2id, max_len)
        Y_predicted = model.predict_for_batch(session, X, X_seq_len)
        Y_predicted = list(map(lambda y: ids_to_sentence(y, id2word), Y_predicted))
        for i in range(len(Y_predicted)):
            g = int(Y_batch[i])
            p = int(''.join([i for i in Y_predicted[i] if i not in ['$', '#']]))
            model_predictions.append(p)
            ground_truth.append(g)

            if g != p:
                invalid_number_prediction_count += 1

    all_model_predictions.append(model_predictions)
    all_ground_truth.append(ground_truth)
    invalid_number_prediction_counts.append(invalid_number_prediction_count)

print('\n...training finished.')
```

```
Start training...

Train: epoch 1
Epoch: [1/10], step: [1/625], loss: 2.710865
Epoch: [1/10], step: [201/625], loss: 1.815758
```

```
Epoch: [1/10], step: [401/625], loss: 1.746631
Epoch: [1/10], step: [601/625], loss: 1.619111
Test: epoch 1 loss: 1.5540242
X: 3709+6639$
Y: 10348$
O: 10045$

X: 4273-2130$
Y: 2143$#
O: 199$##

X: 1744-7181$
Y: -5437$
O: -5722$

Train: epoch 2
Epoch: [2/10], step: [1/625], loss: 1.638060
Epoch: [2/10], step: [201/625], loss: 1.601655
Epoch: [2/10], step: [401/625], loss: 1.464514
Epoch: [2/10], step: [601/625], loss: 1.434389
Test: epoch 2 loss: 1.4030632
X: 7389-2084$
Y: 5305$#
O: 5399$#

X: 8771-9176$
Y: -405$#
O: -100$#

X: 1657+1410$
Y: 3067$#
O: 3999$#

Train: epoch 3
Epoch: [3/10], step: [1/625], loss: 1.470803
Epoch: [3/10], step: [201/625], loss: 1.469481
Epoch: [3/10], step: [401/625], loss: 1.469299
Epoch: [3/10], step: [601/625], loss: 1.386456
Test: epoch 3 loss: 1.3521445
X: 3323+2972$
Y: 6295$#
O: 6882$#

X: 3560-430$#
Y: 3130$#
O: 3883$#

X: 478-88$###
Y: 390$##
O: 333$##

Train: epoch 4
Epoch: [4/10], step: [1/625], loss: 1.382184
Epoch: [4/10], step: [201/625], loss: 1.392265
Epoch: [4/10], step: [401/625], loss: 1.335881
Epoch: [4/10], step: [601/625], loss: 1.344808
Test: epoch 4 loss: 1.3102701
X: 7648+3008$
Y: 10656$
O: 10677$

X: 451+1928$#
Y: 2379$#
O: 2666$#

X: 6010-3590$
Y: 2420$#
O: 2664$#

Train: epoch 5
Epoch: [5/10], step: [1/625], loss: 1.341801
Epoch: [5/10], step: [201/625], loss: 1.340180
Epoch: [5/10], step: [401/625], loss: 1.306067
Epoch: [5/10], step: [601/625], loss: 1.312754
Test: epoch 5 loss: 1.2876402
X: 2874-5182$
Y: -2308$
O: -1741$

X: 9573-8035$
Y: 1538$#
O: 1744$#

X: 3927-4919$
Y: -992$#
O: -1012$

Train: epoch 6
```

```
Epoch: [6/10], step: [1/625], loss: 1.307083
Epoch: [6/10], step: [201/625], loss: 1.319121
Epoch: [6/10], step: [401/625], loss: 1.305488
Epoch: [6/10], step: [601/625], loss: 1.271333
Test: epoch 6 loss: 1.2309101
X: 9721+4424$
Y: 14145$
O: 14000$

X: 5713+2664$
Y: 8377$#
O: 8505$#

X: 5351+4633$
Y: 9984$#
O: 9755$#

Train: epoch 7
Epoch: [7/10], step: [1/625], loss: 1.252283
Epoch: [7/10], step: [201/625], loss: 1.223165
Epoch: [7/10], step: [401/625], loss: 1.254130
Epoch: [7/10], step: [601/625], loss: 1.190828
Test: epoch 7 loss: 1.1391248
X: 232+946$##
Y: 1178$#
O: 1255$#

X: 1453-6528$
Y: -5075$
O: -4988$

X: 4617-777$#
Y: 3840$#
O: 3842$#

Train: epoch 8
Epoch: [8/10], step: [1/625], loss: 1.153738
Epoch: [8/10], step: [201/625], loss: 1.169950
Epoch: [8/10], step: [401/625], loss: 1.097281
Epoch: [8/10], step: [601/625], loss: 1.025529
Test: epoch 8 loss: 1.0085304
X: 2053-2386$
Y: -333$#
O: -277$#

X: 4253-3856$
Y: 397$##
O: 388$##

X: 7085-8346$
Y: -1261$
O: -1330$

Train: epoch 9
Epoch: [9/10], step: [1/625], loss: 1.047751
Epoch: [9/10], step: [201/625], loss: 1.046207
Epoch: [9/10], step: [401/625], loss: 1.021450
Epoch: [9/10], step: [601/625], loss: 1.016213
Test: epoch 9 loss: 0.94405097
X: 1018-2251$
Y: -1233$
O: -1215$

X: 5529-2286$
Y: 3243$#
O: 3211$#

X: 9254-6428$
Y: 2826$#
O: 2850$#

Train: epoch 10
Epoch: [10/10], step: [1/625], loss: 1.022296
Epoch: [10/10], step: [201/625], loss: 0.982518
Epoch: [10/10], step: [401/625], loss: 0.937399
Epoch: [10/10], step: [601/625], loss: 0.974862
Test: epoch 10 loss: 0.92844474
X: 6179+8561$
Y: 14740$
O: 14718$

X: 499-2644$#
Y: -2145$
O: -2223$

X: 6005-2989$
Y: 3016$#
O: 3066$#
```

```
...training finished.
Wall time: 3min 5s
```

Evaluate results

Because our task is simple and the output is straight-forward, we will use MAE (https://en.wikipedia.org/wiki/Mean_absolute_error) metric to evaluate the trained model during the epochs. Compute the value of the metric for the output from each epoch.

In [40]:
```
from sklearn.metrics import mean_absolute_error
```

In [41]:
```
for i, (gts, predictions, invalid_number_prediction_count) in enumerate(zip(all_ground_truth,
                                                                             all_model_predictions,
                                                                             invalid_number_prediction_counts), 1):
    ######### YOUR CODE HERE #############
    mae = mean_absolute_error(gts, predictions)
    print("Epoch: %i, MAE: %f, Invalid numbers: %i" % (i, mae, invalid_number_prediction_count))
```

```
Epoch: 1, MAE: 864.862900, Invalid numbers: 19993
Epoch: 2, MAE: 420.314300, Invalid numbers: 19988
Epoch: 3, MAE: 256.350500, Invalid numbers: 19977
Epoch: 4, MAE: 202.377850, Invalid numbers: 19968
Epoch: 5, MAE: 191.275400, Invalid numbers: 19964
Epoch: 6, MAE: 130.557850, Invalid numbers: 19948
Epoch: 7, MAE: 94.382900, Invalid numbers: 19931
Epoch: 8, MAE: 62.053550, Invalid numbers: 19850
Epoch: 9, MAE: 40.844300, Invalid numbers: 19781
Epoch: 10, MAE: 31.495750, Invalid numbers: 19707
```

In [ ]:
```

```