RECURRENT NEURAL NETWORKS

JEAN-MARTIN ALBERT

1. Introduction

Let $U = \mathbb{R}^n$ and $V = \mathbb{R}^m$ be vector spaces, and consider a function $f : U \to W$. In theory, f here can be any set theoretic function, but for our purpose, we will assume that f is at the very least square integrable. In general, we consider a class $\mathcal{C}$ of functions $U \to V$, and try to find an element $g \in \mathcal{C}$ which is as close as possible to f. In order to do so, we need a notion of distance between functions. There are many possible choices, but some of the most popular are the L^p norms and the L^∞ norm.

While the class $\mathcal{C}$ van be virtually any class of functions, we usually choose a class that is somewhat definable, in the sense that there is a (differentiable) function $F : U \times \mathbb{R}^k \to V$ such that $\mathcal{C} = \{F(x, w) : w \in \mathbb{R}^k\}$. Such classes of functions have the advantage of being equicontinuous.

Example 1.1. Let $f(x) = x(x-1)(x+1)$. As an example, we will try to fit a linear function $\ell(x) = ux+v$ to f. The class $\mathcal{C}$ is defined by $\mathcal{C} = \{ux+v : u, v \in R\}$. For an element $g(x, u, v) \in \mathcal{C}$, we have

$$f(x) - g(x, u, v) = x^3 - x - ux + v$$

From this we get

$$\begin{aligned}(f(x) - g(x, u, v))^2 &= (x^3 - (1-u)x + v)^2 \\ &= x^6 - 2(1-u)x^4 + 2vx^3 + (1-u)^2x^2 - \\ &\quad 2(1-u)vx + v^2\end{aligned}$$

If we compute $\int_{-1}^{1}(f(x) - g(x, u, v))^2$, we get

$$\begin{aligned}\int_{-1}^{1}(f(x) - g(x; u, v))^2 &= \int_{-1}^{1}(x^3 - (1-u)x + v)^2 \\ &= \int_{-1}^{1} x^6 - 2(1-u)x^4 + 2vx^3 + (1-u)^2x^2 - 2(1-u)vx + v^2 \\ &= \frac{x^7}{7} - 2v\frac{x^4}{4} + (1-u)^2\frac{x^3}{3} - 2(1-u)v\frac{x^2}{2} + v^2x \\ &= \frac{1}{7} - 2v\frac{1}{4} + (1-u)^2\frac{1}{3} - 2(1-u)v\frac{1}{2} + v^2 \\ &\quad + \frac{1}{7} - 2v\frac{1}{4} + (1-u)^2\frac{-1}{3} - 2(1-u)v\frac{1}{2} - v^2 \\ &= \frac{2}{7} - (2-u)v + v^2 = \ell(u, v)\end{aligned}$$

Date: August 21, 2017.

and note that the last line above is a function of u and v alone, and we have to minimize it.

2. Back Propagation

In general, in order to minimize a function $f(w)$ of one variable, we make use of the fact that the extreme values of f all have $f'(x) = 0$. If f is a function of two or more variables, then if x is such that $f(x)$ is a local max, or minimum, then all partial derivatives $\frac{\partial f}{\partial w_i}(x) = 0$. If $f : \mathbb{R}^n \to \mathbb{R}$ is a function, then the *gradient* of f is the vector $\nabla f(x) = \left(\frac{\partial f}{\partial w_1}(x), ..., \frac{\partial f}{\partial w_n}(x)\right)$. One interesting property of $\nabla f(x)$ is that, as a vector, it "points" in the direction in which f decreases the most. That is to say, there is a good chance that $f(x - \alpha \nabla f(x)) \leq f(x)$. The chance is better the smaller α is. We see that if we iterate that process, we get a sequence $x_1, ..., x_n, ...$ such that $f(x_{i+1}) \leq f(x_i)$ for every i. Since $m \leq f(x_i) \leq f(x_0)$ for every i, we get a bounded decreasing sequence of real number, which converges.

Caveat: this ideal situation doesn't allways happen.

This iterative process is *(Stocastic) gradient descent*.

Suppose f is a complsition of two functions, and that we can write $f(x, w) = g(h(x, w_1), w_2)$. Then we can write $\nabla_w f(x, w) = \nabla g(h(x, w_1), w_2) \cdot \nabla h(x, w_1)$. Each level of composition corresponds to a layer of neural network. The chain rule transforms function composition into a product. If a network becomes deep, then the derivative of the loss function becomes a long product. When gradients become small in norm (like they do when we approach a min), then the gradient becomes very close to 0, and the update rule for gradient descent stops changing the weights.

This is the vanishing gradient problem for very deep networks, and makes convergence slow.

3. Dealing with sequences

Let A be a finite set. We define the set A^* of all finite sequences of elements of A to be the smallest set X with the property that the empty sequence ϵ is in X, and whenever $w \in X$ and $a \in A$, the sequence $aw \in X$. If A and B are two finite sets, and $f : A \to B$ is any function, then we can extend f extends uniquely to a function $f^* : A^* \to B^*$, which is defined by $f(a_1 a_2, ..., a_n) = f(a_1) f(a_2) \cdots f(a_n)$. This f^* is the only function extending f which satisfies $f(vw) = f(v)f(w)$.

Let S be another finite set with a distinguished element $\perp \in S$, and let $f : A \times S \to B \times S$ be a function. We can use f to define a function $f^* : A^* \to B^*$ in several ways. Consider an auxiliary function $f^* : A^* \to B^* \times S$ by defining First we define $f^*(\epsilon) = (\epsilon, \perp)$. If $a \in A$ then $f^*(a) = f(a, \perp)$. If $v = aw$, then by induction we can compute $f^*(w) = (u, s)$. Write $f(a, s) = (b, s')$, and define $f^*(aw) = (bu, s')$. We can produce a function $f^* : A^* \to B^*$ by taking $f^*(w) = \pi_2 g(w)$.

We can use f to define a function $f^* : A^* \to B$. Consider an auxiliary function $f^* : A^* \to B \times S$ by defining First we define $f^*(\epsilon) = (b_0, \perp)$. If $a \in A$ then $f^*(a) = f(a, \perp)$. If $v = aw$, then by induction we can compute $f^*(w) = (u, s)$. Write $f(a, s) = (b, s')$, and define $f^*(aw) = (b, s')$. We can produce a function $f^* : A^* \to B$ by taking $f^*(w) = \pi_2 g(w)$.

Let S be another finite set with a distinguished element $\perp \in S$, and let $f : A \times S \to B \times S$ be a function. We can use f to define a function $f^* : A \to B^N$ for every N in several ways. Consider an auxiliary function $f^* : A \to B^N \times S$ by defining.

If $a \in A$ then $f^*(a)_1 = f(a, \perp)$. Suppose $f^*(a)_n = (w, s)$, and $f(a, s) = (b, s')$, then $f^*(a)_{n+1} = (bw, s')$. This is an example of a one-to-many function. If A has a distinguished element a_0, then we can define $f^*(a)_1 = f(a, \perp)$, and $f^*(a)_n = (w, s)$, and $f(a_0, s) = (b, s')$, then $f^*(a)_{n+1} = (bw, s')$. This is an example of a one-to-many function. Note that this is a special case of the many to many function, where $f^*(a)$ is just defined as $f^*(aaa \cdots a)$, or $f^*(aa_0 \cdots a_0)$.

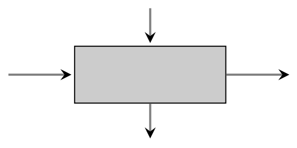
4. The base of recurrent networks

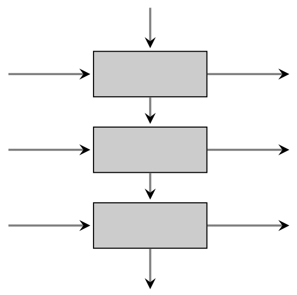
Consider a function $f : U \times S \to V \times S$, where U, V and S are finite dimensional vector spaces. Note that every vector space has a a distinguished element 0, the zero-vector. Let $u_1, ..., u_n$ be a finite sequence of vectors in U. We construct a sequence of vectors in V as follows. Write $f(u_1, 0) = (v_1, s_1)$, and for every i, if $f(u_i, s_{i-1}) = (v_i, s_i)$. Note that this is a state machine, close to a deterministic finite automaton, except that here the state space S is infinite.

The set $\{0, 1\}^*$ of all finite sequences of 0's and 1's is countable, and S, being a real vector space, is uncountable. Therefore, we can encode every element $w \in \{0, 1\}^*$ as a vector in S. Informally, S is enough to encode any finite state space, and every possible value for the content of the tape of a Turing machine. We get:

Theorem 4.1. *Recurrent neural networks are Turing-complete.*

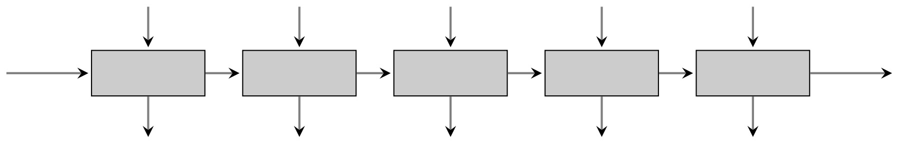
which explains why recurrent neural networks seem to be able to produce results that other networks can't. Training a recurrent network is the same as producing a Turing machine.





5. Backward Propagation Through Time

How do we train recurrent neural networks? we can use back propagation, just like a regular neural network. Heuristically, we unroll the recurrent network "infinitely" many times, until it looks like an ordinary very deep neural network. In practice, since computers only have a finite amount of resources, we only unroll the network a large but finite number of times, and treat it like an ordinary neural network.



6. Types of Recurrent Cells

There are three main types of recurrent cells which are used to build recurrent neural networks. Let us describe them, and give some of their basic properties, along with a naïve implementation in tensorflow.

6.1. **Basic Recurrent Cell.**

We begin with the most basic of recurrent cell. Abstractly, a function $f : U \times S \to V \times S$ can be defined using two functions $u : \mathbb{R}^n \times \mathbb{R}^\ell \to \mathbb{R}^n$ and $v : \mathbb{R}^n \times \mathbb{R}^\ell \to \mathbb{R}^\ell$. We se ethe former function as providing the output, and the latter function as a state updating function. The most common definition for u and v for basic recurrent cells is as follows: $u(x, s) = f(A_u x + B_u s)$, where A nad B are matrices, and f is a non-linear function, and $v(x, s) = \tanh(A_v x + B_v s)$. Te parameters in this definition are A_u, B_u, A_v and B_v. In tehsorflow, the code looks like:

```
def basic_rnn_cell(input_tensor, state_tensor):
  A_u = tf.Variable(shape=[N, L])
  B_u = tf.Variable(shape=[L, L])
  A_v = tf.Variable(shape=[N, L])
  B_v = tf.Variable(shape=[L, L])
  output_tensor = tf.relu(tf.matmul(input_tensor, A_u) +
                          tf.matmul(state_tensor, B_u))
  new_state_tensor = tf.tanh(tf.matmul(input_tensor, A_v) + \
                                tf.matmul(state_tensor, B_v))
  return output_tensor, new_state_tensor
```

6.2. **Long Short-term Memory Cell.**

There are a few problems with the architecture described above. First, it suffers greatly from the vanishing gradient problem, since there is no way to prevent gradients from becoming very small. Secondly, simple recurrent networks have a hard time remembering facts about the input sequence. To remedy this situation, long short-term memory cells were introduced. The overall structure of an LSTM cell is almost the same as the basic RNN cell. This time we take the previous output into account, which gives the structure as $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^m \times \mathbb{R}^\ell$ which can be divided as two function $u : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^m$ and $v : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^\ell$. To define the functione u and v, we use auxiliary functions F, I, and O defined as follows

$$\begin{aligned}
F(x, y) &= \sigma(A_F x + B_F y + b_F) \\
I(x, y) &= \sigma(A_I x + B_I y + b_I) \\
O(x, y) &= \sigma(A_O x + B_O y + b_O) \\
S(x, y) &= \sigma(A_O x + B_O y + b_O)
\end{aligned}$$

The state update is given by $v(x, y, s) = F(x, y) \circ s + I(x, y) \circ \sigma_v(A_v x + B_v y + b_v)$, where $\circ$ denotes pointwise multiplication of vectors, and finally, the output can be defined as $u(x, y, s) = O(x, y) \circ \sigma_u(v(x, y, s))$.

```
def lstm_gate(input_tensor, previous_output, port_op):
  A = tf.Variable(shape=[N, L])
  B = tf.Variable(shape=[L, L])
  b = tf.Variable(shape=[L, L])
  x = tf.matmul(input_tensor, A)+ tf.matmul(previous_output, B) + b
  return post_op(x)

def lstm_cell(input_tensor, output, state):
  F = lstm_gate(input_tensor, output, tf.sigmoid)
  I = lstm_gate(input_tensor, output, tf.sigmoid)
  O = lstm_gate(input_tensor, output, tf.sigmoid)
  S = lstm_gate(input_tensorm output, tf.tanh)
  new_state = tf.mul(output, F) + tf.mul(I, S)
  output = tf.mul(O, tf.tanh(new_state))
  return output, new_state
```

6.3. **Gated Recurrent Unit Cell.** A common variant of the long short term memory cell is the *gated recurrent unit cell*, more commonly known as GRU cells. The philosophy behind their design is similar to the long short term memory. Once again, each step of the computation takes into account a state vector and the output of the previous iteration. GRU's are functions $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^m \times \mathbb{R}^\ell$, which can be divided as two function $u : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^m$ and $v : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^\ell$. To define the functione u and v, we use auxiliary functions U and R defined as follows

$$\begin{aligned} U(x, y) &= \sigma(A_U x + B_U y + b_U) \\ R(x, y) &= \sigma(A_R x + B_R y + b_R) \end{aligned}$$

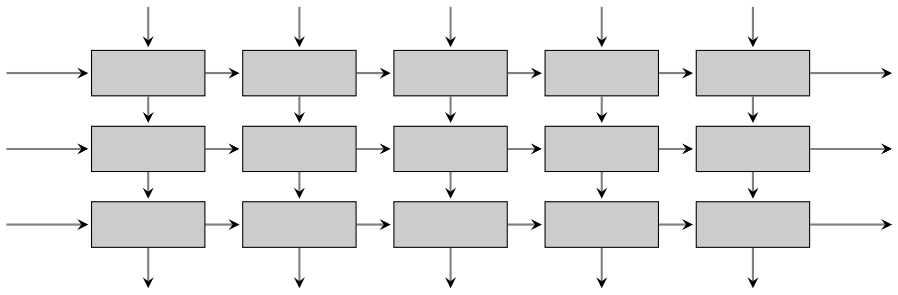
The state update is given by $v(x, y, s) = U(x, y) \circ s + (1 - s) \circ \sigma_h(A_v x + B_v(R(x, y) \circ y) + b_v)$.

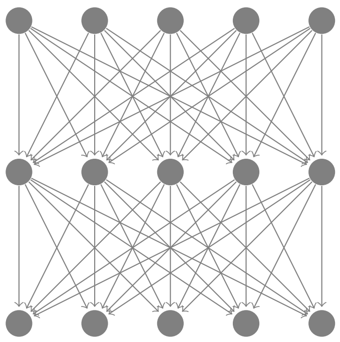
```
def gru_gate(input_tensor, previous_output, port_op):
  A = tf.Variable(shape=[N, L])
  B = tf.Variable(shape=[L, L])
  b = tf.Variable(shape=[L, L])
  x = tf.matmul(input_tensor, A)+ tf.matmul(previous_output, B) + b
  return post_op(x)

def gru_cell(input_tensor, output, state):
  U = gru_gate(input_tensor, output, tf.sigmoid)
  R = gru_gate(input_tensor, output, tf.sigmoid)
  O = gru_gate(input, tf.mul(R, output))
  return tf.mul(R, output) + tf.mul((1-R)O)
```

7. Examples

7.1. **Many-to-One.** The Buzzometer sentiment analysis tool uses a bi-directional GRU model, in which the output a 4-layer bidirectional recurrent network is fed into a two-layer fully connected network which separates the input into four classes, corresponding to neutral, positive, negative and irrelevant messages. The only non-linearities in the network are inside the GRU cells. Graphically, the network can be represented as:





The base input ofor this network is a (unicode) character, which is one-hot encoded before being fed into the recurrent layers. For training, the network is unrolled to a length of 256 characters, which is about the length of an average message in our database. All strings are padded or truncated to a length of 256 characters.

```
SEQ_LENGTH = 256
E_DIM = 128
STATE_DIM = 512
NUM_CLASSES = 4
def inference():
    model_input = tf.placeholder('uint8', shape=[None, SEQ_LENGTH])
    _ = tf.one_hot(Globals.model_input, depth=E_DIM, axis=-1)
    _ = tf.reshape(_, [-1, SEQ_LENGTH, E_DIM])
    fw = multi_layer_rnn(N_LAYERS, STATE_DIM)
    bw = multi_layer_rnn(N_LAYERS, STATE_DIM)
    output, _ = tf.nn.bidirectional_dynamic_rnn(fw, bw, _, dtype=tf.float32)
    fw_output = tf.reshape(output[0][:, -1:], [-1, STATE_DIM])
    bw_output = tf.reshape(output[1][:, :1], [-1, STATE_DIM])
    f = project(fw_output, E_DIM)
    b = project(bw_output, E_DIM)
    e = tf.add(f, b)
    Globals.model_output = project(e, NUM_CLASSES)
    Globals.prediction = tf.cast(tf.argmax(Globals.model_output, 1), tf.uint8)
    return Globals.model_input, Globals.model_output
```

first hour: Loss: 1.331 Accuracy: 45.500 Negative Neutral Positive Irrelevant Negative 291 113 50 127 Neutral 108 164 62 85 Positive 0 0 0 0 Irrelevant 0 0 0 0

Loss: 0.933 Accuracy: 60.300 Negative Neutral Positive Irrelevant Negative 292 52 36 55 Neutral 61 176 21 62 Positive 5 15 65 15 Irrelevant 33 28 14 70

second hour Loss: 0.729 Accuracy: 71.800 Negative Neutral Positive Irrelevant Negative 340 67 18 65 Neutral 22 182 18 36 Positive 10 7 91 10 Irrelevant 8 13 8 105

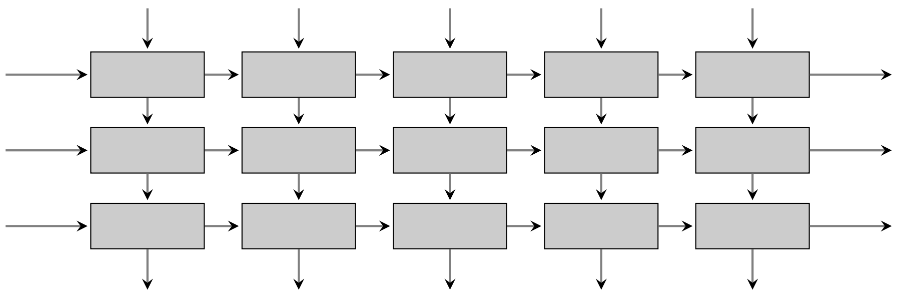
third hour Loss: 0.611 Accuracy: 76.700 Negative Neutral Positive Irrelevant Negative 319 38 11 43 Neutral 20 188 2 20 Positive 10 18 91 10 Irrelevant 27 27 7 169

Loss: 0.468 Accuracy: 85.200 Negative Neutral Positive Irrelevant Negative 334 20 3 18 Neutral 14 242 0 9 Positive 20 10 125 17 Irrelevant 26 10 1 151

0 Negative Neutral Positive Irrelevant Negative 334 4 4 9 Neutral 3 283 2 7 Positive 1 0 125 2 Irrelevant 1 0 4 221

Loss: 0.530 Accuracy: 84.600 Negative Neutral Positive Irrelevant Negative 361 18 13 29 Neutral 22 246 10 18 Positive 6 6 95 6 Irrelevant 11 5 10 144

Loss: 0.275 Accuracy: 90.000 Negative Neutral Positive Irrelevant Negative 321 16 3 20 Neutral 8 286 4 6 Positive 4 2 125 16 Irrelevant 11 9 1 168



7.2. **Many-to-many.**

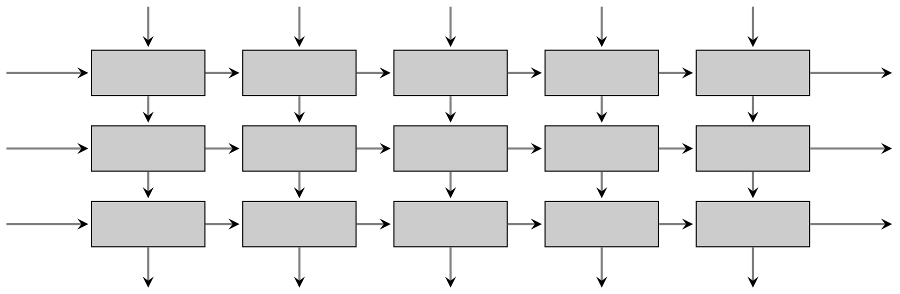
```
SEQ_LENGTH = 256
E_DIM = 128
STATE_DIM = 512
N_LAYERS = 3

def inference():
    model_input = tf.placeholder('uint8', shape=[None, SEQ_LENGTH])
    _ = tf.one_hot(Globals.model_input, depth=E_DIM, axis=-1)
    encode = multi_layer_rnn(N_LAYERS, STATE_DIM)
    state_tuple = tuple(tf.unstack(Globals.initial_state, axis=0))
    output, state = tf.nn.dynamic_rnn(encode, _,
                                      dtype=tf.float32,
                                      initial_state=state_tuple)
    output = tf.reshape(output, [-1, STATE_DIM])
    output = project(output, E_DIM)
```

```
      out = tf.cast(tf.argmax(output, 1), tf.uint8)
      out = tf.reshape(out, [-1, SEQ_LENGTH])
      Globals.generated_sequence = out
      Globals.generated_characters = tf.nn.softmax(output)
      Globals.model_output = output
      Globals.state = state


def generate_text(length, session=None):
    generated_text = ''
    character = [[ord(' ')]]
    istate = np.zeros([N_LAYERS, 1, STATE_DIM])
    while len(generated_text) < length:
        feed_dict = {Globals.model_input: character,
                     Globals.initial_state: istate}
        next_char, state = session.run([Globals.generated_characters,
                                        Globals.state],
                                       feed_dict=feed_dict)
        next_char = np.asarray(next_char).astype('float64')
        next_char = next_char / next_char.sum()
        op = np.random.multinomial
        next_char_id = op(1, next_char.squeeze(), 1).argmax()
        next_char_id = next_char_id if chr(next_char_id) in \
                          string.printable else ord(" ")
        generated_text += chr(next_char_id)
        character = [[next_char_id]]
        istate = state
    return generated_text
```

7.3. **Many-to-one-to-many.** As an example of a recurrent neural network performing a task that ordinary neural network should not be able to perform, we present a sorting function which uses GRU cells. The architecture of the network is the reason we call it a many-to-one-to-many network. One level takes a list of numbers, and produces a vector which we can see as representing all the numbers in the list, and a second layer uses this output vector as input, and produces the original list, sorted in increasing order. The training was done using sequences of natural numbers $n \in \{1, ..., 32\}$, and each of the training sequences of length 32. After about 6 days of training, the accuracy of the network stabilized at around 95%, and it is worth noting that eventhough it took a long time to breach the 90% accuracy mark, very early on in the training did the network output increasing sequences.



```
SEQ_LENGTH = 256
E_DIM = 128
STATE_DIM = 512
N_LAYERS = 4

def inference():
    model_input = tf.placeholder('uint8', shape=[None, SEQ_LENGTH])
    _ = tf.one_hot(Globals.model_input, depth=E_DIM, axis=-1)
    _ = tf.reshape(_, [-1, SEQ_LENGTH, E_DIM])
    encode = multi_layer_rnn(N_LAYERS, STATE_DIM)
    encoded_input, state = tf.nn.dynamic_rnn(encode,
                                             _,
                                             dtype=tf.float32)
    Globals.encoder_output = state
    with tf.variable_scope('decoder'):
        training_decoder_input = tf.zeros_like(Globals.model_input)
        _ = tf.one_hot(training_decoder_input, depth=E_DIM, axis=-1)
        _ = tf.reshape(_, [-1, SEQ_LENGTH, E_DIM])
        decode = multi_layer_rnn(N_LAYERS, STATE_DIM)
        decoded_output, state = tf.nn.dynamic_rnn(decode, _,
                                                  dtype=tf.float32,
                                                  initial_state=state)
        decoded_output = tf.reshape(decoded_output, [-1, STATE_DIM])
        output = project(decoded_output, E_DIM)
        out = tf.cast(tf.argmax(output, 1), tf.uint8)
        out = tf.reshape(out, [-1, SEQ_LENGTH])
        Globals.training_decoder_input = training_decoder_input
        Globals.model_output = output
        Globals.prediction = out
        Globals.decoder = decode
        Globals.decoder_input = _
```