In [1]:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

In [2]:

```
# numpy
size=10
X = np.array(np.random.choice(2, size=(size,)))
print(X)
```

```
[0 1 1 1 1 1 0 1 1 0]
```

In [3]:

```
def gen_data(size=1000000):
    """
    generate 2 lists of binary code with some pattern
    """

    X = np.array(np.random.choice(2, size=(size,)))
    Y = []
    for i in range(size):
        threshold = 0.5
        if X[i-3] == 1:
            threshold += 0.5
        if X[i-8] == 1:
            threshold -= 0.25
        if np.random.rand() > threshold:
            Y.append(0)
        else:
            Y.append(1)
    return X, np.array(Y)

def gen_batch(raw_data, batch_size, num_steps):
    """
    the data generator
    """

    raw_x, raw_y = raw_data
    batch_partition_length = len(raw_x) // batch_size

    # initialize
    data_x = np.zeros([batch_size, batch_partition_length], dtype=np.int32)
    data_y = np.zeros([batch_size, batch_partition_length], dtype=np.int32)

    for i in range(batch_size):
        data_x[i] = raw_x[batch_partition_length * i:batch_partition_length * (i + 1)]
        data_y[i] = raw_y[batch_partition_length * i:batch_partition_length * (i + 1)]

    # further divide batch partitions into num_steps for truncated backprop
    seq_size = batch_partition_length // num_steps

    for i in range(seq_size):
        x = data_x[:, i * num_steps:(i + 1) * num_steps]
        y = data_y[:, i * num_steps:(i + 1) * num_steps]
        yield (x, y)

def gen_epochs(n, batch_size, num_steps):
    for i in range(n):
        yield gen_batch(gen_data(), batch_size, num_steps)
```

In [8]:

```
# not used
# def list_block(a,i,len_block=1):
#    return a[len_block * (i-1):len_block * i]

n_epochs = 5
num_steps = 5 # number of truncated backprop steps ('n' in the discussion above)
batch_size = 200

# now, you can acess the data by:
for epoch in gen_epochs(n_epochs, batch_size, num_steps):
    for (X,Y) in epoch:
        print(X.shape)
        print(Y.shape)
        break
    break
```

```
(200, 5)
(200, 5)
```

we first show how to build rnn purely with basic tensorflow code

In [ ]:

```
# rnn_inputs is a list of num_steps tensors with shape [batch_size, num_classes]
x = tf.placeholder(tf.int32,   [batch_size, num_steps], name='input_placeholder')
y = tf.placeholder(tf.float32, [batch_size, num_steps], name='labels_placeholder')

num_classes = 2

'''
    in tensor flow, the first dimension of tensor is always batch size,
    but for RNN, we need to train each step of the input separately,
    so here we did many operations on the dimension of the tensor.

'''


# Turn our x placeholder into a list of one-hot tensors:
# [1,0,1],[1,0,0] to [0,1][1,0][0,1], [..][..][..]
x_one_hot = tf.one_hot(x, num_classes)

# now the dimension is like (batch, steps, signal)
# I want rnn_input for rnn_inputs, so the second dimension should come first
rnn_inputs = tf.unstack(x_one_hot, axis=1)

"""
see the tf.name_scope and tf.variable_scope

with tf.variable_scope('rnn_cell'):
    # W = Wxa + Waa
    W = tf.get_variable('W', [num_classes + state_size, state_size])
    b = tf.get_variable('b', [state_size], initializer=tf.constant_initializer(0.0))

def rnn_cell(rnn_input, state):
    with tf.variable_scope('rnn_cell', reuse=True):
        W = tf.get_variable('W', [num_classes + state_size, state_size])
        b = tf.get_variable('b', [state_size], initializer=tf.constant_initializer(0.0))

    # return the a (the state)
    return tf.tanh(tf.matmul(tf.concat([rnn_input, state], 1), W) + b)

"""


state_size = 4

with tf.variable_scope('rnn_cell'):
    # W = Wxa + Waa
    W = tf.get_variable('W', [num_classes + state_size, state_size])
    b = tf.get_variable('b', [state_size], initializer=tf.constant_initializer(0.0))

init_state = tf.zeros([batch_size, state_size])
state = init_state
rnn_outputs = []
for rnn_input in rnn_inputs:

    with tf.variable_scope('rnn_cell', reuse=True):
        # W = Wxa + Waa
        W = tf.get_variable('W', [num_classes + state_size, state_size])
        b = tf.get_variable('b', [state_size], initializer=tf.constant_initializer(0.0))

    state = tf.tanh(tf.matmul(tf.concat([rnn_input, state], 1), W) + b)
    rnn_outputs.append(state)
```

```
final_state = rnn_outputs[-1]

W = tf.get_variable('W', [state_size, num_classes])
b = tf.get_variable('b', [num_classes], initializer=tf.constant_initializer(0.0))

logits = [tf.matmul(rnn_output, W) + b for rnn_output in rnn_outputs]
predictions = [tf.nn.softmax(logit) for logit in logits]
# in short = predictions =  [tf.nn.softmax(tf.matmul(rnn_output, W) + b) for rnn_output in rnn_outpu
y_pred = tf.transpose(predictions,[2,1,0])
# 0 stands for the id in sequence
# 1 stands for the id of the input
# 2 stands for the decode of input

# Turn our y placeholder into a list of labels
y_as_list = tf.unstack(y, num=num_steps, axis=1)

#losses and train_step
# losses = [tf.nn.sparse_softmax_cross_entropy_with_logits(labels=label, logits=logit) for \
#           logit, label in zip(logits, y_as_list)]



losses = -y*tf.log(y_pred[0])

total_loss = tf.reduce_mean(losses)
train_step = tf.train.AdagradOptimizer(0.1).minimize(total_loss)
```

In [ ]:

```
def train_network(num_epochs, num_steps, state_size=4, verbose=True):
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        training_losses = []
        for idx, epoch in enumerate(gen_epochs(num_epochs, num_steps)):
            training_loss = 0
            training_state = np.zeros((batch_size, state_size))
            if verbose:
                print("\nEPOCH", idx)
            for step, (X, Y) in enumerate(epoch):
                tr_losses, training_loss_, training_state, _ = \
                    sess.run([losses,
                              total_loss,
                              final_state,
                              train_step],
                             feed_dict={x:X, y:Y, init_state:training_state})
                training_loss += training_loss_
                if step % 100 == 0 and step > 0:
                    if verbose:
                        print("Average loss at step", step,
                              "for last 250 steps:", training_loss/100)
                    training_losses.append(training_loss/100)
                    training_loss = 0

    return training_losses

training_losses = train_network(1,num_steps)
plt.plot(training_losses)

tf.reset_default_graph()
```

now the rnn of tensorflow

In [ ]:

```
###########################################################################
# to undestand and use this, keep in mind:
# what is rnn_inputs
# what is init_state
# what is outputs
# what is final state

# cells = tf.nn.rnn_cell.BasicRNNcell(state_size)
# cells = tf.nn.rnn_cell.BasicGRUcell(state_size)
# cells = tf.nn.rnn_cell.BasicLSTMcell(state_size)

# rnn_outputs, final_state = tf.contrib.rnn.static_rnn(cells,rnn_inputs,initial_state=init_state)


###########################################################################
# dynamic model
# in dynamic model the training data x should be [batch size, num_steps, features(dim of input)]

num_steps = 5 # number of truncated backprop steps ('n' in the discussion above)
batch_size = 200
num_classes = 2
state_size = 4
learning_rate = 0.1

x = tf.placeholder(tf.int32, [batch_size, num_steps], name='input_placeholder')
y = tf.placeholder(tf.int32, [batch_size, num_steps], name='labels_placeholder')

# num_classes is like dimension of input.
# state_size is like dimension of hiden state.
rnn_inputs = tf.one_hot(x, num_classes)
# here the rnn_inputs dimension is not same as before

cell = tf.contrib.rnn.BasicRNNCell(state_size)

init_state = tf.zeros([batch_size, state_size])

rnn_outputs, final_state = tf.nn.dynamic_rnn(cell, rnn_inputs, initial_state=init_state)

W = tf.get_variable('W', [state_size, num_classes])
b = tf.get_variable('b', [num_classes], initializer=tf.constant_initializer(0.0))

# I assume the rnn_outputs in in the form of [batch_size, num_steps, state_size]
logits = tf.reshape(
            tf.matmul(tf.reshape(rnn_outputs, [-1, state_size]), W) + b,
            [batch_size, num_steps, num_classes])

predictions = tf.nn.softmax(logits)

losses = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
# alternative:
# logits = tf.transpose(logits, [2,0,1])
# y_pred = tf.nn.sortmax(logits)
# losses = -y*tf.log(y_pred[0])

total_loss = tf.reduce_mean(losses)
train_step = tf.train.AdagradOptimizer(learning_rate).minimize(total_loss)

training_losses = train_network(1,num_steps)
plt.plot(training_losses)
```