LSTMs/GRUs e a CTC

In [1]:

```
#  Compatibility imports
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
import numpy as np
import pickle
import random
from six.moves import xrange as range

import matplotlib.pyplot as plt
%matplotlib inline
```

```
/Users/marcocristo/Library/Enthought/Canopy_64bit/User/lib/python2.
7/site-packages/matplotlib/font_manager.py:273: UserWarning: Matplot
lib is building the font cache using fc-list. This may take a momen
t.
  warnings.warn('Matplotlib is building the font cache using fc-lis
t. This may take a moment.')
```

Para testarmos a CTC, vamos usar uma pequena coleção de imagens que representam os dígitos de 0 a 9. Cada imagem corresponde a uma sequência aleatória de dígitos de comprimento variável (e número de digitos variáveis).

In [2]:

```
with open('data/test_varlen.pkl', 'rb') as f:
    data = pickle.load(f)
```

In [3]:

```
print (data['chars'])
print (len(data['chars']))
print ('instances', len(data['x']))
# height and weight for first image
print ('height_0:', len(data['x'][0]))
print ('width_0:', len(data['x'][0][0]))
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
10
instances 100
height_0: 9
width_0: 38
```

Abaixo, podemos ver as primeiras duas sequências de dígitos, uma com 4 dígitos (38 pixels de largura) e outra com 9 dígitos (em 60 pixels de largura). O rótulo correspondente à primeira sequência indica os dígitos que estão presentes (0, 1, 2 e 3). Para a segunda, são 4, 5, 6, 7, 8, 9, 5, 3 e 7. Todas as imagens na coleção têm 8 pixels de altura.

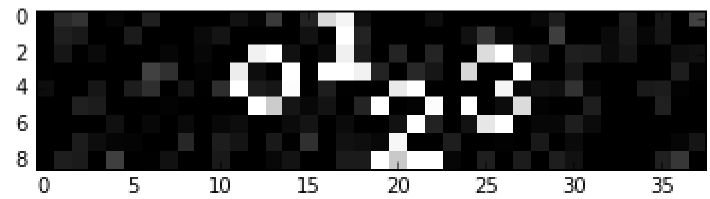
In [4]:

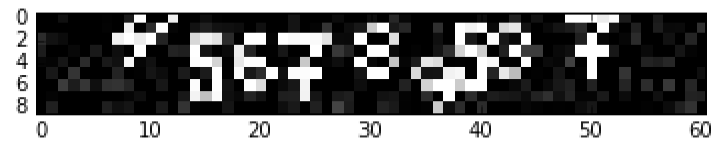
```
slabs = [(np.asarray(data['x'][i]), np.asarray(data['y'][i]))
         for i in range(len(data['x']))]
```

In [5]:

```
for i in range(2):
    print(slabs[i][1])
    plt.figure()
    plt.imshow(slabs[i][0], cmap = 'gray', interpolation = 'nearest')
```

```
[0 1 2 3]
[4 5 6 7 8 9 5 3 7]
```





A função a seguir lê esta coleção para a memória, fornecendo as sequências, seus rótulos correspondentes e o número de sequências válidas lidas.

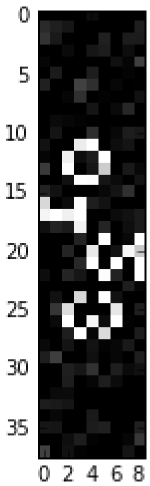
In [6]:

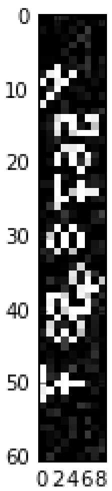
```
# load the training or test dataset from disk
def get_toy_data(pname):
    with open(pname, 'rb') as f:
        data = pickle.load(f)
    num_examples = len(data['x'])
    # note that image shape is modified to columns x lines, since it is expected
 a
    # fixed number of lines and a varying number of columns
    seqs = np.asarray([data['x'][t].swapaxes(0, 1)
                       for t in range(num_examples) if len(data['y'][t]) > 0])
    # get label seqs
    labs = np.asarray([np.asarray(data['y'][t])
                       for t in range(num_examples) if len(data['y'][t]) > 0])
    return seqs, labs, len(labs)
```

Note que as sequências são transpostas de forma a serem representadas por tantos intervalos de tempo quanto forem as suas larguras e 8 'atributos de entrada' já que isto corresponde a sua altura em pixels.

In [7]:

```
seqs, _, _ = get_toy_data('data/test_varlen.pkl')
for i in range(2):
    plt.figure()
    plt.imshow(seqs[i], cmap = 'gray', interpolation = 'nearest')
```





Para usar a CTC do tensorflow, precisamos colocar as sequências de entrada no formato esparso usado pela função.

In [8]:

```
def sparse_tuple_from(sequences, dtype=np.int32):
    """Create a sparse representention of x.
    Args:
        sequences: a list of lists of type dtype where each element is a sequenc
e
    Returns:
        A tuple with (indices, values, shape)
    """
    indices = []
    values = []

    for n, seq in enumerate(sequences):
        indices.extend(zip([n]*len(seq), range(len(seq))))
        values.extend(seq)

    indices = np.asarray(indices, dtype=np.int64)
    values = np.asarray(values, dtype=dtype)
    shape = np.asarray([len(sequences), np.asarray(indices).max(0)[1]+1],
dtype=np.int64)

    return indices, values, shape
```

Por exemplo, duas sequências de entrada, (0, 1, 3) e (5, 8, 9, 3, 2), serão convertidas para a forma I, V, S, onde I corresponde a uma sequência de pares que indicam o índice da sequência e do valor dentro da sequência; V corresponde às observações correspondentes a cada par; S é um par indicando o número de sequências e o número máximos de valores observados.

In [9]:

```
sparse_tuple_from([[0,1,3], [5,8,9,3,2]])
```

Out[9]:

```
(array([[0, 0],
        [0, 1],
        [0, 2],
        [1, 0],
        [1, 1],
        [1, 2],
        [1, 3],
        [1, 4]]), array([0, 1, 3, 5, 8, 9, 3, 2], dtype=int32), arra
y([2, 5]))
```

Como as sequências de entrada podem ter tamanho variável, elas precisam ser preenchidas para terem o mesmo tamanho em um certo batch. A função a seguir faz isso:

In [10]:

```
def pad_sequences(sequences, maxlen=None, dtype=np.float32,
                  padding='post', truncating='post', value=0.):
    '''Pads each sequence to the same length: the length of the longest sequenc
e.
        If maxlen is provided, any sequence longer than maxlen is truncated to
        maxlen. Truncation happens off either the beginning or the end
        (default) of the sequence. Supports post-padding (default) and
        pre-padding.
```

```
        Args:
            sequences: list of lists where each element is a sequence
            maxlen: int, maximum length
            dtype: type to cast the resulting sequence.
            padding: 'pre' or 'post', pad either before or after each sequence.
            truncating: 'pre' or 'post', remove values from sequences larger
            than maxlen either in the beginning or in the end of the sequence
            value: float, value to pad the sequences to the desired value.
        Returns
            x: numpy array with dimensions (number_of_sequences, maxlen)
            lengths: numpy array with the original sequence lengths
    '''
    lengths = np.asarray([len(s) for s in sequences], dtype=np.int64)

    nb_samples = len(sequences)
    if maxlen is None:
        maxlen = np.max(lengths)

    # take the sample shape from the first non empty sequence
    # checking for consistency in the main loop below.
    sample_shape = tuple()
    for s in sequences:
        if len(s) > 0:
            sample_shape = np.asarray(s).shape[1:]
            break

    x = (np.ones((nb_samples, maxlen) + sample_shape) * value).astype(dtype)
    for idx, s in enumerate(sequences):
        if len(s) == 0:
            continue  # empty list was found
        if truncating == 'pre':
            trunc = s[-maxlen:]
        elif truncating == 'post':
            trunc = s[:maxlen]
        else:
            raise ValueError('Truncating type "%s" not understood' % truncating)

        # check `trunc` has expected shape
        trunc = np.asarray(trunc, dtype=dtype)
        if trunc.shape[1:] != sample_shape:
            raise ValueError('Shape of sample %s of sequence at position %s is d
ifferent from expected shape %s' %
                             (trunc.shape[1:], idx, sample_shape))

        if padding == 'post':
            x[idx, :len(trunc)] = trunc
        elif padding == 'pre':
            x[idx, -len(trunc):] = trunc
        else:
            raise ValueError('Padding type "%s" not understood' % padding)
    return x, lengths
```

In [11]:

```
print (pad_sequences([[1,2],[1,2,3,4],[1,2,3]]))
print (pad_sequences([[1,2],[1,2,3,4],[1,2,3]], padding = 'pre'))
print (pad_sequences([[1,2],[1,2,3,4],[1,2,3]], maxlen = 3))
```

```
(array([[ 1.,  2.,  0.,  0.],
       [ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  0.]], dtype=float32), array([2, 4, 3]))
(array([[ 0.,  0.,  1.,  2.],
       [ 1.,  2.,  3.,  4.],
       [ 0.,  1.,  2.,  3.]], dtype=float32), array([2, 4, 3]))
(array([[ 1.,  2.,  0.],
       [ 1.,  2.,  3.],
       [ 1.,  2.,  3.]], dtype=float32), array([2, 4, 3]))
```

Para avaliar e nossa implementação, vamos usar uma função que usa a RNN para estimar o rótulo de uma sequência de entrada e decodifica a saída da RNN em forma de uma sequência. As sequências são exibidas antes das previstas, de forma a avaliar a RNN.

Finalmente, temos a nossa rede, usando uma RNN dinâmica com células LSTM na camada oculta.

In [12]:

```
tf.reset_default_graph()

# number de pixel in columns
num_features = 9
# total of digits + blank = 11 symbols
num_classes = 11

# Hyper-parameters
num_hidden = 35
learning_rate = 1e-2

# data
ftrain = 'data/train_varlen.pkl'
input_seqs, seq_labels, num_examples = get_toy_data(ftrain)
```

In [13]:

```
# e.g: output of a convolutional net
# Has size [batch_size, max_stepsize, num_features], but the
# batch_size and max_stepsize can vary along each step
inputs = tf.placeholder(tf.float32, [None, None, num_features])
shape = tf.shape(inputs)
batch_s, max_timesteps = shape[0], shape[1]

# Here we use sparse_placeholder that will generate a
# SparseTensor required by ctc_loss op.
targets = tf.sparse_placeholder(tf.int32)

# 1d array of size [batch_size]
seq_len = tf.placeholder(tf.int32, [None])

# Defining the cell
cell = tf.contrib.rnn.LSTMCell(num_hidden, state_is_tuple=True)

# The second output is the last state and we will no use that
outputs, _ = tf.nn.dynamic_rnn(cell, inputs, seq_len, dtype=tf.float32)

# Reshaping to apply the same weights over the timesteps
outputs = tf.reshape(outputs, [-1, num_hidden])

# W init = truncated normal with mean 0 and stdev=0.1
# bias init Zero initialization
W = tf.Variable(tf.truncated_normal([num_hidden,
                                     num_classes],
                                    stddev=0.1))
b = tf.Variable(tf.constant(0., shape=[num_classes]))
### do not include sigmoid because CTC will do that
logits = tf.matmul(outputs, W) + b

# Reshaping back to the original shape
logits = tf.reshape(logits, [batch_s, -1, num_classes])
# Time major
logits = tf.transpose(logits, (1, 0, 2))

loss = tf.nn.ctc_loss(targets, logits, seq_len)
cost = tf.reduce_mean(loss)
optimizer = tf.train.MomentumOptimizer(learning_rate,
                                       0.9).minimize(cost)

# Option 2: tf.nn.ctc_beam_search_decoder (slower but better)
decoded, log_prob = tf.nn.ctc_greedy_decoder(logits, seq_len)

# Inaccuracy: label error rate
ler = tf.reduce_mean(tf.edit_distance(tf.cast(decoded[0], tf.int32),
                                      targets))
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

In [14]:

```
def show_decoded_seqs(session, X, Y):
    # Padding input to max_time_step of this batch
    Xp, seqlen = pad_sequences(X)

    # Converting to sparse representation so as to to feed SparseTensor input
    Ys = sparse_tuple_from(Y)

    # Decoding
    d = session.run(decoded[0],
                    feed_dict={inputs: Xp,
                               targets: Ys,
                               seq_len: seqlen})
    d_dense = tf.sparse_tensor_to_dense(d, default_value=-1)
    dense_decoded = d_dense.eval(session=session)
    for i, seq in enumerate(dense_decoded):
        seq = ' '.join([str(s) for s in seq if s != -1])
        print('\t%d %s --> [%s]' % (i, Y[i], seq))
```

In [16]:

```
# You can preprocess the input data here
Xtrain = input_seqs
Ytrain = seq_labels

batch_size = 5
num_epochs = 10
num_batches_per_epoch = int(num_examples/batch_size)

# random cases to evaluate
cases_to_show = np.random.randint(0, num_examples - 1, (10,))

with tf.Session() as s:
    # Initializate the weights and biases
    init.run()

    for e in range(num_epochs):
        train_cost = train_ler = 0
        for batch in range(num_batches_per_epoch):

            # Getting the index
            indexes = [i % num_examples
                       for i in range(batch * batch_size, (batch + 1) * batch_si
ze)]
            Xtrain_b = Xtrain[indexes]

            # Padding input to max_time_step of this batch
            Xtrain_b, seqlen_train_b = pad_sequences(Xtrain_b)

            # Converting to sparse representation so as to to feed SparseTensor
 input
            Ytrain_b = sparse_tuple_from(Ytrain[indexes])

            feed = {inputs: Xtrain_b,
                    targets: Ytrain_b,
                    seq_len: seqlen_train_b}
            batch_cost, _ = s.run([cost, optimizer], feed_dict = feed)
            train_cost += batch_cost*batch_size
            train_ler += s.run(ler, feed_dict=feed)*batch_size

        if e % 2 == 0:
            show_decoded_seqs(s, Xtrain[cases_to_show],
                              Ytrain[cases_to_show])

        # Shuffle the data
        shuffled_indexes = np.random.permutation(num_examples)
        Xtrain = Xtrain[shuffled_indexes]
        Ytrain = Ytrain[shuffled_indexes]

        # Metrics mean
        train_cost /= num_examples
        train_ler /= num_examples

        log = "%2d/%2d, tr_cost = %.3f, tr_ler = %.3f"
        print(log % (e+1, num_epochs, train_cost, train_ler))

    saver.save(s, "/tmp/ctc_model")
```

```
        0 [3 9 1 6 8 4] --> []
        1 [7 4 6 3 0] --> []
        2 [3 1 6 5 9 1 3 7] --> []
        3 [8 4 9 8 3 5] --> []
        4 [6 3 7 8 9] --> []
        5 [3 9 6] --> []
        6 [8 4 2 5 6 0 9] --> []
        7 [2 9 1 2 6 8 3 7] --> []
        8 [7 2 6 8 5 5] --> []
        9 [0 2 4 6 7] --> []
 1/10, tr_cost = 15.391, tr_ler = 0.996
 2/10, tr_cost = 13.369, tr_ler = 0.996
        0 [5 6 1 7 2] --> [5]
        1 [5 7 8 4 5] --> [5 5]
        2 [3 0 9 7] --> []
        3 [8 2 1 5] --> [5]
        4 [0 2 6 9 3 5 4] --> [5]
        5 [0 5 2 8 6] --> [5]
        6 [8 5 2 0] --> [5]
        7 [4 9 8 2 7 6] --> []
        8 [1 3 5 0 2] --> [7 5]
        9 [2 4 3 7] --> [7]
 3/10, tr_cost = 12.652, tr_ler = 0.934
 4/10, tr_cost = 10.840, tr_ler = 0.788
        0 [9 0 1 3 9 6 0] --> [9 1 3 9 6]
        1 [7 9 0] --> [7 9 0]
        2 [7 6 1 5 6] --> [7 6 1 5 6]
        3 [7 3 4] --> [7 9]
        4 [4 0 6 9 7 5 8] --> [6 9 7 5 8]
        5 [0 3 4 5 8 6] --> [8 5 6 6]
        6 [2 3 7 6 1] --> [2 2 7 6 1]
        7 [4 1 0 9] --> [1 9]
        8 [0 1 2 7 3] --> [1 2 7 3]
        9 [9 1 7 5] --> [9 7 7 5]
 5/10, tr_cost = 6.722, tr_ler = 0.426
 6/10, tr_cost = 2.352, tr_ler = 0.080
        0 [8 9 1 5 6 4 7] --> [8 9 1 5 6 4 7]
        1 [1 2 4] --> [1 1 2 4]
        2 [6 2 3 2 5 8] --> [6 2 3 2 5 8]
        3 [3 9 2 8] --> [3 9 2 8]
        4 [9 7 5 2 0 3] --> [9 7 7 5 2 0 3]
        5 [0 9 2] --> [0 9 2]
        6 [8 5 0 7 4] --> [8 5 0 7 4]
        7 [6 2 0 4] --> [6 2 0 4]
        8 [3 8 2 7 9 4] --> [3 8 2 7 9 4]
        9 [9 2 7 8 4 6] --> [9 2 7 8 4 6]
 7/10, tr_cost = 1.196, tr_ler = 0.039
 8/10, tr_cost = 0.695, tr_ler = 0.023
        0 [1 5 9 0] --> [1 5 9 0]
        1 [7 9 0] --> [7 9 0]
        2 [1 5 0 7 9 8 4] --> [1 5 0 7 9 8 4]
        3 [5 2 9 0 4 3] --> [5 2 9 0 4 3]
        4 [0 7 9 8 2] --> [0 7 9 8 2]
        5 [9 4 3 4 2] --> [9 4 3 4 2]
        6 [8 9 6 3 5] --> [8 9 6 3 5]
        7 [5 2 7 1] --> [5 2 7 1]
        8 [2 7 1 6] --> [2 7 1 6]
        9 [9 0] --> [9 0]
 9/10, tr_cost = 0.523, tr_ler = 0.014
10/10, tr_cost = 0.487, tr_ler = 0.013
```

In [17]:

```
# Evaluate on test set
ftest = 'data/test_varlen.pkl'
Xtest, Ytest, num_instances = get_toy_data(ftest)
# Padding input to max_time_step of this batch
Xtest_p, seqlen_test_p = pad_sequences(Xtest)
# Converting to sparse representation so as to to feed SparseTensor input
Ytest_s = sparse_tuple_from(Ytest)
# test cases
cases_to_show = np.random.randint(0, num_instances - 1, (10,))

with tf.Session() as s:
    saver.restore(s, "/tmp/ctc_model")
    test_ler = s.run(ler, feed_dict={inputs: Xtest_p,
                                     targets: Ytest_s,
                                     seq_len: seqlen_test_p})
    print ('Evaluation on test set. Error:', test_ler)
    show_decoded_seqs(s, Xtest[cases_to_show],
                      Ytest[cases_to_show])
```

```
INFO:tensorflow:Restoring parameters from /tmp/ctc_model
Evaluation on test set. Error: 0.0236786
        0 [3 5 0 7 6] --> [3 5 0 7 6]
        1 [1 2 7 8 3 5] --> [1 2 7 8 3 5]
        2 [5 7 2 4 3 8] --> [5 7 2 4 8 8]
        3 [3 0 1 9 7] --> [3 0 1 9 7]
        4 [0 7 5] --> [0 7 5]
        5 [3 1 6 6] --> [3 1 6 6]
        6 [4 6 1 6 2 3 9] --> [4 6 1 6 2 3 9]
        7 [4 1 0] --> [4 1 0]
        8 [2 4 1 7 3 6] --> [2 4 1 7 3 6]
        9 [0 5 3 1 4 0] --> [0 5 3 1 4 0]
```