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)
```

Generating names with recurrent neural networks

This time you'll find yourself delving into the heart (and other intestines) of recurrent neural networks on a class of toy problems.

Struggle to find a name for the variable? Let's see how you'll come up with a name for your son/daughter. Surely no human has expertize over what is a good child name, so let us train RNN instead;

It's dangerous to go alone, take these:

In [3]:

```
import tensorflow as tf
print(tf.__version__)
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import os
import sys
sys.path.append("..")
import keras_utils
import tqdm_utils
```

```
1.14.0
```

```
Using TensorFlow backend.
```

Load data

The dataset contains ~8k earthling names from different cultures, all in latin transcript.

This notebook has been designed so as to allow you to quickly swap names for something similar: deep learning article titles, IKEA furniture, pokemon names, etc.

In [4]:

```
start_token = " "  # so that the network knows that we're generating a first token

# this is the token for padding,
# we will add fake pad token at the end of names
# to make them of equal size for further batching
pad_token = "#"

with open("names") as f:
    names = f.read()[:-1].split('\n')
    names = [start_token + name for name in names]
```

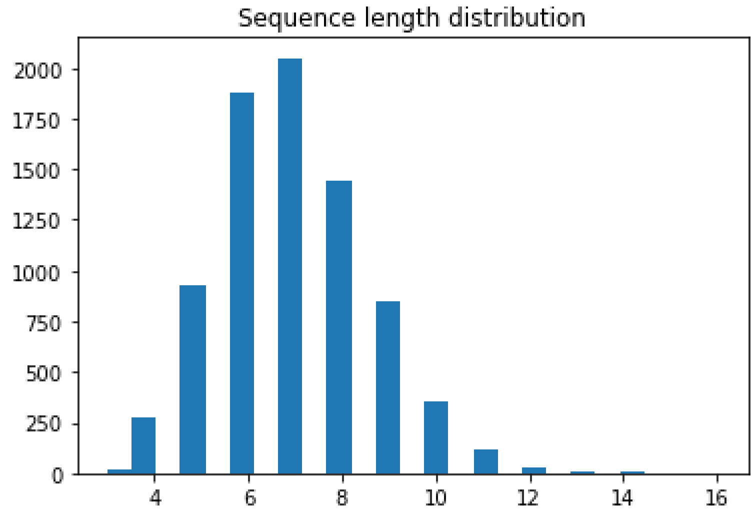
In [5]:

```
print('number of samples:', len(names))
for x in names[::1000]:
    print(x)
```

```
number of samples: 7944
 Abagael
 Claresta
 Glory
 Liliane
 Prissie
 Geeta
 Giovanne
 Piggy
```

In [6]:
```python
MAX_LENGTH = max(map(len, names))
print("max length:", MAX_LENGTH)

plt.title('Sequence length distribution')
plt.hist(list(map(len, names)), bins=25);
```

```
max length: 16
```



Text processing

First we need to collect a "vocabulary" of all unique tokens i.e. unique characters. We can then encode inputs as a sequence of character ids.

In [7]:
```python
### YOUR CODE HERE: all unique characters go here, padding included!
tokens = sorted(set(''.join(names + [start_token, pad_token])))

tokens = list(tokens)
n_tokens = len(tokens)
print ('n_tokens:', n_tokens)

assert 50 < n_tokens < 60
```

```
n_tokens: 56
```

Cast everything from symbols into identifiers

Tensorflow string manipulation is a bit tricky, so we'll work around it. We'll feed our recurrent neural network with ids of characters from our dictionary.

To create such dictionary, let's assign `token_to_id`

In [8]:
```python
### YOUR CODE HERE: create a dictionary of {symbol -> its  index in tokens}
token_to_id = {v:k for k, v in enumerate(tokens)}

assert len(tokens) == len(token_to_id), "dictionaries must have same size"
```

In [9]:
```python
token_to_id
```

Out[9]:
```
{' ': 0,
 '#': 1,
 "'": 2,
 '-': 3,
 'A': 4,
 'B': 5,
 'C': 6,
 'D': 7,
 'E': 8,
 'F': 9,
 'G': 10,
 'H': 11,
 'I': 12,
 'J': 13,
 'K': 14,
 'L': 15,
 'M': 16,
 'N': 17,
 'O': 18,
```

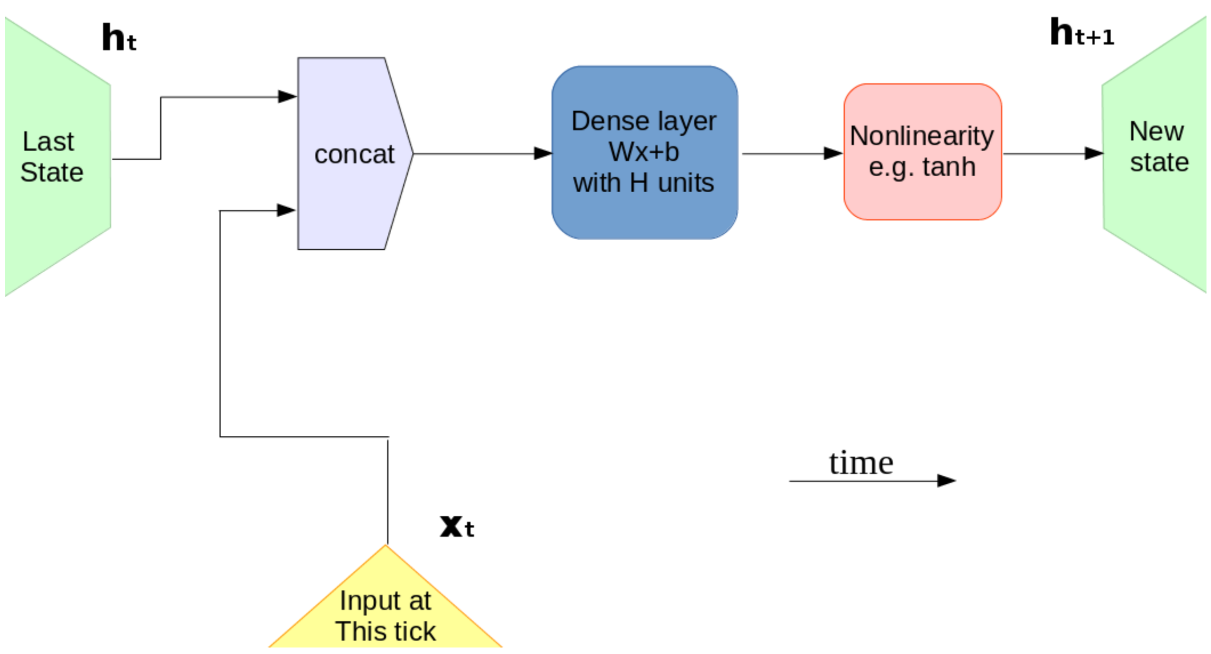
In [10]:
```python
def to_matrix(names, max_len=None, pad=token_to_id[pad_token], dtype=np.int32):
    """Casts a list of names into rnn-digestable padded matrix"""

    max_len = max_len or max(map(len, names))
    names_ix = np.zeros([len(names), max_len], dtype) + pad

    for i in range(len(names)):
        name_ix = list(map(token_to_id.get, names[i]))
        names_ix[i, :len(name_ix)] = name_ix

    return names_ix
```

In [11]:
```python
# Example: cast 4 random names to padded matrices (so that we can easily batch them)
print('\n'.join(names[::2000]))
print(to_matrix(names[:2000]))
```

```
 Abagael
 Glory
 Prissie
 Giovanne
[[ 0  4 31 ...  1  1  1]
 [ 0  4 31 ...  1  1  1]
 [ 0  4 31 ...  1  1  1]
 ...
 [ 0 10 41 ...  1  1  1]
 [ 0 10 41 ...  1  1  1]
 [ 0 10 41 ...  1  1  1]]
```

Defining a recurrent neural network

We can rewrite recurrent neural network as a consecutive application of dense layer to input x_t and previous rnn state h_t. This is exactly what we're gonna do now.



Since we're training a language model, there should also be:

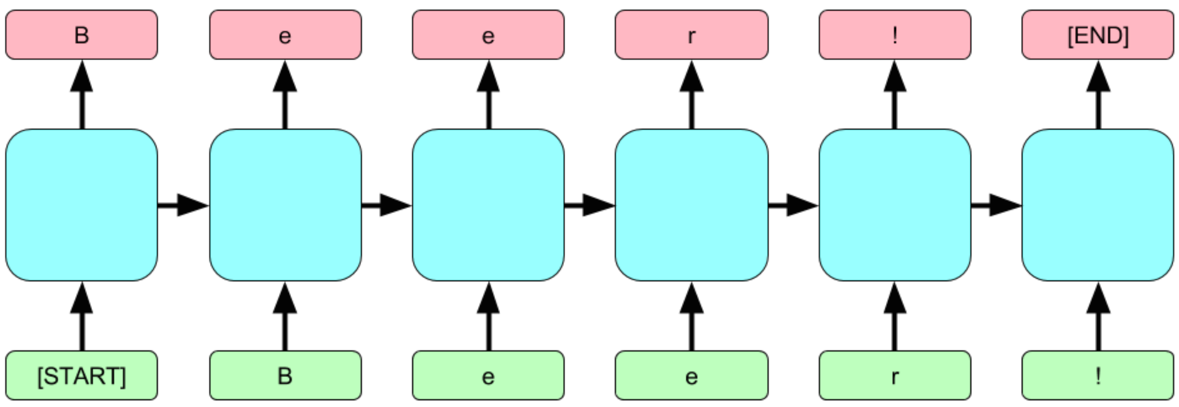
- An embedding layer that converts character id x_t to a vector.
- An output layer that predicts probabilities of next phoneme based on h_t+1

In [12]:
```python
# remember to reset your session if you change your graph!
s = keras_utils.reset_tf_session()
```

```
WARNING:tensorflow:From ..\keras_utils.py:68: The name tf.get_default_session is deprecated. Please use tf.compat.v1.ge
t_default_session instead.

WARNING:tensorflow:From ..\keras_utils.py:75: The name tf.ConfigProto is deprecated. Please use tf.compat.v1.ConfigProt
o instead.

WARNING:tensorflow:From ..\keras_utils.py:77: The name tf.InteractiveSession is deprecated. Please use tf.compat.v1.Int
eractiveSession instead.
```

In [13]:
```python
import keras
from keras.layers import concatenate, Dense, Embedding

rnn_num_units = 64  # size of hidden state
embedding_size = 16  # for characters

# Let's create layers for our recurrent network
# Note: we create layers but we don't "apply" them yet (this is a "functional API" of Keras)
# Note: set the correct activation (from keras.activations) to Dense layers!

# an embedding layer that converts character ids into embeddings
embed_x = Embedding(n_tokens, embedding_size)

# a dense layer that maps input and previous state to new hidden state, [x_t,h_t]->h_t+1
### YOUR CODE HERE
get_h_next = Dense(rnn_num_units, activation='tanh')

# a dense layer that maps current hidden state to probabilities of characters [h_t+1]->P(x_t+1|h_t+1)
### YOUR CODE HERE
get_probas = Dense(n_tokens, activation='softmax')
```

We will generate names character by character starting with `start_token`:



In [14]:
```python
def rnn_one_step(x_t, h_t):
    """
    Recurrent neural network step that produces
    probabilities for next token x_t+1 and next state h_t+1
    given current input x_t and previous state h_t.
    We'll call this method repeatedly to produce the whole sequence.

    You're supposed to "apply" above layers to produce new tensors.
    Follow inline instructions to complete the function.
    """
    # convert character id into embedding
    x_t_emb = embed_x(tf.reshape(x_t, [-1, 1]))[:, 0]

    # concatenate x_t embedding and previous h_t state
    ### YOUR CODE HERE
    x_and_h = concatenate([x_t_emb, h_t], axis=-1)

    # compute next state given x_and_h
    ### YOUR CODE HERE
    h_next = get_h_next(x_and_h)

    # get probabilities for language model P(x_next|h_next)
    ### YOUR CODE HERE
    output_probas = get_probas(h_next)

    return output_probas, h_next
```

RNN: loop

Once `rnn_one_step` is ready, let's apply it in a loop over name characters to get predictions.

Let's assume that all names are at most length-16 for now, so we can simply iterate over them in a for loop.

In [15]:

```
input_sequence = tf.placeholder(tf.int32, (None, MAX_LENGTH))  # batch of token ids
batch_size = tf.shape(input_sequence)[0]

predicted_probas = []
h_prev = tf.zeros([batch_size, rnn_num_units])  # initial hidden state

for t in range(MAX_LENGTH):
    x_t = input_sequence[:, t]  # column t
    probas_next, h_next = rnn_one_step(x_t, h_prev)

    h_prev = h_next
    predicted_probas.append(probas_next)

# combine predicted_probas into [batch, time, n_tokens] tensor
predicted_probas = tf.transpose(tf.stack(predicted_probas), [1, 0, 2])

# next to last token prediction is not needed
predicted_probas = predicted_probas[:, :-1, :]
```

RNN: loss and gradients

Let's gather a matrix of predictions for $P(x_{next}|h)$ and the corresponding correct answers.

We will flatten our matrices to shape [None, n_tokens] to make it easier.

Our network can then be trained by minimizing crossentropy between predicted probabilities and those answers.

In [16]:

```
# flatten predictions to [batch*time, n_tokens]
predictions_matrix = tf.reshape(predicted_probas, [-1, n_tokens])

# flatten answers (next tokens) and one-hot encode them
answers_matrix = tf.one_hot(tf.reshape(input_sequence[:, 1:], [-1]), n_tokens)
```

Usually it's a good idea to ignore gradients of loss for padding token predictions.

Because we don't care about further prediction after the pad_token is predicted for the first time, so it doesn't make sense to punish our network after the pad_token is predicted.

For simplicity you can ignore this comment, it's up to you.

In [17]:

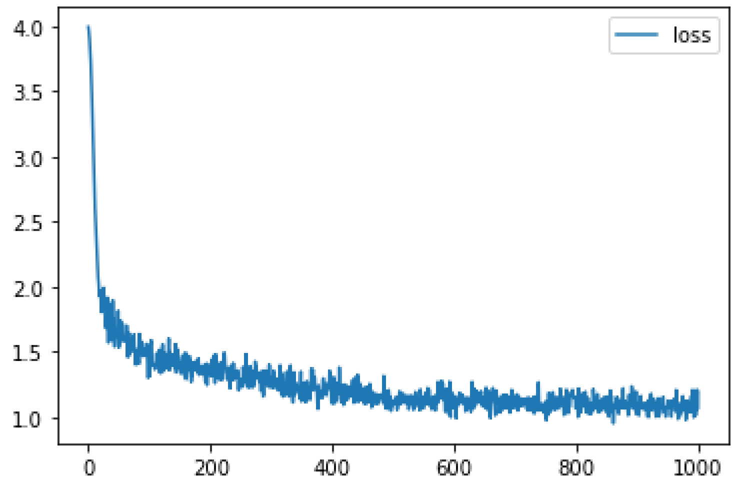
```
# Define the loss as categorical cross-entropy (e.g. from keras.losses).
# Mind that predictions are probabilities and NOT logits!
# Remember to apply tf.reduce_mean to get a scalar loss!
### YOUR CODE HERE
loss = tf.reduce_mean(tf.keras.losses.categorical_crossentropy(answers_matrix, predictions_matrix))

optimize = tf.train.AdamOptimizer().minimize(loss)
```

```
WARNING:tensorflow:From C:\Users\Xiaowei\Anaconda3\envs\tfspark\lib\site-packages\tensorflow\python\ops\math_grad.py:12
50: 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
```

RNN: training

In [18]:

```
from IPython.display import clear_output
from random import sample

s.run(tf.global_variables_initializer())

batch_size = 32
history = []

for i in range(1000):
    batch = to_matrix(sample(names, batch_size), max_len=MAX_LENGTH)
    loss_i, _ = s.run([loss, optimize], {input_sequence: batch})

    history.append(loss_i)

    if (i + 1) % 100 == 0:
        clear_output(True)
        plt.plot(history, label='loss')
        plt.legend()
        plt.show()

assert np.mean(history[:10]) > np.mean(history[-10:]), "RNN didn't converge"
```



RNN: sampling

Once we've trained our network a bit, let's get to actually generating stuff. All we need is the `rnn_one_step` function you have written above.

In [19]:

```
x_t = tf.placeholder(tf.int32, (1,))
h_t = tf.Variable(np.zeros([1, rnn_num_units], np.float32))  # we will update hidden state in this variable

# For sampling we need to define `rnn_one_step` tensors only once in our graph.
# We reuse all parameters thanks to functional API usage.
# Then we can feed appropriate tensor values using feed_dict in a loop.
# Note how different it is from training stage, where we had to unroll the whole sequence for backprop.
next_probs, next_h = rnn_one_step(x_t, h_t)
```

In [20]:

```
def generate_sample(seed_phrase=start_token, max_length=MAX_LENGTH):
    '''
    This function generates text given a `seed_phrase` as a seed.
    Remember to include start_token in seed phrase!
    Parameter `max_length` is used to set the number of characters in prediction.
    '''
    x_sequence = [token_to_id[token] for token in seed_phrase]
    s.run(tf.assign(h_t, h_t.initial_value))

    # feed the seed phrase, if any
    for ix in x_sequence[:-1]:
         s.run(tf.assign(h_t, next_h), {x_t: [ix]})

    # start generating
    for _ in range(max_length-len(seed_phrase)):
        x_probs,_ = s.run([next_probs, tf.assign(h_t, next_h)], {x_t: [x_sequence[-1]]})
        x_sequence.append(np.random.choice(n_tokens, p=x_probs[0]))

    return ''.join([tokens[ix] for ix in x_sequence if tokens[ix] != pad_token])
```

In [21]:
```python
# without prefix
for _ in range(10):
    print(generate_sample())
```

```
Lhanla
Delse
Aleree
Amrin
Asdiy
Larrea
Cylxep
Sribertir
Andie
Kabyr
```

In [22]:
```python
# with prefix conditioning
for _ in range(10):
    print(generate_sample(' Trump'))
```

```
Trumpadno
Trumpa
Trumpis
Trumpie
Trumpa
Trumpenda
Trumpie
Trumpand
Trumpada
Trumpar
```

Submit to Coursera

In [23]:
```python
# token expires every 30 min
COURSERA_TOKEN = "e2f2dGXhTPjAMzyJ"
COURSERA_EMAIL = "lxwvictor@gmail.com"
```

In [24]:
```python
from submit import submit_char_rnn
samples = [generate_sample(' Al') for i in tqdm_utils.tqdm_notebook_failsafe(range(25))]
submission = (history, samples)
submit_char_rnn(submission, COURSERA_EMAIL, COURSERA_TOKEN)
```

100% 25/25 [00:19<00:00, 1.29it/s]

```
You used an invalid email or your token may have expired. Please make sure you have entered all fields correctly. Try g
enerating a new token if the issue still persists.
```

Try it out!

Disclaimer: This part of assignment is entirely optional. You won't receive bonus points for it. However, it's a fun thing to do. Please share your results on course forums.

You've just implemented a recurrent language model that can be tasked with generating any kind of sequence, so there's plenty of data you can try it on:

- Novels/poems/songs of your favorite author
- News titles/clickbait titles
- Source code of Linux or Tensorflow
- Molecules in [smiles (https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system)](https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system) format
- Melody in notes/chords format
- IKEA catalog titles
- Pokemon names
- Cards from Magic, the Gathering / Hearthstone

If you're willing to give it a try, here's what you wanna look at:

- Current data format is a sequence of lines, so a novel can be formatted as a list of sentences. Alternatively, you can change data preprocessing altogether.
- While some datasets are readily available, others can only be scraped from the web. Try `Selenium` or `Scrapy` for that.
- Make sure MAX_LENGTH is adjusted for longer datasets. There's also a bonus section about dynamic RNNs at the bottom.
- More complex tasks require larger RNN architecture, try more neurons or several layers. It would also require more training iterations.
- Long-term dependencies in music, novels or molecules are better handled with LSTM or GRU

Good hunting!

Bonus level: dynamic RNNs

Apart from Keras, there's also a friendly TensorFlow API for recurrent neural nets. It's based around the symbolic loop function (aka tf.scan (https://www.tensorflow.org/api_docs/python/tf/scan)).

RNN loop that we implemented for training can be replaced with single TensorFlow instruction: tf.nn.dynamic_rnn (https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn). This interface allows for dynamic sequence length and comes with some pre-implemented architectures.

Take a look at tf.nn.rnn_cell.BasicRNNCell (https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/BasicRNNCell).

In [ ]:

```
class CustomRNN(tf.nn.rnn_cell.BasicRNNCell):
    def call(self, input, state):
        # from docs:
        # Returns:
        # Output: A 2-D tensor with shape [batch_size, self.output_size].
        # New state: Either a single 2-D tensor, or a tuple of tensors matching the arity and shapes of state.
        return rnn_one_step(input[:, 0], state)

    @property
    def output_size(self):
        return n_tokens

cell = CustomRNN(rnn_num_units)

input_sequence = tf.placeholder(tf.float32, (None, None))

predicted_probas, last_state = tf.nn.dynamic_rnn(cell, input_sequence[:, :, None], dtype=tf.float32)

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
with sess.as_default():
    print('LSTM outputs for each step [batch,time,n_tokens]:')
    print(predicted_probas.eval({input_sequence: to_matrix(names[:10], max_len=50)}).shape)
```

Note that we never used MAX_LENGTH in the code above: TF will iterate over however many time-steps you gave it.

You can also use any pre-implemented RNN cell:

In [ ]:

```
for obj in dir(tf.nn.rnn_cell) + dir(tf.contrib.rnn):
    if obj.endswith('Cell'):
        print(obj, end="\t")
```

In [ ]:

```
input_sequence = tf.placeholder(tf.int32, (None, None))

inputs_embedded = embed_x(input_sequence)

# standard cell returns hidden state as output!
cell = tf.nn.rnn_cell.LSTMCell(rnn_num_units)

state_sequence, last_state = tf.nn.dynamic_rnn(cell, inputs_embedded, dtype=tf.float32)

s.run(tf.global_variables_initializer())

print('LSTM hidden state for each step [batch,time,rnn_num_units]:')
print(state_sequence.eval({input_sequence: to_matrix(names[:10], max_len=50)}).shape)
```

In [ ]:

```

```