PROJECT

CHARACTER PREDICTION USING RECURRENT NEURAL NETWORKS

RHEEYA UPPAAL
3rd Year, Computer Science

A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behaviour. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as unsegmented connected handwriting recognition or speech recognition.

A glaring limitation of Vanilla Neural Networks (and also Convolutional Networks) is that their API is too constrained: they accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes). Not only that:

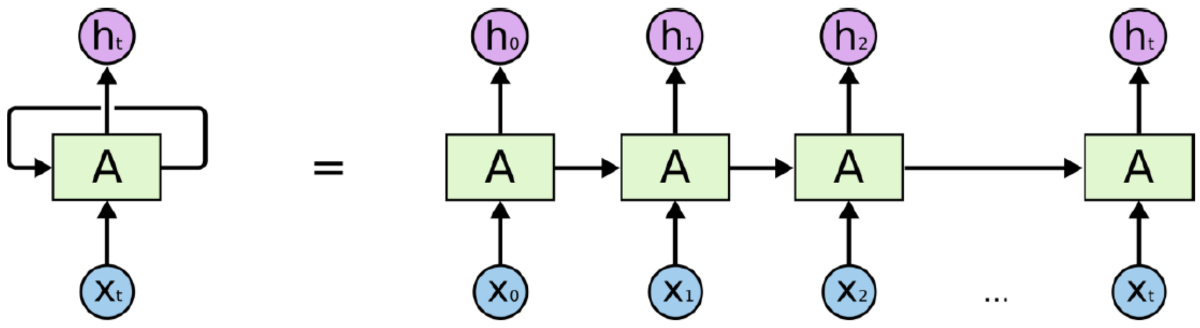

Fig 5.2 Schematic structure of an RNN. The image on the right shows the "unrolling" of an RNN, i.e. the sequential nature of an RNN allows it to behave like a series of regular feed-forward neural networks with varying inputs.

These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model). The core reason that recurrent nets are more exciting is that they allow us to operate over *sequences* of vectors: Sequences in the input, the output, or in the most general case both. RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector.

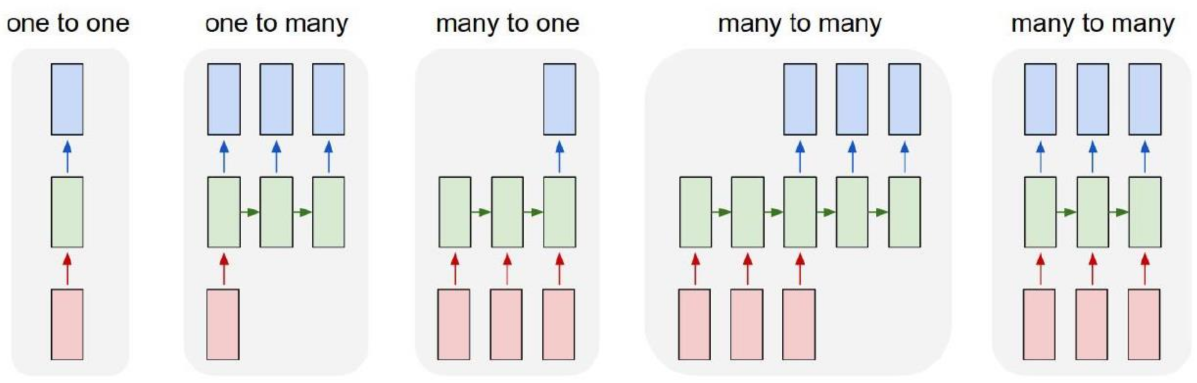

Fig 5.3 Each rectangle is a vector and arrows represent functions like matrix multiply. Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state. In every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.
(1) Processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification).
(2) Sequence output (e.g. image captioning takes an image and outputs a sentence of words).
(3) Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment).
(4) Sequence input and sequence output (e.g. Machine Translation: from English to French).
(5) Synced sequence input and output (e.g. video classification where each frame of the video is to be labelled).

At the core, RNNs have a deceptively simple API: They accept an input vector *x* and give you an output vector *y*. However, crucially this output vector's contents are influenced not only by the input just fed in, but also on the entire history of inputs fed in in the past. Written as a class, the RNN's API consists of a single step function. The RNN class has some internal state that it gets to update every time step is called. In the simplest case this state consists of a single hidden vector *h*. The code below defines the forward pass of a vanilla RNN.

```
class RNN:
        def step(self, x):
                # update the hidden state
                self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
                y = np.dot(self.W_hy, self.h) # compute the output vector
                return y
```

This RNN's parameters are the three matrices *W_hh, W_xh, W_hy*. The hidden state *self.h* is initialized with the zero vector. The *np.tanh* function implements a non-linearity that squashes the activations to the range [-1, 1]. There are two terms inside of the *tanh*: one is based on the previous hidden state and one is based on the current input. *Numpy's np.dot* is matrix multiplication. The two intermediates interact with addition, and then get squashed by the *tanh* into the new state vector. The hidden state update can also be written as: h_t=tanh(W_{hh}ht−1+$W_{xh}x_t$), where *tanh* is applied elementwise.

In this project, an RNN is given a huge chunk of text and it is asked to model the probability distribution of the next character in the sequence given a sequence of previous characters.

As a working example, let a vocabulary consist of four possible letters "helo", be trained on an RNN on the training sequence "hello". This training sequence is a source of 4 separate training examples: 1. The probability of "e" should be likely given the context of "h", 2. "l" should be likely in the context of "he", 3. "l" should also be likely given the context of "hel", and finally 4. "o" should be likely given the context of "hell".

Each character is encoded into a vector using 1-of-k encoding (i.e. all zero except for a single one at the index of the character in the vocabulary), and fed into the RNN one at a time with the step function. A sequence of 4-dimensional output vectors (one dimension per character), which we interpret as the confidence the RNN currently assigns to each character coming next in the sequence is observed.

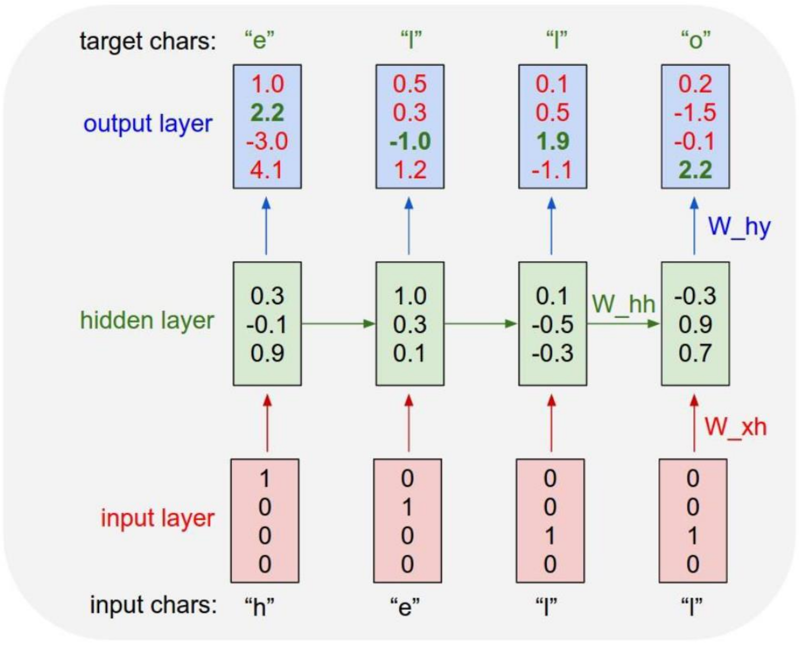

Fig 5.4 An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input. The output layer contains confidences the RNN assigns for the next character.

The green numbers (confidence) should be increased and red numbers (confidence of all other letters) decreased. Once the loss is backpropagated and the RNN's weights are updated, the correct next letters will have higher scores when faced with similar inputs.

The model is defined as a minimal character level vanilla RNN. It consists of one hidden layer with 100 neurons, and is unrolled 25 times. It is given the seven Harry Potter books (1,084,170 words or 4.5 MB of data) as input. The model was trained on a Nvidia GeForce 8400 GPU over a span of two hours. An excerpt of the output is as follows:

"Potsed in it on a meareyy rilly. Dumbfechared out kender Harry Prileever ourdeved in't all on couf on and cat folks and as to the pilly in to"

It is clear that while most "words" created by the model are not part of the English vocabulary, they are similar to the authentic words. Words which frequently apprear in the input data like "Harry" and "Dumbledore" have gained more success. It must be noted here that the character predictions would greatly increase in accuracy if two factors are altered: (1) A more complicated RNN like the Long-Short-Term-Memory (LSTM) is used (2) The model is trained over a longer duration of time. As the training time increases, the RNN undergoes more iterations and the input to the RNN becomes more accurate, resulting in better accuracy output which then leads to more accurate input and so on.

For example, the output from a 3 layer RNN with 512 hidden nodes on each layer, trained on Shakespeare's works predicts characters as follows:

PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

```
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }

# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

def lossFun(inputs, targets, hprev):
  #inputs,targets are both list of integers.  hprev is Hx1 array of
  #initial hidden state returns the loss, gradients on model parameters,
  #and last hidden state
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in xrange(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]

def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in xrange(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes

n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    p = 0 # go from start of data
  inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
  targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```