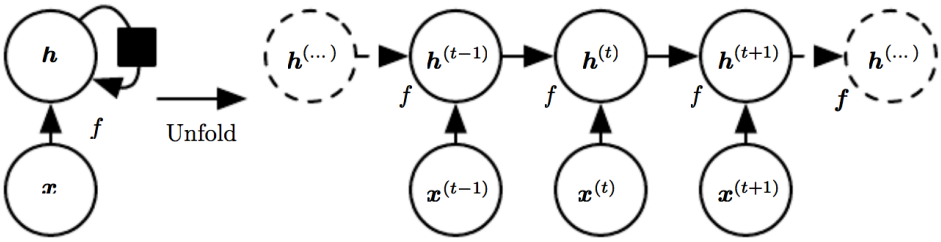
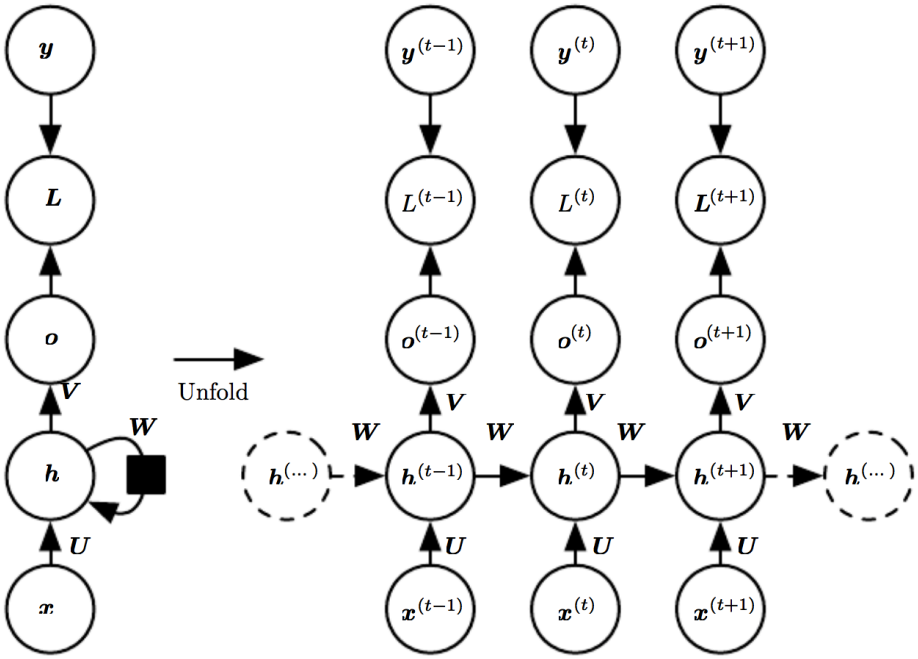
RNN（Recurrent Neural Network）

多层反馈 RNN（Recurrent neural Network、循环神经网络）神经网络是一种节点定向连接成环的人工神经网络。这种网络的内部状态可以展示动态时序行为。不同于前馈神经网络的是，RNN 可以利用它内部的记忆来处理任意时序的输入序列，这让它可以更容易处理如不分段的手写识别、语音识别等。

如下图所示是循环神经网络的典型结构，其中包含一个从隐含层到隐含层的连接（x、h 分别是输入和隐含层），为了更方便我们理解其的工作原理，可将其展开成右边的结构。



RNN 常用于序列数据的标注或预测，例如预测一句话的下一个可能的单词。下图是 RNN 的典型结构，图中 x、o 分别是系统的输入和输出，h 是隐含神经元，y 是训练集中给出的真实值，例如 o 是预测的下一个单词，而 y 是真实出现的下一个单词，L 是损失估计。U、V、W 分别是权重，需要通过训练得到。



我们首先对 RNN 的正向传播（forward propagation）方程进行描述，其中隐含元的激活函数（activation function）也可以有其他选择。

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned} \tag{1}$$

假设损失函数取为负对数似然，则有：

$$\begin{aligned}
&L\left(\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)}\}\right) \\
&= \sum_t L^{(t)} \\
&= -\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\right),
\end{aligned} \tag{2}$$

为了对模型参数进行估计和优化，需要基于反向传播（backward propagation）算法针对损失函数从右至左计算相对各参数的梯度，因此算法的时间和空间复杂度是 O(t)，并称该算法为 back-propagation through time/BPTT。

现在我们推导 RNN 反向传播算法中的梯度。

首先从损失估计节点开始计算如下：

$$\frac{\partial L}{\partial L^{(t)}} = 1.$$

假设损失函数是负对数似然，则相对于输出节点的梯度计算如下：

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i, y^{(t)}}.$$

相对于隐含节点的梯度计算如下：

$$\begin{aligned}
\nabla_{\boldsymbol{h}^{(t)}} L &= (\nabla_{\boldsymbol{h}^{(t+1)}} L) \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} + (\nabla_{\boldsymbol{o}^{(t)}} L) \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \\
&= (\nabla_{\boldsymbol{h}^{(t+1)}} L) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) \boldsymbol{W} + (\nabla_{\boldsymbol{o}^{(t)}} L) \boldsymbol{V}
\end{aligned} \tag{3}$$

相对模型全部参数的梯度计算如下：

$$
\begin{aligned}
\nabla_{\boldsymbol{c}} L &= \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}} = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L \\
\nabla_{\boldsymbol{b}} L &= \sum_t \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}} = \sum_t \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \\
\nabla_{\boldsymbol{V}} L &= \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{V}} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)\top} \\
\nabla_{\boldsymbol{W}} L &= \sum_t \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{W}} = \sum_t \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \boldsymbol{h}^{(t-1)\top} \\
\nabla_{\boldsymbol{U}} L &= \sum_t \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{U}} = \sum_t \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \boldsymbol{x}^{(t)\top}
\end{aligned}
\tag{4}
$$

RNN 的 numpy 实现可以参考这篇文章，其实现了预测句中的下一个词。

http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-2-implementing-a-language-model-rnn-with-python-numpy-and-theano/

核心函数主要有三个：正向传播、损失估计、反向传播，代码如下。

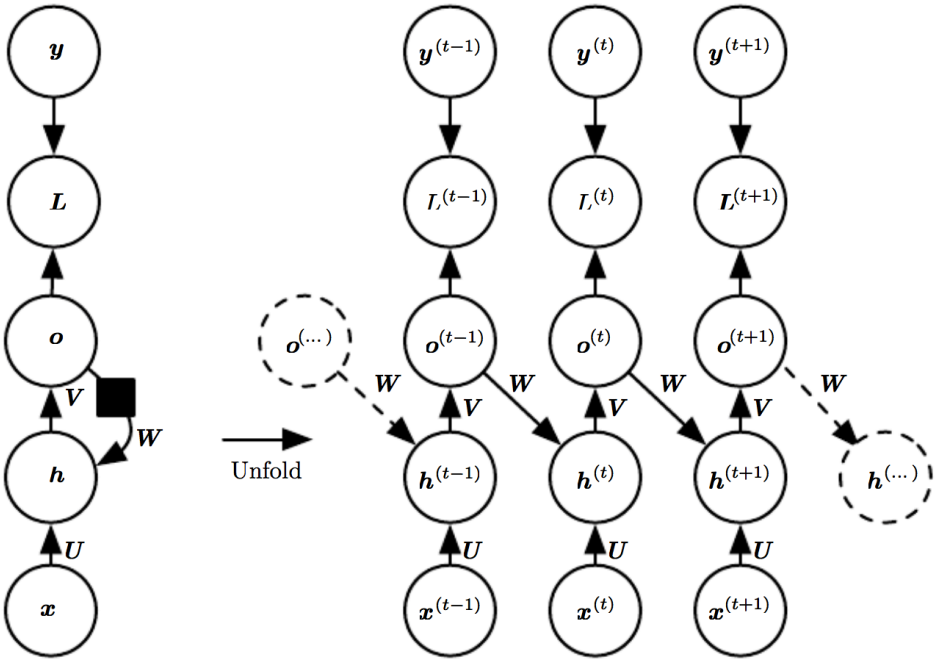
```
def forward_propagation(self, x):
    # The total number of time steps
    T = len(x)
    # During forward propagation we save all hidden states in s because need them later.
    # We add one additional element for the initial hidden, which we set to 0
    s = np.zeros((T + 1, self.hidden_dim))
    s[-1] = np.zeros(self.hidden_dim)
    # The outputs at each time step. Again, we save them for later.
    o = np.zeros((T, self.word_dim))
    # For each time step...
    for t in np.arange(T):
        # Note that we are indxing U by x[t]. This is the same as multiplying U with a one-hot vector.
        s[t] = np.tanh(self.U[:, x[t]] + self.W.dot(s[t - 1]))
        o[t] = softmax(self.V.dot(s[t]))
    return [o, s]
```

```
def calculate_total_loss(self, x, y):
    L = 0
    # For each sentence...
    for i in np.arange(len(y)):
        o, s = self.forward_propagation(x[i])
        # We only care about our prediction of the "correct" words
        correct_word_predictions = o[np.arange(len(y[i])), y[i]]
        # Add to the loss based on how off we were
        L += -1 * np.sum(np.log(correct_word_predictions))
    return L
```

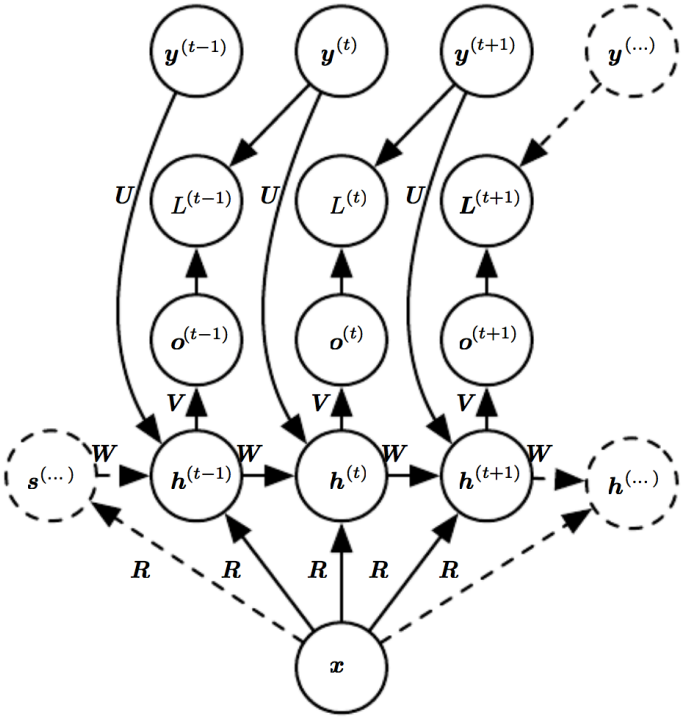
正向传播和损失估计分别对应了公式（1）和（2）。由于 RNN 反向传播中存在梯度消失或爆炸的问题，因此在反向传播算法的实现中对反向传播路径长度进行限制，并借助相邻隐含元 h(t)和 h(t+1)之间的递推公式（3）不断进行递推，对模型参数进行优化，代码实现如下。

```python
def bptt(self, x, y):
    T = len(y)
    # Perform forward propagation
    o, s = self.forward_propagation(x)
    # We accumulate the gradients in these variables
    dLdU = np.zeros(self.U.shape)
    dLdV = np.zeros(self.V.shape)
    dLdW = np.zeros(self.W.shape)
    delta_o = o
    delta_o[np.arange(len(y)), y] -= 1.
    # For each output backwards...
    for t in np.arange(T)[::-1]:
        dLdV += np.outer(delta_o[t], s[t].T)
        # Initial delta calculation
        delta_t = self.V.T.dot(delta_o[t]) * (1 - (s[t] ** 2))
        # Backpropagation through time (for at most self.bptt_truncate steps)
        for bptt_step in np.arange(max(0, t - self.bptt_truncate), t + 1)[::-1]:
            # print "Backpropagation step t=%d bptt step=%d " % (t, bptt_step)
            dLdW += np.outer(delta_t, s[bptt_step - 1])
            dLdU[:, x[bptt_step]] += delta_t
            # Update delta for next step
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step - 1] ** 2)
    return [dLdU, dLdV, dLdW]
```

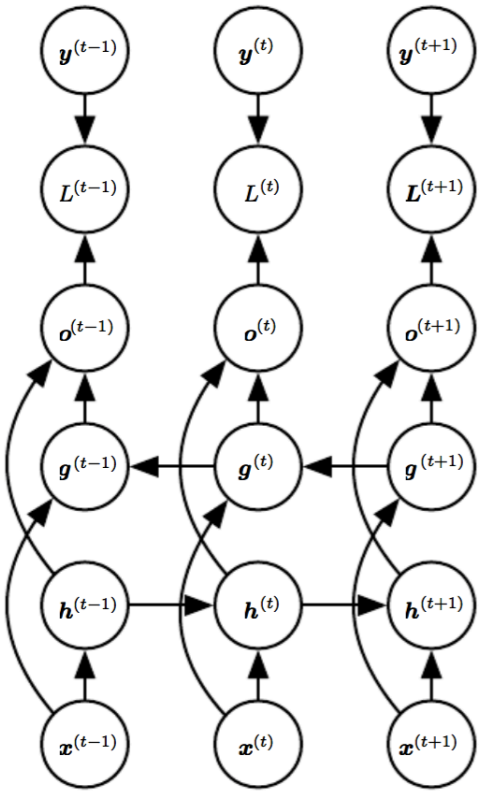
典型的 RNN 除了可以将隐含节点进行连接，也可以将输出节点和隐含节点进行连接，如下图所示。然而由于在递推过程中上一级向下一级的输入被编码为输出向量，因此状态传播收到限制，相比隐含节点之间连接的情况，所能描述的函数空间缩小，但另一方面，由于在训练阶段隐含节点间不存在依赖，因此可以采用 Teacher Forcing 的方法进行并行训练。



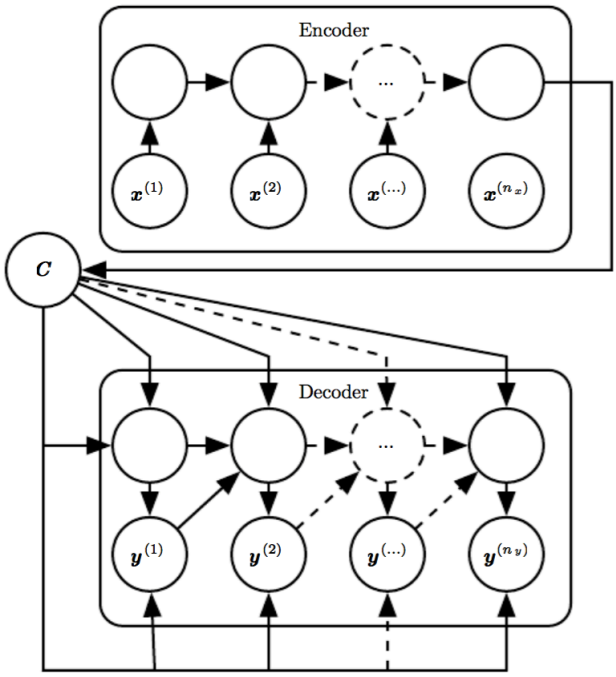
上述 RNN 的输入 x 是一个序列，当 x 是一个定长向量时，可以采用下面的结构，例如可以应用于图像字幕（image captioning）。



上述 RNN 都仅存在单向的“因果”链，如果需要考虑双向的“因果”链，还可采用双向 RNN 结构（Bidirectional RNN），即改变上面预测下一个词的例子为预测一句话中填空中的一个词，此时可以同时利用该填空的前一部分和后一部分单词序列同时进行估计。双向 RNN 结构结构如下。



采用 RNN 还可以实现序列到序列的编解码器结构（Encoder-Decoder Sequence-to-Sequence Architecture）。一个例子是实现字符串的加法，代码参见 Keras 提供的 example：addition_rnn.py。例如输入"535+61"，输出"596"。序列的编解码器结构如下。

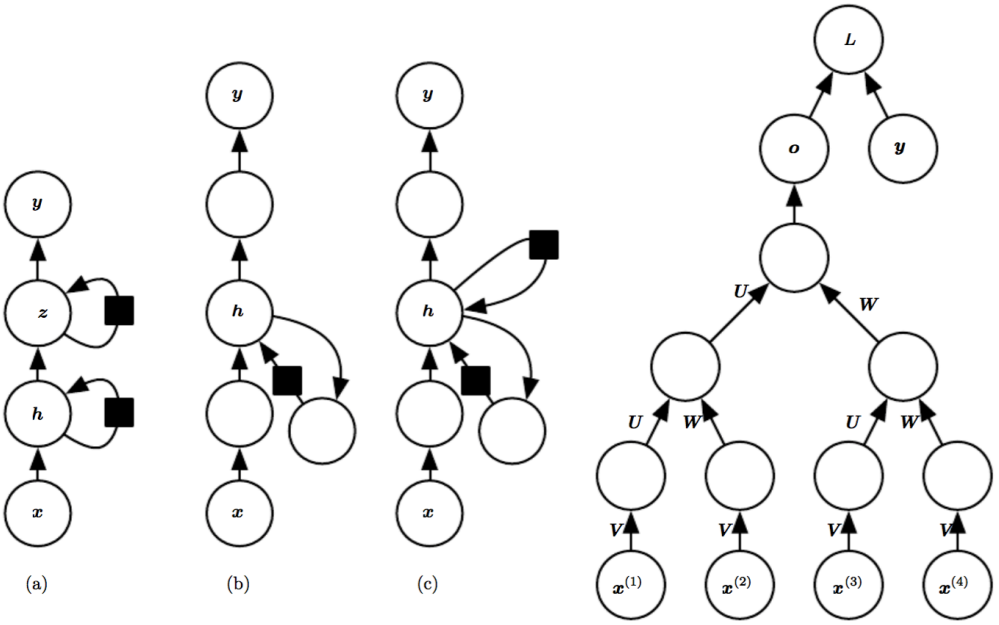


代码中编码器为一层 RNN，最后的隐含节点的输出对应图中的 C，然后复制 DIGIT+1 次作为解码器的输入，解码器包含 LAYERS 层 RNN，最后叠加一层全连接作为输出。

```
model = Sequential()
# "Encode" the input sequence using an RNN, producing an output of HIDDEN_SIZE
# note: in a situation where your input sequences have a variable length,
# use input_shape=(None, nb_feature).
model.add(RNN(HIDDEN_SIZE, input_shape=(MAXLEN, len(chars))))
# For the decoder's input, we repeat the encoded input for each time step
model.add(RepeatVector(DIGITS + 1))
# The decoder RNN could be multiple layers stacked or a single layer
for _ in range(LAYERS):
    model.add(RNN(HIDDEN_SIZE, return_sequences=True))

# For each of step of the output sequence, decide which character should be chosen
model.add(TimeDistributedDense(len(chars)))
model.add(Activation('softmax'))
```

除了单层 RNN 结构，常用结构还有深层循环网络和回归神经网络，其结果如下所示。



RNN 中典型问题是长期依赖（Long-Term Dependencies）。RNN 的优势在于能够通过先前的信息来指导当前的任务，例如预测下一个词，句子是"the clouds are in the _"，容易预测出"sky"，该例子中，相关信息和预测词之间的距离很小，然而，在一个更复杂的场景下，例如句子"I grew up in France...I speak fluent _"，为了正确预测出下一个词所需相关信息的跨度非常大，然而 RNN 在计算过程中后面时间节点相对前面时间节点的感知力下降，因此会导致方法失效。为了解决这个问题，已有不少研究提出了不同策略，例如：Echo State Networks、Leaky Units and Multiple Time Scales、以及基于 Gate 的 Long Short Term Memory /LSTM 等。

参考连接

- http://www.deeplearningbook.org/contents/rnn.html
- http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
- https://github.com/mqliang/rnn_python/blob/master/rnn_numpy.py
- https://github.com/fchollet/keras/blob/master/examples/addition_rnn.py