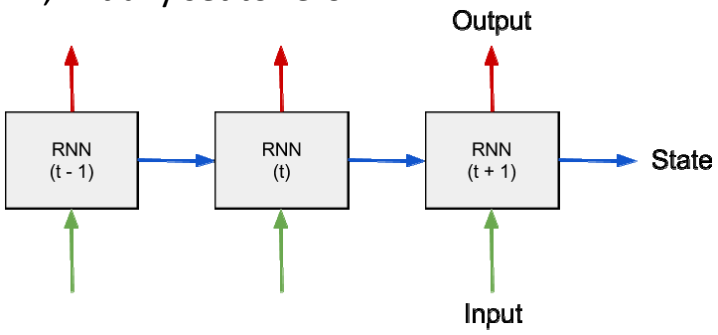
RECURRENT NEURAL NETWORKS (RNNs)

Recurrent Neural Networks

- RNNs can be used when your data is treated as a sequence.
- The most straight-forward example is a time-series of numbers, where the task is to predict the next value given previous values.
- The input to the RNN at every time-step is the *current value* as well as a *state vector* which represent what the network has "seen" at time-steps before. This state-vector is the encoded memory of the RNN, initially set to zero.

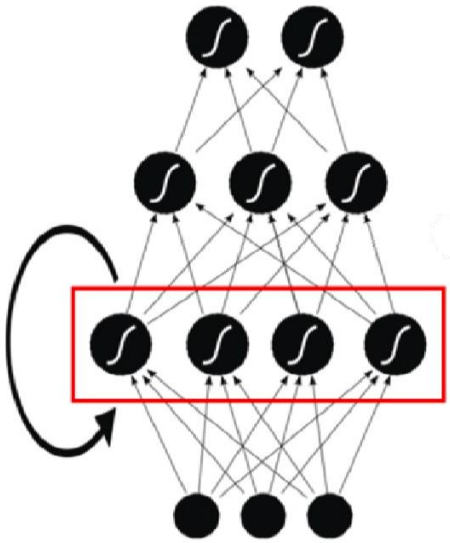


Recurrent Neural Networks

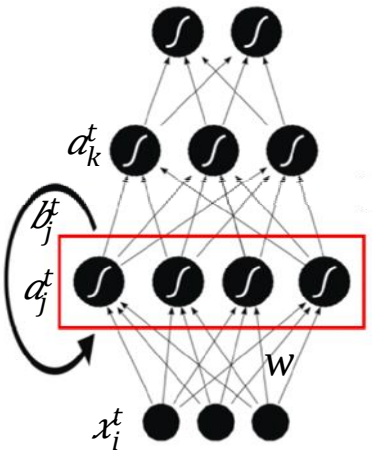
- MLP can only map from input to output vectors
- RNN can map from the entire history of previous inputs to each output.
- An RNN with a sufficient number of hidden units can approximate any measurable sequence-to-sequence mapping to arbitrary accuracy (Hammer, 2000).
  - The key point is that the recurrent connections allow a 'memory' of previous inputs to persist in the network's internal state.

Recurrent Neural Networks

- Forward Pass
  - The forward pass of an RNN is the same as that of a MLP with a single hidden layer.
  - Except that activations arrive at the hidden layer from both the current external input and the hidden layer activations from the previous timestep.



Recurrent Neural Networks



- Consider a length T input sequence, x presented to an RNN with I input units, H hidden units, and K output units.
- Let x_i^t be the value of input i at time t
- a_j^t be the network input to hidden unit j at time t
- b_j^t be the activation output of hidden unit j at time t
- For the hidden units we have

current external input

hidden layer activations from the previous timestep

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1}$$

- Nonlinear, differentiable activation functions are then applied exactly as for an MLP $b_h^t = \theta_h(a_h^t)$

Recurrent Neural Networks

- The initial values b_j^0 are set to zero.
  - However, other researchers have found that RNN stability and performance can be improved by using nonzero initial values.
- The network inputs to the output units can be calculated at the same time as the hidden activations

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t$$

Recurrent Neural Networks

- Backward Pass
  - Given the partial derivatives of some differentiable loss function L with respect to the network outputs,
  - The next step is to determine the derivatives with respect to the weights.
  - Two well-known algorithms have been devised to efficiently calculate weight derivatives for RNNs:
    - real time recurrent learning (RTRL; Robinson and Fallside, 1987)
    - and backpropagation through time (BPTT; Williams and Zipser, 1995; Werbos, 1990).
  - We focus on BPTT since it is both conceptually simpler and more efficient in computation time.

Recurrent Neural Networks

- Like standard BP, BPTT consists of a repeated application of the chain rule.
- The loss function depends on the activation of the hidden layer not only through its influence on the output layer, but also through its influence on the hidden layer at the next timestep

$$\delta_h^t = \theta'(a_h^t)\left(\sum_{k=1}^{K}\delta_k^t w_{hk} + \sum_{h'=1}^{H}\delta_{h'}^{t+1} w_{hh'}\right) \qquad \delta_j^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_j^t}$$

- The complete sequence of δ terms can be calculated by starting at t = T and recursively applying the above equation, decrementing t at each step.

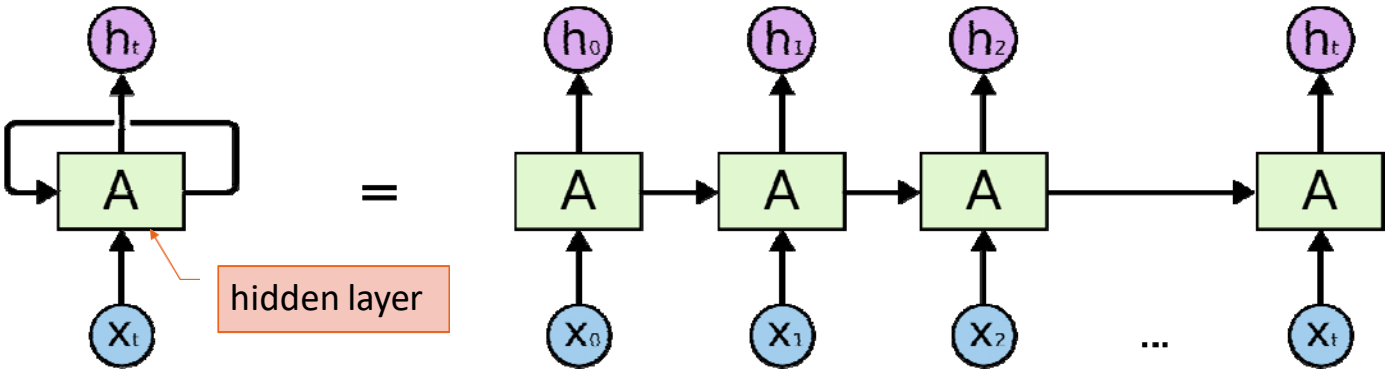
Recurrent Neural Networks

- Finally, bearing in mind that the same weights are reused at every timestep, we sum over the whole sequence to get the derivatives with respect to the network weights:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial a_j^t} \frac{\partial a_j^t}{\partial w_{ij}} = \sum_{t=1}^{T} \delta_j^t b_i^t$$

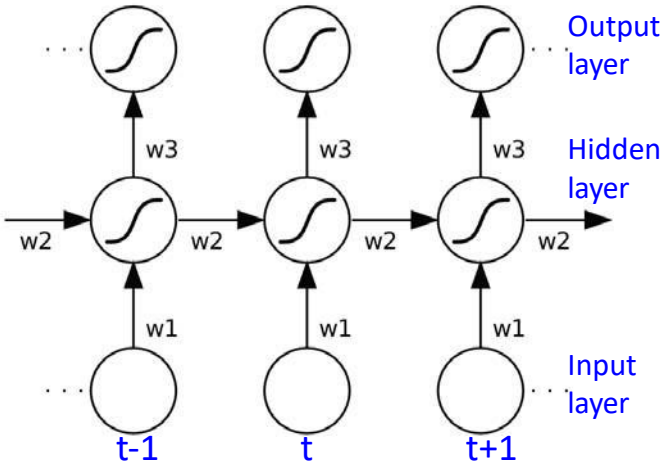
Unfolded recurrent network

- A useful way to visualize RNNs is to consider the update graph formed by 'unfolding' the network along the input sequence.



Unfolded recurrent network

- Each node represents a layer of network units at a single timestep.
- The weighted connections from the input layer to hidden layer are labelled w1, those from the hidden layer to itself (i.e. the recurrent weights) are labelled w2 and the hidden to output weights are labelled w3.
- The same weights are reused at every timestep.
- Bias weights are omitted for clarity.



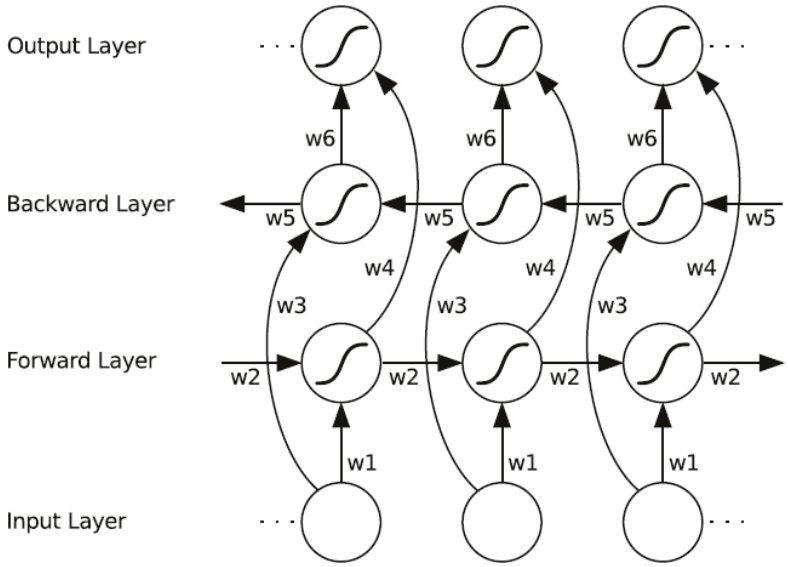
Access to future context

- For many sequence labelling tasks it is beneficial to have access to future as well as past context.
  - For example, when classifying a particular written letter, it is helpful to know the letters coming after it as well as those before.
  - However, since standard RNNs process sequences in temporal order, they ignore future context.

Bidirectional recurrent neural networks

- Bidirectional recurrent neural networks offer a more elegant solution.
- The basic idea of BRNNs is to present each training sequence forwards and backwards to two separate recurrent hidden layers, both of which are connected to the same output layer.
- This structure provides the output layer with complete past and future context for every point in the input sequence, without displacing the inputs from the relevant targets.

Unfolded bidirectional network



Forward pass for the BRNN

- The forward pass for the BRNN hidden layers is the same as for a unidirectional RNN, except that the input sequence is presented in opposite directions to the two hidden layers, and the output layer is not updated until both hidden layers have processed the entire input sequence:

for $t = 1$ to T **do**
  Forward pass for the forward hidden layer, storing activations at each timestep
for $t = T$ to 1 **do**
  Forward pass for the backward hidden layer, storing activations at each timestep
for all t, in any order **do**
  Forward pass for the output layer, using the stored activations from both hidden layers

Backward pass for the BRNN

- The backward pass proceeds as for a standard RNN trained with BPTT, except that all the output layer δ terms are calculated first, then fed back to the two hidden layers in opposite directions:

for all t, in any order **do**
  Backward pass for the output layer, storing δ terms at each timestep
for $t = T$ to 1 **do**
  BPTT backward pass for the forward hidden layer, using the stored δ terms from the output layer
for $t = 1$ to T **do**
  BPTT backward pass for the backward hidden layer, using the stored δ terms from the output layer

Truncated Backpropagation

- In order to make the learning process tractable, it is common practice to create an " unfolded" version of the network, which contains a fixed number (num_steps) of inputs and outputs.
- The model is then trained on this finite approximation of the RNN.
- This can be implemented by feeding inputs of length num_steps at a time and performing a backward pass after each such input block.

Causality for BRNNs

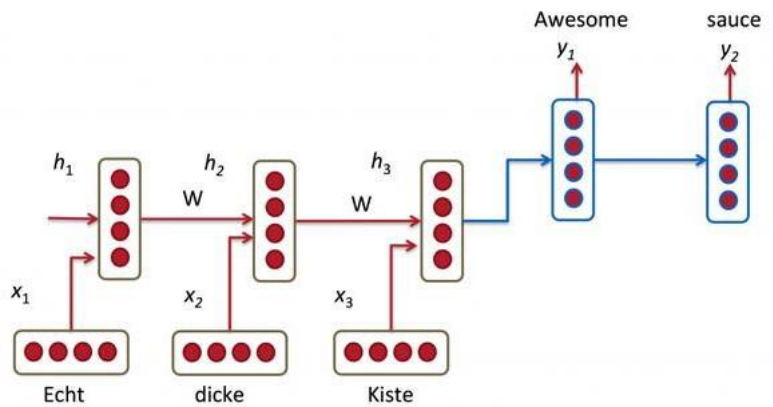
- One objection to bidirectional networks is that they violate causality.
- Clearly, for tasks such as financial prediction or robot navigation, an algorithm that requires access to future inputs is unfeasible.
- However, there are many problems for which causality is unnecessary. Most obviously, if the input sequences are spatial and not temporal there is no reason to distinguish between past and future inputs.
- This is perhaps why protein structure prediction is the domain where BRNNs have been most widely adopted.

Causality for BRNNs

- However BRNNs can also be applied to temporal tasks, as long as the network outputs are only needed at the end of some input segment.
  - For example, in speech and handwriting recognition, the data is usually divided up into sentences, lines, or dialogue turns, each of which is completely processed before the output labelling is required.
  - Furthermore, even for online temporal tasks, such as automatic dictation (聽寫), bidirectional algorithms can be used as long as it is acceptable to wait for some natural break in the input, such as a pause in speech, before processing a section of the data.

Applications

- Language translation
  - No need to have input and output for every step.
  - Output after inputting the entire sentence.



LSTMs (LONG-SHORT-TERM-MEMORIES)

PROPOSED BY HOCHREITER AND SCHMIDHUBER, 1997

Problems in RNNs

- An important benefit of RNNs is their ability to use contextual information when mapping between input and output sequences.
- Problems with simple RNN architectures
  - Vanishing gradient in training
    - Sensitivity to an input at time t decreases rapidly
  - The main drawback of RNN is that it is very difficult to get them to store information for long periods of time.

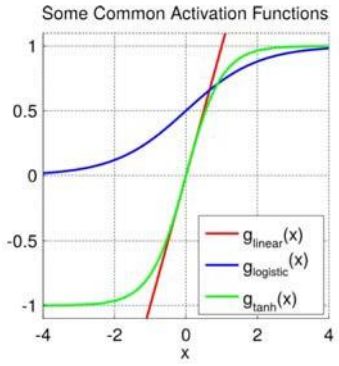
Vanishing gradient problem

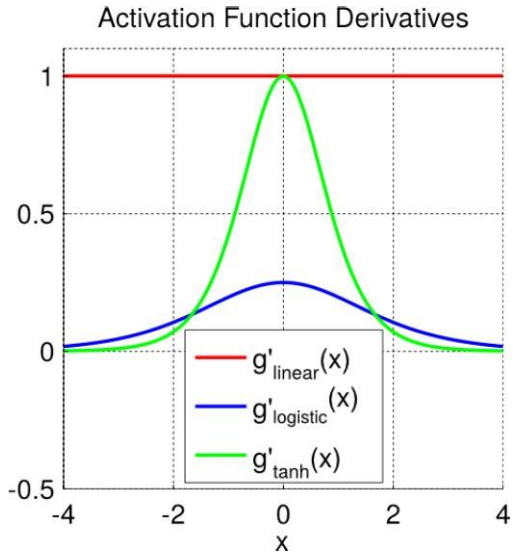
- Unless weights large, error signal will degrade

$$w_{ji}^{(s)}(k+1) = w_{ji}^{(s)}(k) + \mu^{(s)}\delta_j^{(s)}x_{out,i}^{(s-1)} \quad where$$

$$\delta_j^{(s)} = \left(d_{qh} - x_{out,j}^{(s)}\right)g\left(v_j^{(s)}\right) \quad for \quad the \quad output \quad layer$$

$$\delta_j^{(s)} = \left(\sum_{h=1}^{n_{s+1}} \delta_h^{(s+1)} w_{hj}^{(s+1)}\right) g\left(v_j^{(s)}\right) \quad for \quad the \quad hidden \quad layers$$

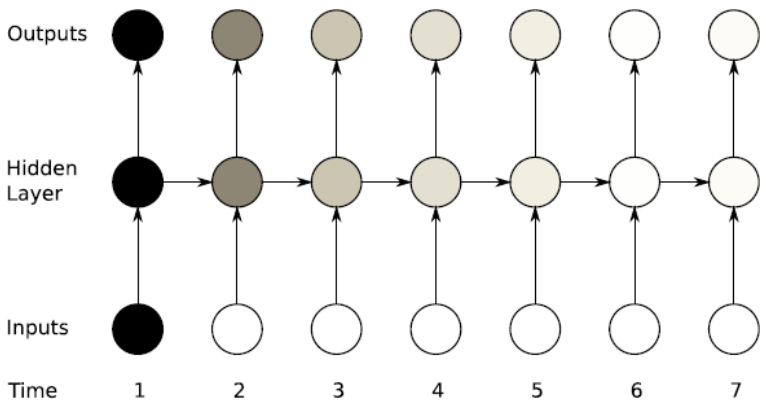




Vanishing gradient problem

- The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity).

The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network 'forgets' the first inputs.

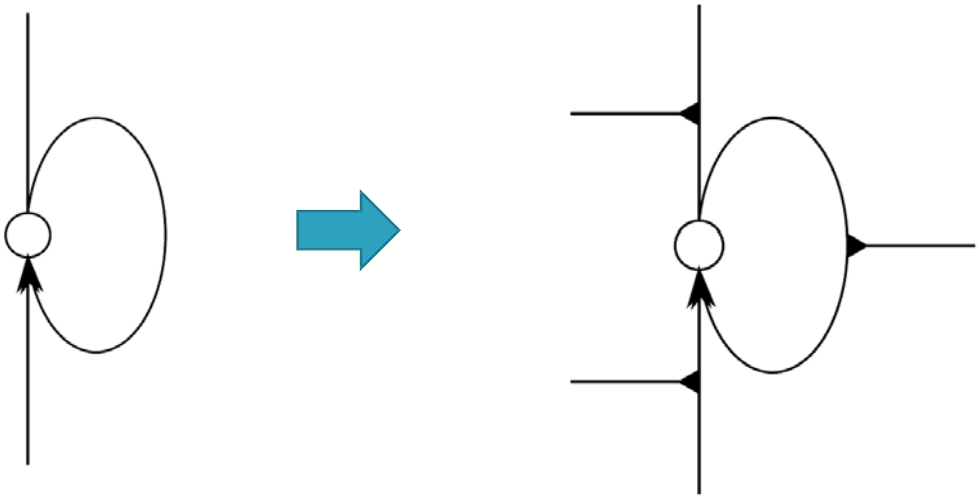


LSTM architecture

- The LSTM architecture consists of a set of recurrently connected subnets, known as memory blocks. (Hochreiter and Schmidhuber, 1997)
- These blocks can be thought of as a differentiable version of the memory chips in a digital computer.
- Each block contains one or more self-connected memory cells and three multiplicative units—the input, output and forget gates—that provide continuous analogues of write, read and reset operations for the cells.

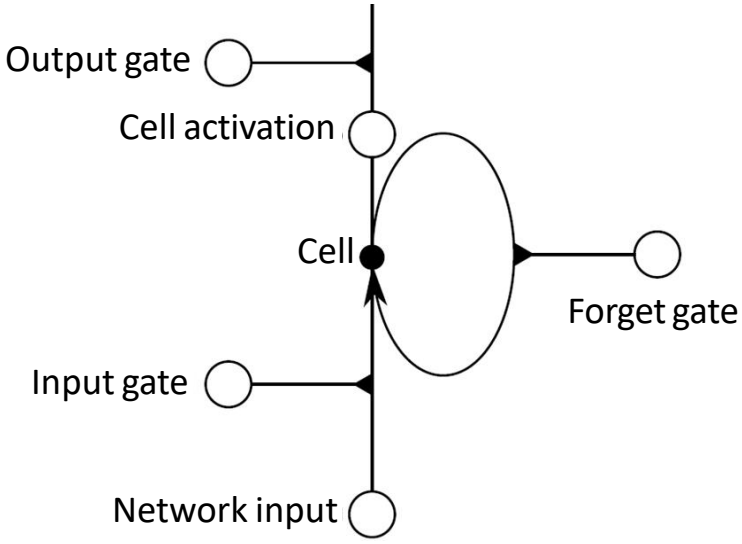
Constructing a memory block

- Start point
- Adding controls



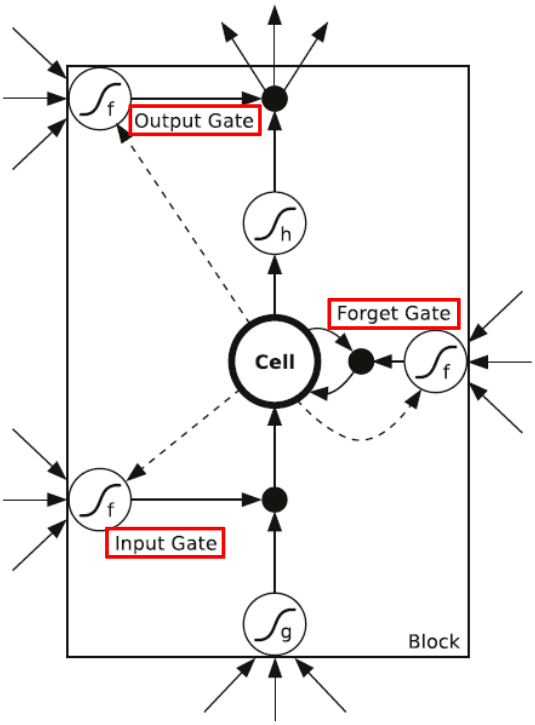
Constructing a memory block

- Let the controls be neurons



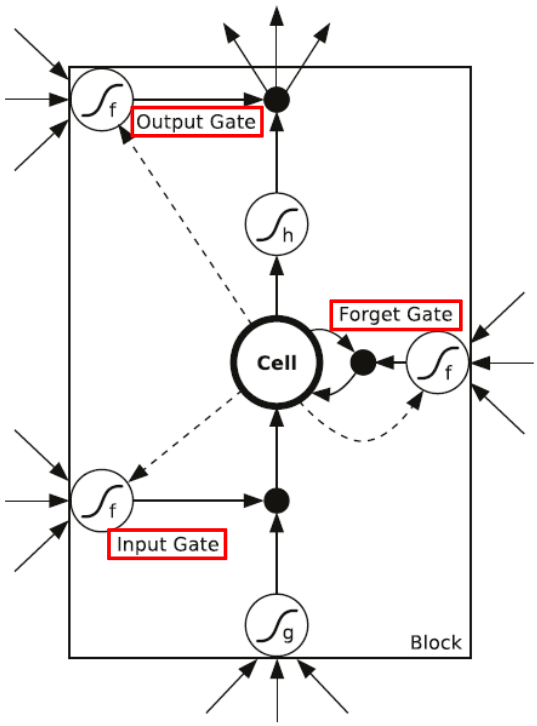
LSTM memory block

- The three gates are nonlinear summation units that collect activations from inside and outside the block, and control the activation of the cell via multiplications (**small black circles**).
- The input and output gates multiply the input and output of the cell while the forget gate multiplies the cell's previous state.
- No activation function is applied within the cell.
- The gate activation function **f** is usually the logistic sigmoid, so that the gate activations are between 0 (gate closed) and 1 (gate open).



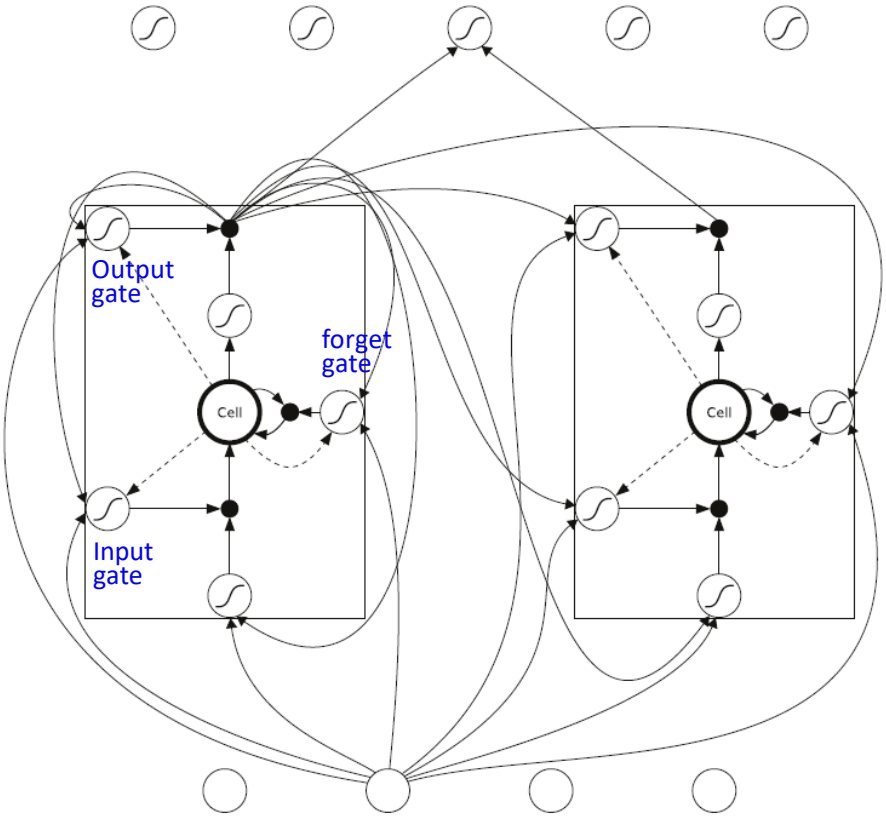
LSTM memory block

- The cell input and output activation functions (**g** and **h**) are usually tanh or logistic sigmoid, though in some cases **h** is the identity function.
- The weighted 'peephole' connections from the cell to the gates are shown with dashed lines.
- Peephole connections allow the gates to not only depend on the previous hidden state, but also on the previous memory c_{t-1}, adding an additional term in the gate equations.
- All other connections within the block are unweighted (or have a fixed weight of 1.0).
- The only outputs from the block to the rest of the network emanate from the output gate multiplication.



LSTM network

- The network consists of four input units, a hidden layer of two single-cell LSTM memory blocks and five output units.
- Note that each block has four inputs but only one output.

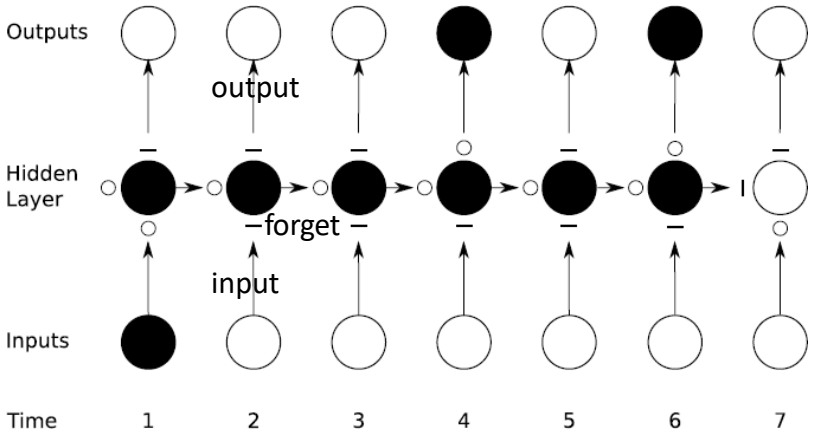


LSTM network

- An LSTM network is the same as a standard RNN, except that the summation units in the hidden layer are replaced by memory blocks.
- LSTM blocks can also be mixed with ordinary summation units, although this is typically not necessary.
- The same output layers can be used for LSTM networks as for standard RNNs.
- The multiplicative gates allow LSTM memory cells to store and access information over long periods of time, thereby mitigating the vanishing gradient problem.
  - For example, as long as the input gate remains closed (i.e. has an activation near 0), the activation of the cell will not be overwritten by the new inputs arriving in the network, and can therefore be made available to the net much later in the sequence, by opening the output gate.

Preservation of gradient information by LSTM

- The black nodes are maximally sensitive and the white nodes are entirely insensitive.
- For simplicity, all gates are either entirely open ('O') or closed ('—').
- The memory cell remembers the first input as long as the forget gate is open and the input gate is closed.
- The sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.



Gradient Calculation

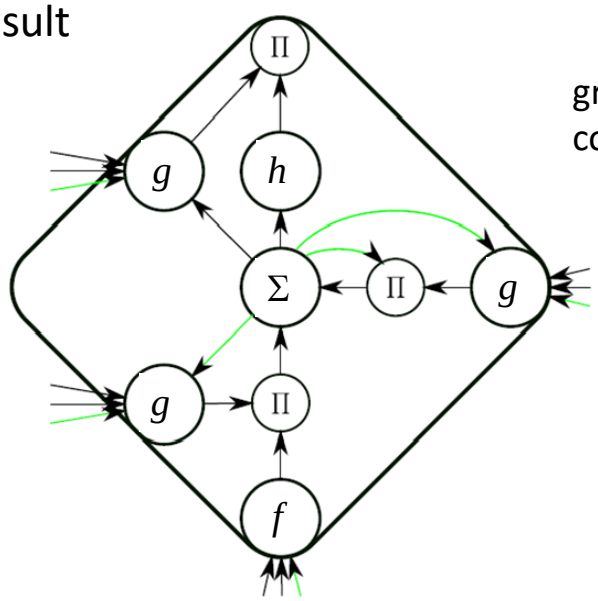
- LSTM is a differentiable function approximator that is typically trained with gradient descent.
  - The original LSTM training algorithm used an approximate error gradient calculated with a combination of Real Time Recurrent Learning (RTRL) and Backpropagation Through Time (BPTT).
  - Recently, non gradient-based training methods of LSTM have also been considered (Wierstra et al., 2005; Schmidhuber et al., 2007)

Gradient Calculation

- The BPTT part was truncated after one timestep, because it was felt that long time dependencies would be dealt with by the memory blocks, and not by the flow of activation around the recurrent connections.
- Truncating the gradient has the benefit of making the algorithm completely online, in the sense that weight updates can be made after every timestep.
- This is an important property for tasks such as continuous control or time-series prediction.

Constructing a memory block

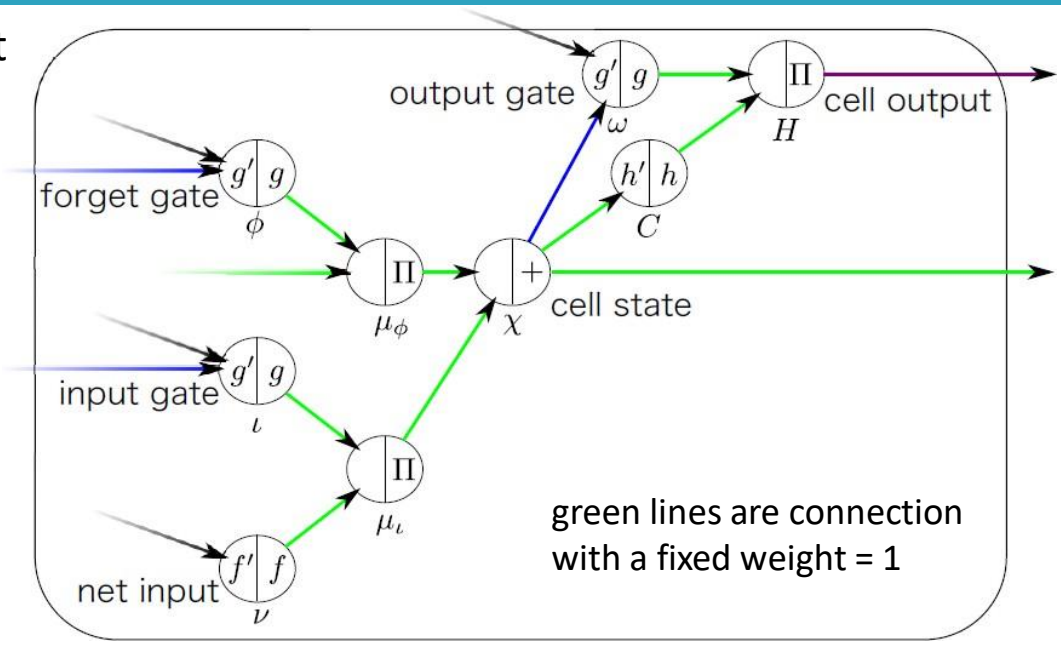
- The final result



green lines are time-delayed connections

Constructing a memory cell

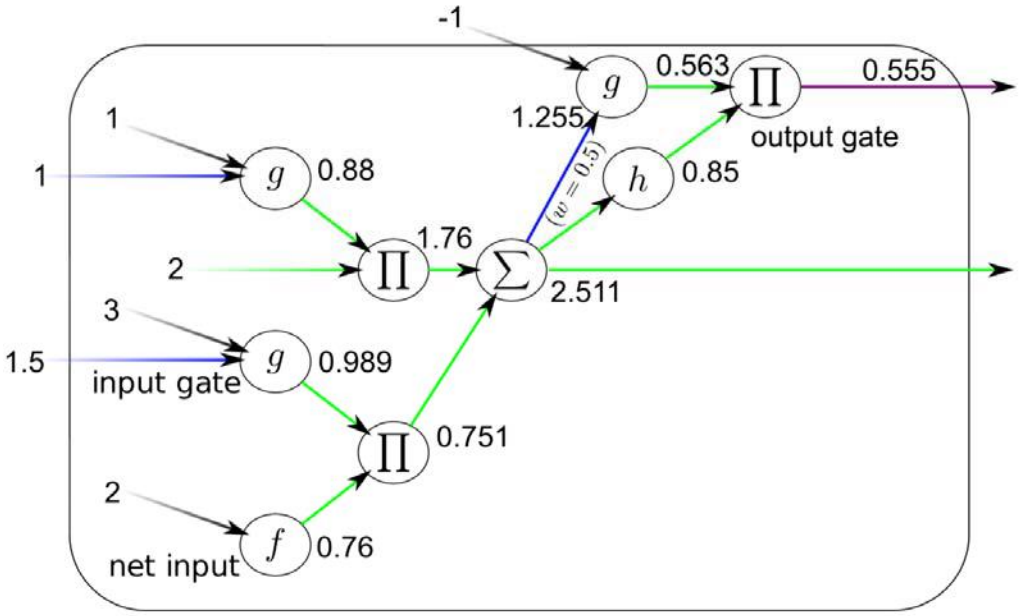
- The final result



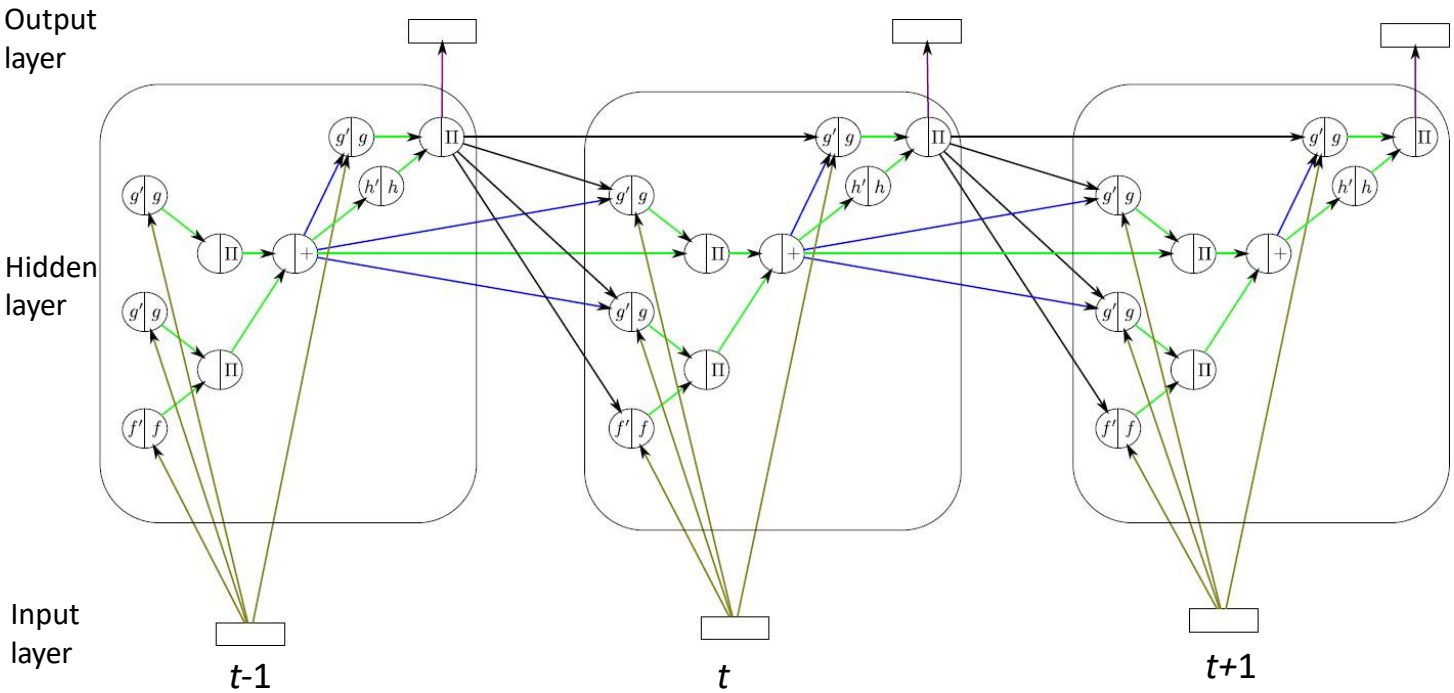
Constructing a memory cell

- Example

$$
\begin{aligned}
g(x) &= (1+\exp(-x))^{-1} \\
f(x) &= 2\,(1+\exp(-x))^{-1} - 1 \\
h(x) &= 2\,(1+\exp(-x))^{-1} - 1
\end{aligned}
$$



Complete Architecture



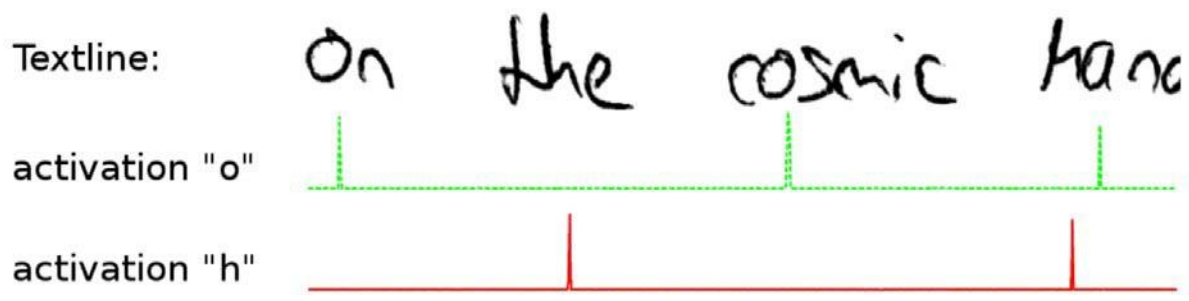
Output Layer

- The output layer can be any arbitrary output layer
- For text recognition, one output node is associated to each recognizable label
- The output layer is usually normalized via softmax

$$y_c = \frac{\exp(a_c)}{\sum_{c'} exp(a_{c'})}$$

Output Layer

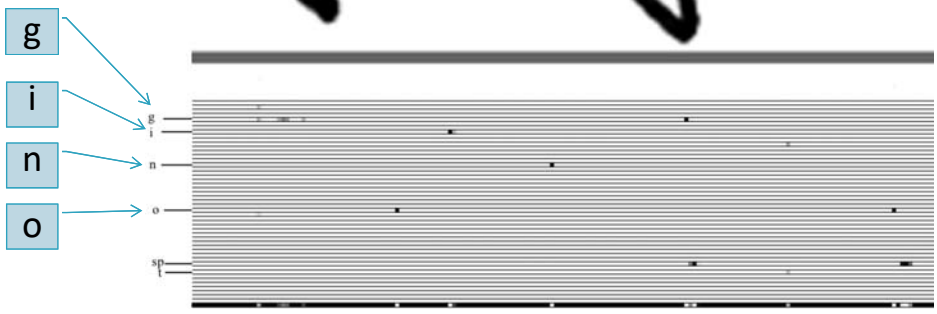
- An extra node, the blank or ε-node, in the output layer can be used as a default output



Real Word Example: Text recognition

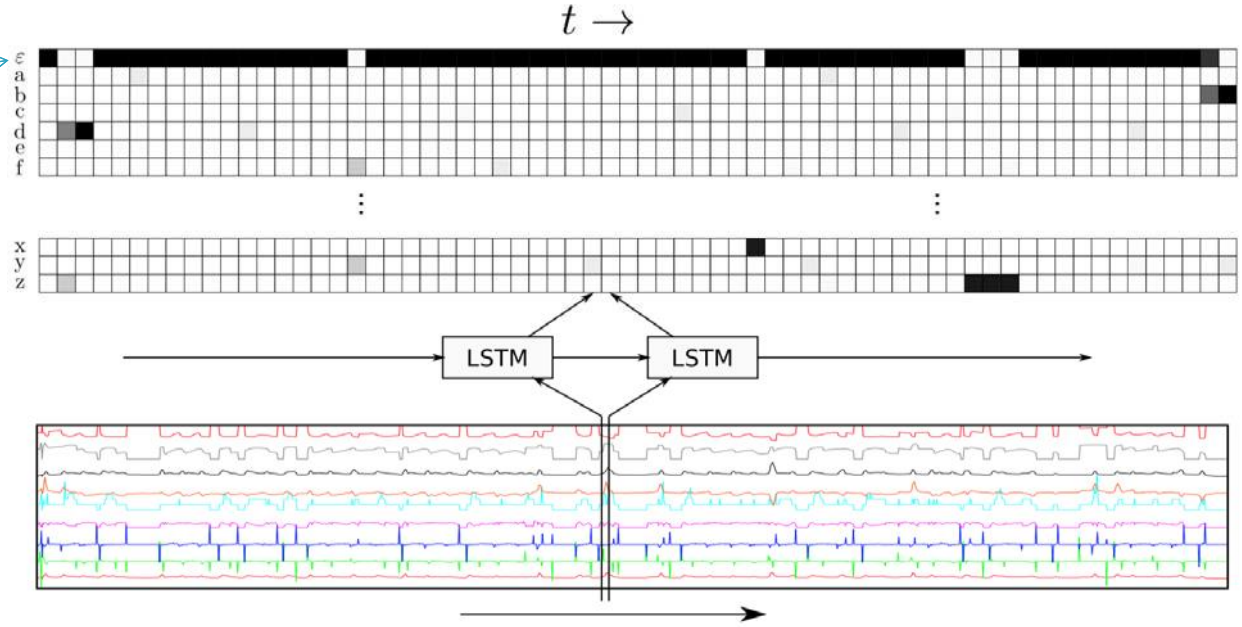
Input

Letter probabilities as returned by the neural network



Real Word Example: Text recognition

ε-node



Success of LSTM

- Over the past decade, LSTM has proved successful at a range of synthetic tasks requiring long range memory, including learning context free languages (Gers and Schmidhuber, 2001), recalling high precision real numbers over extended noisy sequences (Hochreiter and Schmidhuber, 1997) and various tasks requiring precise timing and counting (Gers et al., 2002).
- In particular, it has solved several artificial problems that remain impossible with any other RNN architecture.

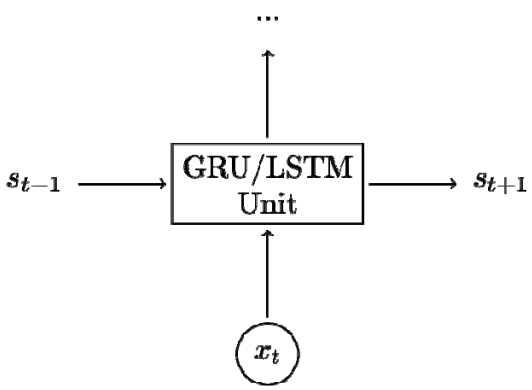
Success of LSTM

- Additionally, LSTM has been applied to various real-world problems, such as protein secondary structure prediction (Hochreiter et al., 2007; Chen and Chaudhari, 2005), music generation (Eck and Schmidhuber, 2002), reinforcement learning (Bakker, 2002), speech recognition (Graves and Schmidhuber, 2005b; Graves et al., 2006) and handwriting recog- nition (Liwicki et al., 2007; Graves et al., 2008). As would be expected, its advantages are most pronounced for problems requiring the use of long range contextual information.

GRUS (GATED RECURRENT UNITS)

Introduction

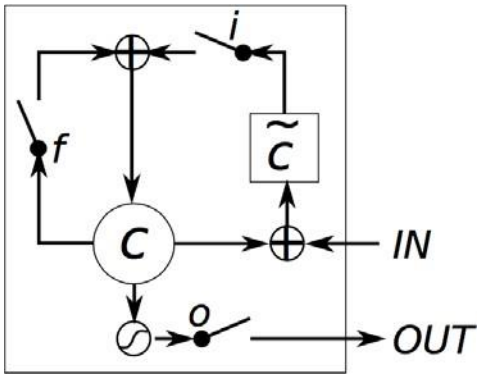
- GRUs, proposed more recently by Cho et al. [2014], are a simpler variant of LSTMs that share many of the same properties.
- You can essentially treat LSTM (and GRU) units as a black boxes. Given the current input and previous hidden state, they compute the next hidden state in some way.

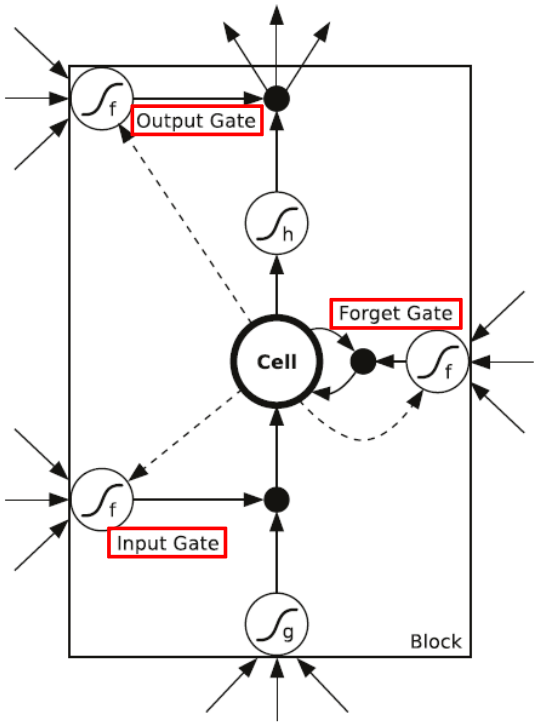


LSTM Gating

- *i*, *f*, *o* are the input, forget and output *gates*, respectively.
- c and $\tilde{c}$ denote the memory cell and the new memory cell content.

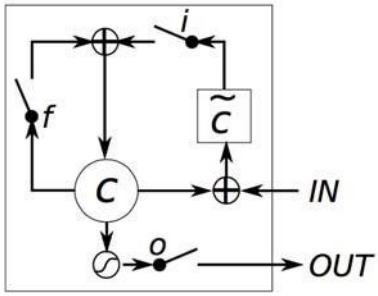
LSTM gating





LSTM Gating

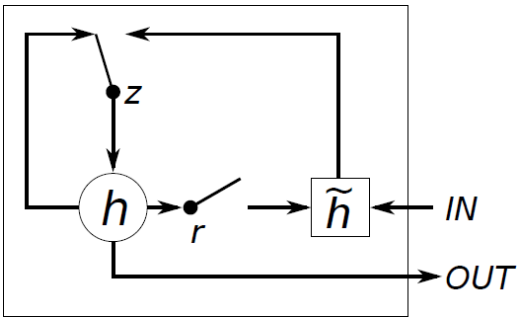
- Plain RNNs could be considered a special case of LSTMs.
- If you fix the input gate all 1's, the forget gate to all 0's (you always forget the previous memory) and the output gate to all one's (you expose the whole memory) you almost get standard RNN.
- By learning the parameters for its gates, the network learns how its memory should behave.



GRU Gating

- r and z are the reset and update gates
- h and $\tilde{h}$ are the activation and the candidate activation.
- GRUs have fewer parameters than LSTM, as they lack an output gate.

GRU Gating



GRU vs LSTM

- According to empirical evaluations in Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling and An Empirical Exploration of Recurrent Network Architectures, there isn't a clear winner.
- In many tasks both architectures yield comparable performance and tuning hyperparameters like layer size is probably more important than picking the ideal architecture.
- GRUs have fewer parameters and thus may train a bit faster or need less data to generalize.
- On the other hand, if you have enough data, the greater expressive power of LSTMs may lead to better results.

References

- **Recurrent Neural Networks Tutorial**
  - Introduction to RNNs
  - Implementing a RNN using Python and Theano
  - Understanding the Backpropagation Through Time (BPTT) algorithm and the vanishing gradient problem
  - Implementing a GRU/LSTM RNN
  - http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

LSTM IMPLEMENTATION USING TENSORFLOW

Language Modeling Example

- **Language Modeling**
  - https://www.tensorflow.org/tutorials/recurrent
  - Language modeling is key to many interesting problems such as speech recognition, machine translation, or image captioning.
  - The goal of the problem is to fit a probabilistic model which assigns probabilities to sentences. It does so by predicting next words in a text given a history of previous words.
  - For this purpose we will use the Penn Tree Bank (PTB) dataset, which is a popular benchmark for measuring the quality of these models.

Download the codes

- Find the following programs from *models/tutorials/rnn/ptb* in the TensorFlow models repo
  - https://github.com/tensorflow/models

File	Purpose
ptb_word_lm.py	The code to train a language model on the PTB dataset.
reader.py	The code to read the dataset.

Download and Prepare the Data

- The data required for this tutorial is in the data/ directory of the PTB dataset from Tomas Mikolov's webpage.
  - http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz
- The dataset is already preprocessed and contains overall 10,000 different words, including the end-of-sentence marker and a special symbol (\<unk>) for rare words.
- In reader.py, we convert each word to a unique integer identifier, in order to make it easy for the neural network to process the data.

LSTM Model

- The core of the model consists of an LSTM cell that processes one word at a time and computes probabilities of the possible values for the next word in the sentence.
- The memory state of the network is initialized with a vector of zeros and gets updated after reading each word.
- For computational reasons, we will process data in mini-batches of size batch_size.
- In this example, it is important to note that current_batch_of_words does not correspond to a "sentence" of words.
- Every word in a batch should correspond to a time t.
- TensorFlow will automatically sum the gradients of each batch for you.

LSTM Model

- For example:

```
 t=0  t=1    t=2  t=3     t=4
[The, brown, fox, is,     quick]
[The, red,   fox, jumped, high]

words_in_dataset[0] = [The, The]
words_in_dataset[1] = [brown, red]
words_in_dataset[2] = [fox, fox]
words_in_dataset[3] = [is, jumped]
words_in_dataset[4] = [quick, high]
batch_size = 2, time_steps = 5
```

```
words_in_dataset = tf.placeholder(tf.float32, [time_steps, batch_size,
num_features])
lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
hidden_state = tf.zeros([batch_size, lstm.state_size])
current_state = tf.zeros([batch_size, lstm.state_size])
state = hidden_state, current_state
probabilities = []
loss = 0.0
for current_batch_of_words in words_in_dataset:
    # The value of state is updated after processing each batch of words.
    output, state = lstm(current_batch_of_words, state)

    # The LSTM output can be used to make next word predictions
    logits = tf.matmul(output, softmax_w) + softmax_b
    probabilities.append(tf.nn.softmax(logits))
    loss += loss_function(probabilities, target_words)
```

The basic pseudocode

Truncated Backpropagation

- In order to make the learning process tractable, it is common practice to create an "unrolled" version of the network, which contains a fixed number (num_steps) of LSTM inputs and outputs.
- The model is then trained on this finite approximation of the RNN.
- This can be implemented by feeding inputs of length num_steps at a time and performing a backward pass after each such input block.
- Here is a simplified block of code for creating a graph which performs truncated backpropagation:

```
# Placeholder for the inputs in a given iteration.
words = tf.placeholder(tf.int32, [batch_size, num_steps])

lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
initial_state = state = tf.zeros([batch_size, lstm.state_size])

for i in range(num_steps):
    # The value of state is updated after processing each batch of words.
    output, state = lstm(words[:, i], state)

    # The rest of the code.
    # ...

final_state = state
```

Truncated Backpropagation

- And this is how to implement an iteration over the whole dataset:

```
# A numpy array holding the state of LSTM after each batch of words.
numpy_state = initial_state.eval()
total_loss = 0.0
for current_batch_of_words in words_in_dataset:
    numpy_state, current_loss = session.run([final_state, loss],
        # Initialize the LSTM state from the previous iteration.
        feed_dict={initial_state: numpy_state, words:
current_batch_of_words})
    total_loss += current_loss
```

Inputs

- The word IDs will be embedded into a dense representation (see the Vector Representations Tutorial) before feeding to the LSTM.
- This allows the model to efficiently represent the knowledge about particular words. It is also easy to write:

```
# embedding_matrix is a tensor of shape [vocabulary_size, embedding size]
word_embeddings = tf.nn.embedding_lookup(embedding_matrix, word_ids)
```

- The embedding matrix will be initialized randomly and the model will learn to differentiate the meaning of words just by looking at the data.

Loss Function

- We want to minimize the average negative log probability of the target words:

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N} \ln p_{\text{target}_i}$$

- It is not very difficult to implement but the function sequence_loss_by_example is already available, so we can just use it here.
- The typical measure reported in the papers is average per-word perplexity (often just called perplexity), which is equal to

$$e^{-\frac{1}{N}\sum_{i=1}^{N} \ln p_{\text{target}_i}} = e^{\text{loss}}$$

Stacking multiple LSTMs

- To give the model more expressive power, we can add multiple layers of LSTMs to process the data. The output of the first layer will become the input of the second and so on.
- We have a class called MultiRNNCell that makes the implementation seamless:

Stacking multiple LSTMs


```
def lstm_cell():
  return tf.contrib.rnn.BasicLSTMCell(lstm_size)
stacked_lstm = tf.contrib.rnn.MultiRNNCell(
    [lstm_cell() for _ in range(number_of_layers)])

initial_state = state = stacked_lstm.zero_state(batch_size, tf.float32)
for i in range(num_steps):
    # The value of state is updated after processing each batch of words.
    output, state = stacked_lstm(words[:, i], state)

    # The rest of the code.
    # ...

final_state = state
```

Run the Code

- Before running the code, download the PTB dataset, as discussed at the beginning of this tutorial. Then, extract the PTB dataset underneath your home directory as follows:

```
tar xvfz simple-examples.tgz -C $HOME
```

(Note: On Windows, you may need to use other tools.)

- Now, clone the TensorFlow models repo from GitHub. Run the following commands:

```
cd models/tutorials/rnn/ptb
python ptb_word_lm.py --data_path=$HOME/simple-examples/data/ --model=small
```

Run the Code

- There are 3 supported model configurations in the tutorial code: "small", "medium" and "large".
  - The difference between them is in size of the LSTMs and the set of hyperparameters used for training.
  - The larger the model, the better results it should get.
- The small model should be able to reach perplexity below 120 on the test set and the large one below 80, though it might take several hours to train.

What Next?

- There are several tricks that we haven't mentioned that make the model better, including:
  - decreasing learning rate schedule,
  - dropout between the LSTM layers.
- Study the code and modify it to improve the model even further.