CSCE 496/896 Lecture 6: Recurrent Architectures

Stephen Scott

(Adapted from Vinod Variyam and Ian Goodfellow)

sscott@cse.unl.edu

Introduction

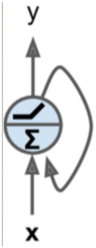
- All our architectures so far work on fixed-sized inputs
- Recurrent neural networks work on **sequences** of inputs
- E.g., text, biological sequences, video, audio
- Can also try 1D convolutions, but lose long-term relationships in input
- Especially useful for NLP applications: translation, speech-to-text, sentiment analysis
- Can also **create novel output:** e.g., Shakespearean text, music

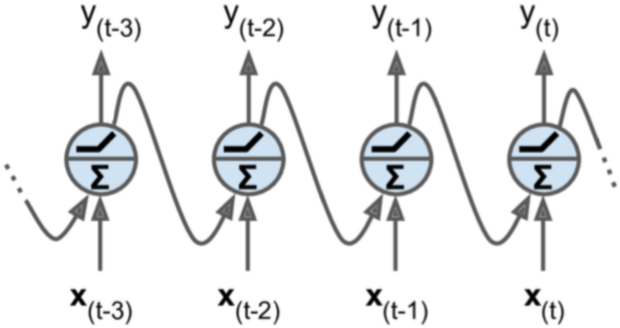
Outline

- Basic RNNs
- Input/Output Mappings
- Example Implementations
- Training
- Long short-term memory
- Gated Recurrent Unit

Basic Recurrent Cell

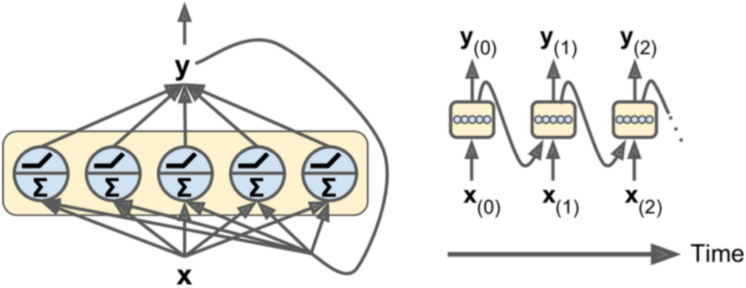
- A recurrent cell (or recurrent neuron) has connections pointing **backward** as well as forward
- At time step (frame) t, neuron receives input vector $\boldsymbol{x}_{(t)}$ as usual, but also receives its own output $y_{(t-1)}$ from previous step





Basic Recurrent Layer

- Can build a layer of recurrent cells, where each node gets both the vector $\boldsymbol{x}_{(t)}$ and the vector $\boldsymbol{y}_{(t-1)}$



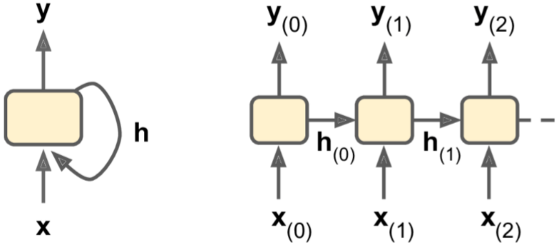
Basic Recurrent Layer

- Each node in the recurrent layer has independent weights for both $\boldsymbol{x}_{(t)}$ and $\boldsymbol{y}_{(t-1)}$
- For a single recurrent node, denote by $\boldsymbol{w_x}$ and $\boldsymbol{w_y}$
- For the entire layer, combine into matrices $W_{\boldsymbol{x}}$ and $W_{\boldsymbol{y}}$
- For activation function ϕ and bias vector $\boldsymbol{b}$, output vector is

$$\boldsymbol{y}_{(t)} = \phi\left(W_{\boldsymbol{x}}^{\top}\boldsymbol{x}_{(t)} + W_{\boldsymbol{y}}^{\top}\boldsymbol{y}_{(t-1)} + \boldsymbol{b}\right)$$

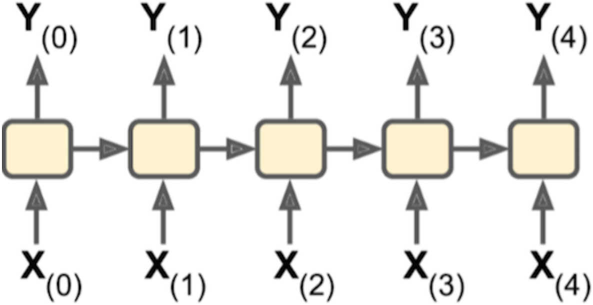
Memory and State

- Since a node's output depends on its past, it can be thought of having **memory** or **state**
- State at time t is $\boldsymbol{h}_{(t)} = f(\boldsymbol{h}_{(t-1)}, \boldsymbol{x}_{(t)})$ and output $\boldsymbol{y}_{(t)} = g(\boldsymbol{h}_{(t-1)}, \boldsymbol{x}_{(t)})$
- State could be the same as the output, or separate
- Can think of $\boldsymbol{h}_{(t)}$ as storing important information about input sequence
- Analogous to convolutional outputs summarizing important image features



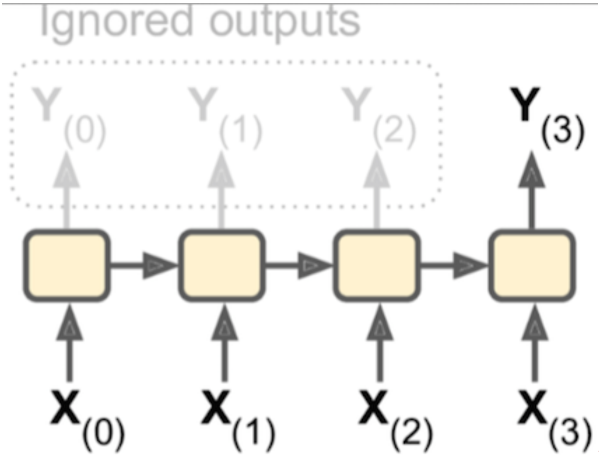
Many ways to employ this basic architecture:

- **Sequence to sequence:** Input is a sequence and output is a sequence
- E.g., series of stock predictions, one day in advance



Input/Output Mappings
Sequence to Vector

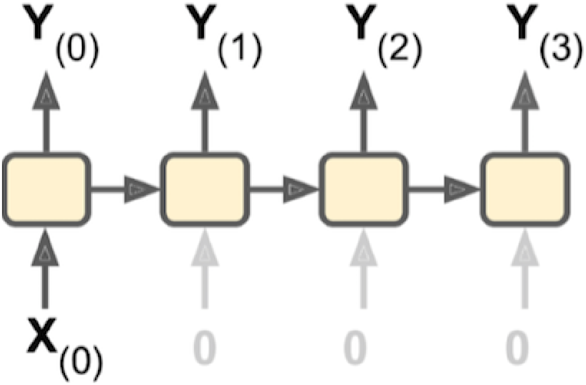
- **Sequence to vector:** Input is sequence and output a vector/score/ classification
- E.g., sentiment score of movie review



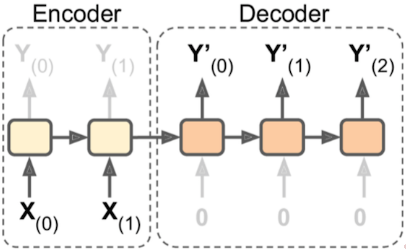
Input/Output Mappings

Vector to Sequence

- **Vector to sequence:** Input is a single vector (zeroes for other times) and output is a sequence
- E.g., image to caption



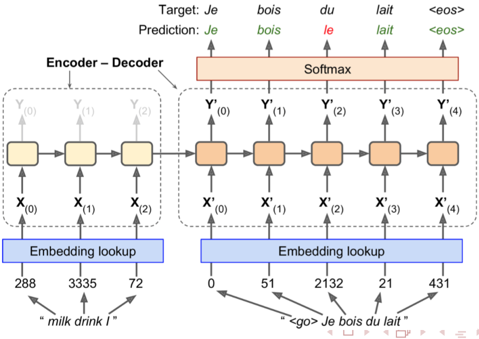
- **Encoder-decoder:** Sequence-to-vector (**encoder**) followed by vector-to-sequence (**decoder**)
- Input sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ yields hidden outputs $(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T)$, then mapped to **context vector** $\boldsymbol{c} = f(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T)$
- Decoder output $\boldsymbol{y}_{t'}$ depends on previously output $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t'-1})$ and $\boldsymbol{c}$
- Example application: **neural machine translation**



Input/Output Mappings

Encoder-Decoder Architecture: NMT Example

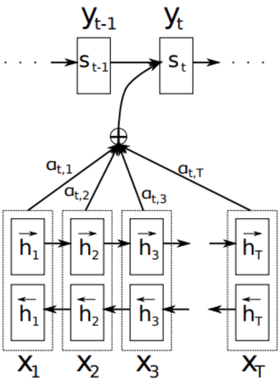
- Pre-trained word embeddings fed into input
- Encoder maps word sequence to vector, decoder maps to translation via softmax distribution
- After training, do translation by feeding previous translated word $y'_{(t-1)}$ to decoder



- Works through an **embedded space** like an autoencoder, so can represent the entire input as an embedded vector prior to decoding
- Issue: Need to ensure that the context vector fed into decoder is sufficiently large in dimension to represent context required
- Can address this representation problem via **attention mechanism** mechanism
  - Encodes input sequence into a vector sequence rather than single vector
  - As it decodes translation, decoder focuses on relevant subset of the vectors

- **Bidirectional RNN** reads input forward and backward simultaneously
- Encoder builds **annotation** $\boldsymbol{h}_j$ as concatenation of $\overrightarrow{\boldsymbol{h}}_j$ and $\overleftarrow{\boldsymbol{h}}_j$
  - $\Rightarrow$ $\boldsymbol{h}_j$ summarizes preceding and following inputs
- ith context vector $\boldsymbol{c}_i = \sum_{j=1}^{T} \alpha_{ij} \boldsymbol{h}_j$, where

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}$$

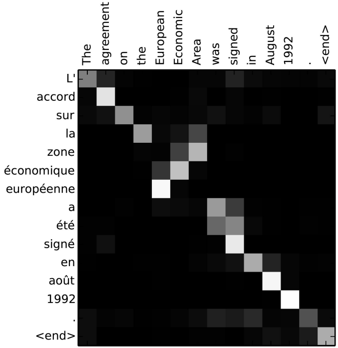


and e_{ij} is an **alignment score** between inputs around j and outputs around i

Input/Output Mappings

E-D Architecture: Attention Mechanism (Bahdanau et al., 2015)

- The ith element of **attention vector** $\boldsymbol{\alpha}_j$ tells us the probability that target output $\boldsymbol{y}_i$ is aligned to (or translated from) input $\boldsymbol{x}_j$
- Then $\boldsymbol{c}_i$ is expected annotation over all annotations with probabilities $\boldsymbol{\alpha}_j$



- Alignment score e_{ij} indicates how much we should focus on word encoding $\boldsymbol{h}_j$ when generating output $\boldsymbol{y}_i$ (in decoder state $\boldsymbol{s}_{i-1}$)
- Can compute e_{ij} via dot product $\boldsymbol{h}_j^\top \boldsymbol{s}_{i-1}$, bilinear function $\boldsymbol{h}_j^\top W \boldsymbol{s}_{i-1}$, or nonlinear activation

Example Implementation
Static Unrolling for Two Time Steps


```
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])
Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons],dtype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons,n_neurons],dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))
Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)
```

Input:

```
# Mini-batch:      instance 0, instance 1, instance 2, instance 3
X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1
```

Can achieve the same thing more compactly via `static_rnn()`

```
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, [X0, X1],
    dtype=tf.float32)
Y0, Y1 = output_seqs
```

Automatically unrolls into length-2 sequence RNN

Can avoid specifying one placeholder per time step via `tf.stack` and `tf.unstack`

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
    X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
    basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
    output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, X_seqs,
                                                    dtype=tf.float32)
    outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
...
X_batch = np.array([
   # t=0        t=1
[[0, 1, 2], [9, 8, 7]], # instance 0
[[3, 4, 5], [0, 0, 0]], # instance 1
[[6, 7, 8], [6, 5, 4]], # instance 2
[[9, 0, 1], [3, 2, 1]], # instance 3
])
```

- Uses `static_rnn()` again, but on all time steps folded into a single tensor
- Still forms a large, static graph (possible memory issues)

Even better: Let TensorFlow unroll **dynamically** via a `while_loop()` in `dynamic_rnn()`

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```

Can also set `swap_memory=True` to reduce memory problems

Example Implementation
Variable-Length Sequences

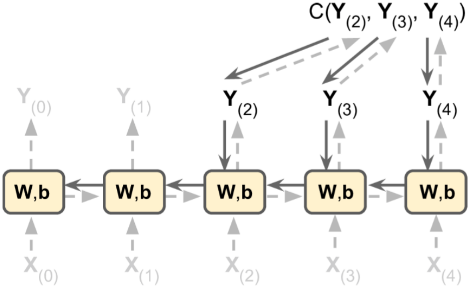
- May need to handle **variable-length inputs**
- Use 1D tensor `sequence_length` to set length of each input (and maybe output) sequence
- Pad smaller inputs with zeroes to fit input tensor
- Use "end-of-sequence" symbol at end of each output

```
seq_length = tf.placeholder(tf.int32, [None])
...
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32,
                                        sequence_length=seq_length)
...
X_batch = np.array([
    # step 0     step 1
    [[0, 1, 2], [9, 8, 7]], # instance 0
    [[3, 4, 5], [0, 0, 0]], # instance 1 (padded with a zero vector)
    [[6, 7, 8], [6, 5, 4]], # instance 2
    [[9, 0, 1], [3, 2, 1]], # instance 3
  ])
seq_length_batch = np.array([2, 1, 2, 2])
...
with tf.Session() as sess:
     init.run()
     outputs_val, states_val = sess.run(
         [outputs, states], feed_dict={X: X_batch, seq_length: seq_length_batch})
```

Training

Backpropagation Through Time (BPTT)

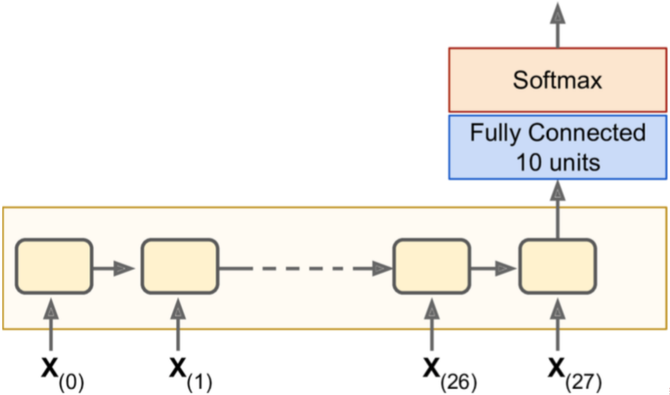
- Unroll through time and use BPTT
- Forward pass mini-batch of sequences through unrolled network yields output sequence $Y_{(t_{\min})}, \ldots, Y_{(t_{\max})}$
- Output sequence evaluated using cost $C\left(Y_{(t_{\min})}, \ldots, Y_{(t_{\max})}\right)$
- Gradients propagated backward through unrolled network (summing over all time steps), and parameters



Training

Example: Training on MNIST as a Vector Sequence

- Consider MNIST inputs provided as sequence of 28 inputs of 28-dimensional vectors
- Feed in input as usual, then compute loss between target and softmax output after 28th input

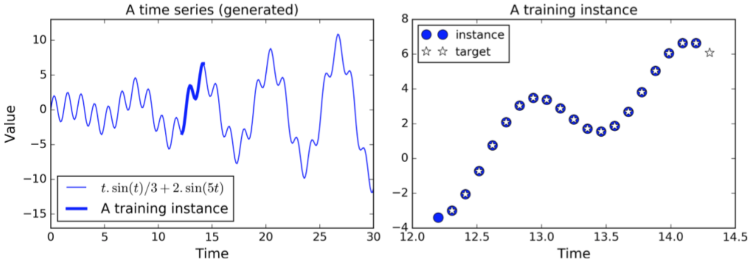


Training

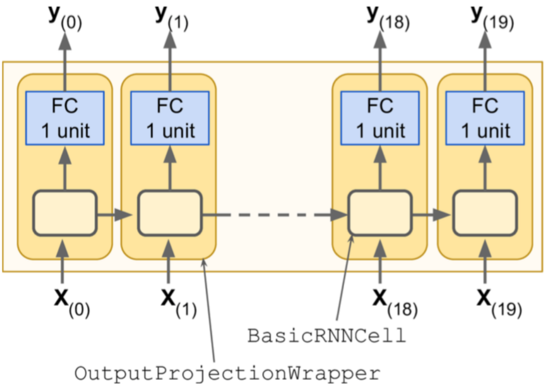
Example: Training on MNIST as a Vector Sequence

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
logits = tf.layers.dense(states, n_outputs)
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                    logits=logits)
loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
init = tf.global_variables_initializer()
```

- Input is **time series**
- Target is same as input, but shifted one into the future
- E.g., stock prices, temperature

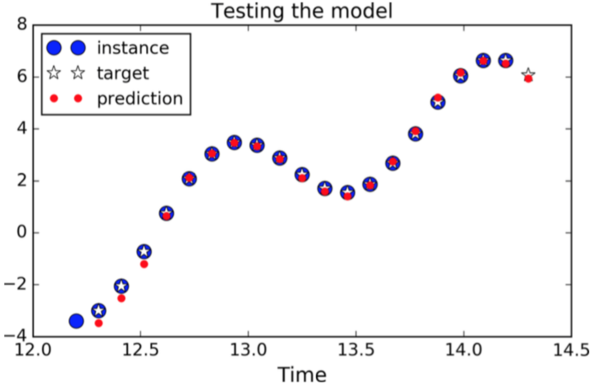


- Use sequences of length `n_steps=20` and `n_neurons=100` recurrent neurons
- Since output size $= 100 > 1 =$ target size, use `OutputProjectionWrapper` to feed recurrent layer output into a linear unit to get a scalar



```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.OutputProjectionWrapper(
        tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu),
        output_size=n_outputs)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

Results on same sequence after 1000 training iterations

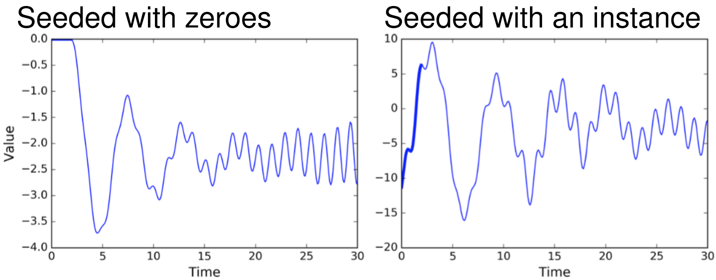


Training

Example: Creating New Time Series

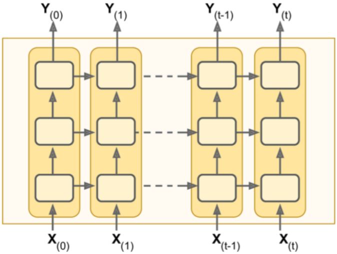
- Feed to trained model **seed sequence** of size `n_steps`, append predicted value to sequence, feed last `n_steps` back in to predict next value, etc.

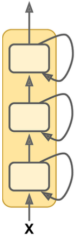
```
sequence = [0.] * n_steps
for iteration in range(300):
    X_batch = np.array(sequence[-n_steps:]).reshape(1, n_steps, 1)
    y_pred = sess.run(outputs, feed_dict={X: X_batch})
    sequence.append(y_pred[0, -1, 0])
```



Deep RNNs

- A **deep RNN** has multiple recurrent layers stacked





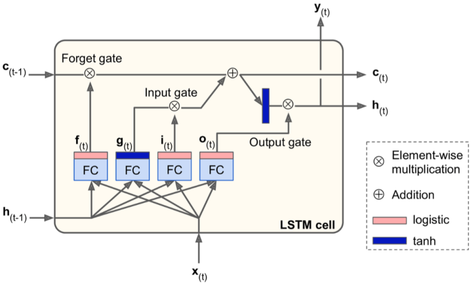

```
n_neurons = 100
n_layers = 3
layers = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
                                      activation=tf.nn.relu)
          for layer in range(n_layers)]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(layers)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

Training over Many Time Steps

- Vanishing and exploding gradients can be a problem with RNNs, like with other deep networks
  - Can as usual address with, e.g., ReLU, batch normalization, gradient clipping, etc.
- Can still suffer from long training times with long input sequences
  - **Truncated backpropagation through time** addresses this by limiting `n_steps`
  - Lose ability to learn long-term patterns
- In general, also have problem of first inputs of sequence have diminishing impact as sequence grows
  - E.g., first few words of long text sequence
- Goal: Introduce **long-term memory** to RNNs
- Allow a network to **accumulate** information about the past, but also decide when to **forget** information

Long Short-Term Memory
Hochreiter and Schmidhuber (1997)

- Vector $\boldsymbol{h}_{(t)}$ = **short-term state**, $\boldsymbol{c}_{(t)}$ = **long-term state**
- At time t, some memories from $\boldsymbol{c}_{(t-1)}$ are forgotten in the **forget gate** and new ones (from **input gate**) added

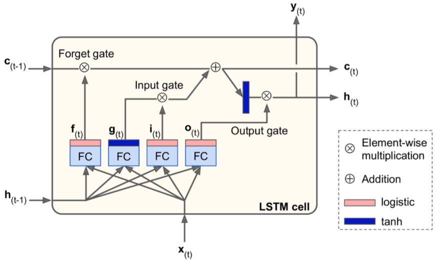


- Result sent out as $\boldsymbol{c}_{(t)}$
- $\boldsymbol{h}_{(t)}$ (and $\boldsymbol{y}_{(t)}$) comes from processing long-term state in **output gate**

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
```

Long Short-Term Memory

Hochreiter and Schmidhuber (1997)

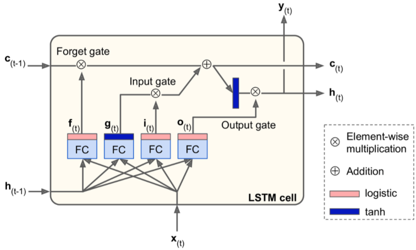


- $\boldsymbol{g}_{(t)}$ combines input $\boldsymbol{x}_{(t)}$ with state $\boldsymbol{h}_{(t-1)}$
- $\boldsymbol{f}_{(t)}, \boldsymbol{i}_{(t)}, \boldsymbol{o}_{(t)}$ are **gate controllers**
- $\boldsymbol{f}_{(t)} \in [0, 1]^n$ controls forgetting of $\boldsymbol{c}_{(t-1)}$
- $\boldsymbol{i}_{(t)}$ controls remembering of $\boldsymbol{g}_{(t)}$
- $\boldsymbol{o}_{(t)}$ controls what of $\boldsymbol{c}_{(t)}$ goes to output and $\boldsymbol{h}_{(t)}$
- Output depends on long- and short-term memory
- Network learns what to remember long-term based on $\boldsymbol{x}_{(t)}$ and $\boldsymbol{h}_{(t-1)}$

Long Short-Term Memory
Hochreiter and Schmidhuber (1997)

- $\boldsymbol{i}_{(t)} = \sigma\left(W_{xi}^\top \boldsymbol{x}_{(t)} + W_{hi}^\top \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_i\right)$
- $\boldsymbol{f}_{(t)} = \sigma\left(W_{xf}^\top \boldsymbol{x}_{(t)} + W_{hf}^\top \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_f\right)$
- $\boldsymbol{o}_{(t)} = \sigma\left(W_{xo}^\top \boldsymbol{x}_{(t)} + W_{ho}^\top \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_o\right)$
- $\boldsymbol{g}_{(t)} = \tanh\left(W_{xg}^\top \boldsymbol{x}_{(t)} + W_{hg}^\top \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_g\right)$

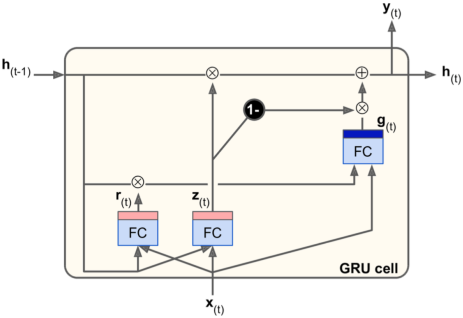
- $\boldsymbol{c}_{(t)} = \boldsymbol{f}_{(t)} \otimes \boldsymbol{c}_{(t-1)} + \boldsymbol{i}_{(t)} \otimes \boldsymbol{g}_{(t)}$
- $\boldsymbol{y}_{(t)} = \boldsymbol{h}_{(t)} = \boldsymbol{o}_{(t)} \otimes \tanh\left(\boldsymbol{c}_{(t)}\right)$



- Can add **peephole connection:** Let $\boldsymbol{c}_{(t-1)}$ affect $\boldsymbol{f}_{(t)}$ and $\boldsymbol{i}_{(t)}$ and $\boldsymbol{c}_{(t-1)}$ affect $\boldsymbol{o}_{(t)}$

Gated Recurrent Unit

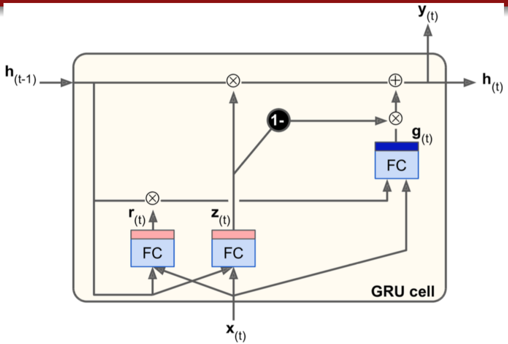
- Simplified LSTM
- Merge $\boldsymbol{c}_{(t)}$ into $\boldsymbol{h}_{(t)}$
- Merge $\boldsymbol{f}_{(t)}$ and $\boldsymbol{i}_{(t)}$ into $z_{(t)}$
  - $z_{(t),i} = 0 \Rightarrow$ forget $\boldsymbol{h}_{(t-1),i}$ and add in $\boldsymbol{g}_{(t),i}$



- $\boldsymbol{o}_{(t)}$ replaced by $\boldsymbol{r}_{(t)} \Rightarrow$ forget part of $\boldsymbol{h}_{(t-1)}$ when computing $\boldsymbol{g}_{(t)}$

```
gru_cell = tf.contrib.rnn.GRUCell(num_units=n_neurons)
```

Gated Recurrent Unit



- $\boldsymbol{z}_{(t)} = \sigma\left(W_{xz}^\top \boldsymbol{x}_{(t)} + W_{hz}^\top \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_z\right)$
- $\boldsymbol{r}_{(t)} = \sigma\left(W_{xr}^\top \boldsymbol{x}_{(t)} + W_{hr}^\top \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_r\right)$
- $\boldsymbol{g}_{(t)} = \tanh\left(W_{xg}^\top \boldsymbol{x}_{(t)} + W_{hg}^\top \left(\boldsymbol{r}_{(t)} \otimes \boldsymbol{h}_{(t-1)}\right) + \boldsymbol{b}_g\right)$
- $\boldsymbol{y}_{(t)} = \boldsymbol{h}_{(t)} = \boldsymbol{z}_{(t)} \otimes \boldsymbol{h}_{(t-1)} + \left(\boldsymbol{1} - \boldsymbol{z}_{(t)}\right) \otimes \boldsymbol{g}_{(t)}$