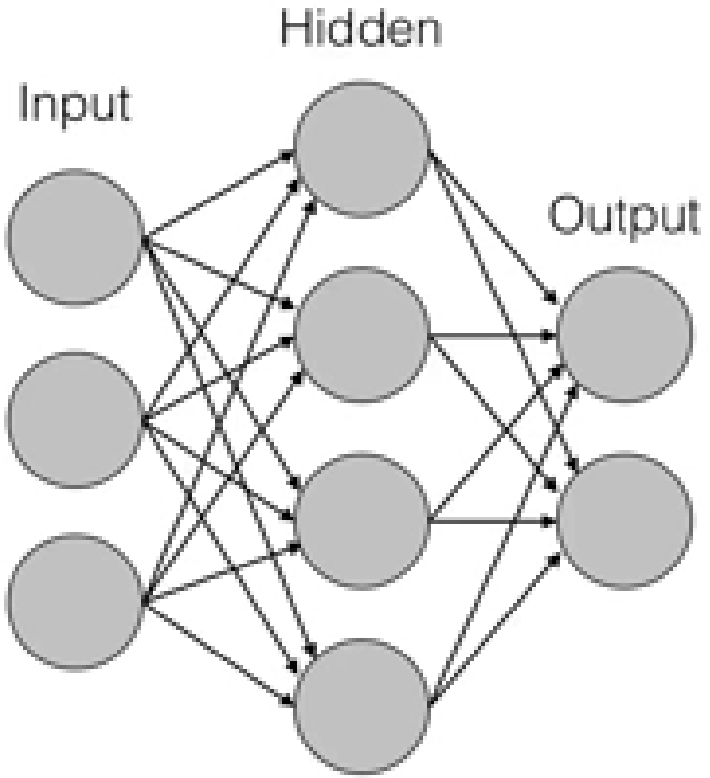
Recurrent Neural Network and LSTM Internals

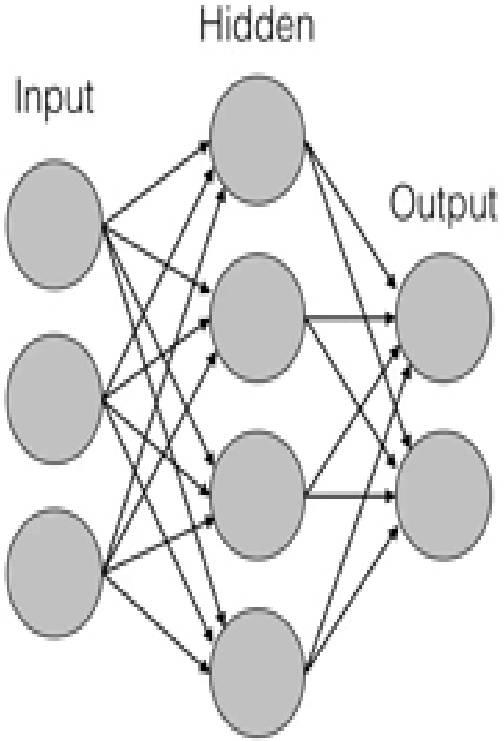
By Mohit Kumar

Recurrent Neural Networks:why?

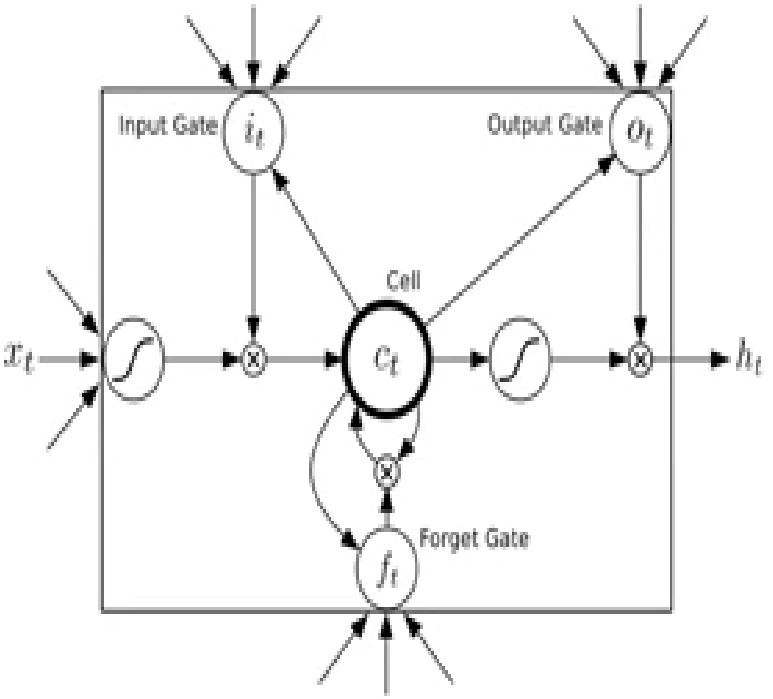


- Independence
- Fixed Length

Recurrent Neural Networks:why?

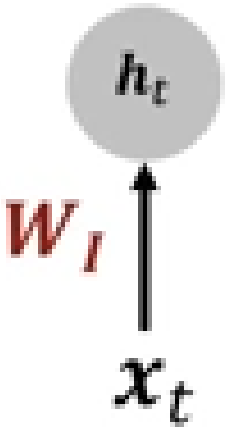


- Independence
- Fixed Length

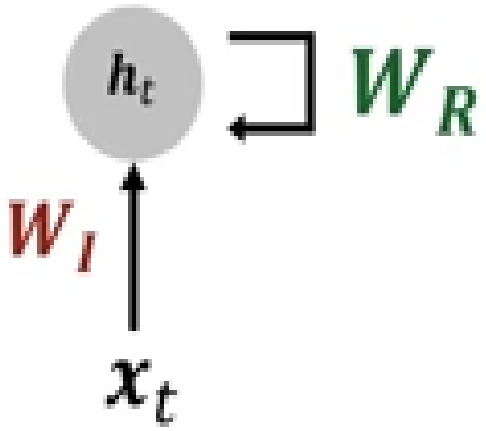


- Temporal dependencies
- Variable sequence length

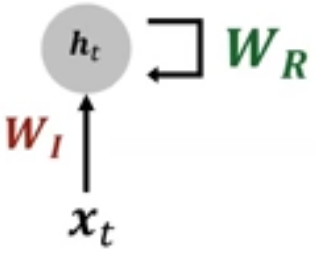
Recurrent Neural Networks:Recurrent Neuron



Recurrent Neural Networks:Recurrent Neuron

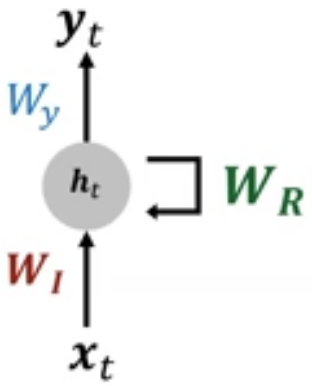


Recurrent Neural Networks:Recurrent Neuron



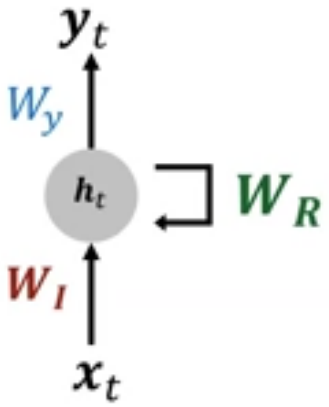
$$\boldsymbol{h}^{(t)} = g_h(W_{\mathrm{I}}\boldsymbol{x}^{(t)} + W_{\mathrm{R}}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

Recurrent Neural Networks:Recurrent Neuron



$$\boldsymbol{h}^{(t)} = g_h(W_{\mathrm{I}}\boldsymbol{x}^{(t)} + W_{\mathrm{R}}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

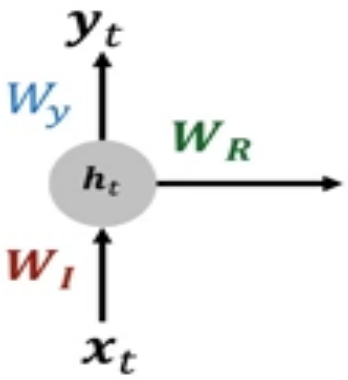
Recurrent Neural Networks:Recurrent Neuron



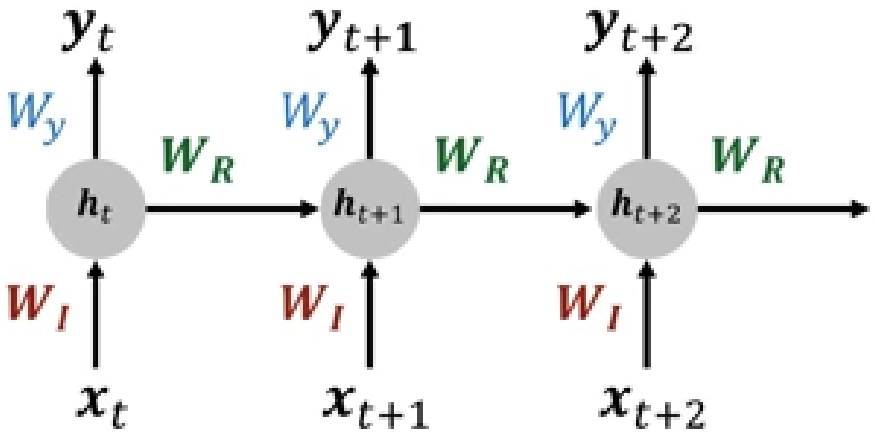
$$\boldsymbol{h}^{(t)} = g_h(W_{\mathrm{I}}\boldsymbol{x}^{(t)} + W_{\mathrm{R}}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

$$\boldsymbol{y}^{(t)} = g_y(W_y\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$

Recurrent Neural Networks:Unrolling an RNN through time into An ANN



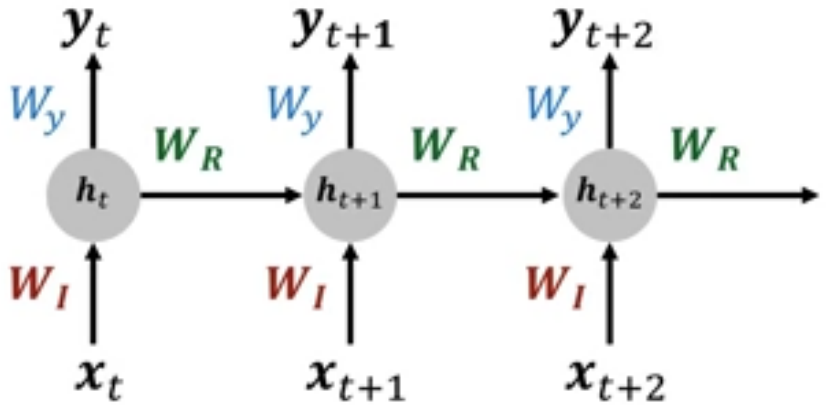
Recurrent Neural Networks:Unrolling an RNN through time into An ANN



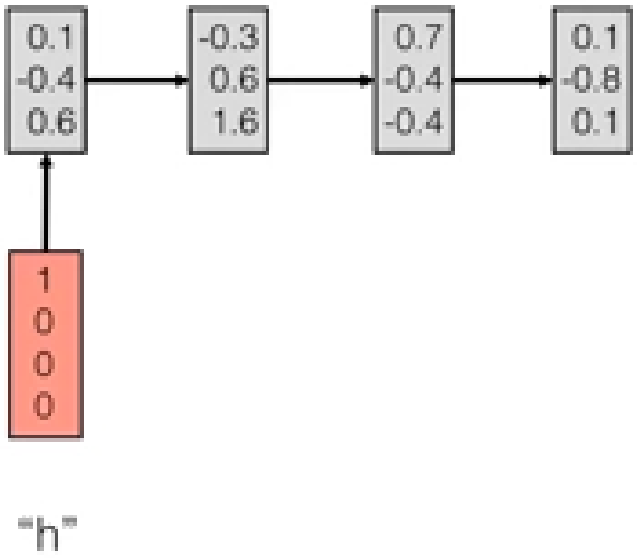
Recurrent Neural Networks:Unrolling an RNN through time into An ANN

$$\boldsymbol{h}^{(t)} = g_h(W_{\mathrm{I}}\boldsymbol{x}^{(t)} + W_{\mathrm{R}}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

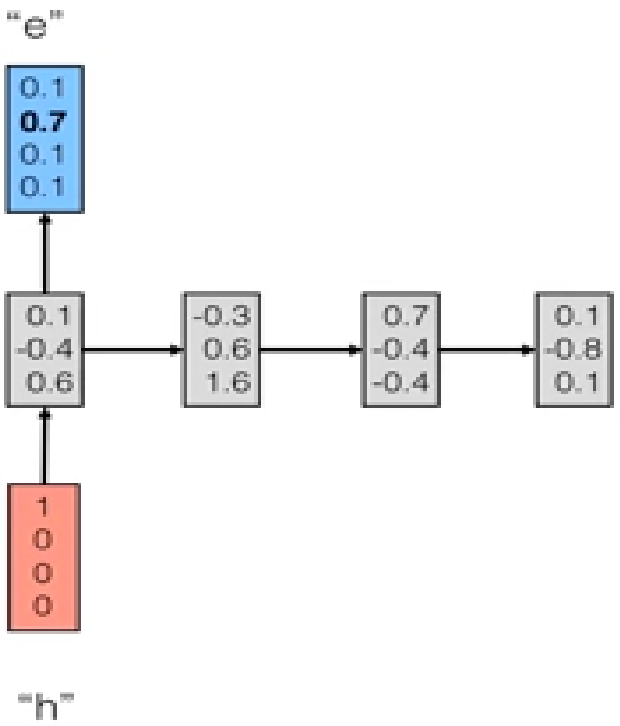
$$\boldsymbol{y}^{(t)} = g_y(W_y\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$



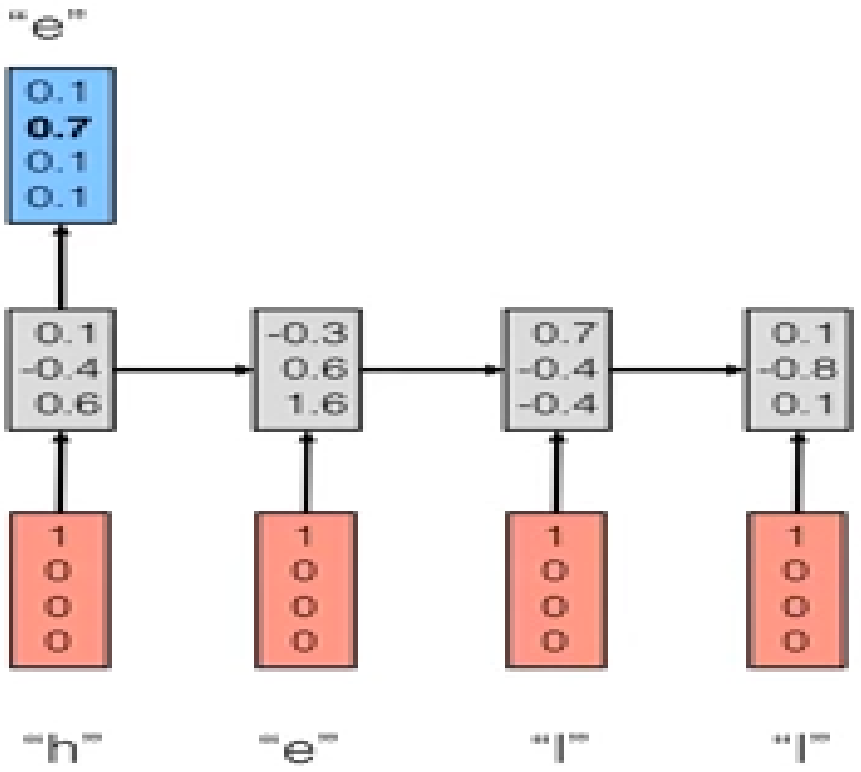
Recurrent Neural Networks:What is it good for?



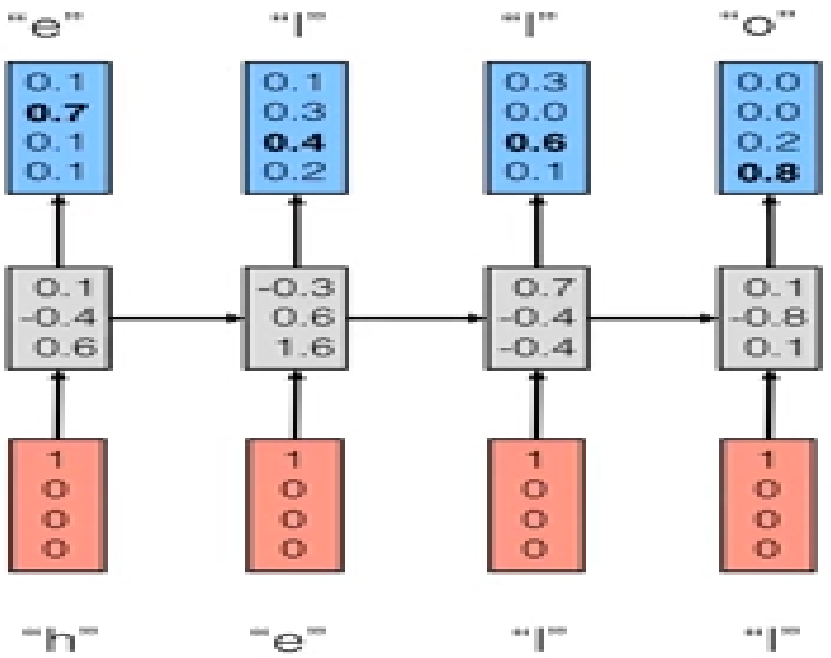
Recurrent Neural Networks:What is it good for?



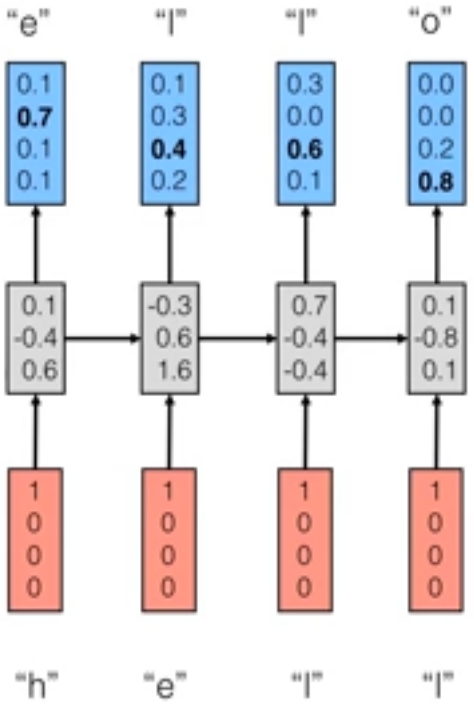
Recurrent Neural Networks:What is it good for?



Recurrent Neural Networks:What is it good for?



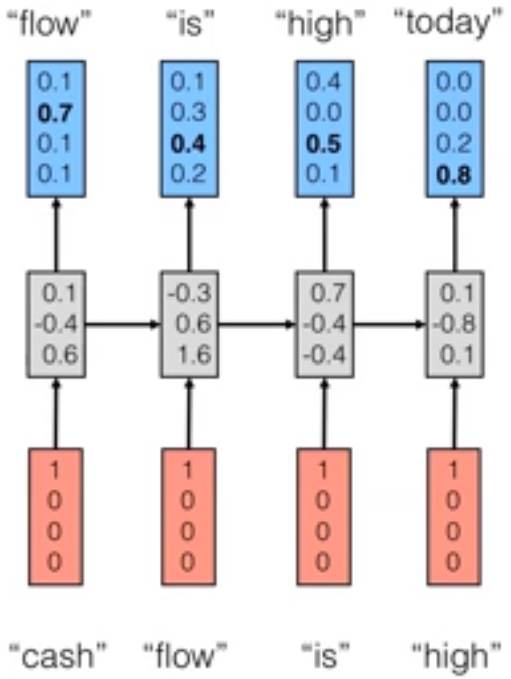
Recurrent Neural Networks:What is it good for?



Learned a language model!

$$P(c_t|\{c_{t-1}, c_{t-2}, \dots, c_0\})$$

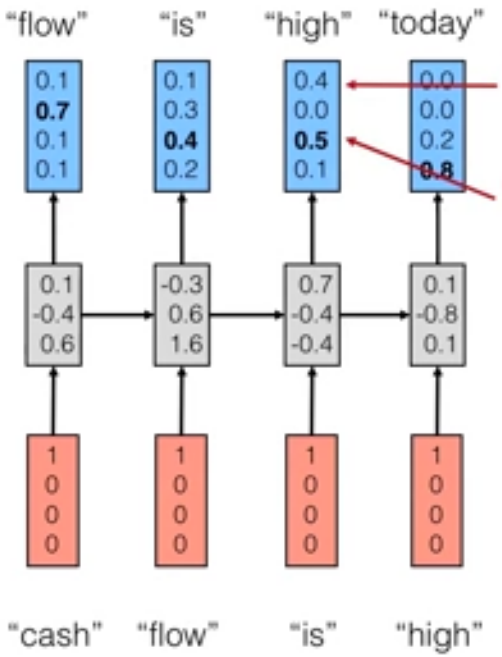
Recurrent Neural Networks:What is it good for?



Learned a language model!

$$P(c_t|\{c_{t-1}, c_{t-2}, \ldots, c_0\})$$

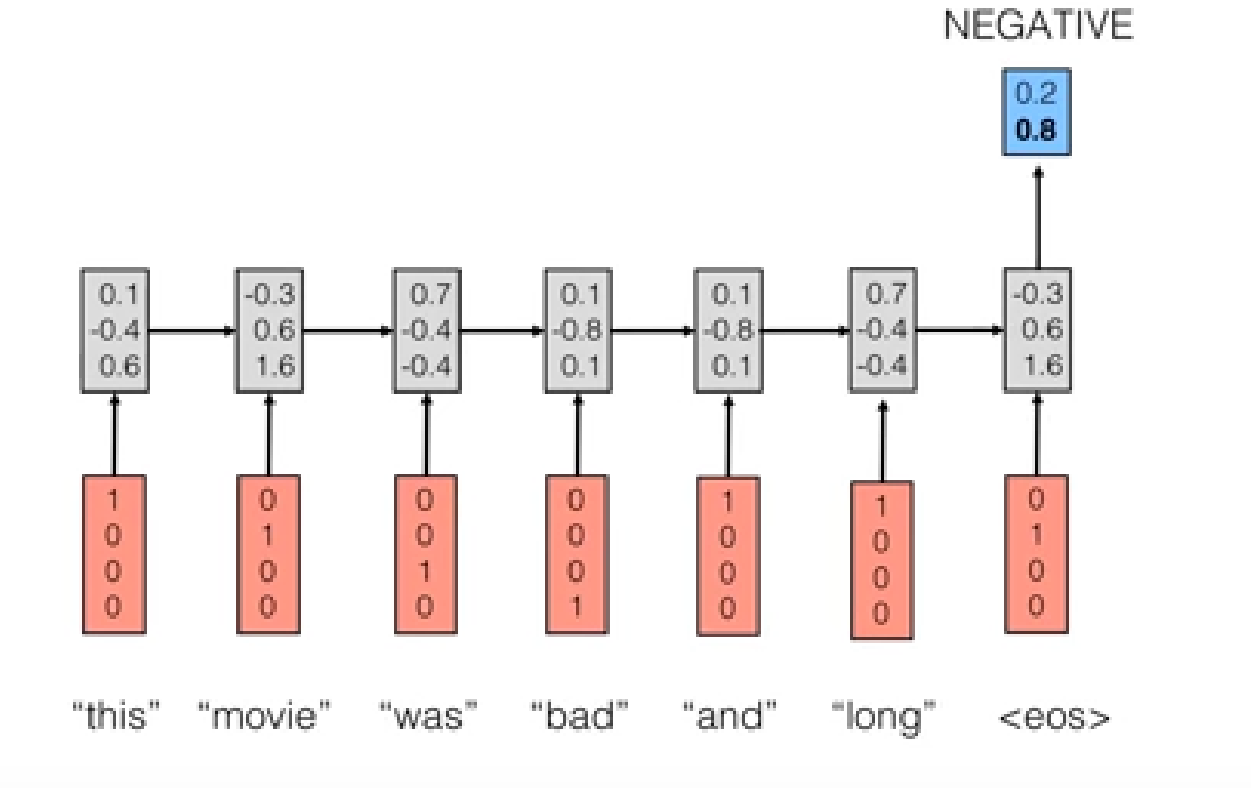
Recurrent Neural Networks:What is it good for?



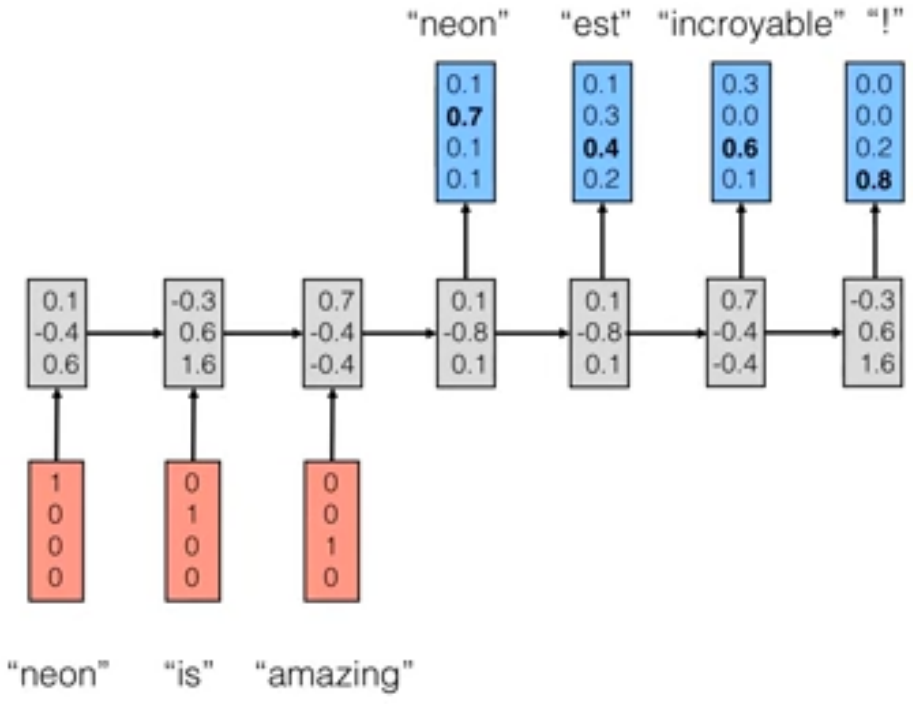
Learned a language model!

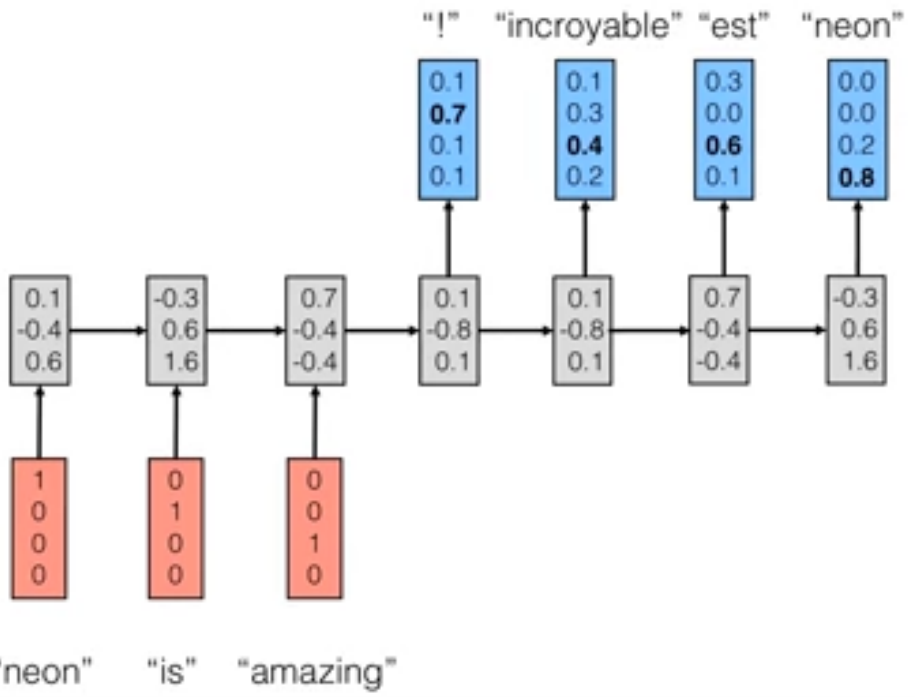
$$P(c_t|\{c_{t-1}, c_{t-2}, \ldots, c_0\})$$

Recurrent Neural Networks:What is it good for?

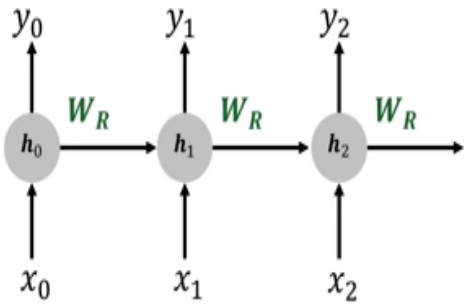


Recurrent Neural Networks:What is it good for?





Recurrent Neural Networks:Training

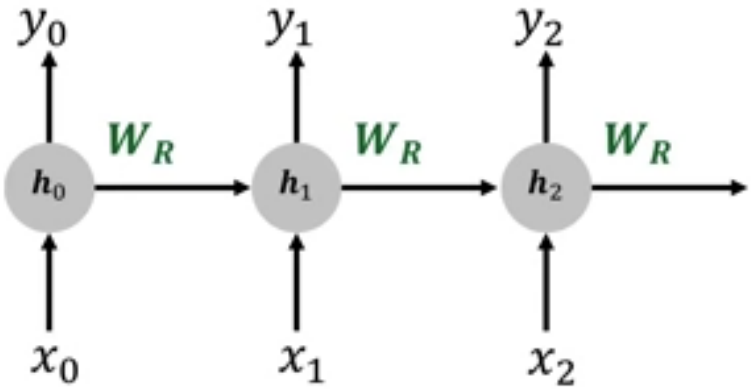


- Even though it looks like a regular RNN, these weights are shared across time steps.

Recurrent Neural Networks:Training

$$\boldsymbol{h}^{(t)} = g_h(W_{\mathrm{I}}\boldsymbol{x}^{(t)} + W_{\mathrm{R}}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

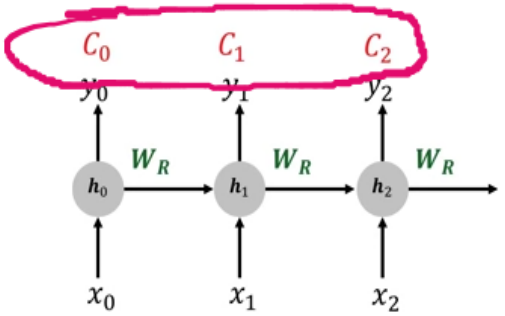
$$\boldsymbol{y}^{(t)} = g_y(W_y\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$



Recurrent Neural Networks:Training

$$\boldsymbol{h}^{(t)} = g_h(W_I \boldsymbol{x}^{(t)} + W_R \boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

$$\boldsymbol{y}^{(t)} = g_y(W_y \boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$



- The other difference being, the way cost is calculated
- With RNNs the network is run through end, error calculated, and then error back propagated.
- Here the network is emiting an output at every time step.

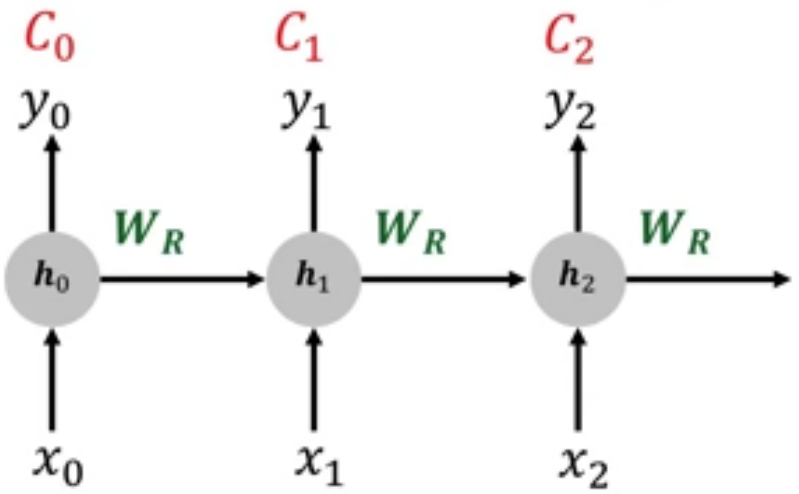
Recurrent Neural Networks:Training

$$\boldsymbol{h}^{(t)} = g_h(W_{\mathrm{I}}\boldsymbol{x}^{(t)} + W_{\mathrm{R}}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

$$\boldsymbol{y}^{(t)} = g_y(W_y\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$

Combine via:

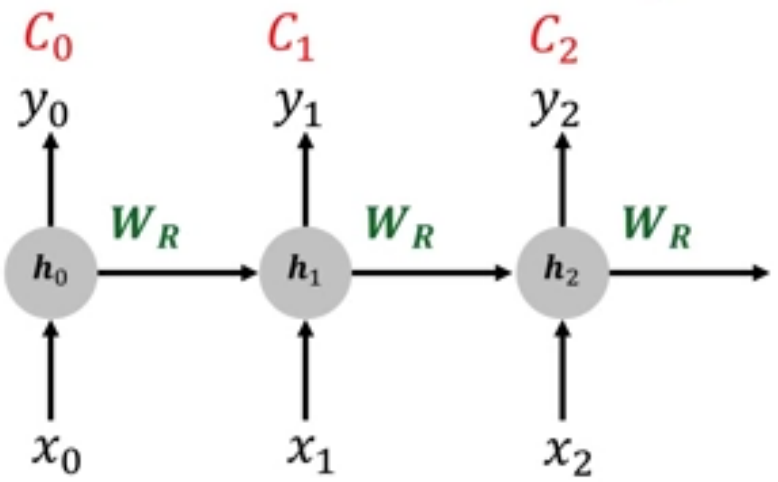
$$\frac{\partial C}{\partial W_R} = \sum_t \frac{\partial C_t}{\partial W_R}$$



Recurrent Neural Networks:Training

$$\boldsymbol{h}^{(t)} = g_h(W_{\mathrm{I}}\boldsymbol{x}^{(t)} + W_{\mathrm{R}}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

$$\boldsymbol{y}^{(t)} = g_y(W_y\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$



Combine via:

$$\frac{\partial C}{\partial W_R} = \sum_t \frac{\partial C_t}{\partial W_R}$$

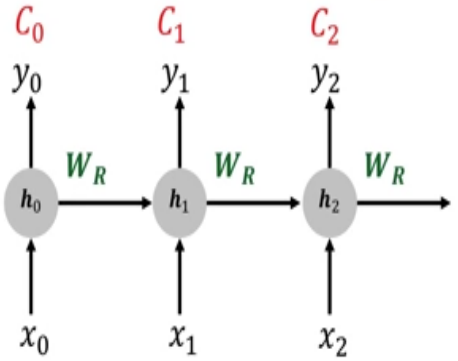
Example gradient:

$$\frac{\partial C_2}{\partial W_R} = \frac{\partial C_2}{\partial y_2}\frac{\partial y_2}{\partial h_2}\frac{\partial h_2}{\partial g}\frac{\partial g}{\partial a}\frac{\partial a}{\partial W_R}$$

Recurrent Neural Networks:Training

$$\boldsymbol{h}^{(t)} = g_h(W_{\mathrm{I}}\boldsymbol{x}^{(t)} + W_{\mathrm{R}}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h)$$

$$\boldsymbol{y}^{(t)} = g_y(W_y\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$



Combine via:

$$\frac{\partial C}{\partial W_R} = \sum_t \frac{\partial C_t}{\partial W_R}$$

Example gradient:

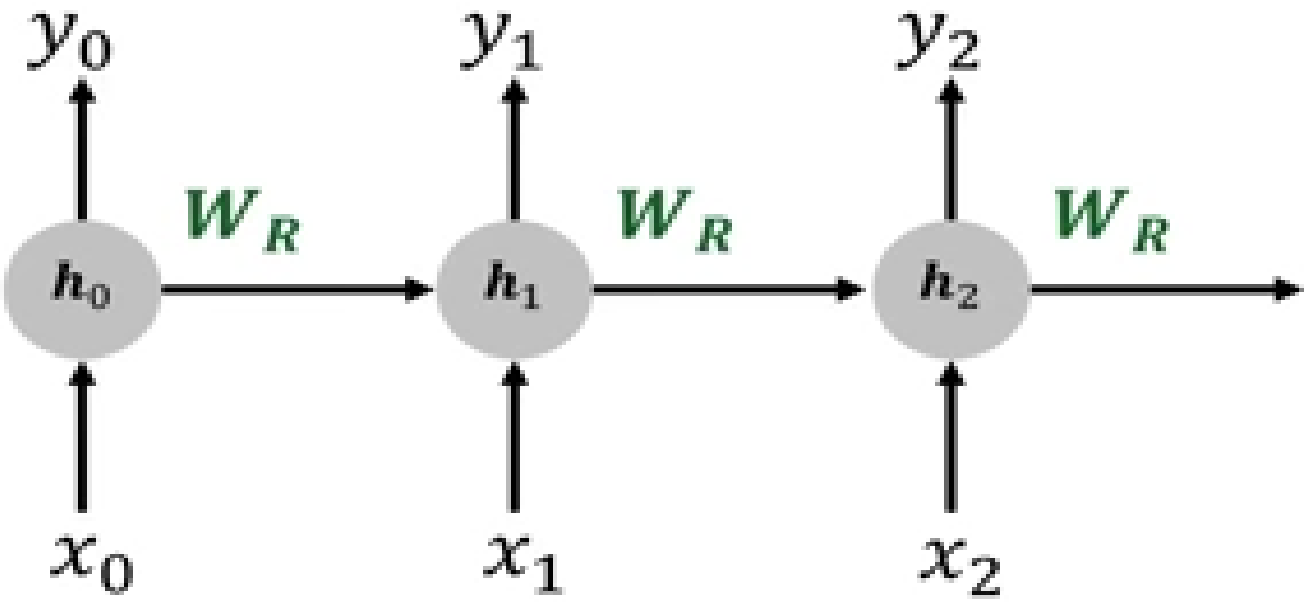
$$\frac{\partial C_2}{\partial W_R} = \frac{\partial C_2}{\partial y_2}\frac{\partial y_2}{\partial h_2}\frac{\partial h_2}{\partial g}\frac{\partial g}{\partial a}\frac{\partial a}{\partial W_R}$$

$$a = (W_{\mathrm{I}}x_2 + W_{\mathrm{R}}h_1 + b_h)$$

Depends on W_R too!

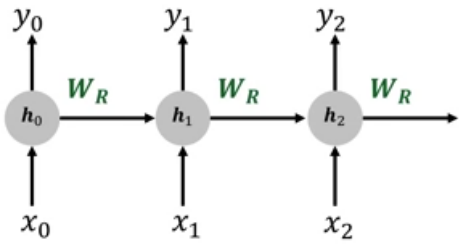
- The gradient has to be propagated through to the time step.

Recurrent Neural Networks:Training:Back Propagation: Vanishing/Exploding Gradient



$$\frac{\partial C_{100}}{\partial W_R} = \frac{\partial C_{100}}{\partial y_{100}} \dots W_R \frac{\partial g_{100}}{\partial a_{100}} \dots W_R \frac{\partial g_{99}}{\partial a_{99}} \dots$$

Recurrent Neural Networks:Training:Back Propagation: Vanishing/Exploding Gradient

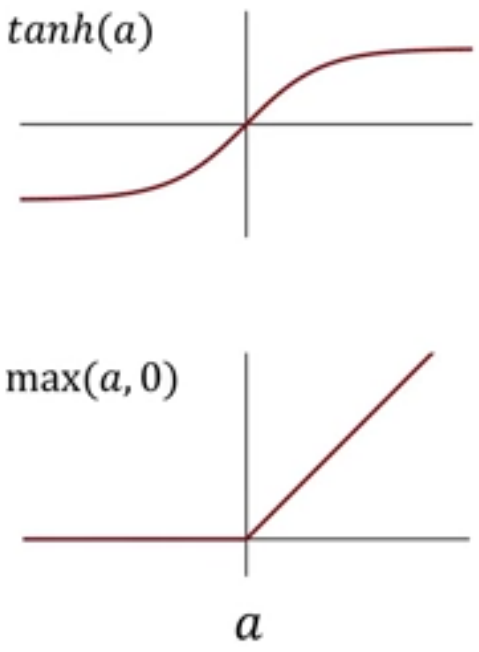


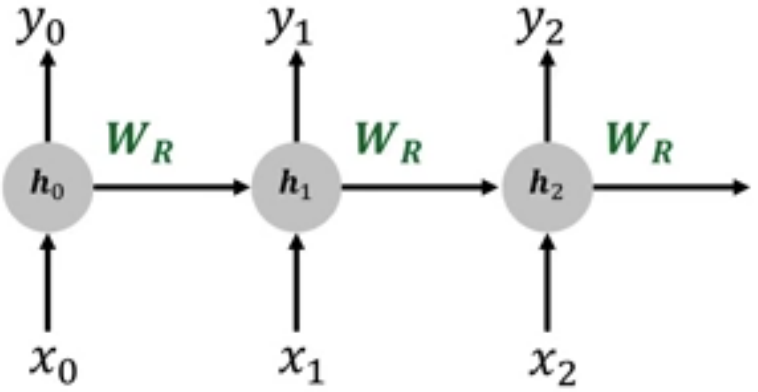
$$\frac{\partial C_{100}}{\partial W_R} = \frac{\partial C_{100}}{\partial y_{100}} \dots W_R \frac{\partial g_{100}}{\partial a_{100}} \dots W_R \frac{\partial g_{99}}{\partial a_{99}} \dots$$

$$\frac{\partial C_T}{\partial W_R} \propto |W_R|^T \left|\frac{\partial g}{\partial a}\right|^T$$

- if W_R is less than one than the derivative is going to be close to 0.
- Alternatively, if it is greater than 0 then we have an exploding gradient.

Recurrent Neural Networks:Training:Back Propagation: Vanishing/Exploding Gradient





$$\frac{\partial C_{100}}{\partial W_R} = \frac{\partial C_{100}}{\partial y_{100}} \dots W_R \frac{\partial g_{100}}{\partial a_{100}} \dots W_R \frac{\partial g_{99}}{\partial a_{99}} \dots$$

$$\frac{\partial C_T}{\partial W_R} \propto |W_R|^T \left|\frac{\partial g}{\partial a}\right|^T$$

Recurrent Neural Networks:Training:Back Propagation: Vanishing/Exploding Gradient

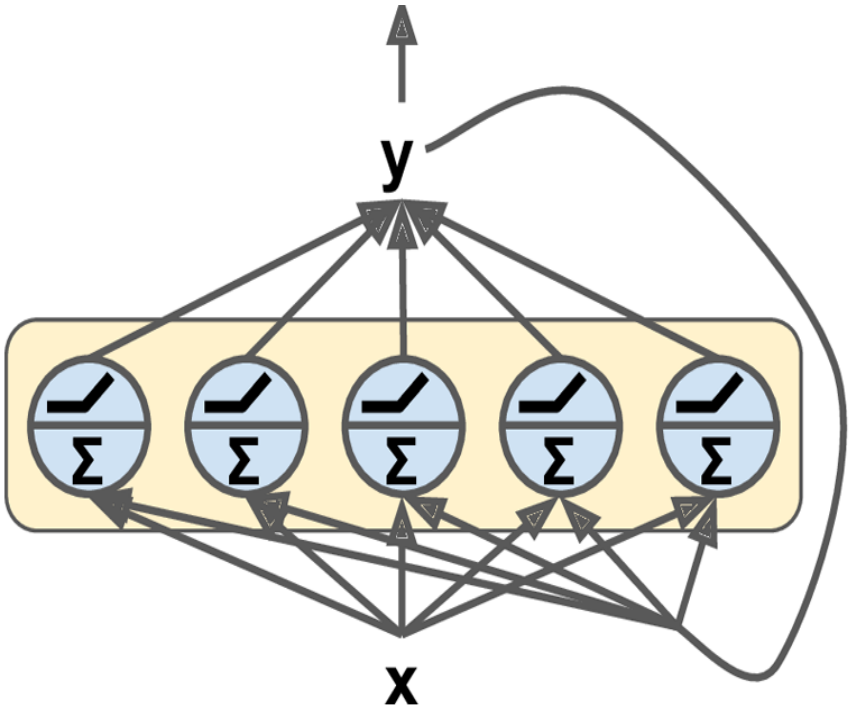
1. Exploding gradients
    - Truncated BPTT
    - Clip gradients at threshold
    - RMSprop to adjust learning rate

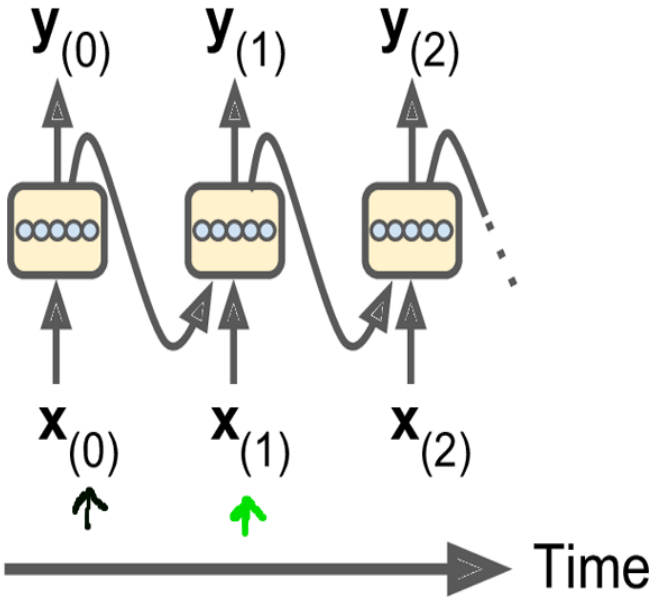
2. Vanishing gradients
    - Harder to detect
    - Weight initialization
    - ReLu activation functions
    - RMSprop
    - LSTM, GRUs

Recurrent Neural Networks:Manual

```
tf.reset_default_graph()

n_inputs = 3
n_neurons = 5

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)

init = tf.global_variables_initializer()

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

with tf.Session() as sess:

    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})


print("Y0_val:",Y0_val)
print("Y1_val:",Y1_val)
```

Recurrent Neural Networks:Manual





$[4\times3]\ [3\times5]\ \tanh[4\times5]$

$x_0 \times W_x + b = Y_0$

t_0

$[4\times5]\ [3\times5] \quad [4\times5]\ [5\times5] + b$

$x_1 \times W_x + Y_0 \times W_y$

$= \tanh[4\times5]$

Recurrent Neural Networks:static unrolling

```
n_inputs = 3
n_neurons = 5

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

init = tf.global_variables_initializer()

X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]])
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]])

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})

print("Y0_val:",Y0_val)
print("Y1_val:",Y1_val)

show_graph(tf.get_default_graph())
```

- Creates the same shapes as before
- This program is identical to previous one.

Recurrent Neural Networks:Static Unrolling

```
tf.reset_default_graph()

n_steps = 2
n_inputs = 3
n_neurons = 5

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])

init = tf.global_variables_initializer()

X_batch = np.array([
        # t = 0      t = 1
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])

with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})

print(np.transpose(outputs_val, axes=[1, 0, 2])[1])
```

- Again identical to previous one except the way we feed data.
- unstack extract the python list of tensors along the first dimension.

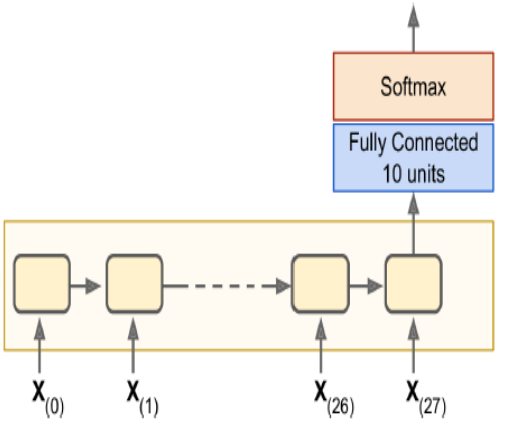
Recurrent Neural Networks:Dynamic Unrolling

```
tf.reset_default_graph()

n_steps = 2
n_inputs = 3
n_neurons = 5

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)

init = tf.global_variables_initializer()
X_batch = np.array([
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])

with tf.Session() as sess:
    init.run()
    print("outputs =", outputs.eval(feed_dict={X: X_batch}))

show_graph(tf.get_default_graph())
```

- dynamic RNN does the unrolling using a while loop.

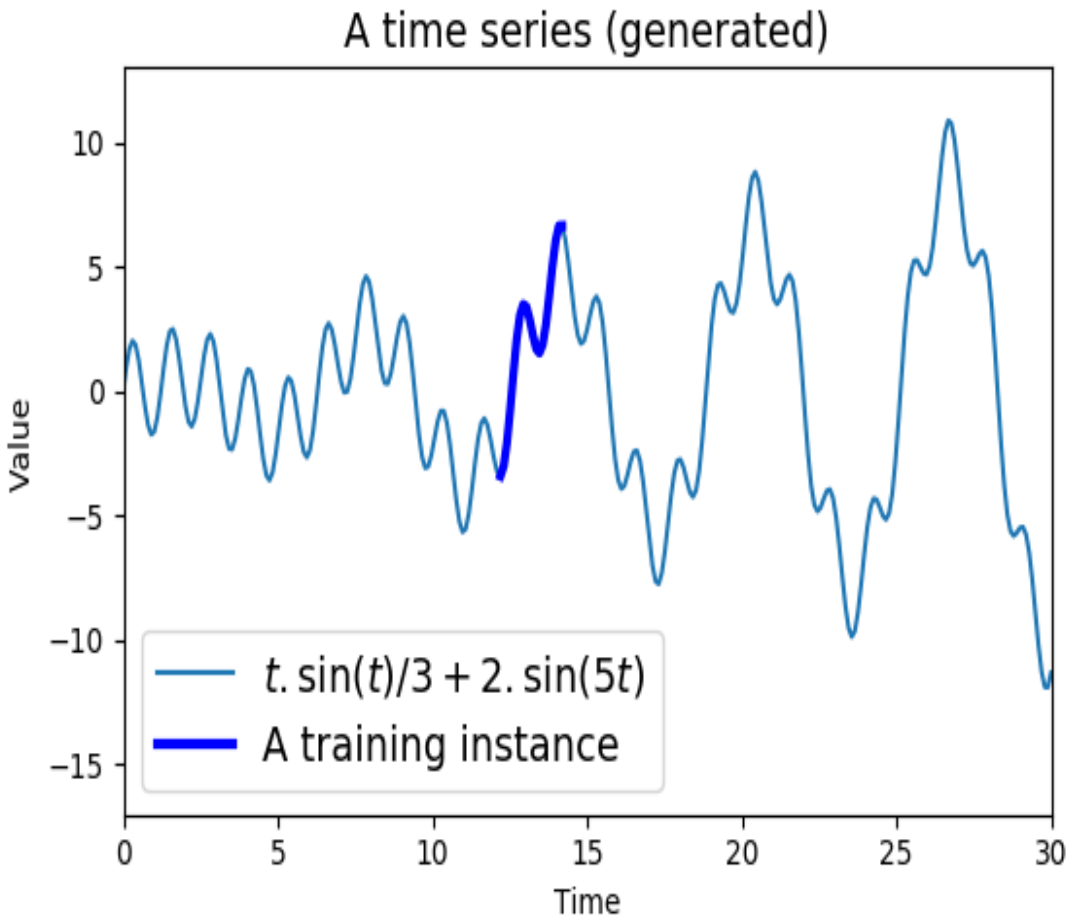
Recurrent Neural Networks:Sequence Classifier

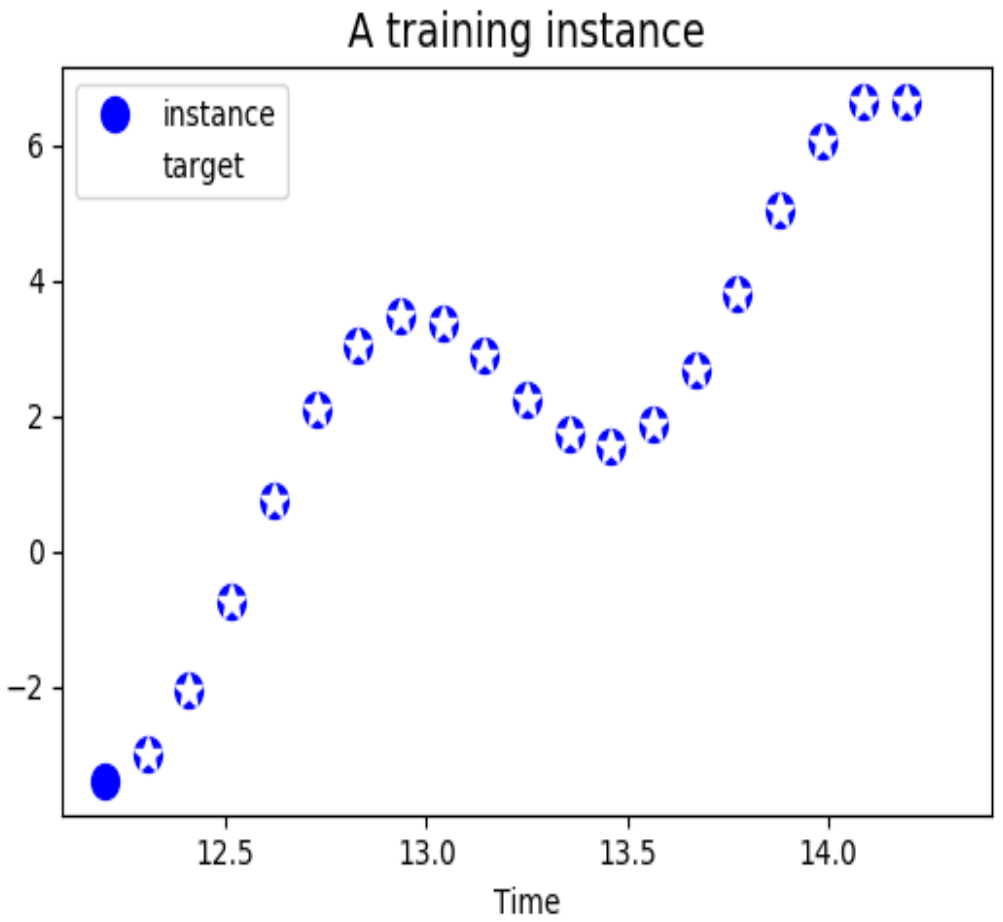
```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

with tf.variable_scope("rnn", initializer=tf.contrib.layers.variance_scaling_initializer()):
    basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
    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()

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
X_test = mnist.test.images.reshape((-1, n_steps, n_inputs))
y_test = mnist.test.labels

n_epochs = 100
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((-1, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
```



- 28x28 pixel of each MNIST image treated as a sequence.
- CNN is a better solution but getting started with something familiar.

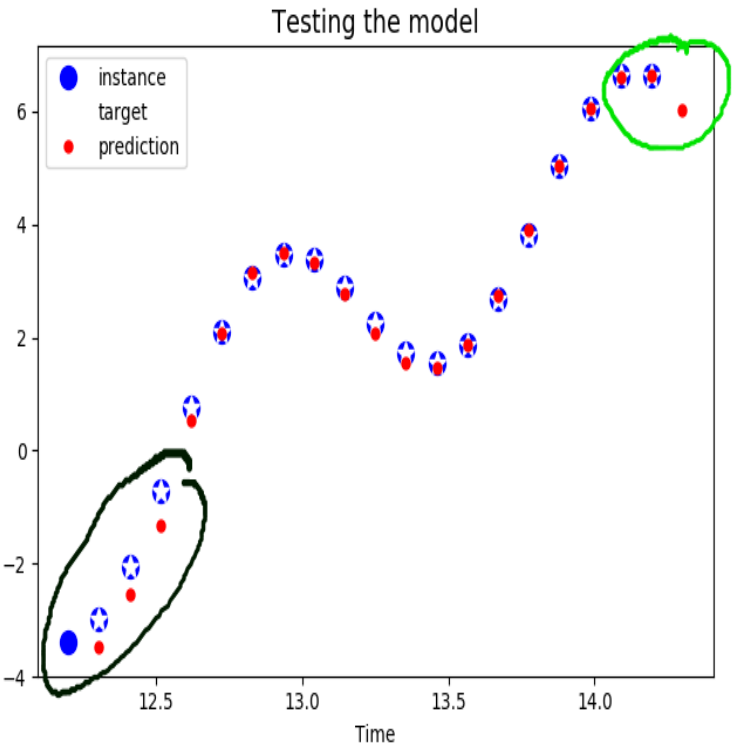
Recurrent Neural Networks:Time series





Recurrent Neural Networks:Time series:OutputProjectionWrapper

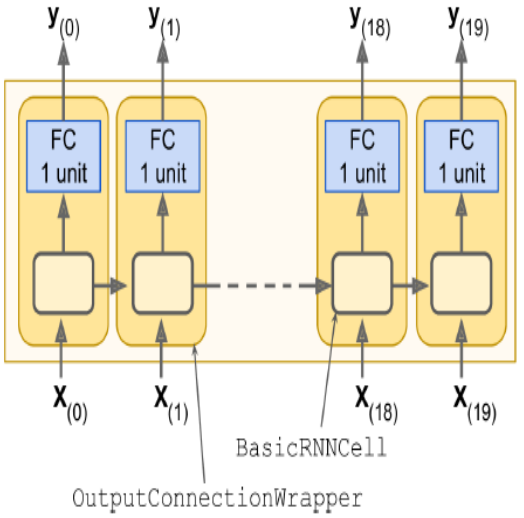
```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])

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)

n_outputs = 1
learning_rate = 0.001

loss = tf.reduce_sum(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()

n_iterations = 1000
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

    X_new = time_series(np.array(t_instance[:-1].reshape(-1, n_steps, n_inputs)))
    y_pred = sess.run(outputs, feed_dict={X: X_new})
    print(y_pred)
```



. A little of in the beginning.

. But makes correct predictions after some training.

Recurrent Neural Networks:Time series:OutputProjectionWrapper

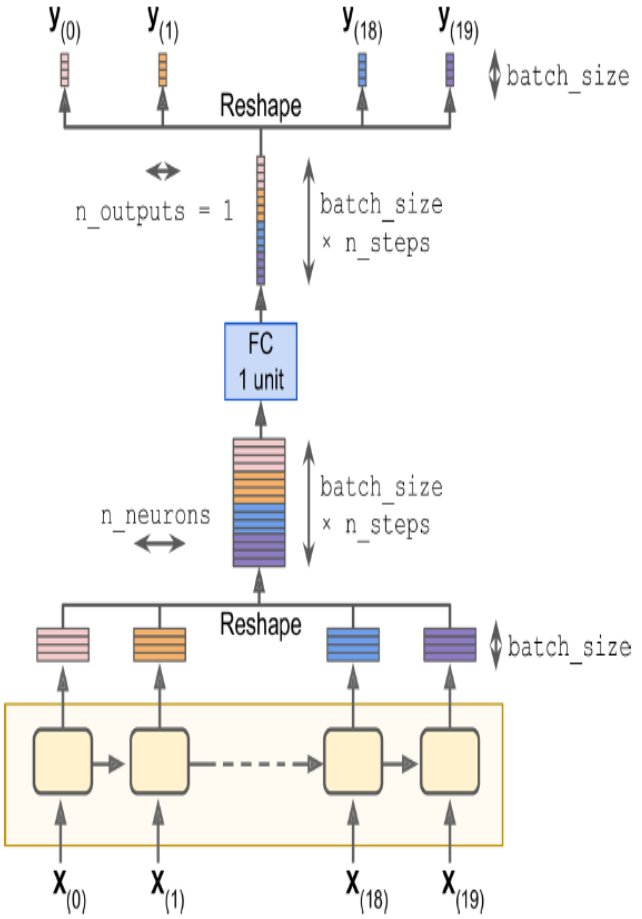


- At each time step we have an output vector of size 100. What we need is a single output at each time step.
- OutputProjectionWrapper - It adds a Fully connected unit without an activation function.

Recurrent Neural Networks:Time series

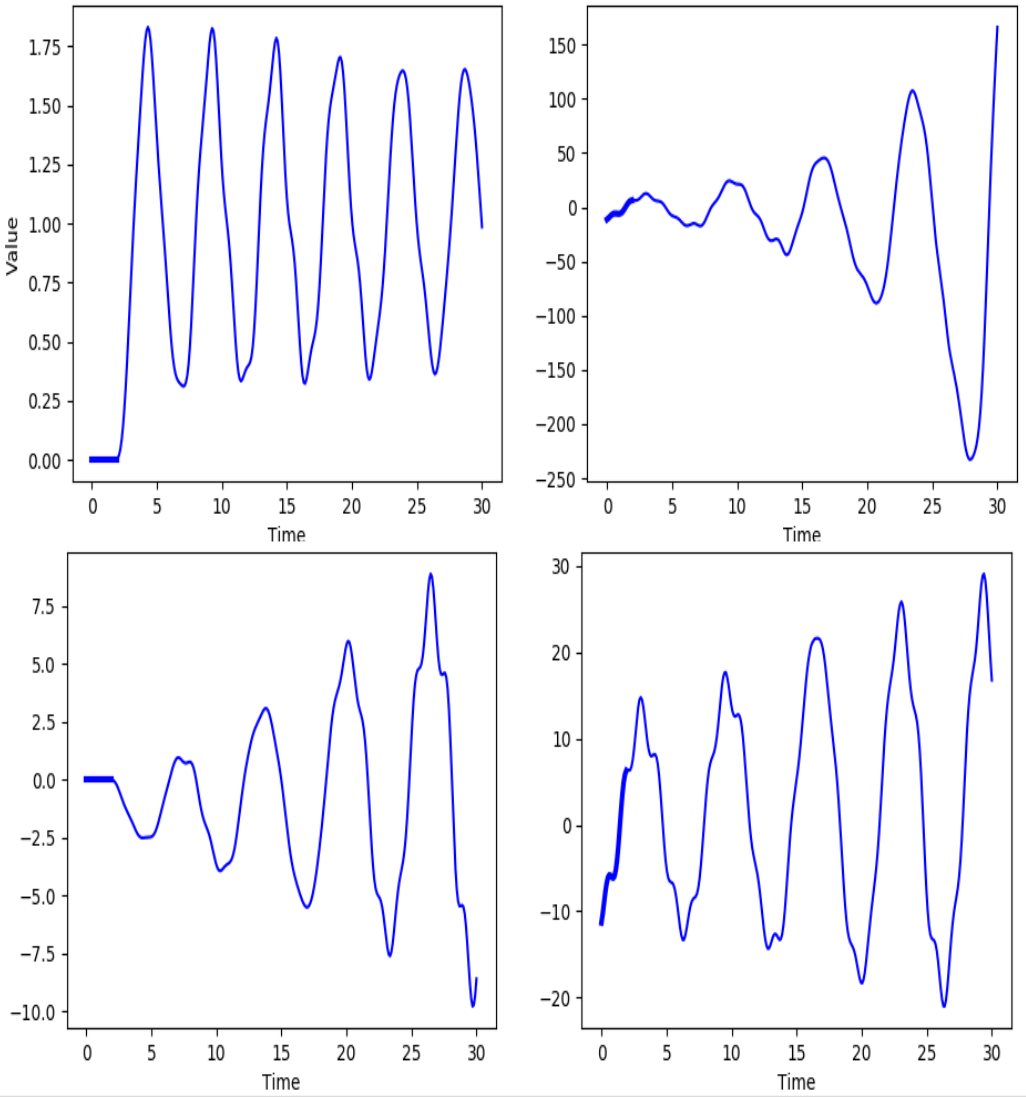
```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
rnn_outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

learning_rate = 0.001

stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_neurons])
stacked_outputs = tf.layers.dense(stacked_rnn_outputs, n_outputs)
outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])

loss = tf.reduce_sum(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()

n_iterations = 1000
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

    X_new = time_series(np.array(t_instance[:-1].reshape(-1, n_steps, n_inputs)))
    y_pred = sess.run(outputs, feed_dict={X: X_new})
    print(y_pred)
```

- More optimized solution without the OPW.



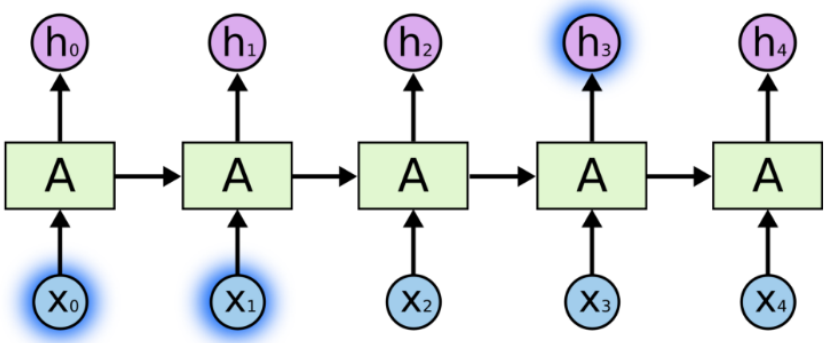
Recurrent Neural Networks:Time series

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

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
rnn_outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
learning_rate = 0.001

stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_neurons])
stacked_outputs = tf.layers.dense(stacked_rnn_outputs, n_outputs)
outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])

loss = tf.reduce_sum(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
init = tf.global_variables_initializer()

n_iterations = 2000
batch_size = 50
with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

    sequence1 = [0. for i in range(n_steps)]
    for iteration in range(len(t) - n_steps):
        X_batch = np.array(sequence1[-n_steps:]).reshape(1, n_steps, 1)
        y_pred = sess.run(outputs, feed_dict={X: X_batch})
        sequence1.append(y_pred[0, -1, 0])

    sequence2 = [time_series(i * resolution + t_min + (t_max-t_min/3)) for i in range(n_steps)]
    for iteration in range(len(t) - n_steps):
        X_batch = np.array(sequence2[-n_steps:]).reshape(1, n_steps, 1)
        y_pred = sess.run(outputs, feed_dict={X: X_batch})
        sequence2.append(y_pred[0, -1, 0])
```



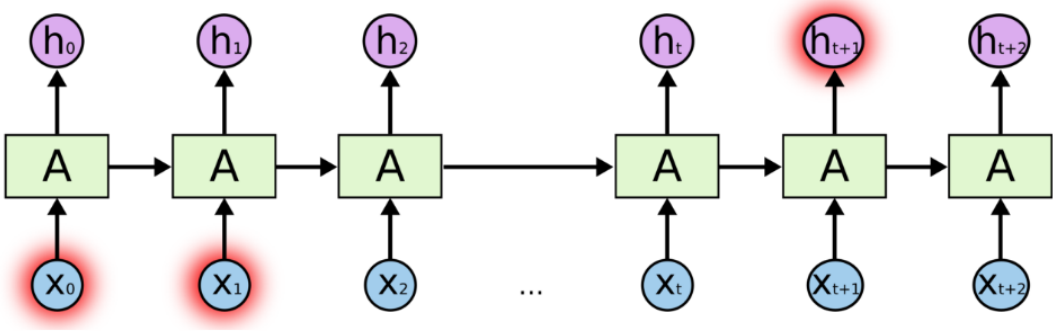
• new creative sequences.

LSTM(Long Short Term Memory):Why



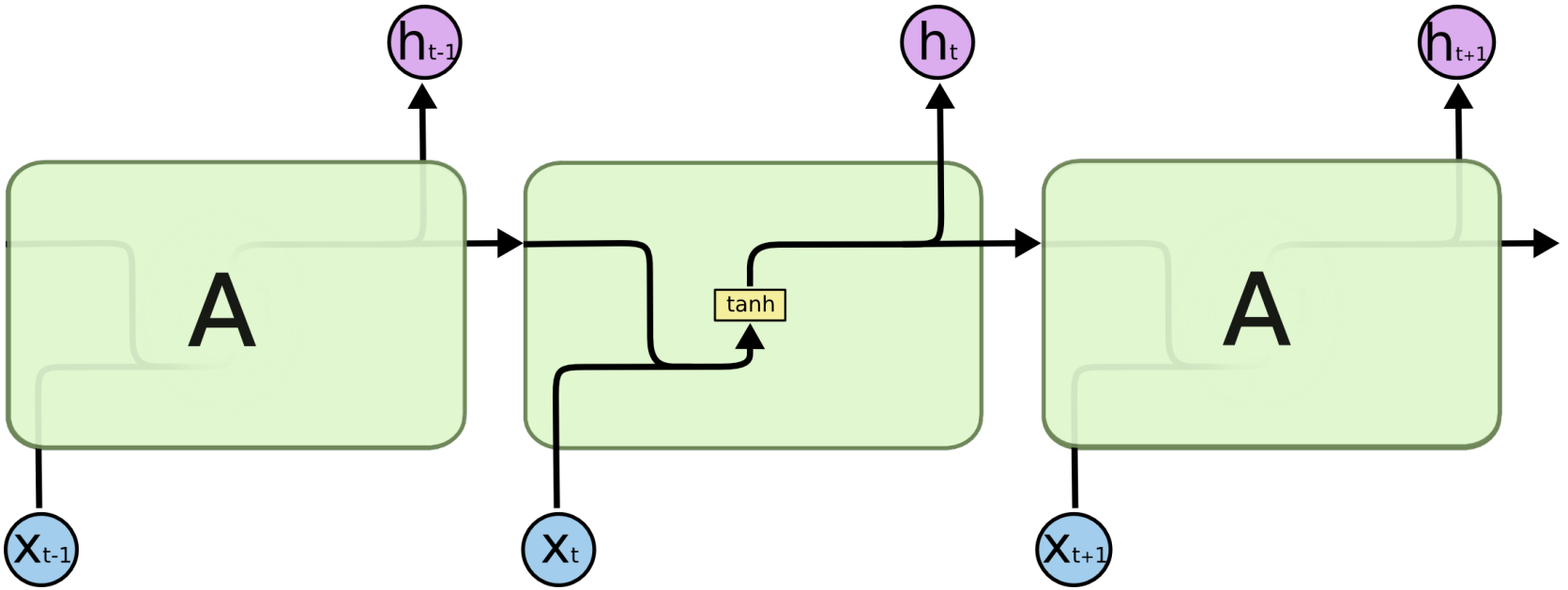
• If the RNN is trying to predict the last word in "the clouds are in the sky", it does not need any further content.

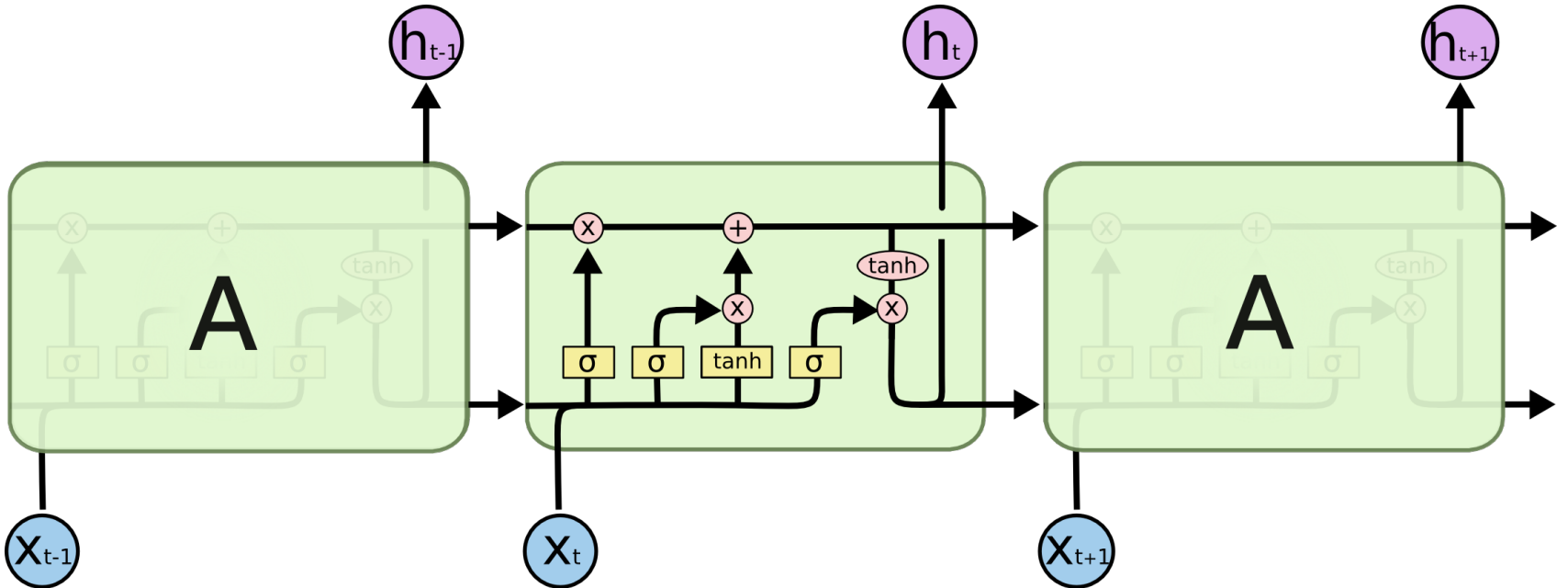
LSTM(Long Short Term Memory):Why



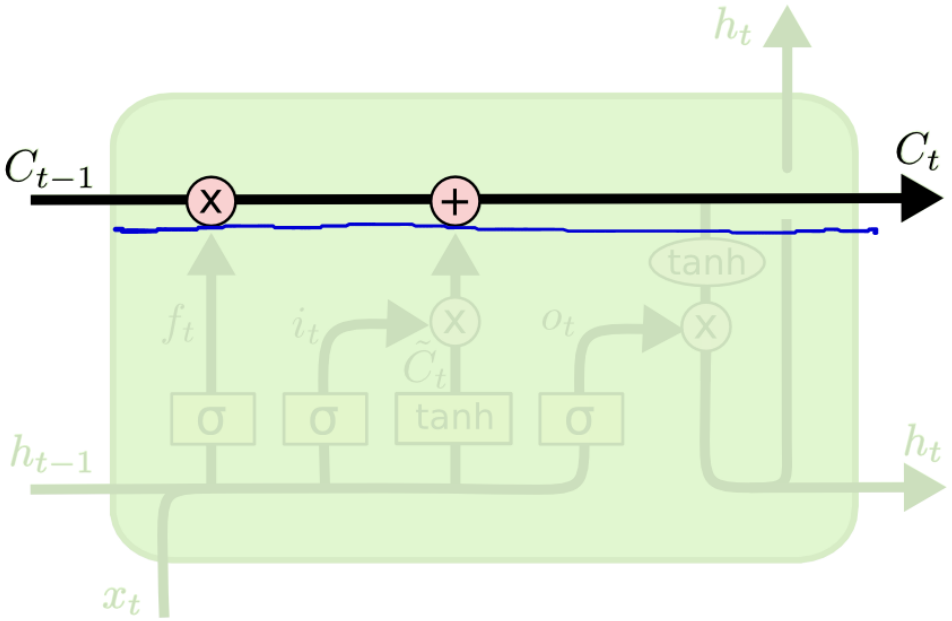
- But there are cases when the RNN would require more content than available in the unrolled RNN.
- Consider "I grew up in France..... I speak fluent French." For this RNN would need the content of "France from further back.

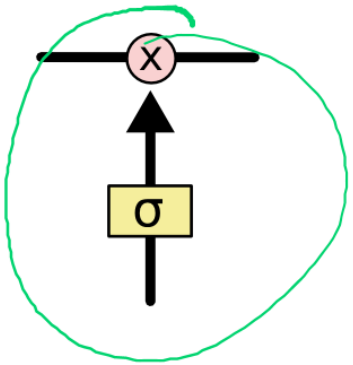
LSTM(Long Short Term Memory):Why





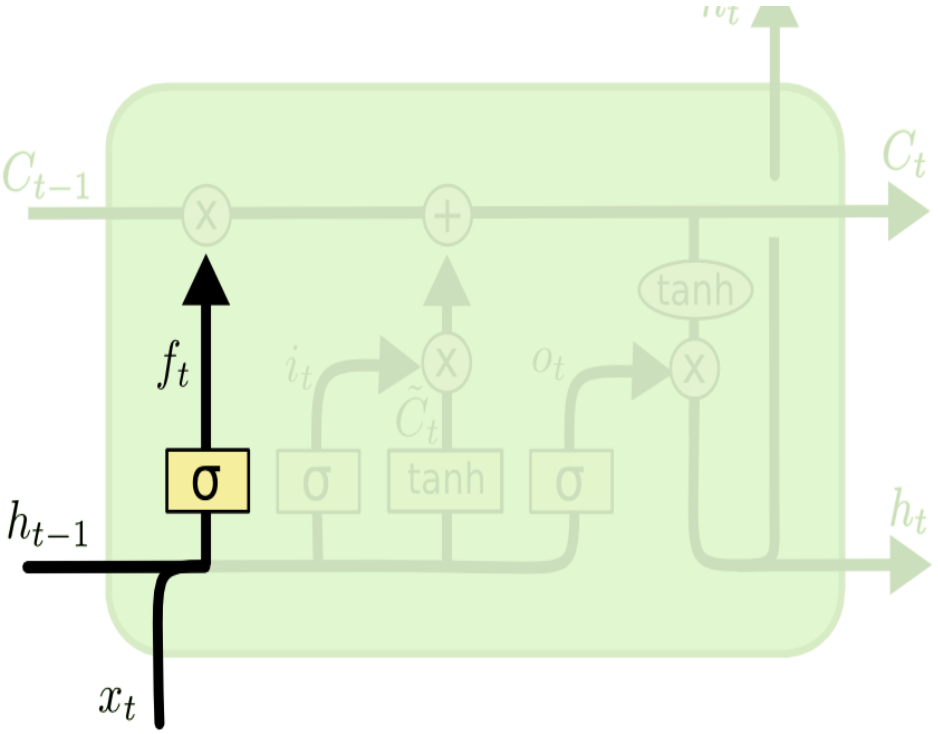
LSTM(Long Short Term Memory):Core Idea





- The key idea to LSTM is the cell state.
- The LSTM has the ability to remove or add information carefully regulated by structures called gates.
- This is composed of sigmoid function. A value of '0' let nothing through and vice versa.

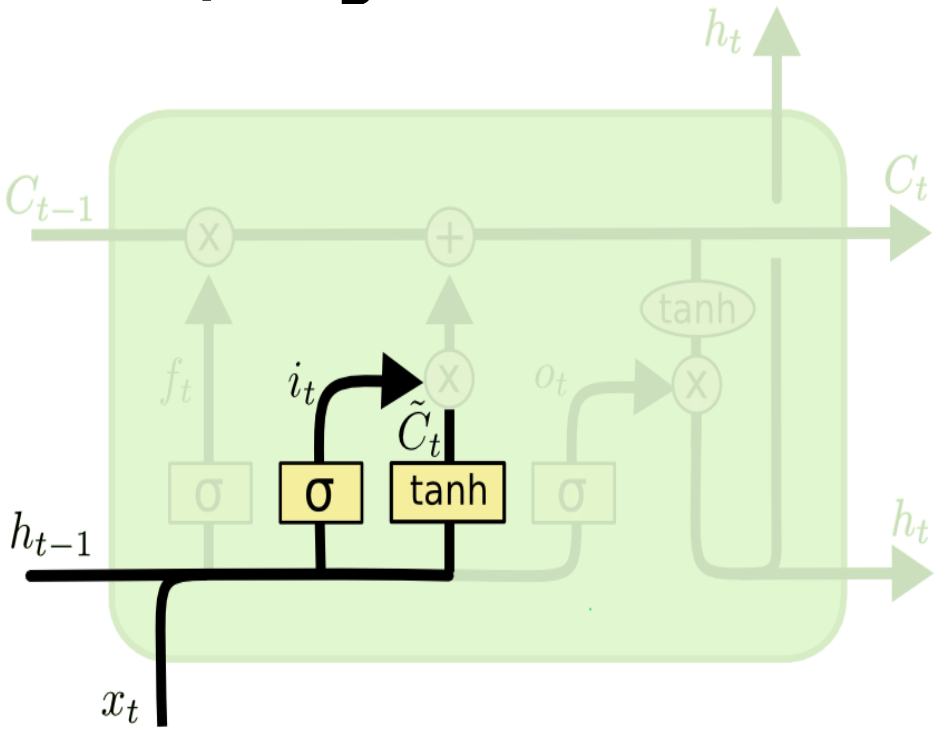
LSTM(Long Short Term Memory):Core Idea



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- Forget gate layer - It decides what information is irrelevant to current subject and decides to forget it.

LSTM(Long Short Term Memory):Core Idea



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

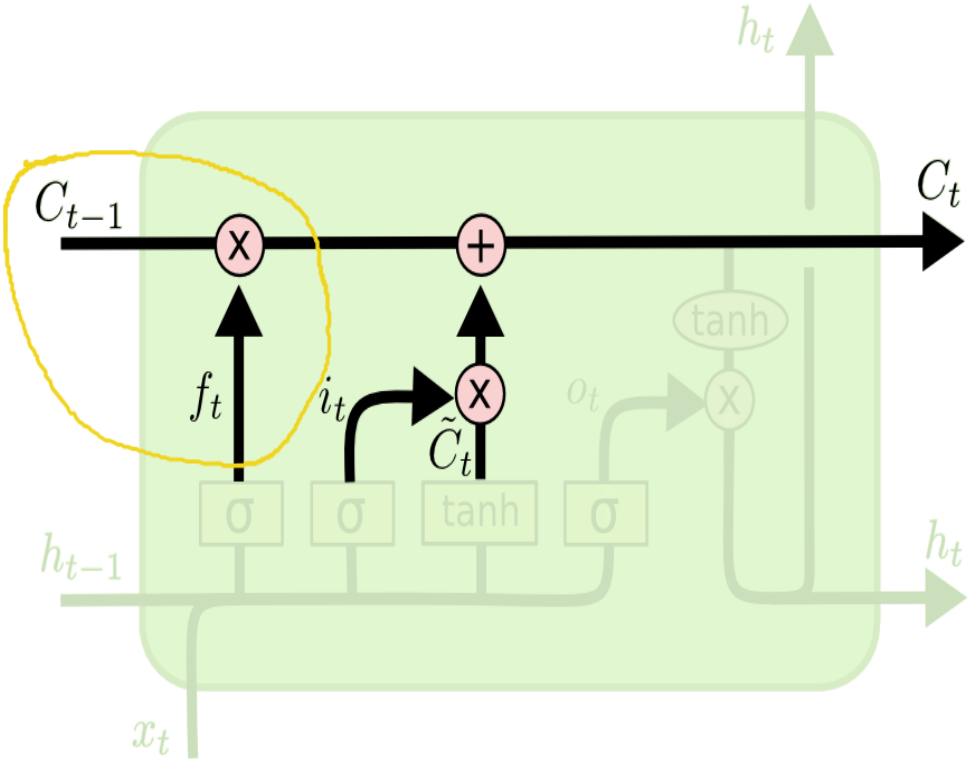
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

- Next step is to decide what information to store

1. Input gate layer which values it is going to update (i_t).
2. tanh layer create a vector of new candidate value ($\tilde{C}_t$), that could be added to state.

(Probably the details of subject.

LSTM(Long Short Term Memory):Core Idea



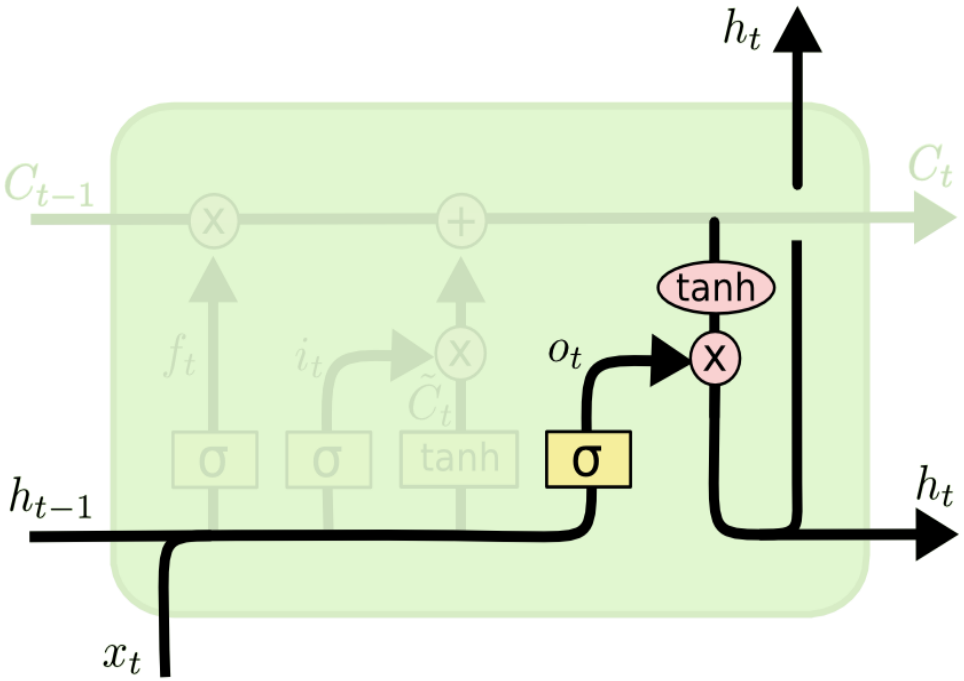
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

scale of how much

Candidate value

- Now time to update the old cell state (C_{t-1}) into the new cell state (C_t).
- Forget what we decided to forget earlier.

LSTM(Long Short Term Memory):Core Idea

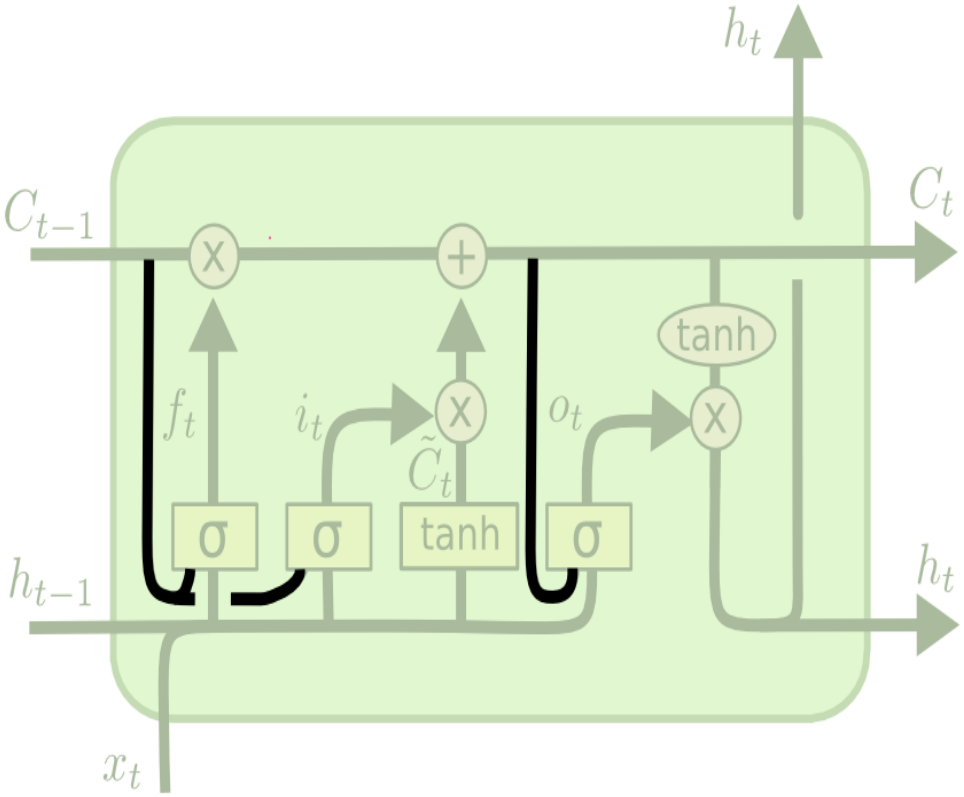


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

- Finally we decide what to output.
- It is cell state but a filtered version.
- For a language model, it might want to use a pronoun relevant to the gender of the subject.

LSTM(Long Short Term Memory):Variants



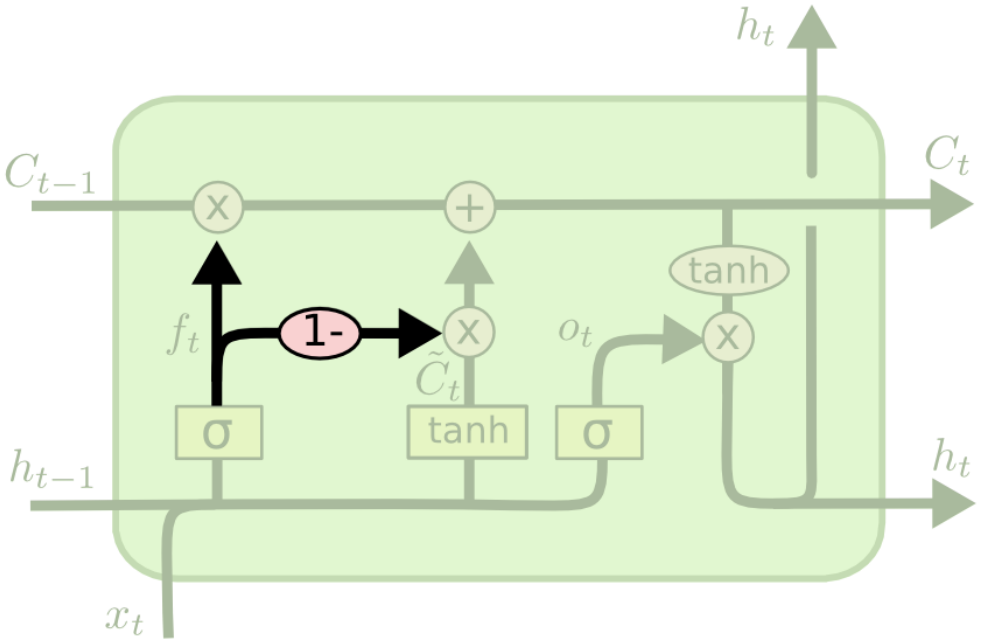
$$f_t = \sigma\left(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i\right)$$

$$o_t = \sigma\left(W_o \cdot [C_t, h_{t-1}, x_t] + b_o\right)$$

- There are many variants.
  - Peep hole version allows gates to look at the cell state.

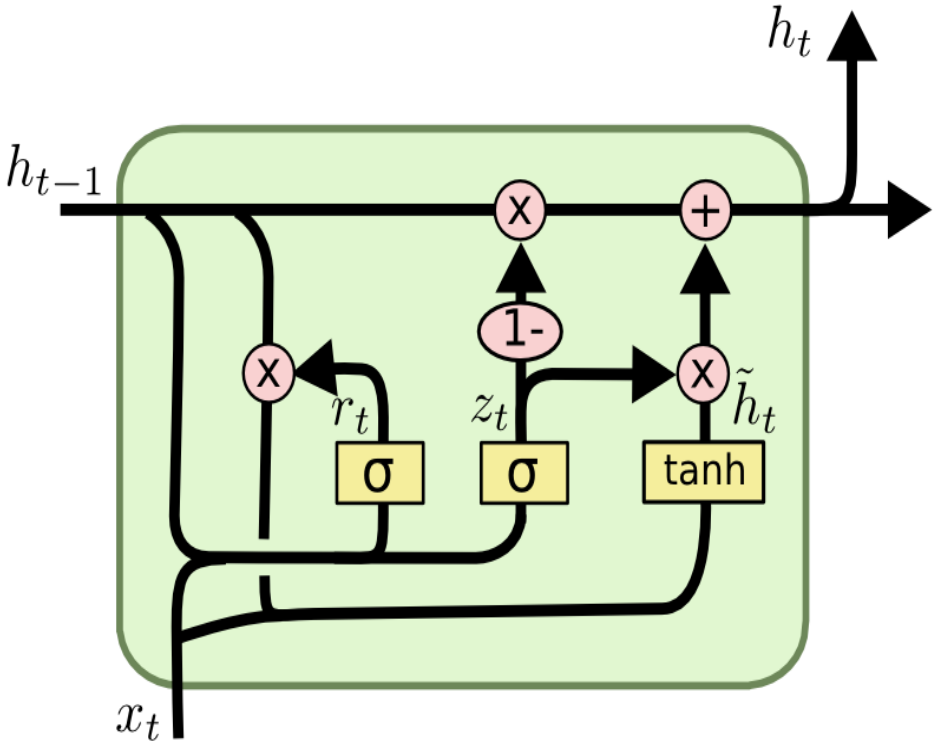
LSTM(Long Short Term Memory):Variants



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

- Another variant couples the forget and input gates.

LSTM(Long Short Term Memory):Variants



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- GRU - It combines forget and the input gate.
- It also combines the hidden state with the cell state.

LSTM(Long Short Term Memory):Tensorflow

```
def lstm_cell():
    cell = tf.contrib.rnn.LSTMCell(n_neurons, reuse=tf.get_variable_scope().reuse,use_peepholes=True)
    return tf.contrib.rnn.DropoutWrapper(cell, output_keep_prob=0.8)

def gru_cell():
    cell = tf.contrib.rnn.GRUCell(n_neurons)
    return tf.contrib.rnn.DropoutWrapper(cell, output_keep_prob=0.8)

multi_cell = tf.contrib.rnn.MultiRNNCell([lstm_cell() for _ in range(num_layers)], state_is_tuple = True)
outputs, states = tf.nn.dynamic_rnn(multi_cell, X, dtype=tf.float32)
top_layer_h_state = states[-1][1]
logits = tf.layers.dense(top_layer_h_state, n_outputs, name="softmax")
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
loss = tf.reduce_mean(xentropy, name="loss")
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()
print("states:",states)
print("top_layer_h_state:",top_layer_h_state)
n_epochs = 10
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((batch_size, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
        print("Epoch", epoch, "Train accuracy =", acc_train, "Test accuracy =", acc_test)
```