TensorFlow Tutorial #04

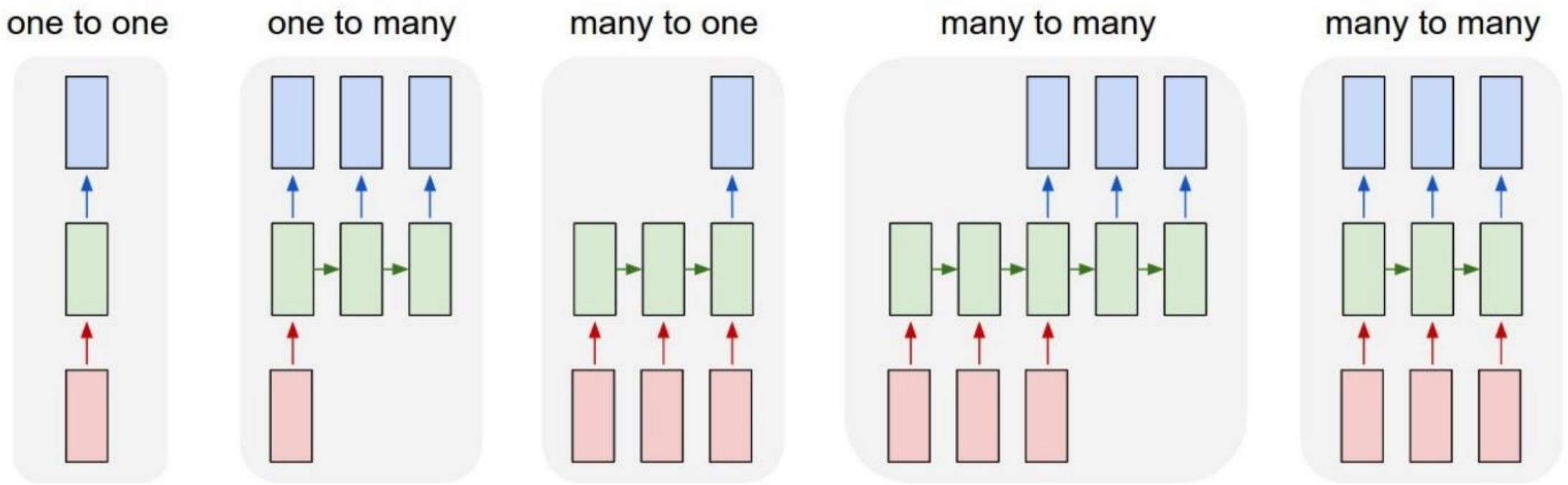
Recurrent Neural Network (RNN)

Seonghyeon Nam, Ph.D. Student
Computational Intelligence and Photography Lab.
Yonsei University

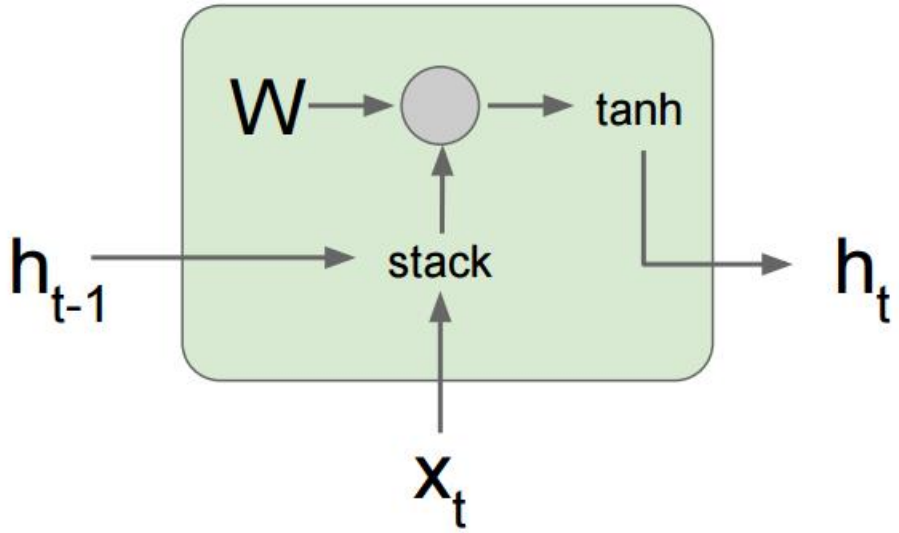
Acknowledgement

1. CS231n: Convolutional Neural Networks for Visual Recognition
   http://cs231n.stanford.edu/
2. CS 20SI: TensorFlow for Deep Learning Research
   http://web.stanford.edu/class/cs20si/
3. Hun Kim, DeepLearningZeroToAll
   http://hunkim.github.io/ml/
4. Sherjilozair, char-rnn-tensorflow
   https://github.com/sherjilozair/char-rnn-tensorflow

Recurrent Neural Networks



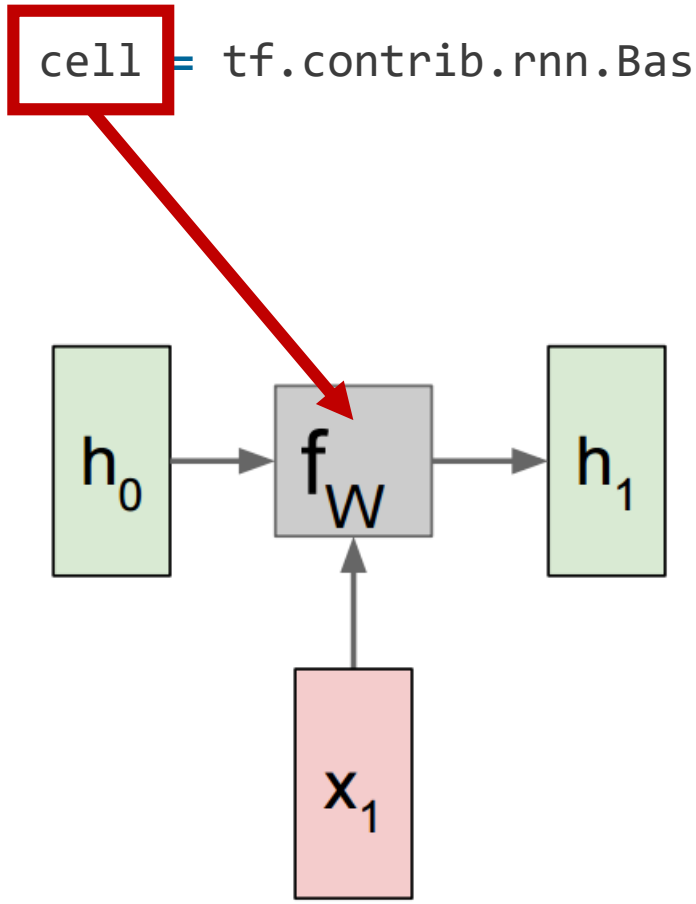
Vanilla RNN



$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)
\end{aligned}
$$

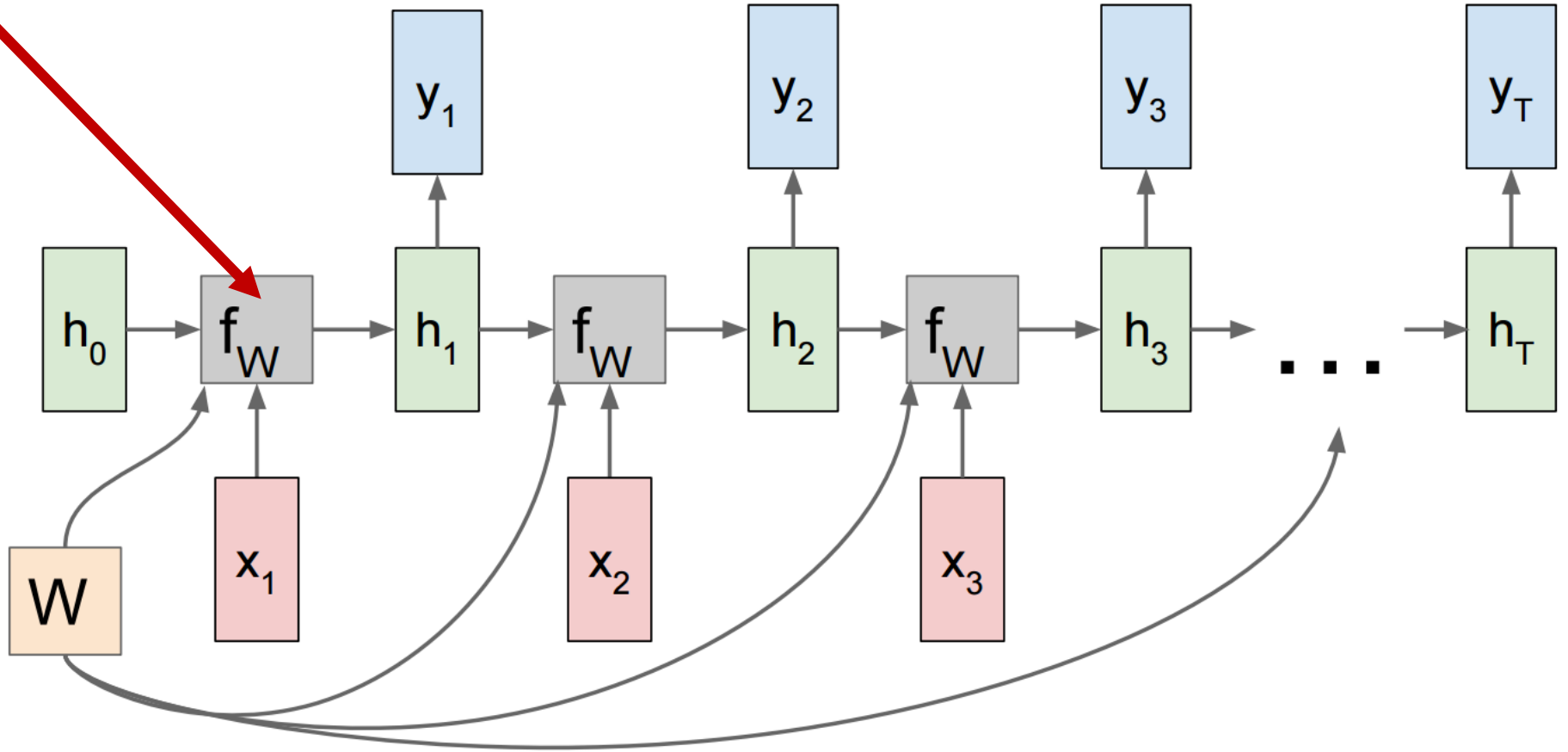
RNNs in TensorFlow

```
cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_size)
```



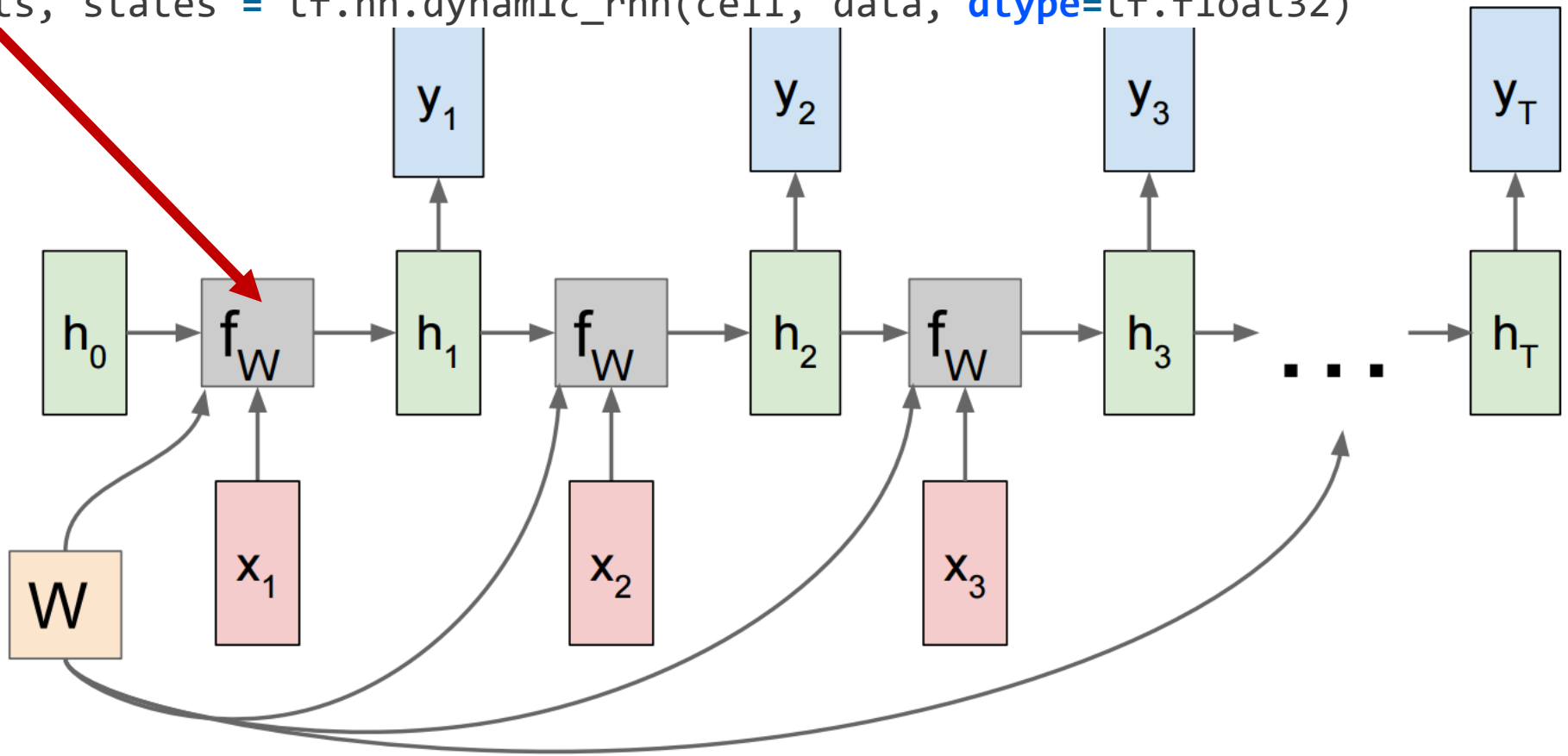
RNNs in TensorFlow

```
cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_size)
```



RNNs in TensorFlow

```
cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_size)

outputs, states = tf.nn.dynamic_rnn(cell, data, dtype=tf.float32)
```



Feeding My Data

Input tensor format:

```
data = [batch, time, input_size]
# batch: the number of examples in a batch
# time: sequence length
# input_size: the dimension of input vector
```

Feeding My Data

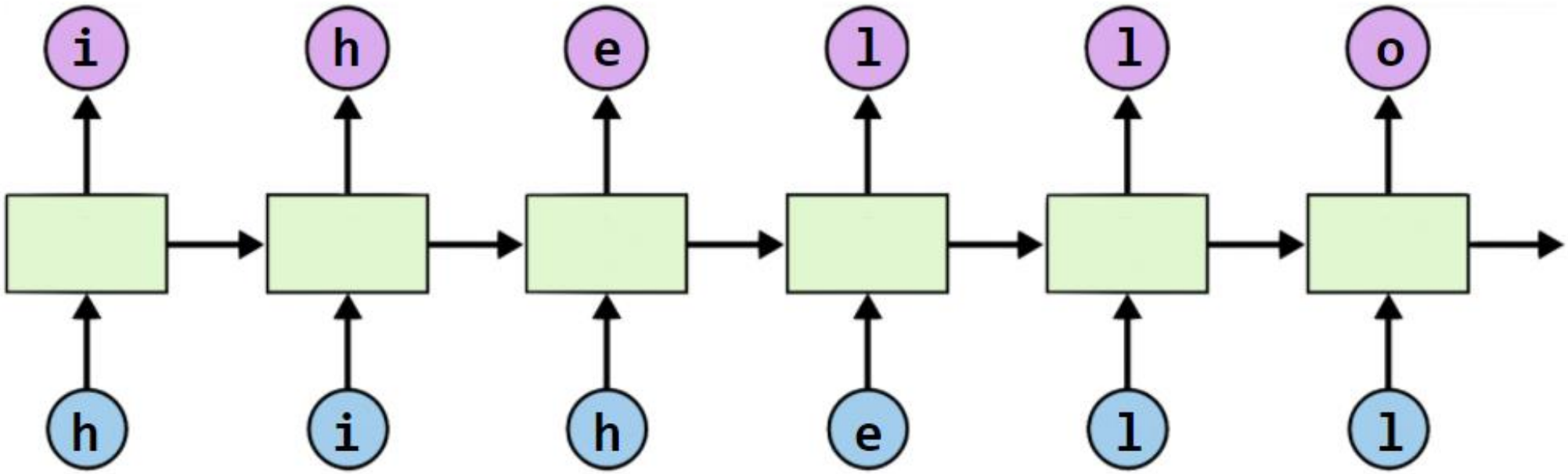
Input tensor format:

```
data = [batch, time, input_size]
# batch: the number of examples in a batch
# time: sequence length
# input_size: the dimension of input vector
```

In natural language processing,
input vector is usually encoded as one-hot vector

```
[1, 0, 0, 0, 0],   # h 0
[0, 1, 0, 0, 0],   # i 1
[0, 0, 1, 0, 0],   # e 2
[0, 0, 0, 1, 0],   # l 3
[0, 0, 0, 0, 1],   # o 4
```

Teaching RNN ‘hihello’

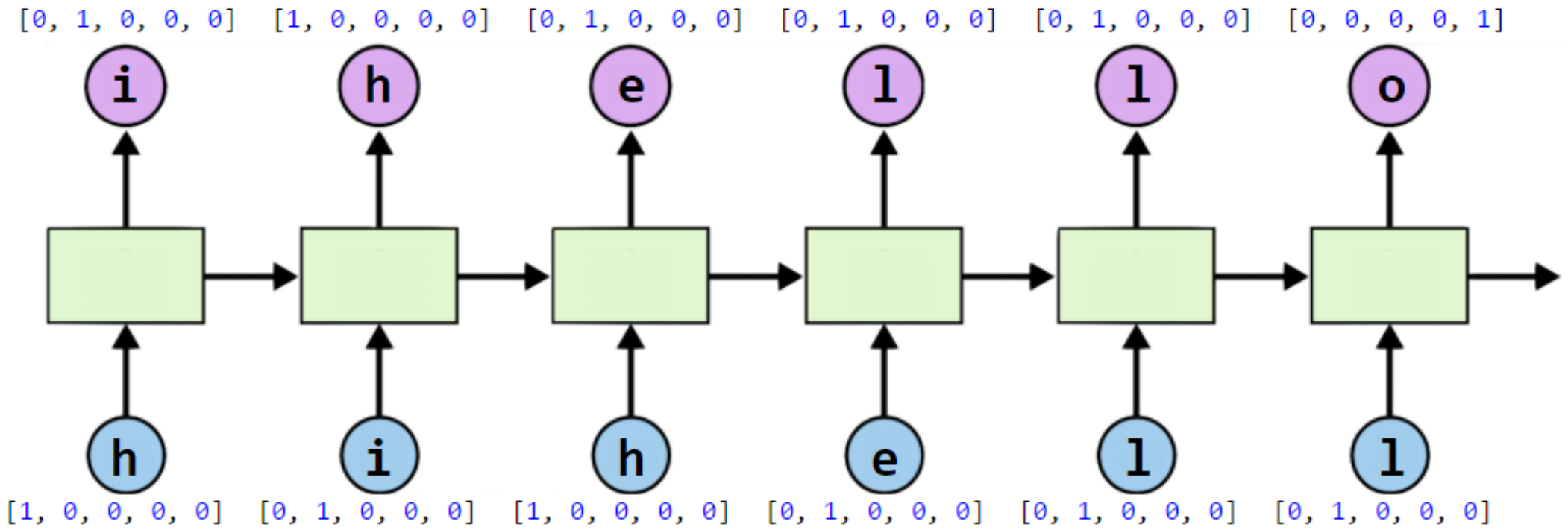


Teaching RNN ‘hihello’

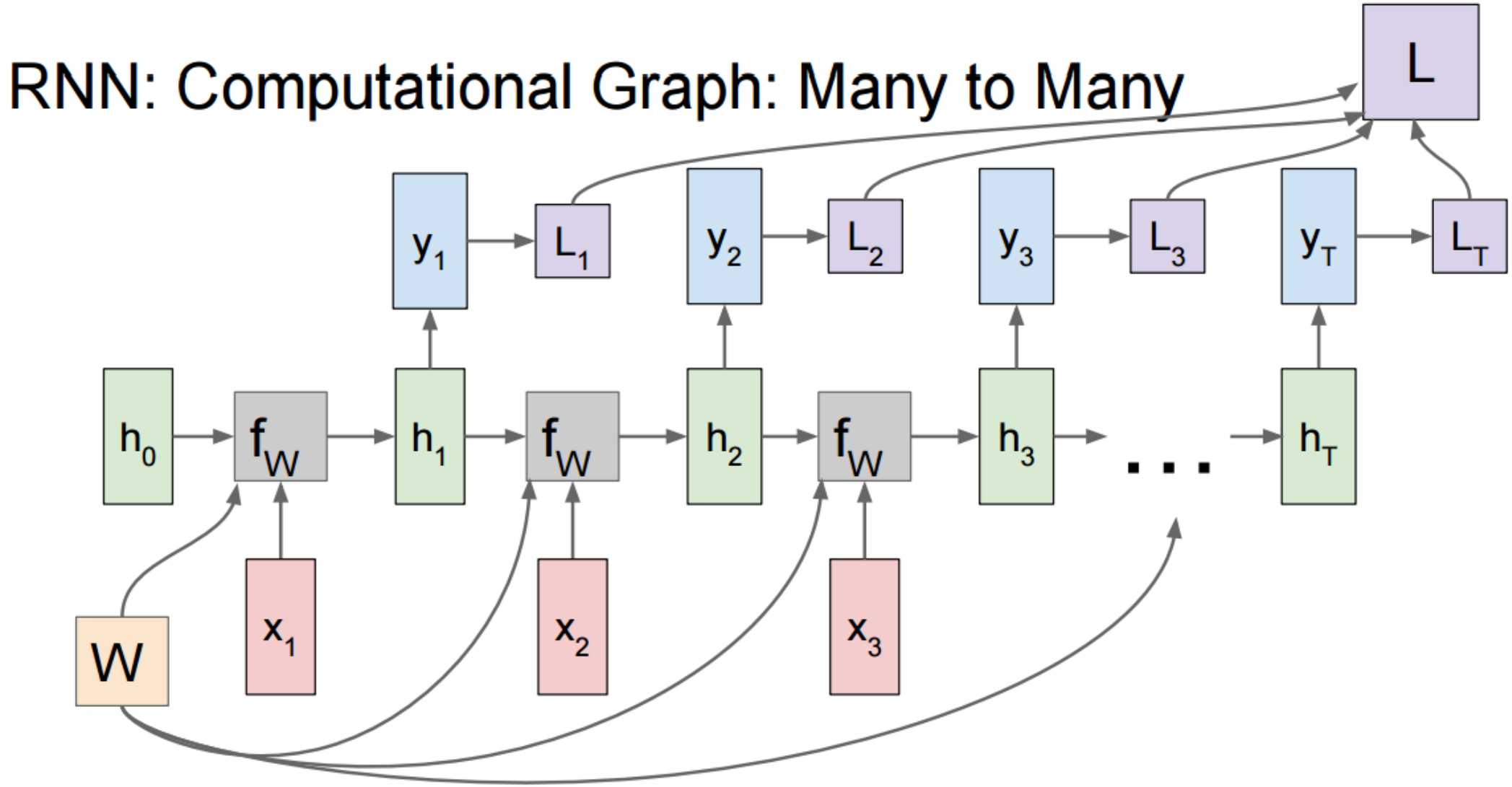
```
text: ‘hihello’
unique chars (vocabulary, voc): h, i, e, l, o
voc index: h:0, i:1, e:2, l:3, o:4
```

```
[1, 0, 0, 0, 0],   # h 0
[0, 1, 0, 0, 0],   # i 1
[0, 0, 1, 0, 0],   # e 2
[0, 0, 0, 1, 0],   # l 3
[0, 0, 0, 0, 1],   # o 4
```

Teaching RNN ‘hihello’



Teaching RNN ‘hihello’



Teaching RNN ‘hihello’

```
idx2char = ['h', 'i', 'e', 'l', 'o']
# Teach hello: hihell -> ihello
x_data = [[0, 1, 0, 2, 3, 3]] # hihell
x_one_hot = [[[1, 0, 0, 0, 0], # h 0
              [0, 1, 0, 0, 0], # i 1
              [1, 0, 0, 0, 0], # h 0
              [0, 0, 1, 0, 0], # e 2
              [0, 0, 0, 1, 0], # l 3
              [0, 0, 0, 1, 0]]] # l 3

y_data = [[1, 0, 2, 3, 3, 4]] # ihello

num_classes = 5
input_dim = 5 # one-hot size
hidden_size = 5 # output from the RNN. 5 to directly predict one-hot
batch_size = 1 # one sentence
sequence_length = 6 # |ihello| == 6
learning_rate = 0.1
```

Teaching RNN ‘hihello’

```
# Step 2: create placeholder
X = tf.placeholder(tf.float32, [None, sequence_length, input_dim])# X one-hot
Y = tf.placeholder(tf.int32, [None, sequence_length]) # Y label

# Step 3: build a model to teach 'hihello'
cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_size)
hidden, _ = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)

# output layer
hidden = tf.reshape(hidden, [-1, hidden_size])
W = tf.get_variable("W", [hidden_size, num_classes])
b = tf.get_variable("b", [num_classes])
outputs = tf.matmul(hidden, W) + b

# reshape out for sequence_loss
outputs = tf.reshape(outputs, [batch_size, sequence_length, num_classes])
```

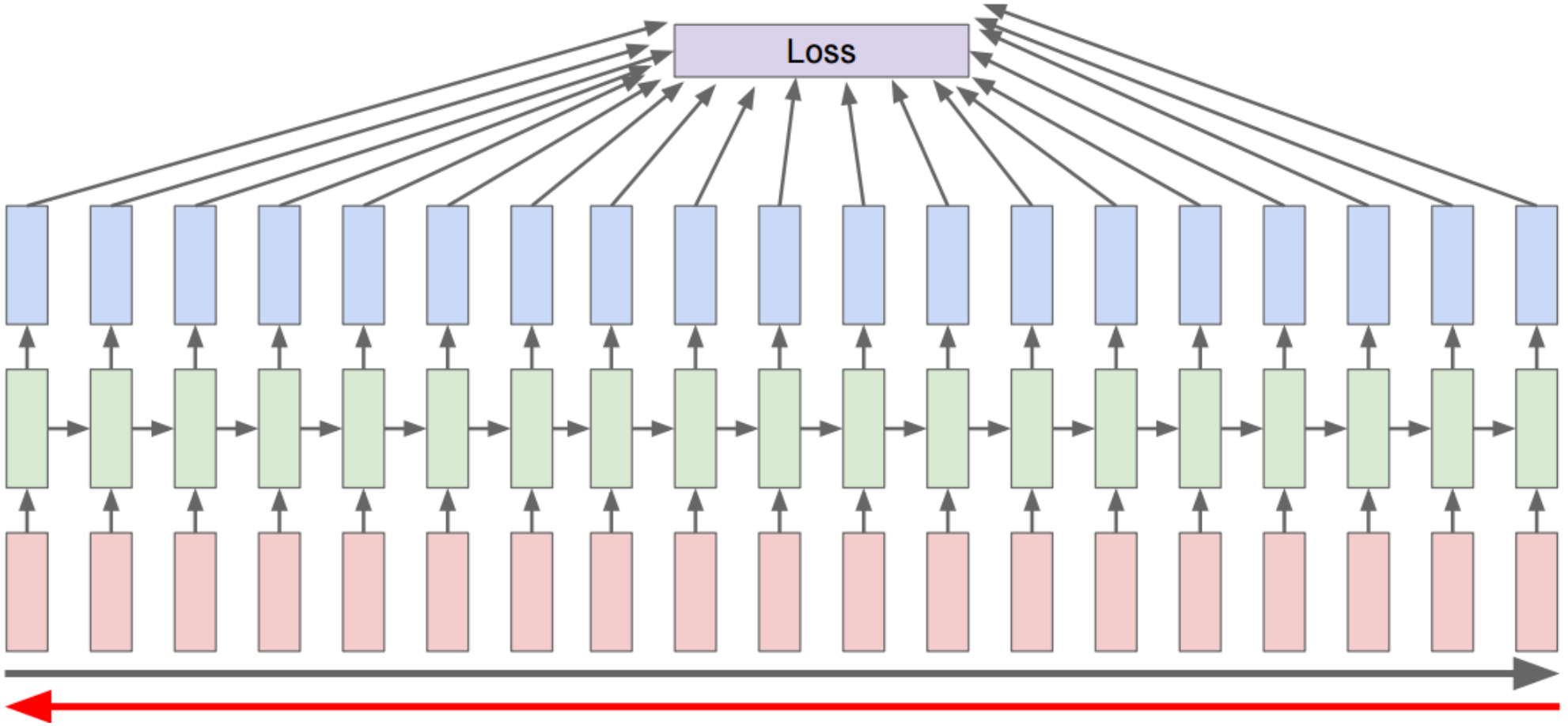
Teaching RNN ‘hihello’

```
# Step 4: define a loss
weights = tf.ones([batch_size, sequence_length])
sequence_loss = tf.contrib.seq2seq.sequence_loss(logits=outputs, targets=Y, weights=weights)
loss = tf.reduce_mean(sequence_loss)

# Step 5: use Adam optimizer to minimize the loss
train = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)

prediction = tf.argmax(outputs, axis=2)

# Step 6: train
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(50):
        l, _ = sess.run([loss, train], feed_dict={X: x_one_hot, Y: y_data})
        result = sess.run(prediction, feed_dict={X: x_one_hot})
        print(i, "loss:", l, "prediction: ", result, "true Y: ", y_data)

# print char using dic
result_str = [idx2char[c] for c in np.squeeze(result)]
print("\tPrediction str: ", ''.join(result_str))
```

Results

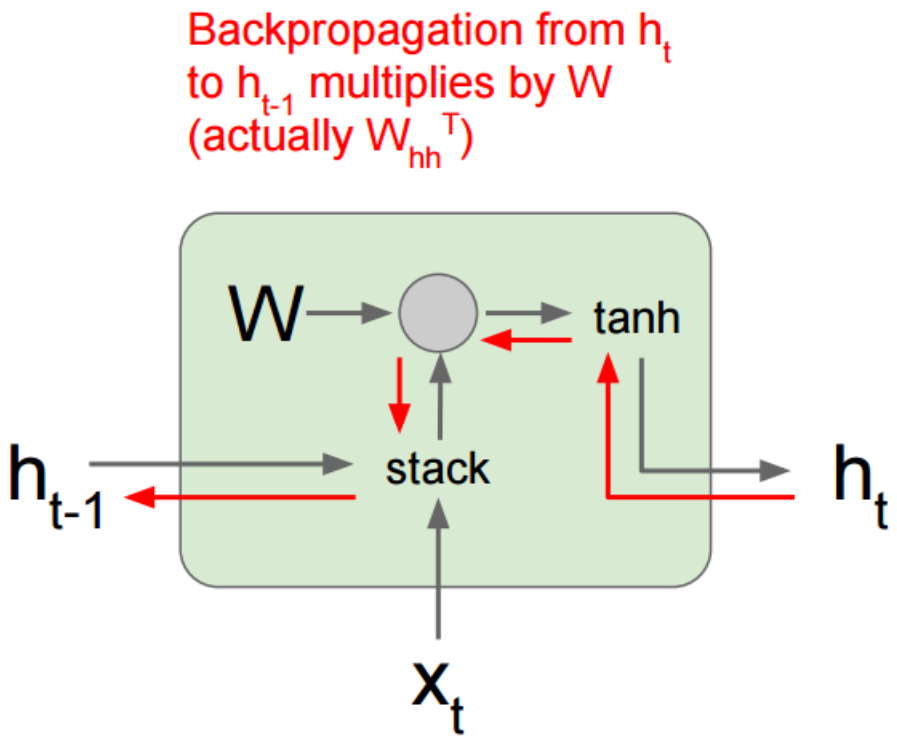
```
0 loss: 1.8184 prediction:  [[2 2 2 2 2 2]] true Y:  [[1, 0, 2, 3, 3, 4]]
        Prediction str:  eeeeee
1 loss: 1.46236 prediction:  [[1 3 2 3 3 4]] true Y:  [[1, 0, 2, 3, 3, 4]]
        Prediction str:  ilello
2 loss: 1.21289 prediction:  [[1 3 2 3 3 4]] true Y:  [[1, 0, 2, 3, 3, 4]]
        Prediction str:  ilello
3 loss: 1.01427 prediction:  [[1 3 1 3 3 4]] true Y:  [[1, 0, 2, 3, 3, 4]]
        Prediction str:  ilillo

...
...

48 loss: 0.0027253 prediction:  [[1 0 2 3 3 4]] true Y:  [[1, 0, 2, 3, 3, 4]]
         Prediction str:  ihello
49 loss: 0.00263653 prediction:  [[1 0 2 3 3 4]] true Y:  [[1, 0, 2, 3, 3, 4]]
         Prediction str:  ihello
```

Limitation of Vanilla RNNs



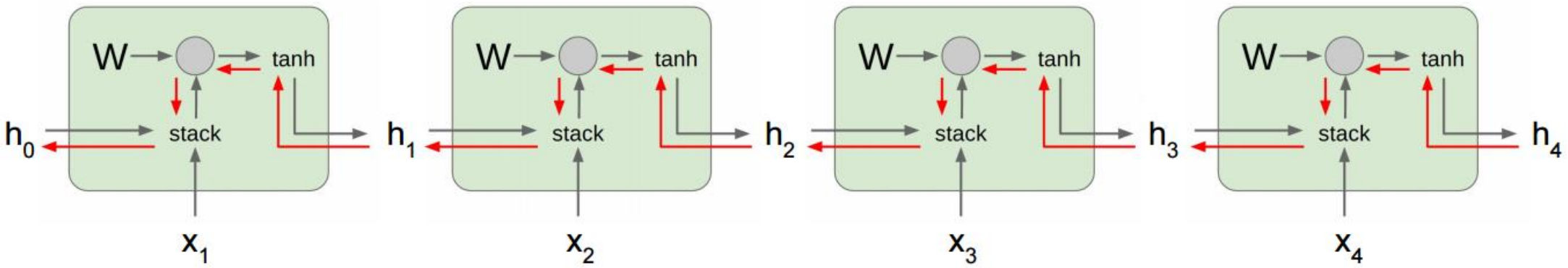
Backpropagation through time (BPTT)

Limitation of Vanilla RNNs



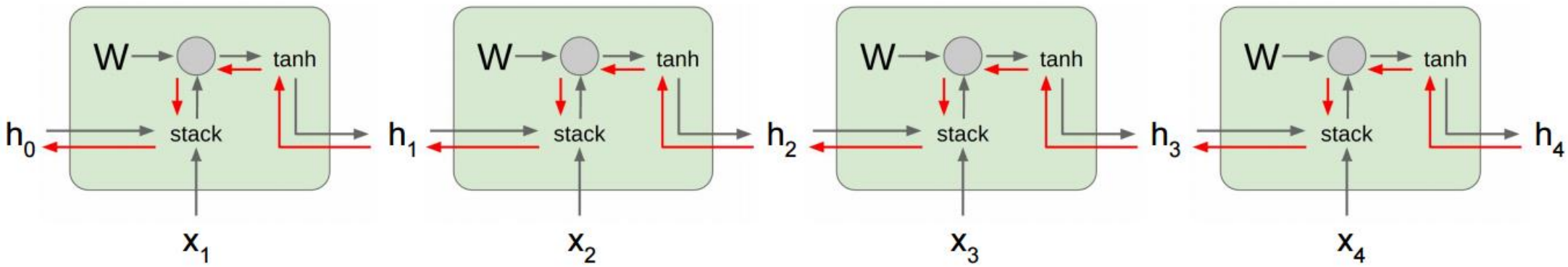
$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)
\end{aligned}
$$

Limitation of Vanilla RNNs



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Limitation of Vanilla RNNs

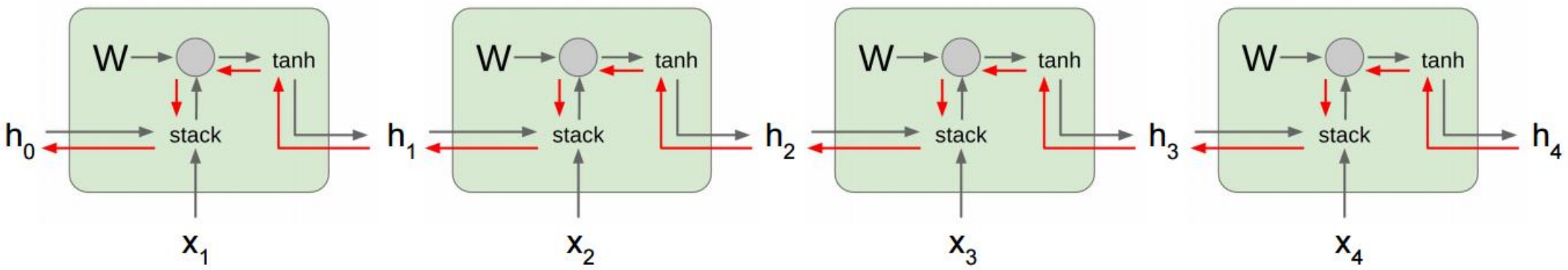


Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

Largest singular value < 1:
Vanishing gradients

Limitation of Vanilla RNNs



Computing gradient of h_0 involves many factors of W (and repeated tanh)

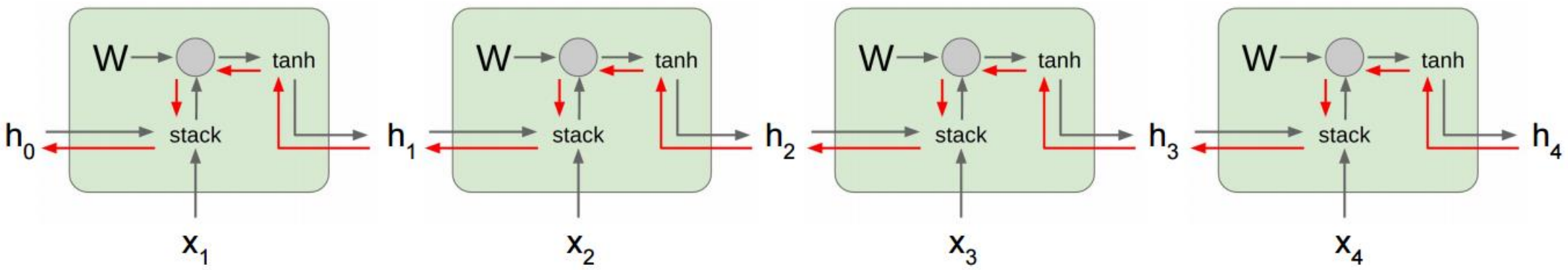
Largest singular value > 1:
Exploding gradients

→ **Gradient clipping**: Scale gradient if its norm is too big

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```

Largest singular value < 1:
Vanishing gradients

Limitation of Vanilla RNNs



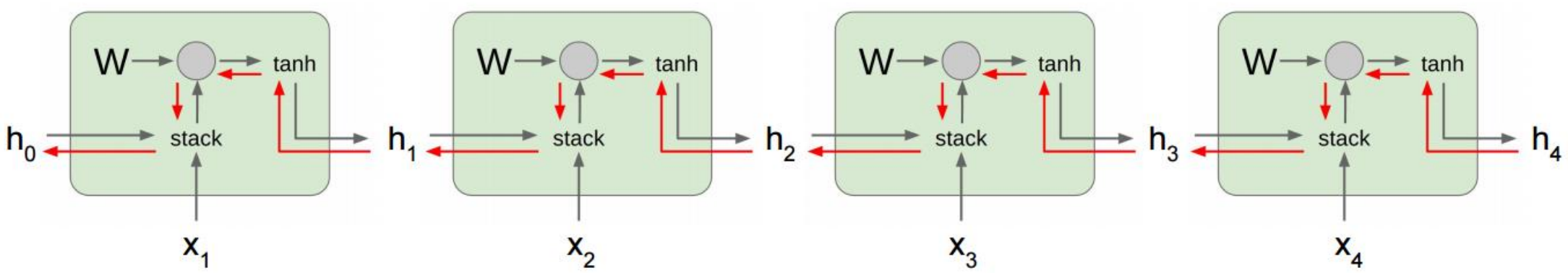
Computing gradient of h_0 involves many factor
(and r

Largest singular value > 1:
Exploding gradients

→ **Gradient clipping**: Scale gradient if its norm is too big

```
gradients = tf.gradients(loss, tf.trainable_variables())
clipped_gradients, _ = tf.clip_by_global_norm(gradients, max_grad_norm)
optimizer = tf.train.AdamOptimizer(learning_rate)
train_op = optimizer.apply_gradients(zip(gradients, trainables))
```

Limitation of Vanilla RNNs

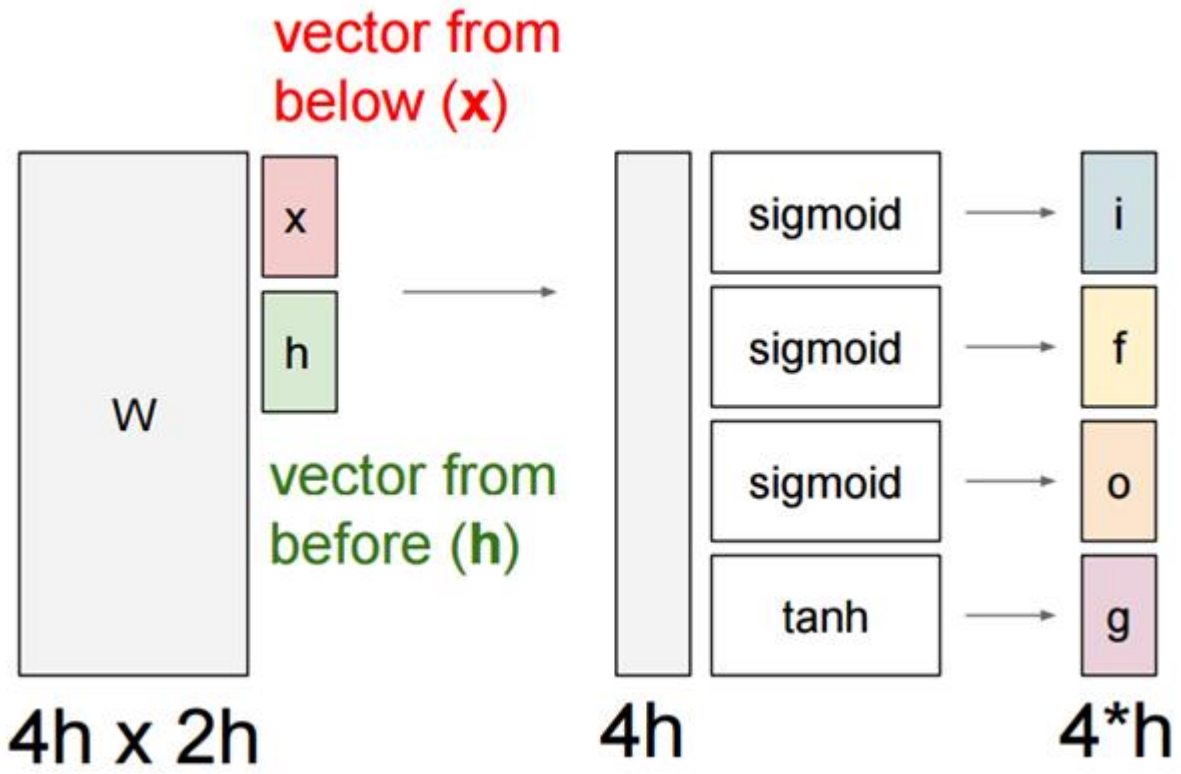


Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

Largest singular value < 1:
Vanishing gradients → Change RNN architecture

Long Short Term Memory (LSTM)



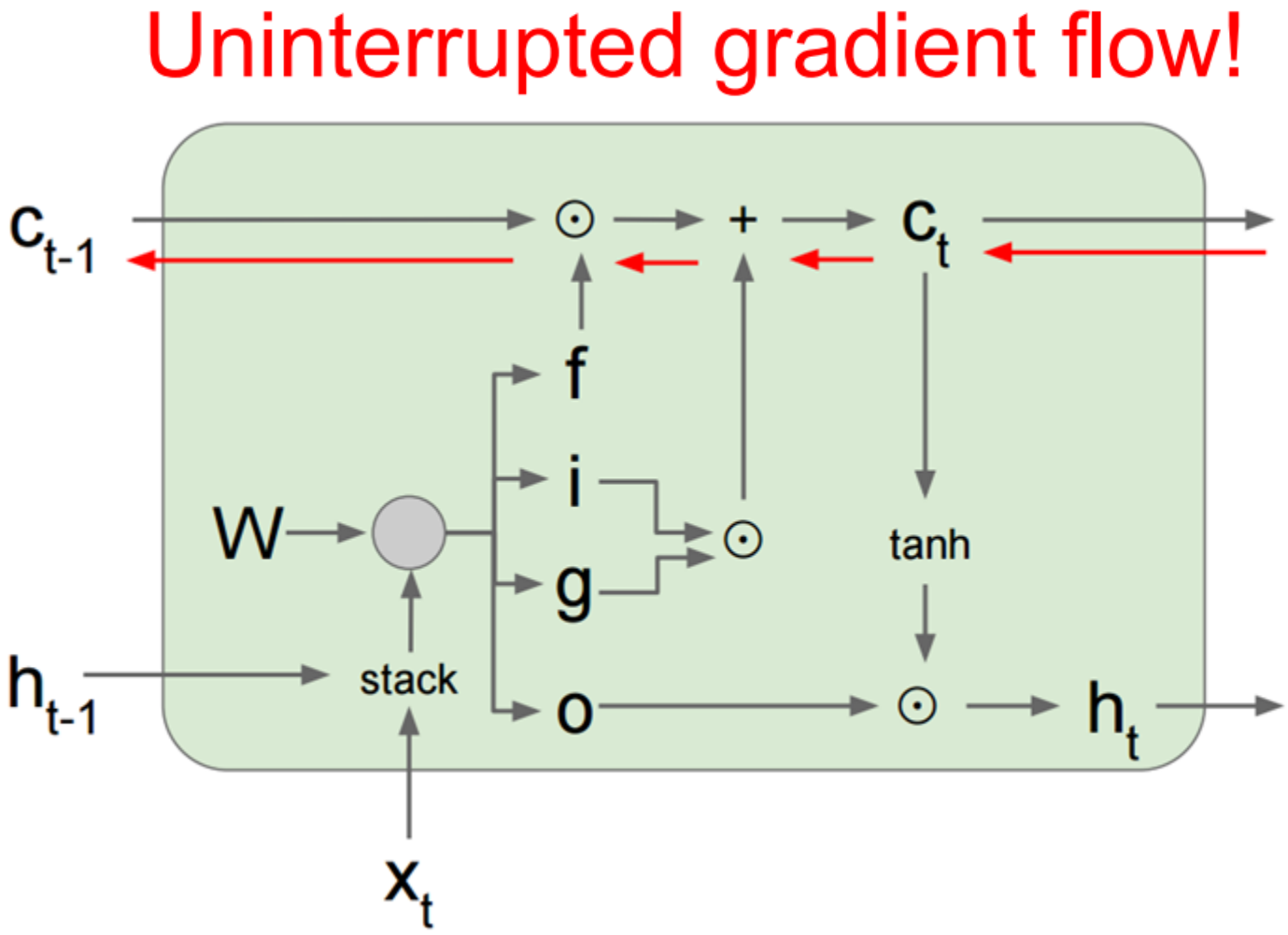
f: Forget gate, Whether to erase cell
i: Input gate, whether to write to cell
g: Gate gate (?), How much to write to cell
o: Output gate, How much to reveal cell

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Gated Recurrent Unit (GRU)

LSTMs work well, but unnecessarily complicated

Gated Recurrent Unit

[Cho et al., EMNLP2014; Chung, Gulcehre, Cho, Bengio, DLUFL2014]

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

$$\tilde{h} = \tanh(W\,[x_t] + U(r_t \odot h_{t-1}) + b)$$

$$u_t = \sigma(W_u\,[x_t] + U_u h_{t-1} + b_u)$$

$$r_t = \sigma(W_r\,[x_t] + U_r h_{t-1} + b_r)$$

Long Short-Term Memory

[Hochreiter & Schmidhuber, NC1999; Gers, Thesis2001]

$$h_t = o_t \odot \tanh(c_t)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

$$\tilde{c}_t = \tanh(W_c\,[x_t] + U_c h_{t-1} + b_c)$$

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

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

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

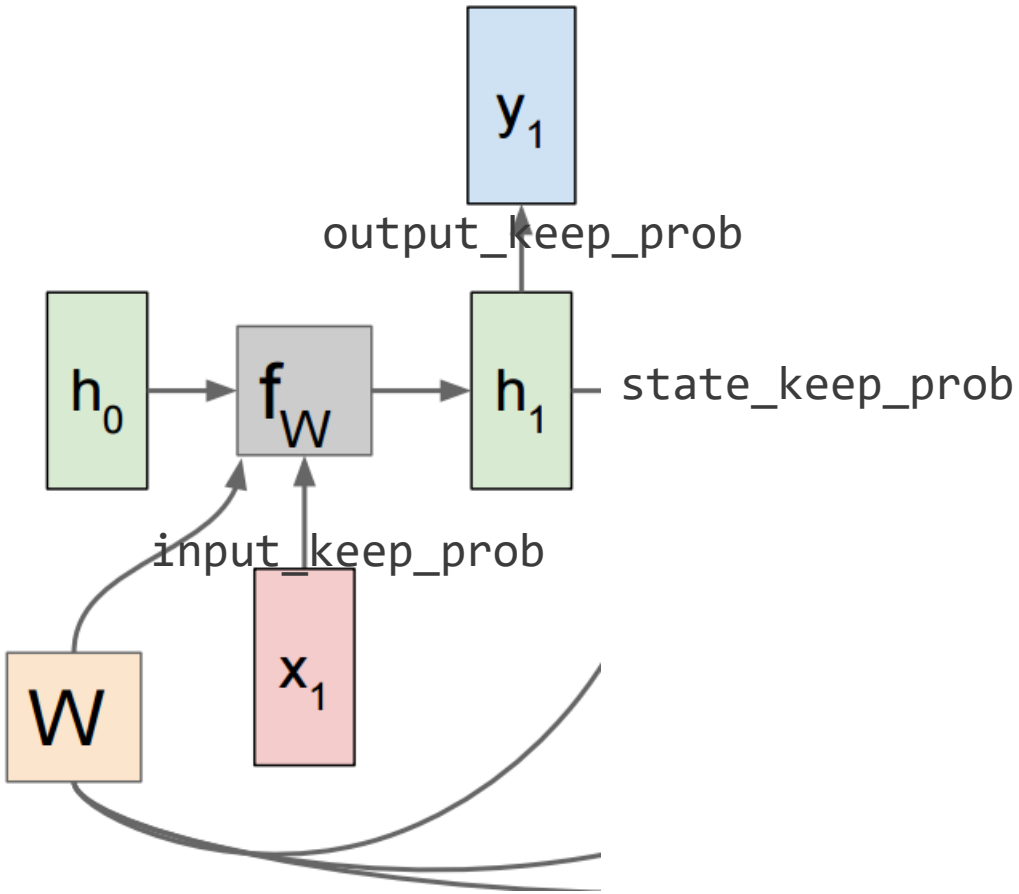
Computationally less expensive
Performance on par with LSTMs

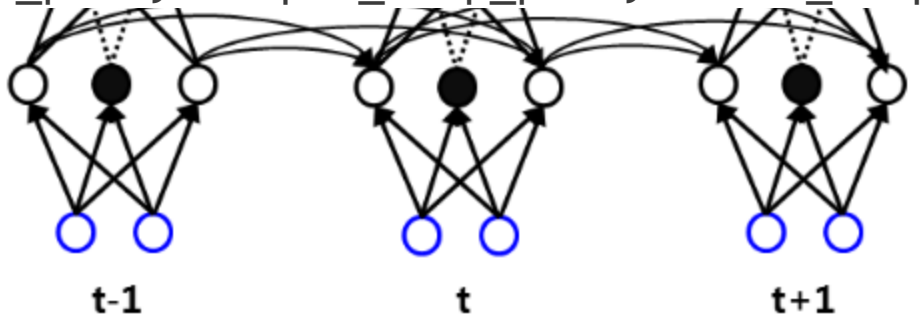
Advanced RNNs in TensorFlow

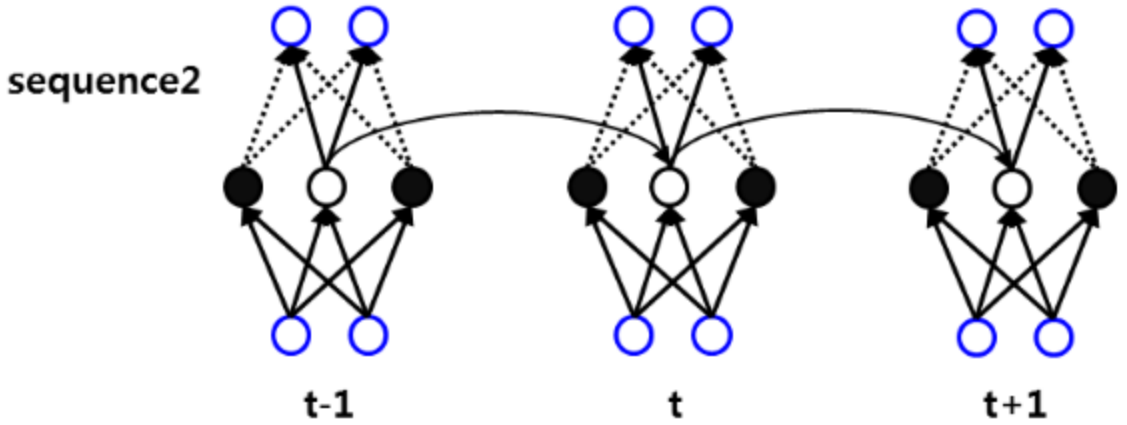
```
cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_size)
cell = tf.contrib.rnn.BasicLSTMCell(num_units=hidden_size)
cell = tf.contrib.rnn.GRUCell(num_units=hidden_size)

outputs, state = tf.nn.dynamic_rnn(cell, data, dtype=tf.float32)
```

Regularization

```
cell = tf.nn.rnn_cell.BasicLSTMCell(hidden_size)
cell = tf.nn.rnn_cell.DropoutWrapper(cell, input_keep_prob, output_keep_prob, state_keep_prob)
```

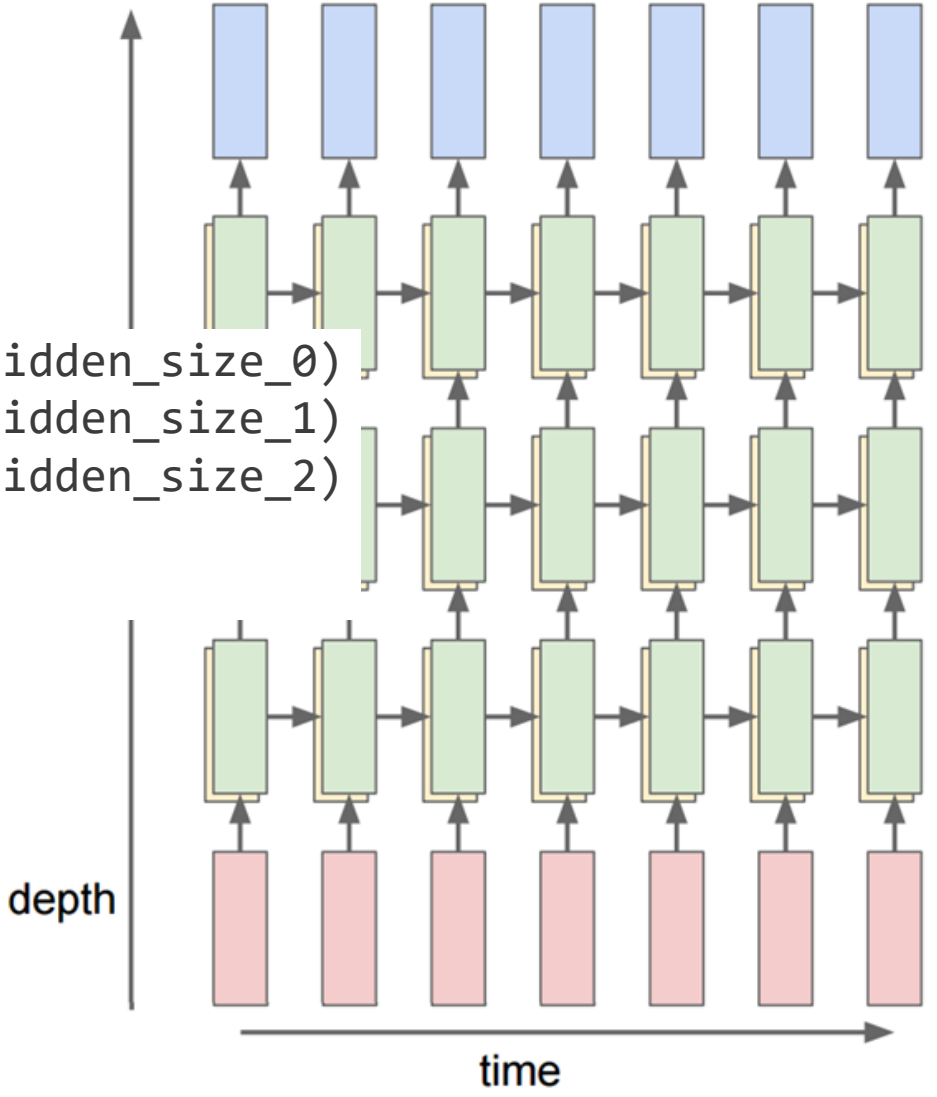






Stack Multiple RNN Cells

```
cell_0 = tf.contrib.rnn.BasicLSTMCell(num_units=hidden_size_0)
cell_1 = tf.contrib.rnn.BasicLSTMCell(num_units=hidden_size_1)
cell_2 = tf.contrib.rnn.BasicLSTMCell(num_units=hidden_size_2)

cell = rnn.MultiRNNCell([cell_0, cell_1, cell_2])
```



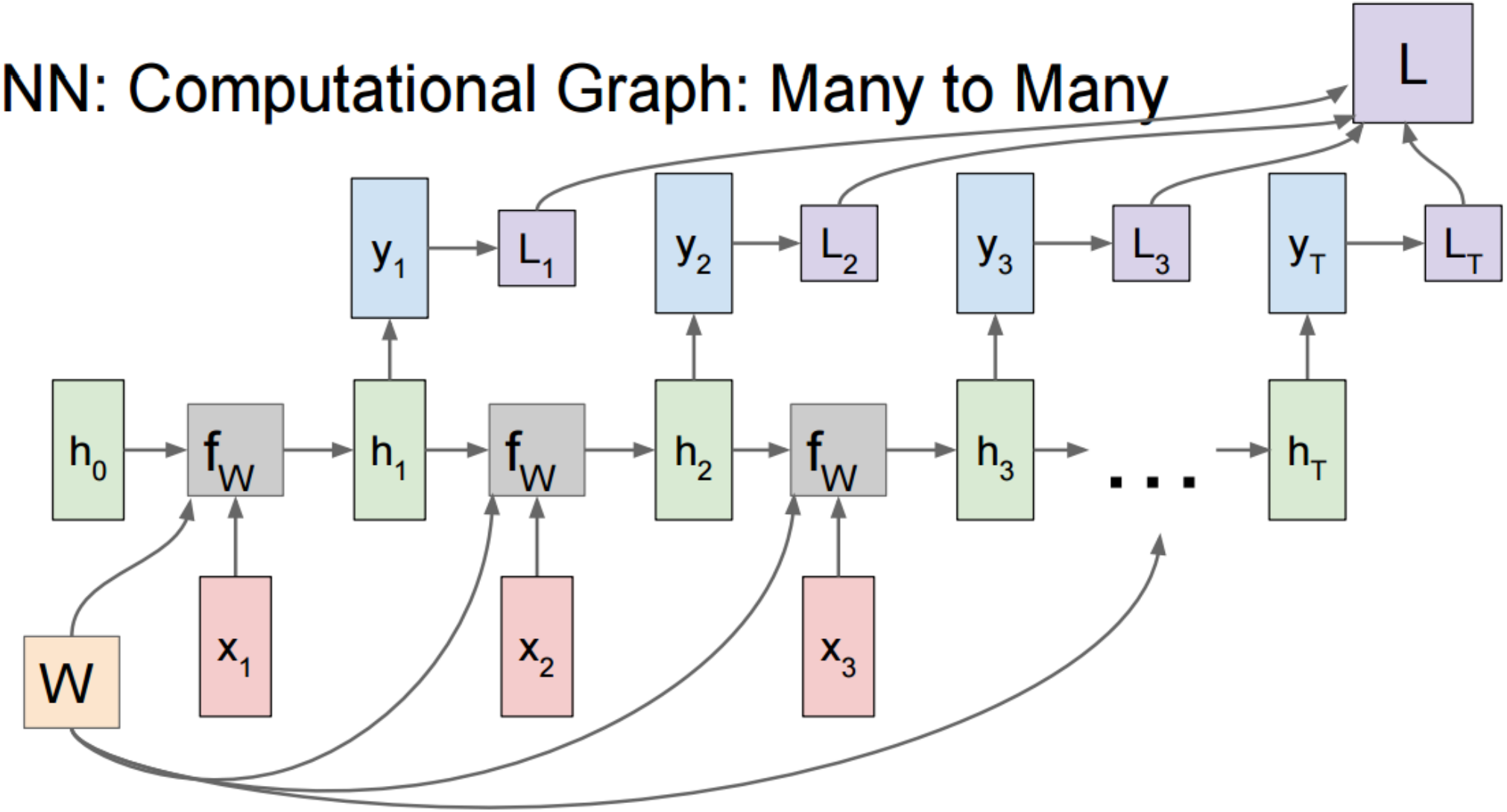
Character-Level Language Model (char-rnn)

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

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

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

```
VIOLA:
Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.
```

By Shakespheare

Character-Level Language Model (char-rnn)



Building a Model

```
# Step 2-1: create placeholder
self.input_data = tf.placeholder(tf.int32, [args.batch_size, args.seq_length])
self.targets = tf.placeholder(tf.int32, [args.batch_size, args.seq_length])
```

Building a Model

```
# Step 2-1: create placeholder
self.input_data = tf.placeholder(tf.int32, [args.batch_size, args.seq_length])
self.targets = tf.placeholder(tf.int32, [args.batch_size, args.seq_length])

# Step 2-2: define multi-layer RNN
cells = []
for _ in range(args.num_layers):
    cell = tf.contrib.rnn.BasicLSTMCell(args.rnn_size)
    if training and (args.output_keep_prob < 1.0 or args.input_keep_prob < 1.0):
        cell = tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=args.input_keep_prob,
                                                   output_keep_prob=args.output_keep_prob)
    cells.append(cell)
```

Building a Model

```
# Step 2-1: create placeholder
self.input_data = tf.placeholder(tf.int32, [args.batch_size, args.seq_length])
self.targets = tf.placeholder(tf.int32, [args.batch_size, args.seq_length])

# Step 2-2: define multi-layer RNN
cells = []
for _ in range(args.num_layers):
    cell = tf.contrib.rnn.BasicLSTMCell(args.rnn_size)
    if training and (args.output_keep_prob < 1.0 or args.input_keep_prob < 1.0):
        cell = tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=args.input_keep_prob,
                                                   output_keep_prob=args.output_keep_prob)
    cells.append(cell)

self.cell = cell = tf.contrib.rnn.MultiRNNCell(cells)
self.initial_state = cell.zero_state(args.batch_size, tf.float32)

embedding = tf.get_variable("embedding", [args.vocab_size, args.rnn_size])
inputs = tf.nn.embedding_lookup(embedding, self.input_data)

outputs, last_state = tf.nn.dynamic_rnn(cell, inputs, initial_state=self.initial_state)
outputs = tf.reshape(outputs, [-1, args.rnn_size])
```

Building a Model

```
# Step 2-3: compute outputs and loss
with tf.variable_scope('rnnlm'):
softmax_w = tf.get_variable("softmax_w", [args.rnn_size, args.vocab_size])
softmax_b = tf.get_variable("softmax_b", [args.vocab_size])

self.logits = tf.matmul(outputs, softmax_w) + softmax_b
self.probs = tf.nn.softmax(self.logits)
loss = tf.contrib.seq2seq.sequence_loss(
    tf.reshape(self.logits, [-1, args.seq_length, args.vocab_size]),
    self.targets,
    tf.ones([args.batch_size, args.seq_length]))
self.cost = tf.reduce_mean(loss)
self.final_state = last_state
```

Stateful

```
for e in range(args.num_epochs):
data_loader.reset_batch_pointer()
state = sess.run(model.initial_state)
    for b in range(data_loader.num_batches):
        start = time.time()
        x, y = data_loader.next_batch()
        feed = {model.input_data: x, model.targets: y}
        # copy the state of previous batch
        for i, (c, h) in enumerate(model.initial_state):
            feed[c] = state[i].c
            feed[h] = state[i].h
        train_loss, state, _ = sess.run([model.cost, model.final_state, train_op], feed)
```

Checkpoint

```
# Step 4: train
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.Saver(tf.global_variables())

    ...

    if (e * data_loader.num_batches + b) % args.save_every == 0\
        or (e == args.num_epochs-1 and b == data_loader.num_batches-1):
        # save for the last result
        checkpoint_path = os.path.join(args.save_dir, 'model.ckpt')
        saver.save(sess, checkpoint_path, global_step=e * data_loader.num_batches + b)
        print("model saved to {}".format(checkpoint_path))
```

Checkpoint

```
saver = tf.train.Saver(tf.global_variables())
ckpt = tf.train.get_checkpoint_state(args.save_dir)
if ckpt and ckpt.model_checkpoint_path:
    saver.restore(sess, ckpt.model_checkpoint_path)
    print(model.sample(sess, chars, vocab, args.n, args.prime, args.sample).encode('utf-8'))
```

Training

```
loading preprocessed files
0/8920 (epoch 0), train_loss = 4.163, time/batch = 0.303
1/8920 (epoch 0), train_loss = 4.131, time/batch = 0.045
2/8920 (epoch 0), train_loss = 4.047, time/batch = 0.041
3/8920 (epoch 0), train_loss = 3.862, time/batch = 0.042
4/8920 (epoch 0), train_loss = 3.562, time/batch = 0.046
5/8920 (epoch 0), train_loss = 3.529, time/batch = 0.036
6/8920 (epoch 0), train_loss = 3.469, time/batch = 0.044
7/8920 (epoch 0), train_loss = 3.404, time/batch = 0.039
...
...
8917/8920 (epoch 19), train_loss = 1.303, time/batch = 0.039
8918/8920 (epoch 19), train_loss = 1.333, time/batch = 0.047
8919/8920 (epoch 19), train_loss = 1.314, time/batch = 0.051
model saved to save\model.ckpt
```

Results

```
HENRVETANTI:
Mast no storions,--lrawled not ured, here, sir.

LADY CAPULET:
You tell the which I possibure goodmbalts':
And come the purpose, from his opproach him and
raste.

ESCALUS:
Come, wrench, it not one grace with respects,
and I lands,
When sheep the thrust the quasser thy converdon;
Burden mannighs, another descent.

PostiUDMesteret:
My lord! visit him the drums and
Oncine for Than him to despicious scurricy,
And she doubt endity fooling appeals of the
nears of the caits be some to a l
```