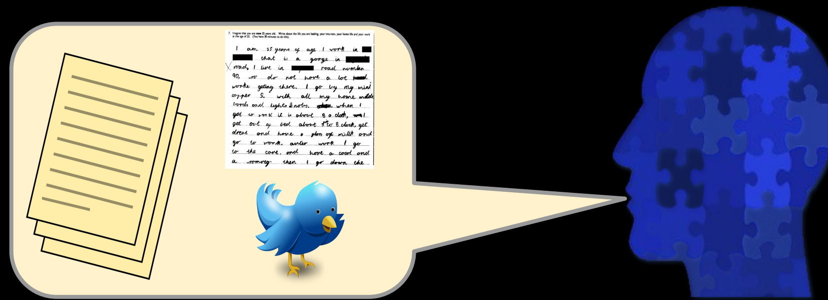
Recurrent Neural Networks for Language Modeling

CSE354 - Spring 2020
Natural Language Processing

Tasks



- Language Modeling:

  Generate next word, sentence

  ≈ capture hidden

  representation of sentences.

how?

- Recurrent Neural Network and Sequence Models

Language Modeling

Task: Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$

:probability of a next word given history

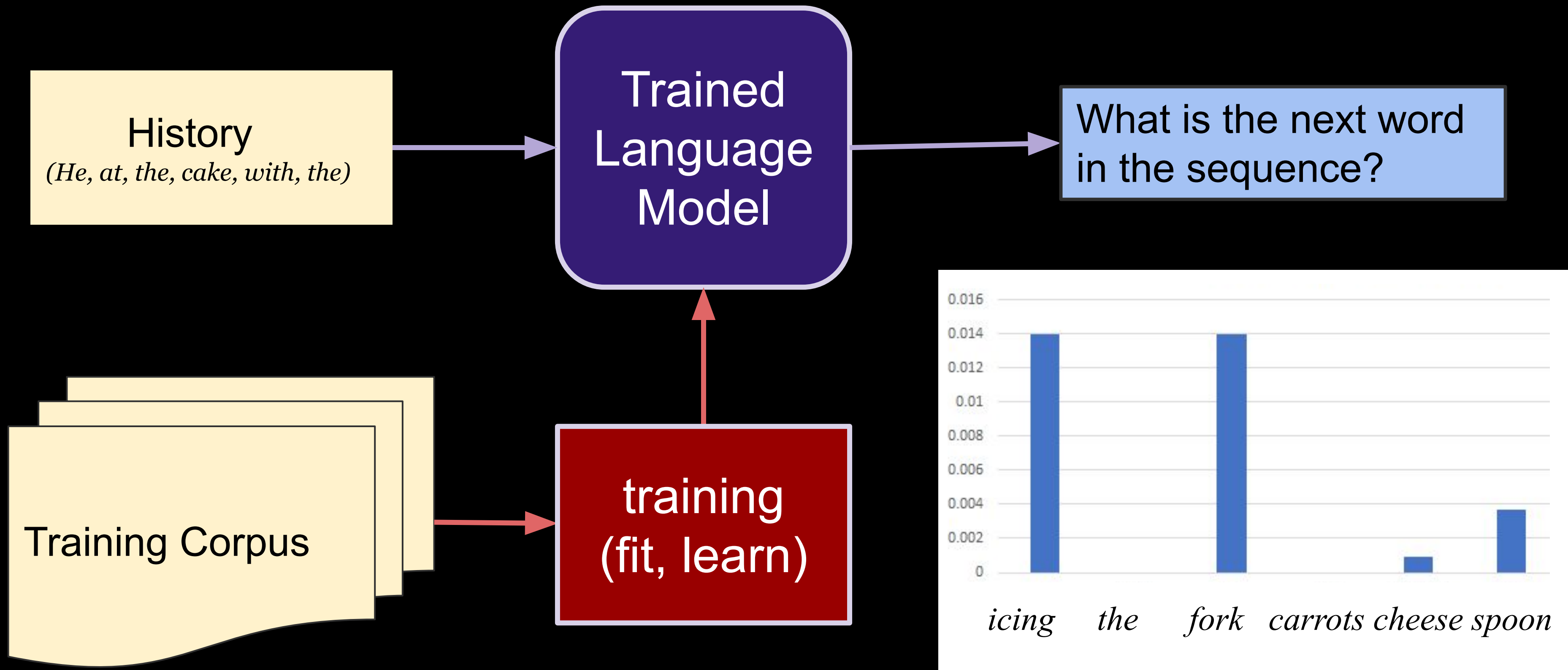
$P(fork \mid He\ ate\ the\ cake\ with\ the) = ?$

Language Modeling

Task: Estimate $P(w_n \mid w_1, w_2, \ldots, w_{n-1})$

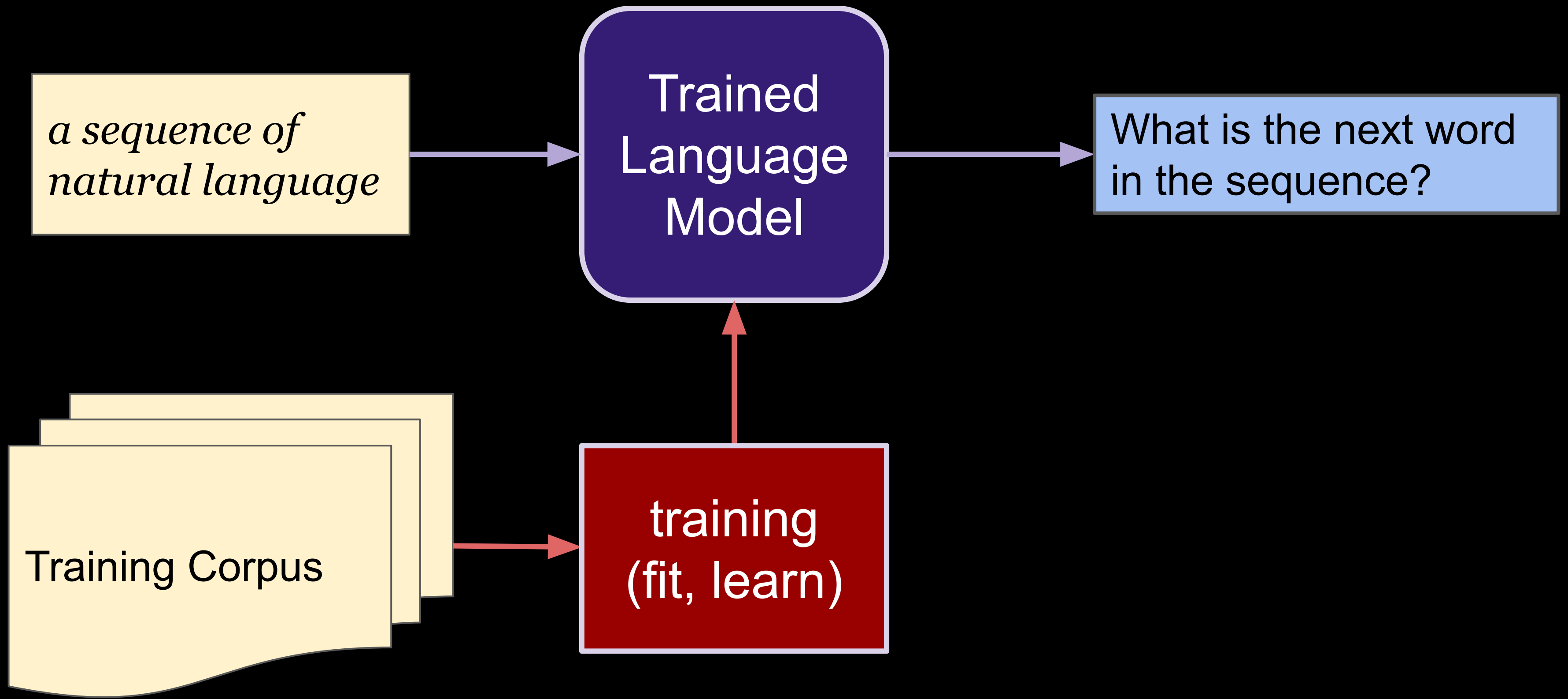
:probability of a next word given history

P(fork | He ate the cake with the) = ?



Language Modeling

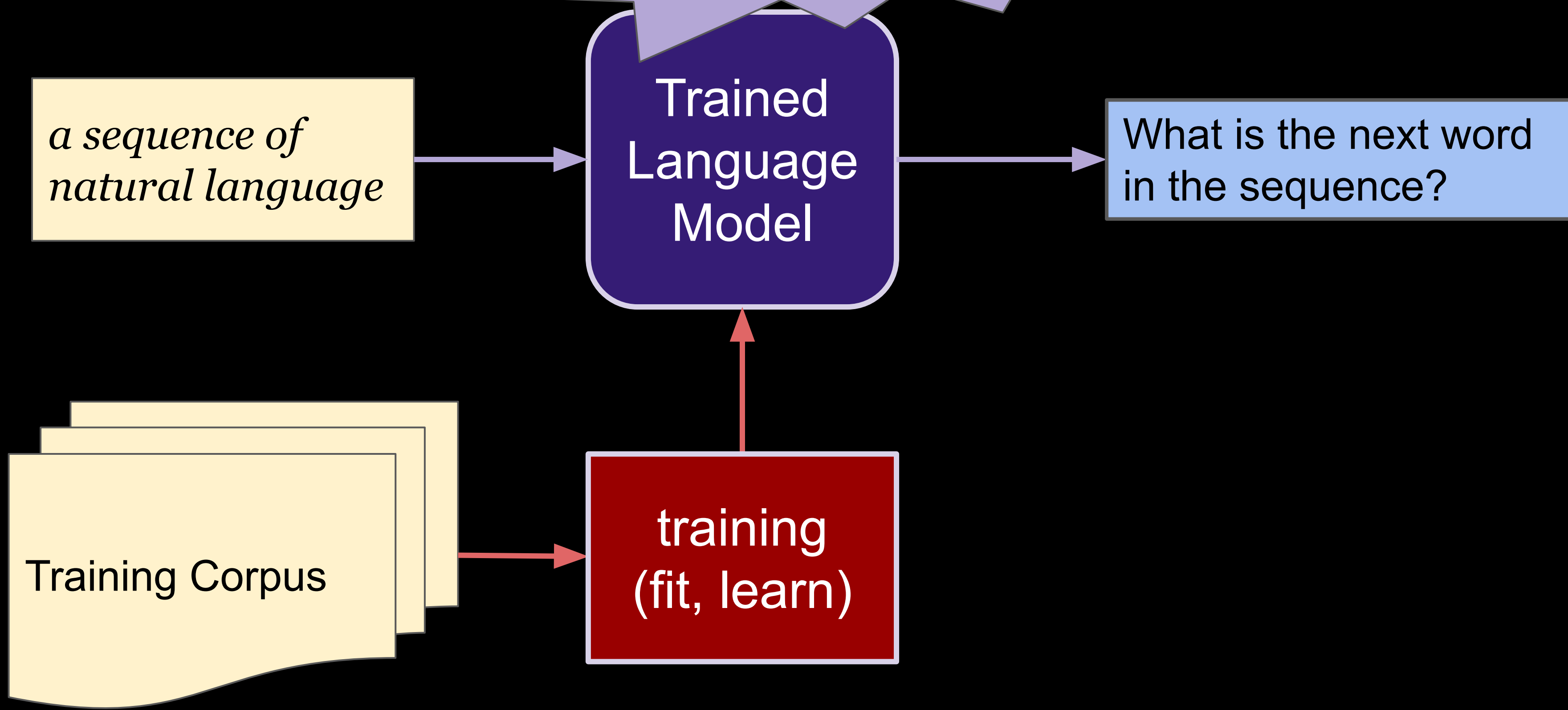
Building a model (or system / API) that can answer the following:



Language Modeling

To fully capture natural language, models get very complex!

Building a model (or s… …er the following:



Neural Networks: Graphs of Operations

Neural Networks: Graphs of Operations (excluding the optimization nodes)

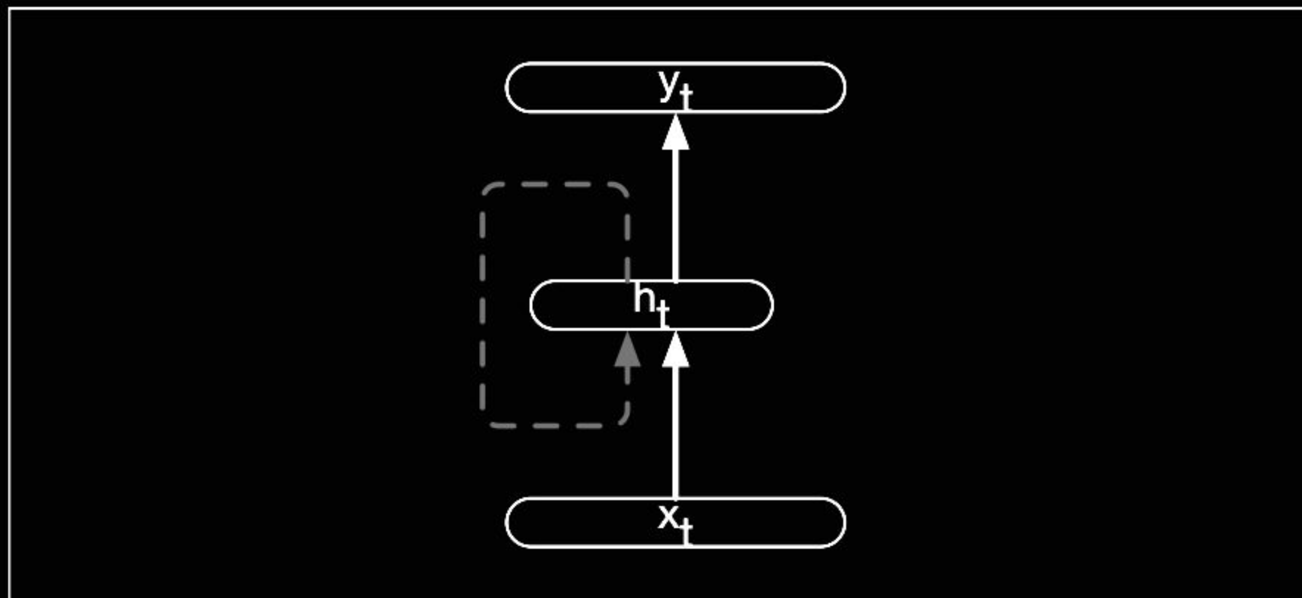


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Neural Networks: Graphs of Operations (excluding the optimization nodes)

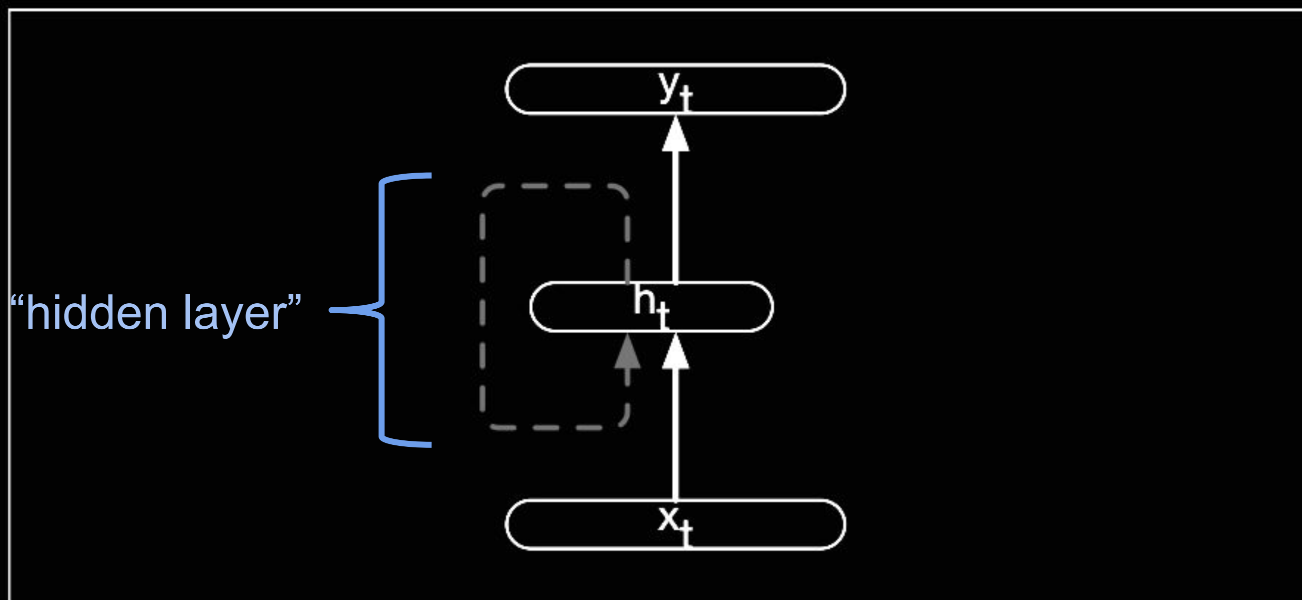
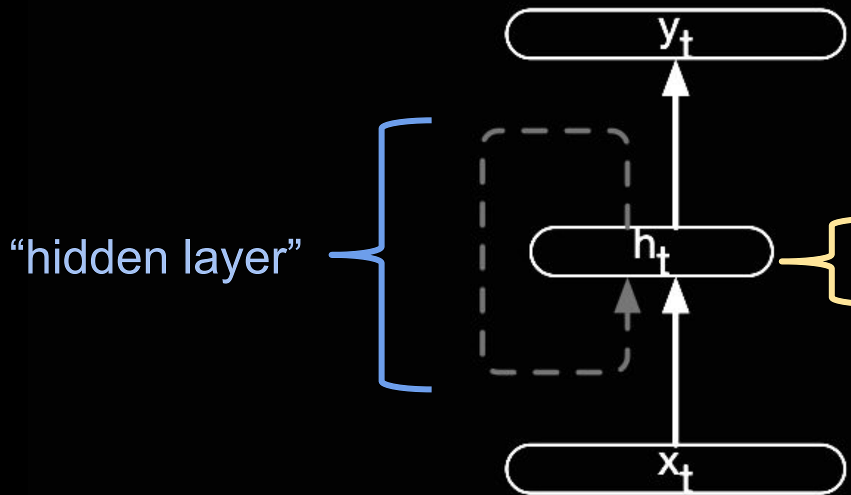


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Neural Networks: Graphs of Operations
(excluding the optimization nodes)

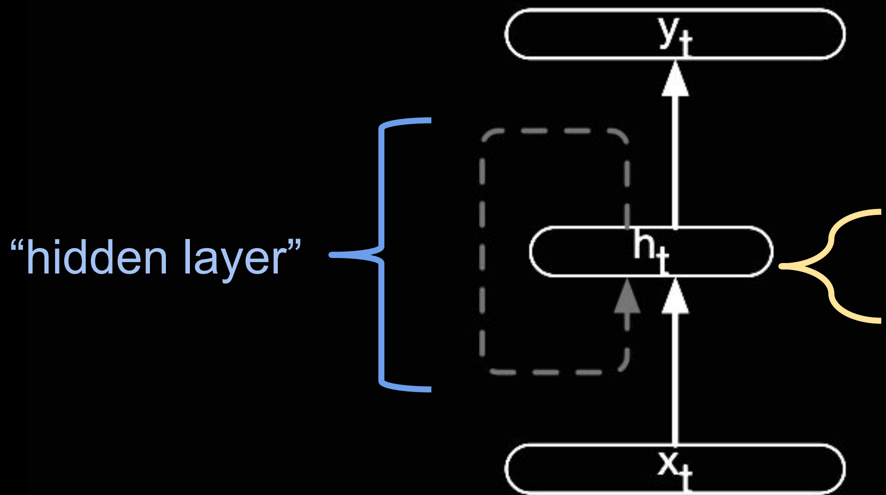


$$h_t = g(vecmul(h_{t-1}, U) + vecmul(x_t, V))$$

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Neural Networks: Graphs of Operations
(excluding the optimization nodes)



$$h_t = g(h_{t-1} U + x_t V)$$

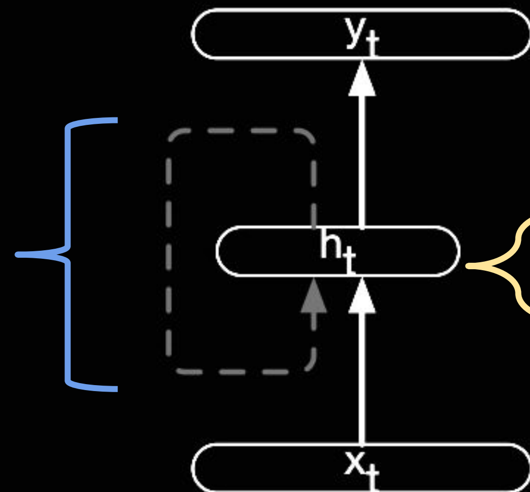
short hand for vector/ matrix multiply

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Neural Networks: Graphs of Operations
(excluding the optimization nodes)

"hidden layer"



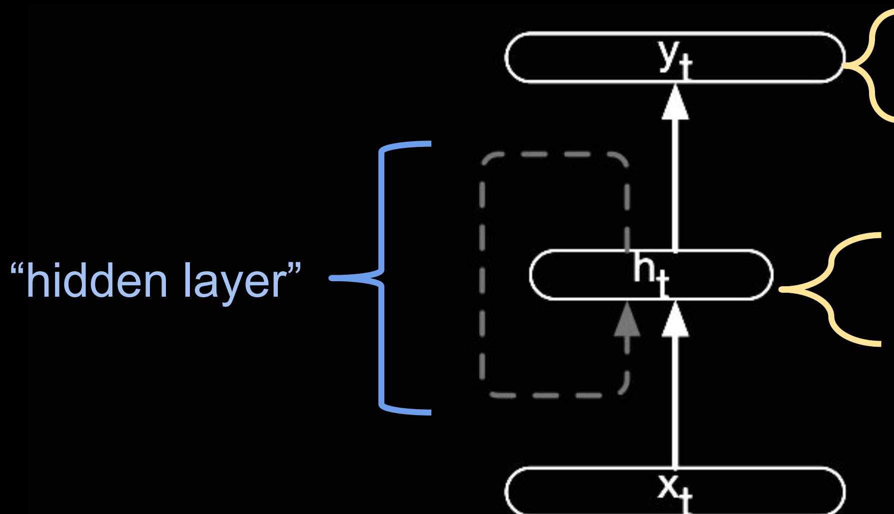
Activation Function

$$h_{(t)} = g(h_{(t-1)}U + x_{(t)}V)$$

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Neural Networks: Graphs of Operations
(excluding the optimization nodes)



$$y_{(t)} = f(h_{(t)}W)$$

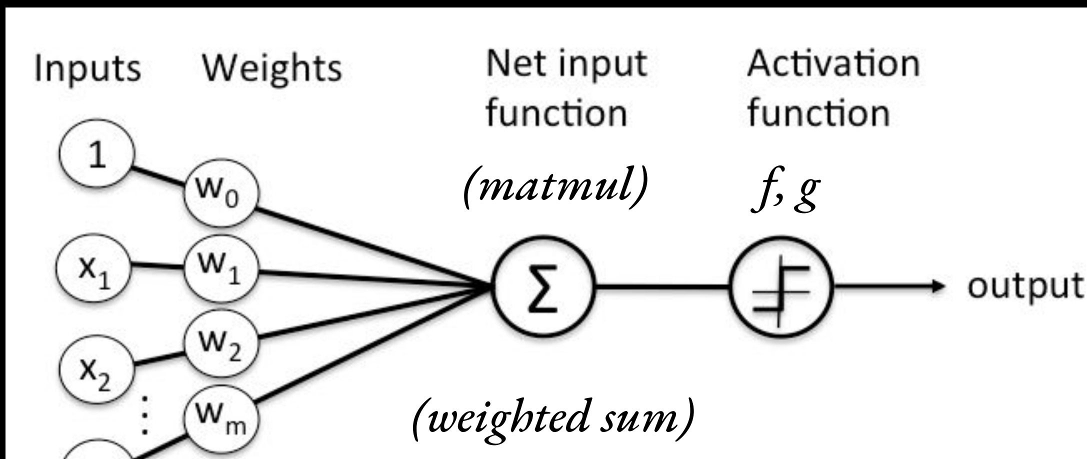
Activation Function

$$h_{(t)} = g(h_{(t-1)}U + x_{(t)}V)$$

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Neural Networks: Graphs of Operations
(excluding the optimization nodes)



(skymind, AI Wiki)

$$y_{(t)} = f(h_{(t)}W)$$

Activation Function

$$h_{(t)} = g(h_{(t-1)}U + x_{(t)}V)$$

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Common Activation Functions

$z = h_{(t)}W$

Logistic: $\sigma(z) = 1 / (1 + e^{-z})$

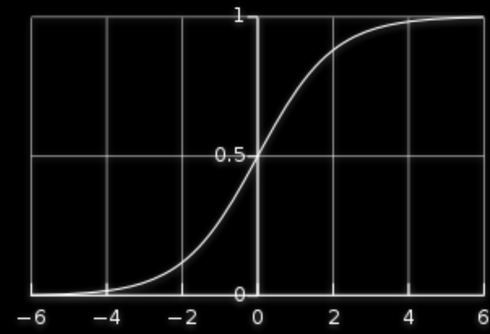
Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1)$

Rectified linear unit (ReLU): $ReLU(z) = \max(0, z)$

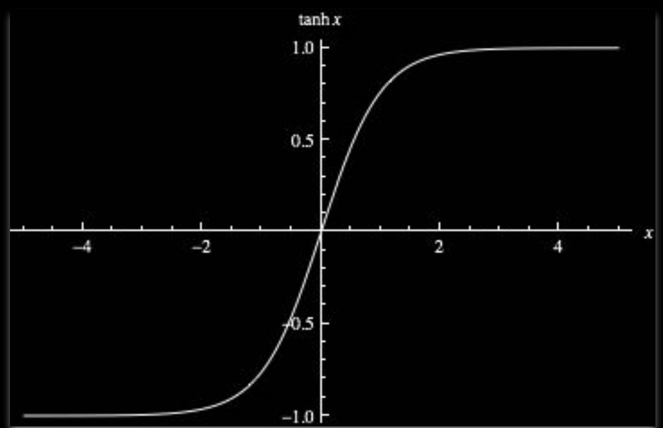
Common Activation Functions

$z = h_{(t)}W$

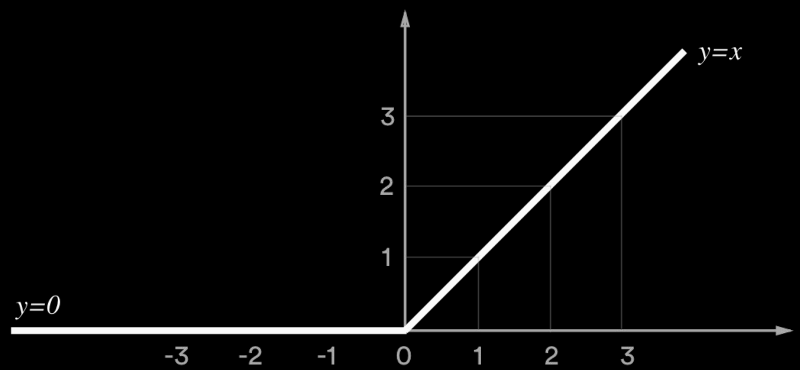
Logistic: $\sigma(z) = 1 / (1 + e^{-z})$



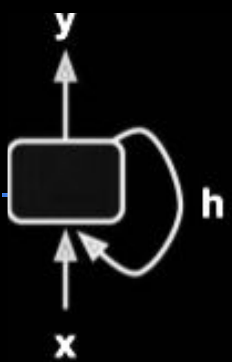
Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1)$



Rectified linear unit (ReLU): $ReLU(z) = \max(0, z)$



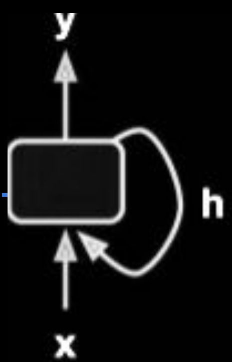
Example: Forward Pass



(Geron, 2017)

```
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = g(U h(i-1) + W x(i)) #update hidden state
    y(i) = f(V h(i)) #update output
```

Example: Forward Pass




```
#define forward pass graph:
h_(0) = 0
for i in range(1, len(x)):
    h_(i) = g(U h_(i-1) + W x_(i)) #update hidden state
    y_(i) = f(V h_(i)) #update output
```

Example: Forward Pass


```
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = tanh(matmul(U,h(i-1))+ matmul(W,x(i))) #update hidden state
    y(i) = softmax(matmul(V, h(i))) #update output
```

Language Modeling

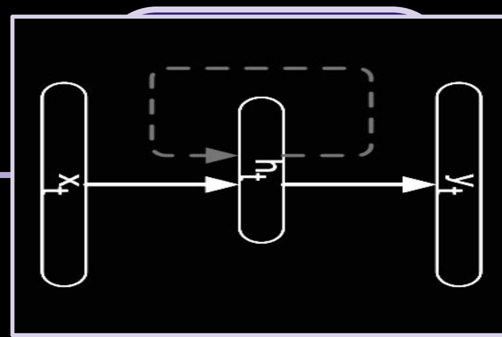
Task: Estimate $P(w_n \mid w_1, w_2, \ldots, w_{n-1})$

:probability of a next word given history

P(fork | He ate the cake with the) = ?

History

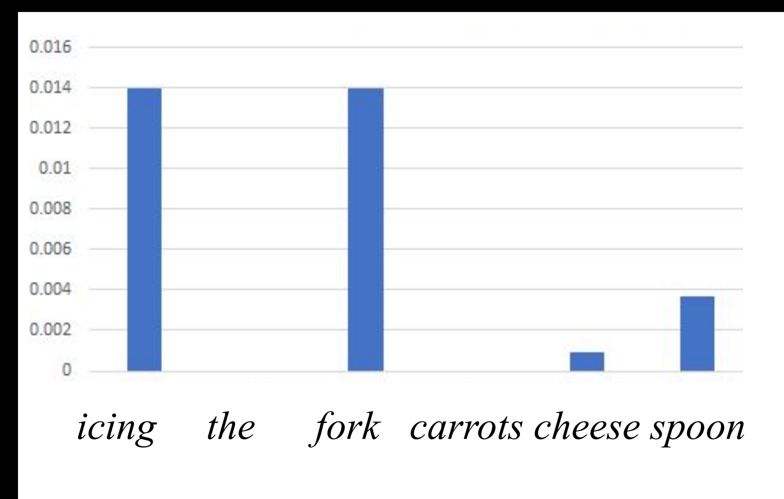
(He, at, the, cake, with, the)



What is the next word in the sequence?

Training Corpus

training (fit, learn)



Language Modeling

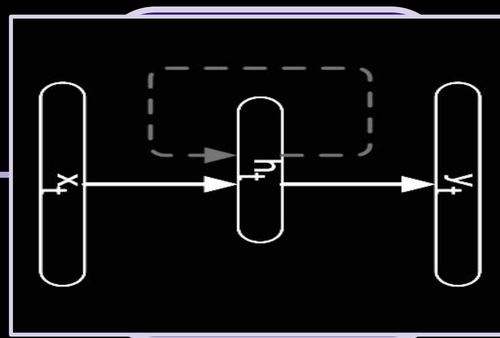
Task: Estimate $P(w_n \mid w_1, w_2, \ldots, w_{n-1})$

:probability of a next word given history

P(fork | He ate the cake with the) = ?

~~History~~Last word

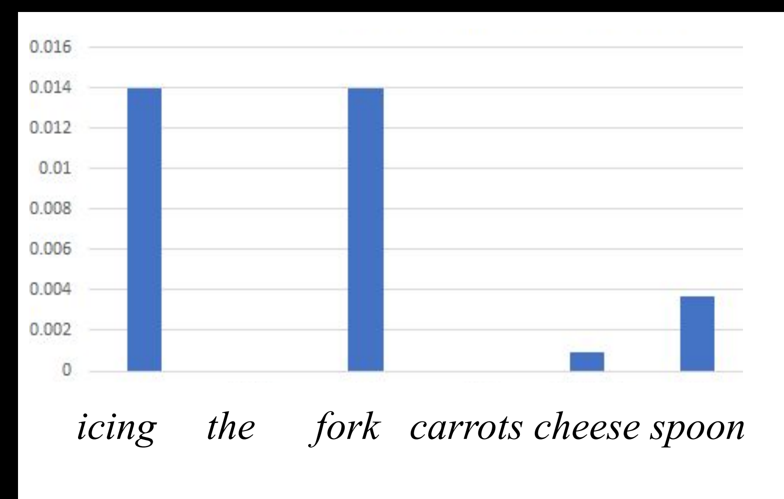
(~~He, at, the, cake, with,~~ the)



What is the next word in the sequence?

Training Corpus

training (fit, learn)



Language Modeling

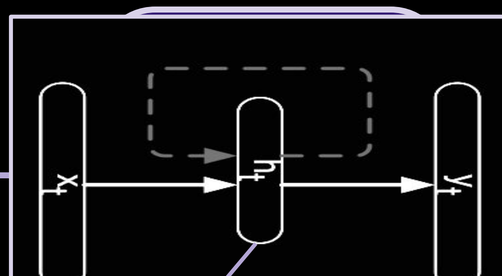
Task: Estimate $P(w_n \mid w_1, w_2, \ldots, w_{n-1})$

:probability of a next word given history

P(fork | He ate the cake with the) = ?

Last word

(the)



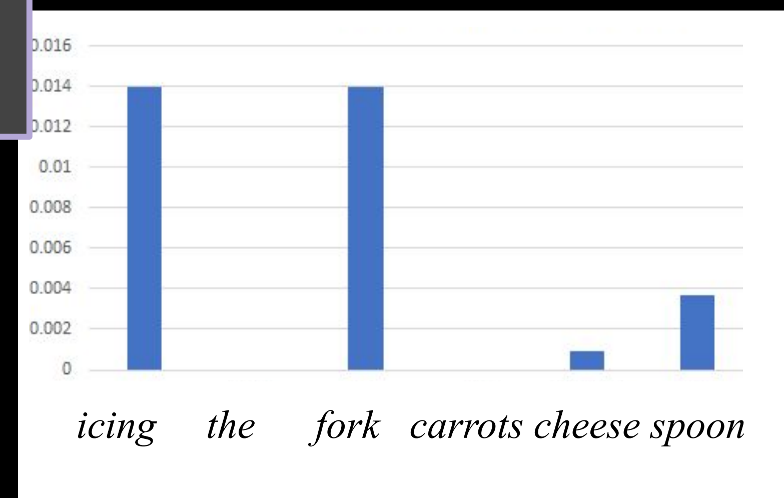
What is the next word in the sequence?

h_t: a vector that we hope "stores" relevant history from previous inputs:

He, at, the, cake, with,

Training Corpus

training (fit, learn)



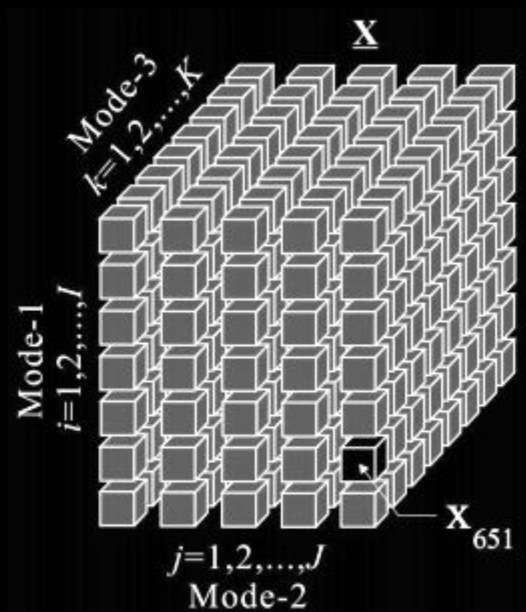
How to program neural networks:

A TensorFlow based approach.

Tensors

Need a workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors



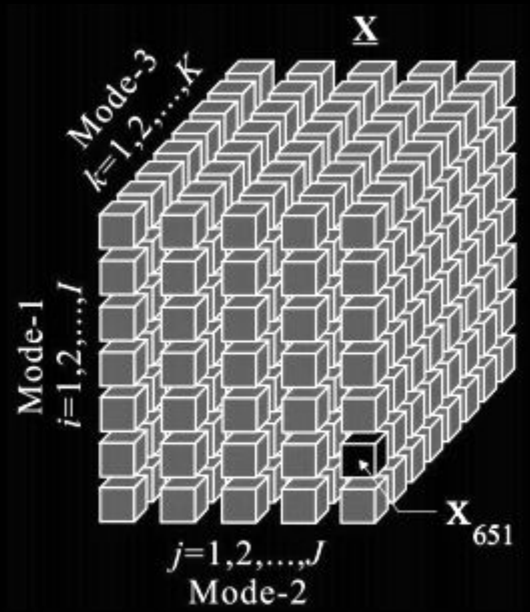
(i.stack.imgur.com)

Tensors

Need a workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors

A multi-dimensional matrix



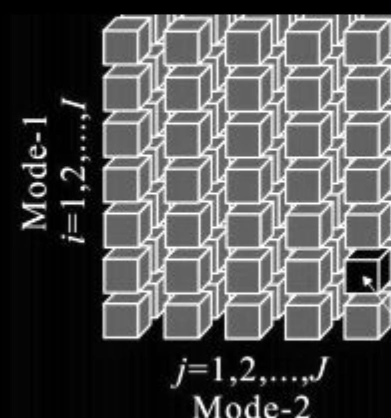
(i.stack.imgur.com)

Tensors

A workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors

A multi-dimensional matrix



A 2-d tensor is just a matrix.

1-d: vector

0-d: a constant / scalar

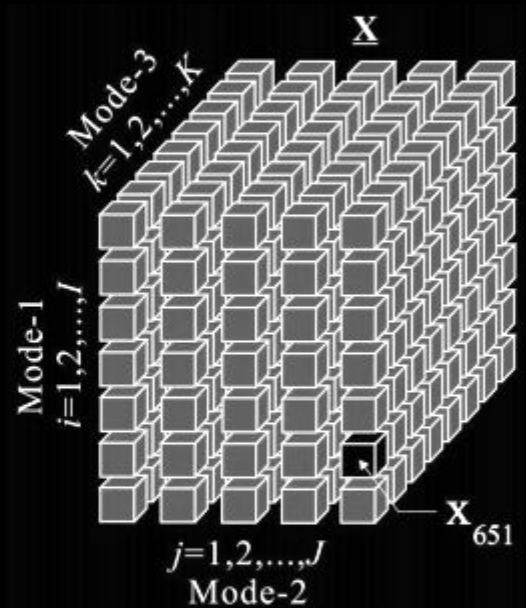
(i.stack.imgur.com)

Tensors

A workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors

A multi-dimensional matrix



A 2-d tensor is just a matrix.

1-d: vector

0-d: a constant / scalar

Linguistic Ambiguity:
“ds” of a Tensor =/=
Dimensions of a Matrix

(i.stack.imgur.com)

Tensors

A workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors

Why?

Efficient, high-level built-in **linear algebra** and **machine learning optimization** *operations* (i.e. transformations).

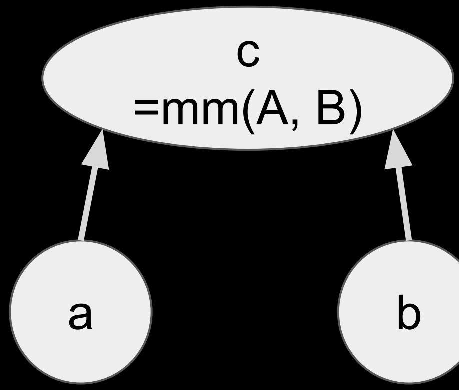
enables complex models, like deep learning

Tensor**Flow**

Operations on tensors are often conceptualized as graphs:

A simple example:

c = tensorflow.matmul(a, b)

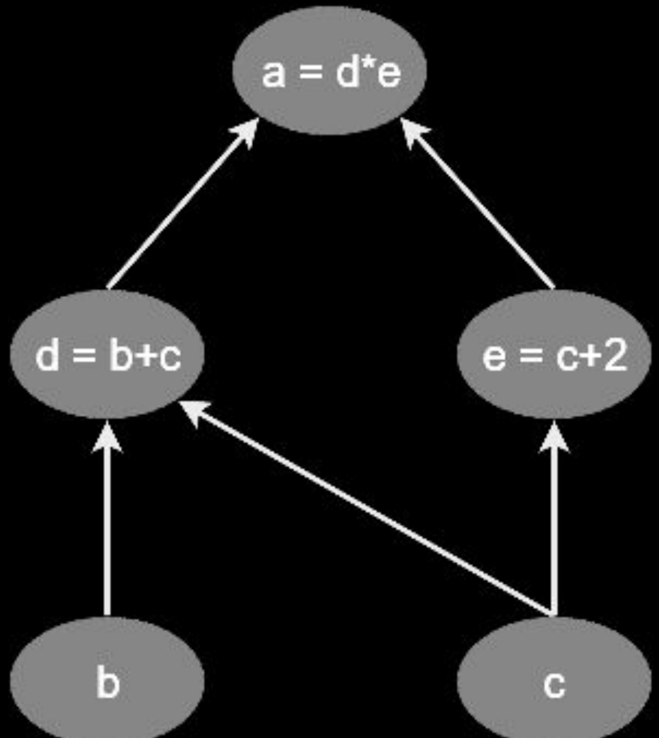


TensorFlow

Operations on tensors are often conceptualized as graphs:

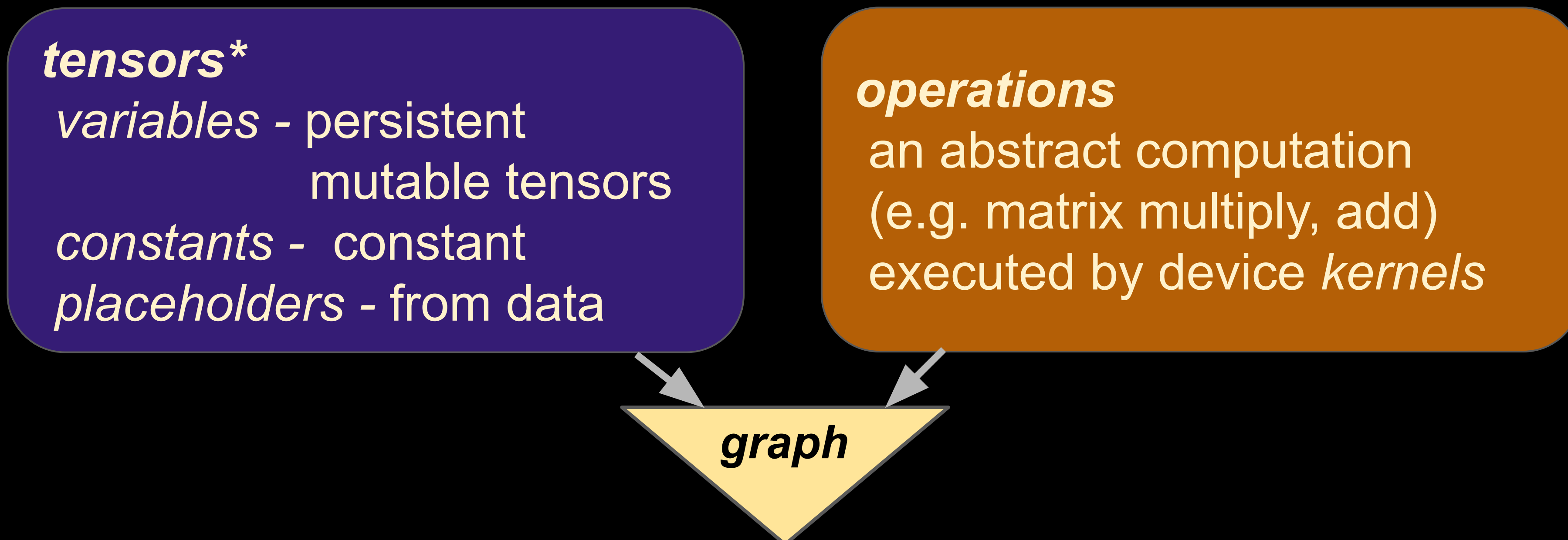
example:

d=b+c
e=c+2
a=d∗e



(Adventures in Machine Learning. *Python TensorFlow Tutorial*, 2017)

Ingredients of a TensorFlow



Ingredients of a TensorFlow

tensors*
variables - persistent mutable tensors
constants - constant
placeholders - from data

- tf.Variable(initial_value, name)
- tf.constant(value, type, name)
- tf.placeholder(type, shape, name)

operations
an abstract computation
(e.g. matrix multiply, add)
executed by device *kernels*

graph

session
defines the environment in which operations *run*.
(like a Spark context)

devices
the specific devices (cpus or gpus) on which to run the session.

Operations

tensors*
variables - persistent
mutable tensors
constants - constant
placeholders - from data

operations
an abstract computation
(e.g. matrix multiply, add)
executed by device *kernels*

Category	**Examples**
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural-net building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

Ingredients of a TensorFlow

tensors*

variables - persistent mutable tensors

constants - constant

placeholders - from data

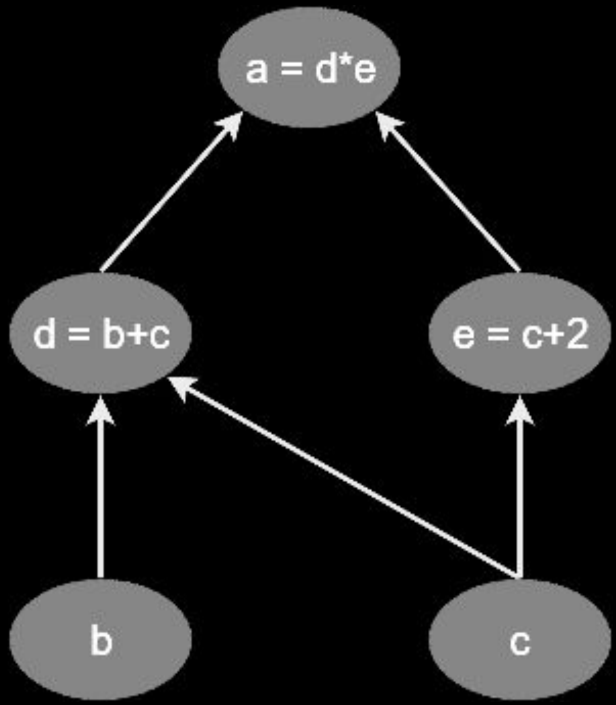
operations

an abstract computation (e.g. matrix multiply, add) executed by device *kernels*

graph

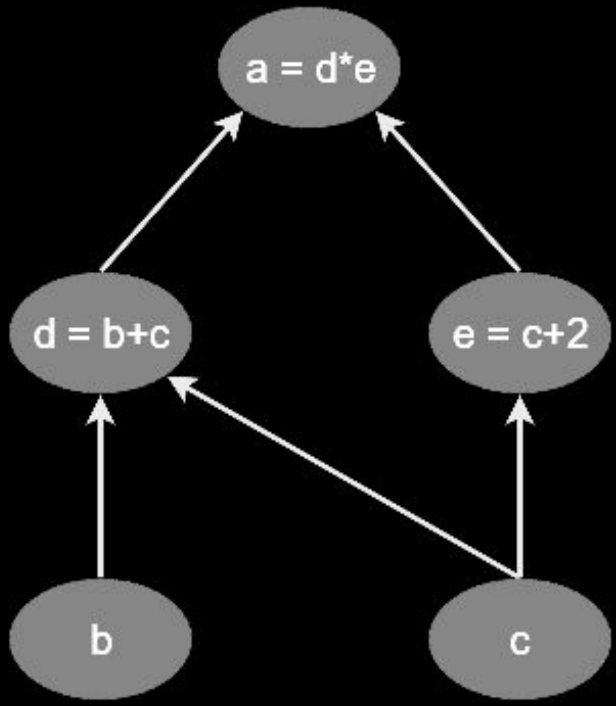
Example

```
import tensorflow as tf
b = tf.constant(1.5, dtype=tf.float32, name="b")
c = tf.constant(3.0, dtype=tf.float32, name="c")




d = b+c
e = c+2

a = d*e
```



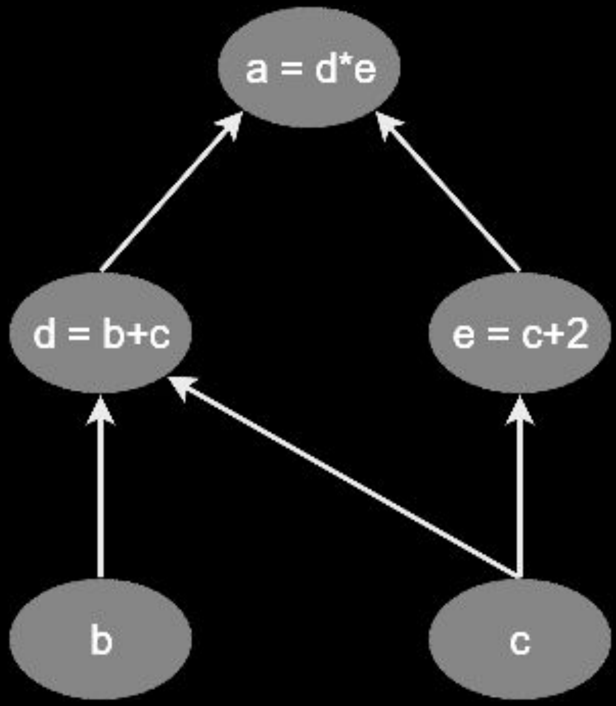
Example

```
import tensorflow as tf
b = tf.constant(1.5, dtype=tf.float32, name="b")
c = tf.constant(3.0, dtype=tf.float32, name="c")




d = b+c  #1.5 + 3
e = c+2  #3+2

a = d*e  #4.5*5 = 22.5
```



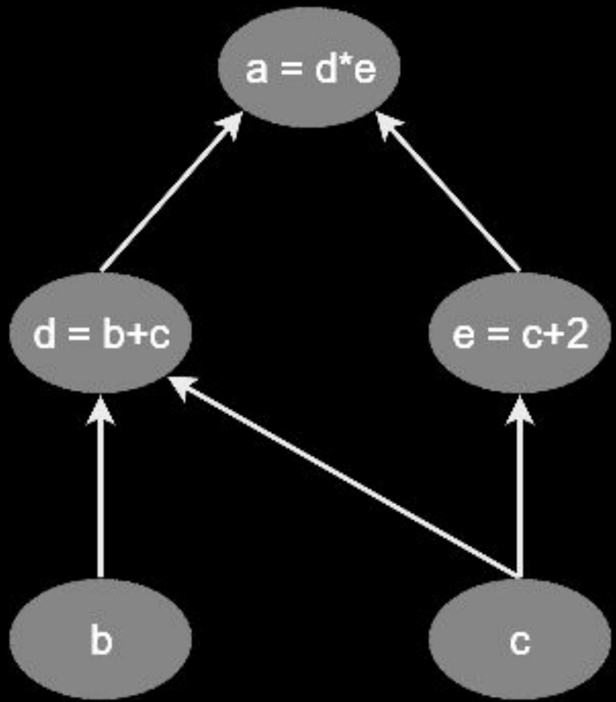
Example (working with 0-d tensors)

```
import tensorflow as tf
b = tf.constant(1.5, dtype=tf.float32, name="b")
c = tf.constant(3.0, dtype=tf.float32, name="c")




d = b+c  #1.5 + 3
e = c+2  #3+2

a = d*e  #4.5*5 = 22.5
```



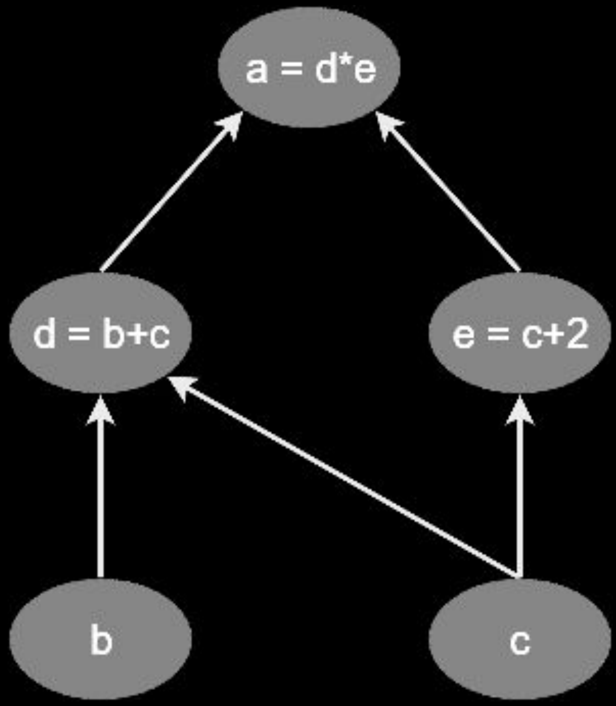
Example: now a 1-d tensor

```
import tensorflow as tf
b = tf.constant([1.5, 2, 1, 4.2],
        dtype=tf.float32, name="b")
c = tf.constant([3, 1, 5, 10],
        dtype=tf.float32, name="c")

d = b+c
e = c+2

a = d*e
```



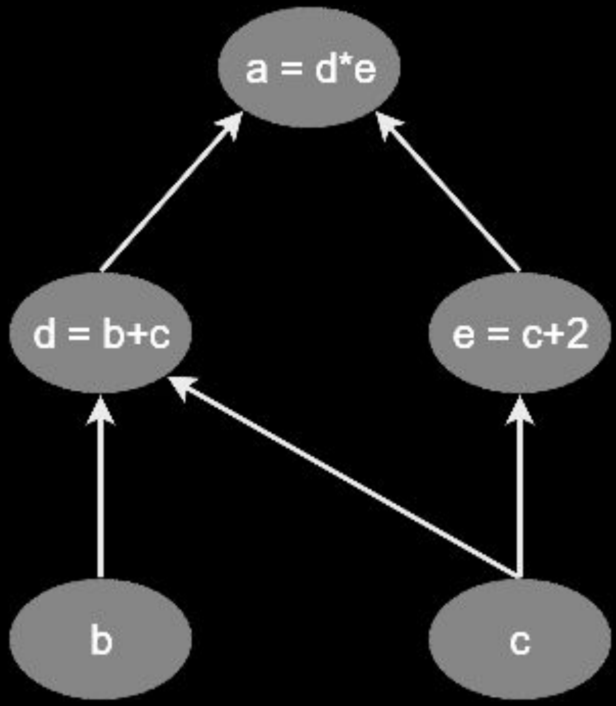
Example: now a 1-d tensor

```
import tensorflow as tf
b = tf.constant([1.5, 2, 1, 4.2],
        dtype=tf.float32, name="b")
c = tf.constant([3, 1, 5, 10],
        dtype=tf.float32, name="c")

d = b+c  #[4.5,  3, 6, 14.2]
e = c+2  #[5, 4, 7, 12]

a = d*e  #??
```



Example: now a 2-d tensor

```
import tensorflow as tf
b = tf.constant([[...], [...]],
          dtype=tf.float32, name="b")
c = tf.constant([[...], [...]],
          dtype=tf.float32, name="c")

d = b+c
e = c+2

a = tf.matmul(d,e)
```



Example: Logistic Regression

```
X = tf.constant([[...], [...]],
        dtype=tf.float32, name="X")
y = tf.constant([...],
        dtype=tf.float32, name="y")
# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1.,
1.), name = "beta")
```

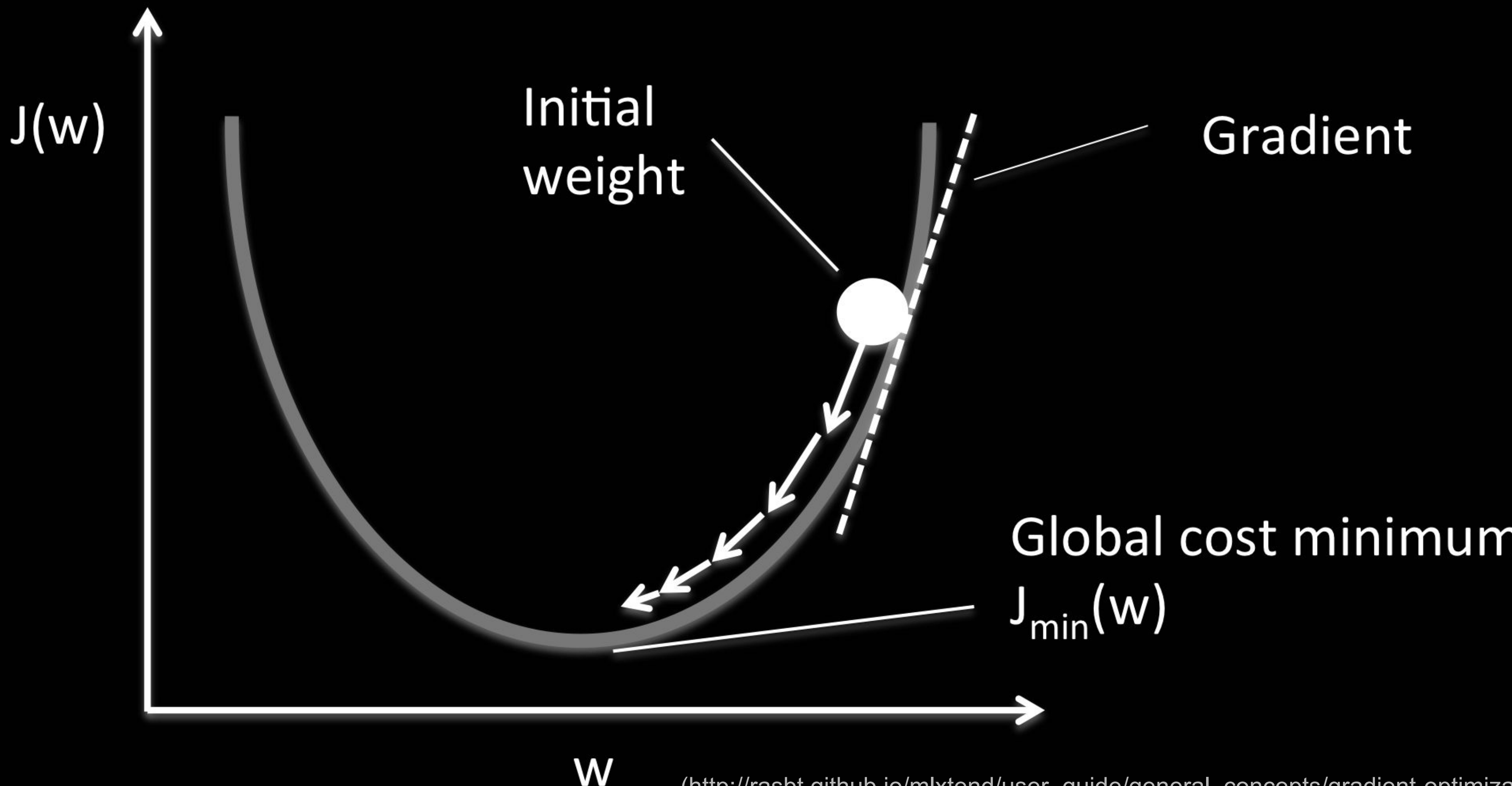
Example: Logistic Regression

```
X = tf.constant([[...], [...]],
        dtype=tf.float32, name="X")
y = tf.constant([...],
        dtype=tf.float32, name="y")
# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1.,
1.), name = "beta")

#then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")
```

Example: Logistic Regression

```
X = tf.constant([[...], [...]],
        dtype=tf.float32, name="X")
y = tf.constant([...],
        dtype=tf.float32, name="y")
# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1.,
1.), name = "beta")

#then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")

#Define a *cost function* to minimize:
penalizedCost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred),
reduction_indices=1)) #conceptually like |y - y_pred|
```

Optimizing Parameters -- derived from **gradients**



(http://rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/)

Example: Logistic Regression

```
X = tf.constant([[...], [...]],
        dtype=tf.float32, name="X")
y = tf.constant([...],
        dtype=tf.float32, name="y")
# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1.,
1.), name = "beta")

#then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")

#Define a *cost function* to minimize:
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred),
reduction_indices=1))
```

Example: Logistic Regression

```
X = tf.constant([[...], [...]], dtype=tf.float32, name="X")
y = tf.constant([...],     dtype=tf.float32, name="y")
# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1., 1.), name = "beta")
#then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")
#Define a *cost function* to minimize:
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred), reduction_indices=1))

#define how to optimize and initialize:
optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()
```

Example: Logistic Regression

```
X = tf.constant([[...], [...]], dtype=tf.float32, name="X")
y = tf.constant([...],     dtype=tf.float32, name="y")
# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1., 1.), name = "beta")
#then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")
#Define a *cost function* to minimize:
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred), reduction_indices=1))

#define how to optimize and initialize:
optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()

#iterate over optimization:
with tf.Session() as sess:
 sess.run(init)
 for epoch in range(n_epochs):
  sess.run(training_op)
 #done training, get final beta:
 best_beta = beta.eval()
```

Language Modeling

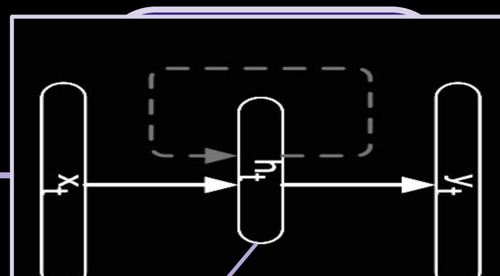
Task: Estimate $P(w_n \mid w_1, w_2, \ldots, w_{n-1})$

:probability of a next word given history

P(fork | He ate the cake with the) = ?

Last word

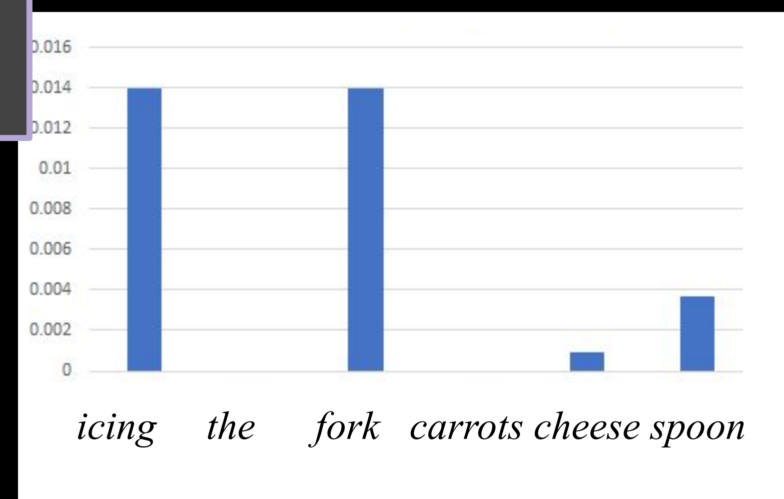
(the)



What is the next word in the sequence?

h_t: a vector that we hope "stores" relevant history from previous inputs:

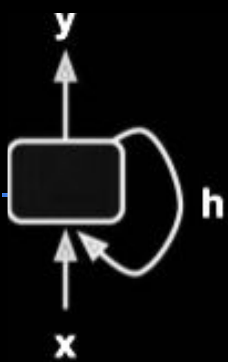
He, at, the, cake, with,



Training Corpus

training (fit, learn)

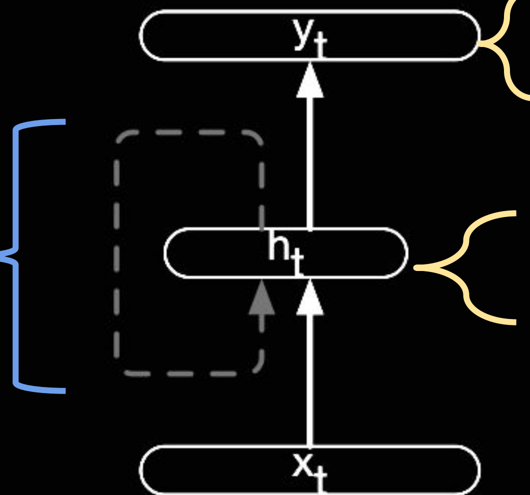
Example: RNN




```
...
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = tf.tanh(tf.matmul(U,h(i-1))+ tf.matmul(W,x(i))) #update hidden state
    y(i) = tf.softmax(tf.matmul(V, h(i))) #update output
...
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
```

Example: RNN

"hidden layer"



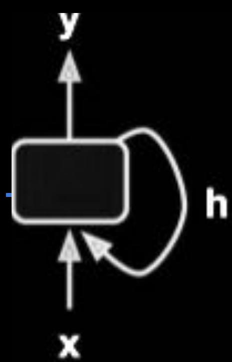
$y_{(t)} = f(h_{(t)}W)$

Activation Function

$h_{(t)} = g(h_{(t-1)}U + x_{(t)}V)$ #update hidden

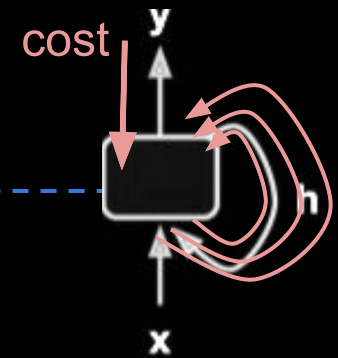
```
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
```

Example: RNN




```
...
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = tf.tanh(tf.matmul(U,h(i-1))+ tf.matmul(W,x(i))) #update hidden state
    y(i) = tf.softmax(tf.matmul(V, h(i))) #update output
...
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
```

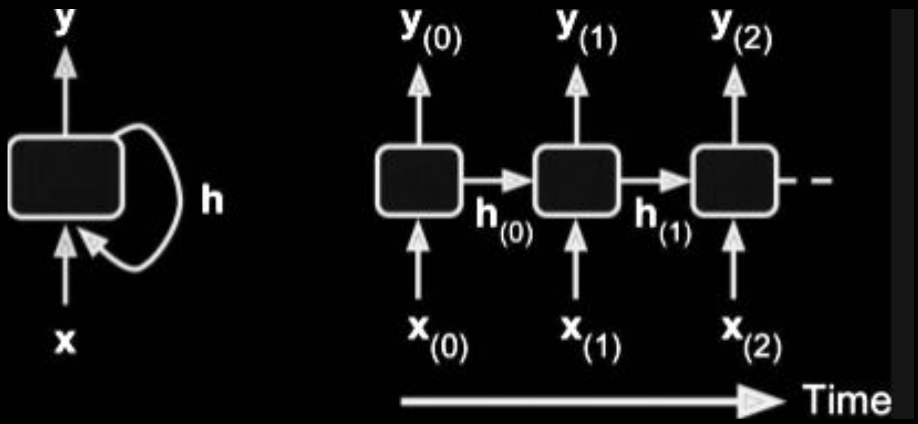
Optimization:



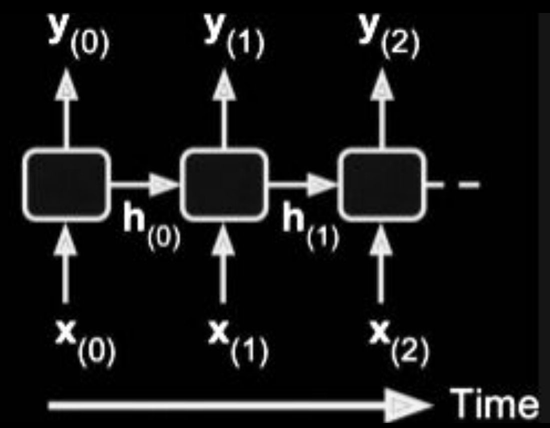
Backward Propagation

```
...
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = tf.tanh(tf.matmul(U,h(i-1))+ tf.matmul(W,x(i))) #update hidden state
    y(i) = tf.softmax(tf.matmul(V, h(i))) #update output
...
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
```

Solution: Unrolling



Solution: Unrolling



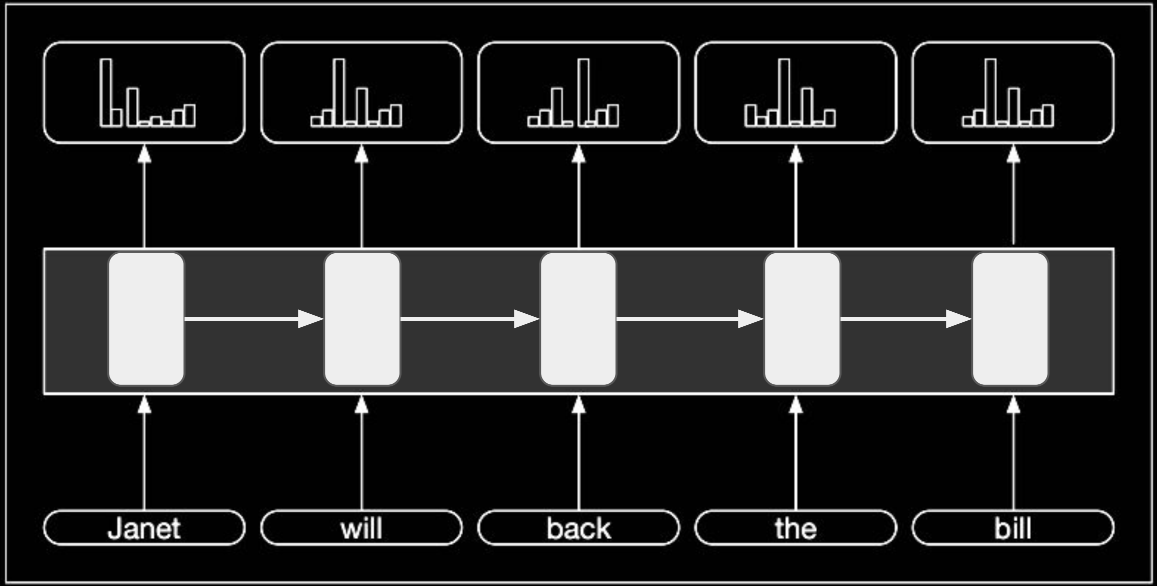
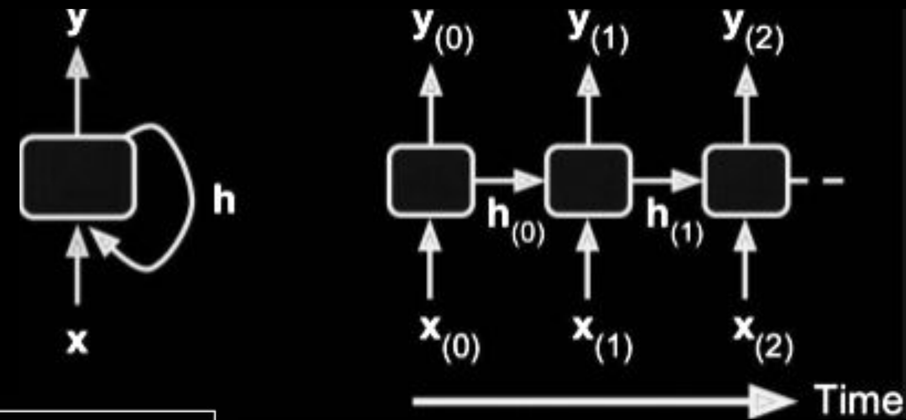
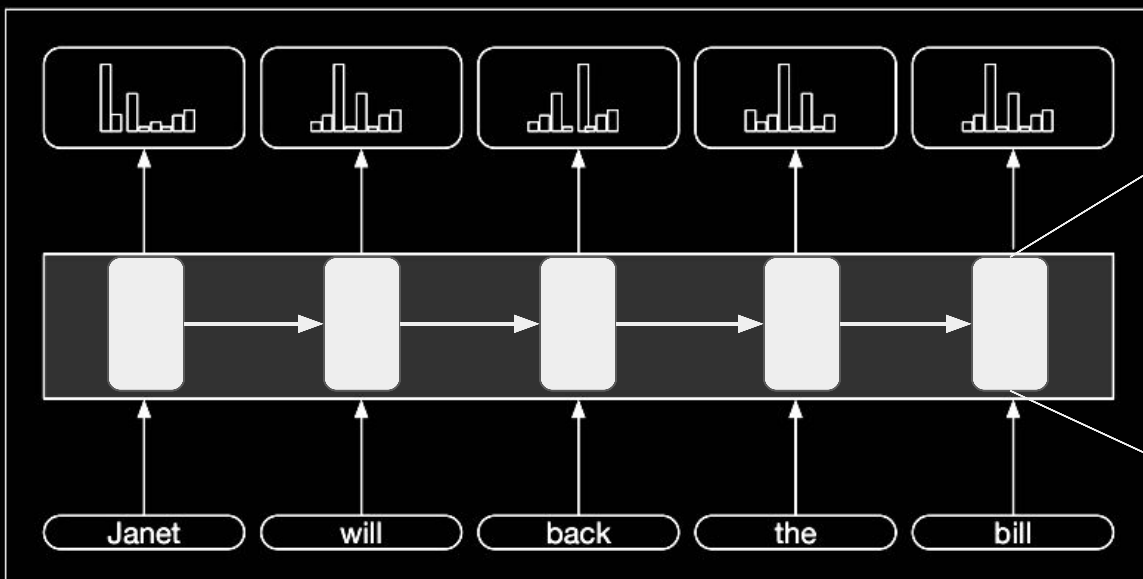


Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

Solution: Unrolling





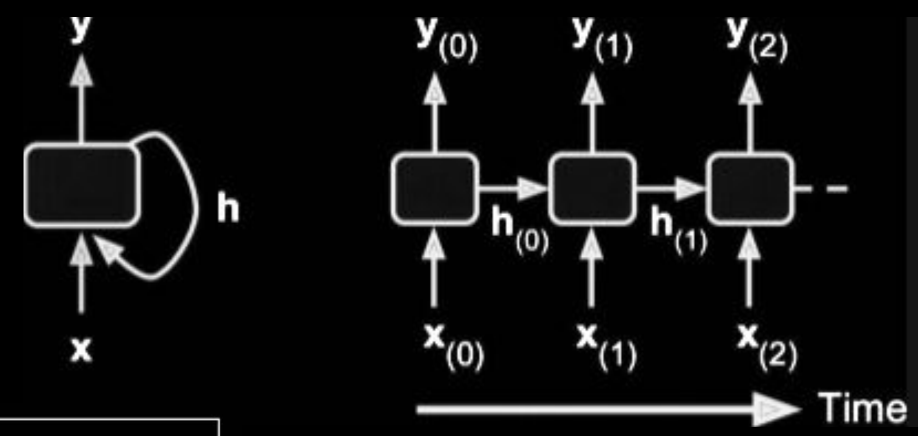
$$y_{(t)} = f(h_{(t)}W)$$

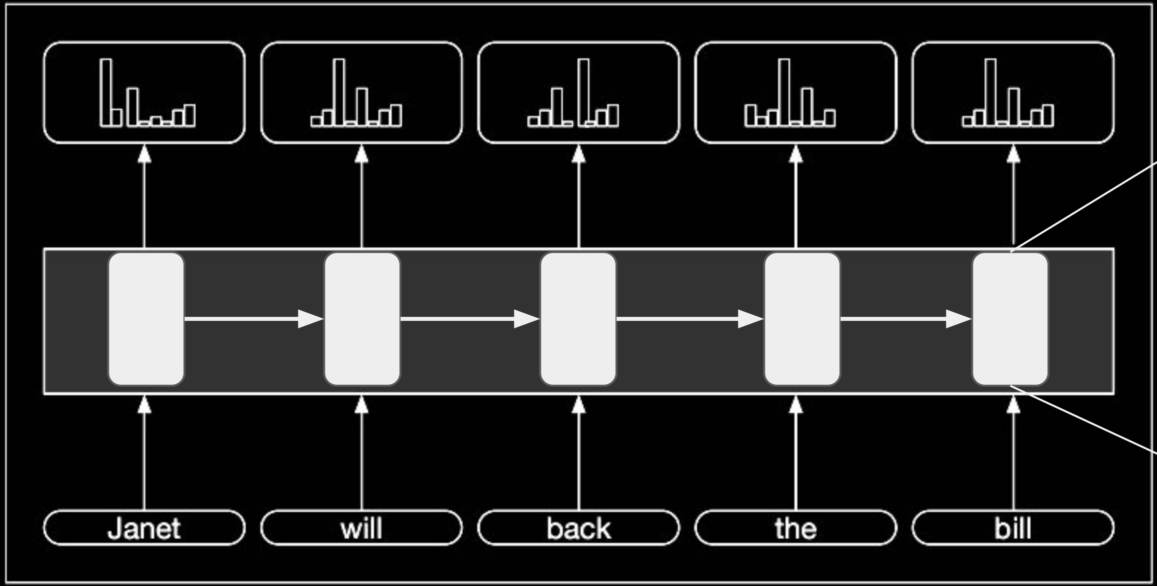
Activation Function

$$h_{(t)} = g(h_{(t-1)}U + x_{(t)}V)$$

Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

Solution: Unrolling





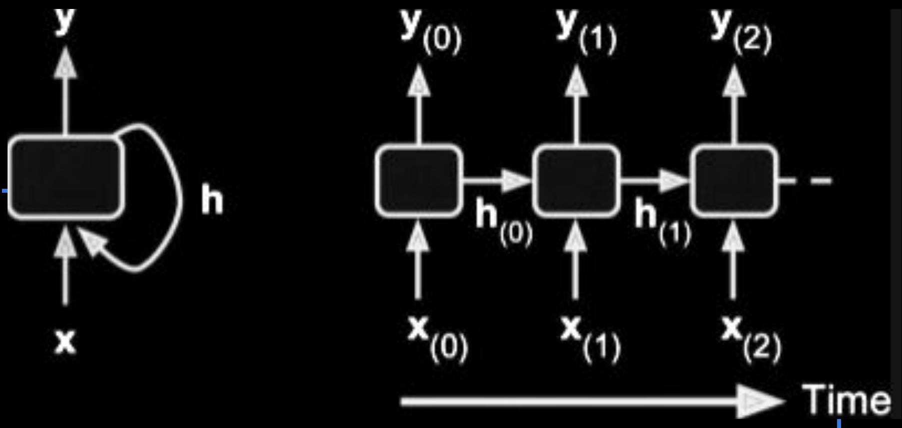
$$y_{("bill")} = f(h_{("bill")}W)$$

Activation Function

$$h_{("bill")} = g(h_{("the")}U + x_{("bill")}V)$$

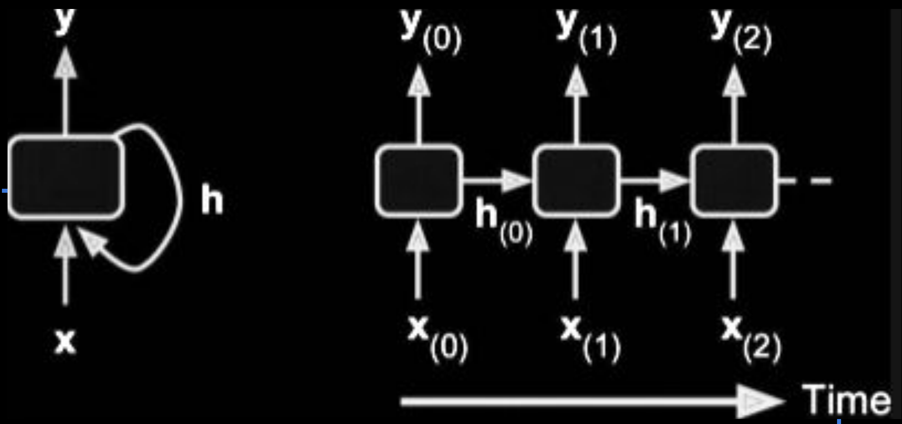
Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

Example: Forward Pass



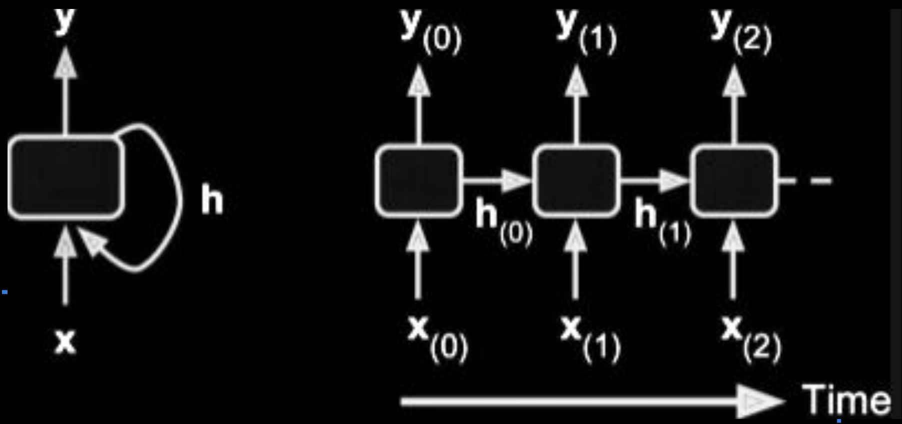

```
#define forward pass graph:

h(i) = tf.nn.relu(tf.matmul(U,h(i-1))+ tf.matmul(W,x(i))) #update hidden state
y(i) = tf.softmax(tf.matmul(V, h(i))) #update output
```

Example: Forward Pass



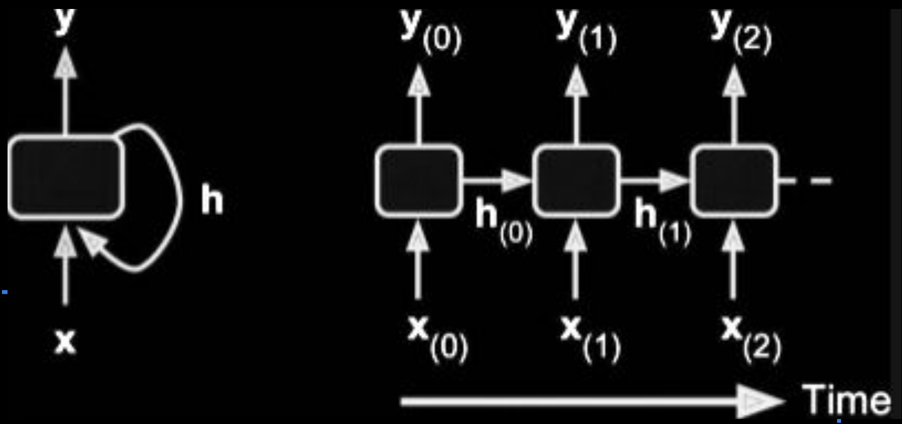

```
hidden_size, output_size = 5, 1


#define forward pass graph:

h(i) = tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu)
y(i) = tf.softmax(tf.matmul(V, h(i))) #update output
```

Example: Forward Pass




```
hidden_size, output_size = 5, 1



#define forward pass graph:

h(i) = tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu)
y(i) = tf.softmax(tf.matmul(V, h(i))) #update output
```

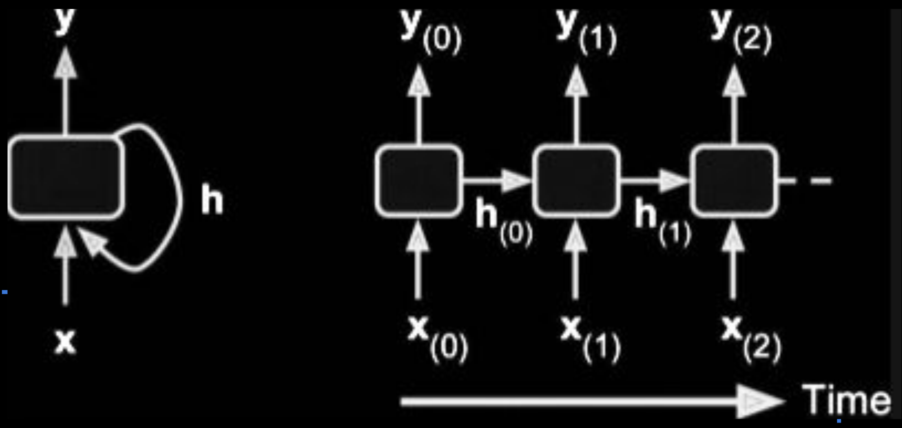
Example: Forward Pass




```
hidden_size, output_size = 5, 1



#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
     tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
     output_size = output_size
```

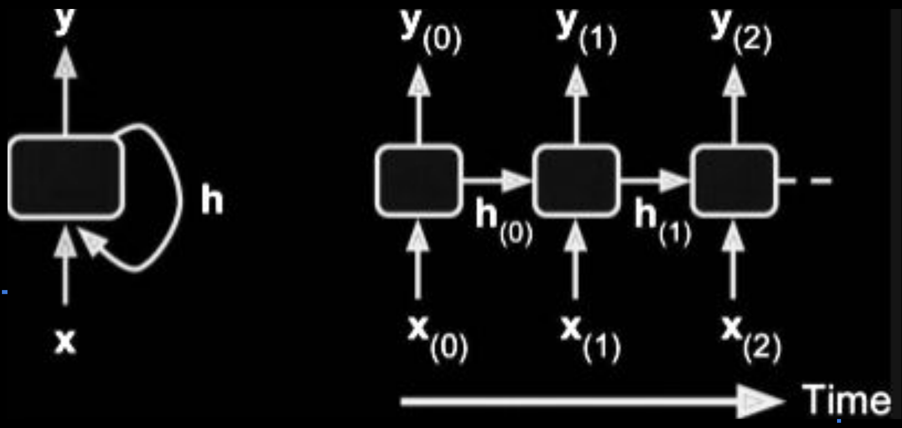
~~$y_{(i)}$ = tf.softmax(tf.matmul(V, $h_{(i)}$)) #update output~~

Example: Forward Pass



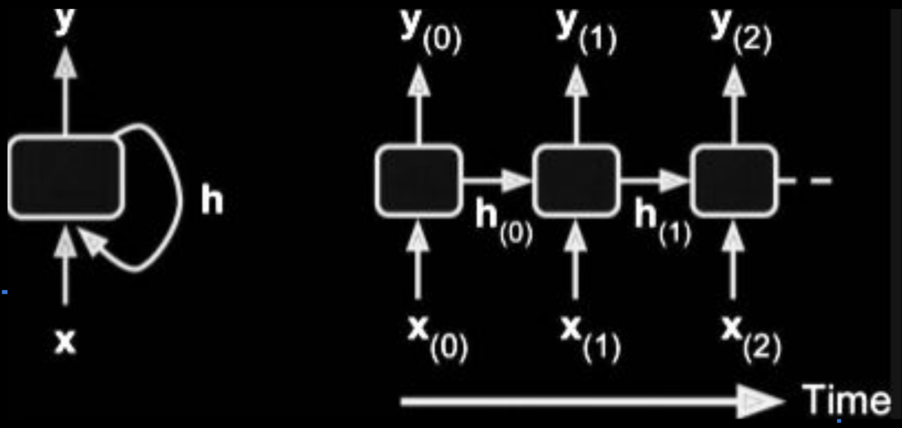

```python
hidden_size, output_size = 5, 1



#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
     tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
     output_size = output_size
```

Example: Forward Pass



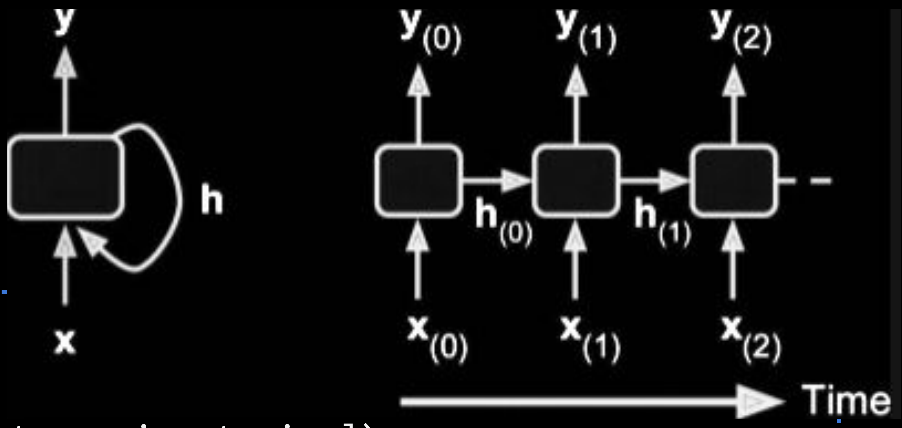

```
hidden_size, output_size = 5, 1



#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
     tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
     output_size = output_size
#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs))  #softmax cost
optimizer = tf.train.AdamOptimizer(learing_rate=learning_rate)
```

Example: Forward Pass



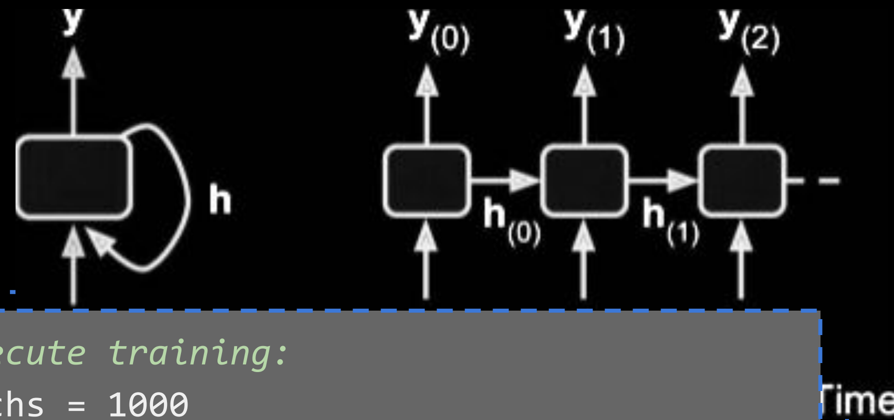

```
hidden_size, output_size = 5, 1



#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
     tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
     output_size = output_size
#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs))  #softmax cost
optimizer = tf.train.AdamOptimizer(learing_rate=learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()
```

Example: Forward Pass




```
hidden_size, output_size = 5, 1
input_size, unroll_steps = 10, 20
X = tf.placeholder(tf.float32, [None, unroll_steps, input_size])
y = tf.placeholder(tf.float32, [None, unroll_steps, output_size])
#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
     tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
     output_size = output_size
#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs))  #softmax cost
optimizer = tf.train.AdamOptimizer(learing_rate=learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()
```

Example: Forward Pass

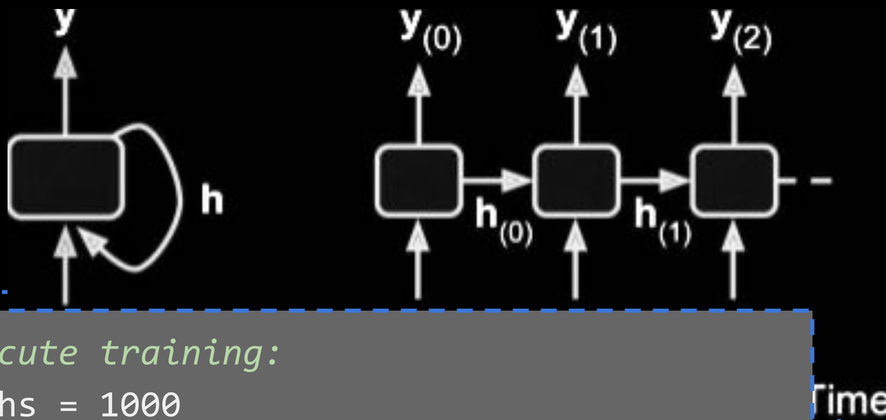



```
hidden_size, output_size = 5, 1
input_size, unroll_steps = 10, 20
X = tf.placeholder(tf.float32, [None, unr
y = tf.placeholder(tf.float32, [None, unr
#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWra
     tf.contrib.BasicRNNCell(num_units=hi
     output_size = output_size
#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*t
optimizer = tf.train.AdamOptimizer(learin
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()
```

```
#execute training:
epochs = 1000
batch_size = 50
with tf.Session() as sess:
  init.run()
```

(Geron, 2017)

Example: Forward Pass

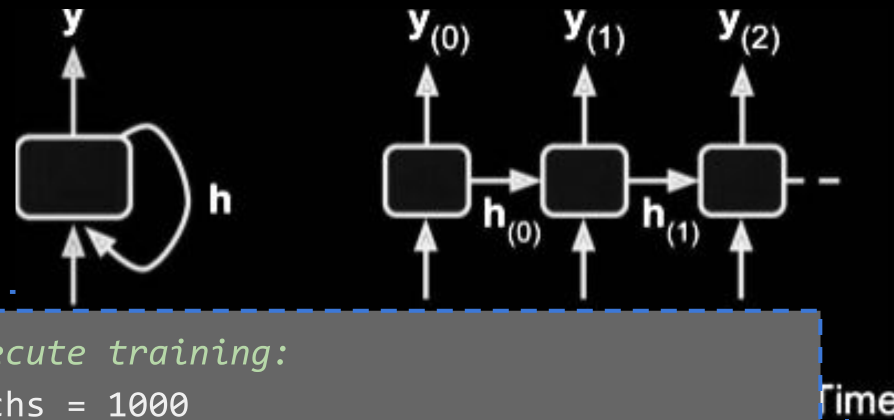



```
hidden_size, output_size = 5, 1
input_size, unroll_steps = 10, 20
X = tf.placeholder(tf.float32, [None, unr
y = tf.placeholder(tf.float32, [None, unr
#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWra
     tf.contrib.BasicRNNCell(num_units=hi
     output_size = output_size
#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*t
optimizer = tf.train.AdamOptimizer(learin
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()
```

```
#execute training:
epochs = 1000
batch_size = 50
with tf.Session() as sess:
  init.run()
  for iter in range(epochs)
     X_batch, y_batch = ...#fetch next batch
     sess.run(training_op, feed_dict=\
              {X:X_batch, y:y_batch})
```

(Geron, 2017)

Example: Forward Pass



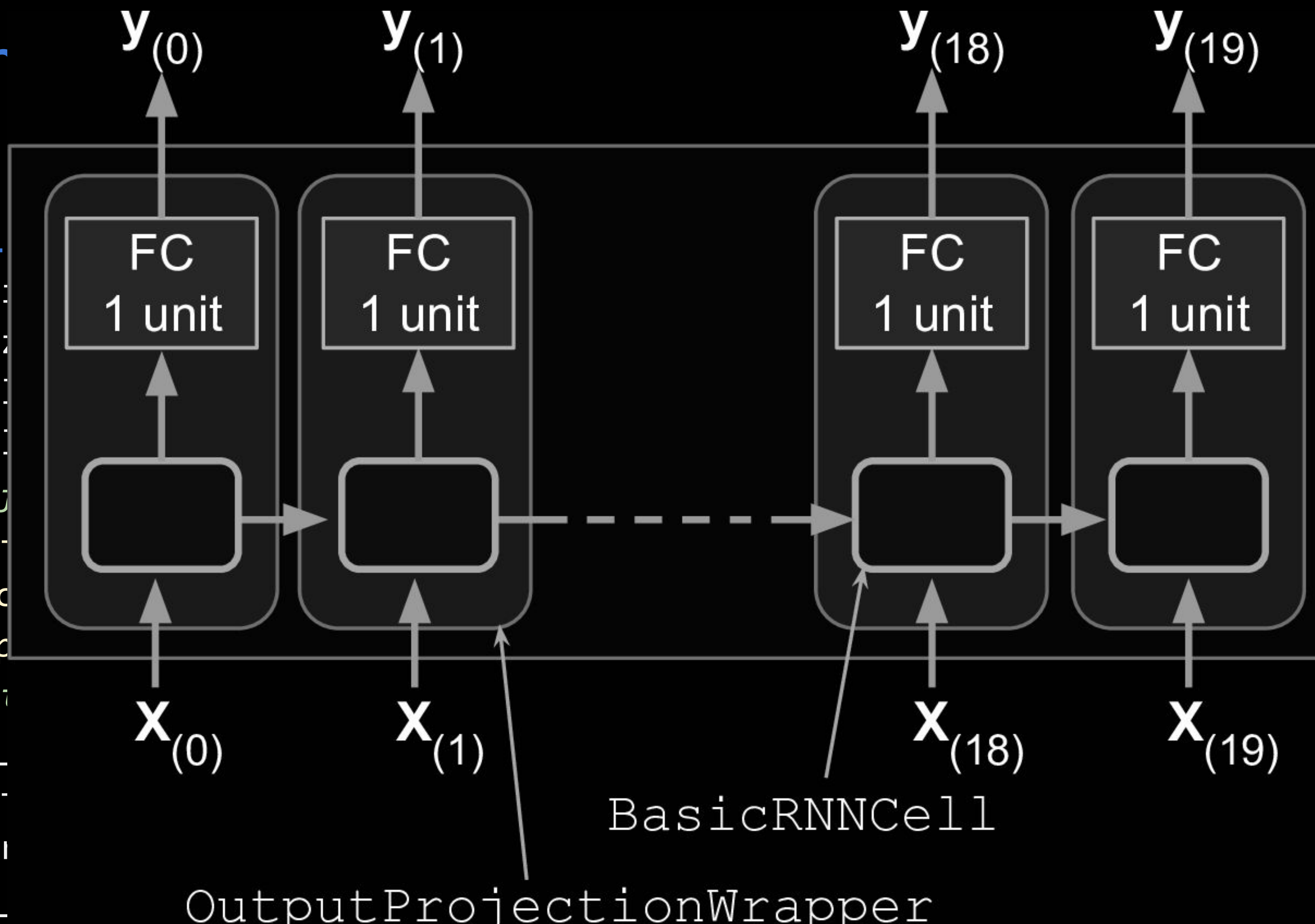

```
hidden_size, output_size = 5, 1
input_size, unroll_steps = 10, 20
X = tf.placeholder(tf.float32, [None, unr
y = tf.placeholder(tf.float32, [None, unr
#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWra
     tf.contrib.BasicRNNCell(num_units=hi
     output_size = output_size
#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*tf
optimizer = tf.train.AdamOptimizer(learin
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()
```

```
#execute training:
epochs = 1000
batch_size = 50
with tf.Session() as sess:
  init.run()
  for iter in range(epochs)
     X_batch, y_batch = …#fetch next batch
     sess.run(training_op, feed_dict=\
              {X:X_batch, y:y_batch})
     if iter % 100 == 0:
          c = cost.eval(feed_dict=\
              {X:X_batch, y:y_batch})
          print(iter, "\tcost: ", c)
```

(Geron, 2017)

Exar

```
hidden_s
input_si
X = tf.p
y = tf.p
#define
cell = t
     tf.c
     outp
#define
learning_
cost = t
optimize
training_
init = tf.global_variables_initializer()
```




```
ext batch
ct=\
ch})
\
ch})
c)
```

(Geron, 2017)

Neural Networks: Graphs of Operations (excluding the optimization nodes)

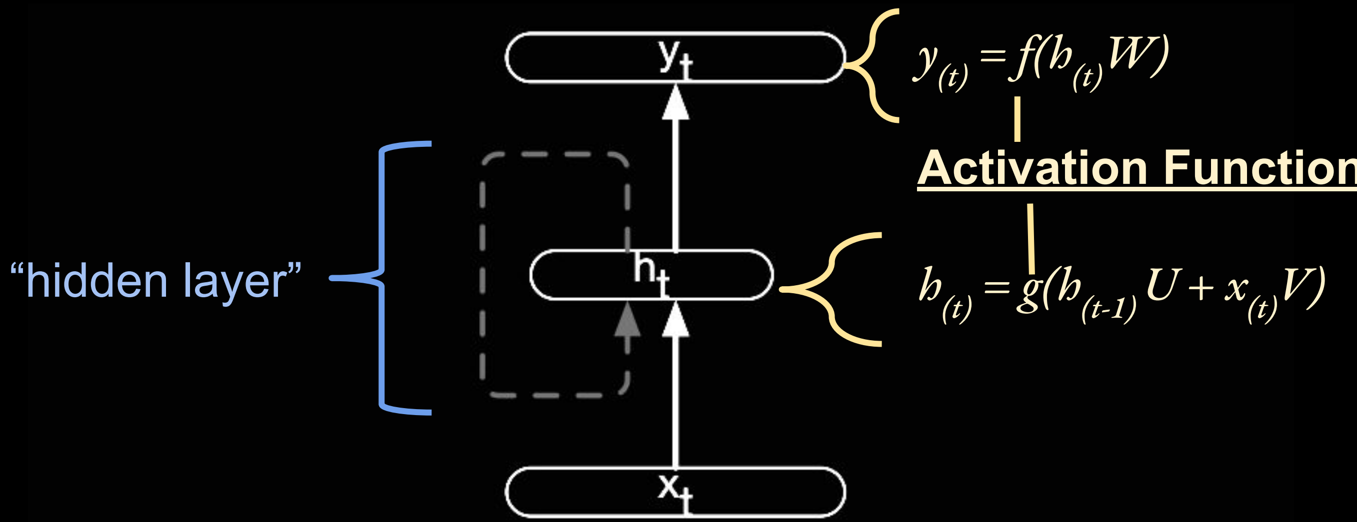


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Language Modeling

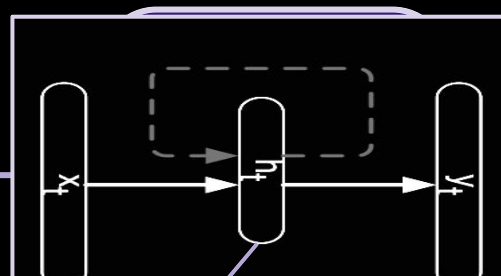
Task: Estimate $P(w_n \mid w_1, w_2, \ldots, w_{n-1})$

:probability of a next word given history

P(fork | He ate the cake with the) = ?

Last word

(the)



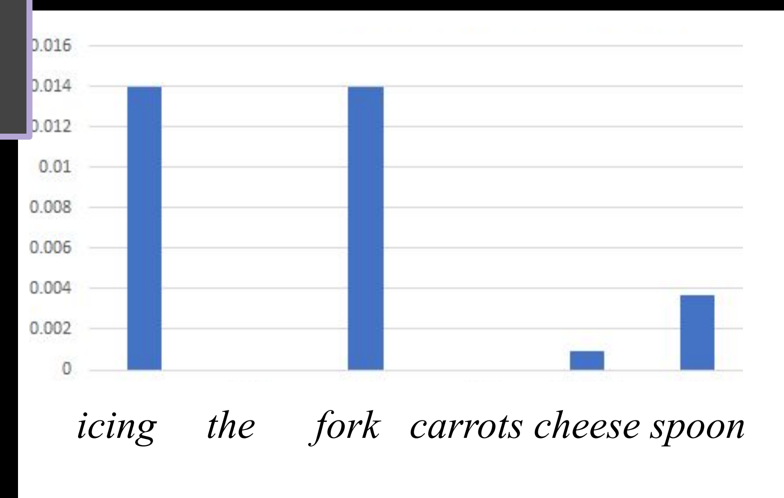
What is the next word in the sequence?

h_t: a vector that we hope "stores" relevant history from previous inputs:

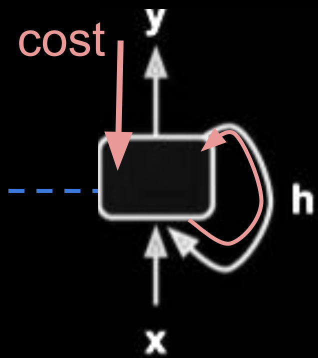
He, at, the, cake, with,

Training Corpus

training (fit, learn)



Optimization:

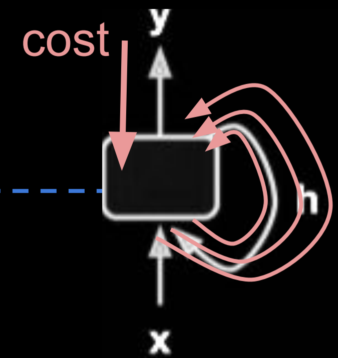


Backward Propagation

To find the gradient for the overall graph, we use **back propogation,** which *essentially* chains together the gradients for each node (function) in the graph.

```
...
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = tf.tanh(tf.matmul(U,h(i-1))+ tf.matmul(W,x(i))) #update hidden state
    y(i) = tf.softmax(tf.matmul(V, h(i))) #update output
...
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
```

Optimization:



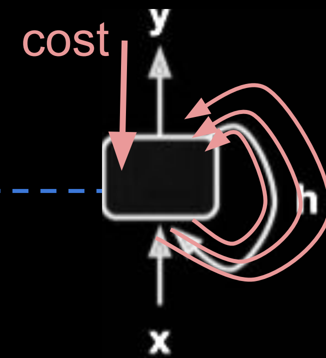
Backward Propagation

```
...
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = tf.tanh(tf.matmul(U,
state
    y(i) = tf.softmax(tf.matmul
...
cost = tf.reduce_mean(-tf.redu
```

To find the gradient for the overall graph, we use **back propogation,** which *essentially* chains together the gradients for each node (function) in the graph.

With many recursions, the gradients can vanish or explode (become too large or small for floating point operations).

Optimization:



Backward Propagation

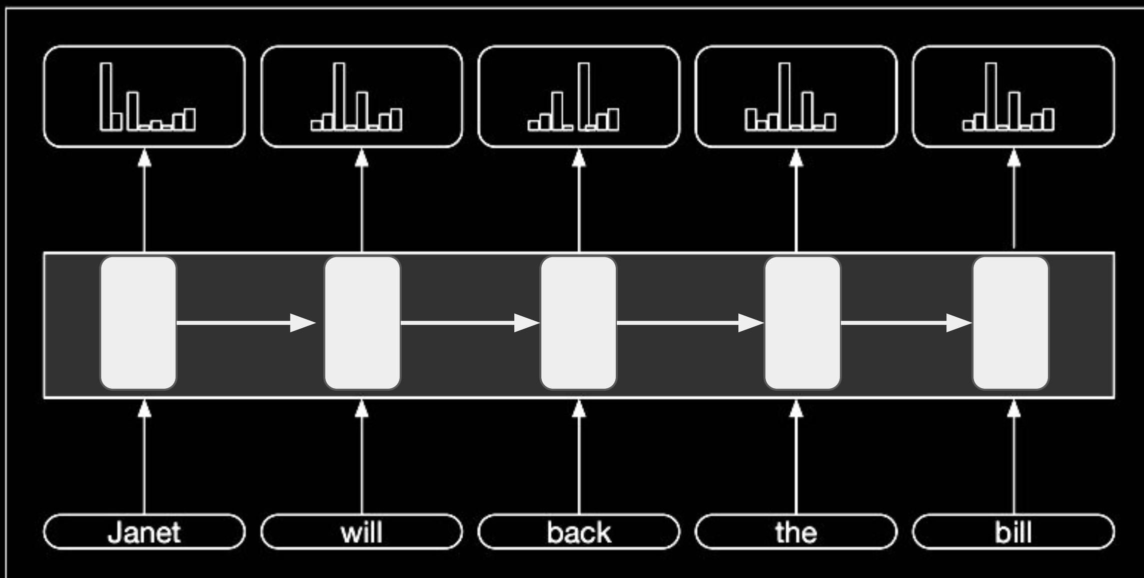


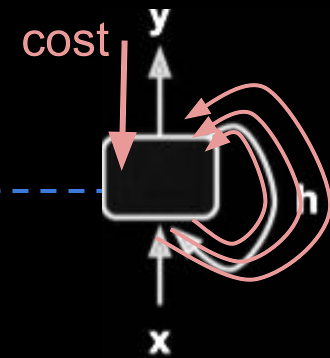
Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

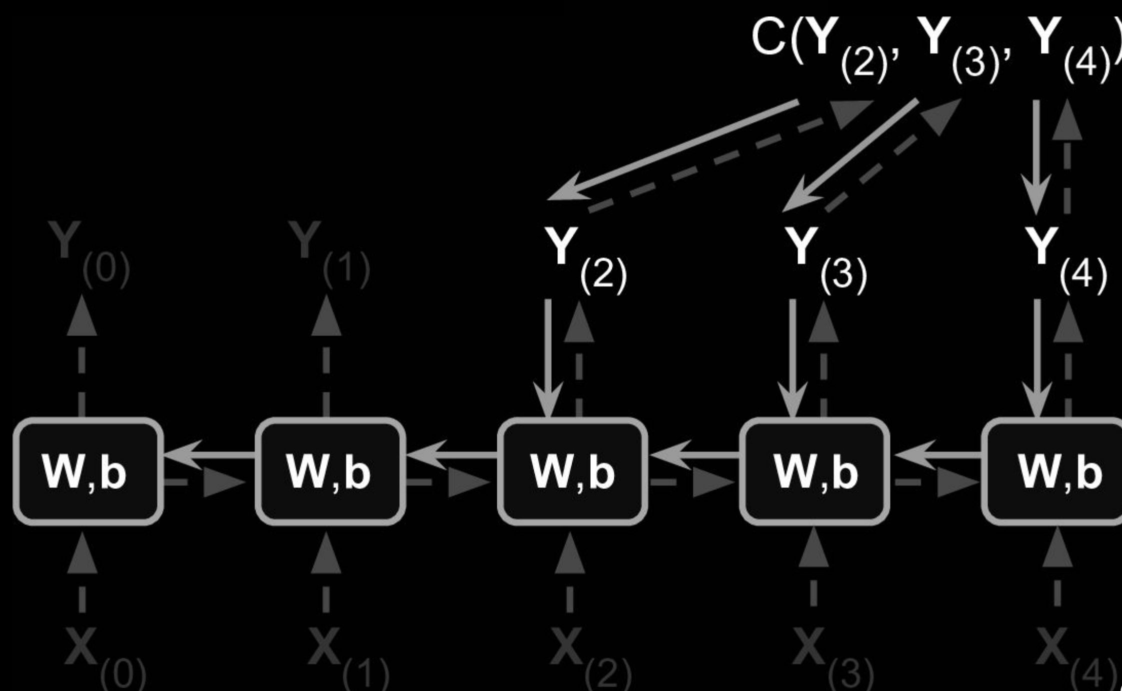
for the overall graph, we
ion, which *essentially*
gradients for each node
ph.

ns, the gradients can
become too large or
int operations).

Optimization:

Backward Propagation





(Geron, 2017)

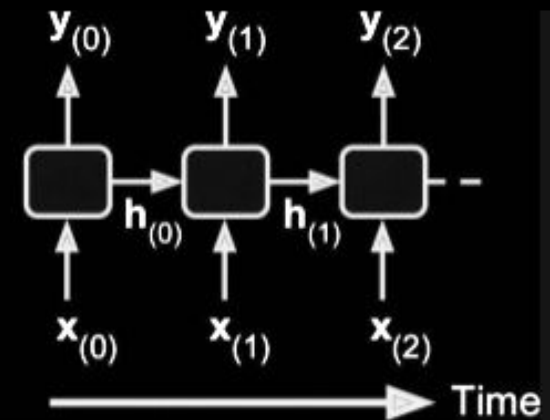
How to address exploding and vanishing gradients?

Ad Hoc approaches: e.g. stop backprop iterations very early. “clip” gradients when too high.

How to address exploding and vanishing gradients?

Dominant approach: Use Long Short Term Memory Networks (LSTM)

RNN model

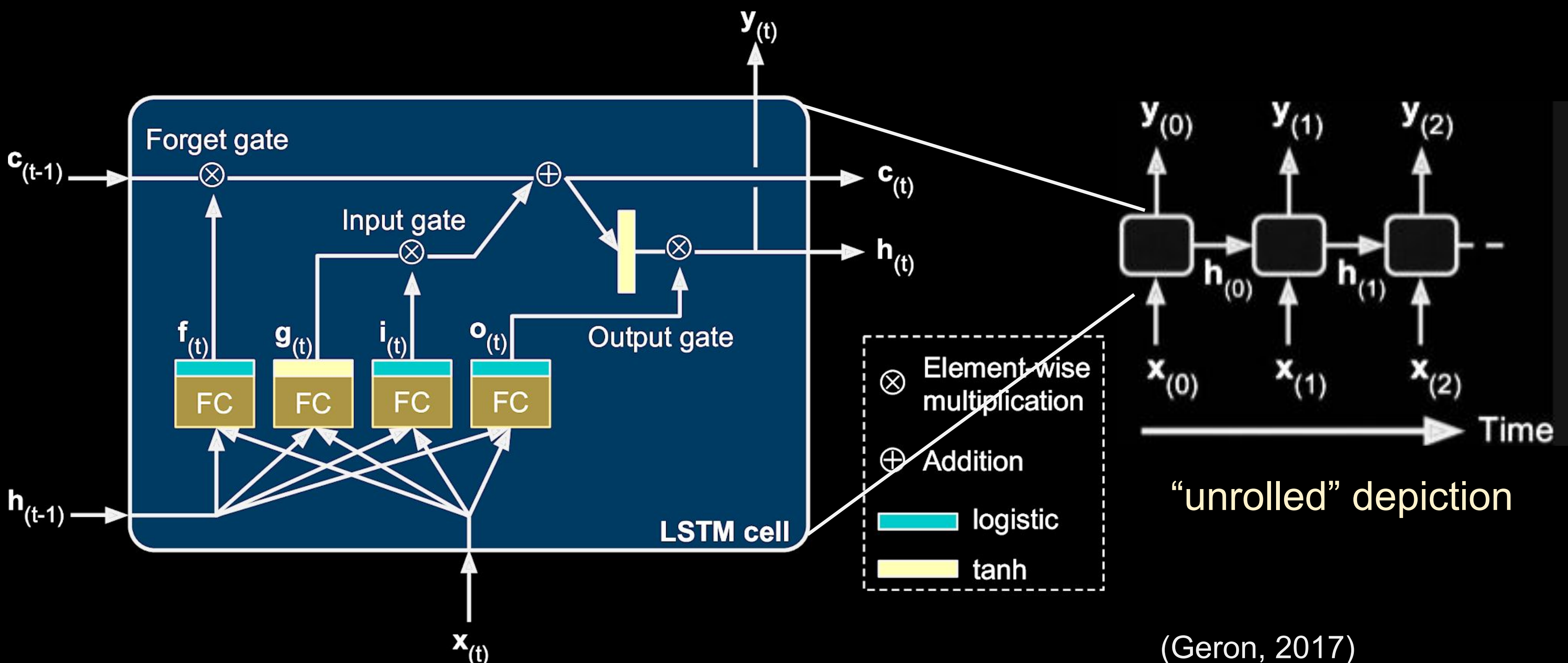


“unrolled” depiction

(Geron, 2017)

How to address exploding and vanishing gradients?

The LSTM Cell

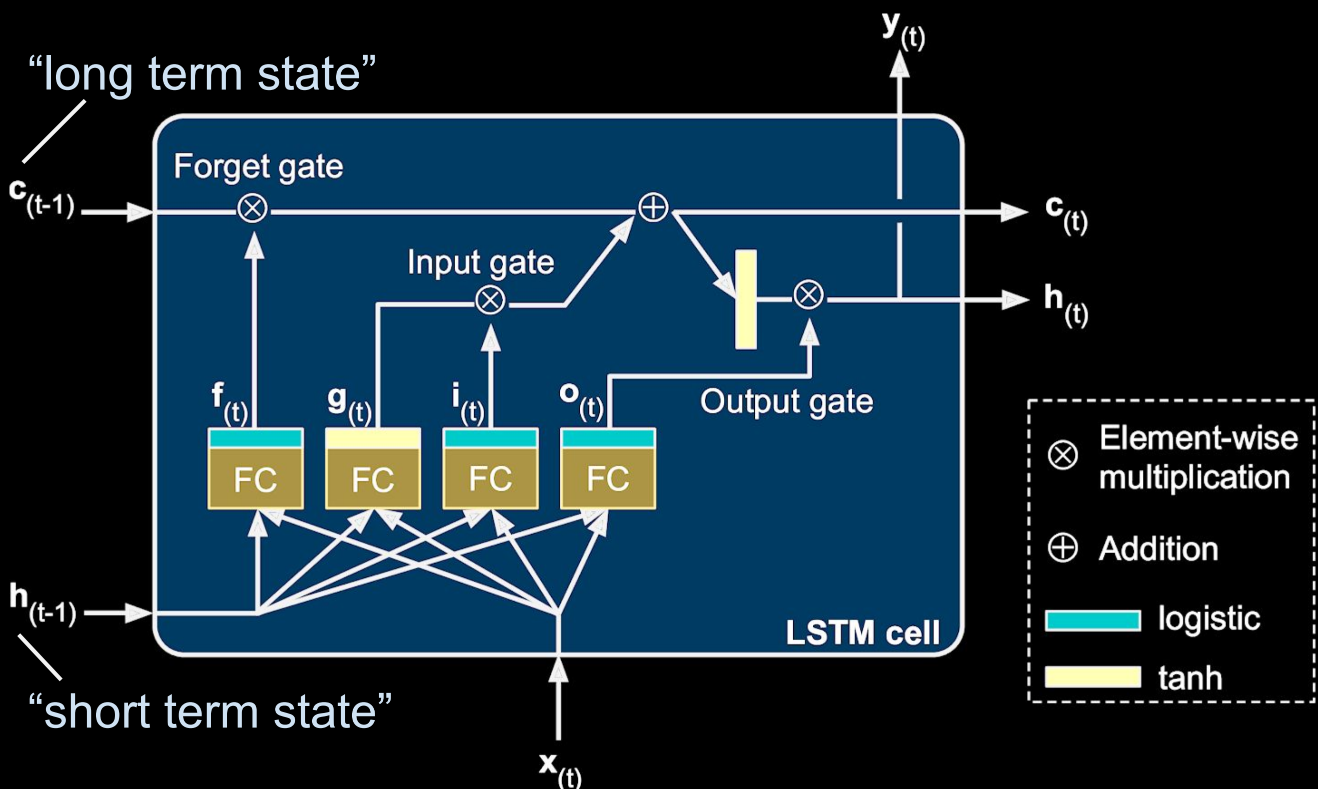


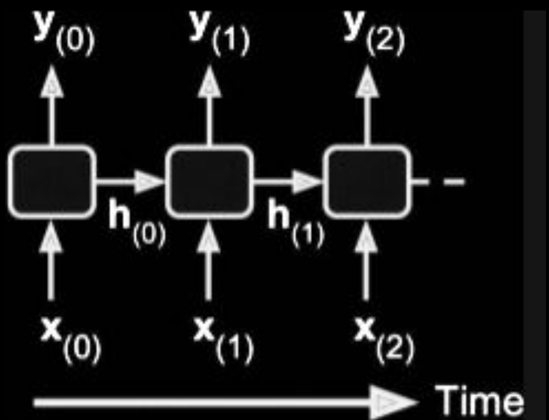
"unrolled" depiction

(Geron, 2017)

How to address exploding and vanishing gradients?

The LSTM Cell



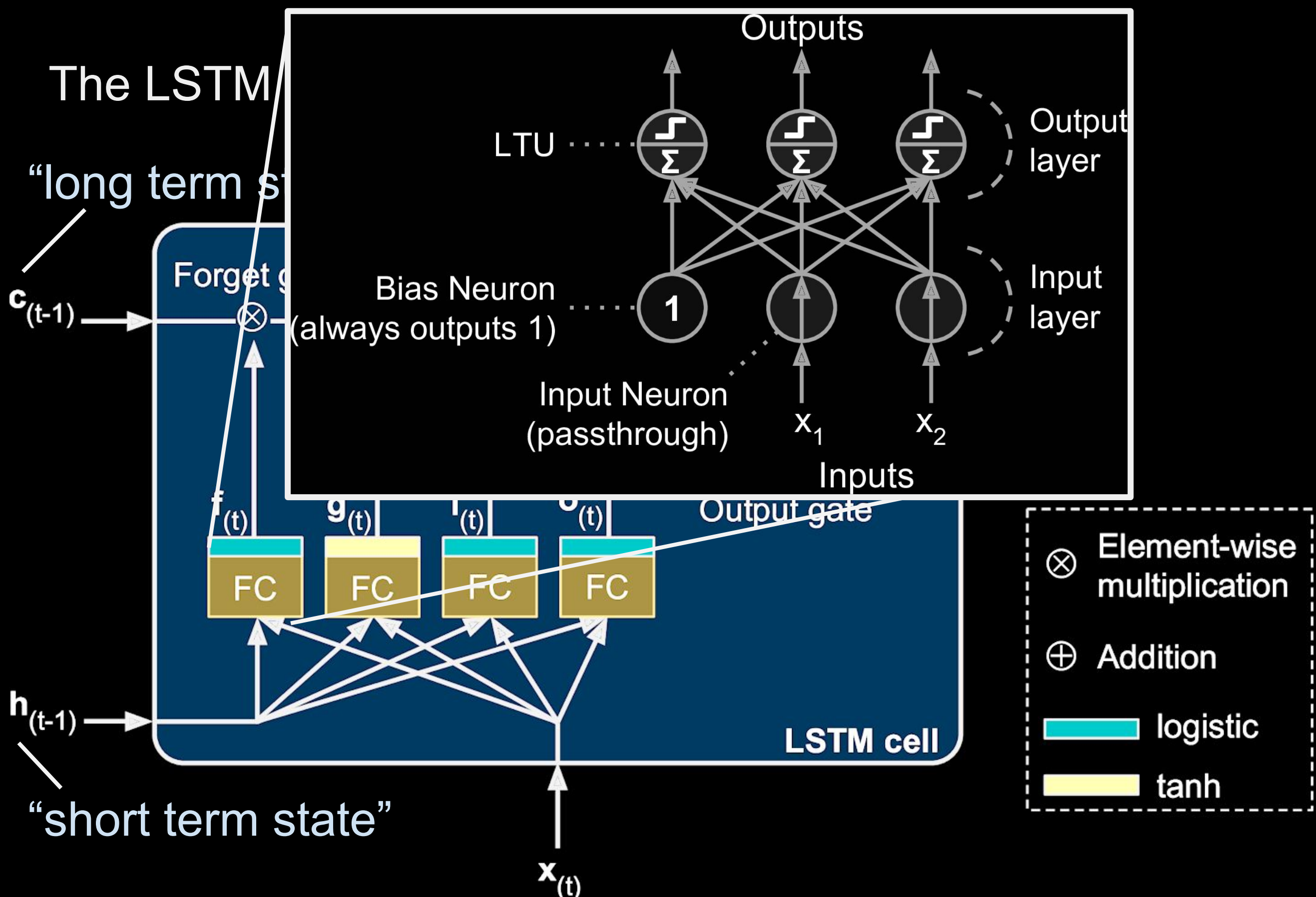


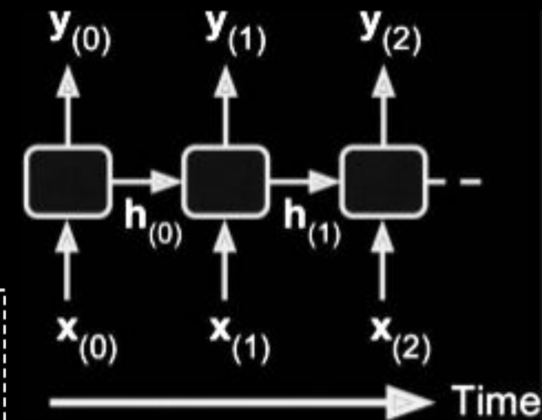
"unrolled" depiction

(Geron, 2017)

How to address exploding and vanishing gradients?

The LSTM



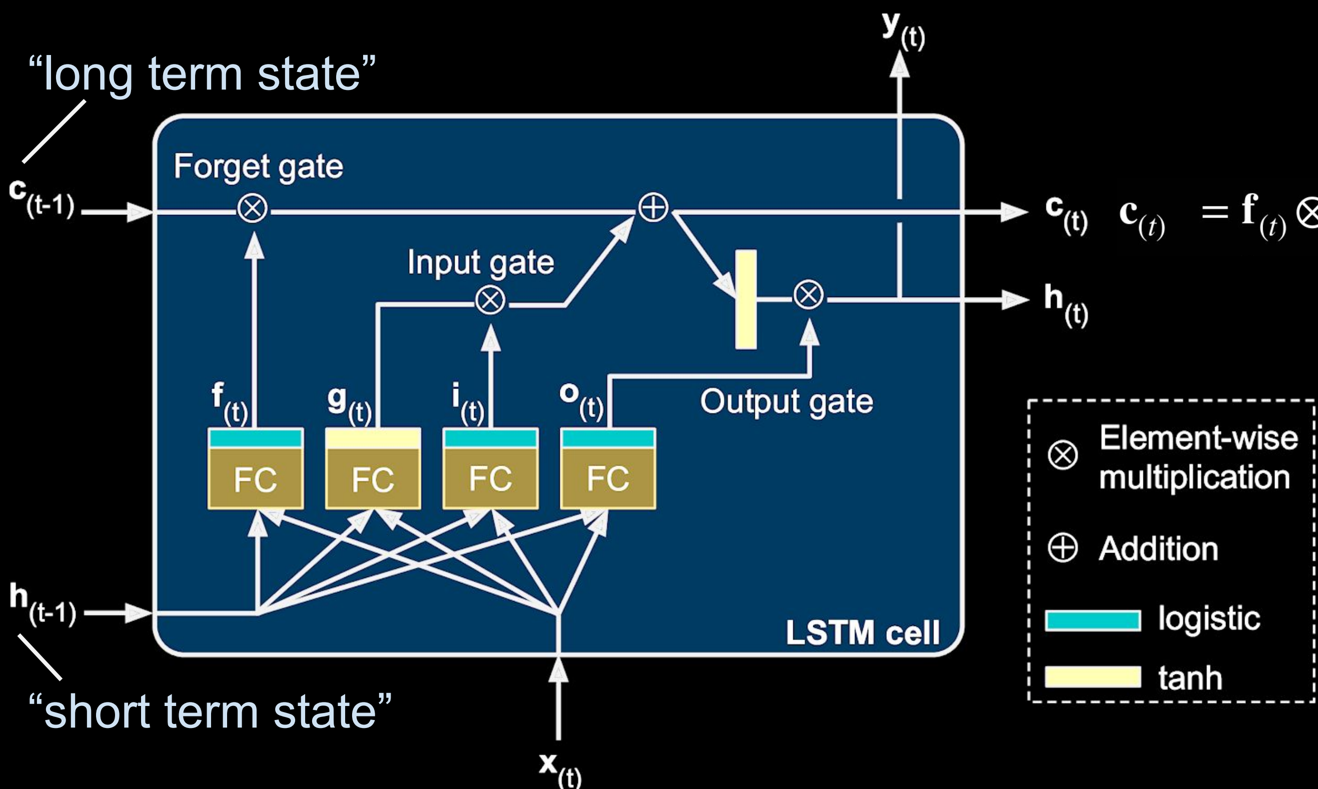


“unrolled” depiction

(Geron, 2017)

How to address exploding and vanishing gradients?

The LSTM Cell

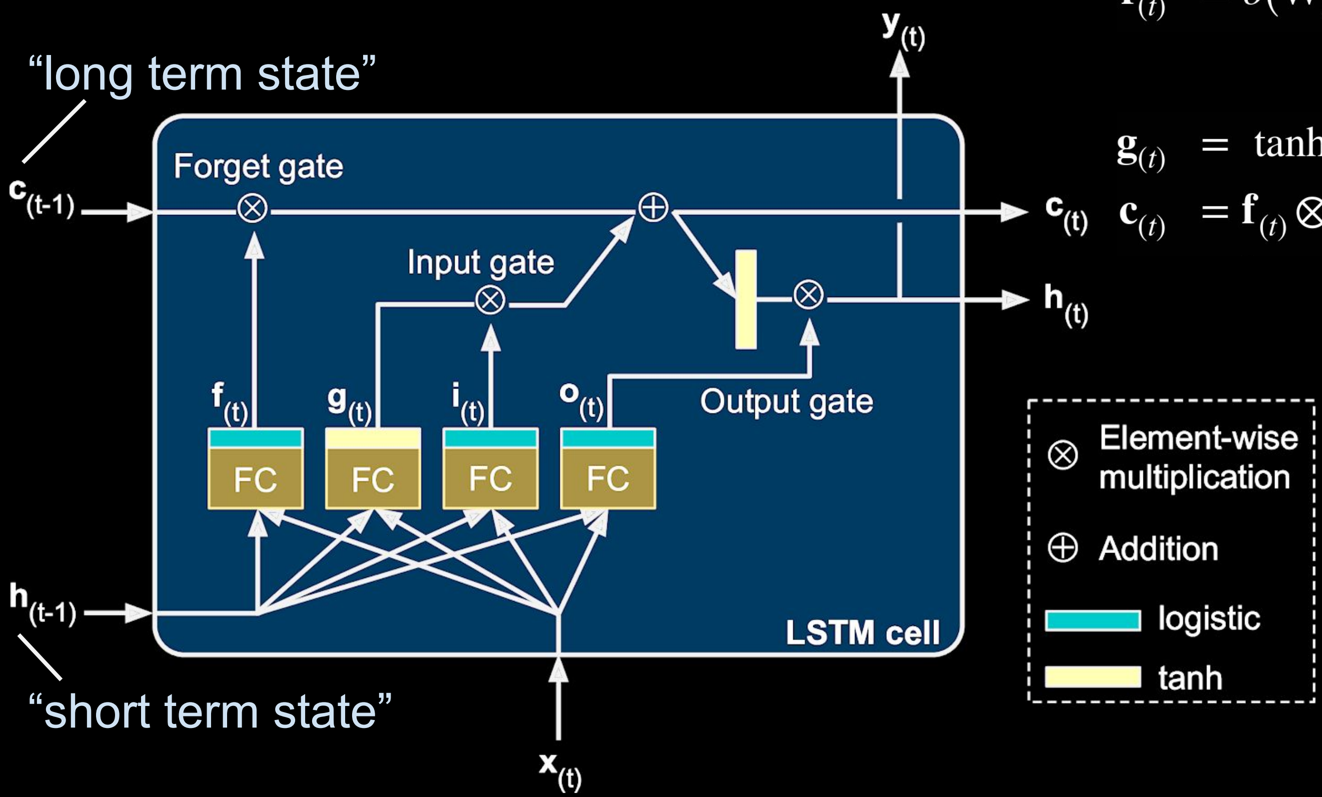


$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}$$

How to address exploding and vanishing gradients?

The LSTM Cell

"long term state"



"short term state"

$$\mathbf{i}_{(t)} = \sigma(\mathbf{W}_{xi}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i)$$

$$\mathbf{f}_{(t)} = \sigma(\mathbf{W}_{xf}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f)$$

$$\mathbf{g}_{(t)} = \tanh(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g)$$

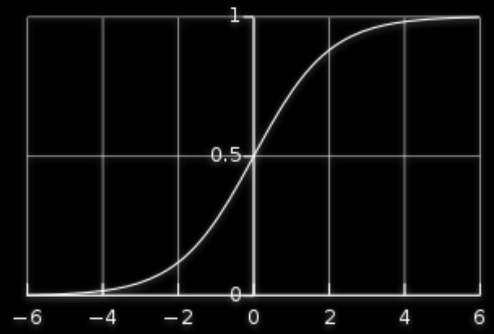
$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}$$

bias term

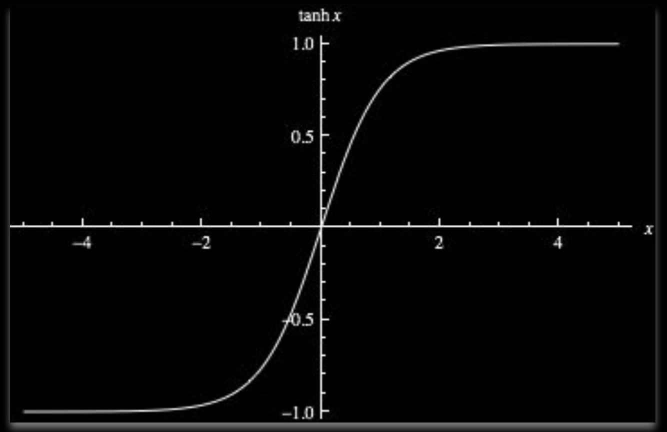
Common Activation Functions

$z = h_{(t)}W$

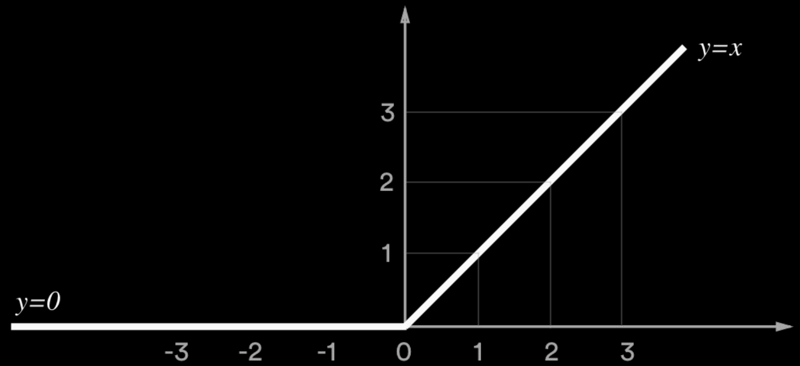
Logistic: $\sigma(z) = 1 / (1 + e^{-z})$



Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1)$



Rectified linear unit (ReLU): $ReLU(z) = \max(0, z)$

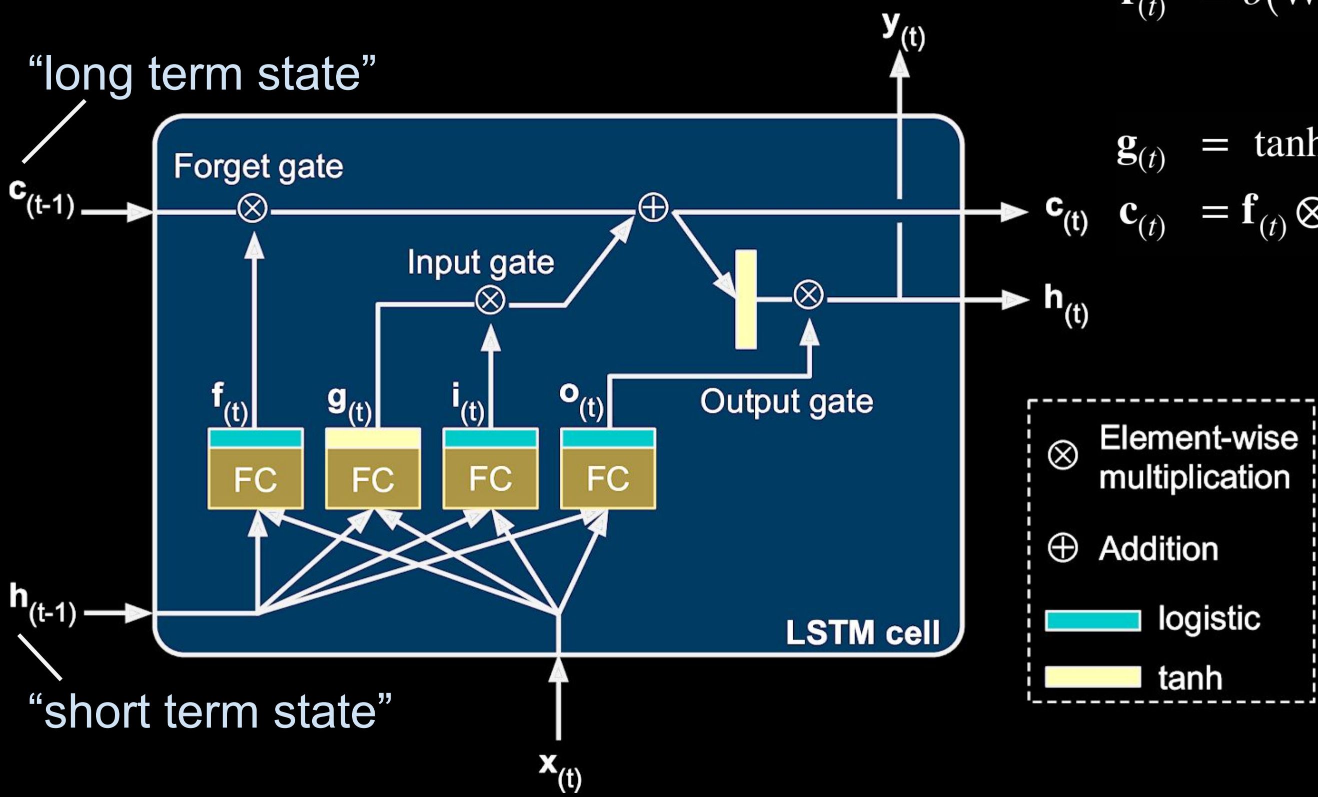


LSTM

The LSTM Cell

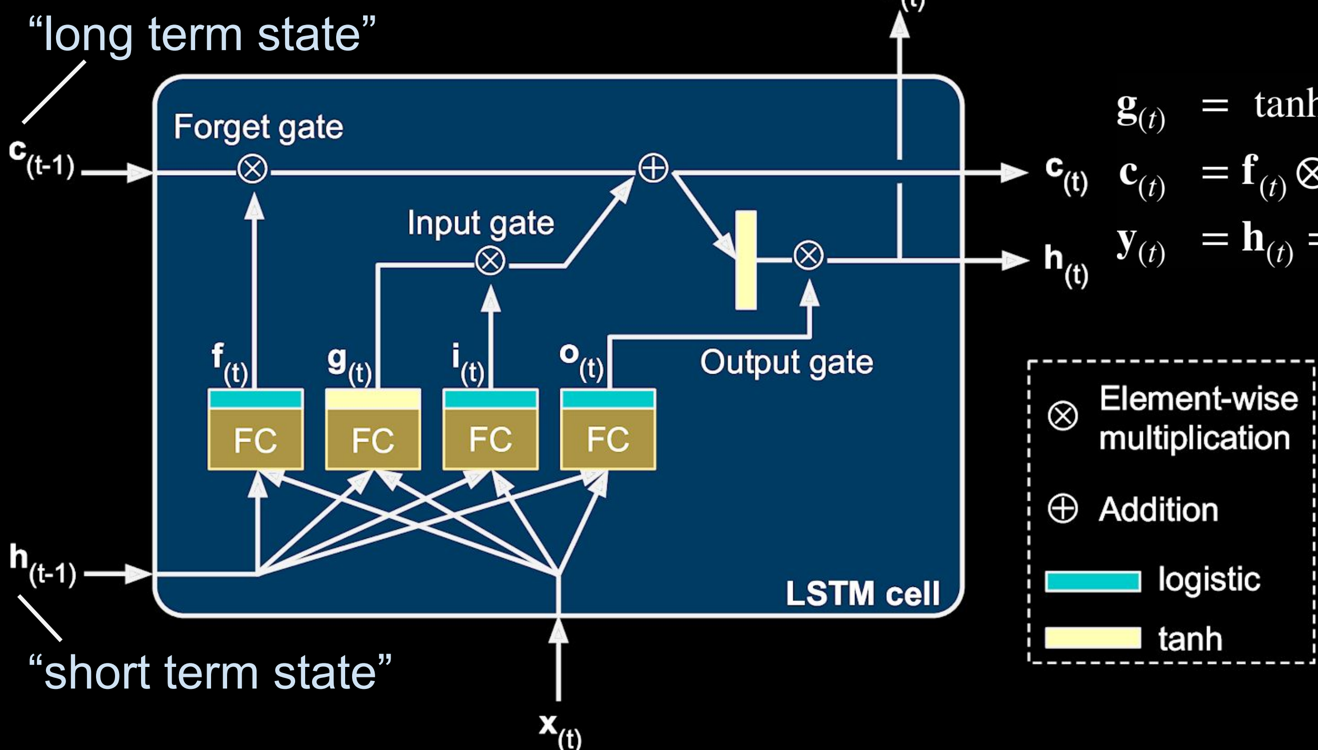
$$\mathbf{i}_{(t)} = \sigma(\mathbf{W}_{xi}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{i})$$
$$\mathbf{f}_{(t)} = \sigma(\mathbf{W}_{xf}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{f})$$

$$\mathbf{g}_{(t)} = \tanh(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{g})$$
$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}$$



LSTM

The LSTM Cell

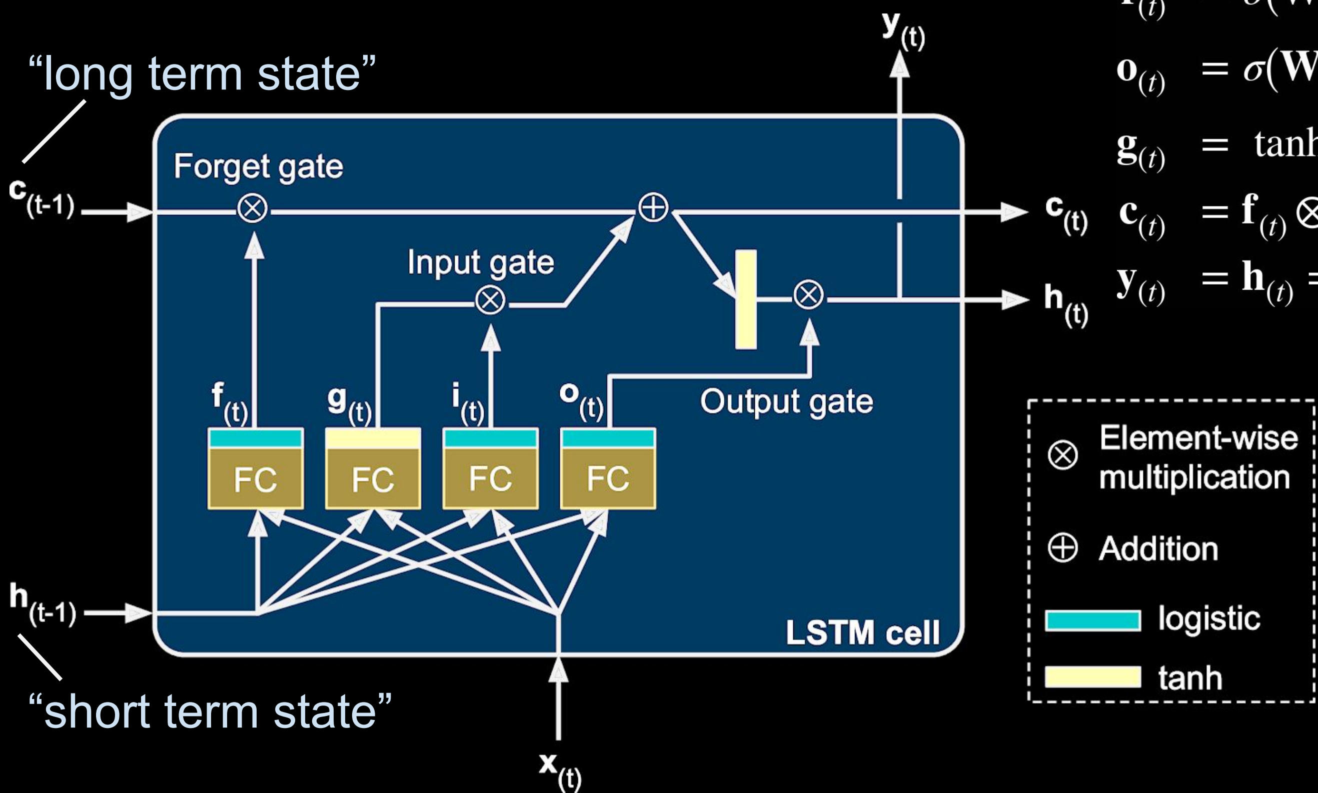


$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma(\mathbf{W}_{xi}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i) \\
\mathbf{f}_{(t)} &= \sigma(\mathbf{W}_{xf}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned}
$$

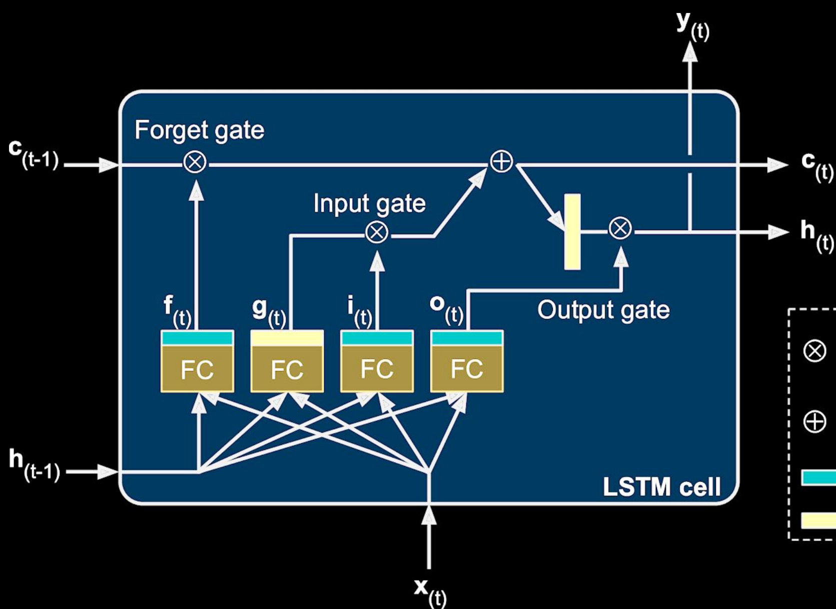
LSTM

The LSTM Cell



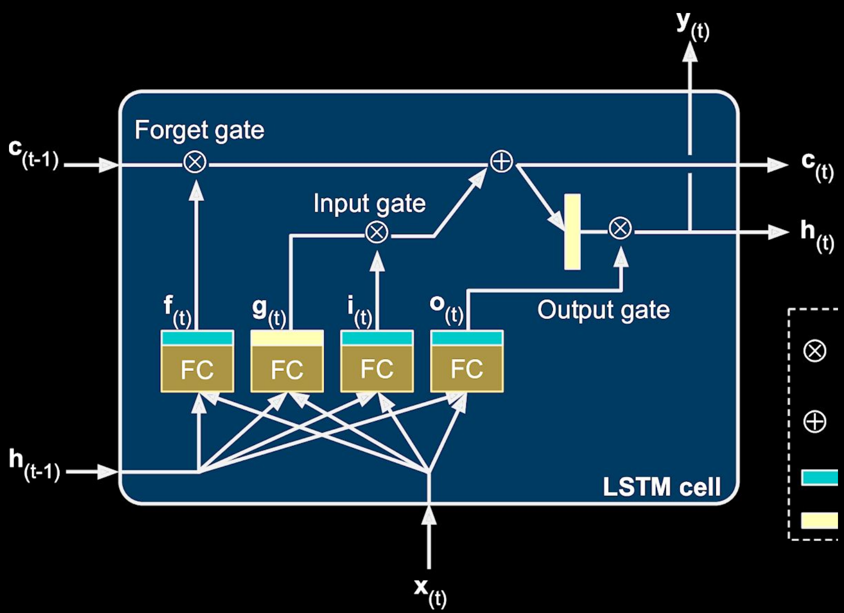
$$\begin{aligned}
\mathbf{i}_{(t)} &= \sigma(\mathbf{W}_{xi}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i) \\
\mathbf{f}_{(t)} &= \sigma(\mathbf{W}_{xf}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f) \\
\mathbf{o}_{(t)} &= \sigma(\mathbf{W}_{xo}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o) \\
\mathbf{g}_{(t)} &= \tanh(\mathbf{W}_{xg}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)})
\end{aligned}$$

Input to LSTM



?

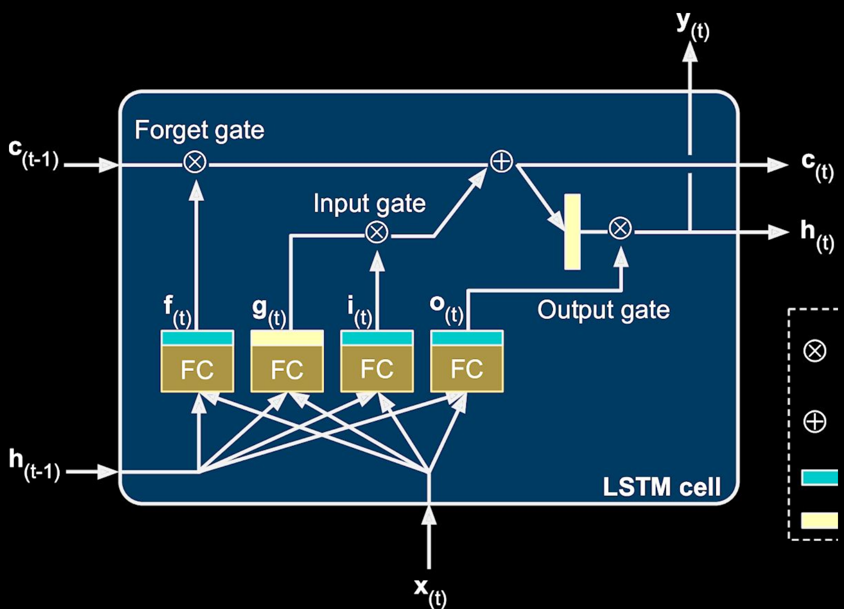
Input to LSTM



?

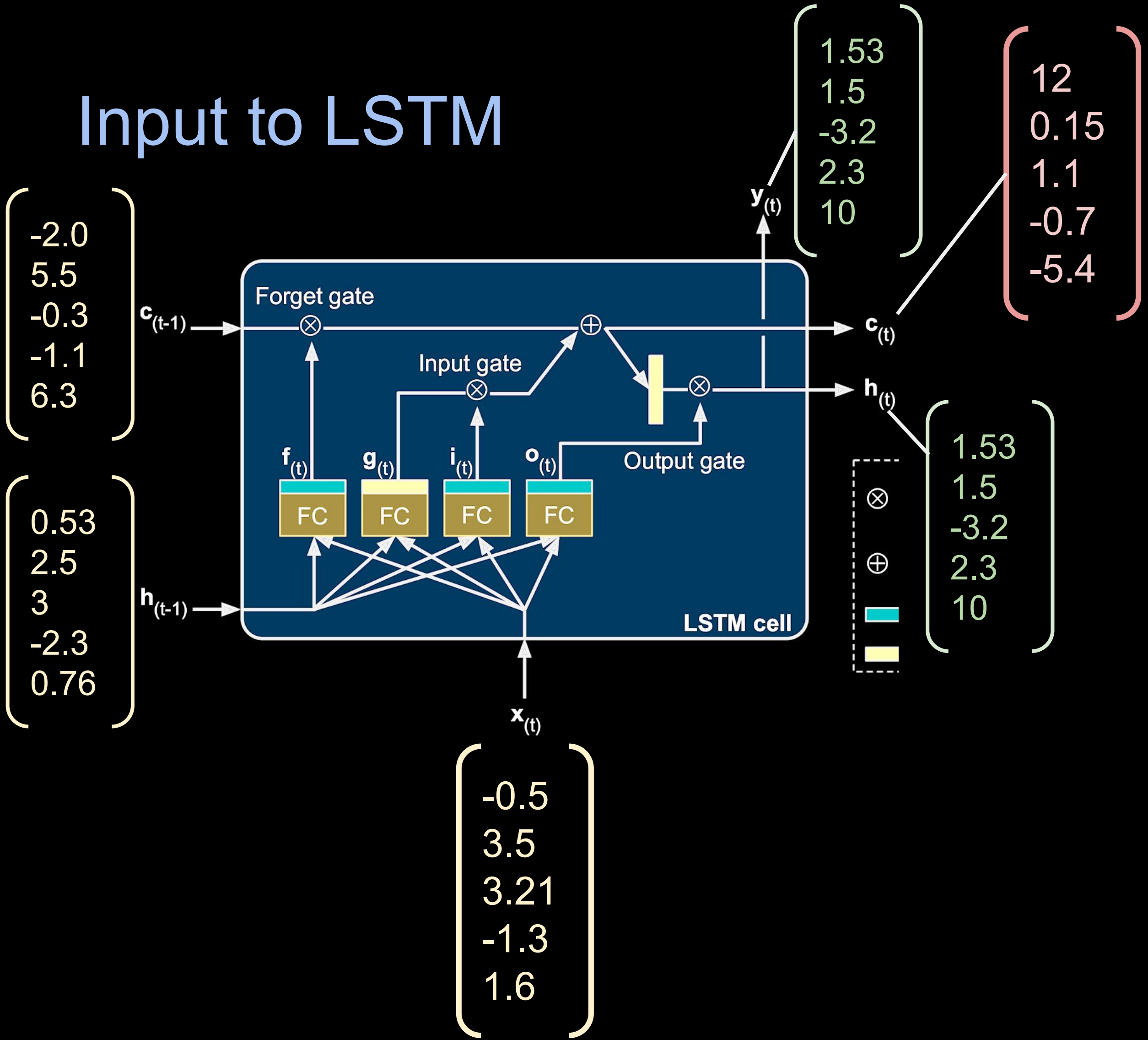
- One-hot encoding?
- Word Embedding

Input to LSTM

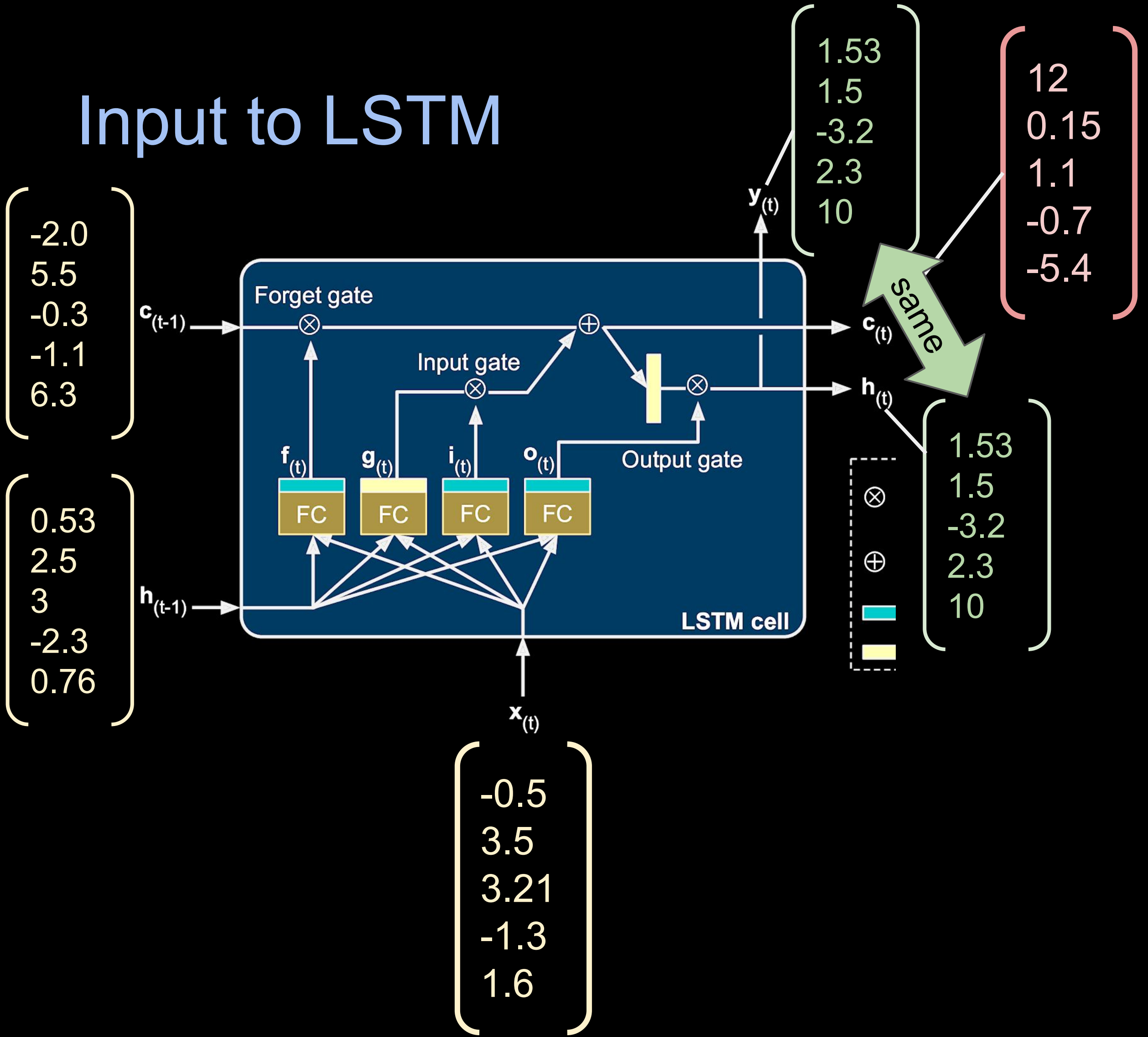


$$\begin{pmatrix} -0.5 \\ 3.5 \\ 3.21 \\ -1.3 \\ 1.6 \end{pmatrix}$$

Input to LSTM

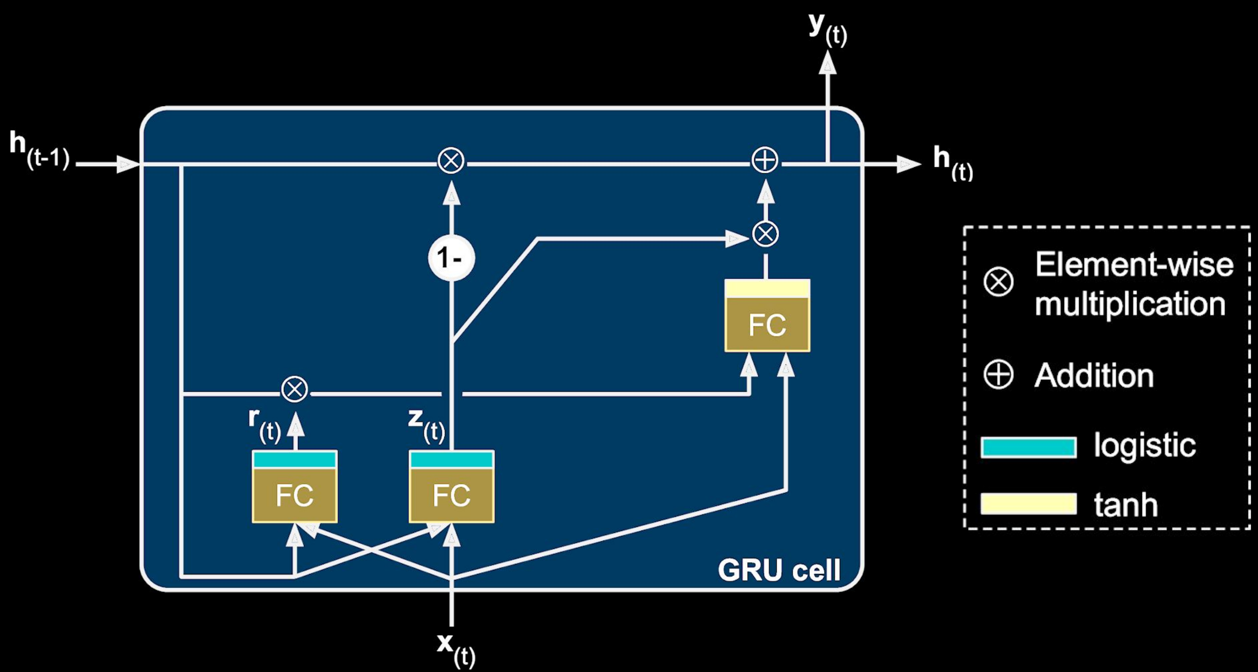


Input to LSTM



The GRU

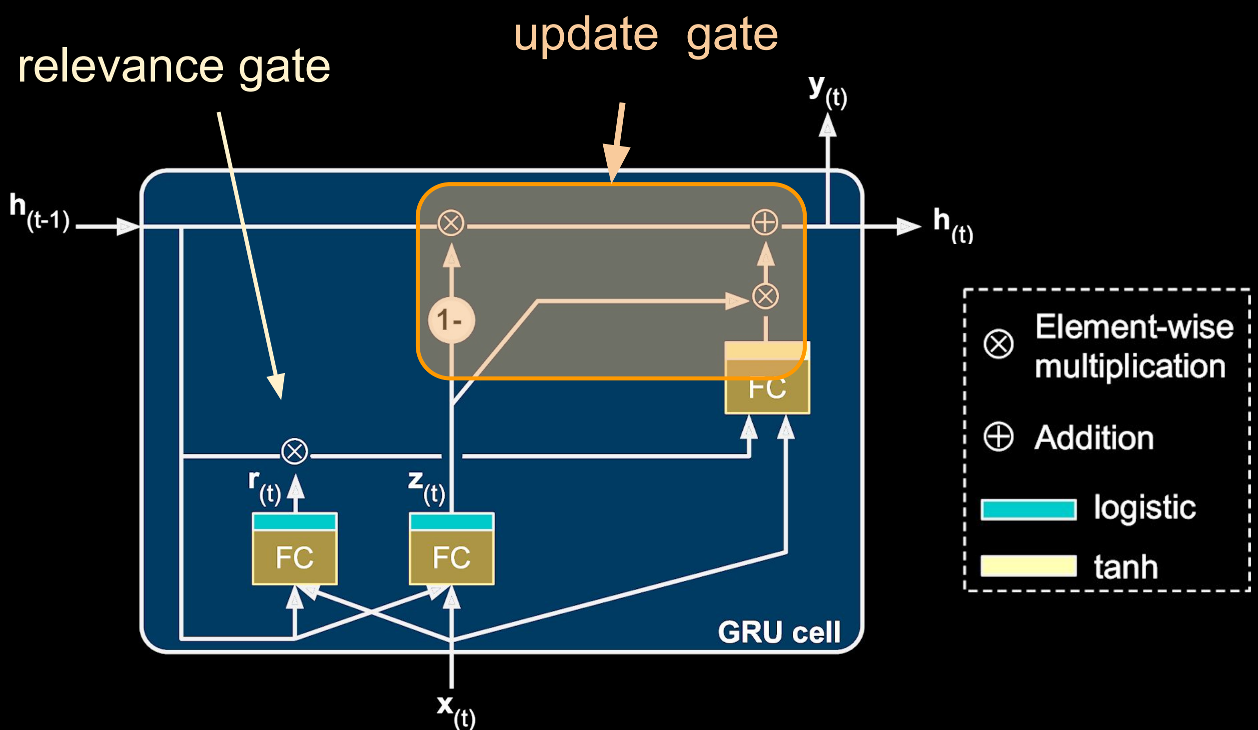
Gated Recurrent Unit



(Geron, 2017)

The GRU

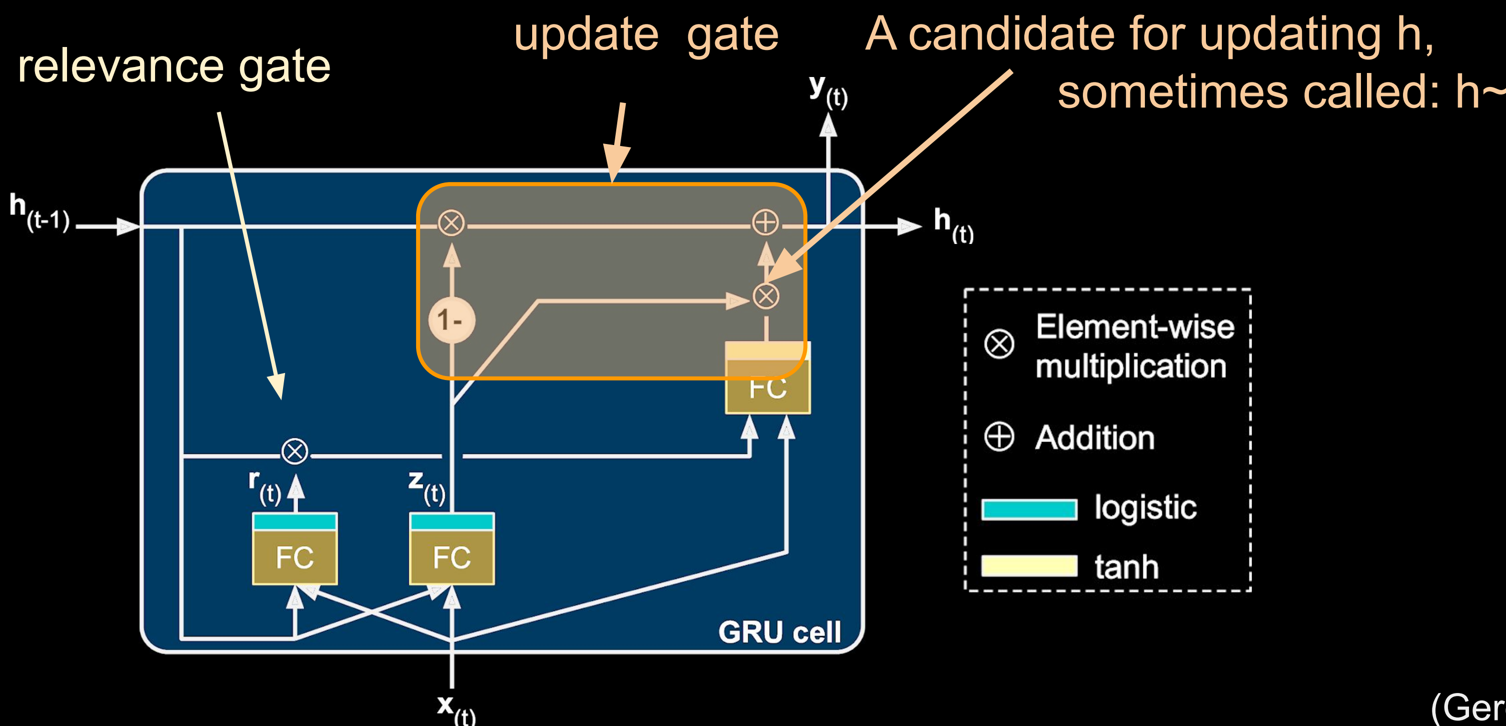
Gated Recurrent Unit



(Geron, 2017)

The GRU

Gated Recurrent Unit

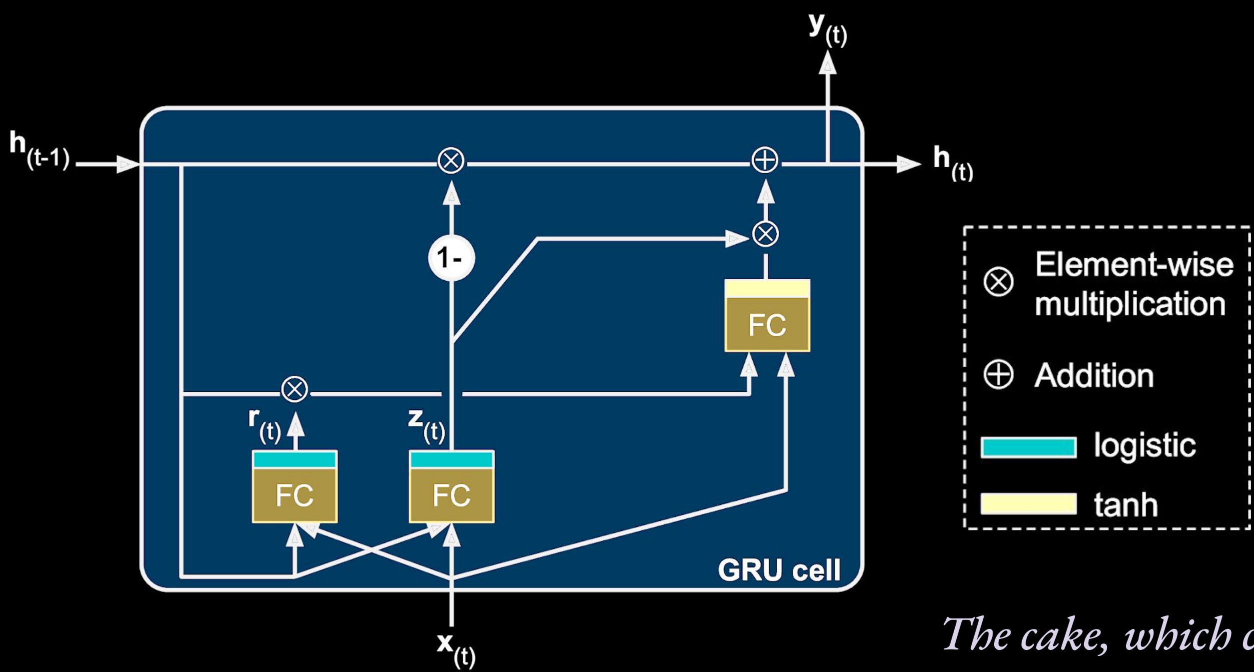


(Geron, 2017)

The GRU

Gated Recurrent Unit

$$\mathbf{z}_{(t)} = \sigma(\mathbf{W}_{xz}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_z)$$
$$\mathbf{r}_{(t)} = \sigma(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r)$$
$$\mathbf{g}_{(t)} = \tanh\,(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g)$$
$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}$$



The cake, which contained candles, was eaten.

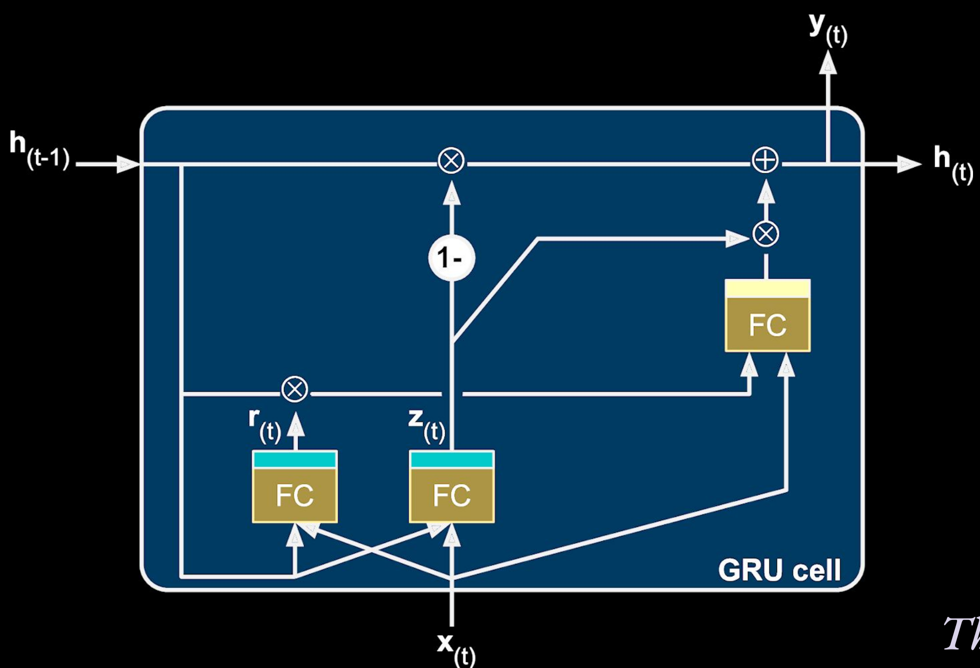
What about the gradient?

$$\mathbf{z}_{(t)} = \sigma(\mathbf{W}_{xz}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_z)$$

$$\mathbf{r}_{(t)} = \sigma(\mathbf{W}_{xr}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r)$$

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^T \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g\right)$$

$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}$$



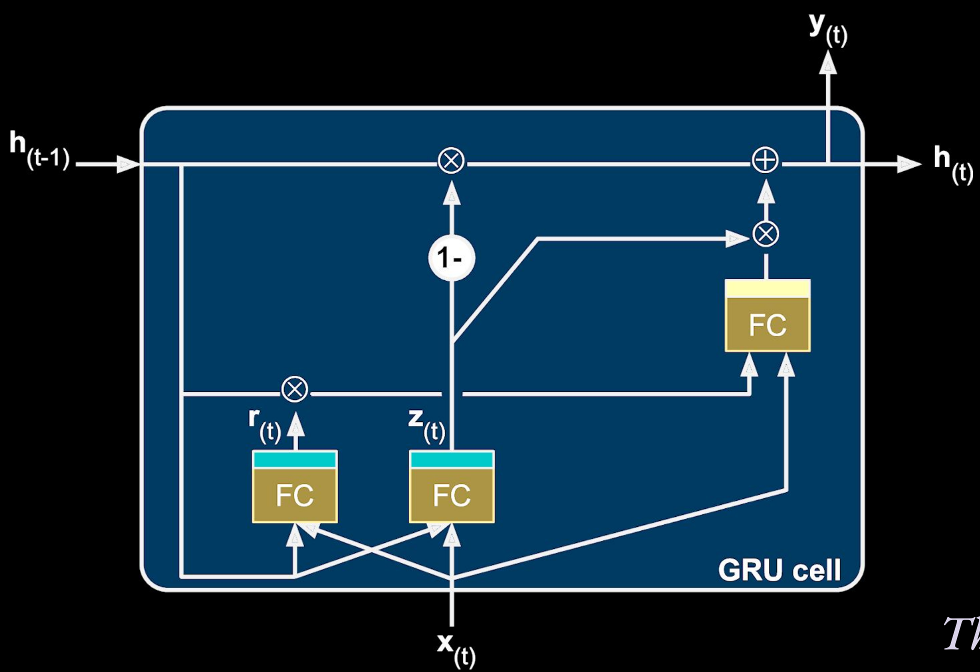
The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of h,

$$h_{(t)} \approx h_{(t-1)}$$

The cake, which contained candles, was eaten.

What about the gradient?

$$\mathbf{z}_{(t)} = \sigma(\mathbf{W}_{xz}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_z)$$
$$\mathbf{r}_{(t)} = \sigma(\mathbf{W}_{xr}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r)$$
$$\mathbf{g}_{(t)} = \tanh\,(\mathbf{W}_{xg}{}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^T \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g)$$
$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}$$



The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of h,

$$h_{(t)} \approx h_{(t-1)}$$

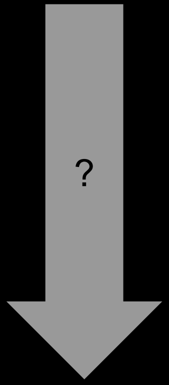
This tends to keep the gradient from vanishing since the same values will be present through multiple times in backpropagation through time. (The same idea applies to LSTMs but is easier to see here).

The cake, which contained candles, was eaten.

How to train an LSTM-style RNN

```
RNN_cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
             #where did this come from?
```

Logistic Regression Likelihood: $L(\beta_0, \beta_1, \ldots, \beta_k | X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i}$



Final Cost Function: $J^{(t)} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{|V|} y_{i,j}^{(t)} log\ \hat{y}_{i,j}^{(t)}$ -- ”cross entropy error”

How to train an LSTM-style RNN

```
RNN_cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
            #where did this come from?
```

Logistic Regression Likelihood: $$L(\beta_0, \beta_1, ..., \beta_k | X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i}$$

Log Likelihood: $$\ell(\beta) = \sum_{i=1}^{N} y_i log\, p(x_i) + (1 - y_i) log\, (1 - p(x_i))$$

Final Cost Function: $$J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{|V|} y_{i,j}^{(t)} log\, \hat{y}_{i,j}^{(t)}$$ -- "cross entropy error"

How to train an LSTM-style RNN

```
RNN_cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
            #where did this come from?
```

Logistic Regression Likelihood: $L(\beta_0, \beta_1, ..., \beta_k | X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i}$

Log Likelihood: $\ell(\beta) = \sum_{i=1}^{N} y_i log\ p(x_i) + (1 - y_i) log\ (1 - p(x_i))$

Log Loss: $J(\beta) = -\frac{1}{N}\sum_{i=1}^{N} y_i log\ p(x_i) + (1 - y_i) log\ (1 - p)(x_i))$

Final Cost Function: $J^{(t)} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{|V|} y_{i,j}^{(t)} log\ \hat{y}_{i,j}^{(t)}$ -- ”cross entropy error”

How to train an LSTM-style RNN

```
RNN_cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
            #where did this come from?
```

Logistic Regression Likelihood: $$L(\beta_0, \beta_1, ..., \beta_k | X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i}$$

Log Likelihood: $$\ell(\beta) = \sum_{i=1}^{N} y_i log\, p(x_i) + (1 - y_i) log\, (1 - p(x_i))$$

Log Loss: $$J(\beta) = -\frac{1}{N}\sum_{i=1}^{N} y_i log\, p(x_i) + (1 - y_i) log\, (1 - p)(x_i))$$

Cross-Entropy Cost: $$J = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{|V|} y_i log\, p(x_{i,j})$$ (a “multiclass” log loss)

Final Cost Function: $$J^{(t)} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{|V|} y_{i,j}^{(t)} log\, \hat{y}_{i,j}^{(t)}$$ -- ”cross entropy error”

How to train an LSTM-style RNN

```
RNN_cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
            #where did this come from?
```

To Optimize Betas (all weights within LSTM cells):

Stochastic Gradient Descent (SGD)

-- optimize over one sample each iteration

Mini-Batch SDG:

--optimize over *b* samples each iteration

Final Cost Function: $J^{(t)} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{|V|} y_{i,j}^{(t)} log\ \hat{y}_{i,j}^{(t)}$ -- ”cross entropy error”

RNN-Based Language Models

Take-Aways

- Simple RNNs are powerful models but they are difficult to train:
  - Just two functions $h_{(t)}$ and $y_{(t)}$ where $h_{(t)}$ is a combination of $h_{(t-1)}$ and $x_{(t)}$.
  - Exploding and vanishing gradients make training difficult to converge.
- LSTM and GRU cells solve
  - Hidden states pass from one time-step to the next, allow for long-distance dependencies.
  - Gates are used to keep hidden states from changing rapidly (and thus keeps gradients under control).
  - To train: mini-batch stochastic gradient descent over cross-entropy cost