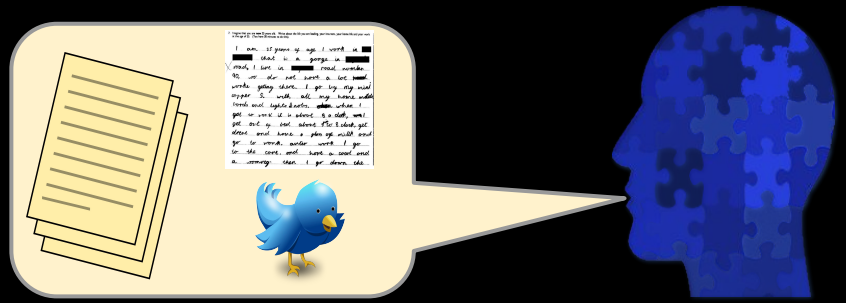
Recurrent Neural Networks for Language Modeling

CSE354 - Spring 2021
Natural Language Processing

Tasks



- Language Modeling:

  Generate next word, sentence

  ≈ capture hidden

  representation of sentences.

- Word, Document Classification

  (named entity tagging; sentiment

  analysis using sequence, etc…)

how?

- Recurrent Neural Network and Sequence Models

Language Modeling

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

:probability of a next word given history

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

Language Modeling

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

:probability of a next word given history

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

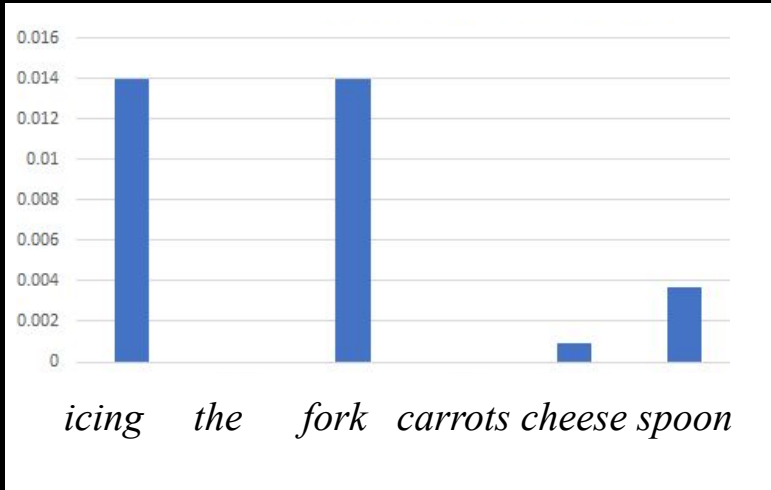
History
(He, at, the, cake, with, the)

Trained Language Model

What is the next word in the sequence?

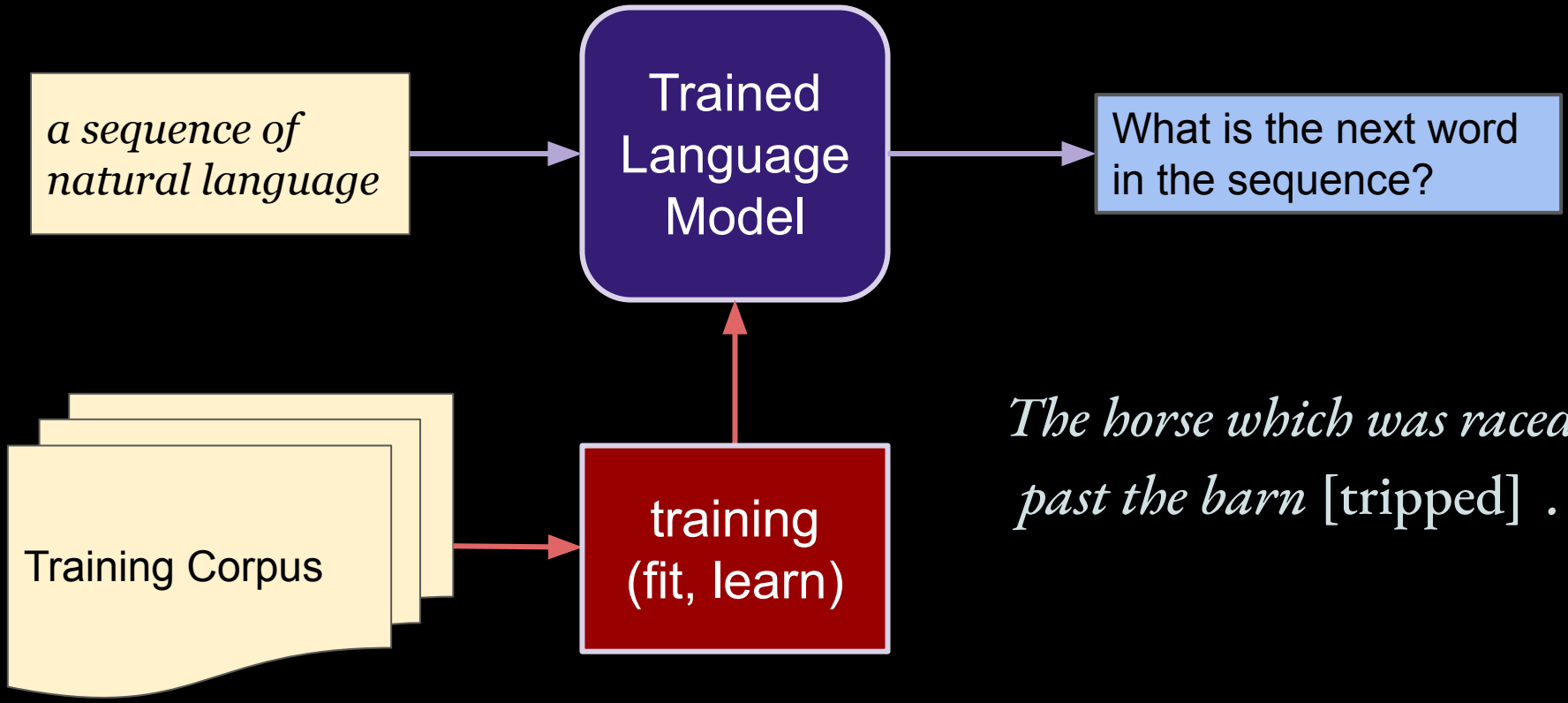
Training Corpus

training (fit, learn)



Language Modeling

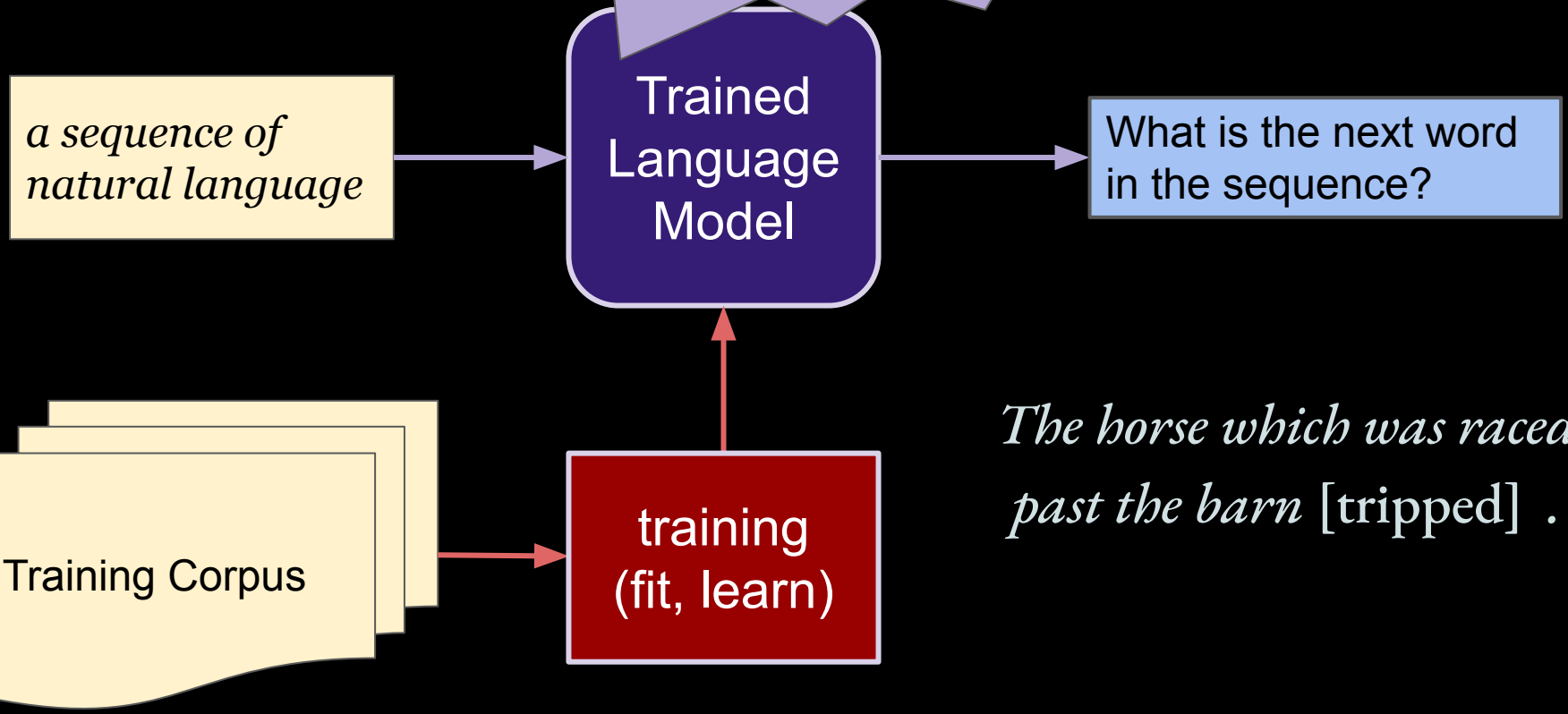
Building a model (or system / API) that can answer the following:



Language Modeling

To fully capture natural language, models get very complex!

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



The horse which was raced past the barn [tripped] .

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

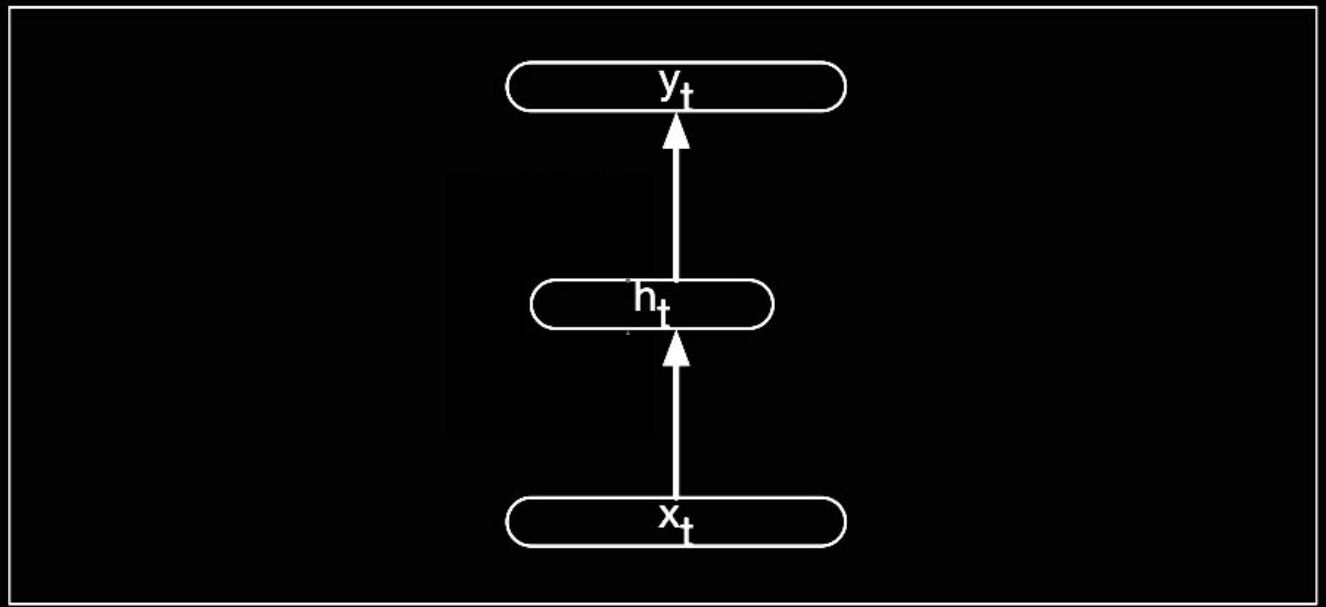


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

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

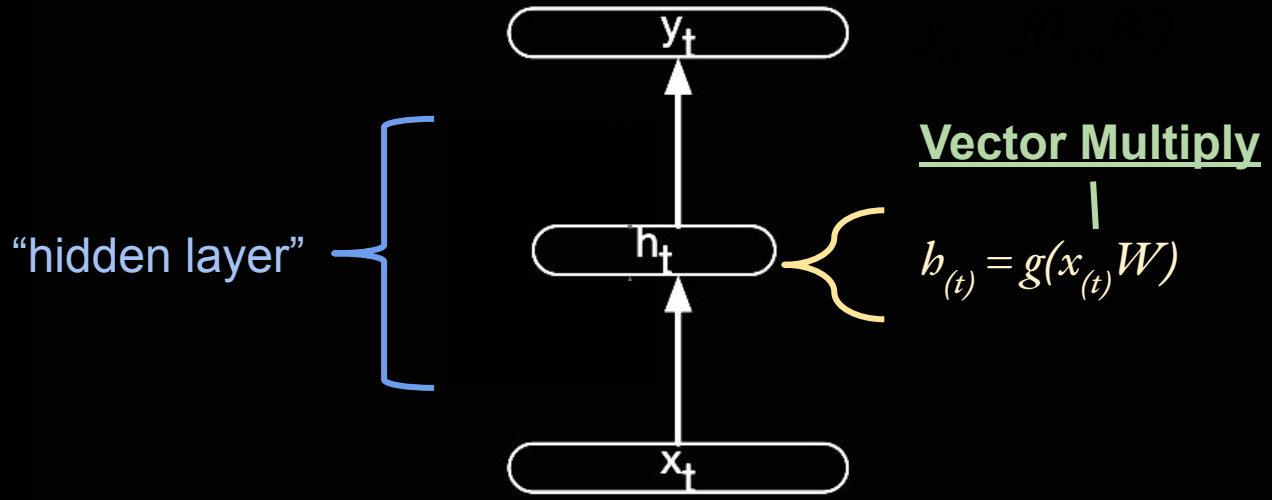


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

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

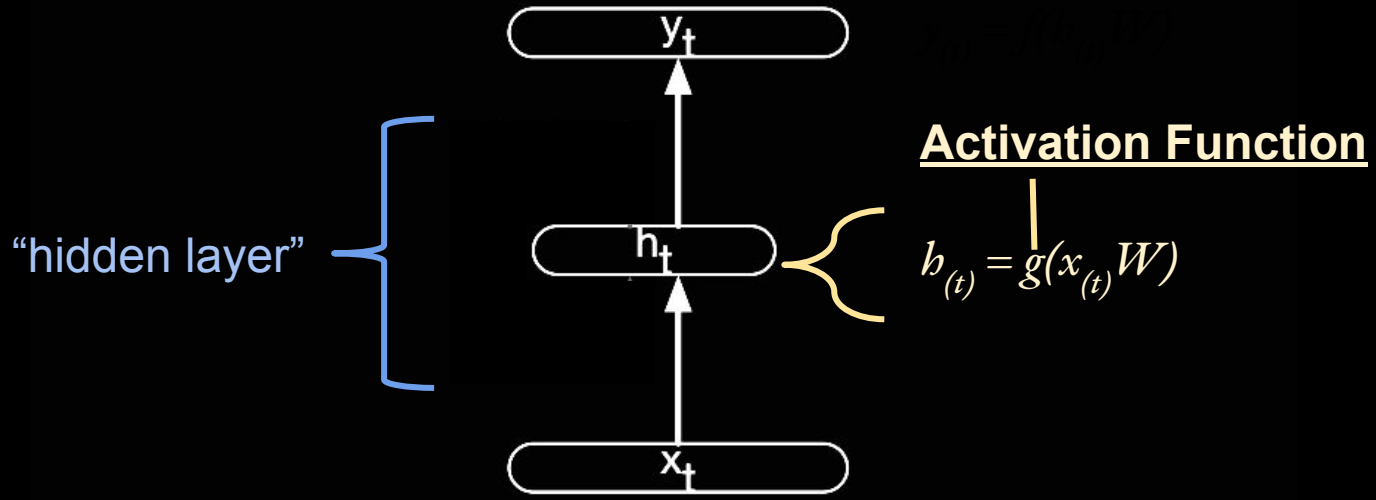


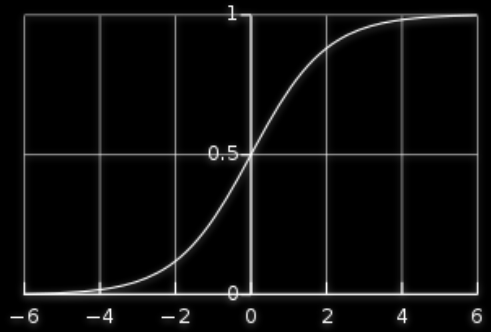
Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Common Activation Functions

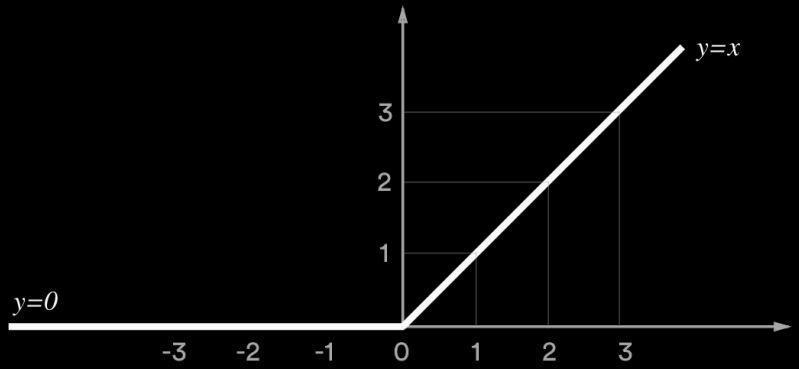
$z = h_{(t)}W$

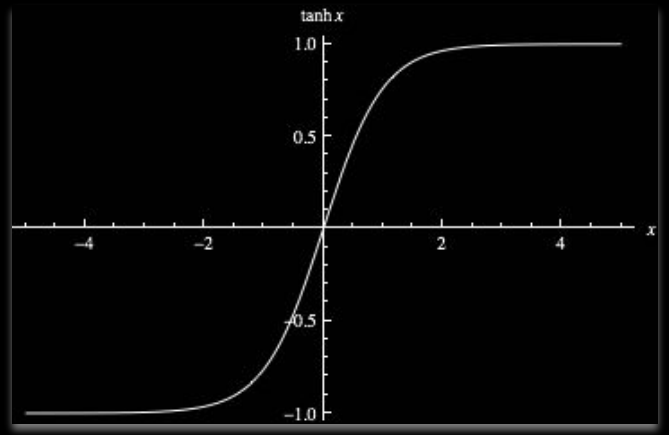
Logistic: $\sigma(z) = 1 / (1 + e^{-z})$



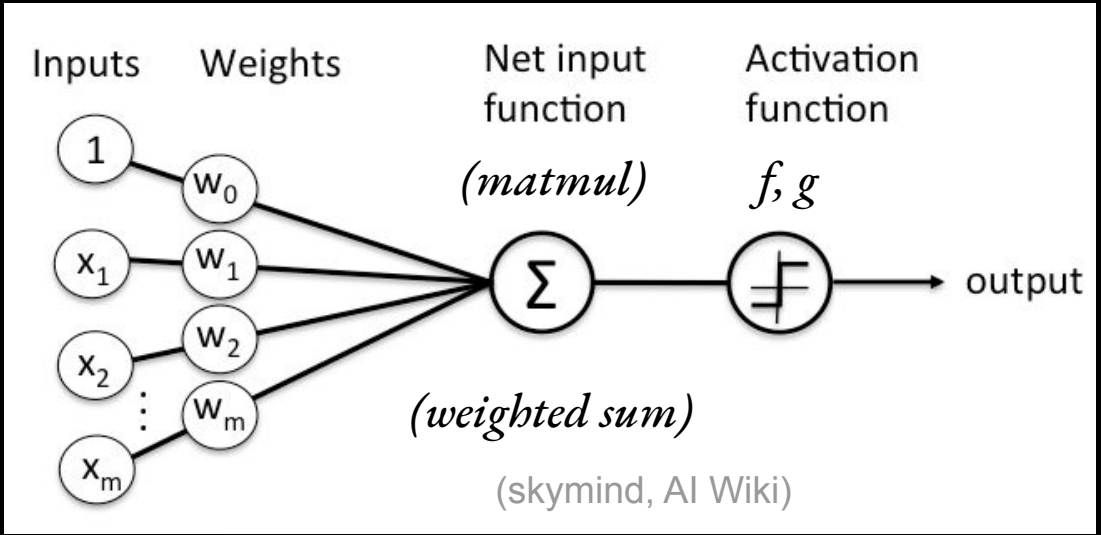
Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1)$

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





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



Activation Function

$$h_{(t)} = g(x_{(t)}W)$$

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)

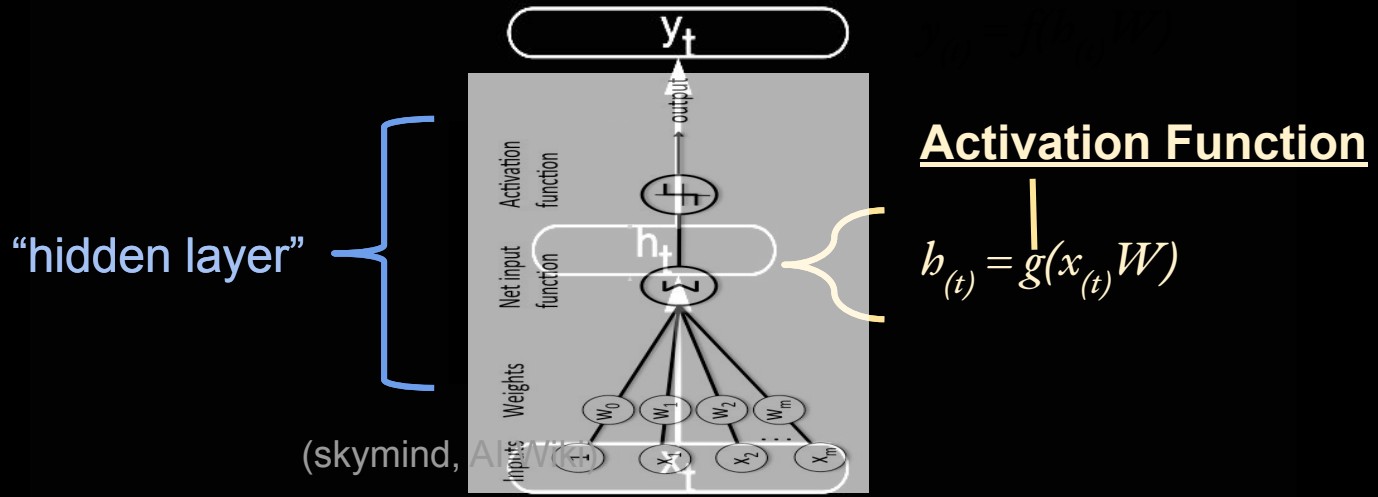


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

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

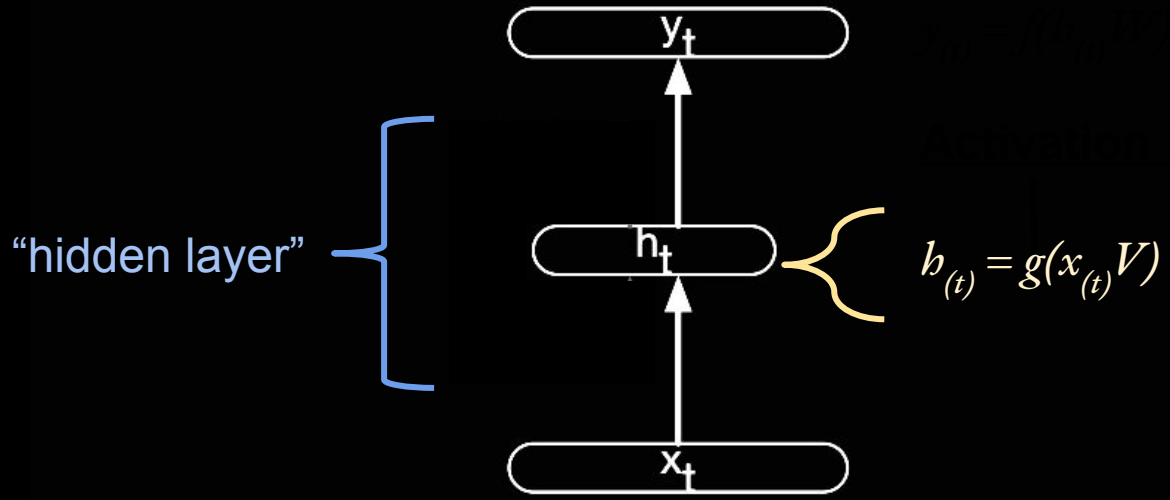


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

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

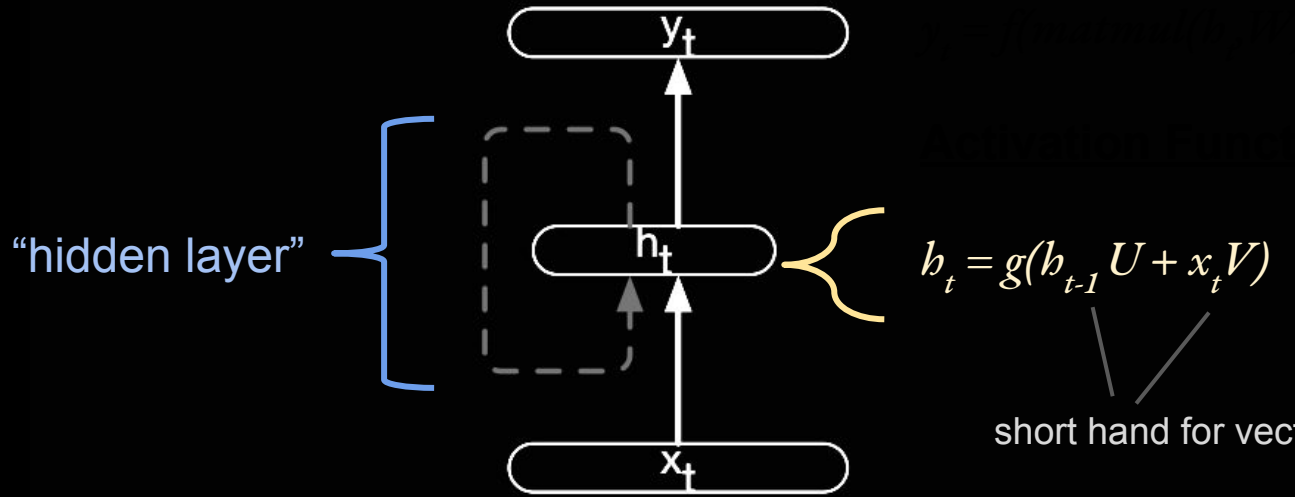


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

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

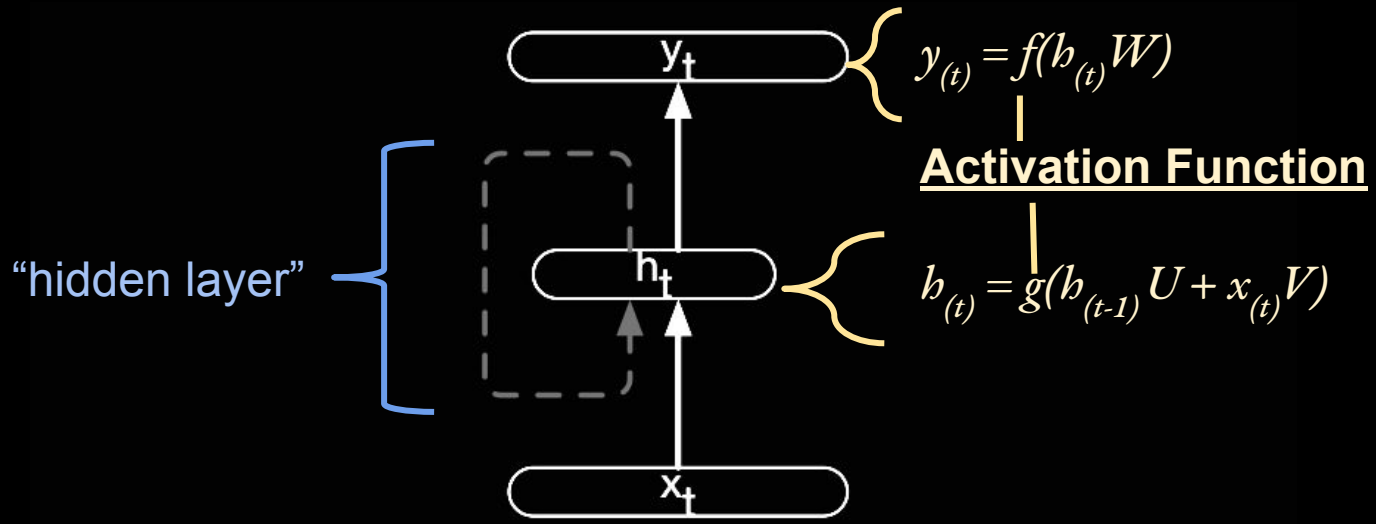


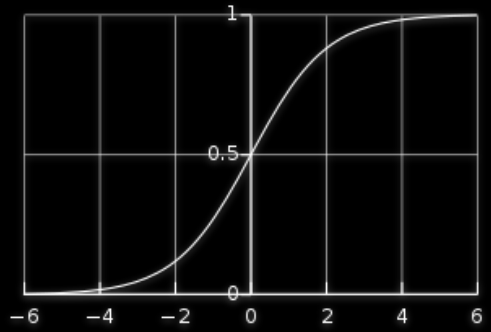
Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Common Activation Functions

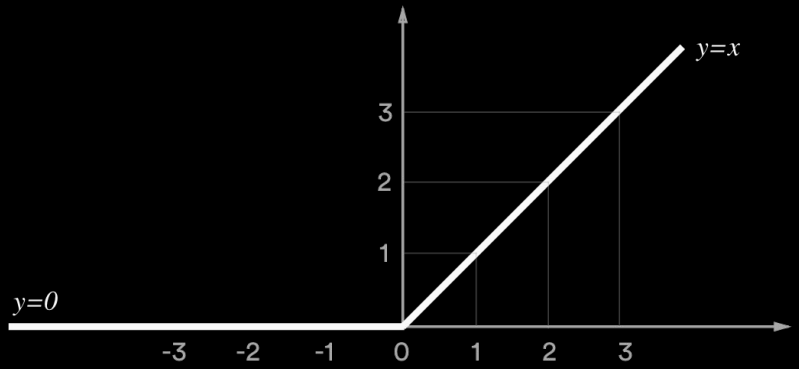
$z = h_{(t)}W$

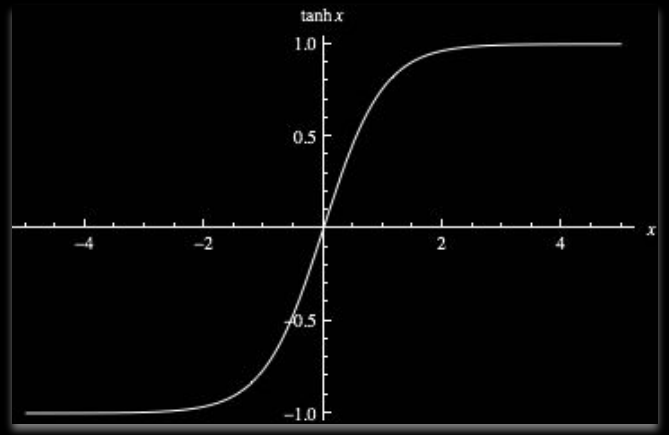
Logistic: $\sigma(z) = 1 / (1 + e^{-z})$



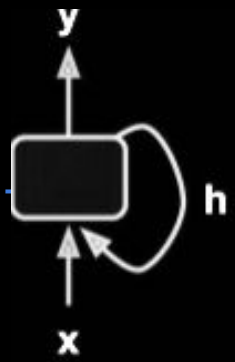
Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1)$

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





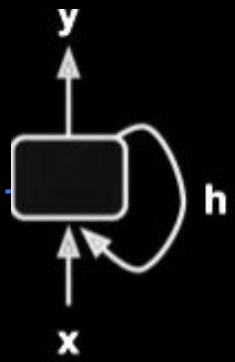
Example: Forward Pass



(Geron, 2017)

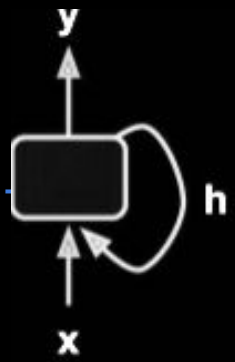
```
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = g(U h(i-1) + W x(i)) #update hidden state
    y(i) = f(V h(i)) #update output
```

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

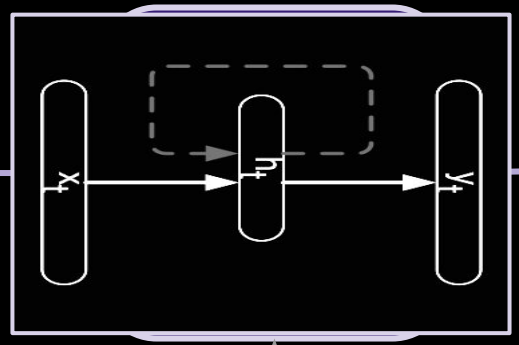
Task: Estimate $P(w_n | w_1, w_2, \ldots, w_{n-1})$

:probability of a next word given history

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

History

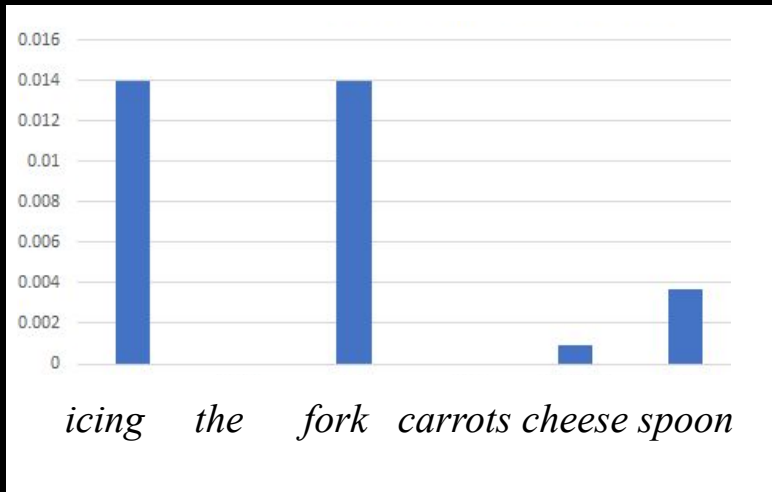
(He, at, the, cake, with, the)



What is the next word in the sequence?

Training Corpus

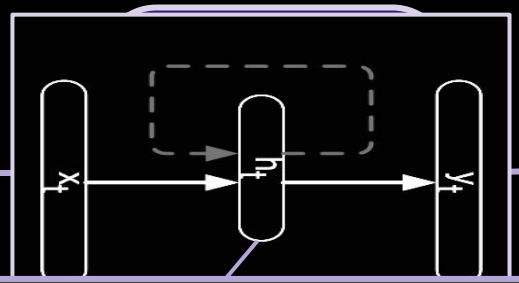
training (fit, learn)



Language Modeling

Task: Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$
:probability of a next word given history
P(fork | He ate the cake with the) = ?

Last word
(the)

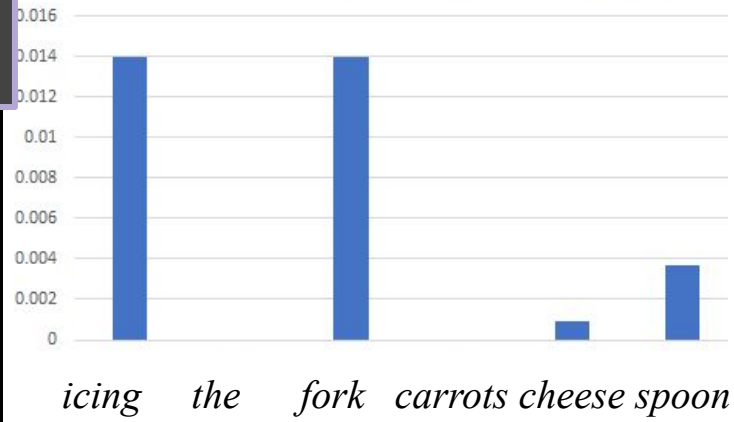


What is the next word in the sequence?

h_t: a vector that we hope "stores" relevant history from previous inputs:
He, at, the, cake, with,

Training Corpus

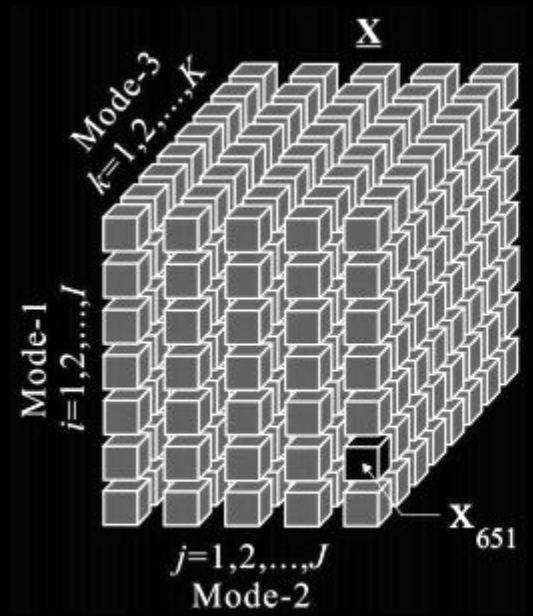
training
(fit, learn)



Tensors in PyTorch

Need a workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors



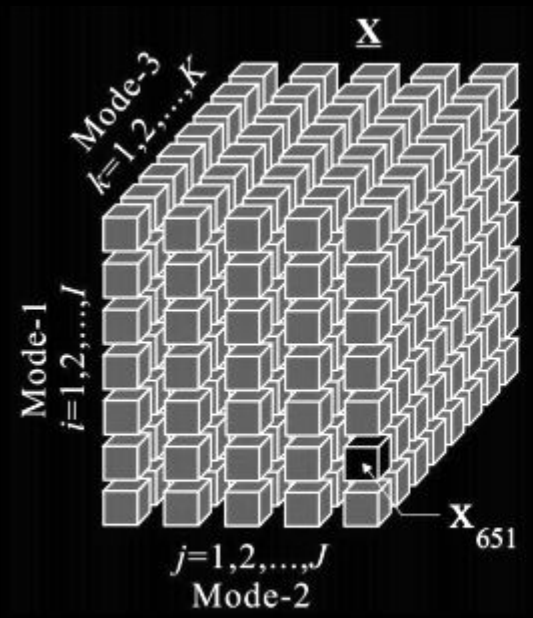
(i.stack.imgur.com)

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)

PyTorch

A workflow system catered to numerical computation.

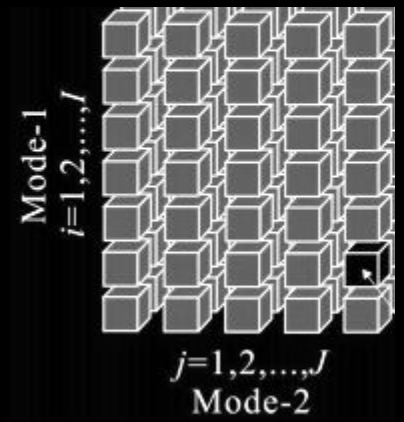
Basic idea: defines a graph of operations on tensors

A multi-dimensional matrix

A 2-d tensor is just a matrix.

1-d: vector

0-d: a constant / scalar



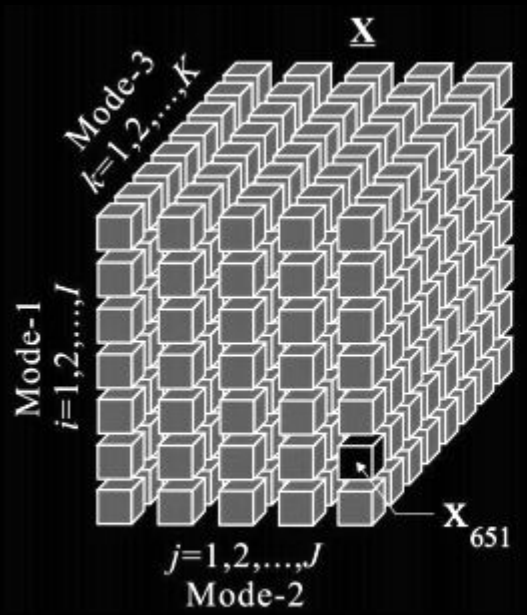
(i.stack.imgur.com)

PyTorch

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)

PyTorch

A workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors

Why?

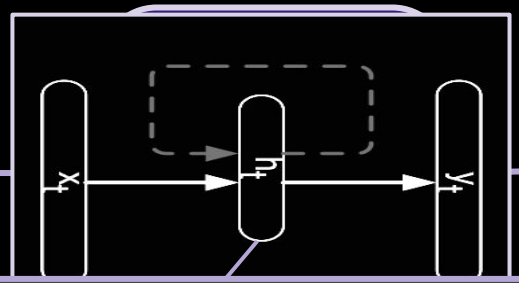
Efficient, high-level built-in **linear algebra** and **machine learning optimization** *operations* (i.e. transformations).

enables complex models, like deep learning

Language Modeling

Task: Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$
:probability of a next word given history
P(fork | He ate the cake with the) = ?

Last word
(the)

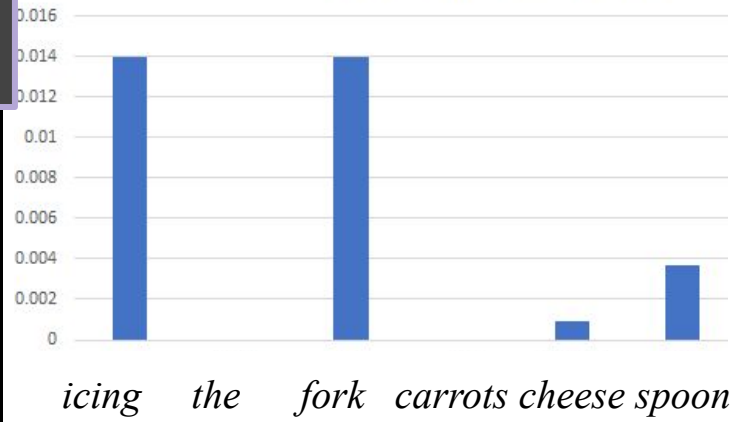


What is the next word in the sequence?

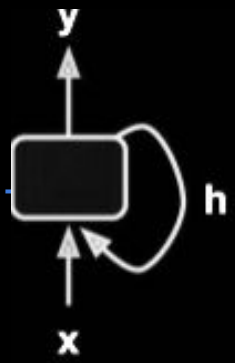
h_t: a vector that we hope "stores" relevant history from previous inputs:
He, at, the, cake, with,

Training Corpus

training
(fit, learn)

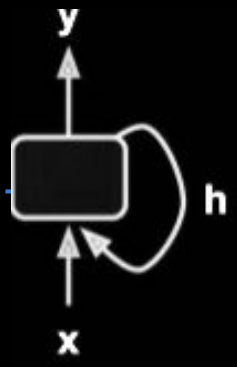


Example: RNN

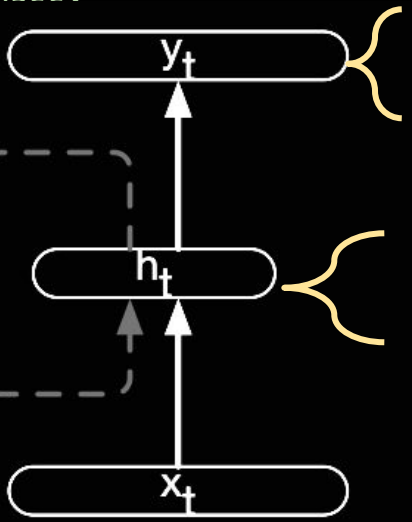



```
def forward(self, X):
    #Basic RNN Forward Pass:
    h_(0) = 0
    for i in range(1, len(x)):
        h_(i) = torch.tanh(torch.matmul(U,h_(i-1))+ torch.matmul(W,x_(i))) #update
    hidden state
        y_(i) = nn.log_softmax(torch.matmul(V, h_(i))) #update output


    ...

loss_func = nn.NLLLoss() #normalized log likelihood loss
            #torch.mean(-torch.sum(y*y_pred))
```

Example: RNN




```
def forward(self, X):
    #Basic RNN Forward Pass:
```



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

Activation Function

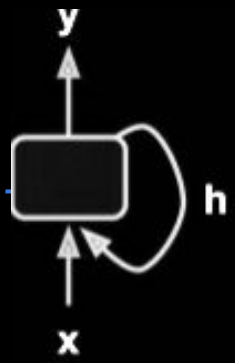
$h_{(t)} = g(h_{(t-1)}U + x_{(t)}V)$

"hidden layer"

```
ul(W,x_(i))) #update
e output
```

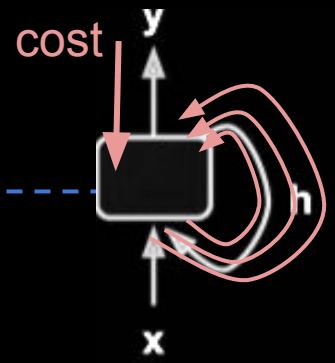
```
loss_func = nn.NLLLoss() #negative log likelihood loss
            #torch.mean(-torch.sum(y*y_pred))
```

Example: RNN



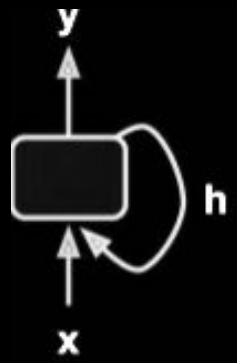

```
def forward(self, X):
    #Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.matmul(U,h(i-1))+ torch.matmul(W,x(i))) #update
    hidden state
        y(i) = nn.log_softmax(torch.matmul(V, h(i))) #update output


    ...

loss_func = nn.NLLLoss() #negative log likelihood loss
            #torch.mean(-torch.sum(y*y_pred))
```

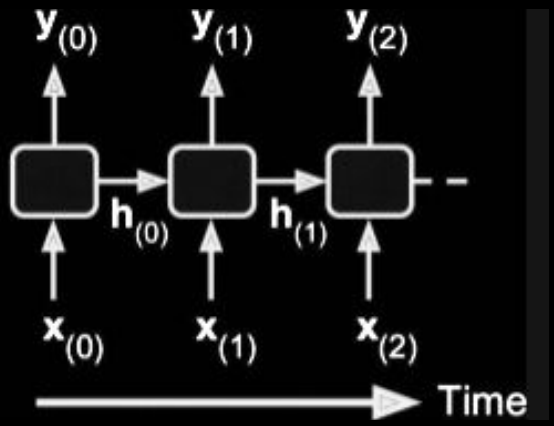
Back Propagation

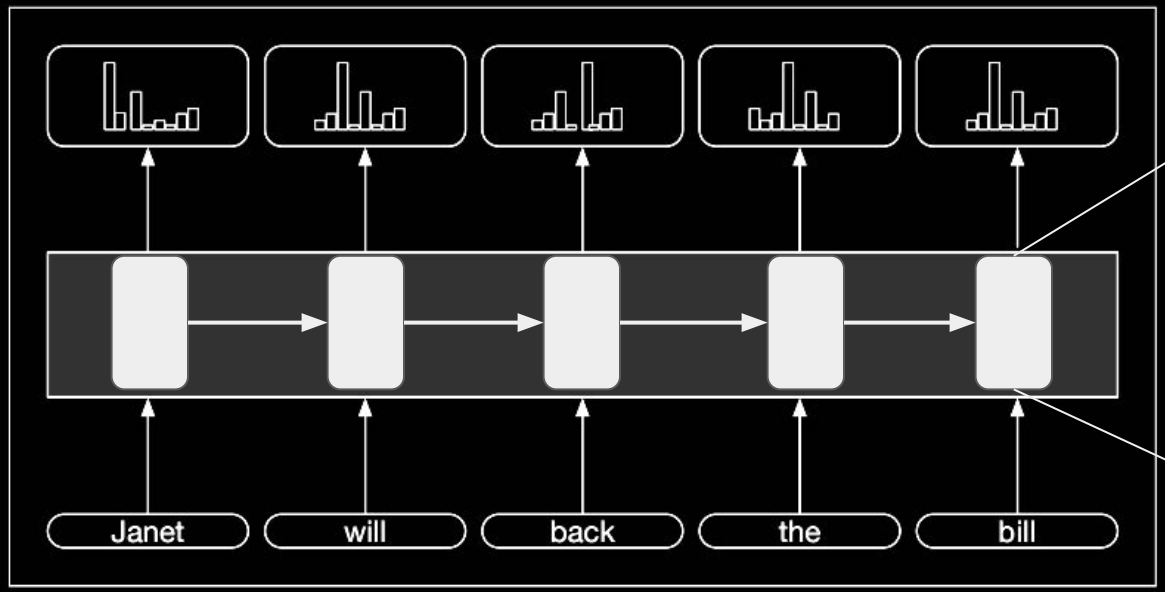



```
def forward(self, X):
    #Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.matmul(U,h(i-1))+ torch.matmul(W,x(i))) #update
    hidden state
        y(i) = nn.log_softmax(torch.matmul(V, h(i))) #update output


    ...

loss_func = nn.NLLLoss() #negative log likelihood loss
            #torch.mean(-torch.sum(y*y_pred))
```

Solution: Unrolling







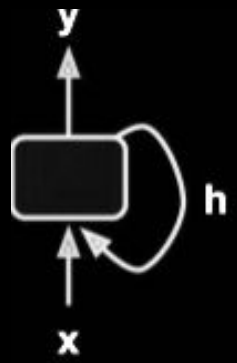
$$y_{(t)} = f(h_{(t)}W)$$

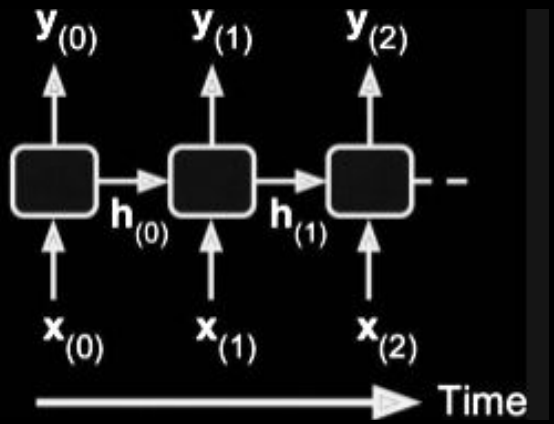
Activation Function

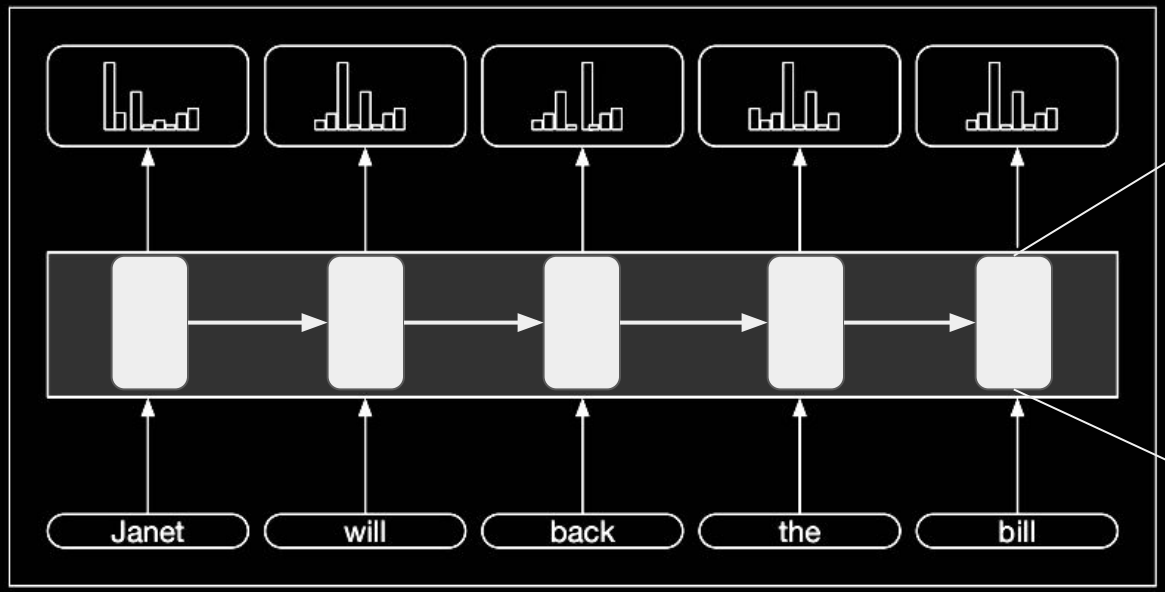
$$h_{(t)} = g(h_{(t-1)}U + x_{(t)}V)$$

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







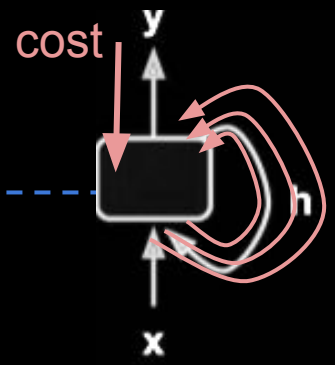
$$y_{("bill")} = f(h_{("bill")}W)$$

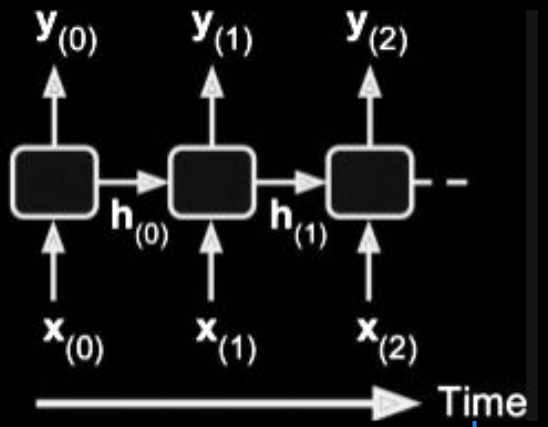
Activation Function

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

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.

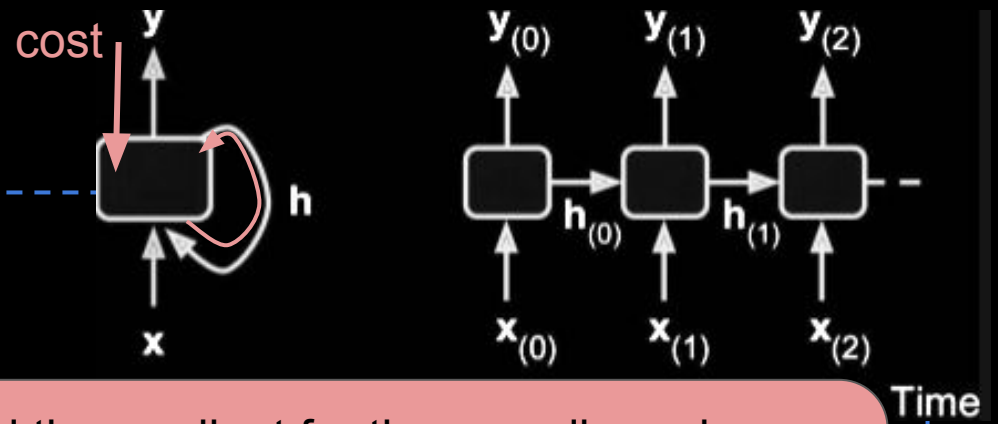
Back Propagation






```
def forward(self, X):
    #Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.matmul(U,h(i-1))+ torch.matmul(W,x(i))) #update
    hidden state
        y(i) = nn.log_softmax(torch.matmul(V, h(i))) #update output


    ...

loss_func = nn.NLLLoss() #negative log likelihood loss
            #torch.mean(-torch.sum(y*y_pred))
```

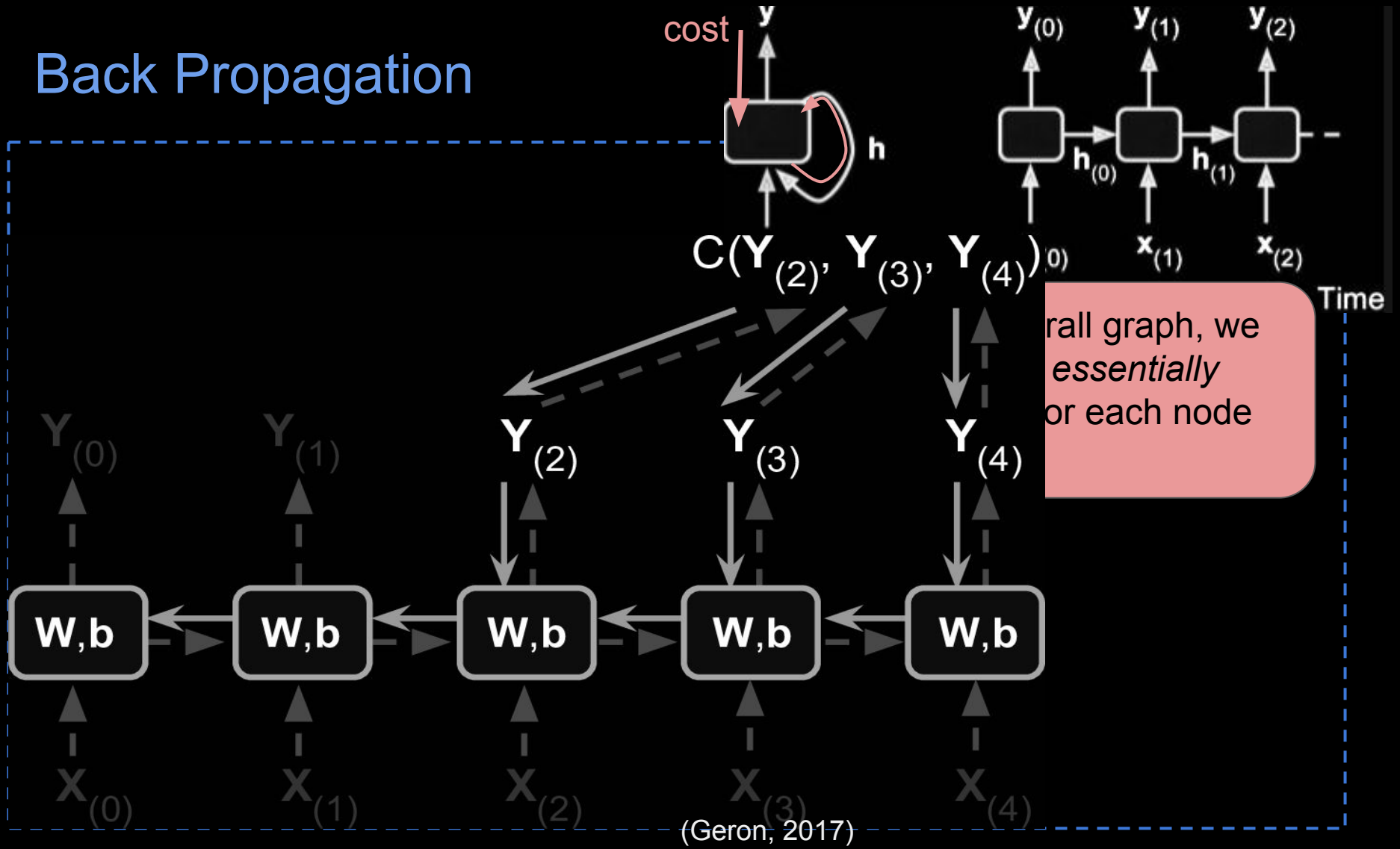
Back Propagation




```
def forward(self, X):
    #Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.
    hidden state
        y(i) = nn.log_softmax(to

    ...

loss_func = nn.NLLLoss() #negative log likelihood loss
            #torch.mean(-torch.sum(y*y_pred))
```

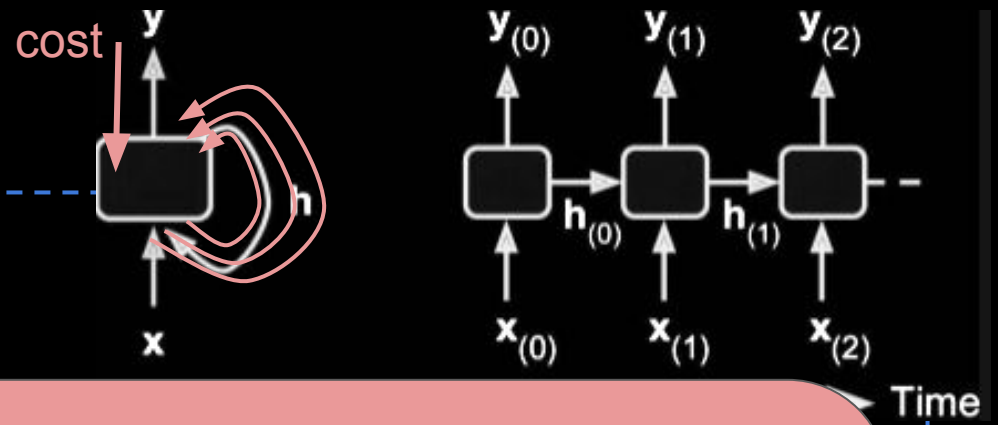
To find the gradient for the overall graph, we use **back propogation,** which *essentially* chains together the gradients for each node (function) in the graph.

Back Propagation



(Geron, 2017)

Back Propagation

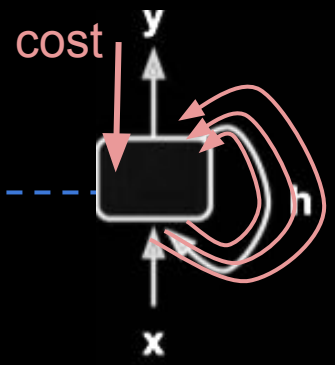


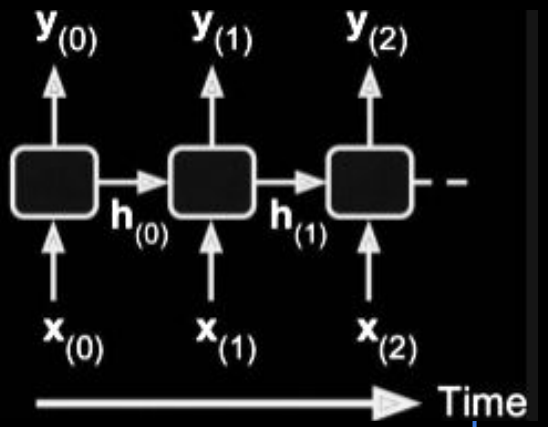

```
def forward(self, X):
    #Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.
    hidden state
        y(i) = nn.log_softmax(to


    ...

loss_func = nn.NLLLoss() #nega
            #torch.mean(-torch.sum(y*y_pred))
```

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).

Back Propagation





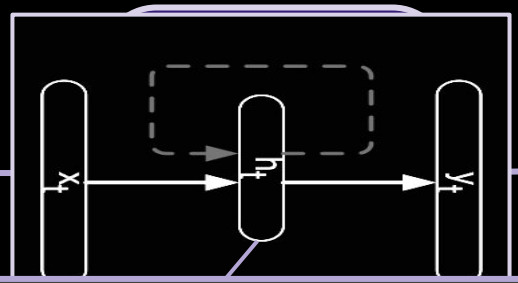

```
def forward(self, X):
    #Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.matmul(U,h(i-1))+ torch.matmul(W,x(i))) #update
    hidden state
        y(i) = nn.log_softmax(torch.matmul(V, h(i))) #update output


    ...

loss_func = nn.NLLLoss() #negative log likelihood loss
            #torch.mean(-torch.sum(y*y_pred))
```

How to address exploding and vanishing gradients?

How to address exploding and vanishing gradients?

Last word

(the)

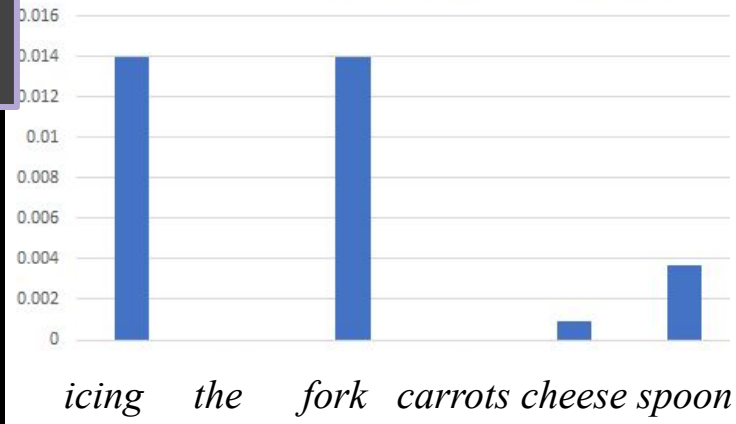


What is the next word in the sequence?

h_t: a vector that we hope "stores" relevant history from previous inputs:
He, at, the, cake, with,

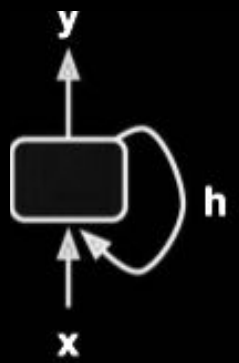
Training Corpus

training (fit, learn)



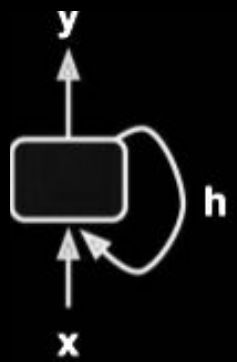
How to address exploding and vanishing gradients?

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



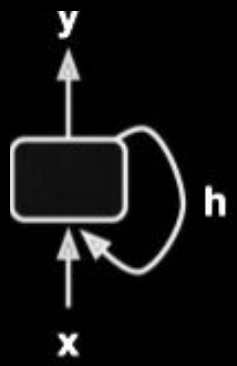
How to address exploding and vanishing gradients?

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

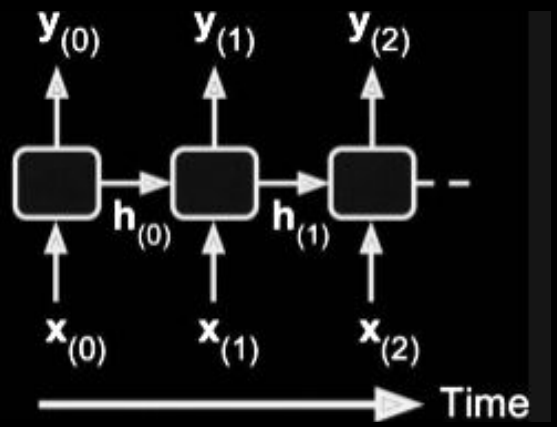


How to address exploding and vanishing gradients?

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



RNN model

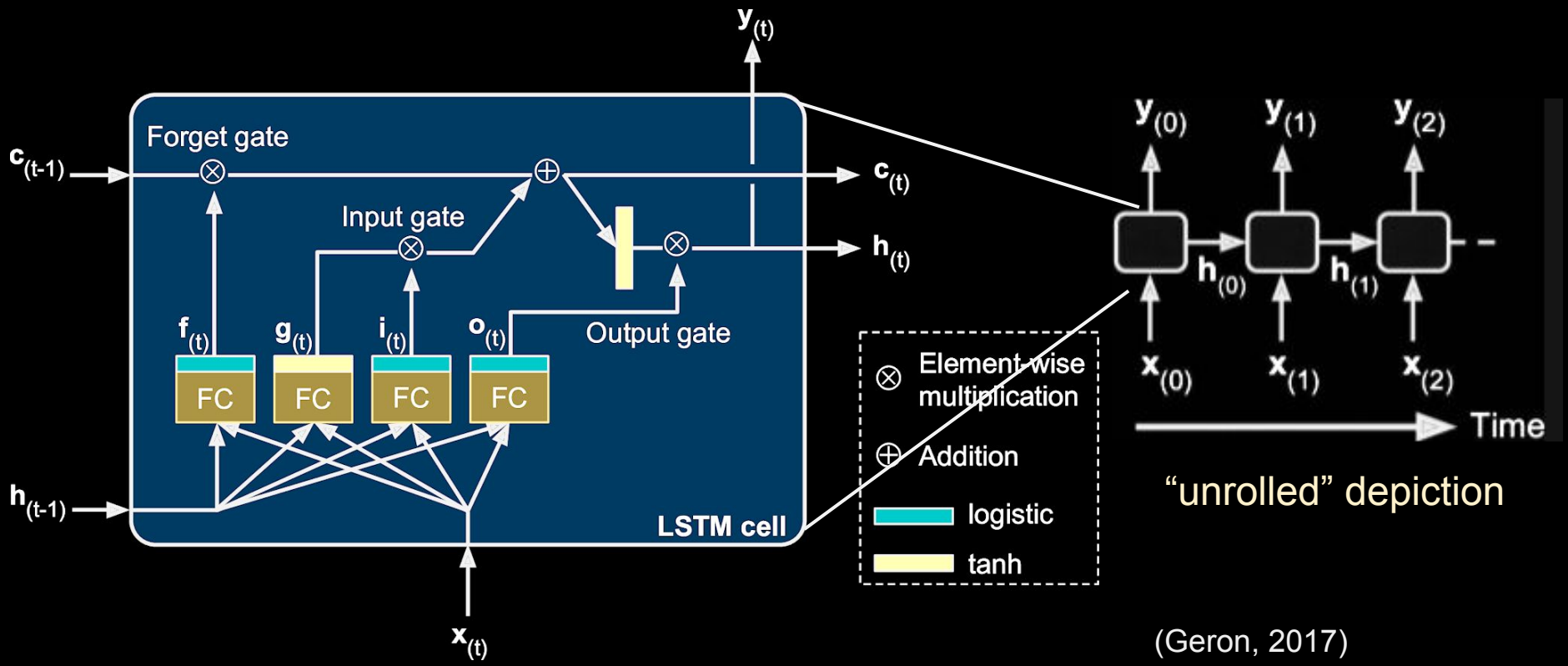


"unrolled" depiction

(Geron, 2017)

How to address exploding and vanishing gradients?

The LSTM Cell

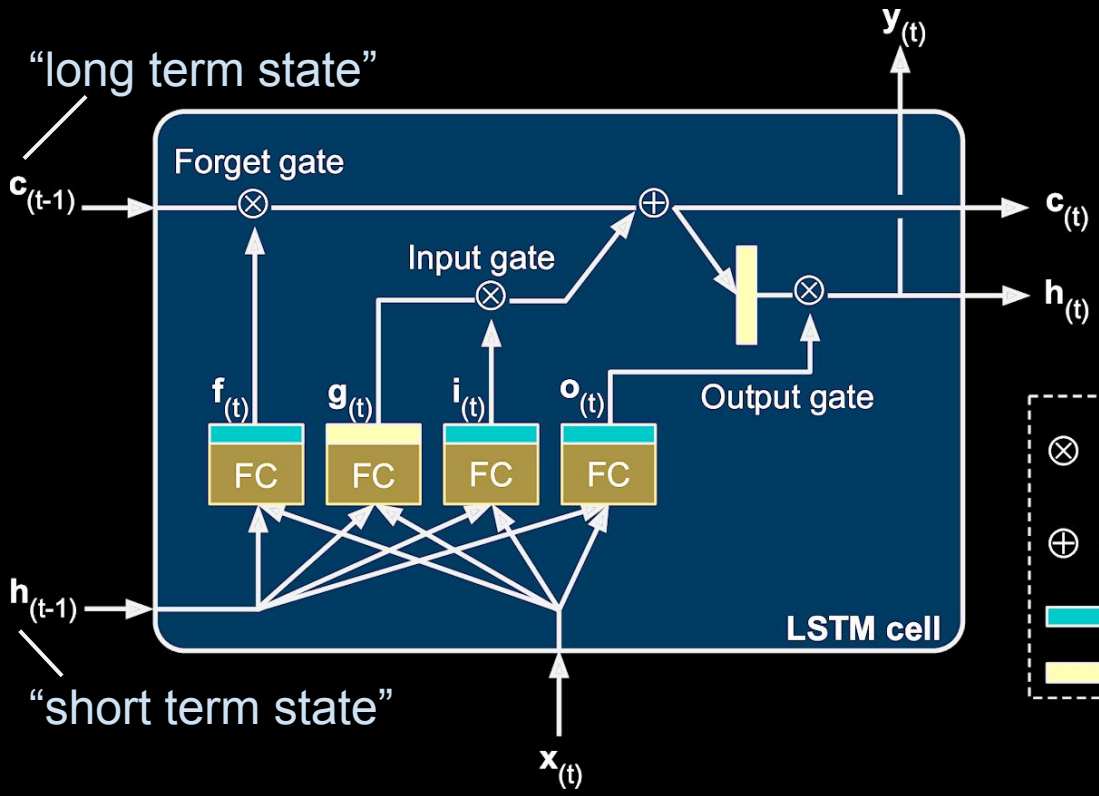


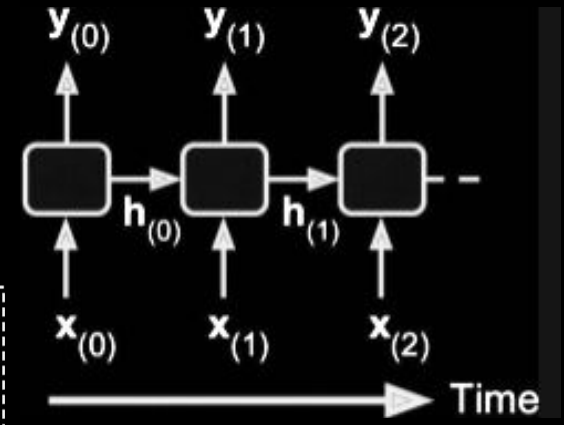
"unrolled" depiction

(Geron, 2017)

How to address exploding and vanishing gradients?

The LSTM Cell





"unrolled" depiction

Element-wise multiplication

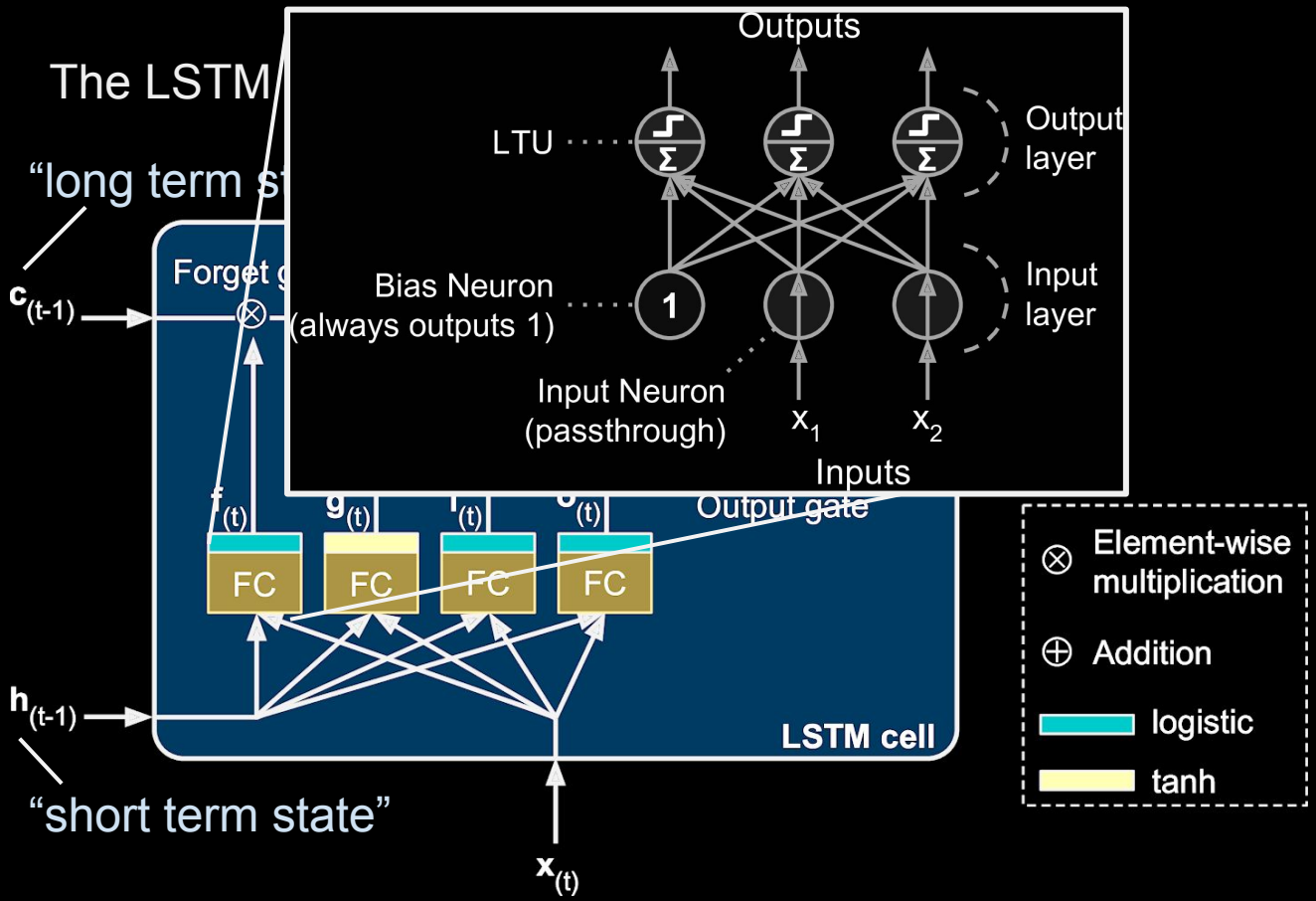
Addition

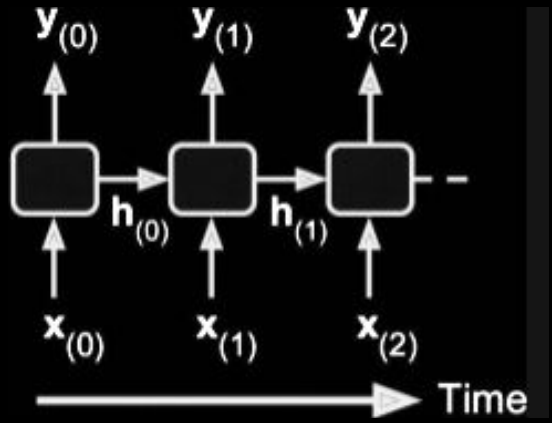
logistic

tanh

(Geron, 2017)

How to address exploding and vanishing gradients?



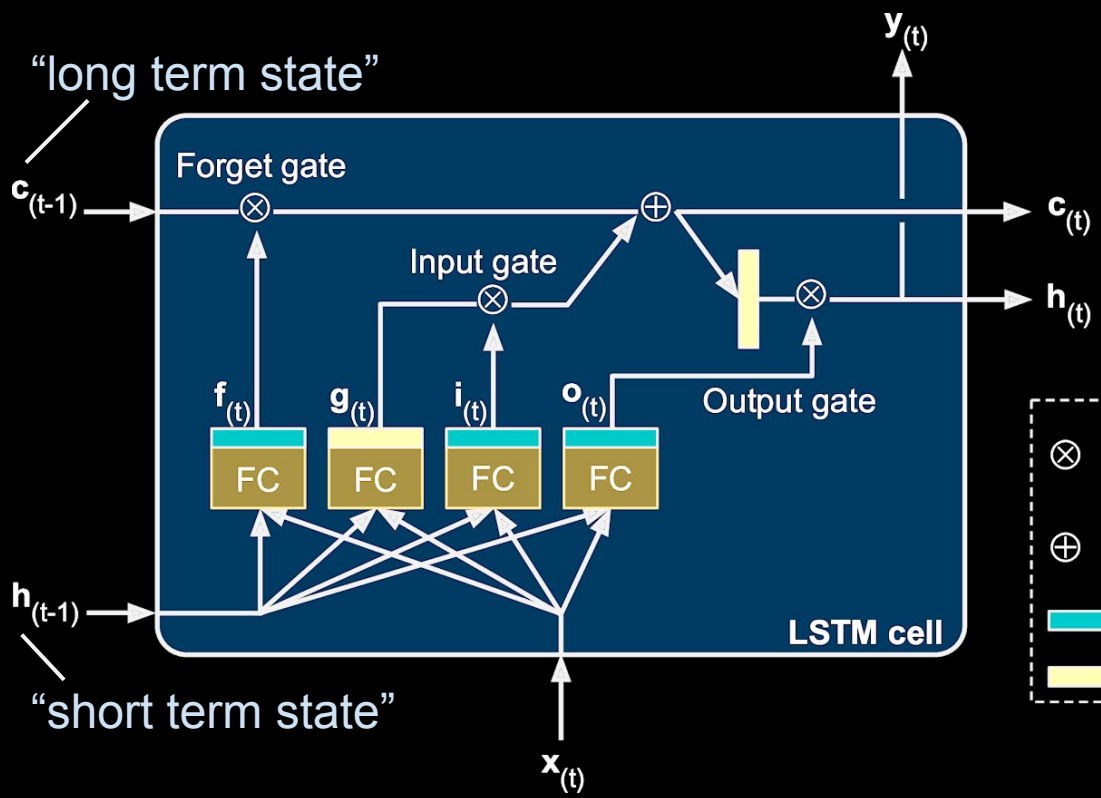


"unrolled" depiction

(Geron, 2017)

How to address exploding and vanishing gradients?

The LSTM Cell



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

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

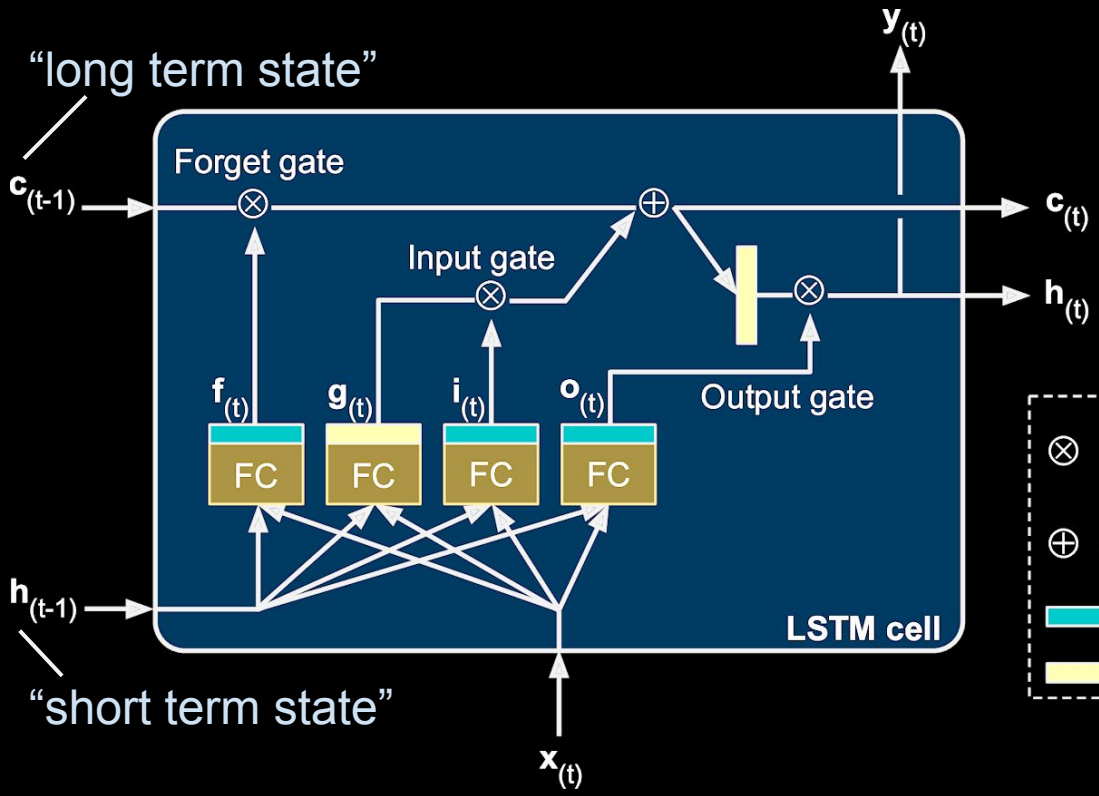
$$\mathbf{g}_{(t)} = \tanh(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g)$$

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

bias term

LSTM

The LSTM Cell



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

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

$$\mathbf{o}_{(t)} = \sigma(\mathbf{W}_{xo}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o)$$

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

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

$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)})$$

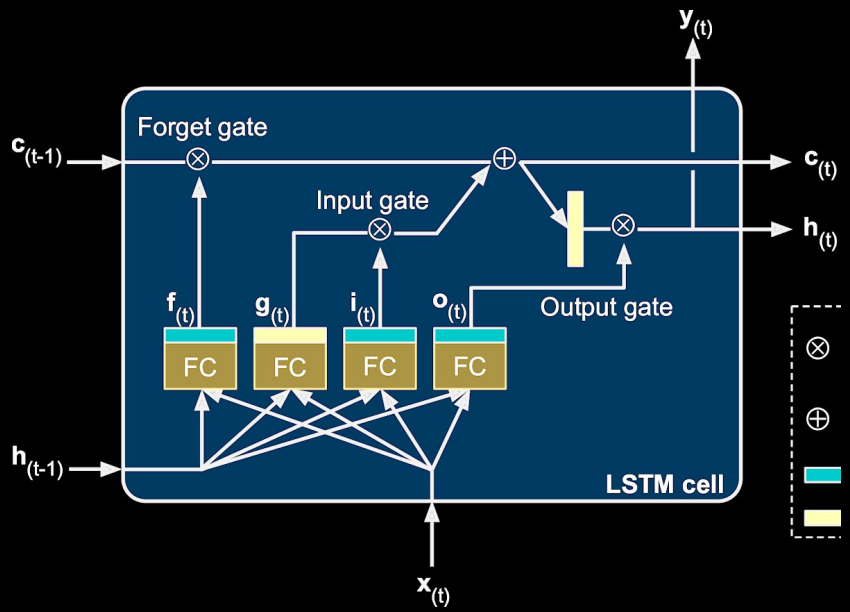
⊗ Element-wise multiplication

⊕ Addition

logistic

tanh

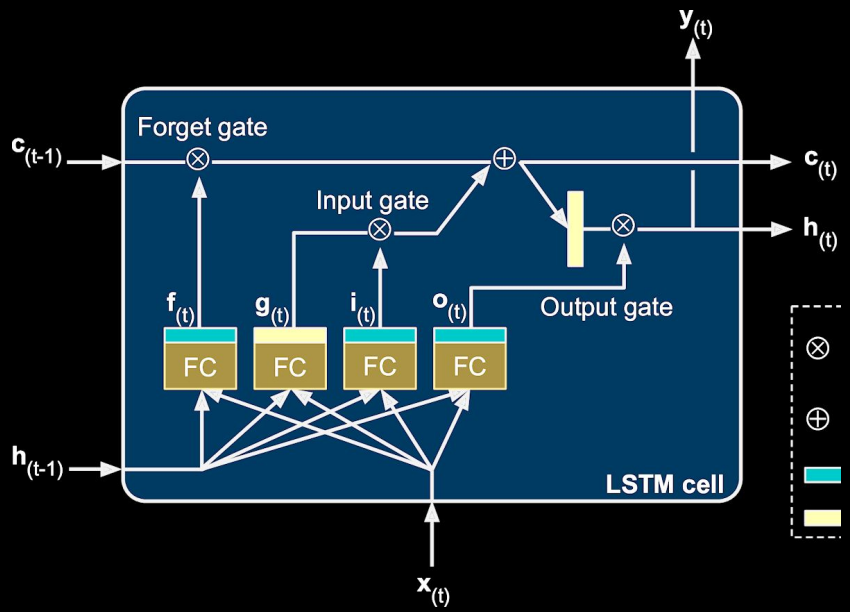
Input to LSTM



?

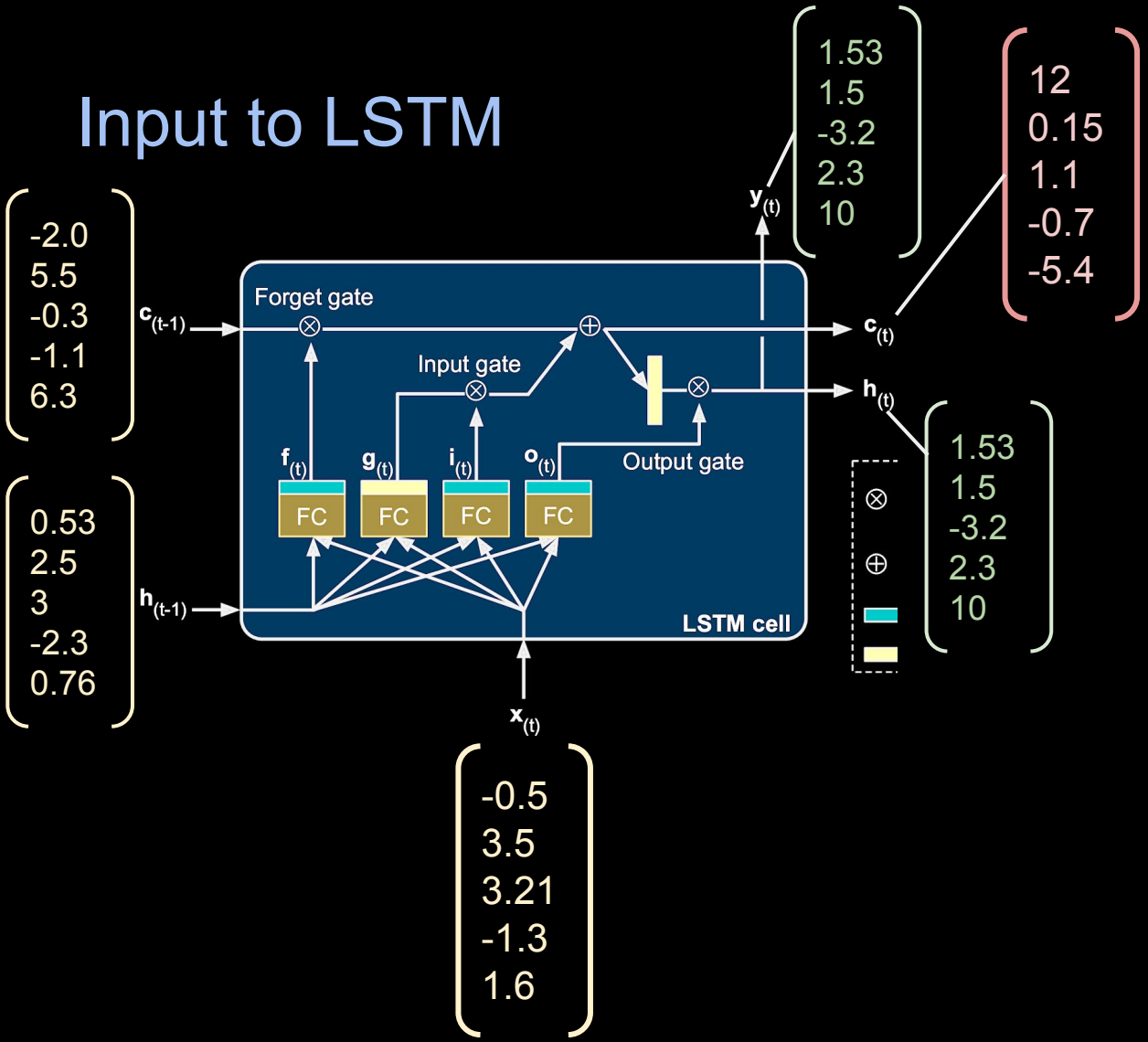
- One-hot encoding?
- Word Embedding

Input to LSTM

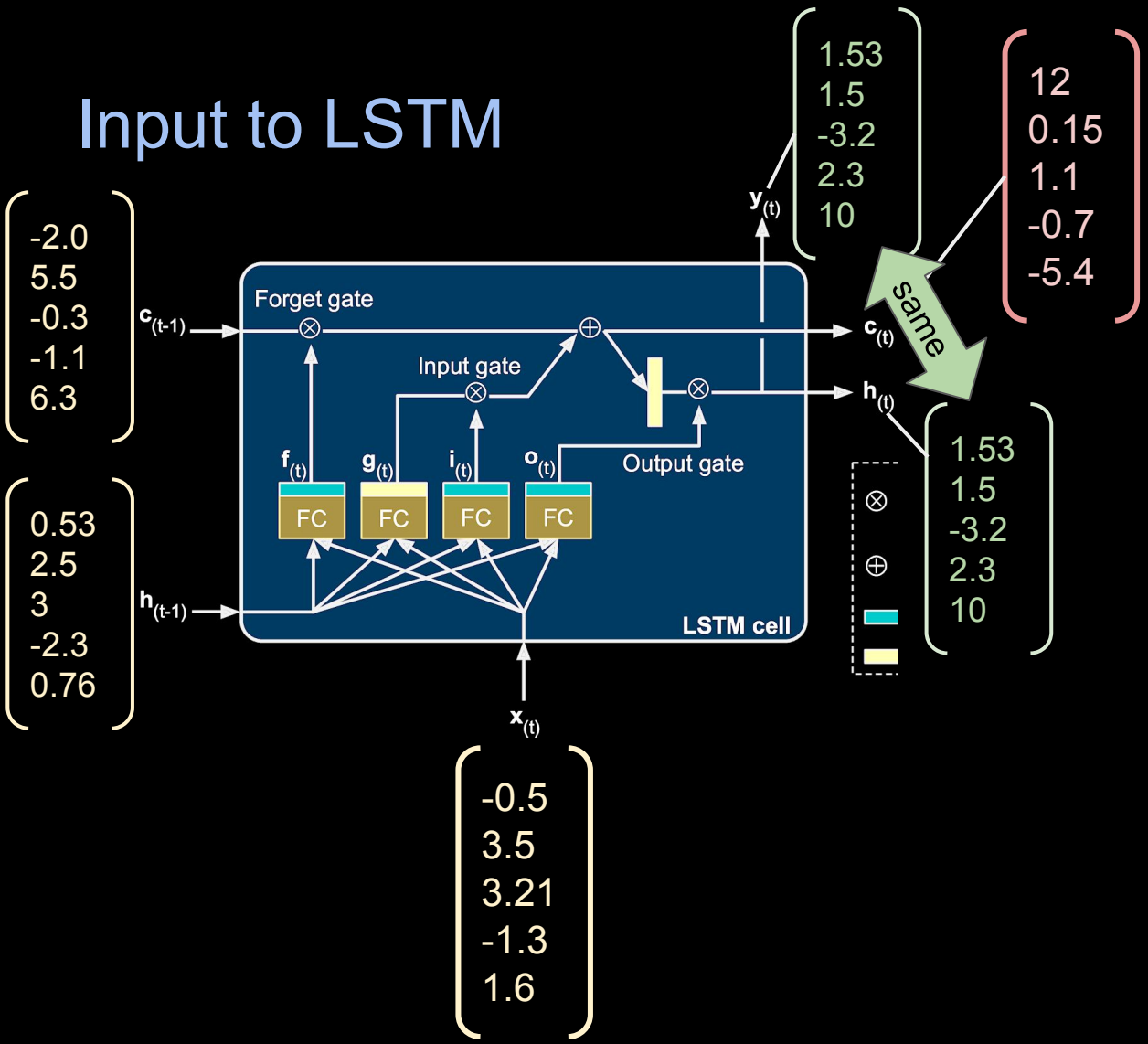


$$\begin{pmatrix} -0.5 \\ 3.5 \\ 3.21 \\ -1.3 \\ 1.6 \end{pmatrix}$$

Input to LSTM

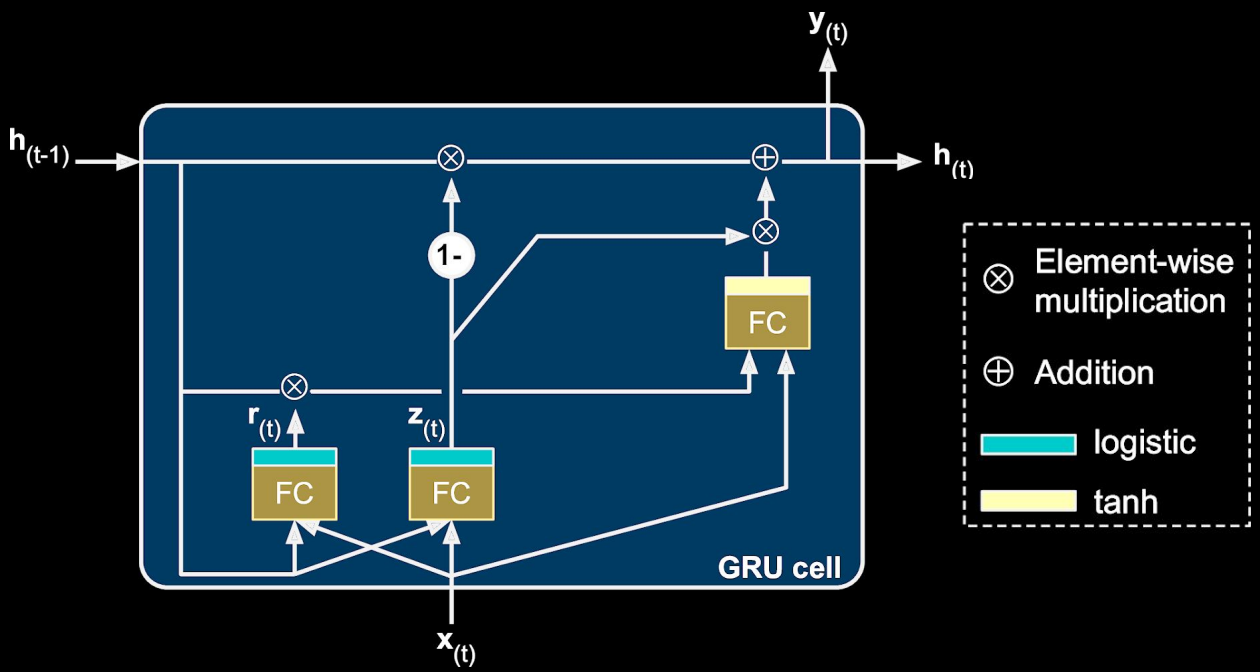


Input to LSTM



The GRU

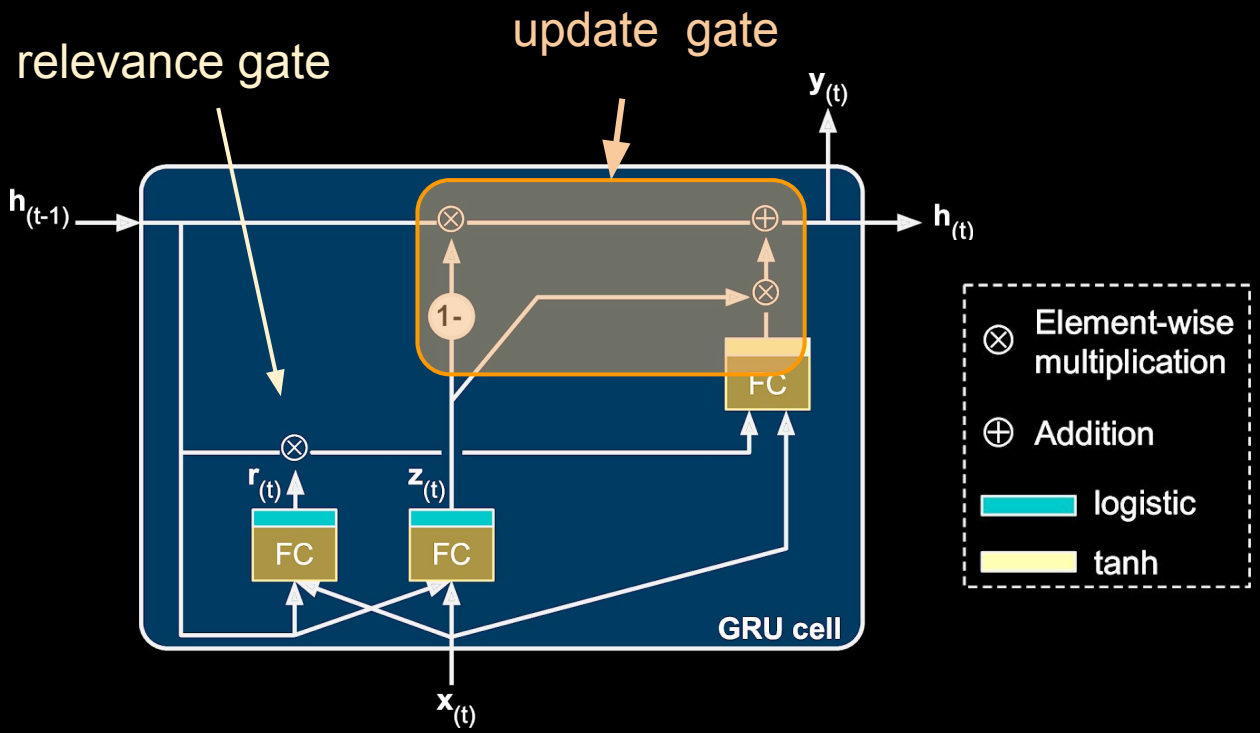
Gated Recurrent Unit



(Geron, 2017)

The GRU

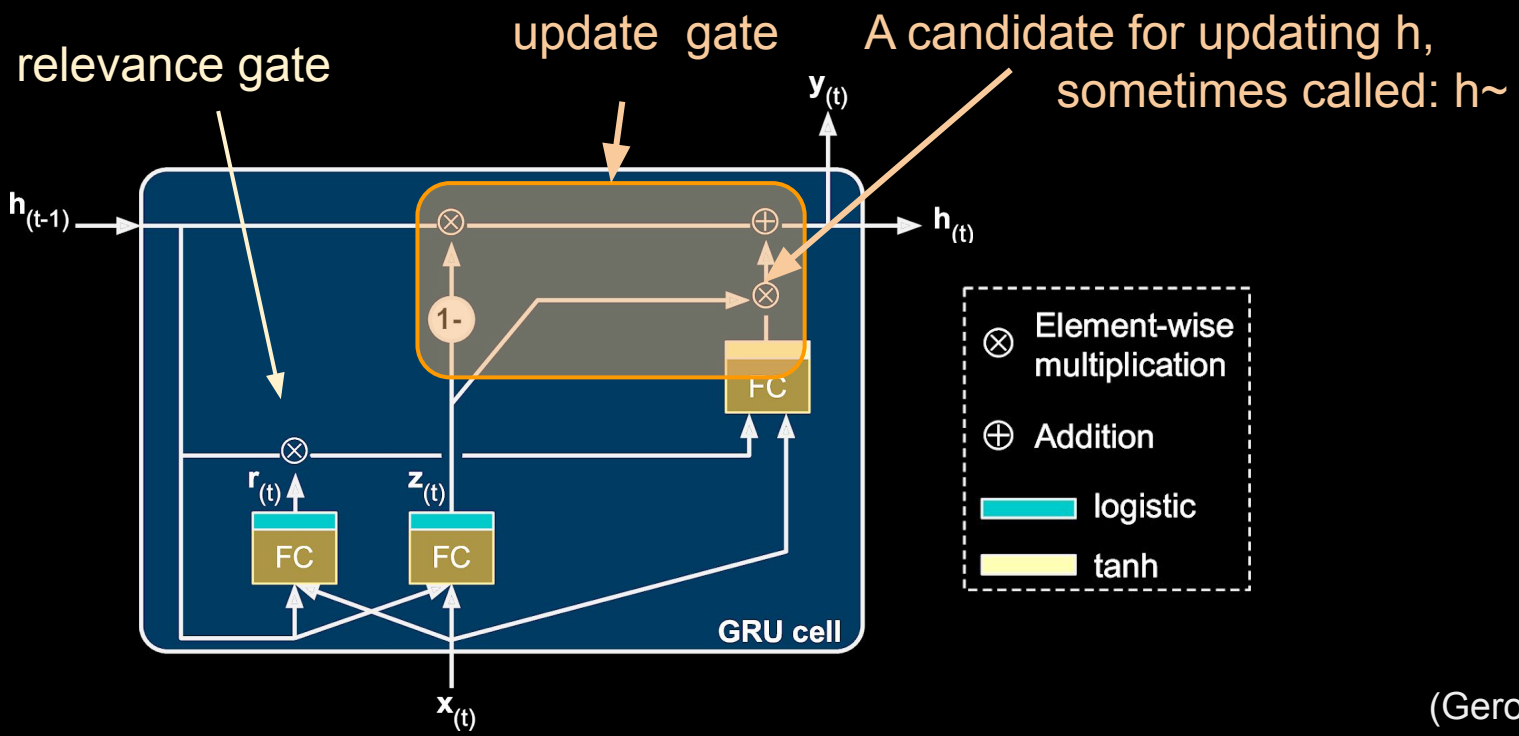
Gated Recurrent Unit



(Geron, 2017)

The GRU

Gated Recurrent Unit

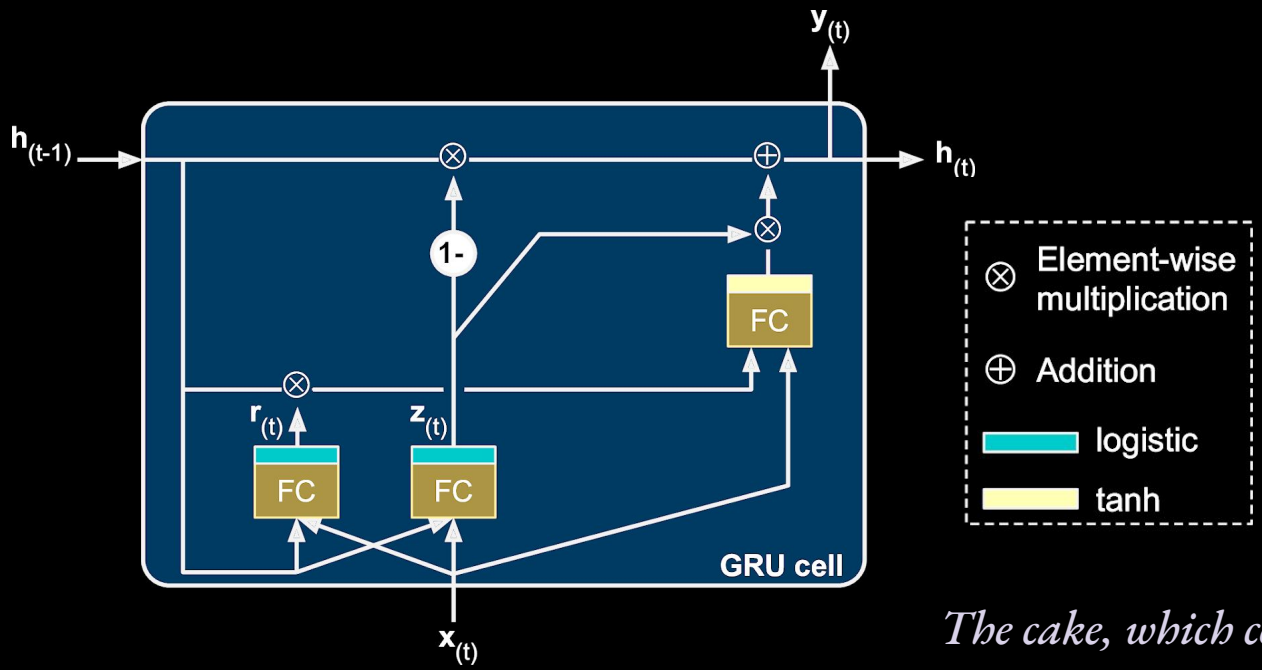


(Geron, 2017)

The GRU

Gated Recurrent Unit

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



The cake, which contained candles, was eaten.

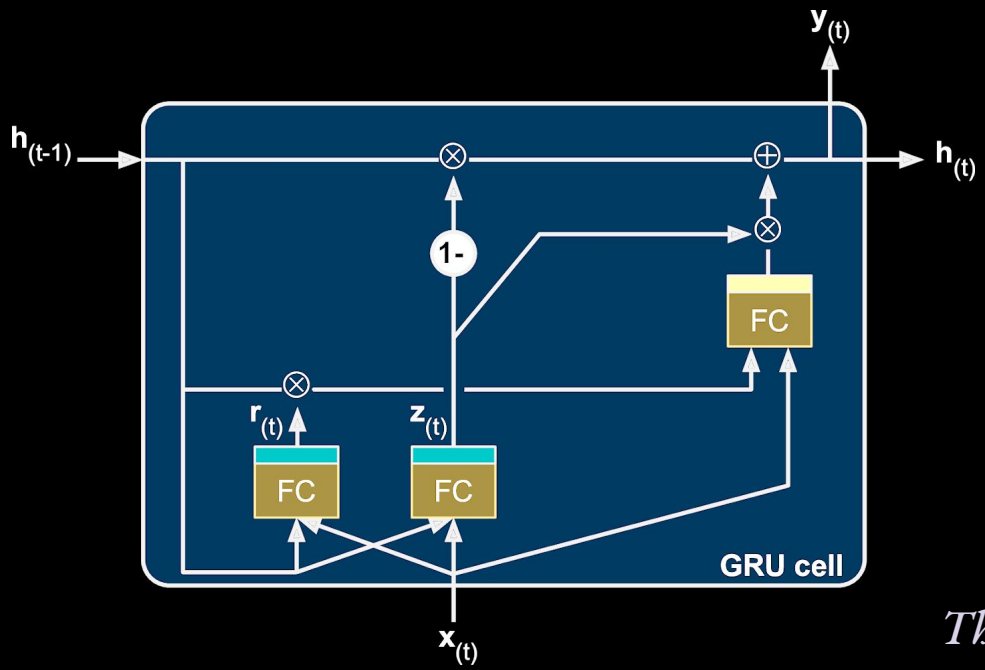
What about the gradient?

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

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

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

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



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,

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

The cake, which contained candles, was eaten.

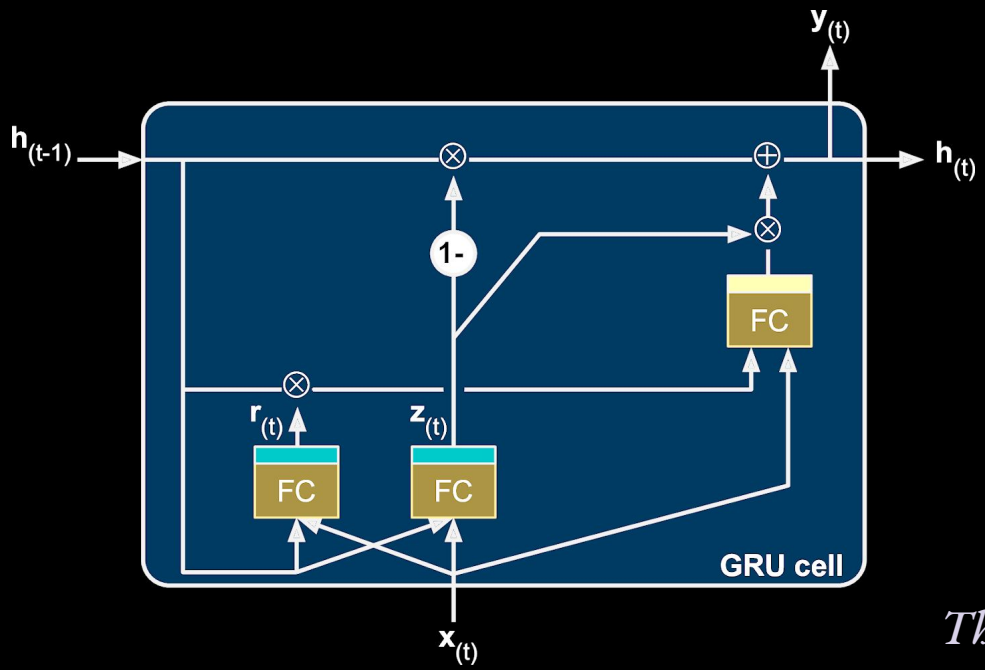
What about the gradient?

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

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

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

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



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,

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

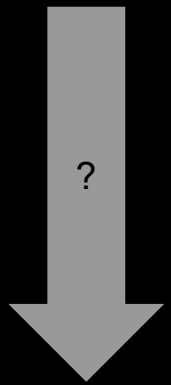
This tends to keep the gradient from vanishing since the same values will be present through multiple times in backpropagation through time. (The same idea applies to LSTMs but is easier to see here).

The cake, which contained candles, was eaten.

How to train an LSTM-style RNN

```
RNN_cost = torch.mean(-torch.sum(y*torch.log(y_pred))
            #where did this come from?
```

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



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 = torch.mean(-torch.sum(y*torch.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 = torch.mean(-torch.sum(y*torch.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 = torch.mean(-torch.sum(y*torch.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 = torch.mean(-torch.sum(y*torch.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