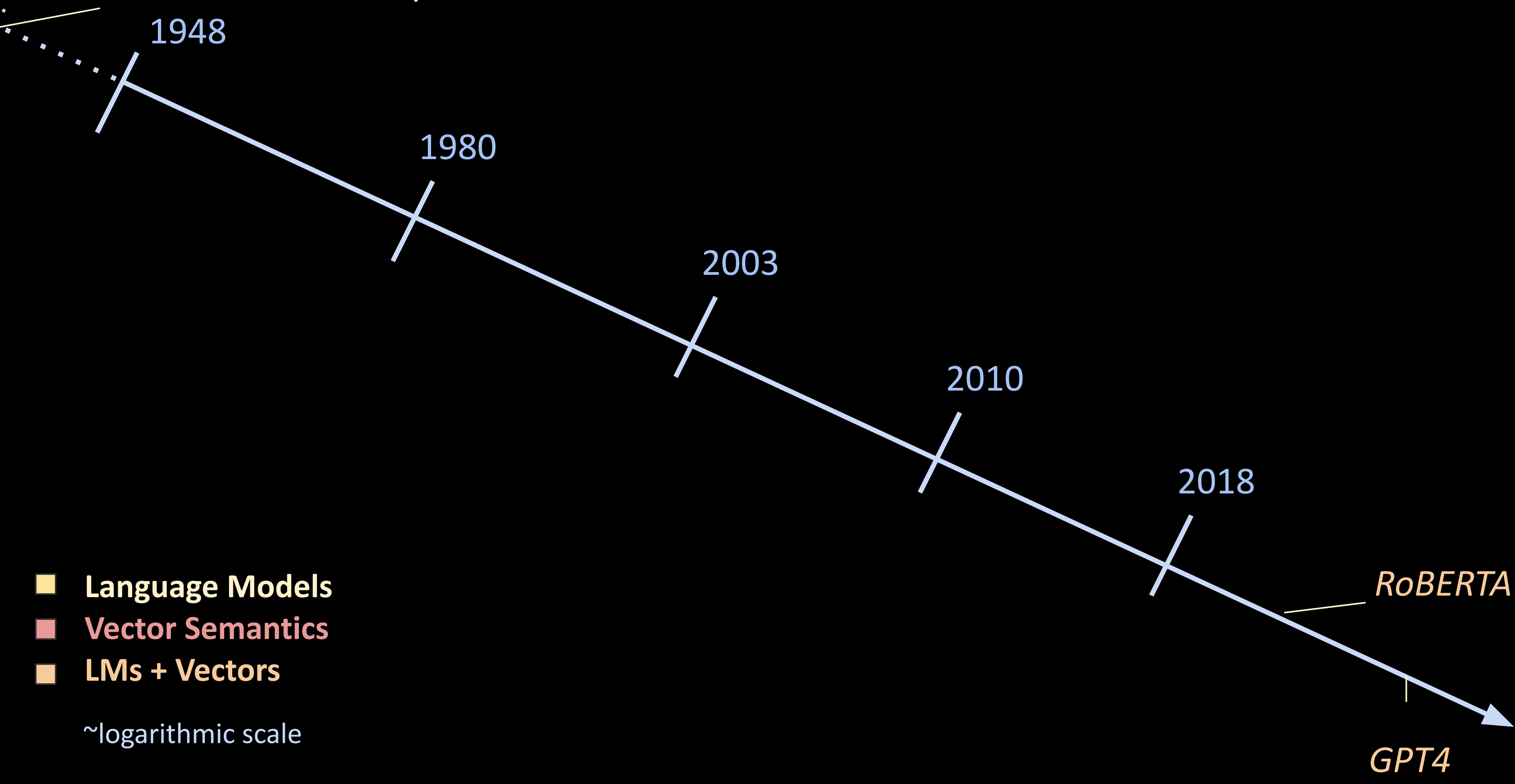
Deep Learning & Recurrent Neural Networks

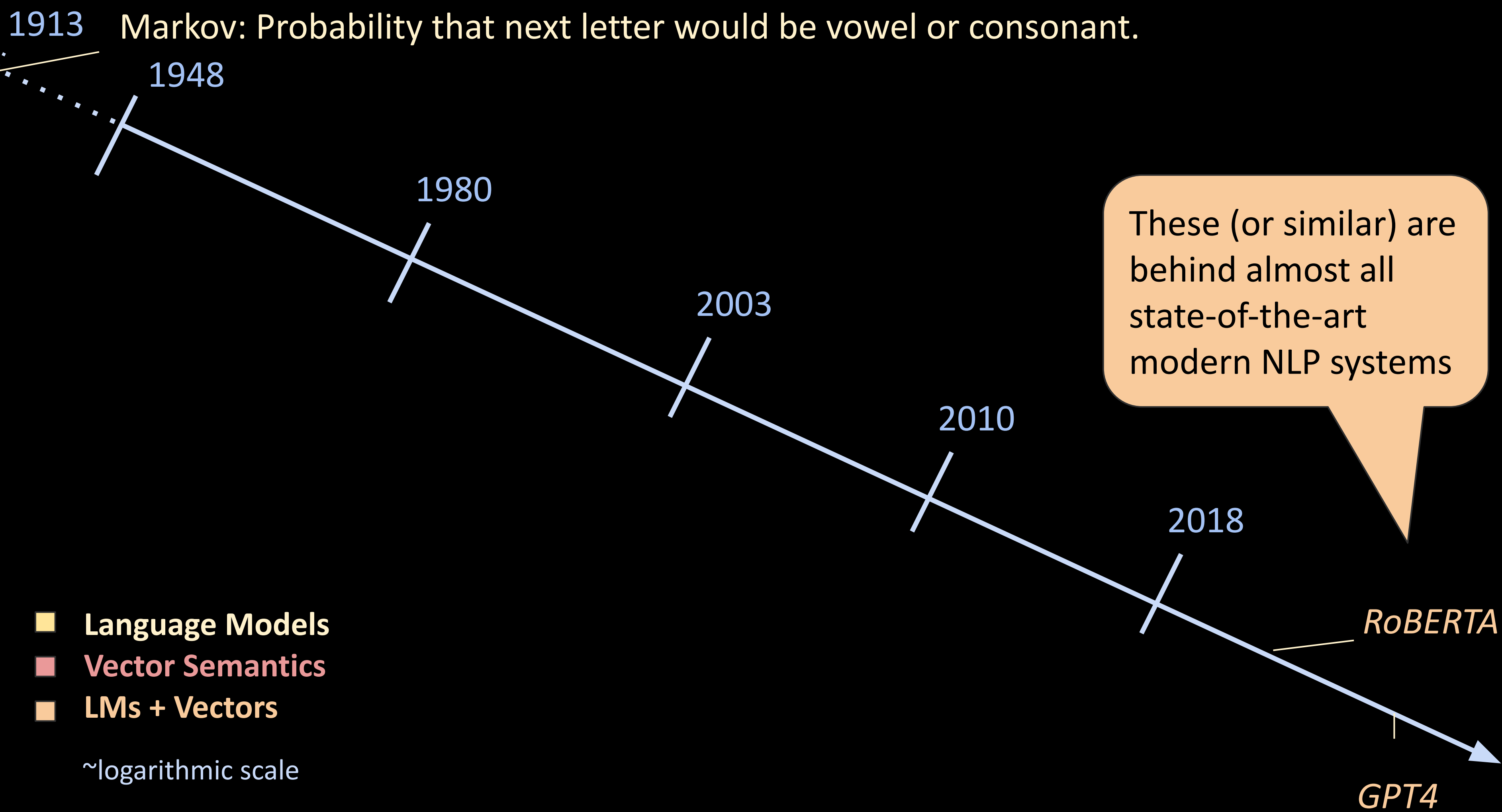
CSE538 - Spring 2025

Timeline: *Language Modeling* and *Vector Semantics*

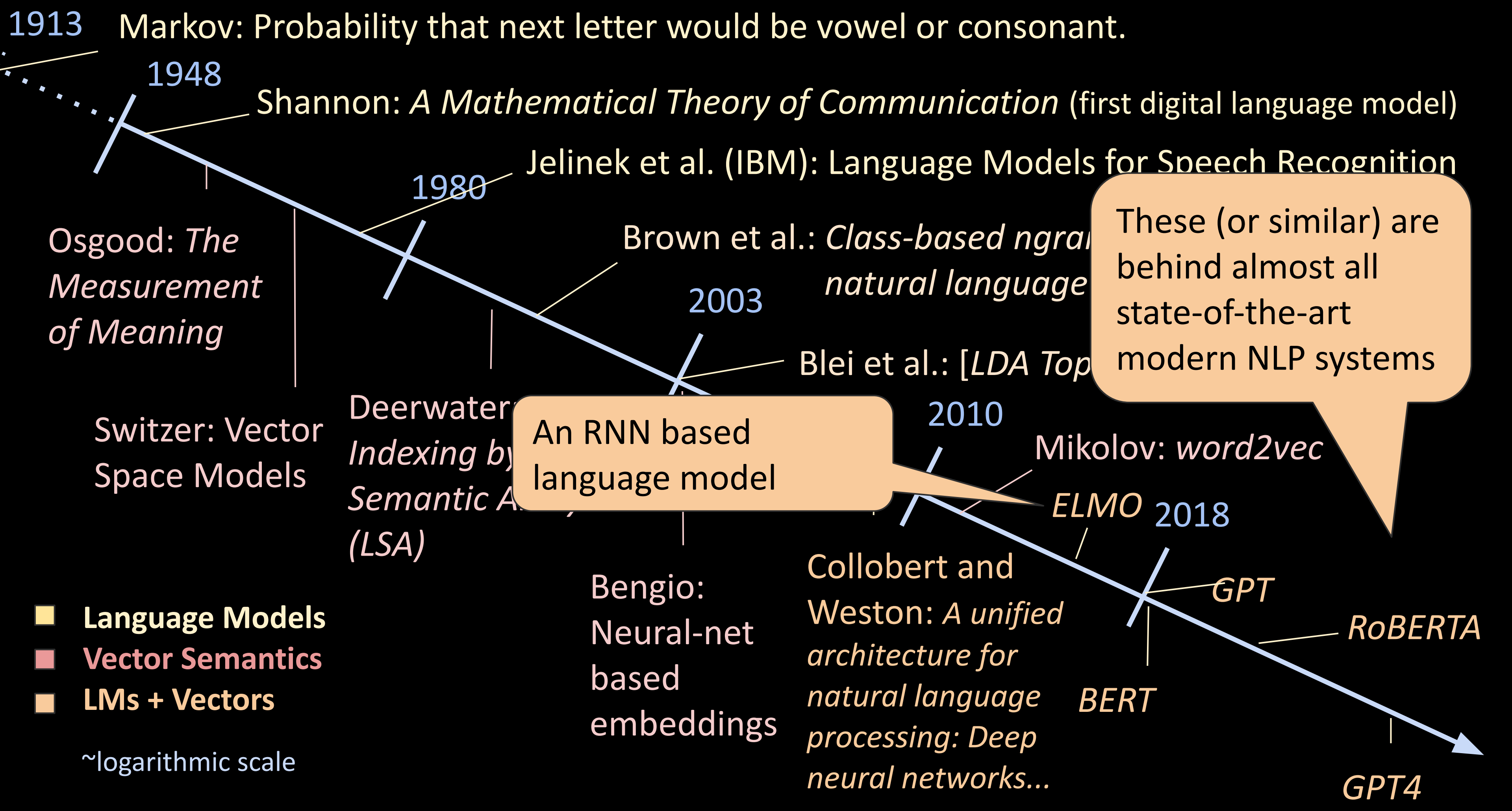
1913 Markov: Probability that next letter would be vowel or consonant.



Timeline: *Language Modeling* and *Vector Semantics*



Timeline: *Language Modeling* and *Vector Semantics*



Attention

⬆

RNNs

⬆

Neural Networks

Artificial Neural Networks

What is it?

Artificial Neural Networks

What is it?

- Biologically inspired computing model
- Learn patterns from the data
- Can even approximate nonlinear functions in the nature!

Artificial Neural Networks

What is it?

- Biologically inspired **computing model**
- Learn patterns from the data
- Can even approximate nonlinear functions in the nature!

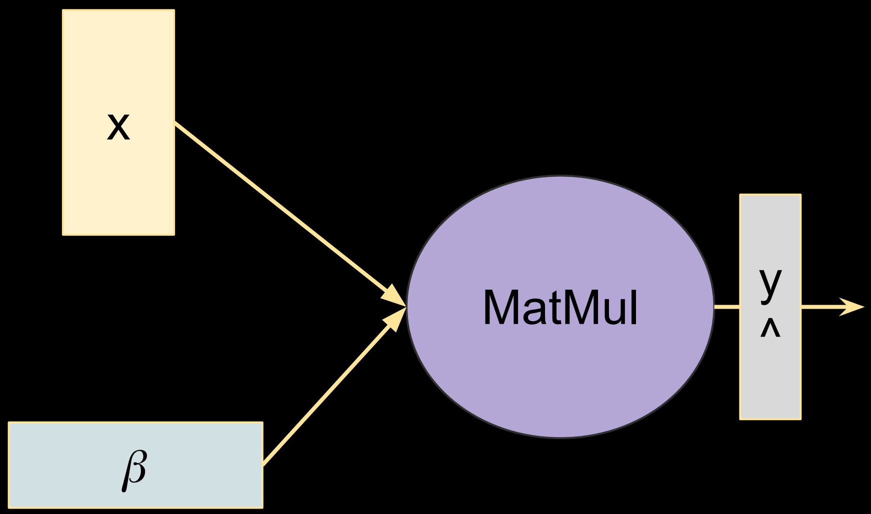
Artificial Neural Networks

What is it?

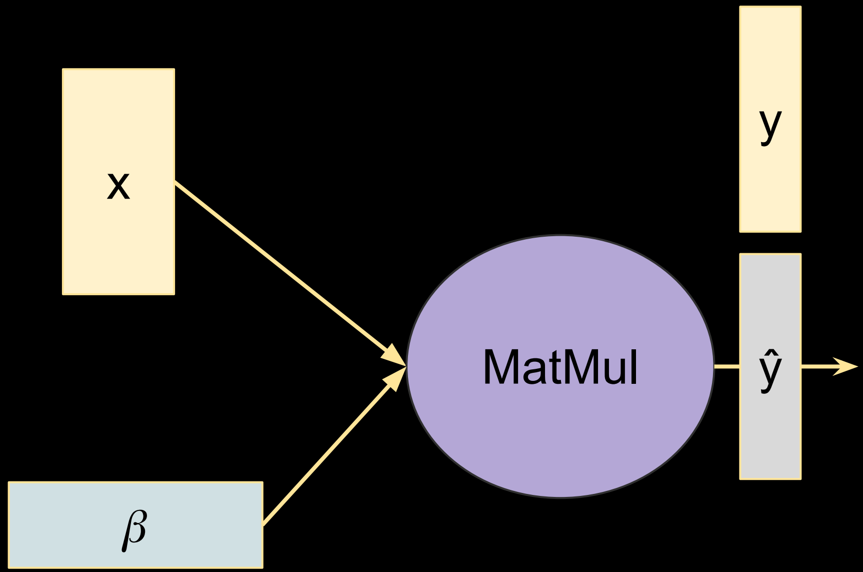
How did we do this?

- Biologically inspired **computing model**
- Learn patterns from the data
- Can even approximate nonlinear functions in the nature!

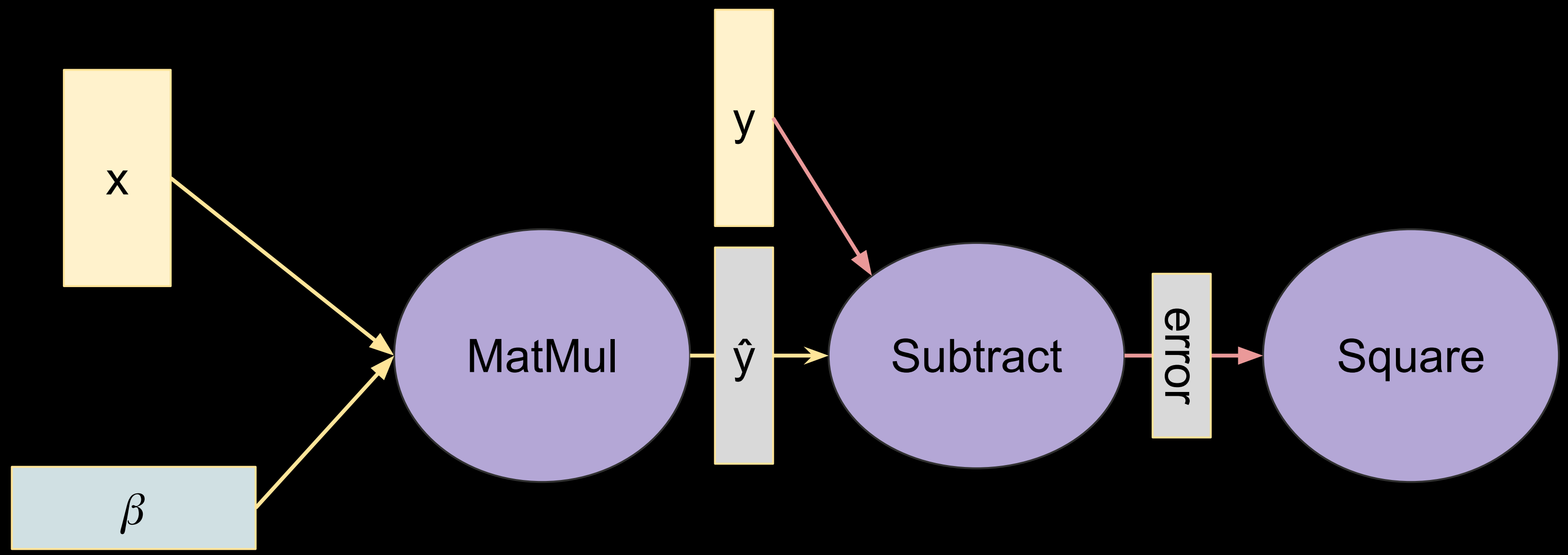
Artificial Neural Networks



Artificial Neural Networks

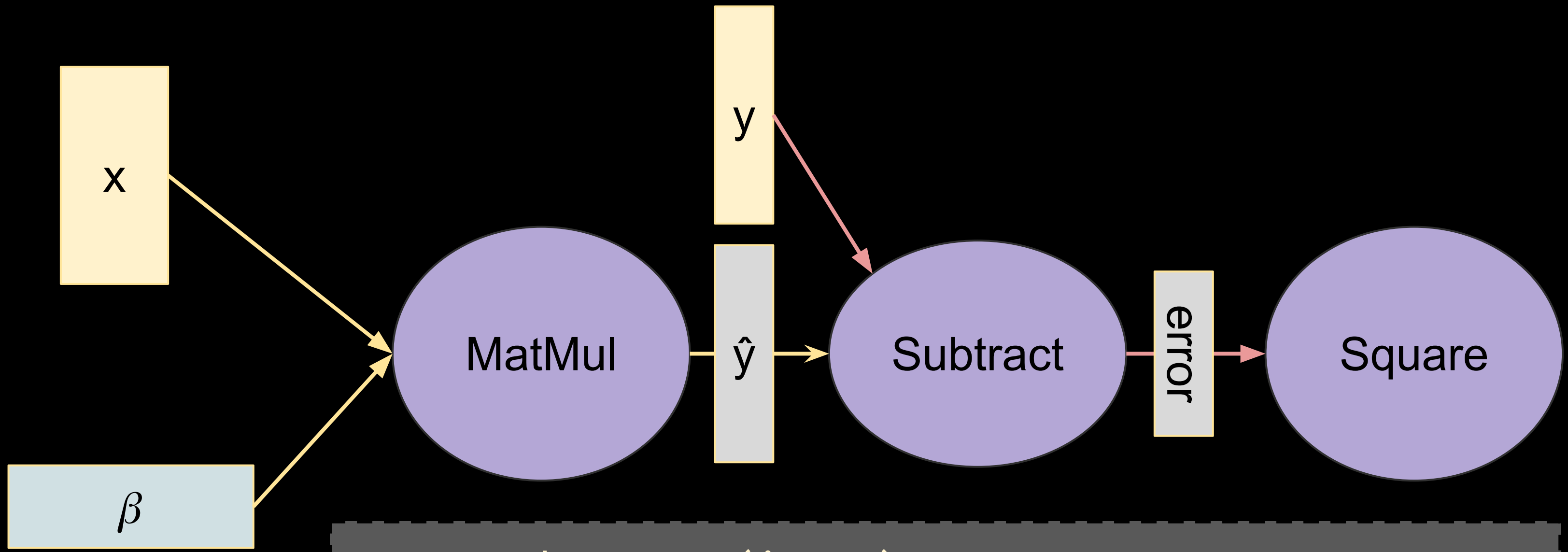


Artificial Neural Networks



Artificial Neural Networks

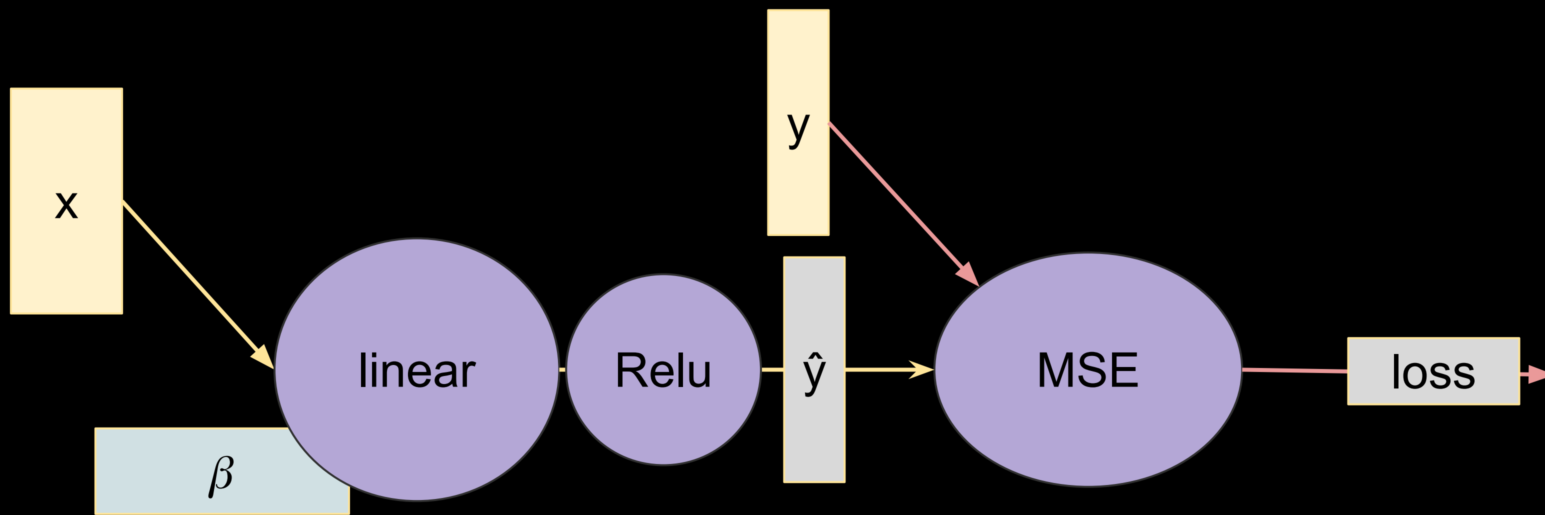
```
import torch
from torch import nn
```




```
x = torch.Tensor(input)
beta = torch.random.randn(X.shape, 1)
yhat = torch.matmul(x, beta)

err = torch.Tenor(y) - yhat
loss = err**2
```

Artificial Neural Networks

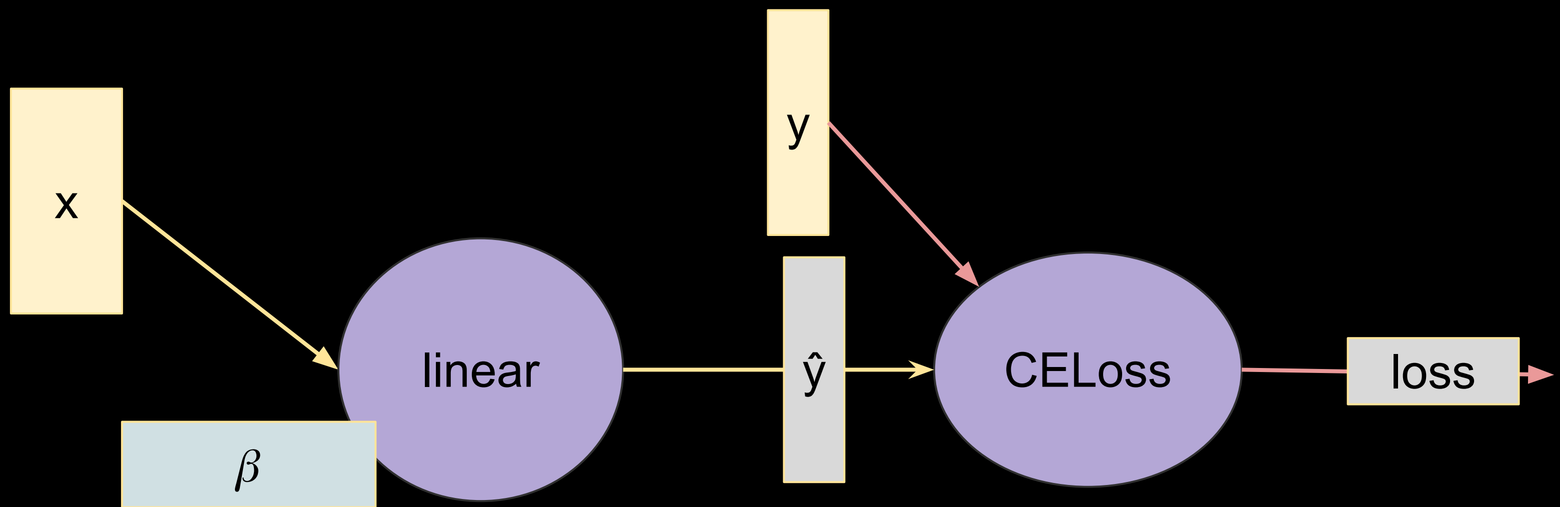
```
import torch
from torch import nn
```




```
x = torch.Tensor(input)
beta = torch.random.randn(X.shape, 1)
z = nn.linear(x) #beta stored within
yhat = nn.functional.relu(z) #activation func

loss = nn.MSELoss(yhat, torch.Tensor(y))
```

Artificial Neural Networks

```
import torch
from torch import nn
```




```
x = torch.Tensor(input)
beta = torch.random.randn(X.shape, 1)
yhat = torch.matmul(x, beta)

loss = nn.nn.CrossEntropyLoss(yhat, torch.Tensor(y))
#^contains logistic activation
```

But, how do we model complex systems using these linear systems?

Deep Learning

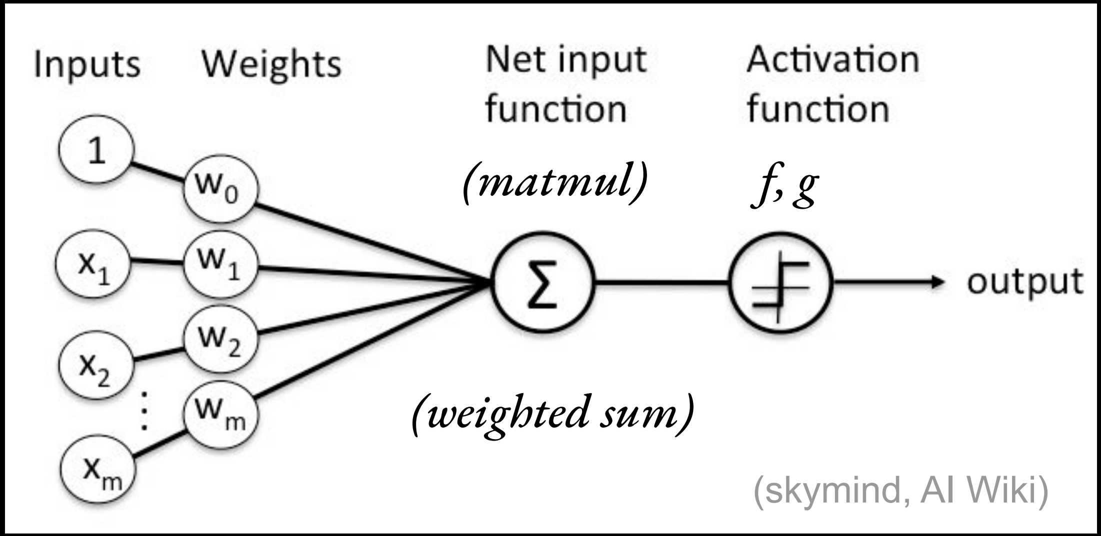
But, how do we model complex systems using these linear systems?

Deep Learning

linear regressions + non-linear activations

Deep Learning

linear regressions + non-linear activations



Activation Functions

$$z = h_{(t)}W$$

Common Activation Functions

$z = h_{(t)}W$

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

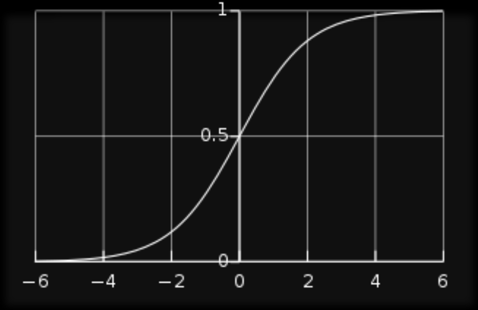
Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1)$

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

Common Activation Functions

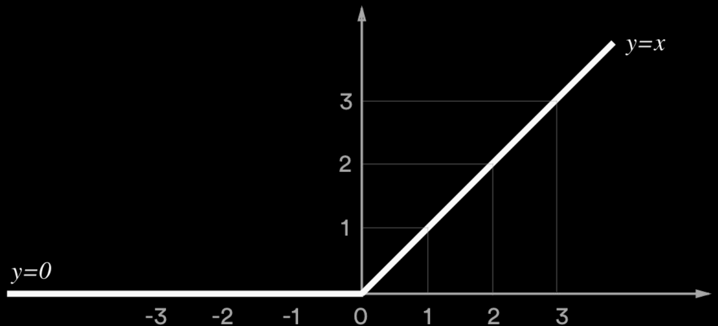
$z = h_{(t)}W$

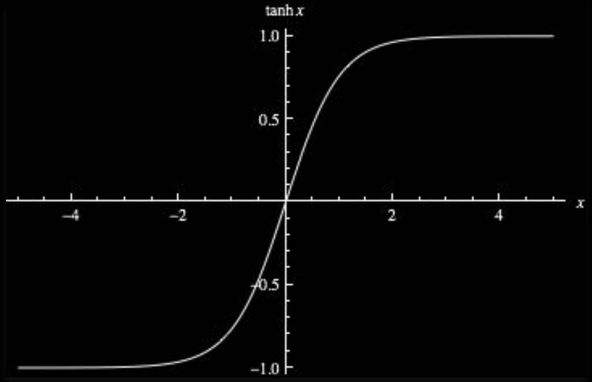
Logistic: $\sigma(z) = 1 / (1 + e^{-z})$



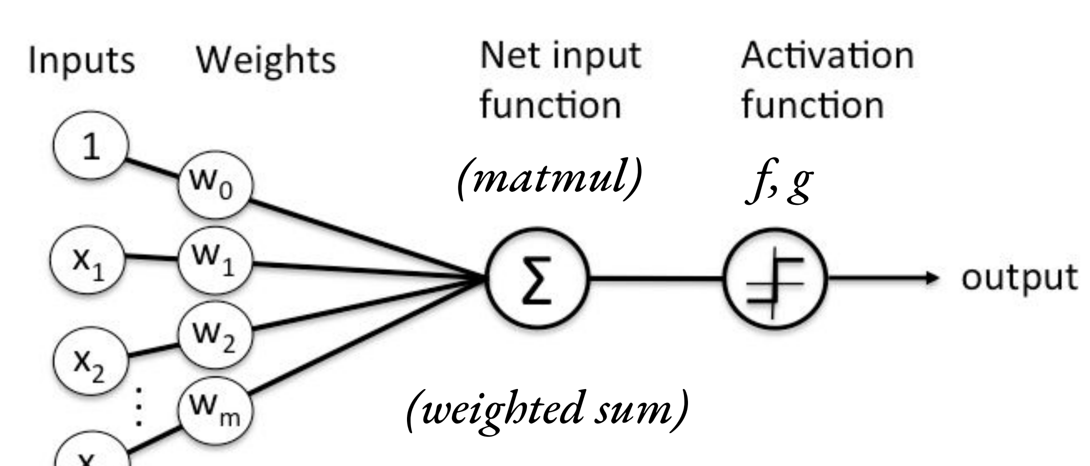
Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1)$

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





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



(skymind, AI Wiki)

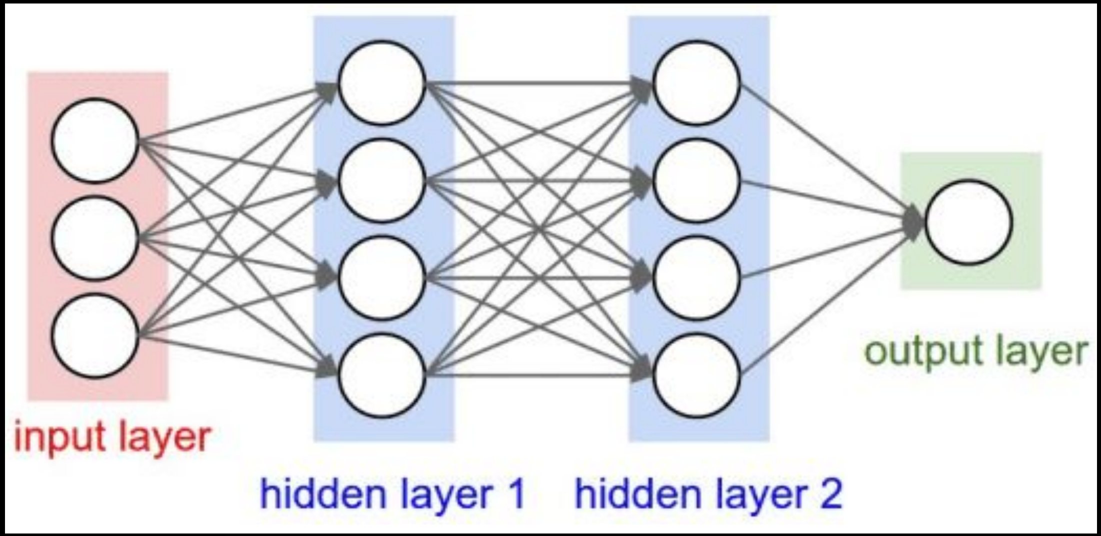
Activation Function

$h_1 = g(xW)$

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)



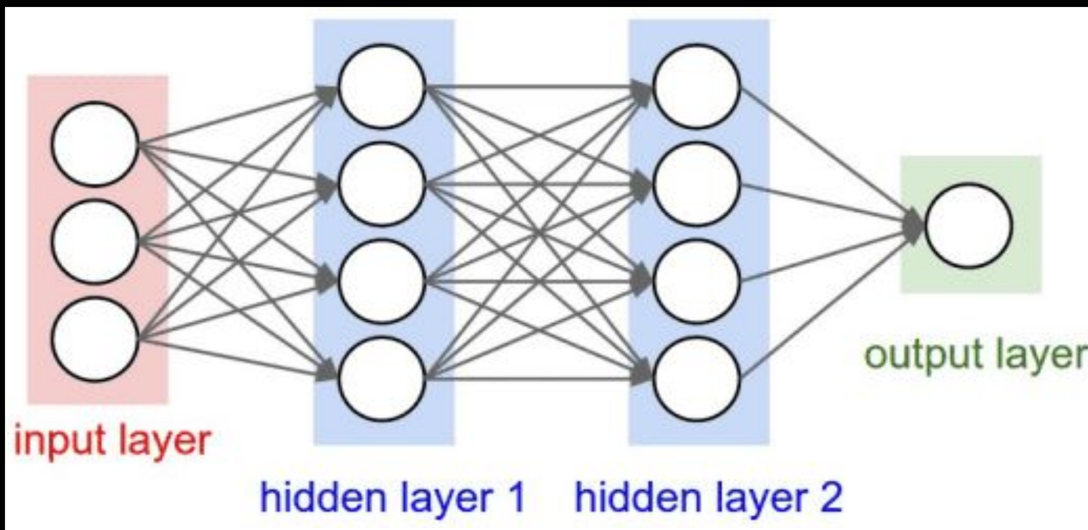
Activation Function

$$h_1 = g_1(xW_1)$$
$$h_2 = g_2(h_1W_2)$$
$$ypred = g_3(h_2W_{final})$$

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)



Activation Function

$h_1 = g_1(xW_1)$ #[n x 3] -> [n x 4]

$h_2 = g_2(h_1W_2)$ #[n x 4] -> [n x 4]

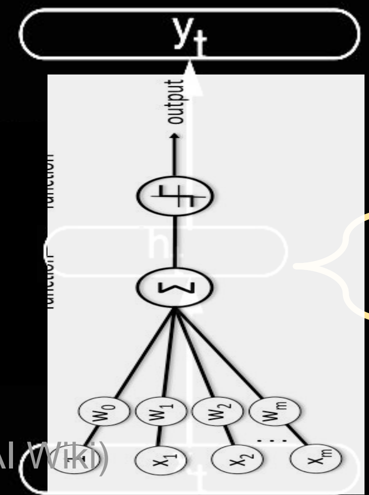
$ypred = g_3(h_2W_{final})$ #[n x 4] -> [n x 1]

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”



(skymind, AI Wiki)

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)

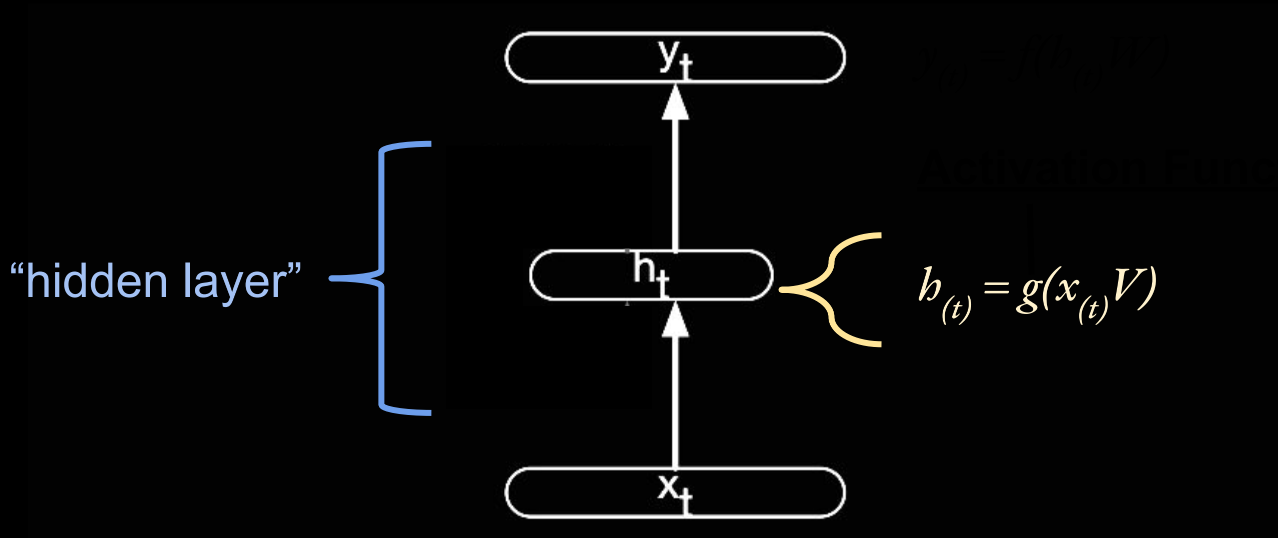
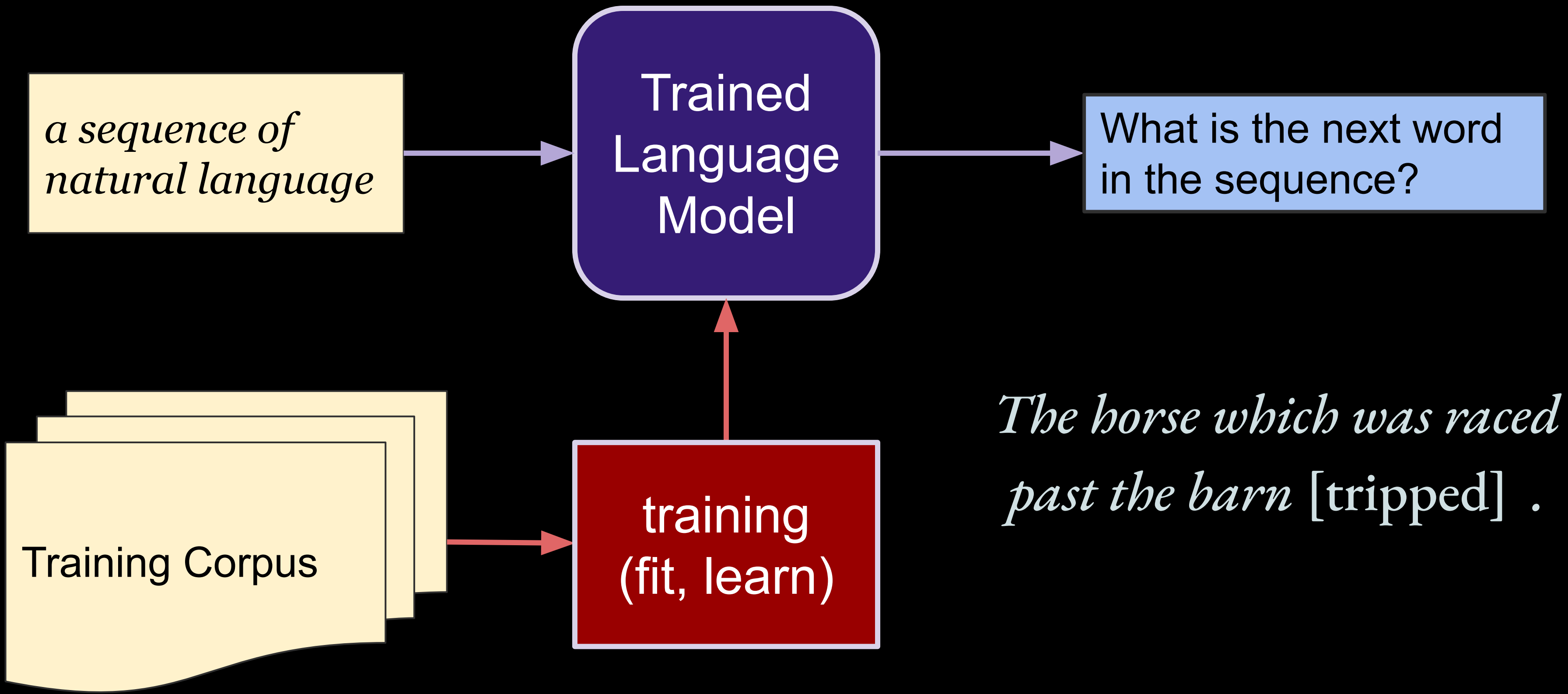


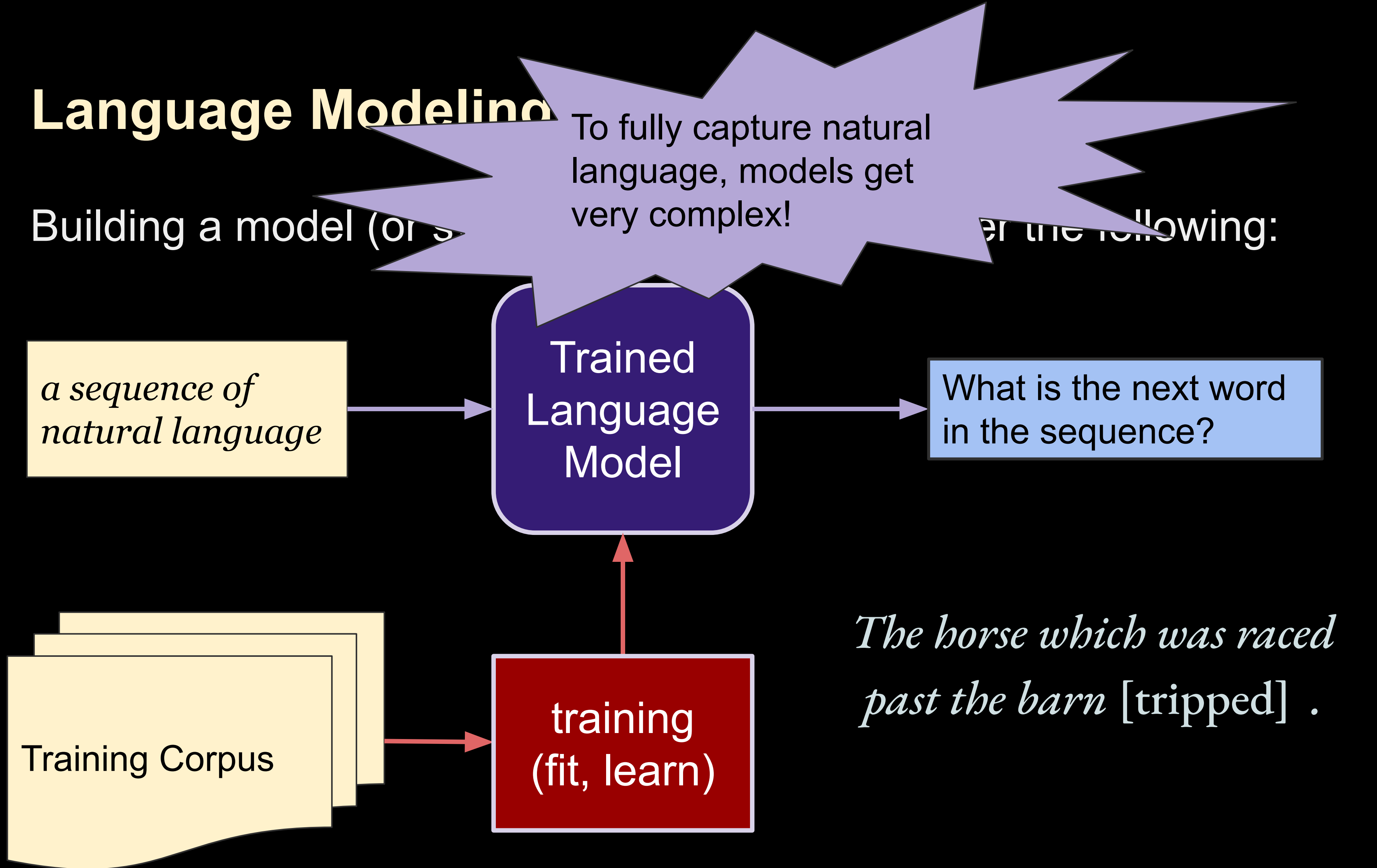
Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

Language Modeling

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




Language Modeling
Building a model (or s
er the following:
To fully capture natural language, models get very complex!
a sequence of natural language
Trained Language Model
What is the next word in the sequence?
Training Corpus
training (fit, learn)
The horse which was raced past the barn [tripped] .

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

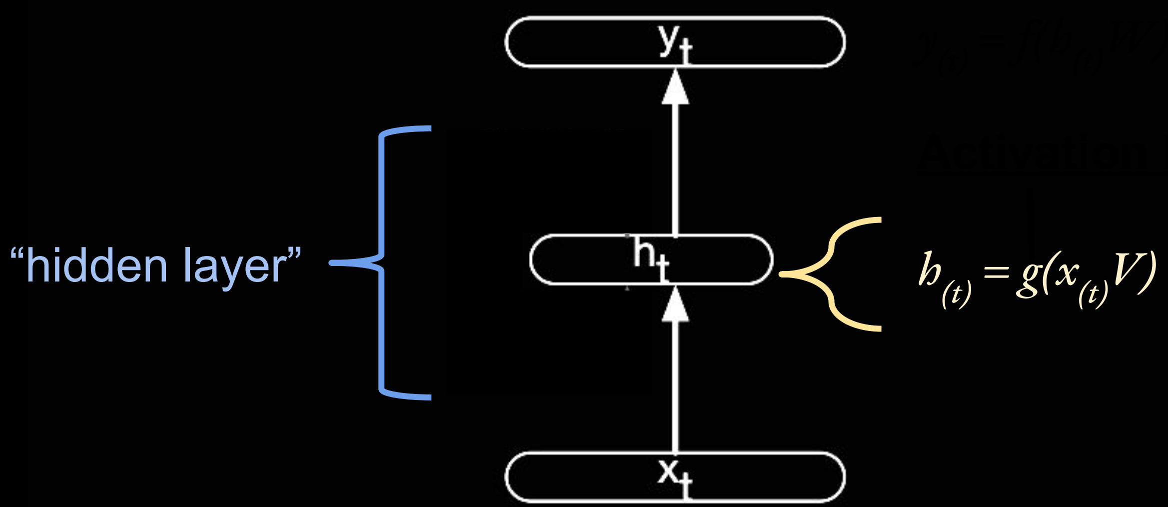
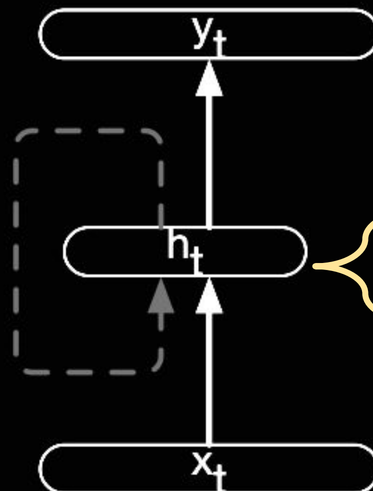


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

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

“hidden layer”



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

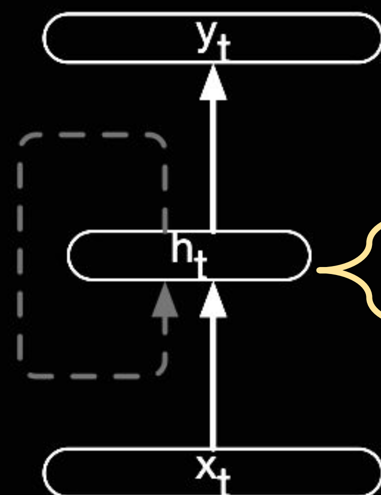
shorthand for vector/ matrix multiply

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

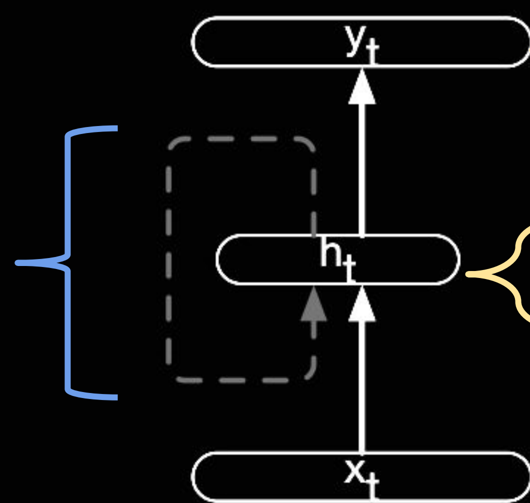
$$h_{(t)} = g(h_{(t-1)}U + x_{(t)}V)$$

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)

The Standard Recurrent Neural Network

“hidden layer”



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

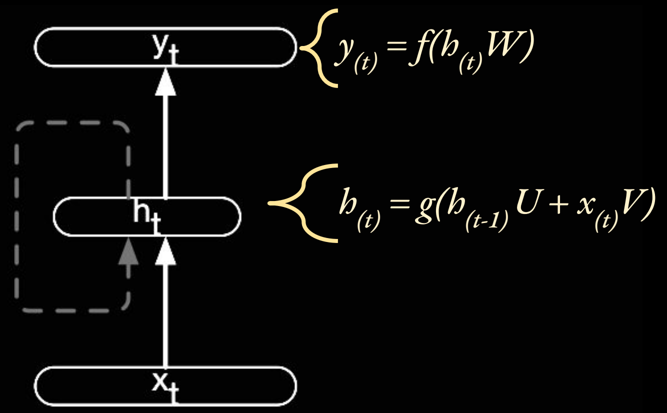
Activation Function

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

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.

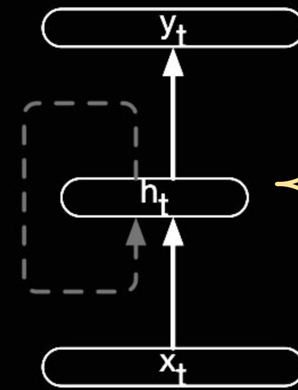
(Jurafsky, 2019)

The Standard RNN




```
#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
```

The Standard RNN

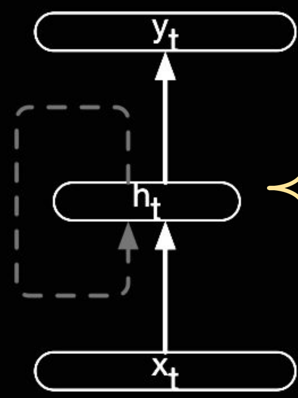


$$y_{(t)} = f(h_{(t)}W)$$

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

```
#forward pass:
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 (for classification)
```

The Standard RNN

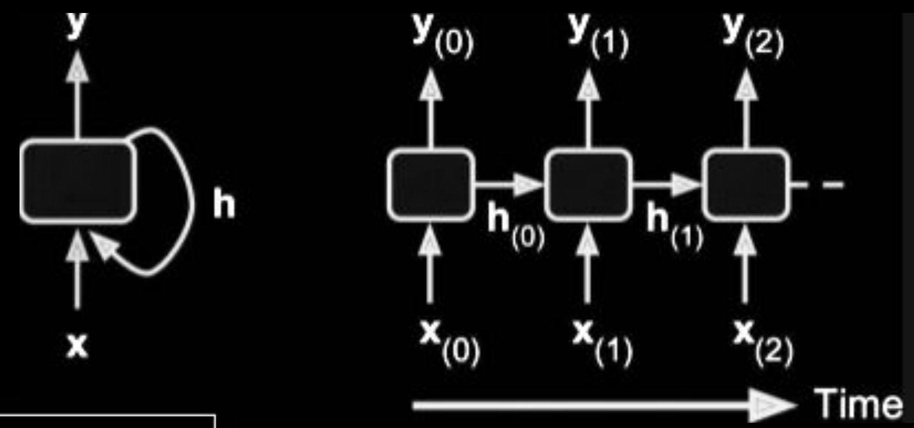


$y_{(t)} = f(h_{(t)}W)$

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

```
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))
```

Visualizing Sequences: Unrolling



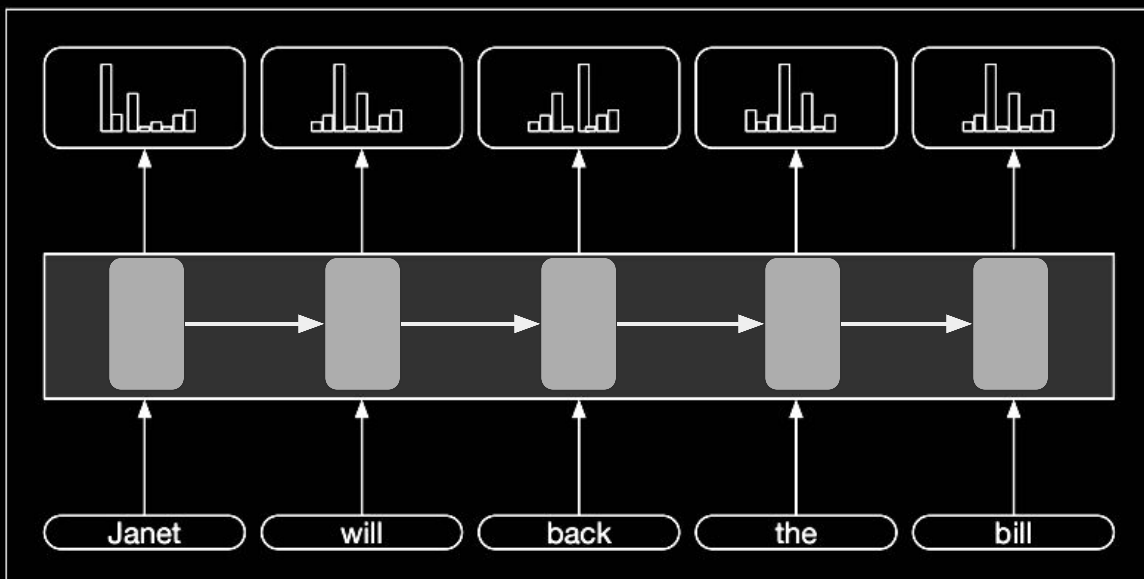
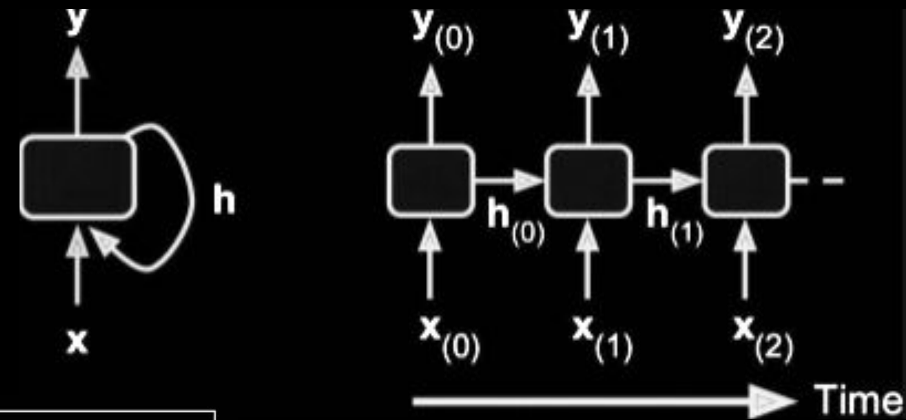


Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

Visualizing Sequences: Unrolling



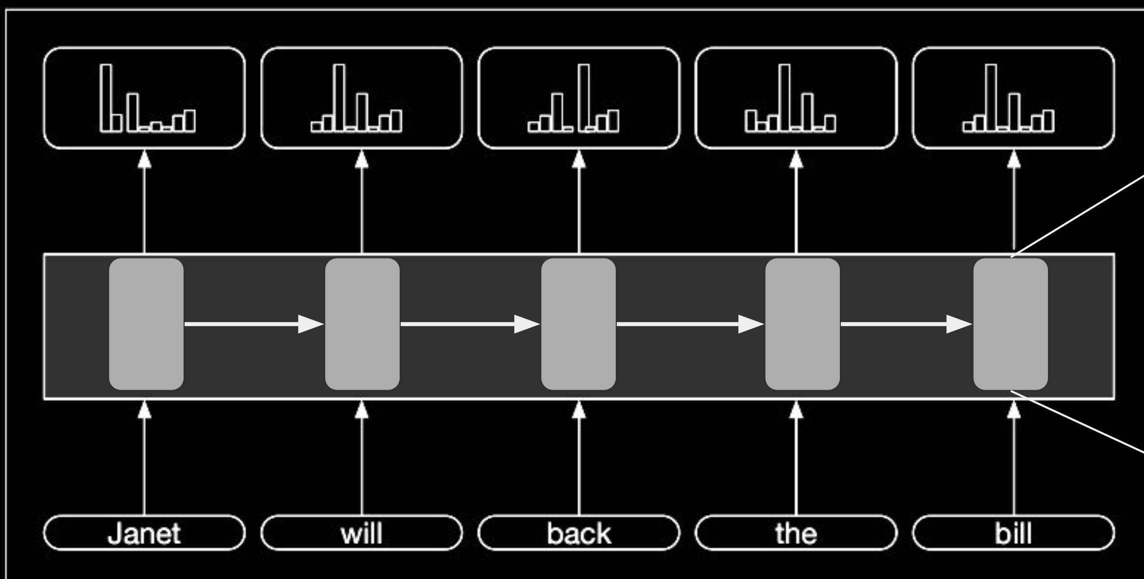


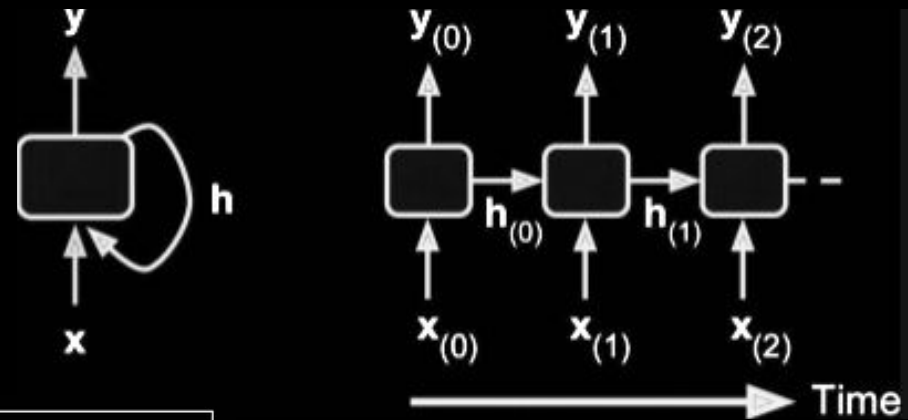
Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

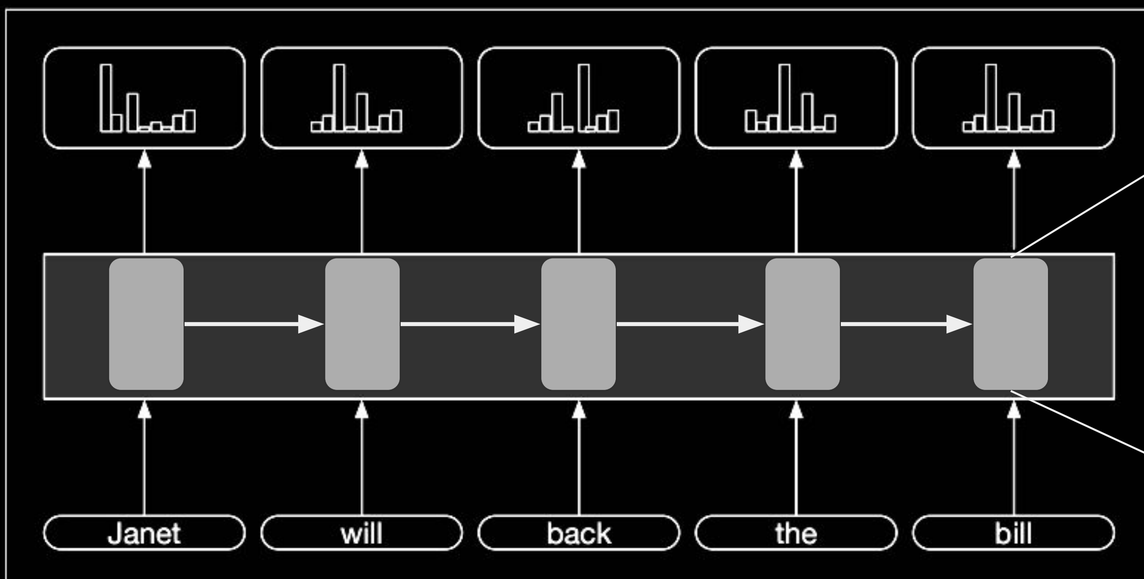
$$y_{(t)} = f(h_{(t)}W)$$

Activation Function

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

Visualizing Sequences: Unrolling





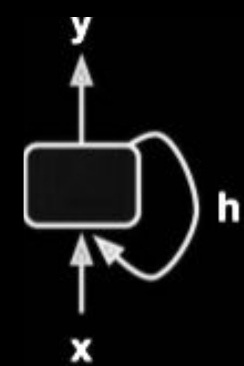
$$y_{("bill")} = f(h_{("bill")}W)$$

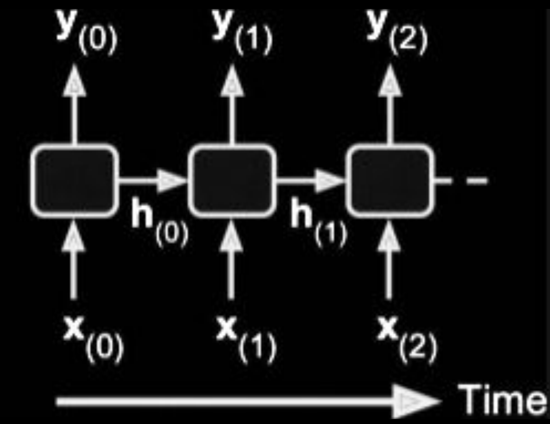
Activation Function

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

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.

Visualizing Sequences: Unrolling





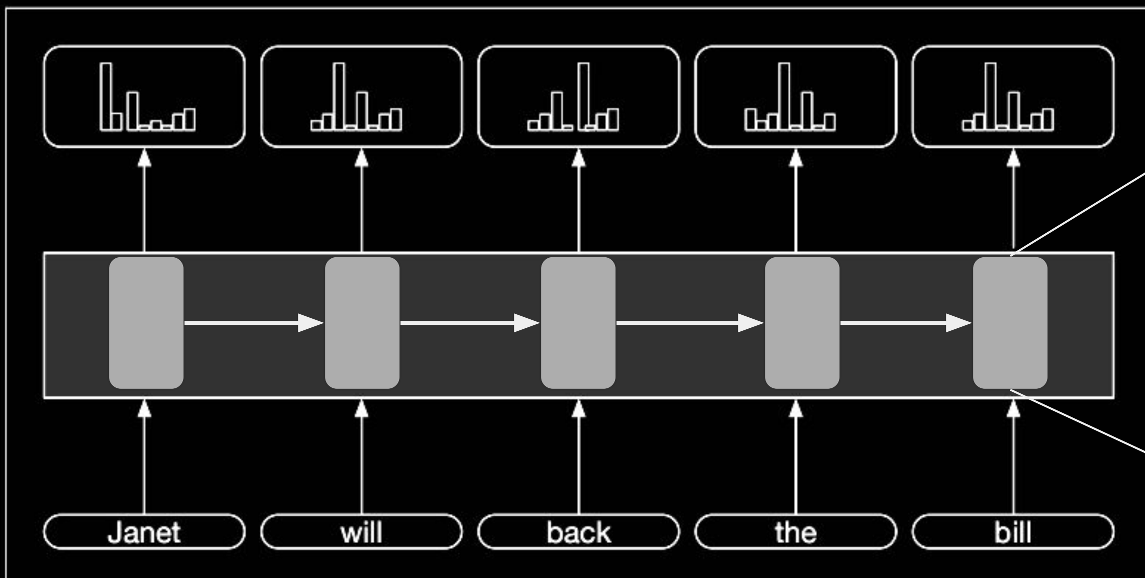
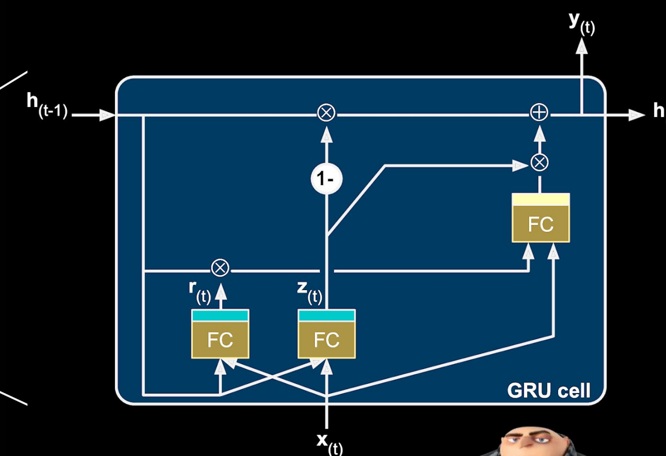


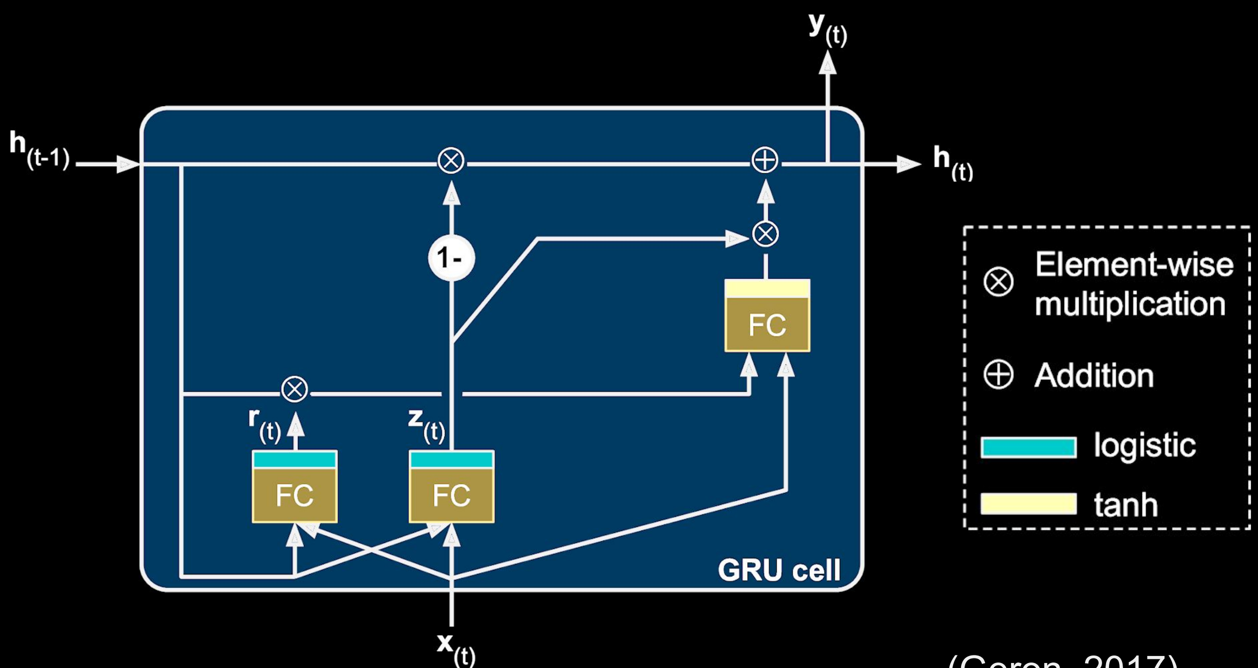
Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.



The GRU!

The GRU

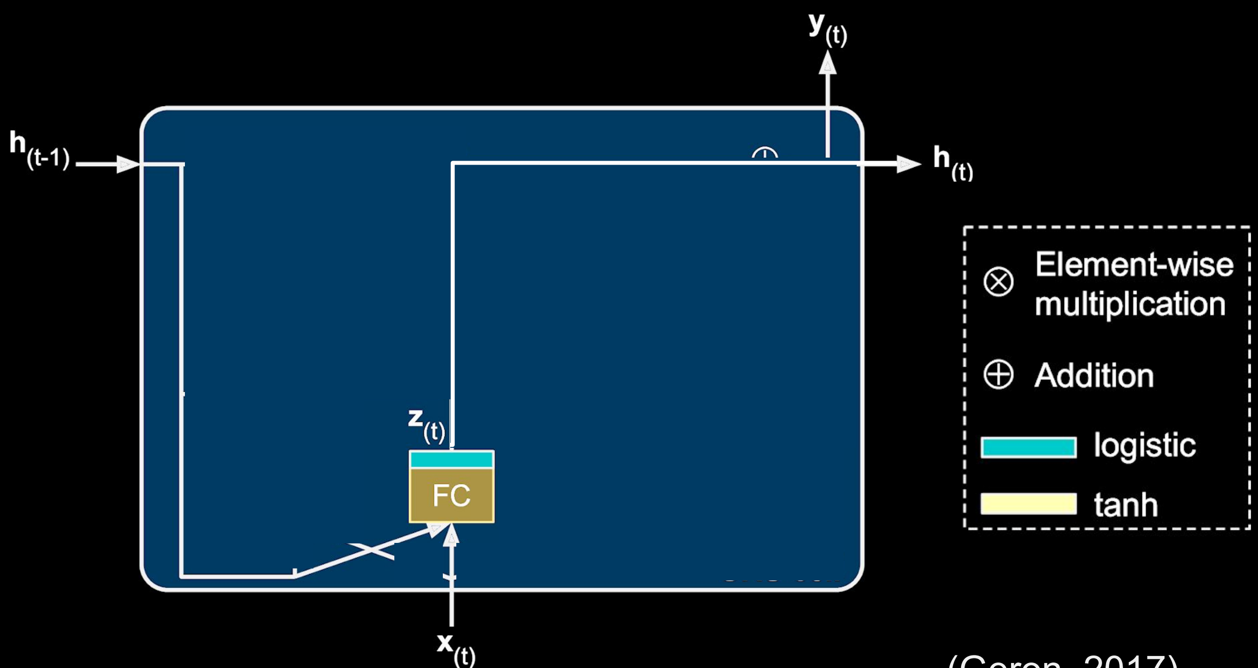
Gated Recurrent Unit



(Geron, 2017)

The GRU

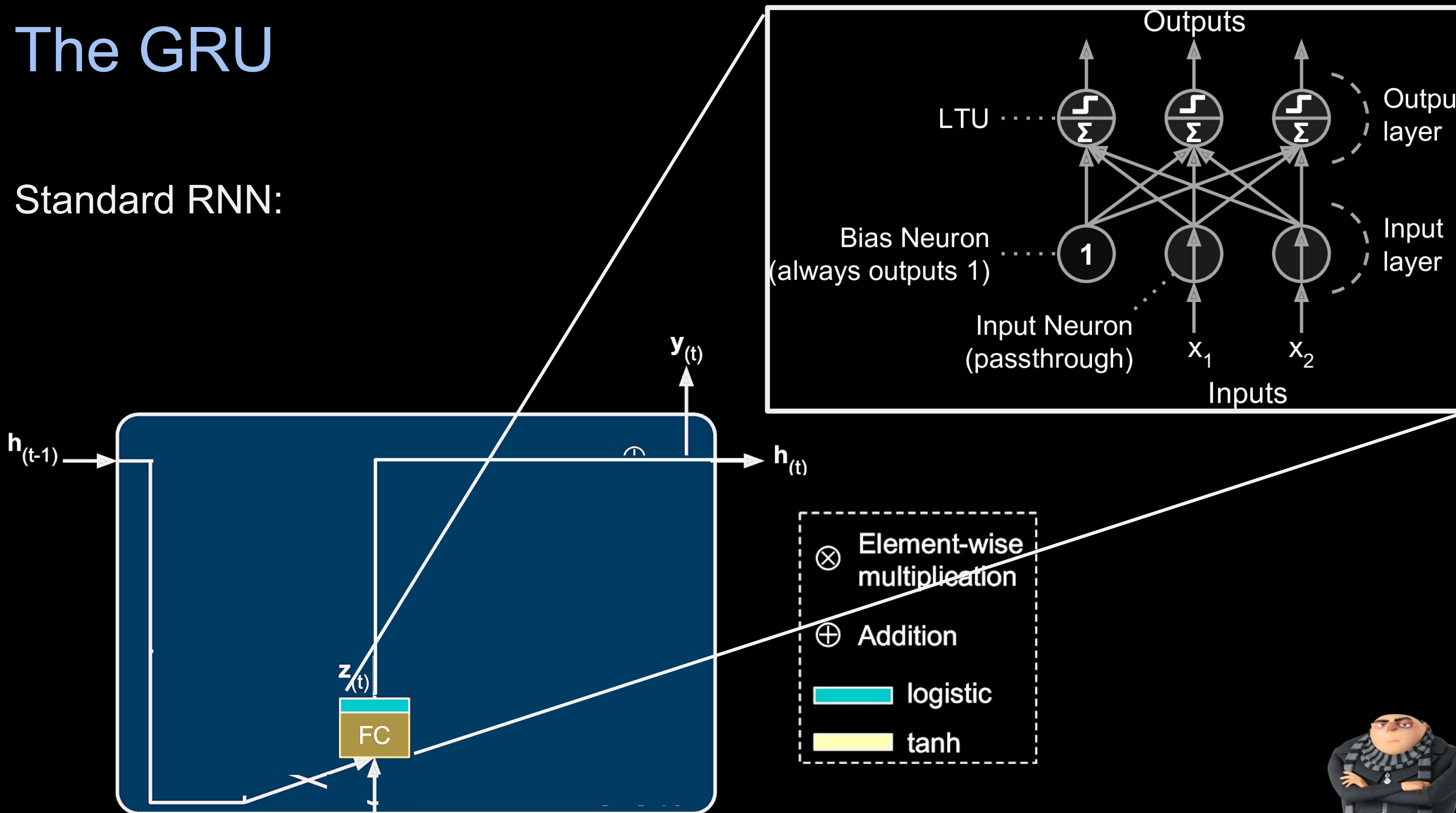
Standard RNN:



(Geron, 2017)

The GRU

Standard RNN:

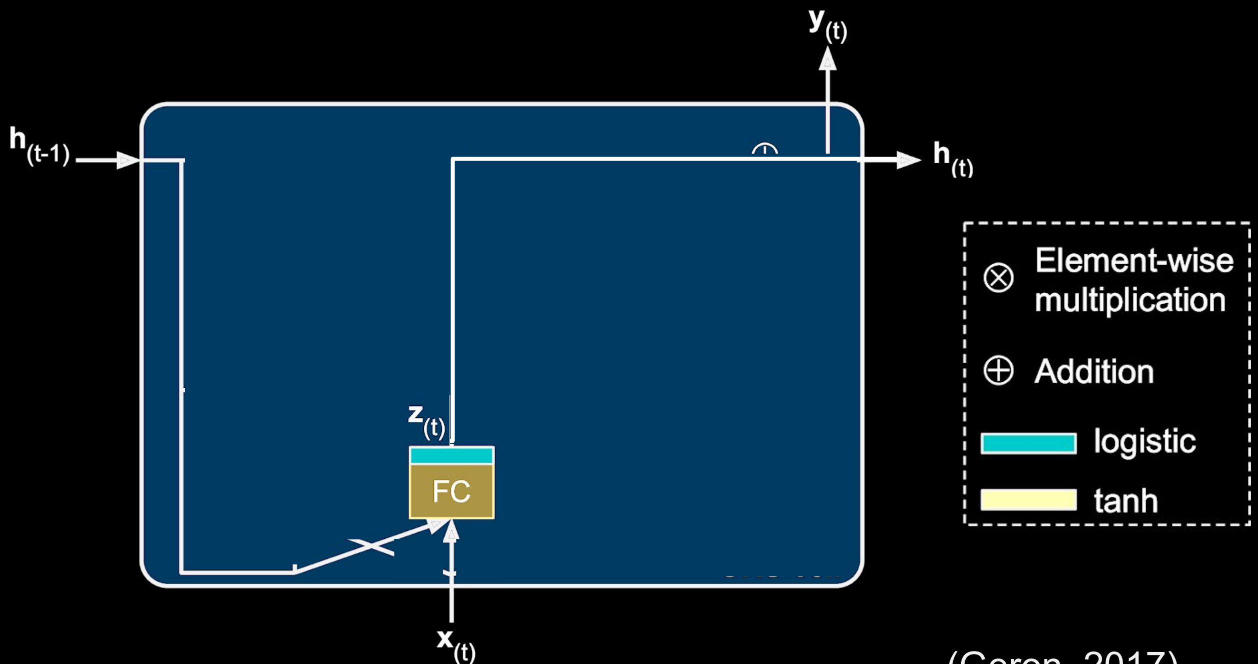


(Geron, 2017)

The GRU

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

Standard RNN:

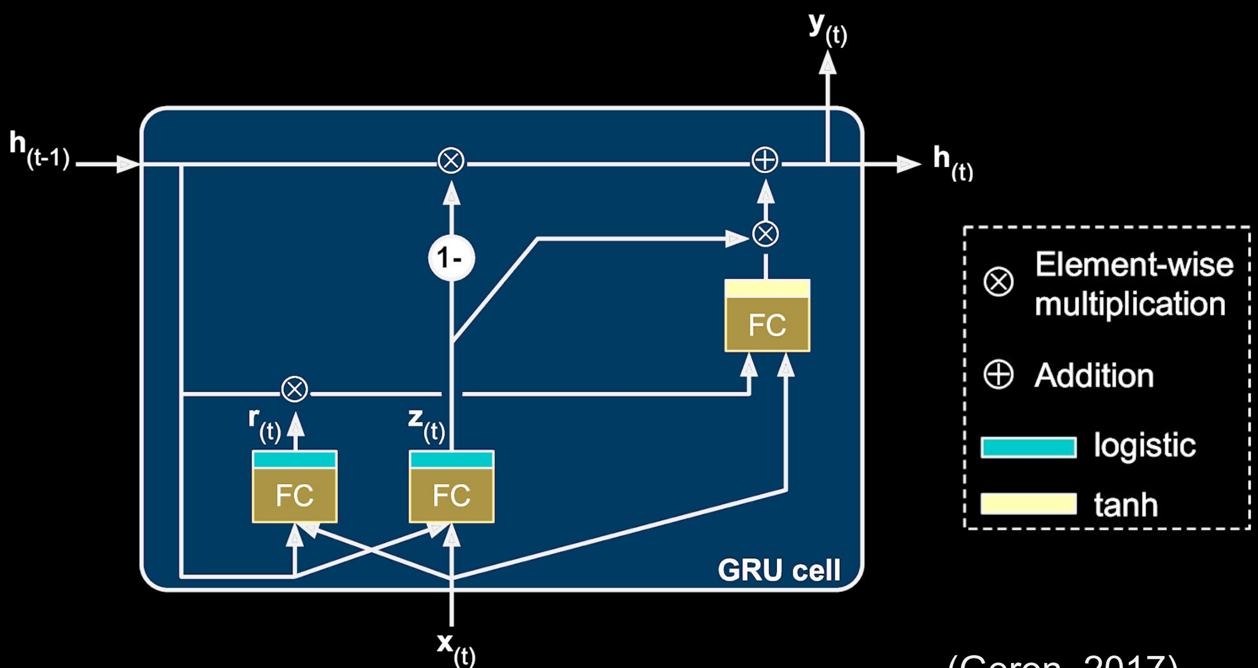


(Geron, 2017)

The GRU

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

Gated Recurrent Unit



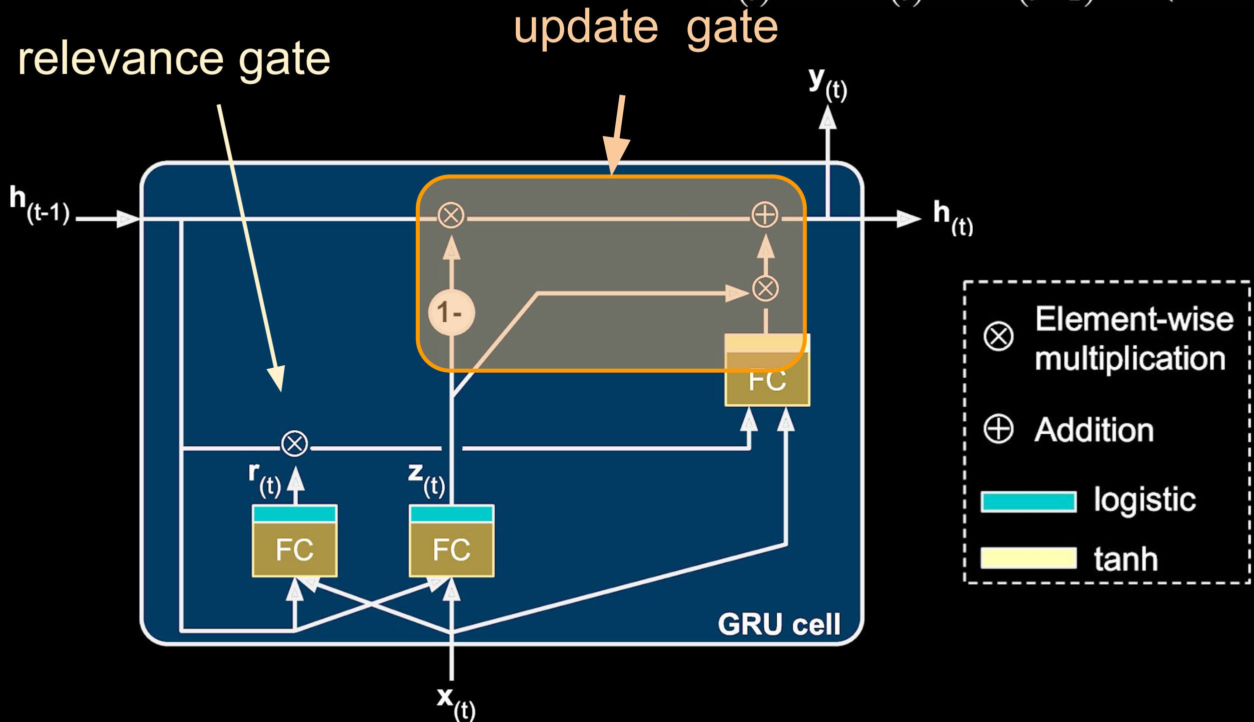
(Geron, 2017)

The GRU

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

Gated Recurrent Unit

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



The GRU

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

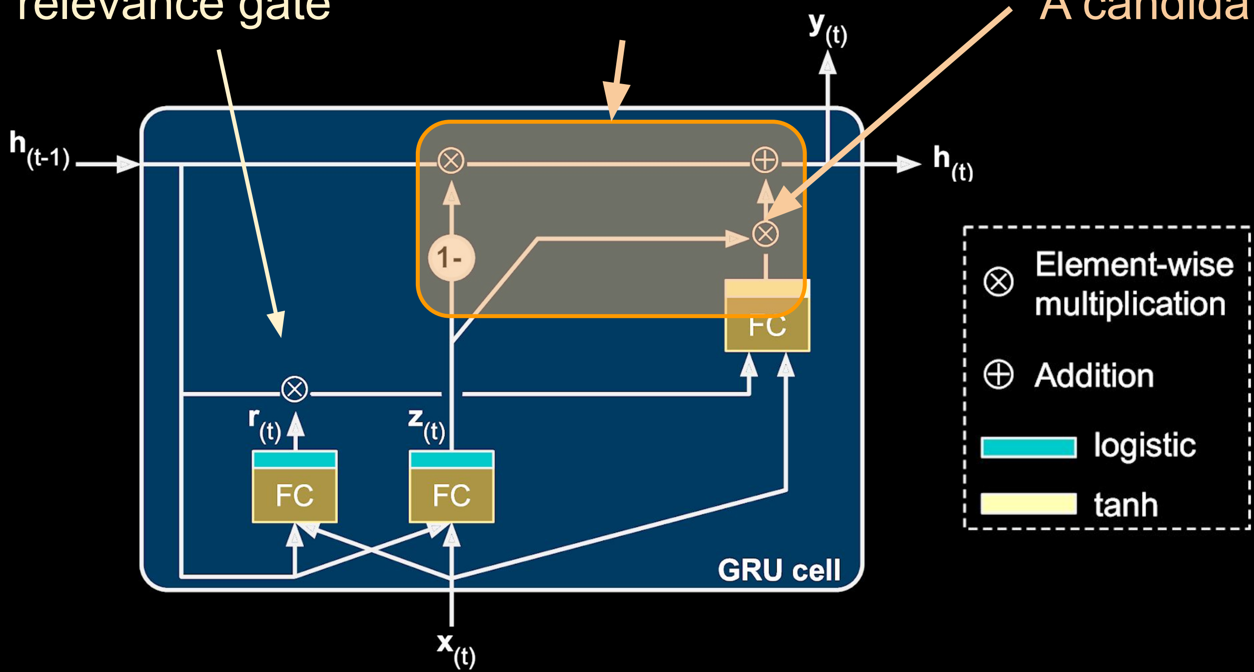
Gated Recurrent Unit

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

update gate

relevance gate

A candidate for updating h, *h~*



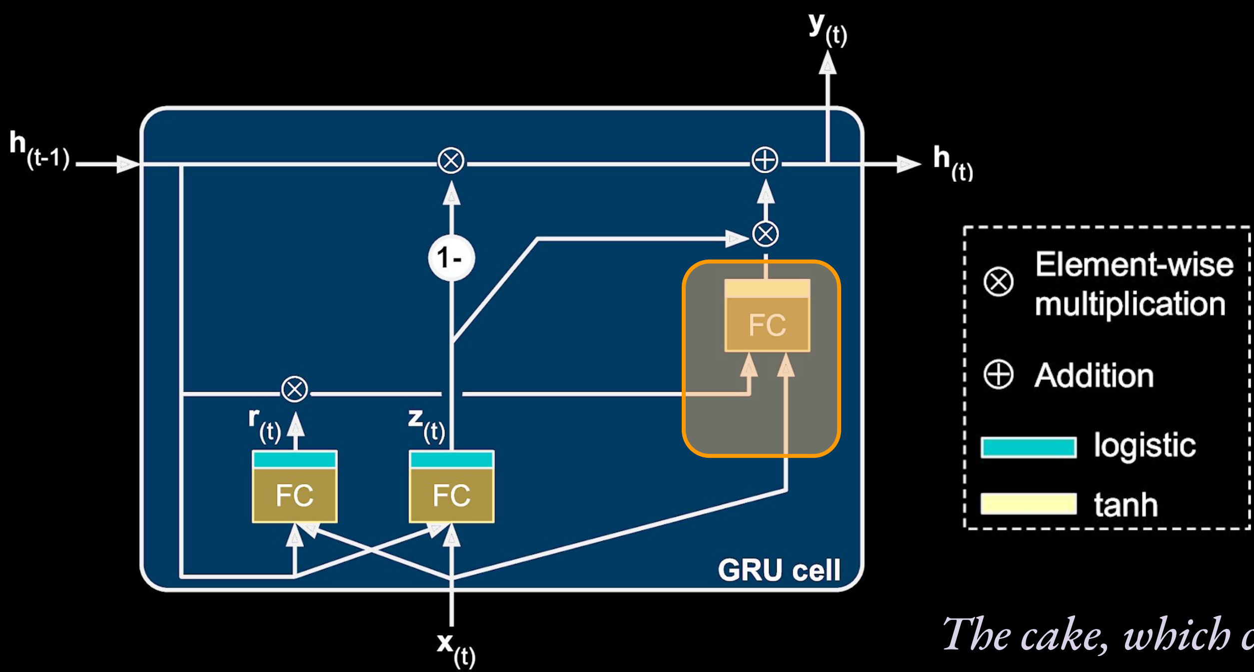
The GRU

Gated Recurrent Unit

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

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

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

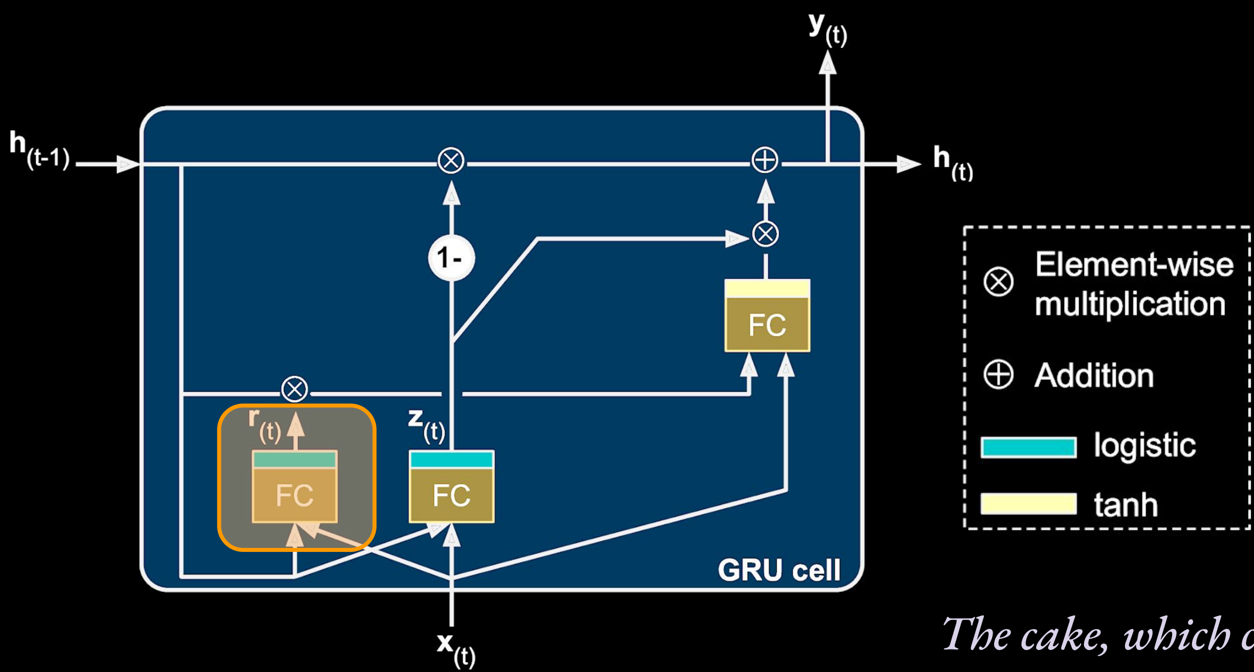


The cake, which contained candles, was eaten.

The GRU

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

Gated Recurrent Unit



The cake, which contained candles, was eaten.

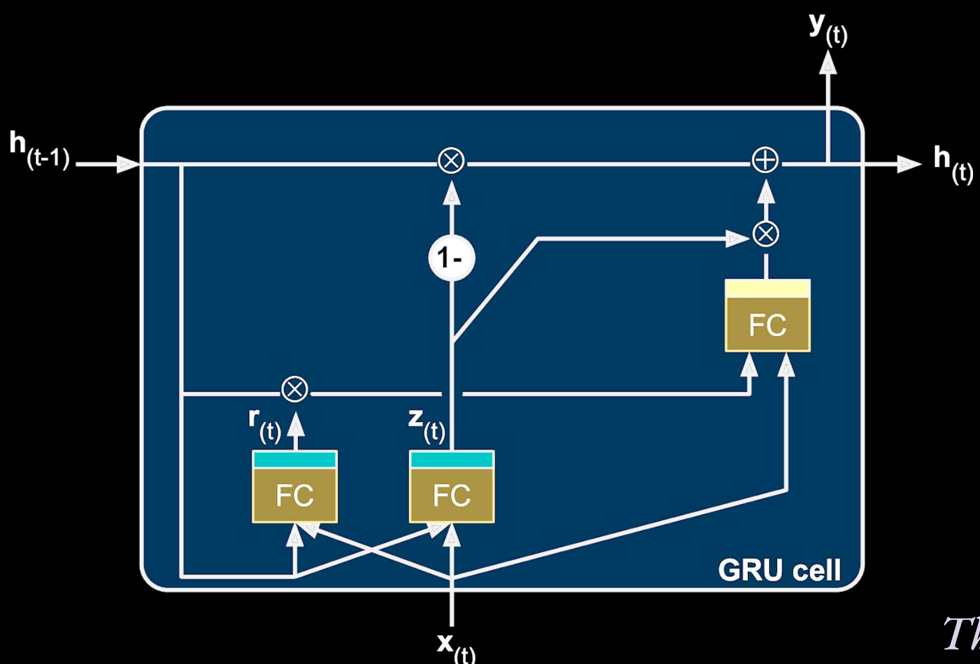
What about the gradient?

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

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

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

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



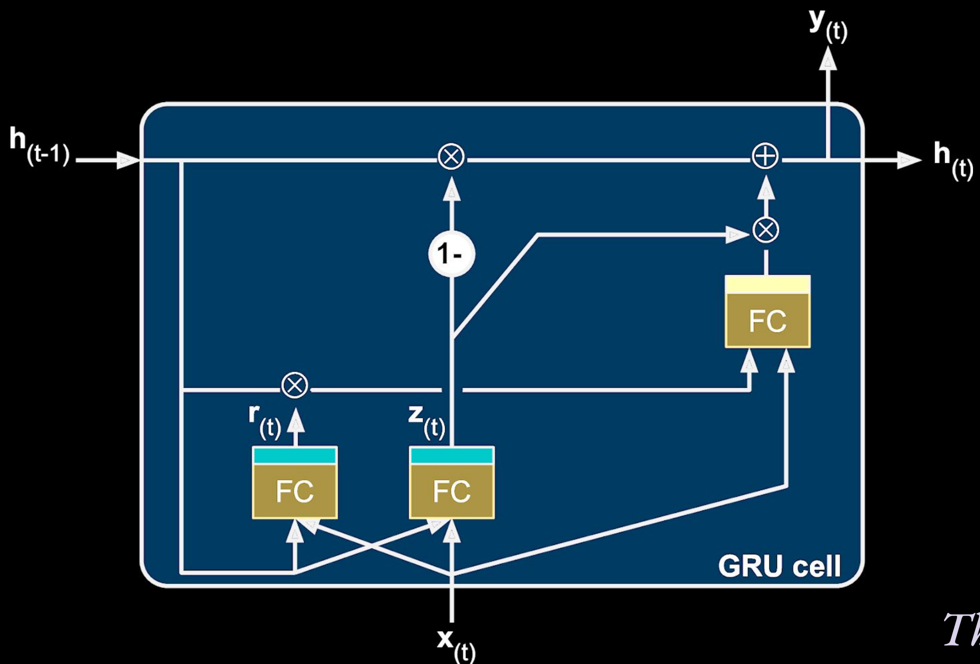
The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of h,

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

The cake, which contained candles, was eaten.

What about the gradient?

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



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,

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

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 a GRU-style RNN

```
RNN_cost = torch.mean(-torch.sum(y*torch.log(y_pred))
```

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))
```

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"

How do we optimize this network?

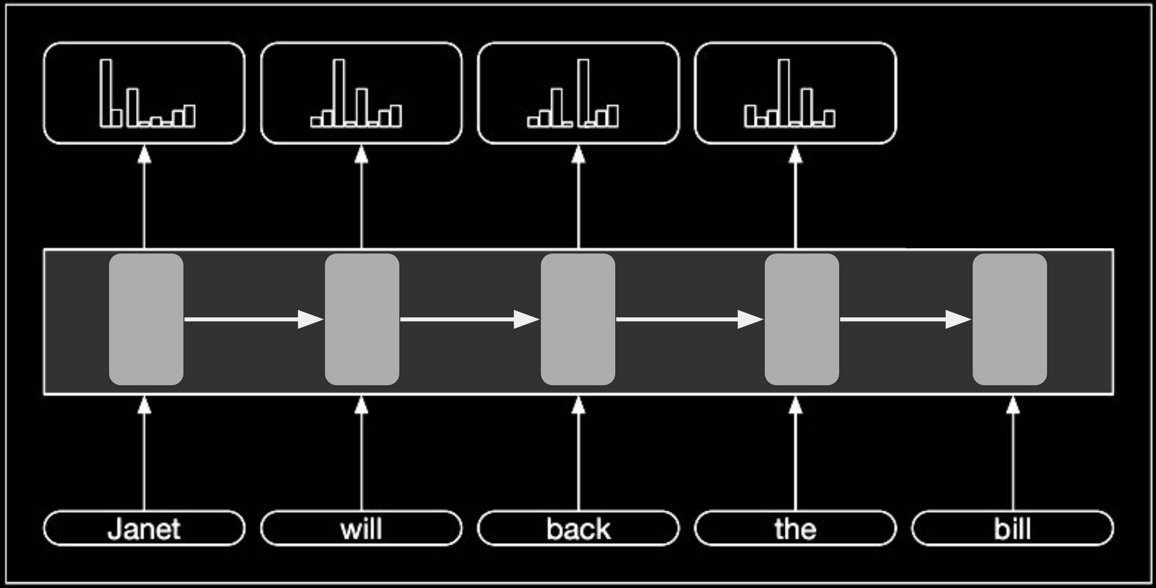
Function

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

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

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

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



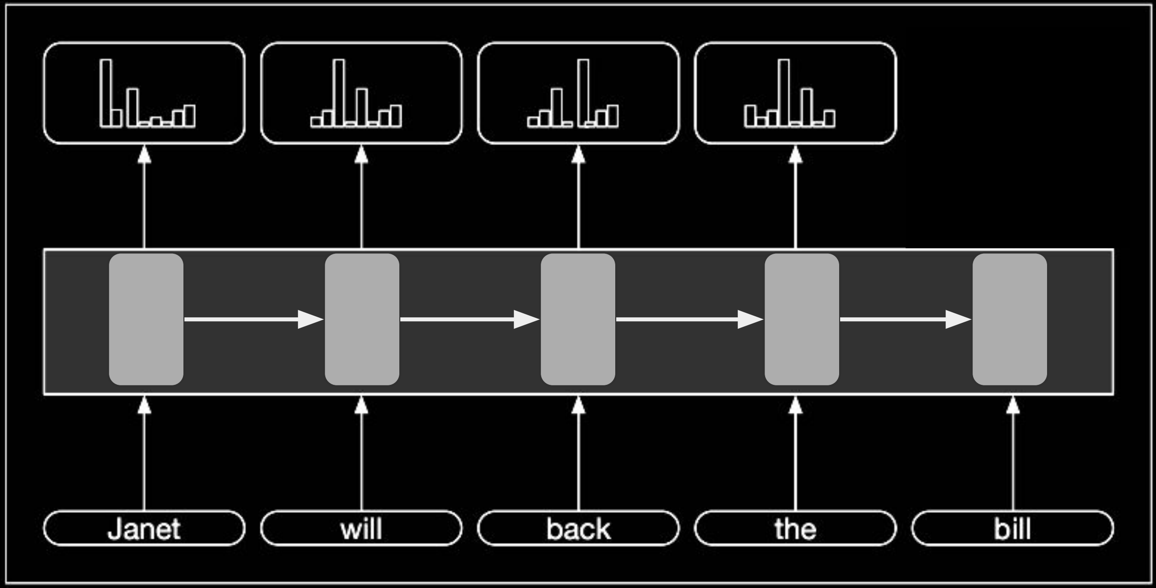
Cost Function

$$J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{|V|} y_{i,j}^{(t)} log\ \hat{y}_{i,j}^{(t)}$$

How do we optimize this network?

Answer: Gradient Descent

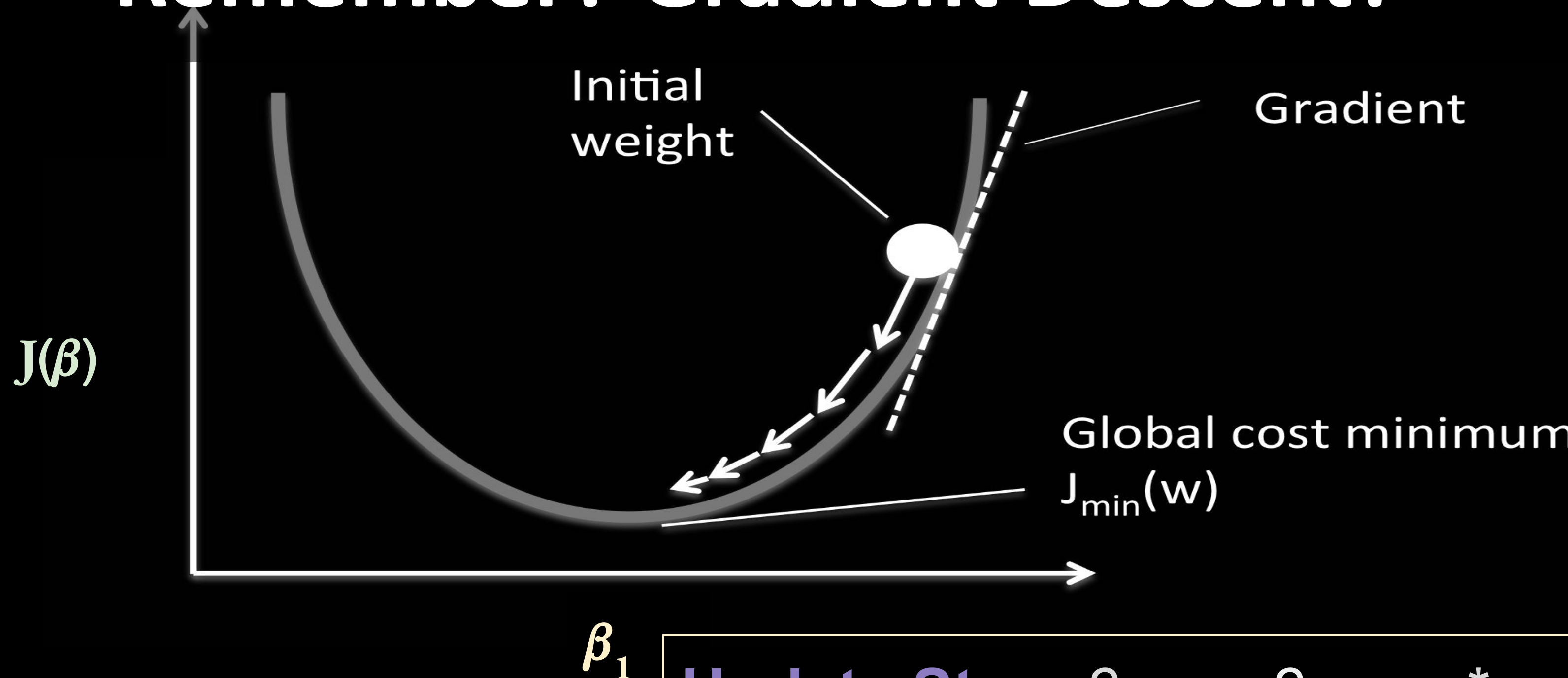
$$\mathbf{z}_{(t)} = \sigma(\mathbf{W}_{xz}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{z})$$
$$\mathbf{r}_{(t)} = \sigma(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{r})$$
$$\mathbf{g}_{(t)}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



Cost Function

$$J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{|V|} y_{i,j}^{(t)} log\ \hat{y}_{i,j}^{(t)}$$

Remember? Gradient Descent?



Update Step: $\beta_{new} = \beta_{old} - \alpha * \text{grad}$

"log loss" or *"normalized log loss"*:

$$J(\beta) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i) + (1 - y_i)\log(1 - p_i)$$

Remember? Gradient Descent?

"log loss" or *"normalized log loss":*

$$\boxed{J(\beta) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i) + (1-y_i)\log(1-p_i)}$$

Remember? Gradient Descent?

"log loss" or *"normalized log loss"*:

$$\boxed{J(\beta) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i) + (1-y_i)\log(1-p_i)}$$

$$p_i = \sigma(x_i;\, \beta) = \sigma_i$$

Remember? Gradient Descent?

"log loss" or *"normalized log loss":*

$$\boxed{J(\beta) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i) + (1 - y_i)\log(1 - p_i)}$$

$$p_i = \sigma(x_i; \beta) = \sigma_i$$

$$\frac{\partial p_i}{\partial \beta} = \sigma_i (1 - \sigma_i)$$

$$\frac{\partial J}{\partial \beta} = -\frac{1}{N}\sum_{i=1}^{N} y_i (1 - \sigma_i) + (1 - y_i)(-\sigma_i)$$

Remember? Gradient Descent?

"log loss" or *"normalized log loss"*:

$$J(\beta) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i) + (1-y_i)\log(1-p_i)$$

$$p_i = \sigma(x_i; \beta) = \sigma_i$$

$$\frac{\partial p_i}{\partial \beta} = \sigma_i (1 - \sigma_i)$$

$$\boxed{\frac{\partial J}{\partial \beta} = -\frac{1}{N}\sum_{i=1}^{N} y_i (1 - \sigma_i) + (1 - y_i)(-\sigma_i)}$$

Simple update step

Update Step:

$\beta_{new} = \beta_{old} - \alpha * \text{grad}$

How do we do Gradient Descent for this Problem?

$$J^{(t)} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{|V|} y_{i,j}^{(t)} log\ \hat{y}_{i,j}^{(t)}$$

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

How do we optimize this network?

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

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

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

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

How do we optimize this network?

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

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

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

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

Compute Gradients:

- **Hidden state gradient:**

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} + \frac{\partial L}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t}$$

where $\hat{y}_t$ is the predicted output.

- **Gradients for the update gate:**

$$\frac{\partial L}{\partial z_t} = \frac{\partial L}{\partial h_t} \odot (\tilde{h}_t - h_{t-1})$$

$$\frac{\partial L}{\partial W_z} = \sum_t \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t) \cdot x_t^T$$

$$\frac{\partial L}{\partial U_z} = \sum_t \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t) \cdot h_{t-1}^T$$

$$\frac{\partial L}{\partial b_z} = \sum_t \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t)$$

- **Gradients for the reset gate:**

$$\frac{\partial L}{\partial r_t} = \left(\frac{\partial L}{\partial \tilde{h}_t} \odot U_h h_{t-1} \right) \odot \sigma'(r_t)$$

- **Gradients for the candidate hidden state:**

$$\frac{\partial L}{\partial \tilde{h}_t} = \frac{\partial L}{\partial h_t} \odot z_t \odot (1 - \tanh^2(\tilde{h}_t))$$

- **Gradient updates for weights and biases:**

$$\frac{\partial L}{\partial W_h} = \sum_t \frac{\partial L}{\partial \tilde{h}_t} \cdot x_t^T$$

$$\frac{\partial L}{\partial U_h} = \sum_t \frac{\partial L}{\partial \tilde{h}_t} \cdot (r_t \odot h_{t-1})^T$$

$$\frac{\partial L}{\partial b_h} = \sum_t \frac{\partial L}{\partial \tilde{h}_t}$$

How do we optimize this network?

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

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

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

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

Compute Gradients:

- **Hidden state gradient:**

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} + \frac{\partial L}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t}$$

where $\hat{y}_t$ is the predicted output.

- **Gradients for the update gate:**

- **Gr**

$$\frac{\partial L}{\partial W_h} = \sum_t \frac{\partial L}{\partial \tilde{h}_t} \cdot x_t^T$$

$$\frac{\partial L}{\partial U_h} = \sum_t \frac{\partial L}{\partial \tilde{h}_t} \cdot (r_t \odot h_{t-1})^T$$

$$\frac{\partial L}{\partial b_h} = \sum_t \frac{\partial L}{\partial \tilde{h}_t}$$

GRU Implementation: PyTorch

```
class RNNPyTorch(nn.Module):
    def __init__(self, input_size:int, hidden_size:int, vocab_size:int):
        self.gru_layer = nn.GRU(input_size=input_size, hidden_size=hidden_size)
        self.lin_layer = nn.Linear(hidden_size, vocab_size)

    def forward(self, input_rep:torch.Tensor):
        output_rep, hidden_rep = self.gru_layer(input_rep)
        output_logits = self.lin_layer(output_rep)
        return output_logits
```

How do we update the Network?

How do we update the Network?

- Update using the:

  loss over the entire train set

  (GD)

GD (aka Batch GD)

```
for epoch in range(num_epochs): # Pass X_tr, y_tr

        # Forward pass: pass the input through the model

        output = model(X_tr)

        # Compute the loss
        loss = criterion(output, y_tr, reduction="mean")

        # Zero out gradients
        optimizer.zero_grad()

        # Backward pass: compute gradients
        loss.backward()

        # Update parameters
        optimizer.step()
```

How do we update the Network?

- Update using the:

  loss over the entire train set

  (GD)

  Accurate gradient
  estimates,
  but impractical as the
  network size grows.

How do we update the Network?

- Update using the:

loss over the entire train set

(GD)

Accurate gradient estimates, but impractical as the network size grows.

loss from each sample

(SGD)

SGD

```
for epoch in range(num_epochs): # Iterate over X_tr, y_tr
    for x_sample, y_sample in zip(X_tr, y_tr):
        # Forward pass: pass the input through the model

        output = model(x_sample)

        # Compute the loss
        loss = criterion(output, y_sample)

        # Zero out gradients
        optimizer.zero_grad()

        # Backward pass: compute gradients
        loss.backward()

        # Update parameters
        optimizer.step()
```

SGD

```
for epoch in range(num_epochs): # Iterate over X_tr, y_tr
    for x_sample, y_sample in zip(X_tr, y_tr):
        # Forward pass: pass the input through the model

        output = model(x_sample)

        # Compute the loss
        loss = criterion(output, y_sample)

        # Zero out gradients
        optimizer.zero_grad()

        # Backward pass: compute gradients
        loss.backward()

        # Update parameters
        optimizer.step()
```

How do we update the Network?

- Update using the:

loss over the entire train set

(GD)

Accurate gradient estimates,
but impractical as the network size grows.

loss from each sample

(SGD)

Easy to implement,
but the gradients could be very noisy leading to suboptimal results

mini-Batch GD

```
for epoch in range(num_epochs): # Iterate over batches of X_tr, y_tr
    for idx in range(0, len(X_tr), batch_size):
        # Forward pass: pass the input through the model

        X_batch, y_batch = X_tr[i:i+batch_size], y_tr[i:i+batch_size]
        output = model(X_tr)

        # Compute the loss
        loss = criterion(output, y_tr, reduction="mean")

        # Zero out gradients
        optimizer.zero_grad()

        # Backward pass: compute gradients
        loss.backward()

        # Update parameters
        optimizer.step()
```

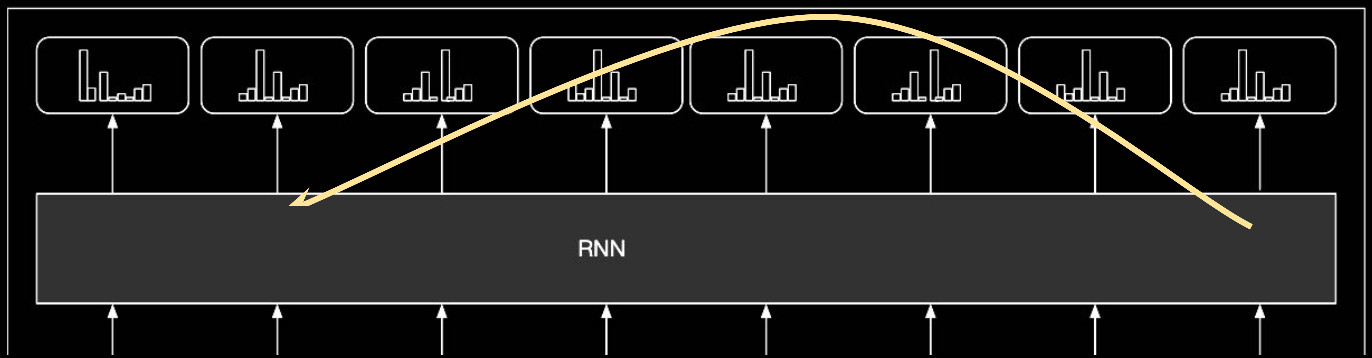
mini-Batch GD

```
for epoch in range(num_epochs): # Iterate over batches of X_tr, y_tr
    for idx in range(0, len(X_tr), batch_size):
        # Forward pass: pass the input through the model

        X_batch, y_batch = X_tr[i:i+batch_size], y_tr[i:i+batch_size]
         output = model(X_tr)

        # Compute the loss
        loss = criterion(output, y_tr)

        # Zero out gradients
        optimizer.zero_grad()

        # Backward pass: compute gradients
        loss.backward()

        # Update parameters
        optimizer.step()
```

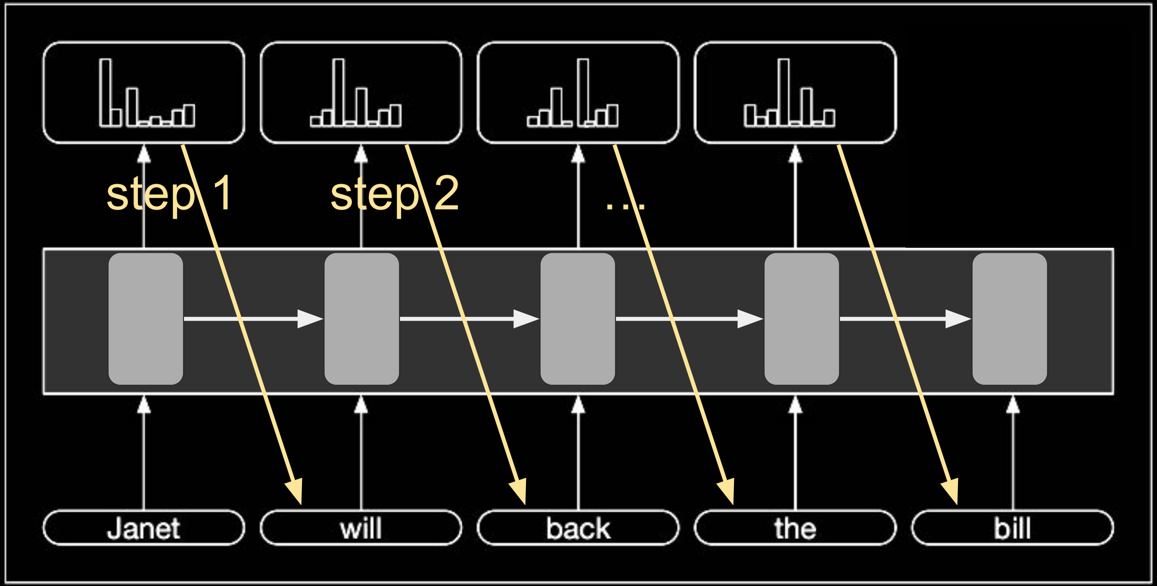
Effective use of hardware, while minimizing the variance of gradient updates.
Offers faster optimization while ensuring optimal solution

RNN Limitation:
Losing Track of Long Distance Dependencies

The horse which was raced past the barn tripped .



RNN



Language modeling with an RNN

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

Recap: RNN Limitations

- Difficult to capture long-distance dependencies
- Not parallelizable -- need sequential processing.
  - Slow computation for long sequences
- Vanishing or exploding gradients

How to use an LM for Generation

- Greedy Search
- Beam Search
- Random Walk

How to use an LM for Generation

- **Greedy Search**
- Beam Search
- Random Walk

Always take the most probable next word:

$$\hat{w}_t = \text{argmax}_{w \in V} P(w | \mathbf{w}_{<t})$$

```
def generateGreedy(model, history=['<s>']):
    vocabProbs = model.getNextProbs(history)
    history += argmax(vocabProbs)
                #word with maximum prob
    if history[-1] == '</s>': return history
    else: return generateGreedy(model, history)
```

How to use an LM for Generation

- **Greedy Search**
- Beam Search
- Random W

Always take the most probable next word:

Problem:

p('<s> ok ok </s>')
=.28

p('<s> yes yes </s>'
=.20

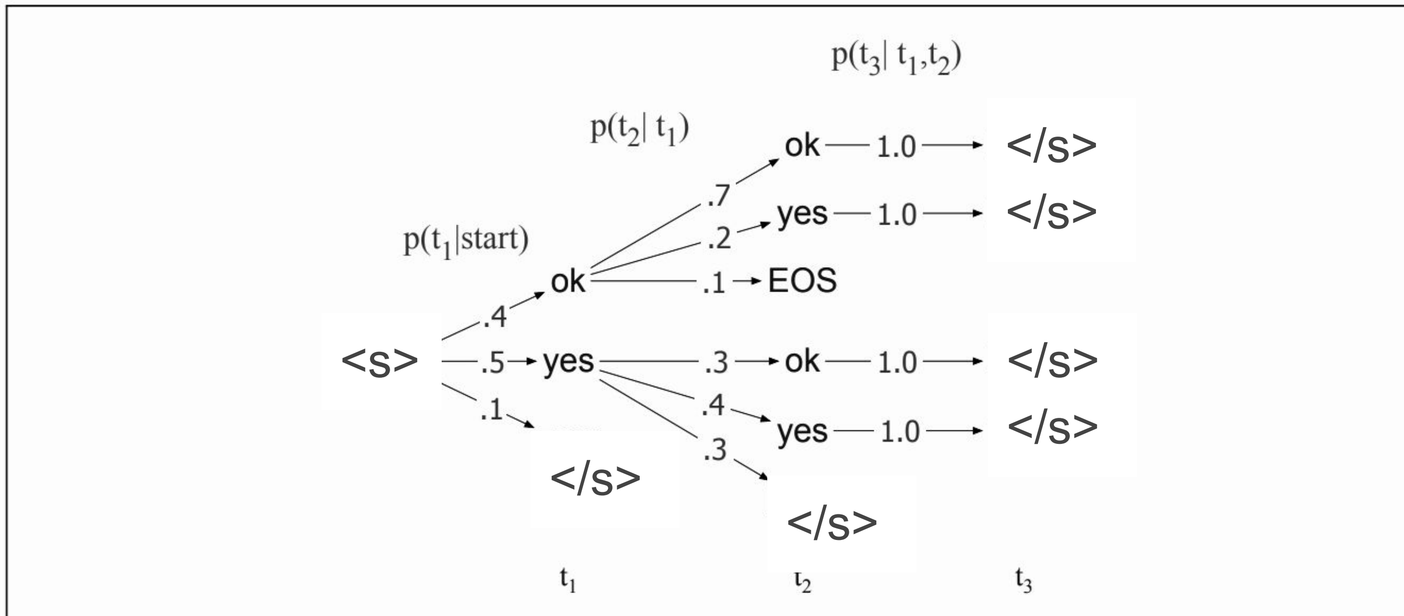


Figure 13.7 A search tree for generating the target string $T = t_1, t_2, \ldots$ from vocabulary $V = \{\text{yes}, \text{ok}, \text{<s>}\}$, showing the probability of generating each token from that state. Greedy search chooses *yes* followed by *yes*, instead of the globally most probable sequence *ok ok*.

How to use an LM for Gene

- **Greedy Search**
- Beam Search
- Random Walk

Always take th

Disadvantage: Focuses on the most probable, which is the most typical. Results in very "average sounding" utterances.

$$\hat{w}_t = \text{argm} \ldots_{\in V} P(w|\mathbf{w}_{<t})$$

```
def generateGreedy(model, history=['<s>']):
    vocabProbs = model.getNextProbs(history)
    history += argmax(vocabProbs)
                  #word with maximum prob
    if history[-1] == '</s>': return history
    else: return generateGreedy(model, history)
```

How to use an LM for Generation

- Greedy Search
- **Beam Search**
- Random Walk

Evaluate among multiple sequences.

Restrict to consider the top k (*beam width*) most probable per step.

```
def generateBeam(model, history=['<s>'], init_prob=1, k=4):
  frontier = [(history, init_prob)]
  max_path = []
  max_path_p = -1.0
  while path, path_p in frontier:
    if path[-1] == "</s>": #current max
      if path_p > max_path_p:
        max_path = path
        map_path_p = path_p
    else:
      vocabProbs = model.getNextProbs(path)
      nextWPs = topK(vocabProbs, k)
      for w, p in nextWPs.items():
        frontier.append((path+w, path_p*p))
  return max_path, max_path_p
```

e an LM for Generation

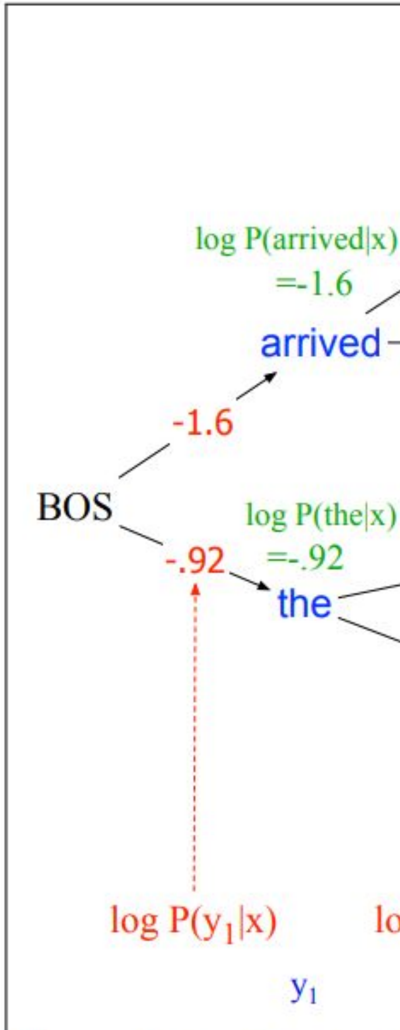


Figure 13.9 Scoring f
of each hypothesis in the
k paths are extended to t

ch
h
k
es.
er
able

```
def generateBeam(model, history='<s>', init_prob=1, k=4):
  frontier = [(history, init_prob)]
  max_path = []
  max_path_p = -1.0
    while path, path_p in frontier:
      if path[-1] == "</s>": #current max
         if path_p > max_path_p:
             max_path = path
             map_path_p = path_p
       else:
         vocabProbs = model.getNextProbs(path)
         nextWPs = topK(vocabProbs, k)
         for w, p in nextWPs.items():
             frontier.append((s+w, path_p*p))
  return max_path, max_path_p
```

Generation

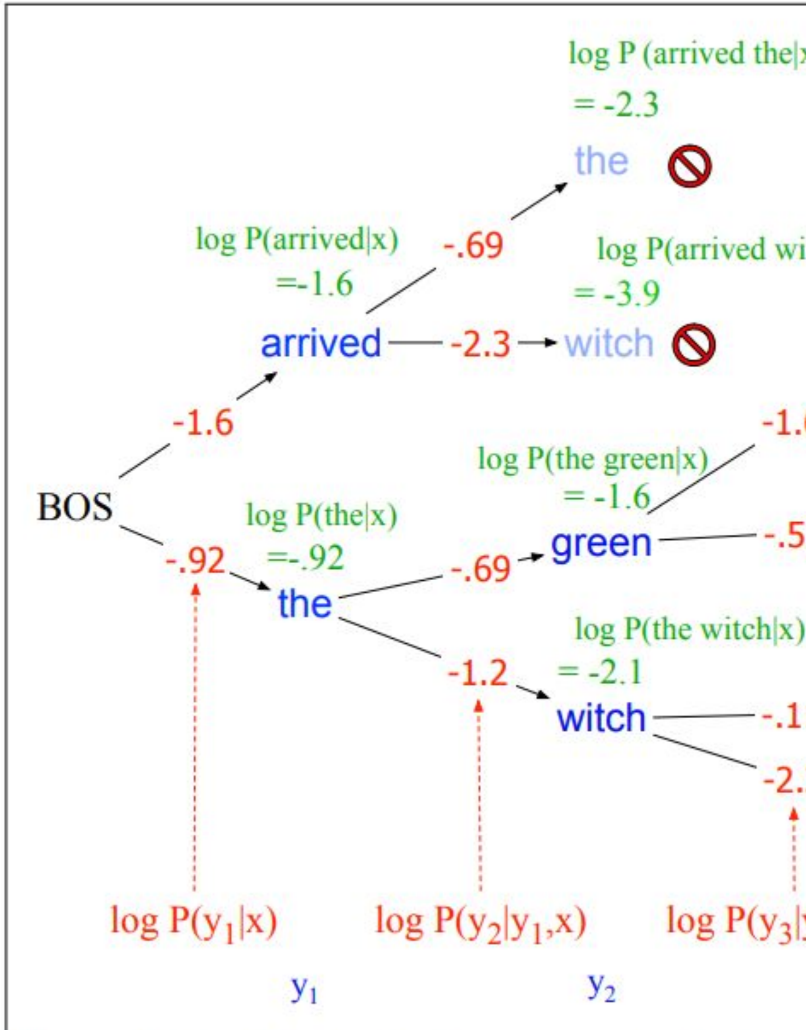


Figure 13.9 Scoring for beam search decoding
of each hypothesis in the beam by incrementally a
k paths are extended to the next step.

[redacted]tion

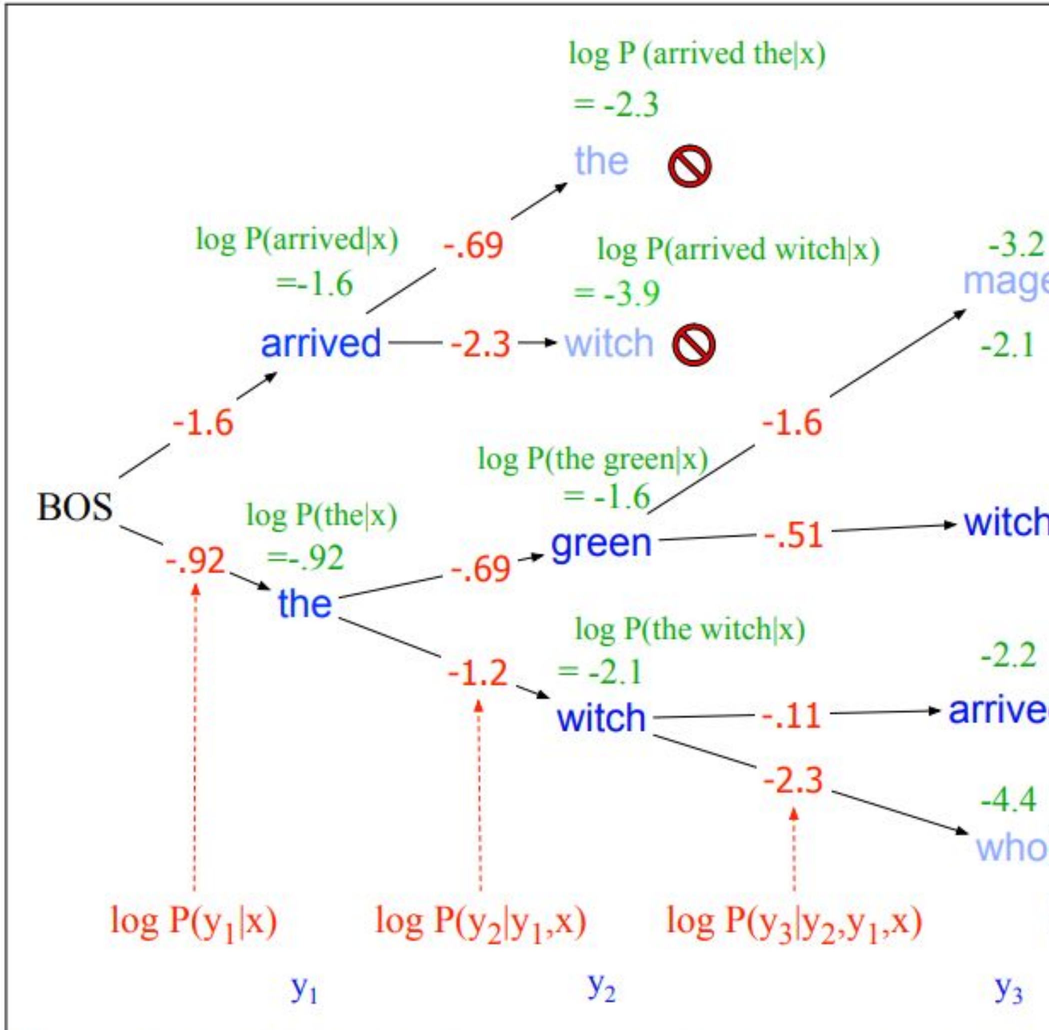


Figure 13.9 Scoring for beam search decoding with a beam w[redacted] of each hypothesis in the beam by incrementally adding the logpr[redacted] *k* paths are extended to the next step.

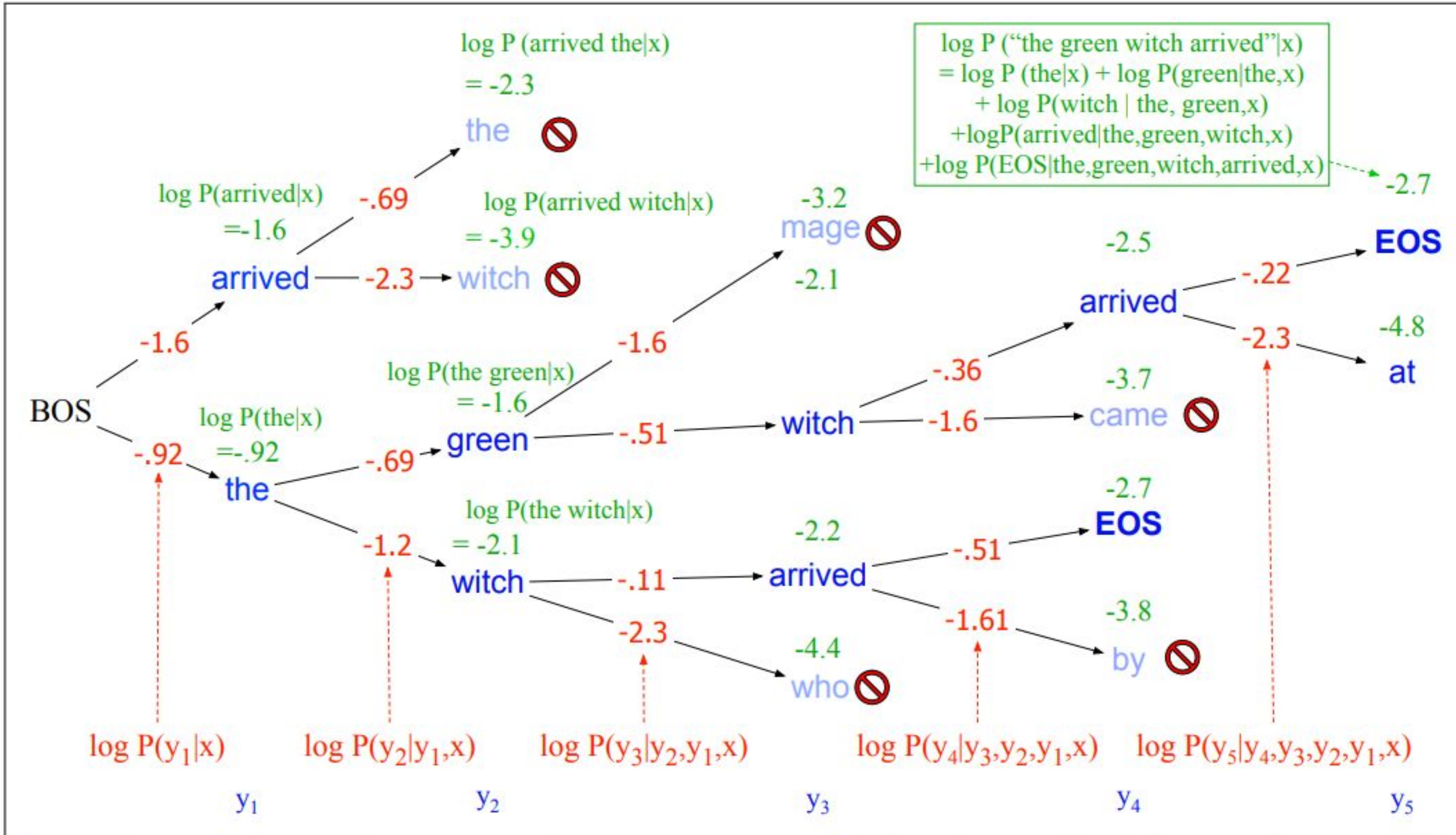
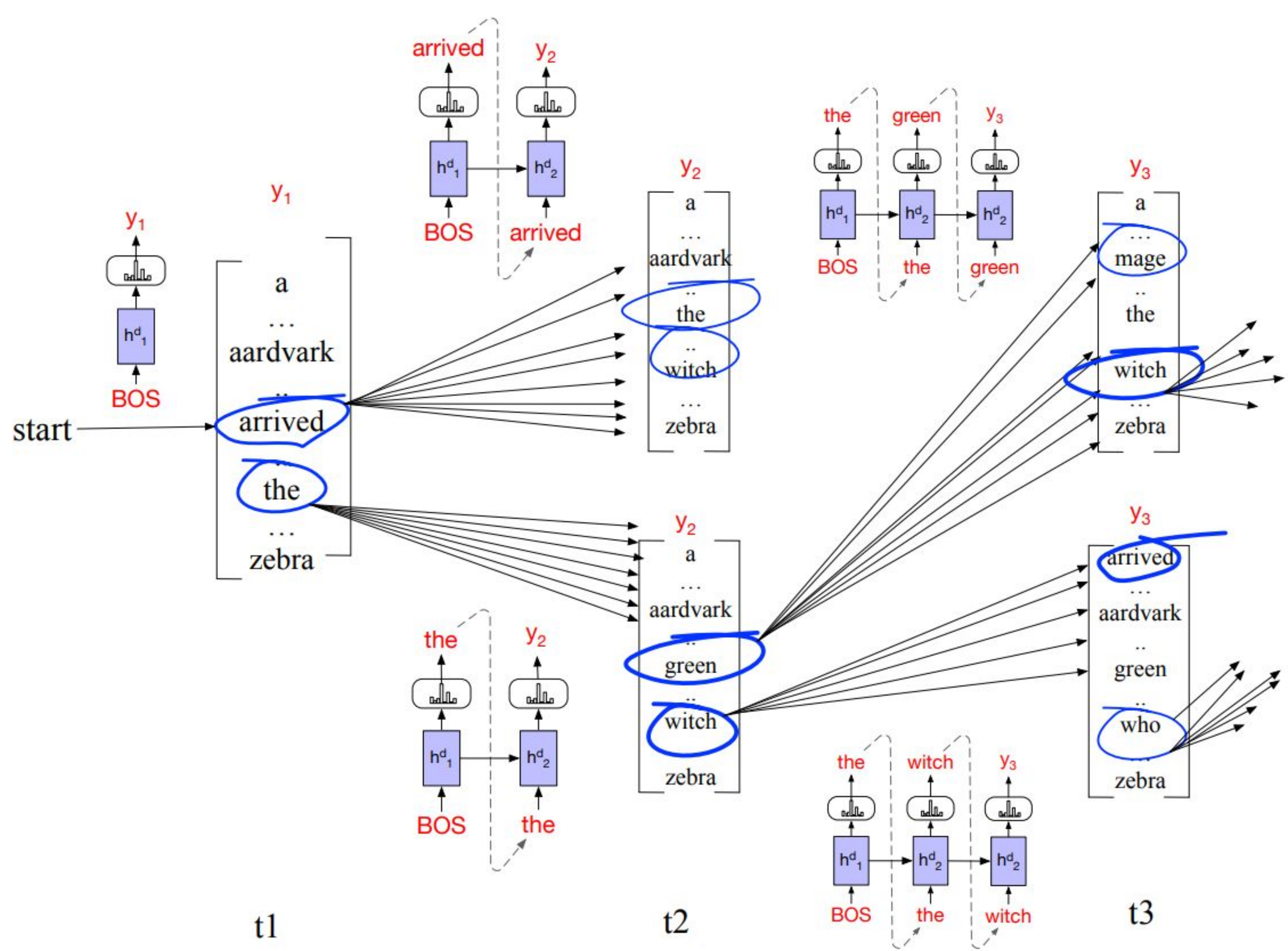


Figure 13.9 Scoring for beam search decoding with a beam width of $k = 2$. We maintain the log probability of each hypothesis in the beam by incrementally adding the logprob of generating each next token. Only the top k paths are extended to the next step.



start
BOS
y_1
h^d_1
a
aardvark
arrived
the
zebra
arrived
y_2
BOS
arrived
h^d_2
the
y_2
BOS
the
y_2
witch
green
the
green
y_3
the
green
y_3
mage
witch
arrived
who
the
witch
y_3
t1
t2
t3

How to use an LM for Generation

- Greedy Search
- **Beam Search**
- Random Walk

Evaluate among multiple sequences.

Restrict to consider the top k (*beam width*) most probable per step.

```
def generateBeam(model, history='<s>', init_prob=1, k=4):
  frontier = [(history, init_prob)]
  max_path = []
  max_path_p = -1.0
    while path, path_p in frontier:
      if path[-1] == "</s>": #current potential end
         if path_p > max_path_p:
             max_path = path
             map_path_p = path_p
      else:
         vocabProbs = model.getNextProbs(path)
         nextWPs = topK(vocabProbs, k)
         for w, p in nextWPs.items():
             frontier.append((s+w, path_p*p))
  return max_path, max_path_p
```

How to use an LM for Generation

- Greedy Search
- **Beam Search**
- Random Walk

Evaluate among multiple sequences.

Restrict to consider the top k (*beam width*) most probable per step.

```
def generateBeam(model, history='<s>', init_prob=1, k=4):
  frontier = [(history, init_prob)]
  max_path = []
  max_path_p = -1.0
    while path, path_p in frontier:
      if path[-1] == "</s>": #current potential end
         if path_p > max_path_p:
             max_path = path
             map_path_p = path_p
      else:
         vocabProbs = model.getNextProbs(path)
         nextWPs = topK(vocabProbs, k)
         for w, p in nextWPs.items():
             frontier.append((s+w, path_p*p))
  return max_path, max_path_p
```

How to use an LM for Generation

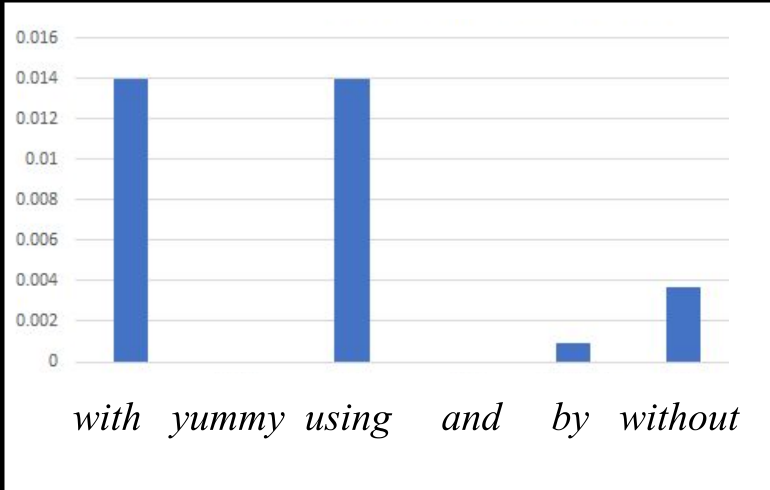
- Greedy Search
- Beam Search
- **Random Walk**

```
def generateRandWalk(model, history=['<s>']):
    vocabProbs = model.getNextProbs(history)
    history += multinomial.draw(vocabProbs)
               #random multinomial draw by probs
    if history[-1] == '</s>': return history
    else: return generateRandWalk(model, history)
```

Task: Estimate $P(w_i \mid w_1,\ldots w_{i-1})$

:P(masked word given history)

P(with | He ate the cake <M>) = ?



How to use an LM for Generation

- Greedy Search
- Beam Search
- **Random Walk**

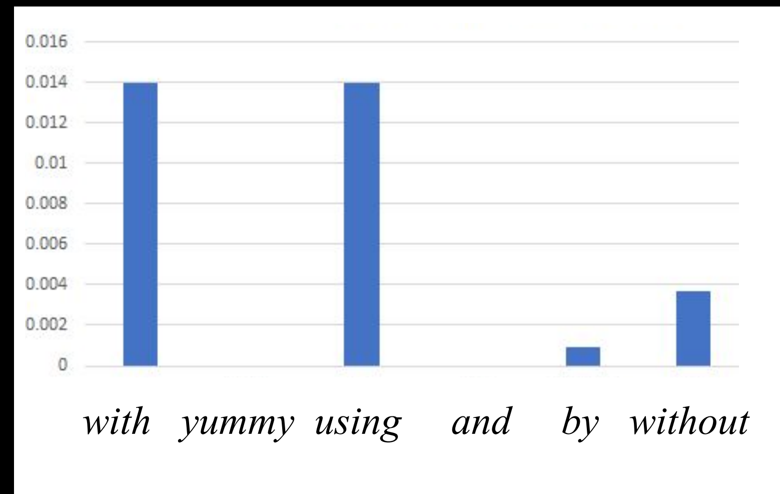
```
def generateRandWalk(model, history=['<s>']):
    vocabProbs = model.getNextProbs(history)
    history += multinomial.draw(vocabProbs)
               #random multinomial draw by probs
    if history[-1] == '</s>': return history
    else: return generateRandWalk(model, history)
```

Easiest for somewhat realistic generation; most true (occasionally picks low prob)

Task: Estimate $P(w_i \mid w_1, \ldots w_{i-1})$

:P(masked word given history)

P(with | He ate the cake <M>) = ?



How to use an LM for Generation

Practical Points

- Use log probs for faster computation tracking maximums.

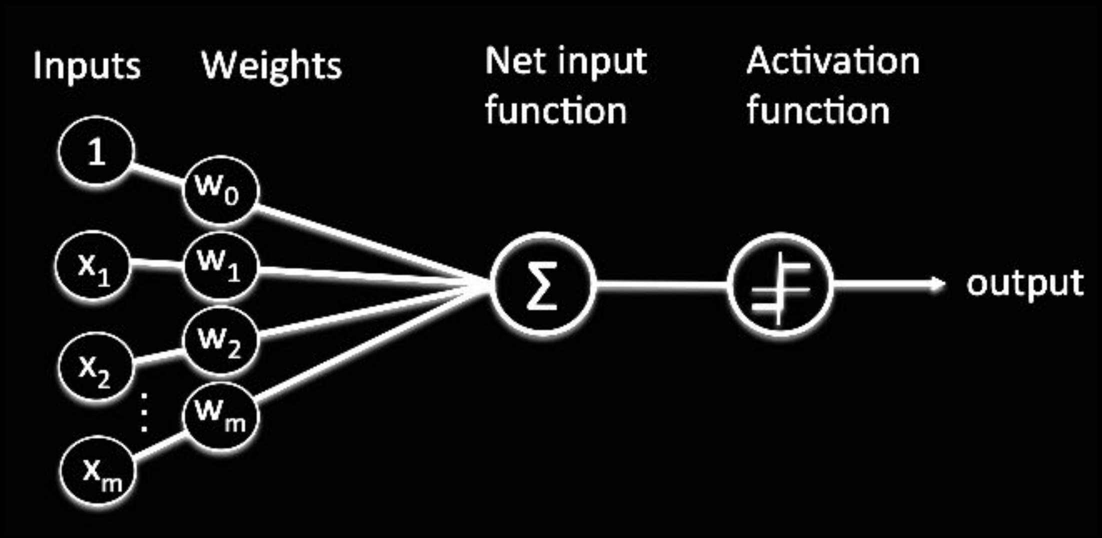
- Can normalize by length to not favor shorter sequences:

$$score(y) = \log P(y|x) \;=\; \frac{1}{t}\sum_{i=1}^{t} \log P(y_i|y_1,\ldots,y_{i-1},x) \qquad (13.16)$$

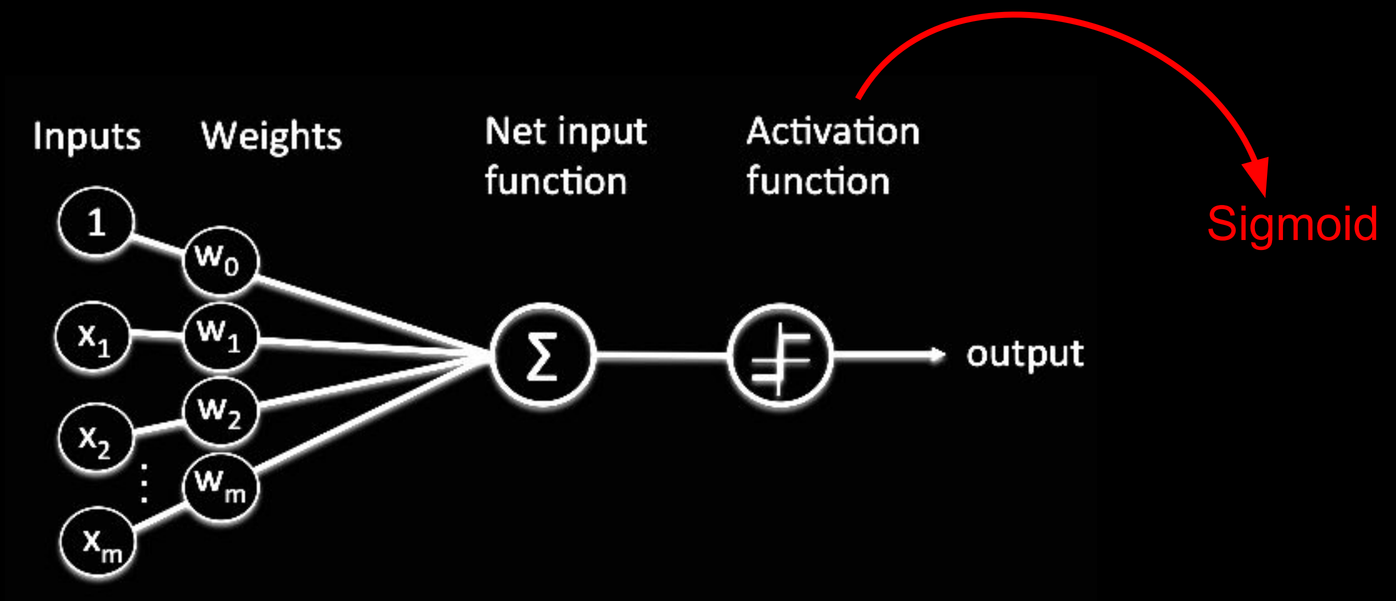
- Combine beam and random walk for more novelty.

How do we Optimize Neural Networks?

How do we Optimize Neural Networks?



How do we Optimize Neural Networks?



How do we Optimize Neural Networks?

"log loss" or *"normalized log loss"*:

$$J(\beta) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i) + (1 - y_i)\log(1 - p_i)$$

$$p_i = \sigma(x_i; \beta) = \sigma_i$$

$$\frac{\partial p_i}{\partial \beta} = \sigma_i (1 - \sigma_i)$$

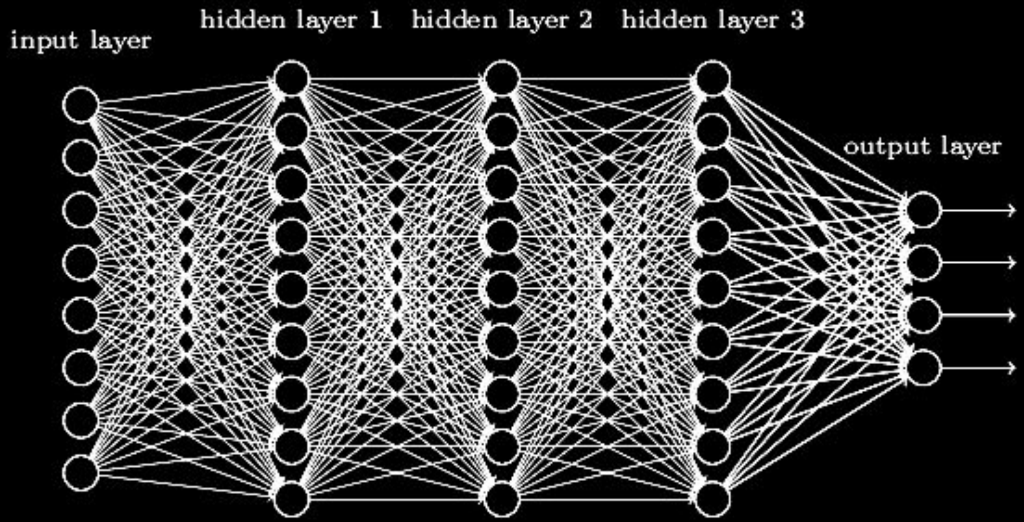
$$\boxed{\frac{\partial J}{\partial \beta} = -\frac{1}{N}\sum_{i=1}^{N} y_i (1 - \sigma_i) + (1 - y_i)(-\sigma_i)}$$

Simple update step

Update Step:

$\beta_{new} = \beta_{old} - \boldsymbol{\alpha} * \text{grad}$

How do we Optimize Complex Neural Networks?



How do we Optimize Complex Neural Networks?

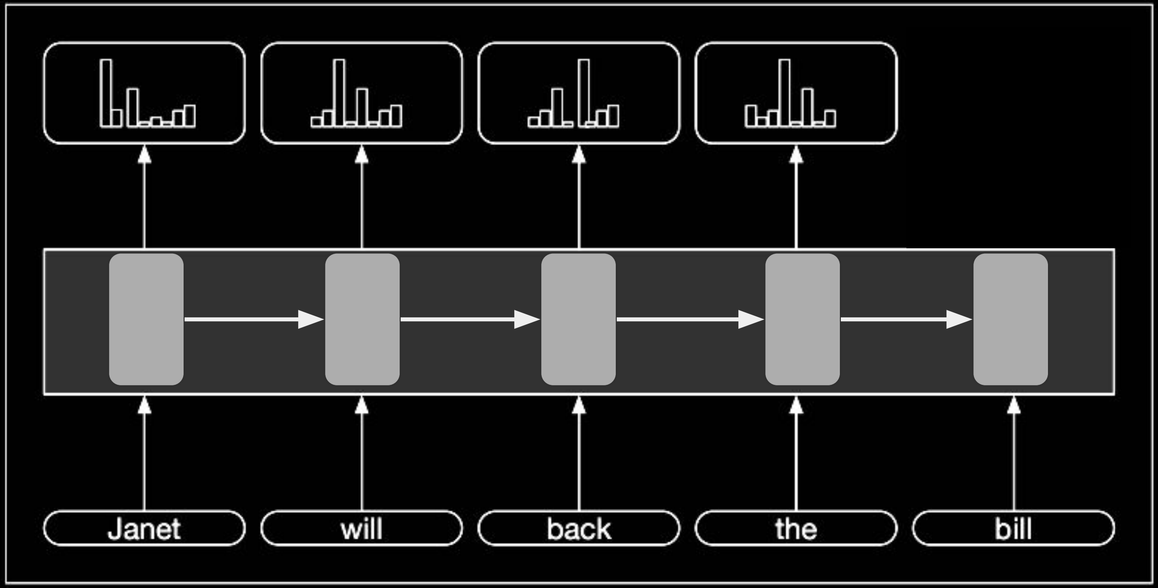
Function

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

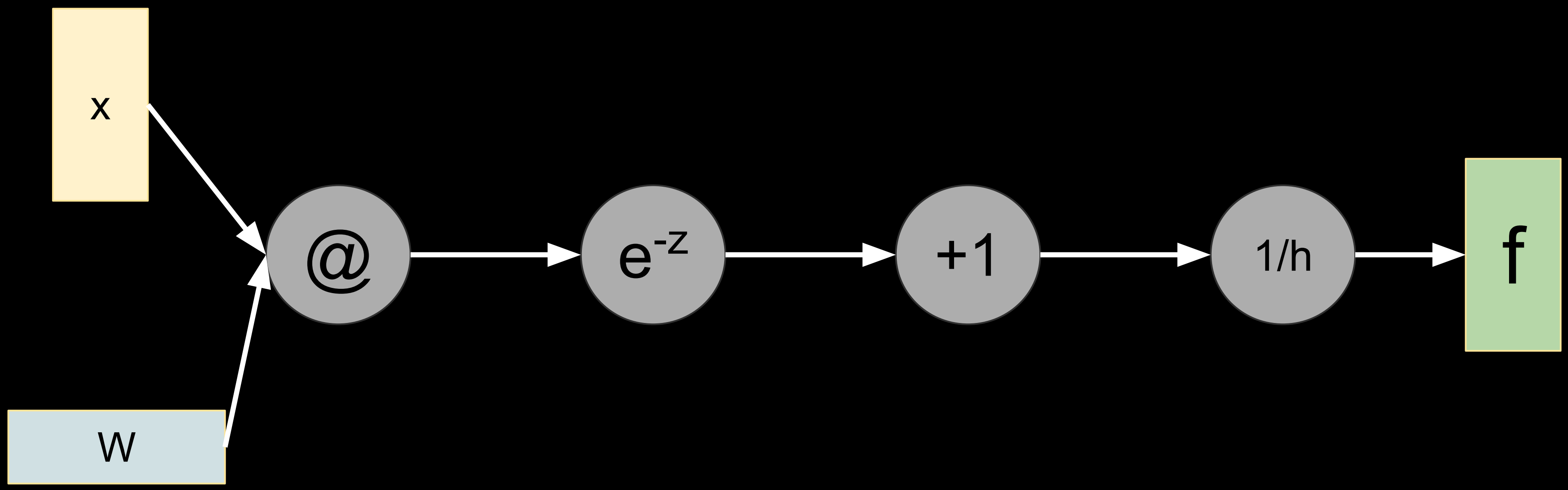
$$\mathbf{r}_{(t)} = \sigma(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r)$$

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

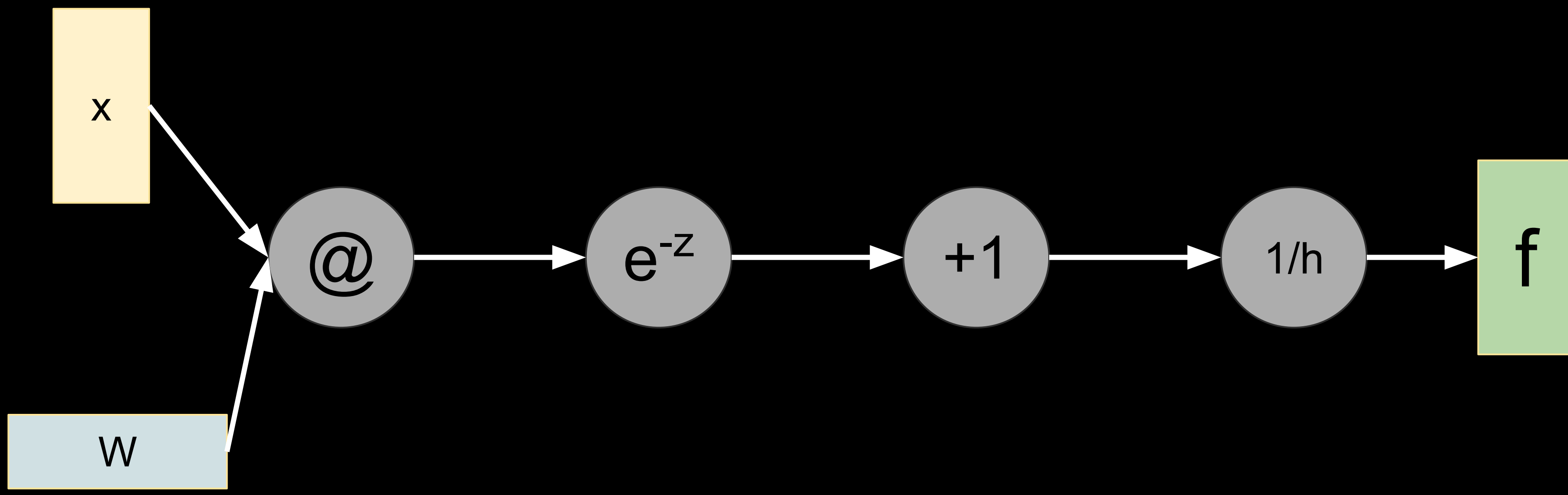
$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}$$



How do we Optimize Complex Neural Networks?

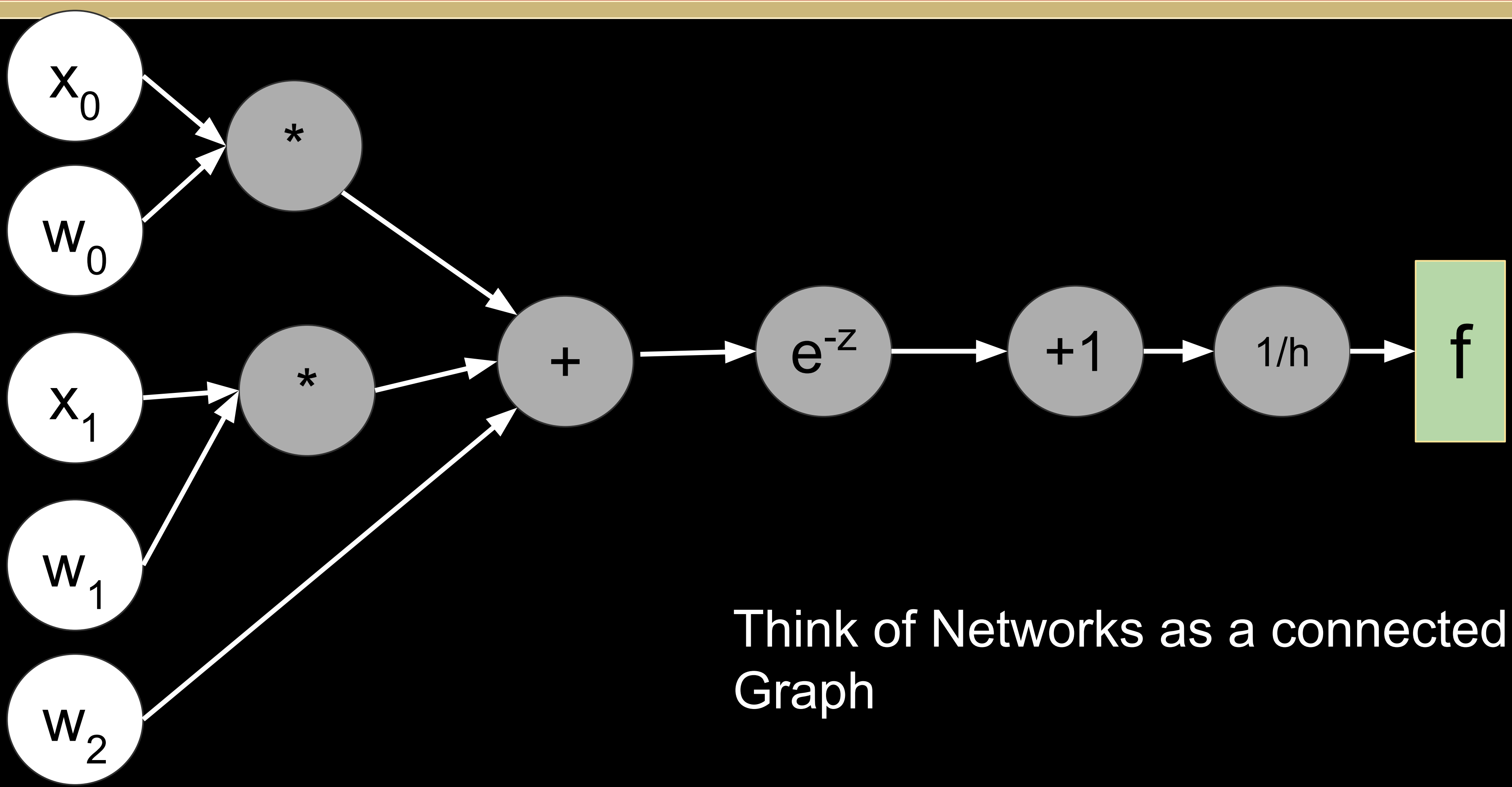


Back Propagation



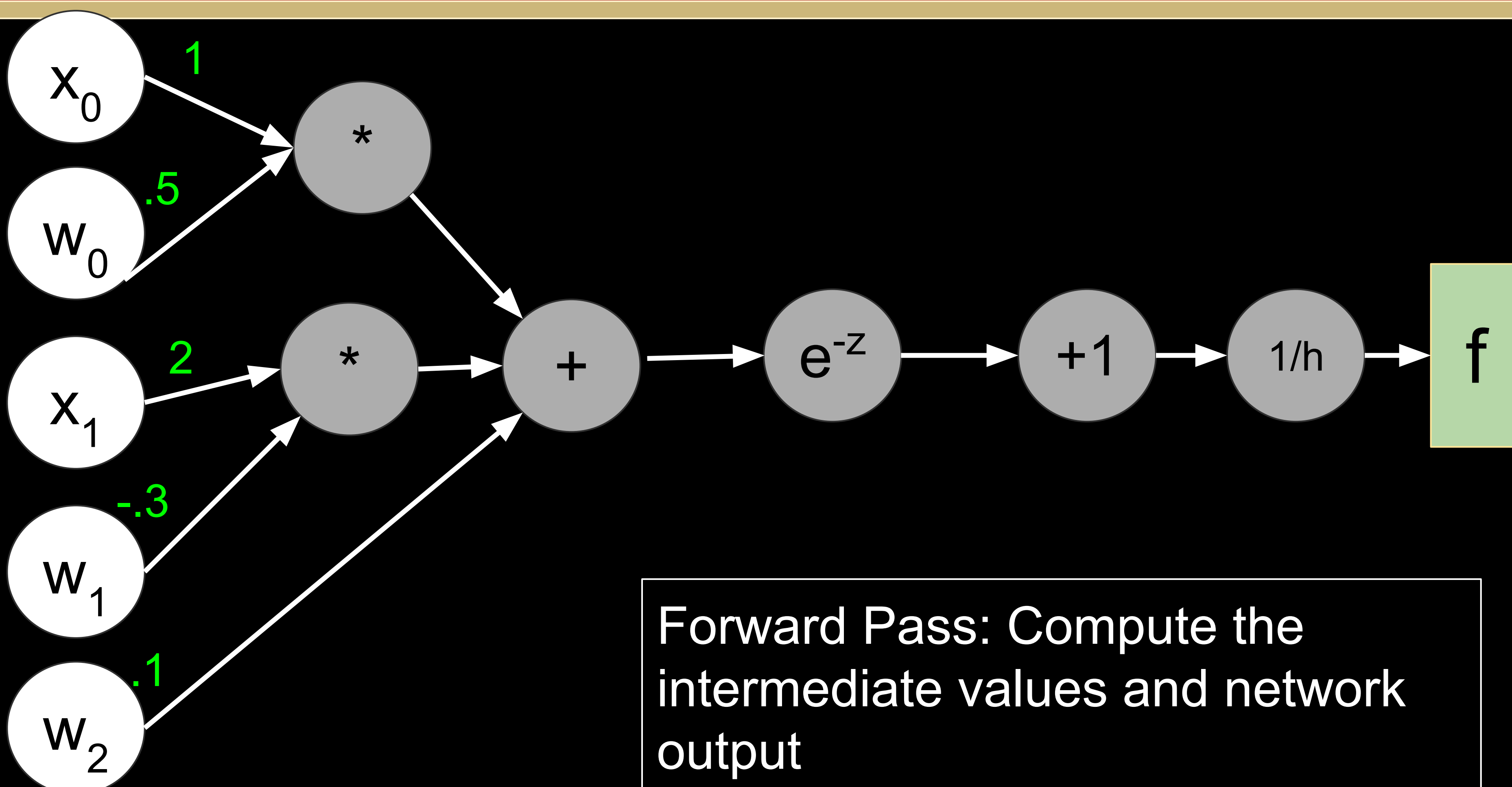
Think of Networks as a connected Graph

Back Propagation



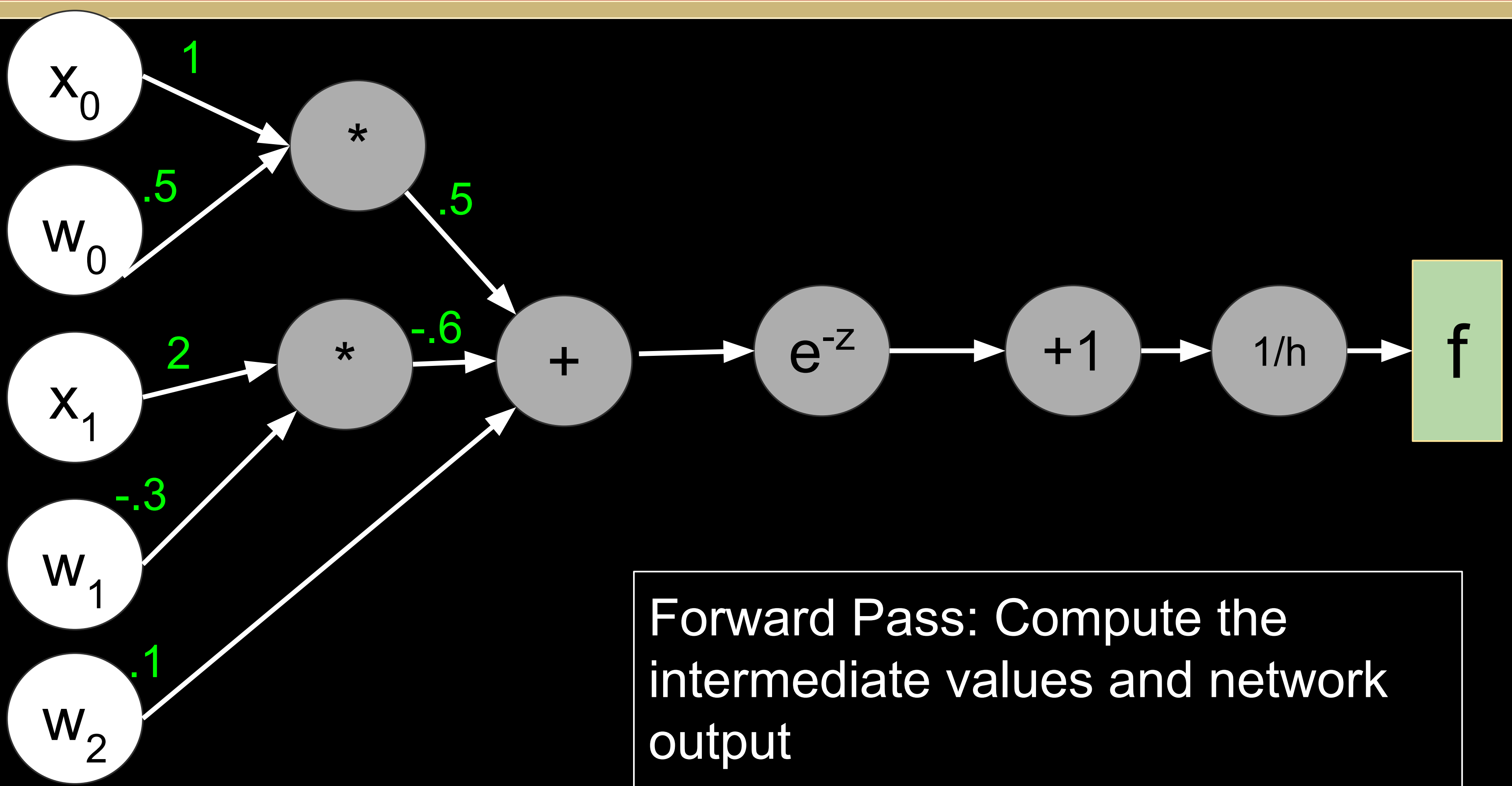
Think of Networks as a connected Graph

Back Propagation



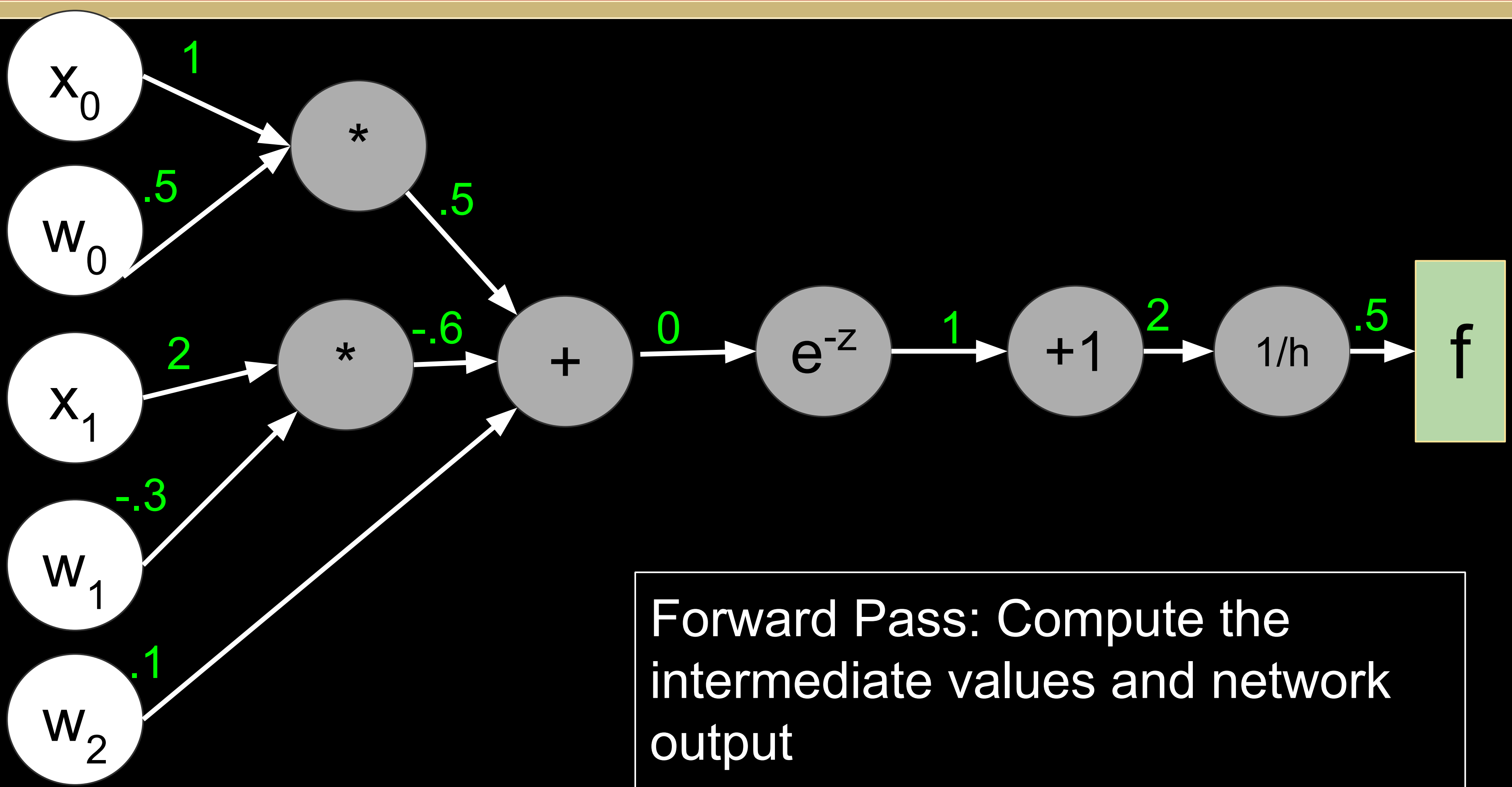
Forward Pass: Compute the intermediate values and network output

Back Propagation



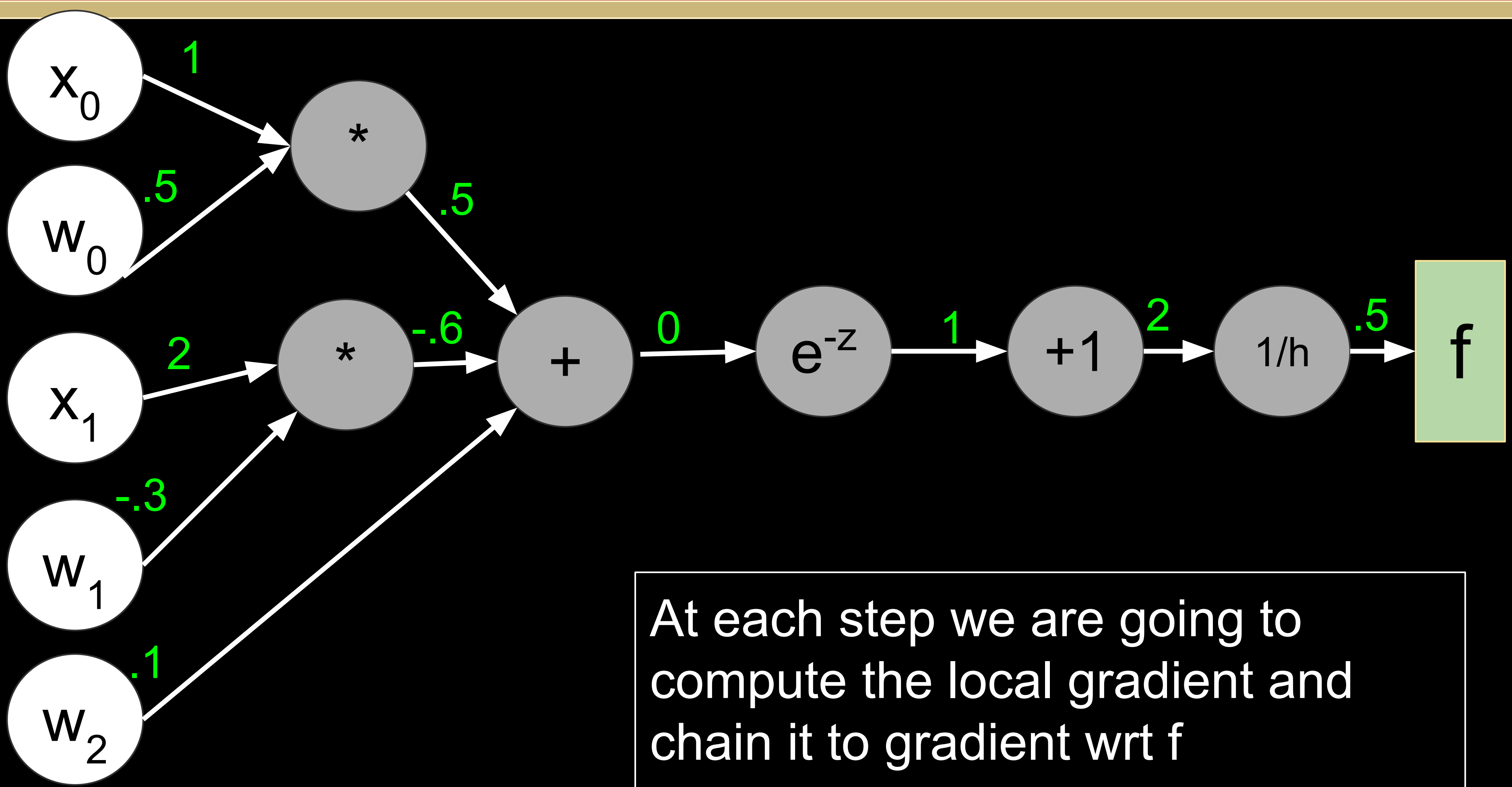
Forward Pass: Compute the intermediate values and network output

Back Propagation



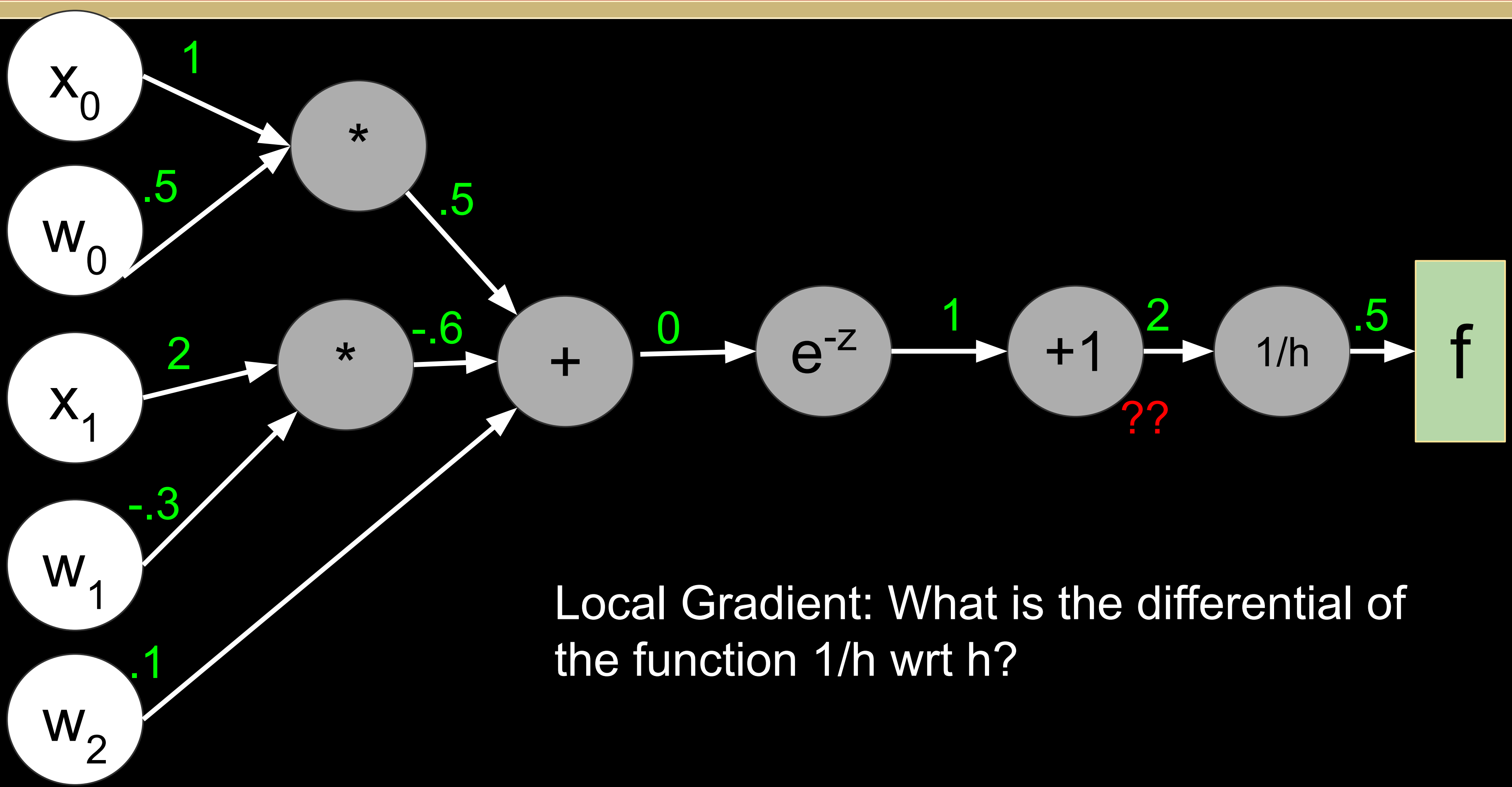
Forward Pass: Compute the intermediate values and network output

Back Propagation



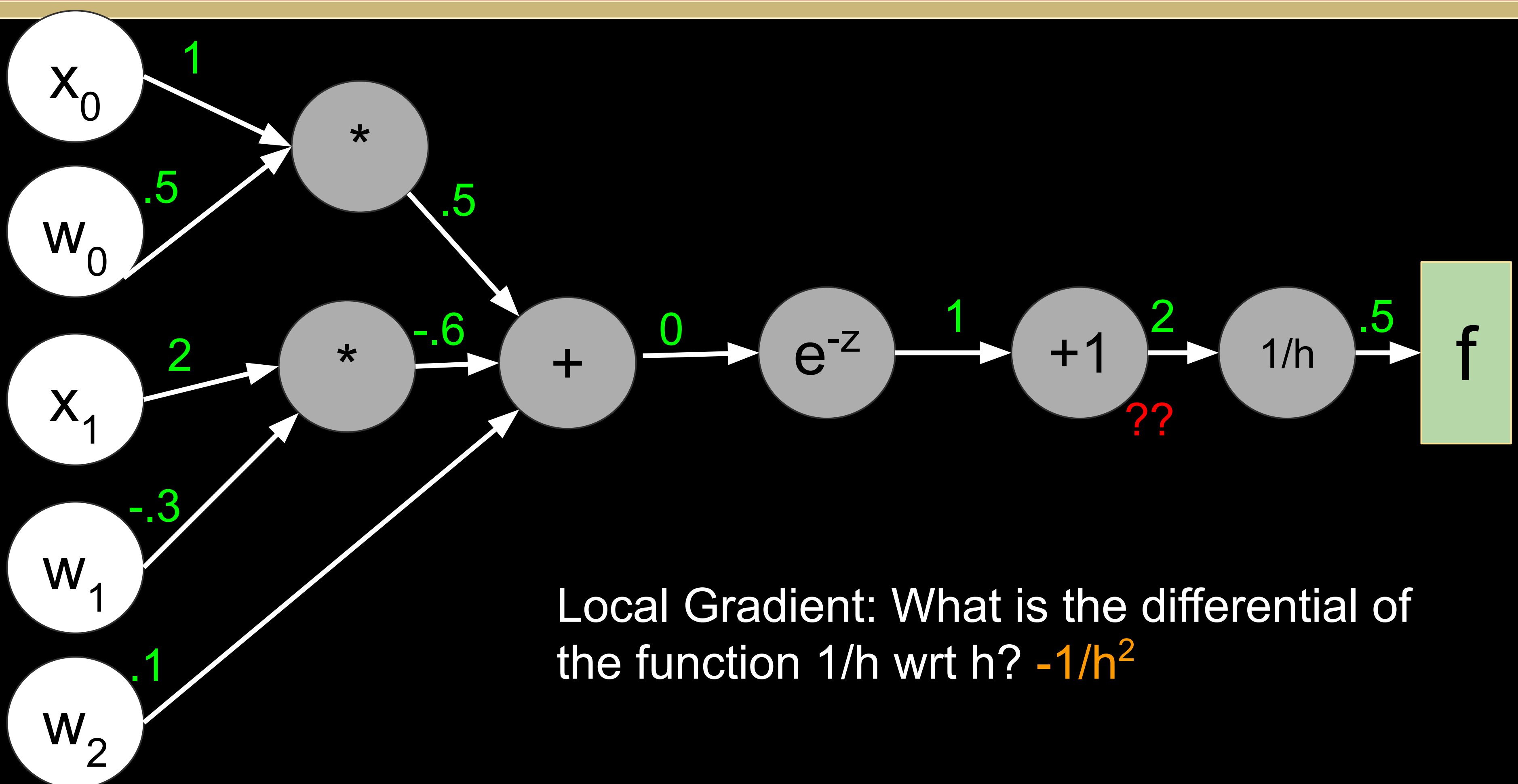
At each step we are going to compute the local gradient and chain it to gradient wrt f

Back Propagation



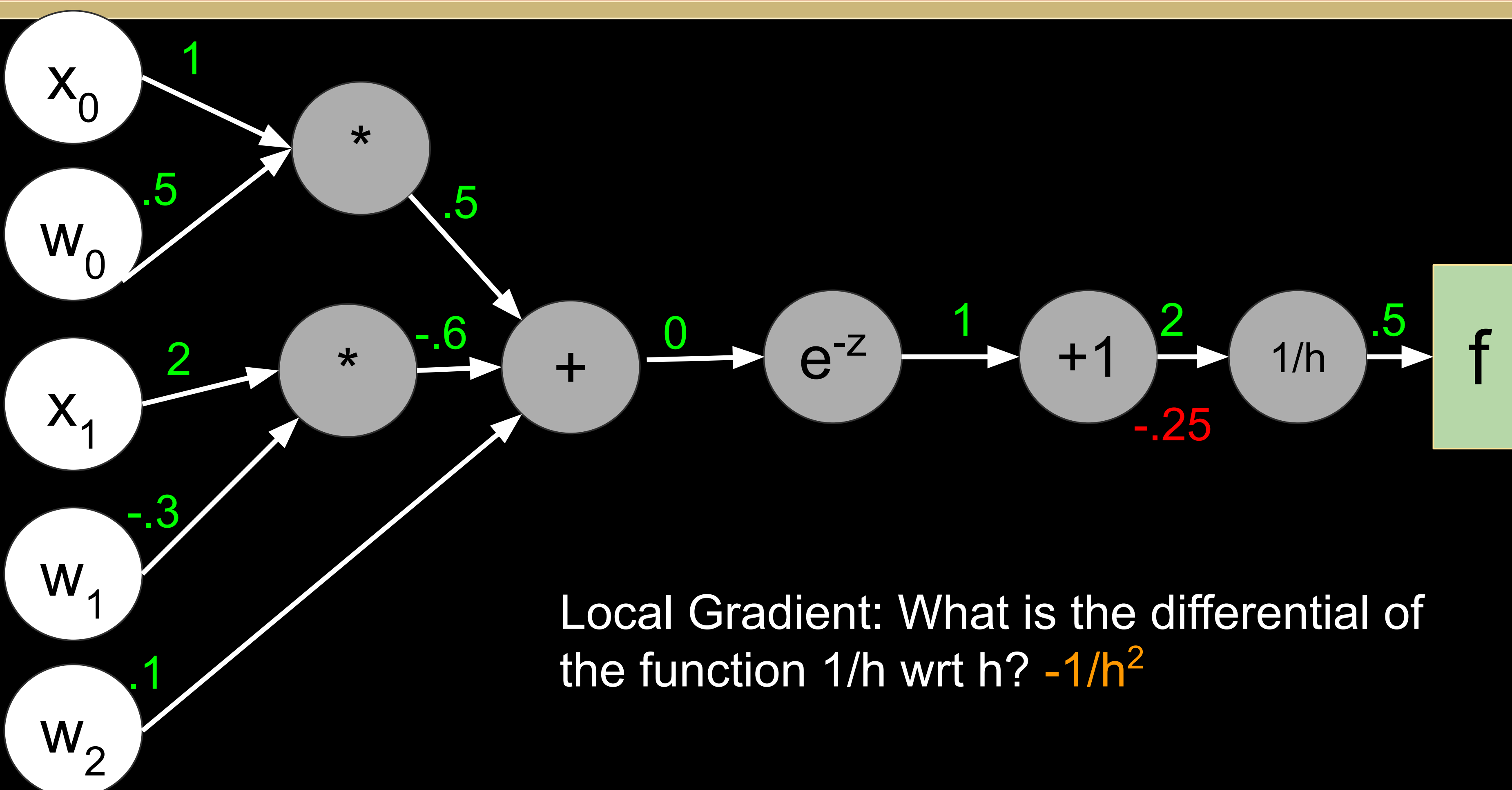
Local Gradient: What is the differential of the function 1/h wrt h?

Back Propagation



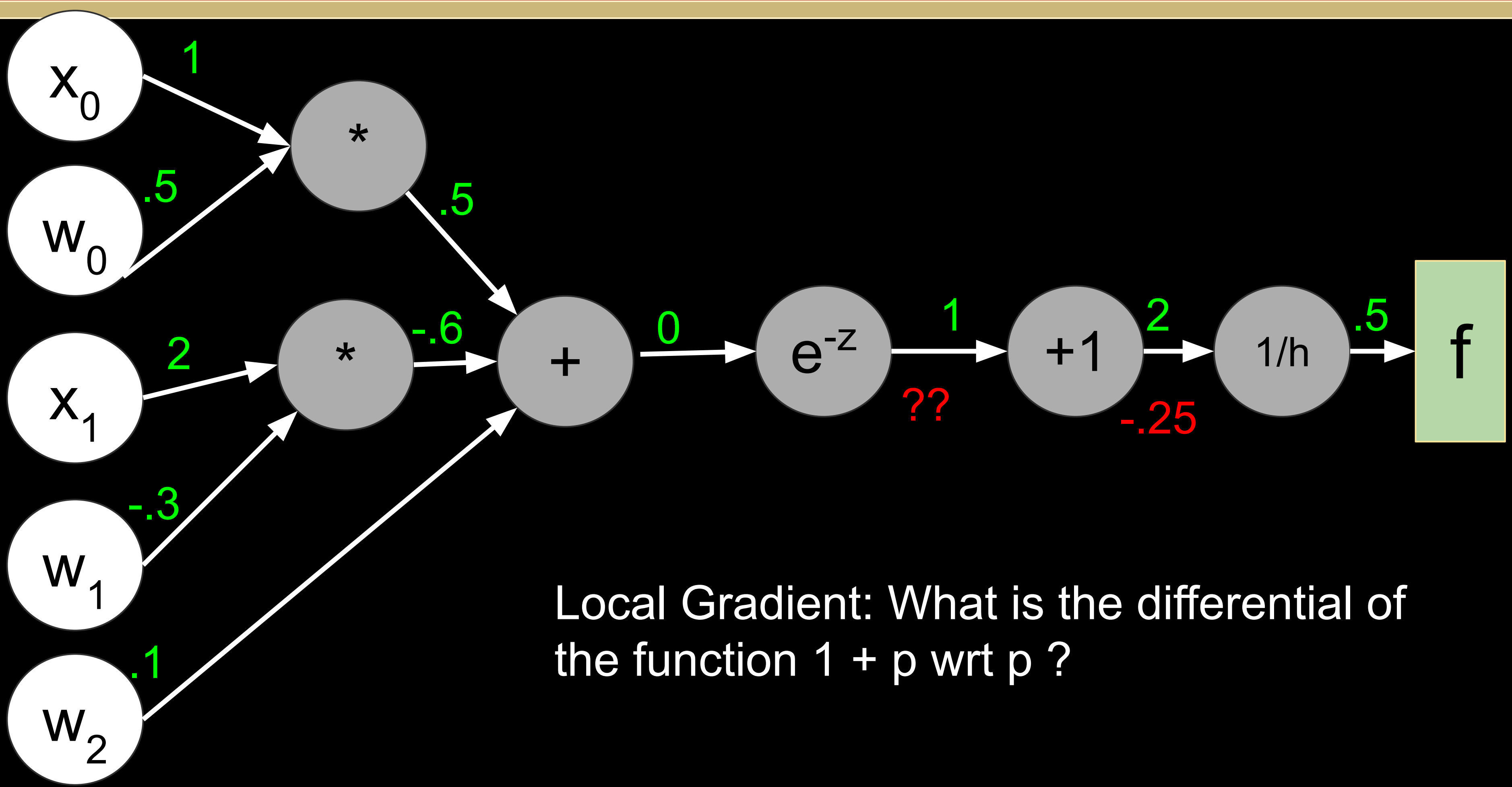
Local Gradient: What is the differential of the function 1/h wrt h? $-1/h^2$

Back Propagation



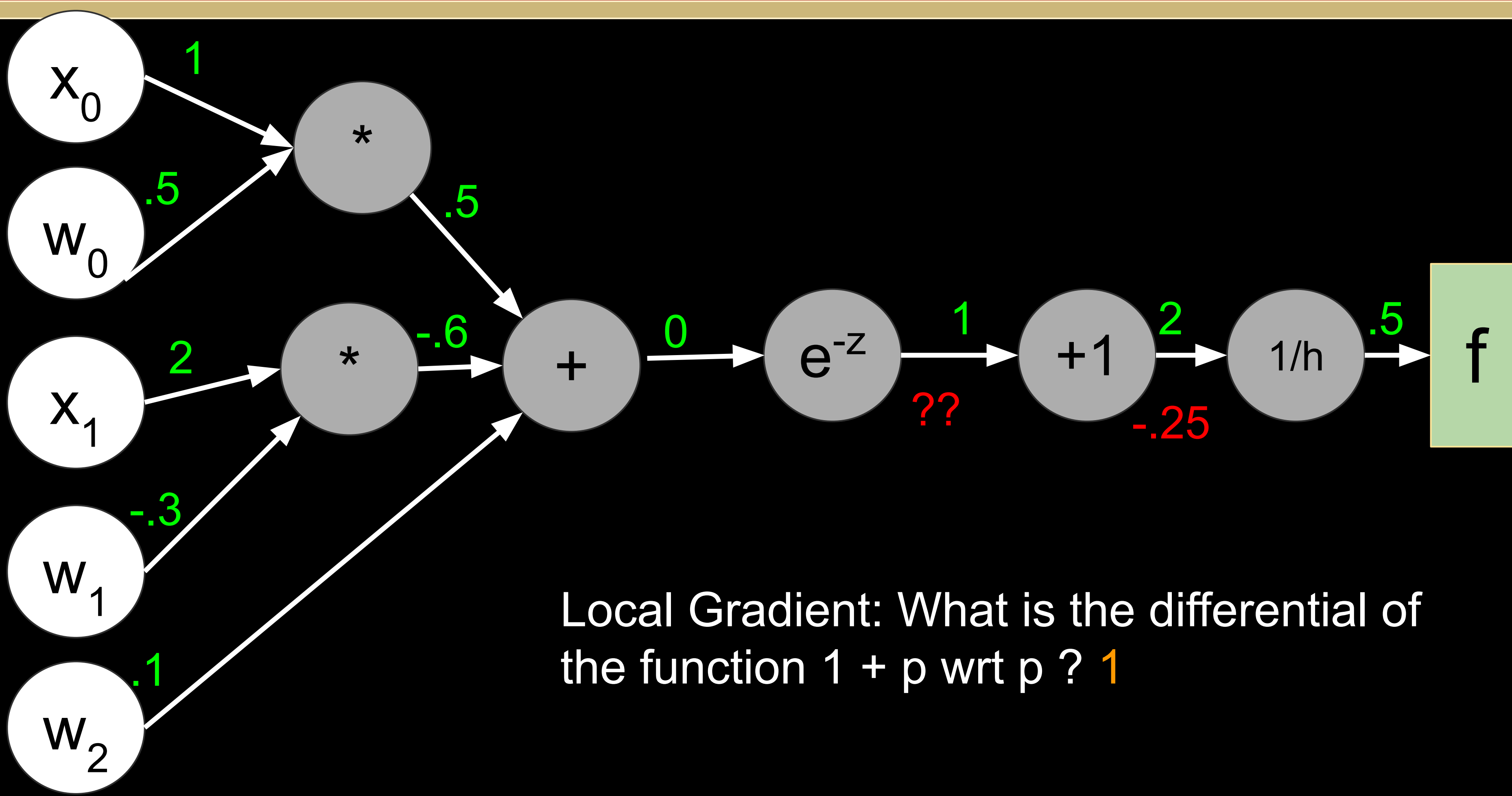
Local Gradient: What is the differential of the function 1/h wrt h? $-1/h^2$

Back Propagation



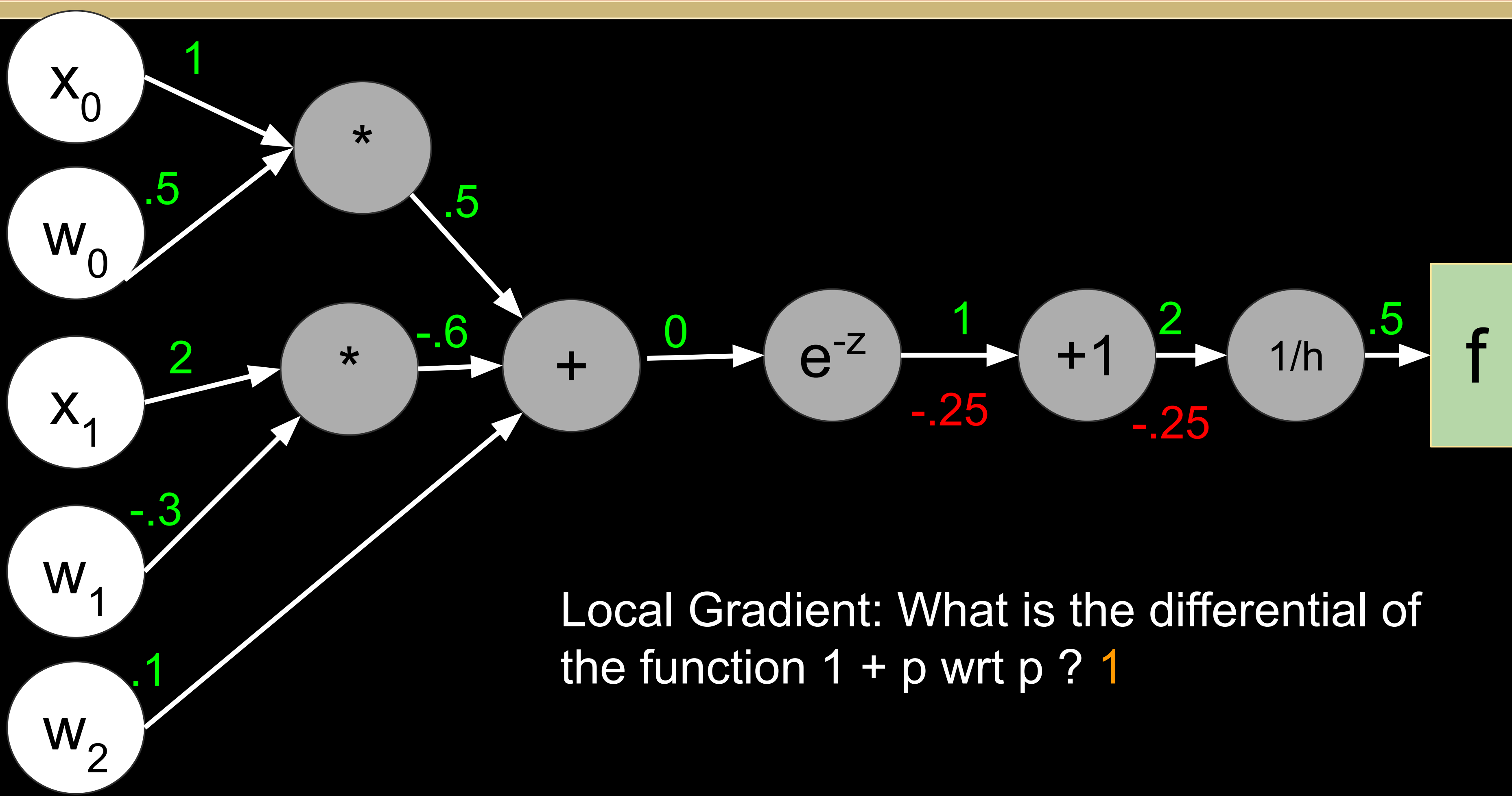
Local Gradient: What is the differential of the function 1 + p wrt p ?

Back Propagation



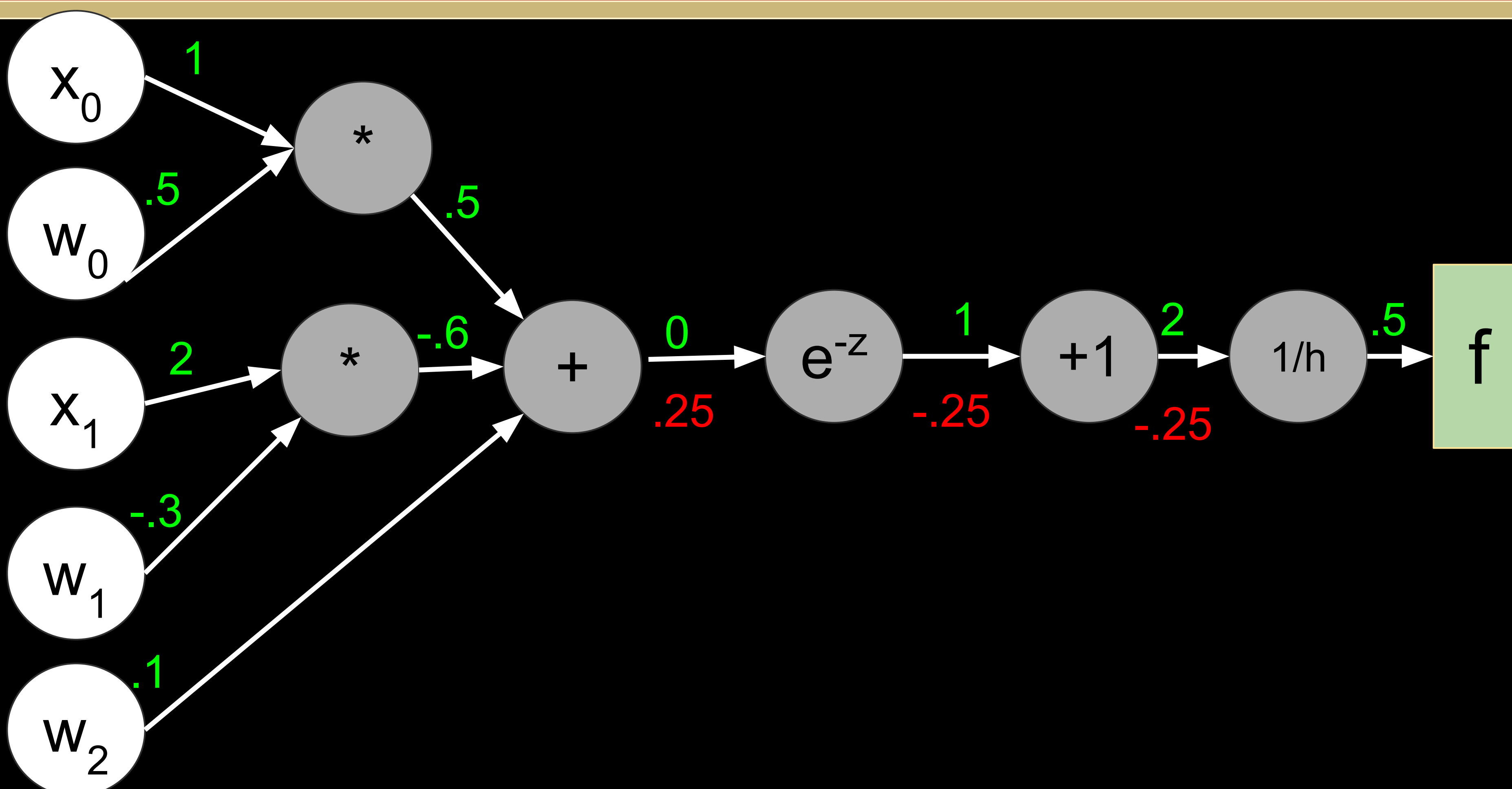
Local Gradient: What is the differential of the function 1 + p wrt p ? 1

Back Propagation

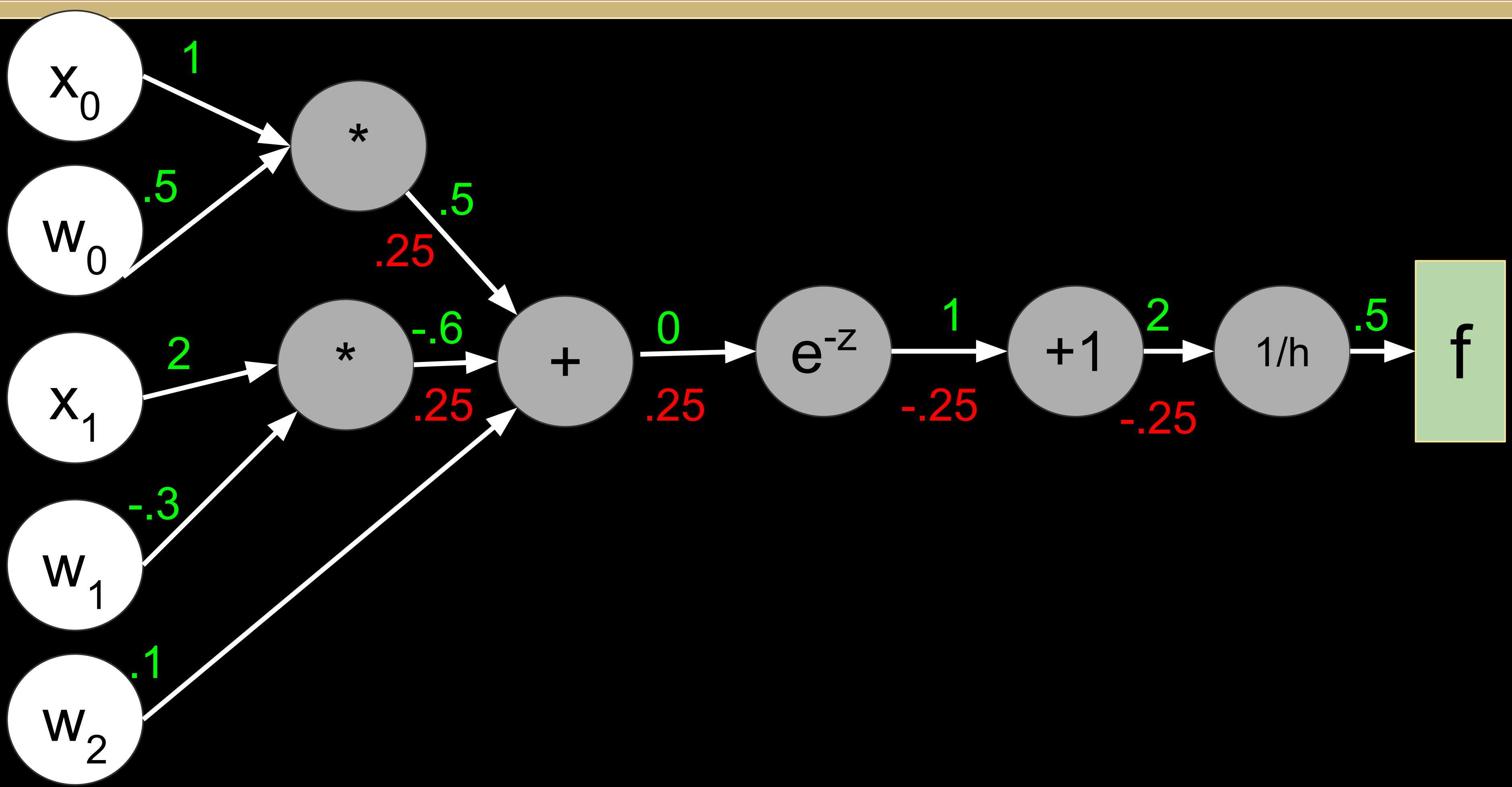


Local Gradient: What is the differential of the function 1 + p wrt p ? 1

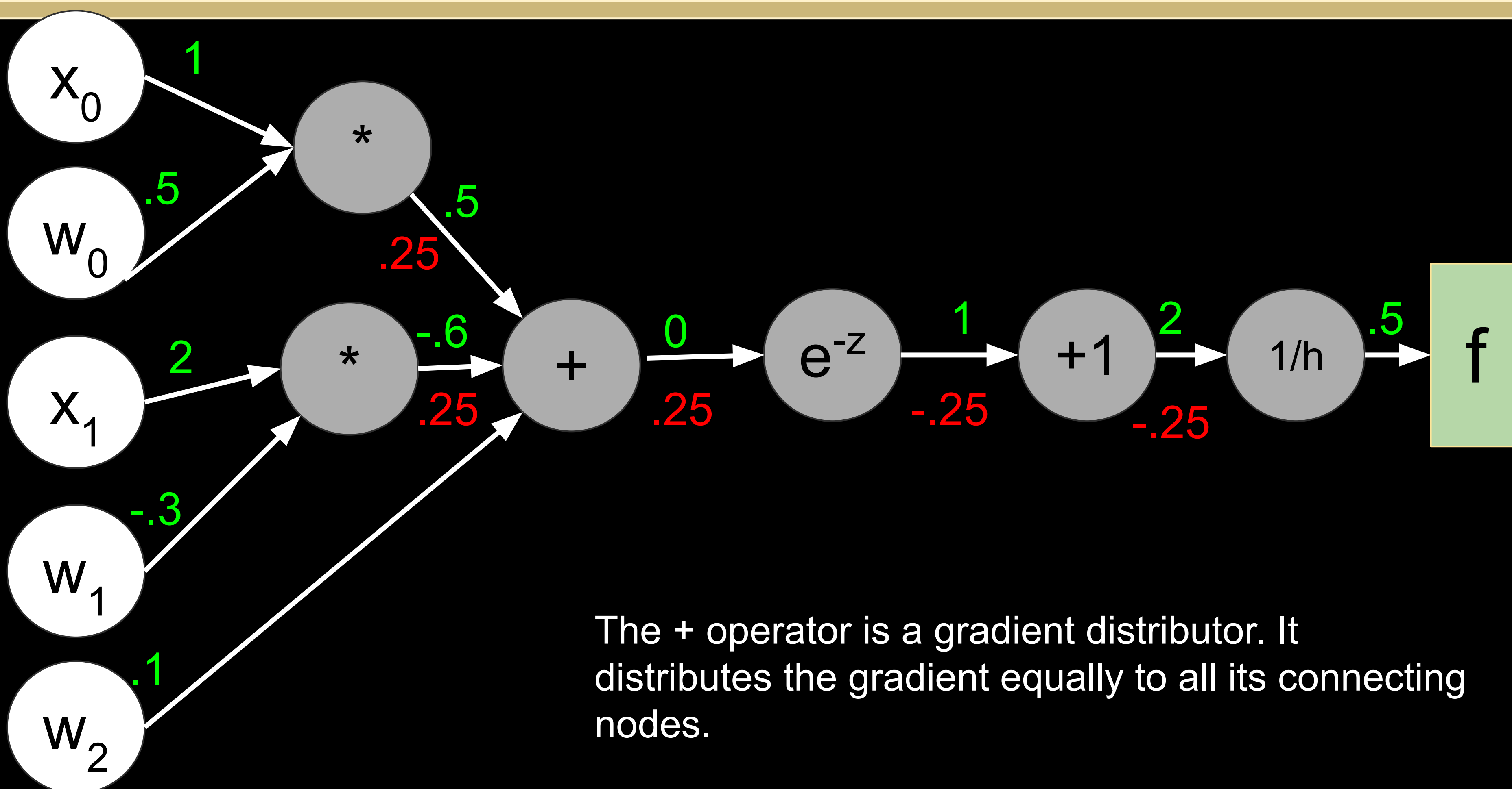
Back Propagation



Back Propagation

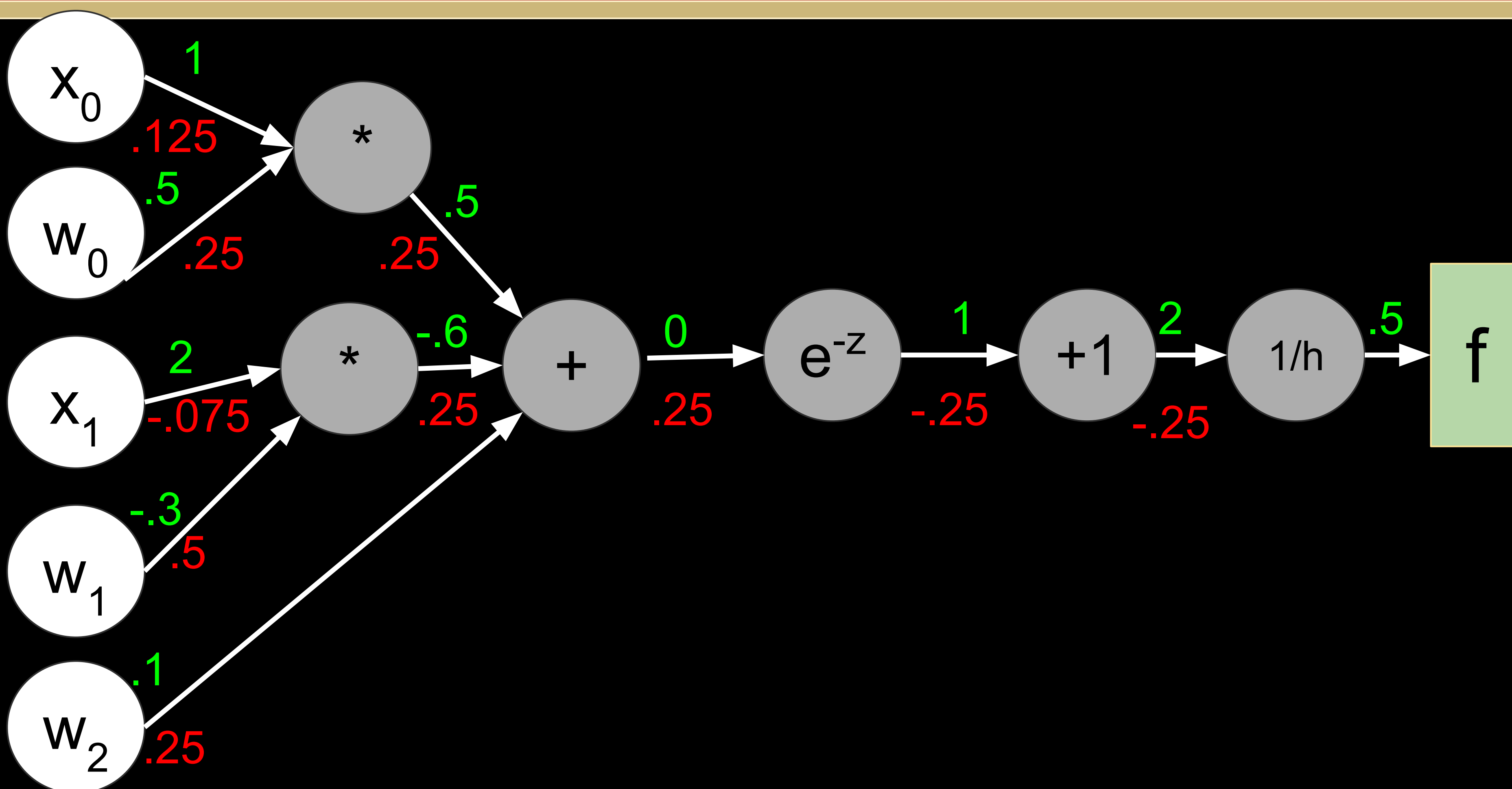


Back Propagation



The + operator is a gradient distributor. It distributes the gradient equally to all its connecting nodes.

Back Propagation



Back Propagation

- Gradients of parameters in a large complex networks can be computed by piecing the local gradients together

Back Propagation

- Gradients of parameters in a large complex networks can be computed by piecing the local gradients together

- Chain rule helps to build large networks without having to compute the gradient rule for each parameter ahead of time

Back Propagation

- Gradients of parameters in a large complex networks can be computed by piecing the local gradients together

- Chain rule helps to build large networks without having to compute the gradient rule for each parameter ahead of time

- Gradient computation of f wrt to a parameter w with intermediate output z is:

$$\partial f / \partial w = \partial f / \partial z * \partial z / \partial w$$

Back Propagation Demonstration

Demonstration ipython notebook:

https://bit.ly/cse538sp25-325-backprop

How can this be done in PyTorch?

```python
class CustomActivation(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        # Forward pass of the custom activation.
        # Compute the output
        # save intermediate variables required for the backward pass.

    @staticmethod
    def backward(ctx, grad_output):
        # Backward pass of the custom activation.
        # Compute the gradient of the loss with respect to the input.
        # Args: grad_output-Gradient of the loss wrt the activation op
        # Returns: Gradient of the loss with respect to the input.

        raise NotImplementedError
```

How can this be done in PyTorch?

```
class SigmoidActivation(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        # Compute the sigmoid function: f(x) = 1 / (1 + exp(-x))
        result = 1 / (1 + torch.exp(-input))
        ctx.save_for_backward(result)
        Return result

    @staticmethod
    def backward(ctx, grad_output):
        (result,) = ctx.saved_tensors
        # The derivative of the sigmoid is: f(x) * (1 - f(x))
        grad_input = grad_output * result * (1 - result)
        Return grad_input
```

Supplemental Review Material

What happens to this update rule when…

- we increase the size of hidden dimensions? Rule stays the same

What happens to this update rule when…

- we increase the size of hidden dimensions? — Rule stays the same
- we change the activation from sigmoid? — Substitute the partial differential of sigmoid

What happens to this update rule when…

- we increase the size of hidden dimensions?
- we change the activation from sigmoid?

- we stack more layers of RNN on top of each

ial

Compute Gradients:

- Compute Gradients:
  - Compute Gradients:
    - Compute Gradients:
      - Compute Gradients:
        - **Hidden state gradient:**

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} + \frac{\partial L}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t}$$

where $\hat{y}_t$ is the predicted output.

- **Gradients for the update gate:**

$$\frac{\partial L}{\partial z_t} = \frac{\partial L}{\partial h_t} \odot (\tilde{h}_t - h_{t-1})$$

$$\frac{\partial L}{\partial W_z} = \sum_t \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t) \cdot x_t^T$$

$$\frac{\partial L}{\partial U_z} = \sum_t \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t) \cdot h_{t-1}^T$$

$$\frac{\partial L}{\partial b_z} = \sum_t \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t)$$

- **Gradients for the reset gate:**

$$\frac{\partial L}{\partial r_t} = \left(\frac{\partial L}{\partial \tilde{h}_t} \odot U_h h_{t-1} \right) \odot \sigma'(r_t)$$

- **Gradients for the candidate hidden state:**

$$\frac{\partial L}{\partial \tilde{h}_t} = \frac{\partial L}{\partial h_t} \odot z_t \odot (1 - \tanh^2(\tilde{h}_t))$$

- **Gradient updates for weights and biases:**

$$\frac{\partial L}{\partial W_h} = \sum_t \frac{\partial L}{\partial \tilde{h}_t} \cdot x_t^T$$

$$\frac{\partial L}{\partial U_h} = \sum_t \frac{\partial L}{\partial \tilde{h}_t} \cdot (r_t \odot h_{t-1})^T$$

$$\frac{\partial L}{\partial b_h} = \sum_t \frac{\partial L}{\partial \tilde{h}_t}$$

What happens to this update rule when…

- we increase the size of hidden dimensions?
- we change the activation from sigmoid?

Rule stays the same

Substitute the partial differential of sigmoid

- we stack more layers of RNN on top of each other?

Hot mess!

What happens to this update rule when…

- we increase the size of hidden dimensions?
- we change the activation from sigmoid?

Rule stays the same

Substitute the partial differential of sigmoid

- we stack more layers of RNN on top of each other?

Hot mess!

But how are we performing Gradient Descent on these Complex Models?

What happens to this update rule when…

- we increase the size of hidden dimensions?
- we change the activation from sigmoid?

Rule stays the same

Substitute the partial differential of sigmoid

- we stack more layers of RNN on top of each other?

Hot mess!

But how are we performing Gradient Descent on these Complex Models? Back Propagation

(More on the lecture after spring break!!!)