Recurrent Neural Networks

Recurrent neural networks have become the model of choice when dealing with sequence data. This slide deck outlines the benefits of gated RNNs and introduces some interesting applications of encoder-decoder systems built for sequence to sequence learning.

ENGG 192 – Thayer School of Engineering
March 14, 2019
Spencer R. Bertsch

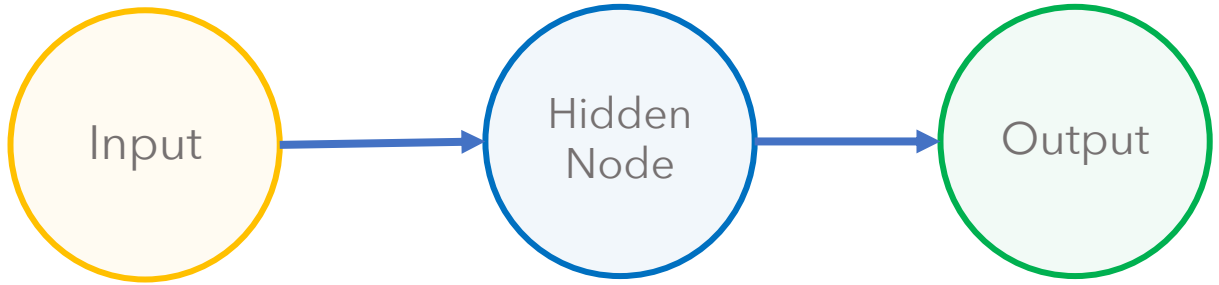
Table of Contents

Recurrent Neural Networks Overview

Review of neural networks and reasons behind inception of RNNs

Simple one-layer (one neuron) network

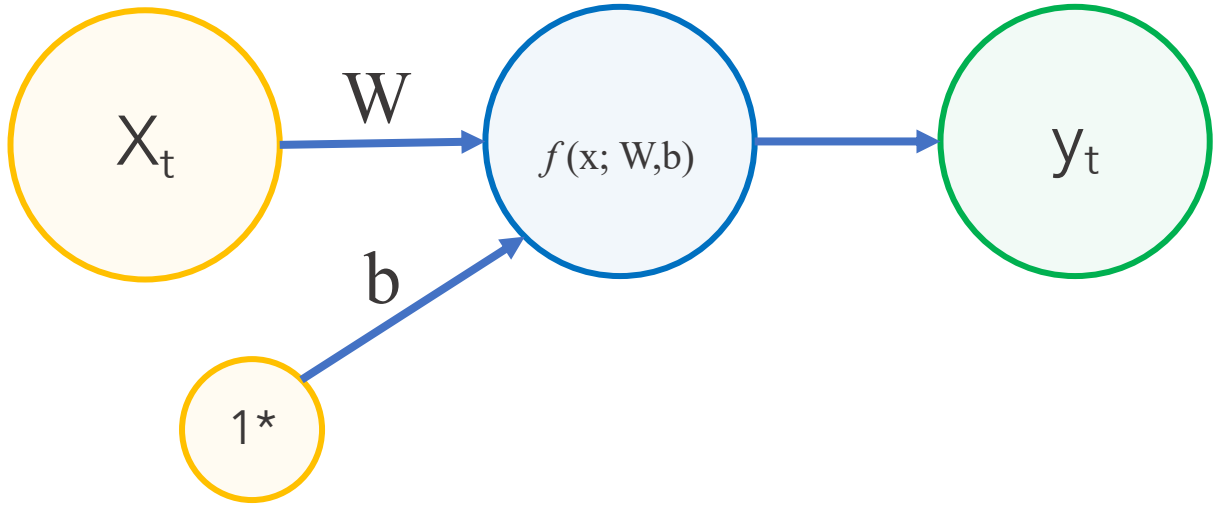
This is a representation of the simplest possible network in which a single input is fed through a single hidden node. The output from this node is then fed through a sigmoid or tanh function to produce the output.



Simple one-layer (one neuron) network

This is a representation of the simplest possible network in which a single input is fed through a single hidden node. The output from this node is then fed through a sigmoid or tanh function to produce the output.

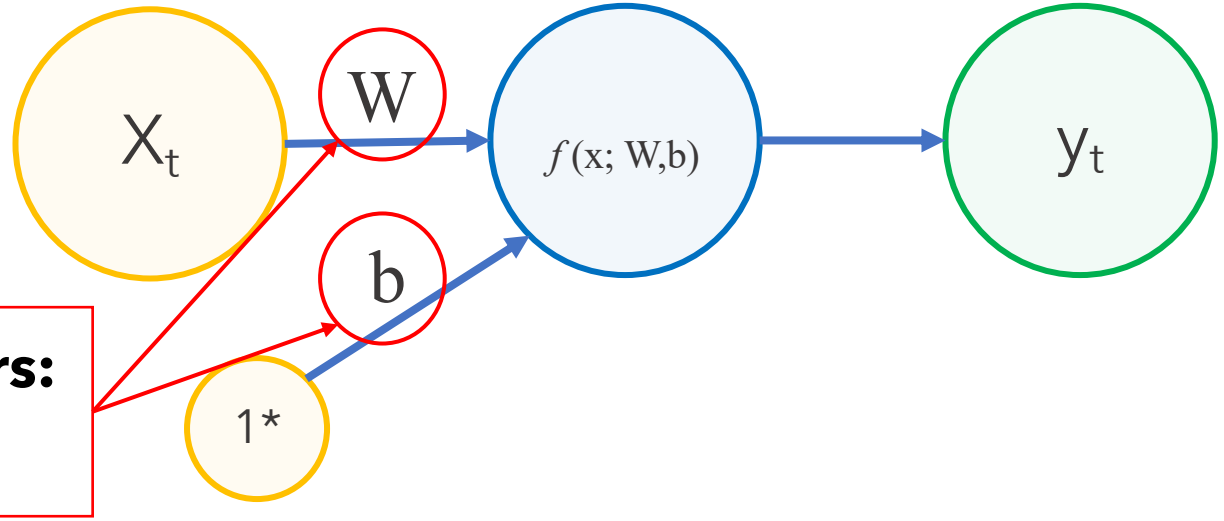
$$f\left[\sigma(W^{T}x + b)\right]$$



Simple one-layer (one neuron) network

This is a representation of the simplest possible network in which a single input is fed through a single hidden node. The output from this node is then fed through a sigmoid or tanh function to produce the output.

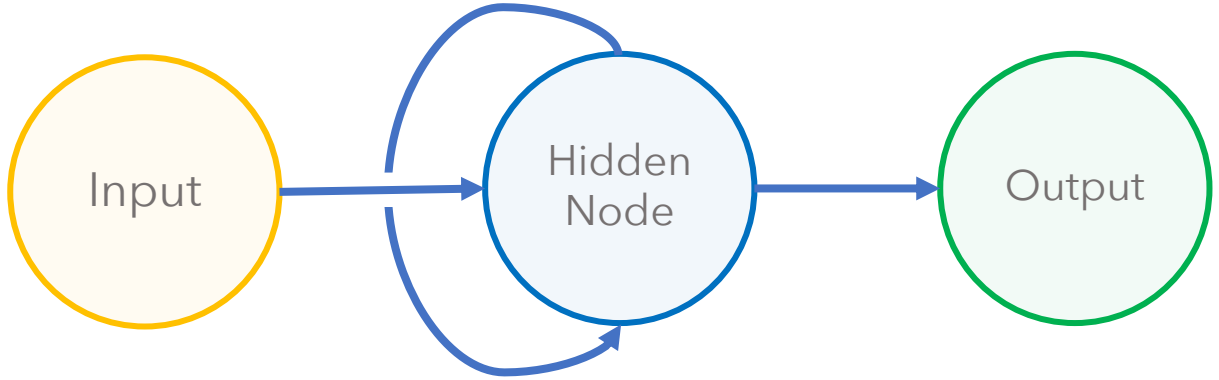
$$f[\sigma(W^T x + b)]$$



Learned Parameters:
W and b

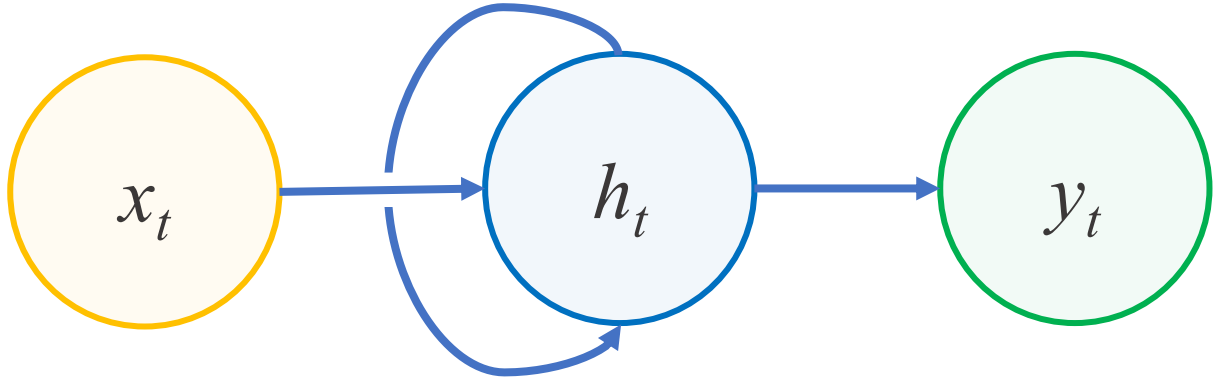
Simple one-layer (one neuron) recurrent network

This is a representation of the simplest possible recurrent network in which a single input is fed through a single hidden node. In addition to using the new input, the hidden node also uses its previous output as an input. The hidden node uses both of these features to produce an output, which is then fed through a sigmoid or tanh function to produce the final prediction.



Simple one-layer (one neuron) recurrent network

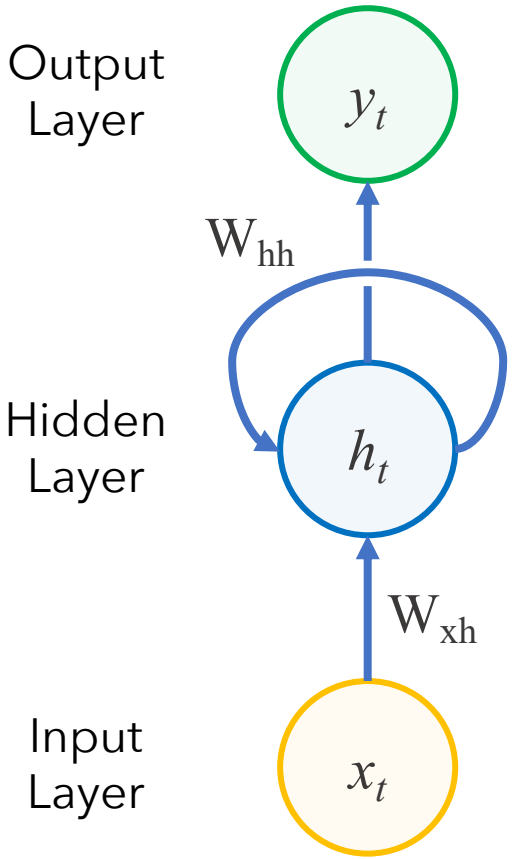
This is a representation of the simplest possible recurrent network in which a single input is fed through a single hidden node. In addition to using the new input, the hidden node also uses its previous output as an input. The hidden node uses both of these features to produce an output, which is then fed through a sigmoid or tanh function to produce the final prediction.



$$h_t = f(h_{t-1}, x_t; \theta)$$

Sources: Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

Calculating each of the hidden states Parameters don't vary with time

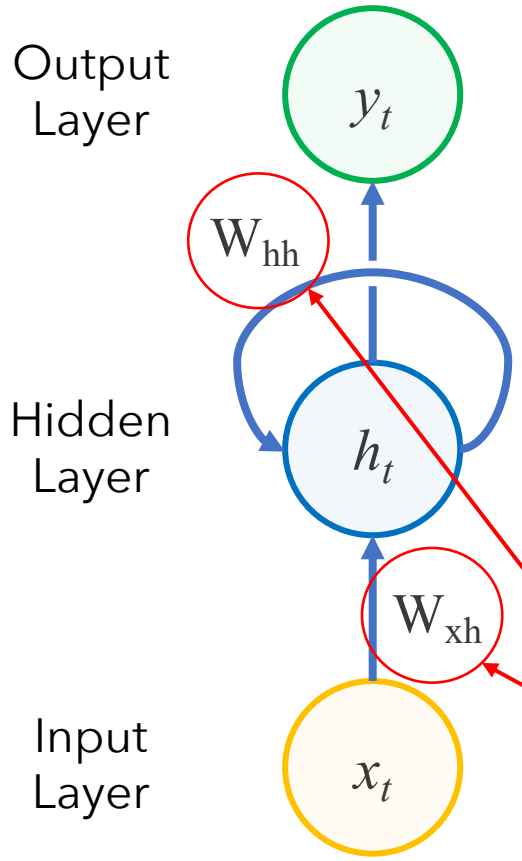


$$\theta_h = W_{hh}, W_{xh}$$

$$h_t = f(h_{t-1}, x_t ; \theta_h) \qquad h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

Calculating each of the hidden states Parameters don't vary with time



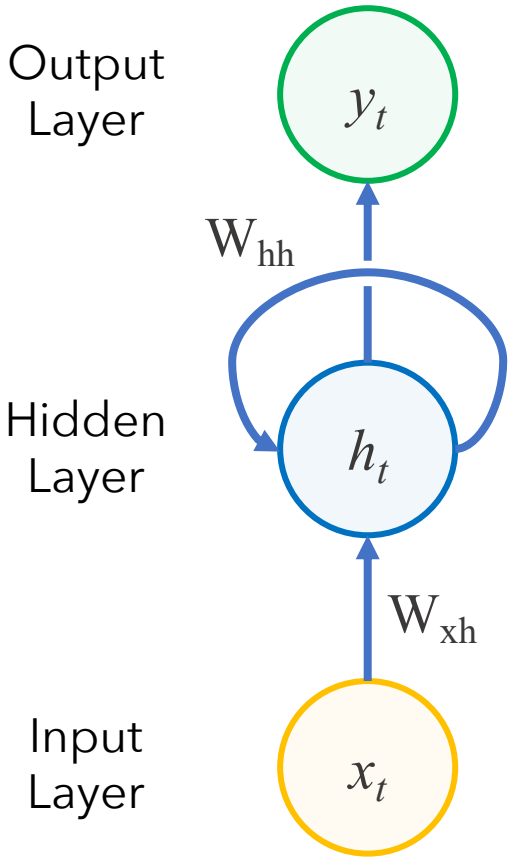
$$\theta_h = W_{hh}, W_{xh}$$

$$h_t = f(h_{t-1}, x_t ; \theta_h) \qquad h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

Learned Parameters:
W_{hh} and W_{xh}

Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

Calculating each of the hidden states Parameters don't vary with time



$$y_t = g(h_t ; \theta_y) \qquad y_t = (h_t W_{hy})$$

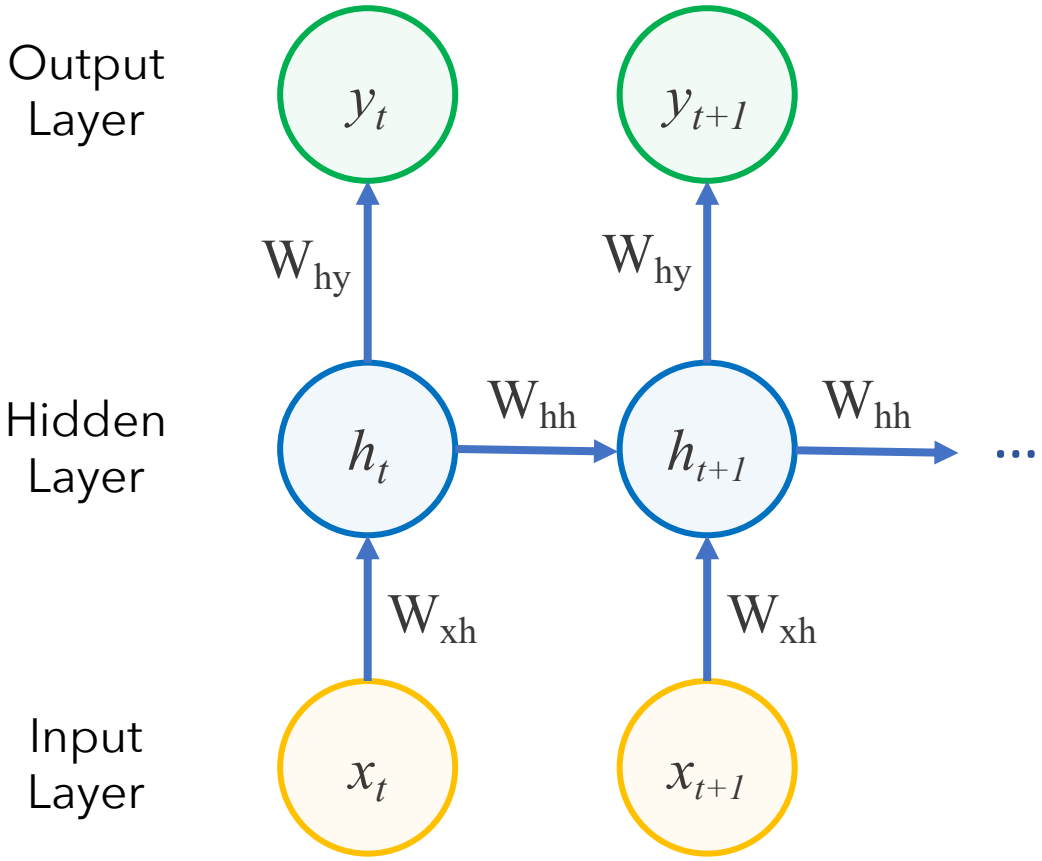
$$\theta_y = W_{hy}$$

$$h_t = f(h_{t-1}, x_t ; \theta_h) \qquad h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$\theta_h = W_{hh,}\, W_{xh}$$

Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

Calculating each of the hidden states Parameters don't vary with time



$$y_t = g(h_t ; \theta_y) \qquad y_t = (h_t W_{hy})$$

$$\theta_y = W_{hy}$$

$$h_t = f(h_{t-1}, x_t ; \theta_h) \qquad h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$x_t = input$$

$$\theta_h = W_{hh}, W_{xh}$$

Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

Calculating each of the hidden states Parameters don't vary with time

Previous hidden state (at *t-1*)

Hidden state parameter matrix

Previous hidden state (at t-1)

$$h_t = f(h_{t-1}, x_t; \theta)$$

$$h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

Input at time=(t)

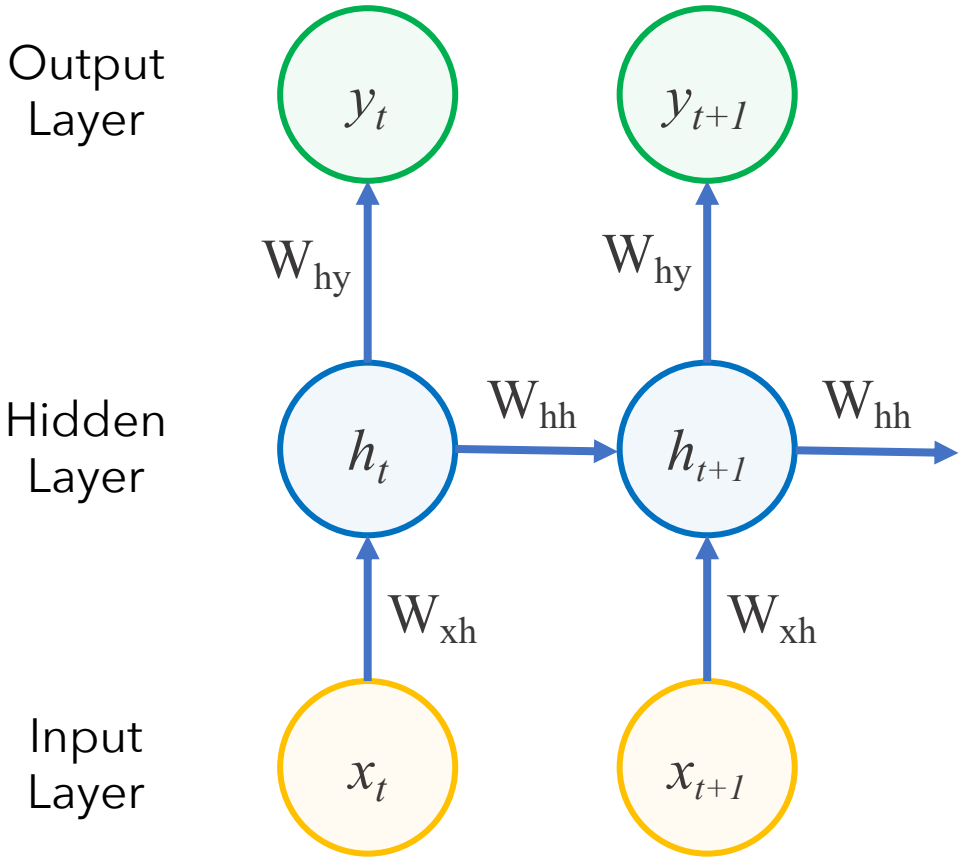
W_{hh} and W_{xh} matrices (Learned Parameters)

Input parameter matrix

Input at time=(t)

Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

Calculating each of the hidden states Parameters don't vary with time



$$y_t = g(h_t ; \theta_y) \qquad y_t = (h_t W_{hy})$$

$$\theta_y = W_{hy}$$

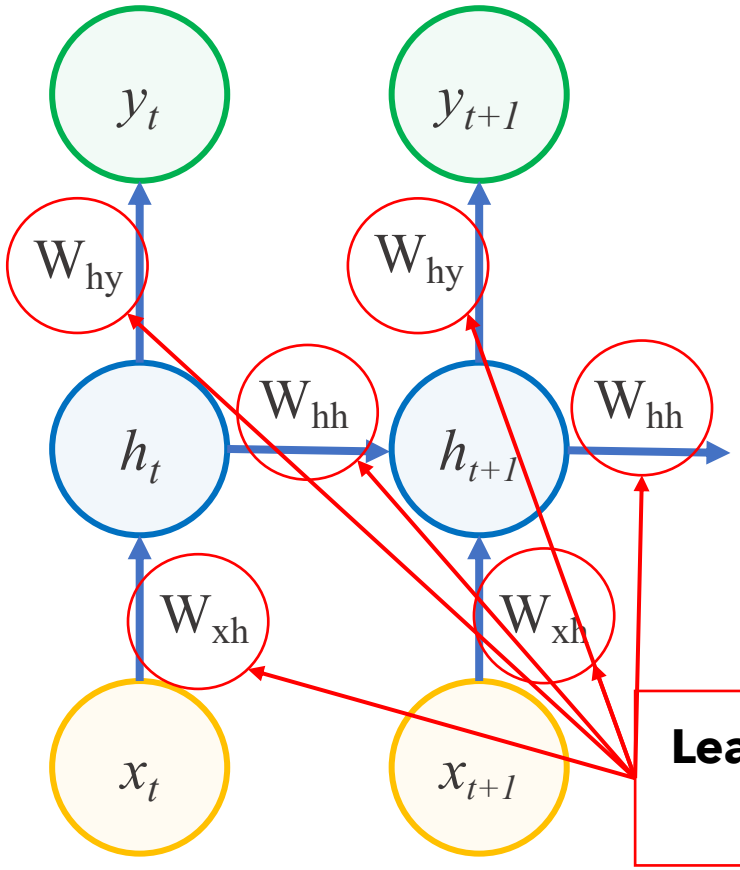
$$h_t = f(h_{t-1}, x_t ; \theta_h) \qquad h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$x_t = input$$

$$\theta_h = W_{hh}, W_{xh}$$

Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

Calculating each of the hidden states Parameters don't vary with time



$$y_t = g(h_t ; \theta_y) \qquad y_t = (h_t W_{hy})$$

$$\theta_y = W_{hy}$$

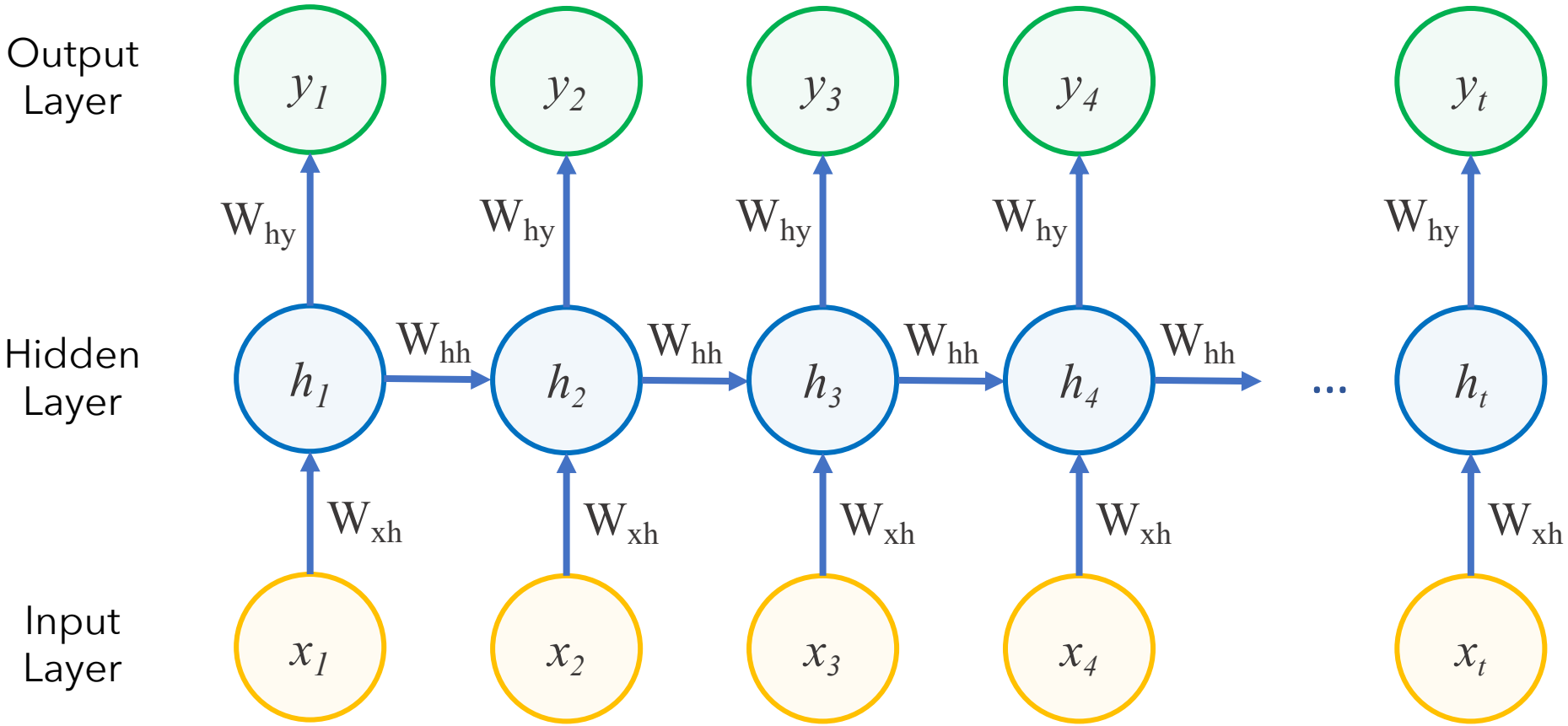
$$\cdots \quad h_t = f(h_{t-1}, x_t ; \theta_h) \qquad h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

Learned Parameters:
W_{hh}, W_{xh}, W_{hy}

$$\theta_h = W_{hh}, W_{xh}$$

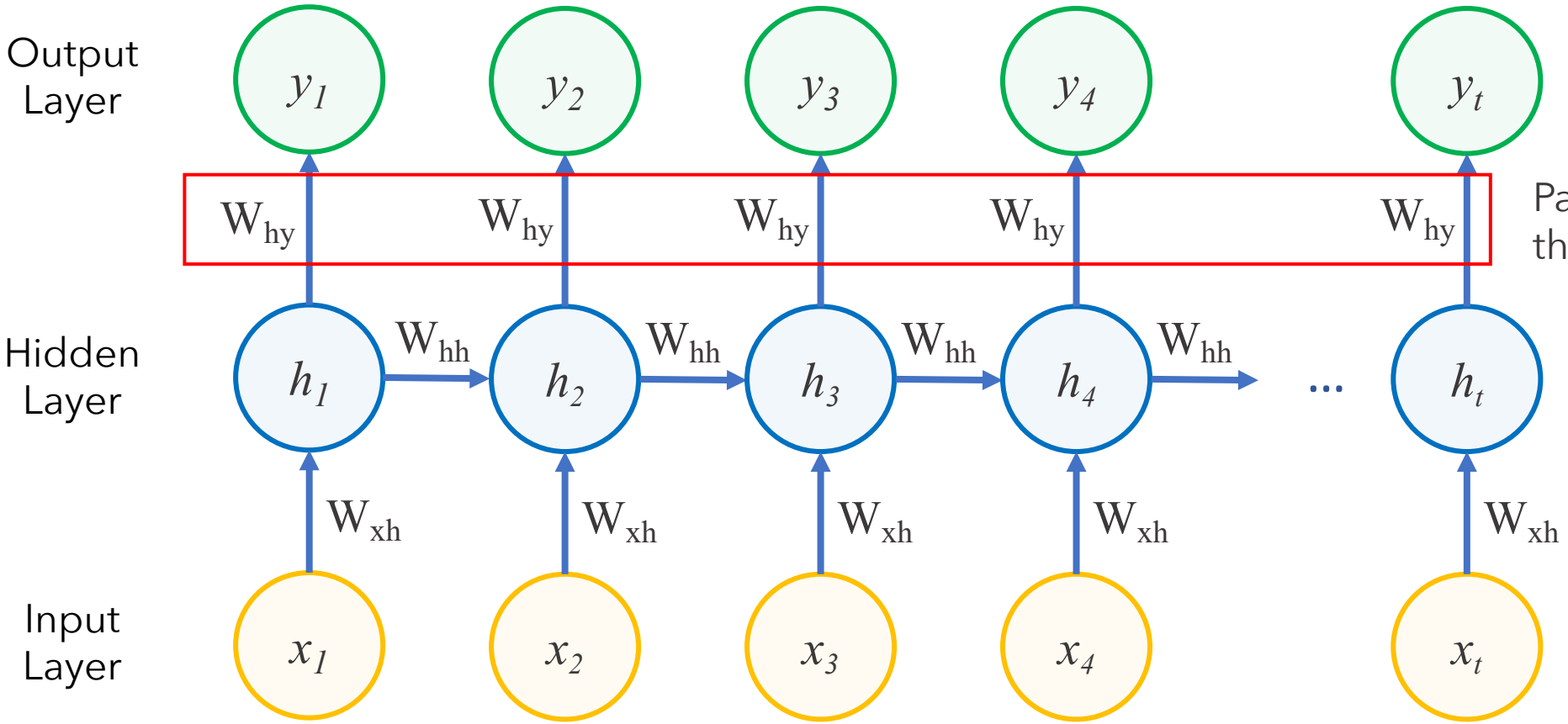
Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

Calculating each of the hidden states Parameters don't vary with time



Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

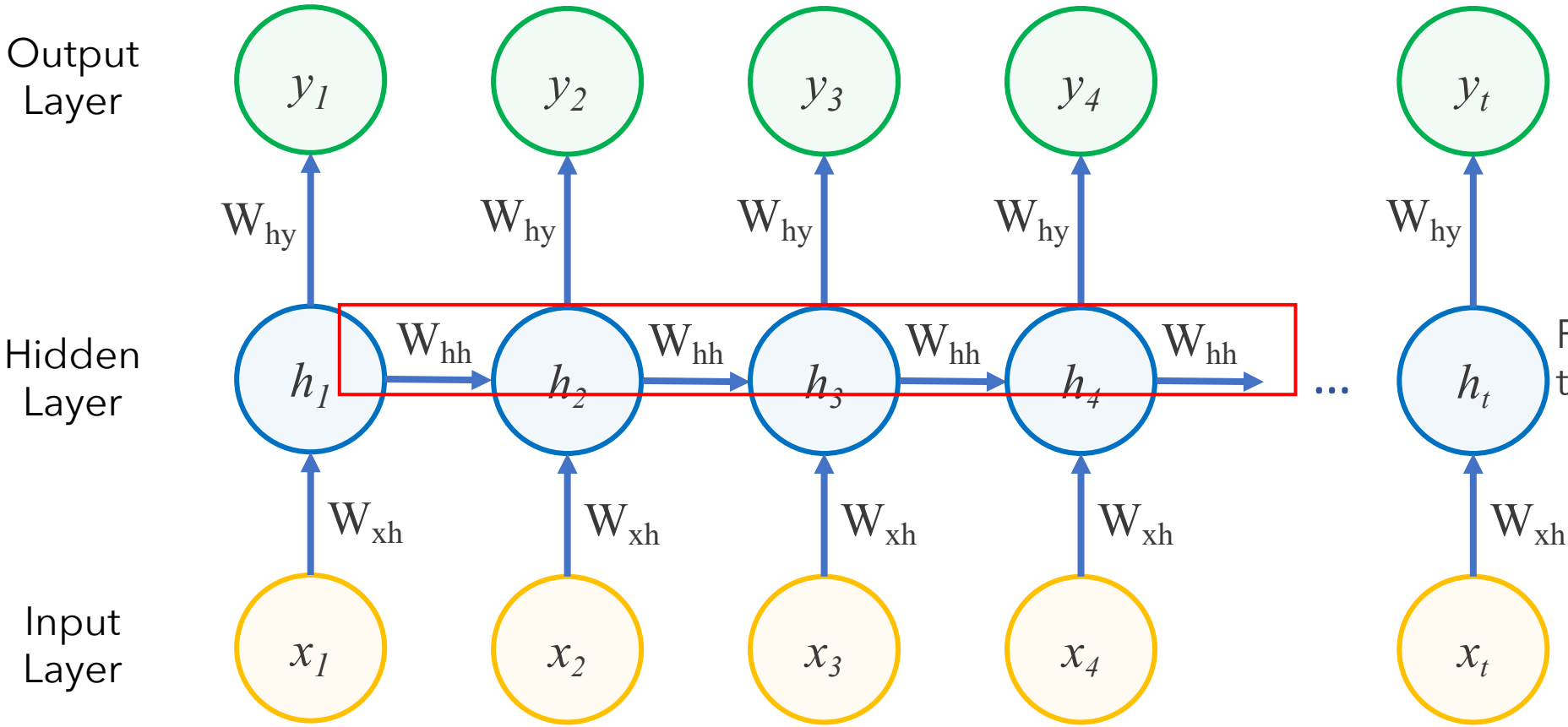
Calculating each of the hidden states Parameters don't vary with time



Parameters remain constant through time steps!

Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

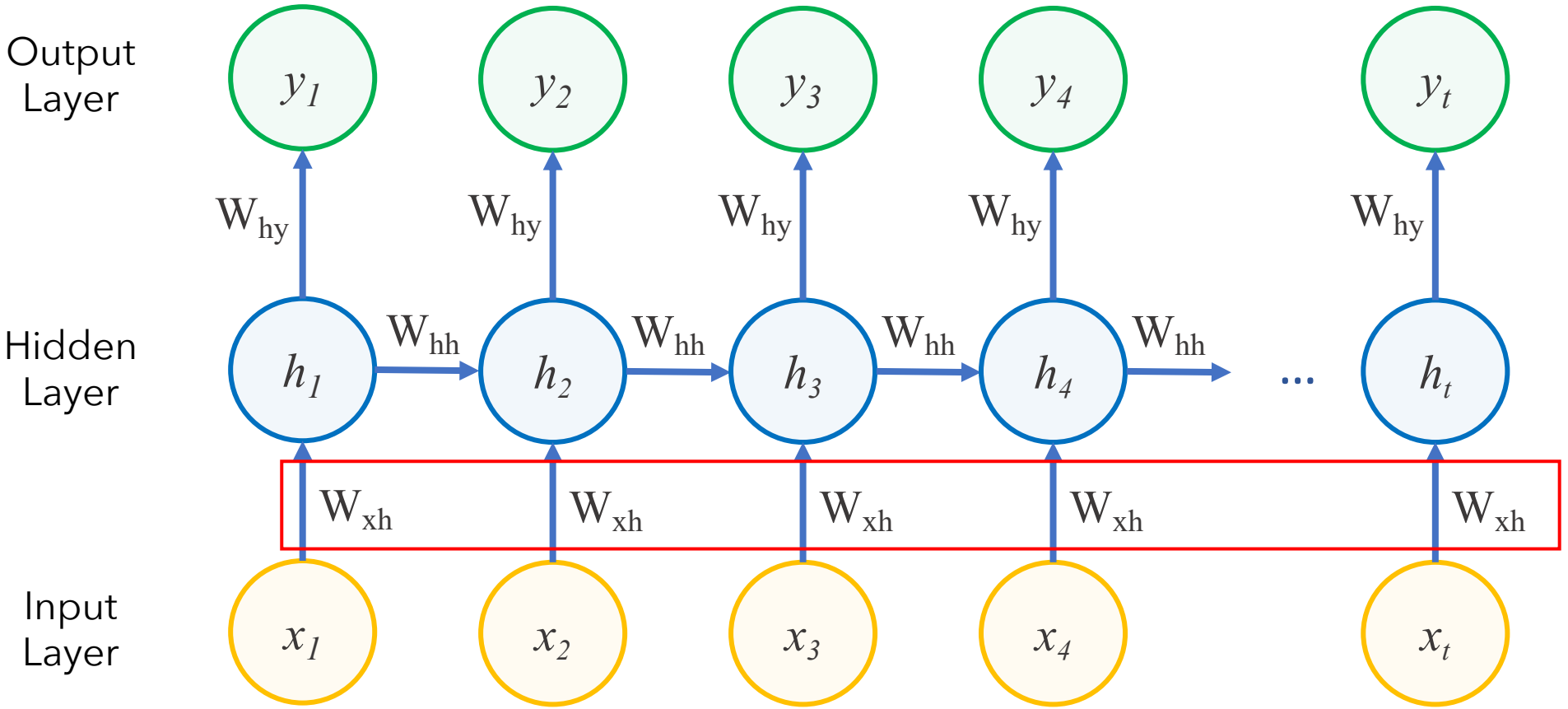
Calculating each of the hidden states Parameters don't vary with time



Parameters remain constant through time steps!

Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

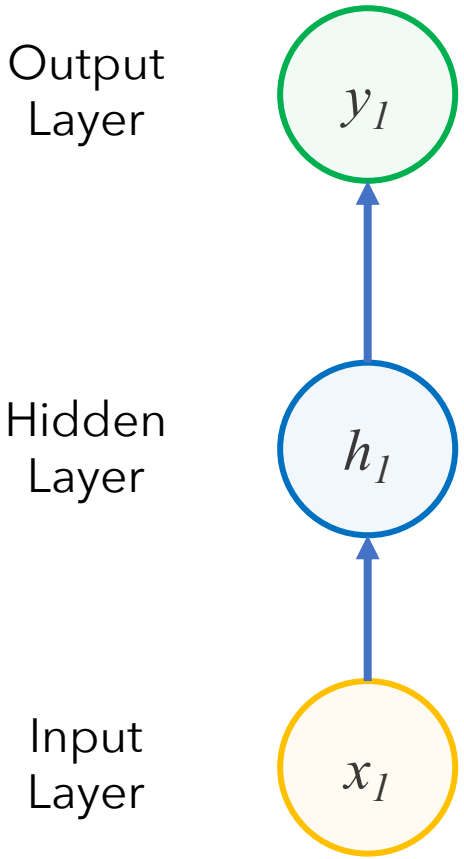
Calculating each of the hidden states Parameters don't vary with time



Parameters remain constant through time steps!

Sources: Torresanni, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Andrej Karpathy

Examples of RNN Applications: Vanilla ANN

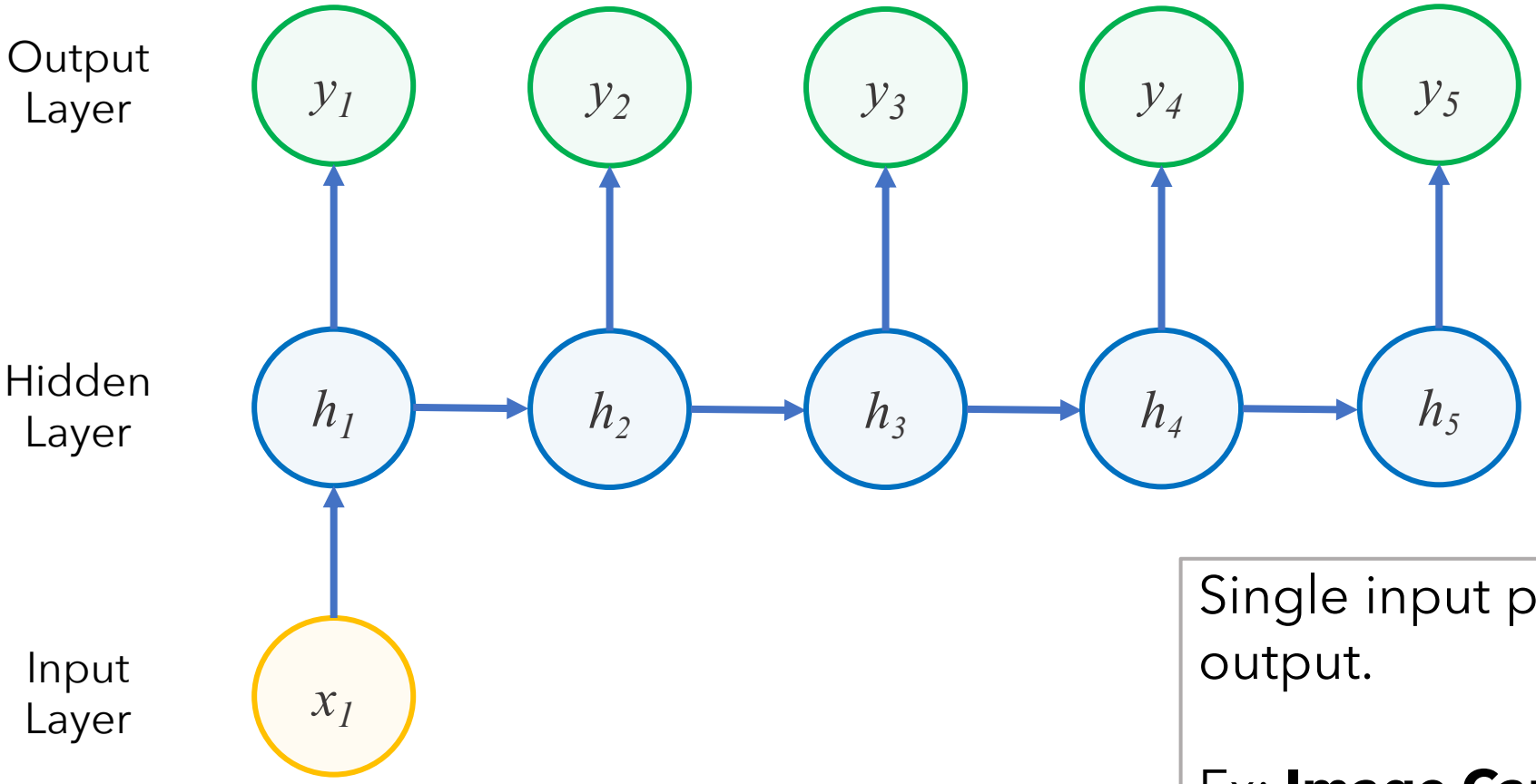


Fixed size input layer to fixed size output layer.

Ex: **Image Classification**

Sources: Andrej Karpathy

Examples of RNN Applications: Sequence Output

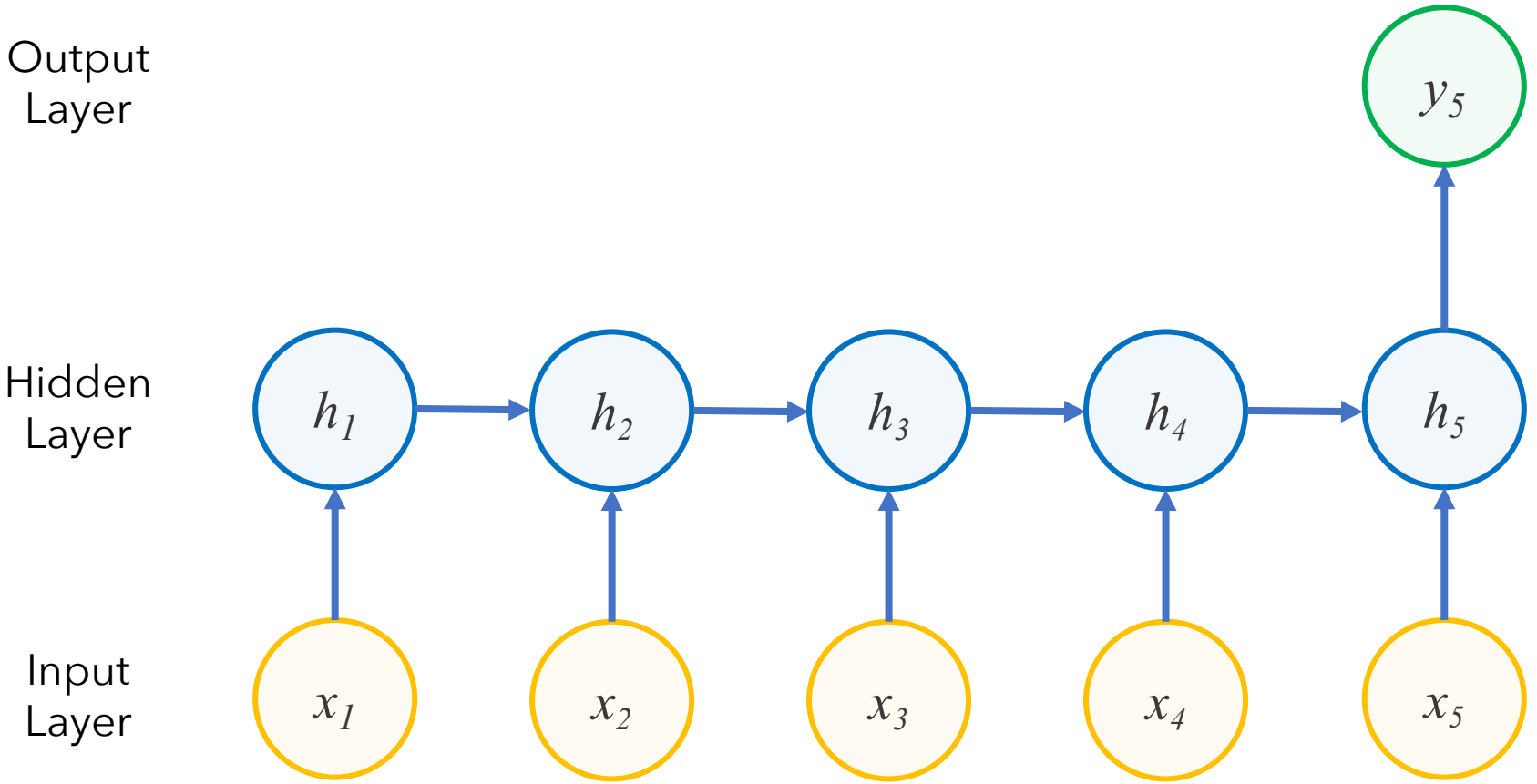


Single input producing a sequence output.

Ex: **Image Captioning**

Sources: Andrej Karpathy

Examples of RNN Applications: Sequence Input

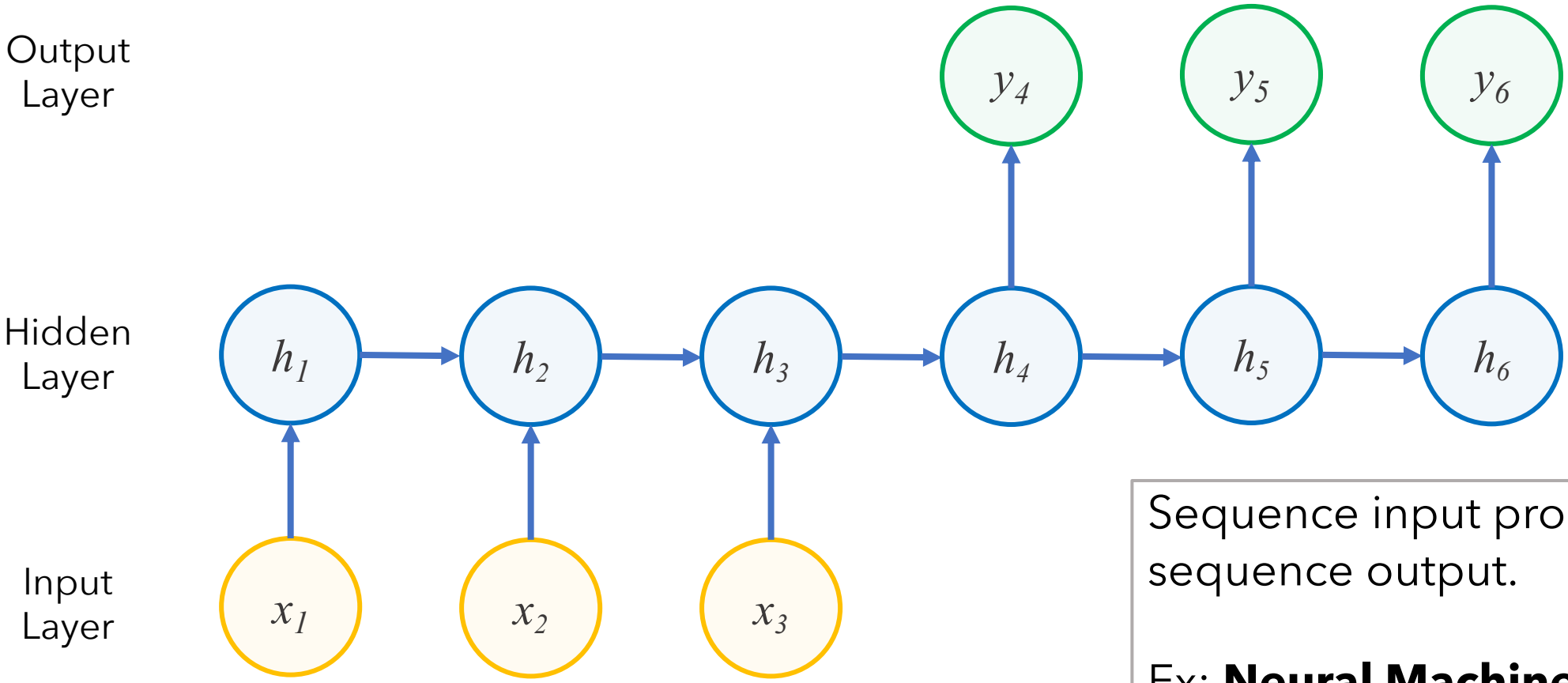


Input layer with multiple time steps producing a single output.

Ex: **Sentiment Analysis**

Sources: Andrej Karpathy

Examples of RNN Applications

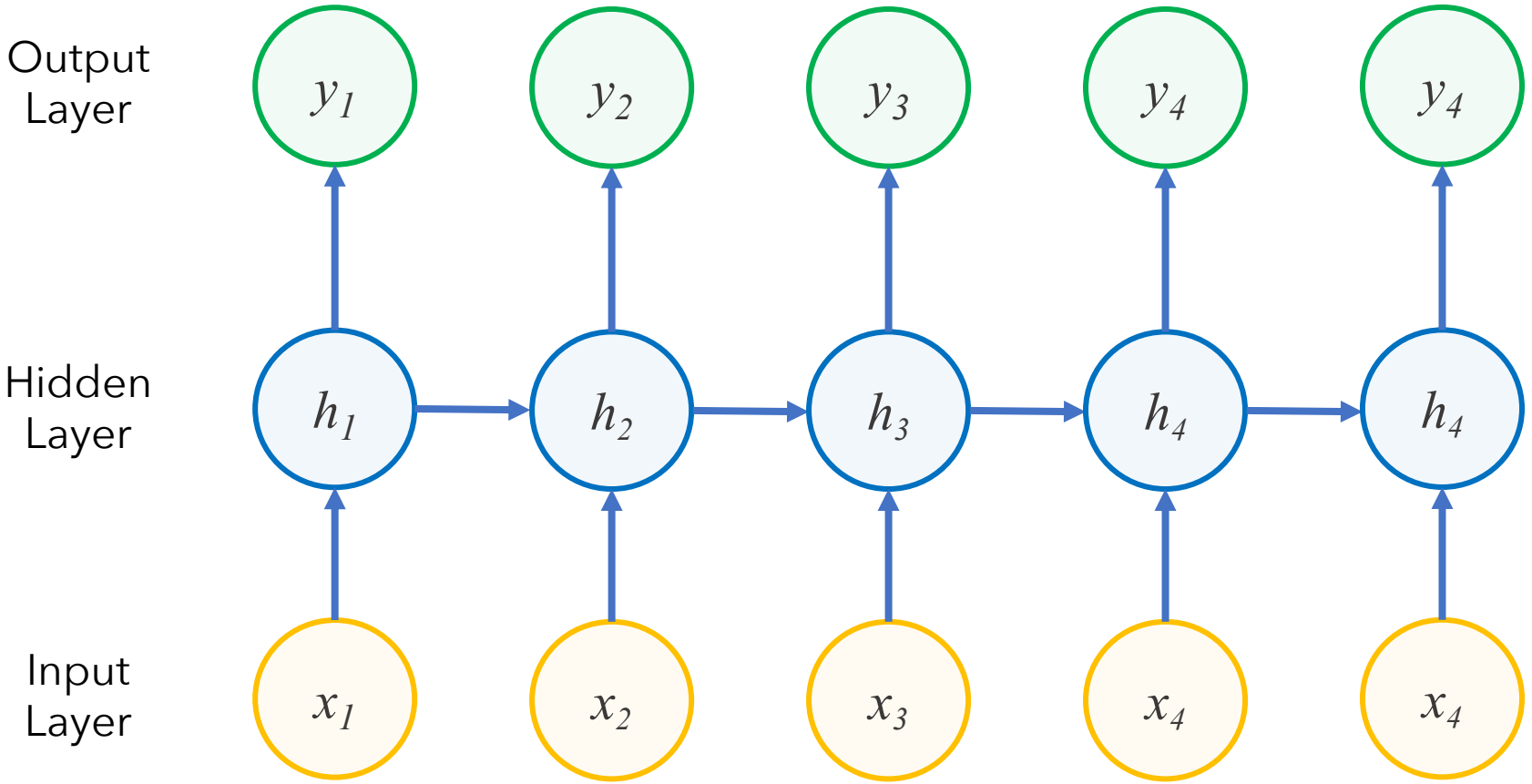


Sequence input produces a sequence output.

Ex: **Neural Machine Translation**

Sources: Andrej Karpathy

Examples of RNN Applications

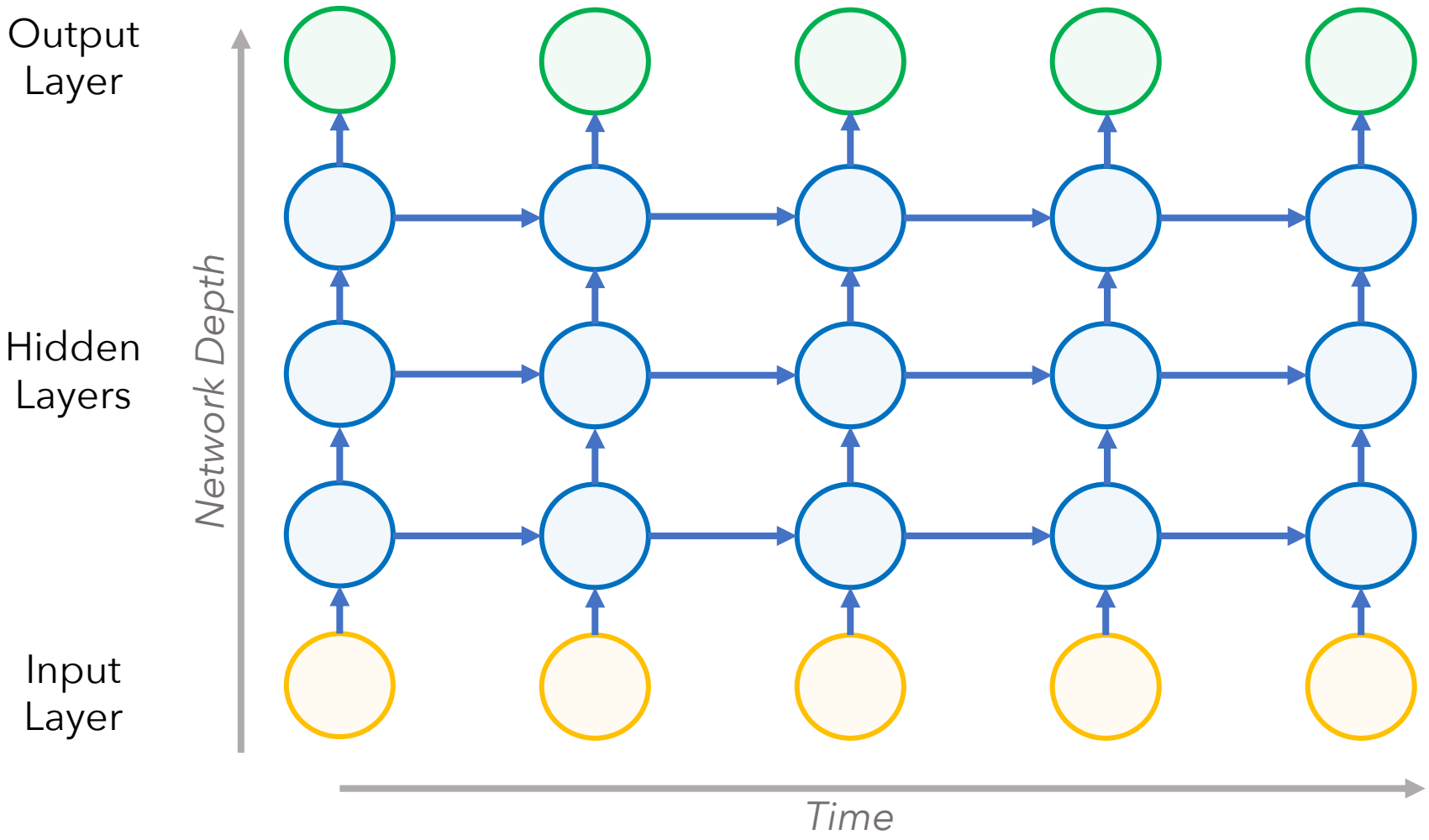


Synced sequence inputs and outputs

Ex: **Video Classification (frame by frame)**

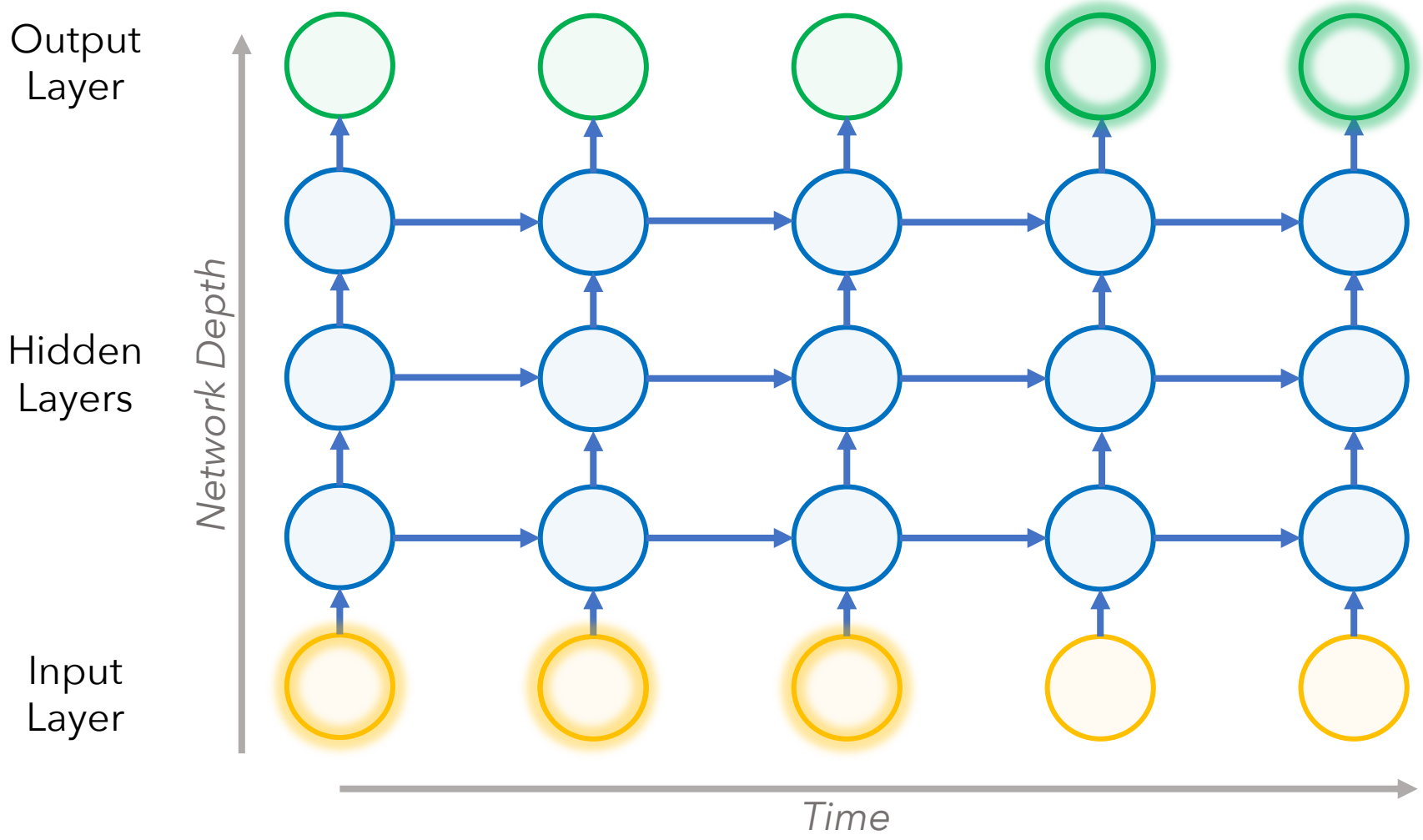
Sources: Andrej Karpathy

We can stack layers to make the model more robust



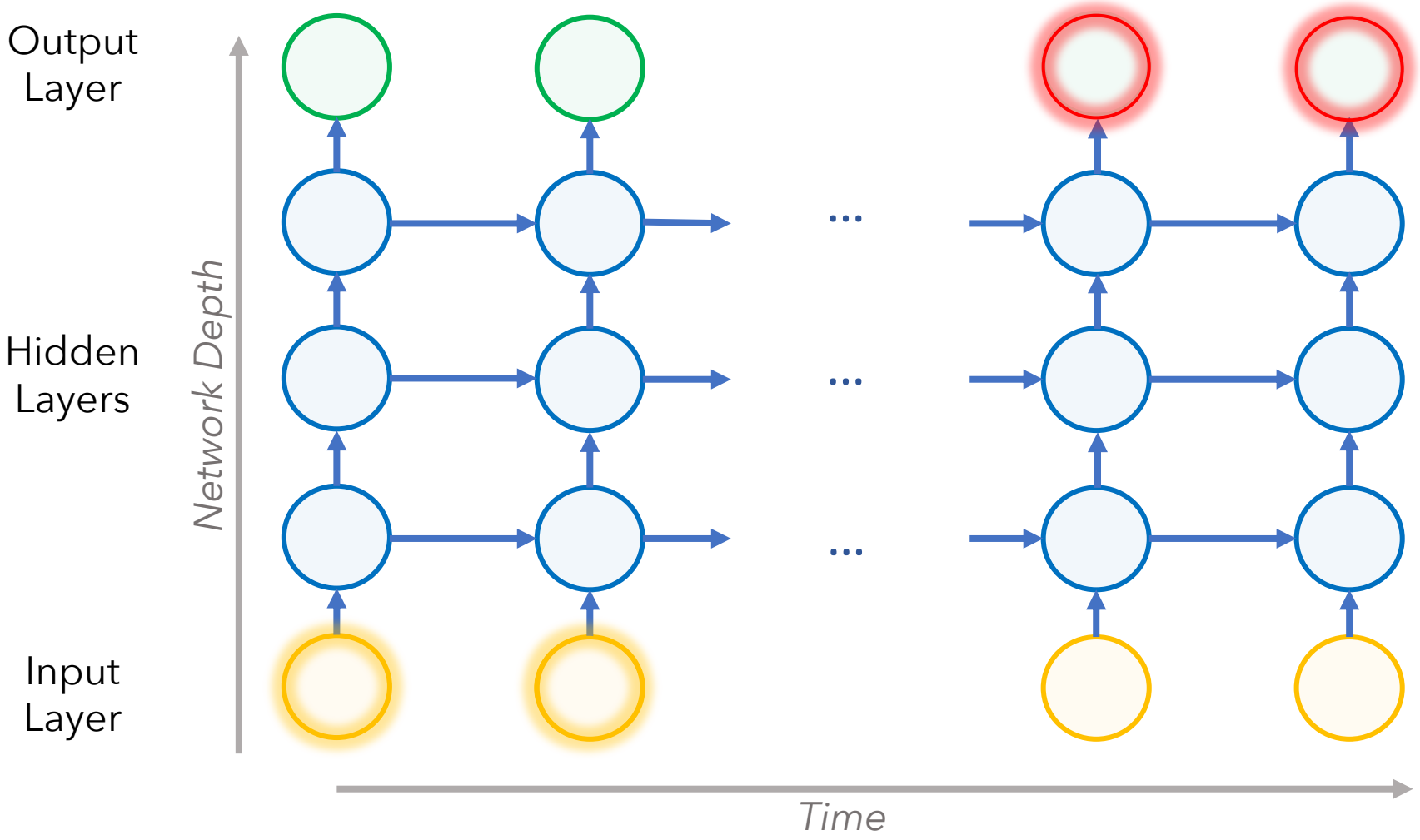
Sources: Andrej Karpathy

We can stack layers to make the model more robust



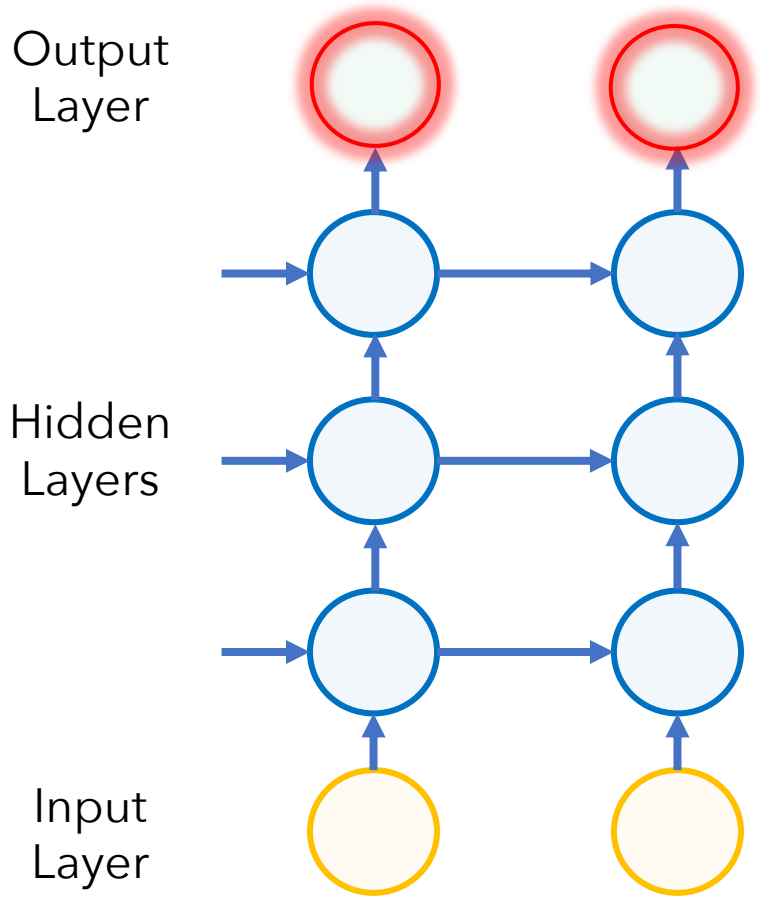
Sources: Andrej Karpathy

Long time sequences lead to vanishing gradients



Sources: Andrej Karpathy

Long time sequences lead to vanishing gradients



New Weight = **Weight** + (**learning rate** * **gradient**)

$$W_{t+1} = W + (\eta * \nabla f)$$

If ∇f is very small

$$5.50025 = 5.5 + (0.25 * 0.001)$$

The network wont learn anything with gradients which are too small!

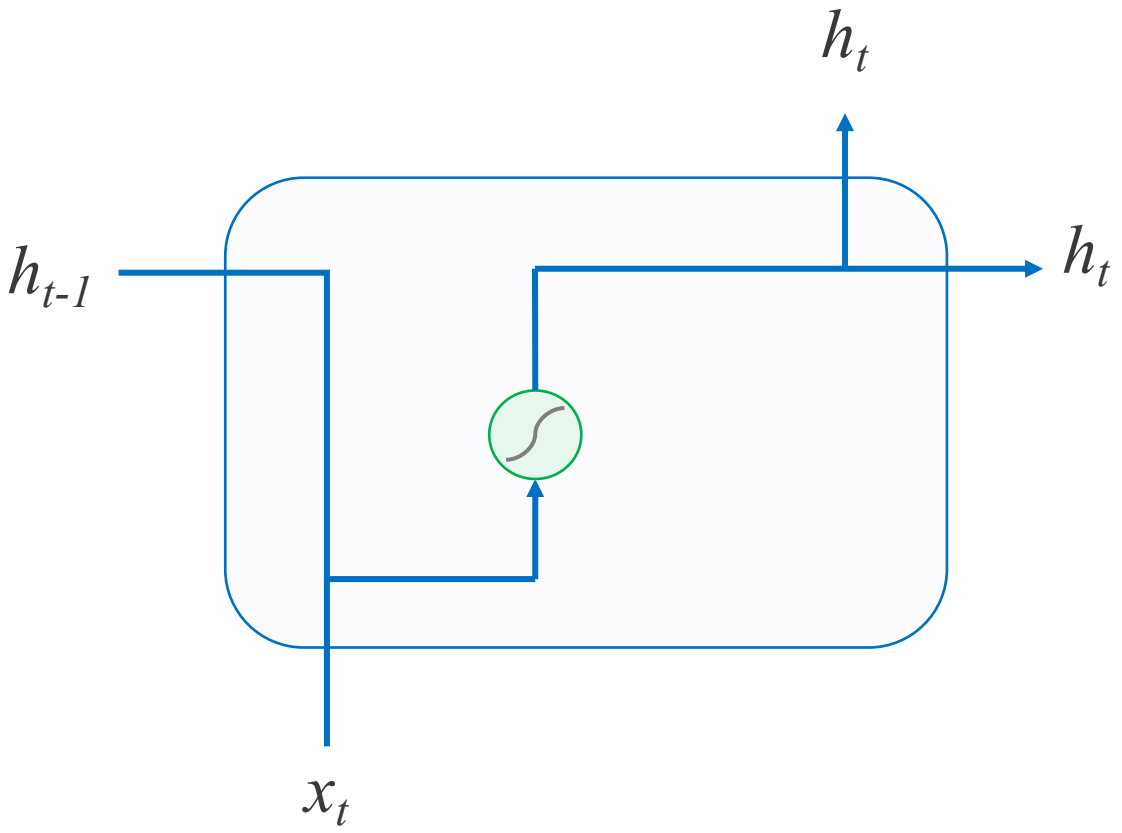
Sources: Andrej Karpathy

Gated Recurrent Neural Networks

Review of gated neural networks and the solution to vanishing gradients

Solution: Gated Recurrent Neural Networks!

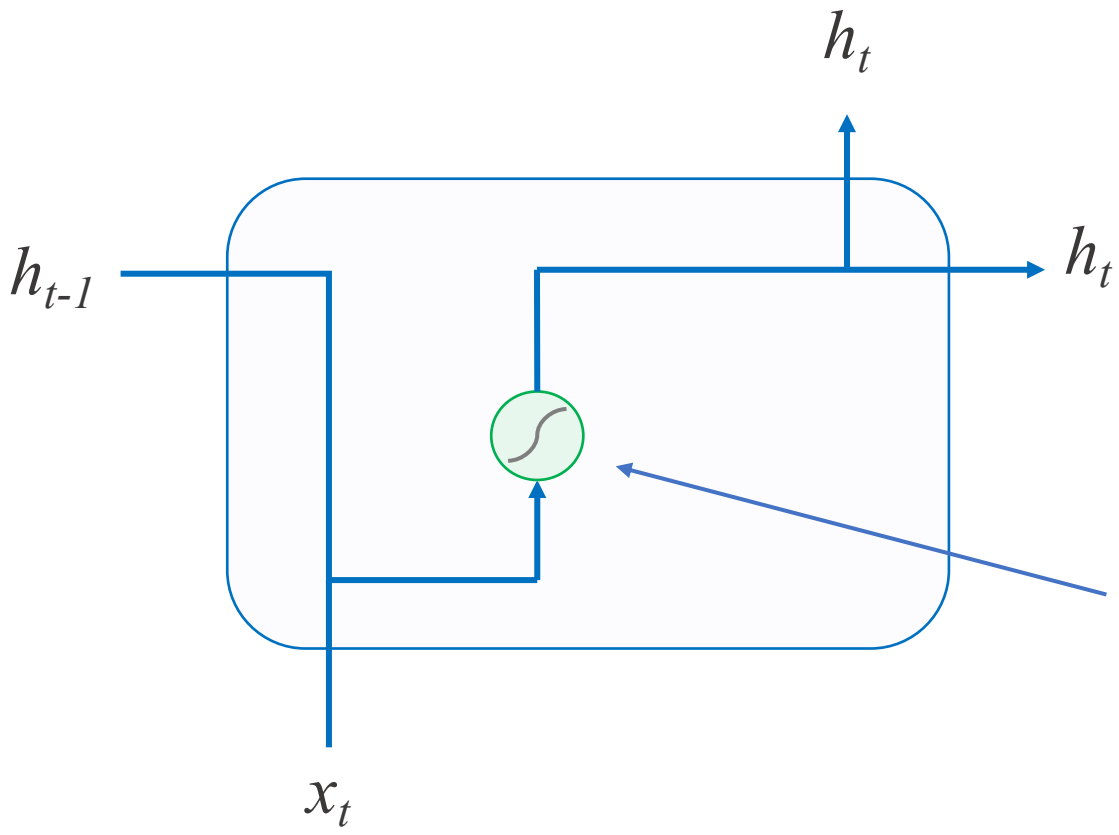
Vanilla RNN – Hidden Cell



Sources: Christopher Olah

Solution: Gated Recurrent Neural Networks!

Vanilla RNN – Hidden Cell

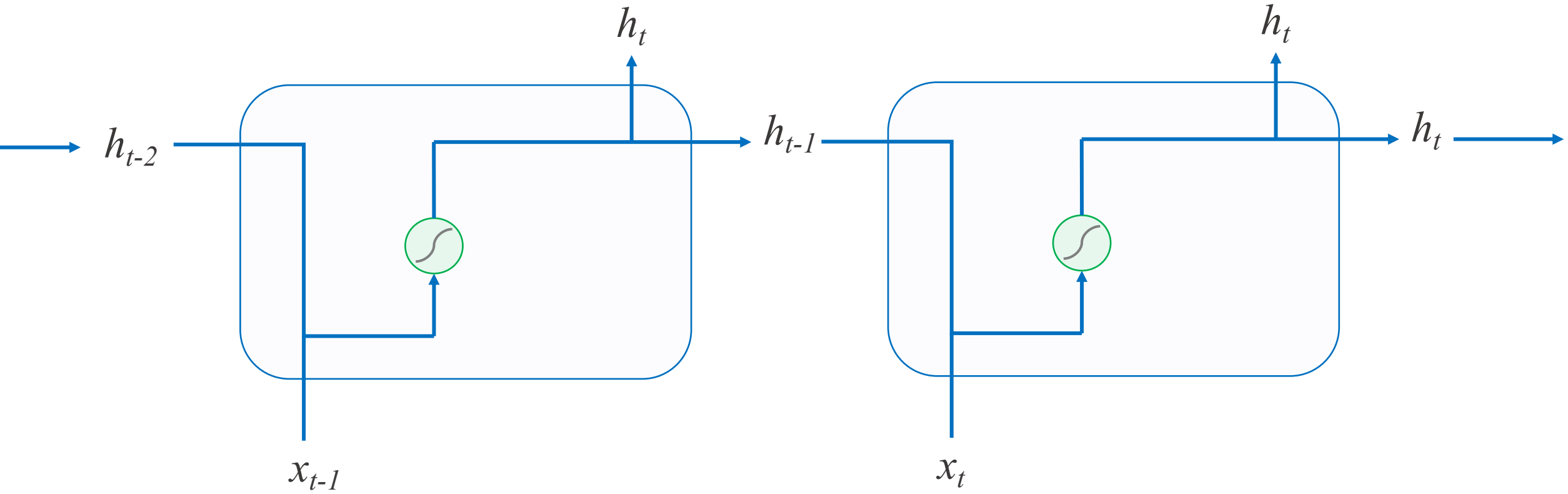


$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Sources: Christopher Olah

Solution: Gated Recurrent Neural Networks!

These cells are connected to calculate each new parameter value via stochastic gradient descent

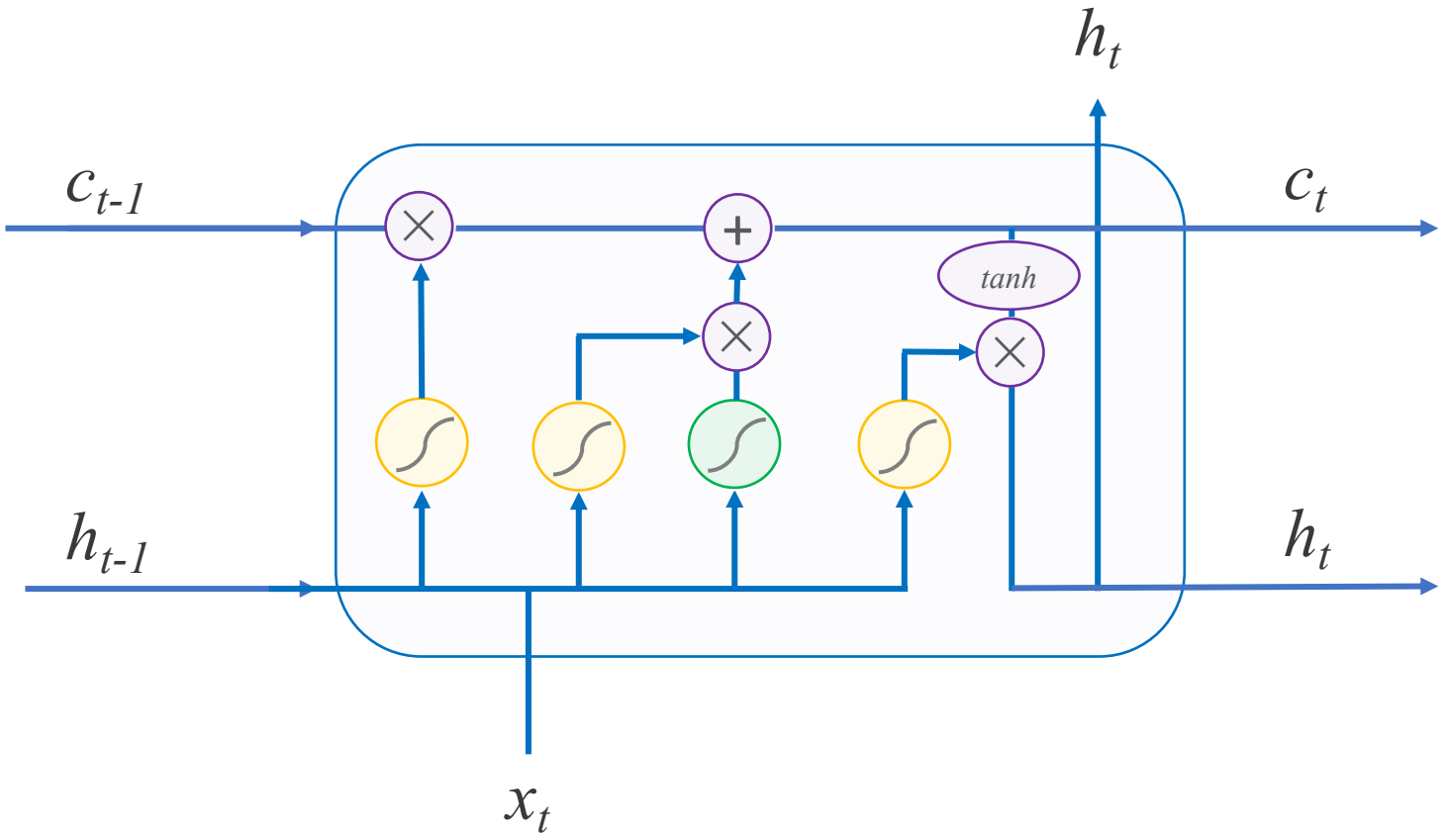


Gradients

Sources: Christopher Olah

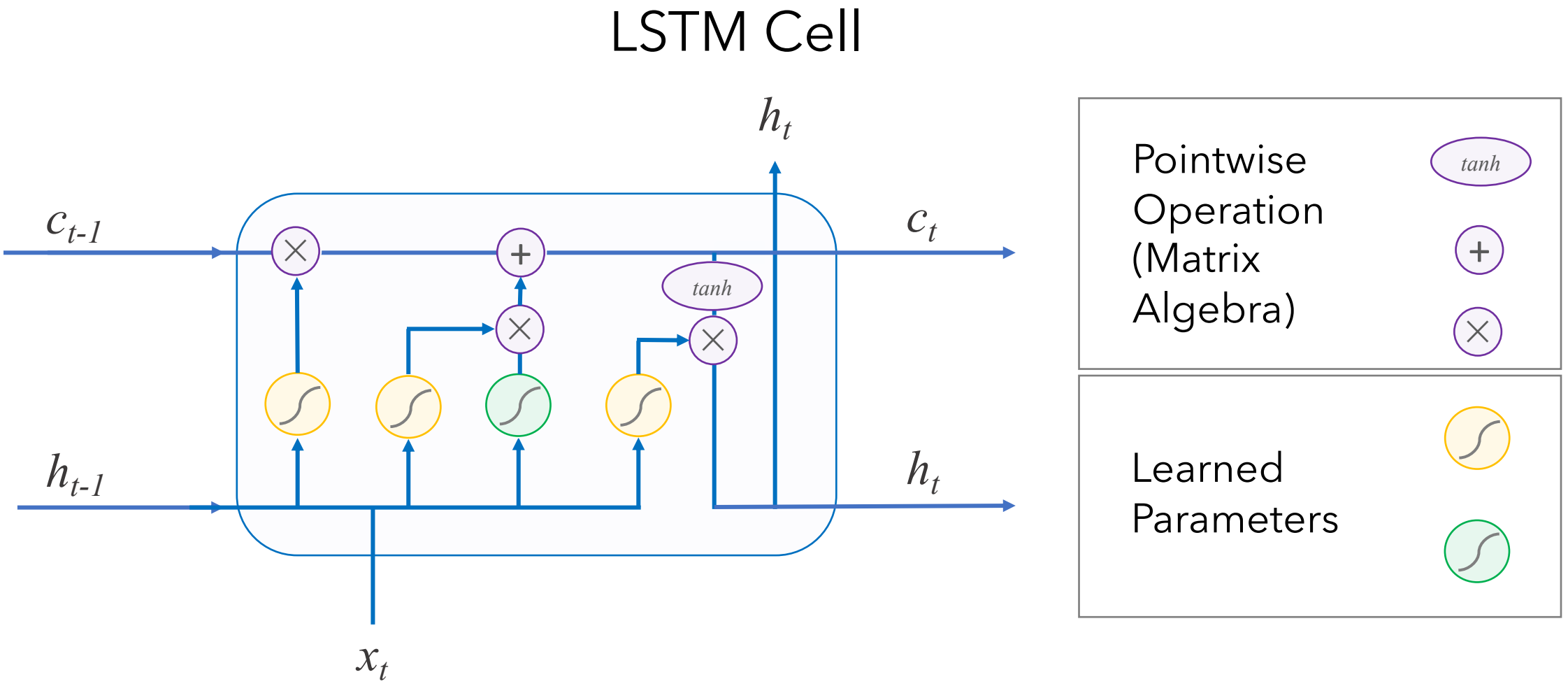
Solution: Gated Recurrent Neural Networks!

LSTM Cell



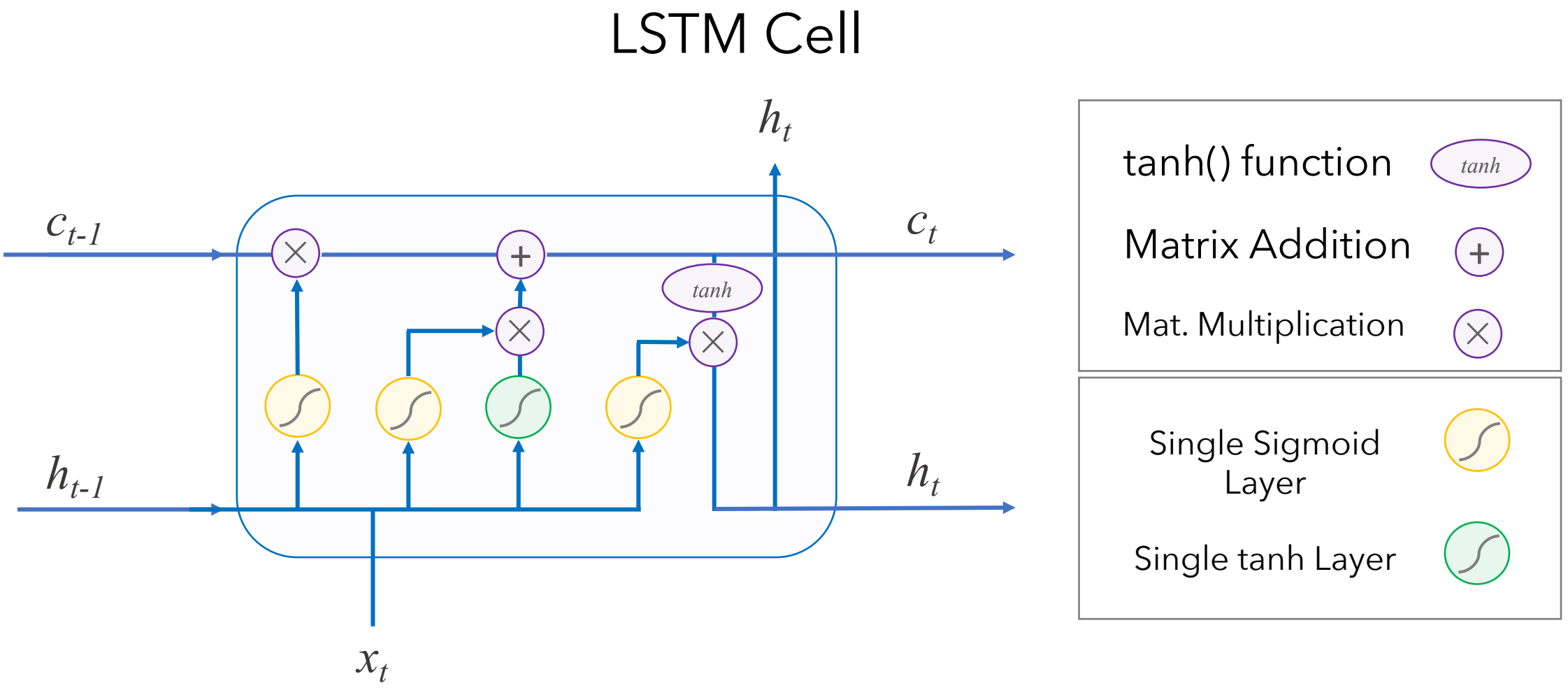
Sources: Christopher Olah

Solution: Gated Recurrent Neural Networks!



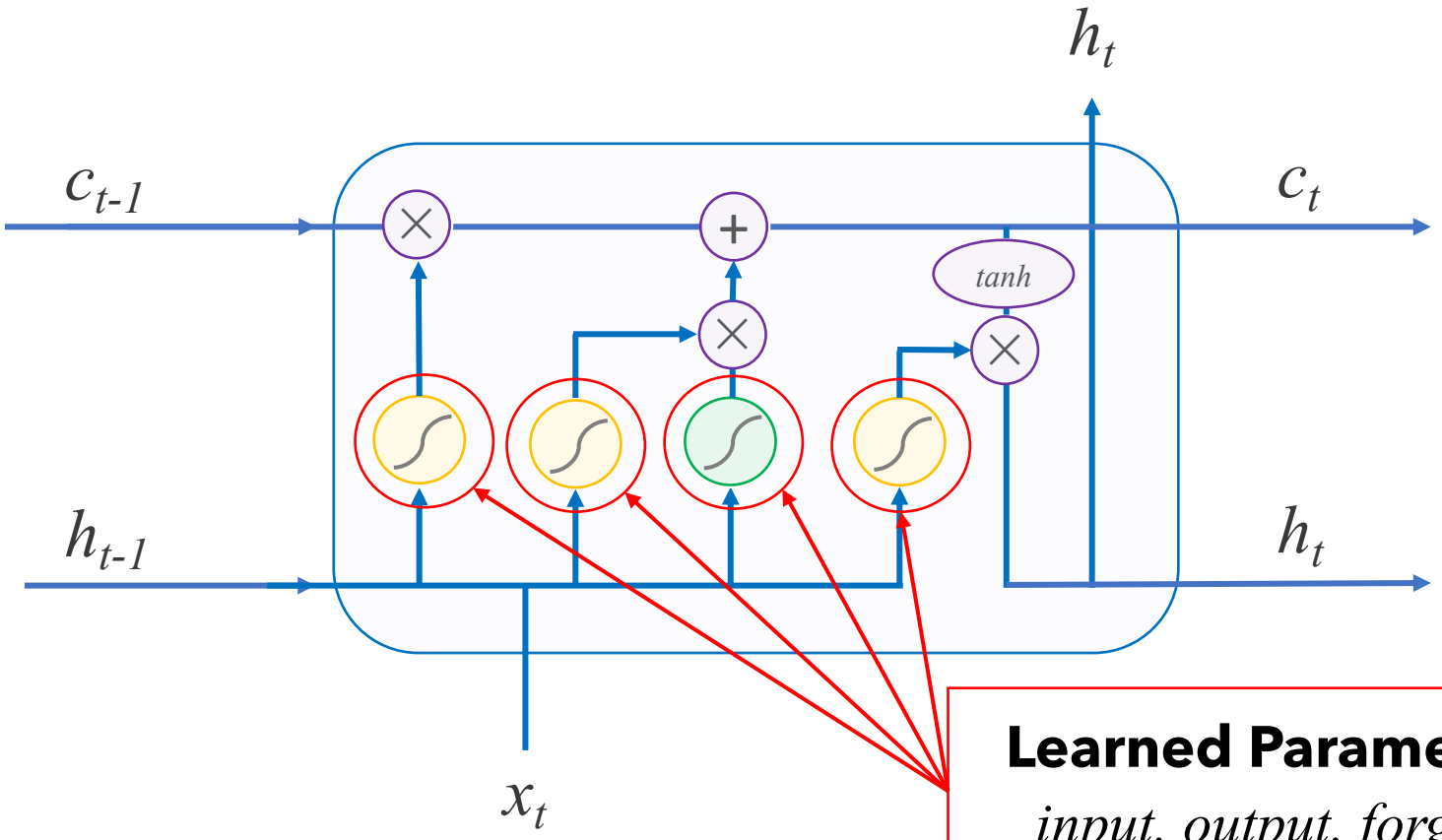
Sources: Christopher Olah

Solution: Gated Recurrent Neural Networks!



Sources: Christopher Olah

I,f,o,g gates are learned by the network through training



Input

Output

Forget

g

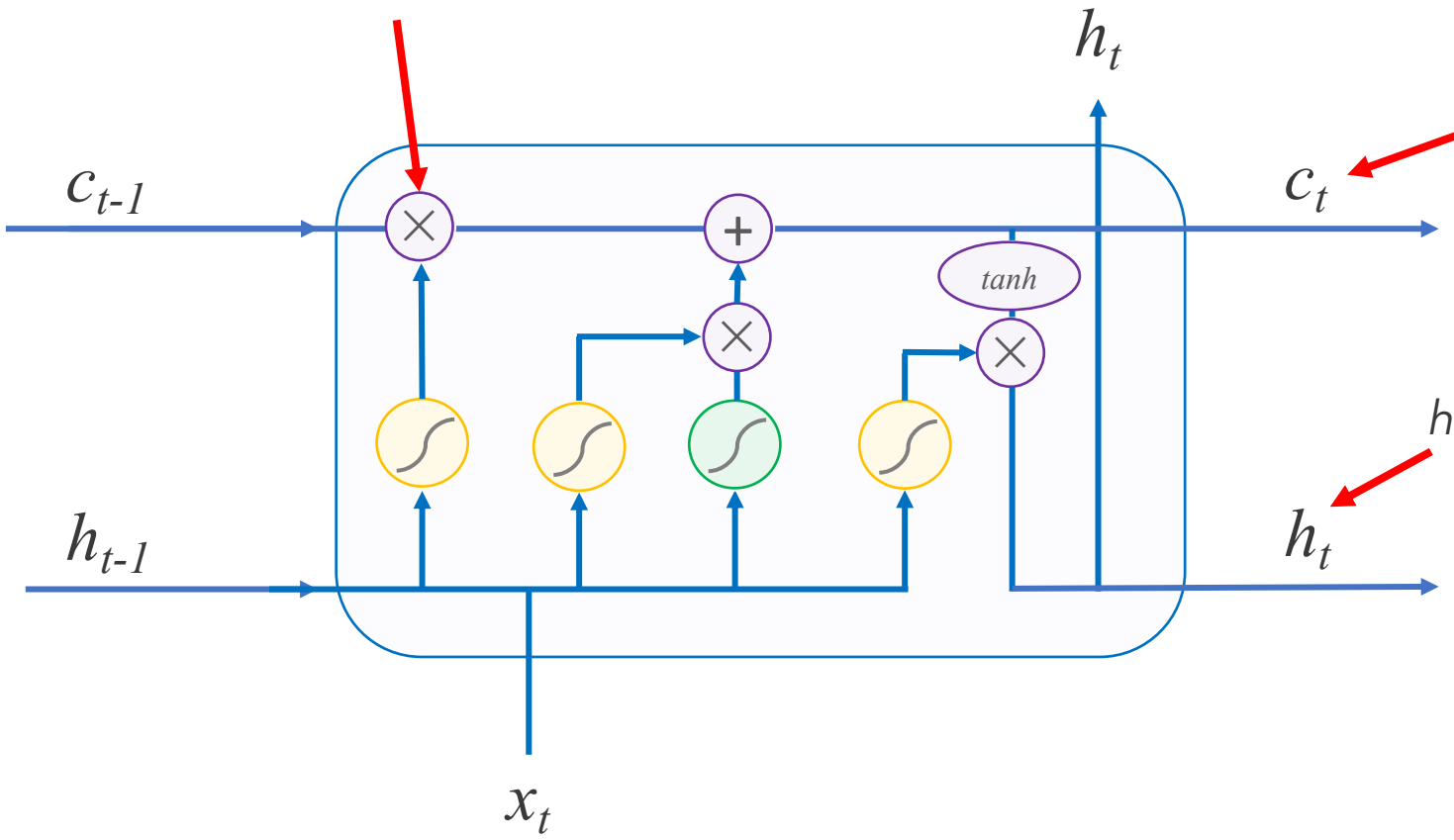
Learned Parameters:

input, output, forget, g

Sources: Christopher Olah

Example of IFOG in natural language processing

Forget gate forgets the plurality of the subject after a period. (Periods trigger forget gates a lot)

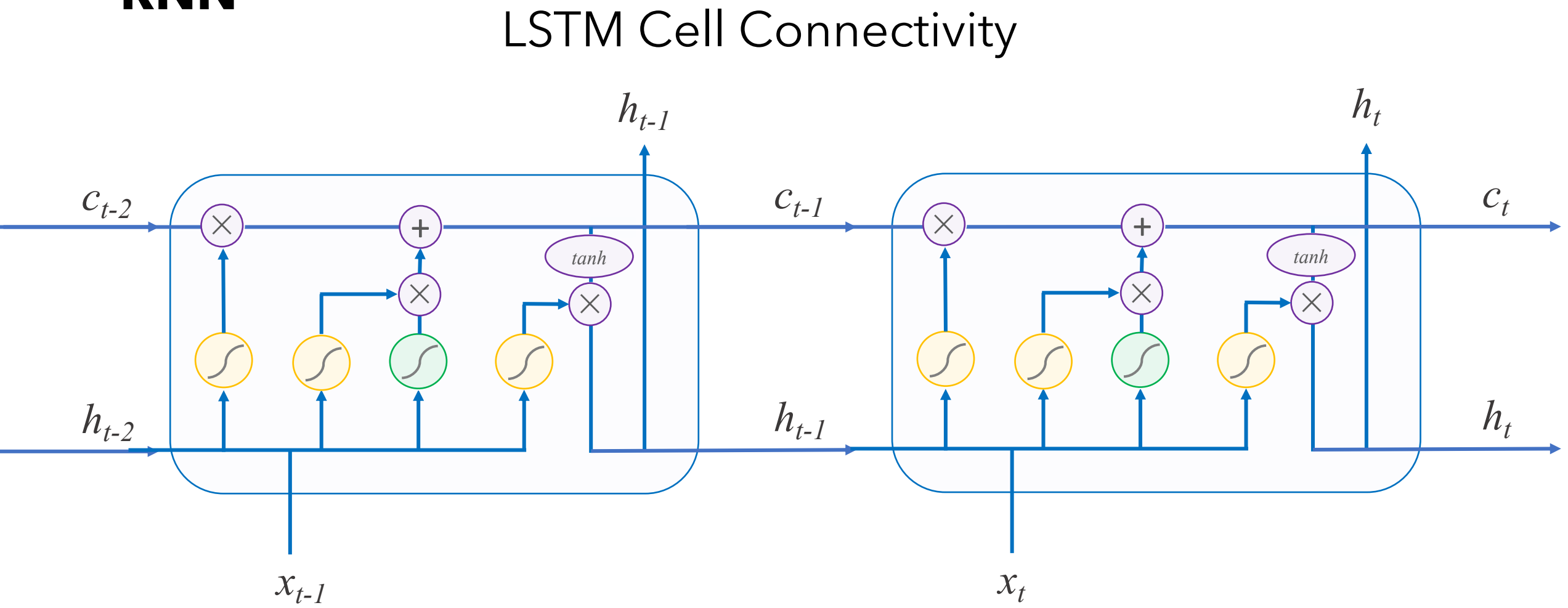


c_t could represent the meaning

h_t could represent the plurality of the subject

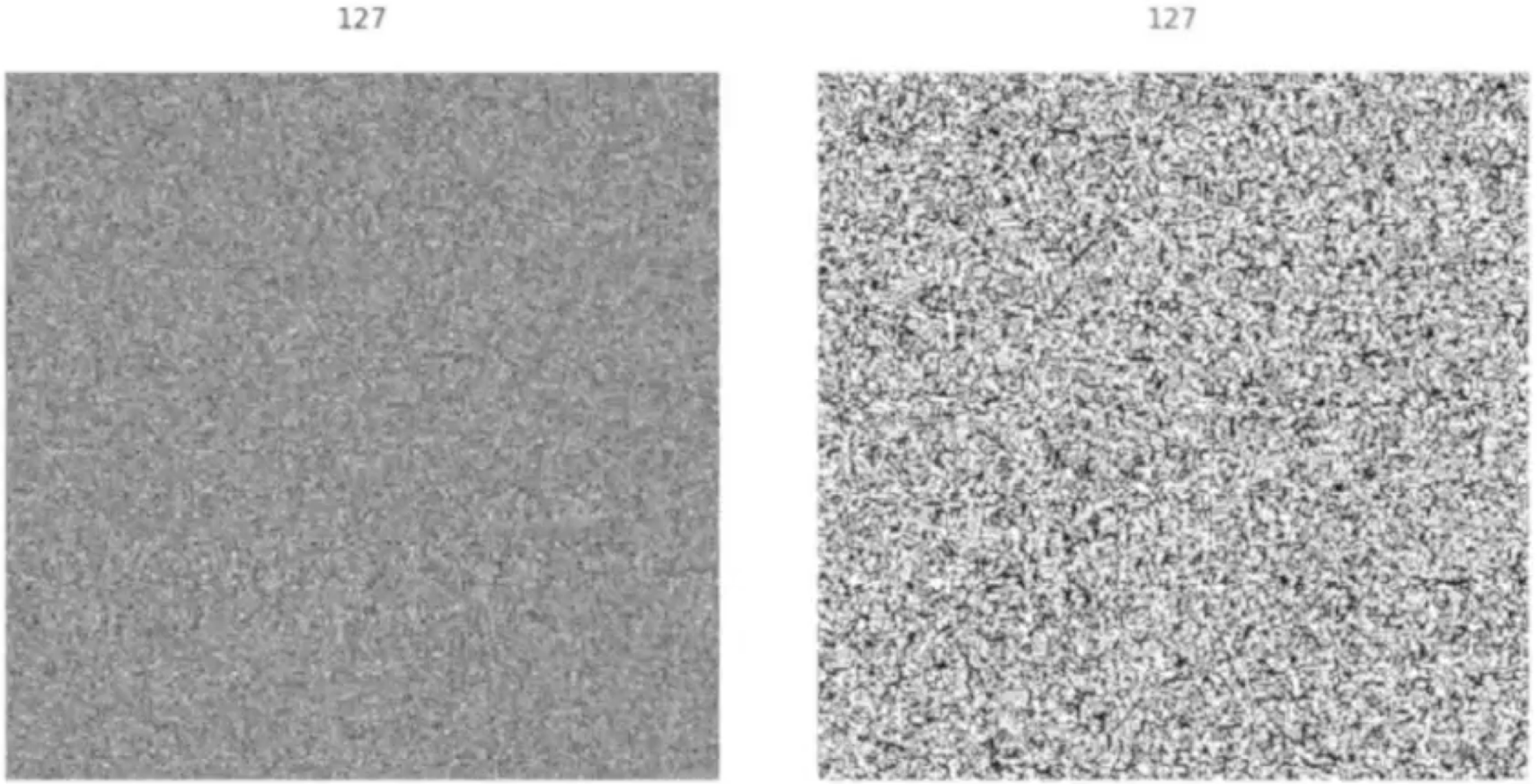
Sources: Christopher Olah

LSTM units stack similarly to hidden units in a traditional RNN



Sources: Christopher Olah

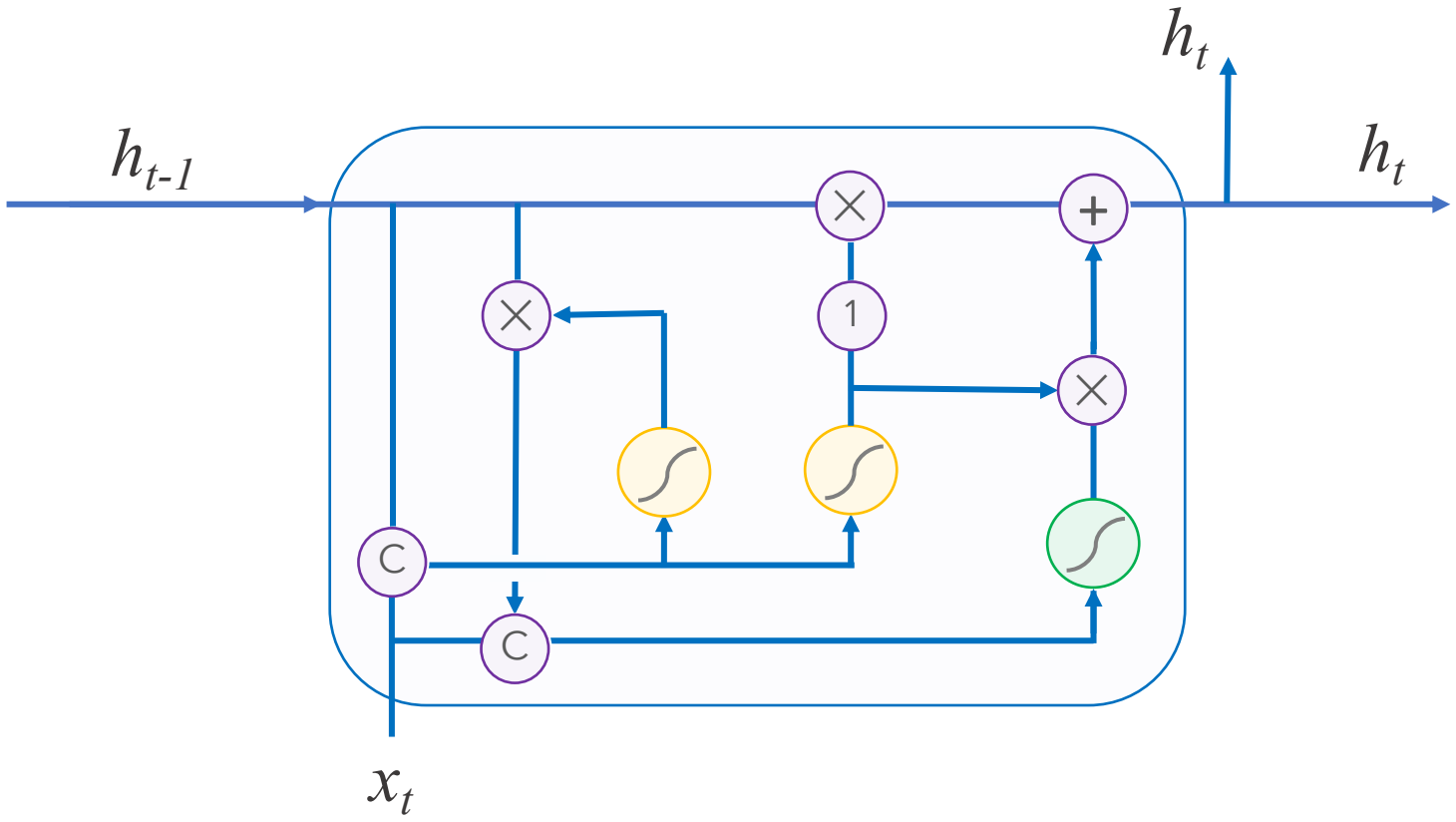
LSTM layers prevent vanishing gradients



Sources: imgur.com/gallery/vaNahKE

Other types of gated recurrent neural networks: GRU

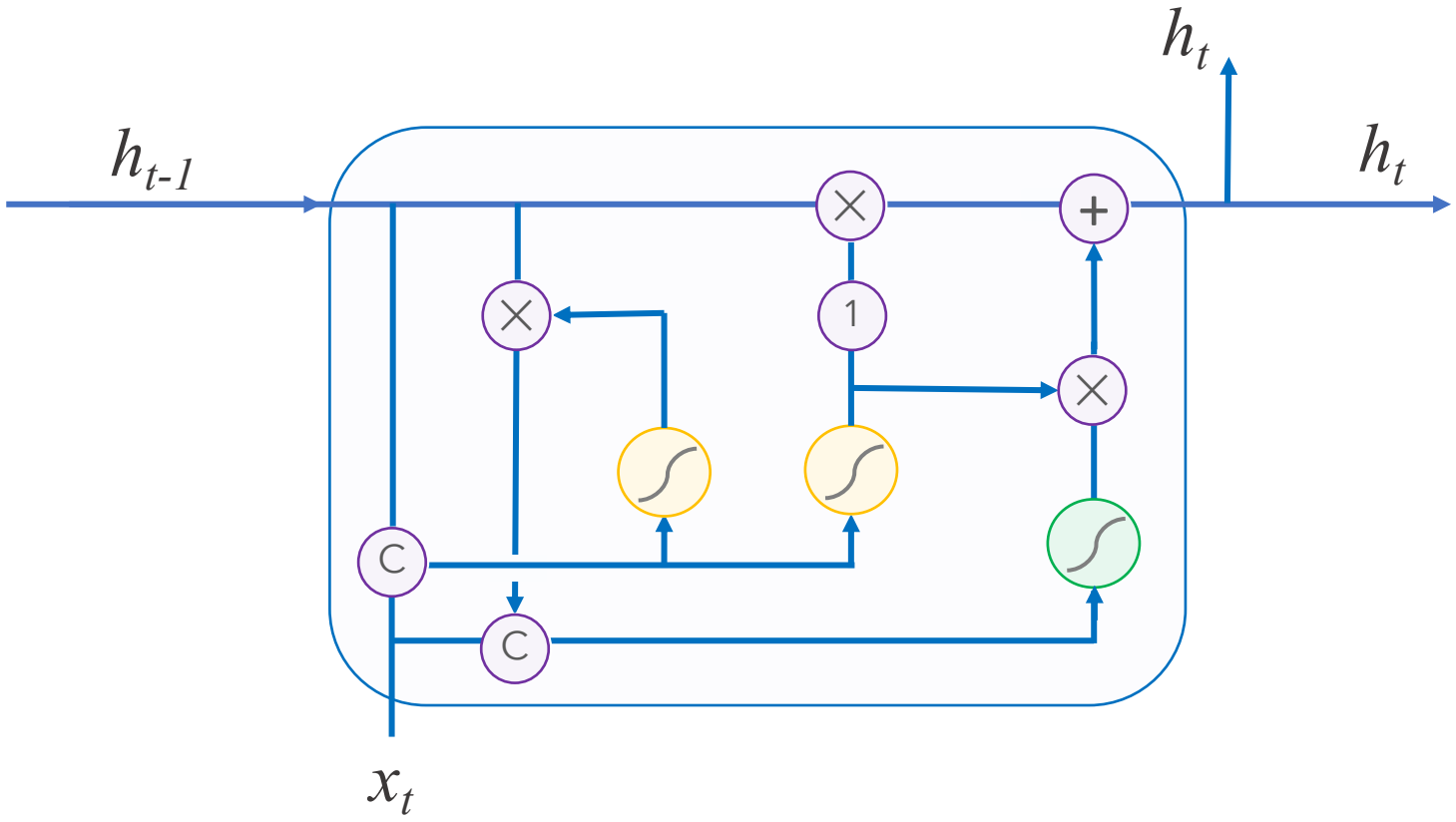
Gated Recurrent Unit



Sources: Christopher Olah

Other types of gated recurrent neural networks: GRU

Gated Recurrent Unit



Inverse 1

Concatenation C

Matrix Addition +

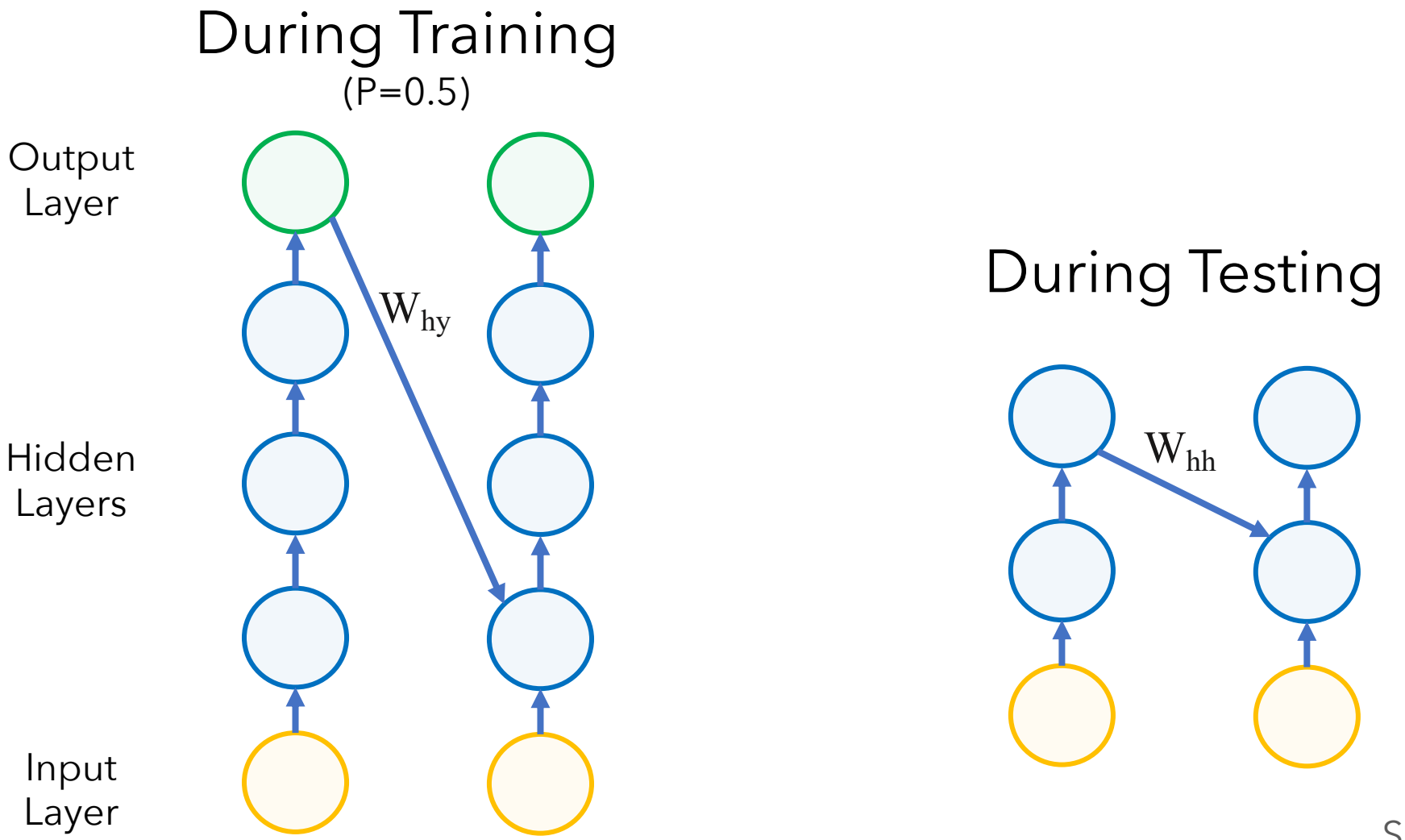
Mat. Multiplication ×

Single Sigmoid Layer

Single tanh Layer

Sources: Christopher Olah

Teacher Forcing and Output Recurrence

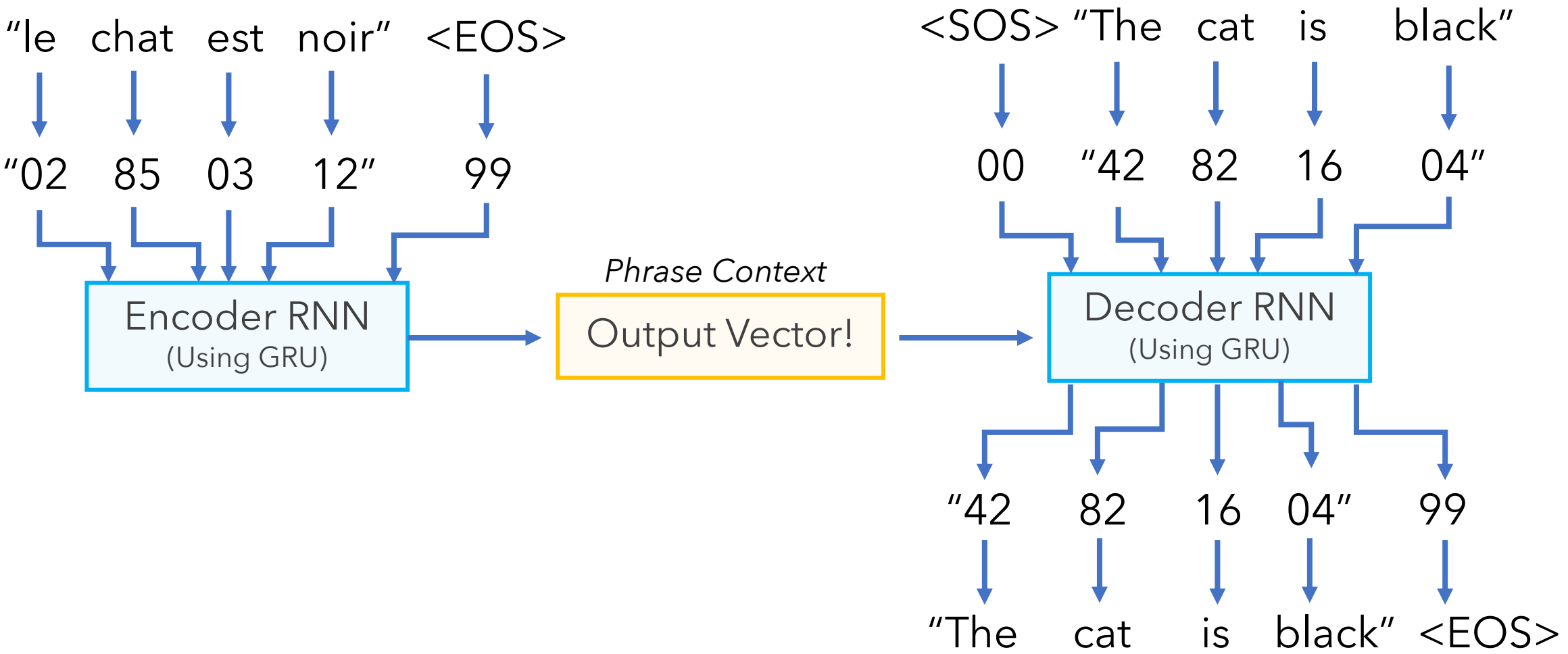


Sources: Yoshua Bengio

Machine Translation

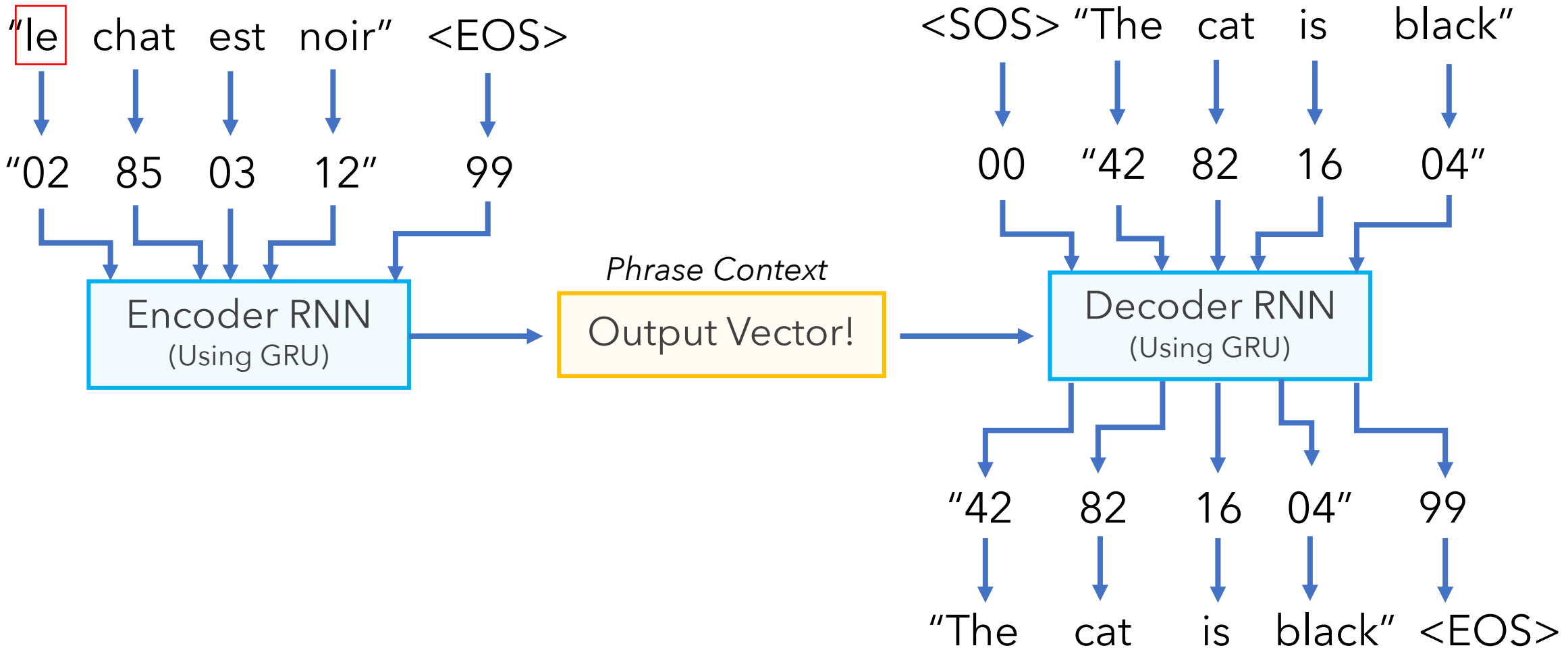
Using encoder-decoder models for machine translation

Encoder – Decoder Model



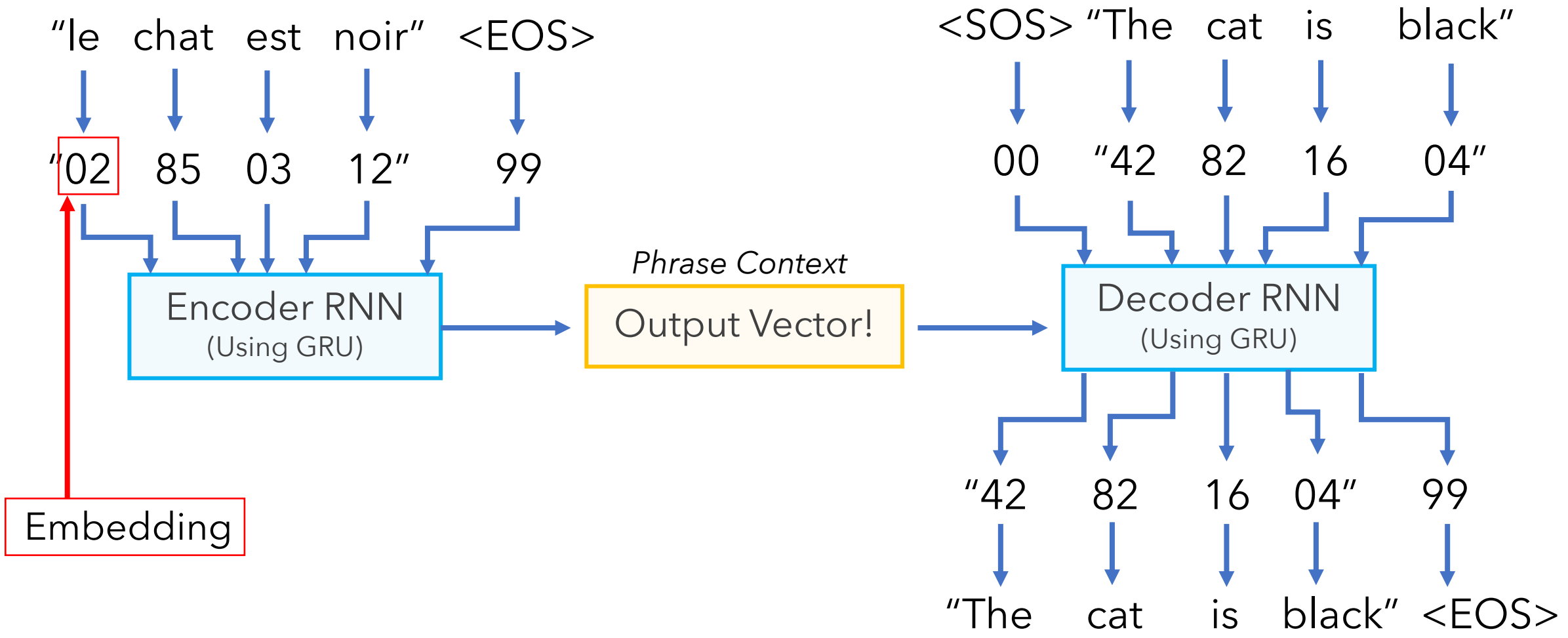
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



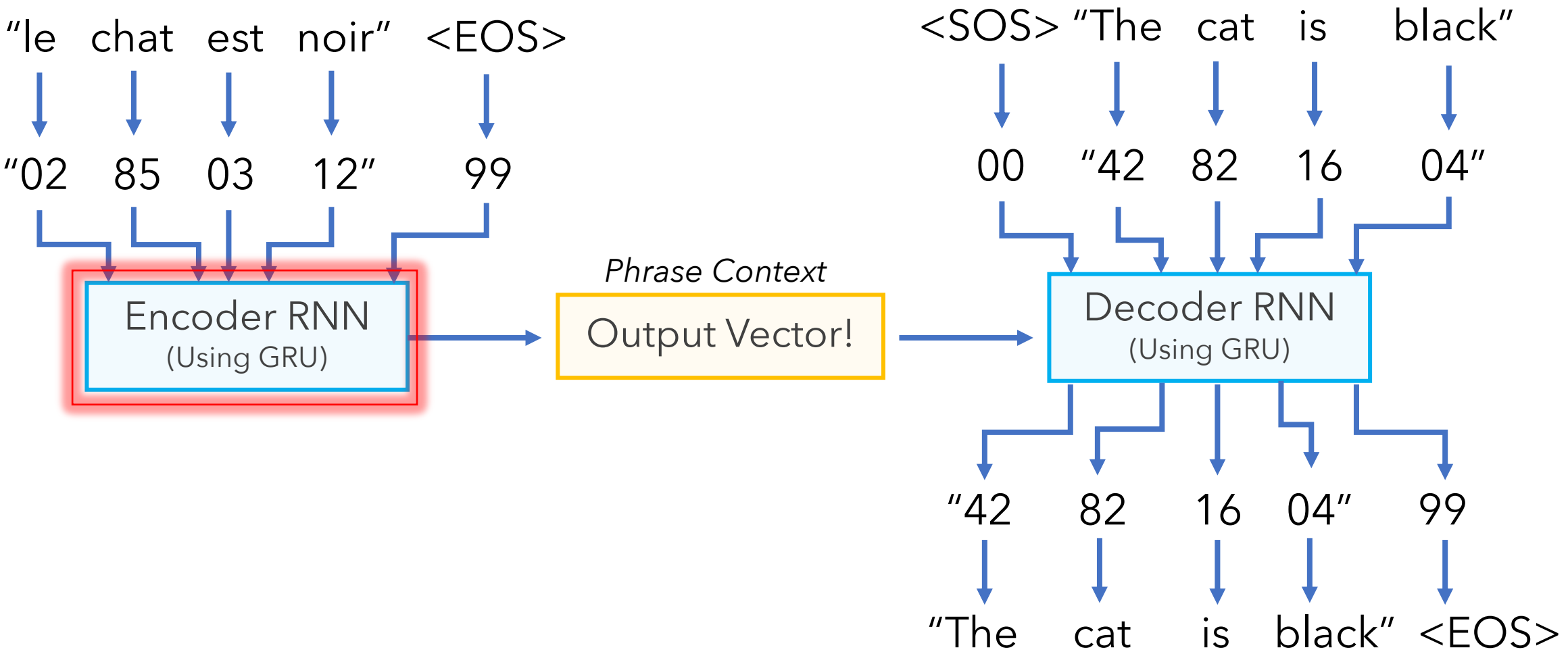
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



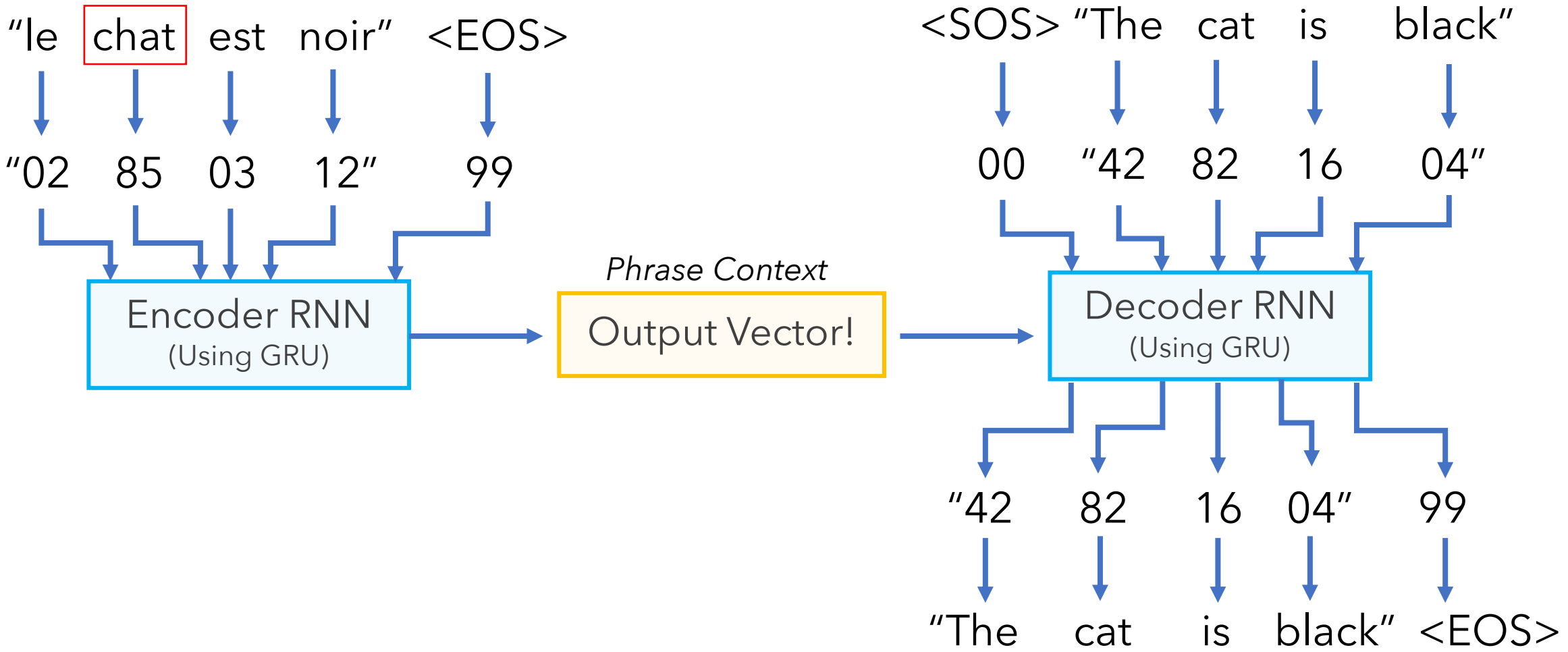
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



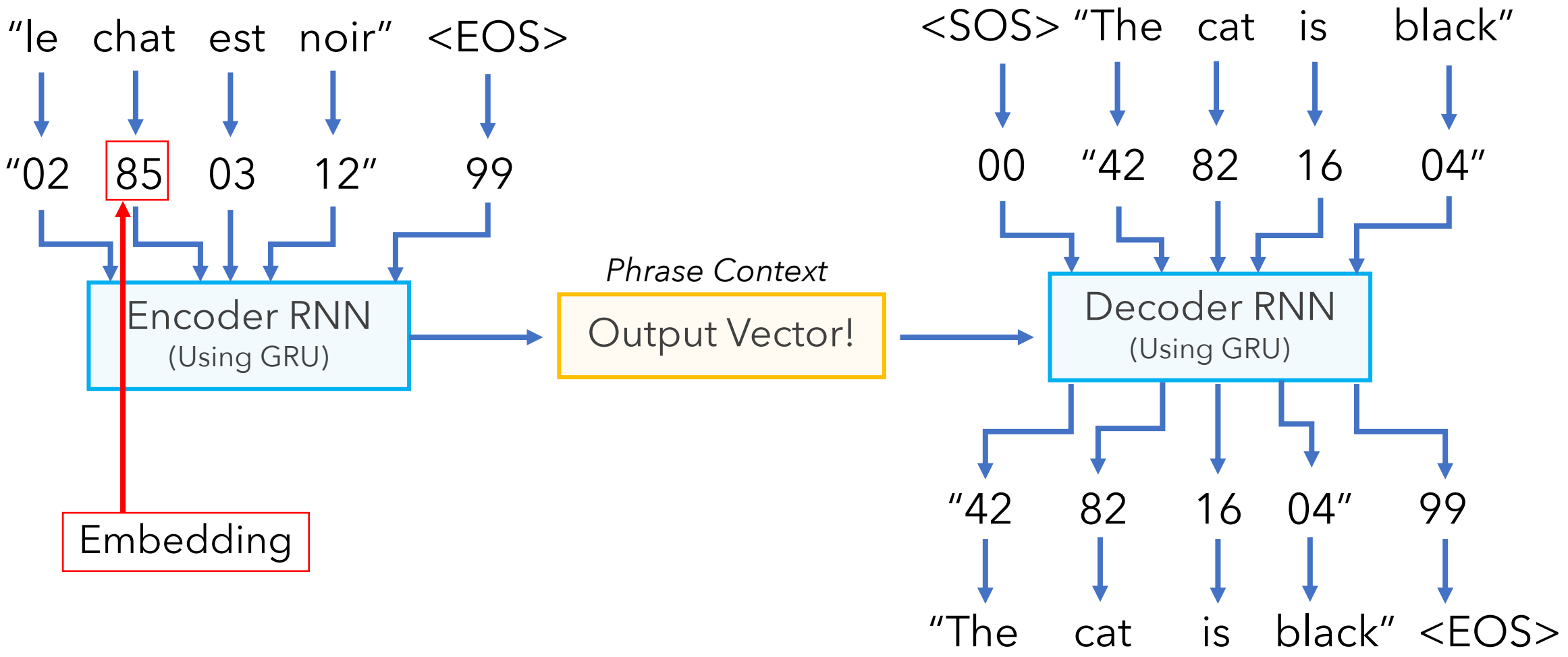
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



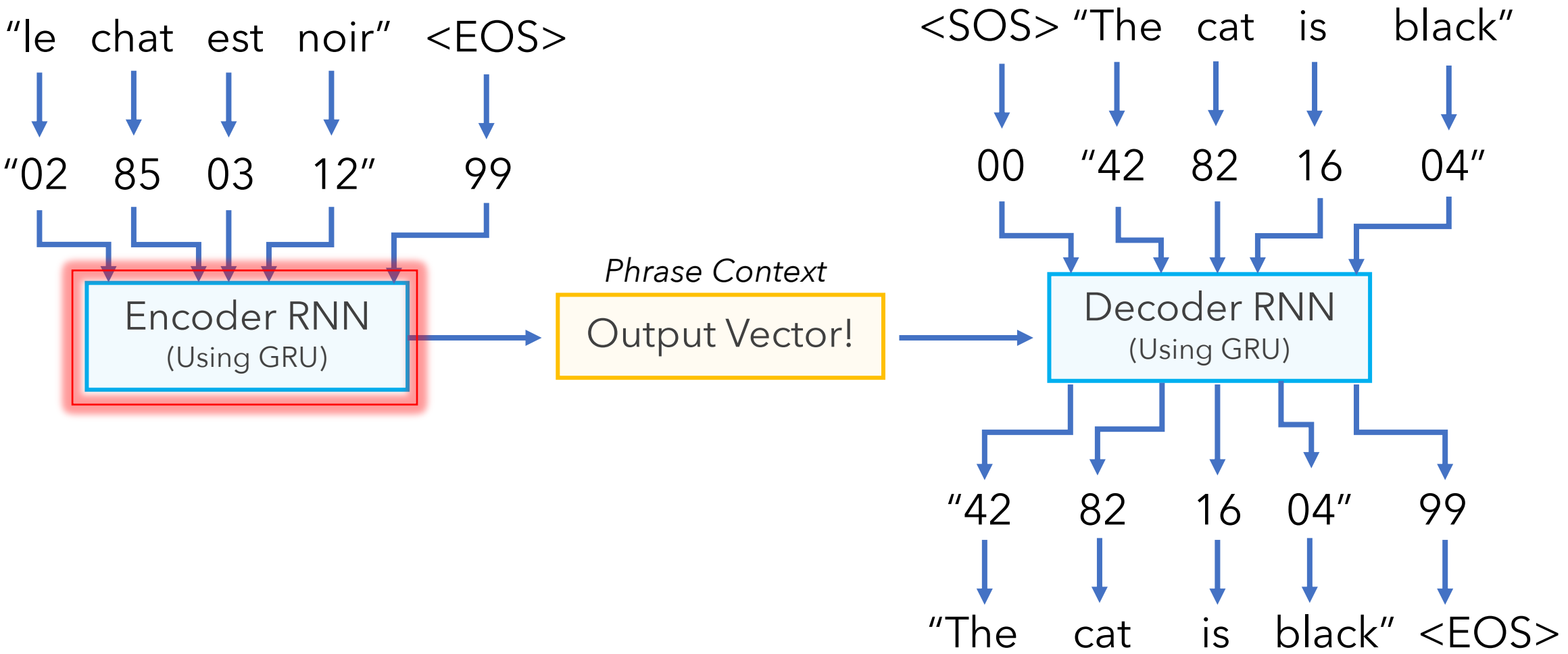
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



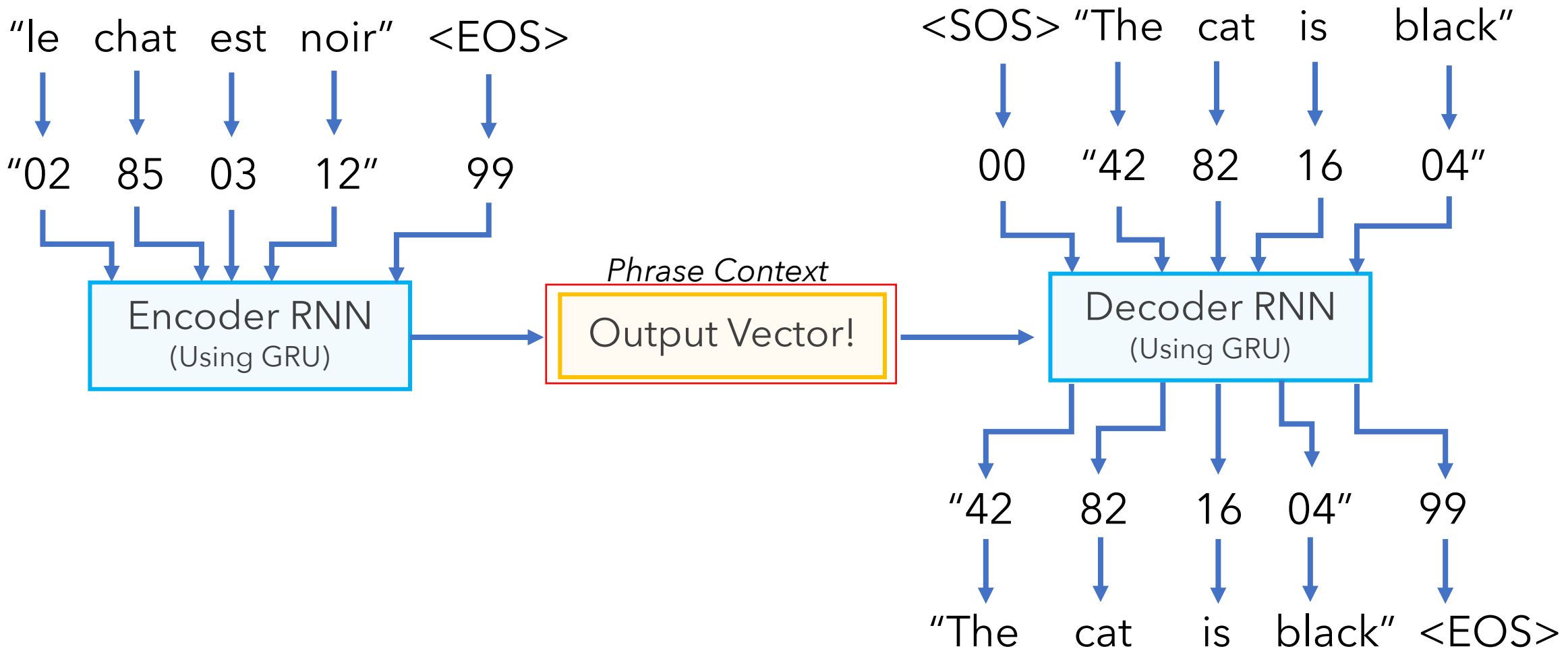
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



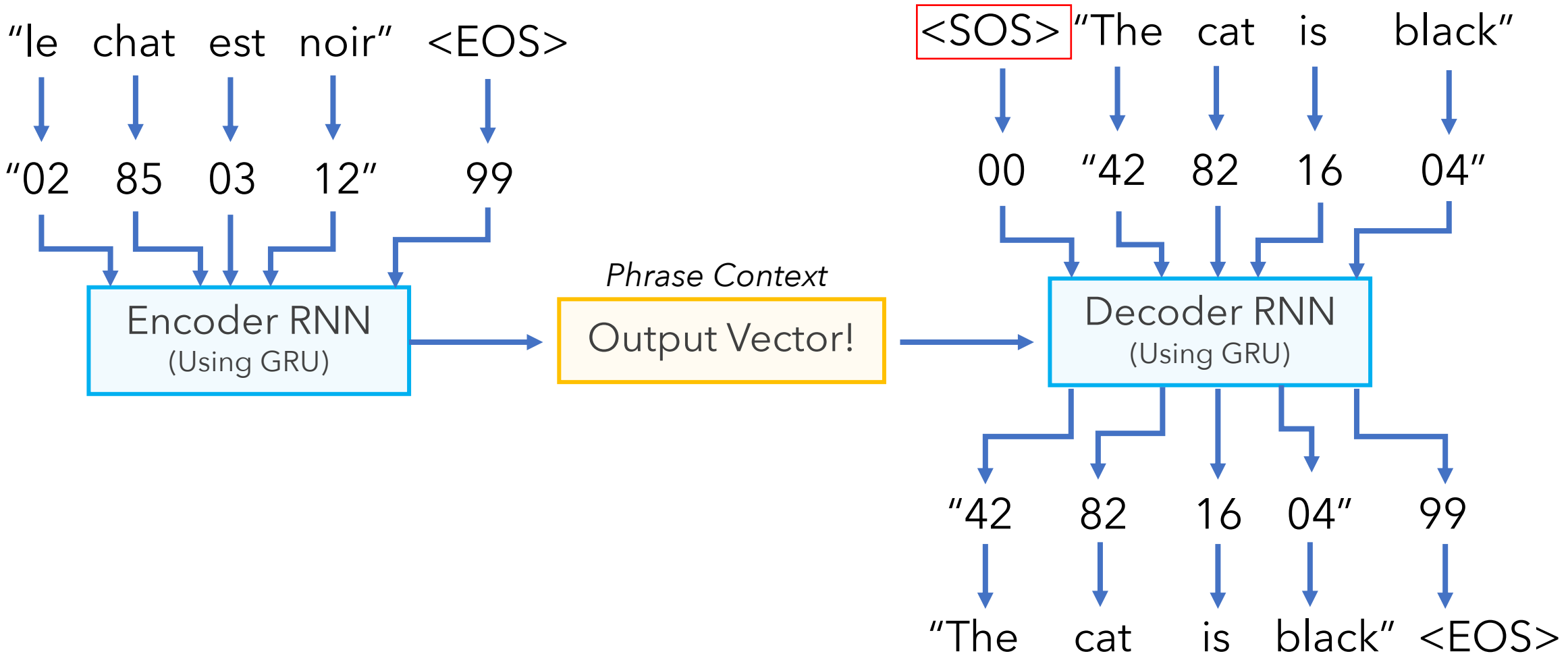
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



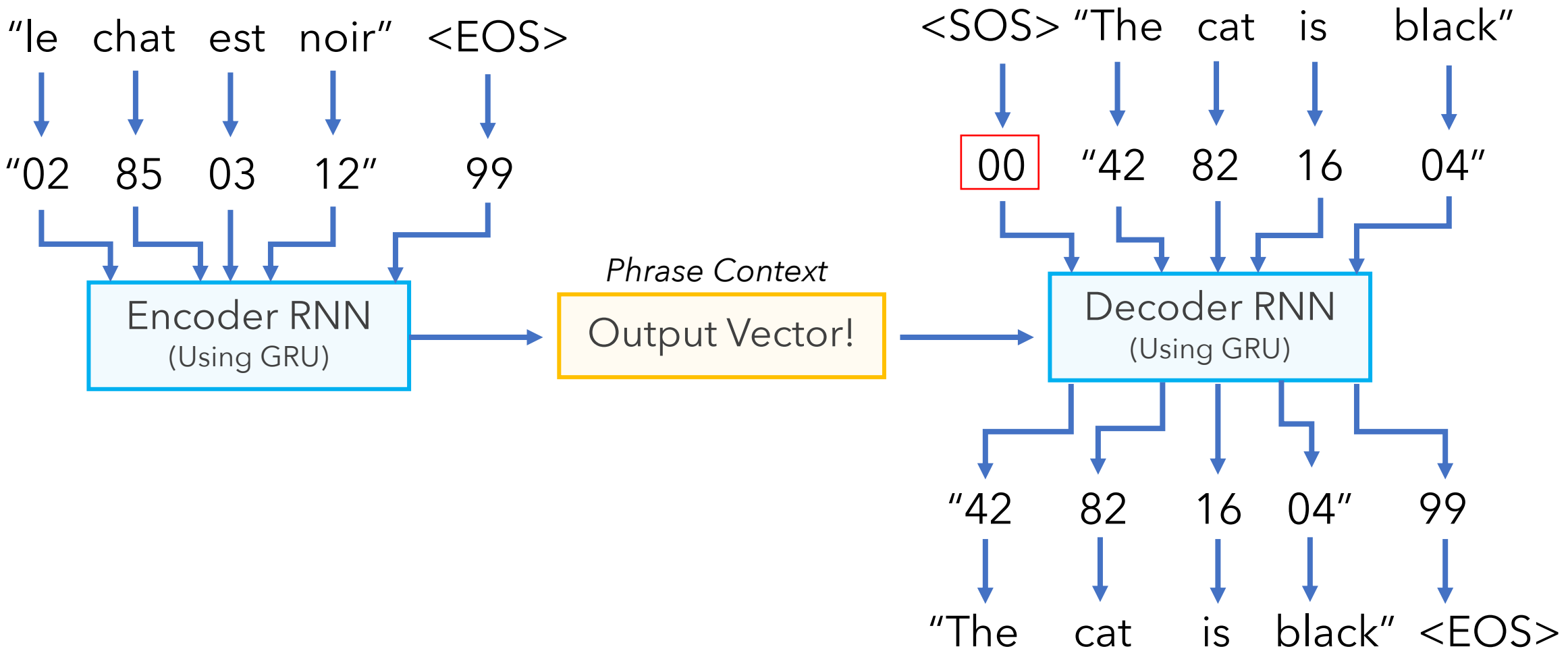
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



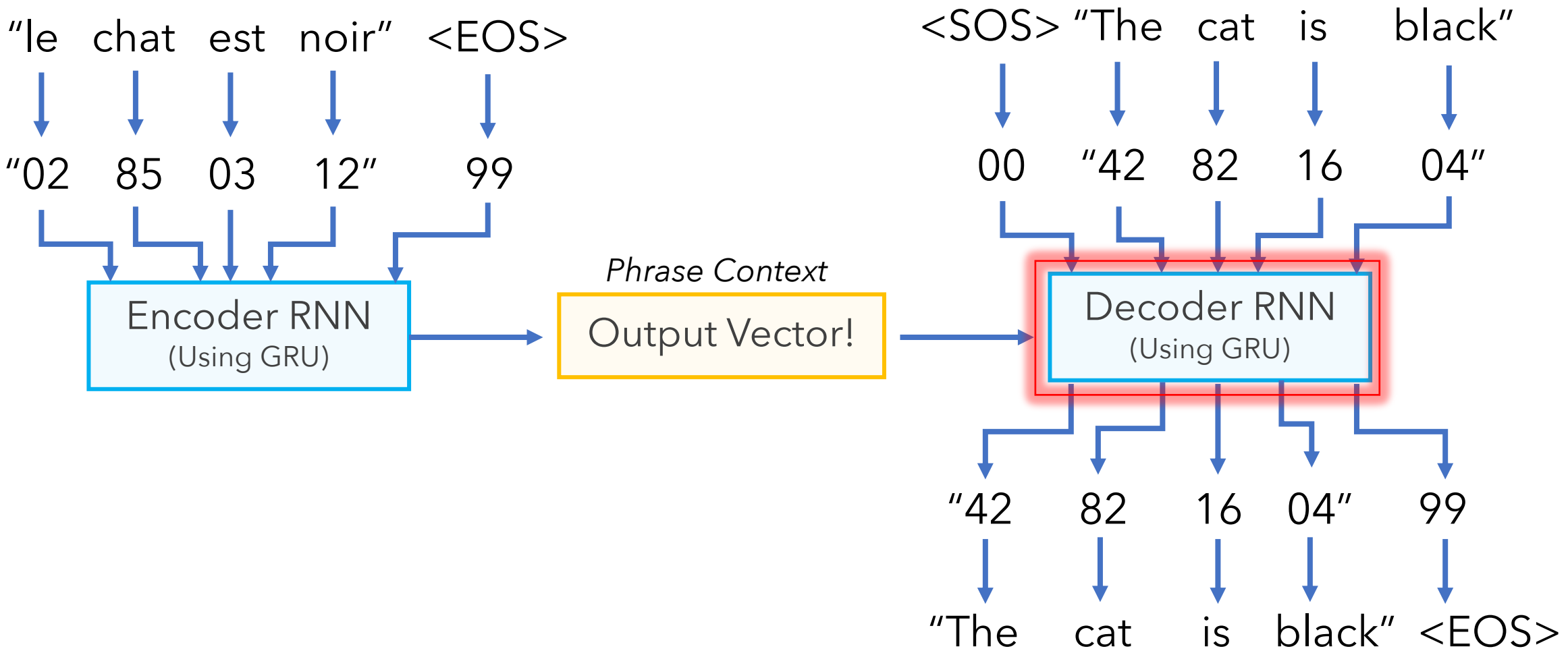
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



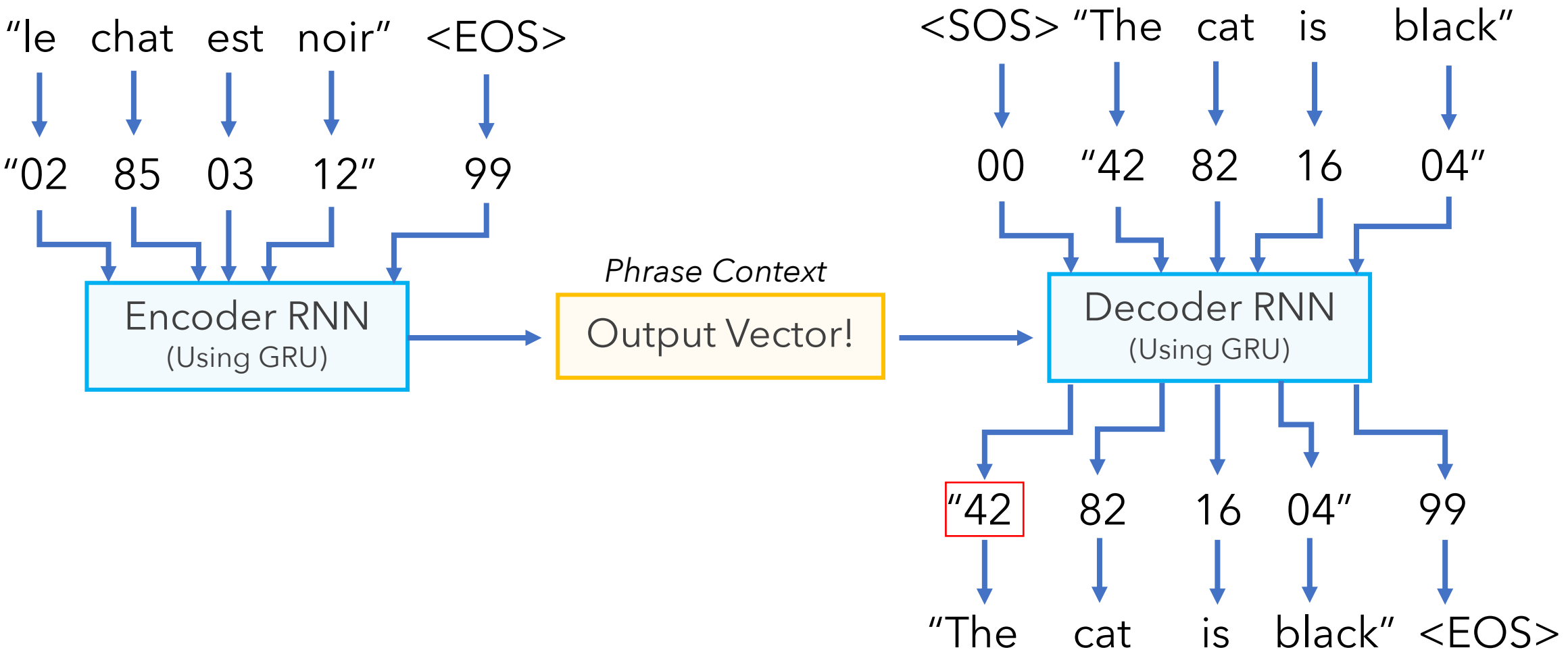
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



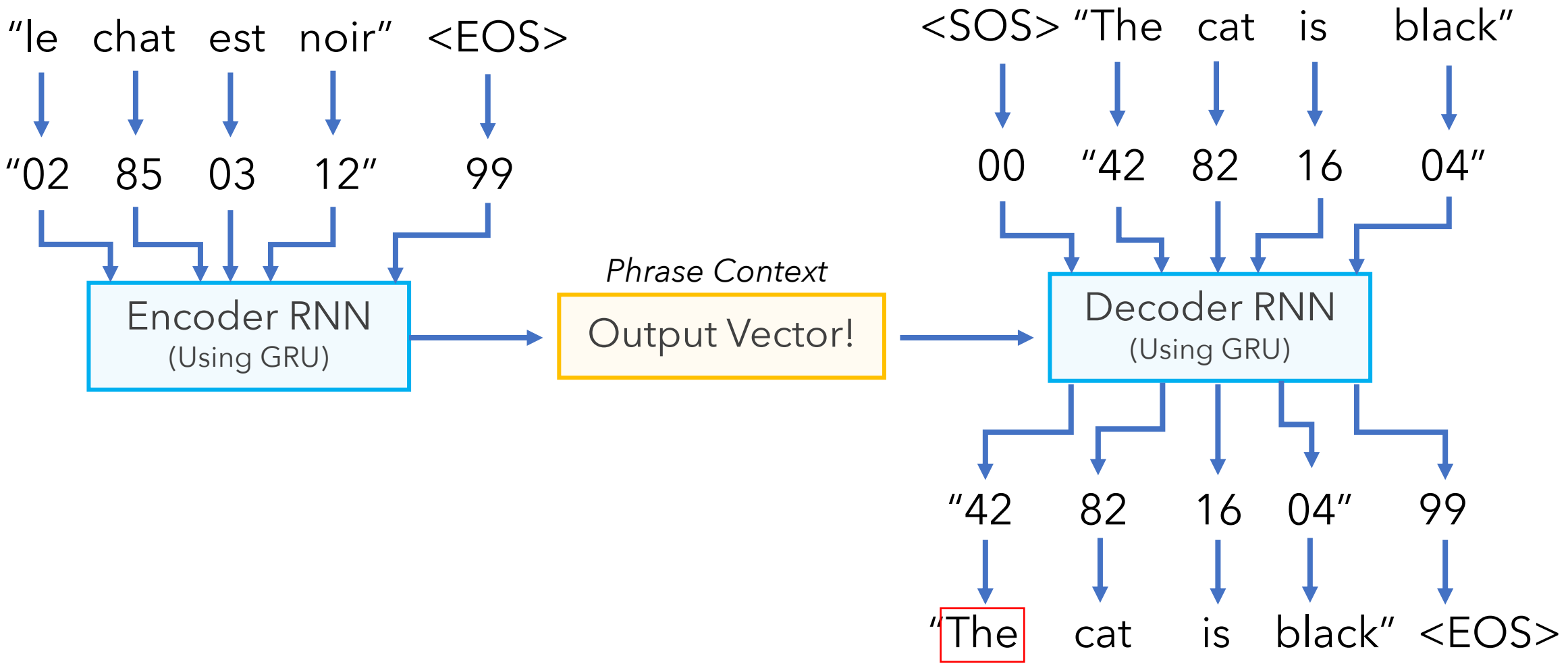
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



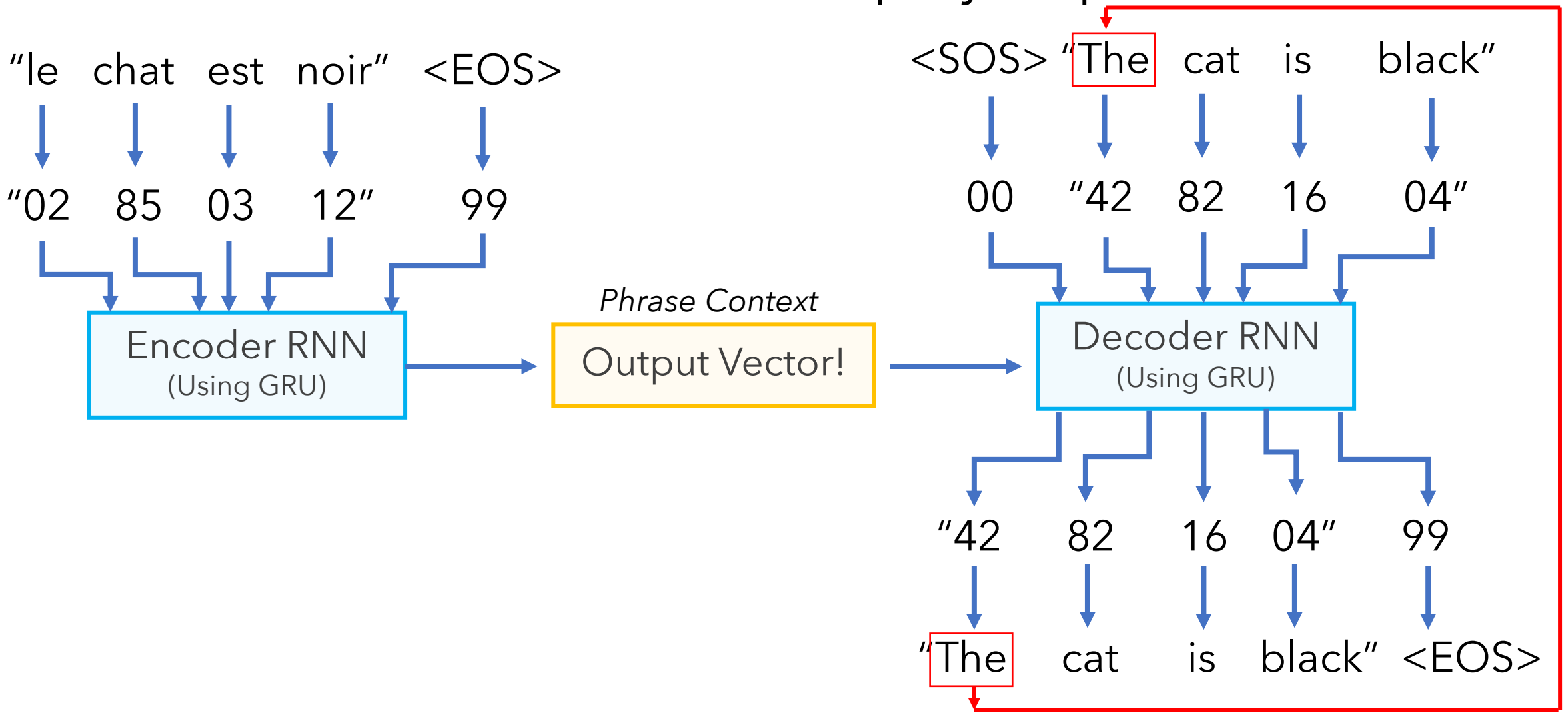
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



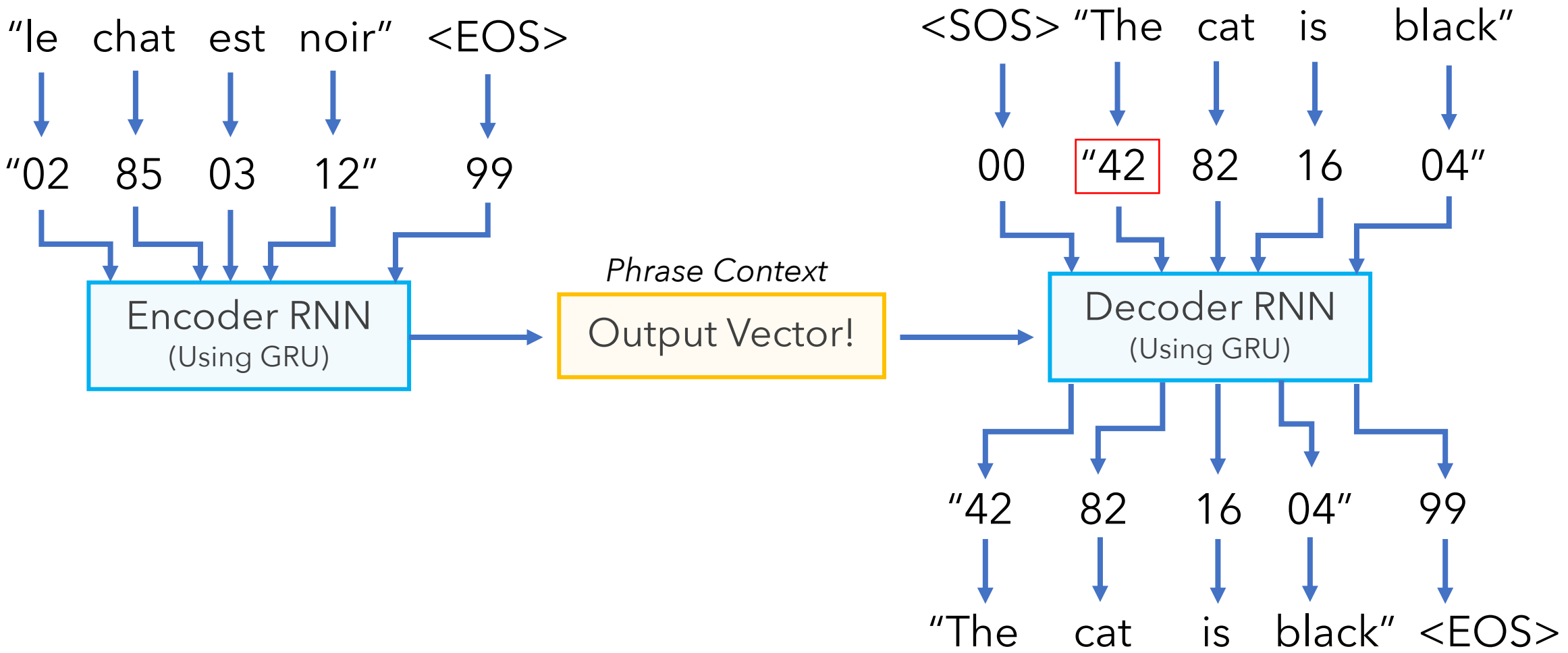
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



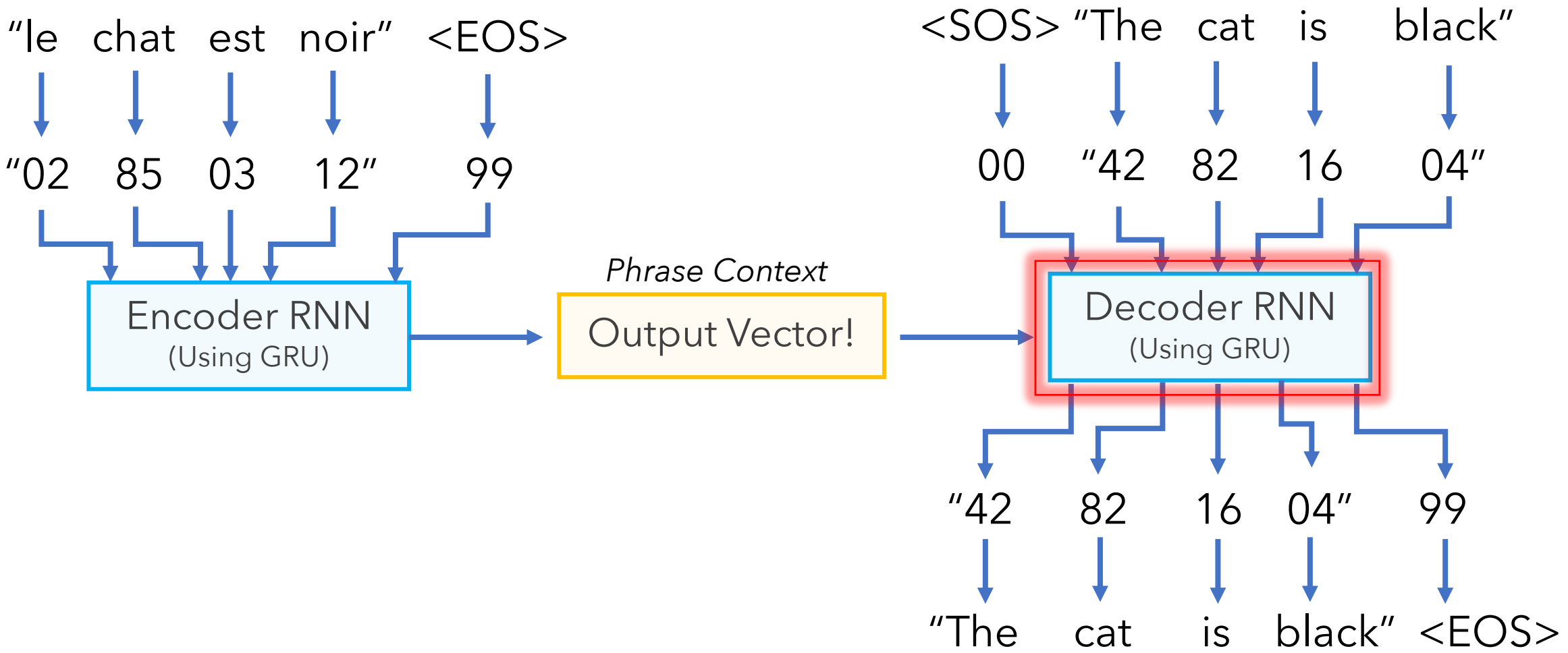
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



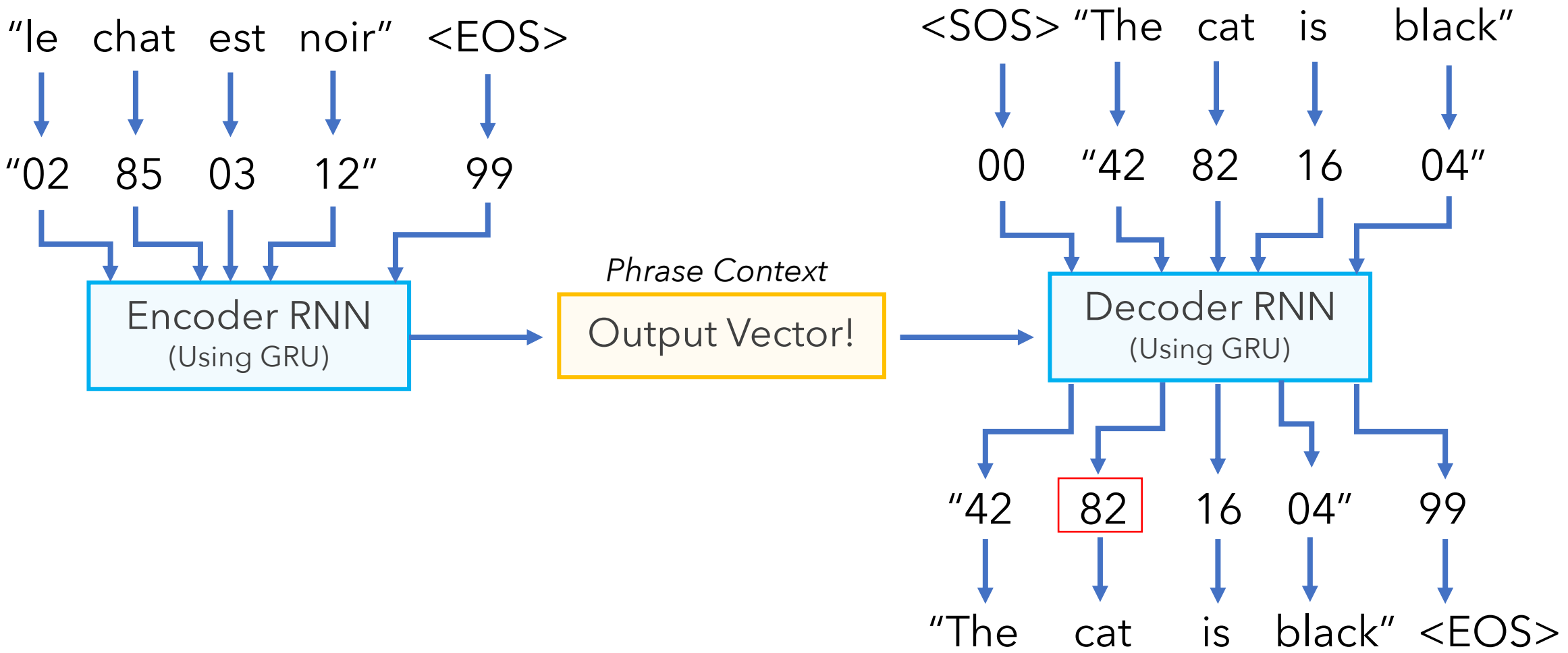
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



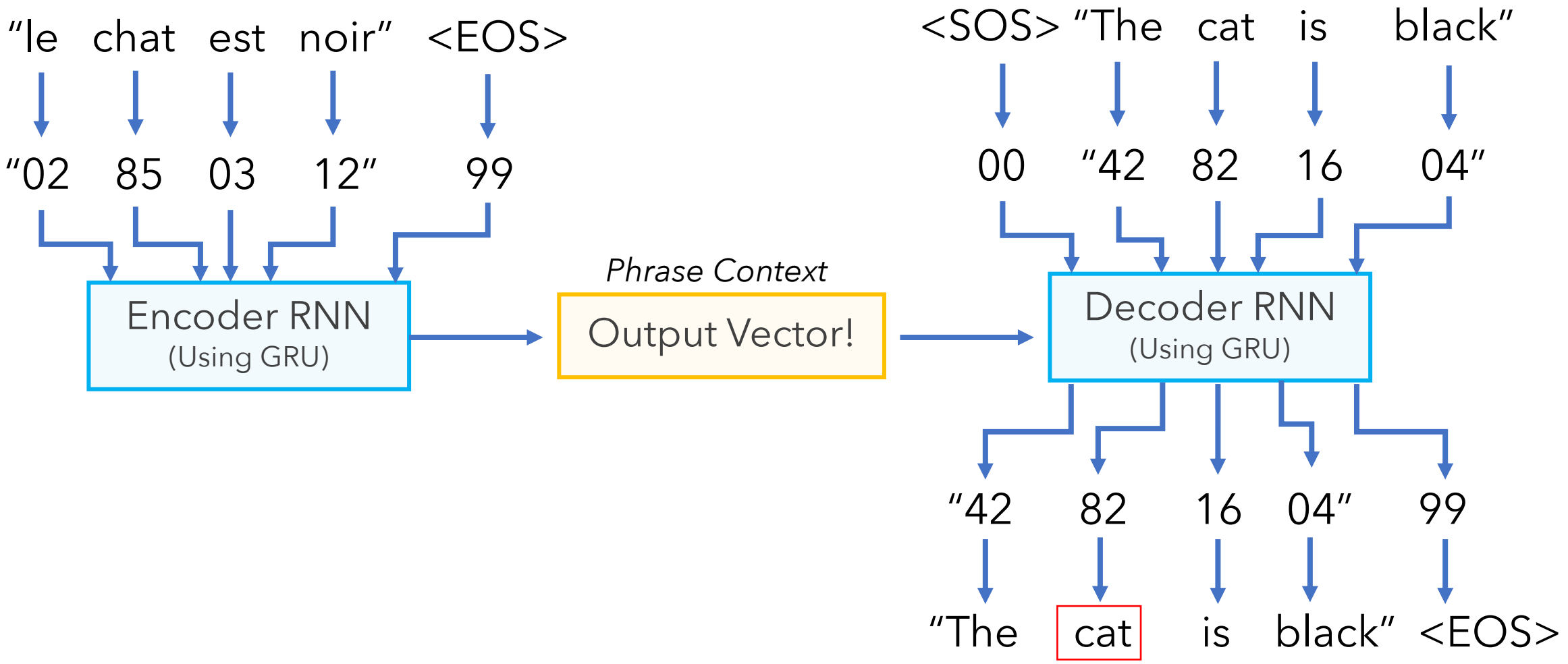
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



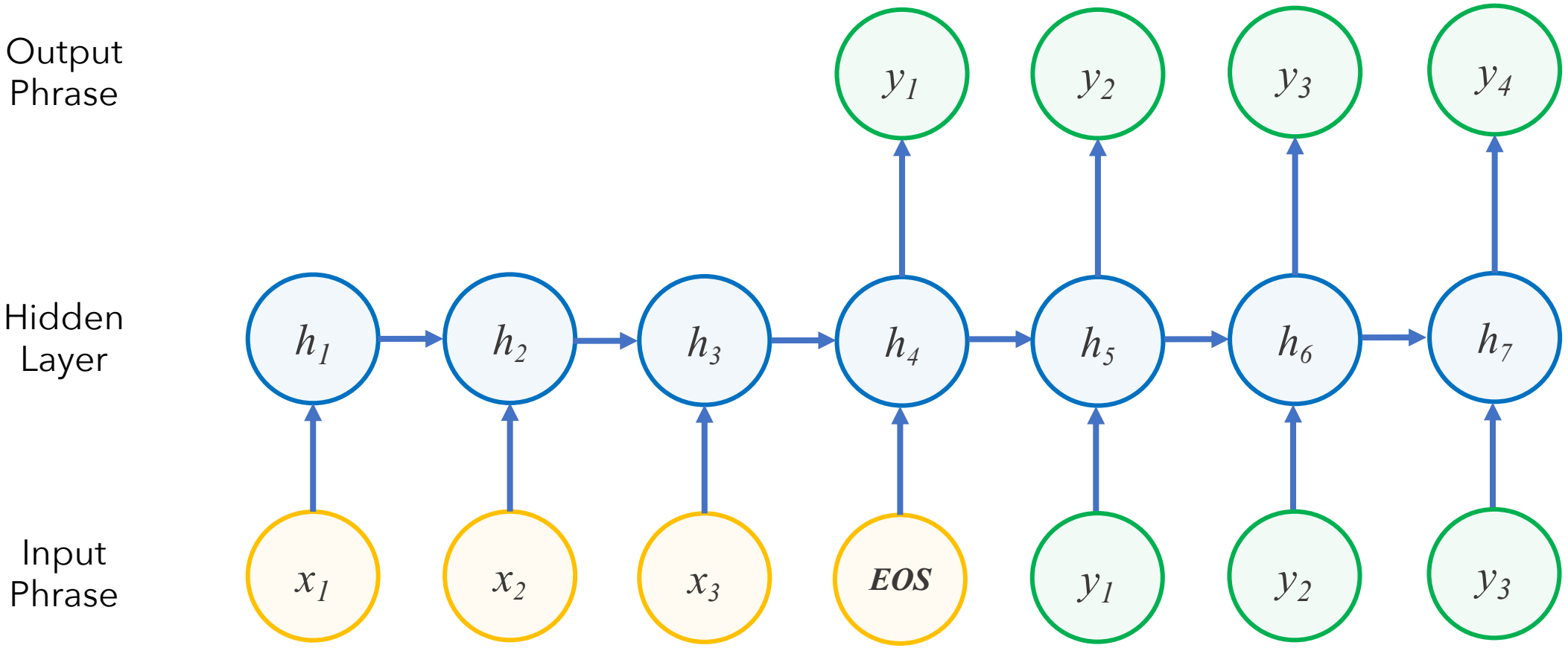
Sources: Sean Robertson

Encoder – Decoder Model: Step by step



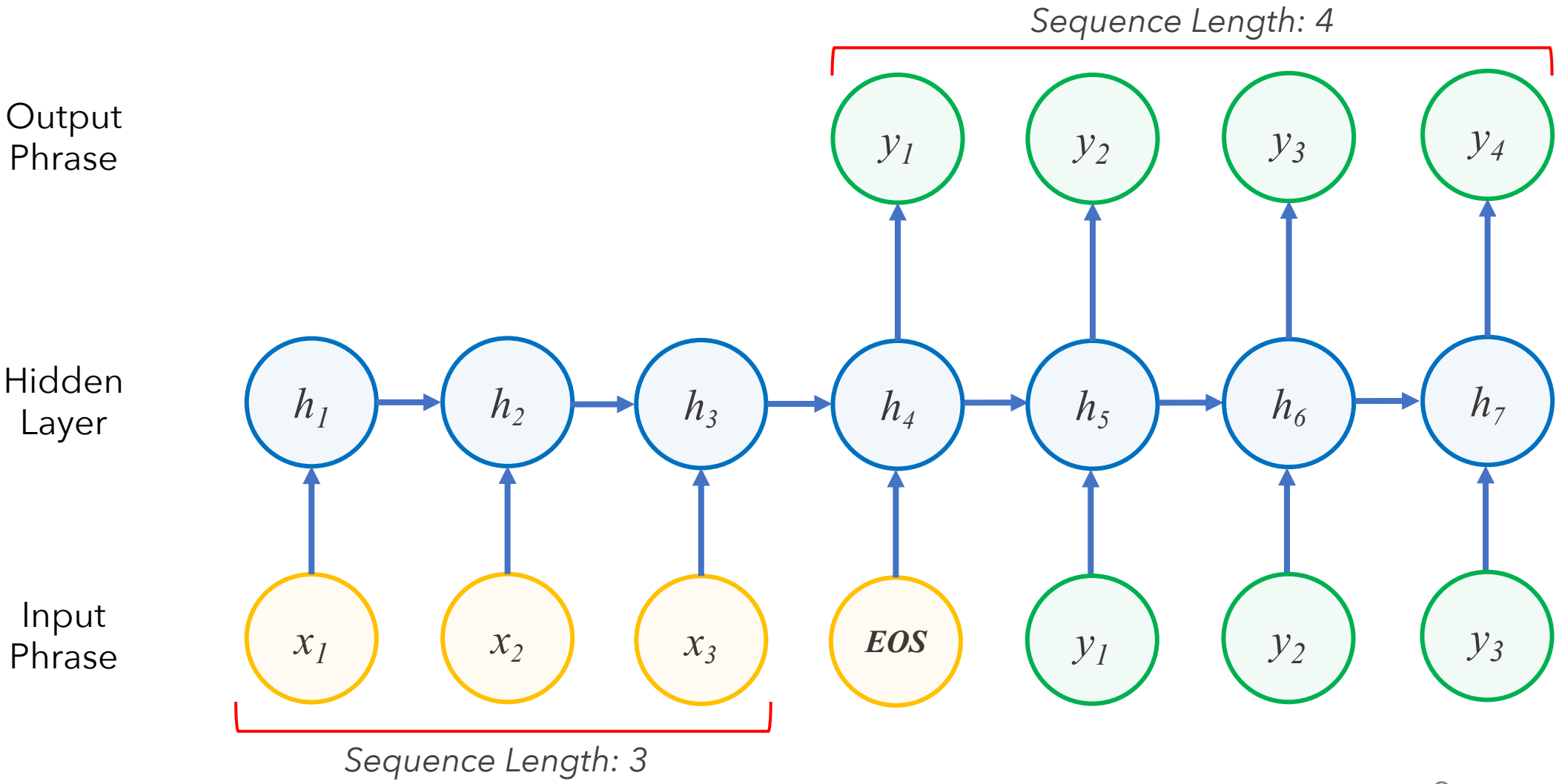
Sources: Sean Robertson

Machine Translation – seq2seq, encoder-decoder



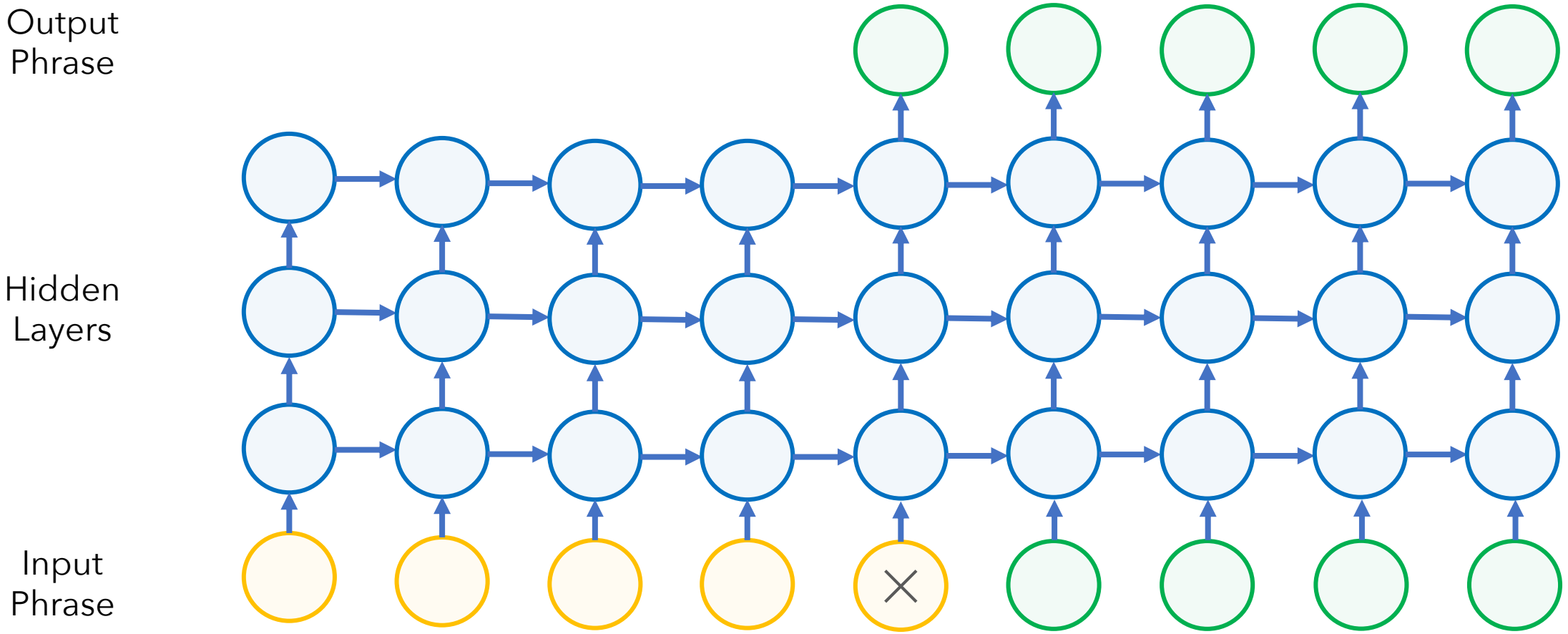
Sources: Ilya Sutskever

Machine Translation – seq2seq, encoder-decoder



Sources: Ilya Sutskever

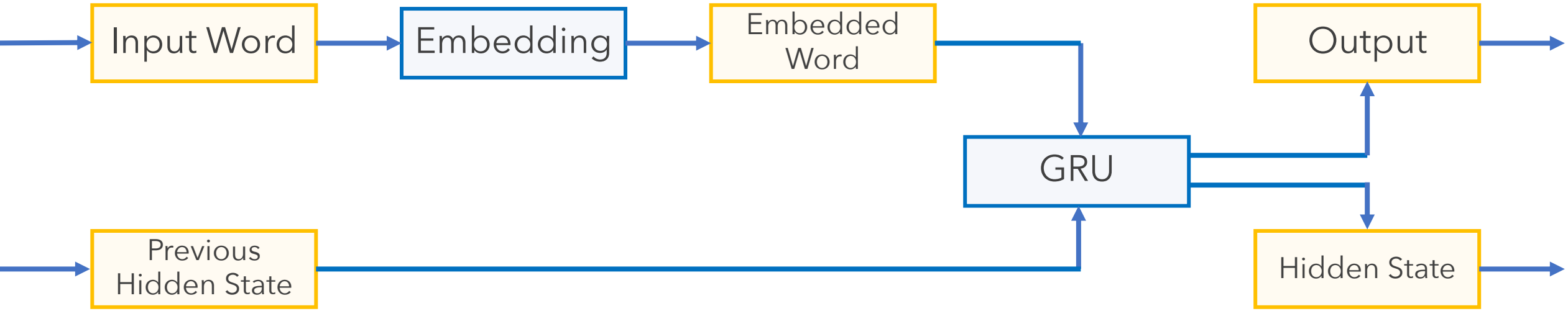
Machine Translation in reality... several hidden layers allows the network more space to store the context of the learned sequence and a better ability to reproduce the translated phrase.



Sources: Ilya Sutskever

Encoder: Produce encoded vector from sequence input

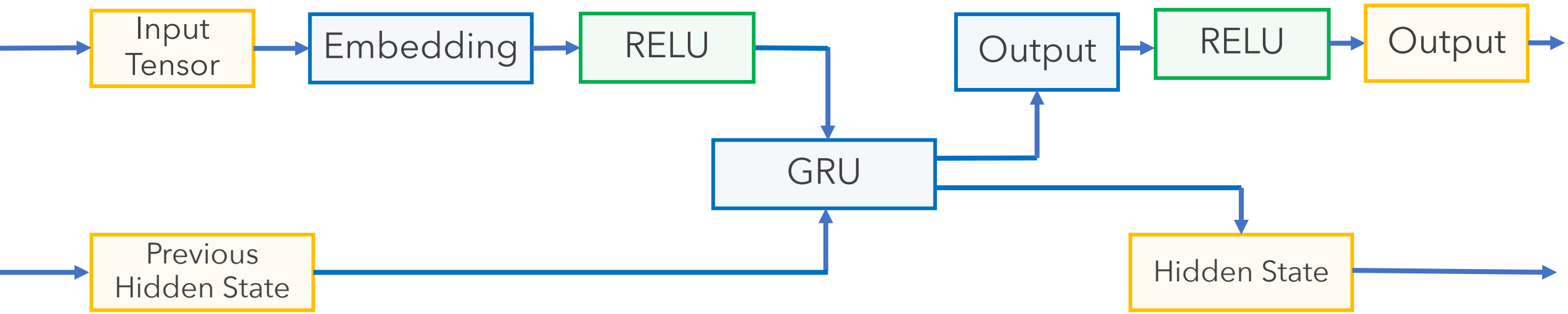
The job of the encoder is simply to take the phrase and convert it into a vector, storing not only an embedded representation of each word, but the **context of the phrase as a whole**.



Sources: Sean Robertson

Decoder: Produce sequence from vector input

The job of the decoder is to take the vector output of the encoder and convert it back into a series of embeddings, which then get converted back to text. Each subsequent word is *not* generated from an input word, but rather from the **input sequence context and meaning**.



Sources: Sean Robertson

My contributions: 1. Network Depth Optimizer 'deepforward' using random search across parameters

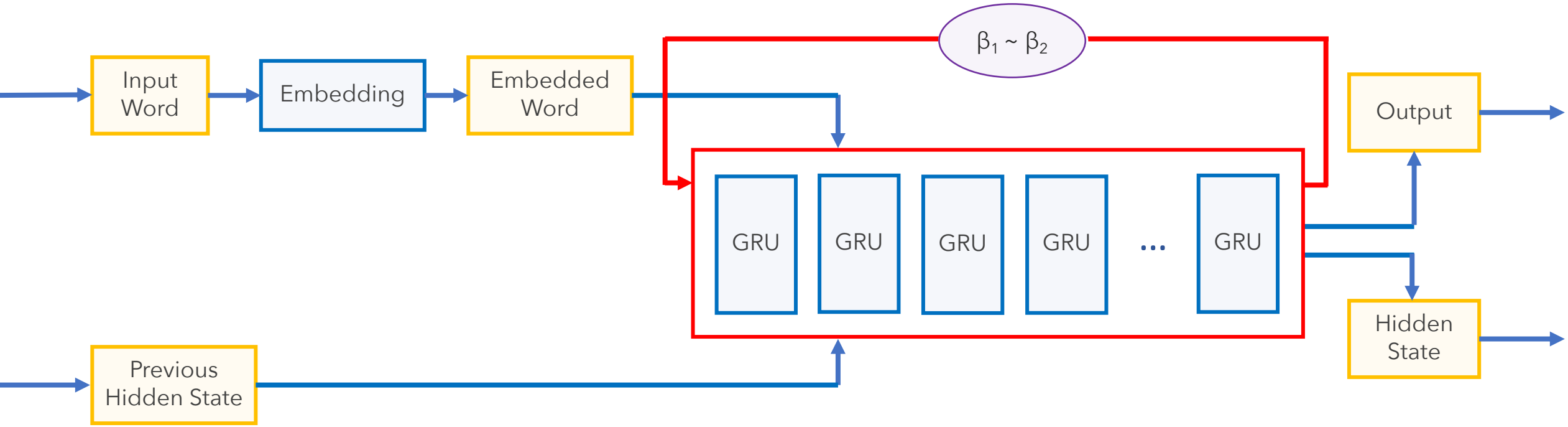
- We know that network topology is itself a hyperparameter, but deeper networks don't always mean better performance. [He et al., 2015]

- We also know that the best way to search for the optimal hyperparameters in order to reach maximum model performance is by using random search across n parameters between preset bounds β_1 and β_2. [Bergstra et al., 2012]

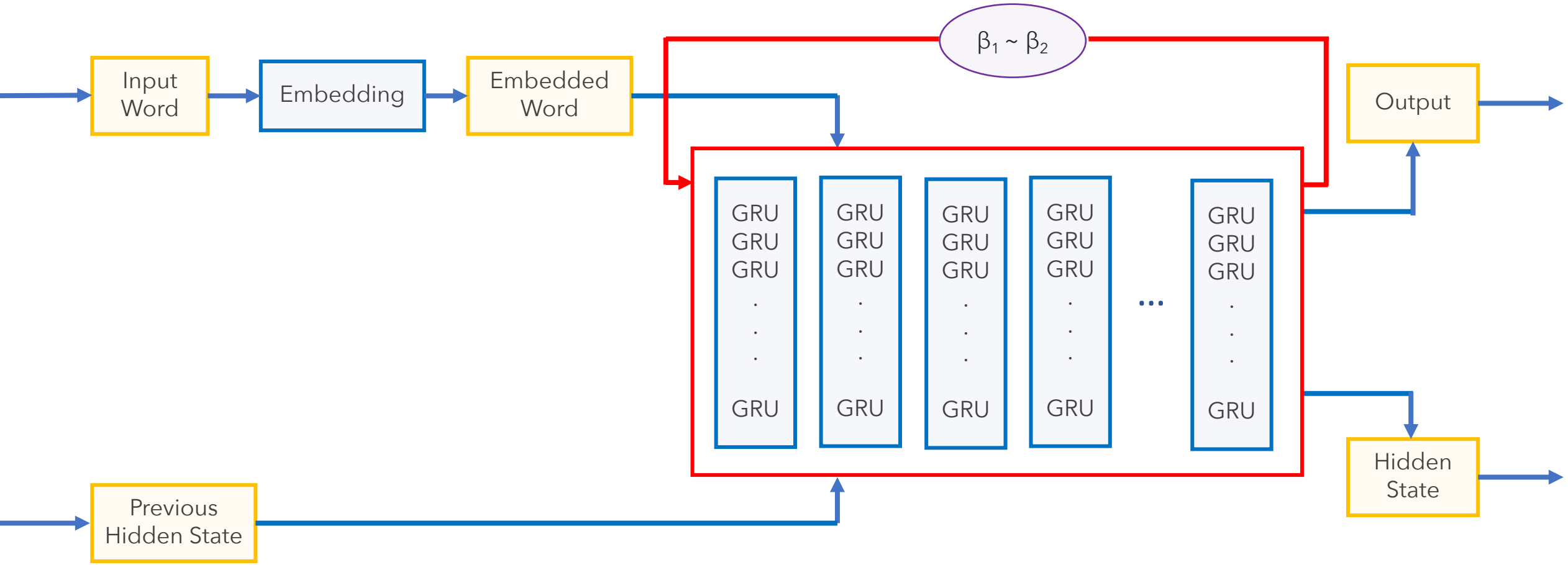
- This approach is greedy and comes at a steep cost of computation time, but it's a method which, especially combined with fine tuning, will yield impressive results

My contributions: 1. Network Depth Optimizer 'deepforward' using random search across parameters

- Sampling from a random uniform distribution between upper and lower bounds β_1 and β_2, we optimize number of GRUs in each layer and the number of layers in the encoder.

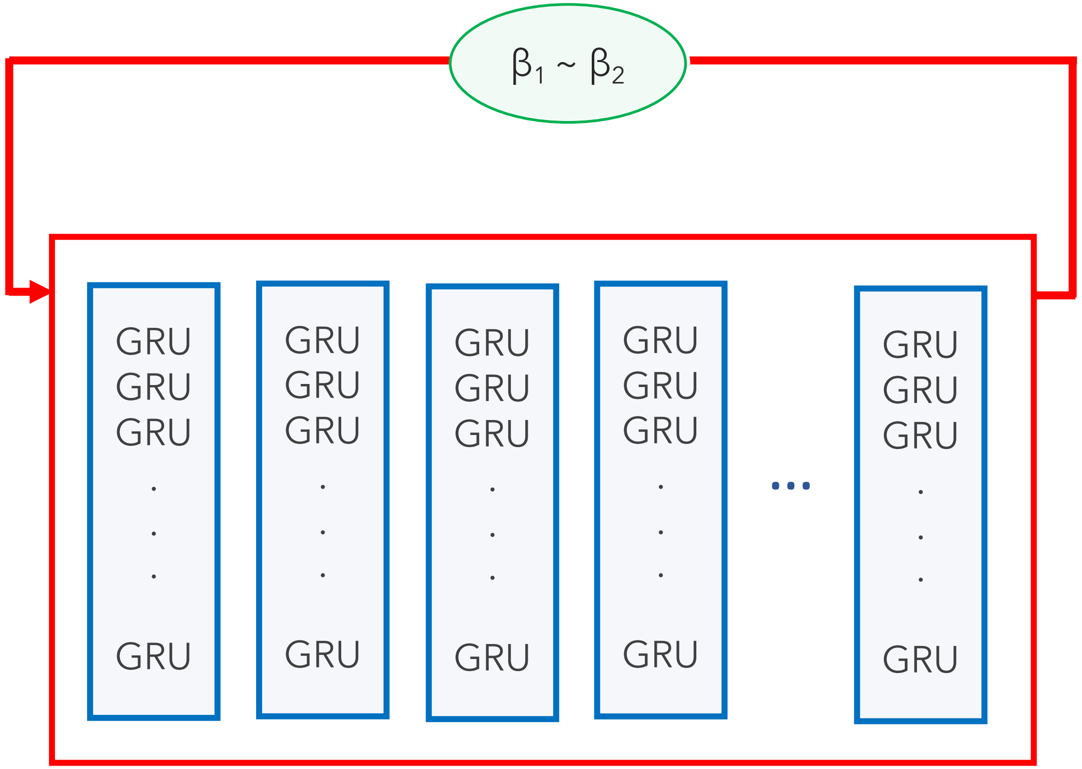


My contributions: 1. Network depth optimizer using random search across parameters



My contributions: 1. Network depth optimizer using random search across parameters

We can update β_1 and β_2 once we find an area which yields the best results to further improve the model results



My contributions: 1. Once we found the optimal parameters, we can shrink our distribution around that area and perform another search.

Initial Parameters:

$\beta_1 = 1, \beta_2 = 25$

Rand layers: [11, 4, 17, 22, 10]

$\beta_1 = 25, \beta_2 = 250$

Rand Nodes: [124, 238, 63, 136, 248]

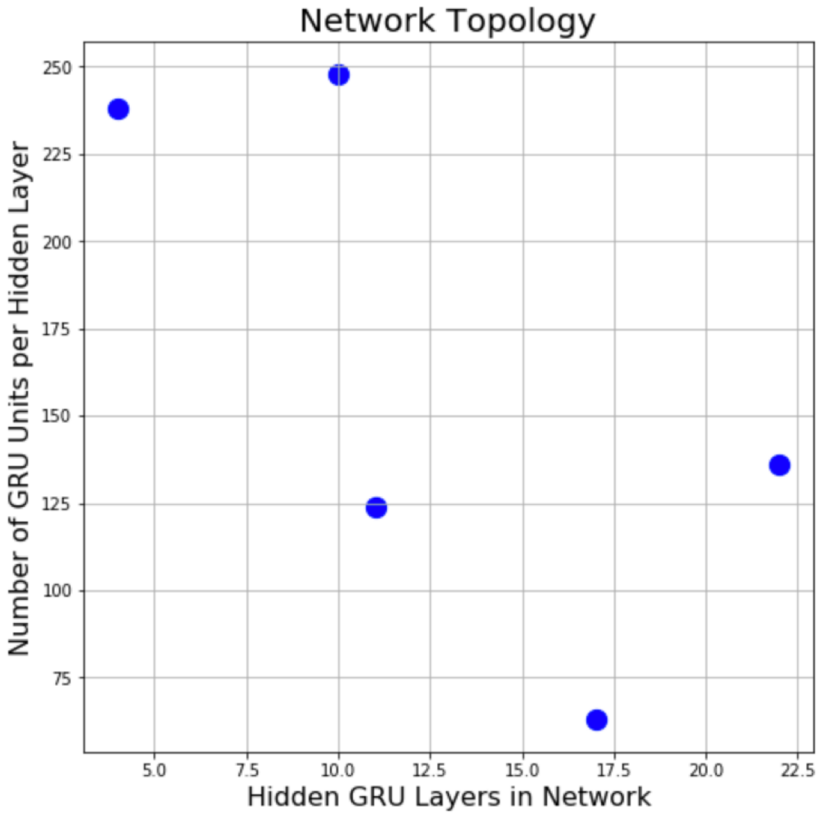
Fine Tuned Parameters:

$\beta_1 = 2, \beta_2 = 7$

Rand layers: [3, 3, 2, 6, 6]

$\beta_1 = 225, \beta_2 = 250$

Rand Nodes: [231, 248, 227, 249, 236]



My contributions: 1. Once we found the optimal parameters, we can shrink our distribution around that area and perform another search.

Initial Parameters:

$\beta_1 = 1, \beta_2 = 25$

Rand layers: [11, 4, 17, 22, 10]

$\beta_1 = 25, \beta_2 = 250$

Rand Nodes: [124, 238, 63, 136, 248]

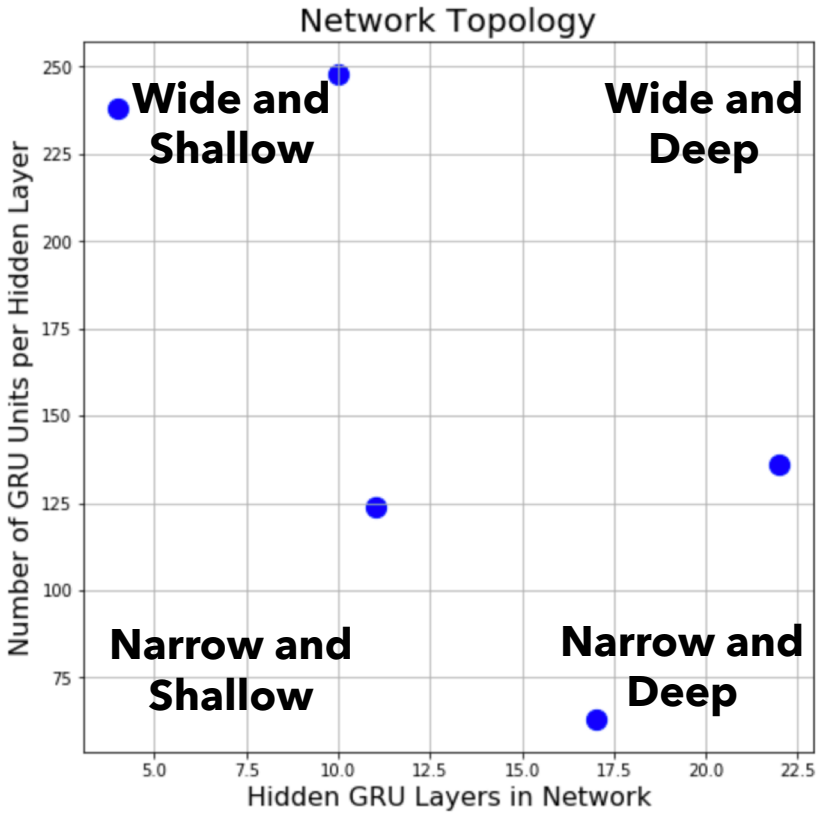
Fine Tuned Parameters:

$\beta_1 = 2, \beta_2 = 7$

Rand layers: [3, 3, 2, 6, 6]

$\beta_1 = 225, \beta_2 = 250$

Rand Nodes: [231, 248, 227, 249, 236]



My contributions: 1. Once we found the optimal parameters, we can shrink our distribution around that area and perform another search.

Initial Parameters:

$\beta_1 = 1, \beta_2 = 25$

Rand layers: [11, 4, 17, 22, 10]

$\beta_1 = 25, \beta_2 = 250$

Rand Nodes: [124, 238, 63, 136, 248]

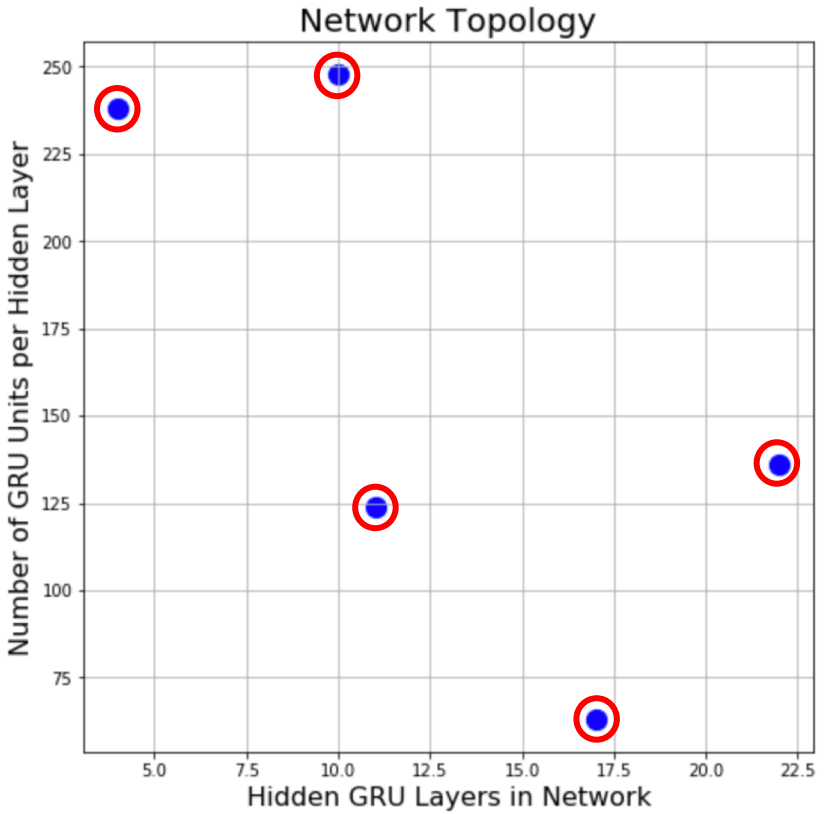
Fine Tuned Parameters:

$\beta_1 = 2, \beta_2 = 7$

Rand layers: [3, 3, 2, 6, 6]

$\beta_1 = 225, \beta_2 = 250$

Rand Nodes: [231, 248, 227, 249, 236]



My contributions: 1. Once we found the optimal parameters, we can shrink our distribution around that area and perform another search.

Initial Parameters:

$\beta_1 = 1, \beta_2 = 25$

Rand layers: [11, 4, 17, 22, 10]

$\beta_1 = 25, \beta_2 = 250$

Rand Nodes: [124, 238, 63, 136, 248]

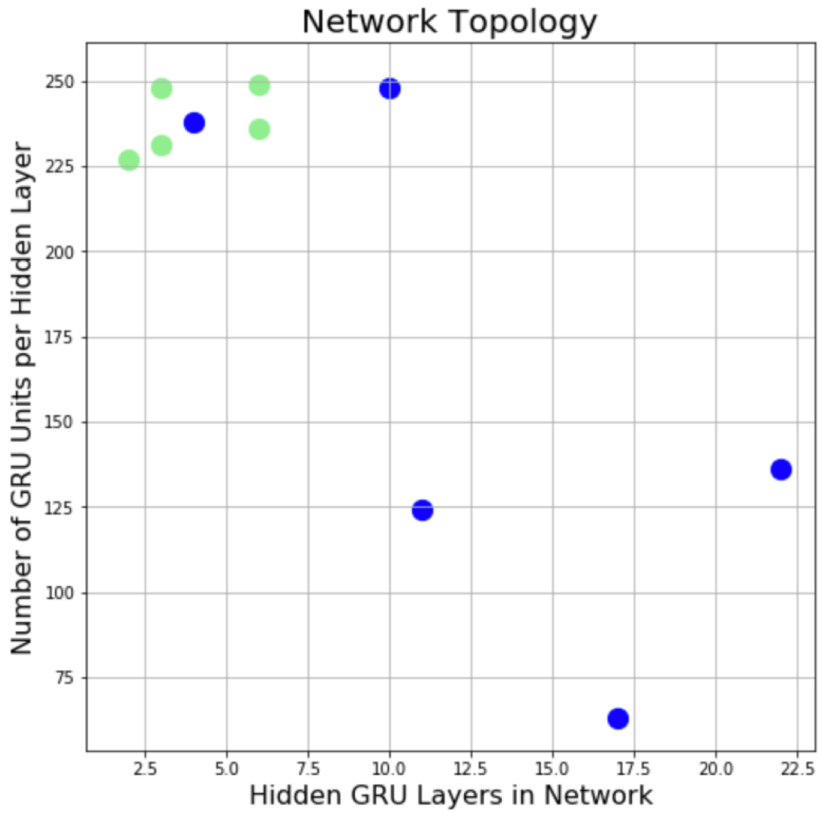
Fine Tuned Parameters:

$\beta_1 = 2, \beta_2 = 7$

Rand layers: [3, 3, 2, 6, 6]

$\beta_1 = 225, \beta_2 = 250$

Rand Nodes: [231, 248, 227, 249, 236]



My contributions: 1. Once we found the optimal parameters, we can shrink our distribution around that area and perform another search.

Initial Parameters:

$\beta_1 = 1, \beta_2 = 25$

Rand layers: [11, 4, 17, 22, 10]

$\beta_1 = 25, \beta_2 = 250$

Rand Nodes: [124, 238, 63, 136, 248]

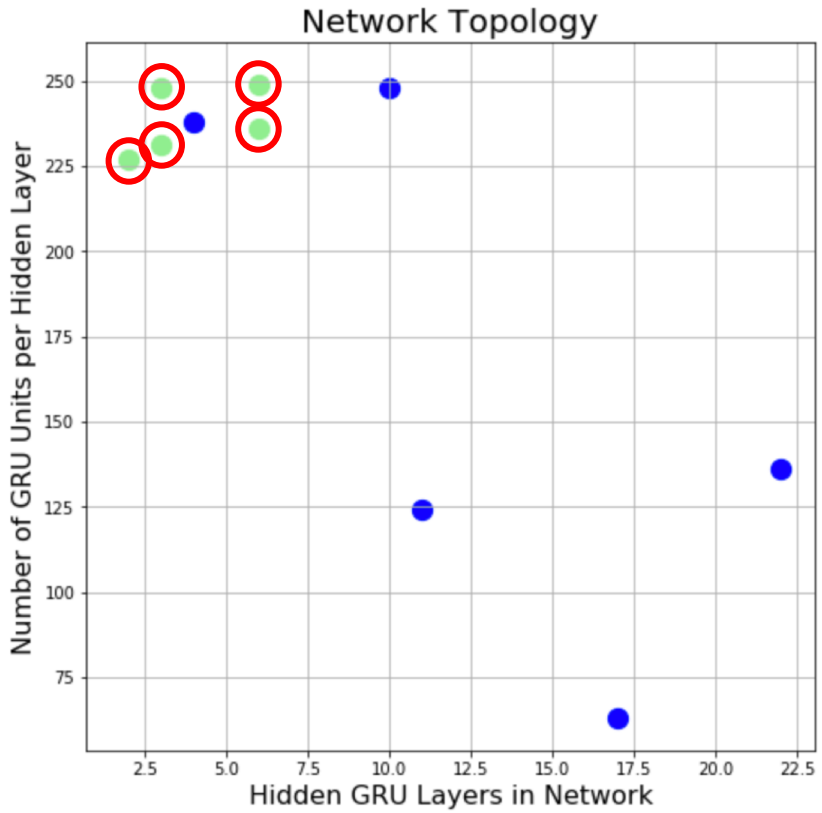
Fine Tuned Parameters:

$\beta_1 = 2, \beta_2 = 7$

Rand layers: [3, 3, 2, 6, 6]

$\beta_1 = 225, \beta_2 = 250$

Rand Nodes: [231, 248, 227, 249, 236]



Results

Results from network optimization

Optimization Results

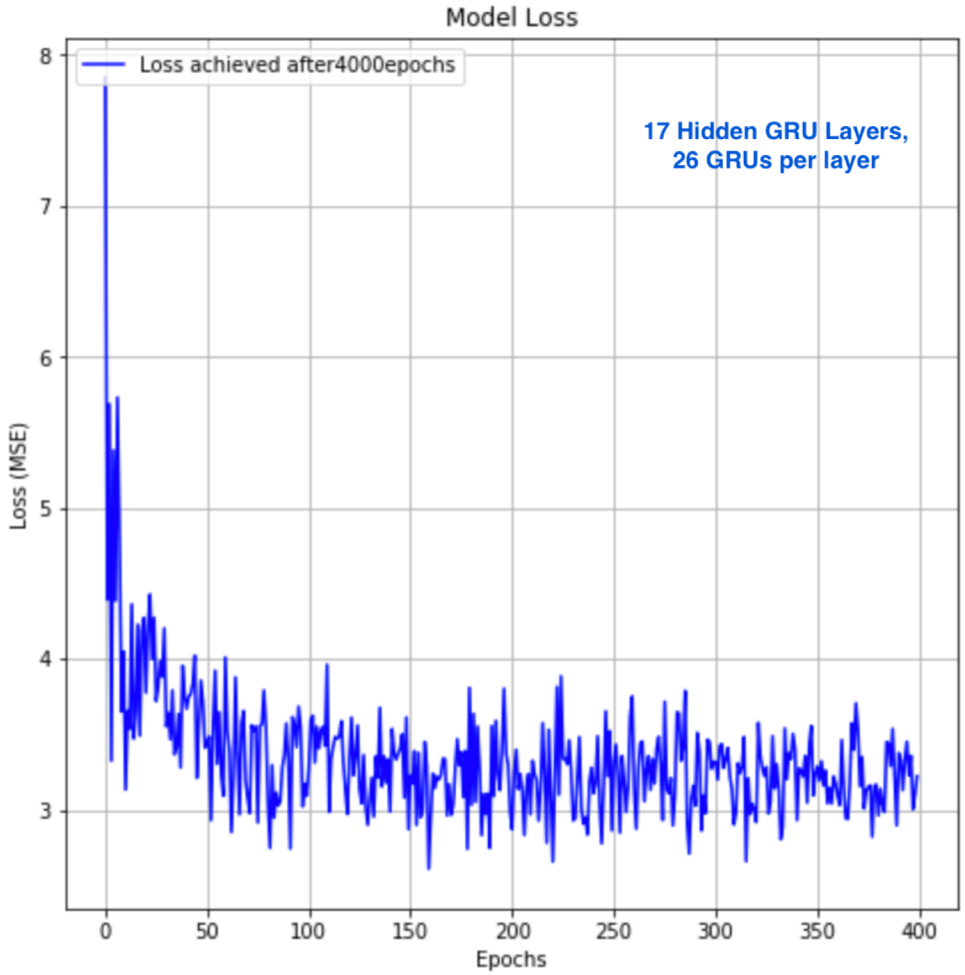
- Random Search optimal network topology:
  - 4 GRU Layers
  - 238 Nodes/Layer

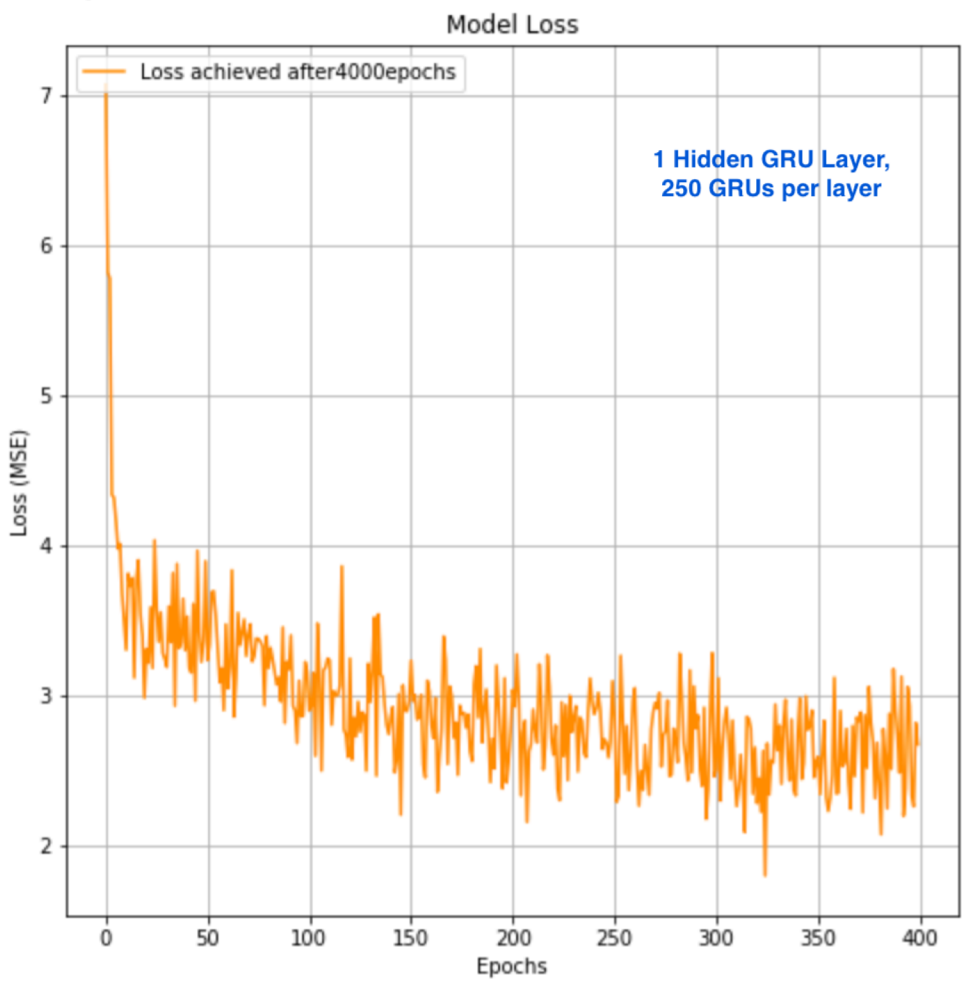
    Loss after 1000 epochs: *3.264*

- Fine Tuning optimal network topology:
  - 3 GRU Layers
  - 248 Nodes/Layer

    Loss after 1000 epochs: *3.211*

Optimization Results

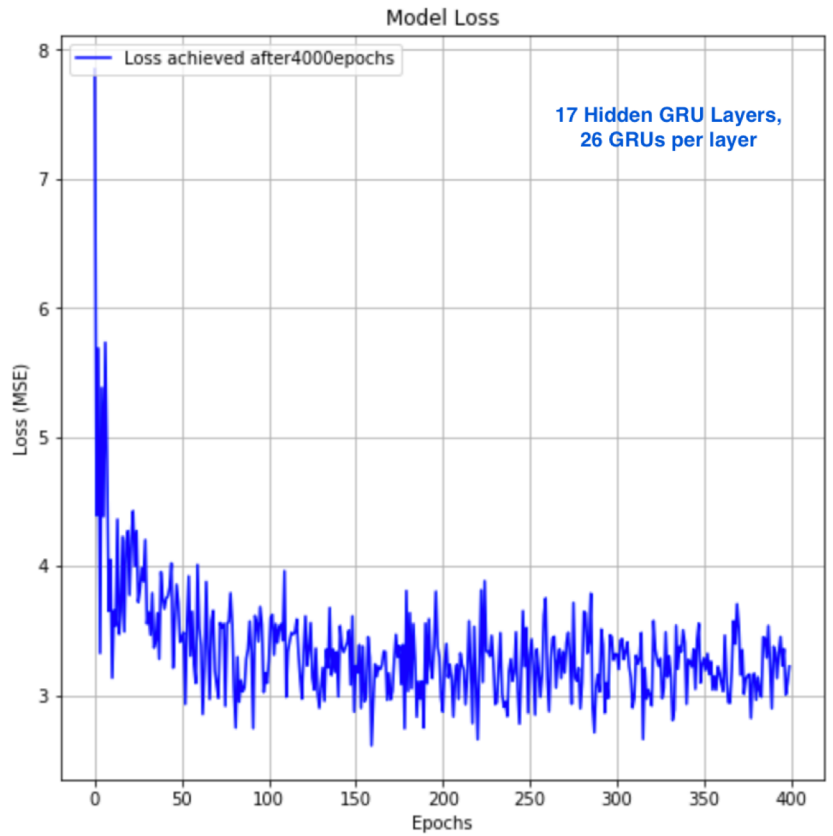




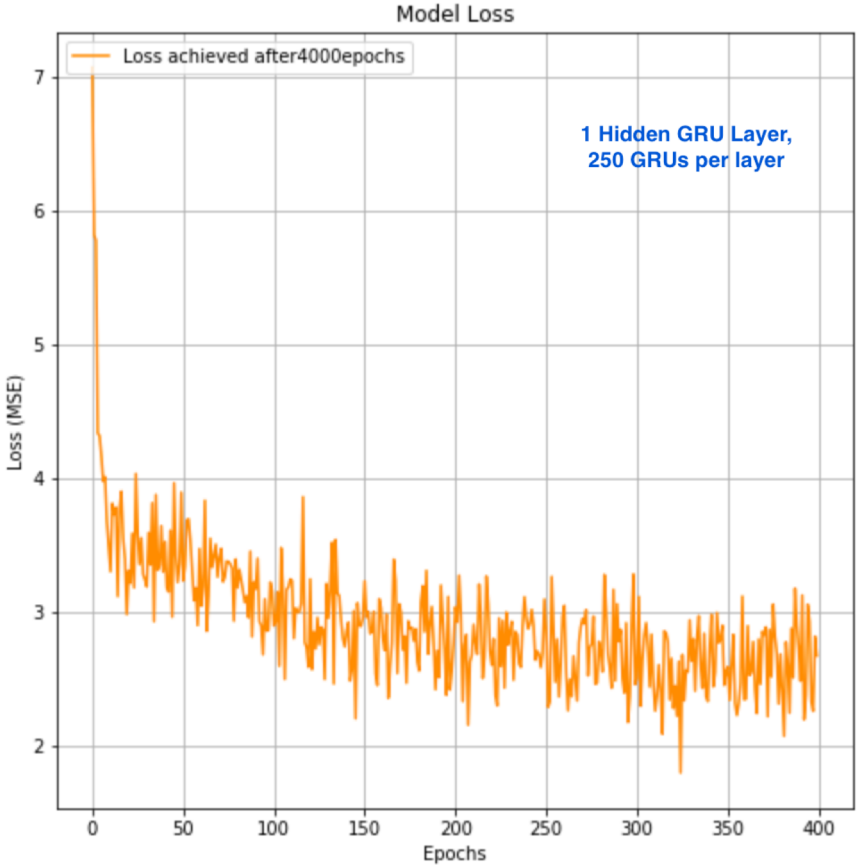
Optimization Results

We can see that the deeper, more narrow network converges more quickly, but never reaches the performance of the wide, shallow network

Deep and narrow



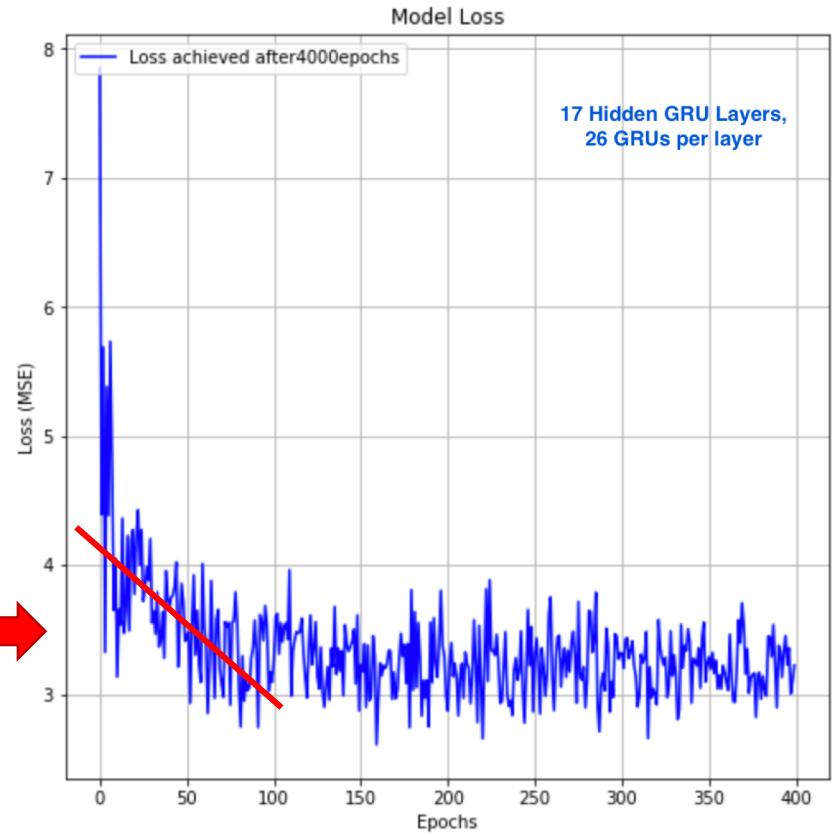
Wide and shallow



Optimization Results

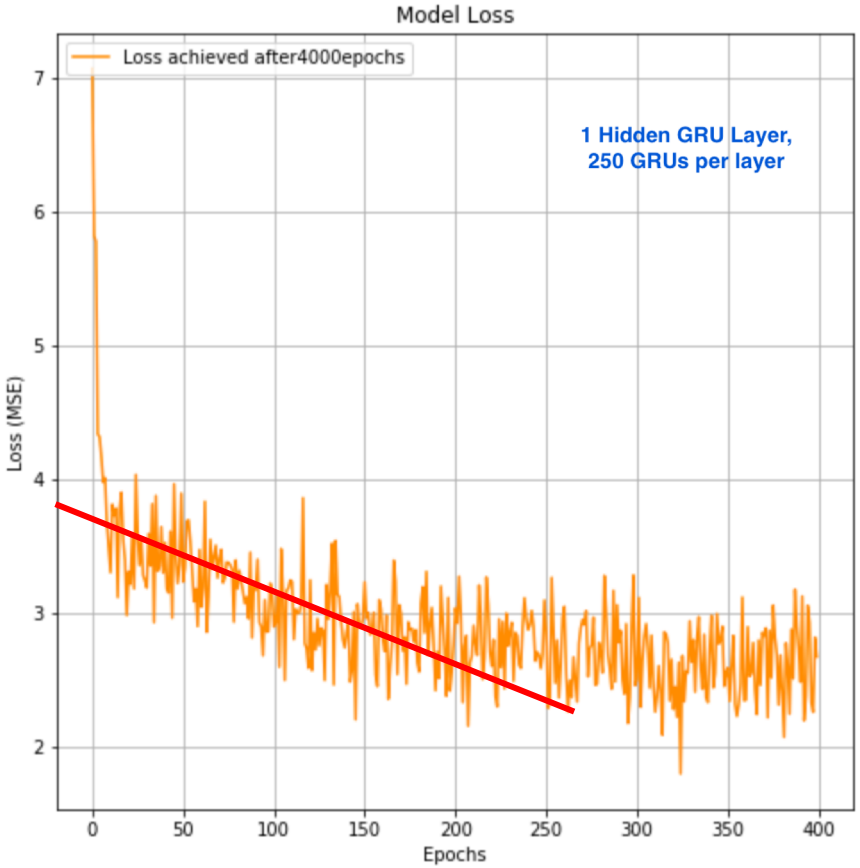
We can see that the deeper, more narrow network converges more quickly, but never reaches the performance of the wide, shallow network

Deep and narrow



Faster Convergence

Wide and shallow



Better Performance

Optimization Results

- We can see that the performance increases with the number of GRU nodes in the hidden state, and the ideal number of hidden GRU layers is three

Test Number	Number of Layers	Hidden State	Loss after 1000 epochs
1	11	124	3.267
2	4	238	3.264
3	17	63	3.321
4	22	136	3.301
5	10	248	3.386

Test Number	Number of Layers	Hidden State	Loss after 1000 epochs
1	1	200	3.219
2	3	231	3.352
3	3	248	3.211
4	2	227	3.375
5	6	249	3.259
6	6	236	3.284

Optimization Results

- We can see that the performance increases with the number of GRU nodes in the hidden state, and the ideal number of hidden GRU layers is three

Test Number	Number of Layers	Hidden State	Loss after 1000 epochs
1	11	124	3.267
2	4	238	**3.264**
3	17	63	3.321
4	22	136	3.301
5	10	248	3.386

Test Number	Number of Layers	Hidden State	Loss after 1000 epochs
1	1	200	3.219
2	3	231	3.352
3	3	248	**3.211**
4	2	227	3.375
5	6	249	3.259
6	6	236	3.284

Optimization Results

We can see that after training for only 1,000 epochs, even the optimal network struggles to translate full sentences

Using optimized network

1,000 epochs
0.5 teacher forcing
3 GRU Layers
248 Hidden nodes/layer

Translation 1

Target: “she s younger than him .”

Translation: “she is a .”

Translation 2

Target: “you re the teacher .”

Translation: “you re the .”

Optimization Results

After training for 15,000 epochs, the sentence structure, subject, and general meaning are captured by the encoder

Using optimized network

15,000 epochs
0.5 teacher forcing
3 GRU Layers
248 Hidden nodes/layer

Translation 1

Target: “you re the master .”

Translation: “you re the oldest.”

Translation 2

Target: “you re alone aren t you.”

Translation: “you re all aren t you .”

Future work – Network optimization methodology

- The methodology employed to optimize the encoder and decoder using other types of gated units such as LSTM units
- Other hyperparameters could be used in addition to number of hidden layers and nodes per layer to further improve model performance
- Epoch range could be increased during testing in order to eliminate the rick of an inherent nonlinearity in the learning curves which would skew results after only training for 1000 epochs

Questions?

Appendix

Word embeddings -

```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding = nn.Embedding(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
>>> embedding(input)
tensor([[[-0.0251, -1.6902,  0.7172],
         [-0.6431,  0.0748,  0.6969],
         [ 1.4970,  1.3448, -0.9685],
         [-0.3677, -2.7265, -0.1685]],

        [[ 1.4970,  1.3448, -0.9685],
         [ 0.4362, -0.4004,  0.9400],
         [-0.6431,  0.0748,  0.6969],
         [ 0.9124, -2.3616,  1.1151]]])
```

Slides prepared by Spencer R. Bertsch