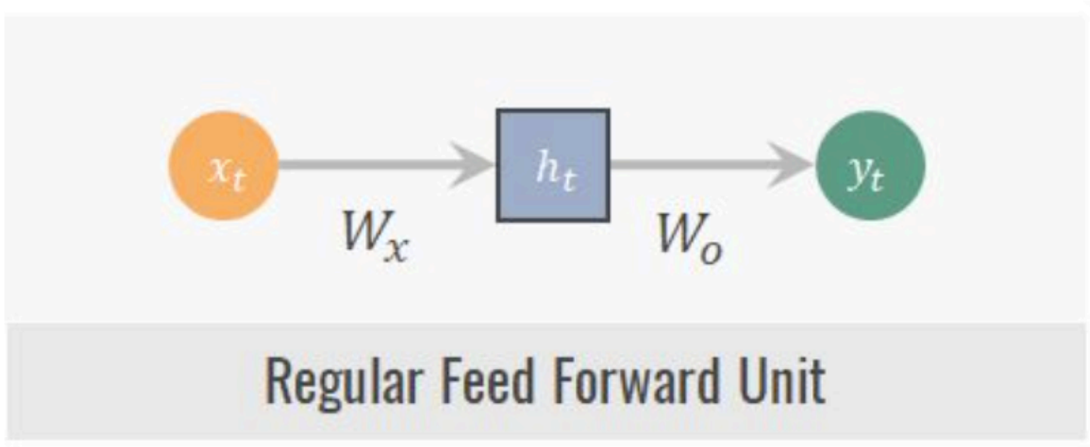
RNN

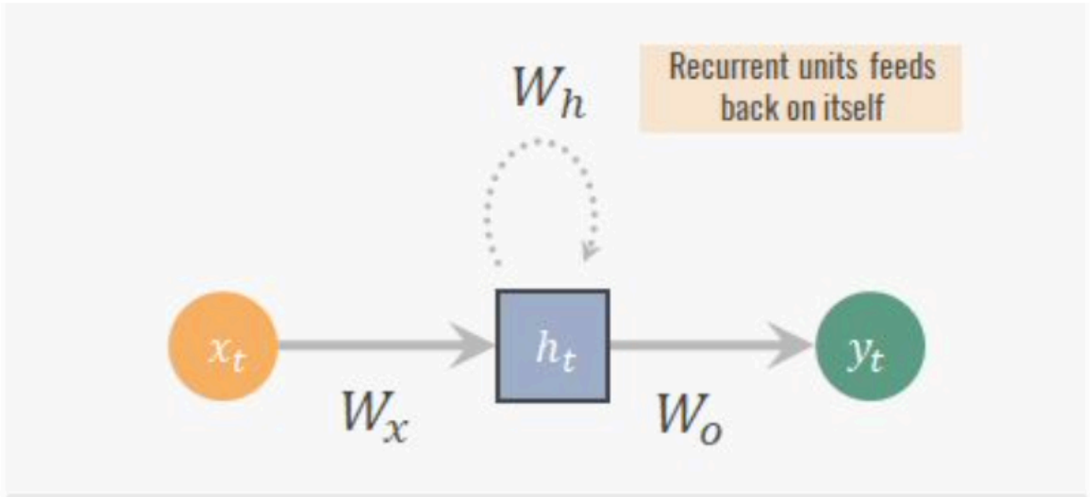
Introduction

- We're going to take a look at a new architectural structure
- The recurrent unit uses past information to predict sequence/ future outputs
- They are well suited to the task of dealing with grammar and syntax structures
- They also allow variable size inputs/outputs and alignment

recurrent net an additional weight matrix W_h



Regular Feed Forward Unit



Simple Recurrent Unit

Standard unit

Elman unit

two weight matrices W_x and W_o

"memory" retain past information

Appeal of RNNs is the idea that they can connect previous information to the present tasks, examples include:

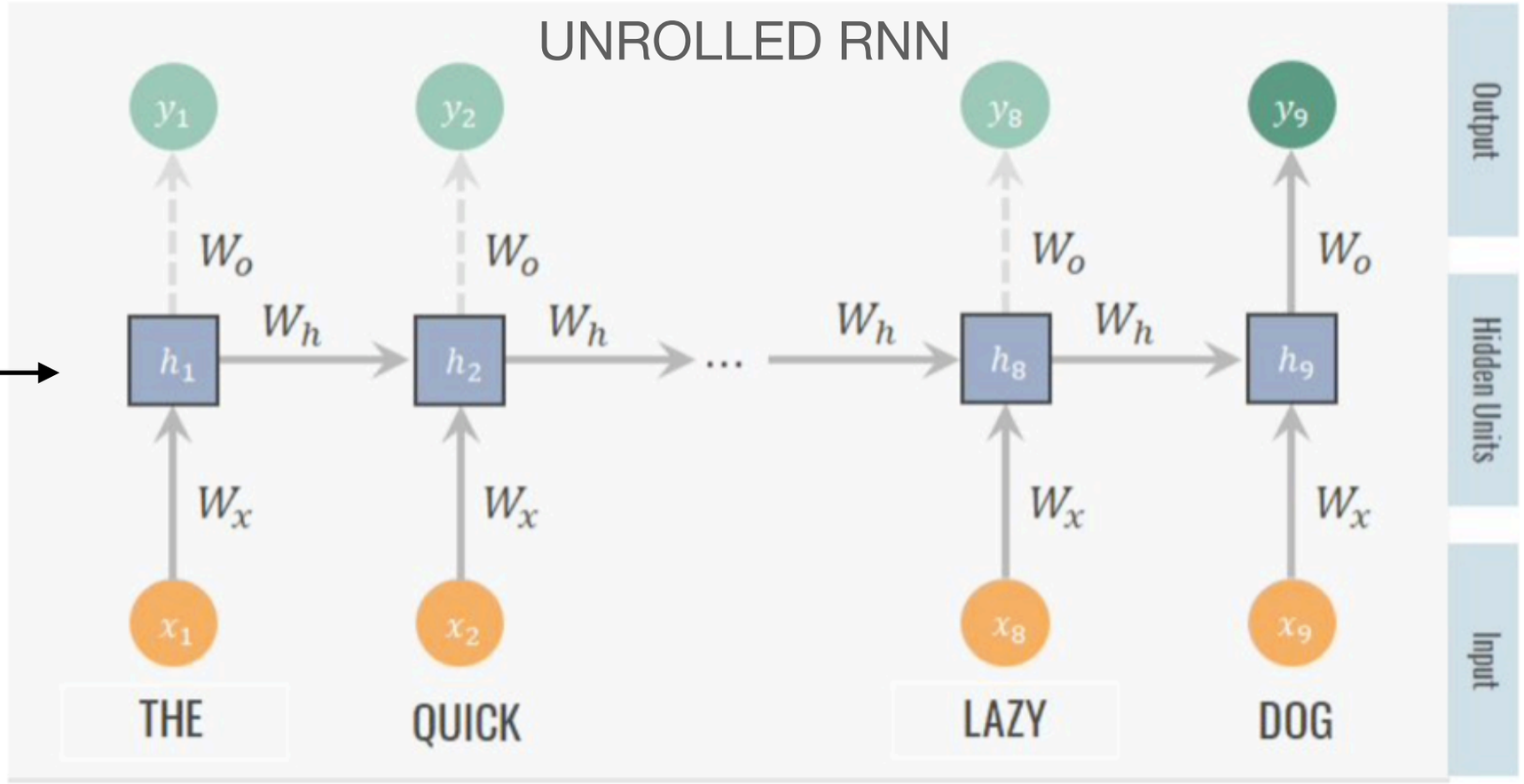
- Earlier video frames
- Word context
- Time series data

- RNNs are inspired by how we read

- h_o is usually initialized to 0

$$h_o = 0$$

$$h_t = f(a_t) \Leftrightarrow f(W_h h_{t-1} + W_x^T x_t + b_h)$$

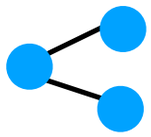


- Weights are shared in a recurrent net

- In previous situations (i.e. classification) our output was one class, RNNs allow us to output a value at every step

- RNNs are trained using back propagation through time (a type of gradient descent)

Example-1



```
model.add(SimpleRNN(2,
    input_shape=(3,1),
    activation='linear',
    bias_initializer="ones",
    kernel_initializer="ones",
    recurrent_initializer="ones",
    return_sequences=True),
    )
```

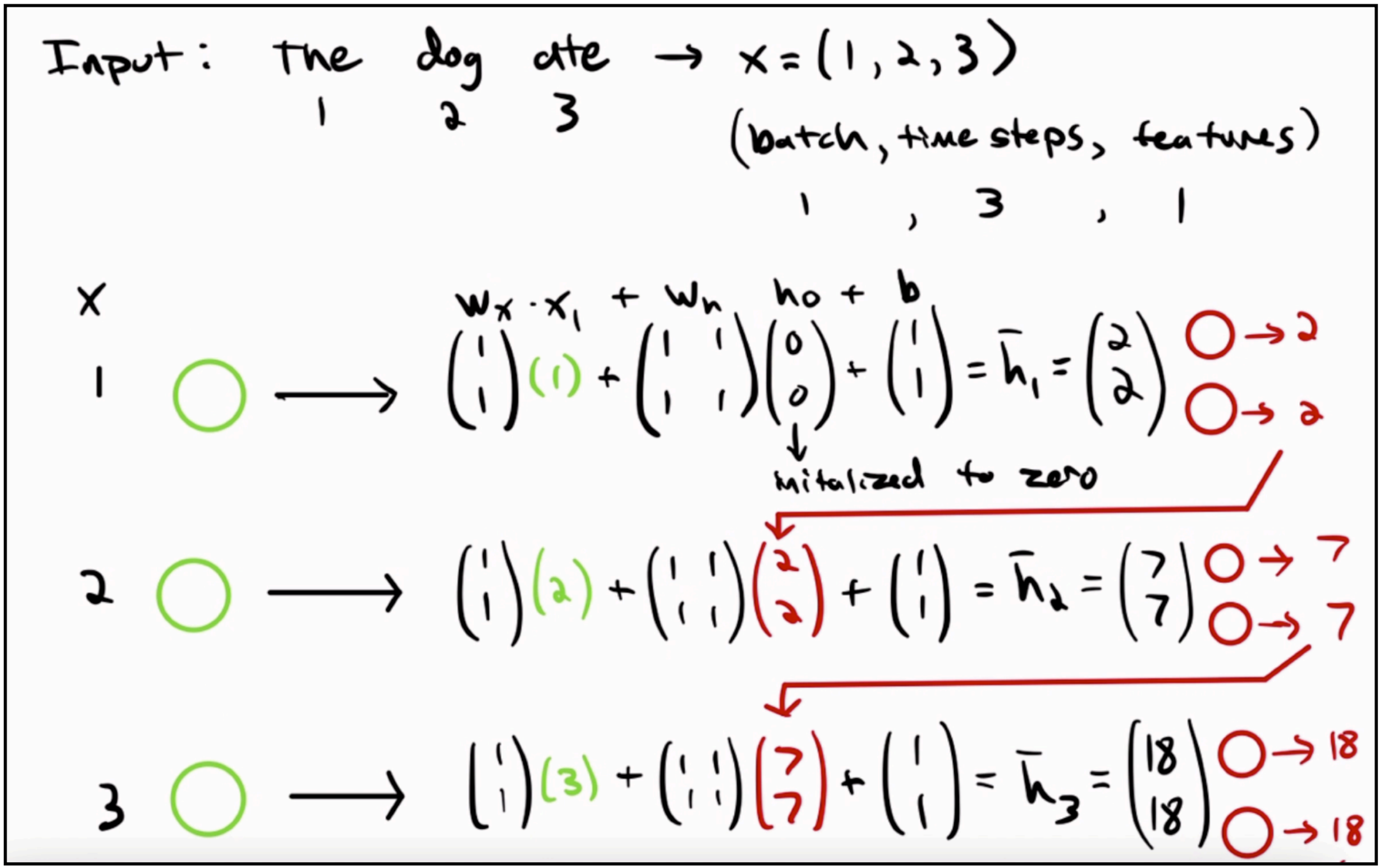
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 3, 2)              8
=================================================================
```

```
x_input (1, 3, 1)
[[[1]
  [2]
  [3]]]
```

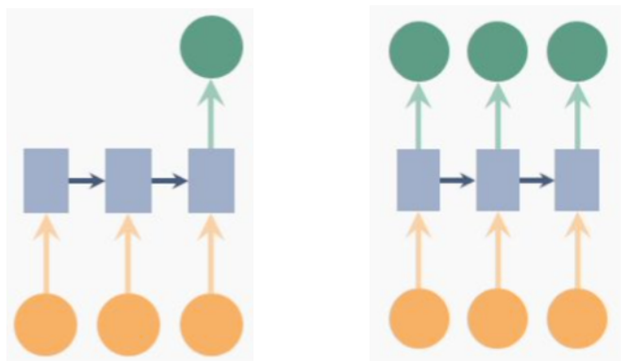
$$h_t = f(a_t) \Leftrightarrow f(W_h h_{t-1} + W_x^T x_t + b_h)$$

```
KERAS Predict
[[[ 2.  2.]
  [ 7.  7.]
  [18. 18.]]]
```

Code Demo: OPEN AND RUN CODE DEMO: KERAS-RNN-LAYER-EXAMPLE.py



- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `False`.



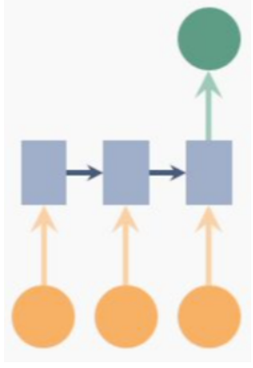
```
KERAS Predict
[[[ 2.  2.]
  [ 7.  7.]
  [18. 18.]]]
```

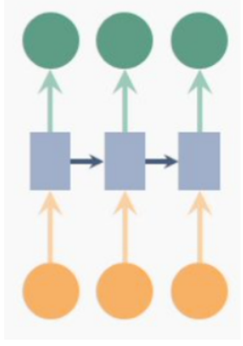
Keras Recurrent Layers

- ★ LSTM layer
- ★ GRU layer
- ★ SimpleRNN layer
- TimeDistributed layer
- ★ Bidirectional layer
- ConvLSTM2D layer
- Base RNN layer

https://keras.io/api/layers/recurrent_layers/

- **units**: Positive integer, dimensionality of the output space.
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `False`.





SimpleRNN class

```
tf.keras.layers.SimpleRNN(
    units,
    activation="tanh",
    use_bias=True,
    kernel_initializer="glorot_uniform",
    recurrent_initializer="orthogonal",
    bias_initializer="zeros",
    kernel_regularizer=None,
    recurrent_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    recurrent_constraint=None,
    bias_constraint=None,
    dropout=0.0,
    recurrent_dropout=0.0,
    return_sequences=False,
    return_state=False,
    go_backwards=False,
    stateful=False,
    unroll=False,
    **kwargs
)
```

Fully-connected RNN where the output is to be fed back to input.

Alternative Representation-1

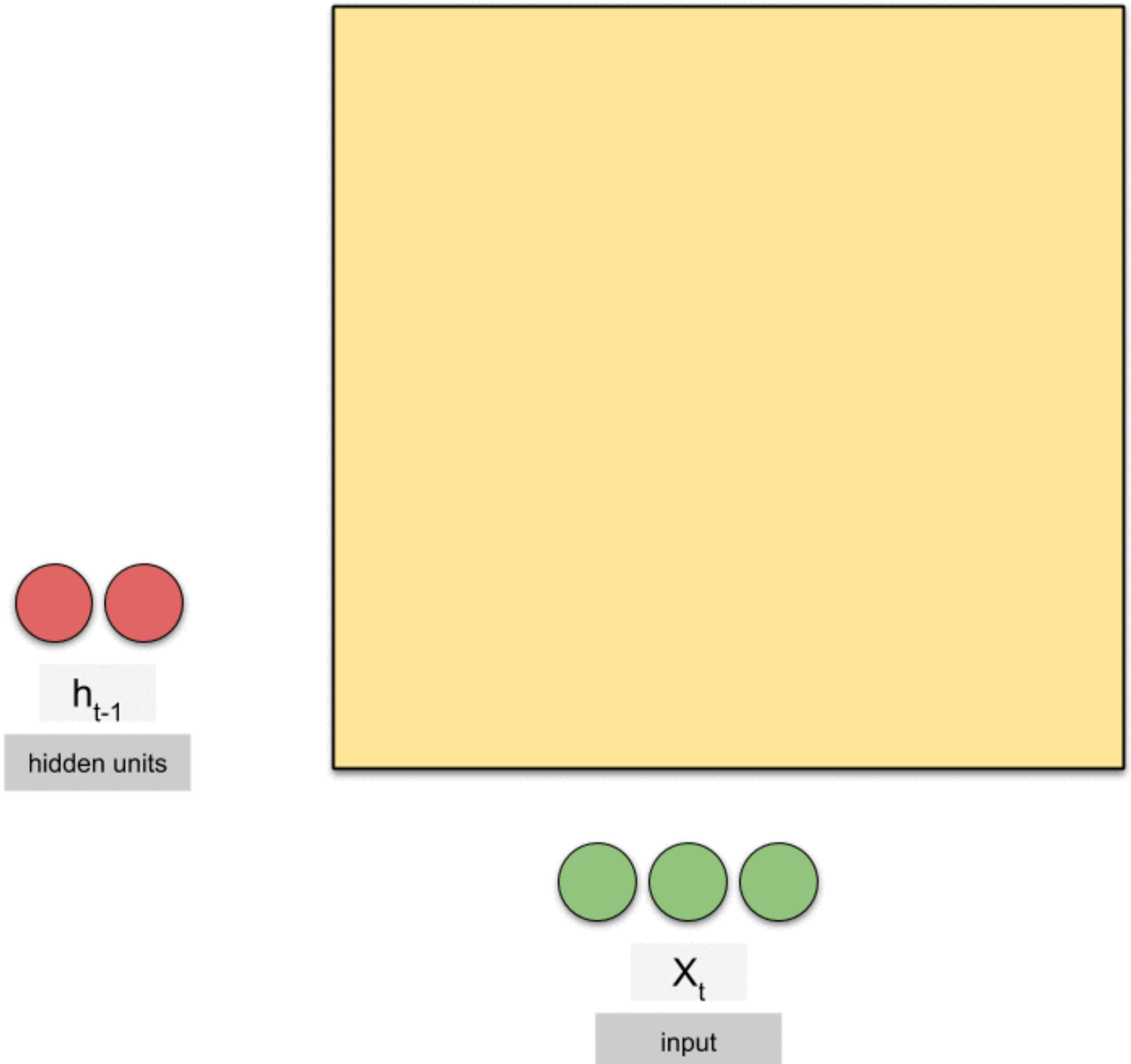
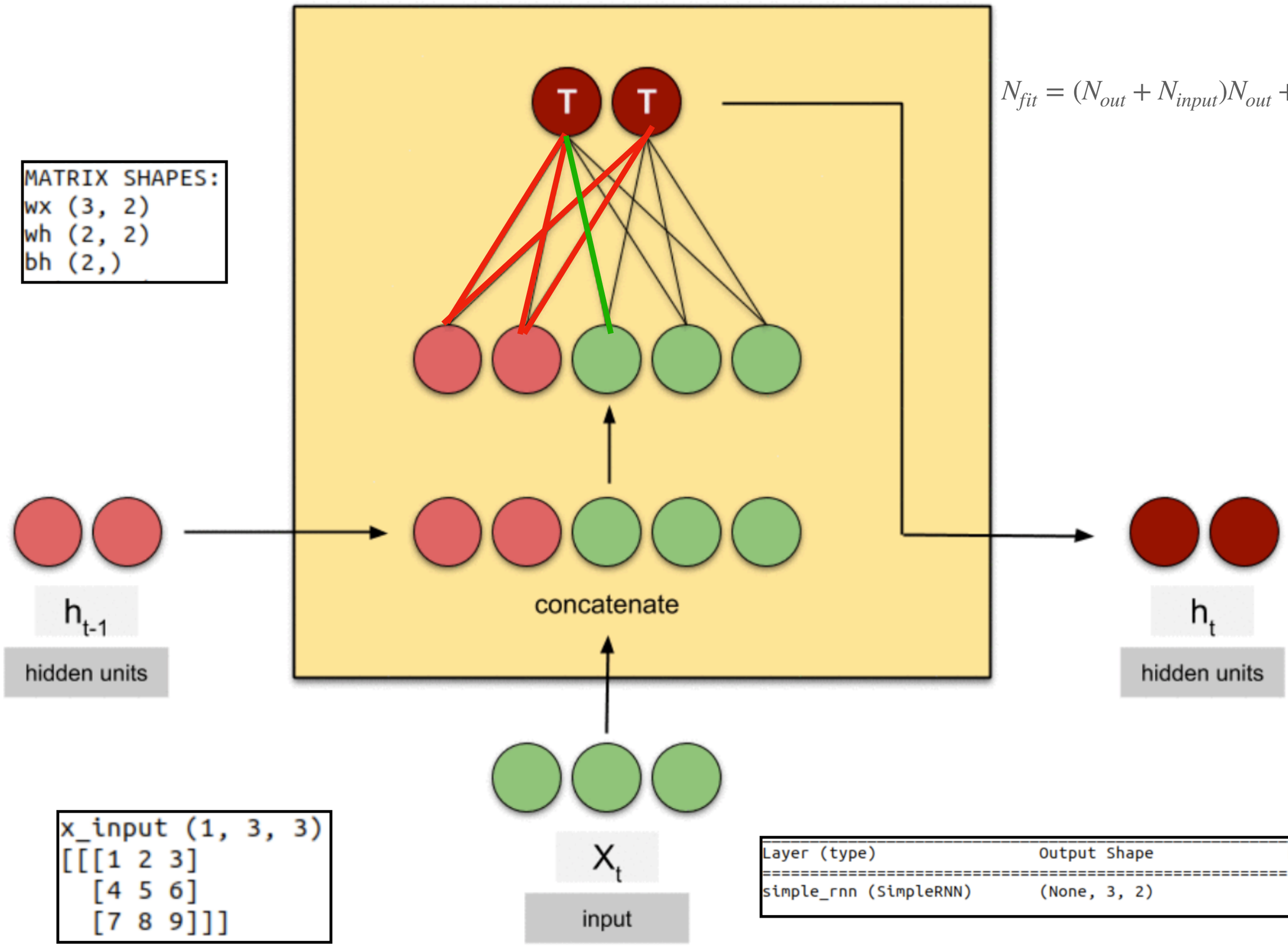


Image Source: https://towardsdatascience.com/animated-rnn-lstm-and-gru-ef124d06cf45?gi=136c797ba8e0

Alternative Representation-2

$$h_t = f(a_t) \Leftrightarrow f(W_h h_{t-1} + W_x^T x_t + b_h)$$

```
MATRIX SHAPES:
wx (3, 2)
wh (2, 2)
bh (2,)
```



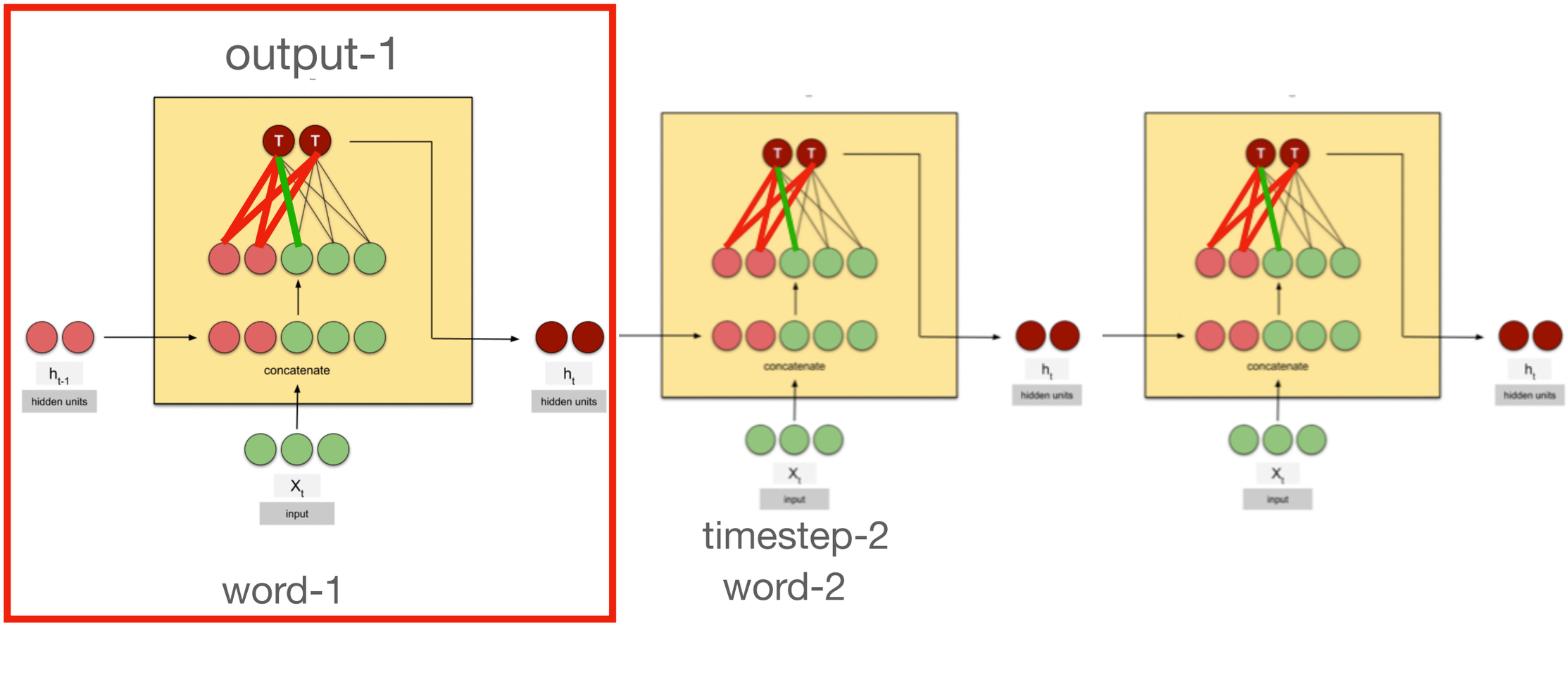
$$N_{fit} = (N_{out} + N_{input})N_{out} + N_{out}$$

```
x_input (1, 3, 3)
[[[1 2 3]
  [4 5 6]
  [7 8 9]]]
```

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 3, 2)	12

Image Source: https://towardsdatascience.com/animated-rnn-lstm-and-gru-ef124d06cf45?gi=136c797ba8e0

Alternative Representation-2



Text classification

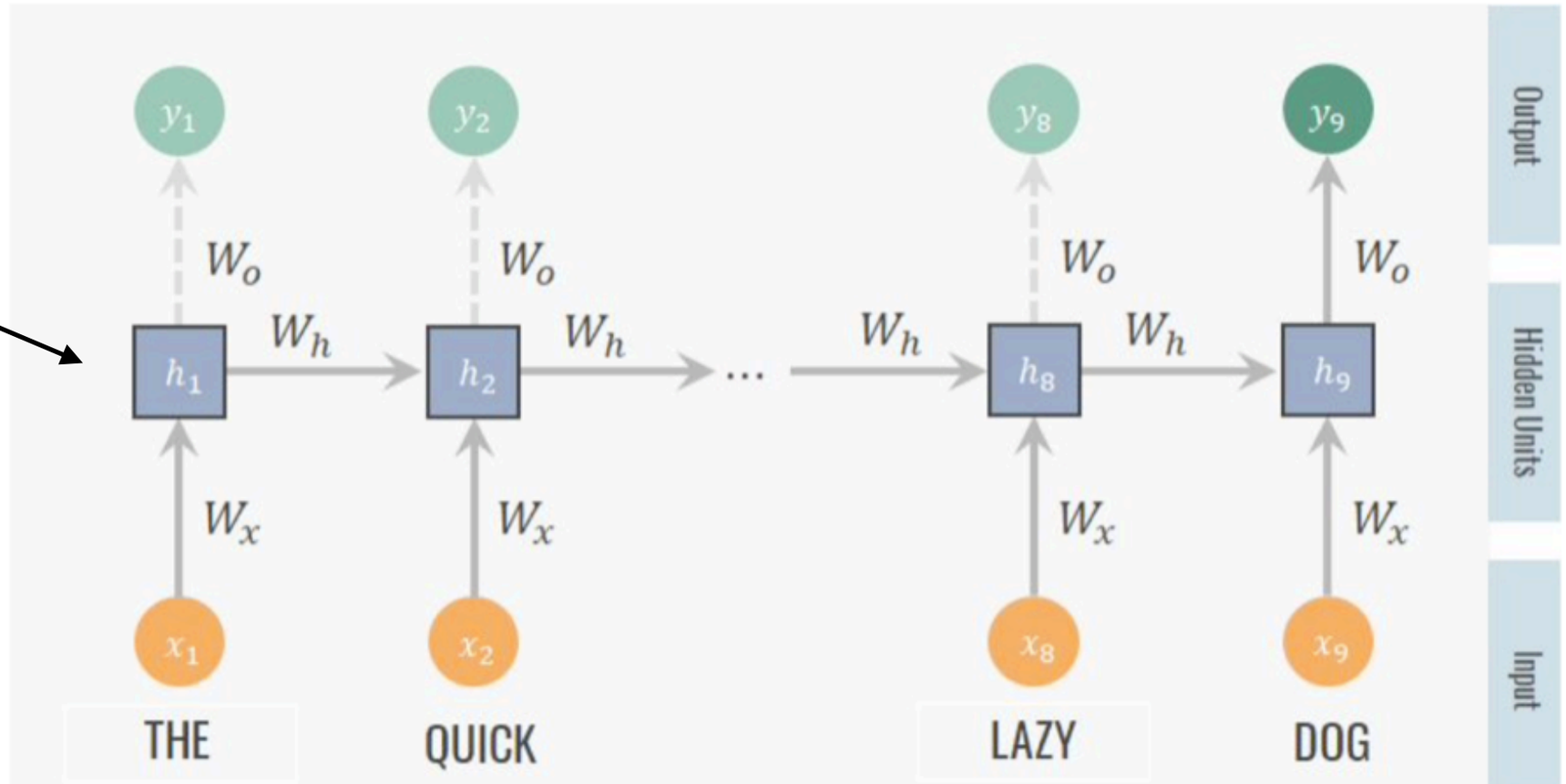
- **Overview**
  - ML technique that assigns a set of predefined categories to open-ended text.
  - Text classifiers can be used to organize, structure, and categorize pretty much any kind of text
    - from documents, medical studies and files, etc.
  - Sentiment analysis, topic labeling, spam-detection, and intent detection.
- **Examples**
  - New articles can be organized by topics (sports, food, etc);
  - Tech-support tickets can be organized by urgency (high, medium, low);
  - Chat conversations can be organized by language
  - Brand mentions can be organized by sentiment
- **Motivation**
  - Around 80% of all information is unstructured.
  - Text data is one of the most common types of unstructured data.
  - Because of the messy nature of text, analyzing, understanding, organizing, and sorting through text data is hard and time-consuming → Therefore most companies fail to use it to its full potential.

Evaluation:

UNROLLED RNN

- h_o is usually initialized to 0

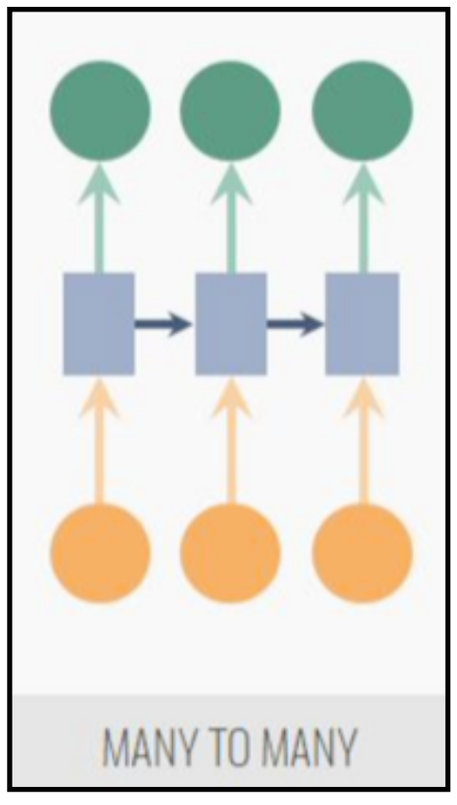
h_o



$$y_t = softmax(W_o^T h_t + b_o)$$

$$h_t = f(a_t) \Leftrightarrow f(W_h h_{t-1} + W_x^T x_t + b_h)$$

$$a_t = W_h h_{t-1} + W_x^T x_t + b_h$$



- Weights are shared in a recurrent net

- In previous situations (i.e. classification) our output was one class, RNNs allow us to output a value at every step

- RNNs are trained using back propagation through time (a type of gradient descent)

Evaluation:

INPUT:

words=features

Time

```
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

sample-2

This movie is the best thing I've ever seen



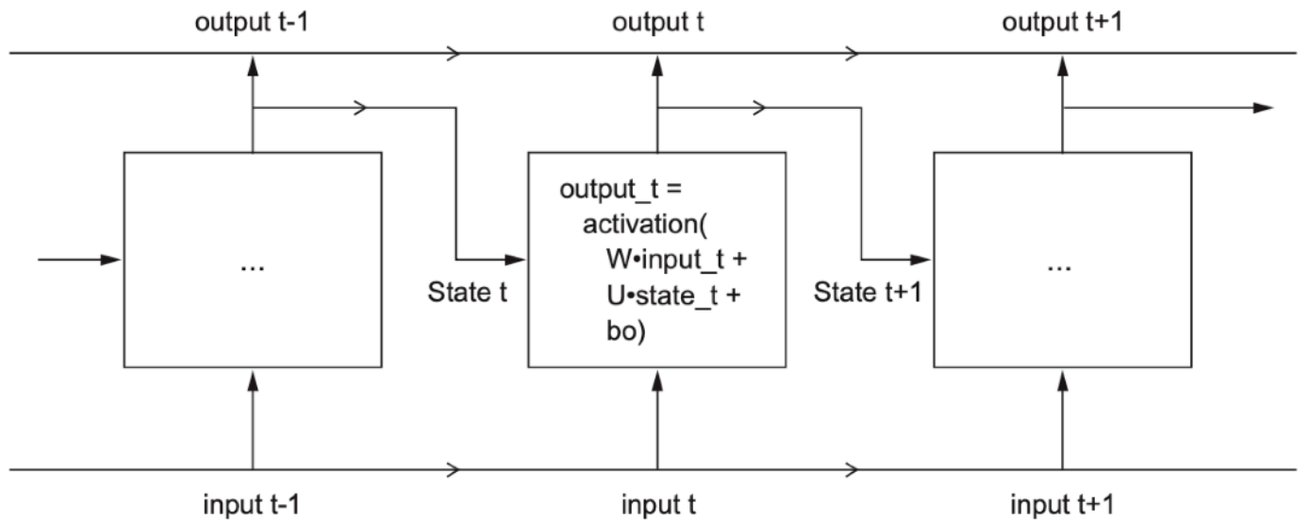
vocabular size=10

max length=9

9x10

ANN fitting parameters:

```
W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))
```



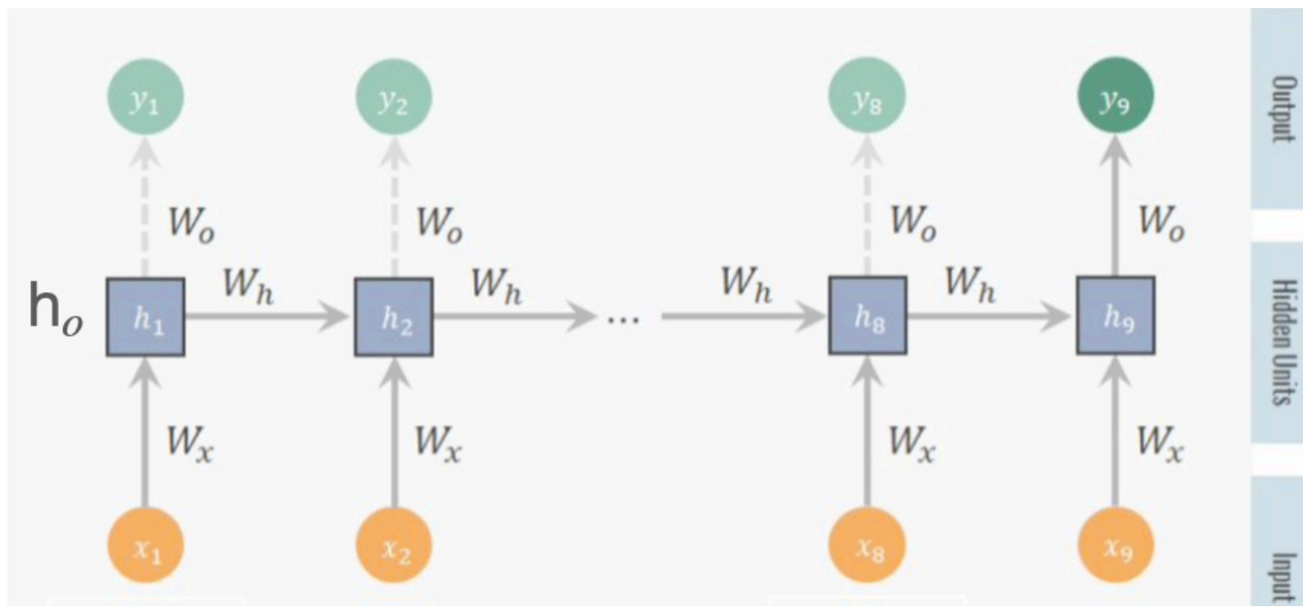

```
#EXPLORE RNN: LISTING 6.21 NUMPY IMPLEMENTATION OF A SIMPLE RNN
timesteps = x.shape[1]
input_features = x.shape[2]
output_features = x.shape[2] #20
print("timesteps,input_features,output_features")
print(timesteps,input_features,output_features)

inputs = x[0]  #np.random.random((timesteps, input_features))


state_t = np.zeros((output_features,)) #INITIALIZE PREVIOUS STATE
W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))
successive_outputs = []
for input_t in inputs: #LOOP OVER TIME
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
    successive_outputs.append(output_t)
    state_t = output_t

    print("input_t:",input_t)
    print("output_t:",output_t)

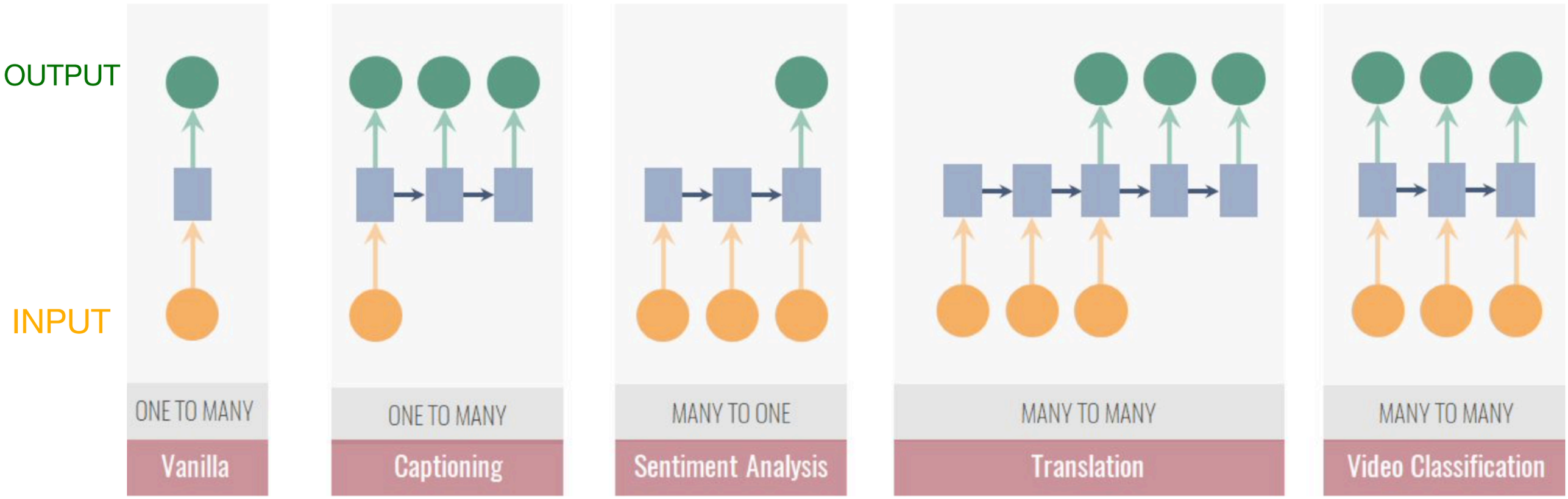
final_output_sequence = np.concatenate(successive_outputs, axis=0)
print("FINAL_OUTPUT_SEQUENCE")
print(final_output_sequence)
```



../../LECTURE-CODES/WEEK8/EXPLORE-CONV1D-TEXT.py

Types of mappings

Different sequence relations for different use cases



The network architecture can have one or many inputs/outputs.
Regardless of the networks structure, sequence information (recurrence) is passed along the hidden layer.

