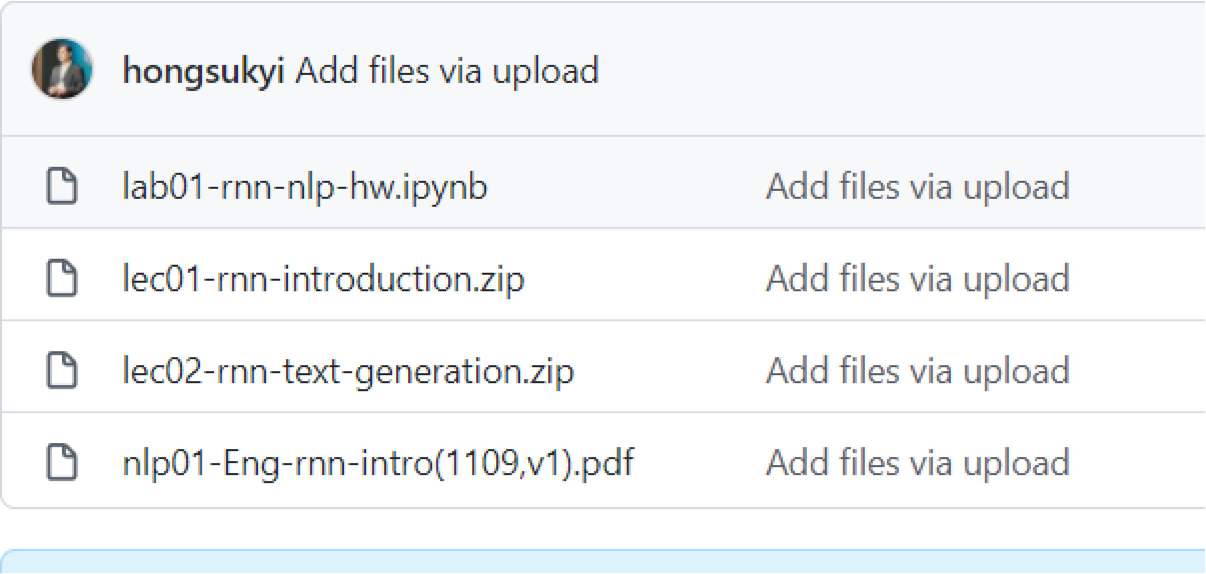
RNN for NLP

RNN, LSTM, GRU

- Hongsuk Yi (KISTI)
- Download material and codes, 강의 자료를 다운로드 받으세요.

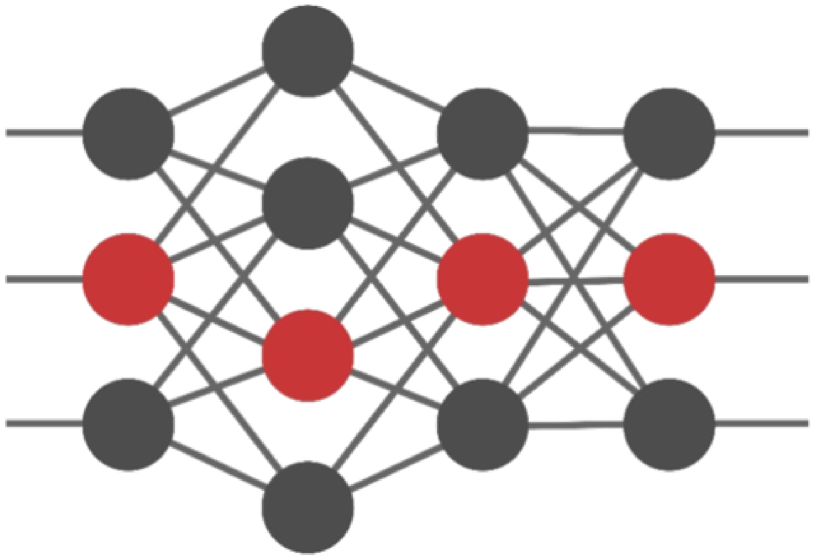
https://github.com/hongsukyi/rnn-nlp-lectures

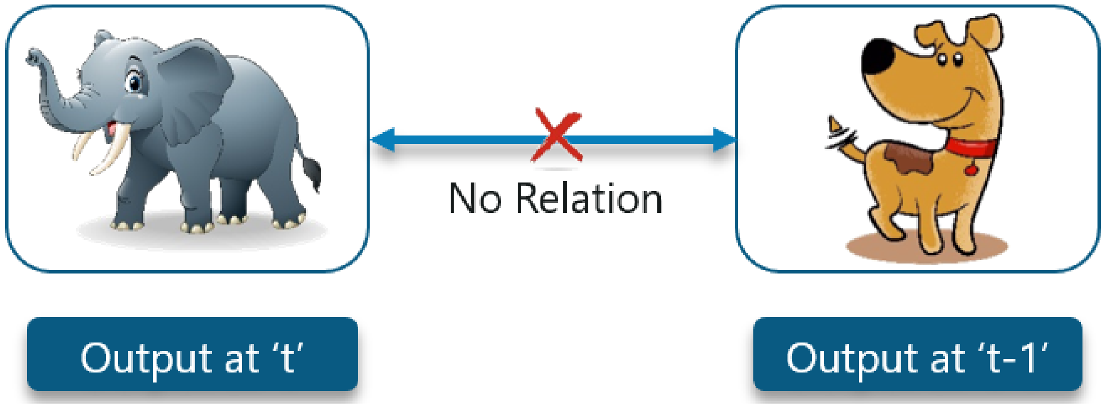


Contents

Why Not Feedforward Networks

- In the existing FeedForWared Networks,
  - ✓ The input is a dog image, and a well-learned neural network outputs a dog label.
  - ✓ The next elephant image is input to output an elephant (label).
  - ✓ However, elephants and dogs are not related to each other and are independent.

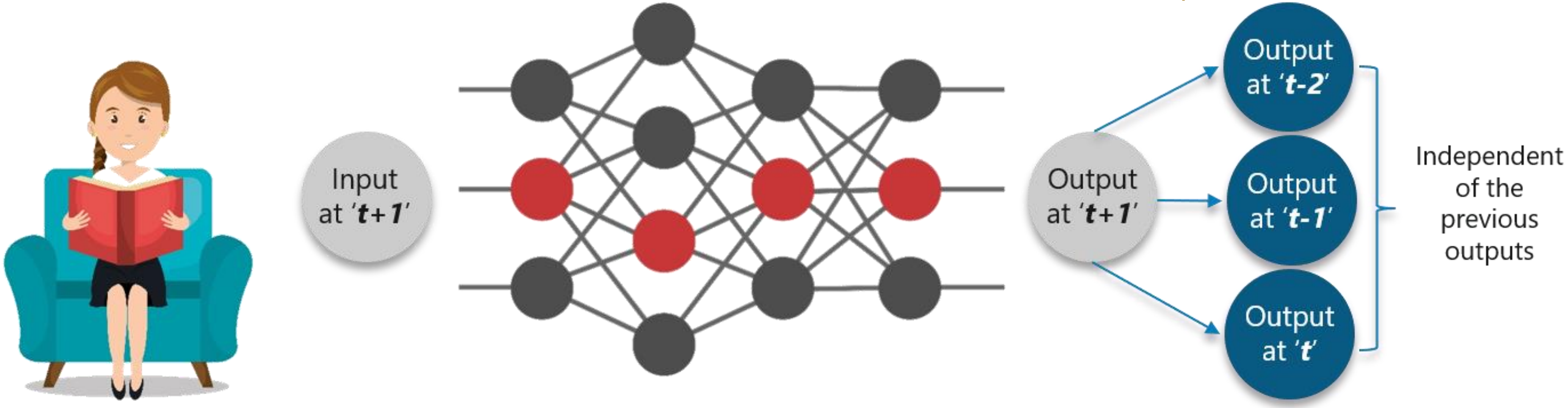




ref: https://www.edureka.co/blog/recurrent-neural-networks/

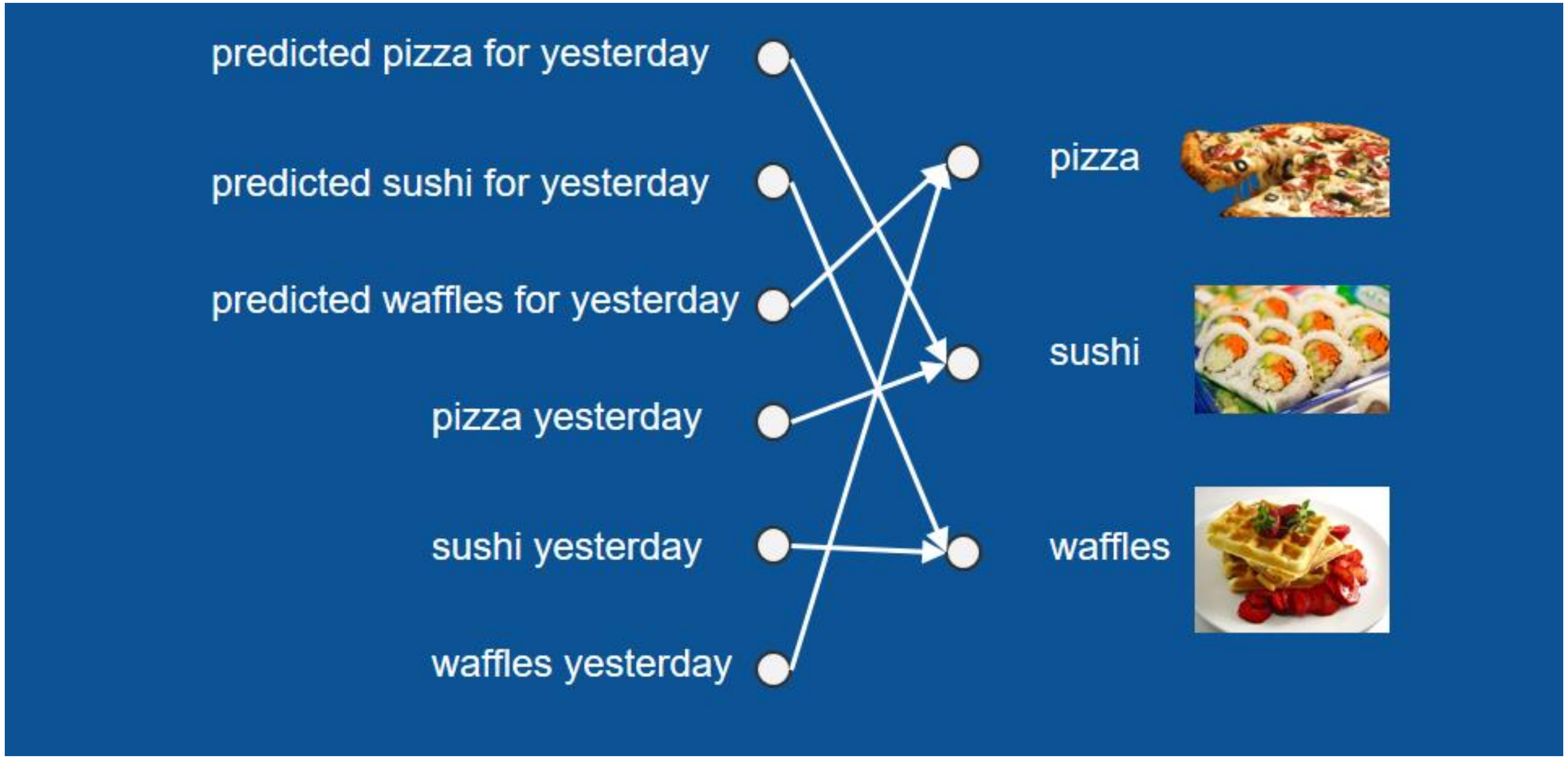
Why Not Feedforward Networks

- If you are reading a book, you should be well aware of the previous page.
  - ✓ In order to understand the context of the book, you must remember the previous contents.
  - ✓ Feeforward network cannot predict the relationship between the following words with previous words.
  - ✓ In other words, there is a need for a network that can remember the previous content (word).



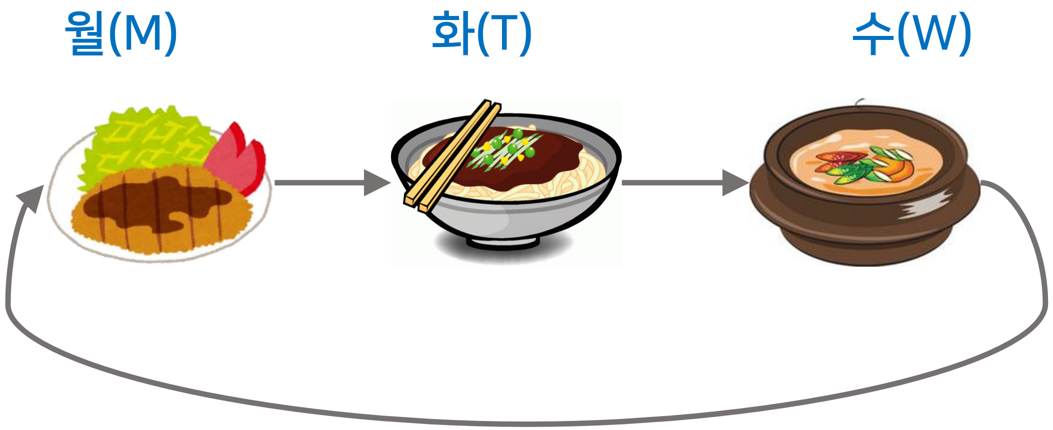
What Are Recurrent Neural Networks?

- Previous memories are also important when eating food
  - ✓ RNNs are a **type** of **artificial neural network** designed to **recognize patterns** in sequences of data



Recurrent Neural Networks : a simple example

- One friend who lives with him cooks only three kinds.
  - ✓ It repeats three foods sequentially: pork cutlet on Mondays, black bean noodles on Tuesdays, and stew on Wednesdays.
  - ✓ The pattern that the roommate remembers is simple.



RNN = Recurrent Neural Network

Recurrent Neural Networks : a simple example

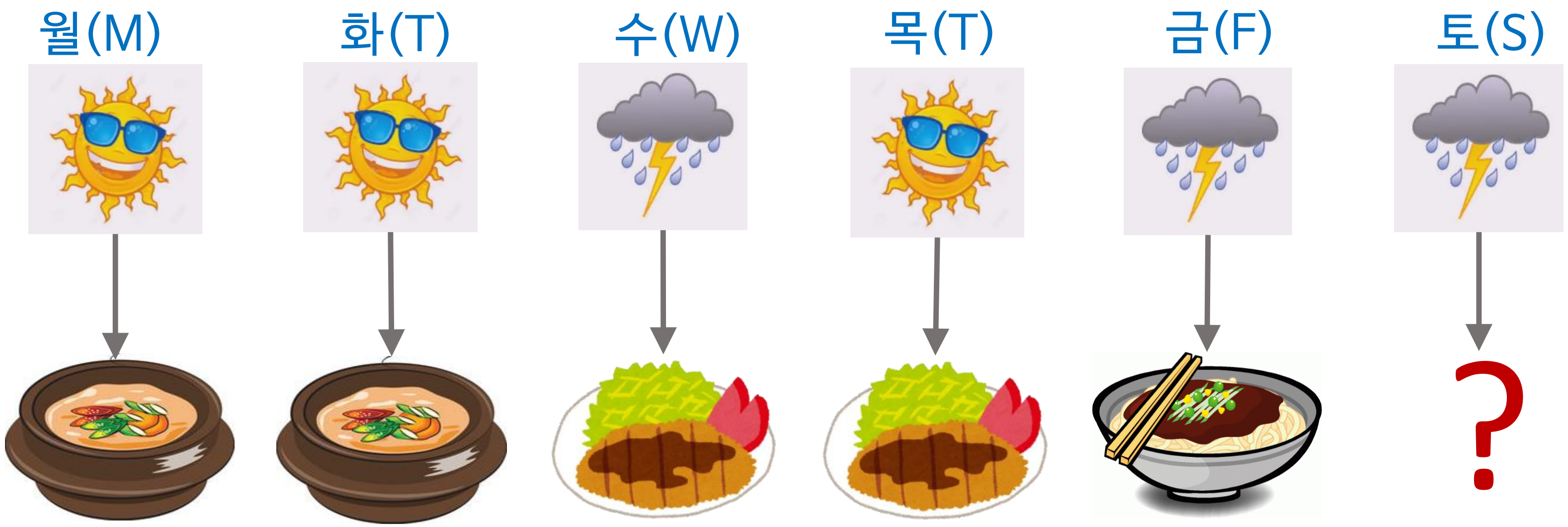
- In fact, roommates have rules on their meal menus depending on the weather.
  - ✓ On a fine day, he goes home late because he is outdoors, so I eat the food I ate the previous day again
  - ✓ However, on rainy days, he return home early and make food according to his patterns.

The food I ate yesterday

The original RNN pattern.

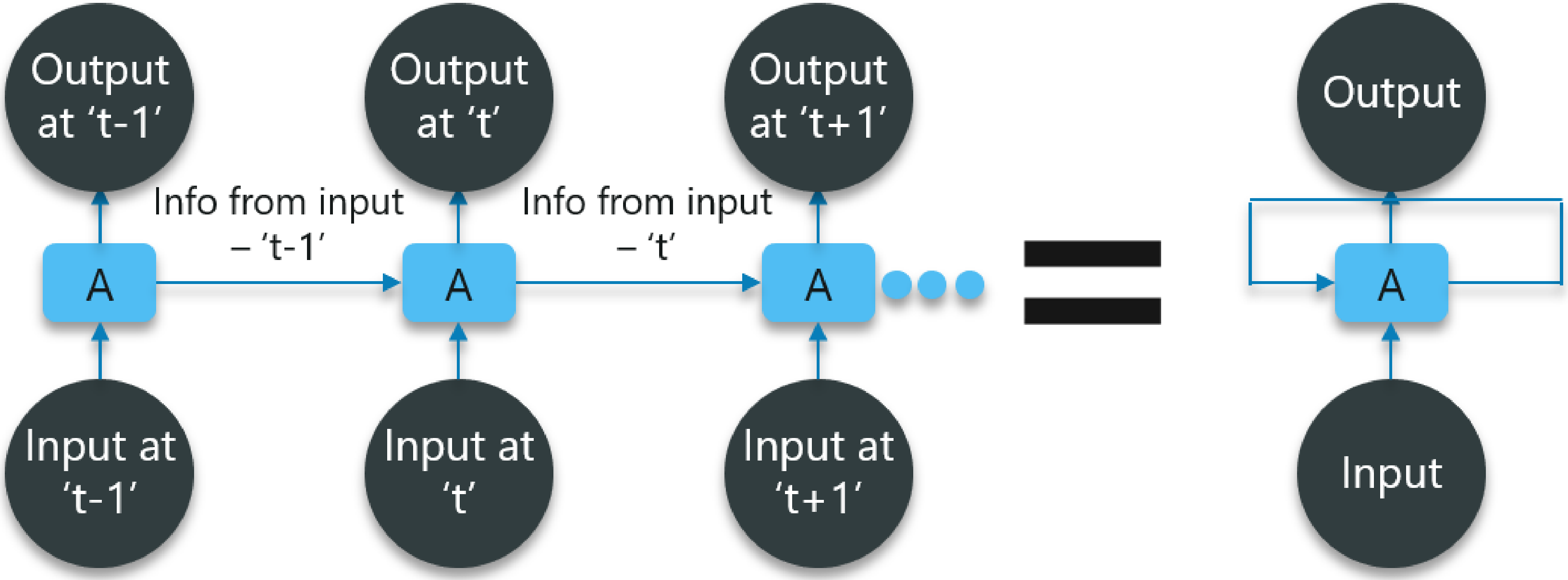
Recurrent Neural Network depending on the weather

- I can predict what to eat in the evening depending on the weather this morning.
  - ✓ On Monday, I ate stew in the evening, and the weather was nice on Tuesday morning, so I ate stew in the evening.
  - ✓ On Friday, it rained, so I ate jajangmyeon, the original pattern.
  - ✓ It rained on Saturday, what was the dinner menu?



RNN diagram

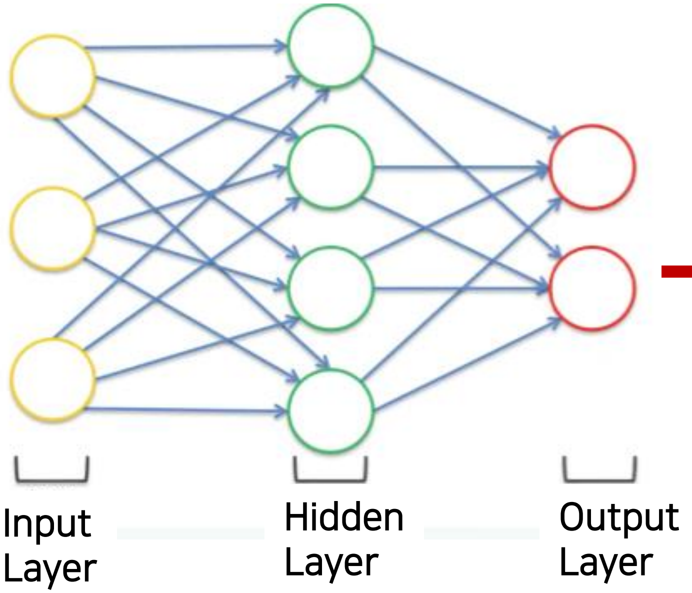
- Let's draw current neural networks (RNNs).
  - ✓ The information (memory pattern) obtained as an input of the previous time t-1 is used as an input at the next time t.
  - ✓ The input of the current time t and the memory of the previous time t-1 are input to output the current time t.



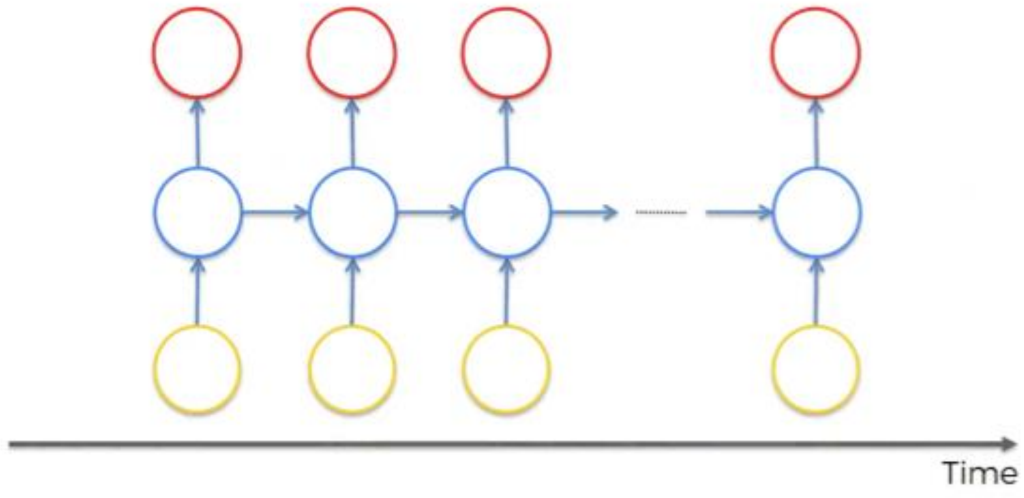
Overview of the feed-forward neural network and RNN structures

- Regarding the difference between the existing ANN and RNN,
  - ✓ Now each circle represents not only one neuron, but a whole layer of neurons.
  - ✓ Hidden layer, the number of neurons, and the number of weights are model parameters.

Unrolling the temporal loop and representing RNNs in a new way.

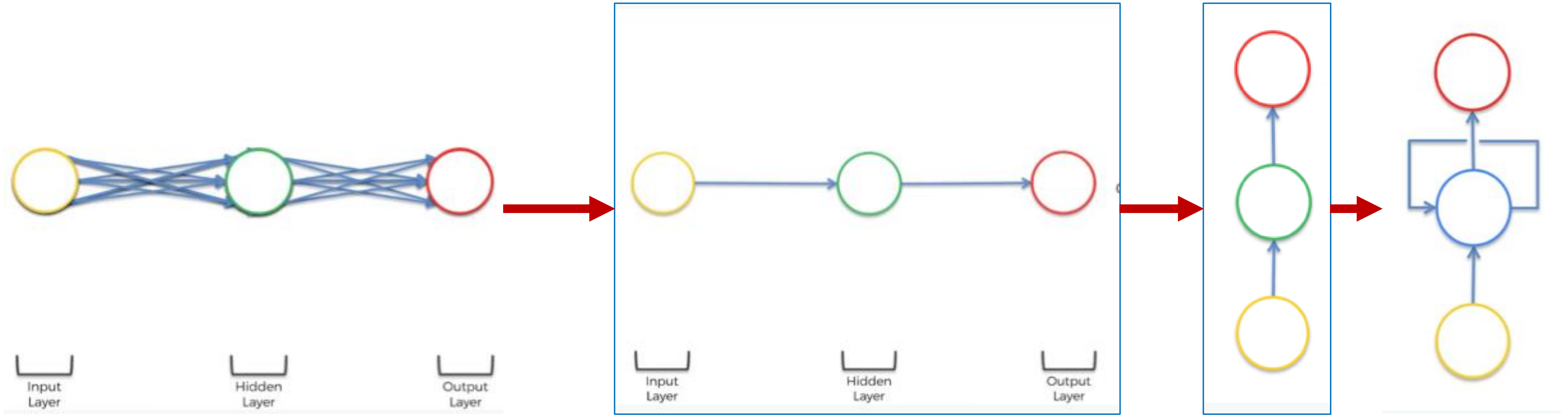


1 Hidden Layer (4 neurons)



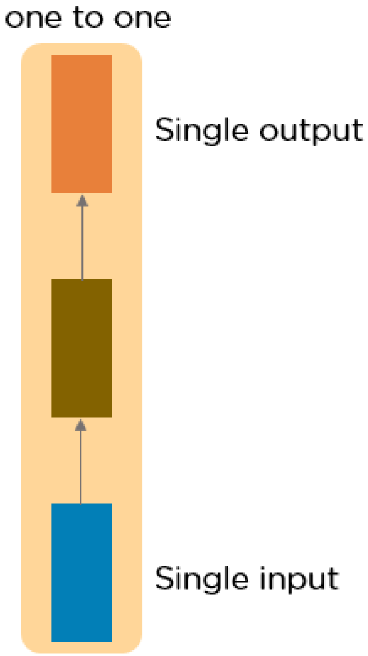
Transformation of a simple ANN into RNN

- Squashing the network.
- Changing the multiple arrows into two.
- Twisting it to make it vertical because that's the standard representation.
- Adding a line, which represents a temporal loop.



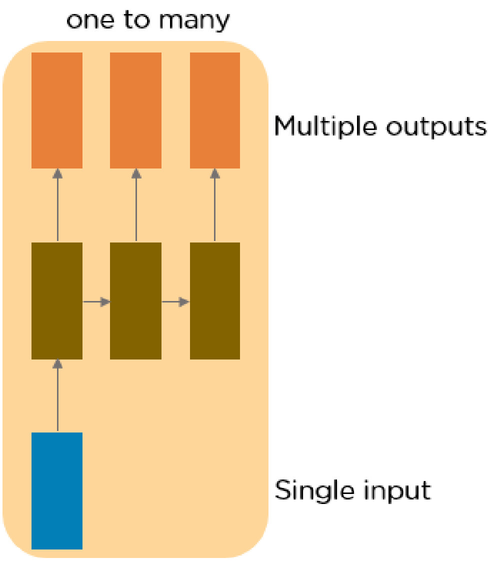
Examples of RNN Application

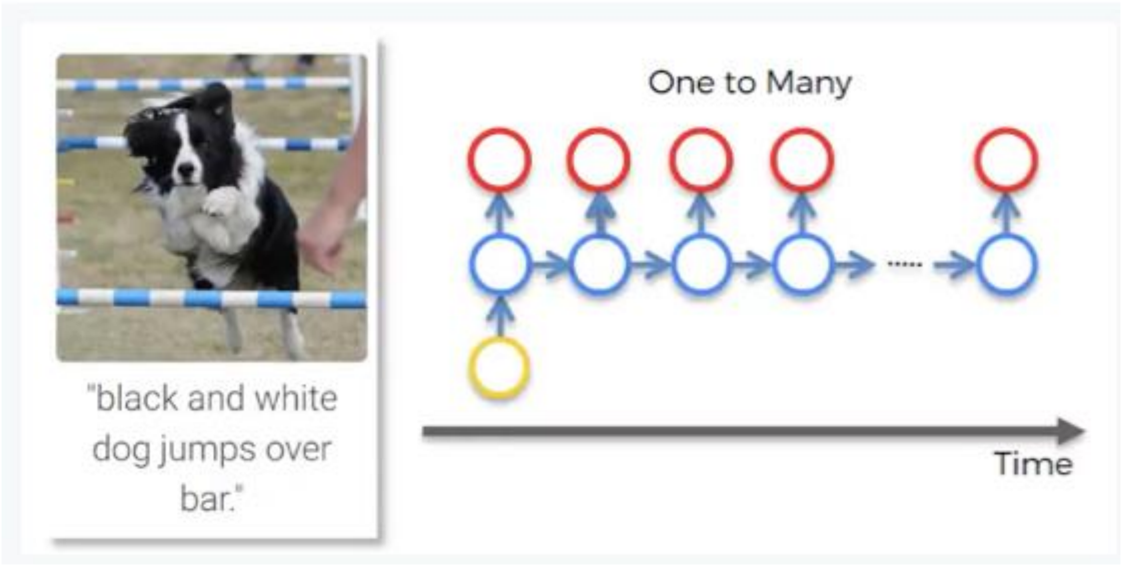
- One to One
  - ✓ This type of neural network is known as the Vanilla Neural Network.
  - ✓ It's used for general machine learning problems, which has a single input and a single output.



Examples of RNN Application

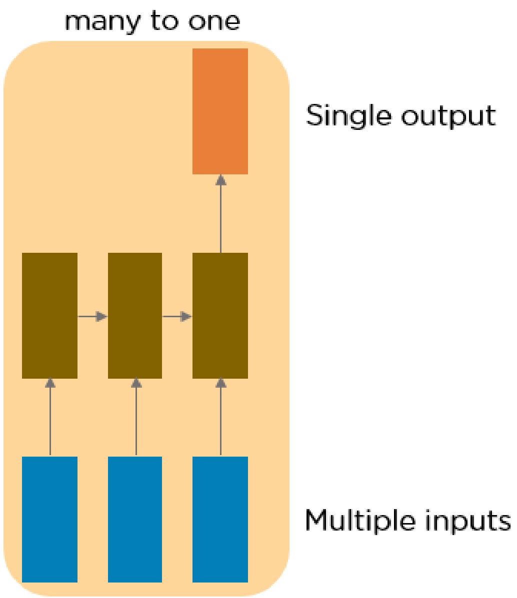
- One to many : This is a network with one input and multiple outputs
  - ✓ it could be an image (input), which is described by a computer with words (outputs).
    - This picture of the dog first went through CNN and then was fed into RNN. The network describes the given picture as “black and white dog jumps over bar”. This is pretty accurate, isn’t it?

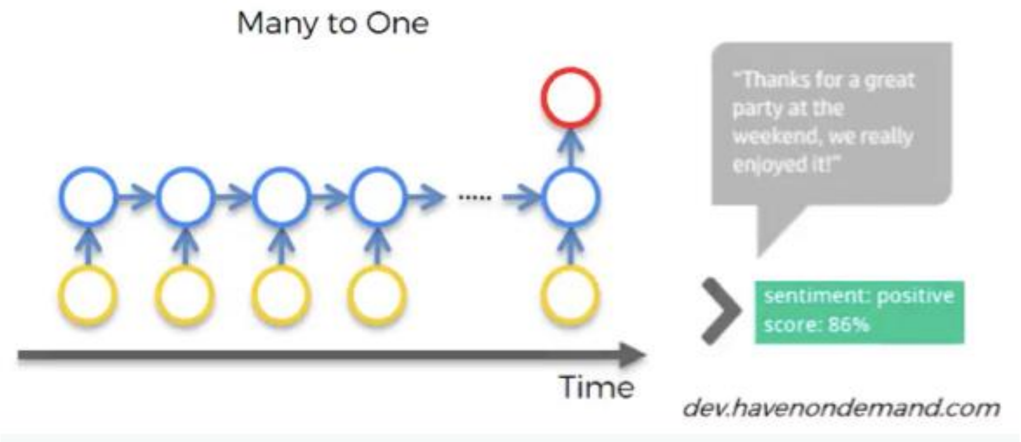




Examples of RNN Application

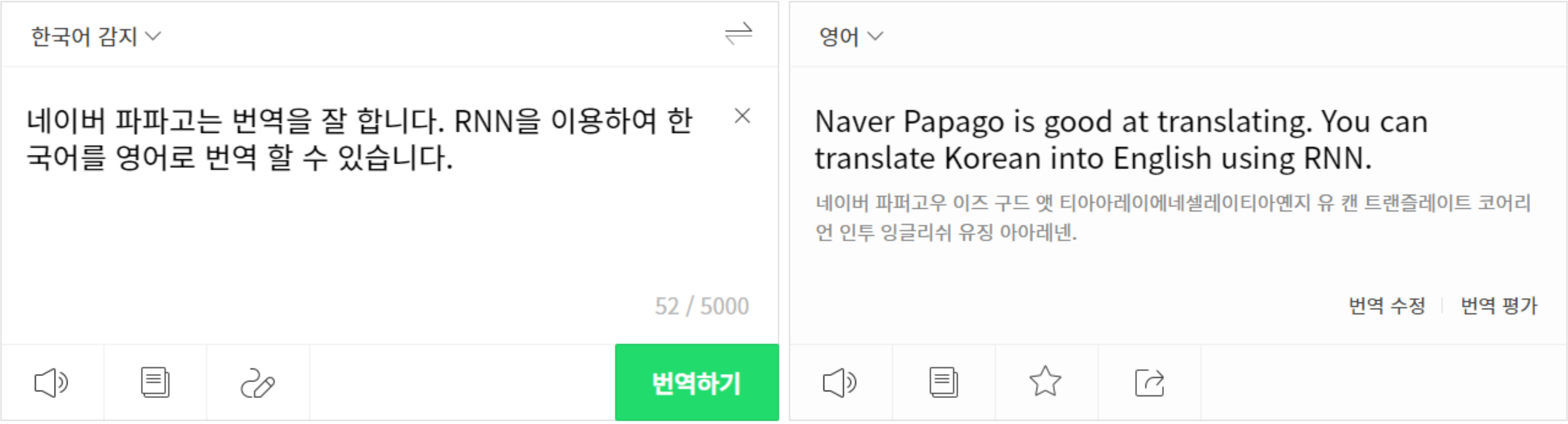
- Many to one : RNN takes a sequence of inputs and generates a single output.
  - ✓ Sentiment analysis is a good example of this kind of network where a given sentence can be classified as expressing positive or negative sentiments.

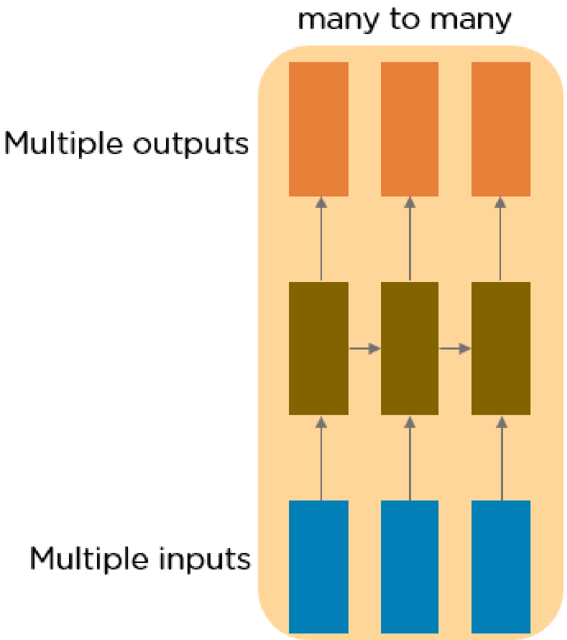




Examples of RNN Application

- Many to many :
  - ✓ RNN takes a sequence of inputs and generates a sequence of outputs.
  - ✓ Machine translation is one of the examples.

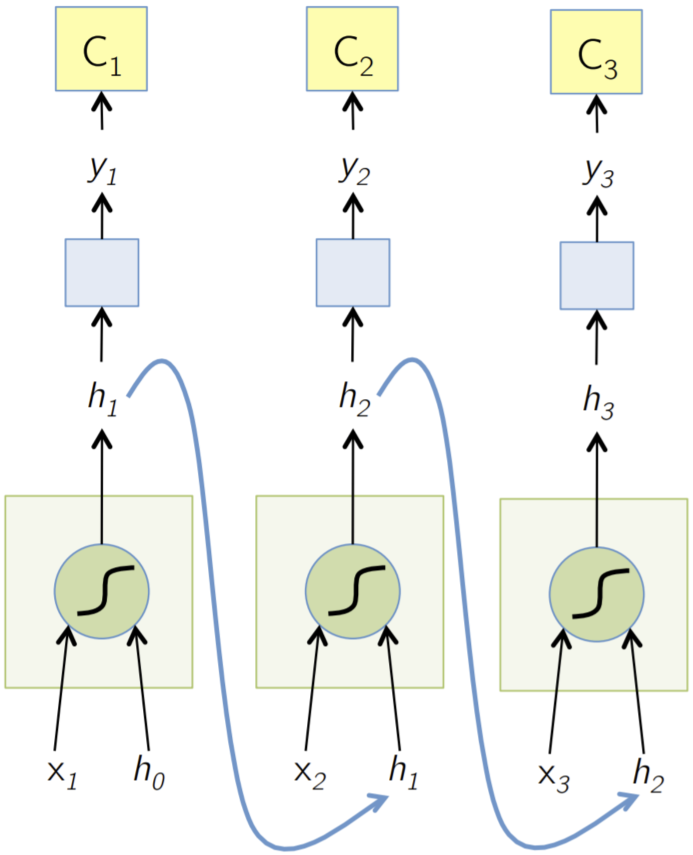




How to train RNN?

Backpropagation Through Time (BPTT)

The Vanilla RNN Forward



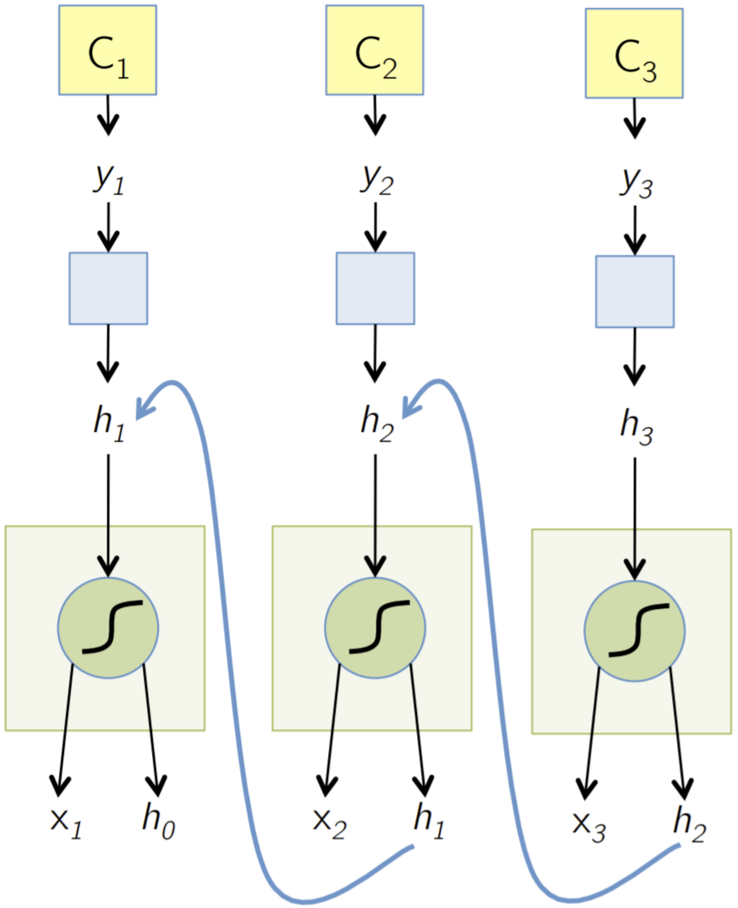
$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$y_t = \mathrm{F}(h_t)$$

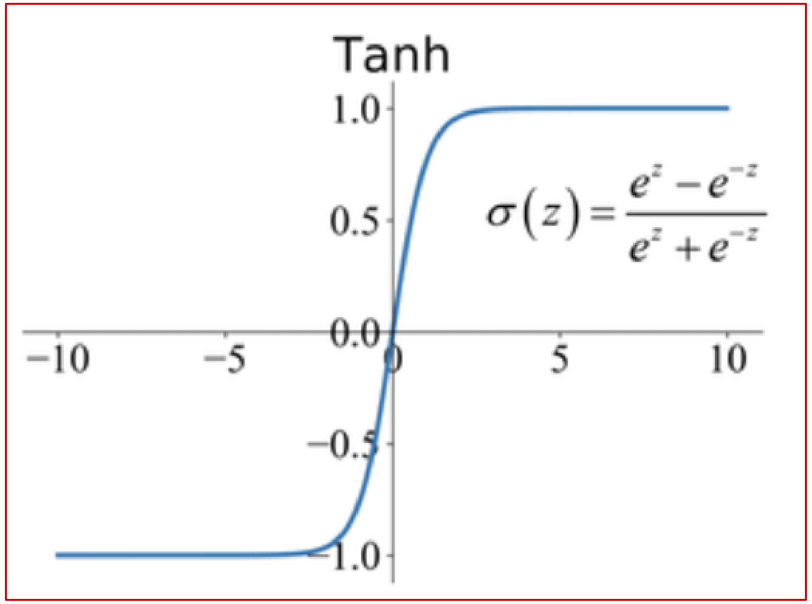
$$C_t = \mathrm{Loss}(y_t, \mathrm{GT}_t)$$

"Unfold" network through time by making copies at each time-step

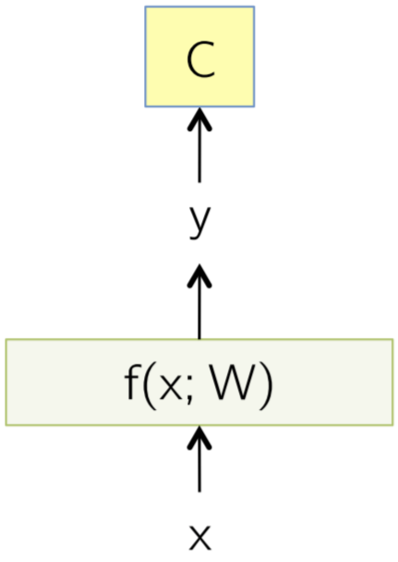
The Unfolded Vanilla RNN Backward



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



BackPropagation Refresher



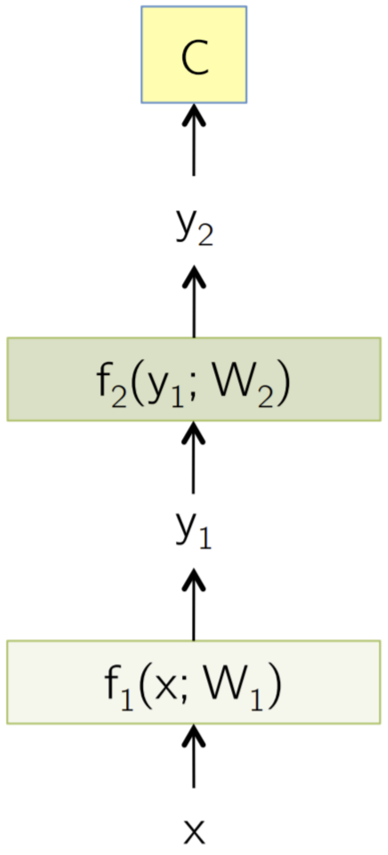
$$y = f(x; W)$$
$$C = \mathrm{Loss}(y, y_{GT})$$

SGD Update

$$W \leftarrow W - \eta \frac{\partial C}{\partial W}$$

$$\frac{\partial C}{\partial W} = \left(\frac{\partial C}{\partial y}\right)\left(\frac{\partial y}{\partial W}\right)$$

Multiple Layers



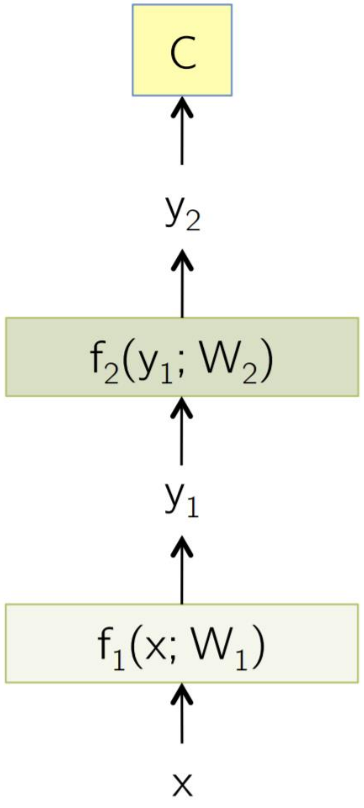
$$y_1 = f_1(x; W_1)$$
$$y_2 = f_2(y_1; W_2)$$
$$C = \text{Loss}(y_2, y_{GT})$$

SGD Update

$$W_2 \leftarrow W_2 - \eta \frac{\partial C}{\partial W_2}$$

$$W_1 \leftarrow W_1 - \eta \frac{\partial C}{\partial W_1}$$

Chain Rule for Gradient Computation



$$y_1 = f_1(x; W_1)$$

$$y_2 = f_2(y_1; W_2)$$

$$C = \mathrm{Loss}(y_2, y_{GT})$$

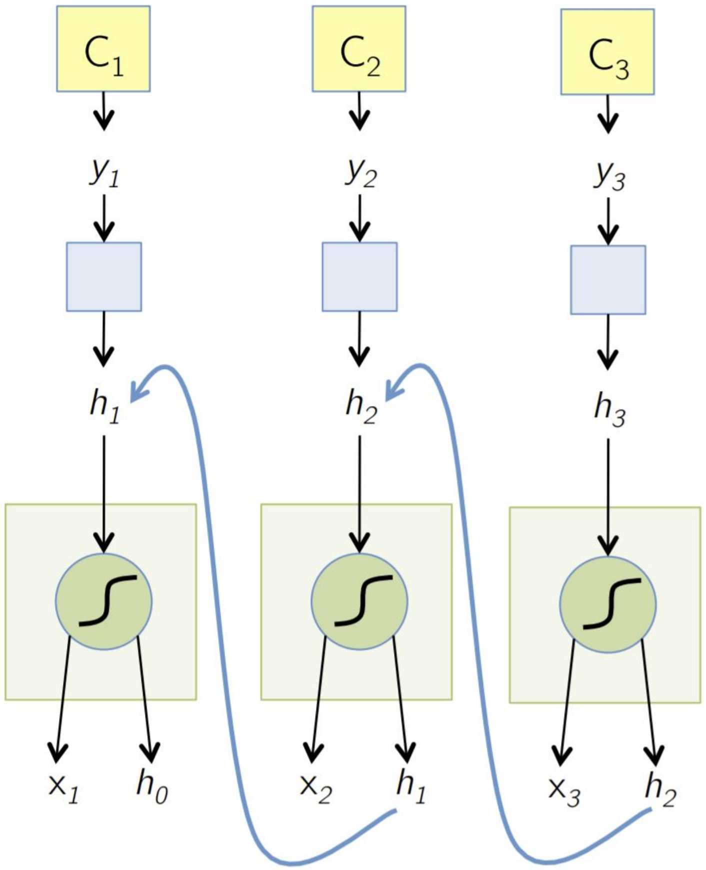
$$\text{Find } \frac{\partial C}{\partial W_1}, \frac{\partial C}{\partial W_2}$$

$$\frac{\partial C}{\partial W_2} = \left(\frac{\partial C}{\partial y_2}\right)\left(\frac{\partial y_2}{\partial W_2}\right)$$

$$\frac{\partial C}{\partial W_1} = \left(\frac{\partial C}{\partial y_1}\right)\left(\frac{\partial y_1}{\partial W_1}\right)$$

$$= \left(\frac{\partial C}{\partial y_2}\right)\left(\frac{\partial y_2}{\partial y_1}\right)\left(\frac{\partial y_1}{\partial W_1}\right)$$

The Vanilla RNN Backward



$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

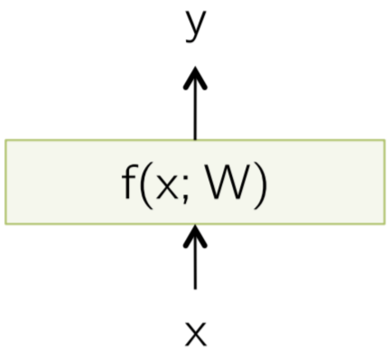
$$y_t = \mathrm{F}(h_t)$$

$$C_t = \mathrm{Loss}(y_t, \mathrm{GT}_t)$$

$$\frac{\partial C_t}{\partial h_1} = \left(\frac{\partial C_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_1}\right)$$

$$= \left(\frac{\partial C_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_t}\right)\left(\frac{\partial h_t}{\partial h_{t-1}}\right)\cdots\left(\frac{\partial h_2}{\partial h_1}\right)$$

Chain Rule for Gradient Computation



Given: $\left(\frac{\partial C}{\partial y}\right)$

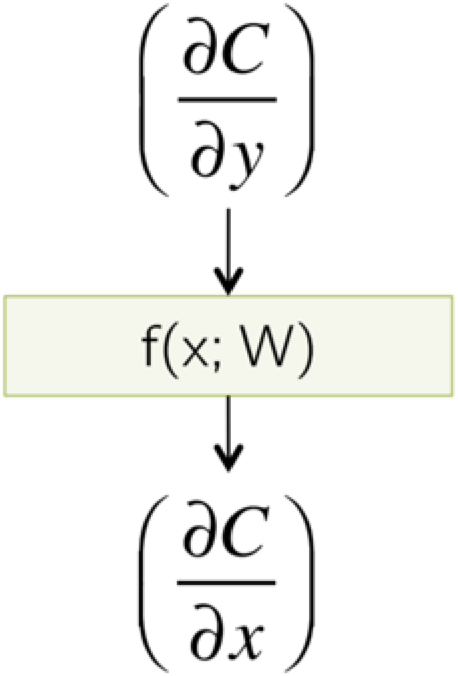
We are interested in computing: $\left(\frac{\partial C}{\partial W}\right), \left(\frac{\partial C}{\partial x}\right)$

Intrinsic to the layer are:

$\left(\frac{\partial y}{\partial W}\right)$ – How does output change due to params

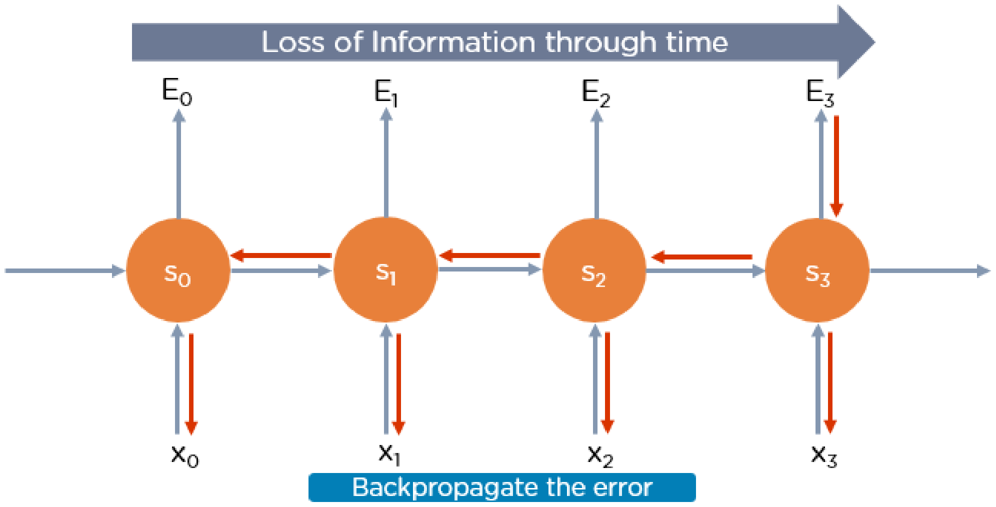
$\left(\frac{\partial y}{\partial x}\right)$ – How does output change due to inputs

$$\left(\frac{\partial C}{\partial W}\right) = \left(\frac{\partial C}{\partial y}\right)\left(\frac{\partial y}{\partial W}\right) \qquad \left(\frac{\partial C}{\partial x}\right) = \left(\frac{\partial C}{\partial y}\right)\left(\frac{\partial y}{\partial x}\right)$$



Backpropagation Through Time (BPTT)

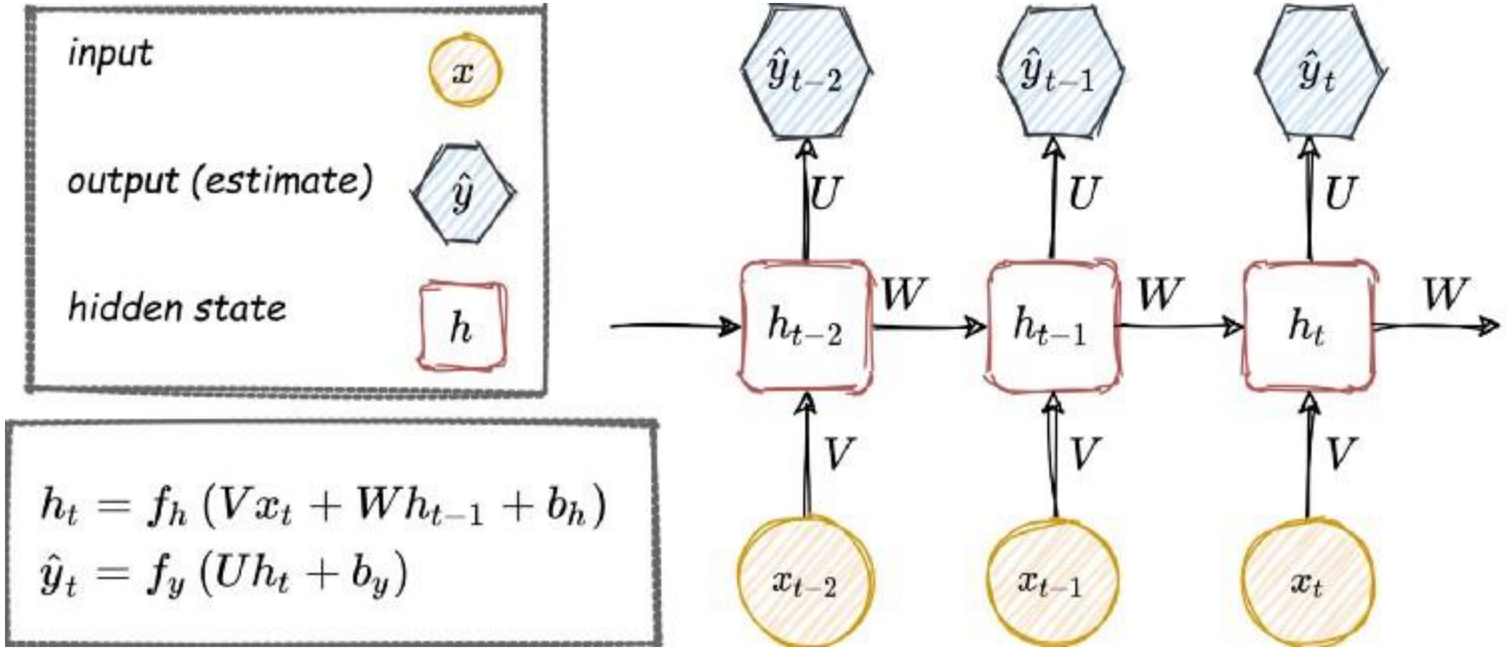
- The training of RNN is not trivial, as we backpropagate gradients through layers and also through time



$$\frac{\partial \mathrm{L}}{\partial W} \propto \sum_{i=0}^{T} \left(\prod_{i=k+1}^{y} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W}$$

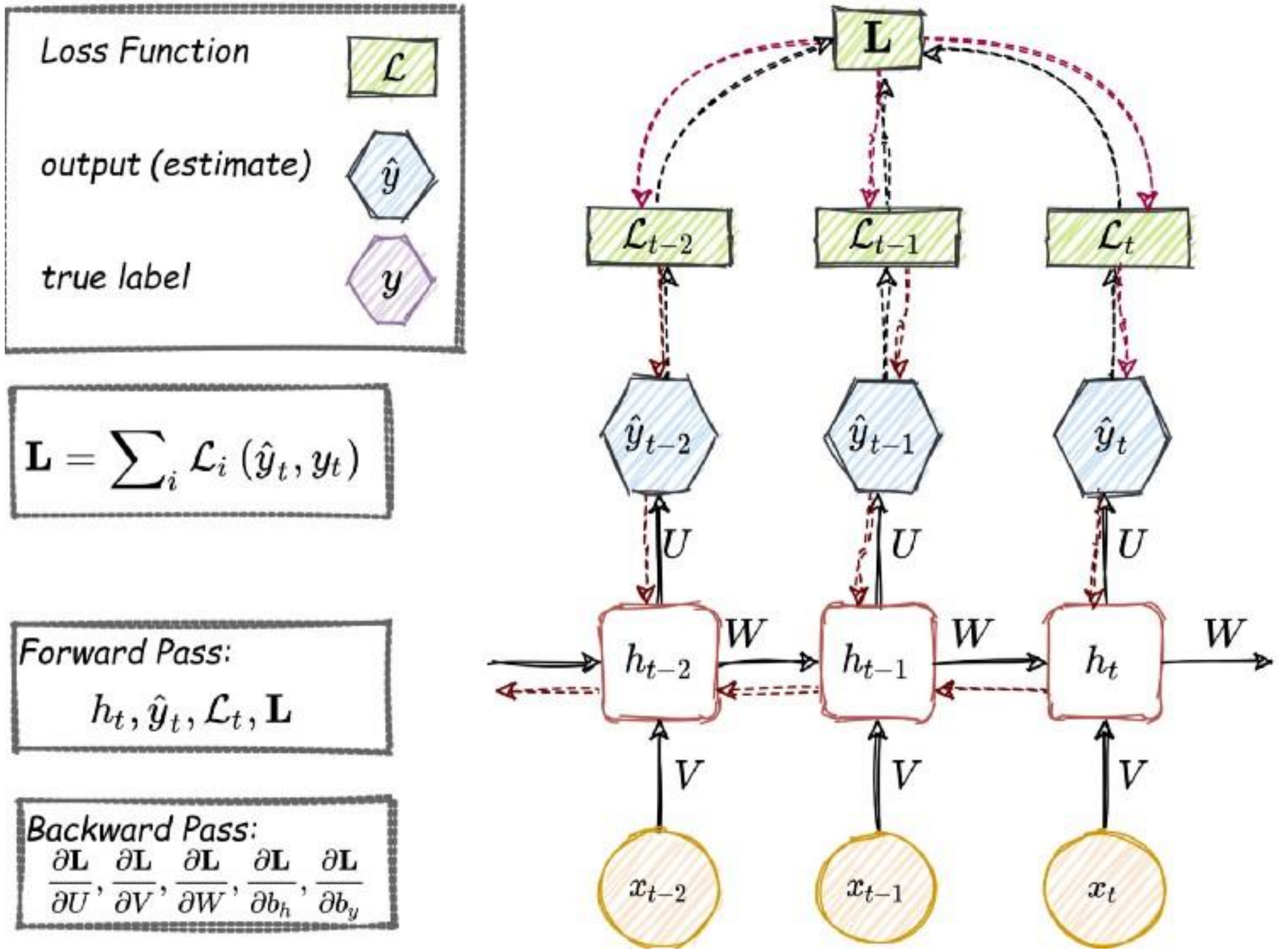
Training for Recurrent Neural Networks

- A simple RNN architecture is shown below, where V, W, and U are the weights matrices, and b is the bias vector.



$$h_t = f_h\left(Vx_t + Wh_{t-1} + b_h\right)$$
$$\hat{y}_t = f_y\left(Uh_t + b_y\right)$$

Backpropagation Through Time (BPTT)



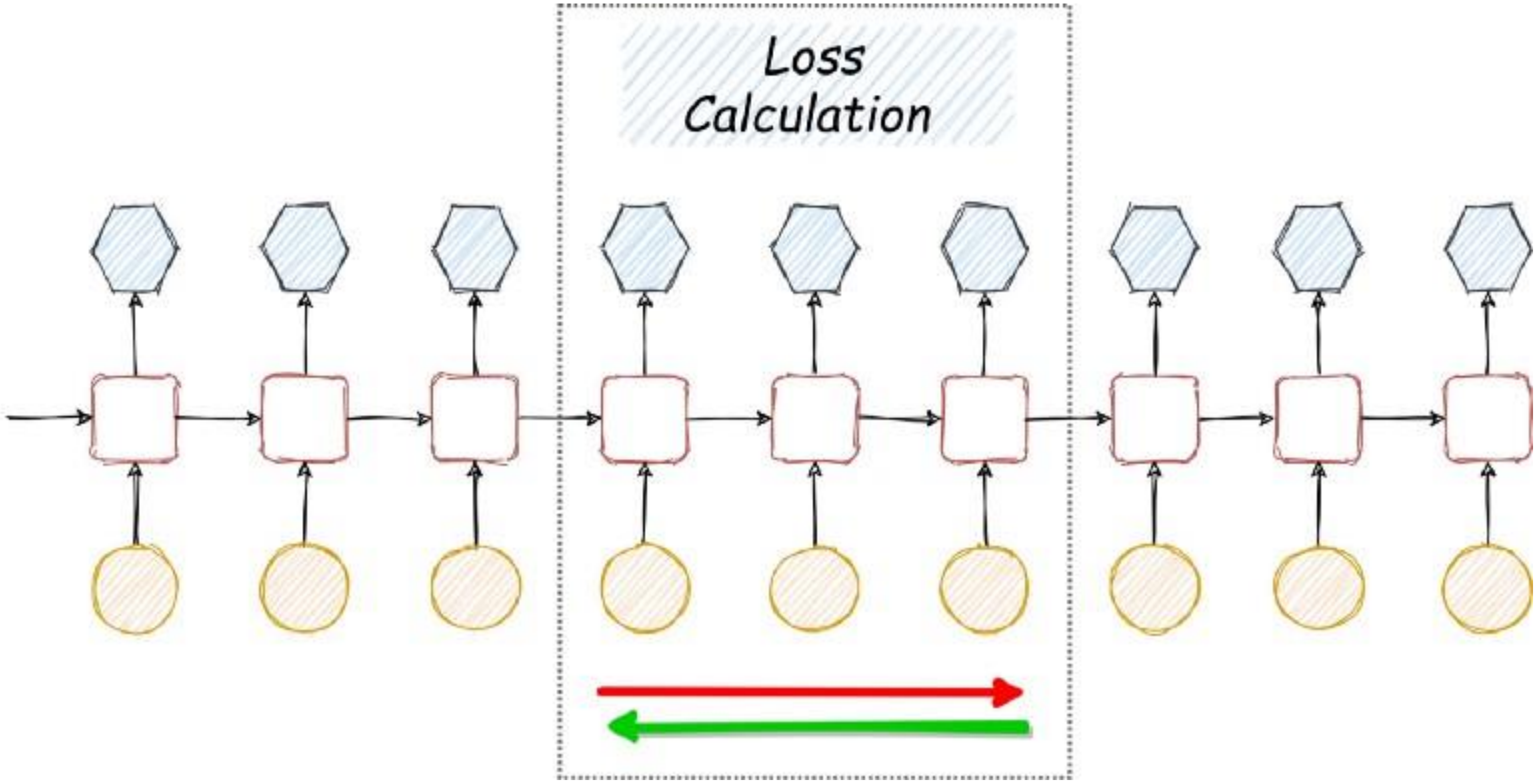
RNNs suffer from the problem of VGP and EGP

- VGP(Vanishing Gradient Problem)
  - ✓ when the gradient becomes too small, the parameter updates become insignificant. This makes the learning of long data sequences difficult.
- EGP(Exploding Gradient Problem)
  - ✓ If the slope tends to grow exponentially instead of decaying, when large error gradients accumulate during the training process.

1. *Vanishing gradient* $\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 < 1$
2. *Exploding gradient* $\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 > 1$

Truncated Backpropagation Through Time (Truncated BPTT)

- Truncated BPTT trick tries to overcome the VGP
  - ✓ by considering a moving window through the training process



Character-level language model

A simple Example

"Modeling word probabilities is really difficult"

Training Sequence modelling

	Supervised learning	Sequence modelling
Data	$\{x, y\}_i$	$\{x\}_i$
Model	$y \approx f_\theta(x)$	$p(x) \approx f_\theta(x)$
Loss	$\mathcal{L}(\theta) = \sum_{i=1}^{N} l(f_\theta(x_i), y_i)$	$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log p(f_\theta(x_i))$
Optimisation	$\theta^* = \arg\min_\theta \mathcal{L}(\theta)$	$\theta^* = \arg\max_\theta \mathcal{L}(\theta)$

Modeling p(x)

Simplest model:
Assume independence of words

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$$

p("modeling") × p("word") × p("probabilities") × p("is") × p("really") × p("difficult")

Word	$p(x_i)$
the	0.049
be	0.028
...	...
really	0.0005
...	...

Modeling p(x)

More realistic model:

Assume conditional dependence of words

$$p(x_T) = p(x_T | x_1, \ldots, x_{T-1})$$

Modeling word probabilities is really ?

Context | Target

Target	p(x|context)
difficult	0.01
hard	0.009
fun	0.005
...	...
easy	0.00001

Modeling p(x)

The chain rule

Computing the joint p(**x**) from conditionals

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1})$$

Modeling

Modeling **word**

Modeling word **probabilities**

Modeling word probabilities **is**

Modeling word probabilities is **really**

Modeling word probabilities is really **difficult**

$$p(x_1)$$

$$p(x_2 | x_1)$$

$$p(x_3 | x_2, x_1)$$

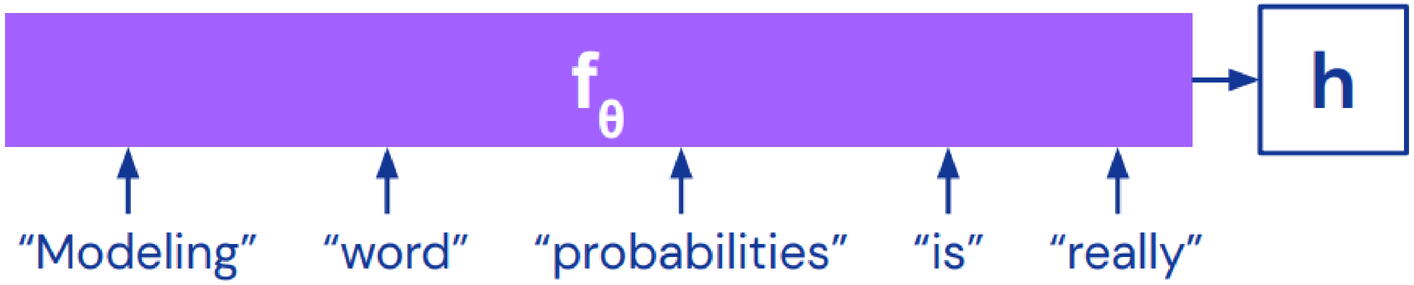
$$p(x_4 | x_3, x_2, x_1)$$

$$p(x_5 | x_4, x_3, x_2, x_1)$$

$$p(x_6 | x_5, x_4, x_3, x_2, x_1)$$

Recurrent Neural Networks (RNNs)

- **Learning to model word probabilities**
  - ✓ Vectorising the context



f_θ summarises the context in $\boxed{h}$ such that:

$$p(x_t | x_1, \ldots, x_{t-1}) \approx p(x_t | h)$$

Desirable properties for f_θ:

- Order matters
- Variable length
- Learnable (differentiable)

Recurrent Neural Networks (RNNs)

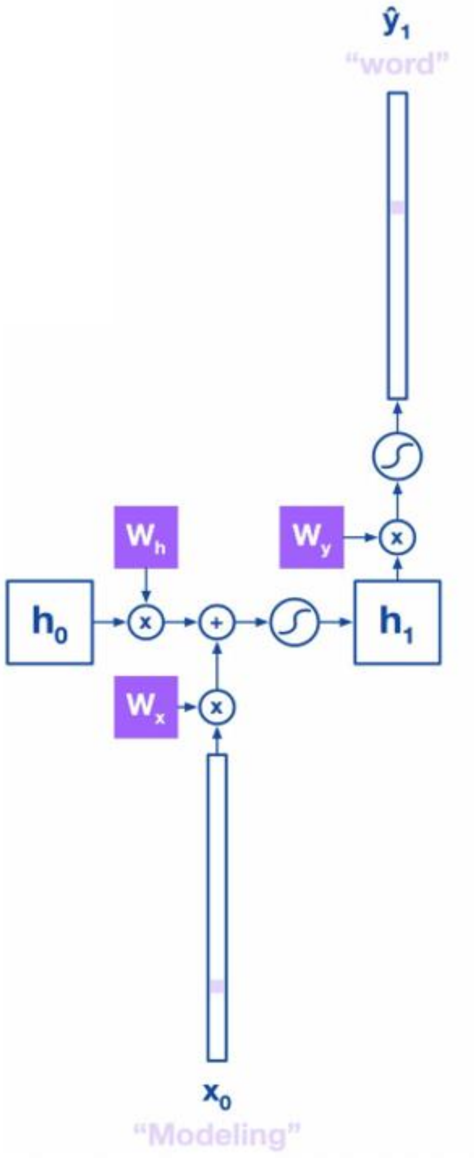
- Persistent state variable h stores information from the context observed so far.

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t)$$

RNNs predict the target **y** (the next word) from the state **h.**

$$p(\mathbf{y_{t+1}}) = softmax(\mathbf{W}_y \mathbf{h}_t)$$

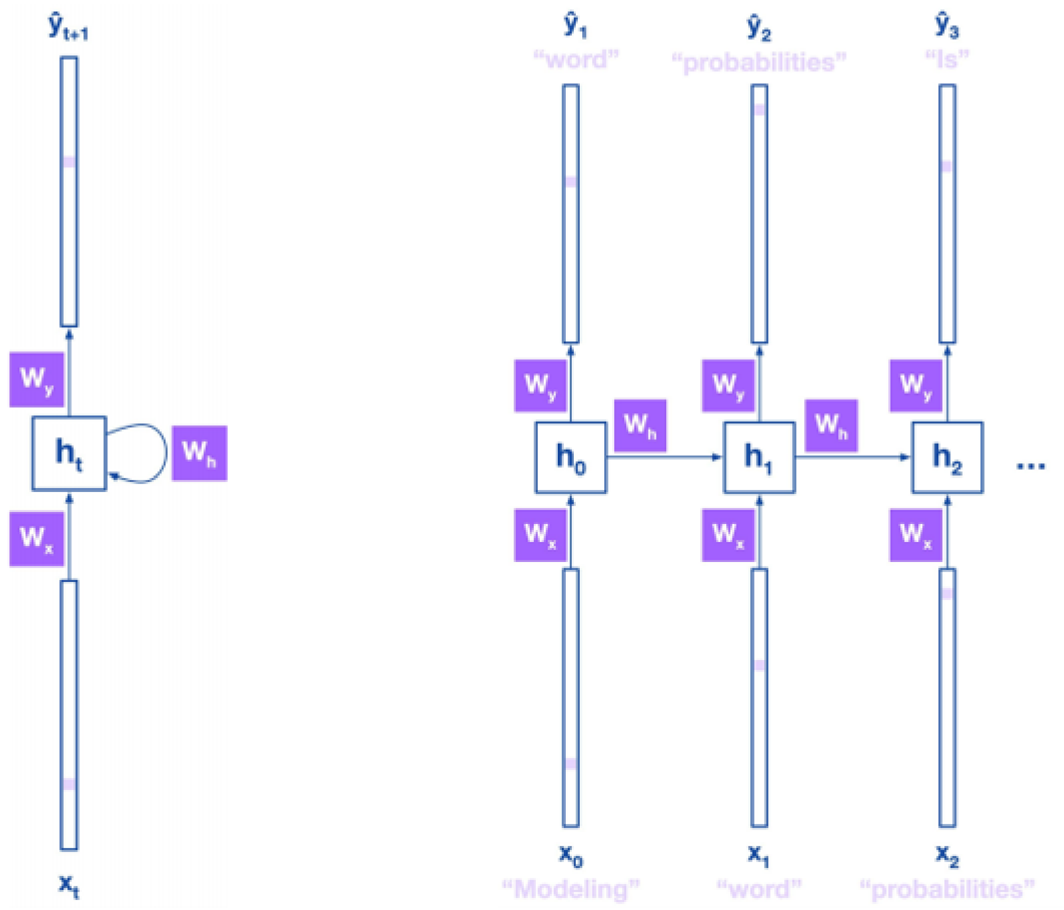
Softmax ensures we obtain a distribution over all possible words.



Recurrent Neural Networks (RNNs)

Weights are shared over time steps

Input next word in sentence x_1



RNN

RNN rolled out over time

Loss: Cross Entropy

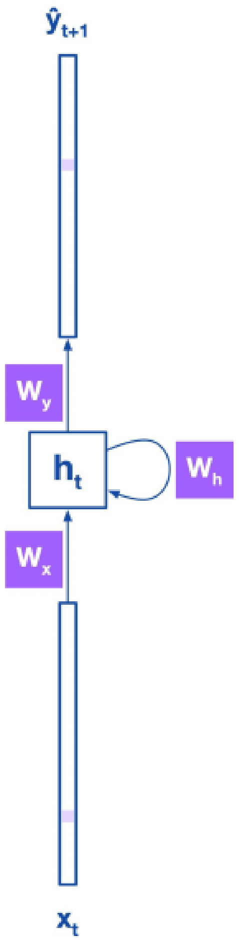
Next word prediction is essentially a classification task where the number of classes is the size of the vocabulary.

As such we use the cross-entropy loss:

For one word: $$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}})_t = -\mathbf{y}_t \log \hat{\mathbf{y}}_t$$

For the sentence: $$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{t=1}^{T} \mathbf{y}_t \log \hat{\mathbf{y}}_t$$

With parameters $\theta = \{\mathbf{W}_y, \mathbf{W}_x, \mathbf{W}_h\}$



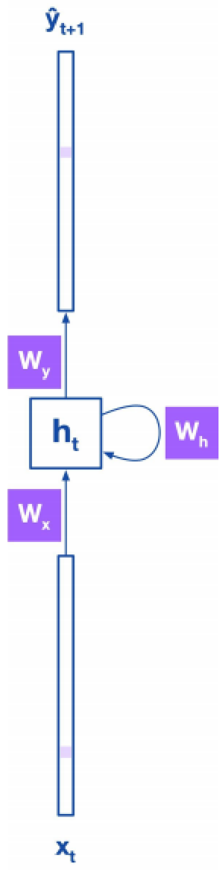
Differentiating weights (w_y, w_x, w_h) from each other

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t)$$

$$p(\mathbf{x}_{t+1}) = softmax(\mathbf{W}_y \mathbf{h}_t)$$

$$\mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}})_t = -\mathbf{y}_t \log \hat{\mathbf{y}}_t$$

$$\frac{\partial \mathbf{L}}{\partial W} = \sum_{i=0}^{T} \frac{\partial \mathcal{L}_i}{\partial W} \propto \sum_{i=0}^{T} \left(\prod_{i=k+1}^{y} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W}$$

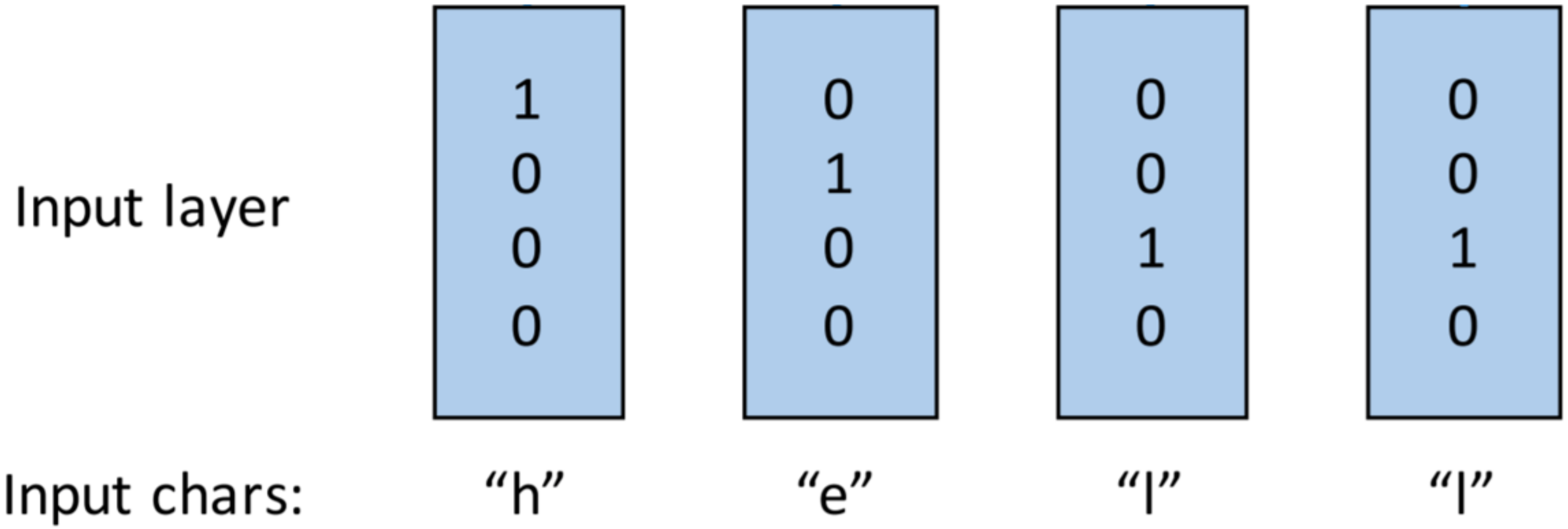


Character-Level Language Models

- We'll give the RNN a huge chunk of text and ask it to model the probability distribution of the next character in the sequence given a sequence of previous characters.
  - ✓ This will then allow us to generate new text one character at a time.

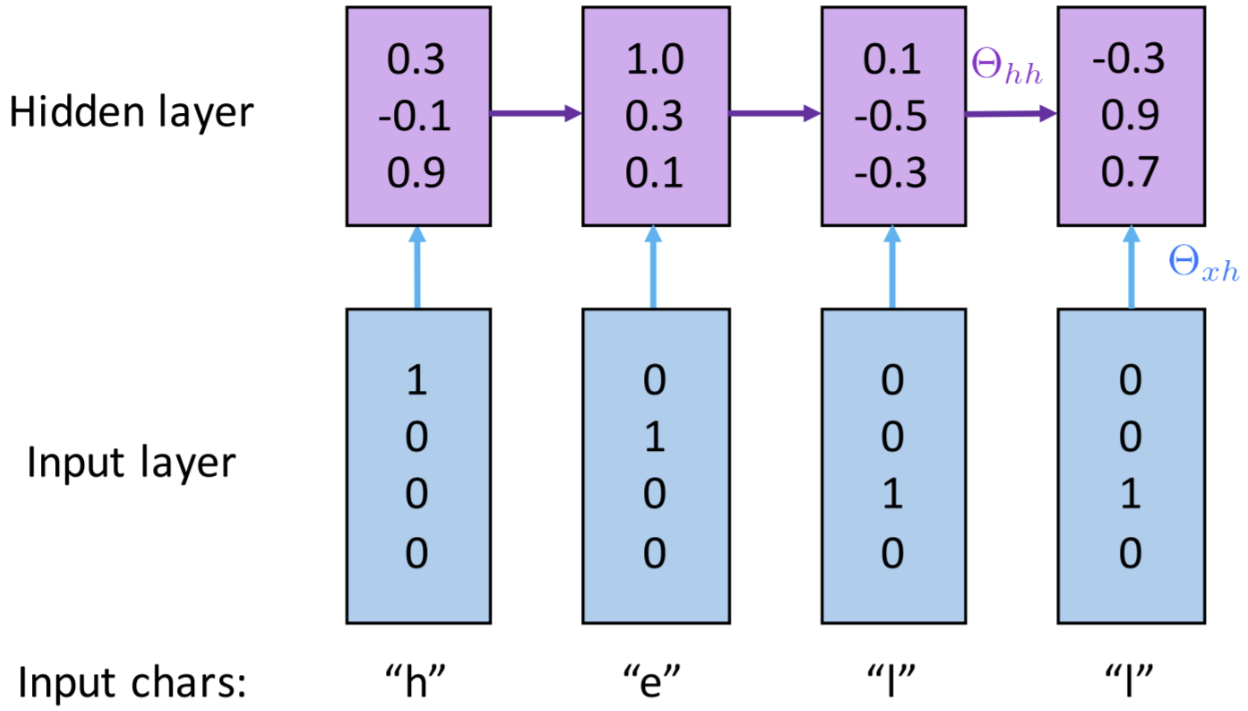
Example training sequence : "hello"

- Character-level language model
- Vocabulary : [h,e,l,o]
- Encoding into a vector using 1-of-k encoding and feed them into the RNN one at a time.

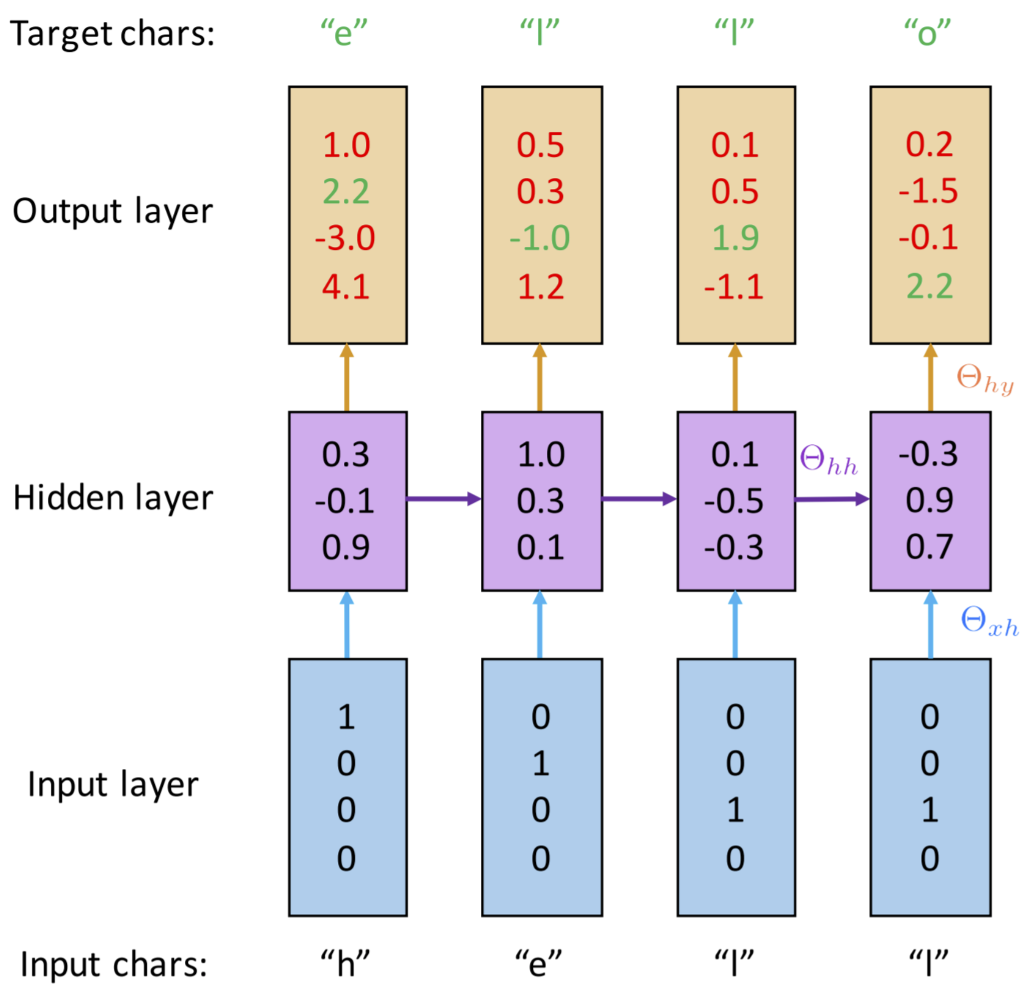


Character-level language model

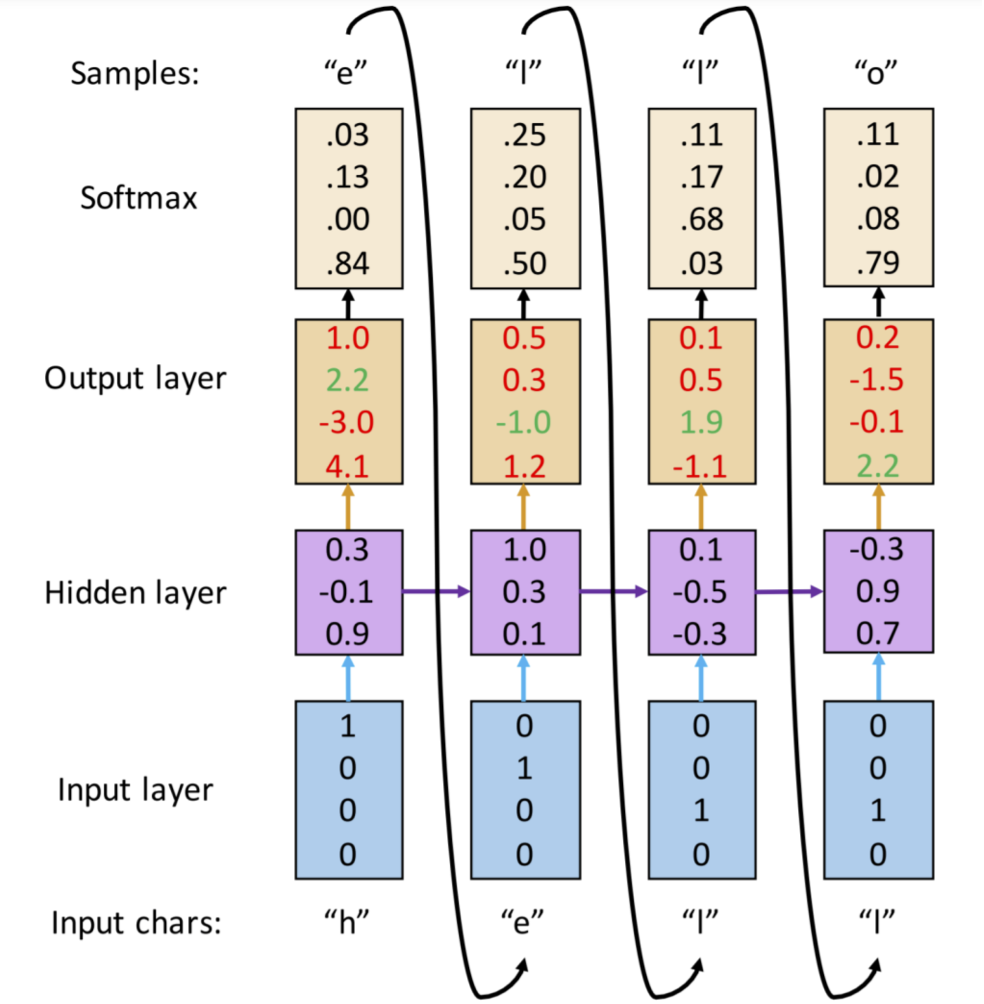
$$\mathbf{h}_t = \tanh(\Theta_{hh}\mathbf{h}_{t-1} + \Theta_{xh}\mathbf{x}_t)$$



For example, we see that in the first time step



a sequence of 4-dimensional output vectors

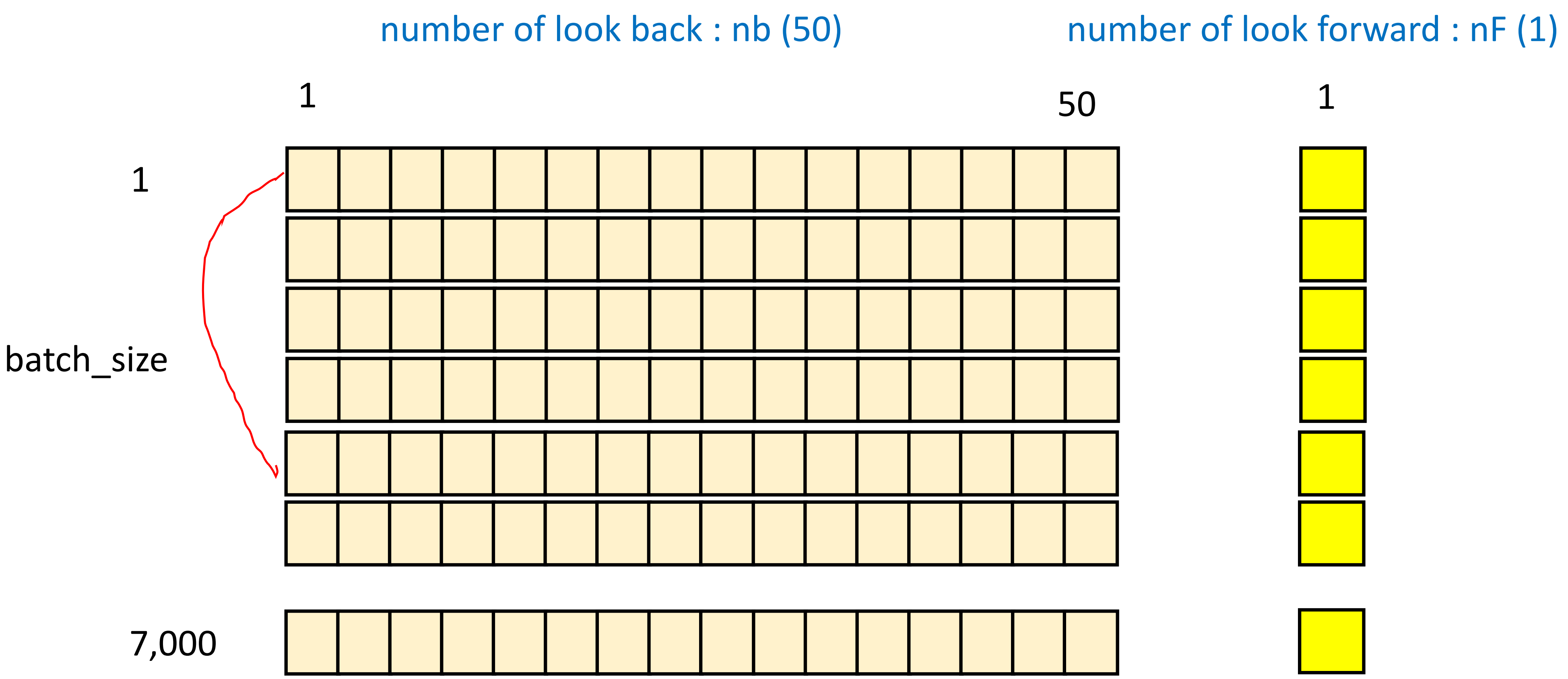


LAB01 : RNN-Basics

RNN, LSTM, GRU

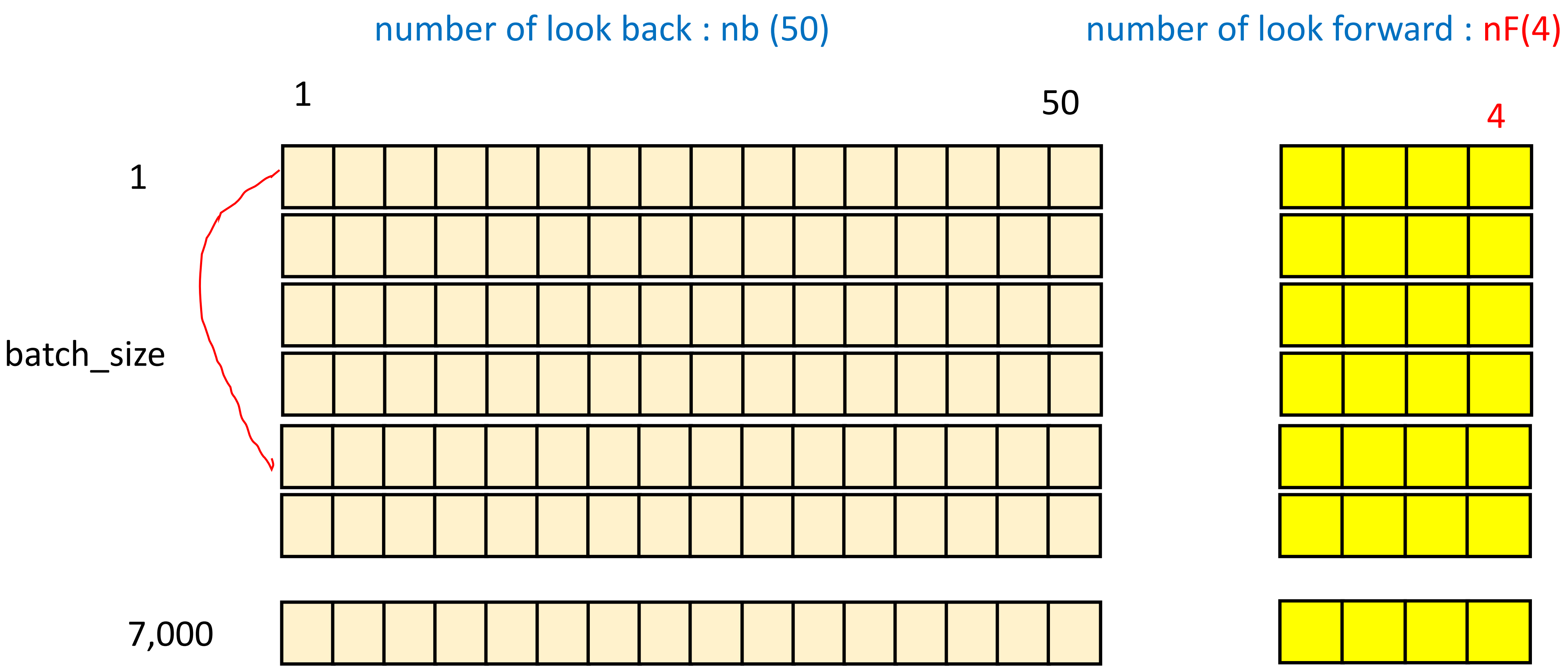
Many-to-One RNN Data Structure

X_train[7000,50,1] y_train[7000,1,1]

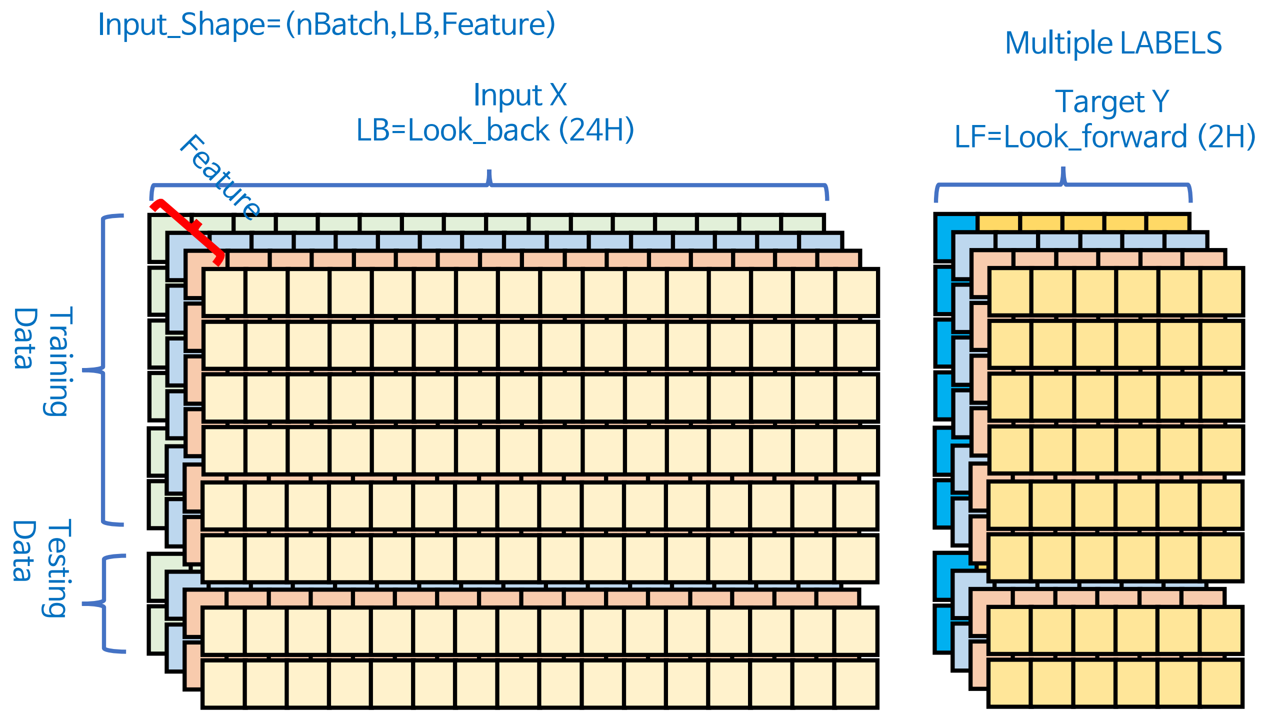


Many-to-Many RNN Data Structure

X_train[7000,50,1]   y_train[7000,nF,1]



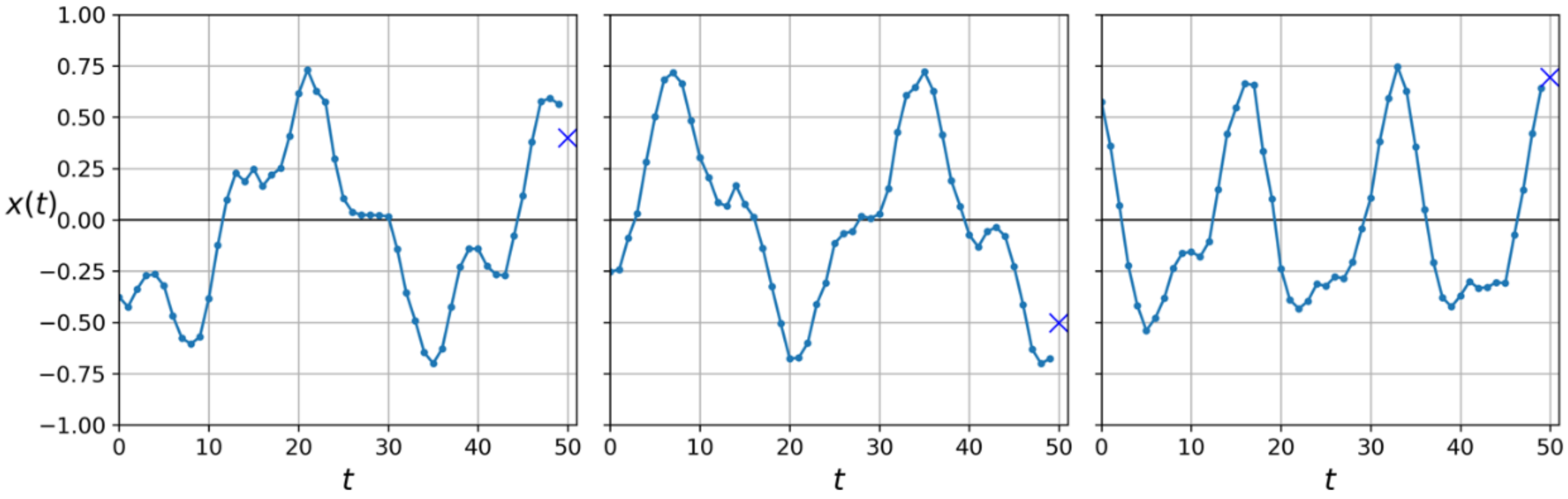
RNN Input-Output Data Structure



VDS, RSE Data Preprocessing for LSTM

Forecasting a Time Series

- There is a single value per time step : *univariate time series*
  - ✓ A typical task is to predict future values, which is called *forecasting*.
- For example, figures shows 3 univariate time series
  - ✓ each of them 50 time steps long, and the goal here is to forecast the value at the next time step (represented by the X) for each of them.



For simplicity, we are using a time series generated by the function

Generate the Dataset ¶

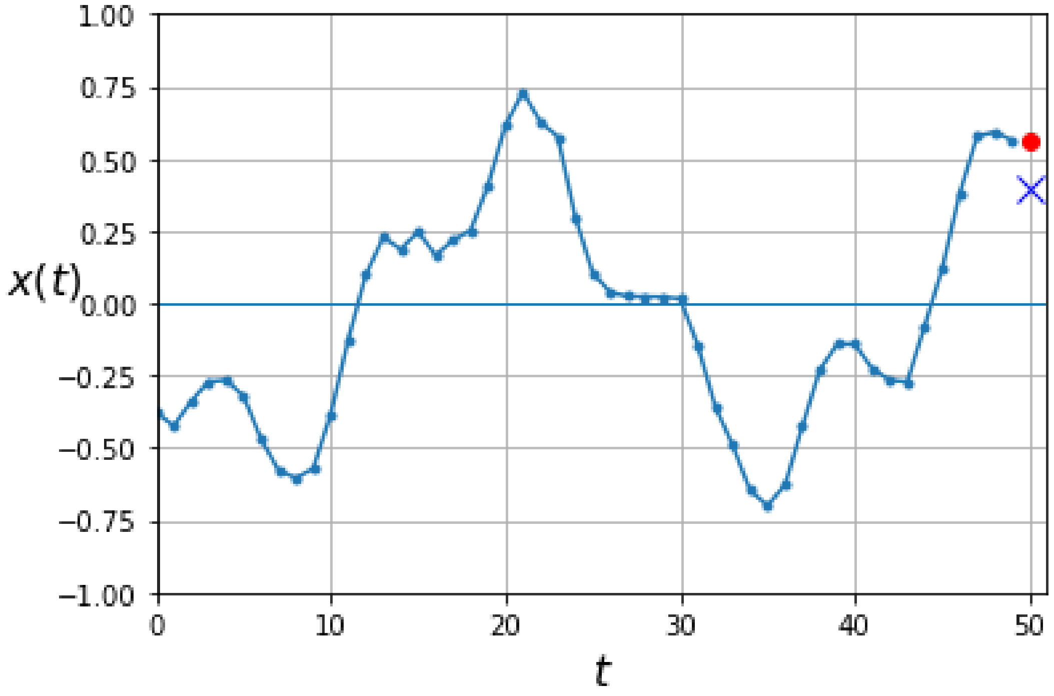
In [3]:
```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))  #   wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)   # + noise
    return series[..., np.newaxis].astype(np.float32)
```

The function returns a NumPy array of shape [*batch size*, *time steps*, 1],

where each series is the sum of two sine waves of fixed amplitudes but random frequencies and phases, plus a bit of noise.

Univariate time series

- The function returns a NumPy array of shape : [*batch size*, *time steps*, 1]
  - ✓ where each series is the sum of two sine waves of fixed amplitudes but random frequencies and phases, plus a bit of noise.



bx : for y_values at 51
ro : for y_prediction at 51

plot series

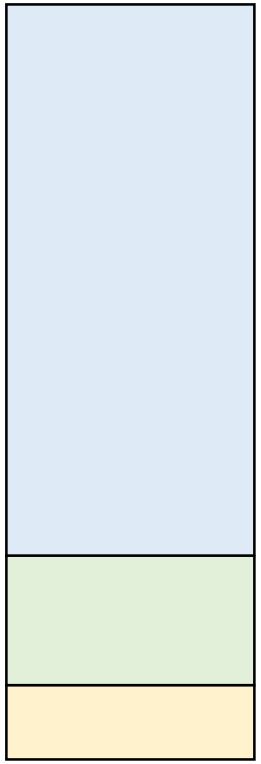
```python
def plot_series(series, y=None, y_pred=None, x_label="$t$", y_label="$x(t)$"):
    plt.plot(series, ".-")
    if y is not None:
        plt.plot(n_steps, y, "rx", markersize=10)
    if y_pred is not None:
        plt.plot(n_steps, y_pred, "ro")
    plt.grid(True)
    if x_label:
        plt.xlabel(x_label, fontsize=16)
    if y_label:
        plt.ylabel(y_label, fontsize=16, rotation=0)
    plt.hlines(0, 0, 100, linewidth=1)
    plt.axis([0, n_steps + 1, -1, 1])

fig, axes = plt.subplots(nrows=1, ncols=3, sharey=True, figsize=(12, 4))
for col in range(3):
    plt.sca(axes[col])
    plot_series(X_valid[col, :, 0], y_valid[col, 0],
                y_label=("$x(t)$" if col==0 else None))

plt.show()
```

Now let's create a training set, a validation set, and a test set

X_train: 0 ~ 6999

X_valid: 7000~8999

X_test: 9000~9999

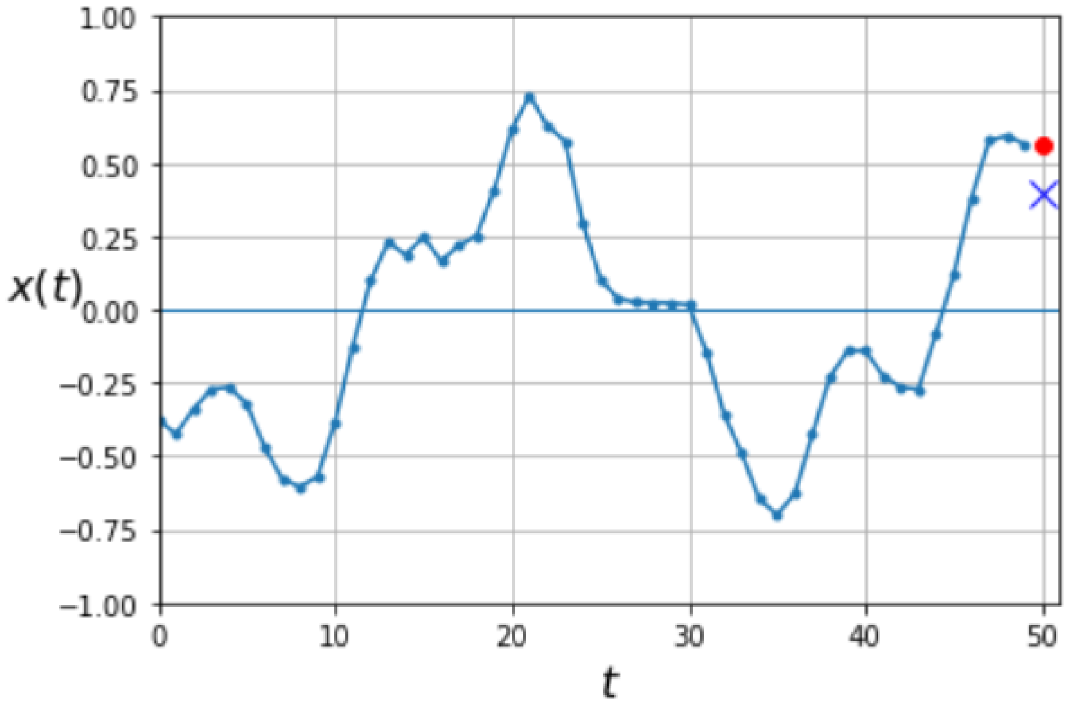


```
np.random.seed(42)

n_steps = 50

series = generate_time_series(10000, n_steps + 1)


X_train, y_train = series[:7000, :n_steps], series[:7000, -1]

X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]

X_test, y_test = series[9000:, :n_steps], series[9000:, -1]


X_train.shape, y_train.shape, X_valid.shape
```

```
((7000, 50, 1), (7000, 1), (2000, 50, 1))
```

model1: Computing Some Baselines

- Naive predictions (just predict the last observed value):
    - ✓ to predict the last value in each series
- In this case, it gives us a mean squared error of about 0.020211:




```
y_pred = X_valid[:, -1]
np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
```

model 2: Linear Predictions

- Another simple approach is to use a fully connected network.
  - ✓ Since it expects a flat list of features for each input, we need to add a Flattern layer.

```
m2 = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
```

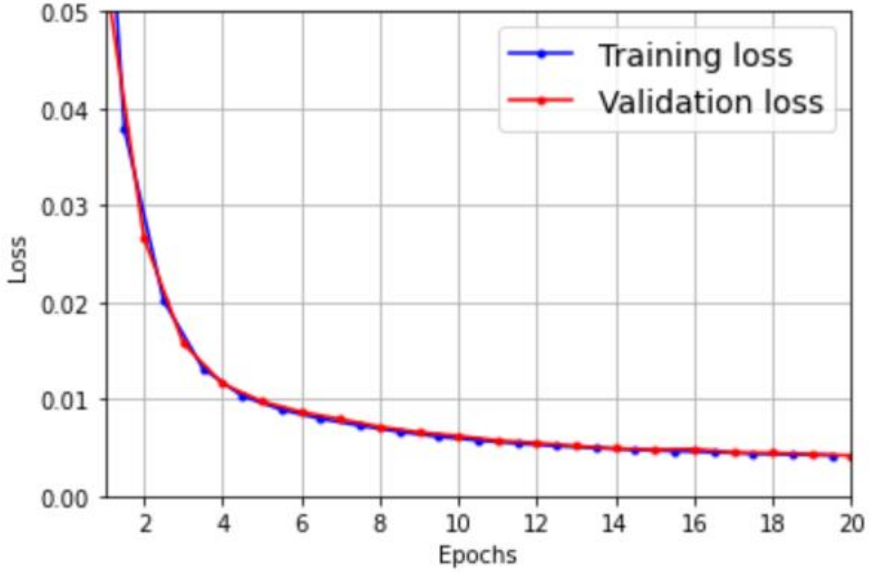
```
m2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 50)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 51
_________________________________________________________________
```

02. Fully connected network : Flatten

- Another simple approach is to use a fully connected network.
  - ✓ Since it expects a flat list of features for each input, we need to add a Flattern layer.
  - ✓ Let's just use a simple Linear Regression model so that each prediction will be a linear combination of the values in the time series:

```
m2 = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])

m2.compile(loss="mse", optimizer="adam")
history = m_lin.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
m2.evaluate(X_valid, y_valid)
```



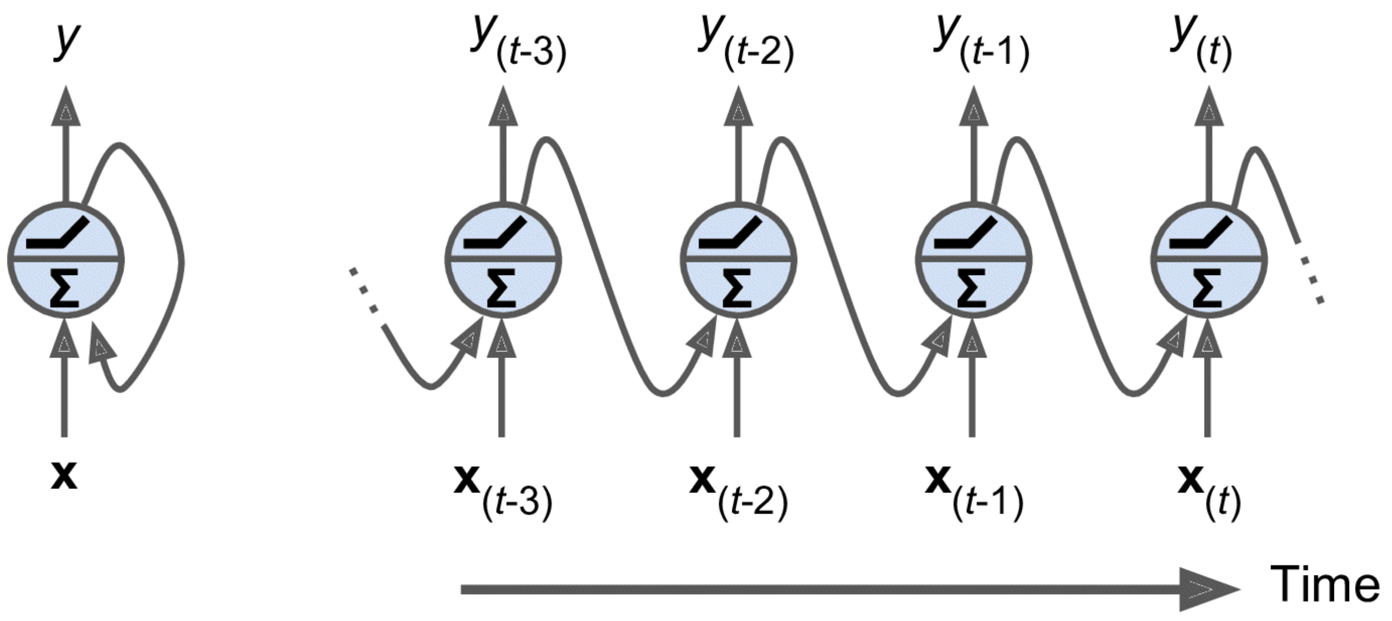

```
Out[9]: 0.004168087150901556
```

Learning Curves function and plot

```python
def plot_learning_curves(loss, val_loss):
    plt.plot(np.arange(len(loss)) + 0.5, loss, "b.-", label="Training loss")
    plt.plot(np.arange(len(val_loss)) + 1, val_loss, "r.-", label="Validation loss")
    plt.gca().xaxis.set_major_locator(mpl.ticker.MaxNLocator(integer=True))
    plt.axis([1, 20, 0, 0.05])
    plt.legend(fontsize=14)
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.grid(True)

plot_learning_curves(history.history["loss"], history.history["val_loss"])
plt.show()
```

03. Implementing a Simple RNN

- Simple RNN : don't need to specify the length of the RNN input sequence
  - ✓ The Simple RNN layer uses the hyperbolic tangent activity function. (-1 ~ 1)
  - ✓ It is set to the initial state h_(init)=0 and transmitted to the circulating neuron together with x(t=0), and then y(0) is output through the activation function.
  - ✓ This new h(0) becomes, and is transferred to the next input x(1) and input.
  - ✓ The last layer will be y (49).



03. Simple RNN

```
m3 = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])

optimizer = keras.optimizers.Adam(lr=0.005)
m3.compile(loss="mse", optimizer=optimizer)
history = m3.fit(X_train, y_train, epochs=20,
                 validation_data=(X_valid, y_valid))
```

We do not need to specify the length of the input sequences (unlike in the previous model), since a recurrent neural network can process any number of time steps

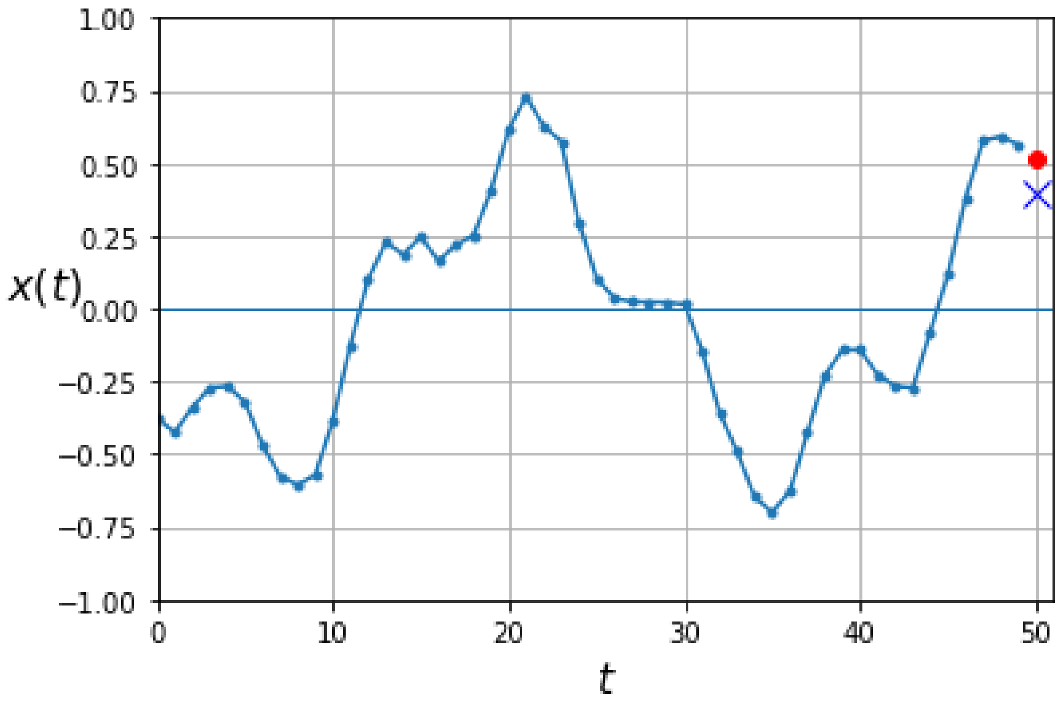
```
m3.summary()
```

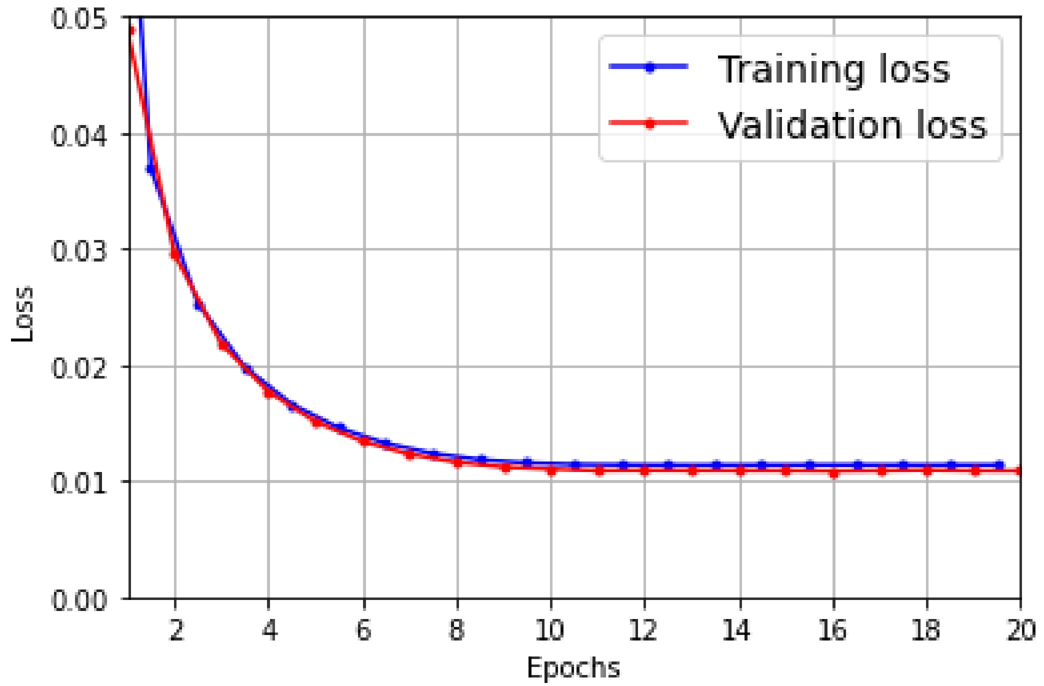
```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 1)                 3
=================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
_________________________________________________________________
```

SimpleRNN layer uses the hyperbolic tangent activation function

```
Out[31]: 0.010881561785936356
```

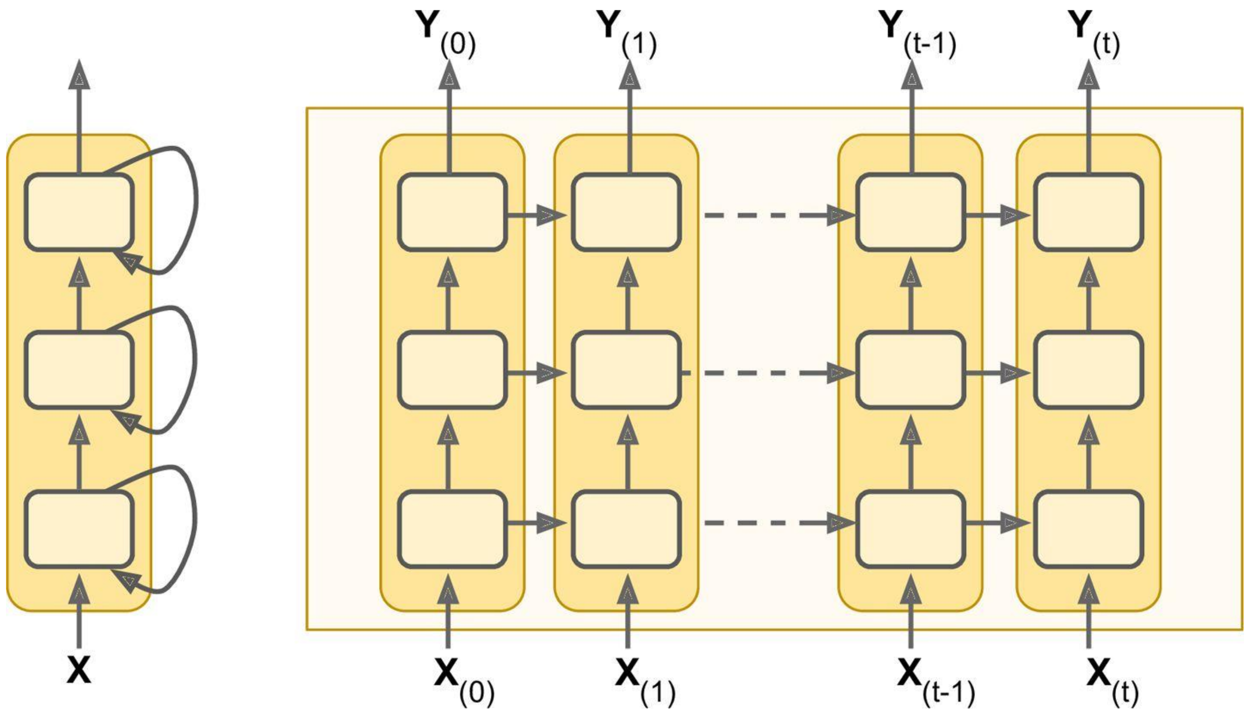
03. Simple RNN





04. Deep RNN (A)

- In Keras, the circulation layer outputs only the final output.
  - ✓ To return the output for each time step, Set setrun_sequences=True



Make sure return_sequence=True for all recurrent layers (except the last one, if you only care about the last output). If you don't, they will output a 2D array (containing only the output of the last time step) instead of a 3D array (containing outputs for all time steps), and the next recurrent layer will complain that you are not feeding it sequences in the expected 3D format.

04. Deep RNN (A)

```
m4 = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_2 (SimpleRNN)     (None, None, 20)          440
_________________________________________________________________
simple_rnn_3 (SimpleRNN)     (None, None, 20)          820
_________________________________________________________________
simple_rnn_4 (SimpleRNN)     (None, 1)                 22
=================================================================
Total params: 1,282
Trainable params: 1,282
Non-trainable params: 0
_________________________________________________________________
```

04. Deep RNN (A)

```
m4.evaluate(X_valid, y_valid)
```

```
63/63 [==============================] - 1s 15ms/step - loss: 0.0029

0.0029105639550834894
```

```
m4 = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```

05. Deep RNN with only single output (unit)

- remove return_sequences = True
  - ✓ it must have a single unit because we want to forecast a univariate time series, and this means we must have a single output value per time step.
  - ✓ However, having a single unit means that the hidden state is just a single number.
  - ✓ That's really not much, and it's probably not that useful; presumably, the RNN will mostly use the hidden states of the other recurrent layers to carry over all the information it needs from time step to time step, and it will not use the final layer's hidden state very much
- Use Dense layer
  - ✓ Simple RNN use tanh(x) activation function (from -1 to 1)
  - ✓ it might be preferable to replace the output layer with a Dense layer

05. Deep RNN with only single output (unit)

```
m5 = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])
```

```
m5.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_7 (SimpleRNN)     (None, None, 20)          440
_________________________________________________________________
simple_rnn_8 (SimpleRNN)     (None, 20)                820
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 21
=================================================================
Total params: 1,281
Trainable params: 1,281
Non-trainable params: 0
_________________________________________________________________
```

05. Deep RNN with only single output (unit)

- If you train this model, you will see that it converges faster and performs just as well. Plus, you could change the output activation function if you wanted.

```
Out[45]: 0.002527124481275677
```