





RNN for NLP

RNN, LSTM, GRU

UST AI Lecture : RNN for NLP

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

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

	hongsukyi Add files via upload	
	lab01-rnn-nlp-hw.ipynb	Add files via upload
	lec01-rnn-introduction.zip	Add files via upload
	lec02-rnn-text-generation.zip	Add files via upload
	nlp01-Eng-rnn-intro(1109,v1).pdf	Add files via upload

Contents

- Recurrent Neural Network
 - ✓ Practice on RNN and LSTM
- Text generation with RNN
 - ✓ Practice on Char-RNN Language Model
- Text classification with RNN
 - ✓ Practice on IMDB Dataset using RNN
- Text generation with RNN
 - ✓ Practice on Text Generation model with RNN

A simple Example

“Modeling word probabilities is really difficult”

Modeling $p(\mathbf{x})$

Simplest model:

Assume independence of words

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

$p(\text{"modeling"}) \times p(\text{"word"}) \times p(\text{"probabilities"}) \times p(\text{"is"}) \times p(\text{"really"}) \times p(\text{"difficult"})$

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

Modeling $p(x)$

More realistic model:

Assume conditional dependence of words

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

Modeling word probabilities is really ?

Context

Target

$p(x|\text{context})$

difficult

0.01

hard

0.009

fun

0.005

...

...

easy

0.00001

Modeling $p(\mathbf{x})$

The chain rule

Computing the joint $p(\mathbf{x})$ from conditionals

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

Modeling

Modeling **word**

Modeling word **probabilities**

Modeling word probabilities **is**

Modeling word probabilities is **really**

Modeling word probabilities is really **difficult**

$$p(x_1)$$

$$p(x_2 | x_1)$$

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

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

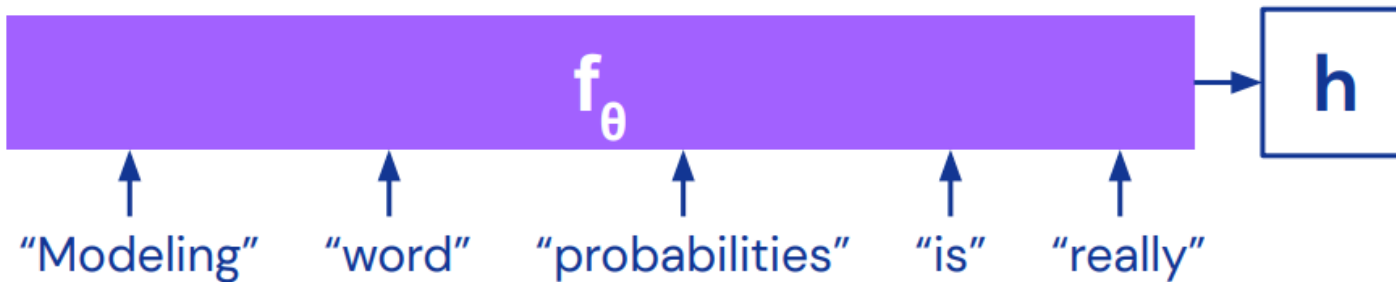
$$p(x_5 | x_4, x_3, x_2, x_1)$$

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

Recurrent Neural Networks (RNNs)

- Learning to model word probabilities

- ✓ Vectorising the context



f_θ summarises the context in h such that:

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

Desirable properties for f_θ :

- Order matters
- Variable length
- Learnable (differentiable)

Recurrent Neural Networks (RNNs)

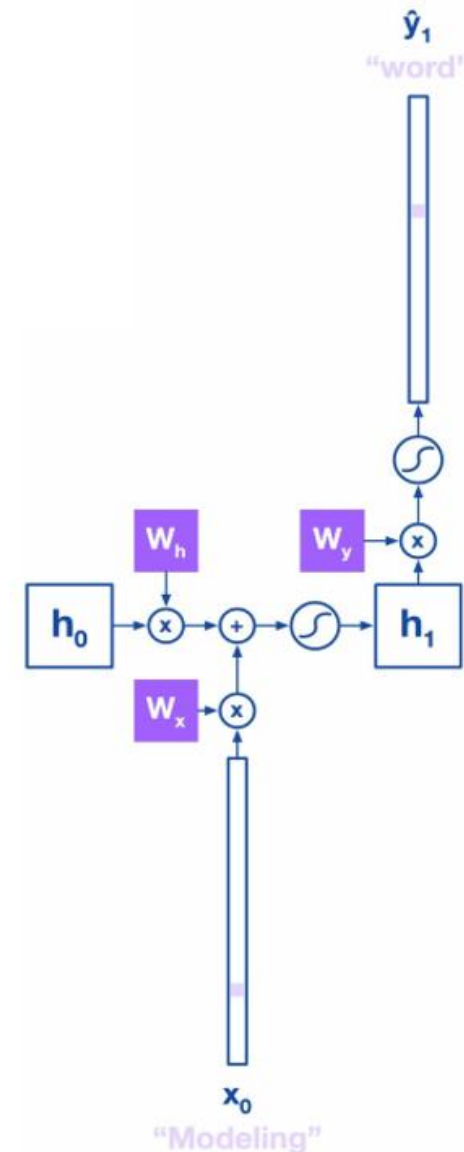
- Persistent state variable \mathbf{h} stores information from the context observed so far.

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

RNNs predict the target \mathbf{y} (the next word) from the state \mathbf{h} .

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

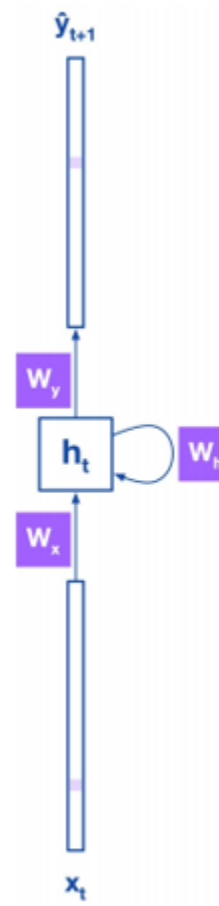
Softmax ensures we obtain a distribution over all possible words.



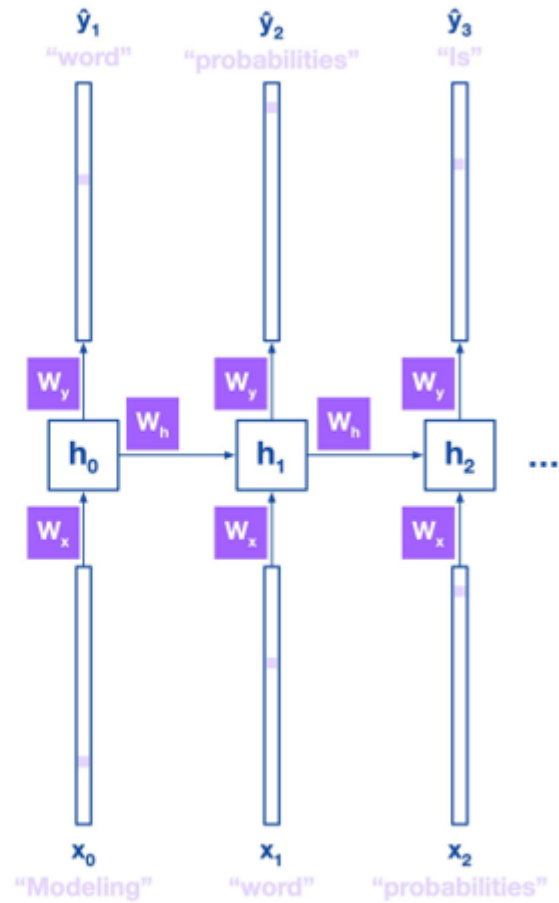
Recurrent Neural Networks (RNNs)

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:

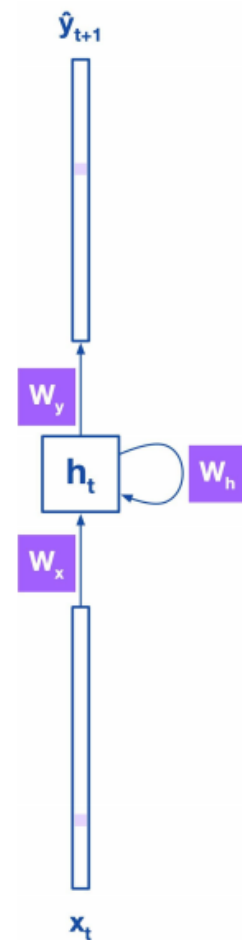
For one word:

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

For the
sentence:

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

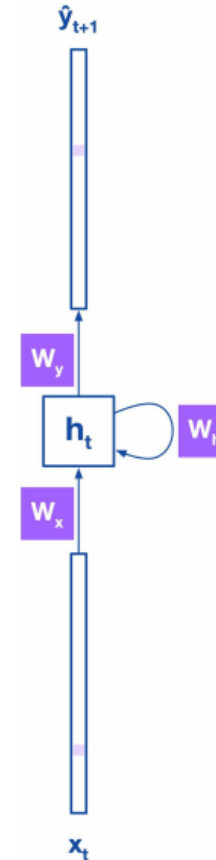
With parameters $\theta = \{\mathbf{W}_y, \mathbf{W}_x, \mathbf{W}_h\}$



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

$$\begin{aligned}\mathbf{h}_t &= \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t) \\ p(\mathbf{x}_{t+1}) &= \text{softmax}(\mathbf{W}_y \mathbf{h}_t) \\ \mathcal{L}_\theta(\mathbf{y}, \hat{\mathbf{y}})_t &= -\mathbf{y}_t \log \hat{\mathbf{y}}_t\end{aligned}$$

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



Character-Level Language Models

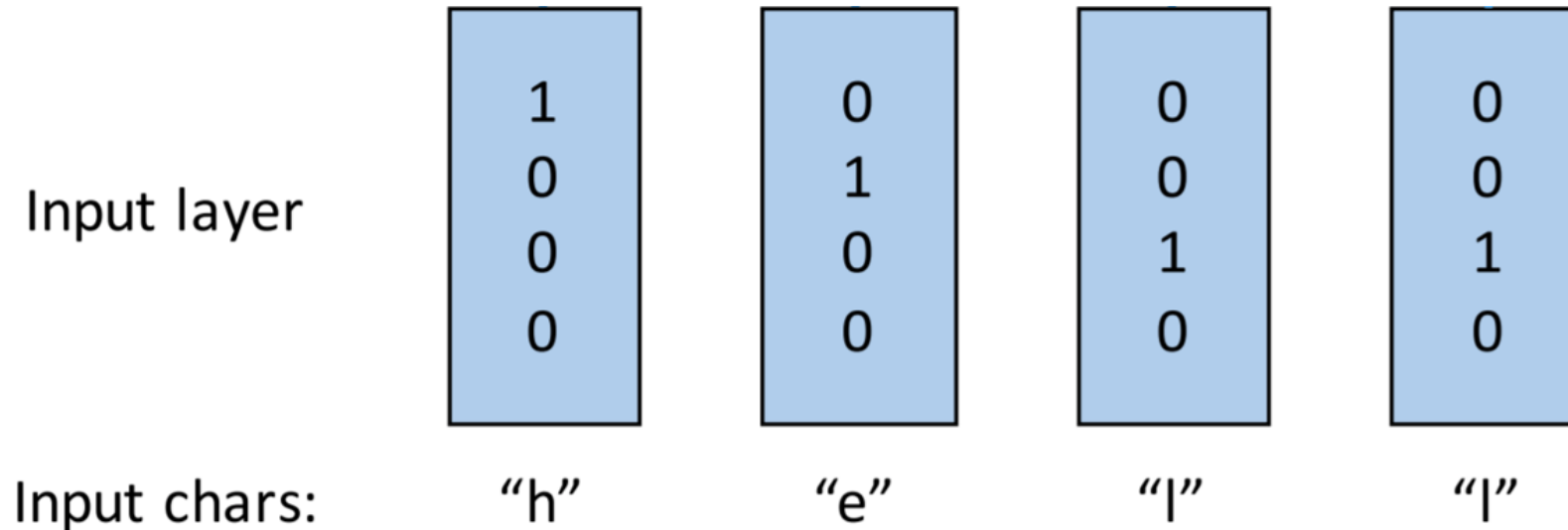
- We'll give the RNN a huge chunk of text and ask it to model the probability distribution of the next character in the sequence given a sequence of previous characters.
 - ✓ This will then allow us to generate new text one character at a time.

Character-level language model

Example training sequence : "hello"

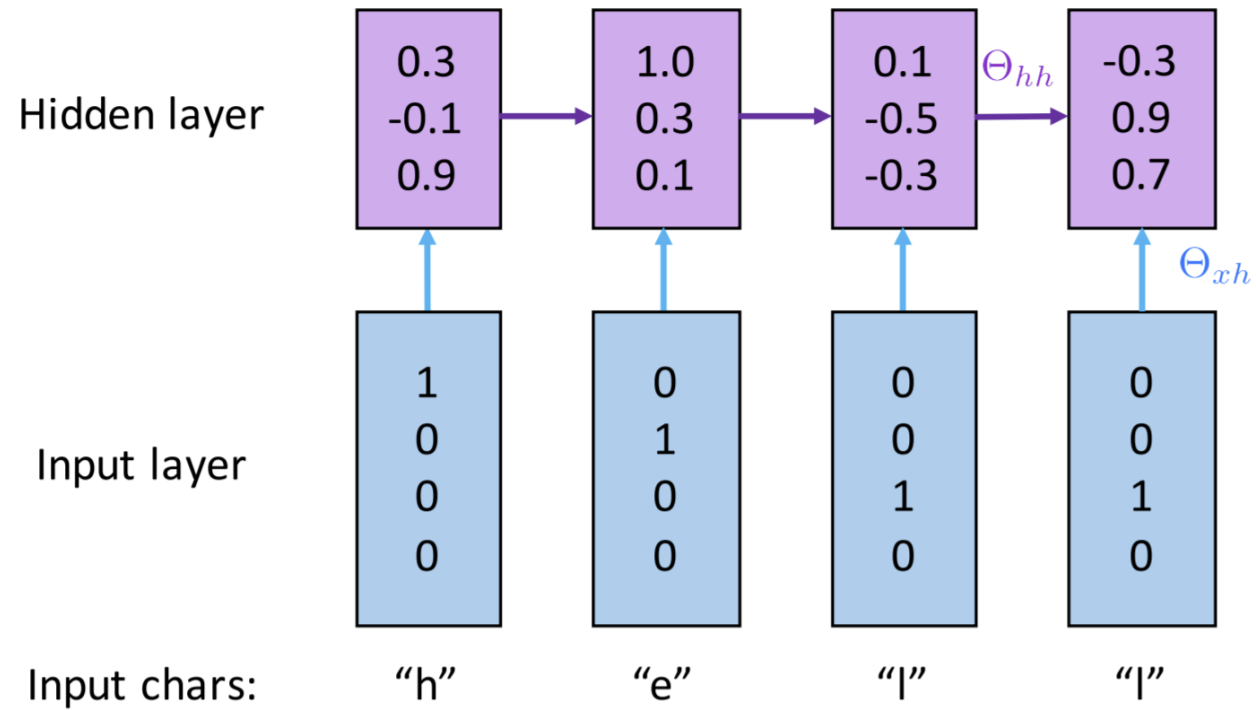
- Character-level language model

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

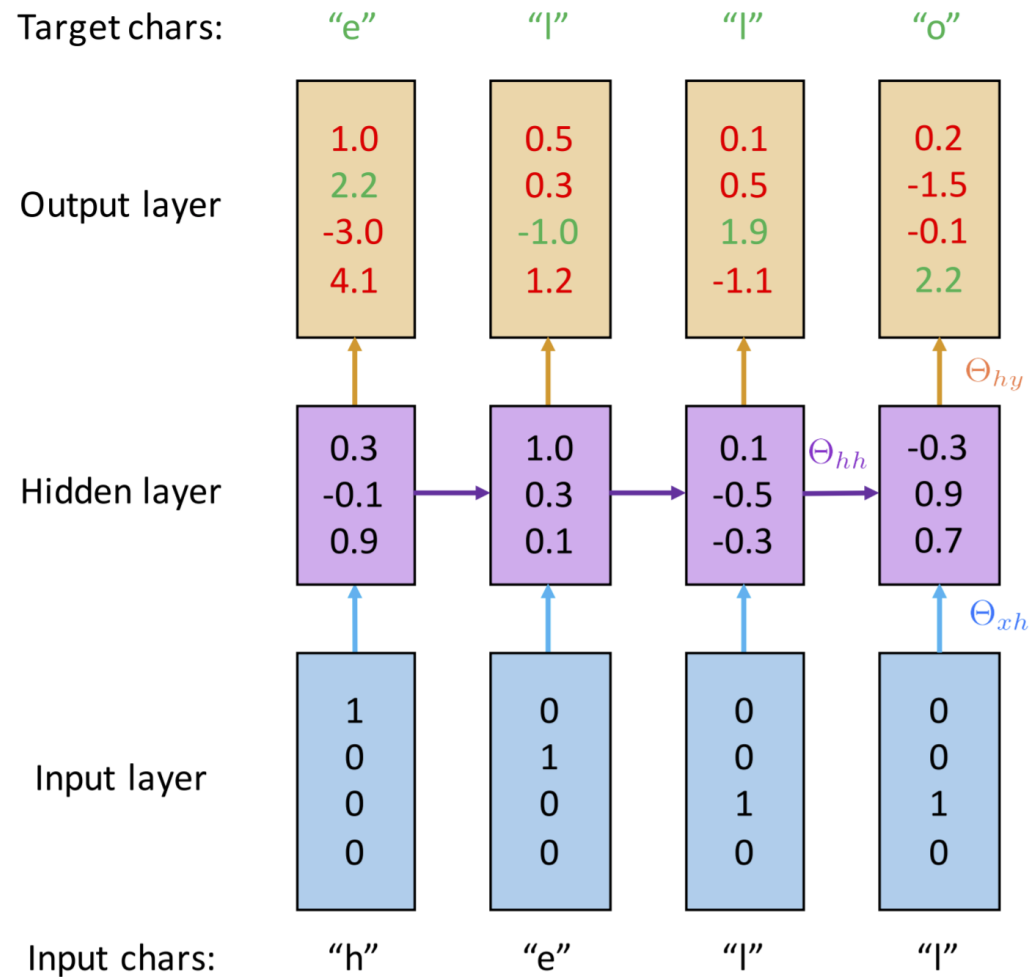


Character-level language model

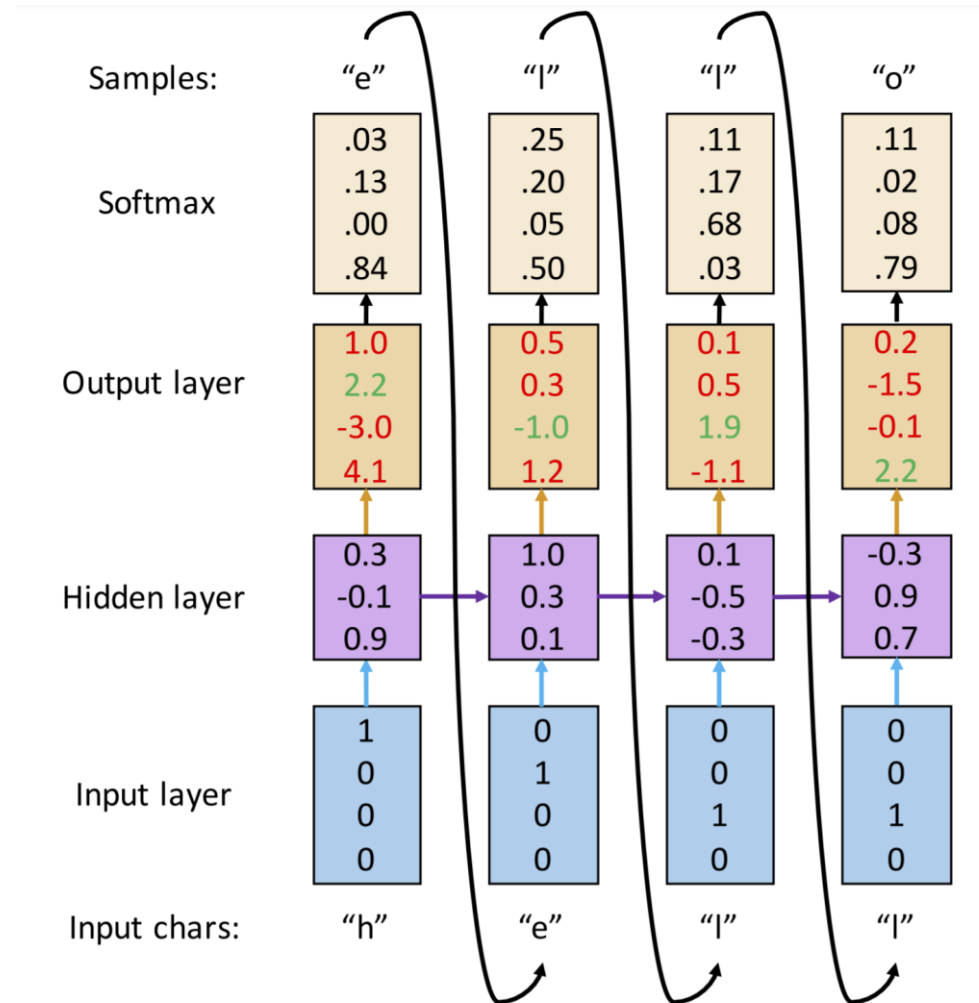
$$\mathbf{h}_t = \tanh(\Theta_{hh}\mathbf{h}_{t-1} + \Theta_{xh}\mathbf{x}_t)$$



For example, we see that in the first time step



a sequence of 4-dimensional output vectors



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	$\theta^* = \arg \min_{\theta} \mathcal{L}(\theta)$	$\theta^* = \arg \max_{\theta} \mathcal{L}(\theta)$

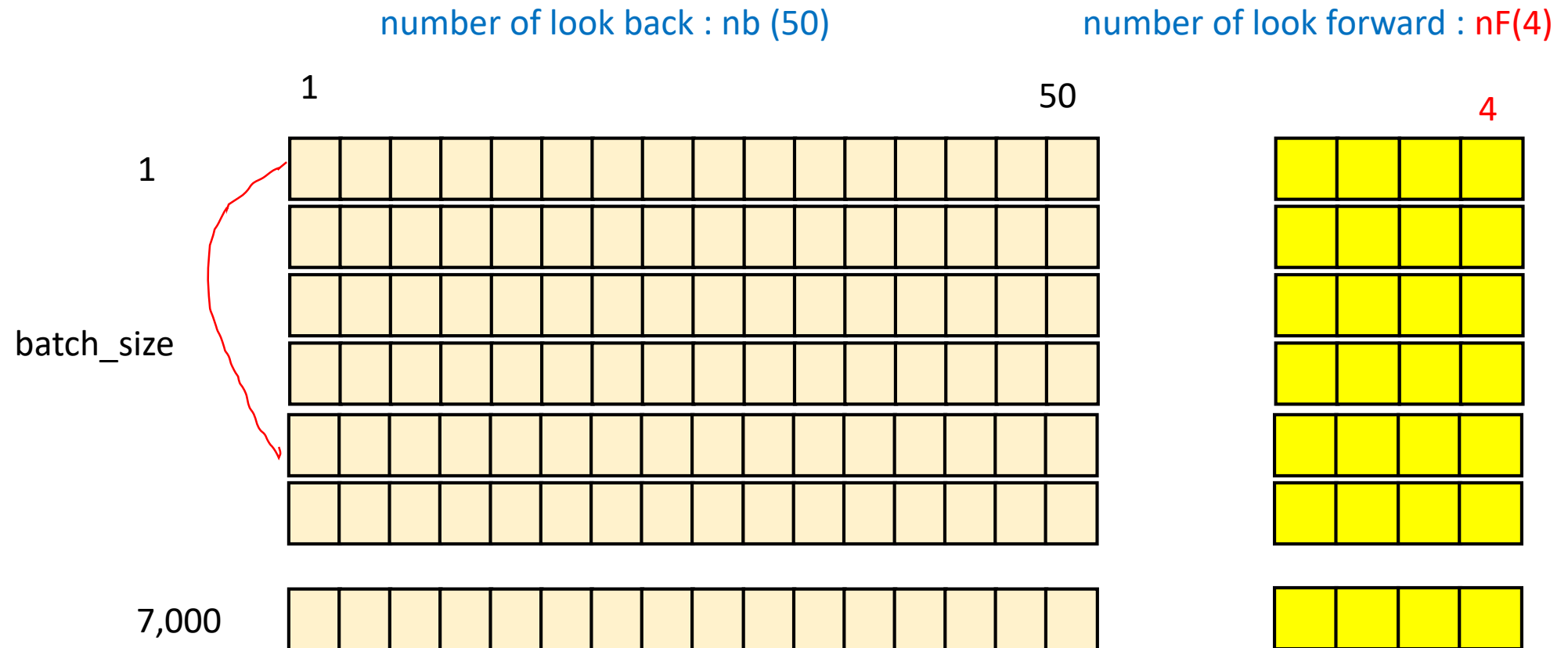
LAB01 : RNN-Basics

RNN, LSTM, GRU

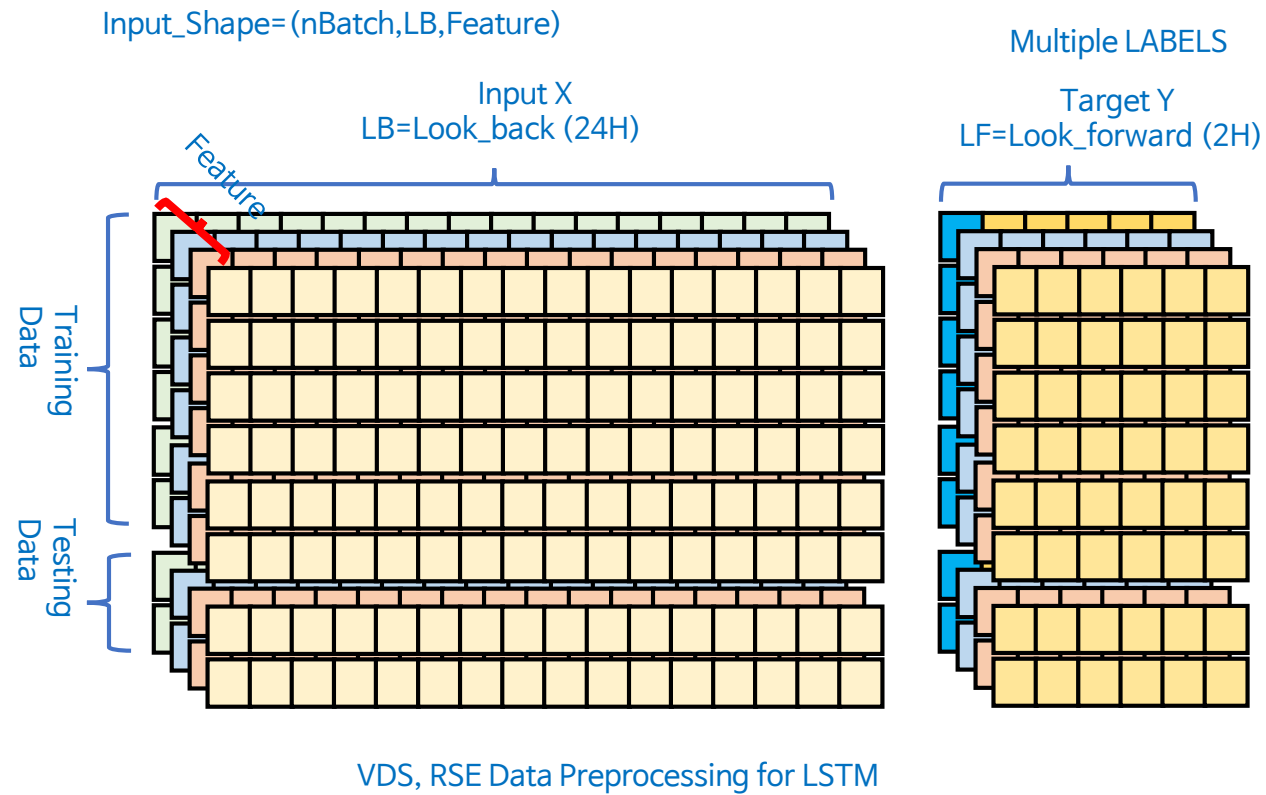
```
X_train[7000,50,1]    y_train[7000,1,1]
```

Many-to-Many RNN Data Structure

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



RNN Input-Output Data Structure



Forecasting a Time Series

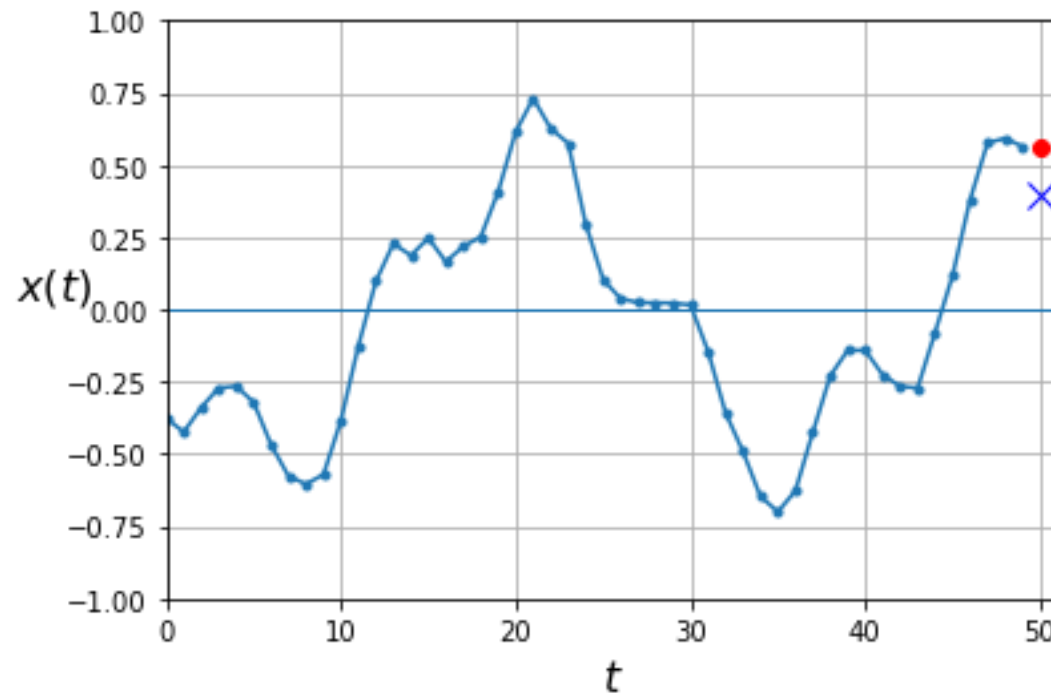
- There is a single value per time step : *univariate time series*
 - ✓ A typical task is to predict future values, which is called *forecasting*.
- For example, figures shows 3 univariate time series
 - ✓ each of them 50 time steps long, and the goal here is to forecast the value at the next time step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

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)
```


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

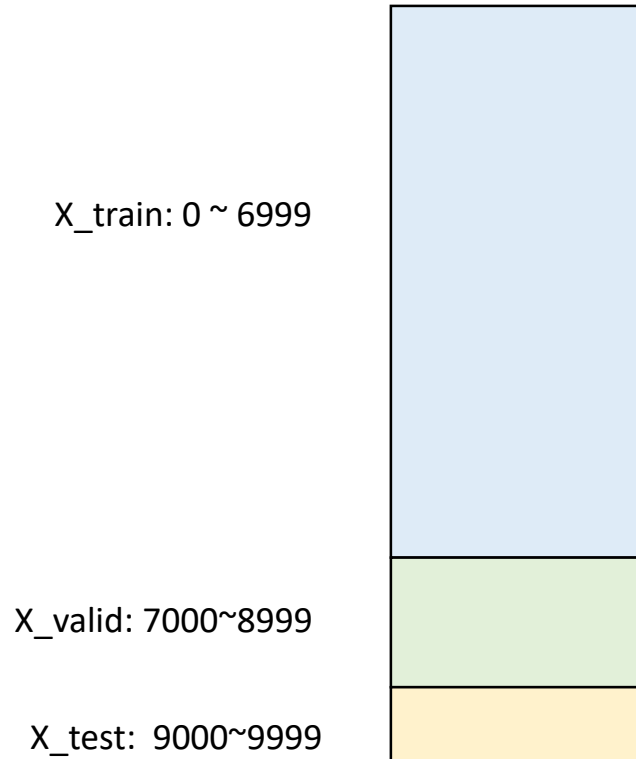
plot series

```
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



```
np.random.seed(42)

n_steps = 50

series = generate_time_series(10000, n_steps + 1)

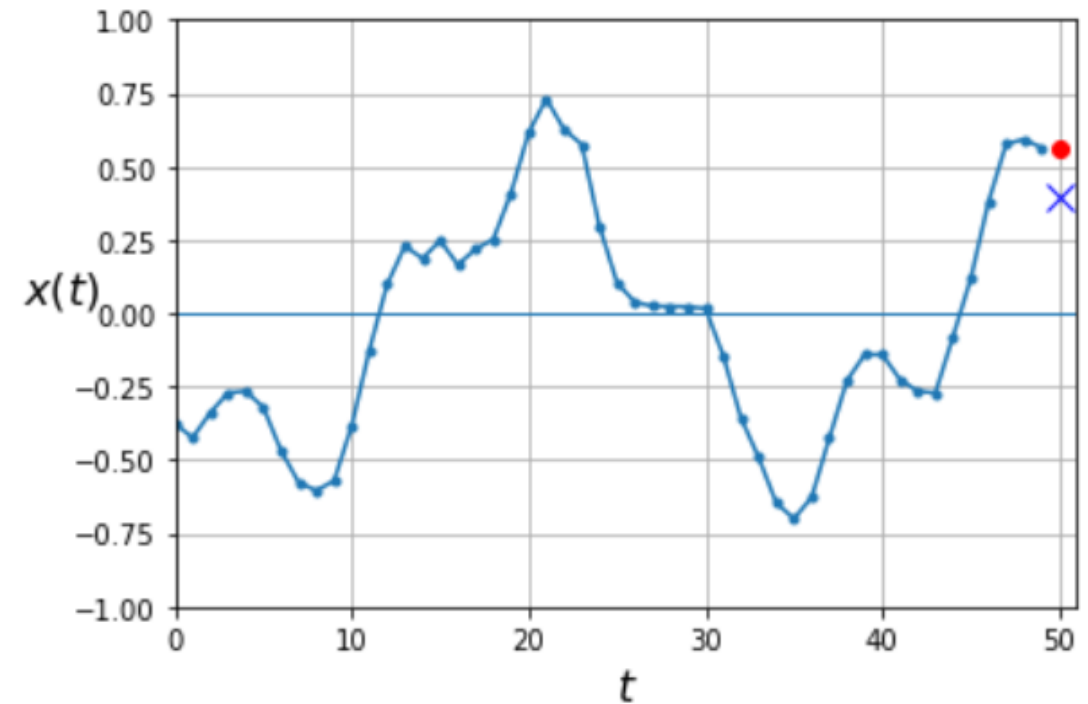
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]

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 Flatten 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

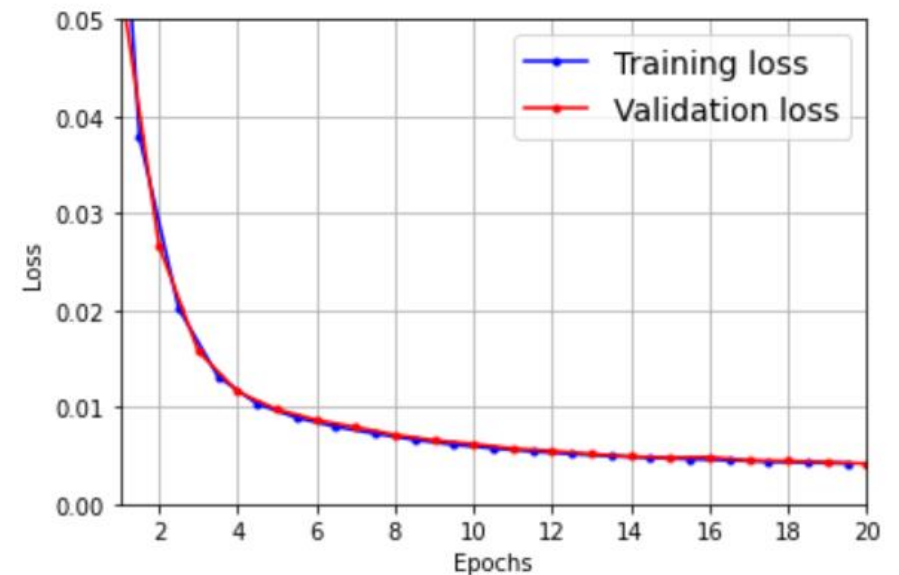
02. Fully connected network : Flatten

- 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 Flatten 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 = m2.fit(X_train, y_train, epochs=20,
                 validation_data=(X_valid, y_valid))
m2.evaluate(X_valid, y_valid)
```

Out [9]: 0.004168087150901556



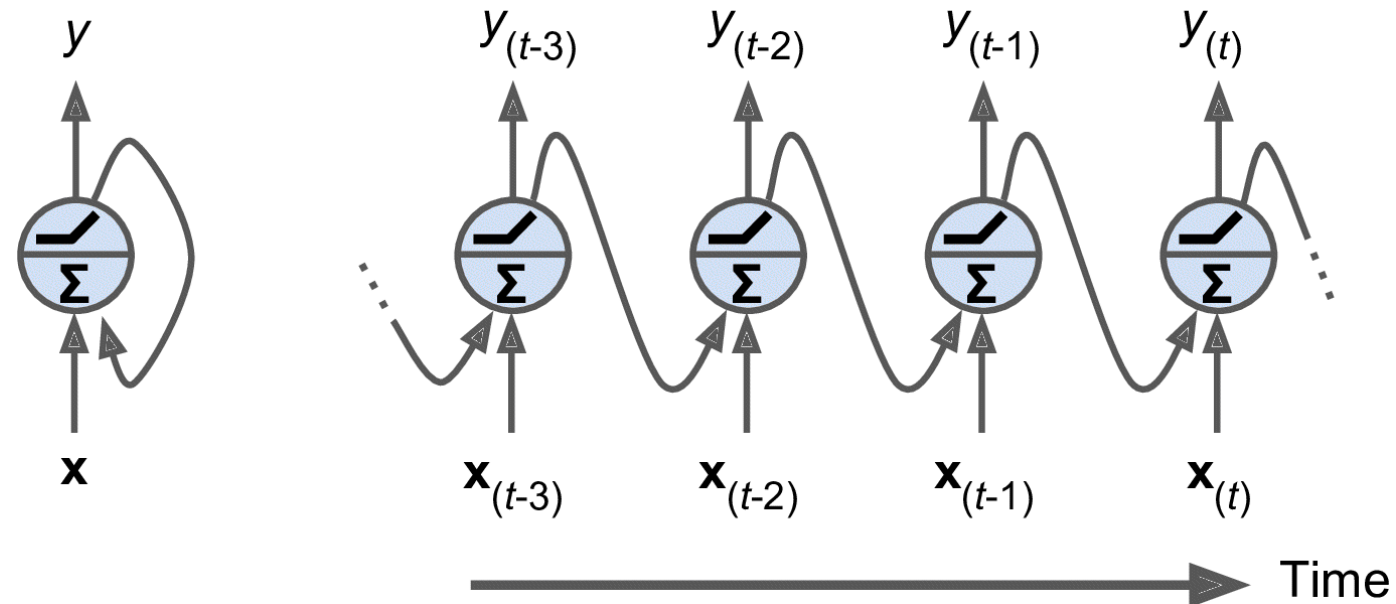
Learning Curves function and plot

```
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. $(-1 \sim 1)$
 - ✓ It is set to the initial state $h_{\text{init}}=0$ 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))
```

```
m3.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 1)	3

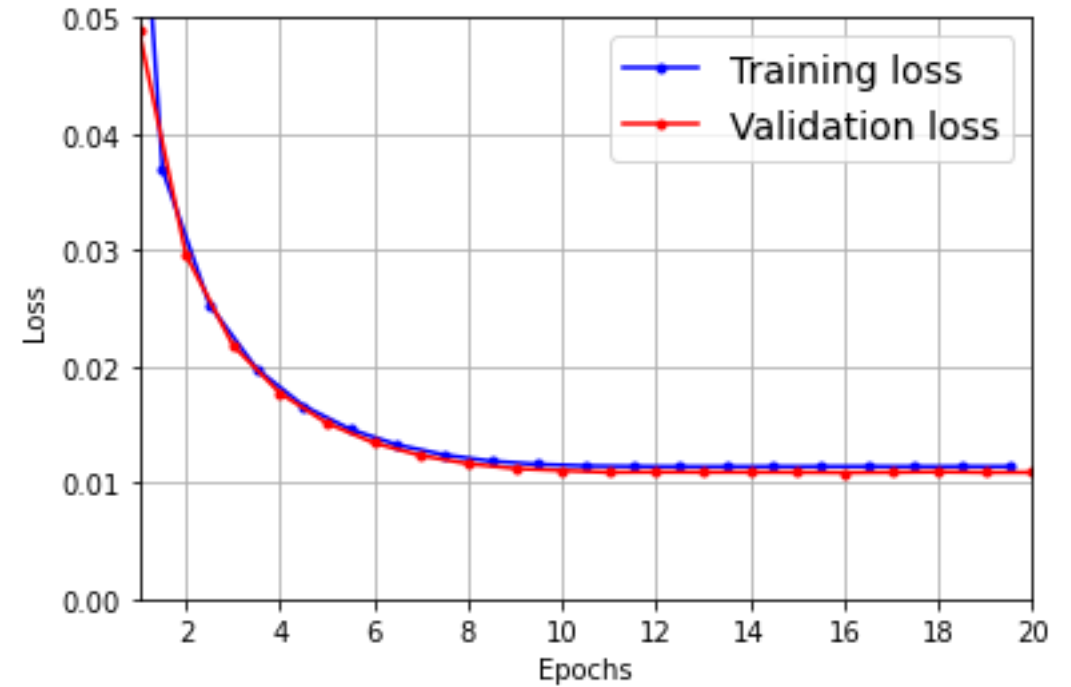
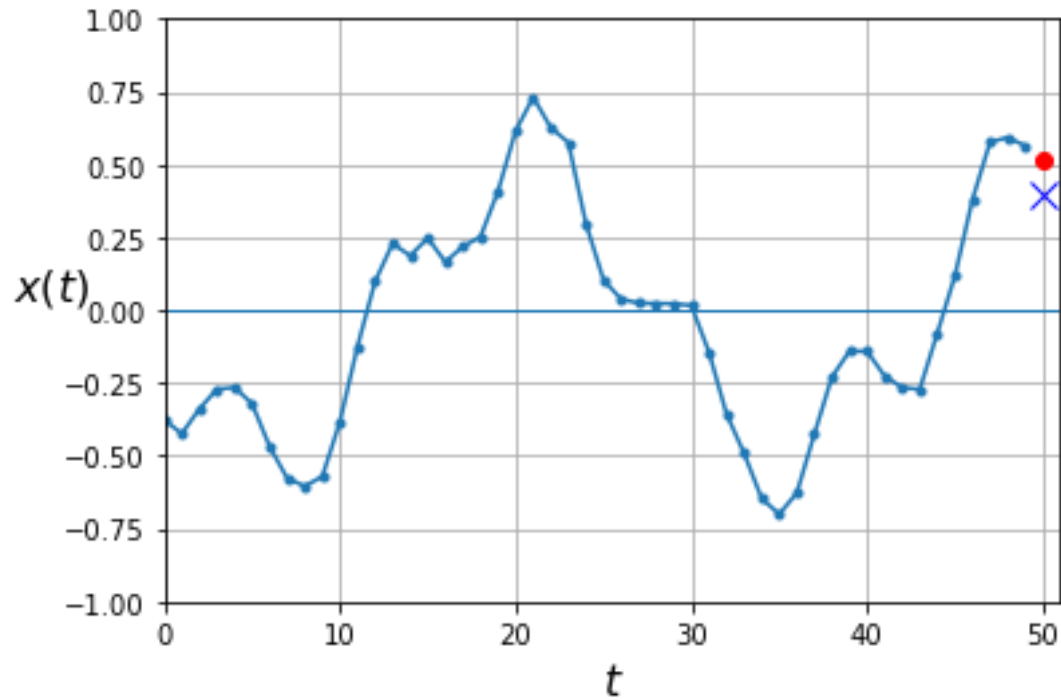
Total params: 3
Trainable params: 3
Non-trainable params: 0

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

SimpleRNN layer uses the hyperbolic tangent activation function

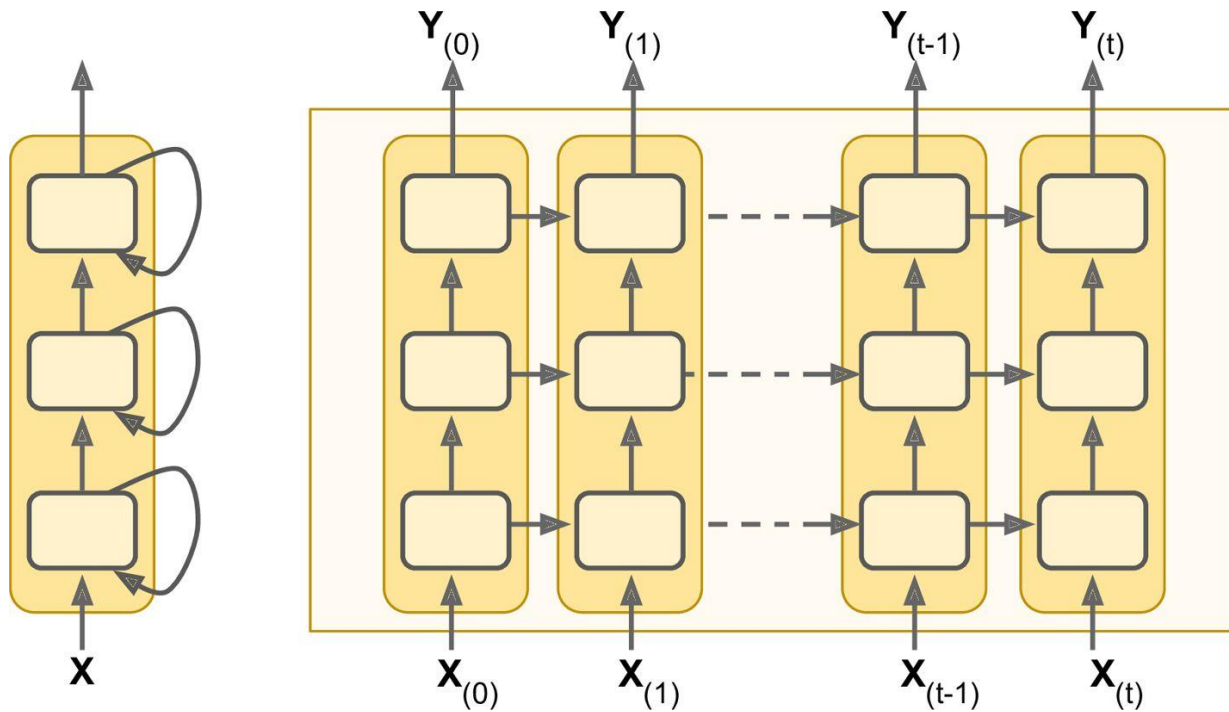
Out [31]: 0.010881561785936356

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 `return_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

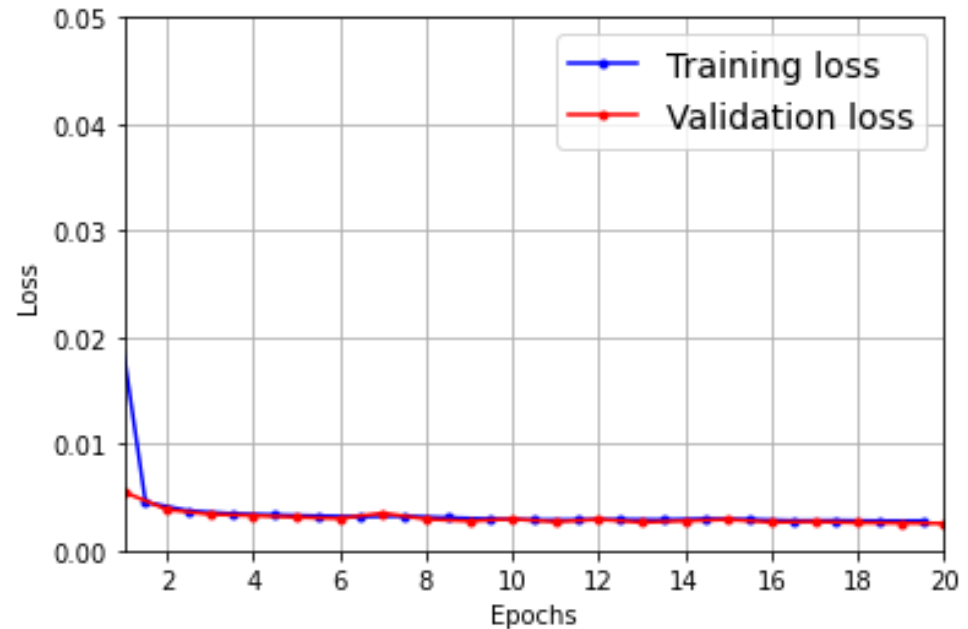
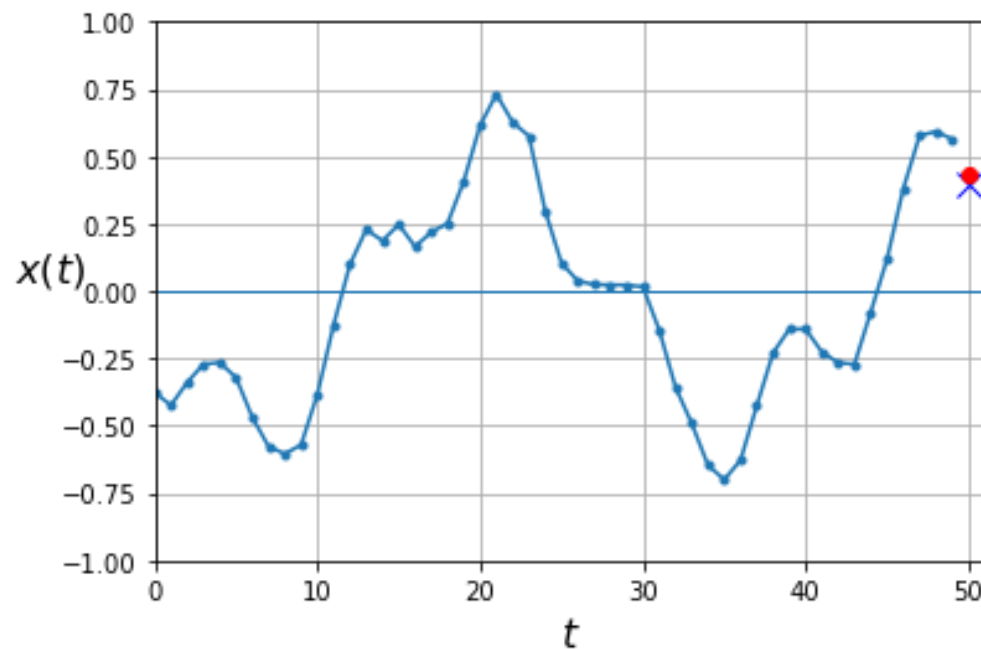
Trainable params: 1,281

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] : 0.0025271244812756777
```



Results Summary of many-to-one

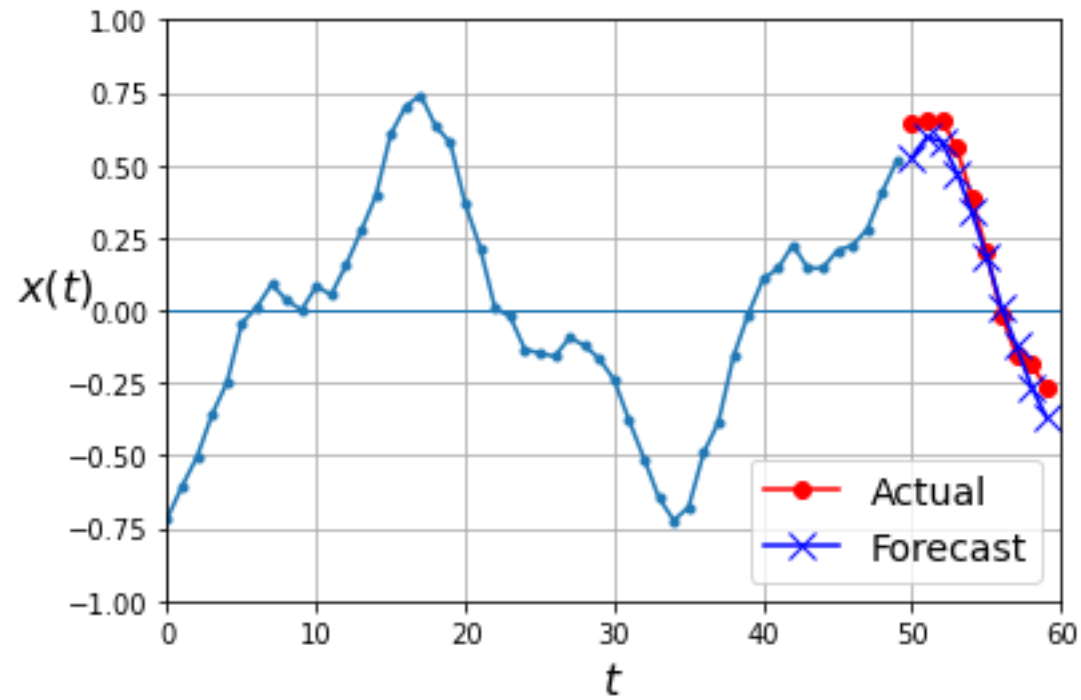
Results Summary of many-to-one ¶

```
In [47]: models = pd.DataFrame({  
        'Model': ['Flatten', 'SimpleRNN', 'DeepRNN', 'DeepRNNDense'],  
        'Score': [a1, a2, a3, a4]})  
models.sort_values(by='Score', ascending=True)
```

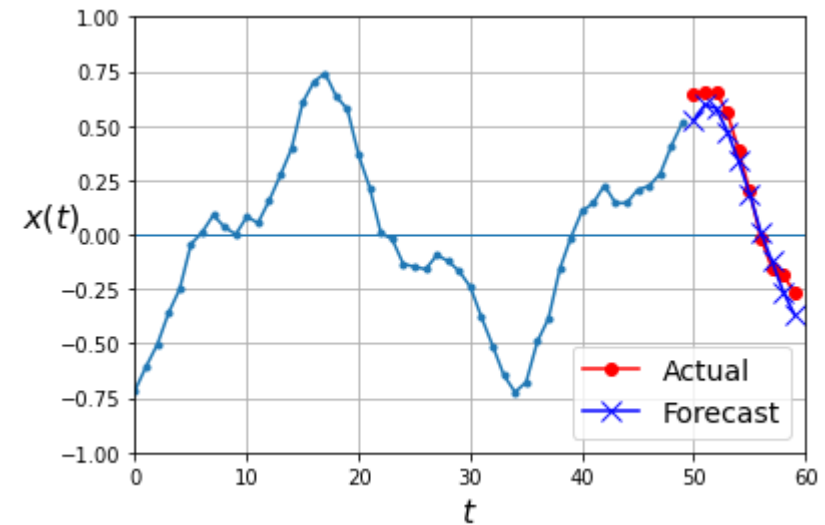
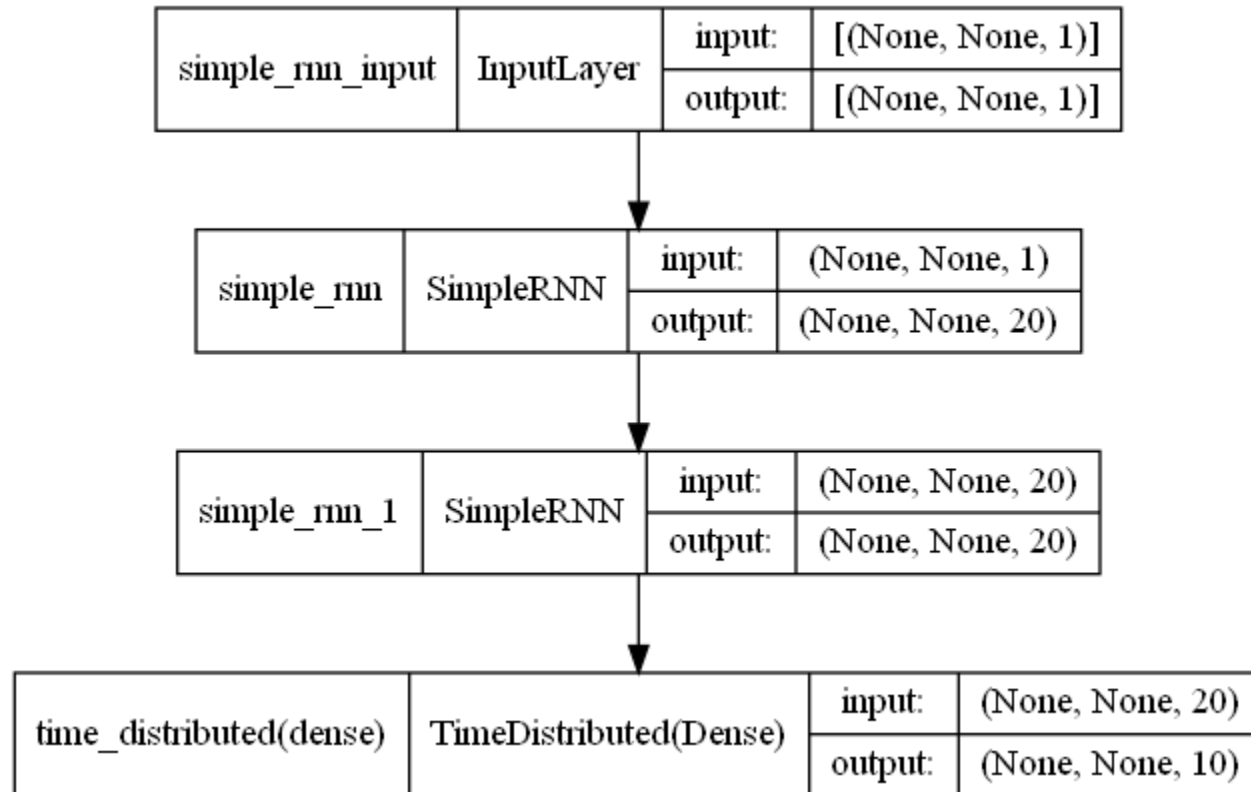
Out[47]:

	Model	Score
3	DeepRNNDense	0.003076
2	DeepRNN	0.003980
0	Flatten	0.010993
1	SimpleRNN	0.018042

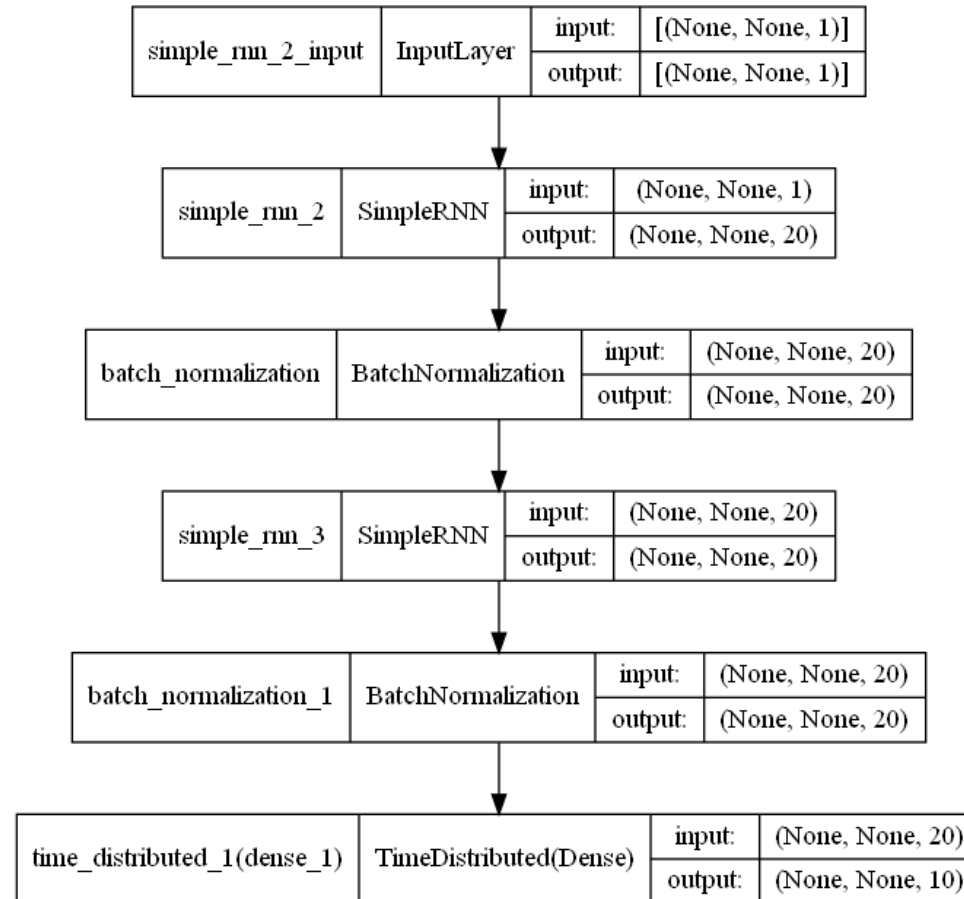
PART B: Many-to-Many



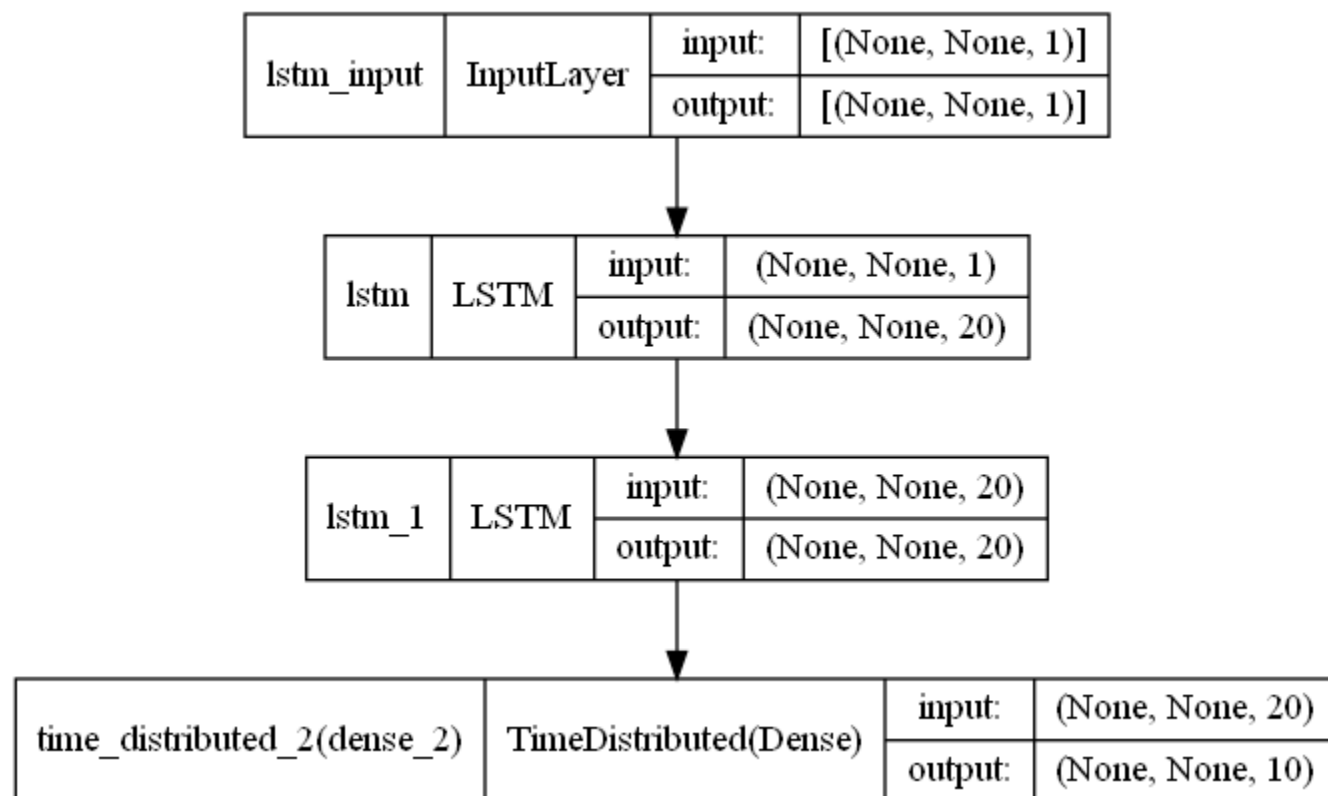
B1. Simple RNN



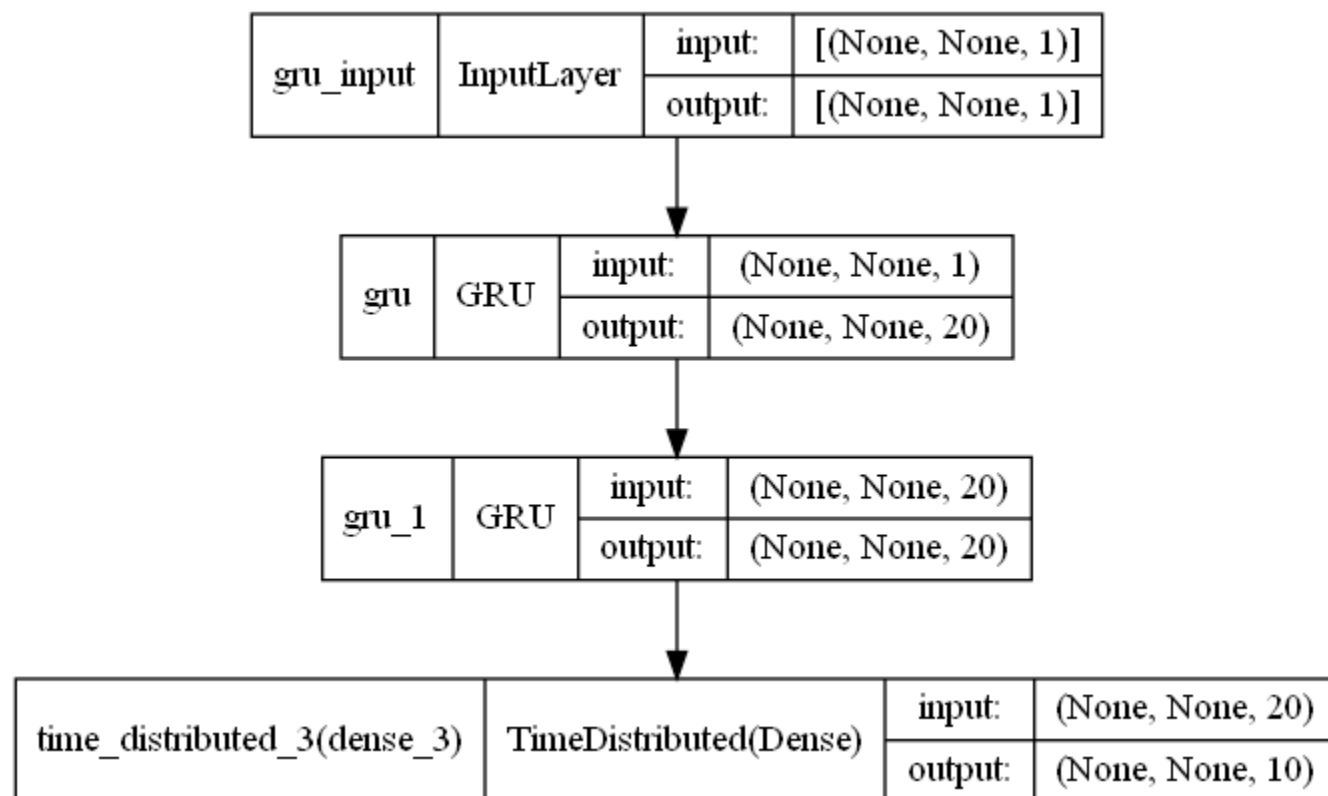
B2. Deep RNN with Batch Norm



B3. LSTM



B4. GRU



B5. 1D-CNN

1D conv layer with kernel size 4, stride 2, VALID padding:

```

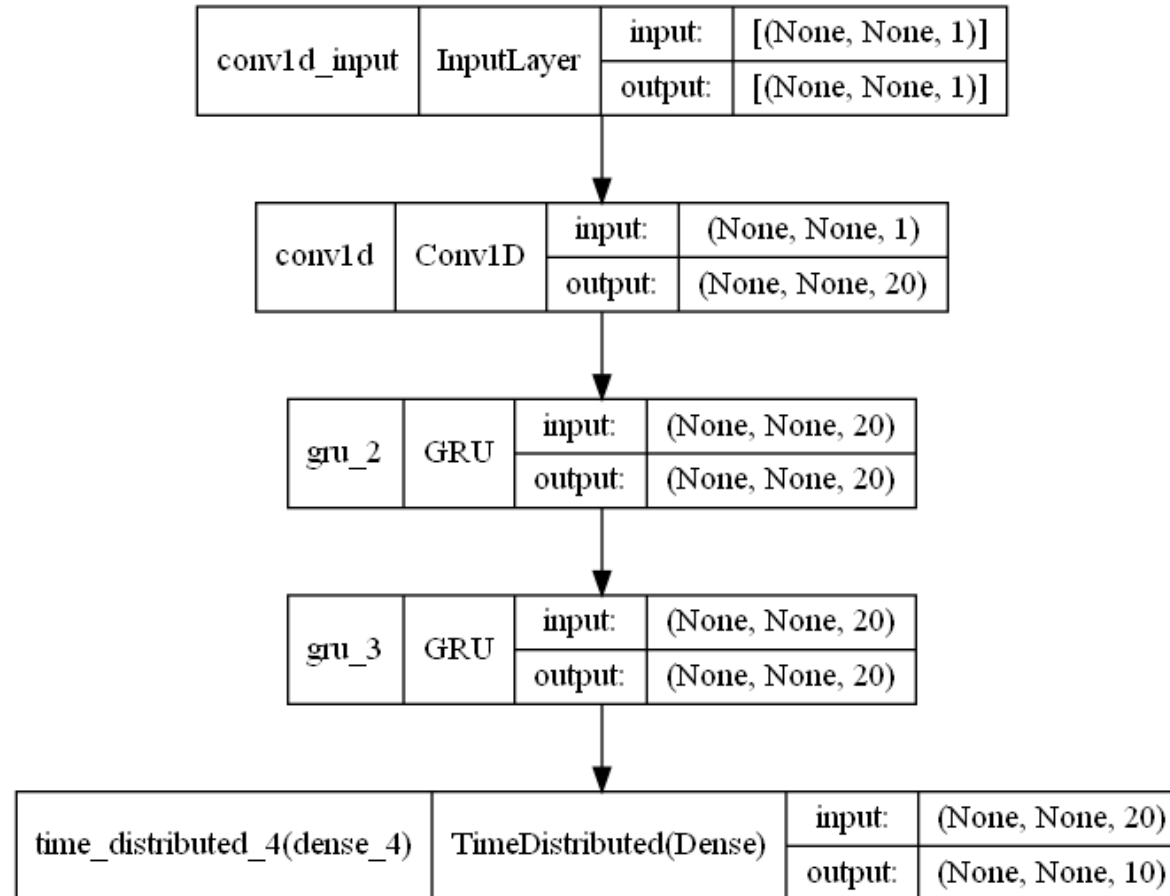
          |-----2-----|          |-----5---...-----|          |-----23-----| |
        |-----1-----|          |-----4-----|          ...          |-----22-----|
      |-----0-----|          |-----3-----|          |---...|-----21-----|
X: 0  1  2  3  4  5  6  7  8  9  10 11 12 ... 42 43 44 45 46 47 48 49
Y: 1  2  3  4  5  6  7  8  9  10 11 12 13 ... 43 44 45 46 47 48 49 50
   /10 11 12 13 14 15 16 17 18 19 20 21 22 ... 52 53 54 55 56 57 58 59
```

Output:

```

X:    0/3   2/5   4/7   6/9   8/11 10/13 .../43 42/45 44/47 46/49
Y:    4/13  6/15  8/17 10/19 12/21 14/23 .../53 46/55 48/57 50/59
```

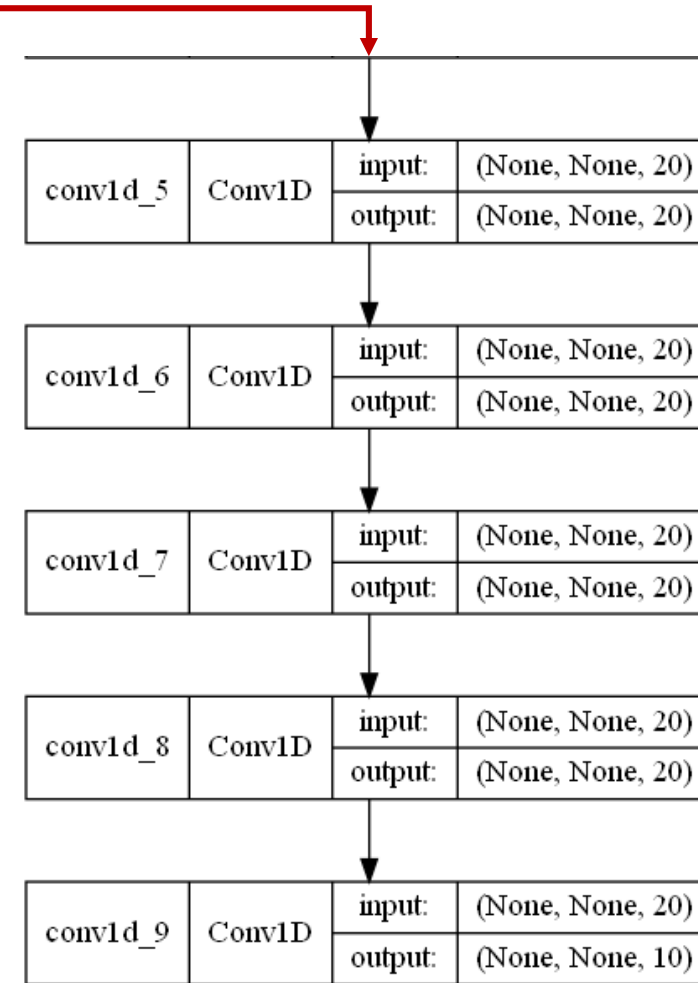
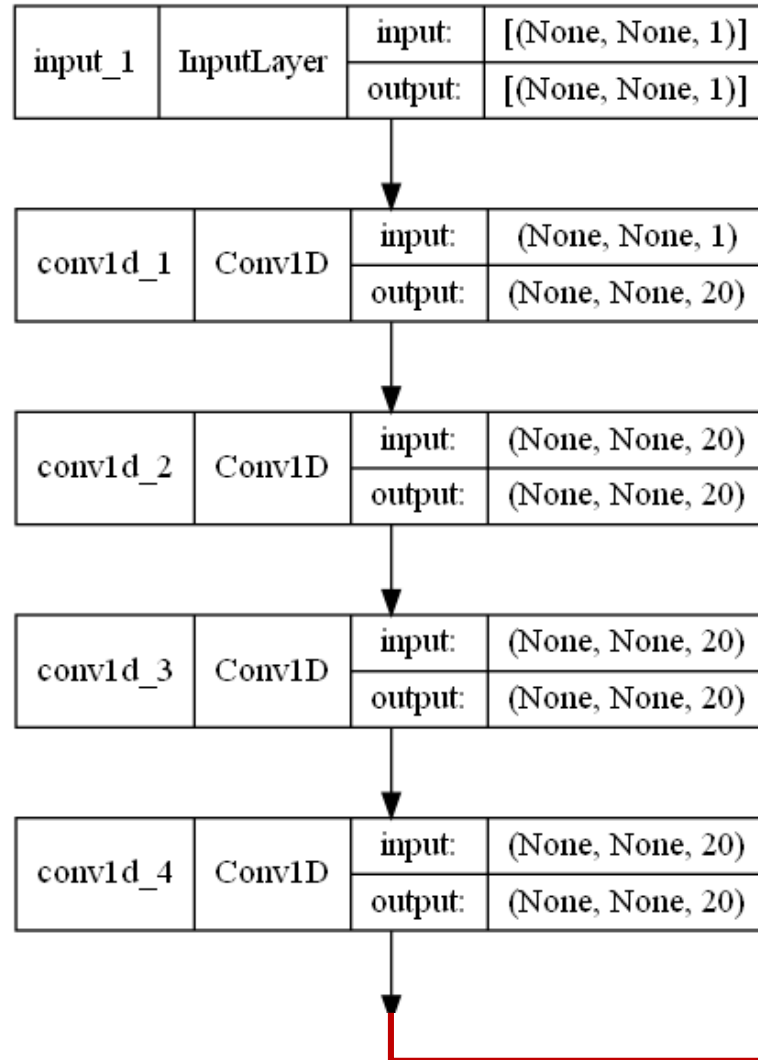
B5. 1D-CNN



B6. WaveNet

C2 /W /W /W /W /W /W /W /W /W /W /W /W /W.../W /W /W /W /W /W /W
 W / W / W / W / W / W / W / W / W / W / W / W / W /
 / W / W / W / W / W / W / W / W / W /
C1 /W /W /W /W /W /W /W /W /W /W /W /W /W.../W /W /W /W /W /W /W /W
X: 0 1 2 3 4 5 6 7 8 9 10 11 12 ... 43 44 45 46 47 48 49
Y: 1 2 3 4 5 6 7 8 9 10 11 12 13 ... 44 45 46 47 48 49 50
 /10 11 12 13 14 15 16 17 18 19 20 21 22 ... 53 54 55 56 57 58 59

B6. WaveNet



Results Summary of Many-to-Many : PART B

```
In [50]: models = pd.DataFrame({  
    'Model': ['SimpleRNN', 'DeepRNN', 'LSTM', 'GRU', '1D-CNN', 'WaveNet'],  
    'Score': [b1, b2, b3, b4, b5, b6]})  
models.sort_values(by='Score', ascending=True)
```

Out [50]:

	Model	Score
4	1D-CNN	0.026147
5	WaveNet	0.026312
0	SimpleRNN	0.027985
2	LSTM	0.033550
3	GRU	0.036676
1	DeepRNN	0.042562