哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Big data technology and Practice

李春山

2020年10月8日

内容提要

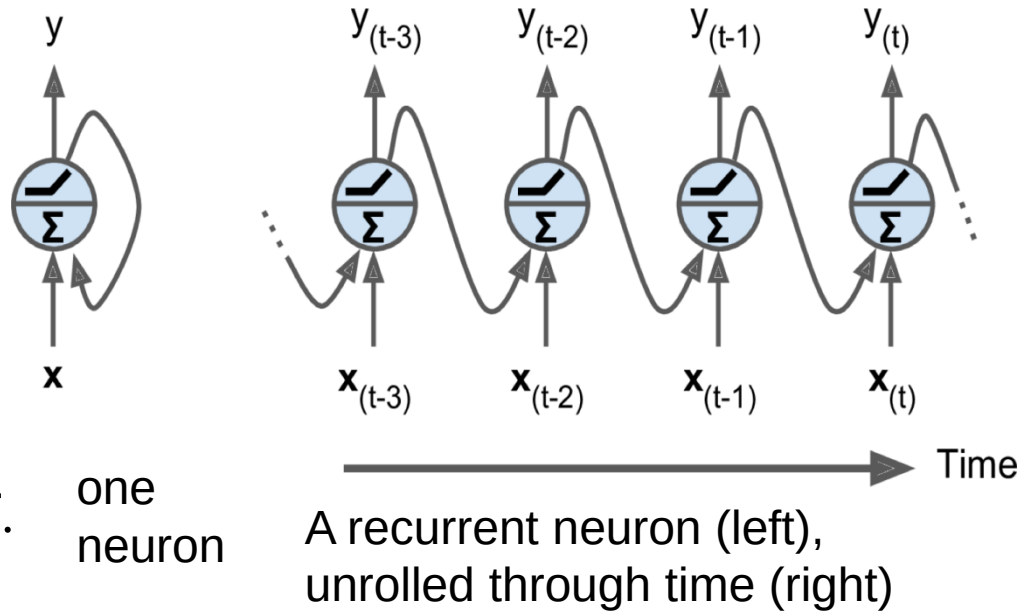
Chapter 15 Recurrent Neural Networks (RNNs)

Chapter 15 Recurrent Neural Networks (RNNs)

- RNNs can predict the future, for examples
  - To analyze time series data of stock prices, and tell you when to buy or sell.
  - In autonomous driving systems, they can anticipate car trajectories and help avoid accidents.
- Generally, they can work on sequences of arbitrary lengths, for example of NPL, taking sentences, documents, or audio samples as input, making them for automatic translation, speech-to-text, or sentiment analysis (e.g., reading movie reviews and extracting the rater's feeling about the movie.
- Moreover, RNNs are capable of surprising "creativity". You can ask them to predict which are the most likely next notes in a melody, then randomly pick one of these notes and play it.

1. Recurrent Neurons

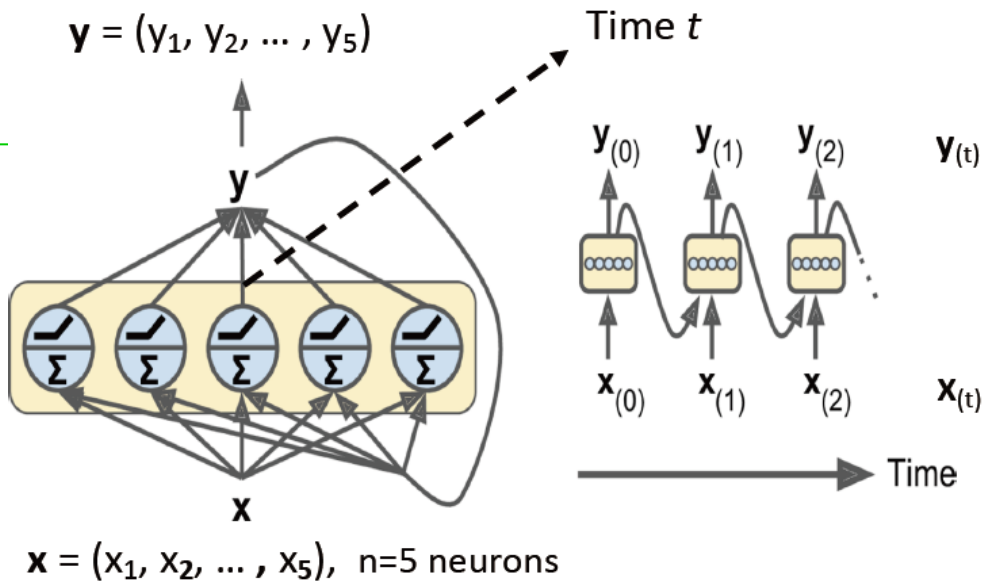
- A RNN looks very much like a feedforward neural network, except it also has connections pointing backward.
- See a simple RNN, composed of just one neuron receiving inputs, producing an output, and sending that output back to itself.



- At each time step t (or frame), this recurrent neuron receives the inputs x(t) as well as its own output from the previous time step, y(t–1).
- We can represent this tiny network against the time axis, as shown. This is called unrolling the network through time.

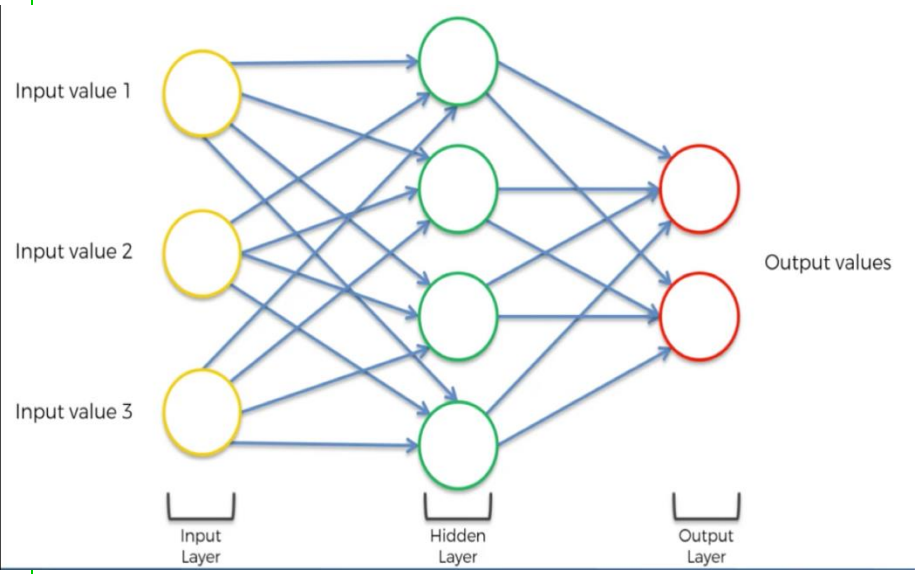
Recurrent Neurons

- You can create a layer of recurrent neurons
- At each time step t, every neuron receives both the input vector x(t) and the output vector from the previous time step y(t–1).



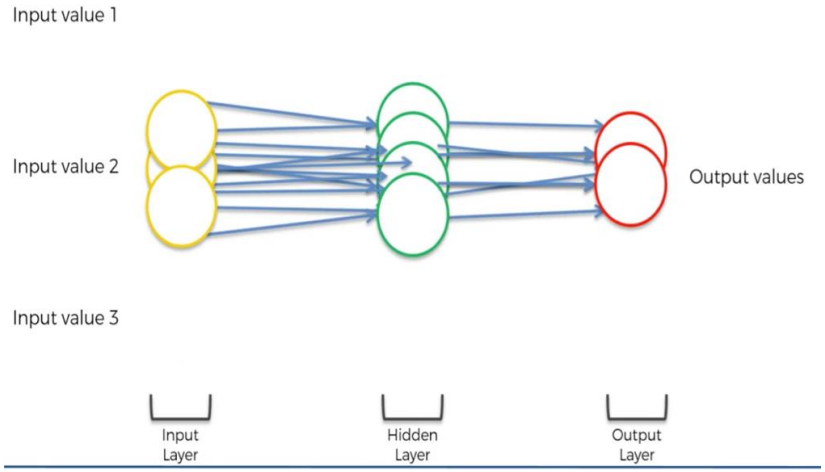
- Note that both the inputs and outputs are vectors now (when there was just a single neuron, the output was a scalar).
- Each recurrent neuron has two sets of weights: one for the inputs x(t) and the other for the outputs of the previous time step, $y_{(t-1)}$. Let's call these weight vectors w_x and w_y. Say, . $\mathbf{wx} = (w_{x1}, w_{x2}, \ldots, w_{x5})$
- The output of a single recurrent neuron, yi, i =1,2,..,5, can be computed as you might expect.

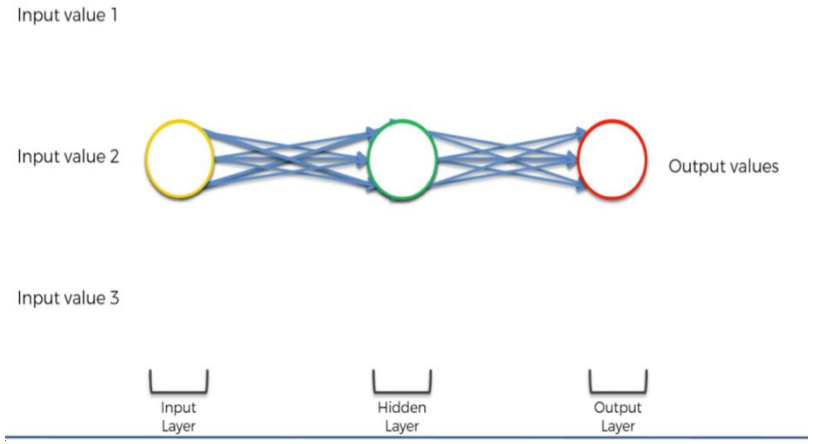
Revisit ANNs for understanding RNNs



A typical artificial neural network with input layer, hidden layer and output layer

If we change an angle to view it:

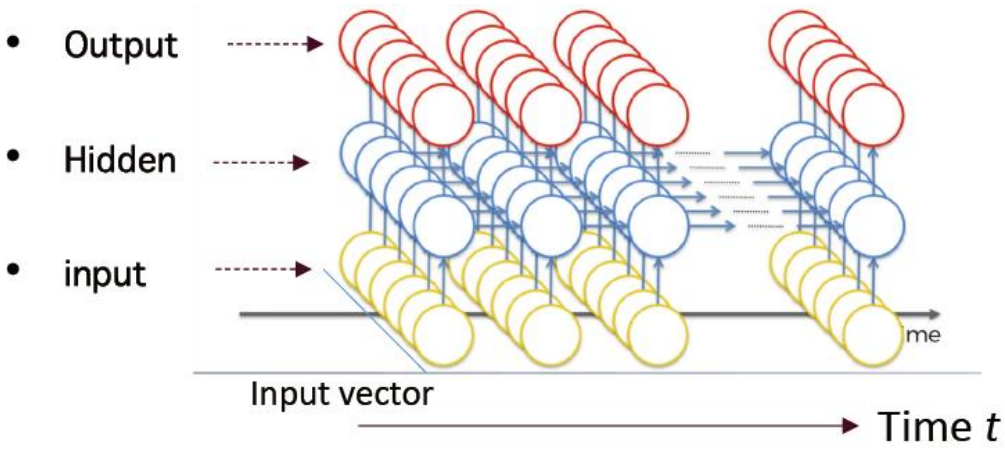




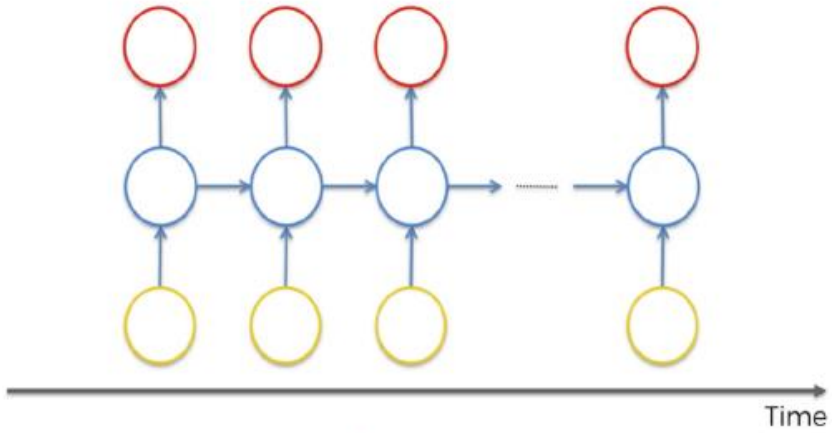
Input vector

Understanding RNNs

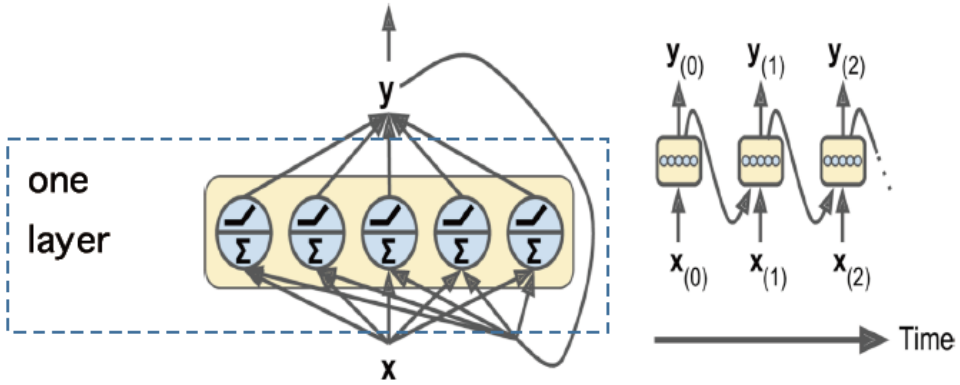
- **Rotate 90 degree**



- **A Recurrent Neural Network**



$\mathbf{y} = (y_1, y_2, \dots, y_5)$



$\mathbf{x} = (x_1, x_2, \dots, x_5)$, n=5

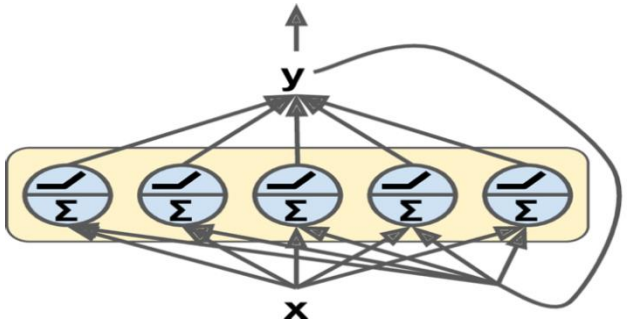
Note:

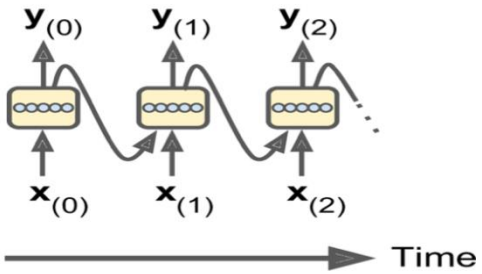
- here, each node represents a whole layer.

Understanding RNNs

- Output of a recurrent layer for a single instance at time t

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{x}_{(t)}^{T} \cdot \mathbf{w}_x + \mathbf{y}_{(t-1)}^{T} \cdot \mathbf{w}_y + b\right)$$





$\mathbf{x} = (x_1, x_2, \ldots, x_n)$

$\mathbf{y} = (y_1, y_2, \ldots, y_n)$

$\mathbf{w_x} = (w_{x1}, w_{x2}, \ldots, w_{xn})$

$\mathbf{w_y} = (w_{y1}, w_{y2}, \ldots, w_{yn})$

- Where b is the bias term and $\phi(.)$ is the activation function e.g., ReLU.
- y(t), x(t), w(x), and w(y) are vectors
- Just like for feedforward neural networks, we can compute a whole layer's output, based on the equation as above:

Understanding RNNs

- Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\mathbf{Y}_{(t)} = \phi\left(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b}\right)$$
$$= \phi\left(\left[\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}\right] \cdot \mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}$$

- Now, y(t), x(t), w(x), and w(y) are matrixs.

- $\mathbf{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and n_{neurons} is the number of neurons).
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- $\mathbf{W}_x$ is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $\mathbf{W}_y$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ are often concatenated into a single weight matrix $\mathbf{W}$ of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of Equation 14-2).
- $\mathbf{b}$ is a vector of size n_{neurons} containing each neuron's bias term.

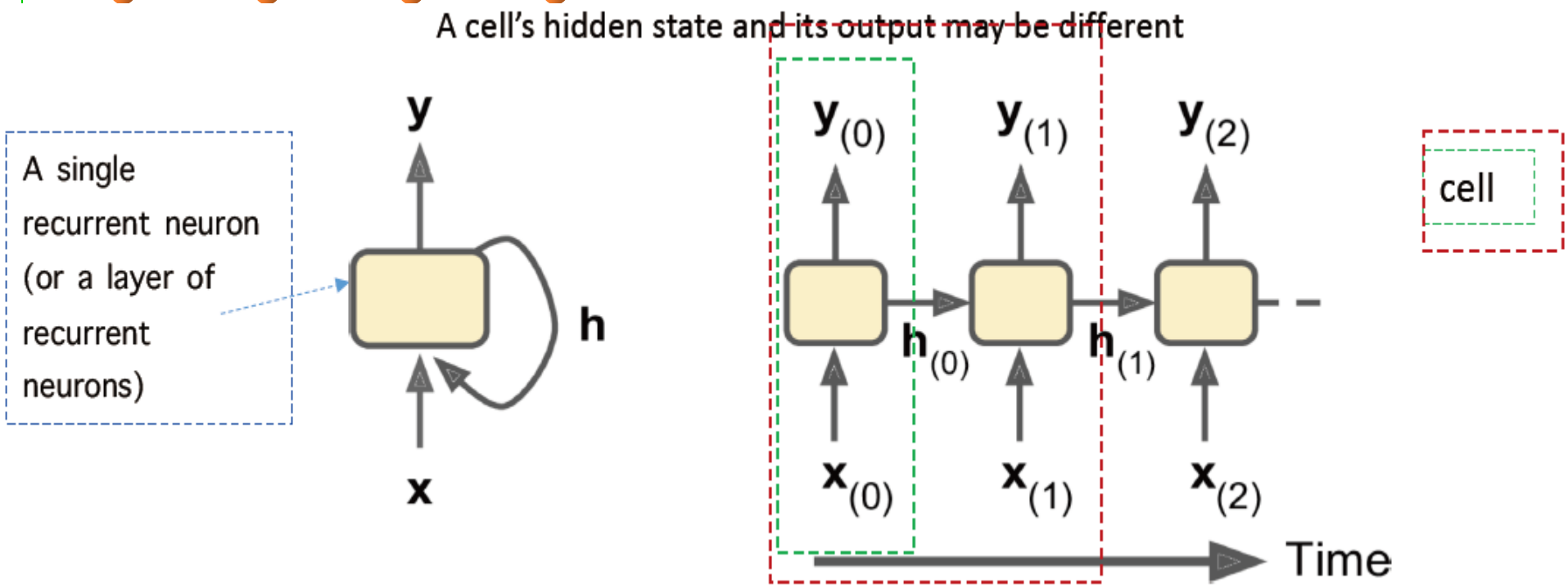
Notice that Y(t) is a function of X(t) and Y(t–1), which is a function of X(t–1) and Y(t–2), which is a function of X(t–2) and Y(t–3), and so on.

This makes Y(t) a function of all the inputs since time t = 0 (that is, X(0), X(1), …, X(t)). At the first time step, t = 0, there are no previous outputs, so they are typically assumed to be all zeros.

Memory Cells

- Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of memory.
- A part of a neural network that preserves some state across time steps is called a memory cell (or simply a cell).
- A single recurrent neuron (or a layer of recurrent neurons) is a very basic cell, but later, more complex and powerful types of cells.

Memory Cells

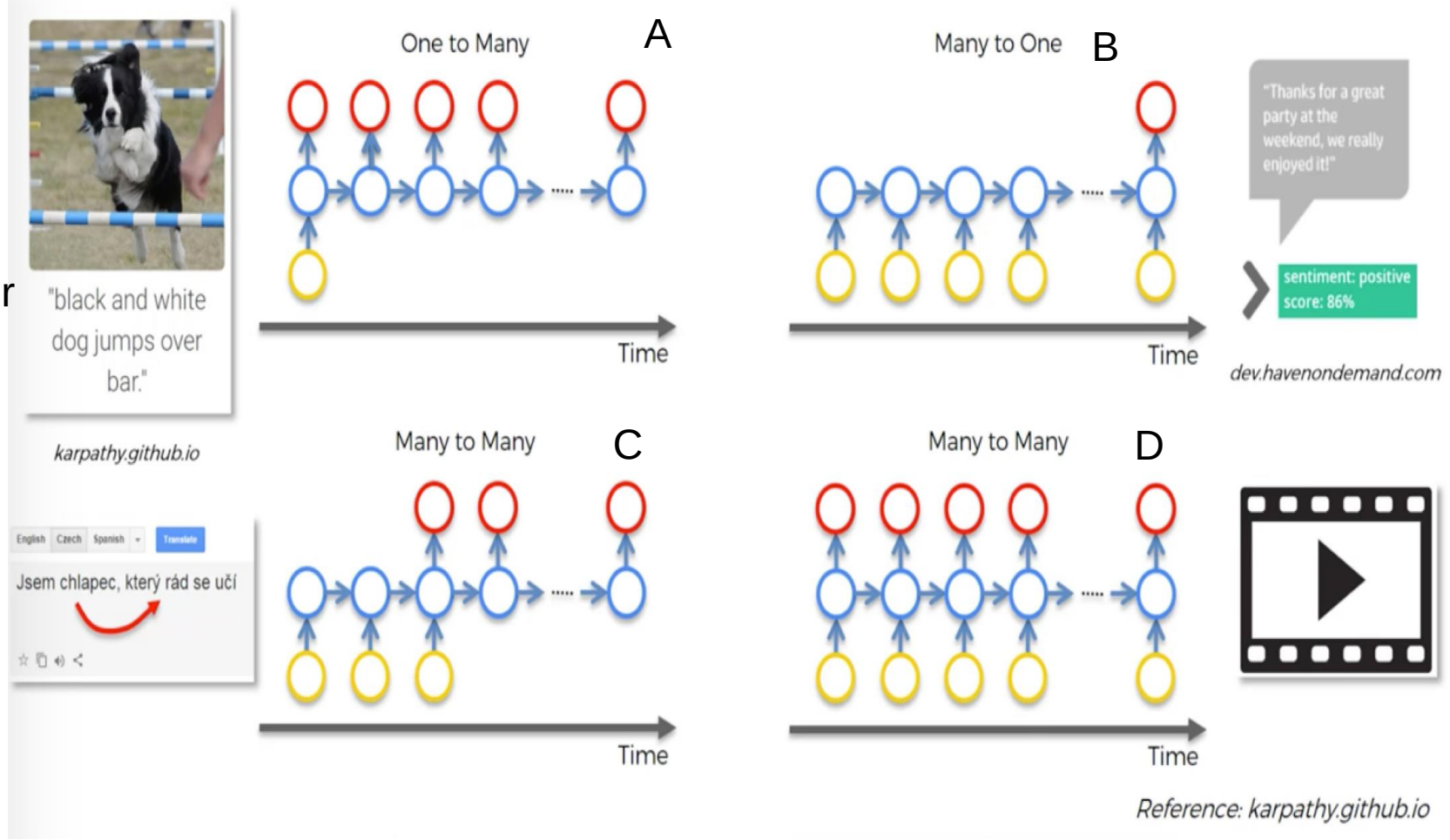


- In general, a cell's state at time step t, denoted h(t) (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: h(t) = f(h(t–1), x(t)). Its output at time step t, denoted y(t) , is also a function of the previous state and the current inputs.

Input and Output Sequence

- An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs. They may have different types, for examples:

A single recurrent neuron (or a layer of recurrent neurons)



Input and Output Sequence

- A. One to many: you could feed the network a single input at the first time step (and zeros for all other time steps), and let it output a sequence (see the bottom-left network). So, the same input is repeatedly processed. For example, the input could be an image, and the output could be a caption for that image.

- B. Many to one: feed the network a sequence of inputs, and ignore all outputs except for the last one. It means that only a sequence of inputs are considered. For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from –1 [hate] to +1 [love]).

Input and Output Sequence

- C. Many to many (Encoder- Decoder): this can be used for translating a sentence from one language to another.

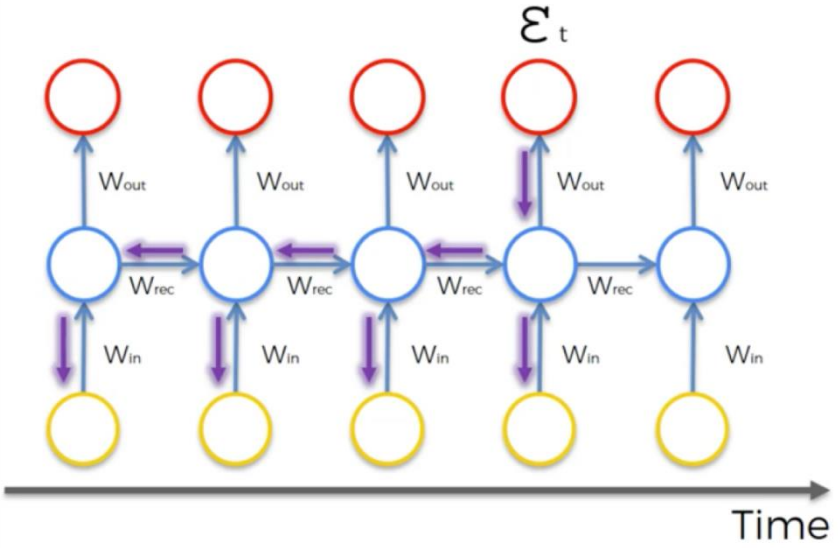
- D. Many to many such as for predicting stock prices, you feed it the prices over the last N days, and it must output the prices shifted by one day into the future (i.e., from N – 1 days ago to tomorrow).

2. The Issues on Training RNNs

- Similar to training ANN, but more complicated.
- Cost functions for each output, can be defined, and use them to evaluate errors For example at time t

During the backpropagation, "vanishing gradient problem" may occur.

It causes a RNN training failed.

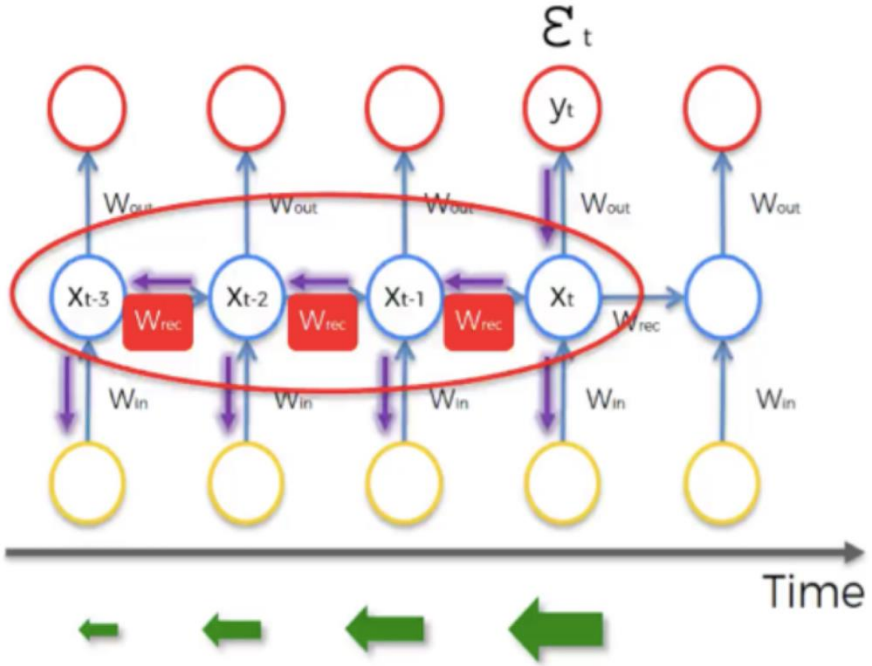


1). Review the concept of Gradient

Just as a straight line expresses a change in x alongside a change in y, the gradient expresses the change in all weights with regard to the change in error.

If we can't know the gradient, we can't adjust the weights in a direction that will decrease error, and our network ceases to learn.

2). Vanishing Gradient Problem



$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \le t \le T} \frac{\partial \mathcal{E}_t}{\partial \theta} \quad (3)$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \quad (4)$$

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \ge i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \ge i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1})) \quad (5)$$

W_{rec} ~ small ⇨ Vanishing

W_{rec} ~ large ⇨ Exploding

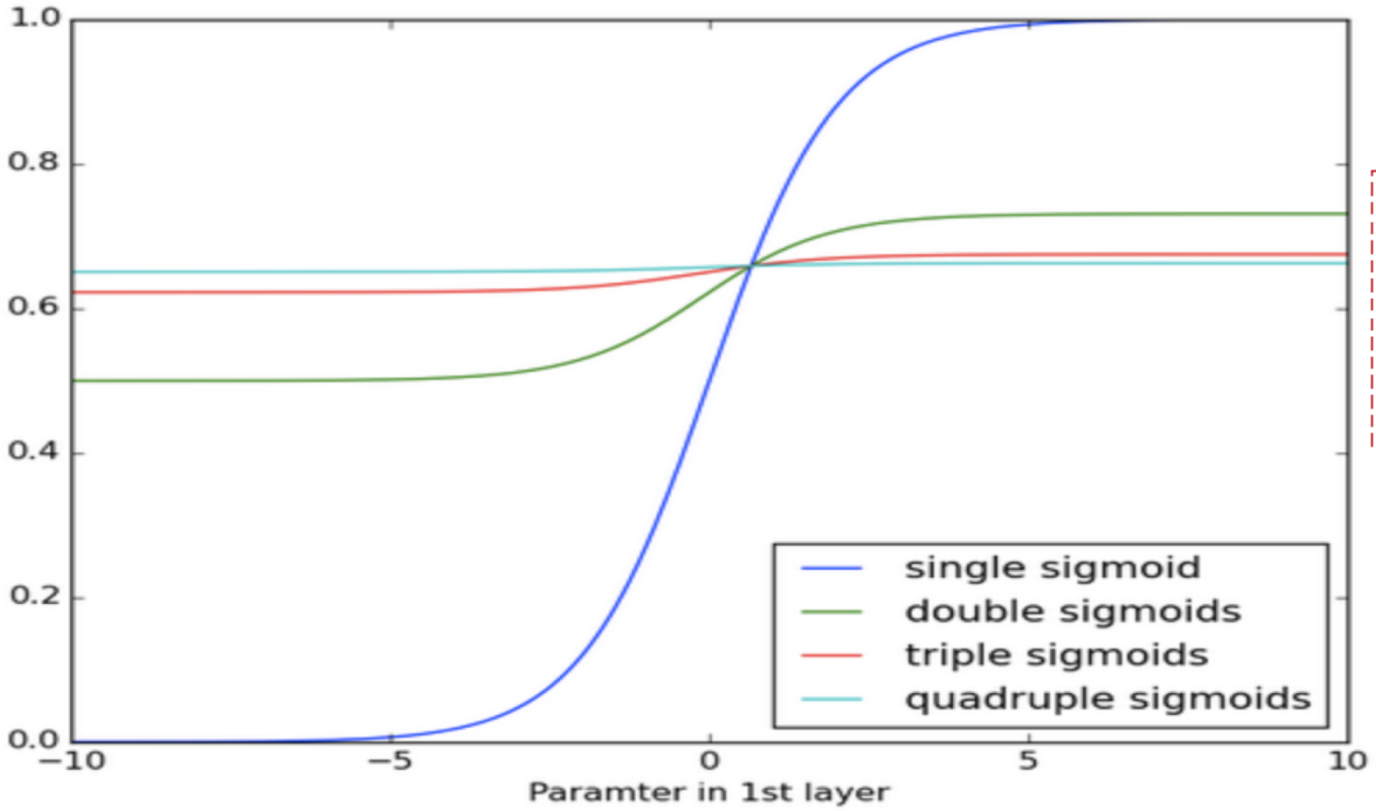
- Reason of Van
- This is partially because the information flowing through neural nets passes through many stages of multiplication.

2). Vanishing Gradient Problem

- Essentially, every single neuron that participated in the calculation of the output, associated with this cost function, should have its weight updated in order to minimize that error.
- But for RNNs training, it's not just the neurons directly below this output layer that contributed, but also all of the neurons far back in time.
- So, you have to propagate all the way back through time to these neurons.
- For instance, to get from x_{t-3} to x_{t-2} we multiply x_{t-3} by w_{rec} .. Then, to get from x_{t-2} to x_{t-1} we again multiply x_{t-2} by w_{rec} .
- So, we multiply with the same exact weight multiple times, and this is where the problem arises: when you multiply something by a small number, your value decreases very quickly. (Gamblers go bankrupt fast when they win just 97 cents on every dollar they put in the slots.)

For example,

- When applying a sigmoid function over and over again, the data is flattened and it has no detectable slope. This is analogous to a gradient vanishing as it passes through many layers.



f(f(f(f(x)...))),

"f" is a sigmoid function

3). Long Short-Term Memory Units (LSTMs)

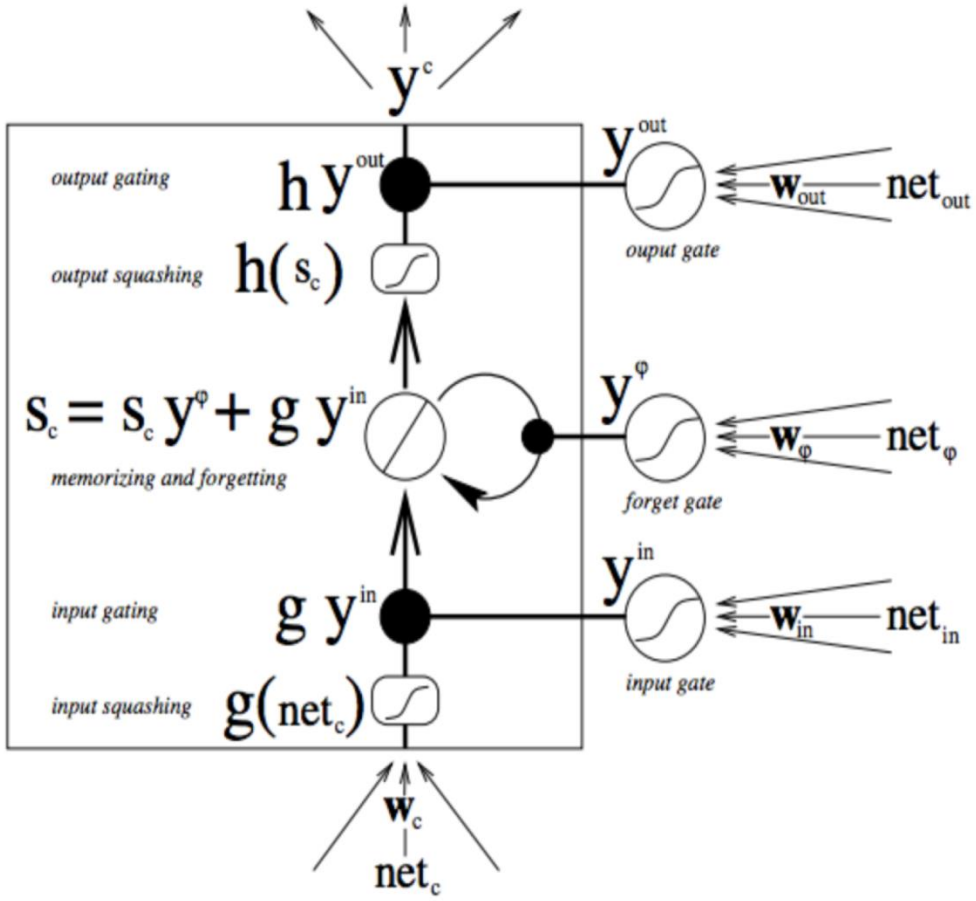
- **Two** possible issues on training a RNN
- Exploding gradients:
  - The weights' gradients become saturated on the high end.
  - Exploding gradients can be solved relatively easily, because they can be truncated or squashed.
- **Vanishing gradients**
  - The gradients become too small for computers to work with or for networks to learn – a harder problem to solve.
- **A popular solution is "LSTM"**
- In the mid-90s, it was proposed by the German researchers Sepp Hochreiter and Juergen Schmidhuber.

Long Short-Term Memory Units (LSTMs)

- LSTMs help preserve the error that can be backpropagated through time and layers.
- By maintaining a more constant error, they allow recurrent nets to continue to learn over many time steps (over 1000).
- LSTMs contain information outside the normal flow of the recurrent network in a gated cell.
- Information can be stored in, written to, or read from a cell, much like data in a computer's memory.
- The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close.

Long Short-Term Memory Units (LSTMs)

- Three control gates
- each gate can be open or shut, and they will recombine their open and shut states at each step.
- **Sc** - the current state of the memory cell
- gyin - the current input to it.
- The large bold letters give us the result of each operation.



LSTM has implemented in the Keras library, vey easy to use.
More details: Article “A Beginner's Guide to LSTMs and Recurrent Neural Networks”, https://skymind.ai/wiki/lstm

2. Implement RNNs with Keras

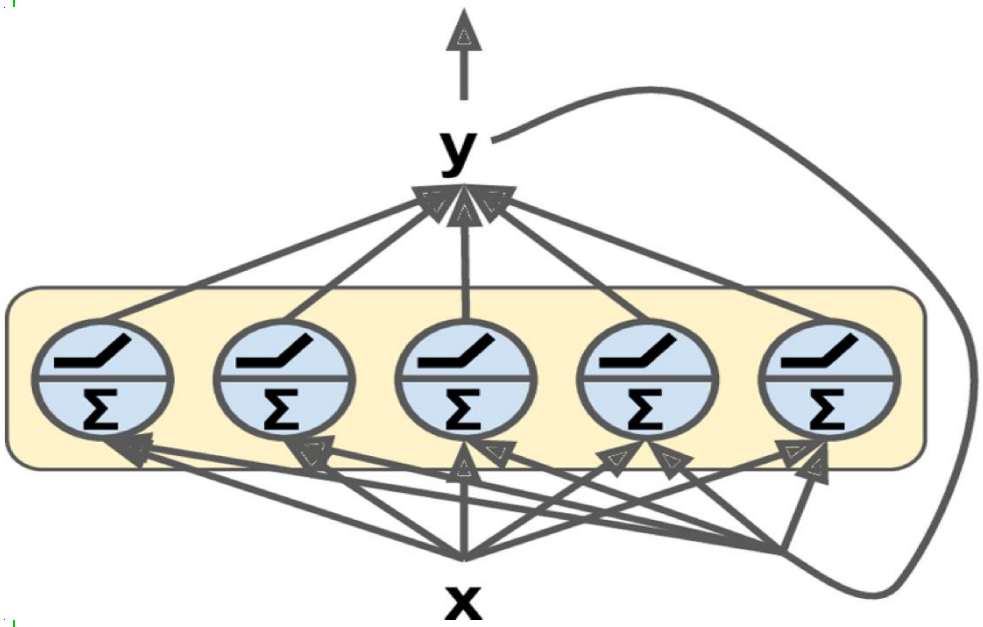
- By using a sample, we discuss this topic, presented in a notebook, RNN.ipynb , located in …/Lab8/RNN_with_Keras/
- In this sample, we will try to predict the upward and downward trends that exist in the Google stock price, by using a Recurrent Neural Network (RNN) with Keras.
- Indeed, there is a Brownian motion that states that the future evaluations of the stock price are independent from the past.
- But it's actually possible to predict some trends.
- **We're going to train our data on five years of the Google stock price, from the beginning of 2012 to the end of 2016 and then try to predict the stock price of the first month of 2017.**

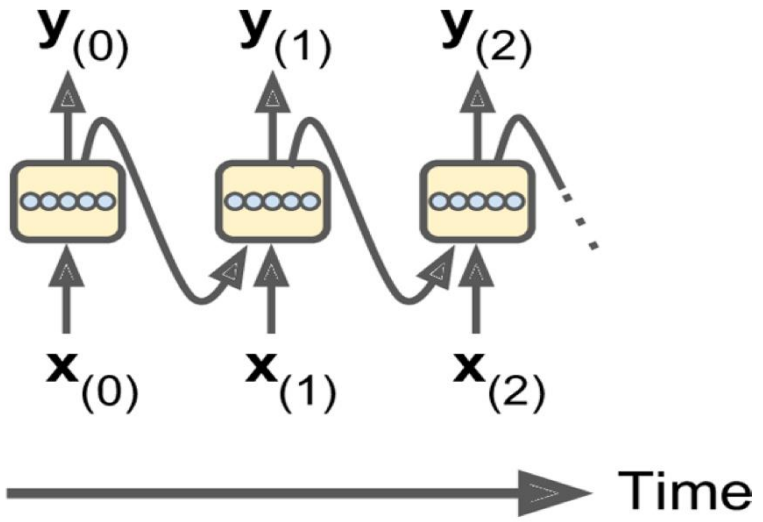
2. Implement RNNs with Keras

- The sample will produce the visualizing prediction of the upward and downward trends that exist in the Google stock price.
- We will have a study on it in Lab class.

3. Implement RNNs in TensorFlow

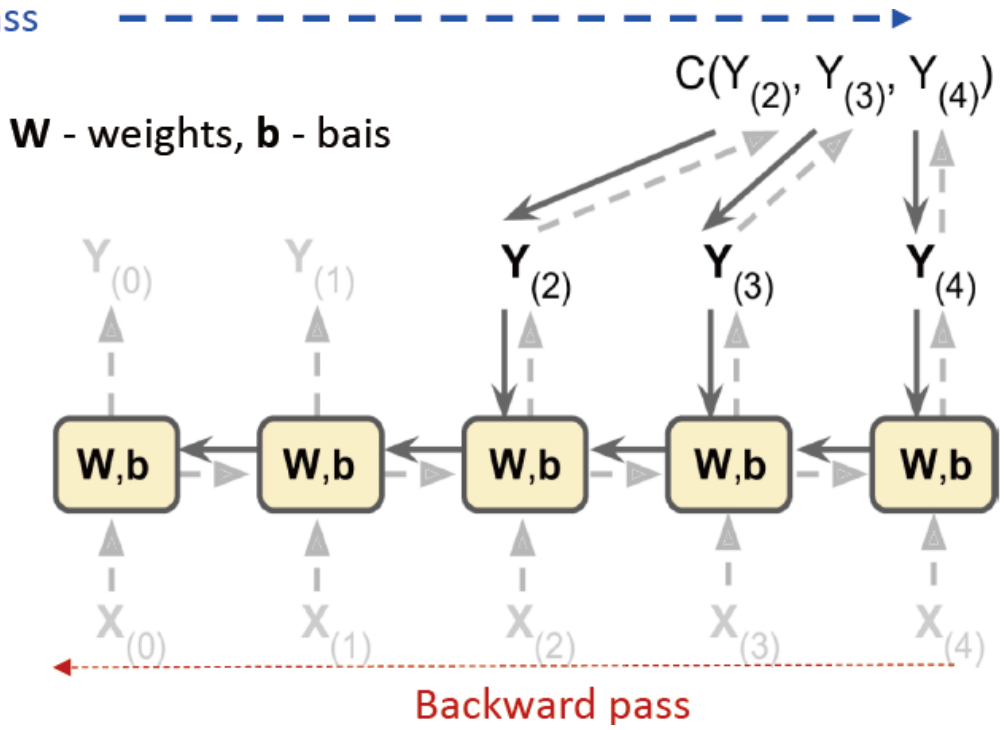
- There are different ways to implement RNNs model.
- We will learn these approaches together with its coding samples in lab time.





1). Training RNNs

- To train an RNN, the trick is to unroll it through time and then simply use regular backpropagation, called backpropagation through time (BPTT).



- Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows); then the output sequence is evaluated using a cost function:

$$C\left(\mathbf{Y}_{(t_{\min})}, \mathbf{Y}_{(t_{\min}+1)}, \cdots, \mathbf{Y}_{(t_{\max})}\right)$$

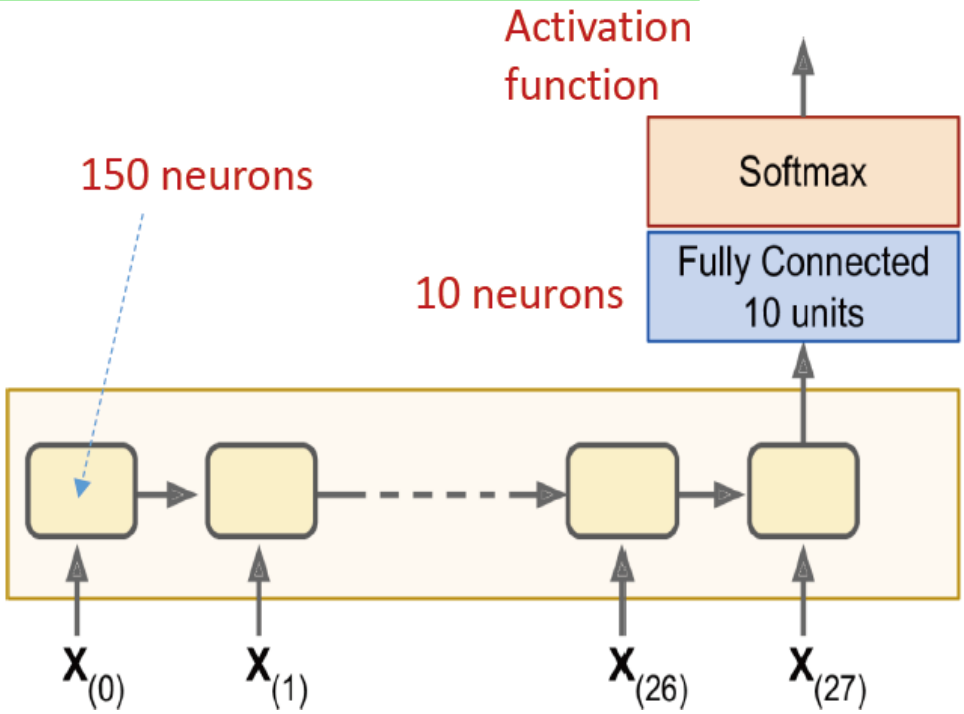
where tmin and tmax are the first and last output time steps, not counting the ignored outputs.

1). Training RNNs

- The gradients of that cost function are propagated backward through the unrolled network (represented by the solid arrows); finally, the model parameters are updated using the gradients computed during BPTT.
- Note that the gradients flow backward through all the outputs used by the cost function, not just through the final output, for this example, the cost function is computed using the last three outputs of the network, Y(2), Y(3), and Y(4), so gradients flow through these three outputs, but not through Y(0) and Y(1).
- Moreover, since the same parameters (or weights) W and b are used at each time step, backpropagation will do the right thing and sum over all time steps. (their values would be different at each step, due to updating).

Case 1: Training a Sequence Classifier

- Let's train an RNN to classify MNIST images.
- A convolutional neural network would be better suited for image classification (in the last Chapter) but , as a simple example that you are already familiar with, we will treat each image as a sequence of 28 rows of 28 pixels each (since each MNIST image is 28 x 28 pixels).

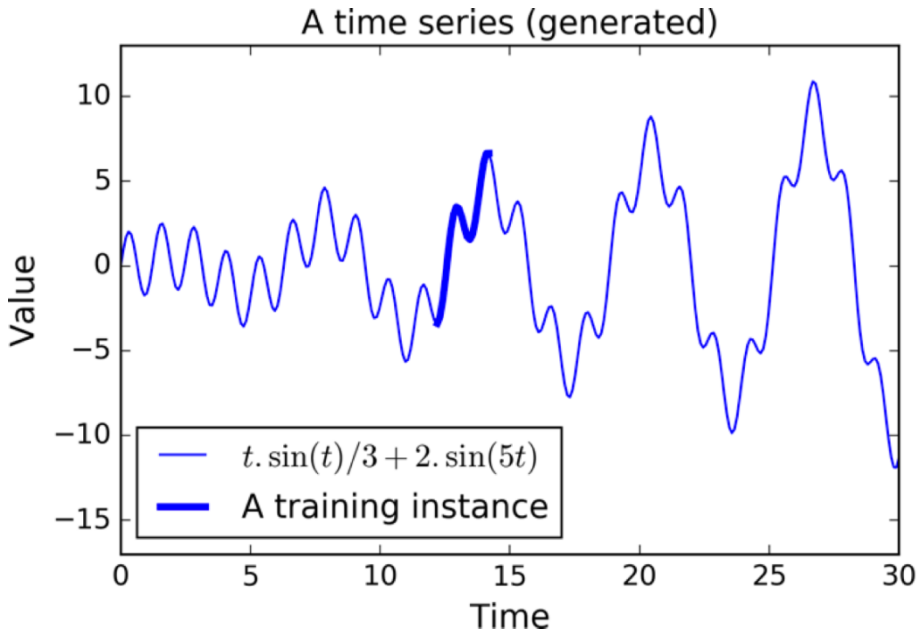


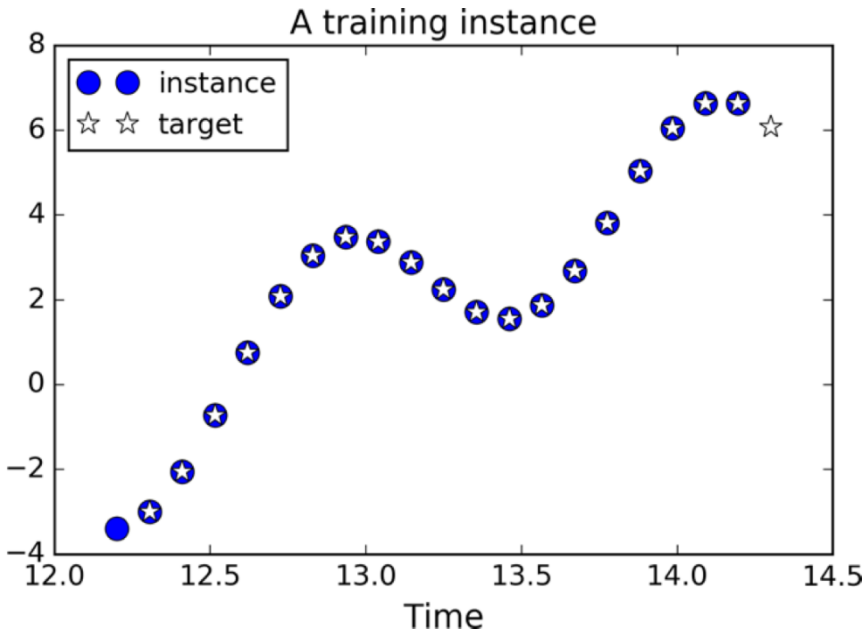
We will use cells of 150 recurrent neurons, plus a fully connected layer containing 10 neurons (one per class) connected to the output of the last time step, followed by a softmax layer.

•See the codes titled "Training a sequence classifier" the notebook, recurrent_neural_networks.ipynb, in Lab time.

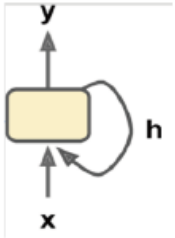
Case 2: Training to Predict Times Series

- Let's look at how to handle time series, such as stock prices, air temperature, brain wave patterns, and so on.
- Here, we will train an RNN to predict the next value in a generated time series. Each training instance is a randomly selected sequence of 20 consecutive values from the time series, and the target sequence is the same as the input sequence, except it is shifted by one time step into the future:



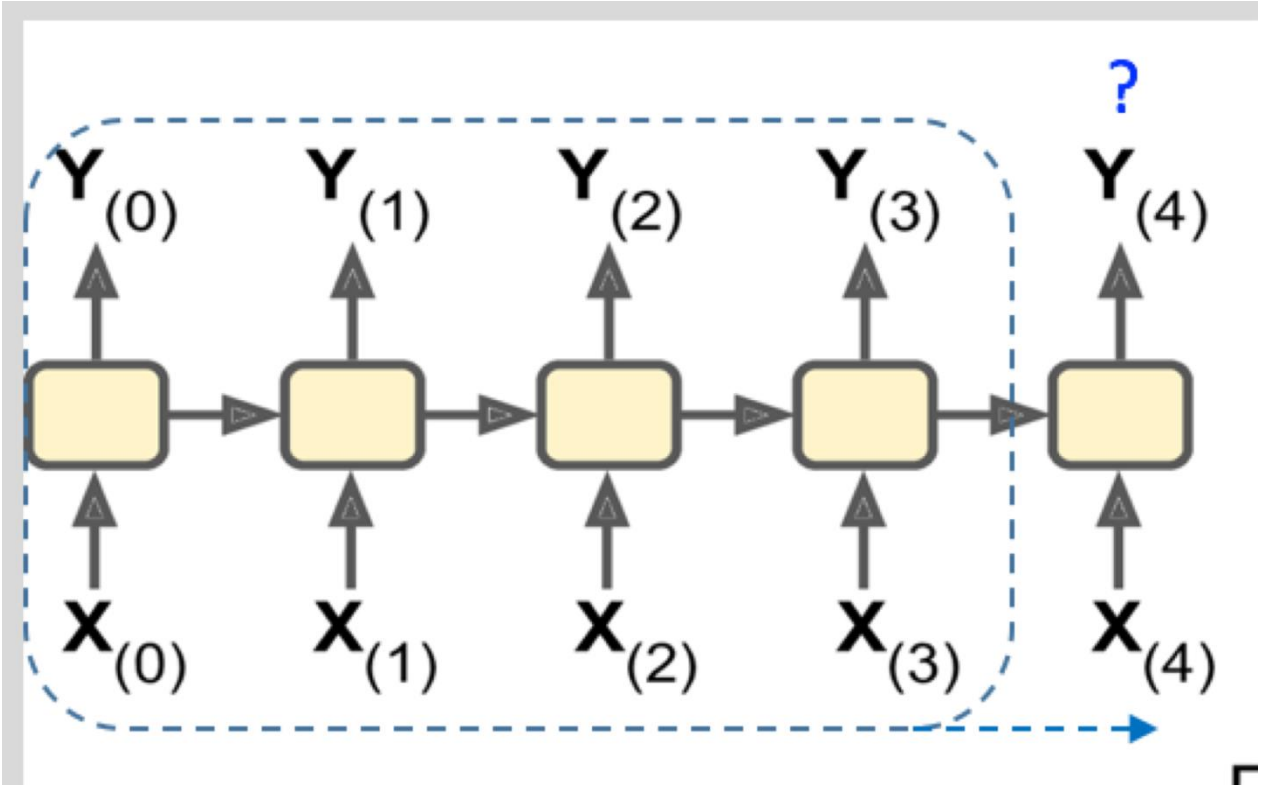


Training to Predict Times Series




```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1
```

- a layer of 100 recurrent neurons)



Training to Predict Times Series

- First, let's create the RNN. It will contain 100 recurrent neurons and we will unroll it over 20 time steps since each training instance will be 20 inputs long.
- Each input will contain only one feature (the value at that time). The targets are also sequences of 20 inputs, each containing a single value. The code is almost the same as earlier:

```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

- See the codes titled "Time Series" the notebook,recurrent_neural_networks.ipynb, in Lab time.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

结束

2020年10月8日