Deep Learning for Timeseries

2021.7.17

10.1 Different kinds of timeseries task

A **timeseries** can be any data obtained via measurements at regular intervals. By far, the most common timeseries-related task is **forecasting**: predicting what happens next in the series. Forecasting is what this chapter focuses on. But there's actually a wide range of other things we can do with timeseries, such as:

Classification: assign one or more categorical labels to a timeseries. For instance, given timeseries of activity of a vistor on a website, classify whether the visitor is a bot or a human.

Event detection: identify the occurence of a specific, expected event within a continuous data stream. A particular useful application is "hotword detection".

Anomaly detection: Detecting anything unusual happening within a continuous datastream. In this chapter, we'll learn about Recurrent Neural Networks (RNNs) and how to apply them to timeseries forecasting.

10.2 A temperature forecasting example

Let's go back to code part.

10.3 Understanding recurrent neural network

Densely connected networks and convnets, they both have no memory. Each input shown to them is processed indenpendently, with no state kept in between inputs. With such networks, in order to process a sequence or a temporal series of data points, we have to show entitre sequences to the network at once: turn it into a sinlge data point. For instance, this is what we did in the densely-connected network example: we flattened our five day of data into a single large vector and processed it in one go. Such networks are called *feedforward networks*.

A RNN adopts the same principle, albeit in an extremely simplified version: it processes sequences by iterating through the sequence elements and maintaining a *state* containing information relative to what it has seen so far. In effect, an RNN is a type of nn that has an internal **loop**.

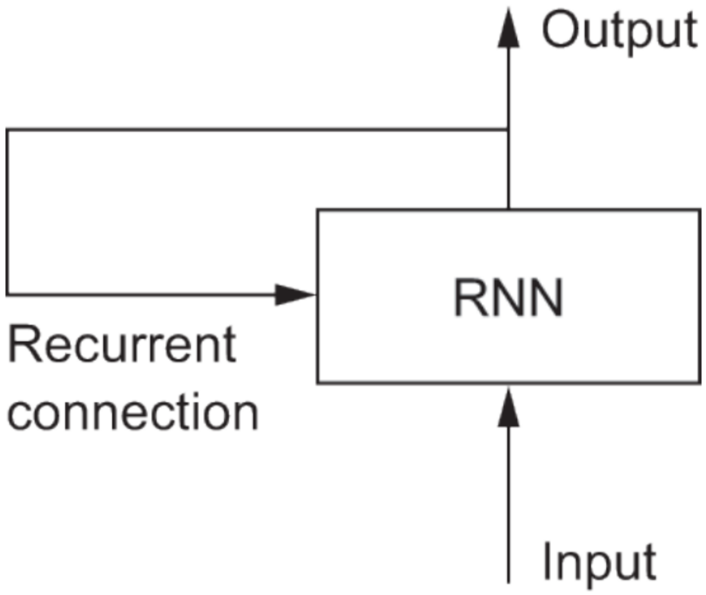

Figure 10.6 A recurrent network: a network with a loop

We still consider one sequence to be a single data point: a single input to the network. What changes is that this data point is no longer processed in a single step; rather, the network internally loops over sequence elements.

```
# Pseudocode RNN
state_t = 0
for input_t in input_sequence:
    output_t = f(input_t, state_t)
    # the previous output becomes the state for the next iteration
    state_t = output_t
```

A more detailed pseudocode for the RNN

```
state_t = 0
for input_t in input_sequence:
    # the trans of the input and state could be parameterized by two matrices, W
   and U, and a bias vector
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

In summary, an RNN is a *for* loop that reuses quantities computed during the previous iteration of the loop. Of course, there are many different RNNs fitting this definition. RNNs are characterized by their step function.

10.3.1 A recurrent layer in Keras

SimpleRNN processes batches of sequences, like all other Keras layers. This means it takes inputs of shape (batch_size, timesteps, input_features). When specifying the shape of argument of our initial input, we can set timesteps entry to be None, which enables our network to process sequences of arbitrary length.

Let's add to SimpleRNN an additional data flow that carries information across timesteps. Call its value at different timestep Ct, where C stands for *carry*. This information will be combined with the input conntection and recurrent connection (via a dense transformation: a dot product with a weight matrix followed by a bias add and the application of an activation function), and it will affect the state being sent to the next timestep (via an activation function and a multiplication operation). Conceptually, the carry dataflow is a way to modulate the next output the next state. So far.

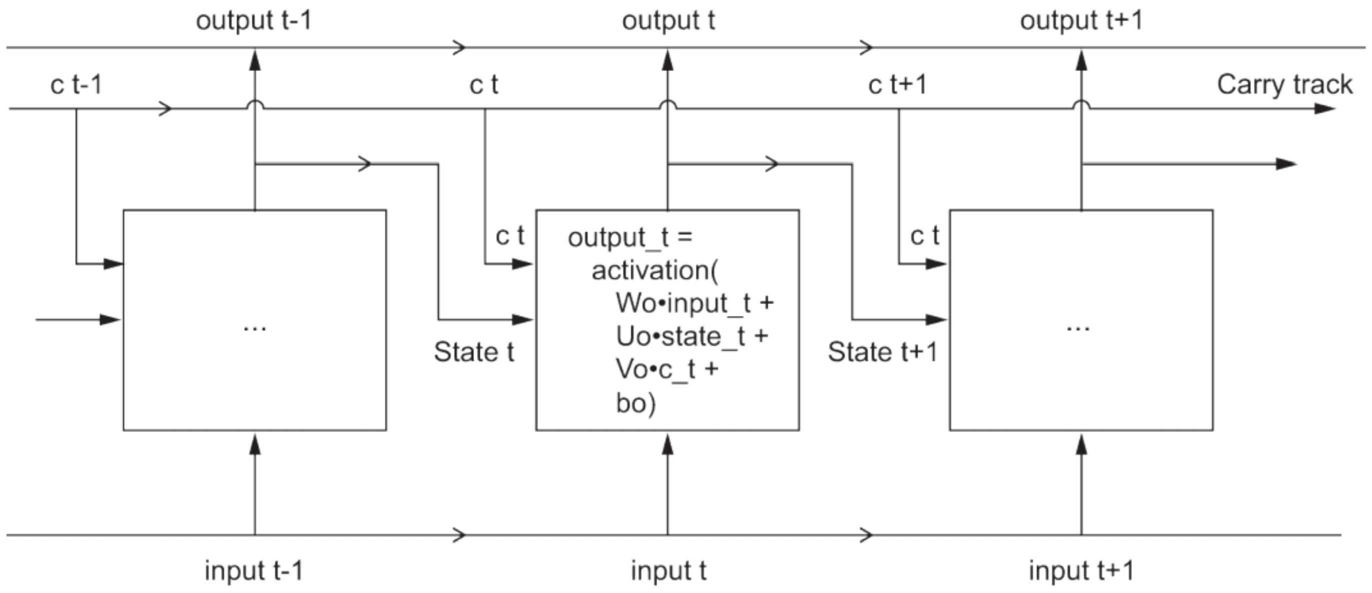

Figure 10.9 Going from a SimpleRNN to an LSTM: adding a carry track

Now the subtely: the way the next Carry dataflow value C_{t+1} is computed. It involves three distinct transformations. All three have the form of a SimpleRNN cell:

```
y = activation(dot(state_t , U) + dot(input_t , W) + b)
```

All three transformation have their own weight matrics, which we'll index

```
output_t = activation(dot(state_t , Uo) + dot(input_t , Wo) + dot(C_t , Vo) + bo)

i_t = activation(dot(state_t , Ui) + dot(input_t , Wi) + bi)
f_t = activation(dot(state_t , Uf) + dot(input_t , Wf) + bf)
k_t = activation(dot(state_t , Uk) + dot(input_t , Wk) + bk)

# we will obtain the new carry state (the next C_t) C_{t+1} by
c_t+1 = i_t * k_t + c_t * f_t
```

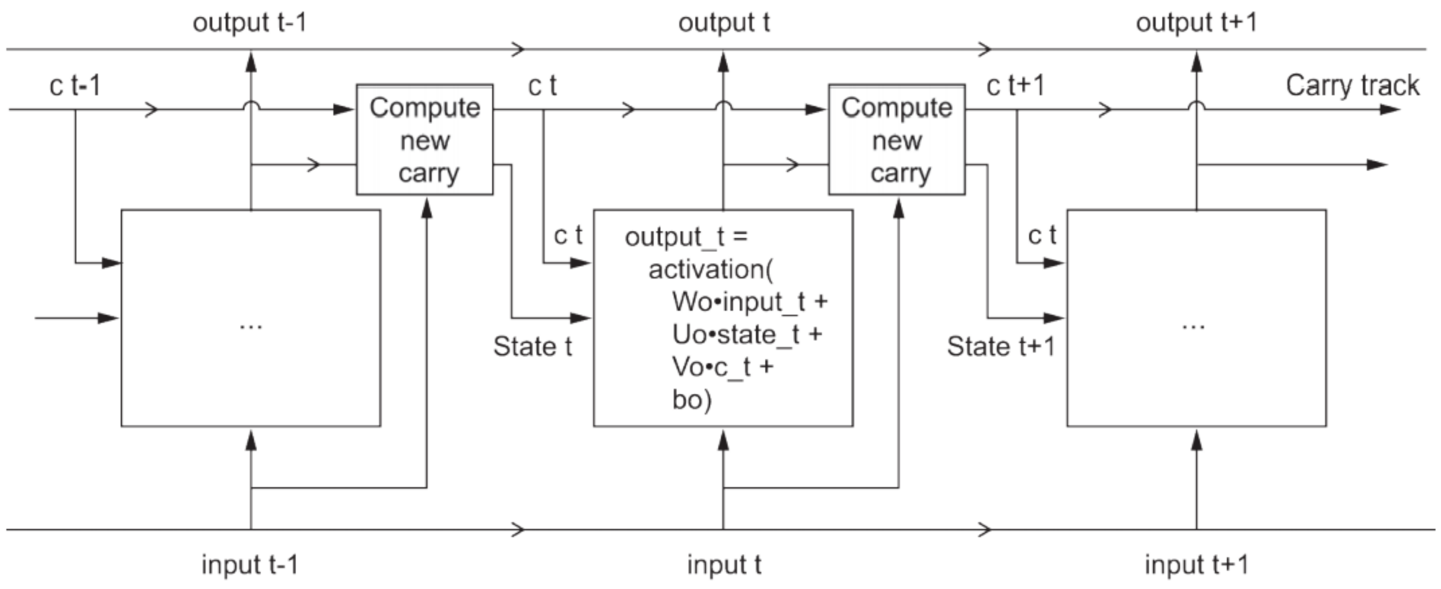

Figure 10.10 Anatomy of an LSTM

In summary, we don't need to understand anything about the specific architecture of an LSTM cell; as an humam, it shouldn't be our job to understand it. Just keep in mind what LSTM cell is meant to do: allow past information to be reinjected at a later time, thus fight against the *vanishing-gradient problem.*

10.4 Advanced use of recurrent neural network.