Deep Learning for Timeseries

Outline

- Machine learning tasks that involve timeseries data
- Understanding recurrent neural networks (RNNs)
- Applying RNNs to a temperature-forecasting example
- Advanced RNN usage patterns

Different kinds of timeseries tasks

- A timeseries can be any data obtained via measurements at regular intervals, like
    - the daily price of a stock,
    - the hourly electricity consumption of a city,
    - or the weekly sales of a store.
- Examples:
    - natural phenomena (like seismic activity, the weather at a location)
    - human activity patterns (like visitors to a website, a country's GDP)

Different kinds of timeseries tasks

- Working with timeseries involves understanding the dynamics of a system—
  - its **periodic cycles**,
  - how it trends over time,
  - its regular regime and
  - its sudden spikes.

Common timeseries-related tasks

- Forecasting — most common timeseries-related task predicting what will happen next in a series.
  - forecast revenue a few months in advance
  - forecast the weather a few days in advance.

Common timeseries-related tasks

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

Common timeseries-related tasks

- Event detection—Identify the occurrence of a specific expected event within a continuous data stream.
  - For instance: hotword detection like “Ok Google” or “Hey Alexa.”

Common timeseries-related tasks

- Anomaly detection—Detect anything unusual happening within a continuous datastream.
  - Unusual readings on a manufacturing line (unsupervised)

A temperature-forecasting example

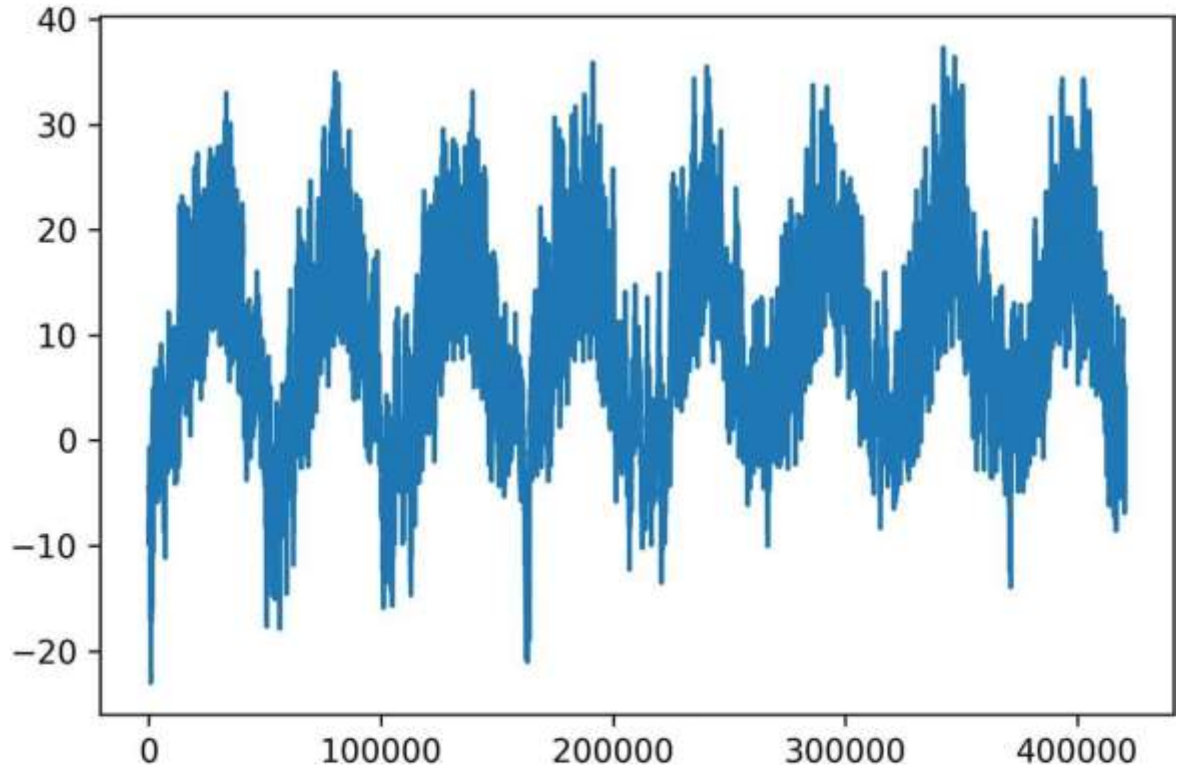
- Predicting the temperature 24 hours in the future,
  - given a timeseries of hourly measurements of quantities such as atmospheric pressure and humidity,
  - recorded over the recent past by a set of sensors on the roof of a building.

About the Timeseries Dataset

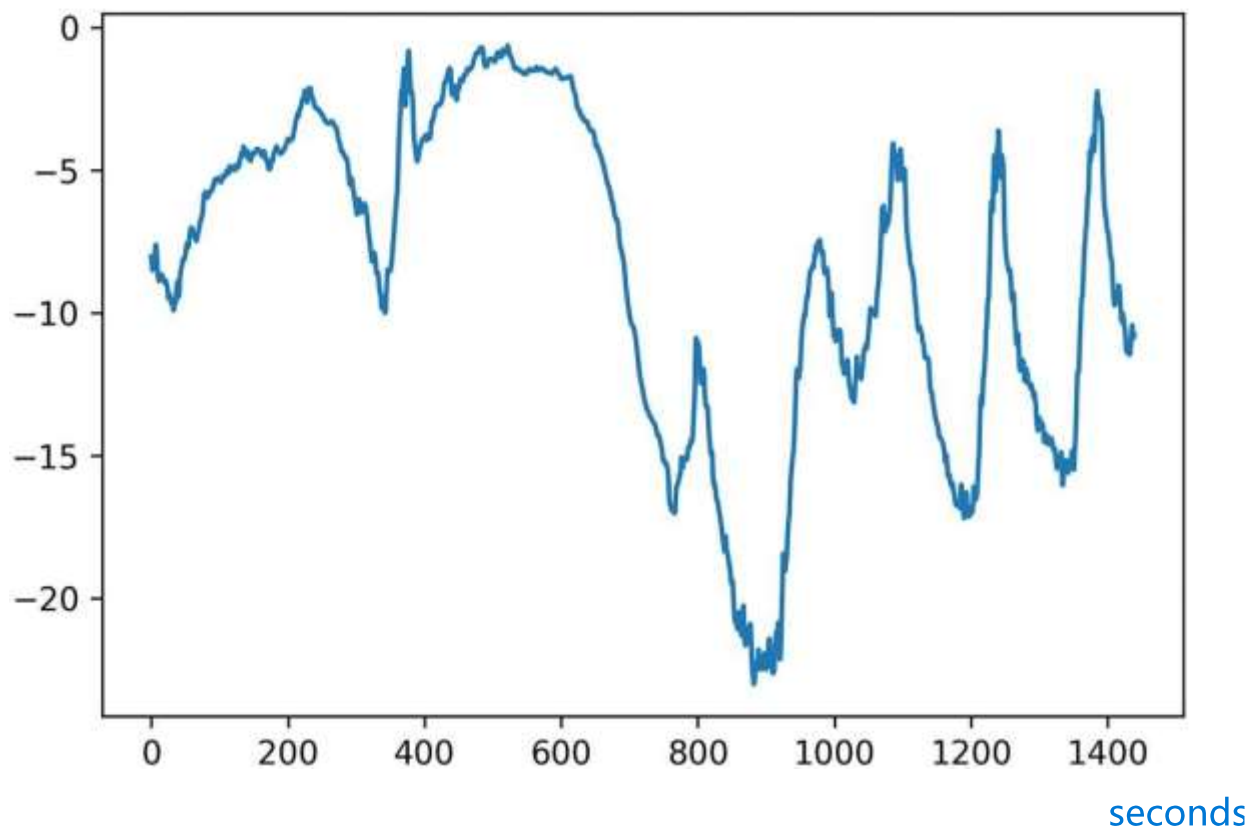
- We’ll work with a weather timeseries dataset recorded at the weather station at the Max Planck Institute for Biogeochemistry in Jena, Germany.
- In this dataset, 14 different quantities (such as temperature, pressure, humidity, wind direction, and so on) were recorded every 10 minutes over several years.

```
!wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
!unzip jena_climate_2009_2016.csv.zip
```

Temperature over the temporal range (ºC)



Temperature over the full temporal range of the dataset (ºC)



Temperature over the first 10 days of the dataset (ºC)

- Periodicity over multiple timescales is an important and very common property of timeseries data.
- Whether you’re looking at the weather, mall parking occupancy, traffic to a website, sales of a grocery store, or steps logged in a fitness tracker, you’ll see daily cycles and yearly cycles.

Experimental setup

- We’ll use the first 50% of the data for training, the following 25% for validation, and the last 25% for testing.
- When working with timeseries data, it’s important to use validation and test data that is more recent than the training data, because you’re trying to **predict the future given the past**, and your **validation/test splits should reflect that**.

Preparing the data

We'll use the following parameter values:

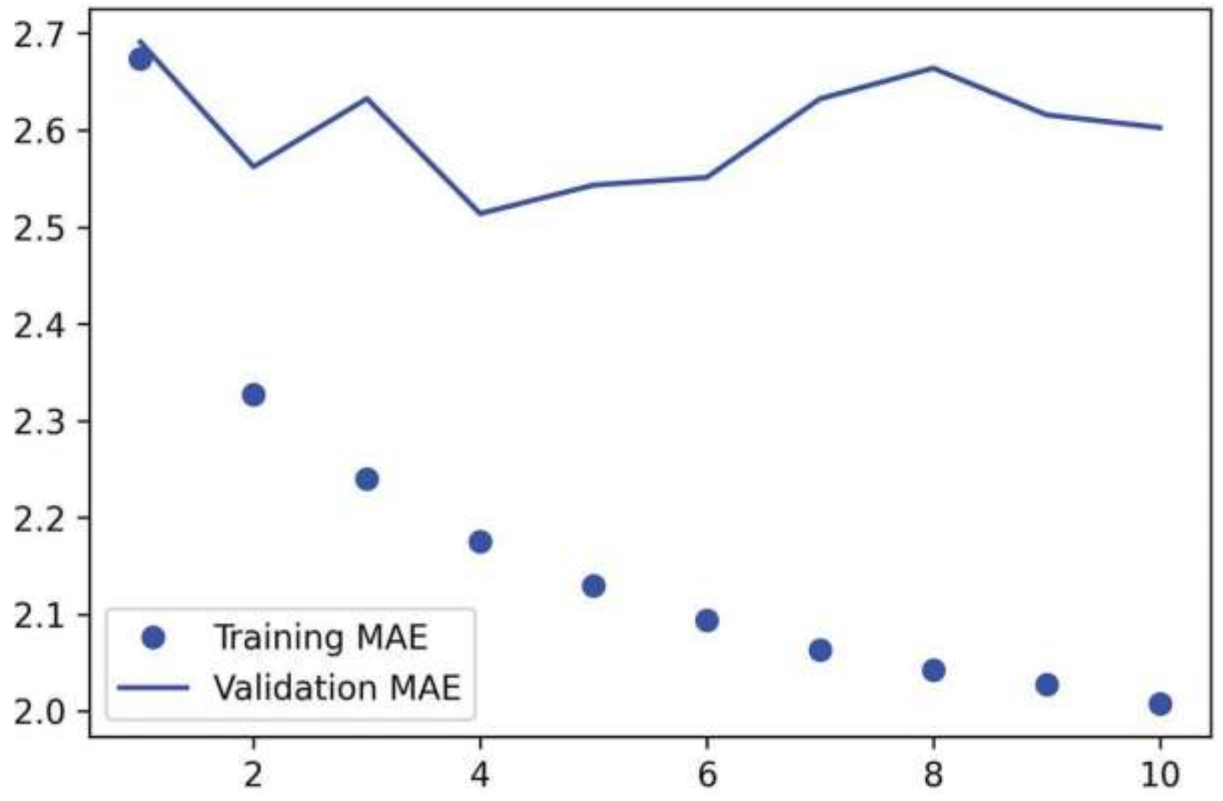
- sampling_rate = 6—Observations will be sampled at one data point per hour: we will only keep one data point out of 6.
- sequence_length = 120—Observations will go back 5 days (120 hours).
- delay = sampling_rate * (sequence_length + 24 - 1)—The target for a sequence will be the temperature 24 hours after the end of the sequence.
- **Problem**: Given data covering the previous five days and sampled once per hour, can we predict the temperature in 24 hours?

Let’s try a basic machine learning model

- It’s useful to establish a common-sense baseline (such as small, densely connected networks) before looking into complicated and computationally expensive models such as RNNs.

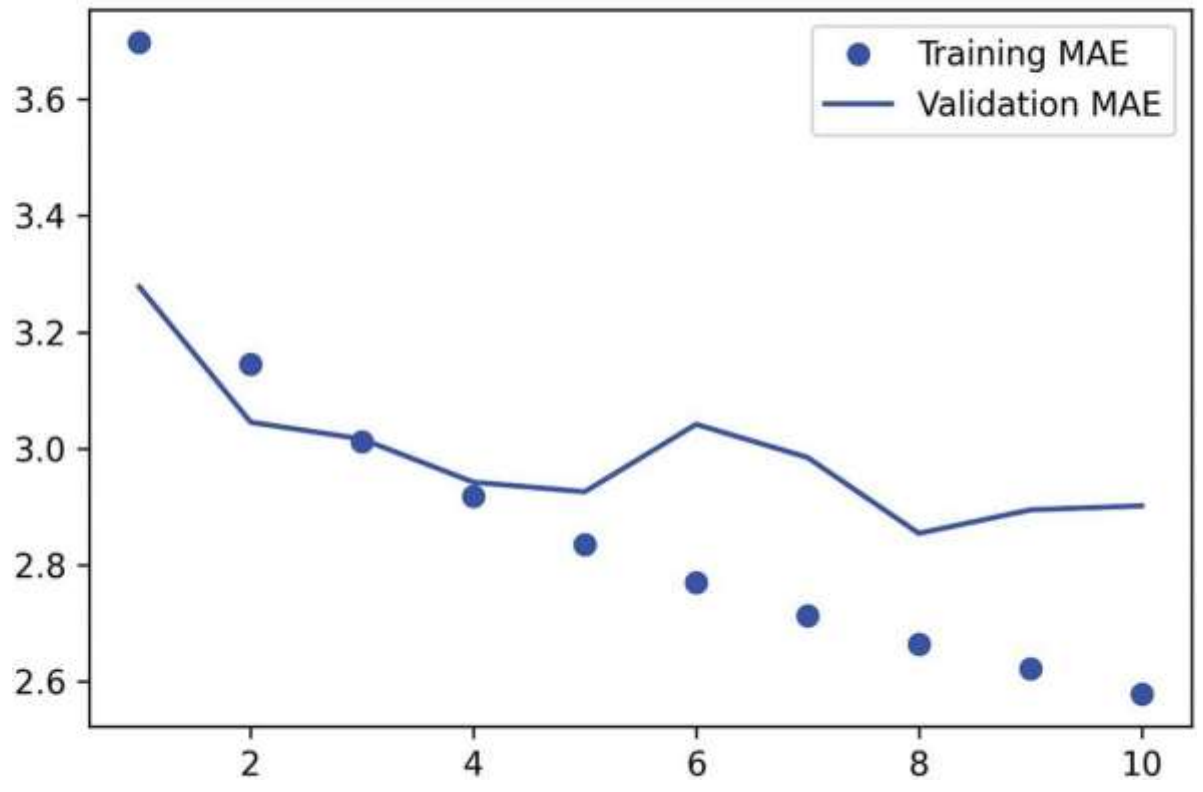
Training and validation MAE on the forecasting task with a simple, densely connected network

- This ML baseline achieves a validation MAE of 2.5 degrees Celsius.



Training and validation MAE on the forecasting task with a 1D convnet

- This model performs even worse than the densely connected one, only achieving a validation MAE of about 2.9 degrees.
- Order in data matters—a lot. The recent past is far more informative for predicting the next day's temperature than data from five days ago.
- A 1D convnet is not able to leverage this fact. In particular, our max pooling and global average pooling layers are largely destroying order information.



A first recurrent baseline

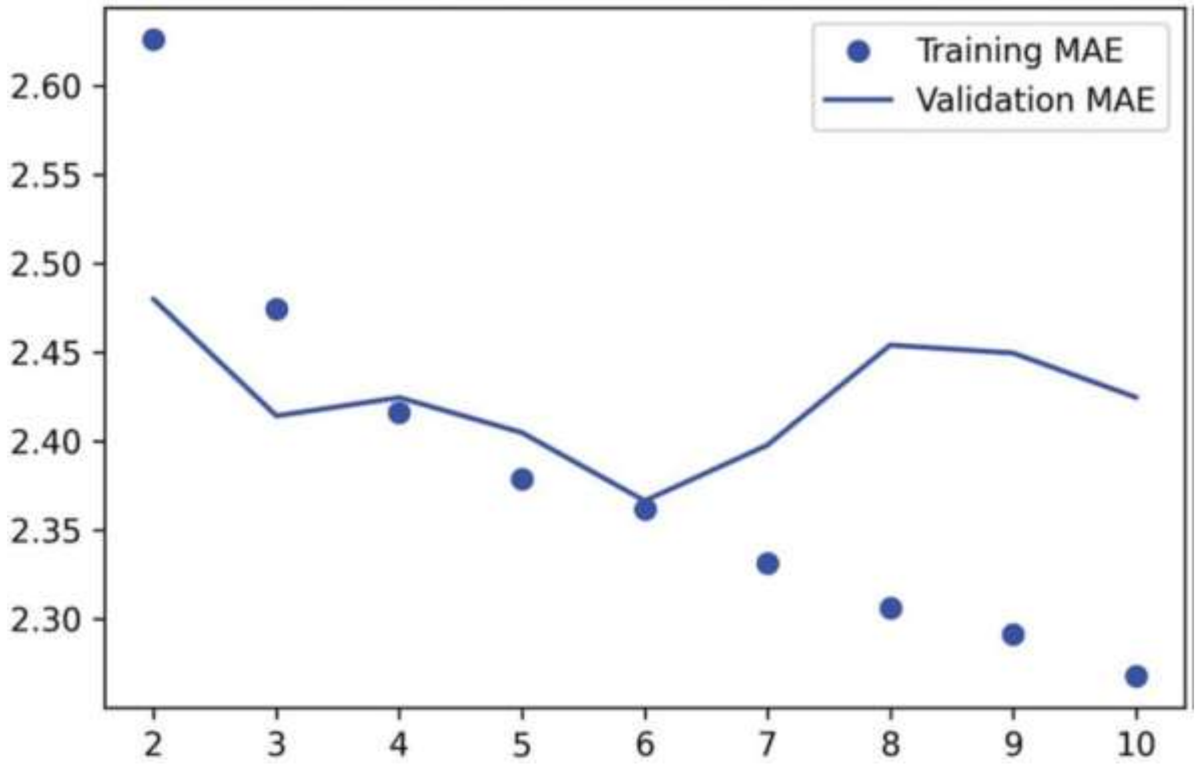
- Neither the fully connected approach nor the convolutional approach did well.
- The densely connected approach first flattened the timeseries, which removed the notion of time from the input data.
- The convolutional approach treated every segment of the data in the same way, even applying pooling, which destroyed order information.
- Our data as what it is: a sequence, where causality and order matter.

A first recurrent baseline

- There's a family of neural network architectures designed specifically for this use case: Recurrent Neural Networks (RNN).
- Among them, the Long Short Term Memory (LSTM) layer has long been very popular.

Training and validation MAE on forecasting task with an LSTM-based model

- Much better! We achieve a validation MAE as low as 2.36 degrees.



Recurrent Neural Network

Understanding recurrent neural networks

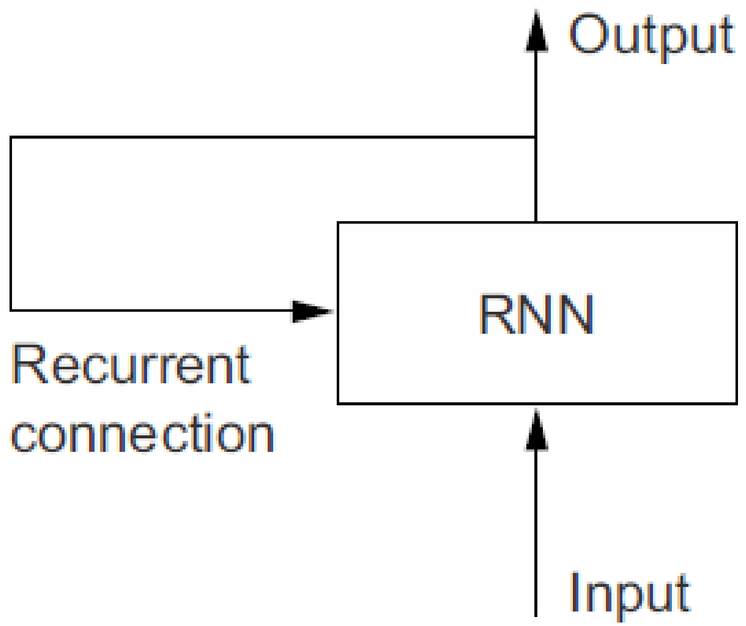
- A major characteristic of all neural networks you've seen so far, such as densely connected networks and convnets, is that they **have no memory**.
- Each input shown to them is processed independently, with no state kept between inputs.
- With such networks, in order to process a sequence or a temporal series of data points, you have to show the entire sequence to the network at once: turn it into a single data point.
  - For instance, we flattened our five days of data into a single large vector and processed it in one go. Such networks are called **feedforward networks**.
- In contrast, as you're reading the present sentence, you're processing it word by word—**while keeping memories of what came before**.

Understanding recurrent neural networks

- CNNs are commonly used in solving problems related to spatial data, such as images (e.g., facial recognition, image classification).
- RNNs are better suited to analyzing temporal, sequential data, such as text or videos (e.g., text translation, sentiment and speech analysis).

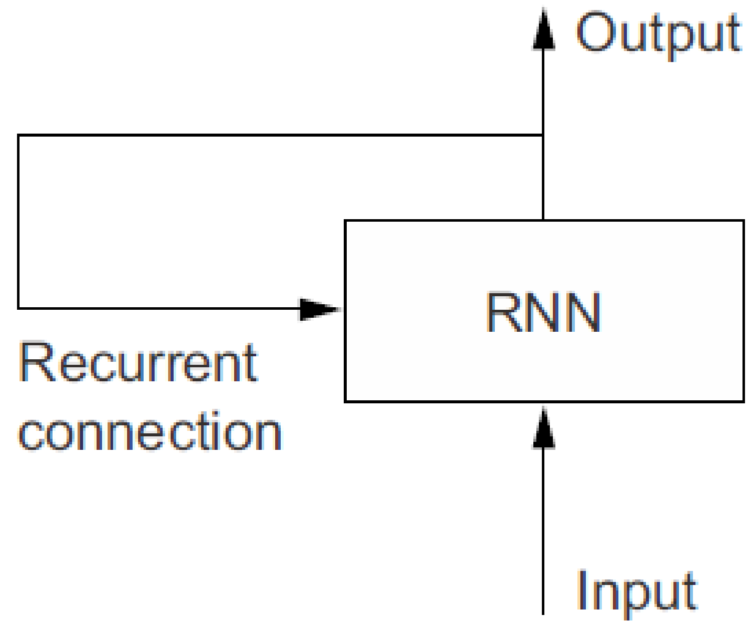
Basic Principle of RNN: A network with a Loop

- A recurrent neural network (RNN) processes sequences by iterating through the sequence elements and maintaining a state that contains information relative to what it has seen so far.
- In effect, an RNN is a type of neural network that has an internal loop

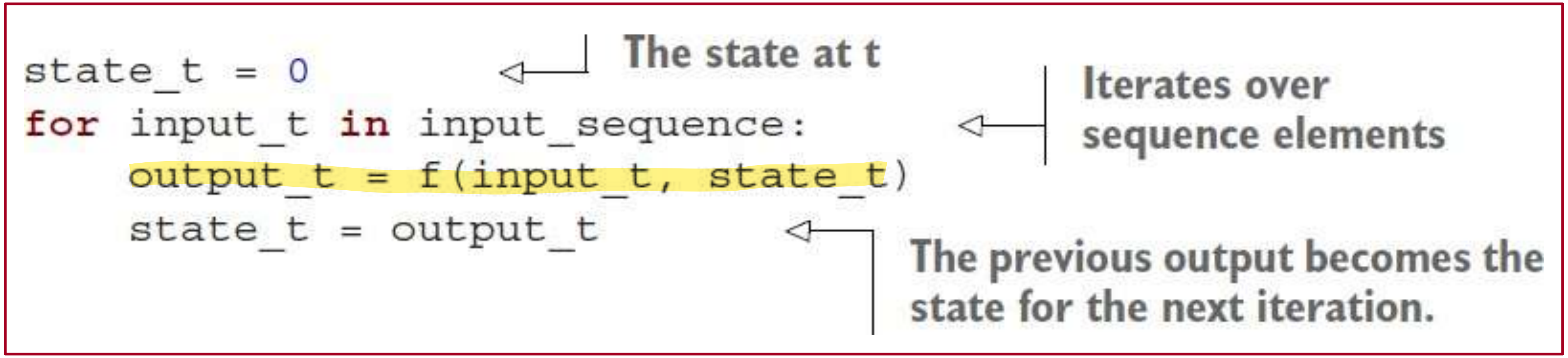


Basic Principle of RNN: A network with a Loop

- The state of the RNN is reset between processing two different, independent sequences (such as two samples in a batch)
- This data point is no longer processed in a single step; rather, the network internally loops over sequence elements.



Pseudocode RNN

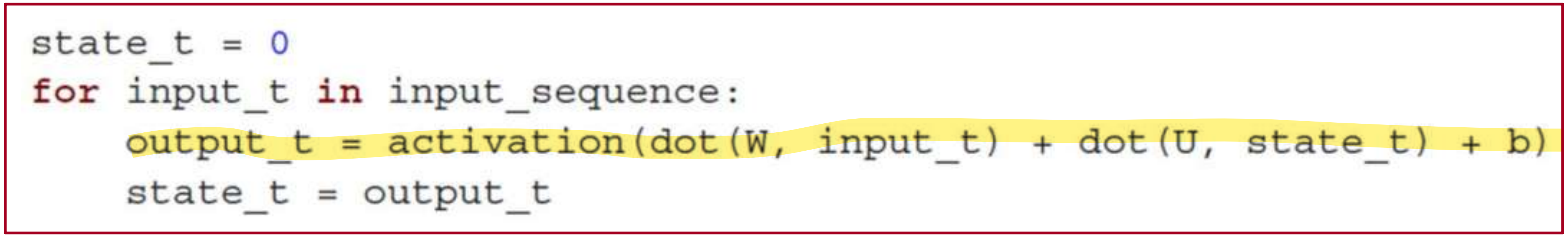


```
state_t = 0
for input_t in input_sequence:
    output_t = f(input_t, state_t)
    state_t = output_t
```

The state at t

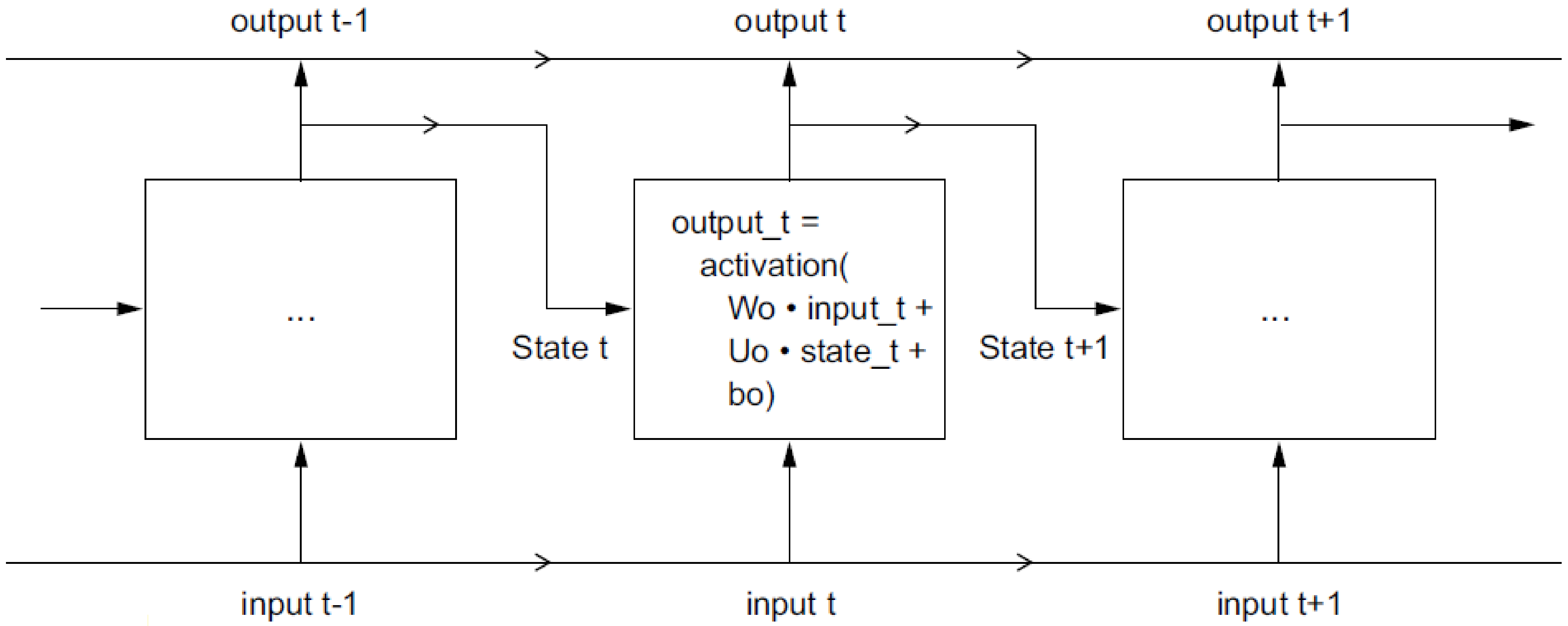
Iterates over sequence elements

The previous output becomes the state for the next iteration.



```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```



A simple RNN, unrolled over time

A NumPy Implementation of RNN

Input data: random noise for the sake of the example

Number of timesteps in the input sequence

Dimensionality of the input feature space

Dimensionality of the output feature space

Initial state: an all-zero vector

Creates random weight matrices

Stores this output in a list

```python
import numpy as np
timesteps = 100
input_features = 32
output_features = 64
inputs = np.random.random((timesteps, input_features))
state_t = np.zeros((output_features,))
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:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
    successive_outputs.append(output_t)
    state_t = output_t
final_output_sequence = np.stack(successive_outputs, axis=0)
```

Combines the input with the current state (the previous output) to obtain the current output. We use tanh to add non-linearity (we could use any other activation function).

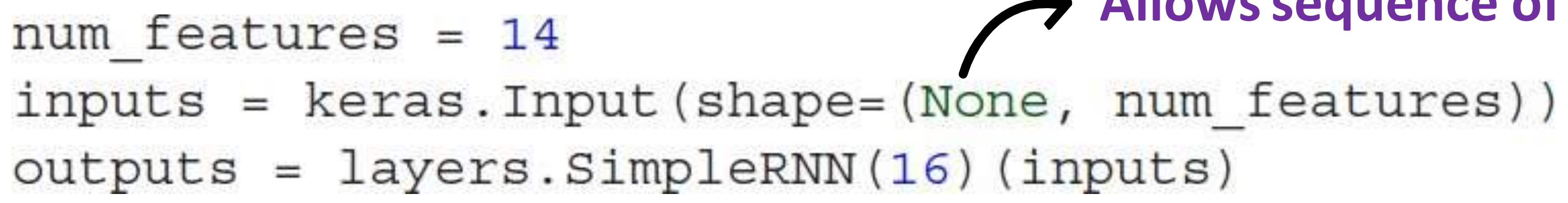
The final output is a rank-2 tensor of shape (timesteps, output_features).

input_t is a vector of shape (input_features,).

Updates the state of the network for the next timestep

A recurrent layer in Keras

- The process we just implemented in NumPy corresponds to an actual Keras layer—the `SimpleRNN` layer.
- One minor difference: `SimpleRNN` processes batches of sequences, like all other Keras layers, not a single sequence as in the NumPy example. This means it takes inputs of shape (batch_size, timesteps, input_features),



```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

Allows sequence of variable length

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
>>> print(outputs.shape)
(None, 16)
```

Recommended for sequence having same length

Note that return_sequences=False is the default.

An RNN layer returns only its last output step

```
>>> num_features = 14
>>> steps = 120
>>> inputs = keras.Input(shape=(steps, num_features))
>>> outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
>>> print(outputs.shape)
(120, 16)
```

An RNN layer returns its full output sequence

Stacking RNN Layers

- It’s useful to stack several recurrent layers one after the other in order to increase the representational power of a network

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```

Multi RNN Layer

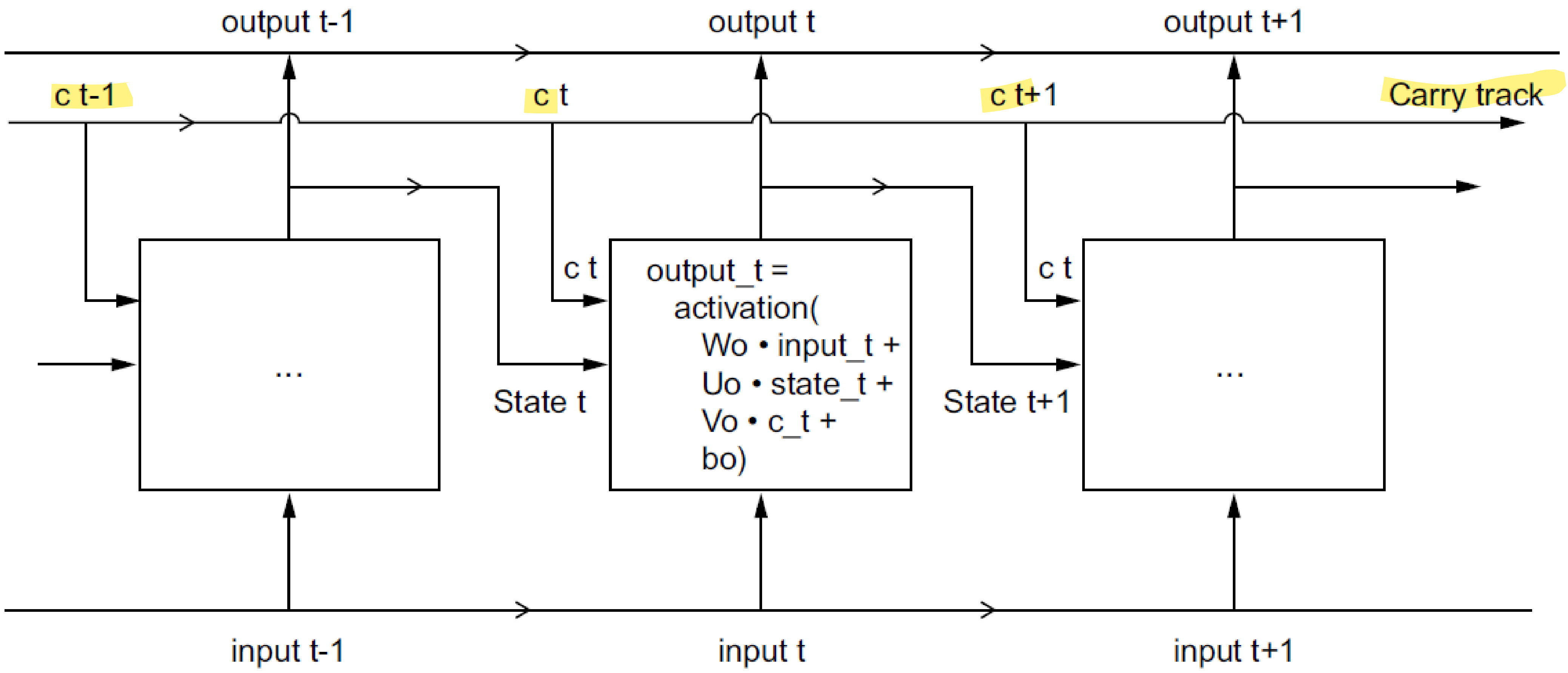
Lacking of a simple RNN

- Although *SimpleRNN* should theoretically be able to retain at time `t` information about inputs seen many timesteps before, such long-term dependencies prove impossible to learn in practice.
- This is due to the *vanishing gradient problem*, an effect that is similar to what is observed in feedforward networks that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable.

Long Short-Term Memory (LSTM) algorithm

- RNN has trouble learning long-term dependencies, which means they don’t comprehend relationships between data that are separated by several steps due to the vanishing gradient problem.
- When anticipating words, for example, we may require more context than simply one prior word.
- The LSTM algorithm, developed by Hochreiter and Schmidhuber in 1997, addresses the vanishing gradient problem.
- It saves information for later, thus preventing older signals from gradually vanishing during processing

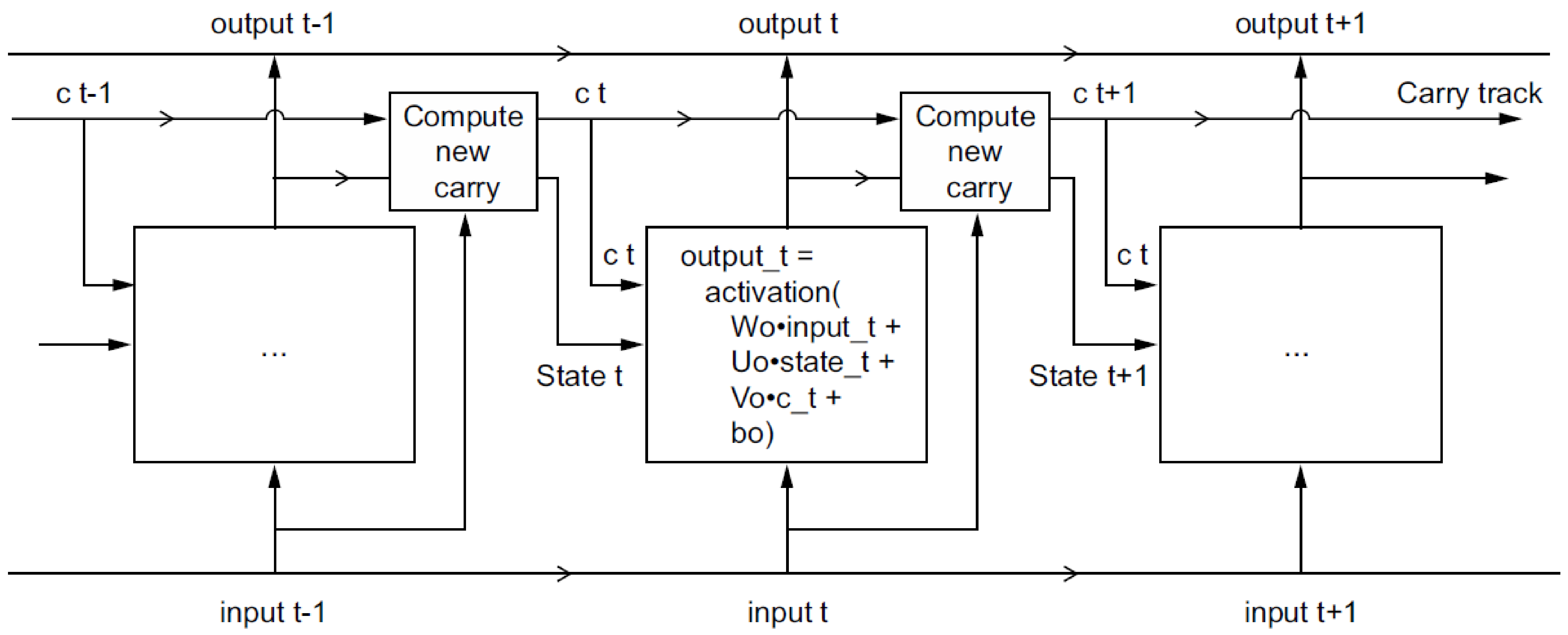
Going from a Simple RNN to an LSTM: adding a carry track



A Simple RNN to an LSTM…

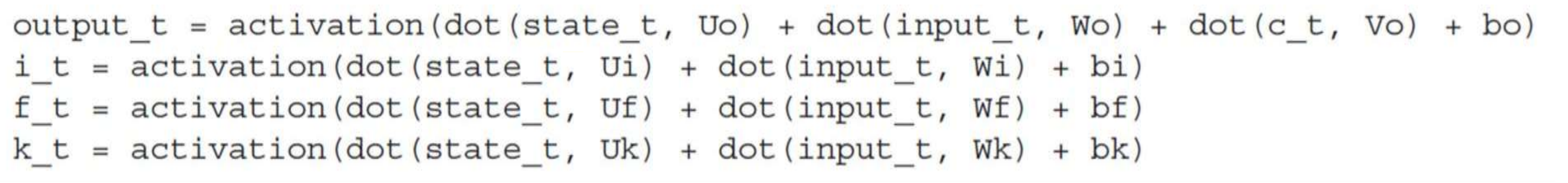
- An additional data flow that carries information across timesteps is added.
- Call its values at different timesteps c_t, where C stands for carry.
- This info is combined with the input connection and the recurrent connection
- Conceptually, the carry dataflow is a way to modulate the next output and the next state.

Anatomy of an LSTM

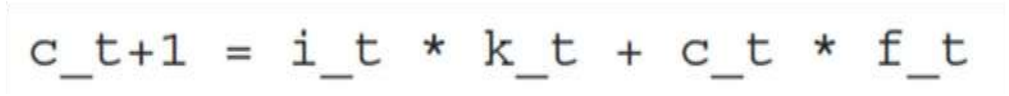


Anatomy of an LSTM

- Computation of the next carry dataflow involves three distinct transformations.

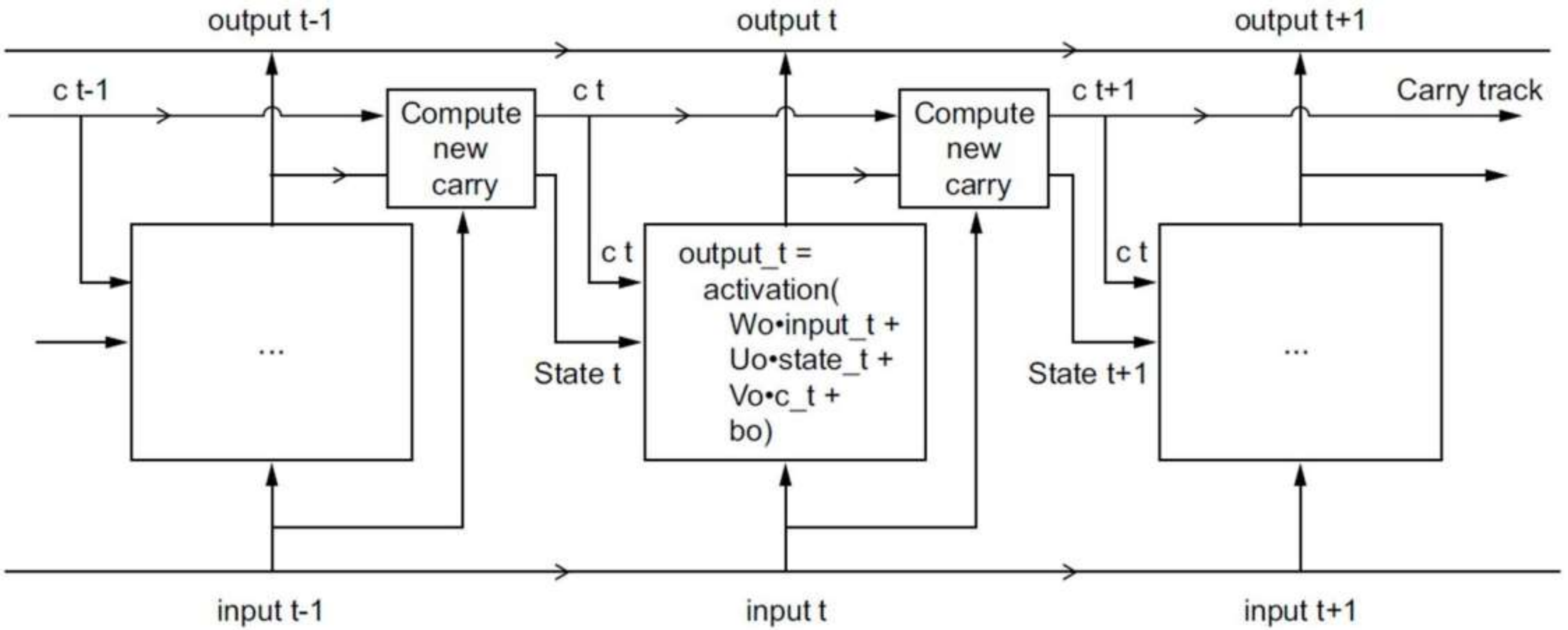


```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(c_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```



```
c_t+1 = i_t * k_t + c_t * f_t
```

- Activation for i_t and f_t is a **sigmoid** function.
- It is **tanh** for k_t. Due to the tanh function, the value of new information will be between -1 and 1. If the value is negative, the information is subtracted from the carry state, and if it is positive, the information is added to the carry state at the current timestamp.



```
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)
```

```
c_t+1 = i_t * k_t + c_t * f_t
```

Anatomy of an LSTM

- Multiplying c_t and f_t is a way to deliberately forget irrelevant information in the carry dataflow, usually referred to as **forget gate,** by setting f_t = 0.

  Ex. ~~Kamal~~ is my friend. Bashar is a nice person.

  Bashar is a nice person. He is brave as well.

- i_t and k_t provide information about the present, updating the carry track with new information, usually referred to as **input gate.**
- Here the recurrent state (state_t) is known as **Short term memory**, and the carry state (c_t) is known as **Long term memory**.
- The **output gate** takes the current input, the previous short term memory and newly computed long term memory to produce the output (new short term memory for the next time stamp).

- In summary, just keep in mind what the LSTM cell is meant to do:
    - Allow past information to be reinjected at a later time, thus fighting the vanishing-gradient problem.

Advanced use of recurrent neural networks

We'll cover the following:

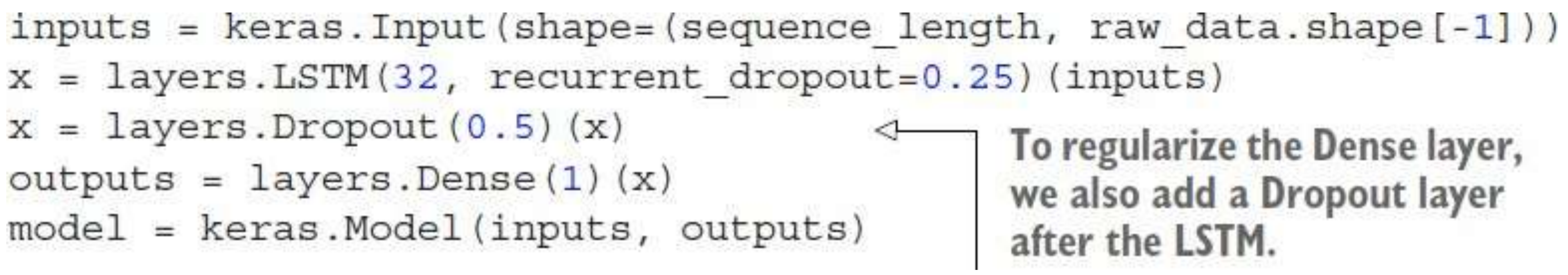
- **Recurrent dropout**—This is a variant of dropout, used to fight overfitting in recurrent layers.
- **Stacking recurrent layers**—This increases the representational power of the model (at the cost of higher computational loads).
- **Bidirectional recurrent layers**—These present the same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues.

Using recurrent dropout to fight overfitting

- It has long been known that applying dropout before a recurrent layer hinders learning rather than helping with regularization.
- In 2016, Yarin Gal has shown that the same dropout mask (the same pattern of dropped units) should be applied at every timestep, instead of using a dropout mask that varies randomly from timestep to timestep.
- What’s more, a temporally constant dropout mask should be applied to the inner recurrent activations of the LSTM layer (a recurrent dropout mask).
- Using the same dropout mask at every timestep allows the network to properly propagate its learning error through time.

Using recurrent dropout to fight overfitting

- Yarin Gal did his research using Keras and helped build this mechanism directly into Keras recurrent layers.



```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

To regularize the Dense layer, we also add a Dropout layer after the LSTM.

- Here, **dropout**, specifies the dropout rate for input units of the layer and **recurrent_dropout**, specifies the dropout rate of the recurrent units.

Stacking recurrent layers

- it’s generally a good idea to increase the capacity of your model until overfitting becomes the primary obstacle
- Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers.
    - for instance, not too long ago the Google Translate algorithm was powered by a stack of seven large LSTM layers
- A stack of two dropout-regularized GRU layers is tried here.
- GRU (Gated Recurrent Unit) is very similar to LSTM—a slightly simpler, streamlined version of the LSTM architecture.

LSTM vs GRU

LSTM and GRU are both variants of RNN that are used to resolve the vanishing gradient issue of the RNN, but they have some differences, which are:

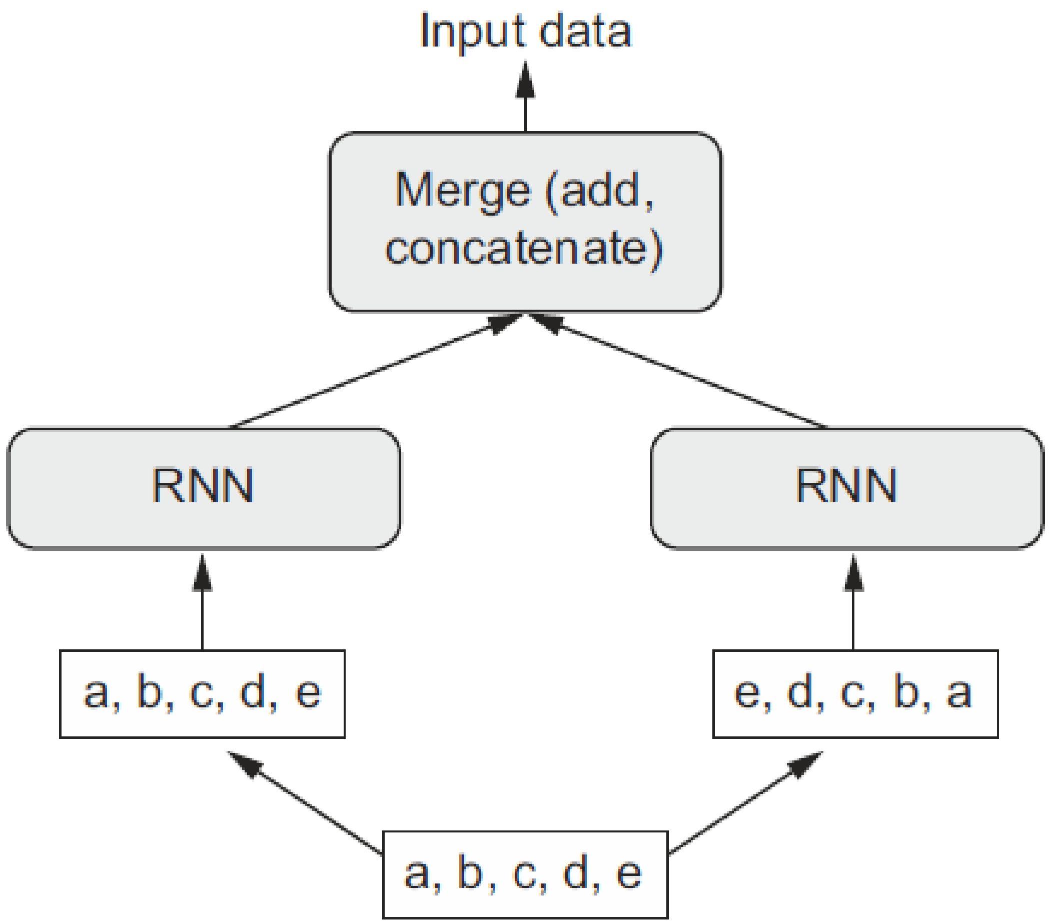
- LSTM uses three gates (forget, input and output) in computing, whereas GRU uses only two gates (reset and update).
- GRUs are generally simpler and faster than LSTM.
- LSTMs are preferred for large datasets, whereas for small datasets GRUs are preferred.

Using bidirectional RNNs

- Bidirectional RNN is a common RNN variant that can offer greater performance than a regular RNN on certain tasks.
- RNN processes the timesteps of their input sequences in order, and shuffling or reversing the timesteps can completely change the results.
- A bidirectional RNN exploits the order sensitivity of RNNs: it uses two regular RNNs (GRU/ LSTM), each of which processes the input sequence in one direction (chronologically and antichronologically), and then merges their representations.
- By processing a sequence both ways, a bidirectional RNN can catch patterns that may be overlooked by a unidirectional RNN.

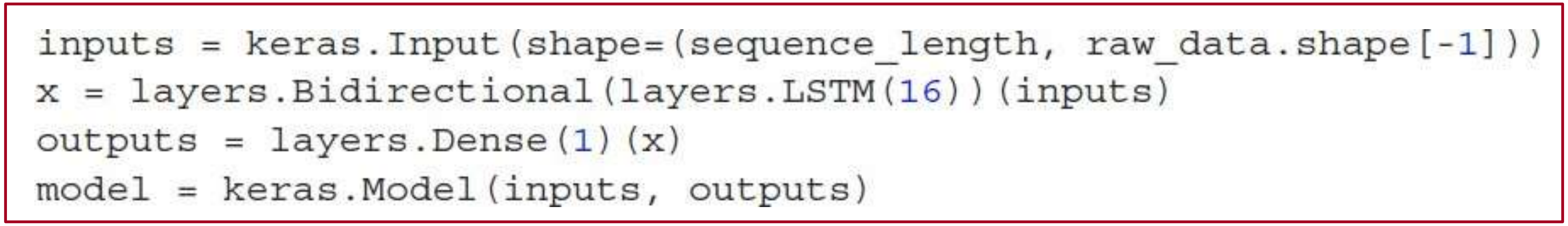
How a bidirectional RNN layer works

- A bidirectional RNN exploits this idea to improve on the performance of chronological-order RNNs.



Bidirectional RNN in Keras

- To instantiate a bidirectional RNN in Keras, you use the Bidirectional layer, which creates two instances and uses one instance for processing the input sequences in chronological order and the other instance for processing the input sequences in reversed order.



```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

- This bidirectional setting doesn’t perform as well as the plain LSTM layer. However, **bidirectional RNNs are a great fit for text data**, or any other kind of data where order matters, yet where which order you use doesn’t matter.

Other things to try

There are many other things you could try in order to improve performance on the temperature-forecasting problem:

- Adjust the number of units in each recurrent layer in the stacked setup, as well as the amount of dropout. The current choices are largely arbitrary.
- Adjust the learning rate used by the RMSprop optimizer, or try a different optimizer.
- Try using a stack of Dense layers as the regressor on top of the recurrent layer, instead of a single Dense layer.
- Improve the input to the model: try using longer or shorter sequences or a different sampling rate, or start doing feature engineering.

Summary

- When approaching a new problem, it’s good to first establish **common-sense baselines** for your metric of choice, otherwise you can’t tell whether you’re making real progress.
- **Try simple models** before expensive ones, to make sure the additional expense is justified. Sometimes a simple model will turn out to be your best option.
- For **data where ordering matters** (timeseries data), recurrent networks are a great fit and easily outperform models that first flatten the temporal data.

Summary

- The two essential RNN layers available in Keras are **the LSTM layer and the GRU layer**.
- To use **dropout with recurrent networks**, you should use a time-constant dropout mask and recurrent dropout mask. These are built into Keras recurrent layers.
- **Stacked RNNs** provide more representational power than a single RNN layer.
- Although more expensive, stacked RNNs offer clear gains on **complex problems** (such as machine translation).