Time Series

CSE 4311 – Neural Networks and Deep Learning

Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

Sequential Data

- **Sequential data**, as the name implies, are **sequences**.
- What is the difference between a **sequence** and a **set**?
- A set is a collection of elements, with no inherent order.
  - {1,2,3} = {3,1,2} = {2,1,3}, the order in which we write the elements does not matter.
- A sequence $\boldsymbol{X}$ is a set of elements, together with a **total order** imposed on those elements.
  - A **total order** describes, for any two elements $\boldsymbol{x_1}, \boldsymbol{x_2}$, which of them comes before and which comes after.
  - Sequences (1,2,3), (3,1,2), (2,1,3) are all different from each other, because the order of elements matters.

Time Series

- A **time series** is a **sequence** of vectors.
- Each vector can be thought of as an observation, or measurement, that corresponds to one specific moment in time.
- Examples:
  - Stock market prices (for a single stock, or for multiple stocks).
  - Heart rate of a patient over time.
  - Position of one or multiple people/cars/airplanes over time.
  - Speech: represented as a sequence of audio measurements at discrete time steps.
  - A musical melody: represented as a sequence of pairs (note, duration).

Dimensionality of Vectors

- In the simplest case, a time series is just a sequence of numbers.
- An example is the sequence of daily high temperatures (in Fahrenheit) in Arlington, from January 1 to January 4, 2022.
  - We get the time series (74, 40, 54, 54).
  - This time series has length 4.
- In general, a time series is a sequence of vectors.
  - All vectors in the time series must have the same dimensionality.
- For example, we can take the previous sequence of daily high temperatures, and include the daily low temperature as well.
  - We get the time series ((74,24), (40,19), (55, 25), (64,33)).
  - It has length 4, and every element is a 2D vector.

Time Series Terminology

- We can use the term “sequence” to refer to a time series.
  - This is correct usage. Any time series is a sequence. The reverse is not true, there are sequences (for example, strings) that are NOT time series.
- The **length** of a sequence is the number of elements in the sequence.
  - For example, sequence ((74,24), (40,19), (55, 25), (64,33)) has length 4.
- A feature refers to a specific dimension of the vectors that the time series contains.
  - For example, in an earlier slide we said that the first feature in sequence ((74,24), (40,19), (55, 25), (64,33)) is the daily high temperature. The second feature is the daily low temperature.
- We can refer to an element of a time series as a “feature vector”.

Strings and Time Series

- Strings are an example of sequential data: a string is a sequence of characters from some alphabet.
- Strings are sequential: the order of the characters matters.
  - **Strings** “mile” and “lime” are not equal.
  - Compare to **sets** {‘m’, ‘i’, ‘l’, ‘e’} and {‘l’, ‘i’, ‘m’, ‘e’}, which are equal.
- Strings are not time series, because their elements are characters (symbols from a finite and discrete alphabet) and not vectors.
- However, we can easily convert a string dataset to a time series dataset.
  - We map each character to a one-hot vector.
  - The dimensionality of these one-hot vectors is equal to the number of letters in the alphabet.

Text and Time Series

- Text is another example of sequential data: a piece of text data can be seen as a sequence of letters, or as a sequence of words.
- Using one-hot vectors we can convert a text dataset to a time series dataset.
  - We can map each letter to a one-hot vector, or each word to a one-hot vector. Mapping words is more common.
  - There are other methods as well for converting a piece of text to a time series.
  - We will cover this topic in detail in a few weeks.

Speech and Time Series

- The difference between speech and text is that speech data comes from audio, whereas text data is written.
- We will not cover speech recognition this semester.

Types of Time Series Analysis

- Forecasting future elements (commodity prices, temperature, …).
- Classification: assigning a class label to a time series.
  - Song recognition: sing a tune to your phone, have it recognize the song.
  - Activity recognition: observe a human activity, recognize the type, such as walking, cooking, talking on the phone, etc.
- Spotting: identifying the start and end points of an event of interest.
  - Find in an audio recording the time interval when a specific phrase is said.
  - Find in a video the time interval when a specific activity occurs.
- Sequence-to-sequence translation.
  - Take an audio recording as input, produce text as output.
  - Take English text as input, produce a Spanish translation as output.

Time Series Example: Signs

- 0.5 to 2 million users of American Sign Language (ASL) in the US.
- Different regions in the world use different sign languages.
  - For example, British Sign Language (BSL) is different than American Sign Language.
- These languages have vocabularies of thousands of signs.

Example: The ASL Sign for "again"

Example: The ASL Sign for "bicycle"

Representing Signs as Time Series

- A sign is a video (or part of a video).
- A video is a sequence of images.
- At every frame i, we extract a **feature vector** $\boldsymbol{x}_i$.
  - How should we define the feature vector? This is a (challenging) computer vision problem.
  - A naïve approach you can start with, if you have a good hand detector, is to have a 4D feature vector, that contains the 2D pixel locations of the centers of the two hands.
- Then, the entire sign is represented as time series $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots \boldsymbol{x}_D)$.
  - D is the length of the sign video, measured in frames.
  - Every $\boldsymbol{x}_t$ is a 4-dimensional feature vector, containing the pixel locations of the centers of the two hands.

Example: Hand Positions as Features

- Here we see how hand locations are used to define feature vectors.

Frame 1:
Right hand center:
$(247,236)$

Left hand center:
$(281,365)$

Feature vector $\boldsymbol{x}_1$:
$(247,236,281,365)$

Example: Hand Positions as Features

- Here we see how hand locations are used to define feature vectors.

Frame 2:
Right hand center:
$(242,236)$

Left hand center:
$(279,367)$

Feature vector $\boldsymbol{x}_2$:
$(242,236,279,367)$

Example: Hand Positions as Features

- Here we see how hand locations are used to define feature vectors.

Frame 3:
Right hand center:
$(235,237)$

Left hand center:
$(277,368)$

Feature vector $\boldsymbol{x}_3$:
$(235,237,277,368)$

Example: The Jena Climate Dataset

- The Jena Climate dataset is a weather time series dataset.
  - Publicly available at: https://www.kaggle.com/mnassrib/jena-climate
- The data was recorded at the Weather Station of the Max Planck Institute for Biogeochemistry in Jena, Germany.
- 8 years of data: January 1, 2009 to December 31, 2016.
- A feature vector was recorded every 10 minutes during those eight years.
- Each feature vector is 14-dimensional.
- These are some of the recorded features:
  - Air temperature.
  - Atmospheric pressure.
  - Humidity.
  - Wind direction…

Plan for the Next Few Weeks

- We will work with the Jena Climate dataset and apply various methods for time series analysis.
- In working with this dataset, we will follow Chapter 10 of the textbook “Deep Learning with Python”, Second Edition, 2021, by François Chollet.
- The learning task we will address is forecasting:
  - Given data from the last five days, the goal is to predict the temperature exactly 24 hours from now.
- We first will try some simple methods, based on what we already know.
- Then we will introduce **Recurrent Neural Networks** (RNNs), which are explicitly designed to process time series data.
  - They are based on a new type of layer, called “Recurrent Layer”.

Plan for the Next Few Weeks

- After that, we will work with text data.
  - One task: recognizing if an IMDB movie review is positive or negative.
  - Another task: translating from English to Spanish.
- On the topic of working with text data, we will follow Chapter 11 of the textbook “Deep Learning with Python”, Second Edition, 2021, by François Chollet.
- We will study various methods for converting text data to time series.
- We will apply and evaluate RNNs on our two tasks.
- We will then introduce and apply yet another type of neural networks, called Transformers.

Jena Climate Dataset: A Closer Look

- You can download the dataset from:

https://www.kaggle.com/mnassrib/jena-climate

- The dataset is saved in a CSV file with 15 columns:
  - 0: Date and Time
  - 1: Atmospheric pressure, in millibars.
  - 2: Temperature in Celsius.
  - 3: Temperature in Kelvin.
  - 4: Temperature in Celsius relative to humidity. According to the dataset web page, “Dew Point is a measure of the absolute amount of water in the air, the DP is the temperature at which the air cannot hold all the moisture in it and water condenses.”
  - 5: Relative humidity.
  - 6: Saturation vapor pressure.
  - 7: Vapor pressure.

Jena Climate Dataset: A Closer Look

- The dataset is saved in a CSV file with 15 columns:
  - 8: Vapor pressure deficit.
  - 9: Specific humidity.
  - 10: Water vapor concentration.
  - 11: Airtight.
  - 12: Wind speed.
  - 13: Maximum wind speed.
  - 14: Wind direction in degrees.

- As you see, some of these features (like “saturation vapor pressure”, “airtight”) are pretty esoteric to non-specialists, whereas others (like temperature, wind speed) have a meaning that we can all understand.

Reading the Data

```
fname = "jena_climate_2009_2016.csv"
with open(fname) as f:
    data = f.read()
lines = data.split("\n")
lines = lines[1:]   # The first line in the file is header information

temperature = np.zeros((len(lines),))
raw_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(",")[1:]]
    temperature[i] = values[1]
    raw_data[i] = values
```

- This code creates two time series: **temperature**, and **raw_data**.

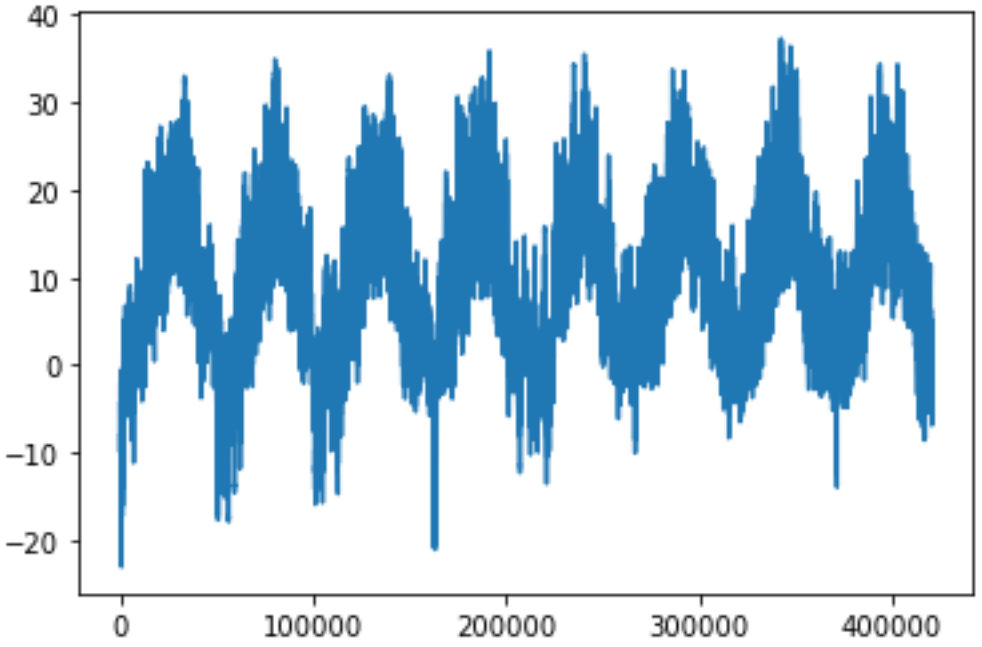
Reading the Data

```
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(",")[1:]]
    temperature[i] = values[1]
    raw_data[i] = values
```

- Variable **temperature** is a 1D time series.
  - **temperature[i]** is the i-th temperature observation, in Celsius.
- Variable **raw_data** is a 14-dimensional time series.
  - **raw_data[i]** is a 14-dimensional vector.
  - From the original 15 columns of the dataset, we exclude column 0, which was the date and time.

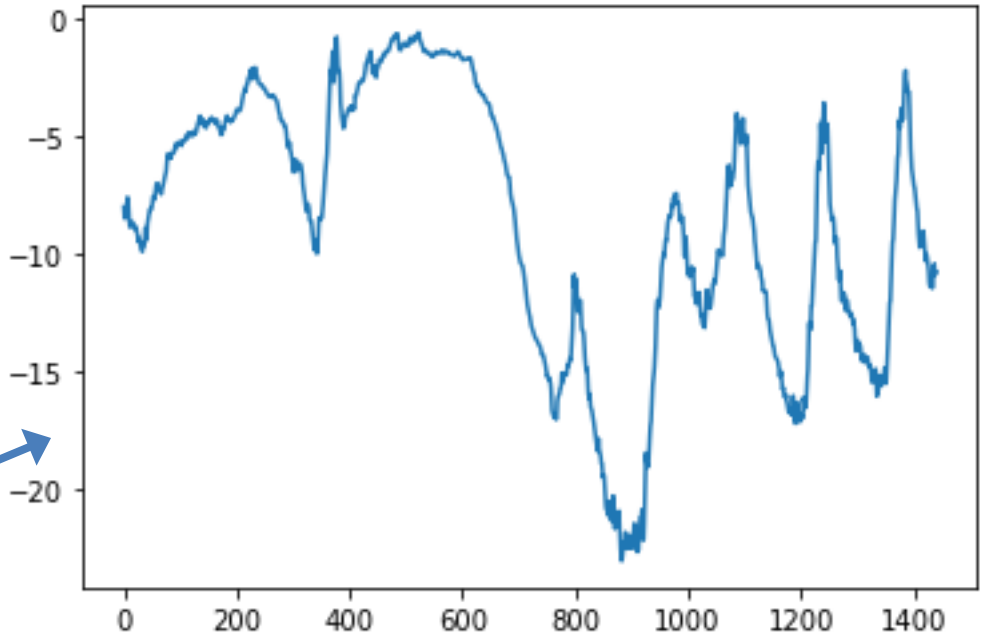
Visualizing the Data

plt.plot(range(0, len(temperature)), temperature)



plots all 420,451 values in the temperature array (one observation every ten minutes, for eight years).

plt.plot(range(0, 1440), temperature[:1440])



plots the first 1440 values, corresponding to the first 10 days.

Training, Validation, Test Data

- Forecasting task: given data from the last five days, the goal is to predict the temperature exactly 24 hours from now.
- We separate our data into:
  - training data (first 50% of the data).
  - validation data (subsequent 25% of the data).
  - test data (last 25% of the data).

```
num_train_samples = int(0.5 * len(raw_data))
num_val_samples = int(0.25 * len(raw_data))
num_test_samples = len(raw_data) - num_train_samples - num_val_samples
```

```
print("num_train_samples:", num_train_samples)
print("num_val_samples:", num_val_samples)
print("num_test_samples:", num_test_samples)
```

Output:

```
num_train_samples: 210225
num_val_samples: 105112
num_test_samples: 105114
```

Normalizing the Data

- We have seen before that we should normalize the data, so that values are not too big or too small.
- One approach we have used is to scale each dimension to have values between 0 and 1.
- Another popular approach is to scale each dimension so that the mean value along that dimension is 0, and the standard deviation is 1.
  - This is the approach we will use here.

Normalizing the Data

```
(number, dimensions) = raw_data.shape
normalized_data = np.zeros((number, dimensions))

for d in range (0, dimensions):
    feature_values = raw_data[0:num_train_samples,d]
    m = np.mean(feature_values)
    s = np.std(feature_values)
    normalized_data[:,d] = (raw_data[:,d] - m) / s
```

- This piece of code does the normalization we want.
- For every dimension (i.e., for every attribute of the data), we subtract the mean, and then we divide by the std.
- After that, each dimension has mean 0 and std 1 (at least for the training data).

Normalizing the Data

```
(number, dimensions) = raw_data.shape
normalized_data = np.zeros((number, dimensions))

for d in range (0, dimensions):
    feature_values = raw_data[0:num_train_samples,d]
    m = np.mean(feature_values)
    s = np.std(feature_values)
    normalized_data[:,d] = (raw_data[:,d] - m) / s
```

- Why do we only compute the mean and std on the training data?
- Why not use all columns, like: feature_values = raw_data[:, d]

Normalizing the Data

```
(number, dimensions) = raw_data.shape
normalized_data = np.zeros((number, dimensions))

for d in range (0, dimensions):
    feature_values = raw_data[0:num_train_samples,d]
    m = np.mean(feature_values)
    s = np.std(feature_values)
    normalized_data[:,d] = (raw_data[:,d] - m) / s
```

- Why do we only compute the mean and std on the training data?
- Why not use all columns, like: feature_values = raw_data[:, d]
- Because, in the real world, when we normalize the training data, we have no idea what test data the system will be applied to.

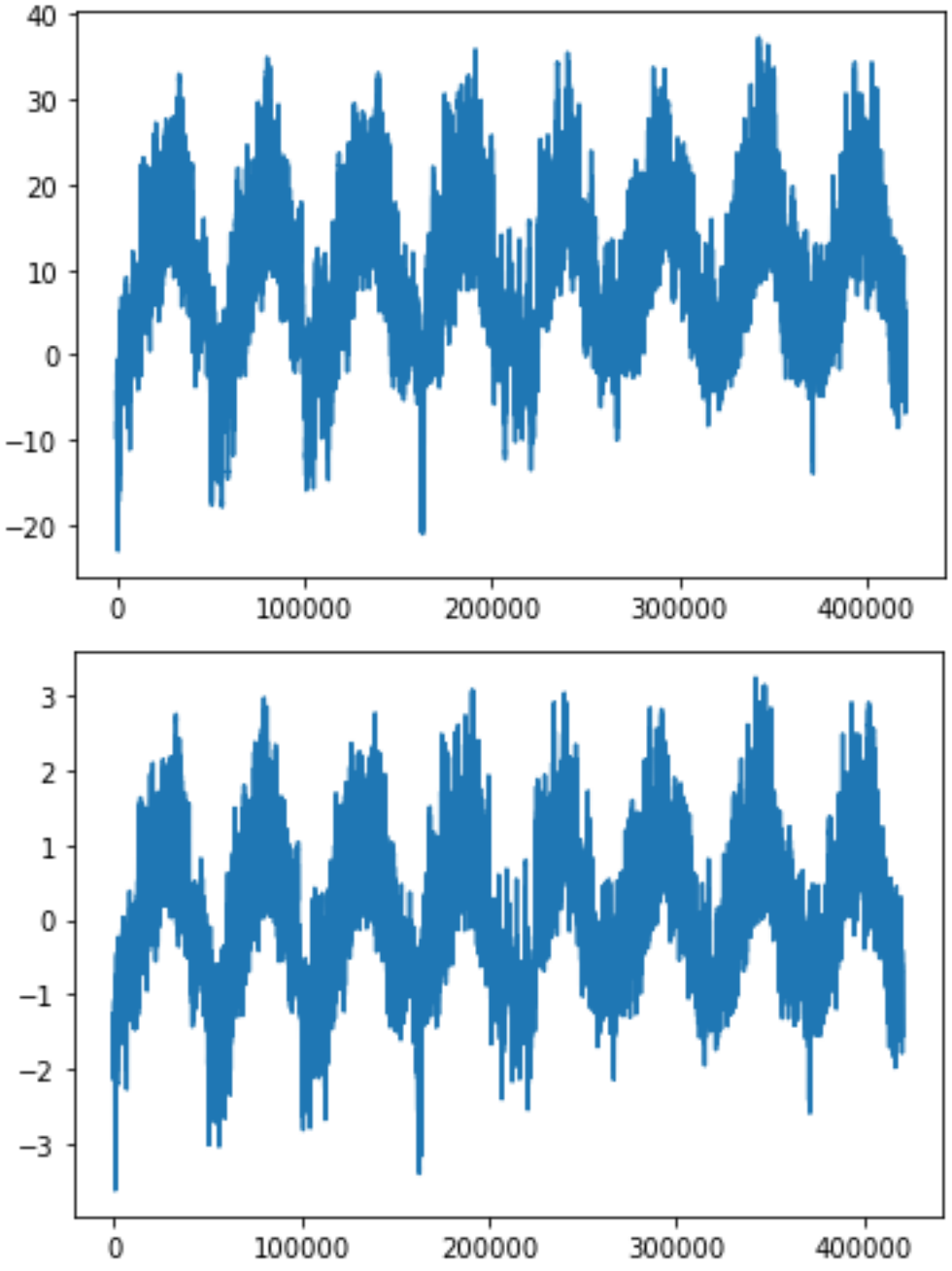
Normalizing the Data

```
mean = raw_data[:num_train_samples].mean(axis=0)
normalized_data = raw_data - mean
std = raw_data[:num_train_samples].std(axis=0)
normalized_data = raw_data / std
```

- A shorter version of the previous code is shown here.
  - No need to loop over each dimension, this code operates on all dimensions simultaneously.
- If you are comfortable with (or interested in learning more about) numpy shortcuts, you can use this type of coding.
- If you would rather use code that is more simple and more readable, then you can use a coding style similar to the previous version.

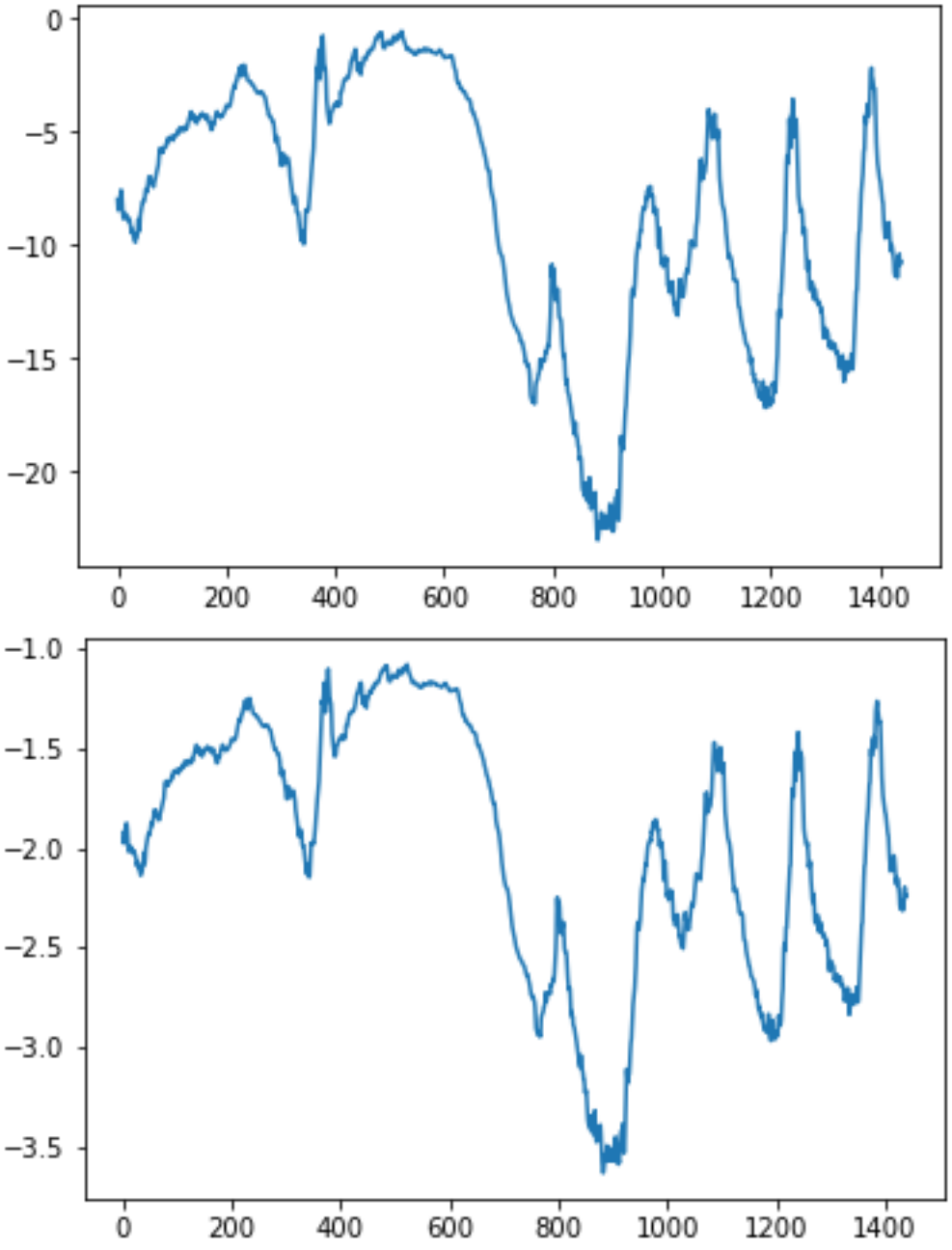
Normalizing the Data

- Top: plot of temperatures over the entire 8 years.
- Bottom: plot of normalized temperatures over the entire 8 years.
- The shapes of the plots are identical, but the ranges are different.
  - Top range: -23.01 to 37.28.
  - Bottom range: -3.63 to 3.24



Normalizing the Data

- Top: plot of temperatures over the first 10 days.
- Bottom: plot of normalized temperatures over the first 10 days.
- Again, the shapes of the plots are identical, but the ranges are different.



Inputs and Target Outputs

- Let's look at our forecasting task description again: given data from the last five days, the goal is to predict the temperature exactly 24 hours from now.
- What will be the input to this "forecasting" system?
- What will be the output of the system?

Inputs and Target Outputs

- Let's look at our forecasting task description again: given data from the last five days, the goal is to predict the temperature exactly 24 hours from now.
- What will be the input to this "forecasting" system?
- What will be the output of the system?
- The input will be a sequence of feature vectors containing all values from a period of five days.
  - Number of columns: 14, since we have 14 features in our data.
  - Number of rows: 5 days * 24 hours * 6 observations per hour = 720.
  - Shape of the input: 720x14, which gives 10080 numbers.
- The output will be a single number: the temperature (in Celsius) 24 hours after the last observation in the input.

Creating a Training Set

- So far, we have said that our training is the first half of all observations.

```
num_train_samples = int(0.5 * len(normalized_data))
training_data = normalized_data[0:num_train_samples, :]
print(training_data.shape)
```

Output:

```
(210225, 14)
```

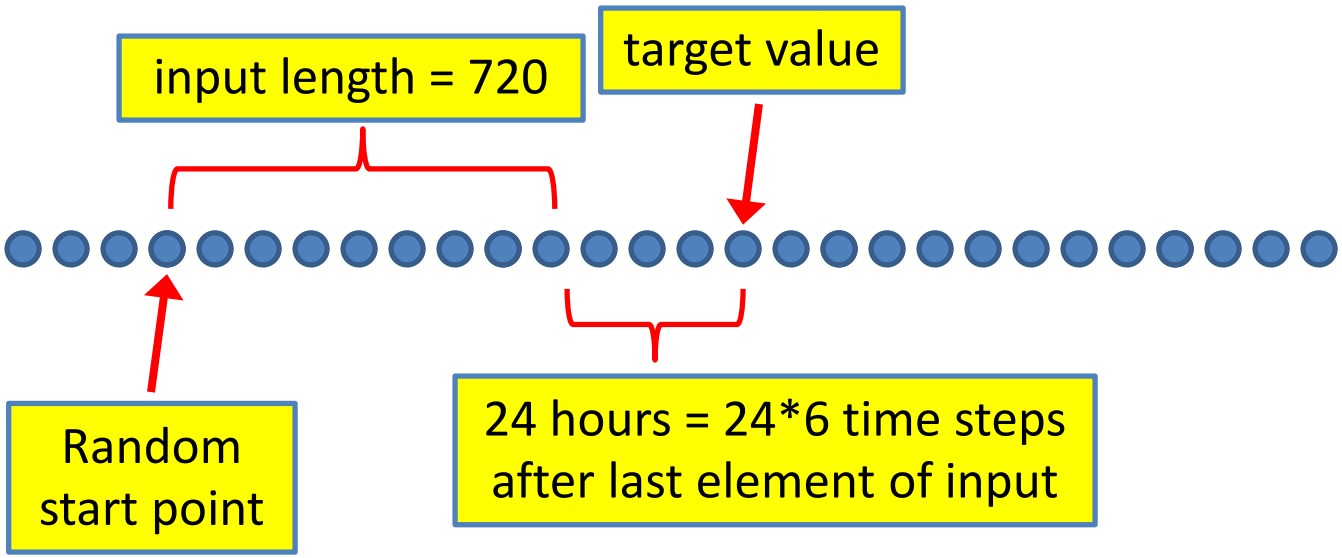
- How do we extract a random training example out of this data?

Creating a Training Set

- Our training data is a 14-dimensional time series of length 210,225.
- We want to extract a random training example of length 720.
- So, we pick a random start point in the time series, and we get the next 720 elements.
- What is the smallest and largest legal value for the start point?
- Smallest: 0
- Largest: timeseries length – 720 – 24*6.
  - Why? We need enough room to choose 720 elements, plus enough room to look 24 hours past the last element, to get the target value that we aim to forecast.

Creating a Training Set

- What is the smallest and largest legal value for the start point?
- Smallest: 0
- Largest: timeseries length – 720 – 24*6.
  - Why? We need enough room to choose 720 elements, plus enough room to look 24 hours past the last element, to get the target value that we aim to forecast.



Selecting a Random Input

```
def random_input_v1(timeseries, length, target_time, target_data):
    (ts_length, dimensions) = timeseries.shape
    max_start = ts_length - length - target_time
    start = np.random.randint(0,max_start)
    end = start + length
    result_input = timeseries[start:end, :]
    target = target_data[end+target_time-1]
    return (result_input, target)

(input1, target1) = random_input_v1(normalized_data, 720, 24*6, temperature)
```

- This code does what we discussed. It returns:
  - A time series called “result_input”, of the desired length.
  - The corresponding target value.

Sampling the Input

- As we said, the input has 14 columns and 720 rows
  - Number of rows: 5 days * 24 hours * 6 observations per hour = 720.
- We can reduce the amount of data, by sampling only one observation per hour, instead of 6.
- The weather does not (usually) fluctuate radically within a single hour, so hopefully the data we discard does not contain much additional information over the data we keep.
  - We can always do an experiment without sampling, to compare running times and accuracy.
- So, this way the input will only have 120 rows instead of 720 rows.
  - In other words, viewed as a time series, the input will have 120 elements instead of 720 elements.

Updated Code for Random Sample

```
def random_input(timeseries, length, target_time, target_data,
                 sampling_rate = 1):
    (ts_length, dimensions) = timeseries.shape
    max_start = ts_length - length - target_time
    start = np.random.randint(0,max_start)
    end = start + length
    result_input = timeseries[start:end:sampling_rate, :]
    target = target_data[end+target_time-1]
    return (result_input, target)

(input2, target2) = random_input(training_data, 720, 24*6, temperature, 6)
```

- This version does sampling.
- The length of the returned sample is length / sampling_rate.

Creating a Set of Examples

```
def example_set(timeseries, number, length, target_time, target_data,
                sampling_rate = 1):
    (ts_length, dimensions) = timeseries.shape
    input_length = math.ceil(length/sampling_rate)
    inputs = np.zeros((number, input_length, dimensions))
    targets = np.zeros((number))

    for i in range(0, number):
        (inp, target) = random_input(timeseries, length, target_time,
                                     target_data, sampling_rate)
        inputs[i] = inp
        targets[i] = target

    return (inputs, targets)
```

To create a set of examples, we just call **random_input()** as many times as we need.

Creating a Training Set

```
training_data = normalized_data[0:num_train_samples, :]
training_temperature = temperature[0:num_train_samples]

(training_inputs, training_targets) = example_set(training_data, 50000, 720,
                                                  24*6, training_temperature, 6)
print("\ntraining time range: (%d, %d)" % (0, num_train_samples))
print("training_inputs.shape:", training_inputs.shape)
print("training_targets.shape:", training_targets.shape)
```

Output:

```
training time range: (0, 210225)
training_inputs.shape: (50000, 120, 14)
training_targets.shape: (50000,)
```

Creating a Validation Set

```
validation_start = num_train_samples
validation_end = validation_start + num_val_samples
validation_data = normalized_data[validation_start:validation_end, :]
validation_temperature = temperature[validation_start:validation_end]

(validation_inputs, validation_targets) = example_set(validation_data, 50000,
                                                      720, 24*6, validation_temperature, 6)
print("\nvalidation time range: (%d, %d)" % (validation_start, validation_end))
print("validation_inputs.shape:", validation_inputs.shape)
print("validation_targets.shape:", validation_targets.shape)
```

Output:

```
validation time range: (210225, 315337)
validation_inputs.shape: (50000, 120, 14)
validation_targets.shape: (50000,)
```

Creating a Test Set

```
test_start = validation_end
test_end = test_start + num_test_samples
test_data = normalized_data[test_start:test_end, :]
test_temperature = temperature[test_start:test_end]

(test_inputs, test_targets) = example_set(test_data, 50000, 720,
                                          24*6, test_temperature, 6)
print("\ntest time range: (%d, %d)" % (test_start, test_end))
print("test_inputs.shape:", test_inputs.shape)
print("test_targets.shape:", test_targets.shape)
```

Output:

```
test time range: (315337, 420451)
test_inputs.shape: (50000, 120, 14)
test_targets.shape: (50000,)
```

A Naïve Method

```
def naive_forecast(timeseries, means, stds):
    (length, dimensions) = timeseries.shape
    temperature_column = 1
    last_temperature = timeseries[length-1, temperature_column]
    mean_temperature = means[temperature_column]
    temperature_std = stds[temperature_column]
    prediction = last_temperature * temperature_std + mean_temperature
    return prediction
```

- Here is a naïve forecasting method, not using machine learning.
- What does this method do?

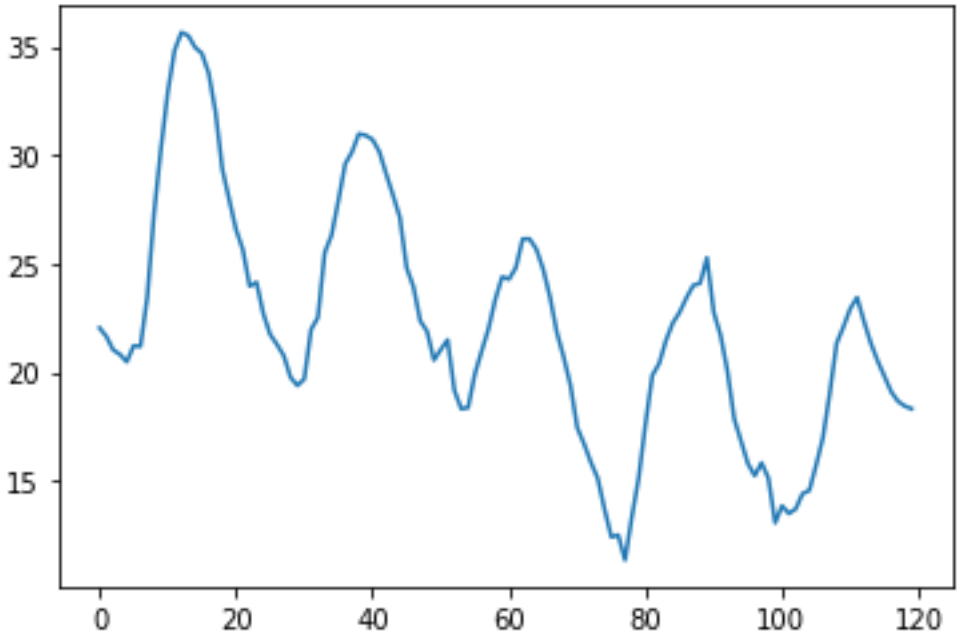
A Naïve Method

```
def naive_forecast(timeseries, means, stds):
    (length, dimensions) = timeseries.shape
    temperature_column = 1
    last_temperature = timeseries[length-1, temperature_column]
    mean_temperature = means[temperature_column]
    temperature_std = stds[temperature_column]
    prediction = last_temperature * temperature_std + mean_temperature
    return prediction
```

- The temperature predicts that the temperature 24 hours later is equal to the temperature right now.
  - It returns the last observed temperature value.
  - Note that we multiply by std and add the mean, to account for the data normalization that we did.

Rationale for the Naïve Method

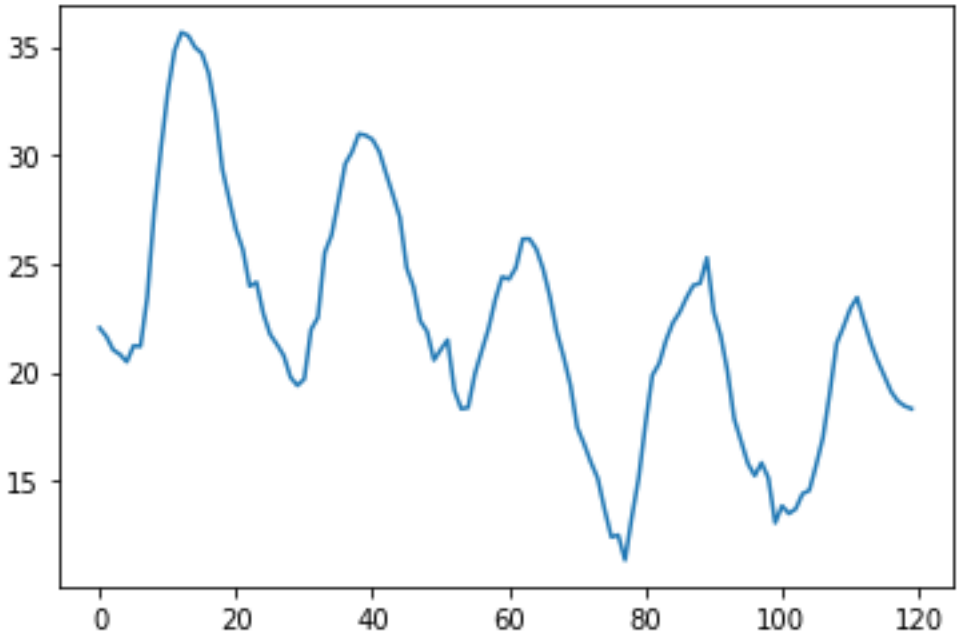
- Here is an example input.
- We can see that temperature values usually follow a 24-hour pattern.
  - Highest values in the afternoon.
  - Lowest values at night.
- So, the current temperature is a reasonable quick guess about the temperature 24 hours later.



Example Results of the Naïve Method

prediction = 18.33

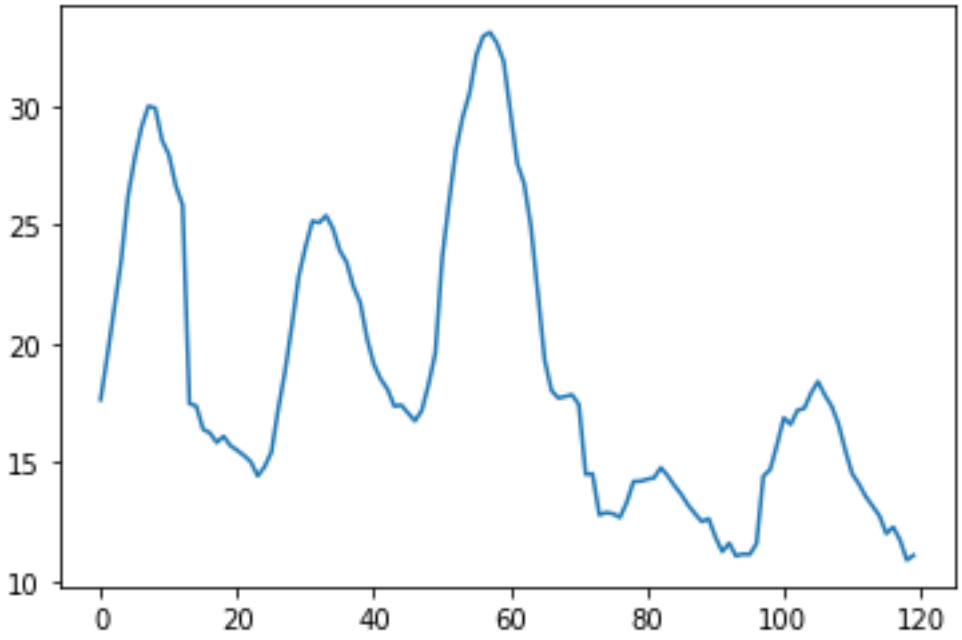
true value = 17.81



Example Results of the Naïve Method

prediction = 11.08

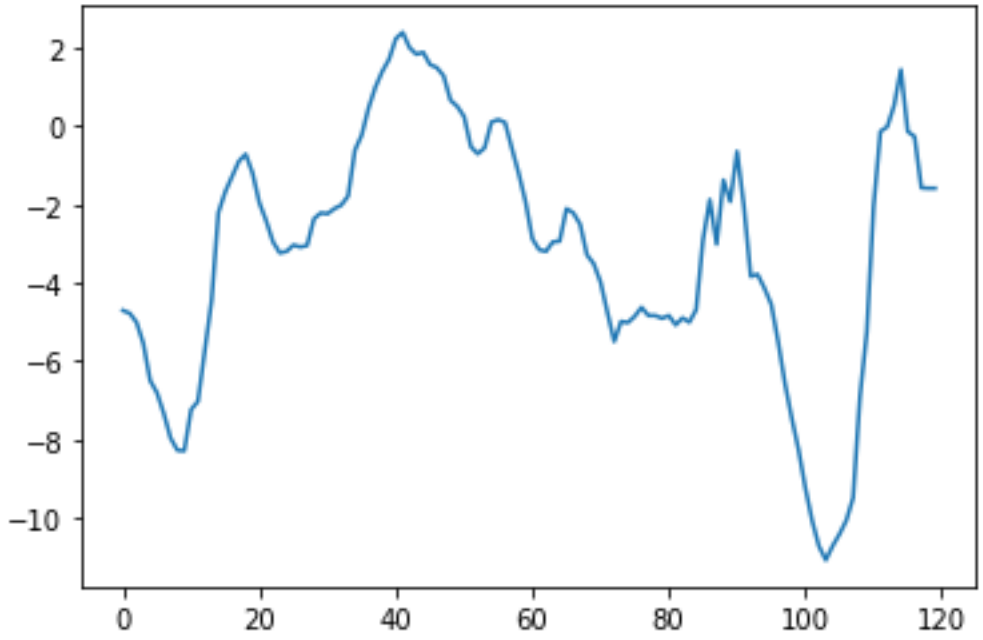
true value = 8.39



Example Results of the Naïve Method

prediction = -1.59

true value = 1.43



Evaluating the Naïve Method

```
def evaluate_naive_forecast(inputs, targets, means, stds):
    (number, ) = targets.shape
    predictions = np.zeros((number,))
    for t in range(0, number):
        predictions[t] = naive_forecast(inputs[t], means, stds)

    errors = np.abs(predictions - targets)
    mean_abs_error = errors.mean()
    return mean_abs_error
```

- This function computes the MAE (mean absolute error) of the predictions on a specific dataset.

Evaluating the Naïve Method

```
training_mae = evaluate_naive_forecast(training_inputs, training_targets,
                                       mean, std)
print("training MAE: %.2f" % (training_mae))

validation_mae = evaluate_naive_forecast(validation_inputs, validation_targets,
                                         mean, std)
print("validation MAE: %.2f" % (validation_mae))

test_mae = evaluate_naive_forecast(test_inputs, test_targets, mean, std)
print("test MAE: %.2f" % (test_mae))
```

Output:

```
training MAE: 2.70
validation MAE: 2.46
test MAE: 2.62
```

Why Bother with a Naïve Method

- In machine learning, it is important to establish some **baseline accuracy**.
- Baseline accuracy is accuracy we can get with relatively little effort.
  - Can be using a naïve method.
  - Can be using a method that already exists and we can use.
- Then, we can compare the results of whatever fancy method we come up with with the baseline method.
  - Is the fancy method more accurate?
  - We hope yes, but sometimes the answer is no. If the answer is no, it is better if you are the first one to find out.
- Comparing to a baseline method helps us find bugs and do sanity checking on our solution.

Using a Neural Network

```
input_shape = training_inputs[0].shape
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.Flatten(),
                          keras.layers.Dense(16, activation="relu"),
                          keras.layers.Dense(1),])
```

- Here is a simple neural network for this task.
- Each input is a 2D array, of dimensions 120x14.
  - The input is a time series of 120 elements.
  - Each element a 14-dimensional vector.
- We flatten each input into a 1D array of size 120*14 = 1680.
- We use one hidden layer, with 16 units, ReLU activation.

MAE on Validation and Test Set

```
val_mae = np.array(history_dense.history["val_mae"])
min_val_mae = val_mae.min()
min_epoch = val_mae.argmin()+1
print("\nmin_val_mae: %.2f, epoch %d" % (min_val_mae, min_epoch))

model = keras.models.load_model("jena_dense1_16.keras")
(loss, test_mae) = model.evaluate(test_inputs, test_targets)
print("Test MAE: %.2f" % (test_mae))
```

Output:

```
Smallest validation MAE: 2.56, reached in epoch 4
Test MAE: 2.64
```

MAE on Validation and Test Set

- I ran the training 8 times.
- The validation MAE ranged from 2.54 to 2.66.
  - With the “naïve” method we got a validation MAE of 2.46.
- The test MAE ranged from 2.63 to 2.71.
  - With the “naïve” method we got a test MAE of 2.62
- Interestingly, the “naïve” method outperforms our neural network.
  - I tried two hidden layers, 120 units per layer, similar results.
- This is a good example of why it is useful to implement and evaluate some “naïve” baseline methods.

“Naïve” vs. Neural Network

- Neural network: validation MAE ranged from 2.54 to 2.66.
- “Naïve” method: validation MAE = 2.46.
- Neural network: test MAE ranged from 2.63 to 2.71.
- “Naïve” method: test MAE = 2.62
- What are some possible reasons that the “naïve” method gives better results than the neural network?

“Naïve” vs. Neural Network

- The “naïve” method is designed by humans, and uses knowledge that we have about this problem.
  - We know that, more often than not, the weather does not change dramatically from one day to the next.
- The neural network training simply searches over a very large space of parameters (26,913 weights).
- The larger the search space, the harder it is to find a good solution.
  - There can be too many local optima that correspond to worse solutions than the “naïve” method.

A Closer Look at the Model

```
input_shape = training_inputs[0].shape
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.Flatten(),
                          keras.layers.Dense(16, activation="relu"),
                          keras.layers.Dense(1),])
```

- Even though this specific model does not work as well, we should take a closer look.
- There are many important new concepts that are introduced here, that we should explain.
  - As we try more sophisticated models, we will continue using some of these concepts.

Output of the Network

```
input_shape = training_inputs[0].shape
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.Flatten(),
                          keras.layers.Dense(16, activation="relu"),
                          keras.layers.Dense(1),])
```

- The output layer has a single unit.
- This makes sense, since we want to predict a single value: the temperature 24 hours after the last observation in the input.
- Notice that we specify no activation function for the output unit.
- This means that the **identity** function will be used as activation.
  - The output unit will output the weighted sum of its inputs.
- Why?

Regression Problem

```
input_shape = training_inputs[0].shape
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.Flatten(),
                          keras.layers.Dense(16, activation="relu"),
                          keras.layers.Dense(1),])
```

- This is our first **regression problem**.
- In a classification problem, the targets are integer class labels, or one-hot vectors.
- In a regression problem, the targets are real numbers.
  - We use no activation for the output layer, so that we do not limit the range of the output values.

Regression Problem

```
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
```

- Notice the use of Mean Squared Error (MSE) as the loss function.
- Mathematically, MSE is equivalent to using the sum-of-squared errors (SSE).
  - We just divide SSE by the number of training objects.
- Categorical Cross-Entropy would not make sense to use here.
  - The targets are real numbers, NOT integer class labels or one-hot vectors.
- Also, note that we specify Mean Absolute Error (MAE) as the metric to watch during training.

Callbacks for Saving Best Model

```
callbacks = [keras.callbacks.ModelCheckpoint("jena_dense1_16.keras",
                                             save_best_only=True)]

history_dense = model.fit(training_inputs, training_targets, epochs=10,
                          validation_data=(validation_inputs, validation_targets),
                          callbacks=callbacks)
```

- The **callbacks** argument to **fit()** specifies actions that should be taken at specific points in the training.
- Here we specify **ModelCheckpoint()** as the callback.
- **ModelCheckpoint()** saves a model to a file during training, so that we keep track of the values of the weights after each epoch.
  - The **save_best_only=True** parameter specifies to save only the best intermediate model (best according to the metric we have specified, on the validation set).

Loading the Best Model

```
model = keras.models.load_model("jena_dense1_16.keras")
(loss, test_mae) = model.evaluate(test_inputs, test_targets)
print("Test MAE: %.2f" % (test_mae))
```

- To measure the test error, we load the best model.
- This is a good example of how to use validation data.
  - The training ran for 20 epochs.
  - It may be that, after a certain point, we started getting overfitting.
  - The error on the validation data is a good way to identify the epoch where the model was the most accurate on non-training data.
  - We save the weights that worked the best on the validation data, and we load that version of the model whenever we need to use the model.

Using the Training History

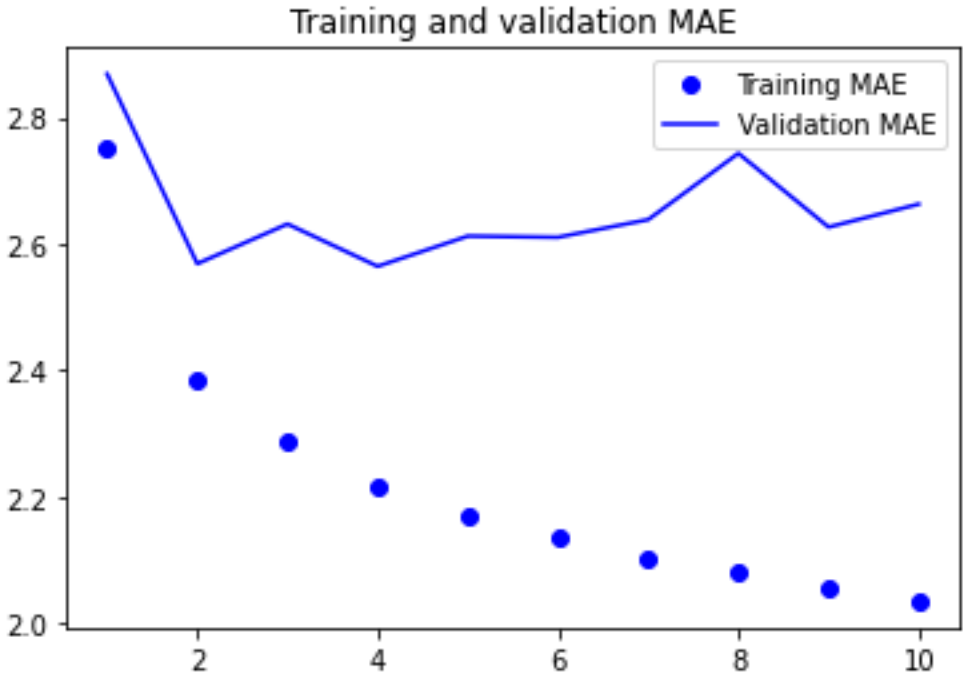
```
history_dense = model.fit(training_inputs, training_targets, epochs=10,
                validation_data=(validation_inputs, validation_targets),
                callbacks=callbacks)

val_mae = np.array(history_dense.history["val_mae"])
min_val_mae = val_mae.min()
min_epoch = val_mae.argmin()+1
print("\nSmallest validation MAE: %.2f, reached in epoch %d" %
    (min_val_mae, min_epoch))
```

Output:

```
Smallest validation MAE: 2.56, reached in epoch 4
```

- We can use the training history to identify the best epoch.

Visualizing the Training Progress

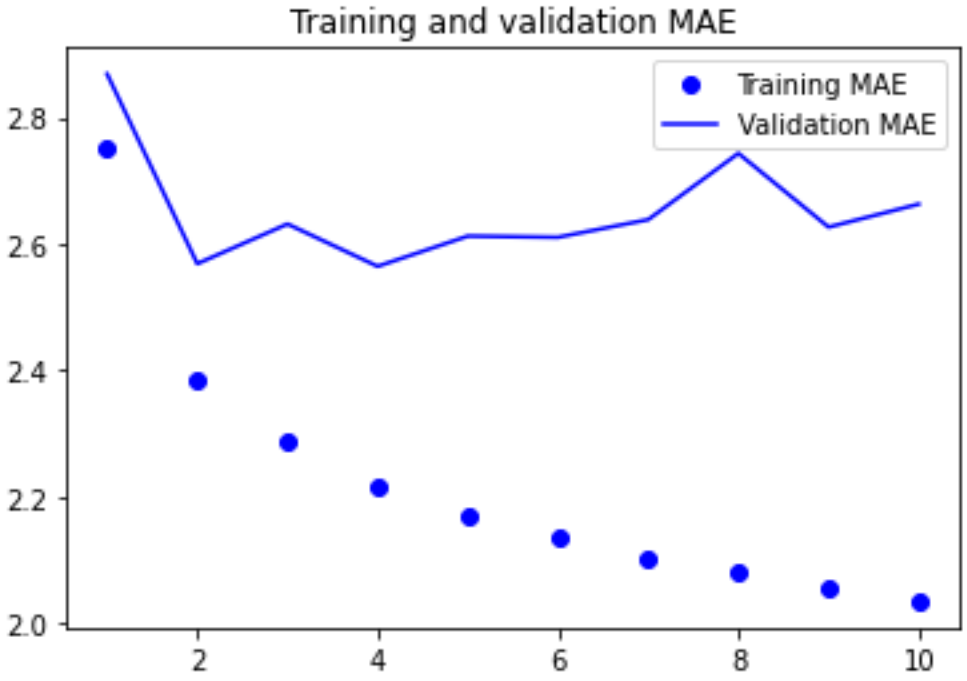


- This piece of code shows the training and validation MAE achieved after each epoch.

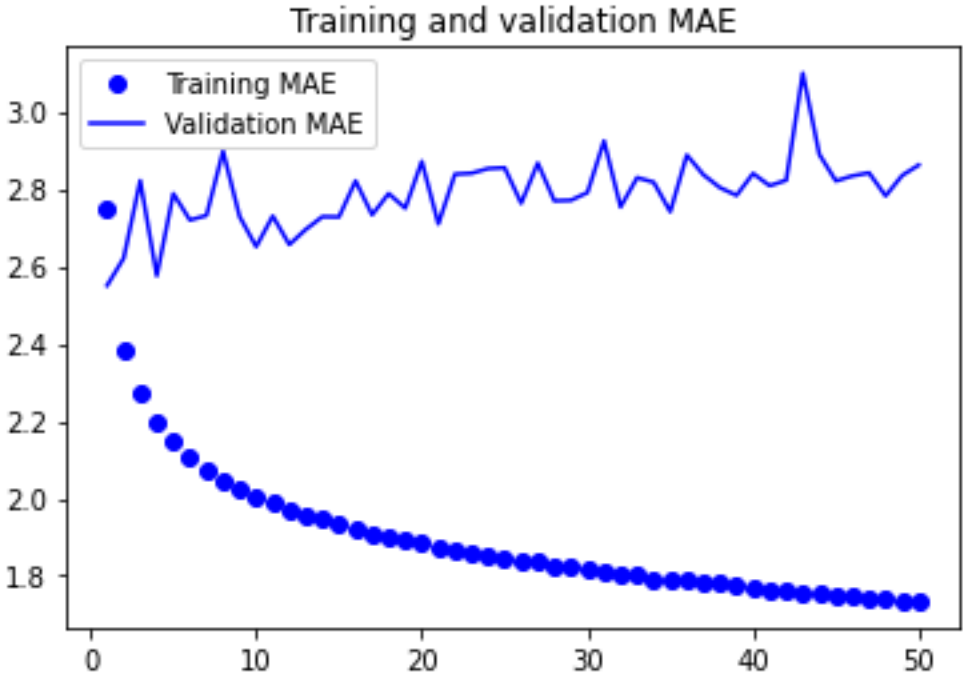
```
loss = history_dense.history["mae"]
val_loss = history_dense.history["val_mae"]
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, "bo", label="Training MAE")
plt.plot(epochs, val_loss, "b", label="Validation MAE")
plt.title("Training and validation MAE")
plt.legend()
plt.show()
```

Visualizing the Training Progress

- We have a mild case of overfitting.
- The training MAE decreases steadily.
- The validation MAE reaches a minimum of 2.56 after epoch 4, and then it gets somewhat worse.



Visualizing the Training Progress



- Here is the result of another experiment, where we train for 50 epochs.
- The behavior is similar.
- The best value for the validation MAE is reached after epoch 4.
- After that, values get slightly worse.

Preview: RNNs

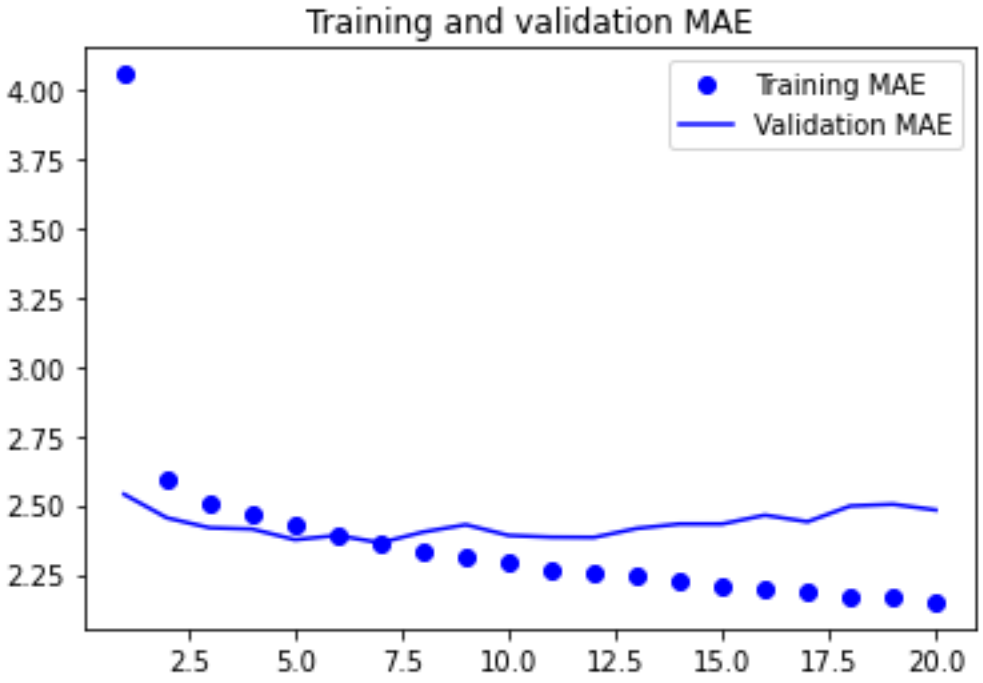
- Our next topic is recurrent neural networks (RNNs).
- RNNs have been explicitly designed for time series.
- Compared to the simple neural network we tried before, the design of RNNs incorporates our human understanding of how we should analyze a time series.
- Not surprisingly, RNNs work (somewhat) better than simple fully-connected models on the temperature forecasting problem, and in many other time-series problems.

Preview: RNNs

```
model = keras.Sequential([keras.Input(shape=input_shape),
                          keras.layers.SimpleRNN(16),
                          keras.layers.Dense(1),])

model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])

history_lstm = model.fit(training_inputs, training_targets, epochs=20,
                         validation_data=(validation_inputs, validation_targets))
```

- This is our first RNN model.
- It uses a “SimpleRNN” layer.
- We will see in detail how RNNs work in the next set of slides.

RNN Results



- Validation MAE:
  - RNN: 2.31 to 2.34.
  - Fully-connected network: 2.54 to 2.66.
  - "Naïve" method: 2.46.
- Test MAE:
  - RNN: 2.45-2.48.
  - Fully-connected network: ranged from 2.63 to 2.71.
  - "Naïve" method: 2.62.
- Each model was trained and evaluated 8 times.
- The RNN works better, but the improvement is not dramatic.
  - In other datasets, bigger (or smaller) differences in accuracy can be observed.