University of Warsaw
Faculty of Physics

Ariel Kwiatkowski
formerly known as (and published as) Jakub Kwiatkowski
Student's book no.: 370359

High-frequency airborne temperature measurements analyzed with artificial intelligence techniques

First cycle degree thesis
field of study Physics, Individual Study Programme

The thesis written under the supervision of
Prof. dr hab. Szymon Malinowski
Department of Atmospheric Physics
Institute of Geophysics

Warsaw, June 2018

Abstract

The purpose of this work is analyzing the temperature measurements from the UltraFast Thermometer 2.0 (UFT-2) during the campaign ACORES 2017. Several types of anomalies in the readings were investigated and labelled using an app written for that purpose. Several machine learning algorithms were tested on the task of automatically detecting said anomalies, using the data from UFT-2 and readings from other devices, including the liquid water content and wind speed.

Key words

machine learning, artificial intelligence, temperature, deep learning, neural neworks

Area of study (codes according to Erasmus Subject Area Codes List)

13.2 Physics

The title of the thesis in Polish

Analiza pomiarów temperatury o wysokiej częstotliwości z użyciem technik sztucznej inteligencji

Table of Contents

1. Introduction 5

2. Measurements 6
- 2.1. ACTOS 6
- 2.2. UFT-2 7
- 2.3. Data overview 8
- 2.4. Anomalies 9

3. Anomaly detection 12
- 3.1. General problem formulation 12
- 3.2. Window classification 13
  - 3.2.1. Single thermometer window classification 14
- 3.3. Data generation 15

4. Machine learning methods 17
- 4.1. Feedforward neural networks 19
- 4.2. Gradient descent 19
  - 4.2.1. Validation split 20
  - 4.2.2. Stochastic Gradient Descent 20
  - 4.2.3. Adam 21
  - 4.2.4. Convergence 22
- 4.3. Convolutional neural networks 22
- 4.4. Recurrent neural networks 23
  - 4.4.1. Gated recurrent layers 23
- 4.5. Regularization 24

5. Experiments and results 25
- 5.1. Evaluation metrics 25
- 5.2. Normalization 26
- 5.3. Single-feature models 26
  - 5.3.1. Feedforward network 26
  - 5.3.2. Convolutional network 26
  - 5.3.3. Recurrent networks 27

5.4. Multi-feature models 28
5.4.1. Feedforward network 28
5.4.2. Convolutional network 28
5.4.3. Recurrent networks 28

6. Summary 30

Chapter 1

Introduction

Artificial intelligence techniques have seen a wide range of applications throughout the last decade, from image classification and natural language processing, to anomaly detection and image generation, with neural networks being exceptionally versatile with the plethora of various architectures like Convolutional Neural Networks[11] particularly effective for processing image data, and Recurrent Neural Networks[12] - for temporal data.

ACORES, which stands for *Azores stratoCumulus measurements Of Radiation, turbulEnce and aeroSols*, is a measurement campaign carried out around the Azores Islands on 1-23 July 2017, with the goal of investigating the structure of certain clouds.

UltraFast Thermometer 2.0 (UFT-2) is an instrument designed to measure airborne temperature at high resolution, both in clouds and outside of them. Its main components are two thin, resistive tungsten wires hidden behind a steel rod to shield them from water droplets.[15]

This work aims to explore the possibility of applying machine learning algorithms to detecting anomalies in the temperature measurements from UFT-2 during the campaign ACORES 2017. The basic idea is splitting the temperature series into shorter windows, and training a supervised learning algorithm to detect whether or not there is any anomaly in a given window.

Chapter 2

Measurements

The main purpose of the ACORES campaign was twofold – to investigate the small-scale structure of the stratocumulus top, with special consideration of the entrainment interface layer (EIL), and to analyze the aerosol and CCN budget in the relatively pristine marine boundary layer.[15] During the campaign, a helicopter was flown around the archipelago of Azores with two external instrumental platforms suspended underneath: SMART-HELIOS[6] maintained by the University of Leipzig and ACTOS[18] maintained by the Leibniz Institute for Tropospheric Research.

During the campaign, 18 flight were carried out, out of which 17 employed the UFT-2, each flight being up to 2 hours long; however the thermometer skipped three flights between the sensing wire breaking and being replaced, giving a total of 14 flights with data from UFT-2[15].

2.1. ACTOS

The Airborne Cloud Turbulence Observation System, or ACTOS in short, is particularly important for this work since various sensors from it were used for the analysis of the UFT-2 data. It was suspended on a 150 m long rope under the helicopter, with two UFT-2 sensors mounted on it, 2 centimeters away from each other vertically, as seen in Figure 2.1.

The platform is fitted with the following sensors:[18]

- Ultrasonic anemometer/thermometer to measure the 3D wind vector and the virtual air temperature, sampled at 100 Hz
- A combination of the dGPS and inertial sensors to measure the attitude angles, angular rates, position and velocity, sampled at 10 Hz (optionally 100 Hz)
- A modified version of the UltraFast Thermometer (UFT)
- A Lyman-α absorption hygrometer to measure humidity fluctuations

- PT-100 resistance-wire thermometers as a reference for the UFT-2 and a capacitive hygrometer to measure air temperature and relative humidity as a reference for Lyman-α sampled at 1 Hz.
- Two condensation particle counters (CPCs) to measure the number concentration of interstitial aerosol particles
- Particle Value Monitor PVM-100A to measure the liquid water content (LWC)
- A modified version of the Fast Forward Scattering Spectrometer Probe (M-Fast-FSSP) to measure cloud droplets

2.2. UFT-2

The design of the UltraFast Thermometer 2.0 is based on its predecessor, UltraFast Thermometer (UFT)[5]. The most notable differences are a much smaller size of the instrument, and a lack of moving parts. Its main components are two ultra-thin (1.25 µm diameter) tungsten wires spanned across the arms of a trident, shielded by a 0.25 mm thick steel rod which serves as protection against cloud droplets and ice crystals, as seen in Figure 2.1.

Its maximum frequency of response reaches 10 kHz in flight conditions, however in this campaign it was oversampled and operated at 20 kHz (with the effective sampling frequency dependent on variables such as the velocity of the platform), which was further downsampled to 100 Hz, matching the sampling frequency of several ACTOS sensors.

The raw output from the UFT-2 consists of two time-indexed voltage vectors (one for each wire), which is then linearly calibrated with reference thermometers and converted to temperatures.

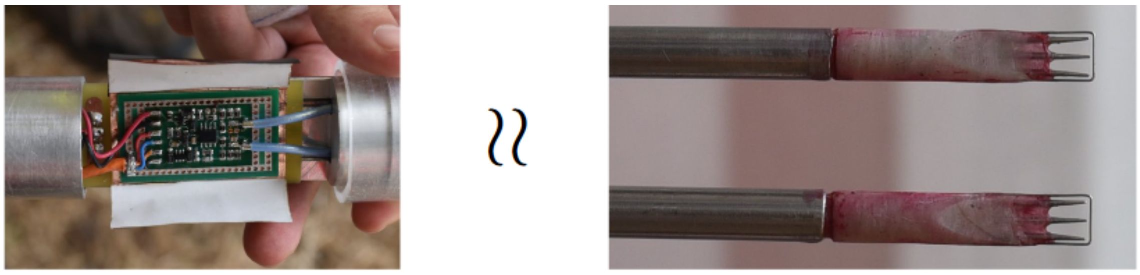

Figure 2.1: Photo of the amplifier (left) and of the temperature sensors (right) used in the UFT-2 during the campaign[15]

2.3. Data overview

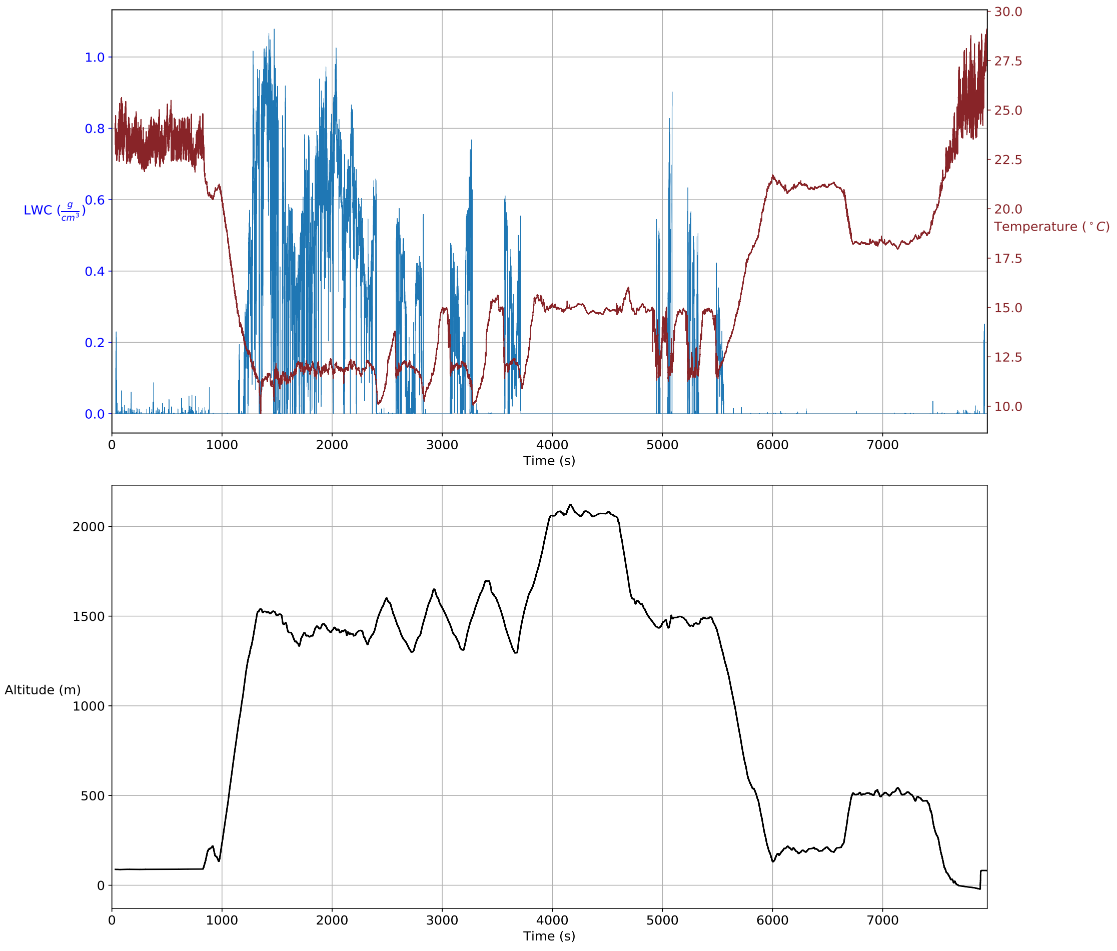

Figure 2.2: (up) The temperature from the lower UFT-2 (the upper one would be indistinguishable at this scale), and the liquid water content throughout the flight.
(down) The altitude of the ACTOS platform, as measured by the mounted GPS.

In Figure 2.2 you can see the temperature and liquid water content measured during Flight 16. The helicopter takes off at approximately 900 seconds (the noisy readings before that moment are due to the fact that the platform was on the ground, affected by disturbances related to helicopter's operations), enters a cloud and stops climbing at the height of about a kilometer. At around 2500 seconds it starts dipping in and out of the cloud several times, until 3700 seconds when it climbs even higher. At 4600 seconds it starts descending and probes the cloud again, until it descends even further between 5500 and 6000 seconds. At 6600 seconds it ascends once again, and at at 7500 seconds it descends and proceeds to finish the flight.

2.4. Anomalies

A closer inspection of the UFT-2 data reveals several distinctive patterns, or anomalies, in the measurements. These anomalies were divided into three categories: jumps, noises and hills. The jumps are characterized by a relatively large change in the temperature value over very few datapoints (up to 3-5). The jump can occur on either only one thermometer or on both, and the value can either increase or decrease. In some cases, the values after the jump go back to their previous range in a very short time, as seen in Figure 2.3. In other cases, the values after the jump stay within the new range, as seen in Figure 2.4.

The second category of anomalies are areas with an increased amplitude of noise, again, either on one or both thermometers. An example can be seen in Figure 2.5, from 1396 seconds onward on the blue line. The last category are areas resembling hills (possibly upside down), often coupled with increased noise, as seen in Figure 2.6.

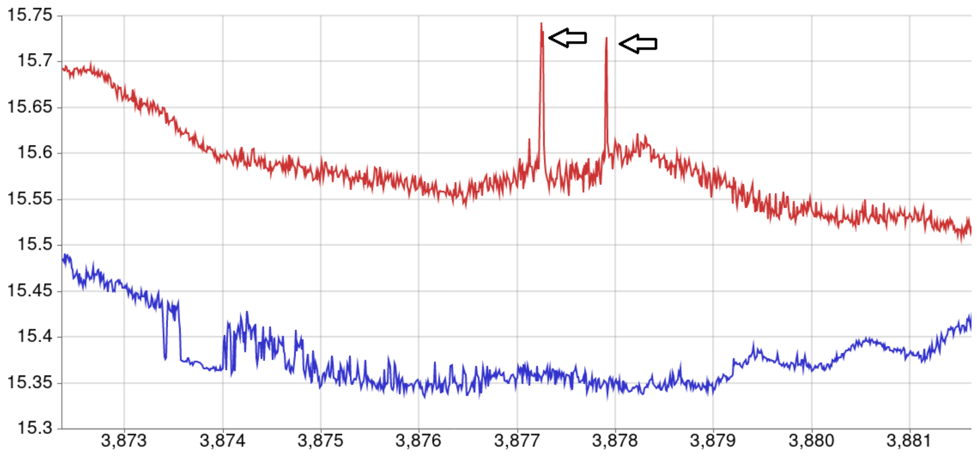

Figure 2.3: An example of an anomalous jump in the readings.

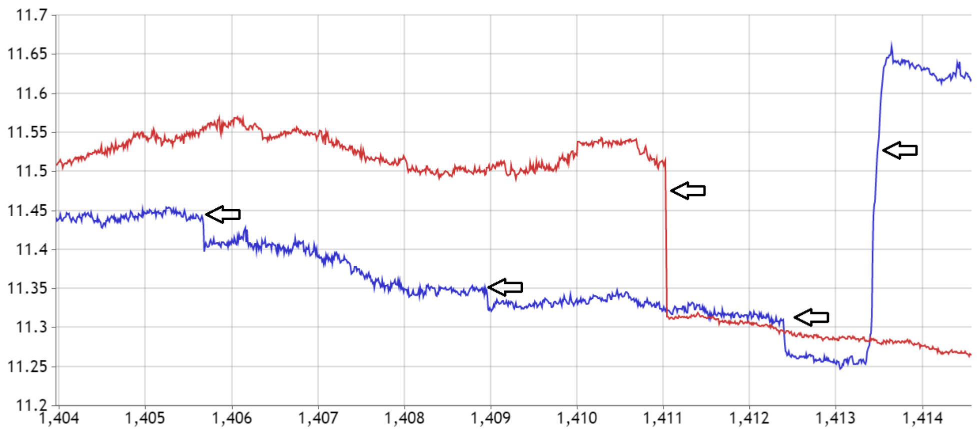

Figure 2.4: An example of an anomalous jump in the readings.

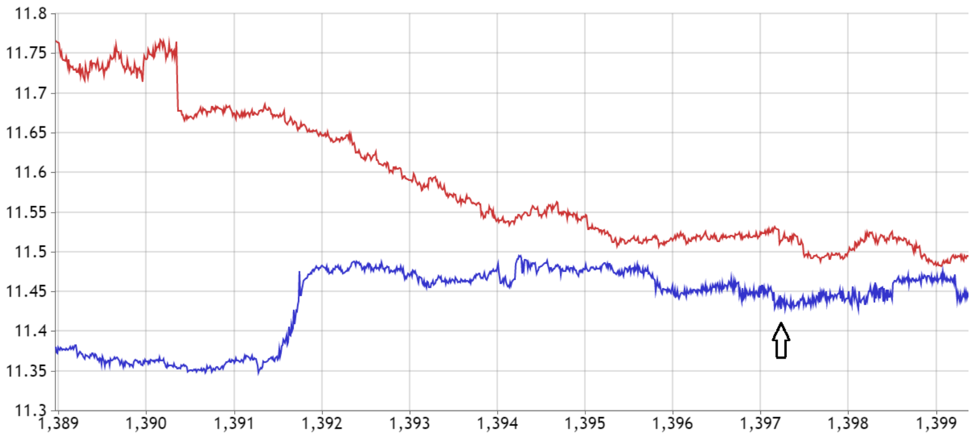

Figure 2.5: An example of increased noise in the readings.

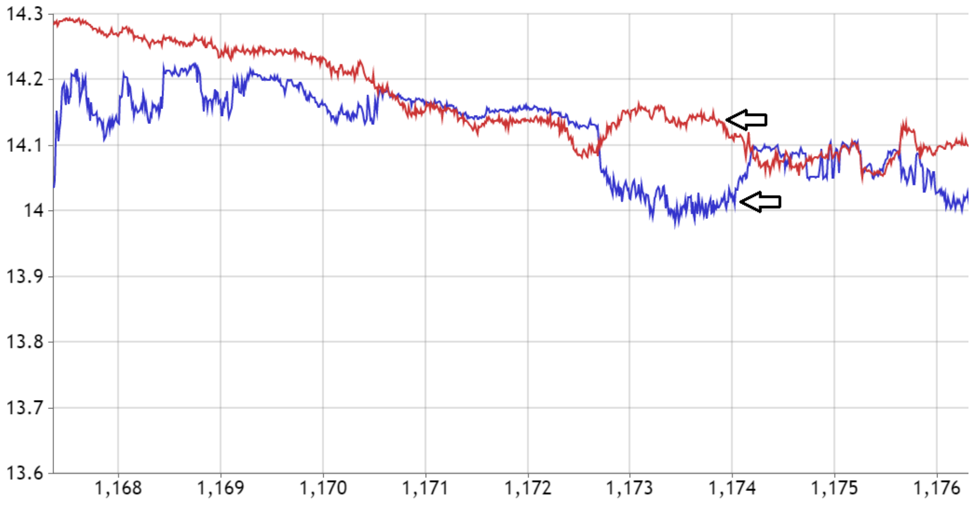

Figure 2.6: An example of an anomalous hill in the readings.

In Figure 2.7 you can see the density of jumps during Flight 16, with the value of each bar in the histogram corresponding to the number of anomalies in that time window. The locations of the jumps have been spotted via visual inspection and marked with the use of a Node.js application written for that purpose.

The total number of jumps spotted is 980 in the lower thermometer, and 1186 in the lower thermometer.

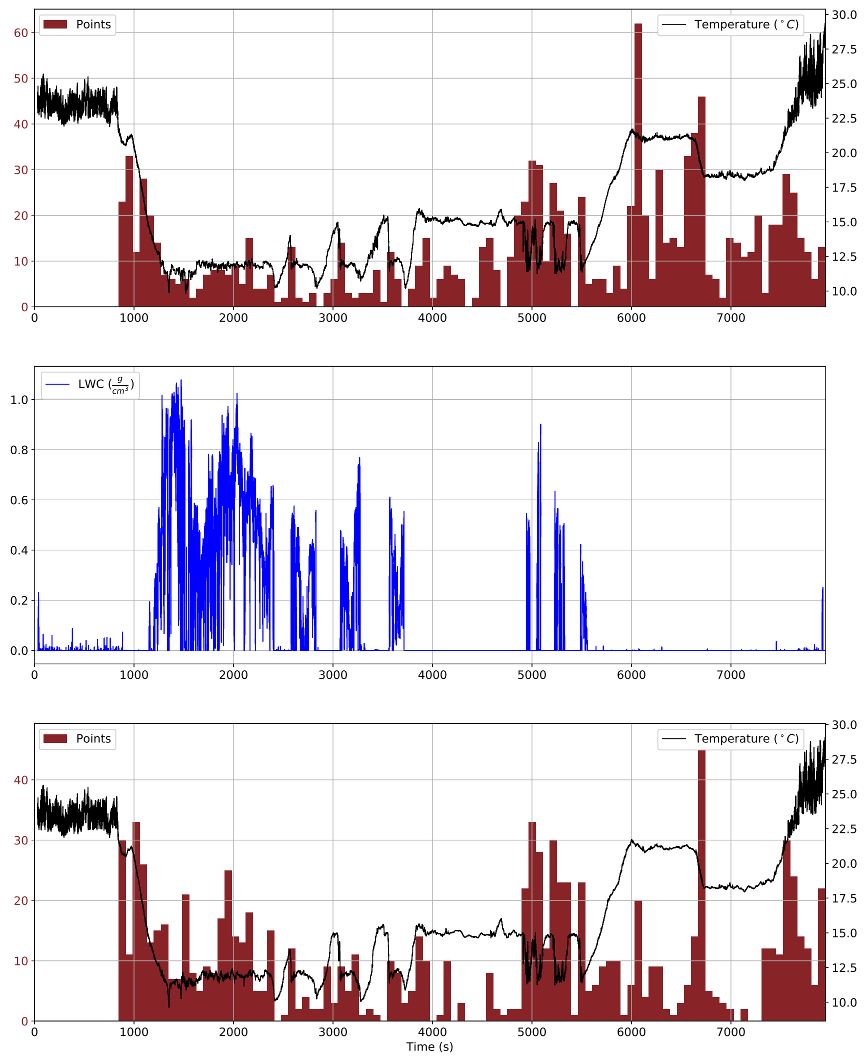

Figure 2.7: (up) Temperature from the upper thermometer, along with the density of anomalies detected on it.
(mid) Liquid water content (LWC) plotted over time
(down) Temperature from the lower thermometer, along with the density of anomalies detected on it.

Chapter 3

Anomaly detection

The main purpose of this work was finding a way of automatically detecting anomalies (specifically, jumps), given the temperature record and other measurements obtained from the ACTOS platform.

3.1. General problem formulation

Let's define $F_i \in \mathbb{R}^{n_f \times L_i}$ to be the flight matrix describing the i-th flight, where L_i is the length of that flight, and n_f is the number of measurements (or features) selected from the ACTOS platform (these will include, for example the temperatures on both thermometers and LWC). Let's also define

$$F := \{F_i\}_{i \in \Lambda}$$

where Λ is a set of indices. In other words, $F \subset \mathbb{R}^{n_f \times L_i}$ is the set of all flight matrices (since, obviously, not all matrices could correspond to an actual, physical flight).

Here's an example of a flight matrix with $n_f = 5$, with the following features: temperatures at the lower and the upper UFT-2, and the three coordinates of the velocity of the wind measured by the ACTOS platform.

$$F_i = \begin{bmatrix} T_0^{low} & T_1^{low} & T_2^{low} & \dots & T_{L_i-1}^{low} \\ T_0^{up} & T_1^{up} & T_2^{up} & \dots & T_{L_i-1}^{up} \\ v_0^1 & v_1^1 & v_2^1 & \dots & v_{L_i-1}^1 \\ v_0^2 & v_1^2 & v_2^2 & \dots & v_{L_i-1}^2 \\ v_0^3 & v_1^3 & v_2^3 & \dots & v_{L_i-1}^3 \end{bmatrix}$$

Let's also define $l_i \in [0, 1]^{2 \times L_i}$ to be the label matrix of the i-th flight, with the first and the second row describing the occurrence of anomalies in the lower and the upper UFT-2 measurement, at any given timestep (indexed by the columns of the label matrix).

[1]The reason why the labels take continuous instead of binary values is two-fold. Firstly, machine learning algorithms generally operate on probability distributions so this formulation will make everything work nicely. Furthermore, due to the purely empirical and unclear definition of a jump, it might not be so clear in some cases whether a specific pattern is one.

Specifically, $(l_i)_{n,t} \in [0,1]$ is the probability[1] that on the t-th timestep there occurred an anomaly on the n-th thermometer.

Then the problem can be rephrased as finding an approximation of the mapping

$$f : \mathbb{R}^{n_f \times L_i} \supset F \longrightarrow [0,1]^{2\times L_i} \subset \mathbb{R}^{2\times L_i}$$

which fulfills the condition that $\forall_{i\in\Lambda} : f(F_i) = l_i$.

Since both the argument and the output of f could be flattened into vectors, then in principle, given enough flight and label matrices of a given length, a simple neural network could be trained to approximate f arbitrarily well for each L_i.[8] This will be described in further detail in Chapter 4.

In practice, however, this approach doesn't work due to the curse of dimensionality[9][2] combined with the heavily limited dataset, with only several flights available, and even fewer of them labelled.

Another obstacle is the fact that a feedforward neural network can only take fixed-length inputs, facilitating the need for multiple networks, one for every value of L_i.[3]

Therefore, some simplifying assumptions need to be made:

- Limited importance of the past and the future – measurements that are far away (temporally) from a specific timestep aren't important for the prediction at that timestep
- Translational invariance – it's not important whether the considered timestep is at the beginning, at the end, or somewhere in the middle of the flight

With these premises in mind, it becomes justified to pose a slightly different problem, which serves as a proxy to solve the original one.

3.2. Window classification

Using the definitions from Section 3.1, let's further define the set of all windows in Flight i, its subsets of fixed-length windows, and the set of fixed-length windows from all flights:

$$W_i := \bigcup_{0\leqslant j\leqslant k\leqslant L_i} \{(F_i)_{:,j:k}\}$$

$$W_i^l := \{w \in W_i | k - j = l\} \subset \mathbb{R}^{n_f \times l}$$

$$W^l := \bigcup_{i\in\Lambda} W_i^l \subset \mathbb{R}^{n_f \times l}$$

[2]The amount of data needed to train a classifier grows exponentially with the dimensionality of the data. In this case L tends to be quite large, going well into hundreds of thousands of values.

[3]This specific issue could be remedied with the use of recurrent models, such as Long Short-Term Memory networks[7]

In other words, a window $w \in W_i^l$ consists of l consecutive columns from F_i. Each of these windows has a label vector $l_w \in [0,1] \times [0,1]$ associated with it, indicating whether or not there's an anomaly on the thermometers:

$$w = (F_i)_{:,j:k} = \begin{bmatrix} T_j^{low} & T_{j+1}^{low} & \cdots & T_{k-1}^{low} \\ T_j^{up} & T_{j+1}^{up} & \cdots & T_{k-1}^{up} \\ v_j^1 & v_{j+1}^1 & \cdots & v_{k-1}^1 \\ v_j^2 & v_{j+1}^2 & \cdots & v_{k-1}^2 \\ v_j^3 & v_{j+1}^3 & \cdots & v_{k-1}^3 \end{bmatrix}$$

$$l_w := \max_{j<n<k-1} \{(l_i)_{:,n}\} \in [0,1] \times [0,1]$$

The modified anomaly detection problem can be described as finding an approximation of the mapping

$$f_l : \mathbb{R}^{n_f \times l} \supset W^l \longrightarrow [0,1] \times [0,1]$$

such that $\forall_{w \in W^l} : f_l(w) = l_w$.

The window length l needs to be chosen based on manual inspection – it needs to be large enough to store information about the entire jump and its immediate surroundings, but at the same time small enough to avoid the curse of dimensionality mentioned before.

After developing a model to solve the window classification problem, it could be applied to every window in the flight to perform a task as described in Section 3.1.

3.2.1. Single thermometer window classification

To further simplify the task at hand, a slight modification can be made to the problem described above. Specifically, instead of predicting the label $l_w \in [0,1] \times [0,1]$, one could predict only one of these labels, with everything else remaining the same. So, we want to approximate the functions:

$$f_l^j : W^l \longrightarrow [0,1]$$

such that $\forall_{w \in W^l} : f_l^j(w) = (l_w)_j$ for $j \in \{1,2\}$.

Having developed models approximating those two functions, those can be ran together to get a model as described in Section 3.2.

The advantage of this modified model is its simplicity – it's just binary classification of a fixed-sized input, which is a well studied topic with plenty of available algorithms and evaluation metrics. It's also easier to construct a well designed dataset (as will be shown in Section 3.3), since only some anomalies coincide on both thermometers, and other times they're in various distances from one another.

On the other hand, training a single model to predict both labels generally has the benefit of sharing parameters between the predictions on both thermometers, which could help with generalization.

Going forward, the single thermometer window classification problem will be considered.

3.3. Data generation

The current task is essentially a standard machine learning problem of performing binary classification on a fixed-size real-valued vector (here $n_f l$-dimensional). One thing that is missing is the specific dataset obtained from a flight. The naive method would be taking all slices of F_i with length l, yielding a total of $L_i - l + 1$ datapoints, with a very high overlap between windows and a vast majority of them having the label 0. What's more, the amount of generated data would be very large, which, combined with the class imbalance, would call for subsampling it. Instead, alternative algorithms were used in order to limit the overlap and the class imbalance.

Algorithm 1 Generating positive datapoints

```
1: $feature\_list \leftarrow$ empty list
2: **Require:** $features$ : flight matrix $n_f \times L_i$
3: **Require:** $labels$ : label matrix $2 \times L_i$
4: **Require:** $l$ : window size
5: **for** $j$ in $0..(L_i - l + 1)$ **do**
6:     $window \leftarrow labels[j : j + l]$
7:     $left \leftarrow \lfloor j + \frac{1}{4}l \rfloor$
8:     $center \leftarrow \lfloor j + \frac{2}{4}l \rfloor$
9:     $right \leftarrow \lfloor j + \frac{3}{4}l \rfloor$
10:    **if** $labels[left] == 1$ or $labels[center] == 1$ or $labels[right] == 1$ **then**
11:        $feature\_list.push(features[j : j + l])$
12: **return** $feature\_list$
```

To generate the positive datapoints, an iteration is performed over all fixed-size windows of the flight. A given window is included in the data if there's an anomaly (i.e. a positive label) in $\frac{1}{4}$, $\frac{2}{4}$ or $\frac{3}{4}$ of its length, as described in Algorithm 1. In this case, l is assumed to be divisible by 4, but otherwise, the nearest point can be used.

To generate the negative datapoints, a safe zone around the window is introduced to ensure that no anomalies are caught in the window and to eliminate the overlap. Let's define $s > l \in \mathbb{N}$ to be the size of the safe zone. Once again, iteration is performed over all fixed-size windows, but the window has to fulfill two conditions to be included in the dataset:

- No anomalies are inside the window or within $\frac{s}{2}$ points from its middle
- It doesn't overlap with any other windows included

Algorithm 2 describes the specific process to achieve that.

The values of l (window length) and s (safe zone length) that turned out to work well and were used for the model are, respectively, 20 and 50.

Algorithm 2 Generating negative datapoints

```
1: $feature\_list \leftarrow$ empty list
2: Require: $features$ : flight matrix $n_f \times L_i$
3: Require: $labels$ : copy of the label matrix $2 \times L_i$
4: Require: $l$ : window size
5: Require: $s$ : safe zone size
6: $pad \leftarrow \lfloor \frac{s-l}{2} \rfloor$
7: for $j$ in $0..(L_i - l + 1)$ do
8:     $window \leftarrow labels[j : j + l]$
9:     if $max(window) == 0$ then
10:        $feature\_list.push(features[j + pad : j + l - pad])$
11:        $labels[j : j + s] \leftarrow 1$
12: return $feature\_list$
```

The next chapter will provide a theoretical description of the machine learning methods and algorithms used to solve the problem stated above, and a general framework for solving such problems, including the evaluation of a solution (model).

Chapter 4

Machine learning methods

The most general supervised learning (which is the branch of machine learning which can be of the most use in this work) problem can be expressed as finding an approximation of a certain probability distribution $p(y|x)$, based on a finite set of (x_i, y_i) datapoints, where x_i is the i-th **feature vector** and y_i is its **label**.

The feature vectors can be elements of a vector space $\mathbb{R}^n$, but some approaches allow for one model to be used with feature vectors of various dimensions.[11][12] In general, x_i can be members of a problem-specific set of vectors.

The labels can take either continuous or discrete values. In the first case, the task is called **regression**, and in the latter – **classification**. A special case of classification is **binary classification**, where the labels take one of two possible values, $y_i \in \{0, 1\}$. In **multiclass classification** the set of possible labels is finite, with a cardinality larger than 2, and in **multilabel classification**, the labels are binary vectors, i.e. $y_i \in \{0, 1\}^n$ for some $n \in \mathbb{N}$.

Some examples include:

- For detecting the presence of a cat in a 32×32 RGB photo, the features are real vectors $x_i \in \mathbb{R}^{32\times32\times3} \cong \mathbb{R}^{3072}$ and the labels are binary values $y_i \in \{0, 1\}$, making it a binary classification problem.

- For time-series prediction, the features can be real-valued sequences of varying length, and the labels are real values corresponding to the values at the next timestep, $y_i \in \mathbb{R}$

- For text summarization[1] (shortening a long text to a shorter one, conveying as much of the original information as possible), the features and the labels could be sequences of real vectors corresponding to word embeddings[13], $(x_i)_t \in \mathbb{R}^e$, where $e \in \mathbb{N}$ is the size of the embeddings.

A machine learning model is an algorithm that can learn from a dataset consisting of (x_i, y_i) pairs, and make correct label predictions about new feature vectors from the same distribution.

A model can be either **parametric** or **nonparametric**. A parametric model can be described as a mapping

$$f : X \times \Theta \longrightarrow Y$$

where X and Y are respectively the sets of feature vectors and labels, and $\Theta \subset \mathbb{R}^p$ is the set of parameter vectors. Parameter vectors are essentially vectors of real numbers, with a fixed dimension p. The standard notation of the model's prediction is

$$\hat{y}_i = f(x_i|\theta)$$

Nonparametric models are all the models that can't be described by a set of parameters – they can, for example, use the entire dataset they're trained with, directly.

One more necessary concept is a **loss function** (also known as the **cost function** or **objective function**) that measures the performance of a model and makes learning possible for parametric models. The motivating idea of a loss function is that it should measure how distant the prediction is from the actual label on a given datapoint.

Formally, a loss function can be defined as

$$\mathcal{L} : \hat{Y} \times Y \longrightarrow \mathbb{R}_{\geqslant 0}$$

where $\hat{Y}$ is the prediction space, i.e. the set of all predictions that can be made by the considered model. This isn't necessarily the same as the set of possible labels – in fact, in general it's not even true that $Y \subset \hat{Y}$ or $\hat{Y} \subset Y$. In the case of binary classification, for example, we often have $Y = \{0, 1\}$, and $\hat{Y} = (0, 1)$, so $Y \cap \hat{Y} = \emptyset$. There is however, the obvious mapping

$$L(\hat{y}) = \begin{cases} 0, & \hat{y} < 0.5 \\ 1, & \hat{y} \geqslant 0.5 \end{cases}$$

that converts predictions to labels for binary classification. Similar mappings can be defined for other prediction spaces.

The most commonly used loss function for binary classification is the binary crossentropy, also known as log-loss, defined as:

$$\mathcal{L}(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

When dealing with a parametric model f, the machine learning task can be defined as finding θ^* defined as follows

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_i \mathcal{L}(f(x_i|\theta), y_i)$$

This turns out to be a well-studied problem of function optimization that can be solved[1] by methods like gradient descent.[17]

[1]Assuming that $\mathcal{L}(f(x_i|\theta), y_i)$ is differentiable almost everywhere with respect to θ

4.1. Feedforward neural networks

Neural networks are parametric models known for their flexibility when it comes to the number of existing architectures, which is why they were chosen to be the most explored models for the purpose of anomaly detection in UFT-2 data.

The most basic neural model is a called a feedforward neural network. In essence, it's just a mapping $\mathbb{R}^n \longrightarrow \mathbb{R}^m$ for some fixed $n, m \in \mathbb{N}$, described by a set of parameters, also known as **weights**.

A deep feedforward neural network can be described as a number of consecutive layers. Each layer consists of a linear transformation and a translation, followed by a nonlinear **activation function**.[2]

Let's define $g^l \in \mathbb{R} \longrightarrow \mathbb{R}$, to be the **activation function** on the l-th layer, $n_l \in \mathbb{N}$ – the layer's **width**, $W^l \in \mathbb{R}^{n_l \times n_{l-1}}$ – its **weight matrix**, and $b^l \in \mathbb{R}^{n_l}$ – its **bias vector**. Let's also define $L \in \mathbb{N}$ to be the total number of layers in the network, excluding the first (input) layer (so there's a total of $L + 1$ weight matrices) – these are called the network's hidden layers. The neural network is a mapping $f : \mathbb{R}^{n_0} \longrightarrow \mathbb{R}^{n_L}$ defined by the following equations:

$$\begin{gathered} f(x) = a^L \\ a^l = g^l(W^l a^{l-1} + b^l),\ 0 < l \leqslant L \\ a^0 = x \end{gathered}$$

The a^l vectors are sometimes called **activations**.

Assuming the activation and loss functions are differentiable almost everywhere, the backpropagation[4] algorithm can be used to compute the exact derivative of the loss function with respect to the weights and biases, making it possible to minimize the loss with the use of gradient descent.

Some of the most commonly used activation functions include the **Rectified Linear Unit (ReLU)**, **sigmoid ($\boldsymbol{\sigma}$)** and **tanh** defined as follows:

$$\begin{aligned} \mathrm{ReLU}(z) &= \max(0, z) \\ \sigma(z) &= \frac{1}{1 + e^{-z}} \\ \tanh(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} \end{aligned}$$

4.2. Gradient descent

Even though the gradient descent algorithm isn't used in this work (and in general) directly, it's a basic version and an inspiration of the most commonly used optimization

[2]An activation function can be any function $\mathbb{R} \longrightarrow \mathbb{R}$ that is applied to a vector pointwise. There are theoretical reasons[8] for the activation function to be a nonlinear, bounded and monotonically-increasing

Algorithm 3 Gradient descent

```
1: Require: $\alpha$ : learning rate, a real number
2: Require: $f(\theta)$ : the function to be optimized
3: Require: $\theta_0$ : Initial parameter vector
4: $t \leftarrow 0$
5: while $\theta_t$ not converged do
6:     $t \leftarrow t + 1$
7:     $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
8:     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot g_t$
9: return $\theta_t$
```

algorithms, including the Adam algorithm described in Section 4.2.3.

The purpose of gradient descent is straightforward – finding the global minimum[3] of a function, given the possibility of evaluating its value and its gradient at any given point. This is achieved by selecting a random initial value and making consecutive steps in the direction opposite to the gradient at the current argument. This process is described in Algorithm 3.

4.2.1. Validation split

In order to estimate a model's performance, a necessary step is setting some of the data aside during the training. In that stage, the gradient estimate g_t is computed based on the so called **training set**, while the remaining **validation set** is used to compute the loss value that can be interpreted as a measure of the model's quality. This is done to ensure that the model is actually able to generalize to previously unseen (potentially unlabelled) datapoints.

Various train-validation splits are used depending on the problem and the size of the data. It's good for the validation set to be as large as possible so that it can give an accurate estimate of the model's performance. On the other hand, more data makes it possible for the model to generalize better to unseen examples and reduces the risk of overfitting. In very large datasets it's common for the validation set to consist of 1% or less of the data. However in the case of smaller datasets, including this work, a 10% validation set is often used.

4.2.2. Stochastic Gradient Descent

This important variant of gradient descent is related to the issue of computing the gradient estimate g_t. By default, the entire training set is used for that purpose, which is called **batch gradient descent**. In **stochastic** or **mini-batch gradient descent**, only a subset of a fixed size is used during each step. This is usually done by randomly choosing (without

continuous function, but in practice, non-bounded functions such as the rectified linear unit[14] (ReLU) are used commonly.

[3]In practice, for highly non-convex functions, a local minimum with a low enough value may suffice.

replacement) a fixed number of datapoints and repeating that process until all of them have been used exactly once. This constitutes a single epoch, and corresponds to a single cycle of the *while* loop described in Algorithm 3. This is repeated until convergence.

The advantage of such an approach is the possibility of lower memory requirements, especially in the case of very large datasets. It can also speed up the training process, since several steps of gradient descent can be taken in the time that would be required to take a single step with batch gradient descent. On the other hand, each step is made based on a noisy estimate of the gradient, which could be detrimental to the speed of the training. In practice, however, it turns out to have a positive impact (and is sometimes necessary due to the memory restrictions).

4.2.3. Adam

Algorithm 4 Adam optimization algorithm. Good default settings are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. All operations on vectors are element-wise, and β_1^t, β_2^t denote β_1 and β_2 to the power t.

```
 1: **Require:** $\alpha$ : stepsize
 2: **Require:** $\beta_1, \beta_2 \in [0, 1)$ : Exponential decay rates for the moment estimates
 3: **Require:** $f(\theta)$ : Function to be optimized
 4: **Require:** $\theta_0$ : Initial parameter vector
 5: $m_0 \leftarrow 0$
 6: $v_0 \leftarrow 0$
 7: $t \leftarrow 0$
 8: **while** $\theta_t$ not converged **do**
 9:     $t \leftarrow t + 1$
10:     $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
11:     $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
12:     $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
13:     $\hat{m}_t \leftarrow m_t \backslash (1 - \beta_1^t)$
14:     $\hat{v}_t \leftarrow v_t \backslash (1 - \beta_2^t)$
15:     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t \backslash (\sqrt{\hat{v}_t} + \epsilon)$
16: **return** $\theta_t$
```

The Adam algorithm[10] is a variant of gradient descent with an adaptive learning rate, widely used in training neural networks. It has proven to yield better results than other gradient-based optimization methods, so it was extensively used in this work to train various types of neural networks. The algorithm involves computing an exponentially weighted moving average of the gradient g_t and its square $g_t^2 = g_t \odot g_t$ (with $\odot$ being the Hadamard product of matrices), and determining θ_{t+1} based on those. By default it's used in the mini-batch variety, analogous to Stochastic Gradient Descent described in Section 4.2.2.

4.2.4. Convergence

The idea of convergence is important in the context of gradient-based learning and is used as somewhat of an umbrella term for various methods of determining how many steps of gradient descent (or another algorithm based on it, like Adam) should be made. Some of the noteworthy approaches include:

- **Early stopping**, which is also a method of regularization. During training, you keep track of the value of the loss function on the validation set, and stop the training when it starts increasing more than a predefined margin.
- **Manual stopping**, which can serve a similar purpose as early stopping, but is controlled by a human monitoring the loss values in real time
- Ending the training when the value of the loss function stops decreasing (i.e. actual convergence)
- Ending the training after a set number of epochs

4.3. Convolutional neural networks

Convolutional neural networks[11], or **ConvNets** in short, are a type of neural models used most commonly with image-like data, but can be used whenever translational invariance of the model is desired, including time series. For example, when detecting anomalies in a time series it doesn't make a difference whether the anomaly occurs in the beginning or the end of the window, which could be helpful in training the model – it doesn't have to consider solutions that don't have this symmetry.

In order to do that, a slightly different representation of the data has to be considered. In the 1D case (time series), which is the most relevant to this work, a single datapoint would be (x_j, y_j), where $x_j \in \mathbb{R}^{L \times n}$. Mathematically, the simplest way to describe a convolutional layer is through an actual discrete convolution.[4]

A convolutional layer between layers l and $l+1$ is a mapping $\mathbb{R}^{L_l \times n_l} \longrightarrow \mathbb{R}^{L_{l+1} \times n_{l+1}}$ that is described by a **kernel** serving as the weights of the layer. The kernel consists of n_{l+1} **filters**, and each filter is simply a matrix $w^l_k \in \mathbb{R}^{s_l \times n_l}$, where k is the index of the filter and s_l is the kernel size (a positive integer). Each filter is then convolved with the input feature matrix, producing a matrix $a^l_k \in \mathbb{R}^{L_{l+1} \times 1}$. The outputs from the convolutions with each filter are then concatenated, yielding an activation matrix $a^{l+1} \in \mathbb{R}^{L_{l+1} \times n_{l+1}}$. It's worth noting that due to different options of handling edge cases, it's possible that $L_{l+1} < L_l$. This issue is known as **padding**, and two approaches are dominant:

- **Same padding**, where the feature matrix is padded with zeros so that $L_{l+1} = L_l$
- **Valid padding**, where there is no padding, which results in $L_{l+1} = L_l - (s_l - 1)$

Due to its construction, a convolutional layer by itself only captures local information, i.e. it cannot learn any relations between distant timesteps. While this may be a limitation, it greatly reduces the number of parameters used in a single layer, and creates a preference for models that focus on local information, which can be desired in some applications.

4.4. Recurrent neural networks

Recurrent models form another class of extensions of the standard neural network. They are applicable whenever the input features have a temporal structure, like a (possibly multivalued) time-series and serve the purpose of leveraging that structure. They also make it possible for a model to accept inputs of varying size, which makes them perfect for problems like natural language processing.[12]

In order to define the basic recurrent layer, let's represent the data in yet another way. The input will be a sequence of constant-sized real vectors $x_t \in \mathbb{R}^n, t \in \overline{1, T}$, which is equivalent to a matrix $\mathbb{R}^{n \times T}$. The output will be a sequence of the same length[4], possibly with different dimensionality: $\hat{y}_t \in \mathbb{R}^m, t \in \overline{1, T}$. Let's also define n_h to be the size of the hidden units.

There are three weight matrices: $W_x \in \mathbb{R}^{n_h \times n}$, $W_h \in \mathbb{R}^{n_h \times n_h}$, $W_y \in \mathbb{R}^{m \times n_h}$, as well as bias vectors $b_1 \in \mathbb{R}^{n_h}, b_2 \in \mathbb{R}^m$ Activation functions g_1, g_2 are also necessary. The recurrent layer's outputs are then defined as follows:

$$\begin{aligned} h_t &= g_1(b_1 + W_{hh}h_{t-1} + W_{hx}x_t) \\ \hat{y}_t &= g_2(b_2 + W_{yh}h_t) \end{aligned}$$

The output is differentiable with respect to all weights and biases, and the gradient can be computed with a slight modification of the backpropagation algorithm, called backpropagation through time.[19]

4.4.1. Gated recurrent layers

A major drawback of recurrent neural networks is that they don't do well in capturing long distance dependencies, which is a symptom of vanishing gradients in backpropagation through time. To counteract this, the idea of gated recurrent layers was introduced, with two varieties being especially useful – **Gated Recurrent Units (GRU)**[3] and **Long Short Term Memory (LSTM)**[7] layers. They both introduce a slight modification to the relations between inputs, hidden units and outputs.

Specifically, for GRUs the defining equations are as follows:

[4]It's possible to make the length different, for example, by discarding some of the outputs

$$
\begin{aligned}
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1}) \\
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1}) \\
h'_t &= \tanh(W_{xh'}x_t + r_t \odot W_{hh'}h_{t-1}) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot h'_t \\
y_t &= \tanh(W_{hy}h_t)
\end{aligned}
$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid activation function and all of $W_{xz}, W_{hz}, W_{xr}, W_{hr}$, $W_{xh'}, W_{hh'}, W_{hy}$ are learnable matrices of appropriate dimensions. z_t and r_t are called the **update gate** and **reset gate** respectively as they control how much of the previous state is to be preserved. It's worth noting that $\forall x \in \mathbb{R}\ \sigma(x) \in (0,1)$, so in particular $z_t, r_t \in (0,1)$.

LSTMs are somewhat more complicated, with additional gates:

$$
\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \\
c_t &= f_t c_{t-1} + i_t \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \\
y_t &= o_t \tanh(c_t)
\end{aligned}
$$

where i_t is called the **input gate**, f_t – **forget gate**, c_t – **cell state**, o_t – **output gate**.

4.5. Regularization

An important issue to consider when training a neural network is the problem of generalization. In the end, we want the model to perform well on previously unseen data, but the training objective might favor just memorizing the entire training set, especially in the case of very large models (with a high number of parameters). To counteract that, several regularization methods can be used, the most commonly used of which include:

- L2 regularization, which involves adding to the loss function a term proportional to the L2 norm of all the weights
- Dropout, which involves randomly setting some of the activations to 0 during the training, which makes the training biased towards models that aren't reliant on very specific patterns

Chapter 5

Experiments and results

This chapter will describe some of the machine learning models trained and evaluated on the task described in Section 3.2.1, which is detection of jump-like anomalies in a window.

All of the following models are some variants of neural networks. The loss function function that was used is the binary crossentropy weighted by the relative occurrence frequency of the classes, optimized with the Adam algorithm, and regularized with dropout. The window size is 20 points – it's a size large enough to capture the entire phenomenon, while at the same time being small enough not to cause large data requirements (as described by the curse of dimensionality[9]). The models were implemented using Keras[2] and the metrics – using scikit-learn[16].

The dataset consisted 15917 datapoints, out of which 10% (1592) were set aside for validation and were not used for training. They were generated using Algorithms 1 and 2, with about 18% of them containing a jump anomaly, and the remaining being clear of them (although they could contain other kinds of anomalies, which the models do not consider by design).

5.1. Evaluation metrics

Due to the class imbalance (specifically, only about 18.4% of the datapoints have a positive label) in the analyzed dataset, a simple accuracy score might be misleading, since a model giving a constant negative prediction would attain an accuracy of about 82%. Thus, other metrics had to be used to evaluate a model's performance.

Let's define TP to be the true positives as predicted by the model, TN – true negatives, FP – false positives, FN – false negatives. Then we can define the following:

	Training data	Validation data
Loss	0.193	0.203
Accuracy	0.912	0.912
Precision	0.777	0.794
Recall	0.729	0.724
F1 score	0.752	0.757

Table 5.1: Results obtained with the single-feature feedforward network

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$
$$\text{recall} = \frac{TP}{TP + FN}$$
$$\text{precision} = \frac{TP}{TP + FP}$$
$$\text{f1 score} = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}}$$

5.2. Normalization

To make the training of the models easier, the data was normalized. The temperature vectors were transformed by substracting the value at the first timestep of the window. Wind velocities were transformed by substracting the mean and dividing by the standard deviation of each component over the entire dataset. The angles (in radians) and the liquid water content were not normalized, since all of their values are in the range $[0, \frac{\pi}{2}]$.

5.3. Single-feature models

These models used only a single record – the temperature on the lower UFT-2. In principle, it should be enough, since the original labelling was based solely on it.

5.3.1. Feedforward network

In this model, the temperature vector was processed by a standard feedforward neural network consisting of four ReLU-activated layers of sizes (50, 100, 100, 100), followed by a sigmoid-activated output layer of size 1. After training it for 400 epochs, it reached the loss and metric values as shown in Table 5.1.

5.3.2. Convolutional network

In this model, the temperature vector was processed by 1-dimensional ReLU-activated convolutional layers with kernel size 3, and with layer sizes (20, 40, 50, 50), followed by a

	Training data	Validation data
Loss	0.180	0.183
Accuracy	0.922	0.924
Precision	0.834	0.845
Recall	0.715	0.737
F1 score	0.770	0.787

Table 5.2: Results obtained with the single-feature convolutional network.

	Training data	Validation data
Loss	0.172	0.180
Accuracy	0.926	0.925
Precision	0.830	0.826
Recall	0.747	0.766
F1 score	0.786	0.795

Table 5.3: Results obtained with the single-feature LSTM network.

sigmoid-activated dense output layer of size 1. After training it for 400 epochs, it reached the loss and metric values as shown in Table 5.2.

5.3.3. Recurrent networks

Two recurrent models were used here - identical with the exception of the recurrent layer being either LSTM or GRU. They consist of tanh-activated recurrent layers maintaining the length of the time series with sizes (20, 50, 50), followed by a sequence-to-vector tanh-activated layer of size 30 and a sigmoid-activated output dense layer of size 1. After training the LSTM network for 400 epochs and the GRU network for 370 epochs, they reach the loss and metric values as shown in Table 5.3 (LSTM) and Table 5.4 (GRU).

[1] In principle, the model could learn to compute the angles based on the velocity vector, however this approach could make the training easier since the model doesn't have to learn a relatively complicated function by itself.

	Training data	Validation data
Loss	0.159	0.172
Accuracy	0.931	0.928
Precision	0.829	0.810
Recall	0.788	0.816
F1 score	0.807	0.813

Table 5.4: Results obtained with the single-feature GRU network.

	Training data	Validation data
Loss	0.223	0.239
Accuracy	0.903	0.898
Precision	0.836	0.851
Recall	0.585	0.563
F1 score	0.689	0.677

Table 5.5: Results obtained with the multi-feature feedforward network.

	Training data	Validation data
Loss	0.253	0.272
Accuracy	0.887	0.883
Precision	0.720	0.734
Recall	0.628	0.609
F1 score	0.671	0.665

Table 5.6: Results obtained with the multi-feature convolutional network.

5.4. Multi-feature models

These models utilised a more complete record, specifically, both temperature vectors, three components of the velocity of the wind, two angles describing the angle of the wind[1], and the liquid water content.

5.4.1. Feedforward network

In this model, the input features were flattened into a vector to be processed by a standard feedforward neural network consisting of four ReLU-activated layers of sizes (50, 100, 100, 100), followed by a sigmoid-activated output layer of size 1. After training it for 400 epochs, it reached the loss and metric values shown in Table 5.5.

5.4.2. Convolutional network

In this model, the features were processed by 1-dimensional ReLU-activated convolutional layers with kernel size 3, and with layer sizes (20, 40, 50, 50), followed by a sigmoid-activated dense output layer of size 1. After training it for 300 epochs, it reached the loss and metric values as shown in Table 5.6.

5.4.3. Recurrent networks

Again, two recurrent models were used here - identical with the exception of the recurrent layer being either LSTM or GRU. They consist of tanh-activated recurrent layers maintaining the length of the time series with sizes (20, 50, 50), followed by a sequence-to-vector tanh-activated layer of size 30 and a sigmoid-activated output dense layer of size 1. After

	Training data	Validation data
Loss	0.132	0.173
Accuracy	0.940	0.932
Precision	0.865	0.850
Recall	0.800	0.783
F1 score	0.829	0.815

Table 5.7: Results obtained with the multi-feature LSTM network.

	Training data	Validation data
Loss	0.137	0.167
Accuracy	0.943	0.933
Precision	0.885	0.872
Recall	0.791	0.760
F1 score	0.835	0.812

Table 5.8: Results obtained with the multi-feature GRU network.

training the LSTM network for 400 epochs and the GRU network for 370 epochs, they reach the loss and metric values as shown in Table 5.7 (LSTM) and Table 5.8 (GRU).

Chapter 6

Summary

Accuracy	Feedforward	Convolutional	Recurrent LSTM	Recurrent GRU
Single feature	0.912	0.924	0.925	0.928
Multi feature	0.898	0.883	0.932	0.933

Table 6.1: Accuracy values on the validation set for various models.

F1 score	Feedforward	Convolutional	Recurrent LSTM	Recurrent GRU
Single feature	0.757	0.787	0.795	0.813
Multi feature	0.677	0.665	0.815	0.812

Table 6.2: F1 score values on the validation set for various models.

In Chapter 2 I described the physical context of the obtained data and the problems within it. Then, in Chapter 3 I built a precise mathematical description of the problem I'm trying to solve in this work, which is detecting anomalies in this signal. In chapter 4 I provided an overview of the machine learning methods used in solving that problem. Finally, in chapter 5 I described the specific algorithms I used for that purpose and evaluated their performances.

A summary of the results described in the previous chapter can be seen in Tables 6.1 and 6.2. Accuracy is shown due to its simplicity and easy interpretability, even though it is flawed when dealing with an imbalanced dataset. F1 score was chosen for the final comparison because it is often used to counteract exactly that problem, and provides a good balance between precision (i.e. trying to make the anomaly detections correct as often as possible) and recall (i.e. trying to detect as many anomalies as possible).

Perhaps surprisingly, some models seem to work better with only the temperature vector. This is most likely related to the aforementioned curse of dimensionality – the amount of data necessary to build a model grows exponentially with the size of the features, which is problematic with a limited dataset. If more labelled data was available, larger models could be trained and they would be able to effectively leverage (or discard) the

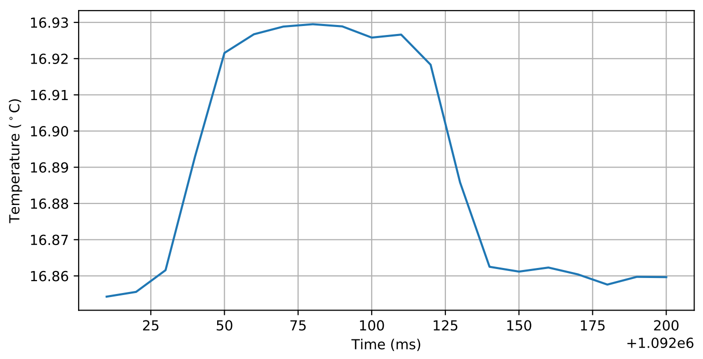

Figure 6.1: An example of a false negative in models' predictions.

additional dimensions of the datapoints. However, this still suggests that pretty much all the information needed to detect an anomaly is within the temperature signal itself. Wind speed can be useful due to some possible correlations with the anomalies, but in the end, it's not the most predictive feature.

Regardless of whether one or multiple features were used, recurrent models generally performed better than feedforward or convolutional ones. Feedforward networks are not able to consider the temporal structure of the data, so they face a more difficult task and need to learn similar features in various positions in time. Convolutional networks, on the other hand, maintain the temporal invariability, but are unable to consider non-local events, which could be necessary in some cases – for example, anomalies that are characterized by the temperature value being different at the beginning and the end of the window.

Recurrent networks, particularly LSTMs, were designed to solve both of these problems – they include the temporal structure and are able to capture long-term dependencies.

Thus, it can be concluded that artificial intelligence techniques, especially recurrent neural networks, can be of use in detecting the anomalies in ACTOS (and similar) data, and also that in this particular scenario, the UFT-2 signal itself can suffice for that goal. What's more, it turns out that for the classification of this specific dataset, the temperature measurement itself is sufficient and doesn't require additional features to attain high accuracy scores.

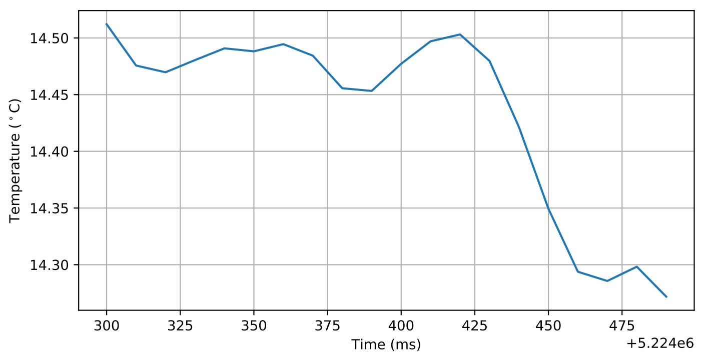

Figure 6.2: An example of a false positive in models' predictions.

Some other valuable insights may be obtained by investigating the kinds of datapoints that the models got wrong. In Figure 6.1 you can see a segment of the temperature data with an anomaly that was mislabelled by both the single-feature feedforward and GRU models, and in Figure 6.2 – a segment of the temperature data which wasn't labelled as an anomaly during the manual labelling, but was picked up by both of those models.

While it's difficult to tell the exact cause those of mistakes, a reasonable guess might be that in Figure 6.1, two opposite jumps occur not far from one another, but not close enough for it to be considered a 'returning' jump. The pattern in Figure 6.2 is more interesting – upon reexamining the data in a fuller context, it turns out to be a region that is difficult to classify, even manually. It is possible that it does in fact correspond to an anomaly in the measurements, in which case it could be said that the student has outgrown its teacher.

Bibliography

[1] M. Allahyari, S. Pouriyeh, M. Assefi, S. Safaei, E. D. Trippe, J. B. Gutierrez, and K. Kochut. Text Summarization Techniques: A Brief Survey. *CoRR*, arXiv: 1707.02268, July 2017.

[2] F. Chollet et al. Keras. https://keras.io, 2015. Access on 14.6.2018.

[3] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, arXiv: 1412.3555, 2014.

[4] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[5] K. E. Haman, A. Makulski, S. P. Malinowski, and R. Busen. A new ultrafast thermometer for airborne measurements in clouds. *Journal of Atmospheric and Oceanic Technology*, 14(2):217–227, 1997.

[6] F. Henrich, H. Siebert, E. Jäkel, R. A. Shaw, and M. Wendisch. Collocated measurements of boundary layer cloud microphysical and radiative properties: A feasibility study. *Journal of Geophysical Research: Atmospheres*, 115(D24):148–227, 2010.

[7] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[8] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[9] E. Keogh and A. Mueen. *Encyclopedia of Machine Learning and Data Mining.* Springer US, Boston, MA, 2017. p. 314-315.

[10] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, arXiv: 1412.6980, 2014.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[12] Z. C. Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, arXiv: 1506.00019, 2015.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. *CoRR*, arXiv: 1301.3781, Jan. 2013.

[14] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[15] J. Nowak, W. Kumala, J. Kwiatkowski, K. Kwiatkowski, D. Czyzewska, K. Karpinska, and S. Malinowski. Geophysical research abstracts. Vol. 20:EGU2018–12492, 2018.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[17] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, arXiv: 1609.04747, 2016.

[18] H. Siebert, H. Franke, K. Lehmann, R. Maser, E. W. Saw, D. Schell, R. A. Shaw, and M. Wendisch. Probing finescale dynamics and microphysics of clouds with helicopter-borne measurements. *Bulletin of The American Meteorological Society*, 87:1727–1738, 2006.

[19] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990.