POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Master's Degree Thesis

An automated method for image classification by selecting important regions applied to a high-resolution meteor imagery dataset: a path to real-time meteor classification.

Supervisor
prof.ssa. Elena Maria Baralis

Candidate
Alessandro Nicolini

Co-supervisor
Andrea Novati
Daniele Gardiol

July 2022

Summary

The formation of the Solar System can be traced back to 4.5 billion years ago. The original cloud of interstellar gas and dust began to collapse, leading to the formation of the Sun and then all the other celestial bodies. Some of them did not evolve into planets and maintained an almost completely pure state. They are called *small bodies* of our Solar System and include *asteroids*, *comets* and *meteoroids*. Some of these bodies can enter the Earth's atmosphere and cause the striking phenomenon of *meteor*; if a fraction of the body can reach the surface of the planet it is called *meteorite*. The study of meteorites is very important because these fragments of small celestial bodies reveal much about the history of our Solar System, and their classification based on structure and composition provides information about what kind of objects exist in our system. These objects may have unusual orbits that pose a threat to life on Earth. Thus, the more information we have, the better we can develop techniques to deflect potentially dangerous bodies. Camera networks such as PRISMA have emerged with the precise goal of observing bright meteors and determining the fall trajectory of the object causing the visible phenomenon in order to delineate the impact area. The currently working detection system is based on an open-source project called FRIPON/freeture, which implements a motion detection algorithm. Due to the high number of false positives generated by the system, a different solution is needed. As part of the PRISMA project, we aim to develop a neural network for real-time detection of transient atmospheric events caused by meteoroids entering the atmosphere. To accomplish this task given the high resolution of the images collected by the cameras and the small size of the target object, the key strategy is to use a model or combination of models capable of extracting the informative regions on the input and using only these to perform a binary classification task. Using these guidelines, two different families of models are compared: *Deep Attention-Sampling* and *Differentiable Patch Selection* models are designed to learn which regions are important in the input images and extract some square patches to perform the binary classification task. At first, the performance and limitations of these models are evaluated on two synthetic datasets called *needle MNIST* and *megapixel MNIST*, both based on the famous MNIST dataset of handwritten digits; then the best solution for the task-specific dataset is investigated. From an initial dataset of video recordings 38 fireball events and 38 non-fireball events are selected, then 3 events for each class are excluded for the final test, and two datasets are created. The first dataset consists of manually cropped 56x56 images representing parts of events and is used to find the best *feature extraction* network, the part of the models that uses the important regions (patches) to classify the whole input image. The training strategy is 7-fold cross-validation, and several techniques are tested to deal with the class imbalance problem: class weights, oversampling, focal loss function, transfer learning and fine

tuning on FASHION MNIST. The weights of the best architecture trained with the best strategy are stored and then loaded into the final models. The second dataset consists of 1296x966 images created from 2s windows with an overlap of 1/4s. The final models are trained on this second dataset using a train validation test strategy and the oversampling method. Both datasets are augmented by a factor of 8 using the same offline data augmentation. One problem that mainly affects the second dataset is the high number of noisy positive class images generated from the original events. Of the two models, the Deep Attention-Sampling model appears to give the best results, but a more detailed analysis reveals its weaknesses: the number of noisy positive class images prevents learning the correct informative regions, and the model associates the noisy extracted patches with the positive class.

Ringraziamenti

Un ringraziamento speciale va soprattutto alla mia famiglia, che malgrado le avversità di questo ultimo periodo è stata sempre presente, e nel corso degli anni mi ha sempre appoggiato e guidato nelle mie scelte. Un grazie va anche ad N3, guidata dal tutor aziendale Andrea Novati, che mi ha permesso di poter sviluppare questo lavoro nell'ambito del progetto PRISMA coordinato dall'INAF, e di confrontarmi con una problematica reale, e tutte le implicazioni che ne conseguono.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence. 8, 9

ATS Deep Attention-Sampling. i, ii, 37, 40, 49, 54, 56, 57, 63–67, 71, 74

CAMS Cameras for All-sky Meteor Surveillance. 32

CNN Convolutional Neural Network. 7, 16, 17, 21, 22, 33

DFN Desert Fireball Network. 31

DL Deep Learning. 8, 10–12

DPS Differentiable Patch Selection. i, 42, 45, 54, 56, 63, 64, 66, 67, 71, 74

FITS Flexible Image Transport System. 6, 51

FNN Feedforward Neural Network. 12, 20–22, 32

FRIPON Fireball Recovery and InterPlanetary Observation Network. 5

INAF Istituto Nazionale di Astrofisica. 5

ML Machine Learning. 8, 10

MLP MultiLayer Perceptron. 13, 14, 17

OOM Out Of Memory. 62

PRISMA Prima Rete Italiana per la Sorveglianza sistematica di Meteore ed Atmosfera. i, 5

ReLU Rectified Linear Unit. 21

SGD Stochastic Gradient Descent. 15

TLE Transient Luminous Event. 5

Chapter 1

Introduction

This first chapter presents some information and concepts necessary to contextualize the work. First, a brief overview of the history of our Solar System is given to introduce the components that characterize it. Then we focus on the small bodies, in particular asteriods and meteoroids, highlighting why they are so important. The information to write this part are taken mainly from the NASA website [27] and the astonomy book [22]. Finally the PRISMA project is described.

1.1 Brief history of the Solar System

The planetary system in which we live is called the *Solar System* and is located in an outer spiral arm of our galaxy, the *Milky Way*. Modern astronomers date its formation back to 4.5 billion years ago and claim that during the first phase of the solar nebula's formation, there was a cloud of interstellar gas and dust that began to spin and collapse, possibly due to the shock wave of an exploding star called a *supernova*. At the center of the swirling disk, increasing gravity attracted more and more matter, leading to the formation of the *Sun*, which reached the minimum temperature necessary to trigger nuclear fusion reactions, releasing an enormous amount of energy. Even though the Sun eventually absorbed more than 99% of the available matter, the remaining material further out in the disk also clumped together, forming the *planetesimals*[1]. Some of them continued to grow to their present size, accumulating the smaller masses and becoming large moons, dwarf planets and planets. In other cases, the planetesimals did not evolved into planets: the *Asteroid Main Belt*, the *Kuiper Belt*, and the *Oort Cloud* are made up of pieces of the early Solar System that never quite merged into a planet. These celestial bodies can be divided into two categories: some of them are composed mainly of metals and rocky material and form the asteroids, while others are composed mainly of dust and ice and form the comet nuclei.

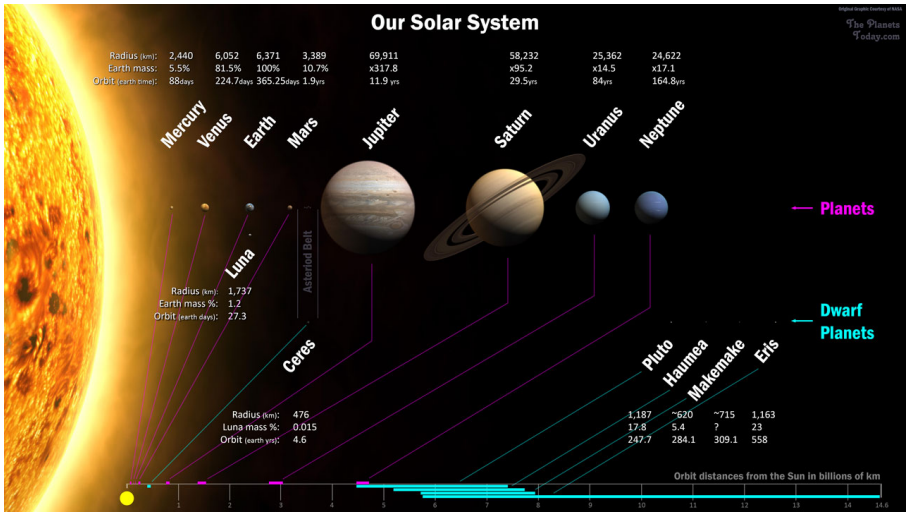


Figure 1.1: Solar system map

The planets that formed in our Solar System are divided into *rocky planets* (Mercury, Venus, Earth and Mars) and *gas planets* (Jupiter, Saturn, Uranus, Neptune). The first category developed in a region of the early solar nebula where the proximity of the Sun suppressed the existence of volatile elements. The second category, on the other hand, is characterised by low mass density and the distance from the Sun allowed these planets to keep massive gas envelopes enclosing their rather small solid cores. In addition to the actual planets there are the dwarf planets. The five best known dwarf planets are Ceres, Pluto, Haumea, Makemake and Eris. With the exception of Ceres, which is in the asteroid belt, the others are in the Kuiper

[1] bodies of solid materials formed in the earliest planetary nebulae that have grown large enough to begin to draw other materials to them by gravitational attraction

belt. They are called dwarf planets because, although they are massive and orbit the Sun, they have not cleared their orbital path compared to the proper planets. The following image provides a view of the Solar System.

1.2 The small bodies of the solar system

The small bodies of our Solar System, namely *asteroids*, *comets* and *meteoroids*, play an important role. As stated earlier, billions of small space rocks did not evolve into planets during the formation of the Solar System, and amazingly, they have maintained an almost completely pristine state during these 4.5 billion years. The importance of studying these small worlds lies in several reasons:

- they tell a lot about the history of our Solar System and provide us with useful information to reconstruct the events that led to the formation of our world. The discovery of the *amino acid glycine*[2] in comet dust supports the theory that some catalyst substances of life arrived on Earth through meteorite and comet impacts;
- their study lets us understand how human expansion in space can be affected by the available resources and the hazards in the Solar System;
- most important is the study of their composition and structure to classify what kind of objects exist in the Solar System. This is because it is known that asteroid and comet impacts may be one of the main causes of the mass extinction events on our planet. These bodies can have unusual orbits and some of them pose a threat to life on Earth. So the more we know about them, the better we can track them and develop techniques to deflect the potentially dangerous bodies.

[2]amino acid used by living things to synthesize proteins

1.3 Meteoroids, meteors and meteorites

All this information can also be gathered directly from where we are, simply by examining the remains of the impact of small "space rocks". It is now necessary to introduce some vocabulary to make things clear.

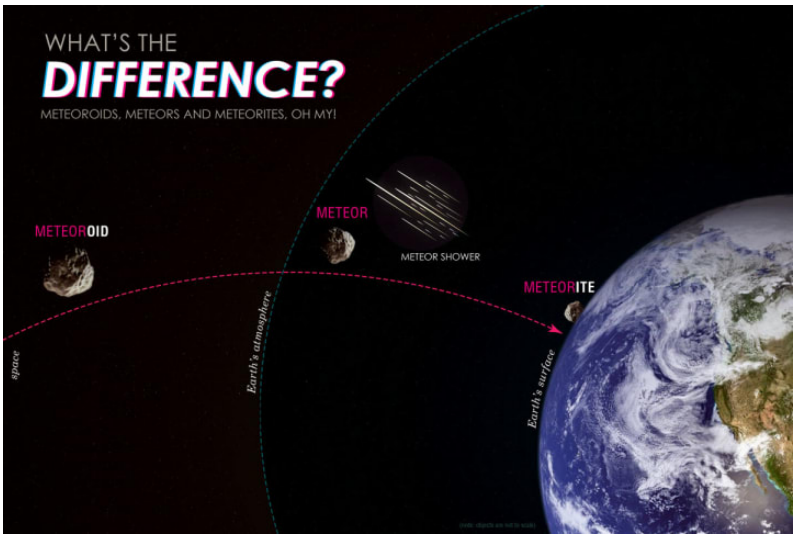


Figure 1.2: Meteoroid, meteor and meteorite are referred to the same space body at different stages of its falling

Meteoroids are objects in space ranging in size from dust grains to small asteroids that could crash into our planet as they orbit. If a meteoroid crashes into the Earth's atmosphere, friction with the air creates the *meteor* phenomenon commonly known as "shooting stars". The meteor phenomenon results from the body's friction with the air: during its fall to the planet's surface, the friction with the increasingly dense atmospheric layers causes the meteoroid to overheat. The kinetic energy is converted into thermal energy, which is distributed between the body itself and the atoms of the atmosphere. The energy released is sufficient to ionize a large number of atoms, leaving the body with an ion tail as it falls. The process of recombination of ions and electrons whose charges are opposite produces the luminous phenomenon we can observe. Because the recombination takes some time, the bright tail can last for a few seconds. Less cohesive meteoroids may break up into several blocks, each of which generates a different meteor. If the original object is large enough and the velocity is not too high, a portion of it may reach the Earth's surface. When the velocity of the body drops below 3-4 km/s, a cooling process begins and the emission of visible radiation stops, the body enters the so-called dark flight phase. What survives the fall is called a *meteorite*. A meteorite is usually no more than 5% of the original object, and its size varies usually from that of a pebble to that of a fist. If the meteor is at least as bright as Venus, it is called a *fireball* or *bolide*. Sometimes, in the case of small asteroids with a diameter larger than 10 m (extreme cases), the meteor phenomenon may be particularly bright (brighter than the Sun) and the meteoroid may completely disintegrate or even explode during the fall; in this case, the meteoroid is called a *superbolide*.

1.4 PRISMA and FRIPON projects

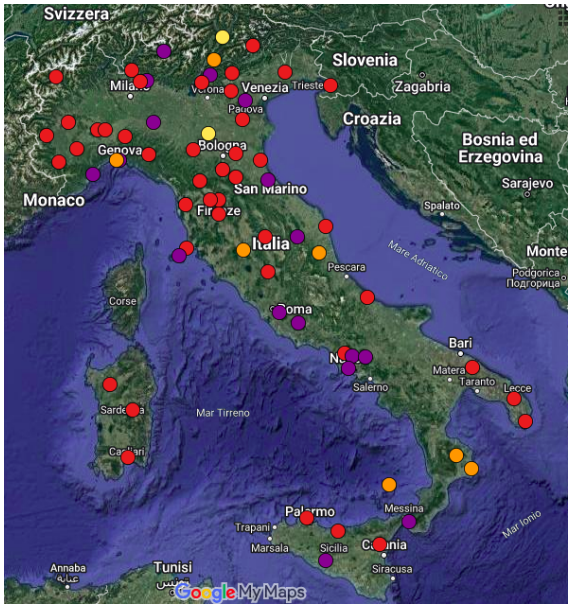


Figure 1.3: PRISMA station locations in Italy.
red: currently working station
purple: station under maintenance
yellow: station being installed
light yellow: station being purchased

The PRISMA [28] network was launched in 2016. The acronym stands for *Prima Rete Italiana per la Sorveglianza sistematica di Meteore ed Atmosfera*, meaning First Italian Network for the Sistematic Monitoring of Meteors and Atmosphere, and is led by INAF, the National Institute for Astrophysics. The project aims to realize an Italian network of all-sky cameras to observe bright meteors (fireballs and bolides) and to determine the orbit of the objects that cause these phenomena, to delineate the impact areas with a high degree of precision [2] in order to recover the meteorites (if any) and, at the same time, to progressively increase the number of monitoring stations throughout the Italian territory to ensure a complete coverage of the sky. The distance between two stations of the network is maximum 80/100 km to ensure that the network works properly and covers the whole country. The network sistematic monitoring action can also be used to collect data about cloud cover and *Transient Luminous Events*[3] (TLEs) in the atmosphere in order to validate meteorologic models. Anyone can participate in this project, which involves research centers, universities, groups of amateur astronomers, astronomical and meteorological observatories, schools, etc., but purchase of the station's hardware is required. PRISMA is a partner of FRIPON [26][3], (Fireball Recovery and InterPlanetary Observation Network), an international European collaboration. It was launched in 2014 and is led by l'Observatorire de Paris, the Muséum National d'Histoire Naturelle, the Université Paris-Sud, the Université Aix Marseille, and the Centre National de la Recherche Scientifique.

The station hardware includes the *Basler ace acA1300-30gm*[4] camera sensor and a mini-PC called *PRISMA node*. The camera is connected to the PC via a 1GB Ethernet interface and transmits images at 30fps. In addition to a calibration process, the node also collects images in a buffer and processes them when an event is detected. The detection software currently used is based on the open source project *freeture*[5]: the software was developed by FRIPON, supports various camera types,

[3]Family of short-lived electrical-breakdown phenomena that occur well above the altitudes of normal lightning and storm clouds.

[4]Camera datasheet can be downloaded from the following link `https://www.baslerweb.com/en/downloads/document-downloads/basler-ace-aca1300-30gm-emva-data/`, accessed 28/06/2022.

[5]`https://github.com/fripon/freeture`, accessed 28/06/2022

and produces output in FITS[6] format. It detects motion by subtracting adjacent frames, but produces a significant number of false positives.

[6]FITS (Flexible Image Transport System) is a portable file standard widely used in the astronomy community to store images and tables.

Chapter 2

Artificial Intelligence and Convolutional Neural Networks

The goal of this chapter is to introduce concepts related to artificial intelligence, emphasizing in particular the role of machine learning and deep learning. First, the goal of machine learning and the main types of learning are described. Then, the first model inspired by the human brain will be the basis for further exploration of concepts related to deep learning, focusing on the classification task. In this setting, Convolutional Neural Networks, abbreviated as CNNs play the most important role, and their main components will be analyzed. Towards the end of the chapter, we will focus more on the training side: how the training dataset can be partitioned to correctly evaluate the model, the problem of overfitting the training data, what evaluation metrics can be used in a binary classification task and some of the best known regularization techniques that help in mitigating the overfitting of the training data. The information to write this chapter are mainly drawn from the deep leaning book [7], the machine learning books [19] and [23] and other web resources that will be accordingly cited.

2.1 Overview on AI

Artificial Intelligence, usually abbreviated AI, is a broad scientific field of research that seeks to develop machines that resemble human-like intelligence. A more technical definition of AI is "the field of study of *intelligent agents*", which is any system that perceives its environment and performs actions that maximize its chances of achieving its goals [29]. Since their beginnings in the 1950s, AI applications have been able to successfully solve problems that are difficult for humans but easy for machines because they can rely on a set of mathematical rules. In contrast they have a much harder time solving tasks that are easy for humans, and the reason is that they are difficult to describe formally (e.g. recognizing faces in images). Performing these intuitive actions requires an enormous amount of knowledge about the world, and machines must be able to acquire that knowledge in order to make intelligent decisions. The subfield of AI that aims to mimic the human cognitive ability to "learn" is referred to as *Machine Learning*, abbreviated as ML. Roughly speaking, the process of learning for a machine involves extracting patterns from raw data, whose *representation* strongly influences how ML algorithms work. By representation is meant any piece of information that is associated with the same real-world entity that the data describes. This information is commonly known as *features*. In figure 2.1, we can see how formulating the same problem using a different representation makes it solvable.

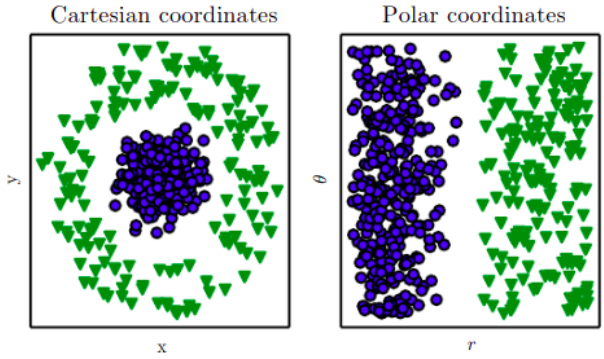


Figure 2.1: Same data are represented using cartesian coordinates (left) and polar coordinates (right). If we wanted to draw a line that separates the two classes, it is impossible with the cartesian representation.

Many tasks require the use of features that are unknown in advance and cannot be designed by hand. The solution to this problem is an approach called *representation learning*, which consists of using ML algorithms to learn how to extract a good set of features from the data. This idea forms the basis for *Deep Learning*, abbreviated as DL, a subfield of ML that has become one of the most promising areas of research. The main idea is to create a hierarchy of representations where more abstract features that encode complex concepts are expressed in terms of features that encode simpler concepts ([7] pp. 1-8). DL has many practical applications, such as human speech understanding (Siri and Alexa), recommendation systems (Amazon, Netflix), self-driving cars (Tesla), medical diagnosis, etc., but it is not a new concept [29]. Indeed, its popularity has seen ups and downs throughout its history, and the main reason for that is the lack of data and hardware resources,

which is no longer a problem today. The various AI approaches presented in this section are summarized in figure 2.2.

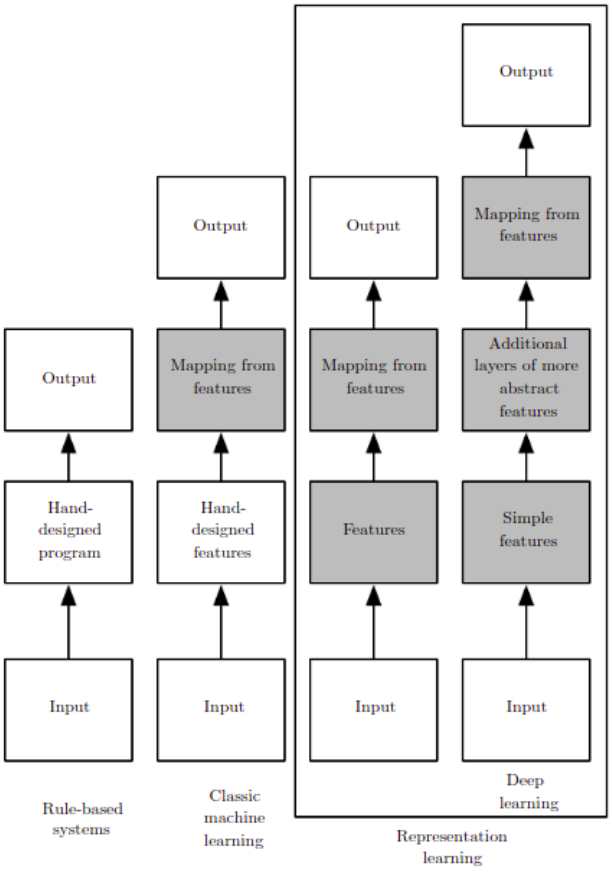


Figure 2.2: Summary of the AI approaches. Each approach is splitted into multiple functional blocks. The grey blocks are intended to be parts that learn from data.

2.2 Machine Learning
Learning frameworks and tasks

ML comprises the whole set of methods we can use to discover patterns in data, and use these patterns to predict future data. It is usually divided into two different approaches ([19] pp. 1-3):

- **supervised learning**: as stated earlier, we define the learning process as "using expertise to gain experience". In the context of supervised learning this means that the environment acts as a supervisor, providing additional information, missing at test time, to the already existing "experience" (the training examples) ([23] pp. 22-23). The acquired expertise in this setting is aimed to predict this missing information. Formally, the goal is to learn $\hat{f}$, the approximation of an unknown underlaying mapping f from inputs $\mathbf{x}$ to outputs y, given a labeled dataset of pairs $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, where $\mathcal{D}$ is the training set, N is the number of training examples, and the additional information is given by the labels y_i. Label availability means we can define a metric to compare predictions to the actual label and measure an error. In practice each input $\mathbf{x}_i$ is fed into the learning algorithm which produce an output $\hat{y}_i = \hat{f}(\mathbf{x}_i)$ and is able to update the input/output relationship $\hat{f}$ according the error measured between $\hat{y}_i$ and y_i. Model training is guided by the reduction of the error metric, and eventually it is hoped that the predictions and real-world results will be close enough for the learning algorithm to be useful for all input sets that may be encountered in the real-world application the model was trained for (that is called *generalization*).

- **unsupervised learning**: in this context, we are only provided with the training samples $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^{N}$ and the goal of the algorithm is to discover knowledge by finding interesting patterns in the data. With respect to the supervised learning setting the problem is much less well defined, as a matter of fact we do not know in advance what kind of patters we are looking for, and there is no obvious metric that can be used to evaluate the errors.

Supervised learning is the framework in which the work of this thesis is developed. In particular, a learning algorithm performs a different task depending on the nature of the output variable: when y_i is categorical ($y_i \in \{1, \ldots, C\}$) the problem is called *classification*, and when y_i is real-valued, the problem is called *regression*. The classification task is the one we are interested in: when $C = 2$ it is called *binary classification* and usually $y_i \in \{0, 1\}$, when $C > 2$ and the classes are mutually exclusive it is called *multi-class classification*, and when $C > 2$ and the classes are not mutually exclusive it is called *multi-label classification*.

Now that we have an idea about the concepts of ML and DL, we can dive into the latter one, starting from the first intuitions that inspired this type of models.

2.3 The roots of Deep Learning

2.3.1 Perceptron

The history of DL begins in the 1940s-1960s, when it was addressed by the name *cybernetics*. During that time, the first artificial neuron was created, a mathematical model inspired by the structure of biological neurons. It is far from being a realistic formalization of a real neuron, but it is an example of how reverse engineering can be used to formalize the concept of intelligence. In 1958, Frank Rosemblatt implemented the *perceptron*, a trainable single artificial neuron inspired by the biological learning theories of Warren McCulloch and Walter Pitts.

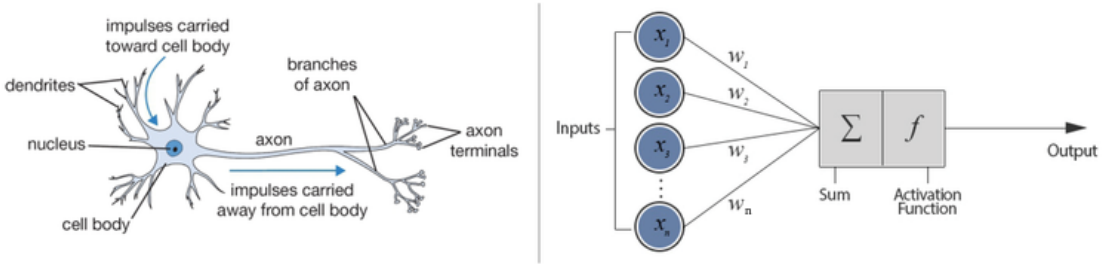


Figure 2.3: Comparison between the structure of a biological neuron (on the left) and an artificial neuron (on the right).

The perceptron model is a precursor to the *sigmoid neuron*, which is the main component of neural networks, but for now we will stick with the former. The model takes multiple real value inputs $x_1, \ldots, x_n$, and outputs a single binary value. Computing the output involves introducing weights $w_1, \ldots, w_n$ that express the importance of the corresponding input. In algebraic terms, the decision rule is:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq \text{threshold}, \\ 1 & \text{if } \sum_i w_i x_i > \text{threshold} \end{cases} \tag{2.1}$$

If we change the values of the weights and the threshold, we can tailor the decision rule to our needs. We can make the decision rule clearer by using vector notation and moving the threshold to the right-hand side of the inequality and replacing it with a term known as perceptron *bias*. The perceptron learning rule is rewritten as follows:

$$\text{output} = f(\mathbf{w} \cdot \mathbf{x} - b) = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} - b \leq 0, \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} - b > 0 \end{cases} \tag{2.2}$$

Where f is called *activation function* and in this case it is the step function. The perceptron lies in the category of *linear models* and performs a binary classification task. Given a dataset whose elements are linearly separable, tuning the weighting values and bias means finding the expression of a particular hyperplane capable of linearly separating the data into two classes. The solution is not optimal, but if the data is linearly separable, it can be found algorithmically. The bias term can also be included as w_0 in the weight vector and the f argument can be rewritten as $\mathbf{w}' \cdot \mathbf{x}'$ where $\mathbf{w}' = [w_0, w_1, \ldots, w_n]^\top$ and $\mathbf{x}' = [1, x_1, \ldots, x_n]^\top$.

Algorithm 1 Perceptron Learning Algorithm

```
P ← inputs with label 1;
N ← inputs with label 0;
Initialize w′ randomly
while !convengence do
    Pick x′ ∈ P ∪ N;
    if x′ ∈ P and w′ · x′ < 0 then
        w′ = w′ + x′;
    end if
    if x′ ∈ N and w′ · x′ ≥ 0 then
        w′ = w′ − x′;
    end if
end while▷ The algorithm converges when all the inputs are classified correctly
```

The weights can be initialized randomly or with zeros, and the algorithm essentially modifies the separable hyperplane to avoid misclassification. The final solution $\mathbf{w}$ is a vector of weights that is a linear combination of all misclassified data points.

2.3.2 The sigmoid neuron and the MultiLayer Perceptron

The second important phase in the history of DL is in the 1980s-1990s, when it was addressed under the name *connectionism*. The main idea of connectionism is that a large number of simple computational units can achieve intelligent behaviour when networked together. To create a network of artificial neurons, called *Feedforward Neural Network*(FNN)), we can group perceptrons into different layers and use the output of a previous layer as input to the following layer, as shown in figure 2.4.

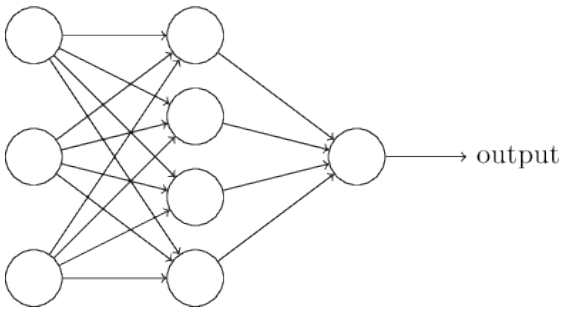


Figure 2.4: Example of an Artificial Neural Network. The first layer is called *input layer*, the last layer is called *output layer*, all the layers in between are called *hidden layers* (the middle one in the picture).

The input layer simply takes the inputs and passes them to each artificial neuron in the hidden layer. The output layer is represented by only one neuron for simplicity. The information in the network flows in a single direction, from input to output, without returning. In fact, the network is represented as a *Directed Acyclic Graph*. Intuitively, combining multiple perceptrons in a network structure leverages the contribution of each of them to perform more complex tasks. For our network to be trainable, a small variation in the value of each weight and bias should result

in a small variation in the output. In this way, we can slowly tune the weights of the network to achieve the desired output behavior. The problem with the perceptron model is that the step function activation can result in a completely unpredictable change in the output when the value of a parameter is changed slightly. This leads to the definition of a smooth activation function, which has the nice property of being derivable. This activation function is called *sigmoid* and an artificial neuron with this type of activation is called *sigmoid neuron*. The sigmoid neuron behaves such as the perceptron but now the computed output is given by the sigmoid function

$$\sigma(\mathbf{w} \cdot \mathbf{x} - b) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} - b)}} \tag{2.3}$$

and is a continuous value in the range (0, 1). Not only does this activation allow for small variations of the output given small variations of the network parameters, but from a practical point of view, the mathematical properties of the exponential function are useful when computing partial derivatives during the learning process.

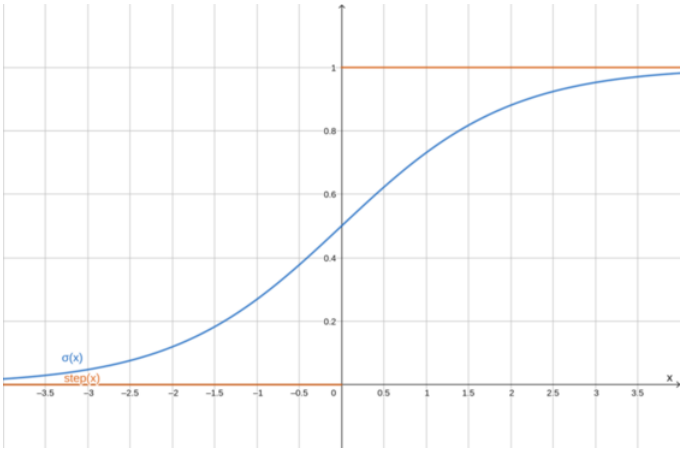


Figure 2.5: Sigmoid σ (blue) and step (orange) activation functions. The former is a smoothed version of the latter.

The single sigmoid neuron is a linear model that performs a binary classification task called *logistic regression* from a statistical point of view. To perform the classification with a continuous output, a threshold is set (usually 0.5), and for values above the threshold, the predicted label is 1, otherwise 0. A neural network consisting of sigmoid neurons is called *MultiLayer Perceptron* (MLP), the first universal function approximator model.

2.3.3 The MLP learning process

So far, we have understood the basic operating principles of a neural network, but how can we train it? Well, as mentioned earlier, we need a labeled dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ and we also need to define a *cost function*[7] that we call $\mathcal{L}$. For completeness, we define the mapping function $f_{\boldsymbol{\Theta}}$ associated with the MLP network, where $\boldsymbol{\Theta}$ is the vector of trainable parameters. The cost function evaluates the amount of error between the predicted value $f_{\boldsymbol{\Theta}}(\mathbf{x}_i)$ and the actual value y_i. The goal of the learning process is to minimize

$$L = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}\big(f_{\boldsymbol{\Theta}}(\mathbf{x}_i),\, y_i\big) \tag{2.4}$$

that is the *empirical risk* computed for our training set. In deep learning jargon, empirical risk is usually called *loss*, while in machine learning theory it is an alternative name of the cost function. Whatever you call it, from now on L will denote an average error, while $\mathcal{L}$ will denote the error evaluated for a single input example. The explicit expression of $\mathcal{L}$ depends on the specific task we are performing and encodes our knowledge of the problem. The change in empirical risk as the parameters vary can be evaluated as follows.

$$\Delta L \approx \frac{\partial L}{\partial \Theta_1}\Delta\Theta_1 + \ldots + \frac{\partial L}{\partial \Theta_P}\Delta\Theta_P = \nabla L \cdot \Delta\boldsymbol{\Theta} \tag{2.5}$$

P is the total number of parameters in the network, $\Delta\Theta_p$ is the variation of the p-th parameter, and ∇L is the gradient of empirical risk with respect to parameters $\boldsymbol{\Theta}$. The training process updates the parameter values to reach the minima of the empirical risk. The optimization algorithm is called *Gradien Descent*: the idea behind this algorithm is that at each training step, we can obtain information about the local structure of the hypersurface on which we are moving by computing the gradient of the empirical risk. Since we know that the gradient indicates the direction of maximum increase, we can take small steps in the exact opposite direction to ensure that we are moving in the direction of maximum decrease. Formally we have that

$$\Delta\boldsymbol{\Theta} = -\eta\nabla L \tag{2.6}$$

$$\boldsymbol{\Theta} \rightarrow \boldsymbol{\Theta}' = \boldsymbol{\Theta} - \eta\nabla L \tag{2.7}$$

where η is called the *learning rate* and measure the step size and the relation 2.7 is the update rule. More precisely, the parameters $\boldsymbol{\Theta}$ are weights and biases for each neuron in each layer, so we can call $w_{i,j}^{l}$ the j-th weight of the i-th neuron in the l-th layer and b_i^l the bias of the i-th neuron in the l-th layer. The update rule for weights and biases can be expressed as follows.

$$w_{i,j}^{l} \rightarrow w_{i,j}^{\prime l} = w_{i,j}^{l} - \eta\frac{\partial L}{\partial w_{i,j}^{l}} \tag{2.8}$$

$$b_i^l \rightarrow b_i^{\prime l} = b_i^l - \eta\frac{\partial L}{\partial b_i^l} \tag{2.9}$$

[7] In machine learning theory is also called *loss* or *objective* function.

In practice, at each training step, the empirical risk is calculated only for a sample of the input data, called a *mini-batch*. Given a random sample $B = \{(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_m, Y_m)\}$ of size m of training data we have the following.

$$\frac{1}{m} \sum_{(X,Y)\in B} \mathcal{L}\big(f_{\boldsymbol{\Theta}}(\mathbf{X}), Y\big) \approx \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\big(f_{\boldsymbol{\Theta}}(\mathbf{x}_i), y_i\big) = L \tag{2.10}$$

This approach is useful to speed up training, although the gradient evaluated is an approximated version of the actual maximum increase direction. How good the approximation is depends on the size of the mini-batch. The larger the mini-batch, the better the approximation, but the slower the training will be. The update rule is

$$w_{i,j}^{l} \to w_{i,j}^{\prime l} = w_{i,j}^{l} - \frac{\eta}{m} \sum_{(X,Y)\in B} \frac{\mathcal{L}\big(f_{\boldsymbol{\Theta}}(\mathbf{X}), Y\big)}{\partial w_{i,j}^{l}} \tag{2.11}$$

$$b_{i}^{l} \to b_{i}^{\prime l} = b_{i}^{l} - \frac{\eta}{m} \sum_{(X,Y)\in B} \frac{\mathcal{L}\big(f_{\boldsymbol{\Theta}}(\mathbf{X}), Y\big)}{\partial b_{i}^{l}} \tag{2.12}$$

This variant of gradient descent is called *Stochastic Gradient Descent*, abbreviated as SGD, and is widely used in practice. SGD is the basis for many other optimization algorithms that have been developed over time, one of the most famous is the Adam optimizer that can be found in Appendix C. The fast algorithm used to compute the derivatives is called *backpropagation* and was developed in the era of connectionism. The central idea of this algorithm is to exploit the graph structure of the network in combination with the chain rule of the derivative. As depicted in figure 2.6, each simple operation performed in the network can be considered as an operational node: from left to right we have the input signals used to perform the operation (multiplication, addition, activation function) and the generated output that is passed forward in the network to other operational nodes, from right to left we have as input the upstream gradient that is multiplied by the local gradient of the operational node to generate a downstream gradient that is passed backward in the network.

The local gradient of the operational node depends on the particular operation and the input signal values. Training the network means repeating several times the following two steps :

- **forward step**: a data mini-batch is fed into the network, which computes the output and evaluates the average error. The data (neuron activations) flow from left to right
- **backward step**: the partial derivatives, which are necessary for the update of the parameters, are calculated with the backpropagation algorithm. The data (gradients) flow from right to left.

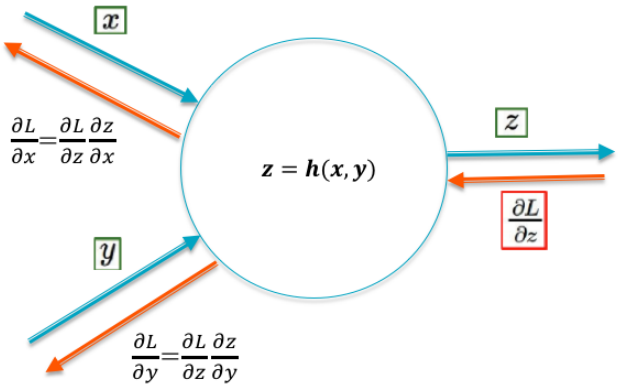


Figure 2.6: The graphic representation of an operational node where h is the operation, x and y are the inputs, z is the output, $\frac{\partial L}{\partial z}$ is the upstream gradient, $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ constitute the local gradients computed with respect to the inputs and $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$ constitute the downstream gradient. Blue arrows represent the data flow during the forward pass, red arrows the gradient flow during the backward pass.

2.4 Convolutional Networks
Towards deeper and deeper networks

Convolutional networks ([7] ch. 9, [20] ch. Deep Learning), also called *Convolutional Neural Networks* (CNNs), are a specialized case of neural nets that are used in dealing with grid-like data, such as time series (1D grid) or images (2D grid). The name convolution is improperly referred to the operation of computing the result of a sliding filter applied to the input data, which mathematically is called *cross-correlation*, and in discrete time can be written as

$$R_{IK}[m,\, n] = (I \star K)[m,\, n] \stackrel{\text{def}}{=} \sum_{j=-\infty}^{+\infty} \sum_{i=-\infty}^{+\infty} I[i,\, j] \cdot K[m+i,\, n+j] \tag{2.13}$$

where I is a 2D image and K is a 2D filter called *kernel*. A network that uses such an operation is called a convolutional network. This type of network is designed to exploit the spatial correlation typical of images by leveraging on the ideas of *sparse interactions*, *parameter sharing*, and *equivariant representation*.

These ideas come from neuroscience experiments conducted in the 1960s to understand how the mammalian visual system works. A simplified view of the brain visual function is based primarily on the *primary visual cortex* V1, an area of the brain located at the back of the head that begins performing advanced processing of visual input. Some important features of this area have inspired the ideas that convolutional networks exploit:

- V1 is arranged in a spatial map that mimics the structure of the image
- V1 consists of many simple cells whose response can be approximately described by a linear function applied to a spatially bounded receptive field

- V1 also constists of complex cells whose response is invariant to spatial shifts in the position of a feature

The simplistic view assumes that these basic blocks of feature recognition and pooling are stacked on top of each other to form a deep structure, that is the approach used by Kunihiko Fukushima to design the *Neocognitron* [5], a precursor of the CNNs. However, convolutional networks today cannot yet replicate the level of generalization that brain cells can achieve, despite the similarities cited above, there are many mechanisms that neuroscience cannot yet explain, and even influences from entirely different fields (optimization or statistics), that led to the creation of such convolutional networks. One of the most interesting features that research is trying to replicate is the attention mechanism that is inherently embedded in our visual system: while neural networks are fed with images representing all objects at the same resolution, only a small region on the retina in our eyes, called *fovea*, is capable of collecting high-resolution data, being responsible for the so called central sharp vision. The fovea is employed for accurate vision in the direction where it is pointed and sees only the central two degrees of the visual field. We will now look at the three most important components in a convolutional network, namely the *convolutional layer*, the *activation function*, the *pooling layer*, the *fully connected layer* and the *loss function*.

2.4.1 Convolutional layer

Standard Convolution

We can think of an input image as a grid of input neurons and the first layer of neurons as a set of filters. Each filter focuses on a different square, small portion of the input image called *receptive field*, unlike MLPs where each neuron considers all the available inputs. The receptive fields of all filters have the same size and are overlapped and regularly spaced. The spacing value is referred to as *stride*. This is what we mean by "sparse interactions", which indirectly allows the deeper layers to interact with a large portion of the input, according to the hierarchical modeling of concept complexity mentioned earlier. Another condition is that all these filters have the same weights, which is what we meant by "patameters sharing". Combined with the idea of sparse interactions, this allows for a massive reduction in the number of weights and can be implemented as a sliding filter, as depicted in figure 2.7, namely the cross-correlation defined in equation 2.13.

A sliding filter that essentially searches for the same feature across the entire image in different positions makes perfect sense in an image processing context, and means the network will be equivariant to translations of target subjects in the input images. This is because what we have called "concepts", or "features", are essentially parts of the image that the filters recognize when they fire, from the first layer that look for corners and edges to the deeper layers that combine the previous activations to recognize more complex parts of the image, e.g. parts of faces, if we are trying to accomplish a face recognition task.

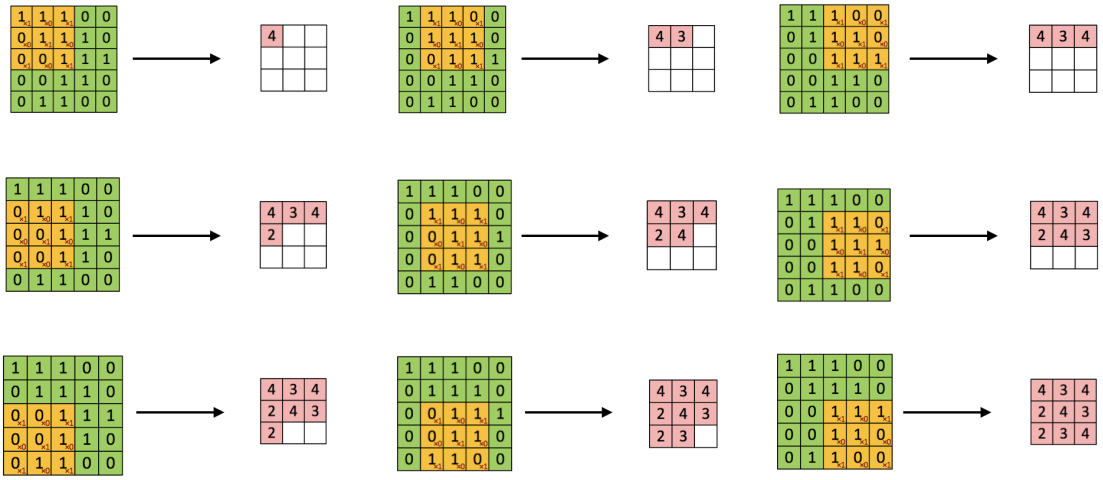


Figure 2.7: Representation of the 2D convolution between a 2D input and a 2D kernel. The green matrix is the input, the yellow region is the receptive field of the kernel and the red values are the kernel values. The stride in this simple case is 1. The result of the operation applied at each position corresponds to the dot product between the flatten kernel and the flatten interested input region.

As a rule, in each convolutional layer the inputs and outputs are actually 3D data structures called *tensors* and the third dimension is called *channels*. The main reason for this is that a generic convolutional layer consists of several 3D kernels, each of which has the same width and height, the *kernel size*, and a number of channels equal to the number of input channels. The filter can move along width and height dimensions, as in the 2D input case depicted in figure 2.7, and at each step the operation performed is the sum of all channel-wise dot products between the kernel and the image region to which the filter is applied. The output of each kernel applied to the entire input and modified by the activation function is a 2D *feature map* and all feature maps together form the output of the convolutional layer, whose number of channels is equal to the number of kernels.

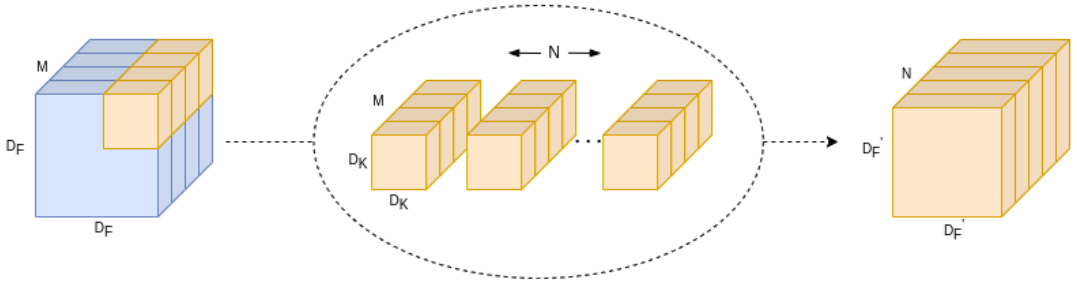


Figure 2.8: 2D convolution between a $D_F \times D_K \times M$ input and a convolutional layer with N $D_K \times D_K \times M$ kernels. The output is a $D'_F \times D'_F \times N$ tensor.

Depthwise Separable Convolution

This particular type of convolution is introduced in [13], with the aim of drastically reducing the computational cost and number of parameters of a model. Suppose we are in the situation showm in figure 2.8, with the standard convolution we have a computational cost (number of multiplications) of $D_K \cdot D_K \cdot M \cdot D'_F \cdot D'_F \cdot N$ that depends mainly on the size of the filters and the output feature maps. The depthwise separable convolution factorises the standard convolution into a *depthwise convolution* and a *pointwise convolution*.

With the depthwise convolution, a single 2D filter is applied to each input channel, which performs a type of channel filtering.

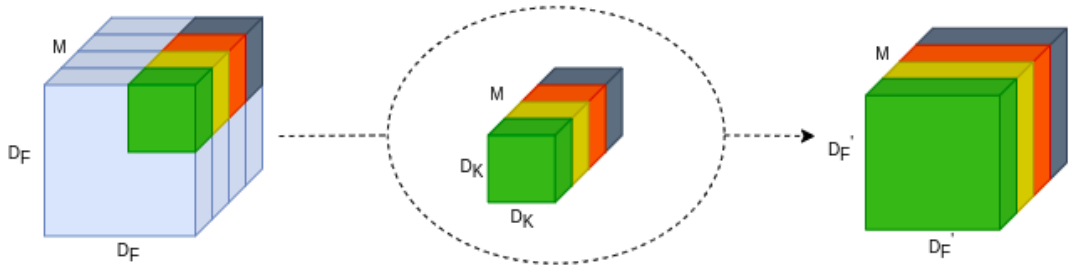


Figure 2.9: Depthwise convolution

The computational cost of the depthwise convolution is $D_K \cdot D_K \cdot D'_F \cdot D'_F \cdot M$.

Then the pointwise convolution applies 1×1 convolutions to combine the information along the channels to create new features.

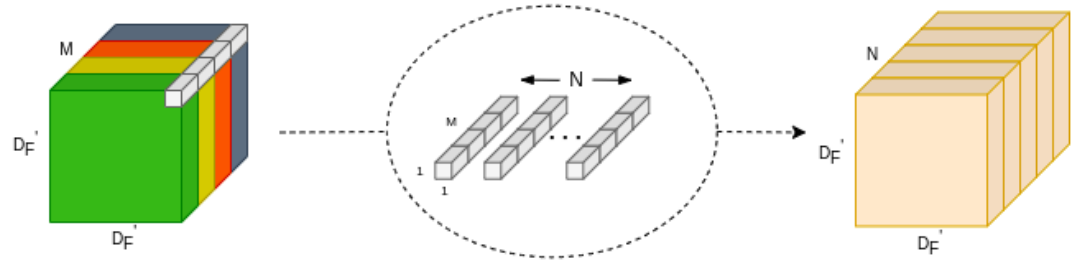


Figure 2.10: Pointwise convolution

The computational cost of the pointwise convolution is $M \cdot D'_F \cdot D'_F \cdot N$.

The gain we obtained can be measured as

$$\frac{D_K \cdot D_K \cdot D'_F \cdot D'_F \cdot M + M \cdot D'_F \cdot D'_F \cdot N}{D_K \cdot D_K \cdot M \cdot D'_F \cdot D'_F \cdot N} = \frac{1}{N} + \frac{1}{D_K^2}$$

2.4.2 Activation functions

To learn complex nonlinear mappings and better fit the data for a neural network, it is necessary to introduce nonlinear activation functions [11]. A good activation function must necessarily have the following properties:

- low computational cost
- it should be differentiable to allow the use of the backpropagation algorithm to update the parameters
- it should prevent the gradient from vanishing for input values far from zero

Sigmoid and tanh

Two common activation functions used with FNNs are simgoid σ and *tanh*. They have a similar form, but the sigmoid returns a value in the range $(0, +1)$, which can be interpreted as a probability distribution over a binary variable, while the latter returns a value between $(-1, +1)$. The two activations are not suitable for training deep newtorks, since the derivative computed for values far from zero is small and can cause the *vanishing gradient* problem: during backpropagation, upstream gradients whose modulus is smaller than 1 lead to the computation of smaller and smaller gradient values, which stop the learning process.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.14}$$

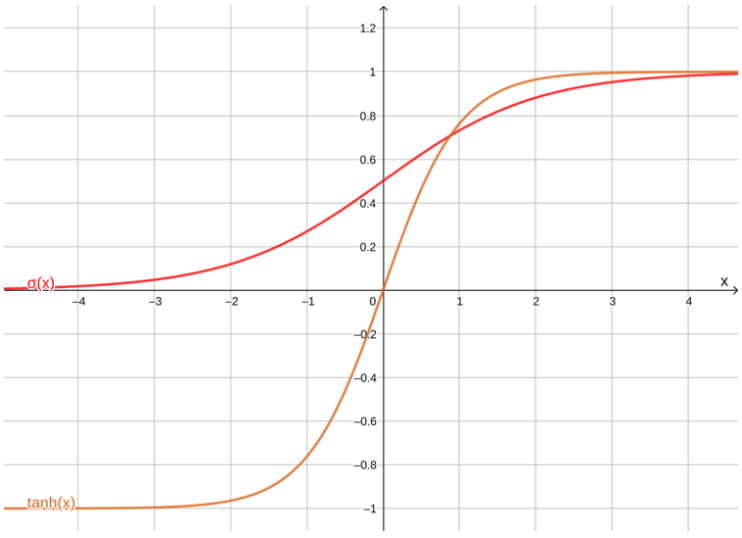


Figure 2.11: Sigmoid (red) and tanh (orange) activation functions.

ReLU and Leaky ReLU

With the CNNs a common activation function is the *Rectified Linear Unit* (ReLU), which fires only when the input value is greater than zero. The drawback of this function is that it cannot map a negative value, which limits the network's ability to learn. A possible solution is represented by the LeakyReLU activation function, which can also map negative values, weighted by a user-specified parameter α.

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \qquad \text{LeakyReLU}(x;\, \alpha) = \begin{cases} \alpha x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{2.15}$$

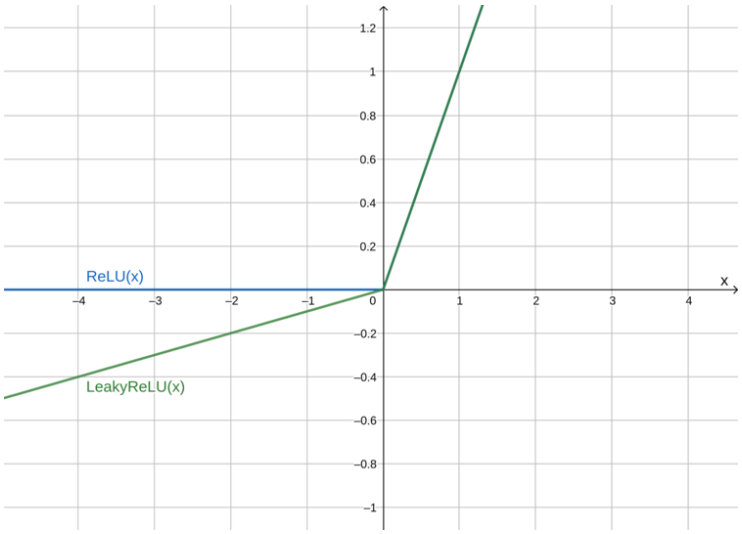


Figure 2.12: ReLU (blue) and LeakyReLU (green, $\alpha = 0.1$) activation functions.

Softmax

Another common activation function used both in FNNs and CNNs is the Softmax activation. This activation is used in the output layer of a neural network for multi-class classification problems and can be considered as a generalization of the sigmoid function. It is used to express a probability distribution over a discrete variable with k possible values and its expression is the following:

$$\text{softmax} : \mathbb{R}^k \to \{\mathbf{x} \in \mathbb{R}^k | x_i > 0,\, \sum_{i=1}^{k} x_i = 1\} \tag{2.16}$$

$$\text{softmax}(\mathbf{x})_j = \frac{e^{x_j}}{\sum_{i=1}^{k} e^{x_i}}. \tag{2.17}$$

2.4.3 Pooling layer

Usually convolutional layers are followed by a *pooling layer* which further modify the output of the activation function. The main purpose of the pooling layer is to generate a concentrated version of a feature map by applying a function which summarize a $d \times d$ region with some statistic. A common procedure for pooling is known as *max-pooling*, which takes the maximum activation value of the region, but there are also other versions such as the *average-pooling*, which computes the average activation value of the region, and *L2-pooling*, which computes the squared root of the sum of the squared activation values.

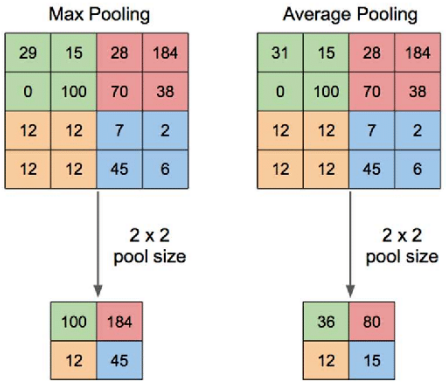


Figure 2.13: Max-pooling and average-pooling applied on the same feature map. The pool size is 2×2 and the stride is 2.

Conceptually, regardless the exact expression, performing the pooling operation means enforcing a representantion that is invariant to small translations. From a more practical point of view it helps to reduce the number of parameters needed in later layers.

The components described are combined to form the *automatic feature extraction block* of a CNN architecture, followed by the *classification block*. This block usually consists of one or more *fully connected layers* and, in the context of classification, and the *Softmax* activation function.

2.4.4 Fully connected layer

The fully connected layer, also called densely connected layer, is a type of layer used both in FNNs and CNNs. In the context of a FNN, it is the main layer that defines the entire network that consists of different sized fully connected layers stacked on top of each other, while in the context of a CNN it is usually used to define the classification block of the network. The characteristic of this layer is that each neuron is connected to all the outputs of the previous layer.

2.4.5 Cross-Entropy loss

One of the most common cost functions used for classification problems is the so called Cross-Entropy loss. When evaluated on a single training instance this cost function can be expressed as

$$CE = -\sum_{c=1}^{C} y_c \log(a_c), \text{ with } a_c = f(s_c) \tag{2.18}$$

where C is the number of classes, y_c is the c-th component of the ground truth vector $\mathbf{y}$, s_c is the c-th component of the score vector $\mathbf{s}$ generated by the network before the activation function, f is the activation function and a_c its ouput when applied to s_c, with $c \in \{1, \ldots, C\}$. The groundtruth label of each training instance is one-hot encoded into a groundtruth vector, i.e. $c \rightarrow \mathbf{y} = [y_1, \ldots, y_C]$ such that only the y_c component is 1 and all the others are 0.

e.g. with $C = 3$ we have the following labeling.

$$\begin{aligned}
&\text{gt label } 0 \rightarrow \mathbf{y} = [1,\, 0,\, 0]^\top \\
&\text{gt label } 1 \rightarrow \mathbf{y} = [0,\, 1,\, 0]^\top \\
&\text{gt label } 2 \rightarrow \mathbf{y} = [0,\, 0,\, 1]^\top
\end{aligned}$$

We can now analyze more in detail this cost function under different settings.

Multi-label classification

In the case of multi-label classification the predicted vector $\mathbf{a}$ is computed with sigmoid activation functions, so we have that $0 < a_c < 1$ for each c. The models predict for a single data instance all the classes c that satisfy $a_c > threshold$, where the threshold value is usually 0.5.

Looking at the expression 2.18, we can now better understand its behaviour: CE is positive and only evaluates the error for the correct class, the lower the score for the correct class, the higher the error, and complementarily, the higher the score, the lower the error.

Multi-class classification

In the multi-class classification case the predicted vector $\mathbf{a}$ is computed with the softmax activation, thus can be interpreted as a probability distribution, i.e.

$$a_c = p(y = c|\mathbf{x}) \text{ and } \hat{y} = \underset{c=1,\ldots,C}{\operatorname{argmax}}\, p(y = c|\mathbf{x}) \tag{2.19}$$

where $\hat{y}$ is the unique predicted label and y it the groundtruth label. In this context the cost function is called *Categorical Cross-Entropy loss*.

Binary classification

In the binary classification case the score is a scalar value s generated by a single sigmoid neuron that outputs the probability that the input instance belongs to the positive class. In this situation the cost function is called *Binary Cross-Entropy loss* and can be expressed as follow.

$$BCE = -y \log\big(p(y=1|\mathbf{x})\big) - (1-y)\log\big(1-p(y=1|\mathbf{x})\big) \tag{2.20}$$

The predicted label $\hat{y}$ is 1 if $p(y=1|\mathbf{x}) > threshold$ where the threshold value is usually 0.5.

The broader picture

The idea of cross-entropy is taken from the information theory field and has a nice probabilistic interpretation. In information theory given a source that generates events represented by a discrete random variable X that follows a probability distribution described by $P(X=x) = p(x)$, the *Entropy* $H(p)$ in equation 2.21 measures the average number of symbols needed to transmit the events without any loss of information, and is the best transmission model we can use. This average number of symbols can be more generally intended as a measure of the level of uncertainty linked to P. Intuitively the uniform distribution where all the events have the same probability has the higher level of uncertainty, while in all the other cases the most probable events (lower information) can be encoded with a smaller number of symbols in order to reduce the cost of transmission.

$$H(p) = E_{X\sim p(x)}[I(X)] = \sum_x p(x)\log\left(\frac{1}{p(x)}\right) \tag{2.21}$$

Given the same set of events, When we use a transmission model optimized for a probability distribution Q, to encode events generated by a probability distribution P, the average amount of symbols needed to transmit the events is measure by the *Cross-Entropy* $H(p,q)$.

$$H(p,q) = \sum_x p(x)\log\left(\frac{1}{q(x)}\right) = -\sum_x p(x)\log\big(q(x)\big) \tag{2.22}$$

$H(p) \leq H(p,q)$, and the equality holds if $p(x) = q(x)$. In our context, for the multi-class and binary classification settings, the two distributions are represented by the one-hot encoded label vector, which has the role of $p(x)$, and the output class distribution generated by the network, which has the role of $q(x)$. Miniminizing the Cross-Entropy means the learning model is trained to output the $q(x)$ that better approximates the $p(x)$.

2.5 Performance assessment and model selection

In order to train our model, we need to define an evaluation metric that tells us how well the model is doing its job. We are interested in obtaining an optimal value for the evaluation metric when the model is tested with previously unseen data, which unfortunately is not available at training time. Looking at the performance measured from the training data may seem a good alternative at first glance, but it is misleading due to the *overfitting* phenomenon: especially for deep learning models, which may have a large number of parameters, if the training data set is not large enough, the model will also learn noisy patterns typical of the available data and perform poorly when tested with unseen data. To avoid this problem, the available data can be partitioned using different approaches:

- **Train-Test splits** - The available data is split into two datasets, the training data, which is usually 80% of the original data, and the test dataset, the remaining 20%. The CNN is trained on the training dataset for several epochs. An epoch ends when the entire training dataset has been fed into the network, which updates the parameters to minimize the overall loss. At the end of training, the performance of the model is evaluated against the test data. Optimal performance on the training data and poor performance on the test data means that the model has overfitted the training data. Finally, after the performance has been evaluated, the model is trained with the combination of training and test data and then deployed.

- **Train-Validation-Test splits** - The available data is now divided into three parts: the training dataset, the validation dataset and the test dataset. The proportion varies depending on the size of the dataset, but is usually 50%, 25%, 25% or 70%, 10%, 20%. Training is performed using the training dataset and the validation dataset, while the test dataset remains unobserved during training. At the end of each epoch, the model is validated against the validation data, with its behaviour observed. Finally, when we are satisfied with the performance of the validation, the model is trained using the combination of training and validation data and the final evaluation is performed using the test dataset.

- **K-folds cross-validation** - If the available dataset has a reduced size, we are interested in retaining most of the data for training. The dataset is divided into k equal parts, the model is trained on $k-1$ and evaluated on the remaining part. The procedure is repeated until all parts are used as the validation dataset and finally the performances measured for each of the k parts are averaged. The model is finally trained on all available data.

The results of the validation dataset can also be used for model selection, i.e. for selecting the final model from different model types based on the results measured by the evaluation metric. The evaluation metric is selected according to the specific task. For classification, there are a number of different metrics that can be used. For a binary classification task, such as the one performed in this thesis, a number of different metrics can be easily derived from the so-called *confusion matrix*.

		predicted class	
		P	N
actual class	P	TP	FN
	N	FP	TN

Table 2.1: Confusion matrix for a binary classification problem.

In table 2.5 the two classes are called P (positive) and N (negative). The components of the matrix are:

- **True Positives** (TP): number of elements correctly classified as Positive
- **False negatives** (FN): number of elements wrongly classified as Negative
- **False Positives** (FP): number of elements wrongly classified as Positive
- **True Negatives** (TN): number of elements correctly classified as Negative

From these four measures we can derive the following metrics and curves:

Accuracy

Measure of the rate of right predictions the model is able to do. The denominator is equal to the total number of elements of the dataset.

$$ACC = \frac{TP + TN}{TP + FN + FP + TN} \tag{2.23}$$

Recall or True Positive Rate

Measure of the hit rate, i.e. how many positive labelled elements the model can recognize over the total positive class.

$$TPR = \frac{TP}{TP + FN} \tag{2.24}$$

Specificity or True Negative Rate

Measures how many negative labelled elements the model can recognize over the total negative class.

$$TNR = \frac{TN}{TN + FP} \tag{2.25}$$

Fall-Out or False Positive Rate

Measures how many elements the model misclassify as positives among the whole negative class.

$$FPR = \frac{FP}{FP + TN} = 1 - TNR \tag{2.26}$$

Precision or Positive Predictive Value

Measures how precise the model is, i.e. the proportion of true positive elements the model recognize among all the positive predictions it makes.

$$PPV = \frac{TP}{TP + FP} \tag{2.27}$$

False Discovery Rate

Measures the proportion of false positive among all the positive predictions the model makes, the lower the better.

$$FDR = \frac{FP}{FP + TP} = 1 - PPV \tag{2.28}$$

F1 score

Harmonic mean between Precision and Recall.

$$F1 = 2 \cdot \frac{PPV \cdot TPR}{PPV + TPR} \tag{2.29}$$

Receiver Operating Characteristic curve

As mentioned in the previous sections, the binary classifier makes its prediction based on a threshold. This means that a change in the threshold affects the confusion matrix described above. The *Receiver Operating Chatacteristic* curve, commonly referred to as the ROC curve, visually illustrates how the confusion matrix changes when the threshold is changed, from 1 (all examples are classified as negative) to 0 (all examples are classified as positive). In the following figure we can see what a typical ROC curve should look like.

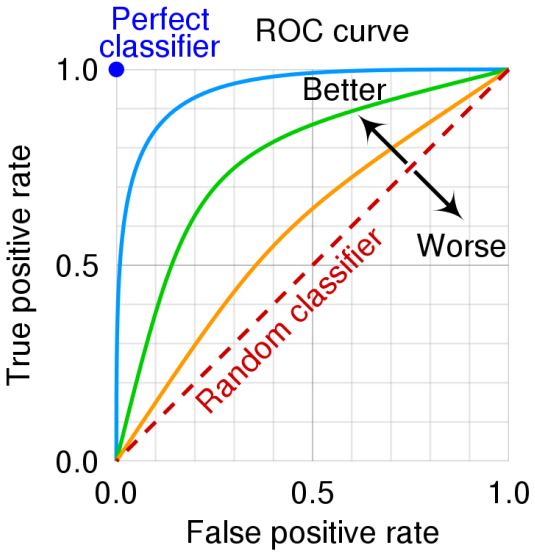


Figure 2.14: Typical ROC curve shapes.

For each threshold, each point on the curve has coordinates (FPR, TPR). The closer the curve is to the upper left corner, the better the classifier. The upper left corner represents the perfect classifier, which is able to correctly classify all positive elements ($TPR = 1$), without making any type-1 error ($FPR = 0$). A good classifier should always produce a ROC curve that lies above the red line. If this is not the case, it means that the classifier performs worse than random guessing. The ROC curve can also be used to compare the performance of different classifiers by comparing the AUC (Area Under the Curve) values: the higher the AUC value, the better the classification model.

Precision-Recall curve

It is recommended for unbalanced domains where the ROC curve might offer an overly optimistic view of performance. As in the case of the ROC curve, the output threshold is varied between 0 and 1 and both precision and recall are calculated for each value, then the corresponding point of coordinates (*recall*, *precision*) is plotted.

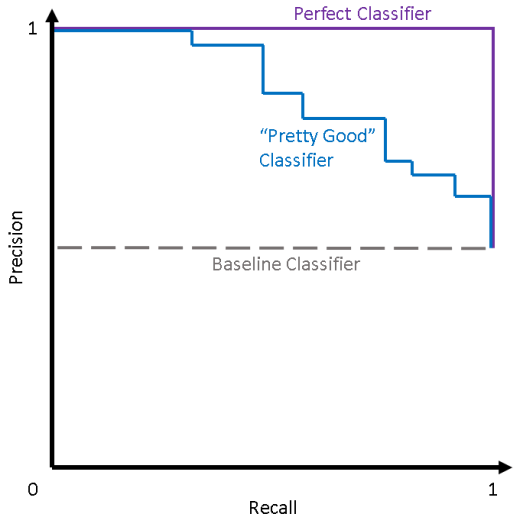


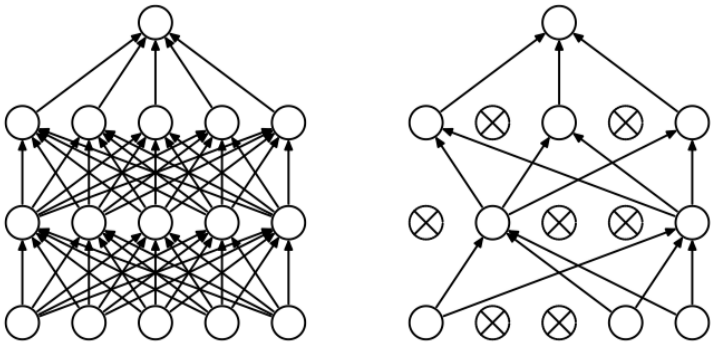
Figure 2.15: Precision-Recall curve shapes.

In figure 2.15, the perfect classifier is represented by the point (1, 1), i.e. it is able to detect all class elements without committing an error. The baseline of the classifier is calculated when the classifier always predicts the positive class, i.e. the recall value is 1 and the precision value depends on the number of positive class elements with respect to the total number of elements. Also in this case, the AUC value can be used to compare the performance of the different classifiers.

2.6 Regularization techniques

It is a set of techniques that aim to reduce the complexity of the model by setting some limits, or making some prior assumptions, to mitigate overfitting. The most common are:

- **Dropout layer** - A special layer that operates only in training mode, randomly removing connections between fully connected layers according to a predetermined probability value. The network learns more robust features.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.16: Neral network before and after dropout application.

- **Data augmentation** - A series of transformations (rotation, mirroring, cropping, brightness and contrast variations) applied to the available data to increase the size of the dataset. The choice of transformations depends on the task and can be applied online (during training) or offline, literally creating an enlarged version of the dataset before training.

- **Early Stopping** - Stopping the training when the monitored metric stops improving prevent overfitting.

- **Weight Decay (L2 regularization)** - Additional penalty term added to the term computed by the loss function, corresponding to the squared norm of the network weights multiplied by a factor λ that modulates the regularisation term.

$$\tilde{L} = L + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

This additional term has the effect of shrinking the weights during backpropagation preventing the network from overfitting the training data as well as the exploding gradient problem.

Chapter 3

Previous work

To improve the quality of the data collected, the meteor observing community has begun to use digital megapixel sensors that produce images with more than one million pixels. As image sizes grow, so does the processing load on detection systems. This increasing load and the need to respond in real time to capture fast events have resulted in the need to revise the detection pipeline to increase computational efficiency. With the proliferation of low-cost cameras, meteor detection networks have sought to automate as many steps in the pipeline as possible, with an emphasis on those steps that require human intervention.

The introduction of deep learning solutions into fireball detection pipeline is a very new field that has only been explored in recent years. In this chapter, a typical meteor processing pipeline is described and an overview of some proposed deep learning solutions is given.

3.1 Meteor detection pipeline

In [10]. Gural P. S. describes the main blocks in a typical meteor image processing system pipeline:

- **Capture** - Collection of video frames from a camera to a local storage device.
- **Compression** - Optional compression of raw data for compact transmission.
- **Detection** - This stage aims at indentifying the position of the meteor (if any) through a series of operation: clean-up preprocessing of imagery data, background estimation, fast thresholding algorithms for pixel exceedances, streak detection algorithms, centroid estimation.
- **Calibration** - Mapping of positional measurements from focal plane to celestial coordinates and photometric calibration to convert instrumental magnitudes to calibrated apperent magnitudes.
- **Confirmation/Classification** - Manual review or use of machine learning algorithms to filter out false positives.
- **Aggregation** - Combination of multi-camera measurements to associate several tracklets to the same observed object.
- **Trajectory** - Estimation of the 3-dimensional atmospheric track and motion dynamics of the object .
- **Orbit** - Estimation of the Keplerian orbital parameters.

3.2 Insertion of neural networks in the pipeline

In the described pipeline neural networks have the goal of minimizing human supervision, and so far they have been mainly used in the confirmation/classification step for false alarms mitigation.

In [6], Yuri Galindo and Ana Carolina Lorena use different approaches to perform transfer learning and show that choosing an appropriate pre-training dataset leads to better results. The dataset they work with was provided by the EXOSS Citizen Science Organization, a non-profit organization that studies and monitors meteors in Brazil. It consists of 1000 meteor images and 660 non-meteor images. The transfer learning approach is justified by the use of a small dataset and is applied to different deep ResNet architectures (namely 18, 34, 50 and 101 layers) pre-trained on ImageNet and Fashion-MNIST. To evaluate the network performances the authors use a 10-folds stratified cross validation approach. The best result of 4% error rate is obtained with a ResNet18 pre-trained on Fashion-MNIST, in combination with a final fine-tuning of the total network weights on the meteors dataset. The application of run-time data augmentation seems to worsen the results.

In [25] Martin C. Towner describes the detection system developed for the Desert Fireball Network (DFN) data processing pipeline. The data are collected with a Nikon D810 taking 25 s exposures every 30 s. Dataset provided by DFN consists of 200 tiles representing meteor streaks, coming from 50 manually selected 7360x4912

images, and 200 negative examples representing background noise or unindentifiable parts of meteor streaks. The dataset in then augmented by a factor of 8 by replication of the available data applying flip or rotation transformations and is split into train set (60%), cross validation set (20%) and test set (20%). The article compares a traditional approach and a neural network-based detection method. Both methods are executed after nightly data collection and must be able to process daily data onsite. The traditional approach is essentially a chain of increasingly computationally intensive image processing operations on fewer and fewer pixels. After the input image is tiled, each tile is compared to the same tile in the previous image and a thresholding operation is used to remove irrelevant tiles. Then, the Hough transform is used to generate line coordinates, which are subsequently filtered. Regarding the neural network approach, it begins with a preprocessing aimed at reducing the background noise of the images. In this step, as in the previous one, a thresholding method is applied to a pair of consecutive preprocessed and then tiled images. For a couple of consecutive images A and B, A is blurred and subtracted from B and vice versa. Binary dilation is applied to the new pair and the resulting images are compared. Each processed image is resized by a factor of 1/2 applying bilinear interpolation and divided into 25x25 adjacent tiles. Each tile is classified by a small FNN that has an input of size 625, 10 hidden neurons, and single output neuron in order to perform binary classification. The model uses a cross entropy loss function in training and the coordinates of the positively classified tiles (coordinates of the central pixel) are then treated as in the classical approach. The results of the neural network show a 14% error rate with a high number of false positive detections.

In [9], Peter S. Gural shows two different applications of neural networks to perform post-detection automated screening. The dataset, consisting of about 100,000 meteor observations and 100,000 non-meteor observations, is provided by the Cameras for All-sky Meteor Surveillance (CAMS) networks and includes two products generated by the processing pipeline. The first product is a dataset of compressed 640x480 image files containing images of 256-frame maximum temporal pixels called *maxpixel images* and other information for each pixel, such as the maximum value frame number, average value and standard deviation. The second product is a dataset of .txt files containing frame-by-frame track measurements consisting of frame number, coordinates of the centroid position of the meteor streak, and spatially integrated intensities around the streak. In order to correctly evaluate the model performances the 85% of the dataset is used to train the models, the 5% to validate them and the 10% to perform the final test. Peter Gural emphasises the need to achieve a high Recall, i.e., the percentage of correctly classified meteors, while keeping the number of false-positive predictions low. The collected time series measurements in the second source are used to train a Recurrent Neural Network (RNN) model that can achieve a Recall value of 98.1% and a False Detection Rate of only 2.1%. The image dataset is used to train a CNN that achieves the best results with 99.94% Recall and 0.4% False Discovery Rate. From the original images the centroid position information is used to get 64x64 patches containing meteor streaks. The network consists of three convolutional layers, each followed by a non-linear activation and a maxpooling layer, and a final fully connected layer with softmax activation. They observed that the network converges in only three epochs. To improve Recall, they decided to lower the decision threshold to 0.2 after

softmax activation, achieving 99.99% Recall and 0.9% False Discovery Rate.

In table 3.1 we can see a recap of the methodologies and best models developed in the works briefly described in this section.

Author	No. meteors	No. non meteors	Augmentation	Image size	Evaluation approach	Model type
Galindo	1,000	660	run-time	-	10-folds stratified cross validation	ResNet
Towner	200	200	offline x8 expansion	25x25	60% train 20% validation 20% test	small FNN
Gural	100,000	100,000	-	64x64	85% train 5% validation 10% test	small CNN

Table 3.1: Recap of methodologies and models used in the reported works

3.3 Takeaways and intuitions

From the preceding works, we can get an idea of what we are up against when performing this particular binary classification task. First of all, as a rule of thumb, if we want to train a model from scratch to get good results, we should have about 5000 examples for each class. If this is not the case, the transfer learning procedure used in [6] might mitigate the problem. The second clue we can draw is that a good preprocessing strategy to remove background noise and static lights is an important step before feeding the network. The classification task is not very complex, we are dealing with high-contrast greyscale images that do not contain many informative parts, so we can assume that we do not need to use a deep model. In [9] Gural shows that just using the spatial information in the maxpixel images to train a small CNN leads to really promising results. The use of a shallow model is also important to achieve real-time processing. The main challenge we have to face in order to successfully develop a real-time meteorite detector is that the direct use of the entire image makes the problem intractable and resizing the image is an option only if too much resolution is not lost, because the informative part usuful for the classification is in a small area of the whole input image. In the next chapter a possible solution to this problem is presented. To this end, we use two models found in the literature that are able to simultaneously learn to recognise important regions of the input image and correctly classify the image based only on these regions.

Chapter 4

Attention models

With the advent of high-resolution imagery, the meteor observing community must confront practical computational issues when using computer vision models to automate the detection step: analysing full-resolution imagery is impossible for a real-time application, and downsizing before processing can remove some important details degrading the performance. Computational complexity, and thus processing time, can be reduced by exploiting the fact that not all parts of the image are equally important in many image processing tasks. In addition, the regions of interest could be processed in more detail. The decision process that determines which regions are worth keeping and which should be discarded is non-trivial and highly task dependent. In general, the relevant regions must be identified first: a suitable formulation for the problem is *"given a regular grid of equally sized patches, decide which patches to process"*. This discrete formulation makes the problem unsuitable for end-to-end learning. Angelos Katharopoulos and François Fleuret in [16] and Jean-Baptiste Cordonnier and Aravindh Mahendrain (and others) in [4] provide different solutions to overcome this problem.

In the next section, we will describe the general structure of the model architecture. Then we will look at the theoretical background of the two models giving some specific implementation details.

4.1 The general architecture

Suppose we perform a classification task with L distinct classes, the network $\widetilde{\Psi}_\Theta(\cdot)$ in figure 4.1 can classify the high-resolution images by evaluating, selecting, and finally processing the most important regions, and is a good approximation of the $\Psi_\Theta(\cdot)$ network that processes all available regions. The model architecture can be divided into different functional blocks, which are shown in the following figure.

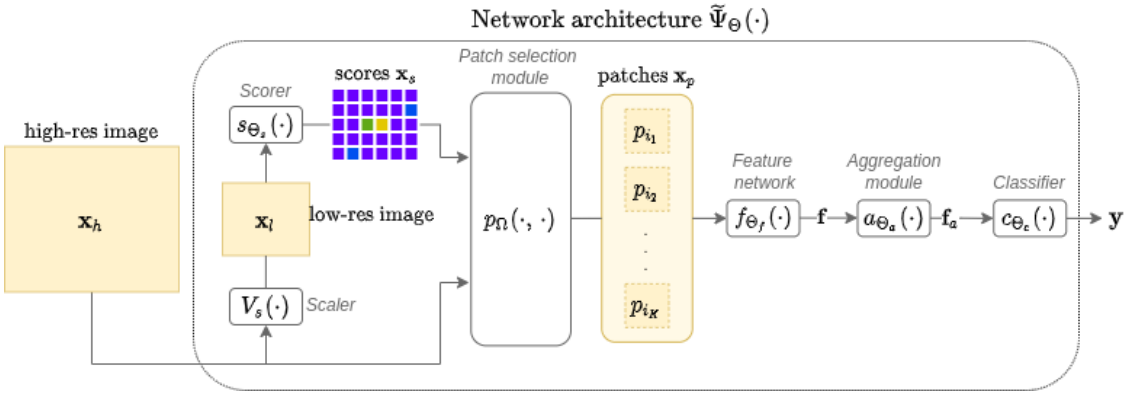


Figure 4.1: Functional blocks of the network.

where $\Theta = \Theta_s \cup \Theta_f \cup \Theta_a \cup \Theta_c$ are the trainable parameters of the network. The model takes in input the high-resolution image $\mathbf{x}_h \in \mathbb{R}^{H_h \times W_h \times C}$ and predicts the vector of probabilities

$$\mathbf{y}_{pr} = \widetilde{\Psi}_\Theta(\mathbf{x}_h) \in \mathbb{R}^L. \tag{4.1}$$

Low-resolution image

Given a high-resolution image $\mathbf{x}_h$, a corresponding low-resolution version

$$\mathbf{x}_l = V_s(\mathbf{x}_h) \in \mathbb{R}^{H_l \times W_l \times C} \tag{4.2}$$

is computed as a scaled view at scale s, where $0 < s < 1$, $H_l = \lfloor s \cdot H_h \rfloor$ and $W_l = \lfloor s \cdot W_h \rfloor$, with $\lfloor \cdot \rfloor$ being a round down approximation.

Scorer network

The scorer network generates a score grid $\mathbf{x}_s$ of dimensions $H_s \times W_s$, given in input the low-resolution $\mathbf{x}_l$, namely

$$\mathbf{x}_s = s_{\Theta_s}(\mathbf{x}_l) \in \mathbb{R}^{H_s \times W_s}. \tag{4.3}$$

We call $N = H_s \cdot W_s$ the total number of patches that can be extracted from the high-resolution input.

Patch selection module

The patch selection module $p_\Omega(\cdot, \cdot)$ performs the selection of K score values according to a certain principle, then maps the position indices of the selected values to the corresponding patches on the high-resolution image $\mathbf{x}_h$ and finally returns the K patches into a tensor $\mathbf{x}_p$ of dimensions $K \times H_p \times W_p \times C$, where $H_P \times W_p$ is the size of the patches

$$\mathbf{x}_p = p_\Omega(\mathbf{x}_s, \mathbf{x}_h) \in \mathbb{R}^{K \times H_p \times W_p \times C} \tag{4.4}$$

where the set of parameters Ω is not made by trainable parameters. The logic behind this module changes between the two proposed solutions and will be investigated in the next section.

Feature newtork

The feature newtork takes in input the extracted patches $\mathbf{x}_p$ and for each one computes a D-dimensional representation, collecting them into a matrix $\mathbf{f}$ of dimension $K \times D$.

$$\mathbf{f} = f_{\Theta_f}(\mathbf{x}_p) \in \mathbb{R}^{K \times D} \tag{4.5}$$

Aggregation module

The aggregation module takes in input the features in the $\mathbf{f}$ matrix and pools this information into a S-dimensional vector

$$\mathbf{f}_a = a_{\Theta_a}(\mathbf{f}) \in \mathbb{R}^S. \tag{4.6}$$

The aggregation modules can perform a simple operation such as taking the mean of the K features, in that case $S = D$, or can be a more complex block such as a transformer.

Classifier

The classifier $c_{\Theta_c}(\cdot)$ is the last part of the network, it takes in input the global information of the K extracted patches embedded into a S-dimensional vector and produces a vector of probabilities

$$\mathbf{y}_{pr} = c_{\Theta_c}(\mathbf{f}_a) \in \mathbb{R}^L. \tag{4.7}$$

4.2 Deep Attention-Sampling Model

The *Deep Attention-Sampling models* (ATS) is a category of models developed by Angelos Katharopoulos and François Fleuret in [16]. The authors' official implementation is available at [15]. We first give a general overview of the ATS approach and then trace it back to the problem described above.

4.2.1 General view of ATS

Suppose to have an input target-pair $\mathbf{x}, y$ from a dataset and a network $\Gamma_\Phi(\cdot)$ parametrized by Φ. Suppose now we can express the network as an intermediate representation $h_{\Phi_h}(\cdot)$ given in input to a classifier $g_{\Phi_g}(\cdot)$, namely

$$\Gamma_\Phi(\mathbf{x}) = g_{\Phi_g}(h_{\Phi_h}(\mathbf{x})) \tag{4.8}$$

where $h_{\Phi_h}(\mathbf{x}) \in \mathbb{R}^{N \times D}$ is a matrix of N D-dimensional features and $\Phi_h \cup \Phi_g \subseteq \Phi$, the set of all trainable parameters. Employing an attention mechanism at the intermediate representation $h_{\Phi_h}(\cdot)$ means defining a function $a_{\Phi_a}(\mathbf{x}) \in \mathbb{R}^N$ s.t. $\sum_{i=1}^N a_{\Phi_a}(\mathbf{x})_i = 1$ and $a_{\Phi_a}(\mathbf{x})_i \geq 0 \ \forall i \in \{1, \dots, N\}$, and then change the definition of the network $\Gamma_\Phi(\cdot)$ as follow

$$\Gamma_\Phi(\mathbf{x}) = g_{\Phi_g}\left(\sum_{i=1}^N a_{\Phi_a}(\mathbf{x})_i \cdot h_{\Phi_h}(\mathbf{x})_i \right) \tag{4.9}$$

where $\sum_{i=1}^N a_{\Phi_a}(\mathbf{x})_i \cdot h_{\Phi_h}(\mathbf{x})_i \in \mathbb{R}^D$ is a weighted sum of all the N D-dimensional features. By definition $a_{\Phi_a}(\cdot)$ is a multinomial distribution over N discrete elements and if we consider the population represented by the N features, we have that the weighted sum corresponds to the population mean

$$\sum_{i=1}^N a_{\Phi_a}(\mathbf{x})_i \cdot h_{\Phi_h}(\mathbf{x})_i = \mathbb{E}_{I \sim a(\mathbf{x})}[h_{\Phi_h}(\mathbf{x})_I]. \tag{4.10}$$

We can now rewrite the whole network as

$$\Gamma_\Phi(\mathbf{x}) = g_{\Phi_g}\left(\sum_{i=1}^N a_{\Phi_a}(\mathbf{x})_i \cdot h_{\Phi_h}(\mathbf{x})_i \right) = g_{\Phi_g}\Big(\mathbb{E}_{I \sim a(\mathbf{x})}[h_{\Phi_h}(\mathbf{x})_I] \Big) \tag{4.11}$$

where $\Phi = \Phi_a \cup \Phi_h \cup \Phi_g$ represents all the parameters of the modified network.

Sampling with replacement

As a consequence of equation 4.11, we can avoid computing all the N features approximanting the expectation with a Monte Carlo estimate, i.e. we sample a set $\mathcal{Q} = \{q_i \sim a_{\Phi_a}(\mathbf{x}) | i \in \{1, \dots, K\}\}$ of K i.i.d. indices according the attention distribution and use the unbiased point estimator of the features population mean

$$\mathbb{E}_{I \sim a(\mathbf{x})}[h_{\Phi_h}(\mathbf{x})_I] \approx \frac{1}{K} \sum_{i=1}^K h_{\Phi_h}(\mathbf{x})_{q_i} \tag{4.12}$$

to approximate the neural network with

$$\Gamma_\Phi(\mathbf{x}) \approx \widetilde{\Gamma}_\Phi(\mathbf{x}) = g_{\Phi_g}\left(\frac{1}{K}\sum_{i=1}^{K} h_{\Phi_h}(\mathbf{x})_{q_i}\right). \tag{4.13}$$

In Appendix A it is shown that this approximation is the best possible one, i.e. the one with minimum variance, when we have no information about the L_2 norm $\|h_{\Phi_h}(\mathbf{x})_i\|_2$, which means that we can impose the features to a uniform norm.

The attention function can be implemented as a shallow neural network, which is addressed as *attention network*. This means that we need to compute the gradient of loss with respect to the parameters of the attention network by sampling the set of indices $\mathcal{Q}$. To do this, we need to define the gradient of the unbiased estimator 4.12 with respect to the parameters of the network. In Appendix A it is shown that we can use a Monte Carlo estimator to approximate it as follows

$$\frac{\partial}{\partial\phi}\frac{1}{K}\sum_{i=1}^{K} h_{\Phi_h}(\mathbf{x})_{q_i} \approx \frac{1}{K}\sum_{i=i}^{K}\left[\frac{\frac{\partial}{\partial\phi}[a_{\Phi_a}(\mathbf{x})_{q_i}\cdot h_{\Phi_h}(\mathbf{x})_{q_i}]}{a_{\Phi_a}(\mathbf{x})_{q_i}}\right]. \tag{4.14}$$

Sampling without replacement

To avoid sampling the same elements more than once, the authors propose a formulation of the problem when sampling is done without replacement. The process of sampling is described as follows: the first index i_1 is sampled with probability $p_1(i) = a_{\Phi_a}(\mathbf{x})_i$, then the second index i_2 is sampled with probability $p_2(i|i_1) = \frac{a_{\Phi_a}(\mathbf{x})_i}{\sum_{j\neq i_1} a_{\Phi_a}(\mathbf{x})_j}$ and so on. Following this procedure, we take the K-th index, given $\mathcal{I} = \{i_1, \ldots, i_{K-1}\}$, with probability

$$p_K(i|i_1, \ldots, i_{K-1}) = \frac{a_{\Phi_a}(\mathbf{x})_i}{\sum_{j\notin\mathcal{I}} a_{\Phi_a}(\mathbf{x})_j}. \tag{4.15}$$

In the process described, each time an index is sampled it is no longer available and the probability distribution must be renormalised according to the remaining cumulative probability. This means we cannot simply average the features of the extracted patches to get an unbiased estimator of 4.10. The authors suggest the following estimator

$$\sum_{j=1}^{K-1} a_{\Phi_a}(\mathbf{x})_{I_j} h_{\Phi_h}(\mathbf{x})_{I_j} + h_{\Phi_h}(\mathbf{x})_{I_K} \sum_{t\notin\{I_1,\ldots I_{K-1}\}} a_{\Phi_a}(\mathbf{x})_t \tag{4.16}$$

and demonstrate that it is unbiased (see Appendix A), i.e.

$$\mathop{\mathbb{E}}_{I_1,\ldots,I_K}\left[\sum_{j=1}^{K-1} a_{\Phi_a}(\mathbf{x})_{I_j} h_{\Phi_h}(\mathbf{x})_{I_j} + h_{\Phi_h}(\mathbf{x})_{I_K} \sum_{t\notin\{I_1,\ldots I_{K-1}\}} a_{\Phi_a}(\mathbf{x})_t\right] = \mathbb{E}_{I\sim a(\mathbf{x})}[h_{\Phi_h}(\mathbf{x})_I] \tag{4.17}$$

with $I_1, \ldots, I_K$ sampled according $p_1(i), \ldots, p_K(i|i_1, \ldots, i_{K-1})$. Looking at the official implementation, the estimator used in practice seems to be different. For this reason, the full theoretical analysis from the reference article is not reproduced here. More details about the implementation can be found in the next section.

4.2.2 ATS in practice

We can now interpret the network architecture described in section 4.1 from the ATS point of view.

The input

the general input $\mathbf{x}$ is represented by the single high-resolution image $\mathbf{x}_h$

The attention network

The attention netwotk is implemented by the scorer $s_{\Theta_s}(\cdot)$ and the attention distribution is computed on $\mathbf{x}_l$.

The patch selection module

The patch selection module is implemented as a combination of three operations:

- the first one is a sampling function $\text{sample}_K(\cdot)$ that samples K indices $\in \{1, \ldots, N\}$ according the flatten attention map $\mathbf{s} = \text{flatten}(\mathbf{x}_s) \in \mathbb{R}^N$ (or attention distribution) computed by the scorer, and returns the corresponding 2D coordinates on the bidimensional attention map for each element, as illustrated in the following image.

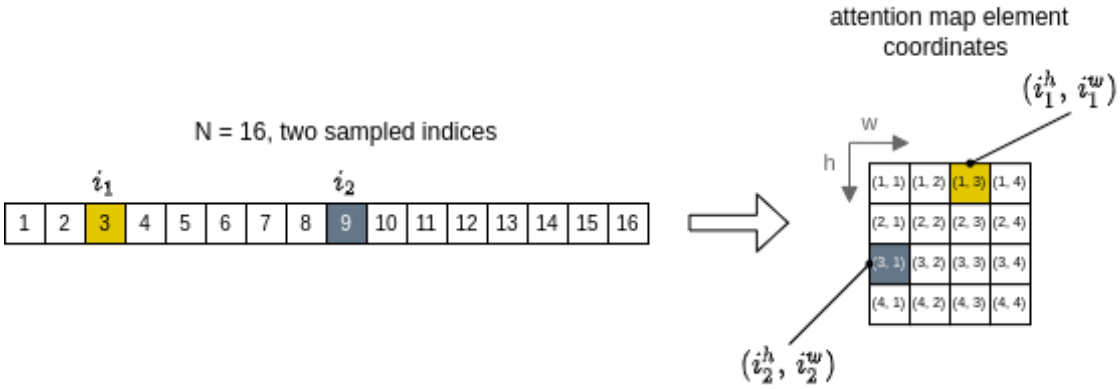


Figure 4.2: Sampling of 2 indices from an attention map of 16 elements, and conversion from index value to coordinates on the 2D attention map.

$$\text{sample}_K(\mathbf{s}) = \begin{bmatrix} i_1^h & i_1^w \\ i_2^h & i_2^w \\ \vdots & \vdots \\ i_K^h & i_K^w \end{bmatrix} = \mathbf{i} \in \mathbb{R}^{K \times 2}. \tag{4.18}$$

  In the case of sampling without replacement the sampling operation is performed with the Gumble-Max trick [14], explained in detail in Appendix A;

- the second operation is a mapping function $m_{\Omega_m}(\cdot) = \lfloor m'_{\Omega_m}(\cdot) \rceil$, that maps the sampled indices to the corresponding locations on the high-resolution input,

i.e. the top-left corner of the patches we want to retrieve, and then round the results to the nearest integer values:

$$m'_{\Omega_m}(\mathbf{j}) = \begin{bmatrix} j_1^h & j_1^w \\ j_2^h & j_2^w \\ \vdots & \vdots \\ j_M^h & j_M^w \end{bmatrix} \cdot \begin{bmatrix} \frac{H_l - r}{H_s} \cdot \frac{H_h}{H_l} \\ \frac{W_l - r}{W_s} \cdot \frac{W_h}{W_l} \end{bmatrix} \tag{4.19}$$

$$+\mathbf{1}_M \cdot \begin{bmatrix} \frac{r \cdot H_h}{2 \cdot H_l} + \frac{H_l - r}{2H_s} \cdot \frac{H_h}{H_l} - \frac{H_p}{2} & \frac{r \cdot W_h}{2 \cdot W_l} + \frac{W_l - r}{2W_s} \cdot \frac{W_h}{W_l} - \frac{W_p}{2} \end{bmatrix} \in \mathbb{R}^{M \times 2} \tag{4.20}$$

where $\mathbf{1}_M = [1, \ldots, 1]^\top \in \mathbb{R}^M$, $\Omega_m = \{H_h, W_h, H_l, W_l, H_s, W_s, r\}$, (H_h, W_h) are related to the high-resolution image size, (H_l, W_l) are related to the low-resolution image size, (H_s, W_s) are related to the output size of the scorer (the attention distribution), (H_p, W_p) are related to the patch size and r is a parameter that can be useful to trim the spacing between the elements of the extraction grid. More details about this mapping function can be found in Appendix A;

- the third operation is an extraction function $e_{\Omega_e}(\cdot, \cdot)$ that, given in input the high-resolution image $\mathbf{x}_h$ and the generic top-left corner coordinates $\mathbf{k} = [[k_1^h, k_1^w], [k_2^h, k_2^w], \cdots, [k_O^h, k_O^w]]^\top$ of O patches on the high-resolution image, extracts the corresponding patches returning a tensor of size $O \times H_p \times W_p \times C$

$$e_{\Omega_e}(\mathbf{x}_h, \mathbf{k}) \in \mathbb{R}^{O \times H_p \times W_p \times C} \tag{4.21}$$

where $\Omega_e = \{H_p, W_p\}$.

To sum up, for the ATS case we have

$$p_\Omega(\mathbf{x}_s, \mathbf{x}_h) = e_{\Omega_e}(\mathbf{x}_h, m_{\Omega_m}(\text{sample}_K(\mathbf{s}))) \tag{4.22}$$

with $\Omega = \{K\} \cup \Omega_m \cup \Omega_e$.

The aggregation module

The aggregation module is differentiable and its implementation changes depending on how the sampling is done, i.e. with or without replacement. In the ATS formulation, this module takes both the D-dimensional features of the extracted patches and the sampled score probabilities as input (although non explicit in the general architecture in Figure 4.1), because they are needed to compute the gradients in the backward pass: given the K patches and the corresponding attention distribution values derived from sampling the indices, we call $\mathbf{f}_i = f_{\Theta_f}(\mathbf{x}_p)_i \in \mathbb{R}^D$ the i-th row of the feature matrix $\mathbf{f}$ and $s_i \in [0, 1]$ the corresponding attention (score) value. When sampling is done without replacement, the authors' proposed implementation does not seem to follow exactly what they wrote in the reference article. The derivation of the gradients is also more difficult in relation to the case of sampling with replacement and there is no clear link between the theoretical justification and the proposed implementation, so only the implemented expressions in relation to the case of sampling without replacement are reported.

- **Forward pass** - For the forward pass the module implements the unbiased estimator 4.12, that in the sampling with replacement case is

$$a_{\Theta_a}(\mathbf{f}) = \frac{1}{K}\sum_{i=1}^{K}\mathbf{f}_i. \tag{4.23}$$

For the case of sampling without replacement the expression implemented by the module is

$$\begin{aligned} a_{\Theta_a}(\mathbf{f}) = \frac{1}{K}\big[&\mathbf{f}_1 && (4.24)\\ &+ (s_1\mathbf{f}_1 + \mathbf{f}_2 w_1) && (4.25)\\ &+ (s_1\mathbf{f}_1 + s_2\mathbf{f}_2 + \mathbf{f}_3 w_2) && (4.26)\\ &+ \dots && (4.27)\\ &+ (s_1\mathbf{f}_1 + s_2\mathbf{f}_2 + \dots + s_{K-1}\mathbf{f}_{K-1} + \mathbf{f}_K w_{K-1})\big] && (4.28)\\ = \frac{1}{K}&\left[\mathbf{f}_1 + \mathbf{1}_{K>1}\sum_{i=2}^{K}\left(\sum_{j=1}^{i-1} s_j\mathbf{f}_j + \mathbf{f}_i w_{i-1}\right)\right] && (4.29) \end{aligned}$$

where $\mathbf{1}_{K>1}$ is 1 if $K > 1$ else 0 and $w_r = 1 - \sum_{i=1}^{r} s_i$, with $r \in [1, 2, \dots, K-1]$. In both cases $\Theta_a = \varnothing$, i.e. there are no trainable parameters in the aggregation module.

- **Backward pass** - We define $\mathbf{g}_s$ and $\mathbf{g}_f$ as the downstream gradients computed for the scorer network (or attention network) and the feature network, and $\mathbf{u} \in \mathbb{R}^D$ as the upstream gradient, which has the same form as the output of the aggregation module.

For the case of sampling with replacement, the implemented gradients can be derived in a similar way as was the case in equation 4.14, but the general trainable parameter ϕ is replaced accordingly by the output of the two networks, giving the following expressions:

$$\mathbf{g}_s = \mathbf{u}^\top \frac{1}{K}\begin{bmatrix} \frac{\mathbf{f}_1}{s_1} \\ \vdots \\ \frac{\mathbf{f}_K}{s_K} \end{bmatrix}, \qquad \mathbf{g}_f = \frac{1}{K}\mathbf{1}_D\mathbf{u}^\top \tag{4.30}$$

where $\mathbf{1}_D = [1, \dots, 1]^\top \in \mathbb{R}^D$.

For the case of sampling without replacement the gradients are computed as follows:

$$\mathbf{g}_s = \mathbf{u}^\top \frac{1}{K} \begin{bmatrix} \frac{\mathbf{f}_1}{s_1} \\ \frac{w_1}{s_2}(s_1\mathbf{f}_1 + w_1\mathbf{f}_2) + \mathbf{f}_1 - \mathbf{f}_2 \\ \frac{w_2}{s_3}(s_1\mathbf{f}_1 + s_2\mathbf{f}_2 + w_2\mathbf{f}_3) + \mathbf{f}_1 + \mathbf{f}_2 - 2\mathbf{f}_3 \\ \vdots \\ \frac{s_K}{w_{K-1}}(\sum_{i=1}^{K-1} s_i\mathbf{f}_i + w_{K-1}\mathbf{f}_K) + \sum_{i=1}^{K-1}\mathbf{f}_i - (K-1)\mathbf{f}_K \end{bmatrix}, \quad (4.31)$$

$$\mathbf{g}_f = \frac{1}{K} \begin{bmatrix} 1 + (K-1)s_1 \\ w_1 + (K-2)s_2 \\ w_2 + (K-3)s_3 \\ \vdots \\ w_{K-2} + s_{K-1} \\ w_{K-1} \end{bmatrix} \mathbf{u}^\top. \quad (4.32)$$

Loss function

The training is driven by the following expression

$$\mathcal{L}'_\Theta(\mathbf{x}_h, y) = \mathcal{L}_\Theta(\mathbf{x}_h, y) - \lambda\mathcal{H}(s_{\theta_s}(\mathbf{x}_l)) \quad (4.33)$$

where $\mathcal{L}$ is a general loss function, in our case the binary cross entropy loss, $\mathcal{H}$ is the entropy of the attention distribution and λ is a hyperparameter that weights the effect of the regularization term. The goal of the regularization term is to promote exploration of the patch space.

4.3 Differentiable Patch Selection Model

The *Differentiable Patch Selection model* (DPS) is proposed by Jean-Baptiste Cordonnier and Aravindh Mahendrain (and others) in [4] and formalizes the patch selection as the Top-K problem that, given $\mathbf{x} \in \mathbb{R}^N$, can be defined as follows:

$$\text{Top-K}(\mathbf{x}) = \mathbf{y} \in \mathbb{R}^K \quad (4.34)$$

where $\mathbf{y}$ contains the indices of the K largest entries in $\mathbf{x}$. Using the *perturbed maximum method* [1], a differentiable Top-K is implemented and used in the patch selection module. For our purpose, the Top-K operation is defined so that the indices are ordered, i.e. $y_1 < y_2 < \ldots < y_K$. Without this constraint, the output could be permuted, breaking the perturbed optimizer method. For the next sections, it is convenient to represent the y_i's as the N dimensional indicator vectors $\{I_{y_1}, I_{y_2}, \ldots, I_{y_K}\}$, which are combined in a matrix $\mathbf{Y} = [I_{y_1}, I_{y_2}, \ldots, I_{y_K}] \in \mathbb{R}^{N\times K}$, so that if we take the tensor of all available patches $\mathbf{P} \in \mathbb{R}^{N\times H_p\times W_p\times C}$, the extracted patches can be written simply as $\mathbf{x}_p = \mathbf{Y}^\top\mathbf{P}$.

In the following sections we will first see how the perturbed maximum method works and then we will see how to use it to define a differentiable Top-K operation.

The official implementation is available at [8] and is implemented using FLAX[8] and JAX[9]. For the purposes of this work, a Tensorflow version is proposed.

[8]FLAX is a high-performance neural network library and ecosystem for JAX that is designed for flexibility, `https://github.com/google/flax`

[9]JAX is a recent machine/deep learning library developed by DeepMind. Unlike Tensorflow,

4.3.1 Differentiable perturbed optimizers

The approach described in this section is used to convert discrete optimizers into differentiable operations. The method perturbs the input of a discrete solver with random noise and considers the perturbed solutions of the problem: the formal expectations of the perturbed solutions are differentiable.

Definition 4.3.1 (Convex hull) *A set $\mathcal{X}$ in Euclidean space is defined convex if it contains the line segments connecting each pait of points. The convex hull of the set $\mathcal{X}$ is the (unique) minimal convex set containing $\mathcal{X}$.*

Given a finite set of distinct points $\mathcal{Y} \in \mathbb{R}^d$ and $\mathcal{C}$ its convex hull, consider the following general discrete optimization problem parametrized by an input $\boldsymbol{\theta} \in \mathbb{R}^d$ as follows:

$$F(\boldsymbol{\theta}) = \max_{\mathbf{y} \in \mathcal{C}} \langle \mathbf{y}, \boldsymbol{\theta} \rangle \tag{4.35}$$

$$y^\star(\boldsymbol{\theta}) = \arg\max_{\mathbf{y} \in \mathcal{C}} \langle \mathbf{y}, \boldsymbol{\theta} \rangle = \nabla_\theta F(\boldsymbol{\theta}) \tag{4.36}$$

where the problem is a linear problem (LP). $y^\star(\boldsymbol{\theta})$ in equation 4.36 is piecewise constant, i.e. its gradient is zero or undefined. To solve this problem, we can use the *stochastic smoothing*, which is defined in [12, chapter 8] as follows:

Definition 4.3.2 (Stochastic Smoothing) *Let $f : \mathbb{R}^N \to \mathbb{R}$ be a function. We define $\tilde{f}(\cdot; \mathcal{D}_\eta)$ to be the* stochastic smoothing *of f with distribution $\mathcal{D}$ and scaling factor $\eta > 0$. The function value at $\mathbf{G}$ is obtained as:*

$$\tilde{f}(\mathbf{G}; \mathcal{D}_\eta) := \mathbb{E}_{\mathbf{z}' \sim \mathcal{D}_\eta}[f(\mathbf{G} + \mathbf{z}')] = \mathbb{E}_{\mathbf{z} \sim \mathcal{D}}[f(\mathbf{G} + \eta \mathbf{z})] \tag{4.37}$$

where we adopt the convention that if z has a distribution $\mathcal{D}$ then the distribution of ηz is denoted by $\mathcal{D}_\eta$.

We essentially draw a random noise vector $\mathbf{Z}$ from a distribution $\mathcal{U}$ described by a density $\mu(\mathbf{z}) \propto \exp(-\nu(\mathbf{z}))$ on $\mathbb{R}^d$, and add $\epsilon \mathbf{Z}$ to $\boldsymbol{\theta}$, where ϵ is a temperature parameter. This induces a probability distribution $\mathcal{P}$ on $\mathbf{Y} \in \mathcal{Y}$ and lets us model the phenomena where agents have optimal $\mathbf{y} \in \mathcal{Y}$ based on an uncertain knowledge of $\boldsymbol{\theta}$. Using stochastic smoothing, we can define smooth versions of the equations 4.35 and 4.36, as follows:

Definition 4.3.3 *For all $\boldsymbol{\theta} \in \mathbb{R}^d$, and $\epsilon > 0$, we define the* ***perturbed maximum*** *as*

$$\widetilde{F}(\boldsymbol{\theta}; \mathcal{U}_\epsilon) = \mathbb{E}_{\mathbf{Z} \sim \mathcal{U}}[F(\boldsymbol{\theta} + \epsilon \mathbf{Z})] = \mathbb{E}_{\mathbf{Z} \sim \mathcal{U}}\Big[\max_{\mathbf{y} \in \mathcal{C}} \langle \mathbf{y}, \boldsymbol{\theta} + \epsilon \mathbf{Z} \rangle\Big] \tag{4.38}$$

and, the ***perturbed maximizer*** *as*

$$\tilde{y}^\star(\boldsymbol{\theta}; \mathcal{U}_\epsilon) = \mathbb{E}_{\mathbf{Z} \sim \mathcal{U}}[y^\star(\boldsymbol{\theta} + \boldsymbol{\epsilon} \mathbf{Z})] = \mathbb{E}_{\mathbf{Z} \sim \mathcal{U}}\Big[\arg\max_{\mathbf{y} \in \mathcal{C}} \langle \mathbf{y}, \boldsymbol{\theta} + \epsilon \mathbf{Z} \rangle\Big] \tag{4.39}$$

$$= \mathbb{E}_{\mathbf{Z} \sim \mathcal{U}}\Big[\nabla_\theta \max_{\mathbf{y} \in \mathcal{C}} \langle \mathbf{y}, \boldsymbol{\theta} + \epsilon \mathbf{Z} \rangle\Big] = \nabla_\theta \widetilde{F}(\boldsymbol{\theta}; \mathcal{U}_\epsilon). \tag{4.40}$$

JAX is not an official Google product and is used for research purposes, `https://github.com/google/jax`

A useful property of the stochastic smoothing is that under certain conditions it is always differentiable, as claimed in the following lemma, also reported from [12, chapter 8]:

Lemma 4.3.1 (Exponential Family Smoothing) *Suppose $\mathcal{D}$ is a distribution oven $\mathbb{R}^N$ with a probability density function of the form $\mu(\mathbf{X}) = \exp(-\nu(\mathbf{X}))/A$ for some normalization constant A. Let $f : \mathbb{R}^N \to \mathbb{R}$ be a function, $\tilde{f}(\cdot; \mathcal{D})$ its stochastic smoothing and $\mathbf{z}$ a random sample drawn from $\mathcal{D}$, then, for any twice differentiable ν we have:*

$$\nabla \tilde{f}(\mathbf{G}; \mathcal{D}) = \mathbb{E}_{\mathbf{Z}\sim\mathcal{D}}[f(\mathbf{G}+\mathbf{Z})\nabla_z \nu(\mathbf{Z})] \tag{4.41}$$

$$\nabla^2 \tilde{f}(\mathbf{G}; \mathcal{D}) = \mathbb{E}_{\mathbf{Z}\sim\mathcal{D}}[f(\mathbf{G}+\mathbf{Z})(\nabla_z \nu(\mathbf{Z})\nabla_z \nu(\mathbf{Z})^\top - \nabla_z^2 \nu(\mathbf{Z}))]. \tag{4.42}$$

Further more if f is convex, we have

$$\nabla^2 \tilde{f}(\mathbf{G}; \mathcal{D}) = \mathbb{E}_{\mathbf{Z}\sim\mathcal{D}}[\nabla f(\mathbf{G}+\mathbf{Z})\nabla_z \nu(\mathbf{Z})]. \tag{4.43}$$

The proof of the lemma can be seen in Appendix B. When the result of the previous lemma is adapted to our perturbed optimizer framework described at the beginning of this section, we have the following result.

Proposition 4.3.1 *For noise $\mathbf{Z}$ drawn from a distribution $\mathcal{U}$ whose PDF is $\mu(\mathbf{Z}) \propto \exp(-\nu(\mathbf{Z}))$ where ν is twice differentiable, the following holds:*

$$\widetilde{F}(\boldsymbol{\theta}; \mathcal{U}_\epsilon) = \mathbb{E}_{\mathbf{Z}\sim\mathcal{U}}[F(\boldsymbol{\theta}+\epsilon\mathbf{Z})] \tag{4.44}$$

$$\tilde{y}^\star(\boldsymbol{\theta}; \mathcal{U}_\epsilon) = \nabla_\theta \widetilde{F}(\boldsymbol{\theta}; \mathcal{U}_\epsilon) = \mathbb{E}_{\mathbf{Z}\sim\mathcal{U}}[F(\boldsymbol{\theta}+\epsilon\mathbf{Z})\nabla_z \nu(\mathbf{Z})/\epsilon] \tag{4.45}$$

$$J_{\boldsymbol{\theta}} \tilde{y}^\star(\boldsymbol{\theta}; \mathcal{U}_\epsilon) = \mathbb{E}_{\mathbf{Z}\sim\mathcal{U}}[F(\boldsymbol{\theta}+\epsilon\mathbf{Z})(\nabla_z \nu(\mathbf{Z})\nabla_z \nu(\mathbf{Z})^\top - \nabla_z^2 \nu(\mathbf{Z}))/\epsilon^2] \tag{4.46}$$

$$= \mathbb{E}_{\mathbf{Z}\sim\mathcal{U}}[y^\star(\boldsymbol{\theta}+\epsilon\mathbf{Z})\nabla_z \nu(\mathbf{Z})/\epsilon] \tag{4.47}$$

with $J_{\boldsymbol{\theta}} \tilde{y}^\star(\boldsymbol{\theta}; \mathcal{U}_\epsilon)$ the Jacobian matrix of $\tilde{y}^\star(\cdot; \mathcal{U}_\epsilon)$ at $\boldsymbol{\theta}$.

For completeness the demostration of the lemma is reported in Appendix B. In conclusion, the derivatives are simple expectations that can be evaluated with Monte Carlo estimates.

4.3.2 Differentiable Top-K

Patch selection as Top-K with sorted indices is equivalent to find a solution for the following linear program

$$\max_{\mathbf{Y}\in\mathcal{C}} \langle \mathbf{Y}, \mathbf{s}\mathbf{1}_K^\top \rangle \tag{4.48}$$

where $\mathbf{s} = \text{flatten}(\mathbf{x}_s) \in \mathbb{R}^N$, $\mathbf{s}\mathbf{1}_K^\top \in \mathbb{R}^{N\times K}$ are th score replicated K times and $\langle\rangle$ flattens the matrices before computing the dot product. The constraint set is defined as

$$\mathcal{C} = \{\mathbf{Y} \in \mathbb{R}^{N\times K} : \mathbf{Y}_{n,k} \geq 0,\ \mathbf{1}^\top\mathbf{Y} = \mathbf{1},\ \mathbf{Y}\mathbf{1} \leq \mathbf{1}, \tag{4.49}$$

$$\sum_{i=1}^{N} i\mathbf{Y}_{i,k} < \sum_{j=1}^{N} j\mathbf{Y}_{j,k'}\ \forall k < k'\} \tag{4.50}$$

where the first condition states that $\mathbf{Y}$ is filled with non-negative entries and each column has a total weight of 1, and the last condition states that the indices must be sorted. As long as the columns of $\mathbf{s}\mathbf{1}_K^\top$ are always identical, we must apply the noise directly to $\mathbf{s}$ before duplicating it in the K columns, and the index-sorted Top-K operator returning indicator vectors is a solution to the linear program described by 4.48, i.e.

$$\arg\max_{\mathbf{Y}\in\mathcal{C}} \langle \mathbf{Y},\, \mathbf{s}\mathbf{1}_K^\top \rangle \equiv \text{ISITop-K}(\mathbf{s}) \tag{4.51}$$

where ISI stands for Indicators of Sorted Indices, meaning it returns the indicator vectors of the sorted integer indices.

4.3.3 Differentiable Patch Selection Model in practice

We can now interpret the network architecture described in section 4.1 from the DPS point of view:

The scorer network

The output $\mathbf{x}_s$ of the scorer $s_{\Theta_s}(\cdot)$ cannot be interpreted as a probability distribution, but is simply normalized to obtain values in the range [0, 1] thanks to the following normalization function:

$$\text{normalize}(\mathbf{x}) = \frac{\mathbf{x} - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})} \tag{4.52}$$

The patch selection module

The patch selection module is implemented as a combination of two operations:

- the first operation is a submodule called DTop-K$(\cdot)$ that takes in input the flatten scores $\mathbf{s}$ and implements the differentiable Top-K as follows:

  - **Forward pass** - Assume that the noise $\mathbf{Z}$ is drawn from a standard multivariate normal distribution $\mathcal{N}$, and that the set of constraints $\mathcal{C}$ is the one described by the constraints 4.49 and 4.50, then the forward pass should perform the following operation

    $$\mathbf{Y}_\sigma = \mathbb{E}_{\mathbf{Z}\sim\mathcal{N}}\left[\arg\max_{\mathbf{Y}\in\mathcal{C}} \langle \mathbf{Y},\, (\mathbf{s} + \sigma\mathbf{Z})\mathbf{1}_K^\top \rangle\right] \tag{4.53}$$

    that in practice is approximated as follows

    $$\mathbf{Y}_\sigma \approx \frac{1}{M}\sum_{i=1}^{M} \text{ISITop-K}(\mathbf{s} + \sigma\mathbf{Z}^{(i)}). \tag{4.54}$$

    In order to meet the memory contraints, M is set to 250 in the experiments.

– **Backward pass** - The Jacobian associated with the above forward pass is given by the equation 4.47, and if the noise distribution is a standard multivariate normal distribution, it has the following form:

$$J_s \mathbf{Y}_\sigma = \mathbb{E}_{\mathbf{Z}\sim\mathcal{N}}\left[\arg\max_{\mathbf{Y}\in\mathcal{C}} \langle \mathbf{Y},\, (\mathbf{s}+\sigma\mathbf{Z})\mathbf{1}_K^\top\rangle \mathbf{Z}^\top/\sigma\right]. \tag{4.55}$$

In practice it is approximated as follows

$$J_s \mathbf{Y}_\sigma \approx \frac{1}{M}\sum_{i=1}^{M} \text{ISITop-K}(\mathbf{s}+\sigma\mathbf{Z}^{(i)})\mathbf{Z}^{(i)^\top}/\sigma; \tag{4.56}$$

- the second operation extracts all the patches from the input image, as in the ATS case, into a tensor $\mathbf{P} \in \mathbb{R}^{N\times H_p\times W_p\times C}$ and returns the results of the patch selection module, that can be finally expressed as

$$p_\Omega(\mathbf{x}_s,\, \mathbf{x}_h) = \mathbf{Y}_\sigma^\top \mathbf{P} \in \mathbb{R}^{K\times H_p\times W_p\times C}. \tag{4.57}$$

The parameters Ω and the extraction operation are exactly the same as in the ATS case, but now the entire set of initial coordinates of the patches is passed to the extraction function.

In the forward pass described by the equation 4.54, the result of the ISITop-K$(\cdot)$ is a matrix of one-hot indicator vectors, but the mean $\mathbf{Y}_\sigma$ may be far from one-hot. This means that in the early phase of training, the K patches extracted by the patch selection module resemble a weighted sum of all available patches. To avoid memory problems, this process is implemented cyclically and parallelized along the batch dimension, updating an intermediate result step by step until all patches have been extracted and the final result can be returned.

The DTop-K$(\cdot)$ is only used during training, in inference mode it is replaced by the hard Top-K operation. To bridge the gap between training and inference resulting from this replacement, both the σ parameter and the learning rate are reduced to 0 during training. In this way, the newtork ends with hard Top-K selection and we avoid exploding gradients that might result from the Jacobian expression 4.56 where σ is in the denominator, because when the learning rate is 0, gradients no longer flow back into the network.

The mapping function that maps the indices of the flatten-score elements to the corresponding patch positions on the high-resolution image is implemented differently from the one used in the ATS case (equation 4.20). In the authors' proposed implementation, the high-resolution image is zero padded along the height and width dimensions with $\left[\lfloor (H_p - H_h/H_s)/2 \rceil,\, \lfloor (W_p - W_h/W_s)/2 \rceil\right]$ zeros and then the following mapping is used, which is only given for a single index for clarity:

$$i \mapsto \left[\left\lfloor \frac{i}{W_s} \right\rfloor \frac{H_h}{W_s},\, (i \bmod W_a)\frac{W_h}{W_s}\right] \tag{4.58}$$

where $\left(\left\lfloor \frac{i}{W_s} \right\rfloor,\; i \bmod W_s\right)$ are the height and width coordinates on $\mathbf{x}_s$ of the same element indexed by i on the flat scores $\mathbf{s}$, and both H_h/H_s and W_h/W_s are the scale values related to the height and width dimensions computed from the

scores map $\mathbf{x}_s$ for the high-resolution image $\mathbf{x}_h$. In the implementation proposed for this work, we do not need to pad the high-resolution image with zeros because the extraction function is written to handle negative coordinate inputs and simply fills these locations with zeros when the patches are returned. This means that the actual mapping function used in the current implementation subtracts from the mapping 4.58 the offsets that result from padding the image with zeros, namely:

$$i \mapsto \left[\left\lfloor \frac{i}{W_s} \right\rfloor \frac{H_h}{W_s},\ (i \bmod W_a)\frac{W_h}{W_s}\right] - \left[\left\lfloor \left(H_p - \frac{H_h}{H_s}\right)\frac{1}{2} \right\rceil,\ \left\lfloor \left(W_p - \frac{W_h}{W_s}\right)\frac{1}{2} \right\rceil\right]. \quad (4.59)$$

In Appendix B is reported a toy example of the mapping.

Aggregation module

The aggregation module is implemented as in the ATS model, i.e. it simply calculates the mean of the features returned by the feature network. The operation is still differetiable but does not require a user-defined definition of the gradients, the standard operations are automatically differentiated by Tensorflow.

Loss function

The model is trained with a binary cross-entropy loss plus the same regularization term used for the ATS case. The loss expression is given by equation 4.33.

4.4 Extraction function

The extraction function is a custom Tensorflow operation written in C++ with both a CPU and GPU implementation based on the CUDA programming model. Tensorflow provides instructions for creating custom ops, which can be found at [24]. The custom op is able to extract a certain number of patches from all the images in a batch, specifying the starting points of each patch and taking advantage of the parallelization offered by a GPU. More specifically, in our case, a number of blocks #blocks = batch_size $\cdot$ #patches is used, so there are 1024 threads[10] that works in parallel to extract each patch [11]. The GPU kernel version of the op must be compiled with the NVIDIA compiler nvcc and it may be necessary to specify the -arch argument, i.e. the GPU architecture type code, for a successful compilation. Even though Tensorflow already provides some ops for performing slicing of tensors, such a custom op[12] is not yet officially implemented.

[10] Single execution unit running the kernel (function running on a GPU). Each thread is associated with a single core in the GPU.

[11] The hierarchy of threads is organized as follows: *threads* $\subset$ *blocks* $\subset$ *grid*, i.e. *threads* are grouped into *blocks* (maximum 1024 threads per block), which in turn are grouped into a *grid*. Each abstraction layer of the hierarchy can be organized into a 1D, 2D, or 3D data structure, and each thread can be uniquely addressed.

[12] The official implementation of ATS provides a version of this custom op but the original GPU kernel does not make good use of parallelization.

Chapter 5

Datasets

To understand the potential and weaknesses of the two proposed models, different datasets were defined. This chapter presents all the datasets that were defined from the exploratory phase to the final result and explains the objective of each dataset. In the first part of this work, we will use two syntethic datasets to get an idea of how the models behave in different situations. We will refer to these two types of datasets as *Needle MNIST* (NMNIST) and *Megapixel MNIST* (MPMNIST). As for the final dataset, it was created starting from a set of video recordings of events collected by the current detection system and then elaborated in different ways, which will be explained in detail later in this chapter.

5.1 Megapixel MNIST

Used by the authors of ATS, Megapixel MNIST is a synthetic dataset based on the popular MNIST dataset of 28×28 handwritten digits. It is used to simulate a situation suitable for the application of the ATS model, i.e. the classification of a noisy image, considering only the important regions. The images of the dataset are created starting from a blank canvas with predefined dimensions, which is then filled with digits from the MNIST dataset and specially created noisy patterns. Each image has a size $w \times h$ of 1500×1500 and contains five digits, three instances of the target class, two instances of digits from other randomly selected classes, and 50 noisy samples. The dataset contains a training split of 5000 items and a test split of 1000 items and is used to perform a classification task with 10 classes.

Figure 5.1: Data items taken from MPMNIST.

5.2 Needle MNIST

The NMNIST dataset is proposed in [21] and is a synthetic dataset created starting from the MNIST dataset. It is mainly used to simulate applications where the informative part of the image is much smaller than the image itself. The images of the dataset are created starting from a blank canvas with predefined dimensions, which is then filled with digits from the MNIST dataset. More specifically, each image has a size $w \times h$ of 1296×966 and contains at least one target digit with probability 0.5. The task is a binary classification, as the one we should perform on the final dataset, so the images with at least one target digit are labelled as 1. The target digits are selected among the images of class 3 in the MNIST dataset, and with a probability of 0.5, two of them can occur in the same image. In addition to the target digits, there are a varying number of *distortions* that are random samples from classes other than 3 from MNIST. Three versions of this dataset were created:

- **NMNIST_0** each image contains 0 distortions, only the target digits a present, if any
- **NMNIST_5** contains 5 distortions besides the taget digits
- **NMNIST_10** contains 10 distortions besides the target digits.

Each of these datasets contains a train split with 6000 elements and a test split with 1500 elements. The main purpose of these datasets is to see how the performance of the models deteriorates as more and more elements similar to the informative ones appear in the same image. Below is an example an item taken from NMNIST_5.

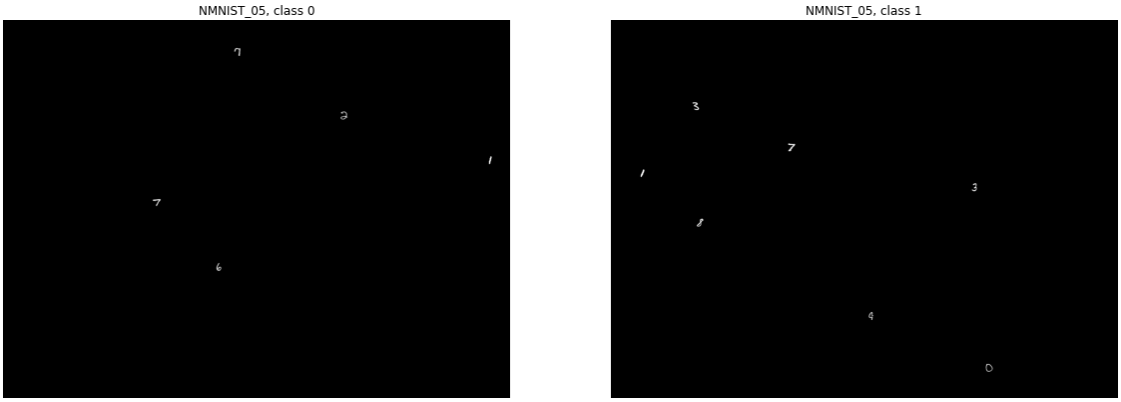


Figure 5.2: Data items taken from NMNIST_05, the class 1 image on the right contains two instancies of the target digit 3. The distortions are randomly placed without overlapping.

5.3 PRISMA dataset

5.3.1 Data exploration

The creation of the Fireballs dataset begins with a series of events collected by PRISMA/FRIPON. Each event is a folder, identified by the name of the viewing location and the date and time, and has the following format

$$SITE\_yymmddThhmmss\_UT.$$

Most folders contain both a subfolder called fits2D, which collects all the frames in FITS format, and a video.avi of the event; sometimes only one of the two is available. In addition, a .txt file containing the frame numbers and the coordinates of the detected object is also available.

Initially, the events are manually divided into fireballs and non-fireballs, i.e. the recorded event did not contain anything noteworthy or an aircraft was probably recorded. From the selection we could identify 38 fireballs events and 38 non-fireballs events.

We can now go through an exploratory phase to better understand the data we are dealing with. First of all, we have two different sources of information, the fits2D folder and the video.avi file. The fits2D folder contains raw data: the image is retrieved using the Python library **astropy**[13], which we can use to extract the image stored in uint16 format with resolution 1296×966. The video.avi is only used when fits2D is not available and the frames can be extracted by the Python library **opencv**[14], which lets us obtain the images stored in uint8 format with resolution 1280×960. The difference between the two data sources will be discussed in the next section. In the following figure we see the two sources in relation to the class label.

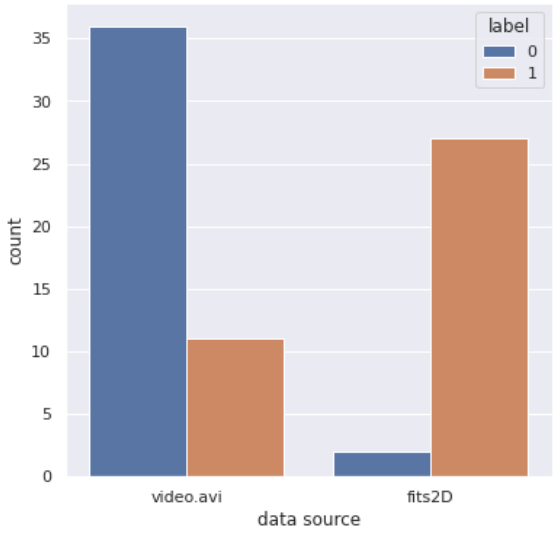


Figure 5.3: Count of events per class vs data sources. 1 corresponds to *fireball* and 0 to *non-fireball*.

[13] https://www.astropy.org/
[14] https://opencv.org/

In addition to the differences in the data sources, it is also worth examining the number of frames. In the following histograms we see the number of events, divided by class, for a given number of frames; this takes into account both the total number of frames collected and the number of frames for which the system believes that there is the detected object (this information can be taken from the .txt file).

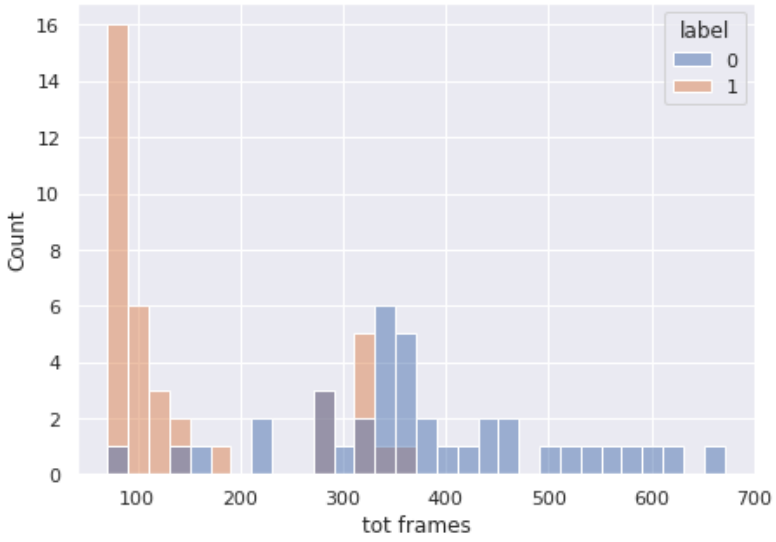


Figure 5.4: Count of events vs number of total frames.

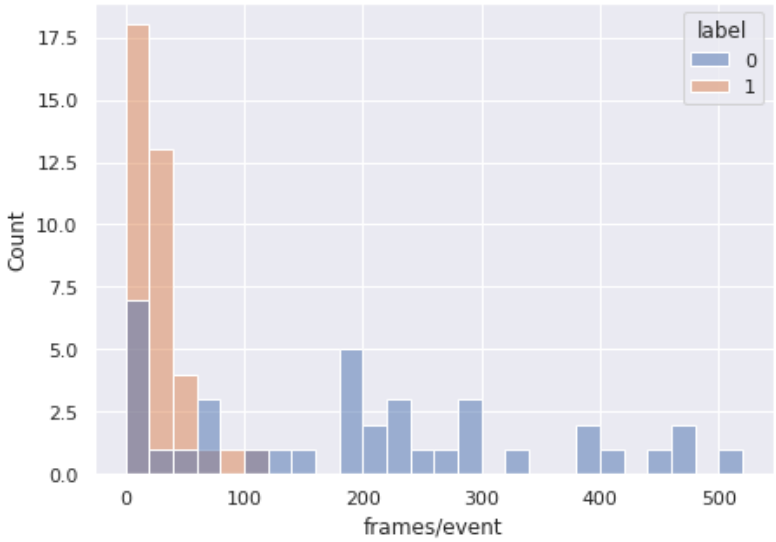


Figure 5.5: Count of events vs number of frames with detected object.

From the histograms above, it can be seen that the number of frames associated with a non-fireball event, both overall and object-wise, is generally higher than the number of frames of fireball events. The reason for this is probably that non-fireball events such as aircrafts tend to be longer than real fireball events, which can last a few seconds at most. This will result in the definition af an unbalanced dataset.

5.3.2 Dataset creation

Before processing the data to define useful datasets, 3 events were randomly selected for each class and set aside to determine the final test set. Then three different data sets are created from the data provided: *small56x56_augmented*, *small28x28_augmented* and *final_augmented*.

small56x56_augmented (small28x28_augmented)

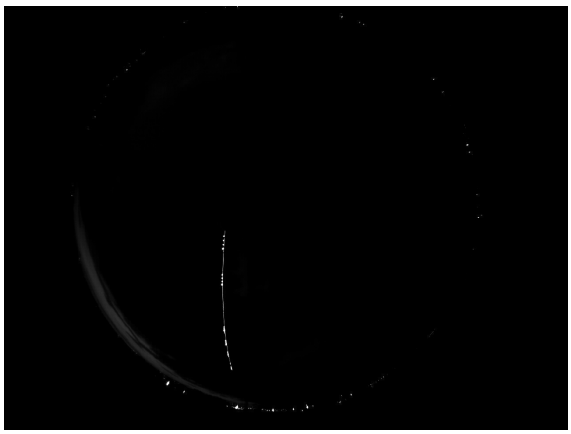


Figure 5.6: maxpixel image created for a non-fireball event.

This dataset is created with the aim of selecting a good feature network architecture. The feature network is then trained on this dataset and the weights are loaded into the corresponding layers in the overall model. The images in this dataset are created in the following way: all the frames that the currently working recognition system associates with an object recognition are merged, considering the maximum pixel value, so that the bright stripe of the object is visible, if present, the result is called *maxpixel image*. Before the maxpixel image is calculated, the frames are preprocessed by dividing each pixel by the maximum pixel value in the frame and multiplying by 255: this process is mainly necessary to make the visible objects from the two different sources comparable, because just by converting uin16 to unit8, the detections from fits2D are much less visible than the detections from video.avi.

Then, several crops of the stripe of size 56×56 are manually selected and stored. Finally, the dataset is enlarged by a factor of 8 by applying the following data augmentation: horizontal flip, vertical flip, both horizontal and vertical flip and random translation applied to the original and the flipped versions of each crop.

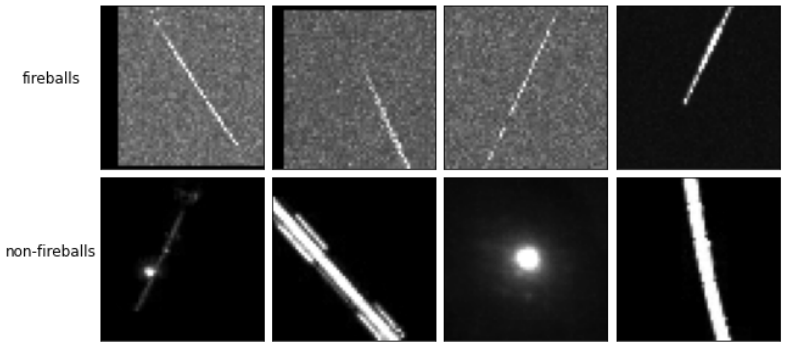


Figure 5.7: Sample of images from small56x56augmented.

The dataset small28x28augmented is created simply by resizing the images to 28x28, and is used in the feature network selection process in order to evaluate a

possible reduction in inference time and to see if the reduction in size has negligible effect on the performances.

final_augmented

This dataset is created to train and validate both the ATS and DPS models. Each frame is processed as in the previous case to make the events from the two data sources comparable. In addition, the frames from the video.avi source are resized to 1296×966, which is the final size of the images created. The creation pipeline is intentionally simple and simulates a pre-processing that can be used in the real application. The total number of frames of each event is processed as follows: a window of frames of about 2 seconds (65 frames) is used to calculate both the maxpixel image and the median image, which is subtracted from the first to delete or at least attenuate the fixed lights ant the background; the goal is to remove as many noisy elements as possible and the windows overlap by about 0.5 seconds (15 frames). The size of the window is sufficient to visualize a kind of strip (when visible) that can be extracted from the models into a patch, and the overlap of 15 frames is necessary to avoid too similar images in the dataset, also considering that they will be manually filtered at the end. On the right you can see an example of a frames window preprocessing. Of the available events, some are used to define a validation set to validate the performance of the models during training. Three sets are finally defined: a training set, a validation set and a test set. Each of these sets is manually filtered to retain the most informative images and then enlarged by a factor of 8, using the same data augmentation as in the previous case, except that the random translation is now a maximum of 100 pixels in each direction.

Figure 5.8: Example of a single 2 seconds window processing, from top to bottom: maxpixel image (MP), median image (M), maxpixel - median (MP-M)

To conclude this chapter, in the following table are reported the number of items for each class in the two datasets.

		fireballs	non-fireballs
small56x56augmented		448	928
final_augmented	train	696	4344
	validation	176	1088
	test	112	504

Table 5.1: Number of elements in the datasets.

The datasets are clearly unbalanced, moreover the majority of the elements in the positive class is created starting from quite noisy video recordings, that could represents a bias when training the final models. An example of a noisy positive class item is reported below.

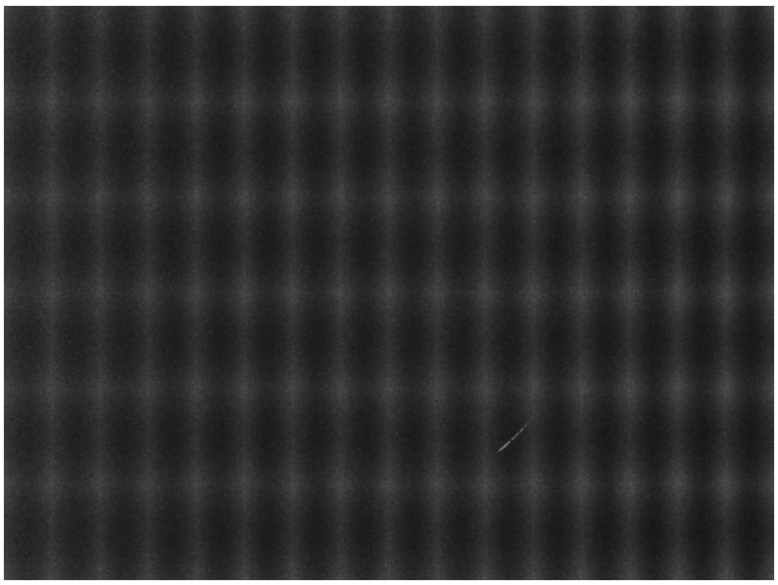


Figure 5.9: Example of a positive class item that shows the presence of a noisy pattern.

Chapter 6

Tests and experimental results

This chapter describes the methodology used to train and test ATS and DPS on the datasets presented in the previous chapter. Each training session is performed by means of Google Colab with the Adam optimizer [17] (more details in Appendix C) to update the network weights and, in the DPS case, a cosine decay scheduler with linear warmup is also used to allow the learning rate to decay slowly to zero. Before starting the training we need to set a number of model and training hyperparameters:

- the **scale** factor used to compute the low resolution image
- the **patch size**, that is a single value because we extract square patches
- the **number of patches** to be extracted
- the λ **regularizer strength** in the loss expression 4.33
- both the σ **noise strength** value and the **number of noise samples** M used in the differentiable TopK implementation given by 4.54 (only on DPS)
- concerning the optimizer, **learning rate**, **clipnorm** (only on ATS) and **clipvalue** (only on DPS)[14]
- concerning the scheduler, **warmup ratio** (only on DPS)
- concerning the training, **number of epochs** and **batch size**.

For each case are reported the specific scorer network and feature network architectures, the training approach, the evaluation metric and the final result.

[14] *clipnorm* for ATS and *clipvalue* for DPS are used as optimizer hyperparameters in their official implementation

6.1 Synthetic datasets

As mentioned earlier, the training with the synthetic datasets only has the purpose of evaluating the behaviour of the model and to show its limitations.

6.1.1 ATS

The architectures used for the scorer network and the feature network are the same as those used in the ATS article [16] and shown below.

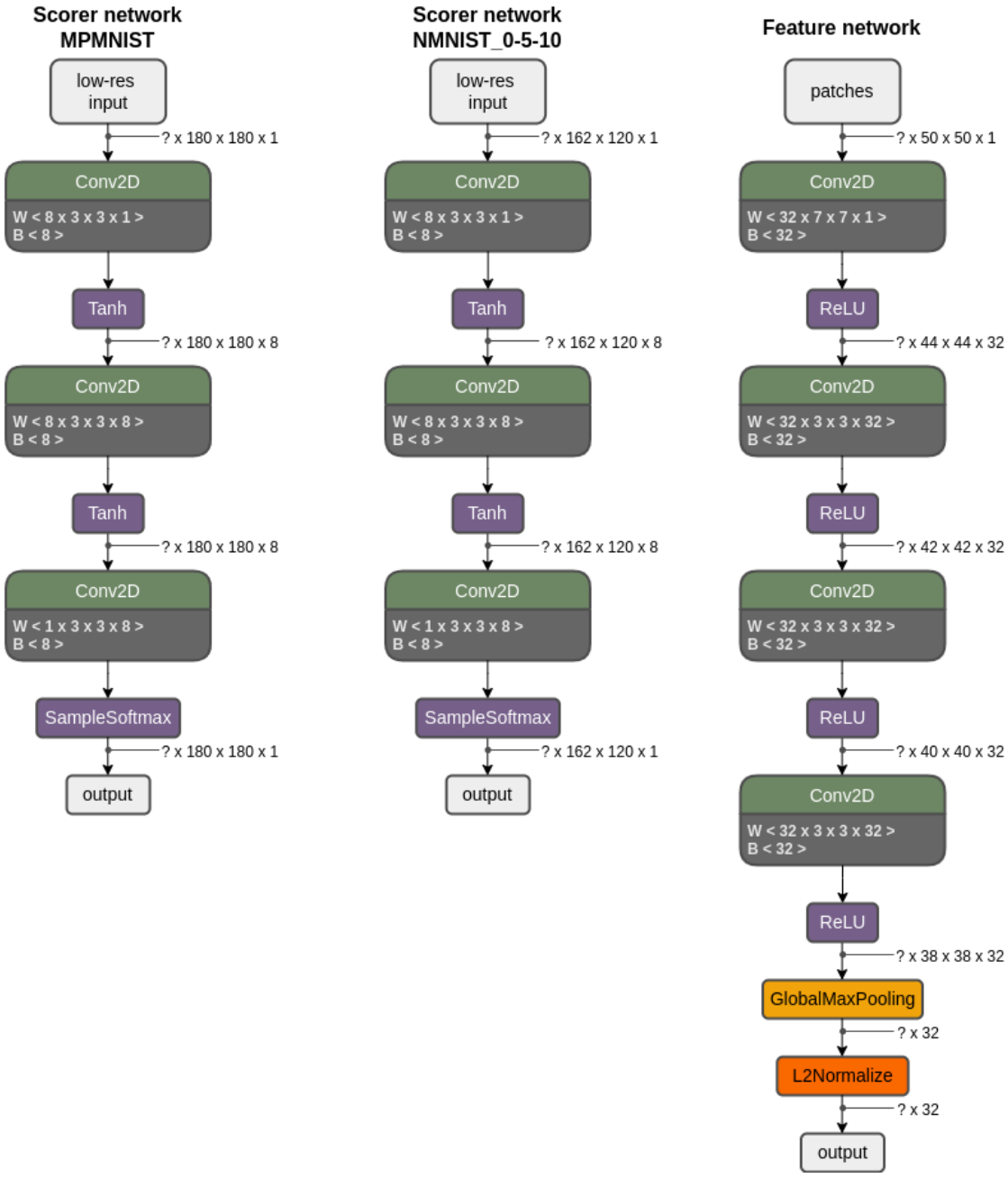


Figure 6.1: Scorer network and feature network architectures used in the ATS model when trained on the synthetic datasets.

For the convolutional layers in the scorer network, the padding parameter is set to *same* (same input-output dimensions), while for the convolutional layers in the feature network, the padding is set to *valid* (no pad), the stride is always 1. *SampleSoftmax* is a user-defined activation that performs the softmax also considering the channel dimension. *L2Nornalize* is a user-defined layer that performs the normalization of the features calculated for each patch, as required by the formulation in the previous chapter.

In the following table are shown the hyperparameter values used with the synthetic datasets.

	MPMNIST	NMNIST_0	NMNIST_5	NMNIST_10
scale	0.12	0.125	0.125	0.125
num patches	10	10	10	10
patch size	50	50	50	50
regularizer strength	0.01	0.005	0.005	0.005
learning rate	0.001	0.001	0.001	0.001
clip norm	1	1	1	1
epochs	180	9	76	91
batch size	128	128	128	128

Table 6.1: ATS hyperparameters used with synthetic datasets

The whole model is trained three times on the same training dataset and the results are eventually averaged. Accuracy is a good metric for the type of task for which the datasets were created, as the classes are balanced. The input images are simply scaled between 0 and 1 by dividing by 255, which is the maximum value each pixel can take. In the following images are represented both average loss and accuracy curves computed on the training set.

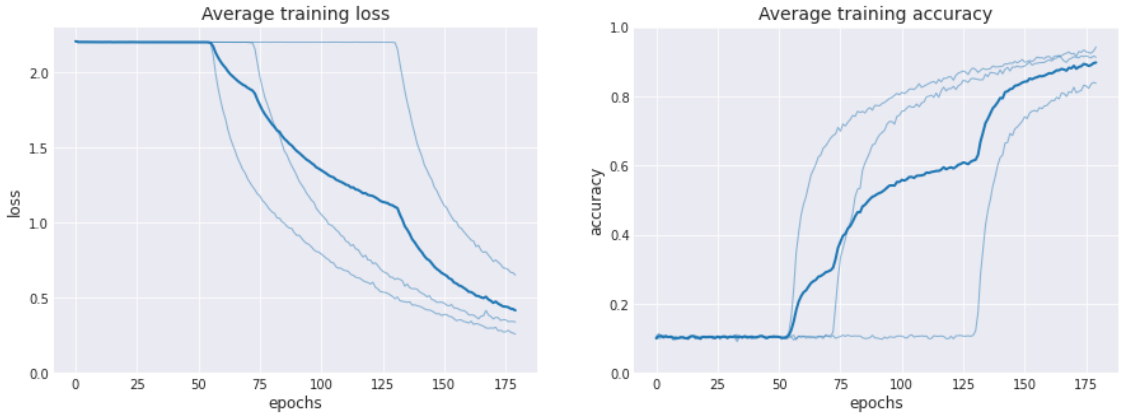


Figure 6.2: Average training accuracy and loss curves on MPMNIST.

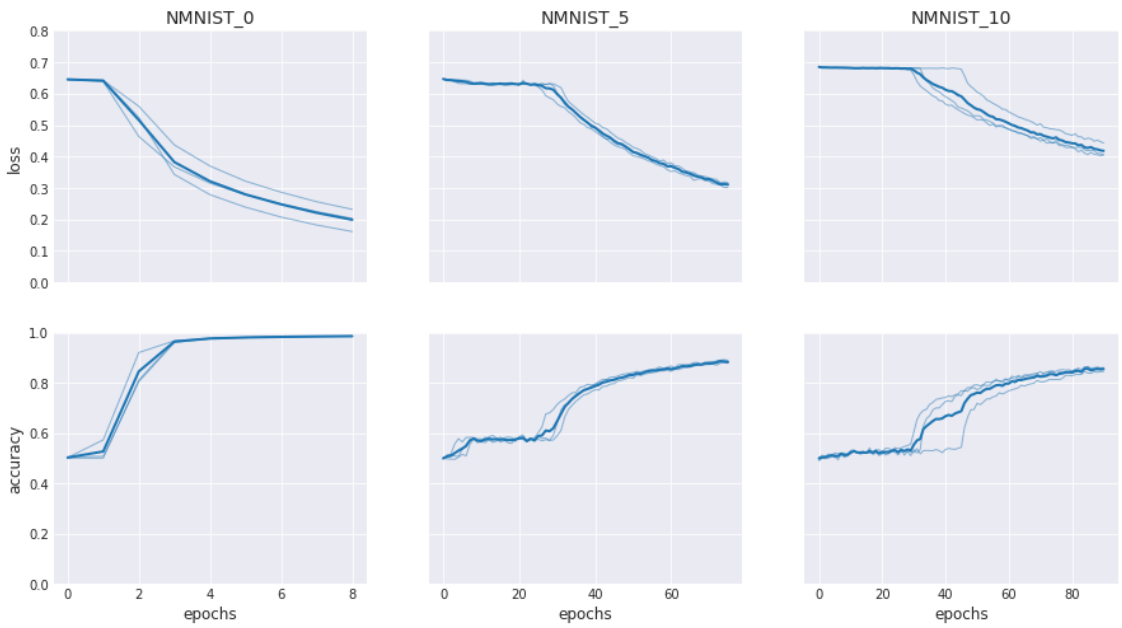


Figure 6.3: Average accuracy and loss curves on NMNIST_0, NMNIST_5 and NMNIST_10.

In the previous figures, the light blue curves refer to the individual runs, while the dark blue curve represents their average. In general, training for this model can be difficult because to improve its predictive ability the feature network needs to be fed with good patches and the scorer network needs the other to correctly classify the patches. From the figures 6.2 and 6.3 it can be seen that the ATS tends not to improve up to a certain point: the reason for this behaviour is that the good patches need to be sampled a few times in a row before the scorer network understands that these regions are more important than others.

In the following table, both the average loss and the average accuracy, as measured on the training set and the test set, are given with their standard deviation[15].

		MPMNIST	NMNIST_0	NMNIST_5	NMNIST_10
accuracy	train	0.899 ± 0.041	0.986 ± 0.000	0.880 ± 0.008	0.860 ± 0.008
	test	0.759 ± 0.011	0.986 ± 0.000	0.865 ± 0.006	0.842 ± 0.011
loss	train	0.396 ± 0.168	0.189 ± 0.028	0.310 ± 0.006	0.414 ± 0.020
	test	0.753 ± 0.039	0.189 ± 0.028	0.337 ± 0.004	0.439 ± 0.012

Table 6.2: Training and test results for the synthetic datasets. The average values and the standard deviation are computed on three runs.

From the reported results, it appears that in the case of NMNIST, the performance decreases as the number of distortions in the images increases. To better understand the behaviour of the model during training, both attention map and

[15]In article [16], the exact results for NMNIST training are not given in a table, instead loss and error (1 - accuracy) charts for a single ATS run can be viewed at `https://github.com/idiap/attention-sampling/tree/master/docs/img`, accessed 17/06/2022.

patches computed on some previously stored images are collected and showed below.

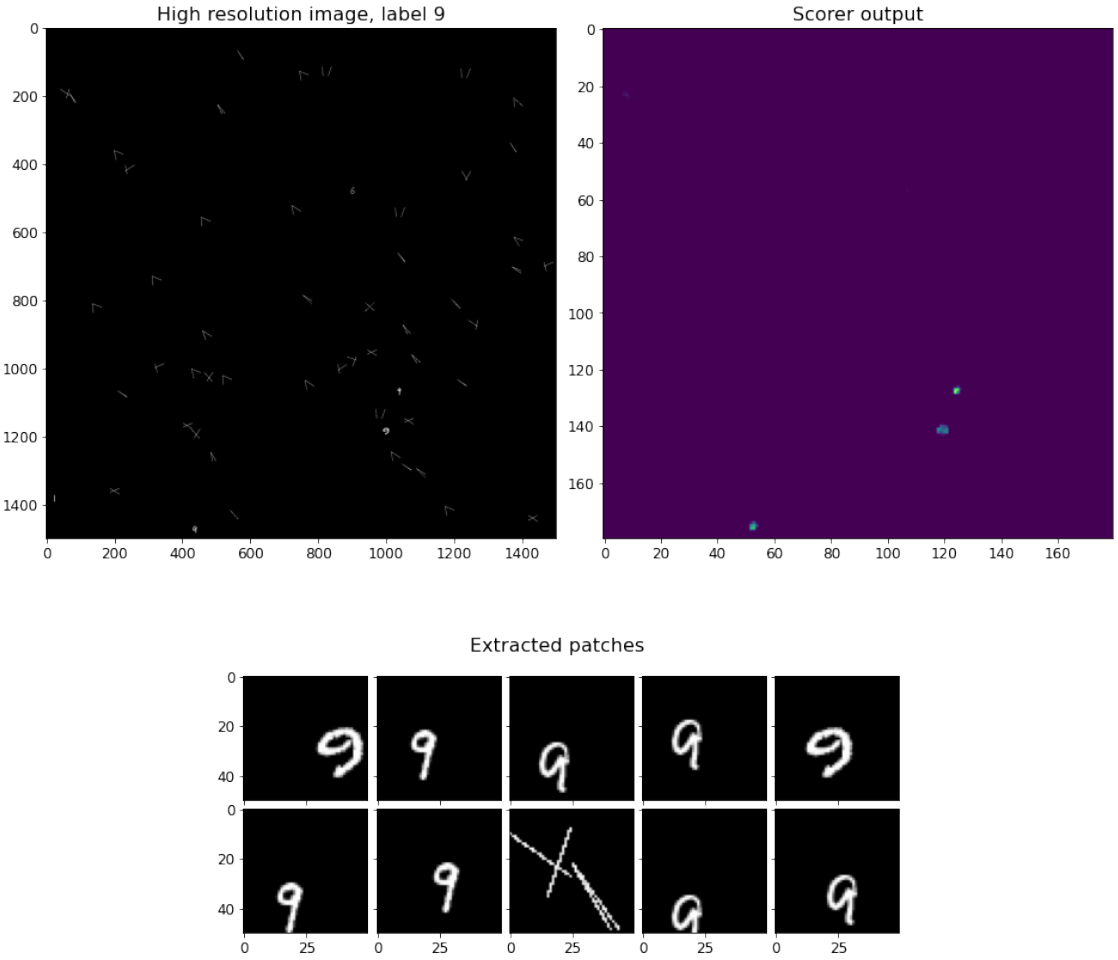


Figure 6.4: On the top left a MPMNIST input image taken from the training set, on the top right the attention map computed by the score network on the last epoch, below the extracted patches.

In the case of NMNIST the attention map clearly highlights the regions that contains a class label 9 in the input image and the patches are extracted consistently.

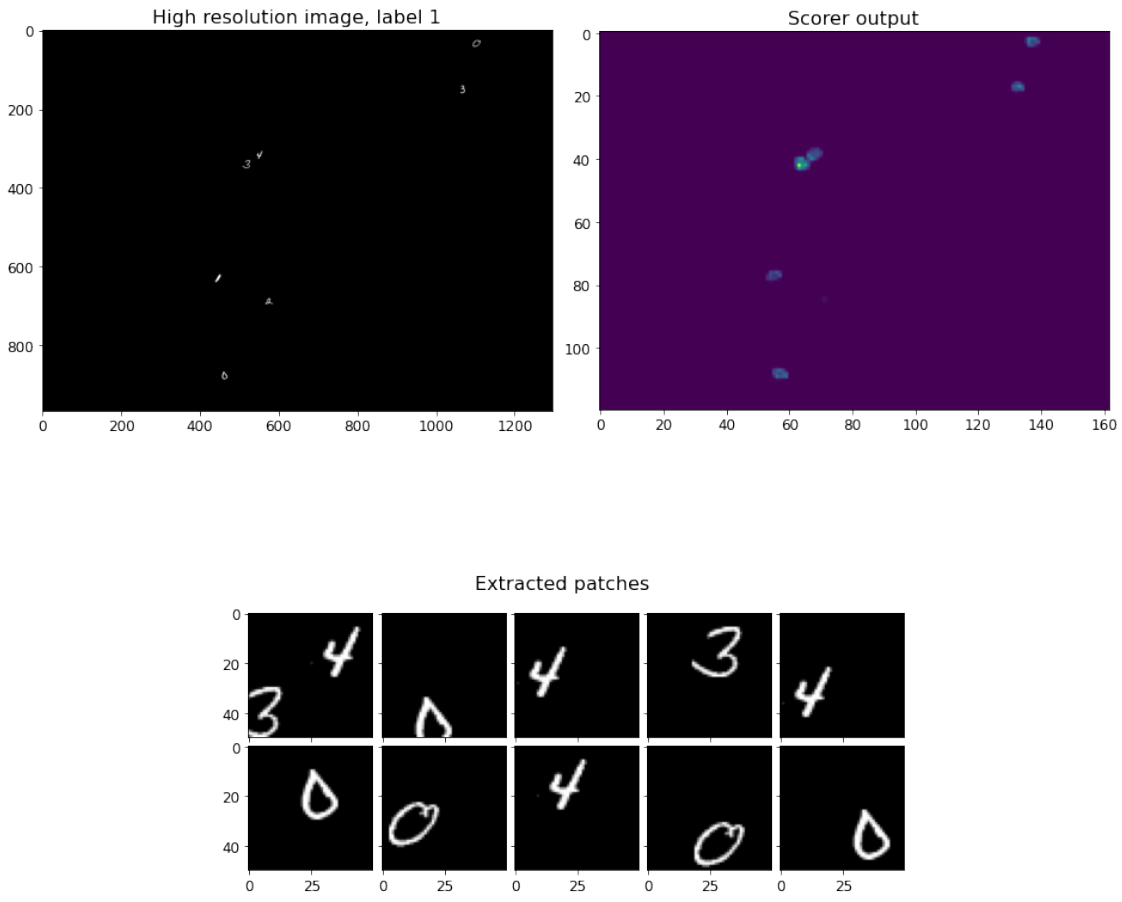


Figure 6.5: On the top left a NMNIST_5 input image taken from the training set, on the top right the attention map computed by the score network on the last epoch, below the extracted patches.

In the case of NMNIST_5, the attention map can distinguish the background from the digits, but does not focus only on the regions containing the target digit 3 in the input image, even if one of the two instances receives a higher attention. Only two of the extracted ten patches contain the target digit 3. This suggests us that the lower the noise in the input image, the better the model can focus on the informative regions.

6.1.2 DPS

In the architectures used for the scorer network and the feature network there are some minor differencies with respect to those of the ATS case.

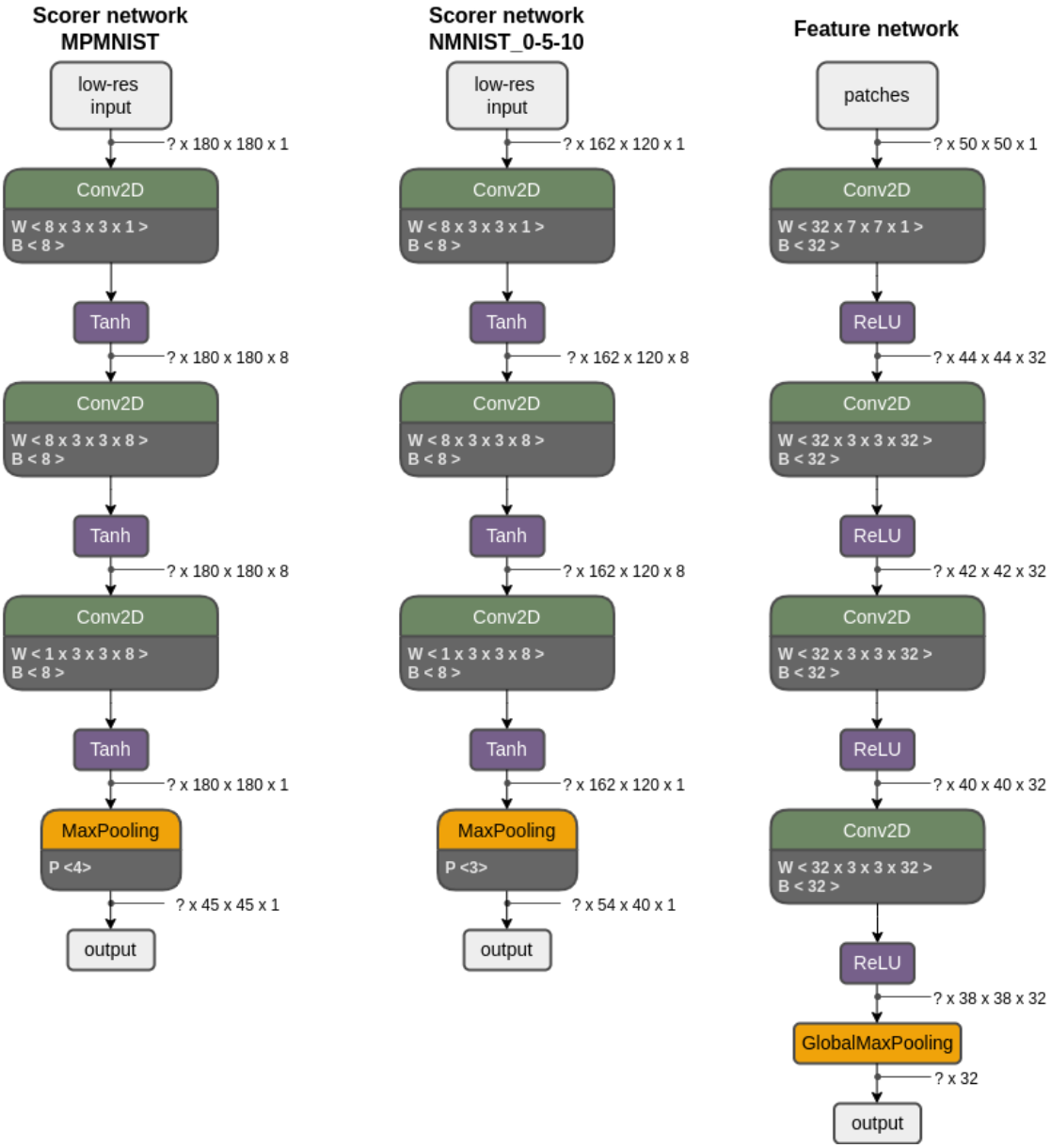


Figure 6.6: Scorer network and feature network architectures used in the DPS model when trained on the synthetic datasets.

In the scorer network, the *SampleSoftmax* is replaced by a standard *tanh* activation function, followed by a *Maxpooling* layer. The latter has the effect of reducing the output size of the scorer, which is beneficial from a memory consumption point of view and avoids an OOM error during the training. In the feature network, the *L2Normalize* layer can be omitted.

The following table shows the hyperparameter values used with the synthetic datasets.

	MPMNIST	NMNIST_0	NMNIST_5	NMNIST_10
scale	0.12	0.125	0.125	0.125
num patches	10	10	10	10
patch size	50	50	50	50
regularizer strength	0.01	0.01	0.01	0.01
noise strength	0.05	0.05	0.05	0.05
num samples	250	250	250	250
learning rate	0.001	0.001	0.001	0.001
clip value	0.05	1	1	1
warmup ratio	0.1	0.	0.	0.
epochs	20	2	10	20
batch size	64	64	64	64

Table 6.3: DPS hyperparameters used with synthetic datasets

In the reference article [4] the authors use 500 noise sample but to meet the memory constraints of Google Colab their number is reduced to 250; also the *batch size* is reduced to 64 for the same reason. The *warmup ratio* does not seems to particularly affect the training, both *warmup ratio* and *clipvalue* are setted to a value that let the model converge to a solution.

In the following images are represented both average loss and accuracy curves computed on the training set.

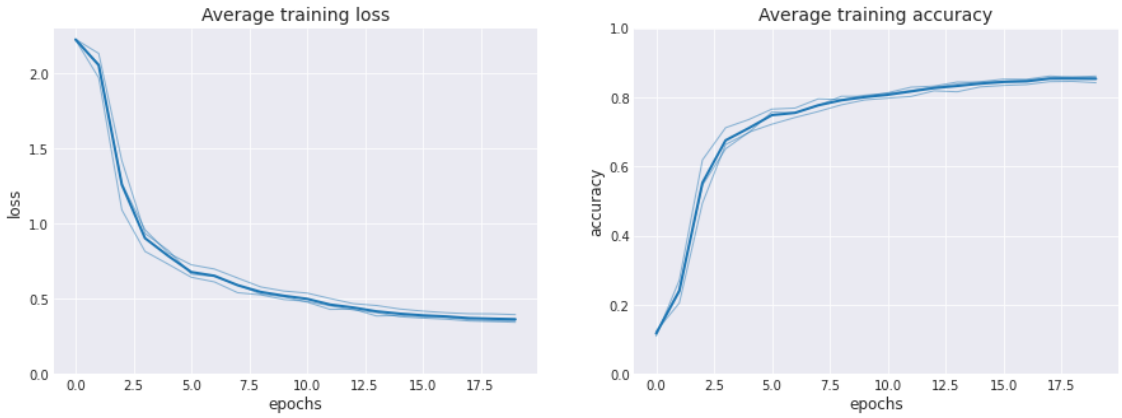


Figure 6.7: Average training accuracy and loss curves on MPMNIST.

The charts show that the model starts learning from the beginning thanks to the weighted average of the patches computed by the Differentiable-TopK during the first training steps. In terms of variability, the DPS seems to be more stable than the ATS, on the other hand, training takes much longer.

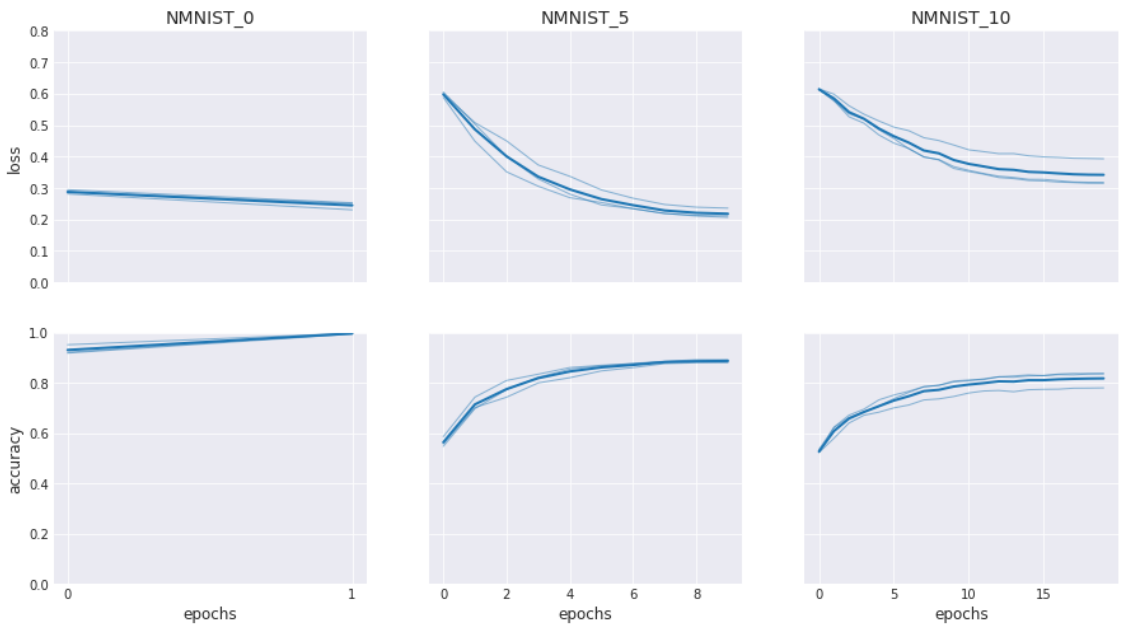


Figure 6.8: Average training accuracy and loss curves on NMNIST_0, NMNIST_5, NMNIST_10.

The following table shows both average accuracy and loss computed on training and test sets.

		MPMNIST	NMNIST_0	NMNIST_5	NMNIST_10
accuracy	train	0.840 ± 0.009	0.997 ± 0.001	0.872 ± 0.024	0.823 ± 0.028
	test	0.765 ± 0.029	0.994 ± 0.000	0.869 ± 0.021	0.834 ± 0.014
loss	train	0.435 ± 0.023	0.238 ± 0.017	0.248 ± 0.050	0.330 ± 0.038
	test	0.763 ± 0.107	0.246 ± 0.017	0.249 ± 0.048	0.308 ± 0.027

Table 6.4: Training and test results for the synthetic datasets. The average values and the standard deviation are computed on three runs.

The accuracy results of ATS and DPS are comparable when looking at their standard deviations.

For completeness, the attention and the patches extracted from some images are reported below. The images are the same as in the ATS case.

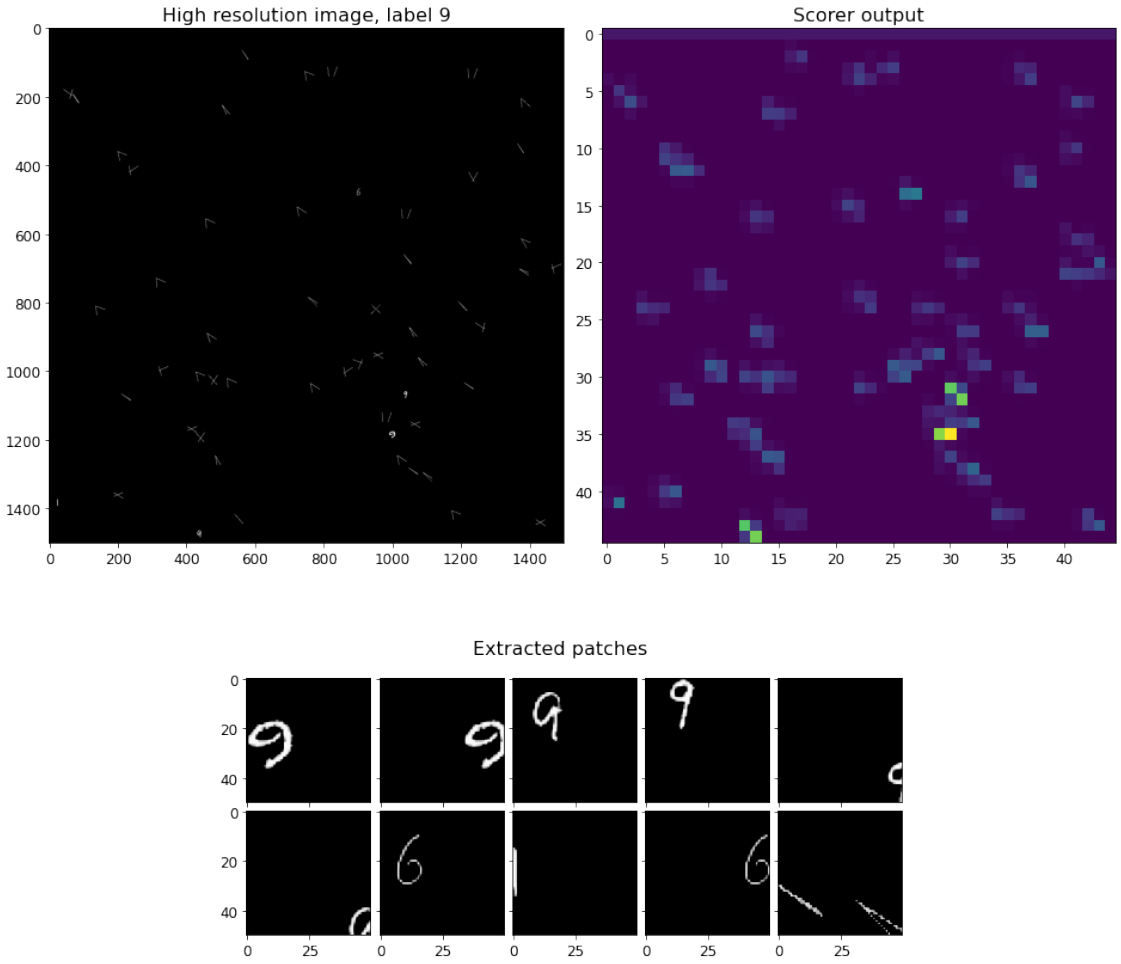


Figure 6.9: On the top left a MPMNIST input image taken from the training set, on the top right the attention map computed by the score network on the last epoch, below the extracted patches.

As in the ATS case the more important regions are those that contains the class label digit 9. The output of the scorer is now smaller so the patches are less densely distributed and this can be seen from the extracted patches: only four of them contain the 9 digit.

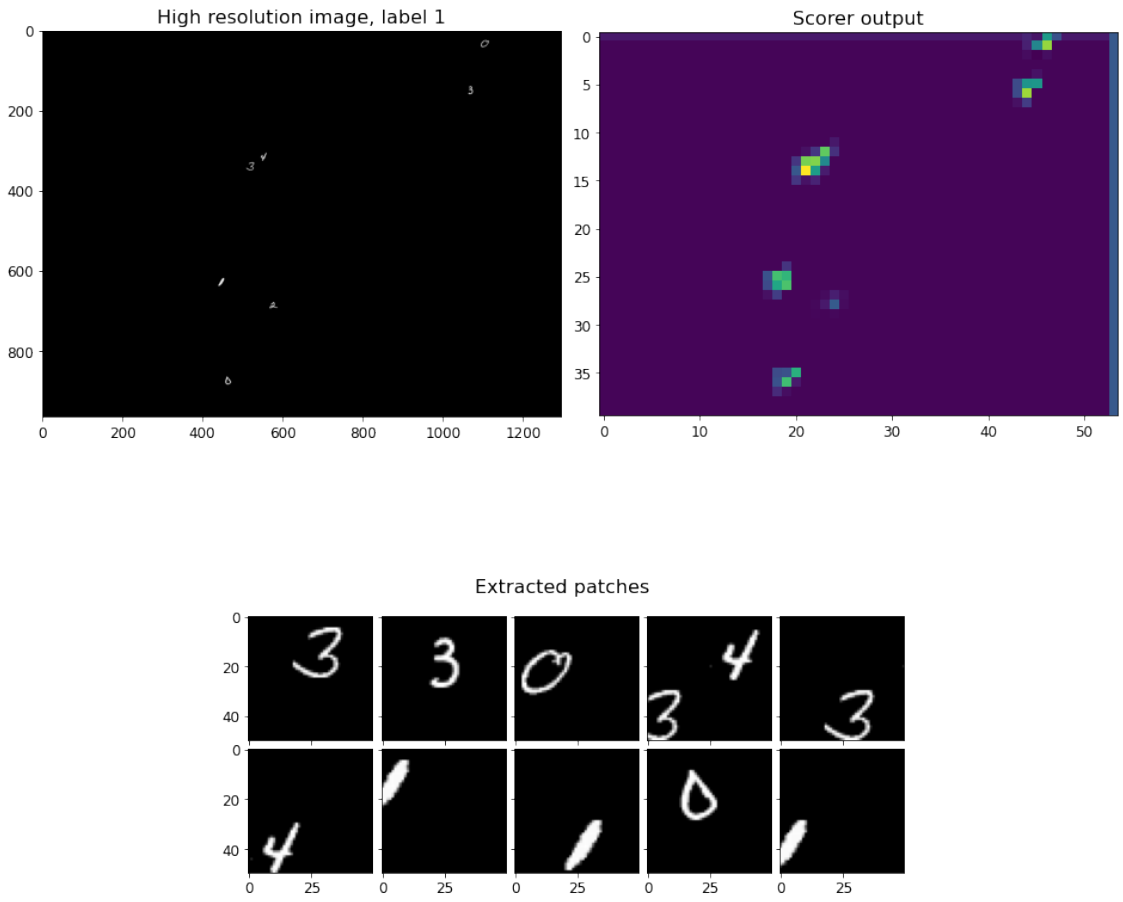


Figure 6.10: On the top left a NMNIST_5 input image taken from the training set, on the top right the attention map computed by the score network on the last epoch, below the extracted patches.

As for the ATS case the DPS can distinguish the background from the actual digits but cannot focus only on the two instancies of the tarfet digit 3, even if one of them gets a higher attention.

6.2 PRISMA dataset

6.2.1 Feature network selection

The selection of a good feature network is made using the *small56x56_ augmented* and *small28x28_ augmented* datasets presented in the previous chapter. The decision to select the feature network separately has several reasons: first, both ATS and DPS require a certain amount of time to train, which, combined with the limited resources provided by Google Colab, prevents a comprehensive study in this direction; second, we need to consider that it may be beneficial for training the entire network if it starts with a set of pre-trained weights for the feature network; furthermore, if the models are ultimately not suitable for the task, we can be sure that we at least have a network that can classify the patches well, and we need to reconsider another strategy for patch extraction.

In the first part of this analysis, the two small datasets with manually selected patches are used to find out which of the following scenarios is better in terms of processing time and classification capability of the feature network: keeping the high-resolution images as they are, extracting 56×56 patches, or resizing the high-resolution images by a factor of $1/2$, extracting 28×28 patches. All the average processing times reported in this section are measured when Google Colab provides us with the NVIDIA Tesla K80 GPU.

In order to retain the most of the data, the feature network models are trained with a K-fold cross-validation with K=7. Due to the fact that the datasets are unbalanced, the metrics *precision*, *recall* and *AUC-PR* are used to evaluate the models. For each model, the number of parameters that can be trained and the average processing time measured when a batch of 10 images is input (a maximum of 10 patches are selected for the whole model) are also given. The general structure of the feature network is shown in figure 6.11.

The first step consists in defining different variants of the proposed architecture, obtained by varying the number of filters f of the convolutional layers, the type of convolution (Standard Convolution vs. Depthwise Separable Convolution), the number of blocks and the number of units in the first dense layer (omitted when zero). For all convolutional layers, the *stride* is set to 1, the *padding* is set to *same* and the *kernel size* is set to 3. Ten configurations, five for each convolution type, are randomly defined. After a suitable architecture is found, four more configurations are defined manually to refine the result. The following tables list all the configurations for each type of convolution. The yellow configurations are randomly defined, the blue ones manually defined.

	Standard Convolution								
model name	model 0	model 1	model 2	model 3	model 4	model 5	model 6	model 7	model 8
no. blocks	4	4	2	3	2	3	3	3	3
no. filters	32 128 128 32	64 128 128 128	128 64	64 64 64	64 64	16 32 64	32 32 64	64 64 64	64 64 64
no. dense units	0	256	0	0	0	256	256	256	128

Table 6.5: Architecture configurations for the Standard Convolution.

	Depthwise Separable Convolution				
model name	model 0	model 1	model 2	model 3	model 4
no. blocks	2	4	3	2	2
no. filters	32 128	64 128 64 64	128 128 32	128 128	128 64
no. dense units	256	256	256	0	0

Table 6.6: Architecture configurations for the Depthwise Separable Convolution.

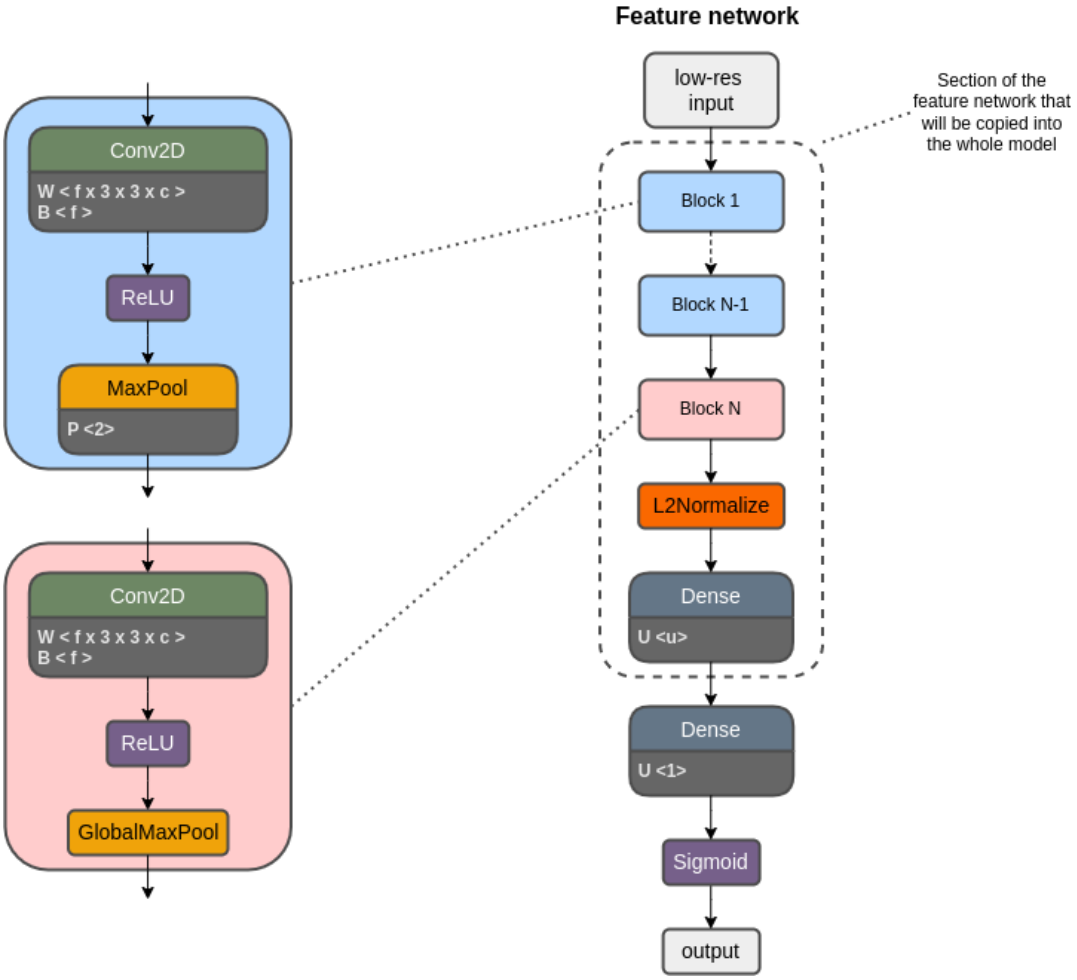


Figure 6.11: General feature network architecture used during the selection process. W stands for weights, B for bias, P for pool size and U for units.

In the previous tables, the configurations highlighted in yellow are used to define the first 10 models. In the first place, the average inference time for a batch of 10 images is measured both for image size 56×56 and 28×28.

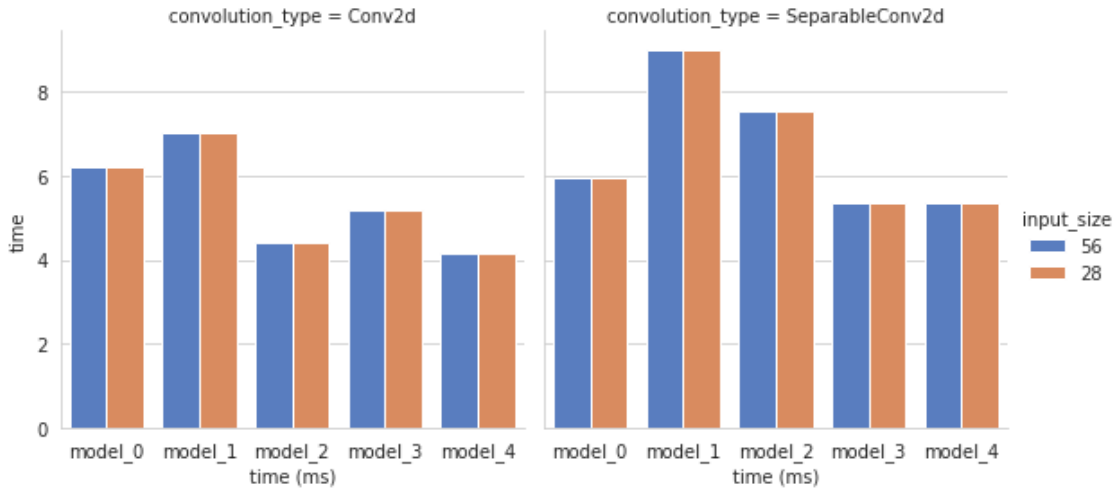


Figure 6.12: Average inference time measured on 1000 repetitions.

The chart shows that the difference between the average inference times for the two sizes of the input image is negligible. Maintaining the original resolution of the high resolution images during the final experiments may be advantageous, so from now on only the *small56x56_augmented* dataset will be used. Comparing the evaluation metrics of the ten models we can see that when using the standard convolution there are better performances.

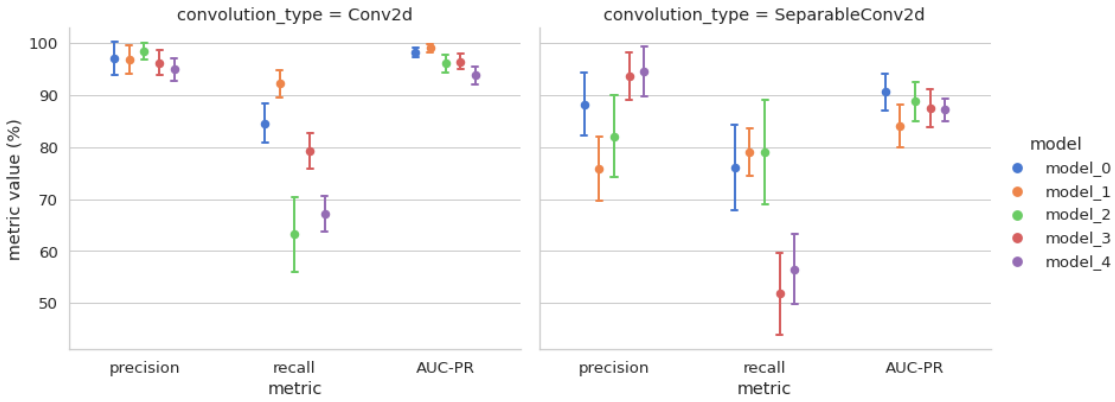


Figure 6.13: Evaluation metrics for the first ten models. The values collected at each different fold are averaged. The standard deviation is encoded into the bars height.

The diagram shows that the best models are model_0, model_1 and model_3, which is also the fastest among the three. It makes sense to look for a configuration of the network that is a refined version of model_3. For this reason, the configurations highlighted in blue in table 6.5 have been defined.

After the new define models are trained with the same procedure, the collected results of all the models that use the Standard Convolution, whose configurations are reported in table 6.5, can be compared.

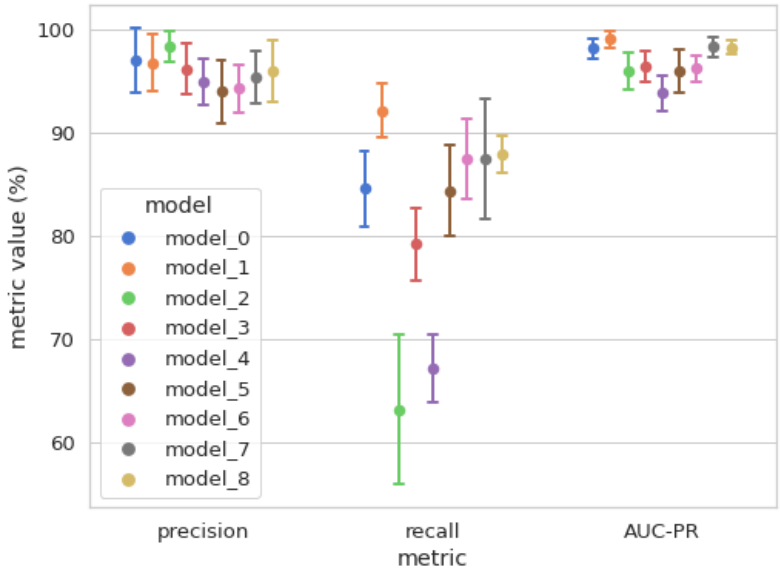


Figure 6.14: Evaluation metrics for all the models that use the Standard Convolution.

The previous diagram shows that both *precision* and *AUC-PR* are quite high, so we can use the *recall* metric as discriminative metric, and also include information about the number of parameters and the average processing time in the following chart.

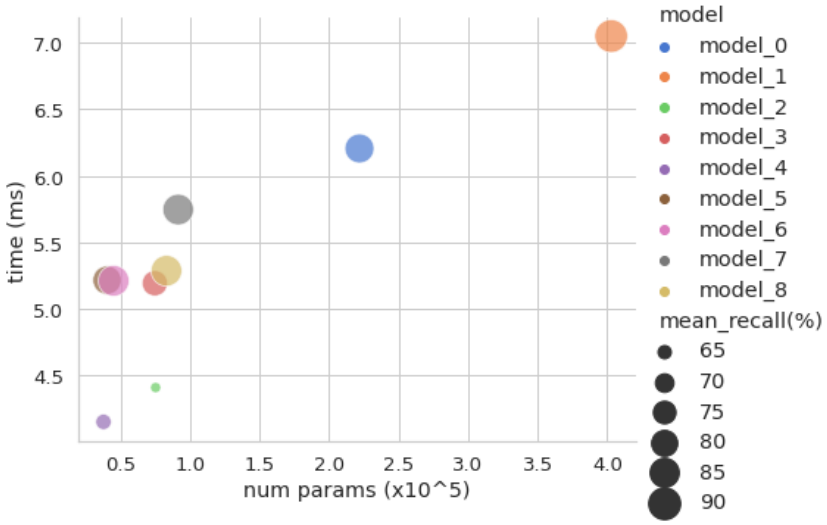


Figure 6.15: Relational chart that shows average inference time vs number of trainable parameters of the Standard Convolution models, encoding the recall into the cicle size.

The configuration used to define the model_8 is a good balance between evaluation metrics, inference time and number of trainable parameters. In the following table are reported the results of all the models evaluate so far.

convolution type	model name	no. parameters	inference time (ms)	accuracy	precision	recall	AUC-PR
Standard Convolution	model 0	221825	6.2 ± 1.0	0.941 ± 0.016	0.97 ± 0.032	0.846 ± 0.037	0.982 ± 0.01
	model 1	402945	7.0 ± 1.2	0.964 ± 0.011	0.968 ± 0.028	0.922 ± 0.026	0.991 ± 0.008
	model 2	75137	4.4 ± 0.8	0.876 ± 0.022	0.984 ± 0.015	0.632 ± 0.073	0.961 ± 0.018
	model 3	74561	5.2 ± 1.0	0.922 ± 0.015	0.962 ± 0.024	0.792 ± 0.035	0.964 ± 0.015
	model 4	37633	4.2 ± 1.0	0.882 ± 0.013	0.95 ± 0.022	0.672 ± 0.033	0.939 ± 0.017
	model 5	40193	5.2 ± 0.9	0.932 ± 0.021	0.94 ± 0.031	0.844 ± 0.044	0.96 ± 0.021
	model 6	44961	5.2 ± 0.8	0.942 ± 0.013	0.943 ± 0.023	0.875 ± 0.038	0.963 ± 0.013
	model 7	91393	5.7 ± 2.9	0.945 ± 0.02	0.954 ± 0.025	0.875 ± 0.058	0.984 ± 0.01
	model 8	82945	5.3 ± 0.9	0.948 ± 0.008	0.96 ± 0.029	0.879 ± 0.018	0.983 ± 0.006
Depthwise Separable Convolution	model 0	37866	6.0 ± 1.0	0.888 ± 0.033	0.882 ± 0.061	0.761 ± 0.082	0.906 ± 0.036
	model 1	40074	9.0 ± 1.3	0.846 ± 0.021	0.758 ± 0.062	0.79 ± 0.046	0.841 ± 0.041
	model 2	31914	7.5 ± 1.2	0.871 ± 0.033	0.821 ± 0.078	0.79 ± 0.101	0.888 ± 0.037
	model 3	18058	5.4 ± 1.0	0.831 ± 0.028	0.936 ± 0.045	0.518 ± 0.079	0.875 ± 0.036
	model 4	9738	5.4 ± 0.9	0.847 ± 0.025	0.945 ± 0.047	0.565 ± 0.067	0.872 ± 0.022

Table 6.7: Summary table of all the models defined so far, the model_8 with Standard Convolution is the best among them.

So far we have found a good architecture for the feature network of the whole model. Starting from this architecture, firstly we try to add the Dropout regularization after the first Dense layer defining three different dropout levels:

- model_0: 0.1,
- model_1: 0.2,
- model_2: 0.4.

Three different methods of dealing with the problem of class imbalance are explored to improve performance and find an approach that can also be used in training the whole final models (ATS and DPS) on the *final_augmented* dataset:

- **Class weight** - Errors of the two classes are weighted differently according the following weights

$$w_{neg} = \frac{n_{tot}}{n_{neg}} \cdot 0.5 \tag{6.1}$$

$$w_{pos} = \frac{n_{tot}}{n_{pos}} \cdot 0.5 \tag{6.2}$$

  where n_{tot} is the size of the training set, n_{neg} and n_{pos} the number of elements of the negative (no fireball) and positive (fireball) class. This way the errors on the elements of the minority class, the positive in our case, are more penalized.

- **Oversampling** - The elements of the minority class in the training set are randomly resampled with replacement until the size of the minority class is equal to the size of the majority class, this way the classes in the training set are balanced.

- **Focal Loss function** - The Focal Loss function is proposed in [18] and addresses class imbalance during training applying a modulating term to the cross-entropy loss in order to focus learning on hard misclassified examples according the following expression

$$FL(p_t) = -\alpha(1 - p_t)^{\gamma} log(p_t). \tag{6.3}$$

where p_t is the predicted probability of the true class. It is essentially a dynamically scaled cross-entropy loss, where the scaling factor decays to zero as confidence in the correct class increases. The different combinations of (α, γ) tested are:

  - model_0: $(0.1, 2)$,
  - model_1: $(0.1, 5)$,
  - model_2: $(0.1, 7)$,
  - model_3: $(0.25, 2)$,
  - model_4: $(0.25, 5)$,
  - model_5: $(0.25, 7)$,
  - model_6: $(0.5, 2)$,
  - model_7: $(0.5, 5)$,
  - model_8: $(0.5, 7)$.

- **Transfer learning** and **fine tuning** - The best found architecture (Standard Convolution model_8) is pre-trained on the FASHION MNIST dataset for 10 epochs, using the Adam optimizer with learning rate 0.0001 and batch size 64. The weights are saved and then loaded before training it with an oversampling approach on the *small56x56_augmented* for 20 epochs, the loaded parameters are freezed, only the last dense layer is trained at first. Eventually the whole network is unfreezed and a final fine tuning of other 10 epochs is done.

The following table shows all the results for the improvement experiments.

improvement	model name	accuracy	precision	recall	AUC-PR
Standard Conv	model 8	0.948 ± 0.008	0.96 ± 0.029	0.879 ± 0.018	0.983 ± 0.006
Dropout	model 0	0.94 ± 0.014	0.948 ± 0.035	0.866 ± 0.033	0.977 ± 0.011
	model 1	0.94 ± 0.008	0.947 ± 0.02	0.864 ± 0.02	0.979 ± 0.007
	model 2	0.94 ± 0.015	0.951 ± 0.027	0.859 ± 0.034	0.974 ± 0.011
Class weight	model 0	0.956 ± 0.009	0.927 ± 0.032	0.94 ± 0.023	0.985 ± 0.008
Oversampling	model 0	0.985 ± 0.006	0.974 ± 0.021	0.982 ± 0.013	0.997 ± 0.003
Focal Loss	model 0	0.927 ± 0.019	0.981 ± 0.031	0.79 ± 0.046	0.983 ± 0.014
	model 1	0.941 ± 0.015	0.984 ± 0.015	0.833 ± 0.044	0.987 ± 0.006
	model 2	0.933 ± 0.017	0.997 ± 0.007	0.797 ± 0.053	0.987 ± 0.008
	model 3	0.941 ± 0.007	0.967 ± 0.032	0.85 ± 0.046	0.986 ± 0.004
	model 4	0.943 ± 0.024	0.963 ± 0.03	0.859 ± 0.067	0.986 ± 0.01
	model 5	0.937 ± 0.024	0.972 ± 0.021	0.833 ± 0.079	0.985 ± 0.011
	model 6	0.951 ± 0.019	0.949 ± 0.014	0.9 ± 0.064	0.987 ± 0.006
	model 7	0.949 ± 0.017	0.938 ± 0.028	0.904 ± 0.037	0.983 ± 0.007
	model 8	0.932 ± 0.012	0.933 ± 0.031	0.855 ± 0.03	0.974 ± 0.011
Transfer learning	model 0	0.821 ± 0.032	0.684 ± 0.061	0.857 ± 0.019	0.816 ± 0.035
Fine tuning	model 0	0.975 ± 0.01	0.943 ± 0.022	0.984 ± 0.012	0.991 ± 0.004

Table 6.8: The green row shows the baseline results of the best found architecture before the improvements. The red row shows the improvement the yield the best results.

The oversampling approach produces the best results, so it will be the approach also used when training the final models. The feature network architecture we

selected before is eventually trained on the entire *small56x56_augmented* dataset with the oversampling of the positive class and then the trained weights are saved.

6.2.2 ATS and DPS

Due to the high number of hyperparameters and the training duration, the exploration of the best configuration is made varying only the patch size, the number of extracted patches, the number of epochs and the scorer architecture, while the other hyperparameters are fixed to values that allow the model to converge. In the following tables can be observed the fixed hyperparameters values, the scorer network architectures and the configurations used to train the models.

	ATS	DPS
scale	0.15	0.125
regularizer strength	0.01	0.01
noise strength	-	0.5
num samples	-	100
learning rate	0.0005	0.0005
clipnorm	1	-
clip value	-	0.5
warmup ration	-	0.
batch size	64	64

Table 6.9: Fixed hyperparameters for ATS and DPS

ATS			DPS	
scorer_0	scorer_1	scorer_2	scorer_0	scorer_1
Conv 3×3, 8	Conv 3×3, 8	Conv 3×3, 8	Conv 3×3, 8	Conv 3×3, 8
Conv 3×3, 8	Conv 3×3, 8	Conv 3×3, 16	Conv 3×3, 8	Conv 3×3, 8
Conv 3×3, 8	Conv 3×3, 8	Conv 3×3, 32	Conv 3×3, 8	Conv 3×3, 8
Conv 3×3, 1	Conv 3×3, 1	Conv 3×3, 1	Conv 3×3, 1	Conv 3×3, 1
SampleSoftmax	MaxPooling 3	SampleSoftmax	MaxPooling 2	MaxPooling 3
	SampleSoftmax			

Table 6.10: Scorer network architectures used in ATS and DPS models. The convolutional layers, excluded the last one in the ATS models, are followed by a *tanh* activation function and have the padding parameter set to *same*.

model name	scorer network	patch size	num. patches	num. epochs
ats_prisma_set0	scorer_0	56	10	17
ats_prisma_set1	scorer_0	64	10	21
ats_prisma_set2	scorer _1	64	5	13
ats_prisma_set3	scorer_1	56	5	13
ats_prisma_set4	scorer_2	56	10	20
ats_prisma_set4_fw	scorer_2	56	10	17+3
dps_prisma_set0	scorer_0	56	10	7
dps_prisma_set1	scorer_1	56	5	10
dps_prisma_set2	scorer_1	64	5	10

Table 6.11: Different hyperparameters configurations used to train ATS and DPS models. The ats_prisma_set4_fw uses the same configuration of ats_prisma_set4 but the feature network weights saved in the feature network selection step are loaded: only the conv layers weights are loaded and freezed during the 17 epochs training, eventually a finetuning of 3 epochs is done with the learning rate lowered to 0.0001.

All the models are trained with the oversampling of the minority class (the fireball class) and for each configurations the results over 3 runs are collected. The DPS training is much more time and memory resources consuming than ATS, this is the reason why this last solution is better explored. The following charts reports the average value of the metrics and their standard deviation.

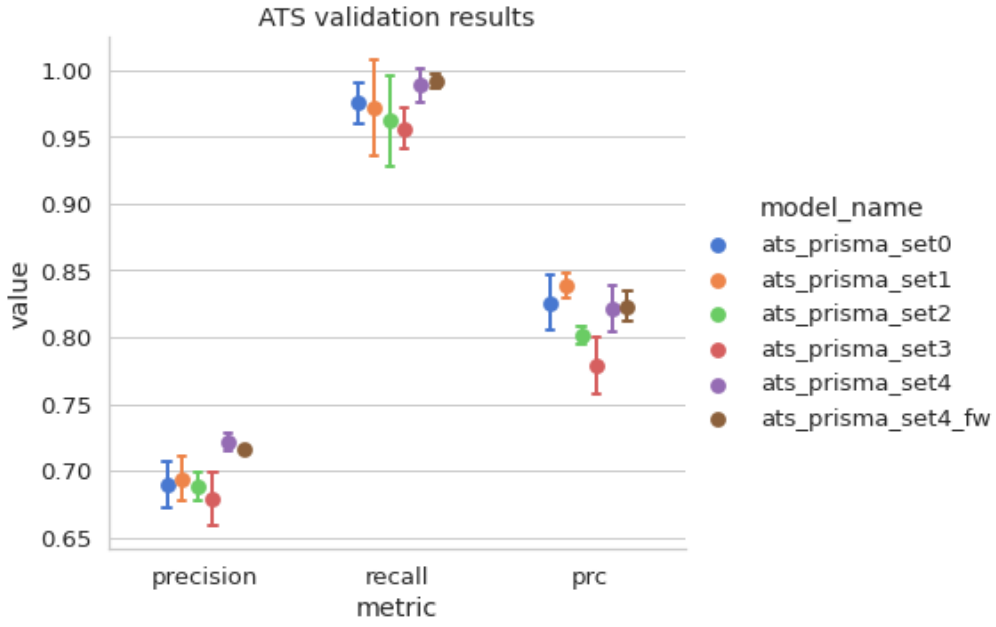


Figure 6.16: Evaluation metrics for all the final ATS models.

The previous chart shows that the ats_prisma_set4_fw gives the best results, still the gap between *precision* and *recall* is quite high, which means the model tends to predict a number of false positives. Given the similarity of the metric measures, the best model is the one with the higher recall value in this case.

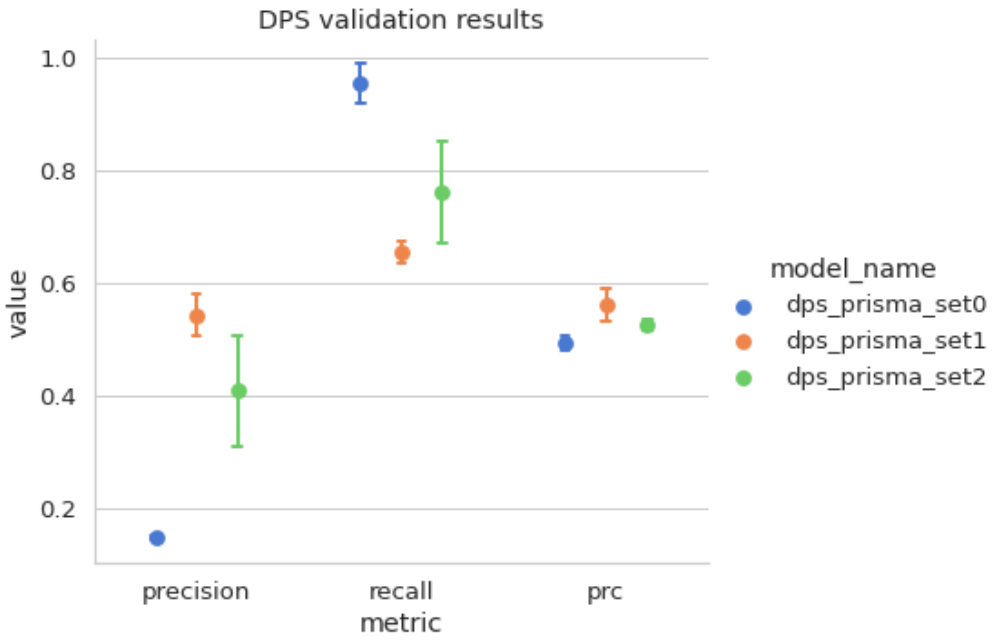


Figure 6.17: Evaluation metrics for all the final DPS models.

The best DPS model dps_prisma_set1 can reach much lower perfromances than the ATS best solution. For the DPS models the best models is the one that has at least the precision higher than 0.5.

The following table summarizes all the validation results of the final ATS and DPS models.

model name	no. params	inference time (ms)	accuracy	precision	recall	AUC-PR
ats_prisma_set0	84266	12.192±1.155	0.935±0.006	0.689±0.021	0.975±0.018	0.826±0.025
ats_prisma_set1	84266	12.347±1.320	0.936±0.009	0.694±0.020	0.972±0.044	0.838±0.011
ats_prisma_set2	84266	12.263±1.264	0.934±0.007	0.688±0.013	0.962±0.042	0.802±0.008
ats_prisma_set3	84266	12.155±1.171	0.931±0.007	0.678±0.024	0.956±0.018	0.779±0.025
ats_prisma_set4	89122	12.803±3.605	0.945±0.002	0.721±0.008	0.989±0.015	0.821±0.021
ats_prisma_set4_fw	89122	12.471±1.345	0.944±0.001	0.716±0.003	0.992±0.007	0.823±0.014
dps_prisma_set0	84266	12.171±1.370	0.236±0.046	0.149±0.005	0.955±0.044	0.493±0.016
dps_prisma_set1	84266	11.956±1.230	0.874±0.015	0.544±0.046	0.655±0.023	0.563±0.036
dps_prisma_set2	84266	12.104±1.275	0.791±0.086	0.409±0.122	0.763±0.111	0.526±0.013

Table 6.12: Final validation results summary, the best model is highlighted in red.

Concernings the inference times, they are comparable and are measured on a Nvidia Tesla T4 GPU: unfortunately the GPU are assigned automatically by Google Colab and during this last period the Nvidia Tesla K80 was never provided.

To better understand how the best model works we can observe some of the false positives reported below.

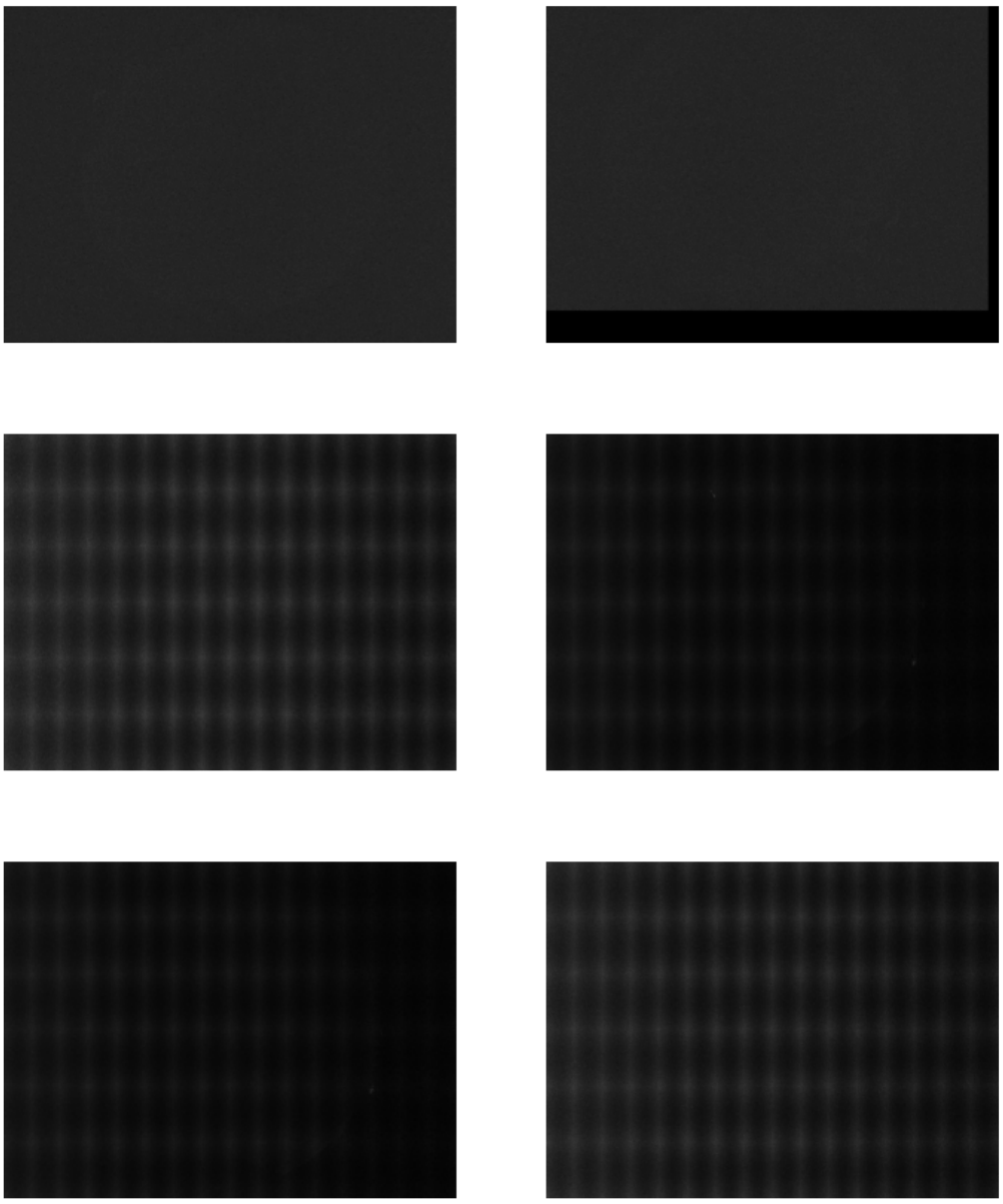


Figure 6.18: False positives on the validation with ats_prisma_set4_fw run 0.

These false positive suggest that the performance of the best model are biased due to the fact that the positive class is made primarily by noisy items. The over-sampling method probably highlights this behaviour, and instead of encouraging the selection of the actual informative regions, it promotes the classification of the positive class based only on the noisy patches that characterizes the class. As a demostration of this intuition, a noisy positive class items and its attention map is reported below.

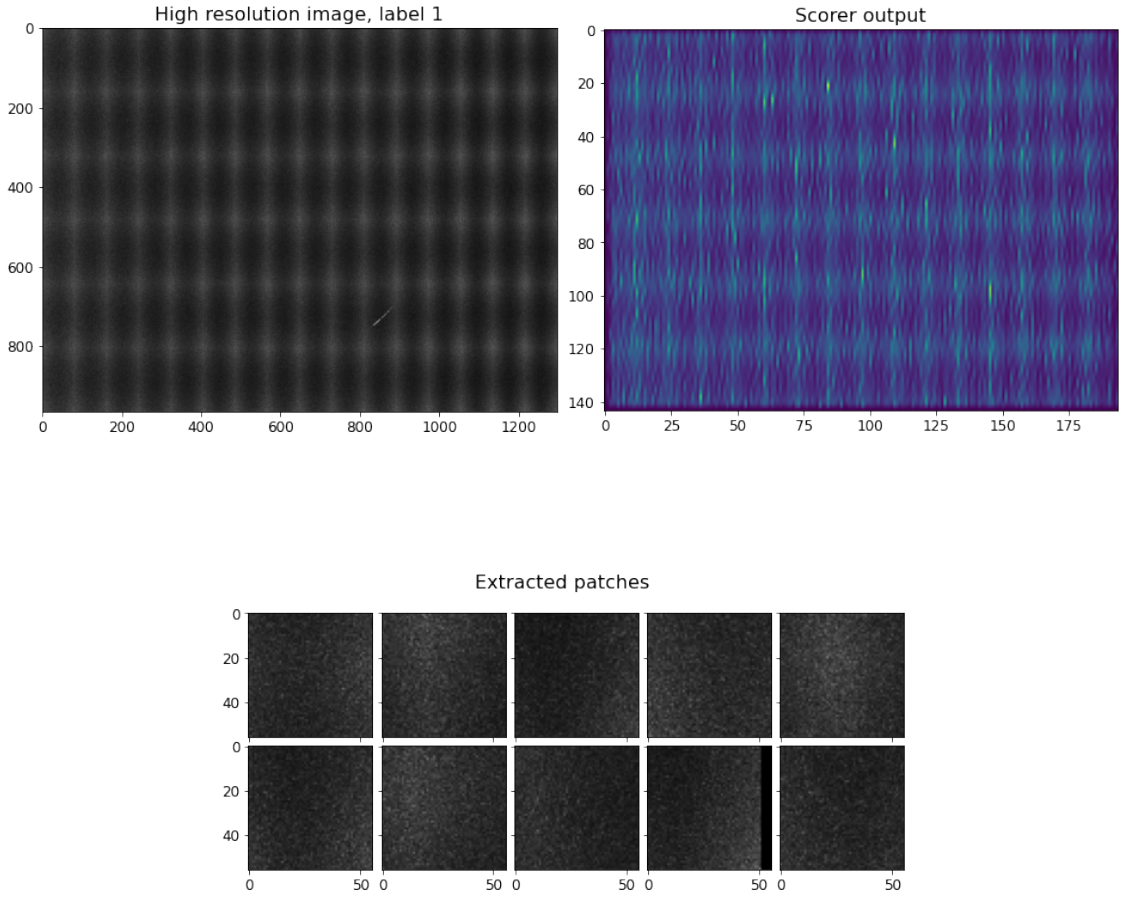


Figure 6.19: Attention and patches evaluated by ats_set4_fw (run 0) on a noisy positive class item.

The model is finally trained on the combination of training and validation set for the evaluation on the test set, which, as expected, reveals the weaknesses of the model. The performance evaluated on the test are:

- accuracy: 0.907
- precision: 0.982
- recall: 0.500
- AUC-PR: 0.602.

The recall value is 0.500 because half of the positive test items are noisy images, that are recognized by the model for the reason previously explained, the other half positive class items are not noisy and are misclassified as non-fireballs.

Chapter 7

Conclusion and future work

The creation of a real-time meteor detector remains an open problem. Although the two proposed models have not shown reliable performances for an actual application, the strategy of focusing only on the informative regions to perform the classification is an important goal to achieve in order to reduce the inference time for such large input images. A solution in this sense could be to abandon the idea of a model that automatically detects which regions are relevant and to use a supervised approach where the scorer and the feature network are treated as two separate entities: at this point, to select a good scorer, a similar training approach as for the object detection models can be used without the need to predict the bounding box, i.e., we should be able to classify the interesting/non-interesting regions of the scaled input given the ground truth "label" as a 2D array of 1s and 0s created knowing the position of the object in the image. The cost function for the single image could be the sum of the binary cross-entropy over all regions of the image. If the noisy images in the dataset are a problem and the noisy pattern appears to be constant, an appropriate approach could be similar to the one currently used to process the frames, but applied to the MP-M shown in 5.8, i.e., every 10 or 20 seconds, a MP image can be placed in a buffer of predetermined size, and when it is filled, the median image MMP can be calculated and subtracted from the next noisy MP-M. One of the provided events was long enough to provide an example, shown in figures 7.1, 7.2 and 7.3. Of course, the risk of the proprosed preprocessing changing the original informative region too much may affect the results and should be carefully considered.

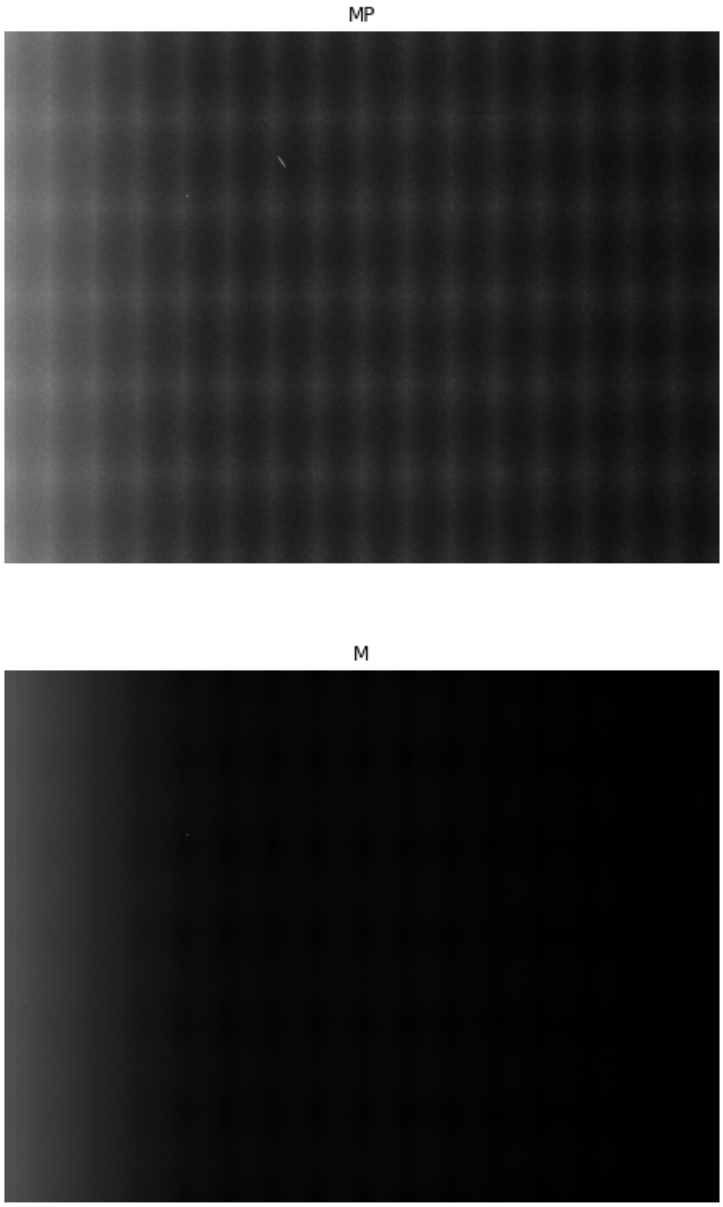


Figure 7.1: MP and M are the maxpixel and the median images computed over a buffer of 65 frames.

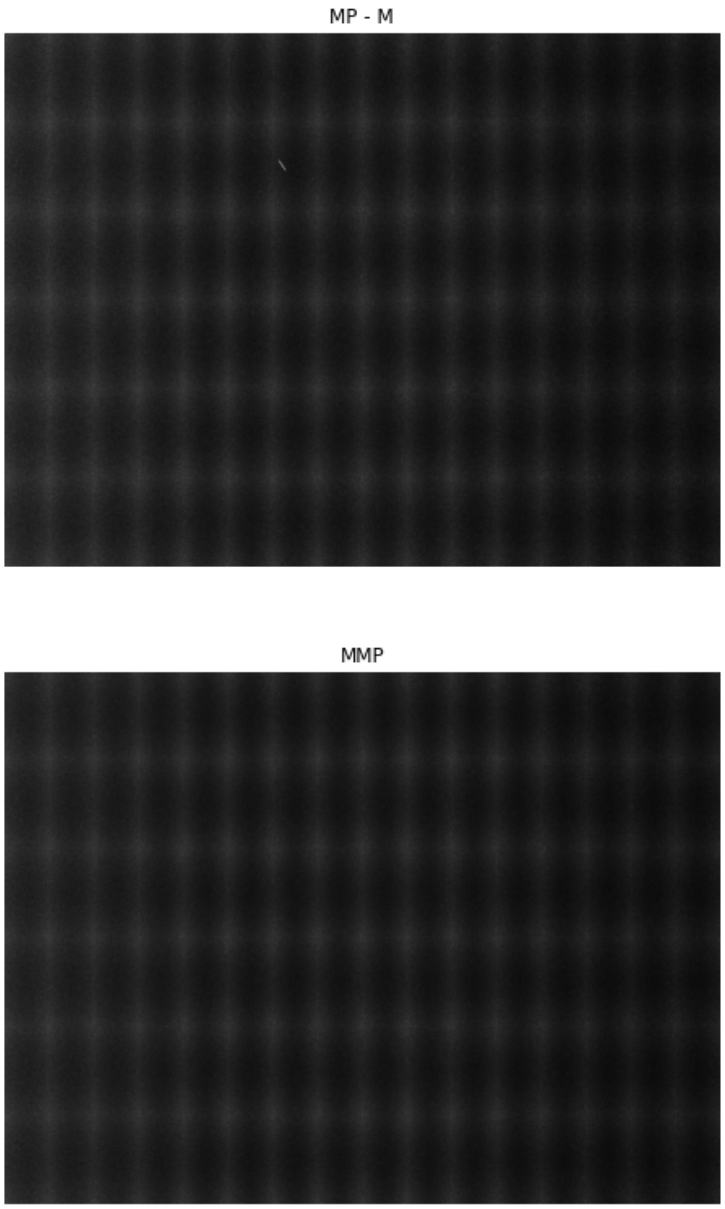


Figure 7.2: MP-M is the current preprocessed image, MMP is the median image computed over a buffer of 7 maxpixel images.

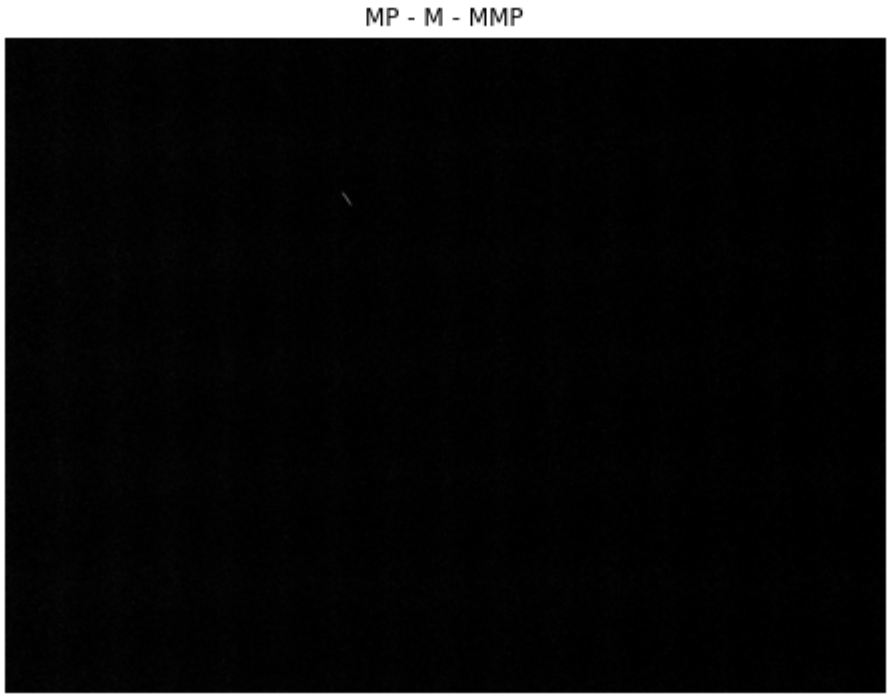


Figure 7.3: MP-M-MMP is the result of the new proposed preprocessing to attenuate the noise.

Appendix A

Attention-Sampling models

A.1 Minimum variance approximation

In this section we show that sampling K i.i.d. indices and using the unbiased estimator of the sample mean is an optimal approximation of the expected value of the population of features.

Let us denote by P a discrete probability distribution on the N feature vectors with probabilities p_i. We want to select the sample from P in such a way that the variance is minimized, i.e. we search $P^\star$ in such a way that

$$P^\star = \arg\min_P \mathbb{V}_{I\sim P}\left[\frac{a_{\Phi_a}(\mathbf{x})_I \cdot h_{\Phi_h}(\mathbf{x})_I}{p_I}\right] \tag{A.1}$$

where we divide by p_I to ensure that the expectation remains the same regardless of P. This property can be easily verified as follows

$$\mathbb{E}_{I\sim P}\left[\frac{a_{\Phi_a}(\mathbf{x})_I \cdot h_{\Phi_h}(\mathbf{x})_I}{p_I}\right] \tag{A.2}$$

$$= \sum_{i=1}^{N} p_i \cdot \frac{a_{\Phi_a}(\mathbf{x})_i \cdot h_{\Phi_h}(\mathbf{x})_i}{p_i} \tag{A.3}$$

$$= \sum_{i=1}^{N} a_{\Phi_a}(\mathbf{x})_i \cdot h_{\Phi_h}(\mathbf{x})_i \tag{A.4}$$

$$= \mathbb{E}_{I\sim a(\mathbf{x})}[h_{\Phi_h}(\mathbf{x})_I]. \tag{A.5}$$

We can now continue with our derivation as follows:

$$\arg\min_P \mathbb{V}_{I\sim P}\left[\frac{a_{\Phi_a}(\mathbf{x})_I \cdot h_{\Phi_h}(\mathbf{x})_I}{p_I}\right] \tag{A.6}$$

$$= \arg\min_P \left\{\mathbb{E}_{I\sim P}\left[\left(\frac{a_{\Phi_a}(\mathbf{x})_I}{p_I}\right)^2 \cdot \|h_{\Phi_h}(\mathbf{x})_I\|_2^2\right] - \left(\mathbb{E}_{I\sim P}\left[\frac{a_{\Phi_a}(\mathbf{x})_I \cdot h_{\Phi_h}(\mathbf{x})_I}{p_I}\right]\right)^2\right\} \tag{A.7}$$

$$= \arg\min_P \mathbb{E}_{I\sim P}\left[\left(\frac{a_{\Phi_a}(\mathbf{x})_I}{p_I}\right)^2 \cdot \|h_{\Phi_h}(\mathbf{x})_I\|_2^2\right] \tag{A.8}$$

$$= \arg\min_P \sum_{i=1}^{N} p_i \cdot \frac{a_{\Phi_a}(\mathbf{x})_i^2}{p_i^2} \cdot \|h_{\Phi_h}(\mathbf{x})_i\|_2^2 \tag{A.9}$$

$$= \arg\min_P \sum_{i=1}^{N} \frac{a_{\Phi_a}(\mathbf{x})_i^2}{p_i} \cdot \|h_{\Phi_h}(\mathbf{x})_i\|_2^2. \tag{A.10}$$

The minimum of the last expression is:

$$p_i^\star \propto a_{\Phi_a}(\mathbf{x})_i \cdot \|h_{\Phi_h}(\mathbf{x})_i\|_2. \tag{A.11}$$

This means that the selection of i.i.d indices according to the attention distribution of is optimal when we have no information about the norm of the features, which is set at 1.

A.2 Sampling with replacement
Gradients derivation

In this section we show the derivation of equation 4.14 by means of the Monte Carlo approximation and the multiply by one trick:

$$\frac{\partial}{\partial\phi}\frac{1}{K}\sum_{i=1}^{K} h_{\Phi_h}(\mathbf{x})_{q_i} \tag{A.12}$$

$$\approx \frac{\partial}{\partial\phi}\sum_{i=1}^{N} a_{\Phi_a}(\mathbf{x})_i \cdot h_{\Phi_h}(\mathbf{x})_i \tag{A.13}$$

$$= \sum_{i=1}^{N} \frac{\partial}{\partial\phi}[a_{\Phi_a}(\mathbf{x})_i \cdot h_{\Phi_h}(\mathbf{x})_i] \tag{A.14}$$

$$= \sum_{i=1}^{N} \frac{a_{\Phi_a}(\mathbf{x})_i}{a_{\Phi_a}(\mathbf{x})_i} \cdot \frac{\partial}{\partial\phi}[a_{\Phi_a}(\mathbf{x})_i \cdot h_{\Phi_h}(\mathbf{x})_i] \tag{A.15}$$

$$= \mathbb{E}_{I\sim a(\mathbf{x})}\left[\frac{\frac{\partial}{\partial\phi}[a_{\Phi_a}(\mathbf{x})_I \cdot h_{\Phi_h}(\mathbf{x})_I]}{a_{\Phi_a}(\mathbf{x})_I}\right] \tag{A.16}$$

$$\approx \frac{1}{K}\sum_{i=1}^{K} \frac{\frac{\partial}{\partial\phi}[a_{\Phi_a}(\mathbf{x})_{q_i} \cdot h_{\Phi_h}(\mathbf{x})_{q_i}]}{a_{\Phi_a}(\mathbf{x})_{q_i}}. \tag{A.17}$$

A.3 Sampling without replacement
Unbiased estimator

In this section is provided the proof of equation 4.17.

$$\mathop{\mathbb{E}}_{I_1,\dots,I_K}\left[\sum_{j=1}^{K-1} a_{\Phi_a}(\mathbf{x})_{I_j} h_{\Phi_h}(\mathbf{x})_{I_j} + h_{\Phi_h}(\mathbf{x})_{I_K} \sum_{t\notin\{I_1,\dots I_{K-1}\}} a_{\Phi_a}(\mathbf{x})_t\right] = \tag{A.18}$$

$$\mathop{\mathbb{E}}_{I_1,\dots,I_{K-1}}\left[\sum_{j=1}^{K-1} a_{\Phi_a}(\mathbf{x})_{I_j} h_{\Phi_h}(\mathbf{x})_{I_j} + \mathop{\mathbb{E}}_{\mathbb{I}_{\mathbb{K}}}\left[h_{\Phi_h}(\mathbf{x})_{I_K} \sum_{t\notin\{I_1,\dots I_{K-1}\}} a_{\Phi_a}(\mathbf{x})_t\right]\right] = \tag{A.19}$$

$$\mathop{\mathbb{E}}_{I_1,\dots,I_{K-1}}\left[\sum_{j=1}^{K-1} a_{\Phi_a}(\mathbf{x})_{I_j} h_{\Phi_h}(\mathbf{x})_{I_j} + \right.$$

$$\left. \sum_{i_K\notin\{I_1,\dots,I_{K-1}\}} p_K(i_K)\left(h_{\Phi_h}(\mathbf{x})_{i_K} \sum_{t\notin\{I_1,\dots I_{K-1}\}} a_{\Phi_a}(\mathbf{x})_t\right)\right] = \tag{A.20}$$

$$\mathop{\mathbb{E}}_{I_1,\dots,I_{K-1}}\left[\sum_{j=1}^{K-1} a_{\Phi_a}(\mathbf{x})_{I_j} h_{\Phi_h}(\mathbf{x})_{I_j} + \right.$$

$$\left. \sum_{i_K\notin\{I_1,\dots,I_{K-1}\}} \frac{a_{\Phi_a}(\mathbf{x})_{i_k}}{\sum_{t\notin\{I_1,\dots I_{K-1}\}} a_{\Phi_a}(\mathbf{x})_t}\left(h_{\Phi_h}(\mathbf{x})_{i_K} \sum_{t\notin\{I_1,\dots I_{K-1}\}} a_{\Phi_a}(\mathbf{x})_t\right)\right] = \tag{A.21}$$

$$\mathop{\mathbb{E}}_{I_1,\dots,I_{K-1}}\left[\sum_{j=1}^{K-1} a_{\Phi_a}(\mathbf{x})_{I_j} h_{\Phi_h}(\mathbf{x})_{I_j} + \sum_{i_K\notin\{I_1,\dots,I_{K-1}\}} a_{\Phi_a}(\mathbf{x})_{i_k} h_{\Phi_h}(\mathbf{x})_{i_K}\right] = \tag{A.22}$$

$$\mathop{\mathbb{E}}_{I_1,\dots,I_{K-1}}\left[\sum_{i=1}^{N} a_{\Phi_a}(\mathbf{x})_{I_i} h_{\Phi_h}(\mathbf{x})_{I_i}\right] = \tag{A.23}$$

$$\sum_{i=1}^{N} a_{\Phi_a}(\mathbf{x})_{I_i} h_{\Phi_h}(\mathbf{x})_{I_i} = \tag{A.24}$$

$$\mathbb{E}_{I\sim a(\mathbf{x})}[h_{\Phi_h}(\mathbf{x})_I] \tag{A.25}$$

A.4 Gumbel-max trick

Categorical distribution

A categorical distribution is used to assign probabilities to N distinct classes, and can be parametrized with *normalized probabilities* $\boldsymbol{\pi}$, *unnormalized probabilities* $\boldsymbol{\theta}$, or *unnormalized log-probabilities* (or *logits*) $log(\boldsymbol{\theta}) = \mathbf{a}/T$, where T is a temperature parameter that controls the distribution entropy (in many cases $T = 1$). The probabilities $\pi_{i;T}$ for $i \in \mathcal{D} = \{1, \ldots, N\}$ are given by

$$\pi_{i;T} = \frac{\theta_{i;T}}{\sum_{j\in\mathcal{D}} \theta_{j;T}} = \frac{e^{a_i/T}}{\sum_{j\in\mathcal{D}} e^{a_j/T}}. \tag{A.26}$$

This expression is also known as *softmax* function.

Gumble distribution

The Gumble distribution is a type I instance of the generalized extreme value distribution, developed within the extreme value theory, the statistical framework to make inferences about the probability of very rare or extreme events. A Gumble random variable is parametrized by two parameters: location $\mu \in \mathbb{R}$ and scale $\beta \in \mathbb{R}_0^+$. The probability density function (PDF) and cumulative density function (CDF) are given by

$$f(x) = \frac{1}{\beta} \cdot e^{-\frac{x-\mu}{\beta}} \cdot e^{-e^{-\frac{x-\mu}{\beta}}} \quad \text{(PDF)}, \tag{A.27}$$

$$F(x) = e^{-e^{-\frac{x-\mu}{\beta}}} \quad \text{(CDF)}. \tag{A.28}$$

When a random variable follow a Gumble distribution we write

$$G_{\mu,\beta} \sim Gumble(\mu, \beta) \tag{A.29}$$

where $G := G_{0,1}$.

The trick

The Gumble-max trick is used to draw a sample from a categorical distribution $Cat(\boldsymbol{\theta})$ parametrized by the unnormalized probabilities $\boldsymbol{\theta} \in \mathbb{R}_0^{+N}$, and works by adding i.i.d. Gumbel noise samples to the unnormalized log-probabilities and then selecting the index with the maximum value, which in turn follows a Gumbel distribution. The trick can be expressed as follows

$$I = \arg\max_{i\in\mathcal{D}} \{log\, \theta_i + G^{(i)}\} \sim Cat(\boldsymbol{\pi}). \tag{A.30}$$

where $\boldsymbol{\pi} = \frac{\boldsymbol{\theta}}{\sum_{i\in\mathcal{D}} \theta_i}$ and the arguments $G_{log\,\theta_i} := log\, \theta_i + G^{(i)}$ are shifted indepen-det Gumbels, usually called *perturbed logits*. Sampling using the Gumbel-max trick is like sampling from the categorical distribution $Cat(\boldsymbol{\pi})$. In order to demostrate the trick we need to show that $P[I = w] = \pi_w$. Intuitively we can start with considering that $I = w \Leftrightarrow G_{log\,\theta_w} > G_{log\,\theta_i}\ \forall i \in \mathcal{D}'$, with $\mathcal{D}' = \mathcal{D}\backslash w$, and we can factorize the probability that all $G_{log\,\theta_i}$ are smaller than $M := G_{log\,\theta_w}$

$$P[I = w] = \mathbb{E}_M\big[p(G_{log\,\theta_i} < M\ \forall i \in \mathcal{D}')\big] \tag{A.31}$$

$$= \mathbb{E}_M\Big[\prod_{i\in\mathcal{D}'} p(G_{log\,\theta_i} < M)\Big] \tag{A.32}$$

$$= \int_{-\infty}^{+\infty} f_w(m) \prod_{i\in\mathcal{D}'} p(G_{log\,\theta_i} < m)\ dm \tag{A.33}$$

$$= \int_{-\infty}^{+\infty} f_w(m) \prod_{i\in\mathcal{D}'} e^{-e^{log\,\theta_i - m}}\ dm \tag{A.34}$$

$$= \int_{-\infty}^{+\infty} f_w(m) \cdot e^{-\sum_{i\in\mathcal{D}'} e^{log\,\theta_i - m}}\ dm \tag{A.35}$$

$$= \int_{-\infty}^{+\infty} e^{log\,\theta_w - m - e^{log\,\theta_w - m}} \cdot e^{-\sum_{i\in\mathcal{D}'} e^{log\,\theta_i - m}}\ dm \tag{A.36}$$

$$= \int_{-\infty}^{+\infty} e^{log\,\theta_w - m - e^{log\,\theta_w - m}} \cdot e^{-\sum_{i\in\mathcal{D}'} e^{log\,\theta_i - m}}\ dm \tag{A.37}$$

$$= \int_{-\infty}^{+\infty} e^{log\,\theta_w - m} \cdot e^{-\sum_{i\in\mathcal{D}} e^{log\,\theta_i - m}}\ dm \tag{A.38}$$

$$= \int_{-\infty}^{+\infty} \theta_w \cdot e^{-m} \cdot e^{-e^{-m}\sum_{i\in\mathcal{D}} \theta_i}\ dm \tag{A.39}$$

$$= \pi_w \cdot Z \int_{-\infty}^{+\infty} e^{-m} \cdot e^{-e^{-m}.Z}\ dm \tag{A.40}$$

$$= \pi_w \cdot Z \cdot \frac{1}{Z} \tag{A.41}$$

$$= \pi_w \tag{A.42}$$

with $Z = \sum_{i\in\mathcal{D}} \theta_i$.

Sampling without replacement means we want to draw a sequence of k samples without repetition. This procedure can be implemented by removing from the sampling domain the sampled element, renormalizing the distribution and continue to draw the next element from the updated domain. For large domains this might be intractable and a good alternative is to use the Gumble-max trick on the updated domain. When we apply the Gumble-max trick repeatedly for sampling without relacement, the N perturbed logits can be reused for all the k samples meaning we can simply select the top-k perturbed logits computed in a single step.

A.5 Mapping function

In this section you will learn more about the mapping function $m_{\Omega_m}(\cdot)$ expressed in equation 4.20. Assuming that a general pair of height and width $\mathbf{p} = (h_i,\, w_i)$ is the corresponding position on the attention map for the sampled index i, the mapping for that single element is

$$(h_i,\, w_i) \mapsto (\lfloor \tilde{h}_i \rceil,\, \lfloor \tilde{w}_i \rceil) \tag{A.43}$$

where

$$\begin{aligned}
\tilde{h}_i &= h_i \cdot \frac{H_l - r}{H_s} \cdot \frac{H_h}{H_l} + \frac{r \cdot H_h}{2 \cdot H_l} + \frac{H_l - r}{2H_s} \cdot \frac{H_h}{H_l} - \frac{H_p}{2} && \text{(A.44)}\\
&= \left(h_i \cdot \frac{H_l - r}{H_s} + \frac{r}{2} + \frac{H_l - r}{2H_s} \right) \cdot \frac{H_h}{H_l} - \frac{H_p}{2} && \text{(A.45)}\\
&= \left(\left(h_i + \frac{1}{2} \right) \cdot \frac{H_l - r}{H_s} + \frac{r}{2} \right) \cdot \frac{H_h}{H_l} - \frac{H_p}{2}; && \text{(A.46)}\\
\tilde{w}_i &= w_i \cdot \frac{W_l - r}{W_s} \cdot \frac{W_h}{W_l} + \frac{r \cdot W_h}{2 \cdot W_l} + \frac{W_l - r}{2W_s} \cdot \frac{W_h}{W_l} - \frac{W_p}{2} && \text{(A.47)}\\
&= \left(w_i \cdot \frac{W_l - r}{W_s} + \frac{r}{2} + \frac{W_l - r}{2W_s} \right) \cdot \frac{W_h}{W_l} - \frac{W_p}{2} && \text{(A.48)}\\
&= \left(\left(w_i + \frac{1}{2} \right) \cdot \frac{W_l - r}{W_s} + \frac{r}{2} \right) \cdot \frac{W_h}{W_l} - \frac{W_p}{2} && \text{(A.49)}
\end{aligned}$$

When $r = 0$ the previous expressions become

$$\begin{aligned}
\tilde{h}_i &= \left(h_i + \frac{1}{2} \right) \cdot \frac{H_l}{H_s} \cdot \frac{H_h}{H_l} - \frac{H_p}{2} = \left(h_i + \frac{1}{2} \right) \cdot \frac{H_h}{H_s} - \frac{H_p}{2}; && \text{(A.50)}\\
\tilde{w}_i &= \left(w_i + \frac{1}{2} \right) \cdot \frac{W_l}{W_s} \cdot \frac{W_h}{W_l} - \frac{W_p}{2} = \left(w_i + \frac{1}{2} \right) \cdot \frac{W_h}{W_s} - \frac{W_p}{2}. && \text{(A.51)}
\end{aligned}$$

The result obtained is easy to interpret: if $r = 0$, $1/2$ is added to find the centre of the location, and the result is multiplied by the scale factor from the high-resolution image to the scorer output to find the corresponding location on the high-resolution image, and finally half the patch size is subtracted to find the coordinates of the upper left corner of the patch.

When $r > 0$, the equations A.46 and A.49 are a two-step mapping: the first step is from the points on the attention map to the corresponding locations on the low-resolution image, and the second step is from the low-resolution to the high-resolution image. Considering only the height coordinate of the mapped point, the first mapping step is given by $\left(h_i + \frac{1}{2}\right) \cdot \frac{H_l - r}{H_s} + \frac{r}{2}$, where the parameter r can be set equal to $\lfloor rf/2 \rceil$, where rf is the receptive field of the scorer newtork. Essentially, we map the positions on the attention map to a regular grid centred on the low-resolution image and taking into account the receptive field of the scorer network.

Appendix B

Differentiable Patch Selector Model

B.1 Lemma proof

From the definition of *stochastic smoothing* we have that

$$\tilde{f}(\mathbf{G};\,\mathcal{D}) = \mathbb{E}_{\mathbf{z}\sim\mathcal{D}}[f(\mathbf{G}+\mathbf{z})] = \int_{-\infty}^{+\infty} f(\mathbf{G}+\mathbf{z})\mu(\mathbf{z})d\mathbf{z} = \int_{-\infty}^{+\infty} f(\widetilde{\mathbf{G}})\mu(\widetilde{\mathbf{G}}-\mathbf{G})d\widetilde{\mathbf{G}} \tag{B.1}$$

where the last equality is obtained with the change of variable $\mathbf{G}+\mathbf{z} = \widetilde{\mathbf{G}}$ and

$$\nabla_G \tilde{f}(\mathbf{G};\,\mathcal{D}) = -\int_{-\infty}^{+\infty} f(\widetilde{\mathbf{G}})\nabla_G \mu(\widetilde{\mathbf{G}}-\mathbf{G})d\widetilde{\mathbf{G}} \tag{B.2}$$

$$\nabla_G^2 \tilde{f}(\mathbf{G};\,\mathcal{D}) = \int_{-\infty}^{+\infty} f(\widetilde{\mathbf{G}})\nabla_G^2 \mu(\widetilde{\mathbf{G}}-\mathbf{G})d\widetilde{\mathbf{G}}. \tag{B.3}$$

Moreover, if ν is twice-differentiable is straightforward to obtain $\nabla\mu = -\mu\nabla\nu$ and $\nabla^2\mu = (\nabla\nu\nabla\nu^\top - \nabla^2\nu)\mu$. Plugging them into B.2 and B.3, and taking into account that $\mathbf{z} = \widetilde{\mathbf{G}} - \mathbf{G}$, we can easily derive the first two results of the lemma, namely equations 4.41 and 4.42. The third result given by 4.43 can be obtained by direct differentiation of B.2: swapping the expectation with the gradient is possible because f is convex and μ continuous everywhere.

B.2 Mapping function

In this section is presetend a toy example aiming at clarifying any doubts. Suppose we have the following parameters

$$
\begin{aligned}
(H_h,\, W_h) &= (21,\, 24) \\
(H_s,\, W_s) &= (7,\, 8) \\
(H_p,\, W_p) &= (7,\, 7) \\
\left(\frac{H_h}{H_p},\, \frac{W_h}{W_a}\right) &= (3,\, 3) \\
(H_{\text{pad}},\, W_{\text{pad}}) &= \left(\left\lfloor\left(H_p - \frac{H_h}{H_s}\right)\frac{1}{2}\right\rceil,\, \left\lfloor\left(W_p - \frac{W_h}{W_s}\right)\frac{1}{2}\right\rceil\right) = (2,\, 2) \\
N &= H_s \cdot W_s = 56
\end{aligned}
$$

idx on $\mathbf{s}$	coords. on $\mathbf{x}_s$	patch start (zero pad)	patch start (non-zero pad)
0	(0,0)	(0,0)	(-2,-2)
3	(0,3)	(0,9)	(-2,7)
37	(4,5)	(12,15)	(10,13)

Where:

- the first column contains the index i of elements in $\mathbf{s}$
- second column is computed with $\left(\left\lfloor\frac{i}{W_s}\right\rfloor,\, i \bmod W_s\right)$
- the third column is computed with the zero-pad mapping proposed by the authors and reported in equation 4.58
- the third column is computed with the mapping used in this work reported in equation 4.59

In the first and second rows of the table, the mapping to the initial position of the patch is a negative position. Starting from these positions, the extraction function copies only those pixels into the patch that are in an actually available position of the high resolution image. The other positions indexed by negative coodinates are filled with zeros because they are outside the actual image area.

Appendix C

Adam optimizer

Adam is an adaptive learning rate optimization algorithm widely used in deep learning applications. It scales the learning rate by estimating the second-order momentum of the gradients and uses the momentum by estimating the first-order momentum of the gradients. These two features promote faster convergence towards the global minima of the cost function. The approximation is made by means of an exponential weighted average of the gradients evaluated on the mini-batches. To estimates the moments at the currents step indexed as t Adam utilizes the following estimators

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{C.1}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{C.2}$$

where the square operation is meant to be element-wise and g_t are the gradients of the cost function with respect the trainable parameters. Being the previous estimates biased towards the values 0, their initialization values at $t = 0$, an unbiased version is computed as follows

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{C.3}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \tag{C.4}$$

The updating rule implemented by Adam is

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{C.5}$$

where α is the stepsize and ϵ a small number to avoid division by zero.

Bibliography

[1] Quentin Berthet, Mathieu Blondel, Olivier Teboul, Marco Cuturi, Jean-Philippe Vert, and Francis R. Bach. Learning with differentiable perturbed optimizers. *ArXiv*, abs/2002.08676, 2020.

[2] Albino Carbognani, Dario Barghini, Daniele Gardiol, Mario di Martino, Giovanni B. Valsecchi, Paolo Trivero, Alberto Buzzoni, S. Rasetti, Danilo Selvestrel, Cristina Knapic, Elisa Londero, Sonia Zorba, Cosimo A. Volpicelli, Matteo Di Carlo, Jeremie Vaubaillon, Chiara Marmo, F. Colas, Diego Valeri, Ferruccio Zanotti, Mara Morini, Paolo Demaria, Brigitte Zanda, Sylvain Bouley, Paolo Vernazza, J. Gattacceca, J. L. Rault, Lucie Maquet, and Mirel Birlan. A case study of the may 30, 2017, italian fireball. *The European Physical Journal Plus*, 135:1–26, 2020.

[3] Florent Colas, Brigitte Zanda, Sylvain Bouley, Simon Jeanne, Adrien Malgoyre, Mirel Birlan, Cyrille Blanpain, Jérôme Gattacceca, Laurent Jorda, Julien Lecubin, Chiara Marmo, J. L. Rault, Jeremie Vaubaillon, Pierre Vernazza, Christophe Yohia, Daniele Gardiol, and A. Nedelcu plus 300 co authors. Fripon: a worldwide network to track incoming meteoroids. *Astronomy and Astrophysics*, 644, 2020.

[4] Jean-Baptiste Cordonnier, Aravindh Mahendran, Alexey Dosovitskiy, Dirk Weissenborn, Jakob Uszkoreit, and Thomas Unterthiner. Differentiable patch selection for image recognition. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2351–2360, 2021.

[5] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 2004.

[6] Yuri Galindo and Ana Carolina Lorena. Deep transfer learning for meteor detection. *Anais do XV Encontro Nacional de Inteligência Artificial e Computacional (ENIAC 2018)*, 2018.

[7] Ian Goodfellow, Aaron Courville, and Yoshua Bengio. *Deep Learning*. MIT Press, 2016.

[8] google research. Differentiable patch selector. `https://github.com/google-research/google-research/tree/master/ptopk_patch_selection`. Accessed: 2022-06-12.

[9] P. Gural. Deep learning algorithms applied to the classification of video meteor detections. *Monthly Notices of the Royal Astronomical Society*, 2019.

[10] P. Gural. Advances in the meteor image processing chain using fast algorithms, deep learning, and empirical fitting. *Planetary and Space Science*, 182:104847, 2020.

[11] Wang Hao, Wang Yizhou, Lou Yaqin, and Song Zhili. The role of activation function in cnn. *2020 2nd International Conference on Information Technology and Computer Application (ITCA)*, pages 429–432, 2020.

[12] Tamir Hazan, George Papandreou, and Daniel Tarlow. *Perturbations, Optimization, and Statistics*. MIT Press, 2017.

[13] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017.

[14] Iris A. M. Huijben, Wouter Kool, Max B. Paulus, and Ruud van Sloun. A review of the gumbel-max trick and its extensions for discrete stochasticity in machine learning. *IEEE transactions on pattern analysis and machine intelligence*, PP, 2022.

[15] Idiap Research Institute. Attention sampling models. `https://github.com/idiap/attention-sampling`. Accessed: 2022-06-12.

[16] Angelos Katharopoulos and Franccois Fleuret. Processing megapixel images with deep attention-sampling models. In *ICML*, 2019.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[18] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42:318–327, 2020.

[19] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[20] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[21] Nick Pawlowski, Suvrat Bhooshan, Nicolas Ballas, Francesco Ciompi, Ben Glocker, and Michal Drozdzal. Needles in haystacks: On classifying tiny objects in large images. *ArXiv*, abs/1908.06037, 2019.

[22] Gianluca Ranzini. *Astronomia: conoscere, riconoscere e osservare gli oggetti della volta celeste, dal sistema solare ai limiti dell'universo*. DeAgostini, 2012.

[23] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[24] Tensorflow. Custom op creation guide. `https://www.tensorflow.org/guide/create_op`. Accessed: 2022-06-27.

[25] Martin C. Towner, Martin Cupák, Jean Deshayes, Robert M. Howie, Benjamin A. D. Hartig, Jonathan P. Paxman, Eleanor K. Sansom, Hadrien A. R. Devillepoix, Trent Jansen-Sturgeon, and Philip A. Bland. Fireball streak detection with minimal cpu processing requirements for the desert fireball network data processing pipeline. *Publications of the Astronomical Society of Australia*, 37, 2020.

[26] FRIPON project. Homepage. `https://www.fripon.org/`. Accessed: 2022-04-02.

[27] NASA Science. Solar system exploration. `https://solarsystem.nasa.gov/`. Accessed: 2022-04-01.

[28] PRISMA project. Homepage. `http://www.prisma.inaf.it/`. Accessed: 2022-04-02.

[29] Wikipedia. Artificial intelligence. `https://en.wikipedia.org/wiki/Artificial_intelligence`. Accessed: 2022-04-05.