Deep Representation Learning and Prediction for Forest Wildfires

by

Pardis Zohouri Haghian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

An average of $8,000$ forest wildfires occurs each year in Canada burning an average of 2.5M ha/year as reported by the Government of Canada. Given the current rate of climate change, this number is expected to increase each year. Being able to predict how the fires spread would play a critical role in fire risk management. However, given the complexity of the natural processes that influence a fire system, most of the models used for simulating wildfires are computationally expensive and need a high variety of information about the environmental parameters to be able to give good performances. Deep learning algorithms allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined in terms of its relation to simpler concepts. We propose a deep learning predictor that uses a Deep Convolutional Auto-Encoder to learn the key structures of a forest wildfire spread from images and a Long Short Term Memory to predict the next phase of the fire. We divided the predictor problem in three phases: find a dataset of wildfires, learning the essential structure of forest fire, and predict the next image. We first present the simulated wildfires dataset and the algorithm we applied on it to make it more suitable to the model. Then we present the Deep Forest Wildfire Auto-Encoder and its implementation using the Caffe framework. Particular attention is given to the design considerations and to the best practice used to implement the model. We also present the design of the Deep Forest Wildfire Predictor, and some possible future variations of it.

Acknowledgements

First, I would like to express my sincere gratitude to my advisor Dr. Mark Crowley for his patience, motivation, and continuous support during my MASc study and related research.

I would also like to thank my friends and labmates for all the support they have shown me over these past years. A special thanks to Laura for all her guidance. I feel really fortunate to have met such wonderful people who inspired me, help me during the difficulties, and with whom I had fun.

A particular thanks to my husband, Navid, for his patience, love, and support. For the countless documents he proofread for me. No words can express how grateful I am to have you as my husband.

Last but not least, to my parents, parents-in-law, sisters, and brothers. The support you showed me during my research and thesis is the reason I could achieve this milestone.

Dedication

This thesis is dedicated to my parents.
For their endless love, support, and encouragement.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AE Auto-Encoder 5, 16, 17, 29, 32, 60

ANN Artificial Neural Network 7–11, 13, 54, 65

CAE Convolutional Auto-Encoder 5, 17, 19, 21, 22, 30, 33, 36, 39, 45–47, 51, 54, 58, 60, 61

CNN Convolutional Neural Network 10, 11, 13, 15, 17, 19–22, 30, 45–47

DFWAE Deep Forest Wildfire Auto-Encoder 5, 6, 24, 27–30, 32–36, 38–40, 44, 47, 48, 50, 54, 55, 63–65

DFWP Deep Forest Wildfire Predictor 5, 6, 18, 19, 28, 29, 35, 40, 44–46, 48, 62, 65

DNN Deconvolutional Neural Network 13, 15, 17, 21, 22, 30, 45–47

ELU Exponential linear units 54

FC Fully Connected 8–11, 13, 15, 16, 19–21, 30, 33, 34, 37, 45–47, 57, 58

LMDB Lightning Memory-Mapped Database 32

LRCN Long-term Recurrent Convolutional Networks 10, 19, 20, 22, 32, 34, 39, 45, 47

LRN Local Response Normalization 19

LSTM Long Short Term Memory 19, 20, 29, 34, 40, 42, 44–46, 58, 62–64

ReLu Rectified Linear Units 9, 19, 32, 33, 45, 48, 51, 54

SGD Stochastic Gradient Descent 38, 50

Chapter 1

Introduction

The general problem addressed in this thesis is a spatially spreading elements problem, where the goal is to study the behavior of an agent spreading in a defined environment. For the purpose of this thesis, the problem has been specialized to the spreading of a forest wildfire.

From the Government of Canada website, an average of $8,000$ wildfires occurs each year in Canada, burning an average of $2.5 Mha/year$. Given the current rate of climate change, this number is expected to increase each year. In Canada, around 55% of all fires are human-caused, while the remaining 45% are caused by lightning. However, human-caused fires usually occur in populated areas and they are usually promptly reported and extinguished, while lightning-originated ones often occur in remote locations and in clusters so they cause larger burnt areas.

There is no doubt that being able to predict how a wildfire would spread would play a critical role in fire risk management. For example, knowing that the fire would naturally die in one direction without expanding, while another area may cause more damages, would be helpful in focusing firefighting resources (e.g. prioritizing targets for air tankers and ground crews). Moreover, a wildfire predictor can be used to optimize simulators for areas of high lightning risk, allowing for taking preventative actions and designing quick risk management solutions.

In the forest wildfire problem, most of the knowledge of what elements (e.g. humidity, temperature, wind, kind of trees) and how they influence the fire are known. In fact, there are multiple studies on how wildfires spread. In particular, [9] builds a strong fire spread theory using lab experiments, real forest condition simulations, and trials. This fire behavior theory is used to increase model reliability. These improved models are used by

the USDA Forest Service [10], however, they are computationally expensive to run. There are multiple studies on applying the rules of the fire spread theory presented by [9] on models which range from trying to have more precise results to making the model faster and less expensive to run. However, in all these scenarios, a high amount of information is needed to obtain a good performance. This information may not be always available, particularly in those areas that are more remote and less populated. Thus, a different approach is needed that permits obtaining a good performance even in scenarios with less amount of information available.

Deep learning algorithms "allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined in terms of its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all of the knowledge that the computer needs" as per [13]. Thus, the solution would be to optimize fire spread models by learning its behavior directly from the data.

Given the growing amount of interest within deep learning in the last few years and the importance of this major environmental issue, its no wonder that some researchers started to combine the two together. For example, [52] uses a combination of Cellular Automation, a relatively simple modelling approach compatible with Geographical Information System (GIS) and for this reason used to reproduce the evolution of some natural phenomena such as wildfire, and Extreme Learning Machine (ELM) to predict the spreading of a wildfire. The idea presented by the paper is to use a data-driven learning method, the ELM, to create the local evaluation rules of fire spreading. This approach allows having good simulations without considering the complicated theory of traditional forest fire modelling approach and severe physical parameters. However, even if their model uses GIS data, because they are focusing on the local spread of the fire from a cell to its adjacent cells, their approach is not able to reproduce behaviors like fire spotting successfully, in which sparks or embers are carried up by the wind initiating a new fire outside a burn unit, or fire spreading across rivers.

By applying deep learning (and in particular by using convolutional layers) our predictive model would be able to obtain good results in predicting the spreading of the wildfire by using only a sequence of images representing previous timestamps of the fire. By using a convolutional approach to extract the essential structure of forest fire from the images we are going to avoid the problems encountered when analyzing how the fire locally spreads. Moreover, a deep learning model learns in a different way than a human would, seeing hidden relations and concepts. Thus, while a human analyzing a fire would need more information about it than just the images, the model would be able to give an adequate approximation. Moreover, if other features are added the performance of the model would

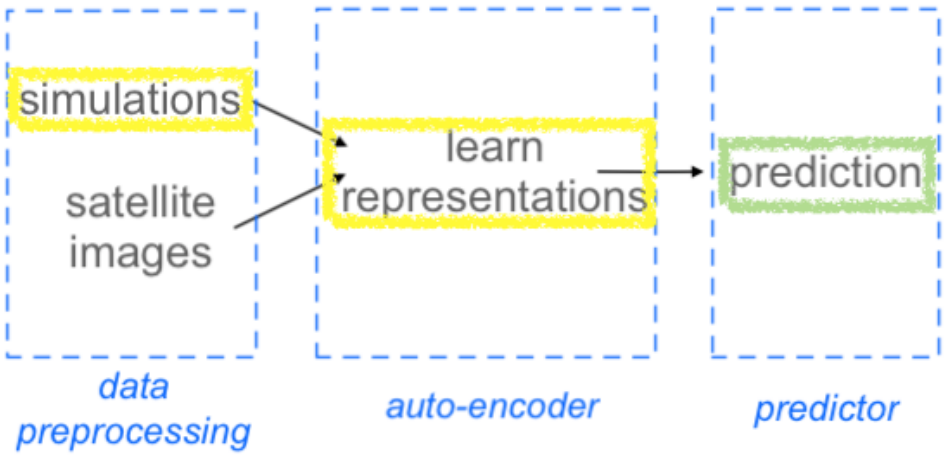


Figure 1.1: Schema of the goals of this thesis

improve.

1.1 Problem definition

Advances in machine learning and deep learning allows for novel methods to analyze big data and complex scenarios and make predictions. Today's environmental problems, such as forest wildfire are areas of high interest, however, they still lack good solutions for their complexity and for low availability of key factors such as wind speed, humidity, etc. Using deep learning permits to circumnavigate these problems and find solutions based only on the image data.

As can be observed from Figure 1.1 in this thesis we divided the prediction problem in 3 sequential tasks:

- data preprocessing - the goal is finding forest wildfires datasets to use to run our model. In particular, we divided the datasets into two categories simulated data and satellite data. For the purpose of this thesis, we focused on simulated datasets. The reason being that simulated data are less influenced by noise and distortion (in particular, geometric and radiometric distortions [5]) and, in general, permit us to better validate our approach. Ideally, the model should be able to run on any kind of dataset without needing too much preprocessing. However, in the simulated

data used for the purpose of this thesis (see Section 3.1), there are symbols and representations that are not present in satellite data. For this reason, preprocessing algorithms that correct the simulated data by making them resemble the satellite data are studied too. It is assumed that the performance of this model will be suitable for satellite images based on the specific operations used in the model. This, however, needs to be evaluated in future works where the model is applied on satellite images.

- learning representation - it represents the main focus of this thesis. The goal is to create a model that manages a two way operation: it should be able to learn how to extract from image data key features, which are going to be used by the predictor to make the actual predictions, and it should be able to reconstruct the image from the key features, which are the prediction returned by the predictor. In other words, the goal is to develop a deep autoencoder framework for extracting the representation of the essential structure of forest fire from a series of images. The advantages of using an autoencoder to do feature extraction are many, to cite few an autoencoder can use convolutional layers (Subsections 2.1.3 and 2.1.5) which usually give better results with images and videos, also using an autoencoder to learn the representation of the images allows us to apply transfer learning, training first the autoencoder part and then using the pretrained layers for the predictor part, finally, because the labels are the same of the data, autoencoders permit to do unsupervised learning.

- prediction - the goal is, given a sequence of images representing a wildfire spread, predict its development. For the purpose of this thesis we just presented the design of this model, leaving the development for future works.

In this thesis, particular attention is given to the design considerations relative to creating a deep learning model and to use the deep learning framework Caffe (a description of the framework can be found in Subsection 2.1.6).

1.1.1 Research questions

To achieve the goal of this thesis we had to respond to the following research questions:

- What kind of simulated data can we use to achieve a good performance and replicate as close as possible a satellite dataset?

- What is the dimension (in depth) of the model that gives the best performance? How do we define that?

- The model presented in this thesis needs to be designed so that can run with different kind of datasets. One of the challenges that this entails is that the dataset can have different sizes. Taking this into consideration, can we create a model that gives good performance with both big datasets and small datasets?

1.1.2 Thesis Contributions

As presented previously, the forest wildfire prediction problem has been divided into 3 tasks: finding and preprocessing the dataset, learning the representation of the images, and predicting the development of the fire. The Deep Forest Wildfire Auto-Encoder (DFWAE) model is a Auto-Encoder (AE) developed to learn the essential structure of the forest and the fire while the Deep Forest Wildfire Predictor (DFWP) is the model designed to predict the spreading of the fire. The main focus of this thesis is on the definition of the DFWAE model while focusing on the design considerations relative to creating a deep learning model and using the Caffe framework. Therefore, the main contributions of this thesis are:

- **Creation of the preprocessed dataset**: We wrote a preprocessing algorithm that would make the simulated images more similar to satellite images by replacing stylized symbols of trees and fire with the respective color.
- **Creation of the DFWAE model**: We presented a Convolutional Auto-Encoder (CAE) model which performs well also with smaller datasets.
- **Discussing techniques to design a deep learning model**: While designing a deep learning model most of the decision are usually empirical. However, in this thesis, we discuss some general guidelines on how to make these decisions. We presented the best practices we applied and the restrictions we encountered in designing and developing our deep learning model.
- **Visualization of the layers output**: Given the complexity of Deep Learning frameworks, visualizing the output of the internal layers allows for observing how the model is learning. We presented a Python algorithm that prints the output matrix of each vision layer. Moreover, this technique can be used for debugging proposes to define which layer causes vanishing gradients or a similar problem.
- **Caffe documentation**: In this thesis, we indicated some best practices and challenges of Caffe which are not well documented and we presented solutions where possible.

1.2 Outline

The remainder of the thesis is structured as follows:

In Chapter 2, some basic notions and terminologies required for this thesis, and a comparison with two existing models that inspire the DFWP, and therefore the DFWAE model.

In Chapter 3, the dataset used to run the model, the preprocessing operations applied to it, and the design considerations relative to it are presented.

In Chapter 4, the DFWAE architecture is defined. In particular, each layer of the model and the solver parameters, defining how the hyperparameters update the model, are described. The simulation environment parameters are also reported.

In Chapter 5, the results of the DFWAE model, some methodological decisions, taken during its implementation, and some challenges addressed are discussed. Additionally, a first unsuccessful version of the DFWAE is analyzed, with an explanation of the improvements applied to create the final DFWAE.

In Chapter 6, the conclusions of this thesis are drawn and the DFWP design is presented as future work, with some ideas on alternative versions and applications of the model.

Chapter 2

Background

2.1 Notions

2.1.1 Artificial Neural Network

The name Artificial Neural Network (ANN) derives by the fact that this framework of machine learning algorithms is loosely inspired by the structure and behavior of a biological brain. In particular, the system is composed of interconnected nodes, also called artificial neurons. The connections between the nodes, called edges, transmit a signal from one node to another, similar to a synapse in a biological brain. When an artificial neuron receives a signal, it processes it and sends the processed signal to other nodes.

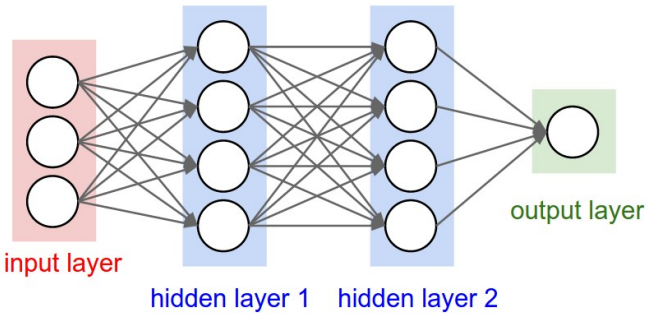


Figure 2.1: A 3-layer Neural Network: 2 hidden layers of 4 neurons and one output layer with a single neuron

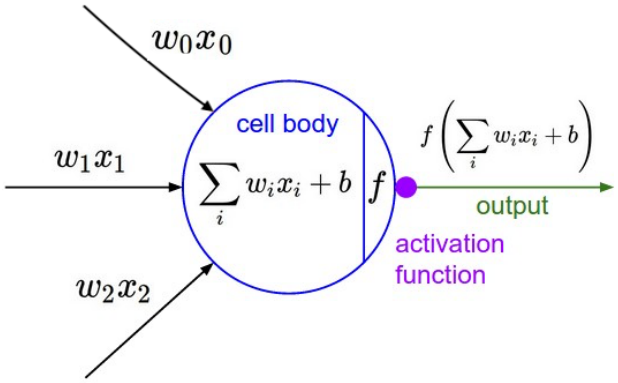
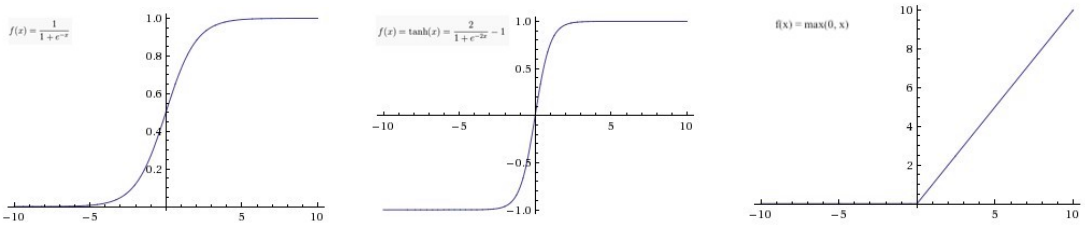


Figure 2.2: Representation of a neuron

An ANN is usually composed of an input layer, one or multiple hidden layers, and an output layer. The organization of a ANN is layer-wise, thus each layer (except the input layer) takes the output of the previous layer as input data. Moreover, the neurons in each layer do not share any connection. In Figure 2.1 a 3-layer ANN is shown. The network is composed of 2 hidden layers and they both are Fully Connected (FC) layers of 4 neurons and an output layer with a single neuron. A FC layer is the most common layer type: each neuron between two adjacent layers are fully pairwise connected.

In Figure 2.2 is shown a single neuron and how it works.

- a represents the outputs of the connected neurons of the previous layer.
- w represents the weights assigned to each connection. The weight indicates how much the input from that defined connection influences the new neuron. The initialization of the weights is defined when developing the model, and their value is updated during the training phase.
- b represents the bias. This value is optional but it is usually added as a threshold to shift the activation function.
- $\sum$ is the propagation function $\sum_i w_i x_i + b$. The output from the connected neurons in the previous adjacent layer are multiplied by the respective weight and then they are all summed together. If a bias is defined it's then added.



(a) Sigmoid Function (b) Tanh Function (c) ReLU Function

Figure 2.3: Activation Functions most commonly used in Neural Networks

- f represents the activation function. It can be a linear or non-linear function, however, usually, a non-linear activation function is used. The function improves the capacity of the model to generalize or adapt with a variety of data and to distinguish between the output.

Activation Functions

There are multiple activation functions but the most commonly used are Sigmoid or Logistic function, Tanh or Hyperbolic Tangent function, and Rectified Linear Units (ReLu) function (Figure 2.3).

- **Sigmoid function** - this function returns a value between 0 and 1. For this reason, it is particularly used for models that predict a probability as an output.
- **Tanh function** - this function is a sigmoidal function but the output values are between -1 and 1. This provides the advantage that the negative input values are mapped more strongly negative and the zero ones are mapped near zero.
- **ReLu function** - this function is half rectified. The function returns zero if the input value is negative or the value itself.

In a ANN the activation function is represented by the activation layer. However, it always follows defined layers (e.g. FC layers and convolutional layers), thus they are sometimes omitted in the representation of the layer. For example, in Figure 2.1 the FC layers are shown but not the activation layers that follow each layer. This omission is due to the fact that even if the two type of layers are separate, a FC layer will always be followed by an activation layer, making this one de facto part of the first one.

How does an ANN learn

An ANN learns how to perform tasks just by analyzing examples, generally without having any task-specific rules. During the training of a model, a loss function drives the learning by specifying the goal of learning by comparing the inaccuracy of the predicted result with the actual result in order to evaluate the correctness of the parameter settings of the model. In other words, the predicted results are compared to the actual result using a loss function, then the error is backpropagated along the model, and the parameters of the model (e.g. the weights) are updated in order to minimize the error.

2.1.2 Feed-forward and Feed-back Neural Networks

The different types of ANN are usually classified in feed-forward and feed-back networks. In a feed-forward network, the signals travel only forward. Each layer receives an input data, performs its computations and returns it as an output data for the following layer. In other words, the top blob will never be used as the bottom blob of the same layer. Each batch of data goes from the data layer through all the other layers sequentially, from the bottom to the top, until it reaches the last layer and determinates the output. Some example of feed-forward networks are ANN with only FC layers and a Convolutional Neural Network (CNN).

In a feed-back (or recurrent or interactive) network, loops are used to permit the signal to also travel back. In a recurrent network, the data computed previously is fed back into the network. This creates a kind of memory, which is the reason this kind of networks are used for sequential tasks such as time series prediction and sequence classification. A feed-back network is a non-linear dynamic system, which changes continuously until it reaches a state of equilibrium. An example of feed-back network is the Long-term Recurrent Convolutional Networks (LRCN) network described in Subsection 2.2.1.

2.1.3 Convolutional Neural Network

A CNN is a feed-forward network. As in the ANN, each neuron receives some input from the connected neurons of the previous adjacent layer, performs a dot product between them and each respective weight and if a bias is defined, it performs the shift, finally, an activation function is applied on the result of each neuron. Moreover, as an ANN, a CNN uses a loss function to compare the predicted results with the actual result to train the

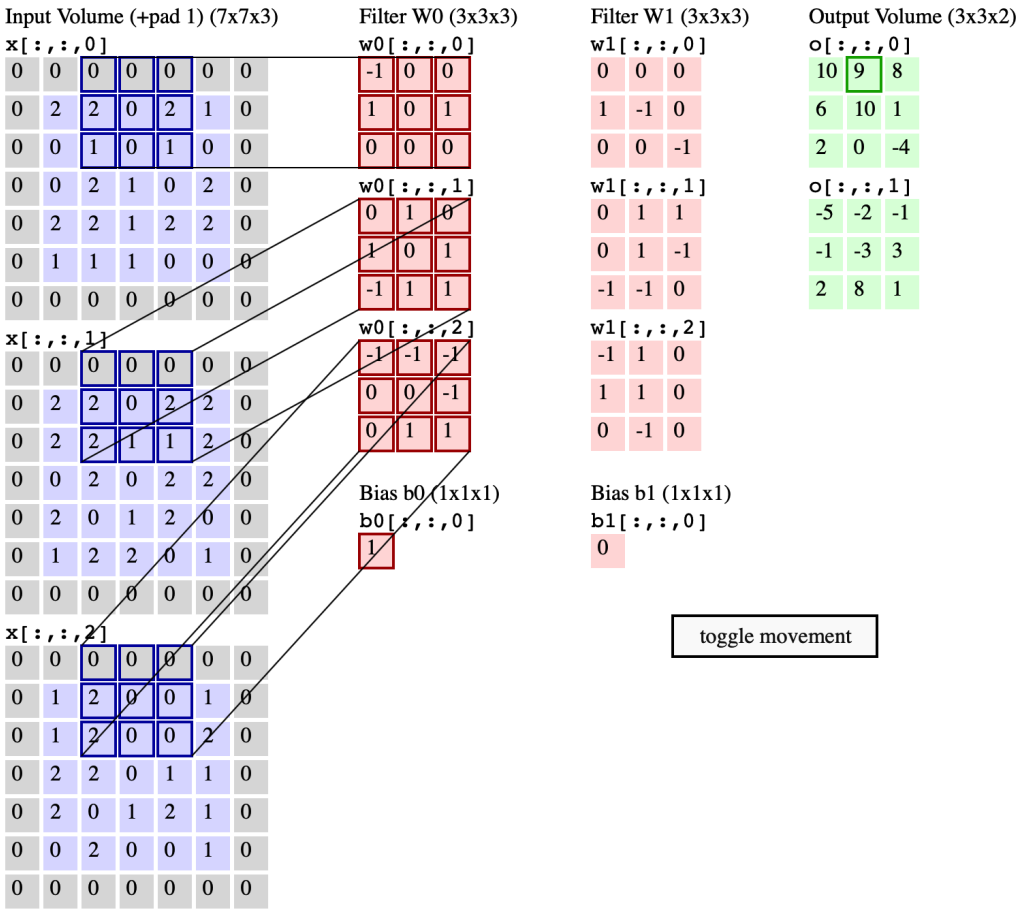


Figure 2.4: Example of convolutional operation

model. However, instead of having a distinct weight for each pairwise connection, in a CNN the same weights are shared for each submatrix of pairwise connections. The main idea is that an ANN with only FC layers has a really high number of weights to calculate whenever the model has more depth (more hidden layers are added). A CNN uses filters, also called kernels, which are 3-dimensional matrices of weights of given width and height, and its depth is equal to the depth of the previous adjacent layer output. As shown in Figure 2.4, the filter is passed (convolved) across the width and height of the input volume and a dot product is applied between the entries of the filter and the input at any position.

In other words, the width and height of the output matrix, w_o and h_o, are given by the width and height of the input matrix, w_i and h_i, the kernel size, F the padding applied to

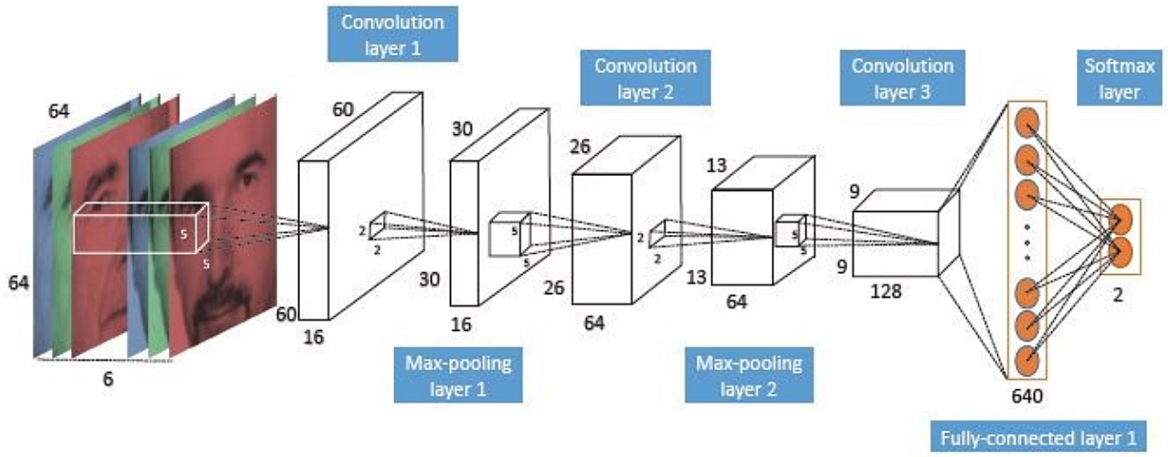


Figure 2.5: A Convolutional Neural Network

the input matrix before applying the convolution, P, and the stride, s, of the shift of the filter on the input matrix:

$$\begin{aligned} w_o &= \frac{w_i - F_w + 2P}{s_w} + 1 \\ h_o &= \frac{h_i - F_h + 2P}{s_h} + 1 \end{aligned} \tag{2.1}$$

As previously stated, when each filter passes across the input matrix a dot product is applied. Thus, the depth of the result of each filter is 1, hence the depth of the output matrix after the convolution is equal to the number of filters used in the layer. In a convolutional layer, each filter has 3 dimensions, however, because the convolution happens only along the height and width dimension and not on the depth, the convolution is considered a 2D convolution. For this reason, in the literature, the depth of the kernel is omitted in the annotations. Moreover, it is common to have the width and the height to have the same value and fall within the interval of 3 to 11. However, as a rule, the value has to be odd.

Types of layers of a Convolutional Neural Network

Similar to the general ANN described in Subsection 2.1.1, a CNN has a layer wise organization. There are three main types of layers that compose a CNN: convolutional layer, pooling layer, and FC layer.

- **convolutional layer** - it performs the convolutional operation described above. The layer is defined by the following hyperparameters: number of filters, size of the filters, stride for which the filter is moved across the input matrix, and padding added on the input matrix before applying convolution. Each convolutional layer is followed by an activation layer.
- **pooling layer** - it performs a dimensionality reduction on the width and height of the input matrix. This is achieved by moving a filter of defined dimension across the input matrix, then according to the type of pooling defined, a single value is selected as representative of that patch. The most common type is MAX-pooling. As suggested from the name, it replaces the evaluated submatrix with the maximum value in it. The layer is defined by the following hyperparameters: dimensions of the filter and stride for which the filter is moved across the input matrix.
- **FC layer** - as described in Subsection 2.1.1 each neuron in this layer is fully pairwise connected with the neurons in the previous adjacent layer. The layer hyperparameter is the number of nodes.

2.1.4 Deconvolutional Neural Network

A simple Deconvolutional Neural Network (DNN) is another example of a feed-forward network. A deconvolution layer applies a transposed convolution to the input matrix. An example of deconvolution can be observed in Figure 2.6. In the example the input matrix has dimension 2×2, the filter is a matrix of ones of dimensions 3×3, no padding is applied, and the stride is 2. First, it applies an element-wise multiplication of each element in the input matrix with the filter. This operation returns a number of matrices equal to the number of elements in the input matrix, with dimensions equal to that of the filter. The obtained matrices are then combined accordingly with the stride. In the example, for each cell in the Result matrix, the elements in red are the number summed to obtain the final value.

The width and height of the output matrix, w_o and h_o, are given by the width and height of the input matrix, w_i and h_i, the kernel size, F the padding applied on the input

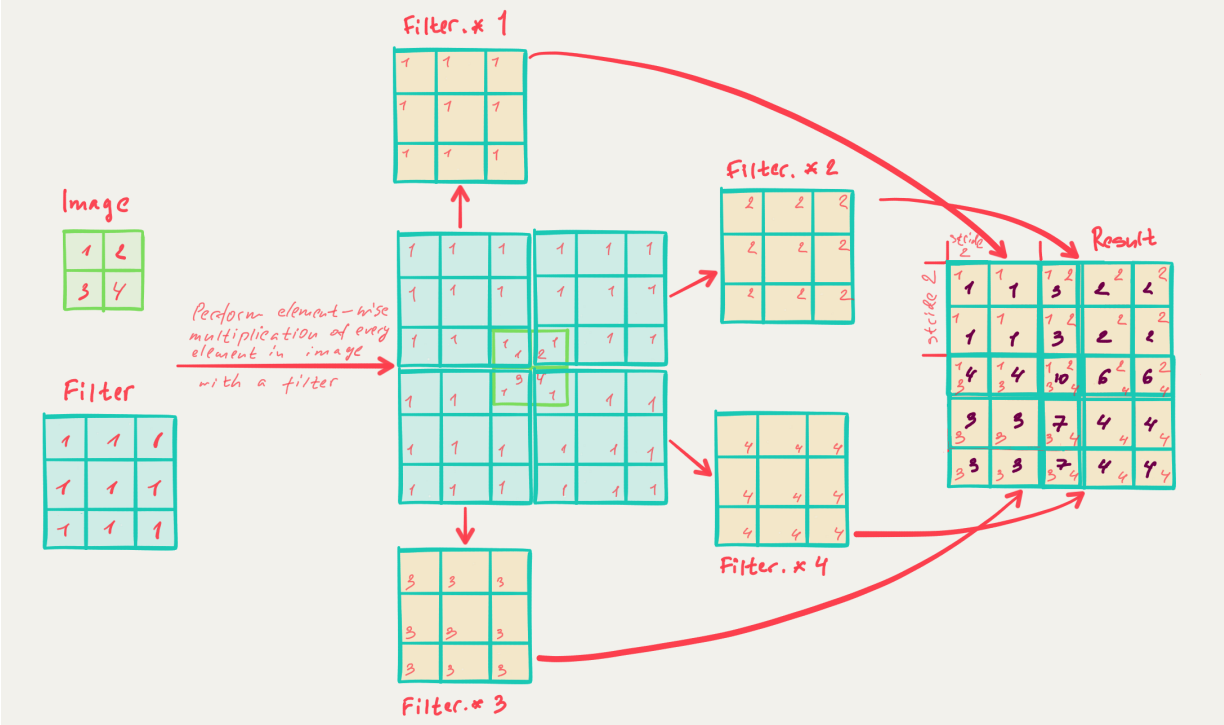


Figure 2.6: Example of a deconvolutional operation with kernel size 3×3 and stride 2

matrix before applying the convolution, P, and the stride, s, of the shift of the filter on the input matrix:

$$\begin{aligned} w_o &= s_w(w_i - 1) + F_w - 2P \\ h_o &= s_h(h_i - 1) + F_h - 2P \end{aligned} \tag{2.2}$$

As for the convolution, when each filter passes across the input matrix a dot product is applied. Thus, the depth of the result of each filter is 1, hence the depth of the output matrix after the deconvolution is equal to the number of filters used in the layer. Similarly to the convolution layer, in a deconvolutional layer, even if each filter has 3 dimensions because the deconvolution happens only along the height and width dimension and not on the depth, the deconvolution is considered a 2D deconvolution. For this reason, in the literature, the depth of the kernel is omitted in the annotations. It is common to have the width and the height to have the same value and fall within the interval of 3 to 11. However, as a rule, the value has to be odd.

Types of layers of a Deconvolutional Neural Network

As previously stated, a DNN applies a transpose operation with respect to CNN. Similar to the CNN, it also has a layer-wise organization. There are three main types of layers that compose a CNN: deconvolutional layer, unpooling layer, and FC layer.

- **deconvolutional layer** - it performs the deconvolutional operation described above. In Figure 2.8 the relation between the operation of convolution and of deconvolution can be observed. The layer is defined by the following hyperparameters: number of filters, size of the filters, stride for which the filter is moved across the input matrix, and padding added on the input matrix before applying deconvolution. Each deconvolutional layer is followed by an activation layer.

- **unpooling layer** - it performs a dimensionality enlargement on the width and height of the input matrix. Thus, it performs an inverse operation relative to the pooling operation as shown in Figure 2.8. This is achieved by increasing every cell of the dimension of the kernel, this submatrix is going to have all values to 0 except one cell which has the same value of the input cell. Switch variables are used to define the position assigned to the input value, in particular when an unpooling layer is designed as inverse to a pooling layer, the switch variables are passed from the pooling layer to the unpooling one to indicate which are the original cells with the maximum value.

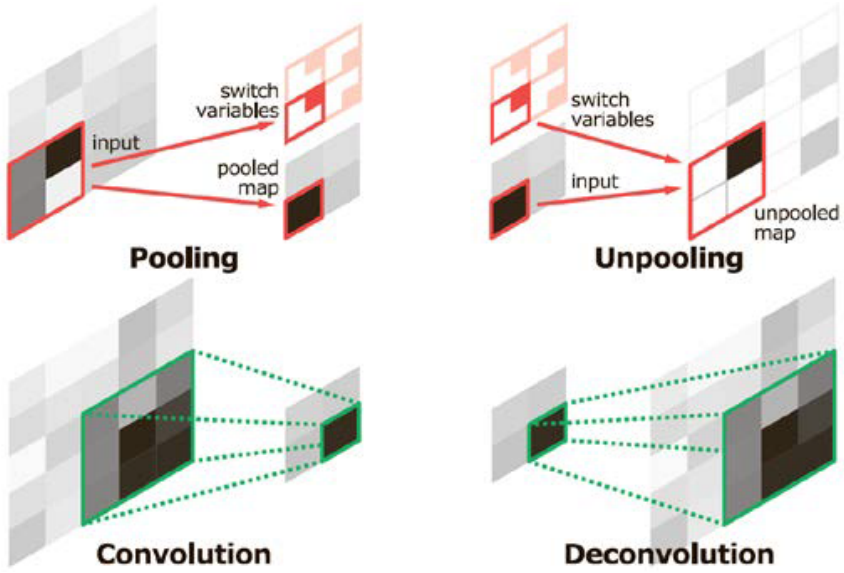


Figure 2.7: Representation of pooling and unpooling operations and convolution and deconvolution

If switch variables are not used, the value is assigned to a default position of the submatrix. The layer is defined by the following hyperparameters: dimensions of the filter, stride for which the filter is moved across the input matrix.

- **FC layer** - as described in Subsection 2.1.1 each neuron in this layer is fully pairwise connected with the neurons in the previous adjacent layer. The layer hyperparameter is the number of nodes.

2.1.5 Auto-Encoder

An AE is a feed-forward network composed of an encoder network and a decoder network. The encoder portion transforms the input data into a typical lower-dimensional representation and the decoder portion reconstructs the original image from its encoded representation. The AE is built with the minimization of a loss function, called reconstruction error, which represents the difference between the original data and the reconstructed data. In particular, the required gradients needed are obtained by backpropagation of the error through first the decoder network and then the encoder network [16][42]. In other words, if the encoder function is represented by $h = f(x)$ and the decoder by $r = g(h)$,

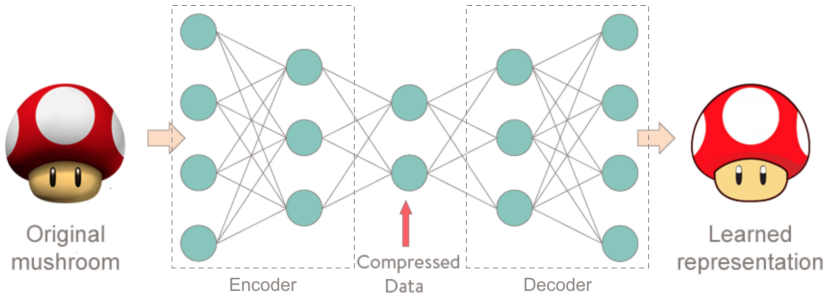


Figure 2.8: Example of autoencoder

the AE learning process is described by the function $min(L(x, g(f(x)))$ with L being a loss function.

An AE is generally considered under the category of unsupervised learning because it doesn't use labeled data. However, because the data is also a target variable for the model, in some papers it is also referred to as supervised or semi-supervised. AEs are applied to multiple tasks, such as dimensionality reduction and information retrieval, image colorization, generating higher resolution images.

Convolutional Auto-Encoder

An CAE is an auto-encoder in which the encoder part is a CNN. The advantages of using a CNN is not only the fact that it permits the use of bigger sized datasets, but also a decreased redundancy problem[27]. The decoder part can be another CNN [1] or a DNN [32][29].

2.1.6 Caffe

Caffe is a deep learning framework developed by Berkeley AI Research (BAIR) [22].

There are many advantages of using Caffe to deploy our model, the following are some of the major advantages:

- A model zoo with reference CNN implementations. A database of Caffe models for different tasks with all kinds of architectures and data created from researches during the years.

- Models and optimization are expressed by plaintext schemas. Thus, when creating a new model or working with an existing one, there is minimum requirement of writing code.
- It has a big community with multiple sources of feedbacks and opensource models. Many common and new layers are available.
- Caffe supports also GPU training and switching from CPU to GPU just requires changing a single line in the solver.
- Presents a pretty good Matlab and Python interface. This aspect, in particular, had been shown to be really useful to create a code for the visualization of the layers 5.3.4.

The Caffe layers used in the DFWP architecture, are described in detail in Chapter 4.

Blob, layer, and net From [35]: "Caffe defines a net layer-by-layer in its own model schema. The network defines the entire model bottom-to-top from input data to loss". The layers of the network are the fundamental unit of computation. For each type of layer three key operations are defined:

- setup - this operation is done once during the initialization of the model and consists of initializing the layer and its connections;
- forward - this operation computes the given input read from the bottom blob and returns the result as top blob;
- backward - this is a backward induction operation and computes the given gradient with regards to the top output and sends the result to the bottom. Moreover, if the layer has parameters, they are processed with regard to the gradient and the result is stored internally.

In Caffe, a blob is a wrapper over the actual data being passed, stored, and manipulated across the layers and the entire net. The dimension of each blob is: $batch\_size \times numer\_of\_channels \times height \times width$.

Iteration and epoch In Caffe, an iteration represents a forward-backward pass over a single batch of training data. An epoch represents the forward-backward pass over all the training data. Therefore, the relation between iteration and epoch can be seen as:

$$epoch = \left\lceil \frac{tot\_img}{batch\_size} \right\rceil \tag{2.3}$$

Kernel and filter In a convolutional layer and in a deconvolutional layer the term kernel and filter can be used interchangeably. However, in the literature its size is mostly referred to as kernel size (most probably because the Caffe parameter is called kernel_size) and the numbers in the layer is mostly referred to as filter numbers. Thus, in this thesis, the same precautions are taken when using these terms.

2.2 Existing models

In the formulation of the DFWP, a number of models and papers have been studied, however, two models have mainly influenced the design of our model: the LRCN presented in [7], more specifically, the activity recognition model and Model 4 of the CAE presented in [42]. In particular, we merged LRCN to our model to take advantage of the spatial expressivity of CNN and the temporal representation capability of the Long Short Term Memory (LSTM), and we designed the decoder part consistently with the results shown from [42]. Both models have been implemented using the Caffe framework.

2.2.1 LRCN

The activity recognition model presented in [7], shown in Figure 2.9 takes a sequence of frames as the input and classifies the action acted in the video. In other words, it takes a sequence of frames, it uses CNN to extract the features of each frame and LSTM to analyze all of them sequentially, returning a single label representing the prediction.

In particular, the LRCN uses a Python layer as the input data layer, which returns three blobs: data, label and clip markers. The CNN is then composed of 5 convolutional layers interspersed by 3 pooling layers and a final FC layer. Each convolutional and FC layer is followed by a ReLu layer and each pooling layer is followed by a Local Response Normalization (LRN) layer. In the model, there are two dropout layers: one after the CNN and one after the LSTM. After the CNN dropout layer, the data is reshaped before being

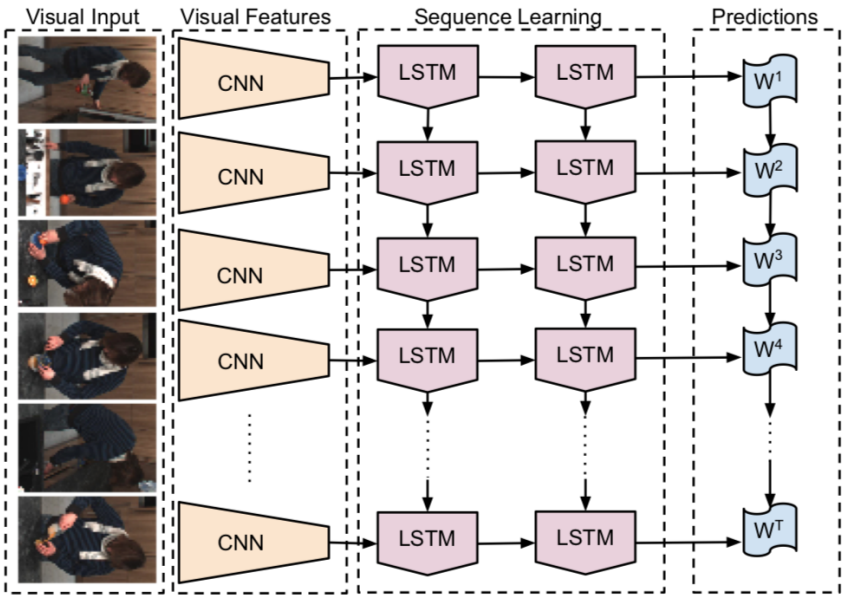


Figure 2.9: Long-term Recurrent Convolutional Network model

used by LSTM, so are the label and the clip markers blobs obtained by the Python layer. The LSTM layer takes the three reshaped blobs to predict the correct classification. After the LSTM dropout layer there is a final FC layer. Furthermore, a soft-max-with-loss layer and an accuracy layer are used to build the model.

The LRCN is a well-suited architecture because it combines both the spatial expressivity of a CNN with the temporal representation capability of the LSTM. CNN has considerably outperformed traditional machine learning approaches in a wide range of computer vision and pattern recognition tasks [47]. Given LSTM property of selectively remembering patterns for long durations of time, it has considerably outperformed conventional feed-forward neural networks and RNN in sequence prediction problems. While [7] have initially applied this architecture for action identification from video sequences, this architecture is also gone on to become the gold standard for many temporal imaging domains including gesture recognition on a smartphone [25], detecting eagle species both appearance and bird motion cues [48], to cite few. This, therefore, makes the LRCN a solid foundation on which to base our architecture.

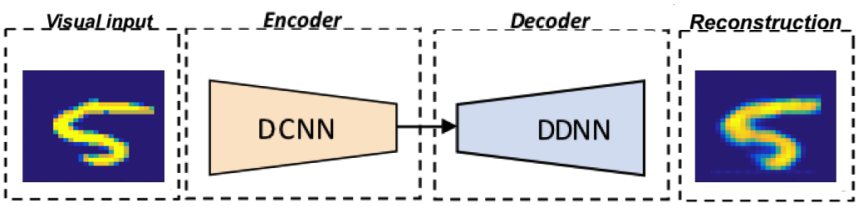


Figure 2.10: Deep Convolutional Auto-Encoder model

2.2.2 DCAE

I n [42] five CAE are presented. Model1 architecture is present in all the other models and can be described as the following:

- an HDF5 data layer to read MNIST images;
- an CNN as encoder composed by 2 convolutional layers and a FC layer;
- a FC layer that connects the CNN and the DNN;
- a DNN as decoder composed by 2 deconvolutional layers and 1 FC layer;
- a deconvolutional layer that acts as a reconstruction layer;
- two loss layers, Sigmoid cross entropy loss and Euclidean loss;
- a Sigmoid function is applied on the blobs after each convolutional, each deconvolutional, and the two inner products in the CNN and DNN;
- the encoder and decoder architectures are symmetrical in terms of the feature maps size and the number of neurons in all the hidden layers.

In Model2 two pooling layers are added in the CNN, one after each convolutional layer. This model is the only one presented in the paper in which the encoder and decoder portion don't have a symmetrical architecture. Furthermore, in Model3 two unpooling layers with switch variables are added in the DNN, one after each deconvolutional layer. The switch variables are sent from the pooling layer to the corresponding unpooling layer and represent where the operation of pooling had been done. The difference between Model4 and Model3 lies within the utilization of switch variables, where Model4 does not have switch variables. Finally, Model5 has the same architecture of Model4 with the exception of using hyperbolic tangent activation function instead of a sigmoid.

The result from the comparison of the models presented in the paper shows that Model3 and Model4 provide the best reconstruction quality of the images. In Figure 2.10, a high level representation of the CAE is shown. The example used in the figure is from the execution of Model 4.

It is important to observe that there are wide varieties of Auto-Encoders, e.g. Variational Auto-Encoders and Denoising Auto-Encoders, and even CAE differ between having the decoder composed by a CNN instead of a DNN. However, because we were inspired by the LRCN model we decided to apply a simple CAE. Moreover, as presented in Section 5.2, by using this architecture we obtained a good performance. For the decoder portion, in the literature using a DNN instead of CNN has proven to give a better performance.

Chapter 3

Dataset

Our goal is to build a machine learning framework that can be applied to any kind of dataset representing a forest wildfire spreading. In particular, the initial idea was to use satellite images. There are large varieties of satellite datasets, however, to run the model successfully we would need high-resolution data, which are proprietary and difficult to obtain. Moreover, to run the model the dataset should represent at least a couple of dozen different fires with the images representing close timestamps, ideally not more than a couple of days. For this reason, the decision was taken to first build the model on simulation data. Simulation data are also less influenced by noise and distortion, making them a better choice to validate our approach. Given the restriction in time, creating a simulator that would represent fire expansion was not achievable. Furthermore, other simulations considered used older versions of Windows or did not return images as an output. The dataset used to run the model for this work was obtained from a simulator called Fire Assessment, which uses probability to choose if the fire has spread to an adjacent tree or not.

3.1 Forest simulation dataset

The dataset used to train the model is obtained by running the following simulator http://www.shodor.org/interactivate/activities/FireAssessment/, designed by Shodor Education Foundation, Incorporated. This particular simulator was chosen due to its probabilistic behavior in simulating fire images. Additionally, its ability to generate step-by-step images of the fires progression allows for easy integration with Deep Learning models. The Fire Assessment learner is a simple model in which any tree close to the fire

has the same given probability of catching fire. To obtain the dataset, the program was run with a probability of 0.5 over a 25 trees-over-25 trees grid. The dataset has a dimension of 27 fire simulations of variable dimensions, ranging from 10 frames to 78 frames, with a total of 1165 images. In this work, the images representing the different steps of the wildfire spreading are also called frames as they can be seen as screenshots of the fire expansion. The dataset was saved with the following structure: each simulation was saved in a separate folder and each frame was saved in a PNG format.

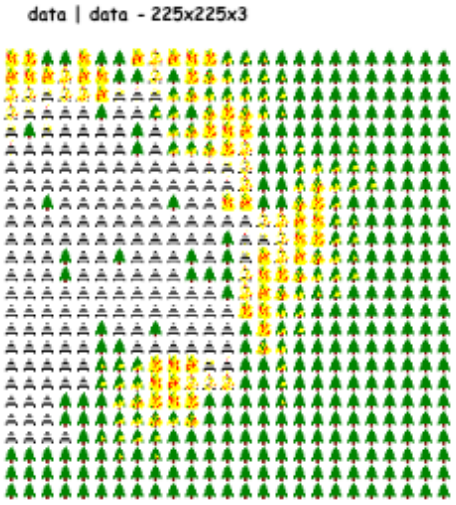
3.1.1 Preprocessing

The DFWAE model was initially run with both the original RGB version of the dataset and a grayscale version. However, the model would return incorrect results. In particular, the model would focus on the shape of the unburnt tree. Thus, as shown in Figure 3.1, it would consider the fire as noise and reconstruct the forest with all the trees unburnt. After removing the green trees the focus of the model was to rebuild the burnt trees and removed them, the focus was on the shape of the trees on fire. To circumnavigate the reconstruction problem the data was preprocessed before using it for the model. The algorithm used is written in Python and uses NumPy and OpenCV libraries to read, modify and save the images. In this new dataset, each of the cells of the 25×25 grid was replaced by a color corresponding to the status of the cell. In particular, as shown in Figure 3.2 the fire is represented by different tonality between yellow and red:

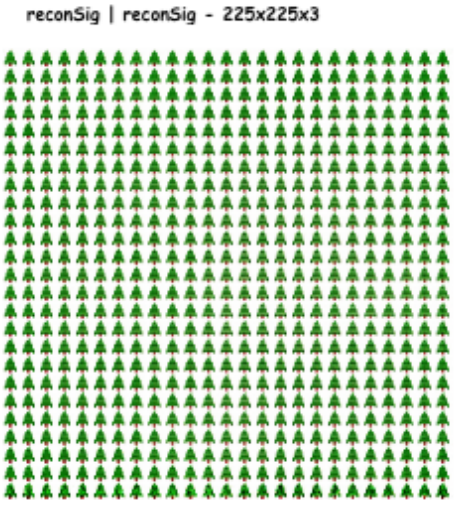
- a yellow cell represents a tree that has just 10/81 pixels on fire;
- a cell in a light orange color represents a ratio of 25/81 pixels on fire;
- a cell in a dark orange color represents a ratio of 55/81 pixels on fire;
- a red cell represents a tree with over 56/81 pixels on fire.

Moreover, the burnt trees are represented by black cells and the white portions are the healthy trees still not affected by the fire.

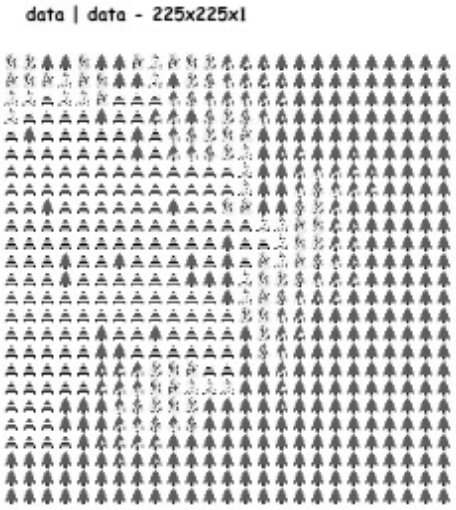
We conjecture that this preprocessing phase would not be necessary when applying the algorithm to satellite data. The problem encountered by using symbols is not present in the satellite images and overall the preprocessed simulation images are more similar to the satellite images. However, in the unlikely scenario where a similar problem is encountered on the satellite images, in which the model focuses on a specific detail of the image and not on the fire, this patching technique can be tailored as a solution.



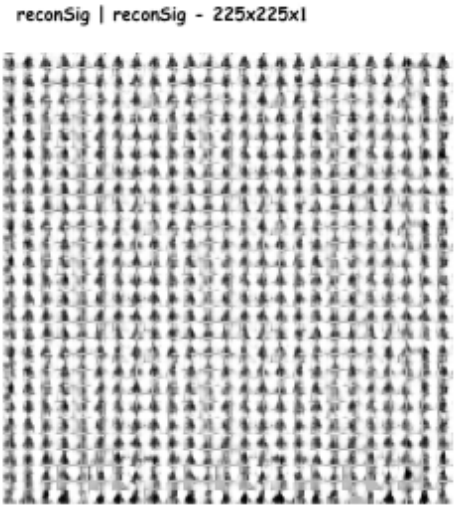
(a) Input image RGB



(b) Reconstructed image RGB

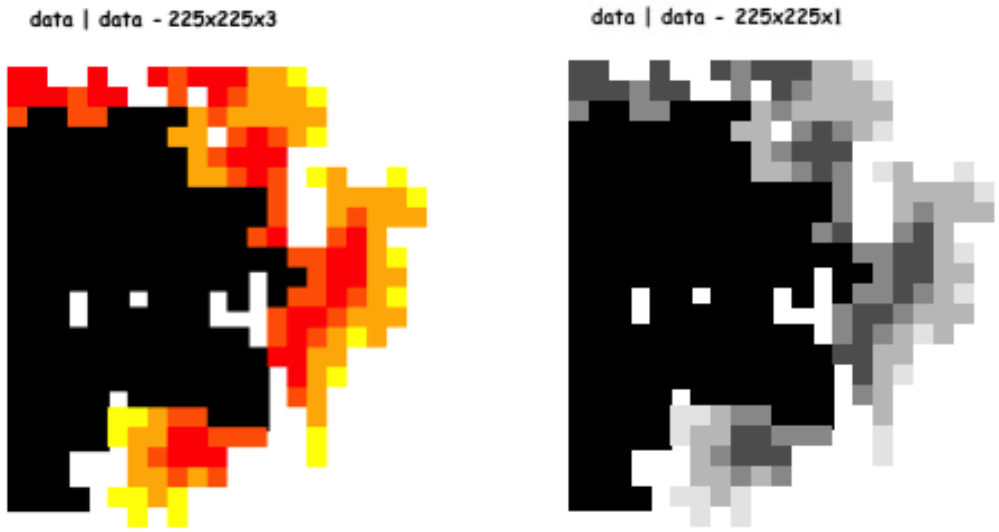


(c) Input image grayscale



(d) Reconstructed image grayscale

Figure 3.1: First run in RGB and in grayscale of the model without preprocessing the images



(a) Test image RGB after preprocessing (b) Test image grayscale after preprocessing

Figure 3.2: Preprocessed dataset used to run the model

3.2 Data augmentation

As discussed at the beginning of this chapter, the idea of this thesis was to create a model usable with satellite data. Moreover, as discussed in the introduction, one of the main reasons this project uses deep learning is its ability to obtain hidden information from the images, such as weather conditions, moisture, and typologies of vegetations, which are not always available during prediction but may be extractable from the satellite images.
Due to having a limited dataset, data augmentation was used to increase the dimension of the dataset. However, if some data-augmentation techniques, such as rotation, scale, or methods causing change to the color of the image are applied on the dataset the resulting images will have distortion, which causes hidden information such as wind, humidity or other factors to become false data. Nevertheless, there are other data-augmentation techniques that would not negatively affect the model. For example, if a dataset with bigger images where available, a solution would have been to crop the images to smaller ones, paying attention to maintain the same scale for all the input data. In this scenario, translation can be applied as well, with the images overlapping and the fire location in the image moving. A data augmentation technique that creates a distortion of the data

but may be effective in this model scenario is the adding of Gaussian noise. However, the decision of using this technique depends on the resolution of the satellite images. From Subsection 3.1.1, the model shows to be strong against noise but the satellite images may contain key information, which the addition of noise may hide.

3.3 Train and test set

As discussed in Section 3.1, the forest simulation dataset has been used to build the DFWAE. However, to do so the dataset has to be divided into train, validation, and test sets. The decision of the partition has been made by considering two key aspect discussed in the following subsections. The conclusion is that two datasets are used for running the model (training and test set), with no distinction between validation and test set and 5-fold cross validation is applied to estimate the performance of the model. Therefore, the model is trained 5 different times using a different fifth of the dataset as the test set and the remaining $\frac{4}{5}$ as the training set every time.

3.3.1 Validation and test set

The DFWAE model divides the dataset into 2 distinct sets, each used for a specific phase:

- **training set** - in the training phase a train set is used to fit the parameters, for example, the weights of the model are obtained so that the loss function is minimized. During this phase, the model is trained on the dataset using a supervised learning method;
- **test set** - in the validation and test phases a set distinct from the training set is used to estimate how well the model has been trained. The two phases are usually quite distinct. In particular, the validation phase usually happens during training and it tunes the hyperparameters of the model (for example learning rate). In contraposition, the testing phase happens only once the model is completely trained and it reports the estimation of the model without changing it. However, in Caffe the hyperparameters are tuned in the training phase and not during validation. Thus, in Caffe, this phase does the same operation of test with the only exception being that it is applied during training. For this reason, in addition to the small dimension of the fire simulation dataset, the test phase and validation phase are merged and the model uses just a single test set.

3.3.2 5-fold cross validation

As mention before, the dataset used to run the DFWAE model is composed of 1165 images. Therefore, relative to other datasets usually used for deep learning such as MNIST or CIFAR-10 (both having 60K images for a train set and 10K for test set) or ImageNet (1,500K elements) to cite a few, the dimension of the simulated fire dataset is undeniably smaller. For this reason, k-fold cross validation is used. This method permits to have an adequate training set, meanwhile leaving a sufficient test set. The dataset is divided into k subsets, and at each run one of the subsets is kept apart for testing and the rest are used for training the model. This operation is done for k times so that each subset is used as a test set once, and then the loss is averaged. Generally, even if there is no recommendation on how to pick k, it is chosen as 5 or 10. However, applying 10-fold cross validation on the simulation dataset would have two disadvantages: the test sets would be consistently small and running the model 10 times, with the available machines, would take close to a month. Thus, in the DFWAE model, Section 4.1, a 5-fold cross validation is used. Therefore, having 1165 images which result in 932 frames in the train set and 233 frames in the test set. Conversely, in the design of the DFWP model, Subsection 4.2, instead of considering the single frame each sequence is considered as an entity. Thus, having 130 sequences (Table 4.4) the train set is composed of 104 sequences, while the test set is composed of 26. Considering that each sequence is composed of 10 frames, the number of images in each set are: 1,300 totals, 1040 test set, and 260 train set.

Chapter 4

Model Architecture

Given a sequence of frames representing forest wildfire spreading, the DFWP returns the frame representing the next step of the evolution of the fire. As shown in Figure 4.1, the model consist of three parts:

- **Deep Convolutional Encoder** - this represents the feature extractor of the model. The operation is applied to each frame of the sequence individually;
- **LSTM** - this represents the prediction part of the model. It uses the features, obtained by the encoder, relative to all the frames of the sequence to predict the features of the new frame.
- **Deep Convolutional Decoder** - this represents the reconstruction part of the model, from the features to the image. It reconstructs the frame represented by the features returned by the LSTM.

During the implementation, the encoder and the decoder were grouped together in a single part of the DFWP as a DFWAE model. In contrast, this combined part and the prediction part have been designed so that they would be developed, fine-tuned, and trained sequentially. First, the DFWAE was completed and then the objective was to use the weights obtained from the DFWAE to train the predictor. For this reason, during the design and development of the DFWAE we always held in consideration the DFWP, so that the decision we made for the AE would be valid also for the predictor.
In the following sections, the architecture of the DFWAE model and the DFWP model are described. Moreover, in Section 4.3 we present the system in which the DFWAE model is deployed and in Section 4.4 we show a comparison between the DFWP and the two models presented in Section 2.2.

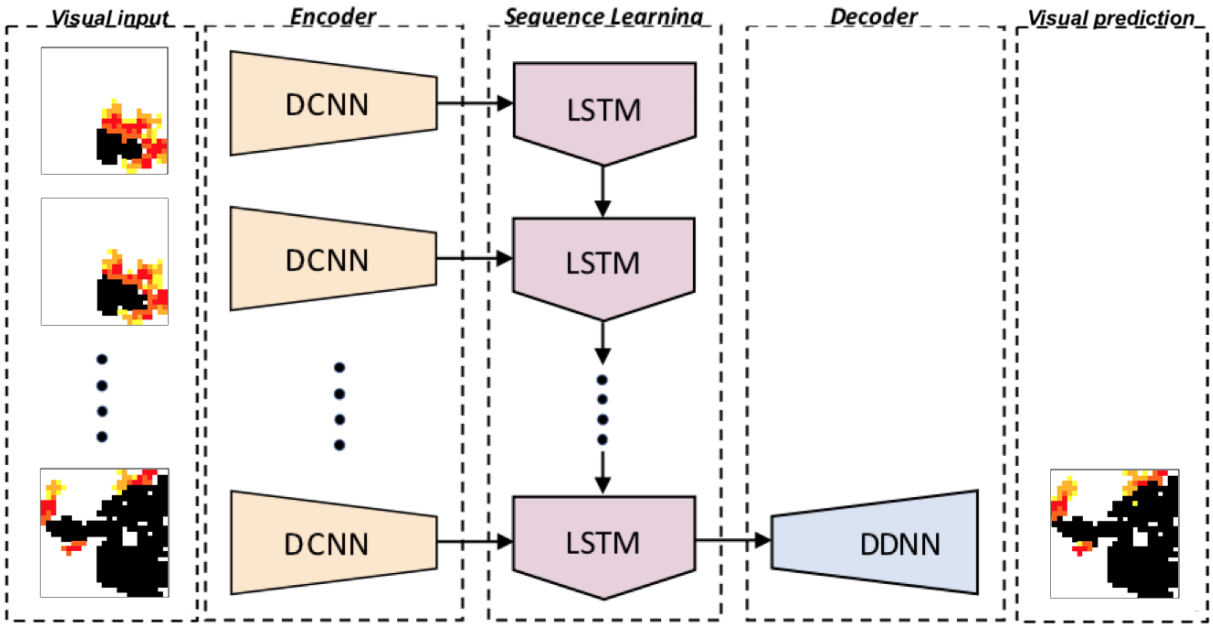


Figure 4.1: High level representation of the Deep Forest Wildfire Predictor

4.1 Deep Forest Wildfire Auto-Encoder

The overall architecture of the model is represented in Figure 4.2. As it can be seen, the DFWAE architecture is symmetrical in terms of the feature maps and in terms of neurons in the hidden layers. As previously discussed in Subsection 2.2.2, in [42] five different CAE are compared, concluding that Model3 and Model4, which have a symmetrical architecture with pooling and unpooling layers, show the best reconstruction quality of MINST images. Although in the paper the models are tested on MINST images while the focus on this thesis is forest fire simulation images, the problem the model is learning is still reconstructing an image independently from its domain, therefore it was hypothesized that these models would give good performances in this domain as well. The data layer is followed by the encoder part, which is represented by a 7-layered CNN having 3 convolutional layers as hidden layers, interspersed with 3 pooling layers and a final FC layer. This is followed by the decoder part, which is represented by a 7-layered DNN with FC as a hidden layer (the reshape layer is considered as part of the FC6-t layer), 3 deconvolutional layers interspersed with 3 unpooling layers. Finally, there is a deconvolutional layer, which acts as a reconstruction layer and two loss layers. The layers of the DFWAE are explained in more detail below.

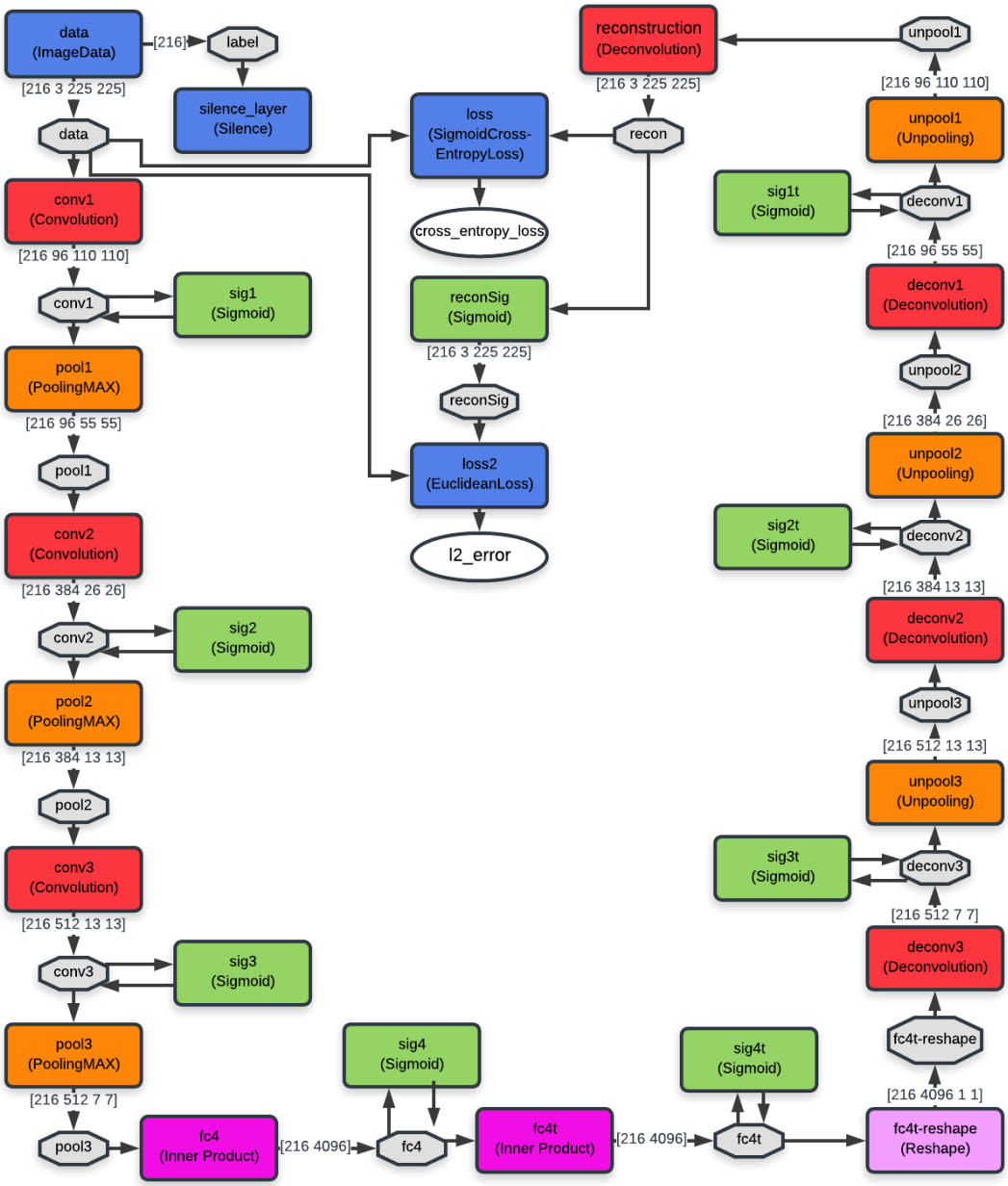


Figure 4.2: Graphical representation of the Deep Forest Wildfire Auto-Encoder Model in the training phase with a RGB dataset. In the testing phase, the batch size would be 27 instead of 216 and with a grayscale dataset the output blob of the data, reconstruction, reconSig layers would have 1 channel instead of 3

Data Layer The data layer used for this model is ImageData. A data layer simply functions as the entrance for the data in the Caffe model. The ImageData layer is a type of Caffe data layer that reads the images directly from disk and returns 2 top blobs: image and label. An alternative to ImageData would be to use a Lightning Memory-Mapped Database (LMDB) data layer, which allocates memory on the GPU and reads the images from there. The advantage of using the second layer would have been the faster training, however, this model has been designed to also predict based on satellite images, which would mean a larger dataset. Thus, the memory allocation of GPU, in that case, would not be practical. For this reason, it was decided to use the ImageData layer for the DFWAE model.
The data is the single frame representing the fire spreading (see Chapter 3 for more detail about the dataset). Each frame consists of 225×225 pixels. The model has been trained and tested on both grayscale (1 channel) and RGB (3 channels) frames. In both scenarios, the color range of the channel was normalized to {0,1}, by multiplying by (1/256). The batch size was 216 for training and 27 for validation. At each epoch for both the train and validation sets the list of frames was shuffled.
An AE is an unsupervised model, thus the model does not use labels. For this reason, the label blob was managed by a silence layer. A silence layer simply manages the unused data so that it is not printed in the output.

Convolutional Layer In the encoder part of the model, 3 convolutional layers were incorporated. As described in Subsection 2.1.3 a convolutional layer convolves the input data with a set of learnable filters, producing feature maps in the top blob. A previous version of the DFWAE was composed of 5 convolutional layers, which is the same number of convolutional layers used in the LRCN model. However, with that version of the model we encountered the problem of vanishing gradient, which was resolved by reducing the number of convolutional layers from 5 to 3 (more details in Subsection 5.1. The number of filters, kernel size, stride, and padding for each of the convolutional layers used in the DFWAE are shown in Table 4.1. The weights of each convolutional layer are initialized with Xavier initialization (see Subsection 5.3.2) and the bias is initialized with a constant (default value is 0). The learning rate multiplier and weight decay for the filters of each convolutional layer are 1; the learning rate multiplier and weight decay for the biases of each convolutional layer are respectively 2 and 0.

Activation Layer The activation function is a very important element of neural networks. In fact, it transforms the input in a non-linear manner in order to facilitate learning and performing more complex tasks. In the literature, ReLu is the most activation function

Layer	# filters	kernel size	stride	padding
conv1	96	7×7	2	0
conv2	384	5×5	2	0
conv3	512	3×3	1	1

Table 4.1: Description of the convolutional layers in the model

used, however, in the DFWAE model a sigmoid layer gives better performance. The discussion about the use of sigmoid functions instead of ReLu functions is further discussed in 5.3.1. In Caffe, a sigmoid layer, applying an elementwise operation also supports in-place computation. Thus, once the layer applies the sigmoid function on the data, it saves the results over the existing data in the bottom blob. In other words, the top blob of the layer becomes the same as the bottom blob, allowing to preserve memory consumption. For this reason, all the activation functions in the model (except for the one that follows the reconstruction layer) are in-place sigmoid layers. The sigmoid layer following the reconstruction layer, reconSig, differs from the other layers. In fact, we couldn't make the reconSig layer in-place, because its top blob, deconv1neuring, is used only by the second loss layer (Euclidean layer) while the first loss layer (sigmoid cross-entropy layer) uses the top blob of the reconstruction layer without the sigmoid function. The reason is that the sigmoid cross-entropy loss layer already has the sigmoid operation in itself, so it would be incorrect to apply the operation twice. Thus, the reconSig layer cannot operate in-place, otherwise, the value of the top blob of the reconstruction layer would have been overwritten.

Pooling Layer A pooling layer performs a downsampling operation along the width and height of the bottom blob. In DFWAE a max pooling function is applied after layers conv1, conv2, and conv3. The kernel size and the stride of all the pooling layers of the model are respectively 2×2 and 2. Therefore, the dimension of the top blob is the dimension of the bottom blob rounded up and divided by 2.

Fully Connected Layer A InnerProduct layer, usually called FC layer, applies an inner product with a matrix of weights. In particular, given a bottom blob $b_i = (N, c_i, h_i, w_i)$, with N being the batch size, c the number of channels, h the height, and w the width, the FC layer first reshapes the data as N vectors of $1 \times x_i$ with $x_i = c_i \times h_i \times w_i$. Then it multiplies each vector $1 \times x_i$ with the weight matrix $W(W_r \times W_c)$ to obtain the output vectors of dimension $1 \times c_o$, which will constitute the top blob $b_o = (N, c_o)$. In particular, in the DFWAE two FC layers are used: fc4, which marks the end of the encoder part of the model and fc4t, which marks the start of the decoder part of the model. Usually, in a CAE

the encoder and decoder part each have a single FC layer and an additional FC layer is used to further reduce the dimensions of the extracted features obtained from the encoder and it is used to connect a possible classifier/predictor. However, as previously stated, the DFWAE has been designed considering the predictor part as well, in which the encoder results would be handled by an LSTM making the use of the third FC layer superfluous. This decision has also been influenced by the LRCN model in which the encoder part is directly connected to the LSTM layer.

In both the layers the number of the filter is set to 4,096, so the dimensions of the weight matrix in each FC layer is calculated as:

- the bottom blob of fc4 is first reshaped to vectors of dimension $1 \times (384 \times 7 \times 7) = 1 \times 18,816$. Thus, $W(18,816 \times 4,096)$;
- the bottom blob of fc4t is from the previous FC layer, so $W(4,096 \times 4,096)$.

The weight initialization uses Xavier (see Subsection 5.3.2) and the bias is initialized with a constant (default value is 0). As standard practice in Caffe, the learning rate multiplier and weight decay for the filters of each convolutional layer are 1; the learning rate multiplier and weight decay for the biases of each convolutional layer are respectively 2 and 0.

A reshape layer (fc6t-reshape) follows the second FC layer. The reshape layer doesn't modify the values of the data but its dimension only: in particular from $batch\_size \times 4096$ to $batch\_size \times 4096 \times 1 \times 1$. In previous versions of Caffe, this operation would be automatically managed from the FC layer depending on the necessity of the subsequent layer.

Deconvolutional Layer As presented previously in this section, the DFWAE is symmetrical. Due to this characteristic, there are 3 deconvolutional layers in the decoder part of the model (the fourth one, reconstruction, is not considered as part of the decoder part as the data layer is not considered part of the encoder). As shown in Figure 4.2, the relation between the convolutional layer and its corresponding deconvolutional layer is indicated by the number in their name (e.g. conv2 is mirrored by deconv2). As described in Subsection 2.1.4, a deconvolution layer applies a transposed convolution to the input matrix.

Table 4.2 shows the number of filters, kernel size, stride, and padding of each deconvolutional layers. These values are obtained using the following rules:

- the feature maps of each deconvolutional layers has to be equivalent to the respective convolutional layer. To guarantee this, the Formula 2.2 is applied to any new combination of parameters.
- if possible, maintain the same value for the parameters of the respective convolutional layer;
- kernel intervals are $\{3 \times 3\}$ and $\{5 \times 5\}$, stride interval and padding interval are $\{0, 2\}$.

Analogous to the convolutional layers, Xavier (see Subsection 5.3.2) initialization [11] is used in all the layers to initialize the weight and the bias is initialized with a constant. Moreover, the learning rate multiplier and weight decay for the filters of each convolutional layer are 1; the learning rate multiplier and weight decay for the biases of each convolutional layer are respectively 2 and 0.

Layer	# filters	kernel size	stride	padding
deconv3	512	7×7	1	0
deconv2	384	3×3	1	1
deconv1	96	7×7	2	1
reconstruction	1[3]	7×7	2	0

Table 4.2: Description of the deconvolutional layers in the model

Table 4.2 provides a list of the parameters of the reconstruction layer. This layer is defined using the same initialization of the other deconvolutional layers and its parameters are obtained by following the same rules listed above. In the table, there are two values for the number of filters of the reconstruction layer. The number of filters used is dependent on whether the images in the dataset are in grayscale or RGB.

Unpooling Layer Opposite to the pooling layer, an unpooling layer performs an upsampling operation along the width and height of the bottom blob. In the DFWAE model, a max unpooling method is applied after the layers deconv1, deconv2, and deconv3. The kernel size and the stride of all the max unpooling layers of the model are respectively 2x2 and 2. In the layers of the model, there are no switch variables. This is due to the fact that the layers of the DFWAE should be the same of the predictor (DFWP) as much as possible, and in DFWP the frames decoded are representing the frame successive to the ones encoded. In particular, a pool_mask variable defines the precise position to restore the

maxed-pooled feature within the unpooled output feature map. As a pool_mask was not used in this case, the maxed-pooled feature was restored in a predefined position, which in the unpooling layer implementation, [19] is the left-top corner position $\{0, 0\}$ of each pool. Because the top blob of the unpooling layer should have the exact size of the bottom blob of the respective pooling layer, an additional parameter, unpool_size, is added to the unpooling function to specify the width and height of the blob. In Figure 4.2 it can be observed that pool3 transforms the input blob filter dimensions from 13×13 to 7×7 - the pooling layer rounds up the output blob filter size. Therefore, if the unpool_size variable wasn't defined in the unpool3 layer, the output blob filter size would be incorrectly defined as 14×14.

Loss Layers Commonly in machine learning, and in Caffe in particular, the loss function drives the learning by specifying the goal of learning by comparing the inaccuracy of the predicted result with the actual result in order to evaluate the correctness of the parameter settings of the model [37]. As suggested by [39][42], for a CAE, it is better to use the following two loss layers as cost functions: sigmoid cross-entropy loss and Euclidean loss. Both the layers compare the input frame and the reconstructed frame, with the difference being that the first function gets the input blob directly from the deconvolutional reconstruction layer, while the second loss function gets the blob after the sigmoid function has been applied to it.

The cross-entropy (logistic) loss function is defined as:

$$E = -\frac{1}{N}\sum_{n=1}^{N}[y_n \log \hat{y}_n + (1 - y_n)\log(1 - \hat{y}_n)], \tag{4.1}$$

where N is the number of samples, y are the targets $y \in 0, 1$, $\hat{y}_n \equiv f(w \cdot x_n)$, where f is a logistic function, w is the vector of weights optimized through gradient descent and x is the input vector.

The Euclidean (L2) loss function is defined as:

$$E = -\frac{1}{2N}\sum_{n=1}^{N}\|\hat{y}_n - y_n\|_2^2, \tag{4.2}$$

where $\hat{y}$ are the predictions $\hat{y} \in \{-\infty, +\infty\}$ and y are the targets $y \in \{-\infty, +\infty\}$.

In the DFWAE we defined both the loss functions to observe the progress of the training. However, in Caffe only one loss function can be used to backpropagate the error, following [42]in the DFWAE this function is the sigmoid cross-entropy loss function.

4.1.1 Architecture parameters

The learning parameters are calculated in the following ways:

- the data layer, pooling layer, unpooling layer, sigmoid layer, and loss layers don't have any learning parameters. In fact, the data layer is just an input for the data and the other layer apply fixed functions on the data, that won't change during the learning process;
- the learning parameters of a FC layer correspond to the number of weights plus the number of biases of the layer, if applicable. The number of weights is equal to the number of inputs to the layer multiplied by the number of nodes of the layer. If the layer contains biases, their number corresponds to the number of nodes of the layer;
- similarly, the learning parameters of a convolutional layer and a deconvolutional layer correspond to the number of weights plus the number of biases of the layer, if applicable. However, in this case, the number of weights is calculated by multiplying the kernel size by the number of filters by the number of inputs to the layer. Moreover, if the layer contains biases, their number corresponds to the number of filters.

Two other observations have to be added to the previous list:

- the input of the first convolutional layer is equal to 1 if the images of the dataset are grayscale, otherwise it is equal to 3. In Table 4.3 the first row shows the calculation for the model, using grayscale images and the second row shows the calculation by using the RBG images;
- when passing outputs from the last pooling layer of the encoder to the first FC layer the output has to be flattened by multiplying the dimension of the data from the convolutional layer by the number of filters of that layer. Thus, the number of inputs of the first FC is $384 \times 7 \times 7 = 18,816$.

4.1.2 Solver

The solver manages model optimization by coordinating the network's forward inference and backward gradients to update the parameters in the model in a way that minimizes the loss. In general, during the learning of the model while the Net updates loss and gradients,

Number of trainable parameters,w(weights)+b(biases), i(inputs), o(outputs)		
Encoder part	Decoder part	Total
$conv1 \rightarrow ((7 \times 7w \times 1i + 1b) \times 96o)+$ $conv2 \rightarrow ((5 \times 5w \times 96i + 1b) \times 384o)+$ $conv3 \rightarrow ((3 \times 3w \times 384i + 1b) \times 512o)+$ $fc6 \rightarrow (18,816i \times 4,096o + 4,096b) =$ $(4,704w + 96b) + (921,600w + 384b)+$ $(1,769,472w + 512b) + (77,070,336w+$ $4096b) =$**79,771,200**	$fc6t \rightarrow (4,096i \times 4,096o + 4,096b) =$ $deconv3 \rightarrow ((7 \times 7w \times 4096i + 1b) \times 512o)+$ $deconv2 \rightarrow ((3 \times 3w \times 512i + 1b) \times 384o)+$ $deconv1 \rightarrow ((7 \times 7w \times 384i + 1b) \times 96o)+$ $reconstruction \rightarrow ((7 \times 7w \times 96i + 1b) \times 1o) =$ $(16,777,216w + 4,096b) + (102,760,448w+$ $512b) + (1,769,472w + 384b)+$ $(1,806,336w + 96b) + (4,704w+$ $1b) =$ **123,123,265**	**202,894K**
$conv1 \rightarrow ((7 \times 7w \times 3i + 1b) \times 96o)+$ $conv2 \rightarrow ((5 \times 5w \times 96i + 1b) \times 384o)+$ $conv3 \rightarrow ((3 \times 3w \times 384i + 1b) \times 512o)+$ $fc6 \rightarrow (18,816i \times 4,096o + 4,096b) =$ $(14,112w + 96b) + (921,600w + 384b)+$ $(1,769,472w + 512b) + (77,070,336w+$ $4096b) =$**79,780,608**	$fc6t \rightarrow (4,096i \times 4,096o + 4,096b) =$ $deconv3 \rightarrow ((7 \times 7w \times 4096i + 1b) \times 512o)+$ $deconv2 \rightarrow ((3 \times 3w \times 512i + 1b) \times 384o)+$ $deconv1 \rightarrow ((7 \times 7w \times 384i + 1b) \times 96o)+$ $reconstruction \rightarrow ((7 \times 7w \times 96i + 1b) \times 3o) =$ $(16,777,216w + 4,096b) + (102,760,448w+$ $512b) + (1,769,472w + 384b)+$ $(1,806,336w + 96b) + (14,112w+$ $3b) =$ **123,132,675**	**202,913K**

Table 4.3: Calculation of the number of trainable parameters in the Deep Forest Wildfire Auto-Encoder model

the solver oversees the optimization and generates parameter updates.
In machine learning the parameters of the model are those properties that are learned during the training of the model, for example, the weights and the bias of the layers, while the hyperparameters cannot be learned and are set beforehand, like the number and size of the hidden layers, activation functions, learning rate, and its decay, etc. In Caffe, the hyperparameters that are not defined directly in the model architecture are defined in the solver. In our thesis, most of the decisions made regarding the hyperparameters of the solver were made by following the guidelines of these websites [38][40].

SGD In DFWAE a Stochastic Gradient Descent (SGD) without momentum is used to optimize the model. Formally, the weights W are updated at iteration $t + 1$ as shown:

$$V_{t+1} = V_t - \alpha \nabla L(W_t) W_{t+1} = W_t + V_{t+1} \tag{4.3}$$

Where V_{t+1} represents the new weight update, α is the learning rate, and $\nabla L(W_t)$ is the negative gradient of V_t.

Learning rate The base learning rate of the network, or in other words, the initial learning rate with which the model starts training, is 1e-05. The value of this parameter has been chosen empirically, a more detailed discussion of it can be found in 5.3.3.
To decide which learning policy to use in the DFWAE, the "step" policy was attempted first with momentum 0.9 and gamma 0.1 used by the LRCN presented in [7], following by the "fixed" from the CAE presented in [42]. The "fixed" presented significantly better results.

Test As discussed in previous chapters, equation 2.3 describes the relations between iteration and epoch. In particular, as discussed in Section 3.3, the train set has 932 images and the test set has 233 images. Thus, given the batch sizes of 216 for training and 27 for testing, each epoch corresponds to relatively 4 train iterations and 9 test iterations.

In the DFWAE network, the training phase occurs every 100 iterations, thus every 25 epochs. And in each testing phase there are 100 iterations, thus 12 epochs.

Number of iterations The max number of training iterations has been set to 120,000, which is equivalent to 30,000 epochs. The number was chosen empirically by picking a value after the loss curve of the validation decelerates and before it gets too close to the loss curve of the training.

Snapshots A snapshot is taken every 30,000 iterations. Thus, when running the network 4 snapshots are taken. The use of snapshots is just for practical proposes. In fact, if the system on which the model is training stops, the snapshot allows for restarting from the given point. However, because they occupy a considerable amount of memory the number of snapshots has to be low. During the training of the model, the VMs on which the model was running had certain issues, such as problems with the GPU or unforeseen shutdowns of the machine. Having snapshots during the training permitted to avoid running the model from the beginning, which sometimes saved more than a day of training.

Solver mode The network is run on a single GPU.

4.2 Deep Forest Wildfire Predictor

This part of the research project has been designed and initially tested. But as the scope of the DFWAE component grew and required more experimentation this prediction phase will remain a preliminary work with full experimental validation reserved for future work. The overall architecture of the model is represented in Figure 4.3. As it can be observed from the figure, the following layer changes were applied to the DFWAE:

- the data layer has been modified to a Python data layer so that instead of single frames the model reads as input sequences of 9 frames and reads the 10th frames as the label. Moreover, the layer returns clip markers that indicate the beginning and the end of the sequence. A more detailed explanation of this layer is described below;
- a reshape layer was added after the encoder part so that the bottom blob of the LSTM layer has a size of $(number\_of\_frames \times number\_of\_videosfeatures)$;
- the LSTM layer is added to read the sequence of frames and predict the following frame. A more detailed explanation of this layer is described below;
- the batch size of the encoder part and the batch size of the decoder part differ. In fact, the encoder batch size is given by $(number\_of\_frames \times number\_of\_videos)$ while the decoder batch size is just the number of videos of the batch;
- the loss functions take the label frame representing the successive frame as arguments, instead of the input frame;

The new layers added to the DFWAE (Section 4.1) to make the DFWP are explained in more detail below.

Python Data Layer This layer reads all the simulations of fire spread of the dataset, divides each of them into 10-frames sequences, creates the train and validation sets, and returns the data, label, and clip_marks blobs. A clip_marker is used with LSTM to indicate when a new sequence starts. In particular, a clip_marker is a boolean vector with the same dimension of the sequence it is referring to and has 0 as the first element, while all others are equal to 1.
For each fire spread simulation the algorithm counts the number of frames to calculate the number of 10-frames sequences obtainable, $\left\lceil \frac{\#frames}{10} \right\rceil$. Thus, it returns the maximum number of sequences with the minimum overlaps between two adjacent sequences. Below is the Python code used to calculate the starting point of each sequence.

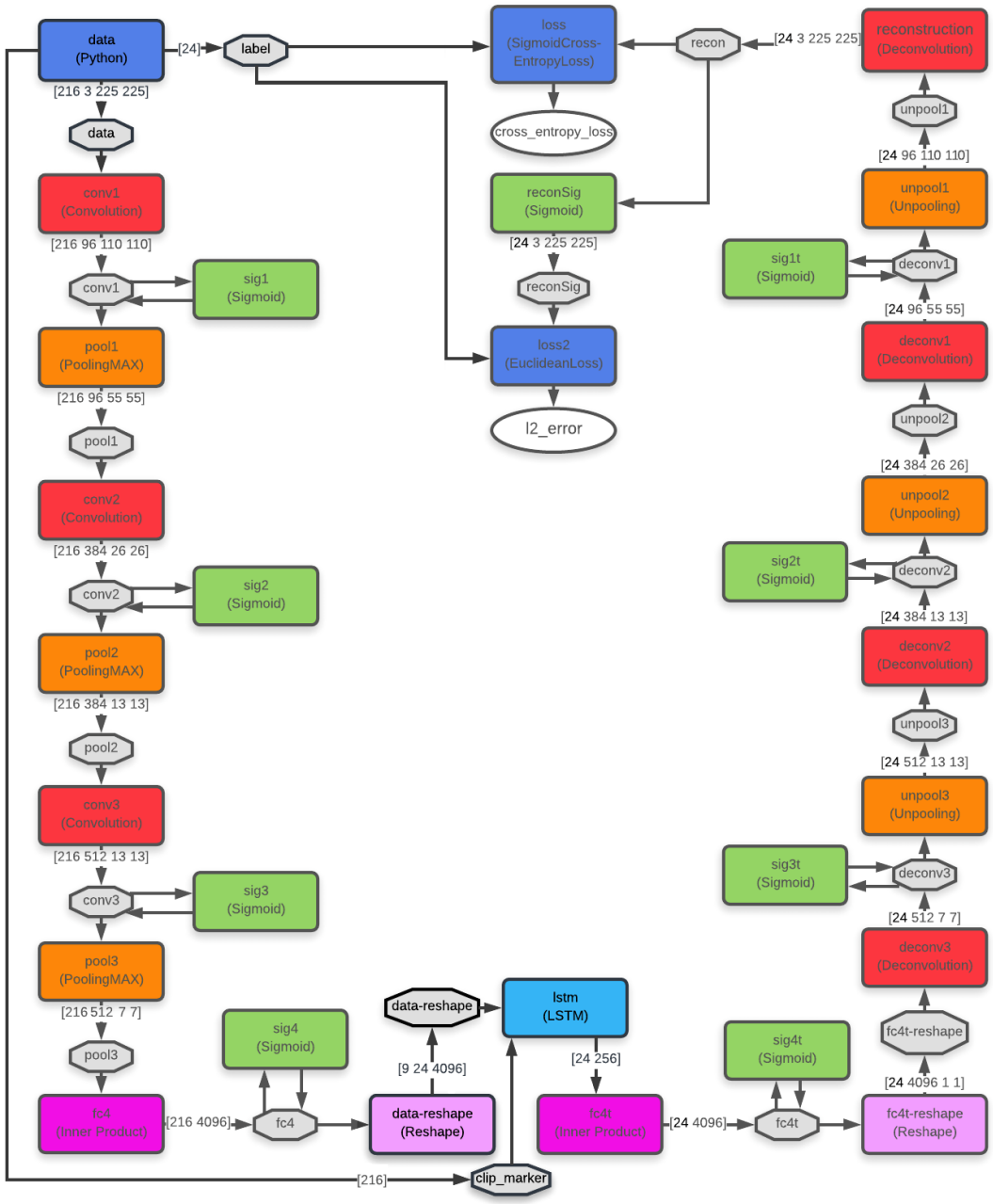


Figure 4.3: Graphical rapresentation of the Deep Forest Wildfire Predictor Model in the training phase with a RGB dataset. In the test phase the batch size was (3×9) instead of (24×9), thus in the encoder a batch of 27 images was used instead of 216, the data-reshape dimension was [9 3 4096], and in the decoder part a batch of 3 images was used instead of 24.

```
data_seq = []
for simulation in next(os.walk(dataset))[1]: #for each simulation
  dir_x= dataset+"/"+simulation;
  num_frames = len(os.listdir(dir_x));
  num_seq = int(math.ceil(num_frames / 10.0));
  idx = 0;
  #print ("Elem "+str(num_frames)+ " seq "+str(num_seq));
  for seq in range(1,num_seq):
    data_seq.append(dir_x+"/"+str(idx)+".png");
    overlap = int((10*(num_seq-seq+1) - num_frames)/(num_seq-seq));
    #print "Seq " +str(seq) +"start "+str(idx) + " overlap "+str(
        overlap)
    idx = idx + (10 - overlap);
    num_frames -= (10 - overlap);
  data_seq.append(dir_x+"/"+str(idx)+".png");
```

Listing 4.1: Calculation of the sequences of input

The commented lines have been used to obtain the information about the sequences, as the number of sequence for each simulation and the number of overlapping frames for each adjacent sequences, reported in Table 4.4. As it can be observed from the table, and as presented in Section 3.1 the fire simulations have variable dimensions, ranging from 10 frames to 78 frames, depending on the time it took the fire to die.

A 5-fold cross validation similar to the one discussed in Section 3.3 is then applied to the data. However, instead of considering the single frame, each sequence is considered as an entity. Finally, the training and the validation sets are divided into batches, each batch contains the data and the label which are saved in the respective blobs. In particular, for the training data, each batch of data contains 24 sequences of 10 frames, with the first 9 frames being data and the 10th frame being the label. The 24 sequences are put in the blob data interposed from each other. Because the input data is a sequence of fixed dimension during training the clip_marker blob will always be comprised of 0 for the first 24 elements, to indicate the start of each sequence and of 1 for the other elements. The same happens for the validation data, labels and blobs with the consideration that instead of 24 sequences there are 3 in each batch.

LSTM LSTM is a recurrent neural network, see Section 2.1.2. The main difference of a LSTM from a standard recurrent neural network is given by the cell state. In Figure 4.4, the cell state is represented by the horizontal line going from C_{t-1} (the previous cell

simulation	# frames	# 10frame-sequence	overlapping elements
1	14	2	6
2	65	7	0 in the first and 1 in the others
3	49	5	0 in all except 1 in the last
4	15	2	5
5	21	3	4 and 5
6	44	5	1 in all except 2 in the last two
7	47	5	0 in first one and 1 in the others
8	65	7	0 in the first and 1 in the others
9	75	8	0 in the first two and 1 in the others
10	10	1	0
11	20	2	0
12	41	5	2 in all except 3 in the last
13	54	6	1 in all except 2 in the last
14	78	8	0 in all except 1 in last two
15	36	4	1 in all except 2 in the last
16	15	2	5
17	54	6	1 in all except 2 in the last
18	29	3	0 in all except 1 in the last
19	42	5	2
20	33	4	2 in all except 3 in the last
21	31	4	3
22	52	6	1 in first two and 2 in the others
23	72	8	1 in all except 2 in the last
24	46	5	1
25	46	5	1
26	33	4	2 in all except 3 in the last
27	78	8	0 in all except 1 in the last 2
TOTAL	**1165**	**130**	

Table 4.4: Simulation dataset division in sequences of 10-frames

state) to C_t (the current cell state). In this figure, the yellow rectangles represent 4 neural networks layer. From the left, the first one which outputs f_i indicates what information of the previous cell state should be forgotten and what should be saved given the result of the previous cell and the new input. The two following layers define what of the new information should be used to update the cell state. Finally, the last layer selects what

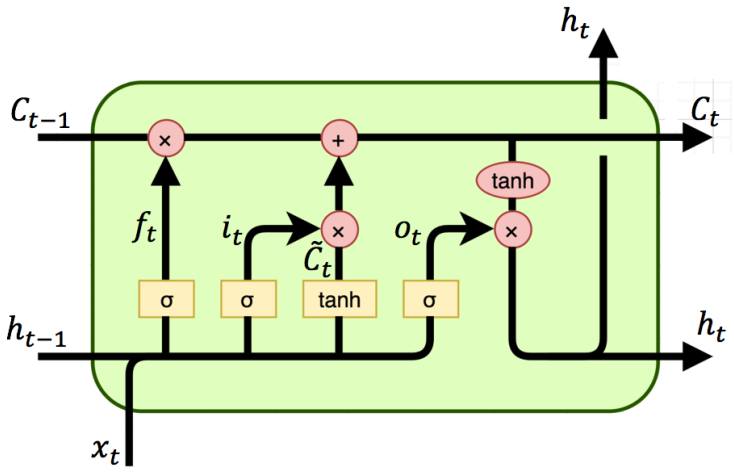


Figure 4.4: Long Short Term Memory

parts of the updated cell status should be returned as the output of the layer.

In DFWP the LSTM layer takes as bottom blobs the data-reshape, which is the feature maps obtained from the encoder part, and the clip_marker. The decision to set the output value to 256 has been taken considering the result obtained by [7]. In particular, they run their model with 256, 512, and 1024 output and they concluded that the best result with RGB data was with 256. The layer uses a uniform filler for the weights with interval $\{-0,01; 0,01\}$ and a constant filler for the bias.

4.3 Environment Specifications

The model has been developed using Caffe [22] implemented in [46]. This version of Caffe includes all the layers used in the model except for the unpooling layer. The unpooling layer included is from [19], which they obtained by implementing [29] [18]. To run the DFWAE model, VMs in the Google Cloud Platform were used. In particular, the machines were clones of the Caffe Python 3.6 NVidia GPU Production available on the platform. The framework runs on a Debian 9 operative system and contains Caffe 1.0, OpenCV 3.3.0, Python 3.6.3, CUDA 9.0.176, cuDNN 7.0.5, and NVidia drivers 384.98. The deployed VMs have 8 cores vCPUs with 52 GB memory, 1 Nvidia Tesla P100 GPU, and 40GB of the standard persistent disk. During the environment setup, Caffe 1.0 was overwritten with the ones discussed at the beginning of this section, in order to have all the required classes available.

4.4 Comparison with existing models

As mentioned in Section 2.2, the DFWP model has been created by taking two models as examples: the LRCN presented in [7]. More specifically, the activity recognition model and Model 4 of the CAE presented in [42]. In particular, from [7] the model takes the notion of using a CNN for feature extraction of each frame by using LSTM to make a prediction, which it considers the features extracted from all the frames of the video. From [42] the model takes the notion of using a CAE with the encoder and decoder parts symmetric for image reconstruction. In the following subsections is presented a brief comparison of the models to DFWP by listing their analogies and their dissimilarity.

4.4.1 LRCN

The DFWP is similar to the LRCN model for the following points:

- CNN + LSTM structure, or in other words use of a CNN for feature extraction plus the use of LSTM for prediction;
- have as input a sequence of frames and return a single element;
- the kernel_size, padding, stride, number of filters of conv1, conv2, and conv3 in DFWP are the same of respectively conv1, conv2, and conv5 in LRCN;
- 3 max-pooling with kernel size and stride of 2;
- FC layers after CNN and LSTM;
- dataset is divided in two subsets, train and validation, instead of three (see Section 3.3.1).

Other than for the most obvious aspects, such as returning a next step prediction instead of a classification and having a DNN part, DFWP differs from the LRCN model for the following aspects:

- CNN composed by 3 convolutional layers instead of 5 (see Section 5.1);
- sigmoid activation functions instead of a ReLu (see Section 5.3.1);
- weight initialization with Xavier instead of Gaussian;
- use of Euclidian loss layer instead of accuracy layer. In general, accuracy is used for classification while Euclidian loss is used for regression.

4.4.2 DCAE

As presented previously, the result from the comparison of the models presented in the paper shows that Model3 and Model4 provide the best reconstruction quality of the images. However, in the DFWP model, the use of switch variables would not make sense due to the fact that the encoded images (representing the previous steps of the fire) and the decoded image (the following step) are not the same. For these reasons, the DFWP has been designed based on Model4. In particular, the two models are similar for the following points:

- CNN + DNN structure, or in other words use of a CNN for encoding plus the use of DNN for decoding;
- encoder and decoder architecture are symmetrical;
- every convolutional layer is followed by a pooling layer and every deconvolutional layer by a unpooling layer;
- weight initialization and bias in the convolutional layers;
- no switch variables;
- sigmoid as activation function;
- deconvolutional layer as reconstruction layer;
- Sigmoid cross entropy and Euclidean loss functions.

Aside from the most obvious aspects, such as decoding a next step prediction instead of the same frame and having a LSTM, DFWP and CAE Model4 have the following differences:

- Python data layer instead of HDF5 data layer;
- use of reshapes after FC layer in the decoding part. This is given by the fact that the paper uses an older version of Caffe;
- number of layers in the encoder and decoder;
- the kernel_size, padding, stride, number of filters.

Moreover, this thesis takes the idea of printing the filters of each layer for debugging proposes from [42]. However, the paper used Matlab for the deploying phase, while in this thesis Python was used to write the deploying phase.

Chapter 5

Results and Observations

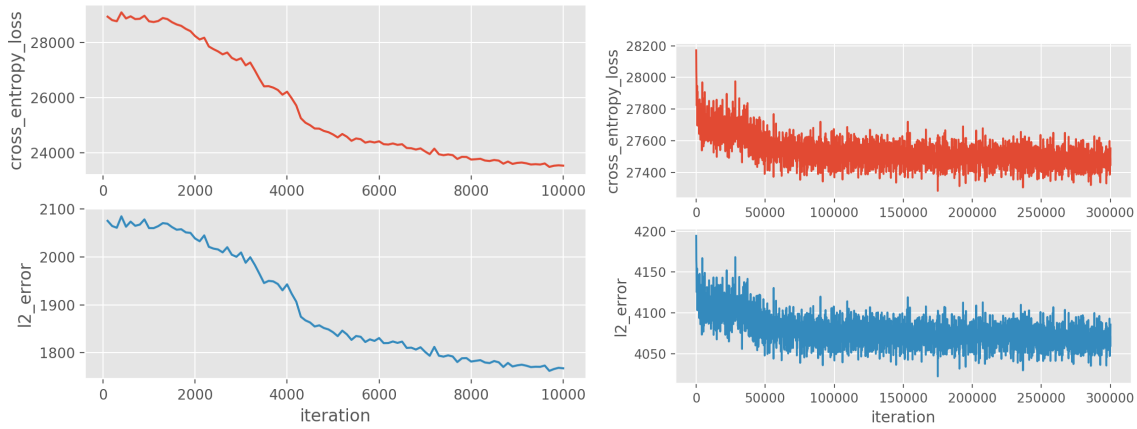
5.1 9-layered Auto-Encoder

The DFWAE model was initially designed to be a 9-layered CAE. In fact, the encoder architecture was built with the same number of layers of the CNN of the LRCN model presented in [7]. In particular, the encoder was a 9-layered CNN composed of 5 convolutional layers, 3 pooling layers, and a final FC. Moreover, following the idea of having the encoder and the decoder architecture symmetric, the decoder part was a 9-layered DNN composed of a FC layer, 5 deconvolutional layers, and 3 unpooling layers.

As described in Subsection 3.1.1, we first run the model with the original unprocessed data and as it can be seen in Figure 5.1a, the loss functions would converge. However, as discussed in the same subsection, the learner would focus on reconstructing the single trees, seeing the fire as noise. In other words, the 9-layered model was learning, however, it was learning the wrong information.

As presented in Subsection 3.1.1, we resolved this problem by preprocessing the data, and removing the symbols, before using it in the model. However, as shown in Figure 5.1b when we run the model with the modified dataset, it didn't converge even after $300K$ iterations ($1,389$ epochs). Likewise, in Figure 5.2 the output images of the same learner applied to 2 different images are shown.

The problem we encountered was the vanishing gradients problem [15][2][11] and one



(a) Run with original dataset

(b) Run with preprocessed dataset

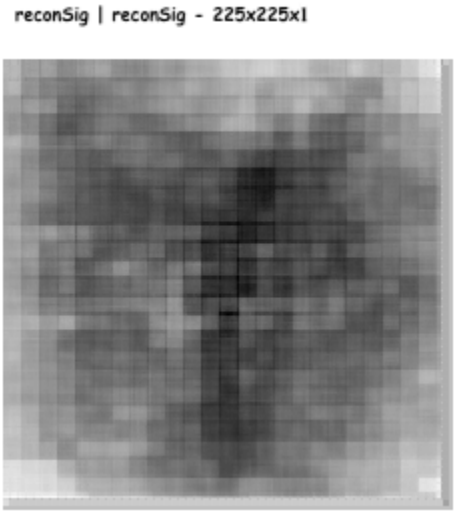
Figure 5.1: Loss functions of the 9-layered Deep Forest Wildfire Auto-Encoder

of the causes is learning from a small dataset using a neural network architecture that is too deep.

This problem is resolved when reducing the depth, hence the number of layers, of the DFWAE model. In particular, we removed the third and fourth convolutional layers of the original model. However, before reducing the depth of the model three common solutions were considered:

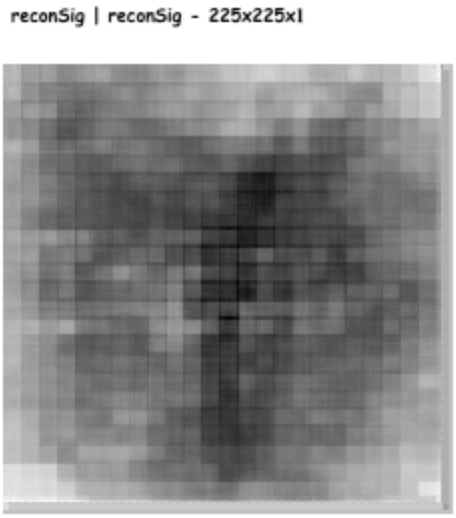
- normalizing the input data - which is done in the input layer, 4.1;
- using switch variables - in particular, they would connect points of the encoder to the decoder. However, as discussed in Section 4.1, in DFWP the image reconstructed from the decoder portion is not the same transformed from the encoder portion. Hence the use of switch variables in the DFWAE in this scenario would not provide any benefits;
- use of ReLu [20] - however, in the DFWAE the use of ReLu instead of Sigmoid would introduce an error as discussed in Subsection 5.3.1.

(a) Input image grayscale (1)



(b) Reconstructed image grayscale (1)

(c) Input image grayscale (2)



(d) Reconstructed image grayscale (2)

Figure 5.2: 9-layered Deep Forest Wildfire Auto-Encoder deployment of two grayscale frames after $30K$ iterations of training

5.2 Results

The experimental results of the DFWAE model are presented in Table 5.1. As presented in Section 3.3, the results are obtained by running the DFWAE model using 5-fold cross validation on the RGB and grayscale of the forest wildfire simulator dataset presented in Subsection 3.1.1. The values of the loss functions are presented for the training set and test set. In particular, for each of them, the values of both loss layers Sigmoid_Cross_Entropy_Loss and Euclidean_Loss are separated by a semicolon.

The learning parameters of the solver for all runs are the ones presented in Subsection 4.1.2: the models are run for $120K$ training iterations (thus $30K$ epoch); SGD without momentum is used to optimize the model; base learning rate of $1e^{-5}$ and "fixed" learning policy are used.

Model	Sigmoid_Cross_Entropy_Loss; Euclidean_Loss	
	Train	Test
Grayscale	5,852.75; 373.64	5,987.55; 393.21
RGB	10,405.22; 897.61	11,521.44; 1,065.098

Table 5.1: Performance of the model after $120K$ iterations of training

Figure 5.3 shows the cross entropy loss function and the Euclidean loss function of the DFWAE model run on the preprocessed simulated dataset, in both RGB and grayscale versions. As it can be seen:

- all the functions have the first downfall in common, which happens at the first iteration and stems from the fact that the model is not pretrained. Therefore, this value is not really representative of the model per se and can be omitted;
- then the functions stay relatively constant until they have another sudden decrease, which happens between the $25K$ to $30K$ iterations in the RGB run and $50K$ to $75K$ iterations for the grayscale run of the model. These decreases indicate the points where the model starts to learn the main features of the image to be able to reconstruct it correctly;
- around $40K$ iterations for the RGB run and $90K$ iterations for the grayscale run, the function starts decreasing at a substantially slower rate. Although the function never reaches a constant value in the $120K$ iteration interval, the loss function slope decreases enough that the $60K$ iteration for the RGB run and the $100K$ iteration

for the grayscale run can be considered a good point after which the function can be considered to be constant. Thus, these many iterations are needed to train the model.

After training the model for $120K$ iterations, the obtained network's weights (stored in a caffemodel file) were used for deployment. This stage has the objective to apply the trained model on a dataset towards obtaining an actual prediction. Figure 5.4 shows the reconstruction of the same image in the RGB and grayscale version. As it can be observed, the model is able to reconstruct really well the area of burnt trees and to recognize the areas in which the fire is burning. Moreover, the model gives an acceptable performance in individuating the different level of fires: different shades of red to yellow for the RGB and different shades of gray for the grayscale version. The presence of the left and bottom border is quite notable within both the reconstructed images. These can be corrected by modifying the paddings of the convolutional/deconvolutional layers.

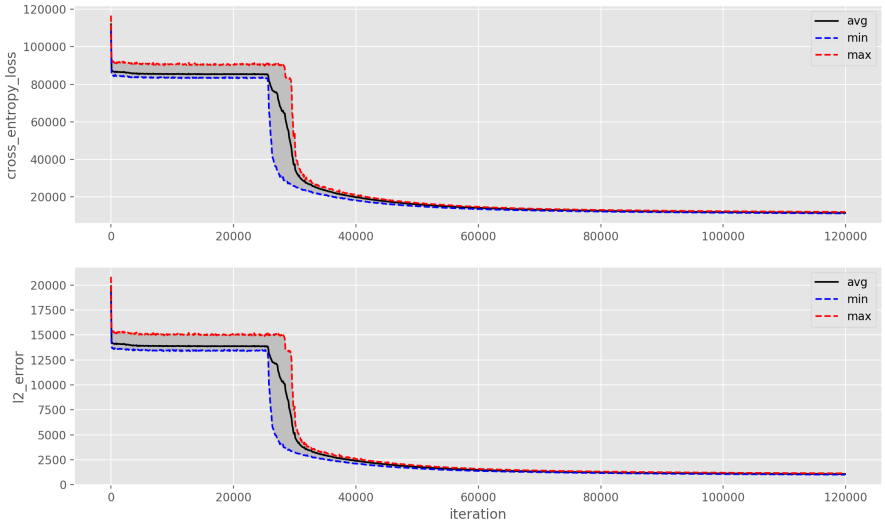
5.3 Methodological consideration

5.3.1 ReLu vs Sigmoid

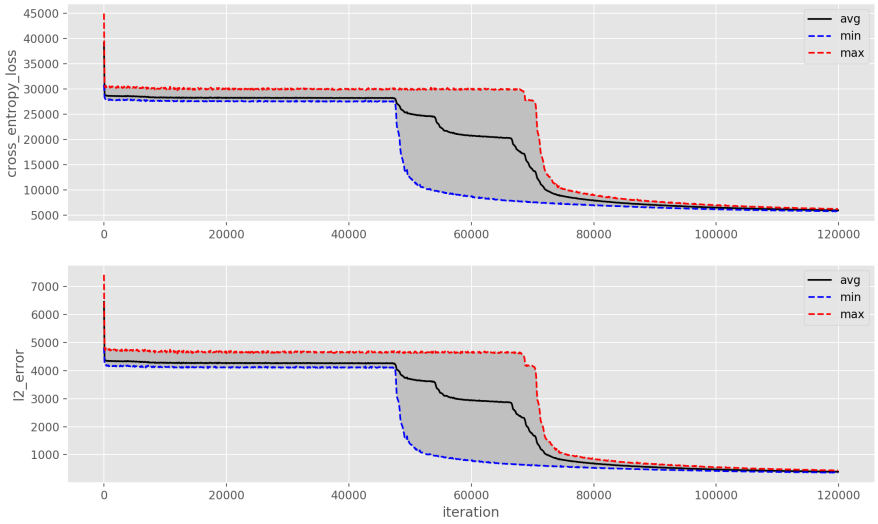
The use of ReLu as an activation function has the main advantage of reducing the probability of vanishing gradients and being faster to train compared to a standard sigmoid function [20][4].

However, when we used ReLu in the model, we found that the loss functions did not converge and the reconstructed image, in the RGB scenario, was a solid color which ranged from black to red. Black represented a NaN output which is caused by the model being trained on the original dataset, before the preprocessing discussed in Subsection 3.1.1, or being trained with the wrong hyperparameters. The image would be reconstructed as a solid dark red with optimal parameters and on the preprocessed dataset. This problem is generally encountered when using ReLu in a CAE [42]. In [12] the problem is analyzed and divided into two points:

- the hard saturation below the ReLu function threshold. Thus, whenever the model tries to reconstruct a zero where there should be a non-zero the gradient cannot be backpropagated;
- the unbounded behavior of ReLu.



(a) Loss functions of the model on the RGB simulation dataset



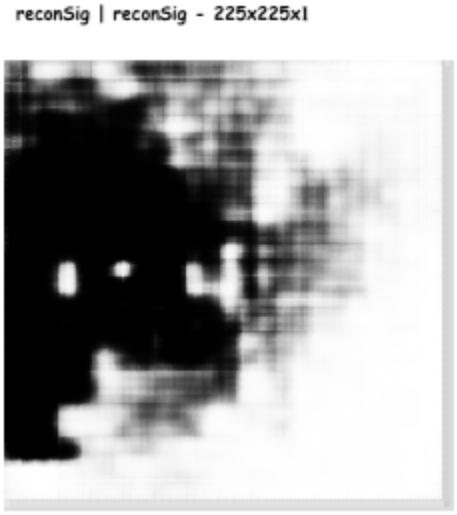
(b) Loss functions of the model on the grayscale simulation dataset

Figure 5.3: Loss functions obtained by the 5-fold cross validation of the model

(a) Input image RGB

(b) Reconstructed image RGB

(c) Input image grayscale



(d) Reconstructed image grayscale

Figure 5.4: Deployment of Deep Forest Wildfire Auto-Encoder on input images (a) and (c), resulting in reconstruction images (b) and (d) in RGB and grayscale respectively

There are solutions that can be applied to use ReLu in a CAE, for example, to cite a few:

- use variation of ReLu such as Exponential linear units (ELU) or leaky ReLu;
- in the reconstruction layer, use a softplus activation function along with a quadratic cost;
- scale the blob coming from the ReLu layer, so that the data is in the interval (0,1) and then use a sigmoid function on the data returned by the reconstruction layer. Finally, use a cross-entropy for the reconstruction loss;

In DFWAE the sigmoid activation function has been used instead of ReLu. The problem of the vanishing gradients has been resolved by reducing the depth of the model (see Section 5.1) and the model returns good results as presented in Section 5.2.

5.3.2 Weight initialization

In ANN, initializing the network's layers with the correct weights can highly influence when the model converges, in particular, if this happens in a reasonable amount of iterations or not. A best practice is to initialize the weights based on which non-linear activation function is used in the model [8]. In particular, a heuristic is to use Xavier initialization [11] when using a sigmoid activation function.
When deciding on the weight initialization method for a network that uses a sigmoid activation function two cases should be considered [23]:

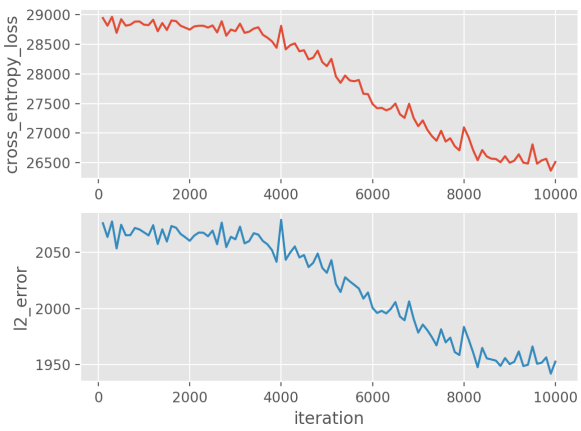
- the weights too small - the variance of the input data will start to decrease as it goes through the layers of the network, dropping eventually to such a low value that won't be useful anymore. In fact, if we observe the sigmoid function on Figure 2.3a, we see that if we apply it to values close to zero the function becomes approximately linear. By not having any non-linearity we lose the advantages of having multiple layers.
- the weights are too large - the variance of the input data will rapidly increase as it goes through the layers of the network, reaching such a large value that won't be useful anymore. In fact, if we observe the sigmoid function on Figure 2.3a, we see that function becomes flat for larger values. Thus, it will become saturated and the gradients will become close to zero.

Using Xavier initialization keeps the signal from exploding to high value or vanishing to zero by assigning the weights from a Gaussian distribution with zero mean and a variance of 1/N, where N specifies the number of input neurons. In the original paper [11], to preserve the backpropagation signal as well, the authors took the average of the number input neurons and the output neurons as N. However, that is more computationally complex to implement so in Caffe, like in most practical implementations, only the number of input neurons is used.

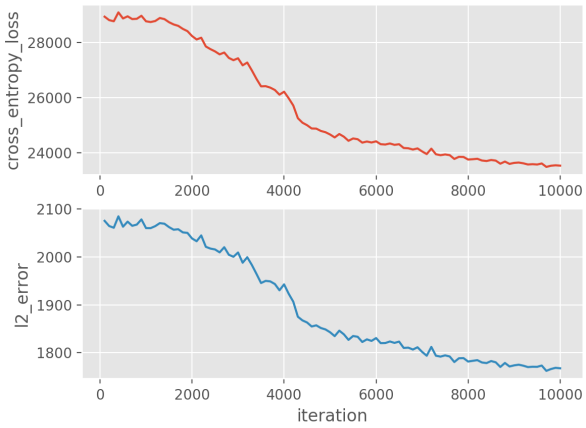
5.3.3 How to decide the initial learning rate

Like most of the hyperparameters of the solver (Subsection 4.1.2), the decision of the initial learning rate is made empirically. If the initial learning rate is too low the loss functions will diverge, returning NaN or Inf. For example, building the DFWAE model with an initial learning rate of 0.01, which usually represent the starting value used when selecting this parameter, would return NaN. To obtain a better value, the parameter is divided by 10 until the loss curve reaches the best convergence curve. In Figure 5.5 three results are shown, each of them representing the build of the same model with a different initial learning rate $1e^{-4}$, $1e^{-5}$, and $1e^{-6}$. The displayed results are from the previous version of the model 5.1 on the original simulation data with a step policy.

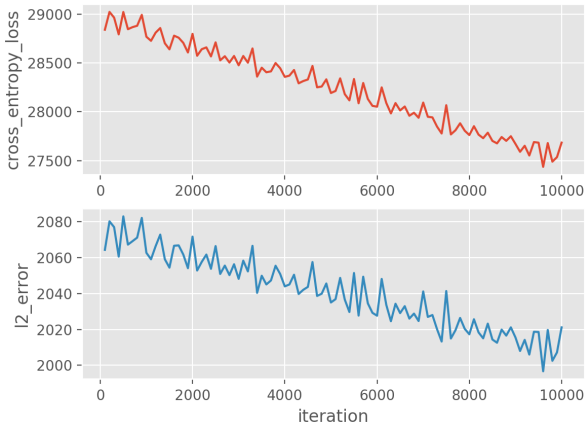
The initial learning rate depends on the architecture of the model and the parameters of the model. Thus, during the development of the model, subsequent to any substantial change of the model, such as changing the learning policy from step to fixed or 7-layered DFWAE instead of the 9-layered, we had to find the new best initial learning rate. However, building a deep model can take days, which means the loss of substantial time and money. To expedite the toning of new versions of the model, we used hyperparameters from the previous version of the model. Moreover, for the initial learning rate, not only did we use the previous initial learning rate but we only would recalculate it if the model didn't converge. However, for the final version of the model, the best learning rate would be calculated. Similar to the previous versions, to speed up the operation, the first initial learning rate considered was equivalent to the one for the previous version, which is $1e^{-5}$. In fact, in all the versions of the model this learning rate value made the loss functions converge so it was never changed. Then, to confirm that the value represents the actual best learning rate in the new model, a smaller value ($1e^{-6}$) and a higher value ($1e^{-4}$) were tested as well.



(a) $base\_lr = 1e^{-4}$



(b) $base\_lr = 1e^{-5}$



(c) $base\_lr = 1e^{-6}$

Figure 5.5: Performance comparison of the model, trained using different initial learning rate

5.3.4 Visualization of the layers output

To debug and observe the model a Python visualization algorithm has been created. The algorithm runs the trained model on a chosen image and returns the visualization of all the output/filters for all the vision layers of the model. The results of the FC and the Reshape layer are not printed. The reason for overlooking the FC layer is that the code would need an additional library to be able to plot the FC graph and for both layers, the result would not be adding any additional information.

Figure 5.6 shows an example output of the different layers of an encoded-decoded image. In each panel, the output filters of a layer are shown, thus the data saved in the top blob of the layer. As can be observed in Figure 5.6 for the data, conv1, pool1, deconv1, unpool1, reconstruction, and reconSig layers the results of all the filters are shown, while for the other layers the figure shows only the results of a subset of the filters. We decide to show only a subset because the number of filters applied in the middle layers is too high and if all of them were shown the image would become unclear. Over each panel is indicated from which layer the blob images are coming from and the size of the blob - the number of filters and the size of each filter. As described previously in Section 4.1, the sigmoid functions are applied to the blobs in-place. For this reason and because the print of the panel happens after the run of the entire model, the output relative to the convolutional and deconvolutional layers have the sigmoid activation function applied to them. As can be seen from the figure, this is specifically indicated on the top of each panel. The exception is the reconstruction layer on which the activation function is applied separately.

Even if the reconstructed image is really close to the original image, the filter reconstructed in the decoding portion does not mirror the encoding portion of the model. This is unfortunate because if the reconstruction outputs were mirroring the encoding outputs, the debugging and optimization of the model would have been easier. However, to force this behavior, switch variables would have been necessary and as discussed in Subsection 4.1 they are not applicable to the model. From the figure, we can also observe that the results of the filters are more clear in the first 2 convolutional layers, and as we go more in depth in the encoder the less the results are interpretable. In particular, in conv1 we can easily observe that the layer is analyzing different combinations between burnt areas, fire, and unburned areas. Moreover, in some images of the panel, we can still differentiate between different levels of fire. In conv2 the images look fuzzier and we don't see the different levels of fire in the same image. However, in different images, we can observe the different levels of fire. As an example, this can be observed if we compare the image in the second row, second column and the image in the third row, first column. From conv3 the details studied from the layer become less intuitive. Conversely to the encoder, in the

decoder, the first deconvolutional layer doesn't return interpretable results but as we go further in depth the results become more clear. Because deconv3 gets its input data from the FC layer, we expected its results to not be easily interpretable. In some images of the deconv2 layer, we can observe a fuzzy representation of the fire. In deconv1 we can observe some images with the burnt areas, some representing it jointly with the fire, also in some of the images we can observe a moderate distinction between the different levels of fire. It is interesting to observe that from the earliest layers the decoder identifies the image as a grid on which the pixel represents the status of the fire.

The visualization was useful mostly in the debugging phase. For example, in the case of the vanishing gradients problem, the visualization of the output of all the visual layers permits to observe at which level the information is lost and this helps to find possible solutions.

5.3.5 Integrating Unpooling Layer in Caffe

Integrating the unpooling layers from [19] to [46] Caffe presented some challenges. In fact, the two repositories implement two different versions of Caffe:

- [19] implements the old version of Caffe which is characterized by having the convolutional layer, pooling layer, unpooling layer, and deconvolutional layer defined in vision_layers.hpp and RNN and LSTM defined in sequence_layers.hpp;
- [46] implements the new version of Caffe, in which all the vision layers and sequence layers are defined individually.

This distinction not only implies that the unpooling layer definition has to be moved from the vision_layers.hpp and defined individually, but also the implementation has to be adjusted to correctly manage the instantiation and registration of the new class. To achieve this the instructions presented in the Wiki in [45] are followed.

5.4 Challenges encountered during deployment

While setting up the environment and deploying the CAE we encountered certain challenges which are listed below.

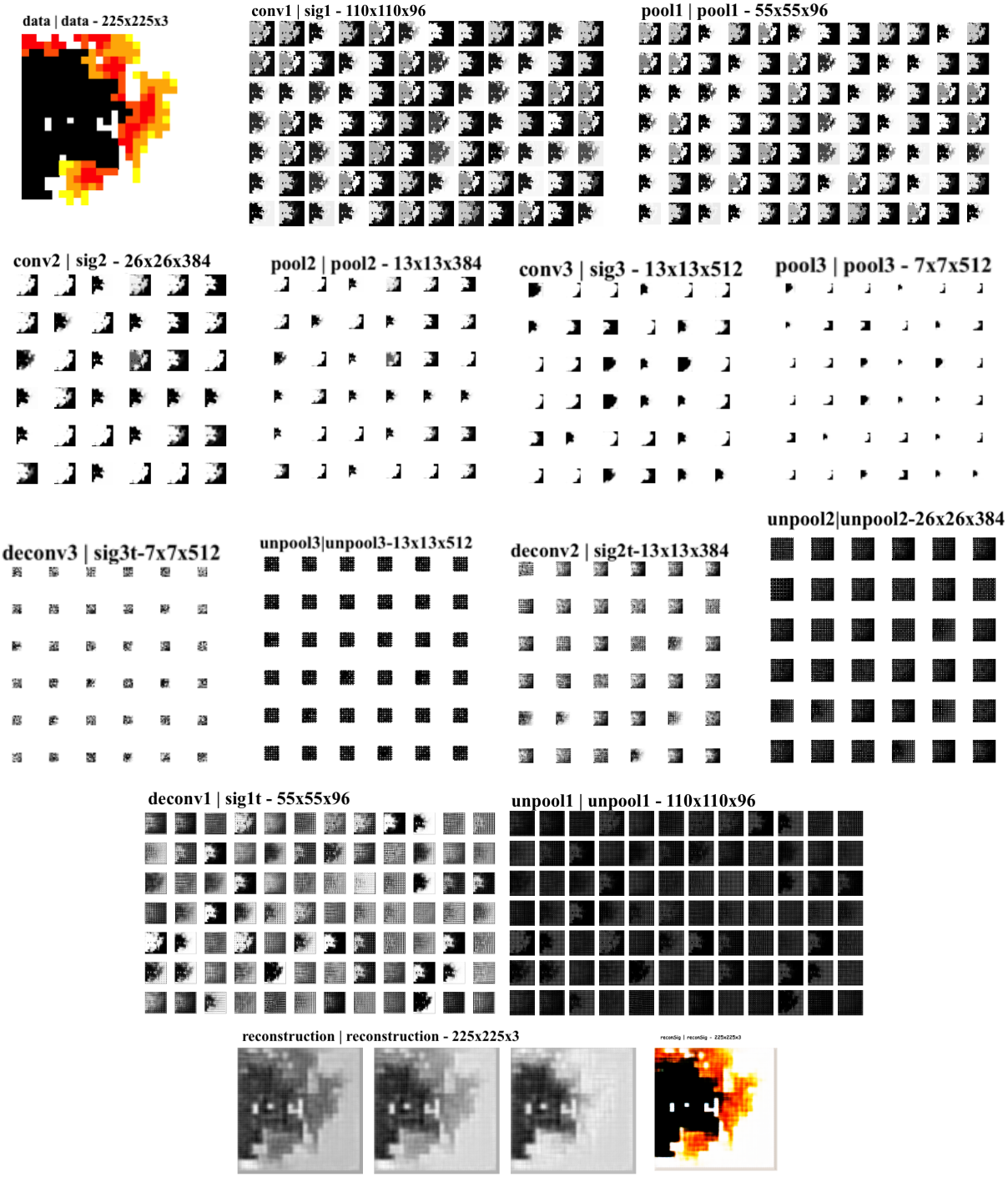


Figure 5.6: Output of each of the visualization layers of the Deep Forest Wildfire Auto-Encoder

5.4.1 Runtime cost of servers for deep learning

The CAE is a deep learning model based on Caffe. The model runs need machines with a high performance GPU to be able to give results in a reasonable amount of time. However, high performance machines have also high prices. When using the Google cloud server, as presented in Section 4.3, VMs with a Nvidia Tesla P100 GPU has been used. To obtain the minimum results needed to theorize if a run was valid or problematic, it would take at least 2 days. While to run the model with 5-fold cross validation, after the debugging and to compare final results took over half a month. For this reason, a minimum of 3 different VMs had to be used to expedite the debugging process and optimization process.

In future works, when the model will be applied to satellite images the computational complexity may increase. The dimensions of satellite images vary from case to case. We could select a dataset in which the satellite images have dimensions 225x225, like the forest fire simulation dataset. Taking images of this dimension would guarantee that a decent portion of the fire is represented in the image. However, depending on the satellite dataset, it may be possible to reduce the dimension of the input (and output). Regarding the number of layers in the input (and output), the simulation dataset consists of 1 or 3 channels relatively if the input data are grayscale or RGB images. For satellite images, the number of bands (channels) can vary substantially, however, a general estimate is 5 or 7 bands. With 5 bands we speculate that the model may be able to achieve good performance with the initial computational complexity, however, with 7 bands it is possible that to achieve good performance the computational complexity needed would have to increase significantly. We can initially leave the CAE architecture as is but we may need to increase the complexity of the internal AE representation to allow for capturing the increased subtleties of the new data. However, once the satellite images dataset is available this change may be avoided if the dimension of the width and height of the images can be reduced.

Another server service used for this thesis was Azure, but they grant only VMs with the Nvidia Tesla K80 and the Nvidia Tesla M60 GPU with their sponsorship.

5.4.2 Caffe

Caffe is a strong tool for creating deep learning models. It allows for easy creation of new layers, modification of existing ones and it has a large community which makes it easier to find new presented layers. Moreover, it has a good Python interface, permitting the model to be run using Python and/or to create Python layers. However, using Caffe presented

some negative aspects, to cite a few it is not well documented, running a model on multiple GPU is highly complex. The following are two aspects of Caffe that penalized the time needed for the deployment of the model.

Setting up the environment for Caffe The installation guide presented in the Caffe website [34] [30] have been followed to set up Caffe and its dependencies. However, during installation, some problems had been encountered, which are mainly incompatibilities between version (or subversions) of the different libraries. Fortunately, both the Google Cloud Server and the Azure Server have VM images with their Caffe installed on it. Moreover, from the experience obtained during the several deployment of the CAE model, setting up Caffe by compiling the CMake build decreases the probability of encountering errors.

Debugging phase with Caffe The management of errors in Caffe unfortunately in some cases doesn't facilitate the debugging. In fact, some errors are not reported and just the layer affected is indicated. This is the case for the Python layer: to debug it the layer should be run separately as a Python unit test and then integrated to the model. Moreover, in some cases errors are not directly identified through highlights, bolding or changing of the fonts, which makes debugging more difficult.

Chapter 6

Conclusion and Future Work

6.1 Future works

6.1.1 Variation of the DFWP model

The DFWP has been designed so that new behaviors can be easily tested by making no or minimal changes to the model. In this subsection, some of the possible changes are presented.

Input sequences of different length The DFWP model described in the previous Section has as input sequences of fixed length. In fact, as described in Paragraph 4.2 the fire propagation sequences are divided into fixed sequences of 10 frames (9 frames as input sequence plus 1 as the label). However, because the model uses clip markers, the model can be easily adapted to manage variable sequence lengths. It is important to observe that this means that the numbers of frames in each sequence can change but not the dimensions of the frames, which have to be fixed. In this scenario the input sequences are divided into fixed sub-sequences as before, however, the new sequences are not overlapping but adjacent and also use padding images to fill the final frames. The padding image can be represented by zeros as suggested by [44] and have a 0 clip_marker assigned, the LSTM will easily learn to ignore them for the prediction. Finally, the model can be trained by calculating the loss for the 10th frame of each sub-sequence, as presented in 4.2, or only by analyzing the last frame of the entire sequence.

Returns kth timestamp frame The model can be adjusted to return the kth timestamp frame after the input sequence instead of the first consecutive frame. The problem in this scenario changes from many-to-1 to many-to-many. In other words, the LSTM will return a sequence of frames representing the development of the fire from the last input frame of the sequence to the successive k-frames. To make the change, an additional parameter should be sent from the input layer to the LSTM layer with the k value. Moreover, the scenario has an additional distinction to consider: if the model should return all the newly found frames or just the one relative to the k-timestamp. If the second option is selected the decoding portion of the model should not be modified. Otherwise, the new batch-size has to be modified to $(k \times sequences)$ for the decoder.

Use different images Eventually, the goal of this work is to operate on aerial or satellite images, so it would be interesting observing the model on satellite data. The preprocessing applied on the simulated data, described in Subsection 3.1.1, removed the stylized images of the trees to improve the results of the model. However, this operation made the simulated data more similar to a satellite image in which the precise shape of each object is not usually defined.

Additional information as input The idea of using images for prediction is given mainly for the lack of key information such as humidity, age, and the wind. Being able to obtain these data, or their effect, by using only satellite images would permit the use of the model in a broader spectrum of scenarios. However, in some cases some of the information may be available and using them would help to improve the precision of the model. This change can be done in two ways:

- Adding an additional channel in the input blob for each new type of information. For example, in the RGB scenario, if the humidity data is available, the input blob will have a fourth channel, having the same dimension of the other 3 channels. Similarly, the reconstruction layer will return a 4-channel blob. The advantage of this approach is that the changes are easy to implement, however, the disadvantages are that there is a high possibility of redundant information and that each new information would need a new layer, which would lead to a memory problem. Using the humidity example, the additional parameter can be a matrix considerably smaller than the size of the image. This would require further modification of the humidity matrix in order to become a suitable input blob by adding redundancy to the humidity matrix.
- Creating a parallel encoder for the additional information and then combining the feature extracted from them with the one obtained by DFWAE before sending the

complete information to LSTM. In a similar way, an additional decoder can be added to decode the additional information relative to the new frame. However, it may not be necessary if the only goal is to obtain the final image of the fire with no additional information. This approach would remove the problem of redundancy and memory shortage; however. the changes are not as linear and simple and would need deeper studies and design.

Other scenarios Following the idea of running the model using satellite images, it would be interesting to observe how it performs on images representing more complex wildfire scenario with obstacles such as bridges, streets, or cities. If enough training data are given the hypothesis is that the model should be able to learn the new behavior of the fire associated with the defined obstacle. As the DFWAE is able to extract a representation of the essential structure of the forest fire, because it uses convolution to do that, it should be able to extract also the key features of the structure of the new environment. Once the model is able to identify and extract the key features of the new environment, the next step would be predicting the features of the next timestamp of the fire spreading which is similarly addressed in the predictor part of this work.
It would also be interesting to observe the performance of the model with other natural disasters such as flooding. The flooding scenario has the similarity with the wildfire scenario that a body is spreading and if we consider a city fire instead of a forest fire, it has also the additional characteristic of the body damaging the object on which it comes into contact with, such as buildings collapsing. Moreover, in the flooding scenario the damages may not be caused just by the flooding per se but from the objects dragged by the current.

3D images This particular variation of the model would not be minimal but could result in significant performance enhancements. The model can be changed to be applied to a 3D reconstruction of forest wildfires. In other words, this new version of the model would consider also the heights of the forest, allowing to differentiate between surface fires and crown fires. Surface fires are low to high intensity fires, and even if they reach the canopy it is not to the extent that the fire can be carried by it. Crown fires occur in extreme intensity fires. In this scenario, the fire spreads throughout the canopy of the trees, and for this reason, they are usually consistently faster to spread than surface fires. This new version of the model would be also interesting when applied to fires having human-built obstacles, such as bridges or cities. Although the general architecture of the model would be valid for the new model; the individual layers have to be changed to be applicable to 3D images.

6.2 Conclusion

In this thesis, the DFWAE is presented. The model is able to extract from an image the key features representing the structure of the forest fire and reconstruct the original image from its features. Moreover, we presented an empirical validation of the good design for the DFWAE. In particular, we showed that the model is able to:

- recognize areas where there are burnt trees;
- recognize areas where there are the burning trees;
- in the areas of burning trees differentiate correctly the fire level in the areas of burning trees. Furthermore, even if ANN usually gives better performances on big datasets, the model showed good performance on a smaller dataset. This would indicate that the DFWP has a good depth: it is deep enough to be able to reconstruct successfully the forest wildfire image, and at the same time the number of layers is small enough to not cause vanishing gradients (5.1) when applied on a smaller dataset.

We also introduced the design of the DFWP, a prediction model for fire image sequences (in particular for sequences of 9 frames). In the design, we described the layers added to the DFWAE to make the predictor. We also showed the Python algorithm used to create the input subsequences from the dataset.
Besides introducing a new model, the goal of this thesis was to explain some of the design considerations relative to creating the DFWAE. In fact, in Deep Learning a wide set of choices is made empirically. However, there are general pointers that can guide what are the best practices, e.g. deciding the weight initialization based on the activation function and how to empirically find the best learning rate, and general methodologies that make debugging a model easier. To facilitate debugging, we also introduced a Python algorithm which permits the visualization of the output of each layer of the model. Moreover, as stated previously, Caffe is not well documented, even if it has an active forum. Thus, the decisions influenced by Caffe, e.g. the division of the dataset in 2 sets (3.3.1), and some of the problems and solution of using Caffe have been discussed.

References

[1] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv preprint arXiv:1511.00561*, 2015.

[2] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[3] Isaac Changhau. LSTM and GRU – Formula Summary. https://isaacchanghau.github.io/post/lstm-gru-formula/, July, 22 2017.

[4] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8609–8613. IEEE, 2013.

[5] David DiBiase et al. Nature of geographic information, 2008.

[6] Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[7] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2625–2634, 2015.

[8] Neerja Doshi. Deep Learning Best Practices(1) - Weight Initialization. https://medium.com/usf-msds/deep-learning-best-practices-1-weight-initialization-14e5c0295b94, 2018.

[9] Mark A Finney, Jack D Cohen, Sara S McAllister, and W Matt Jolly. On the need for a theory of wildland fire spread. *International journal of wildland fire*, 22(1):25–36, 2013.

[10] Sriram Ganapathi Subramanian. Reinforcement learning for determining spread dynamics of spatially spreading processes with emphasis on forest fires. Master's thesis, 2018.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[12] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[13] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[14] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, volume 2, pages 1735–1742. IEEE, 2006.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[18] Seunghoon Hong, Hyeonwoo Noh, and Bohyung Han. Decoupled deep neural network for semi-supervised semantic segmentation. In *Advances in neural information processing systems*, pages 1495–1503, 2015.

[19] HyeonwooNoh. GitHub repository: Caffe. https://github.com/HyeonwooNoh/caffe, 2015. Commit: d9e8494.

[20] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[21] Anil K Jain, Jianchang Mao, and K Moidin Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[23] Prateek Joshi. Understanding Xavier Initialization In Deep Neural Networks. https://prateekvjoshi.com/2016/03/29/understanding-xavier-initialization-in-deep-neural-networks/, March, 29 2016.

[24] Andrej Karpathy. CS231n: Convolutional Neural Networks for Visual Recognition - Course Materials. Stanford. https://cs231n.github.io.

[25] Huy Viet Le. Hand-and-finger-awareness for mobile touch interaction using deep learning.

[26] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[27] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning–ICANN 2011*, pages 52–59, 2011.

[28] Mahmoud S Nasr, Medhat AE Moustafa, Hamdy AE Seif, and Galal El Kobrosy. Application of Artificial Neural Network (ANN) for the prediction of EL-AGAMY wastewater treatment plant performance-EGYPT. *Alexandria Engineering Journal*, 51(1):37–43, 2012.

[29] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pages 1520–1528, 2015.

[30] NVIDIA. Download and install Caffe on NVIDIA GPUs. https://www.nvidia.in/object/caffe-installation-in.html. Accessed: 2019-01-22.

[31] NVIDIA. NVIDIA CUDA Installation Guide for Linux. https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#ubuntu-installation. Accessed: 2019-01-22.

[32] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pages 2863–2871, 2015.

[33] Christopher Olah. Understanding LSTM Networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, August, 27 2015.

[34] Berkeley Artificial Intelligence Research. Caffe - installation. http://caffe.berkeleyvision.org/installation.html. Accessed: 2019-01-22.

[35] Berkeley Artificial Intelligence Research. Caffe tutorial - Blobs, Layers, and Nets: anatomy of a Caffe model. http://caffe.berkeleyvision.org/tutorial/net_layer_blob.html. Accessed: 2019-01-22.

[36] Berkeley Artificial Intelligence Research. Caffe tutorial - Convolution Layer. http://caffe.berkeleyvision.org/tutorial/layers/convolution.html. Accessed: 2019-01-22.

[37] Berkeley Artificial Intelligence Research. Caffe tutorial - Loss Function. http://caffe.berkeleyvision.org/tutorial/loss.html. Accessed: 2019-01-22.

[38] Berkeley Artificial Intelligence Research. Caffe tutorial - Solver. http://caffe.berkeleyvision.org/tutorial/solver.html. Accessed: 2019-01-22.

[39] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 833–840. Omnipress, 2011.

[40] Andres Rodriguez. Training and deploying deep learning networks with Caffe. http://rodriguezandres.github.io/2016/04/28/caffe, April, 28 2016.

[41] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[42] Volodymyr Turchenko, Eric Chalmers, and Artur Luczak. A Deep Convolutional Auto-Encoder with Pooling-Unpooling Layers in Caffe. *arXiv preprint arXiv:1701.04949*, 2017.

[43] Volodymyr Turchenko and Artur Luczak. Creation of a deep convolutional auto-encoder in caffe. *arXiv preprint arXiv:1512.01596*, 2015.

[44] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015.

[45] Berkeley Vision and Learning Center. Wiki documentation - Development. https://github.com/BVLC/caffe/wiki/Development, February, 9 2016.

[46] Berkeley Vision and Learning Center. GitHub repository: Caffe. https://github.com/BVLC/caffe, 2018. Commit: 99bd997.

[47] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[48] Chenyu Wang. Fine grained video classification for endangered bird species protection.

[49] SHI Xingjian, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wangchun Woo. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems*, pages 802–810, 2015.

[50] Matthew D Zeiler, Graham W Taylor, and Rob Fergus. Adaptive deconvolutional networks for mid and high level feature learning. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2018–2025. IEEE, 2011.

[51] Yi Zhao, Jiale Ma, Xiaohui Li, and Jie Zhang. Saliency detection and deep learning-based wildfire identification in uav imagery. *Sensors*, 18(3):712, 2018.

[52] Zhong Zheng, Wei Huang, Songnian Li, and Yongnian Zeng. Forest fire spread simulating model using cellular automaton with extreme learning machine. *Ecological Modelling*, 348:33–43, 2017.