International Journal of Research in Engineering, Science and Management
Volume-3, Issue-3, March-2020
www.ijresm.com | ISSN (Online): 2581-5792

# Forest Fire Smoke Recognition and Temperature Prediction using Data Science

J. P. Deepa Lakshmi[1], A. Shivani[2], R. Swetha[3], V. Anitha Moses[4]
[1,2,3]*Student, Department of Computer Science and Engineering, Panimalar Engineering College, Chennai, India*
[4]*Assistant Professor, Department of Computer Science and Engineering, Panimalar Engineering College, Chennai, India*

***Abstract*: Detection forest fire smoke during the initial stage is vital for preventing forest for events. Recent studies have been shown that exploring fire can be detected by using the video based real capturing and monitoring real time using the camera (K210) image processing and predicting the fire based on the trained network (CNNs) by the temperature sensor if the temperature falls high than usual then it is analyzed that there is a chance of fire so the alert message or the notification is sent to the forest department.**

***Keywords*: Fire detection, Notification, Temperature measuring.**

## 1. Introduction

Smoke detection is done with the real time video based camera which will divide the video into frames and those frames are processed as image and compared with the database which is trained with certain images are differentiated with fire and non-fire in forest and trees and it will be. This paper is will detect the fire and will send the notification to the forest department. This can also predict the temperature using the sensor if the temperature goes higher than the certain rate according to the history of forest fire records it will intimate that there is a chance for fire. So this paper gives accurately 90% of exact result. In this we used a recent technology to use both Artificial Intelligence and image processing.

Deep learning is also known as deep structured learning. It is a subset of machine learning where artificial neural network algorithms inspired by the human brain. It is similar to how we learn from experience. Learning can be supervised, unsupervised and semi supervised. Deep learning architectures such as deep neural network, deep belief network, recurrent neural networks and convolutional neural network have been applied to the fields including computer vision, speech recognition, social network filtering and so on.

### *A. MNIST datasets*

The MNIST Datasets consists of handwritten digit image and it is divided in 60,000 examples for the training sets 10,000 examples for testing. In many papers as well as in this tutorial, the official training set of 60,000 is divided into an actual training set of 50,000 examples and 10,000 validation examples (for selecting hyper-parameters like learning rate and size of the model). All digit images have been size- normalized and centered in a fixed size image of 28 x 28 pixels. In the original dataset each pixel of the image is represented by a value between 0 and 255, where 0 is black, 255 is white and anything in between is a different shade of grey.

### *B. Notation*

Dataset notation We label data sets as D. When the distinction is important, we indicate train, validation, and test sets as: Dtrain, Dvalid and Dtest. The validation set is used to perform model selection and hyper-parameter selection, whereas the test set is used to evaluate the final generalization error and compare different algorithms in an unbiased way.

Embedded system is a special purpose computer controlled electro-mechanical system in which the computer is completely encapsulated by the device it controls. An embedded system has specific requirements and performs pre- defined tasks, unlike a general-purpose personal computer. An embedded system is a computer- controlled system.

The core of any embedded system is a microprocessor, programmed to perform a few tasks (often just one task). This is to be compared to other computer systems with general purpose hardware and externally loaded software loaded software. Embedded systems are often designed for mass production.

### *C. Characteristics*

Embedded systems are computer systems in the widest sense. They include all computers. Most commercial embedded systems are designed to do some task at a low cost. Most, but not all have real- time system constraints that must be met. They may need to be very fast for some functions, but most other functions will probably not need speed. These systems meet their real- time constraints with a combination of special purpose hardware and software tailored to the system requirements. It is difficult to characterize embedded systems by speed or cost, but for high volume systems, cost usually dominates the system design. Often many parts of an embedded system need low performance compared to the primary mission of the system. This allows an embedded system to be intentionally simplified to lower costs compared to a general-purpose computer accomplishing the same task, by using a CPU that is just "good enough" for these secondary functions. Embedded systems reside in machines that are expected to run continuously for years without errors. Therefore, the software is usually developed and tested more carefully than software for personal computers. Many embedded systems avoid mechanical moving parts such as disk drives, switches or buttons because these are unreliable compared to solid- state parts such as Flash memory.

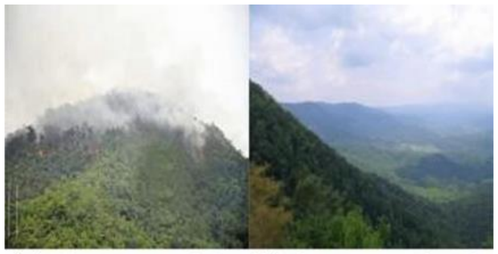

(a) A forest image with fire,
(b) A forest image without fire

Fig. 1. Image with fire and non-fire in camera

In addition, the embedded systems may be outside the reach of humans, so the embedded system must be able to restart itself even if catastrophic data corruption has taken place. This is usually accomplished with a standard electronic part called a watchdog timer that resets the computer unless the software periodically resets the timer.

## D. *Tensor flow*

Tensor flow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library and is also used for machine learning applications such as neural networks. It is used for both research and production at google. The tensor flow was written in python, c++. It works on the platform like linux, macos, windows, android.

Tensor Data Structure Tensors are used as the basic data structures in Tensor Flow language. Tensors represent the connecting edges in any flow diagram called the Data Flow Graph. Tensors are defined as multidimensional array or list.

## E. *Embedded System*

### 1) *Autonomous*

Autonomous Systems function in standalone mode. Many embedded systems used for process control in manufacturing units and automobiles fall under this category. In process control systems the inputs originated from transducers that convert a physical quantity, such as temperature into an electrical signal. The system's output controls the device. In standalone systems, the deadlines or response times are not critical. An air-conditioner can be set to turn on when the temperature reaches a certain level, measuring instruments and CD players are examples of Autonomous Systems.

### 2) *Real-Time*

Real-Time embedded systems are required to carry out specific tasks in a specified amount of time. These systems are extensively used to carry out time-critical task in process-control. For instance, a boiler plant must open the valves if the pressure exceeds a particular threshold. If the job is not carried out in the stipulated time, a catastrophe may result.

### 3) *Networked*

Networked embedded systems monitors plant parameters, such as temperature, pressure and humidity will send the data over the network to a centralized system for online monitoring. A networked-enabled web camera monitoring the plant floor transmits its video output to a remote controlling organization.

### 4) *Mobile*

Mobile gadgets need to store databases locally in their memory. These gadgets have powerful computing and communication capabilities to perform Real-Time as well as non-real-time tasks and handle multimedia applications. The gadgets embedded powerful processor and OS, and a lot of memory with minimal power consumption.

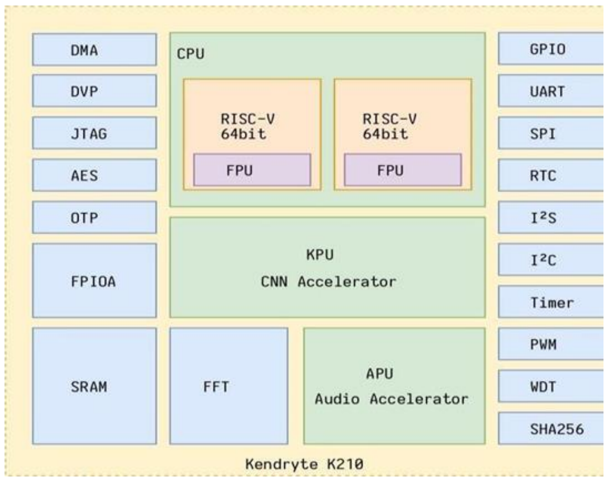


Fig. 2. Core of kendryte

## F. *Deep Learning*

### 1) *Classifying MNIST digits using logistic regression*

Logistic regression is a probabilistic, linear classifier. It is parametrized by a weight matrix W and a bias vector b. classification is done by projecting an input vector onto a set of hyperplanes each of which corresponds to a word. The distance from the input to a hyper plane reflects the probability that the input is a member of the corresponding class.

Mathematically, the probability that an input vector is a member of a class,a value of a stochastic variable Y,can be written as:

P(Y = i|x,W,b) = softmaxi(Wx + b)= eWix+bi Pj eWjx+bj Defining a loss function.

Learning optimal model parameters involves minimizing a loss function. In the case of multi-class logistic regression, it is very common to use the negative log-likelihood as the loss. This is equivalent to maximizing the likelihood of the data set D under the model parameterized by θ.

### 2) *Multilayer perceptron*

The next architecture we are going to present using Theano is the single-hidden-layer Multi-Layer Perceptron (MLP). An MLP can be viewed as a logistic regression classifier where the input is first transformed using a learnt non-linear transformation Φ.

This transformation projects the input data into a space where it becomes linearly separable. This intermediate layer is referred to as a hidden layer. A single hidden layer is sufficient to make MLPas universal approximator. However, we will see later on that there are substantial benefits to using many such hidden layers, i.e. the very premise of deep learning.

*The model:*

An MLP (or Artificial Neural Network - ANN) with a single hidden layer can be represented.

Formally, a one-hidden-layer MLP is a function f: RD → RL, where D is the size of input vector x and L 35

Deep Learning Tutorial, Release 0.1 is the size of the output vector f(x), such that, in matrix notation:

f(x) = G(b(2) + W(2)(s(b(1) + W(1)x))),with bias vectors b(1), b(2); weight matrices W(1), W(2) and activation functions G and s. The vector h(x) = Φ(x) = s(b(1) + W(1)x) constitutes the hidden layer.

*3) Convolutional neural networks (LENET)*

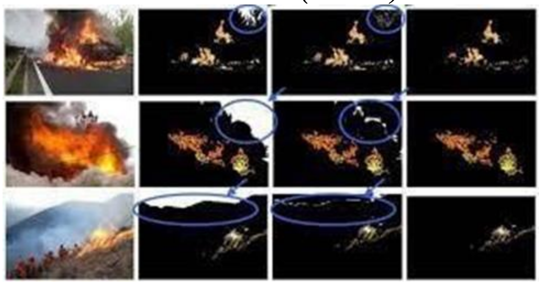


Fig. 3. Grey scale image in deep learning

Convolutional Neural Networks (CNN) are biologically-inspired variants of MLPs. we know the visual cortex contains a complex arrangement of cells. These cells are sensitive to small sub-regions of the visual field, called a receptive field. The sub-regions are tiled to cover the entire visual field. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images. Additionally, two basic cell types have been identified: Simple cells respond maximally to specific edge-like patterns within their receptive field. Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern.

*4) Sparse connectivity*

CNNs exploit spatially-local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. In other words, the inputs of hidden units in layer m are from a subset of units in layer m-1, units that have spatially contiguous receptive fields. We can illustrate this graphically as follows:

Imagine that layer m-1is the input retina. Units in layer m have receptive fields of width 3 in the input retina and are thus only connected to 3 adjacent neurons in the retina layer. Units in layer m+1 have a similar connectivity with the layer below. We say that their receptive field with respect to the layer below is also 3, but their receptive field with respect to the input is larger.

Each unit is unresponsive to variations outside of its receptive field with respect to the retina. The architecture thus ensures that the learnt "filters" produce the strongest response to a spatially local input pattern.

*5) Max pooling*

Another important concept of CNNs is max-pooling, which is a form of non-linear down-sampling. Max- pooling partitions the input image into a set of non- overlapping rectangles and, for each such sub-region, outputs the maximum value. Max-pooling is useful in vision for two reasons:

1. By eliminating non-maximal values, it reduces computation for upper layers.
2. It provides a form of translation invariance. Imagine cascading a max-pooling layer with a convolutional layer.

There are 8 directions in which one can translate the input image by a single pixel. If max-pooling is done over a 2x2 region, 3 out of these 8 possible configurations will produce exactly the same output at the convolutional layer.

*The Full Model: LeNet*

Sparse, convolutional layers and max-pooling are at the heart of the LeNet family of models. While the exact details of the model will vary greatly, the figure below shows a graphical depiction of a LeNet model.

The lower-layers are composed to alternating convolution and max-pooling layers. The upper-layers however are fully-connected and correspond to a traditional MLP (hidden layer + logistic regression). The input to the first fully-connected layer is the set of all features maps at the layer below. From an implementation point of view, this means lower-layers operate on 4D tensors. These are then flattened to a 2Dmatrix of rasterized feature maps, to be compatible with our previous MLP implementation.

*6) Denoising Autoencoders (DA)*

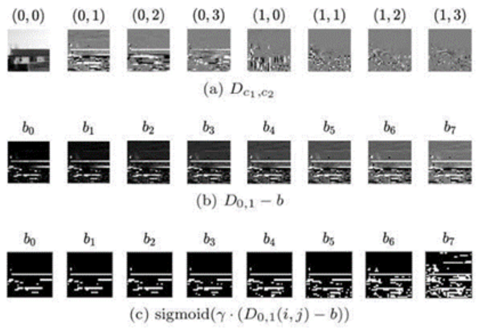
An autoencoder takes an input x ∈ [0,1]d and first maps it (with an encoder) to a hidden representation y ∈ [0,1]d0 through a deterministic mapping, e.g.:

$$y = s(Wx+b)$$

Where s is a non-linearity such as the sigmoid. The latent representation y, or code is then mapped back (with a decoder) into a reconstruction z of the same shape as x. The mapping happens through a similar transformation, e.g.:

z = s(W0y+b0) Denoising Autoencoders

The idea behind denoising auto encoders is simple. In order to force the hidden layer to discover more robust features and prevent it from simply learning the identity, we train the auto encoder to reconstruct the input from a corrupted version of it. The denoising auto-encoder is as to chastic version of the auto-encoder. Intuitively, a denoising auto-encoder does two things: try to encode the input (preserve the information about the input), and try to bound other effect of a corruption process stochastically applied to the input of the auto-encoder. The latter can only be done by capturing the statistical dependencies between the inputs. The denoising auto-encoder can be understood from different perspectives (the manifold learning perspective, stochastic operator perspective, bottom- up – information theoretic perspective, top-down – generative model perspective), all of which are explained for an overview of auto-encoders. The stochastic corruption process randomly sets some of the inputs (as many as half of them) to zero. Hence the denoising auto-encoder is trying to predict the corrupted (i.e. missing) values from the uncorrupted (i.e., non- missing) values, for randomly selected subsets of missing patterns. Note how being able to predict any subset of variables from the rest is a sufficient condition for completely capturing the joint distribution between a set of variables (this is how Gibbs sampling works). To convert the auto encoder class into a denoising auto encoder class, all we need to do is to add a stochastic corruption step operating on the input. The input can be corrupted in many ways, but in this tutorial we will stick to the original corruption mechanism of randomly masking entries of the input by making them zero.


(0, 0)
(0, 1)
(0, 2)
(0, 3)
(1, 0)
(1, 1)
(1, 2)
(1, 3)
(a) $D_{c_1,c_2}$
$b_0$ $b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$
(b) $D_{0,1} - b$
$b_0$ $b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$
(c) sigmoid($\gamma \cdot (D_{0,1}(i,j) - b)$)
Fig. 4. Linear matric of grey scale images

## G. *Stacked Denoising Encoders (SDA)*

### 1) *Stacked Auto encoders*

Denoising auto encoders can be stacked to form a deep network by feeding the latent representation (output code) of the denoising auto encoder found on the layer below as input to the current layer. The unsupervised pre-training of such an architecture is done one layer at a time. Each layer is trained as a denoising auto encoder by minimizing the error in reconstructing its input (which is the output code of the previous layer). Once the first k layers are trained, we can train the k +1-the layer because we can now compute the code or latent representation from the layer below. Once all layers are pre-trained, the network goes through a second stage of training called fine-tuning. Here we consider supervised fine- tuning where we want to minimize prediction error on a supervised task. For this, we first add a logistic regression layer on top of the network (more precisely on the output code of the output layer). We then train the entire network as we would train a multilayer perceptron. At this point, we only consider the encoding parts of each auto-encoder. This stage is supervised, since now we use the target class during training. This can be easily implemented in Theano, using the class defined previously for a denoising auto encoder. We can see the stacked denoising auto encoder as having two facades: a list of autoencoders, and an MLP. During pre-training we use the first facade, i.e., we treat our model as a list of auto encoders, and train each auto encoder seperately. In the second stage of training, we use the second facade. These two facades are linked because:

- The auto encoders and the sigmoid layers of the MLP share parameters, and
- The latent representations computed by intermediate layers of the MLP are fed as input to the auto encoders.

### 2) *Restricted Bolzmann Machine (RBM)*

#### 1. *Energy-Based Models (EBM)*

Energy-based models associate a scalar energy to each configuration of the variables of interest. Learning corresponds to modifying that energy function so that its shape has desirable properties. For example, we would like plausible or desirable configurations to have low energy. Energy-based probabilistic models define a probability distribution through an energy function, as follows: p(x) = e−E(x) Z.

The normalizing factor Z is called the partition function by analogy with physical systems.

*Restricted Boltzmann Machines (RBM)*

Boltzmann Machines (BMs) are a particular form of log-linear Markov Random Field (MRF), i.e., for which the energy function is linear in its free parameters. To make them powerful enough to represent complicated distributions (i.e., go from the limited parametric setting to a non-parametric one), we consider that some of the variables are never observed (they are called hidden). By having more hidden variables (also called hidden units), we can increase the modeling capacity of the Boltzmann Machine (BM). Restricted Boltzmann Machines further restrict BMs to those without visible-visible and hidden-hidden connections. A graphical depiction of an RBM is shown below.

The energy function E(v,h) of an RBM is defined as: E(v,h) =−b0v−c0h−h0Wv where W represents the weights connecting hidden and visible units and b, c are the offsets of the visible and hidden layers respectively.

RBMs with binary units In the commonly studied case of using binary units (where vj and hi ∈ {0,1})

*Sampling in an RBM*

Samples of p(x) can be obtained by running a Markov chain to convergence, using Gibbs sampling as the transition operator. Gibbs sampling of the joint of N random variables S = (S1,...,SN) is done through a sequence of N sampling sub-steps of the form Si ∼ p(Si|S−i) where S−i contains the N −1 other random variables in S excluding Si. For RBMs, S consists of the set of visible and hidden units. However, since they are conditionally independent, one can perform block Gibbs sampling. In this setting, visible units are sampled simultaneously given fixed values of the hidden units. Similarly, hidden units are sampled simultaneously given the visible.

*3) Deep belief networks*

RBM scan be stacked and trained in a greedy manner to form so called Deep Belief Networks (DBN). DBNs are graphical models which learn to extract a deep hierarchical representation of the training data. They model the joint distribution between observed vector x and the hidden layers hk as follows: P(x, h1,...,h`) = `−2 Y k=0 P(hk|hk+1)!P(h`−1,h`)

The principle of greedy layer-wise unsupervised training can be applied to DBNs with RBMs as,

The building blocks for each layer The process is as follows: 1. Train the first layer as an RBM that models the raw input x = h(0) as its visible layer. 2. Use that first layer to obtain a representation of the input that will be used as data for these condlayer. Two common solutions exist. This representation can be chosen as being the mean activations p(h(1) = 1|h(0)) or samples of p(h(1)|h(0)). 3. Train the second layer as an RBM, taking the transformed data (samples or mean activations) as training examples (for the visible layer of that RBM). 4. Iterate (2 and 3) for the desired number of layers, each time propagating upward either samples or mean values.

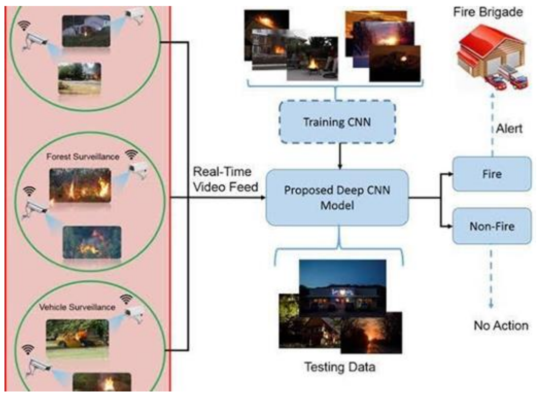


Fig. 5. Deep belief networks

*4) Hybrid monte-carlo sampling*

In HMC, model samples are obtained by simulating a physical system, where particles move about a high-dimensional landscape, subject to potential and kinetic energies. Adapting the notation from [Neal 93], particles are characterized by a position vector or state s ∈RD and velocity vector φ ∈RD. The combined state of a particle is denoted as χ = (s,φ). The Hamiltonian is then defined as the sum of potential energy E(s) (same energy function defined by energy- based models) and kinetic energy K(φ), as follows: H(s,φ) = E(s)+ K(φ) = E(s)+ 1 2X iφ2i

HMC Algorithm in this tutorial, we obtain a new HMC sample as follows:

1. Sample a new velocity from a univariate Gaussian distribution.
2. Perform n leapfrog steps to obtain the new state χ0
3. Perform accept/reject move of χ0

HMC_sampler We finally tie everything together using the HMC_Sampler class. Its main elements are: new_from_shared_positions: a constructor method which allocates various shared variables and strings together the calls to hmc_move and hmc_updates. It also builds the theano function simulate, whose sole purpose is to execute the updates generated by hmc_updates.

draw: a convenience method which calls the Theano function simulate and returns a copy of the contents of the shared variable self-positions.

*5) Recurrent neural network model*

*Raw input encoding:*

A token corresponds to a word. Each token in the ATIS vocabulary is associated to an index. Each sentence is a array of indexes (int32).

Then, each set (train, valid, test) is a list of arrays of indexes. A python dictionary is defined for mapping the space of indexes to the space of words.

*Context window*

Given a sentence i.e. an array of indexes, and a window size i.e. 1,3,5,..., we need to convert each word in the sentence to a context window surrounding this particular word. In details, we have:

*Word embeddings:*

Once we have the sentence converted to context windows i.e. a matrix of indexes, we have to associate these indexes to the embeddings (real-valued vector associated to each word).

*Elman recurrent neural network:*

The following (Elman) recurrent neural network (E-RNN) takes as input the current input (time t) and the previous hidden t state (time t-1). Then it iterates. In the previous section, we processed the input to fit this sequential/temporal structure. It consists in a matrix where the row 0 corresponds to the time step t=0, the row 1 corresponds to the time step t=1, etc. The parameters of the E-RNN to be learned are:

- the word embeddings (real-valued matrix)
- the initial hidden state (real-value vector)
- two matrices for the linear projection of the input t and the previous hidden layer state t-1 (optional) bias. Recommendation: don't use it.
- softmax classification layer on top The hyper parameters define the whole architecture:
    - dimension of the word embedding
- size of the vocabulary
    - number of hidden units
    - number of classes
- random seed + way to initialize the model

## H. Tensor flow

*1) Tensor flow-line regression*

Linear regression implementation using tensor flow. Logistic regression or linear regression is a supervised machine learning approach for the classification of order discrete categories. Our goal in this chapter is to build a model by which a user can predict the relationship between predictor variables and one or more independent variables.

The relationship between these two variables is considered linear. If y is the dependent variable and x is considered as the independent variable, then the linear regression relationship of two variables will look like the following equation:

$$Y= Ax+b$$

We will design an algorithm for linear regression. This will allow us to understand the following two important concepts:

- Cost function
- Gradient descent algorithms

### *I. Tensorflow object detection*

TensorFlow includes a special feature of object detection and these objects or images are stored in a specific folder. With relatively same object, it will be easy to implement this logic for security purposes.

The folder structure of object detection code implementation is as shown below:

The dataset_object includes the related images, which need to be loaded. We will focus on object detection with our logo defined in it. The objects are loaded with "load_data.py" script, which helps in keeping a note on various object detection modules within them.

## 2. Conclusion

This paper presented an overview on forest fire smoke recognition and temperature prediction using data science.

## References

[1] H. Tian, W. Li, P. O. Ogunbona, and L. Wang, "Detection and Separation of Smoke from Single Image Frames," *IEEE Transactions on Image Processing*, vol. PP, no. 99, pp. 1–1, 2018.
[2] Z. Yin, B. Wan, F. Yuan, X. Xia, and J. Shi, "A Deep Normalization and Convolutional Neural Network for Image Smoke Detection," *IEEE Access*, vol. 5, pp. 18429–18438, 2017.
[3] F. Yuan, X. Xia, J. Shi, H. Li, and G. Li, "Non-Linear Dimensionality Reduction and Gaussian Process Based Classification Method for Smoke Detection," *IEEE Access*, vol. 5, no. 99, pp. 6833–6841, 2017.
[4] K. Dimitropoulos, P. Barmpoutis, and N. Grammalidis, "Higher Order Linear Dynamical Systems for Smoke Detection in Video Surveillance Applications," *IEEE Transactions on Circuits & Systems for Video Technology*, vol. PP, no. 99, pp. 1–1, 2017.
[5] G. Lin, Y. Zhang, G. Xu, and Q. Zhang, "Smoke Detection on Video Sequences Using 3D Convolutional Neural Networks," *Fire Technol*, Feb. 2019.
[6] A. E. Çetin *et al.*, "Video fire detection – Review," *Digital Signal Processing*, vol. 23, no. 6, pp. 1827–1843, 2013.
[7] J. A. Ojo and J. A. Oladosu, "Video-based Smoke Detection Algorithms: A Chronological Survey," *Computer Engineering and Intelligent Systems*, vol. 5, no. 7, pp. 38–50, 2014.
[8] C. Long *et al.*, "Transmission: A New Feature for Computer Vision Based Smoke Detection," in *Artificial Intelligence and Computational Intelligence*, vol. 6319.
[9] F. L. Wang, H. Deng, Y. Gao, and J. Lei, F. N. Yuan, "A double mapping framework for extraction of shape- invariant features based on multi-scale partitions with AdaBoost for video smoke detection," *Pattern Recognition*, vol. 45, no. 12, pp. 4326– 4336, Dec. 2012.
[10] H. Tian, W. Li, L. Wang, and P. Ogunbona, "Smoke Detection in Video: An Image Separation Approach," *International Journal of Computer Vision*, vol. 106, no. 2, pp. 192–209, Jan. 2014.