university of groningen / faculty of science and engineering

Isolating Wildfires using a Convolutional Neural Network based Multi-Agent System

Bachelor's Project Thesis

Niels Rocholl, S3501108, n.m.m.rocholl@student.rug.nl,
Supervisor: dr. M.A. Wiering

Abstract: This study tests the ability of a convolutional neural network (ConvNet) based multi-agent system (MAS) to isolate wildfires in a simulation. The ConvNet is designed to map the environmental input to useful moves. The MAS consists of either two or four agents. All systems are tested in grid environments of size 21×21, 41×41, and 61×61. All environments are simulated with a **normal** and a **fast** fire propagation speed. Furthermore, a single agent system is implemented for the purpose of comparison. The results show that the four agent system outperforms the single-agent system in all environments. Moreover, the four agent system is able to isolate more than 99% of fires in all environments except one. In the 61×61 environment with a **fast** fire propagation speed, the four agent system is able to isolate 90.5% of the fires on average.

1 Introduction

1.1 Wildfires

Wildfires are a vital natural process in our environment. However, in recent years the frequency and scale of wildfires have been increasing at an alarming rate. Every year wildfires burn millions of acres and destroy thousands of homes, leaving many people without a roof over their heads. Due to the advent of global warming, we are now facing the reality that these fires will only become worse in the foreseeable future.

Fighting wildfire can be extremely challenging. To effectively fight wildfire, factors such as weather, fuel, and topography need to be analyzed. Moreover, it is of great importance that firefighters keep track of multiple dimensions at all times, as the fire can spread through trees, soil, or even through the air when embers are carried by the wind. Fire managers use this information to determine the behavior of a wildfire. However, when the fire becomes sufficiently complex, it is almost impossible for a human to devise an effective plan to fight the fire. For this reason, this study will focus on the application of machine learning to the problem of wildfire control.

In order to put out a fire, one needs to remove oxygen, heat, or fuel. At the moment, the two most widely used fire fighting techniques focus on removing oxygen by putting water, dirt, or retardant on the fire or removing fuel by digging firebreaks. This project will focus on the latter technique. A firebreak is a gap in vegetation or other combustible material that acts as a barrier to stop a wildfire's progress. However, doing this effectively is easier said than done. Many factors need to be considered to dig effective firebreaks, such as fire propagation speed, distance to the fire, and wind direction. Furthermore, it can be a hazardous job since one could be engulfed in flames within seconds if the wind picks up.

There have been several endeavours in which the application of machine learning on the problem of wildfire control is researched. One of these is enforced subpopulations (Wiering, Mignogna, and Maassen, 2005). In this approach enforced subpopulations are used to evolve neural network controllers. The system works by generating subgoals and assigning them to different agents. In this system, the task assignment module and subgoal generator are modeled as multi-layer perceptrons, which are evolved to minimize the damage caused by the spreading wildfire.

Another approach focused on the application of reinforcement learning (Wiering and Doringo, 1998). In this approach reinforcement learning (RL) is used to learn reactive policies that map an agent's input to the appropriate action. RL algorithms learn from trial and error in order to predict the total expected cost of a unique situation. The policy can then use the predicted costs to select an action that minimizes the expected future cost.

Yet another approach focussed on a combination of RL and learning from demonstration (LfD) (Hammond, Schaap, Sabatelli, and Wiering, 2020). In this approach four model-free RL algorithms are combined with a multi layer perception that serves as a value function approximator. Furthermore, the researchers use demonstration data that is inserted in an experience-replay memory buffer before learning to enable the algorithms to better cope with the difficulty to contain the forest fires when they start learning.

1.2 Background

This study is inspired by previous research that used LfD and ConvNets to isolate wildfires in a simulation (Ywema and Wiering, 2020). This approach showed promising results, however the system in this study consisted of only a single agent. A multi-agent system might be better suited to the problem of wildfire isolation. For that reason, this study will present a new approach that combines a convolutional neural network (ConvNet) with a multi-agent system. Furthermore, a single agent system will also be implemented and tested for the purpose of comparison.

Multi-Agent Systems

A multi-agent system (MAS) is a system that is designed to deal with the behavior of multiple computing entities called "agents." In a MAS, the agents work together to solve a problem that would be beyond the abilities of a single agent. Multi agent systems have been proven to be useful in multiple fields such as electronic business, the semantic web, and grid computing (Oprea, 2004)

Convolutional Neural Networks

The field of deep learning has been witness to several great successes over the past years. Many of the advances in the field have been made possible through the application of a particular algorithm, the convolutional neural network (ConvNet). A ConvNet is a deep learning algorithm similar to an ordinary neural network (NN). ConvNets have enabled many outstanding achievements within the field of artificial intelligence, with LeCun being one of the earliest adopters. He showed that ConvNets could classify handwritten digits (LeCun, Boser, Denker, Henderson, Howard, Hubbard, and Jackel, 1989). In 2016, almost 27 years after LeCun's work, the team at Google's Deepmind showed that it was possible to master the game of Go by leveraging the power of ConvNets (Silver et al., 2016).

A ConvNet, just like a normal NN, is made up of learnable weights and biases. However, unlike a NN, a ConvNet has its neurons arranged in three dimensions (width, height, depth). A ConvNet can capture the spatial dependencies within the input matrix through the use of so-called **filters**. Together, these characteristics allow for the classification of complex input matrices that would not be feasible with ordinary NNs.

1.3 Research Question

This study aims to determine if a multi-agent system, which utilizes a ConvNet for action selection, can effectively isolate wildfires in a simulated environment. The reason for choosing a ConvNet is its ability to capture the spatial information from the simulation, which will be used by the agents to make predictions on whether to dig a firebreak or move in a certain direction. The reason for using a multi-agent system is because of its ability to handle complex environments. Moreover, another advantage of using multiple agents is that the system can tolerate failures by one or more agents, since other agents can make up for their mistake. In other words, a multi-agent system has the benefit of robustness. Taken together, these qualities make a multi-agent system better suited to this problem than a single agent system. The research question to be answered in this study is:
"Can a multi-agent system which utilizes a convolutional neural network for action selection effectively be used to isolate wildfires in a simulation?"
Furthermore, this paper will study the effect of including the agent's positional history in the input for the convolutional neural network. This will be

done by storing the previous positions of an agent in one of the ConvNets input layers. A similar technique was successfully used in previous research to increase the accuracy of a ConvNet (Wagenaar, Okafor, Frencken, and Wiering, 2017). In this research the input for the ConvNet was enhanced by including so-called **object trails**, which indicate previous soccer positions.

2 The Wildfire Simulation

This section will discuss how the environment for the wildfire simulation was implemented and how it was represented.

2.1 Glossary

Active agent: The agent which is currently taking an action.
Inactive agent: The agent which is currently not taking an action because it is waiting for its turn.
FPS: Fire propagation speed

2.2 Environment

The environment that was used in this study represents a simulated top-down view of a forest, such as shown in figure 2.1. It is a grid structure consisting of cells, with a length N and width N, adding up to a total amount of $N \times N$ cells.

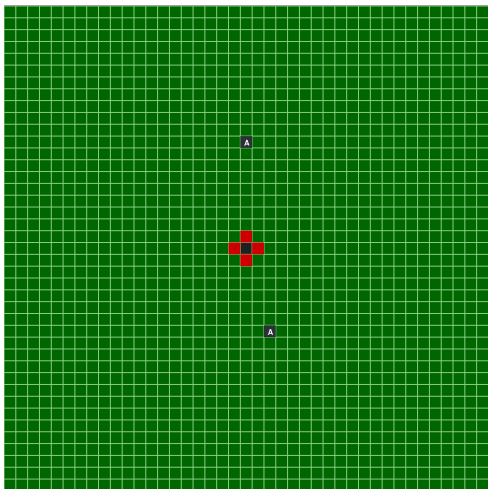


Figure 2.1: Snapshot of the 41×41 environment containing two agents (grey with "A" inside), fire (red), trees (green) and burnt (black)

2.2.1 Cells

The cells of the environment can be classified as one of the following five types:

- *Agent*
- *Tree*
- *Dirt*
- *Burnt*
- *Fire*

In addition, every cell has four attributes:

- *Fuel*
- *Temperature*
- *Ignition threshold*
- *Heat potential*

The type of every cell is visible to the agents at all times. The attributes however, are not visible to the agents. The attribute *fuel* determines how long a cell can burn. The *temperature* of a cell determines whether it should ignite, which will happen once the temperature passes the *ignition threshold.* Lastly, the attribute *heat potential* determines how much heat is applied to surrounding cells.

2.2.2 Agents

The simulation can be run with a single agent or multiple agents. At the start of the simulation the agent(s) will be positioned at a random location. To prevent the agent from being positioned outside the map, the starting location of the agent has to adhere to a maximum radius of:

$$max(r) = \frac{N}{2} - 1 \qquad (2.1)$$

In which r is the radius and N is the map size. An agent can choose between six different actions:

- *Move Up*
- *Move Down*
- *Move Left*
- *Move Right*
- *Dig*
- *Wait*

An agent is modeled by a ConvNet, which takes as input the current state of the environment and outputs a class score for every possible action. The agent then selects the action with the highest class score. Every action takes one time-step to execute. However, certain restrictions apply to the action an agent may take given a particular state. The agent cannot traverse fire. If it tries to do so, the action is canceled, and the agent's position will not change. This can be seen as the agent having a heat sensor that prevents it from moving into the fire. Furthermore, the fire cannot spread to an agent's location.

2.2.3 Fire

The fire always starts in the middle of the map. It only knows four directions of movement, namely horizontal and vertical movement, meaning that it is not allowed to move diagonally. The fire spreads by igniting all neighboring tree cells. Every time-step the fire looks for a possible spreading path through the use of the Astar algorithm (Hart, Nilsson, and Raphael, 1968). If the fire does not find a spreading path, then it has been isolated and the simulation is terminated. If the fire reaches the border of the map, then the fire is out of control and the simulation is also terminated. For the distance heuristic, the simulation uses Manhattan distance (L_1). Lastly, the fire can be simulated with a **normal** fire propagation speed and a **fast** fire propagation speed (FPS). The **normal** FPS ignites all burnable neighboring cells after 38 time-steps. The **fast** propagation speed is double as fast as the **normal** propagation speed, lighting all burnable neighboring cells after 19 time-steps.

2.3 Representation

The ConvNet is designed to receive an abstract representation of the environment's current state and output a class score for all actions. In this study, the input data for the ConvNet was relatively simple, making it a perfect candidate for the use of binary vision grids, which have been proven to be useful in the game of Tron (Knegt, Drugan, and Wiering, 2018). Vision grids have been shown to enhance performance and increase the learning speed of the system. A vision grid shows part of the environment from the agent's perspective. For this study, multiple vision grids were used to represent a single state. Each vision grid only holds binary values. These values represent essential information in the environment, like the agent's location or the location of the fire. The size of the vision grids is the same size as the environment, meaning that the agent can observe the whole environment.

The input data for the ConvNet contains eleven vision grids, which can be configured in seven different ways in order to test the effects of including 0, 3, 5, or all previous agent positions in the input. The eleven different vision grids are:

1. *Current position of the active agent*
2. *Current position of the inactive agent(s)*
3. *Previous 3 positions of the active agent*
4. *Previous 5 positions of the active agent*
5. *All previous positions of the active agent*
6. *Previous 3 positions of the inactive agent(s)*
7. *Previous 5 positions of the inactive agent(s)*
8. *All previous positions of the inactive agent(s)*
9. *Fire positions*
10. *Dirt positions*
11. *Tree positions*

These vision grids are matrices filled with zeros and ones. Figure 2.2 shows the translation of the environment into seven binary maps.

2.4 Input Data

The data used in this project was created by a human. This was done by controlling the agents during a simulation and saving the state of the environment and corresponding actions in a folder. The data was separated by environment size, FPS, and the number of agents. As stated before, the environment is saved as a stack of matrices. These matrices are the same width and height as the environment, and they are filled with binary values. A **1** represents that a specific cell type is located at that position, and a **0** represents that it is not located at that position. The size of the generated data can vary drastically depending on the setup of the environment. An environment of size 61×61 with two agents and a **fast** FPS will generate a lot more data than an environment of size 21×21

with two agents and a **normal** FPS. This can be explained by the fact that it takes more steps for the agents to reach and isolate the fire, meaning more states and actions need to be saved.

As mentioned in the previous section, this study also tested the effects of including 0, 3, 5, or all previous agent positions in the input data. All eleven vision girds mentioned in section 2.3 were saved during the data generation process. This allowed for the generation of different input configurations from the same data set. To give an example: in order to test the effects of including the previous **five** positions of all agents, we configured the input data to include vision grids 1, 2, 4, 7, 9, 10, and 11 from the list in section 2.3.

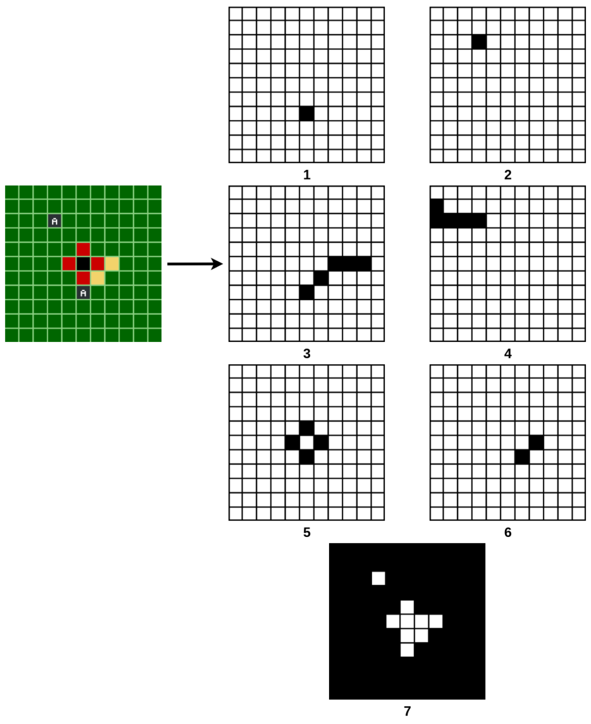


Figure 2.2: Vision grids displaying current agent positions (1 and 2), 5 previous agent positions (4 and 4) and all fire (5), dirt (6) and forest (7) positions

3 Convolutional Neural Network

A convolutional neural network is a type of neural network which was first introduced by LeCun in the 1980's (LeCun et al., 1989). LeCun's work showed that it was possible to classify handwritten digits. Nowadays, ConvNets are used for considerably more complex tasks like autonomous driving and facial recognition. The workings and benefits of ConvNets will be presented in the following sections.

3.1 Neural Network

A neural network (NN) is a type of algorithm that can be trained to recognize certain patterns. A simple NN like the one depicted in figure 3.1 consists of neurons and the connections between them. Simply said, a neuron is a thing that holds a number, typically between 0 and 1. The number inside the neuron is called its *activation*. A simple NN has three different types of layers: the input layer, the hidden layer(s), and the output layer. All layers hold some amount of neurons. The *activations* in one layer determine the *activations* in the next. This mechanism aims to achieve that each neuron from the hidden layer(s) will correspond to a specific piece of information from the training data. In digit recognition for example, the goal is that each neuron in the hidden layer corresponds to a sub-component of the digits. The subcomponent could for example be the loop in the number 9 or the straight line in the number 5. A properly trained network would then know which combination of subcomponents corresponds to which number. Such a simple NN is trained by tweaking the connections between the neurons. In a fully connected NN, every neuron in one layer is connected to every neuron in the next layer. These connections are called weights, and these weights are the things the network tweaks during training. The value of a neuron from the hidden layer (h_j) which is connected to a neuron from the input layer (x_i) through a weight (w_{ij}) is calculated by:

$$h_j = f(\sum_{i=1}^{n} w_{ij} * x_i) \qquad (3.1)$$

This equation basically calculates the weighted sum of all the activations from the previous layer.

$f()$ is a nonlinear activation function. This mechanism is used for both the hidden and output nodes. Next to weights, a neuron is typically also connected to some bias, which is an indication of whether a neuron tends to be active or inactive. The weights and biases are tweaked through an optimization algorithm, which will adjust the weights so as to improve the NNs performance. This is done by comparing the output of the network to the target output. The difference between these two is called *error*. An example of how a weight would be updated is:

$$w = w_{old} + \alpha * (y_t - y) * x \tag{3.2}$$

In which w is the weight between neuron x and neuron y, w_{old} is the old value of weight w, y_t is the target output for neuron y and α is the learning rate. The learning rate is often a small positive number. Updating all weights in the network can be done through the implementation of the *backpropagation* algorithm (Rumelhart, Hinton, and Williams, 1986).

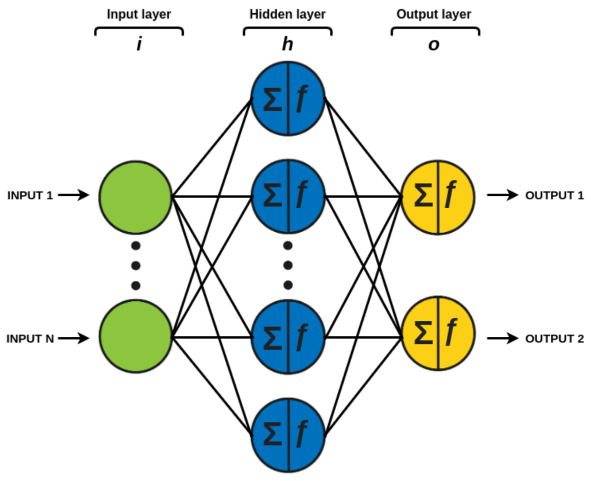


Figure 3.1: A simple neural network architecture

As stated before in section 1.2, this study used a convolutional neural network, or ConvNet for short. A ConvNet is a specialized type of neural network that consists of three or more layers, making it a so-called **deep** neural network. ConvNet architectures also have an input and output layer. In addition to that they also typically have three main types of hidden layers: a convolutional layer, pooling layer, and fully-connected or dense layer. These layers will be stacked on top of each other in order to form a full ConvNet architecture. All three layers will be now explained in more detail.

3.2 Convolutional Layer

The convolutional layer is at the core of the ConvNet since it carries the largest portion of the computational load. The convolutional layer performs a dot product between two matrices where one matrix is a restricted portion of the input matrices, and the other matrix is a so-called filter. Filters are learnable parameters that are spatially small in width and height but the depth of the filter covers all channels. During the feedforward pass, the filter slides across the input and computes the dot product between the input at any position and the entries of the filter. This process produces a so-called *activation map*. This map gives responses of that filter for every spatial position. The goal is for the network to learn filters that are activated if they see a particular visual feature. A single convolutional layer can have many filters. The hyperparameters of a convolutional layer include the filter size F, the number of filters K, the amount of zero padding P and the stride S. Stride specifies how much we slide the filter. A stride of 2 means we move two matrix entries at a time. Zero padding is a border of zeros around the input volume, enabling control over the spatial size of the output volume.

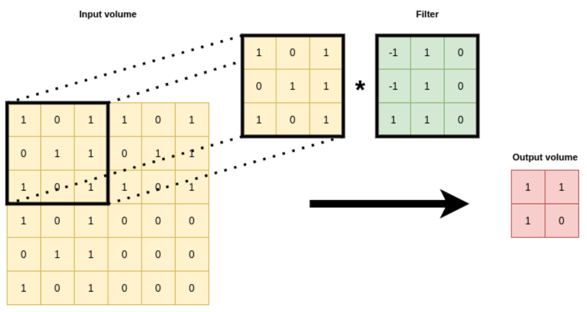


Figure 3.2: A filter (green), input volume (yellow) and output volume (red)

3.3 Pooling Layer

A pooling layer performs a downsampling operation on the input volume. Pooling layers are commonly placed after a convolutional layer. The purpose of these layers is to reduce the representation's spatial size to lower the number of parameters and thereby the number of computations. This layer performs its operations independently on every layer along the depth of the input volume. Just like in a convolutional layer, the operation is performed by sliding a "view" over the input volume. Different types of pooling layers perform different computations on the values in that "view". The most common types of pooling layers are *max pooling* and *average pooling* layers. The Max pooling operation selects the highest value in the view. The average pooling operation averages the values in the current view. In a pooling layer, the height and width of the volume is reduced. However, the depth remains the same. A pooling layer requires two hyperparameters: the stride S and the spatial extent F.

3.4 Fully Connected Layer

Fully connected layers are often found towards the end of a ConvNet architecture. A fully connected layer performs its computations on a flattened input so that all neurons in one layer can be connected to all neurons in the next, i.e., fully connected. Typically, the result of the convolution and pooling process is fed into a fully connected layer which drives the final classification process.

3.5 Activation Functions

Activation functions are used to determine the output of a neuron by mapping the values between a certain range. This study uses the *rectified linear unit* and *softmax activation function*, therefore we will only discuss these two.

ReLU (Rectified Linear Unit)

ReLU is one of the most popular activation functions. It is defined by:

$$R(y) = max(0, y) \tag{3.3}$$

In this equation y is the output from the neuron. ReLU is half rectified. It outputs 0 if y is less than zero and otherwise it outputs y, meaning that it maps the output values between 0 and infinity. A problem with this function is that all negative values become zero, meaning that it does not properly map negative values. The advantage of ReLU is the reduced likelihood of the gradient vanishing. The vanishing gradient problem can prevent a weight from changing its value during training.

Softmax

The softmax activation function turns values into probabilities that sum to one. The output is a vector that represents the probability distribution of a list of classes. This function is therefore useful for classification problems which consist of multiple classes. The softmax function is defined by:

$$\mathrm{S}(x)_i = \frac{\exp(x_i)}{\sum_j^c \exp(x_j)} \tag{3.4}$$

In this equation x_i is the output of the neural network for action i. C is the number of classes. x_j are the scores inferred by the net for each class in C and $\mathrm{S}(x)_i$ is the translated output into a probability value.

3.6 Loss Function

During training the model tries to minimize the *loss*, which is the difference between the output and the target output. The *loss* is calculated via the loss function. This study uses categorical cross entropy, which is defined by:

$$CE = -\sum_i^C t_i log(\mathrm{S}(x)_i) \tag{3.5}$$

In this equation CE is the cross entropy loss, C is the class, t_i is the target output for action i and $\mathrm{S}(x)_i$ is the softmax function discussed in section 3.5.

3.7 Adam Optimizer

In a NN, the optimizer is used to minimize the loss through a process called gradient descent, which tries to find a local or global minimum by analyzing the slope of the loss function. This project uses the Adam optimizer (Kingma and Ba, 2014) (adaptive moment estimation), which is currently recommended as the default optimization algorithm (Ruder, 2016). The advantages of the Adam optimizer according to its inventors are: "it only requires first-order gradients with little memory requirement" and "the method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients" (Kingma and Ba, 2014). The Adam optimizer uses stochastic gradient descent (SGD), meaning that some randomness is introduced during the process of optimization. Adam computes the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.6}$$

$$v_t = \beta_2 m_{t-1} + (1 - \beta_2) g_t^2 \tag{3.7}$$

β is the exponential decay rate, m_t is an estimate of the first momentum of the gradient and v_t is an estimate of the second momentum. The creators of Adam observed that these two values are biased toward zero during the first steps. This bias can be counteracted by computing new corrected estimates for the first and second momentum, which is done via:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.8}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.9}$$

These are used to update the parameters θ (weights, biases) through the following update rule:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.10}$$

α is the learning rate. The creators of Adam propose the following values as default: $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1e - 07$.

3.8 Validation

Ideally, the system should stop training once the system performs best. However, this exact point can be hard to pin down. During each training epoch the accuracy of the system is calculated via:

$$Accuracy = \frac{cp}{tp} \tag{3.11}$$

With *cp* being the correct predictions and *tp* being the total amount of predictions. A correct prediction means that the system accurately classified an input. In order to prevent overfitting, the model is not only tested on the training data but also on validation data. The validation data contains previously unseen input for the system, which was not used during training. This data is therefore used to give an indication of how well the system performs on new and unseen data.

3.9 Architecture

The architecture of the ConvNet used in this study will now be discussed. The network consists of six layers:

1. Input layer
2. Convolutional layer
3. Max Pooling layer
4. Convolutional layer
5. Convolutional layer
6. Output layer (dense/fully connected)

The architecture of the network can also be seen in figure 3.3. The shape of the input layer can range from $N \times N \times 5$ to $N \times N \times 7$, depending on the configuration mentioned in section 2.3 and the size of the map. The second layer is a convolutional layer characterized by a stride of 1, with 100 filters of size 2×2. The activation function used for this layer was ReLU. The third layer is a max-pooling layer characterized by a pool size of 3×3 and a stride of 1. The fourth layer is again a convolutional layer with the same hyperparameters as the second layer, i.e. a filter size of 2×2, 100 filters and stride of 1. This layer also utilized the ReLU function. The fifth layer is another convolutional layer characterized by a filter size of 2×2 with 25 filters, again a stride of 1, and again the ReLU function. The final layer is a fully connected layer containing six output nodes representing the six possible actions. For this fully connected layer, the softmax activation function was used. As stated before, this network used the Adam optimizer for optimization. Moreover, categorical cross-entropy was used as the loss function. Furthermore, the *early stopping* and *model checkpoints* methods were used to find the optimal model. Early stopping rules prevent the model from overfitting by tracking the performance of the model on the validation dataset and stopping the learning process once this performance starts to degrade. Model checkpoints save the best model so that the system can go on with training and if the performance decreases the saved model can be loaded.

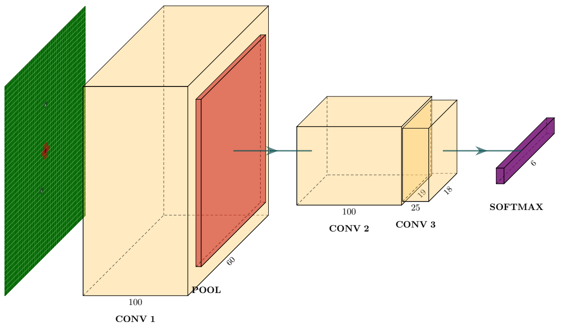


Figure 3.3: Architecture of the ConvNet used in this study for a 61×61 map. Three convolutional layers (yellow), 1 max pooling layer (red) and the softmax layer (purple)

3.10 Data Augmentation

ConvNets need a large amount of training data. In order to provide the system with a sufficient amount of data, data augmentation techniques were applied to the existing data set. Every entry in the data set consists of a stack of matrices and a label. This data was augmented by rotating the matrix 90, 180, or 270 degrees, followed by a correction of the attached label. To give an example, if the agent is at a position above the fire, it needs to perform a "down" action to move closer to the fire. If the input matrices are then rotated 180 degrees, then the action should be an "up" action instead of a "down" action, which can be seen in figure 3.5. This method increased the size of the data set by 400 percent.

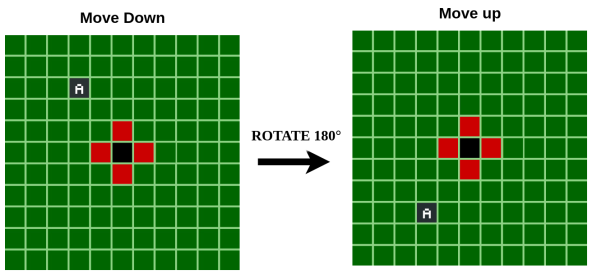


Figure 3.4: Input rotated 180°

4 Experimental Setup

This section will cover the experimental setup for all performed experiments. First, we will discuss the experimental setup used to test the effects of different ConvNet input configurations. Subsequently, we will discuss the setup for the final experiments. The final experiment used the best input configuration, found in the **input configuration experiment**. Lastly, we will discuss early stopping, model checkpoint, the system's performance measure and the hyperparameters used for the Adam optimizer.

4.1 Setup Input Configuration Experiments

To test the effects of including different amounts of the agent's previous position on the model's performance, we tested seven input configurations. These input configurations were tested with a two agent system on a 41×41 environment with a **fast** FPS. This environment was chosen because it is one of the more challenging environments. The models were trained on 200 training and 80 validation episodes worth of data. Seventy-five percent of this data was generated through the data augmentation techniques mentioned in section 3.10. With this data, ten models were trained, which were all tested during 100 episodes. A single episode is finished once the fire is isolated or if the fire reaches the map's border, i.e. when it is out of control.

4.2 Setup Final Experiments

The final experiments were done with one, two and four agents, which were tested on a 21×21, 41×41, and 61×61 map with a **normal** FPS and a **fast** FPS. This resulted in a total of 18 experimental setups. These experiments were done with the best input configuration found during the testing mentioned in section 4.1. In every experiment, 20 models were trained on 200 training episodes and 80 validation episodes, of which 75 percent was generated through the data augmentation techniques mentioned in section 3.10. Again, each model was tested on 100 episodes.

4.3 Adam, Early Stopping and Model Checkpoint

As mentioned in section 3.9, the ConvNet used in this study used early stopping and model checkpoint to find the best model. Early stopping used a patience of 10, meaning that if the loss is not reduced during ten epochs, then the training was stopped. Model checkpoint saves the model with the highest accuracy on the validation data. The hyperparameters for Adam can be found in table 4.1. The value for α was chosen based on preliminary tests. ϵ, β_1 and β_2 are set to the default values suggested by Adams creators.

ϵ	1e-7
β_1	0.9
β_2	0.999
α	0.005

Table 4.1: Hyper parameters for the Adam optimizer

4.4 Performance Measure

The performance of the models was measured by the average number of fires they isolated during 100 episodes. Furthermore, the percentage of burnt cells at the end of a successful round was also measured. A successful round is defined as a round in which the fire is isolated.

5 Results

This section will cover the results for the input configuration experiment, followed by the results for the final experiments. There are two main performance measures. The first is the average amount of fires isolated (out of 100 episodes). The second is the percentage of burnt cells within the map at the end of a successful episode.

5.1 Results Input Configuration Experiment

This section will cover the results of the input configuration experiment. As stated before, the goal of this experiment was to find the best input configuration for the final experiments. The input configurations were tested with a two-agent system on a 41×41 environment with a **fast** FPS. As can be seen in figure 5.1, the results show that the left-most bar labeled "0" performed the best, with a mean of 87.0 and a standard deviation of 4.6. This input configuration has a history length of 0, meaning that the input does not include any previous steps from the agents. In other words, the input only contains the current position of all agents and the positions of the forest, fire, and dirt cells. The exact statistical results can be found in table A.1 of appendix A. All further experiments will exclusively use an input configuration containing only current agent positions and forest, fire and dirt cell positions.

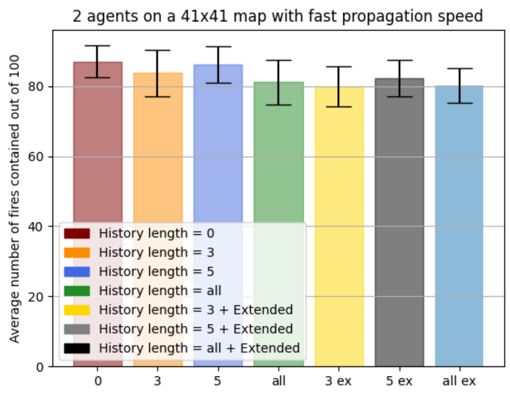
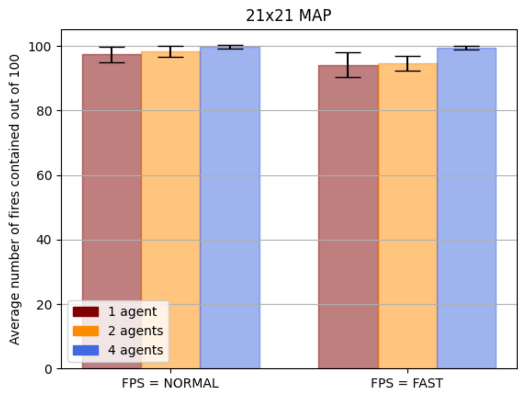


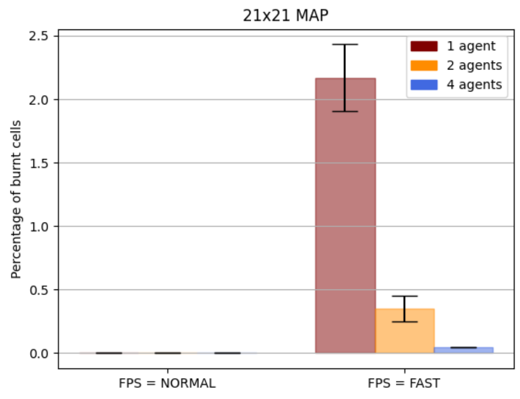
Figure 5.1: 41×41 **map with 2 agents and** ***fast*** **FPS**

5.2 21x21 map

This section will cover the results for the 21×21 map. The results for the 21×21 map can be found in figure 5.2.



(a) Average amount of fires contained on a 21×21 **map**



(b) Percentage of burnt cells on a 21×21 **map**

Figure 5.2: Results for the 21×21 **environment**

Figure 5.2a shows the average amount of fires contained during 100 episodes. The three leftmost bars represent the results for the simulation with a **normal** FPS and the three rightmost bars represent the results for the simulation with a **fast** FPS. The exact results concerning the number of contained fires can be found in table A.2 of appendix A and table B.1 of appendix B. Figure 5.2b shows the average percentage of burnt cells. Again the three leftmost bars represent the results for the simulation with a **normal** FPS and the three rightmost represent the results for the simulation with a **fast** FPS.

From figure 5.2a it can be seen that all models perform quite well. The one-agent models contain 97.4 fires on average in the normal FPS environment and 94.1 fires in the fast FPS environment. The two-agent models perform slightly better, containing 98.4 fires in the normal FPS environment and 94.7 fires in the fast FPS environment. However, for both the fast and normal FPS environment, the difference between the one and two-agent models is not significant. The four-agent models perform significantly better than both the one and two-agent models, containing 99.7 fires in the normal FPS environment and 99.6 fires in the fast FPS environment.

Figure 5.2b shows no bars on the left, which is due to the percentage of burnt cells being zero in these cases. However, in the fast FPS case it can be seen that a one-agent model leads to the highest percentage of burnt cells (2.2%), followed by the two-agent model (0.4%). The four-agent model has the lowest amount of burnt cells (0.5%).

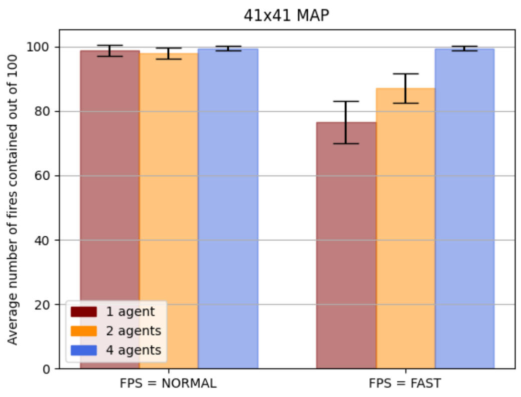
5.3 41x41 map

This section will cover the results for the 41×41 map. The results for the 41×41 map can be found in figure 5.3. Figure 5.3a shows the average amount of fires contained during 100 episodes. The exact result concerning the number of contained fires can be found in table A.3 of appendix A and table B.2 of appendix B. Figure 5.3b shows the average percentage of burnt cells.

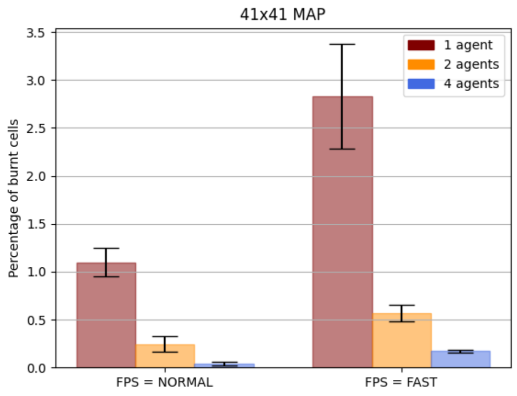
From figure 5.3a it can be seen that the one-agent models contain 98.8 fires on average in the normal FPS environment and 76.5 fires in the fast FPS environment. The two-agent models perform slightly worse for the normal FPS environment, containing 98.0 fires on average. However this difference is not statistically significant. For the fast FPS environment, the two-agent models perform significantly better than the one-agent models, containing 87.0 fires on average. The four-agent models contained 99.4 fires in the normal FPS environment, which is significantly better than then the two agent model, but not significantly better than the one-agent model. In the fast fps environment the four-agent model contained 99.5 fires on average, which is significantly better than both the one and two-agent models.

Figure 5.3b shows that a one-agent model again leads to the highest percentage of burnt cells (1.1% for normal FPS and 2.3% for fast), followed by the two-agent model (0.2% for normal FPS and

0.6% for fast). The four-agent model has the lowest amount of burnt cells (0.04% for normal FPS and 0.2% for fast).



(a) Average amount of fire contained on a 41×41 **map**



(b) Percentage of burnt cells on a 41×41 **map**

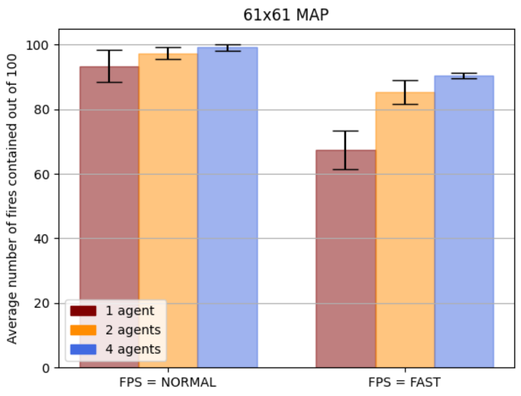
Figure 5.3: Results for the 41×41 **environment**

5.4 61x61 map

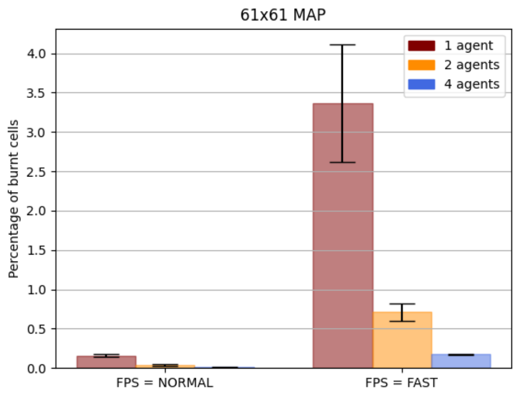
This section will discuss the results for the 61×61 map. The results for the 61×61 map can be found in figure 5.4. Figure 5.4*a* shows the average amount of fires contained during 100 episodes. The exact result concerning the number of contained fires can be found in table A.4 of appendix A and table B.3 of appendix B. Figure 5.4*b* shows the average percentage of burnt cells.

From figure 5.4*a* it can be seen that the one-agent models perform the worst, containing on average 93.4 fires in the normal FPS environment and 67.3 fires in the fast FPS environment. The two-agent models perform significantly better, containing 97.4 fires on average in the normal FPS environment and 85.4 fires in the fast FPS environment. The four-agent models perform significantly better than the one and two-agent models, containing 99.2 fires in the normal FPS environment and 90.5 fires in the fast FPS environment.

Figure 5.3*b* shows that a one-agent model again leads to the highest percentage of burnt cells (0.2% for normal FPS and 3.4% for fast), followed by the two-agent model (0.04% for normal FPS and 0.7% for fast). The four-agent model has the lowest amount of burnt cells (0.01% for normal FPS and 0.2% for fast).



(a) Average amount of fire contained on a 61×61 **map**



(b) Percentage of burnt cells on a 61×61 **map**

Figure 5.4: Results for the 61×61 **environment**

6 Conclusions

In this paper, we tested the ability of a single and multi-agent system to isolate wildfires in a simulation. The multi-agent systems consisted of either two or four agents. The tests were done in three different environment sizes. Namely in a 21×21, 41×41, and 61×61 environment. Next to the environment sizes all environments were simulated with a **normal** and **fast** fire propagation speed. Additionally, we tested which input configuration would result in the best performance. The results show that an input configuration without information about the agents previous positions yields the highest performance.

The one-agent system performed the worst. It consistently resulted in the highest amount of burnt cells. Furthermore, it also resulted in the lowest amount of contained fires for all environments except one (41×41, FPS=normal). However, this difference was less than one (see table A.3). The reason for this difference is most likely the quality of the training data which the two-agent system was trained on. The data might contain some bad moves which decrease the accuracy of the trained model.

The two-agent system performed better than the one-agent system. It contained more fires in all experiments except for the one mentioned above. The average amount of burnt cells was lower for the two-agent system compared to the one-agent system for all environments. However, the two-agent system never outperformed the four-agent system.

The four-agent system performed the best in all experiments. For the experiments with a normal FPS it was able to isolate 99.7 percent on average in a 21×21 environment, 99.4 in the 41×41 environment, and 99.2 in the 61×61 environment. For the experiments with a fast FPS it was able to isolate 99.6 percent on average in a 21×21 environment, 99.5 in the 41×41 environment, and 90.5 in the 61×61 environment. Furthermore, the four-agent system always resulted in the lowest amount of burnt cells.

7 Discussion

This study showed that a convolutional neural network based multi-agent system can effectively be used to isolate wildfire in a simple simulation. The results indicate that in most cases more agents result in better performance. However, it has to be stressed that the results from this study are not indicative of the effectiveness of such a system in the real world.

It would be interesting to see if the technique used in this study could also effectively be used in larger and more complex environments. The environments in this study were only defined by their size and fire propagation speed. In order to move closer to the real world, one could introduce different types of environmental factors such as terrain, wind, elevation, etc.

Furthermore, the results of this study showed that an input without the previous agent positions leads to the best performance. However, it seems like much valuable information is lost if the model does not consider the historical information of the environment. It would be interesting to see if other machine learning techniques could be used to take historical information into consideration during action selection. In addition to that, the agents used in this study did not communicate with each other. A study into the effects of a similar system with communication between agents could prove worthwhile.

Finally, with the danger of wildfires rising worldwide, the application of machine learning to the problem of wildfire control could prove beneficial. More research into this subject could pave the way for a future in which machine learning contributes to the fight against wildfires.

References

T. Hammond, D. J. Schaap, M. Sabatelli, and M. Wiering. Forest fire control with learning from demonstration and reinforcement learning. In *2020 International Joint Conference on Neural Networks (IJCNN)*, 09 2020.

P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

S. Knegt, M. Drugan, and M. Wiering. Opponent modelling in the game of tron using reinforcement learning. *In ICAART (2)*, page 29–40, 2018.

Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Handwritten digit recognition with a back-propagation network. In *NIPS*, 1989.

Mihaela Oprea. Applications of multi-agent systems. In *Information Technology*, pages 239–270. Springer US, 2004.

S. Ruder. An overview of gradient descent optimization algorithms. *ArXiv*, abs/1609.04747, 2016.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529: 484–489, 01 2016.

M. Wagenaar, E. Okafor, W. Frencken, and M. Wiering. Using deep convolutional neural networks to predict goal-scoring opportunities in soccer. In *ICPRAM*, 2017.

M. Wiering and M. Doringo. Learning to control forest fires. *Proceedings of the 12th international Symposium on 'Computer Science for Environmental Protection*, page 378–388, 1998.

M. Wiering, F. Mignogna, and B. Maassen. Evolving neural networks for forest fire control. *Proceedings of the 14th Belgian-Dutch Conference on Machine Learning*, page 113–120, 2005.

Y. Ywema and M. Wiering. Learning from demonstration for isolating forest fires using convolutional neural networks. *Bachelor's Thesis*, 2020.

A Appendix

	Average number of fires contained (out of 100)	standard deviation
History length = 0	87.0	4.6
History length = 3	83.7	6.6
History length = 5	86.2	5.2
History length = ALL	81.1	6.5
History length = 3 + EX	79.9	5.7
History length = 5 + EX	82.3	5.2
History length = ALL + EX	80.2	5.0

Table A.1: Results for the input configuration experiment on a 41x41 map with 2 agents and fast fire propagation speed

FPS	Number of agents	Average number of fires contained (out of 100)	Standard deviation
Normal	1 Agent	97.4	2.4
	2 Agents	98.4	1.7
	4 Agents	99.7	0.6
Fast	1 Agents	94.1	4.0
	2 Agents	94.7	2.4
	4 Agents	99.6	0.8

Table A.2: Contained fires 21×21 **map**

FPS	Number of agents	Average number of fires contained (out of 100)	Standard deviation
Normal	1 Agent	98.8	1.6
	2 Agents	98.0	1.8
	4 Agents	99.4	0.7
Fast	1 Agents	76.5	6.5
	2 Agents	87.0	4.6
	4 Agents	99.5	0.7

Table A.3: Contained fires 41×41 **map**

FPS	Number of agents	Average number of fires contained (out of 100)	Standard deviation
Normal	1 Agent	93.4	5.0
	2 Agents	97.4	1.8
	4 Agents	99.2	0.9
Fast	1 Agents	67.3	6.0
	2 Agents	85.4	3.7
	4 Agents	90.5	3.0

Table A.4: Contained fires 61×61 **map**

FPS	Number of agents	Percentage burnt cells (%)	Standard deviation
Normal	1 Agent	0.0	0.0
	2 Agents	0.0	0.0
	4 Agents	0.0	0.0
Fast	1 Agents	2.2	0.3
	2 Agents	0.4	0.1
	4 Agents	0.05	0.01

Table A.5: Percentage burnt cells 21×21 **map**

FPS	Number of agents	Percentage burnt cells (%)	Standard deviation
Normal	1 Agent	1.1	0.02
	2 Agents	0.2	0.08
	4 Agents	0.04	0.02
Fast	1 Agents	2.8	0.5
	2 Agents	0.6	0.1
	4 Agents	0.2	0.03

Table A.6: Percentage burnt cells 41×41 **map**

FPS	Number of agents	Percentage burnt cells (%)	Standard deviation
Normal	1 Agent	0.2	0.02
	2 Agents	0.04	0.01
	4 Agents	0.01	0.002
Fast	1 Agents	3.4	0.7
	2 Agents	0.7	0.1
	4 Agents	0.2	0.02

Table A.7: Percentage burnt cells 61×61 **map**

B Appendix

FPS	Comparison	T-value	P-value	Significant?
Normal	1 agent vs. 2 agents	1.5757	0.1234	NO
	1 agent vs. 4 agents	4.1255	0.0002	YES
	2 agents vs 4 agents	3.1421	0.0032	YES
Fast	1 agent vs. 2 agents	0.5999	0.5521	NO
	1 agent vs. 4 agents	5.2926	>0.0001	YES
	2 agents vs 4 agents	8.7373	>0.0001	YES

Table B.1: Comparison between the one, two and four-agent system on a 21×21 map. Unpaired t-test is used to compare the mean (contained fires) between two systems

FPS	Comparison	T-value	P-value	Significant?
Normal	1 agent vs. 2 agents	1.5895	0.1202	NO
	1 agent vs. 4 agents	1.5446	0.1307	NO
	2 agents vs 4 agents	3.4777	0.0013	YES
Fast	1 agent vs. 2 agents	5.9300	>0.0001	YES
	1 agent vs. 4 agents	15.6933	>0.0001	YES
	2 agents vs 4 agents	12.0804	>0.0001	YES

Table B.2: Comparison between the one, two and four-agent system on a 41×41 map. Unpaired t-test is used to compare the mean (contained fires) between two systems

FPS	Comparison	T-value	P-value	Significant?
Normal	1 agent vs. 2 agents	3.3623	0.0018	YES
	1 agent vs. 4 agents	5.1093	>0.0001	YES
	2 agents vs 4 agents	3.9689	0.0003	YES
Fast	1 agent vs. 2 agents	11.5240	>0.0001	YES
	1 agent vs. 4 agents	15.4530	>0.0001	YES
	2 agents vs 4 agents	4.7860	>0.0001	YES

Table B.3: Comparison between the one, two and four-agent system on a 61×61 map. Unpaired t-test is used to compare the mean (contained fires) between two systems