# Physarum AI

**un progetto di:** Niccolò Maria Di Santo
**per il corso di:** AiLab Computer Vision and NLP

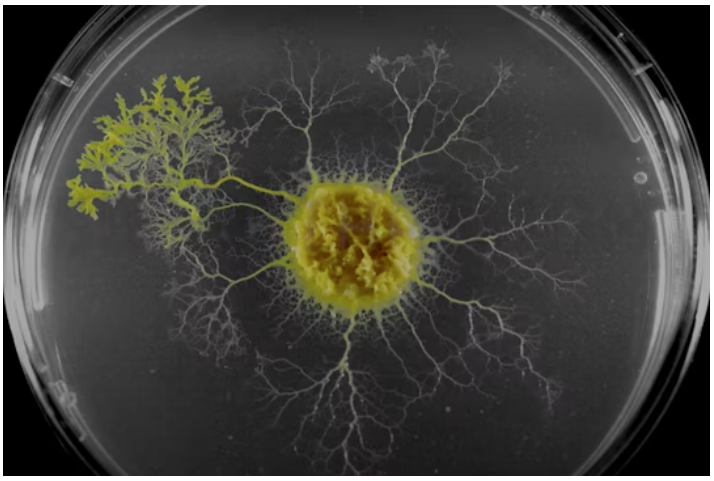
1. **Introduction**

   Slime mould (*Physarum polycephalum*) is a simple organism that grows toward food and builds efficient networks. It is very interesting because the way it moves toward food makes it seem like it has a brain, when in reality not only does it not have one, but it is a single-celled organism. Its behavior has inspired many projects in science and artificial intelligence. For example, the **Physarum Chip** project used the organism as a living computer to make logic operations.

   Physarum is also the main subject of this project. I've used the images of its growth and tested two models — **KNN, CNN** — to see if they can imitate the mold.

   The idea is to take the growth images, turn them into binary matrices (background / Physarum), then mark by hand (to help the models) the points where food (oat) is placed. These matrices are used as input and target for training. In this way, the models can learn that Physarum grows from food points and expands following specific patterns.

   The goal is not only to recreate the image, but to try to simulate the behavior of the mould in a simple way, using machine learning techniques without very complex networks.



2. **Method**

   **2.1 Dataset and image transformation**

   The first idea was to create an original dataset. For this reason, I bought a strain of *Physarum polycephalum* and tried to record a time-lapse in home conditions.

Unfortunately, the culture did not grow as expected, so it was not possible to collect my own video material.

Because of this, I had to look for external video sources. Found it and at first I extracted **20 frames**, taken at regular time intervals. But during the first tests with predictive models, I saw a problem: the frames were too similar. The models learned only to copy the input (an “identity function”) instead of learning the real growth dynamics.

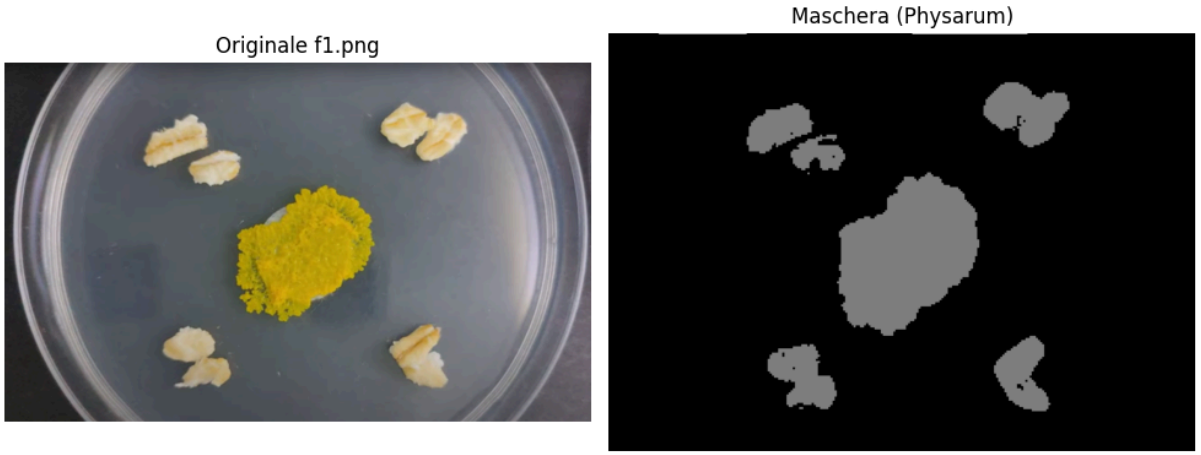
To fix this problem, I reduced the frames from **20** to **8**. These **8** frames could keep more distance in time between them. This way, the images show clearer differences (expansion, retraction, and movement toward food), and the models can learn better transitions.

Each frame was then processed through a pipeline to convert the original color image into a binary or multi-class mask.

The main steps were:

- **Color conversion** – The image is converted from BGR (OpenCV) to HSV (Hue, Saturation, Value). This helps to isolate the yellow-orange color of Physarum.
- **Color thresholding** – A range on the H and S channels selects the pixels of the organism.
- **Resizing** – The binary mask is resized to 200x280 pixels using interpolation *INTERN_NEAREST* to keep the values discrete (0 or 255).
- **Binarization and normalization** – Pixel values are normalized to [0.0, 1.0], where 1.0 = Physarum and 0.0 = background.

The final mask has three classes: Background (0), Physarum (1), Food (2).



### 2.2 Final Dataset

The final dataset is made of 8 processed frames, each with a size of 200x280 pixels and 3 classes: **background (0), Physarum (1), and food (2)**. These frames form a short time sequence of growth.

For training, the dataset is organized as pairs of input and target:

- **X[t]** is the state of the system at time *t*.
- **y[t]** is the state of the system at time *t+1*.

This means the model sees how the mould looks in one frame and tries to predict how it will look in the next. In this way, the dataset does not only give static images, but also teaches the idea of growth and change over time.

Finally, the dataset is saved in **NumPy (.npy)** format, so it is easy to load and use for experiments with different models.

### 2.3 Data Preparation for Training

After creating the masks, the data must be put into the right format to be used by the Machine Learning models. This step is very important and has three main parts:

**Creating Input-Target Pairs**
The sequence of 8 frames is turned into pairs for supervised learning. The frame at time *t* (X) is used to predict the frame at time *t+1* (y).

**Reshaping for the Model**
The data is reshaped depending on the model type:

- For **KNN**: the 2D images (H, W) are flattened into one 1D vector of length H * W. Each pixel is seen as an independent feature.
- For **CNN**: one extra “channel” dimension is added, creating a 4D tensor of shape (7, 1, H, W). This lets the network keep the spatial relations between pixels.

**Normalization**
The pixel values (0 = background, 1 = Physarum, 2 = food) are scaled into a smaller range, for example [0, 1], by dividing by the maximum value. This makes training more stable and efficient for neural networks.

## 4. Models

### 4.1 K-Nearest Neighbors KNN

**Choice of the Model**: A Simple and Interpretable Baseline
The K-Nearest Neighbours (KNN) algorithm was chosen as the first baseline model.

*Easy to implement:* KNN is a “lazy” algorithm, which means it does not need a real training phase. It only stores the data. This made it possible to build and test the prediction pipeline very quickly.

*Intuitive*: To predict the future state, the model looks for the most similar past configuration and “remembers” what happened next. This tests the basic idea that there is temporal similarity in the dataset.

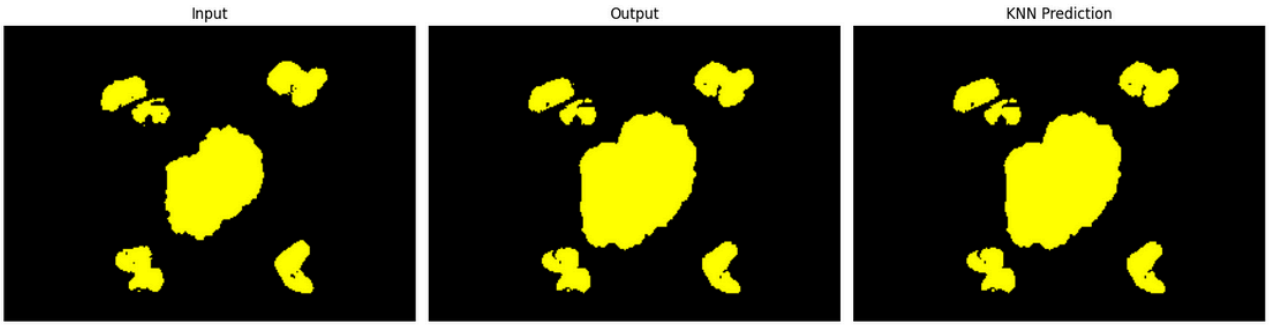
**Implementation:**

For this project, the implementation used the scikit-learn library
A custom version called *PhysarumKNN* was created with two main changes compared to standard KNN:

- **Cosine distance**: The model uses cosine distance (`metric='cosine'`). This compares the “shape” of the configurations instead of their absolute values, making it more robust.

- **Weighted average**: The prediction is made by a weighted mean, where closer neighbors have higher weight (inverse of distance). This improves the result compared to a simple average.

In short, to predict the next state, the model finds the *k* most similar past states and combines their future states.



## 4.2 Convolutional Neural Network (CNN)

**Choice of the Model**

The Convolutional Neural Network (CNN) was chosen because it can keep and analyze the spatial relations inside an image. Unlike KNN and MLP, which flatten the data and lose the 2D structure, the CNN works directly on the image geometry.

With convolution filters, it can recognize patterns from simple edges and textures up to more complex shapes. This is important for a spatial prediction task like the growth of *Physarum*, where the local context of each pixel (neighbors, edges, directions) helps to predict its future state.

**Implementation:**

The implementation followed a simple encoder architecture for image regression with a training of 10 epochs.

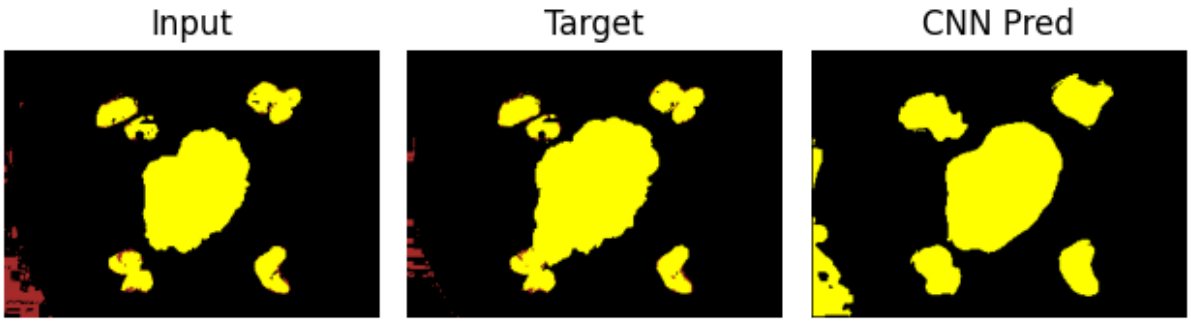
**Input preparation:** The input tensor `X_cnn` has shape `(7, 1, H, W)`, where 1 is the single channel (grayscale mask).

***Convolutional blocks (Encoder)*:** The network is made of repeated blocks.

– **Convolutional layer (`Conv2d`):** Applies filters to the image, each filter detecting a feature like an edge or an active region.
– **Activation function (ReLU):** Adds non-linearity, replacing negative values with zero and allowing complex patterns.
– **Pooling (`MaxPool2d`):** Reduces size while keeping the most important features, making the representation more compact and robust.

***Final layer (Decoder/Output*):** The last layer is a convolutional layer with one filter and a sigmoid activation.
– One filter is used because the output must be a single-channel image of the same size as the input.
– The sigmoid activation compresses pixel values into `[0, 1]`, the same range as the normalized target masks. This layer “reconstructs” the image from the abstract features.

**Loss function and optimization:** The model was trained to minimize the **Mean Squared Error (MSE)** between prediction and target, pixel by pixel. The optimizer (Adam) updated the weights to reduce this error.



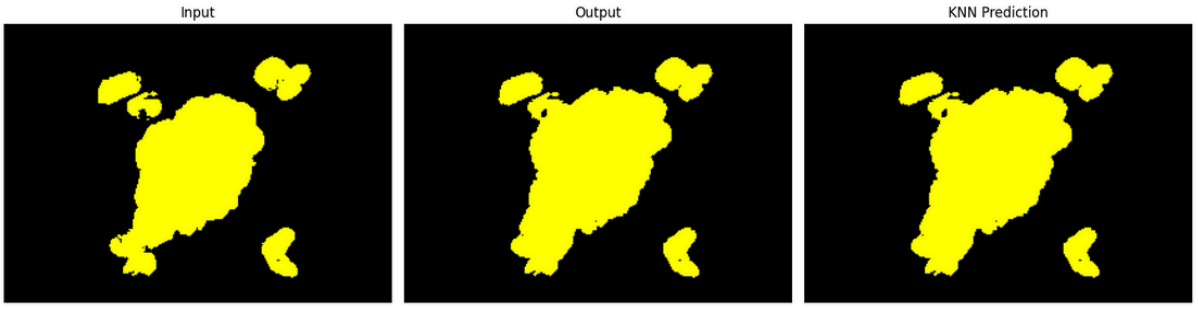
## 5. Conclusions CNN vs KNN

The experiments with KNN and CNN models showed very different results.

The ***KNN*** model had a very high success rate of *100%*. This means it could always find similar patterns in the training data. However, the predictions were often blurry and not precise because ***KNN*** just copies from existing examples without really understanding the growth patterns.

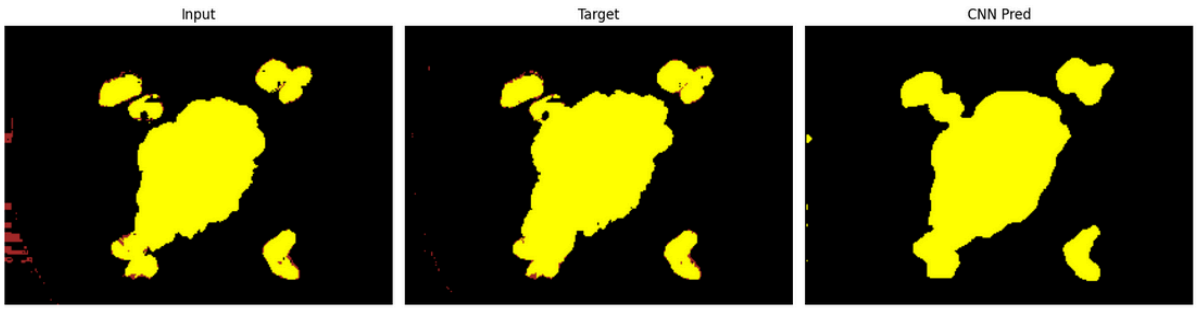
The ***CNN*** model had a loss value going from 0.08 to 0.01 during training with a success rate of *90%*, this number may seems to be high but with a steady background it turns out to be low. This means the ***CNN*** was learning to reduce its errors. But the problem was that the ***CNN*** learned to *"cheat"* instead of predicting real growth, it just learned to copy the input image with small changes. The low loss value doesn't mean it was making good predictions; it only means it was good at copying.

***Knn***:




```
Frame  2: Accuracy = 100.0%
```

***Cnn***:




```
Frame  2: Accuracy = 95.0%
```

**References**:

**1** Physarum Chip: Developments in growing computers from slime mould

James G.H. Whiting, Ben P.J. de Lacy Costello, Andrew Adamatzky

**2** First Image:

https://massivesci.com/articles/slime-mold-ants-audrey-dussutour-breakthrough/

**3** Youtube Video of the mold:

https://youtu.be/u8H2iOICdIE?si=XrlDHx-5eBQHSLGJ