CS-2490: Advanced Machine Learning

Course Project Report

Personalized Eye Gaze Tracker

-Dhruva Panyam (UG '21)

Spring 2021

1. Introduction

2. Problem Analysis

2.1. Parameters

The problem of mapping an image of an eye to specific screen coordinates requires a few parameters. Given the gaze vector of the eye, we must find the point where it intersects the screen's plane. To find this, we need the *screen's dimensions*, the *normal distance* of the eye from the screen, and the *lateral distance* of the eye from the center of the screen. Using the normal distance and the lateral distance, the position of the eye relative to the screen can be found. The position of the eye, the gaze direction vector and the screen dimensions are sufficient to calculate the mapped coordinates on the screen.

2.2. Personalization

For the purpose of this project, several of the problem's parameters that had negligible change were assumed to be constant. For instance, the screen dimensions remained the same throughout, and the normal distance of the eye from the plane was assumed to be constant. (Note, however, that this assumption could definitely contribute to the model's error rate).

To adapt this application to the desired result, rather than map the eye to specific screen coordinates, a grid system was implemented, and the problem was treated as a classification problem. The grid system was tested with sizes 3-5 for best performance and utility. Classification problems are in some ways simpler to work with, and CNNs are most widely used for classification.

3. Data Collection

Since the application aimed to be a personalized eye gaze tracker, it was possible to generate my own dataset for training and testing. The necessary parameters for the problem are the lateral distance of the eye from the center of the screen, an image of the eye, and the corresponding grid cell that the eye is looking at. The dataset was generated using a program that carried out the following:

a) Using OpenCV, it captured image feed continuously from my webcam in a region of interest (ROI) of the frame.
b) Given the current grid cell, a shell command ('xdotool') was used to move the mouse randomly within the cell.
c) The image was saved, labelled with the corresponding grid cell.

d) Steps (b) and (c) were repeated for all grid cells, and the entire process was repeated in different lighting conditions.

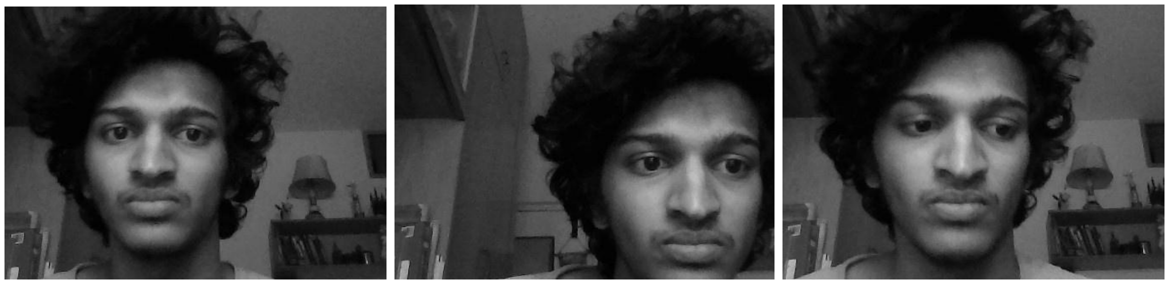

(a) (b) (c)

Fig 3.1: *ROI images captured with the eye in different positions of the frame. Fig (a) is labelled with the grid cell [0,2,95] (row #2, col #2), fig (b) is labelled with the grid cell [1,3,115], fig (c) is labelled with the grid cell [3,3,76]*

4. Data Processing

4.1. Eye Detection

To train a convolutional network to detect the gaze direction of an eye, treating the entire ROI as input will be problematic, since the network learns using features of the image that do not relate to the problem at hand. So, the input must be specific to the cause for maximum efficacy. This involves detecting the eyes from the labelled image.

For a large set of people, detecting eyes from a face requires a pre-trained model for shape predictions, or landmark predictions, which takes time to run in realtime. However, since this project was personalized, there were more efficient methods to detect the eye with very high accuracies. Since the eye will always look similar, template matching was a much faster algorithm to use. Template matching involves shifting a template image across a larger image to find the position that results in the smallest image difference (using a specific loss function). Given that the labelled images were in different illuminations, and the pupil shifted in different directions, having multiple templates for lighting and eye shape improved the detection accuracy vastly. Further, to account for the normal distance of the eye from the screen, the templates were matched in various sizes to find the best match, and in turn find the relative normal distance.

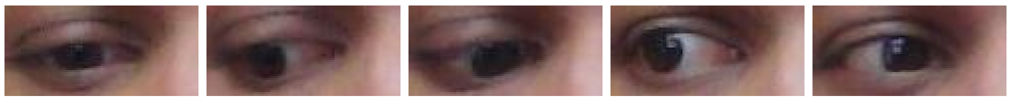
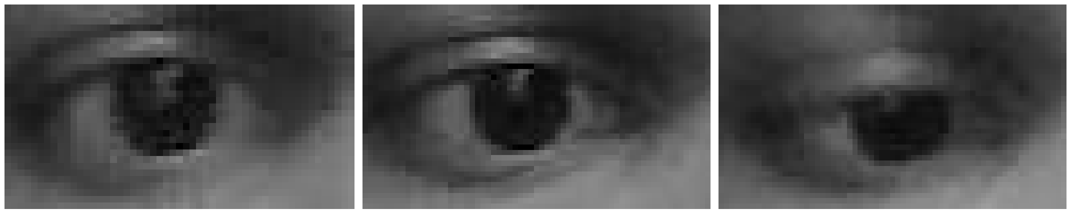


Fig 4.1: *Examples of templates used to match the images to detect the eye's most likely position in that image. Knowing which template matched with a higher accuracy can also aid in predicting the approximate grid cell.*

4.2. Annotation

The labelled ROIs were processed to generate the training dataset. Processing involved template matching to find the left eye position, store the image indices of the best template match, and note the size of the template that was matched. So, the image of the left eye is saved, labelled with the grid cell and the coordinates of the top left corner of the eye.

<insert eye>



(a) Grid: [0,2] Eye: [112,113] (b) Grid: [1,3] Eye: [274,166] (c) Grid: [3,3] Eye: [131,119]

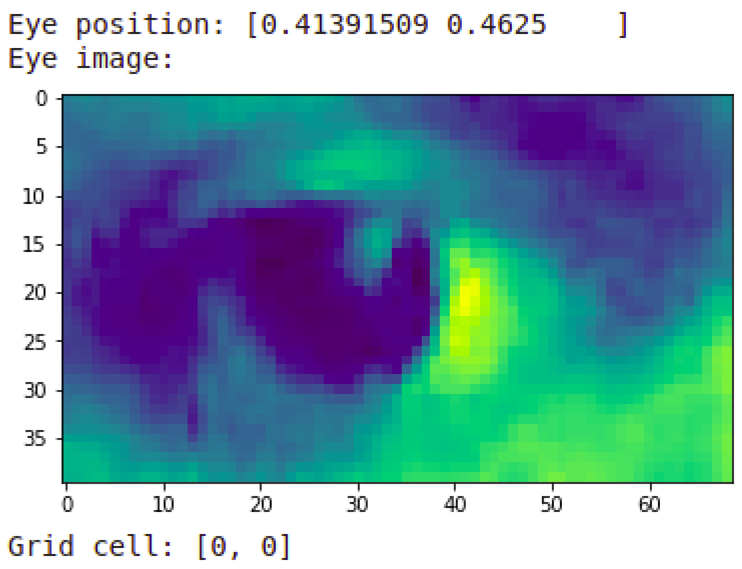
Fig 4.2: *Images of detected eyes from the images in Fig. 3.1. Each image is annotated with the grid cell and the eye position.*

5. Training the Model

5.1. Inputs

The input to the model consisted of an image of the left eye and the normalized coordinates of the position of the eye in the webcam's frame. Perhaps another input could have been the size scale of the template of the eye that was used to match this image, but for simplicity this is ignored.

<insert input>



```
Eye position: [0.41391509 0.4625    ]
Eye image:
```

```
Grid cell: [0, 0]
```

Fig 5.1: *Input of a training data sample.*

5.2. Model Architecture

The model's architecture involves two separate networks that are merged together. The image undergoes a convolutional network with 4 blocks. The kernel size was kept at (3,3) and the number of filters increased from 32 to 256? These filters are then flattened and connected fully to a hidden layer. The normalized position (array of size 2) undergoes 2 hidden fully connected layers. The two results are then concatenated and densely connected to an output of size *grid_size^2*.

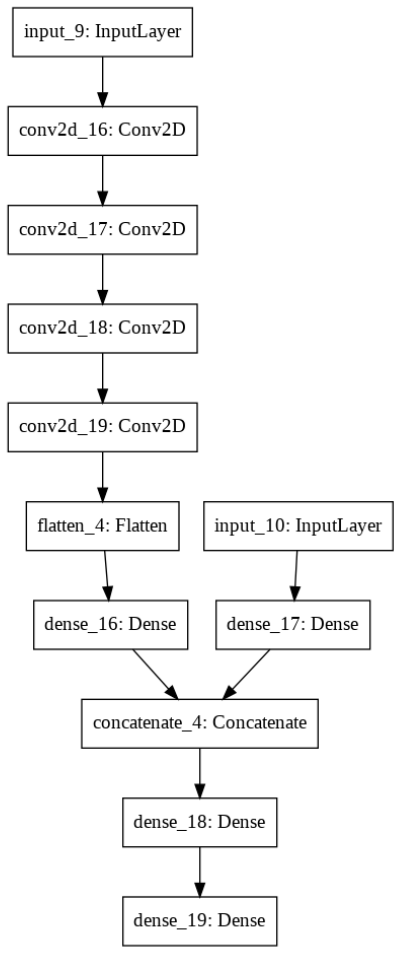


Fig 5.2: *Image of the model architecture. Note: Every convolutional layer was followed by a MaxPooling2D layer and a Dropout layer. Dropout layers were also added to the fully connected layers after concatenation.*

5.3. Outputs

The output of the model is an array of confidence values for each cell in the screen's grid. These confidence values can be used to find the highest value as the prediction, but are also used to evaluate the performance, and combined to receive more accurate results. This is explained further in the next sections.

6. Evaluations

Below is a plot of 25 random samples from the testing dataset along with the guess and truth.

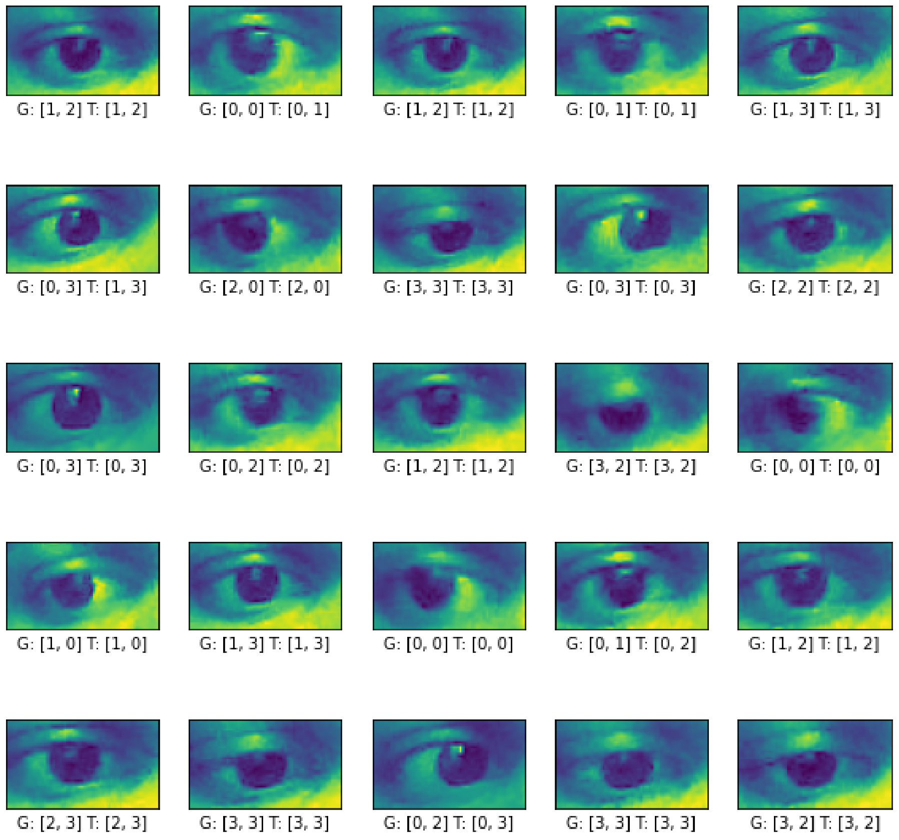


Fig 6.1: *A random sample of 25 testing images. Each image has the model's guess (G) and the true value (T). It can be seen that almost all inaccuracies are 1 row or column off the actual value.*

The model was trained for 100 epochs with a scheduled learning rate based on the epoch number. The following is the history of the model's training process.

<insert plot>

One thing to note is that the validation data was sometimes part of the same distribution as the training data (from the same run of data generation). While this does not affect the model's training, evaluating the model on a different distribution gives us a much better understanding.

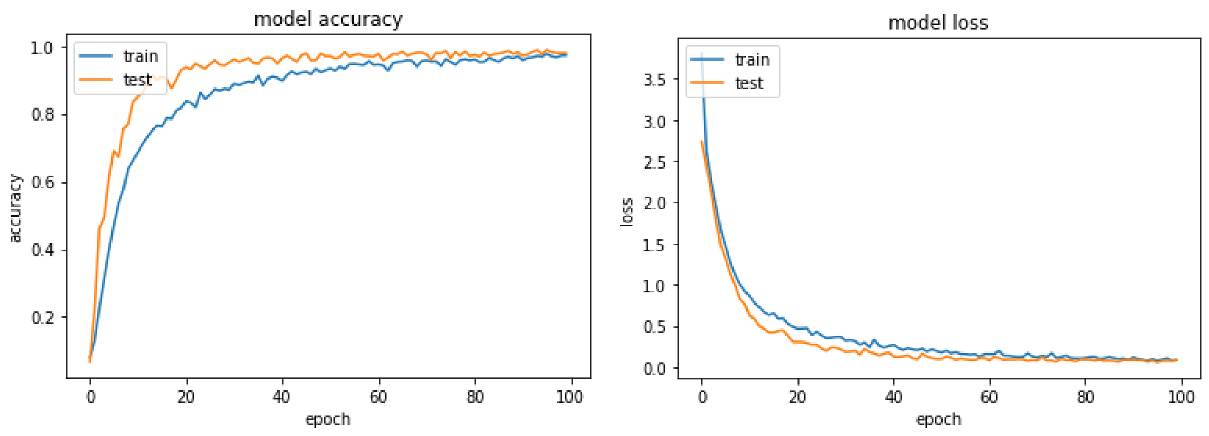
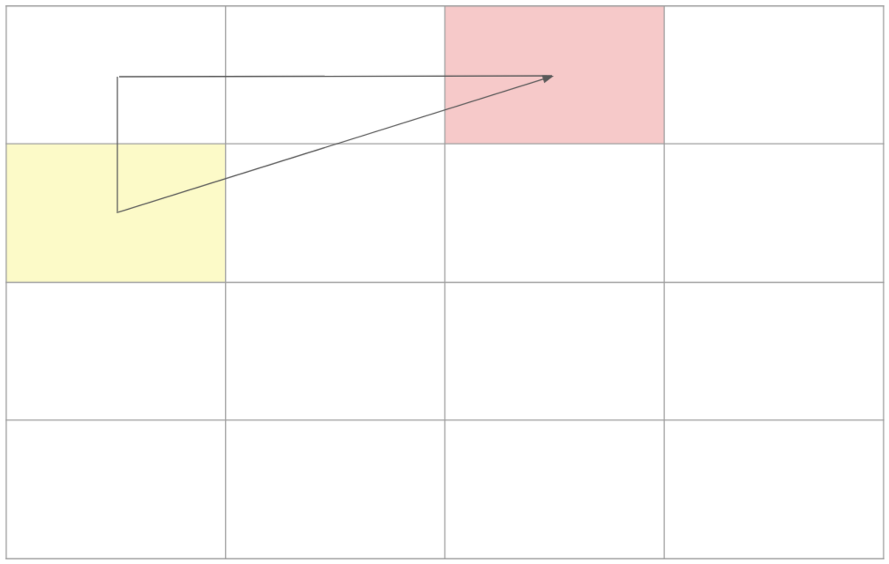


Fig 6.2: *Plot of the model's history.*

Since the output of the model is a confidence score for each grid cell, the predicted cell and the actual cell can be used to generate an estimate of the Euclidean loss of the prediction. If we assume all the predictions and truth cells to be the coordinates of the center of the cell, the average distance between the two can be calculated as the Euclidean loss.



```
Average inaccuracy rate (in pixels): 43.245547569580516
```

Fig 6.1: *The Euclidean loss for the grid system is calculated by using the centers of each grid. While this may not give an accurate measure of the Euclidean loss for a particular input, the average Euclidean loss over a large dataset will result in a similar value. Note that the value 43.24 is lower than the expected value*

since correct guesses contribute 0 Euclidean loss, but the actual Euclidean loss is likely to be small and positive.

An additional approach made for the application was to train two different models. One model was trained on processed ROIs that were constricted within a small region, whereas the other was trained on images where the eye moved to all parts of the frame. This also generated slightly blurry images, and caused higher inaccuracies because of the wide lateral distance causing larger errors. However, it provides a larger range for tracking the user's eyes.

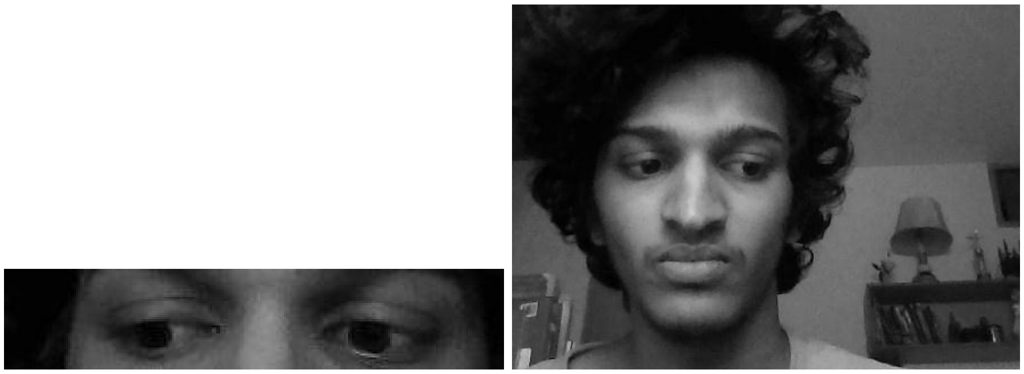


Fig 6.2: *Images from the two approaches mentioned. The left image's approach offers higher accuracy with stability, but is restricted to images from that region of interest. The right image offers a much larger span of the frame for good results, but are more erroneous.*

7. Testing

Once the model was trained, it was tested using a small HTML and JavaScript program that picks a random grid cell and highlights it if the user is looking at the correct cell. An animated GIF of the same can be found below.

<insert gif>

The main aim of the application was to design a mechanism to generate a heatmap / confusion matrix of the various grid cells in the time of the application's run. Once the program ends, it returns a heatmap based on the amount of time spent looking at each cell.

<insert heatmap>

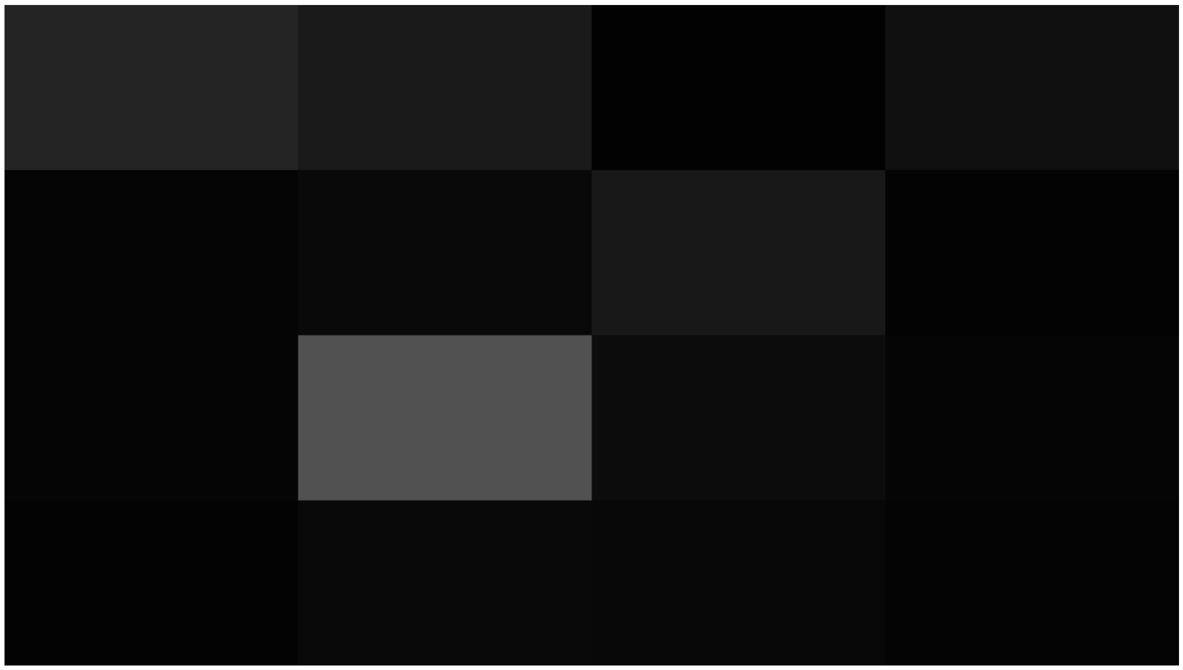


Fig 7.1: *An example of a heatmap (unlabelled) showing the cells of the grid proportional to the time the user's eyes spent looking at them.*

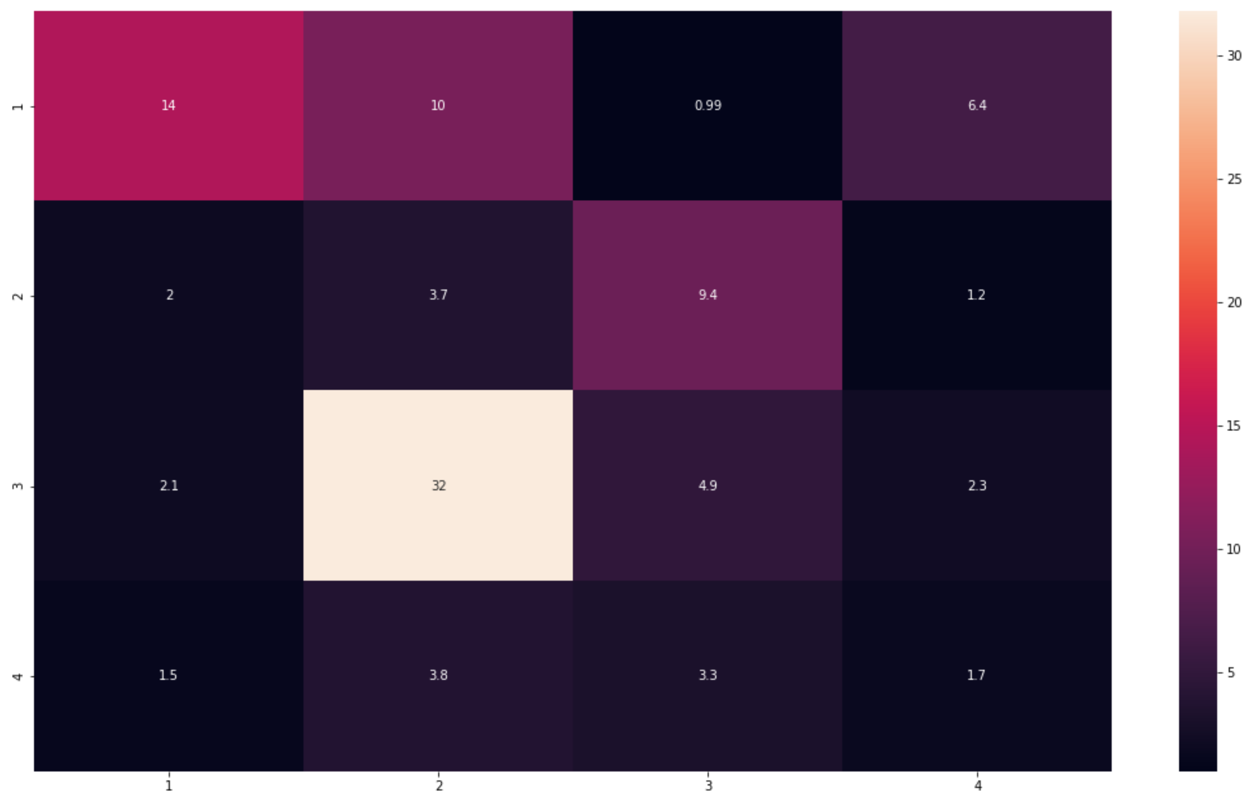


Fig 7.2: *The same heatmap as Fig 7.1, but visualized as a confusion matrix showing the percentage of time spent on each grid cell.*

8. Comparison with Existing Approaches

9. Future Work

10. References