Assignment 1

Section I

In this section, we create a function (we called prob1()) that takes a given hour and minute as input and outputs the image of a clock, similar to examples of figure 1.

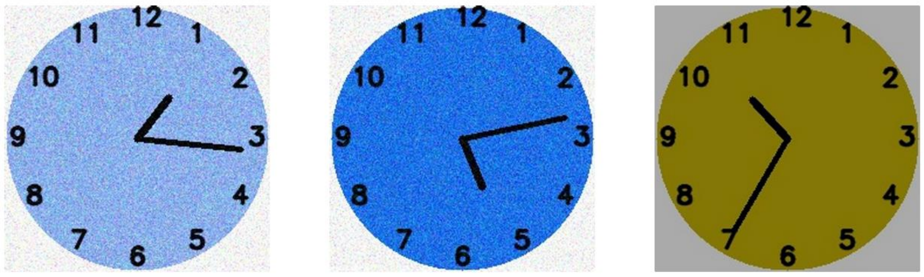


Fig.1 Examples of clocks

The first thing that we need to define is how to obtain the (x, y) coordinates for the hour and minute hand. There are (360/60 →) 6 degrees every minute and there are (360/12 →) 30 degrees every hour. Thus, we need to multiply the minutes by 6 and hours by 30. Since the 0 degree starts at 3 o'clock , we need to add 270 (which is the correspondent degree for 12 o'clock). Lastly, as we need to have degrees that are not greater than 359 we use **math.fmod(..., 360)**. While for minutes we do not need any extra steps, in the case of the hour, we can notice that in the analog clock the hour hand is influenced by the minutes and there are 60 minutes in an hour, and the hour hand moves 30 degrees every 60 minutes, that means that the hour hand moves 30 / 60, or 1/2 degrees every minute, as a result we need to add minute/2 to the hour.

```
minute_angle = math.fmod(minute * 6 + 270, 360)
hour_angle = math.fmod((hour * 30) + (minute / 2) + 270, 360)
```

Now that we have obtained the angles, we can use simple trigonometry to obtain the (x,y) coordinates for the minute and hour. For example, for the minute x coordinate, we add the center (which is 113) to the minute length (90) for the hand to start from the center of the image and multiply that for the cosine of the minute angle in radians. We do the same process for the y coordinate but we use the sine instead of the cosine. For the hour, we can do similarly.

```
minute_x = int(center[0] + minute_hand_length * math.cos(minute_angle * math.pi / 180))
minute_y = int(center[1] + minute_hand_length * math.sin(minute_angle * math.pi / 180))
```

Then, we also need to put the hour text in the clock, and to do so we need to take into account the fact that 0 degree is three o'clock. We basically start from the 3 o'clock for the 0 degree and go up to 14 but then we use the modulo operator to ensure that we get 1 and 2 and not 13 and 14 for the text. (more details in the code).

```
h = 2
for i in range(len(hours_coord)):
    h += 1
    if h > 12:
       h = h % 12
```

Another important aspect is the color. We want the clock to have a different color for the background and the foreground. If we simply use the == comparison we might obtain very similar colors, for example (255,255,255) and (255,254,255) would be regarded as different but they are actually so similar they look the same. To circumvent this conundrum we calculate and compare the intensities. We compute the intensities using ITU-R BT.601.

```
0.299 * color[2] + 0.587 * color[1] + 0.114 * color[0]
```

Images can also have noise. To take into account this we use **np.random.normal()** with mean zero and the standard deviation is randomly chose between 0 (no noise) and 1.

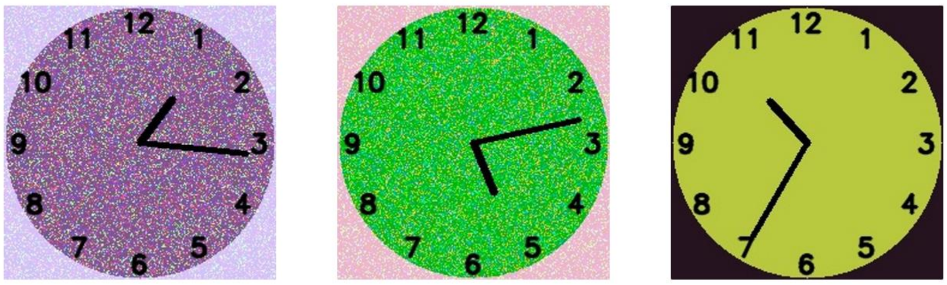


Fig.2 Samples from prob1()

In our implementation, there is an additional option in the function prob1() which is set to False by default, called “interactive”, if set to True the user will be asked to interactively input an hour and minute and a clock will be displayed in the screen.

Section II

In this section, we design a model that takes as input an image and predicts the hour and minute of the clock. There are several possible alternatives to solve the problem. Among them, we chose to train a multi-label model for classifying the hour and the minute. If we approached it as a multi-class problem, we would have had to deal with 720 classes, which would have made the task considerably more challenging due to the requirement for a large amount of data. Instead, we implemented a two-headed architecture (consisting of fully connected layers), one for the hour and one for the minutes. This configuration results in 12 classes for the hour in one head and 60 classes for the minutes in the other head, effectively addressing the problem as a multi-label task.

Since the clock images have simple patterns, we opted not to use any pre-trained models that might be designed for more complex tasks with intricate patterns. Instead, we employed a simple convolutional neural network architecture. Specifically, each convolution layer has kernels of size 2, a stride of 1, and padding of 1.

In the experiments detailed in section III, we evaluated the performance of various architectures, with a particular emphasis on the number of convolutional layers. Surprisingly, even a single convolution layer exhibited decent performance, underscoring the effectiveness of simplicity in this context. We identified an architecture that outperformed the others. The most optimal architecture (figure 3) consists of two convolutional layers, a fully connected layer, followed by two heads (fully connected layers) responsible for classifying the hour and the minutes, respectively.

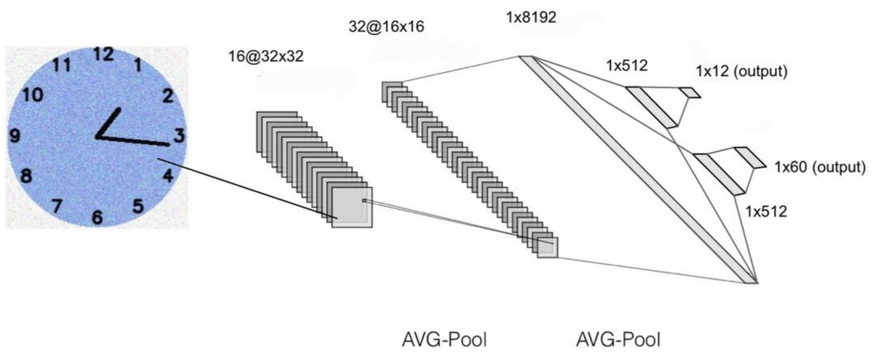


Fig3. Model architecture

Section III

In this section, we develop a code to train the model. The first step is to load the images and labels and then create a custom dataset to pair them in order to load them to the data loader. We load the images using PIL instead of cv2 because it is known to work better with torch transformations. After we load the images we convert them to torch tensors and center crop them to the size of 64x64.

```
my_transform = T.Compose(
        [  # Convert the image array to tensor
            T.ToTensor(),
            # Center crop the image
            T.CenterCrop((64, 64)),
        ]
    )
```

The images the model see are depicted in figure 4. We chose to show the model only the center because the hour and minute are determined by the position of the hands in relation to the center. The needed visual distance depends on heuristics but we found the 64x64 to work well in this case.

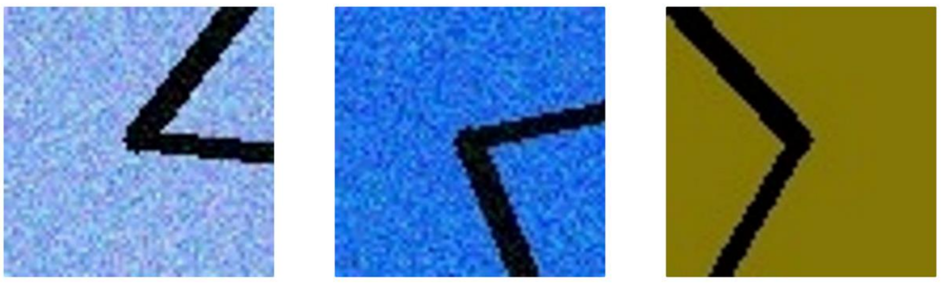


Fig.4 Images input to the model

For the custom dataset we use the torch Dataset library. We follow the standard procedure and define the init, len and getitem functions for the custom dataset class. The core difference is that we choose to save the data using a dict to easily access the data. We then load the data to the torch Dataloader to obtain batches for more efficient training.

```
  sample = dict()
  sample["img"] = self.target_img[idx]
  sample["hour"] = self.target_hour[idx]
  sample["minute"] = self.target_minute[idx]
```

One important implementation detail when loading the images is that we need the list of loaded images, hour list and minute list. They must be input to My_Dataset().

```
train_data = My_Dataset(img_list, hour_labels, minute_labels, transform=my_transform)
```

We define the optimizer as stochastic gradient descent with a learning rate of 0.01. We did not conduct any hyperparameter tuning since the performance on the validation set was deemed adequate. Additionally, we use cross entropy loss as our loss function, given that we defined the task as classification. We aggregate losses for multi label learning. Following the standard training structure in Torch, we load the model onto the device, set it to train mode, zero out gradients at each iteration to prevent accumulation, load images and labels onto the device, compute and aggregate losses, perform backpropagation, and finally update the parameters.

Experiments

In this subsection, we conducted experiments to determine the optimal number of layers for our architecture, as well as whether it is more advantageous to incorporate one fully connected layer with two heads or simply utilize two heads. We explored varying numbers of convolutional (CNN) layers, ranging from 1 to 4. For each configuration, we tested both with a fully connected layer followed by two heads, and with just the two heads.

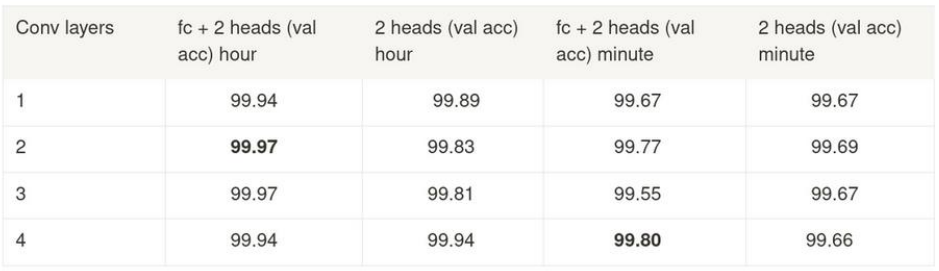
The train dataset comprises 10,800 samples, providing 15 images for each unique hour-minute combination. Similarly, the validation dataset consists of 3,600 samples, with 5 images allocated to each combination. The noise incorporated into the dataset is randomly selected, (refer to section I).

We trained each model for 100 epochs and saved the model based on the highest validation accuracy. When a 5 minute error is allowed for the prediction (table I), the best model is one with two convolution layers and a fully connected layer followed by 2 heads. The rest of the models also have a good performance, even one with only one convolution layer.

Conv layers	fc + 2 heads (val acc)	2 heads (val acc)
1	99.81	99.78
2	**99.95**	99.81
3	99.75	99.62
4	99.89	99.92

Table I. Accuracy (5 min error)

The accuracy for the hour and minute (table II), also showed that in the case of the hour the best model is the one with two convolution layers and a fully connected layer followed by 2 heads but for the minutes the best model was one with four convolution layers.



Conv layers	fc + 2 heads (val acc) hour	2 heads (val acc) hour	fc + 2 heads (val acc) minute	2 heads (val acc) minute
1	99.94	99.89	99.67	99.67
2	**99.97**	99.83	99.77	99.69
3	99.97	99.81	99.55	99.67
4	99.94	99.94	**99.80**	99.66

Table II. Accuracy (exact)

The training epochs required for the model vary depending on the number of convolution layers. As we can see in figure 4 illustrates the error plot for the best-performing model, the model requires only a few epochs to reach a point where the losses are decreasing slowly. In our experiments, the model achieved 99% accuracy after 11 epochs. The validation loss continues to decrease until the epoch 98 but the improvement is not significant after the epoch number 20.

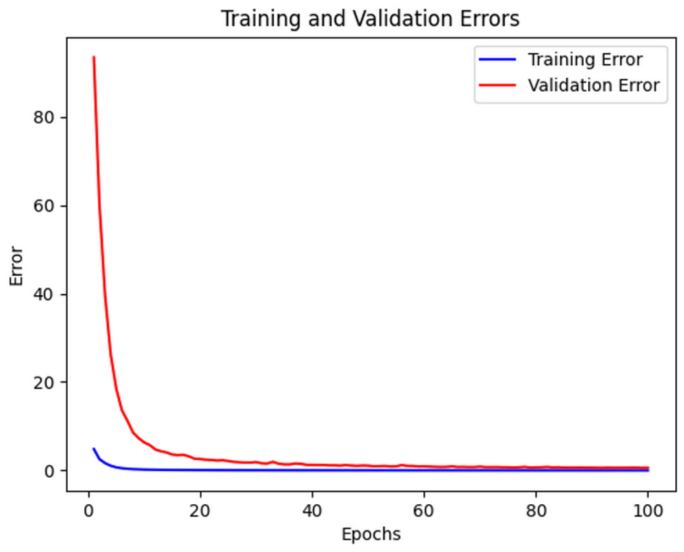


Fig 5. Error plot or best model

Section IV

In this section, we develop a code for testing our model that takes as input the filename of the image and outputs the predicted hour and minute.

The function name is prob4() and it loads the image using PIL, then it transform the image to convert it to a tensor and center crop the image. The model parameters are loaded and the mode is load to the device and set to test. As the image is a single image but the model takes batches, we resize the image using unsqueeze to add the dimension. Then the image is loaded to device and input to the model which outputs the predicted hour and minute. The hour and minute predictions are probabilities for each class and we return the ones with the highest probability.

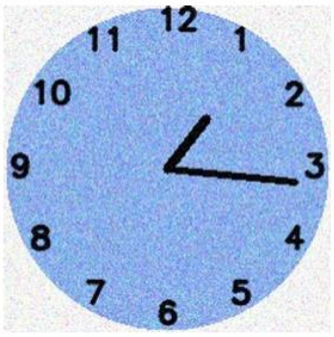
More details regarding this section can be found in the code.

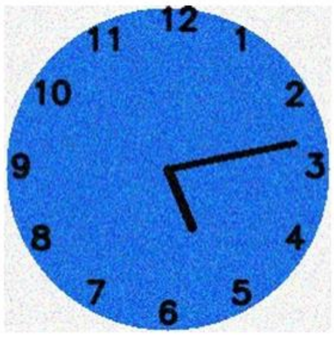
Section IV

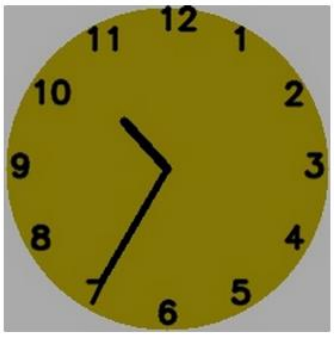
In this section, we comprehensively evaluate the performance of the model, aiming to discern both its accomplishments and areas for improvement. To achieve a robust assessment, we employ two distinct evaluation approaches one for individual inputs and one for varying sample sizes.

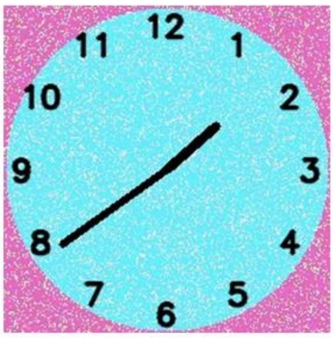
The first evaluation method involves testing the model with individual inputs, each characterized by varying levels of noise. Specifically, we consider three noise levels which are no noise, a standard deviation of 1, and a standard deviation of 10. This allows us to observe how the performance of the model is affected by different degrees of noise interference.

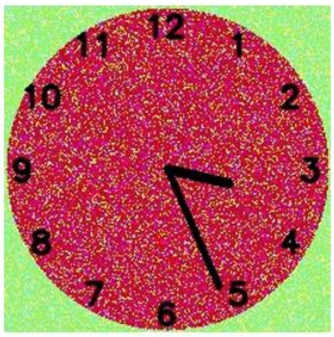
Most of the testing images were accurately predicted. However, the model struggles to accurately predict the time when the clock is black or dark in a color similar to the hands. It seems that when the image is too dark the model predicts the hour as 10 and minute as 4 (at least in the set used for conducting the experiments).

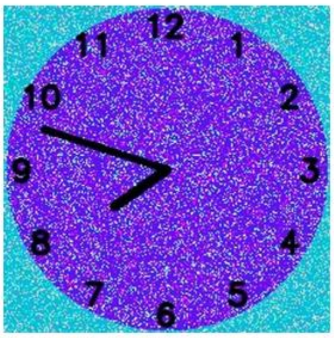

Predicted: 1h 16m

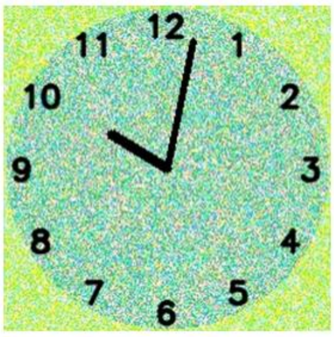

Predicted: 5h 13m

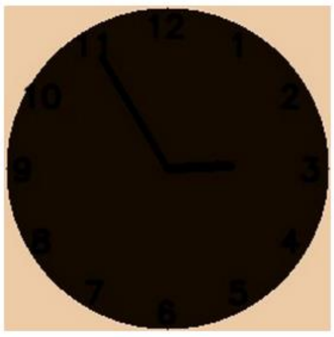

Predicted: 10h 35m

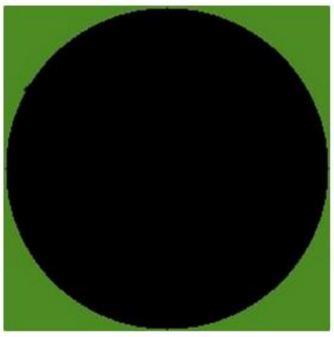

Predicted: 1h 39m (std: 1)


Predicted: 3h 26m (std: 1)

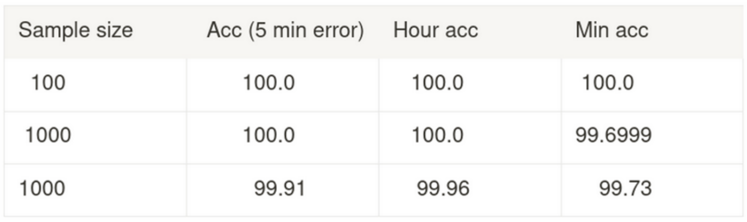

Predicted: 7h 48m (std: 10)


Predicted: 10h 2m (std: 10)


Predicted: 10h 4 m (2:55)


Predicted: 10h 4 m (11:12)

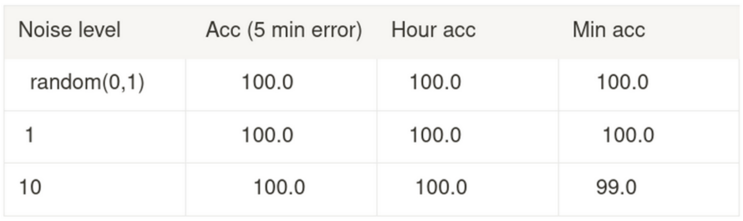
The second evaluation approach broadens the scope by examining the performance of the model across varying test sample sizes. We conduct experiments with 100, 1000, and 10000 samples, providing a broader perspective on the scalability and generalization. In table III, we notice that as the data size increases both the minute and hour accuracy decreases, this might be due to the higher likelihood of clocks that have a foreground color similar to the hand color.



Sample size	Acc (5 min error)	Hour acc	Min acc
100	100.0	100.0	100.0
1000	100.0	100.0	99.6999
1000	99.91	99.96	99.73

Table III. Varying data sample size for test set

For the 100 sample evaluation, we introduce random noise, mirroring the conditions observed during training and validation and we also introduced noise to all images with standard deviations of 1 and 10, to gauge robustness to different noise levels. In table IV, we observe that the noise affects the minutes as it increases.



Noise level	Acc (5 min error)	Hour acc	Min acc
random(0,1)	100.0	100.0	100.0
1	100.0	100.0	100.0
10	100.0	100.0	99.0

Table IV. Varying noise level for test set

Discussion

The approach of reducing the complexity of the problem through effective cropping and centering of the image has proven to be highly beneficial for the model's learning process. This strategy enables simpler architectures to grasp underlying patterns more efficiently and accelerates the learning curve.

While noise in images can introduce challenges, particularly in minute predictions, the model demonstrates a notable level of robustness in predicting the hour. Even in the presence of some degree of noise, the accuracy of the hour prediction remains relatively stable. It is worth noting that the success of the model's predictions is contingent on the color differentiation between the hands and the foreground. When the colors are significantly similar, it may pose a considerable challenge for the model to make accurate predictions.