Automatic Image to Tactile Format Conversion

Aatreyi Pranavbhai Mehta
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
aatreyipranavbhaime@cmail.carleton.ca

April 15, 2023

Abstract

The present report depicts an overall execution, performance and results produced by two machine learning models, namely the RGB model and the Channel-wise model. It provides an in-depth view on the results produced on the synthetically generated diverse data of 5 distinct kinds: horizontal bar charts, vertical bar charts, bezier curves, scatter plots and polygons. Furthermore, it discusses the technical details, setup and procedure of data generation, adding data diversity, training as well as testing on the data set.

1 Introduction

Major amount of information in the current world is provided graphically. However, as per a recent estimate, about 2.2 billion people around the world suffer from some kind of visual impairment[8]. The graphical information that is being used lately is not accessible to these visually impaired individuals, instead they depend on tactile diagrams. Tactile diagrams are the ones that have raised lines, elevated graphics, texture and they are supposed to be touched to understand rather than being seen[6]. Because instructional materials at education institutions usually convey information using diagrams and geometric figures[7], tactile graphics are thought to be a crucial component for students seeking a quality education in various fields. Braille transcribers frequently create their own tactile images by hand as part of the creation of Braille textbooks. In some instances, automated procedures using different software programs help in tactile image generation[5]. Our project walks on a similar path and helps in RGB to tactile format conversion through two deep machine learning models using Generative Adversarial Network(GAN). The goal of this project is to effectively generate variety of diverse data through random point generation and then convert these RGB plots to gray-scale tactile images using a Pix2Pix architecture, while also providing a channel-wise representation of the same using the channel-wise architecture. This project works as a pipeline where the synthetic RGB plots are first converted to gray-scale tactile images with the help of the RGB model(having pix2pix architecture) and later these gray-scale outputs of the RGB Model are provided as an input to the Channel-wise model, and it produces a channel-wise output of the graphs covering all the necessary aspects, i.e. the axes of the graphs, the content of the graphs and the grids present on the graphs.

2 Technical Details

The training and testing requirements for both the RGB as well as the Channel-wise model asks for an input source image and a target image. The source image is the one that will be produced using random point generation, it is divided in four categories: bar charts, bezier curves, scattered plots and polygons. The bar charts are further divided in two categories - vertical bar charts and horizontal bar charts. The target image is the image that replicates the supposed output of trained models, it appears like a sketch of the source image and is comprised of only the important components of the source image like the x and y axis, content and the grid lines.

2.1 Source and Target Generation for Both Models

The major goal while producing the synthetic source images is to add specifications and diversity in a way that it closely replicates the real world plots and graphs. For generating vertical bar charts and horizontal bar charts, we will require individual programming codes for each as they have different content orientations, x/y values, grid line orientation, etc. On the contrary, the curves, polygons and scattered plots will all be generated with the help of a single programming code. The programming language we have used for the current project is Python and the explanation of source and target image generation for all of the above mentioned categories is provided in the upcoming sub-sections.

2.1.1 Generating Vertical Bar charts

To begin with, we have created a file called "bar_generator_vertical.py" that holds all the plot properties, characteristics and specifications. Within it, we begin with importing the python packages that will be utilized while programming. random is one such package that will help with random number generation. We are using the plotly library for generating the bar charts. Initially we will define the PLOT_ORDER and SIZEREFS, both of which will contain key value pairs to store the order in which the plots will be displayed and the reference value for size element in the charts, respectively. We will then define generate_metadata function that will be in-charge of generating random metadata for a graph with a certain number of bars and groups. We will further sort the plot ordering by taking a list of graphs as input and produce data for multiple bar graphs. We also define generate_marker_colors function to generate colors for the bars in the graphs. This function is defined such that the bars will either produce a single color or multiple colors by selecting a random number between 1 and 255. There is also a compute_scatter_width function that computes the size reference of circle scatter points based on the size of the graphs and number of bars and bar groups. Further, we define other random specifications like the bar width, bar gaps, tick size, the random background colors and the paper color for the graphs. For the target images, the bar charts are converted to circle bar charts. The points for these circle-bar charts will be generated using generate_circle_bars function. The code generates points to depict each bar as a circle for a chart by iterating through each group and bar within it. We have also defined styles for these graph like the color of the target plots, minimum and maximum values for axis, bar gaps, number of bars and bar groups.

Now, we start plotting the source graphs that have five different image dimensions including 512x512, 1024x512, 760x512, 512x1024, 512x760. As seen in Figure 1, we first define write_source_data() that has 5 parameters - data, filepath, figsize, draw_grid and

```
#Plotting the source domain(vertical barcharts) for 512x512 image dimension
def write_source_data(data, filepath, figsize=(512, 512), draw_grid=False, tick_step=10):

    linewidth = random.uniform(1,5)   #generating linewidth of random thickness for every image
    fig = go.Figure()

    #initializing word, that selects random string of alphabets every time
    alphabet = string.ascii_lowercase
    word = ''.join(random.choice(alphabet) for i in range(randint(2,3)))

    #setting the x and y values for vertical barchart, that will be exactly reverse as that of horizontal barcharts
    for r in range(len(data["y_values"])):
        fig.add_trace(go.Bar(x=data["x_values"],
                        y=data["y_values"][r],
                        #orientation=random.choice(open("orientation.txt","r").readline().split()),
                        name=word,
                        #name=random.choice(open("legend.txt","r").readline().split()),
                        marker_color=data["marker_colors_rgba"][r] if len(data["y_values"]) > 1 else data["marker_colors_rgba"],
                        marker_line_width=2
                        ))
```

Figure 1: Initial plotting function for source images

tick_step. The figsize here is 512x512, which signifies it will generate source images of dimension 512x512. Now, we initialize word that generates random strings of alphabet. We have further set the x and y values for the vertical bar charts and declared the marker width. Moving further, within the figure layout, we set the margin, x and y axis range, bar gap, etc. While setting the x and y axis titles, we declare them as:

1. xaxis_title=random.choice(open("label.txt","r").readline().split()),
2. yaxis_title=random.choice(open("label.txt","r").readline().split()),

These lines of code will help us to pick a random string of alphabets from the file "label.txt" and for every image a new axis title will be generated. We then set the x and y axis properties like the axis color and axis width. The axis width is also set to random and it will select a random number between 1 and 5 every time. For setting the layout for legend, we have set the orientation by selecting a random word every time from file "orientation.txt" that has 2 values: h representing horizontal orientation and v representing vertical orientation. We then open and read the content of x.txt and y.txt to set the x anchor and y anchor positioning so that the legend can randomly appear on top left corner, top right corner, bottom left or bottom right corner for every source image. The fonts for legend include verdana, arial, sans-serif, overpass, balto and roboto, which will be picked randomly for each image from the font.txt file every time. The legend layout structure can be seen in Figure 2. Coming to the tick labels, in Figure 3, one can see that the x-axis tick labels on a plot are assigned randomly generated strings of 2–3 letters from the alphabet list. The random strings are produced using a for loop that executes ten times and concatenates two to three random characters from the alphabet into a string. The tick labels are then given these strings and integers from 0 to 9 are used as the tick positions.

The overall font size and font style for the source image graphs are also set to random, and a different font size as well as different font style will be generated for each graph. The chart title, however, has been set separately such that it is bigger in font as compared to the

```
with open("x.txt","r") as f:
    x_options = f.readline().split()

with open("y.txt","r") as f:
    y_options = f.readline().split()

#layout for legend
fig.update_layout(legend=dict(
    orientation=random.choice(open("orientation.txt","r").readline().split()),
    font_family=random.choice(open("font.txt","r").readline().split()),
    x=float(random.choice(x_options)),
    y=float(random.choice(y_options))
))
```

Figure 2: Layout to set properties for the chart legend for source image

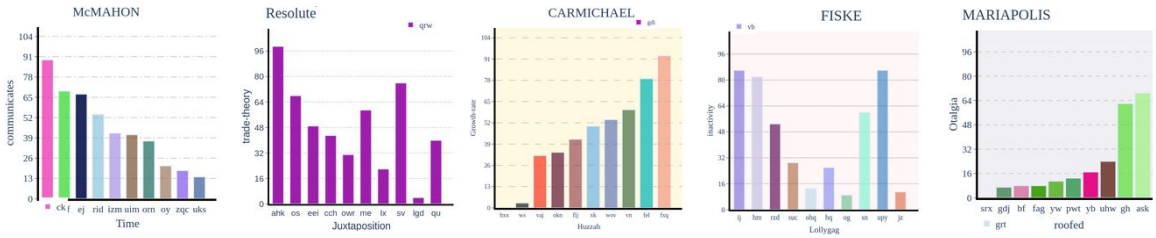
rest of the texts in the graph and generates different title each time. Since the graphs here are vertical bar charts, we will set the horizontal grid lines for it. If the showgrid parameter is set to False, no grids will be produced but if it is set to True, grids of type solid, dot, dash, longdash, dashdot or longdashdot will be produced based on which of these words is randomly picked from "grid.txt" file. Samples of some of these grid types are portrayed in Figure 4. After assigning all of the properties mentioned above, the source graph is saved as an image to a file. This is where the write_source_data() function ends.

The channelwise target generation start with defining write_circle_target_data() function with 5 parameters: data, filepath, figsize, draw_grid and tick_step. The figsize will again be 512x512 as the target will replicate the source's dimension. In a list of dictionaries called data, the code in Figure 5 generates a scatter plot using circles as markers for each item. With use of x and y values and a width value, each dictionary forms a circle. go.Scatter is used to create these scatter traces, with the marker size that is determined by the widths of the circles and aspect ratio of the graph. The size reference will change for each dimension based on the marker gaps, for 512x512 the sizeref is 1. Each trace is then added to the graph via the fig.add trace() method, and the resulting plot will display circular markers for every item in the data set. On the top of this, we set the properties like the background color, height and width of the plot, tickmarks, range of the axes, linewidth, etc. There is no need to set the axes labels, chart titles, tick labels or legend for the target images. The code in Figure 6 will save a plot as three different images, displaying the axes, the circle markers i.e. the bar content and the grid lines. The first image just displays the axes in which there are no grid lines or data points present. The second image displays just the grid lines if draw_grid is set to True, whereas the third image displays the bar content and data points. The fp_parts variable defines the file path, extension and it is used to create the file names for the images.

Finally comes the write_circle_target_data1() function, for plotting the gray-scale pix2pix target images for 512x512 image dimension. It will also have the same 5 parameters as

```
alphabet = string.ascii_lowercase

ticktext = []   #initializing tick text for ticks
for i in range(10):
    word = ''.join(random.choice(alphabet) for i in range(randint(2,3)))
    ticktext.append(word)

fig.update_layout(
    xaxis = dict(
    tickmode = 'array',
    tickvals = [0,1,2,3,4,5,6,7,8,9],  #positions of the tick step
    #tickvals = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14],
    ticktext = ticktext,

))
```

Figure 3: Layout for setting tick and tick labels for source image



Grid type: dash-dotted | Grid type: dotted | Grid type: dash | Grid type: solid | Grid type: lond-dash

Figure 4: Examples of the included grid types for source images

channel-wise target function i.e. data, filepath, figsize, draw_grid and tick_step. Next, it has all the same properties, specifications, most of functions and figure layout as the channel-wise target. The only portion of code that will change can be seen in Figure 7, which mainly deals with creating the plot, setting its style and saving the image. fig.update_traces() function in the code snippet helps in setting the marker width, marker color, marker type that modifies the scatter plot's appearance and makes the trace in figure visible. If draw_grid is set to True, the code updates images with grid lines, otherwise not. The figure is then saved as a PNG image file using the file name, format, and size supplied by pio.write_image().

In the same file, bar_generator_vertical.py, we have created other functions like write _source_data_elongated(), write_circle_target_data_elongated() and write_circle_target_data _elongated1() for generating image dimension with ratio 1024x512 and also functions write _source_data_1(), write_circle_target_data_1(), write_circle_target _data1_1() for generating image dimension 760x512. These functions will decide and set the properties for the source, channel-wise target and gray-scale target images for 2:1 and 1.5:1 aspect ratios. All the

```
def write_circle_target_data(data, filepath, figsize=(512, 512), draw_grid=False, tick_step=10):
    fig = go.Figure()
    for i in range(len(data)):
        for j in range(len(data[i])):
            fig.add_trace(go.Scatter(x=data[i][j]["x"],
                        y=data[i][j]["y"],
                        marker = {"size":np.array(data[i][j]["widths"])*np.sqrt(figsize[1]/figsize[0]),
            "sizemode":'diameter',
            "sizeref": 1,   #size reference for 512x512 images
                                },
                        ))
```

Figure 5: Initial plotting function defined for channel-wise target of 512x512 image dimension

```
#saving the axes part as an image
pio.write_image(fig=fig, file=f"{fp_parts[0]}_axes.{fp_parts[1]}", format="png", width=figsize[0], height=figsize[1])

#defining grid properties for when grids are produced
if draw_grid:
    fig.update_yaxes(showgrid=True, gridcolor='black', griddash='dash', gridwidth=1)
fig.update_layout(xaxis={"linecolor": 'white'}, yaxis={"linecolor": 'white', "ticklen": 0, "tickwidth": 0})
#saving the grid part as an image
pio.write_image(fig=fig, file=f"{fp_parts[0]}_grids.{fp_parts[1]}", format="png", width=figsize[0], height=figsize[1])

fig.update_traces(mode='markers', marker_line_width=2, marker_color="white", marker_line_color="black", visible=True)
fig.update_layout(yaxis={"showgrid":False})
#saving the bar content an an image
pio.write_image(fig=fig, file=f"{fp_parts[0]}_content.{fp_parts[1]}", format="png", width=figsize[0], height=figsize[1])
```

Figure 6: Saving the axes, content and grids individually as different files

details of these functions will be same as the write_source_data(), write_circle_target_data(), write_circle_target_data1() functions except for the size reference for the go.Scatter portions of channel-wise and gray-scale target. The sizeref for 1024x512 targets will be 0.35 and that for 760x512 aspect ratio is 0.80. However, the sizeref for 1024x512 ratio is a bit large which can result in overlapping so we have made changes in the number of markers by reducing it in the channel-wise target and gray-scale target for 1024x512 aspect ratio. We have added variable n = 2 as seen in Figure 8. The number of markers plotted on the graph is controlled by slicing the x and y arrays in the data list using [::n]. By setting n=2, we have decreased the number of circle markers that will be produced for the channel-wise and grayscale target and the sizeref 0.35 maintains the size of each marker so that there is not a lot of variations in the data set and training can happen in an efficient manner. The 1024x512 target graphs will end up looking similar to the example shown in Figure 9. However, this change only applies to 2:1 ratio and not for 1.5:1 ratio.

For the purpose of including more variation, we have also considered image dimensions 512x1024 and 512x760. These aspect ratios will generate tall and narrow width bar charts which can result in overlapping of bars. To avoid this, we have created an-

```
fig.update_traces(mode='markers', marker_line_width=2, marker_color="white", marker_line_color="black", visible=True)
if draw_grid:
    fig.update_yaxes(showgrid=True, gridcolor='black', griddash='dot', gridwidth=1)

#saving the rgb/grayscale output as an image
pio.write_image(fig=fig, file=f"{fp_parts[0]}.{fp_parts[1]}", format="png", width=figsize[0], height=figsize[1])
```

Figure 7: The code snippet to save the RGB model's target

```
def write_circle_target_data1_elongated(data, filepath, figsize=(1024, 512), draw_grid=False, tick_step=10):
    fig = go.Figure()
    for i in range(len(data)):
        for j in range(len(data[i])):
            n = 2 # change this value to control the number of markers

            fig.add_trace(go.Scatter(x=data[i][j]["x"][::n],
                        y=data[i][j]["y"][::n],
                        marker = {"size":np.array(data[i][j]["widths"])*np.sqrt(figsize[1]/figsize[0]),
                        "sizemode":'diameter',
                        "sizeref": 0.35,
                                },
                        ))
```

Figure 8: The code snippet to control the marker for target for 2:1 aspect ratio

other file by name "bar_generator_vertical2.py". The whole format of this file will be same as bar_generator_vertical.py file, the change will just be in the generate_metadata() function and the functions generating the source and target images. The generate_metadata() function in Figure 10 creates a list x used as the x-coordinates of the bars in a graph and odd numbers are distributed evenly across the list to leave a gap between the bars. The num_bars variable controls how many elements there are in the list. The list x will contain five odd values for example: 1, 3, 5, 7, 9, since num_bars is set 5. So basically, we have reduced the number of bars that will be produced and just plotted them at alternate tick positions(refer Figure 11 for better understanding). Other than this variation, there will be a write_source_data_2() function that will generate source images for 512x1024 dimension, write_circle_target_data_2() for plotting 512x1024 channel-

RGB Model's target Channel-wise model's target

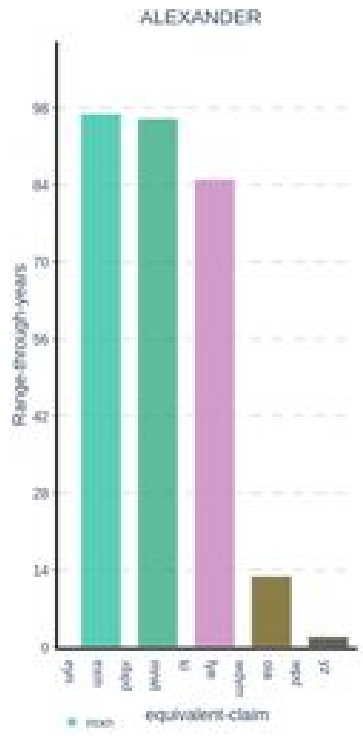
Figure 9: Sample of 1024x512 grayscale target and channel-wise target

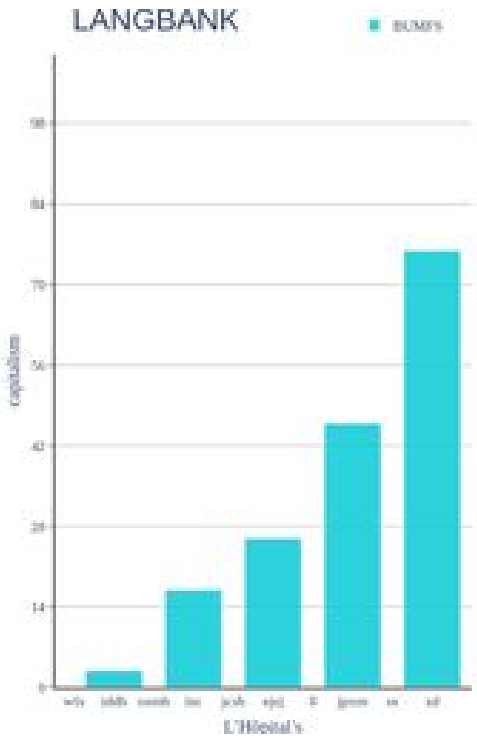
wise target and write_circle_target_data1_2() for plotting 512x1024 gray-scale target. These will be same in format as the functions generating source and targets for 512x512 in bar_generator_vertical.py except for the sizeref, which will be 0.72 for 512x1024 targets for maintaining appropriate gaps between bars. Similarly, the write_source_data_3(), write_circle _target_data_3(), write_circle_target_data1_3() functions for 512x760 dimension will replicate the functions generating source and targets for 512x512 in bar_generator_vertical.py except for the sizeref, which will be 0.85 for 512x760 targets. All other layouts and functions will remain same for all these aspect ratios.

```
def generate_metadata(min_y = 0, max_y = 100):
    num_bars = 5
    x = list(range(1, num_bars*2, 2))

    num_groups = np.random.randint(low=1, high=1+1)

    # Generate random values for bars
    y = np.random.randint(low = min_y, high = max_y + 1, size=(num_groups, num_bars))

    # Sort y plots accordingly
    sort_plots(y)

    return {'x': x, 'y': y}
```

Figure 10: The modified generate_metadata() for 1:2 and 1:1.5 aspect ratios

For producing these source and target images that we defined all the characteristics and properties for, we will create a file draw_bar_vertical.py. This file begins with importing the functions from bar_generator_vertical.py for plotting the layouts for source and target images for 1:1 image ratio, 2:1 image ratio and 1.5:1 image ratio. It will further import the maskgen and postprocessing function from utils module which will respectively resize the channel-wise target and source images to 256x256 dimension. Next we have defined NUM_SAMPLES that will generate 1000 samples for each mentioned dimension and set the grid probability as 0.8 out of 1, that is 80 percent produced images will have grids. Later, we make 3 directories in the data folder: source for saving the source images, tactile for storing the channel-wise target images and tactile1 for the gray-scale target images. We have defined 3 figure sizes: 512x512, 1024x512 and 760x512 and later called the respective source and target plotting functions, resized them and saved the resulting images in appropriate folders. We have also created a draw_bar_vertical2.py for generating 512x1024 and 512x760 dimension which works in the similar manner as draw_bar_vertical.py. It will import the functions from bar_generator_vertical2.py for replicating the layouts for source and target images for 1:2 image ratio and 1:1.5 image ratio and then resize and save the source and target in their respective folders. This brings us to the end of generating vertical bar charts. Below, in Figures 12, 13, 14, 15, 16 and Figure 17 one can find some samples of the types of vertical bar charts produced by the overall integrated code.





Aspect ratio- 1 : 2

Aspect ratio- 1 : 1.5

Figure 11: The bar charts produced with reduced number of bars by bar_generator_vertical2.py file

2.1.2 Generating Horizontal Bar charts

For the horizontal bar charts, the data will also be represented in form of rectangular bars for the source images and go.scatter() will be used to create circular bar charts for both RGB and channel-wise targets. The change here would majorly be in terms of orientation of the bar content and grids. The bars will now originate from left and extend to the right instead of the originating from bottom and extend to top as seen in vertical bar charts. The bars will be oriented horizontally and will have vertical grid lines. For this, we begin with a bar_generator_horizontal.py file that will have all the properties like background, bar gap, bar width, metadata generation, setting style of the plot, etc. similar to that as the vertical bar charts. We have 3 image aspect ratios in this file: 512x512, 512x1024 and 512x700. The function for plotting the source images for 512x512 ratio is defined write_source_data(). The difference that will be seen here, will be in the orientation of those bar charts. As specified before, the bars will run horizontally from left to right for displaying which we will need to change the values of the data points in use. All the x values in the functions seen in vertical bar charts will be switched with y in the functions that plot the source and even the target data for the horizontal bar charts. For instance, consider the fig.add_trace() function of write_source_data() in Figure 18, one can notice that all the x values are switched with y values and vice versa; even the orientation is changed to horizontal. The "xaxis = dict"

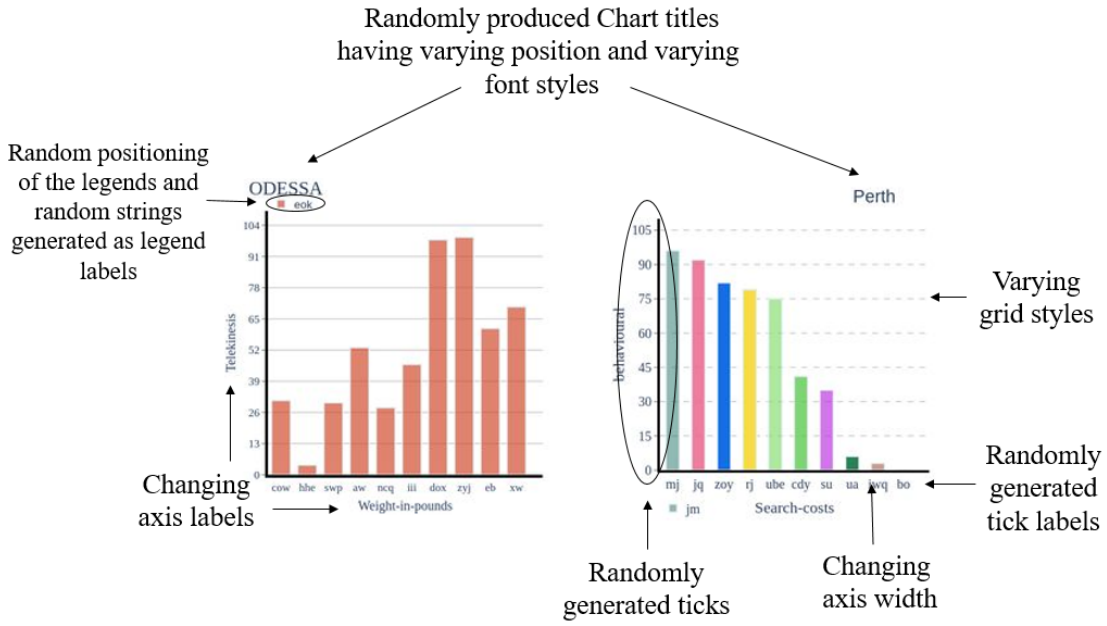


Figure 12: A representation of diversity in vertical bar charts

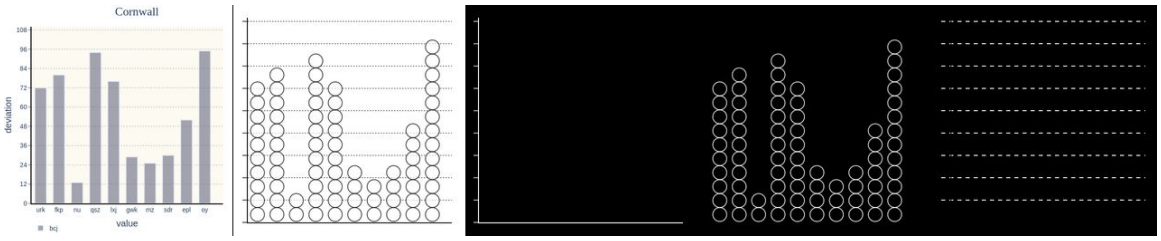


Figure 13: A sample of 512x512 dimension source(left), RGB target(centre) and channelwise target(right)

will change to "yaxis = dict" in fig.update_layout(), as seen in Figure 19. Even the x and y axis range for the source and targets will be switched. The final change for the source image would be including vertical grids for the bar charts. For this, we have changed the fig.update_yaxes() with fig.update_xaxes() as seen in Figure 20. These changes apply to all the functions plotting source images of other 2 image dimensions for this file.

The write_circle_target_data() for the channel-wise target will also be modified. The same changes that are applied to fig.add_trace() for source images, will be applicable to fig.add_trace(go.Scatter()) for the target images i.e. the x values will be switched with y values as seen in Figure 21 and the orientation of the grids is changed to vertical with the help of modifications shown in Figure 22. Similar changes apply for gray scale target, however the portion where the images are saved will obviously be different for gray-scale target images. The channel-wise targets will be saved as three individual images unlike the gray-scale target. The sizeref for all three dimensions will be distinct. The sizeref for 512x512 is 1, the same for 512x1024 and 512x700 is 0.35 and 0.82, respectively. The sizeref

Figure 14: A sample of 1024x512 dimension source(left), RGB target(centre) and channel-wise target(right)

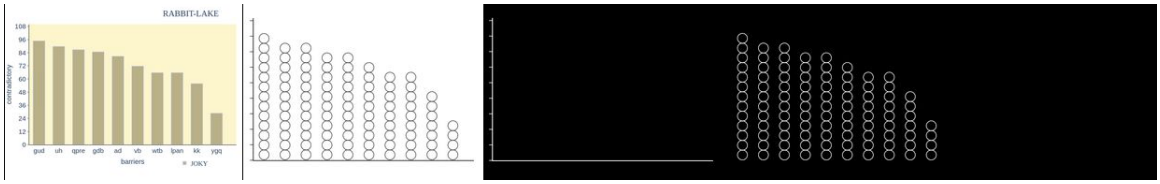


Figure 15: A sample of 760x512 dimension source(left), RGB target(centre) and channelwise target(right)

for 512x1024 is quite large and can lead to marker overlapping due to which we declare n=2 and use it in fig.add_trace() to reduce the number of markers so as to generate relevant targets. Each and every other function will remain same as bar_generator_vertical.py. We have further included 2 more image dimensions for horizontal charts(1024x512 and 760x512) in file bar_generator_horizontal2.py which will generate tall and narrow width graphs, hence we will reduce the number of bars and generate them on alternate ticks in a similar way we did in bar_generator_vertical2.py. The image dimension 1024x512 will have sizeref 0.71, 760x512 images will have a sizeref of 0.87 and all the other layouts, style setup, properties will be same as functions plotting the source and target for 1:1 aspect ratio 512x512 image dimension in bar_generator_horizontal.py.

There will also be files draw_bar_horizontal.py and draw_bar_horizontal2.py which will import the bar_generator_horizontal and bar_generator_horizontal2, respectively. Both then import the maskgen and postprocessing modules from utils. Finally, both of these files will create source, tactile and tactile1 folders, call source and target functions, resizes the source and channel-wise target images to 256x256 dimension and store them in appropriate folders. Below, the Figures 23 to Figure 27, represent some samples of the kind of data produced for the horizontal bar charts.

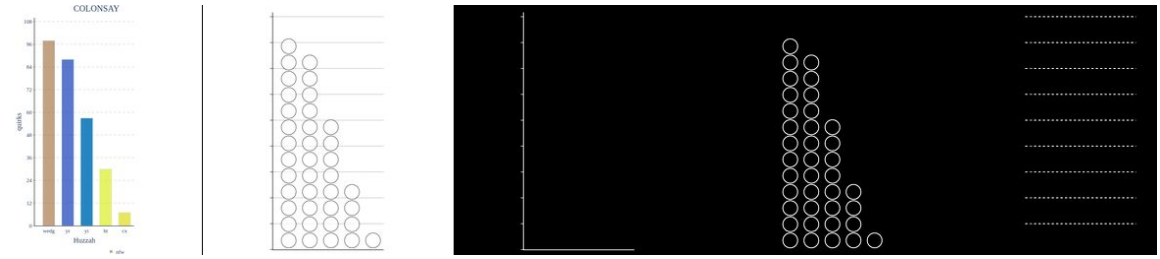


Figure 16: A sample of 512x1024 dimension source(left), RGB target(centre) and channel-wise target(right)

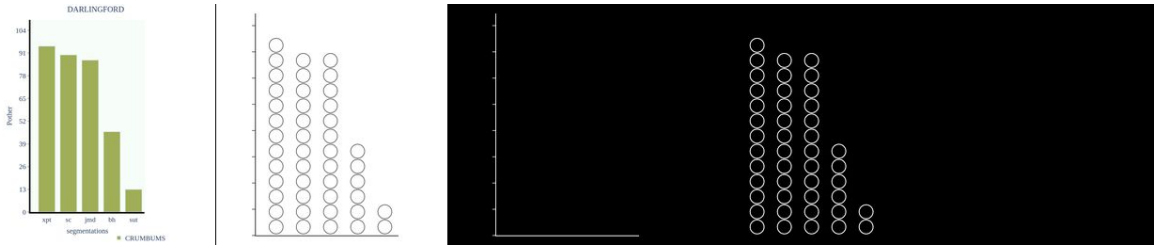


Figure 17: A sample of 512x760 dimension source(left), RGB target(centre) and channelwise target(right)

```
for r in range(len(data["y_values"])):
    fig.add_trace(go.Bar(y=data["x_values"],
                    x=data["y_values"][r],
                    orientation="h",    #horizontal orientation
                    width=0.7,
                    #orientation=random.choice(open("orientation.txt","r").readline().split()),
                    name=word,
                    #name=random.choice(open("legend.txt","r").readline().split()),
                    marker_color=data["marker_colors_rgba"][r] if len(data["y_values"]) > 1 else data["marker_colors_rgba"],
                    marker_line_width=2
                    ))
```

Figure 18: The figure displays the change in fig.add_trace() function for source of horizontal bar charts

2.1.3 Generating Bezier Curves, Scattered Plots and Polygons

There will be a single file generating the curves, scattered plots as well as the polygons. Curves, polygons and scattered plots all represent data in the 2D co-ordinate system[3]. Scattered plots are individual data points represented in form of separate dots without any connecting lines in between them. Curves are lines that connect data points to represent a certain pattern, whereas polygons are closed shapes that are formed by connecting data points with the help of straight lines. We have a file bezier_generator.py, a part of which is seen in Figure 28, that uses control points to produce points on a 2D Bezier curve. In order to determine the curve at a specific location, it iterates through the control points using a mathematical formula. The outcome is then kept in an array with the name "b." The x-coordinate is added as the second member if "b" is a column vector, and the resulting array is then written to a file. For a predetermined number of samples, this procedure is repeated.

Other than bezier_generator, there will be another file draw_plot.py which will import the modules from bezier_generator.py, modules from polygon_gen.py as well as from utils. polygon_gen.py has a generate_polygon function(refer Figure 29) that helps in producing polygons centered at a specific positions with certain number of vertices, a defined radius, irregularity, and spikiness. A function to plot the source images called draw_pair is defined next with certain parameters like color, grid_param, figsize, filename and some additional arguments. This function generates a pair of plots with a randomly selected background color, x and y axes markers, linewidth, face color and desired styles. The graphs are produced using the random data generator and the matplotlib library. Additionally, it

```
ticktext = []    #initializing tick text for ticks
for i in range(10):
    word = ''.join(random.choice(alphabet) for i in range(3))
    ticktext.append(word)


fig.update_layout(

     yaxis = dict(
     tickmode = 'array',
     #tickvals = [20,40,60,80,100,120,140,160,180,200],
     tickvals = [0.1,1,2,3,4,5,6,7,8,9],
     ticktext = ticktext

    ))
```

Figure 19: The change made in layout for setting tick properties for horizontal bar charts

```
fig.update_xaxes(showgrid=False)
if draw_grid:
    fig.update_xaxes(showgrid=True, gridcolor='#aaaaaa', gridwidth=1, griddash=random.choice(open("grid.txt","r").readline().split()))
```

Figure 20: Changing orientation of the produced grids to vertical such that they run parallel to x axis

generates random integers for some of the parameters, including axis widths and marker types, using the random library. The probability of the grid lines being produced depends on the grid_p variable as it will determine if grids will be produced or not. As described in Figure 30, a bezier curve defined by the provided points will be plotted if the “bezier” argument is specified in **kwargs. Scatter points will be plotted if the “scatter” argument is specified in **kwargs. Otherwise, a polygon specified by the provided points will be plotted. It will execute the function draw_grids with a randomly chosen grid style out of double dash, dash, dash dot or None from a file called “grid.txt” to plot grid lines on the plot. Furthermore, the properties for the legend, chart title, axes are defined randomly in their respective layouts. The function then uses the postprocessing() function to post-process the source and channel-wise target image and resize them to 256x256 imade dimension before saving the produced plot as a PNG file in the source directory. Figure 31 portrays the diversity included in bezier data for 3 different image aspect ratios - 1:1, 1:2 and 2:1.

The further part of the code does the work of plotting the channel-wise target. The axis is set, the right and top spines are removed, the x and y tick labels are set to empty strings, and the image is created with the provided figsize. The tick direction is also specified to be in and out, with dimensions of 20 length and 1 breadth. Next it searches the kwargs dictionary for bezier curves, scatter points, and polygons before adding them to the plot with the

```
#Plotting channelwise target images for 512x512 image dimension
def write_circle_target_data(data, filepath, figsize=(512, 512), draw_grid=False, tick_step=10):
    fig = go.Figure()
    for i in range(len(data)):
        for j in range(len(data[i])):
            fig.add_trace(go.Scatter(x=data[i][j]["y"],
                        y=data[i][j]["x"],
                        marker = {"size":np.array(data[i][j]["widths"])*np.sqrt(figsize[0]/figsize[1]),
            "sizemode":'diameter',
            "sizeref": 1,    #size reference for 512x512 images
                                },
                        ))
```

Figure 21: Modifications in fig.add_trace() function for channel-wise target of horizontal bar charts

```
#saving the axes part as an image
pio.write_image(fig=fig, file=f"{fp_parts[0]}_axes.{fp_parts[1]}", format="png", width=figsize[0], height=figsize[1])

#defining grid properties for when grids are produced
if draw_grid:
    fig.update_xaxes(showgrid=True, gridcolor='black', griddash='dash', gridwidth=1)

fig.update_layout(yaxis={"linecolor": 'white'}, xaxis={"linecolor": 'white', "ticklen": 0, "tickwidth": 0})
fig.update_layout(yaxis={"showgrid":True})
#saving the grid part as an image
pio.write_image(fig=fig, file=f"{fp_parts[0]}_grids.{fp_parts[1]}", format="png", width=figsize[0], height=figsize[1])

fig.update_traces(mode='markers', marker_line_width=2, marker_color="white", marker_line_color="black", visible=True)
#fig.update_layout(yaxis={"showgrid":False})
fig.update_layout(xaxis={"showgrid":False})
```

Figure 22: Changing grid orientation to vertical for channel-wise target

selected colours and linewidths. There is also a zorder parameter that determines the order in which the elements should be displayed. Now using ax.plot() and the transform argument in Figure 32, the x and y axes are set and they are the only elements displayed in the first saved TIFF image ending with "_axes.tiff" in the tactile folder. The code then adjusts the plot to remove the spines, the ticks and the tick labels after saving the first image, and then it removes the formerly plotted axes. The function draw_grids() is used to plot the grids if the plot contains grid lines and fig.savefig() is used to save the grid lines as another TIFF image ending with "_grids.tiff" in the tactile folder. The bezier curves, scatter points, and polygons are then plotted into the plot to provide a tactile depiction of the plot's content. This TIFF image is saved in the tactile folder with a filename ending with "_content.tiff. These images are then resized to 256x256 by calling the maskgen method present in utils, as displayed in Figure 33.

The later section of code again establishes the specifications for the gray scale target image's figure size, spines, tick labels, and axis. The code then verifies whether the keyword arguments passed to the function include "bezier," "scatter," and "polygon," and plots them if they do. The axes are then plotted, axes spines are removed and if "grid_param" is less

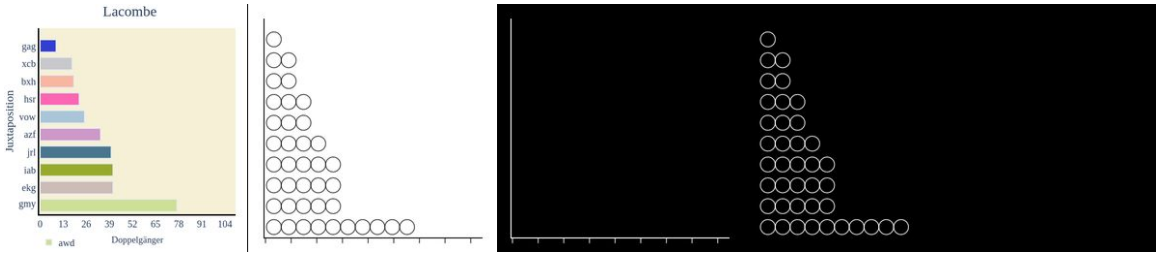


Figure 23: A sample of horizontal 512x512 dimension source(left), RGB target(centre) and channelwise target(right)

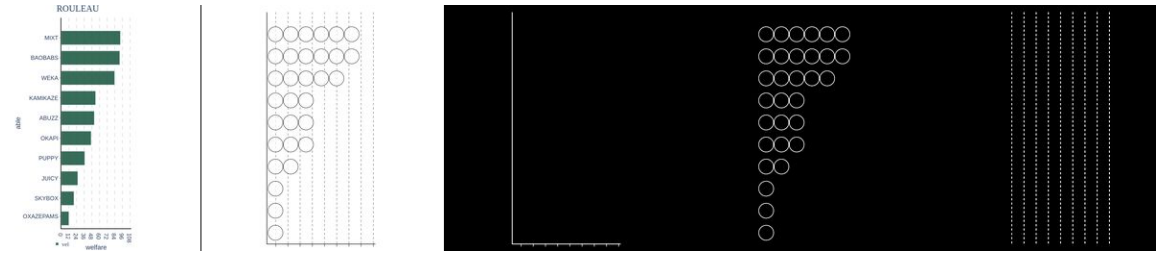


Figure 24: A sample of horizontal 512x1024 dimension source(left), RGB target(centre) and channelwise target(right)

than "grid_p," grids are added. The resulting target images will have some images having black curves, some having white plot with black scatter points and rest having polygon with black edges, black vertices but a white shape fill. All these plots will be saved in the tactile1 folder for the RGB target. Overall, 2000 bezier curves, 1500 scatter plots, and 1500 polygons are generated. The figure aspect ratios included for this file are 1:1, 1:2 and 2:1 as seen before. Following the creation of directories for storing the generated graphs, the script starts to produce the plots by looping through each element and executing functions to generate them using defined parameters. Lastly, all the plots that have been made are saved in their respective directories with filenames that reflect the sequence in which they were produced. Figure 34, Figure 35, Figure 36 portrays some of the plots created by draw_plot.py.

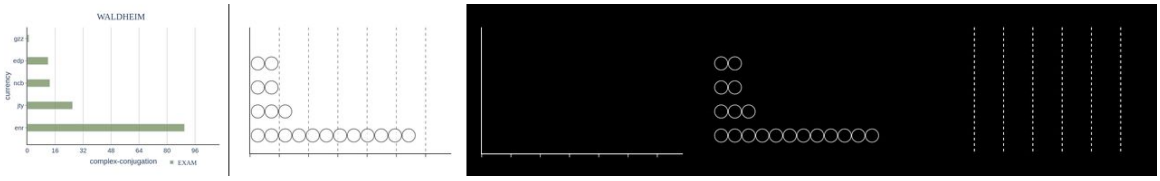


Figure 25: A sample of horizontal 760x512 dimension source(left), RGB target(centre) and channelwise target(right)

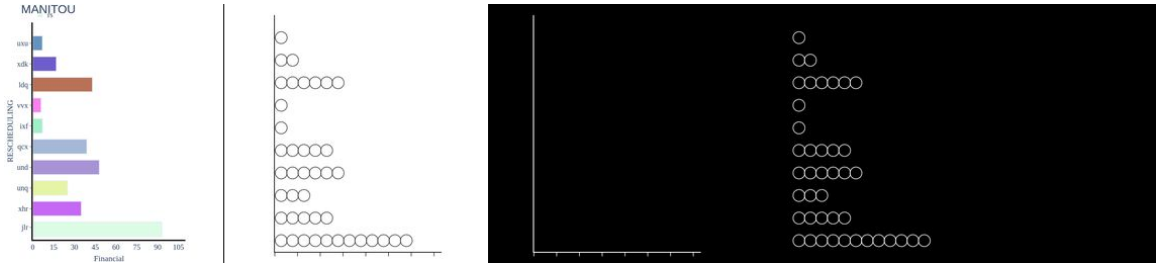


Figure 26: A sample of horizontal 512x700 dimension source(left), RGB target(centre) and channelwise target(right)

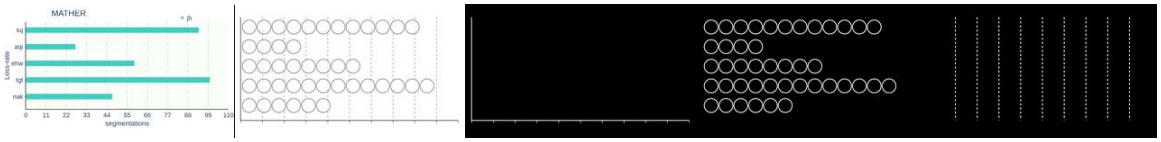


Figure 27: A sample of horizontal 1024x512 dimension source(left), RGB target(centre) and channelwise target(right)

2.2 Data Training for RGB Model

At the end of data generation, we have a diverse data set containing categories: horizontal and vertical bar charts, curves, scatter plots and polygons. With all the data produced, we proceed to the process of data training. The first model for training our data sets is the RGB Model(also addressed as the Pix2Pix Model as it uses Pix2Pix Architecture) which implements a Generative Adversarial Network for Pix2Pix Image-to-Image Translation task. As seen in Figure 37, it has a Train_Pix2Pix class that does the task of setting up prerequisites for the Pix2Pix task and training the GAN. The Train_Pix2Pix class has __init__ method() that initializes the class and loads the data set. A DataLoader object is defined that iterates over the training data set in set batch sizes, using mentioned threads worker process to load the data. Moving further, we create the generator self.netG, with input parameters like the generator architecture, number of input and output channels, number of filters and the type of normalization. The self.netD discriminator is called with discriminator architecture, concatenated number of input and output channels and use_sigmoid parameters. Lastly, the weights of generator and discriminator are initialised by calling the init weights method.

Next we define the loss function used in training the Pix2Pix model. The code specifies that binary cross-entropy loss is used, if opt.loss is set to “bce”, squared error loss is used, if it is set to “ls” and Wasserstein loss is used, if it is set to “wloss”. We define the label values for real and fake images in the loss function. Then, the optimizers for generator and discriminator are defined and append those to the list of optimizers. self.schedulers is created to store and get the the learning rate for the optimizers. For the final portion of the __init__ method(), lists are created to store the loss values during training process. The function loads a saved checkpoint containing the optimizer states and weights and sets them to the appropriate variables.

Moving to the further part of the code, the train() function starts the training of the model. Next, the perceptual loss during training is calculated. The train() function then

```
def generate_bezier(x, P):
    n = P.shape[0]   #n is equal to number of control points
    b = 0   #variable b is initialized to 0

    for i in range(n):
        a = comb(n,i)*((1-x)**(n-i))*(x**i)
        b = b + a*P[i].reshape(1,-1)

    if b.shape[1] == 1:
        b = np.concatenate((x,b), axis=1)
    print('b',b)
    return b

if __name__ == "__main__":
    P_1D = 0.25
    CROSS = []
    NUM_SAMPLES = 50000   #number of samples to generate
    for i in range(NUM_SAMPLES):
        x = np.linspace(0, 1, 10000).reshape(-1,1)
        P = np.array([[random.randint(-20,20), random.randint(-20,20)] for i in range(random.randint(2,20))])

        if random.random() <= P_1D:
            P = P.reshape(-1,1).flatten()[::2]
        b = generate_bezier(x, P)
        print(f"{i+1}.npy generated")
        np.save(f"./points/{i+1}.npy", b)   #saves b as a numpy array in a file named "{i+1}.npy"
```

Figure 28: A portion with function defined for generating curves

counts the number of iterations for training. For every iteration, current epoch is stored in the epoch variable. Moreover, the function initializes lossdlist to monitor discriminator loss, lossglist to monitor generator loss, lossl1list for L1 loss and lossperlist for keeping a track of the perceptual loss. Real_A and Real_B as seen in Figure 38, represent the input images and target images. Each batch in self.dataset has a set of input and target images, and the function loops through these batches. A batch of data is loaded and moved to the designated device during each loop and the print statement is returns the training process's progress along with the running epoch count to the console.

The discriminator loss is calculated for a single batch of data. If "wloss" is not the loss type, the loss for the fake photos is first calculated by comparing the fake image predictions and fake labels using the given criterion. Comparing the real predictions with the real labels, helps in determining the loss for the real images. The next step is to calculate the discriminator's total loss, which is calculated as the sum of the two losses multiplied by lambda GP. The next step would be to optimize the generator and update the generator's parameters. We need to make sure that the discriminator's weight's are not altered during this iteration. The generator's loss is calculated based on two things: GAN loss and pixel-wise L1 loss. The fake image is combined with the input image, and the discriminator's output is then compared to the real labels to determine the GAN loss. Whereas, the difference between the fake image and the target image is measured on a per-pixel basis by the pixel-wise L1 loss. The code then updates the learning rate and outputs the current loss values for the generator, discriminator, L1 loss, and perceptual loss after optimising

```
def generate_polygon(center, avg_radius, irregularity, spikiness, num_vertices):
    irregularity *= 2 * math.pi / num_vertices
    spikiness *= avg_radius
    angle_steps = random_angle_steps(num_vertices, irregularity)

    # now generate the points
    points = []
    angle = random.uniform(0, 2 * math.pi)
    for i in range(num_vertices):
        radius = clip(random.gauss(avg_radius, spikiness), 0, 2 * avg_radius)
        point = (center[0] + radius * math.cos(angle),
                 center[1] + radius * math.sin(angle))
        points.append(point)
        angle += angle_steps[i]

    return np.array(points)
```

Figure 29: The generate_polygon function imported from file polygon_gen.py

the generator in the GAN. Additionally, it outputs information about the training status, including the current loss values and anticipated remaining time, and saves the model at regular intervals.

Figure 39 and Figure 40, describe the further portion of the code, where five methods are defined:

- The get_scheduler method, which returns an object with a learning rate scheduler. It also configures the learning rate for the optimizer at specific training checkpoints and these checkpoints are defined as a 10 valued array, evenly spread between opt.iter_constant and opt.total_iters.
- The save_model method, which saves the generator and discriminator models' current states, together with their corresponding optimizers.
- The save_arrays method which saves the lists of generator loss, discriminator loss, and L1 loss values as numpy arrays.
- The save_hyper_params() method saves the model's hyperparameters to a JSON file.
- The get_w_loss() method employs the discriminator to create predictions for both the real and fake images after receiving the input of both types of images. It also calculates the Wasserstein loss with gradient penalty. The final loss is calculated based on the gradient penalty and the mean difference between the predictions of the discriminator for the real and fake images.

The arguments that will be used while training the Pix2Pix RGB Model are listed below:

1. dir: defines the data directory.
2. batch_size: is the required batch size for data training, has a default value of 1.
3. test_batch_size: is the required batch size for data testing, has a default value of 16.
4. input_dim: defines input depth size and has a default value of 3.

```
#adding bezier curves to the plot
if "bezier" in kwargs:
    b = kwargs["bezier"]
    plt.plot(b[:,0], b[:,1], c=color, zorder=3)

#adding scatter points to the plot
if "scatter" in kwargs and kwargs["scatter"] is not None:
    points = kwargs["scatter"]
    plt.scatter(points[:,0], points[:,1], s=50, c=color, zorder=4, marker=random.choice(open("scatter.txt","r").readline().split()))

#adding polygons to the plot
if "polygon" in kwargs:
    polygon = kwargs["polygon"]
    plt.fill(polygon[:,0], polygon[:,1], fc=f"{color}33", ec=color, zorder=3)


if grid_p < grid_param:
    draw_grids(ax, linestyle=random.choice(open("grid.txt","r").readline().split()))
```

Figure 30: Statements mentioning if a curve, scatter plot or polygon will be generated based on argument specified in kwargs

5. output_dim: defines output depth size and has a default value of 3.
6. gen_filters: defines starting number of filters for the generator, with a default value of 64.
7. disc_filters: defines starting number of filters for the discriminator, with a default value of 64.
8. epoch_count: defines the starting epoch, has a default value of 1.
9. total_iters: specifies the total number of epochs to train. The total iteration value we have used is 135.
10. iter_constant: is the number of epochs to keep learning rate constant for. The value we have set is 25.
11. lr: learning rate for the optimizer, with a default value of 0.0002.
12. label_smoothing: specifies whether to use one-sided label smoothing, with a default value of False.
13. beta1: defines the beta1 parameter for the Adam optimizer, with a default value of 0.01.
14. cuda: specifies whether or not to use GPU accelerated training, it has a default value of True.
15. threads: specifies the number of CPU threads that will be used for loading the data set, we have the default value of this argument as 8.
16. lambda_A: defines the L1 regularization lambda value, has a default value of 10.
17. lambda_per: defines the perceptual lambda value, has a default value of 0.

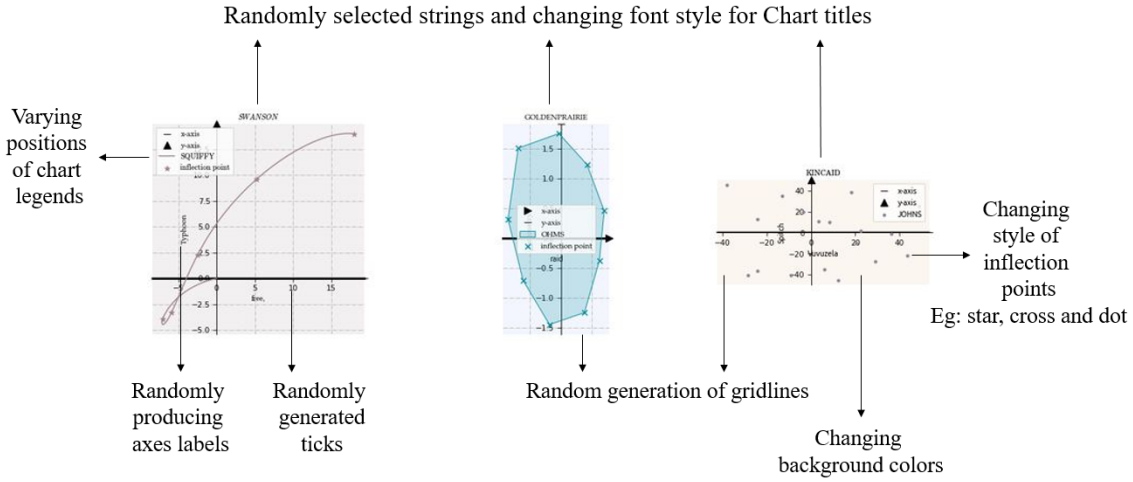


Figure 31: The diversity observed in source images of curves, polygons and scatter plots

18. lambda_GP: defines gradient penalty loss lambda value, has a default value of 0.1.

19. norm: specifies the type of normalization mode to use, with a default value of "instance".

20. gen: denotes the generator architecture that will be used. The provided choices are Resnet, UNet++, UNet, UNet-no-skips, etc.

21. disc: denotes the discriminator architecture that will be used. The provided choices are Global and Patch.

22. loss: specifies the loss function to use, the provided choices are ls, bce, wloss.

23. no_aug: specifies whether to disable data augmentation, with a default value of False.

24. folder_save: is the folder where we should save the model, with a default value set as "pix2pix"

25. folder_load: is the folder where we should load the model from, with a default value set as "pix2pix".

26. checkpoint_interval: it is the interval between saving model checkpoints, with a default value of -1 which indicates that no checkpoints will be saved while training.

27. continue_training: determines whether to continue training from a previously saved checkpoint, with a default value set as False.

Finally, an instance of the Train_Pix2Pix class is created using the parsed command line arguments and the training data set. After that, it generates directories for storing the checkpoints, trained models, and hyper-parameters before training the Train_Pix2Pix object and for storing the finished trained model, loss, hyper-parameters, etc.

```
#setting further axis properties
ax.plot((1), (0), ls="", ms=10, color="k",
transform=ax.get_yaxis_transform(), clip_on=False, zorder=2)
wr1 = ref(ax.lines[-1])
ax.plot((0), (1), ls="", ms=10, color="k",
transform=ax.get_xaxis_transform(), clip_on=False, zorder=2)
wr2 = ref(ax.lines[-1])
fig.savefig(f'./tactile/t_{filename}_axes.tiff', dpi=75)   #saving the axes part as an image

ax.tick_params(direction='inout', length=0, width=0, zorder=3)
ax.lines.remove(wr1())
ax.lines.remove(wr2())
for _, item in ax.spines.items():
    item.set_visible(False)

#setting grid properties if grids are produced
if grid_p < grid_param:
    draw_grids(ax, color='k', linestyle='--', linewidth=1)


#saving the grids part as an image
fig.savefig(f'./tactile/t_{filename}_grids.tiff', dpi=75)
plt.grid(False)
```

Figure 32: Code snippet displaying saving the axes and grid files as images for channel-wise target

2.3 Data Testing for RGB Model

Testing begins with importing all the necessary python packages like json, numpy, matplotlib and many more. We later define three functions(refer Figure 41) by name load_opt, load_model, load_data with class Opt. load_opt takes a JSON file path, opens the file and load it's contents as a dictionary, it then provides the dictionary as a argument to the constructor and creates an Opt instance. load_model creates a generator model with the help of create_gen function, sets the device model and loads the saved state of the model. load_model requires three arguments: model_path, opt and device. Whereas, the third function, load_data, require two arguments photo_path and opt. It uses the photo_path and get_dataset function to create a data set and then it also produces a DataLoader object that wraps the data set.

Further, we perform linear transformation on the input tensor 'a' and define concatenate function that will be called while producing the final output. Figure 42 explains that later a save_images function is initialized that takes a PyTorch data set and filepath as input and it extracts the source image(real_A) and target image(real_B) from the batch, it then applies a generator to input image to produce out i.e. the trained output image and then it transfers the output image to CPU and unnormalizes the source, target as well as the output. We will save the final images in format "(i+1).png" and they will be concatenated

```
#saving the content of bezier data as an image
fig.savefig(f'./tactile/t_{filename}_content.tiff', dpi=75)
maskgen(f'./tactile/t_{filename}.tiff')
plt.close('all')
```

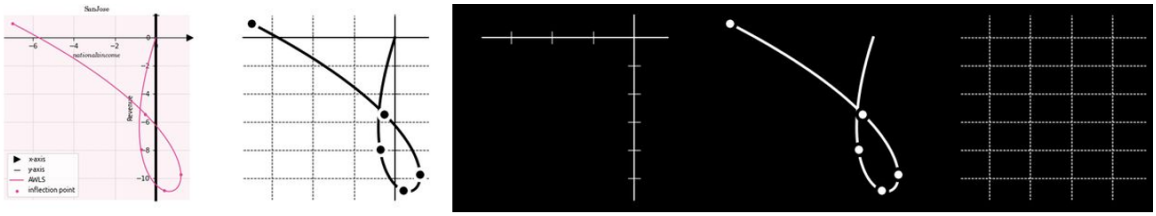
Figure 33: Code snippet displaying saving the content file as image for channel-wise target

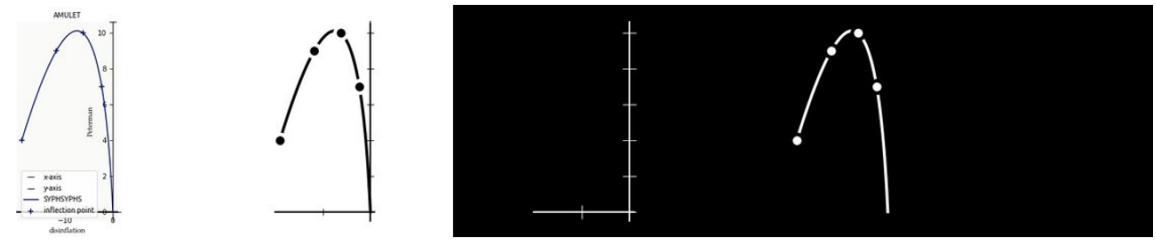
horizontally with each other in format source image + target image + produced output. An example of the final output produced by the RGB Model is seen in Figure 43.

The later part of the code then loads parameters from a pre-trained model and configures the computing equipment to use a CUDA GPU. The test photos are then loaded into a PyTorch data set after the code defines their location and the final outputs are saved to the output directory "pix2pix" that has been selected. With this, the testing of RGB model comes to an end.

2.4 Data Training for Channelwise Model

The class Train_Pix2Pix defined while data training for channel-wise model is in charge of developing a GAN model to implement the Pix2Pix Image to Image translation[9]. Within this class, the __init__ function is defined with Opt and traindataset arguments. The object "opt" holds a number of specific options, including batch size, learning rate, loss function, etc. and the training dataset, is then loaded with the help of DataLoader. The functions create_gen and create_disc are used to create the generator and discriminator models, respectively. The init_weights function is used to initialise the weights of the models and then we define the GAN loss function. The learning rate is set in opt when the Adam optimizer is used to define the optimizers. Also the schedulers for learning rate are connected to the optimizers. The lists gen loss, disc loss, l1 loss, per loss, and gp loss respectively provide the generator loss, discriminator loss, L1 loss, perceptual loss and gradient penalty loss. Next, the model and optimizer states are loaded from a stored checkpoint file if the continue training option is set to True. The further part of code defines the train function containing parameters self and opt. The opt object is likely to contain variety of hyperparameters and options for the training process. Each iteration of the loop, which takes place per opt.total_iters number of epochs of training, lasts for the specified number of iterations. The loop iterates over all data batches, loading each batch and sending it through the generator network(self.netG) to produce fake images for each epoch. The self.netD i.e. the discriminator network is then optimised on both the false and actual image predictions using self.gan loss. After calculating the discriminator network's loss and updating its weights, as per Figure 44, the method determines whether or not to regularise the discriminator with a gradient penalty. The gradient penalty loss (gp_loss) is calculated using gradient_penalty function and added to the loss_D(discriminator's loss) if method decides that gradient penalty needs to be used. The discriminator network is then used to backpropagate the gradient penalty loss. The generator network is then optimised by computing the generator's loss (loss_G) and changing the discriminator's gradients to False. The loss_G_GAN i.e. the loss of the generator is a combination of the GAN loss, the L1 loss,





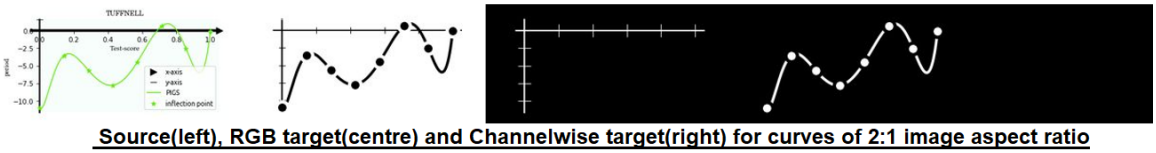


Figure 34: The curves generated by our code

and the perceptual loss. Using the pred_fake from the discriminator and the real labels, the GAN loss for the generator is calculated. The L1 loss access the pixel-level difference between the actual and false images[2], whereas the perceptual loss access the variation in the discriminator's characteristics between the real and fake images[4]. The generator's weights are updated and the loss is back propagated in the end. For tracking the training process, the lossgpdlist for the discriminator and lossglist, lossl1list, lossperlist for the generator are attached to their respective lists.

Further, two more functions are defined. The get_scheduler() that takes an optimizer object as input and returns a scheduler object and a gamma value of 0.8 after dividing the total number of repeats into ten evenly spaced intervals. Next is gradient_penalty that takes the real image, the real and fake masks, along with a number of additional parameters as inputs. The method determines a "gradient penalty" value that is utilized to boost the discriminator's performance. The generator and discriminator models, along with associated optimizers, are saved in a specific location via the save model() method. It creates the directory, if it doesn't already exist. The save_arrays() method is used to save generator loss, discriminator loss, L1 loss, perceptual loss, and gradient penalty loss to distinct numpy arrays in a designated directory. The GAN model's hyper parameters are saved as a JSON file in a designated directory using the save_hyper_params() method.

The arguments used for training the channel-wise model's data are as listed in Table 1.

Finally, we set up the folders for saving the arrays, checkpoints and the trained model. The trained model is saved to disk, while the hyperparameters are saved to the appropriate directories and this marks the end of training of channel-wise model.

argument	function	default value
data	The dataset is stored in this directory	/data
batch_size	defines number of samples in each batch for training	4
input_dim	input dimension's depth size	3
output_dim	output dimension's depth size	3
epoch_count	the starting epoch value	1
total_iters	the number of epochs we will train the data for	135
lr	the learning rate	0.002
label_smoothing	if this is set, we will not use one sided label smoothing	True
beta1	parameter for Adam optimizer	0.01
cuda	if this is set, we will not use GPU training	True
threads	required for loading the dataset	6
lambda_a	coefficient of L1 loss	5
lambda_gp	coefficient of gradient penalty	0.1
lambda_per	coefficient of perceptual loss	0.2
w_per	perceptual weights	0,.1,.3,.6
gen	generator architecture	UNet++
disc	discriminator architecture	Patch
loss	loss function for ganloss	ls
no_aug	if set, we won't argument the data set	False
folder_save	location to save the model	pix2obj
folder_load	location from where we want to load the model	pix2obj
checkpoint_interval	interval between model checkpoints	-1
continue_training	to load the weights for the network before training	False
d_reg_every	how frequently we should regularize the discriminator	4

Table 1: The parameters used while training the channel-wise model

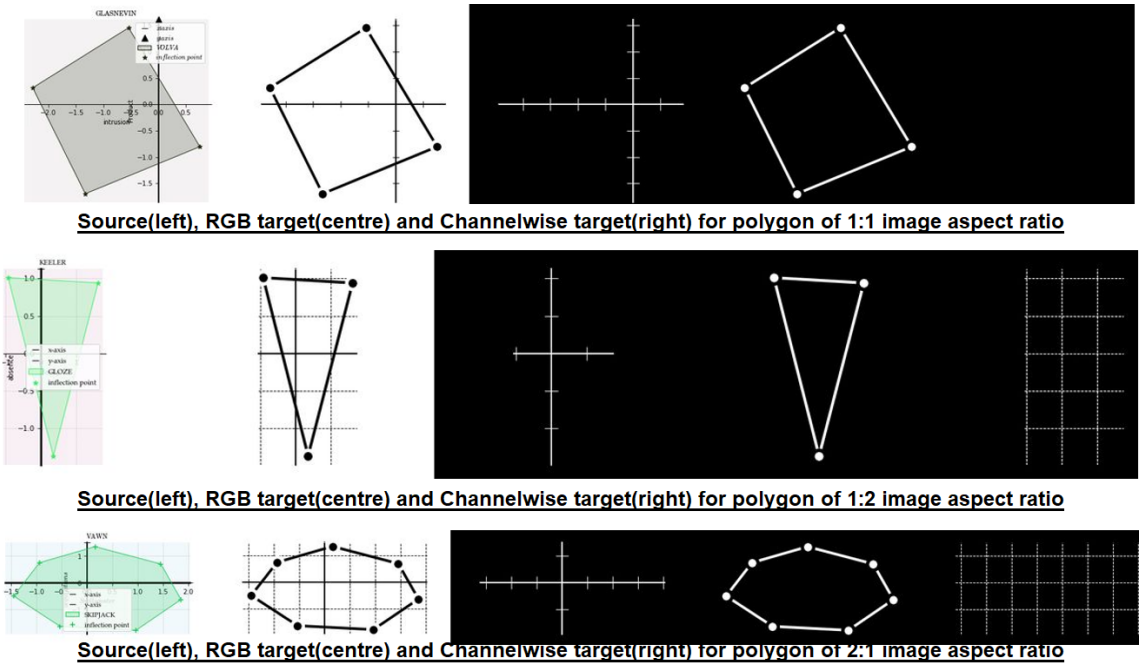


Figure 35: The polygons generated by our code

2.5 Data Testing for Channelwise Model

We begin the testing by declaring an Opt class that has parameters self and dictionary. We then define load_opt function that will load a JSON file and return an Opt instance. Next, we load the generator model and it is returned. A generator is created with the help of create_gen function and torch.load loads the state dictionary of the model from a checkpoint file and sets the state dictionary of the generator to that value. The image data is loaded from a path with the help of get_dataset function. It then returns a DataLoader object, which may be used to go through batches of the data during training. We further load the numpy arrays from several dictionaries and return them in one dictionary. Even the training losses like generator loss, L1 loss, gradient penalty loss and perceptual loss are loaded now. Moving ahead, we have defined a visualise() function in Figure 45 that returns a visual representation of the output after receiving a list as input. This list consist of three binary masks representing the positions of axes, content and grids in the output image. The ToPILImage() method is first applied to transform each mask into a PIL image and the np.expand_dims() is applied next to expand each image by giving it a new dimension. A blank array composed of all zeros is then created with np.zeros(). The content and grid images are then joined onto the axis image using the paste function after converting the final image back to a PIL image before returning the final image.

In the meanwhile, the losses are plotted and saved in a dictionary with the save_plot() function. There is also a eval_model function that has 3 parameters: model, dataset and path. eval_model() calculates three metrics—pixel accuracy, Dice coefficient, and Jaccard index and assess the model using the dataset. It then prints the average values for the three metrics and saves the results to a text file in the chosen directory. Next comes the save_images() function(refer Figure 46), which iterates through the dataset using a for loop,

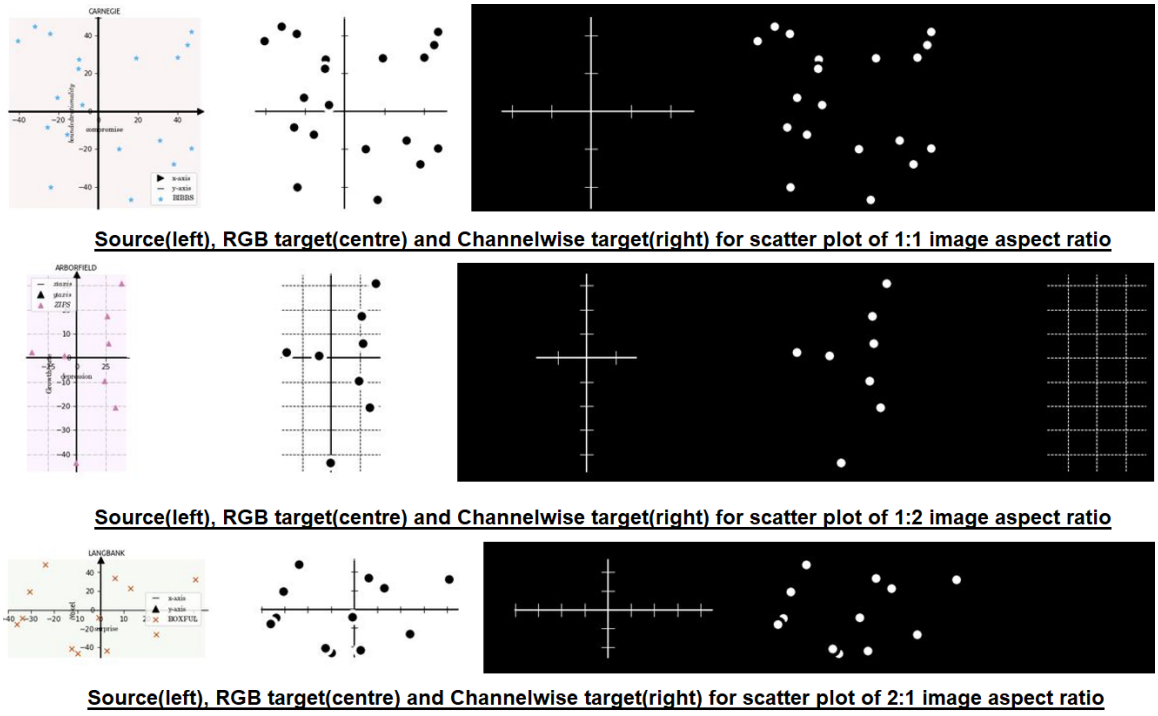


Figure 36: The scatter plots generated by our code

producing output photos based on the trained model for each batch. After extracting the source image, target image, and output from the batch, it uses the visualise() function to produce PIL images. Using out_img.save, the output image is saved in format "o_(i).png". We have also created an empty image of 256x256 dimension, to attach in front of and behind the source to make it the same dimension as the channel-wise target and output, we name the new source as a_elements. Now, with the help of the concat_images(), we vertically concatenate the new source image, the channel-wise target and the output image. The concatenated images will be saved in format "elm_(i).png". An example of final produced output by channel-wise model can be seen in Figure 47.

At last, we have also opened and iterated through the o_(i).png kind of images in "pix2obj" output folder to convert the contents of it to the color: black and white and we have saved them again in the same folder. This portion takes care of any inconsistency in terms of color seen in the final outputs and that ends the testing procedure too.

3 Procedure and Setup

Once the VM Environment is set, one can download the contents of the PROJECT folder from the shared drive. The PROJECT folder has 7 sub folders, 3 out of which deal with synthetic data generation and the rest are for RGB and Channel-wise training as well as testing. The first in list is Bar Channelwise-RGB Merged(Horizontal), as the name suggest it has files that will generate the horizontal bar charts. For producing the bar-charts, one can open a terminal and run the command: python draw_bar_horizontal.py, once this command is successfully executed and finished, run another command: python

```
class Train_Pix2Pix:
    '''
    GAN model for Pix2Pix implementation. Trains models, saves models, saves training results and details about the models.
    '''
    def __init__(self,opt,traindataset):

        #load in the datasets
        self.dataset = DataLoader(dataset=traindataset, batch_size=opt.batch_size, shuffle=True,num_workers=opt.threads)


        #tensorflow logger
        self.device = torch.device("cuda:0" if opt.cuda else "cpu")

        #create generator and discriminator
        self.netG = create_gen(opt.gen,opt.input_dim,opt.output_dim,opt.gen_filters,opt.norm)
        self.netG.to(self.device)
        init_weights(self.netG)
        use_sigmoid = True if opt.loss == "bce" else False
        self.netD = create_disc(opt.disc,opt.input_dim+opt.output_dim,use_sigmoid)
        self.netD.to(self.device)
        init_weights(self.netD)
```

Figure 37: Defining the Train_Pix2Pix class for initializing the methods and loading data set

draw_bar_horizontal2.py. As and when both these commands will finish executing, you can open the data folder, which will now have a source directory containing the source images, there will be a tactile directory containing the channel-wise targets for the source images and there will be a tactile1 directory having the gray-scale targets for RGB model. The next folder is Bar Channelwise-RGB Merged(Vertical), it has the draw_bar_vertical.py and draw_bar_vertical2.py, both of which can be run in the terminal to produce source images, channel-wise target and RGB target images for the vertical bar charts in directories source, tactile and tactile1, respectively. This way we will have the barchart data almost ready for training. Finally, we can go to the Bezier Channelwise-RGB Merged folder, open the terminal and run the file draw_plot.py. This will produce and save the curves, polygons and then the scattered plots in directories source, tactile and tactile1 in the data folder. Now, we will have all 5 kinds of data ready for further processing.

Simultaneously, if we check the tactile1 folders for all these data categories, we can observe that the images are original size, unlike the resized 256x256 dimension images in source and tactile folders. The training and testing models ask for the input source and targets of form 256x256 image dimension, due to this factor we will now batch resize the images in all the tactile1 folders. We can open the Bar Channelwise-RGB Merged(Horizontal) folder, start the terminal and run command: python Resize.py. This will resize all the gray-scale targets in the tactile1 folder to 256x256 dimension and save them with a TIFF extension. This will result in duplicate target images in tactile1 folder. To resolve this issue, we can run the file RemovePng.py which will remove the duplicate original sized PNG files from the tactile1 folder. We can repeat the above mentioned steps for vertical barcharts, curves, polygons and scatter plots. At the end of this, we will have uniform, authentic data as required for data training and testing.

Subsequently, we now go to the folder Pix2Pix-main(RGB Model-BarChart) for training the data for RGB Model. The steps to go about would look like:

```
def train(self,opt):
    '''
    Starts the training process. The details and parts of the model were already initialized in __init__
    '''

    # load vgg if we want to use perceptual loss
    if opt.lambda_per != 0:
        perceptual = VGGPerceptualLoss(resize=True)

    for i in range(opt.total_iters):
        epoch = i + opt.epoch_count
        #monitor each minibatch loss
        lossdlist = []
        lossglist = []
        lossl1list = []
        lossperlist = []

        t1 = time.time()

        for j, batch in enumerate(self.dataset):
            if j % 100 == 0:
                print("training epoch ",epoch,"batch", j,"/",len(self.dataset))
            real_A, real_B = batch[0].to(self.device), batch[1].to(self.device) #load in a batch of data
```

Figure 38: The train() function that starts the training of the model

1. Create a subfolder “data”.
2. Create folders train and test within data.
3. Copy the source and tactile1 directories from data folder of Bar Channelwise-RGB Merged(Horizontal) and paste it in the train and test folders of Pix2Pix-main(RGB Model-BarChart).
4. Rename the tactile1 as tactile.
5. Open the terminal.
6. Execute command python train.py, this will print the status of the training for each iteration.
7. Create a folder Outputs and a subfolder “pix2pix”.
8. Once training is completed without any errors, start the testing with the help of command python test.py. On running this, the final outputs start printing and the location where these are saved are returned.
9. Go to pix2pix, you can see all the trained and produced outputs for the horizontal barcharts.

For training the model for vertical barchart data, you can repeat the above steps while replacing the horizontal barcharts with vertical barchart’s source and grayscale target from

```
@staticmethod
def get_scheduler(optimizer):
    milestone = np.int16(np.linspace(opt.iter_constant, opt.total_iters, 11)[:-1])
    scheduler = lr_scheduler.MultiStepLR(optimizer, milestones=list(milestone), gamma=0.8)
    return scheduler

def save_model(self,modelpath):
    '''
    Saves the models as well as the optimizers
    '''
    torch.save({
        'gen': self.netG.state_dict(),
        'disc': self.netD.state_dict(),
        'optimizerG_state_dict': self.optimizer_G.state_dict(),
        'optimizerD_state_dict': self.optimizer_D.state_dict(),

        }, modelpath)

def save_arrays(self,path):
    '''
    save the losses to numpy arrays
    '''
    np.save( os.path.join(path,"genloss"),np.asarray(self.gen_loss))
    np.save( os.path.join(path,"discloss"),np.asarray(self.disc_loss))
    np.save( os.path.join(path,"l1loss"),np.asarray(self.l1_loss))
```

Figure 39: The get_scheduler, save_model and save_arrays methods for data training

Bar Channelwise-RGB Merged(Vertical). Moving to the curves, polygons and scatter plots, we copy the source and tactile1 directories from the data folder of Bezier Channelwise-RGB Merged. We go to Pix2Pix-main(RGB Model)(Beizer) folder, create train and test subfolders within data folder, and paste the source and tactile1 directories there. We rename the tactile1 as tactile and put the data for training first and then testing using the similar commands as seen above. This will complete the training and testing of all 5 categories of data for the RGB model.

The inputs for Channel-wise model will vary in some proportions. Starting with the vertical barcharts, we will open the Pix2Pix-obj folder and go along with the following steps:

1. Create a subfolder “data”.
2. Create folders train and test within data.
3. Copy the tactile and tactile1 directories from data folder of Bar Channelwise-RGB Merged(Vertical) and paste it in the train and test folders of Pix2Pix-obj.
4. Rename the tactile1 as source.

```
def save_hyper_params(self,folderpath,opt):
    '''
    We need to load back in the details of the model when we test so we save these as well
    '''
    with open(os.path.join(folderpath,'params.txt'), 'w') as file:
         file.write(json.dumps(opt.__dict__))

def get_w_loss(self,real_AB,fake_AB, pred_fake, pred_real, gp_lambda):
    '''
    Wasserstein Loss with Gradient Penalty
    '''
    epsilon = torch.rand(len(real_AB), 1, 1, 1, requires_grad=True).to(self.device)
    interpolated = real_AB*epsilon + fake_AB * (1 - epsilon)
    mixed_pred = self.netD(interpolated)
    gradient = torch.autograd.grad(inputs=interpolated,outputs=mixed_pred,
                grad_outputs=torch.ones_like(mixed_pred), create_graph=True,retain_graph=True,)[0]

    gradient = gradient.view(len(gradient), -1)
    gradient_norm = gradient.norm(2, dim=1)
    gp = (gradient_norm -1)**2
    gp = gp.mean()

    loss = pred_fake.mean() - pred_real.mean() + gp* gp_lambda
    return loss
```

Figure 40: The save_hyper_params and get_w_loss() method for data training

5. Open the terminal.
6. Execute command python train.py.
7. Create a folder “Outputs” and a subfolder “pix2obj”.
8. Once training is completed without any errors, start the testing with the help of command python test.py. On running this, the final outputs start printing and the location where these are saved are returned.
9. Go to pix2obj, you can see all the trained and produced outputs for the vertical barcharts.

For training the horizontal barchart data for Channel-wise model, you can repeat the above steps while replacing vertical barcharts with horizontal barchart’s tactile and tactile1 folders from Bar Channelwise-RGB Merged(Vertical). Finally for training the curves, polygons and scatter plots for Channel-wise model, we copy the tactile and tactile1 directories from the data folder of Bezier Channelwise-RGB Merged. We go to the last folder: Pix2Pixobj (copy 1), create train and test subfolders within data folder, and paste the tactile and tactile1 directories there. We rename the tactile1 as source and put the data for training first and then testing using the similar commands as seen above. The results will be saved in the pix2obj directory within Outputs. This brings us to the end of training and testing of data for Channel-wise model.

```
#Opt takes a dictionary as input and converts its key-value pairs to object attributes
class Opt:
     def __init__(self, dictionary):
        for k, v in dictionary.items():
             setattr(self, k, v)

#load_opt  loads a JSON file from the specified path and returns an Opt instance
def load_opt(path):
    with open(path) as json_file:
        opt = json.load(json_file)

    opt = Opt(opt)
    return opt

#load_model loads a generator model from the specified path and returns it
def load_model(model_path,opt,device):
    G = create_gen(opt.gen,opt.input_dim,opt.output_dim,opt.gen_filters,opt.norm,multigpu=False)
    G.to(device)

    checkpoint = torch.load(model_path)
    G.load_state_dict(checkpoint["gen"], strict=False)
    return G

def load_data(photo_path,opt):
    data = get_dataset(photo_path, opt, mode='test')
    dataset = DataLoader(dataset=data, batch_size=1, shuffle=False,num_workers=4)
    return dataset
```

Figure 41: The functions load_opt, load_model, load_data within class Opt

4 Test cases and Results

The previous batch of results had 40 percent grid probability and was trained on 125 epoch counts initially. The produced output of the samples with grids displayed many inconsistencies related to bar content, undetected grids for most outputs and inferior quality. It was later determined that due to lower frequency of grid lines, the condition of the outputs is suffering. Owing to this reason, a fresh batch of data was regenerated with increased grid probability of 80 percent. The data sets were then individually trained on both the models for 135 iterations, 200 iterations and 300 iterations to compare which experiment yielded the best outcomes. For our overall evaluation, we have trained our models on 135 iterations and the comprehensive final outputs on training all five data types of different aspect ratios on both the RGB as well as channel-wise model can be seen in Figure 48 and onward.

5 Conclusion

In this project, different data categories have been generated, trained and assessed in order to produce efficient tactile images for the visually impaired individuals. Out of all the test

```
def save_images(dataset,path):
    for i, batch in enumerate(dataset):
        real_A, real_B = batch[0], batch[1]
        with torch.no_grad():
            out = Gen(real_A.to(device))[0].cpu()

        a = unnormalize(real_A[0])   #unnormalized source image
        b = unnormalize(real_B[0])   #unnormalized target image
        out = unnormalize(out)      #unnormalized output image

        file_name = str(i+1) +".png"   #saving the concatenated output as an image
        save_image(concat_images(a,b,out), os.path.join(path,file_name))   #horizontally concatenated source, target and trained output images
        print(f"file saved at: {os.path.join(path,file_name)}")

opt_path = os.path.join(os.getcwd(),"models","pix2pix","params.txt")   #output folder
opt = load_opt(opt_path)
device = torch.device("cuda:0")

model_path = os.path.join(os.getcwd(),"models",opt.folder_load,"final_model.pth")
Gen = load_model(model_path,opt,device)

photo_path_test = os.path.join(os.getcwd(),"data","test", "source")
dataset = load_data(photo_path_test,opt)

output_path = os.path.join(os.getcwd(),"Outputs",opt.folder_load)
mkdir(output_path)
save_images(dataset,output_path)
```

Figure 42: Code snippet displaying the unnormalization and concatenation of inputs and produced output

cases, the results produced by training and testing our data sets on RGB and channel-wise model for 135 epoch counts turned out to be the most clear, appropriate and intelligible. On training the data for epoch counts more than 135 resulted in over-fitting which in turn negatively impacted the quality of output. In terms of the RGB model, the categories: curves, scatter plots and polygons were more precise while generating the final output which is likely as the bar chart data is a little more complex and detailed in comparison. After evaluating the results of both the models, it can be determined that the channel-wise model performed better than the RGB model as the target was almost completely replicated by the training output for every instance. For the channel-wise model, we have computed the pixel accuracy, Dice coefficient and Jaccard index. Pixel accuracy is one of the most simple indicator that can be used to assess the total number of pixels accurately classified in the ground truth or the resulting segmentation mask[1]. Dice coefficient and Jaccard index are generally used to compare the similarities in between 2 data sets. Here, they are used to assess the degree of overlap between the target segmentation mask and the output produced as a result of training. The average values of pixel accuracy, Dice coefficient and Jaccard index computed by our model for the data categories are 92.87 percent, 0.927 and 0.908, respectively.

References

[1] Rajat Garg, Anil Kumar, Nikunj Bansal, Manish Prateek, and Shashi Kumar. Semantic segmentation of polsar image data using advanced deep learning model. *Scientific Reports*, 11:1–18, 2021.

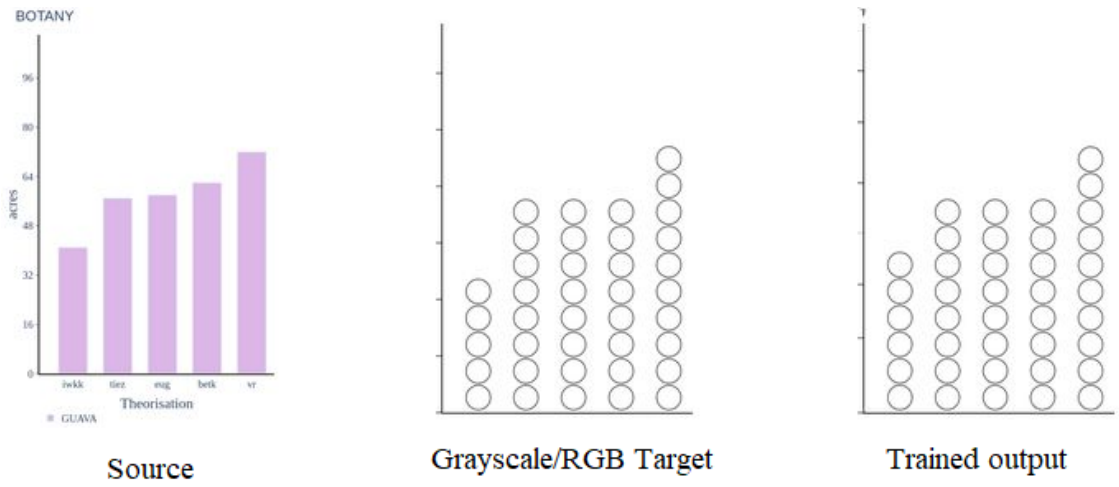


Figure 43: A representation of horizontally concatenated final output produced on training vertical barchart on RGB Model

[2] Bydder M Zhou Z Yin W Nguyen KL Yang Y Hu P. Ghodrati V, Shao J. Mr image reconstruction using deep learning: evaluation of network structure and loss functions. pages 1516–1527, 2019.

[3] Mohammad Mahdi Heydari Dastjerdi. Towards end-to-end semantically aware 2d plot tactile generation. *CURVE*, 2022.

[4] Seyun Lee, Ji-Hwan Kim, and Jae-Pil Heo. Super-resolution of license plate images via character-based perceptual loss. pages 560–563, 2020.

[5] Ashani Shalika Ranasinghe. Braille, auditory and tactile tool for children with visual impairments. *IEEE*, pages 1–3, 2019.

[6] Andreas Reichinger, Stefan Maierhofer, and Werner Purgathofer. High-quality tactile paintings. *J. Comput. Cult. Herit.*, 4, 2011.

[7] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *CoRR*, abs/1708.08296, 2017.

[8] Jiaji Wang, Shuihua Wang, and Yudong Zhang. Artificial intelligence for visually impaired. *Displays*, 77:102391, 2023.

[9] Qiu Yu, Jamal Malaeb, and Wenjun Ma. Architectural facade recognition and generation through generative adversarial networks. pages 310–316, 2020.

```
#now that we have the full loss we back propogate
loss_D.backward()
self.optimizer_D.step()

d_regularize = (opt.d_reg_every!=0) and (epoch % opt.d_reg_every == 0) and (opt.lambda_gp!=0)
if d_regularize:
    self.optimizer_D.zero_grad()
    gp_loss = self.gradient_penalty(real_A, real_B, fake_B, lambda_gp=opt.lambda_gp)
    gp_loss.backward(retain_graph=True)
    self.optimizer_D.step()
    lossgpdlist.append(gp_loss.item())
else:
    lossgpdlist.append(0)


# Optimize G ##################################
set_requires_grad(nets=self.netD, requires_grad=False) #dont want the discriminator to update weights this round
self.optimizer_G.zero_grad()
pred_fake = self.netD(real_A, fake_B) #generate D predictions of fake images

loss_G_GAN = self.gan_loss(pred_fake, True, for_discriminator=False).mean()
lossglist.append(loss_G_GAN.item())
```

Figure 44: Backpropagating loss and optimizing the generator

```
def visualize(out):
    ax_msk = invert(ToPILImage()(out[0]))   #binary mask indicating axis location in the output image
    grid_msk = ToPILImage()(out[1])   #binary mask indicating grid location in the output image
    content_msk = ToPILImage()(out[2])   #binary mask indicating content location in the output image

    #ax, content, grid are created from PIL images with extra dimension added
    ax = np.expand_dims(np.array(ax_msk), axis=2)
    content = np.expand_dims(np.array(content_msk), axis=2)
    grid = np.expand_dims(np.array(grid_msk), axis=2)

    #creating a blank array containing all zeros
    blk = np.zeros((256,256,3), dtype=np.uint8)

    ax = np.concatenate((ax,ax,ax), axis=2)

    content = np.concatenate((content, blk), axis=2)
    grid = np.concatenate((blk, grid), axis=2)

    #ax, content, grid are converted back to PIL images
    ax = Image.fromarray(ax)
    content = Image.fromarray(content)
    grid = Image.fromarray(grid)

    ax.paste(grid, (0,0), grid_msk)
    ax.paste(content, (0,0), content_msk)

    return ax   #returning the resulting image
```

Figure 45: The visualize() method of test_channelwise.py file

```
def save_images(model, dataset, path):
    for i, batch in enumerate(tqdm(dataset)):
        real_A, real_B = batch[0], batch[1]
        with torch.no_grad():
            out = model(real_A.to(device)).cpu()

        #a = unnormalize(real_A[0])
        a = real_A[0]   #source image
        b = real_B[0]   #target image
        out = out[0]   #output of trained data

        a_img = visualize(a)
        b_img = visualize(b)
        out_img = visualize(out)

        out_img.save(os.path.join(path,f"o_{i+1}.png"))
        empty_image = Image.new('RGB', (256, 256), color='white')   #an empty white image
        a_elements = concat_images(empty_image, ToPILImage()(a[0]), empty_image)
        b_elements = concat_images(ToPILImage()(b[0]), ToPILImage()(b[1]), ToPILImage()(b[2]))
        out_elements = concat_images(ToPILImage()(out[0]), ToPILImage()(out[1]), ToPILImage()(out[2]))
        concat_images(a_elements, b_elements,out_elements, mode="v").save(os.path.join(path,f"elm_{i+1}.png"))
```

Figure 46: save_images method calling visualize() method and concatenating the inputs along with trained output

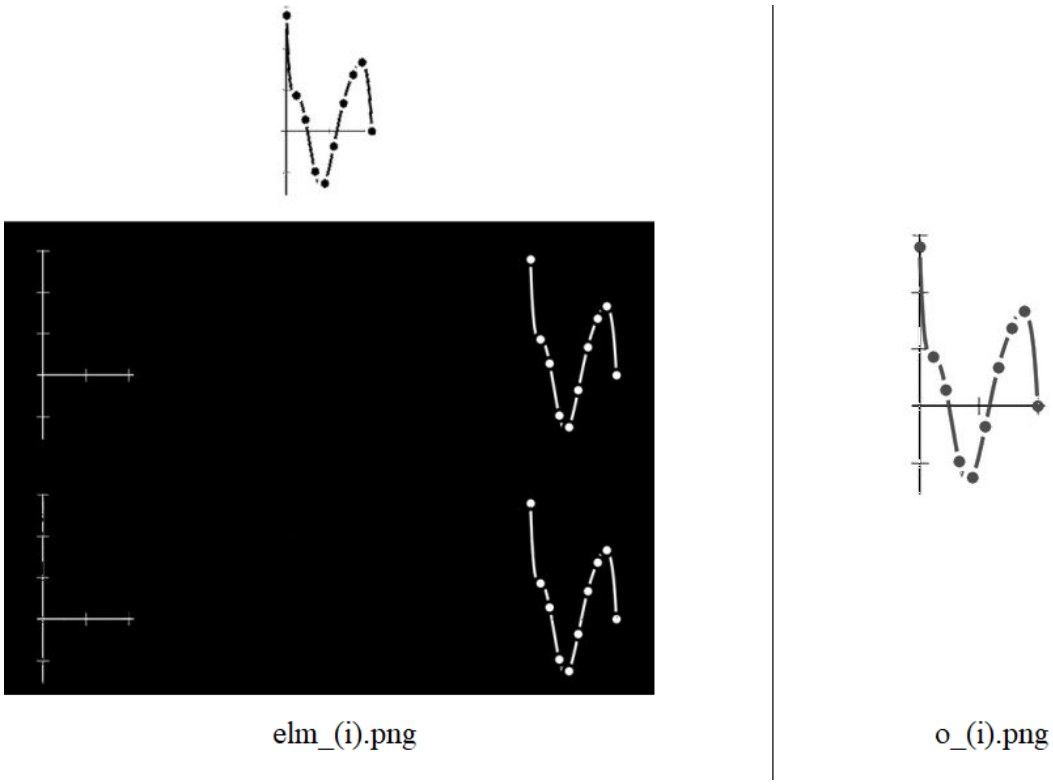


Figure 47: A representation of vertically concatenated final output elm_(i).png(left) and o_(i).png(right) produced on training curves on Channel-wise Model

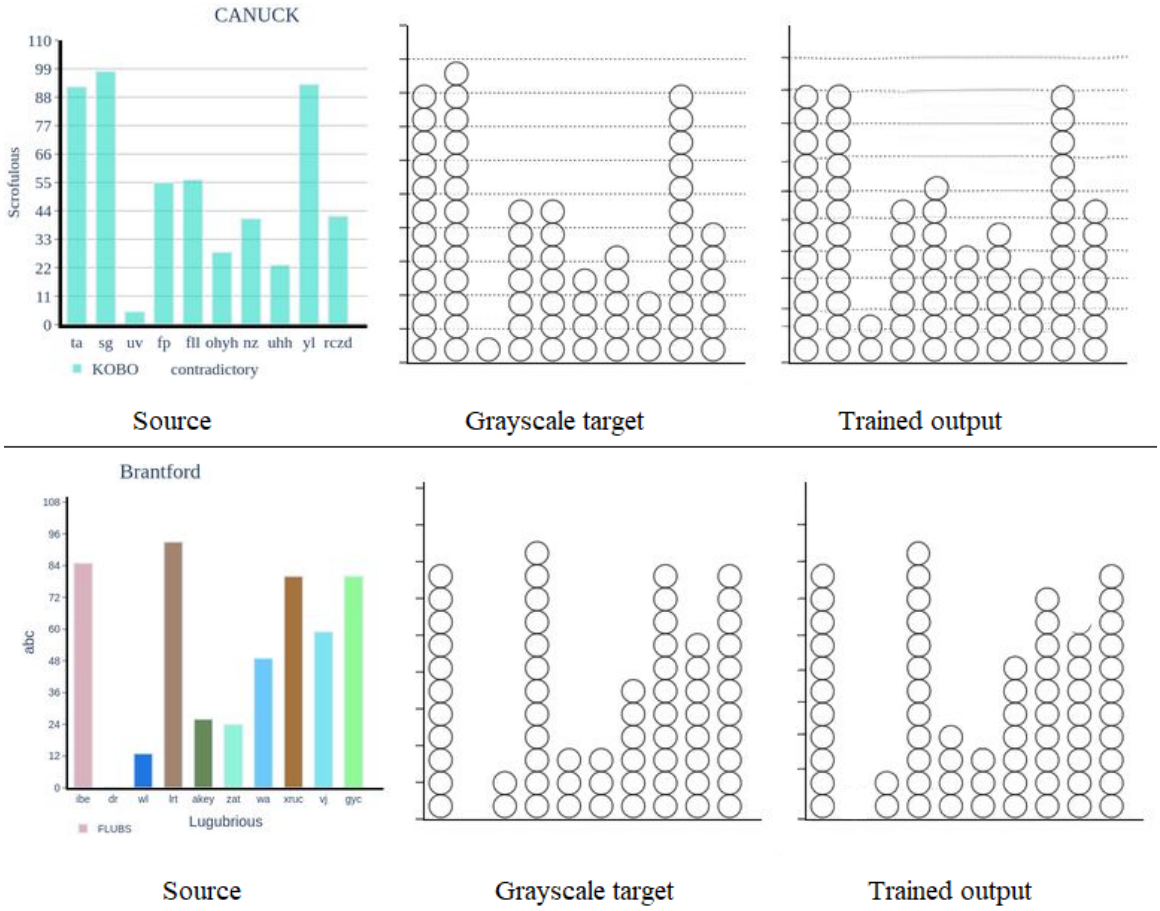


Figure 48: The final output produced by RGB Model on vertical bar charts with and without grids for 512x512 image dimension

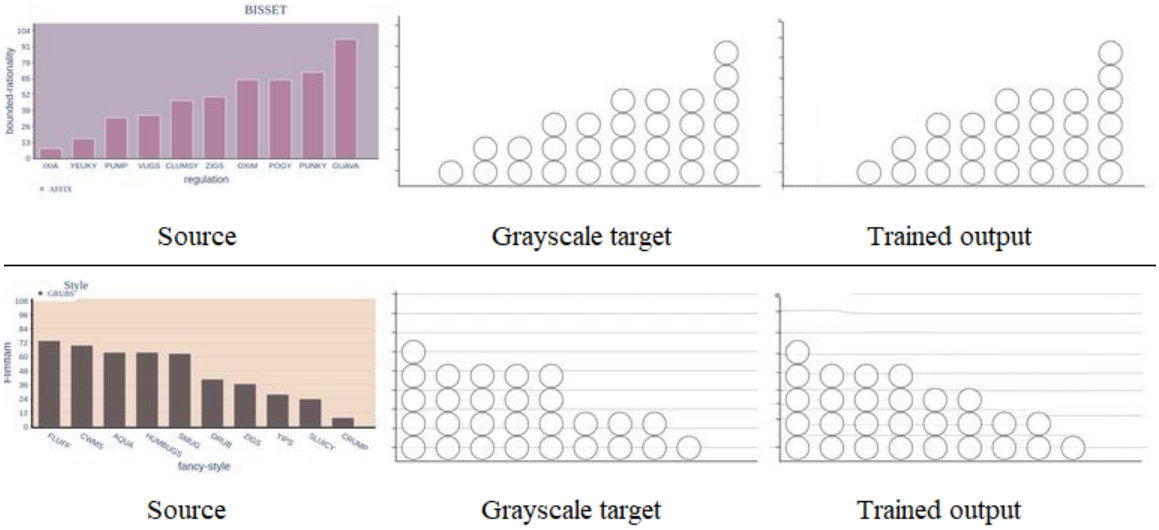


Figure 49: The final output produced by RGB Model on vertical bar charts with and without grids for 1024x512 image dimension

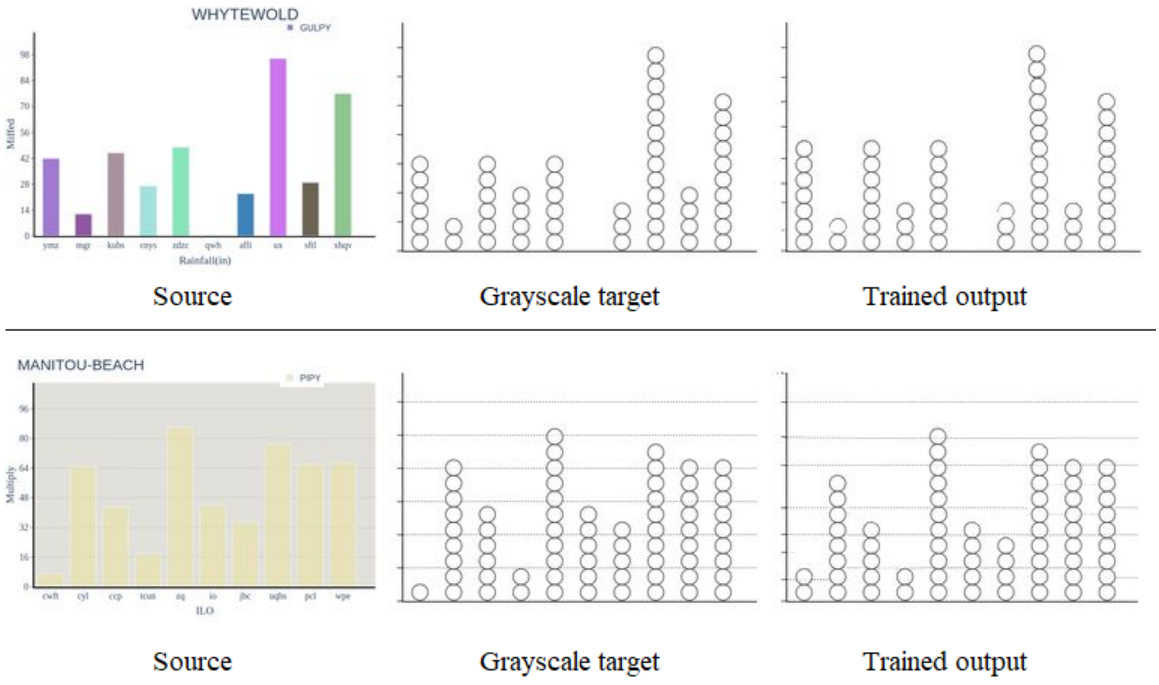


Figure 50: The final output produced by RGB Model on vertical bar charts with and without grids for 760x512 image dimension

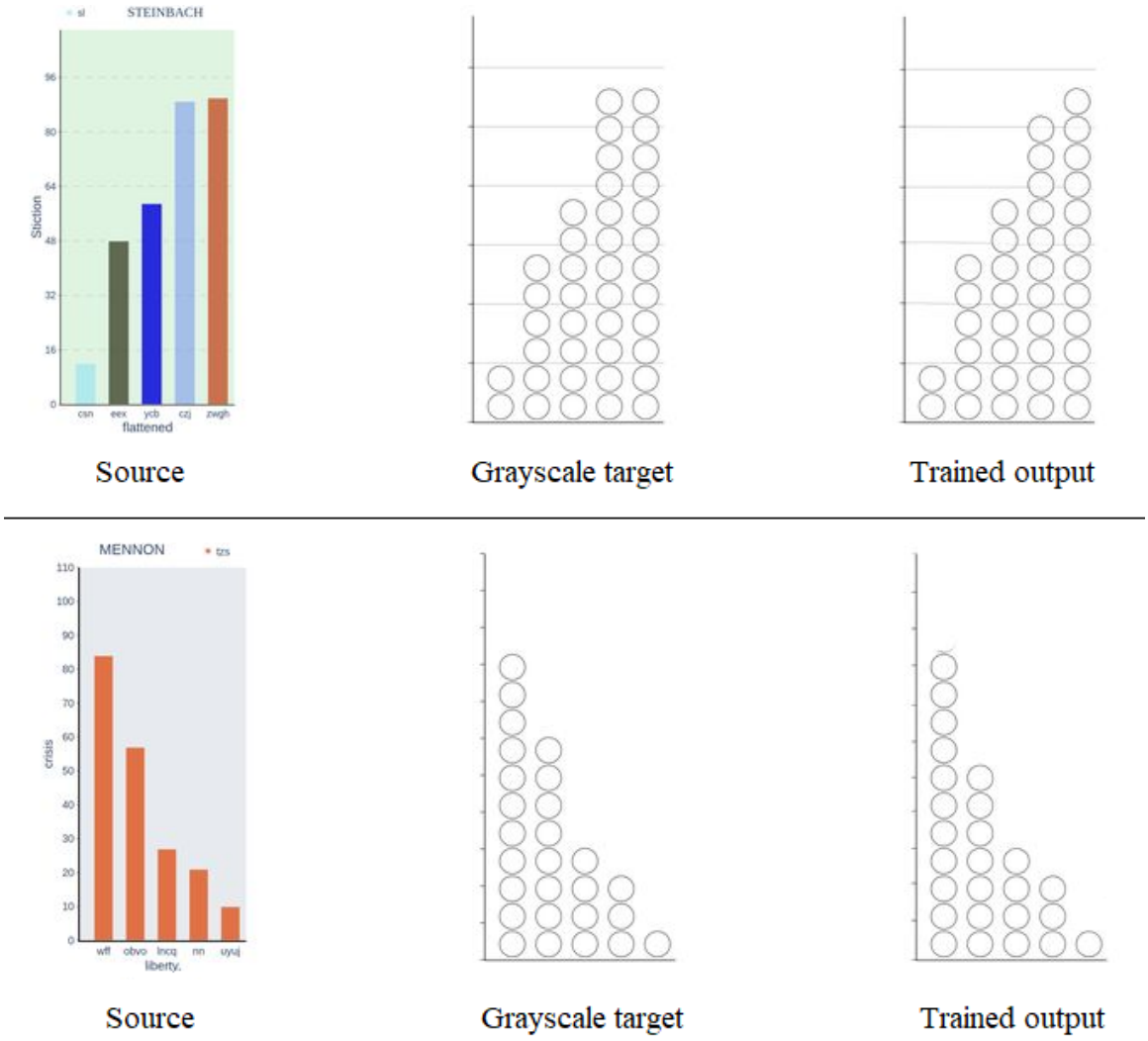


Figure 51: The final output produced by RGB Model on vertical bar charts with and without grids for 512x1024 image dimension

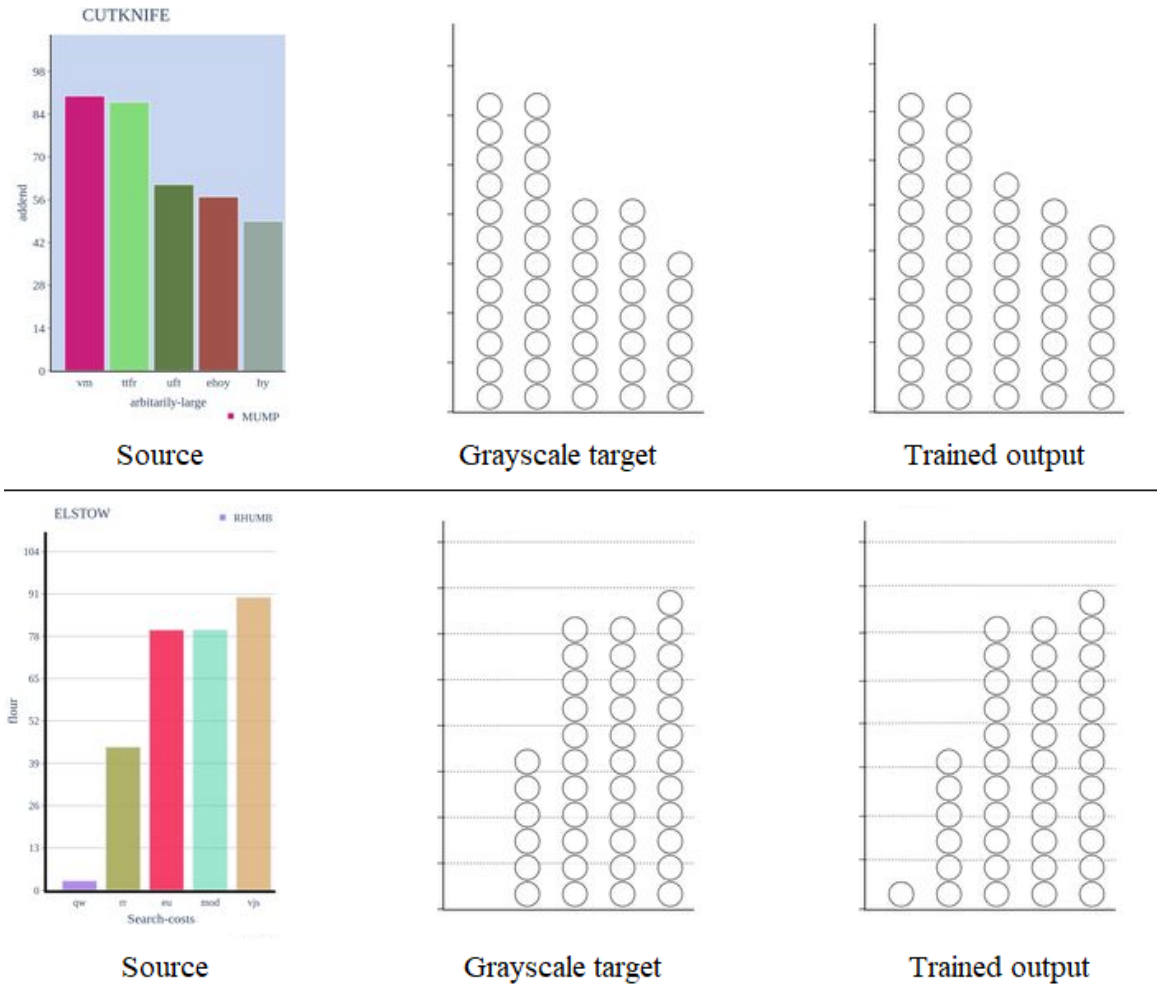


Figure 52: The final output produced by RGB Model on vertical bar charts with and without grids for 512x760 image dimension

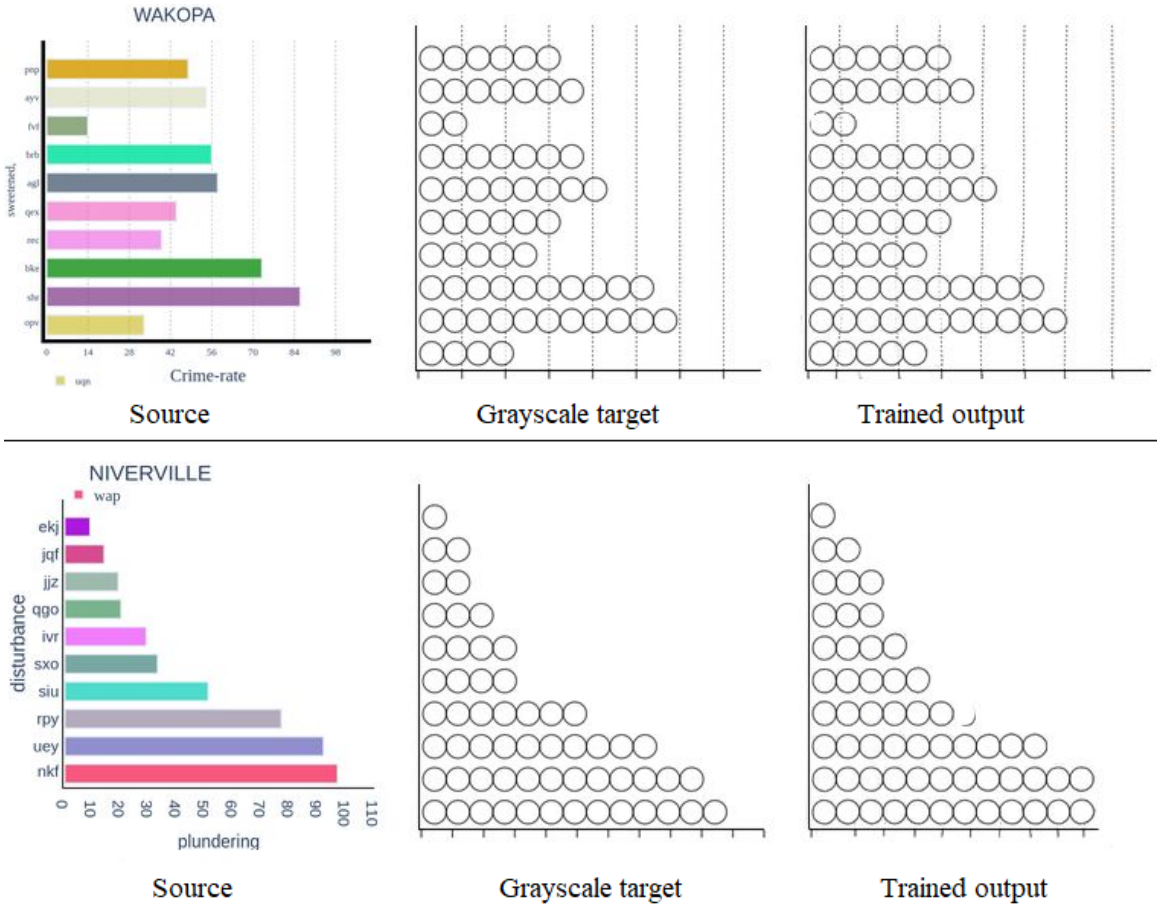


Figure 53: The final output produced by RGB Model on horizontal bar charts with and without grids for 512x512 image dimension

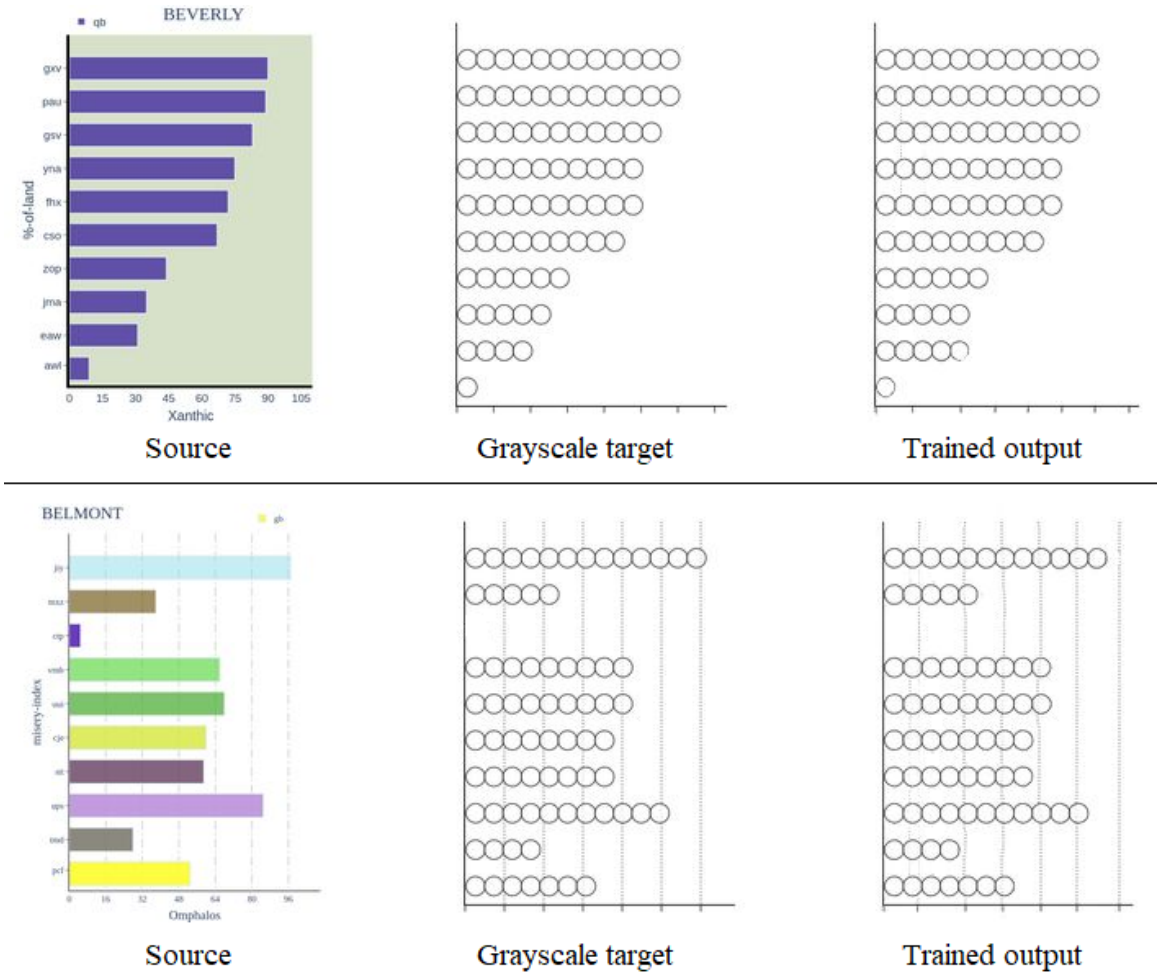


Figure 54: The final output produced by RGB Model on horizontal bar charts with and without grids for 512x700 image dimension

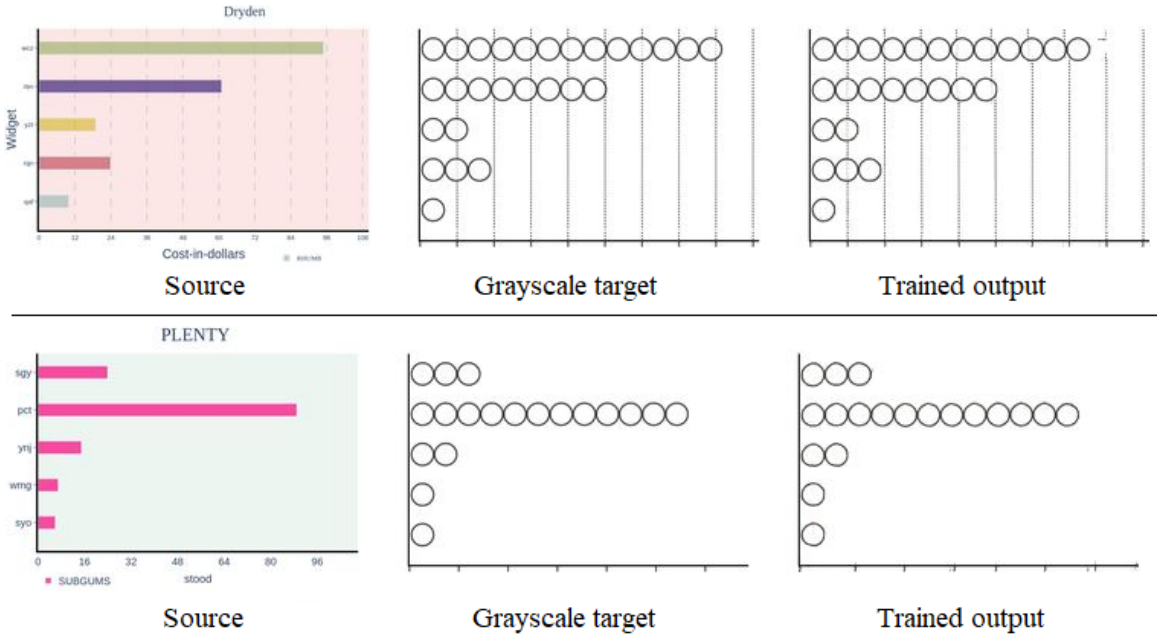


Figure 55: The final output produced by RGB Model on horizontal bar charts with and without grids for 760x512 image dimension

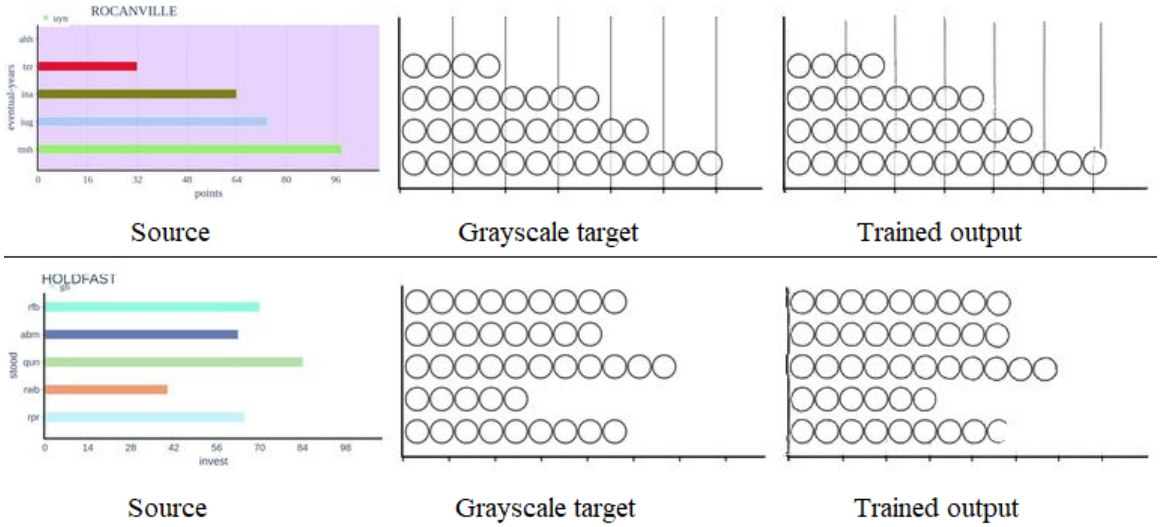


Figure 56: The final output produced by RGB Model on horizontal bar charts with and without grids for 1024x512 image dimension

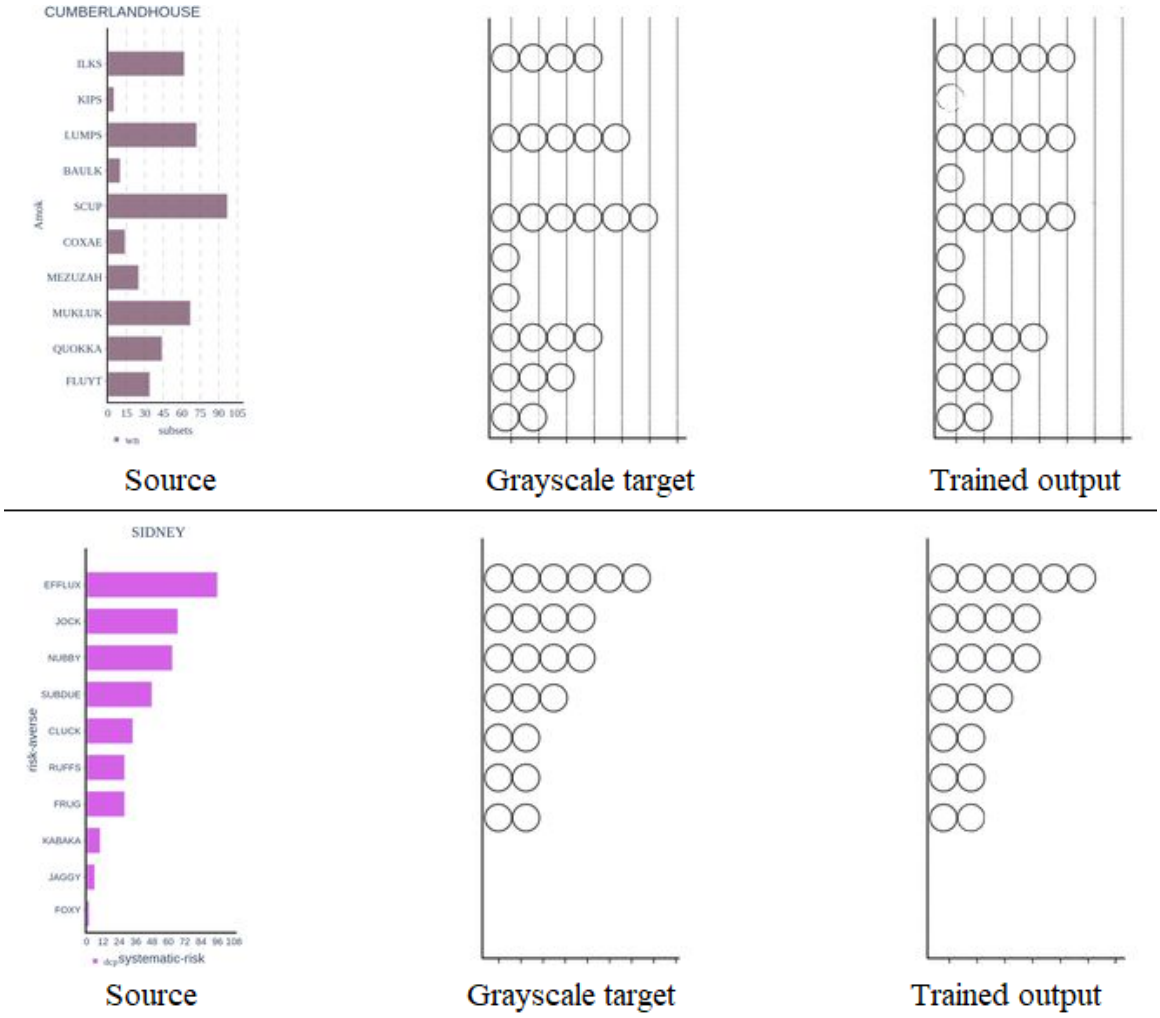


Figure 57: The final output produced by RGB Model on horizontal bar charts with and without grids for 512x1024 image dimension

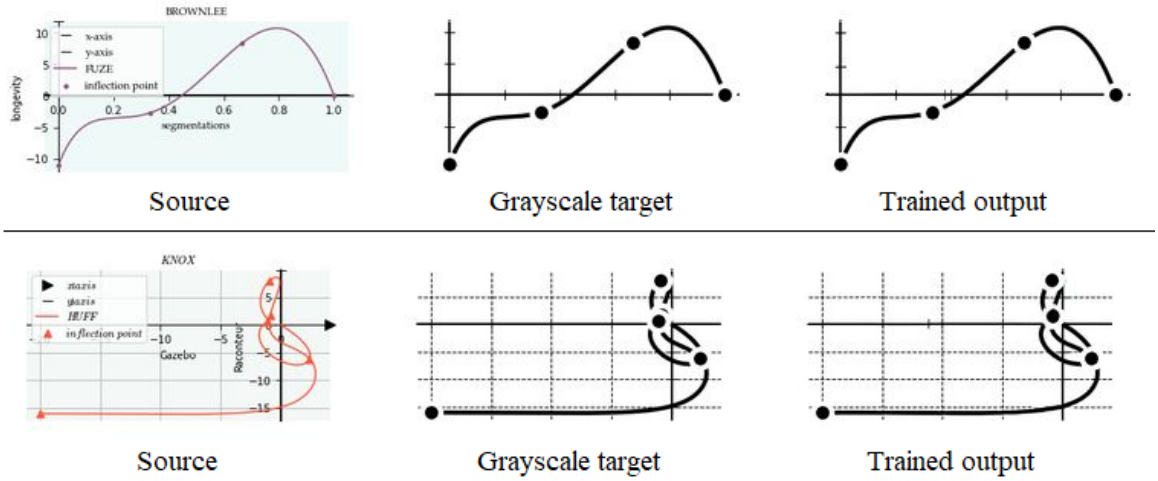


Figure 58: The final output produced by RGB Model on bezier curves with and without grids for 2:1 image aspect ratio

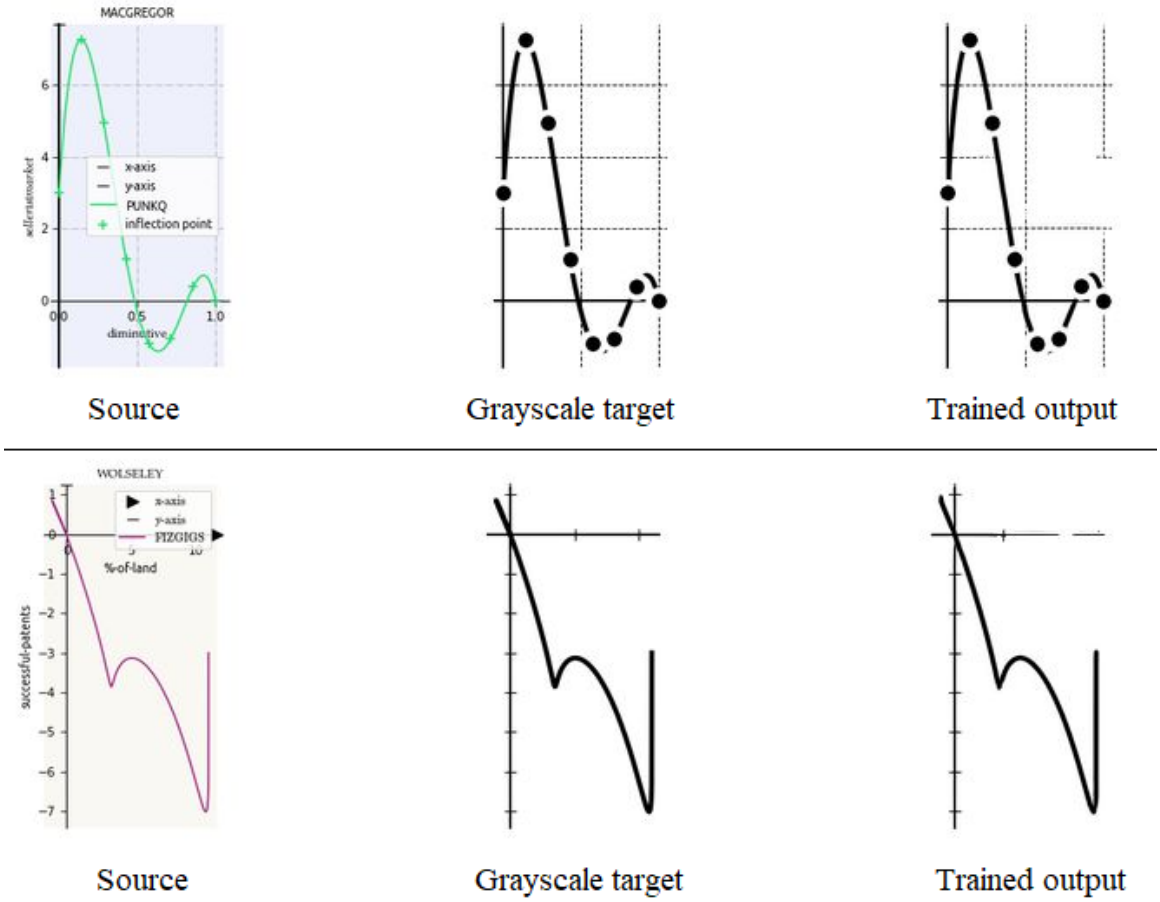


Figure 59: The final output produced by RGB Model on bezier curves with and without grids for 1:2 image aspect ratio

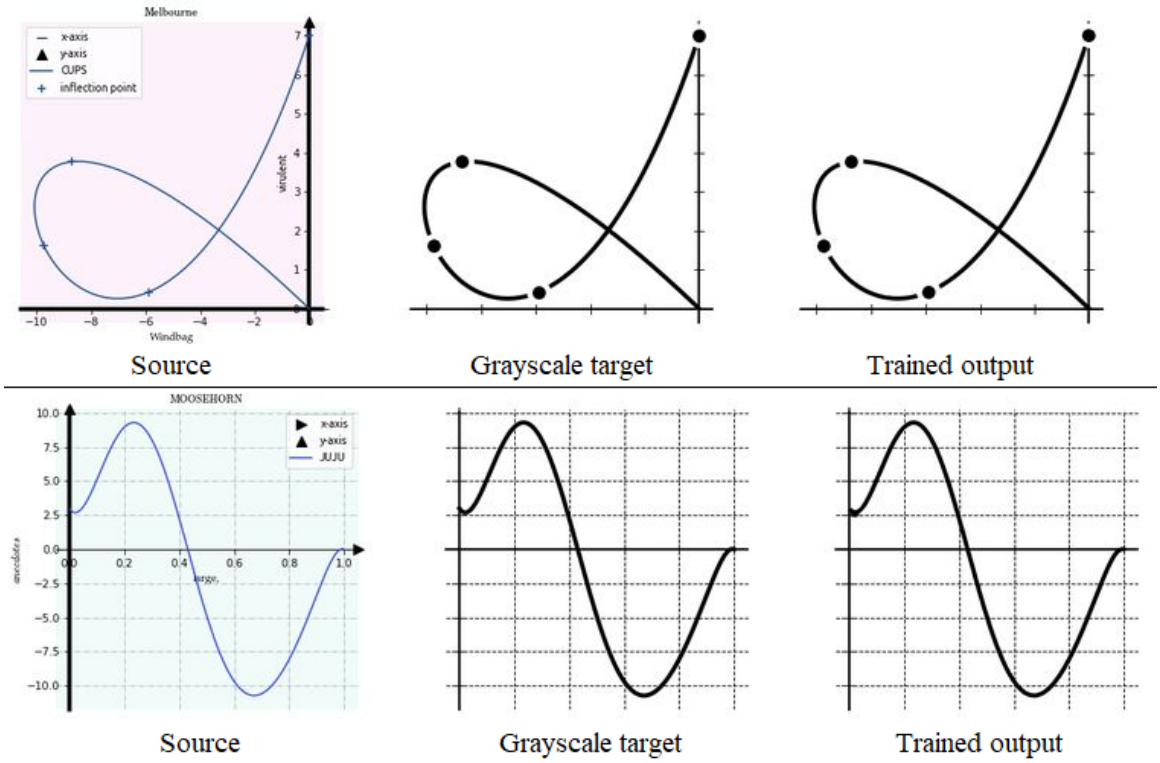


Figure 60: The final output produced by RGB Model on bezier curves with and without grids for 1:1 image aspect ratio

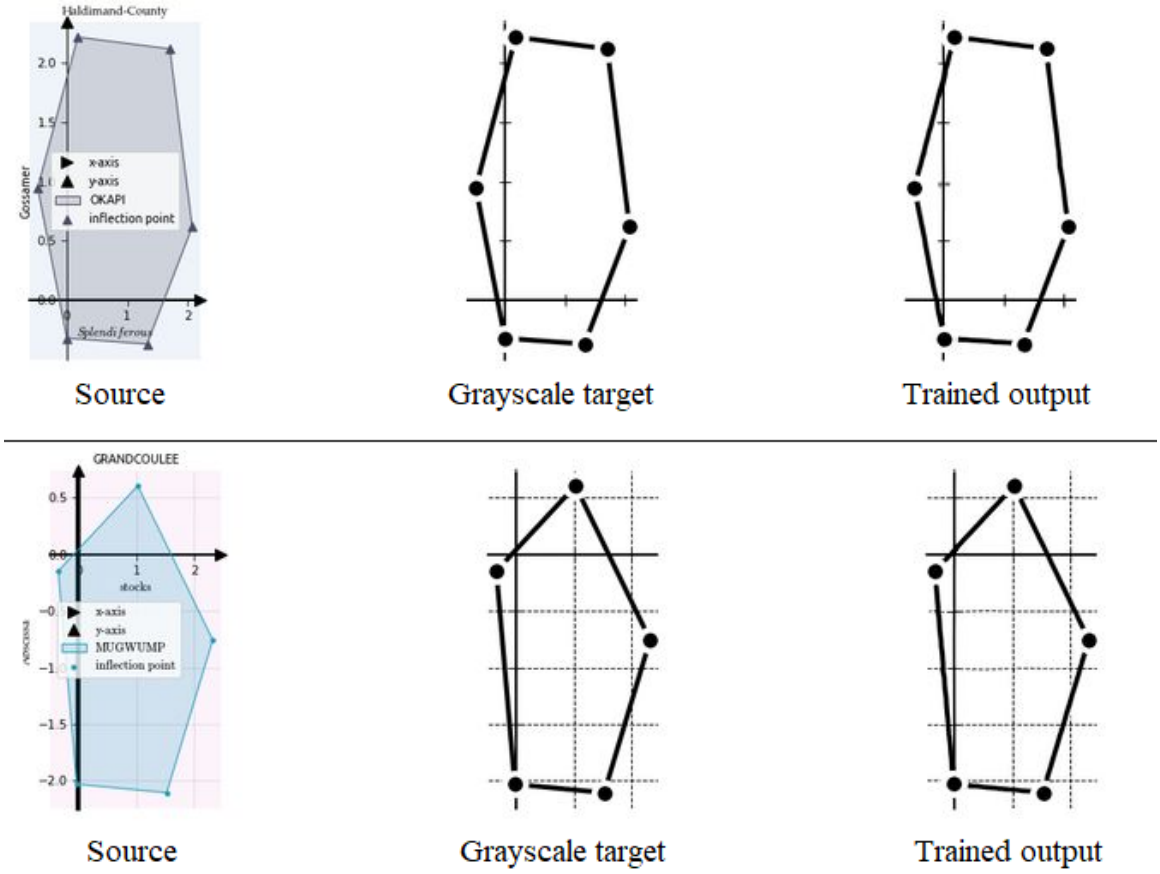


Figure 61: The final output produced by RGB Model on polygons with and without grids for 1:2 image aspect ratio

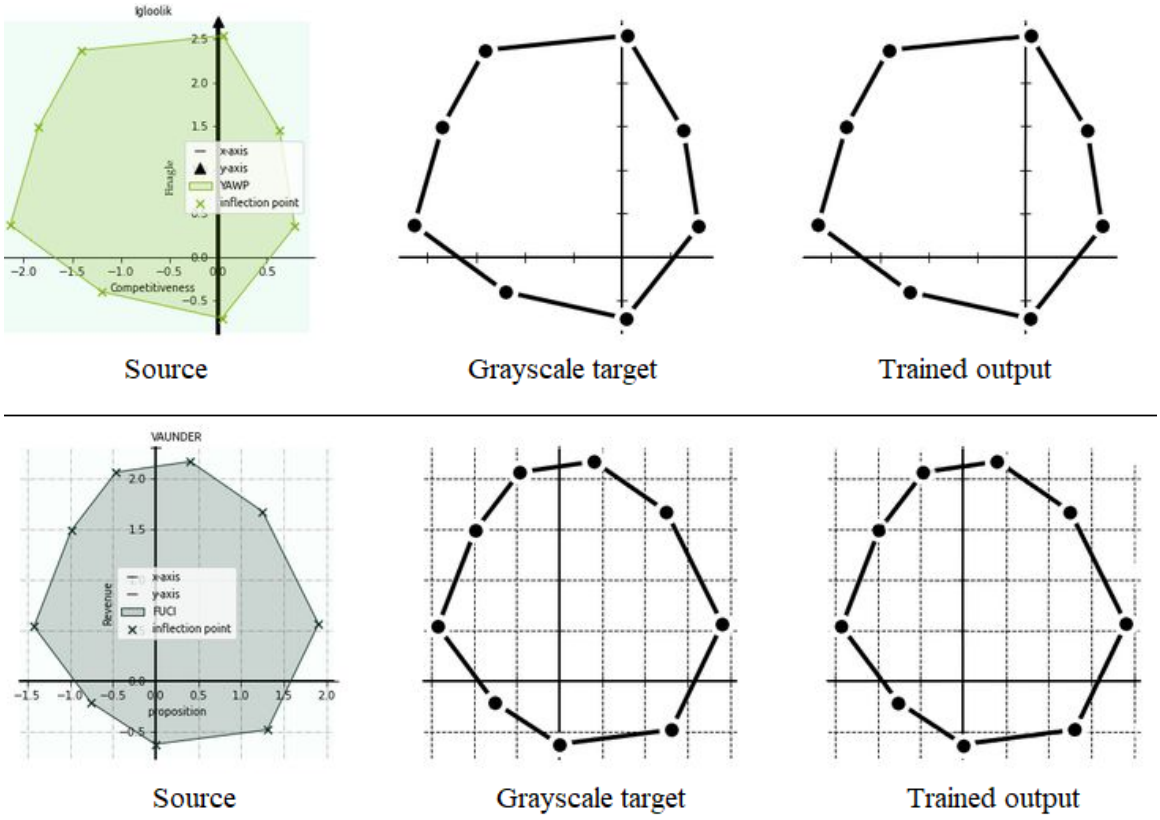


Figure 62: The final output produced by RGB Model on polygons with and without grids for 1:1 image aspect ratio

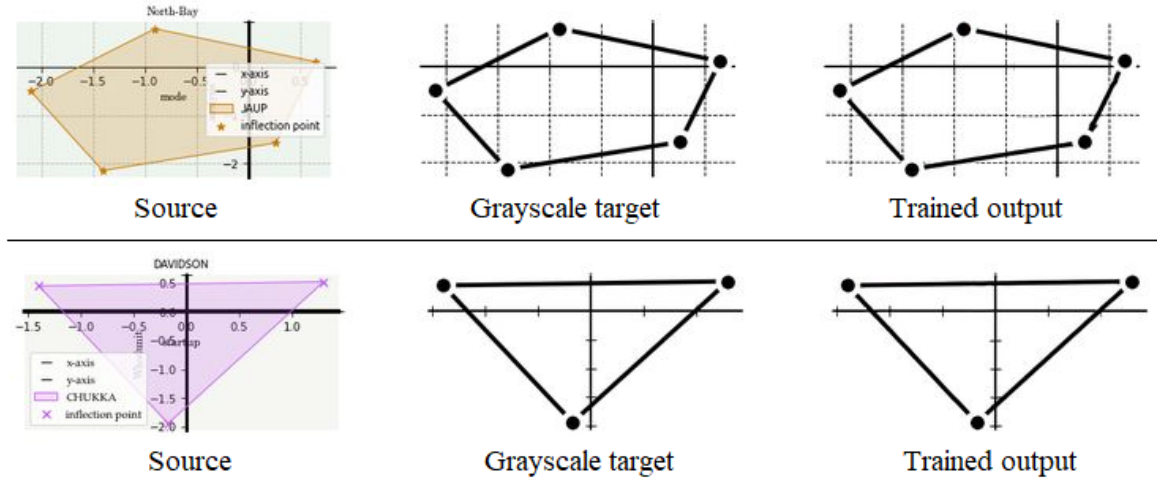


Figure 63: The final output produced by RGB Model on polygons with and without grids for 2:1 image aspect ratio

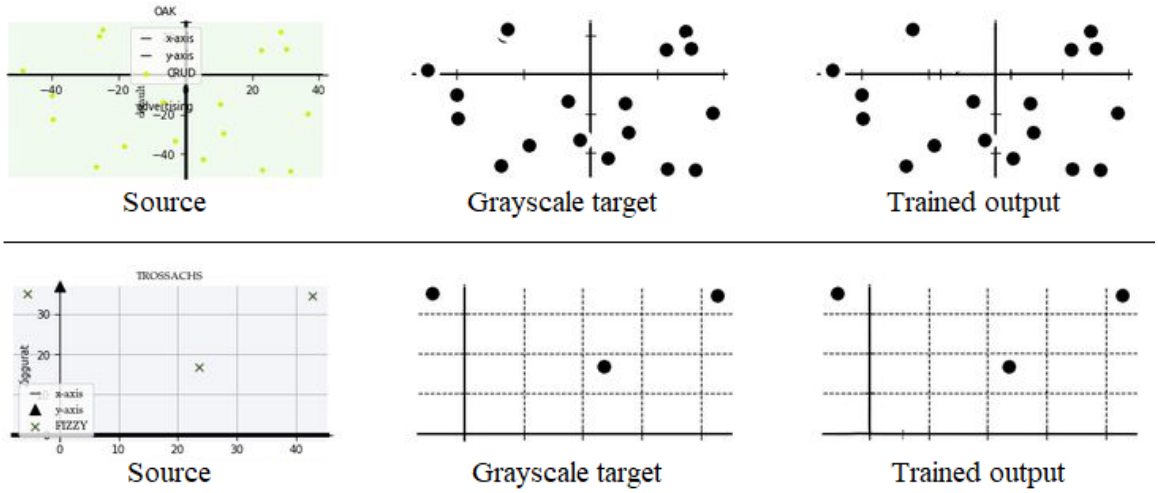


Figure 64: The final output produced by RGB Model on scatter plots with and without grids for 2:1 image aspect ratio

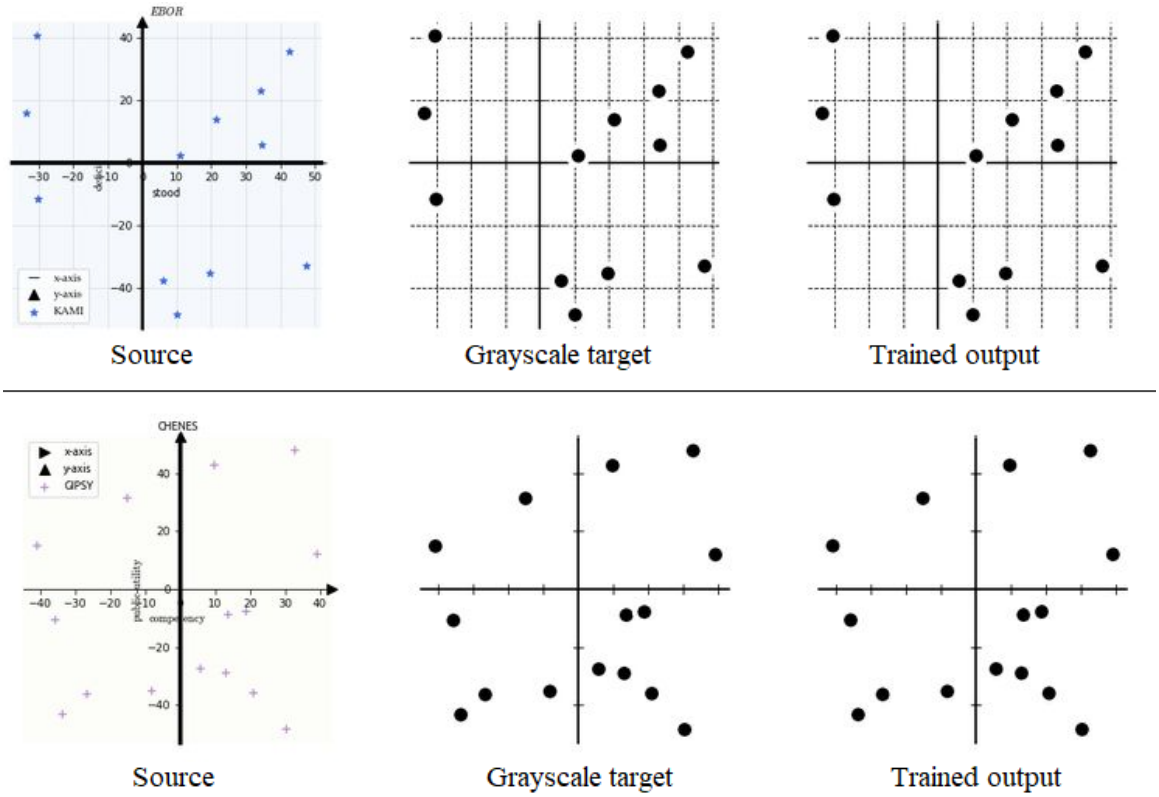


Figure 65: The final output produced by RGB Model on scatter plots with and without grids for 1:1 image aspect ratio

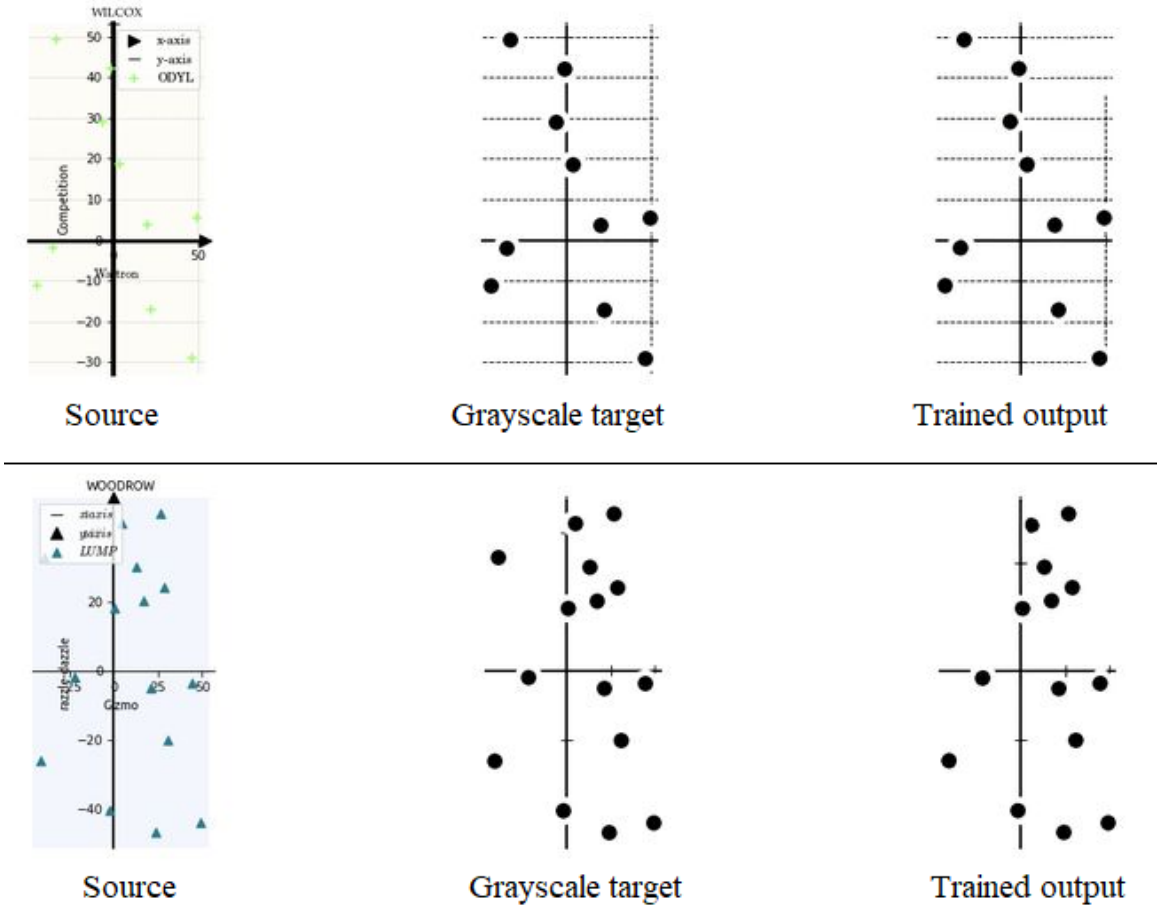


Figure 66: The final output produced by RGB Model on scatter plots with and without grids for 1:2 image aspect ratio

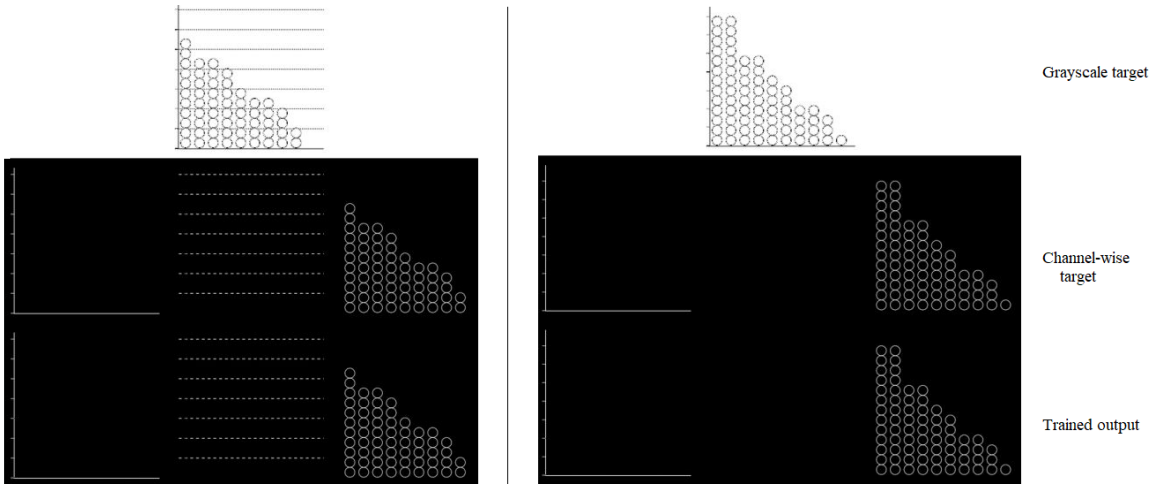


Figure 67: The final output produced by Channel-wise Model on vertical bar charts with and without grids for 512x512 image dimension

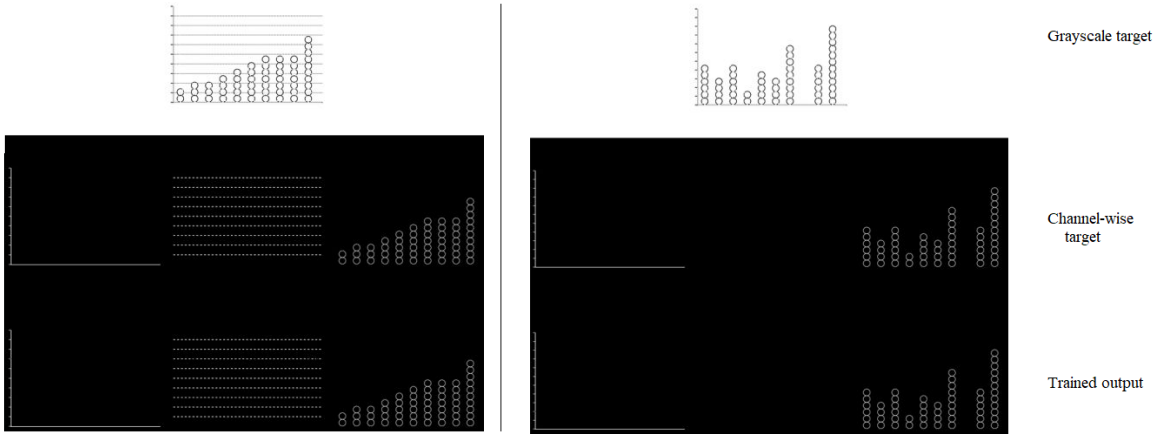


Figure 68: The final output produced by Channel-wise Model on vertical bar charts with and without grids for 760x512 image dimension

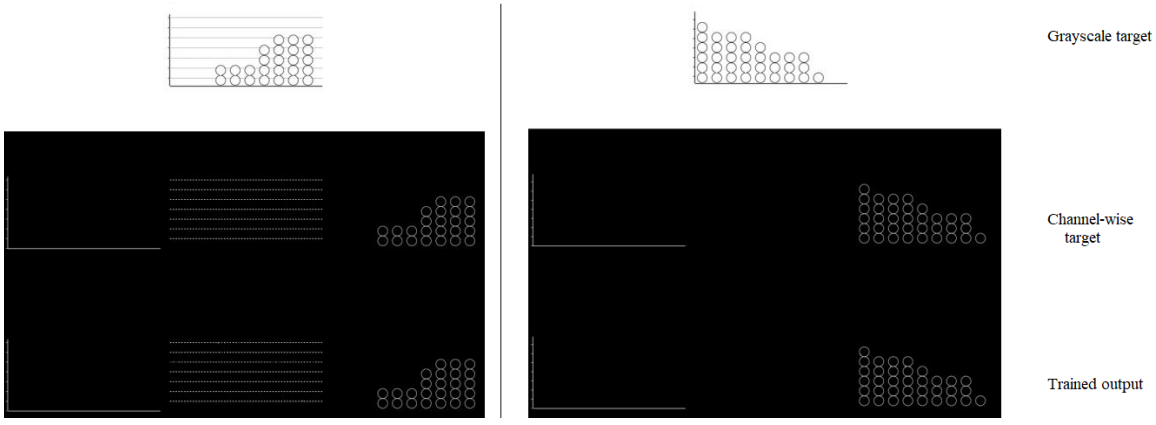


Figure 69: The final output produced by Channel-wise Model on vertical bar charts with and without grids for 1024x512 image dimension

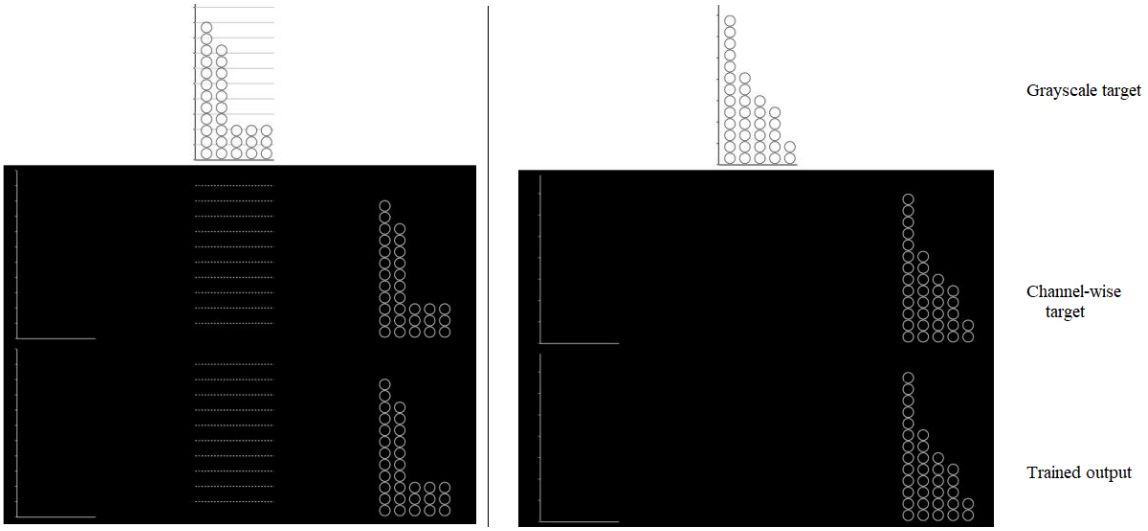


Figure 70: The final output produced by Channel-wise Model on vertical bar charts with and without grids for 512x1024 image dimension

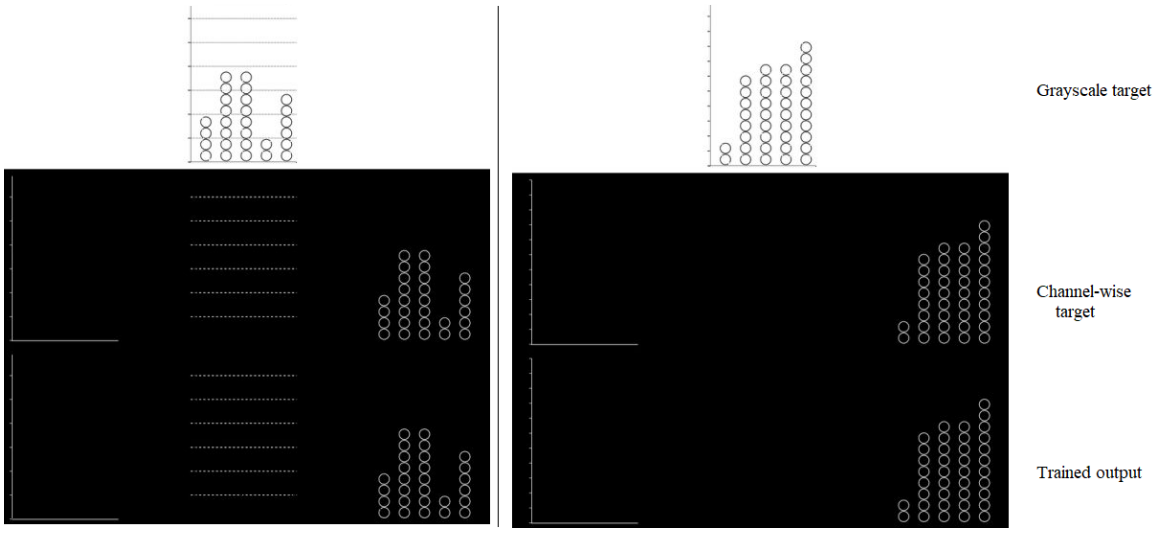


Figure 71: The final output produced by Channel-wise Model on vertical bar charts with and without grids for 512x760 image dimension

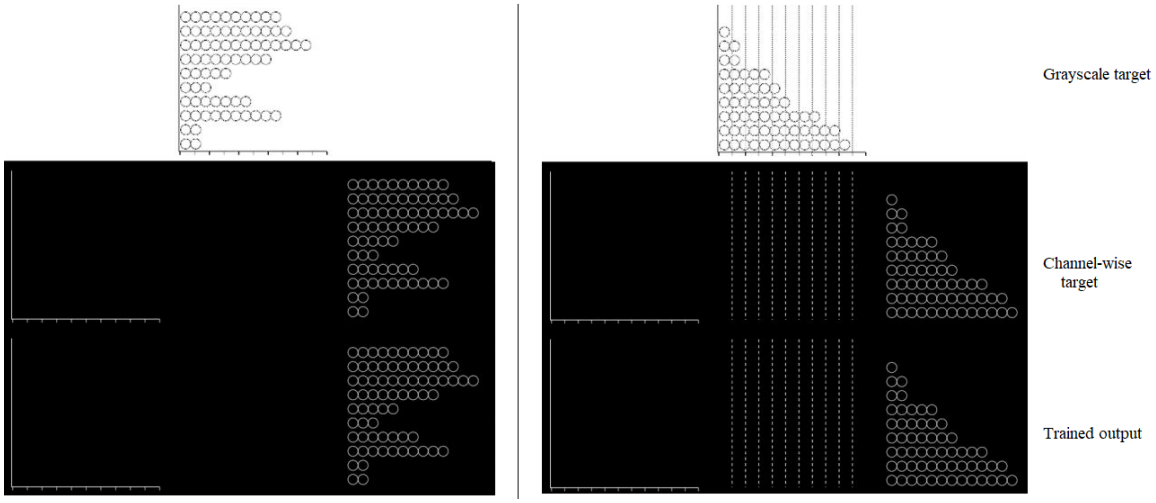


Figure 72: The final output produced by Channel-wise Model on horizontal bar charts with and without grids for 512x512 image dimension

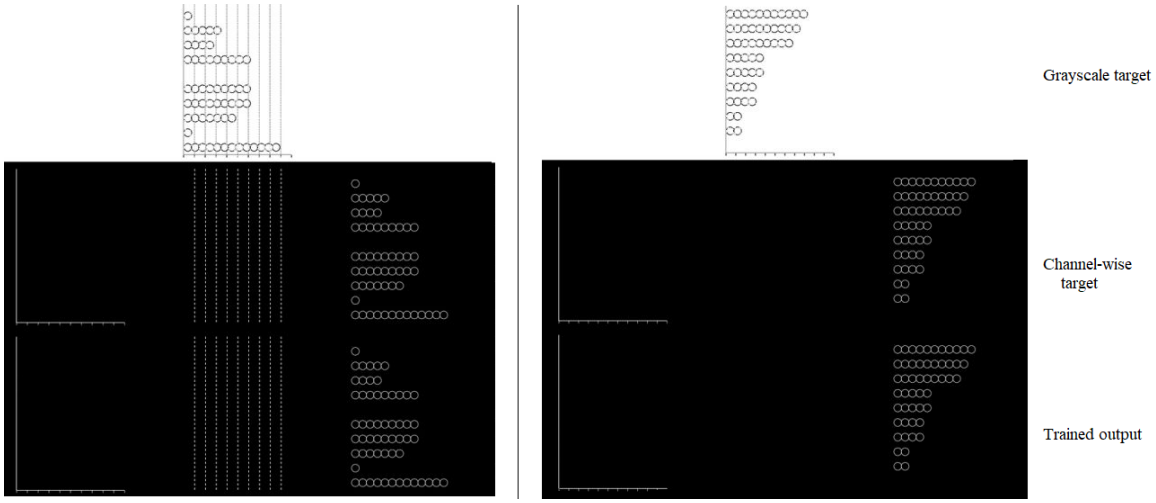


Figure 73: The final output produced by Channel-wise Model on horizontal bar charts with and without grids for 512x700 image dimension

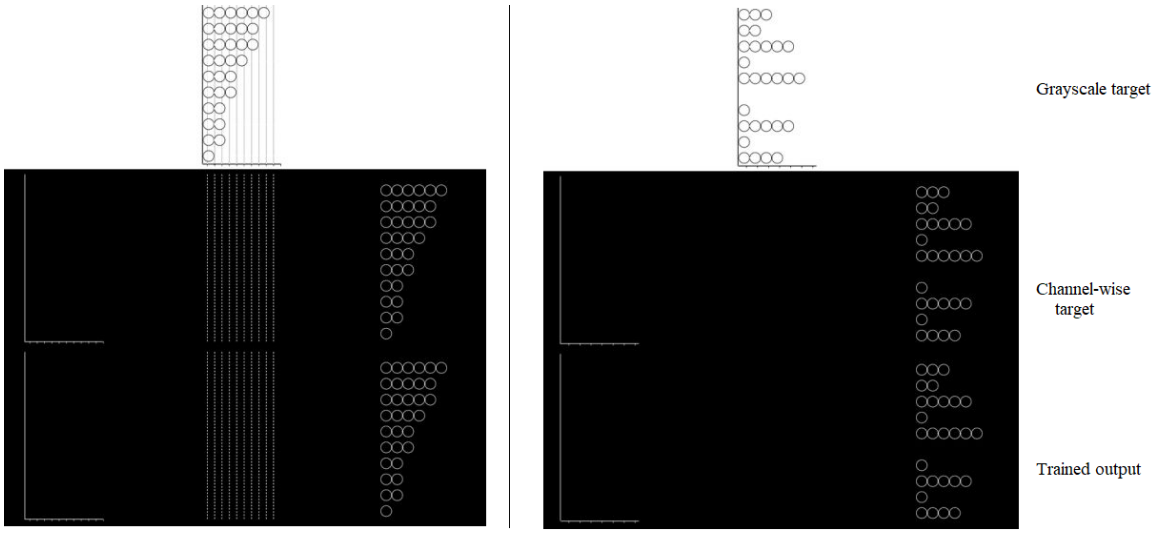


Figure 74: The final output produced by Channel-wise Model on horizontal bar charts with and without grids for 512X1024 image dimension

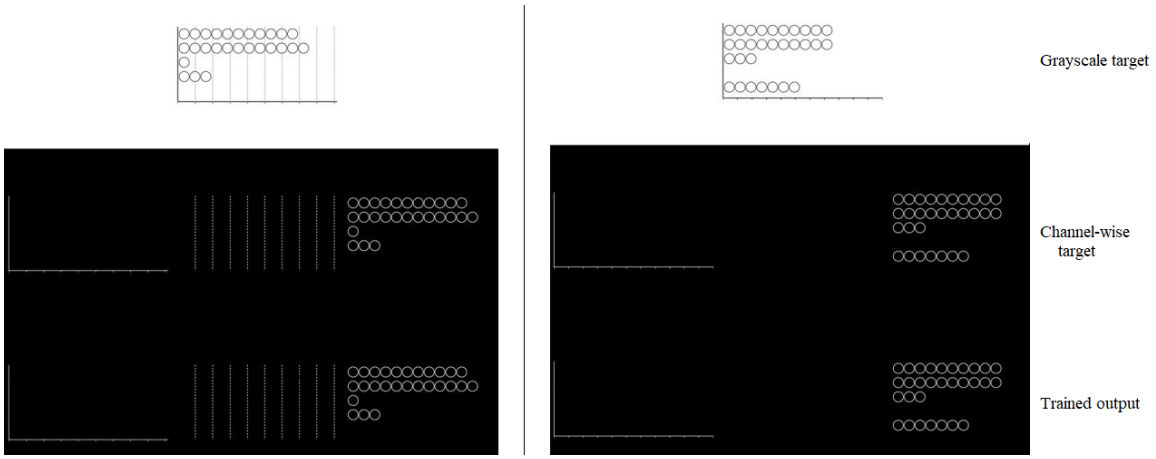


Figure 75: The final output produced by Channel-wise Model on horizontal bar charts with and without grids for 1024x512 image dimension

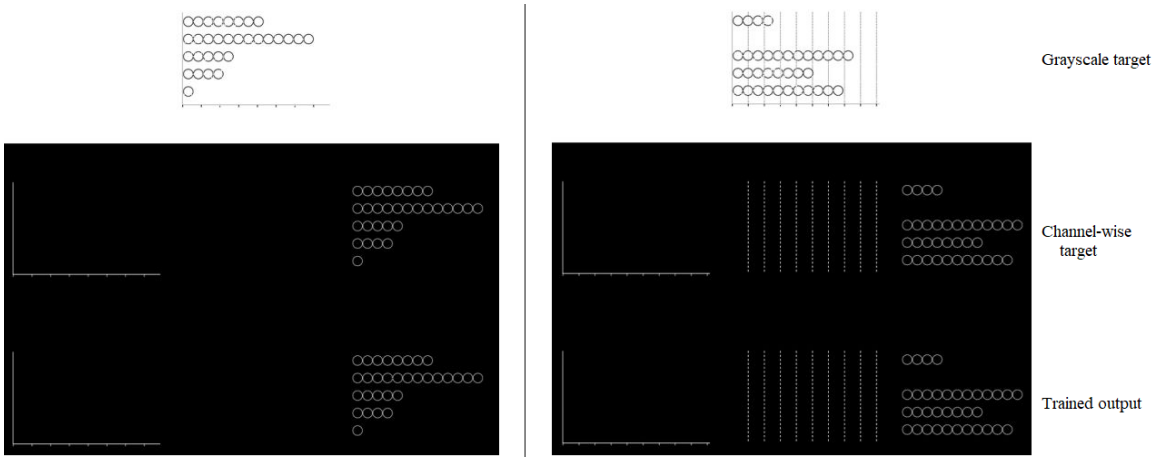


Figure 76: The final output produced by Channel-wise Model on horizontal bar charts with and without grids for 760X512 image dimension

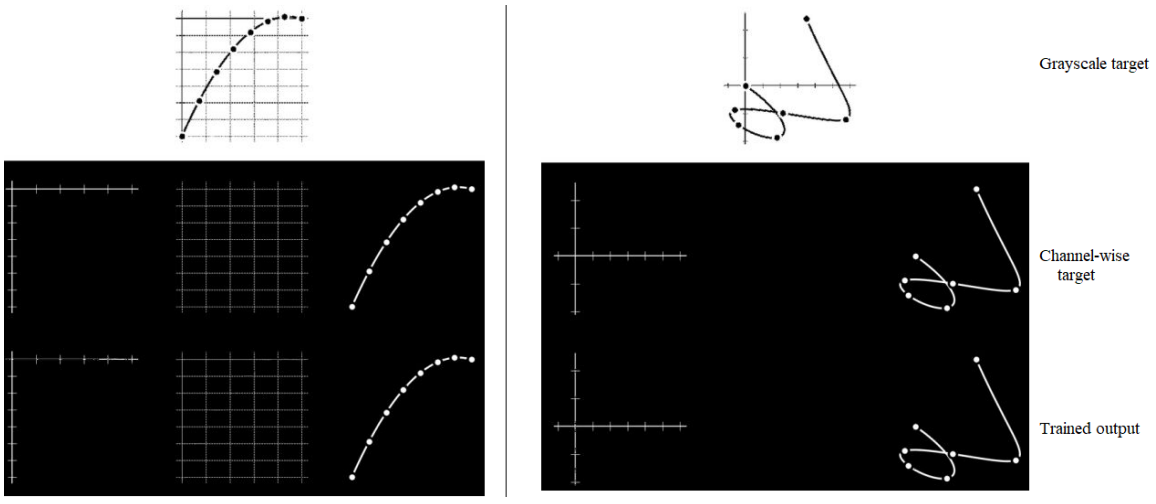


Figure 77: The final output produced by Channel-wise Model on bezier curves with and without grids for 1:1 image aspect ratio

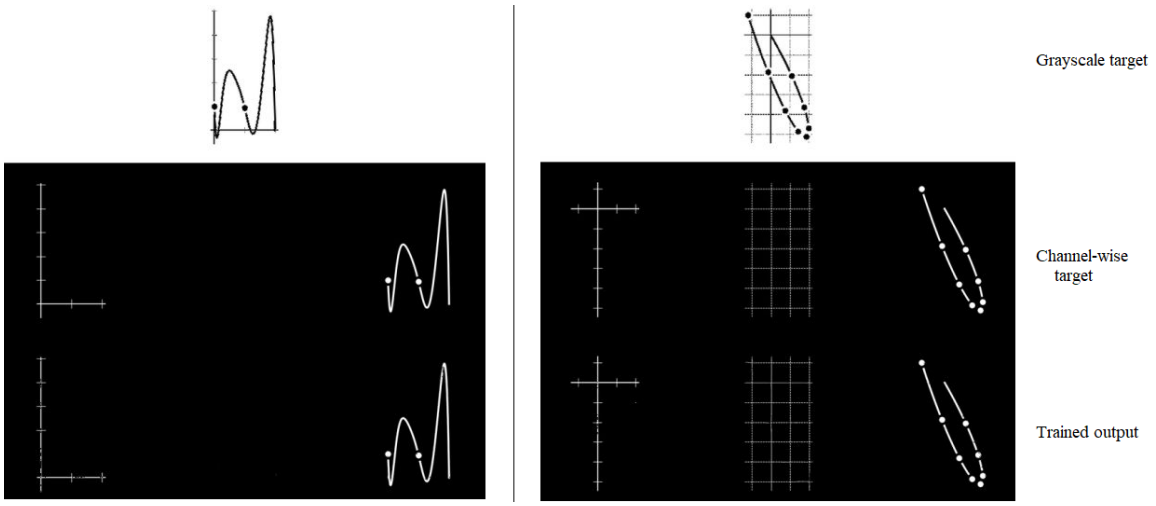


Figure 78: The final output produced by Channel-wise Model on bezier curves with and without grids for 1:2 image aspect ratio

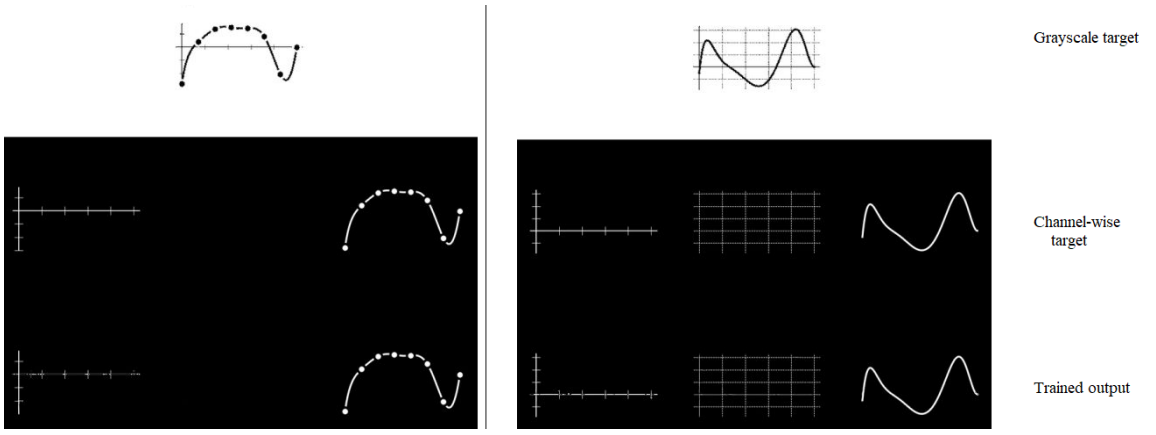


Figure 79: The final output produced by Channel-wise Model on bezier curves with and without grids for 2:1 image aspect ratio

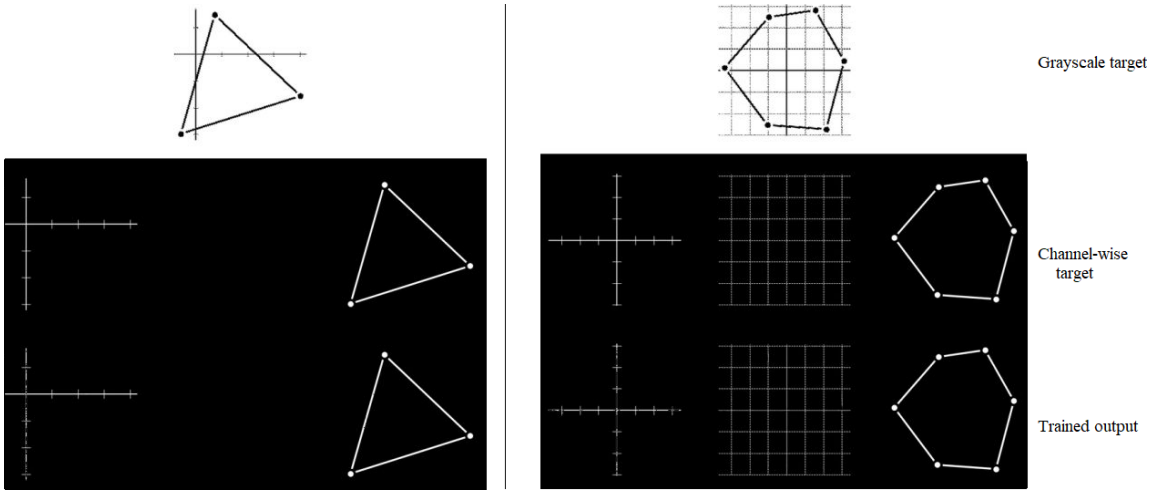


Figure 80: The final output produced by Channel-wise Model on polygons with and without grids for 1:1 image aspect ratio

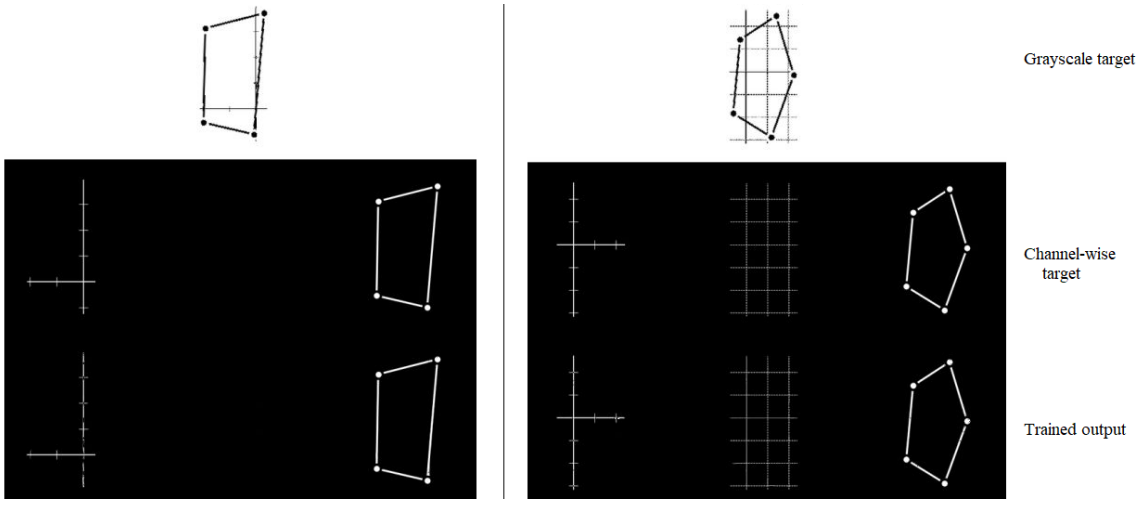


Figure 81: The final output produced by Channel-wise Model on polygons with and without grids for 1:2 image aspect ratio

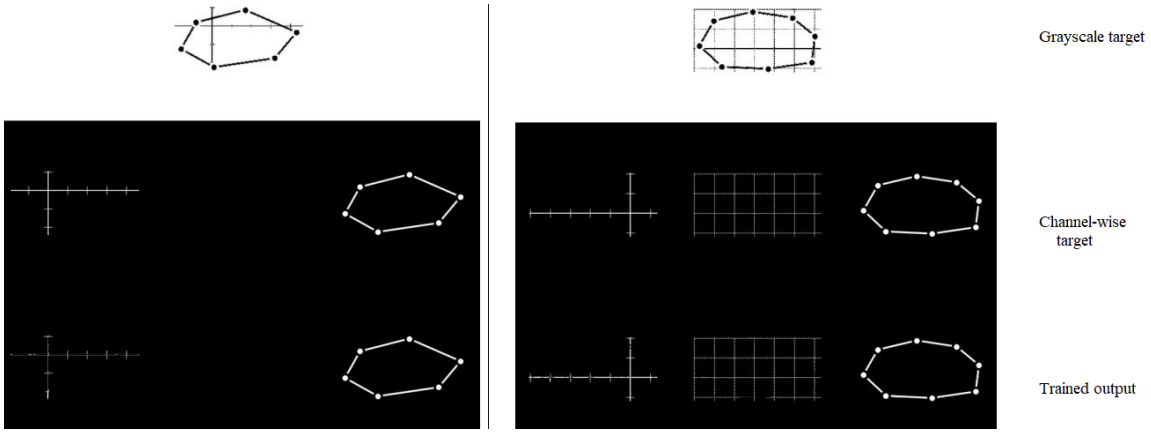


Figure 82: The final output produced by Channel-wise Model on polygons with and without grids for 2:1 image aspect ratio

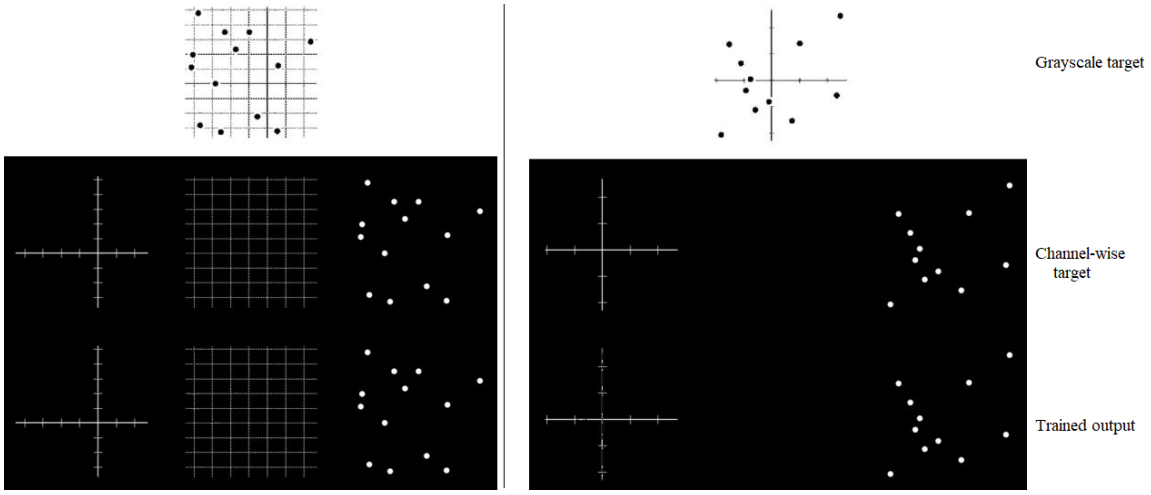


Figure 83: The final output produced by Channel-wise Model on scatter plots with and without grids for 1:1 image aspect ratio

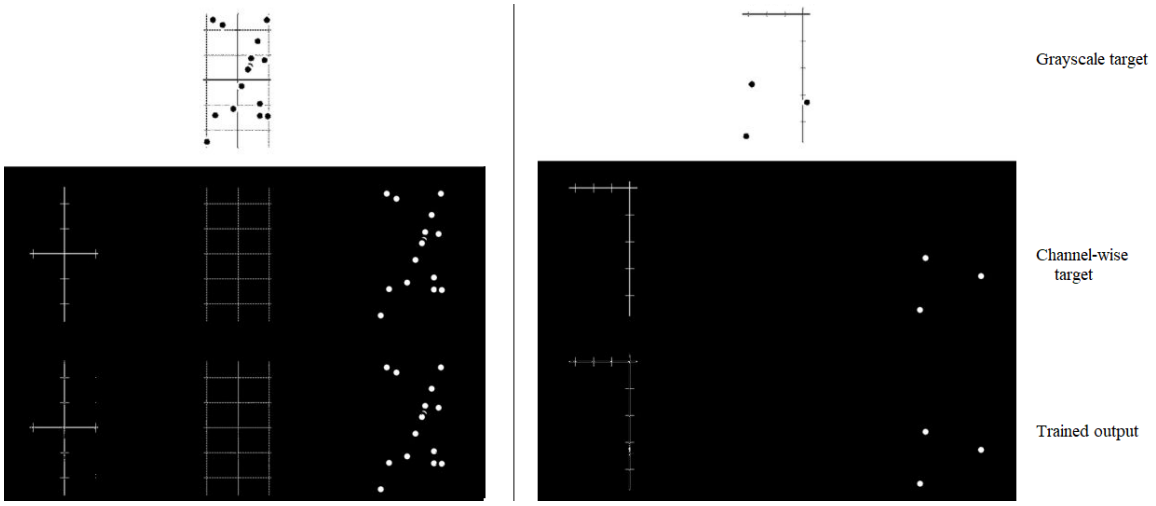


Figure 84: The final output produced by Channel-wise Model on scatter plots with and without grids for 1:2 image aspect ratio

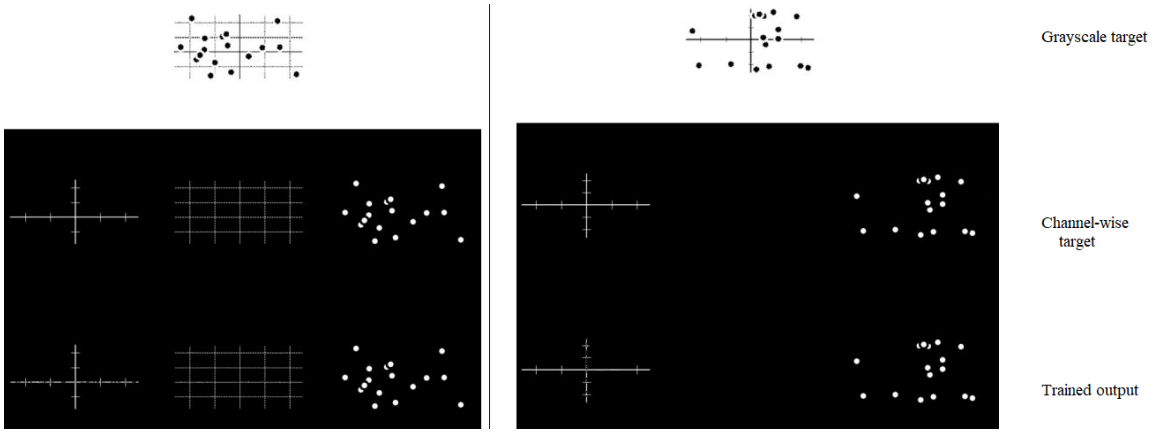


Figure 85: The final output produced by Channel-wise Model on scatter plots with and without grids for 2:1 image aspect ratio