CS 520 Introduction to Artificial Intelligence
Professor Wes Cowan
Assignment 4: Colorization

Keya Desai (kd706), Prakruti Joshi (phj15), Rushabh Bid (rhb86)

December 2019

1 Introduction

The purpose of this project is to understand the concept of neural networks in Artificial Intelligence and explore various implementations of the same. The given problem aims towards teaching a computer about images and colors. Given a gray scale image, the task is to add a splash of color to it. Typically, each pixel of an image has three channels, *(r, g, b)*, where each value represents the intensities of color red, green and blue respectively. Learning models are used by the computer to comprehend things that need rationality. The input to any model would be a gray scale image. The model will then learn based on some data, and generate output based on what it has learned. Neural networks can be visualized as highly parameterized functions that fit to the data provided to it while trying to generalize the parameters for other similar inputs. Tuning the neural network with appropriate parameters and learning the parameters according to our problem is the main challenge. For this problem, we have implemented and tested the following approaches.

1. Direct mapping from a single gray-scale value gray to a corresponding color *(r, g, b)* on a pixel by pixel basis
2. Mapping a set of gray values to a single *(r, g, b)* value, which is the color of the central pixel of the set. This is the essence of Convolutional Neural Network.
3. Shifting from a *regression* problem to a *discrete classification* problem. Instead of trying to determine the exact color of the pixel, it maps to one colour out of a palette of K colors.

Each approach is discussed in detail in the following sections.

2 Data

A set of colour images is taken as the data. First, we convert the coloured images to gray images using the formula given by:

$$Gray(r, g, b) = 0.21r + 0.72g + 0.07b \tag{1}$$

where the resulting value Gray(r,g,b) is between 0 and 255, representing the corresponding shade of gray (from totally black to completely white).

The gray images are given as the input and the corresponding colored images are given as the desired output. For testing purposes, we over-fit the model by training the model to a particular class such as images of tree, sky, apples etc. We then, tested if the model was able to generalize it to images of the same class.

3 Approaches

In this section we describe the approaches we have tried out to solve the problem. For all approaches, the following things are general:

- Perceptual error: Since, we are working with images, the evaluation of the accuracy is much better described by the perceptual error. While, we can define a mathematical loss function, perceptual error is of more relevance as we should be able to relate to the coloring of the image.
- Activation function: We tried several activation functions - Linear, Tanh, Sigmoid, ReLU and Leaky ReLU. Out of this, sigmoid was giving the best results. This seems plausible. The other three activation functions have a much bigger range than the range of desired output. For the model to learn to confine the output in the output range, it would require a larger training data-set. On the other hand, if we normalize the pixel values to [0,1], the sigmoid activation is already confined to the acceptable range for the output.
- Weight initialization: We have tried various weight initialization techniques - random initialization between [0,1] for linear activation, Xavier initialization for tanh and sigmoid activation and He initialization for relu and leaky relu activation.
- How did you determine convergence? How did you avoid overfitting? The goal of training is to minimize the overall loss of the network. It can be observed that the mathematical mean-squared loss decreases over epochs, but we desire to minimize the perceptual loss as we need the images to be sensible rather than exactly matching the desired result. We tune the number of epochs to get an appreciable perceptual accuracy and declare it a point of convergence.
  However, we do not want a converged neural network that only works for the training data. We want to generalize the learning to some extent. Given the conversion formula from a color image to gray-scale, a single gray-scale value maps to a raft of *(r, g, b)* values. As a result, the probability that the model would over-fit to the input data is minimal in a practical number of epochs.
- Evaluating the model:
  1. Loss/Error function: Mean squared error.

$$loss = \frac{1}{2} \sum_{i \in data} (predicted_i - output_i)^2 \tag{2}$$

  2. Learning Algorithm: Mini Batch Gradient Descent for vectorised implementation
  3. Learning factors: Learning rate, Batch size, Number of epochs, Number of hidden layers, Nodes per hidden layer, Activation function

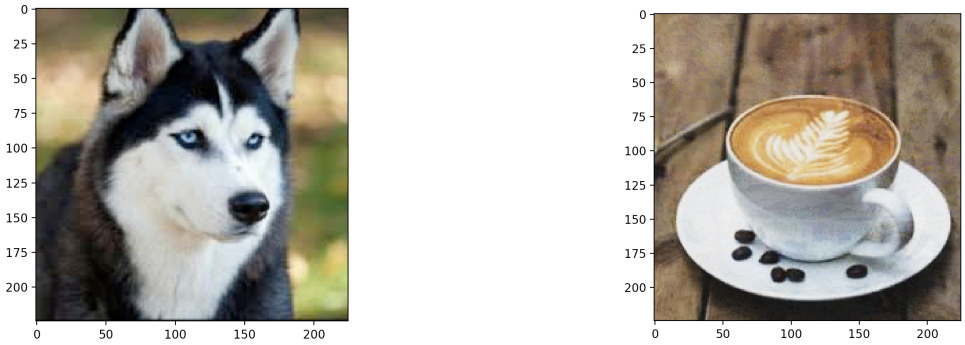
3.1 Vanilla Neural Network

We started with a basic neural network with specified number of hidden layers and number of nodes in each layer. The main aim was partly to understand and implement the working and performance of neural networks and partly to observe the efficiency of neural networks on predicting the direct mapping from gray to *(r,g,b)* of a particular pixel. Thus, this forms a regression problem with the output values for each class in the range of [0,255]

Representing the Process: We took 80 images of shape (225×225) from the internet. The images belonged to several different classes such as tree, dog, cat, ships, sky etc to train and test the model on. To reduce the complexity of the input data and improve learning of essential features, we took images of the same shape so that the shape of the network is uniform. One of the drawbacks of basic neural networks is that it dictates the input images to be of the same shape. Thus, in our problem, if we had images of different sizes in the data-set, we would have to downsize them or add padding to the smaller images to bring them to a common size. This might have resulted in loss of information. We are normalizing the training and testing input and output by 255 (the range of gray-scale intensity).

Model:
Input space: Single gray image (of a coloured image) is given as input. Each image has (225 × 225) pixels amounting to 50625 nodes in the input node as we have flattened the 2-D image into a single array.
Output space: The corresponding coloured image of the gray image is the output. Each pixel has three channels as RGB values, thus there are three nodes in the output. The aim of the model is to accurately predict the RGB values for each pixel of the image.

Results: The vanilla neural network takes a large input in terms of the number of gray values and maps to an even larger input in terms of the corresponding color values. To train a vanilla neural network on such large images, it would need a large number of dense hidden layers to generalize. That would become computationally expensive. Training it with minimal hidden layers with a small number of neurons, it will adjust the weights to be so high that the output will be independent of the input and simply return an over-fit output. This over-fit is depicted in Figure 1.



(a) Training input

(b) Training output

Figure 1: Result of Vanilla Neural Networks: For every gray-scale input image, the model returns the same coloured image which was the last in the training set.

3.2 Single CNN layer

In this approach, we feed the model with the surrounding pixels w.r.t. the center pixel along with the central pixel in the form of a window of size (n × n). Thus, suppose the window size is (3 × 3), the input to the model will be a flattened array of (3 × 3) with the center value as the pixel whose coloured image we want to predict. Thus, in this example, the input array will be (1 × 9). Depending on the window size of the filter, we pad the image from all the sides to avoid bias for central pixels. Passing the neighborhood information ensures that the model is learning the additional context and information to generate the color for the central pixel. This is followed by the hidden layers whose activation values are used to predict the output (RGB) values. This acts as a single layer of CNN with the model learning the filter values as the weights of the first layer.

Representing the process: In this model, we pre-process the data by generating training and testing data by giving the entire window of the pixel as the input and the corresponding colored central pixel value as the output. We have formed the batches such that one batch has the decomposed values of input and output per pixel of an entire image. Thus, by using batch size as one, we update the weights after the error of an entire image is calculated. We similarly reconstruct the image while testing and prediction.

Model:
Input space: Input to a single pass of the model is the gray-scale pixel values in a window w.r.t to a central pixel. This is passed for all the pixels in an image.
Output space: The output space is similar to the basic neural network as the model is predicting the RGB

values for a corresponding pixel.
The corresponding coloured image of the central gray pixel is the output of one pass of the CNN layer.
Learning Algorithm: Batch Gradient descent

Implementation Variants:

- **Separate networks for RGB value:** Training the model to predict the values for red, green as well as blue was giving some "gray" results. One possible reason could be that since the values of the three channels are independent of each other, training them together could introduce some dependencies.
- **Stochastic Gradient Descent:** Earlier in batch gradient descent, we were updating the weights and propagating the error after an entire batch was forward passed into the network and loss was stored. In our context, while performing batch gradient descent (one batch is one image), we are propagating the overall average error of each color instead of propagating the error for each pixel individually. The model should sustain the individual fluctuations in each update observed in stochastic gradient descent. However, in our case, due to less data and less parameters, we want to update the weights for each pixel to make it learn better. Thus, we tried to implement the stochastic gradient descent on our existing model.

Results: The results obtained using the CNN are better than the vanilla neural networks. Training the neural net on a given class of images (greenery), it generalizes good enough to add a tinge of green on other tree images. Similarly, training it on images with water, it starts adding blue to the image. However, when we train it on generalized data, the result generated is pretty much gray. The image is still well segmented, but no such colors are added to it. These results can be observed in Figure 6.

(a) Testing Input

(b) Testing output

(c) Testing Input

(d) Testing output

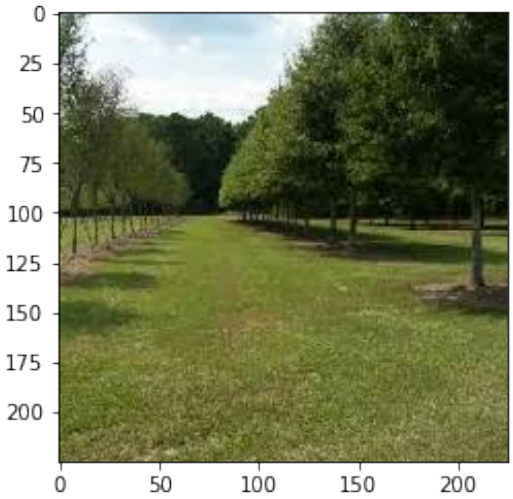
(e) Testing Input

(f) Testing output

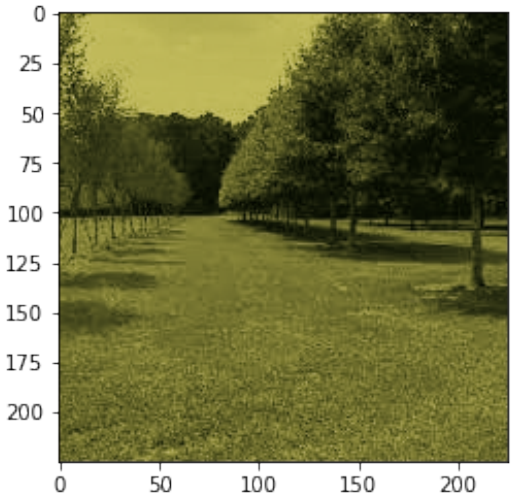
Figure 2: Result of CNN: Training the model on similar images of Trees and water bodies, the model is able to capture some properties of a new image of the same class.

(a) Ground Truth (b) Model prediction

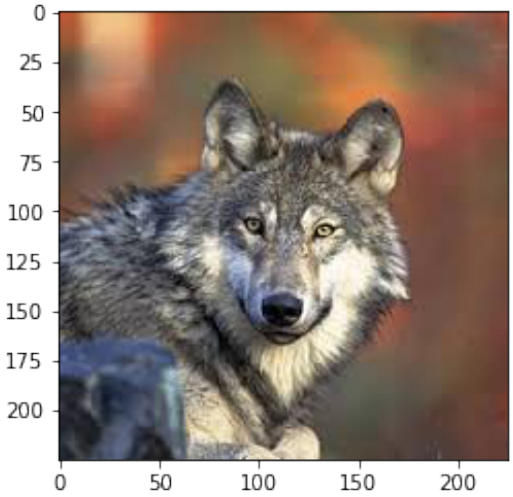
Figure 3: Result of training red, green and blue channels separately:



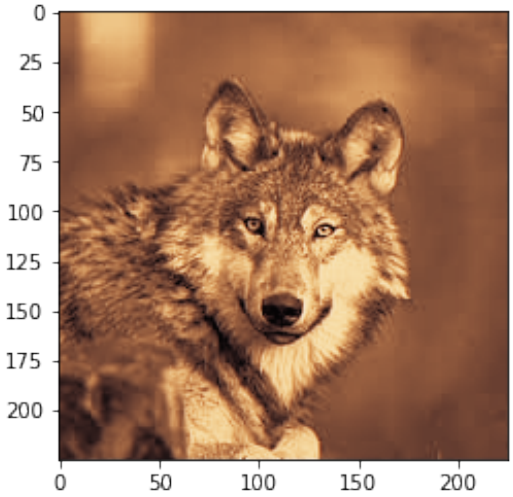
(a) Testing Input



(b) Testing output



(c) Testing Input



(d) Testing Input

Figure 4: Result of CNN using Stochastic Gradient Descent

3.3 Classification model

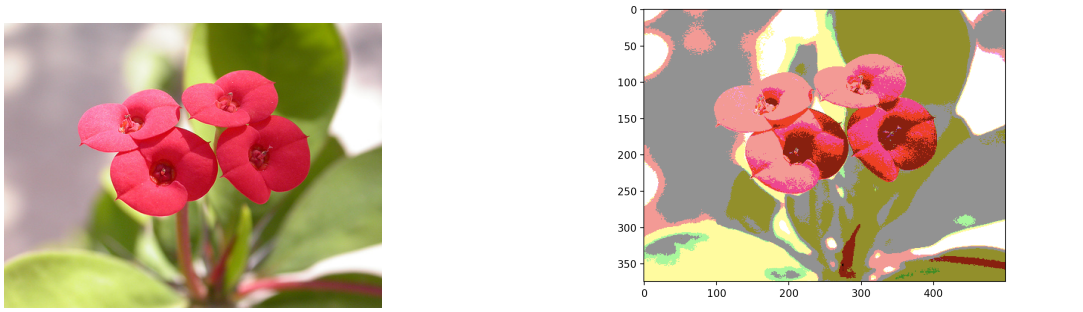
The final approach we try is to shift from a regression problem to a classification problem. We built a palette of K = 27 representative colors and map each (r,g,b) value to a corresponding color in the palette using Manhattan distance. Hence, each pixel in the image belongs to one of the 27 classes. In this way, the problem is converted into a classification problem. The output will be one hot encoding of the classes. The number of nodes in the output layer will be equal to K instead of earlier 3 nodes. The input nodes remain the same. Soft-max activation is used in the last layer, for each node in the output layer to correspond to a probability of belonging to that class, since the soft-max returns a vector where the values of all the elements sum to 1. The soft-max activation is given by

$$y(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{3}$$

For the softmax function, a better cost function than the mean squared error cost function is the cross-entropy function, given by

$$H(y, \hat{y}) = -\sum_i y_i log(\hat{y}_i) \tag{4}$$

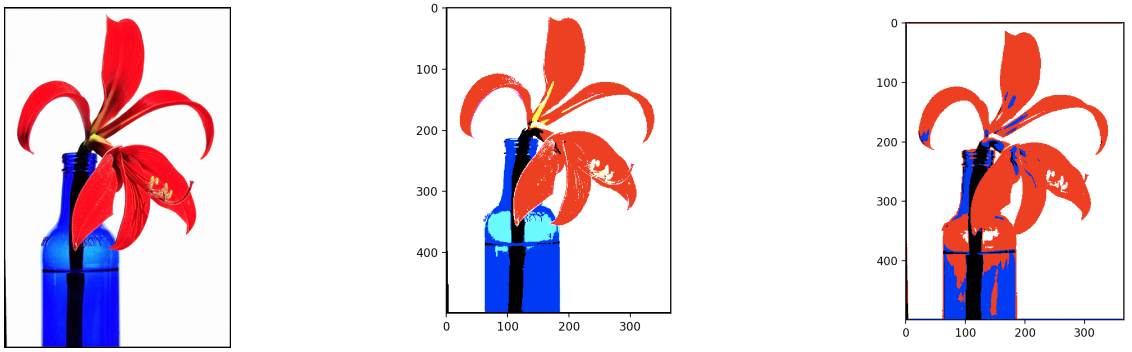
where y is the actual output vector and $\hat{y}$ is the predicted output vector. Figure 5 shows the mapping of a sample colored image to the palette of 27 colors. We got perceptually good results for images with fewer number of colors and those with fewer patches of color. One such example can be seen in Figure **??**. When an image with many color patches is given as an input, a cat in Figure 4, the model is not able to learn it even when run for more number of epochs. Next, we trained the model on similar and simpler images and test it on an image of the same class. The result is shown in Figure 8. The model is able to capture properties of a new image well. But when a class of more complicated images is given, it is again not able to generalize well.



(a) Original image

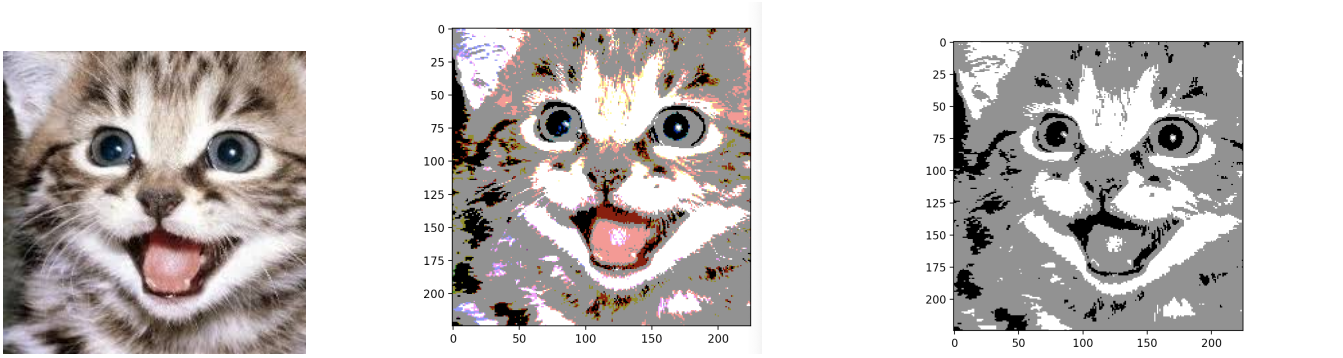
(b) Mapping of original image to 27 color classes

Figure 5: Classification Neural Networks: Mapping of (r,g,b) colors to 27 representative colors.



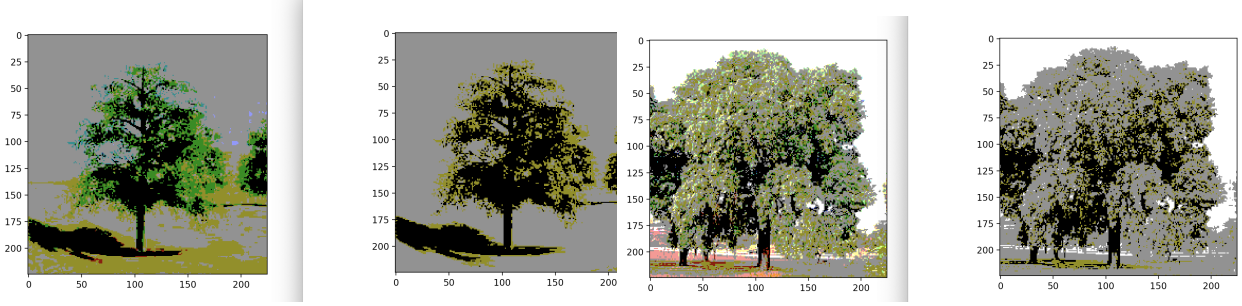
(a) Original image (b) Mapping to 27 color palette (c) Model output with (b) as the groundtruth

Figure 6: Result of Classification Neural Networks: When the model is trained on the same image, it produces the output (c) after 200 epochs.



(a) Original image (b) Mapping to 27 color palette (c) Model output with (b) as the groundtruth

Figure 7: Result of Classification Neural Networks: When the model is trained on the same image, it produces the output (c) after 200 epochs.



(a) Training Output (b) Testing output

Figure 8: Result of Classification Neural Networks: Training the model on 5 similar images of Trees, the model is able to capture some properties of a new image of a tree.

4 Assessing the Final Project

How good is your final program, and how can you determine that? How did you validate it? What is your program good at, and what could use improvement? Do your program's mistakes 'make sense'? What happens if you try to color images unlike anything the program was trained on?
The final model performs fairly well when similar data-set is given for training and testing. However, it is unable to generalize on new data-set or some different image of some different class. We validate the data by generating a colored image for a new image of similar class such as tree etc (image which is not included in the training or testing). The program's mistake of capturing the major color and outputting it throughout makes somewhat sense as the model tries to minimize the overall loss and thus in some sense takes the median value and propagates this majorly.
What kind of models and approaches, potential improvements and fixes, might you consider if you had more time and resources?
Given more time and resources, we would have tried to implement the advanced neural networks such as encoder-decoder models. Also, we ardently wanted to work the model of stacking up convolutional neural network layers and tune it to perform better on generalizing new data-set. We also wished to refine the model proposed for reconstructing damaged images and implement it to validate it. In the classification model, we would have implemented the colour buckets depending on the more commonly seen colors or some other color bucket.

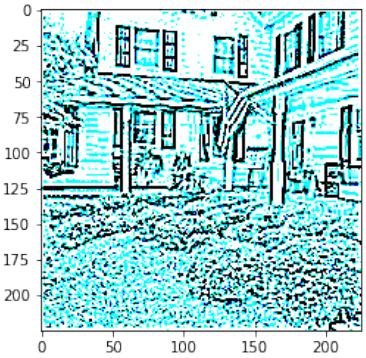
5 Bonus

Imagine trying to reconstruct damaged images after an event like a fire. How could you build a system to 'in-paint' or fill in holes/missing areas in images? Consider the five areas outlined above in designing such a thing. Build it.

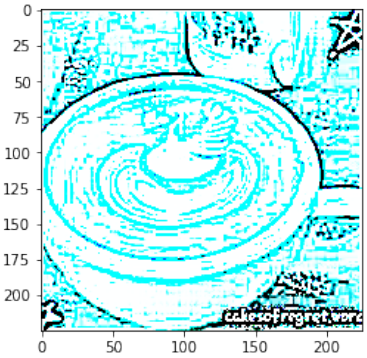
After discussing this bonus problem, we concluded that reconstructing the coloured image on the output end makes more sense and seems easier to extend in the existing implementations. As the first two approaches are fundamentally regression models used to predict the RGB values, these models can be used to reconstruct the damaged images. Also, the neural network work as a highly parameterized function approximators. Thus, we understand this by visualising the problem as an extrapolation of the undamaged neighborhood of damaged pixel. Since neighborhood information and context plays a major role in accurately predicting the parts of the damaged parts, the model similar to CNN will perform the best out of the above models. The training and testing will be done on undamaged images (for better results - similar undamaged images). The output image shape is the same size as that of the input image shape.

6 Fascinating Results

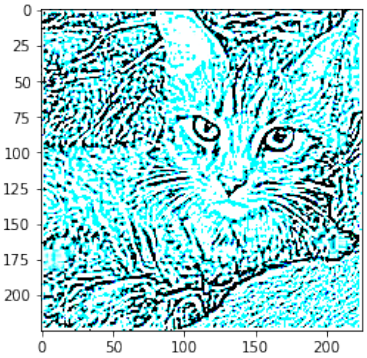
While exploring the implementation variants, we came across some fascinating results. These came out as surprises, so we do not have proper justification for them. Unfortunately, the final program does not provide these outputs.



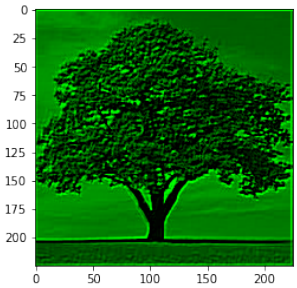
(a) Cyan Edges



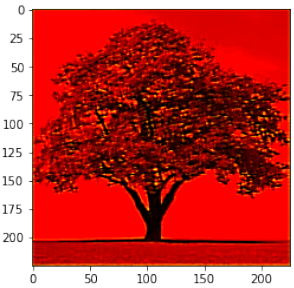
(b) Text boundaries



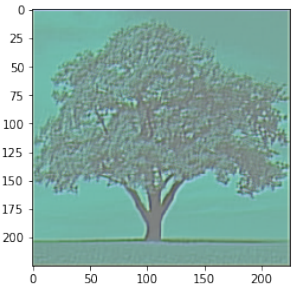
(c) Fancy Cat



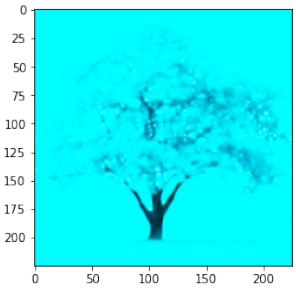
(a) Leaky RelU



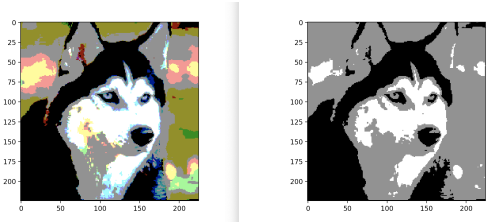
(b) ReLu



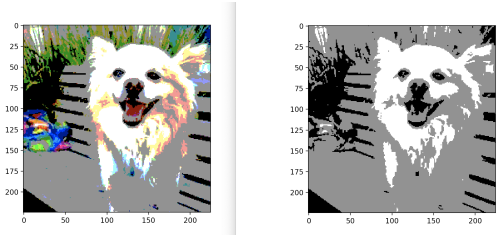
(c) Sigmoid



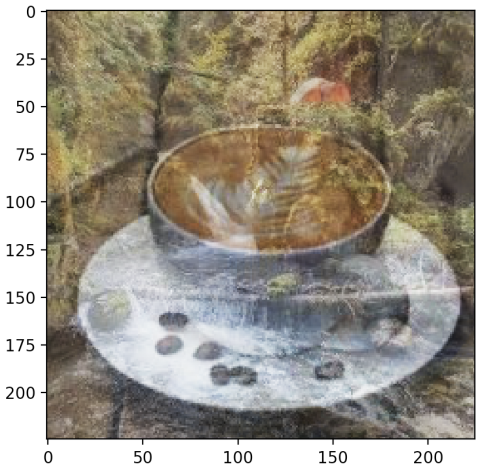
(d) Tanh



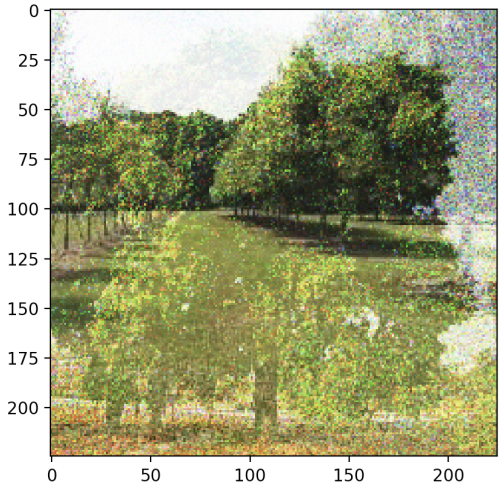
(a) Training Output



(b) Testing output



(a) Overlapping Images



(b) Overlapping Images