ECE457B - Assignment 2

Sheen Thusoo | 20728766

Question 1

The following libraries were used for this question: pandas, numpy, sklearn.

a.

The Wine dataset was first loaded using the *pandas* library. Then the "quality" column was extracted into a list named *labels* as these are the target values. The feature columns (excluding quality) were then normalized using sklearn's *MinMaxScaler*. These data pre-processing steps are shown in Figure 1. Lastly, the data was split using sklearn's *train_test_split* so that 80% of the data was used for training and 20% of the data was used for testing.

```
import pandas as pd
import numpy as np

# Load Wine Dataset
df = pd.read_csv('/content/Wine_Dataset.csv')

# Labels -> Quality attribute
Labels = np.array(df['quality'].to_list())

# Features df
Features = df.copy()
Features.drop('quality', inplace=True, axis=1)
Features
```

```
[2]  # Normalize data using MinMaxScaler
     from sklearn.preprocessing import MinMaxScaler
     scaler = MinMaxScaler().fit(Features)
     FeaturesStand = scaler.transform(Features)
     FeaturesStand = pd.DataFrame(FeaturesStand)
     FeaturesStand
```

Figure 1: Data Preprocessing Code for Question 1

Next, SVM was used via sklearn's library to classify wine quality. The models were trained using the training data and their accuracies were found using the test data. The RBF and Poly kernels were used with the following C (regularization parameter) values: 1, 10, 50, 100. The testing accuracies of the varying parameters in the SVM model are shown in Table 1. A linear kernel SVM was also used with the following C values: 1, 10, 20, 30. The resulting test accuracies of the varying parameters with a linear kernel are shown in Table 2.

Table 1: Testing Accuracies (%) for RBF and Poly Kernels

		C Value			
		1	10	50	100
Kernel Type	Poly	55.6923%	56.6923%	56.9231%	56.6923%
	RBF	55.0769%	57.2308%	57.9999%	58.0769%

Table 2: Testing Accuracies (%) for Linear Kernel

		C Value			
		1	10	20	30
Kernel Type	Linear	54.4615%	54.6154%	54.6923%	54.6923%

b.

From the results found in part a), it can be seen that the SVM models with varying regularization parameter values had similar test accuracies, around 54-58%. The SVM model with a linear kernel and a C value of 1 had lowest test accuracy of 54.46%. The SVM model with an RBF kernel and a C value of 100 had the highest test accuracy of 58.08% and thus had the best performance. The linear kernel has a poor performance on the data since it is not linear, the RBF kernel has a much better performance as this kernel is suited to non-linear data such as the Wine dataset. The Poly kernel performed better than the linear kernel but worse than the RBF kernel since it was unable to accurately separate the highly non-linear data. The RBF kernel is able to adapt to the size and complexity of the data.

c.

The best kernel found was the RBF kernel. Thus, its C value was varied with the following values: [75, 100, 150, 200] and its gamma values were varied with the following values: [0.01, 1, 10, 100]; thus, creating a total of 16 models with an RBF kernel. The testing accuracies of each of these models are shown in Table 3.

Table 3: Testing Accuracies (%) for RBF Kernel with varying values of C and Gamma

		C Value			
		75	**100**	**150**	**200**
Gamma Value	**0.01**	54.5385%	54.5385%	54.4615%	54.4615%
	1	57.3077%	57.5385%	57.6154%	57.5385%
	10	59.3077%	59.0769%	59.6923%	60.0%
	100	65.2308%	65.4615%	65.4615%	65.4615%

From Table 3, it was found that the gamma value had more of an influence on the test accuracy than the C value. For instance, when gamma was set to 0.01, the test accuracies remain near 54.5% for all C values and when gamma was set to 1, the test actuaries remain near 57.5% for all C values observed. The best models (three tied) with the highest test accuracy of 65.46% had a gamma value of 100 and C values of 100, 150 and 200. The C parameter determines the tradeoff between correct classification and maximization of the margin. The gamma value has a higher influence on the accuracy since this value defines the curvature of the decision boundary. When gamma is low, the curve of the decision boundary is low and its decision region is more broad; thus, the model cannot capture the complexity (non-linearity) of the data. When gamma is high, the curve of the decision boundary is high and it can better capture this complexity. Since our data is non-linear, a higher gamma value was able to improve accuracy. However, it should be noted that if gamma is too large, overfitting can occur.

Question 2

The following libraries were used for this question: matplotlib and numpy.

a.

A Kohonen self-organizing map was created with a training input of 24 colours. The code from the kSOM tutorial, *ksom.ipynb,* was updated by adding a new neighbourhood function as well as a new learning rate update formula. The 24 input colours were normalized using Min-Max normalization; in this case, we know the minimum and maximum values (0 and 255 respectively) for each colour, thus we can divide each colour by 255. Next, the weights were randomly initialized using *np.random.uniform* between the values of 0 and 1 with a size of 100 by 100 (grid of neurons). These random weights can be seen in Figure 2.

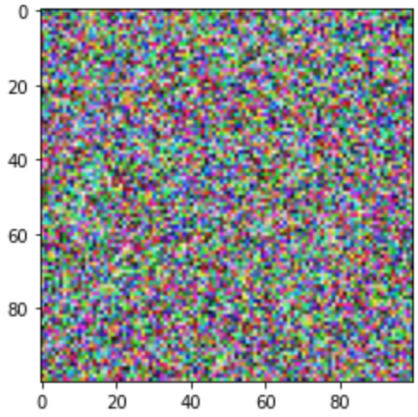


Figure 2: Randomly Initialized Weights

Next, the variables needed for the kSOM algorithm were initialized according to the values given in the question. The initial alpha value was set at 0.8, the initial sigma value was set to 10 (as mentioned in Piazza note @166), and the maximum number of epochs was set to 1000. Then, a while loop was created in order to update the grid at each epoch. Inside the while loop, the difference between each neuron weight and the colour was calculated. The index of the neuron with the minimum difference is then found; this is the best neuron. Next, the distance between this best neuron and every other neuron is calculated. These distance values are then used to update the neighborhood function using the following equation as stated in the question:

$$N_{i,j}(k) = exp\left(-\frac{d_{i,j}^2}{2\sigma^2(k)}\right), \text{ where } \sigma(k) = \sigma_0 exp\left(-\frac{k}{T}\right)$$

The weights are then updated using the learning rate, α, the result of the neighborhood function, and the difference between the colour and the weight. The code for this question is shown in Figure 3.

```
from IPython import display
# Constants
alpha_0 = 0.8
alpha = alpha_0
sigma_0 = 10
sigma = sigma_0
epoch = 1
max_epochs = 1000
rows, cols = len(w), len(w[0])
while epoch <= max_epochs:
  for x in normRGB:
    # calculate performance index
    diff = np.linalg.norm( np.subtract(x,w) , axis=2)
    # find index of winning node
    ind = np.array( np.unravel_index(np.argmin(diff, axis=None), diff.shape) )

    for i in range(rows):
      for j in range(cols):
        if i >= 0 and j >= 0 and i < space_size and j < space_size:
          # Calculate distance of winning node from other neurons
          neuron = np.asarray([i, j])
          d = np.linalg.norm( ind - neuron )

          N = np.exp(- np.square(d) / (2 * np.square(sigma) ) )
          # Update weights for neighbourhood
          w[i][j] += alpha * N *( x - w[i][j] )

    # Update learning rate and sigma
    sigma = sigma_0 * np.exp(- epoch / max_epochs )
    alpha = alpha_0 * np.exp(- epoch / max_epochs )

  plot_ind = [20, 40, 100, 1000]
  if epoch in plot_ind:
    print("Epoch Number: {}".format(epoch))
    plt.imshow(w)
    display.display(plt.gcf())

  epoch += 1
```

Figure 3: kSOM Algorithm Code

The results of the kSOM grid after 20, 40, 100, and 1000 epochs are shown in Figures 4-7.

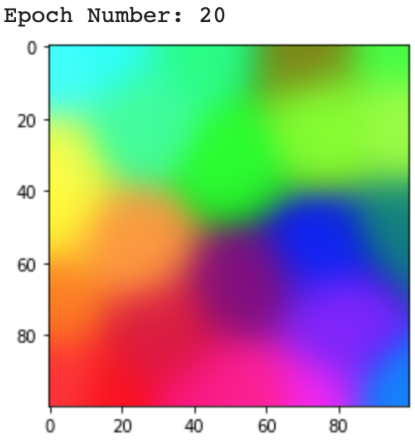


Figure 4: Grid After 20 Epochs

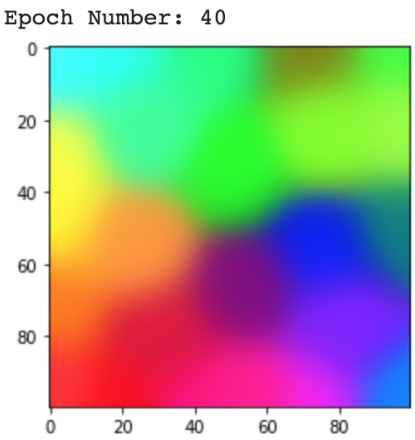


Figure 5: Grid After 40 Epochs

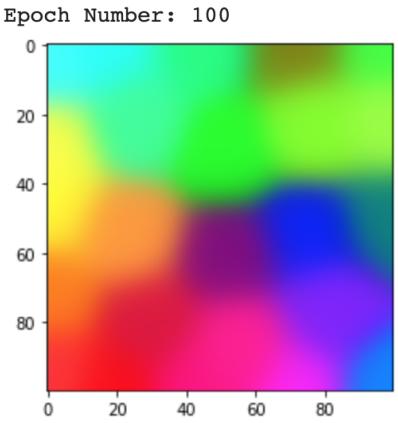


Figure 6: Grid After 100 Epochs

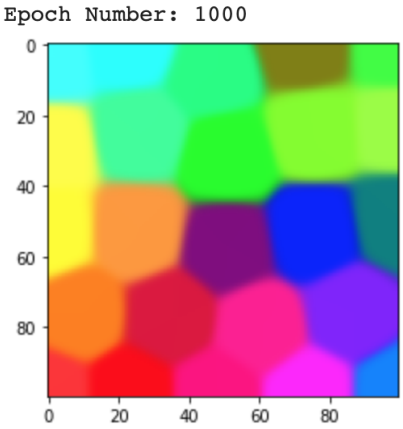


Figure 7: Grid After 1000 Epochs

b.

Based on the results observed in part a), it is seen that as the algorithm progresses at each epoch, the clusters spread to cover the entire grid and the SOM converges quickly. It is noticeable that colours that are similar are located near each other. For instance, the green shades are near the top right of the grid. At 1000 epochs, the clusters have well-defined edges. The fast convergence of the algorithm can be attributed to the neighbourhood function and the initial sigma value that was set to 10. Figure 8 shows the neighbourhood function plotted when $k = 0$ (first epoch); it is clear that the function approaches 0 as the number of neurons (x-axis) approaches 8. This indicates that only neurons within 8 units of the winning neuron would be updated. This neighbourhood size is large enough to result in convergence over the entire grid.

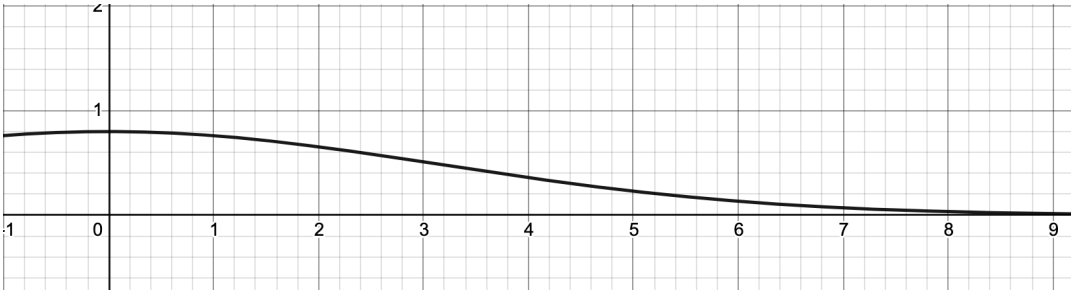


Figure 8: Neighbourhood Function with $\sigma_0 = 10$

The kSOM algorithm was also run with $\sigma_0 = 1$ for comparison. From Figures 8 to 11 it is clear that the SOM does not converge over the entire grid; instead, there are localized clusters of colours that are created. These grids look very different from those in figures 4 to 7 due to the change in the neighbourhood function. In this case, the initial sigma value has been reduced to 1; this causes the sigma update function to change which affects our neighbourhood function. This can be seen in Figure 12; here, the new neighbourhood function drops off to 0 when the number of neurons (x-axis) is around 3. This means that the weights of the neurons within only 3 units of the winning node will be updated. This explains why the clusters are much smaller (smaller neighbourhood size) and the SOM does not converge over the entire grid even after 1000 epochs.

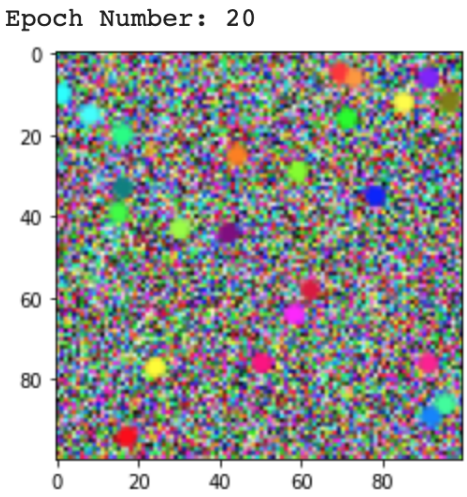


Figure 8: Grid After 20 Epochs

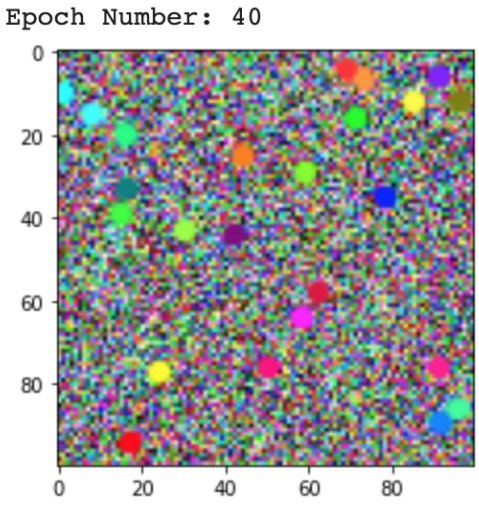


Figure 9: Grid After 40 Epochs

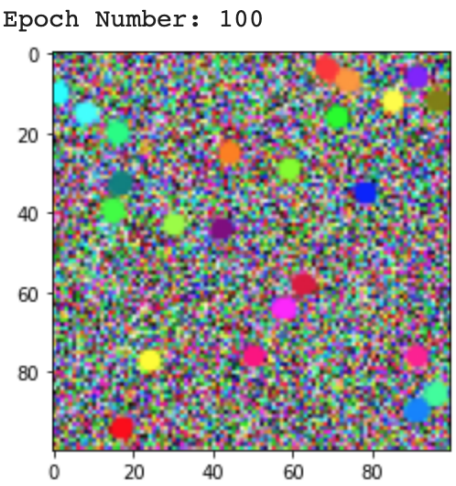


Figure 10: Grid After 100 Epochs

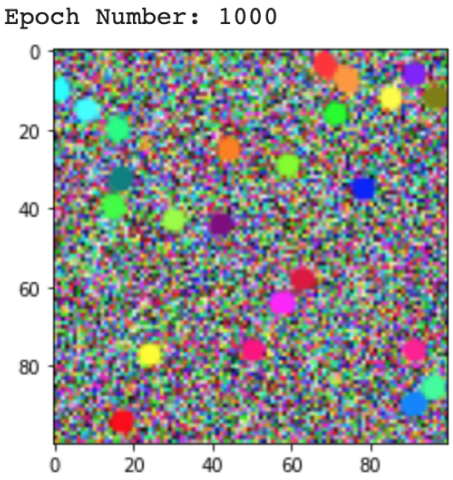


Figure 11: Grid After 1000 Epochs

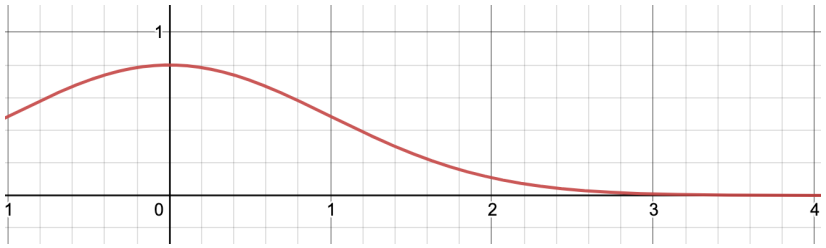


Figure 12:Neighbourhood Function with $\sigma_0 = 1$

Question 3

The following libraries were used for this question: numpy, tensorflow, and keras.

a.

Using tensorflow and keras, three models were created and trained to classify images from the CIFAR10 dataset. First the training and test sets were loaded using the API - *datasets.cifar10.load_data()*. Then a sample size of 20% of the training data was used to generate random indices. These indices were used to get training images along with their corresponding training labels. Using this, 20% of the original training data was sampled. Then, min-max normalization was performed on both the training and test images. The training and test labels were converted to binary class matrices (one-hot encoding) using the *to_cateogrical()* function from the keras library. Finally, the three models were defined as specified in the question. The first model was an MLP with Dense layers and sigmoid activation. The second model was a CNN with two Convolutional layers with ReLu activation, a flatten layer, and Dense layers with sigmoid activation. Lastly, the third model was also a CNN but with Convolutional layers with ReLu activation along with Max Pooling layers, Dense layers, and Dropout Regularization. The code showing each model definition is shown in Figure 13.

```
## Model 1 ##
def get_model1():
  model1 = models.Sequential()
  model1.add(Flatten(input_shape=(32, 32, 3)))
  model1.add(layers.Dense(512, activation='sigmoid', input_shape=(3072,)))
  model1.add(layers.Dense(512, activation='sigmoid'))
  model1.add(layers.Dense(10, activation='softmax'))

  model1.compile(optimizer='adam',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])
  return model1

## Model 2 ##
def get_model2():
  model2 = models.Sequential()
  model2.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)))
  model2.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model2.add(layers.Flatten())
  model2.add(layers.Dense(512, activation='sigmoid'))
  model2.add(layers.Dense(512, activation='sigmoid'))
  model2.add(layers.Dense(10, activation='softmax'))

  model2.compile(optimizer='adam',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])
  return model2

## Model 3 ##
def get_model3():
  model3 = models.Sequential()
  model3.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)))
  model3.add(layers.MaxPooling2D((2, 2)))
  model3.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model3.add(layers.MaxPooling2D((2, 2)))
  model3.add(layers.Flatten())
  model3.add(layers.Dense(512, activation='sigmoid'))
  model3.add(Dropout(0.2))
  model3.add(layers.Dense(512, activation='sigmoid'))
  model3.add(Dropout(0.2))
  model3.add(layers.Dense(10, activation='softmax'))

  model3.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
  return model3
```

Figure 13: Model 1, 2, and 3 definitions

These three models were trained on the sampled training data with a batch size of 32 and for 5 epochs. The training and testing accuracies of all three models are shown in Table 4.

Table 4: Training and Test Accuracies (%) of all Models

	Training Accuracy	**Test Accuracy**
Model 1	36.950%	36.540%
Model 2	89.890%	51.370%
Model 3	58.550%	54.040%

The MLP model, Model 1, had the worst performance with a training accuracy of 36.95% and a test accuracy of 36.54%. These numbers indicate that the model was underfitting and is unable to learn the features of the data. The first CNN model, Model 2, has a high training accuracy of 89.89% and a much lower test accuracy of 51.37%. The second CNN model, Model 3, which is more complex than the second CNN model, has a training accuracy of 58.55% and a test accuracy of 54.04%.

From these results, it is clear that the CNN models perform much better on the data than the MLP model with a test accuracy increase between 15-18% as seen in Table 4. These results make sense since CNNs are known to perform well on image data whereas MLPs are not. MLPs consist of fully connected Dense layers in which each node in each layer is connected. For large amounts of data, such as images, the amount of parameters becomes unmanageable and inefficient. As a result, the MLP is unable to generalize and learn the features of the data. CNNs perform better on image data since they preserve spatial relations which helps to classify images as adjacent pixels are usually more strongly related than farther ones. This is due to the Convolutional layer which consists of a filter that is convoluted over the image; this generates a feature map of the image. CNNs reduce images into a form which is easier to process while maintaining features that are critical for good predictions.

b.

The training and validation curves for the first CNN model, Model 2, and the second CNN model, Model 3 are shown in Figure 14 and 15 below. Figure 14 shows training/validation losses whereas Figure 15 shows training/validation accuracies.

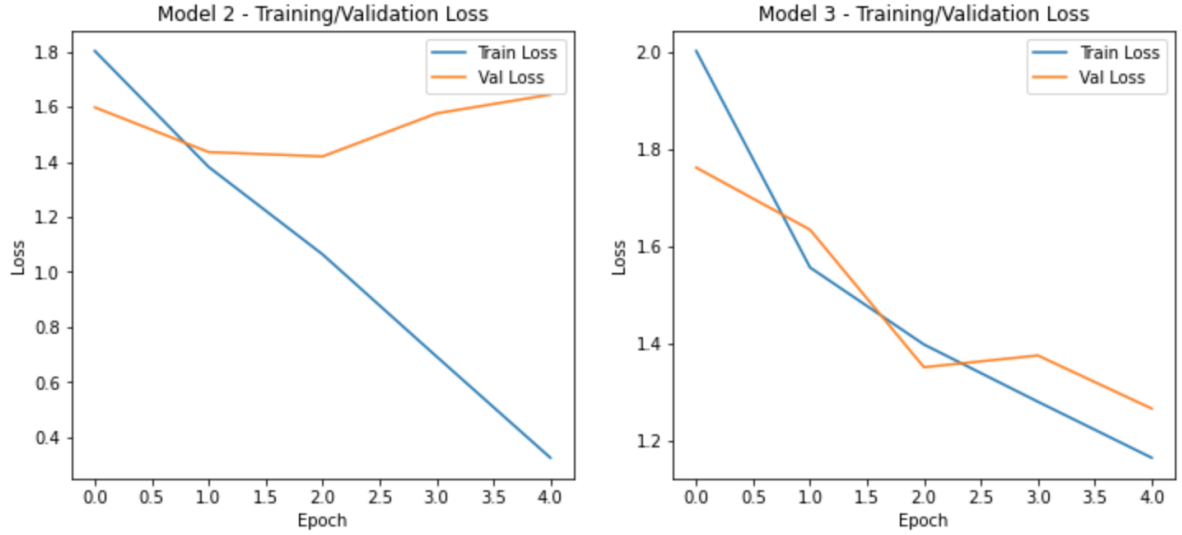


Figure 14: Training and Validation Losses for Model 2 (left) and Model 3 (right)

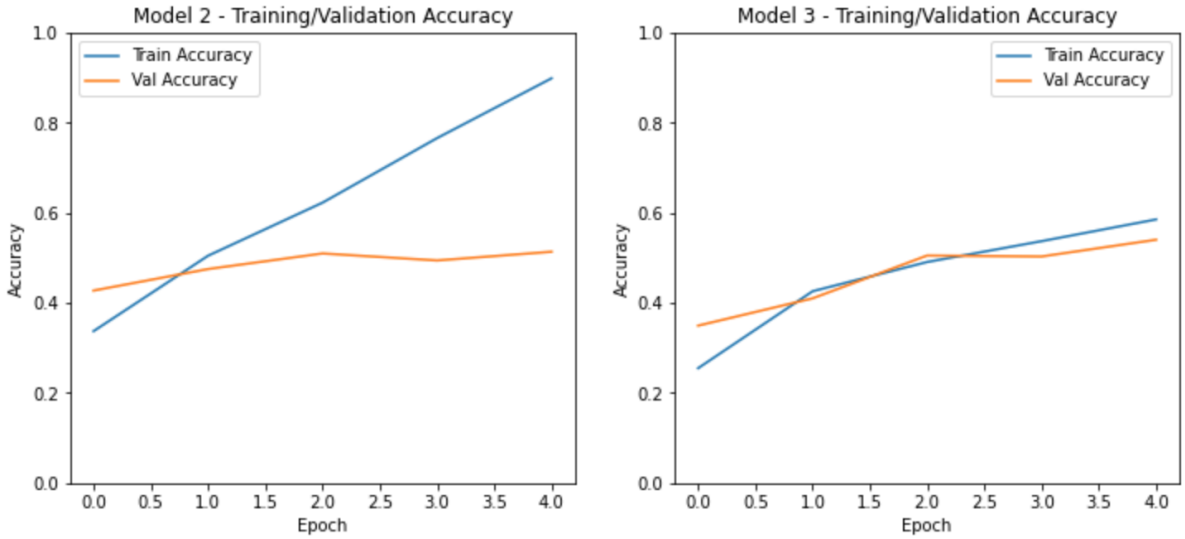


Figure 15: Training and Validation Accuracies for Model 2 (left) and Model 3 (right)

From Figure 14, it is clear that the training loss for Model 2 drastically decreases however the validation loss decreases slightly and then increases again over 5 epochs. In contrast, Model 3's training and validation loss both decrease over 5 epochs. From Figure 15, we see a similar pattern where Model 2's training accuracy increases drastically over 5 epochs however its validation accuracy increases slightly and then decreases but remains at a similar value. Model 3's training and validation accuracies both increase similarly throughout the epochs.

Thus, it is clear that Model 2 is overfitting; its training accuracy is much higher than its validation accuracy and its training loss is much lower than its validation loss. This indicates that the model is memorizing the training data and the noise associated with it; due to this, it is unable to generalize and performs poorly on the validation data. On the other hand, Model 3 does not overfit as the training and validation accuracies and losses are similar. However, Model 3 is underfitting as both the training and validation accuracies are low, with values of around 55-59% at epoch 5. This indicates that the model is not able to capture the features in the data accurately.

```
Model 2 Training Time: 709.303 seconds
Model 3 Training Time: 202.609 seconds
```

Figure 16: Training Times for Model 2 and 3

From Figure 16, the training time for Model 2 is much longer than Model 3. Model 2 takes about 700 seconds to train whereas Model 3 takes around 200 seconds to train. Thus, Model 2 takes about 8 minutes longer to train than Model 3.

The architecture of these models influence the resulting training times shown in Figure 16. Model 2 consists of two 2D Convolutional layers whereas Model 3 consists of a 2D Convolutional layer followed by a 2x2 MaxPooling Layer (this is repeated twice). MaxPooling is performed to prevent overfitting but also to reduce computational cost by

decreasing the number of parameters to learn; it reduces the dimensions of the data. Thus, the two MaxPooling layers significantly reduce the number of parameters to learn, the dimensions of the data and reduce computational time.

Another architecture difference is that Model 3 has two Dropout layers with a rate of 0.2 whereas Model 2 does not, it only has Dense fully-connected layers. A Dropout layer randomly sets nodes to 0 with a frequency equivalent to the dropout rate (in this case 0.2). This helps prevent overfitting by forcing the network to explore more ways of learning patterns rather than depending too much on some features. Due to the addition of this Dropout regularization, Model 3 has a better ability to generalize than Model 2. This is seen as Model 2 overfits (validation loss is much higher than training loss) whereas Model 3 does not (training and validation losses are similar). Model 3 also has a slightly higher test accuracy than Model 2.

If the networks were trained for more epochs, Model 2 would most likely continue overfitting to a point where the training accuracy would be near 100% whereas the validation accuracy would be much lower. Within 5 epochs, the model already overfits as it has a high training accuracy but a much lower validation accuracy. Model 2 memorizes the training data along with the noise associated with it and thus, is unable to generalize to new unseen data (validation set). If trained for more epochs, Model 3 would likely perform better in terms of training and validation accuracy. With 5 epochs, Model 5 is currently underfitting as both its training and validation accuracy are low. The model architecture seems complex enough, due to the MaxPooling and Dropout layers, for the data and is unlikely to overfit. It is possible that the model underfits because it is not being trained for long enough. Training the model for more epochs would likely increase its performance. However, the training/validation accuracies should be monitored as training for too many epochs may lead to overfitting.

ECE 457B - Assignment 2

By: Sheen Thusoo, 20728766

Problem 1: Support Vector Machines

We need to construct SVM classifier to classify data from Wine Dataset, our main target is to try to classify Wine's Quality based on the inputs. Please use Wine_Dataset.csv file uploaded to learn for this task. (The dataset is originally from: https://archive.ics.uci.edu/ml/datasets/wine+quality (https://archive.ics.uci.edu/ml/datasets/wine+quality))

a. Load Wine dataset and discover its parameters, you may need to normalize data if it improves your results. Then use SVM in Sklearn library to classify Wine quality, vary the hyperparameters as follows: Use Kernels of RBF, Linear, and Poly. Use the regularization parameter "C" as [1,10,50,100] for RBF and Poly, and [1,10,20,30] for Linear Kernel. Add results to a table.

In [ ]:
```
import pandas as pd
import numpy as np

# Load Wine Dataset
df = pd.read_csv('/content/Wine_Dataset.csv')

# Labels -> Quality attribute
Labels = np.array(df['quality'].to_list())

# Features df
Features = df.copy()
Features.drop('quality', inplace=True, axis=1)
Features
```

Out[ ]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcol
0	7.0	0.270	0.36	20.7	0.045	45.0	170.0	1.00100	3.00	0.45	
1	6.3	0.300	0.34	1.6	0.049	14.0	132.0	0.99400	3.30	0.49	
2	8.1	0.280	0.40	6.9	0.050	30.0	97.0	0.99510	3.26	0.44	1
3	7.2	0.230	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	
4	7.2	0.230	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	
...	...	...	...	...	...	...	...	...	...	...	
6492	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	1
6493	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	1
6494	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	1
6495	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	1
6496	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	1

6497 rows × 12 columns

In [ ]:
```
# Normalize data using MinMaxScaler
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler().fit(Features)
FeaturesStand = scaler.transform(Features)
FeaturesStand = pd.DataFrame(FeaturesStand)
FeaturesStand
```

Out[ ]:

	0	1	2	3	4	5	6	7	8
0	0.264463	0.126667	0.216867	0.308282	0.059801	0.152778	0.377880	0.267785	0.217054
1	0.206612	0.146667	0.204819	0.015337	0.066445	0.045139	0.290323	0.132832	0.449612
2	0.355372	0.133333	0.240964	0.096626	0.068106	0.100694	0.209677	0.154039	0.418605
3	0.280992	0.100000	0.192771	0.121166	0.081395	0.159722	0.414747	0.163678	0.364341
4	0.280992	0.100000	0.192771	0.121166	0.081395	0.159722	0.414747	0.163678	0.364341
...	...	...	...	...	...	...	...	...	...
6492	0.198347	0.346667	0.048193	0.021472	0.134551	0.107639	0.087558	0.150183	0.565891
6493	0.173554	0.313333	0.060241	0.024540	0.088040	0.131944	0.103687	0.154425	0.620155
6494	0.206612	0.286667	0.078313	0.026074	0.111296	0.097222	0.078341	0.166377	0.542636
6495	0.173554	0.376667	0.072289	0.021472	0.109635	0.107639	0.087558	0.161172	0.658915
6496	0.181818	0.153333	0.283133	0.046012	0.096346	0.059028	0.082949	0.161558	0.519380

6497 rows × 12 columns

In [ ]:
```
# Train / Test Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(FeaturesStand, Label
s, test_size=0.20)

print('Train Features Shape: {}'.format(X_train.shape))
print('Train Labels Shape: {}'.format(y_train.shape))
print('Test Features Shape: {}'.format(X_test.shape))
print('Test Labels Shape: {}'.format(y_test.shape))
```

```
Train Features Shape: (5197, 12)
Train Labels Shape: (5197,)
Test Features Shape: (1300, 12)
Test Labels Shape: (1300,)
```

In [ ]:
```
# Use SVM
from sklearn import svm

# RBF Kernel
C = [1,10, 50, 100]
print('*************RBF Kernel*************')
for val in C:
  svm_ = svm.SVC(C=val, kernel='rbf')
  svm_.fit(X_train, y_train)
  print('C={} ------- Accuracy: {}%'.format(val, svm_.score(X_test,y_tes
t)*100) )

print('\n')
# Poly Kernel
print('*************Poly Kernel*************')
for val in C:
  svm_ = svm.SVC(C=val, kernel='poly')
  svm_.fit(X_train, y_train)
  print('C={} ------- Accuracy: {}%'.format(val, svm_.score(X_test,y_tes
t)*100) )

print('\n')
# Linear Kernel
C1 = [1, 10, 20, 30]
print('*************Linear Kernel*************')
for val in C1:
  svm_ = svm.SVC(C=val, kernel='linear')
  svm_.fit(X_train, y_train)
  print('C={} ------- Accuracy: {}%'.format(val, svm_.score(X_test,y_tes
t)*100) )
```

```
*************RBF Kernel*************
C=1 ------- Accuracy: 55.07692307692308%
C=10 ------- Accuracy: 57.230769230769226%
C=50 ------- Accuracy: 57.99999999999999%
C=100 ------- Accuracy: 58.07692307692308%


*************Poly Kernel*************
C=1 ------- Accuracy: 55.69230769230769%
C=10 ------- Accuracy: 56.6923076923077%
C=50 ------- Accuracy: 56.92307692307692%
C=100 ------- Accuracy: 56.6923076923077%


*************Linear Kernel*************
C=1 ------- Accuracy: 54.46153846153846%
C=10 ------- Accuracy: 54.61538461538461%
C=20 ------- Accuracy: 54.69230769230769%
C=30 ------- Accuracy: 54.69230769230769%
```

b. Explain your findings of the results from Part a.

c. Modify your best kernel model to find a better performing model by changing hyperparameters "C and Gamma" by trying at least 10 different combinations and record results in a table. Comment briefly on your findings. DON'T USE LIBARIES TO AUTOMATE THIS PROCESS!!

In [ ]:
```
# Best Kernel Model - RBF
C = [75, 100, 150, 200]
gamma = [0.01, 1, 10, 100]
kernel = 'rbf'
for val in C:
  for g in gamma:
    svm_ = svm.SVC(gamma=g, C=val, kernel=kernel)
    svm_.fit(X_train, y_train)
    print('C={}, Gamma={} ------- Accuracy: {}%'.format(val, g, svm_.sco
re(X_test,y_test)*100) )

  print('\n')
```

```
C=75, Gamma=0.01 ------- Accuracy: 54.53846153846153%
C=75, Gamma=1 ------- Accuracy: 57.30769230769231%
C=75, Gamma=10 ------- Accuracy: 59.30769230769231%
C=75, Gamma=100 ------- Accuracy: 65.23076923076923%


C=100, Gamma=0.01 ------- Accuracy: 54.53846153846153%
C=100, Gamma=1 ------- Accuracy: 57.53846153846154%
C=100, Gamma=10 ------- Accuracy: 59.07692307692308%
C=100, Gamma=100 ------- Accuracy: 65.46153846153845%


C=150, Gamma=0.01 ------- Accuracy: 54.46153846153846%
C=150, Gamma=1 ------- Accuracy: 57.61538461538461%
C=150, Gamma=10 ------- Accuracy: 59.692307692307686%
C=150, Gamma=100 ------- Accuracy: 65.46153846153845%


C=200, Gamma=0.01 ------- Accuracy: 54.46153846153846%
C=200, Gamma=1 ------- Accuracy: 57.53846153846154%
C=200, Gamma=10 ------- Accuracy: 60.0%
C=200, Gamma=100 ------- Accuracy: 65.46153846153845%
```

Problem 2 (Kohonen Self Organizng Map: Unsupervised Learning)

We need to design a Kohonen self organizing map (SOM), which gives as an output some shades of color mapped over 100 by 100 grid of neurones. The training input of the SOM are 24 colors (use shades of red, green, blue, with some yellow, teal and pink) which you can chose from the "RGB Color Table: Basic Colors" section of this page: http://www.rapidtables.com/web/color/RGB_Color.htm (http://www.rapidtables.com/web/color/RGB_Color.htm)

Using a time varying learning rate $\alpha(k) = \alpha(0)\exp\left(-\frac{k}{T}\right)$ where k is the current training epoch (starts with epoch 0), $\alpha(0) = 0.8$, and T is the total number of training epochs equal to 1000.

Note that the epoch training involves all twenty four input samples for the 24 chosen colors to the network (hint: calibrate the color codes to values between 0 and 1, instead of being between 0 and 255). Initial weights are randomized.

The topological neighbourhood $N_{i,j}(k)$ of node (j) around the winning unit (i) is given by

$$N_{i,j}(k) = \exp\left(-\frac{d_{i,j}^2}{2\sigma^2(k)}\right)$$

where

$$\sigma(k) = \sigma_0 \exp\left(-\frac{k}{T}\right)$$

and $d_{i,j}$ is the distance between winning node i and surrounding node j. Initial value of $\sigma_0 = 1$.

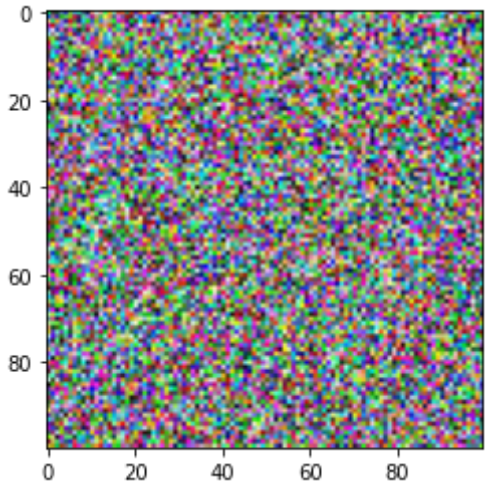
a) Generate, a figure of the original grid (randomly selected) followed by figures of the SOM grid after 20, 40, 100, 1000 epochs.

In [6]:
```
from matplotlib import pyplot as plt
import numpy as np

inputRGB = np.array([
[255,0,0],
[0,255,0],
[0,0,255],
[255,255,0],
[255,0,255],
[0,255,255],
[128,128,0],
[128,0,128],
[0,128,128],
[255,128,0],
[255,0,128],
[128,255,0],
[0,255,128],
[128,0,255],
[0,128,255],
[255,20,147],
[220,20,60],
[255,51,51],
[255,153,51],
[255,255,51],
[51,255,51],
[153,255,51],
[51,255,153],
[51,255,255]])

# Normalize the input (min-max normalize, but we already know the minimu
m and maximum)
normRGB = inputRGB/255

plt.imshow(np.reshape(normRGB,(normRGB.shape[0],1,3)))
print(normRGB.shape)
```

```
(24, 3)
```

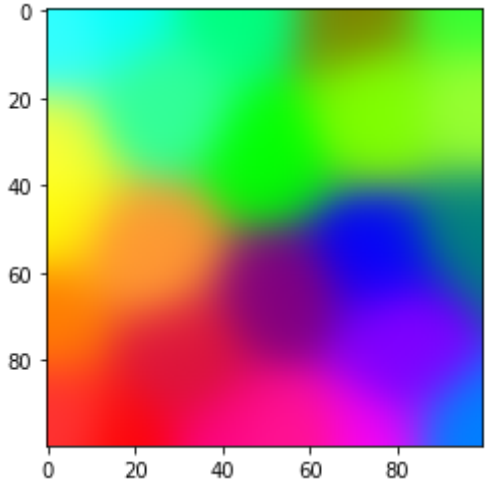
In [7]:
```
from matplotlib import pyplot as plt
# Initialize the system
space_size = 100 # 100 x 100 grid of neurons

# Initialize random weights
weights = np.random.uniform(0, 1, (space_size,space_size,3))
plt.imshow(weights)
```

Out[7]: <matplotlib.image.AxesImage at 0x7f58a3304450>

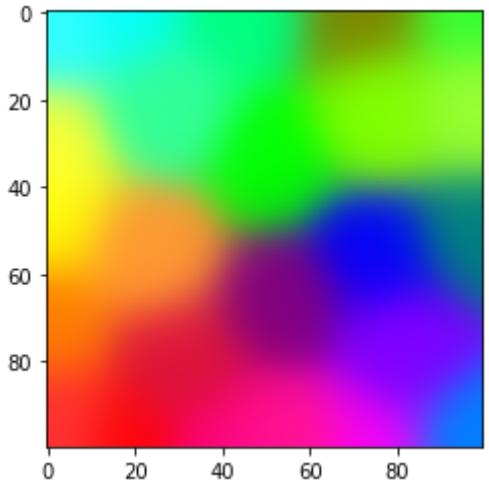


In [8]:
```
from IPython import display
# Constants
alpha_0 = 0.8
alpha = alpha_0
sigma_0 = 10
sigma = sigma_0
epoch = 1
max_epochs = 1000
w = weights.copy()
rows, cols = len(w), len(w[0])
while epoch <= max_epochs:
  for x in normRGB:
    # calculate performance index
    diff = np.linalg.norm( np.subtract(x,w) , axis=2)
    # find index of winning node
    ind = np.array( np.unravel_index(np.argmin(diff, axis=None), diff.sh
ape) )

    for i in range(rows):
      for j in range(cols):
        if i >= 0 and j >= 0 and i < space_size and j < space_size:
          # Calculate distance of winning node from other neurons
          neuron = np.asarray([i, j])
          d = np.linalg.norm( ind - neuron )

          N = np.exp(- np.square(d) / (2 * np.square(sigma) ) )
          # Update weights for neighbourhood
          w[i][j] += alpha * N *( x - w[i][j] )

    # Update learning rate and sigma
    sigma = sigma_0 * np.exp(- epoch / max_epochs )
    alpha = alpha_0 * np.exp(- epoch / max_epochs )

  plot_ind = [20, 40, 100, 1000]
  if epoch in plot_ind:
    print("Epoch Number: {}".format(epoch))
    plt.imshow(w)
    display.display(plt.gcf())

  epoch += 1
```

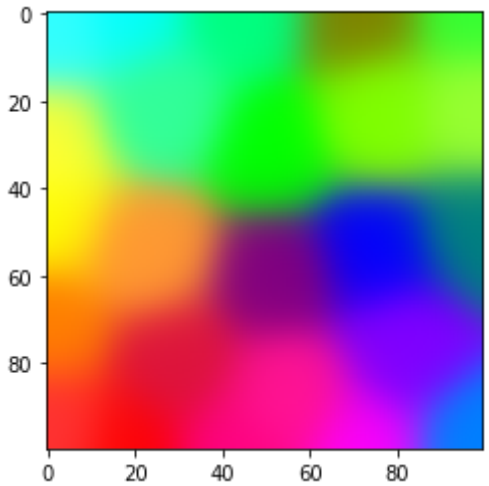
```
Epoch Number: 20
```



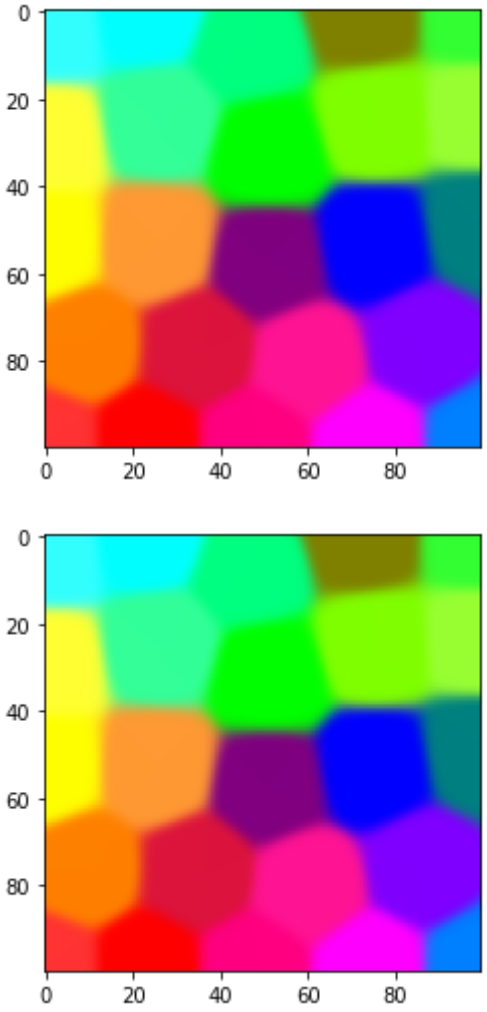

```
Epoch Number: 40
```




```
Epoch Number: 100
```



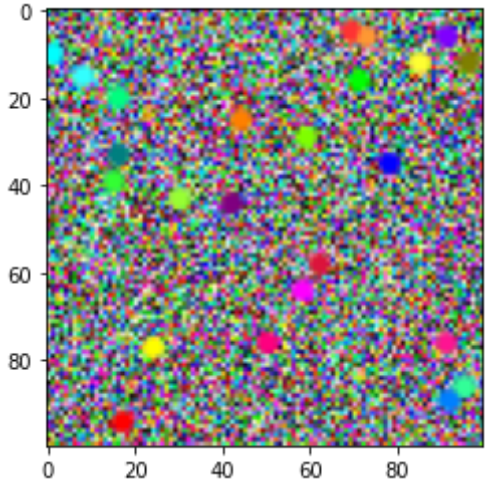

```
Epoch Number: 1000
```



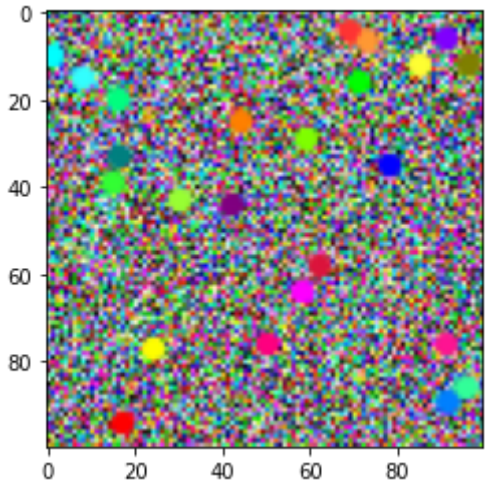
b) Draw your conclusions.
Hint: There is a Matlab library and some code under this link, which you may use:
http://www.cis.hut.fi/somtoolbox/ (http://www.cis.hut.fi/somtoolbox/)

In [9]:
```
from IPython import display
# Constants
alpha_0 = 0.8
alpha = alpha_0
sigma_0 = 1
sigma = sigma_0
epoch = 1
max_epochs = 1000
w = weights.copy()
rows, cols = len(w), len(w[0])
while epoch <= max_epochs:
  for x in normRGB:
    # calculate performance index
    diff = np.linalg.norm( np.subtract(x,w) , axis=2)
    # find index of winning node
    ind = np.array( np.unravel_index(np.argmin(diff, axis=None), diff.sh
ape) )

    for i in range(rows):
      for j in range(cols):
        if i >= 0 and j >= 0 and i < space_size and j < space_size:
          # Calculate distance of winning node from other neurons
          neuron = np.asarray([i, j])
          d = np.linalg.norm( ind - neuron )

          N = np.exp(- np.square(d) / (2 * np.square(sigma) ) )
          # Update weights for neighbourhood
          w[i][j] += alpha * N *( x - w[i][j] )

    # Update learning rate and sigma
    sigma = sigma_0 * np.exp(- epoch / max_epochs )
    alpha = alpha_0 * np.exp(- epoch / max_epochs )

  plot_ind = [20, 40, 100, 1000]
  if epoch in plot_ind:
    print("Epoch Number: {}".format(epoch))
    plt.imshow(w)
    display.display(plt.gcf())

  epoch += 1
```

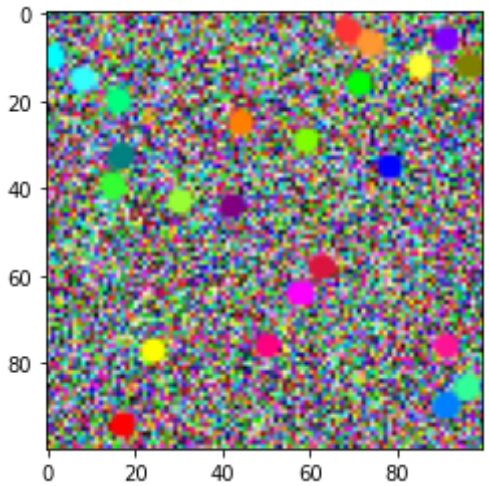
```
Epoch Number: 20
```



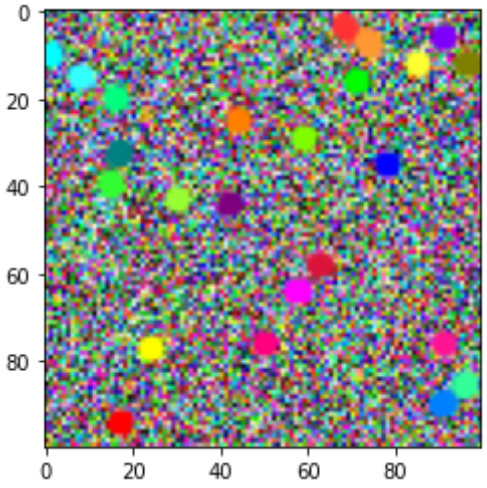

```
Epoch Number: 40
```

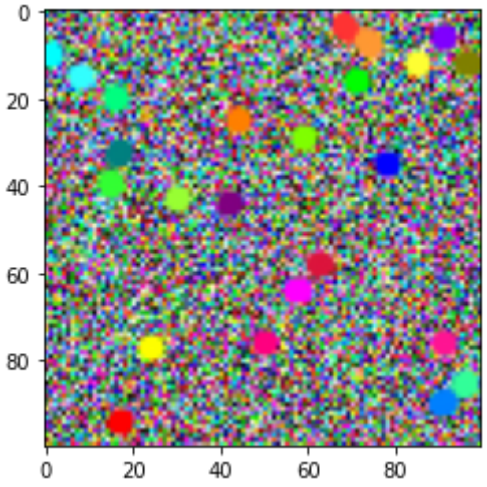



```
Epoch Number: 100
```




```
Epoch Number: 1000
```


0
20
40
60
80
0
20
40
60
80


0
20
40
60
80
0
20
40
60
80

Problem 3

Using your preferred deep learning library, train a small convolutional neural network (CNN) to classify images from the CIFAR10 dataset. Note that most libraries have utility functions to download and load this dataset (TensorFlow, PyTorch). Using the API for loading the dataset will readily divide it into training and testing sets. Randomly sample 20% of the training set and use that as your new training set for the purposes of this problem. Use the test set from the original dataset for validation. Normalize your training and testing sets using Min-Max normalization.

1- Build a multi-layer perceptron with the following layers:
• Dense layer with 512 units and a sigmoid activation function
• Dense layer with 512 units and a sigmoid activation function
• Dense layer (output layer) with 10 units (representing 10 classes in the dataset) and a suitable activation function for the classification task

2- Build a Convolutional neural network with the following architecture:
• 2D Convolutional layer with 64 filters (size of 3x3) and ReLU activation function
• 2D Convolutional layer with 64 filters (size of 3x3) and ReLU activation function
• Flatten layer (to pass to the Fully Connected layers)
• Fully connected (Dense) layer with 512 units and a sigmoid activation function
• Fully connected layer with 512 units and a sigmoid activation function • Dense layer (output layer) with 10 units and a suitable activation function for the classification task

3- Build a Convolutional Neural network with the following architecture:
• 2D Convolutional layer with 64 filters (size of 3x3) and ReLU activation function
• 2x2 Maxpooling layer
• 2D Convolutional layer with 64 filters (size of 3x3) and ReLU activation function
• 2x2 Maxpooling layer
• Flatten layer (to pass to the Fully Connected layers)
• Fully connected (Dense) layer with 512 units and a sigmoid activation function
• Dropout layer with 0.2 dropout rate
• Fully connected layer with 512 units and a sigmoid activation function
• Dropout layer with 0.2 dropout rate
• Dense layer (output layer) with 10 units and a suitable activation function for the classification task

Use a batch size of 32, utilize Adam as the optimizer and choose an appropriate loss function while monitoring the accuracy in both networks. Train each network for 5 epochs.

In [ ]:
```
from __future__ import division
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import keras
import random
from keras import datasets, layers, models
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPool2D, Flatten, Dropout
from tensorflow.keras.utils import to_categorical
```

In [ ]:
```
# Load train & test data
train, test = datasets.cifar10.load_data()
```

In [ ]:
```
# Get indexes for 20% of train data
sample_size = len(train[0])*0.2
sampled_indexes = np.random.randint(low=0, high=49999, size=10000)
```

In [ ]:
```
# Sample training data
train_images_sampled = []
train_labels_sampled = []
for i in sampled_indexes:
  train_images_sampled.append(train[0][i])
  train_labels_sampled.append(train[1][i])

train_images_sampled = np.array(train_images_sampled)
```

In [ ]:
```
# Min-Max Normalization
train_images, test_images = train_images_sampled / 255.0, test[0] / 255.
0
train_labels, test_labels = np.array(train_labels_sampled), test[1]

# To Categorical
train_labels = to_categorical(train_labels, num_classes=10)
test_labels = to_categorical(test_labels, num_classes=10)
```

a) Report the training and testing accuracy for all three networks and comment on the performance of the MLP vs CNNs.

In [ ]:
```
######## DEFINE MODELS ########

## Model 1 ##
def get_model1():
  model1 = models.Sequential()
  model1.add(Flatten(input_shape=(32, 32, 3)))
  model1.add(layers.Dense(512, activation='sigmoid', input_shape=(3072
,)))
  model1.add(layers.Dense(512, activation='sigmoid'))
  model1.add(layers.Dense(10, activation='softmax'))

  model1.compile(optimizer='adam',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])
  return model1

## Model 2 ##
def get_model2():
  model2 = models.Sequential()
  model2.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(3
2, 32, 3)))
  model2.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model2.add(layers.Flatten())
  model2.add(layers.Dense(512, activation='sigmoid'))
  model2.add(layers.Dense(512, activation='sigmoid'))
  model2.add(layers.Dense(10, activation='softmax'))

  model2.compile(optimizer='adam',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])
  return model2

## Model 3 ##
def get_model3():
  model3 = models.Sequential()
  model3.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(3
2, 32, 3)))
  model3.add(layers.MaxPooling2D((2, 2)))
  model3.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model3.add(layers.MaxPooling2D((2, 2)))
  model3.add(layers.Flatten())
  model3.add(layers.Dense(512, activation='sigmoid'))
  model3.add(Dropout(0.2))
  model3.add(layers.Dense(512, activation='sigmoid'))
  model3.add(Dropout(0.2))
  model3.add(layers.Dense(10, activation='softmax'))

  model3.compile(optimizer='adam',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])
  return model3
```

In [ ]:
```
from time import time

# Training and Test Accuracies
model1 = get_model1()
history1 = model1.fit(train_images, train_labels, batch_size=32, epochs=
5,
                      validation_data=(test_images, test_labels), verbose=
1)
print('*************** MODEL 1 ***************')
print('Training Accuracy: {}'.format(history1.history['accuracy'][-1]))
print('Testing Accuracy: {}'.format(history1.history['val_accuracy'][-1
]))
```

```
Epoch 1/5
313/313 [==============================] - 7s 20ms/step - loss: 2.0742
- accuracy: 0.2287 - val_loss: 1.9412 - val_accuracy: 0.2898
Epoch 2/5
313/313 [==============================] - 6s 19ms/step - loss: 1.9013
- accuracy: 0.3051 - val_loss: 1.9171 - val_accuracy: 0.2909
Epoch 3/5
313/313 [==============================] - 6s 19ms/step - loss: 1.8426
- accuracy: 0.3281 - val_loss: 1.8317 - val_accuracy: 0.3360
Epoch 4/5
313/313 [==============================] - 6s 19ms/step - loss: 1.7949
- accuracy: 0.3406 - val_loss: 1.8487 - val_accuracy: 0.3273
Epoch 5/5
313/313 [==============================] - 6s 19ms/step - loss: 1.7450
- accuracy: 0.3695 - val_loss: 1.7524 - val_accuracy: 0.3654
*************** MODEL 1 ***************
Training Accuracy: 0.3695000112056732
Testing Accuracy: 0.3653999865055084
```

In [ ]:
```
model2 = get_model2()
start2 = time()
history2 = model2.fit(train_images, train_labels, batch_size=32, epochs=
5,
                      validation_data=(test_images, test_labels), verbose=
1)
model2_training_time = time() - start2
print('*************** MODEL 2 ***************')
print('Training Accuracy: {}'.format(history2.history['accuracy'][-1]))
print('Testing Accuracy: {}'.format(history2.history['val_accuracy'][-1
]))
```

```
Epoch 1/5
313/313 [==============================] - 141s 449ms/step - loss: 1.80
21 - accuracy: 0.3374 - val_loss: 1.5975 - val_accuracy: 0.4275
Epoch 2/5
313/313 [==============================] - 142s 454ms/step - loss: 1.38
10 - accuracy: 0.5049 - val_loss: 1.4350 - val_accuracy: 0.4753
Epoch 3/5
313/313 [==============================] - 142s 455ms/step - loss: 1.06
38 - accuracy: 0.6228 - val_loss: 1.4194 - val_accuracy: 0.5099
Epoch 4/5
313/313 [==============================] - 142s 454ms/step - loss: 0.69
34 - accuracy: 0.7661 - val_loss: 1.5755 - val_accuracy: 0.4944
Epoch 5/5
313/313 [==============================] - 142s 453ms/step - loss: 0.32
58 - accuracy: 0.8989 - val_loss: 1.6437 - val_accuracy: 0.5137
*************** MODEL 2 ***************
Training Accuracy: 0.8988999724388123
Testing Accuracy: 0.513700008392334
```

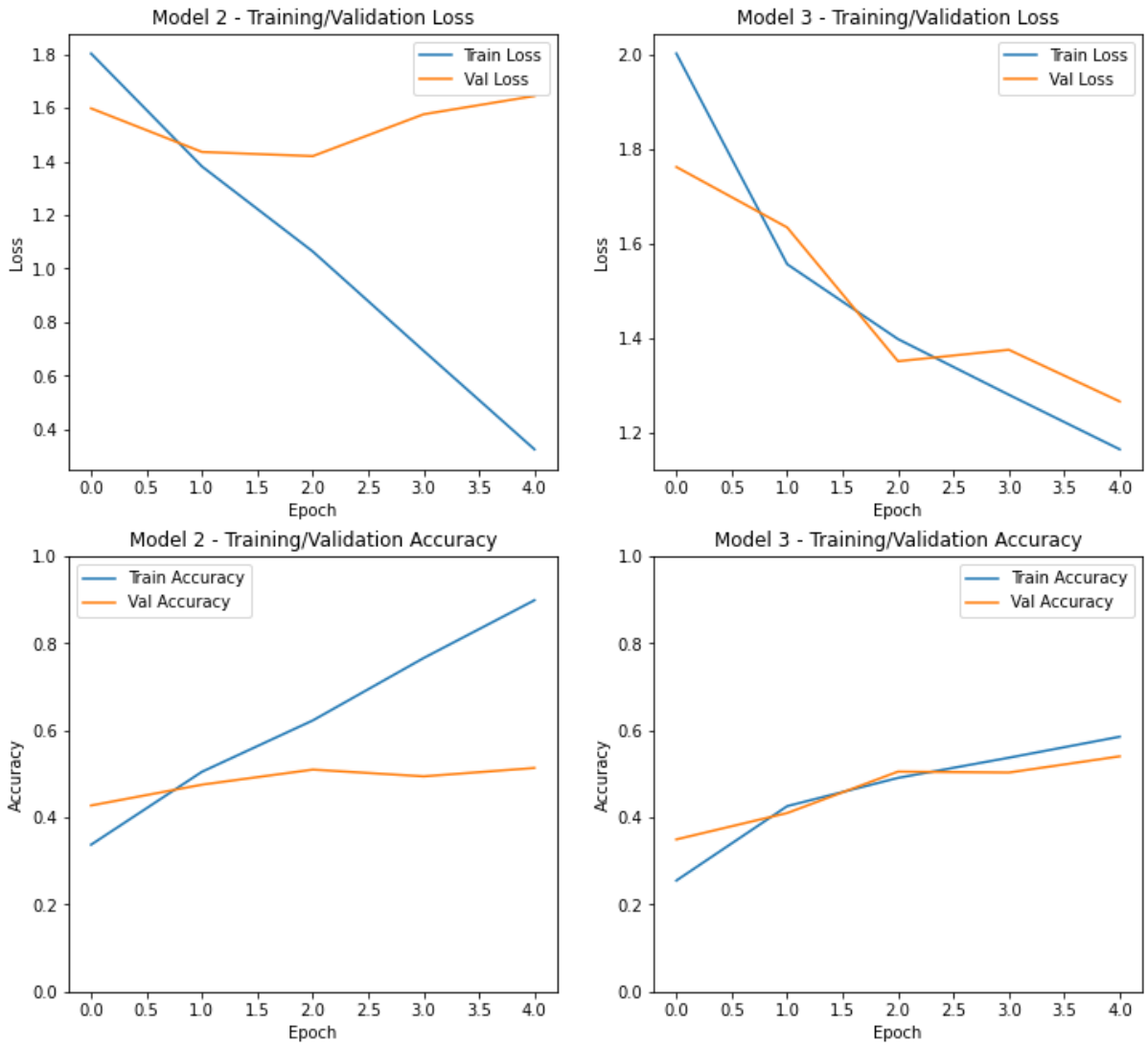
In [ ]:
```
model3 = get_model3()
start3 = time()
history3 = model3.fit(train_images, train_labels, batch_size=32, epochs=
5,
                      validation_data=(test_images, test_labels), verbose=
1)
model3_training_time = time() - start3
print('*************** MODEL 3 ***************')
print('Training Accuracy: {}'.format(history3.history['accuracy'][-1]))
print('Testing Accuracy: {}'.format(history3.history['val_accuracy'][-1
]))
```

```
Epoch 1/5
313/313 [==============================] - 32s 101ms/step - loss: 2.003
1 - accuracy: 0.2551 - val_loss: 1.7629 - val_accuracy: 0.3496
Epoch 2/5
313/313 [==============================] - 31s 100ms/step - loss: 1.557
0 - accuracy: 0.4260 - val_loss: 1.6348 - val_accuracy: 0.4102
Epoch 3/5
313/313 [==============================] - 31s 100ms/step - loss: 1.398
4 - accuracy: 0.4909 - val_loss: 1.3516 - val_accuracy: 0.5053
Epoch 4/5
313/313 [==============================] - 32s 101ms/step - loss: 1.280
6 - accuracy: 0.5372 - val_loss: 1.3758 - val_accuracy: 0.5032
Epoch 5/5
313/313 [==============================] - 31s 101ms/step - loss: 1.165
1 - accuracy: 0.5855 - val_loss: 1.2664 - val_accuracy: 0.5404
*************** MODEL 3 ***************
Training Accuracy: 0.5855000019073486
Testing Accuracy: 0.5404000282287598
```

b) Plot the training and validation curves for the two CNNs and comment on the output. How does the training time compare for each of the CNNs? How does the different architectures influence these results? What do you expect the accuracies to be if the networks were trained for more epochs?

In [ ]:
```
# MODEL 2 & 3 - LOSS
fig, ax = plt.subplots(2, 2, figsize=(12, 11))
ax[0,0].set_title('Model 2 - Training/Validation Loss')
ax[0,0].plot(history2.history['loss'], label='Train Loss')
ax[0,0].plot(history2.history['val_loss'], label = 'Val Loss')
ax[0,0].set_xlabel('Epoch')
ax[0,0].set_ylabel('Loss')
ax[0,0].legend(loc='best')

ax[0,1].set_title('Model 3 - Training/Validation Loss')
ax[0,1].plot(history3.history['loss'], label='Train Loss')
ax[0,1].plot(history3.history['val_loss'], label = 'Val Loss')
ax[0,1].set_xlabel('Epoch')
ax[0,1].set_ylabel('Loss')
ax[0,1].legend(loc='best')

# MODEL 2 & 3 - ACCURACY
ax[1,0].set_title('Model 2 - Training/Validation Accuracy')
ax[1,0].plot(history2.history['accuracy'], label='Train Accuracy')
ax[1,0].plot(history2.history['val_accuracy'], label = 'Val Accuracy')
ax[1,0].set_xlabel('Epoch')
ax[1,0].set_ylabel('Accuracy')
ax[1,0].set_ylim(0,1)
ax[1,0].legend(loc='best')

ax[1,1].set_title('Model 3 - Training/Validation Accuracy')
ax[1,1].plot(history3.history['accuracy'], label='Train Accuracy')
ax[1,1].plot(history3.history['val_accuracy'], label = 'Val Accuracy')
ax[1,1].set_xlabel('Epoch')
ax[1,1].set_ylabel('Accuracy')
ax[1,1].set_ylim(0,1)
ax[1,1].legend(loc='best')
```

Out[ ]: <matplotlib.legend.Legend at 0x7f30bb9f9b10>



In [ ]:
```
# Training Time
print('Model 2 Training Time: {} seconds'.format(round(model2_training_t
ime,3)))
print('Model 3 Training Time: {} seconds'.format(round(model3_training_t
ime,3)))
```

```
Model 2 Training Time: 709.303 seconds
Model 3 Training Time: 202.609 seconds
```