The State University of New York Buffalo, New York 14260

CSE 574: Introduction to Machine Learning Project 2 Report Fall 2020

Sargur N. Srihari, Mihir Chauhan, Sougata Saha

Neural Network Classifier for Image Classification using Supervised Learning and Unsupervised Learning

Hemant Kumar Das

50356421

hemantku@buffalo.edu

Introduction:

Artificial neural networks (ANNs) are statistical learning algorithms that are inspired by properties of the biological neural networks. They are used for a wide variety of tasks, from relatively simple classification problems to speech recognition and computer vision. ANNs are loosely based on biological neural networks in the sense that they are implemented as a system of interconnected processing elements, sometimes called nodes, which are functionally analogous to biological neurons. The connections between different nodes have numerical values, called weights, and by altering these values in a systematic way, the network is eventually able to approximate the desired function.

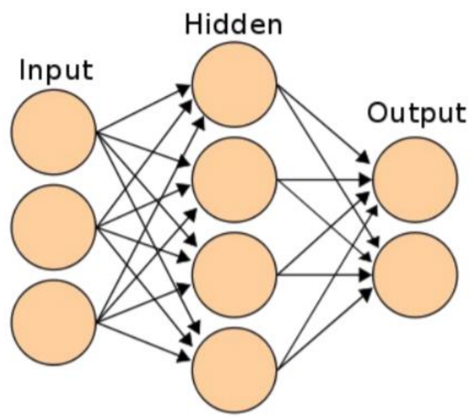
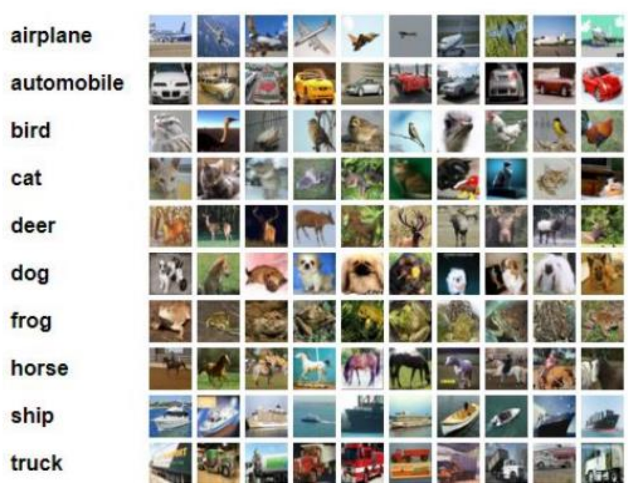


Figure 1. A simple neural network with one hidden layer.[2]

In this project we will be using CIFAR 10 dataset and we will train our neural network which has only one hidden layer in order to predict the classes of the test data set. The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The classes are completely mutually exclusive.

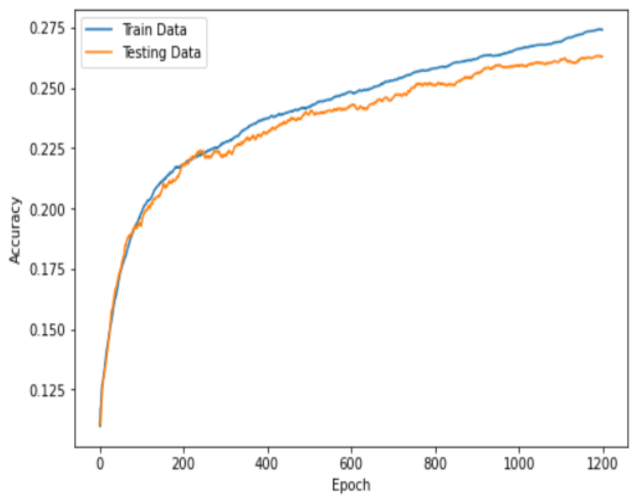


---***---

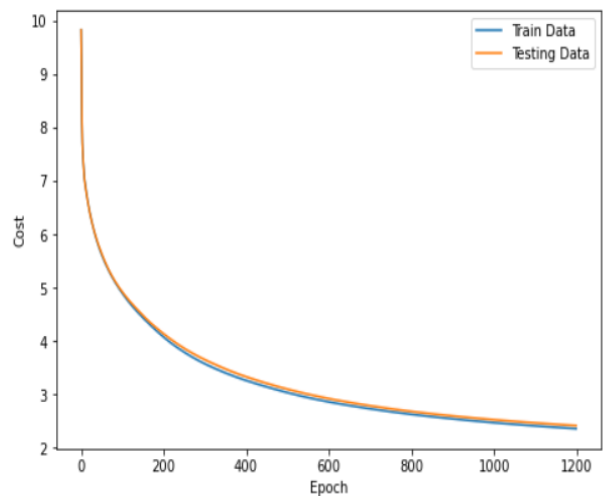
Part 1: Supervised Learning(SLA)

In this part of the project, we are using a neural network in order to predict classes of the CIFAR-10 dataset. After carefully observing the results and trying a different set of values for the hyperparameters, I found out the best hyperparameters for this task. (Learning Rate= 0.1, Hidden Neurons=500, Epochs=1200). The activation function for the layer between inputs and the hidden layer is sigmoid. The output is coming from SoftMax activation which is required for Multi-class classification.

<matplotlib.legend.Legend at 0x1ed58af2dc0>



<matplotlib.legend.Legend at 0x1ed58a6e2b0>

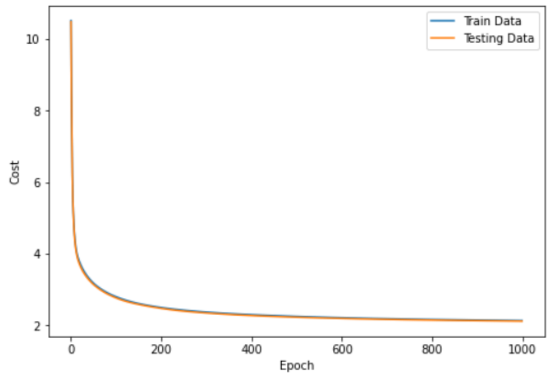



```
Confusion Matrix

[[416  42  74  32  62  32  24  51 203  64]
 [ 76 255  57  72  35  55  41  82 111 216]
 [172  40 196  57 157  67 127  73  59  52]
 [ 73  78 123 161  96 147 117  89  51  65]
 [ 83  41 147  77 237  84 150  82  51  48]
 [ 89  43 114 144 121 209 100  85  53  42]
 [ 45  44 116 115 148  76 288  72  31  65]
 [ 77  52  96  88 114  94  78 234  63 104]
 [149  83  33  33  37  41  22  31 469 102]
 [ 77 164  37  50  47  37  40  70 128 350]]
0.2815
```

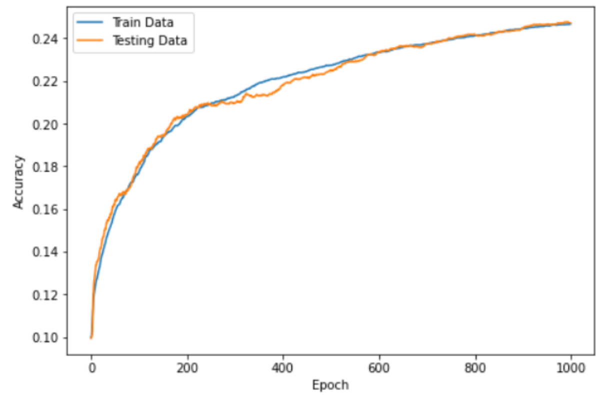
This is the Confusion Matrix and the accuracy score for the test data which has come to be 0.2815. The reason for this low accuracy is because our model looks at each pixel rather than the whole picture. The accuracy does not increase significantly and remains almost the same after increasing epochs more than 1200 while keeping other hyperparameters the same. The neural network can be improved by adding more hidden layers in order to provide the network with the tools to learn more complex relationships in the data.

The accuracy seems to increase slightly when we increase the number of hidden neurons up to 500. There after adding more neurons is not taking my accuracy higher. The below image is for 100 hidden neurons trained for 1000 epochs. I got accuracy for test data as 0.2474.

<matplotlib.legend.Legend at 0x7f61d32e03c8>



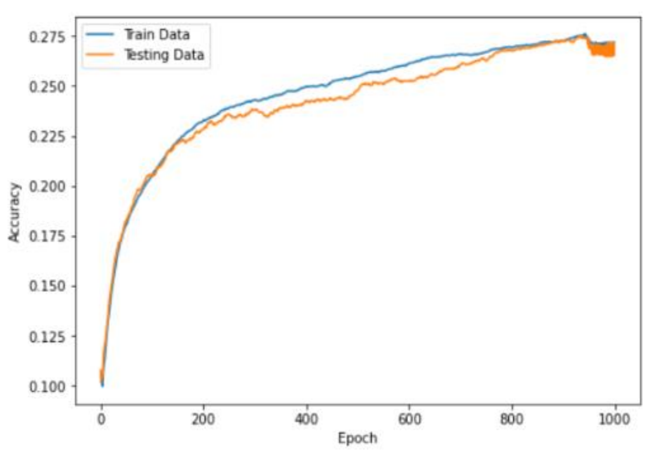
<matplotlib.legend.Legend at 0x7f61d2e40f98>



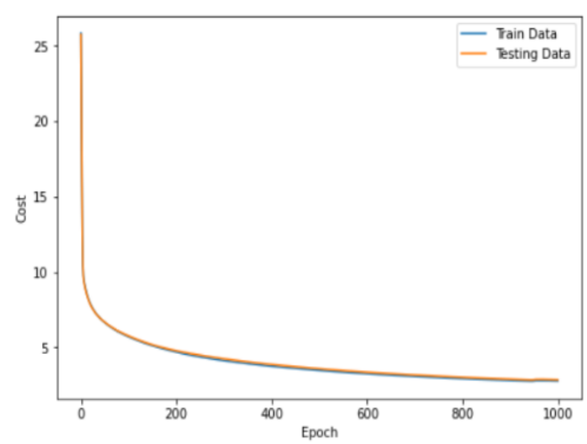

```
Confusion Matrix

[[502  44  95  29  30  10  29  27 182  52]
 [120 258  58  62  32  36  56  62 144 172]
 [216  55 211  69 169  35 100  67  50  28]
 [117  82 129 119 146  87 118  94  42  66]
 [158  42 232  77 247  28  93  56  32  35]
 [128  64 126 144 135 107 111  91  50  44]
 [ 81  94 169  84 166  56 217  66  22  45]
 [126  68 126  82 146  44 106 146  45 111]
 [295  81  32  31  22  22  15  21 396  85]
 [110 170  49  36  35  18  54  72 185 271]]
0.2474
```

The below image is for 1200 hidden neurons trained up to 1000 epochs. This accuracy and cost are almost the same as that of 500 hidden neurons. So, increasing hidden neurons beyond 500 does not bring much change in accuracy and cost.

<matplotlib.legend.Legend at 0x25c65861880>



<matplotlib.legend.Legend at 0x25c657ec760>



---***---

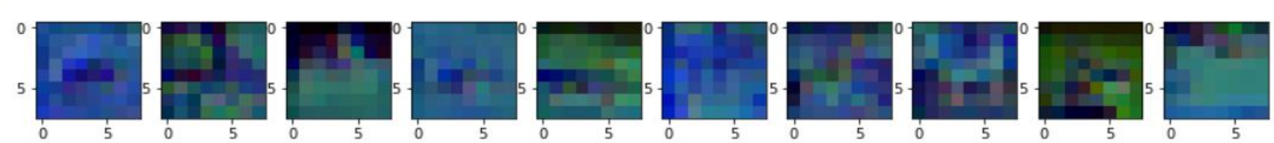
Part 2: Unsupervised Learning (USLA)

In this part of the project, we are using a Convolutional Autoencoder using Keras library and then using the KMeans clustering algorithm in order to cluster classes of the CIFAR-10 dataset. The images are classified into clusters based on the similarity of pixel values. Each image is assigned a cluster label value given by kmeans.fit_predict. So kmeans.fit_predict is an array of length 50000 as there are 50000 images in the training set. Then we map the cluster numbers with the most probable label. The autoencoder model and results of images from the autoencoder is shown below. There are only 10 epochs in the autoencoder. The loss is "MSE" (mean squared error) and the optimizer is "SGD" (stochastic gradient descent). Activation function "Relu" is used in every convolution layer.

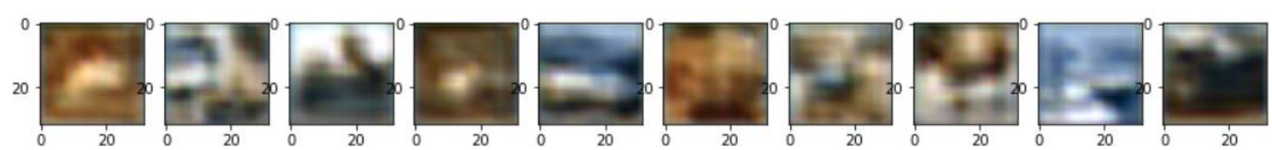
```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 32, 32, 3)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 32, 32, 16)        448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 3)         435
_________________________________________________________________
CodeLayer (MaxPooling2D)     (None, 8, 8, 3)           0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 8, 8, 3)           84
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 16, 16, 3)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 16, 16, 16)        448
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 32, 32, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 3)         435
=================================================================
Total params: 1,850
Trainable params: 1,850
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/10
1563/1563 [==============================] - 84s 54ms/step - loss: 0.0769
Epoch 2/10
1563/1563 [==============================] - 82s 53ms/step - loss: 0.0376
Epoch 3/10
1563/1563 [==============================] - 83s 53ms/step - loss: 0.0294
Epoch 4/10
1563/1563 [==============================] - 83s 53ms/step - loss: 0.0253
Epoch 5/10
1563/1563 [==============================] - 84s 54ms/step - loss: 0.0215
Epoch 6/10
1563/1563 [==============================] - 84s 54ms/step - loss: 0.0188
Epoch 7/10
1563/1563 [==============================] - 84s 54ms/step - loss: 0.0170
Epoch 8/10
1563/1563 [==============================] - 84s 54ms/step - loss: 0.0158
Epoch 9/10
1563/1563 [==============================] - 83s 53ms/step - loss: 0.0150
Epoch 10/10
1563/1563 [==============================] - 83s 53ms/step - loss: 0.0144
<tensorflow.python.keras.callbacks.History at 0x7ff50df7bef0>
```

Actual Images



Coded Images



Decoded Images

All 50000 images of the train data set are given to the autoencoder and coded images from the latent layer are fetched and given to the KMeans model to cluster them. Below is the confusion matrix for y_train and generated cluster ids. Then we map the cluster ids with the most probable value of y_train. The model accuracy on the unseen test data set is 0.245 and the confusion matrix between test cluster ids and y_test is shown below.

```
[[1052   94  438  518  108  290  933  162  226 1179]
 [ 320  274  316 1228  469  585  705  377  645   81]
 [ 380  400  731  524  368  229  196 1051  846  275]
 [ 277  454  793  460  877  320  132  739  891   57]
 [ 147  697  489  431  740  272  210 1062  875   77]
 [ 172  421  710  624 1123  202   91  798  797   62]
 [ 151 1127  604  284  347  169   33 1152 1111   22]
 [ 129  291  680  372  690 1001  257 1013  540   27]
 [ 444   15  265  899  264  316 1684   62  165  886]
 [ 201  128  395  704  160 1469 1247  398  242   56]]
```

```python
#mapping with most probabale values of cluster ids
max_val=np.argmax(cf,axis=0)
print("Max value of cluster id's",max_val)
```

```
Max value of cluster id's [0 6 3 1 5 9 8 6 6 0]
```

```
The accuracy of test data is  0.245

 The confusion matrix is
[[487 109   0  85  37  18  39   0 171  54]
 [ 83 256   0  75  71  73 198   0 145  99]
 [141 110   0 152 209  75 238   0  35  40]
 [ 69  97   0 158 158 158 273   0  24  63]
 [ 33  86   0 108 224 131 336   0  31  51]
 [ 57 114   0 151 174 220 222   0  18  44]
 [ 33  56   0 130 223  61 454   0   9  34]
 [ 37  69   0 142 173 143 151   0  60 225]
 [257 175   0  41  19  48  33   0 356  71]
 [ 62 145   0  78  85  27  64   0 244 295]]
```

There are cases when the most probable y values for some clusters are the same. We can change repeating y's which may improve the accuracy of our model. Given below is an example of how repeating digits are replaced by digits that were not present in the original y. The right image below shows 3 vectors out of which 1st vector is where repeating digits are replaced with 'x'. 2nd vector is digits which were missing in original y and 3rd vector is my new y's for the cluster ids.

```
[27]  max_val

array([9, 2, 5, 4, 6, 8, 5, 6, 0, 9])
```

```python
max_val_new=matrix_new(max_val)
print(max_val_new)
```

```
['x', 2, 'x', 4, 'x', 8, 5, 6, 0, 9]
[1, 3, 7]
[1 2 3 4 7 8 5 6 0 9]
```

This technique can be efficient but most of the times it will fail and gives us a better accuracy only 1 out of ten times. So, we do not include this in our model.

```
Epoch 11/100
1563/1563 [==============================] - 106s 68ms/step - loss: 0.0177
Epoch 12/100
1563/1563 [==============================] - 105s 67ms/step - loss: 0.0174
Epoch 13/100
1563/1563 [==============================] - 106s 68ms/step - loss: 0.0172
Epoch 14/100
1563/1563 [==============================] - 106s 68ms/step - loss: 0.0171
Epoch 15/100
1563/1563 [==============================] - 106s 68ms/step - loss: 0.0169
Epoch 16/100
1563/1563 [==============================] - 107s 68ms/step - loss: 0.0168
Epoch 17/100
1563/1563 [==============================] - 106s 68ms/step - loss: 0.0166
Epoch 18/100
1563/1563 [==============================] - 105s 67ms/step - loss: 0.0165
Epoch 19/100
1563/1563 [==============================] - 106s 68ms/step - loss: 0.0164
Epoch 20/100
1563/1563 [==============================] - 106s 68ms/step - loss: 0.0163
Epoch 21/100
1563/1563 [==============================] - 106s 68ms/step - loss: 0.0162
Epoch 22/100
```

Also, training our model beyond 10 epochs does not decrease the loss by a significant amount. KMeans has default iterations set as 300. Increasing the KMeans iterations greater than 300 also does not increase my accuracy.

We can also try another way to map cluster ids to the value of y_train. This method is called linear assignment. We need to import linear assignment from sklearn.utils.linear_assignment. We can also use scipy.optimize to import linear_sum_assignment if we get warning while using the first.

```
from scipy.optimize import linear_sum_assignment
c_val=linear_sum_assignment(cf,maximize=True)[1]
```

```
c_val
```

```
array([6, 4, 7, 0, 2, 5, 8, 3, 1, 9])
```

The method is also known as the Munkres or Kuhn-Munkres algorithm or Hungarian Algorithm.

```
[51]    print(accuracy_score(y_train,img_label))
```

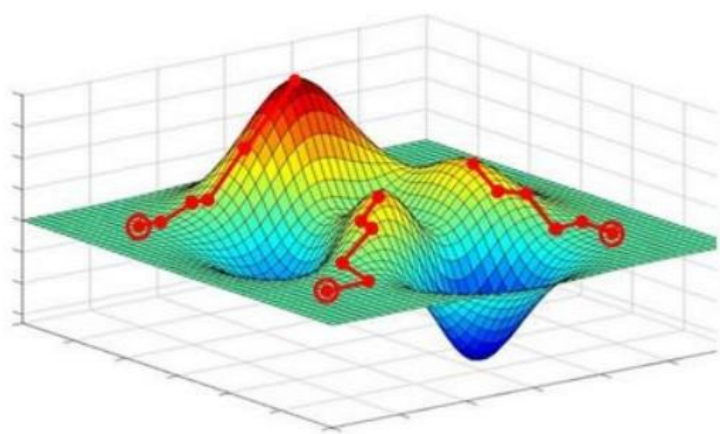
```
0.13284
```

After mapping the cluster ids according to linear assignment, we do not get a very good accuracy on the train dataset. The accuracy is only 0.13284. We do not use this for the test dataset.

---***---

Conclusions:

Part 1 (Supervised Learning) gives us a test accuracy of 0.2815 for multiclass classification. Part 2 (Unsupervised Learning) gives us a test accuracy of 0.245 for clustered data. Both of these uses the gradient descent approach to minimize loss and improve accuracy of the model. Gradient descent works by finding the gradient of the loss function with respect to the weights of the network, telling us the direction where the function increases the most. In order to minimize the loss, it is required to subtract a fraction of the gradient from the corresponding weight vector.



Weight calculation is done using backpropagation. It calculates the gradient of the loss function with respect to all the weights in the network by iteratively applying the multivariable chain rule. Loss here is the cross-entropy loss.

$\partial \text{loss} / \partial w2 = (\partial \text{loss} / \partial z)\ (\partial z / \partial w2)$

$\partial \text{loss}\ \partial w1 = (\partial \text{loss} / \partial z)\ (\partial z / \partial y)\ (\partial y / \partial w1)$

The aim of this project was to implement a simple feed-forward neural network and to build a convolutional autoencoder with any number of pool and conv layers with KMeans Clustering. Both of these goals have been achieved.

References:

http://www.cs.toronto.edu/~kriz/cifar.html

http://en.wikipedia.org/wiki/Backpropagation#Summary

http://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/Artificial_neural_network.svg/350pxArtificial_neural_network.svg.png

http://www.forexartilect.com/img/momo.jpg

---***---