CSE 474/574: Introduction to Machine Learning Project 2 Report

Sargur N. Srihari, Mihir Chauhan, Sougata Saha

University at Buffalo, The State University of New York Buffalo, New York 14260

- SURYA MUTHIAH PILLAI

TABLE OF CONTENTS

1. Supervised Learning (SLA) vs Unsupervised Learning (USLA)

Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs. Basically, it is a learning in which we train the machine using data which is labeled that means some data is already tagged with the correct answer.

Unsupervised learning is a type of machine learning that looks for previously undetected patterns in a data set with no pre-existing labels and with a minimum of human supervision. It is the training of machine using information that is neither classified nor labeled and allowing the algorithm to act on that information without guidance.

2. Task Given

The task of this project is to classify an image into one of ten classes using CIFAR- 10 dataset for this classification task. Two approaches implemented are:

1. **Supervised Learning Approach (SLA):** Build a Neural Network Classifier (NN) with one hidden layer to be trained and tested on CIFAR-10 dataset.

2. **Unsupervised Learning Approach (USLA):** USLA is a two-step learning approach for image classification.

   (a) STEP 1: Extract image features using a Convolutional AutoEncoder (Conv-AE) for CIFAR- 10 dataset using an open-source neural-network library, Keras.

   (b) STEP 2: Classify Auto-Encoded image features using K-Means clustering algorithm using sklearns.cluster.KMeans (off-the-shelf clustering libraries)

3. CIFAR-10 Dataset

For training and testing of the classifiers, we will use the **CIFAR-10** dataset. 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.

4. Import Libraries and Load Data (Step 1 & Step 2)

Step 1: Import Libraries

- **NumPy** - NumPy is used to work with arrays
- **sklearn.preprocessing.MinMaxScaler** - Transform features by scaling each feature to a given range
- **matplotlib.pyplot** - Provides a MATLAB-like plotting framework to plot accuracy and loss function
- **tensorflow.keras.layers** - Layers are the basic building blocks of neural networks in Keras
- **sklearn.metrics.confusion_matrix** - Compute confusion matrix to evaluate the accuracy of a classification
- **sklearn.metrics.accuracy_score** - Accuracy classification score
- **sklearn.cluster.KMeans** - K-Means clustering to cluster the input
- **scipy.optimize.linear_sum_assignment** – To solve the confusion matrix and calculate accuracy

Step 2: Data Loading

We use **tensorflow.keras.datasets.cifar10.load_data()** to load the CIFAR-10 dataset into Training and Testing data.

5. Implementation of SLA (Step 3 – Step 7)

- After importing the required libraries and loading the CIFRA-10 dataset, we define the required methods used by our Supervised Learning Approach (SLA).

<u>Defined Methods:</u>

1. *Minmax(x)* – return minmax-scaled value of x

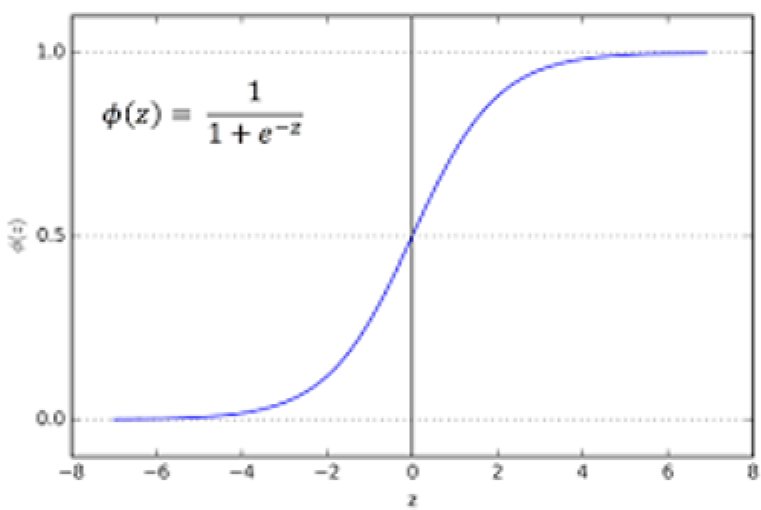
The transformation is given by:

$X_{std} = X - X_{min}\ X_{max} - X_{min}$

$X_{scaled} = X_{std} * (maxRange - minRange) + minRange$

$maxRange = 1,\ minRange = 0$

2. *Sigmoid(z)* - return (1 / (1 + e^(-z)))



3. *Softmax(z)* –

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

4. *Calculate_accuracy(y_pred, y_true)* – return the accuracy of our predicted labels with respect to true labels.

5. *Calculate_loss(y_pred, y_true)* – return the loss calculated by the given loss entropy. The loss entropy we use for our approach is **Categorical Cross Entropy Loss (L)**, which is given by,

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\\text{size}}} y_i \cdot \log \hat{y}_i$$

Step 3: Scaling Image Pixel Values

- We scale the input features using MinMaxScaler.
- We use *Minmax(x)* on x_train and x_test to scale the features of training and testing data respectively.

Step 4: One Hot Encoding of target variable

- A one hot encoding is a representation of categorical variables as binary vectors. This first requires that the categorical values be mapped to integer values. Then, each integer value is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1.
- For example, if a target variable belongs to class 6 out of the 10 classes, our true label is marked as [5] (starting from 0). We then use one hot encoding to change it to a vector of binary targets [0 0 0 0 0 1 0 0 0 0] where 1 is at the 6^{th} position denoting the class 6.
- We apply one hot encoding to both train and test target data.

Step 5: Initialization of Variables

(i) Epochs = 700

(ii) Learning_rate = 0.1

(iii) Initialize Weights and biases:

Since we are using a Neural network with one hidden layer and the genesis equation as y = SoftMax(W2.(W1X +b1)+b2), we need two weights and two biases.

- ✓ W1 = np.random.randn(n_h,n_x)* 0.1
- ✓ b1 = np.zeros((n_h,1))
- ✓ W2 = np.random.randn(n_y,n_h) * 0.1
- ✓ b2 = np.zeros((n_y,1))

n_x = number of features = 3072 (32x32x3 = 3072)

n_y = number of classes = 10

n_h = size of hidden layers = any (in our case, n_h = 300)

Therefore, the initialization looks like,

```
epochs=700

learning_rate = 0.1

m=x_train.shape[0]

W1 = np.random.randn(300,3072)*0.1
b1 = np.zeros((300,1))
W2 = np.random.randn(10,300)*0.1
b2 = np.zeros((10,1))
```

We multiply the weights with 0.1 to make the random values work in accordance with our data. Multiplying the weights with 0.1 shows a greater accuracy when compared to not multiplying with it. Normally, the learning_rate would be 0.01 or 0.001, and this yielded a result which required a longer time to improve the accuracy, i.e., accuracy increased at a slower rate. So, by tuning up the learning_rate a bit higher (0.001 **x10** =0.1) and dividing the weights by the same factor (W = W **x0.1**), we achieve a greater accuracy in a lesser time.

Step 6: Training a Neural Network with 1 hidden layer using Gradient Descent Algorithm

X – 3072x50000 (features x no. of images)

Y – 10x50000 (classes x no. of images)

Step 6.1: We use genesis equation y = SoftMax(W2.(W1X +b1)+b2) to predict our class for the input features X

```
# Step 6.1: Use genesis equation y_pred = SM(W2.(W1X +b1)+b2)
Z1 = np.dot(W1,X) + b1
A1 = sigmoid(Z1)
Z2 = np.dot(W2,A1) + b2
A2 = softmax(Z2)
```

Step 6.2: Find Categorical Cross Entropy Loss for predicted value **A2** and truth value **Y**

As we have defined the methods for calculating the categorical loss entropy, which is given by

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\\text{size}}} y_i \cdot \log \hat{y}_i$$

We call our method Calculate_loss(A2, Y) to calculate the loss which is defined by

```
def calculate_loss(y,p):
    value = []
    y=y.T
    p=p.T
    for i in range(y.shape[0]):
        val = np.multiply(y[i],np.log(p[i]))
        value.append(sum(val))
    CCE = -(1 / y.shape[0]) *np.sum(value)
    return CCE
```

Step 6.3: Find dW1, db1, dW2, db2

We need to calculate the partial derivatives of the weights and biases in order to update them in the next epoch. So, we calculate it as follows,

```
# Step 6.3: Find dW1, db1, dW2, db2
dZ2 = A2 - Y
dW2 = 1 / m * (np.dot(dZ2,A1.T))
db2 = 1 / m * (np.sum(dZ2,axis = 1,keepdims = True))
dZ1 = np.dot(W2.T,dZ2) * (A1*(1-A1))
dW1 = 1 / m * (np.dot(dZ1,X.T))
db1 = 1 / m * (np.sum(dZ1,axis = 1,keepdims = True))
```

Step 6.4: Update W1, W2, b1, b2 using learning_rate (η) as follows,

```
# Step 6.4: Update W1, W2, b1, b2 using learning rate
W1 = W1 - learning_rate * dW1
b1 = b1 - learning_rate * db1
W2 = W2 - learning_rate * dW2
b2 = b2 - learning_rate * db2
```

Step 6.5: Find Train and Test Accuracy

To calculate the accuracy we call our defined method as *Calculate_accuracy(A2, Y)*

We define the method as follows,

```
def calc_accuracy(p,y):
    p=p.T
    y=y.T
    p_class=[]
    y_class=[]
    for i in range(p.shape[0]):
        p_class.append(np.argmax(p[i]))
        y_class.append(np.argmax(y[i]))
    return accuracy_score(y_class, p_class)*100
```

where, p_class is our predicted class and y_class is the true class. We use numpy.argmax() to find our class which is the max value in our 10-label output and use sklearn.metrics.accuracy_score to find the accuracy of our classification.

Step 7: Evaluation of SLA by tuning Tune-Hyperparameters

Now we evaluate our model by adjusting the hyper parameters. When fix the epochs as 700 and learning rate = 0.01. The accuracy is too low at the beginning and increases very slowly as our learning rate is less. So, we increase the learning_rate to 0.1 and if tested, it can be found out that the accuracy is so inconsistent and fluctuates very often which also happens to be the case in loss. Since we need a better accuracy at a faster increasing rate as each epoch takes a considerable amount of time, we initialize the weights by multiplying it by 0.1 and keep the learning_rate as 0.1. Now it can be seen the accuracy is increasing by each epoch at a higher rate and loss is decreased gradually as the epoch goes higher. So, the milestones achieved are as follows,

Epoch = 1 to 5

```
epoch :  1 / 500
train_acc = 8.676
test_acc = 8.7
loss = 2.5433297517899596

epoch :  2 / 500
train_acc = 9.956
test_acc = 10.25
loss = 2.3756007802371184

epoch :  3 / 500
train_acc = 10.92
test_acc = 11.379999999999999
loss = 2.3256765499392227

epoch :  4 / 500
train_acc = 11.776
test_acc = 12.24
loss = 2.3122033349453033

epoch :  5 / 500
train_acc = 12.642000000000001
test_acc = 12.950000000000001
loss = 2.300795958305784
```

Epoch = 95 to 100

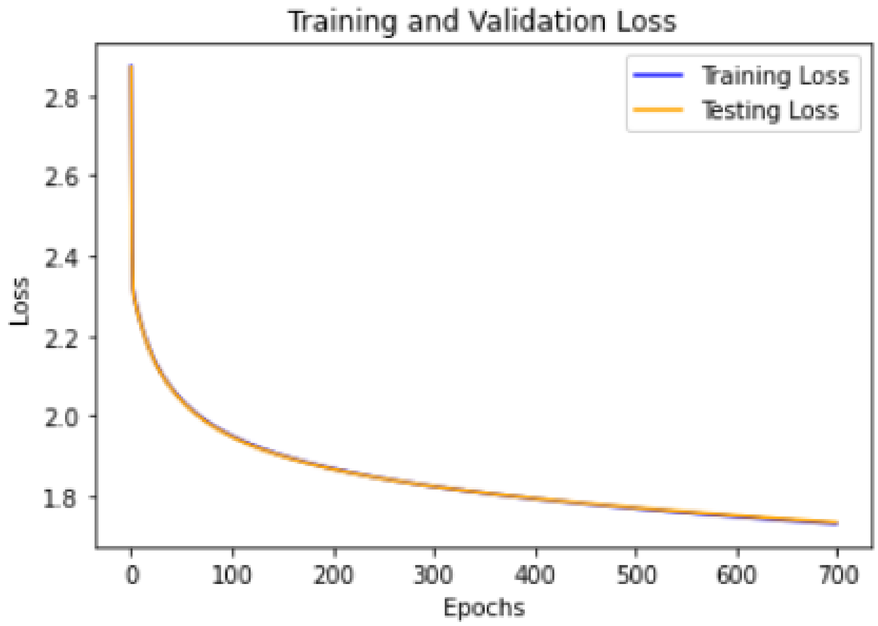
```
epoch :  95 / 500
train_acc = 31.416
test_acc = 31.28
loss = 1.9661042249724114

epoch :  96 / 500
train_acc = 31.491999999999997
test_acc = 31.34
loss = 1.9647008513800432

epoch :  97 / 500
train_acc = 31.552000000000003
test_acc = 31.369999999999997
loss = 1.9633143698043212

epoch :  98 / 500
train_acc = 31.596000000000004
test_acc = 31.41
loss = 1.9619444676992004

epoch :  99 / 500
train_acc = 31.646
test_acc = 31.509999999999998
loss = 1.9605908423525493

epoch :  100 / 500
train_acc = 31.702
test_acc = 31.6
loss = 1.9592532002788188
```

Epoch =695 to 700

```
epoch :  695 / 700
train_acc = 39.910000000000004
test_acc = 40.02
loss = 1.7328806984585106

epoch :  696 / 700
train_acc = 39.914
test_acc = 40.02
loss = 1.7327112921529133

epoch :  697 / 700
train_acc = 39.924
test_acc = 40.01
loss = 1.7325420547463133

epoch :  698 / 700
train_acc = 39.928000000000004
test_acc = 40.0
loss = 1.7323729856502679

epoch :  699 / 700
train_acc = 39.932
test_acc = 40.0
loss = 1.7322040842787838

epoch :  700 / 700
train_acc = 39.936
test_acc = 40.0
loss = 1.7320353500483001
```

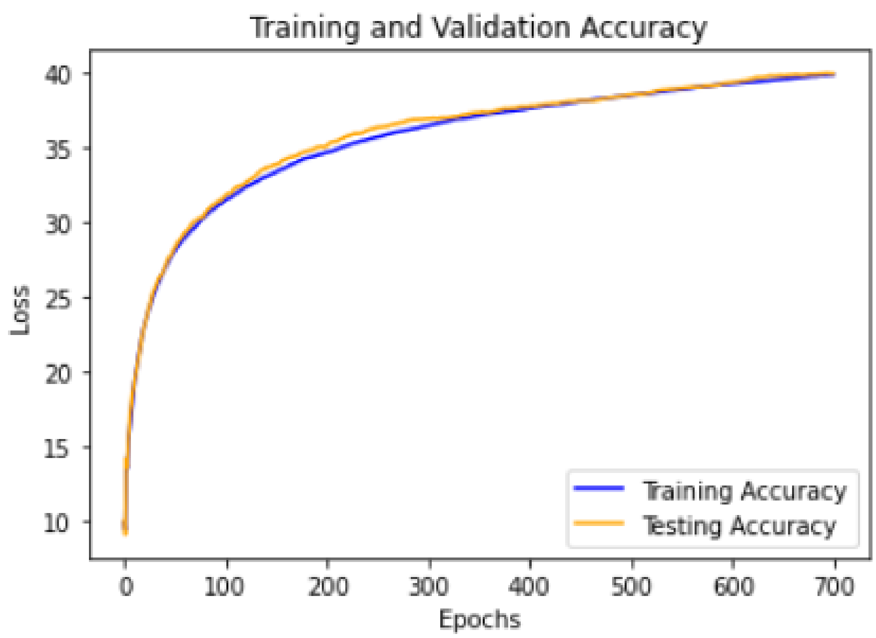
As we can see, there is no significant change in accuracy when epoch nears 700. So, we stop there and plot the Training and Testing data's accuracy and loss against epochs.

Training and Testing Cost vs Number of Epochs:



As the epoch approaches 700, there is not much reduction of a loss. And as we can see, the loss over the 700 epochs has a range from loss=2.54 to loss=1.73

Training and Testing Accuracy vs Number of Epochs:



Our classification accuracy from epoch = 1-700 has increased from 8.7% to 40.0% for the CIFAR-10 dataset.

Confusion Matrix:

A confusion matrix is a matrix that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. Also known as an error matrix, it is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one.

Since our cifar-10 dataset is 10-class model, our confusion matrix is 10x10 i.e., (true_labels x predicted_lables)

The matrix can be created using sklearn.metric.confusion_matrix

So, our test data yielded the following confusion_matrix.

$$
\begin{array}{cc}
\text{y\_true} & \begin{bmatrix}
\boxed{469} & 40 & 61 & 17 & 15 & 18 & 29 & 52 & 227 & 72 \\
64 & \boxed{455} & 25 & 46 & 20 & 36 & 48 & 40 & 93 & 173 \\
126 & 47 & \boxed{261} & 73 & 124 & 87 & 140 & 65 & 51 & 26 \\
51 & 77 & 80 & \boxed{259} & 48 & 207 & 120 & 64 & 33 & 61 \\
74 & 33 & 148 & 63 & \boxed{301} & 72 & 160 & 96 & 30 & 23 \\
44 & 34 & 97 & 163 & 66 & \boxed{324} & 120 & 79 & 49 & 24 \\
16 & 51 & 80 & 93 & 92 & 78 & \boxed{497} & 35 & 16 & 42 \\
49 & 56 & 74 & 74 & 98 & 64 & 68 & \boxed{394} & 42 & 81 \\
138 & 78 & 15 & 26 & 6 & 43 & 9 & 25 & \boxed{566} & 94 \\
60 & 162 & 11 & 38 & 14 & 25 & 49 & 48 & 117 & \boxed{476}
\end{bmatrix} \\
 & \text{y\_pred}
\end{array}
$$

The diagonal elements have the number of correctly predicted values for each class.

So, the accuracy is nothing but the sum of all diagonal elements divided by the total data points (i.e., 10000 since this is test data)

Accuracy = [(469+455+261+259+301+324+497+394+566+476) / 10000] *100

$\approx$ 40 %

6. Implementation of USLA (Step 8 – Step 10)

Step 8: Convolutional AutoEncoder

First, we build a convolutional autoencoder to create layers for compiling and fitting our data using Keras library.

Building an AutoEncoder:

```
# Step 8: Convolutional AutoEncoder

input_img = tf.keras.Input(shape=(32, 32, 3))

x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)
x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)
x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)

x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
decoded = tf.keras.layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = tf.keras.Model(input_img, decoded)
optim = tf.keras.optimizers.SGD(learning_rate=0.1, name='SGD')
autoencoder.summary()
```

Summary of the layers in our autoencoder are as follows,

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 32, 32, 3)]       0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 32, 32, 16)        448
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 16, 16, 16)        0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 16, 16, 8)         1160
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 8, 8, 8)           584
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 4, 4, 8)           0
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 4, 4, 8)           584
_________________________________________________________________
up_sampling2d_9 (UpSampling2 (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 8, 8, 8)           584
_________________________________________________________________
up_sampling2d_10 (UpSampling (None, 16, 16, 8)         0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 16, 16, 16)        1168
_________________________________________________________________
up_sampling2d_11 (UpSampling (None, 32, 32, 16)        0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 32, 32, 3)         435
=================================================================
Total params: 4,963
Trainable params: 4,963
Non-trainable params: 0
```

encoding layers

encoder of output
feature map size 128
(4*4*8)

decoding layers

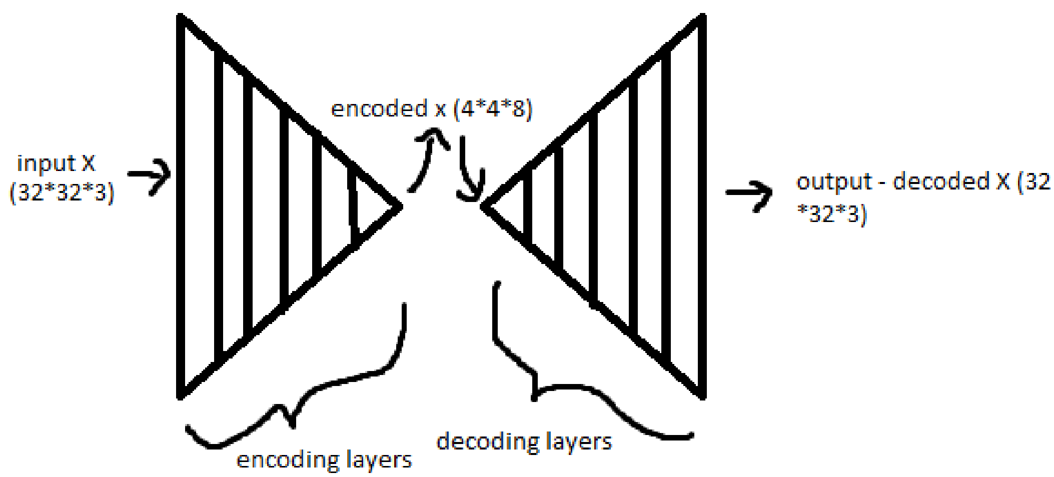
deccoder of output
feature map size 3072
(32*32*3)

Now that we have built the convolutional autoencoder, the next step is to compile and fit our input in the model, thus passing the input features into all the layers and updating weights over the specified epochs within the keras model, so that our model gets trained by self-learning with the transformed features and comparing itself with the input features.

Training the input data

```
autoencoder = tf.keras.Model(input_img, decoded)
optim = tf.keras.optimizers.SGD(learning_rate=0.1, name='SGD')
autoencoder.summary()
autoencoder.compile(optimizer=optim, loss='mse')
autoencoder.fit(x_train, x_train,epochs=25,batch_size=128,shuffle=True,validation_data=(x_test, x_test))
```

- We define the autoencoder as the model that has all the layers we have created, which includes both the encoding and decoding layers.
- Autoencoder is a combined model containing both encoder and decoder, so that the model can encode our features using the encode layer which provides us with an output size of (4*4*8) (that is used for clustering), and then we decode the same features and extract it to an output size as same as the input features (i.e., 32*32*3).
- So, when we get an output features size of 32*32*3, the model compares it with the input features and train the weights accordingly over each epoch within the model.
- All the above are done implicitly within the autoencoder.fit() method and we are provided output with the loss and accuracy of every epoch that runs.



X → encoded → decoded → output

```
autoencoder.fit(x_train, x_train,epochs=25,batch_size=128,shuffle=True,validation_data=(x_test, x_test))
```

The first 10 epochs of the produced output from fitting the data (training) is as follow:

```
Epoch 1/25
391/391 [==============================] - 96s 246ms/step - loss: 0.0627 - val_loss: 0.0613
Epoch 2/25
391/391 [==============================] - 96s 246ms/step - loss: 0.0526 - val_loss: 0.0406
Epoch 3/25
391/391 [==============================] - 100s 256ms/step - loss: 0.0344 - val_loss: 0.0302
Epoch 4/25
391/391 [==============================] - 98s 251ms/step - loss: 0.0302 - val_loss: 0.0297
Epoch 5/25
391/391 [==============================] - 99s 253ms/step - loss: 0.0284 - val_loss: 0.0285
Epoch 6/25
391/391 [==============================] - 100s 255ms/step - loss: 0.0271 - val_loss: 0.0266
Epoch 7/25
391/391 [==============================] - 100s 257ms/step - loss: 0.0262 - val_loss: 0.0279
Epoch 8/25
391/391 [==============================] - 100s 256ms/step - loss: 0.0254 - val_loss: 0.0241
Epoch 9/25
391/391 [==============================] - 100s 255ms/step - loss: 0.0246 - val_loss: 0.0253
Epoch 10/25
391/391 [==============================] - 100s 257ms/step - loss: 0.0242 - val_loss: 0.0237
```

- The optimizer we have used is SGD (Stochastic Gradient Descent)
- The loss we have used is MSE (Mean Squared Error)
- Epochs = 25
- Learning_rate = 0.1
- Batch_size = 128

- So, the next step is K-means Clustering.
- But before we do that, we need to encode our data (the images) using the encoding layer.
- We pass our input X into the encoder layer and fetch the image features for entire dataset from latent layer of auto-encoder (output of encoder).
- Then, we pass the encoded images to the K-Means clustering function to cluster our encoded data (and not the decoded one – the decoder is used only to get our auto-encoded values for training).

Step 9: K-means clustering

- partition *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.
- Number of clusters = number of classes = **10**
- After we get the encoded images from the encoder, we cluster the data using **sklearn.cluster.KMeans**

(I) Getting the encoded images

```
# Fetch the image features for entire dataset from latent layer of auto-encoder

encoder=keras.Model(input_img,encoded)

encoded_imgs=encoder.predict(x_train)
encoded_imgs=encoded_imgs.reshape(x_train.shape[0],128)

encoded_imgs_test=encoder.predict(x_test)
encoded_imgs_test=encoded_imgs_test.reshape(x_test.shape[0],128)
```

(II) Clustering the encoded images

```
# Step 9: Kmeans clustering

kmeans=KMeans(n_clusters=10)
encoded_imgs_fit=kmeans.fit(encoded_imgs)

kmeans_test=KMeans(n_clusters=10)
encoded_imgs_fit_test=kmeans_test.fit(encoded_imgs_test)
```

Step 10: Evaluation USLA

- Once clustering is done, we evaluate our model using the assigned labels (clusters)
- 10 clusters for 10 labels are created and assigned.
- So, the clustering is done on 10 pseudo-labels (pseudo - random numbering of clusters and not the correct labels).
- We find the pseudo-labels and match it according to the true labels with the help of confusion-matrix.
- In the 10x10 confusion matrix we will get once we plot it using the **pseudo_labels** against the **true_labels** (the same way as done in supervised approach), we find the maximum value in a row which corresponds to the true_label (or) the correct class.
- Since the clustering is not efficient and may lead to multiple predicted pseudo_labels belonging to one true_label which causes ambiguity and is not correct, we use linear assignment to find the true clusters.

We do it as follows,

```
labels=encoded_imgs_fit.labels_

confusionmatrix=confusion_matrix(labels,Y)
print(confusionmatrix)

index = linear_assignment(-confusionmatrix)
index = list(index)
index[1] = list(index[1])
true_clusters=index[1]
print("True cluster label order:",true_clusters)
labelmap=[]
for i in range(10):
    labelmap.append(confusionmatrix[i][true_clusters[i]])
```

The marked variable true_clusters will have the correct labels calculated from the confusion matrix plotted using the pseudo_labels against the true_labels.

Now that we have got the true_clusters, we can now find the accuracy using the same.

Finding the accuracy (Train and Test):

- After finding the true clusters, we take the corresponding values from the confusion matrix and store in a label map of size 10 (for 10 classes), which has the number of correctly predicted values for each class.
- Since, we have the number of correctly predicted values for each class, we just simply need to sum it all up and divide by the total data points to find the accuracy.
- In the above figure, the variable labelmap[] has the list of number of correctly predicted value for each class.

Therefore, the accuracy is given by,

```
n = np.sum(confusionmatrix)
accuracy = np.sum(labelmap) / n * 100
```

Since our confusion matrix has all the data points' predictions, we simply take the sum(confusionmatrix) to calculate the total number of data points.

Therefore, Our **Train Accuracy** is:

```
n = np.sum(confusionmatrix)
train_accuracy = np.sum(labelmap) / n * 100
print(train_accuracy)
```

```
True cluster label order: [8, 4, 0, 6, 2, 5, 7, 1, 9, 3]
20.055999999999997
```

And **Test Accuracy** is:

```
n = np.sum(confusionmatrix_test)
test_accuracy = np.sum(labelmap) / n * 100
print(test_accuracy)
```

```
True cluster label order: [6, 2, 7, 8, 5, 0, 3, 4, 9, 1]
20.14
```

Confusion Matrix

After getting the true cluster labels, now we can rearrange the labels and print the confusion matrix of the test data.

```
for i in range(x_test.shape[0]):
    labels_test[i]=true_clusters[labels_test[i]]

confusionmatrix_test_data=confusion_matrix(labels_test,y_test)

print(confusionmatrix_test_data)
```

```
[[217  53  78  61  17  52  44  30  45  48]
 [105 145  70  86  60 142  51  78 158  82]
 [211  56 141 103  72 131  63 104  89  76]
 [ 56 113 126 199 192 170 179 129  66  45]
 [ 71 166 166 104 150  83 169 137  74 148]
 [ 43  30  45 121  70 163  42 103  63  18]
 [ 35 101 159 113 246  81 290  66  16  45]
 [ 74 100 152 144 127 113 120 136  45  59]
 [111 110  27  21  11  21  10  61 266 172]
 [ 77 126  36  48  55  44  32 156 178 307]]
```

7. Code

Code – Raw Text

```
# Step 1: Import Libraries

import tensorflow as tf
import numpy as np
import keras
from keras import layers
from sklearn import preprocessing
from sklearn.metrics import confusion_matrix,accuracy_score
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from scipy.optimize import linear_sum_assignment as linear_assignment


### SUPERVISED LEARNING APPROACH (SLA) ###


# Step 2: Data Loading:

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data
()


# Defining required methods

def minmax(x):
    x_scaled=[]
    for i in range(len(x)):
        x_scaled.append(x[i].flatten())
    scaler = preprocessing.MinMaxScaler()
    x_scaled = scaler.fit_transform(x_scaled)
    return x_scaled

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def softmax(z):
    z=z.T
    for i in range(z.shape[0]):
        z[i] = np.exp(z[i])
        sum_ = sum(z[i])
        z[i] = z[i]/sum_
    return z.T
```

```
def calc_accuracy(p,y):
    p=p.T
    y=y.T
    p_class=[]
    y_class=[]
    for i in range(p.shape[0]):
        p_class.append(np.argmax(p[i]))
        y_class.append(np.argmax(y[i]))
    return accuracy_score(y_class, p_class)*100

def calculate_loss(y,p):
    value = []
    y=y.T
    p=p.T
    for i in range(y.shape[0]):
        val = np.multiply(y[i],np.log(p[i]))
        value.append(sum(val))
    CCE = -(1 / y.shape[0]) *np.sum(value)
    return CCE


# Step 3: Scaling Image Pixel Values

x_train =minmax(x_train)
x_test  =minmax(x_test)

x_train =np.array(x_train).reshape(50000,32,32,3)
x_test  =np.array(x_test).reshape(10000,32,32,3)


# Step 4: One Hot Encoding of target variable

Train_y=[]

for i in range(y_train.shape[0]):
    val = [0 for j in range(10)]
    val[int(y_train[i])]=1
    Train_y.append(val)

y_train=np.asarray(Train_y)

Test_y=[]

for i in range(y_test.shape[0]):
    val = [0 for j in range(10)]
```

```
    val[int(y_test[i])]=1
    Test_y.append(val)

y_test=np.asarray(Test_y)


# Step 5: Initialization of Variables

epochs=700

learning_rate = 0.1

m=x_train.shape[0]

W1 = np.random.randn(300,3072)*0.1
b1 = np.zeros((300,1))
W2 = np.random.randn(10,300)*0.1
b2 = np.zeros((10,1))



# Flattening the features: (32,32,3) --> (3072)
Train_x=[]
for i in range(x_train.shape[0]):
    Train_x.append(x_train[i].flatten())

x_train = np.asarray(Train_x)

Test_x=[]
for i in range(x_test.shape[0]):
    Test_x.append(x_test[i].flatten())

x_test = np.asarray(Test_x)

# Initialising loss_track and accuracy_track variables
losstrack_train = []
losstrack_test = []

train_accuracy = []
test_accuracy = []

x_train=x_train.T
y_train=y_train.T

x_test=x_test.T
y_test=y_test.T
```

```
#Step 6: Training a Neural Network with 1 hidden layer using Gradient Desc
ent Algorithm

for epoch in range(epochs):

    # Step 6.1: Use genesis equation y_pred = SM(W2.(W1X +b1)+b2)
    # train
    Z1 = np.dot(W1,x_train) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(W2,A1) + b2
    A2 = softmax(Z2)
    # test
    Z11 = np.dot(W1,x_test) + b1
    A11 = sigmoid(Z11)
    Z22 = np.dot(W2,A11) + b2
    A22 = softmax(Z22)

    # Step 6.2: Find Categorical Cross Entropy Loss (L) for predicted valu
e A2 and truth value y
    # train
    CCE_train = calculate_loss(y_train,A2)
    losstrack_train.append(np.squeeze(CCE_train))
    #test
    CCE_test = calculate_loss(y_test,A22)
    losstrack_test.append(np.squeeze(CCE_test))

    # Step 6.3: Find dW1, db1, dW2, db2
    dZ2 = A2 - y_train
    dW2 = 1 / m * (np.dot(dZ2,A1.T))
    db2 = 1 / m * (np.sum(dZ2,axis = 1,keepdims = True))
    dZ1 = np.dot(W2.T,dZ2) * (A1*(1-A1))
    dW1 = 1 / m * (np.dot(dZ1,x_train.T))
    db1 = 1 / m * (np.sum(dZ1,axis = 1,keepdims = True))

    # Step 6.4: Update W1, W2, b1, b2 using learning rate
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2


    # Step 6.5: Find Train and Test Accuracy
    train_acc = calc_accuracy(A2,y_train)
    test_acc  = calc_accuracy(A22,y_test)
```

```
    train_accuracy.append(np.squeeze(train_acc))
    test_accuracy.append(np.squeeze(test_acc))


    if epoch%1==0:
        print("epoch : ",epoch+1,"/",epochs)
        print("train_acc = " + str(train_acc) )
        print("test_acc = " + str(test_acc) )
        print("loss = " + str(CCE_train) )
        print("")
        # print("test_loss = " + str(CCE_test) )

# Step 7: Evaluation of SLA by tuning Tune-Hyperparameters

#loss Plot
plt.title('Training and Validation Loss')
plt.plot(losstrack_train,color='blue',label='Training Loss')
plt.plot(losstrack_test,color='orange', label='Testing Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()


#Accuracy Plot
plt.title('Training and Validation Accuracy')
plt.plot(train_accuracy,color='blue',label='Training Accuracy')
plt.plot(test_accuracy,color='orange', label='Testing Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

#Confusion Matrix
Z1_test = np.dot(W1,x_test) + b1
A1_test = sigmoid(Z1_test)
Z2_test = np.dot(W2,A1_test) + b2
A2_test = softmax(Z2_test)

y_test  = np.argmax(y_test,axis=0)
A2_test = np.argmax(A2_test,axis=0)

cm = confusion_matrix(y_test, A2_test)
print(cm)
```

```
### UNSUPERVISED LEARNING APPROACH (USLA) ###
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data
()


# Scaling Features

x_train = x_train / 255
x_test = x_test / 255


# Step 8: Convolutional AutoEncoder

input_img = tf.keras.Input(shape=(32, 32, 3))

x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(
input_img)
x = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)
x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x
)
x = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)
x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x
)
encoded = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)

x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(e
ncoded)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x
)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(
x)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
decoded = tf.keras.layers.Conv2D(3, (3, 3), activation='sigmoid', padding=
'same')(x)

autoencoder = tf.keras.Model(input_img, decoded)
optim = tf.keras.optimizers.SGD(learning_rate=0.3, name='SGD')
autoencoder.summary()
autoencoder.compile(optimizer=optim, loss='mse')
autoencoder.fit(x_train, x_train,epochs=1,batch_size=128,shuffle=True,vali
dation_data=(x_test, x_test))
```

```
# Fetch the image features for entire dataset from latent layer of auto-
encoder

encoder=keras.Model(input_img,encoded)

encoded_imgs=encoder.predict(x_train)
encoded_imgs=encoded_imgs.reshape(x_train.shape[0],128)

encoded_imgs_test=encoder.predict(x_test)
encoded_imgs_test=encoded_imgs_test.reshape(x_test.shape[0],128)

# Step 9: Kmeans clustering

kmeans=KMeans(n_clusters=10)
encoded_imgs_fit=kmeans.fit(encoded_imgs)

kmeans_test=KMeans(n_clusters=10)
encoded_imgs_fit_test=kmeans_test.fit(encoded_imgs_test)

# Step 10: Evalutation USLA

#finding train_accuracy
labels=encoded_imgs_fit.labels_

confusionmatrix=confusion_matrix(labels,y_train)

index = linear_assignment(-confusionmatrix)
index = list(index)
index[1] = list(index[1])
true_clusters=index[1]
print("True cluster label order:",true_clusters)
labelmap=[]
for i in range(10):
    labelmap.append(confusionmatrix[i][true_clusters[i]])

n = np.sum(confusionmatrix)
train_accuracy = np.sum(labelmap) / n * 100
print(train_accuracy)


# finding test_accuracy
labels_test=encoded_imgs_fit_test.labels_

confusionmatrix_test=confusion_matrix(labels_test,y_test)

index = linear_assignment(-confusionmatrix_test)
```

```
index = list(index)
index[1] = list(index[1])
true_clusters=index[1]
print("True cluster label order:",true_clusters)
labelmap=[]
for i in range(10):
    labelmap.append(confusionmatrix_test[i][true_clusters[i]])

n = np.sum(confusionmatrix_test)
test_accuracy = np.sum(labelmap) / n * 100
print(test_accuracy)


#confusion Matrix for test data
for i in range(x_test.shape[0]):
    labels_test[i]=true_clusters[labels_test[i]]

confusionmatrix_test_data=confusion_matrix(labels_test,y_test)

print(confusionmatrix_test_data)
```