American Sign Language (ASL) Pattern Recognition

Introduction

Sign language is an important means of communication for individuals who are deaf or hard of hearing, and developing accurate and reliable sign language recognition systems can have a significant impact on their daily lives.

I started this project with this dataset, https://www.kaggle.com/datasets/nikhilgawai/sign-language-dataset . This dataset has a total of 103 images and 11 classes. Since each image has different size, I first resized all the images to (224,224). And then did preprocessing.

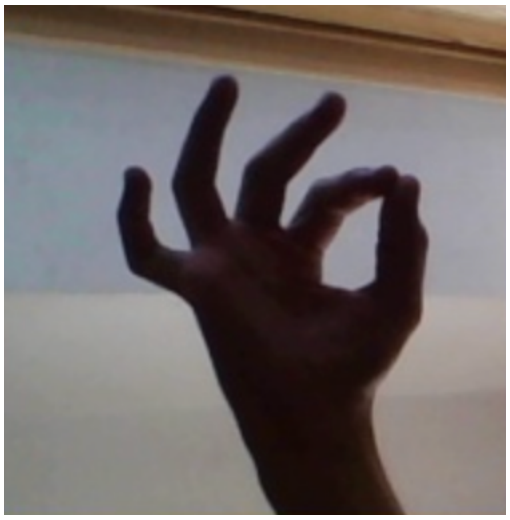
For preprocessing, since this is an image dataset, following steps were required.

- Resizing the Image.
- Convert to gray scale.
- Normalize the images.
- Compress the data/reducing the dimensions.
- When I started working on this dataset**, I realized that encoder decoder needs more images to perform properly. So, I had to change my dataset to a new one**.

Dataset - Modified

https://www.kaggle.com/datasets/grassknoted/asl-alphabet

I am using this dataset. The training data set contains 87,000 images which are 200x200 pixels. There are 29 classes, of which 26 are for the letters A-Z and 3 classes for SPACE, DELETE and NOTHING.



Dimensionality Reduction

- The dataset contains 200X200 size for each image, which makes 40,000 features to represent just one image. To reduce the features of each image we would be using encoder decoder module.

The purpose of encoder decoder is producing the compressed representation of an image or input features this can be very helpful when you have image dataset.

PCA – Limitation

We cannot use PCA for dimensionality reduction in images as:

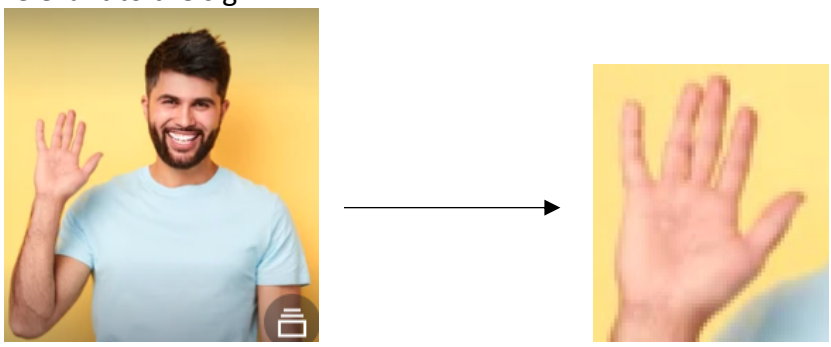
- It assumes that the dataset is linearly separable, which means that there is a linear relationship between the features. But for the image dataset we might have non- linear relationship which PCA won't be able to capture.

- Images have spatial structure since the close pixels are more likely to be related than the far features which is not captured by PCA.

Encoder Decoder Challenges

The problems faced for the construction of encoder decoder are as follows:

1. Since the dataset was very small, for the encoder decoder to work properly instead of giving the same image as input and output to reconstruct, I cropped the images manually to store as target for my dataset (103 images) so that it can learn to extract the features which are most relevant to the sign.



Input Output

But this didn't solve my problem as the dataset was very small for the model to learn such complex things.

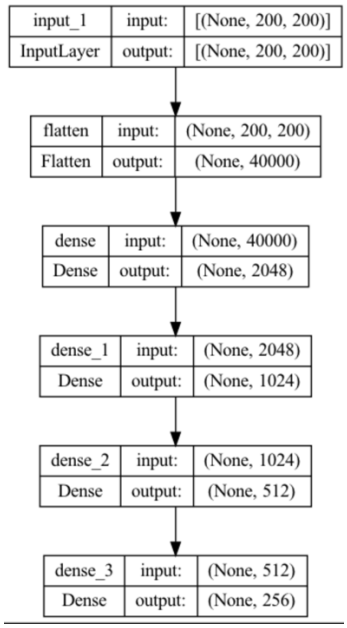
I tried to experiment with the number of neurons to a very big number but still it did not work.

```python
def trainEncoderDecoder(x_train,y_train,x_test,y_test):
    encoder_units = [2048, 1024, 512]
    decoder_units = [ 512, 1024, 2048]
    input_shape = (200, 200)
    output_shape = (200, 200)
    # Define the encoder model
    encoder_input = tf.keras.Input(shape=input_shape)
    x = tf.keras.layers.Flatten()(encoder_input)
    for units in encoder_units:
        x = tf.keras.layers.Dense(units, activation='relu')(x)
    encoder_output = tf.keras.layers.Dense(256, activation='relu')(x)
    encoder_model = tf.keras.Model(encoder_input, encoder_output)

    # Define the decoder model
    decoder_input = tf.keras.Input(shape=(256,))
    x = decoder_input
    for units in decoder_units:
        x = tf.keras.layers.Dense(units, activation='relu')(x)
    x = tf.keras.layers.Dense(tf.reduce_prod(output_shape), activation='relu')(x)
    decoder_output = tf.keras.layers.Reshape(output_shape)(x)
    decoder_model = tf.keras.Model(decoder_input, decoder_output)

    # Define the encoder-decoder model
    autoencoder_input = tf.keras.Input(shape=input_shape)
    encoded = encoder_model(autoencoder_input)
    decoded = decoder_model(encoded)
    autoencoder_model = tf.keras.Model(autoencoder_input, decoded)

    autoencoder_model.compile(optimizer='adam', loss='mse')
```

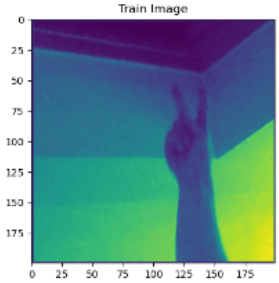
2. I switched my dataset to something which contains hand signs only so that the model might be able to extract those features and generate the original image.

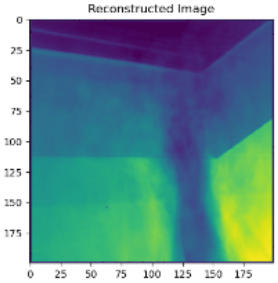
This dataset can be found at this link : https://www.kaggle.com/datasets/grassknoted/asl-alphabet

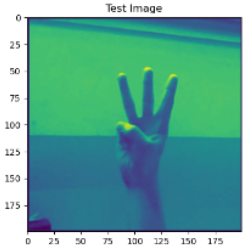
The encoder that I designed initially was by hit and trial, and to extract the nonlinear relationship from data, I tried by checking different set of neurons and experimenting with the number of neurons.

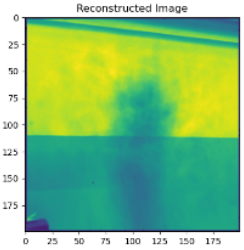


3. When I was training my encoder decoder using this dataset, the problem was that since the dataset was huge, I couldn't train it on whole dataset, so I took 500 images from each folder and trained it only on those images.
    a. The other problem was that since the dataset contains images of hands and the foreground(signs) is dull as compared to background the model was capturing background only and not the foreground.



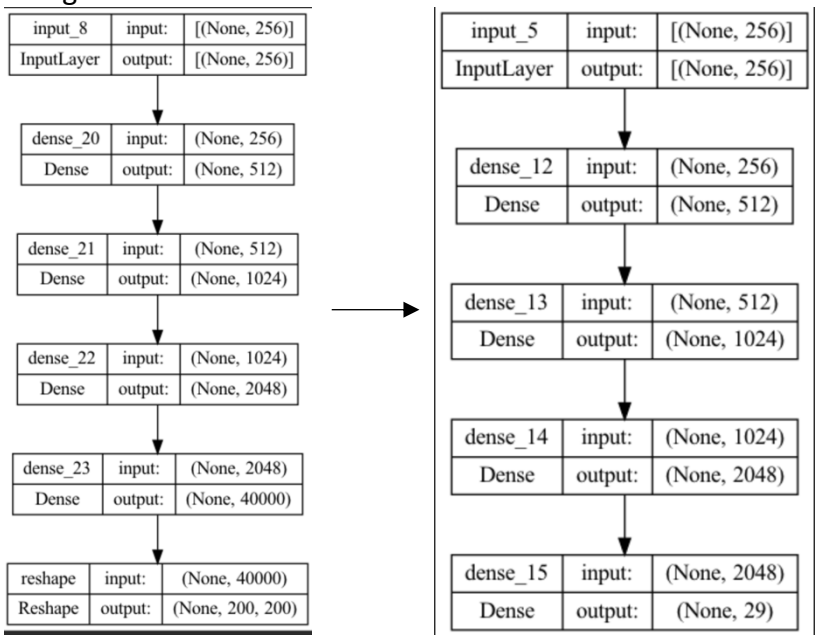




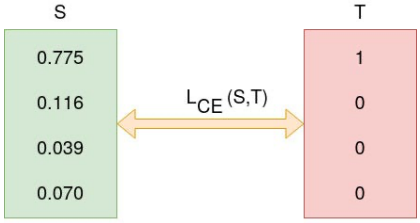


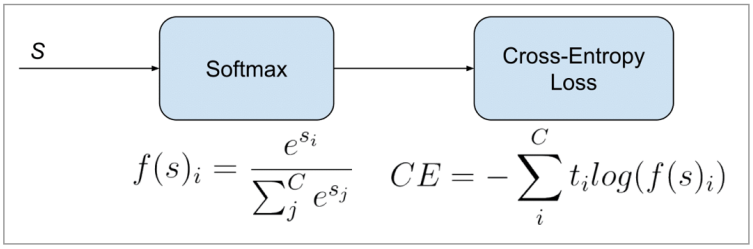
    b. As we can see the signs for A and S appears similar because of lighting issues.

4. To solve the above problem and the encoded representation of image capturing the signs properly, I switched my decoder model to produce the categorical representation of alphabet.
    a. Since I have 29 classes in this dataset: A-Z (26 alphabets), del, space, nothing.
    b. The activation function of last layer would now be SoftMax.
    c. Output would be a vector of 29 elements with each element being the probability of each class.
    d. Categorical loss function is used here since we have 29 classes.



I now am using categorical loss function since I have classes and sigmoid function in last layer. My understanding was that since I am passing classes as output, **the model should capture common pattern(signs) from images of each class and perform better.**

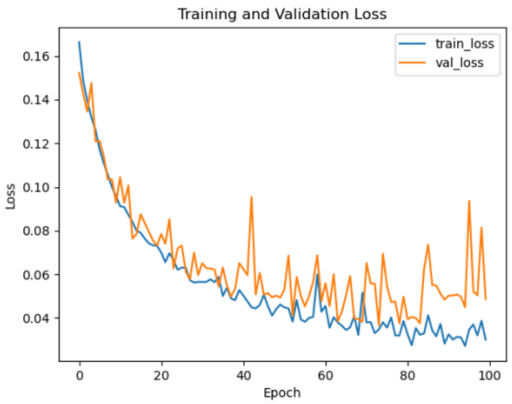




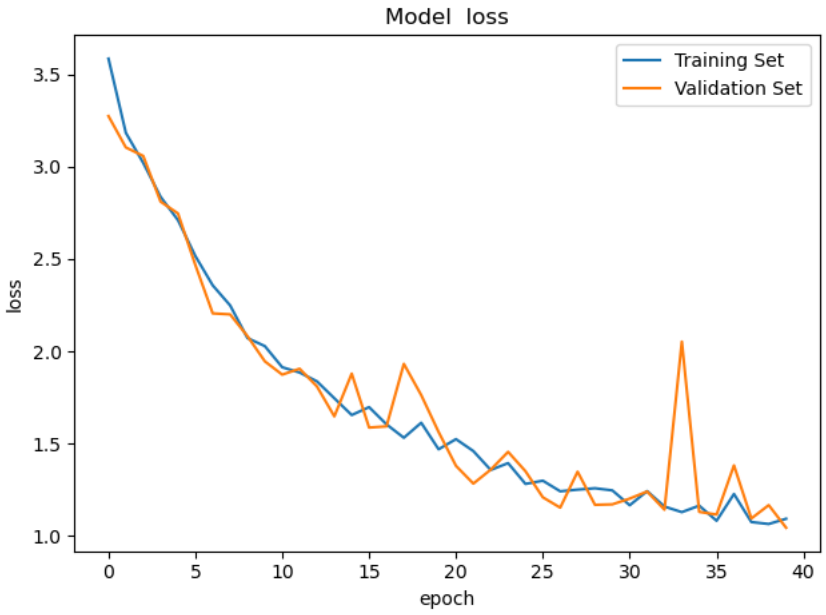
Reference of Image

```python
def trainEncoderDecoderWithTarget(x_train,y_train,x_test,y_test):
    encoder_units = [2048, 1024, 512]
    decoder_units = [ 512, 1024, 2048]
    input_shape = (200, 200)
    #output_shape = (200, 200)
    num_classes = 29
    # Define the encoder model
    encoder_input = tf.keras.Input(shape=input_shape)
    x = tf.keras.layers.Flatten()(encoder_input)
    for units in encoder_units:
        x = tf.keras.layers.Dense(units, activation='relu')(x)
    encoder_output = tf.keras.layers.Dense(256, activation='relu')(x)
    encoder_model = tf.keras.Model(encoder_input, encoder_output)

    # Define the decoder model
    decoder_input = tf.keras.Input(shape=(256,))
    x = decoder_input
    for units in decoder_units:
        x = tf.keras.layers.Dense(units, activation='relu')(x)
    decoder_output = tf.keras.layers.Dense(num_classes, activation='softmax')(x)
    #x = tf.keras.layers.Dense(tf.reduce_prod(num_classes), activation='relu')(x)
    #decoder_output = tf.keras.layers.Reshape((num_classes,))(x)
    #decoder_label_output = tf.keras.layers.Dense(num_classes, activation='softmax')(decoder_output)
    decoder_model = tf.keras.Model(decoder_input, decoder_output)

    # Define the encoder-decoder model
    autoencoder_input = tf.keras.Input(shape=input_shape)
    encoded = encoder_model(autoencoder_input)
    decoded = decoder_model(encoded)
    autoencoder_model = tf.keras.Model(autoencoder_input, decoded)

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

5. After converting this, the model accuracy is still not what I was expecting but I tried to move ahead with this model because of time constraints. As we can see here the loss function is going down.



6. **After the presentation** I made few modifications to my model again. As per my understanding since the bottleneck layer was not able to properly represent my classes, **I increased the number of neurons in my bottleneck and switched it to 512 neurons instead of 256.**




```
Model: "model_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_13 (InputLayer)       [(None, 200, 200)]        0

 flatten_4 (Flatten)         (None, 40000)             0

 dense_32 (Dense)            (None, 2048)              81922048

 dense_33 (Dense)            (None, 1024)              2098176

 dense_34 (Dense)            (None, 700)               717500

 dense_35 (Dense)            (None, 512)               358912

=================================================================
Total params: 85,096,636
Trainable params: 85,096,636
Non-trainable params: 0
_________________________________________________________________
```

This is the encoder model and for the decoder model

```
>>> decoder_model1.summary()
Model: "model_13"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_14 (InputLayer)       [(None, 512)]             0

 dense_36 (Dense)            (None, 700)               359100

 dense_37 (Dense)            (None, 1024)              717824

 dense_38 (Dense)            (None, 2048)              2099200

 dense_39 (Dense)            (None, 29)                59421

=================================================================
Total params: 3,235,545
Trainable params: 3,235,545
Non-trainable params: 0
_________________________________________________________________
```

```python
def trainEncoderDecoderWithTargetIncresedDimesnion(x_train,y_train,x_test,y_test):
    encoder_units = [2048, 1024,700]
    decoder_units = [ 700, 1024,2048]
    input_shape = (200, 200)
    #output_shape = (200, 200)
    num_classes = 29
    # Define the encoder model
    encoder_input = tf.keras.Input(shape=input_shape)
    x = tf.keras.layers.Flatten()(encoder_input)
    for units in encoder_units:
        x = tf.keras.layers.Dense(units, activation='relu')(x)
    encoder_output = tf.keras.layers.Dense(512, activation='relu')(x)
    encoder_model = tf.keras.Model(encoder_input, encoder_output)

    # Define the decoder model
    decoder_input = tf.keras.Input(shape=(512,))
    x = decoder_input
    for units in decoder_units:
        x = tf.keras.layers.Dense(units, activation='relu')(x)
    decoder_output = tf.keras.layers.Dense(num_classes, activation='softmax')(x)
    #x = tf.keras.layers.Dense(tf.reduce_prod(num_classes), activation='relu')(x)
    #decoder_output = tf.keras.layers.Reshape((num_classes,))(x)
    #decoder_label_output = tf.keras.layers.Dense(num_classes, activation='softmax')(decoder_output)
    decoder_model = tf.keras.Model(decoder_input, decoder_output)

    # Define the encoder-decoder model
    autoencoder_input = tf.keras.Input(shape=input_shape)
    encoded = encoder_model(autoencoder_input)
    decoded = decoder_model(encoded)
    autoencoder_model = tf.keras.Model(autoencoder_input, decoded)

    autoencoder_model.compile(optimizer='adam', loss='categorical_crossentropy',metrics=['accuracy'])
    # Train the model
```

Since the features are more here, I think it would perform better, but I think I would continue experimenting with the number of neurons in bottle neck to extract the features from images.

After researching more I found out more ways to extract hands features from an image, but I was not able to use them for my project because of not enough mathematical understanding behind those concepts, the methods that I would be using in future would be:

- Template matching that uses the concept of cross correlation.
- Pose estimator.
- Edge detection and then using those features as well to train my model.

- Also, I found out that having hand segmentation as output is also used to extract hand out of image.

Unfortunately, my encoder decoder model is not working as I was expecting them to be, but I will continue experimenting with the layers and number of neurons in my encoder decoder. The major advantage that I think I have over Deep neural network layers is that there is no constraint on the number of neurons that I have in my layers which provides me with a lot of flexibility.

After I got the encoded representation of image features, I used that encoder model to get the compressed representation of the image. I would be using the encoder used in presentation for further investigation in report since all the diagrams that I have right now are from the previous model.

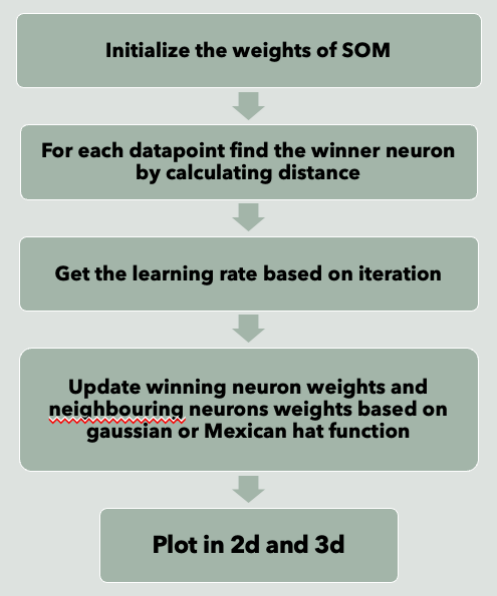
Motivation

The motivation behind using a clustering algorithm for image dataset was to see how SOM would perform with such high image data features. And how the features of similar images would form clusters in space. This kind of representation can be very helpful in performing image recognition, image labelling or grouping similar images together for tasks like recommendation systems, image captioning. And I also wanted to compare the time for image recognition using clustering and classification using deep neural networks.

SOFM (Self Organizing Feature Map)

I tried to implement SOM from scratch. My initial ambition was to get it represent the clusters in 3d, since with high number of classes it is best to have a 3rd dimension as well.

And, I wanted to understand this clustering algorithm in detail, so as to experiment with different distance functions.



The basic principle behind SOM is that as soon as the input vector is presented the weight of the neuron which is closest to input vector should move towards that vector.
So the steps are as follows:

- Initialization : Initializes the neurons which are arranged in grid, the algorithm randomly initializes the values.
- Training :
    - As the set of input vector is presented one at a time
    - The algorithm calculates the distance of input vector from all neurons.
    - The closest neuron is selected as the winning neuron.
    -

```python
def find_winner(self, input_vect):
    """
    Closest neuron to the presented input
    """
    min_dist = np.inf #min distance
    winner = None
    for i in range(self.n_neurons):
        for j in range(self.m_neurons):
            weight_vect = self.weights[i, j, :]
            # distance from input vector
            dist = np.linalg.norm(input_vect - weight_vect)
            #if the distance is less than min distance,
            # than make this neuron as the winner neuron
            if dist < min_dist:
                min_dist = dist
                winner = (i, j)
    return winner
```

- Updating :
    - The SOM updates the weights of the neurons and its neighboring neurons in the grid. The weight vectors of these neurons are adjusted so that they move closer to the input vector.
    - How close they move to the input is regulated by the learning rate.
      So, we will regulate the learning rate so that weight update is stable, we would start with a large learning rate and decrease the learning rate slowly.

$$
\begin{aligned}
{}_i\mathbf{w}(q) &= {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \\
&= (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q)
\end{aligned}
\qquad i \in N_{i^*}(d),
$$

```python
def cal_learning_rate(self, t, max_iter):
    """
    Learning rate update to stabilizes the network towards end of epochs
    is a function of max_teration and current iteration since the learning rate
    should be less than 1
    """
    return self.learning_rate * (1 - t / max_iter)
```

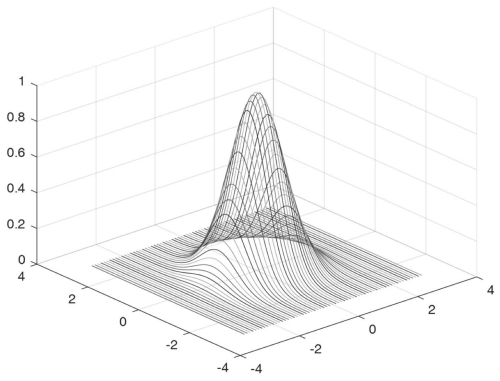
```python
def mexican_hat(self, d, sigma):
    """
    Calculate the value of the Mexican hat function for a given distance d
    and sigma parameter.
    """
    c = (2 / (np.sqrt(3 * sigma) * (np.pi ** 0.25))) #for normalize
    return c*((1 - ((d ** 2) / (sigma ** 2))) * np.exp(-(d ** 2) / (2 * sigma ** 2)))
def _update_weights(self, input_vect, bmu, t, max_iter):
    """
    Update the weights of the BMU and its neighbors
    based on the input_vect and the current iteration number t.
    """
    for i in range(self.n_neurons):
        for j in range(self.m_neurons):
            weight_vect = self.weights[i, j, :]
            dist_to_bmu = self.neighborhood_distance((i, j), bmu)
            if self.neighborhood_function == 'gaussian':
                neighbor_strength = np.exp(-(dist_to_bmu ** 2) / (2 * (self.sigma ** 2)))
            elif self.neighborhood_function == 'mexican_hat':
                neighbor_strength = self.mexican_hat(dist_to_bmu, self.sigma)
            else:
                raise ValueError('Please choose the neighborhood from guassian and mexican_hat')
            learning_rate = self.cal_learning_rate(t, max_iter)
            delta = learning_rate * neighbor_strength * (input_vect - weight_vect)
            self.weights[i, j, :] += delta
```

- The above two steps are iterated continuously until convergence.
- After training the SOM can be visualized as a map of neurons, where each neuron corresponds to a specific region of the input space.

```python
def plot_weights_target(self,targets,data):

    plt.figure(figsize=(16, 16))
    for i, (x, t) in enumerate(zip(data, targets)):
        w = self.find_winner(x)#find_winner
        #print(w)
        plt.text(w[0] + .5, w[1] + .5, str(t), color=plt.cm.tab20(t / 10.), fontdict={'weight': 'bold', 'size': 11})
    plt.axis([0, self.n_neurons, 0, self.m_neurons])
    plt.title('SOM Clusters')
    plt.show()
```

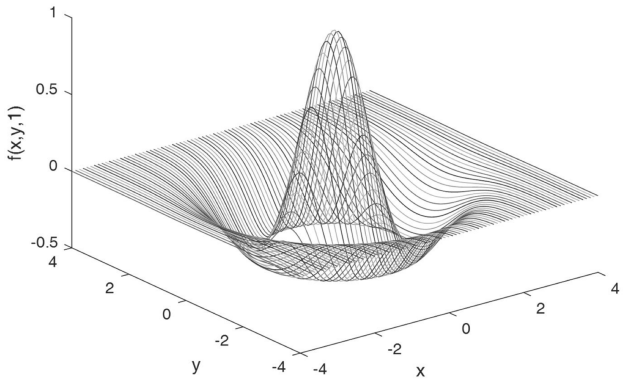
The learning rate is dynamic here, to make the network stable after a point.
https://coursepages2.tuni.fi/tiets07/wp-content/uploads/sites/110/2019/01/Neurocomputing3.pdf

Gaussian Function



.3 A two-dimensional Gaussian function.

The Mexican hat function is used here to update the weights,



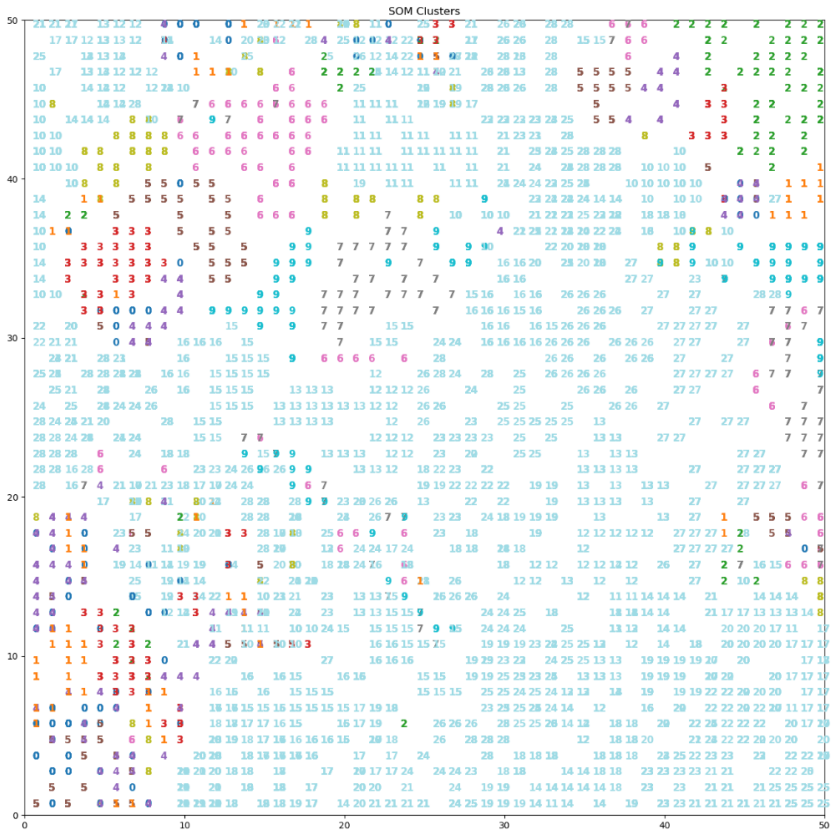
A two-dimensional Mexican hat function.

$$f(\mathbf{x},\mathbf{c},w)=\left(1-\frac{\|\mathbf{x}-\mathbf{c}\|^2}{w}\right)e^{-\frac{\|\mathbf{x}-\mathbf{c}\|^2}{2w}} \qquad (3.6)$$

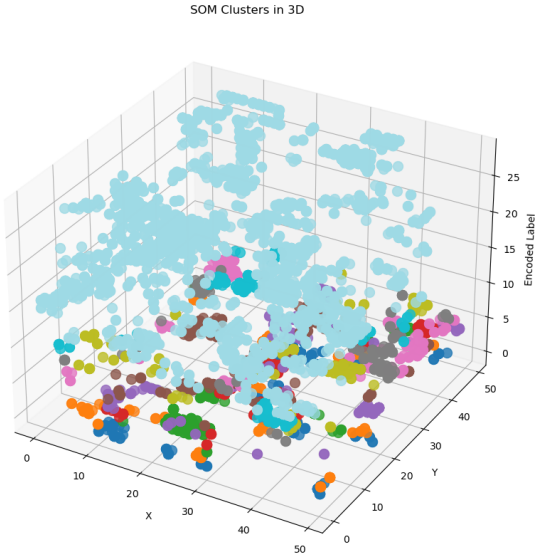
$$\psi(t)=\frac{2}{\sqrt{3\sigma}\pi^{1/4}}\left(1-\left(\frac{t}{\sigma}\right)^2\right)e^{-\frac{t^2}{2\sigma^2}}$$

: $\frac{2}{\sqrt{3\sigma}\pi^{1/4}}$ This part of Mexican hat ensures that the area under curve is 1 and the highest peak is at 1.

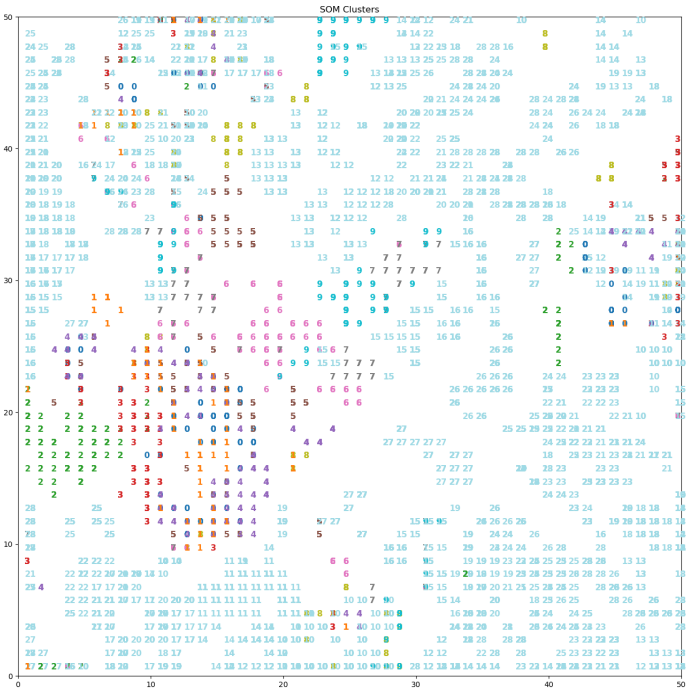
Results from SOFM



Since few of the datapoints are overlapping here, I was interested in getting a 3D structure for neurons grid. However, I couldn't implement a 3D neuron network, hence, I have used targets as the 3rd dimension to visualize the results more clearly.



These are the results from newly encoded data.



As we can see the SOM is able to cluster the encoded representation little bit, but I think it would perform better once my encoder decoder starts performing well.

Future Scope

- The encoder decoder can further be trained with more data and more epochs, and once it's starts working, the encoded representation of the model can be beneficial in a lot of tasks.
- I have also implemented few other parts of SOFM, but they are not working properly so I would work to implement them.
- I tried to perform other models as well on the encoded data but since the representation is not perfect, I still need to improve it. With the current representation, SVM was giving the accuracy of 60% on test data, which is not upto the mark, so I would work on improving it further.
- I think the mathematical algorithms like background subtraction, edge detection, hand posture extraction, segmentation would give me better results as compared to encoder decoder. So, I would research more about those techniques and use them to try the same thing.
- For images, to use multi-layer perceptron, I would like to further preprocess my data using above algorithms, to get the important features and then apply encoder decoder to see the how well the multi-layer perceptron-based network can perform for an image.
- LVQ can be applied to the generated SOFM map for supervised classification.

Limitation

- Because of low processing speed of my laptop, I was not able to train my model on all the images.
- I think the better approach to solve this technique would be to use better technique for extracting the features of hands, I lack the background knowledge of those methods but would definitely try to implement them in future.

Line of code :

References:

1. https://coursepages2.tuni.fi/tiets07/wp-content/uploads/sites/110/2019/01/Neurocomputing3.pdf
2. Brownlee, J. (2022). Display Deep Learning Model Training History in Keras. MachineLearningMastery.com. https://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/
3. Wikipedia contributors. (2022). Ricker wavelet. Wikipedia. https://en.wikipedia.org/wiki/Ricker_wavelet
4. https://d2l.ai/chapter_recurrent-modern/encoder-decoder.html