

Computer Intelligence Minor Project (CS 354)

Project Team Members:

Anmol Gomra (180001007)

Shubham Nimesh(180001054)

Problem Statement

To detect shapes of 2d geometrical shapes using Convolutional Neural Network. The shapes include:

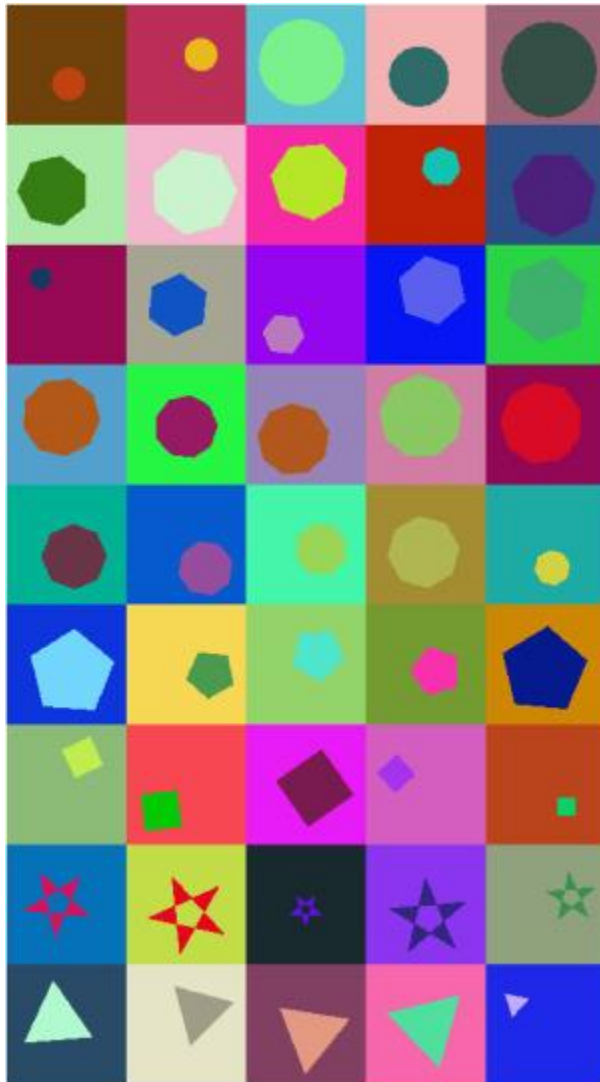
1. Circle
2. Square
3. Star
4. Rectangle
5. Heptagon
6. Hexagon
7. Nonagon
8. Pentagon
9. Triangle

Dataset Description

The dataset is composed of 2D 9 geometric shapes, each shape is drawn randomly on a **200x200** RGB image. During the generation of this dataset, the perimeter and the position of each shape are selected randomly and independently for each image, the rotation angle of each shape is selected randomly for each image within an interval between -180° and 180° , as well the background colour of each image and the filling colour of each shape is selected randomly and independently. The published dataset is composed of 9 data classes, each class represents a type of geometric shape (Triangle, Square, Pentagon, Hexagon, Heptagon, Octagon, Nonagon, Circle and Star). Each class is composed of **10k** generated images. The proposed dataset aims to provide a perfectly clean dataset, for classification as well clustering purposes, the fact that this dataset is generated synthetically provides the ability to use it to study the behaviour of machine learning models independently of the nature of the dataset or the possible noise or data leak that can be found in any other datasets. Moreover, the choice of a 2D geometrical shape dataset provides the ability to understand as well to have good knowledge of the number of patterns stored inside each data class.

Link: <https://www.sciencedirect.com/science/article/pii/S2352340920309847>

What Does the Dataset Look Like ?



Model Structure

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	(None, 20, 20, 16)	2368
batch_normalization (Batch Normalization)	(None, 20, 20, 16)	64
max_pooling2d (MaxPooling2D)	(None, 18, 18, 16)	0
conv2d_1 (Conv2D)	(None, 16, 16, 32)	4640
batch_normalization_1 (Batch Normalization)	(None, 16, 16, 32)	128
max_pooling2d_1 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_2 (Conv2D)	(None, 13, 13, 32)	9248
batch_normalization_2 (Batch Normalization)	(None, 13, 13, 32)	128
max_pooling2d_2 (MaxPooling2D)	(None, 12, 12, 32)	0
conv2d_3 (Conv2D)	(None, 10, 10, 16)	4624
batch_normalization_3 (Batch Normalization)	(None, 10, 10, 16)	64
max_pooling2d_3 (MaxPooling2D)	(None, 9, 9, 16)	0
flatten (Flatten)	(None, 1296)	0
dense (Dense)	(None, 128)	166016
dense_1 (Dense)	(None, 128)	16512

dense_2 (Dense)	(None, 9)	1161
-----------------	-----------	------

=====

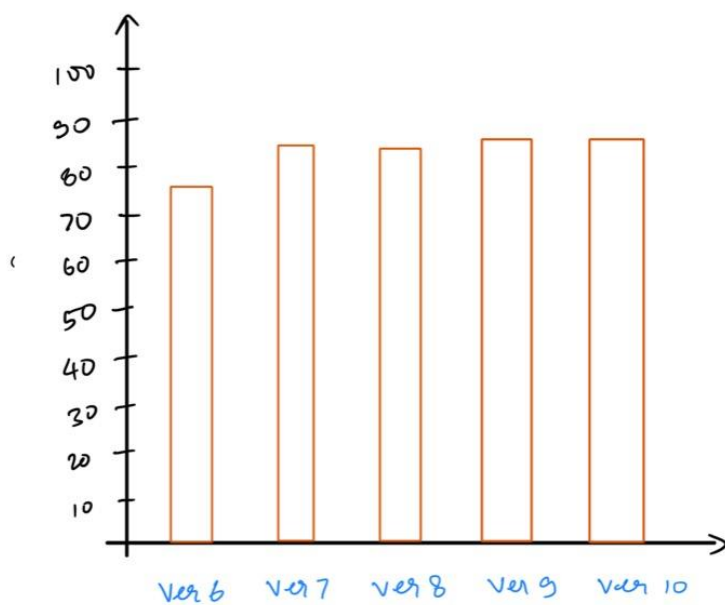
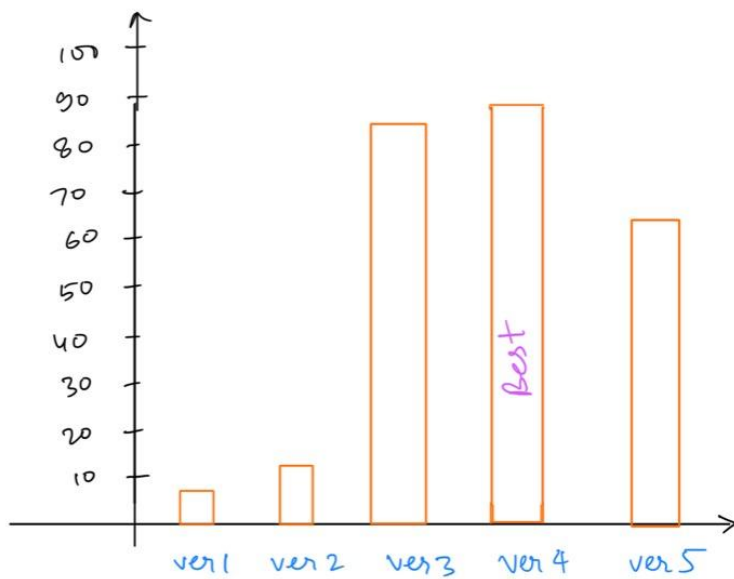
Total params: 204,953

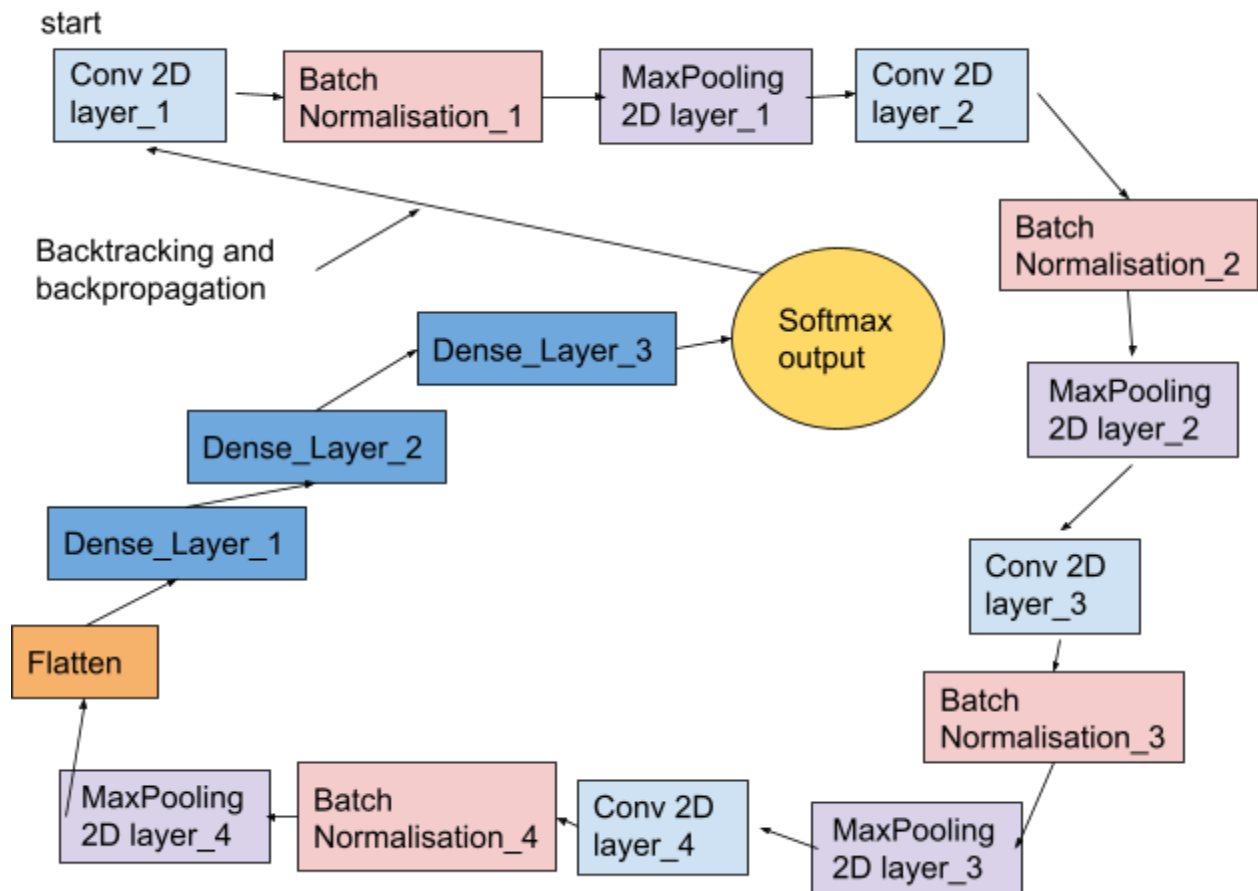
Trainable params: 204,761

Non-trainable params: 192

Accuracy

Categorical Accuracy achieved: **88%**





Layer Roles:

1. **Conv_2D:** Extract important and significant parts of the image. There are 2 main parts to a Conv2D layer:
 - a) In an image we don't need everything and hence we don't need to concentrate on every single detail. There with the help of a feature detector we create a feature map. We use a feature detector of size 7 x 7 and 3 x 3.
 - b) The feature map is then fed to either RELU layer or ELU layer (only one). These are sub parts of the Conv_2D layer. We use 'ELU' as that gives us the best result. It is responsible for removing the negatives from the image so that linearity could be removed.

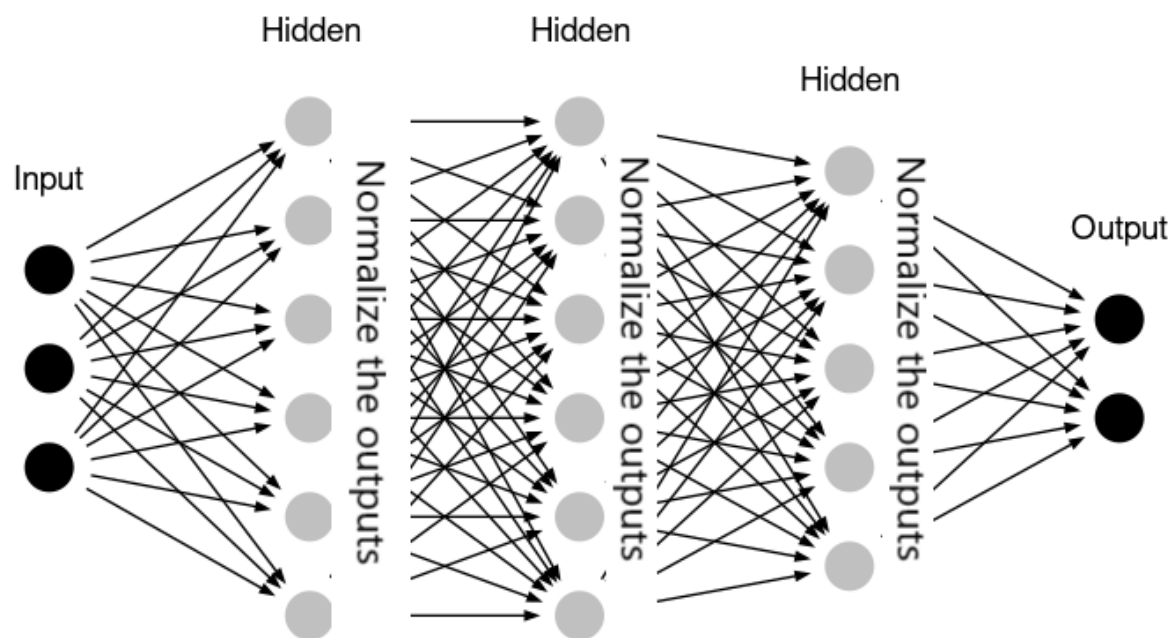
0	0	0	0	0	0	0
0	60	113	56	139	85	0
0	73	121	54	84	128	0
0	131	99	70	129	127	0
0	80	57	115	69	134	0
0	104	126	123	95	130	0
0	0	0	0	0	0	0

Kernel

0	-1	0
-1	5	-1
0	-1	0

114				

- Batch Normalisation Layer:** Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.



3. **MaxPooling Layer:** Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling **layer** would be a feature map containing the most prominent features of the previous feature map. We used a max pool feature detector of size 3 x 3 and 2 x 2.

Feature Map

6	4	8	5
5	4	5	8
3	6	7	7
7	9	7	2

Max-Pooling

4. **Flattening Layer:** This is used to convert 2D matrix generated via all previous steps to single 1d Array which will be fed to Final Dense layers of cnn. (or basically now to an ANN).

1	1	0
4	2	1
0	2	1

1
1
0
4
2
1
0
2
1

5. **Dense Layer:** For dense layers(two 128 units each of neurons) 'elu' is used rather than 'relu' as it gives much better results. For the final output layer (one output layer, 9 units of neuron) 'softmax' activation function is used as it gives results as probabilities. We get how much closer an input image is based on data provided during training. We have multiple inputs therefore we used 'softmax'. If we had 2 inputs we would have used 'sigmoid' which gives output in either 0 / 1 or -1 / 1

Optimiser

'Adam' optimiser is used for improving learning rate over time.

Adam stands for **adaptive moment estimation**.

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data.

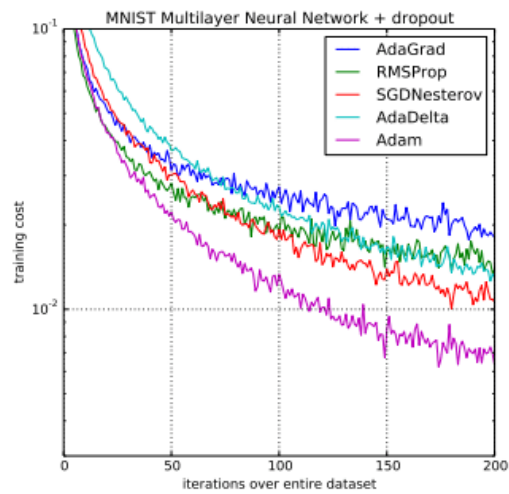
Adam is different to classical stochastic gradient descent.

Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. A learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds.

The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.

- **Adaptive Gradient Algorithm** (AdaGrad) *that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).*
- **Root Mean Square Propagation** (RMSProp) *that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).*

Adam realizes the benefits of both AdaGrad and RMSProp.



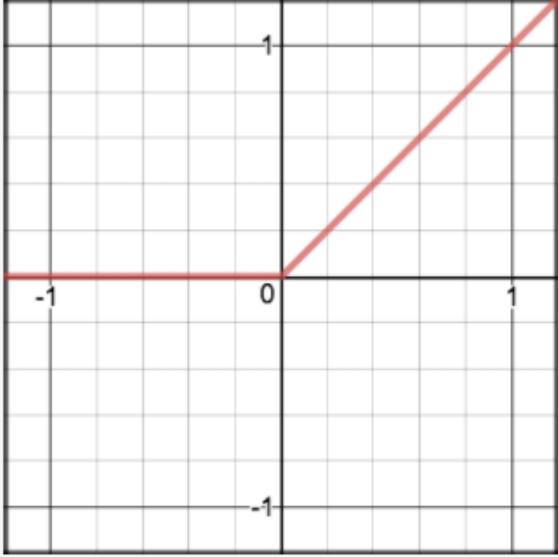
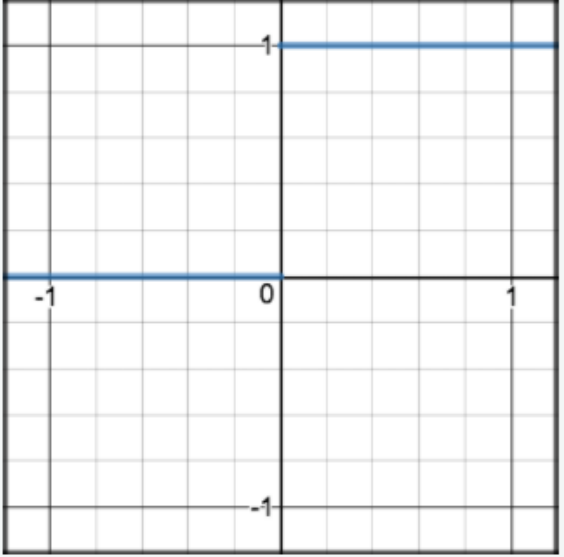
ReLU vs ELU

ReLU

A recent invention which stands for Rectified Linear Units. The formula is deceptively simple:

$\max(0, z)$

$\max(0, z)$. Despite its name and appearance, it's not linear and provides the same benefits as Sigmoid but with better performance.

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$
	
<pre>def relu(z): return max(0, z)</pre>	<pre>def relu_prime(z): return 1 if z > 0 else 0</pre>

Pros

- It avoids and rectifies vanishing gradient problems.
- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

Cons

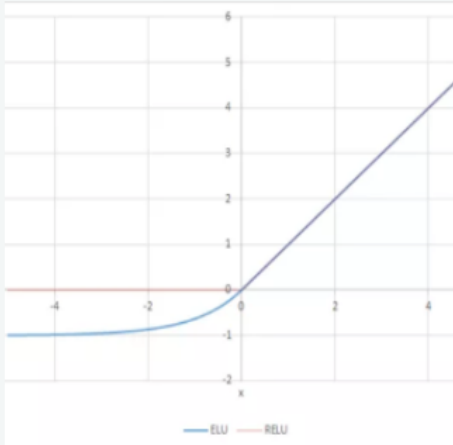
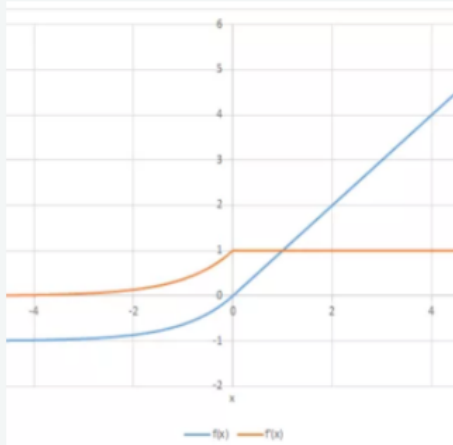
- One of its limitations is that it should only be used within Hidden layers of a Neural Network Model.
- Some gradients can be fragile during training and can die. It can cause a weight update which will make it never activate on any data point again. Simply saying that ReLu could result in Dead Neurons.

- In other words, For activations in the region ($x < 0$) of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called the dying ReLu problem.
- The range of ReLu is $[0, \infty)$. This means it can blow up the activation.

ELU

Exponential Linear Unit or its widely known name ELU is a function that tends to converge cost to zero faster and produce more accurate results. Different from other activation functions, ELU has an extra alpha constant which should be a positive number.

ELU is very similar to RELU except negative inputs. They are both in identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output equals $-\alpha$ whereas RELU sharply smooths.

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ \alpha \cdot (e^z - 1) & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha \cdot e^z & z < 0 \end{cases}$
	
<pre>def elu(z,alpha): return z if z >= 0 else alpha*(e^z -1)</pre>	<pre>def elu_prime(z,alpha): return 1 if z > 0 else alpha*np.exp(z)</pre>

Pros

- ELU becomes smooth slowly until its output equal to $-\alpha$ whereas ReLU sharply smoothes.
- ELU is a strong alternative to ReLU.
- Unlike ReLU, ELU can produce negative outputs.

Cons

- For $x > 0$, it can blow up the activation with the output range of $[0, \infty]$.

References:

- <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning>
- <https://ml-cheatsheet.readthedocs.io/>
- https://s3-us-west-2.amazonaws.com/static.pyimagesearch.com/keras-conv2d/keras_conv2d_padding.gif
- <https://shafeentejani.github.io/assets/images/pooling.gif>

