Ain Shams University

Faculty of Engineering

Mechatronics Engineering Department

CSE473s
Computational Intelligence

Milestone (2):

Name	ID	Section
Bassam Sobhy Abd altawab	180 4313	3
Bavly ehab william	1803239	3
Youssef malak samir	1806173	3

Contents

1. Problem definition and importance

I. Introduction to CNN

Convolutional neural networks present important way in images' features classification.

In this method (as the case of traditional NN) the model is trained by introducing "training" dataset containing considerable number of training images along with their "labels" or their true classes which they already belong to.

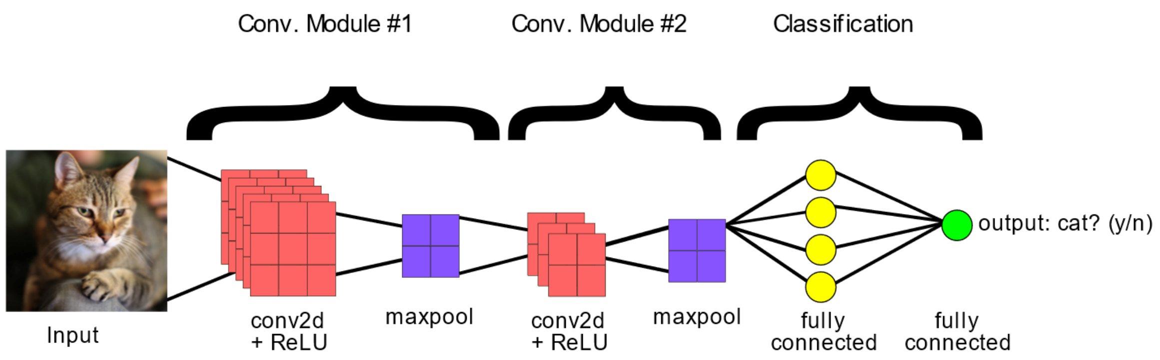
Training takes place when a portion of this dataset (around 80%) is processed in number of epochs where in each epoch, the model builds knowledge of classes and features distinguishing them.

After training stage is done, the rest of the remaining 20% of the dataset is used in validation & testing to make sure that our model is working properly with satisfying accuracy and acceptable losses before letting it deal with the open world

II. CNN vs Traditional neural networks

Traditional NN are pretty good in classification in general; but when it comes to machine vision, an extension is needed to complete its job properly…this extension is number of layers (Net) where kernels are convoluted to the image sequentially to export max out of the image features along with max pooling layers to filter out the non-primary features then the output parameters remain manageable by the NN.

Before neural network starts its job, a feature vector is formed from the flattening stage where all previous stages final output is flattened from 3D matrix into 2D vector can be passed to the classifying deep NN.



2. Algorithms and Methods used

I. Overview about the net

Our CNN consists of 5 convolutional layers and 3 Max pooling layers with the following details

Stage	Input	Output	Parameters
Convolutional_layer_1A	32x32x3	30x30x64	1792
Convolutional_layer_1B	30x30x64	28x28x64	36928
Maxpooling_2D_Layer_1	28x28x64	14x14x64	
Convolutional_layer_2A	14x14x64	12x12x128	73856
Convolutional_layer_2B	12x12x128	10x10x128	147584
Maxpooling_2D_Layer_2	10x10x128	5x5x128	
Convolutional_layer_3A	5x5x128	3x3x256	295168
Maxpooling_2D_Layer_3	3x3x256	1x1x256	
Flatten	1x1x256	256x1	
Hidden_layer_1	256x1	256x1	263168
Hidden_layer_2	256x1	1024x1	524800
Hidden_layer_3	1024x1	512x1	131328
Hidden_layer_4	512x1	256x1	25700
Softmax_layer	256x1	100x1	10100

II. Data normalization

First, we need to calculate batch statistics, in particular, the mean and variance for each of the different activations across a batch. Since each layer's output serves as an input into the next layer in a neural network, by standardizing the output of the layers, we are also standardizing the inputs to the next layer in our model (though in practice, it was suggested in the original paper to implement batch normalization before the activation function, however there's some debate over this).

So, we calculate:

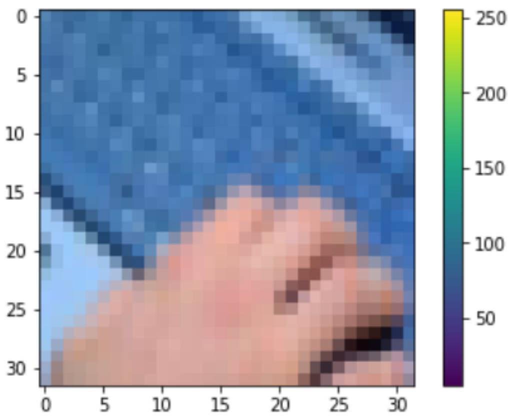
$$\hat{\mu} = \sum_{(n,h,w)\in N*H*W} x_{n,h,w}$$

$$s^2 = \sum_{(n,h,w)\in N*H*W} \left(x_{n,h,w} - \hat{\mu}\right)^2$$

$$x_{n,h,w}^{normalized} = \frac{x_{n,h,w} - \hat{\mu}}{s}$$

Data visualization before normlization

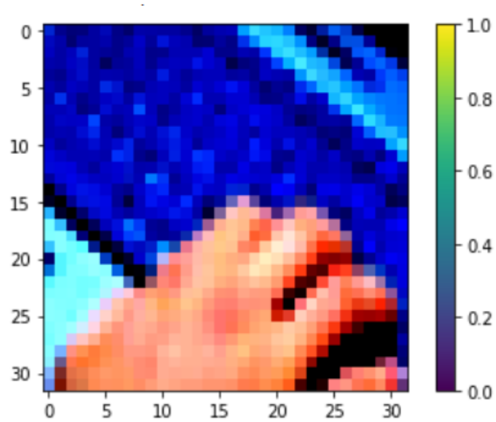
```
index = 10
plt.figure()
plt.imshow(training_images[index])
plt.colorbar()
plt.show()
print(training_labels[index])
```




```
[39]
```

Data visualization after normlization

```
index = 10
plt.figure()
plt.imshow(training_images[index])
plt.colorbar()
plt.show()
print(training_labels[index])
```




```
[39]
```

III. Convolutional layers

i. Input

Input is an image of certain dimensions, in our case in our “Cifar-100” dataset is 32x32 images where this dataset consists of 60,000 images with 100 classes 600 images each. 50,000 images from dataset is already specialized for training while the other 10,000 for testing.

ii. Kernels

Each input image to a convolution layer is subjected to number of kernels (masks) which from their side extract features from this image and they can share their outputs which is good in optimizing the number of needed parameters used to extract these features.

Kernels vary in their sizes 3x3 (our used mask size) or 5x5 but they must have the same depth dimension as that of the input image.

iii. Output

Output of convolution might have a different dimension than the input of the stage, we can describe this output dimensions with the following formulae:

- Calculate the size of conventional layer

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$

$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$

$$D_2 = K$$

IV. Max pooling layers

i. Input

After some convolutions done to an image, pooling process will be beneficial to keep number of parameters as low as possible in order to get the maximum benefit with minimum computations.

ii. Kernels

Our pooling kernel is of size 2x2; notice that there're many types of pooling that can be done in such a layer, we present some of them here :

Max-pooling:

$$h_j^n(x, y) = max_{\bar{x} \in N(x), \bar{y} \in N(y)} h_j^{n-1}(\bar{x}, \bar{y})$$

Average-pooling:

$$h_j^n(x, y) = 1/K \sum_{\bar{x} \in N(x), \bar{y} \in N(y)} h_j^{n-1}(\bar{x}, \bar{y})$$

L2-pooling:

$$h_j^n(x, y) = \sqrt{\sum_{\bar{x} \in N(x), \bar{y} \in N(y)} h_j^{n-1}(\bar{x}, \bar{y})^2}$$

L2-pooling over features:

$$h_j^n(x, y) = \sqrt{\sum_{k \in N(j)} h_k^{n-1}(x, y)^2}$$

V. Flattening layer

i. Input

After number of convolutions and pooling are done on an input image, a resulting 3D array of extracted features are present now; but in order to proceed to the deep NN part we need to have the features arranged in the form of a vector to be able to be passed to the network.

In our case, we have 3x3x256 input array

ii. Output

Output will be 1024x1 vector passed to the deep neural network in order to be classified to one of the 100 classes that we have in our training model.

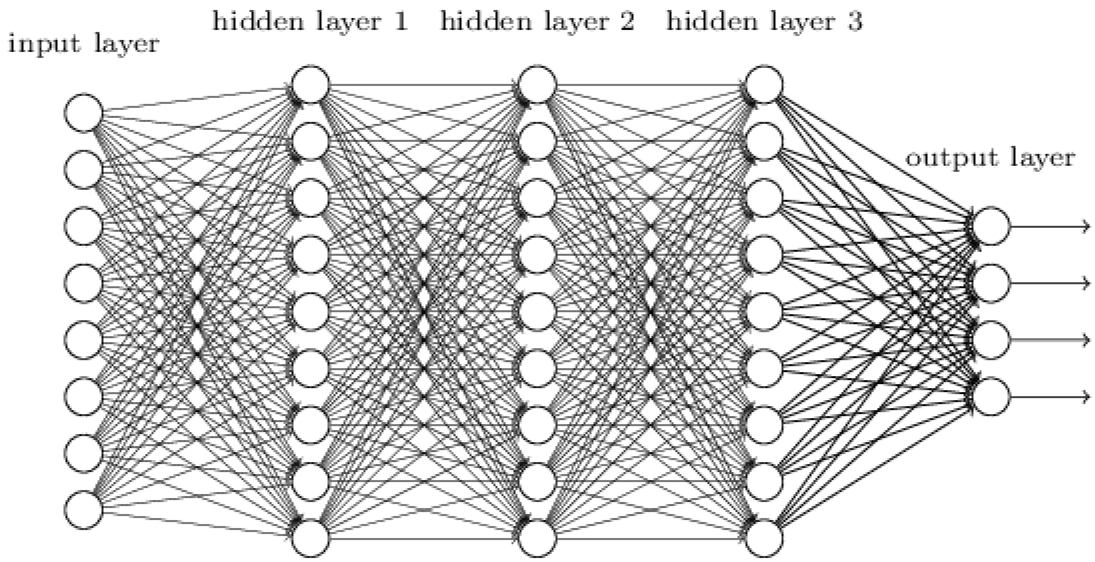
VI. Deep neural network & classification

i. Fully connected layers

In order to enhance the learning process and deepen it, 4 layers of perceptron are used containing perceptron each.

ii. Soft max layer (output_layer)

Probabilistic output is obtained from this layer as a final stage in the classification process where the class with the highest probability is chosen as the classification result.



VII. Cross validation & testing

In this section we're describing the importance of validation and testing in neural networks.

Cross validation is important to avoid overfitting of the model to the training set where it might show good accuracy and low loss during the training but it doesn't classify well with new data as the parameters of the model have been tailored to set of inputs only (the training set).

K-folds cross validation is a way we used where training, validation and also testing are all done in number 'K' of folds, instead of taking each one at once.

3. Trials

i. First CNN

```
[ ] cnn_model = ks.models.Sequential()
    cnn_model.add(ks.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3), name='Convolutional_layer'))

[ ] cnn_model.add(ks.layers.MaxPooling2D((2, 2), name='Maxpooling_2D'))

[ ] cnn_model = ks.models.Sequential()
    cnn_model.add(ks.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3), name='Convolutional_layer'))

[ ] cnn_model.add(ks.layers.MaxPooling2D((2, 2), name='Maxpooling_2D'))

    cnn_model.add(ks.layers.Flatten(name='Flatten'))
    cnn_model.add(ks.layers.Dense(128, activation='relu', name='Hidden_layer_1'))
    cnn_model.add(ks.layers.Dense(64, activation='relu', name='Hidden_layer_2'))
    cnn_model.add(ks.layers.Dense(100, activation='softmax', name='Output_layer'))
                                                  + Code    + Text
```

Figure 1. Model Configuration

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Convolutional_layer (Conv2D  (None, 30, 30, 32)       896
 )

 Maxpooling_2D (MaxPooling2D  (None, 15, 15, 32)       0
 )

 Flatten (Flatten)           (None, 7200)              0

 Hidden_layer_1 (Dense)      (None, 128)               921728

 Hidden_layer_2 (Dense)      (None, 64)                8256

 Output_layer (Dense)        (None, 100)               6500

=================================================================
Total params: 937,380
Trainable params: 937,380
Non-trainable params: 0
```

```
1563/1563 [==============================] - 45s 29ms/step - loss: 0.1909 - accuracy: 0.9551
Epoch 287/300
1563/1563 [==============================] - 44s 28ms/step - loss: 0.1743 - accuracy: 0.9565
Epoch 288/300
1563/1563 [==============================] - 44s 28ms/step - loss: 0.1753 - accuracy: 0.9580
Epoch 289/300
1563/1563 [==============================] - 44s 28ms/step - loss: 0.1684 - accuracy: 0.9578
Epoch 290/300
1563/1563 [==============================] - 43s 28ms/step - loss: 0.1760 - accuracy: 0.9570
Epoch 291/300
1563/1563 [==============================] - 45s 29ms/step - loss: 0.1736 - accuracy: 0.9578
Epoch 292/300
1563/1563 [==============================] - 44s 28ms/step - loss: 0.1766 - accuracy: 0.9568
Epoch 293/300
1563/1563 [==============================] - 44s 28ms/step - loss: 0.1572 - accuracy: 0.9605
Epoch 294/300
1563/1563 [==============================] - 45s 29ms/step - loss: 0.1732 - accuracy: 0.9578
Epoch 295/300
1563/1563 [==============================] - 44s 28ms/step - loss: 0.1848 - accuracy: 0.9566
Epoch 296/300
1563/1563 [==============================] - 45s 29ms/step - loss: 0.1833 - accuracy: 0.9562
Epoch 297/300
1563/1563 [==============================] - 43s 28ms/step - loss: 0.1617 - accuracy: 0.9606
Epoch 298/300
1563/1563 [==============================] - 43s 28ms/step - loss: 0.1758 - accuracy: 0.9583
Epoch 299/300
1563/1563 [==============================] - 44s 28ms/step - loss: 0.1707 - accuracy: 0.9586
Epoch 300/300
1563/1563 [==============================] - 44s 28ms/step - loss: 0.1624 - accuracy: 0.9594
<keras.callbacks.History at 0x7fbd0fbc1610>
```

ii. VGG Net

```
[ ] cnn_model = ks.models.Sequential()

[ ] cnn_model.add(ks.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3), name='Convolutional_layer_1A'))
    cnn_model.add(ks.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(30, 30, 64), name='Convolutional_layer_1B'))

[ ] cnn_model.add(ks.layers.MaxPooling2D((2, 2),input_shape=(28,28,64), name='Maxpooling_2D_Layer_1'))

[ ] #cnn_model = ks.models.Sequential()
    cnn_model.add(ks.layers.Conv2D(128, (3, 3), activation='relu', input_shape=(13, 13, 64), name='Convolutional_layer_2A'))
    cnn_model.add(ks.layers.Conv2D(128, (3, 3), activation='relu', input_shape=(11, 11, 128), name='Convolutional_layer_2B'))

[ ] cnn_model.add(ks.layers.MaxPooling2D((2, 2),input_shape=(9,9,128), name='Maxpooling_2D_Layer_2'))

[ ] #cnn_model = ks.models.Sequential()
    cnn_model.add(ks.layers.Conv2D(256, (3, 3), activation='relu', input_shape=(5, 5, 256), name='Convolutional_layer_3A'))

[ ] cnn_model.add(ks.layers.MaxPooling2D((2, 2),input_shape=(3,3,256), name='Maxpooling_2D_Layer_3'))

[ ] cnn_model.add(ks.layers.Flatten(name='Flatten'))

[ ]
    cnn_model.add(ks.layers.Dense(256, activation='relu', name='Hidden_layer_1'))

    cnn_model.add(ks.layers.Dense(1024, activation='relu', name='Hidden_layer_2'))

    cnn_model.add(ks.layers.Dense(512, activation='relu', name='Hidden_layer_3'))

    cnn_model.add(ks.layers.Dense(256, activation='relu', name='Hidden_layer_4'))

    cnn_model.add(ks.layers.Dense(100, activation='softmax', name='Output_layer'))
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Convolutional_layer_1A (Con  (None, 30, 30, 64)       1792
 v2D)

 Convolutional_layer_1B (Con  (None, 28, 28, 64)       36928
 v2D)

 Maxpooling_2D_Layer_1 (MaxP  (None, 14, 14, 64)       0
 ooling2D)

 Convolutional_layer_2A (Con  (None, 12, 12, 128)      73856
 v2D)

 Convolutional_layer_2B (Con  (None, 10, 10, 128)      147584
 v2D)

 Maxpooling_2D_Layer_2 (MaxP  (None, 5, 5, 128)        0
 ooling2D)

 Convolutional_layer_3A (Con  (None, 3, 3, 256)        295168
 v2D)

 Maxpooling_2D_Layer_3 (MaxP  (None, 1, 1, 256)        0
 ooling2D)

 Flatten (Flatten)           (None, 256)               0
```

```
[ ]
500/500 [==============================] - 7s 14ms/step - loss: 0.2863 - accuracy: 0.9196
Epoch 41/50
500/500 [==============================] - 7s 14ms/step - loss: 0.3233 - accuracy: 0.9110
Epoch 42/50
500/500 [==============================] - 7s 14ms/step - loss: 0.3274 - accuracy: 0.9086
Epoch 43/50
500/500 [==============================] - 7s 14ms/step - loss: 0.3124 - accuracy: 0.9135
Epoch 44/50
500/500 [==============================] - 7s 14ms/step - loss: 0.2925 - accuracy: 0.9194
Epoch 45/50
500/500 [==============================] - 7s 14ms/step - loss: 0.3124 - accuracy: 0.9145
Epoch 46/50
500/500 [==============================] - 7s 14ms/step - loss: 0.2961 - accuracy: 0.9164
Epoch 47/50
500/500 [==============================] - 7s 14ms/step - loss: 0.3088 - accuracy: 0.9124
Epoch 48/50
500/500 [==============================] - 7s 14ms/step - loss: 0.3085 - accuracy: 0.9134
Epoch 49/50
500/500 [==============================] - 7s 14ms/step - loss: 0.3202 - accuracy: 0.9118
Epoch 50/50
500/500 [==============================] - 7s 14ms/step - loss: 0.2879 - accuracy: 0.9184
```

Figure 2. Results showing accuracy & loss

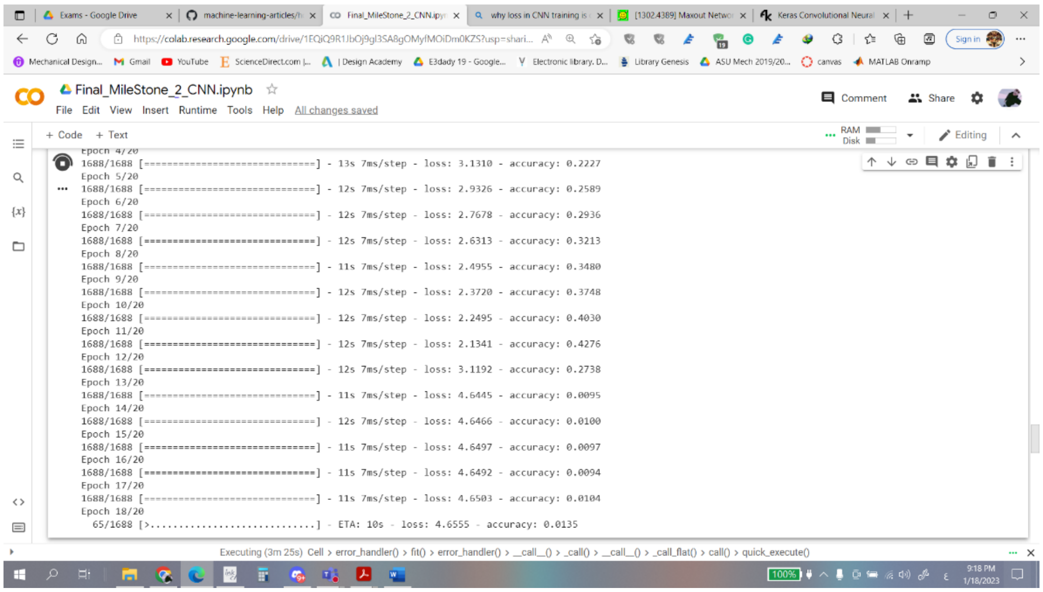
iii. VGG Net with cross validation

```
#VGGNet_Approach

cnn_model.add(ks.layers.Conv2D(64, (3, 3), activation='elu', input_shape=(32, 32, 3), name='Convolutional_layer_1A'))
cnn_model.add(ks.layers.Conv2D(64, (3, 3), activation='elu', input_shape=(30, 30, 64), name='Convolutional_layer_1B'))

cnn_model.add(ks.layers.MaxPooling2D((2, 2),input_shape=(28,28,64), name='Maxpooling_2D_Layer_1'))
#Drop out unneeded data to avoid overfitiing
tf.keras.layers.Dropout( rate = 0.1)
cnn_model.add(ks.layers.Conv2D(128, (3, 3), activation='elu', input_shape=(13, 13, 64), name='Convolutional_layer_2A'))
cnn_model.add(ks.layers.Conv2D(128, (3, 3), activation='elu', input_shape=(11, 11, 128), name='Convolutional_layer_2B'))

cnn_model.add(ks.layers.MaxPooling2D((2, 2),input_shape=(9,9,128), name='Maxpooling_2D_Layer_2'))

#Drop out unneeded data to avoid overfitiing
tf.keras.layers.Dropout( rate = 0.25)
cnn_model.add(ks.layers.Conv2D(256, (3, 3), activation='elu', input_shape=(5, 5, 256), name='Convolutional_layer_3A'))

cnn_model.add(ks.layers.MaxPooling2D((2, 2),input_shape=(3,3,256), name='Maxpooling_2D_Layer_3'))

#Drop out unneeded data to avoid overfitiing
tf.keras.layers.Dropout( rate = 0.5)

cnn_model.add(ks.layers.Flatten(name='Flatten'))
```

```
##################################################################################
cnn_model.add(ks.layers.Dense(1024, activation='elu', name='Hidden_layer_1'))
#Drop out unneeded nodes to avoid overfitiing
#tf.keras.layers.Dropout( rate = 0.2, seed=2 )
cnn_model.add(ks.layers.Dense(512, activation='elu', name='Hidden_layer_2'))
#Drop out unneeded nodes to avoid overfitiing
#tf.keras.layers.Dropout( rate = 0.2, seed=2 )
cnn_model.add(ks.layers.Dense(512, activation='elu', name='Hidden_layer_3'))
#Drop out unneeded nodes to avoid overfitiing
#tf.keras.layers.Dropout( rate = 0.2, seed=2 )
cnn_model.add(ks.layers.Dense(100, activation='elu', name='Hidden_layer_4'))
##################################################################################
```

4. Trials and Problems

VGG_Net_KFold_1st_trial



With 6 hidden layers and 512 nodes and make normalization between hidden layers to training images

He makes an over fitting at high no of epochs.

This output occurs at 20 epochs, and we monitor, overfitting is happening.

```
Epoch 2/300
750/750 [==============================] - 7s 10ms/step - loss: 4.6055 - accuracy: 0.0099
Epoch 3/300
750/750 [==============================] - 7s 10ms/step - loss: 4.6055 - accuracy: 0.0096
Epoch 4/300
750/750 [==============================] - 7s 10ms/step - loss: 4.6055 - accuracy: 0.0096
Epoch 5/300
750/750 [==============================] - 7s 10ms/step - loss: 4.6055 - accuracy: 0.0100
Epoch 6/300
750/750 [==============================] - 8s 11ms/step - loss: 4.6055 - accuracy: 0.0091
Epoch 7/300
750/750 [==============================] - 8s 10ms/step - loss: 4.6055 - accuracy: 0.0101
Epoch 8/300
750/750 [==============================] - 8s 10ms/step - loss: 4.6055 - accuracy: 0.0095
Epoch 9/300
750/750 [==============================] - 8s 10ms/step - loss: 4.6055 - accuracy: 0.0099
Epoch 10/300
750/750 [==============================] - 8s 10ms/step - loss: 4.6055 - accuracy: 0.0101
Epoch 11/300
750/750 [==============================] - 7s 10ms/step - loss: 4.6055 - accuracy: 0.0101
Epoch 12/300
750/750 [==============================] - 7s 10ms/step - loss: 4.6055 - accuracy: 0.0098
Epoch 13/300
750/750 [==============================] - 7s 10ms/step - loss: 4.6055 - accuracy: 0.0099
Epoch 14/300
750/750 [==============================] - 8s 10ms/step - loss: 4.6055 - accuracy: 0.0096
Epoch 15/300
750/750 [==============================] - 7s 10ms/step - loss: 4.6055 - accuracy: 0.0091
Epoch 16/300
750/750 [==============================] - 8s 10ms/step - loss: 4.6055 - accuracy: 0.0104
Epoch 17/300
750/750 [==============================] - 8s 10ms/step - loss: 4.6055 - accuracy: 0.0097
Epoch 18/300
750/750 [==============================] - 8s 10ms/step - loss: 4.6055 - accuracy: 0.0097
```

Also, at trial from trials we note loss is constant and by search we know the problem which is gradient of data after normalization is nearly to zero so no optimization will occur and far from the range of activation function.

From solution to avoid this problem is changing the activation function and we change ReLU activation function to exponential ReLU 'elu'.

VGG_Net_KFold_2nd_trial

```
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 3.308101177215576 - Accuracy: 31.48333430290222%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 2 - Loss: 3.30979323387146 - Accuracy: 33.41666758060455%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 3 - Loss: 4.646727085113525 - Accuracy: 1.0833333246409893%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 4 - Loss: 4.637165546417236 - Accuracy: 1.2333333492279053%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 5 - Loss: 4.676224231719971 - Accuracy: 0.8500000461935997%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 6 - Loss: 3.2374491691589355 - Accuracy: 32.98333287239075%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 7 - Loss: 3.1567890644073486 - Accuracy: 33.283331990242004%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 8 - Loss: 4.236500263214111 - Accuracy: 5.316666513681412%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 9 - Loss: 4.64518404006958 - Accuracy: 0.983333308249712%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 10 - Loss: 3.2801849842071533 - Accuracy: 32.16666579246521%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 17.279999908059835 (+- 15.443173172237424)
> Loss: 3.9134118795394897
------------------------------------------------------------------------
```

```
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
Convolutional_layer_1A (Con  (None, 30, 30, 64)       1792
v2D)

Convolutional_layer_1B (Con  (None, 28, 28, 64)       36928
v2D)

Maxpooling_2D_Layer_1 (MaxP  (None, 14, 14, 64)       0
ooling2D)

Convolutional_layer_2A (Con  (None, 12, 12, 128)      73856
v2D)

Convolutional_layer_2B (Con  (None, 10, 10, 128)      147584
v2D)

Maxpooling_2D_Layer_2 (MaxP  (None, 5, 5, 128)        0
ooling2D)

Convolutional_layer_3A (Con  (None, 3, 3, 256)        295168
v2D)

Maxpooling_2D_Layer_3 (MaxP  (None, 1, 1, 256)        0
ooling2D)

Flatten (Flatten)           (None, 256)               0

Hidden_layer_1 (Dense)      (None, 1024)              263168

Hidden_layer_2 (Dense)      (None, 512)               524800

Hidden_layer_3 (Dense)      (None, 512)               262656

Hidden_layer_4 (Dense)      (None, 100)               51300

Output_layer (Dense)        (None, 100)               10100

=================================================================
Total params: 1,667,352
Trainable params: 1,667,352
Non-trainable params: 0
_________________________________________________________________
```

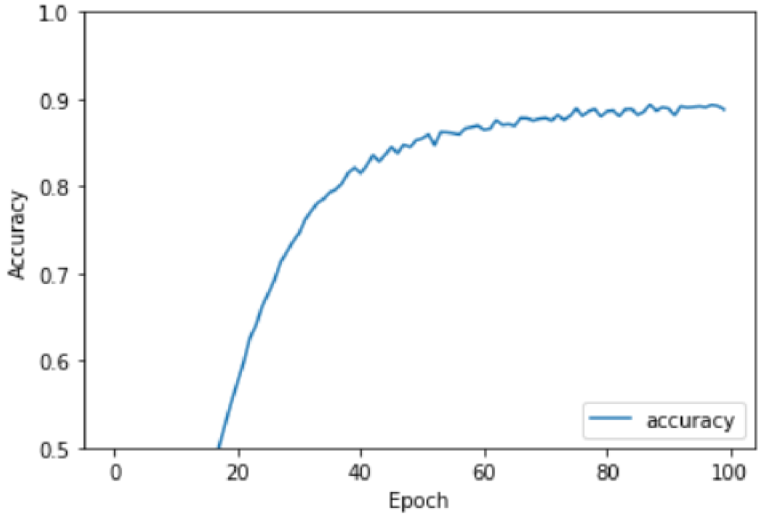
VGG_Net_KFold_3rd_trial

After updating the layers

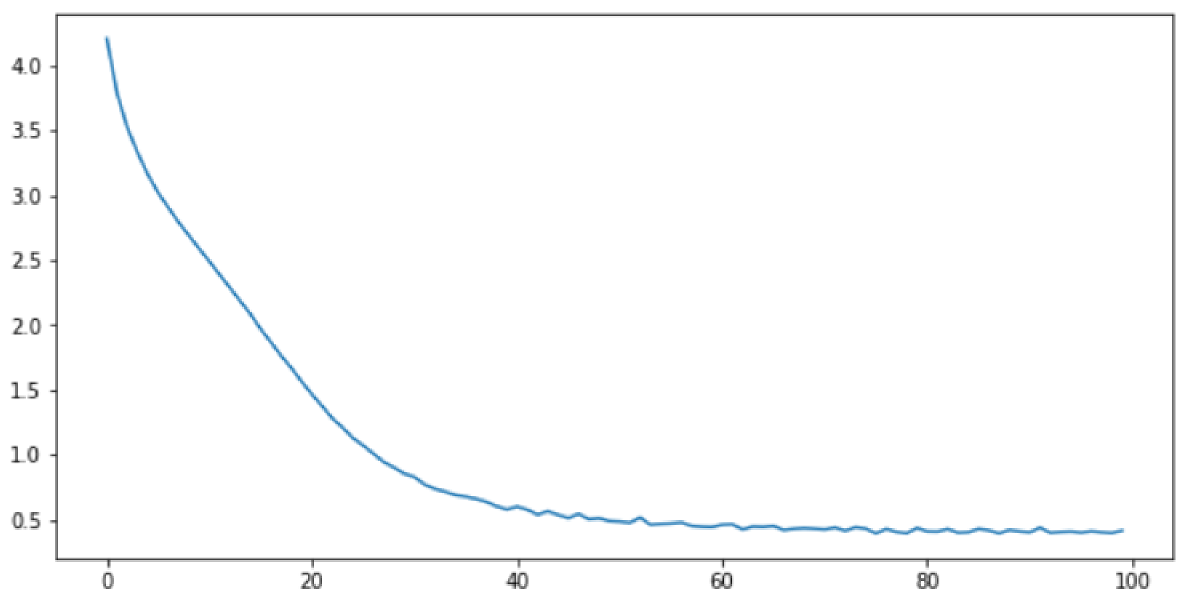
```
fold_no = 1
for train, test in kfold.split(inputs, targets):

  cnn_model = ks.models.Sequential()

  cnn_model.add(ks.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3), name='Convolutional_layer_1A'))
  cnn_model.add(ks.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(30, 30, 64), name='Convolutional_layer_1B'))

  cnn_model.add(ks.layers.MaxPooling2D((2, 2),input_shape=(28,28,64), name='Maxpooling_2D_Layer_1'))

  cnn_model.add(ks.layers.Conv2D(128, (3, 3), activation='relu', input_shape=(13, 13, 64), name='Convolutional_layer_2A'))
  cnn_model.add(ks.layers.Conv2D(128, (3, 3), activation='relu', input_shape=(11, 11, 128), name='Convolutional_layer_2B'))

  cnn_model.add(ks.layers.MaxPooling2D((2, 2),input_shape=(9,9,128), name='Maxpooling_2D_Layer_2'))

  cnn_model.add(ks.layers.Conv2D(256, (3, 3), activation='relu', input_shape=(5, 5, 256), name='Convolutional_layer_3A'))

  cnn_model.add(ks.layers.MaxPooling2D((2, 2),input_shape=(3,3,256), name='Maxpooling_2D_Layer_3'))

  cnn_model.add(ks.layers.Flatten(name='Flatten'))
  #normlization

  ################################
  cnn_model.add(ks.layers.Dense(1024, activation='relu', name='Hidden_layer_1'))
  cnn_model.add(ks.layers.Dense(1024, activation='relu', name='Hidden_layer_2'))
  cnn_model.add(ks.layers.Dense(100, activation='softmax', name='Output_layer'))
```

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 11.419368743896484 - Accuracy: 26.11333429813385%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 18.728333967737854 (+- 10.42481094194868)
> Loss: 9.662007570266724
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 2 - Loss: 4.606935501098633 - Accuracy: 0.8200000040233135%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 18.728333967737854 (+- 10.42481094194868)
> Loss: 9.662007570266724
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 3 - Loss: 11.029722213745117 - Accuracy: 25.42000114917755%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 18.728333967737854 (+- 10.42481094194868)
> Loss: 9.662007570266724
------------------------------------------------------------------------
------------------------------------------------------------------------
> Fold 4 - Loss: 11.59200382232666 - Accuracy: 22.5600004196167%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 18.728333967737854 (+- 10.42481094194868)
> Loss: 9.662007570266724
------------------------------------------------------------------------
```

```
Average scores for all folds:
> Accuracy: 18.728333967737854 (+- 10.42481094194868)
> Loss: 9.662007570266724
------------------------------------------------------------------------
<matplotlib.legend.Legend at 0x7f81c02a59a0>
```



- Loss Vs no_epochs



5. Appendix

You can refer to Full Code:

- Co-lab code:

  https://colab.research.google.com/drive/1EQiQ9R1JbOj9gl3SA8gOMyfMOiDm0KZS?usp=sharing