# Deep Learning

This assignment aims to implement deep learning for image recognition specifically on the MNIST digit dataset with 64000 training data and 100000 testing data. The assignment is divided into following tasks –

Training -> Testing -> Examine the network -> Transfer learning -> Hyperparameter optimization.

## A. Training & Testing

Training is done on the 64000 MNIST hand drawn images. The Neural Network used is of the following architecture –

- A convolution layer with 10 5x5 filters.
- A max pooling layer with a 2x2 window and a ReLU function applied.
- A convolution layer with 20 5x5 filters.
- A dropout layer with a 0.5 dropout rate (50%)
- A max pooling layer with a 2x2 window and a ReLU function applied.
- A flattening operation followed by a fully connected Linear layer with 50 nodes and a ReLU function on the output.
- A final fully connected Linear layer with 10 nodes and the log_softmax function applied to the output.

Following are the outputs required from training task –

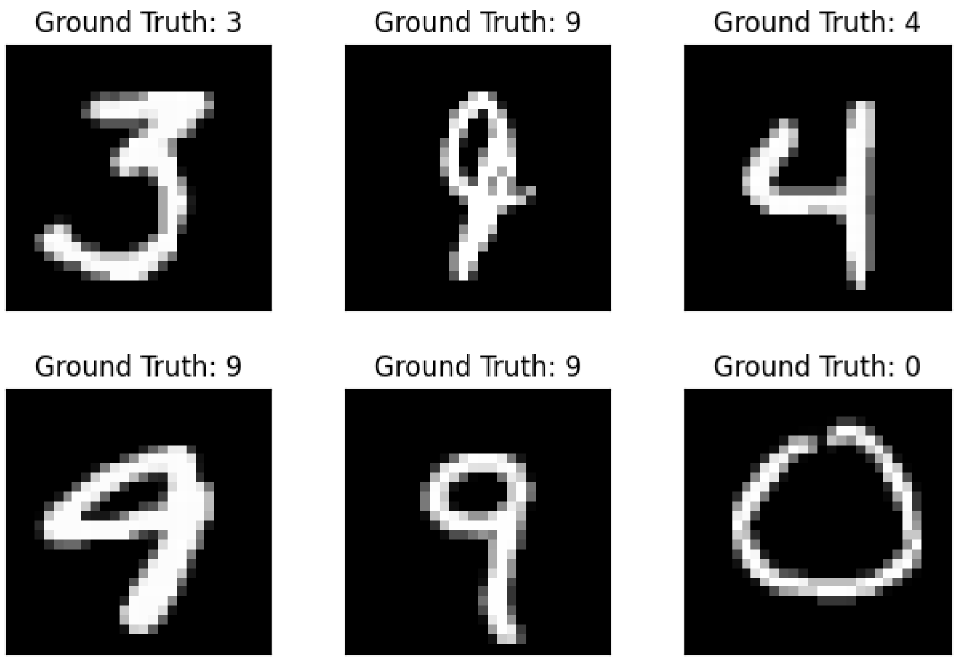
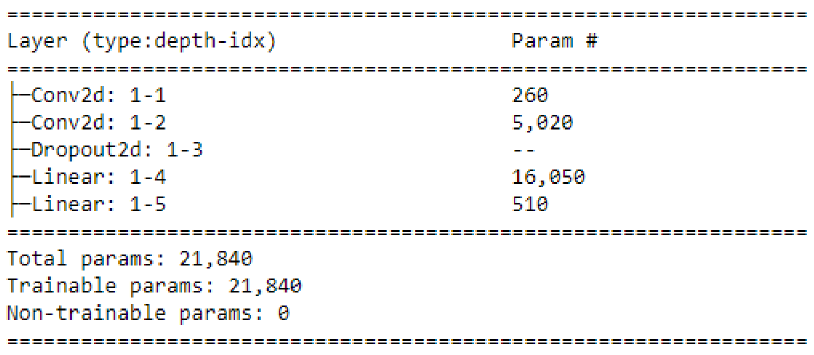


Fig 1. – First 6 examples of the hand drawn digits.



```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            260
├─Conv2d: 1-2                            5,020
├─Dropout2d: 1-3                         --
├─Linear: 1-4                            16,050
├─Linear: 1-5                            510
=================================================================
Total params: 21,840
Trainable params: 21,840
Non-trainable params: 0
=================================================================
```

Fig 2. – Neural Network architecture

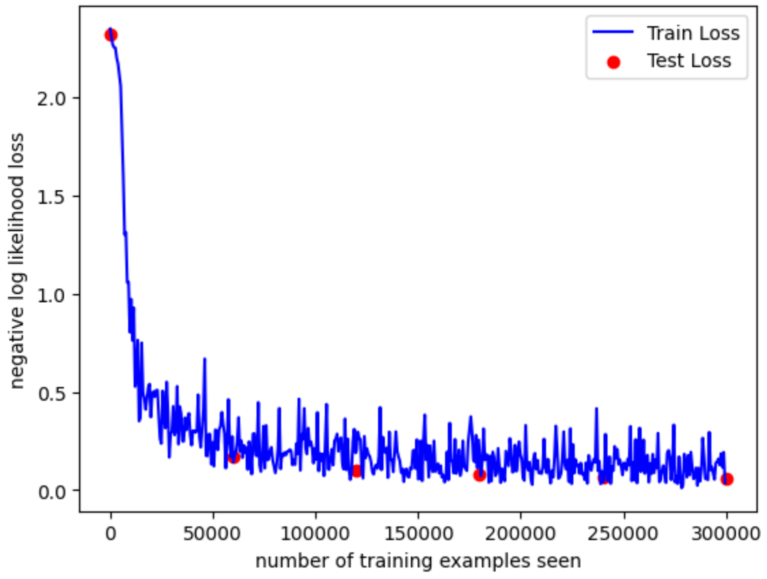


Fig 3. – Train-Test loss error with iterations

Testing on the test set –

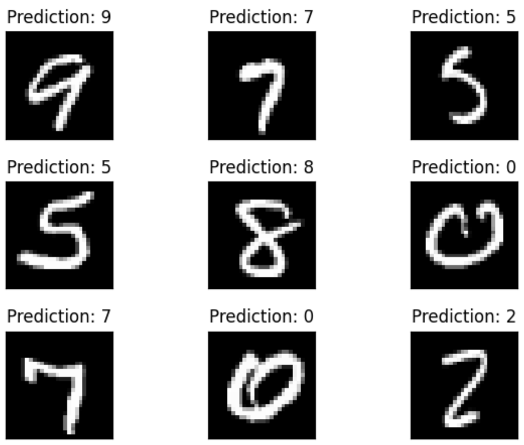
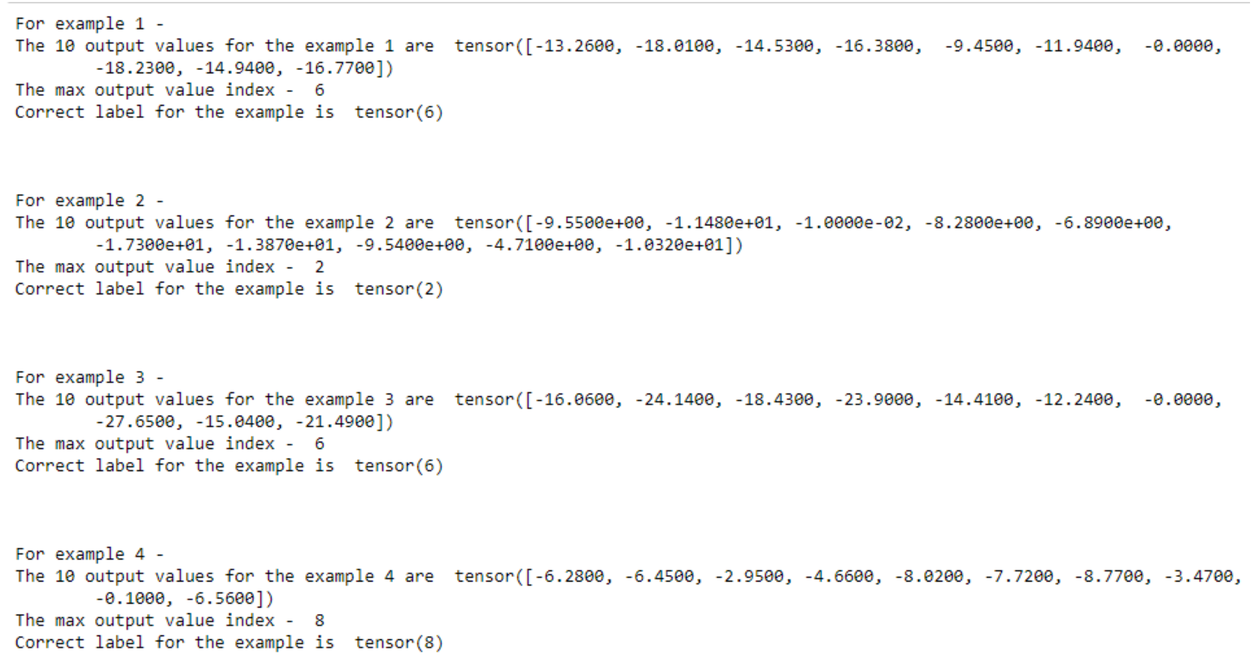


Fig 4. – Predictions on our test set



```
For example 1 -
The 10 output values for the example 1 are  tensor([-13.2600, -18.0100, -14.5300, -16.3800,  -9.4500, -11.9400,  -0.0000,
        -18.2300, -14.9400, -16.7700])
The max output value index -  6
Correct label for the example is  tensor(6)


For example 2 -
The 10 output values for the example 2 are  tensor([-9.5500e+00, -1.1480e+01, -1.0000e-02, -8.2800e+00, -6.8900e+00,
        -1.7300e+01, -1.3870e+01, -9.5400e+00, -4.7100e+00, -1.0320e+01])
The max output value index -  2
Correct label for the example is  tensor(2)


For example 3 -
The 10 output values for the example 3 are  tensor([-16.0600, -24.1400, -18.4300, -23.9000, -14.4100, -12.2400,  -0.0000,
        -27.6500, -15.0400, -21.4900])
The max output value index -  6
Correct label for the example is  tensor(6)


For example 4 -
The 10 output values for the example 4 are  tensor([-6.2800, -6.4500, -2.9500, -4.6600, -8.0200, -7.7200, -8.7700, -3.4700,
        -0.1000, -6.5600])
The max output value index -  8
Correct label for the example is  tensor(8)
```

Fig 5. – Output layer values for each example, its max value index and the correct label from test set.

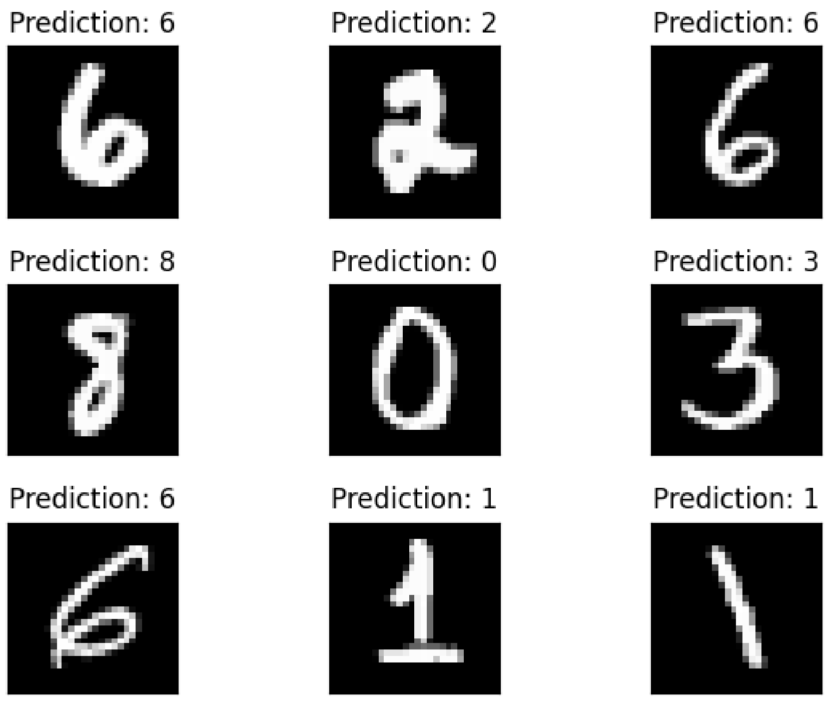


Fig 6. – Predictions for first 9 digits of test set.

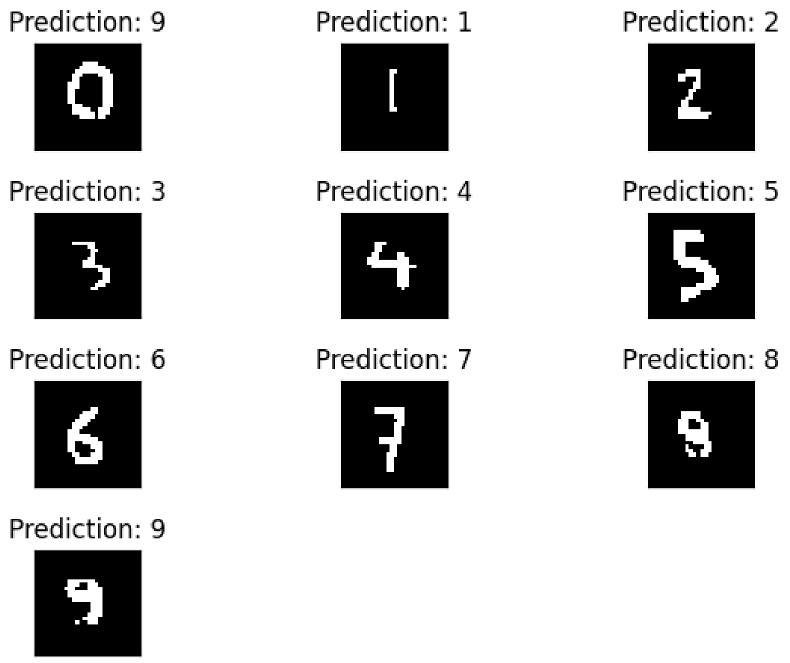


Fig 7. – Predictions for our hand drawn digits.

On our hand drawn digits, the performance is pretty good with 9/10 digits correctly identified or 90% accuracy with problem in recognizing 0 which might be classified to 9 because of the break in the loop which looks like 9.

## B. Analysis

In this section we are observing the influences of our trained filters individually on the MNIST digits. The filters trained for CNN layer 1 looks like below –

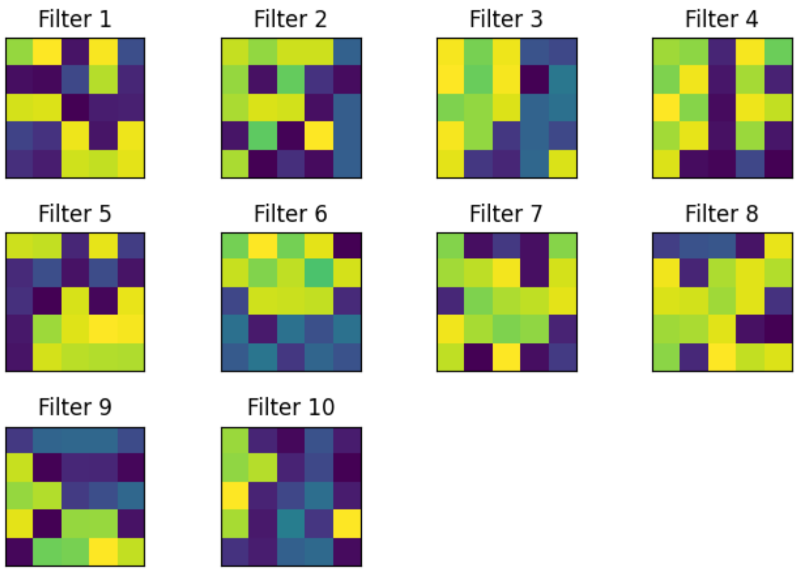


Fig 8. – Filters visualization of our first convolution neural network

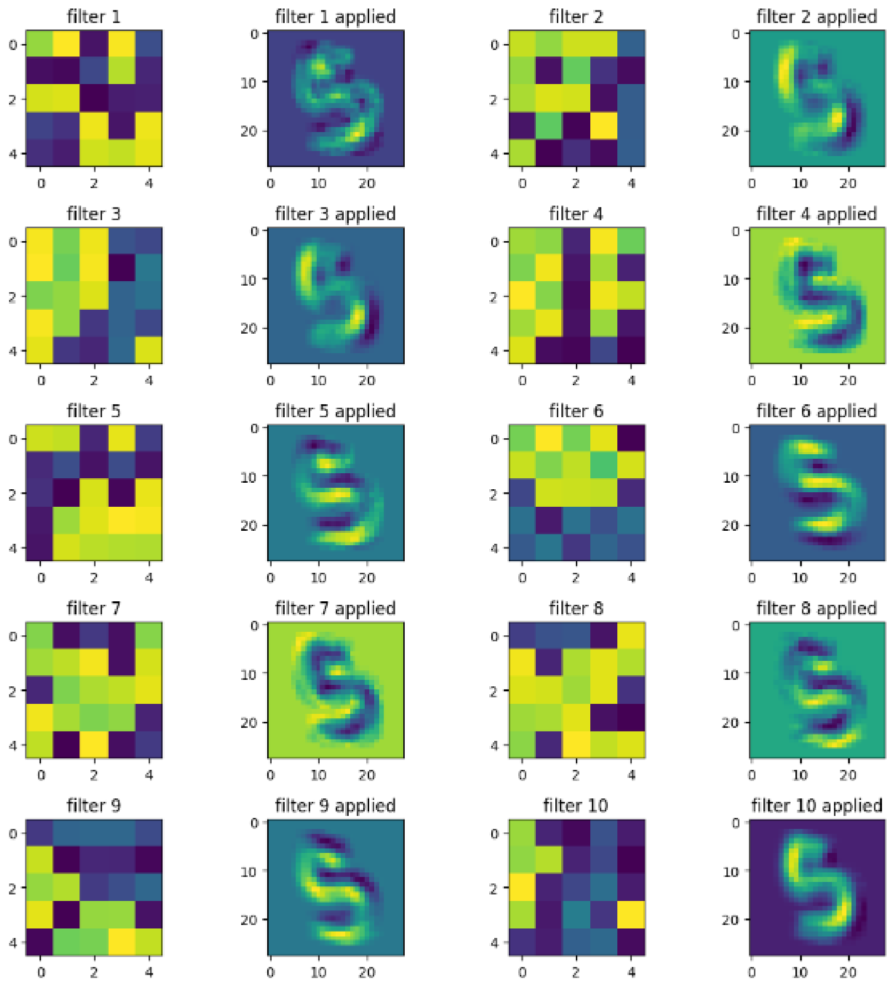


Fig 9. – Filters impact on the digit image from MNIST

From comparison of the trained filters and the corresponding image, each filter is extracting a different kind of edge from our number. The filters can be understood as gradient filters for recognizing edges of our digits from different angles in the image.

## C. Transfer Learning

After training our model on the MNIST digit dataset, we transfer our trained model for recognizing first 3 letters of Greek language which are alpha, beta, and gamma. This is done by freezing the trained weights of our learned model and replacing the last layer of the model with only three outputs for our three letters.

Following are the outputs required for this task –

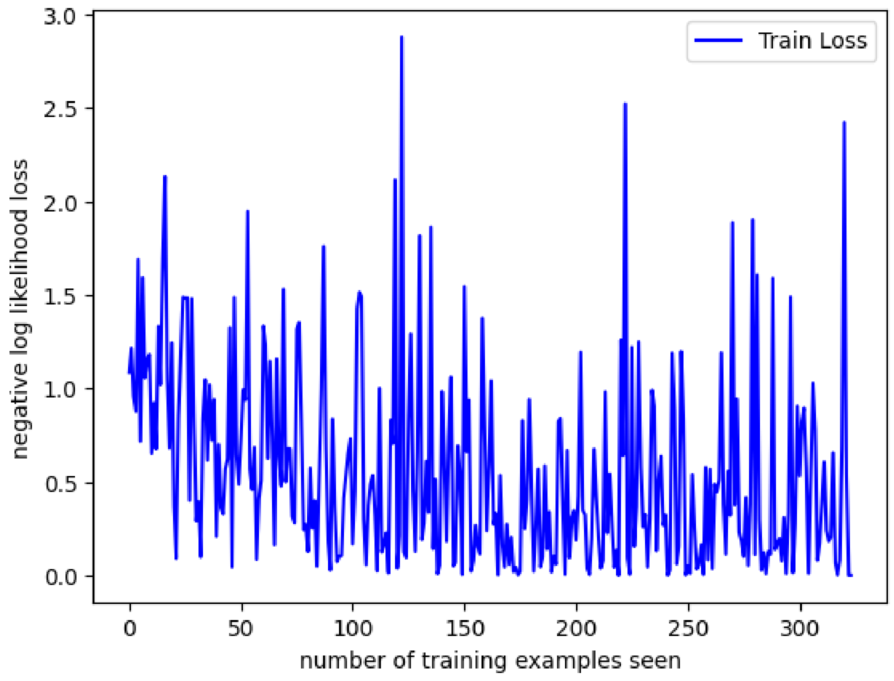
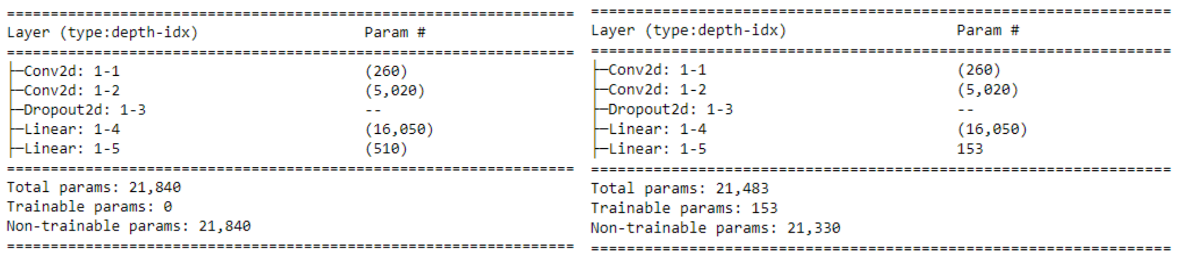


Fig. 10 – Training loss with iterations (for 12 epochs 100% train accuracy)



```
================================================================
Layer (type:depth-idx)                   Param #
================================================================
├─Conv2d: 1-1                            (260)
├─Conv2d: 1-2                            (5,020)
├─Dropout2d: 1-3                         --
├─Linear: 1-4                            (16,050)
├─Linear: 1-5                            (510)
================================================================
Total params: 21,840
Trainable params: 0
Non-trainable params: 21,840
================================================================
```

```
================================================================
Layer (type:depth-idx)                   Param #
================================================================
├─Conv2d: 1-1                            (260)
├─Conv2d: 1-2                            (5,020)
├─Dropout2d: 1-3                         --
├─Linear: 1-4                            (16,050)
├─Linear: 1-5                            153
================================================================
Total params: 21,483
Trainable params: 153
Non-trainable params: 21,330
================================================================
```

Fig. 11 – Training loss with iterations (for 12 epochs 100% train accuracy)

Test accuracy on our hand drawn Greek letters –

```
Test set: Avg. loss: 0.8956, Accuracy: 12/15 (80%)
```

## D. Design of own experiment –

In this section, we have experimented with our parameters in the neural networks. These are the parameters on which experimentation was performed –

drop_percent = [0.3, 0.4, 0.5, 0.6, 0.7]

batch_size = [40, 80, 100, 120, 150]

kernel_s = [4, 5]

**Hypothesis -**

**Batch size –**

- Faster convergence: Larger batch sizes can result in faster convergence during training, as the optimization algorithm has access to more information about the gradient at each iteration. This is because the gradient estimate becomes more accurate with more examples.
- Generalization: Smaller batch sizes may result in better generalization performance. This is because smaller batches allow the network to see a more diverse set of examples during training, which can help the network learn more robust and generalizable features.
- Memory usage: Larger batch sizes require more memory to store the intermediate computations during training, which can be an issue on devices with limited memory. Smaller batch sizes can help mitigate this problem.
- Accuracy: The accuracy of a neural network may also be affected by the batch size. In some cases, larger batch sizes may result in better accuracy, while in other cases, smaller batch sizes may be better. The optimal batch size may depend on the specific problem and architecture being used.

**Drop-out percentage –**

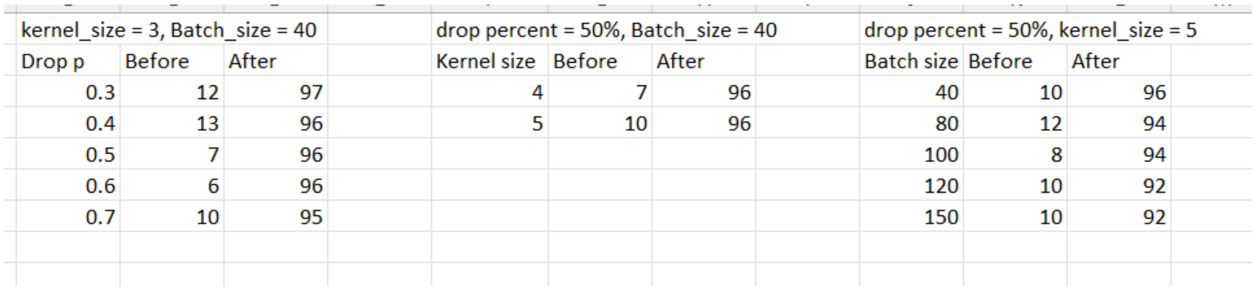
- Regularization: Increasing the dropout percentage generally increases the amount of regularization applied during training. This can help prevent overfitting, which is a common problem in deep neural networks.
- Training time: Increasing the dropout percentage can also increase the amount of time required to train the network, as more iterations may be required to achieve convergence.
- Accuracy: The optimal dropout percentage for a given network architecture and dataset depends on various factors, including the complexity of the problem and the amount of available training data. In some cases, increasing the dropout percentage can improve accuracy by preventing overfitting, while in other cases, it may decrease accuracy by underfitting the data.
- Robustness: Dropout can also improve the robustness of a network by reducing its sensitivity to small changes in the input data. This can be especially useful in applications where the input data may be noisy or corrupted.

**Kernel size –**

- Receptive field: The kernel size determines the size of the receptive field of each neuron in the convolutional layer. A larger kernel size can result in a larger receptive field, which can allow the network to capture more complex spatial patterns in the input data. However, using too large of a kernel size can lead to information loss and reduced spatial resolution.
- Computational complexity: Increasing the kernel size can increase the computational complexity of the network. This can be a problem for large networks or when training on limited computational resources.
- Overfitting: Using a smaller kernel size can increase the risk of overfitting, as the network may not capture enough spatial information to generalize well to new data. On the other hand, using a larger kernel size can help prevent overfitting by capturing more complex patterns in the data.

- Performance: The optimal kernel size depends on the specific problem and architecture being used. In some cases, a larger kernel size may improve performance, while in other cases, a smaller kernel size may be more appropriate.

**Observations –**
We performed our analysis by changing the kernel size of filters, batch size for training, and drop-out percentage. Following are the observations by isolating one and observing the changes –



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| kernel_size = 3, Batch_size = 40 | | | drop percent = 50%, Batch_size = 40 | | | drop percent = 50%, kernel_size = 5 | | |
| Drop p | Before | After | Kernel size | Before | After | Batch size | Before | After |
| 0.3 | 12 | 97 | 4 | 7 | 96 | 40 | 10 | 96 |
| 0.4 | 13 | 96 | 5 | 10 | 96 | 80 | 12 | 94 |
| 0.5 | 7 | 96 | | | | 100 | 8 | 94 |
| 0.6 | 6 | 96 | | | | 120 | 10 | 92 |
| 0.7 | 10 | 95 | | | | 150 | 10 | 92 |

**Conclusion -**
From the observations, we can clearly conclude that decreasing drop_out percentage increases the accuracy, but this may be the case where our test data is very similar to train data hence the regularization is inversely affecting the accuracy. We tested on 2 kernel sizes and observed that the changes in the accuracy are not that severe while computation time greatly increases with reducing filter size. The final conclusion which can be reported is increasing the batch size for training decreased accuracy because of the reasons stated above. Thus our hypothesis justifies our observations.

## Extension –
We have created a deep net model which predicts class of animal.

A. Exploring the dataset –

The dataset is downloaded from Kaggle and is of 10 animal classes collected from google images - link. It has over 28K medium quality images. The classes in the dataset and its distribution is stated below.

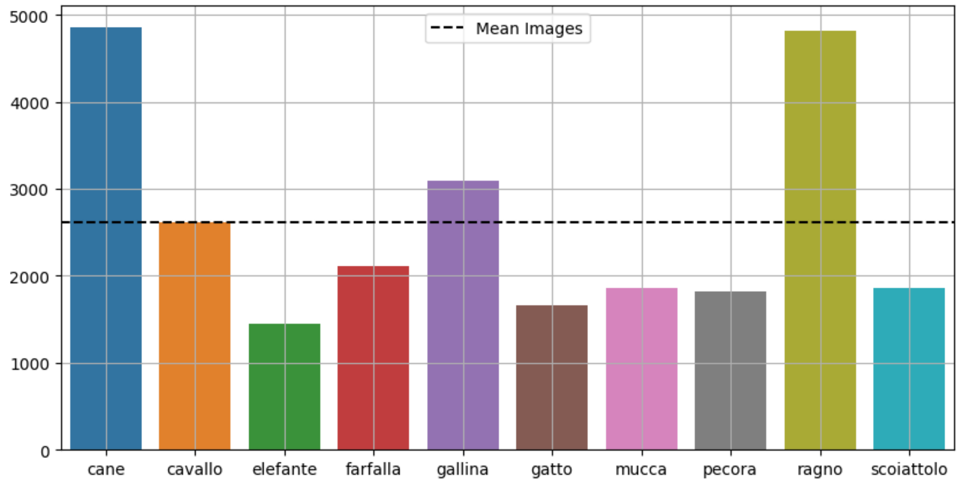


Fig. 12 – Classes of animals in the dataset.

Here are few examples of the classified images with its ground truth –

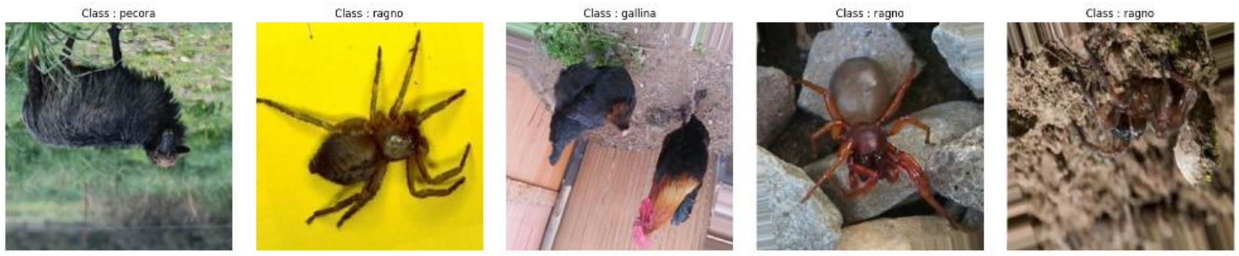
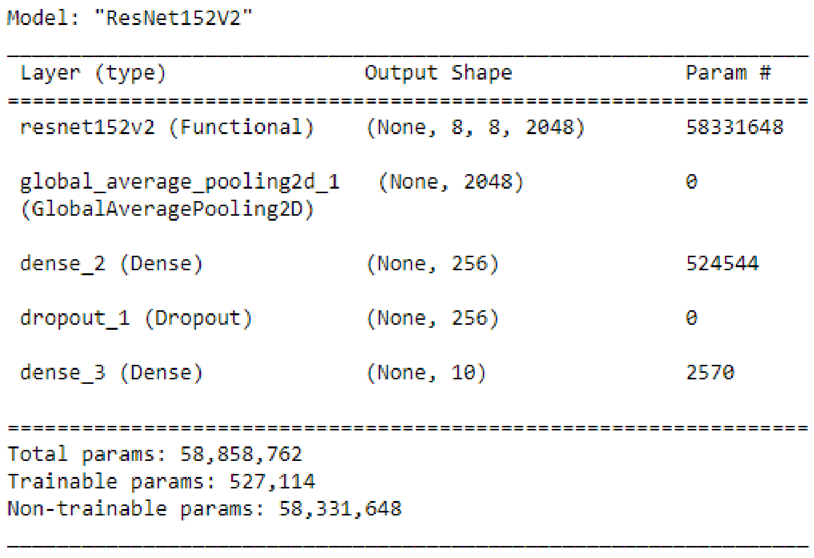


Fig. 13 – Visualizing the data with its class of family.

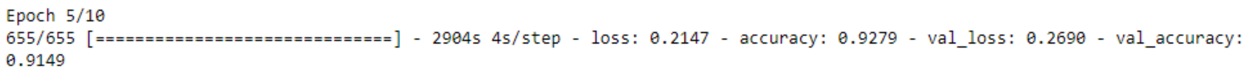
B. Building the model –

We are using a pretrained ResNet model - ResNet152V2 and transferring the learning for our data. The architecture of the model can be seen below –



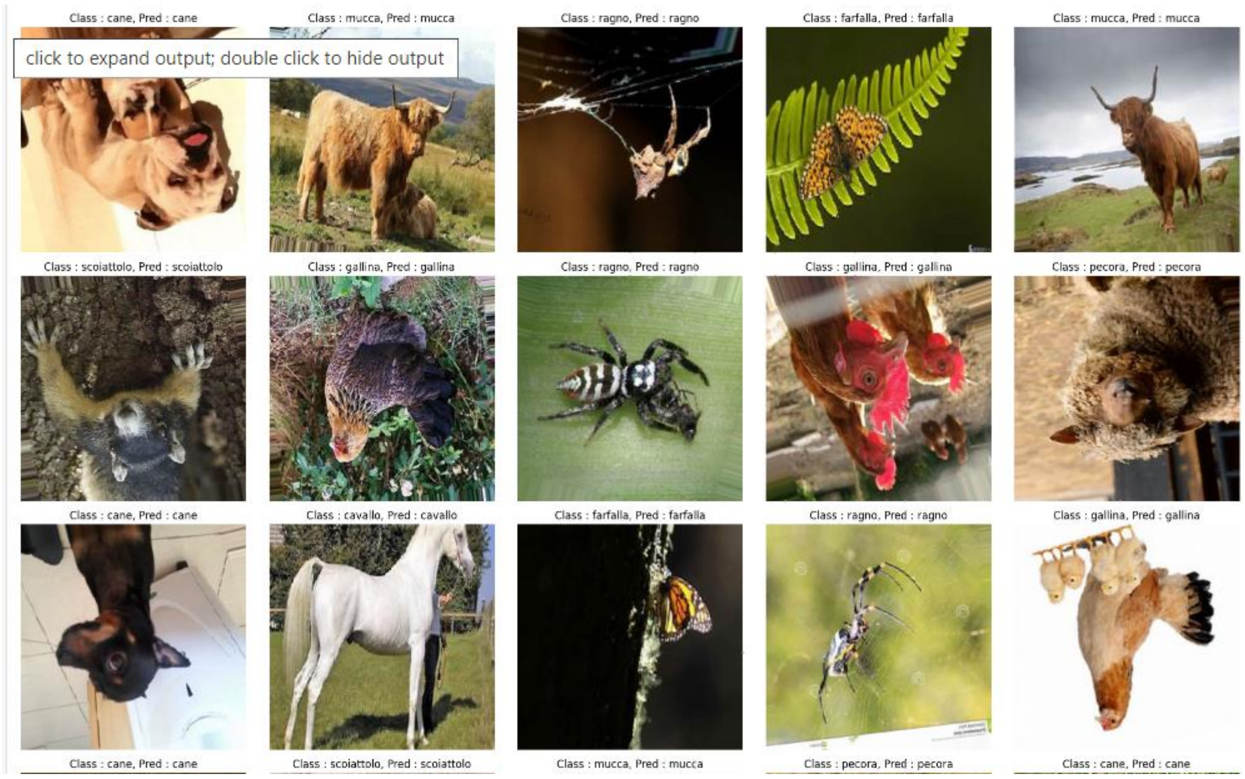
```
Model: "ResNet152V2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet152v2 (Functional)    (None, 8, 8, 2048)        58331648

 global_average_pooling2d_1   (None, 2048)             0
 (GlobalAveragePooling2D)

 dense_2 (Dense)             (None, 256)               524544

 dropout_1 (Dropout)         (None, 256)               0

 dense_3 (Dense)             (None, 10)                2570

=================================================================
Total params: 58,858,762
Trainable params: 527,114
Non-trainable params: 58,331,648
_________________________________________________________________
```

This is done for 5 epochs, and we get the following output after finishing training –



```
Epoch 5/10
655/655 [==============================] - 2904s 4s/step - loss: 0.2147 - accuracy: 0.9279 - val_loss: 0.2690 - val_accuracy:
0.9149
```

The final predictions on test images are shown below –


Class : cane, Pred : cane
Class : mucca, Pred : mucca
Class : ragno, Pred : ragno
Class : farfalla, Pred : farfalla
Class : mucca, Pred : mucca
click to expand output; double click to hide output
Class : scoiattolo, Pred : scoiattolo
Class : gallina, Pred : gallina
Class : ragno, Pred : ragno
Class : gallina, Pred : gallina
Class : pecora, Pred : pecora
Class : cane, Pred : cane
Class : cavallo, Pred : cavallo
Class : farfalla, Pred : farfalla
Class : ragno, Pred : ragno
Class : gallina, Pred : gallina
Class : cane, Pred : cane
Class : scoiattolo, Pred : scoiattolo
Class : mucca, Pred : mucca
Class : pecora, Pred : pecora
Class : cane, Pred : cane

## Reflection –

Deep nets are amazing. Although lots of it seems to be a black box, when we do analysis part of this assignment we can dig in deeper and understand how they function. Looking forward to explore more.

## Acknowledgement –

Used Pytorch, Tensorflow documentation. Stackoverflow for resolving errors. Kaggle for dataset of extension and code base from Kaggle.