Project Number 2 : Report

Student : Gabriel Pecoraro

Hawk ID : A20593087

Due on 12/1/2024

Class : Machine Learning and Deep Learning
ECE 566

College : Illinois Institute of Technology

Made by Gabriel Pecoraro

‘

1 Introduction

This project aims to design the most efficient and sustainable Convolutional Neural Network (CNN) based on a predefined 3-layer model.

The initial step involves analyzing the given model, explaining its structure, and demonstrating its limitations through performance evaluation and results.

Subsequently, the optimized model will be presented, with a detailed explanation of its parameters, functions, and improvements. Performance tests will be conducted, and the architecture will be visualized through comprehensive plots.

The dataset used is the MNIST dataset which gathers several gray colored hand-written digits from 0 to 9. This is a standard dataset which does not requires a huge amount of feature extraction as the images does not contain a lot information. The code will be run using the language Python and the module `numpy` and `keras`. The notebook was ran using Kaggle Notebook as it allows a 30-hour GPU access.

2 Initial Model

This first model is a 3-layer CNN featuring sequentially of a $2D$ Convolution cell, a flatten cell and a dropout cell.

- The convolutional layer apply a convolutional operation to input images, using filters (also known as kernels) to detect features such as edges, textures, and more complex patterns.
- The flatten cell serves as a bridge between the convolutional and pooling layers and the fully connected (dense) layers. Its primary function is to transform the multi-dimensional output of the convolutional and pooling layers into a one-dimensional array, which is then fed into the fully connected layers suitable for classification or regression tasks.
- The dense layer applies a linear operation on the layer's input vector enable the final prediction.

The given model includes three key tunable hyper parameters :

- The batch size - which is strictly inferior to the number of samples - allows to make a faster training as it requires less memory. However, it is important to bear in mind that the smaller the batch size is the more the gradient fluctuates, which can be an issue.
- The Validation Split serves to keep track on the ability of the model to adapt on unseen data in order to track a possible overfitting. It is a part of the training set allocated for validation purpose.

- The number of epoch indicates the number of time the model went through the training dataset. As the number of epoch gets higher, the possibility of learning increases but the risk of overfitting the model as well. That is why, we have to find the best tradeoff between achieving the best learning and avoid overfitting.

Finally using the `keras` library, there are plenty of gradient descent possibility, three are applicable :

- Adelta
- SGD
- Adam

A first test using a batch size of 64, allowing 0.2% of the samples available in the training dataset for validation running on 20 epochs was performed. The graphs are below :

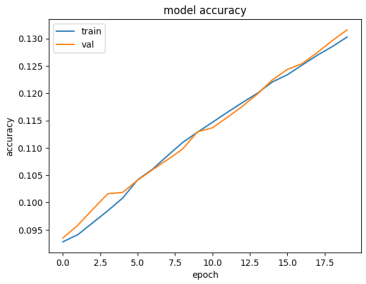

Figure 1: Accuracy of the model using AdaDelta

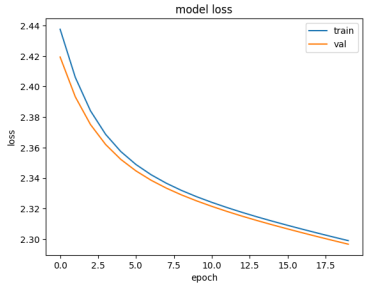

Figure 2: Loss of the model using AdaDelt

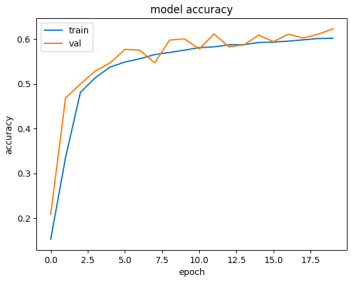

Figure 3: Accuracy of the model using SGD

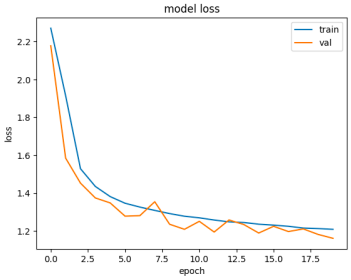

Figure 4: Loss of the model using SGD

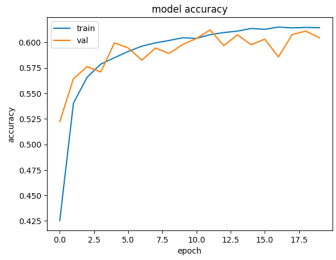

Figure 5: Accuracy of the model using Adam

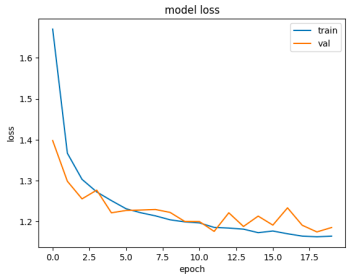

Figure 6: Loss of the model using Adam

Layer (type)	Output Shape	Param #
conv2d_54 (Conv2D)	(None, 28, 28, 1)	10
flatten_12 (Flatten)	(None, 676)	0
dense_30 (Dense)	(None, 10)	6,770

Figure 7: Model Architecture

At first glance, all the plots above emphasize that the performances of this model are poor and not conclusive due to the shallowness of the network which does not allow to go deep in the image analyze to extract all the information

to have an efficient model. This is foregrounded by the figure 7 which gives an indication concerning the number of features extracted.

However, there is a dramatic difference in the performance between the training using the *AdaDelta* optimizer with the *SGD* and *Adam* optimizer. Overall, the results using the *AdaDelta* optimizer are not conclusive, therefore for our next model. It will not be considered. Now let's compare the result on *Adam* and *SGD*. Both of them show good results on the training accuracy and loss. Nonetheless , `Adam` appears to be the best optimizer among the three, as it provides the fastest and most stable convergence in both accuracy and loss curves. In Addition, the `Adam` optimizer is reputed to be the most consistent gradient descend, that is why I chose it.

Thus, those figures provide an insight on how the model can be enhanced. First of all, there is an urging need to add more convolutional layer to obviously retrieve more features and parameters from the data regardless the fact that the structure also needs to be strengthened using other layers such as pooling, Batch Normalization or adding activation function.

To sum up this first study of the based predefined model points out a dramatic lack of performance. In order to cope with it, I will enhance this model by providing to this latter a higher complexity. This goes by giving more complex layers, fine tunning hyper parameters and changing parameters of the layers. Moreover, in order to keep the processing time reasonable, processing features such as Early Stopping or Reducing the Learning Rate on Plateau might be added.

3 Proposed Model

From now on, our main goal is to create an enhanced neural network. In order to achieve that, we ought to add depth in the neural network. Therefore, I inspired myself from the VGG architecture which is one of the most efficient and I tried to adapt it to the problem. The VGG architecture contains a several amount of layers that may be too much regarding the simplicity of our dataset and what we want to achieve. Therefore, at first I decided to reduce the number of convolutional layer which starts at 16 for simplest VGG architectures to 6. In fact, when I tested from 16 to 10 layers, and, regardless the processing time of one minute per epoch, I did not achieve higher performances. Furthermore, having a very deep architecture is interesting when the dataset contains images with a tremendous amount of features but here, we only have hand-written numbers. Henceforth, the number of parameter extracted does not have to be the highest to achieve good performances.

Now, each part and function of my model will be explained.

3.1 Convolutional Part

First of all, I added a consequent amount of convolutional layer which an increasing number of filter sizes $(64, 128)$ to capture a hierarchy of features, from simple edges in the initial layers to more complex patterns in deeper layers. Overall, I implemented 6 convolutional layers. The depth of this model enhances the training on the input data by allowing them to learn more abstract representation. The kernel size of those convolutional layers is $(3, 3)$ with a stride at 1 and an ReLU activation function at the end. This activation function will bring non-linearity to the model avoiding it to compute only linear function. I used ReLU because the implementation and resource usage is the most convenient due to the simplicity of the function.

3.2 MaxPooling Part

A crucial part that was lacking in the predefined model is a down-sampling part such as MaxPooling or AveragePooling. Those are placed after a convolutional function or at the end of a layer in order to reduce the dimension and minimize the use of computational resources. Both of these pooling techniques will give more importance to the highest pixel value which represents the most informative pixel. Nevertheless, MaxPooling tends to be used when the position of the object that have to be located is not relevant. This is our case here, that is why, I will be using only this method. I set the kernel of the MaxPooling at $(2, 2)$;I did not implement a Pooling function on each layer to avoid an over-down-sampling which could lead to size issues.

3.3 Dropout Part

The Dropout part will be randomly ignoring nodes. This is a regularization technique where some number of layer outputs will be randomly ignored in order to force the model to train on noisy data and henceforth, avoiding a possible overfitting of the model. I set each dropout rate in the first layers at 0.2 in order the reduce consequently the amount of output layers but also keeping the model large enough to get an efficient training.

3.4 Flatten Part

I kept the flatten part as it is a vital component for the good behavior of the network. It is not removable

3.5 Fully Connected Part

A fully connected layer in neural networks is particularly useful because it ensures that every neuron in the layer is connected to every neuron in the previous layer. This dense connectivity allows the model to learn complex representations of data, enabling it to capture interactions between all features processed by previous layers. Before the fully connected part begins, I put a 0.5 rate

dropout function, it will ignore half of the neurons in order to enhance the training but also to avoid the fully connected part of being overwhelmed.

Afterwards, I put a two dense layers with 256 neurons and one dense layer with 100 neurons. This layer serves as a fully connected layer that can learn non-linear combinations since we used the 'ReLU' activation function that enables the network to learn more complex functions.

At the very end, I finally put a dense layer with 10 neurons (one for each digit) and with a Softmax activation function in order to perform the classification. This function converts the output to probability-like values, with each neuron's output representing the probability that the input image belongs to one of the 10 classes.

3.6 Batch Normalization

The Batch Normalization is a normalization technique done between the layers of a Neural Network instead of in the raw data. It means that we normalize the data using the following formula :

$$\hat{x}^{(k)} = \frac{x^k - \mathbf{E}[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

It is usually inserted between the convolutional layer and the non-linearity. I inserted it on each convolutional layer. The main advantage of using this technique stands in the fact that it eases the training on deep neural network, so it may ease the training of the new model. Furthermore, it allows a faster convergences of the gradient.

Overall, this method is a good asset for us as it enhances dramatically our training process.

3.7 Data Augmentation

All the upgrades and tunings that I made on the precedent model to create mine will drastically enhance the training accuracy for sure and also maybe the test accuracy.

However, the main point of a deep neural network is to have a good accuracy on unseen data. Hence, the goal is to accustom the model already during the training process to unseen data in order increase the test accuracy.

To achieve that, I performed what is called Data Augmentation from `keras`. By using in the `layers` package the module `RandomZoom` allows to apply random transformation - in this case random zoom - on the image of the dataset creating new training data. Henceforth, the model is used to cope with unseen data and make it more robust. I did not implemented other image transformation process such as `RandomFlip`, `RandomTranslation` or `RandomCrop` because there is no sense to flip, to crop or to shift an image with a number. I tried to implement it but it confused the model and led to a depletion in the performances. There was also a possibility to change the contrast of the images but I did not explore

this possibility considering that the MNIST dataset is a gray scale image so there was no point.

3.8 Running Parameters

3.8.1 Early Stopping

As said before, the main goal is to improve the test accuracy although training accuracy is important as well but it is still less relevant. To achieve that, combined with the Data Augmentation. I tried to implement the Early Stopping and the Reduce Learning Rate on Plateau.

The Early Stopping will automatically stop the execution if the monitored value - here the validation loss - is not decreasing anymore after a defined 'patience' amount of epochs. It enhanced the performance of our model by pushing it to his training limits but it allows to take out the overfitting burden by interrupting the process. Moreover, it resolves the burden of fine-tuning the number of epochs. Yet, this latter is not significant anymore because the process will automatically stop

3.8.2 Reduce Learning Rate On Plateau

The issue also I wanted to deal with by looking on the first plots is the stagnation problem. Indeed, sometimes the accuracy and loss are leveling off and process will just wait for the last epoch to finish. The Reduce Learning Rate On Plateau method copes with this problem. This callback monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced. Models often benefit from reducing the learning rate by a factor of $[0.3, 0.01]$ once the learning stagnates.

Those two running parameters will help the model to train the most efficiently as possible

3.9 Results

3.9.1 Selection of Hyper-parameters without Early Stopping and ReduceLROnPlateau

I changed the values of hyper-parameters. At first glance, without using the early stopping, I trained the model on 20 epochs as the model converges quickly using the `Adam` optimizer. As the training accuracy had at the beginning a slower increasing trend compared to the validation accuracy, I decided to put a consequent number of batches to enhance the training. Furthermore, I put the validation split at 0.1 as it yields the best results although I tried other configurations.

3.9.2 Fine-Tuning with the Early Stopping and ReduceLROnPlateau parameters

For this test, as the number of epoch and the potential overfitting is not a problem anymore, I set the model running with 120 epochs in order to achieve as much training accuracy as possible. I kept though the same parameters because they led to the best results.

For the Early Stop parameters, I set a 'patience' of epoch at 30 in order to ensure myself that I am not missing a potential improvement part. I also did that because I noticed that actually, there is no overfitting on the validation accuracy, at some point it is just stagnating around a certain value. I put the mode in minimal so the model is able to detect when the validation loss is not decreasing anymore. Finally, the Early Stopping can be applied only after 50 epoch, in order to make sure that the model is solely leveling off indefinitely and not skipping a potential improvement step.

The test was run first using the raw architecture without all the model enhancement such as Batch Normalization, Early Stopping and so forth, and another test was performed using all those latter. The results are displayed below :

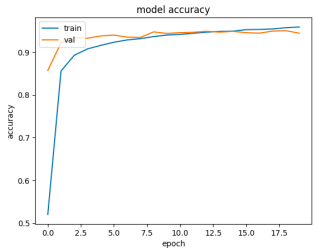

Figure 8: Accuracy on the non-enhanced model

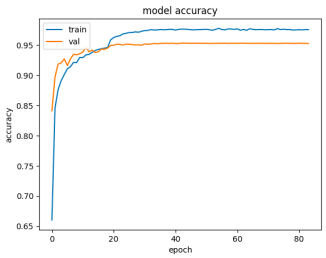

Figure 9: Accuracy on the enhanced model

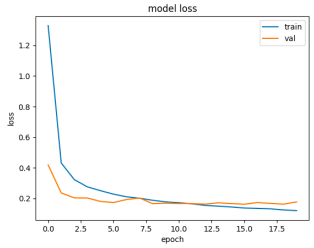

Figure 10: Loss of the non-enhanced model

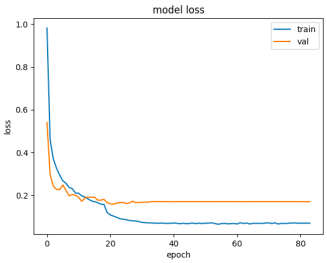

Figure 11: Loss of the enhanced model

```
Number of classes: 10
Shape of x_train: (10000, 28, 28, 1)

313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Accuracy in test set is: 0.9404
```

Figure 12: Test Accuracy on the non-enhanced model

```
Number of classes: 10
Shape of x_train: (10000, 28, 28, 1)

313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Accuracy in test set is: 0.9466
```

Figure 13: Test Accuracy on the enhanced model

3.10 Model Architecture Schematic

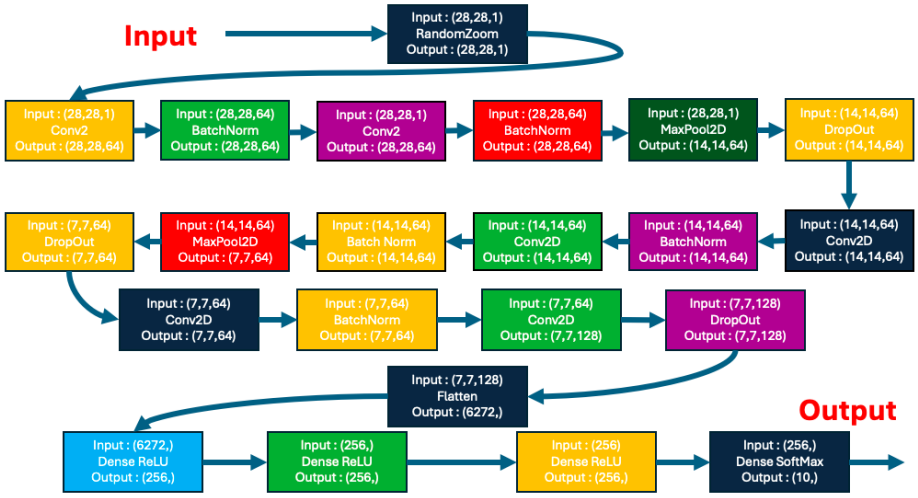

Figure 14: Global Structure of the Model

4 Conclusion

The performance comparison between the non-enhanced and enhanced models highlights the impact of incorporating advanced training techniques and hyper parameter tuning on a convolutional neural network (CNN) with a shared fully connected architecture. The non-enhanced model, utilizing a 6-convolutional-layer algorithm without batch normalization, early stopping, data augmentation, or learning rate adjustments, achieved a test accuracy of 94.04% over 20 epochs. In contrast, the enhanced model, which included these techniques and was trained over 120 epochs, demonstrated a slightly higher test accuracy of 94.66%.

Beyond the small improvement in test accuracy, the enhanced model showed significant advantages in terms of training and validation stability. The loss and accuracy curves of the enhanced model exhibited smoother convergence, lower overall loss, and a reduced risk of overfitting compared to the non-enhanced model. This demonstrates the efficacy of batch normalization in stabilizing training, data augmentation in improving generalization, and early stopping combined with learning rate adjustments in fine-tuning the optimization process. With those technics, the training accuracy went from 93.57% to 97.76% while the validation accuracy went from 94.25% to 95.25%. The total training time went from 1 minute and 20 seconds to more than 3 *minutes* as there where more epochs.

This project was a good opportunity to encompass and learn all the possible architecture that offers the Convolutional Neural Network.

5 References

Pooling Techniques
Data Augmentation

6 Appendix : Code

Listing 1: Source Code

```
1
2  # %matplotlib inline
3  import os
4  import numpy as np
5  import pickle
6  import matplotlib.pyplot as plt
7
8  from keras.utils import to_categorical
9  from keras.callbacks import ReduceLROnPlateau,
       ModelCheckpoint, EarlyStopping
10 from keras.optimizers import Adadelta, Adam, SGD
11 from keras.layers import Input, Conv2D, Dense, MaxPooling2D,
        Dropout, Flatten, AveragePooling2D, Conv2DTranspose,
       UpSampling2D, BatchNormalization, RandomFlip, RandomZoom,
        RandomContrast, RandomBrightness, RandomCrop,
       RandomTranslation
12 from keras.models import Sequential
13 from keras.losses import categorical_crossentropy
14
15 """# **Loading Data**"""
16
17 data = np.load('/kaggle/input/mnist-corrnoise-npz/
       MNIST_CorrNoise.npz')
18
19 x_train = data['x_train']
20 y_train = data['y_train']
21
22 num_cls = len(np.unique(y_train))
23 print('Number of classes: ' + str(num_cls))
24
25 print('Example of handwritten digit with correlated noise: \
       n')
26
27 k = 3000
28 plt.imshow(np.squeeze(x_train[k,:,:]))
29 plt.show()
30 print('Class: '+str(y_train[k])+'\n')
31
32 # RESHAPE and standarize
33 x_train = np.expand_dims(x_train/255,axis=3)
```

```
34
35 # convert class vectors to binary class matrices
36 y_train = to_categorical(y_train, num_cls)
37
38 print('Shape of x_train: '+str(x_train.shape))
39 print('Shape of y_train: '+str(y_train.shape))
40
41 """# Training classic CNN"""
42
43 model_name='CNN' # To compare models, you can give them
       different names
44
45 pweight='/kaggle/working/weights_' + model_name  + '.keras'
46
47 if not os.path.exists('/kaggle/input/weights'):
48   print("Does Not exists")
49   os.mkdir('./weights')
50
51 ## EXPLORE VALUES AND FIND A GOOD SET
52 b_size = 64 # batch size
53 val_split = 0.2 # percentage of samples used for validation
       (e.g. 0.5)
54 ep = 20 # number of epochs
55
56 """## Train the model
57
58 # Param for My Model
59 """
60
61 model_name='CNN' # To compare models, you can give them
       different names
62
63 pweight='/kaggle/working/weights_' + model_name  + '.keras'
64
65 if not os.path.exists('/kaggle/input/weights'):
66   print("Does Not exists")
67   os.mkdir('./weights')
68
69 ## EXPLORE VALUES AND FIND A GOOD SET
70 b_size = 128 # batch size
71 val_split = 0.1 # percentage of samples used for validation
       (e.g. 0.5)
72 ep = 120 # number of epochs
73
74 # Setting Input
75 input_shape = x_train.shape[1:4] #(28,28,1)
76
77 # Set input shape and number of classes
78 input_shape = (28, 28, 1)  # Grayscale 28x28 images
79 num_classes = 10
```

```
80
81 # Model Definition
82 model = Sequential()
83
84 # Data Augmnetation
85 model.add(RandomZoom(height_factor=0.1, width_factor=0.1))
86
87 # 1st Layer
88 model.add(Conv2D(64, kernel_size=(3,3), padding="same",
       input_shape=input_shape,activation="relu"))
89 model.add(BatchNormalization())
90 model.add(Conv2D(64, kernel_size=(3,3), padding="same",
       activation="relu"))
91 model.add(BatchNormalization())
92 model.add(MaxPooling2D((2, 2)))
93 model.add(Dropout(0.2))
94
95 # 2nd Layer
96 model.add(Conv2D(64, kernel_size=(3,3), padding="same",
       activation="relu"))
97 model.add(BatchNormalization())
98 model.add(Conv2D(64, kernel_size=(3,3), padding="same",
       activation="relu"))
99 model.add(BatchNormalization())
100 model.add(MaxPooling2D((2, 2)))
101 model.add(Dropout(0.2))
102
103 #3rd Layer
104 model.add(Conv2D(128, kernel_size=(3,3), padding="same",
       activation="relu"))
105 model.add(BatchNormalization())
106 model.add(Conv2D(128, kernel_size=(3,3), padding="same",
       activation="relu"))
107 model.add(Dropout(0.2))
108
109 # Flattening Layer
110 model.add(Flatten())
111
112 # Fully Connected Layer
113 model.add(Dropout(0.5))
114 model.add(Dense(256, activation='relu'))
115 model.add(Dense(256, activation='relu'))
116 model.add(Dense(100, activation='relu'))
117 model.add(Dense(10, activation='softmax'))
118
119
120 pweight = '/kaggle/working/weights_CNN.keras'
121 # CheckPointer
122 checkpointer = ModelCheckpoint(filepath=pweight, verbose=1,
       save_best_only=True)
```

```
# Early Stopping
early_loss = EarlyStopping(monitor = "val_loss", patience =
    30, start_from_epoch = 50, mode="min", verbose=1)
# Reduce Learning Rate
reduce_lr = ReduceLROnPlateau(monitor = "val_loss", patience
     = 7,  mode = "min", verbose=1, factor=0.15)

# Callback List
callbacks_list = [checkpointer, early_loss, reduce_lr]

# Compiling model
model.compile(loss=categorical_crossentropy,
              optimizer=Adam(),
              metrics=['accuracy'])

# Fitting model
history=model.fit(x_train, y_train,
                        epochs=ep,
                        batch_size=b_size,
                        verbose=1,
                        shuffle=True,
                        validation_split = val_split,
                        callbacks=callbacks_list)


print('CNN weights saved in ' + pweight)

# Plot loss vs epochs
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper right')
plt.show()

# Plot accuracy vs epochs
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()

from keras.models import load_model

## LOAD DATA
data = np.load('/kaggle/input/mnist-corrnoise-npz/
    MNIST_CorrNoise.npz')
```

```
170
171 x_test = data['x_test']
172 y_test = data['y_test']
173
174 num_cls = len(np.unique(y_test))
175 print('Number of classes: ' + str(num_cls))
176
177 # RESHAPE and standarize
178 x_test = np.expand_dims(x_test/255,axis=3)
179
180 print('Shape of x_train: '+str(x_test.shape)+'\n')
181
182 ## Define model parameters
183 model_name='CNN' # To compare models, you can give them
      different names
184 pweight='/kaggle/working/weights_' + model_name  + '.keras'
185
186 model = load_model(pweight)
187 # Instead of predict_classes, use predict and argmax
188 y_pred_probs = model.predict(x_test)  # Get predicted
      probabilities
189 y_pred = np.argmax(y_pred_probs, axis=-1) # Get class with
      highest probability
190
191 Acc_pred = sum(y_pred == y_test)/len(y_test)
192
193 print('Accuracy in test set is: '+str(Acc_pred))
```