JSS Science and Technology University Mysuru

Classification of Aquatic Animals Using Convolutional Neural Network

Shreyas BP

01JST17IS045

Department of Information Science and Engineering

DECLARATION

This Third Year Project is presented in fulfillment of the requirements for a Computer Vision Mini Project Under the guidance of Dr. B S Harish.

It is entirely my work and has not been submitted to any other University or higher education institution, or for any other academic award in this University. Where use has been made of the work of other people it has been fully acknowledged and fully referenced.

Signature: ___________________

Date: _______________________

TABLE OF CONTENT

ABSTRACT:

The Convolutional Neural Network also known as CNN is one of the most used techniques for image regarding computation etc. It has become an integral part of most of the peoples in that way they do not know. To learn and understand is the main aim and also to understand the view of computer Vision i.e. how computer recognizes an image, how it trains, understands, and classifies it. Through this project, we will get a basic idea of the Convolutional Neutral Network and some basic Computer Vision.

KEYWORDS:

CNN, Conv2D, ConvNet, Aquatic animals, Keras, CIFAR-100, CIFAR-10, forward, and Backpropagation.

CHAPTER 1: INTRODUCTION

1.1 Problem Statement:

Classification of images into beaver, dolphin, otter, seal, whale, aquarium fish, flatfish, ray, shark, and trout using Convolutional Neural Network.

1.2 General Introduction:

In this project I have used the **CIFAR-100** dataset (Canadian Institute for Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. This dataset consists of 100 classes out of which I extract 10 classes and apply various Computer Vision Algorithms to achieve Object. I decide to use Convolutional Neural Network as it is very good at image classification and expect to give good accuracy that other normal algorithm. Convolutional neural networks typically consist of an input layer, several hidden layers, followed by a softmax classification layer. The input layer, and each of the hidden layers, is represented by a three-dimensional array with size, say, M x N x N. The second and third dimensions are spatial. The first dimension is simply a list of features available in each spatial location. Now let's see about the selected classes from the dataset,

1.1.1 Aquatic Animals:

An aquatic animal is an animal, either vertebrate or invertebrate, which lives in the water for most or all of its lifetime. The term aquatic can be applied to animals that live in either freshwater (freshwater animals) or saltwater (marine animals). However, the adjective marine is most commonly used for animals that live in saltwater, i.e. in oceans, seas, etc.

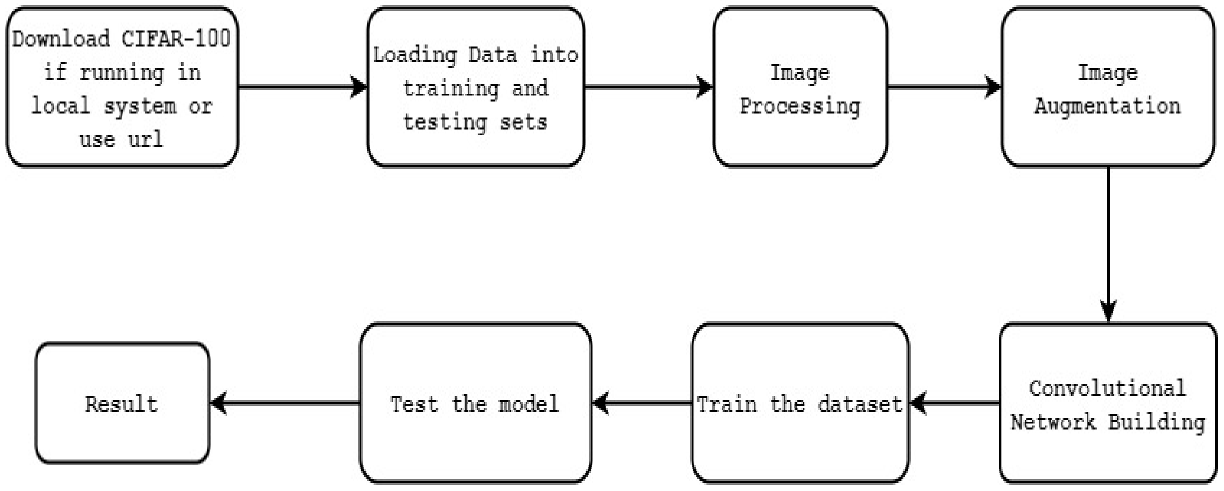
1.1.2 Aquatic Mammals:

Aquatic mammals are a diverse group of mammals that dwell partly or entirely in bodies of water. They include the various marine mammals who dwell in oceans, as well as various freshwater species, such as the European otter. They are not a taxon and are not unified by any distinct biological grouping, but rather their dependence on and integral relation to aquatic ecosystems. In this project I have taken 5 classes of Aquatic Mammals, they are beaver, dolphin, otter, seal, and whale.

1.1.3 Fishes:

Fish are gill-bearing aquatic craniate animals that lack limbs with digits. They form a sister group to the tunicates, together forming the olfactory. Included in this definition are the living hagfish, lampreys, and cartilaginous and bony fish as well as various extinct related groups. Tetrapod's emerged within lobe-finned fishes, so cladistical they are fish as well. However, traditionally fish are rendered paraphyletic by excluding the tetrapod's (i.e., the amphibians, reptiles, birds, and mammals which all descended from within the same ancestry). In this project I have chosen only 5 classes of fishes, they are aquarium fish, flatfish, ray, shark, and trout.

The below diagram represents a simplified block diagram of the entire process.


Download CIFAR-100 if running in local system or use url
Loading Data into training and testing sets
Image Processing
Image Augmentation
Convolutional Network Building
Train the dataset
Test the model
Result

Block Diagram

1.3 Application:

Since the main goal of this project is to understand the working of various algorithms and to understand how the computer identifies the image and learn them and try to identify through that learning. Hence, this project does not have any specific applications.

1.4 Challenges:

One of the main challenges of Computer Vision is the Input size. The more the dimension of the image more the complexity. So, handling the size is an important factor in Computer Vision. The main challenges I face during this project duration are

- Input Size: Since I have only 6000 images, out of which 1000 are for testing. Left images for training are only 5000. This may be not enough to train the ConvNet.
- Hardware resource: Since CIFAR-100 has 60000 images loading it on small RAM pc may lead to a lag in the system and may cause slow computation.
- All these 10 classes are closely related, so classifying in challenging.

1.5 Motivation:

The main aim of this project is to understand various concepts of Computer Vision and apply them through this project, and during this project, I got a chance for understanding Convolutional Neural Network and my Machine Learning concepts as we use them in this project. Since I have chosen the first 10 classes which belong to 2 super classes i.e. Aquatic mammals and Fishes. Most of these images have same background.

1.6 Objectives:

The main objective of this project is to understand computer vision course and applying the learned method to classify Aquatic Animals into Aquatic Mammals and Fishes as Superclass and further Aquatic mammals into beaver, dolphin, otter, seal, whale, and Fishes into aquarium fish, flatfish, ray, shark, and trout. Understanding the working of ConvNet.

CHAPTER 2: LITERATURE SURVEY

In this paper [3] they have presented high-performance GPU-based CNN variants trained by on-line gradient descent. Principal advantages include state-of-the-art generalization capabilities, great flexibility, and speed. All structural CNN parameters such as input image size, number of hidden layers, number of maps per layer, kernel sizes, skipping factors, and connection tables are adaptable to any particular application. They applied their networks to benchmark datasets for digit recognition (MNIST), 3D object recognition (NORB), and natural images (CIFAR10). On MNIST the best network achieved a recognition test error rate of 0.35% , on NORB 2.53% , and CIFAR10 19.51% . The results were raising the bars for all three benchmarks. The particular CNN types discussed in this paper seem to be the best adaptive image recognizers, provided there is a labeled dataset of sufficient size. No unsupervised pertaining is required. Good results require big and deep but sparsely connected CNNs, computationally prohibitive on CPUs, but feasible on current GPUs, where our implementation is 10 to 60 times faster than a compiler optimized CPU version. This paper [3] gave me a rough idea of how various parameters affect the image classification.

The writer of the paper [4] has investigated multiple techniques to improve upon the current state of the art deep convolutional neural network-based image classification pipeline. The techniques include adding more image transformations to the training data, adding more transformations to generate additional predictions at test time, and using complementary models applied to higher resolution images.

In this paper, they have shown several ways to improve neural network-based image classification systems. They first showed some new useful image transformations to increase the effective size of the training set. These were based on using more of the image to select training crops and additional color manipulations. They have also shown useful image transformations for generating testing predictions. We made predictions at different scales and generated predictions on different views of the image. These additional predictions can slow down the system so they showed a simple greedy algorithm that reduces the number of predictions needed. Finally, they showed an efficient way to train higher resolution models that generate useful complementary predictions. A single base model and a single high-resolution model are as good as 5 base models. These improvements to the image classification pipeline are easy to implement and should be able to improve other convolutional neural network-based image classification systems [4].

In this paper [5], they have proposed a fine-grained classification pipeline combining bottom-up and two top-down attentions. The object-level attention feeds the network with patches relevant to the task domain with different views and scales. This leads to better CNN feature for fine-grained classification, as the network is driven by domain-relevant patches that are also rich with shift/scale variances. The part-level attention focuses on local discriminate patterns and also achieves pose normalization. Both levels of attention can bring significant gains, and they compensate each other nicely with late fusion. One important advantage of the method that is used in this paper is that the attention is derived from CNN trained with classification task, thus it can be conducted under the weakest supervision setting where the only class label is provided [5].

After reading the above three papers, I had a basic idea of what CNN is and why it is best for image classification. Then I read a few papers on forward and backpropagation.

This paper [2] discusses the basic definitions concerning the multi-layer feed-forward neural networks are given. The back-propagation training algorithm is explained. Partial derivatives of the objective function concerning the weight and threshold coefficients are derived. These derivatives are valuable for an adaptation process of the considered neural network. Training and generalization of multi-layer feed-forward neural networks are discussed. Improvements in the standard back-propagation algorithm are reviewed. Further applications of neural networks in chemistry are reviewed. The advantages and disadvantages of multilayer feed-forward neural networks are discussed [2].

This paper [1], talks about backpropagation. This paper demonstrates how such constraints can be integrated into a backpropagation network through the architecture of the network. The approach mentioned in this paper has been successfully applied to the recognition of handwritten zip code digits provided by the U.S. Postal Services. A single network leans the entire recognition operation, going from the normalized image of the character to the final classification. In this paper, they have successfully applied back-propagation learning to a large, real-world task. Their network was trained on the low-level representation of data that had minimal pre-processing. Their work points out the necessity of having flexible “network design” software tools that ease the design of complex, specialized network architecture.

For understanding Keras, so that I should be able to build the model, I referred to this [7] book and Keras Documentation [8]. Here, they have explained everything about the Keras, from installing the Keras to using every API’s and functions or modules that the Keras provide. From the point of extracting the data till building the ConvNet and getting test results and evaluating the model [7] [8].

After the model is ready and is test, now we have to calculate time complexity of the model. This paper [6], talks about recent advanced convolutional neural networks (CNNs), and how they have been improving the image recognition accuracy, the models are getting more complex and time consuming. For real-world applications in industrial and commercial scenarios, engineers and developers are often faced with the requirement of constrained time budget. In this paper, we investigate the accuracy of CNNs under constrained time cost. Under this constraint, the designs of the network architectures should exhibit as trade-offs among the factors like depth, numbers of filters, filter sizes, etc. With a series of controlled comparisons, we progressively modify a baseline model while preserving its time complexity. This is also helpful for understanding the importance of the factors in network designs.

CHAPTER 3: PROPOSED METHOD

3.1 Design Methodology:

This design process is a sequence of steps that define a structured and predictable approach to the development of this ConvNet. If a process is structured, then we know what tasks must be completed. If a process is predictable, then we can estimate how long each task will take.

Their 3 main phases in,

- Extract the required classes from the data set and load them as train and test sets.

- Build a ConvNet and train the dataset.

- Test the built ConvNet.

3.1.1 Extraction:

In this phase, we design the methodology to extract the required 10 classes from a dataset of 100 classes. Since the dataset comes with labels along with the images, we use this to our advantage. We use the ***Chainer*** framework, using which we download the cifar100 dataset. Chainer is an open-source deep learning framework written purely in Python on top of Numpy and CuPy Python libraries. The development is led by Japanese venture company Preferred Networks in partnership with IBM, Intel, Microsoft, and NVidia. For further details refer references for URL.

Chainer supports a common interface for training and validation of the ***dataset***. The dataset support consists of three components: dataset's, iterator, and batch conversion functions. We use a dataset that has a function get_cifar100 (), which returns the dataset in the form of training and testing set. The data set has 100 class and each class has 500 training and 100 testing images. Hence, a total of 600 images per class. Total of 60000 images.

Since we use Chainer framework, it returns each image with corresponding class number i.e.

CIFAR100_LABELS_LIST = ['apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed', 'bee', 'beetle', 'bicycle', 'bottle', 'bowl', 'boy', 'bridge', 'bus', 'butterfly', 'camel', 'can', 'castle', 'caterpillar', 'cattle', 'chair', 'chimpanzee', 'clock', 'cloud', 'cockroach', 'couch', 'crab', 'crocodile', 'cup', 'dinosaur', 'dolphin', 'elephant', 'flatfish', 'forest', 'fox', 'girl', 'hamster', 'house', 'kangaroo', 'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion', 'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle', 'mountain', 'mouse', 'mushroom', 'oak_tree', 'orange', 'orchid', 'otter', 'palm_tree', 'pear', 'pickup_truck', 'pine_tree', 'plain', 'plate', 'poppy', 'porcupine', 'possum', 'rabbit', 'raccoon', 'ray', 'road', 'rocket', 'rose', 'sea', 'seal', 'shark', 'shrew', 'skunk', 'skyscraper', 'snail', 'snake', 'spider', 'squirrel', 'streetcar', 'sunflower', 'sweet_pepper', 'table', 'tank', 'telephone', 'television', 'tiger', 'tractor', 'train', 'trout', 'tulip', 'turtle', 'wardrobe', 'whale', 'willow_tree', 'wolf', 'woman', 'worm'].

The index of each label indicates the class number. Using this number we extract the required classes. The required classes are,

Selected Class = { 1:'aquarium fish', 4:'beaver', 30:'dolphin', 32:'flatfish', 55:'otter', 72:'seal', 95:'whale', 57:'ray', 73:'shark', 91:'trout'}

Once we extract we need to pre-process and split then into x_train, y_train, and x_test and y_test. Now the data is read, the next step is to build a ConvNet so that it can learn from the dataset and try to classify the images.

3.1.2 Building ConvNet:

In this phase, we build a ConvNet that once trained will be able to classify the trained class's images into corresponding classes. We use the most common and popular module to build that ConvNet i.e. is ***Keras***. Kerasis an open-source neural network library written in Python. It is capable of running on top of Tensor Flow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. Keras provides a wide range of API's and function that helps us to build a ConvNet and train them. The API's and functions we use are

Most of the function's and API's I use in this project are from Keras. Few of them are,

- From **preprocessing** we import **ImageDataGenerator()**:

  We use this function to augment the images for better accuracy and training purposes.

- From **models** we import **Sequential()**:

  Using this we stack the model into a layer using the **add** method.

- From **layers** we import **Dense**, **Dropout**, **Activation**, **Flatten**, **BatchNormalization**, **Conv2D**, **MaxPooling2D**:

  Each of these API's and functions have together build up a Convolutional Neural Network (ConvNet).

- From **losses** import **categorical_crossentropy**.

  3.1.2.1 Architecture:

  The model consists of three convolutional layers with a single rectified linear unit (ReLU) and a max-pooling layer in between. The input to the model is an image from the dataset, of shape 32 x 32 x 3. The first convolutional layer consists of 64 filters, each with shape 3 x 3, with a stride of 1 and no padding. The height and width of the output of the convolutional layer are defined below, where H and W are the output height and width, H and W are the input height and width, the pad is the padding size, the stride is the stride length, and F is the filter size.

$$H = 1 + \frac{H + 2pad - F}{stride}$$

$$W = 1 + \frac{W + 2pad - F}{stride}$$

  Following the convolutional layer is the ReLU layer, which is defined as:

$$f(x) = max(0, x)$$

  Where x is the input of the ReLU layer.

  After the ReLU layer, we run a 3 x 3 max-pooling layer, with default stride. Max pooling will take the maximum value of a 3 x 3 section of the input layer and keep only that value. We repeat the same process with 128 filters with shape 3 x 3, another layer of 256 filters with shape 3 x 3, and finally last layer with 512 filters with shape 3 x 3 with the ReLU and max-pooling layers of shape 2 X in between each. Afterward, we flatten the output layer to be of shape 1 x 256 and perform a fully connected layer to produce an output vector of length 10. Next, we compute the loss of the model. We use a cross-entropy loss. Cross entropy loss is defined as

$$L\left(f\left(x\right),y\right) = -yln\left(f\left(x\right)\right) - \left(1-y\right)ln\left(1-f\left(x\right)\right)$$

Where x is the output vector of length 56 and y is a label of either 0 or 1, depending on if the classification if incorrect or correct, respectively.

3.1.2.2 Optimizer:

We tested three different optimizers to see which would give the best results for our model. The three optimizers were adaptive moment estimation (**Adam**), **RMSprop**, and stochastic gradient descent (**SGD**). Each of the optimizers has its advantages and disadvantages that we will discuss below.

SGD updates the parameters for every training example. It is fast due to these frequent updates, but this causes a higher variance in the loss and makes it more challenging to converge due to these fluctuations. To reduce the number of oscillations that occur in SGD, we add variable for momentum, which will accelerate the descent. The equations for SGD that we used are shown below:

$$v = m*\ v\ -\ lr\ *\ dx$$

$$x = x + v$$

Where v is the velocity, m is the momentum, lr is the learning rate, dx is the gradient of x, and x the position.

RMSprop is an adaptive learning rate method that uses the root mean square of the gradients to update the weight of our model. It updates the learning rate individually for each parameter and divides the learning rate using an exponentially decaying average of square gradients. RMSprop is unpublished but is widely used due to the accuracy it gives in most models. The equations for RMSprop that we used are shown below:

$$cache = \alpha * cache + (1 \ - \ \alpha) * \ dx2$$

$$x = x - lr * \ \frac{dx}{\sqrt{cache + eps}}$$

Where α is a smoothing coefficient.

Adam is another adaptive learning rate method, like RMSprop, but incorporates momentum changes for each parameter. It uses both the mean and variance of the gradients to update the parameters of the model. Adam was made to address some of the drawbacks of SGD and RMSprop, so it is by far the more robust optimizer. The equations for Adam that we used are shown below:

$$m = beta\ _1 * m + (1 - \ beta_1) * \ dx$$

$$m_t = \frac{m}{1 - beta_1^t}$$

$$v = beta\ _2 * v + (1 - \ beta_2) * \left(dx\ ^2\right)$$

$$v_t = \frac{v}{1 - beta_2^t}$$

$$x = x - lr * \frac{m_t}{\sqrt{v_t} + eps}$$

Where m is the smoothed version of the gradient, t is an iterator that allows for bias correction, and beta1 and beta2 are coefficients for averaging the mean and variance. For testing, we used standard values for learning rate, alpha, betas, and momentum of 0.001, 0.01, 0.1 and 0.999, and 0.9, respectively.

Based on all experiments, I finalize to use RMSprop.

3.1.2.2 Normalizer:

We also tested how normalization affects the run time and the accuracy of our classifier. To normalize our dataset, we took each image and subtracted its mean and divided by its standard deviation. The equation is defined as:

$$x = \frac{x-\mu}{\sigma}$$

Where μ is the mean and σ is the standard deviation.

3.1.3 Testing:

Finally, once the model is trained, we use the test dataset to test the accuracy of the model. Plot various graphs using the Matplotlab to conclude the working of the dataset and model.

3.2 Algorithm:

The below steps gives an abstract idea of how the program works,

Algorithm:

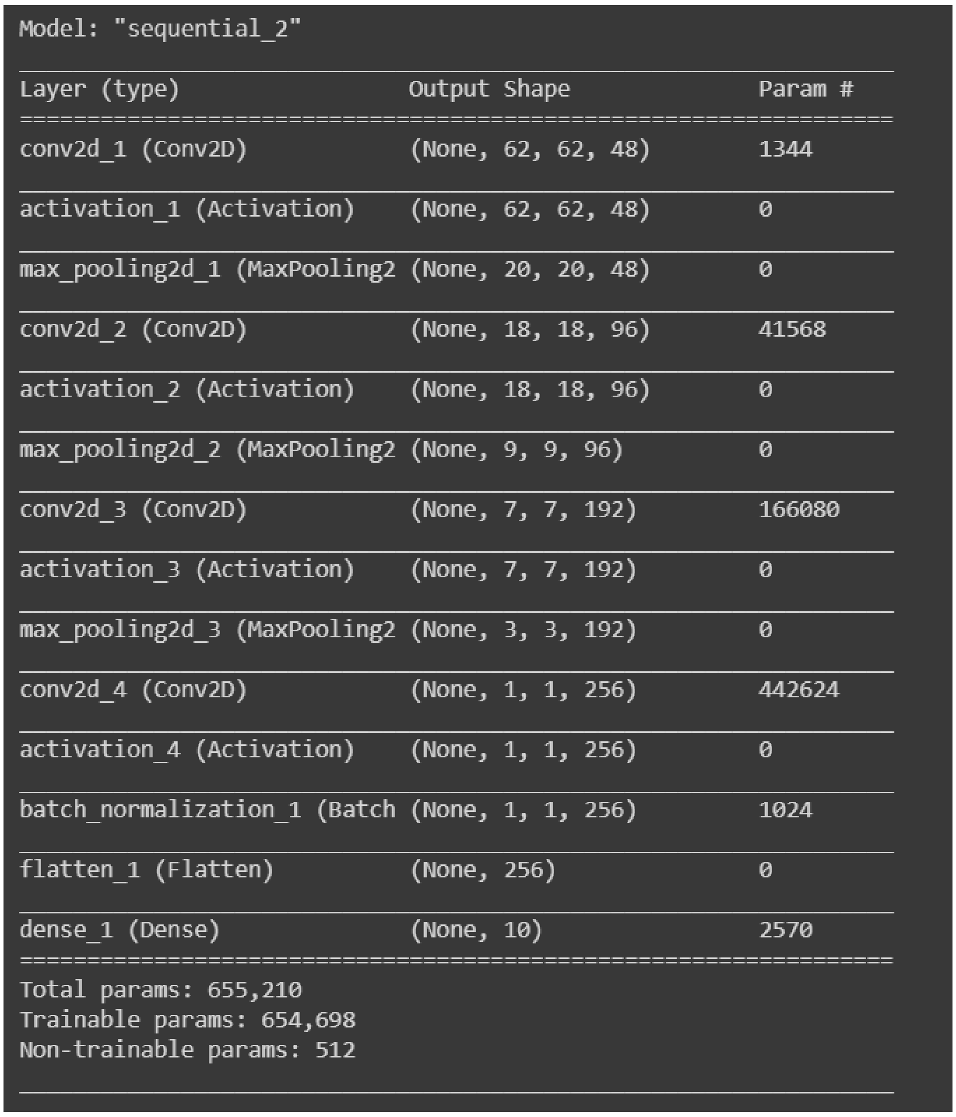
Step 1: Input Image 32 x 32x 3.

Step 2: Extraction of required classes and split them into train and test.

Step 3: Feeding the training dataset is fed into the build model.

Step 4: Test the model using the test dataset.

Step 5: Plot graphs of accuracy and loss.



```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 62, 62, 48)        1344
_________________________________________________________________
activation_1 (Activation)    (None, 62, 62, 48)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 20, 20, 48)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 18, 18, 96)        41568
_________________________________________________________________
activation_2 (Activation)    (None, 18, 18, 96)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 9, 9, 96)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 7, 192)         166080
_________________________________________________________________
activation_3 (Activation)    (None, 7, 7, 192)         0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 3, 3, 192)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 1, 1, 256)         442624
_________________________________________________________________
activation_4 (Activation)    (None, 1, 1, 256)         0
_________________________________________________________________
batch_normalization_1 (Batch (None, 1, 1, 256)         1024
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                2570
=================================================================
Total params: 655,210
Trainable params: 654,698
Non-trainable params: 512
_________________________________________________________________
```

Model Summary

CHAPTER 4: EXPERIMENTAL ANALYSIS

4.1 CIFAR-100

This dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).

Here is the list of classes in the CIFAR-100:

Sr. No.	Super Class	Class
1	Aquatic Mammals	beaver, dolphin, otter, seal, whale
2	Flowers	orchids, poppies, roses, sunflowers, tulips
3	Fish	aquarium fish, flatfish, ray, shark, trout
4	Food containers	bottles, bowls, cans, cups, plates
5	Household electrical devices	clock, computer keyboard, lamp, telephone, television
6	Fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
7	Household furniture	bed, chair, couch, table, wardrobe
8	Large carnivores	bear, leopard, lion, tiger, wolf
9	Insects	bee, beetle, butterfly, caterpillar, cockroach
10	Large man-made outdoor things	bridge, castle, house, road, skyscraper
11	Large natural outdoor scenes	cloud, forest, mountain, plain, sea
12	Medium-sized mammals	fox, porcupine, possum, raccoon, skunk
13	Large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
14	Non-insect invertebrates	crab, lobster, snail, spider, worm
15	Reptiles	crocodile, dinosaur, lizard, snake, turtle
16	People	baby, boy, girl, man, woman
17	Trees	maple, oak, palm, pine, willow
18	Small mammals	hamster, mouse, rabbit, shrew, squirrel
19	Vehicles 1	bicycle, bus, motorcycle, pickup truck, train
20	Vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

The dataset structure is quite the same as the MNIST dataset, it is Tuple Dataset.

Train[i] represents i-th data, there are 50000 training data. Total train data is the same size while the number of class labels increased. So the training data for each class label is fewer than the CIFAR-10 dataset.

The test data structure is the same, with 10000 test data. Total of 60000.

Out of all these available classes we extract the First 10 classes and preprocess them and feed them to our model.

4.2 Experimental Settings:

Due to the unavailability of resources that are required to run this program, I have used ***Google Colaboratory***, or "***Colab***" for short, allows you to write and execute Python in your browser, with zero configuration required, free access to GPUs. It has almost all the library of modules that we require, which allows us to experiment more, as the resource limitations extend.

First, we downloaded the data from the Chainer dataset and load it into the train and test. Later, we extract only required classes.

Our datasets image is of shape 32 x 32 x 3 i.e. image height and width are 32, 32, and 3 represents the number of channels (RGB). But for this classification, since the number of images is small I have resized images into 64 x 64 x 3.

After using various optimizer I have decided to use RMSprop for this classification with the learning rate is 0.0001 and decay is le-6.

I have varied epochs and batch numbers to understand how a model is training and what the progress is.

Convert x_train and x_test into float32 using astype() and normalize them.

We use categorical_crossentropywhen usingthe categorical_crossentropy loss, your targets should be in categorical format i.e. we 10 classes, and the target for each sample should be a 10-

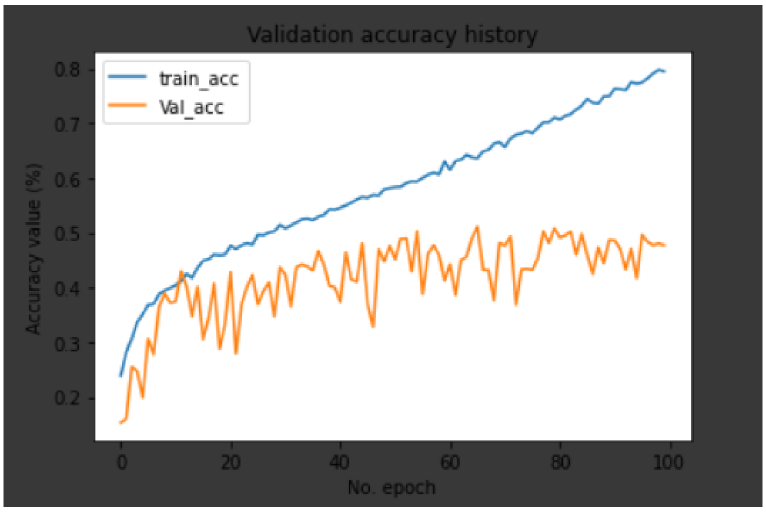
dimensional vector that is all-zeros except for a 1 at the index corresponding to the class of the sample). To convert integer targets into categorical targets, we can use the Keras utility to_categorical.

The data is ready, we build the model.I have used various filters and filter sizes and also may Conv2D layers, dense layer, and MaxPooling2D layers with different filters, padding, and strides.

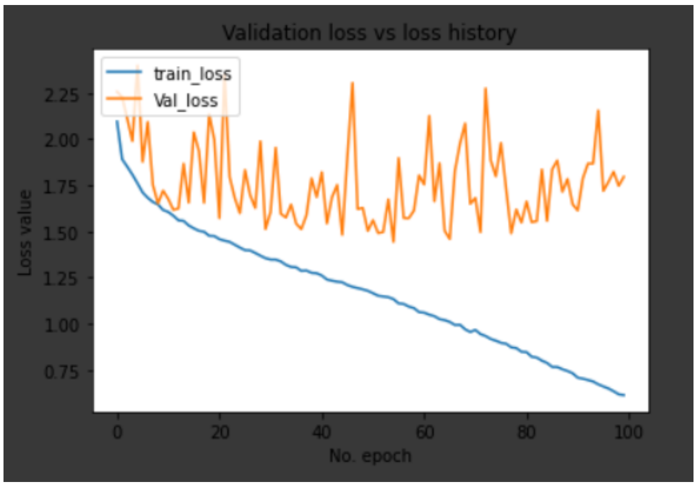
But finally settled to this simple Structure as shown in the model summary.

4.3 Results:

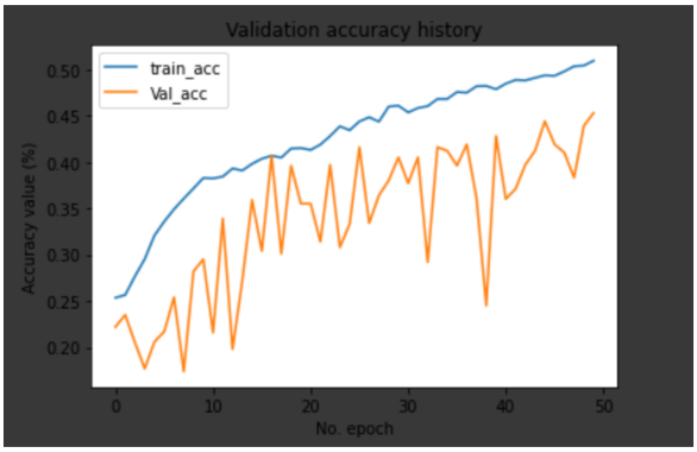
Batch Size	Number of epochs	Test Accuracy	Average train Accuracy
32	50	52.1%	56%
32	100	47.1%	58%
64	50	45.6%	42.5%
64	100	49.5%	50.7%



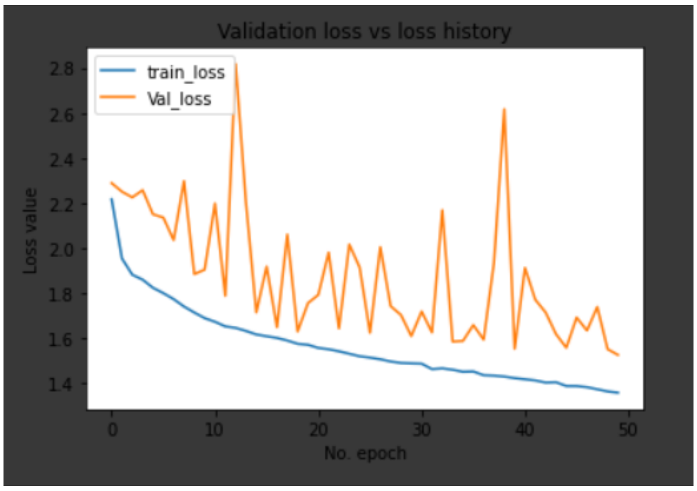
Accuracy vs **Validation Accuracy** (Batch Size = 32)



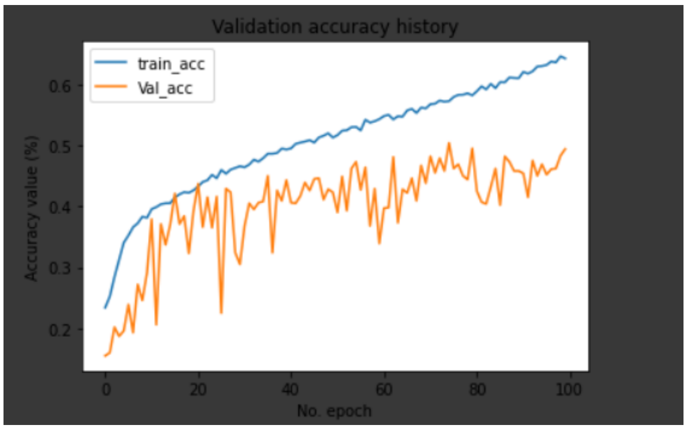
Loss vs **Validation Loss** (Batch Size = 32)



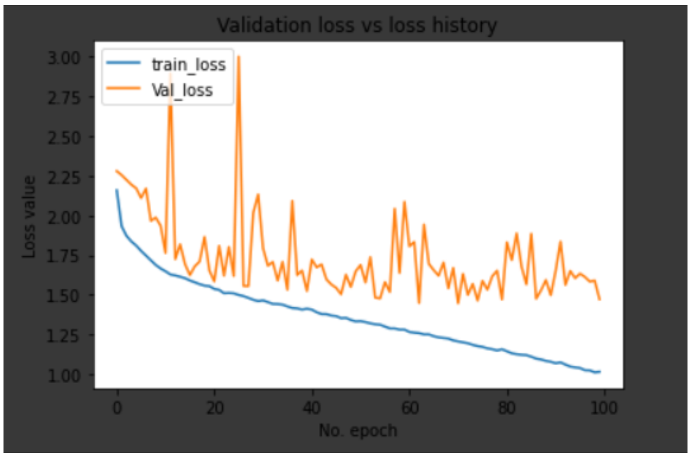
Accuracy vs **Validation Accuracy** (Batch Size = 64)



Loss vs **Validation Loss** (Batch Size = 64)

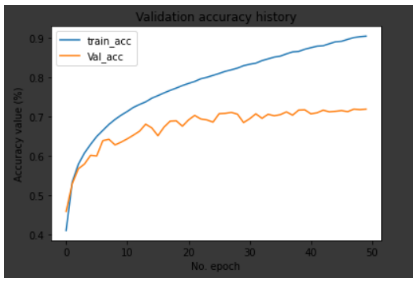


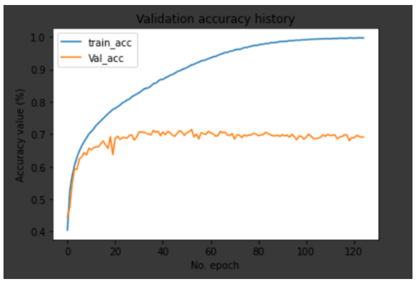
Accuracy vs **Validation Accuracy** (Batch Size = 64)

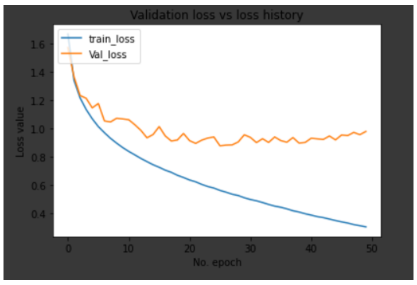


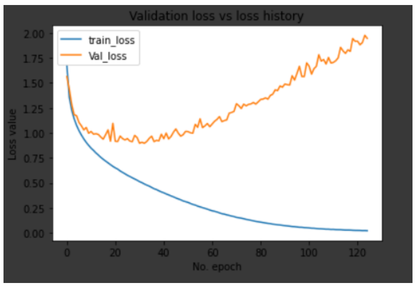
Loss vs **Validation Loss** (Batch Size = 64)

For Evaluation purpose we used CIFAR-10 dataset and trained it to the same model and got these outputs







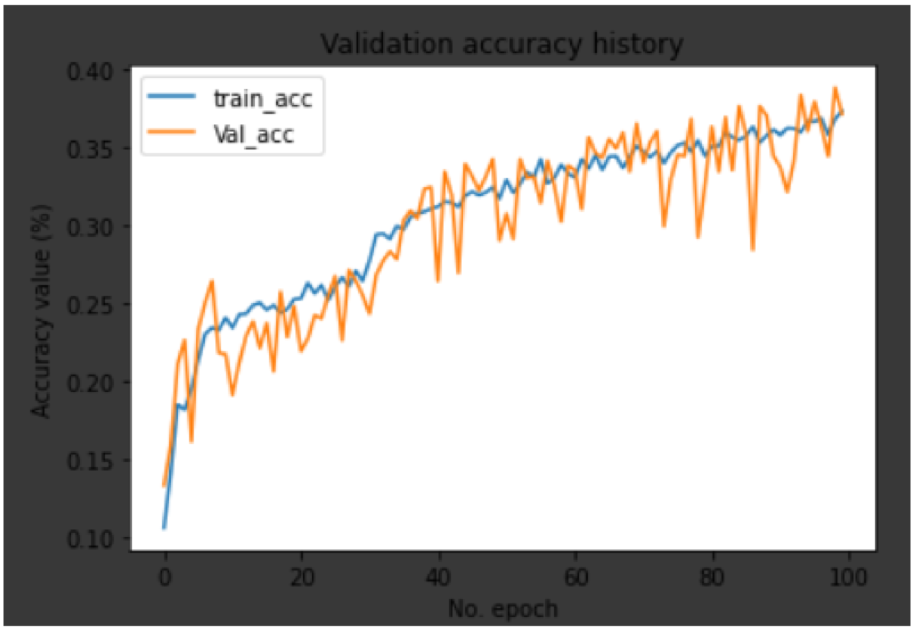


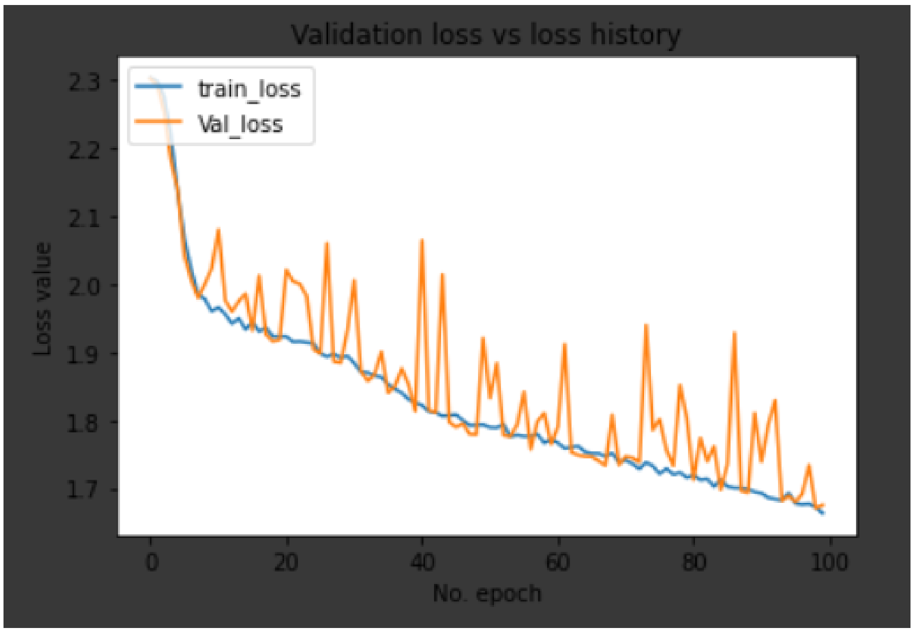
Batch Size = 64

Epoch = 50

Batch Size = 64

Epoch = 100



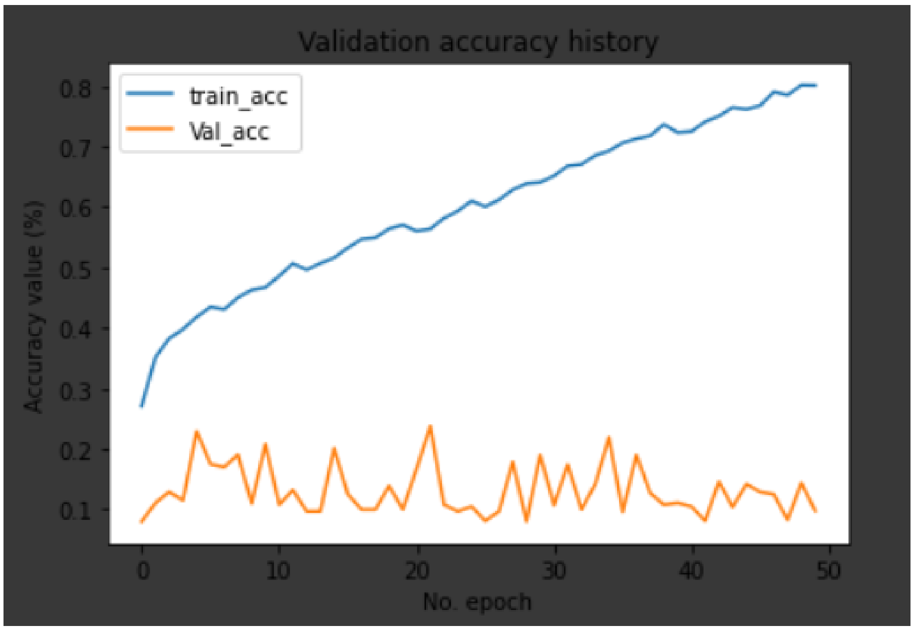


The above graphs show the output of the model when No Batch Normalization for our data set with the batch size is 64 and the epoch is 100.

4.4 Discussion on Results:

One of the main things that we first notice is thatthe model ***overfitted*** the data since the training set constantly eliminates loss while the test set is nearly constant after the 20-30th epoch and even getting higher.

The second thing that I noticed what that when we used Adam() optimizer, the training accuracy of the model was very good but the validation accuracy was too and was constant. Hence, I chose RMSprop.



Adam optimizer is used

The third thing we can notice is that our model was able to get near 70-80 & for the CIFAR-10 but not for our dataset. The reason was, the dataset size, we had 5000 images for training and out of that 1000 for training, therefore, our model had got enough data to be able to predict more precisely.

We also noticed that the initial accuracy after the first training epoch is significantly higher for model with Batch Normalization versus without Batch Normalization, which is probably a factor that defines how high our accuracy could get if we have a larger dataset for more training.

4.5 The complexity of the Algorithm:

The main part of the classification program is the Convolution part. It takes most of the run time for training and validation etc.

We ac say that the Convolution Neural Network mainly works on two process

- Forward Propagation
- Backward Propagation

4.5.1 Forward Propagation:

We assume the input-vector can be described as:

$$x \in R^n$$

Where the first element is the bias unit: $x_0 = \ 1$

The input is treated in the same as any other activation matrix, and has the index: $x = a^{(0)}$.

The zeroth element, $a_0^{(0)}$ is as usual the bias unit with a value of 1.

About forward propagation, we can write:

$$z^{(k)} = \theta^{(k)} a^{(k-1)}$$

$$z^{(k)} \in R^{1 \ x \ \left(m | \theta^{(k)} \in R^{mxn}\right)}$$

$$a^{(k)} = g\left(z^{(k)}\right)$$

Where ***g(x)*** is the activation function which is evaluated elementwise. We, therefore, know that $a^{(k)}$ has the same dimensions as $z^{(k)}$.

We see that for each layer a matrix multiplication and an activation function are computed. We know from the previous essay that naive matrix multiplication has an asymptotic run-time of $O\left(n^3\right)$, and since $g(x)$ is an element-wise function, we know that it has a run-time of $O(n)$.

The total run-time therefore becomes:

$$O\left(n^4+n^2\right) \iff$$

$$O\left(n^4\right) \because \forall \geq 1 | n^4+n^2 \leq 2n^4$$

4.5.2 Backpropagation:

We can find the run-time complexity of backpropagation similarly.

We can safely ignore ∇_a as it will be in the order of 1:

$$time \nabla_a = k$$

This gives us:

$$time_{error} = \{^{\alpha.\beta|\nabla_a^{(L)} \in R^{\alpha x \quad \beta} \quad ifk=L}_{\left(\theta^{(k+1)^T}\delta^{(k+1)}\right)'\left(z^{(k)}\right)}$$

If we assume that there are n neurons in each layer, we find that:

$$n^2 \quad = \quad O\left(\alpha.\beta|\nabla_a^{(L)} \in R^{\alpha x \quad \beta}\ \right)$$

$$n^3 = O\left(\left(\theta^{(k+1)^T}\delta^{(k+1)}\right)'\left(z^{(k)}\right)\right)$$

The total run-time for the delta error then becomes:

$$O\left(time_{weights}\right) = n^2 + n^3\left(L-1\right)$$

If we assume there are n *layers as we did during forwarding propagation, we find:*

$$O\left(time_{error}\right) = n^4$$

To find all the weights between a layers, we get:

$$O\left(time_{weights}\right) = O\left(time_{error}\right) + n^3 = n^4$$

Plugging this into gradient descent we get:

$$time_{\text{gradient descent}} = n_{\text{gradient iterations}} \cdot time_{weights}$$

If we assume n gradient iterations:

$$O\left(time_{\text{gradient descent}}\right) = n \cdot n^4 = n^4$$

So by assuming that gradient descent runs for n iterations and that there are n layers each with n neurons as we did with forwarding propagation, we find the total run-time of backpropagation to be:

$$O\left(n^5\right)$$

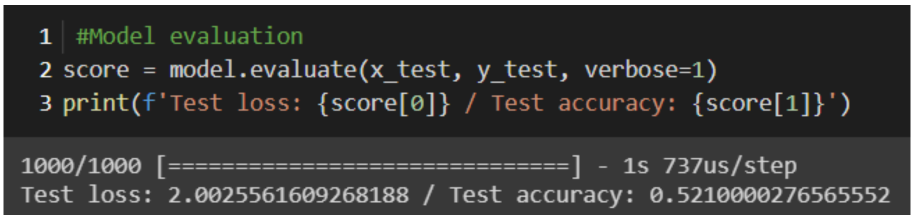
4.5.3 Conclusion:

We have derived the computational complexity of a feed-forward neural network, and seen why it's attractive to split the computation up in the training and an inference phase since backpropagation, $O\left(n^5\right)$

is much slower than the forward propagation, $O\left(n^4\right)$.

We have considered the large constant factor of gradient descent required to reach an acceptable accuracy which strengthens the argument.

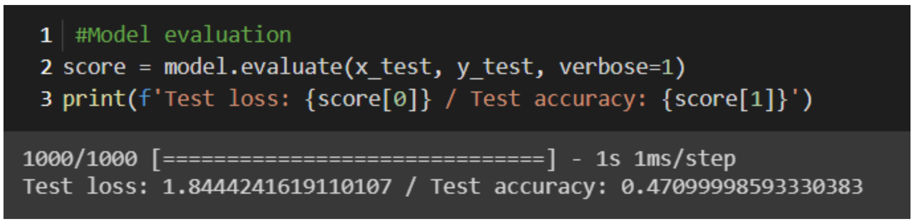
4.6 Snap Shots of results:



```
#Model evaluation
score = model.evaluate(x_test, y_test, verbose=1)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
```

```
1000/1000 [==============================] - 1s 737us/step
Test loss: 2.0025561609268188 / Test accuracy: 0.5210000276565552
```

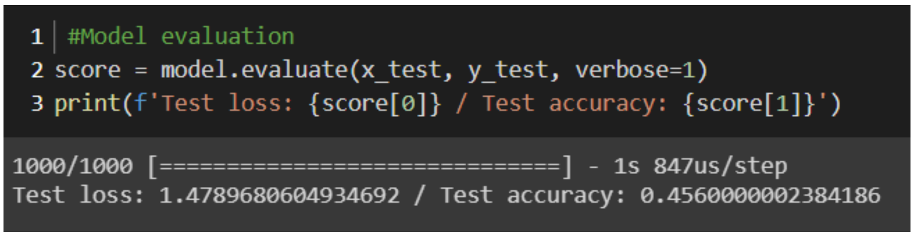
Test result for batch size = 32 and epochs = 50



```
#Model evaluation
score = model.evaluate(x_test, y_test, verbose=1)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
```

```
1000/1000 [==============================] - 1s 1ms/step
Test loss: 1.8444241619110107 / Test accuracy: 0.47099998593330383
```

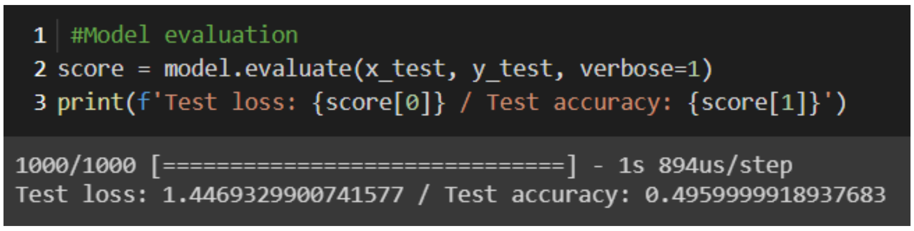
Test result for batch size = 64 and epochs = 100



```
#Model evaluation
score = model.evaluate(x_test, y_test, verbose=1)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
```

```
1000/1000 [==============================] - 1s 847us/step
Test loss: 1.4789680604934692 / Test accuracy: 0.4560000002384186
```

Test result for batch size = 64 and epochs = 50



```
#Model evaluation
score = model.evaluate(x_test, y_test, verbose=1)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
```

```
1000/1000 [==============================] - 1s 894us/step
Test loss: 1.4469329900741577 / Test accuracy: 0.4959999918937683
```

Test result for batch size = 64 and epochs = 100

CHAPTER 5: CONCLUSION AND FUTURE SCOPE

The main aim was to understand the basics of Computer Vision concepts and Convolutional Neural Network. Our ConvNet was able to achieve around 50-55% maximum accuracy due to various constraints. The model could be increased to achieve better accuracy. But in the end, the same concept is to be used.

The future scope is that with more hardware resources and dataset availability we can achieve higher accuracy.

References

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. Neural Computation, In Massachusetts Institutue of Technology, 1989.

[2] Daniel Svozil, Vladimir KvasniEka, Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. Department of Mathematics, Faculty of Chemical Technology, Slovak Technical University, Bratislava, SK-81237, Slovakia, 6th June 1997.

[3] Dan C. Cires¸an, Ueli Meier, Jonathan Masci, Luca M. Gambardella, Jurgen Schmidhuber. Flexible, High Performance Convolutional Neural Networks for Image Classification. IDSIA, USI and SUPSI Galleria 2, 6928 Manno-Lugano, Switzerland, 2011.

[4] Andrew G. Howard. Some Improvements on Deep Convolutional Neural Network Based Image Classification. Andrew Howard Consulting Ventura, CA 93003, 2013.

[5] Tianjun Xiao, Yichong Xu, Kuiyuan Yang, Jiaxing Zhang2, Yuxin Peng, Zheng Zhang. The Application of Two-level Attention Models in Deep Convolutional Neural Network for Finegrained Image Classification. Institute of Computer Science and Technology, Peking University, Microsoft Research, Beijing, New York University Shanghai, 2015.

[6] Kaiming He, Jian Sun. Convolutional Neural Networks at Constrained Time Cost. Microsoft Research, 2015.

[7] Antonio Gulli, Sujit Pal. Deep Learning with Keras. First published April 2017.

[8] Jonathan Griffin, Andrea Ramirez. Convolutional Neural Networks for Eye Tracking Algorithm. Stanford University.

[9] https://keras.io/

[10] https://kasperfred.com/series/introduction-to-neural-networks/how-does-backpropagation-work