Performance Evaluation of Convolution Neural Networks on classifying Image data

[1]Anuradha Surabhi, [2]Lakshman Kumar Bendpudi
[1]Assistant Professor [2]Consultant
[1]Department of Computer Science and Engineering,
[1]Neil Gogte Institute of Technology, Hyderabad, Telangana, India.

Abstract: Classification of image data is more enthusiastic and challenging data mining technique. In this study, we used supervised learning mechanism, classification on color image data which consists of huge set of samples. We experiment on input data with a sequence of convolution and max pooling layers with a sequence of filters in each layers to study the local pattern matching of images. Then it is compared with a fully connected Feed Forward Neural Network architecture to better understand the performance of CNN. Convolutional Neural Networks is a Deep learning algorithm which can take image data as input, process, and classify it under certain categories.

IndexTerms - **conv2D, dense, Deep learning, flatten, image classification, keras, pooling, padding, strides, Tensor flow.**

I. INTRODUCTION

An image is represented by its resolution hxwxd(h-Height, w-Width, d-Dimension). For example an image with 5x5x3 is an array of matrix of RGB(3 referes to RGB value) and an image of 5x5x1 is an array of matrix of grayscale image. Deep learning CNN models trained by passing input image through a series of convolution layers with filters, Pooling, Fully connected layers and a Softmax function is applied to classify an image object with probabilistic values between 0 and 1. The complete flow of CNN to process and classify an input image object is given in the below figure 1.

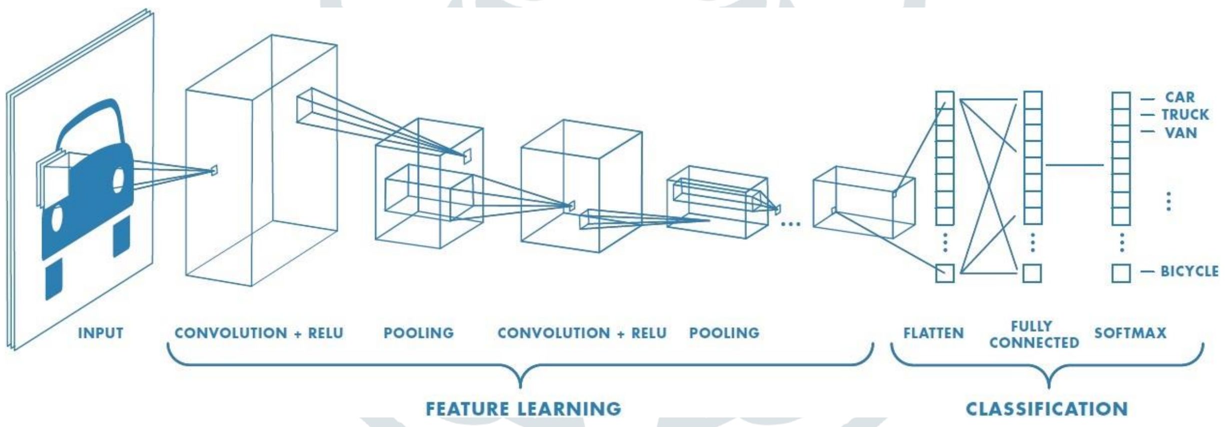


Figure 1: Convolution Neural Network to process and classify an image data

II. EXPERIMENTAL PROCEDURE

Input Dataset
CIFAR10[1-3] is a huge data set of 60000[4] color images each of size 32x32x3 and 10 different classes. Each image is one of the 10 different classes. Among this, 50000 samples were used for training the model and the remaining 10000 samples were used for testing the classification accuracy of the CNN model.

Experimental Setup
Experiment is done in Colab notebook with Tensorflow backend.

Importing Tensorflow

```
from __future__ import absolute_import, division, print_function, unicode_literals
!pip install tensorflow-gpu==2.0.0-beta1
from tensorflow import keras
from tensorflow.keras import datasets, layers, models
import datetime, os
import tensorflow as tf
```

Importing necessary libraries

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
% matplotlib inline
from keras.models import Sequential
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.layers import Activation, Flatten, Dense, Dropout
from keras.layers.normalization import BatchNormalization
from keras.utils import np_utils
```

Importing dataset

Cross validation is the general strategy used to split the data into training and test datasets. Cifar10 data set is available with splitting of 50000 samples for training and 10000 samples for testing. We only required loading the data set from tensorflow keras dataset

```
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.cifar10.
load_data()
```

Here, train_images is a tensor of shape (50000,32,32,3) which represents that 50000 images each of size 32x32x3 (Figure 3 (a)). train_labels is a tensor of shape (50000,1) which represents that 50000 images each with one class attribute. Similarly test_images and test_labels are also tensors of shapes (10000,32,32,3) and (10000,1) respectively. We can see the sample images of each of the 10 classes from figure2.

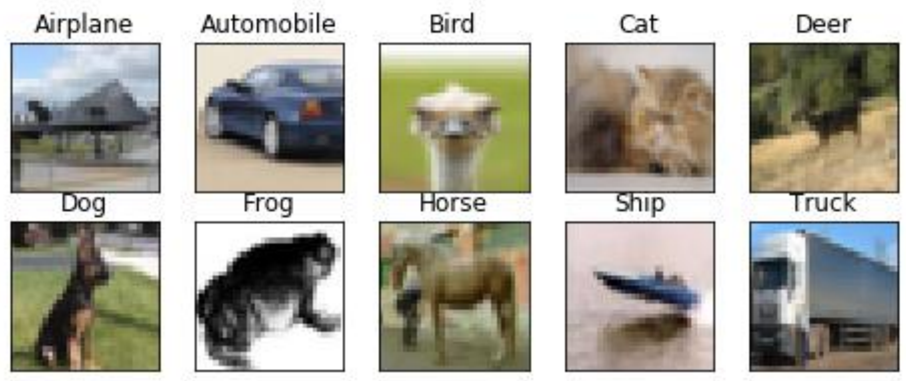


Figure 2: sample images of each class

Data Preprocessing

We have to normalize the pixel values of the image to be in between 0 and 1, thus image data is being pre-processed.

```
train_images = train_images / 255
test_images = test_images / 255
```

Then the class labels of both training and test data samples are converted into binary class labels

```
train_labels = np_utils.to_categorical(train_labels, num_classes)
test_labels = np_utils.to_categorical(test_labels, num_classes)
```

Now the convolutional base model is constructed with a stack of Conv2D and MaxPooling2D layers. CNN takes an input tensor of shape (iamge_height, image_width, color_channels), we do this by passing the argument input_shape to the first layer. Convolution operation involves a filter which captures local pattern and applied it on the image. The filter is a3D tensor of specific height, width and depth. Each entry in the filter is the real number. The entries in the filter are learned during CNN transition. The filter slides across width, height and depth starting at all possible positions through a local 3D patch of surrounding features. In case of color images, for every image we have 3 channels: red, green and blue. We are taking a standard 3x3 filter with 3 channels (3x3x3). We are transforming each patch with convolution filter into a number or 1D array. To understand what happens when we position the filter on a particular patch of an image, the weights (w1,w2……w9) in the 9 positions of the filter mapped into a patch of image with 9 input values (x1,x2….x9). The linear combination of weights with input values;

w1x1+w1x2+…………..+w9x9, and a bias term b, will be added. When we apply 'relu' activation on this linear combination which will gives a scalar value for a given positioning of a patch on the image.

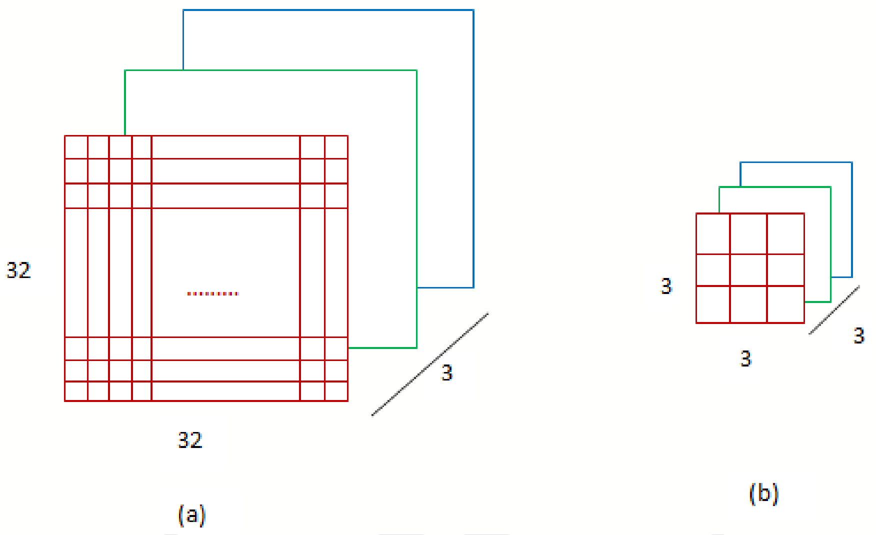


Figure 3: (a) 32x 32 images with 3 channels

(b) 3x3 filter with 3 channels

In convolution operation, we define a bunch of filters and each filter is used to calculate activation at a particular point in the image. To slide each filter across image, strides will be used. The output of convolution operation is usually smaller than the size of the input image. In order to keep the output of the same shape as the input we use padding. Apart from convolution the second important operation in CNN is called pooling. Pooling aggressively down sample the output of convolution and stride helps us to calculate the next position to place the filter across each axis starting from the current position. The default value for stride is 1, we can specify different stride across each axis but, that is used quite rarely. Stride convolution down sample the size by the factor proportional to stride. With a 32x32 image and a 3x3 filter, stride=1 shifts filter to the right by one column, thus it will continue till the last column, once finished it will shift downwards by one row. This is how we slide the filter across the image and try to match the pattern in the image.

We have 32X32x3 image and 3X3X3 filter (figure 3). Then we will be able to position the filter at 30 possible positions along the width as well as height. Thus 30X30x1 is the output of the convolution (see Table 1). In convolution we use k different filters. All these filters can be defined with conv2D layer. In conv2D (no_of_filters, filter_size, activaion_function, input_shape. depth size is usually not mentioned in the filter_size. After applying the convolution, we observed that the input get shrunk by some amount (see Table 1). If we don't want the input get shrunk, we use padding by adding some dummy columns and rows for the input and then apply convolution on it. For a convolution of 3X3 filter, we add a dummy column to the left and right and a dummy row to the top and bottom of the image, which ensure that the shape of the output is same as the shape of the input.

Model building

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.Conv2D(32, (3, 3), activation='relu',))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu',))
model.add(layers.Conv2D(64, (3, 3), activation='relu',))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu',))
model.add(layers.Conv2D(128, (3, 3), activation='relu',))
```

Table 1: Model summary with a sequence of convolution and max pooling layers

```
Model: "sequential"
_______________________________________________________________
Layer (type)                 Output Shape              Param #
===============================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_______________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        9248
_______________________________________________________________
```

```
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 12, 64)        18496
_______________________________________________________________
conv2d_3 (Conv2D)            (None, 10, 10, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________

conv2d_4 (Conv2D)            (None, 3, 3, 128)         73856

_________________________________________________________________

conv2d_5 (Conv2D)            (None, 1, 1, 128)         147584
=================================================================
Total params: 287,008
Trainable params: 287,008
Non-trainable params: 0
```

In the above, you can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as we go deeper in the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64 or 128). Typically, as the width and height shrink, we can afford computationally to add more output channels in each Conv2D layer.

To complete our model, we will feed the last output tensor from the convolution base of shape (1, 1, 128)) into one or more Dense layers[3] to perform classification. Dense layers take vectors as inputs which are 1D, while the current output is a 3D tensor (see Table 2). First, we will flatten or unroll the 3D output to 1D, then add one or more dense layers on top. CIFAR10 has 10 output classes, so we use a final dense layer with 10 outputs and 'softmax' activation.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Table 2: Summary of the complete CNN model

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 12, 64)        18496
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 10, 10, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 3, 3, 128)         73856
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 1, 128)         147584
_________________________________________________________________
flatten (Flatten)            (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 64)                8256
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 295,914
Trainable params: 295,914
Non-trainable params: 0

_________________________________________________________________
```

Now the model is compiled with adam optimizer, categorical_ crossentropy loss function, and accuracy as a metric

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

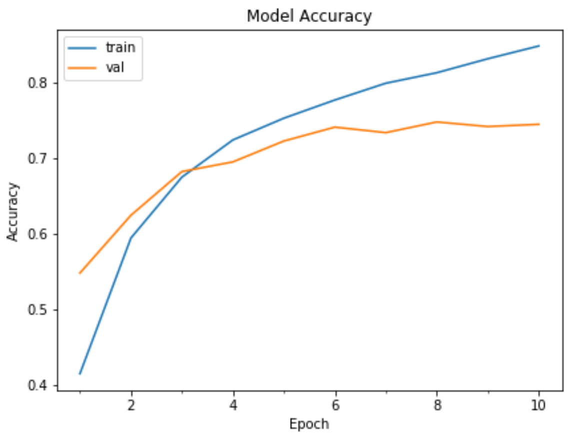
Then the model is trained for 10 epochs and evaluated with test data to calculate test_loss and test_accuracy.

```
history = model.fit(train_images, train_labels,validation_data = (test_images, test_la
bels), epochs=10)

test_loss, test_acc = model.evaluate(test_images, test_labels)
print("Test Accuracy : %.2f" %(test_acc*100})
print("Test Loss     : %.2f" %(test_loss*100))
```

The results obtained are:

```
10000/10000  [==============================]  -  1s  70us/sample  -  loss:  0.8186  -
accuracy: 0.7449
Test Accuracy :  74.48
Test Loss     :  81.86
```



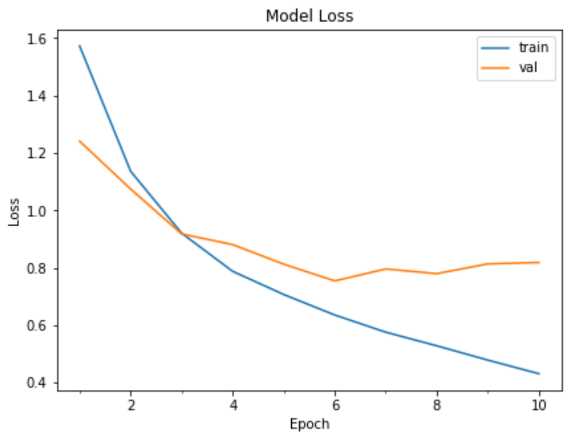


Figure 4: (a) Model Accuracy w.r.t. Epochs (b) Model Loss w.r.t. Epochs

III. Classification with Fully Connected Feed Forward Neural Network

With an interest this architecture is compared with a fully connected feed forward neural network with a hidden layer of 128 neurons. In this architecture we observed 394k parameter which is high in number as compared with 295K parameters that we obtained in CNN architecture.

```
model_ffnn = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(32, 32, 3)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')])


model_ffnn.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Table 3: Summary of the fully connected FFNN model

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               393344
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290
=================================================================
Total params: 394,634
Trainable params: 394,634
Non-trainable params: 0
_________________________________________________________________
```

```
history = model_ffnn.fit(x_train, y_train, validation_data = (x_test, y_test),
epochs=10)
test_loss,test_acc = model_ffnn.evaluate(x_test, y_test)
print("Test Accuracy : %.2f" %(test_acc*100))
print("Test Loss : %.2f" %(test_loss*100))
```

```
Test Accuracy : 37.72
Test Loss : 173.60
```

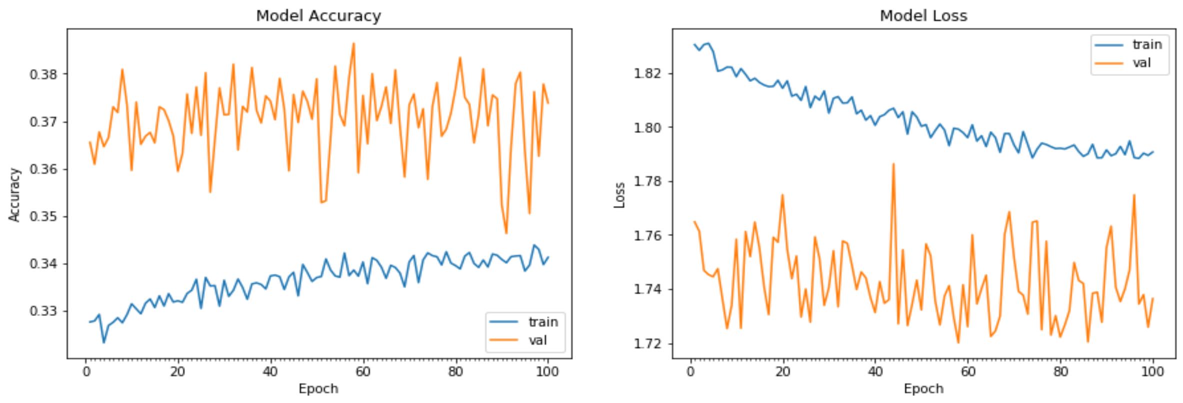


Figure 5: (a) Model Accuracy with fully connected FFNN
(b) Model Loss with fully connected FFNN

IV. RESULTS & DISCUSSION

In this work we achieved best classification results with CNN rather than a classic fully connected Feed Forward Neural Network (Figure 4 and Figure 5). The number of parameters is also reduced with CNN architecture. In case of CNN we are defining patches, and we are taking an image patch and performing a convolution operation with a filter. Thus we perform a linear combination of each position in the image with each parameter in the filter. We performed linear combination followed by non-linear activation. While in case of FFNN we take an entire image, flatten it which gives an array which is then passed through a hidden layer with 128 neurons and then it is given to a dense layer of 10 output neurons (Table 3) which will give output. We observed that CNN proceeds by capturing local patterns whereas in a fully connected FFNN, we learn that global patterning is involved.

REFERENCES

[1] Lee, Chen-Yu, Patrick W. Gallagher, and Zhuowen Tu. "Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree." Artificial Intelligence and Statistics. 2016.
[2] Springenberg, Jost Tobias, et al. "Striving for simplicity: The all convolutional net." arXiv preprint arXiv:1412.6806 (2014).
Basu, S. 1997. The Investment Performance of Common Stocks in Relation to their Price to Earnings Ratio: A Test of the Efficient Markets Hypothesis. Journal of Finance, 33(3): 663-682.
[3] Krizhevsky, Alex, and G. Hinton. "Convolutional deep belief networks on cifar-10." Unpublished manuscript 40 (2010): 7.
[4] Krizhevsky, Alex, Vinod Nair, and Geoffrey Hinton. "The CIFAR-10 dataset." online: http://www.cs.toronto.edu/kriz/cifar.html (2014).