ACADEMY OF SKILLS AND DEVELOPMENT

(Industrial training and Internship)

PROJECT: Image-classification

**BATCH NAME: VT-ON-ADSM-B4-APR**

**TOPIC: DATA SCIENCE AIML USING PYTHON**

PRESENTED BY: PARSHANT KUMAR

INSTITUTE: Indian Institute of Information Technology, Guwahati (CSE)

**INTRODUCTION**

Image classification is a fundamental task in computer vision that involves assigning labels or categories to image based on their visual content. I play a crucial role in various fields, including healthcare, autonomous driving, security systems, and e-commerce. Accurately classifying images enables us to automate tasks, extract meaningful insights, and make informed decisions based on visual data.

We are surrounded by an immense amount of visual information. From social media platforms to surveillance cameras, images are being generated and shared at an unprecedented rate. Image classification algorithms leverage the power of machine learning and deep learning to analyse and understand these images, enabling computers to recognize objects scenes or patterns.

**OBJECTIVE**

Objective of this project is to create a machine learning model using CNN to predict the image’s content.

**Why CNN over another algorithm?**

* **Spatial Hierarchical Feature learning**: CNNs are specially designed to capture spatial dependencies and hierarchical features in images. They consist of convolutional layers that apply filters across the input image, enabling the network to automatically learn relevant local patterns, edges and textures.
* **Parameter Sharing:** CNNs employ parameters sharing, which significantly reduces the number of parameters compared to fully connected networks. By sharing weights across different regions of the input image, CNNs achieve a form of translational invariance. This property enables the network to recognize patterns regardless of their specific location in the image, making CNNs more robust to variations in position, scales and orientation.
* **Local Receptive Fields:** CNNs use local receptive fields, meaning each neuron in a given layer is connected to only a small region of the previous layer. This local connectivity helps preserve spatial information while reducing computational complexity. By focusing on local regions, CNNs can capture local patterns and gradually combine them to learn more complex and abstract features.
* **Pooling Layers:** CNNs often incorporate pooling layers to downsample features maps, reducing the spatial dimensions. Pooling helps retain the most salient features while discarding irrelevant details, making the network more robust to small translations and variations. Additionally, pooling reduces the computational burden by decreasing the number of parameters in subsequent layers.
* **Pretrained Models and Transfer Learning:** CNNs have been extensively pretrained on large-scale image datasets, such as ImageNet, with millions of labelled images. These pretrained models, such as VGG16, ResNet, or Inception, have learned a diverse set of features from different image categories. By leaving transfer learning, you can initialize your CNN model with these pretrained weights and fine-tune it on your specific image classification task. This approach often leads to faster convergence and improved performance, particularly when you have limited labelled data.
* **Efficiency and Scalability:** CNNs are highly efficient and scalable for image processing tasks. They can process images parallel, making them suitable for implementation on GPUs, which excel in parallel computations. This parallelization allows CNNs to handle large-scale datasets and significantly reduce training time compared to other models.
* **Automatic Feature Extraction:** One of the major advantages of CNNs is their ability to automatically extract relevant features from raw image data. Traditional machine learning algorithms typically require manual feature engineering, where domain knowledge and expertise are needed to design and select appropriate features. In contrast, CNNs learn hierarchical representations of features directly from the input data, eliminating the need for manual feature extraction.
* **Ability to Learn Complex Features:** CNNs excel at learning complex and abstract features by stacking multiple convolutional layers. These layers progressively capture higher-level features as the network goes deeper. This depth allows CNNs to learn intricate representations of objects, textures, and shapes, enabling them to handle sophisticated classification tasks that may involve fine-grained details or subtle difference between classes.

**In conclusion, Convolutional Neural Networks (CNNs) widely used for image classification projects due to their ability to effectively capture spatial dependencies, extract relevant features, handle high-dimensional data, and achieve high accuracy. Their hierarchical feature learning, translation invariance, and scalability make them valuable tools for various image-related tasks. With their numerous advantages, CNNs have become a crucial component in the field of machine learning of image classification using frameworks like TensorFlow.**

**METHODOLOGY AND RESOURCES USED**

* **LIBRARIES**

[NumPy](https://numpy.org/doc/stable/reference/index.html), [pandas](https://pandas.pydata.org/docs/reference/index.html), [matplotlib](https://matplotlib.org/stable/api/index.html), [sklearn](https://scikit-learn.org/stable/modules/classes.html)

* **FRAMEWORK**

[TENSORFLOW.KERAS](https://www.tensorflow.org/api_docs/python/tf)

**MODULES**

[matplotlib.pyplot](https://matplotlib.org/stable/api/pyplot_summary.html), %matplotlib inline

[sklearn.metrics](https://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics)

[tensorflow.keras.utils](https://www.tensorflow.org/api_docs/python/tf)

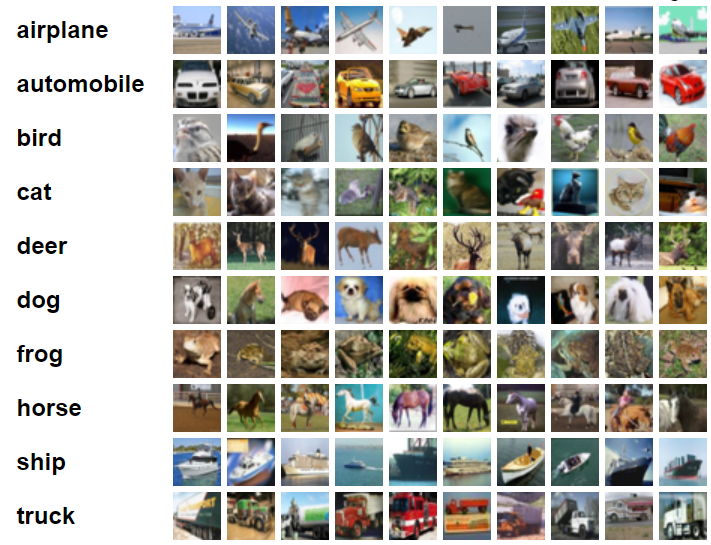
[tensorflow.keras.models](https://www.tensorflow.org/api_docs/python/tf)

[tensorflow.keras.layers](https://www.tensorflow.org/api_docs/python/tf)

[tensorflow.keras.preprocessing.image](https://www.tensorflow.org/api_docs/python/tf)

* **DATASETS**

[Cifar-10](https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)



Cifar-10 dataset contains distinct images of 10 types (as shown above).

It contains 60,000 images available in arrays format.

**TENSORFLOW AND MODEL ARCHITECTURE**

In this project, we leverage the capabilities of TensorFlow to build our image classification system. TensorFlow allows us to efficiently handle the complexities of deep learning architectures and provides a high-level interface for implementing and training neural networks.

The model architecture we employed for image classification is based on convolutional neural networks (CNNs). CNNs are widely used in computer vision tasks, including image classification, due to their ability to automatically learn hierarchical representations from images.

Our model consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers. The convolutional layers are responsible for extracting meaningful features from the input images by convolving filters over the image grid. The pooling layers help reduce the spatial dimensions and capture important information. The fully connected layers, also known as dense layers, take the extracted features and map them to the different image categories present in the CIFAR-10 dataset.

The CIFAR-10 dataset is a popular benchmark dataset in computer vision. It consists of 60,000 32x32 color images belonging to ten different classes, such as airplanes, cars, cats, and dogs. This dataset provides a diverse range of images for training and evaluating our image classification model.

By utilizing TensorFlow and implementing a suitable CNN architecture, we can effectively train our model on the CIFAR-10 dataset. Through an iterative training process, the model learns to optimize its parameters, adjusting the weights and biases to minimize the classification and improves its accuracy on unseen images.

**MODEL TRAINING AND EVALUATION**

We are creating the multiple convolutional layers which includes the pooling layers in between and also the dropout layers.

In convolutional layers, there are 32 filters and we are using “activation = ‘relu’” as the activation to make the model inclined towards the aspect of input data to be non-linear.

And the input shape is (32 , 32, 3), here the 32 x 32 is the image size in pixels and 3 is the color channels.

We are applying the pooling layers in between the convolutional layers which helps to reduce the dimensions of the data so the kernel can do less computations.

In pooling layer, we are using a pool size of (2 x 2).

These layers are introduced in between the convolutional layers in order to make to model learn on every possible concise data.

We are dropping the data also so that the model doesn’t overfit the data. It is a good practice to resolve the issue of regularization. So, we are using the dropout layers for the regularization.

And the remaining, fully connected layers are basically the deep neural network which is applied on a very small data or so as the CNN have done the majority portion of learning.

Here is the code for the training the model using CNN.

INPUT\_SHAPE = (32, 32, 3)

KERNEL\_SIZE = (3, 3)

model = Sequential()

# Convolutional Layer

model.add(Conv2D(filters=32, kernel\_size=KERNEL\_SIZE, input\_shape=INPUT\_SHAPE, activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(Conv2D(filters=32, kernel\_size=KERNEL\_SIZE, input\_shape=INPUT\_SHAPE, activation='relu', padding='same'))

model.add(BatchNormalization())

# Pooling layer

model.add(MaxPool2D(pool\_size=(2, 2)))

# Dropout layers

model.add(Dropout(0.25))

model.add(Conv2D(filters=64, kernel\_size=KERNEL\_SIZE, input\_shape=INPUT\_SHAPE, activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(Conv2D(filters=64, kernel\_size=KERNEL\_SIZE, input\_shape=INPUT\_SHAPE, activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(MaxPool2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(filters=128, kernel\_size=KERNEL\_SIZE, input\_shape=INPUT\_SHAPE, activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(Conv2D(filters=128, kernel\_size=KERNEL\_SIZE, input\_shape=INPUT\_SHAPE, activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(MaxPool2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

# model.add(Dropout(0.2))

model.add(Dense(128, activation='relu'))

model.add(Dropout(0.25))

model.add(Dense(10, activation='softmax'))

METRICS = [

    'accuracy',

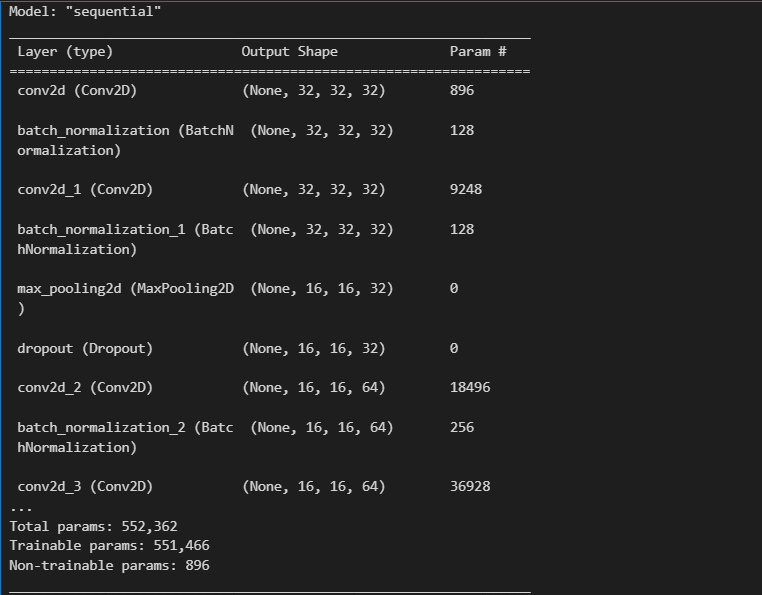
    tf.keras.metrics.Precision(name='precision'),

    tf.keras.metrics.Recall(name='recall')

]

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=METRICS)

**MODEL’S SUMMARY :-**



The batch size(number of samples that will be processed in each training iteration), model is using is 32 .

We are using the ImageDataGenerator() class from the keras.preprocessing module. It is used to perform the real time data augmentation on images during training.

Next data\_generator.flow() is used to generate augmented image data in batches. It takes the training images and their corresponding labels as inputs, along with the defined batch size (which is 32 in this case).

Now, after all these steps we have to fit the data in the model in order to make the model learn from it and it’s done using the fit() method.

batch\_size = 32

data\_generator = ImageDataGenerator(width\_shift\_range=0.1, height\_shift\_range=0.1,horizontal\_flip=True)

train\_generator = data\_generator.flow(X\_train, y\_cat\_train, batch\_size)

steps\_per\_epoch = X\_train.shape[0] // batch\_size

r = model.fit(train\_generator, epochs=20, steps\_per\_epoch=steps\_per\_epoch, validation\_data=(X\_test, y\_cat\_test))

Epoch 1/20

1562/1562 [==============================] - 106s 66ms/step - loss: 1.6122 - accuracy: 0.4181 - precision: 0.6287 - recall: 0.2102 - val\_loss: 1.2116 - val\_accuracy: 0.5668 - val\_precision: 0.7252 - val\_recall: 0.4154

Epoch 2/20

1562/1562 [==============================] - 120s 77ms/step - loss: 1.2101 - accuracy: 0.5749 - precision: 0.7306 - recall: 0.4145 - val\_loss: 1.6112 - val\_accuracy: 0.5207 - val\_precision: 0.5816 - val\_recall: 0.4441

Epoch 3/20

1562/1562 [==============================] - 144s 92ms/step - loss: 1.0167 - accuracy: 0.6459 - precision: 0.7767 - recall: 0.5205 - val\_loss: 0.9030 - val\_accuracy: 0.6945 - val\_precision: 0.7980 - val\_recall: 0.6011

Epoch 4/20

1562/1562 [==============================] - 121s 77ms/step - loss: 0.9243 - accuracy: 0.6827 - precision: 0.7951 - recall: 0.5690 - val\_loss: 1.0230 - val\_accuracy: 0.6775 - val\_precision: 0.7434 - val\_recall: 0.6071

Epoch 5/20

1562/1562 [==============================] - 106s 68ms/step - loss: 0.8490 - accuracy: 0.7111 - precision: 0.8129 - recall: 0.6123 - val\_loss: 0.7182 - val\_accuracy: 0.7523 - val\_precision: 0.8346 - val\_recall: 0.6784

Epoch 6/20

1562/1562 [==============================] - 124s 80ms/step - loss: 0.7920 - accuracy: 0.7306 - precision: 0.8253 - recall: 0.6425 - val\_loss: 0.6655 - val\_accuracy: 0.7762 - val\_precision: 0.8605 - val\_recall: 0.6949

Epoch 7/20

1562/1562 [==============================] - 156s 100ms/step - loss: 0.7485 - accuracy: 0.7470 - precision: 0.8336 - recall: 0.6645 - val\_loss: 0.6916 - val\_accuracy: 0.7743 - val\_precision: 0.8376 - val\_recall: 0.7237

Epoch 8/20

1562/1562 [==============================] - 205s 131ms/step - loss: 0.7179 - accuracy: 0.7567 - precision: 0.8384 - recall: 0.6818 - val\_loss: 0.6461 - val\_accuracy: 0.7846 - val\_precision: 0.8452 - val\_recall: 0.7266

Epoch 9/20

1562/1562 [==============================] - 107s 68ms/step - loss: 0.6840 - accuracy: 0.7691 - precision: 0.8467 - recall: 0.6974 - val\_loss: 0.6716 - val\_accuracy: 0.7786 - val\_precision: 0.8426 - val\_recall: 0.7214

Epoch 10/20

1562/1562 [==============================] - 119s 76ms/step - loss: 0.6553 - accuracy: 0.7771 - precision: 0.8514 - recall: 0.7092 - val\_loss: 0.7011 - val\_accuracy: 0.7692 - val\_precision: 0.8311 - val\_recall: 0.7226

Epoch 11/20

1562/1562 [==============================] - 131s 84ms/step - loss: 0.6364 - accuracy: 0.7837 - precision: 0.8546 - recall: 0.7202 - val\_loss: 0.5675 - val\_accuracy: 0.8124 - val\_precision: 0.8709 - val\_recall: 0.7571

Epoch 12/20

1562/1562 [==============================] - 139s 89ms/step - loss: 0.6107 - accuracy: 0.7919 - precision: 0.8602 - recall: 0.7318 - val\_loss: 0.6458 - val\_accuracy: 0.7856 - val\_precision: 0.8417 - val\_recall: 0.7393

Epoch 13/20

...

Epoch 19/20

1562/1562 [==============================] - 133s 85ms/step - loss: 0.5066 - accuracy: 0.8293 - precision: 0.8819 - recall: 0.7816 - val\_loss: 0.4860 - val\_accuracy: 0.8410 - val\_precision: 0.8811 - val\_recall: 0.8036

Epoch 20/20

1562/1562 [==============================] - 106s 68ms/step - loss: 0.5018 - accuracy: 0.8295 - precision: 0.8810 - recall: 0.7821 - val\_loss: 0.4657 - val\_accuracy: 0.8459 - val\_precision: 0.8849 - val\_recall: 0.8079

Here the model is being trained for 20 epochs and the,

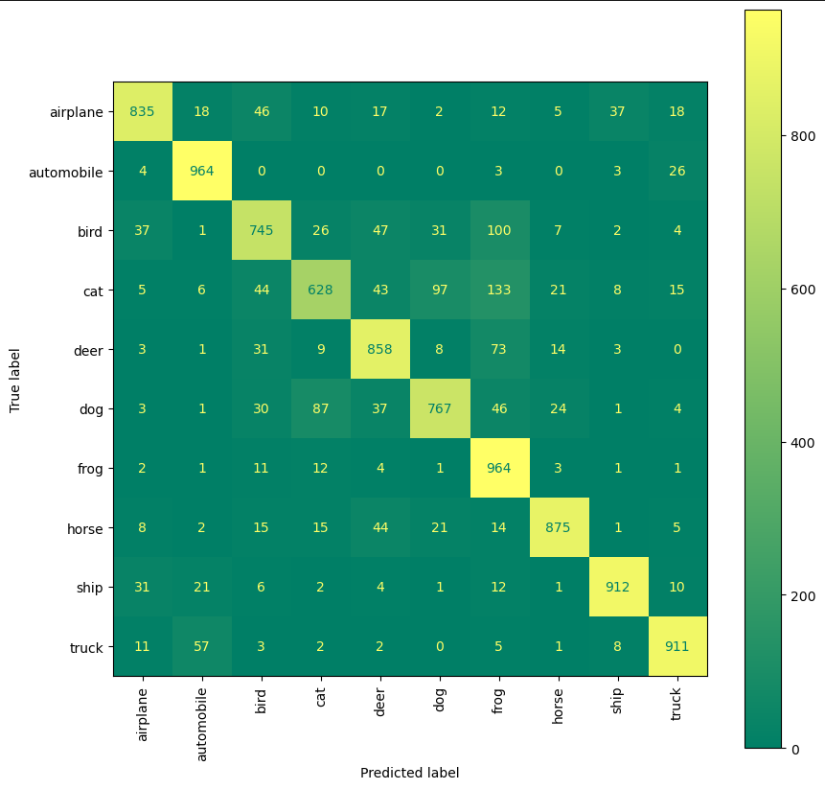
**accuracy: 0.8295 ; val\_accuracy : 0.8459**

**precision: 0.8810 ; val\_precision : 0.8849**

**loss: 0.5018 ; val\_loss : 0.4657**

**EVALUATION**

Let us see how the model predicted the images using confusion matrix.



In the above confusion matrix, the x-axis is the predicted results and the y-axis is the actual image. The model was provided 1000 samples as test datasets and the confusion matrix shows how much the model have predicted correctly. The diagonal with yellowish colour tells us the correct predictions.

**For example, if we take airplane as the input test-sample, 835 were predicted accurately and the rest are predicted wrong like 18 were predicted automobile which were actually airplane.**

predictions = model.predict(X\_test)

num\_rows = 8

num\_cols = 5

num\_images = num\_rows \* num\_cols

plt.figure(figsize=(2 \* 2 \* num\_cols, 2 \* num\_rows))

print("Predicted : red\t Actual : blue")

for i in range(num\_images):

    plt.subplot(num\_rows, 2 \* num\_cols, 2 \* i + 1)

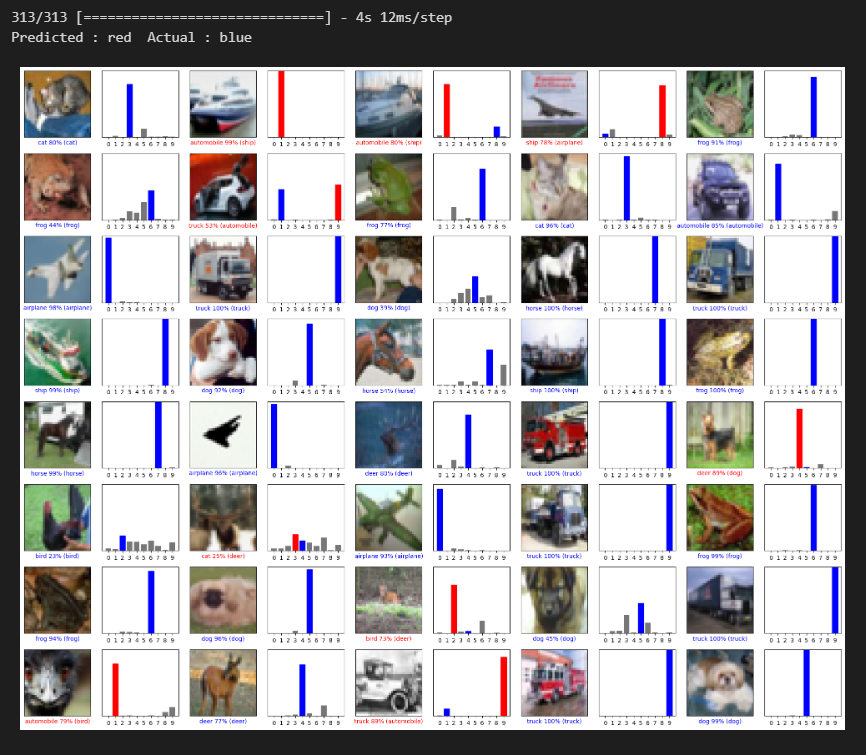
    plot\_image(i, predictions[i], y\_test, X\_test)

    plt.subplot(num\_rows, 2\*num\_cols, 2\*i+2)

    plot\_value\_array(i, predictions[i], y\_test)

plt.tight\_layout()

plt.show()



In the above output the **red** is the prediction of the image and **blue** shows what the actual image was.

**CONCLUSION**

The model focused on to predict the image given it as input.

Model’s training is done on cifar-10 datasets and the accuracy it achieved is 82.95% accuracy while exploring the potential of deep learning and the convolutional neural networks (CNN).

The dataset’s diversity and complexity tested the robustness of our model and allowed to gain a deeper understanding of real-world image classification scenarios.

By leveraging TensorFlow and implementing a CNN architecture, we were able to build a robust model that accurately classified unseen images.

**APPLICATION:** Accurate image classification has the potential to revolutionize fields such as healthcare, where it can assist in the diagnosing medical conditions from medical imaging data.

Image classification can be used in e-commerce product categorization.

**ACKNOWLEDGEMENT**

**Mentor:** **Shouvik sarkar**

**Organisation: Academy of Skills Development**

**Datasets:** [**cifar-10**](https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)