AD18511 – DEEP LEARNING LABORATORY

DATE:

EX.NO: 10 **IMPLEMENTATION OF VGG ARCHITECTURE.**

AIM:

To use the VGG (Visual Geometric group) architecture to CIFAR-10 datasets using Tensorflow.

DESCRIPTION:

- VGG stands for Visual Geometry Group; it is a standard deep Convolutional Neural Network (CNN) architecture with multiple layers.
- The ―deep‖ refers to the number of layers with VGG-16 or VGG-19 consisting of 16 and 19 convolutional layers.
- The VGG architecture is the basis of ground-breaking object recognition models. Developed as a deep neural network, the VGGNet also surpasses baselines on many tasks and datasets beyond ImageNet. Moreover, it is now still one of the most popular image recognition architectures.
- The VGG network is constructed with very small convolutional filters. The VGG-16 consists of 13 convolutional layers and three fully connected layers.
- Input:The VGGNet takes in an image input size of 224×224
- Convolutional Layers: VGG's convolutional layers leverage a minimal receptive field, i.e., 3×3, the smallest possible size that still captures up/down and left/right.
- Hidden Layers: All the hidden layers in the VGG network use ReLU. VGG does not usually leverage Local Response Normalization (LRN) as it increases memory consumption and training time.
- Fully-Connected Layers: The VGGNet has three fully connected layers. Out of the three layers, the first two have 4096 channels each, and the third has 1000 channels, 1 for each class.

PROGRAM:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import numpy as np

# Load and preprocess CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
train_images, test_images = train_images / 255.0, test_images / 255.0
```

OUTPUT:

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

170498071/170498071 [==============================] - 14s 0us/step

```
model = models.Sequential()
model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False)) # Add BatchNormalization layer
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
```

```
model.add(layers.Conv2D(128, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False)) # Add BatchNormalization layer
model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(256, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False)) # Add BatchNormalization layer
model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(512, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False)) # Add BatchNormalization layer
model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(512, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False)) # Add BatchNormalization layer
model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(4096, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(4096, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))

# creating the model

model.summary()
```

OUTPUT:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 64)        1792

 batch_normalization (Batch  (None, 32, 32, 64)        192
 Normalization)

 conv2d_1 (Conv2D)           (None, 32, 32, 64)        36928

 max_pooling2d (MaxPooling2D)  (None, 16, 16, 64)      0


 conv2d_2 (Conv2D)           (None, 16, 16, 128)       73856

 batch_normalization_1 (Bat  (None, 16, 16, 128)       384
 chNormalization)

 conv2d_3 (Conv2D)           (None, 16, 16, 128)       147584
```

```
max_pooling2d_1 (MaxPoolin  (None, 8, 8, 128)        0
g2D)

conv2d_4 (Conv2D)           (None, 8, 8, 256)        295168

batch_normalization_2 (Bat  (None, 8, 8, 256)        768
chNormalization)

conv2d_5 (Conv2D)           (None, 8, 8, 256)        590080

max_pooling2d_2 (MaxPoolin  (None, 4, 4, 256)        0
g2D)

conv2d_6 (Conv2D)           (None, 4, 4, 512)        1180160

batch_normalization_3 (Bat  (None, 4, 4, 512)        1536
chNormalization)

conv2d_7 (Conv2D)           (None, 4, 4, 512)        2359808

max_pooling2d_3 (MaxPoolin  (None, 2, 2, 512)        0
g2D)

conv2d_8 (Conv2D)           (None, 2, 2, 512)        2359808

batch_normalization_4 (Bat  (None, 2, 2, 512)        1536
chNormalization)

conv2d_9 (Conv2D)           (None, 2, 2, 512)        2359808

max_pooling2d_4 (MaxPoolin  (None, 1, 1, 512)        0
g2D)

flatten (Flatten)           (None, 512)              0

dense (Dense)               (None, 4096)             2101248

dropout (Dropout)           (None, 4096)             0

dense_1 (Dense)             (None, 4096)             16781312

dropout_1 (Dropout)         (None, 4096)             0

dense_2 (Dense)             (None, 10)               40970

=================================================================
Total params: 28332938 (108.08 MB)
Trainable params: 28329994 (108.07 MB)
Non-trainable params: 2944 (11.50 KB)
_________________________________________________________________
```

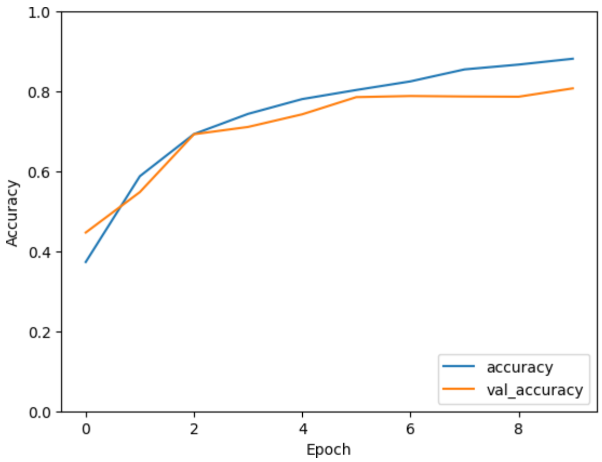
```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

OUTPUT:

```
Epoch 1/10
1563/1563 [==============================] - 57s 27ms/step - loss: 1.6695 - accuracy: 0.3731 - val_loss: 1.6695 - val_accuracy: 0.4469
Epoch 2/10
1563/1563 [==============================] - 39s 25ms/step - loss: 1.1939 - accuracy: 0.5878 - val_loss: 1.3521 - val_accuracy: 0.5481
Epoch 3/10
1563/1563 [==============================] - 40s 25ms/step - loss: 0.9247 - accuracy: 0.6933 - val_loss: 0.8926 - val_accuracy: 0.6929
Epoch 4/10
1563/1563 [==============================] - 40s 25ms/step - loss: 0.7902 - accuracy: 0.7438 - val_loss: 0.8881 - val_accuracy: 0.7111
Epoch 5/10
1563/1563 [==============================] - 40s 25ms/step - loss: 0.6856 - accuracy: 0.7808 - val_loss: 0.8698 - val_accuracy: 0.7427
Epoch 6/10
1563/1563 [==============================] - 40s 25ms/step - loss: 0.6300 - accuracy: 0.8035 - val_loss: 0.8138 - val_accuracy: 0.7857
Epoch 7/10
1563/1563 [==============================] - 41s 26ms/step - loss: 0.5603 - accuracy: 0.8251 - val_loss: 1.0362 - val_accuracy: 0.7885
Epoch 8/10
1563/1563 [==============================] - 45s 29ms/step - loss: 0.4660 - accuracy: 0.8551 - val_loss: 0.7054 - val_accuracy: 0.7873
Epoch 9/10
1563/1563 [==============================] - 39s 25ms/step - loss: 0.4313 - accuracy: 0.8671 - val_loss: 2.1532 - val_accuracy: 0.7867
Epoch 10/10
1563/1563 [==============================] - 39s 25ms/step - loss: 0.3916 - accuracy: 0.8817 - val_loss: 4.2583 - val_accuracy: 0.8076
```

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()
```

OUTPUT:




```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f"Test accuracy: {test_acc}")
```

OUTPUT:
313/313 - 2s - loss: 4.2583 - accuracy: 0.8076 - 2s/epoch - 6ms/step Test accuracy: 0.8076000213623047

```
# Make predictions
predictions = np.argmax(model.predict(test_images), axis=-1)
```

OUTPUT:
313/313 [==============================] - 2s 6ms/step

```
# Make predictions
predictions = np.argmax(model.predict(test_images), axis=-1)
# Generate classification report
report = classification_report(test_labels, predictions)

# Print confusion matrix and classification report
print("Confusion Matrix:\n", cm)
print("Classification Report:\n", report)
```

OUTPUT:

Confusion Matrix:

```
 [[825 8 43 37 10 3 7 26 6 35]
 [ 7 923 8 3 2 0 5 3 1 48]
 [ 46   0 655 31 76 81 56 43 0 12]
 [  4   3  29 549 35 258 62 44 2 14]
 [  4   1  16 20 795 39 25 97 1 2]
 [  1   1   9 73 19 838 7 50 0  2]
 [  2   1  28 33 19 16 886 10 2 3]
 [  4   1   4 12 14 26 1 931 0  7]
 [ 79  52 10 11 4 2 17 10 748 67]
 [ 8   26 2 5 1 3 4 14 11 926]]
```

Classification Report:

	precision	recall	f1-score	support
0	0.84	0.82	0.83	1000
1	0.91	0.92	0.92	1000
2	0.81	0.66	0.73	1000
3	0.71	0.55	0.62	1000
4	0.82	0.80	0.81	1000
5	0.66	0.84	0.74	1000
6	0.83	0.89	0.86	1000
7	0.76	0.93	0.84	1000
8	0.97	0.75	0.84	1000
9	0.83	0.93	0.88	1000
accuracy			0.81	10000
macro avg	0.81	0.81	0.81	10000
weighted avg	0.81	0.81	0.81	10000

RESULT:

Thus the VGG architecture has been implemented successfully.