| **Practical Evaluation Task**  To study and implement a Convolutional Neural Network (CNN) for image classification on the CIFAR-100 dataset with real world examples. |
| --- |

| **Problem Description:** The objective is to design and implement a Convolutional Neural Network (CNN) for image classification using the CIFAR-100 dataset. After loading and normalizing the dataset, data augmentation techniques like rotations, shifts and flips are applied to enhance diversity, which helps the model generalize better. During training, the model’s accuracy is monitored and expected to improve with each epoch. By the end, the model should demonstrate robust accuracy when tested on unseen images, showing its effectiveness in real-world image classification tasks. |
| --- |

| **Code:**  import pandas as pd  import numpy as np  import matplotlib.pyplot as plt  from pylab import rcParams  import tensorflow as tf  from tensorflow.keras.datasets import cifar100  from tensorflow.keras.models import Sequential  from tensorflow.keras.layers import Conv2D, MaxPool2D, Dropout, Flatten, Dense, GlobalAveragePooling2D, BatchNormalization, InputLayer  from tensorflow.keras.preprocessing.image import ImageDataGenerator  from tensorflow.keras.optimizers import Adam  from tensorflow.keras.callbacks import Callback, EarlyStopping, ReduceLROnPlateau  from tensorflow.keras.utils import to\_categorical  from tensorflow.keras.applications.efficientnet import EfficientNetB0  from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report, ConfusionMatrixDisplay  from skimage.transform import resize  from sklearn.model\_selection import train\_test\_split, StratifiedShuffleSplit  import seaborn as sns  import cv2  import albumentations as albu  (x\_train, y\_train), (x\_test, y\_test) = cifar100.load\_data(label\_mode='fine')  # Flatten image data for easier DataFrame handling (reshape to 1D for each image)  x\_train\_flat = x\_train.reshape(x\_train.shape[0], -1)  x\_test\_flat = x\_test.reshape(x\_test.shape[0], -1)  # Load CIFAR-100 class names (fine labels)  fine\_labels = [  "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"  ]  x\_train.shape, x\_test.shape, y\_train.shape, y\_test.shape  n\_classes = 100  y\_train = to\_categorical(y\_train, n\_classes)  y\_test = to\_categorical(y\_test, n\_classes)  x\_train\_data, x\_val\_data, y\_train\_data, y\_val\_data = train\_test\_split(x\_train, y\_train, test\_size=0.2, random\_state=123, stratify=y\_train)  print("Number of training samples: ", x\_train\_data.shape[0])  print("Number of validation samples: ", x\_val\_data.shape[0])  # EfficientNetB0, the images need to be resized to the size (224, 224)  height = 224  width = 224  channels = 3  n\_classes = 100  input\_shape = (height, width, channels)  epochs = 15  batch\_size = 16  def resize\_img(img, shape):  return cv2.resize(img, (shape[1], shape[0]), interpolation=cv2.INTER\_CUBIC)  class DataGenerator(tf.keras.utils.Sequence):  def \_\_init\_\_(self, images, labels=None, mode='fit', batch\_size=batch\_size, dim=(height, width), channels=channels, n\_classes=n\_classes, shuffle=True, augment=False):  #initializing the configuration of the generator  self.images = images  self.labels = labels  self.mode = mode  self.batch\_size = batch\_size  self.dim = dim  self.channels = channels  self.n\_classes = n\_classes  self.shuffle = shuffle  self.augment = augment  self.on\_epoch\_end()  super().\_\_init\_\_()  #method to be called after every epoch  def on\_epoch\_end(self):  self.indexes = np.arange(self.images.shape[0])  if self.shuffle == True:  np.random.shuffle(self.indexes)  #return numbers of steps in an epoch using samples and batch size  def \_\_len\_\_(self):  return int(np.floor(len(self.images) / self.batch\_size))  #this method is called with the batch number as an argument to obtain a given batch of data  def \_\_getitem\_\_(self, index):  #generate one batch of data  #generate indexes of batch  batch\_indexes = self.indexes[index \* self.batch\_size:(index+1) \* self.batch\_size]  #generate mini-batch of X  X = np.empty((self.batch\_size, \*self.dim, self.channels), dtype=np.float32)  for i, ID in enumerate(batch\_indexes):  #generate pre-processed image  img = self.images[ID]  #image rescaling  img = img.astype(np.float32)/255.0  #resizing as per new dimensions  img = resize\_img(img, self.dim)  X[i] = img  #generate mini-batch of y  if self.mode == 'fit':  y = self.labels[batch\_indexes]  #augmentation on the training dataset  if self.augment == True:  X = self.\_\_augment\_batch(X)  return X, y  elif self.mode == 'predict':  return (X,)  else:  raise AttributeError("The mode should be set to either 'fit' or 'predict'.")  #augmentation for one image  def \_\_random\_transform(self, img):  composition = albu.Compose([albu.HorizontalFlip(p=0.5),  albu.VerticalFlip(p=0.5),  albu.GridDistortion(p=0.2),  albu.ElasticTransform(p=0.2)])  return composition(image=img)['image']  #augmentation for batch of images  def \_\_augment\_batch(self, img\_batch):  for i in range(img\_batch.shape[0]):  img\_batch[i] = self.\_\_random\_transform(img\_batch[i])  return img\_batch  train\_data\_generator = DataGenerator(x\_train\_data, y\_train\_data, augment=True)  valid\_data\_generator = DataGenerator(x\_val\_data, y\_val\_data, augment=False)  efnb0 = EfficientNetB0(weights='imagenet', include\_top=False, input\_shape=input\_shape, classes=n\_classes)  model = Sequential()  model.add(InputLayer(input\_shape=input\_shape))  model.add(efnb0)  model.add(GlobalAveragePooling2D())  model.add(Dropout(0.5))  model.add(Dense(n\_classes, activation='softmax'))  model.summary()  optimizer = Adam(learning\_rate=0.0001)  # early stopping  early\_stopping = EarlyStopping(monitor='val\_accuracy', mode='max', verbose=1, patience=10, restore\_best\_weights=True)  # reducing learining rate on plateau  rlrop = ReduceLROnPlateau(monitor='val\_loss', mode='min', patience=5, factor=0.5, min\_lr=1e-6, verbose=1)  model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy'])  model\_history = model.fit(train\_data\_generator,  validation\_data=valid\_data\_generator,  callbacks=[early\_stopping, rlrop],  verbose=1,  epochs=epochs)  #saving the trained model as data file in .h5 format  model.save('cifar\_100\_efficientnetb0\_model.keras')  plt.figure(figsize=(18,8))  plt.suptitle('Loss and Accuracy Plots', fontsize=18)  plt.subplot(1,2,2)  plt.plot(model\_history.history['accuracy'], label='Train Accuracy')  plt.plot(model\_history.history['val\_accuracy'], label='Validation Accuracy')  plt.legend()  plt.xlabel('Number of epochs', fontsize=14)  plt.ylabel('Accuracy', fontsize=14)  plt.show()  y\_pred = model.predict(DataGenerator(x\_test, mode='predict', augment=False, shuffle=False), verbose=1)  y\_pred = np.argmax(y\_pred, axis=1)  test\_accuracy = accuracy\_score(np.argmax(y\_test, axis=1), y\_pred)  print('Test Accuracy: ', round((test\_accuracy \* 100), 2), "%")  def preprocess\_image(image\_path):  image = Image.open(image\_path).resize((32, 32))  image = image.convert('RGB')  image = np.array(image) / 255.0 # Normalize the image  image = np.expand\_dims(image, axis=0) # Add batch dimension  return image  def predict\_image\_label(image\_path):  image = preprocess\_image(image\_path)  predictions = model.predict(image)  predicted\_label = class\_names[np.argmax(predictions)]  confidence = np.max(predictions) # Get the confidence of the prediction  return predicted\_label, confidence  # Get the number of uploaded images  num\_images = len(uploaded)  # Calculate grid dimensions (e.g., 2 rows if 2-4 images, 3 rows if 5-9 images, etc.)  num\_rows = int(np.ceil(np.sqrt(num\_images)))  num\_cols = int(np.ceil(num\_images / num\_rows))  # Create a figure and subplots  fig, axes = plt.subplots(num\_rows, num\_cols, figsize=(12, 12))  fig.subplots\_adjust(hspace=0.5) # Adjust spacing between subplots  # Flatten the axes array for easier indexing  axes = axes.flatten()  # Iterate through uploaded images and display them  for i, file\_name in enumerate(uploaded.keys()):  predicted\_label, confidence = predict\_image\_label(file\_name)  print(f"Predicted Label for {file\_name}: {predicted\_label} (Confidence: {confidence:.2f})")  # Display image on the corresponding subplot  img = Image.open(file\_name)  axes[i].imshow(img)  axes[i].set\_title(f"Predicted: {predicted\_label} ({confidence:.2f})")  axes[i].axis('off')  # Hide any unused subplots  for j in range(num\_images, num\_rows \* num\_cols):  axes[j].axis('off')  """Function to resize the image"""  def resize\_test\_image(test\_img):  img = cv2.imread(test\_img)  #plt.imshow(img)  img\_RGB = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)  #plt.imshow(img\_RGB)  resized\_img = cv2.resize(img\_RGB, (224, 224))  #plt.imshow(resized\_img)  resized\_img = resized\_img / 255.0  # plt.imshow(resized\_img)  return resized\_img  def predict\_test\_image(test\_img):  resized\_img = resize\_test\_image(test\_img)  prediction = model.predict(np.array([resized\_img]))  return prediction  def sort\_prediction\_test\_image(test\_img):  prediction = predict\_test\_image(test\_img)  index = np.arange(0,100)  for i in range(100):  for j in range(100):  if prediction[0][index[i]] > prediction[0][index[j]]:  temp = index[i]  index[i] = index[j]  index[j] = temp  return index  def df\_top5\_prediction\_test\_image(test\_img):  sorted\_index = sort\_prediction\_test\_image(test\_img)  prediction = predict\_test\_image(test\_img)  class\_name = []  prediction\_score = []  k = sorted\_index[:5]  for i in range(5):  class\_name.append(fine\_labels[k[i]])  prediction\_score.append(prediction[0][k[i]])  df = pd.DataFrame(list(zip(class\_name, prediction\_score)), columns=['Label', 'Probability'])  return df  def df\_top5\_prediction\_test\_image(test\_img):  prediction = predict\_test\_image(test\_img)  sorted\_index = sort\_prediction\_test\_image(test\_img)  class\_name = []  prediction\_score = []  k = sorted\_index[:5]  for i in range(5):  class\_name.append(fine\_labels[k[i]])  prediction\_score.append(prediction[0][k[i]])  df = pd.DataFrame(list(zip(class\_name, prediction\_score)), columns=['Label', 'Probability'])  return df  plot\_top5\_prediction\_test\_image('src/orange.jpeg')  plot\_top5\_prediction\_test\_image('src/cloud.jpeg')  plot\_top5\_prediction\_test\_image('src/can.jpg')  plot\_top5\_prediction\_test\_image('src/house.jpg')  plot\_top5\_prediction\_test\_image('src/worm.jpeg') |
| --- |

| **Results:**  **→ Test Accuracy : 83.41 %**    **→ Predictions :** |
| --- |