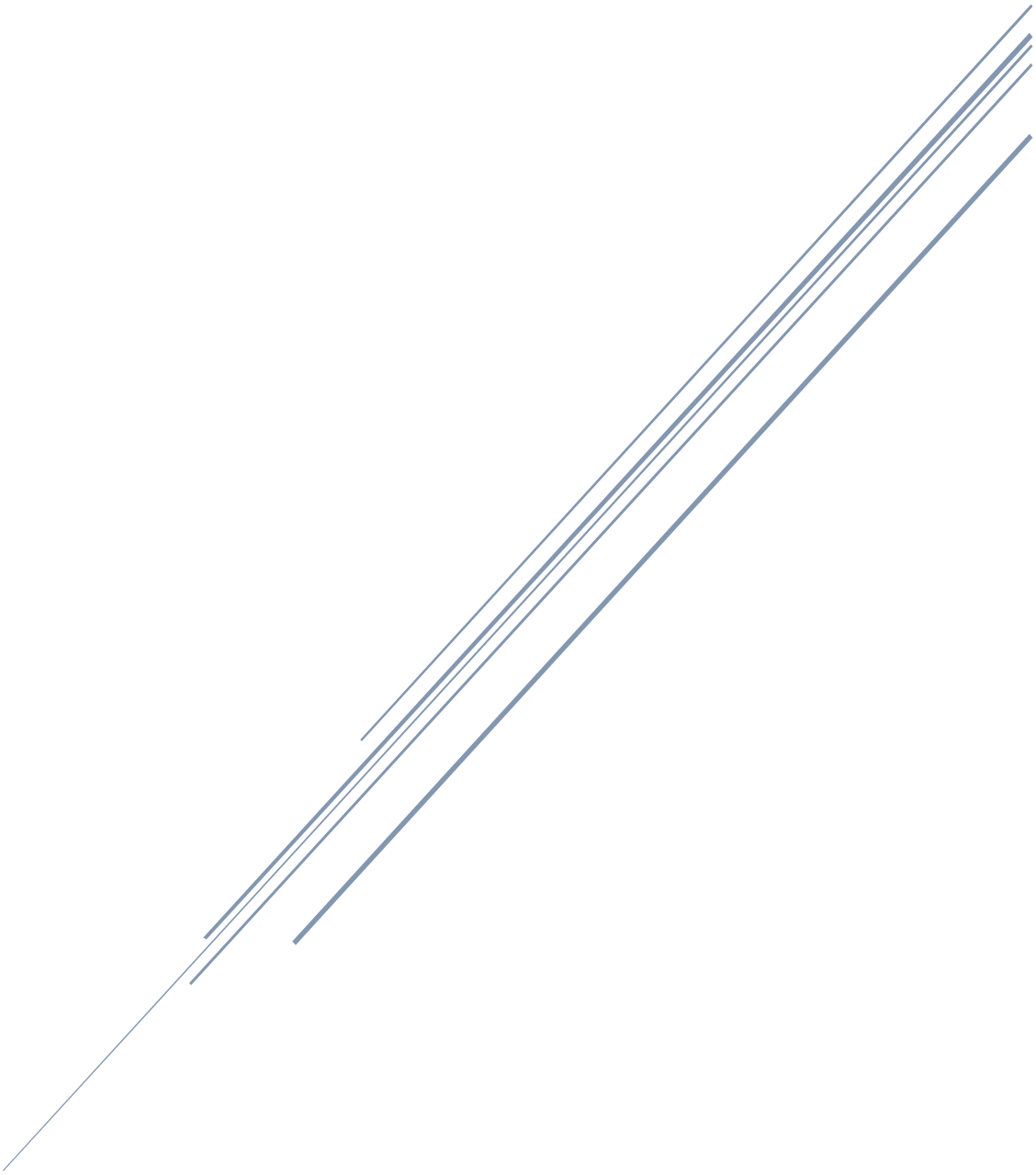
KNOWLEDGE TEST PRACTICAL DOCUMENTATION



Akash T
ADIT CALICUT

Activity 1

Aim

Build and compile a simple neural network using Keras to classify the MNIST dataset (handwritten digits). The model should include at least one hidden layer. Provide the code and briefly explain each step.

Software/Hardware Requirements

- Windows PC
- Python

Procedure

1. Import Libraries

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.utils import to_categorical
```

✓ 10.3s

- numpy: This library is used for numerical operations and handling arrays.
- keras.datasets.mnist: This module is used to load the MNIST dataset, which contains 60,000 training images and 10,000 testing images of handwritten digits.
- keras.models.Sequential: This class is used to create a linear stack of layers for the neural network model.
- keras.layers.Dense: This layer is a fully connected (dense) neural network layer.
- keras.layers.Flatten: This layer is used to flatten the input, transforming it from a 2D matrix into a 1D vector.
- keras.utils.to_categorical: This function converts integer labels to one-hot encoded labels, which is a common format for categorical classification tasks.

2. Load and Preprocess Data

```
# Load the dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

✓ 3.5s

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 3s 0us/step
```

```
# Normalize the images to a range of 0 to 1
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
```

✓ 0.2s

```
# Convert labels to categorical one-hot encoding
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

✓ 0.0s

- mnist.load_data(): This function loads the MNIST dataset and returns two tuples: (x_train, y_train) for training data and (x_test, y_test) for testing data. The images are 28x28 pixels and the labels are integers from 0 to 9.
- Normalization: The pixel values of the images are originally in the range 0 to 255. Dividing by 255 normalizes the values to the range 0 to 1, which helps in faster and more efficient training of the neural network.
- One-hot Encoding: The labels are converted to one-hot encoded format. For example, the label 3 is converted to [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

3. Build the Model

```
# 2. Build the model
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

- Sequential: This creates a linear stack of layers.
- Flatten: This layer flattens the input image from a 28x28 matrix to a 784-dimensional vector. This transformation allows the data to be fed into the dense layers.
- Dense (Hidden Layer): This layer has 128 neurons and uses the ReLU (Rectified Linear Unit) activation function. ReLU introduces non-linearity to the model, which helps it learn complex patterns.
- Dense (Output Layer): This layer has 10 neurons (one for each digit) and uses the softmax activation function. Softmax converts the output into probability distributions, which is useful for multi-class classification.

4. Compile the Model

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

✓ 0.0s
```

- optimizer='adam': Adam is an optimization algorithm that adjusts the learning rate during training. It is popular for its efficiency and performance.
- loss='categorical_crossentropy': This loss function is used for multi-class classification problems. It measures the difference between the true labels and the predicted labels.
- metrics=['accuracy']: This specifies that we want to track the accuracy of the model during training

5. Train the Model

```
# 4. Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

✓ 23.6s

```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8686 - loss: 0.4756 - val_accuracy: 0.9547 - val_loss: 0.1535
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9611 - loss: 0.1346 - val_accuracy: 0.9673 - val_loss: 0.1126
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9745 - loss: 0.0858 - val_accuracy: 0.9707 - val_loss: 0.0955
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9821 - loss: 0.0605 - val_accuracy: 0.9744 - val_loss: 0.0875
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9870 - loss: 0.0464 - val_accuracy: 0.9732 - val_loss: 0.0894
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9897 - loss: 0.0350 - val_accuracy: 0.9752 - val_loss: 0.0907
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9923 - loss: 0.0266 - val_accuracy: 0.9749 - val_loss: 0.0907
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9929 - loss: 0.0238 - val_accuracy: 0.9750 - val_loss: 0.0958
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9944 - loss: 0.0186 - val_accuracy: 0.9762 - val_loss: 0.0928
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9968 - loss: 0.0118 - val_accuracy: 0.9702 - val_loss: 0.1218

<keras.src.callbacks.history.History at 0x1c854794b30>
```

- x_train, y_train: The training data and labels.
- epochs=10: The number of times the entire training dataset will pass through the network. More epochs can lead to better accuracy but may also cause overfitting.
- batch_size=32: The number of samples that will be passed through the network at one time. Smaller batch sizes can lead to more accurate updates but take longer to train.
- validation_split=0.2: This specifies that 20% of the training data should be used as validation data. Validation data helps monitor the model's performance on unseen data during training.

6. Evaluate the Model

```
# 5. Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy}')
```

✓ 0.5s

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9665 - loss: 0.1298
Test accuracy: 0.97079998254776
```

- model.evaluate(x_test, y_test): This function evaluates the model on the test data and returns the loss and accuracy.
- print(f'Test accuracy: {test_accuracy}'): This prints the test accuracy, giving an indication of how well the model performs on new, unseen data.

Activity 2

Aim

Implement data augmentation on a given image dataset using Keras. Show at least three different augmentation techniques and explain how they help improve model performance.

Software/Hardware Requirements

- Windows PC
- Python

Procedure

Import Necessary Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Flatten, Dense
```

✓ 0.0s

- **Import essential libraries:** This section imports libraries needed for numerical operations (NumPy), plotting (Matplotlib), image preprocessing (ImageDataGenerator), loading datasets (mnist), creating neural networks (Sequential), and defining layers (Flatten, Dense).

Load and Preprocess Data

```
# Load and preprocess the data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

# Normalize the images to the range [0, 1]
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
```

✓ 0.4s

- Load MNIST dataset: Fetches the MNIST dataset containing handwritten digits.
- Reshape data: Adds an extra dimension to the image data for compatibility with the neural network.
- Normalize data: Scales pixel values to a range of 0 to 1 for better performance

Create Image Data Generator

```
# Define the ImageDataGenerator with augmentation parameters
datagen = ImageDataGenerator(
    rotation_range=30,
    zoom_range=0.2,
    horizontal_flip=True
)
```

✓ 0.0s

```
# Fit the data generator to the training data
datagen.fit(x_train)
```

✓ 0.0s

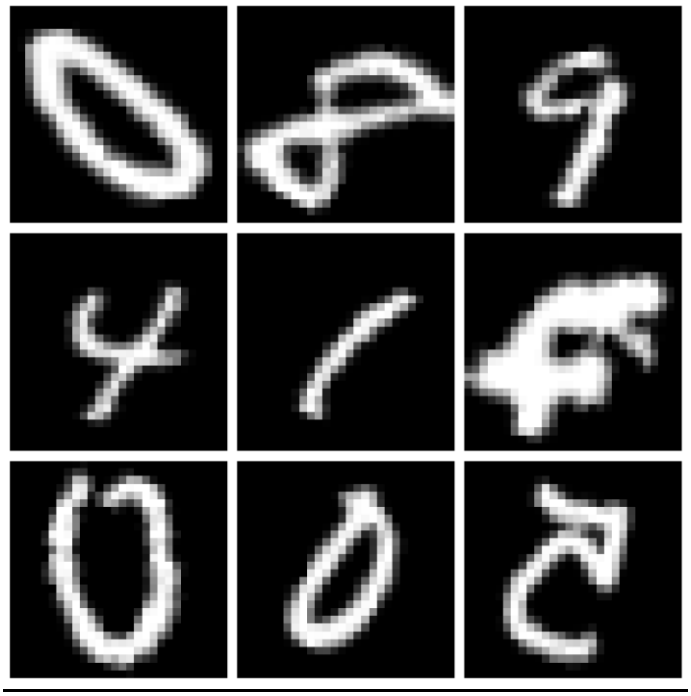
- Create data augmentation: Creates an object to generate augmented images by randomly rotating, zooming, and flipping images.

- Fit data generator: Adapts the augmentation parameters to the training data.

Define Function to Display Augmented Images

```python
# Display some augmented images
def plot_augmented_images(datagen, images):
    augmented_images = [next(datagen.flow(images, batch_size=1)) for _ in range(9)]
    fig, axes = plt.subplots(3, 3, figsize=(10, 10))
    axes = axes.flatten()
    for img, ax in zip(augmented_images, axes):
        ax.imshow(np.squeeze(img), cmap='gray')
        ax.axis('off')
    plt.tight_layout()
    plt.show()

# Display the augmented images
plot_augmented_images(datagen, x_train)
```



- Create a function: Defines a function to visualize the effects of data augmentation by displaying a grid of augmented images.

Create Neural Network Model

```
# Use the augmented data to train a model
model = Sequential([
    Flatten(input_shape=(28, 28, 1)),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```
✓ 0.0s

- Create model architecture: Builds a simple neural network with a flattening layer and two dense layers.
- Compile model: Configures the model with the Adam optimizer, sparse categorical crossentropy loss, and accuracy metric.

Train the Model

```
# Fit the model using the data generator
model.fit(datagen.flow(x_train, y_train, batch_size=32), epochs=10, validation_data=(x_test, y_test))
```
✓ 2m 10.1s

```
Epoch 1/10
   7/1875 ━━━━━━━━━━━━━━━━━━━━ 19s 10ms/step - accuracy: 0.1134 - loss: 2.3186
c:\Users\akash\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121:
  self._warn_if_super_not_called()
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - accuracy: 0.7330 - loss: 0.8456 - val_accuracy: 0.9172 - val_loss: 0.2690
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 12s 7ms/step - accuracy: 0.8897 - loss: 0.3644 - val_accuracy: 0.9362 - val_loss: 0.2017
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9115 - loss: 0.2922 - val_accuracy: 0.9449 - val_loss: 0.1812
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9199 - loss: 0.2621 - val_accuracy: 0.9482 - val_loss: 0.1671
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9251 - loss: 0.2425 - val_accuracy: 0.9546 - val_loss: 0.1433
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9308 - loss: 0.2275 - val_accuracy: 0.9554 - val_loss: 0.1407
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9370 - loss: 0.2064 - val_accuracy: 0.9548 - val_loss: 0.1372
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9364 - loss: 0.2087 - val_accuracy: 0.9581 - val_loss: 0.1297
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9400 - loss: 0.1967 - val_accuracy: 0.9586 - val_loss: 0.1320
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9412 - loss: 0.1979 - val_accuracy: 0.9617 - val_loss: 0.1223

<keras.src.callbacks.history.History at 0x2463da891c0>
```

- Train the model: Trains the neural network using the augmented data generated by the data generator, with 10 epochs and validation on the test set.

Improvements:

- Experiment with hyperparameters: While this code demonstrates the basic usage, you can improve the model's performance by trying different hyperparameters like the number of epochs, learning rate, or the number of neurons in the hidden layer.
- Additional Augmentation techniques: Consider adding more augmentation techniques like adding noise, shifting, or elastic transformations for even more diverse training data.
- Early stopping: Implementing early stopping can prevent overfitting by stopping the training if the validation accuracy doesn't improve for a certain number of epochs.
- More complex model architecture: For more complex datasets, consider exploring deeper convolutional neural network architectures like CNNs with convolutional and pooling layers.

Activity 3

Aim

Implement a custom loss function in TensorFlow/Keras. Explain the purpose of the loss function and provide an example scenario where it would be useful.

Software/Hardware Requirements

- Windows PC
- Python

Procedure

Import Necessary Libraries

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
```

✓ 4.8s

- Import TensorFlow: Imports the TensorFlow library, which is the core framework for building and training machine learning models.
- Import MNIST dataset: Imports functions to load the MNIST dataset, a standard dataset of handwritten digits.
- Import Sequential model: Imports the Sequential model class from Keras, used for creating layer-by-layer neural network models.
- Import Flatten and Dense layers: Imports the Flatten and Dense layer types for building the neural network.

Load and Preprocess Data

```
# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```
✓ 0.2s

```
# Preprocess data
x_train = x_train.reshape(-1, 28*28) / 255.0
x_test = x_test.reshape(-1, 28*28) / 255.0

# Convert class vectors to binary class matrices
num_classes = 10
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```
✓ 0.1s

- Load MNIST data: Loads the MNIST dataset into training and testing sets.
- Reshape data: Reshapes the image data from 28x28 matrices to a single vector of 784 pixels for each image.
- Normalize data: Scales pixel values to a range of 0 to 1.
- Convert labels: Converts class labels (digits 0-9) into one-hot encoded format for multi-class classification.

Define Focal Loss Function

```
# Define focal loss function
def focal_loss(gamma=2.0, alpha=0.25):
    def focal_loss_fixed(y_true, y_pred):
        pt_1 = tf.where(tf.equal(y_true, 1), y_pred, tf.ones_like(y_pred))
        pt_0 = tf.where(tf.equal(y_true, 0), y_pred, tf.zeros_like(y_pred))
        return -alpha * tf.pow(1.0 - pt_1, gamma) * tf.math.log(pt_1) \
               - (1 - alpha) * tf.pow(pt_0, gamma) * tf.math.log(1.0 - pt_0)
    return focal_loss_fixed
```
✓ 0.0s

- Define focal loss function: Creates a custom loss function called focal_loss with parameters gamma and alpha to control the focus on hard examples and class weighting.

Create Neural Network Model

```
# Create a simple model
model = Sequential([
    Dense(256, activation='relu', input_shape=(784,)),
    Dense(128, activation='relu'),
    Dense(num_classes, activation='softmax')
])
```

✓ 0.1s

Create model: Defines a sequential model with three layers:

- Input layer with 256 neurons and ReLU activation.
- Hidden layer with 128 neurons and ReLU activation.
- Output layer with 10 neurons (for 10 classes) and softmax activation for probability distribution.

Compile the Model

```
# Compile the model with focal loss
model.compile(loss=focal_loss(), optimizer='adam', metrics=['accuracy'])
```

✓ 0.0s

- Compile model: Configures the model with the defined focal loss function, Adam optimizer, and accuracy metric for evaluation.

Train the Model

```
# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))
```

✓ 40.4s

```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8952 - loss: 0.0070 - val_accuracy: 0.9623 - val_loss: 0.0025
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9698 - loss: 0.0021 - val_accuracy: 0.9731 - val_loss: 0.0019
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9786 - loss: 0.0015 - val_accuracy: 0.9741 - val_loss: 0.0017
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9810 - loss: 0.0012 - val_accuracy: 0.9779 - val_loss: 0.0017
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9876 - loss: 8.3027e-04 - val_accuracy: 0.9766 - val_loss: 0.0018
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9897 - loss: 7.4659e-04 - val_accuracy: 0.9796 - val_loss: 0.0016
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9900 - loss: 6.8916e-04 - val_accuracy: 0.9779 - val_loss: 0.0020
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9922 - loss: 5.6682e-04 - val_accuracy: 0.9811 - val_loss: 0.0017
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9936 - loss: 4.4842e-04 - val_accuracy: 0.9796 - val_loss: 0.0020
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9931 - loss: 4.6813e-04 - val_accuracy: 0.9812 - val_loss: 0.0019
```

- Train model: Trains the model on the training data for 10 epochs with a batch size of 32, using the test data for validation.

A **loss function** is a mathematical function that measures how well a machine learning model is performing for a given dataset. It quantifies the error between the model's predicted output and the actual ground truth.

Key purpose:

- **Evaluation:** It provides a metric to assess the model's performance.
- **Optimization:** It guides the model's learning process by calculating the gradient, which is used to update the model's parameters in order to minimize the loss.

Example Scenario: Image Classification with Imbalanced Dataset

Problem: Imagine a medical image classification task where you're trying to identify rare diseases. In this case, the dataset will be heavily imbalanced, with many more images of healthy patients than those with the rare disease.

Solution: Using a standard loss function like categorical cross-entropy might lead the model to prioritize accuracy on the majority class (healthy patients), ignoring the minority class (patients with the disease).

Activity 4

Aim

Use a pre-trained model (such as VGG16 or ResNet) available in Keras for a simple image classification task. Fine-tune the model for a new dataset and describe the steps taken.

Software/Hardware Requirements

- Windows PC
- Python

Procedure

Import Libraries

```
import tensorflow as tf
from tensorflow.keras.applications import VGG16
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
```

✓ 0.0s

- Import Libraries: Import TensorFlow, Keras, and other necessary libraries.

Load and Preprocess the Data

```
# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize the images to the range [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert class vectors to binary class matrices
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

✓ 56.9s

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 54s 0us/step
```

- Load the CIFAR-10 dataset and normalize the images.

Load the Pre-trained VGG16 Model

```
# Load the VGG16 model without the top layers
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Freeze the layers of the base model
for layer in base_model.layers:
    layer.trainable = False
```

✓ 0.1s

- Load the VGG16 model without the top layers and freeze its weights.

Add Custom Layers

```
# Add custom layers
x = base_model.output
x = Flatten()(x)
x = Dense(512, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# Create the final model
model = Model(inputs=base_model.input, outputs=predictions)
```

✓ 0.0s

- Add custom layers on top of the base model.

Compile the Model

```
model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
```

✓ 0.0s

+ Code + Markdown

- Compile the model with an appropriate optimizer and loss function.

Train the Model

```
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))
```

✓ 10m 9.5s

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 62s 39ms/step - accuracy: 0.4787 - loss: 1.4747 - val_accuracy: 0.5644 - val_loss: 1.2234
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 60s 38ms/step - accuracy: 0.5887 - loss: 1.1592 - val_accuracy: 0.5896 - val_loss: 1.1645
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 60s 38ms/step - accuracy: 0.6213 - loss: 1.0790 - val_accuracy: 0.5890 - val_loss: 1.1612
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 59s 38ms/step - accuracy: 0.6411 - loss: 1.0208 - val_accuracy: 0.6081 - val_loss: 1.1174
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 64s 41ms/step - accuracy: 0.6605 - loss: 0.9668 - val_accuracy: 0.6172 - val_loss: 1.0930
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 58s 37ms/step - accuracy: 0.6718 - loss: 0.9255 - val_accuracy: 0.6195 - val_loss: 1.1011
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 62s 40ms/step - accuracy: 0.6946 - loss: 0.8747 - val_accuracy: 0.6088 - val_loss: 1.1394
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 60s 38ms/step - accuracy: 0.7045 - loss: 0.8397 - val_accuracy: 0.6104 - val_loss: 1.1395
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 64s 41ms/step - accuracy: 0.7225 - loss: 0.7831 - val_accuracy: 0.6172 - val_loss: 1.1522
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 59s 38ms/step - accuracy: 0.7366 - loss: 0.7540 - val_accuracy: 0.6186 - val_loss: 1.1569

<keras.src.callbacks.history.History at 0x2dd66d9a930>
```

- Train the model on the CIFAR-10 dataset.

Evaluate the Model

```
loss, accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {accuracy}')
```

✓ 9.9s

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 31ms/step - accuracy: 0.6166 - loss: 1.1521
Test accuracy: 0.6186000108718872
```

- Evaluate the model's performance on the test set.