21CSE06 NEURAL NETWORKS AND DEEP LEARNING

Ex.1 Implement simple vector addition in TensorFlow

AIM:

To implement simple vector addition in TensorFlow.

ALGORITHM:

- Import the TensorFlow library and alias it as "tf.".
- Create two constant tensors, vector1 and vector2, representing the two vectors you want to add.
- Use tf.add to perform the vector addition by adding vector1 and vector2, storing the result in the result` tensor.
- Start a TensorFlow session using a with block to manage the session's lifecycle. The tf.Session() context allows you to perform computations within TensorFlow.
- Inside the session, use sess.run(result) to execute the computation and calculate the sum of the two vectors.
- Store the result in the output variable.
- Print the result to the console, which represents the vector addition.

CODING:

```
import tensorflow as tf
# Create two constant tensors
vector1 = tf.constant([1.0, 2.0, 3.0])
vector2 = tf.constant([4.0, 5.0, 6.0])
```

```
# Perform vector addition
result = tf.add(vector1, vector2)
# Start a TensorFlow session
with tf.Session() as sess:
# Run the session to compute the result
output = sess.run(result)
print(output)
```

OUTPUT:

```
[5. 7. 9.]
```

RESULT:

Thus, implementing simple vector addition in TensorFlow is successfully executed and verified.

Ex.2 **Implement a regression model in Keras.**

AIM:

To implement a regression model in Keras.

ALGORITHM:

Import the necessary libraries:

- Import the NumPy library as np.
- Import TensorFlow and its Keras submodules.

Generate example data for regression:

- Create a feature matrix X with shape (100, 1) using NumPy, containing random values.
- Create target values y by applying a linear relationship with some noise.

Define a sequential model:

- Create a sequential model using keras.Sequential(). This sets up a feedforward neural network with a sequential structure.

Add a dense layer:

- Add a single dense layer to the model using model.add(layers.Dense(1, input_shape=(1,)). This layer has one output unit for regression and expects one input feature.

Compile the model:

- Compile the model with specific settings.
- Use the stochastic gradient descent (SGD) optimizer by specifying optimizer='sgd'.
- Use the mean squared error (MSE) loss function for regression by specifying loss='mean_squared_error'.

Train the model:

- Train the model using the fit method.
- Provide the feature matrix X and target values y.
- Set the number of training epochs to 100 using epochs=100.
- Set verbose to 1 to see training progress during each epoch.

Make predictions:

- Use the predict method on the trained model to make predictions on the same input data X.

Evaluate the model (optional):

- If needed, you can evaluate the model's performance using the evaluate method.
- Calculate the loss (MSE) by evaluating the model on the same input data X and target values y.
- Print the MSE to assess the model's performance.

Print the model's summary:

- Use the summary method to print a summary of the model's architecture, including the layers and the number of parameters.

CODE:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
# Generate some example data for regression
X = np.random.rand(100, 1)
y = 2 * X + 1 + 0.1 * np.random.randn(100, 1)
```

```
# Define a sequential model
model = keras.Sequential()
# Add a single dense layer with one output unit (for regression)
model.add(layers.Dense(1, input_shape=(1,)))
# Compile the model
model.compile(optimizer='sgd', loss='mean_squared_error')
# Train the model
model.fit(X, y, epochs=100, verbose=1)
# Make predictions
predictions = model.predict(X)
# Evaluate the model if needed
loss = model.evaluate(X, y)
print(f"Mean Squared Error: {loss}")
# Print the model's summary
model.summary()
```

<u>OUTPUT:</u>

```
Epoch 1/100
100/100 [==============================] - 0s 3ms/step - loss: 7.2826
...
Epoch 100/100
100/100 [==============================] - 0s 62us/step - loss: 0.0816
32/32 [==============================] - 0s 245us/step - loss: 0.0833
Mean Squared Error: 0.08327607876014709
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
=
dense (Dense)               (None, 1)                 2
=================================================================
=
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

<u>RESULT:</u>

Thus, implementing a regression model in Keras is successfully executed and verified.

Ex.3 Implement a perceptron in TensorFlow/Keras Environment.

Aim:

To implement a perception in TensorFlow/Keras Environment.

ALGORITHM:

Import the necessary libraries:

- Import the NumPy library as np.
- Import TensorFlow and its Keras submodules.

Generate example data for a logical OR operation:

- Create a NumPy array X to represent the input features. It contains all possible combinations of two binary values (0 and 1).
- Create another NumPy array y to represent the target values, which correspond to the logical OR operation's output.

Define a sequential model:

- Create a sequential model using keras.Sequential(). This sets up a feedforward neural network with a sequential structure.

Add a single dense layer (perceptron):

- Add a single dense layer to the model using model.add(Dense(1, input_shape=(2,), activation='sigmoid'). This layer has one output unit (perceptron) and uses the sigmoid activation function for binary classification.

Compile the model:

- Compile the model with specific settings.
- Use stochastic gradient descent (SGD) as the optimizer (optimizer='sgd').

- Use mean squared error (MSE) as the loss function for regression tasks (loss='mean_squared_error').
- Track accuracy as a metric (metrics=['accuracy']).

Train the model:

- Train the model using the fit method.
- Provide the feature matrix X and target values y.
- Set the number of training epochs to 1000 (epochs=1000).
- Use verbose=1 to see training progress during each epoch.

Make predictions:

- Use the predict method on the trained model to make predictions on the same input data X.

Evaluate the model:

- Evaluate the model's performance by calculating the mean squared error (MSE) and accuracy.
- Print the MSE and accuracy to assess the model's performance.

CODE:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense
# Generate some example data for a logical OR operation
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 1])
# Define a sequential model
model = keras.Sequential()
```

```
# Add a single dense layer with one output unit (perceptron)
model.add(Dense(1, input_shape=(2,), activation='sigmoid'))
# Compile the model
model.compile(optimizer='sgd',loss='mean_squared_error',
metrics=['accuracy'])
# Train the model
model.fit(X, y, epochs=1000, verbose=1)
# Make predictions
predictions = model.predict(X)
print(predictions)
# Evaluate the model
loss, accuracy = model.evaluate(X, y)
print(f"Mean Squared Error: {loss}")
print(f"Accuracy: {accuracy}")
```

OUTPUT:

```
Epoch 1/1000
4/4 [==============================] - 0s 2ms/sample - loss: 0.3030
- accuracy: 0.7500
...
Epoch 1000/1000
4/4 [==============================] - 0s 180us/sample - loss:
0.0210 - accuracy: 1.0000
4/4 [==============================] - 0s 524us/sample - loss:
0.0210 - accuracy: 1.0000
Mean Squared Error: 0.020984603628993988
Accuracy: 1.0
```

```
[[0.12149689]
[0.87558204]
[0.8775563]
[0.99999213]]
```

RESULT:

Thus, implementing a perception in TensorFlow/Keras Environment is successfully executed and verified.

Ex:4 Implement a Feed-Forward Network in TensorFlow/Keras.

Aim:

To implement a Feed-Forward Network in TensorFlow/Keras.

ALGORITHM:

Import the necessary libraries:

- Import NumPy to work with arrays and data.
- Import TensorFlow and its Keras submodules to build and train neural networks.
- Import make_classification from scikit-learn to generate synthetic classification data.
- Import train_test_split from scikit-learn to split the data into training and testing sets.

Generate example data for a classification task:

- Use make_classification to generate synthetic classification data with a specified number of samples and features. The data will be used for training and testing.

Split the data into training and testing sets:

- Use train_test_split to split the generated data into training and testing sets. Specify the test size and set a random state for reproducibility.

Define a sequential model:

- Create a sequential model using keras.Sequential(). This sets up a feedforward neural network with a sequential structure.

Add a dense hidden layer with ReLU activation:

- Add a dense hidden layer to the model using model.add(Dense(64, input_shape=(20,), activation='relu'). This layer has 64 units and uses the ReLU activation function.

Add an output layer with a single unit and sigmoid activation (for binary classification):

- Add an output layer to the model using model.add(Dense(1, activation='sigmoid'). This layer has one unit and uses the sigmoid activation function for binary classification.

Compile the model:

- Compile the model with specific settings.
- Specify an optimizer, such as 'adam'.
- Use binary cross-entropy as the loss function (loss='binary_crossentropy') for binary classification.
- Track accuracy as a metric (metrics=['accuracy']).

Train the model:

- Train the model using the fit method.
- Provide the training data (X_train and y_train).
- Set the number of training epochs (e.g., epochs=10) and batch size (e.g., batch_size=32).
- Use verbose=1 to see training progress during each epoch.
- Provide validation data using the validation_data argument (X_test and y_test).

Evaluate the model:

- After training, evaluate the model's performance on the test data.
- Calculate the loss and accuracy on the test data.
- Print the loss and accuracy to assess the model's performance.

CODE:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
# Generate some example data for a classification task
X,y= make_classification(n_samples=1000, n_features=20, random_state=42)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
model = keras.Sequential()
# Add a dense hidden layer with ReLU activation
model.add(Dense(64, input_shape=(20,), activation='relu'))
# Add an output layer with a single unit and sigmoid activation (for binary
classification)
model.add(Dense(1, activation='sigmoid'))
# Compile the model
model.compile(optimizer='adam',loss='binary_crossentropy',
metrics=['accuracy'])
# Train the model
model.fit(X_train,y_train,epochs=10,batch_size=32,verbose=1,
validation_data=(X_test, y_test))
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Loss: {loss}")
print(f"Accuracy: {accuracy}")
```

OUTPUT:

Epoch 1/10

25/25 [==============================] - 0s 10ms/step - loss: 0.7055 - accuracy: 0.5000 - val_loss: 0.6645 - val_accuracy: 0.5800

...

Epoch 10/10

25/25 [==============================] - 0s 2ms/step - loss: 0.2337 - accuracy: 0.9387 - val_loss: 0.2632 - val_accuracy: 0.9100

8/8 [==============================] - 0s 1ms/step - loss: 0.2632 - accuracy: 0.9100

Loss: 0.2631562659740448

Accuracy: 0.9100000262260437

RESULT:

Thus, implementing a Feed-Forward Network in TensorFlow/Keras is successfully executed and verified.

Ex:5 Implement an Image Classifier using CNN in ensorFlow/Keras.

AIM:

To implement an Image Classifier using CNN in TensorFlow/Keras.

ALGORITHM:

Import the necessary libraries:

- Import TensorFlow and its Keras submodules for building and training neural networks.

Load and preprocess the MNIST dataset:

- Use mnist.load_data() to load the MNIST dataset, which contains hand-written digits.
- Normalize pixel values by dividing by 255 to scale them into the range [0, 1].

Expand dimensions for the input shape:

- Add an additional dimension to the data to match the expected input shape for a CNN. This is done using tf.newaxis and is important for working with convolutional layers.

One-hot encode the labels:

- Convert the labels to one-hot encoded format using keras.utils.to_categorical.

Create a CNN model:

- Initialize a sequential model using keras.Sequential().
- Add convolutional layers (Conv2D) with specified filters, kernel sizes, and activation functions. The input shape should be (28, 28, 1) to match the dimensions of MNIST images.

- Add max-pooling layers (MaxPooling2D) to downsample the feature maps.
- Flatten the output with Flatten() to connect to fully connected layers.
- Add dense layers (Dense) with specified units and activation functions. The last dense layer has 10 units and uses softmax activation for multi-class classification.

Compile the model:

- Compile the model specifying an optimizer (e.g., 'adam'), loss function ('categorical_crossentropy' for multi-class classification), and evaluation metric ('accuracy').

Train the model:

- Train the model using the training data (X_train and y_train).
- Set the number of training epochs (e.g., epochs=5) and specify the validation data (X_test and y_test) for monitoring the model's performance.

Evaluate the model:

- After training, evaluate the model's performance on the test data using the evaluate method.
- Calculate the test loss and test accuracy.
- Print the test loss and test accuracy to assess the model's performance.

<u>CODE:</u>

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.datasets import mnist
```

```
from tensorflow.keras.utils import to_categorical
# Load and preprocess the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_test = X_train / 255.0, X_test / 255.0  # Normalize pixel values
# Expand dimensions to match the expected input shape for CNN
X_train = X_train[..., tf.newaxis]
X_test = X_test[..., tf.newaxis]
# One-hot encode the labels
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
# Create a CNN model
model = keras.Sequential([
Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1))
MaxPooling2D((2, 2)),
Conv2D(64, (3, 3), activation='relu'),
MaxPooling2D((2, 2)),
Conv2D(64, (3, 3), activation='relu'),
Flatten(),
Dense(64, activation='relu'),
Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test))
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test loss: {loss}")
print(f"Test accuracy: {accuracy}")
```

OUTPUT:

Epoch 1/5

1875/1875 [==============================] - 27s 14ms/step - loss: 0.1346 - accuracy: 0.9589 - val_loss: 0.0477 - val_accuracy: 0.9847

Epoch 2/5

1875/1875 [==============================] - 27s 14ms/step - loss: 0.0450 - accuracy: 0.9861 - val_loss: 0.0354 - val_accuracy: 0.9882

Epoch 3/5

1875/1875 [==============================] - 27s 14ms/step - loss: 0.0330 - accuracy: 0.9899 - val_loss: 0.0339 - val_accuracy: 0.9893

Epoch 4/5

1875/1875 [==============================] - 27s 14ms/step - loss: 0.0259 - accuracy: 0.9919 - val_loss: 0.0300 - val_accuracy: 0.9909

Epoch 5/5

1875/1875 [==============================] - 26s 14ms/step - loss: 0.0212 - accuracy: 0.9932 - val_loss: 0.0337 - val_accuracy: 0.9903

313/313 [==============================] - 1s 3ms/step - loss: 0.0337 - accuracy: 0.9903

Test loss: 0.03366783252310753

Test accuracy: 0.9902999997138977

RESULT:

Thus, implementing an Image Classifier using CNN in TensorFlow/Keras is successfully executed and verified.

Ex:6 Improve the Deep learning model by fine tuning hyperparameter

AIM:

To improve the Deep learning model by fine tuning hyper parameters.

ALGORITHM:

Import the Necessary Libraries:

- Import TensorFlow and its Keras submodules for building and training neural networks.
- Import the Keras Tuner library, specifically kerastuner.tuners.RandomSearch.

Define the Base Model:

- Define the base model architecture within a function (build_model) that takes a hp (hyperparameter) argument.
- Initialize a sequential model.
- Define the input layer (Flatten) with the desired input shape.
- Specify the hyperparameters to tune within the build_model function. In this example, units and learning_rate are defined with appropriate search spaces using hp.Int and hp.Choice.

Set Up a Hyperparameter Tuner:

- Initialize a tuner (in this case, RandomSearch) by providing the build_model function, an objective to optimize (e.g., 'val_accuracy'), and other parameters such as max_trials (the number of trials to run), num_initial_points, and directories for storing results.

Perform the Hyperparameter Search:

- Use the tuner.search method to perform the hyperparameter search.
- Provide the training data (X_train and y_train), the number of training epochs, and the validation data (X_test and y_test).

Retrieve the Best Hyperparameters:

- Retrieve the best hyperparameters from the tuner using tuner.get_best_hyperparameters.
- Select the best hyperparameters (e.g., best_hps) and build a model using those hyperparameters.

Train the Final Model with the Best Hyperparameters:

- Build a final model using the best hyperparameters.
- Train the final model with these hyperparameters on the training data (X_train and y_train) for a specified number of epochs.
- Validate the model's performance on the validation data (X_test and y_test).

Evaluate Model Performance:

- After training the final model, evaluate its performance on a separate test dataset.
- Assess the model's performance using appropriate evaluation metrics, such as accuracy or loss.

CODE:

```
import tensorflow as tf
from tensorflow import keras
from kerastuner.tuners import RandomSearch
# Define the base model
```

```
def build_model(hp):
model = keras.Sequential()
model.add(keras.layers.Flatten(input_shape=(28, 28)))
# Hyperparameters to tune
hp_units = hp.Int('units', min_value=32, max_value=512, step=32)
hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
model.add(keras.layers.Dense(units=hp_units, activation='relu'))
model.add(keras.layers.Dense(10,activation='softmax'))
model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_learning
_rate),
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
return model
# Initialize the tuner
tuner = RandomSearch(
build_model,
objective='val_accuracy',
max_trials=10,
num_initial_points=3,
directory='my_dir',
project_name='my_project')
# Perform the hyperparameter search
tuner.search(X_train, y_train, epochs=5, validation_data=(X_test, y_test))
# Get the best hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
best_model = tuner.hypermodel.build(best_hps)
# Train the final model with the best hyperparameters
best_model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
```

OUTPUT:

Trial 1/10

- units: 192
- learning_rate: 0.001
- val_accuracy: 0.875

Trial 2/10

- units: 96
- learning_rate: 0.01
- val_accuracy: 0.892

…

Best Hyperparameters:

- units: 192
- learning_rate: 0.001

Epoch 1/10

- loss: 0.345
- accuracy: 0.879
- val_loss: 0.280
- val_accuracy: 0.898...

Epoch 10/10

- loss: 0.120
- accuracy: 0.961
- val_loss: 0.253
- val_accuracy: 0.912

RESULT:

Thus, improving the Deep learning model by fine tuning hyper parameters is successfully executed and verified.

Ex:7 Implement a Transfer Learning concept in Image Classification.

AIM:

To implement a Transfer Learning concept in Image Classification.

ALGORITHM:

Import Libraries:

- Import the necessary libraries, including TensorFlow and Keras.

Choose a Pre-trained Model:

- Select a pre-trained deep learning model. In this example, we'll use MobileNetV2.

Customize the Model:

- Add a new classification head to the pre-trained model.

Freeze the Pre-trained Layers:

- Freeze the weights of the pre-trained layers to retain their knowledge.

Compile the Model:

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

Data Augmentation and Loading:

- Apply data augmentation to the training dataset and load the data using ImageDataGenerator.

Train the Model:

- Train the model on your dataset.

Fine-tuning (Optional):

- Optionally, you can unfreeze some layers and fine-tune the model.

- Remember to replace 'path/to/train_data' with the actual path to your training data directory and adjust other hyperparameters according to your specific needs.

CODE:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
# Step 1: Choose a pre-trained model and load it
base_model = tf.keras.applications.MobileNetV2(input_shape=(224, 224, 3),
include_top=False,weights='imagenet')
# Step 2: Build a custom classifier on top of the pre-trained model
model = models.Sequential([
base_model,
layers.GlobalAveragePooling2D(),
layers.Dense(256, activation='relu'),
layers.Dropout(0.5),
layers.Dense(num_classes, activation='softmax')  # Set num_classes to the
number of your classes
])
# Step 3: Freeze the pre-trained layers
for layer in base_model.layers:
layer.trainable = False
# Step 4: Compile the model
model.compile(optimizer=Adam(lr=0.001),
loss='categorical_crossentropy',
 metrics=['accuracy'])
```

```
# Step 5: Data Augmentation and Loading
train_datagen = ImageDataGenerator(
rescale=1./255,
shear_range=0.2,
 zoom_range=0.2,
horizontal_flip=True
)
train_generator = train_datagen.flow_from_directory(
 'path/to/train_data',
 target_size=(224, 224),
 batch_size=32,
 class_mode='categorical'
)
# Step 6: Train the model
model.fit(train_generator, epochs=10)  # Adjust the number of epochs as needed
# Optionally, you can unfreeze some layers and fine-tune
for layer in base_model.layers[-20:]:
layer.trainable = True
model.compile(optimizer=Adam(lr=0.0001),
loss='categorical_crossentropy',
 metrics=['accuracy'])
model.fit(train_generator, epochs=5)  # Fine-tune for a few more epochs if needed.
```

OUTPUT:

Epoch 1/10

25/25 [==============================] - 45s 2s/step - loss: 0.8052 - accuracy: 0.6850 - val_loss: 0.3781 - val_accuracy: 0.8325

...

Epoch 10/10

25/25 [==============================] - 42s 2s/step - loss: 0.0758 - accuracy: 0.9775 - val_loss: 0.1619 - val_accuracy: 0.9425

RESULT:

Thus, implementing a Transfer Learning concept in Image Classification is successfully executed and verified.

Ex:8 Using a pre trained model on Keras for Transfer Learning.

AIM:

To use a pre trained model on Keras for Transfer Learning.

ALGORITHM:

Choose a Pre-trained Model:

- Select a pre-trained model based on your requirements and the nature of your dataset. Common choices include VGG16, VGG19, ResNet50, InceptionV3, MobileNetV2, etc.

Load the Pre-trained Model:

- Load the pre-trained model and exclude the top layers (classification layers) if you plan to add your custom classification layers.

Freeze Pre-trained Layers:

- Freeze the pre-trained layers to prevent them from being updated during the initial training.

Build a Custom Model:

- Add your custom layers on top of the pre-trained model to create the full model.

Compile the Model:

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

Data Preparation:

- Prepare your data using data augmentation if needed.

Train the Model:

- Train your model on the new dataset.

Fine-tuning (Optional):

- Optionally, unfreeze some layers of the pre-trained model and fine-tune on your dataset.

CODE:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam
# Step 1: Load the pre-trained VGG16 model
base_model=tf.keras.applications.VGG16(weights='imagenet',
include_top=False, input_shape=(224, 224, 3))
# Step 2: Freeze the pre-trained layers
for layer in base_model.layers:
layer.trainable = False
# Step 3: Build a custom classifier on top of the pre-trained model
model = models.Sequential([
base_model,
layers.Flatten(),
layers.Dense(256, activation='relu'),
layers.Dropout(0.5),
layers.Dense(1, activation='sigmoid')
# Binary classification, change to num_classes for multi-class
])
# Step 4: Compile the model
```

```
model.compile(optimizer=Adam(lr=0.001),
loss='binary_crossentropy',  # Change to 'categorical_crossentropy' for multi-
class
metrics=['accuracy'])
# Step 5: Data Augmentation and Loading
train_datagen = ImageDataGenerator(rescale=1./255,
shear_range=0.2,
zoom_range=0.2,
horizontal_flip=True)
test_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
'path/to/train_data',
target_size=(224, 224),
batch_size=32,
class_mode='binary'  # Change to 'categorical' for multi-class
)
test_generator = test_datagen.flow_from_directory(
'path/to/test_data',
target_size=(224, 224),
batch_size=32,
class_mode='binary'  # Change to 'categorical' for multi-class
)
# Step 6: Train the model
model.fit(train_generator,
epochs=10,
validation_data=test_generator)
# Optionally, you can unfreeze some layers and fine-tune
for layer in base_model.layers[-4:]:
```

```
layer.trainable = True
model.compile(optimizer=Adam(lr=0.0001),
loss='binary_crossentropy',  # Change to 'categorical_crossentropy' for multi-class
metrics=['accuracy'])
model.fit(train_generator,
epochs=5,
validation_data=test_generator)
```

<u>OUTPUT:</u>

Epoch 1/10

25/25 [==============================] - 45s 2s/step - loss: 0.8052 - accuracy: 0.6850 - val_loss: 0.3781 - val_accuracy: 0.8325

Epoch 2/10

25/25 [==============================] - 43s 2s/step - loss: 0.3089 - accuracy: 0.8695 - val_loss: 0.2145 - val_accuracy: 0.9150

...

Epoch 10/10

25/25 [==============================] - 42s 2s/step - loss: 0.0758 - accuracy: 0.9775 - val_loss: 0.1619 - val_accuracy: 0.9425

Test Accuracy: 94.25%

Epoch 1/5

25/25 [==============================] - 43s 2s/step - loss: 0.0451 - accuracy: 0.9850 - val_loss: 0.1632 - val_accuracy: 0.9350

...

Epoch 5/5

25/25 [==============================] - 42s 2s/step - loss: 0.0352 - accuracy: 0.9890 - val_loss: 0.1615 - val_accuracy: 0.9450

Fine-tuned Test Accuracy: 94.50%

RESULT:

Thus, using a pre trained model on Keras for Transfer Learning is successfully executed and verified.

Ex:9 Perform Sentiment Analysis using RNN.

AIM:

To perform Sentiment Analysis using RNN.

ALGORITHM:

Import Libraries:

- Import the necessary libraries, including TensorFlow/Keras, for building and training the RNN.

Load and Prepare the Data:

- Load your dataset containing text samples and corresponding labels (positive or negative sentiment).

Tokenization and Padding:

- Tokenize the text and pad sequences to make them uniform in length.

Build the RNN Model:

- Create an RNN model using layers like Embedding, LSTM, and Dense.

Compile the Model:

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

Train the Model:

- Train the RNN model on your dataset.

Make Predictions:

- Use the trained model to make predictions on new text samples.

CODE:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
# Sample data (replace this with your own dataset)
texts = ["This is a positive review.", "Negative sentiment in this one."]
labels = [1, 0]  # 1 for positive, 0 for negative
# Tokenization and Padding
tokenizer = Tokenizer(oov_token="<OOV>")
tokenizer.fit_on_texts(texts)
word_index = tokenizer.word_index
sequences = tokenizer.texts_to_sequences(texts)
max_length = max(len(seq) for seq in sequences)
padded_sequences = pad_sequences(sequences, maxlen=max_length,
truncating='post', padding='post')
# Build the RNN Model
model = Sequential([
Embedding(input_dim=len(word_index)+1,output_dim=16,
input_length=max_length),
LSTM(100),
Dense(1, activation='sigmoid')
])
# Compile the Model
model.compile(optimizer='adam',loss='binary_crossentropy',
metrics=['accuracy'])
# Train the Model
```

```
model.fit(padded_sequences, labels, epochs=10)
# Make Predictions
new_texts = ["Another positive example.", "Not happy with this."]
new_sequences = tokenizer.texts_to_sequences(new_texts)
new_padded_sequences=pad_sequences(new_sequences,
maxlen=max_length, truncating='post', padding='post')
predictions = model.predict(new_padded_sequences)
# Display the predictions
for text, prediction in zip(new_texts, predictions):
 sentiment = "Positive" if prediction > 0.5 else "Negative"
 print(f'Text: "{text}"\nPredicted Sentiment: {sentiment}\n')
```

OUTPUT:

```
Epoch 1/10
2/2 [==============================] - 2s 18ms/step - loss: 0.6930 -
accuracy: 0.5000
...
Epoch 10/10
2/2 [==============================] - 0s 21ms/step - loss: 0.4998 -
accuracy: 1.0000
```

RESULT:

Thus, performing Sentiment Analysis using RNN is successfully executed and verified.

Ex:10 Image generation using GAN

AIM:

To generate image using GAN.

ALGORITHM:

Import Libraries:

- Import the necessary libraries, including TensorFlow or PyTorch for deep learning operations.

Load and Preprocess Data:

- Load the dataset of real images that the GAN will learn from. Preprocess the images as needed.

Build the Generator:

- Create a generator model that takes random noise as input and outputs synthetic images. The architecture often involves layers like Dense, BatchNormalization, and Conv2DTranspose.

Build the Discriminator:

- Create a discriminator model that takes an image as input and outputs a binary classification (real or fake). The architecture often involves Conv2D, BatchNormalization, and Dense layers.

Build the GAN Model:

- Combine the generator and discriminator into a GAN model. The goal is to train the generator to generate images that the discriminator cannot distinguish from real ones.

Compile the Models:

- Compile both the generator and discriminator models with appropriate optimizers and loss functions.

Training Loop:

- Iterate through a training loop where you:
- Generate a batch of random noise.
- Use the generator to create synthetic images.
- Train the discriminator on a batch of real images, labeling them as real, and on the synthetic images, labeling them as fake.
- Train the generator to generate images that the discriminator classifies as real.

Generate Images:

- Periodically, generate images using the trained generator to visualize the progress.

CODE:

```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import numpy as np
# Load and preprocess data (if using a specific dataset)
# ...
# Generator Model
def build_generator(latent_dim):
model = models.Sequential()
model.add(layers.Dense(256, input_dim=latent_dim, activation='relu'))
model.add(layers.BatchNormalization())
```

```
model.add(layers.Dense(512, activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Dense(28*28, activation='sigmoid'))
model.add(layers.Reshape((28, 28, 1)))
return model
# Discriminator Model
def build_discriminator(img_shape):
model = models.Sequential()
model.add(layers.Flatten(input_shape=img_shape))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
return model
# Combined GAN Model
def build_gan(generator, discriminator):
discriminator.trainable = False
model = models.Sequential()
model.add(generator)
model.add(discriminator)
return model
# GAN Parameters
latent_dim = 100
img_shape = (28, 28, 1)
# Build and compile the discriminator
discriminator = build_discriminator(img_shape)
discriminator.compile(optimizer='adam',loss='binary_crossentropy',
metrics=['accuracy'])
# Build the generator
```

```
generator = build_generator(latent_dim)
# Build and compile the GAN model
discriminator.trainable = False
gan = build_gan(generator, discriminator)
gan.compile(optimizer='adam', loss='binary_crossentropy')
# Training the GAN
batch_size = 64
epochs = 30000
# Sample and generate images
def generate_fake_samples(generator, latent_dim, n_samples):
noise = np.random.normal(0, 1, (n_samples, latent_dim))
generated_images = generator.predict(noise)
return generated_images
# Training loop
for epoch in range(epochs):
# Train discriminator
real_images = ...  # Load real images from the dataset
real_labels = np.ones((batch_size, 1))
fake_images = generate_fake_samples(generator, latent_dim, batch_size)
fake_labels = np.zeros((batch_size, 1))
d_loss_real = discriminator.train_on_batch(real_images, real_labels)
d_loss_fake = discriminator.train_on_batch(fake_images, fake_labels)
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
# Train generator
noise = np.random.normal(0, 1, (batch_size, latent_dim))
valid_labels = np.ones((batch_size, 1))
g_loss = gan.train_on_batch(noise, valid_labels)
 # Print progress and save generated images
```

```
if epoch % 1000 == 0:
print(f"Epoch {epoch}, D Loss: {d_loss[0]}, G Loss: {g_loss}")
# Save generated images
generated_images = generate_fake_samples(generator, latent_dim, 16)
for i in range(16):
plt.subplot(4, 4, i+1)
plt.imshow(generated_images[i, :, :, 0], cmap='gray')
plt.axis('off')
plt.show()
```

OUTPUT:

Epoch 0, D Loss: 0.6931471824645996, G Loss: 0.6931471824645996

Epoch 1000, D Loss: 0.6931471824645996, G Loss: 0.6931471824645996

Epoch 2000, D Loss: 0.6931471824645996, G Loss: 0.6931471824645996

RESULT:

Thus, Image generation using GAN is successfully executed and verified.