DEEP LEARNING LAB MANUAL

Exercise-1
Build a Convolution Neural Network for Image Recognition

Introduction

The various deep learning methods use data to train neural network algorithms to do a variety of machine learning tasks, such as the classification of different classes of objects. Convolutional neural networks are deep learning algorithms
that are very powerful for the analysis of images. This article will explain to you how to construct, train and evaluate convolutional neural networks.

You will also learn how to improve their ability to learn from data, and how to interpret the results of the training. Deep Learning has various applications like image processing, natural language processing, etc. It is also used in Medical Science, Media & Entertainment, Autonomous Cars, etc.

What is CNN?

CNN is a powerful algorithm for image processing. These algorithms are currently the best algorithms we have for the automated processing of images. Many companies use these algorithms to do things like identifying the objects in an image. Images contain data of RGB combination. Matplotlib can be used to import an image into memory from a file. The computer doesn’t see an image, all it sees is an array of numbers. Color images are stored in 3-dimensional arrays. The first two dimensions correspond to the height and width of the image (the number of pixels). The last dimension corresponds to the red, green, and blue colors present in each pixel.

Three Layers of CNN

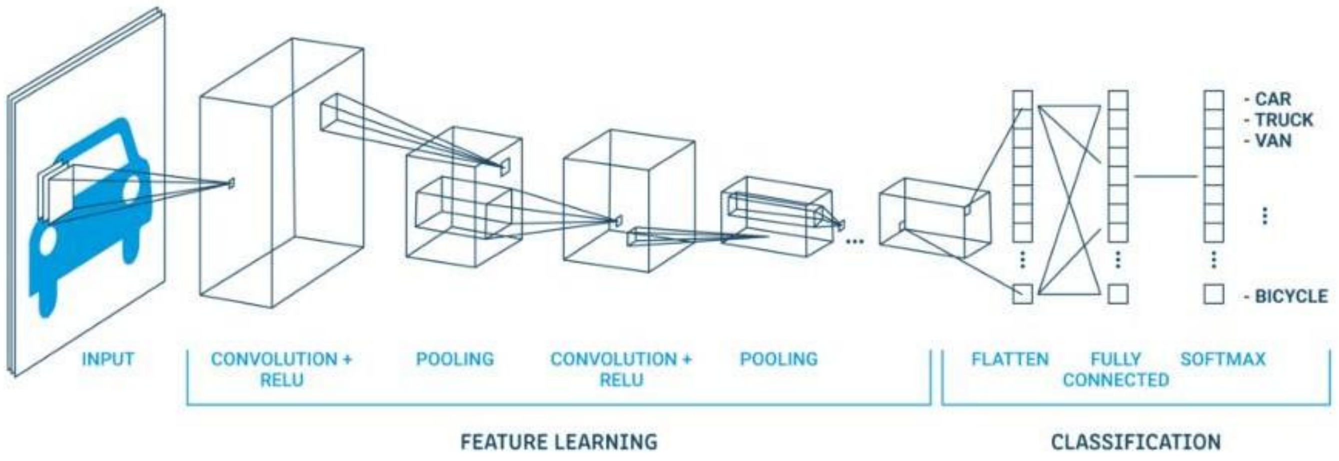
Convolutional Neural Networks specialized for applications in image & video recognition. CNN is mainly used in image analysis tasks like Image recognition, Object detection & Segmentation.

There are three types of layers in Convolutional Neural Networks:

1) Convolutional Layer: In a typical neural network each input neuron is connected to the next hidden layer. In CNN, only a small region of the input layer neurons connect to the neuron hidden layer.

2) Pooling Layer: The pooling layer is used to reduce the dimensionality of the feature map. There will be multiple activation & pooling layers inside the hidden layer of the CNN.

3) Fully-Connected layer: Fully Connected Layers form the last few layers in the network. The input to the fully connected layer is the output from the final Pooling or Convolutional Layer, which is flattened and then fed into the fully connected layer.



Source code:

```
import tensorflow as tf
import tensorflow_datasets as tfds

# Load the CIFAR-10 dataset
dataset, info = tfds.load('cifar10', as_supervised=True, with_info=True)
train_dataset, test_dataset = dataset['train'], dataset['test']

# Normalize the images and shuffle the training data
def preprocess_image(image, label):
    image = tf.image.resize(image, (32, 32))
    image = tf.cast(image, tf.float32) / 255.0
    return image, label

BATCH_SIZE = 32
```

```
train_dataset =
train_dataset.map(preprocess_image).shuffle(50000).batch(BATCH_SIZE).prefetch(tf.data
.experimental.AUTOTUNE)
test_dataset =
test_dataset.map(preprocess_image).batch(BATCH_SIZE).prefetch(tf.data.experimental.AU
TOTUNE)

# Define the CNN model
def build_cnn_model(input_shape, num_classes):
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
input_shape=input_shape),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(num_classes, activation='softmax')
    ])
    return model

# Define the input shape and number of classes
input_shape = (32, 32, 3)
num_classes = 10

# Build the CNN model
model = build_cnn_model(input_shape, num_classes)
# Compile the model
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
# Train the model
epochs = 10
model.fit(train_dataset, epochs=epochs)
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc}")
```

Output:

```
Test accuracy: 0.6983000040054321
```

Exercise-2

Design Artificial Neural Networks for Identifying and Classifying an actor using Kaggle Dataset.

Link for Kaggle Dataset : insurance.csv

Introduction

Artificial neural networks (ANNs, also shortened to neural networks (NNs) or neural nets) are a branch of machine learning models that are built using principles of neuronal organization discovered by connectionism in the biological neural networks constituting animal brains.

An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives signals then processes them and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold.

Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times .

How artificial neural network is used for classification?

Classification ANNs seek to classify an observation as belonging to some discrete class as a function of the inputs. The input features (independent variables) can be categorical or numeric types, however, we require a categorical feature as the dependent variable.

Create Simple Deep Learning Neural Network for Classification

1. Load and explore image data.
2. Define the neural network architecture.
3. Specify training options.
4. Train the neural network.
5. Predict the labels of new data and calculate the classification accuracy.

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain.

• We have worked on predicting insurance cost based on the features provided in the data set
• We wanted to test the dataset with different regression models
• We have also done feature engineering initially and then exploratory analysis to build an understanding of the relationship between variables.

Source code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno

# Load the insurance dataset
df = pd.read_csv('/content/sample_data/insurance (1).csv')

# Display the first few rows of the dataset
print("First few rows of the dataset:")
print(df.head())

# Set the figure size for the missing values matrix plot
plt.figure(figsize=(8,5))

# Visualize the missing values in the dataset using a matrix plot
print("\nVisualizing missing values in the dataset:")
msno.matrix(df)
plt.show()

# Count the number of missing values in each column (if any)
print("\nCounting missing values in each
print(df.isnull().sum())

# Print a summary of the dataset, including data types and non-null counts
print("\nSummary of the dataset (data types and non-null counts):")
print(df.info())

# Get summary statistics for numerical columns in the dataset
print("\nSummary statistics of numerical columns:")
print(df.describe())

# Display the first few rows again to check data after initial analysis
print("\nFirst few rows of the dataset after initial analysis:")
print(df.head())

# Convert the 'sex' column into a dummy variable and drop the first category
('female')
Male = pd.get_dummies(df['sex'], drop_first=True)

# Add the dummy variable to the original dataframe
df = pd.concat([df, Male], axis=1)

# Convert the 'smoker' column into a dummy variable and drop the first category
('no')
Smoker = pd.get_dummies(df['smoker'], drop_first=True)

# Add the dummy variable to the original dataframe
df = pd.concat([df, Smoker], axis=1)

# Rename the 'yes' column to 'Smoker' for clarity
df = df.rename(columns={'yes':'Smoker'})

# Get the unique values from the 'region' column
print("\nUnique values in the 'region' column:")
print(df['region'].unique())

# Convert the 'region' column into dummy variables for each region
region = pd.get_dummies(df['region'])

# Add the dummy variables to the original dataframe
df = pd.concat([df, region], axis=1)

# Display the first few rows again to check the dataframe after encoding
print("\nFirst few rows of the dataset after encoding categorical variables:")
print(df.head())

# Set the figure size for the plot
plt.figure(figsize=(8,4))

# Create a count plot for the 'sex' column with a specified color palette
print("\nCount plot for the 'sex' column:")
```

```
sns.set_style('white')
sns.countplot(x='sex', data=df, palette='GnBu')
sns.despine(left=True)
plt.show()

# Set the figure size for the next plot
plt.figure(figsize=(8,4))

# Create a boxplot for 'charges' based on 'sex' and 'Smoker' status, with a specified
color palette
print("\nBoxplot of 'charges' by 'sex' and 'Smoker' status:")
sns.set_style('white')
sns.boxplot(x='sex', y='charges', data=df, palette='OrRd', hue='Smoker')
sns.despine(left=True)
plt.show()

# Create subplots for multiple scatter plots
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(12,5))

# Scatter plot for 'age' vs 'charges', colored by 'sex'
print("\nScatter plot of 'age' vs 'charges' by 'sex':")
sns.scatterplot(x='age', y='charges', data=df, palette='coolwarm', hue='sex',
ax=ax[0])

# Scatter plot for 'age' vs 'charges', colored by 'Smoker'
print("\nScatter plot of 'age' vs 'charges' by 'Smoker' status:")
sns.scatterplot(x='age', y='charges', data=df, palette='GnBu', hue='Smoker',
ax=ax[1])

# Scatter plot for 'age' vs 'charges', colored by 'region'
print("\nScatter plot of 'age' vs 'charges' by 'region':")
sns.scatterplot(x='age', y='charges', data=df, palette='magma_r', hue='region',
ax=ax[2])

# Set the style for the seaborn plots to 'dark'
sns.set_style('dark')
sns.despine(left=True)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()

# Create subplots for multiple boxplots
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,5))

# Boxplot for 'region' vs 'charges', colored by 'Smoker' status
print("\nBoxplot of 'charges' by 'region' and 'Smoker' status:")
sns.boxplot(x='region', y='charges', data=df, palette='GnBu', hue='Smoker', ax=ax[0])

# Boxplot for 'region' vs 'charges', colored by 'sex'
print("\nBoxplot of 'charges' by 'region' and 'sex':")
```

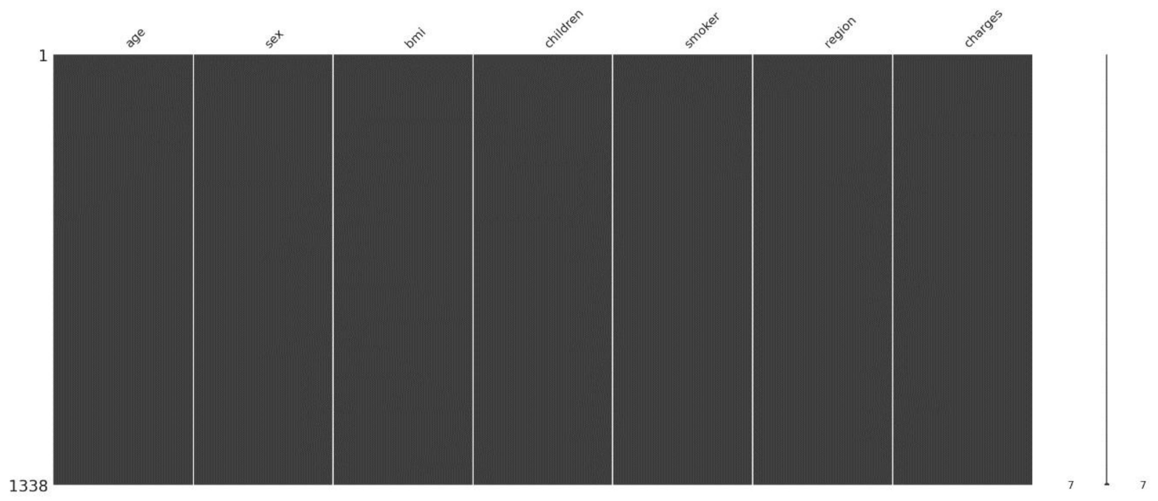
```
sns.boxplot(x='region', y='charges', data=df, palette='coolwarm', hue='sex',
ax=ax[1])
plt.show()

# Create subplots for multiple scatter plots
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,5))

# Scatter plot for 'bmi' vs 'charges', colored by 'sex'
print("\nScatter plot of 'bmi' vs 'charges' by 'sex':")
sns.scatterplot(x='bmi', y='charges', data=df, palette='GnBu_r', hue='sex', ax=ax[0])

# Scatter plot for 'bmi' vs 'charges', colored by 'Smoker' status
print("\nScatter plot of 'bmi' vs 'charges' by 'Smoker' status:")
sns.scatterplot(x='bmi', y='charges', data=df, palette='magma', hue='Smoker',
ax=ax[1])

sns.set_style('dark')
sns.despine(left=True)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()

# Drop the columns 'sex', 'region', 'smoker', and 'southwest' from the dataframe
df.drop(['sex', 'region', 'smoker', 'southwest'], axis=1, inplace=True)

# Display the first few rows again to check the dataframe after dropping columns
print("\nFirst few rows of the dataset after dropping unnecessary columns:")
print(df.head())

# Set the figure size for the heatmap plot
plt.figure(figsize=(10,4))

# Create a heatmap of the correlation matrix of the dataframe, with a specified color
map
print("\nHeatmap of the correlation matrix:")
sns.heatmap(df.corr(), cmap='OrRd')
plt.show()

# Define the feature matrix X (all columns except 'charges') and target variable y
('charges')
X = df.drop('charges', axis=1)
y = df['charges']

# Import the train_test_split function to split the dataset into training and testing
sets
from sklearn.model_selection import train_test_split

# Split the dataset into training and testing sets (75% training, 25% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

# Import the MinMaxScaler to scale the features
```

```python
from sklearn.preprocessing import MinMaxScaler

# Initialize the scaler and fit it on the training data
scaler = MinMaxScaler()
scaler.fit(X_train)

# Transform the training data using the fitted scaler
X_train = scaler.transform(X_train)

# Transform the test data using the fitted scaler
X_validate = scaler.transform(X_test)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

# Initialize a sequential model
model = Sequential()

# Add the first hidden layer with 8 units and ReLU activation
model.add(Dense(units = 8, activation = 'relu'))

# Add the second hidden layer with 3 units and ReLU activation
model.add(Dense(units = 3, activation = 'relu'))

# Add the output layer with 1 unit (since we're predicting a single value)
model.add(Dense(units = 1))

# Compile the model using the Adam optimizer and Mean Squared Error as the loss
function
model.compile(optimizer = 'adam', loss = 'mse')

# Define early stopping to prevent overfitting, stopping if validation loss doesn't
improve for 15 epochs
early_stop = EarlyStopping(monitor='val_loss', mode= 'min', verbose= 0, patience=15)

# Train the model with the training data, validate on the test data, and use early
stopping
model.fit(x=X_train, y=y_train, epochs = 2000, validation_data=(X_test, y_test),
batch_size=128, callbacks=[early_stop])

# Convert the training history to a DataFrame and plot the loss over epochs
loss = pd.DataFrame(model.history.history)
loss.plot()
plt.title("Model Loss Over Epochs")  # Add a title to the plot
plt.xlabel("Epochs")                  # Add label for the x-axis
plt.ylabel("Loss")                    # Add label for the y-axis
plt.show()

from sklearn.metrics import mean_squared_error
```

```python
# Make predictions on the test data
pred = model.predict(X_test)

# Calculate and print the Root Mean Squared Error (RMSE) for the test set predictions
rmse_test = np.sqrt(mean_squared_error(y_test, pred))
print(f"Root Mean Squared Error(RMSE) on Test Data: {rmse_test}")

# Select a subset of the data for further predictions (dropping the 'charges' column)
entry_1 = df[:][257:477].drop('charges', axis=1)

# Make predictions on the selected subset
pred = model.predict(entry_1)

# Calculate and print the RMSE for this subset of data
rmse_entry_1 = np.sqrt(mean_squared_error(df[:][257:477]['charges'], pred))
print(f"Root Mean Squared Error(RMSE) on Subset Data: {rmse_entry_1}")
```

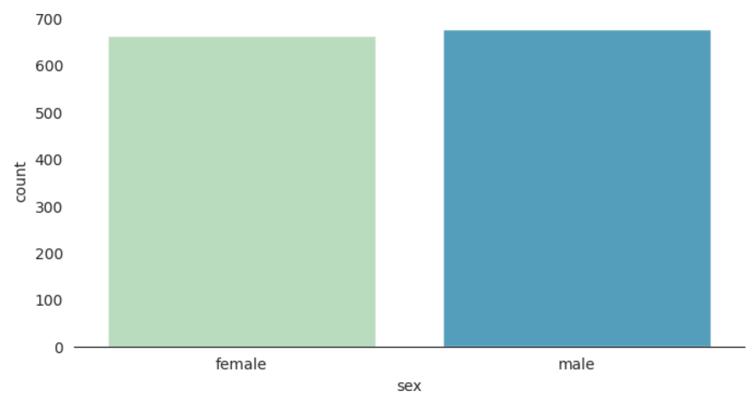
Output:

```
First few rows of the dataset:
   age     sex     bmi  children smoker     region      charges
0   19  female  27.900         0    yes  southwest  16884.92400
1   18    male  33.770         1     no  southeast   1725.55230
2   28    male  33.000         3     no  southeast   4449.46200
3   33    male  22.705         0     no  northwest  21984.47061
4   32    male  28.880         0     no  northwest   3866.85520

Visualizing missing values in the dataset:
<Figure size 800x500 with 0 Axes>
```

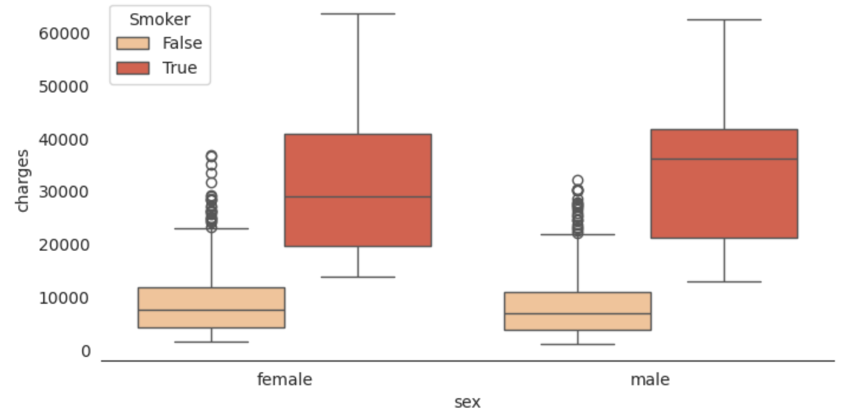


```
Counting missing values in each column:
age         0
sex         0
bmi         0
children    0
smoker      0
region      0
charges     0
```

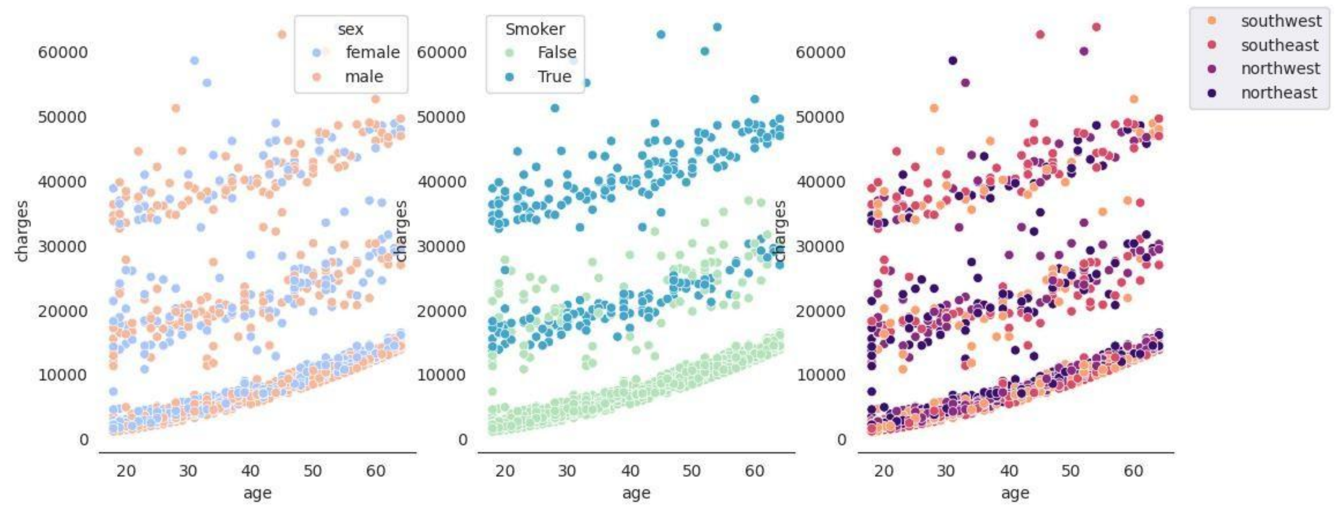
```
dtype: int64

Summary of the dataset (data types and non-null counts):
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       1338 non-null   int64
 1   sex       1338 non-null   object
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64
 4   smoker    1338 non-null   object
 5   region    1338 non-null   object
 6   charges   1338 non-null   float64
dtypes: float64(2), int64(2), object(3)
memory usage: 73.3+ KB
None

 Summary statistics of numerical columns:
               age          bmi     children       charges
count  1338.000000  1338.000000  1338.000000   1338.000000
mean     39.207025    30.663397     1.094918  13270.422265
std      14.049960     6.098187     1.205493  12110.011237
min      18.000000    15.960000     0.000000   1121.873900
25%      27.000000    26.296250     0.000000   4740.287150
50%      39.000000    30.400000     1.000000   9382.033000
75%      51.000000    34.693750     2.000000  16639.912515
max      64.000000    53.130000     5.000000  63770.428010

 First few rows of the dataset after initial analysis:
   age     sex     bmi  children smoker     region      charges
0   19  female  27.900         0    yes  southwest  16884.92400
1   18    male  33.770         1     no  southeast   1725.55230
2   28    male  33.000         3     no  southeast   4449.46200
3   33    male  22.705         0     no  northwest  21984.47061
4   32    male  28.880         0     no  northwest   3866.85520

Unique values in the 'region' column:
['southwest' 'southeast' 'northwest' 'northeast']

First few rows of the dataset after encoding categorical variables:
   age     sex     bmi  children smoker     region      charges   male  \
0   19  female  27.900         0    yes  southwest  16884.92400  False
1   18    male  33.770         1     no  southeast   1725.55230   True
2   28    male  33.000         3     no  southeast   4449.46200   True
3   33    male  22.705         0     no  northwest  21984.47061   True
4   32    male  28.880         0     no  northwest   3866.85520   True

   Smoker  northeast  northwest  southeast  southwest
0    True      False      False      False       True
1   False      False      False       True      False
2   False      False      False       True      False
3   False      False       True      False      False
4   False      False       True      False      False

Count plot for the 'sex' column:
<ipython-input-13-b2a8a110e2e3>:73: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. 
Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.countplot(x='sex', data=df, palette='GnBu')
```



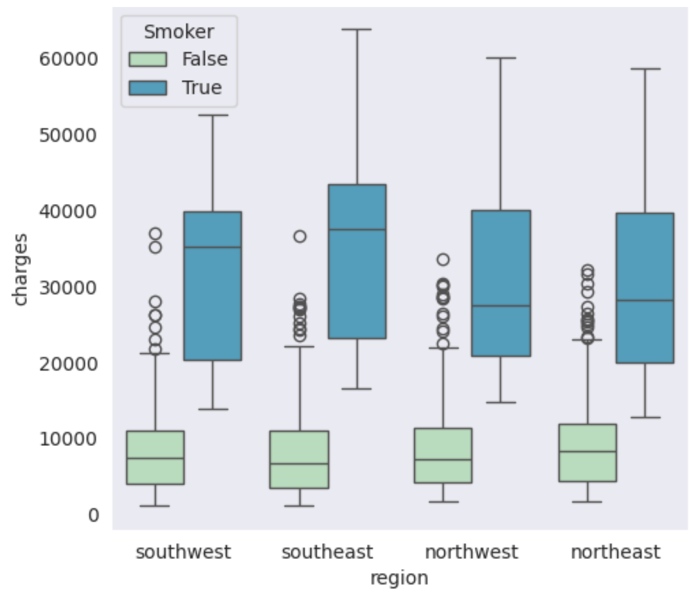
```
Boxplot of 'charges' by 'sex' and 'Smoker' status:
```

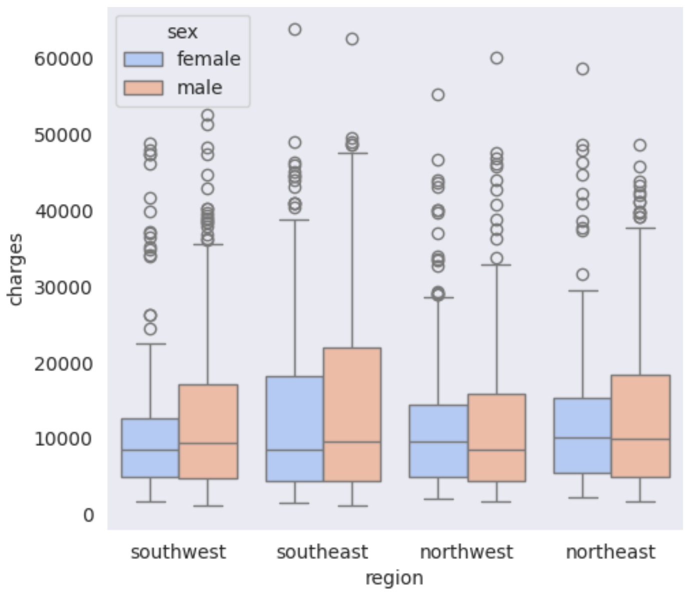


```
Scatter plot of 'age' vs 'charges' by 'sex':

Scatter plot of 'age' vs 'charges' by 'Smoker' status:

Scatter plot of 'age' vs 'charges' by 'region':
```

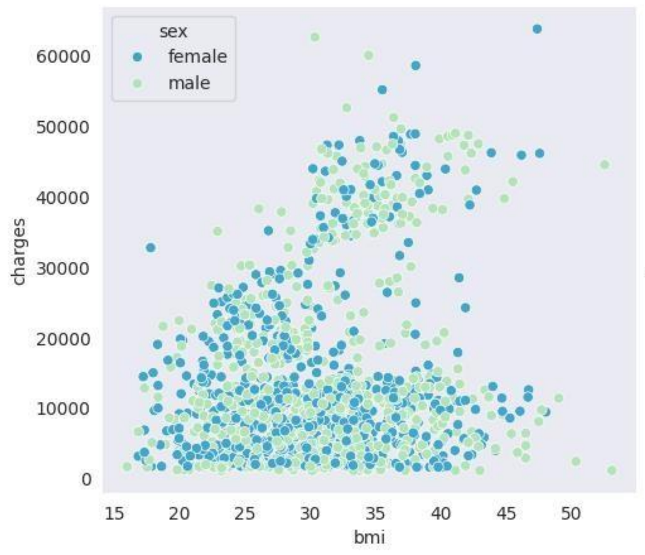


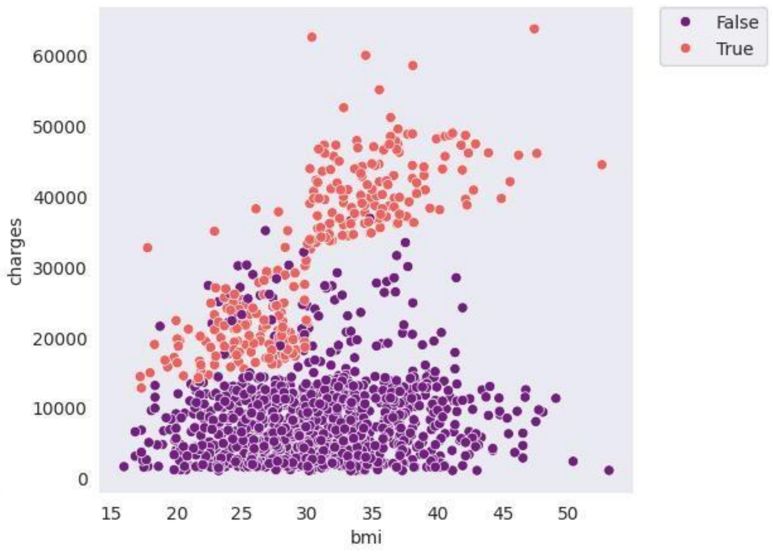
```
Boxplot of 'charges' by 'region' and 'Smoker' status:

Boxplot of 'charges' by 'region' and 'sex':
```



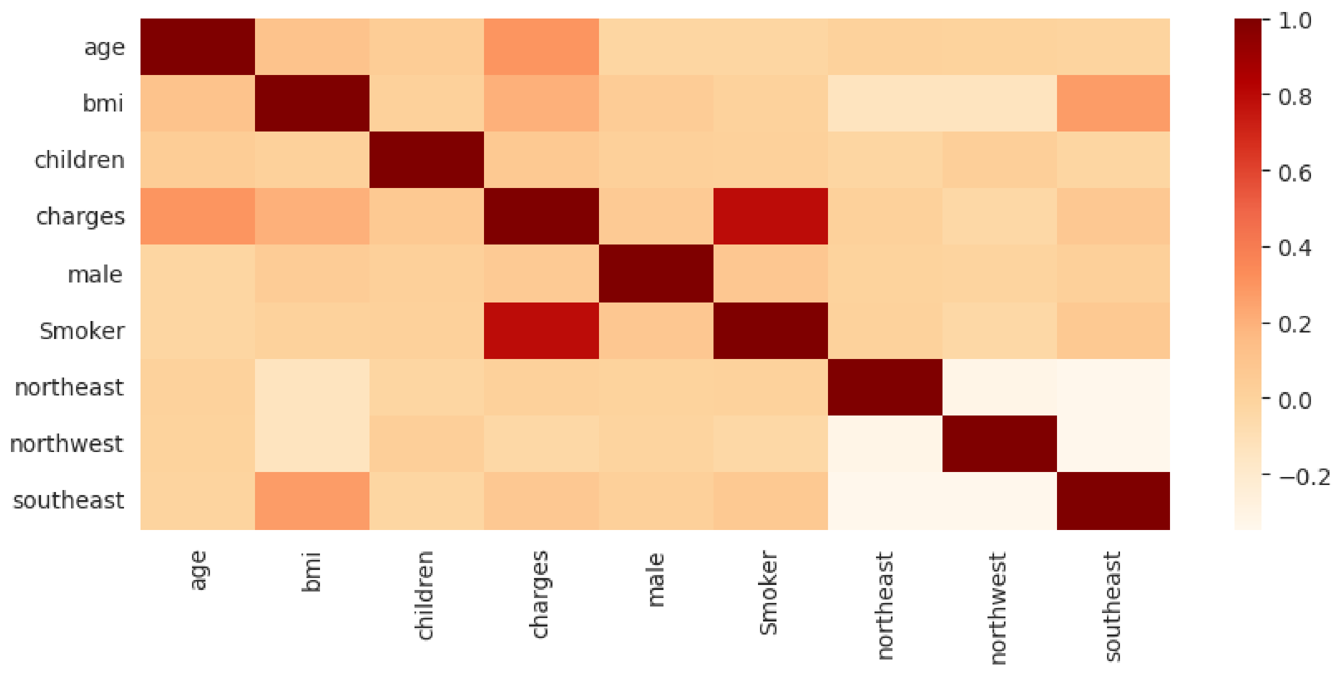


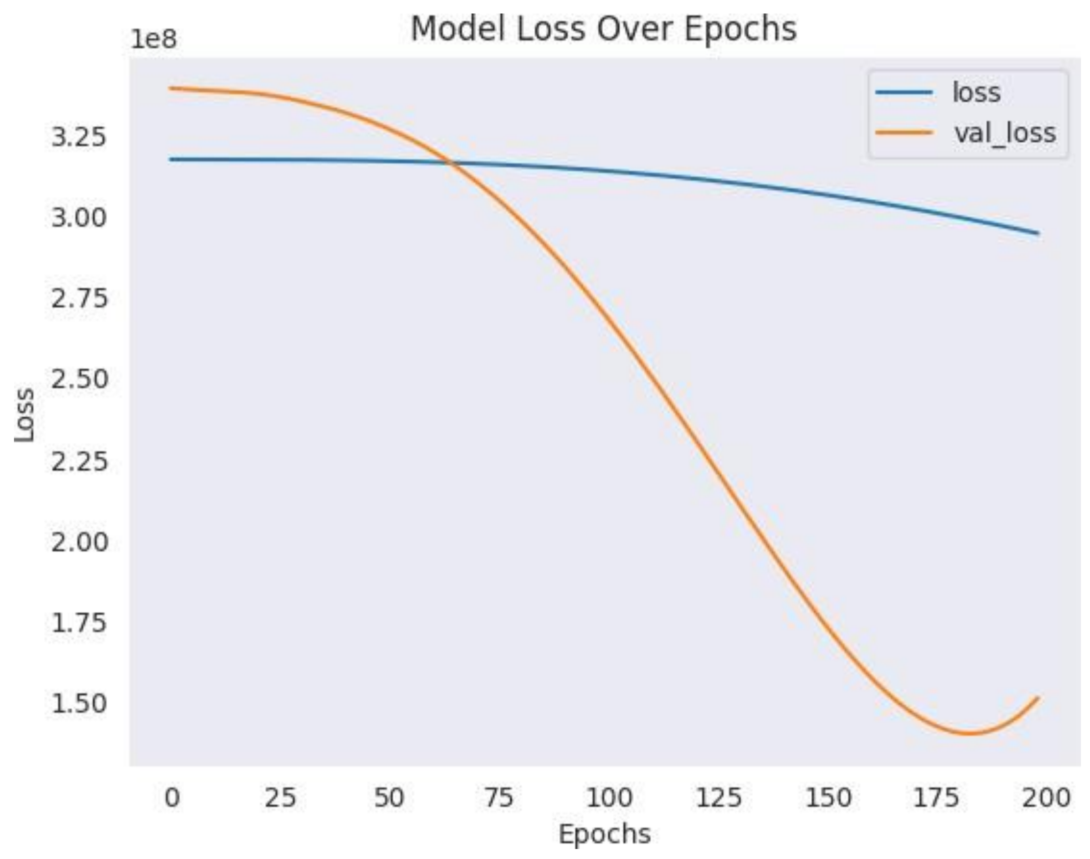
```
Scatter plot of 'bmi' vs 'charges' by 'sex':

Scatter plot of 'bmi' vs 'charges' by 'Smoker' status:
```





```
First few rows of the dataset after dropping unnecessary columns:
   age     bmi  children      charges   male  Smoker  northeast  northwest  \
0   19  27.900         0  16884.92400  False    True      False      False
1   18  33.770         1   1725.55230   True   False      False      False
2   28  33.000         3   4449.46200   True   False      False      False
3   33  22.705         0  21984.47061   True   False      False       True
4   32  28.880         0   3866.85520   True   False      False       True

   southeast
0      False
1       True
2       True
3      False
4      False

Heatmap of the correlation matrix:
```





```
Root Mean Squared Error(RMSE) on Test Data: 12304.159184071554

Root Mean Squared Error(RMSE) on Subset Data: 11313.821472559966
```

Exercise-3

Module name : Understanding and Using CNN : Image recognition

Design a CNN for Image Recognition which includes hyper parameter tuning.

Introduction

Deep Learning models have important applications in image processing. However, one of the challenges in this field is the definition of hyperparameters. Thus, the objective of this work is to propose a rigorous methodology for hyperparameter tuning of Convolutional Neural Network for building construction image classification.

Neural Network Hyperparameters (Deep Learning)

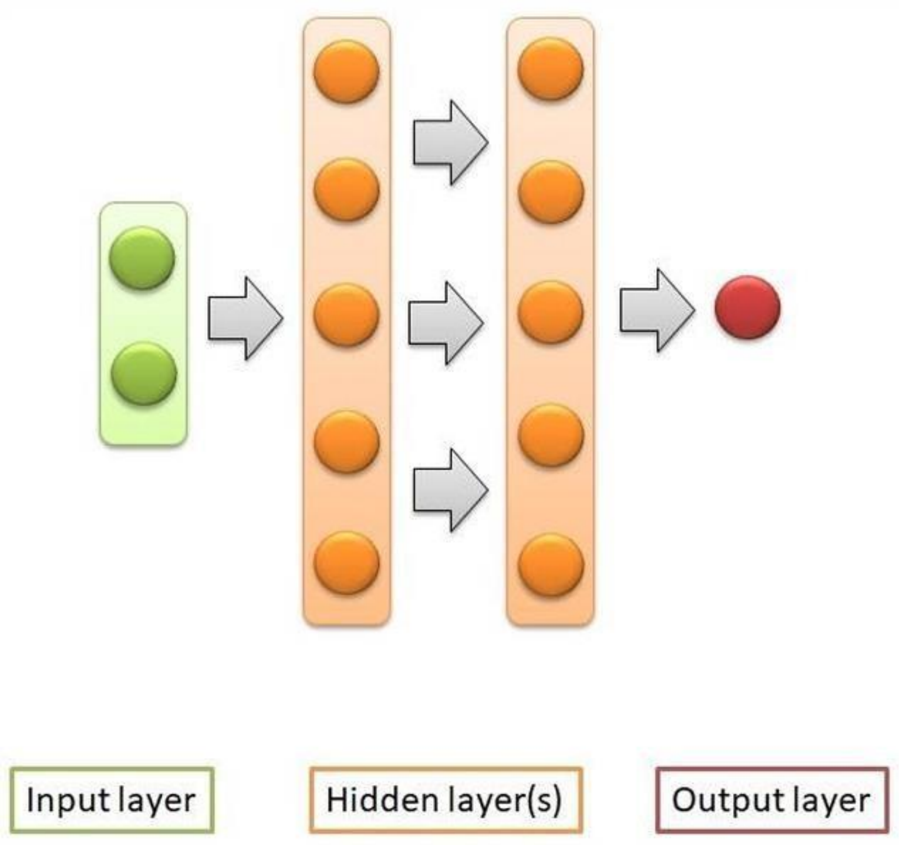
Neural network hyperparameters are like settings you choose before teaching a neural network to do a task. They control things like how many layers the network has, how quickly it learns, and how it adjusts its internal values. Picking the right hyperparameters is important to help the network learn effectively and solve the task accurately. It’s a bit like adjusting the knobs on a machine to make it work just right for a particular job.

Neural Network is a Deep Learning technic to build a model according to training data to predict unseen data using many layers consisting of neurons. This is similar to other Machine Learning algorithms, except for the use of multiple layers. The use of multiple layers is what makes it Deep Learning. Instead of directly building Machine Learning in 1 line, Neural Network requires users to build the architecture before compiling them into a model. Users will have to arrange how many layers and how many nodes or neurons to build. This is not found in other conventional Machine Learning algorithms.

Different datasets require different sets of hyperparameters to predict accurately. But, the large number of hyperparameters makes users difficult to decided which one to choose. There is no answer to how many layers are the most suitable, how many neurons are the best, or which optimizer suits the best for all datasets. Hyperparameter-tuning is important to find the possible best sets ofhyperparameters to build the model from a specific dataset.

Here we will demonstrate the process to tune 2 things of Neural Network: (1) the hyperparameters and (2) the layers. I find it more difficult to find the latter tutorials than the former. The first one is the same as other conventional Machine Learning algorithms. The hyperparameters to tune are the number of neurons, activation function, optimizer, learning rate, batch size, and epochs. The second step is to tune the number of layers. This is what other conventional algorithms do not have. Different layers can affect the accuracy. Fewer layers may give an underfitting result while too many layers may make it overfitting.

For the hyperparameter-tuning demonstration, I use a dataset provided by Kaggle. I build a simple Multilayer Perceptron (MLP) neural network to do a binary classification task with prediction probability. The used package in Python is Keras built on top of Tensorflow. The dataset has an input dimension of 10. There are two hidden layers, followed by one output layer. The accuracy metric is the accuracy score. The callback of EarlyStopping is used to stop the learning process if there is no accuracy improvement in 20 epochs. Below is the illustration.



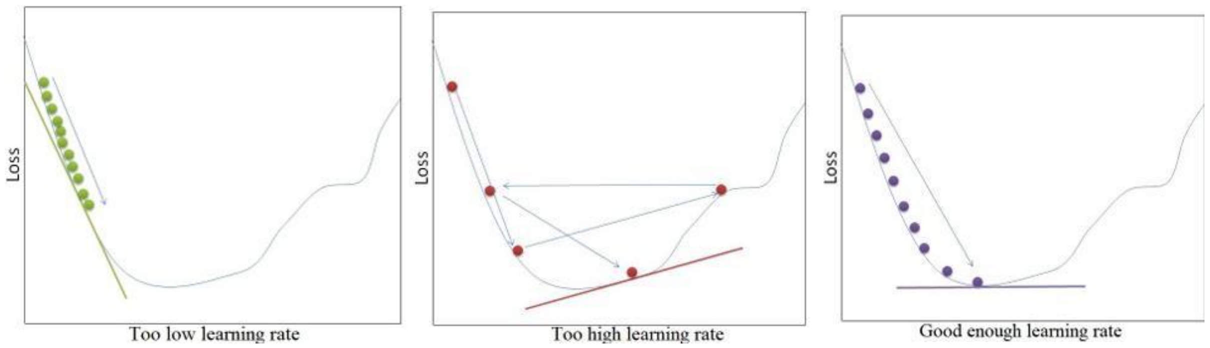
Hyperparameter Tuning in Deep Learning

The first hyperparameter to tune is the number of neurons in each hidden layer. In this case, the number of neurons in every layer is set to be the same. It also can be made different. The number of neurons should be adjusted to the solution complexity. The task with a more complex level to predict needs more neurons. The number of neurons range is set to be from 10 to 100.

An activation function is a parameter in each layer. Input data are fed to the input layer, followed by hidden layers, and the final output layer. The output layer contains the output value. The input values moving from a layer to another layer keep changing according to the activation function. The activation function decides how to compute the input values of a layer into output values. The output values of a layer are then passed to the next layer as input values again. The next layer then computes the values into output values for another layer again. There are 9 activation functions to tune in to this demonstration. Each activation function has its own formula (and graph) to compute the input values

The layers of a neural network are compiled and an optimizer is assigned. The optimizer is responsible to change the learning rate and weights of neurons in the neural network to reach the minimum loss function. Optimizer is very important to achieve the possible highest accuracy or minimum loss. There are 7 optimizers to choose from. Each has a different concept behind it.

One of the hyperparameters in the optimizer is the learning rate. We will also tune the learning rate. Learning rate controls the step size for a model to reach the minimum loss function. A higher learning rate makes the model learn faster, but it may miss the minimum loss function and only reach the surrounding of it. A lower learning rate gives a better chance to find a minimum loss function. As a tradeoff lower learning rate needs higher epochs, or more time and memory capacity resources.



Source code:

For installing keras-tuner (run in another cell)

```
!pip install keras-tuner
```

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import cifar10
from kerastuner.tuners import RandomSearch

# Step 1: Load and preprocess the dataset (CIFAR-10)
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert labels to one-hot encoding
y_train = keras.utils.to_categorical(y_train, 10)  # 10 classes in CIFAR10
y_test = keras.utils.to_categorical(y_test, 10)

# Step 2: Define the CNN architecture
def build_model(hp):
    model = keras.Sequential()

    # Tune the number of convolutional layers and filters
    for i in range(hp.Int('num_conv_layers', 2, 4)):
        model.add(layers.Conv2D(hp.Int(f'conv_{i}_filters', 32, 256, 32), (3, 3),
activation='relu', padding='same'))
        model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Flatten())

    # Tune the number of dense layers and units
    for i in range(hp.Int('num_dense_layers', 1, 3)):
        model.add(layers.Dense(units=hp.Int(f'dense_{i}_units', 64, 512, 32),
activation='relu'))

    # Output layer
    model.add(layers.Dense(10, activation='softmax'))

    # Compile the model
    model.compile(optimizer=keras.optimizers.Adam(hp.Choice('learning_rate', [1e-3,
1e-4])),
    loss='categorical_crossentropy', metrics=['accuracy'])

    return model

# Step 3: Initialize the Keras Tuner RandomSearch and search for the best
hyperparameters
```

```
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,  # Number of different hyperparameter combinations to try
    directory='hyperparameter_tuning',  # Directory to store the results
    project_name='cifar10_tuner'  # Name of the project
)

# Search for the best hyperparameter configuration
tuner.search(x_train, y_train, epochs=1, validation_data=(x_test, y_test))

# Step 4: Get the best hyperparameters and build the final model
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
final_model = build_model(best_hps)

# Step 5: Train and evaluate the final model
final_model.fit(x_train, y_train, epochs=1, validation_data=(x_test, y_test))

# Evaluate the final model on the test set
loss, accuracy = final_model.evaluate(x_test, y_test)
print(f'Test accuracy: {accuracy}')
```

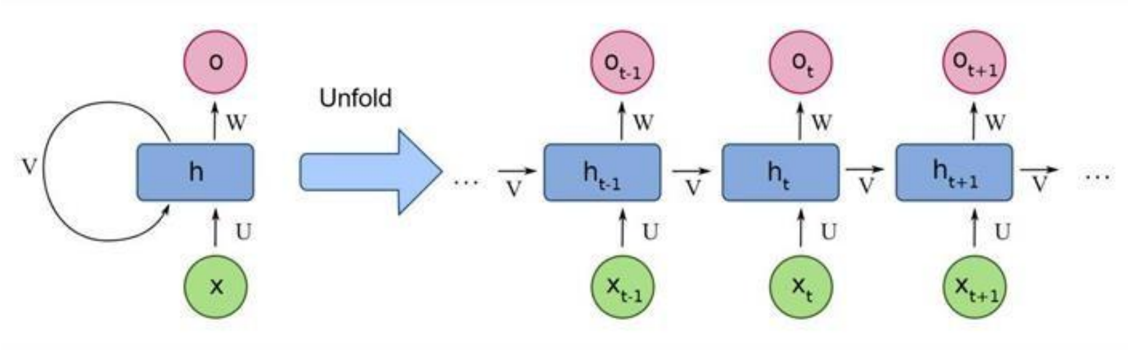
Output:

```
Test accuracy: 0.5800999999046326
```

Exercise-4
Module name: Predicting Sequential Data
Exercise: Implement a Recurrence Neural Network for Predicting Sequential Data.

Introduction

A Deep Learning approach for modelling sequential data is **Recurrent Neural Networks (RNN)**. RNNs were the standard suggestion for working with sequential data before the advent of attention models. Specific parameters for each element of the sequence may be required by a deep feedforward model. It may also be unable to generalize to variable-length sequences.



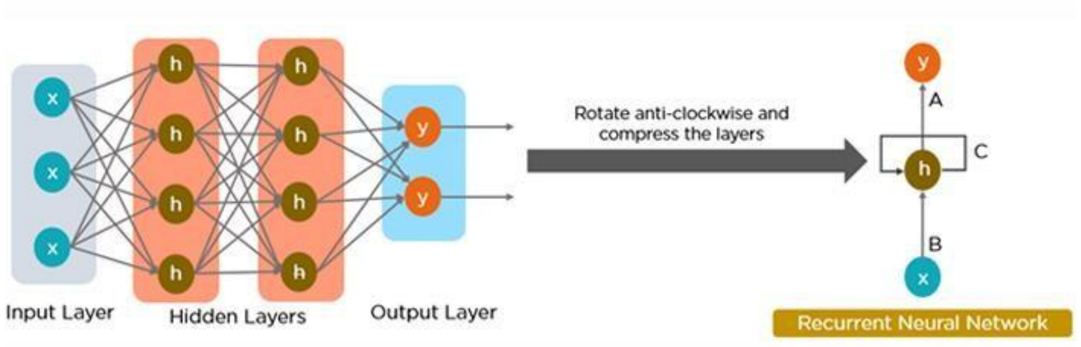
Recurrent Neural Networks use the same weights for each element of the sequence, decreasing the number of parameters and allowing the model to generalize to sequences of varying lengths. RNNs generalize to structured data other than sequential data, such as geographical or graphical data, because of its design.

Recurrent neural networks, like many other deep learning techniques, are relatively old. They were first developed in the 1980s, but we didn’t appreciate their full potential until lately. The advent of long short-term memory (LSTM) in the 1990s, combined with an increase in computational power and the vast amounts of data that we now have to deal with, has really pushed RNNs to the forefront.

What is a Recurrent Neural Network (RNN)?

Neural networks imitate the function of the human brain in the fields of AI, machine learning, and deep learning, allowing computer programs to recognize patterns and solve common issues.
RNNs are a type of neural network that can be used to model sequence data. RNNs,

which are formed from feedforward networks, are similar to human brains in their behaviour. Simply said, recurrent neural networks can anticipate sequential data in a way that other algorithms can't.

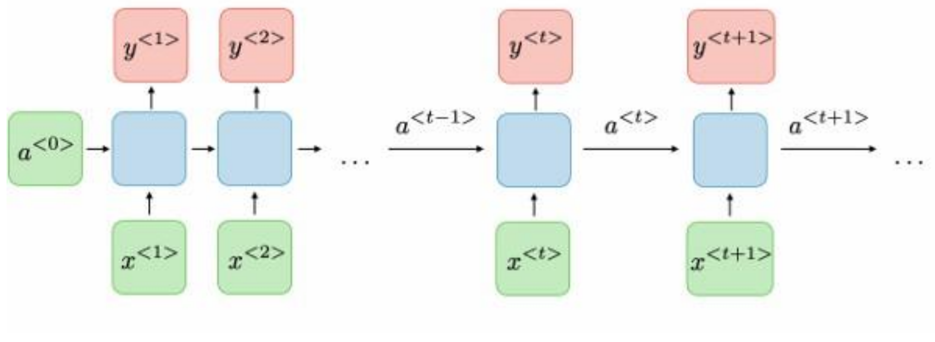


All of the inputs and outputs in standard neural networks are independent of one another, however in some circumstances, such as when predicting thenext word of a phrase, the prior words are necessary, and so the previous
words must be remembered. As a result, RNN was created, which used a Hidden Layer to overcome the problem. The most important component of RNN is the Hidden state, which remembers specific information about a sequence.

RNNs have a Memory that stores all information about the calculations. It employs the same settings for each input since it produces the same outcome by performing the same task on all inputs or hidden layers.

The Architecture of a Traditional RNN

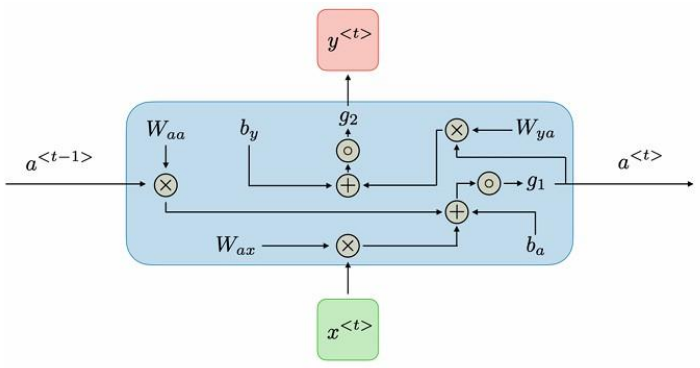
RNNs are a type of neural network that has hidden states and allows past
outputs to be used as inputs. They usually go like this:



For each timestep t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$\boxed{a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)} \quad \text{and} \quad \boxed{y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)}$$

where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 activation functions.



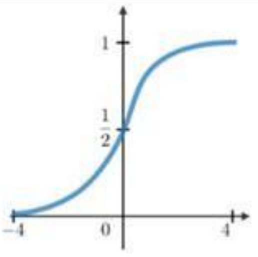
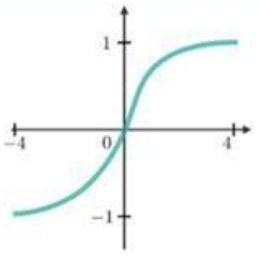
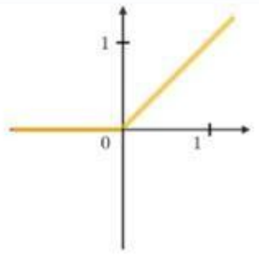
RNN architecture can vary depending on the problem you’re trying to solve. From those with a single input and output to those with many (with variations between).

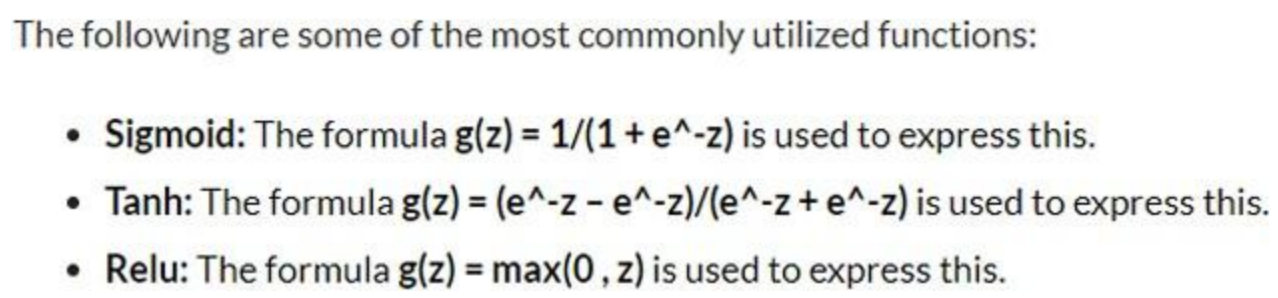
Below are some examples of RNN architectures that can help you better understand this.

• **One To One:** There is only one pair here. A one-to-one architecture is used in traditional neural networks.
• **One To Many:** A single input in a one-to-many network might result in numerous outputs. One too many networks are used in the production of music, for example.
• **Many To One:** In this scenario, a single output is produced by combining many inputs from distinct time steps. Sentiment analysis and emotion identification use such networks, in which the class label is determined by a sequence of words.
• **Many To Many:** For many to many, there are numerous options. Two inputs yield three outputs. Machine translation systems, such as English to French or vice versa translation systems, use many to many networks.

Common Activation Functions

A neuron’s activation function dictates whether it should be turned on or off. Nonlinear functions usually transform a neuron’s output to a number between 0 and 1 or -1 and 1.

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$
		


The following are some of the most commonly utilized functions:

- **Sigmoid:** The formula **g(z) = 1/(1 + e^-z)** is used to express this.
- **Tanh:** The formula **g(z) = (e^-z – e^-z)/(e^-z + e^-z)** is used to express this.
- **Relu:** The formula **g(z) = max(0 , z)** is used to express this.

Source code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN

# Generate some sample sequential data
def generate_data(num_samples, time_steps):
    data = np.arange(0, num_samples * time_steps)
    data = np.sin(data * 0.1)  # A simple sine wave as sequential data
    data = data.reshape(num_samples, time_steps, 1)
    return data

num_samples = 1000
time_steps = 10  # Number of time steps in each input sequence

data = generate_data(num_samples, time_steps)

# Split the data into training and testing sets
train_ratio = 0.8
train_size = int(train_ratio * num_samples)
train_data = data[:train_size]
test_data = data[train_size:]

# Prepare the data for training and testing
train_input = train_data[:, :-1, :]  # Input data for training (all time steps except
the last one)
train_target = train_data[:, 1:, :]  # Target data for training (all time steps
except the first one)

test_input = test_data[:, :-1, :]  # Input data for testing (all time steps except
the last one)
test_target = test_data[:, 1:, :]  # Target data for testing (all time steps except
the first one)

# Build the RNN model
model = Sequential()
model.add(SimpleRNN(32, input_shape=(time_steps - 1, 1)))  # 32 units in the hidden
layer
model.add(Dense(1))  # Output layer with 1 unit
```

```
# Compile the model
model.compile(loss='mean_squared_error', optimizer='adam')

# Train the model
epochs = 50
batch_size = 32
model.fit(train_input, train_target, epochs=epochs, batch_size=batch_size)

# Evaluate the model
loss = model.evaluate(test_input, test_target)
print("Test loss:", loss)


# Make predictions
num_predictions = 10
initial_input = test_input[0:1, :, :]  # Take the first sequence from the test set
and add batch dimension
predictions = []
for _ in range(num_predictions):
    # Predict the next value and update the input sequence
    prediction = model.predict(initial_input)
    predictions.append(prediction[0, 0])  # Store the predicted value (assuming 1
output unit)
    initial_input = np.concatenate((initial_input[:, 1:, :], prediction[:,
np.newaxis, :]), axis=1)
print("Predictions:", predictions)
```

Output:

```
Test loss: 0.03264236077666283
Predictions: [0.55553997, 0.42471844, 0.5136012, 0.57316655, 0.4701597,
0.71725094, 0.5853208, 0.44183043, 0.7576916, 0.41853464]
```

Exercise-5

Module name: Removing noise from the images

Exercise: Implement Multi-Layer Perceptron algorithm for Image denoising hyperparameter tuning.

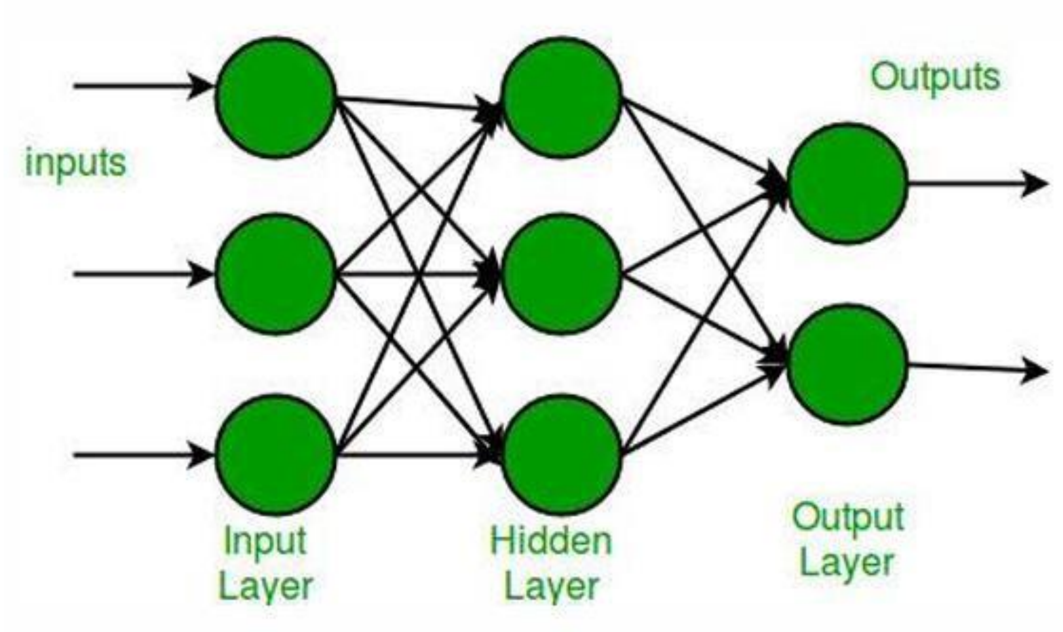
Introduction

Multi-layer perception is also known as MLP. It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together so that the outputs of some neurons are inputs of other neurons.

A gentle introduction to **neural networks and TensorFlow** can be found here:

- Neural Networks
- Introduction to TensorFlow

A multi-layer perceptron has one input layer and for each input, there is one neuron(or node), it has one output layer with a single node for each output and it can have any number of hidden layers and each hidden layer can have any number of nodes. A schematic diagram of a Multi-Layer Perceptron (MLP) is depicted below.



In the multi-layer perceptron diagram above, we can see that there are three inputs and thus three input nodes and the hidden layer has three nodes. The output layer gives two outputs, therefore there are two output nodes. The nodes in the input layer take input and forward it for further process, in the diagram above the nodes in the input layer forwards their output to each of the three nodes in the hidden layer, and in the same way, the hidden layer processes the information and passes it to the output layer.

Every node in the multi-layer perception uses a sigmoid activation function. The sigmoid activation function takes real values as input and converts them to numbers between 0 and 1 using the sigmoid formula.

$$\boldsymbol{(x) = 1 \,/\, (1 + \exp(-x)\,)}$$

Now that we are done with the theory part of multi-layer perception, let’s go ahead and implement some code in python using the TensorFlow library.

Source code:

```
import numpy as np
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error

# Generate synthetic noisy image data for demonstration purposes
np.random.seed(42)
image_size = (100, 100)
num_samples = 1000
noise_level = 0.1
clean_image = np.random.random(image_size)
noisy_image = clean_image + noise_level * np.random.random(image_size)

#  Flatten the image data for MLP input
X_train = noisy_image.flatten().reshape(-1, 1)
y_train = clean_image.flatten()

# Create the MLP regressor
mlp = MLPRegressor(max_iter=200, random_state=42)

# Define the hyperparameters and their values to search
param_grid = {
    'hidden_layer_sizes': [(100,), (50, 50), (25, 25, 25)],
    'activation': ['relu', 'tanh'],
    'alpha': [0.0001, 0.001, 0.01],
}

# Create the GridSearchCV object to perform hyperparameter tuning
grid_search = GridSearchCV(mlp, param_grid, cv=3,
scoring='neg_mean_squared_error', n_jobs=-1)

# Perform hyperparameter tuning
grid_search.fit(X_train, y_train)

# Get the best hyperparameters and the best MLP model
best_params = grid_search.best_params_
best_mlp = grid_search.best_estimator_

# Predict the denoised image using the best model
denoised_image = best_mlp.predict(X_train).reshape(image_size)

# Calculate mean squared error between the denoised image and the clean image
mse = mean_squared_error(clean_image, denoised_image)

print("Best Hyperparameters:", best_params)
print("Mean Squared Error:", mse)
```

Output:

```
Best Hyperparameters: {'activation': 'relu', 'alpha': 0.001,
'hidden_layer_sizes': (50, 50)}
Mean Squared Error: 0.0007964874470608723
```

Exercise-6

Module name : Advanced Deep Learning Architectures

Exercise: Implement Object Detection Using YOLO.

Introduction

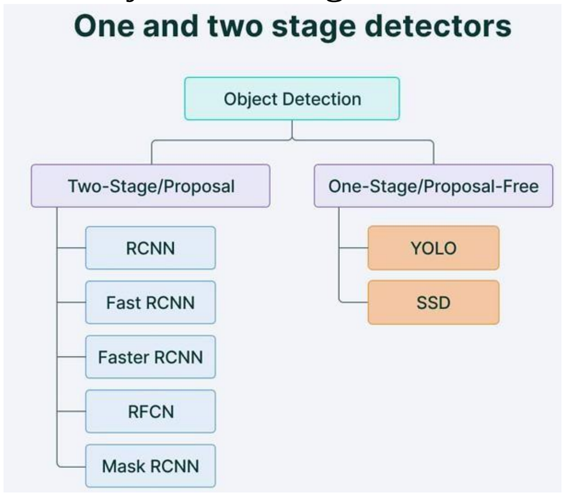
Object detection is a popular task in computer vision.
It deals with localizing a region of interest within an image and classifying this region like a typical image classifier. One image can include several regions of interest pointing to different objects. This makes object detection a more advanced problem of image classification.

YOLO (You Only Look Once) is a popular object detection model known for its speed and accuracy. It was first introduced by Joseph Redmon et al. in 2016 and has since undergone several iterations, the latest being YOLO v7.

What is object detection?

Object detection is a computer vision task that involves identifying and locating objects in images or videos. It is an important part of many applications, such as surveillance, self-driving cars, or robotics. Object detection algorithms can be divided into two main categories: singleshot detectors and two-stage detectors.

One of the earliest successful attempts to address the object detection problem using deep learning was the R-CNN (Regions with CNN features) model, developed by Ross Girshick and his team at Microsoft Research in 2014. This model used a combination of region proposal algorithms and convolutional neural networks (CNNs) to detect and localize objects in images.



Object detection algorithms are broadly classified into two categories based on how many times the same input image is passed through a network.

Single-shot object detection

Single-shot object detection uses a single pass of the input image to make predictions about the presence and location of objects in the image. It processes an entire image in a single pass, making them computationally efficient.

However, single-shot object detection is generally less accurate than other methods, and it's less effective in detecting small objects. Such algorithms can be used to detect objects in real time in resource-constrained environments.

YOLO is a single-shot detector that uses a fully convolutional neural network (CNN) to process an image. We will dive deeper into the YOLO model in the next section.

Two-shot object detection

Two-shot object detection uses two passes of the input image to make predictions about the presence and location of objects. The first pass is used to generate a set of proposals or potential object locations, and the second pass is used to refine these proposals and make final predictions. This approach is more accurate than single-shot object detection but is also more computationally expensive.

Overall, the choice between single-shot and two-shot object detection depends on the specific requirements and constraints of the application.

Generally, single-shot object detection is better suited for real-time applications, while two-shot object detection is better for applications where accuracy is more important.

What is YOLO?

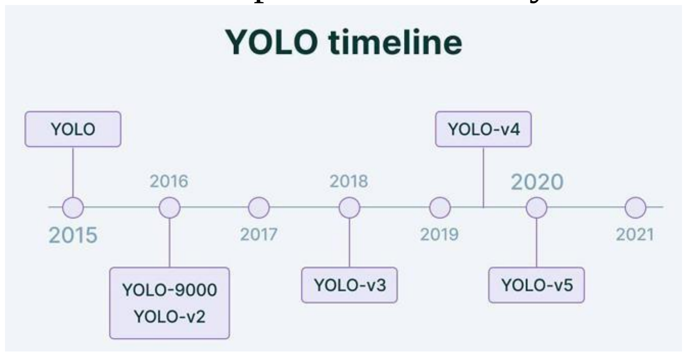
You Only Look Once (YOLO) proposes using an end-to-end neural network that makes predictions of bounding boxes and class probabilities all at once. It differs from the approach taken by previous object detection algorithms, which repurposed classifiers to perform detection.
Following a fundamentally different approach to object detection, YOLO achieved

state-of-theart results, beating other real-time object detection algorithms by a large margin.

While algorithms like Faster RCNN work by detecting possible regions of interest using the Region Proposal Network and then performing recognition on those regions separately, YOLO performs all of its predictions with the help of a single fully connected layer.

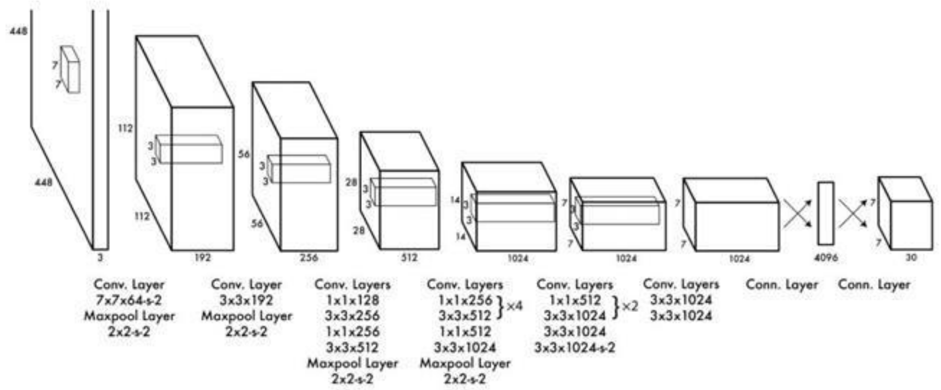
Methods that use Region Proposal Networks perform multiple iterations for the same image, while YOLO gets away with a single iteration.

Several new versions of the same model have been proposed since the initial release of YOLO in 2015, each building on and improving its predecessor. Here's a timeline showcasing YOLO's development in recent years.



How does YOLO work? YOLO Architecture

The YOLO algorithm takes an image as input and then uses a simple deep convolutional neural network to detect objects in the image. The architecture of the CNN model that forms the backbone of YOLO is shown below.


448
448
7
7
3
112
112
3
3
192
56
56
3
3
256
28
28
3
3
512
14
14
3
3
1024
7
7
3
3
1024
7
7
1024
4096
7
7
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 ×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 ×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1 × 1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224 × 224 input image) and then double the resolution for detection.

The first 20 convolution layers of the model are pre-trained using ImageNet by plugging in a temporary average pooling and fully connected layer. Then, this pre-trained model is converted to perform detection since previous research showcased that adding convolution and connected layers to a pre-trained network improves performance. YOLO's final fully connected layer predicts both class probabilities and bounding box coordinates.

YOLO divides an input image into an S × S grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and how accurate it thinks the predicted box is.

YOLO predicts multiple bounding boxes per grid cell. At training time, we only want one bounding box predictor to be responsible for each object. YOLO assigns one predictor to be "responsible" for predicting an object based on which prediction has the highest current IOU with the ground truth. This leads to specialization between the bounding box predictors. Each predictor gets better at forecasting certain sizes, aspect ratios, or classes of objects, improving the overall recall score. One key technique used in the YOLO models is non-maximum suppression (NMS). NMS is a post-processing step that is used to improve the accuracy and efficiency of object detection. In object detection, it is common for multiple bounding boxes to be generated for a single object in an image. These bounding boxes may overlap or be located at different positions, but they all represent the same object. NMS is used to identify and remove redundant or incorrect bounding boxes and to output a single bounding box for each object in the image. Now, let us look into the improvements that the later versions of YOLO have brought to the parent model.

Download the following files for the code:

Yolov3.weights

Yolov3.cfg

Coco.names

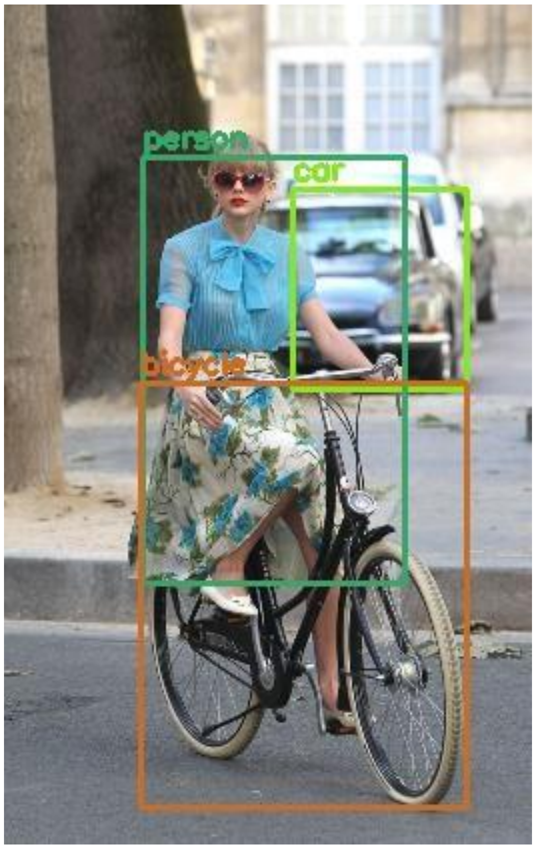
Source code:

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

# Load Yolo
print("LOADING YOLO")
net = cv2.dnn.readNet("/content/drive/MyDrive/4-1 DL lab/yolov3.weights",
"/content/drive/MyDrive/4-1 DL lab/yolov3.cfg")

# Save all the names in file to the list classes
classes = []
with open("/content/drive/MyDrive/4-1 DL lab/coco.names", "r") as f:
    classes = [line.strip() for line in f.readlines()]

# Get layers of the network
layer_names = net.getLayerNames()

# Determine the output layer names from the YOLO model
output_layers = [layer_names[i - 1] for i in
net.getUnconnectedOutLayers()]
print("YOLO LOADED")

# Capture frame-by-frame
img = cv2.imread("/content/drive/MyDrive/4-1 DL lab/sample2.jpg")
img = cv2.resize(img, None, fx=0.4, fy=0.4)
height, width, channels = img.shape
print("Input Image:")
cv2_imshow(img)
# Using blob function of opencv to preprocess image
blob = cv2.dnn.blobFromImage(img, 1 / 255.0, (416, 416), swapRB=True,
crop=False)

# Detecting objects
net.setInput(blob)
outs = net.forward(output_layers)

# Showing informations on the screen
class_ids = []
confidences = []
boxes = []
for out in outs:
    for detection in out:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]
        if confidence > 0.3:
            # Object detected
            center_x = int(detection[0] * width)
            center_y = int(detection[1] * height)
            w = int(detection[2] * width)
            h = int(detection[3] * height)

            # Rectangle coordinates
            x = int(center_x - w / 2)
            y = int(center_y - h / 2)

            boxes.append([x, y, w, h])
            confidences.append(float(confidence))
            class_ids.append(class_id)

# We use NMS function in opencv to perform Non-maximum Suppression
# we give it score threshold and nms threshold as arguments.
indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
colors = np.random.uniform(0, 255, size=(len(classes), 3))
for i in range(len(boxes)):
    if i in indexes:
        x, y, w, h = boxes[i]
        label = str(classes[class_ids[i]])
        color = colors[class_ids[i]]

        # Draw the bounding box
        cv2.rectangle(img, (x, y), (x + w, y + h), color, 2)

        # Draw the label
        cv2.putText(img, label, (x, y - 5), cv2.FONT_HERSHEY_SIMPLEX, 1/2, color, 2)
        print(f"Detected object: {label}, confidence: {confidences[i]}, box: {x},
{y}, {w}, {h}")

# Display the image
cv2_imshow(img)
cv2.waitKey(0)
```

Output:

```
LOADING YOLO
YOLO LOADED
Input Image:
```

```
Detected object: car, confidence: 0.9670625925064087, box: 144, 92, 87, 100
Detected object: person, confidence: 0.975329577922821, box: 69, 76, 131, 213
Detected object: bicycle, confidence: 0.9977372884750366, box: 68, 189, 163, 212
```



```
-1
```

Exercise-7

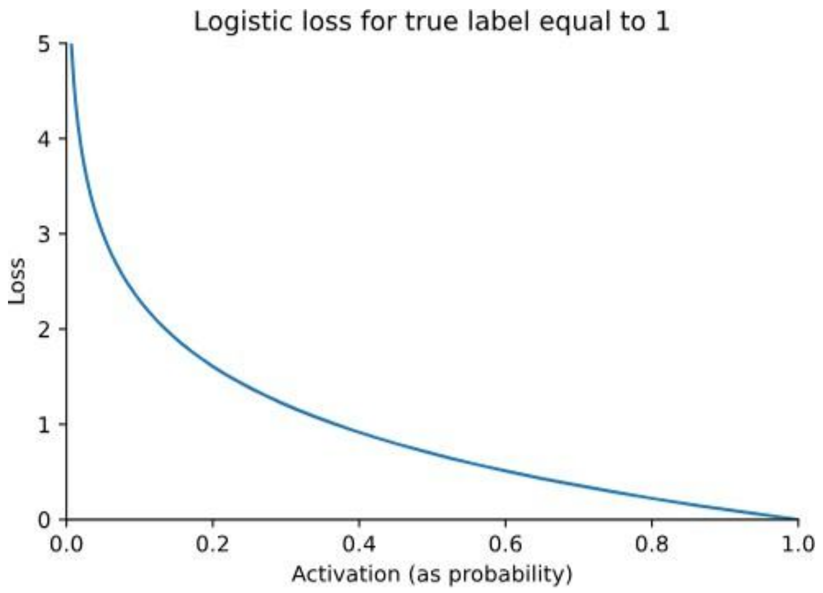
Module name : Optimization of Training in Deep Learning

Exercise: Design a Deep learning Network for Robust Bi-Tempered Logistic Loss.

Introduction

Logistic loss

In order to understand the bi-tempered logistic loss, let's recap the standard version of this loss function. Logistic loss is the most common loss function used for training binary classifiers. You can look up the exact formula and other details here. The plot below displays logistic loss responses for different network predictions:



As can be seen, the loss is zero if the response is correct (the activation value is equal to 1), and grows exponentially when the error increases, reaching its limit of infinitely large loss when the activation is equal to zero (i.e the loss is an unbounded function of error). This shape results in large errors being much more heavily penalized than relatively small ones (e.g the error for 0.45 is seven times greater than the one for 0.9).

In the context of noisy data, it's reasonable to expect that the network will produce huge errors for incorrectly labelled (i.e. noisy) data. As the majority of data annotations are correct, the network progressively learns the general function underlying the data. When encountering a mislabelled sample, a well-trained network will produce a correct label that doesn't match with the wrong annotation and causes the error to be very large. By allowing the losses in such scenarios to grow infinitely large, the training may become unstable, as the network will tend to "overreact" to such misclassifications (in extreme cases, this might lead to gradients exploding).

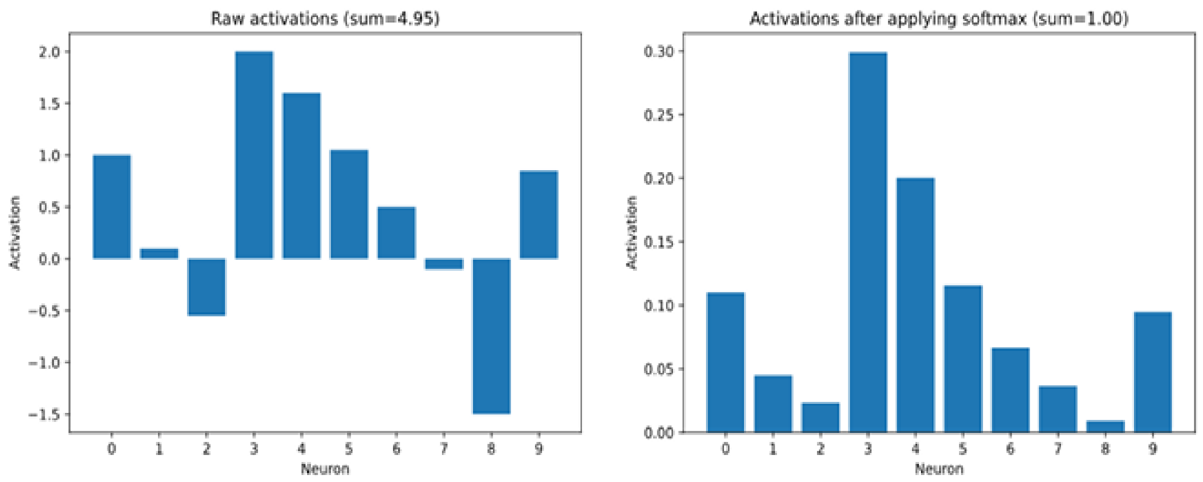
An important note on naming - logistic loss is the same as binary cross-entropy. Cross-entropy is a general term applicable to multiclass classification, but you might encounter the same function called logistic loss in the literature, which is not entirely correct as the logistic loss is for binary classification only.

Softmax

Logistic loss work by comparing the expected probability distribution to the network response. In order for this to be feasible, it has to be ensured that the network outputs can be interpreted as a probability distribution over multiple classes, i.e. individual elements are from the range of [0,1] and add up to one. To meet this requirement, at the end of a typical classification network, a softmax layer is used.

Softmax is a pretty straightforward function that normalizes a vector of numbers in such a way as to conform to the aforementioned probability axioms. The normalization uses the exponential function, so it doesn't preserve relative ratios between individual elements. For the exact formula and more details, you can refer to this article.

The plots below show the results of passing a set of neuron activations through the softmax function.



Notice how all the values became positive, they sum up to 1, and the relative magnitude of each value is preserved, i.e. if a given value is the 3rd largest in the input vector, it stays at this position after the transformation. If we consider this set of 10 neurons to be an output layer of a classifier network, we can read from the second plot that the network considers the fourth class (denoted with "3") to be the most probable, with the probability of around 30%. The ninth class, with the lowest value prior to the transformation, stays the least probable.

Bi-tempered logistic loss

Bi-tempered logistic loss is a modification of the standard logistic loss.

The authors propose a slightly altered, parametric version of the typical logistic loss. The new function has some desirable features which make it useful for training neural networks that are more robust against noisy data. This means that imperfections in the training data don't impact the model quality as much as they would normally if the network was trained with the typical loss.

The function differs from its standard counterpart by being parametric. Under the hood, it uses parametric alternatives to the logarithm and softmax functions, called tempered logarithm and tempered exponential functions. Let's look into each of them to build an understanding of their impact on the final loss function.

Procedure :

Designing a deep learning network involves several steps, and incorporating a custom loss function like the Bi-Tempered Logistic Loss requires careful consideration. Below, I'll outline the steps you can take to design a deep learning network using the Bi-Tempered Logistic Loss:

### Step 1: Define the Network Architecture
Choose an architecture suitable for your specific task. This could be a Convolutional Neural Network (CNN) for computer vision tasks, a Recurrent Neural Network (RNN) for sequential data, or a feedforward neural network for simpler tasks. Make sure to tailor the architecture to the requirements of your problem.

### Step 2: Import Libraries and Modules
```python
import tensorflow as tf
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.models import Sequential
```

### Step 3: Create the Network
```python
def create_network(input_shape, num_classes):
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(num_classes, activation='softmax')
        ])
    return model
```
Replace this architecture with one that suits your specific problem.

### Step 4: Define Bi-Tempered Logistic Loss
You'll need to define the Bi-Tempered Logistic Loss function. Make sure to define the loss and the associated gradient.

```python
def bi_tempered_logistic_loss(y_true, y_pred, t1=0.8, t2=1.2,label_smoothing=0.1):
    """
    Bi-Tempered Logistic Loss.
    """
    y_pred = tf.clip_by_value(y_pred, 1e-7, 1.0 - 1e-7)
    temp1 = (1 - y_true) * (y_pred ** (t1 - 1)) / t1
    temp2 = (1 - y_true) * ((1 - y_pred) ** (t2 - 1)) / t2
    loss_values = temp1 + temp2
    loss = tf.reduce_sum(loss_values, axis=-1)
    loss = tf.reduce_mean(loss)
    return loss
```
### Step 5: Compile the Model
```python
model = create_network(input_shape, num_classes)
```

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

### Step 6: Train the Model
```python
model.fit(x_train, y_train, epochs=num_epochs, batch_size=batch_size , validation_data=(x_val, y_val))
```
### Step 7: Evaluate the Model
```python
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy*100:.2f}%')
```

### Step 8: Fine-tuning (Optional)
Depending on the performance of your model, you may need to fine-tune the architecture, hyperparameters, or even experiment with different variants of the Bi-Tempered Logistic Loss.

Remember to preprocess your data appropriately, use data augmentation if necessary, and adjust hyperparameters like learning rate, batch size, etc., based on your specific problem.

This is a basic template to get you started. You might need to customize and adapt these steps based on the specific details of your problem, dataset, and requirements.

Source code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential

# Step 1: Define a simple network architecture
def create_network(input_shape):
    model = Sequential([
    Dense(64, activation='relu', input_shape=(input_shape,)),
    Dense(1, activation='sigmoid') ])  # Added the sigmoid activation here
    return model
```

```
# Step 4: Define Bi-Tempered Logistic Loss
def bi_tempered_logistic_loss(y_true, y_pred, t1=0.8, t2=1.2):
    y_pred = tf.clip_by_value(y_pred, 1e-7, 1.0 - 1e-7)
    temp1 = (1 - y_true) * (y_pred ** (t1 - 1)) / t1
    temp2 = (1 - y_true) * ((1 - y_pred) ** (t2 - 1)) / t2
    loss_values = temp1 + temp2
    loss = tf.reduce_sum(loss_values, axis=-1)
    loss = tf.reduce_mean(loss)
    return loss

# Step 5: Compile the model
input_shape = 10  # Example input shape for a feature vector of size 10
model = create_network(input_shape)
model.compile(optimizer='adam', loss=bi_tempered_logistic_loss)

# Step 6: Generate synthetic data for binary classification
np.random.seed(0)
X = np.random.rand(1000, input_shape)
y = np.random.rand(1000)  # Changed y to be float32

# Step 7: Train the model
model.fit(X, y, epochs=10, batch_size=32)

# Step 8: Evaluate the model
test_loss = model.evaluate(X, y)
print(f'Test loss: {test_loss}')
```

Output:

```
Test loss: 0.6651512384414673
```

Exercise-8
Module name : Advanced CNN
Exercise: Build AlexNet using Advanced CNN

Introduction

The main content of this exercise will present how the AlexNet Convolutional Neural Network(CNN) architecture is implemented using TensorFlow and Keras.

Here are some of the key learning objectives from this article:

1. Introduction to neural network implementation with Keras and TensorFlow
2. Data preprocessing with TensorFlow
3. Training visualization with TensorBoard
4. Description of standard machine learning terms and terminologies
5. AlexNet Implementation

AlexNet CNN is probably one of the simplest methods to approach understanding deep learning concepts and techniques.

AlexNet is not a complicated architecture when it is compared with some state of the art CNN architectures that have emerged in the more recent years.

AlexNet is simple enough for beginners and intermediate deep learning practitioners to pick up some good practices on model implementation techniques.

Source code:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

def alexnet_model(input_shape, num_classes):
    model = Sequential([
        # First Convolutional Layer
        Conv2D(96, (11,11), strides=(4,4), activation='relu',
input_shape=input_shape, padding='valid'),
        MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'),

        # Second Convolutional Layer
        Conv2D(256, (5,5), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'),

        # Third Convolutional Layer
        Conv2D(384, (3,3), activation='relu', padding='same'),

        # Fourth Convolutional Layer
        Conv2D(384, (3,3), activation='relu', padding='same'),

        # Fifth Convolutional Layer
        Conv2D(256, (3,3), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='valid'),

        # Flatten the CNN output to feed it into a Fully-Connected Dense Layer
        Flatten(),

        #  First  Dense  Layer
        Dense(4096, activation='relu'),

        # Add a Dropout layer to reduce overfitting
        Dropout(0.5),

        #  Second Dense Layer
        Dense(4096, activation='relu'),

        # Add another Dropout layer
        Dropout(0.5),

        # Output Layer
        Dense(num_classes, activation='softmax')
    ])

    return model

# Define the input shape and number of classes
input_shape = (227, 227, 3)  # Adjust input shape if necessary
num_classes = 1000  # Number of ImageNet classes

# Create an instance of the model
model = alexnet_model(input_shape, num_classes)

# Print the model summary
model.summary()
```

Output:

```
Model: "sequential_2"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                      │ (None, 55, 55, 96)          │          34,944 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d (MaxPooling2D)         │ (None, 27, 27, 96)          │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_1 (Conv2D)                    │ (None, 27, 27, 256)         │         614,656 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_1 (MaxPooling2D)       │ (None, 13, 13, 256)         │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_2 (Conv2D)                    │ (None, 13, 13, 384)         │         885,120 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_3 (Conv2D)                    │ (None, 13, 13, 384)         │       1,327,488 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_4 (Conv2D)                    │ (None, 13, 13, 256)         │         884,992 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_2 (MaxPooling2D)       │ (None, 6, 6, 256)           │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ flatten (Flatten)                    │ (None, 9216)                │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_3 (Dense)                      │ (None, 4096)                │      37,752,832 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout (Dropout)                    │ (None, 4096)                │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_4 (Dense)                      │ (None, 4096)                │      16,781,312 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout_1 (Dropout)                  │ (None, 4096)                │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_5 (Dense)                      │ (None, 1000)                │       4,097,000 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 62,378,344 (237.95 MB)
 Trainable params: 62,378,344 (237.95 MB)
 Non-trainable params: 0 (0.00 B)
```

Exercise-9

Module name : Autoencoders Advanced

Exercise: Demonstration of Application of Autoencoders

Introduction

Autoencoders are a type of neural network that can be used for various tasks, such as dimensionality reduction, anomaly detection, and even generation of new data. I'll demonstrate how autoencoders can be applied for dimensionality reduction using a simple example.

Example: Dimensionality Reduction with Autoencoders

Let's say you have a dataset with high-dimensional features, and you want to reduce the dimensionality while retaining as much information as possible.

In this example, we:

1. Generate synthetic high-dimensional data (100 features per sample).
2. Define an autoencoder architecture with one hidden layer in both the encoder and decoder.
3. Compile the autoencoder with Mean Squared Error (MSE) loss.
4. Train the autoencoder on the same data as input and output (unsupervised learning).
5. Extract the encoder part as it represents the learned low-dimensional representation of the data.

Keep in mind that in a real-world scenario, you'd replace the synthetic data with your actual dataset and adapt the architecture and hyperparameters based on your specific problem.

The encoded_features will contain the reduced-dimensional representations of the input data. You can use these for downstream tasks like clustering, classification, or visualization.

Source code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# Generate synthetic high-dimensional data
np.random.seed(0)
X = np.random.rand(1000, 100)

# Define the autoencoder architecture
input_dim = X.shape[1]
encoding_dim = 10

# Encoder
input_layer = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_layer)

# Decoder
decoded = Dense(input_dim, activation='sigmoid')(encoded)

# Autoencoder model
autoencoder = Model(input_layer, decoded)

# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder
autoencoder.fit(X, X, epochs=50, batch_size=32)

# Use the encoder for dimensionality reduction
encoder = Model(input_layer, encoded)
encoded_features = encoder.predict(X)

print(f"Original shape: {X.shape}")
print(f"Encoded shape: {encoded_features.shape}")
```

Output:

```
Original shape: (1000, 100)
Encoded shape: (1000, 10)
```

Exercise-10

Module name : Advanced GANs

Exercise: Demonstration of GAN

Introduction

Generative Adversarial Networks (GANs) are a type of deep learning model used for generating new data samples that are similar to a given dataset. GANs consist of two neural networks, a generator and a discriminator, which are trained simultaneously through adversarial training.

In this example, I'll demonstrate how to create a simple GAN to generate handwritten digits (similar to those in the MNIST dataset).

In this example, we:

Load the MNIST dataset and preprocess it.

Define the generator, discriminator, and the GAN.

Compile the discriminator and the GAN.

Train the GAN in a loop.

After training, you'll see generated images saved in the current directory with filenames like gan_generated_image_epoch_X.png.

Keep in mind that this is a very basic GAN example. Real-world GAN applications require more advanced architectures, training techniques, and careful hyperparameter tuning.

Source code:

```python
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, LeakyReLU, BatchNormalization, Reshape,
Flatten, Input
from tensorflow.keras.optimizers import Adam

# Load MNIST data
(X_train, _), (_, _) = mnist.load_data()
X_train = (X_train.astype(np.float32) - 127.5) / 127.5
X_train = X_train.reshape(X_train.shape[0], 784)

# Define GAN architecture
def build_generator(latent_dim):
    model = Sequential([
        Dense(128, input_dim=latent_dim),
        LeakyReLU(alpha=0.2),
        BatchNormalization(momentum=0.8),
        Dense(256),
        LeakyReLU(alpha=0.2),
        BatchNormalization(momentum=0.8),
        Dense(512),
        LeakyReLU(alpha=0.2),
        BatchNormalization(momentum=0.8),
        Dense(784, activation='tanh')
    ])
    return model

def build_discriminator(image_shape):
    model = Sequential([
        Dense(512, input_dim=image_shape),
        LeakyReLU(alpha=0.2),
        Dense(256),
        LeakyReLU(alpha=0.2),
        Dense(1, activation='sigmoid')
    ])
    return model

def build_gan(generator, discriminator):
    model = Sequential([generator, discriminator])
    return model

# Define hyperparameters
latent_dim = 100
image_shape = 784
```

```python
# Build and compile the discriminator
discriminator = build_discriminator(image_shape)
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5),
metrics=['accuracy'])

# Build and compile the generator
generator = build_generator(latent_dim)
discriminator.trainable = False
gan = build_gan(generator, discriminator)
gan.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))

# Define training loop
def train_gan(epochs=1, batch_size=128):
    batch_count = X_train.shape[0] // batch_size

    for e in range(epochs):
        for _ in range(batch_count):
            noise = np.random.normal(0, 1, size=[batch_size, latent_dim])
            generated_images = generator.predict(noise)
            image_batch = X_train[np.random.randint(0, X_train.shape[0],
size=batch_size)]

            X = np.concatenate([image_batch, generated_images])
            y_dis = np.zeros(2 * batch_size)
            y_dis[:batch_size] = 0.9

            discriminator.trainable = True
            d_loss = discriminator.train_on_batch(X, y_dis)

            noise = np.random.normal(0, 1, size=[batch_size, latent_dim])
            y_gen = np.ones(batch_size)
            discriminator.trainable = False
            g_loss = gan.train_on_batch(noise, y_gen)

        print(f"Epoch {e} - D loss: {d_loss[0]} | D accuracy: {100 * d_loss[1]} | G
loss: {g_loss}")

        if e % 1 == 0:
            plot_generated_images(e, generator, latent_dim)

# Function to plot generated images
def plot_generated_images(epoch, generator, latent_dim, examples=10, dim=(1, 10),
figsize=(10, 1)):
    noise = np.random.normal(0, 1, size=[examples, latent_dim])
    generated_images = generator.predict(noise)
    generated_images = generated_images.reshape(examples, 28, 28)

    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i], interpolation='nearest', cmap='gray_r')
        plt.axis('off')

    plt.tight_layout()
    plt.savefig(f'gan_generated_image_epoch_{epoch}.png')

# Train the GAN
train_gan(epochs=1, batch_size=128)
```

Output:

```
Epoch 0 - D loss: 0.3723898231983185 | D accuracy:
44.20572817325592 | G loss: [array(0.37238982, dtype=float32),
array(0.37238982, dtype=float32), array(0.44205728,
dtype=float32)]
```