SGPC's
Guru Nanak Institute of Management Studies
(Management Institute of G N Khalsa College), Matunga, Mumbai – 400 019

INDEX

Subject: MCALE322 – Deep Learning (Elective)

Sr. No.	PRACTICAL PROBLEM STATEMENT	DATE	SIGNATURE
1	Introduction to Tensor flow / Keras -Importing Libraries and Modules.	22-07-2025	
2	Loading the dataset, splitting dataset into training and testing data sets.	31-07-2025	
3	Implementation of Data preprocessing techniques.	01-08-2025	
4	Implementation of Artificial Neural Networks – a. McCulloch-Pitts neuron with ANDNOT function, b. Back propagation Network for XOR function with Binary Input and Output.	04-08-2025 06-08-2025	
5	Implementation of Regularization Techniques a. Dataset Augmentation, b. Early Stopping, c. Dropout.	11-08-2025 14-08-2025	
6	Implementation and analysis of Deep Neural - network algorithm: Convolutional neural network (CNN) a. Object identification and classification, b. Image recognition.	01-09-2025 03-09-2025	
7	Implementation and analysis of Deep Neural network algorithm: Recurrent neural network (RNN) – a. Character Recognition and b. Web Traffic Image classification.	04-09-2025 19-09-2025	
8	LSTM Network: Sentiment analysis using LSTM	24-09-2025	

Practical 1

Aim: Introduction to Tensor flow / Keras -Importing Libraries and Modules.

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

[2]:
```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense,Flatten
from tensorflow.keras.datasets import mnist
```

[3]:
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train/255.0
x_test = x_test/255.0
```

[4]:
```
model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(128,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

```
/usr/local/lib/python3.12/dist-
packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

[5]:
```
from IPython.display import Image
Image(filename = "/content/drive/Othercomputers/My Laptop/Documents/MCA␣
↪2024-2026/Sem 3/Practicals/MCALE32 - Deep Learning Lab/Practical 1/image.
↪png")
```

[5]:

[6]:
```
model.
  compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
model.fit(x_train,y_train,epochs=10)
model.evaluate(x_test,y_test)
```

```
Epoch 1/10
1875/1875          6s 2ms/step -
accuracy: 0.8770 - loss: 0.4325
Epoch 2/10
1875/1875          4s 2ms/step -
accuracy: 0.9634 - loss: 0.1235
Epoch 3/10
1875/1875          5s 2ms/step -
```

```
accuracy: 0.9763 - loss: 0.0801
Epoch 4/10
1875/1875            4s 2ms/step -
accuracy: 0.9819 - loss: 0.0600
Epoch 5/10
1875/1875            4s 2ms/step -
accuracy: 0.9868 - loss: 0.0443
Epoch 6/10
1875/1875            5s 2ms/step -
accuracy: 0.9891 - loss: 0.0350
Epoch 7/10
1875/1875            4s 2ms/step -
accuracy: 0.9918 - loss: 0.0283
Epoch 8/10
1875/1875            4s 2ms/step -
accuracy: 0.9935 - loss: 0.0217
Epoch 9/10
1875/1875            4s 2ms/step -
accuracy: 0.9946 - loss: 0.0176
Epoch 10/10
1875/1875            5s 3ms/step -
accuracy: 0.9952 - loss: 0.0147
313/313              2s 3ms/step -
accuracy: 0.9734 - loss: 0.0965
```

[6]: [0.08700499683618546, 0.9758999943733215]

Practical 2

Aim: Loading the dataset, splitting dataset into training and testing data sets.

[1]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder,␣
↪OneHotEncoder
from tensorflow.keras.utils import to_categorical
import tensorflow as tf
```

[2]:
```
data = {
    'Age' : [25,30,np.nan,35,40],
    'Salary'  : [50000,60000,65000,np.nan  ,70000],
    'Gender' : ['Male','Female','Female','Male','Male'],
    'Department' : ['HR','Finance','IT','IT','HR'],
    'Purchased' : ['No','Yes','No','Yes','Yes']
}
```

[3]:
```
df = pd.DataFrame(data)
print(df)
```

```
    Age   Salary  Gender Department Purchased
0  25.0  50000.0    Male         HR        No
1  30.0  60000.0  Female    Finance       Yes
2   NaN  65000.0  Female         IT        No
3  35.0      NaN    Male         IT       Yes
4  40.0  70000.0    Male         HR       Yes
```

[4]:
```
df['Age'].fillna(df['Age'].mean(),inplace=True)
df['Salary'].fillna(df['Salary'].median(),inplace=True)
```

```
/tmp/ipython-input-645796033.py:1: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
```

```
instead, to perform the operation inplace on the original object.


  df['Age'].fillna(df['Age'].mean(),inplace=True)
/tmp/ipython-input-645796033.py:2: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  df['Salary'].fillna(df['Salary'].median(),inplace=True)
```

[5]:
```
df['Age'] = df['Age'].astype(int)
df['Salary'] = df['Salary'].astype(int)
```

[6]:
```
le_gender = LabelEncoder()
df['Gender'] = le_gender.fit_transform(df['Gender'])

df = pd.get_dummies(df,columns=['Department'])
le_purchase = LabelEncoder()
df['Purchased'] = le_purchase.fit_transform(df['Purchased'])
```

[7]:
```
scaler = MinMaxScaler()
df[['Age','Salary']]  = scaler.fit_transform(df[['Age','Salary']])
print("Preprocessed DataFrame:")
df
```

```
Preprocessed  DataFrame:
```

[7]:
```
        Age  Salary  Gender  Purchased  Department_Finance  Department_HR  \
0  0.000000   0.000       1          0               False           True
1  0.333333   0.500       0          1                True          False
2  0.466667   0.750       0          0               False          False
3  0.666667   0.625       1          1               False          False
4  1.000000   1.000       1          1               False           True

   Department_IT
0          False
1          False
2           True
3           True
```

```
4         False
```

[8]:
```
X = df.drop('Purchased',axis=1)
y = df['Purchased']

print(X.dtypes)

X = X.astype(float)
```

```
Age                    float64
Salary                 float64
Gender                   int64
Department_Finance        bool
Department_HR             bool
Department_IT             bool
dtype: object
```

[9]:
```
X_train,X_test,y_train,y_test  =  train_test_split(X,y,test_size=0.
  ↪2,random_state=42)
```

[10]:
```
X_train = X_train.to_numpy()
X_test = X_test.to_numpy()
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Practical 3

Aim: Implementation of Data preprocessing techniques.

[1]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder,␣
↪OneHotEncoder
from tensorflow.keras.utils import to_categorical
import tensorflow as tf
```

[2]:
```
data = {
    'Age' : [25, 30, np.nan, 35, 40],
    'Salary' : [50000, 60000, 65000, np.nan, 70000],
    'Gender' : ['Male', 'Female', 'Female', 'Male', 'Male'],
    'Department' : ['HR', 'Finance', 'IT', 'IT', 'HR'],
    'Purchased' : ['No', 'Yes', 'No', 'Yes', 'Yes']
}
```

[3]:
```
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
```

```
Original DataFrame:
    Age   Salary  Gender Department Purchased
0  25.0  50000.0    Male         HR        No
1  30.0  60000.0  Female    Finance       Yes
2   NaN  65000.0  Female         IT        No
3  35.0      NaN    Male         IT       Yes
4  40.0  70000.0    Male         HR       Yes
```

[4]:
```
df['Age'].fillna(df['Age'].mean(), inplace=True)
df['Salary'].fillna(df['Salary'].median(), inplace=True)
```

```
/tmp/ipython-input-4184148624.py:1: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.
```

```
For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  df['Age'].fillna(df['Age'].mean(), inplace=True)
/tmp/ipython-input-4184148624.py:2: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  df['Salary'].fillna(df['Salary'].median(), inplace=True)
```

[5]:
```
df['Age'] = df['Age'].round().astype(int)
df['Salary'] = df['Salary'].round().astype(int)
```

[6]:
```
le_gender = LabelEncoder()
df['Gender'] = le_gender.fit_transform(df['Gender'])

# One-hot encoding for Department
df = pd.get_dummies(df, columns=['Department'])

# Encode target variable
le_purchase = LabelEncoder()
df['Purchased'] = le_purchase.fit_transform(df['Purchased'])
```

[7]:
```
scaler = MinMaxScaler()
df[['Age', 'Salary']] = scaler.fit_transform(df[['Age', 'Salary']])

print("\nPreprocessed DataFrame:")
print(df)
```

```
Preprocessed DataFrame:
        Age  Salary  Gender  Purchased  Department_Finance  Department_HR  \
0  0.000000   0.000       1          0               False           True
1  0.333333   0.500       0          1                True          False
2  0.466667   0.750       0          0               False          False
3  0.666667   0.625       1          1               False          False
4  1.000000   1.000       1          1               False           True
```

```
   Department_IT
0          False
1          False
2           True
3           True
4          False
```

[8]:
```
X = df.drop('Purchased', axis=1)
y = df['Purchased']

X = X.astype(float)
```

[9]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[10]:
```
X_train = X_train.to_numpy()
X_test = X_test.to_numpy()
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

[11]:
```
print("\n Data Preprocessing Completed Successfully!")
print("Training Data Shape:", X_train.shape)
print("Testing Data Shape:", X_test.shape)
```

```
 Data Preprocessing Completed Successfully!
Training Data Shape: (4, 6)
Testing Data Shape: (1, 6)
```

Practical 4

Aim: Implementation of Artificial Neural Networks –

McCulloch-Pitts neuron with ANDNOT function

[1]:
```
import pandas as pd
import numpy as np
```

[2]:
```
def mcp_andnot(input_row,weights,threshold):
    inputs = np.array([input_row['A'],input_row['B']])
    weighted_sum = np.dot(inputs,weights)
    if weighted_sum >= threshold:
        return 1
    else:
        return 0
```

[3]:
```
data = {
    'A': [1, 1, 0, 0],
    'B': [1, 0, 1, 0]
}
df = pd.DataFrame(data)
```

[4]:
```
weights = np.array([1,-1])
threshold = 1
```

[5]:
```
outputs = []
for i in range(len(df)):
    output = mcp_andnot(df.iloc[i],weights,threshold)
    outputs.append(output)
```

[6]:
```
df["Output"] = outputs
```

[7]:
```
print(f"MCP Model: A AND NOT B: \n{df}")
```

```
MCP Model: A AND NOT B:
   A  B  Output
0  1  1       0
1  1  0       1
2  0  1       0
3  0  0       0
```

Back propagation Network for XOR function with Binary Input and Output.

[1]:
```
import pandas as pd
import numpy as np
```

[2]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

[3]:
```
def sigmoid_derivative(x):
    return x * (1 - x)
```

[4]:
```
X    =    np.array([[1,1],[1,0],[0,1],[0,0]])
y = np.array([[0],[1],[1],[0]])
```

[5]:
```
np.random.seed(42)
```

[6]:
```
input_layer_size = 2
hidden_layer_size = 3
output_layer_size = 1
```

[7]:
```
w1 = np.random.uniform(-1,1,(input_layer_size, hidden_layer_size))
w2 = np.random.uniform(-1,1,(hidden_layer_size, output_layer_size))
```

[8]:
```
b1 = np.random.uniform(-1, 1, (1, hidden_layer_size))
b2 = np.random.uniform(-1, 1, (1, output_layer_size))
```

[9]:
```
epochs = 10000
learning_rate = 0.1
```

[10]:
```
for epoch in range(epochs):
    z1 = np.dot(X, w1) + b1
    a1 = sigmoid(z1)

    z2 = np.dot(a1, w2) + b2
    a2 = sigmoid(z2)

    error = y - a2
    d_a2 = error * sigmoid_derivative(a2)
```

```
    error_hidden = d_a2.dot(w2.T)
    d_a1 = error_hidden * sigmoid_derivative(a1)

    w2 += a1.T.dot(d_a2) * learning_rate
    b2 += np.sum(d_a2, axis=0, keepdims=True) * learning_rate

    w1 += X.T.dot(d_a1) * learning_rate
    b1 += np.sum(d_a1, axis=0, keepdims=True) * learning_rate

final_output = sigmoid(np.dot(sigmoid(np.dot(X,w1) + b1),w2) + b2)
```

[11]:
```
results = pd.DataFrame(np.hstack((X,final_output.round())),columns = ['Input 1','Input 2','Predicted Output'])
```

[12]:
```
print("XOR Prediction After Training:\n")
print(results)
```

```
XOR Prediction After Training:

   Input 1  Input 2  Predicted Output
0      1.0      1.0               0.0
1      1.0      0.0               1.0
2      0.0      1.0               1.0
3      0.0      0.0               0.0
```

Practical zzz5

Aim: Implementation of Regularization Techniques

Dataset Augmentation

[1]:
```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
```

[2]:
```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train,x_test = x_train/255.0, x_test/255.0
```

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 **4s**
0us/step

[3]:
```
datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

[4]:
```
sample_image = x_train[0:1]
i = 0
plt.figure(figsize=(10,10))
for batch in datagen.flow(sample_image, batch_size=1):
    plt.subplot(3,3,i+1)
    plt.imshow(batch[0])
    plt.axis('off')
    i += 1
    if i % 9 == 0:
        break
plt.show()
```

Early Stopping,

[1]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
import numpy as np
```

[2]:
```
X_train = np.random.rand(1000, 10)
y_train = np.random.randint(0,2, size=(1000, 1))
X_val = np.random.rand(100, 10)
y_val = np.random.randint(0,2, size=(100, 1))
```

[3]:
```
model = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[4]:
```
model.compile(optimizer='adam', loss='binary_crossentropy',
  metrics=['accuracy'])
```

[5]:
```
early_stopping = EarlyStopping(monitor='val_loss', patience=5,
  restore_best_weights=True)
```

[6]:
```
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_val,
  y_val), callbacks=[early_stopping])
```

```
Epoch 1/10
32/32          1s 10ms/step -
accuracy: 0.5153 - loss: 0.6931 - val_accuracy: 0.5000 - val_loss: 0.6927
Epoch 2/10
32/32          0s 4ms/step -
accuracy: 0.5271 - loss: 0.6891 - val_accuracy: 0.5400 - val_loss: 0.6925
```

```
Epoch 3/10
32/32 0s 4ms/step -
accuracy: 0.5532 - loss: 0.6852 - val_accuracy: 0.5000 - val_loss: 0.6943
Epoch 4/10
32/32 0s 4ms/step -
accuracy: 0.5442 - loss: 0.6886 - val_accuracy: 0.5400 - val_loss: 0.6938
Epoch 5/10
32/32 0s 4ms/step -
accuracy: 0.5282 - loss: 0.6880 - val_accuracy: 0.5100 - val_loss: 0.6954
Epoch 6/10
32/32 0s 4ms/step -
accuracy: 0.5451 - loss: 0.6841 - val_accuracy: 0.4800 - val_loss: 0.6952
Epoch 7/10
32/32 0s 4ms/step -
accuracy: 0.5425 - loss: 0.6866 - val_accuracy: 0.4700 - val_loss: 0.6957
```

[7]:

```
= model
```

```
4/4 0s 9ms/step -
accuracy: 0.5431 - loss: 0.6903
Validation Accuracy: 0.5400
Validation Loss: 0.6925
```

Dropout.

[1]:
```
import numpy as np
import random
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
```

[2]:
```
X_train = np.random.rand(1000,20)
y_train = np.random.randint(0,2, size=(1000,1))
X_val = np.random.rand(1000,20)
y_val = np.random.randint(0,2, size=(1000,1))
```

[3]:
```
model = Sequential([
    Dense(64, activation='relu', input_shape=(20,)),
    Dropout(0.5),
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[4]:
```
model.compile(optimizer=Adam(), loss='binary_crossentropy',
 metrics=['accuracy'])
```

[5]:
```
history = model.fit(X_train, y_train, validation_data=(X_val, y_val))
```

```
32/32 2s 11ms/step -
accuracy: 0.5063 - loss: 0.7197 - val_accuracy: 0.4960 - val_loss: 0.6938
```

[6]:
```
loss,accuracy = model.evaluate(X_val, y_val)
print(f"Validation loss: {loss}")
print(f"Validation accuracy: {accuracy}")
```

```
32/32 0s 2ms/step -
accuracy: 0.4826 - loss: 0.6936
Validation loss: 0.6937578916549683
Validation accuracy: 0.4959999918937683
```

Practical 6

Aim: Implementation and analysis of Deep Neural - network algorithm: Convolutional neural network (CNN)

Object identification and classification

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
 ↪Dropout
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint,
 ↪ReduceLROnPlateau
```

[2]:
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32')/255.0
x_test = x_test.astype('float32')/255.0

x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

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

[3]:
```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])
```

```
model.compile(optimizer='adam',   loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,      **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204,928
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 225,034 (879.04 KB)

Trainable params: 225,034 (879.04 KB)

Non-trainable params: 0 (0.00 B)

[4]:
```
callBacks = [
    ModelCheckpoint('best_model.h5',  monitor='val_accuracy',␣
 ↪save_best_only=True),
    EarlyStopping(monitor='val_accuracy',    patience=3,restore_best_weights=True),
```

```
    ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_lr=6)
]
```

[5]:
```
history = model.fit(
    x_train, y_train,
    validation_split = 0.2,
    epochs=10,
    batch_size=128,
    callbacks=callBacks,
    verbose = 1
)
```

```
Epoch 1/10
375/375              0s 107ms/step -
accuracy: 0.7898 - loss: 0.6770

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.

375/375              46s 115ms/step -
accuracy: 0.7901 - loss: 0.6761 - val_accuracy: 0.9762 - val_loss: 0.0794 -
learning_rate: 0.0010
Epoch 2/10
375/375              0s 107ms/step -
accuracy: 0.9642 - loss: 0.1170

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.

375/375              81s 114ms/step -
accuracy: 0.9643 - loss: 0.1169 - val_accuracy: 0.9835 - val_loss: 0.0538 -
learning_rate: 0.0010
Epoch 3/10
375/375              0s 104ms/step -
accuracy: 0.9762 - loss: 0.0804

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.

375/375              42s 111ms/step -
accuracy: 0.9762 - loss: 0.0803 - val_accuracy: 0.9874 - val_loss: 0.0443 -
```

learning_rate: 0.0010
Epoch 4/10
375/375 **0s** 105ms/step -
accuracy: 0.9812 - loss: 0.0612

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

375/375 **42s** 112ms/step -
accuracy: 0.9812 - loss: 0.0612 - val_accuracy: 0.9887 - val_loss: 0.0400 -
learning_rate: 0.0010
Epoch 5/10
375/375 **0s** 104ms/step -
accuracy: 0.9852 - loss: 0.0509

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

375/375 **42s** 111ms/step -
accuracy: 0.9852 - loss: 0.0509 - val_accuracy: 0.9890 - val_loss: 0.0395 -
learning_rate: 0.0010
Epoch 6/10
375/375 **0s** 104ms/step -
accuracy: 0.9859 - loss: 0.0460

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

375/375 **82s** 111ms/step -
accuracy: 0.9859 - loss: 0.0460 - val_accuracy: 0.9898 - val_loss: 0.0379 -
learning_rate: 0.0010
Epoch 7/10
375/375 **85s** 119ms/step -
accuracy: 0.9877 - loss: 0.0406 - val_accuracy: 0.9898 - val_loss: 0.0370 -
learning_rate: 0.0010
Epoch 8/10
375/375 **79s** 110ms/step -
accuracy: 0.9891 - loss: 0.0343 - val_accuracy: 0.9883 - val_loss: 0.0422 -
learning_rate: 0.0010
Epoch 9/10
375/375 **0s** 101ms/step -
accuracy: 0.9901 - loss: 0.0310

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

375/375 **41s** 110ms/step -
accuracy: 0.9901 - loss: 0.0310 - val_accuracy: 0.9902 - val_loss: 0.0375 -
learning_rate: 0.0010
Epoch 10/10
375/375 **0s** 101ms/step -
accuracy: 0.9902 - loss: 0.0308

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

375/375 **41s** 109ms/step -
accuracy: 0.9902 - loss: 0.0308 - val_accuracy: 0.9904 - val_loss: 0.0377 -
learning_rate: 0.0010

[6]:
```
test_loss, test_acc = model.evaluate(x_test, y_test,verbose=0)
print(f'Test accuracy: {test_acc:4f}')
```

Test accuracy: 0.992100

[7]:
```
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label = 'Val Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Trainning vs Validation Accuracy')
plt.legend()
plt.show()
```

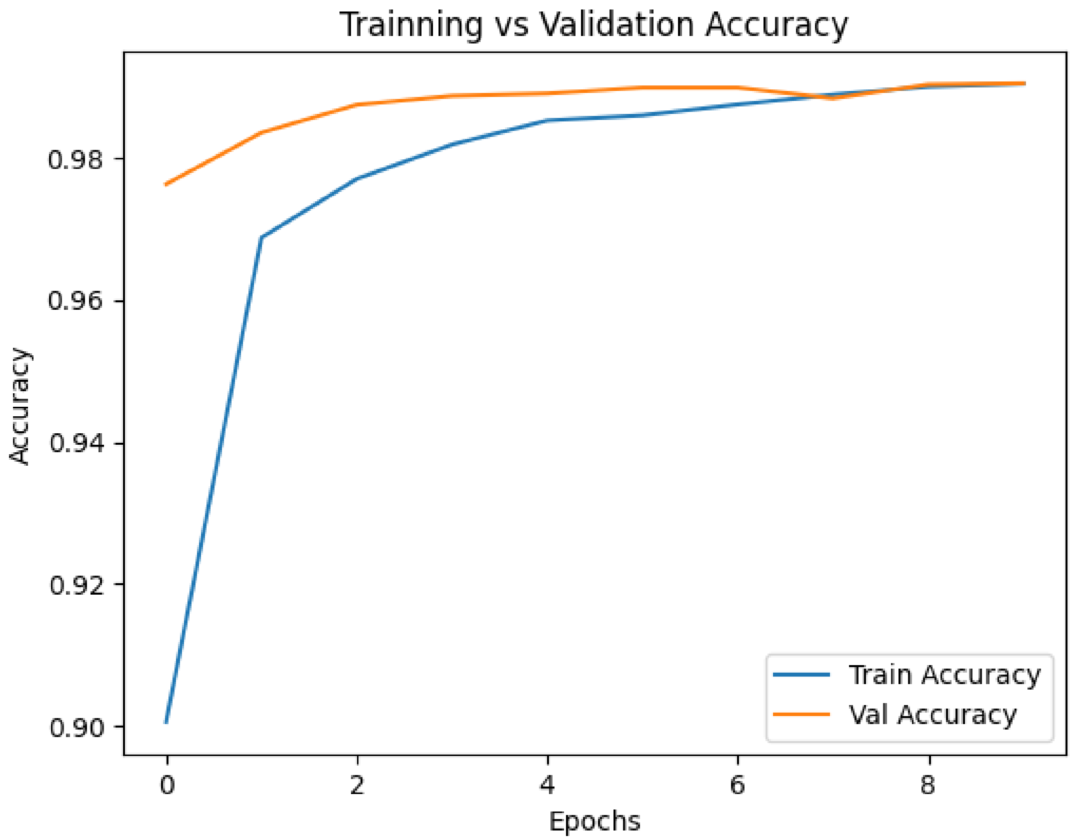

Trainning vs Validation Accuracy
Accuracy
0.98
0.96
0.94
0.92
0.90
0
2
4
6
8
Epochs
Train Accuracy
Val Accuracy

Image recognition.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, Model, Input
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping,␣
↪ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
np.random.seed(42)
tf.random.set_seed(42)
```

[2]:
```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
y_train = y_train.ravel()
y_test = y_test.ravel()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.
↪2, random_state=42, stratify=y_train)
```

[3]:
```
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    fill_mode='reflect'
)
datagen.fit(x_train)
```

[4]:
```
input_shape = (32, 32, 3)
num_classes = 10
inputs = Input(shape=input_shape)

x = layers.Conv2D(32, (3, 3), padding='same')(inputs)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
conv1 = layers.Conv2D(32, (3, 3), padding='same')(x)
```

```
x = layers.BatchNormalization()(conv1)
x = layers.Activation('relu')(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.25)(x)

conv2 = layers.Conv2D(64, (3, 3), padding='same')(x)
x = layers.BatchNormalization()(conv2)
x = layers.Activation('relu')(x)
x = layers.Conv2D(64, (3, 3), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.25)(x)

x = layers.Conv2D(128, (3, 3), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.25)(x)

x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(128, activation='relu')(x)
x = layers.Dropout(0.25)(x)
outputs = layers.Dense(num_classes, activation='softmax')(x)
```

[5]:
```
model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy'])
model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 32, 32, 3)	0
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
activation (Activation)	(None, 32, 32, 32)	0
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9,248
batch_normalization_1	(None, 32, 32, 32)	128

(BatchNormalization)		
activation_1 (Activation)	(None, 32, 32, 32)	0
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 64)	256
activation_2 (Activation)	(None, 16, 16, 64)	0
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 64)	256
activation_3 (Activation)	(None, 16, 16, 64)	0
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 128)	512
activation_4 (Activation)	(None, 8, 8, 128)	0
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 128)	0
dense (Dense)	(None, 128)	16,512
dropout_3 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 158,506 (619.16 KB)

Trainable params: 157,866 (616.66 KB)

Non-trainable params: 640 (2.50 KB)

[6]:
```
callbacks = [
    ModelCheckpoint('best_cnn.h5', monitor='val_accuracy', save_best_only=True,␣
 ↪verbose=1),
    EarlyStopping(monitor='val_accuracy', patience=10,␣
 ↪restore_best_weights=True, verbose=1),
    ReduceLROnPlateau(monitor='val_loss', patience=5, factor=0.5, min_lr=1e-6,␣
 ↪verbose=1)
]
```

[7]:
```
batch_size = 128
epochs = 10
steps_per_epoch = max(1, len(x_train) // batch_size)

history = model.fit(
    datagen.flow(x_train, y_train, batch_size=batch_size),
    steps_per_epoch=steps_per_epoch,
    epochs=epochs,
    validation_data=(x_val, y_val),
    callbacks=callbacks,
    verbose=1
)
```

```
/usr/local/lib/python3.12/dist-
packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121:
UserWarning: Your `PyDataset` class should call `super()._init_(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()
Epoch 1/10
312/312          0s 990ms/step -
accuracy: 0.3160 - loss: 1.8514
Epoch 1: val_accuracy improved from -inf to 0.17100, saving model to best_cnn.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.
```

```
312/312                 333s 1s/step -
accuracy: 0.3163 - loss: 1.8507 - val_accuracy: 0.1710 - val_loss: 2.9679 -
learning_rate: 0.0010
Epoch 2/10
  1/312                 4:08 798ms/step -
accuracy: 0.4766 - loss: 1.3370

/usr/local/lib/python3.12/dist-
packages/keras/src/trainers/epoch_iterator.py:116: UserWarning: Your input ran
out of data; interrupting training. Make sure that your dataset or generator can
generate at least `steps_per_epoch * epochs` batches. You may need to use the
`.repeat()` function when building your dataset.
  self._interrupted_warning()


Epoch 2: val_accuracy did not improve from 0.17100
312/312                 17s 54ms/step -
accuracy: 0.4766 - loss: 1.3370 - val_accuracy: 0.1698 - val_loss: 3.0837 -
learning_rate: 0.0010
Epoch 3/10
312/312                 0s 995ms/step -
accuracy: 0.4999 - loss: 1.3774
Epoch 3: val_accuracy improved from 0.17100 to 0.53570, saving model to
best_cnn.h5

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.

312/312                 331s 1s/step -
accuracy: 0.4999 - loss: 1.3773 - val_accuracy: 0.5357 - val_loss: 1.2565 -
learning_rate: 0.0010
Epoch 4/10
  1/312                 4:15 823ms/step -
accuracy: 0.5156 - loss: 1.3456
Epoch 4: val_accuracy did not improve from 0.53570
312/312                 18s 54ms/step -
accuracy: 0.5156 - loss: 1.3456 - val_accuracy: 0.5296 - val_loss: 1.2924 -
learning_rate: 0.0010
Epoch 5/10
312/312                 0s 989ms/step -
accuracy: 0.5603 - loss: 1.2179
Epoch 5: val_accuracy did not improve from 0.53570
312/312                 362s 1s/step -
accuracy: 0.5603 - loss: 1.2179 - val_accuracy: 0.5333 - val_loss: 1.4467 -
learning_rate: 0.0010
Epoch 6/10
  1/312                 4:18 832ms/step -
```

```
accuracy: 0.5391 - loss: 1.2019
Epoch 6: val_accuracy did not improve from 0.53570
312/312                 17s 53ms/step -
accuracy: 0.5391 - loss: 1.2019 - val_accuracy: 0.5038 - val_loss: 1.6084 -
learning_rate: 0.0010
Epoch 7/10
312/312                 0s 980ms/step -
accuracy: 0.5982 - loss: 1.1170
Epoch 7: val_accuracy improved from 0.53570 to 0.56930, saving model to
best_cnn.h5

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.

312/312                 323s 1s/step -
accuracy: 0.5982 - loss: 1.1170 - val_accuracy: 0.5693 - val_loss: 1.2282 -
learning_rate: 0.0010
Epoch 8/10
  1/312                 4:17 827ms/step -
accuracy: 0.6406 - loss: 0.9842
Epoch 8: val_accuracy did not improve from 0.56930
312/312                 17s 54ms/step -
accuracy: 0.6406 - loss: 0.9842 - val_accuracy: 0.5556 - val_loss: 1.2785 -
learning_rate: 0.0010
Epoch 9/10
312/312                 0s 988ms/step -
accuracy: 0.6201 - loss: 1.0601
Epoch 9: val_accuracy improved from 0.56930 to 0.63550, saving model to
best_cnn.h5

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.

312/312                 326s 1s/step -
accuracy: 0.6201 - loss: 1.0600 - val_accuracy: 0.6355 - val_loss: 1.0508 -
learning_rate: 0.0010
Epoch 10/10
  1/312                 4:14 819ms/step -
accuracy: 0.6250 - loss: 1.0512
Epoch 10: val_accuracy improved from 0.63550 to 0.63780, saving model to
best_cnn.h5

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
```

recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
'my_model.keras')`.

312/312 **21s** 66ms/step -
accuracy: 0.6250 - loss: 1.0512 - val_accuracy: 0.6378 - val_loss: 1.0545 -
learning_rate: 0.0010
Restoring model weights from the end of the best epoch: 10.

[8]:
```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print('Test Accuracy:', test_acc)
```

313/313 - 18s - 57ms/step - accuracy: 0.6301 - loss: 1.0666
Test Accuracy: 0.6301000118255615

[9]:
```
y_pred_probs = model.predict(x_test)
y_pred = np.argmax(y_pred_probs, axis=1)

class_names = 
 ['airplane','automobile','bird','cat','deer','dog','frog','horse','ship','truck']
```

313/313 **17s** 55ms/step

[10]:
```
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=class_names))
```

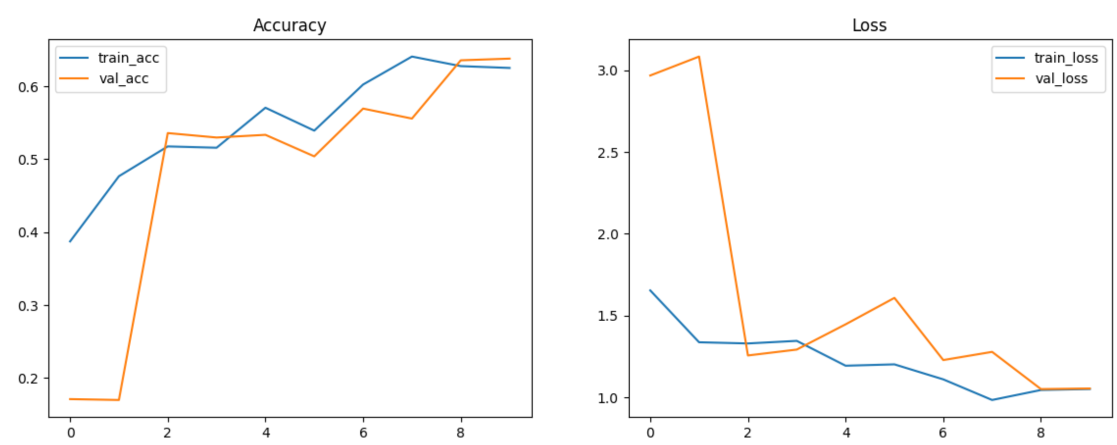
Classification Report:

	precision	recall	f1-score	support
airplane	0.70	0.61	0.65	1000
automobile	0.65	0.85	0.74	1000
bird	0.74	0.24	0.36	1000
cat	0.52	0.35	0.42	1000
deer	0.53	0.63	0.58	1000
dog	0.65	0.45	0.53	1000
frog	0.71	0.70	0.70	1000
horse	0.54	0.86	0.66	1000
ship	0.83	0.71	0.77	1000
truck	0.59	0.90	0.71	1000
accuracy			0.63	10000
macro avg	0.65	0.63	0.61	10000
weighted avg	0.65	0.63	0.61	10000

[11]:
```
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:\n", cm)
```

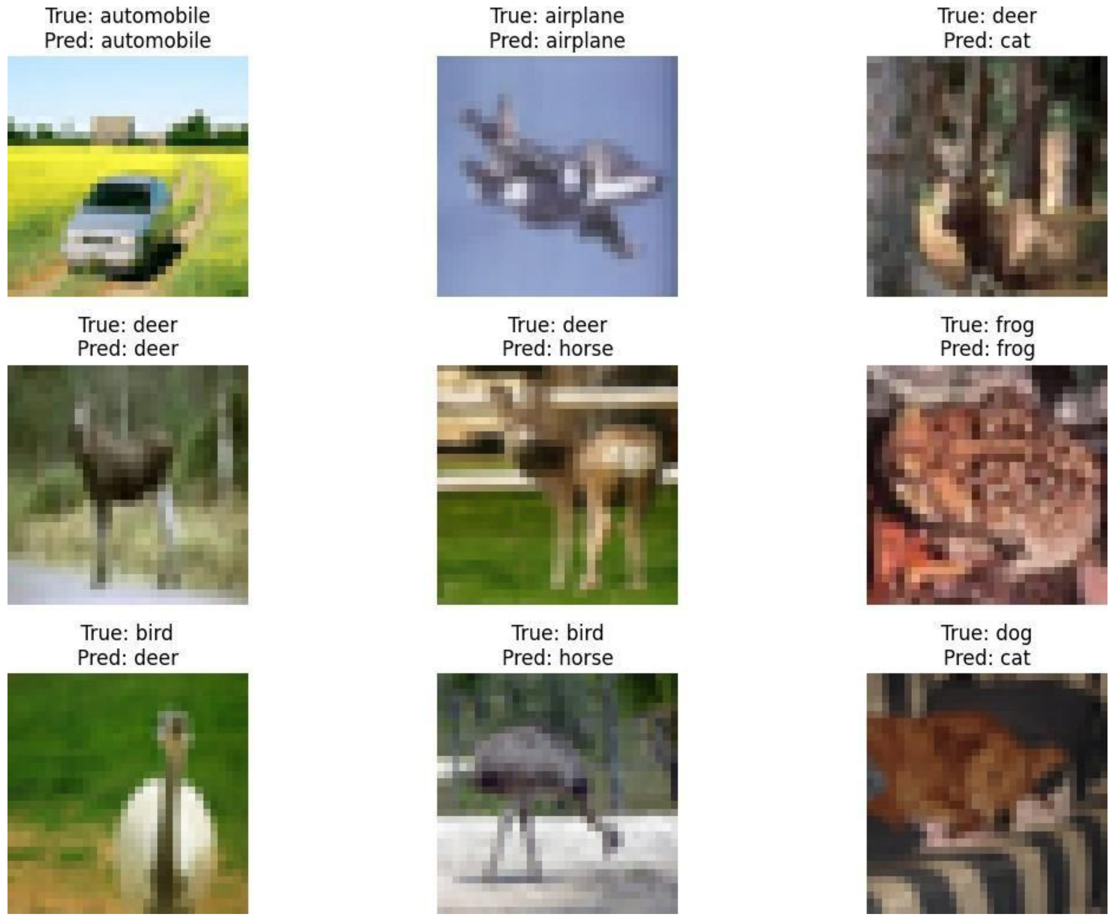
```
Confusion Matrix:
 [[606 121  33   6   8   3   8  29  87  99]
 [  5 848   0   0   3   1   0   5   3 135]
 [118  28 240  54 224  46 101 141  12  36]
 [ 21  61  12 353  85 153  89 117  18  91]
 [ 19   9  13  33 633   9  64 196   7  17]
 [ 11  21   8 147  77 449  23 211   4  49]
 [  8  34  11  58 100  13 701  25   5  45]
 [  8   6   4  19  46  15   3 859   2  38]
 [ 69  96   1   4   8   0   1   3 714 104]
 [  6  75   2   0   1   0   0   9   9 898]]
```

[12]:
```python
plt.figure(figsize=(14,5))
plt.subplot(1,2,1)
plt.plot(history.history.get('accuracy',   []),   label='train_acc')
plt.plot(history.history.get('val_accuracy', []), label='val_acc')
plt.title('Accuracy')
plt.legend()

plt.subplot(1,2,2)
plt.plot(history.history.get('loss', []), label='train_loss')
plt.plot(history.history.get('val_loss', []), label='val_loss')
plt.title('Loss')
plt.legend()
plt.show()
```



[13]:
```python
def plot_samples(x, y_true, y_pred, class_names, n=9):
    idxs = np.random.choice(len(y_true), size=n, replace=False)
    plt.figure(figsize=(12, 8))
    for i, idx in enumerate(idxs):
        plt.subplot(3, 3, i + 1)
```

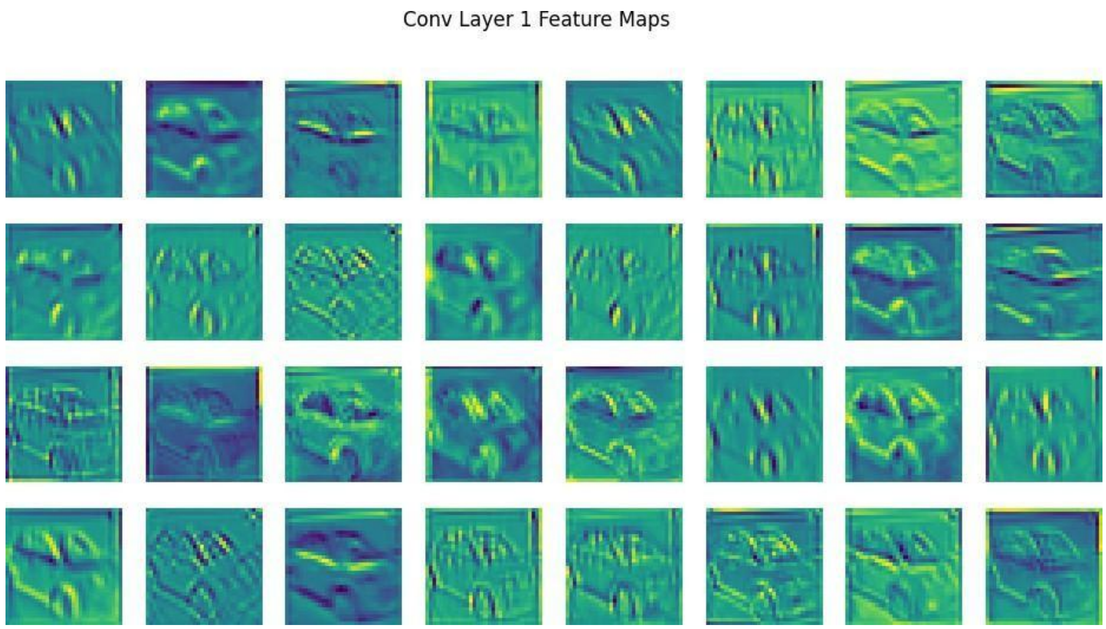
```
        plt.imshow(x[idx])
        plt.title(f"True:  {class_names[y_true[idx]]}\nPred:␣
↪{class_names[y_pred[idx]]}")
        plt.axis('off')
    plt.tight_layout()
    plt.show()

plot_samples(x_test, y_test, y_pred, class_names, n=9)
```

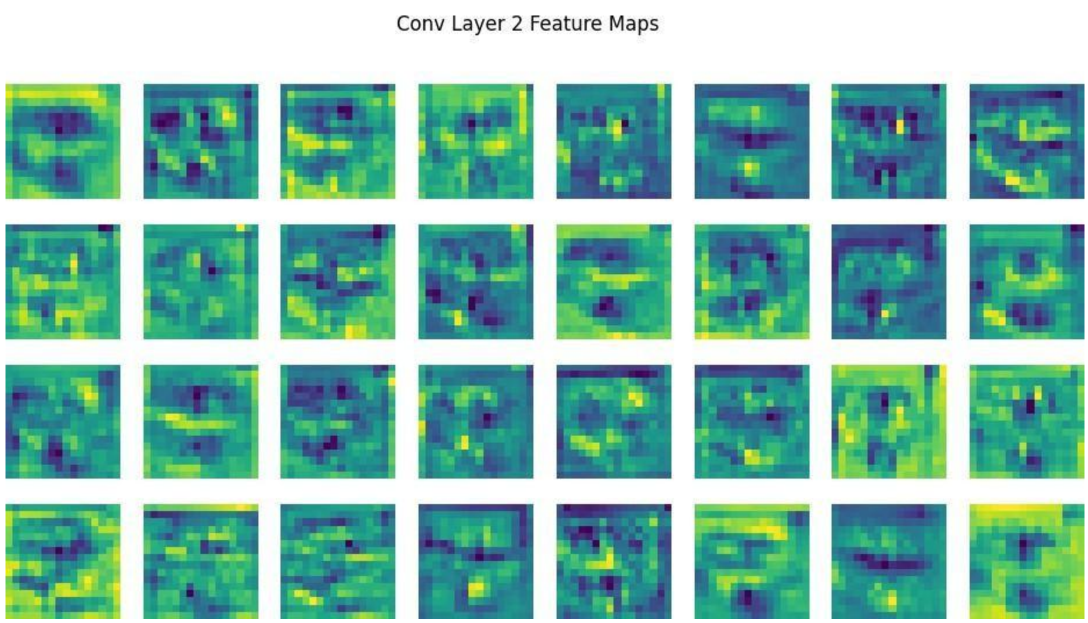


[14]:
```
activation_model = Model(inputs=model.input, outputs=[conv1, conv2])

img  =  x_test[np.random.randint(len(x_test))]
activations = activation_model.predict(np.expand_dims(img, axis=0))

for layer_idx, layer_act in enumerate(activations):
    num_maps = min(32, layer_act.shape[-1])
    plt.figure(figsize=(12, 6))
    for i in range(num_maps):
```

```
        plt.subplot(4, 8, i+1)
        plt.imshow(layer_act[0, :, :, i], cmap='viridis')
        plt.axis('off')
    plt.suptitle(f'Conv Layer {layer_idx+1} Feature Maps')
    plt.show()
```

1/1 **0s** 119ms/step





Practical 7

Aim: Implementation and analysis of Deep Neural network algorithm: Recurrent neural network (RNN) –

Character Recognition and

[1]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense, Flatten
from tensorflow.keras.utils import to_categorical
```

[2]:
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

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

[3]:
```
X_train = X_train / 255.0
X_test = X_test / 255.0
```

[4]:
```
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

[5]:
```
model = Sequential([
    SimpleRNN(128, input_shape=(28, 28), activation='tanh'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)

[6]:
```
model.compile(optimizer='adam', loss='categorical_crossentropy',
 metrics=['accuracy'])
```

[7]:
```
history = model.fit(X_train, y_train, epochs=10, batch_size=128,
 validation_split=0.1)
```

```
Epoch 1/10
422/422 13s 26ms/step -
accuracy: 0.7436 - loss: 0.8016 - val_accuracy: 0.9467 - val_loss: 0.1782
Epoch 2/10
422/422 11s 27ms/step -
accuracy: 0.9380 - loss: 0.2079 - val_accuracy: 0.9330 - val_loss: 0.2202
Epoch 3/10
422/422 11s 25ms/step -
accuracy: 0.9520 - loss: 0.1589 - val_accuracy: 0.9687 - val_loss: 0.1074
Epoch 4/10
422/422 11s 25ms/step -
accuracy: 0.9618 - loss: 0.1291 - val_accuracy: 0.9725 - val_loss: 0.0964
Epoch 5/10
422/422 11s 26ms/step -
accuracy: 0.9683 - loss: 0.1092 - val_accuracy: 0.9732 - val_loss: 0.0922
Epoch 6/10
422/422 10s 23ms/step -
accuracy: 0.9690 - loss: 0.1028 - val_accuracy: 0.9697 - val_loss: 0.1078
Epoch 7/10
422/422 11s 25ms/step -
accuracy: 0.9701 - loss: 0.0963 - val_accuracy: 0.9657 - val_loss: 0.1154
Epoch 8/10
422/422 11s 26ms/step -
accuracy: 0.9733 - loss: 0.0865 - val_accuracy: 0.9748 - val_loss: 0.0846
Epoch 9/10
422/422 11s 25ms/step -
accuracy: 0.9739 - loss: 0.0853 - val_accuracy: 0.9777 - val_loss: 0.0789
Epoch 10/10
422/422 11s 26ms/step -
accuracy: 0.9777 - loss: 0.0759 - val_accuracy: 0.9773 - val_loss: 0.0822
```

[8]:

```
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"\n Test Accuracy: {test_acc*100:.2f}%")
```

```
313/313 1s 4ms/step -
accuracy: 0.9619 - loss: 0.1286

 Test Accuracy: 96.82%
```

Web Traffic Image classification.

[1]:
```
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense, Dropout
from tensorflow.keras.utils import to_categorical
```

[2]:
```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

[3]:
```
X_train = X_train / 255.0
X_test = X_test / 255.0
```

[4]:
```
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

[5]:
```
X_train = X_train.reshape(-1, 32, 96)
X_test = X_test.reshape(-1, 32, 96)
```

[6]:
```
model = Sequential([
    SimpleRNN(128, input_shape=(32, 96), activation='tanh',
 return_sequences=False),
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(**kwargs)
```

[7]:
```
model.compile(optimizer='adam', loss='categorical_crossentropy',
 metrics=['accuracy'])
```

[8]:
```
history = model.fit(X_train, y_train, epochs=5, batch_size=128,
 validation_split=0.1)
```

```
Epoch 1/5
352/352            16s 39ms/step -
```

```
accuracy: 0.2068 - loss: 2.1332 - val_accuracy: 0.3184 - val_loss: 1.8595
Epoch 2/5
352/352 13s 36ms/step -
accuracy: 0.3122 - loss: 1.8695 - val_accuracy: 0.3324 - val_loss: 1.8001
Epoch 3/5
352/352 13s 37ms/step -
accuracy: 0.3247 - loss: 1.8237 - val_accuracy: 0.3100 - val_loss: 1.8755
Epoch 4/5
352/352 13s 37ms/step -
accuracy: 0.3397 - loss: 1.7957 - val_accuracy: 0.3816 - val_loss: 1.6776
Epoch 5/5
352/352 13s 37ms/step -
accuracy: 0.3706 - loss: 1.7262 - val_accuracy: 0.3734 - val_loss: 1.7109
```

[9]:
```
loss, acc = model.evaluate(X_test, y_test)
print(f"\n Test Accuracy: {acc*100:.2f}%")
```

```
313/313 2s 8ms/step -
accuracy: 0.3751 - loss: 1.6981

 Test Accuracy: 37.44%
```

Practical 8

Aim: LSTM Network: Sentiment analysis using LSTM

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

[2]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
```

[3]:
```
text_data = """
Artificial intelligence is transforming technology and society
Machine learning helps computers learn from data
Deep learning uses neural networks for better predictions
Recurrent neural networks understand sequences over time
AI models can generate text and recognize speech
Neural networks are inspired by the human brain
LSTM models remember information over long sequences
Data is the key ingredient for training AI systems
Predictive models improve accuracy with more data
AI helps automate complex decision making tasks
"""
```

[4]:
```
tokenizer = Tokenizer()
tokenizer.fit_on_texts([text_data])
total_words = len(tokenizer.word_index) + 1
```

[5]:
```
print("Total unique words:", total_words)
print("Word index:", tokenizer.word_index)
```

Total unique words: 58
Word index: {'data': 1, 'neural': 2, 'networks': 3, 'ai': 4, 'models': 5, 'is': 6, 'and': 7, 'learning': 8, 'helps': 9, 'for': 10, 'sequences': 11, 'over': 12, 'the': 13, 'artificial': 14, 'intelligence': 15, 'transforming': 16, 'technology': 17, 'society': 18, 'machine': 19, 'computers': 20, 'learn': 21,

'from': 22, 'deep': 23, 'uses': 24, 'better': 25, 'predictions': 26, 'recurrent': 27, 'understand': 28, 'time': 29, 'can': 30, 'generate': 31, 'text': 32, 'recognize': 33, 'speech': 34, 'are': 35, 'inspired': 36, 'by': 37, 'human': 38, 'brain': 39, 'lstm': 40, 'remember': 41, 'information': 42, 'long': 43, 'key': 44, 'ingredient': 45, 'training': 46, 'systems': 47, 'predictive': 48, 'improve': 49, 'accuracy': 50, 'with': 51, 'more': 52, 'automate': 53, 'complex': 54, 'decision': 55, 'making': 56, 'tasks': 57}

[6]:
```
input_sequences = []
for line in text_data.split("\n"):
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i + 1]
        input_sequences.append(n_gram_sequence)
```

[7]:
```
print("\nSample Input Sequences:")
print(input_sequences[:5])
```

Sample Input Sequences:
[[14, 15], [14, 15, 6], [14, 15, 6, 16], [14, 15, 6, 16, 17], [14, 15, 6, 16, 17, 7]]

[8]:
```
max_sequence_len = max([len(x) for x in input_sequences])
input_sequences = np.array(pad_sequences(input_sequences,
 maxlen=max_sequence_len, padding='pre'))
```

[9]:
```
X = input_sequences[:, :-1]
y = input_sequences[:, -1]
y = tf.keras.utils.to_categorical(y, num_classes=total_words)
```

[10]:
```
print("\nShape of X:", X.shape)
print("Shape of y:", y.shape)
```

Shape of X: (65, 8)
Shape of y: (65, 58)

[11]:
```
model = Sequential([
    Embedding(total_words, 128, input_length=max_sequence_len-1),
    LSTM(256),
    Dense(total_words, activation='softmax')
])
```

/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(

[12]:
```
model.compile(loss='categorical_crossentropy',     optimizer='adam',␣
↪metrics=['accuracy'])
model.fit(X, y, epochs=200, verbose=1)
```

```
Epoch 1/200
3/3                 2s 37ms/step -
accuracy: 0.0116 - loss: 4.0628
Epoch 2/200
3/3                 0s 31ms/step -
accuracy: 0.0736 - loss: 4.0418
Epoch 3/200
3/3                 0s 37ms/step -
accuracy: 0.0194 - loss: 4.0299
Epoch 4/200
3/3                 0s 36ms/step -
accuracy: 0.0116 - loss: 4.0183
Epoch 5/200
3/3                 0s 36ms/step -
accuracy: 0.0620 - loss: 3.9999
Epoch 6/200
3/3                 0s 35ms/step -
accuracy: 0.0930 - loss: 3.9797
Epoch 7/200
3/3                 0s 32ms/step -
accuracy: 0.1124 - loss: 3.9416
Epoch 8/200
3/3                 0s 46ms/step -
accuracy: 0.0736 - loss: 3.8750
Epoch 9/200
3/3                 0s 33ms/step -
accuracy: 0.0426 - loss: 3.8082
Epoch 10/200
3/3                 0s 33ms/step -
accuracy: 0.0426 - loss: 3.8607
Epoch 11/200
3/3                 0s 36ms/step -
accuracy: 0.0504 - loss: 3.8230
Epoch 12/200
3/3                 0s 32ms/step -
accuracy: 0.0891 - loss: 3.7143
Epoch 13/200
3/3                 0s 33ms/step -
accuracy: 0.1008 - loss: 3.7292
Epoch 14/200
3/3                 0s 32ms/step -
accuracy: 0.0852 - loss: 3.7278
Epoch 15/200
```

```
3/3                 0s 33ms/step -
accuracy: 0.1006 - loss: 3.7292
Epoch 16/200
3/3                 0s 33ms/step -
accuracy: 0.1591 - loss: 3.6830
Epoch 17/200
3/3                 0s 35ms/step -
accuracy: 0.1475 - loss: 3.6947
Epoch 18/200
3/3                 0s 32ms/step -
accuracy: 0.0852 - loss: 3.7116
Epoch 19/200
3/3                 0s 33ms/step -
accuracy: 0.0968 - loss: 3.6090
Epoch 20/200
3/3                 0s 33ms/step -
accuracy: 0.1591 - loss: 3.5762
Epoch 21/200
3/3                 0s 37ms/step -
accuracy: 0.2017 - loss: 3.5595
Epoch 22/200
3/3                 0s 44ms/step -
accuracy: 0.1704 - loss: 3.5170
Epoch 23/200
3/3                 0s 39ms/step -
accuracy: 0.2014 - loss: 3.5162
Epoch 24/200
3/3                 0s 33ms/step -
accuracy: 0.2365 - loss: 3.4730
Epoch 25/200
3/3                 0s 41ms/step -
accuracy: 0.2365 - loss: 3.4327
Epoch 26/200
3/3                 0s 31ms/step -
accuracy: 0.2209 - loss: 3.3652
Epoch 27/200
3/3                 0s 33ms/step -
accuracy: 0.2559 - loss: 3.3422
Epoch 28/200
3/3                 0s 32ms/step -
accuracy: 0.2480 - loss: 3.2948
Epoch 29/200
3/3                 0s 32ms/step -
accuracy: 0.2559 - loss: 3.2209
Epoch 30/200
3/3                 0s 45ms/step -
accuracy: 0.2366 - loss: 3.1942
Epoch 31/200
```

```
3/3              0s 33ms/step -
accuracy: 0.1897 - loss: 3.1375
Epoch 32/200
3/3              0s 32ms/step -
accuracy: 0.2793 - loss: 2.9803
Epoch 33/200
3/3              0s 33ms/step -
accuracy: 0.2521 - loss: 2.8295
Epoch 34/200
3/3              0s 32ms/step -
accuracy: 0.2984 - loss: 2.7590
Epoch 35/200
3/3              0s 32ms/step -
accuracy: 0.2985 - loss: 2.6913
Epoch 36/200
3/3              0s 32ms/step -
accuracy: 0.2481 - loss: 2.6297
Epoch 37/200
3/3              0s 32ms/step -
accuracy: 0.2365 - loss: 2.5706
Epoch 38/200
3/3              0s 40ms/step -
accuracy: 0.2130 - loss: 2.5127
Epoch 39/200
3/3              0s 33ms/step -
accuracy: 0.3100 - loss: 2.3960
Epoch 40/200
3/3              0s 36ms/step -
accuracy: 0.3254 - loss: 2.2925
Epoch 41/200
3/3              0s 32ms/step -
accuracy: 0.3139 - loss: 2.2310
Epoch 42/200
3/3              0s 32ms/step -
accuracy: 0.3489 - loss: 2.1218
Epoch 43/200
3/3              0s 35ms/step -
accuracy: 0.3721 - loss: 2.0452
Epoch 44/200
3/3              0s 36ms/step -
accuracy: 0.3915 - loss: 2.0578
Epoch 45/200
3/3              0s 35ms/step -
accuracy: 0.4651 - loss: 1.9886
Epoch 46/200
3/3              0s 49ms/step -
accuracy: 0.4770 - loss: 1.9006
Epoch 47/200
```

```
3/3 0s 36ms/step -
accuracy: 0.4225 - loss: 1.8876
Epoch 48/200
3/3 0s 38ms/step -
accuracy: 0.4534 - loss: 1.8273
Epoch 49/200
3/3 0s 32ms/step -
accuracy: 0.4417 - loss: 1.8036
Epoch 50/200
3/3 0s 33ms/step -
accuracy: 0.4689 - loss: 1.7523
Epoch 51/200
3/3 0s 37ms/step -
accuracy: 0.5659 - loss: 1.7113
Epoch 52/200
3/3 0s 39ms/step -
accuracy: 0.5737 - loss: 1.5949
Epoch 53/200
3/3 0s 36ms/step -
accuracy: 0.5579 - loss: 1.5945
Epoch 54/200
3/3 0s 44ms/step -
accuracy: 0.5775 - loss: 1.6152
Epoch 55/200
3/3 0s 70ms/step -
accuracy: 0.5815 - loss: 1.4951
Epoch 56/200
3/3 0s 64ms/step -
accuracy: 0.6009 - loss: 1.4189
Epoch 57/200
3/3 0s 68ms/step -
accuracy: 0.6241 - loss: 1.3993
Epoch 58/200
3/3 0s 86ms/step -
accuracy: 0.6358 - loss: 1.3428
Epoch 59/200
3/3 0s 64ms/step -
accuracy: 0.6241 - loss: 1.3444
Epoch 60/200
3/3 0s 64ms/step -
accuracy: 0.5853 - loss: 1.3450
Epoch 61/200
3/3 0s 59ms/step -
accuracy: 0.5931 - loss: 1.2968
Epoch 62/200
3/3 0s 69ms/step -
accuracy: 0.5737 - loss: 1.3442
Epoch 63/200
```

```
3/3 0s 61ms/step -
accuracy: 0.5775 - loss: 1.4262
Epoch 64/200
3/3 0s 62ms/step -
accuracy: 0.5855 - loss: 1.3153
Epoch 65/200
3/3 0s 62ms/step -
accuracy: 0.5504 - loss: 1.3370
Epoch 66/200
3/3 0s 63ms/step -
accuracy: 0.6007 - loss: 1.2974
Epoch 67/200
3/3 0s 66ms/step -
accuracy: 0.5739 - loss: 1.2298
Epoch 68/200
3/3 0s 61ms/step -
accuracy: 0.5427 - loss: 1.3117
Epoch 69/200
3/3 0s 61ms/step -
accuracy: 0.5347 - loss: 1.2952
Epoch 70/200
3/3 0s 59ms/step -
accuracy: 0.6319 - loss: 1.1780
Epoch 71/200
3/3 0s 75ms/step -
accuracy: 0.7171 - loss: 1.0859
Epoch 72/200
3/3 0s 65ms/step -
accuracy: 0.6395 - loss: 1.0468
Epoch 73/200
3/3 0s 72ms/step -
accuracy: 0.6395 - loss: 1.0515
Epoch 74/200
3/3 0s 59ms/step -
accuracy: 0.6511 - loss: 1.0320
Epoch 75/200
3/3 0s 70ms/step -
accuracy: 0.7286 - loss: 1.0031
Epoch 76/200
3/3 0s 62ms/step -
accuracy: 0.7870 - loss: 0.8976
Epoch 77/200
3/3 0s 39ms/step -
accuracy: 0.7791 - loss: 0.8862
Epoch 78/200
3/3 0s 31ms/step -
accuracy: 0.7597 - loss: 0.8595
Epoch 79/200
```

```
3/3 0s 38ms/step -
accuracy: 0.7870 - loss: 0.8230
Epoch 80/200
3/3 0s 46ms/step -
accuracy: 0.7093 - loss: 0.8229
Epoch 81/200
3/3 0s 38ms/step -
accuracy: 0.7169 - loss: 0.8533
Epoch 82/200
3/3 0s 32ms/step -
accuracy: 0.6782 - loss: 0.8288
Epoch 83/200
3/3 0s 37ms/step -
accuracy: 0.7635 - loss: 0.7799
Epoch 84/200
3/3 0s 33ms/step -
accuracy: 0.7288 - loss: 0.8611
Epoch 85/200
3/3 0s 37ms/step -
accuracy: 0.7870 - loss: 0.8996
Epoch 86/200
3/3 0s 33ms/step -
accuracy: 0.7404 - loss: 0.8805
Epoch 87/200
3/3 0s 36ms/step -
accuracy: 0.7751 - loss: 0.9607
Epoch 88/200
3/3 0s 36ms/step -
accuracy: 0.8528 - loss: 0.9772
Epoch 89/200
3/3 0s 32ms/step -
accuracy: 0.7751 - loss: 1.0330
Epoch 90/200
3/3 0s 33ms/step -
accuracy: 0.7947 - loss: 0.9918
Epoch 91/200
3/3 0s 38ms/step -
accuracy: 0.8334 - loss: 0.9227
Epoch 92/200
3/3 0s 33ms/step -
accuracy: 0.8215 - loss: 0.8858
Epoch 93/200
3/3 0s 37ms/step -
accuracy: 0.7905 - loss: 0.8882
Epoch 94/200
3/3 0s 32ms/step -
accuracy: 0.7909 - loss: 0.7794
Epoch 95/200
```

```
3/3                    0s 40ms/step -
accuracy: 0.8528 - loss: 0.7324
Epoch 96/200
3/3                    0s 37ms/step -
accuracy: 0.7948 - loss: 0.7644
Epoch 97/200
3/3                    0s 47ms/step -
accuracy: 0.7945 - loss: 0.8327
Epoch 98/200
3/3                    0s 35ms/step -
accuracy: 0.7442 - loss: 0.8114
Epoch 99/200
3/3                    0s 38ms/step -
accuracy: 0.7325 - loss: 0.7729
Epoch 100/200
3/3                    0s 35ms/step -
accuracy: 0.7870 - loss: 0.6796
Epoch 101/200
3/3                    0s 38ms/step -
accuracy: 0.8800 - loss: 0.6443
Epoch 102/200
3/3                    0s 34ms/step -
accuracy: 0.8681 - loss: 0.6349
Epoch 103/200
3/3                    0s 39ms/step -
accuracy: 0.8681 - loss: 0.6181
Epoch 104/200
3/3                    0s 36ms/step -
accuracy: 0.8916 - loss: 0.5291
Epoch 105/200
3/3                    0s 33ms/step -
accuracy: 0.8954 - loss: 0.5705
Epoch 106/200
3/3                    0s 36ms/step -
accuracy: 0.8838 - loss: 0.5255
Epoch 107/200
3/3                    0s 39ms/step -
accuracy: 0.8993 - loss: 0.5008
Epoch 108/200
3/3                    0s 34ms/step -
accuracy: 0.9070 - loss: 0.5035
Epoch 109/200
3/3                    0s 32ms/step -
accuracy: 0.8838 - loss: 0.5216
Epoch 110/200
3/3                    0s 33ms/step -
accuracy: 0.8723 - loss: 0.5304
Epoch 111/200
```

3/3 **0s** 38ms/step –
accuracy: 0.8412 – loss: 0.5262
Epoch 112/200
3/3 **0s** 39ms/step –
accuracy: 0.8487 – loss: 0.5404
Epoch 113/200
3/3 **0s** 41ms/step –
accuracy: 0.8450 – loss: 0.5609
Epoch 114/200
3/3 **0s** 37ms/step –
accuracy: 0.8293 – loss: 0.5881
Epoch 115/200
3/3 **0s** 36ms/step –
accuracy: 0.8293 – loss: 0.5888
Epoch 116/200
3/3 **0s** 32ms/step –
accuracy: 0.8334 – loss: 0.5294
Epoch 117/200
3/3 **0s** 45ms/step –
accuracy: 0.8722 – loss: 0.4867
Epoch 118/200
3/3 **0s** 39ms/step –
accuracy: 0.8681 – loss: 0.4684
Epoch 119/200
3/3 **0s** 38ms/step –
accuracy: 0.8876 – loss: 0.4470
Epoch 120/200
3/3 **0s** 46ms/step –
accuracy: 0.9226 – loss: 0.3965
Epoch 121/200
3/3 **0s** 45ms/step –
accuracy: 0.8954 – loss: 0.4398
Epoch 122/200
3/3 **0s** 37ms/step –
accuracy: 0.9264 – loss: 0.4132
Epoch 123/200
3/3 **0s** 32ms/step –
accuracy: 0.9381 – loss: 0.3854
Epoch 124/200
3/3 **0s** 32ms/step –
accuracy: 0.9768 – loss: 0.3938
Epoch 125/200
3/3 **0s** 38ms/step –
accuracy: 0.8606 – loss: 0.5248
Epoch 126/200
3/3 **0s** 42ms/step –
accuracy: 0.8876 – loss: 0.5240
Epoch 127/200

```
3/3                    0s 33ms/step -
accuracy: 0.9496 - loss: 0.4955
Epoch 128/200
3/3                    0s 36ms/step -
accuracy: 0.9264 - loss: 0.5027
Epoch 129/200
3/3                    0s 34ms/step -
accuracy: 0.8760 - loss: 0.6240
Epoch 130/200
3/3                    0s 37ms/step -
accuracy: 0.8916 - loss: 0.5464
Epoch 131/200
3/3                    0s 37ms/step -
accuracy: 0.8681 - loss: 0.5582
Epoch 132/200
3/3                    0s 37ms/step -
accuracy: 0.8876 - loss: 0.5113
Epoch 133/200
3/3                    0s 33ms/step -
accuracy: 0.8876 - loss: 0.5171
Epoch 134/200
3/3                    0s 48ms/step -
accuracy: 0.9226 - loss: 0.4116
Epoch 135/200
3/3                    0s 37ms/step -
accuracy: 0.9186 - loss: 0.4562
Epoch 136/200
3/3                    0s 37ms/step -
accuracy: 0.9070 - loss: 0.3907
Epoch 137/200
3/3                    0s 38ms/step -
accuracy: 0.9458 - loss: 0.3251
Epoch 138/200
3/3                    0s 38ms/step -
accuracy: 0.9380 - loss: 0.3393
Epoch 139/200
3/3                    0s 34ms/step -
accuracy: 0.9806 - loss: 0.3332
Epoch 140/200
3/3                    0s 39ms/step -
accuracy: 0.9884 - loss: 0.3162
Epoch 141/200
3/3                    0s 36ms/step -
accuracy: 0.9768 - loss: 0.2896
Epoch 142/200
3/3                    0s 47ms/step -
accuracy: 0.9574 - loss: 0.2971
Epoch 143/200
```

```
3/3                    0s 37ms/step -
accuracy: 0.9884 - loss: 0.2794
Epoch 144/200
3/3                    0s 33ms/step -
accuracy: 0.9806 - loss: 0.2834
Epoch 145/200
3/3                    0s 36ms/step -
accuracy: 0.9806 - loss: 0.2600
Epoch 146/200
3/3                    0s 38ms/step -
accuracy: 0.9806 - loss: 0.2461
Epoch 147/200
3/3                    0s 33ms/step -
accuracy: 0.9806 - loss: 0.2360
Epoch 148/200
3/3                    0s 39ms/step -
accuracy: 0.9768 - loss: 0.2489
Epoch 149/200
3/3                    0s 79ms/step -
accuracy: 0.9768 - loss: 0.2493
Epoch 150/200
3/3                    0s 75ms/step -
accuracy: 0.9612 - loss: 0.2639
Epoch 151/200
3/3                    0s 57ms/step -
accuracy: 0.9690 - loss: 0.2611
Epoch 152/200
3/3                    0s 63ms/step -
accuracy: 0.9690 - loss: 0.2778
Epoch 153/200
3/3                    0s 58ms/step -
accuracy: 0.9806 - loss: 0.2546
Epoch 154/200
3/3                    0s 61ms/step -
accuracy: 0.9884 - loss: 0.2333
Epoch 155/200
3/3                    0s 69ms/step -
accuracy: 0.9690 - loss: 0.2411
Epoch 156/200
3/3                    0s 60ms/step -
accuracy: 0.9574 - loss: 0.2405
Epoch 157/200
3/3                    0s 66ms/step -
accuracy: 0.9652 - loss: 0.2181
Epoch 158/200
3/3                    0s 67ms/step -
accuracy: 0.9380 - loss: 0.2382
Epoch 159/200
```

3/3 **0s** 63ms/step –
accuracy: 0.9574 – loss: 0.2088
Epoch 160/200
3/3 **0s** 65ms/step –
accuracy: 0.9612 – loss: 0.2087
Epoch 161/200
3/3 **0s** 67ms/step –
accuracy: 0.9884 – loss: 0.1962
Epoch 162/200
3/3 **0s** 63ms/step –
accuracy: 0.9806 – loss: 0.1943
Epoch 163/200
3/3 **0s** 62ms/step –
accuracy: 0.9884 – loss: 0.1973
Epoch 164/200
3/3 **0s** 64ms/step –
accuracy: 0.9884 – loss: 0.1765
Epoch 165/200
3/3 **0s** 66ms/step –
accuracy: 0.9884 – loss: 0.1763
Epoch 166/200
3/3 **0s** 64ms/step –
accuracy: 0.9806 – loss: 0.1731
Epoch 167/200
3/3 **0s** 73ms/step –
accuracy: 0.9806 – loss: 0.1740
Epoch 168/200
3/3 **0s** 72ms/step –
accuracy: 0.9884 – loss: 0.1602
Epoch 169/200
3/3 **0s** 48ms/step –
accuracy: 0.9806 – loss: 0.1716
Epoch 170/200
3/3 **0s** 40ms/step –
accuracy: 0.9612 – loss: 0.2487
Epoch 171/200
3/3 **0s** 36ms/step –
accuracy: 0.9613 – loss: 0.2746
Epoch 172/200
3/3 **0s** 36ms/step –
accuracy: 0.9574 – loss: 0.2645
Epoch 173/200
3/3 **0s** 37ms/step –
accuracy: 0.9690 – loss: 0.2333
Epoch 174/200
3/3 **0s** 36ms/step –
accuracy: 0.9690 – loss: 0.2134
Epoch 175/200

```
3/3                 0s 40ms/step -
accuracy: 0.9612 - loss: 0.2255
Epoch 176/200
3/3                 0s 37ms/step -
accuracy: 0.9652 - loss: 0.2212
Epoch 177/200
3/3                 0s 37ms/step -
accuracy: 0.9302 - loss: 0.2569
Epoch 178/200
3/3                 0s 36ms/step -
accuracy: 0.9418 - loss: 0.2386
Epoch 179/200
3/3                 0s 38ms/step -
accuracy: 0.9574 - loss: 0.1874
Epoch 180/200
3/3                 0s 44ms/step -
accuracy: 0.9380 - loss: 0.2211
Epoch 181/200
3/3                 0s 45ms/step -
accuracy: 0.9458 - loss: 0.1988
Epoch 182/200
3/3                 0s 37ms/step -
accuracy: 0.9496 - loss: 0.1872
Epoch 183/200
3/3                 0s 46ms/step -
accuracy: 0.9690 - loss: 0.1838
Epoch 184/200
3/3                 0s 37ms/step -
accuracy: 0.9418 - loss: 0.2024
Epoch 185/200
3/3                 0s 33ms/step -
accuracy: 0.9574 - loss: 0.1957
Epoch 186/200
3/3                 0s 37ms/step -
accuracy: 0.9496 - loss: 0.2034
Epoch 187/200
3/3                 0s 45ms/step -
accuracy: 0.9496 - loss: 0.1823
Epoch 188/200
3/3                 0s 33ms/step -
accuracy: 0.9496 - loss: 0.1744
Epoch 189/200
3/3                 0s 38ms/step -
accuracy: 0.9690 - loss: 0.1480
Epoch 190/200
3/3                 0s 33ms/step -
accuracy: 0.9806 - loss: 0.1529
Epoch 191/200
```

```
3/3                    0s 38ms/step -
accuracy: 0.9806 - loss: 0.1723
Epoch 192/200
3/3                    0s 38ms/step -
accuracy: 0.9884 - loss: 0.1643
Epoch 193/200
3/3                    0s 38ms/step -
accuracy: 0.9884 - loss: 0.1648
Epoch 194/200
3/3                    0s 38ms/step -
accuracy: 0.9884 - loss: 0.1618
Epoch 195/200
3/3                    0s 37ms/step -
accuracy: 0.9806 - loss: 0.1781
Epoch 196/200
3/3                    0s 37ms/step -
accuracy: 0.9884 - loss: 0.1436
Epoch 197/200
3/3                    0s 41ms/step -
accuracy: 0.9806 - loss: 0.1522
Epoch 198/200
3/3                    0s 37ms/step -
accuracy: 0.9884 - loss: 0.1212
Epoch 199/200
3/3                    0s 37ms/step -
accuracy: 0.9806 - loss: 0.1406
Epoch 200/200
3/3                    0s 36ms/step -
accuracy: 0.9806 - loss: 0.1290
```

[12]: <keras.src.callbacks.history.History at 0x7cf104eee2a0>

[13]:
```
seed_text = "I will"
next_words = 10

for _ in range(next_words):
    token_list = tokenizer.texts_to_sequences([seed_text])[0]
    token_list = pad_sequences([token_list], maxlen=max_sequence_len - 1,␣
 ↪padding='pre')
    predicted = np.argmax(model.predict(token_list), axis=-1)[0]

    output_word = ""
    for word, index in tokenizer.word_index.items():
        if index == predicted:
            output_word = word
            break
    seed_text += " " + output_word
```

```
print("\nGenerated Text:")
print(seed_text)
```

```
1/1          0s 176ms/step
1/1          0s 35ms/step
1/1          0s 33ms/step
1/1          0s 33ms/step
1/1          0s 36ms/step
1/1          0s 33ms/step
1/1          0s 34ms/step
1/1          0s 34ms/step
1/1          0s 45ms/step
1/1          0s 32ms/step

Generated Text:
I will learning helps computers learn from data data systems systems systems
```