EXPERIMENT-4

Aim:

Write a program to implement Artificial Neural Network.

Platform Used: Google Colab.

Theory:

The term "Artificial Neural Network" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.

Input Layer:

As the name suggests, it accepts inputs in several different formats provided by the programmer.

Hidden Layer:

The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

Output Layer:

The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

Input:

```
import numpy as np

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):

    return x * (1 - x)

input_layer_size = 2

hidden_layer_size = 1

output_layer_size = 1

np.random.seed(0)

weights_input_hidden = np.random.randn(input_layer_size, hidden_layer_size)

biases_hidden = np.random.randn(hidden_layer_size)

weights_hidden_output = np.random.randn(hidden_layer_size, output_layer_size)

biases_output = np.random.randn(output_layer_size)

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

```
y = np.array([[0], [1], [1], [0]])
epochs = 10000
learning_rate = 0.1
for epoch in range(epochs):
    # Forward pass
    hidden_layer_activation = np.dot(X, weights_input_hidden) + biases_hidden
    hidden_layer_output = sigmoid(hidden_layer_activation)
    output_layer_activation = np.dot(hidden_layer_output, weights_hidden_output) + biases_output
    output = sigmoid(output_layer_activation)
    # Backward pass
    error = y - output
    output_gradient = sigmoid_derivative(output)
    output_error = error * output_gradient
    hidden_layer_error = output_error.dot(weights_hidden_output.T)
    hidden_layer_gradient = sigmoid_derivative(hidden_layer_output)
    hidden_layer_error_term = hidden_layer_error * hidden_layer_gradient
    # Update weights and biases
    weights_hidden_output += hidden_layer_output.T.dot(output_error) * learning_rate
    biases_output += np.sum(output_error) * learning_rate
    weights_input_hidden += X.T.dot(hidden_layer_error_term) * learning_rate
    biases_hidden += np.sum(hidden_layer_error_term) * learning_rate
# Print the final output
print("Final output after training:")
print(output)
```

Output:

```
Final output after training:
[[0.05990788]
 [0.66293873]
 [0.66295926]
 [0.6675519 ]]
```

EXPERIMENT-5

Aim:

Write a program to implement the CNN.

Platform Used: Google Colab.

Theory:

CNN stands for Convolutional Neural Network, which is a type of deep neural network commonly used for analyzing visual imagery. CNNs are particularly well-suited for tasks like image recognition and classification.

Key components of a CNN include:

1. **Convolutional layers**: These layers apply convolution operations to the input, using filters (also known as kernels) to extract features from the input data.
2. **Pooling layers**: Pooling layers reduce the spatial dimensions of the input data by down-sampling, which helps in reducing computation and controlling overfitting.
3. **Activation functions**: Typically, ReLU (Rectified Linear Unit) is used as the activation function in CNNs to introduce non-linearity into the network.
4. **Fully connected layers**: These layers connect every neuron in one layer to every neuron in the next layer, similar to a traditional neural network.
5. **Output layer**: The final layer of the CNN produces the output, which could be probabilities for different classes in a classification task or continuous values in a regression task.

CNNs have been highly successful in various applications, including image recognition, object detection, image segmentation, and more. Their ability to automatically learn features from raw data, along with their hierarchical structure, makes them powerful tools for tasks involving visual data.

Input:

```
import numpy as np

class CNN:

  def __init__(self):

    pass

  def convLayer(self, input_shape, channels, strides, padding, filter_size):

    pass

  def maxPooling(self, input_matrix):

    pass

  def flatten(self, input_matrix):

    pass

  def dropout(self, input_matrix, dropout_rate = 0):

    pass
```

```
def convLayer(self, input_shape, channels, strides, padding, filter_size):
    height, width = input_shape
    input_shape_with_channels = (height, width, channels)
    print("Input Shape (with channels):", input_shape_with_channels)
    # for random input and filter matrix
    input_matrix = np.random.randint(0, 10, size=input_shape_with_channels)
    filter_matrix = np.random.randint(0, 5, size=filter_size)
    input_matrix = np.array([
            [1, 1, 1, 0, 0],
            [0, 1, 1, 1, 0],
            [0, 0, 1, 1, 1],
            [0, 0, 1, 1, 0],
            [0, 1, 1, 0, 0]
    ])
    filter_matrix = np.array([
            [1, 0, 1],
            [0, 1, 0],
            [1, 0, 1]
    ])
    print("\nInput Matrix:")
    print(input_matrix)
    print("\nFilter Matrix:")
    print(filter_matrix)
    padding.lower()
    padSize = 0
    if padding == 'same':
        # Calculate padding needed for each dimension
        pad_height = ((height - 1) * strides[0] + filter_size[0] - height) // 2
        pad_width = ((width - 1) * strides[1] + filter_size[1] - width) // 2
        # Apply padding to the input matrix
        input_matrix = np.pad(input_matrix, ((pad_height, pad_height), (pad_width, pad_width),
                                    (0, 0)), mode='constant')
```

```
            # Adjust height and width to consider the padding
            height += 2 * pad_height
            width += 2 * pad_width
        elif padding == 'valid':
            padSize = filter_size[0] // 2
            print("\nPad Size: ", padSize)
        else:
            return "Invalid Padding!!"
        # output dimension
        conv_height = (height - filter_size[0]) // strides[0] + 1
        conv_width = (width - filter_size[1]) // strides[1] + 1
        output_matrix = np.zeros((conv_height, conv_width))
        # Convolution Operation
        for i in range(0, height - filter_size[0] + 1, strides[0]):
            for j in range(0, width - filter_size[1] + 1, strides[1]):
                receptive_field = input_matrix[i:i + filter_size[0], j:j + filter_size[1]]
                output_matrix[i // strides[0], j // strides[1]] = np.sum(receptive_field * filter_matrix)
        return output_matrix
    def maxPooling(self, input_matrix, pool_size, strides_pooling):
        pool_height, pool_width = pool_size
        stride_height, stride_width = strides_pooling
        pooled_height = (input_matrix.shape[0] - pool_height) // stride_height + 1
        pooled_width = (input_matrix.shape[1] - pool_width) // stride_width + 1
        pooled_matrix = np.zeros((pooled_height, pooled_width))
        for i in range(pooled_height):
            for j in range(pooled_width):
                patch = input_matrix[i * stride_height: i * stride_height + pool_height,
                                     j * stride_width: j * stride_width + pool_width]
                pooled_matrix[i, j] = np.max(patch)
        return pooled_matrix
    def flatten(self, input_matrix):
```

```
        return input_matrix.flatten()
    def dropout(self, input_matrix, dropout_rate = 0):
        dropout_mask = np.random.binomial(1, 1 - dropout_rate, size=input_matrix.shape)
        return input_matrix * dropout_mask
input_shape = (5, 5)
channels = 1
strides = (1, 1)
padding = 'valid'
filter_size = (3, 3)
cnn_model = CNN()
conv1 = cnn_model.convLayer(input_shape, channels, strides, padding, filter_size)
conv1
pool_size = (2, 2)
strides_pooling = (1, 1)
maxPool = cnn_model.maxPooling(conv1, pool_size, strides_pooling)
maxPool
flattened_output = cnn_model.flatten(maxPool)
flattened_output
dropout_output = cnn_model.dropout(flattened_output, 0.3)
dropout_output
```

Output:

```
Input Shape (with channels): (5, 5, 1)

Input Matrix:
[[1 1 1 0 0]
 [0 1 1 1 0]
 [0 0 1 1 1]
 [0 0 1 1 0]
 [0 1 1 0 0]]

Filter Matrix:
[[1 0 1]
 [0 1 0]
 [1 0 1]]

Pad Size:  1
```

```
array([[4., 4.],
       [4., 4.]])
```

```
array([[4., 3., 4.],
       [2., 4., 3.],
       [2., 3., 4.]])
```

```
array([4., 0., 4., 4.])
```

```
array([4., 4., 4., 4.])
```

Experiment -6

Aim:

Implement RNN network.

Theory:

Recurrent Neural Network (RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other. Still, in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its Hidden state, which remembers some information about a sequence. The state is also referred to as Memory State since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

Input:

```
[1] import tensorflow as tf
    import tensorflow_datasets as tfds
    import numpy as np
    import matplotlib.pyplot as plt
```

```
[2] # Obtain the imdb review dataset from tensorflow datasets
    dataset = tfds.load('imdb_reviews', as_supervised=True)
    # Seperate test and train datasets
    train_dataset, test_dataset = dataset['train'], dataset['test']
    # Split the test and train data into batches of 32
    # and shuffling the training set
    batch_size = 32
    train_dataset = train_dataset.shuffle(10000)
    train_dataset = train_dataset.batch(batch_size)
    test_dataset = test_dataset.batch(batch_size)
```

```
Downloading and preparing dataset 80.23 MiB (download: 80.23 MiB, generated: Unknown size, total: 80.23 MiB) to /root/tensorflow_datasets/imdb_reviews/plain_text/1.0.0...
Dl Completed...: 100% 1/1 [00:10<00:00, 16.27s/ url]
Dl Size...: 100% 80/80 [00:10<00:00, 5.40 MiB/s]
Dataset imdb_reviews downloaded and prepared to /root/tensorflow_datasets/imdb_reviews/plain_text/1.0.0. Subsequent calls will reuse this data.
```

```
[3] example, label = next(iter(train_dataset))
    print('Text:\n', example.numpy()[0])
    print('\nLabel: ', label.numpy()[0])
```

```
Text:
 b"This is a really well made movie. Sumitra Bhave has always made sensible cinema and this is my favourite film by her. This movie should have won the National Award and would have !

Label:  1
```

```
# Using the TextVectorization layer to normalize, split, and map strings
# to integers.
encoder = tf.keras.layers.TextVectorization(max_tokens=10000)
encoder.adapt(train_dataset.map(lambda text, _: text))
# Extracting the vocabulary from the TextVectorization layer.
vocabulary = np.array(encoder.get_vocabulary())
# Encoding a test example and decoding it back.
original_text = example.numpy()[0]
encoded_text = encoder(original_text).numpy()
decoded_text = ' '.join(vocabulary[encoded_text])
print('original: ', original_text)
print('encoded: ', encoded_text)
print('decoded: ', decoded_text)
```

```
original:  b"This is a really well made movie. Sumitra Bhave has always made sensible cinema and this is my favourite film by her. This movie should have won the National Award and w
encoded:  [  11    7    4   63   74   91   18    1    1   44  203   93 6750  443
    3   11    7   58 1577   20   33   40   11   18  139   26 1149    2
 1983 1398    3   59   26   75   56 1273    6 4186 2564   31    2 1968
    9    7   32  212    4 1286  211  125   71    1   60    7  166    6
    2 3568   30 2044   11 8542   53    0    7  136    4 1202   12    2
 1508   43   11    3   32   81 1345   85   21  951    7 1664    3  505
   54  359    1    3  251    4   53  665  100    8   11   18    3   27
    7   22   54 5342   31  128 4436    2  343    5    2  177    3 1031
  100   24   22 1042   31   32 2079    5 6365   16   11   18   24   22
  335   54   51    1    3 1149    2 1345 1983 1398   16   11   18   48
   14   22   54    4  687 2248  177   26    1   13    1    7   22    4
 6472   18    9    7    3    3 2664 2986    2  636    2  845  702  125
   22   76    1   58    8    3    1    3    1 5030    1    9    7    4
  146  119    1   64   12    7   81   27    4 1237 1836    3    1   13
    1   60   69 7024   28 5363   35  105  362    7    8   64    5  105
 2072    1    3    3    9    7  157    8   53  500   64    1    1  125
```

```
[5] # Creating the model
    model = tf.keras.Sequential([
      encoder,
      tf.keras.layers.Embedding(
        len(encoder.get_vocabulary()), 64, mask_zero=True),
      tf.keras.layers.Bidirectional(
        tf.keras.layers.LSTM(64, return_sequences=True)),
      tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(1)])
    # Summary of the model
    model.summary()
    # Compile the model
    model.compile(
      loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
      optimizer=tf.keras.optimizers.Adam(),
      metrics=['accuracy'])
```

```
# Training the model and validating it on test set
history = model.fit(
   train_dataset,
   epochs=5,
   validation_data=test_dataset,)
```

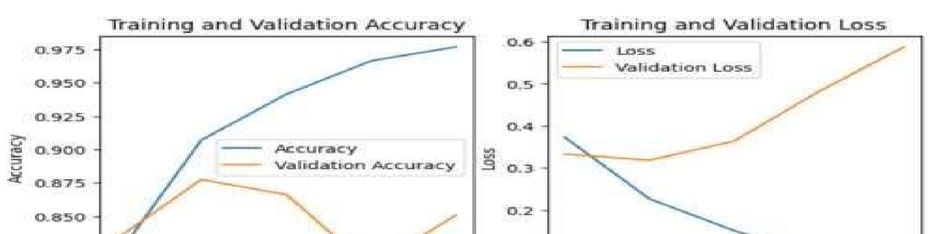
```
# Plotting the accuracy and loss over time
# Training history
history_dict = history.history
# Seperating validation and training accuracy
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
# Seperating validation and training loss
loss = history_dict['loss']
val_loss = history_dict['val_loss']
# Plotting
plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.plot(acc)
plt.plot(val_acc)
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['Accuracy', 'Validation Accuracy'])
plt.subplot(1, 2, 2)
plt.plot(loss)
plt.plot(val_loss)
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Loss', 'Validation Loss'])
plt.show()
```

```
[8] # Making predictions
    sample_text = (
      '''The movie by GeeksforGeeks was so good and the animation are so dope.
      I would recommend my friends to watch it.''')
    predictions = model.predict(np.array([sample_text]))
    print(*predictions[0])
    # Print the label based on the prediction
    if predictions[0] > 0:
      print('The review is positive')
    else:
      print('The review is negative')
```

Output:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 text_vectorization (TextVe  (None, None)              0
 ctorization)
 embedding (Embedding)       (None, None, 64)          640000
 bidirectional (Bidirection  (None, None, 128)         66048
 al)
 bidirectional_1 (Bidirecti  (None, 64)                41216
 onal)
 dense (Dense)               (None, 64)                4160
 dense_1 (Dense)             (None, 1)                 65
=================================================================
Total params: 751489 (2.87 MB)
Trainable params: 751489 (2.87 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
Epoch 1/5
782/782 [==============================] - 1885s 2s/step - loss: 0.3731 - accuracy: 0.8197 - val_loss: 0.3332 - val_accuracy: 0.8331
Epoch 2/5
782/782 [==============================] - 1877s 2s/step - loss: 0.2266 - accuracy: 0.9071 - val_loss: 0.3184 - val_accuracy: 0.8775
Epoch 3/5
782/782 [==============================] - 1844s 2s/step - loss: 0.1511 - accuracy: 0.9414 - val_loss: 0.3641 - val_accuracy: 0.8664
Epoch 4/5
782/782 [==============================] - 1836s 2s/step - loss: 0.0899 - accuracy: 0.9664 - val_loss: 0.4820 - val_accuracy: 0.8116
Epoch 5/5
782/782 [==============================] - 1884s 2s/step - loss: 0.0633 - accuracy: 0.9770 - val_loss: 0.5875 - val_accuracy: 0.8510
```




```
1/1 [==============================] - 6s 6s/step
4.84247
The review is positive
```

EXPERIMENT-7

Aim: Implement LSTM network.

Platform used: Colab

Theory:

Long Short-Term Memory is an improved version of recurrent neural network designed by Hochreiter & Schmidhuber. LSTM is well-suited for sequence prediction tasks and excels in capturing long-term dependencies. Its applications extend to tasks involving time series and sequences. LSTM's strength lies in its ability to grasp the order dependence crucial for solving intricate problems, such as machine translation and speech recognition. A traditional RNN has a single hidden state that is passed through time, which can make it difficult for the network to learn long-term dependencies. LSTMs address this problem by introducing a memory cell, which is a container that can hold information for an extended period. LSTM networks are capable of learning long-term dependencies in sequential data, which makes them well-suited for tasks such as language translation, speech recognition, and time series forecasting. LSTMs can also be used in combination with other neural network architectures, such as Convolutional Neural Networks (CNNs) for image and video analysis. The memory cell is controlled by three gates: the input gate, the forget gate, and the output gate. These gates decide what information to add to, remove from, and output from the memory cell. The input gate controls what information is added to the memory cell. The forget gate controls what information is removed from the memory cell. And the output gate controls what information is output from the memory cell. This allows LSTM networks to selectively retain or discard information as it flows through the network, which allows them to learn long-term dependencies.

Input:

```
pip install torch
```

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
filename = "/content/wonderland.txt"# load ascii text and covert to lowercase
raw_text = open(filename, 'r', encoding='utf-8').read()
raw_text = raw_text.lower()
chars = sorted(list(set(raw_text)))# create mapping of unique chars to integers
char_to_int = dict((c, i) for i, c in enumerate(chars))
n_chars = len(raw_text)# summarize the loaded data
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
seq_length = 100   # prepare the dataset of input to output pairs encoded as integers
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
X = torch.tensor(dataX, dtype=torch.float32).reshape(n_patterns, seq_length, 1)# reshape X to be [samples, time steps, features]
X = X / float(n_vocab)
y = torch.tensor(dataY)
class CharModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.lstm = nn.LSTM(input_size=1, hidden_size=256, num_layers=1, batch_first=True)
```

```
        self.dropout = nn.Dropout(0.2)
        self.linear = nn.Linear(256, n_vocab)
    def forward(self, x):
        x, _ = self.lstm(x)
        x = x[:, -1, :] # take only the last output
        x = self.linear(self.dropout(x)) # produce output
        return x
n_epochs = 5
batch_size = 12
model = CharModel()
optimizer = optim.Adam(model.parameters())
loss_fn = nn.CrossEntropyLoss(reduction="sum")
loader = data.DataLoader(data.TensorDataset(X, y), shuffle=True, batch_size=batch_size)
best_model = None
best_loss = np.inf
for epoch in range(n_epochs):
    model.train()
    for X_batch, y_batch in loader:
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()# Validation
    model.eval()
    loss = 0
    with torch.no_grad():
        for X_batch, y_batch in loader:
            y_pred = model(X_batch)
            loss += loss_fn(y_pred, y_batch)
        if loss < best_loss:
            best_loss = loss
            best_model = model.state_dict()
        print("Epoch %d: Cross-entropy: %.4f" % (epoch, loss))
torch.save([best_model, char_to_int], "single-char.pth")
```

```
seq_length = 100
start = np.random.randint(0, len(raw_text)-seq_length)
prompt = raw_text[start:start+seq_length]
```

```
import numpy as np
import torch
import torch.nn as nn

best_model, char_to_int = torch.load("single-char.pth")
n_vocab = len(char_to_int)
int_to_char = dict((i, c) for c, i in char_to_int.items())
class CharModel(nn.Module):# reload the model
    def __init__(self):
        super().__init__()
        self.lstm = nn.LSTM(input_size=1, hidden_size=256, num_layers=1, batch_first=True)
        self.dropout = nn.Dropout(0.2)
        self.linear = nn.Linear(256, n_vocab)
    def forward(self, x):
        x, _ = self.lstm(x)# take only the last output
        x = x[:, -1, :]
        x = self.linear(self.dropout(x)) # produce output
        return x
model = CharModel()
model.load_state_dict(best_model)
filename = "wonderland.txt"# randomly generate a prompt
seq_length = 100
raw_text = open(filename, 'r', encoding='utf-8').read()
raw_text = raw_text.lower()
start = np.random.randint(0, len(raw_text)-seq_length)
prompt = raw_text[start:start+seq_length]
pattern = [char_to_int[c] for c in prompt]
model.eval()
print('Prompt: "%s"' % prompt)
with torch.no_grad():
    for i in range(1000):  # format input array of int into PyTorch tensor
        x = np.reshape(pattern, (1, len(pattern), 1)) / float(n_vocab)
```

```
raw_text = open(filename, 'r', encoding='utf-8').read()
raw_text = raw_text.lower()
start = np.random.randint(0, len(raw_text)-seq_length)
prompt = raw_text[start:start+seq_length]
pattern = [char_to_int[c] for c in prompt]
model.eval()
print('Prompt: "%s"' % prompt)
with torch.no_grad():
    for i in range(1000):  # format input array of int into PyTorch tensor
        x = np.reshape(pattern, (1, len(pattern), 1)) / float(n_vocab)
        x = torch.tensor(x, dtype=torch.float32)
        prediction = model(x)# generate logits as output from the model
        index = int(prediction.argmax())# convert logits into one character
        result = int_to_char[index]
        print(result, end="")
        pattern.append(index) # append the new character into the prompt for the next iteration
        pattern = pattern[1:]
print()
print("Done.")
```

Output:

```
Total Characters:  144598
Total Vocab:  50
Total Patterns:  144498
Epoch 0: Cross-entropy: 379872.6562
Epoch 1: Cross-entropy: 356066.7500
Epoch 2: Cross-entropy: 336361.0312
Epoch 3: Cross-entropy: 318729.7812
Epoch 4: Cross-entropy: 305848.5938
```

```
Prompt: "ant things, all because they
_would_ not remember the simple rules their friends had taught them:
su"
en a lott oo the tai so tee soee i shink ier sea toee i shanl toee i shanl toued to tee soee i shanl toee i shan the war io the
Done.
```

Conclusion:

Successfully implemented the LONG SHORT TERM MEMORY network.

AMITY SCHOOL OF ENGINEERING & TECHNOLOGY

AMITY UNIVERSITY CAMPUS, SECTOR-125, NOIDA-201303

Deep Learning and Neural Network Lab

PRACTICAL FILE

COURSE CODE: AIML302

Submitted to:
Dr. Pintu Kumar Ram

Submitted by:
Boddu Asmitha Bhavya
A2305221386
6CSE-6X

INDEX

S.No.	Experiment	Date	Signature
1.	Exploring Kaggle Website and DownloadingDataset to work upon it.		
2.	Write a python program on taking a iris data setfrom Kaggle and Perform these on that data set 1.missing values 2.outliers 3.reputation		
3.	Program that simulates the operation of an AND andOR gate with adjustable weights		
4.	Implementing Artificial Neural Network.		
5.	Implementing Convolutional Neural Network.		
6.	Implement RNN network		
7.	Implement LSTM network		
8.	Perform the sentiment analysis based on data (userchoice data) end to end workflow (loading preprocessing modelling compiling)		