EXPERIMENT - 9

AIM: Applying the Deep Learning Models in the field of Natural Language Processing

Theory:

NLP involves processing and understanding human language. Deep learning models such as LSTMs and GRUs are used for tasks like text generation, translation, and sentiment analysis, as they handle sequential data efficiently.

Source Code:

```
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
import numpy as np

# Sample text data
texts = ["Deep learning is amazing", "Natural Language Processing is fun"]
tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

# Pad sequences
data = pad_sequences(sequences, maxlen=10)

# Build LSTM model for NLP
model = Sequential()
model.add(Embedding(input_dim=1000, output_dim=64, input_length=10))
model.add(LSTM(64))
model.add(Dense(1, activation='sigmoid'))

# Compile and train the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()

# Sample training data (replace with your own data)
X_train = np.array(data)
y_train = np.array([1, 0])  # Example labels

# Train the model
model.fit(X_train, y_train, epochs=10)

# Sample evaluation data (replace with your own data)
X_eval = np.array(data)
y_eval = np.array([1, 0])  # Example labels

# Evaluate the model
loss, accuracy = model.evaluate(X_eval, y_eval)
print("Loss:", loss)
print("Accuracy:", accuracy)
```

Output:

```
Model: "sequential_3"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                  ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ embedding_3 (Embedding)               │ ?                             │     0 (unbuilt) │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ lstm_3 (LSTM)                         │ ?                             │     0 (unbuilt) │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ dense_3 (Dense)                       │ ?                             │     0 (unbuilt) │
└───────────────────────────────────────┴───────────────────────────────┴─────────────────┘
 Total params: 0 (0.00 B)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 0 (0.00 B)
Epoch 1/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 3s 3s/step - accuracy: 1.0000 - loss: 0.6873
Epoch 2/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 195ms/step - accuracy: 1.0000 - loss: 0.6836
Epoch 3/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 136ms/step - accuracy: 1.0000 - loss: 0.6797
Epoch 4/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - accuracy: 1.0000 - loss: 0.6757
Epoch 5/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - accuracy: 1.0000 - loss: 0.6714
Epoch 6/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 1.0000 - loss: 0.6669
Epoch 7/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - accuracy: 1.0000 - loss: 0.6619
Epoch 8/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - accuracy: 1.0000 - loss: 0.6566
Epoch 9/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 1.0000 - loss: 0.6508
Epoch 10/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 1.0000 - loss: 0.6444
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 437ms/step - accuracy: 1.0000 - loss: 0.6375
Loss: 0.6374600529670715
Accuracy: 1.0
```

EXPERIMENT - 10

AIM: Applying the Convolution Neural Network on computer vision problems

Theory:

CNNs have revolutionized computer vision by enabling accurate detection and classification of objects in images. They use convolutional layers to detect patterns like edges and shapes, followed by pooling and fully connected layers for decision-making.

Source Code:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Load CIFAR-10 dataset (for computer vision tasks)
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Preprocess data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Build CNN model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile and train the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.summary()
model.fit(x_train, y_train, epochs=10, batch_size=32)
```

Output:

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 6s 0us/step
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
flatten (Flatten)	(None, 7200)	0
dense (Dense)	(None, 128)	921,728
dense_1 (Dense)	(None, 10)	1,290

```
 Total params: 923,914 (3.52 MB)
 Trainable params: 923,914 (3.52 MB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.4149 - loss: 1.6340
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.5959 - loss: 1.1527
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.6480 - loss: 1.0168
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.6763 - loss: 0.9258
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.7025 - loss: 0.8501
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7308 - loss: 0.7741
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.7449 - loss: 0.7192
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.7663 - loss: 0.6661
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7869 - loss: 0.6094
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8090 - loss: 0.5504
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6478 - loss: 1.1648
Test Accuracy: 0.6431000232696533
```

EXPERIMENT - 11

AIM: Implement Deep Q Networks for CartPole problem where the agent has to balance a pole on a cart.

Theory: Deep Q-Networks combine Q-learning with deep neural networks to handle environments with large state spaces. In the CartPole problem, the goal is to balance a pole on a cart by applying forces to the left or right. The network learns the Q-value function to predict the best action for each state.

Source Code:

```
import gym
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import random
from collections import deque

# Initialize environment
env = gym.make('CartPole-v1')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n

# Build a simpler Deep Q-Network
model = Sequential()
model.add(Dense(16, input_dim=state_size, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(action_size, activation='linear'))
model.compile(loss='mse', optimizer=tf.keras.optimizers.Adam(learning_rate=0.001))

# Parameters for Deep Q-Learning
gamma = 0.95
epsilon = 1.0
epsilon_min = 0.01
epsilon_decay = 0.995
batch_size = 64
memory = deque(maxlen=500)

def remember(state, action, reward, next_state, done):
    memory.append((state, action, reward, next_state, done))

def act(state):
    if np.random.rand() <= epsilon:
        return env.action_space.sample()
    q_values = model.predict(state, verbose=0)
    return np.argmax(q_values[0])

def replay():
    global epsilon
    if len(memory) < batch_size:
        return
    minibatch = random.sample(memory, batch_size)
    for state, action, reward, next_state, done in minibatch:
        target = reward if done else reward + gamma * np.amax(model.predict(next_state, verbose=0)[0])
        target_f = model.predict(state, verbose=0)
        target_f[0][action] = target
        model.fit(state, target_f, epochs=1, verbose=0)
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay
```

```
# Training the agent with fewer episodes and steps
num_episodes = 10
for e in range(num_episodes):
    state = env.reset()
    state = np.reshape(state, [1, state_size])
    for time in range(200):
        action = act(state)
        next_state, reward, done, _ = env.step(action)
        reward = reward if not done else -10
        next_state = np.reshape(next_state, [1, state_size])
        remember(state, action, reward, next_state, done)
        state = next_state
        if done:
            print(f"Episode: {e+1}/{num_episodes}, Score: {time}, Epsilon: {epsilon:.2f}")
            break
        replay()

# Output: Model Summary
model.summary()
```

Output:

```
Episode: 1/10, Score: 30, Epsilon: 1.00
Episode: 2/10, Score: 10, Epsilon: 1.00
Episode: 3/10, Score: 27, Epsilon: 0.97
Episode: 4/10, Score: 26, Epsilon: 0.85
Episode: 5/10, Score: 21, Epsilon: 0.77
Episode: 6/10, Score: 17, Epsilon: 0.70
Episode: 7/10, Score: 11, Epsilon: 0.67
Episode: 8/10, Score: 12, Epsilon: 0.63
Episode: 9/10, Score: 13, Epsilon: 0.59
Episode: 10/10, Score: 8, Epsilon: 0.56
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 16)	80
dense_7 (Dense)	(None, 16)	272
dense_8 (Dense)	(None, 2)	34

```
Total params: 1,160 (4.54 KB)
Trainable params: 386 (1.51 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 774 (3.03 KB)
```

EXPERIMENT - 12

AIM: Demonstrate the application of transfer learning using Cartpole dataset and MountainCar dataset.

Theory: Transfer learning allows a model trained on one problem to be used in another related problem by fine-tuning. In this case, a model trained on CartPole can be adapted for the MountainCar problem, transferring knowledge about control systems between tasks.

Source Code:

```
import gym
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Categorical

# Set up Cartpole environment
cartpole_env = gym.make('CartPole-v0')
cartpole_state_size = cartpole_env.observation_space.shape[0]
cartpole_action_size = cartpole_env.action_space.n

# Set up MountainCar environment
mountaincar_env = gym.make('MountainCar-v0')
mountaincar_state_size = mountaincar_env.observation_space.shape[0]
mountaincar_action_size = mountaincar_env.action_space.n

# Define the base model
class BaseModel(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(BaseModel, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, hidden_size)
        self.fc4 = nn.Linear(hidden_size, hidden_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        x = self.fc4(x)
        x = self.relu(x)
        return x

# Create the Cartpole model
class CartpoleModel(nn.Module):
    def __init__(self, base_model):
        super(CartpoleModel, self).__init__()
        self.base_model = base_model
        self.fc3 = nn.Linear(base_model.fc2.out_features, cartpole_action_size)

    def forward(self, x):
        x = self.base_model(x)
        x = self.fc3(x)
        return x

cartpole_model = CartpoleModel(BaseModel(cartpole_state_size, 128))

# Create the MountainCar model
class MountainCarModel(nn.Module):
    def __init__(self, base_model):
        super(MountainCarModel, self).__init__()
```

```python
        self.base_model = base_model
        self.fc3 = nn.Linear(base_model.fc2.out_features, mountaincar_action_size)

    def forward(self, x):
        x = self.base_model(x)
        x = self.fc3(x)
        return x

mountaincar_model = MountainCarModel(BaseModel(mountaincar_state_size, 128))

# Train and evaluate the models
def train_model(model, env, num_episodes=500, gamma=0.99):
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    for episode in range(num_episodes):
        state = env.reset()
        done = False
        rewards = []
        log_probs = []

        while not done:
            state = torch.from_numpy(state).float().unsqueeze(0)
            output = model(state)
            dist = Categorical(logits=output)
            action = dist.sample().item()
            next_state, reward, done, _ = env.step(action)
            log_prob = dist.log_prob(torch.tensor(action))
            rewards.append(reward)
            log_probs.append(log_prob)
            state = next_state

        returns = []
        R = 0
        for r in rewards[::-1]:
            R = r + gamma * R
            returns.insert(0, R)

        loss = 0
        for log_prob, R in zip(log_probs, returns):
            loss -= log_prob * R

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

def evaluate_model(model, env, num_episodes=10):
    total_reward = 0
    for _ in range(num_episodes):
        state = env.reset()
        done = False
        while not done:
            state = torch.from_numpy(state).float().unsqueeze(0)
            action = model(state).argmax().item()
            state, reward, done, _ = env.step(action)
            total_reward += reward
    return total_reward / num_episodes

# Show Model
def print_model_summary(model):
    # Print each layer of the model
    print(f"Model Summary:")
    for name, layer in model.named_children():
        print(f"{name}: {layer}")

# Print the summary of the Cartpole model
print("\nCartpole Model Summary:")
print_model_summary(cartpole_model)

# Print the summary of the MountainCar model
print("\nMountainCar Model Summary:")
print_model_summary(mountaincar_model)
```

```
# Train and evaluate the Cartpole model
train_model(cartpole_model, cartpole_env, num_episodes=500)
cartpole_score = evaluate_model(cartpole_model, cartpole_env)
print(f"Cartpole score: {cartpole_score}")

# Train and evaluate the MountainCar model
train_model(mountaincar_model, mountaincar_env, num_episodes=1000)
mountaincar_score = evaluate_model(mountaincar_model, mountaincar_env)
print(f"MountainCar score: {mountaincar_score}")
```

Output:

```
Cartpole Model Summary:
Model Summary:
base_model: BaseModel(
  (fc1): Linear(in_features=4, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=128, bias=True)
  (relu): ReLU()
)
fc3: Linear(in_features=128, out_features=2, bias=True)

MountainCar Model Summary:
Model Summary:
base_model: BaseModel(
  (fc1): Linear(in_features=2, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=128, bias=True)
  (relu): ReLU()
)
fc3: Linear(in_features=128, out_features=3, bias=True)
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_
  if not isinstance(terminated, (bool, np.bool8)):
Cartpole score: 63.7
MountainCar score: -200.0
```