Universidade Federal do Rio de Janeiro
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

Programa de Engenharia de Sistemas e Computação

CPS769 - Introdução à Inteligência Artificial e Aprendizagem Generativa

Prof. Dr. Edmundo de Souza e Silva (PESC/COPPE/UFRJ)
Profa. Dra. Rosa M. Leão (PESC/COPPE/UFRJ)
Participação Especial: Gaspare Bruno (Diretor Inovação, ANLIX)

Lista de Exercícios 1a

Luiz Henrique Souza Caldas
email: lhscaldas@cos.ufrj.br

15 de julho de 2024

Questão 1

Esse exemplo simples é para auxiliar a discussão do artigo "Serial Order A Parallel Distributed Processing Approach" que todos já devem ter lido. O objetivo é prever um padrão de figura, por exemplo um quadrado, usando uma Rede Neural Recorrente (RNN). Fornecemos o código em Python de um exemplo de geração do padrão 2-D de quadrados e treinamento de uma RNN para prever a sequência cíclica [0, 25, 0, 25], [0, 75, 0, 25], [0, 75, 0, 75], [0, 25, 0, 75], [0, 25, 0, 25].

1. Entenda o código e explique qual a RNN que ele modela (faça o desenho). Explique a parte do código que define a RNN.

   Resposta:

   O código pode ser explicado dividindo-o em 6 partes:

   (a) Definição do caminho quadrado: O código define um conjunto de coordenadas que formam um caminho quadrado na variável *square_path*.

```
square_path = np.array([
    [0.25, 0.25],
    [0.75, 0.25],
    [0.75, 0.75],
    [0.25, 0.75],
    [0.25, 0.25]
])
```

   (b) Geração dos dados de treinamento: O caminho quadrado é repetido várias vezes para formar os dados de treinamento.

```
num_repeats = 4
data = np.tile(square_path, (num_repeats, 1))
x_train = data[:-1].reshape(-1, 1, 2)
y_train = data[1:].reshape(-1, 2)
```

   (c) Definição e compilação do modelo RNN: O modelo RNN é definido usando uma camada LSTM (*long short-term memory*) seguida de uma camada densa e depois é compilado configurando o algoritmo ADAM como otimizador e o Erro Médio Quadrático como função de perda.

```
model = models.Sequential([
    layers.LSTM(50, activation='relu', input_shape=(num_repeats, 2)),
    layers.Dense(2)
])
model.compile(optimizer='adam', loss='mse')
```

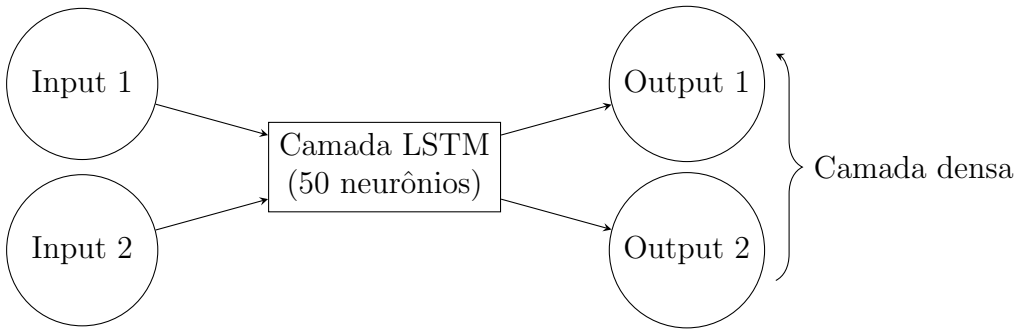


Figura 1: Diagrama de uma Rede Neural Recorrente (RNN) com dois neurônios de entrada e dois neurônios de saída (camada densa) e uma camada LSTM modelada no código fornecido.

(d) Treinamento do modelo: O modelo é treinado com os dados gerados, utilizando inicialmente 300 épocas.

```
model.fit(x_train, y_train, epochs=300, verbose=0)
```

(e) Geração das previsões: As previsões são geradas.

```
predictions = model.predict(x_train[:5])
```

(f) Plotagem dos resultados: As previsões são plotadas e comparadas com o caminho original.

```
plt.plot(data[:, 0], data[:, 1], label='Original Path', linestyle='
    dashed', color='gray')
plt.plot(predictions[:, 0], predictions[:, 1], label='Predicted Path',
     color='blue')
plt.scatter(square_path[:, 0], square_path[:, 1], color='red')
plt.legend()
plt.show()
```

2. Treine a rede. Aprenda como fazer, e explique.

   Resposta:

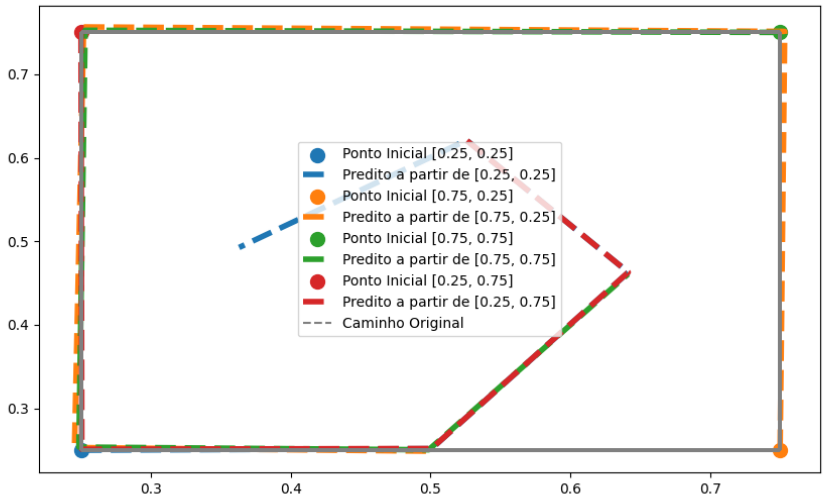
   Como dito no passo (d) do item anterior, o treinamento é ralizado utilizando a função "fit"do modelo. Utilizando a configuração inicial, com 300 épocas, o treinamento demorou cerca de 11 segundos.

3. Faça a previsão de algumas trajetórias, quando o ponto inicial varia. O que você conclui?

   Resposta:

   Foi feita a previsão para o ponto inicial original do código dado, ($x_1 = 0.25$ e $x_2 = 0.25$) e depois foram testados os outros vértices do quadrado. Para isso foi implementada a função *plot_predictions_with_varied_initial_points* (código no final do relatório), na qual os pontos são previstos em sequência, sendo a previsão anterior a entrada da próxima previsão, somando um total de 4 previsões para cada ponto inicial. O resultado pode ser observado na figura abaixo.

Figura 2: Previsão para diferentes pontos iniciais



Todas as previsões divergem do caminho original no ponto ($x_1 = 0.50$ e $x_2 = 0.25$), o que indica uma provavél incompatibilidade do modelo para prever esse caminho, uma vez que foram testadas diversas variações de número de épocas e de repetições do caminho original no treinamento.

4. Modifique a RNN usada e observe o que acontece.

   Resposta:

   Para este teste, o ponto inicial foi retornado para a configuração original ($x_1 = 0.25$ e $x_2 = 0.25$) e foram testadas diferentes combinações de épocas e número de repetições.

Tabela 1: Resultados das modificações

Nº Repetições	Épocas	Tempo Treinamento (s)	Tempo Total (s)
4	300	12	12.2
40	300	13.5	13.7
400	300	26.4	26.6
40	600	19.6	19.8
40	900	29.2	29.1

Pela tabela 1 é possível observar o aumento do tempo de execução, mais específicamente do tempo de treinamento, tanto com o aumento do número de repetições quanto com o aumento do número de épocas. Os resultados das previsões podem ser visualizados nas figuras abaixo.

Figura 3: Previsão para 4 repetições e 300 épocas

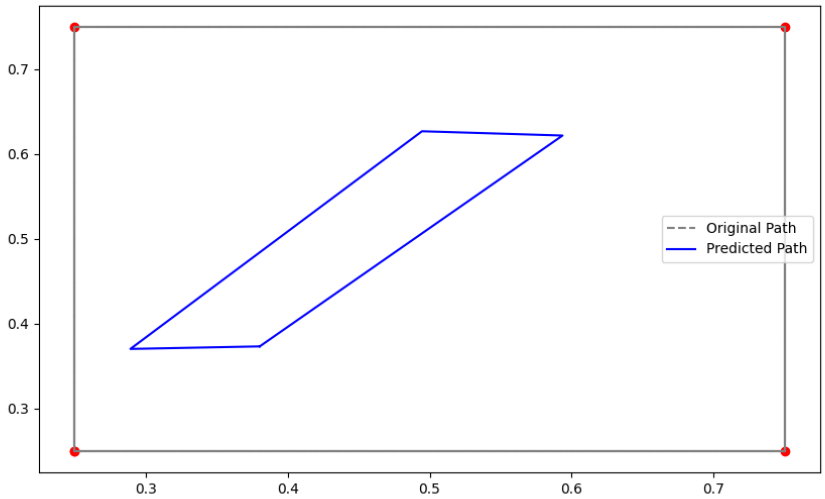


Figura 4: Previsão para 40 repetições e 300 épocas

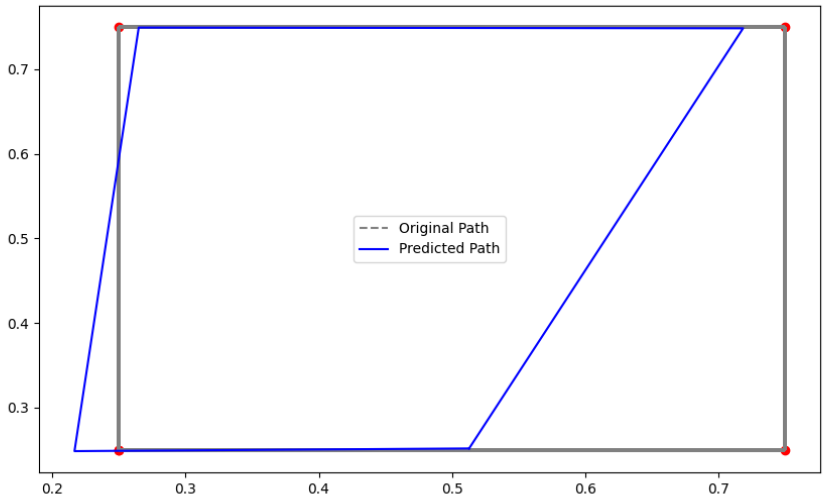


Figura 5: Previsão para 400 repetições e 300 épocas

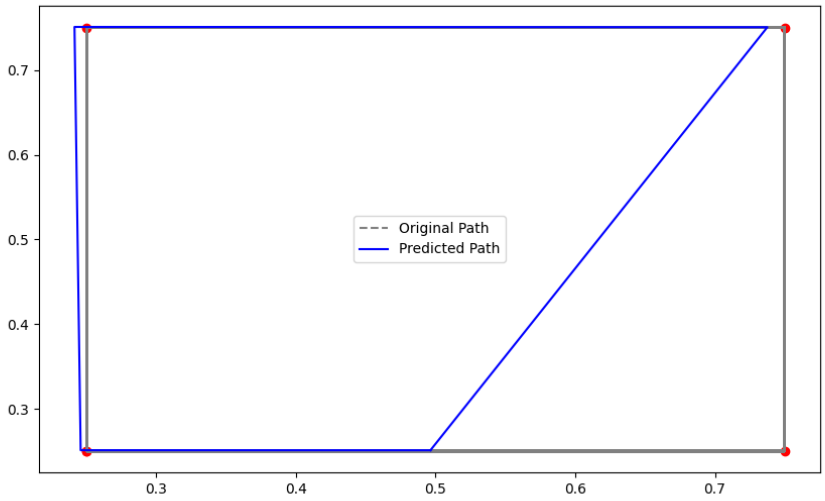


Figura 6: Previsão para 40 repetições e 600 épocas

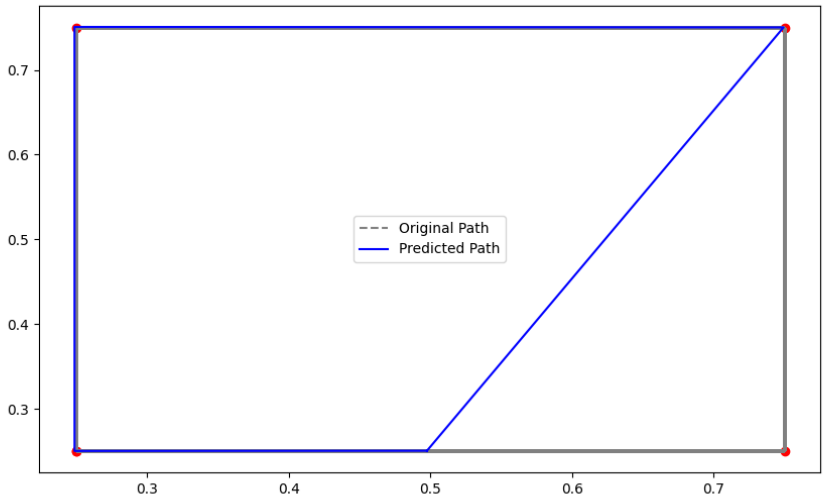
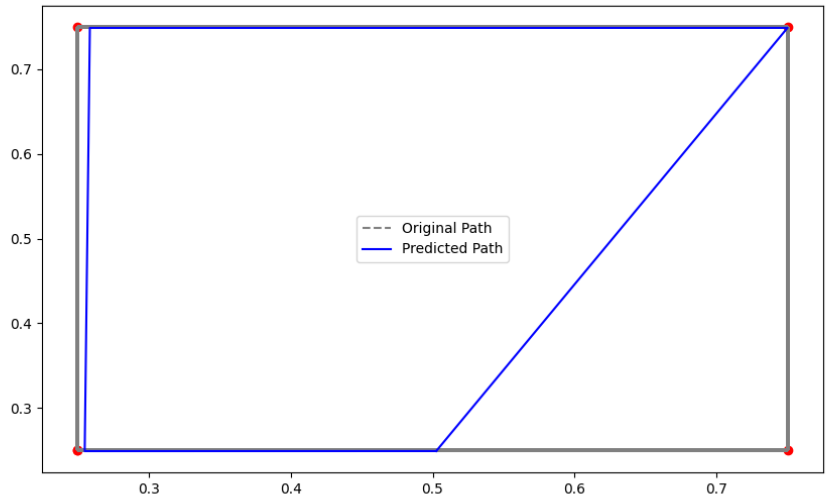


Figura 7: Previsão para 40 repetições e 900 épocas



Pelas figuras foi possível observar que o impacto do aumento do número de repetições na precisão da previsão foi muito maior que o impacto do aumento do número de épocas, o que demonstra a importância do tamanho do dataset de treinamento para o resultado de suas previsões.

5. Quais os pontos principais que você concluiu do artigo "Serial Order A Parallel Distributed Processing Approach"?

   Resposta:

   A teoria de Michael I. Jordan sobre ordem serial em sequências de ações usa redes neurais para entender e reproduzir a ordem das ações ao longo do tempo. Essas redes mantêm uma "memória"do que já aconteceu, usando conexões que alimentam as saídas de volta para as entradas, ajudando a lembrar das ações passadas. A rede aprende ajustando seus parâmetros para reduzir erros entre o que foi previsto e o que realmente aconteceu.

   Isso faz com que a rede consiga generalizar a partir de sequências aprendidas e continuar funcionando bem, mesmo com pequenas perturbações. Essencialmente, a rede se torna uma memória dinâmica que pode voltar às suas trajetórias aprendidas, garantindo que as sequências de ações sejam produzidas corretamente, mesmo começando de pontos diferentes.

6. Qual a differença da RNN usada no código em relação ao artigo "Serial Order A Parallel Distributed Processing Approach"?

   Resposta:

   No artigo, a rede é composta por unidades de plano, estado e saída, com conexões recorrentes que definem a função de próximo estado, e enfatiza a capacidade de aprender e generalizar trajetórias no espaço de estado. Em contraste, a RNN do código usa uma arquitetura LSTM simples, que captura dependências temporais através de sua memória interna, sem distinções

explícitas entre plano e estado. A abordagem de Jordan foca em representações distribuídas e paralelismo, enquanto a RNN do código adota técnicas convencionais de redes neurais recorrentes.

7. Inclua, nos dados de treino uma figura de uma espiral quadrada, e experimente o que a rede aprendeu.

   Resposta:

   Para gerar o caminho em espiral foi implementada a função *generate_spiral_square* (código no final do relatório), a qual gera uma lista no mesmo formato da *square_path* fornecida, porém com os pontos formando uma espiral quadrada. O resultado da previsão pode ser observado na figura abaixo.

Código

O código abaixo encontra-se no repositório https://github.com/lhscaldas/cps769-ai-gen, bem como o arquivo LaTex com o relatório.

Código 1: código fornecido completo com algumas modificações

```python
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

# Desabilitar GPU para treinamento (se necessário)
os.environ['CUDA_VISIBLE_DEVICES'] = '-1'

# Definir as coordenadas do caminho quadrado
square_path = np.array([
    [0.25, 0.25],
    [0.75, 0.25],
    [0.75, 0.75],
    [0.25, 0.75],
    [0.25, 0.25]
])

def generate_spiral_square(turns=3, points_per_turn=4):
    path = []
    step = 0.1
    x, y = 0.5, 0.5
    for turn in range(turns):
        path.append([x, y])
        x += step * (turn + 1)
        path.append([x, y])
        y += step * (turn + 1)
        path.append([x, y])
        x -= step * (turn + 1)
        path.append([x, y])
        y -= step * (turn + 1)
    return np.array(path)
```

```
33
34 spiral_path = generate_spiral_square(turns=3)
35
36 # Gerar os dados de treinamento repetindo o caminho quadrado
37 num_repeats = 40
38 data = np.tile(spiral_path, (num_repeats, 1))
39
40 # Preparar os dados de treinamento
41 x_train = data[:-1].reshape(-1, 1, 2)
42 y_train = data[1:].reshape(-1, 2)
43
44 # Definir o modelo RNN
45 model = models.Sequential([
46     layers.LSTM(50, activation='relu', input_shape=(1, 2)),
47     layers.Dense(2)
48 ])
49
50 # Compilar o modelo
51 model.compile(optimizer='adam', loss='mse')
52
53 # Treinar o modelo
54 model.fit(x_train, y_train, epochs=500, verbose=0)
55
56 # Função para fazer previsões com pontos iniciais variados
57 def plot_predictions_with_varied_initial_points(model, initial_points):
58     plt.figure(figsize=(10, 6))
59
60     for point in initial_points:
61         current_input = np.array(point).reshape(1, 1, 2)
62         predictions = [point]
63
64         for _ in range(4):
65             next_prediction = model.predict(current_input)
66             predictions.append(next_prediction.flatten())
67             current_input = next_prediction.reshape(1, 1, 2)
68
69         predictions = np.array(predictions)
70
71         # Plotar o ponto inicial
72         plt.scatter(point[0], point[1], marker='o', s=100, label=f'Ponto
               Inicial {point}')
73
74         # Plotar as previsões
75         plt.plot(predictions[:, 0], predictions[:, 1], label=f'Predito a partir
               de {point}', linestyle='dashed', linewidth=4)
76
77     # Plotar o caminho original para referência
78     plt.plot(spiral_path[:, 0], spiral_path[:, 1], label='Caminho Original',
           linestyle='dashed', color='gray')
79     plt.legend()
80     plt.show()
81
82 # Pontos iniciais para a previsão
83
```

```
initial_points = [
    [0.25, 0.25],
    [0.75, 0.25],
    [0.75, 0.75],
    [0.25, 0.75]
]

initial_points = [
    [0.50, 0.50],
]

# Plotar previsões com pontos iniciais variados
plot_predictions_with_varied_initial_points(model, initial_points)
```