Relatório do Laboratório 8: Imitation Learning com Keras

Isabelle Ferreira de Oliveira
CT-213 - Engenharia da Computação 2020
Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos, Brasil
isabelle.ferreira3000@gmail.com

***Resumo*—Esse relatório documenta a cópia de um movimento de caminhada de um robô humanoide usando a técnica chamada *imitation learning*. Para isso, foi utilizado o *framework* de *Deep Learning* Keras, que facilita o uso do *framework* Tensorflow.**

***Index Terms*—*Imitation learning*, *deep learning*, Keras, Tensorflow**

I. Introdução

Deep learning é um tipo de aprendizado de máquina que treina para aprender através do reconhecimento de padrões em várias camadas de processamento. Entre as tarefas possíveis realizáveis estão o reconhecimento de fala, identificação de imagem e previsões, entre outras tarefas realizadas por seres humanos.

Esse tipo de estratégia possibilita, também, aprender por imitação. Assim, o comportamento desejado (uma política de controle, por exemplo) é copiado usando aprendizado supervisionado.

O *framework* Keras facilita o uso do *framework* Tensorflow para problemas de *Deep learning*, transformando a implementação, treino e resultado da rede neural em chamadas de API. O funcionamento dessas chamadas pode ser visto a seguir. Em seguida, será apresentado como isso foi aplicado no contexto do laboratório.

```
from keras import models

# Creates the neural network model in Keras
model = models.Sequential()
```

Sequential() cria uma pilha linear de camadas.

```
from keras import layers, activations,
    regularizers

# Adds the first layer
model.add(layers.Dense(num_neurons,
   activation=activations.some_function,
   input_shape=(input_size,),
   kernel_regularizer=regularizers.l2(lambda)))

# Adds another layer (not first)
model.add(layers.Dense(num_neurons,
   activation=activations.some_function,
   kernel_regularizer=regularizers.l2(lambda)))
```

Para a criação de uma camada na rede neural através do Keras, utiliza-se a função *model.add(layers.Dense())*, sendo o primeiro argumento referente ao número de neurônios nessa camada; "activation"configura a função de ativação; "input_shape"representa o tamanho da entrada; e "kernel_regularizer"configura a regularização para essa camada.

```
from keras import losses, optimizers, metrics

# Configures the model for training
model.compile(optimizer=optimizers.Adam(),
   loss=losses.binary_crossentropy,
   metrics=[metrics.binary_accuracy])

# Trains the model for a given number of
    epochs
history = model.fit(inputs,
   expected_outputs,
   batch_size=size_of_batch,
   epochs=num_epochs)
```

Por fim, configura-se o modelo para o treino, escolhendo o otimizer, a função de custo e as métricas; e se treina o modelo para um determinado conjunto de entrada, tendo as saídas esperadas, o tamanho do batch e quantas épocas serão efetuadas.

II. Implementação

Para a implementação da rede neural conforme os parâmetros requisitados pelo roteiro do laboratório [1] e apresentada na tabela da Figura 1, era necessário utilizar do código de adição de camadas a uma rede, além de configuração e treino do modelo, apresentado na Introdução.

Tendo em vista que em *keras.activations* não há função de ativação Leaky ReLU, utilizou-se a recomendação sugerida pelo roteiro [1] para usar Leaky ReLU no Keras, ou seja, foi adicionado uma camada do tipo LeakyRelu após ter definido uma camada (usando função de ativação linear).

Além disso, para configurar o modelo, setou-se o parâmetro *loss* da função *compile()* para *losses.mean_squared_error*, uma vez que foi utilizado erro quadrático.

Por fim, para treinar, o tamanho do *batch* foi o tamanho total de entradas, para que seja usado todo o *dataset* em cada iteração do treinamento.

Layer	Neurons	Activation Function
Dense	75	Leaky ReLU ($\alpha = 0,01$)
Dense	50	Leaky ReLU ($\alpha = 0,01$)
Dense	20	Linear

Figura 1. Exemplo de neurônio. Essa imagem de exemplo foi apresentada no roteiro [1]

Recebendo os valores de *input*, a função *forward_propagation()* foi implementada conforme apresentado no pseudo-código da Introdução, com o detalhe que a função de ativação g de todos os neurônios utilizada foi a *sigmoid*. Além disso, tanto a função *sigmoid* quanto sua derivada já eram fornecidos pelo código base.

Já na implementação de *back_propagation()*, também foi implementado conforme o sugerido no pseudo-código da Introdução, ou seja, atualizando os valores de pesos e *biases* subtraindo de seu valor um fator de aprendizado *alpha* vezes os gradientes de pesos e *biases*, respectivamente. Esses valores de gradientes são aqueles calculados em *compute_gradient_back_propagation()*, supondo que essa função já foi corretamente implementado. Foi utilizado para *alpha* o mesmo que o sugerido pelo código base.

Finalmente, para implementar a função *compute_gradient_back_propagation()*, também foi feito de forma bastante semelhante ao apresentado na Introdução, com o detalhe de usar *numpy.multiply* para multiplicar termo a termo os vetores referentes aos pesos e a derivada da função de ativação.

Para testar o funcionamento dessas implementações, foi inicialmente alterado o valor da variável *classification_function* (entre "sum_gt_zero"no arquivo *test_neural_network.py* do código base, gerando imagens dos resultados da classificação de cada *dataset* para cada uma dessas funções (função "soma $>$ 0"e função "xor") usando rede neural.

Já tendo ciência da correta implementação dos algoritmos de aprendizado, foi feito por fim um teste da segmentação de cores, executando o código do arquivo *test_color_segmentation.py*, conforme indicado no roteiro [1]. Os gráficos de resultados tanto dessa segmentação de cores, quanto das funções anteriores, foram apresentados nas Figuras **??** a **??**.

III. Resultados e Conclusões

A. Teste das Funções soma > 0 e xor

O aprendizado com a rede neural foi executado para as duas funções já citadas na seção anterior. Os resultados dessas execuções foram satisfatórios e saíram conforme o esperado, comprovando o correto funcionamento da implementação e a validade da utilização de rede neural com aprendizado supervisionado no aprendizado dessas funções. Esses resultados foram apresentados nas Figuras de **??** a **??**.

Vale reparar que, nas Figuras **??** e **??**, os custos não são sempre decrescentes com o passar das épocas. Isso, entretanto, não configura erro, uma vez que os ruídos nos *dataset* acabam interferindo durante o aprendizado. O resultado continua correto uma vez que a tendência geral é a diminuição desse custo, até a convergência. O mesmo vale, inclusive, para o gráfico da Figura **??**, que será melhor discutido na subseção seguinte.

A comparação entre as Figuras **??** e **??** e entre as Figuras **??** e **??** demonstra que a classificação do *dataset* aconteceu de maneira satisfatória.

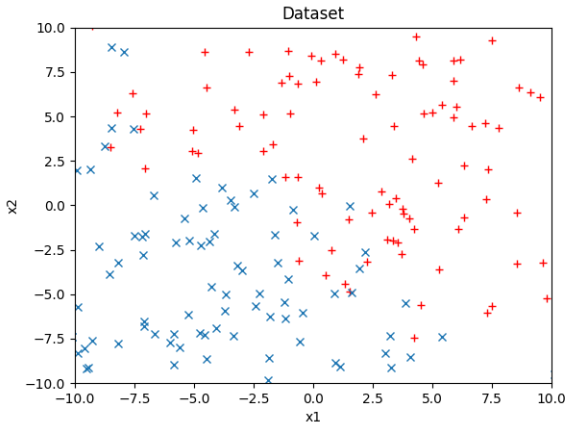

Figura 2. Exemplo de neurônio. Essa imagem de exemplo foi apresentada no roteiro [1]

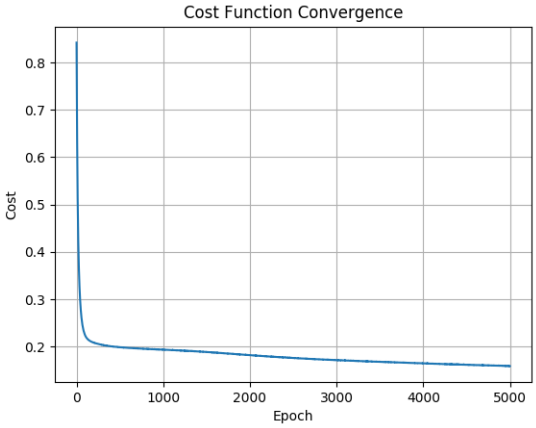

Figura 3. Exemplo de rede neural, com duas camadas (1 camada de entrada, 1 camada escondida e 1 camada de saída), como a trabalhada nesse laboratório. Essa imagem de exemplo foi apresentada no site [2]

B. Teste da segmentação de cores

O custo na Figura **??** segue a tendência geral de diminuição e convergência com o passar das épocas, apesar de alguns picos já explicados na subseção anterior. Esse resultado é satisfatório e pode ser melhor observado na comparação entre as Figuras **??** e **??**, que demonstra que a rede neural realmente conseguiu aprender a classificar as cores verdes e branco com considerável acerto. Vale notar também as cores não

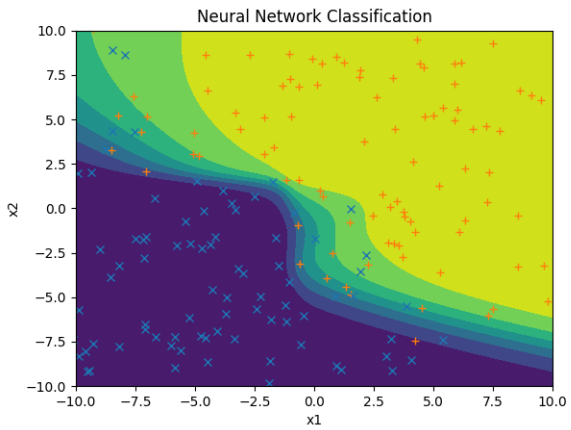

Figura 4. Convergência da função de custo para a função *soma* > *0*.

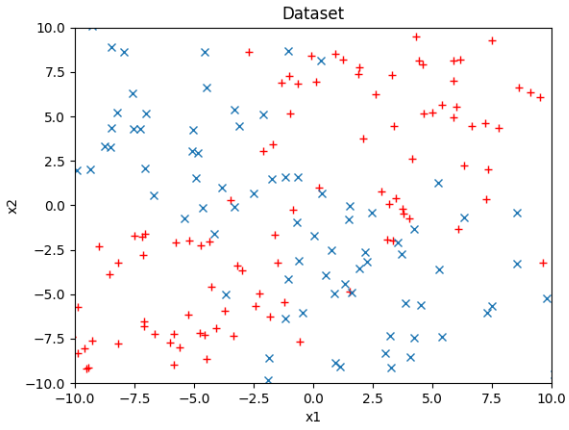

Figura 7. Resultado da classificação por rede neural para a função *soma* > *0*.

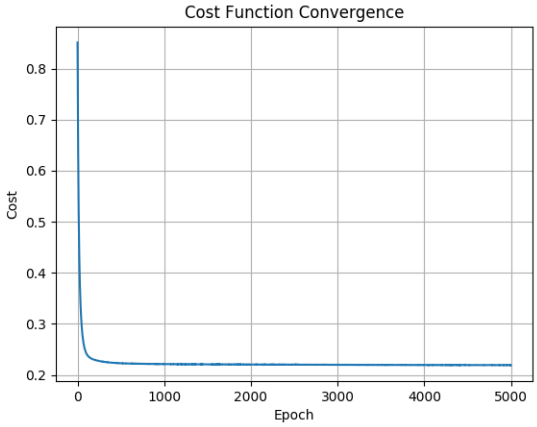

Figura 5. *Dataset* utilizado para o aprendizado da função *xor*.

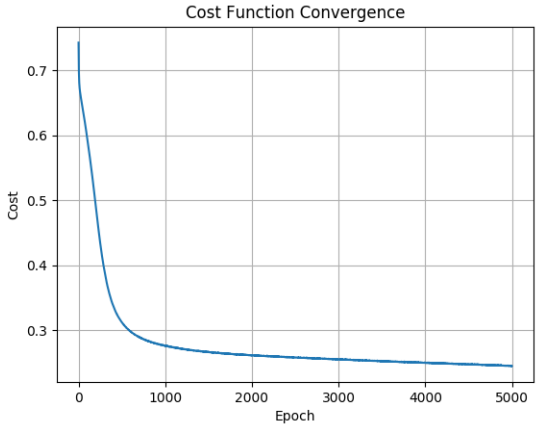

Figura 8. *Dataset* utilizado para o aprendizado da função *soma* > *0*.

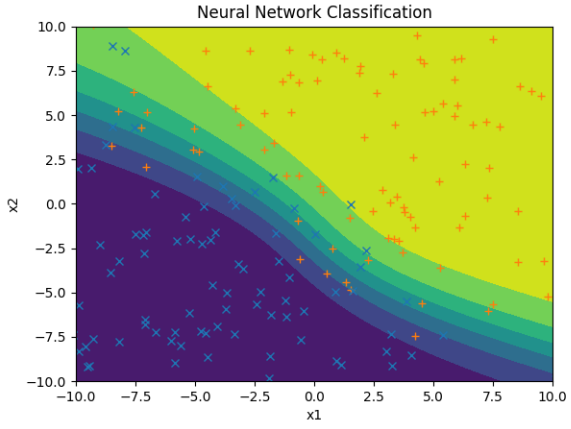

Figura 6. Convergência da função de custo para a segmentação de cores.

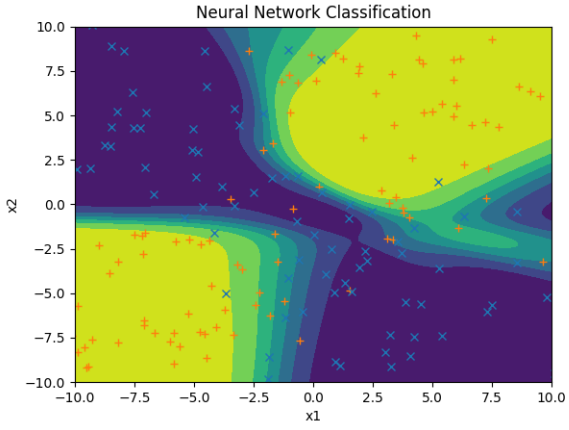

Figura 9. Convergência da função de custo para a função *xor*.

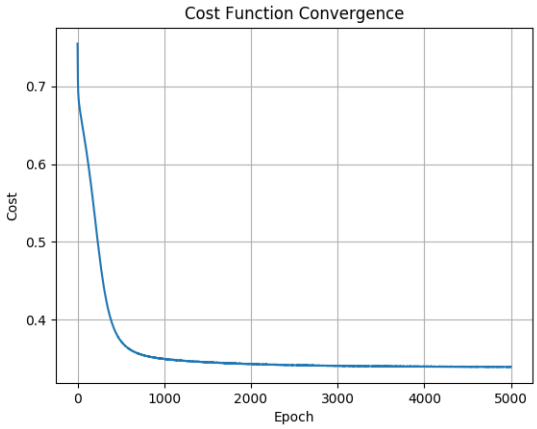

Figura 10. Imagem original a ter cores segmentadas pela rede neural.

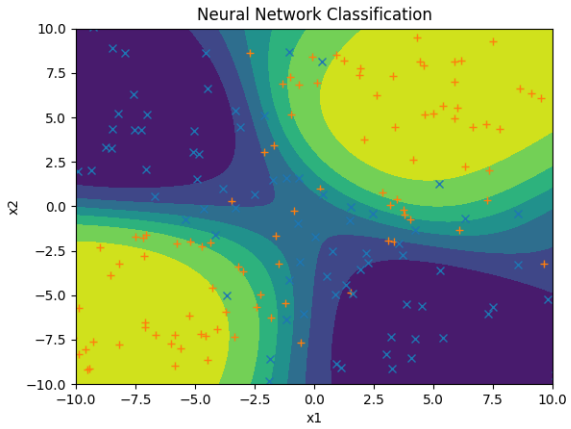

Figura 11. Imagem já com as cores segmentadas pela rede neural.

identificadas, como a bola laranja, que foi deixada como preta (cor representativa de impossibilidade de identificar).

Tendo em vista o que foi apresentado, pode-se notar, por fim, que esses algoritmos realmente se demonstraram eficazes em encontrar os pesos e *biases* dessa rede neural, além de demonstrar a capacidade de aprendizado de uma rede neural para esses problemas de classificação.

REFERÊNCIAS

[1] M. Maximo, “Roteiro: Laboratório 5 - Estratégias Evolutivas”. Instituto Tecnológico de Aeronáutica, Departamento de Computação. CT-213, 2019.

[2] Towards Data Science, "Neural Net from scratch". Acessado em https://towardsdatascience.com/neural-net-from-scratch-using-numpy-71a31f6e3675.

[3] SAS, "Redes Neurais: O que são e qual a sua importância?". Acessado em https://www.sas.com/pt_br/insights/analytics/neural-networks.html.