

Questão 1

Coefficiente de regularização utilizado (taxa de acertos):

675.58805031572205734846647829

Coefficiente de regularização para erro quadrático: 55.71523605095193687475330080

A matriz confusão obtida pelo classificador na questão 1 está expressa abaixo:

2119,	12,	3,	3,	7,	0,	9,	25,	1,	0
94,	1628,	40,	33,	6,	76,	44,	56,	5,	33
41,	61,	1739,	6,	37,	25,	35,	46,	44,	16
32,	5,	3,	1744,	14,	18,	5,	18,	91,	3
27,	14,	160,	34,	1313,	65,	23,	79,	54,	48
33,	19,	1,	15,	12,	1821,	0,	14,	1,	27
76,	11,	20,	41,	3,	0,	1862,	3,	96,	28
178,	25,	75,	28,	64,	22,	12,	1507,	57,	23
19,	7,	39,	118,	5,	1,	132,	12,	1581,	19
2,	6,	5,	11,	19,	22,	4,	27,	1,	1902

Tabela 1: Matriz de confusão do classificador linear.

Cada linha da matriz representa, respectivamente, de 1 a 10, cada classe pretendida pelo treinamento (onde a classe 10 representa o dígito zero) e, nas colunas, quantas vezes cada classe, também de 1 a 10, foi escolhida pelo classificador quando este deveria escolher aquela da referida linha.

Nota-se que a classe reconhecida com maior precisão pelo sistema após o treinamento foi o dígito 1, sendo que a classe que mais gerou equívocos foi a do dígito 5, muitas vezes confundida com o dígito 3.

Nas figuras 1 e 2, temos exemplos de dígitos que foram reconhecidos erroneamente pelo classificador. Na figura 1, percebe-se que a curva superior distorcida e a curva inferior exageradamente grande fizeram o dígito 3 ficar parecido com o 5, fazendo o classificador se confundir. Para a figura 2, a parte superior do 5 foi feita de maneira muito discreta, fazendo o classificador classificá-lo como um 3.

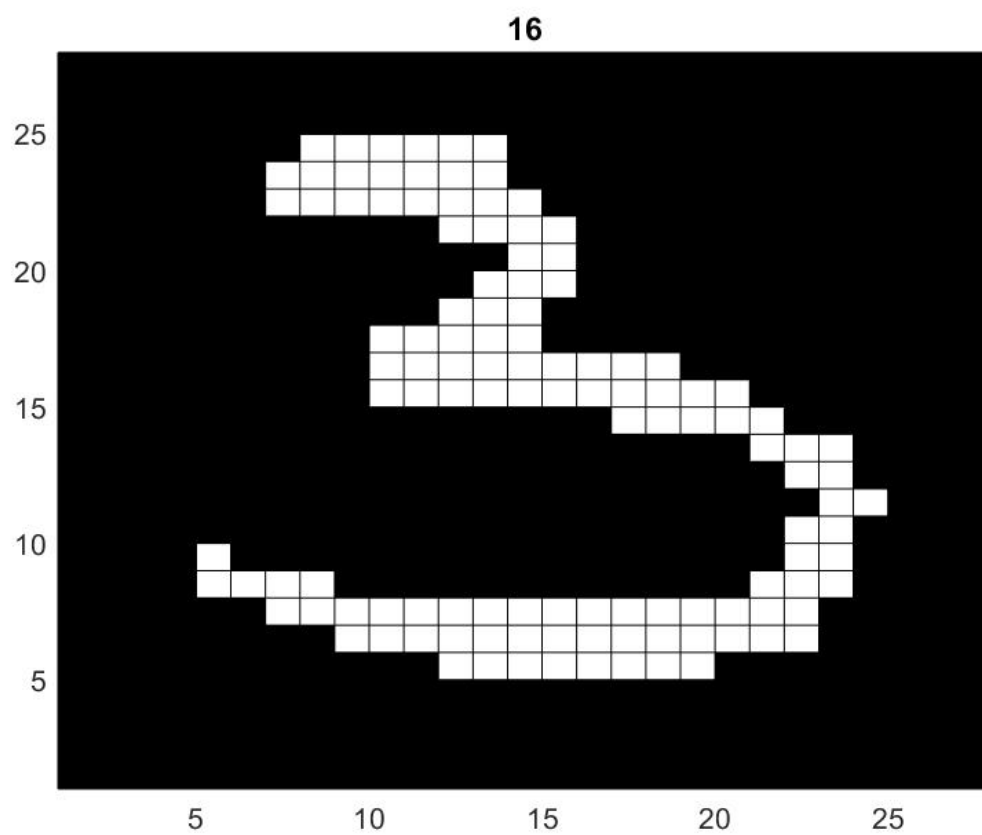


Figura 1: Um dígito três que foi reconhecido como cinco.

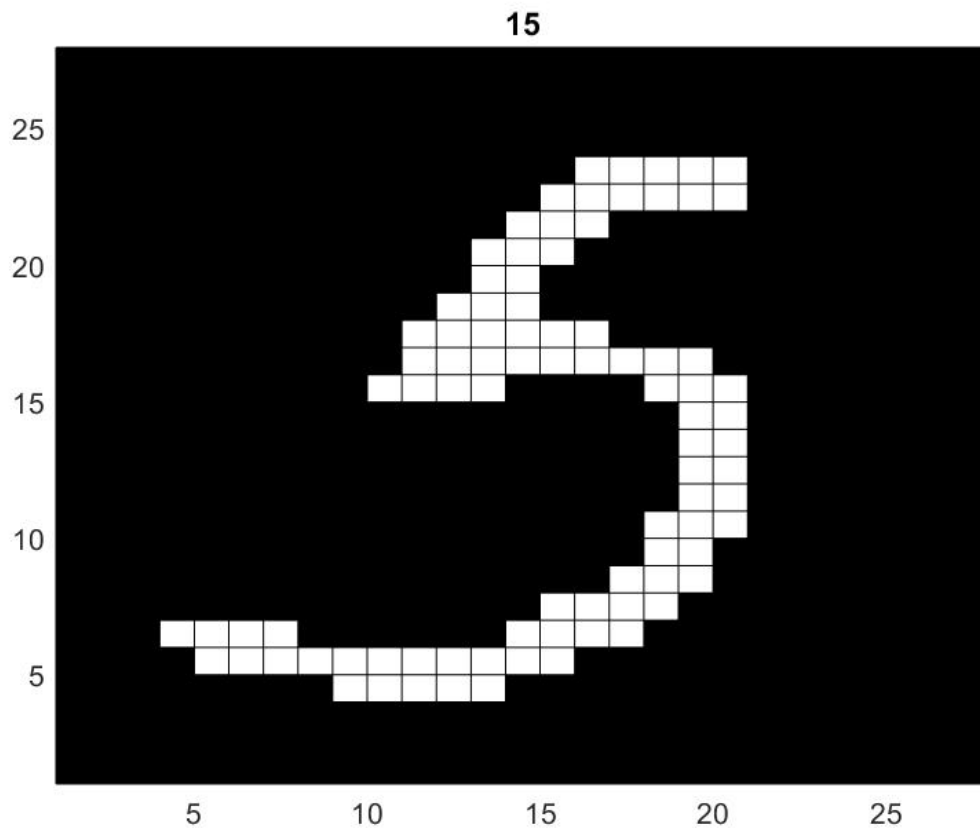


Figura 2: Um dígito cinco que foi reconhecido como três.

Gráficos de busca do coeficiente de regularização no anexo 1.

Gráficos de calor da matriz W no anexo 2.

Questão 2

Coeficiente de regularização utilizado (taxa de acertos):

64.000000000000000000000000000000

Coeficiente de regularização para erro quadrático: 24.25146506416636427161392930

Q2.1)

A matriz confusão obtida pela ELM na questão 2 está expressa abaixo:

2140,	11,	3,	3,	7,	6,	1,	5,	2,	1
18,	1824,	28,	21,	6,	27,	29,	32,	7,	23
11,	52,	1819,	2,	44,	17,	22,	49,	25,	9
20,	7,	0,	1780,	5,	14,	5,	8,	94,	0
19,	11,	76,	11,	1575,	49,	12,	14,	25,	25
4,	14,	0,	14,	17,	1860,	1,	12,	2,	19
34,	26,	6,	31,	6,	1,	1962,	3,	54,	17
45,	24,	58,	11,	47,	17,	17,	1714,	37,	21
9,	8,	26,	63,	12,	4,	63,	24,	1712,	12
1,	11,	3,	7,	11,	16,	1,	19,	3,	1927

Tabela 2: Matriz de confusão da ELM.

A matriz de confusão da ELM segue a mesma distribuição da questão 1. Nota-se que a taxa de acertos em todas as classes aumentou, mas o dígito mais bem reconhecido continua sendo o 1, provavelmente por seu formato simples e singular, enquanto que o dígito 5 continua sendo o mais complexo para a máquina, frequentemente confundido com o 3, provavelmente por seu traçado passar por muitos pontos em comum.

Na figura 3, foi reconstruído um dígito 5 que foi reconhecido como 3, provavelmente por seu formato distorcido. Na figura 4, um dígito 3 reconhecido como 5, provavelmente por suas curvas acentuadas e distorcidas.

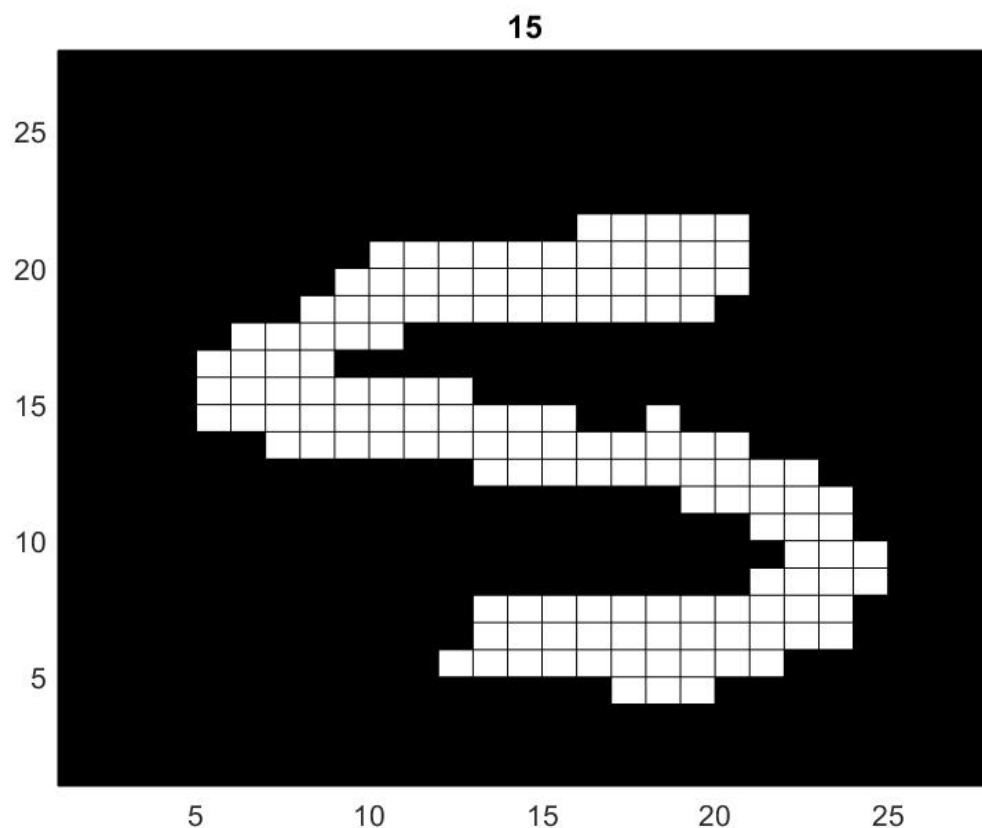


Figura 3: Um dígito cinco que foi reconhecido como três.

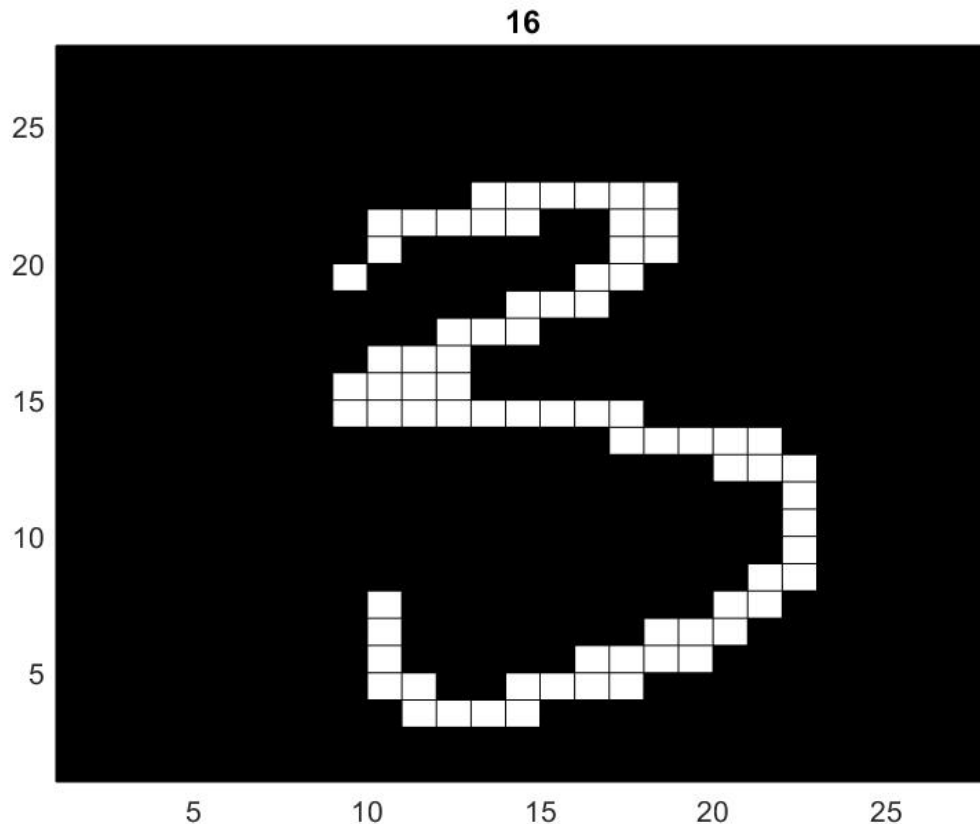


Figura 4: Um dígito três que foi reconhecido como cinco.

Q2.2) O ganho de desempenho se deve ao mapeamento com a função de tangente hiperbólica, que permite a saturação da saída do neurônio e evita que poucas entradas bem alinhadas, porém de magnitude alta, induzam um resultado incorreto. Assim, ficam sendo mais influentes as entradas com muitos pontos coincidentes aos maiores valores dos classificadores, em detrimento daquelas que possuem poucos pontos de grande valor sendo multiplicadas pelos elementos do respectivo classificador.

A diminuição do tempo de execução se deve ao fato de que a matriz de entrada e a matriz W reduziram de 784 para 500 (sem considerar o offset) seu número de colunas e linhas, respectivamente. Com isso, a quantidade de operações diminui significativamente, pois a quantidade de elementos a serem operados nestas matrizes é grande e seu número foi reduzido em pouco mais de 1 terço.

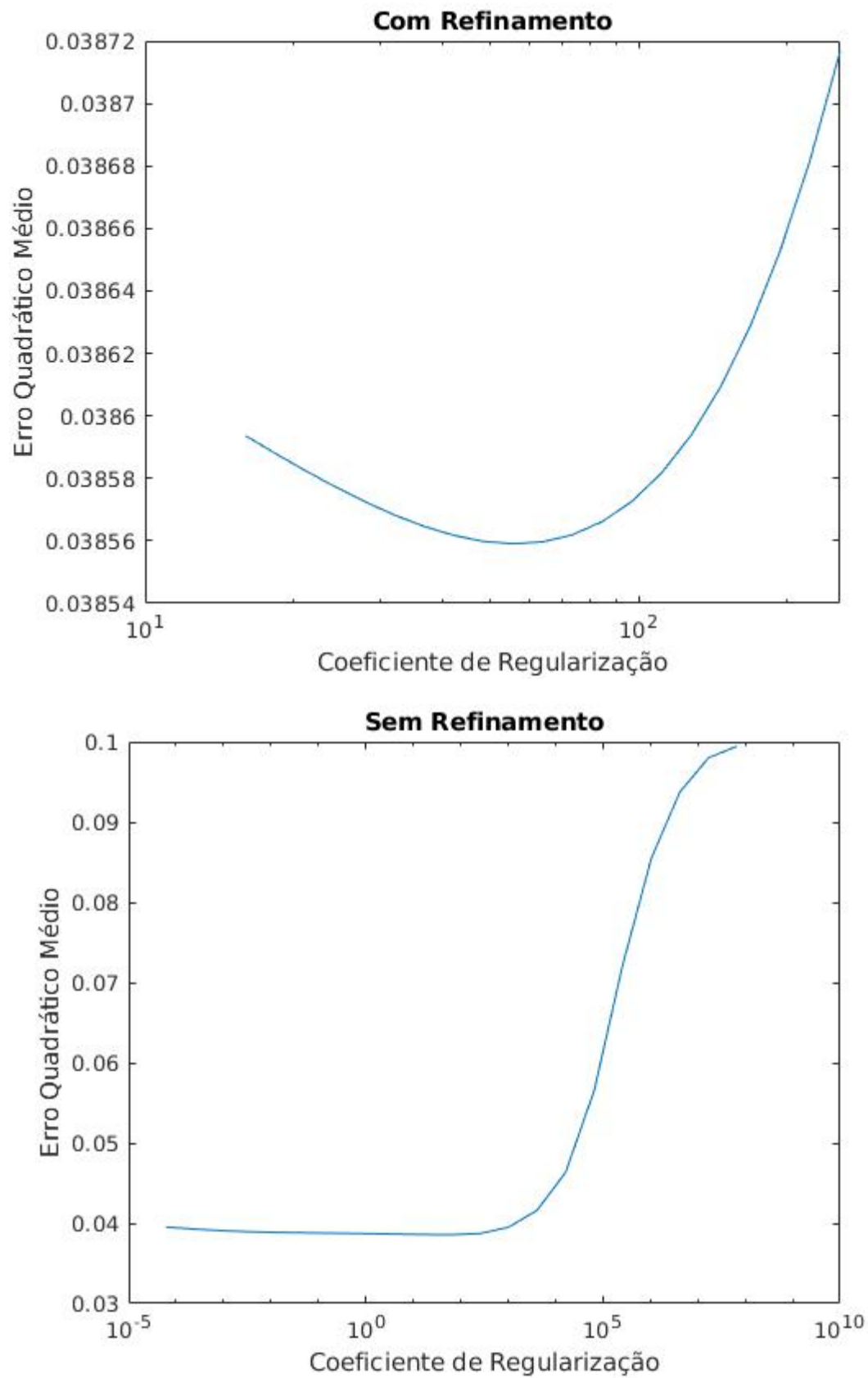
Q2.3) Como a matriz H é produzida através da multiplicação entre a entrada X e a matriz V de números normalmente distribuídos, o posto de H é maior que o de X , o que faz com que a necessidade de regularização no caso da ELM seja menor. Como calculamos a regularização de forma a penalizar o crescimento dos vetores de W , o coeficiente de regularização será o menor possível para realizar a tarefa. Portanto, o coeficiente de regularização utilizado no classificador linear (675.588) foi maior do que para a ELM (64.000).

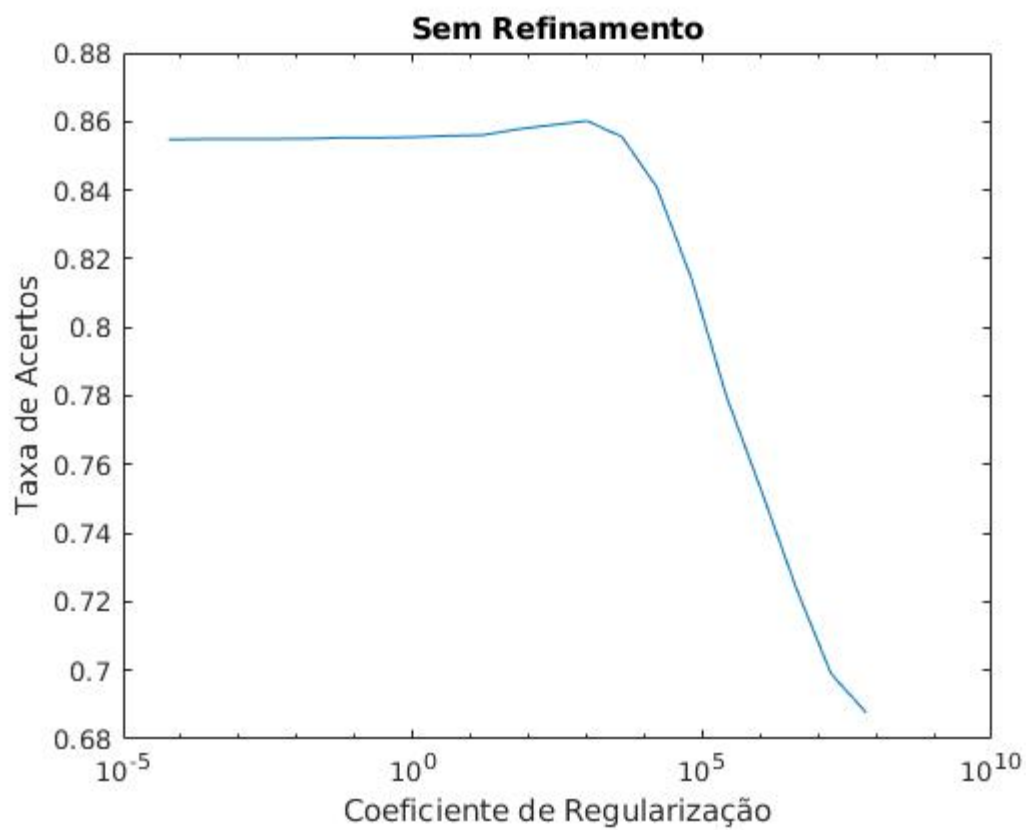
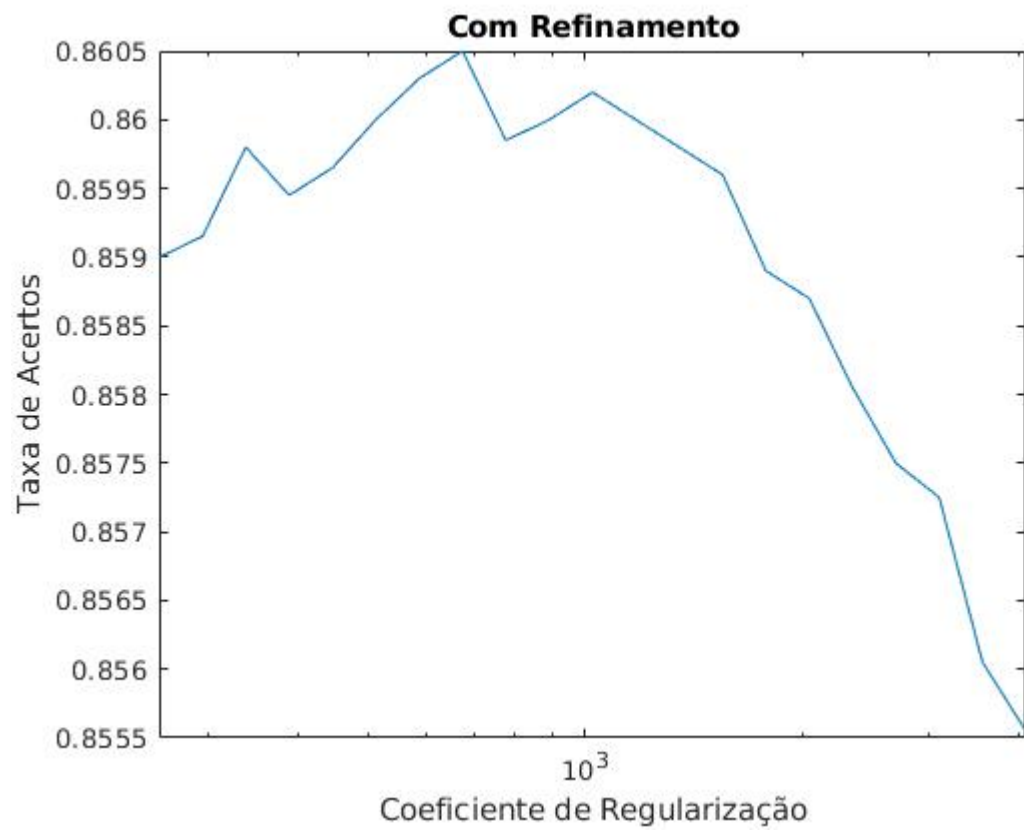
Q2.4) Como a matriz de entrada será diferente para cada caso dos pesos aleatórios, o

coeficiente de regularização pode ser alterado, para mais ou para menos, dentro de uma faixa de valores que é relativa ao desvio padrão escolhido para a distribuição normal dos pesos aleatórios.

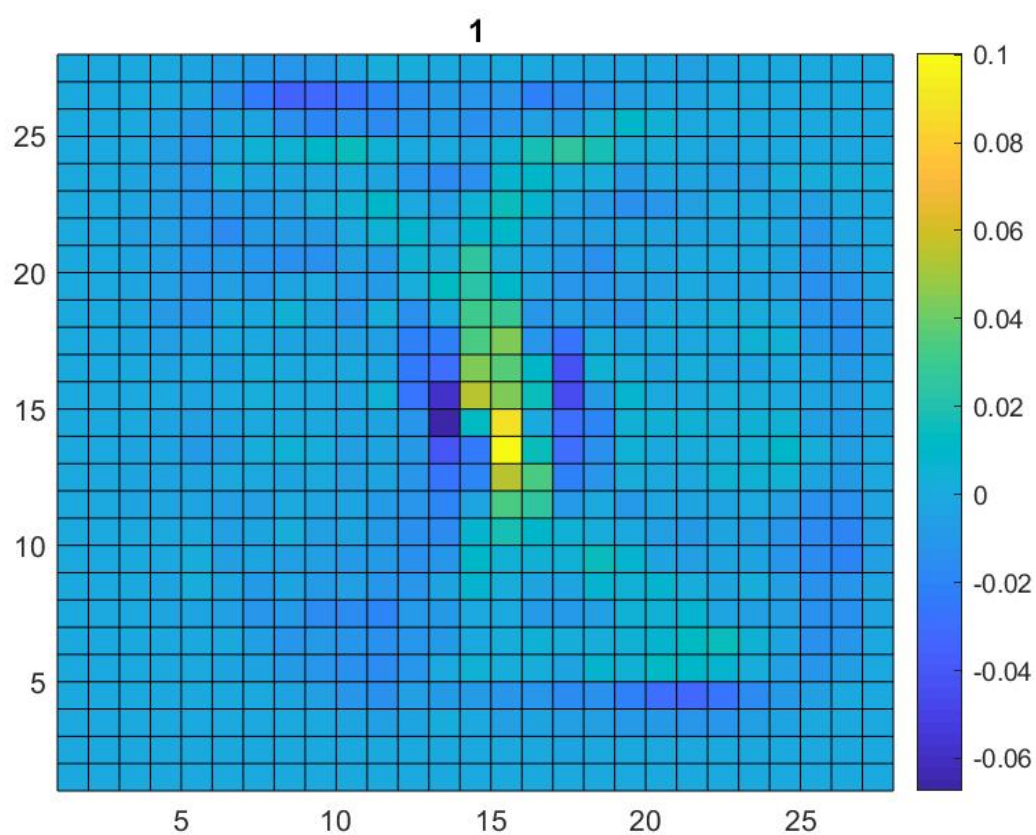
Gráficos de busca do coeficiente de regularização no anexo 3.

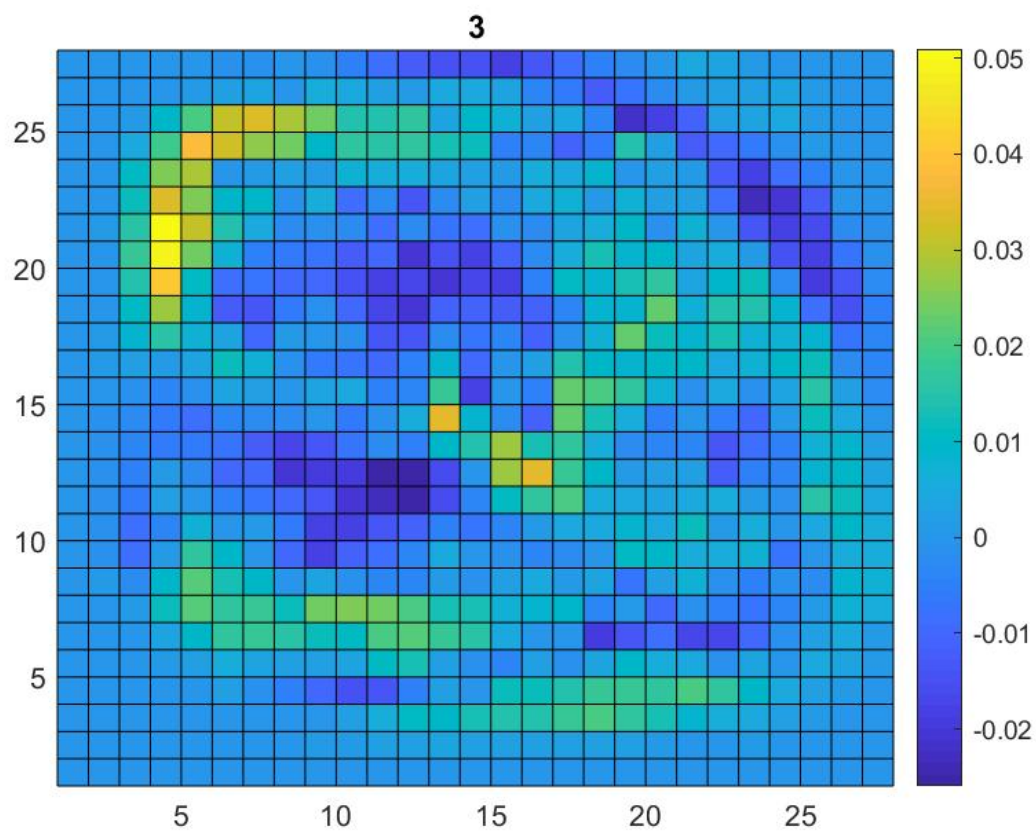
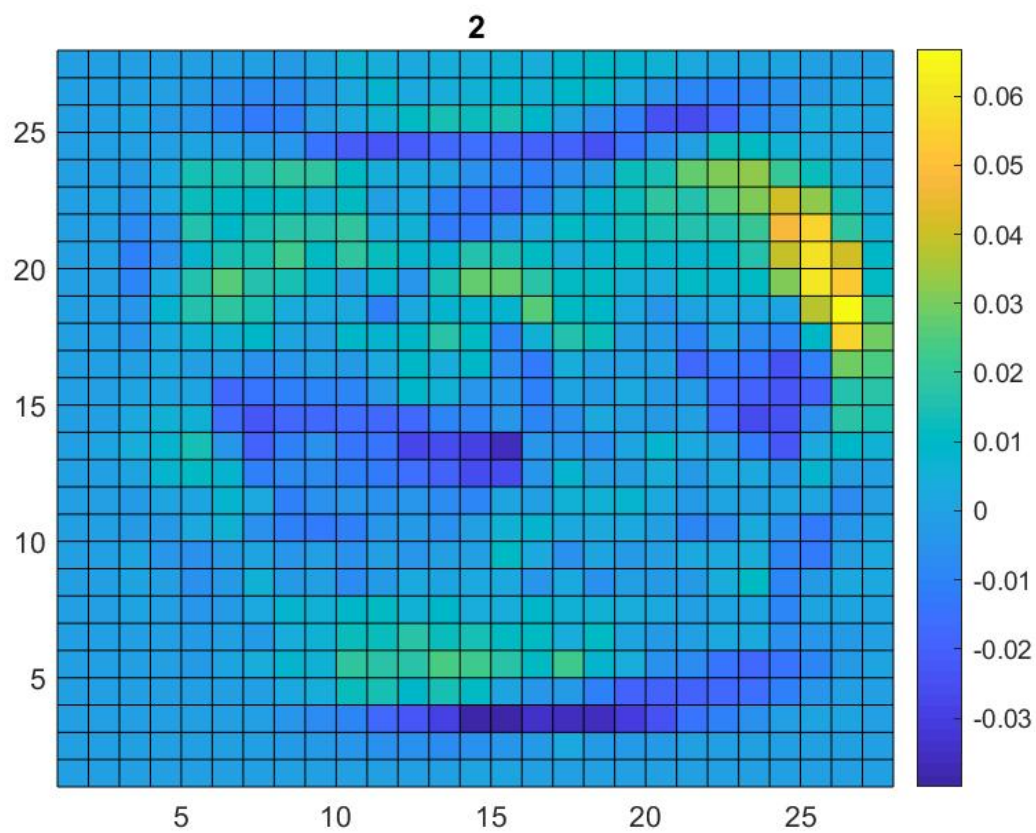
Anexo 1 - Busca pelo coeficiente de regularização da atividade 1

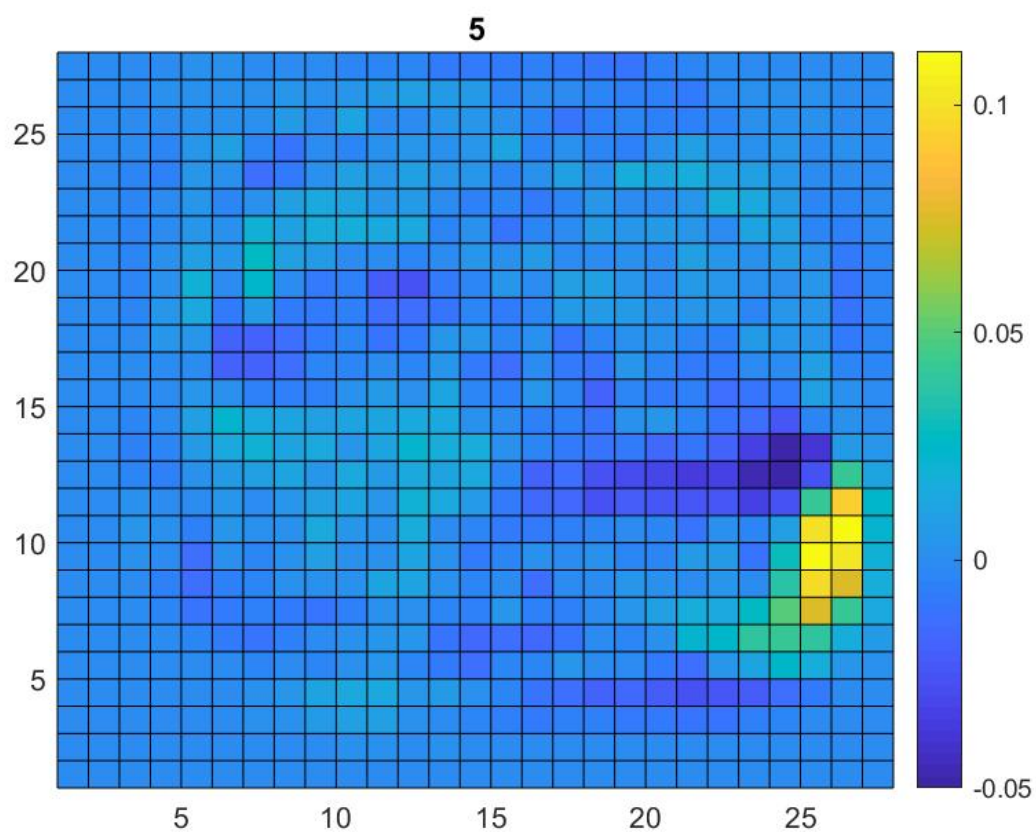
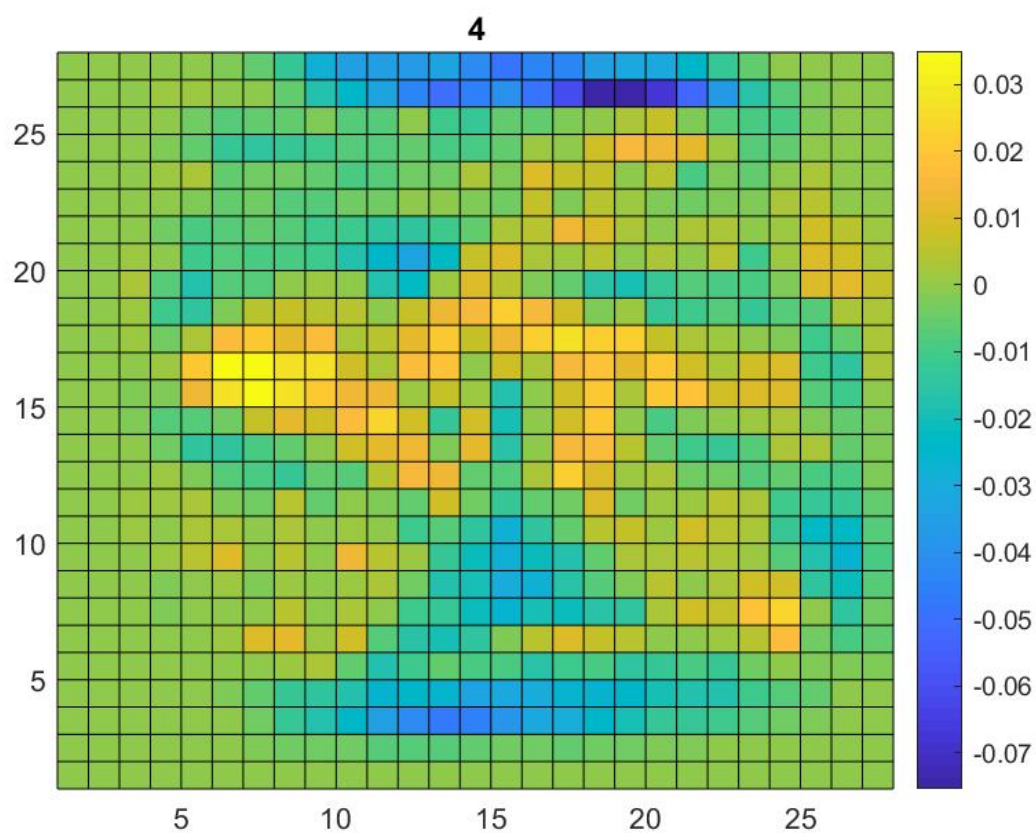


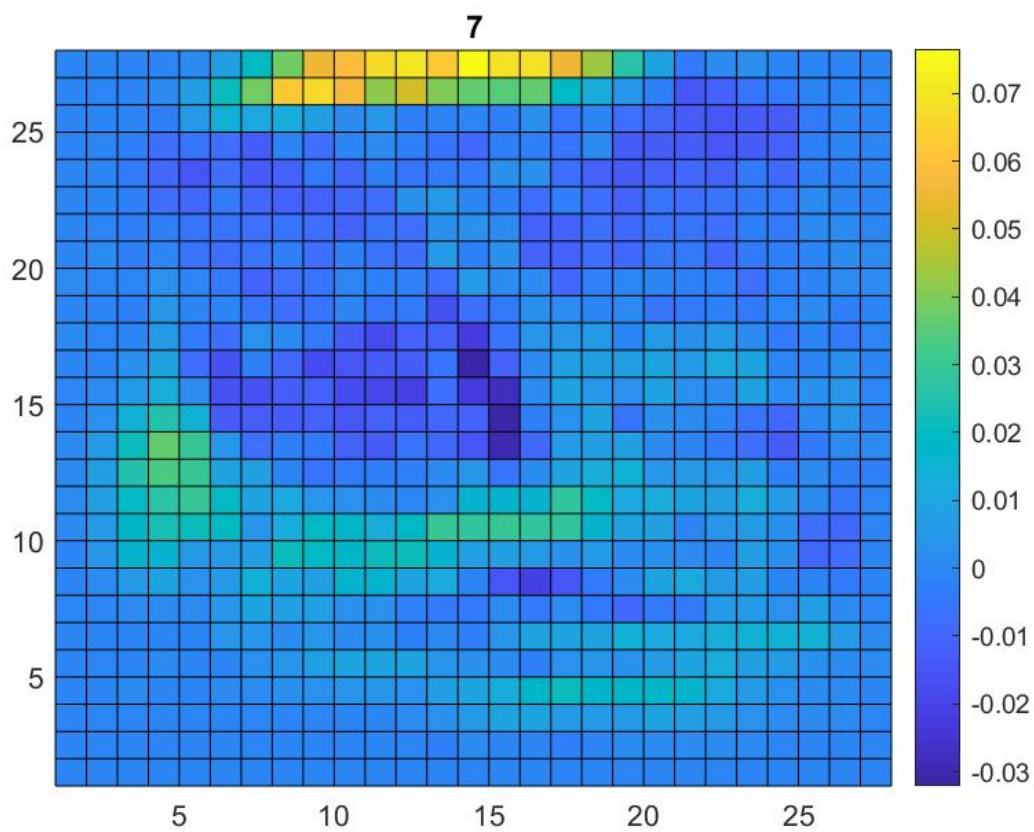
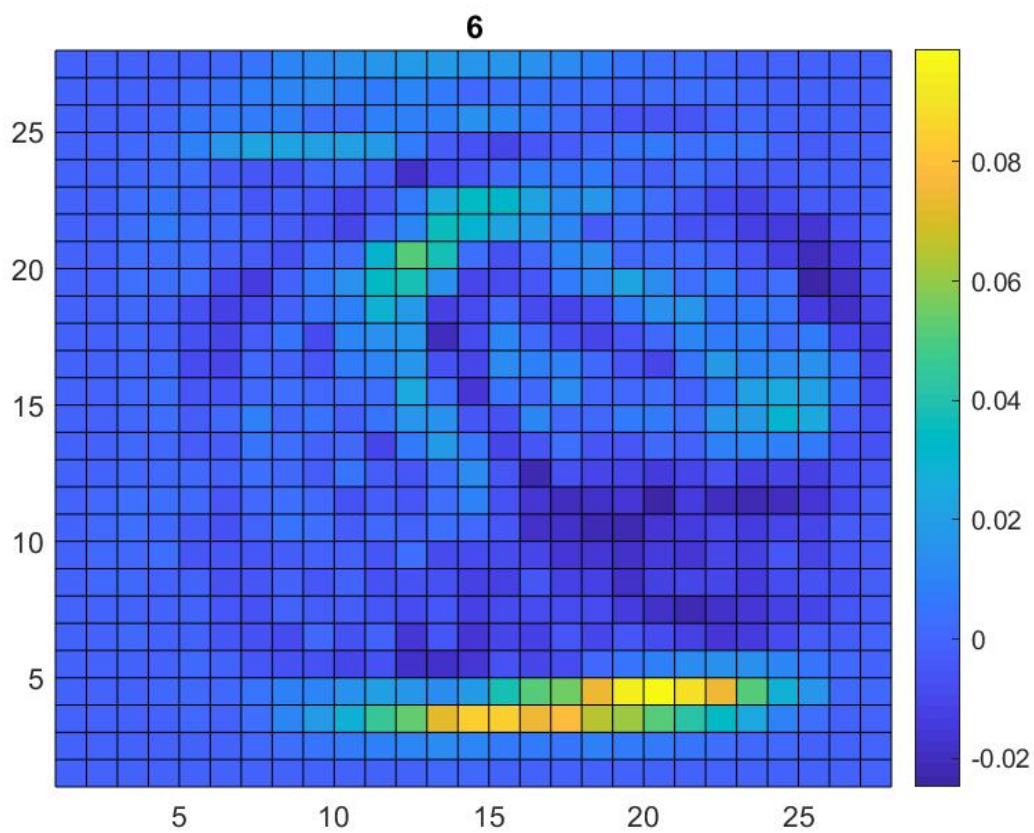


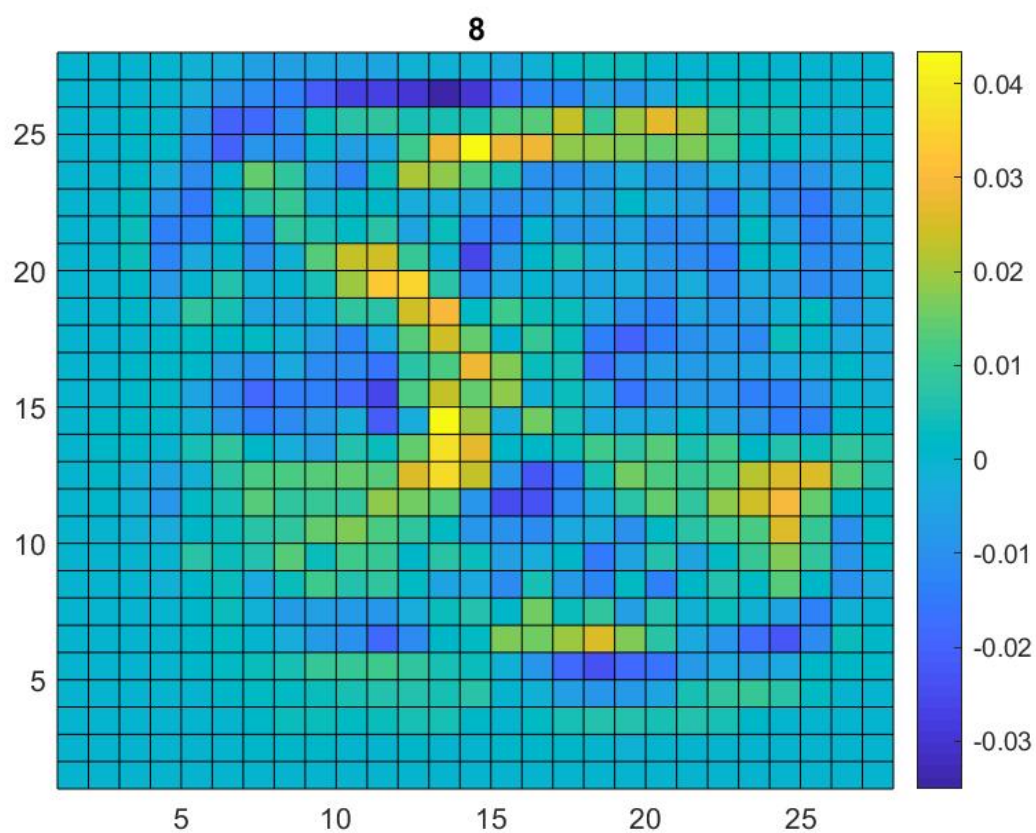
Anexo 2 - Gráficos de calor da atividade 1

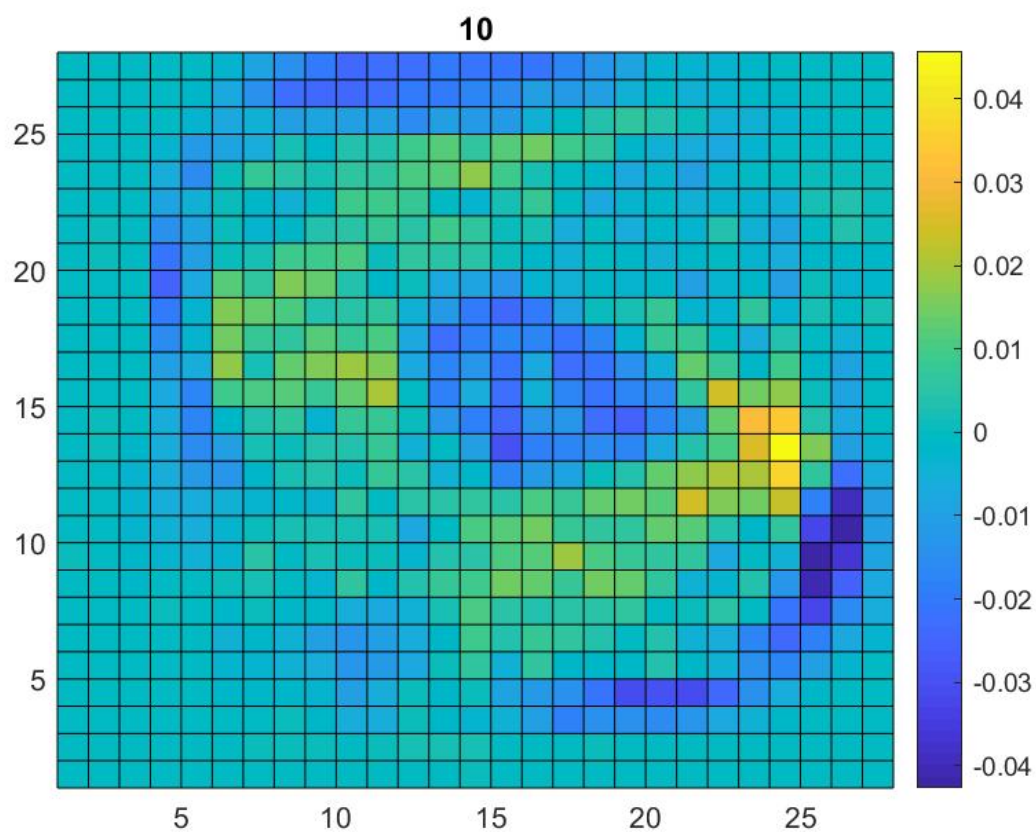
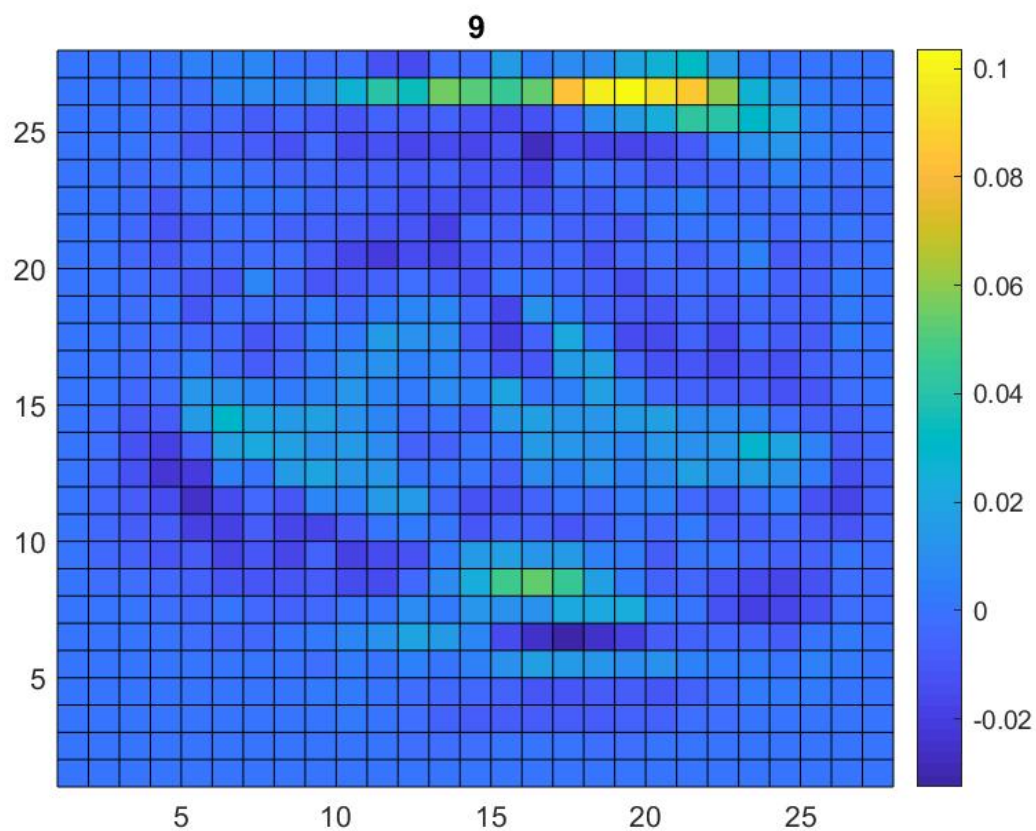




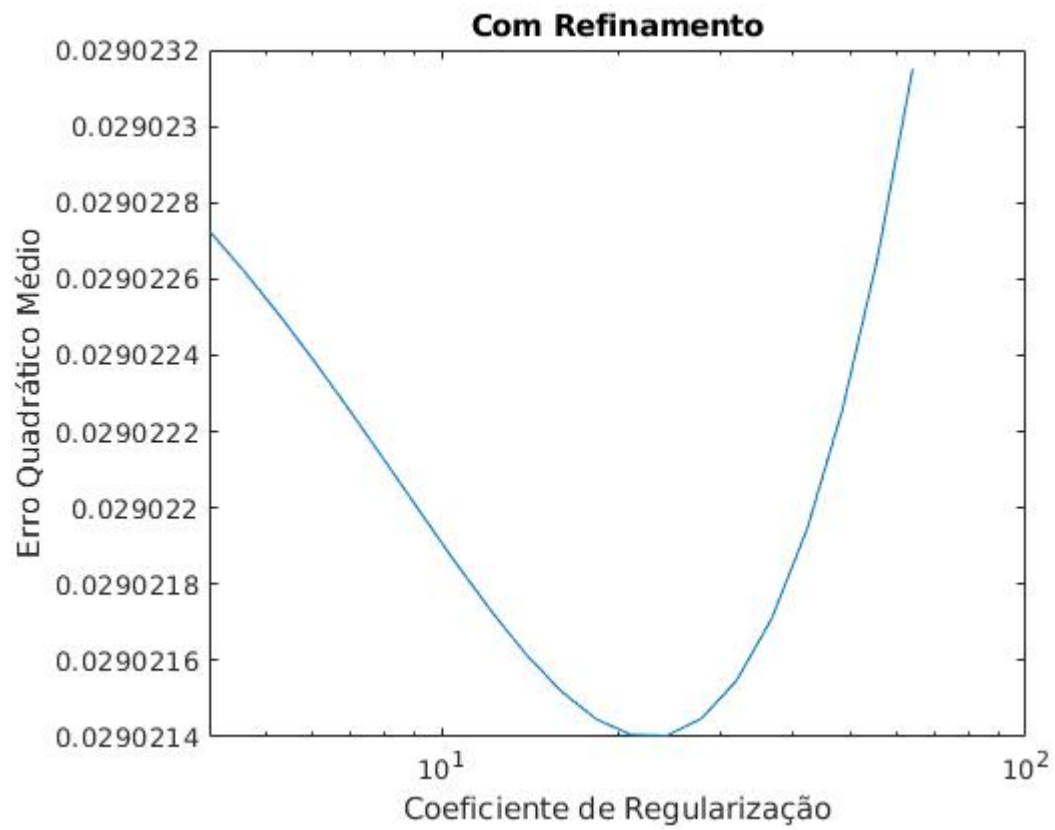


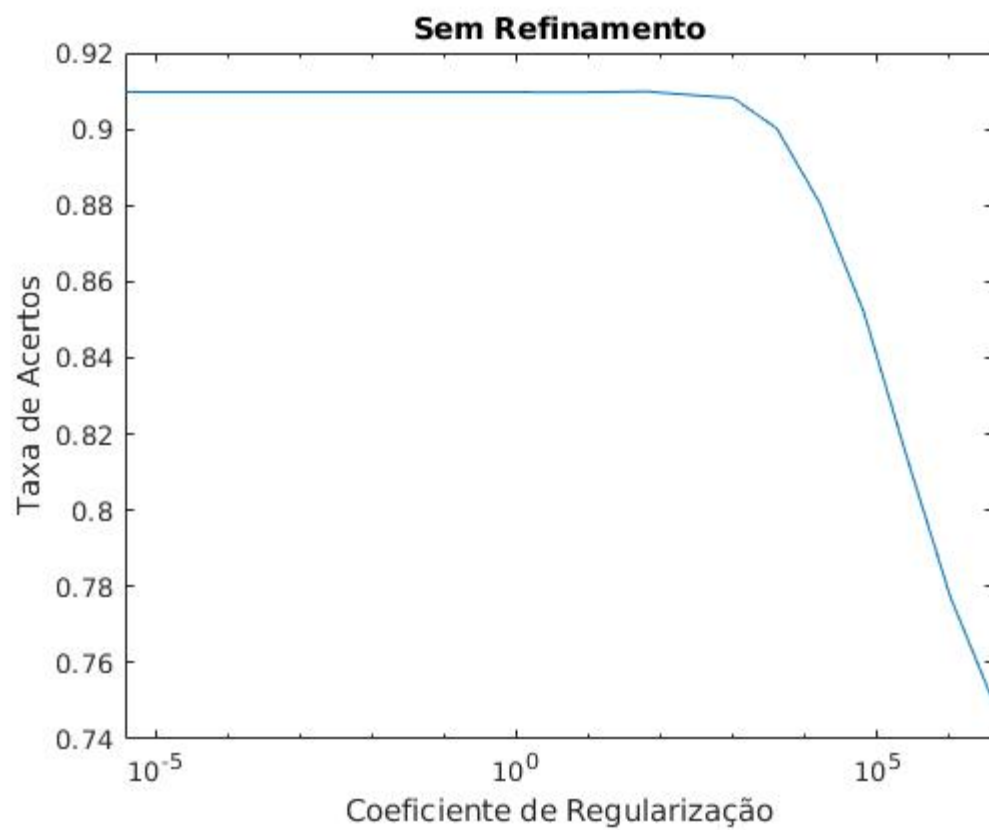
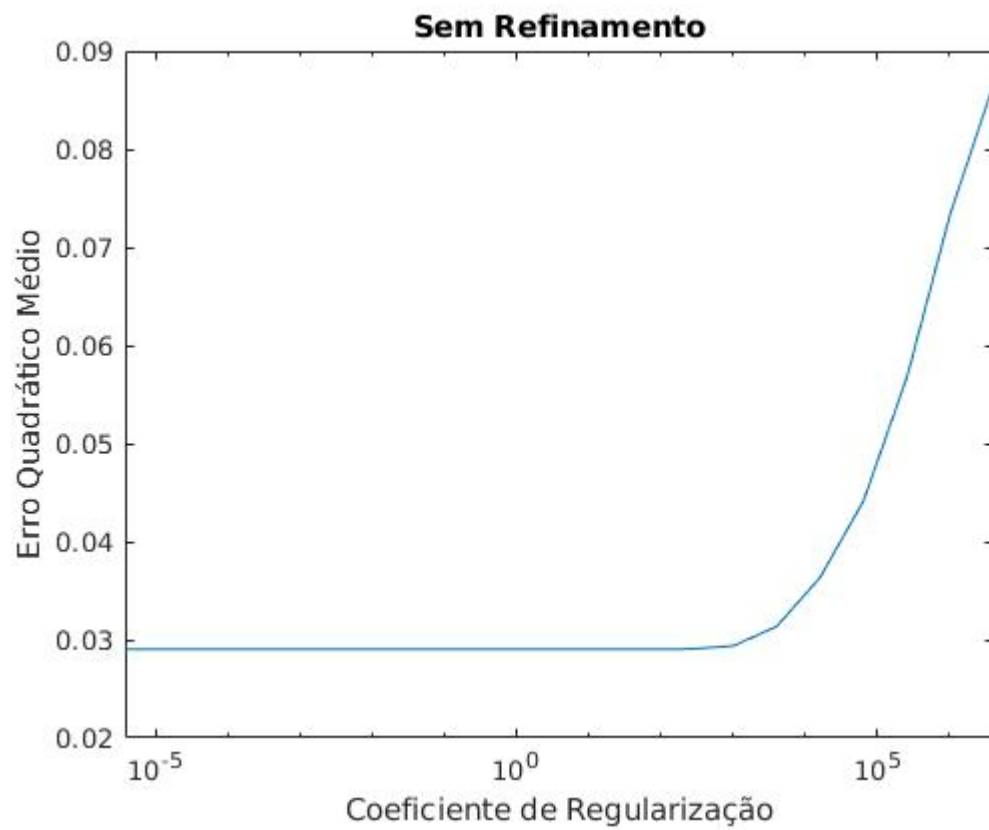


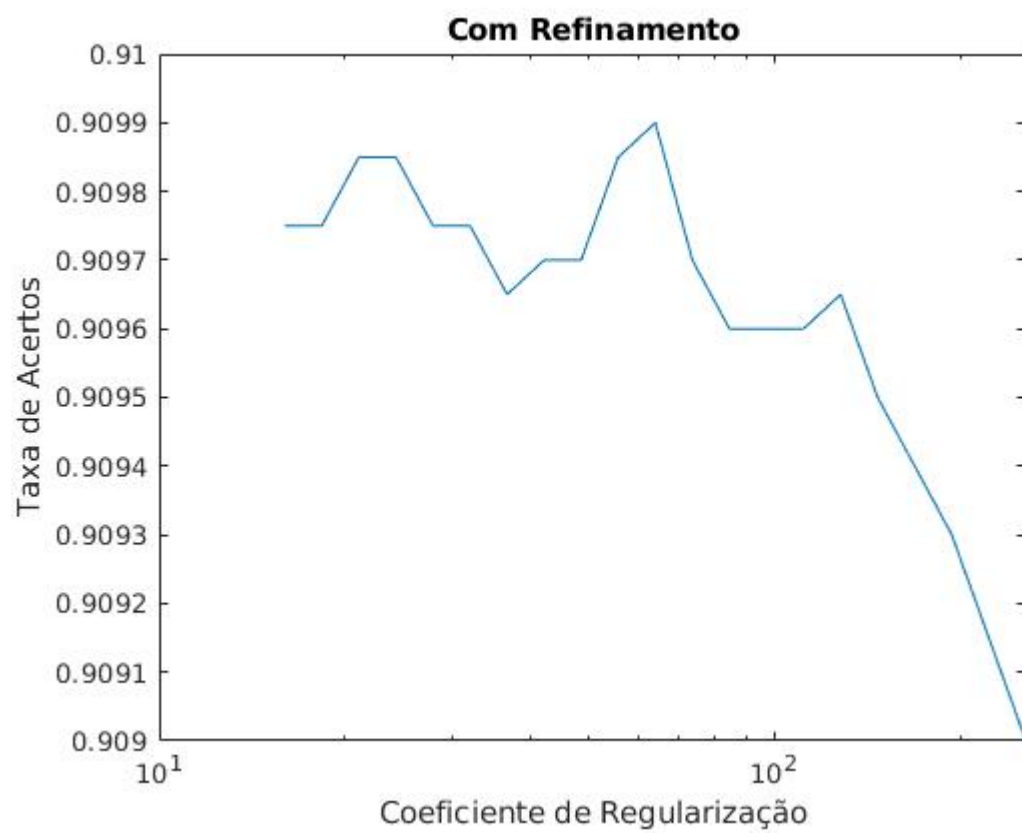




Anexo 3 - Busca pelo coeficiente de regularização da atividade 2







Patrick de Carvalho Tavares Rezende Ferreira - RA: 175480 - EFC1

Repositório: <https://github.com/patrickctrf/EA072-Inteligencia-Artificial-IA/tree/master/EFC1>
(<https://github.com/patrickctrf/EA072-Inteligencia-Artificial-IA/tree/master/EFC1>).

In [0]:

```
%cd /content/drive/My\ Drive/P0DE\ APAGAR/EA072-EF1  
  
/content/drive/My Drive/P0DE APAGAR/EA072-EF1
```

Questão 3

Inicialmente, executou-se 5 vezes o código sugerido inicial a fim de verificar o desempenho da proposta. Seu desempenho foi de:

- Loss: 0.0733; Acurácia: 0.9775.

Utilizando-se o método de tentativa e erro, foi criado um script que verificava o desempenho da rede para diferentes parâmetros alterados como dropout (0.1 a 0.6), número de camadas (1 a 4 intermediárias), épocas de treinamento (4 a 8) e número de neurônios por camada (256 a 512). O script executava esta mudança de parâmetros dentro de loops "for" para executar todas as combinações possíveis e tirava também a média das múltiplas execuções com mesmos parâmetros, a fim de se obter uma média de desempenho mais confiável. Os resultados desta varredura eram salvos ao final das execuções em um arquivo "listas.txt", permitindo ao usuários verificar qual a configuração obteve melhor desempenho. Foram utilizadas 4 threads - para varredura de redes de 1 a 4 camadas - durante o treinamento, a fim de promover paralelismo e diminuir o tempo requerido, que chegava a dezenas de horas.

Analisando as configurações que obtiveram o melhor desempenho, pode-se notar que as características que o maximizavam eram: 2 camadas, 512 neurônios, taxa de dropout próxima de 0.4 e 8 épocas de treinamento.

Portanto, para a proposta final deste modelo, foi executada mais um treinamento com uso dos atributos analisados acima e os parâmetros que resultaram no melhor desempenho durante a varredura foram:

- Camadas: 2; Neurônios por camada: 512; Dropout: 0.4; Épocas: 8.

O desempenho médio obtido foi de:

- Loss: 0.0663; Acurácia: 0.9823.

Ambas as soluções consumiram um tempo de execução da ordem de poucos minutos e a diferença de desempenho foi cerca de 0,5% em ganho.

Os arquivos utilizados foram (no diretório q3):

Proposta Inicial: q3Inicial.py

Script de Varredura de parâmetros: q3.py

Proposta Final: q3Final.py

In [0]:

```
# q3Inicial.py

import tensorflow as tf
import os
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
model_json = model.to_json()
json_file = open("model_MLP.json", "w")
json_file.write(model_json)
json_file.close()
model.save_weights("model_MLP.h5")
print("Model saved to disk")
os.getcwd()
```

```
Epoch 1/5
60000/60000 [=====] - 6s 103us/sample - los
s: 0.2698 - acc: 0.9197
Epoch 2/5
60000/60000 [=====] - 6s 101us/sample - los
s: 0.1385 - acc: 0.9579
Epoch 3/5
60000/60000 [=====] - 6s 103us/sample - los
s: 0.1080 - acc: 0.9668
Epoch 4/5
60000/60000 [=====] - 6s 101us/sample - los
s: 0.0928 - acc: 0.9710
Epoch 5/5
60000/60000 [=====] - 6s 101us/sample - los
s: 0.0821 - acc: 0.9736
10000/10000 [=====] - 1s 68us/sample - los
s: 0.0640 - acc: 0.9797
Model saved to disk
```

Out[0]:

```
'/content/drive/My Drive/PODE APAGAR/EA072-EF1'
```

In [0]:

```
#q3.py

import tensorflow as tf
import os
import threading

myMutex = threading.Lock()
value = "teste"

numeroDeNeuronios = []
numeroDeEpocas = []
numeroDeCamadas = []
numeroDeDropout = []
taxaDeAcertos = []

# Vamos colocar uma thread para treinar cada rede com um numero especifico de camadas.
def thread1Camadas(camadas):

    # Para tirar a media das iteracoes, somaremos todas aqui e dividiremos pelo total.
    somaDasEficienciasDeCadaIteracao = 0

    # Os valores que utilizaremos para dropout variarao de 10% a 90% (instrucao abaixo).
    valoresDropout = range(10, 60, 10)# Variaremos de 10% em 10%.
    valoresDropout = [i/100 for i in valoresDropout]# Converte de porcentagem para escala de 0 a 1.

    # Testando resultados com diferentes quantidades de epocas.
    for epocas in [8, 4]:

        # Testando resultados com diferentes quantidades de neuronios.
        for neuronios in [256, 512]:

            # So para indicar em que passo da execucao estamos.
            print("\n\nepocas: " + str(epocas) + "\nCAMADAS" + str(camadas) + ": " + str(neuronios) + "\n\n")

            # Este loop fica responsável por treinar com diferentes taxas de dropout.
            # "i" eh o valor a cada iteracao.
            for taxaDropout in valoresDropout:

                # Repetimos o treinamento algumas vezes para tirar uma media da eficiencia
                for iteracaoMedia in range(1,4):
                    mnist = tf.keras.datasets.mnist
                    (x_train, y_train),(x_test, y_test) = mnist.load_data()
                    x_train, x_test = x_train / 255.0, x_test / 255.0
                    model = tf.keras.models.Sequential([
                        tf.keras.layers.Flatten(),
                        tf.keras.layers.Dense(neuronios, activation=tf.nn.relu),
                        tf.keras.layers.Dropout(taxaDropout),#
                        tf.keras.layers.Dense(10, activation=tf
```

```

.nn.softmax)

    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    model.fit(x_train, y_train, epochs=epoca
s)

    value = model.evaluate(x_test, y_test)
    model_json = model.to_json()
    json_file = open("model_MLP1.json", "w")
    json_file.write(model_json)
    json_file.close()
    model.save_weights("model_MLP1.h5")
    print("Model saved to disk")
    os.getcwd()

    somaDasEficienciasDeCadaIteracao = value
[1] + somaDasEficienciasDeCadaIteracao

    myMutex.acquire()
    numeroDeNeuronios.append(neuronios)
    numeroDeEpocas.append(epocas)
    numeroDeCamadas.append(camadas)
    numeroDeDropout.append(taxaDropout)
    taxaDeAcertos.append(somaDasEficienciasDeCadaIte
racao/iteracaoMedia)

    myMutex.release()

    # Reiniciamos a soma.
    somaDasEficienciasDeCadaIteracao = 0

# Vamos colocar uma thread para treinar cada rede com um numero especifico de ca
madas.
def thread2Camadas(camadas):

    # Para tirar a media das iteracoes, somaremos todas aqui e dividiremos p
elo total.
    somaDasEficienciasDeCadaIteracao = 0

    # Os valores que utilizaremos para dropout variarao de 10% a 90% (instru
cao abaixo).
    valoresDropout = range(10, 60, 10)# Variaremos de 10% em 10%.
    valoresDropout = [i/100 for i in valoresDropout]# Converte de percentage
m para escala de 0 a 1.

    # Testando resultados com diferentes quantidades de epocas.
    for epocas in [8, 4]:

        # Testando resultados com diferentes quantidades de neuronios.
        for neuronios in [256, 512]:

            # So para indicar em que passo da execucao estamos.
            print("\n\nepocas: " + str(epocas) + "\nCAMADAS" + str(c
amadas) + ": " + str(neuronios) + "\n\n")

            # Este loop fica responsável por treinar com diferentes
taxas de dropout.

            # "i" eh o valor a cada iteracao.
            for taxaDropout in valoresDropout:

```

```

# Repetimos o treinamento algumas vezes para tirar uma media da eficiencia
for iteracaoMedia in range(1,4):
    mnist = tf.keras.datasets.mnist
    (x_train, y_train),(x_test, y_test) = mnist.load_data()
    x_train, x_test = x_train / 255.0, x_test / 255.0

    model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(neuronios, activation=tf.nn.relu),
        tf.keras.layers.Dropout(taxaDropout),#Diferentes valores de dropout.
        tf.keras.layers.Dense(neuronios, activation=tf.nn.relu),
        tf.keras.layers.Dropout(taxaDropout),#Diferentes valores de dropout.
        tf.keras.layers.Dense(10, activation=tf.nn.softmax)
    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    model.fit(x_train, y_train, epochs=epochs)

    value = model.evaluate(x_test, y_test)
    model_json = model.to_json()
    json_file = open("model_MLP2.json", "w")
    json_file.write(model_json)
    json_file.close()
    model.save_weights("model_MLP2.h5")
    print("Model saved to disk")
    os.getcwd()

    somaDasEficienciasDeCadaIteracao = value

[1] + somaDasEficienciasDeCadaIteracao

myMutex.acquire()
numeroDeNeuronios.append(neuronios)
numeroDeEpocas.append(epocas)
numeroDeCamadas.append(camadas)
numeroDeDropout.append(taxaDropout)
taxaDeAcertos.append(somaDasEficienciasDeCadaIteracao/iteracaoMedia)

myMutex.release()

# Reiniciamos a soma.
somaDasEficienciasDeCadaIteracao = 0

# Vamos colocar uma thread para treinar cada rede com um numero especifico de camadas.
def thread3Camadas(camadas):

    # Para tirar a media das iteracoes, somaremos todas aqui e dividiremos pelo total.
    somaDasEficienciasDeCadaIteracao = 0

```

```

# Os valores que utilizaremos para dropout variarao de 10% a 90% (instrucao abaixo).
valoresDropout = range(10, 60, 10)# Variaremos de 10% em 10%.
valoresDropout = [i/100 for i in valoresDropout]# Converte de porcentagem para escala de 0 a 1.

# Testando resultados com diferentes quantidades de epocas.
for epocas in [8, 4]:

    # Testando resultados com diferentes quantidades de neuronios.
    for neuronios in [256, 512]:

        # So para indicar em que passo da execucao estamos.
        print("\n\nepocas: " + str(epocas) + "\nCAMADAS" + str(camadas) + ": " + str(neuronios) + "\n\n")

        # Este loop fica responsável por treinar com diferentes taxas de dropout.
        # "i" eh o valor a cada iteracao.
        for taxaDropout in valoresDropout:

            # Repetimos o treinamento algumas vezes para tirar uma media da eficiencia
            for iteracaoMedia in range(1,4):
                mnist = tf.keras.datasets.mnist
                (x_train, y_train),(x_test, y_test) = mnist.load_data()
                x_train, x_test = x_train / 255.0, x_test / 255.0

                model = tf.keras.models.Sequential([
                    tf.keras.layers.Flatten(),
                    tf.keras.layers.Dense(neuronios, activation=tf.nn.relu),
                    tf.keras.layers.Dropout(taxaDropout),# Diferentes valores de dropout.
                    tf.keras.layers.Dense(neuronios, activation=tf.nn.relu),
                    tf.keras.layers.Dropout(taxaDropout),# Diferentes valores de dropout.
                    tf.keras.layers.Dense(neuronios, activation=tf.nn.relu),
                    tf.keras.layers.Dropout(taxaDropout),# Diferentes valores de dropout.
                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)

                ])
                model.compile(optimizer='adam',
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])
                model.fit(x_train, y_train, epochs=epocas)

                value = model.evaluate(x_test, y_test)
                model_json = model.to_json()
                json_file = open("model_MLP3.json", "w")
                json_file.write(model_json)
                json_file.close()
                model.save_weights("model_MLP3.h5")
                print("Model saved to disk")
                os.getcwd()

            somaDasEficienciasDeCadaIteracao = value

```

```
[1] + somaDasEficienciasDeCadaIteracao
```

```

myMutex.acquire()
numeroDeNeuronios.append(neuronios)
numeroDeEpocas.append(epocas)
numeroDeCamadas.append(camadas)
numeroDeDropout.append(taxaDropout)
taxaDeAcertos.append(somaDasEficienciasDeCadaIte
racao/iteracaoMedia)

myMutex.release()

# Reiniciamos a soma.
somaDasEficienciasDeCadaIteracao = 0

# Vamos colocar uma thread para treinar cada rede com um numero especifico de ca
madras.
def thread4Camadas(camadas):

    # Para tirar a media das iteracoes, somaremos todas aqui e dividiremos p
elo total.
    somaDasEficienciasDeCadaIteracao = 0

    # Os valores que utilizaremos para dropout variarao de 10% a 90% (instru
cao abaixo).
    valoresDropout = range(10, 60, 10)# Variaremos de 10% em 10%.
    valoresDropout = [i/100 for i in valoresDropout]# Converte de porcentagem
para escala de 0 a 1.

    # Testando resultados com diferentes quantidades de epocas.
    for epocas in [8, 4]:

        # Testando resultados com diferentes quantidades de neuronios.
        for neuronios in [256, 512]:

            # So para indicar em que passo da execucao estamos.
            print("\n\nepocas: " + str(epocas) + "\n\nCAMADAS" + str(c
amadas) + ": " + str(neuronios) + "\n\n")

            # Este loop fica responsável por treinar com diferentes
taxas de dropout.

            # "i" eh o valor a cada iteracao.
            for taxaDropout in valoresDropout:

                # Repetimos o treinamento algumas vezes para tir
ar uma media da eficiencia

                for iteracaoMedia in range(1,4):
                    mnist = tf.keras.datasets.mnist
                    (x_train, y_train),(x_test, y_test) = mn
ist.load_data()
                    x_train, x_test = x_train / 255.0, x_tes
t / 255.0

                    model = tf.keras.models.Sequential([
                        tf.keras.layers.Flatten(),
                        tf.keras.layers.Dense(neuronios, activa
tion=tf.nn.relu),
                        tf.keras.layers.Dropout(taxaDropout),#
                        tf.keras.layers.Dense(neuronios, activa
tion=tf.nn.relu),

```



```

    Diferentes valores de dropout.
    tion=tf.nn.relu),
    Diferentes valores de dropout.
    tion=tf.nn.relu),
    Diferentes valores de dropout.
    .nn.softmax)

s)

tf.keras.layers.Dropout(taxaDropout),#
tf.keras.layers.Dense(neuronios, activa
tf.keras.layers.Dropout(taxaDropout),#
tf.keras.layers.Dense(neuronios, activa
tf.keras.layers.Dropout(taxaDropout),#
tf.keras.layers.Dense(10, activation=tf

])
model.compile(optimizer='adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
model.fit(x_train, y_train, epochs=epoca

value = model.evaluate(x_test, y_test)
model_json = model.to_json()
json_file = open("model_MLP4.json", "w")
json_file.write(model_json)
json_file.close()
model.save_weights("model_MLP4.h5")
print("Model saved to disk")
os.getcwd()

somaDasEficienciasDeCadaIteracao = value
[1] + somaDasEficienciasDeCadaIteracao

myMutex.acquire()
numeroDeNeuronios.append(neuronios)
numeroDeEpocas.append(epocas)
numeroDeCamadas.append(camadas)
numeroDeDropout.append(taxaDropout)
taxaDeAcertos.append(somaDasEficienciasDeCadaIte

racao/iteracaoMedia)

myMutex.release()

# Reiniciamos a soma.
somaDasEficienciasDeCadaIteracao = 0

if __name__ == '__main__':

    camadas1 = threading.Thread(target=thread1Camadas, args=(1,))
    camadas2 = threading.Thread(target=thread2Camadas, args=(2,))
    camadas3 = threading.Thread(target=thread3Camadas, args=(3,))
    camadas4 = threading.Thread(target=thread4Camadas, args=(4,))

    camadas1.start()
    camadas2.start()
    camadas3.start()
    camadas4.start()

    try:
        camadas4.join();

```

```
except:
    pass;

try:
    camadas3.join();
except:
    pass;

try:
    camadas2.join();
except:
    pass;

try:
    camadas1.join();
except:
    pass;

listasFile = open("listasFFULLYCONNECTED.txt", "w")
listasFile.write(str(numeroDeNeuronios) + "\n")
listasFile.write(str(numeroDeEpocas) + "\n")
listasFile.write(str(numeroDeCamadas) + "\n")
listasFile.write(str(numeroDeDropout) + "\n")
listasFile.write(str(taxaDeAcertos) + "\n")
listasFile.close()
```

epocas: 8
CAMADAS1: 256

epocas: 8
CAMADAS2: 256

epocas: 8
CAMADAS3: 256

epocas: 8
CAMADAS4: 256

[illegible]

```
60000/60000 [=====] - 29s 480us/sample -
```

[illegible]

[illegible]

```
Epoch 6/8
60000/60000 [=====] - 26s 433us/sample -
loss: 0.0426 - acc: 0.9862
Epoch 7/8
60000/60000 [=====] - 24s 407us/sample -
loss: 0.0294 - acc: 0.9901
Epoch 8/8
60000/60000 [=====] - 29s 483us/sample -
loss: 0.0676 - acc: 0.9805
20800/60000 [=====>.....] - ETA: 17s - loss: 0.
0353 - acc: 0.9887Epoch 6/8
60000/60000 [=====] - 28s 466us/sample -
loss: 0.0511 - acc: 0.9841
48320/60000 [=====>.....] - ETA: 4s - loss: 0.0
232 - acc: 0.9922Epoch 7/8
60000/60000 [=====] - 25s 420us/sample -
loss: 0.0236 - acc: 0.9921
60000/60000 [=====] - ETA: 11s - loss: 0.
0595 - acc: 0.9814 - 27s 447us/sample - loss: 0.0373 - acc: 0.9877
Epoch 8/8
10000/10000 [=====] - 4s 375us/sample - l
oss: 0.0729 - acc: 0.9801
20000/60000 [=====>.....] - ETA: 18s - loss: 0.
0454 - acc: 0.9864Model saved to disk
53888/60000 [=====>....] - ETA: 3s - loss: 0.0
614 - acc: 0.9814Epoch 1/8
60000/60000 [=====] - 30s 493us/sample -
loss: 0.0606 - acc: 0.9817
Epoch 7/8
60000/60000 [=====] - 28s 464us/sample -
loss: 0.0472 - acc: 0.9858
31488/60000 [=====>.....] - ETA: 12s - loss: 0.
3076 - acc: 0.9106Epoch 8/8
60000/60000 [=====] - 27s 458us/sample -
loss: 0.0322 - acc: 0.9895
10000/10000 [=====] - 4s 384us/sample - l
oss: 0.0723 - acc: 0.9808
19488/60000 [=====>.....] - ETA: 19s - loss: 0.
0432 - acc: 0.9868Model saved to disk
60000/60000 [=====]24224/60000 [=====
==>.....] - 25s 413us/sample - loss: 0.2396 - acc: 0.
9302
- ETA: 16s - loss: 0.0442 - acc: 0.9870Epoch 2/8
48096/60000 [=====>.....] - ETA: 5s - loss: 0.0
543 - acc: 0.9839Epoch 1/8
60000/60000 [=====] - 29s 484us/sample -
loss: 0.0564 - acc: 0.9833
Epoch 8/8
60000/60000 [=====] - 28s 472us/sample -
loss: 0.0457 - acc: 0.9860
10000/10000 [=====] - 4s 385us/sample - l
oss: 0.0767 - acc: 0.9823
29312/60000 [=====>.....] - ETA: 14s - loss: 0.
0480 - acc: 0.9863Model saved to disk
37056/60000 [=====>.....] - ETA: 10s - loss: 0.
0472 - acc: 0.9868Epoch 1/8
60000/60000 [=====] - 25s 417us/sample -
loss: 0.1027 - acc: 0.9683
52416/60000 [=====>....]37376/60000 [=====
=====>.....] - ETA: 3s - loss: 0.2329 - acc: 0.9288 - ET
A: 10s - loss: 0.0471 - acc: 0.9868Epoch 3/8
```

[illegible]

[illegible]

```
loss: 0.1132 - acc: 0.9664
Epoch 3/8
60000/60000 [=====] - 27s 458us/sample -
loss: 0.0408 - acc: 0.9875
60000/60000 [=====] - 29s 476us/sample -
loss: 0.0537 - acc: 0.9843
43232/60000 [=====>.....] - ETA: 6s - loss: 0.0
836 - acc: 0.9732Epoch 8/8
60000/60000 [=====] - 27s 445us/sample -
loss: 0.2484 - acc: 0.9248
Epoch 2/8
10000/10000 [=====] - 4s 371us/sample - l
oss: 0.0868 - acc: 0.9798
2656/60000 [>.....] - ETA: 30s - loss: 0.
0440 - acc: 0.9880Model saved to disk
11552/60000 [=====>.....] - ETA: 20s - loss: 0.
1213 - acc: 0.9619Epoch 1/8
60000/60000 [=====] - 24s 405us/sample -
loss: 0.0831 - acc: 0.9736
Epoch 4/8
60000/60000 [=====] - 27s 447us/sample -
loss: 0.1159 - acc: 0.9639
Epoch 3/846304/60000 [=====>.....] - ETA: 6s -
loss: 0.2920 - acc: 0.9104
60000/60000 [=====] - 29s 484us/sample -
loss: 0.0479 - acc: 0.9863
60000/60000 [=====] - 25s 418us/sample -
loss: 0.0653 - acc: 0.9794
Epoch 5/8
10000/10000 [=====] - 4s 363us/sample - l
oss: 0.0785 - acc: 0.9799
60000/60000 [=====] - 28s 468us/sample -
loss: 0.2632 - acc: 0.9195
14688/60000 [=====>.....] - ETA: 19s - loss: 0.
0876 - acc: 0.9732Epoch 2/8
15776/60000 [=====>.....] - ETA: 18s - loss: 0.
0869 - acc: 0.9730Model saved to disk
19200/60000 [=====>.....] - ETA: 14s - loss: 0.
0521 - acc: 0.9834Epoch 1/8
60000/60000 [=====] - 25s 424us/sample -
loss: 0.0923 - acc: 0.9713
44096/60000 [=====>.....] - ETA: 6s - loss: 0.1
359 - acc: 0.9593Epoch 4/8
60000/60000 [=====] 6624/60000 [==
>.....] - 24s 403us/sample - loss: 0.0542 -
acc: 0.9826
- ETA: 25s - loss: 0.0616 - acc: 0.9802Epoch 6/8
60000/60000 [=====] - 27s 449us/sample -
loss: 0.1332 - acc: 0.9600
10432/60000 [=====>.....] - ETA: 21s - loss: 0.
0445 - acc: 0.9868Epoch 3/8
17920/60000 [=====>.....]60000/60000 [=====
=====] - ETA: 19s - loss: 0.1048 - acc: 0.9670 - 3
0s 503us/sample - loss: 0.2882 - acc: 0.9119
30560/60000 [=====>.....] - ETA: 12s - loss: 0.
0452 - acc: 0.9858Epoch 2/8
60000/60000 [=====] - 27s 445us/sample -
loss: 0.0739 - acc: 0.9762
41408/60000 [=====>.....] - ETA: 8s - loss: 0.1
045 - acc: 0.9689Epoch 5/8
60000/60000 [=====] - 25s 418us/sample -
```

◀ ▶

In [0]:

```
# q3Final.py

import tensorflow as tf
import os
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=8)
model.evaluate(x_test, y_test)
model_json = model.to_json()
json_file = open("model_MLP.json", "w")
json_file.write(model_json)
json_file.close()
model.save_weights("model_MLP.h5")
print("Model saved to disk")
os.getcwd()
```

```
Epoch 1/8
60000/60000 [=====] - 5s 84us/sample - los
s: 0.2446 - acc: 0.9246
Epoch 2/8
60000/60000 [=====] - 5s 83us/sample - los
s: 0.1283 - acc: 0.9617
Epoch 3/8
60000/60000 [=====] - 5s 82us/sample - los
s: 0.1008 - acc: 0.9691
Epoch 4/8
60000/60000 [=====] - 5s 82us/sample - los
s: 0.0869 - acc: 0.9737
Epoch 5/8
60000/60000 [=====] - 5s 83us/sample - los
s: 0.0779 - acc: 0.9765
Epoch 6/8
60000/60000 [=====] - 5s 82us/sample - los
s: 0.0712 - acc: 0.9787
Epoch 7/8
60000/60000 [=====] - 5s 83us/sample - los
s: 0.0653 - acc: 0.9803
Epoch 8/8
60000/60000 [=====] - 5s 83us/sample - los
s: 0.0618 - acc: 0.9819
10000/10000 [=====] - 1s 58us/sample - los
s: 0.0669 - acc: 0.9823
Model saved to disk
```

Out[0]:

'/content'

Questão 4

Inicialmente, executou-se 5 vezes o código sugerido inicial a fim de verificar o desempenho da proposta. Seu desempenho foi de:

- Loss: 0.0260; Acurácia: 0.9909.

Utilizando-se o método de tentativa e erro, foi criado um script que verificava o desempenho da rede para diferentes parâmetros alterados como dropout (0.1 a 0.6), número de filtros (32 a 64), épocas de treinamento (2 a 6) e formato dos kernel utilizados (2x2 ou 3x3). O script executava esta mudança de parâmetros dentro de loops "for" para executar todas as combinações possíveis e tirava também a média das múltiplas execuções com mesmos parâmetros, a fim de se obter uma média de desempenho mais confiável. Os resultados desta varredura eram salvos ao final das execuções em um arquivo "listas.txt", permitindo ao usuários verificar qual a configuração obteve melhor desempenho. Foram utilizadas 4 threads - para varredura de redes de 1 a 4 camadas - durante o treinamento, a fim de promover paralelismo e diminuir o tempo requerido, que era ainda maior que o demandado para a questão 1. Verificou-se que com 6 épocas de treinamento o resultado era levemente melhorado, mas não significativamente. A variação das demais grandezas fazia o desempenho diminuir nos testes. Então, após a varredura, foi realizado mais uma tentativa de treinamento com adição de uma camada convolucional e dropout de 0.3, o que elevou os resultados e nos trouxe à proposta final de código para esta questão.

Através da varredura, foi possível perceber que as alterações que implicavam em aumento de desempenho eram: Maior número de filtros, 2 camadas convolucionais, taxa de dropout próxima de 0.3 e kernel 3x3 (com max pool 2x2).

Portanto, para a proposta final deste modelo, os parâmetros alterados que resultaram no melhor desempenho durante a varredura foram:

- Adição de duas camadas convolucionais com kernel 3x3 (seguida de uma max pool em 2x2) após a saída da primeira layer de max pool; 8 épocas de treinamento. Camadas convolucionais todas com 512 filtros e dropout de 0,3. Os demais parâmetros foram mantidos por não apresentar vantagem média significativa.

O desempenho médio obtido foi de:

- Loss: 0.0190; Acurácia: 0.9935.

Ambas as soluções consumiram um tempo de execução da ordem de poucos minutos e a diferença de desempenho foi cerca de 0,22% em ganho.

Os arquivos utilizados foram (no diretório q2):

Proposta Inicial: q4Inicial.py

Script de Varredura de parâmetros: q4.py

Proposta Final: q4Final.py

Comparação entre ELM, MLP e CNN

Desempenho:

- ELM: 91,09%
- MLP: 98,23%

- CNN: 99,35%

Nota-se claramente que a CNN apresenta o melhor desempenho dentre as 3 melhores técnicas utilizadas. Porém, o processo de treinamento para otimização desta toma dezenas de horas, enquanto que a ELM requer apenas alguns minutos para ser ajustada e ficar cerca de 8% abaixo em desempenho. Logo, se

In [0]:

```
# q4Inicial.py

import tensorflow as tf
import os
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][pixels]
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_train, x_test = x_train / 255.0, x_test / 255.0
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3),
    activation='relu',
    input_shape=(28, 28, 1)))
model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.25))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
model.compile(optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
model_json = model.to_json()
json_file = open("model_CNN.json", "w")
json_file.write(model_json)
json_file.close()
model.save_weights("model_CNN.h5")
print("Model saved to disk")
os.getcwd()
```

Epoch 1/5

```
60000/60000 [=====] - 16s 268us/sample - loss: 0.2004 - acc: 0.9403
```

Epoch 2/5

```
60000/60000 [=====] - 15s 256us/sample - loss: 0.0851 - acc: 0.9741
```

Epoch 3/5

```
60000/60000 [=====] - 15s 251us/sample - loss: 0.0628 - acc: 0.9809
```

Epoch 4/5

```
60000/60000 [=====] - 15s 252us/sample - loss: 0.0532 - acc: 0.9837
```

Epoch 5/5

```
60000/60000 [=====] - 15s 249us/sample - loss: 0.0461 - acc: 0.9857
```

```
10000/10000 [=====] - 1s 114us/sample - loss: 0.0294 - acc: 0.9906
```

Model saved to disk

Out[0]:

```
'/content/drive/My Drive/PODE APAGAR/EA072-EF1'
```


In [0]:

```

# q4.py

import tensorflow as tf
import os
import threading

myMutex = threading.Lock()
value = "teste"

numeroDeNeuronios = []
numeroDeEpocas = []
numeroDeCamadas = []
numeroDeDropout = []
taxaDeAcertos = []

# Vamos colocar uma thread para treinar cada rede com um numero especifico de ca
# madas.
def thread1Camadas(camadas):

    # Para tirar a media das iteracoes, somaremos todas aqui e dividiremos p
    # elo total.
    somaDasEficienciasDeCadaIteracao = 0

    # Os valores que utilizaremos para dropout variarao de 10% a 90% (instru
    # cao abaixo).
    valoresDropout = range(10, 40, 10)# Variaremos de 10% em 10%.
    valoresDropout = [i/100 for i in valoresDropout]# Converte de porcentage
    m para escala de 0 a 1.

    # Testando resultados com diferentes quantidades de epocas.
    for epocas in [2, 6]:

        # Testando resultados com diferentes quantidades de filtros.
        for filtros in [32, 64]:

            # So para indicar em que passo da execucao estamos.
            print("\n\nepocas: " + str(epocas) + "\nCAMADAS" + str(c
            amadas) + ": " + str(filtros) + "\n\n")

            # Este loop fica responsável por treinar com diferentes
            # "i" eh o valor a cada iteracao.
            for taxaDropout in valoresDropout:

                # Repetimos o treinamento algumas vezes para tir
                ar uma media da eficiencia
                for iteracaoMedia in range(1,3):
                    mnist = tf.keras.datasets.mnist
                    (x_train, y_train),(x_test, y_test) = mn
                    ist.load_data()
                    # reshape to be [samples][width][height]
                    [pixels]
                    x_train = x_train.reshape(x_train.shape[
                    0], 28, 28, 1)
                    x_test = x_test.reshape(x_test.shape[0],
                    28, 28, 1)
                    x_train, x_test = x_train / 255.0, x_tes
                    t / 255.0
                    model = tf.keras.models.Sequential()

```

```

, kernel_size=(3, 3),
*2, (3, 3), activation='relu'))
ool_size=(2, 2))
opout))

ivation='relu'))
opout))
vation='softmax'))

s)

model.add(tf.keras.layers.Conv2D(filtros
    activation='relu',
    input_shape=(28, 28, 1)))
model.add(tf.keras.layers.Conv2D(filtros
model.add(tf.keras.layers.MaxPooling2D(p
model.add(tf.keras.layers.Dropout(taxaDr
opout))

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, act
model.add(tf.keras.layers.Dropout(taxaDr
opout))

model.add(tf.keras.layers.Dense(10, acti
model.compile(optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
model.fit(x_train, y_train, epochs=epoca
s)

value = model.evaluate(x_test, y_test)
model_json = model.to_json()
json_file = open("model_CNN1.json", "w")
json_file.write(model_json)
json_file.close()
model.save_weights("model_CNN1.h5")
print("Model saved to disk")
os.getcwd()

somaDasEficienciasDeCadaIteracao = value
[1] + somaDasEficienciasDeCadaIteracao

myMutex.acquire()
numeroDeNeuronios.append(filtros)
numeroDeEpocas.append(epocas)
numeroDeCamadas.append(camadas)
numeroDeDropout.append(taxaDropout)
taxaDeAcertos.append(somaDasEficienciasDeCadaIte
racao/iteracaoMedia)

myMutex.release()

# Reiniciamos a soma.
somaDasEficienciasDeCadaIteracao = 0

# Vamos colocar uma thread para treinar cada rede com um numero especifico de ca
madas.
def thread2Camadas(camadas):

    # Para tirar a media das iteracoes, somaremos todas aqui e dividiremos p
elo total.
    somaDasEficienciasDeCadaIteracao = 0

    # Os valores que utilizaremos para dropout variarao de 10% a 90% (instru
cao abaixo).
    valoresDropout = range(10, 40, 10)# Variaremos de 10% em 10%.
    valoresDropout = [i/100 for i in valoresDropout]# Converte de porcentage
m para escala de 0 a 1.

```

```

# Testando resultados com diferentes quantidades de epocas.
for epocas in [2, 6]:

    # Testando resultados com diferentes quantidades de filtros.
    for filtros in [32, 64]:

        # So para indicar em que passo da execucao estamos.
        print("\n\nepocas: " + str(epocas) + "\nCAMADAS" + str(camadas) + ": " + str(filtros) + "\n\n")

        # Este loop fica responsável por treinar com diferentes
        # "i" eh o valor a cada iteracao.
        for taxaDropout in valoresDropout:

            # Repetimos o treinamento algumas vezes para tirar uma media da eficiencia
            for iteracaoMedia in range(1,3):
                mnist = tf.keras.datasets.mnist
                (x_train, y_train),(x_test, y_test) = mnist.load_data()

                # reshape to be [samples][width][height][pixels]
                x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
                x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
                x_train, x_test = x_train / 255.0, x_test / 255.0

                model = tf.keras.models.Sequential()
                model.add(tf.keras.layers.Conv2D(filtros, kernel_size=(2, 2),
                activation='relu',
                input_shape=(28, 28, 1)))
                model.add(tf.keras.layers.Conv2D(filtros*2, (3, 3), activation='relu'))
                model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
                model.add(tf.keras.layers.Conv2D(filtros*2, (3, 3), activation='relu'))
                model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
                model.add(tf.keras.layers.Dropout(taxaDropout))
                model.add(tf.keras.layers.Flatten())
                model.add(tf.keras.layers.Dense(128, activation='relu'))
                model.add(tf.keras.layers.Dropout(taxaDropout))
                model.add(tf.keras.layers.Dense(10, activation='softmax'))

                model.compile(optimizer='adam',
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])
                model.fit(x_train, y_train, epochs=epocas)

                value = model.evaluate(x_test, y_test)
                model_json = model.to_json()
                json_file = open("model_CNN2.json", "w")
                json_file.write(model_json)

```

```

        json_file.close()
        model.save_weights("model_CNN2.h5")
        print("Model saved to disk")
        os.getcwd()

        somaDasEficienciasDeCadaIteracao = value
[1] + somaDasEficienciasDeCadaIteracao

        myMutex.acquire()
        numeroDeNeuronios.append(filtros)
        numeroDeEpocas.append(epocas)
        numeroDeCamadas.append(camadas)
        numeroDeDropout.append(taxaDropout)
        taxaDeAcertos.append(somaDasEficienciasDeCadaIte
        racao/iteracaoMedia)

        myMutex.release()

        # Reiniciamos a soma.
        somaDasEficienciasDeCadaIteracao = 0

# Vamos colocar uma thread para treinar cada rede com um numero especifico de ca
# madas.
def thread3Camadas(camadas):

    # Para tirar a media das iteracoes, somaremos todas aqui e dividiremos p
    # elo total.
    somaDasEficienciasDeCadaIteracao = 0

    # Os valores que utilizaremos para dropout variarao de 10% a 90% (instru
    # cao abaixo).
    valoresDropout = range(10, 40, 10)# Variaremos de 10% em 10%.
    valoresDropout = [i/100 for i in valoresDropout]# Converte de porcentaje
    m para escala de 0 a 1.

    # Testando resultados com diferentes quantidades de epocas.
    for epocas in [2, 6]:

        # Testando resultados com diferentes quantidades de filtros.
        for filtros in [32, 64]:

            # So para indicar em que passo da execucao estamos.
            print("\n\nepocas: " + str(epocas) + "\nCAMADAS" + str(c
            amadas) + ": " + str(filtros) + "\n\n")

            # Este loop fica responsável por treinar com diferentes
            # taxas de dropout.
            # "i" eh o valor a cada iteracao.
            for taxaDropout in valoresDropout:

                # Repetimos o treinamento algumas vezes para tir
                ar uma media da eficiencia
                for iteracaoMedia in range(1,3):
                    mnist = tf.keras.datasets.mnist
                    (x_train, y_train),(x_test, y_test) = mn
                    ist.load_data()

                    # reshape to be [samples][width][height]
                    [pixels]
                    x_train = x_train.reshape(x_train.shape[

```

```

0], 28, 28, 1)

28, 28, 1)

t / 255.0

, kernel_size=(3, 3),

*2, (2, 2), activation='relu'))

ool_size=(2, 2)))

opout))

ivation='relu'))

opout))

vation='softmax'))

s)

value = model.evaluate(x_test, y_test)
model_json = model.to_json()
json_file = open("model_CNN3.json", "w")
json_file.write(model_json)
json_file.close()
model.save_weights("model_CNN3.h5")
print("Model saved to disk")
os.getcwd()

somaDasEficienciasDeCadaIteracao = value

[1] + somaDasEficienciasDeCadaIteracao

myMutex.acquire()
numeroDeNeuronios.append(filtros)
numeroDeEpocas.append(epocas)
numeroDeCamadas.append(camadas)
numeroDeDropout.append(taxaDropout)
taxaDeAcertos.append(somaDasEficienciasDeCadaIte

racao/iteracaoMedia)

myMutex.release()

# Reiniciamos a soma.
somaDasEficienciasDeCadaIteracao = 0

# Vamos colocar uma thread para treinar cada rede com um numero especifico de ca
madas.
def thread4Camadas(camadas):

    # Para tirar a media das iteracoes, somaremos todas aqui e dividiremos p
elo total.
    somaDasEficienciasDeCadaIteracao = 0

```

```

# Os valores que utilizaremos para dropout variarao de 10% a 90% (instrucao abaixo).
valoresDropout = range(10, 40, 10)# Variaremos de 10% em 10%.
valoresDropout = [i/100 for i in valoresDropout]# Converte de porcentagem para escala de 0 a 1.

# Testando resultados com diferentes quantidades de epocas.
for epocas in [2, 6]:

    # Testando resultados com diferentes quantidades de filtros.
    for filtros in [32, 64]:

        # So para indicar em que passo da execucao estamos.
        print("\n\nepocas: " + str(epocas) + "\nCAMADAS" + str(camadas) + ": " + str(filtros) + "\n\n")

        # Este loop fica responsável por treinar com diferentes taxas de dropout.
        # "i" eh o valor a cada iteracao.
        for taxaDropout in valoresDropout:

            # Repetimos o treinamento algumas vezes para tirar uma media da eficiencia
            for iteracaoMedia in range(1,3):
                mnist = tf.keras.datasets.mnist
                (x_train, y_train),(x_test, y_test) = mnist.load_data()
                # reshape to be [samples][width][height][pixels]
                x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
                x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
                x_train, x_test = x_train / 255.0, x_test / 255.0

                model = tf.keras.models.Sequential()
                model.add(tf.keras.layers.Conv2D(filtros, kernel_size=(3, 3),
                    activation='relu', input_shape=(28, 28, 1)))
                model.add(tf.keras.layers.Conv2D(filtros*2, (3, 3), activation='relu'))
                model.add(tf.keras.layers.MaxPooling2D(pool_size=(3, 3)))
                model.add(tf.keras.layers.Conv2D(filtros*2, (3, 3), activation='relu'))
                model.add(tf.keras.layers.MaxPooling2D(pool_size=(3, 3)))
                model.add(tf.keras.layers.Dropout(taxaDropout))
                model.add(tf.keras.layers.Flatten())
                model.add(tf.keras.layers.Dense(128, activation='relu'))
                model.add(tf.keras.layers.Dropout(taxaDropout))
                model.add(tf.keras.layers.Dense(10, activation='softmax'))

                model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

```

```

s)
    model.fit(x_train, y_train, epochs=epoca

value = model.evaluate(x_test, y_test)
model_json = model.to_json()
json_file = open("model_CNN4.json", "w")
json_file.write(model_json)
json_file.close()
model.save_weights("model_CNN4.h5")
print("Model saved to disk")
os.getcwd()

somaDasEficienciasDeCadaIteracao = value
[1] + somaDasEficienciasDeCadaIteracao

myMutex.acquire()
numeroDeNeuronios.append(filtros)
numeroDeEpocas.append(epocas)
numeroDeCamadas.append(camadas)
numeroDeDropout.append(taxaDropout)
taxaDeAcertos.append(somaDasEficienciasDeCadaIte
racao/iteracaoMedia)

myMutex.release()

# Reiniciamos a soma.
somaDasEficienciasDeCadaIteracao = 0

if __name__ == '__main__':

    camadas1 = threading.Thread(target=thread1Camadas, args=(1,))
    camadas2 = threading.Thread(target=thread2Camadas, args=(2,))
    camadas3 = threading.Thread(target=thread3Camadas, args=(3,))
    camadas4 = threading.Thread(target=thread4Camadas, args=(4,))

    camadas1.start()
    camadas2.start()
    camadas3.start()
    camadas4.start()

    try:
        camadas4.join();
    except:
        pass;

    try:
        camadas3.join();
    except:
        pass;

    try:
        camadas2.join();
    except:
        pass;

    try:
        camadas1.join();

```

```
except:
    pass;

listasFile = open("listasCONV.txt", "w")
listasFile.write(str(numeroDeNeuronios) + "\n")
listasFile.write(str(numeroDeEpocas) + "\n")
listasFile.write(str(numeroDeCamadas) + "\n")
listasFile.write(str(numeroDeDropout) + "\n")
listasFile.write(str(taxaDeAcertos) + "\n")
listasFile.close()
```


epocas: 2
CAMADAS4: 32

32/41

[illegible]

```
10000/10000 [=====] 8864/60000 [====>.....] - 5s 490us/sample - loss: 0.0226 - acc: 0.9920
10176/60000 [====>.....] - ETA: 30s - loss: 0.0408 - acc: 0.8755Model saved to disk
16736/60000 [=====>.....] - ETA: 22s - loss: 0.3068 - acc: 0.9071Epoch 1/2
3808/60000 [>.....] - ETA: 41s - loss: 0.6087 - acc: 0.8072Epoch 1/2
23264/60000 [=====>.....] - ETA: 19s - loss: 0.2558 - acc: 0.9233Epoch 1/2
60000/60000 [=====] - 44s 727us/sample - loss: 0.1517 - acc: 0.9544
42400/60000 [=====>.....] - ETA: 14s - loss: 0.1568 - acc: 0.9520Epoch 2/2
60000/60000 [=====]17280/60000 [=====>.....] - 50s 831us/sample - loss: 0.1301 - acc: 0.9600
- ETA: 37s - loss: 0.0582 - acc: 0.9823Epoch 2/2
60000/60000 [=====] - 53s 879us/sample - loss: 0.1961 - acc: 0.9370
24256/60000 [=====>.....] - ETA: 30s - loss: 0.0578 - acc: 0.9824
60000/60000 [=====] - 53s 886us/sample - loss: 0.1372 - acc: 0.9575
1120/60000 [.....] - ETA: 53s - loss: 0.0643 - acc: 0.9812Epoch 2/2
60000/60000 [=====] - 51s 858us/sample - loss: 0.0561 - acc: 0.9828
10000/10000 [=====] - 7s 663us/sample - loss: 0.0444 - acc: 0.9855
44224/60000 [=====>.....] - ETA: 13s - loss: 0.0500 - acc: 0.9847Model saved to disk
51232/60000 [=====>.....] - ETA: 7s - loss: 0.0617 - acc: 0.9809Epoch 1/2
60000/60000 [=====] - 49s 811us/sample - loss: 0.0476 - acc: 0.9852
60000/60000 [=====] - 49s 812us/sample - loss: 0.0613 - acc: 0.9812
10000/10000 [=====] - 7s 687us/sample - loss: 0.0332 - acc: 0.9880
60000/60000 [=====] - 50s 826us/sample - loss: 0.0504 - acc: 0.9844
- ETA: 36s - loss: 0.3671 - acc: 0.8889 2688/10000 [=====>.....] - ETA: 4s - loss: 0.0465 - acc: 0.9855Model saved to disk
10000/10000 [=====]16960/60000 [=====>.....] - 5s 456us/sample - loss: 0.0287 - acc: 0.9910
18208/60000 [=====>.....] - ETA: 27s - loss: 0.2861 - acc: 0.9138Model saved to disk
9856/10000 [=====>.] - ETA: 0s - loss: 0.0253 - acc: 0.9915
10000/10000 [=====] - 4s 429us/sample - loss: 0.0251 - acc: 0.9916
992/60000 [.....] - ETA: 54s - loss: 1.2646 - acc: 0.5978Model saved to disk
8960/60000 [====>.....] - ETA: 30s - loss: 0.3878 - acc: 0.8802Epoch 1/2
```

```
10528/60000 [====>.....]30944/60000 [=====
====>.....] - ETA: 28s - loss: 0.3538 - acc: 0.8903 - E
TA: 17s - loss: 0.2135 - acc: 0.9359Epoch 1/2
60000/60000 [=====] - 43s 716us/sample -
loss: 0.1497 - acc: 0.9546
Epoch 2/2
60000/60000 [=====] - 48s 803us/sample -
loss: 0.1308 - acc: 0.9597
22080/60000 [=====>.....] - ETA: 31s - loss: 0.
0559 - acc: 0.9824Epoch 2/2
60000/60000 [=====] - 52s 870us/sample -
loss: 0.1849 - acc: 0.9408
59712/60000 [=====>.....] - ETA: 0s - loss: 0.1
432 - acc: 0.9552Epoch 2/2
60000/60000 [=====] - 52s 861us/sample -
loss: 0.1431 - acc: 0.9552
384/60000 [.....] - ETA: 43s - loss: 0.
0399 - acc: 0.9870Epoch 2/2
25632/60000 [=====>.....]60000/60000 [=====
=====] - ETA: 30s - loss: 0.0634 - acc: 0.9808 - 5
0s 833us/sample - loss: 0.0557 - acc: 0.9826
10000/10000 [=====] - 7s 692us/sample - l
oss: 0.0404 - acc: 0.9863
46592/60000 [=====>.....]35456/60000 [=====
=====>.....] - ETA: 11s - loss: 0.0487 - acc: 0.9851 - E
TA: 20s - loss: 0.0542 - acc: 0.9831Model saved to disk
40320/60000 [=====>.....] - ETA: 16s - loss: 0.
0610 - acc: 0.9809Epoch 1/2
60000/60000 [=====] - 51s 849us/sample -
loss: 0.0487 - acc: 0.9850
10000/10000 [=====] - 7s 680us/sample - l
oss: 0.0326 - acc: 0.9892
18240/60000 [=====>.....] - ETA: 34s - loss: 0.
2834 - acc: 0.9126Model saved to disk
60000/60000 [=====] - 50s 833us/sample -
loss: 0.0607 - acc: 0.9815
60000/60000 [=====] - 50s 835us/sample -
loss: 0.0509 - acc: 0.9845
8288/10000 [=====>.....] - ETA: 0s - loss: 0.0
270 - acc: 0.9906Epoch 1/2
10000/10000 [=====] - 5s 457us/sample - l
oss: 0.0252 - acc: 0.9914
10000/10000 [=====] - 5s 486us/sample - l
oss: 0.0356 - acc: 0.9888
29696/60000 [=====>.....]
31072/60000 [=====>.....] - ETA: 20s - loss: 0.
2140 - acc: 0.9350Model saved to disk
41152/60000 [=====>.....] - ETA: 12s - loss: 0.
1846 - acc: 0.9438Epoch 1/2
416/60000 [.....] - ETA: 2:32 - loss:
2.2737 - acc: 0.1514
60000/60000 [=====] - 43s 712us/sample -
loss: 0.1542 - acc: 0.9530
30432/60000 [=====>.....] - ETA: 22s - loss: 0.
2103 - acc: 0.9360Epoch 2/2
60000/60000 [=====] - 49s 809us/sample -
loss: 0.1484 - acc: 0.9548
45248/60000 [=====>.....] - ETA: 13s - loss: 0.
2581 - acc: 0.9160
60000/60000 [=====] - 53s 890us/sample -
loss: 0.2196 - acc: 0.9287
```

epocas: 2
CAMADAS3: 64

[illegible]

[illegible]

Epoch 2/2

```
5856/60000 [=>.....] - ETA: 1:17 - loss:
```

0.0470 - acc: 0.984

[illegible]

Buffered data was truncated after reaching the output size limit.

In [0]:

```
# q4Final.py
import tensorflow as tf
import os
mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][pixels]
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_train, x_test = x_train / 255.0, x_test / 255.0
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3),
    activation='relu',
    input_shape=(28, 28, 1)))
model.add(tf.keras.layers.Conv2D(512, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Conv2D(512, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Conv2D(512, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
model.compile(optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
model.fit(x_train, y_train, epochs=10)
model.evaluate(x_test, y_test)
model_json = model.to_json()
json_file = open("model_CNN.json", "w")
json_file.write(model_json)
json_file.close()
model.save_weights("model_CNN.h5")
print("Model saved to disk")
os.getcwd()
```

```
Epoch 1/10
60000/60000 [=====] - 79s 1ms/sample - los
s: 0.2189 - acc: 0.9315
Epoch 2/10
60000/60000 [=====] - 78s 1ms/sample - los
s: 0.0785 - acc: 0.9787
Epoch 3/10
60000/60000 [=====] - 78s 1ms/sample - los
s: 0.0625 - acc: 0.9830
Epoch 4/10
60000/60000 [=====] - 78s 1ms/sample - los
s: 0.0532 - acc: 0.9855
Epoch 5/10
60000/60000 [=====] - 78s 1ms/sample - los
s: 0.0457 - acc: 0.9871
Epoch 6/10
60000/60000 [=====] - 78s 1ms/sample - los
s: 0.0438 - acc: 0.9879
Epoch 7/10
60000/60000 [=====] - 78s 1ms/sample - los
s: 0.0376 - acc: 0.9899
Epoch 8/10
60000/60000 [=====] - 78s 1ms/sample - los
s: 0.0353 - acc: 0.9904
Epoch 9/10
60000/60000 [=====] - 78s 1ms/sample - los
s: 0.0340 - acc: 0.9907
Epoch 10/10
60000/60000 [=====] - 78s 1ms/sample - los
s: 0.0335 - acc: 0.9909
10000/10000 [=====] - 4s 428us/sample - los
s: 0.0298 - acc: 0.9935
Model saved to disk
```

Out[0]:

'/content'