Redes neurais artificiais

June 9, 2020

1 Relatório do trabalho

Nome: Álvaro Leandro Cavalcante Carneiro **Linguagem utilizada:** Python 3.6

Os códigos e o relatório foram desenvolvidos em um Jupyter notebook, a explicação de cada bloco de código se encontra acima do mesmo e em algumas partes também existem linhas comentadas.

2 Qual o problema ?

O primeiro problema foi utilizar uma rede neural no modelo perceptron para identificar três classes de iris diferentes baseados em suas características de tamanho de pétalada e sépala.

2.1 Importação das bibliotecas

Ferramentas usadas em todo o processo de desenvolvimento

[153]:
```
import random
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
import time
import itertools
from sklearn.metrics import confusion_matrix
```

2.2 Análise Exploratória dos Dados (AED)

A ideia a princípio é entender um pouco mais do *dataset* e deixar os dados em um formato que seja favorável para o aprendizado da rede neural perceptron.

O primeiro passo foi ler o arquivo encontrado aqui: http://archive.ics.uci.edu/ml/datasets/Iris. O arquivo em questão não possuia colunas, portanto eu modifiquei o mesmo para adicionar o nome das colunas na primeira linha e deixá-lo no formato .CSV ao invés do .DATA.

O conjunto de dados está sendo lido com o método *read_csv* do pandas, que transforma o *dataset* em um *dataframe*, que por sua vez possui diversos métodos nativos para manipular os dados. Um desse métodos é o *head*, onde é possível visualizar as primeiras linhas do *dataframe*.

[154]:
```
dataframe = pd.read_csv('/home/alvaro/Documentos/mestrado/computação bio/redes␣
↪neurais/datasets/iris2.csv', header = 0)

dataframe.head()
```

[154]:

	sepal-length	sepal-width	petal-length	petal-width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Um pré-requisito na mineração dos dados é verificar se o *dataframe* possui inconsistências quanto aos valores, podendo ser algum outlier, ruído, valor vazio, etc...

Para isso, é utilizado o método *isna* para contabilizar os registros vazios por coluna e também o método *describe* que gera um sumário de estatísticas por coluna dos valores ali contidos.

[155]:
```
print('Valores nulos:')
print(dataframe.isna().sum())
dataframe.describe()
```

```
Valores nulos:
sepal-length    0
sepal-width     0
petal-length    0
petal-width     0
class           0
dtype: int64
```

[155]:

	sepal-length	sepal-width	petal-length	petal-width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

2.2.1 Mostrando a dispersão dos dados

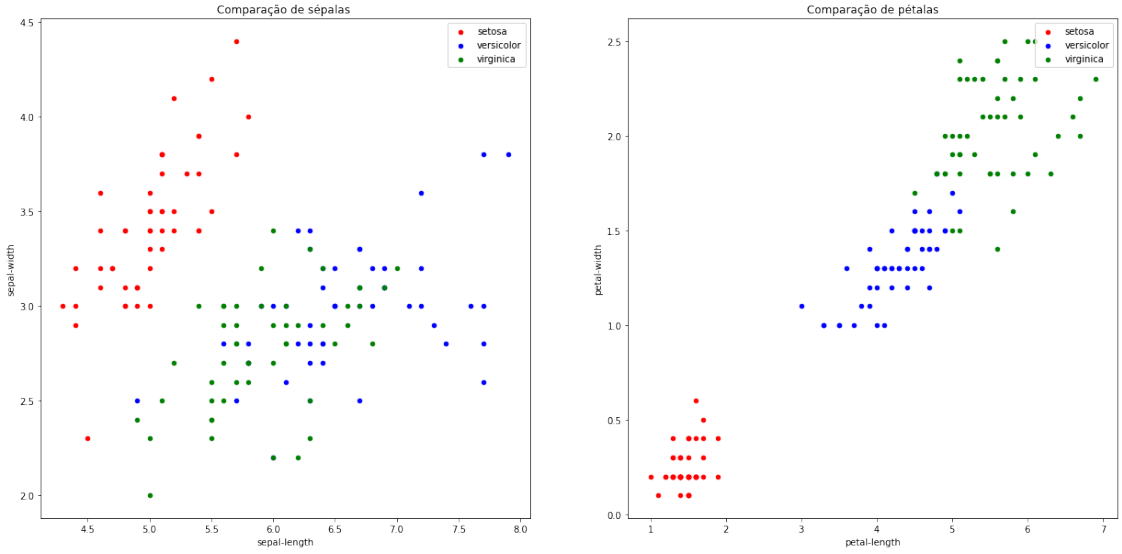
É interessante também plotar um gráfico para mostrar o comportamento da dispersão das classes do conjunto, sendo possível inclusive ver se o problema é linearmente separável.

Antes disso, é necessário dividir o *dataframe* em uma variável chamada **previsores** e outra chamada **classe**. Como o nome sugere, os previsores são as colunas com as características das flores (atributos previsores) que serão utilizados para tentar ajustar os pesos da rede de maneira à generalizar uma solução que encontre as classes corretamente.

[156]:
```
previsores = dataframe.iloc[:, 0:4]
classe = dataframe['class']
```

[157]:
```
# iniciando a figura
plt.figure()
fig,ax=plt.subplots(1,2,figsize=(21, 10))

# separando o dataset por classe
setosa = dataframe[dataframe['class']=='Iris-setosa']
versicolor = dataframe[dataframe['class']=='Iris-versicolor']
virginica = dataframe[dataframe['class']=='Iris-virginica']

# plotando os conjuntos no gráfico de dispersão
setosa.plot(x="sepal-length", y="sepal-width",␣
 ↪kind="scatter",ax=ax[0],label='setosa',color='r')
virginica.
 ↪plot(x="sepal-length",y="sepal-width",kind="scatter",ax=ax[0],label='versicolor',color='b')
versicolor.plot(x="sepal-length", y="sepal-width", kind="scatter", ax=ax[0],␣
 ↪label='virginica', color='g')

setosa.plot(x="petal-length", y="petal-width",␣
 ↪kind="scatter",ax=ax[1],label='setosa',color='r')
versicolor.
 ↪plot(x="petal-length",y="petal-width",kind="scatter",ax=ax[1],label='versicolor',color='b')
virginica.plot(x="petal-length", y="petal-width", kind="scatter", ax=ax[1],␣
 ↪label='virginica', color='g')
# Adicionado legendas
ax[0].set(title='Comparação de sépalas ', ylabel='sepal-width')
ax[1].set(title='Comparação de pétalas',  ylabel='petal-width')
ax[0].legend()
ax[1].legend()
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



Conclui-se que as classes virgínica e versicolor são as mais complicadas de serem separadas, principalmente em relação ao tamanho das pétalas.

2.3 Preparando os dados para o treinamento

Agora que a análise e entendimento do conjunto foi feito, a ideia é fazer as transformações necessárias nos dados para deixar em uma formato adequado para o treinamento da rede neural

2.3.1 Normalização dos atributos previsores

O método *isnul()* mostrou que não há nenhum registro vazio no *dataset* e é possível observar que os valores também parecem estar todos coerentes, sem a presença de outliers, como podemos notar pelo desvio padrão, mínimo e máximo de cada coluna e também no gráfico de dispersão.

Todavia, existe uma variação relativamente grande dentro do nosso domínio de atributos previsores. O atributo *petal-width* por exemplo, tem uma média de valor de 1.1, enquanto o *sepal-length* possui uma média de 5.8. Dito isso, se faz necessário a padronização desses valores, para que nosso ajuste dos pesos não seja muito influenciado por essa diferença no tamanho da entrada.

O tipo de normalização escolhido foi o **Z-score**, de forma arbitrária, por ser bastante comum em problemas como esse. Sua fórmula é bastante simples e foi representada no método *normalizacao_z_score*.

[158]:
```
def normalizacao_z_score(valor):
    media = previsores[valor.name].mean()
    desvio_padrao = previsores[valor.name].std()

    return (valor - media) / desvio_padrao
```

O método *apply()* do pandas juntamente com a **lambda** aplicam o processo matemático do método de normalização em cada um dos registros do *dataframe*. Os novos registros normalizados podem ser vistos abaixo.

[159]:
```
previsores = previsores.apply(lambda row: normalizacao_z_score(row))
previsores.head()
```

[159]:

	sepal-length	sepal-width	petal-length	petal-width
0	-0.897674	1.028611	-1.336794	-1.308593
1	-1.139200	-0.124540	-1.336794	-1.308593
2	-1.380727	0.336720	-1.393470	-1.308593
3	-1.501490	0.106090	-1.280118	-1.308593
4	-1.018437	1.259242	-1.336794	-1.308593

2.3.2 Tratando valores categóricos

O próximo passo será transformar o valor da classe de categórico para discreto, para que seja possível aplicar os cálculos, como o erro da saída por exemplo.

Para isso foi criado o método *get_dicionario_classes* que gera uma estrutura de dicionário dinâmica, baseado na quantidade de classes do problema que está sendo tratado. O processo é muito simples, basta percorrer as classes existentes e atribuir um valor inteiro para cada classe.

[160]:
```
def get_dicionario_classes(classe):
    dict_classes = {}
    count = 0

    for i in classe.unique():
        dict_classes[i] = count
        count += 1

    return dict_classes
```

[161]:
```
dict_classes = get_dicionario_classes(classe)
print(dict_classes)
```

```
{'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-virginica': 2}
```

Podemos ver acima os valores que o método atribuiu para cada uma das classes desse problema.

Basta agora repetir o processos anterior de usar o método *apply()*, porém agora passando no *lambda* o método que vai atribuir a classe a seu determinado valor no dicionário que foi criado anteriormente.

[162]:
```
def transformar_categorico_em_numerico(valor, dict_classes):
    return dict_classes[valor]

classe = classe.apply(lambda row: transformar_categorico_em_numerico(row,␣
 ↪dict_classes))
print(classe.value_counts())
```

```
2    50
1    50
0    50
Name: class, dtype: int64
```

2.3.3 Lidando com problemas multi-classe

O problema em questão é multi-classe, ou seja, possuímos mais de duas classes como resposta na camada de saída, podendo ser, íris-setosa, versicolor ou virginica. Para problemas binários utilizar um único neurônio com a saída de 0 e 1 nos basta, todavia aqui, vamos precisar criar um novo neurônio para cada classe, totalizando 3 na nossa camada de saída.

Além de modificar a estrutura da rede neural, também será preciso codificar os valores, uma vez que, ao invés de um valor escalar será trabalhado com um array na saída da rede, sendo este representado por: [1,0,0], [0,1,0] e [0,0,1].

[163]:
```
def codificar_classe():
    classe_codificada = {}
    array_classe = [1] + [0] * (len(classe.unique()) - 1) #cria um array␣
↪dinâmico baseado na
    #quantidade de classes, é [1,0,0] para esse problema mas poderia ser␣
↪[1,0,0,0....,0].
    count = 1
    classe_codificada[0] = array_classe.copy()

    for i in range(len(classe.unique()) - 1): # percorre todas as classes -1,␣
↪pois já temos a primeira posição do dicionário por padrão
        array_classe[count - 1] = 0 # o valor 1 atual vira 0
        array_classe[count] = 1  # a próxima casa do array vira 1
        classe_codificada[count] = array_classe.copy()
        count += 1

    return classe_codificada

classe_codificada = codificar_classe()
```

[164]:
```
classe_codificada
```

[164]:
```
{0: [1, 0, 0], 1: [0, 1, 0], 2: [0, 0, 1]}
```

A ideia do método *codificar_classe* é criar mais um dicionário, como é possível ver acima, onde cada posição representa uma classe codificada em um array de binários. O tamanho desse array é dinâmico dependendo do número de classes e a ideia é ir movimentando o valor do 1 conforme as iterações.

Feito isso, basta repetir o processo para substituir o valor da classe.

[165]:
```
def substituir_classe_codificada(valor, classe_codificada):
    return classe_codificada[valor]

classe = classe.apply(lambda row: substituir_classe_codificada(row,␣
 ↪classe_codificada))
print(classe.head())
```

```
0    [1, 0, 0]
1    [1, 0, 0]
2    [1, 0, 0]
3    [1, 0, 0]
4    [1, 0, 0]
Name: class, dtype: object
```

Com isso, a classe agora está em uma estrutura que vai suportar o problema multi-classe.

2.3.4 Divisão do dataframe

Agora que os dados do *dataframe* já estão no formato necessário, basta dividir as bases em treinamento, validação e teste, usando a proporção de 70%, 15% e 15%.

Para isso foi criado o método *dividir_dataframe* onde será utilizado o método *sample* do pandas para pegar amostras aleatórias sem reposição do dataframe, e a partir dessa amostra criar os demais conjuntos.

O *x_treinamento* vai ser a fatia responsável por treinar a rede e ajustar os pesos. O teste do treinamento será feito ao final de cada época na base chamada *x_teste*, proporcionando uma avaliação não enviesada dos resultados da rede em dados não vistos no treino.

Após a rede estar completamente treinada, iremos usar a base de *x_validacao* para gerar novas previsões baseado em atributos previsores nunca antes vistos pela rede, dando uma validação final da eficácia do treinamento.

Os parâmetros com sulfixo *p* indicam o porcentagem que será atribuida para cada base de dados e o último parâmetro de época vai mudar o retorno da função dependendo se o perceptron é do tipo que atualiza os pesos por registro ou por época (trabalhado posteriormente).

[166]:
```
def dividir_dataframe(previsores, classe, p_treinamento, p_teste, p_validacao,␣
 ↪epoca = False):
    x_treinamento = previsores.sample(frac = p_treinamento)
    y_treinamento = classe[x_treinamento.index]

    x_teste_sem_previsores = previsores.drop(x_treinamento.index)
    nova_p_teste = p_teste / (1 - p_treinamento)

    x_teste = x_teste_sem_previsores.sample(frac = nova_p_teste)
    y_teste = classe[x_teste.index]

    x_validacao = x_teste_sem_previsores.drop(x_teste.index)
```

```
    y_validacao = classe[x_validacao.index]

    if epoca == False:
        return x_treinamento.reset_index(drop=True), y_treinamento.
↪reset_index(drop=True), \
        x_teste.reset_index(drop=True), y_teste.reset_index(drop=True),␣
↪x_validacao.reset_index(drop=True), y_validacao.reset_index(drop=True)
    else:
        # não tem reset index na classe
        return x_treinamento.reset_index(drop=True), y_treinamento, \
        x_teste.reset_index(drop=True), y_teste, x_validacao.
↪reset_index(drop=True), y_validacao
```

A nomenclatura de "x" representa os atributos previsores e "y" a classe.

Depois de criar a fração de treinamento, é removido dos previsores todos os dados que estão na porção de treinamento, pois uma regra importante a ser seguida na divisão dos dados é o particionamento, ou seja, nenhum dos registros de treinamento deve estar no conjunto de teste e vice versa.

Depois disso, o conjunto total se torna o resto que não está no conjunto de treinamento, portanto as porcentagens também são redimensionadas, por exemplo, se antes a proporção era de 15% do conjunto para teste e 15% para validação, agora cada um desses 15% representa 50%, pois o novo 100% está sem os registros de treinamento.

Feito isso, basta dividir novamente a base em teste e validação e retornar os conjuntos divididos.

2.4 Implementação e treinamento da rede perceptron

Os próximos passos são relativos à implementação dos métodos usados na rede perceptron para realizar o treinamento e também exibição dos resultados.

2.4.1 Inicialização dos pesos

Os pesos serão inicializados de forma aleatória para então serem gradativamente ajustados conforme a rede neural converge. Para isso, foi criado o método *inicializar_pesos*, que percorre cada um dos neurônios e gera um vetor da quantidade de pesos que ele possui baseado nas suas conexões sinapticas com os neurônios da próxima camada.

Além disso, o método também recebe um parâmetro chamado dominio, que é o intervalo de valores que os pesos serão gerados, os testes a princípio foram realizados em um domínio de [0,1].

[167]:
```
def inicializar_pesos(dominio):
    pesos_final = []

    for i in range(len(previsores.columns)):
        pesos = []
        for j in range(len(dict_classes)):
```

```
            pesos.append(random.uniform(dominio[0], dominio[1]))
        pesos_final.append(pesos)
    return pesos_final
```

[168]:
```
pesos = inicializar_pesos([0, 1])
print('Pesos:', pesos)
```

```
Pesos: [[0.17768656001752492, 0.8170163419118891, 0.40378016499446556],
[0.0693378669592496, 0.2860022635053818, 0.7141138726767527],
[0.3007106638195035, 0.6329457363931004, 0.17640956297949717],
[0.9965508984553443, 0.6719292126986111, 0.23033774072641533]]
```

Como é possível observar, o array de pesos para esse problema possui 4 posições com 3 pesos em cada uma das posições. Isso acontece porque possuímos 4 neurônios (os atributos de entrada/previsores sem considerar o bias até então) conectados à 3 neurônios (um neurônio para cada saída possível), portanto cada um dos 4 neurônios possui 3 pesos (conexões) cada um.

Para obter o número de conexões por camada basta multiplicar o número de neurônios da camada atual pelos número de neurônios na próxima camada, dessa forma temos: $4 * 3 = 12$ conexões com pesos para serem atualizados.

2.4.2 Função de soma

A função de soma acontece em todos os neurônios, somando o valor do produto da multiplicação entre os neurônios adjacentes anteriores com os pesos das sinapses artificiais. Esse valor de soma é o valor final do neurônio após receber todas as sinapses e será usado na função de ativação para indicar se o neurônio em questão foi excitado ou inibido.

A soma do produto pode ser feita de forma simples usando o método *dot* do numpy, retornando um produto escalar.

[169]:
```
def somatoria(entradas, pesos):
    return np.dot(entradas, pesos)
```

2.4.3 Função de ativação

A função de ativação por *default* no perceptron é a chamada "*step function* (função degrau)", onde o neurônio artificial é excitado ou não baseado em um *threshold* (limiar) pré definido. Nesse caso, se o valor do neurônio for maior que 0 ele retorna o 1, excitando a célula, caso contrário, retorna 0.

Por ser um problema multiclasse, foi criado um laço *for* para percorrer o array da classe e excitar todas as posições em que o valor é maior que 0.

Claro que isso acaba gerando um problema onde mais de um neurônio por vez na camada de saída pode ser excitado.

[170]:
```
def funcao_ativacao_step(soma):
    ativacao = []
```

```
    for i in soma:
        if i > 0:
            ativacao.append(1)
        else:
            ativacao.append(0)

    return ativacao, ativacao
```

2.4.4 Função de custo

A função de custo é a responsável por calcular o erro da rede neural ao comparar o valor correto com o valor que foi previsto.

A variável de *erro* indica se a rede neural errou a previsão ou acertou (uma vez que os pesos aqui não serão atualizados em caso de acerto), já a variável *valor_erro*, indica o valor exato em cada neurônio de saída que a rede classificou incorretamente, para alcançar uma precisão maior na atualização dos pesos.

Por exemplo, se o valor previsto foi: [0.8,0.37,0.16] e o valor real é: [0,1,0], a variável *valor_erro* vai trazer o erro do algoritmo por posição: [0.8, 0.63, 0.16], totalizando uma soma de 1.59.

[171]:
```
def funcao_custo(valor_correto, valor_previsto, valor_ativacao):
    erro = valor_correto != valor_previsto
    valor_erro = list(abs(np.array(valor_correto) - np.array(valor_ativacao)))
    return erro, sum(valor_erro) # valor escalar
```

2.4.5 Função de atualização de peso

A formula da atualização de pesos no perceptron foi representada no método de *atualizar_peso*

Com ela, conseguimos ajustar os pesos seguindo uma taxa de aprendizado (basicamente o tamanho do ”passo”) além de levar em consideração a grandeza da entrada e o quanto a previsão estava incorreta.

Algo importante de se considerar é que o perceptron faz a atualização dos pesos POR REGISTRO, ou seja, a cada registro apresentado a rede neural que é classificado de forma incorreta é gerado uma atualização nos pesos, podendo dificultar a convergência devido à sensibilidade aos dados contidos no *dataframe* e também aumentando o tempo de execução do algoritmo.

[172]:
```
def atualizar_peso(entrada, peso, erro, tx_aprendizado):
    novo_peso = peso + (tx_aprendizado * entrada * erro)
    return novo_peso
```

2.4.6 Bias

O Bias é a constante que será adicionada como sendo uma das colunas do *dataframe*. Essa coluna, assim como as demais, irá se transformar em um neurônio da rede, que vai ajudar nos cálculos dos

pesos.

O Bias tem uma atualização diferenciada, pois não leva em consideração a entrada, portanto foi criado um método para atualizar o bias.

[173]:
```
def atualizar_bias(entrada, peso, erro, tx_aprendizado):
    novo_peso = peso + np.float64(tx_aprendizado * erro)
    return novo_peso
```

[174]:
```
previsores['bias'] = 1 # adicionando o bias na coluna
```

2.4.7 Matriz de confusão

Foi utilizado a biblioteca do *sklearn* para a implementação da matriz de confusão, sendo necessário apenas adaptar o formato dos dados para um array de valores escalares. Esse processo de adaptação foi feito no método *get_matriz_confusao*.

Também foi necessário adicionar um *try/except* para retornar um array vazio em caso de erros na sua geração. Isso se deu pelo fato de que ao utilizar a função de ativação "degrau" mais de um neorônio poderia ser ativado ao mesmo tempo, gerando problemas de incompatibilidade na geração da matriz.

[175]:
```
def get_matriz_confusao(valor_correto, valor_previsto):
    try:
        previsao = np.array(valor_previsto.copy()) # deep copy da variável
        previsao = np.where(previsao == 1)[1] # transformando em um array de␣
↪valores escalares

        correto = np.array(list(valor_correto.values))
        correto = np.where(correto == 1)[1]

        matriz_confusao = confusion_matrix(correto, previsao)

        return matriz_confusao
    except:
        return []
```

2.4.8 Implementação do perceptron

O método *treinar* é o que vai implementar de fato todas as etapas que foram mostradas até agora, recebendo como parâmetro o número de épocas que o perceptron será executado, onde cada época representa a passagem de todo o *dataframe* pela rede, a função de ativação que será utilizada, a função de custo, os conjuntos de treinamento e teste e a taxa de aprendizado.

[176]:
```
def treinar(epocas, f_ativacao, f_custo, pesos, x_treinamento, y_treinamento,␣
↪x_teste, y_teste,
            tx_aprendizado):
```

```
   execucoes = 0
   precisoes_treinamento = [] # convergência da base de treinamento ao longo␣
↪das épocas
   precisoes_teste = [] # convergência da base de teste ao longo das épocas
   melhores_pesos = [] # registra o melhor conjunto de pesos para a validação␣
↪posterior
   melhor_matriz_treinamento = [] # matriz confusão de treinamento
   melhor_matriz_teste = [] #  matriz confusão de teste

   while execucoes < epocas: # parada ao executar todas as épocas.
       precisao = 0
       iteracao = 0
       valores_previstos = []
       # x_treinamento = x_treinamento.sample(frac=1).reset_index(drop=True) #␣
↪embaralhar os valores dos previsores, por que sem isso, podemos ter sempre␣
↪uma ordem fixa de ajuste de pesos, prejudicando a rede

       for i in x_treinamento.values: # percorre cada registro individualmente
           entradas = i
           soma = somatoria(entradas, pesos)

           # a ativacao retorna qual dos 3 neurônios de saíde foram excitados␣
↪e também o
           # valor real da ativação, para calculo do erro.
           neuronio_excitado, valor_ativacao = f_ativacao(soma)
           valores_previstos.append(neuronio_excitado)

           erro, valor_erro = f_custo(y_treinamento[iteracao],␣
↪neuronio_excitado,
                                      valor_ativacao)
           # atualiza os pesos em caso de erro
           if erro == True:
               count = 0
               # percorre cada coluna para atualizar o peso e o bias
               for i in entradas:
                   if count == len(entradas) - 1:
                       novo_peso = atualizar_bias(i, pesos[count], valor_erro,␣
↪tx_aprendizado)
                   else:
                       novo_peso = atualizar_peso(i, pesos[count], valor_erro,␣
↪tx_aprendizado)

                   pesos[count] = novo_peso
                   count += 1
           else:
```

```
                precisao += 100 / len(x_treinamento) # aumenta precisão␣
↪gradativamente

            iteracao += 1

        precisoes_treinamento.append(precisao) # registra precisão ao fim da␣
↪época
        melhor_matriz_treinamento = get_matriz_confusao(y_treinamento,␣
↪valores_previstos) if precisoes_treinamento[execucoes] >=␣
↪max(precisoes_treinamento) else melhor_matriz_treinamento
        melhores_pesos = pesos.copy() if precisoes_treinamento[execucoes] >=␣
↪max(precisoes_treinamento) else melhores_pesos

        teste_rede = testar(pesos, x_teste, y_teste, f_ativacao, f_custo)
        precisoes_teste.append(teste_rede[0])
        melhor_matriz_teste = teste_rede[1] if precisoes_teste[execucoes] >=␣
↪max(precisoes_teste) else melhor_matriz_teste
        execucoes += 1

    return precisoes_treinamento, precisoes_teste, melhores_pesos,␣
↪melhor_matriz_treinamento, melhor_matriz_teste
```

2.4.9 Método de teste

Ao final de cada época a rede teve seu desempenho avaliado na base de teste, isso foi feito no método *testar* onde foi utilizado os pesos ajustados no treinamento apenas para pegar os acertos e erros no conjunto de teste.

[177]:
```
def testar(pesos, x_previsores, y_classe, f_ativacao, f_custo):
    precisao = 0
    iteracao = 0
    valores_previstos = [] # armazena os valores previstos para cada registro

    for i in x_previsores.values:
        entradas = i
        soma = somatoria(entradas, pesos)

        neuronio_excitado, valor_ativacao = f_ativacao(soma)
        valores_previstos.append(neuronio_excitado)

        erro, valor_erro = f_custo(y_classe[iteracao], neuronio_excitado,␣
↪valor_ativacao)

        # faz a contagem da precisão, incrementando por acerto baseado no total␣
↪de registros
        if erro == 0:
```

```
            precisao += 100 / len(x_previsores)

        iteracao += 1

    matriz_confusao = get_matriz_confusao(y_classe, valores_previstos)

    return precisao, matriz_confusao
```

2.4.10 Exibindo os resultados

Os resultados de precisão, média, desvio padrão e os gráficos de convergência são exibidos nos método abaixo de *exibir_resultados* e *plotar_convergencia*.

[178]:
```
def exibir_resultados(precisao_treinamento, precisao_teste, resultado_final):
    print('Melhor precisão de treinamento', max(precisao_treinamento))
    print('Melhor precisão de teste', max(precisao_teste))
    print('Melhor precisão de validação', max(resultado_final))
    print('Média precisão de treinamento', np.mean(precisao_treinamento))
    print('Média precisão de teste', np.mean(precisao_teste))
    print('Média precisão de validação', np.mean(resultado_final))
    print('Desvio Padrão precisão de treinamento', np.std(precisao_treinamento))
    print('Desvio Padrão precisão de teste', np.std(precisao_teste))
    print('Desvio Padrão precisão de validação', np.std(resultado_final))
```

[179]:
```
def plotar_convergencia(precisao_treinamento, precisao_teste):
    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 8)) # iniciar a␣
 ↪figura
    # plotar a figura de treinamento
    axes[0].plot(precisao_treinamento, color = 'blue')
    axes[0].legend(['Treinamento'])
    # plotar a figura de teste
    axes[1].plot(precisao_teste, color = 'orange')
    axes[1].legend(['Teste'])

    plt.xlabel('Épocas')
    plt.ylabel('Precisão')
    plt.show()
```

2.5 Testes no perceptron

Uma vez que a rede perceptron foi criada basta testar seus resultados com diferentes parâmetros. Para isso, foi criado a função genérica chamada *executar_perceptron*, que recebe os parâmetros desejados e treina a rede.

Cada vez que o método *executar_perceptron* é chamado, o algoritmo é inicializado 30 vezes, sem nenhuma mudança nos parâmetros, apenas gerando um conjunto inicial de pesos diferente para

cada inicialização e também uma partição nova dos dados, pois ambos são aleatórios.

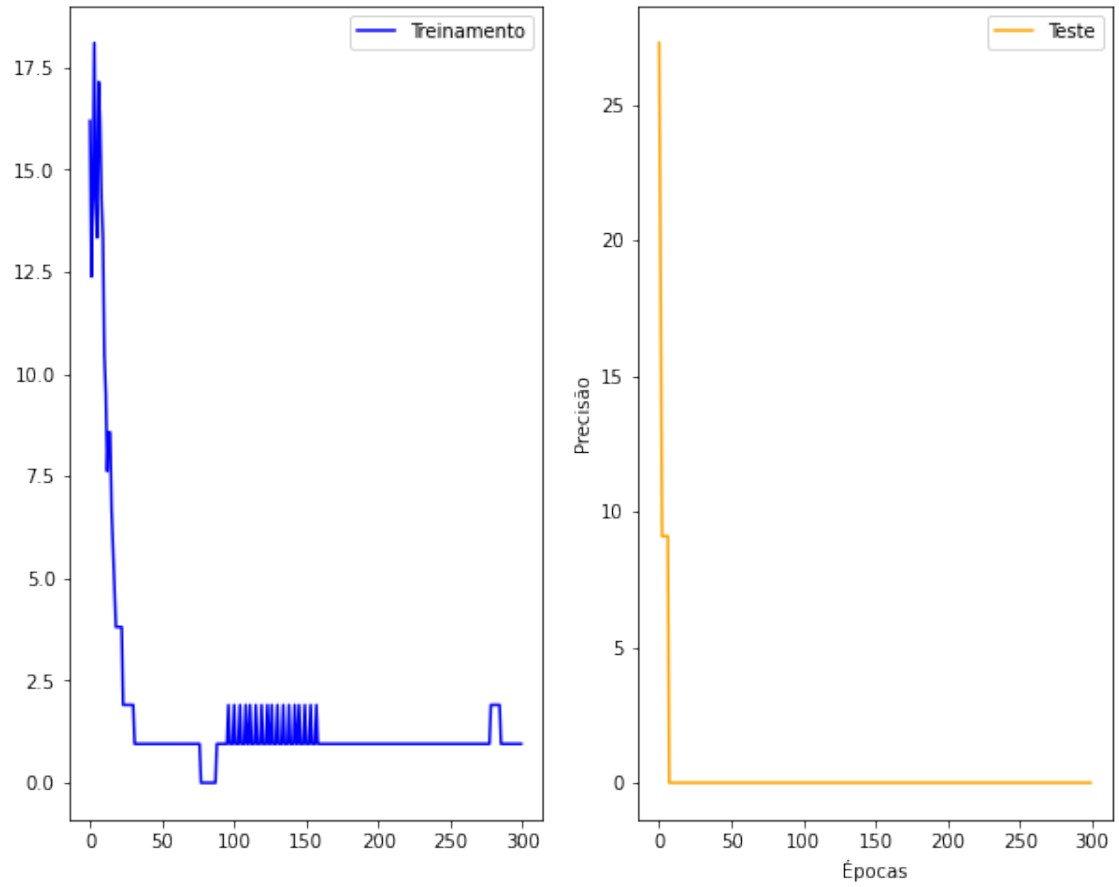
Isso é feito porque as redes neurais são algoritmos extremamente estocásticos, o que faz com que sejam inclusive evitados em algumas áreas, portanto é interessante tentar treinar a rede várias vezes para capturar seus melhores resultados.

O método *executar_perceptron* também registra o desempenho, matriz de confusão e convergência da rede durante as iterações, além de retornar os valores máximos ao final, para que possamos usar posteriormente na otimização de parâmetros.

[180]:
```
def executar_perceptron(funcao_ativacao, funcao_custo, epocas, dominio_pesos =␣
↪[0, 1],
                        tx_aprendizado = 0.001):
    # os arrays servem para registrar os valores de cada inicialização da rede
    convergencia_treinamento = [0]
    convergencia_teste = [0]
    precisao_treinamento = []
    precisao_teste = []
    resultado_final = []
    matriz_confusao_treinamento = []
    matriz_confusao_teste = []
    matriz_confusao_validacao = []
    start_time = time.time() # tempo de execução

    for i in range(30): # 30 execuções
        x_treinamento, y_treinamento, x_teste, y_teste, \
        x_validacao, y_validacao = dividir_dataframe(previsores, classe, 0.7, 0.
↪15, 0.15)

        pesos = inicializar_pesos(dominio_pesos)

        treinamento = treinar(epocas, funcao_ativacao, funcao_custo, pesos,␣
↪x_treinamento,
                                      y_treinamento, x_teste, y_teste,␣
↪tx_aprendizado)

        # É salvo apenas o melhor resultado da convergência, para plotar um␣
↪único gráfico
        convergencia_treinamento = treinamento[0] if max(treinamento[0]) >= \
                                   max(convergencia_treinamento) else␣
↪convergencia_treinamento

        convergencia_teste = treinamento[1] if max(treinamento[1]) >=␣
↪max(convergencia_teste) \
                                        else convergencia_teste

        precisao_treinamento.append(max(treinamento[0]))
        precisao_teste.append(max(treinamento[1]))
```

```
        # avaliação do algoritmo ao final do treinamento na base de validação
        teste_final = testar(treinamento[2], x_validacao, y_validacao,
                                    funcao_ativacao, funcao_custo)
        resultado_final.append(teste_final[0])
        # Salvamos apenas a melhor matriz de confusão, para ser exibida
        matriz_confusao_treinamento = treinamento[3] if max(treinamento[0]) >=␣
↪max(precisao_treinamento) else matriz_confusao_treinamento
        matriz_confusao_teste = treinamento[4] if max(treinamento[1]) >=␣
↪max(precisao_teste) else matriz_confusao_teste
        matriz_confusao_validacao = teste_final[1] if teste_final[0] >=␣
↪max(resultado_final) else matriz_confusao_validacao

    plotar_convergencia(convergencia_treinamento, convergencia_teste)
    exibir_resultados(precisao_treinamento, precisao_teste, resultado_final)
    print('Matriz de confusão de treinamento:\n', matriz_confusao_treinamento)
    print('Matriz de confusão de teste:\n', matriz_confusao_teste)
    print('Matriz de confusão de validação:\n', matriz_confusao_validacao)
    print("Tempo de execução: %s Segundos" % (time.time() - start_time))
```

[49]:
```
executar_perceptron(funcao_ativacao_step, funcao_custo, 300)
```



```
Melhor precisão de treinamento 18.095238095238095
Melhor precisão de teste 27.272727272727277
Melhor precisão de validação 17.391304347826086
Média precisão de treinamento 5.206349206349206
Média precisão de teste 6.969696969696968
Média precisão de validação 3.623188405797101
Desvio Padrão precisão de treinamento 4.879397093800851
Desvio Padrão precisão de teste 7.490428542005015
Desvio Padrão precisão de validação 4.773817128116117
Matriz de confusão de treinamento:
 []
Matriz de confusão de teste:
 []
Matriz de confusão de validação:
 []
Tempo de execução: 54.61538052558899 Segundos
```

2.5.1 Resultados iniciais e ZeroR

Os testes inicias mostram uma precisão extremamente baixa, algo está definitivamente se comportando mal no algoritmo. Podemos ter certeza que o culpado é a implementação do algoritmo e não a base de dados devido a análise exploratória que foi feita anteriormente, revelando as características linearmente separáveis que deveriam proporcionar uma precisão maior que a encontrada até agora.

O "Zero R" pode ser utilizado para saber a precisão mínima que o algoritmo deve ter para ser considerado melhor do que não usar algoritmo algum. A precisão do zeroR, considerado o limiar mínimo, é dado pela proporção de registros da classe majoritária no *dataframe*.

Nesse *dataframe* as classes estão **normalmente distribuídas** em 3 partes iguais, fazendo com que a precisão majoritária sem algoritmos seja de **33%**, portanto, qualquer precisão inferior a isso torna o uso do algoritmo injustificável.

Todavia, ainda que nosso algoritmo seja considerado útil com 34% de acerto, ainda está longe de algo desejável.

2.6 Melhorando os resultados

Os passos a seguir foram usados como meio de melhorar a precisão da rede perceptron.

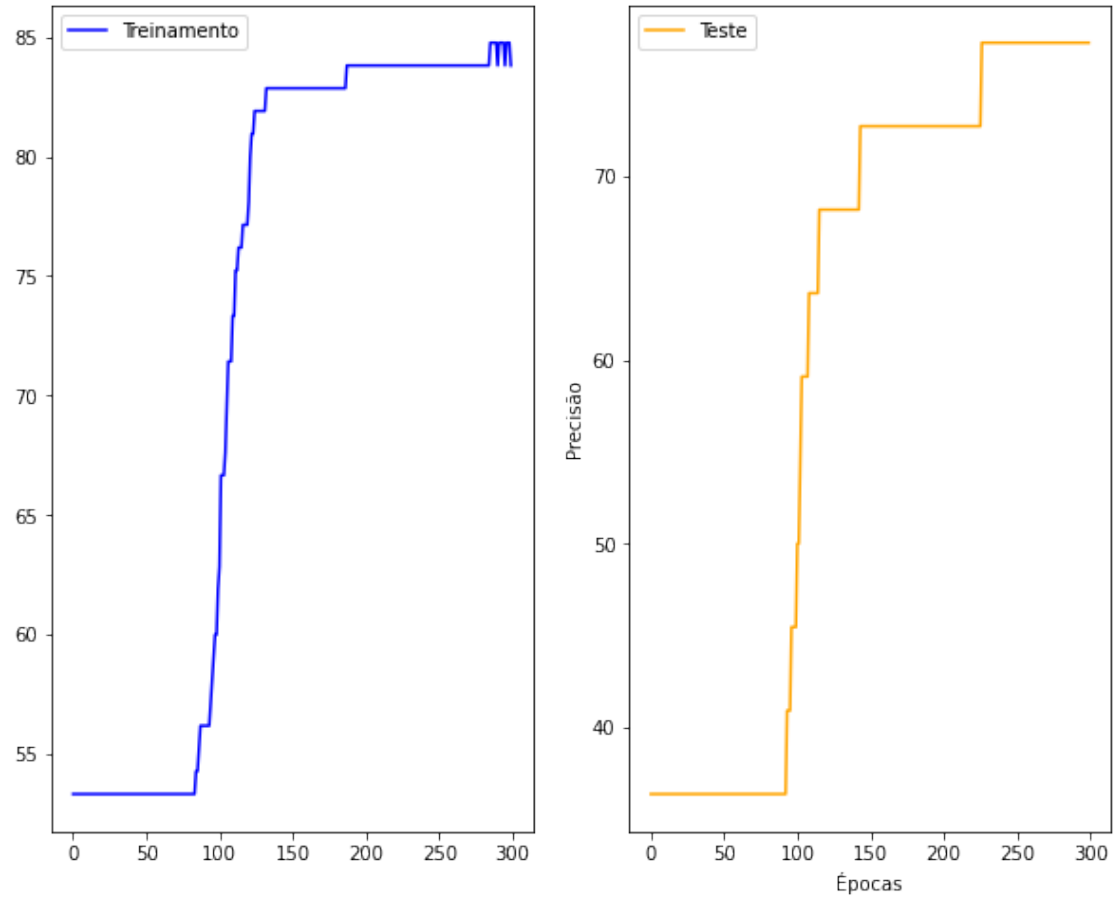
2.6.1 Função de ativação sigmoid

Como foi dito anteriormente, utilizar a função degrau para ativação pode ocasionar o aumento dos erros, devido a possibilidade de excitar vários neurônios ao passar pelo limiar especificado. Portanto foi criado uma nova função de ativação baseado na fórmula da sigmoid.

Com isso, os valores ficam em um intervalo de 0 e 1 e o neurônio escolhido para ser excitado é aquele com o maior valor, dessa forma, apenas um neurônio é excitado por vez

[181]:
```
def funcao_ativacao_sigmoid(soma):
    valor_ativacao = list(1 / (1 + math.e ** -soma))
    index_excitacao = valor_ativacao.index(max(valor_ativacao))# pegar neurônio␣
↪com maior valor
    neuronio_excitado = [0] * len(soma) # zerar os valores dos neurônios de␣
↪saída
    neuronio_excitado[index_excitacao] = 1 # definir o valor 1 para o neurônio␣
↪com maior valor

    return neuronio_excitado, valor_ativacao
```

[51]:
```
executar_perceptron(funcao_ativacao_sigmoid, funcao_custo, 300)
```




```
Melhor precisão de treinamento 84.76190476190457
Melhor precisão de teste 77.27272727272728
Melhor precisão de validação 78.26086956521739
```

```
Média precisão de treinamento 46.66666666666661
Média precisão de teste 45.9090909090909
Média precisão de validação 46.23188405797102
Desvio Padrão precisão de treinamento 19.123657749350244
Desvio Padrão precisão de teste 20.94035870250392
Desvio Padrão precisão de validação 20.284160691614606
Matriz de confusão de treinamento:
 [[33  2  0]
 [ 0 36  4]
 [ 0 10 20]]
Matriz de confusão de teste:
 [[9 0 0]
 [0 3 0]
 [0 5 5]]
Matriz de confusão de validação:
 [[5 1 0]
 [0 6 1]
 [0 3 7]]
Tempo de execução: 74.09392404556274 Segundos
```

A aplicação da função sigmoid gerou melhorias consideráveis nos resultados. Dessa vez, foi possível gerar as matrizes de confusão das classes 0, 1 e 2 respectivamente, onde a linha é o valor real e a coluna o valor que foi previsto.

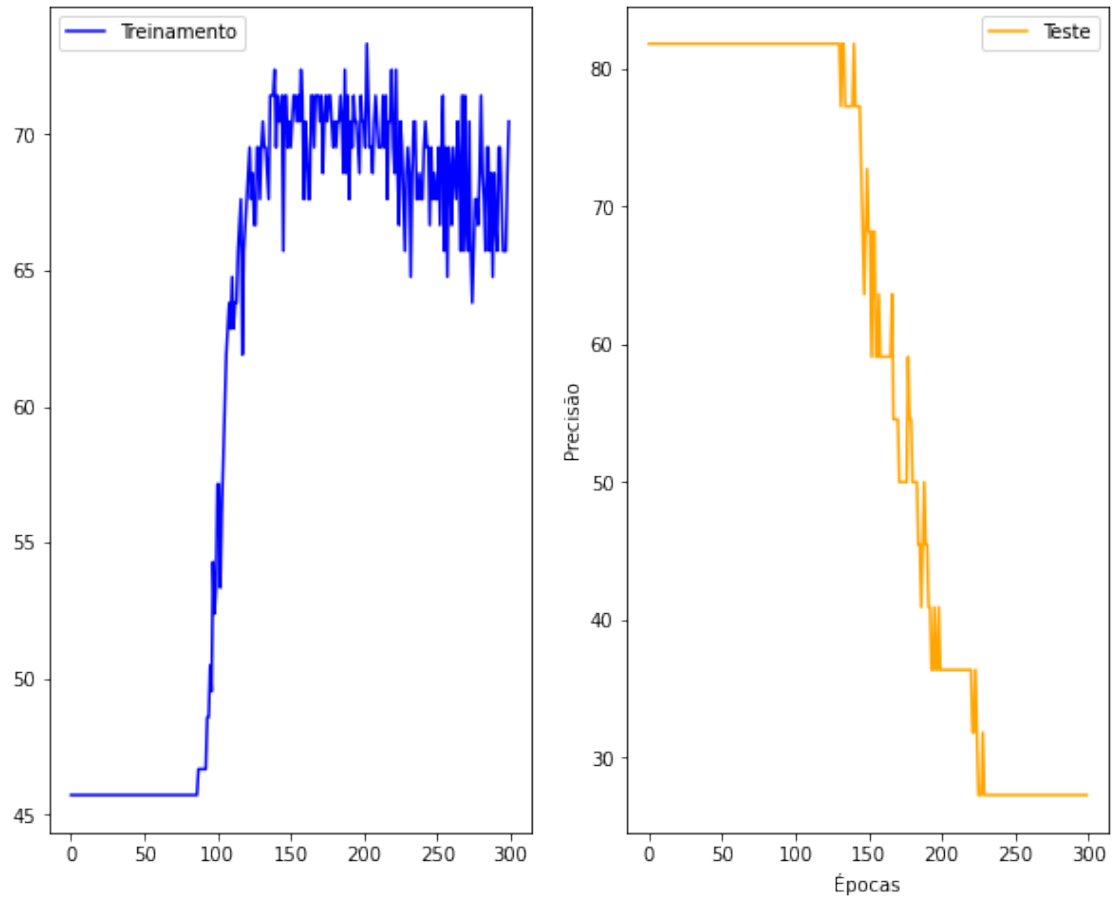
Como é possível observar, a maior parte das classificações incorretas está entre a classe de iris versicolor e virginica, representado pelos valores 1 e 2 segundo o dicionário de classe criado anteriormente. Esse comportamente já era esperado, uma vez que suas características se misturam mais no espaço, dificultando o trabalho da rede perceptron.

Na matriz de teste por exemplo, é possível notar que 50% das íris virginicas foram classificadas como sendo versicolor, portanto ainda é preciso otimizar mais os parâmetros do algoritmo a fim de melhorar sua capacidade de divisão no espaço de características.

Um outro parâmetro que pode ser otimizado é o domínio dos pesos, aumentando a precisão da busca pelos melhores pesos, que é refletido por sua vez em um maior número de oscilações nos gráficos de precisão.

Um domínio mais próximo a zero faz com que os valores comecem pequenos e aumentem gradativamente (baseado também no tamanho da taxa de aprendizado), fazendo uma busca mais aprofundada pelo ótimo global.

[53]:
```
executar_perceptron(funcao_ativacao_sigmoid, funcao_custo, 300, [-0.005, 0.005])
```




```
Melhor precisão de treinamento 73.33333333333319
Melhor precisão de teste 81.81818181818183
Melhor precisão de validação 78.26086956521739
Média precisão de treinamento 50.317460317460245
Média precisão de teste 51.06060606060607
Média precisão de validação 46.23188405797102
Desvio Padrão precisão de treinamento 15.795525134053552
Desvio Padrão precisão de teste 17.979301434523588
Desvio Padrão precisão de validação 20.408040884544054
Matriz de confusão de treinamento:
 [[30  3  0]
 [ 2 38  0]
 [ 1 22  9]]
Matriz de confusão de teste:
 [[6 0 0]
 [2 3 2]
 [0 0 9]]
Matriz de confusão de validação:
 [[12  0  1]
```

```
 [ 0  5  2]
 [ 0  2  1]]
Tempo de execução: 69.85689520835876 Segundos
```

2.6.2 Mean Squared Error e Root Mean Squared error

A função de custo que está sendo utilizada atualmente é a mais simples possível, onde comparamos a diferença entre a previsão atual com o valor esperado. Existem outras fórmulas um pouco mais completas que possuem um nível de precisão maior em estimar os custos.

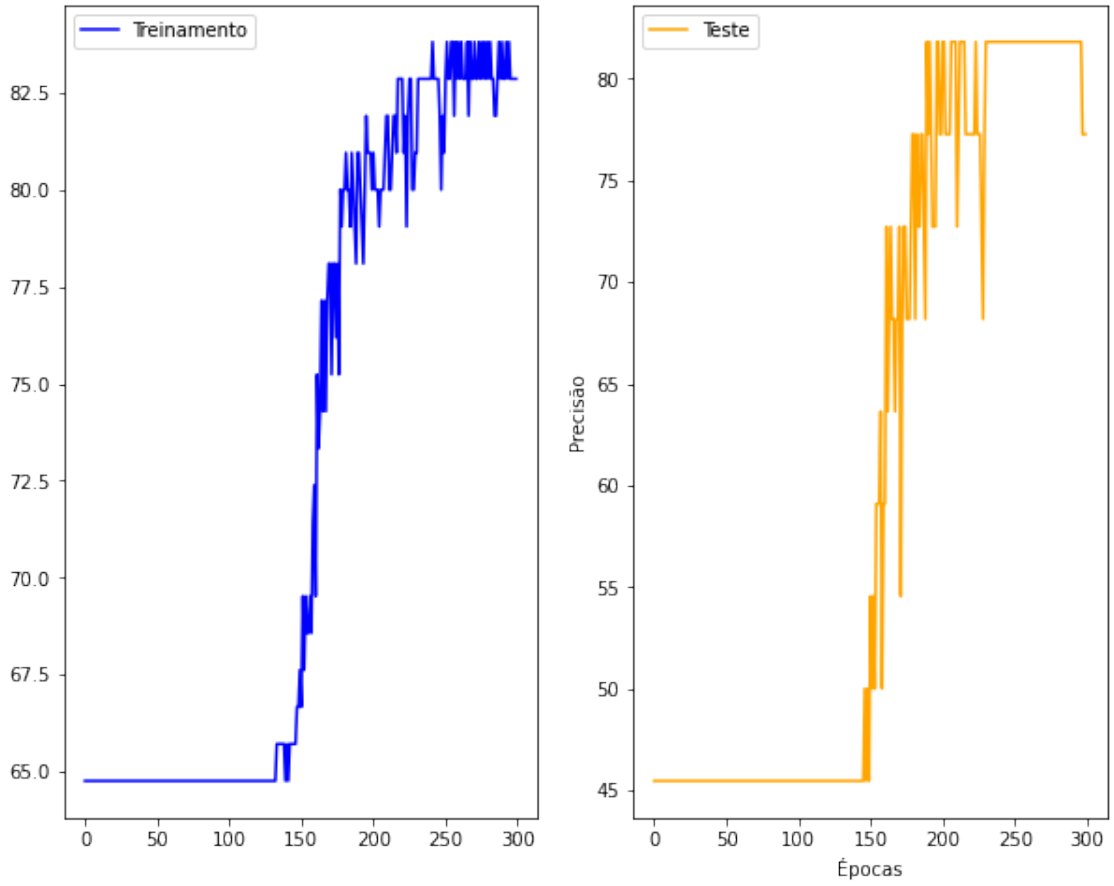
A primeira delas é a função Mean Squared Error, representada no método *funcao_custo_mse*, que além de calcular o valor correto subtraído do valor previsto também eleva o resultado da subtração ao quadrado e os soma, gerando um valor escalar que pune mais erros maiores, deixando eles mais expressivos.

A Root Mean Squared Error, representada pelo método *funcao_custo_rmse* segue a mesma formula, porém submetendo os resultados finais a uma raiz quadrada.

[182]:
```
def funcao_custo_mse(valor_correto, valor_previsto, valor_ativacao):
    erro = valor_correto != valor_previsto

    valor_erro = list(abs(np.array(valor_correto) - np.array(valor_ativacao)))
    erro_quadratico = list(map(lambda x: math.pow(x, 2), valor_erro))
    soma_erro_quadratico = sum(erro_quadratico)

    return erro, soma_erro_quadratico
```

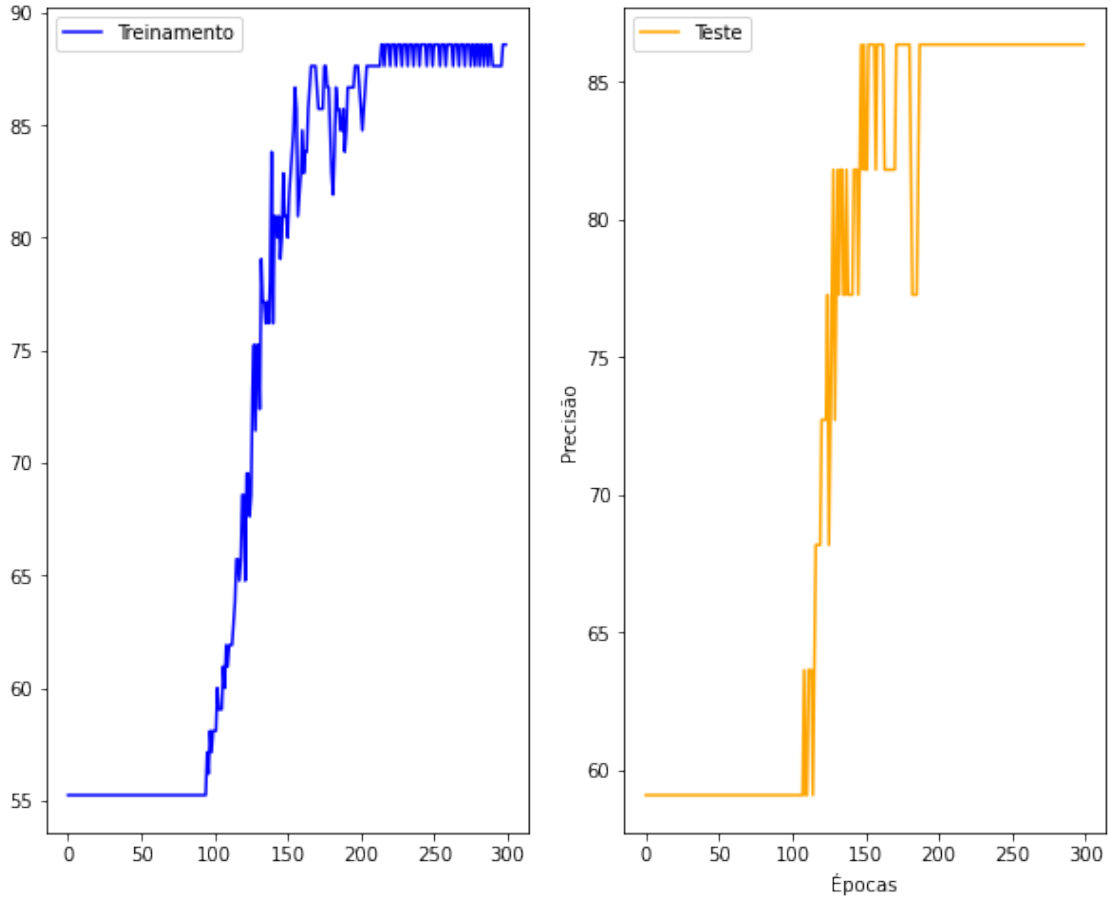
[183]:
```
def funcao_custo_rmse(valor_correto, valor_previsto, valor_ativacao):
    erro, valor_erro = funcao_custo_mse(valor_correto, valor_previsto,␣
 ↪valor_ativacao)

    return erro, math.sqrt(valor_erro)
```

[185]:
```
executar_perceptron(funcao_ativacao_sigmoid, funcao_custo_mse, 300, [-0.05, 0.
 ↪05])
executar_perceptron(funcao_ativacao_sigmoid, funcao_custo_rmse, 300, [-0.05, 0.
 ↪05])
```




```
Melhor precisão de treinamento 83.80952380952363
Melhor precisão de teste 81.81818181818183
Melhor precisão de validação 69.56521739130436
Média precisão de treinamento 41.68253968253963
Média precisão de teste 41.21212121212122
Média precisão de validação 37.39130434782609
Desvio Padrão precisão de treinamento 19.222504028448192
Desvio Padrão precisão de teste 20.0183570483803
Desvio Padrão precisão de validação 18.77133316949818
Matriz de confusão de treinamento:
 [[29  1  0]
 [ 9 30  3]
 [ 0  4 29]]
Matriz de confusão de teste:
 [[11  0  0]
 [ 0  2  0]
 [ 0  4  5]]
Matriz de confusão de validação:
 [[3 3 0]
```

```
 [0 8 2]
 [0 2 5]]
Tempo de execução: 69.14211058616638 Segundos
```




```
Melhor precisão de treinamento 88.57142857142837
Melhor precisão de teste 86.36363636363637
Melhor precisão de validação 95.65217391304346
Média precisão de treinamento 49.99999999999993
Média precisão de teste 47.575757575757585
Média precisão de validação 48.40579710144928
Desvio Padrão precisão de treinamento 17.221600456920555
Desvio Padrão precisão de teste 18.247356627060118
Desvio Padrão precisão de validação 21.410069927674652
Matriz de confusão de treinamento:
 [[35  2  0]
 [ 1 33  0]
 [ 1  8 25]]
Matriz de confusão de teste:
 [[6 1 0]
```

```
 [0 8 0]
 [0 2 5]]
Matriz de confusão de validação:
 [[5 1 0]
 [0 8 0]
 [0 0 9]]
Tempo de execução: 72.14907193183899 Segundos
```

As duas funções de custo se saíram bem na busca dos resultados, sendo inclusive superiores ao método anterior mais simplificado.

2.6.3 Problema do gradiente explodindo ou desaparecendo

Um problema enfrentado foi o dos pesos explodindo ou sumindo. No primeiro caso, o valor dos pesos aumentava de forma exponencial, chegando a valores próximos dos 2000 mil por conexão sináptica. Isso gerou uma deteorização na precisão, fazendo com que os valores iniciais da rede sejam sempre os mais altos.

Já no gradiente sumindo, o problema foi uma taxa de aprendizado e um conjunto de pesos muito pequeno, fazendo com que a rede se mantenha uma linha reta, incapaz de aprender nada.

Para resolver esse problema foram feitos vários testes de diferentes pesos e taxas de aprendizado a fim de conseguir chegar a um meio termo, pois esses dois parâmetros são os principais responsáveis por fazer que isso ocorra.

2.6.4 Atualização dos pesos por época

As redes neurais multicamadas geralmente atualizam os pesos da rede após passar todos os registros pela rede, ou pelo menos uma parte dos registros (batch). A atualização por época possui como vantagem um tempo de processamento menor, pois executa todo o processo na rede de uma vez só, sem precisar passar registro por registro, menos sensibilidade à dados ruidosos, pois generaliza os resultados de toda a rede e também um código de implementação mais simples (com menos linhas). Todavia também possui algumas desvantagens, como maior uso de memória, pois precisa carregar todos os registros, o que pode ser impossível quando se trabalha com uma massa maior de dados além de que essa generalização dos resultados do *dataset* pode acabar levando à convergência a um ótimo local.

Pensando nisso, foi implementado uma versão que trabalha por épocas da rede perceptron para testar seu comportamento nos problemas aqui trabalhados. Foram necessárias algumas mudanças nos métodos já apresentados anteriormente.

Primeiro, os valores de classe e previsores foram reiniciados.

[37]:
```
previsores = dataframe.iloc[:, 0:4]
previsores = previsores.apply(lambda row: normalizacao_z_score(row))
previsores['bias'] = 1
classe = dataframe['class']
classe = classe.apply(lambda row: transformar_categorico_em_numerico(row,␣
 ↪dict_classes))
```

Isso é necessário pois a classe foi codificada em um novo formato de matriz ao invés de array, para trabalhar com as operações matemáticas de uma vez só em todos os registros. Para isso, foram feitas alterações no método de *codificar_classe*.

[38]:
```
def codificar_classe_epoca():
    classe_codificada = {}

    array_classe = np.array([[1]  + ([0] * (len(classe.unique()) - 1)) ]) #␣
 ↪estrutura de matriz

    count = 1
    classe_codificada[0] = array_classe.copy()

    for i in range(len(classe.unique()) - 1):
        array_classe[0][count - 1] = 0
        array_classe[0][count] = 1
        classe_codificada[count] = array_classe.copy()
        count += 1

    return classe_codificada
```

Uma vez criado a estrutura de classe por matriz, é criado a variável *classe_nova* que vai transformar todas as classes do problema no formato de matriz.

[39]:
```
classe_codificada = codificar_classe_epoca()
classe_nova = []

for i in classe: # percorre as classes do dataframe
    classe_nova.append(classe_codificada[i])

classe_nova = np.array(classe_nova).reshape(len(classe), 3) # redimensiona para␣
 ↪criar a matriz
print(classe_nova.shape)
```

```
(150, 3)
```

A classe agora, ao invés de um array [0,1,0] é uma matriz com três colunas, uma para cada posição do array, como podemos ver no *print* do método *shape*, nos indicando as 150 linhas (registros) e três colunas (quantidade de classes).

Foi feito uma pequena modificação no método *dividir_dataframe*, porém não foi criado um novo método, apenas utilizamos um parâmetro adicional nesse caso.

A função sigmoid também recebe algumas modificações de sintax, pois ao invés de pegar o valor máximo de apenas um array e excitá-lo, o processo é feito de uma vez só no conjunto inteiro.

[40]:
```
def funcao_ativacao_sigmoid_epoca(soma):
    valor_ativacao = 1 / (1 + math.e ** -soma)
    index_excitacao = np.argmax(valor_ativacao, 1)
```

```
    count = 0
    neuronios_excitado = valor_ativacao.copy()

    for i in index_excitacao:
        neuronios_excitado[count] = 0
        neuronios_excitado[count][i] = 1
        count += 1

    return neuronios_excitado, valor_ativacao
```

A função de custo utilizada foi a de Mean Squared Error, pois mostrou um bom desempenho nos testes. Foram feitas algumas pequenas adaptações para trabalhar com a operação matemática em todos os registros, mas a principal diferença é que na atualização por época, ao invés de contabilizar a precisão incrementalmente, ela é calculada de uma vez só, baseado no número de acertos em todos os registros.

[41]:
```
def funcao_custo_mse_epoca(valor_correto, valor_previsto, valor_ativacao):
    erro = list(abs(np.array(valor_correto) - np.array(valor_previsto)))
    valor_erro = list(abs(np.array(valor_correto) - np.array(valor_ativacao)))

    acerto = 0
    for i in erro:
        if sum(i) == 0: # verifica se o registro está correto, ou seja, soma de␣
↪erro é igual a 0.
            acerto += 1 # incrementa um no acerto

    erro_quadratico = list(map(lambda x: x**2, valor_erro))
    erro_quadratico_medio = sum(erro_quadratico) / len(valor_correto)

    return sum(erro), acerto, sum(erro_quadratico_medio)
```

A atualização dos pesos é a soma do valor atual do peso com o produto da multiplicação entre taxa de aprendizado, entrada e erro. Nesse caso, como trabalhamos com todas as entradas de uma vez só, é feito a média desse produto.

Essa média é a responsável por fazer a generalização de todas as entradas em um valor que será adicionado ao novo peso, sendo uma vantagem e ao mesmo tempo desvantagem dessa abordagem, como já foi discutido

[42]:
```
def atualizar_peso_epoca(entrada, peso, erro, tx_aprendizado):
    novo_peso = peso + np.mean((tx_aprendizado * entrada * erro))
    return novo_peso
```

A matriz de confusão sofreu uma pequena modificação, pois os valores não precisam serem convertidos para um numpy array, uma vez que aqui está sendo trabalhado com a estrutura de matriz.

[43]:
```
def get_matriz_confusao_epoca(valor_correto, valor_previsto):
    previsao = valor_previsto.copy()
    previsao = np.where(previsao == 1)[1]

    correto = valor_correto.copy()
    correto = np.where(correto == 1)[1]

    matriz_confusao = confusion_matrix(correto, previsao)

    return matriz_confusao
```

O método de testar fica mais simplificado, pois não precisa de loops em cada registro, além de já obter os acertos na própria função de ativação

[44]:
```
def testar_epoca(pesos, x_previsores, y_classe, f_ativacao, f_custo):
    entradas = x_previsores.values
    soma = somatoria(entradas, pesos)

    neuronio_excitado, valor_ativacao = f_ativacao(soma)
    matriz_confusao = get_matriz_confusao_epoca(y_classe, neuronio_excitado)

    erro, acertos, valor_erro = f_custo(y_classe, neuronio_excitado,␣
↪valor_ativacao)

    return acertos / len(x_previsores), matriz_confusao
```

Assim como o teste, o treinamento acaba ficando mais simples, lidando com todos os dados de uma vez só

[45]:
```
def treinar_epoca(epocas, f_ativacao, f_custo, pesos, x_treinamento,␣
↪y_treinamento,
                                    x_teste, y_teste, tx_aprendizado):
    execucoes = 0
    precisoes_treinamento = []
    precisoes_teste = []
    melhores_pesos = []
    melhor_matriz_treinamento = []
    melhor_matriz_teste = []

    while execucoes < epocas:
        entradas = x_treinamento.values
        soma = somatoria(entradas, pesos)

        neuronio_excitado, valor_ativacao = f_ativacao(soma)

        erro, acertos, valor_erro = f_custo(y_treinamento, neuronio_excitado,␣
↪valor_ativacao)
```

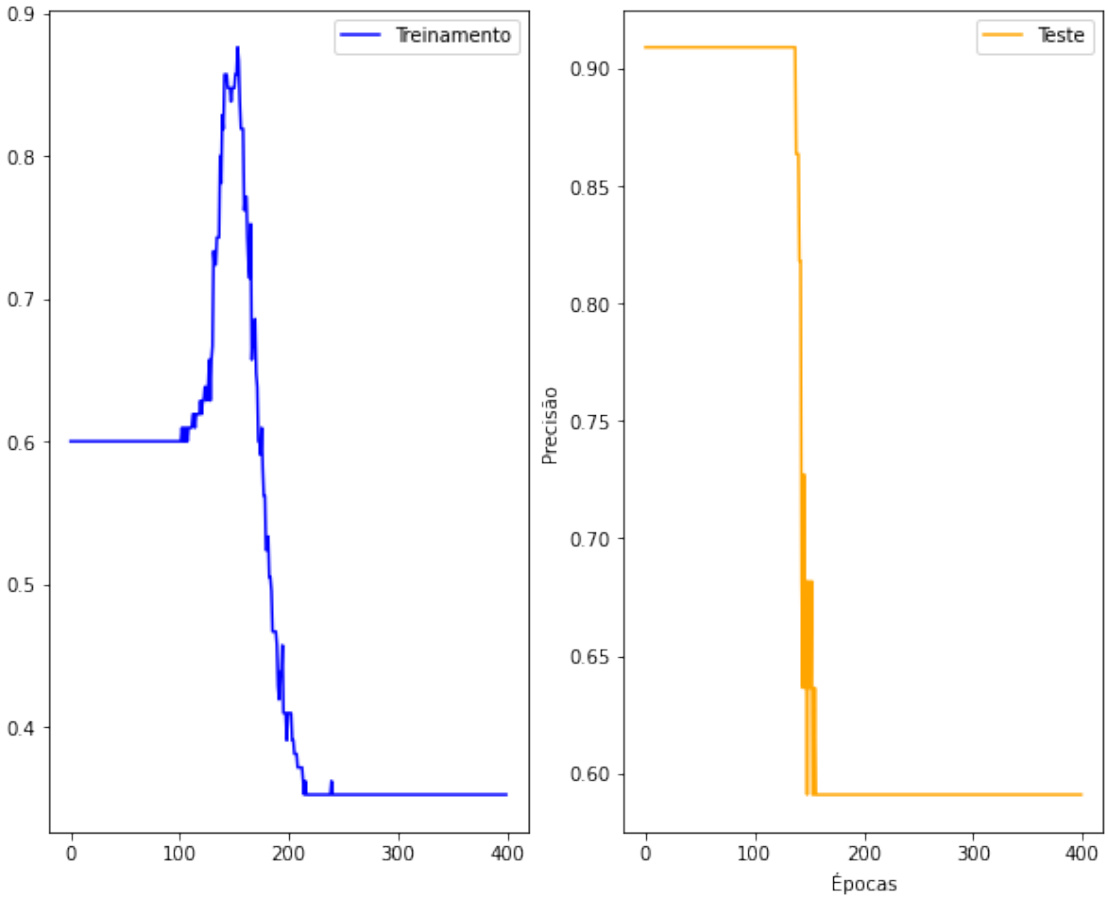
```
        count = 0
        precisoes_treinamento.append(acertos / len(x_treinamento))
        melhor_matriz_treinamento = get_matriz_confusao_epoca(y_treinamento,␣
↪neuronio_excitado) if precisoes_treinamento[execucoes] >=␣
↪max(precisoes_treinamento) else melhor_matriz_treinamento
        melhores_pesos = pesos.copy() if precisoes_treinamento[execucoes] >=␣
↪max(precisoes_treinamento) else melhores_pesos

        for i in range(entradas.shape[1]): # o for tem que atualizar cada peso␣
↪da camada
            if i == 4:
                novo_peso = atualizar_bias(entradas[:, i], pesos[i],␣
↪valor_erro, tx_aprendizado)
            else:
                novo_peso = atualizar_peso_epoca(entradas[:, i], pesos[i],␣
↪valor_erro, tx_aprendizado)
            pesos[count] = novo_peso
            count += 1

        teste_rede = testar_epoca(pesos, x_teste, y_teste, f_ativacao, f_custo)
        precisoes_teste.append(teste_rede[0])
        melhor_matriz_teste = teste_rede[1] if precisoes_teste[execucoes] >=␣
↪max(precisoes_teste) else melhor_matriz_teste
        execucoes += 1

    return precisoes_treinamento, precisoes_teste, melhores_pesos,␣
↪melhor_matriz_treinamento, melhor_matriz_teste
```

O novo método de executar perceptron vai ser muito parecido com o anterior, porém irá chamar todos os novos métodos que foram criados e executar a rede por épocas.

[46]:
```
def executar_perceptron_epoca(funcao_ativacao, funcao_custo, epocas,␣
↪dominio_pesos = [0, 1],
                        tx_aprendizado = 0.1, mostrar_resultados = True):

    convergencia_treinamento = [0]
    convergencia_teste = [0]
    precisao_treinamento = []
    precisao_teste = []
    resultado_final = []
    matriz_confusao_treinamento = []
    matriz_confusao_teste = []
    matriz_confusao_validacao = []
    start_time = time.time()

    for i in range(30):
```

```
        pesos = inicializar_pesos(dominio_pesos) # Alterando os pesos em cada␣
↪inicialização
        x_treinamento, y_treinamento, x_teste, y_teste, x_validacao,␣
↪y_validacao = dividir_dataframe(previsores, classe_nova, 0.7, 0.15, 0.15,␣
↪True)

        treinamento = treinar_epoca(epocas, funcao_ativacao, funcao_custo,␣
↪pesos, x_treinamento, y_treinamento, x_teste, y_teste, tx_aprendizado)

        convergencia_treinamento = treinamento[0] if max(treinamento[0]) >= \
                                   max(convergencia_treinamento) else␣
↪convergencia_treinamento
        convergencia_teste = treinamento[1] if max(treinamento[1]) >=␣
↪max(convergencia_teste) \
                                        else convergencia_teste

        precisao_treinamento.append(max(treinamento[0]))
        precisao_teste.append(max(treinamento[1]))

        teste_final = testar_epoca(treinamento[2], x_validacao, y_validacao,
                                   funcao_ativacao, funcao_custo)

        resultado_final.append(teste_final[0])

        matriz_confusao_treinamento = treinamento[3] if max(treinamento[0]) >=␣
↪max(precisao_treinamento) else matriz_confusao_treinamento
        matriz_confusao_teste = treinamento[4] if max(treinamento[1]) >=␣
↪max(precisao_teste) else matriz_confusao_teste
        matriz_confusao_validacao = teste_final[1] if teste_final[0] >=␣
↪max(resultado_final) else matriz_confusao_validacao

    if mostrar_resultados == True: # condição para caso não tenha interesse em␣
↪plotar gráficos
        plotar_convergencia(convergencia_treinamento, convergencia_teste)
        exibir_resultados(precisao_treinamento, precisao_teste, resultado_final)
        print("Tempo de execução: %s Segundos" % (time.time() - start_time))
        print('Matriz de confusão de treinamento:\n',␣
↪matriz_confusao_treinamento)
        print('Matriz de confusão de teste:\n', matriz_confusao_teste)
        print('Matriz de confusão de validação:\n', matriz_confusao_validacao)

    return max(precisao_treinamento), max(precisao_teste), max(resultado_final)
```

[80]:
```
executar_perceptron_epoca(funcao_ativacao_sigmoid_epoca,␣
↪funcao_custo_mse_epoca, 400, [-0.0005, 0.0005])
```




```
Melhor precisão de treinamento 0.8761904761904762
Melhor precisão de teste 0.9090909090909091
Melhor precisão de validação 0.8695652173913043
Média precisão de treinamento 0.4930158730158731
Média precisão de teste 0.49696969696969695
Média precisão de validação 0.4942028985507247
Desvio Padrão precisão de treinamento 0.1796895505951765
Desvio Padrão precisão de teste 0.18141369147994868
Desvio Padrão precisão de validação 0.18083449121091413
Tempo de execução: 23.791481494903564 Segundos
Matriz de confusão de treinamento:
 [[37  0  0]
 [ 3 22  8]
 [ 0  2 33]]
Matriz de confusão de teste:
 [[13  0  0]
 [ 1  0  1]
 [ 0  0  7]]
Matriz de confusão de validação:
```

```
[[6 0 0]
 [0 5 3]
 [0 0 9]]
```

[80]: `(0.8761904761904762, 0.9090909090909091, 0.8695652173913043)`

2.7 Resultados do treinamento por época

Os resultados de precisão foram bons, chegando em torno dos 90% no teste da rede neural. Nesse caso, os resultados não foram tão diferentes do que executar o perceptron e atualizar os pesos por registro individual, mas aqui valem dois pontos de atenção. O primeiro é que a base de dados da iris não é de um problema muito complexo, além de ter uma quantidade pequena e separável de atributos, portanto atualizar olhando para atributos individualmente ou para todos os atributos de uma vez não fez tanta diferença. O segundo é que, como esperado, a execução do algoritmo por épocas conseguiu se sair até 6 vezes mais rápido do que o anterior, levando menos de 20 segundos para finalizar todas as execuções. Essa velocidade de processamento no possibilita testar uma combinação maior de hiperparâmetros, favorecendo a otimização da rede neural.

2.7.1 Encontrando os melhores parâmetros

Existem algumas formas de encontrar os melhores resultados que um determinado algoritmo pode proporcionar. Uma delas é pelo teste exaustivo de parâmetros, onde são testados todas as combinações de parâmetros a fim de encontrar os melhores resultados.

Algoritmos como as redes neurais que possuem um quantidade maior de parâmetros se beneficiam desse tipo de abordagem, todavia o tempo de execução para esses testes muitas das vezes acaba sendo alto.

Para isso, foi criado o método *buscar_parametros*, que recebe um dicionário com uma lista de parâmetros, e cria uma lista com todos os parâmetros combinados.

Após isso, cada elemento da lista é executado, representando uma diferente possibilidade de combinação de parâmetros. Essa combinação é executada por 30 vezes no método *executar_perceptron*.

Ao final de todas as iterações, vamos ter os resultados finais obtidos bem como os melhores parâmetros para o algoritmo.

[47]:
```python
def buscar_parametros(lista_parametros, executar):
    # cria uma única lista com todos os parâmetros
    parametros = [lista_parametros['custo'],
    lista_parametros['tx_aprendizado'], lista_parametros['pesos']]

    # Combinação de cada um desses parâmetros
    combinacao_parametros = list(itertools.product(*parametros))

    melhores_parametros = []
    melhor_precisao_teste = 0
    melhor_precisao_treinamento = 0
```

```
    melhor_precisao_validacao = 0
    # nesse for os parâmetros são testados
    for i in combinacao_parametros:
        precisao_treinamento, precisao_teste, resultado_final =␣
↪executar(funcao_ativacao_sigmoid_epoca, i[0], 400, [-i[2], i[2]], i[1],␣
↪False)
        # pegando os melhores resultados
        if resultado_final >= melhor_precisao_validacao:
            melhor_precisao_teste = precisao_teste
            melhor_precisao_treinamento = precisao_treinamento
            melhor_precisao_validacao = resultado_final
            melhores_parametros = i

    return melhores_parametros, melhor_precisao_teste,␣
↪melhor_precisao_treinamento, melhor_precisao_validacao
```

[49]:
```
lista_parametros_epoca = { 'custo' : [funcao_custo_mse_epoca],
                      'tx_aprendizado': [0.1, 0.01, 0.0001],
                      'pesos': [0.5, 0.05, 0.005, 0.0005]
}

teste_parametrico = buscar_parametros(lista_parametros_epoca,␣
 ↪executar_perceptron_epoca)
print('Melhores parâmetros', teste_parametrico[0])
print('Melhor precisão teste', teste_parametrico[1])
print('Melhor precisão treinamento', teste_parametrico[2])
print('Melhor precisão validação', teste_parametrico[3])
```

```
Melhores parâmetros (<function funcao_custo_mse_epoca at 0x7f50bb69af28>, 0.01,
0.0005)
Melhor precisão teste 0.8181818181818182
Melhor precisão treinamento 0.6190476190476191
Melhor precisão validação 0.9130434782608695
```

Os melhores parâmetros para esse problema na rede perceptron encontrado pelo algoritmo foi uma taxa de aprendizado de 0.01 e a inicialização dos pesos em 0.0005. Existem outros parâmetros que poderiam ser também otimizados, mas foram deixados de lado por conta do tempo de processamento. É importante notar também que a métrica usada para buscar os melhores parâmetros foi a de precisão na base de validação, chegando a 91%. Nesse caso em específico a precisão não ficou tão alta na base de treinamento, pelo fato do algoritmo buscar otimizar a precisão em apenas uma base, podendo ser uma melhoria futura considerar o cenário todo. Poderia também ser utilizado uma semente gerado fixa para tentar controlar a estocasticidade da rede, todavia como cada teste foi executado 30 vezes esse problema além de ser minimizado também garantiu uma cobertura maior do conjunto de dados, pois em cada execução o conjunto foi reamostrado.

Por fim, podemos concluir que o método de busca de parâmetros consegue automatizar o processo de encontrar os melhores parâmetros ao custo de um tempo de processamento maior. A busca foi feita apenas no algoritmo de atualização por época, por que na versão anterior de atualização de

pesos por registros, levou mais de duas horas e ainda assim a busca não havia terminado, portanto foi interrompida.

Na prática, a busca por melhores parâmetros pode levar dezenas de horas, porém é sempre interessante fazer testes manuais para reduzir a quantidade de combinações de parâmetros e assim reduzir esse tempo.

3 Qual o problema?

Utilizar a base de dados de vinhos contidos aqui: http://archive.ics.uci.edu/ml/datasets/Wine para prever qual o tipo de vinho (1, 2 ou 3) baseado nas suas características.

O arquivo em questão não possuia colunas, assim como no conjunto da iris, portanto eu modifiquei o mesmo para adicionar o nome das colunas na primeira linha e deixá-lo no formato .CSV ao invés do .DATA.

3.1 Anásile exploratória dos dados

Conhecendo um pouco mais dos dados da base de dados.

[50]:
```
dataframe = pd.read_csv('/home/alvaro/Documentos/mestrado/computação bio/redes␣
↪neurais/datasets/wine.csv', header = 0)
dataframe.head()
```

[50]:

	Wine	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	\
0	1	14.23	1.71	2.43	15.6	127	
1	1	13.20	1.78	2.14	11.2	100	
2	1	13.16	2.36	2.67	18.6	101	
3	1	14.37	1.95	2.50	16.8	113	
4	1	13.24	2.59	2.87	21.0	118	

	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	\
0	2.80	3.06	0.28	2.29	
1	2.65	2.76	0.26	1.28	
2	2.80	3.24	0.30	2.81	
3	3.85	3.49	0.24	2.18	
4	2.80	2.69	0.39	1.82	

	Color intensity	Hue	OD280	Proline
0	5.64	1.04	3.92	1065
1	4.38	1.05	3.40	1050
2	5.68	1.03	3.17	1185
3	7.80	0.86	3.45	1480
4	4.32	1.04	2.93	735

[51]:
```
print('Valores nulos:')
print(dataframe.isna().sum())
dataframe.describe()
```

```
Valores nulos:
Wine                    0
Alcohol                 0
Malic acid              0
Ash                     0
Alcalinity of ash       0
Magnesium               0
Total phenols           0
Flavanoids              0
Nonflavanoid phenols    0
Proanthocyanins         0
Color intensity         0
Hue                     0
OD280                   0
Proline                 0
dtype: int64
```

[51]:

	Wine	Alcohol	Malic acid	Ash	Alcalinity of ash	\
count	178.000000	178.000000	178.000000	178.000000	178.000000	
mean	1.938202	13.000618	2.336348	2.366517	19.494944	
std	0.775035	0.811827	1.117146	0.274344	3.339564	
min	1.000000	11.030000	0.740000	1.360000	10.600000	
25%	1.000000	12.362500	1.602500	2.210000	17.200000	
50%	2.000000	13.050000	1.865000	2.360000	19.500000	
75%	3.000000	13.677500	3.082500	2.557500	21.500000	
max	3.000000	14.830000	5.800000	3.230000	30.000000	

	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	\
count	178.000000	178.000000	178.000000	178.000000	
mean	99.741573	2.295112	2.029270	0.361854	
std	14.282484	0.625851	0.998859	0.124453	
min	70.000000	0.980000	0.340000	0.130000	
25%	88.000000	1.742500	1.205000	0.270000	
50%	98.000000	2.355000	2.135000	0.340000	
75%	107.000000	2.800000	2.875000	0.437500	
max	162.000000	3.880000	5.080000	0.660000	

	Proanthocyanins	Color intensity	Hue	OD280	Proline
count	178.000000	178.000000	178.000000	178.000000	178.000000
mean	1.590899	5.058090	0.957449	2.611685	746.893258
std	0.572359	2.318286	0.228572	0.709990	314.907474
min	0.410000	1.280000	0.480000	1.270000	278.000000
25%	1.250000	3.220000	0.782500	1.937500	500.500000

50%	1.555000	4.690000	0.965000	2.780000	673.500000
75%	1.950000	6.200000	1.120000	3.170000	985.000000
max	3.580000	13.000000	1.710000	4.000000	1680.000000

Nesse *dataset* todos os atributos são contínuos, inclusive a classe. Podemos notar que não existe nenhum valor vazio bem como aparentemente nenhum outlier/ruído.

3.2 Normalização dos dados

Assim como no problema da iris, esse dataset também precisa ser normalizando, ainda mais por conter uma quantidade maior de atributos previsores contínuos.

[66]:
```
previsores = dataframe.iloc[:, 1:14]
classe = dataframe['Wine']
```

[67]:
```
previsores = previsores.apply(lambda row: normalizacao_z_score(row))
previsores.head()
```

[67]:

	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	\
0	1.514341	-0.560668	0.231400	-1.166303	1.908522	
1	0.245597	-0.498009	-0.825667	-2.483841	0.018094	
2	0.196325	0.021172	1.106214	-0.267982	0.088110	
3	1.686791	-0.345835	0.486554	-0.806975	0.928300	
4	0.294868	0.227053	1.835226	0.450674	1.278379	

	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	\
0	0.806722	1.031908	-0.657708	1.221438	
1	0.567048	0.731565	-0.818411	-0.543189	
2	0.806722	1.212114	-0.497005	2.129959	
3	2.484437	1.462399	-0.979113	1.029251	
4	0.806722	0.661485	0.226158	0.400275	

	Color intensity	Hue	OD280	Proline
0	0.251009	0.361158	1.842721	1.010159
1	-0.292496	0.404908	1.110317	0.962526
2	0.268263	0.317409	0.786369	1.391224
3	1.182732	-0.426341	1.180741	2.328007
4	-0.318377	0.361158	0.448336	-0.037767

A classe do tipo de vinho é distribuída em 1, 2 e 3, todavia para que meu método de codificar funcione ele precisa começar em 0, por isso, a classe de vinho foi submetida ao método de *transformar_categorico_em_numerico*, que vai transformá-la na sequência de 0, 1 e 2.

[68]:
```
dict_classes = get_dicionario_classes(classe)
classe = classe.apply(lambda row: transformar_categorico_em_numerico(row,␣
 ↪dict_classes))
classe.value_counts()
```

[68]:
```
1    71
0    59
2    48
Name: Wine, dtype: int64
```

Agora a classe vai ser codificada em uma array assim como aconteceu com a iris, pois temos novamente um problema do tipo multi-classe.

[69]:
```
classe_codificada = codificar_classe()
classe = classe.apply(lambda row: substituir_classe_codificada(row,␣
↪classe_codificada))
classe.head()
```

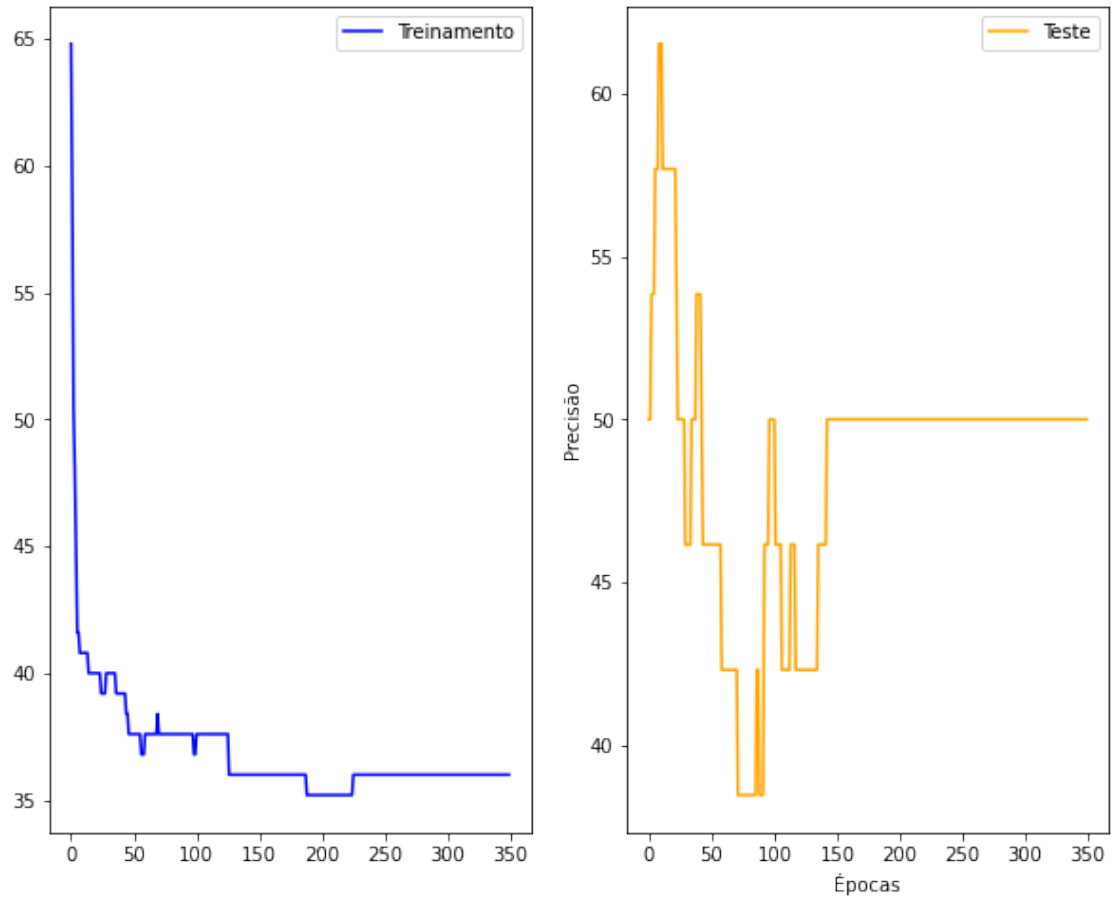
[69]:
```
0    [1, 0, 0]
1    [1, 0, 0]
2    [1, 0, 0]
3    [1, 0, 0]
4    [1, 0, 0]
Name: Wine, dtype: object
```

Agora basta adicionar o neurônio na camada de entrada para ser o bias e executar o perceptron novamente para conferir os resultados da rede.

[70]:
```
previsores['bias'] = 1
```

[72]:
```
executar_perceptron(funcao_ativacao_sigmoid, funcao_custo_mse, 350, [-0.005, 0.
↪005], 0.1)
```

```
/home/alvaro/.local/lib/python3.6/site-packages/ipykernel_launcher.py:2:
RuntimeWarning: overflow encountered in power
```




```
Melhor precisão de treinamento 64.7999999999999
Melhor precisão de teste 61.53846153846154
Melhor precisão de validação 74.07407407407408
Média precisão de treinamento 41.78666666666664
Média precisão de teste 42.3076923076923
Média precisão de validação 40.12345679012344
Desvio Padrão precisão de treinamento 9.738300102630244
Desvio Padrão precisão de teste 9.209376921060498
Desvio Padrão precisão de validação 12.725634646968087
Matriz de confusão de treinamento:
 [[40  1  3]
 [ 6 37  5]
 [12 17  4]]
Matriz de confusão de teste:
 [[10  3  0]
 [ 7  0  0]
 [ 0  0  6]]
Matriz de confusão de validação:
 [[ 6  0  0]
```

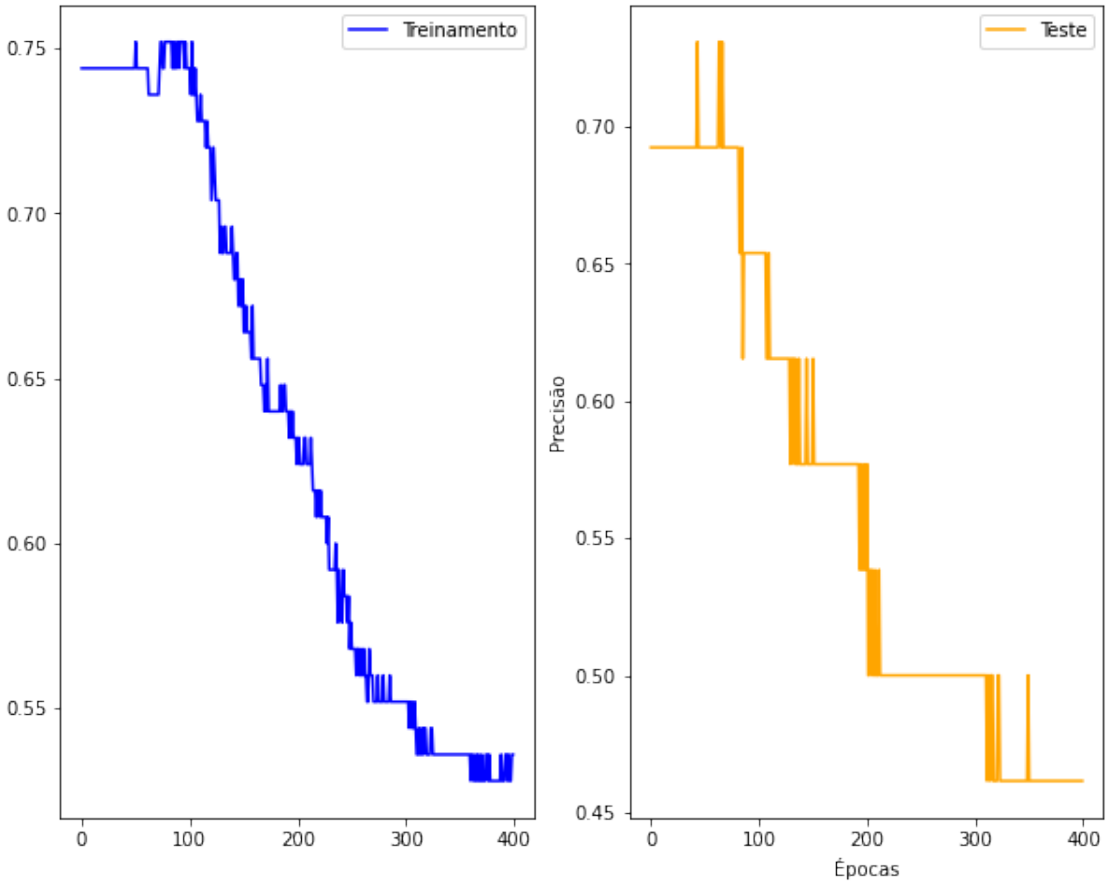
```
 [ 3 14  1]
 [ 3  0  0]]
Tempo de execução: 123.64418005943298 Segundos
```

Os resultados iniciais mostram que a precisão ainda está longe do ideal, a rede acaba gerando uma maior confusão entre os tipos de vinho 2 e 3.

Agora o processo de leitura dos dados e pré-processamento deve ser repetido, porém de maneira um pouco diferente, para adequar os dados à rede perceptron por épocas, assim como foi feito anteriormente no problema das iris, dando a possibilidade de executar a busca por melhores parâmetros e dessa forma melhorar os resultados obtidos.

[63]:
```
previsores = dataframe.iloc[:, 1:14]
classe = dataframe['Wine']
previsores = previsores.apply(lambda row: normalizacao_z_score(row))

dict_classes = get_dicionario_classes(classe)
classe = classe.apply(lambda row: transformar_categorico_em_numerico(row,␣
 ↪dict_classes))
classe_codificada = codificar_classe_epoca()

classe_nova = []

for i in classe:
    classe_nova.append(classe_codificada[i])

classe_nova = np.array(classe_nova).reshape(len(classe), 3)

previsores['bias'] = 1
```

[64]:
```
executar_perceptron_epoca(funcao_ativacao_sigmoid_epoca,␣
 ↪funcao_custo_mse_epoca, 400, [-0.005, 0.005])
```




```
Melhor precisão de treinamento 0.752
Melhor precisão de teste 0.7307692307692307
Melhor precisão de validação 0.6666666666666666
Média precisão de treinamento 0.4277333333333333
Média precisão de teste 0.4589743589743589
Média precisão de validação 0.42839506172839503
Desvio Padrão precisão de treinamento 0.11294332895552335
Desvio Padrão precisão de teste 0.11188422681677654
Desvio Padrão precisão de validação 0.12854336164815103
Tempo de execução: 26.6903395652771 Segundos
Matriz de confusão de treinamento:
 [[39  0  1]
 [18 26  6]
 [ 4  2 29]]
Matriz de confusão de teste:
 [[6 0 1]
 [5 7 1]
 [0 0 6]]
Matriz de confusão de validação:
```

```
[[5 1 0]
[7 6 1]
[0 0 7]]
```

[64]:
```
(0.752, 0.7307692307692307, 0.6666666666666666)
```

Os testes inicias do perceptron por épocas foram um pouco mais promissores em relação à versão anterior, todavia ainda será realizado a busca exaustiva de parâmetros.

[65]:
```
lista_parametros_epoca = { 'custo' : [funcao_custo_mse_epoca],
                        'tx_aprendizado': [0.1, 0.01, 0.0001],
                        'pesos': [0.5, 0.05, 0.005, 0.0005]
}

teste_parametrico = buscar_parametros(lista_parametros_epoca,␣
 ↪executar_perceptron_epoca)
print('Melhores parâmetros', teste_parametrico[0])
print('Melhor precisão teste', teste_parametrico[1])
print('Melhor precisão treinamento', teste_parametrico[2])
print('Melhor precisão validação', teste_parametrico[3])
```

```
Melhores parâmetros (<function funcao_custo_mse_epoca at 0x7f50bb69af28>, 0.01,
0.005)
Melhor precisão teste 0.5769230769230769
Melhor precisão treinamento 0.704
Melhor precisão validação 0.8888888888888888
```

A busca de parâmetros nos resultou em uma taxa de aprendizado de 0.01 acompanhado de um domínio de pesos de 0.005, atingindo um resultado próximo aos 89% de precisão na base de validação.

A rede perceptron, emborar bastante simples, conseguiu um desempenho considerado satisfatório nesse problema com características um pouco mais complexas.

4 Qual o problema?

O desafio escolhido no terceiro exercício foi construir uma rede neural multicamadas sem auxilio de bibliotecas prontas.

O conjundo de dados usado nesse desafio foi o de câncer de mama de Winsconsin, podendo ser encontrado aqui: https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

4.1 Análise exploratória dos dados

Explorando o *dataset* e deixando em um formato correto para ser trabalhado.

[73]:
```
dataframe = pd.read_csv('/home/alvaro/Documentos/mestrado/computação bio/redes␣
↪neurais/datasets/breast cancer.csv', header = 0)
dataframe.head()
```

[73]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	\
0	842302	M	17.99	10.38	122.80	1001.0	
1	842517	M	20.57	17.77	132.90	1326.0	
2	84300903	M	19.69	21.25	130.00	1203.0	
3	84348301	M	11.42	20.38	77.58	386.1	
4	84358402	M	20.29	14.34	135.10	1297.0	

	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	\
0	0.11840	0.27760	0.3001	0.14710	
1	0.08474	0.07864	0.0869	0.07017	
2	0.10960	0.15990	0.1974	0.12790	
3	0.14250	0.28390	0.2414	0.10520	
4	0.10030	0.13280	0.1980	0.10430	

	…	texture_worst	perimeter_worst	area_worst	smoothness_worst	\
0	…	17.33	184.60	2019.0	0.1622	
1	…	23.41	158.80	1956.0	0.1238	
2	…	25.53	152.50	1709.0	0.1444	
3	…	26.50	98.87	567.7	0.2098	
4	…	16.67	152.20	1575.0	0.1374	

	compactness_worst	concavity_worst	concave points_worst	symmetry_worst	\
0	0.6656	0.7119	0.2654	0.4601	
1	0.1866	0.2416	0.1860	0.2750	
2	0.4245	0.4504	0.2430	0.3613	
3	0.8663	0.6869	0.2575	0.6638	
4	0.2050	0.4000	0.1625	0.2364	

	fractal_dimension_worst	Unnamed: 32
0	0.11890	NaN
1	0.08902	NaN
2	0.08758	NaN
3	0.17300	NaN
4	0.07678	NaN

[5 rows x 33 columns]

O *dataframe* é composto basicamente de variáveis contínuas, com excessão apenas da classe, que possui um valor binário de M e B, indicando se o tumor extraído é malígno ou benígno.

Os atributos previsores são as características do tumor, como o formato, textura, área, entre outros.

Antes de mais nada é necessário dividir o conjunto em treinamento e teste, além de apagar algumas colunas que vieram junto no arquivo mas não serão utilizadas.

[74]:
```
dataframe = dataframe.drop(columns = ['id', 'Unnamed: 32']) # ruído nos dados

previsores = dataframe.iloc[:, 1:32] # previsores
classe = dataframe['diagnosis'] # nome da coluna que representa a classe
```

[75]:
```
print(dataframe.isna().sum())
```

```
diagnosis                  0
radius_mean                0
texture_mean               0
perimeter_mean             0
area_mean                  0
smoothness_mean            0
compactness_mean           0
concavity_mean             0
concave points_mean        0
symmetry_mean              0
fractal_dimension_mean     0
radius_se                  0
texture_se                 0
perimeter_se               0
area_se                    0
smoothness_se              0
compactness_se             0
concavity_se               0
concave points_se          0
symmetry_se                0
fractal_dimension_se       0
radius_worst               0
texture_worst              0
perimeter_worst            0
area_worst                 0
smoothness_worst           0
compactness_worst          0
concavity_worst            0
concave points_worst       0
symmetry_worst             0
fractal_dimension_worst    0
dtype: int64
```

[76]:
```
previsores.describe()
```

[76]:

	radius_mean	texture_mean	perimeter_mean	area_mean	\
count	569.000000	569.000000	569.000000	569.000000	
mean	14.127292	19.289649	91.969033	654.889104	
std	3.524049	4.301036	24.298981	351.914129	
min	6.981000	9.710000	43.790000	143.500000	

```
25%      11.700000     16.170000       75.170000   420.300000
50%      13.370000     18.840000       86.240000   551.100000
75%      15.780000     21.800000      104.100000   782.700000
max      28.110000     39.280000      188.500000  2501.000000

       smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
count       569.000000        569.000000      569.000000           569.000000
mean          0.096360          0.104341        0.088799             0.048919
std           0.014064          0.052813        0.079720             0.038803
min           0.052630          0.019380        0.000000             0.000000
25%           0.086370          0.064920        0.029560             0.020310
50%           0.095870          0.092630        0.061540             0.033500
75%           0.105300          0.130400        0.130700             0.074000
max           0.163400          0.345400        0.426800             0.201200

       symmetry_mean  fractal_dimension_mean           …            \
count     569.000000              569.000000           …
mean        0.181162                0.062798           …
std         0.027414                0.007060           …
min         0.106000                0.049960           …
25%         0.161900                0.057700           …
50%         0.179200                0.061540           …
75%         0.195700                0.066120           …
max         0.304000                0.097440           …

       radius_worst  texture_worst  perimeter_worst   area_worst  \
count    569.000000     569.000000       569.000000   569.000000
mean      16.269190      25.677223       107.261213   880.583128
std        4.833242       6.146258        33.602542   569.356993
min        7.930000      12.020000        50.410000   185.200000
25%       13.010000      21.080000        84.110000   515.300000
50%       14.970000      25.410000        97.660000   686.500000
75%       18.790000      29.720000       125.400000  1084.000000
max       36.040000      49.540000       251.200000  4254.000000

       smoothness_worst  compactness_worst  concavity_worst  \
count        569.000000         569.000000       569.000000
mean           0.132369           0.254265         0.272188
std            0.022832           0.157336         0.208624
min            0.071170           0.027290         0.000000
25%            0.116600           0.147200         0.114500
50%            0.131300           0.211900         0.226700
75%            0.146000           0.339100         0.382900
max            0.222600           1.058000         1.252000

       concave points_worst  symmetry_worst  fractal_dimension_worst
count            569.000000      569.000000               569.000000
```

```
mean                0.114606        0.290076                0.083946
std                 0.065732        0.061867                0.018061
min                 0.000000        0.156500                0.055040
25%                 0.064930        0.250400                0.071460
50%                 0.099930        0.282200                0.080040
75%                 0.161400        0.317900                0.092080
max                 0.291000        0.663800                0.207500

[8 rows x 30 columns]
```

Aparentemente essa base de dados também está em um formato já pronto para ser submetido à um algoritmo de aprendizado de máquina, não possuindo valores nulos ou outliers, porém é necessário antes normalizar os valores utilizando o z-score, da mesma forma que os conjuntos anteriores.

[77]:
```
previsores = previsores.apply(lambda row: normalizacao_z_score(row))
previsores.head()
```

[77]:
```
   radius_mean  texture_mean  perimeter_mean  area_mean  smoothness_mean  \
0     1.096100     -2.071512        1.268817   0.983510         1.567087
1     1.828212     -0.353322        1.684473   1.907030        -0.826235
2     1.578499      0.455786        1.565126   1.557513         0.941382
3    -0.768233      0.253509       -0.592166  -0.763792         3.280667
4     1.748758     -1.150804        1.775011   1.824624         0.280125

   compactness_mean  concavity_mean  concave points_mean  symmetry_mean  \
0          3.280628        2.650542             2.530249       2.215566
1         -0.486643       -0.023825             0.547662       0.001391
2          1.052000        1.362280             2.035440       0.938859
3          3.399917        1.914213             1.450431       2.864862
4          0.538866        1.369806             1.427237      -0.009552

   fractal_dimension_mean           …             radius_worst  \
0                2.253764           …                 1.885031
1               -0.867889           …                 1.804340
2               -0.397658           …                 1.510541
3                4.906602           …                -0.281217
4               -0.561956           …                 1.297434

   texture_worst  perimeter_worst  area_worst  smoothness_worst  \
0      -1.358098         2.301575    1.999478          1.306537
1      -0.368879         1.533776    1.888827         -0.375282
2      -0.023953         1.346291    1.455004          0.526944
3       0.133866        -0.249720   -0.549538          3.391291
4      -1.465481         1.337363    1.219651          0.220362

   compactness_worst  concavity_worst  concave points_worst  symmetry_worst  \
0           2.614365         2.107672              2.294058        2.748204
```

```
1          -0.430066        -0.146620              1.086129       -0.243675
2           1.081980         0.854222              1.953282        1.151242
3           3.889975         1.987839              2.173873        6.040726
4          -0.313119         0.612640              0.728618       -0.867590

   fractal_dimension_worst
0                 1.935312
1                 0.280943
2                 0.201214
3                 4.930672
4                -0.396751

[5 rows x 30 columns]
```

[78]:
```
classe.value_counts()
```

[78]:
```
B    357
M    212
Name: diagnosis, dtype: int64
```

Não será necessário transformar a classe em uma array ou uma matriz, pois se trata de um problema binário. Também é possível abservar que as classes estão relativamente bem balanceadas, uma vez que problemas envolvendo patologias geralmente são tratados como uma detecção de anomalia, pelo fato do paciente com a doença representar uma pequena fração de predominância no conjunto de dados, porém não foi o que aconteceu nesse caso.

Ainda assim, será necessário converter a classe de valores categóricos em discretos.

[79]:
```
dict_classes = get_dicionario_classes(classe)
classe = classe.apply(lambda row: transformar_categorico_em_numerico(row,␣
 ↪dict_classes))
dict_classes
```

[79]:
```
{'M': 0, 'B': 1}
```

Com isso, o câncer maligno recebe o valor 0 e o benigno recebe o valor 1.

4.2 Implementação do multilayer perceptron

A ideia da rede neural perceptron multicamadas é parecido com a anterior, porém alguns conceitos matemáticos são adicionados, portanto os métodos tiveram que ser reconstruídos, com exceção apenas da função de soma que se mantém igual.

O método de *inicializar_pesos* por exemplo, acabou recebendo uma pequena modificação para conseguir distribuir dinamicamente os pesos baseado na quantidade de neurônios em cada camada e também na próxima camada adjacente.

[80]:
```
def inicializar_pesos_mlp(neuronios_camada, dominio = [-1, 1]):
    pesos_final = []

    for i in range(len(neuronios_camada) - 1):
        pesos = []
        for j in range(neuronios_camada[i]):
            pesos.append([random.uniform(dominio[0], dominio[1]) for i in␣
↪range(neuronios_camada[i + 1])])
        pesos_final.append(pesos)
    return pesos_final
```

A função de custo e a sigmoid foram simplificadas para representar diretamente suas fórmulas matemáticas

[87]:
```
def funcao_ativacao_sigmoid_mlp(valor):
    resultado = 1 / (1 + np.exp(-valor))
    return resultado
```

[88]:
```
def funcao_custo_mlp(valor_correto, valor_previsto):
    valor_erro = valor_correto - valor_previsto
    return valor_erro
```

4.2.1 Feedfoward

O feedfoward é o nome dado ao processo de cálculo do resultado da função de ativação. O processo é simples igual no perceptron, todavia ele deve ser repetido para todas as camadas até chegar na camada de saída, diferentemente da versão anterior onde esse processo de somatória e aplicação da função de ativação era realizado apenas uma vez.

[89]:
```
def feed_foward(pesos, x_treinamento, f_ativacao):
    ativacao = []
    for i in range(len(pesos)): # percorre todas as camadas
        if i == 0: # na camada de entrada os valores são os próprios atributos␣
↪previsores
            soma_sinapse = np.dot(x_treinamento, pesos[i])
            ativacao.append(f_ativacao(soma_sinapse))
        else: # nas próximas camadas, o resultado anteriores dos neurônios são␣
↪considerados.
            soma_sinapse = np.dot(ativacao[i - 1], pesos[i])
            ativacao.append(f_ativacao(soma_sinapse))

    return ativacao
```

4.2.2 Calculando a derivada e o delta da camada de saída

A derivada e o delta da camada de saída são valores importantes para o ajuste dos pesos, usados no processo de retropropagação do erro. Portanto, foram criados funções que vão calcular os respectivos valores.

[90]:
```
def calcular_derivada_parcial(valor):
    return valor * (1 - valor)
```

[91]:
```
def calcular_delta(erro, derivada):
    return erro * derivada
```

O processo de cálculo de derivada e delta para as camadas ocultas é um pouco diferente, portanto foi criado uma função para servir a esse propósito específico.

[92]:
```
def calcular_delta_oculto(pesos, delta_saida, derivada):
    matriz_pesos = np.transpose(np.asmatrix(pesos)) # conceito de matriz␣
↪transposta

    pesos_delta_saida = delta_saida.dot(matriz_pesos)

    return derivada * np.array(pesos_delta_saida) # as matrizes precisam estar␣
↪em uma dimensão diferente uma da outra
```

[93]:
```
def get_delta_oculto(pesos, delta_saida, ativacao):
    deltas_camadas_ocultas = []  # pegar a derivada da saída

    for i in range(len(pesos) -1):
        if i == 0:
            derivada = calcular_derivada_parcial(ativacao[len(ativacao) - (i +␣
↪2)]) # pegar de trás para frente a derivada de cada neurônio
            deltas_camadas_ocultas.
↪append(calcular_delta_oculto(pesos[len(pesos) - (i + 1)], delta_saida,␣
↪derivada))
        else:
            derivada = calcular_derivada_parcial(ativacao[len(ativacao) - (i +␣
↪2)]) # pegar de trás para frente a derivada de cada neurônio
            deltas_camadas_ocultas.
↪append(calcular_delta_oculto(pesos[len(pesos) - (i + 1)],␣
↪deltas_camadas_ocultas[i - 1], derivada))

    return deltas_camadas_ocultas
```

4.2.3 Backpropagation

Agora com todos os neurônios com os respectivos valores de delta e as derivadas, basta aplicar o algoritmo de retropropagação, começando da camada de saída até a camada de entrada.

O método recebe todos os valores que foram calculados até então, além da taxa de aprendizado e o momentum.

[94]:
```
def backpropagation(pesos, ativacao, delta_saida, delta_oculto, x_treinamento,␣
↪tx_aprendizado = 0.3, momento = 1):
    for i in range(len(pesos)):
        if i == len(pesos) - 1:
            valor_neuronio_transposto = np.transpose(x_treinamento.values)
            delta_x_entrada = valor_neuronio_transposto.dot(delta_oculto[0])
            pesos[len(pesos) - (1 + i)] = (pesos[len(pesos) - (1 + i)] *␣
↪momento) + (tx_aprendizado * delta_x_entrada)
        elif i == 0:
            valor_neuronio_transposto = np.transpose(ativacao[len(ativacao) -␣
↪(i + 1)])
            delta_x_entrada = valor_neuronio_transposto.dot(delta_saida)
            pesos[len(pesos) - (1 + i)] = (pesos[len(pesos) - (1 + i)] *␣
↪momento) + (tx_aprendizado * delta_x_entrada)
        else:
            valor_neuronio_transposto = np.transpose(ativacao[len(ativacao) -␣
↪(i + 1)])
            delta_x_entrada = valor_neuronio_transposto.
↪dot(delta_oculto[len(delta_oculto) - i])
            pesos[len(pesos) - (1 + i)] = (pesos[len(pesos) - (1 + i)] *␣
↪momento) + (tx_aprendizado * delta_x_entrada)

    return pesos
```

Uma dificuldade encontrada foi ajustar as dimensões das matrizes para que as fórmulas sejam corretamente aplicadas, portanto por enquanto o perceptron só aceita camadas ocultas com a mesma quantidade de neurônios.

4.2.4 Precisão

O método *get_precisao* foi utilizado para calcular a diferença absoluta nas previsões da última camada do modelo com o valor real da classe, gerando assim a precisão do algoritmo.

[95]:
```
def get_precisao(valor_correto, valor_previsto):
    previsao = valor_previsto.copy()

    previsao[previsao >= 0.5] = 1
    previsao[previsao < 0.5] = 0

    precisao = (valor_correto == previsao).sum() / len(valor_correto)
    return precisao
```

4.2.5 Matriz de confusão

A matriz de confusão é especialmente útil nesses casos, uma vez que estamos lidando com vidas de pacientes. Sabemos que em problemas envolvendo patologias, é preferível um número maior de falsos positivos do que falsos negativos, pois é melhor diagnosticar o paciente com a doença e depois descartar através de mais exames do que descartar a possibilidade de doença e deixar que a condição de saúde seja agravada.

O método de gerar a matriz de confusão foi adaptado aqui também, para trabalhar diretamente com matrizes.

[96]:
```
def get_matriz_confusao(valor_correto, valor_previsto):
    previsao = valor_previsto.copy()

    previsao[previsao >= 0.5] = 1
    previsao[previsao < 0.5] = 0

    matriz_confusao = confusion_matrix(valor_correto, previsao)

    return matriz_confusao
```

4.2.6 Treinamento do algoritmo

O método *treinar_mlp* é o responsável por aplicar todos os métodos que foram criados anteriormente, atualizado os pesos em cada uma das épocas.

[111]:
```
def treinar_mlp(epocas, neuronios_camada, f_ativacao, f_custo, pesos,␣
↪x_treinamento,
                                    y_treinamento, x_teste, y_teste,␣
↪tx_aprendizado):
    execucoes = 0
    precisoes_treinamento = []
    precisoes_teste = []
    melhores_pesos = []
    melhor_matriz_treinamento = []
    melhor_matriz_teste = []

    while execucoes < epocas:
        ativacao = feed_foward(pesos, x_treinamento, f_ativacao)

        resultado_camada_saida = ativacao[len(ativacao) - 1]
        classe_reshaped = y_treinamento.values.reshape(-1,1)

        erro = f_custo(classe_reshaped, resultado_camada_saida)

        precisoes_treinamento.append(get_precisao(classe_reshaped,␣
↪resultado_camada_saida))
```

```
        derivada_saida = calcular_derivada_parcial(resultado_camada_saida)
        delta_saida = calcular_delta(erro, derivada_saida)

        delta_camada_oculta = get_delta_oculto(pesos, delta_saida, ativacao)

        melhores_pesos = pesos.copy() if precisoes_treinamento[execucoes] >=␣
↪max(precisoes_treinamento) else melhores_pesos
        melhor_matriz_treinamento = get_matriz_confusao(classe_reshaped,␣
↪resultado_camada_saida) if precisoes_treinamento[execucoes] >=␣
↪max(precisoes_treinamento) else melhor_matriz_treinamento

        pesos = backpropagation(pesos, ativacao, delta_saida,␣
↪delta_camada_oculta, x_treinamento)

        teste_rede = testar_mlp(pesos, x_teste, y_teste, f_ativacao, f_custo)
        precisoes_teste.append(teste_rede[0])
        melhor_matriz_teste = teste_rede[1] if precisoes_teste[execucoes] >=␣
↪max(precisoes_teste) else melhor_matriz_teste
        execucoes += 1

    return precisoes_treinamento, precisoes_teste, melhores_pesos,␣
↪melhor_matriz_treinamento, melhor_matriz_teste
```

Para testar o desempenho ao final de cada época o método testar também passou por adaptações

[106]:
```
def testar_mlp(pesos, x_previsores, y_classe, f_ativacao, f_custo):
    precisao = 0

    ativacao = feed_foward(pesos, x_previsores, f_ativacao)

    resultado_camada_saida = ativacao[len(ativacao) - 1]
    classe_reshaped = y_classe.values.reshape(-1,1)

    matriz_confusao = get_matriz_confusao(classe_reshaped,␣
↪resultado_camada_saida)
    precisao = get_precisao(classe_reshaped, resultado_camada_saida)

    return precisao, matriz_confusao
```

4.2.7 Adicionando camadas

A lógica foi construída para inicializar a estrutura de pesos baseado em uma Array que vai indicar o número de camadas e de neurônios em cada camada, portanto o processo de expandir a rede se torna simples.

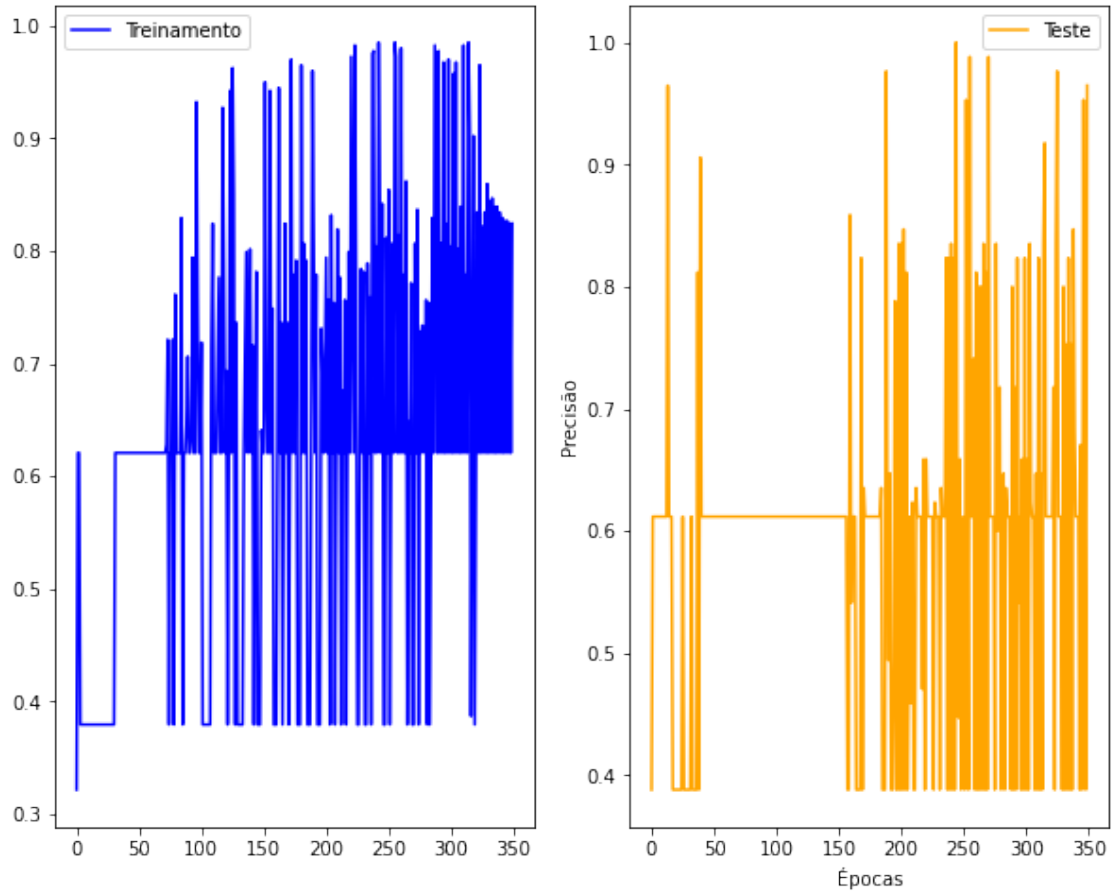
[99]:
```
neuronios_camada = [len(previsores.columns)] # adicionado neurônios da camada␣
↪de entrada
neuronios_camada.append(10) # camada oculta
neuronios_camada.append(1) # camada de saída.
```

Foi criado, por fim, o método *executar_mlp* com o mesmo objetivo dos métodos anteriores de executar, fazendo os tester na rede com 30 inicializações.

[113]:
```
def executar_mlp(funcao_ativacao, funcao_custo, epocas, dominio_pesos = [-1, 1],
                        tx_aprendizado = 0.001):

    convergencia_treinamento = [0]
    convergencia_teste = [0]
    precisao_treinamento = []
    precisao_teste = []
    resultado_final = []
    matriz_confusao_treinamento = []
    matriz_confusao_teste = []
    matriz_confusao_validacao = []
    start_time = time.time() # tempo de execução

    for i in range(30):
        x_treinamento, y_treinamento, x_teste, y_teste, \
        x_validacao, y_validacao = dividir_dataframe(previsores, classe, 0.7, 0.
↪15, 0.15)

        pesos = inicializar_pesos_mlp(neuronios_camada, dominio_pesos)

        treinamento = treinar_mlp(epocas, neuronios_camada, funcao_ativacao,␣
↪funcao_custo, pesos, x_treinamento, y_treinamento, x_teste, y_teste,␣
↪tx_aprendizado)

        convergencia_treinamento = treinamento[0] if max(treinamento[0]) >= \
                                    max(convergencia_treinamento) else␣
↪convergencia_treinamento

        convergencia_teste = treinamento[1] if max(treinamento[1]) >=␣
↪max(convergencia_teste) \
                                        else convergencia_teste

        precisao_treinamento.append(max(treinamento[0]))
        precisao_teste.append(max(treinamento[1]))
        teste_final = testar_mlp(treinamento[2], x_validacao, y_validacao,
                                    funcao_ativacao, funcao_custo)
        resultado_final.append(teste_final[0])
```

```
        matriz_confusao_treinamento = treinamento[3] if max(treinamento[0]) >=␣
↪max(precisao_treinamento) else matriz_confusao_treinamento
        matriz_confusao_teste = treinamento[4] if max(treinamento[1]) >=␣
↪max(precisao_teste) else matriz_confusao_teste
        matriz_confusao_validacao = teste_final[1] if teste_final[0] >=␣
↪max(resultado_final) else matriz_confusao_validacao

    plotar_convergencia(convergencia_treinamento, convergencia_teste)
    exibir_resultados(precisao_treinamento, precisao_teste, resultado_final)
    print('Matriz de confusão de treinamento:\n', matriz_confusao_treinamento)
    print('Matriz de confusão de teste:\n', matriz_confusao_teste)
    print('Matriz de confusão de validação:\n', matriz_confusao_validacao)
    print("Tempo de execução: %s Segundos" % (time.time() - start_time))
```

[114]:
```
executar_mlp(funcao_ativacao_sigmoid_mlp, funcao_custo_mlp, 350)
```




```
Melhor precisão de treinamento 0.9849246231155779
Melhor precisão de teste 1.0
Melhor precisão de validação 0.9883720930232558
```

```
Média precisão de treinamento 0.8716080402010049
Média precisão de teste 0.8631372549019609
Média precisão de validação 0.8709302325581395
Desvio Padrão precisão de treinamento 0.1357006464781066
Desvio Padrão precisão de teste 0.1319355801470074
Desvio Padrão precisão de validação 0.11889878409017271
Matriz de confusão de treinamento:
 [[146   5]
 [  1 246]]
Matriz de confusão de teste:
 [[33  0]
 [ 0 52]]
Matriz de confusão de validação:
 [[29  1]
 [ 0 56]]
Tempo de execução: 14.686784744262695 Segundos
```

4.3 Resultados da rede perceptron multicamadas

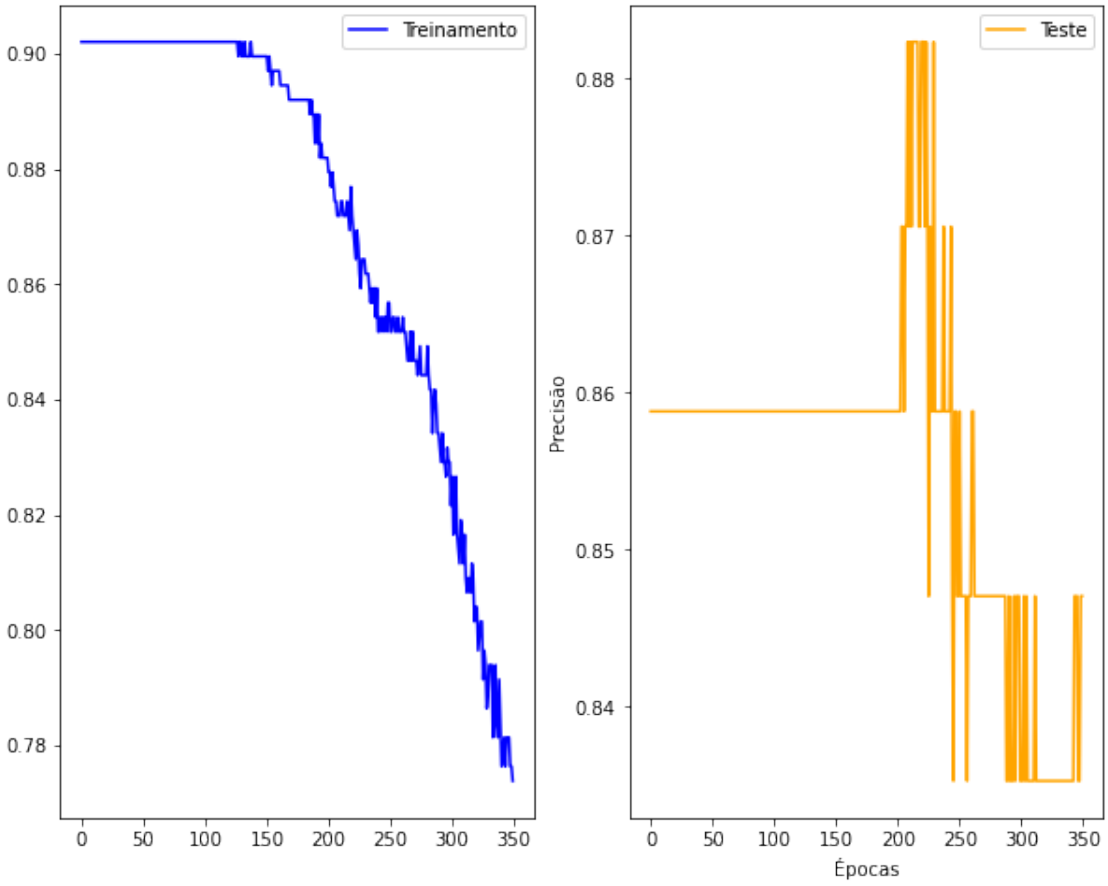
O algoritmo, ainda que sem muitos ajustes nos parâmetros, conseguiu uma precisão de quase 100% em todas os conjuntos de dados, também sendo possível ver a consistência do mesmo, com uma média próxima aos 90% em todas as 30 execuções.

O que também impressiona é o tempo de processamento, sendo mais veloz até que o perceptron por épocas, claro que isso ocorre por se tratar de um problema simples, onde não foi necessário construir uma arquitetura mais larga ou profunda da rede neural, inclusive em alguns testes realizados o resultado foi melhor ao adicionar apenas uma camada oculta, com 10 ou 15 neurônios.

Um resultado não tão positivo observado é que os erros da perceptron multicamadas, ainda que poucos, são justamente falsos negativos (primeira linha, segunda coluna), ou seja, um câncer que foi dito como sendo benigno mas na verdade era maligno. Um ajuste que pode ser interessante fazer nessa situação seria alterar a métrica de sucesso do algoritmo para buscar uma maior acurácia ao invés de precisão, dando maior ênfase na diminuição de falsos positivos.

[115]:
```
dict_classes = get_dicionario_classes(classe)
classe_codificada = codificar_classe()

classe_nova = []

for i in classe:
    classe_nova.append(classe_codificada[i])

classe_nova = np.array(classe_nova).reshape(len(classe), 2)
```

[117]:
```
executar_perceptron_epoca(funcao_ativacao_sigmoid_epoca,␣
 ↪funcao_custo_mse_epoca, 350, [-0.005, 0.005])
```




```
Melhor precisão de treinamento 0.9020100502512562
Melhor precisão de teste 0.8823529411764706
Melhor precisão de validação 0.8837209302325582
Média precisão de treinamento 0.5242043551088776
Média precisão de teste 0.5466666666666667
Média precisão de validação 0.4992248062015504
Desvio Padrão precisão de treinamento 0.20922320744955683
Desvio Padrão precisão de teste 0.20324321560756622
Desvio Padrão precisão de validação 0.2141752586172404
Tempo de execução: 45.186291456222534 Segundos
Matriz de confusão de treinamento:
 [[128   6]
 [ 33 231]]
Matriz de confusão de teste:
 [[24  5]
 [ 5 51]]
Matriz de confusão de validação:
 [[33  2]
 [ 8 43]]
```

[117]: (0.9020100502512562, 0.8823529411764706, 0.8837209302325582)

A rede perceptron não se saiu mal, porém foi nitidamente inferior, tanto nos resultados de precisão como em um maior tempo de processamento, além da instabilidade das média e desvio padrão.

[118]:
```
lista_parametros_epoca = { 'custo' : [funcao_custo_mse_epoca],
                      'tx_aprendizado': [0.1, 0.01, 0.0001],
                      'pesos': [0.5, 0.05, 0.005, 0.0005]
}

teste_parametrico = buscar_parametros(lista_parametros_epoca,␣
 ↪executar_perceptron_epoca)
print('Melhores parâmetros', teste_parametrico[0])
print('Melhor precisão teste', teste_parametrico[1])
print('Melhor precisão treinamento', teste_parametrico[2])
print('Melhor precisão validação', teste_parametrico[3])
```

```
Melhores parâmetros (<function funcao_custo_mse_epoca at 0x7f50bb69af28>,
0.0001, 0.0005)
Melhor precisão teste 0.8941176470588236
Melhor precisão treinamento 0.8844221105527639
Melhor precisão validação 0.9534883720930233
```

Os resultados da busca de parâmetros encontraram uma taxa de aprendizado de 0.0001 e um domínio de pesos de 0.0005 como melhor resultado no conjuntos de validação, todavia ainda é possível observar a superioridade da rede perceptron multicamadas.

4.4 Próximos passos

O algoritmo de perceptron multicamadas construído, apesar de eficiente, foi bem simples e algumas medidas poderiam ser feitas a fim de deixar ainda mais robusto: * Descida do gradiente estocástica * Dropout * Parada no platô e redução dinâmica de taxa de aprendizado * Otimização do *momentum* * Utilizar outras métricas de sucesso (*recall, acurácia, f1-score*)

5 Conclusões

Os algoritmos de redes neurais são extremamente competentes para resolver problemas mais complexos. O Perceptron consegue ser eficiênte quando lidamos com características linearmente separáveis e as redes perceptron multicamadas são as ideias em problemas mais complexos de serem separados espacialmete.

Podemos notar que os problemas aqui propostos puderam ser eficientemente resolvidos com o uso desses algoritmos, ainda que em uma forma mais simplificada devido à limitações da própria implementação.

Um problema enfrentado está no aumento do tempo de processamento e poder computacional requerido para a utilização desses algoritmos, além do fato do aumento na quantidade de parâmetros

que devem ser cuidadosamente ajustados, necessitando de um conhecimento mais especializado por trás do responsável por desenvolver a solução. Também não podemos esquecer da aleatoriedade desse tipo de algoritmo, alguns trabalhos de *explainable IA* estão sendo desenvolvidos para que redes densas e complexas sejam mais facilmente interpretadas e deixem de ser um *black box*.

As desvantagens estão cada dia mais sendo ultrapassadas pelas soluções modernas e os algoritmos envolvendo redes neurais se consolidando para resolução de problemas complexos.