Universidade Federal do Pará

Redes Neurais Artificiais

Implementação Algoritmo de Backpropagation

Helder Mateus dos Reis Matos

Dra. Adriana Rosa Garcez Castro

21 de Novembro de 2019

Conteúdo

1 Introdução **2**

2 Implementação **2**

2.1 Inicializando pesos 2

2.2 Feedforward (Alimentação Adiante) 3

2.3 Backpropagation (Retropropagação) 4

2.4 Treino 5

2.5 Validação 6

3 Conclusão **7**

1 Introdução

O algoritmo de Backpropagation (retropropagação) é um dos mais utilizados principalmente por iniciantes, para criar redes neurais. Sua simplicidade e capacidade de encontrar soluções para problemas de ordem significante o tornam o principal ponto de partida para o estudo de aprendizadado de máquina.

O backpropagation pode ser dividido em duas etapas principais: feedfoward, onde o sinal de entrada da rede é propagado ao longo da mesma e fornece uma saída, e o de backpropagation, onde a saída é retro-alimentada na rede, de modo que os pesos sinápticos dos neurônios sejam ajustados de acordo com as interações de aprendizado.

Este trabalho tem o objetivo de fazer a implementação de uma Rede Neural Artificial com algoritmo de backpropagation em Python, fornecendo dados relativos ao erro da predição dos valores, assim como gráfico de evolução do erro ao longo do tempo, alterando parâmetros como a quantidade de neurônios na camada escondida, a taxa de aprendizado e a quantidade de passos de aprendizagem.

2 Implementação

A rede foi organizada em uma lista de três camadas, uma de entrada, uma escondida de e uma de saída, e cada camada é estruturada como um dicionário que, inicialmente, guarda os pesos sinápticos dos neurônios. Durante as fases de alimentação adiante e retropropagação, esse léxico recebe valores de saída da rede e de ajuste de pesos (regra delta).

2.1 Inicializando pesos

```
1
2  # inicializando rede com pesos aleatorios
3  def inicializar_rede(n_inp, n_hid, n_out):
4      rede = []
5
6      # cada neuronio da escondida possui n_inp entradas + 1 bias
7      camada_hid = [{'pesos': [random() for i in range(n_inp + 1)
       ]} for i in range(n_hid)]
8      rede.append(camada_hid)
9
10     # cada neuronio da saida possui n_hid entradas + 1 bias
11     camada_out = [{'pesos': [random() for i in range(n_hid + 1)
       ]} for i in range(n_out)]
```

```
12      rede.append(camada_out)
13
14      # a rede eh um array de camadas, e cada camada um dicionario
```

Os pesos são valores aleatórios uniformemente distribuídos, entre 0 e 1. O número de entradas de cada neurônio da camada escondida é equivalente à quantidade de neurônios na camada de entrada mais um bias na última posição, assim como o número de entradas de cada neurônio da camada de saída é equivalente à quantidade de neurônios na camada de entrada mais um bias.

2.2 Feedforward (Alimentação Adiante)

```
1
2  # calcular ativacao do neuronio para uma entrada
3  def ativacao(pesos, entradas):
4      ativacao = pesos[-1] # adiciona o bias
5      for i in range(len(pesos) - 1): # soma ponderada das
       entradas
6          ativacao += pesos[i] * entradas[i]
7      return ativacao
8
9  # funcao de ativacao: sigmoide
10 def transferencia(ativacao):
11     return 1.0 / (1.0 + math.exp(-ativacao))
12
13 # alimentacao adiante, obtendo vetor de saida
14 def feedforward(rede, datarow):
15     entradas = datarow # a entrada da rede contem uma linha do
       dataset
16     for camada in rede: # calcular a saida para cada camada
17         prox_entrada = [] # saida de uma camada -> entrada da
       proxima
18         for neuronio in camada: # calcular a saida para cada
       neuronio
19             vetor_ativacao = ativacao(neuronio['pesos'],
       entradas)
20             neuronio['saida'] = transferencia(vetor_ativacao)
21             prox_entrada.append(neuronio['saida'])
22         entradas = prox_entrada # para proxima camada
```

Na etapa de feedforward, definida na função `def feedforward`, os sinais de entrada são propagados por toda a estrutura do neurônio. Assim que as entradas são passadas para a rede, é calculado campo local induzido, dado por

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) \cdot y_i(n)$$

Essa relação é definida em uma função `def ativacao`. Para cada linha da entrada, o bias da camada é dado pelo último elemento da lista. Para as outras entradas, é efetuado o campo induzido, que é o valor retornado da função.

Após calcular os campos $v_j(n)$ para uma camada, é gerada a saída $y_j(n) = \varphi_j(v_j(n))$ através da função de ativação, φ_j. Foi utilizada a função logística sigmóide, dada por

$$\varphi(v) = \frac{1}{1 + e^{-v}}$$

O valor de saída do neurônio é retornado da função `def transferencia`. As entradas são direcionadas diretamente para a camada encondida, é gerada uma saída para essa camada. Essa mesma saída é direcionada para a camada de saída, que gera a saída final da rede.

2.3 Backpropagation (Retropropagação)

```
1
2  # calcular inclinacao da saida
3  def derivada(saida):
4      return saida * (1.0 - saida)
5
6  # calcula o erro para cada camada e retropropaga
7  def back_prop(rede, esperado):
8      # calculando o erro de tras pra frente
9      for i in reversed(range(len(rede))):
10         camada = rede[i]
11         erros = []
12         if i != len(rede) - 1: # se nao for a ultima camada
13             for j in range(len(camada)):
14                 erro = 0.0
15                 for neuronio in rede[i + 1]: # erro = peso *
    erro da saida
16                     erro += (neuronio['pesos'][j] * neuronio['
    delta'])
17                 erros.append(erro)
18         else: # se for a camada de saida
19             for j in range(len(camada)): # erro = esperado -
    saida
20                 neuronio = camada[j]
21                 erros.append(esperado[j] - neuronio['saida'])
22         for j in range(len(camada)):
23             neuronio = camada[j]
24             neuronio['delta'] = erros[j] * derivada(neuronio['
    saida'])
25
```

```
# atualiza pesos a partir dos erros
def atualizar_pesos(rede, datarow, taxa_apre):
    for i in range(len(rede)):
        entradas = datarow[:-1]
        if i != 0:
            entradas = [neuronio['saida'] for neuronio in rede[i
    -1]]
        for neuronio in rede[i]:
            for j in range(len(entradas)):
                neuronio['pesos'][j] += taxa_apre * neuronio['
    delta'] * entradas[j]
```

De posse da saída, serão calculados os erros em comparação com as saídas desejadas, com o objetivo de retropropagar os mesmos na rede e ajustar os pesos até que o erro seja suficientemente reduzido.

Esse levantamento dos erros é feito de maneira reversa, onde são calculados inicialmente os erros da camada de saída. Na função `def back_prop`, para cada neurônio na camada de saída, o erro é dado por $e^j(n) = d_j(n) - o_j(n)$, onde d_j é a saída desejada e o_j é a saída obtida.

De posse dos erros para a camada de saída, obtemos o gradiente local para esses neurônios, ou seja, a direção para a mudança de pesos que reduza o erro. O gradiente local é expresso por $\delta(n) = e(n) \cdot \varphi`(v)$, onde $e(n)$ são os erros da camada e $\varphi`(v) = \varphi(v) \cdot (1 - \varphi(v))$. Cada um dos gradientes é adicionado a uma lista de deltas (δ), que é anexada ao dicionário da camada. Cada δ será utilizado para atualizar os pesos em breve.

Para encontrar os deltas da camada escondida, basta encontrar o produto de $\varphi`(v)$ pela soma ponderada dos δ da camada de saída. Assim, o gradiente local da camada escondida é dado por

$$\delta(n) = \varphi`(v) \cdot (\sum_k \delta_k^{saida} \cdot w_k^{saida})$$

`def atualiza_pesos` faz o ajuste dos pesos da rede, a partir dos valores obtidos durante a retropropagação. Esse ajuste é realizado pela regra delta, dada por $w = \nu \cdot \delta(n) \cdot y(n)$, onde ν é o parâmetro da taxa de aprendizagem e δ e y são os deltas e valores de entrada de uma camada, respectivamente.

2.4 Treino

```
# treino da rede
def treinar_rede(rede, treino, taxa_apre, n_epoca, n_out):
    vetor_de_erros = []
    vetor_de_epocas = []
```

```
    for epoca in range(n_epoca): # sgd para um numero de epocas
        vetor_de_epocas.append(epoca)
        sum_erro = 0
        for datarow in treino: # feedforward, backprop e
    atualiza os pesos
            saidas = feedforward(rede, datarow)
            esperado = [0 for i in range(n_out)]
            esperado[datarow[-1]] = 1
            sum_erro += sum([(esperado[i] - saidas[i])**2 for i
    in range(len(esperado))])
            back_prop(rede, esperado)
            atualizar_pesos(rede, datarow, taxa_apre)
        vetor_de_erros.append(sum_erro)
        print('EPOCA = %d, TAXA_APRE = %.3f, ERRO = %.3f' % (
    epoca, taxa_apre, sum_erro))
    fig, ax = plt.subplots()
    plt.grid()
    plt.title('Evolucao do MSE ao longo das epocas')
    plt.ylabel('Erro Medio Quadratico')
    plt.xlabel('Epocas')
    ax.plot(vetor_de_epocas, vetor_de_erros, color='red')
```

Uma época se refere ao processo ou ciclo completo de aprendizagem, desde a entrada dos dados no estágio de feedforward até a atualização dos pesos após o backpropagation. Assim a função `def treinar_rede` fará todo o processo quantas vezes forem estipuladas por uma variável `n_epoca`.

Neste estágio também é gerador o Erro Quadrático Médio em relação as saídas desejadas e obtidas da rede. O MSE (Mean Squared Error) é dado por

$$MSE = \sum_{n=1}^{N} (d_i(n) - y_i(n))^2$$

2.5 Validação

```
def validacao(rede, datarow):
    saidas = feedforward(rede, datarow)
    return saidas.index(max(saidas)) # em saidas, retorna o
    valor maximo para a classe

def minmax_data(dataset):
    return [[min(coluna), max(coluna)] for coluna in zip(*
    dataset)]

def normalizar_dataset(dataset, minmax):
    for linha in dataset:
        for i in range(len(linha) - 1):
            linha[i] = (linha[i] - minmax[i][0]) / (minmax[i][1]
    - minmax[i][0])
```

```
    return dataset

def rodar_rede(dataset, n_hid,taxa_apre,epocas):
    minmax = minmax_data(dataset)
    datatreino = normalizar_dataset(dataset, minmax)
    datavalid = normalizar_dataset(dataset, minmax)
    datatreino = dataset[0:10]
    datavalid = dataset[10:20]
    print(datatreino)

    n_inp = len(datatreino[0]) - 1
    n_out = len(set([datarow[-1] for datarow in datatreino]))
    seed(1)
    rede = inicializar_rede(n_inp,n_hid,n_out)
```

A função `rodar_rede` é responsável por executar todas as outras sub-rotinas implementadas até então. Além de treinar a rede ela também toma os dados para validação e verifica se as saídas esperadas são equivalentes ao que o algoritmo tenta predizer. Antes de começar o treino, os dados precisam passar por um processo de normalização, onde o dataset é distribuído em valores reais entre 0 e 1, através das funções `def minmax_data` e `def normalizar_dataset`. Para usar os dados de validação, basta aplicá-los na sub-rotina de feed-forward, já que os pesos já foram definidos durante a fase de treino.

3 Conclusão

A implementação desta rede possibilitou o realização de um planejamento para a criação do projeto, onde diversos fatores contribuiram significativamente para um bom desempenho do algoritmo. Interagir diretamente na arquitetura de uma rede desta configuração consolidou conhecimentos intrínsecos desse modelo e abre perspectivas de melhorias para o algoritmo, tais como a alteração de funções de ativação, acréscimo de camadas escondidas e uma otimização da inicialização dos pesos.

Referências

[1] Symon Haykin. *Redes Neurais: Princípios e prática*. Bookman, 2001.

[2] William W. Hines, Douglas C. Montgomery, David M. Goldsman e Connie M. Borror. *Probabilidade e Estatística na Engenharia*. John Wiley & Sons, 2003.

[3] Jason Browlee. *Machine Learning Algorithms From Scratch.* `https://machinelearningmastery.com/machine-learning-algorithms-from-scratch/`