Universidade Federal do Pará

Redes Neurais Artificiais

Implementação e Aplicação de Algoritmo de Backpropagation

Helder Mateus dos Reis Matos

Dra. Adriana Rosa Garcez Castro

21 de Novembro de 2019

Conteúdo

1 Introdução

2 Estrutura de uma Rede Neural Artificial

3 Backpropagation

4 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).

4.1 Inicializando pesos

```

# inicializando rede com pesos aleatorios
def inicializar_rede(n_inp, n_hid, n_out):
    rede = []

    # cada neuronio da escondida possui n_inp entradas + 1 bias
    camada_hid = [{'pesos': [random() for i in range(n_inp + 1)
    ]} for i in range(n_hid)]
    rede.append(camada_hid)

    # cada neuronio da saida possui n_hid entradas + 1 bias
    camada_out = [{'pesos': [random() for i in range(n_hid + 1)
    ]} for i in range(n_out)]
    rede.append(camada_out)

    # 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.

4.2 Feedforward (Alimentação Adiante)

```

# calcular ativacao do neuronio para uma entrada
def ativacao(pesos, entradas):
    ativacao = pesos[-1] # adiciona o bias
    for i in range(len(pesos) - 1): # soma ponderada das
    entradas
        ativacao += pesos[i] * entradas[i]
    return ativacao

# funcao de ativacao: sigmoide
def transferencia(ativacao):
    return 1.0 / (1.0 + math.exp(-ativacao))

# alimentacao adiante, obtendo vetor de saida
def feedforward(rede, datarow):
```

```
    entradas = datarow # a entrada da rede contem uma linha do dataset
    for camada in rede: # calcular a saida para cada camada
        prox_entrada = [] # saida de uma camada -> entrada da proxima
        for neuronio in camada: # calcular a saida para cada neuronio
            vetor_ativacao = ativacao(neuronio['pesos'], entradas)
            neuronio['saida'] = transferencia(vetor_ativacao)
            prox_entrada.append(neuronio['saida'])
        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.

4.3 Backpropagation (Retropropagação)

```

# calcular inclinacao da saida
def derivada(saida):
    return saida * (1.0 - saida)

# calcula o erro para cada camada e retropropaga
def back_prop(rede, esperado):
```

```
    # calculando o erro de tras pra frente
    for i in reversed(range(len(rede))):
        camada = rede[i]
        erros = []
        if i != len(rede) - 1: # se nao for a ultima camada
            for j in range(len(camada)):
                erro = 0.0
                for neuronio in rede[i + 1]: # erro = peso * erro da saida
                    erro += (neuronio['pesos'][j] * neuronio['delta'])
                erros.append(erro)
        else: # se for a camada de saida
            for j in range(len(camada)): # erro = esperado - saida
                neuronio = camada[j]
                erros.append(esperado[j] - neuronio['saida'])
        for j in range(len(camada)):
            neuronio = camada[j]
            neuronio['delta'] = erros[j] * derivada(neuronio['saida'])

# 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.

4.4 Treino

```python
# 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

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

4.5 Validação

```python
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])

def rodar_rede(datatreino, datavalid, n_hid,taxa_apre,epocas):
    minmax = minmax_data(datatreino)
    normalizar_dataset(datatreino, minmax)
    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)
    treinar_rede(rede, datatreino, taxa_apre, epocas, n_out)
    for camada in rede:
        print(camada)
    for datarow in datavalid:
        valid = validacao(rede, datarow)
        print('ESPERADO = %d, OBTIDO = %d' % (datarow[-1], valid))
```

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.

5 Aplicação

5.1 Escolha e tratamento do Dataset

5.2 Aplicação na Rede Neural

5.3 Análise dos Resultados

5.3.1 Erro Médio Quadrático

5.3.2 Saída Desejada × Saída Obtida

5.3.3 Comparações para Diferentes Topologias

6 Conclusão

7 Referências

8 Anexos