Relatório 12 - Prática: Redes Neurais (II)

Thassiana C. A. Muller

Introdução

O card aborda os principais conceitos de redes neurais artificiais, a teoria aplicada de classificação binária em um exemplo de existência ou não existência de câncer de mama, classificação multiclasses com o dataset íris, regressão com predição de preço de carros e regressão de múltiplas saídas com um dataset de videogames

O panorama geral de técnicas e aplicações do aprendizado de máquina pode ser descrito na figura a seguir:

Aprendizagem supervisionada	Aprendizagem não supervisionada
Redes Neurais Artificiais classificação e regressão	Mapas auto organizáveis detecção de características e agrupamento
Redes Neurais Convolucionais visão computacional	Boltzmann machines sistemas de recomendação redução de dimensionalidade
Redes Neurais Recorrentes análise de séries temporais	Autoencoders redução de dimensionalidade
	Redes adversariais generativas geração de imagens

Redes neurais artificiais

Perceptron e redes neurais de camada única

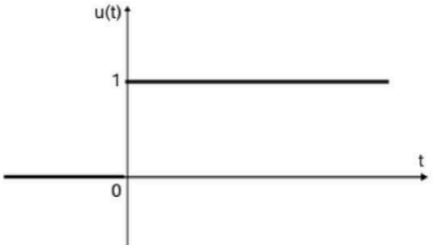
Também chamado de neurônio artificial, o perceptron imita o funcionamento de um neurônio humano. Ele recebe entradas associadas a um peso e calcula suas somas ponderadas. O resultado dessa soma passa por uma função de ativação, como a função degrau unitário, se o resultado dessa função for suficiente o perceptron é ativado passa sua saída adiante.

Função soma, tal que xi serão entradas e wi serão os pesos:

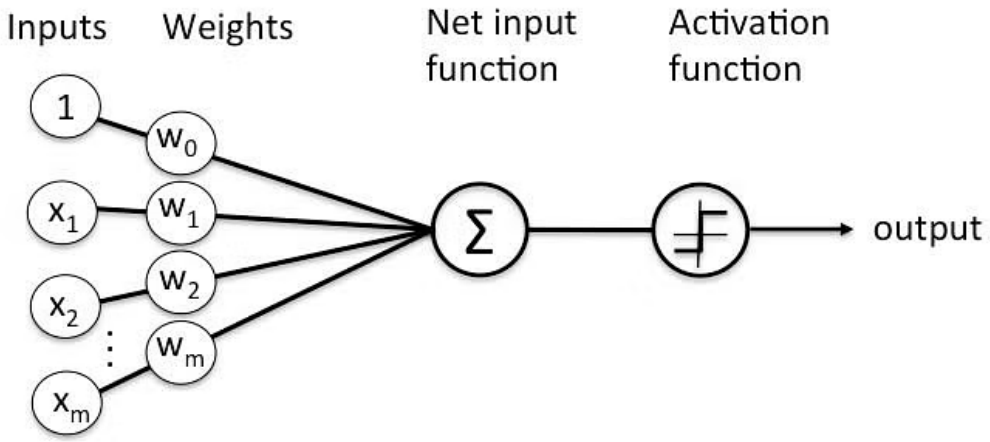
$$soma = \sum_{i=1}^{n} xi * wi$$

Função degrau unitário:

$$u(t) = \begin{cases} 0, & t<0 \\ 1, & t>0 \end{cases}$$



Assim, uma representação gráfica do perceptron é dada por



(1)

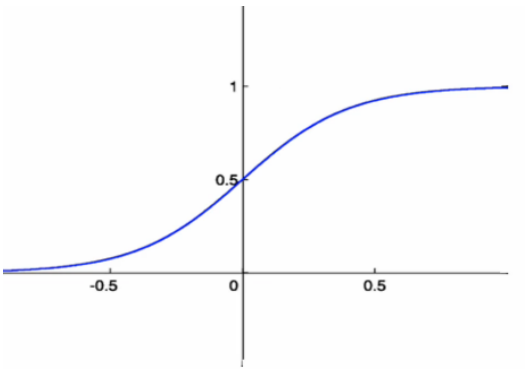
Redes neurais multicamadas

Para solucionar problemas mais complexos, pode-se encadear mais de uma camada de perceptrons e alterar a função de ativação.

Alguns exemplos de função de ativação (onde x é o resultado da função soma) são:

Função sigmóide:

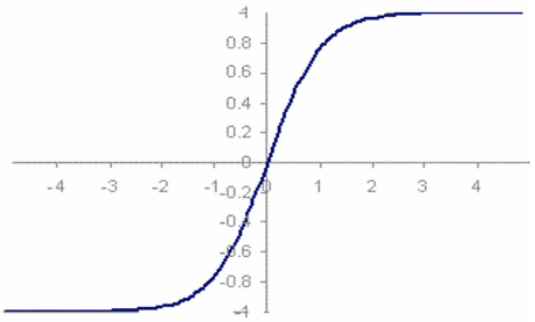
- **Faixa de valores**: (0, 1)
- **Uso**: Em camadas de saída para problemas de classificação binária (ex.: detecção de spam).
- **Vantagens**: Interpretação probabilística (valores entre 0 e 1).
- **Desvantagens**: Pode causar o problema de gradientes desaparecendo em redes profundas, tornando o treinamento lento.



$$y = \frac{1}{1 + e^{-x}}$$

Função tangente hiperbólica:

- **Faixa de valores**: (-1, 1)
- **Uso**: Em camadas intermediárias, especialmente em redes recorrentes (RNNs).
- **Vantagens**: Centraliza os dados em torno de zero, o que muitas vezes acelera a convergência.
- **Desvantagens**: Também pode sofrer do problema de gradientes desaparecendo, mas menos severamente que a sigmoid.



$$Y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

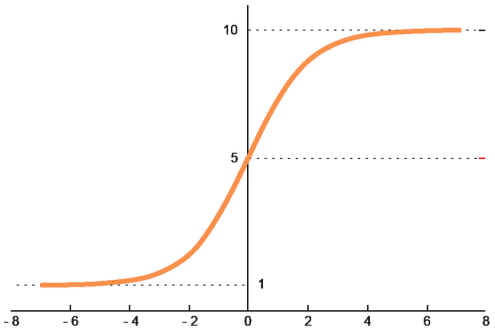
Função RELU:

- **Faixa de valores**: [0, ∞)
- **Uso**: Em camadas intermediárias de redes profundas, como CNNs e redes feedforward.
- **Vantagens**: Simples e eficiente, ajuda a evitar o problema de gradientes desaparecendo, e acelera a convergência.
- **Desvantagens**: Pode sofrer do problema de "neurônios mortos", onde um grande número de unidades pode parar de aprender (quando o valor é sempre negativo).

$$Y = \max(0, x)$$

Função SoftMax:

- **Faixa de valores**: (0, 1), mas a soma das saídas é 1.
- **Uso**: Em camadas de saída para problemas de classificação multi-classes.
- **Vantagens**: Fornece uma distribuição de probabilidade sobre várias classes.
- **Desvantagens**: Mais computacionalmente caro que a sigmoid ou ReLU.



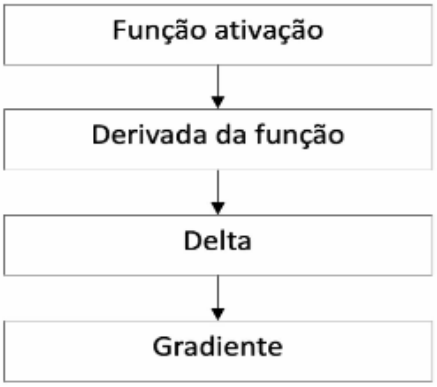
$$Y = \frac{e(x)}{\sum e(x)}$$

Dessa forma, a rede neural precisa de memória e aprendizado para superar obstáculos, assim, recursivamente, o próximo peso é definido com base no anterior de forma que:

peso(n + 1) = peso(n) + (taxaAprendizagem * entrada * erro)

Dessa forma a rede neural encontra o melhor conjunto de pesos para um determinado problema e ganha plasticidade em aprender novas informações.

O processo matemático para a definição de pesos segue a seguinte sequência:



Erro

O erro pode ser obtido através de:

erro = resultadoEsperado - resultadoObtido

Outras formas de obtenção de erro que penalizam mais os erros grandes são:

Mean Squared Error(MSE),:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(f_i - y_i)^2$$

Root Mean Squared Error(RMSE),

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(f_i - o_i)^2}$$

:

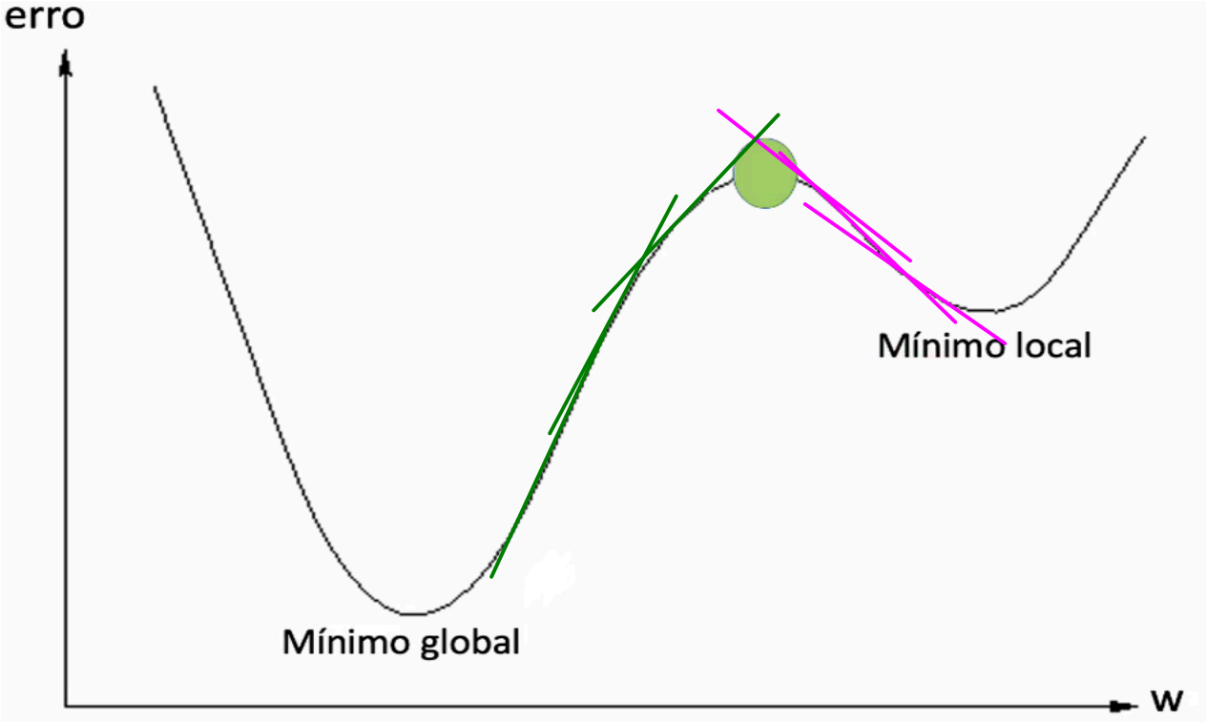
Sendo que a média absoluta da rede neural é a divisão do erro pela quantidade de classes, um bom treinamento da rede visa minimizar o erro a cada época treinada, otimizando os pesos.

Delta e Gradiente

Para ser possível a atualização dos pesos para redução do erro utiliza-se o gradiente. O gradiente segue a derivada do erro em relação aos pesos, ele indica a direção e a magnitude da mudança que devemos aplicar aos pesos para reduzir o erro.

No exemplo da figura abaixo, o gradiente é o mínimo global, os pesos são definidos para chegar até o erro mínimo, como as derivadas à esquerda (linha verde) são maiores

que à direita (linha rosa) os pesos serão diminuídos



Assim, a **derivada da função sigmóide** que irá indicar a direção e magnitude da mudança de pesos é dada por:

$$d = y * (1 - y)$$

Delta

Após o cálculo da derivada da função de ativação deve-se ponderar seu resultado com o erro de forma a calcular o valor delta que pode ser então expresso por:

delta = erro * derivadaAtivacao

Backpropagation

Ajusta os pesos das redes neurais multicamadas, calculando como o erro na saída da rede deve ser propagado de volta para ajustar os pesos em cada camada. Ele utiliza derivadas da função de ativação e gradientes para se orientar. Pode ser calculado da seguinte forma:

peso(n+1) = (peso(n) * momento) + (entrada * delta * taxa de aprendizagem)

Os parâmetros momento e taxa de aprendizagem podem acelerar a aprendizagem.

Batch gradient descent vs Stochastic gradient descent

A forma de atualização de pesos se dá principalmente de duas maneiras. Ou é calculado o erro para todos os registros e só então os pesos são atualizados, esse método

é chamado de Batch gradient descent, ou a cada registro calculado os pesos já são atualizados, esse método é chamado de Stochastic gradient descent.

O método Stochastic além de ser mais rápido, ajuda na prevenção de mínimos locais, já o método Batch pode trazer mais precisão na análise ao escolher o número de registros a se mandar por vez. Assim, é necessário definir além do *Learning rate* e das *Epochs* o tamanho do lote ou *Batch size* dos registros a serem computados

Bias

É um parâmetro adicional adicionado a cada neurônio, além das entradas ponderadas, que dá mais espaço para a rede para ajustar seus resultados.

Overfitting e Dropout

Para evitar o supertreinamento em um conjunto reduzido de dados na qual o modelo não consegue mais generalizar-se (overfitting), pode-se utilizar a técnica de dropout que consiste em ignorar uma fração dos neurônios do modelo.

```
def criar_modelo(optimizer, loss, kernel_initializer, activation, neurons):
    model = Sequential([tf.keras.layers.InputLayer(shape=(30,)),
            tf.keras.layers.Dropout(rate = 0.2), # eliminar 20% dos dados de
entrada
            tf.keras.layers.Dense(units=neurons, activation=activation,
kernel_initializer=kernel_initializer),
            tf.keras.layers.Dense(units=neurons, activation=activation,
kernel_initializer=kernel_initializer),
            tf.keras.layers.Dropout(rate = 0.2), # eliminar 20% dos dados de
entrada
            tf.keras.layers.Dense(units=1, activation='sigmoid')])
    model.compile(optimizer=optimizer, loss=loss, metrics=['binary_accuracy'])
    return model
```

Tuning

Para obter um melhor conjunto de parâmetros no treinamento da rede neural, pode-se utilizar o GridSearch.

```
# No dicionário parâmetros, os parâmetros que serão passados para a função
criar modelo são prefixados com model__. Esse prefixo é uma convenção usada
pelo Scikit-Learn para indicar que esses parâmetros devem ser passados para a
função de criação do modelo, ou seja,para a função criar_modelo:
```

```
parametros={
    'batch_size': [10,30],
    'epochs': [50, 100],
    'model__optimizer': ['adam', 'sgd'],
    'model__loss': ['binary_crossentropy'],
    'model__kernel_initializer': ['random_uniform', 'normal'],
    'model__activation': ['relu'],
    'model__neurons': [16,30]
}

grid_search = GridSearchCV(estimator=rede_neural, param_grid=parametros,
scoring='accuracy', cv=5)

grid_search = grid_search.fit(X,y)
melhores_parametros = grid_search.best_params_
print(melhores_parametros)
```

{'batch_size': 30, 'epochs': 50, 'model__activation': 'relu', 'model__kernel_initializer': 'normal', 'model__loss': 'binary_crossentropy', 'model__neurons': 16, 'model__optimizer': 'adam'}

Referências

BANOULA, M. What is Perceptron: A Beginners Guide for Perceptron. Disponível em: <https://www.simplilearn.com/tutorials/deep-learning-tutorial/perceptron>.