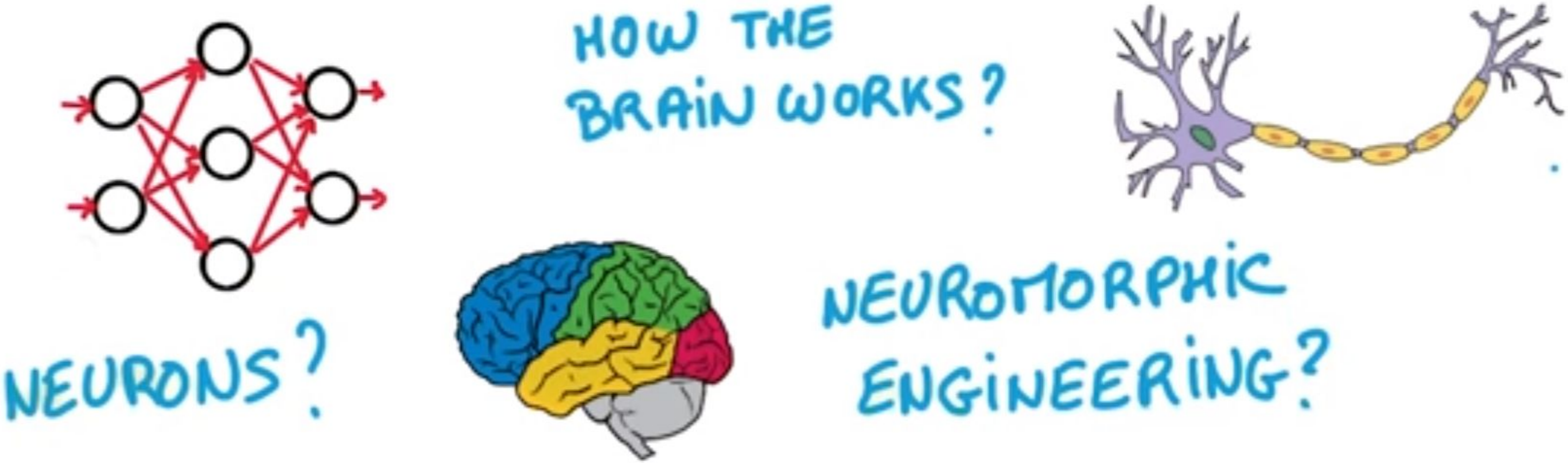
Redes Neurais Artificiais



Redes Neurais: Introdução à uma Rede Neural Simples

O diagrama abaixo mostra uma rede simples. A combinação linear dos pesos, inputs e viés formam o input *h*, que então é passado pela função de ativação *f*(*h*), gerando o output final do perceptron, etiquetado como *y*.

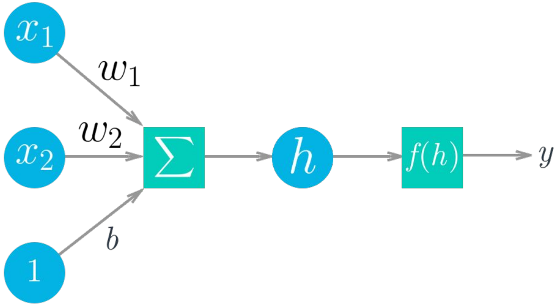


Diagrama de uma rede neural simples. Círculos são unidades, caixas são operações.

O que faz as redes neurais possíveis, é que a função de ativação, *f*(*h*) pode ser qualquer função, não apenas a função degrau.

Rede Neural Simples

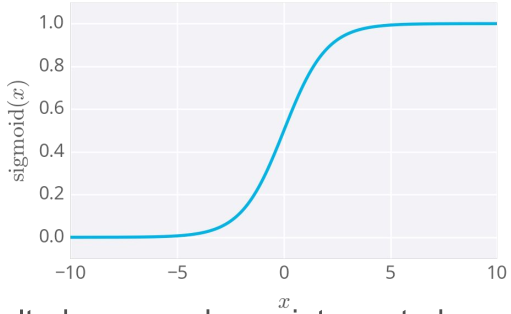
Por exemplo, caso $f(h)=h$, o output será o mesmo que o input. Agora o output da rede é

$$y = \sum_i w_i x_i + by = \sum i\, wi\, xi + b$$

Essa equação deveria ser familiar para você, pois é a mesma do modelo de regressão linear!

Outras funções de ativação comuns são a função logística (também chamada de sigmóide), tanh e a função softmax. Nós iremos trabalhar principalmente com a função sigmóide pelo resto dessa aula:

$$\text{sigmoide}(x) = 1/(1 + e^{-x})$$



A função sigmóide só retorna números entre 0 e 1 e além disso tem um resultado que pode ser interpretado como uma probabilidade de sucesso.

Rede Neural Simples

Exercício de rede simples

Abaixo, você usará o Numpy para calcular o output de uma rede simples com dois nós de input e um nó de output com uma função de ativação sigmóide. Para isso, será necessário:

- Implementar a função sigmóide.
- Calcular o output da rede.

Para a exponenciação, é possível utilizar a função do numpy, `np.exp`.

E o output da rede é $y = f(h) = \mathrm{sigmoid}(\sum_i w_i x_i + b) y = f(h) = sigmoid(\sum i \, wi \, xi + b)$

Para a soma dos pesos, é possível fazer uma multiplicação e soma elemento à elemento simples, ou então usar a [função de produto escalar function] do Numpy.

Rede Neural Simples

```python
import numpy as np

def sigmoid(x):
    # TODO: Implement sigmoid function
    pass

inputs = np.array([0.7, -0.3])
weights = np.array([0.1, 0.8])
bias = -0.1

# TODO: Calculate the output
output = None

print('Output:')
print(output)
```

Rede Neural Simples

Gradiente Descendente

Queremos que a rede faça previsões o mais próximas possíveis dos valores reais. Para medir isso, precisamos de uma medida de quão distantes as previsões estão da verdade, ou seja, um método de calcular o **erro**. Uma medida comum é a soma quadrática dos erros (SQE):

$$E = \frac{1}{2} \sum_{\mu} \sum_{j} \left[y_j^{\mu} - \hat{y}_j^{\mu} \right]^2$$

onde j representa as unidades de output da rede e μ é a soma de todos os dados. Então soma-se essas diferenças quadradas para cada dado. Isso resulta no erro médio para todos os outputs previstos em relação a todos os dados.

Rede Neural Simples

Gradiente Descendente

Lembre-se que o output de uma rede neural, a previsão, sempre depende dos pesos $\hat{y}_j^\mu = f\left(\sum_i w_{ij} x_i^\mu\right)$

e também o erro depende dos pesos $E = \frac{1}{2} \sum_\mu \sum_j \left[y_j^\mu - f\left(\sum_i w_{ij} x_i^\mu\right)\right]^2$

Nós queremos que o erro de previsão da rede seja o menor possível e os pesos são as alavancas que podemos ajustar para fazer isso acontecer.

Nosso objetivo é encontrar os pesos ***wij*** que minimizem o erro quadrático E. Para fazer isso com redes neurais, tipicamente o que se usa é o **gradiente descendente.**

Rede Neural Simples

Gradiente Descendente

Com o gradiente descendente, nós damos pequenos passos em direção ao objetivo. Neste caso, queremos mudar os pesos a cada passo para reduzir o erro.

***Sugestão: Dê uma olhada nas aulas do Khan Academy sobre este assunto.**

O gradiente é uma derivada generalizada para funções com mais do que uma variável. Nós podemos usar o cálculo para encontrar o gradiente de qualquer ponto na nossa função de erro, a qual depende dos pesos dos inputs.

Rede Neural Simples

Gradiente Descendente: o código

Como vimos antes, a atualização de um peso pode ser calculada da seguinte maneira:

$$\delta x_i \Delta wi = \eta, \delta xi$$

com o termo δ para erro como:

$$\delta = (y - y) f'(h) = (y - y) f'(\sum wi \, xi)$$

Lembre-se, na equação acima $(y - \hat{y})$ é o erro do output, e $f'(h)$ se refere à derivada da função de ativação, ***f(h).*** Podemos chamar essa derivada de gradiente da saída.

Rede Neural Simples

Gradiente de descida: o código

Agora, escrevendo o código levando em conta o caso de apenas uma unidade de saída e a função sigmóide como função de ativação $f(h)$.

*No GoogleClassroom fazer a trarefa 5 baseado no código ao lado.

```
# Definindo a função sigmóide para ativações
def sigmoid(x):
    return 1/(1+np.exp(-x))

# Derivada da função sigmóide
def sigmoid_prime(x):
    return sigmoid(x) * (1 - sigmoid(x))

# Dados de Input
x = np.array([0.1, 0.3])
# Alvo
y = 0.2
# Peso do Input para o Output
weights = np.array([-0.8, 0.5])

# Taxa de aprendizado, eta na equação de passo
learnrate = 0.5

# a combinação linear feita no nó (h em f(h) e f'(h))
h = x[0]*weights[0] + x[1]*weights[1]
# or h = np.dot(x, weights)

# O output da rede neural (y-circunflexo)
nn_output = sigmoid(h)

# O erro do output (y - y-circunflexo)
error = y - nn_output

# gradiente do output (f'(h))
output_grad = sigmoid_prime(h)

# termo do error (delta minúsculo)
error_term = error * output_grad

# passo do Gradiente descendente
del_w = [ learnrate * error_term * x[0],
          learnrate * error_term * x[1]]
# or del_w = learnrate * error_term * x
```

Rede Neural Simples - exemplo.

Implementando o gradiente de descida

Sabemos como atualizar pesos: $\Delta w_{ij} = \eta * \delta_j * x_i$

Você aprendeu como implementar isso para uma única atualização, mas como traduzir esse código de modo que ele calcule muitas atualizações de peso de modo que a rede aprenda?

Aqui estão alguns links úteis para acompanhar a implementação do código:

- Visualização do gradiente descendente
- Erro quadrático médio
- Derivadas parciais em relação a ***b*** e ***m***