REDES NEURAIS

Introdução à uma Rede Neural Simples

Histórico

As primeiras informações sobre neurocomputação surgiram em 1943, em artigos do neuroanatomista e psiquiatra Warren McCulloch, do Instituto Tecnológico de Massachusetts, e do matemático Walter Pitts, da Universidade de Illinois. Os autores fizeram uma analogia entre células nervosas vivas e o processo eletrônico, em um trabalho publicado sobre "neurônios formais"; simulando o comportamento do neurônio natural, no qual o neurônio possuía apenas uma saída, que era uma função da soma de valor de suas diversas entradas. O trabalho consistia num modelo de resistores variáveis e amplificadores, representando conexões sinápticas de um neurônio biológico.

- Máquina de Turing (1936).
- *The General and Logical Theory of Automata*, no final da década de 1940, escrito por Von Neumann.
- O livro Cybernetics de Wiener publicado em 1948 descreve alguns conceitos sobre controle, comunicação e processamento estatístico de sinais.
- Em meados de 1948, o polonês Stanilaw Jaśkowskipor, discípulo de Łukasiewicz, publicou estudos sobre cálculo proposicional paraconsistente.
- Em 1949 Hebb introduziu a capacidade de aprender através de seu livro "The Organization of Behavior".
- Em 1951, foi construído o primeiro neuro computador denominado Snark, por Mavin Minsky.
- O livro de Ashby (Design for a Brain: The Origin of Adaptive Behavior) que foi publicado em 1952.
- Gabor, um dos pioneiros da teoria da comunicação e o inventor da holografia, propôs em 1954 a ideia de um filtro adaptativo não linear.
- Em 1954-1955, Cragg e Tamperley observaram que os átomos em uma rede têm seus spins apontando "para cima" ou "para baixo", assim como os neurônios podem ser "disparados" (ativados) ou "não disparados" (quiescentes).

Histórico

- Em 1956 no Darthmouth College nasceram os dois paradigmas da Inteligência Artificial, a simbólica e o conexionista.
- Em 1958, Frank Rosenblatt criou o Perceptron, um modelo cognitivo que consistia de unidades sensoriais conectadas a uma única camada de neurônios de Warren McCulloch e Pitts, capaz de aprender tudo o que pudesse representar.
- No início da década de 1960, Widrow e Hoff publicam um artigo no qual especificam um neurônio artificial baseado no modelo de McCulloch e Pitts, denominado ADALINE.
- Em 1967, Cowan caracterizou o disparo "sigmóide" e a condição de disparo suave para um neurônio baseando-se na função logística.
- Em 1967-1968, Grossberg envolvendo equações não-lineares de diferenças/diferenciais introduziu o modelo aditivo de um neurônio e como base para a memória de curto prazo explorou o uso do modelo.
- A publicação de Perceptrons de Minsky e Papert, em 1969, expôs as limitações do modelo de Rosenblatt, provando que tais redes não são capazes de resolver uma ampla classe de problemas devido às restrições de representação.

Nos anos 70, não ocorreram grandes progressos nos estudos de redes neurais devido um entusiasmo exagerado de muitos pesquisadores, que passaram a publicar mais e mais artigos e livros que faziam uma previsão pouco confiável para a época, sobre máquinas tão poderosas quanto o cérebro humano que surgiriam em um curto espaço de tempo. Isto tirou quase toda a credibilidade dos estudos desta área e causou grandes aborrecimentos aos técnicos de outras áreas.

Histórico

- Em 1972 foi introduzido de forma independente por Amari o modelo aditivo de um neurônio para estudar o comportamento dinâmico de elementos semelhantes a neurônios conectados aleatoriamente.
- Ainda em 1972, Wilson e Cowan derivaram equações diferenciais não-lineares acopladas correspondentes à dinâmica de populações localizadas no espaço, contendo neurônios tanto excitadores como inibitórios.
- 1974: WERBOS lançou bases para o algoritmo de retropropagação (backpropagation).
- Em 1975, Little e Shaw descreveram e usaram o modelo probabilístico de um neurônio, disparando ou não um potencial de ação, para desenvolver uma teoria da memória de curto prazo.
- Em 1977, Andereson, Silverstein, Ritz e Jones, sugeriram o modelo do estado cerebral em uma caixa (BSB, brain-state-in-a-box), compondo-se de uma rede associativa simples acoplada a uma dinâmica não-linear.
- Em 1982, a introdução do modelo conexionista conhecido pelo nome de seu idealizador, John Hopfield.
- Teuvo Kohonen publicou um artigo descrevendo uma rede neural artificial baseada em auto-organização e nas características de aprendizado adaptativo do cérebro humano.
- Hinton e Seynowsky, em 1983, estenderam o modelo de Hopfield com a incorporação de dinâmica estocástica. Este modelo de rede neural passou a ser conhecido como Máquina de Boltzmann.
- Em meados da década de 1980 surgiu a descrição do algoritmo de treinamento backpropagation (Rumelhart, Hinton, & Williams 1986).

Histórico

- Em 1972 foi introduzido de forma independente por Amari o modelo aditivo de um neurônio para estudar o comportamento dinâmico de elementos semelhantes a neurônios conectados aleatoriamente.
- Ainda em 1972, Wilson e Cowan derivaram equações diferenciais não-lineares acopladas correspondentes à dinâmica de populações localizadas no espaço, contendo neurônios tanto excitadores como inibitórios.
- 1974: WERBOS lançou bases para o algoritmo de retropropagação (backpropagation).
- Em 1975, Little e Shaw descreveram e usaram o modelo probabilístico de um neurônio, disparando ou não um potencial de ação, para desenvolver uma teoria da memória de curto prazo.
- Em 1977, Andereson, Silverstein, Ritz e Jones, sugeriram o modelo do estado cerebral em uma caixa (BSB, brain-state-in-a-box), compondo-se de uma rede associativa simples acoplada a uma dinâmica não-linear.
- Em 1982, a introdução do modelo conexionista conhecido pelo nome de seu idealizador, John Hopfield.
- Teuvo Kohonen publicou um artigo descrevendo uma rede neural artificial baseada em auto-organização e nas características de aprendizado adaptativo do cérebro humano.
- Hinton e Seynowsky, em 1983, estenderam o modelo de Hopfield com a incorporação de dinâmica estocástica. Este modelo de rede neural passou a ser conhecido como Máquina de Boltzmann.
- Em meados da década de 1980 surgiu a descrição do algoritmo de treinamento backpropagation (Rumelhart, Hinton, & Williams 1986).

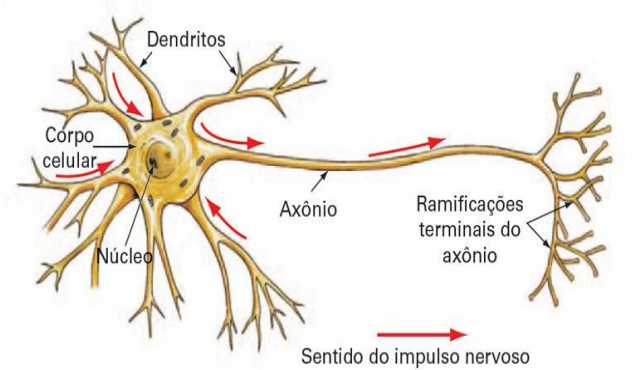
Histórico

- Em 1995, Dana Cortes e Vladimir Vapnik desenvolveram a máquina de vetor de suporte ou Support Vector Machine (um sistema para mapear e reconhecer dados semelhantes).
- O LSTM (Long-Short Term Memory) para redes neurais recorrentes foi desenvolvido em 1997, por Sepp Hochreiter e Juergen Schmidhuber.
- Em torno do ano 2000, apareceu o problema conhecido como Vanishing Gradient.
- Em 2001, um relatório de pesquisa do Grupo META (agora chamado Gartner) descreveu os desafios e oportunidades no crescimento do volume de dados.
- Em 2009, Fei-Fei Li, professora de IA em Stanford na Califórnia, lançou o ImageNet e montou uma base de dados gratuita de mais de 14 milhões de imagens etiquetadas.
- Até 2011, a velocidade das GPUs aumentou significativamente, possibilitando a formação de redes neurais convolutivas “sem” o pré-treino camada por camada.
- Também em 2012, o Google Brain lançou os resultados de um projeto incomum conhecido como The Cat Experiment.

...

O Neurônio Biológico

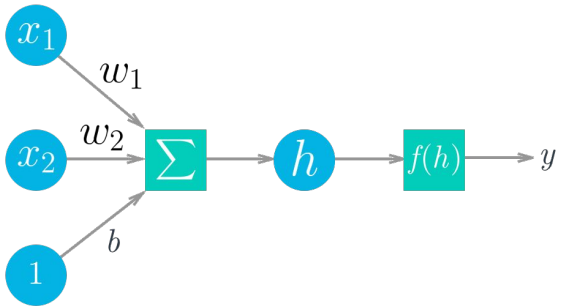
O neurônio é a unidade básica do cérebro humano, sendo uma célula especializada na transmissão de informações, pois nelas estão introduzidas propriedades de excitabilidade e condução de mensagens nervosas. O neurônio é constituído por 3 partes principais: a soma ou corpo celular, do qual emanam algumas ramificações denominadas de dendritos, e por uma outra ramificação descendente da soma, porém mais extensa, chamada de axônio. Nas extremidades dos axônios estão os nervos terminais, pelos quais é realizada a transmissão das informações para outros neurônios. Esta transmissão é conhecida como sinapse.



Rede Neural Simples

A partir da estrutura e funcionamento do neurônio biológico, pesquisadores tentaram simular este sistema em computador. O modelo mais bem aceito foi proposto por Warren McCulloch e Walter Pitts em 1943, o qual implementa de maneira simplificada os componentes e o funcionamento de um neurônio biológico. Em termos simples, um neurônio matemático de uma rede neural artificial é um componente que calcula a soma ponderada de vários inputs, aplica uma função e passa o resultado adiante.

O diagrama ao lado 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*.



Rede Neural Simples

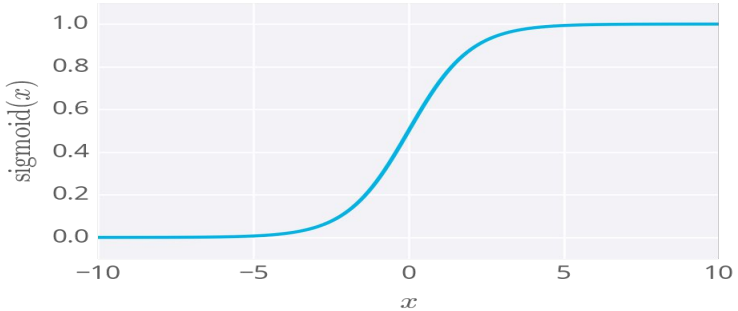
Por exemplo, caso $f(h)=h$, o output será o mesmo que o input. Agora o output da rede é

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

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.

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



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

Tarefa 1 - 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 $$y = f(h) = \text{sigmoid}(\sum_i w_i x_i + b) y = f(h) = sigmoid(\sum i\, wi\, xi + b)$$

Para a soma dos pesos, é possível fazer uma multiplicação e soma elemento a 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

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

Tarefa 2 - Gradiente de descida

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

```
# 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

Aplicando 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?

Rede Neural Simples

Aplicando o gradiente de descida

Média do erro quadrado

Iremos fazer uma pequena mudança no modo como calculamos o erro aqui. Em vez de usarmos o SQE, usaremos a **média** dos erros quadrados (MEQ). Agora que estamos usando muitos dados, somar todos os pesos e todos os passos pode criar passos muito grandes, que fazem o gradiente descendente divergir. Para compensar isso, será necessário usar uma taxa de aprendizagem muito pequena. Em vez disso, nós dividiremos pelo número de dados observados, m para tirar a média. Desse modo, não importa quantos dados usemos, as taxas de aprendizado estarão tipicamente no intervalo entre 0,01 e 0,001. Então, podemos usar a MEQ (abaixo) para calcular o gradiente e o resultado da mesma forma que antes, porém, em um valor médio, em vez de uma somatória.

$$E = \frac{1}{2m} \sum_{\mu} (y^{\mu} - \hat{y}^{\mu})^2$$

Rede Neural Simples

Algoritmo geral para atualizar os pesos com gradiente descendente:

- Defina o passo como zero: $\Delta w_i = 0$
- Para cada observação nos dados de treinamento:
  - Passe adiante na rede, calculando o output $\hat{y} = f(\sum_i w_i x_i)$
  - Calcule o erro para aquela unidade de output, $\delta = (y - \hat{y}) * f'(\sum_i w_i x_i)$
  - Atualize o passo $\Delta w_i = \Delta w_i + \delta x_i$
- Atualize os pesos $w_i = w_i + \eta \Delta w_i / m$ onde η é a taxa de aprendizado e m é o número de observações. Aqui estamos tirando a média dos passos para ajudar a reduzir quaisquer variações nos dados de treinamento
- Repita por e rodadas.

Rede Neural Simples

Continuação… Algoritmo Geral:

É possível também atualizar os pesos para cada observação ao invés de tirar a média dos passos depois de passar por todas as observações.

Lembre-se que estamos usando a função de ativação sigmóide,

$$f(h) = 1/(1 + e^{-h})$$

E que o gradiente de uma sigmóide é

$$f'(h) = f(h)(1 - f(h)),$$

onde h é o input da unidade de output,

$$h = \sum_i w_i x_i$$

Rede Neural Simples

Implementando com Numpy

Em grande parte, isso é bem simples de fazer com Numpy. Primeiro, é necessário inicializar os pesos. Queremos que eles sejam tão pequenos quanto o input para uma função sigmóide é na região linear próxima ao 0 e não espremido nas pontas alta e baixa. Também é importante inicializar-los aleatoriamente de modo que todos comecem de lugares diferentes e divirjam, quebrando simetrias. Portanto, inicializamos os pesos de uma distribuição normal com centro em 0. Um bom valor para essa escala é $1/\sqrt{n}$ onde n é o número de unidades de input. Isso mantém o input da sigmóide baixo para números de unidades de input maiores.

```
weights = np.random.normal(scale=1/n_features**.5, size=n_features)
```

Rede Neural Simples

Cont... Implementando com Numpy

O Numpy tem uma função que calcula o produto escalar de dois vetores, o que convenientemente calcula h para nós. O produto escalar multiplica dois vetores elemento por elemento, o primeiro elemento do vetor 1 é multiplicado pelo primeiro elemento do vetor 2 e assim por diante. Então, cada produto é somado.

```
# input to the output layer
output_in = np.dot(weights, inputs)
```

Por fim, podemos atualizar Δw_i e w_i incrementando-os com `weights += ...` o que é um jeito rápido de escrever `weights = weights + ...`.

Rede Neural Simples

Por fim, podemos atualizar Δw_i e w_i incrementando-os com `weights += ...` o que é um jeito rápido de escrever `weights = weights + ...`.

Dica de eficiência!

É possível poupar alguns cálculos já que estamos usando uma sigmóide. Na função sigmóide, $f'(h) = f(h)(1-f(h))$.

Isso significa que uma vez calculado $f(h)$, a ativação da unidade de output, você pode usá-la para calcular o gradiente para o gradiente de erro.

Rede Neural Simples

Tarefa 3 - Exercício de programação

O objetivo é treinar a rede até que ela alcance um mínimo na média de erros quadrados (MEQ) dos dados de treinamento. Você deverá implementar:

- O output da rede: `output`.
- O erro do output: `error`.
- O termo do erro: `error_term`.
- Atualizar o passo: `del_w +=`.
- Atualizar os pesos: `weights +=`.

OBS: Para realizar o exercício, utilize os códigos disponíveis na tarefa 6. O arquivo a ser alterado é o gradient.py.

Perceptron Multicamadas - MPL

Implementando a camada oculta

Pré-requisitos

Abaixo, detalharemos a matemática das redes neurais em um perceptron multi-camada. Com múltiplos perceptrons, nós iremos utilizar vetores e matrizes. Para revisar estes tópicos, dê uma olhada em:

1. Introdução aos vetores da Khan Academy.
2. Introdução às matrizes da Khan Academy.

Perceptron Multicamadas - MPL

Cont… Implementando a camada oculta

Para inicializar esses pesos usando o Numpy, temos que fornecer o formato da matriz. Caso a variável `features` seja um vetor de duas dimensões contendo os dados de entrada:

```
# Número de observações e unidades de input
n_observacoes, n_inputs = features.shape
# Número de unidades ocultas
n_ocultas = 2
pesos_inputs_para_ocultas = np.random.normal(0, n_inputs**-0.5, size=(n_inputs, n_ocultas))
```

Isso cria um vetor 2D (ou seja, uma matriz) chamado `pesos_inputs_para_ocultas` com as dimensões `n_inputs` por `n_ocultas`. Lembre-se de que o input de uma unidade oculta é a soma de todos os inputs multiplicado pelos pesos da unidade oculta.

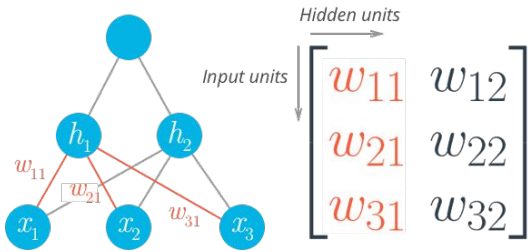
Perceptron Multicamadas - MPL

Implementando a camada oculta

Derivação

Antes, estavamos lidando apenas com um nó de output, o que simplifica o código. No entanto, agora que temos múltiplas unidades de input e múltiplas unidades ocultas, os pesos entre elas precisam de dois índices: w_{ij} sendo que i indica as unidades de input e j as unidades ocultas.

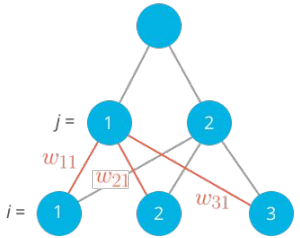
Por exemplo, a imagem a seguir mostra a nossa rede, com as unidades de input marcadas como x_1,x_2 e x_3 assim como as unidades ocultas marcadas como h_1 e h_2:



Perceptron Multicamadas - MPL

Cont… Implementando a camada oculta

Lembre-se de que o input de uma unidade oculta é a soma de todos os inputs multiplicado pelos pesos da unidade oculta. Então para cada unidade da camada oculta h_j, calcularemos o seguinte: $h_j = \sum_i w_{ij} x_i$



$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

Perceptron Multicamadas - MPL

Cont… Implementando a camada oculta

Para o input da segunda unidade da camada oculta, se calcula o produto escalar dos inputs com a segunda coluna da matriz de pesos. Esse padrão continua.

Com o Numpy, é possível fazer isso para todos os inputs e todos os outputs de uma vez usando `np.dot`

```
inputs_ocultos = np.dot(inputs, pesos_inputs_para_ocultas)
```

Você pode definir a matriz de pesos de modo que ela tenha as dimensões `n_ocultas` por `n_inputs` e então multiplicar de modo que os inputs formem um *vetor coluna*:

$$h_j = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Para uma melhor notação e mantendo os mesmos valores dos pesos, porém trocando os índices. Tenha em mente que essa é a mesma matriz de peso de antes, mas com um giro de modo que a primeira coluna agora é a primeira linha, a segunda coluna agora é a segunda linha.

Perceptron Multicamadas - MPL

Cont… Implementando a camada oculta

Criando um vetor coluna

Vimos antes que as vezes o desejado é ter um vetor coluna, ainda que os vetores Numpy, por padrão, funcionem como vetores linha. É possível fazer a transposta de qualquer vetor usando `arr.T`, mas para um vetor de uma dimensão, a transposta será um vetor linha. Ao invés disso, use `arr[:,None]` para criar um vetor coluna:

```
print(features)
> array([ 0.49671415, -0.1382643 ,  0.64768854])

print(features.T)
> array([ 0.49671415, -0.1382643 ,  0.64768854])

print(features[:, None])
> array([[ 0.49671415],
       [-0.1382643 ],
       [ 0.64768854]])
```

Perceptron Multicamadas - MPL

Cont… Implementando a camada oculta

Outra opção é criar vetores com duas dimensões, então usar `arr.T` para obter o vetor coluna.

```
np.array(features, ndmin=2)
> array([[ 0.49671415, -0.1382643 ,  0.64768854]])

np.array(features, ndmin=2).T
> array([[ 0.49671415],
       [-0.1382643 ],
       [ 0.64768854]])
```

Tarefa 4 - MPL 3X4x2

A seguir, você implementará uma rede 4x3x2 orientada a frente, com funções de ativação sigmóide em ambas as camadas.

Coisas a fazer:

- Calcular o input da camada oculta.
- Calcular o output da camada oculta.
- Calcular o input da camada de output.
- Calcular o output da rede.

Perceptron Multicamadas - MPL

Backpropagation

Agora chegamos ao problema de como fazer uma rede neural de múltiplas camadas *aprender*. Antes, vimos como atualizar os pesos usando o gradiente descendente. O algoritmo de Retropropagação (backpropagation daqui em diante) é apenas uma extensão disso, usando a regra de cadeia para encontrar o erro respeitando os pesos conectando a camada de input para a camada oculta (numa rede de duas camadas).

Para atualizar os pesos das camadas ocultas usando o gradiente descendente, é necessário saber quanto contribuiu cada unidade oculta para a produção daquele erro no output final. Uma vez que o output de uma camada é determinado pelos pesos entre camadas, o erro resultante de unidades é proporcional aos pesos ao longo da rede. Uma vez que sabemos o erro no output, nós usamos os pesos para trazê-lo de volta às camadas ocultas.

Perceptron Multicamadas - MPL

Cont… Backpropagation

Por exemplo, na camada de outuput, temos erros δ_k^o atribuídos para cada unidade de output k. Então, o erro atribuído para a unidade oculta j é igual aos erros dos outputs, proporcional aos pesos entre as camadas oculta e de output (levando o gradiente em conta):

$$\delta_j^h = \sum W_{jk} \delta_k^o f'(h_j)$$

Então, o passo do gradiente descendente é o mesmo que antes, apenas com os novos erros:

$$\Delta w_{ij} = \eta \delta_j^h x_i$$

onde w_{ij} são os pesos entre o inputs e a camada oculta e x_i são os valores de input da unidade. Esse formato continua válido para qualquer quantidade de camadas. O passos de peso são iguais ao tamanho do passo vezes o erro de output da camada vezes os valores de input daquela camada

$$\Delta w_{pq} = \eta \delta_{output} V_{in}$$

Aqui, temos o erro de output, δ_{output} , ao propagar os erros retrospectivamente de camadas mais altas. E os valores de input, V_{in} são os inputs da camada, as ativações da camada oculta para camada de output, por exemplo.

Perceptron Multicamadas - MPL

Exemplo no material auxiliar...

Implementação com Numpy

Você já possui a maior parte das coisas necessárias para implementar a backpropagation com o Numpy.

No entanto, antes só devíamos lidar com erros de uma unidade. Agora, na atualização dos pesos, temos que considerar o erro de *cada unidade* da camada oculta, δ_j: $\Delta w_{ij} = \eta \delta_j x_i$

Tarefa 5 - Backpropagation

Exercício de Backpropagation

Abaixo, você implementará o código para calcular uma rodada de atualização com backpropagation para dois conjuntos de pesos. Escrevi o andamento para frente, o seu objetivo é escrever o andamento para trás.

Coisas a fazer

- Calcular o erro da rede.
- Calcular o gradiente de erro da camada de output.
- Usar a backpropagation para calcular o erro da camada oculta.
- Calcular o passo de atualização dos pesos.

Perceptron Multicamadas - MPL

Implementando backpropagation

Agra que vimos que o erro da camada de output é

$$\delta_k = (y_k - \hat{y}_k) f'(a_k)$$

e que o erro da camada oculta é

$$\delta_j = \sum [w_{jk} \delta_k] f'(h_j)$$

Por enquanto iremos contar com uma rede simples com uma camada oculta e uma unidade de output. Eis o algoritmo geral para atualizar os pesos com backpropagation:

- Defina os passos de atualização dos pesos para cada camada como zero
  - Os pesos do input para oculta $\Delta w_{ij} = 0$
  - Os pesos da oculta para o output $\Delta W_j = 0$
- Para cada observação dos dados de treinamento:
  - Faça um andamento adiante pela rede, calculando o output $\hat{y}$
  - Calcule o gradiente de erro da unidade de output, $\delta^o = (y - \hat{y}) f'(z)$ em que $z = \sum_j W_j a_j$, o input da unidade de output.
  - Propague os erros para a camada oculta $\delta_j^h = \delta^o W_j f'(h_j)$
  - Atualize os passos dos erros:
    - \Delta W_j = \Delta W_j + \delta^o a_j
    - \Delta w_{ij} = \Delta w_{ij} + \delta^h_j a_i
- Atualize os pesos, onde η é a taxa de aprendizado e m é o número de observações:
  - W_j = W_j + \eta \Delta W_j / m
  - w_{ij} = w_{ij} + \eta \Delta w_{ij} / m
- Repita por e épocas.

Tarefa 6 - Backpropagation_2

Implementando o Backpropagation

Agora você implementará o algoritmo de backpropagation em uma rede treinada com os dados de admissão em universidades. Você possui tudo o que precisa para resolver esse exercício nos exercícios anteriores.

Seus objetivos aqui:

- Implementar o andamento adiante.
- Implementar o algoritmo de backpropagation.
- Atualizar os pesos.

Outras leituras

Backpropagation é fundamental para entender o deep learning. O TensorFlow e outras bibliotecas irão executar o backprop para você, mas é importante que você compreenda *de verdade* esse algoritmo. Nós ainda vamos falar sobre ele, mas aqui temos algumas referências extras para você:

- De Andrej Karpathy: Sim, você devia entender o backprop
- Também de Andrej Karpathy, uma palestra do curso CS231n de Stanford