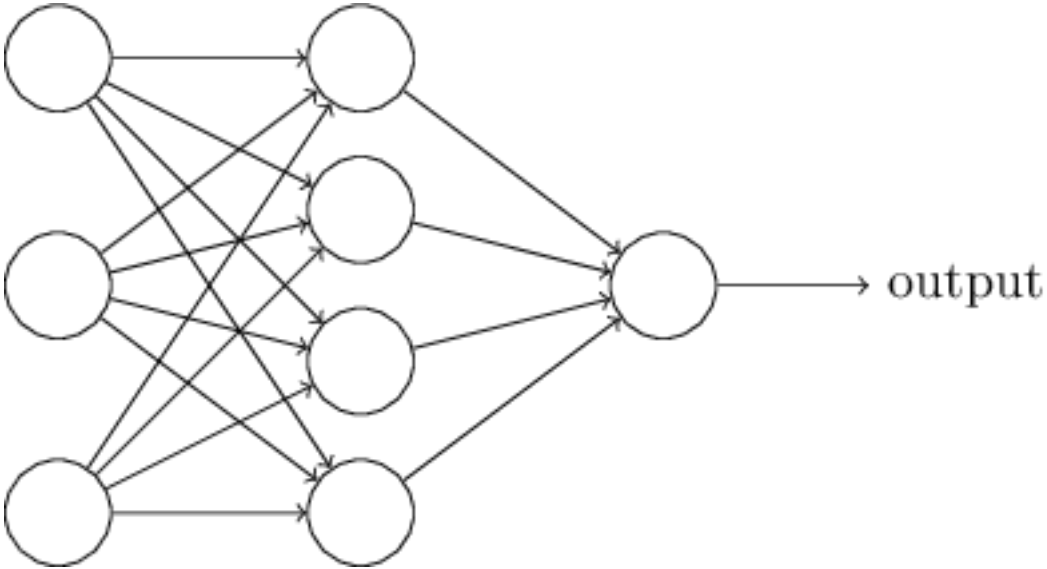
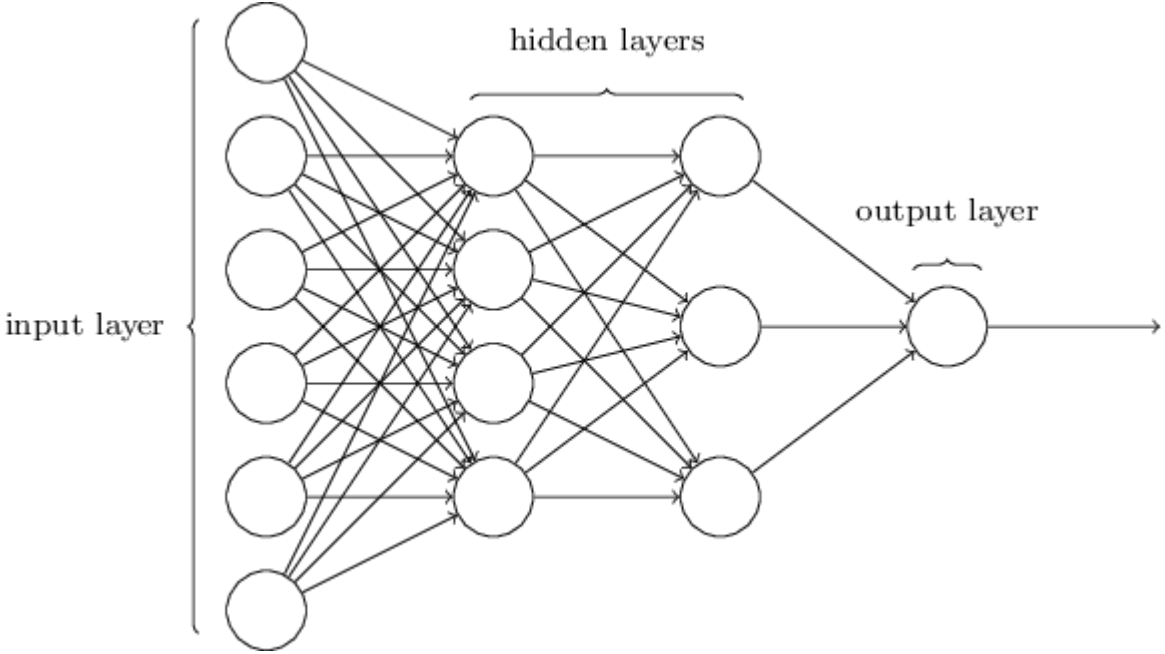
Redes neurais

A Arquitetura das Redes Neurais



A Arquitetura das Redes Neurais

Tais redes de camadas múltiplas são chamados de Perceptrons Multicamadas ou MLPs (Multilayer Perceptrons), ou seja, uma rede neural formada por Perceptrons (embora na verdade seja uma rede de neurônios sigmóides, como veremos mais adiante).



A Arquitetura das Redes Neurais

O design das camadas de entrada e saída em uma rede geralmente é direto. Por exemplo, suponha que estamos tentando determinar se uma imagem manuscrita representa um “9” ou não. Uma maneira natural de projetar a rede é codificar as intensidades dos pixels da imagem nos neurônios de entrada. Se a imagem for uma imagem em escala de cinza 64 x 64, teríamos 64 × 64 = 4.096 neurônios de entrada, com as intensidades dimensionadas adequadamente entre 0 e 1. A camada de saída conterá apenas um único neurônio com valores inferiores a 0,5 indicando que “a imagem de entrada não é um 9” e valores maiores que 0,5 indicando que “a imagem de entrada é um 9”.

A Arquitetura das Redes Neurais

1 – Redes Neurais Feed-Forward

Estes são o tipo mais comum de rede neural em aplicações práticas. A primeira camada é a entrada e a última camada é a saída. Se houver mais de uma camada oculta, nós as chamamos de redes neurais “profundas” (ou Deep Learning). Esses tipos de redes neurais calculam uma série de transformações que alteram as semelhanças entre os casos. As atividades dos neurônios em cada camada são uma função não-linear das atividades na camada anterior.

A Arquitetura das Redes Neurais

2 – Redes Recorrentes

Estes tipos de redes neurais têm ciclos direcionados em seu grafo de conexão. Isso significa que às vezes você pode voltar para onde você começou seguindo as setas. Eles podem ter uma dinâmica complicada e isso pode torná-los muito difíceis de treinar. Entretanto, estes tipos são mais biologicamente realistas.

Atualmente, há muito interesse em encontrar formas eficientes de treinamento de redes recorrentes. As redes neurais recorrentes são uma maneira muito natural de modelar dados sequenciais. Eles são equivalentes a redes muito profundas com uma camada oculta por fatia de tempo; exceto que eles usam os mesmos pesos em cada fatia de tempo e recebem entrada em cada fatia. Eles têm a capacidade de lembrar informações em seu estado oculto por um longo período de tempo, mas é muito difícil treiná-las para usar esse potencial.

A Arquitetura das Redes Neurais

3 – Redes Conectadas Simetricamente

Estas são como redes recorrentes, mas as conexões entre as unidades são simétricas (elas têm o mesmo peso em ambas as direções). As redes simétricas são muito mais fáceis de analisar do que as redes recorrentes. Elas também são mais restritas no que podem fazer porque obedecem a uma função de energia. As redes conectadas simetricamente sem unidades ocultas são chamadas de “Redes Hopfield”. As redes conectadas simetricamente com unidades ocultas são chamadas de “Máquinas de Boltzmann”.

Principais redes

- Redes Multilayer Perceptron
- Redes Neurais Convolucionais
- Redes Neurais Recorrentes
- Long Short-Term Memory (LSTM)
- Redes de Hopfield
- Máquinas de Boltzmann
- Deep Belief Network
- Deep Auto-Encoders
- Generative Adversarial Network
- Deep Neural Network Capsules (este é um tipo completamente novo de rede neural, lançado no final de 2017)

Machine Learning

O Aprendizado de Máquina (Machine Learning) é necessário para resolver tarefas que são muito complexas para os humanos. Algumas tarefas são tão complexas que é impraticável, senão impossível, que os seres humanos consigam explicar todas as nuances envolvidas. Então, em vez disso, fornecemos uma grande quantidade de dados para um algoritmo de aprendizado de máquina e deixamos que o algoritmo funcione, explorando esses dados e buscando um modelo que alcance o que os Cientistas de Dados e Engenheiros de IA determinaram como objetivo. Vejamos estes dois exemplos:

Machine Learning

- É muito difícil escrever programas que solucionem problemas como reconhecer um objeto tridimensional a partir de um novo ponto de vista em novas condições de iluminação em uma cena desordenada. Nós não sabemos qual programa de computador escrever porque não sabemos como ocorre o processo em nosso cérebro. Mesmo se tivéssemos uma boa ideia sobre como fazê-lo, o programa poderia ser incrivelmente complicado.

- É difícil escrever um programa para calcular a probabilidade de uma transação de cartão de crédito ser fraudulenta. Pode não haver regras que sejam simples e confiáveis. Precisamos combinar um número muito grande de regras fracas. A fraude é um alvo em movimento, mas o programa precisa continuar mudando.

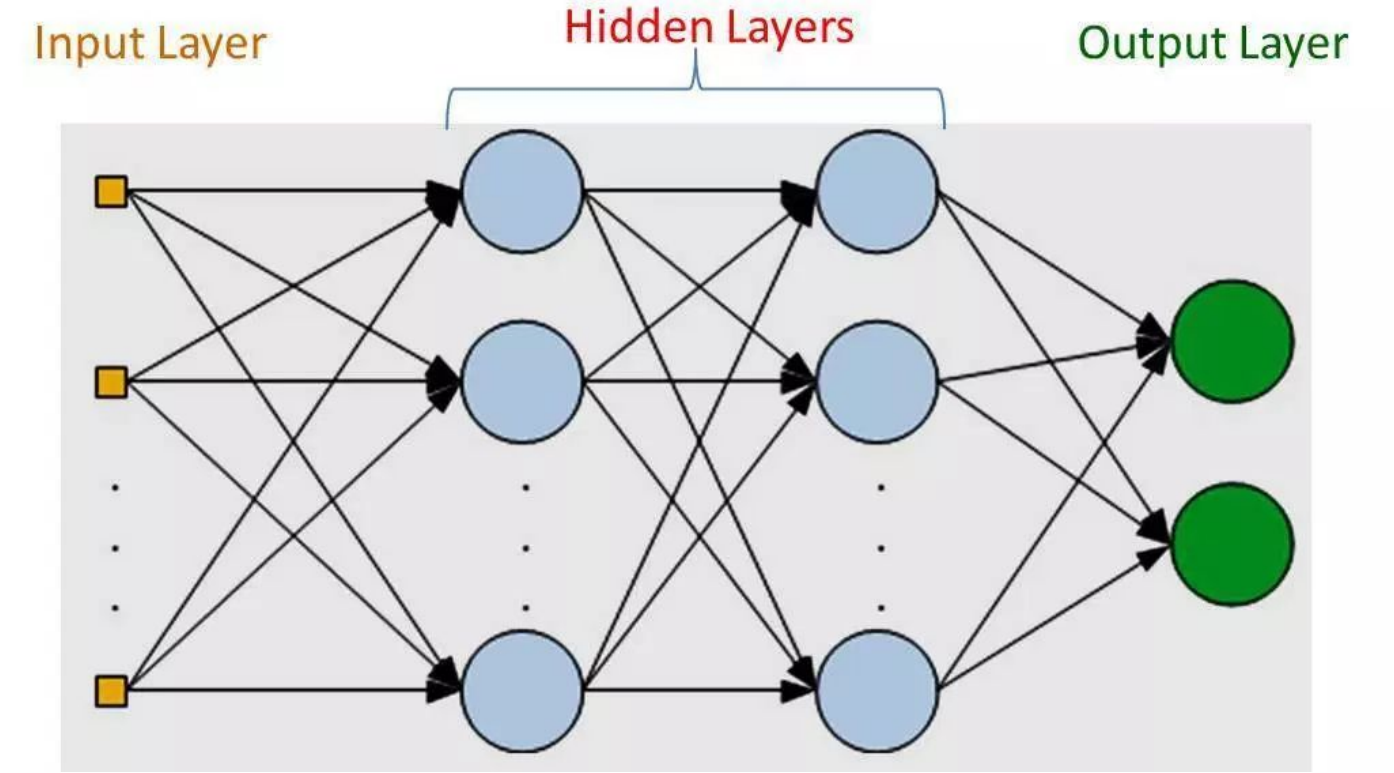
Machine Learning

Em vez de escrever um programa à mão para cada tarefa específica, nós coletamos muitos exemplos que especificam a saída correta para uma determinada entrada. Um algoritmo de aprendizagem de máquina recebe esses exemplos e produz um programa que faz o trabalho.

Alguns exemplos de tarefas melhor resolvidas pela aprendizagem de máquina incluem:

- Reconhecimento de padrões: objetos em cenas reais, identidades faciais ou expressões faciais, palavras escritas ou faladas.
- Detecção de anomalias: sequências incomuns de transações de cartão de crédito, padrões incomuns de leituras de sensores em máquinas de uma indústria têxtil.
- Previsão: preços de ações futuros ou taxas de câmbio, quais filmes uma pessoa gostaria de assistir, previsão de vendas.

Redes Multilayer Perceptrons



Perceptron x MLP

- Um Perceptron é um classificador linear
- A entrada geralmente é um vetor de recursos **x** multiplicado por pesos **w** e adicionado a um viés (ou bias) **b**

$$y = w * x + b$$

MLP

Um Multilayer Perceptron (MLP) é uma rede neural artificial composta por mais de um Perceptron.

- Entrada
- Oculta
- Saída

A MLP é uma espécie de “Hello World” da aprendizagem profunda: uma boa forma de começar quando você está aprendendo sobre **Deep Learning**.

MLP

As redes feed forward, como MLPs, são como ping-pong. Elas são principalmente envolvidas em dois movimentos, uma constante de ida e volta. Na passagem para a frente, o fluxo de sinal se move da camada de entrada através das camadas ocultas para a camada de saída e a decisão da camada de saída é medida em relação às saídas esperadas.

MLP

Na passagem para trás, usando o backpropagation e a regra da cadeia (Chain Rule), derivadas parciais da função de erro dos vários pesos e bias são reproduzidos através do MLP.

A rede continua jogando aquele jogo de ping-pong até que o erro não possa mais ser reduzido (chegou ao mínimo possível). Este estado é conhecido como convergência.

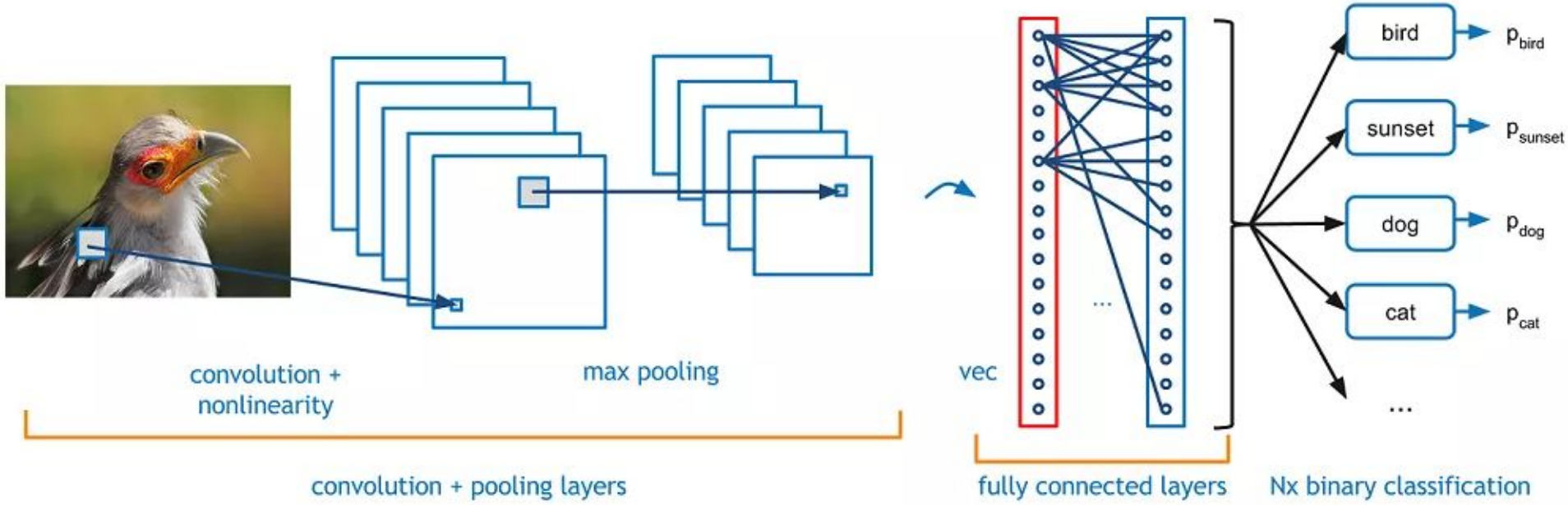
2- Redes Neurais Convolucionais

Em 1998, Yann LeCun e seus colaboradores desenvolveram um reconhecedor, realmente bom, para dígitos manuscritos chamado LeNet. Ele usou o backpropagation em uma rede feed forward com muitas camadas ocultas, muitos mapas de unidades replicadas em cada camada, agrupando as saídas de unidades próximas, formando uma rede ampla que pode lidar com vários caracteres ao mesmo tempo, mesmo se eles se sobrepõem e uma inteligente maneira de treinar um sistema completo, não apenas um reconhecedor. Mais tarde, esta arquitetura foi formalizada sob o nome de redes neurais convolucionais.

2- Redes Neurais Convolucionais

As Redes Neurais Convolucionais (ConvNets ou CNNs) são redes neurais artificiais profundas que podem ser usadas para classificar imagens, agrupá-las por similaridade (busca de fotos) e realizar reconhecimento de objetos dentro de cenas. São algoritmos que podem identificar rostos, indivíduos, sinais de rua, cenouras, ornitorrincos e muitos outros aspectos dos dados visuais.

2- Redes Neurais Convolucionais



2- Redes Neurais Convolucionais

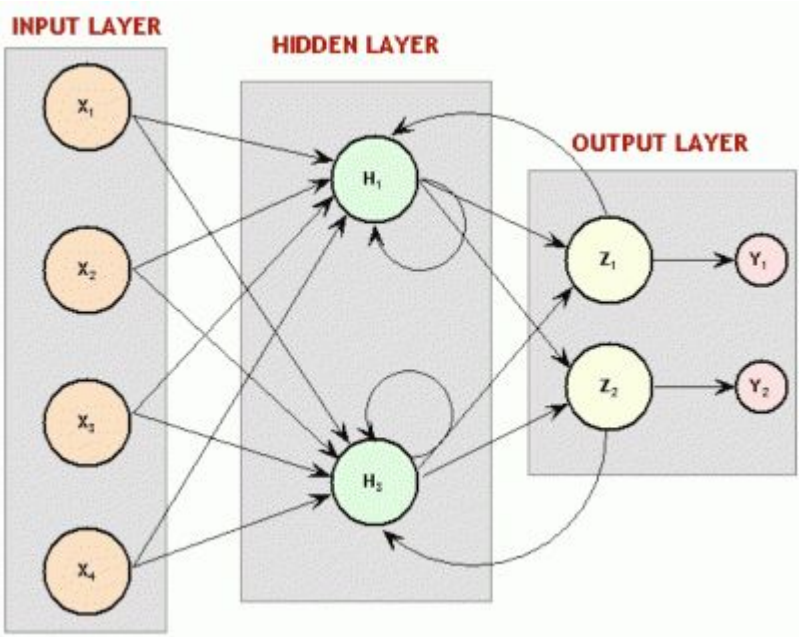
À medida que as imagens se movem através de uma rede convolucional, descrevemos em termos de volumes de entrada e saída, expressando-as matematicamente como matrizes de múltiplas dimensões dessa forma: 30x30x3. De camada em camada, suas dimensões mudam à medida que atravessam a rede neural convolucional até gerar uma série de probabilidades na camada de saída, sendo uma probabilidade para cada possível classe de saída.

3- Redes Neurais Recorrentes

As redes recorrentes são um poderoso conjunto de algoritmos de redes neurais artificiais especialmente úteis para o processamento de dados sequenciais, como som, dados de séries temporais ou linguagem natural.

As redes recorrentes são um poderoso conjunto de algoritmos de redes neurais artificiais especialmente úteis para o processamento de dados sequenciais, como som, dados de séries temporais ou linguagem natural.

3- Redes Neurais Recorrentes

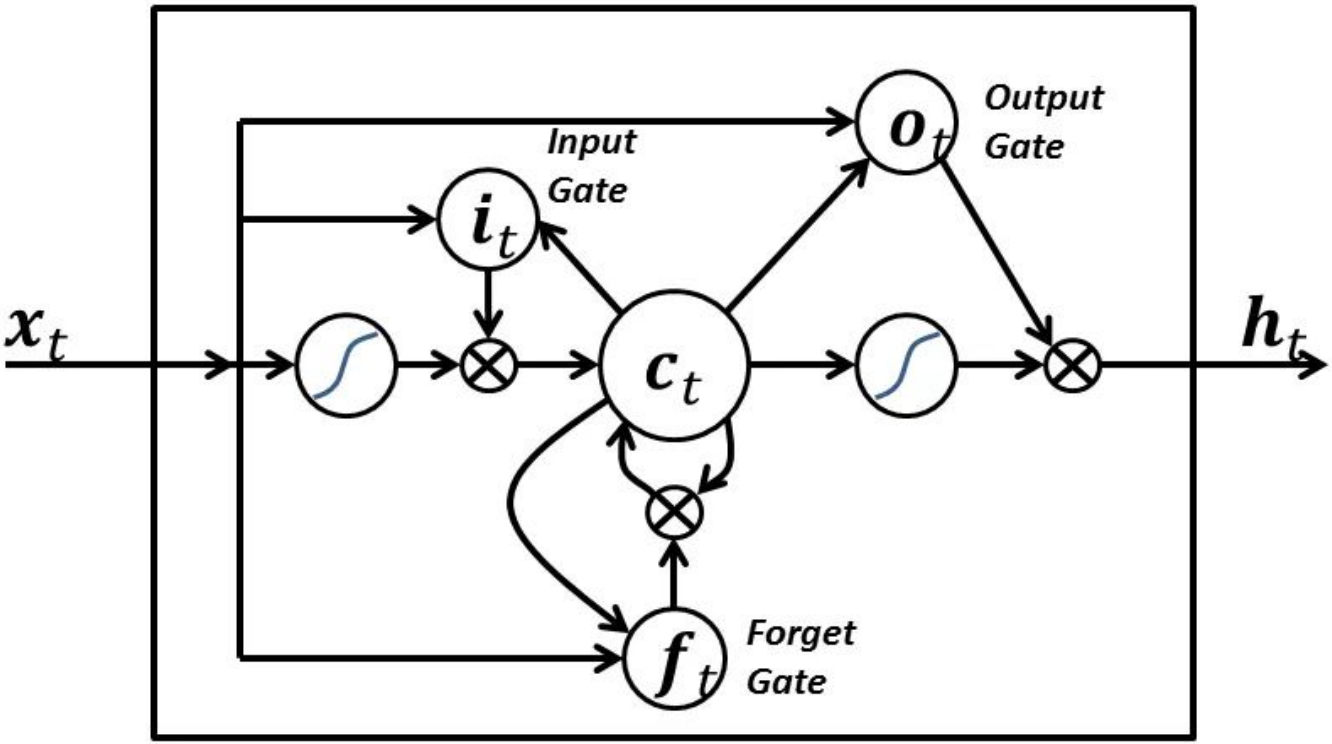


4- Long Short-Term Memory (LSTM)

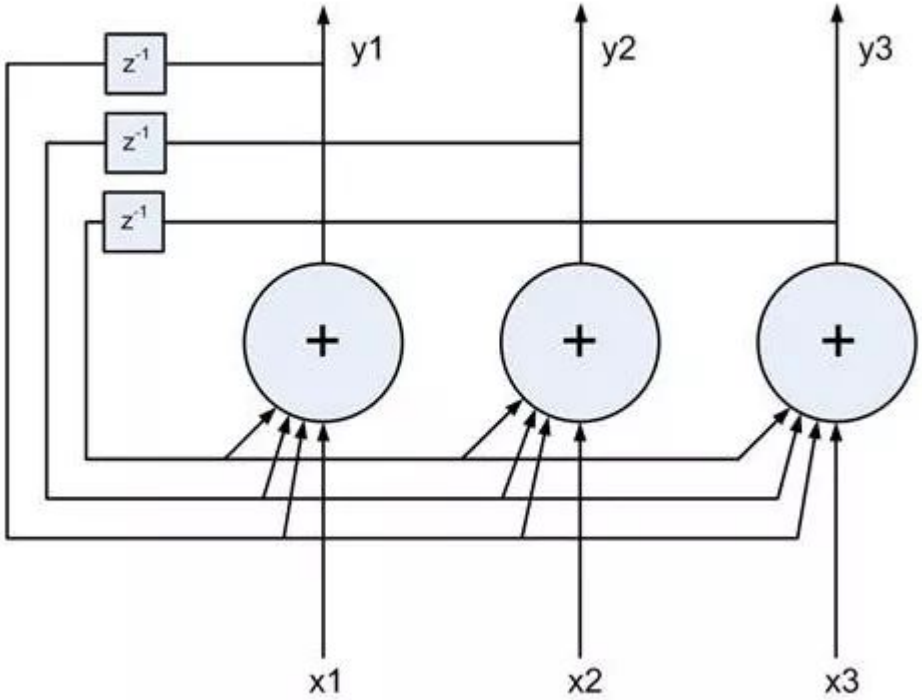
Em meados dos anos 90, a proposta dos pesquisadores alemães Sepp Hochreiter e Juergen Schmidhuber apresentou uma variação da rede recorrente com as chamadas unidades de Long Short-Term Memory, como uma solução para o problema do vanishing gradient, problema comum em redes neurais recorrentes.

Os LSTMs ajudam a preservar o erro que pode ser copiado por tempo e camadas.

4- Long Short-Term Memory (LSTM)



5- Redes de Hopfield



5- Redes de Hopfield

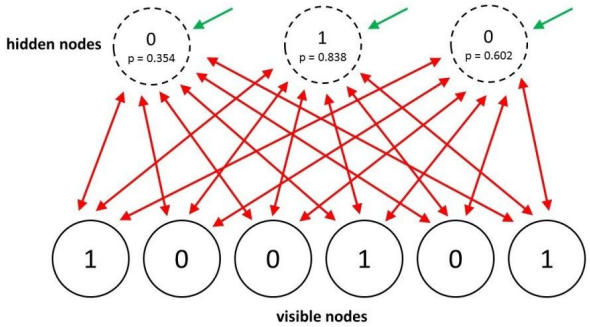
Em 1982, John Hopfield percebeu que, se as conexões são simétricas, existe uma função de energia global.

Cada “configuração” binária de toda a rede possui energia, enquanto a regra de decisão do limite binário faz com que a rede se conforme com um mínimo desta função de energia.

Uma excelente maneira de usar esse tipo de computação é usar memórias como energia mínima para a rede neural. Usar mínimos de energia para representar memórias resulta em uma memória endereçável ao conteúdo. Um item pode ser acessado por apenas conhecer parte do seu conteúdo. É robusto contra danos no hardware.

6- Máquinas de Boltzmann

Uma Máquina de Boltzmann é um tipo de rede neural recorrente estocástica. Pode ser visto como a contrapartida estocástica e generativa das Redes Hopfield. Foi uma das primeiras redes neurais capazes de aprender representações internas e é capaz de representar e resolver problemas combinatórios difíceis.



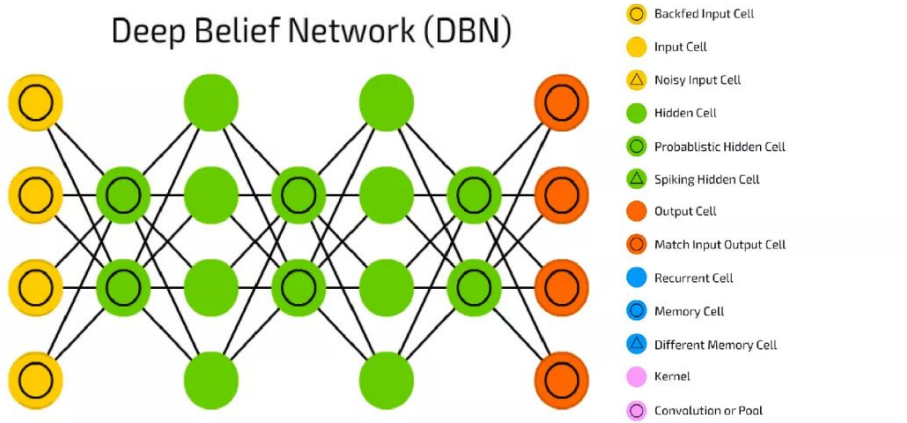
7- Deep Belief Network

Para superar as limitações do backpropagation, os pesquisadores consideraram o uso de abordagens de aprendizado sem supervisão.

Em particular, eles ajustam os pesos para maximizar a probabilidade de um modelo gerador ter gerado a entrada sensorial. A questão é que tipo de modelo generativo devemos aprender? Pode ser um modelo baseado em energia como uma Máquina de Boltzmann? Ou um modelo causal feito de neurônios? Ou um híbrido dos dois?

7- Deep Belief Network

Uma Deep Belief Network pode ser definida como uma pilha de Máquinas de Boltzmann Restritas (RBM - Restricted Boltzmann Machines), em que cada camada RBM se comunica com as camadas anterior e posterior.

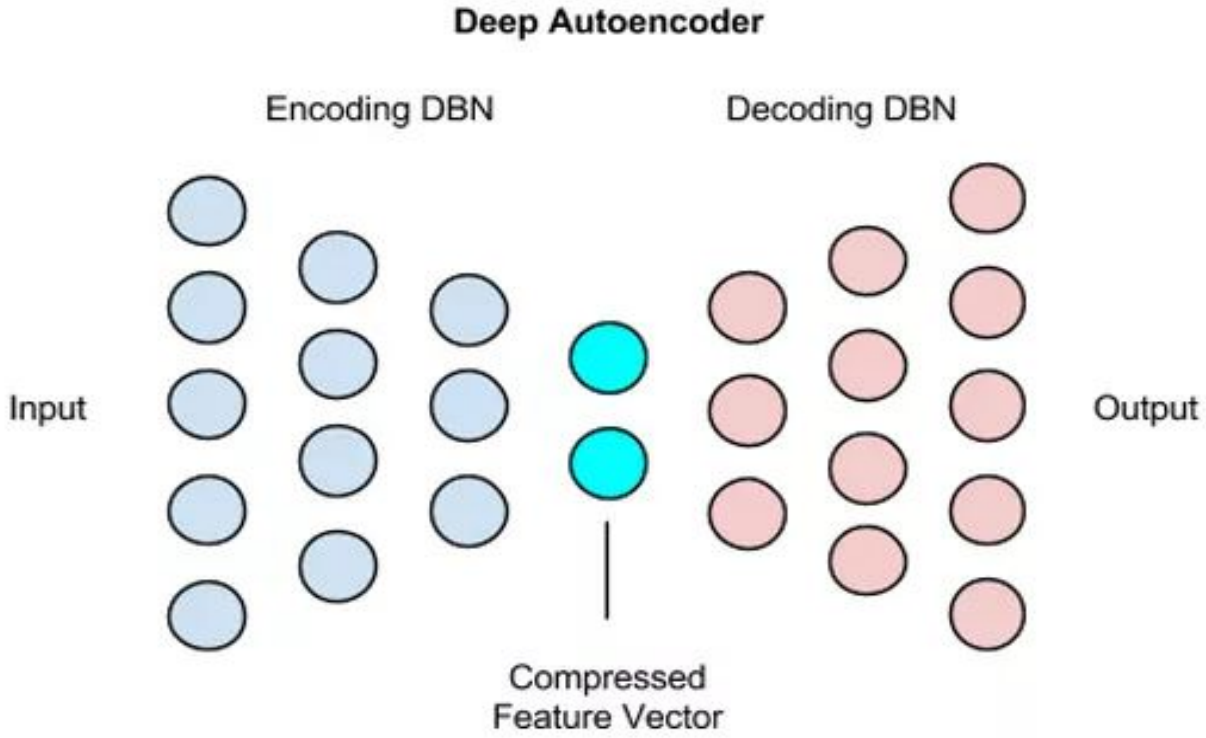


8- Deep Auto-Encoders

Um Deep Auto-Encoder é composto por duas redes simétricas Deep Belief que tipicamente têm quatro ou cinco camadas rasas que representam a metade da codificação (encoder) da rede e o segundo conjunto de quatro ou cinco camadas que compõem a metade da decodificação (decoder).

Os Deep Auto-Encoders são úteis na modelagem de tópicos ou modelagem estatística de tópicos abstratos que são distribuídos em uma coleção de documentos. Isso, por sua vez, é um passo importante em sistemas de perguntas e respostas como o IBM Watson.

8- Deep Auto-Encoders

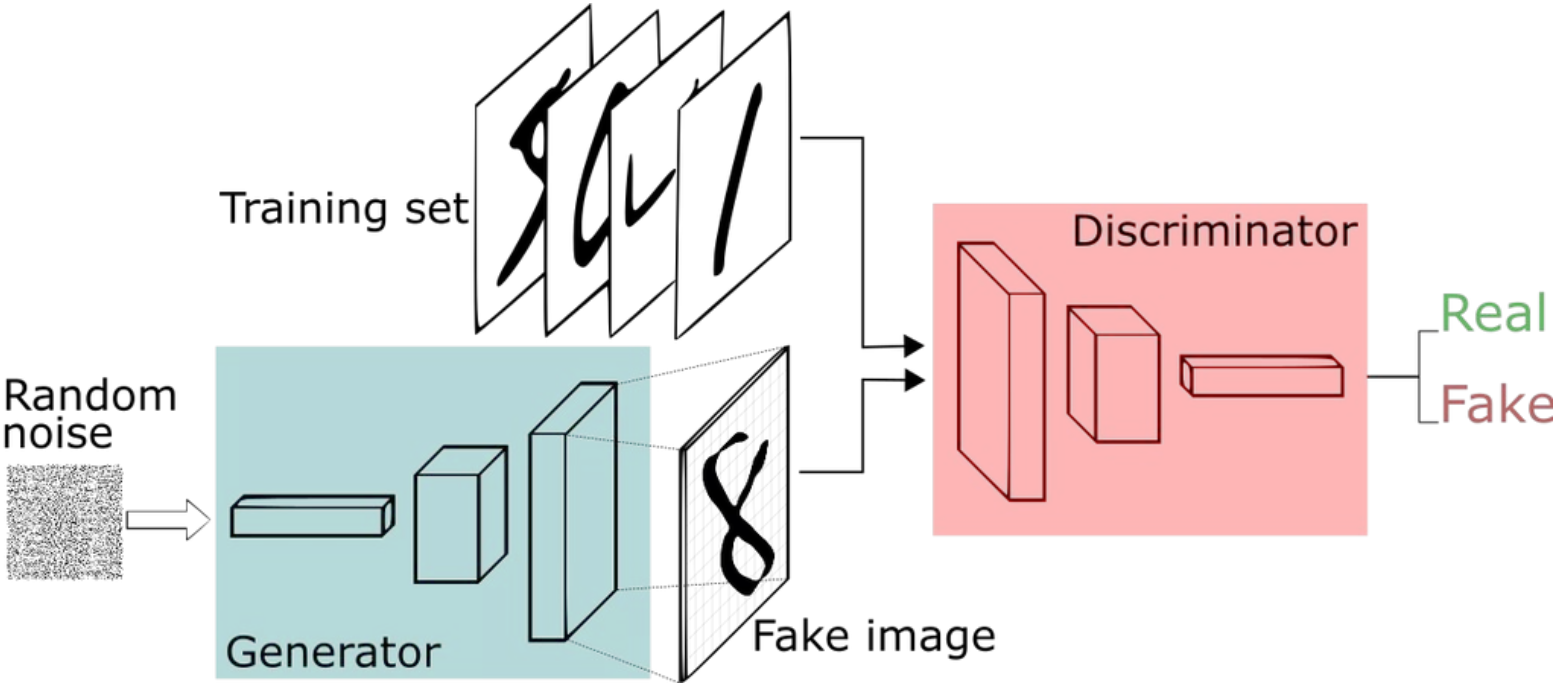


9- Generative Adversarial Network

As Generative Adversarial Networks (GANs) são arquiteturas de redes neurais profundas compostas por duas redes, colocando uma contra a outra (daí o nome, “adversária”).

O potencial de GANs é enorme, porque eles podem aprender a imitar qualquer distribuição de dados. Ou seja, os GANs podem ser ensinados a criar mundos estranhamente semelhantes aos nossos em qualquer domínio: imagens, música, fala, prosa. Eles são artistas robôs em um sentido, e sua produção é impressionante - até mesmo pungente.

9- Generative Adversarial Network



MLP -Voltemos a essa bagaça

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
```

rede1 = Network([2, 3, 1])

MLP

Observe também que os bias e pesos são armazenados como listas de matrizes Numpy. Assim, por exemplo, `rede1.weights[1]` é uma matriz Numpy armazenando os pesos conectando a segunda e terceira camadas de neurônios. (Não é a primeira e segunda camadas, uma vez que a indexação da lista em Python começa em 0.)

MLP

$$a' = \sigma(wa + b)$$

Onde, a é o vetor de ativações da segunda camada de neurônios.Para obter um **a'** multiplicamos a pela matriz de peso w, e adicionamos o vetor b com os bias (se você leu os capítulos anteriores, isso não deve ser novidade agora). Em seguida, aplicamos a função σ de forma elementar a cada entrada no vetor **wa + b**.

```
# Função de Ativação Sigmóide
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

```python
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
        training_data = list(training_data)
        n = len(training_data)

        if test_data:
            test_data = list(test_data)
            n_test = len(test_data)

        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [training_data[k:k+mini_batch_size] for k in range(0, n, mini_batch_size)]

            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)

            if test_data:
                print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test));
            else:
                print("Epoch {} finalizada".format(j))
```

MLP - Treinamento

O código funciona da seguinte forma. Em cada época, ele começa arrastando aleatoriamente os dados de treinamento e, em seguida, particiona-os em mini-lotes de tamanho apropriado. Esta é uma maneira fácil de amostragem aleatória dos dados de treinamento. Então, para cada mini_batch, aplicamos um único passo de descida do gradiente. Isso é feito pelo código self.update_mini_batch (mini_batch, eta), que atualiza os pesos e os bias da rede de acordo com uma única iteração de descida de gradiente, usando apenas os dados de treinamento em mini_batch.

MLP - Backpropagation

O backpropagation é indiscutivelmente o algoritmo mais importante na história das redes neurais

O algoritmo backpropagation foi originalmente introduzido na década de 1970, mas sua importância não foi totalmente apreciada até um famoso artigo de 1986 de David Rumelhart, Geoffrey Hinton e Ronald Williams.

MLP - Backpropagation

O algoritmo de backpropagation consiste em duas fases:

1. O passo para frente (forward pass), onde nossas entradas são passadas através da rede e as previsões de saída obtidas (essa etapa também é conhecida como fase de propagação).

2. O passo para trás (backward pass), onde calculamos o gradiente da função de perda na camada final (ou seja, camada de previsão) da rede e usamos esse gradiente para aplicar recursivamente a regra da cadeia (chain rule) para atualizar os pesos em nossa rede (etapa também conhecida como fase de atualização de pesos ou retro-propagação).

MLP - Backpropagation (Forward Pass)

O propósito do passo para frente é propagar nossas entradas (os dados de entrada) através da rede aplicando uma série de dot products (multiplicação entre os vetores) e ativações até chegarmos à camada de saída da rede (ou seja, nossas previsões).

X0	X1	Y
0	0	0
0	1	1
1	0	1
1	1	0

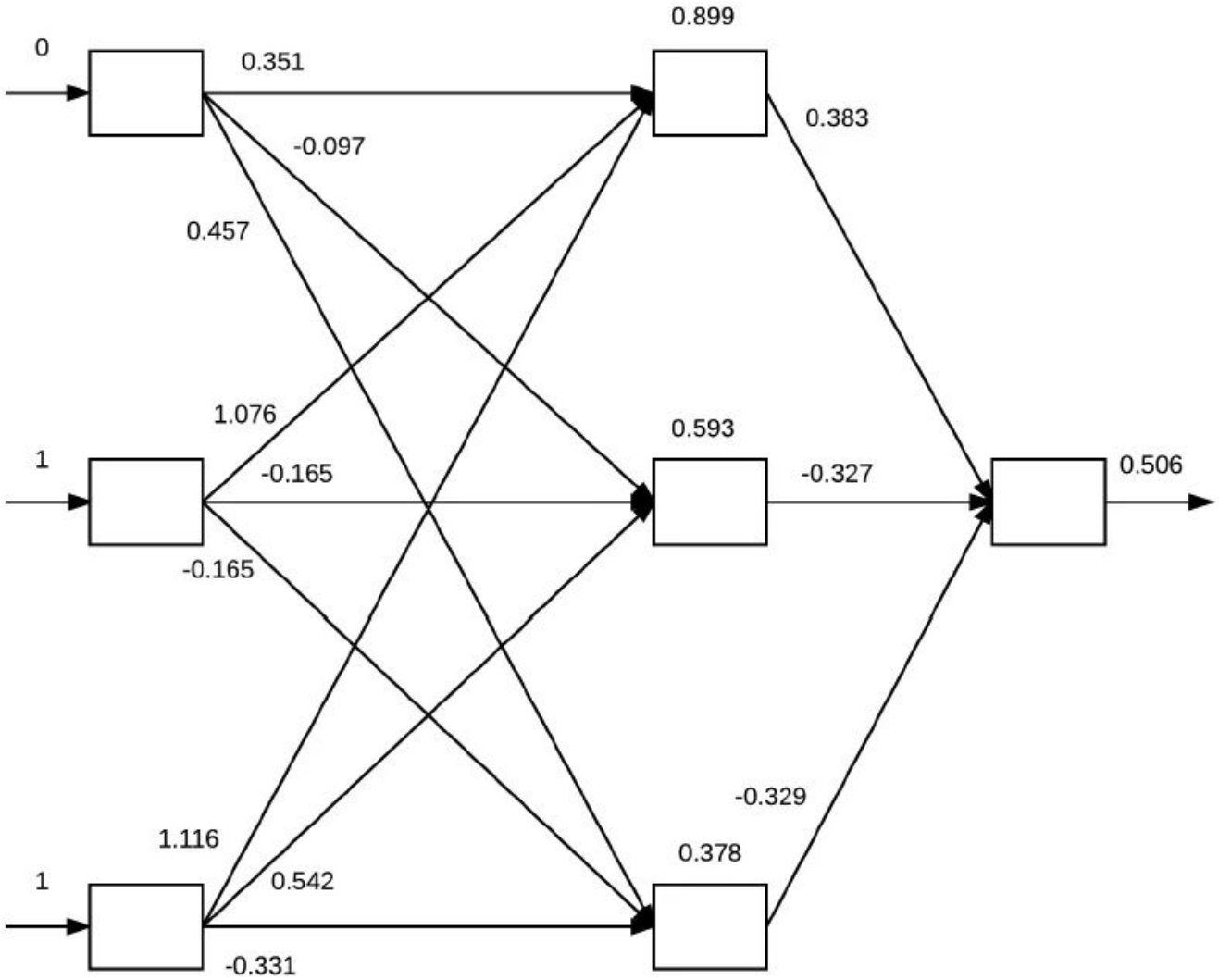
MLP - Backpropagation

Este é um bom começo, no entanto, estamos esquecendo de incluir o bias. Existem duas maneiras de incluir o bias b em nossa rede. Nós podemos:

1. Usar uma variável separada.

2. Tratar o bias como um parâmetro treinável dentro da matriz, inserindo uma coluna de 1s nos vetores de recursos.

X0	X1	X2 (bias)	Y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

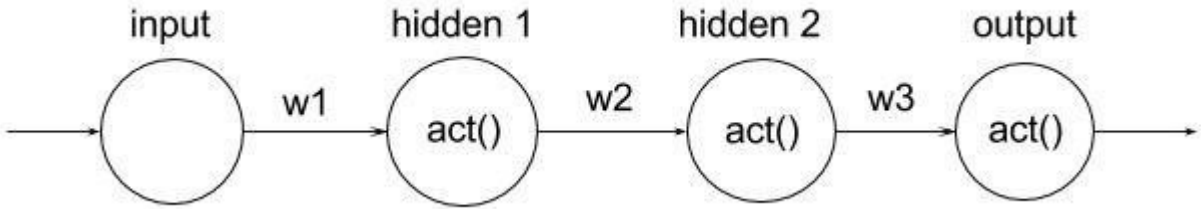

0
1
1
0.351
-0.097
0.457
1.076
-0.165
-0.165
1.116
0.542
-0.331
0.899
0.593
0.378
0.383
-0.327
-0.329
0.506

MLP - Backpropagation

$$f(saida) = \begin{cases} \mathbf{1} \text{ se saida} > 0 \\ \mathbf{0} \text{ se caso contrário} \end{cases}$$

MLP - Backpropagation

O objetivo do backpropagation é otimizar os pesos para que a rede neural possa aprender a mapear corretamente as entradas para as saídas.



MLP - Backpropagation

$$output = act(w3 * hidden2)$$
$$hidden2 = act(w2 * hidden1)$$
$$hidden1 = act(w1 * input)$$

$$output = act(w3 * act(w2 * act(w1 * input)))$$

MLP - Backpropagation

Para aplicar o algoritmo de backpropagation, nossa função de ativação deve ser diferenciável, de modo que possamos calcular a derivada parcial do erro em relação a um dado peso wi,j, loss(E), saída de nó oj e saída de rede j.

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{i,j}}$$

Aqui, a saída é uma função composta dos pesos, entradas e função (ou funções) de ativação.

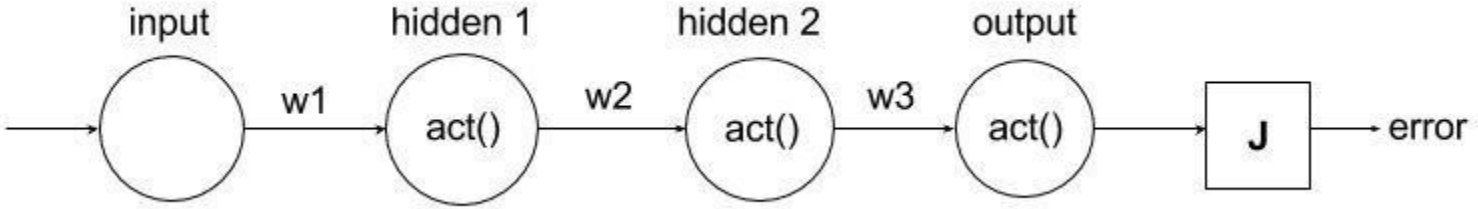
MLP - Backpropagation

Se fôssemos então tirar a derivada da função com relação a algum peso arbitrário (por exemplo, w1), aplicaríamos iterativamente a regra da cadeia

$$\frac{\partial}{\partial w1} output = \frac{\partial}{\partial hidden2} output * \frac{\partial}{\partial hidden1} hidden2 * \frac{\partial}{\partial w1} hidden1$$

MLP - Backpropagation

Agora, vamos anexar mais uma operação à cauda da nossa rede neural. Esta operação irá calcular e retornar o erro - usando a função de custo - da nossa saída:



$$\frac{\partial error}{\partial w1} = \frac{\partial error}{\partial output} * \frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1} * \frac{\partial hidden1}{\partial w1}$$

MLP - Backpropagation

Calculamos as derivadas de erro w.r.t. para todos os outros pesos na rede e aplicamos gradiente descendente da mesma maneira. Isso é backpropagation - simplesmente o cálculo de derivadas que são alimentadas para um algoritmo de otimização.

Assim, como regra geral de atualizações de peso, podemos usar a Regra Delta (Delta Rule):

Novo Peso = Peso Antigo - Derivada * Taxa de Aprendizagem

MLP - Backpropagation

A taxa de aprendizagem (learning rate) é introduzida como uma constante (geralmente muito pequena), a fim de forçar o peso a ser atualizado de forma suave e lenta (para evitar grandes passos e comportamento caótico).

MLP - Backpropagation

Existem vários métodos de atualização de peso. Esses métodos são frequentemente chamados de otimizadores. A regra delta é a mais simples e intuitiva, no entanto, possui várias desvantagens.

MLP - Backpropagation

Quantas iterações são necessárias para convergir (ou seja, alcançar uma função de perda mínima global)? Isso vai depender de diversos fatores:

- Depende de quão forte é a taxa de aprendizado que estamos aplicando. Alta taxa de aprendizado significa aprendizado mais rápido, mas com maior chance de instabilidade.
- Depende também dos hyperparâmetros da rede (quantas camadas, quão complexas são as funções não-lineares, etc..). Quanto mais variáveis, mais leva tempo para convergir, mas a precisão tende a ser maior.
- Depende do uso do método de otimização, pois algumas regras de atualização de peso são comprovadamente mais rápidas do que outras.
- Depende da inicialização aleatória da rede. Talvez com alguma sorte você inicie a rede com pesos quase ideais e esteja a apenas um passo da solução ideal. Mas o contrário também pode ocorrer.
- Depende da qualidade do conjunto de treinamento. Se a entrada e a saída não tiverem correlação entre si, a rede neural não fará mágica e não poderá aprender uma correlação aleatória.

MLP - Backpropagation

A maior parte do trabalho é feita pela linha:

```
delta_nabla_b, delta_nabla_w = self.backprop (x, y)
```

```python
def backprop(self, x, y):

    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # Feedforward
    activation = x

    # Lista para armazenar todas as ativações, camada por camada
    activations = [x]

    # Lista para armazenar todos os vetores z, camada por camada
    zs = []

    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)

    # Backward pass
    delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())

    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```

MLP - Backpropagation

Observe o método backprop. Começamos inicalizando as matrizes de pesos (nabla_w) e bias (nabla_b) com zeros. Essas matrizes serão alimentadas com valores durante o processo de treinamento. Isso é o que a rede neural artificial efetivamente aprende.

MLP - Backpropagation

Na passada para trás (Backward Pass) calculamos as derivadas e fazemos as multiplicações de matrizes mais uma vez (o funcionamento de redes neurais artificiais é baseado em um conceito elementar da Álgebra Linear, a multiplicação de matrizes). Repare que chamamos o método Transpose() para gerar a transposta da matriz e assim ajustar as dimensões antes de efetuar os cálculo. Por fim, retornamos bias e pesos.

Entendera?

Agora esqueçam

Bibliotecas para python

- Scikit-learn
- Tensorflow
- keras
- mxnet

Links úteis

http://deeplearningbook.com.br

https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/