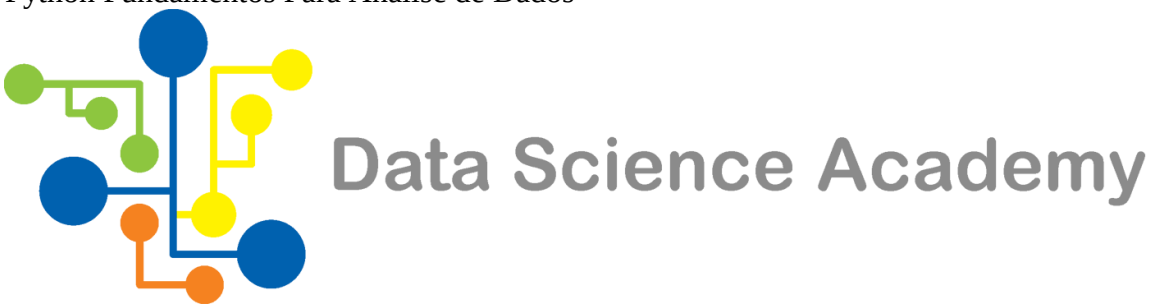
Python Fundamentos Para Análise de Dados



Backpropagation e o Treinamento de Redes Neurais

O backpropagation é indiscutivelmente o algoritmo mais importante na história das redes neurais – sem backpropagation (eficiente), seria impossível treinar redes de aprendizagem profunda da forma que vemos hoje. O backpropagation pode ser considerado a pedra angular das redes neurais modernas e aprendizagem profunda. Neste capítulo, vamos compreender como o backpropagation é usado no treinamento das redes neurais: Algoritmo Backpropagation Parte 2 – Treinamento de Redes Neurais.

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).

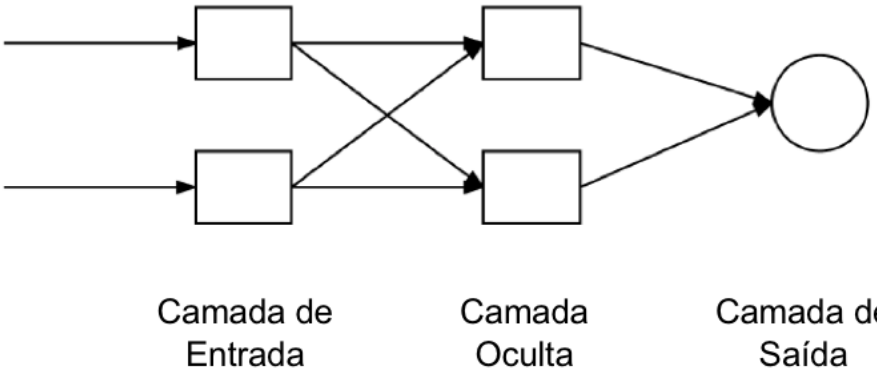
Vamos analisar cada uma dessas fases e compreender como funciona o backpropagation no treinamento nas redes neurais. No próximo capítulo, voltaremos ao script em Python para compreender como é a implementação do algoritmo. Let’s begin!

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). Para visualizar esse processo, vamos primeiro considerar a tabela abaixo. Podemos ver que cada entrada X na matriz é 2-dim (2 dimensões), onde cada ponto de dado é representado por dois números. Por exemplo, o primeiro ponto de dado é representado pelo vetor de recursos (0, 0), o segundo ponto de dado por (0, 1), etc. Em seguida, temos nossos valores de saída Y como a coluna da direita. Nossos valores de saída são os rótulos de classe. Dada uma entrada da matriz, nosso objetivo é prever corretamente o valor de saída desejado. Em resumo, X representa as entradas e Y a saída.

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

Para obter uma precisão de classificação perfeita nesse problema, precisamos de uma rede neural feedforward com pelo menos uma camada oculta. Podemos então começar com uma arquitetura de 2-2-1 conforme a imagem abaixo.



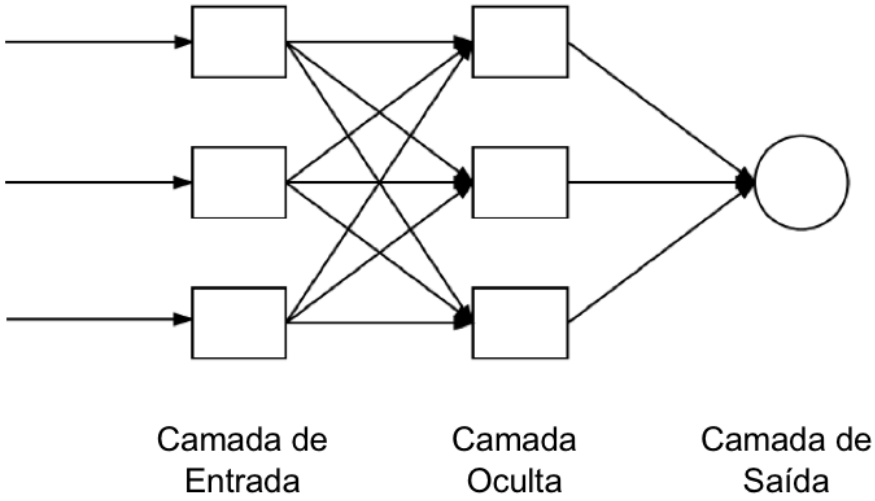
Este é um bom começo, no entanto, estamos esquecendo de incluir o bias. Existem duas maneiras de incluir o bias b em nossa rede. Nós podemos:

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.

Inserir uma coluna de 1s no nosso vetor de recursos é feito de forma programática, mas para garantir a didática, vamos atualizar nossa matriz para ver isso explicitamente, conforme tabela abaixo. Como você pode ver, uma coluna de 1s foi adicionada aos nossos vetores de recursos. Na prática você pode inserir essa coluna em qualquer lugar que desejar, mas normalmente a colocamos como a primeira entrada no vetor de recursos ou a última entrada no vetor de recursos.

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

Como nós mudamos o tamanho do nosso vetor de recursos de entrada (normalmente o que é realizado dentro da implementação da rede em si, para que não seja necessário modificar explicitamente a nossa matriz), isso muda nossa arquitetura de rede de 2-2-1 para uma arquitetura 3-3-1, conforme imagem abaixo. Ainda nos referimos a essa arquitetura de rede como 2-2-1, mas quando se trata de implementação, na verdade, é 3-3-1 devido à adição do termo de bias incorporado na matriz.



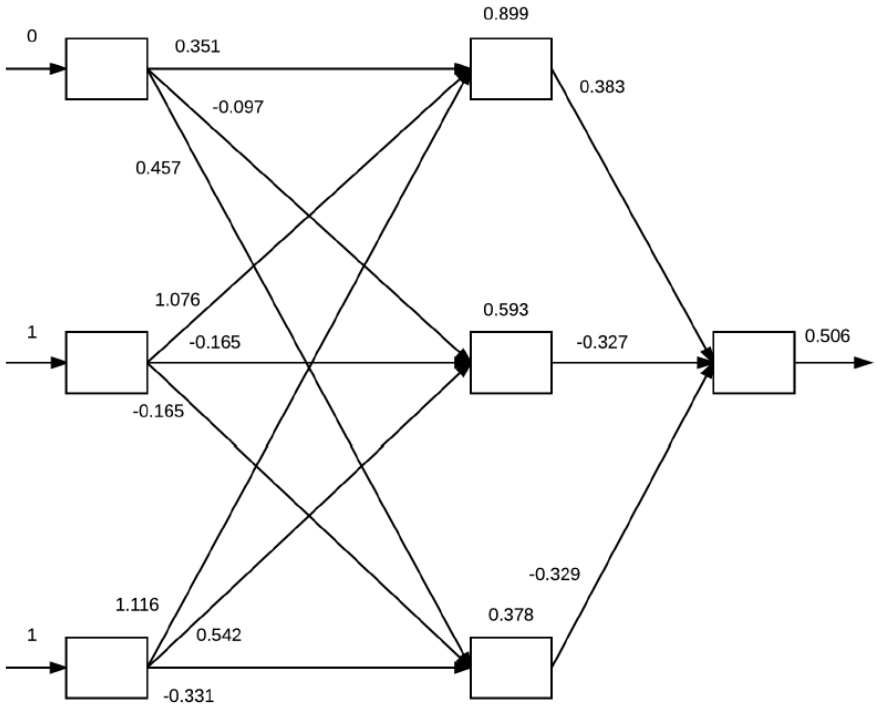
Finalmente, lembre-se de que tanto nossa camada de entrada quanto todas as camadas ocultas exigem um termo de bias. No entanto, a camada de saída final não requer um bias. O bias agora é um parâmetro treinável dentro da matriz de peso, tornando o treinamento mais eficiente e substancialmente mais fácil de implementar. Para ver o forward pass em ação, primeiro inicializamos os pesos em nossa rede, conforme figura abaixo. Observe como cada seta na matriz de peso tem um valor associado a ela – esse é o valor de peso atual para um determinado nó e significa o valor em que uma determinada entrada é amplificada ou diminuída. Este valor de peso será então atualizado durante a fase de backpropgation (lembre-se que ainda estamos no forward pass). Existem várias formas de inicializar o vetor de pesos e isso pode influenciar diretamente no treinamento da rede, como veremos mais abaixo.

Na extrema esquerda da figura abaixo, apresentamos o vetor de recursos (0, 1, 1) e também o valor de saída 1 para a rede, pois depois precisamos calcular os erros de previsão. Aqui podemos ver que 0,1 e 1 foram atribuídos aos três nós de entrada na rede. Para propagar os valores através da rede e obter a classificação final, nós precisamos do dot product entre as entradas e os valores de peso, seguido pela aplicação de um função de ativação (neste caso, a função sigmóide s). Vamos calcular as entradas para os três nós nas camadas ocultas:

1. $s\ ((0\ x\ 0.351)\ +\ (1\ x\ 1.076)\ +\ (1\ x\ 1.116))\ =\ 0.899$

2. $s\ ((0\ x\ 0.097)\ +\ (1\ x\ 0.165) + (1\ x\ 0.542))\ =\ 0.593$

3. $s\ ((0x\ 0.457)\ +\ (1\ x\ 0.165) + (1\ x\ 0.331))\ =\ 0.378$



Observando os valores dos nós das camadas ocultas (camadas do meio), podemos ver que os nós foram atualizados para refletir nossa computação. Agora temos nossas entradas para os nós da camada oculta. Para calcular a previsão de saída, uma vez mais usamos o dot product seguido por uma ativação sigmóide:

$$s\ ((0.899\ x\ 0.383)\ +\ (0.593\ x\ -\ 0.327)\ +\ (0.378\ x\ -0.329))\ =\ 0.506$$

A saída da rede é, portanto, 0.506. Podemos aplicar uma função de etapa (step function) para determinar se a saída é a classificação correta ou não:

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

Aplicando a step function com saida = 0.506, vemos que nossa rede prevê 1 que é, de fato, o rótulo de classe correto. No entanto, a nossa rede não está muito confiante neste rótulo de classe. O valor previsto 0.506 está muito próximo do limite da etapa. Idealmente, esta previsão deve ser mais próxima de 0.98 ou 0.99., implicando que a nossa rede realmente aprendeu o padrão no conjunto de dados. Para que nossa rede realmente "aprenda", precisamos aplicar o backpropagation.

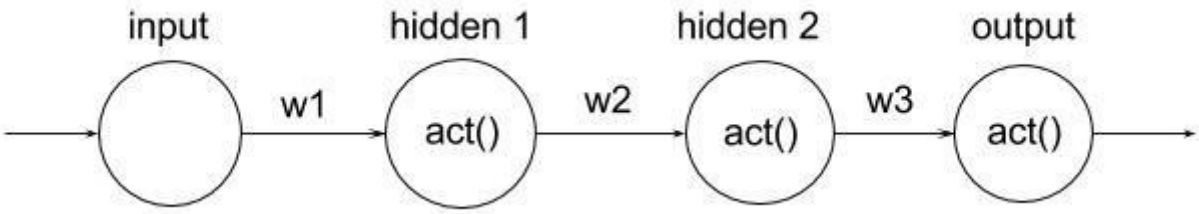
Backpropagation

Para qualquer problema de aprendizagem supervisionada, nós selecionamos pesos que fornecem a estimativa ótima de uma função que modela nossos dados de treinamento. Em outras palavras, queremos encontrar um conjunto de pesos W que minimize a saída de J(W), onde J(W) é a função de perda, ou o erro da rede. Nos capítulos anteriores, discutimos o algoritmo de gradiente descendente, em que atualizamos cada peso por alguma redução escalar negativa da derivada do erro em relação a esse peso. Se optarmos por usar gradiente descendente (ou quase qualquer outro algoritmo de otimização convexo), precisamos encontrar as derivadas na forma numérica.

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

Para outros algoritmos de aprendizado de máquina, como regressão logística ou regressão linear, o cálculo das derivadas é uma aplicação elementar de diferenciação. Isso ocorre porque as saídas desses modelos são apenas as entradas multiplicadas por alguns pesos escolhidos e, no máximo, alimentados por uma única função de ativação (a função sigmóide na regressão logística). O mesmo, no entanto, não pode ser dito para redes

neurais. Para demonstrar isso, aqui está um diagrama de uma rede neural de dupla camada:



Como você pode ver, cada neurônio é uma função do anterior conectado a ele. Em outras palavras, se alguém alterasse o valor de w1, os neurônios “hidden 1” e “hidden 2” (e, finalmente, a saída) mudariam. Devido a essa noção de dependências funcionais, podemos formular matematicamente a saída como uma função composta extensiva:

$$output = act(w3 * hidden2)$$

$$hidden2 = act(w2 * hidden1)$$

$$hidden1 = act(w1 * input)$$

ou simplesmente:

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

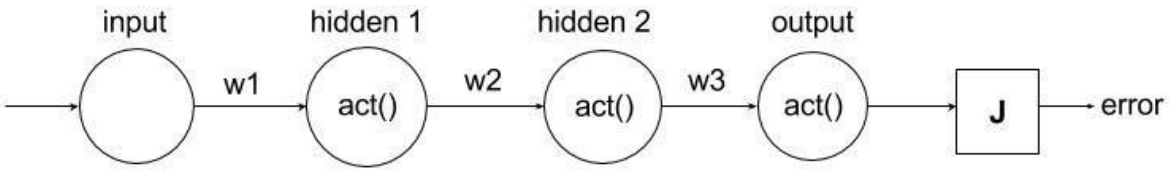
Para aplicar o algoritmo de backpropagation, nossa função de ativação deve ser diferenciável, de modo que possamos calcular a derivada parcial do erro em relação a um dado peso wi,j, loss(E), saída de nó oj e saída de rede j.

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

Aqui, a saída é uma função composta dos pesos, entradas e função (ou funções) de ativação. É importante perceber que as unidades / nós ocultos são simplesmente cálculos intermediários que, na realidade, podem ser reduzidos a cálculos da camada de entrada. Se fôssemos então tirar a derivada da função com relação a algum peso arbitrário (por exemplo, w1), aplicaríamos iterativamente a regra da cadeia. O resultado seria semelhante ao seguinte:

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

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:



Tudo o que fizemos foi adicionar outra dependência funcional; nosso erro é agora uma função da saída e, portanto, uma função da entrada, pesos e função de ativação. Se fôssemos calcular a derivada do erro com qualquer peso arbitrário (novamente, escolheríamos w1), o resultado seria:

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

Cada uma dessas derivações pode ser simplificada, uma vez que escolhemos uma função de ativação e erro, de modo que todo o resultado represente um valor numérico. Nesse ponto, qualquer abstração foi removida e a derivada de erro pode ser usada na descida do gradiente (como discutido anteriormente aqui no livro) para melhorar iterativamente o peso. 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 convexa. Chamamos isso de "retropropagação" porque estamos usando o erro de saída para atualizar os pesos, tomando passos iterativos usando a regra da cadeia até que alcancemos o valor de peso ideal.

Depois de compreender o funcionamento do algoritmo backpropagation, você percebe sua simplicidade. Claro, a aritmética/cálculos reais podem ser difíceis, mas esse processo é tratado pelos nossos computadores. Na realidade, o backpropagation é apenas uma aplicação da regra da cadeia (chain rule). Como as redes neurais são estruturas de modelo de aprendizado de máquina multicamadas complicadas, cada peso "contribui" para o erro geral de uma maneira mais complexa e, portanto, as derivadas reais exigem muito esforço para serem produzidas. No entanto, uma vez que passamos pelo cálculo, o backpropagation das redes neurais é equivalente à descida de gradiente típica para regressão logística / linear.

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

$$Novo\ Peso\ =\ Peso\ Antigo\ -\ Derivada\ *\ Taxa\ de\ Aprendizagem$$

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).

Para validar esta equação:

- Se a Derivada for positiva, isso significa que um aumento no peso aumentará o erro, portanto, o novo peso deverá ser menor.
- Se a Derivada é negativa, isso significa que um aumento no peso diminuirá o erro, portanto, precisamos aumentar os pesos.
- Se a Derivada é 0, significa que estamos em um mínimo estável. Assim, nenhuma atualização nos pesos é necessária -> chegamos a um estado estável.

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. Confira nas referências ao final do capítulo, um excelente artigo sobre otimizadores.

Como atualizamos os pesos com uma pequena etapa delta de cada vez, serão necessárias várias iterações para ocorrer o aprendizado. Na rede neural, após cada iteração, a força de descida do gradiente atualiza os pesos para um valor cada vez menor da função de perda global. A atualização de peso na rede neural é guiada pela força do gradiente descendente sobre o erro.

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.

Ou seja, treinar uma rede neural não é tarefa simples. Imagine agora treinar uma rede profunda, com várias camadas intermediárias e milhões ou mesmo bilhões de pontos de dados e você compreende o quão trabalhoso isso pode ser e quantas decisões devem ser tomadas pelo Cientista de Dados ou Engenheiro de IA. E aprender a trabalhar de forma profissional, requer tempo, dedicação e preparo e melhor ainda se isso puder ser 100% em português para acelerar seu aprendizado. Construir aplicações de IA é uma habilidade com demanda cada vez maior no mercado.

Pensando nisso, a Data Science Academy oferece um programa completo, onde esses e vários outros conceitos são estudados em detalhes e com várias aplicações práticas, usando TensorFlow. A Formação Inteligência Artificial é composta de 9 cursos, tudo 100% online e 100% em português, que aliam teoria e prática na medida certa, com aplicações reais de Inteligência Artificial. Confira o programa completo dos cursos: Formação Inteligência Artificial. Várias empresas em todo Brasil já estão treinando seus profissionais conosco! Venha fazer parte da revolução da IA.

Fonte: Deep Learning Book

http://deeplearningbook.com.br/algoritmo-backpropagation-parte-2-treinamento-de-redes-neurais/