Implementação de Redes MLP em Haskell com Gradiente Automático

Patrick Oliveira

December 2020

1 Descrição Formal de uma Rede MLP

Uma MLP (*Multilayer Perceptron*) é uma arquitetura de rede neural de aprendizado profundo do tipo feedforward, construída como uma sequência de aplicações compostas parametrizadas por um conjunto de pesos ajustáveis.

Definindo de maneira geral, suponha que a rede possua L camadas, onde as camadas 1 e L são, respectivamente, as camadas de entrada e saída. Suponha que a camada l, para $l = 1, 2, \ldots, L$ contém n_l neurônios - portanto n_1 é a dimensão dos dados de entrada. Definimos $\mathbf{W}^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ como a matriz de pesos da camada l, e $\mathbf{b}^{[l]} \in \mathbb{R}^{n_l}$ é o vetor de viéses da camada l. De modo geral, a rede é um mapeamento de $\mathbb{R}^{n_1}$ em $\mathbb{R}^{n_L}$. Dado um input $\mathbf{x} \in \mathbb{R}^{n_1}$, o mapeamento da rede pode ser descrito por

$$\mathbf{y}^{[1]} = \mathbf{x} \in \mathbb{R}^{n_1} \tag{1}$$

$$\mathbf{y}^{[l]} = \sigma(\mathbf{W}^{[l]} y^{[l-1]} + \mathbf{b}^{[l]}) \in \mathbb{R}^{n_l} \tag{2}$$

Caso se tenha um conjunto de dados de N amostras em $\mathbb{R}^{n_1}$, $\{\mathbf{x}^{\{i\}}\}_{i=1}^{N}$, com valores objetivo $\{\overline{\mathbf{y}}(x^{\{i\}})\}_{i=1}^{N}$ em $\mathbb{R}^{n_L}$, pode-se definir uma função de custo parametrizada pelos pesos adaptativos da rede, digamos $\mathbb{L}(\mathbf{W})$, e assim, o problema de aprendizado dos melhores parâmetros para a rede é identificado com o problema de minimizar o custo $\mathbb{L}(\mathbf{W})$ em relação a $\mathbf{W}$. Porém, devido à composição de funções não-lineares ao longo das camadas da rede, a função de custo pode se tornar não-convexa, perdendo a garantia de ótimos globais ou soluções analíticas, e para o treinamento de RNAs passa-se a utilizar métodos de otimização iterativos baseados no operador gradiente, que é capaz de alcançar mínimos locais – ainda assim, muitas vezes suficientes para um dado problema.

Um dos métodos de ajuste dos pesos da rede é denominado *Gradiente Descendente*, em que se aplica iterativamente a correção dos pesos na direção do gradiente da função de custo, i.e.

$$\mathbf{W}_{i+1} \to \mathbf{W}_i - \eta \nabla \mathbb{L}(\mathbf{W}_i) \tag{3}$$

onde $\eta \in \mathbb{R}$ é a *taxa de aprendizado* e ∇ é o operador gradiente. Quando se tem um número de pesos ajustáveis e de amostras muito grande, o cálculo do gradiente se torna demasiadamente custoso e uma alternativa passa a ser o método de *Gradiente Descendente Estocástico*, em

que o ajuste pode ser feito em N etapas, onde em cada uma delas uma amostra é selecionada aleatoriamente, sem reposição, para o cálculo do custo e ajuste dos pesos.

O cálculo do gradiente pode encontrar dificuldades quando a função de custo ou as funções de ativação das camadas intermediárias são difíceis de se derivar. Uma solução é utilizar um algoritmo de *derivação automática*, que calcula o gradiente de qualquer função em um ponto a partir da decomposição da função em operações elementares compostas e a aplicação da regra da cadeia.

2 Implementação em Haskell

A execução do programa pode ser feito via *stack run*, que executará as implementações de diferentes estruturas em diferentes datasets, apresentando o valor do custo antes e depois do treinamento em cada caso. Um dos exemplos utiliza o dataset "MNIST", o que pode levar um tempo para executar. Uma alternativa é abrir o *ghci* e executar cada exemplo separadamente - o ghci carrega todos os módulos e as funções tem os mesmos nomes dos arquivos. Um exemplo de execução pode ser encontrado no youtube [1].

A implementação da arquitetura da MLP em Haskell levou em consideração a possibilidade de aplicação do algoritmo de gradiente automático, disponibilizado pelo pacote AD, no ajustamento dos pesos para o aprendizado por gradiente descendente. Brevemente, a função *grad* do pacote possibilita o cálculo do gradiente de qualquer função com respeito a uma variável que seja uma instância de *Traversable*. Assim, pode-se definir uma estrutura de rede da classe Traversable, passada como argumento para uma função de custo cujo gradiente será calculado pela função *grad*. A função retorna um valor do mesmo tipo do argumento da função de custo - no caso, retorna uma rede inteira com os pesos para ajuste. A correção é feita subtraindo da rede inicial a rede devolvida por *grad*, segundo a Eq. 3.

```
1  newtype Network a = Network [Component a]
2              deriving (Functor, Foldable, Traversable)
3
4  data Component a = Component { layers :: [Weights a]
5                               , activationFunction :: ActivationFunction
6                               , propagate :: PropagationFunction}
7                               deriving (Functor, Foldable, Traversable)
8  data PropagationFunction = Linear | Pass
9  data ActivationFunction = Sigmoid | ReLU | Tanh | Id
10 data Loss = SE | BCE
11 forwardApply :: ((InputZ a, InputY a) -> (OutputZ a, OutputY a)) ->
12                 (InputZ a, InputY a) -> (OutputZ a, OutputY a)
13 triggerComponent :: (Ord a, Floating a) => Component a ->
14                     ((InputZ a, InputY a) -> (OutputZ a, OutputY a))
15 _Linear :: (Floating a) => (Weights a, Weights a) ->
16                 ActivationFunction -> Component a
17 costGrad :: (Ord a, Floating a) => Network a -> (Input a, Target a)
18             -> Loss -> Network a
```

Listing 1: Principais definições de estruturas e funções.

[1] https://youtu.be/HFXyeMn7dic

Buscou-se implementar as estruturas forma modular, isto é, o usuário pode construir as camadas independentemente, escolhendo suas dimensões e funções de ativação, compondo-as em uma estrutura final. Pode, ainda, escolher uma função de custo qualquer. No fim, o algoritmo dá conta de calcular a estimativa da rede e de corrigi-la, dada uma amostra ou um dataset. O uso final da rede torna-se, portanto, bastante simples e adaptável.

A rede segue a definição teórica, entendendo a rede como um tipo **Network** que envolve uma lista de tipos **Component** que, por sua vez, são constituídos por uma lista de pesos, uma função de ativação e uma função de propagação. A definição genérica tem por objetivo possibilitar a construção de redes distintas de uma MLP utilizando esta mesma infraestrutura. No caso da MLP, as componentes são camadas ditas *lineares*, construídas pela função **_Linear** que cria uma componente com uma matriz **W** de pesos e um vetor **b** de vieses e uma função de ativação dentre aquelas contempladas no tipo de dado **ActivateFunction**. Para cada tipo de componente pode-se definir uma função de propagação, listando-a no tipo **PropagationFunction** - neste caso, tem-se a função de propagação do tipo "Linear" que faz a aplicação tal como definida anteriormente ("Pass" é a identidade). Dessa forma, torna-se fácil expandir o projeto a fim de acrescentar outros tipos de camadas, bastando implementar novas funções de ativação, propagação e construtores de componentes. A correção é feita por gradiente automático, de modo que é não necessário implementar um algoritmo de backpropagation, nem se limitar, por este motivo, ao uso de funções de custo de fácil derivação. Torna-se mais fácil, portanto, experimentar com estruturas de redes.

Para a propagação de uma entrada em uma camada, é aplicada a função **triggerComponent** que recebe uma componente e aplica parcialmente a função de propagação implementada (selecionada pelo tipo PropagationFunction) com os parâmetros especificados, quais sejam: os pesos e a função de ativação, retornando uma função f que é aplicada em uma entrada **x** através da função **forwardApply**. A propagação de uma entrada na rede consiste em um **scanl** na lista de componentes, utilizando a composição das funções "triggerComponent" e "forwardApply". O ajuste da rede é realizado com o gradiente retornado pela função **costGrad** que recebe uma rede, uma amostra e um indicador de função de custo (o tipo **Loss**), calculando o gradiente com relação à própria rede. Tanto a rede quanto suas componentes instanciam Functor, Applicative e Traversable (instância derivada pelo próprio Haskell) a fim de que sejam utilizáveis pelo pacote AD, e instanciam a classe Num, em particular as funções de soma e subtração, a fim de facilitar o ajuste por gradiente descendente.

Para a estruturação da rede utilizou-se o pacote Matrix, de modo que inputs, outputs e pesos são tipos "Matrix a", possibilitando o uso das operações usuais de matrizes oferecidas pelo pacote. Não foi implementado nas funções mecanismos (como o uso do Maybe) para prevenir erros decorrentes da não-combinação das dimensões das matrizes, durante as operações. Supondo, porém, que o usuário definiu corretamente a estrutura da rede considerando a dimensão das amostras, não ocorrem erros nesse sentido.

A possibilidade de derivação automática das instâncias de Applicative e Traversable mostrou-se bastante útil uma vez que não ficou evidente de que forma as operações dessas classes funcionariam nas estruturas definidas. Porém, isso, aliado ao parco entendimento da estrutura interna do pacote AD, dificultou melhorias no sistema a fim de torná-lo mais rápido - como resultado, o treinamento da rede pode ser bastante lento. Uma solução seria utilizar o gradiente automático apenas na última componente da rede, corrigindo o restante da rede por backpropagation e fazendo uso de paralelismo nas operações matriciais.