Trabalho prático 2

Regressão Logística e Redes Neurais

Julianny Favinha Donda
156059
julianny.favinha@gmail.com

José Henrique Ferreira Pinto
155976
joseh.fp@gmail.com

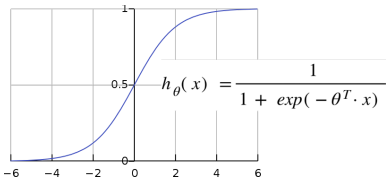
Introdução

Para fixarmos o conhecimento sobre algoritmos de classificação, vamos explorar regressões logísticas e redes neurais. O objetivo deste trabalho é discutir e classificar as imagens da base de dados Fashion-MNIST[1], distribuídas em 10 classes distintas. A base possui um total de 70000 imagens, separadas em 50000 para treino, 10000 para validação e 10000 para teste.

Regressão Logística *One-vs-All*

A regressão logística encontra uma fronteira de decisão entre os dados, indicando se determinada imagem pertence ou não a uma classe, ou seja, a regressão é um classificador binário. Para conseguir classificar mais de uma classe, um método conhecido é o *One-vs-all, que* propõe o treinamento de um classificador por classe.

Função sigmóide



$$h_\theta(x) = \frac{1}{1 + exp(-\theta^T \cdot x)} \quad (1)$$

onde θ^T são os parâmetros e x são as *features*.

Descida do gradiente

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x_j^{(i)}$$
$$\left(simultaneously\ update\ \theta_j\ for\ j = 0,\ 1,\ \ldots,\ n \right) \quad (2)$$

onde α é o *learning rate*, m é a quantidade de dados e $y^{(i)}$ são os *targets*.

Função de custo

$$J(\theta) = -\frac{1}{m} [y^{(i)} \log(h(x^{(i)})) + (1 - y^{(i)}) \log(1 - h(x^{(i)}))] \quad (3)$$

1. Soluções propostas

Inicialmente, implementaremos as funções básicas descritas anteriormente, adequando conforme necessário para efeito de exploração. Utilizaremos para essa implementação a função sigmóide descrita pela equação (1). A observação dos erros pela função (3) nos guiará para encontrar parâmetros (*learning rate* e número de iterações) melhores. Esta função penaliza fortemente um modelo que prediz que uma observação pertence a uma classe sendo que ela pertence a outra.

2. Discussão e experimentos

O algoritmo foi inicializado com valores de coeficientes iguais a 1. Mesmo começando com um valor pequeno para o *learning rate* (0,00001), a soma da descida do gradiente tomava valores infinitos, o que fazia com que o gradiente divergisse. Isso indicou que o código poderia apresentar falhas. Um dos maiores problemas foi o intervalo de valores de entrada, representando a intensidade da cor dos pixels, que estava entre $[0,\ 255]$. Esses valores, aplicados à exponencial da função sigmóide, geravam somas infinitas. Percebido isso, normalizamos para o intervalo $[0,\ 1]$. Isso permitiu fazer experimentos com valores melhores, e principalmente aumentar o *learning rate* para acelerar a convergência. Os valores bases usados foram learning rate 0,01 e 100 iterações.

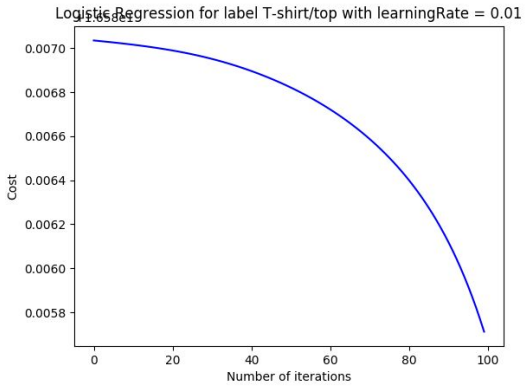


Figura 1 - Gráfico do custo pelo número de iterações para o método de regressão logística one-vs-all, usando o rótulo "T-shirt/top", *learning rate* igual a 0,01 e 100 iterações.

Nota-se que são necessárias mais iterações para que o custo diminua mais e tenha uma variação menor entre duas iterações. Dessa forma, o *learning rate* foi mantido e o número de iterações aumentado para 1000.

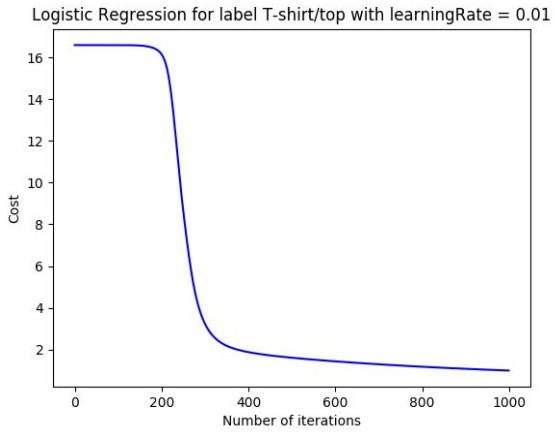


Figura 2 - Gráfico do custo pelo número de iterações para o método de regressão logística one-vs-all, usando o rótulo "T-shirt/top", *learning rate* igual a 0,01 e 1000 iterações.

Nota-se que, para a classe T-shirt/top, o custo diminuiu drasticamente entre 200 e 400 iterações e tem pouca variação para iterações maiores. Algo similar aconteceu com os outros rótulos.

A métrica escolhida para comparar os resultados das diferentes abordagens foi a *accuracy score* da biblioteca Scikit-Learn[4].

Para cada modelo obtido, obtivemos resultados de *accuracy score* para o conjunto de validação entre 79,0% e 91,7%.

```
python3 TrainLogisticRegression.py
Using learning rate = 0.01 and 1000 iterations
Performing fit for label T-shirt/top...
Performing predictions for label T-shirt/top...
[[8383  594]
 [ 545  478]]
Accuracy score = 88.6%
Elapsed time: 93.864468 s
```

Figura 3 - Trecho da execução do programa TrainLogisticRegression.py, evidenciando os resultados obtidos para a classe "T-shirt/top". As linhas da matriz de confusão representam a classificação real ("Not t-shirt/top", "T-shirt/top") e as colunas da representam a classificação do regressor logístico ("Not t-shirt/top", "T-shirt/top").

Com todos os modelos treinados, a predição para o conjunto de validação foi feita e atribuído àquela imagem a classe onde o maior valor de predição foi obtido. A acurácia obtida foi de 63,8%.

Regressão Logística Multinomial

Esta regressão, diferentemente da *One-vs-all*, que trata as classes entre pares, suporta diretamente todas as classes do conjunto de dados, sem a necessidade de treinar um modelo para cada classe.[2]

Função *Score* e função *Softmax*

$$s_k(x) = (\theta^{(k)})^T \cdot x \quad (5)$$

$$p_k = \frac{exp(s_k(x))}{\sum_{j=1}^{k} exp(s_j(x))} \quad (6)$$

onde k é a classe.

Função *Cross Entropy*

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \cdot \log\left(p_k^{(i)}\right) \quad (7)$$

1. Soluções propostas

Similar ao método anterior, vamos utilizar funções de decisão e erro para implementar a regressão. No caso da regressão multinomial utilizamos as funções de *softmax* e *cross entropy* dadas no livro texto[2].

2. Discussão e experimentos

O algoritmo foi inicializado com valores de coeficientes iguais a 1. Tivemos problemas em entender o conceito de multinomial, perdendo muito tempo ao tentar reaproveitar a regressão logística anterior, e utilizar os treinos para fornecer os coeficientes para a função (6). Percebemos nosso erro e implementamos as funções (5) e (6) separadamente. Assim, nossa descida de gradiente levava em consideração todas as classes ao mesmo tempo. Outro problema encontrado foi a falta de parametrização do resultado. Utilizamos *one-hot encoding*[3] no vetor de resultados (*target*), para mapear para um formato entendível pelas funções.

Observamos o gráfico de custo por iterações e este continuava a nos mostrar que a regressão estava divergindo. Tentamos primeiramente diminuir o *learning rate* imaginando que esse era o possível problema. Sem sucesso, avaliamos o código e encontramos incoerências na função de *one-hot encoding*: o erro foi formatar com o shape errado e isso afetou gravemente os resultados. Percebemos também problemas na função de custo. Com as correções, conseguimos ver o custo diminuir como esperávamos, e usamos isso para testar com outros *learning rates* e quantidades de iteração. Começando com um *learning rate* igual a 0,01, percebemos que o custo diminuiu gradualmente, obtendo resultado de acurácia 77,6%. Resolvemos aumentar o valor do *learning rate* para 0,1 e obtivemos o custo a seguir.

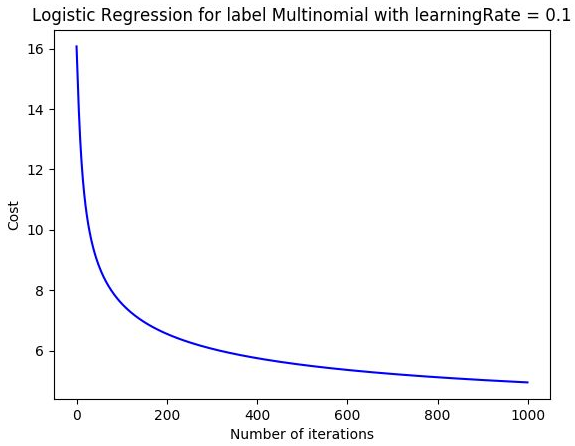


Figura 4 - Gráfico do custo pelo número de iterações para o método de Regressão Logística Multinomial, usando *learning rate* igual a 0,1.

Ao executar o modelo para o conjunto de validação, obtivemos acurácia de 83,2%. Nota-se que chegamos a um resultado melhor em relação à regressão *one-vs-all*.

Redes neurais

As redes neurais artificiais baseiam-se na ideia de funcionamento dos neurônios biológicos, onde uma ativação de um neurônio influencia a ativação do próximo. Na verdade, o método consiste em treinar uma rede de funções de forma que ao final, tenhamos um classificador, capaz de decidir a saída desejada.

Funções de ativação

Além da sigmóide(1), usamos a função ReLU.

$$ReLU(x) = max(0, x) \quad (9)$$

1. Soluções propostas

Para este trabalho, desejamos treinar uma rede neural básica, com apenas uma camada escondida.

Para a primeira rede neural, propomos usar 785 neurônios na camada de entrada, 784 pixels e um *bias*, 256 neurônios na camada escondida e 10 neurônios na camada de saída, indicando o número de classes do conjunto de dados. O número de unidades na camada escondida foi escolhido a esmo, para dar início aos experimentos, se necessário será alterado. Para a segunda rede neural, propomos usar 785 neurônios na camada de entrada, 256 nas duas camadas escondidas e 10 neurônios na camada de saída.

2. Discussão e experimentos

O algoritmo foi inicializado com valores aleatórios de coeficientes. Apesar de entender o funcionamento, escrever a base do programa foi mais complicado do que o esperado. Começamos com uma rede pequena, com 3 neurônios na primeira camada, 3 na segunda e 2 na saída. Escrevemos as funções necessárias e melhoramos nosso algoritmo. Após alguns testes, alteramos o código para aceitar as imagens do conjunto de dados para dar início aos treinamentos reais.

Todos os neurônios utilizando como ativação uma função sigmóide (1). Na camada de saída, temos as 10 classes mapeadas, onde aplicamos a função softmax (6). Nosso input foi carregado diretamente da entrada, o que gerou problemas com a sigmóide. Os valores entre $[0, 255]$, aplicados ao expoente, e à divisão, causaram problemas de arredondamento no python. Para resolver isso, normalizamos o input, deixando os valores entre $[0, 1]$.

Observando o erro por iteração, vimos que este não estava diminuindo. Recorremos aos vídeos[5] indicados em aula para buscar soluções e percebemos que erramos vários conceitos na implementação. Com base neles, reescrevemos nosso código.

Como aprendemos no vídeo, utilizamos um treinamento *batch*, onde a entrada é dada por uma matriz com *pixels* de todas as imagens, uma em cada linha. Iniciamos com uma imagem e uma iteração, para garantir que o código funcionava com essas dimensões de matriz. Aumentamos então, gradativamente, o número de imagens do batch e de iterações.

Percebemos então uma situação no treinamento, a rede treinada dava o mesmo resultado para todos os inputs. Isso ocorria em duas situações distintas:

- Quando o input continha muitas imagens (mais de 500), já em pouquíssimas iterações, entre 5 e 50.
- Quando o input era menor (500 imagens) eram necessárias mais iterações, entre 500 e 1000, mas o problema ainda ocorria.

Para o input muito grande, rastreamos a falha até as multiplicações matriciais, que com tantos valores, gerou problemas de arredondamento, principalmente ao somar 50000 valores do conjunto de treinamento.

Para os inputs menores, tentamos diminuir o *learning rate* para ganharmos mais iterações, e extrair previsões melhores desses dados. E por um tempo isso se mostrou promissor.

Como mapeamos a maioria dos nossos problemas à exponencial da função sigmóide, decidimos trocar a função de ativação das camadas, alterando o código para utilizar a função ReLU. No entanto, o problema persistiu. O

melhor resultado que obtivemos foi usando a sigmóide e uma camada escondida. Para uma execução, obtivemos acurácia de 19,3%. Como os pesos são inicializados aleatoriamente, a acurácia pode variar.

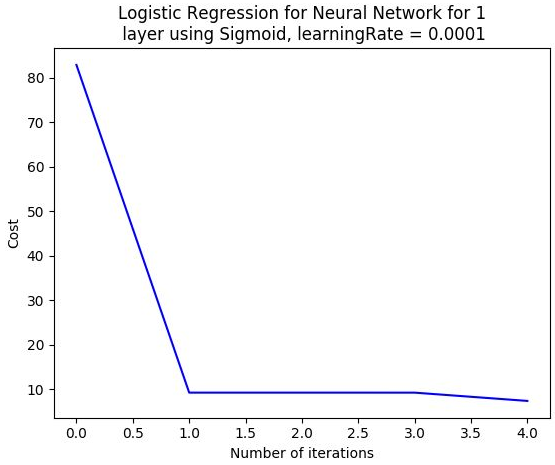


Figura 5 - Gráfico do custo pelo número de iterações para o método de Redes Neurais, usando *learning rate* igual a 0,1, uma camada escondida e sigmóide como função de ativação.

Apesar do gráfico anterior nos mostrar que o custo reduziu bastante ao longo das iterações, o resultado não foi satisfatório p.

Conclusões e trabalho futuro

Conseguimos implementar as funções de regressão logística, encontrando resultados compatíveis com o esperado de um código simples como o que implementamos. O modelo que conseguiu uma acurácia melhor foi o Multinomial. Para os dados de teste, obtivemos 79,7% de acurácia.

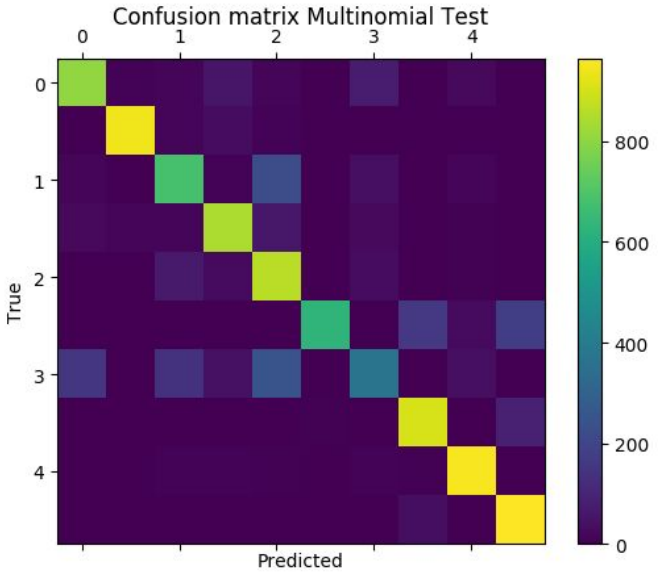


Figura 6 - Matriz de confusão da regressão logística Multinomial para os dados de teste.

Por outro lado tivemos graves problemas com a implementação da rede neural. Porém isso nos fez estudar nuances da implementação e do método que não teríamos tido chance se tivéssemos usado um código pronto. Não encontramos respostas para o comportamento errôneo de nosso código nem com o auxílio direto do monitor Alceu Bissoto. Uma das hipóteses levantadas propunha a implementação de regularização nas funções de erro, para evitar overfit, o que poderia ter melhorado nosso resultado e talvez evitado a saturação dos pesos.

Trabalho Futuro

Na regressão logística *One vs all*, percebemos que o modelo classificou melhor o caso de uma imagem não pertencer à determinada classe. É necessário investigar esse resultado, e alterar o código caso necessário.

Como tivemos muitos problemas com a implementação básica e não tivemos tempo para continuar a implementação, precisamos primeiramente detectar os erros do nosso código, e implementar funções de análises de métricas mais avançadas e *cross validation*.

Referências

[1] **Fashion-MNIST dataset.** Disponível em: <https://github.com/zalandoresearch/fashion-mnist>. Acesso em 20 set 2018.

[2] **Hands-on Machine Learning with Scikit-Learn and TensorFlow - Training Models.** Disponível em: <https://www.oreilly.com/library/view/hands-on-machine-learning/9781491962282/ch04.html> Acesso em 29 set 2018.

[3] **Using Categorical Data with One Hot Encoding.** Disponível em <https://www.kaggle.com/ayushkaul/using-categorical-data-with-one-hot-encoding> Acesso em 1 out 2018.

[4] **Scikit-Learn Metrics.** <http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics> Acesso em 3 out 2018.

[5] **Neural Networks Demystified.** <https://www.youtube.com/playlist?list=PLiaHhY2iBX9hdHaRr6b7XevZtgZRa1PoU> Acesso em 29 set 2018.