Entrega 1: Word2Vec

Motivação

O objetivo[1] deste exercício-programa é aumentar a compreensão e a familiarização do mecanismo word2vec na versão *skipgram* (*salta-grama*).

Vamos relembrar rapidamente o algoritmo word2vec. O principal insight por trás do word2vec é que "diga me com quem a palavra anda e eu te direi quem a palavra é". O word2vec usa um truque: treinar uma rede neural simples com uma única camada oculta para realizar uma determinada tarefa, mas não vamos realmente usar essa rede neural para a tarefa em que a treinamos! Em vez disso, o objetivo é apenas aprender os pesos da camada oculta – veremos que esses pesos são na verdade os "vetores de palavras" que estamos tentando aprender.

Concretamente, suponha que temos uma palavra "central" c e uma janela contextual em torno de c. Devemos nos referir às palavras que se encontram nesta janela contextual como "palavras externas". Por exemplo, na Figura 1, vemos que a palavra central c é 'endereço'. Como o tamanho da janela de contexto é 2, as palavras externas são 'correio', 'o', 'estava' e 'errado'.

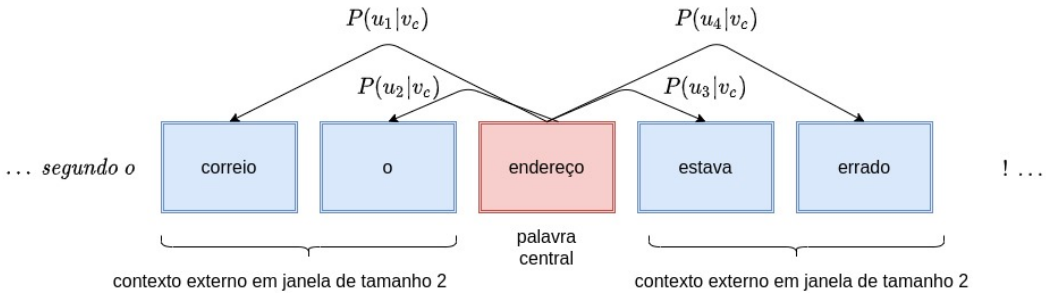

Fig. 1: O modelo de predição word2vec *skipgram* com janela de tamanho 2

No word2vec, a distribuição de probabilidade condicional de se observar a palavra o na janela contextual em torno da palavra central c é dada tomando o produto interno dos vetores u_o e v_c e aplicando a função softmax:

$$P(O = o|C = c) = \frac{\exp\left(u_o^\top \cdot v_c\right)}{\sum_{w \in Vocab} \exp(u_w^\top \cdot v_c)} \tag{1}$$

[1]Este exercício foi baseado em um trabalho da disciplina "CS224n: Natural Language Processing with Deep Learning"da universidade de Standford.

Neste caso, u_o é o vetor "externo" que representa a palavra externa o, e v_c é o vetor "central" que representa a palavra central c. Para armazenar esses parâmetros, temos duas matrizes, U e V. As colunas de U consistem dos vetores "externos" u_w. As colunas de V contêm todos os vetores "centrais" v_w. Ambos U e V contêm um vetor para cada $w \in$ *Vocabulário*; assumimos que cada palavra em nosso vocabulário corresponde a um número inteiro k, o índice da palavra no vocabulário, u_k representa tanto a k-ésima coluna de U quanto o vetor de palavras "externas" à palavra indexada por k. E v_k é tanto a k-ésima coluna de V quanto o vetor de palavra "central" para a palavra indexada por "k". Para simplificar a notação, usamos de maneira intercambiável k para referir à palavra e ao índice da palavra.

Para um único par de palavras c e o, o custo (*loss*) é dado por:

$$J_{\textit{naive-softmax}}(v_c, o, U) = -\log P(O = o|C = c). \tag{2}$$

Outra maneira de ver esse custo é como a *entropia cruzada*[2] entre a distribuição real y e a distribuição prevista $\hat{y}$. Tanto y quanto $\hat{y}$ são vetores com comprimento igual ao número de palavras no vocabulário. Além disso, a k-ésima entrada nesses vetores indica a probabilidade condicional de a k-ésima palavra ser uma palavra externa a c. A verdadeira distribuição empírica y é um vetor one-hot com 1 para a verdadeira palavra externa o e 0 em todos os outros lugares. A distribuição prevista $\hat{y}$ é a distribuição de probabilidade $P(O|C = c)$ dado pela equação (1).

Implementação do word2vec

Nesta parte você irá implementar o modelo word2vec e treinar seus próprios vetores de palavras com gradiente estocástico descendente (SGD). Para cada um dos métodos que você precisa implementar, incluímos aproximadamente quantas linhas de código nossa solução tem nos comentários do código. Esses números são incluídos para orientá-lo. Você não tem que se limitar a eles, você pode escrever um código mais curto ou mais longo como desejar. Se você acha que sua implementação é significativamente mais longa do que nossa sugestão, é um sinal de que existem alguns métodos de bibliotecas (e.g. `numpy`) que você pode utilizar para tornar seu código mais curto e mais rápido. Os loops `for` em Python demoram muito para serem concluídos quando usados em vetores e matrizes grandes, então esperamos que você utilize métodos `numpy`.

Pede-se:

(a) Primeiro, implemente o método `sigmoid`, que recebe um vetor e aplica a função sigmóide a ele. Em seguida, implemente o custo e o gradiente softmax no método `naiveSoftmaxLossAndGradient` e o custo da amostragem negativa e gradiente no método `negSamplingLossAndGradient`. Finalmente, preencha a implementação para o modelo skip-gram no método `skipgram`. Quando terminar, teste sua implementação executando `python word2vec.py`.

(b) Conclua a implementação para seu otimizador SGD no método `sgd` do arquivo *sgd.py*. Teste sua implementação executando `python sgd.py`.

[2] O custo por entropia cruzada entre a distribuição (discreta) de probabilidade verdadeira p e outra distribuição q é: $-\sum_i p_i \log(q_i)$.

(c) Hora da verdade! Agora vamos carregar alguns dados reais e treinar vetores de palavras com tudo que você acabou de implementar! Vamos usar o conjunto de dados do córpus da B2W para treinar vetores word2vec. Você precisará baixar os conjuntos de dados nesse endereço https://github.com/alan-barzilay/NLPortugues/blob/master/EP%201/data/b2w.csv para a pasta `dados`. Não há código adicional a ser escrito para esta parte; somente execute `python run.py`.

Observação: *o processo de treinamento pode demorar muito, dependendo da eficiência de sua implementação e o poder de computação de sua máquina (uma implementação eficiente leva de uma a duas horas). Planeje-se de acordo!*

Após o treinamento (40.000 iterações), o script será concluído e uma visualização para alguns vetores de palavras por nós selecionados aparecerá. Também será salvo como *vetores_de_palavras.png* no diretório do projeto. Inclua o arquivo PNG em sua entrega.