

Universidade Federal do Pampa

Marcos Vinícius Treviso

Aprendizagem Guiada para Análise Morfossintática usando Redes Neurais Recursivas

Alegrete

2015

Marcos Vinícius Treviso

Aprendizagem Guiada para Análise Morfossintática usando Redes Neurais Recursivas

Projeto de Trabalho de Conclusão de Curso
apresentado ao Curso de Graduação em Ci-
ênci da Computação da Universidade Fede-
ral do Pampa como requisito parcial para a
obtenção do título de Bacharel em Ciência
da Computação.

Orientador: Fábio Natanael Kepler

Alegrete

2015

"Existem muitas hipóteses em ciência que estão erradas. Isso é perfeitamente aceitável, eles são a abertura para achar as que estão certas."
(Carl Sagan - O Mundo Assombrado Pelos Demônios)

Resumo

Part-of-speech Tagging consiste em classificar uma palavra pertencente a um conjunto de textos em uma classe gramatical. Em Processamento de Linguagem Natural estamos sempre buscados métodos modernos para o processo de Part-of-speech Tagging, pois ele pode ser usado como pré-processamento de várias aplicações. Nós estudamos diferentes métodos para gerar representações de palavras (*word embeddings*) a partir de uma coleção de textos denominada *córpus*. Criamos então um modelo baseado em aprendizagem profunda utilizando uma rede neural recursiva que recebe essas representações como entrada e gera como saída uma classe gramatical associada. O modelo neural é guiado, onde palavras mais fáceis de serem analisadas são classificadas primeiro. A saída da rede é uma classe vetorizada, que é complementada na entrada com o vetor da própria palavra classificada. O treinamento é feito sobre três diferentes *córpus* etiquetados para o português brasileiro.

Palavras-chave: Aprendizado de Máquina. Aprendizagem Profunda. Processamento de Linguagem Natural. Part-of-speech Tagging. Redes Neurais Recursivas. Word Embeddings.

Abstract

Part-of-speech Tagging consists in classify a given word, that belongs to a collections of texts, with particular part of speech tag. Part-of-speech Tagging can be used as pre-processing of many applications, so, in Natural Language Processing we are always searching for improvement methods. We study different methods to generate words representations (word embeddings) from a collection of texts denominated corpora. We create a deep learning model by using a recursive neural network that receives as input the word embeddings and predicts a part-of-speech tag to the associated word. The neural model is guided, where easy words to predict are classified first. The output of the network is a vector tag, that is then complemented with the input vector of the same word classified. The training process occur over three different tagged corpora for brazilian portuguese.

Key-words: Machine Learning. Deep Learning. Natural Language Processing. Part-of-speech Tagging. Recursive Neural Networks. Word Embeddings.

Listas de ilustrações

Figura 1 – Exemplo de classificação gramatical	17
Figura 2 – Diagrama para resultado final	21
Figura 3 – Demonstração de regressão	22
Figura 4 – Demonstração de classificação	23
Figura 5 – Função sigmoide	25
Figura 6 – Representação dos casos da função de custo	27
Figura 7 – Função de custo - $J(\theta_0, \theta_1)$	28
Figura 8 – Funcionamento do Gradiente Descendente	29
Figura 9 – Exemplo de <i>underfitting</i> e <i>overfitting</i>	31
Figura 10 – Exemplo de matriz de coocorrência para criação de palavras vetorizadas	34
Figura 11 – Exemplo de vetores de <i>features</i> de cinco dimensões representando palavras	35
Figura 12 – t-SNE: Visualização para <i>word embeddings</i>	36
Figura 13 – Exemplo de palavras similares	37
Figura 14 – Exemplos de representação de sensores no cérebro	38
Figura 15 – Representação de um neurônio	38
Figura 16 – Abstração matemática de um neurônio	39
Figura 17 – Rede neural com uma camada oculta	40
Figura 18 – Modelo de rede neural - Passos do <i>Forward Propagation</i>	43
Figura 19 – Passos do Backpropagation	46
Figura 20 – Modelo de aprendizagem profunda	48
Figura 21 – Sentido do aprendizado de máquina	49
Figura 22 – Tamanho das redes neurais ao longo dos anos	50
Figura 23 – Funcionamento de aprendizagem profunda em POS Tagging	50
Figura 24 – Modelo da rede neural	55
Figura 25 – Grafo de transições de palavras mais fáceis	56

Lista de tabelas

Tabela 1 – Notação utilizada para classificação	24
Tabela 2 – Dados dos <i>córpus</i>	33
Tabela 3 – Notação utilizada para representação em redes neurais	40
Tabela 4 – Notação utilizada para aprendizado em redes neurais	44
Tabela 5 – Comparativo das técnicas encontradas na literatura para POS Tagging	52
Tabela 6 – Comparativo dos melhores resultados encontrados na literatura para POS Tagging	52
Tabela 7 – Notação utilizada para o treinamento do modelo	53
Tabela 8 – Cronograma de atividades restantes	59

Lista de siglas

FDV Fora do Vocabulário

GloVe *Global Vectors*

HAL *Hyperspace Analogue to Language*

NLM *Neural Language Model*

PLN Processamento de Linguagem Natural

POS *Part-of-speech*

SG *Skip-Gram*

Sumário

1	INTRODUÇÃO	17
1.1	O problema	18
1.2	Objetivo	18
1.3	Estrutura do trabalho	19
2	FUNDAMENTOS	21
2.1	Aprendizado de máquina	21
2.1.1	Classificação	23
2.1.2	Função de hipótese	24
2.1.3	Função de custo	26
2.1.4	Minimização da função de custo	28
2.1.5	Classificação multiclasse	30
2.1.6	Regularização	30
2.2	<i>Corpus</i> e seu conjunto de classes gramaticais	32
2.3	Representação das palavras	33
2.4	Redes neurais	37
2.4.1	Neurocomputação	37
2.4.2	Representação do modelo	38
2.4.3	Representação vetORIZADA do modelo	41
2.4.4	Classificação multiclasse	42
2.4.5	Função de custo	43
2.4.6	Algoritmo <i>Backpropagation</i>	44
2.4.7	Treinamento	45
2.5	Aprendizagem profunda	47
3	TRABALHOS RELACIONADOS	51
4	METODOLOGIA	53
4.1	Representação das palavras	53
4.2	Pontuações para estrutura gramatical	54
4.3	Treinamento	55
5	CRONOGRAMA DE ATIVIDADES	59
	Referências	61

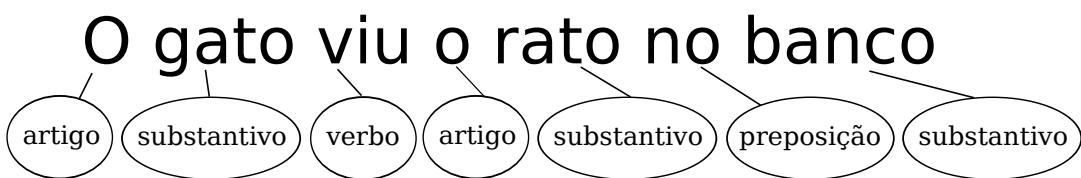
1 Introdução

A análise morfossintática de uma palavra consiste em atribuí-la à sua correta classe gramatical de acordo com seu contexto. O ato de classificar uma palavra pertencente a um conjunto de textos em uma classe gramatical depende de sua estrutura morfossintática. Esse ato é conhecido no campo de Processamento de Linguagem Natural ([PLN](#)) como *Part-of-speech (POS) Tagging*, ou etiquetagem na literatura brasileira. A [Figura 1](#) ilustra esse processo.

A principal medida dessa classificação é justamente a eficiência com o qual cada classe gramatical é atribuída para certa palavra, nesse quesito, há vários métodos propostos recentemente que conseguiram uma eficiência de cerca de 97% ([SANTOS; ZADROZNY, 2014](#); [COLLOBERT, 2011](#); [FONSECA; ROSA; ALUÍSIO, 2015](#)). Ou seja, cerca de uma palavra a cada vinte é classificada errada.

Apesar de muitos desses métodos já serem utilizados em larga escala, em [PLN](#) estamos sempre buscando ganhar mais desempenho, já que [POS Tagging](#) pode ser aplicada em uma grande variedade de aplicações, como tradução automática e extração de informações de textos ([MANNING; SCHÜTZE, 1999](#)), ferramentas de auxílio à leitura e escrita ([MARQUIAFÁVEL, 2010](#)), entre outras.

[Figura 1: Exemplo de classificação gramatical](#)



A ambiguidade é uma propriedade que faz com que um objeto linguístico possa ser interpretado de modos diferentes. Existem dois tipos de ambiguidade: a ambiguidade sintática, que ocorre quando um lexema pode pertencer a mais de uma classe gramatical; ambiguidade semântica, que ocorre quando um objeto linguístico traz significados diferentes, dando sentidos diferentes à sentença. A ambiguidade¹ é um grande problema a ser considerado no processo de [POS Tagging](#).

¹ Quando é empregado o termo “ambiguidade” no contexto de [POS Tagging](#), fica subentendido de que se trata de uma ambiguidade sintática.

1.1 O problema

POS Tagging é um processo difícil de ser realizado em **PLN**, pois linguagens naturais tem bastante ambiguidade. Sendo que há muita ambiguidade no Português do Brasil, visto que é uma língua com uma sintaxe flexível e que possui uma rica morfologia. Essa ambiguidade dificulta a análise morfossintática, porque não é possível determinar a priori qual classe gramatical a palavra sendo analisada pertence. No exemplo anterior, da [Figura 1](#), não fica claro qual é o tipo de *banco* que está sendo referenciado, onde pode ser classificado como um verbo ou como um substantivo. Para resolver o problema da ambiguidade, é necessário analisar os lexemas vizinhos de uma dada palavra, ou seja, é preciso analisar o seu contexto associado.

Uma estratégia trivial seria utilizar um dicionário com uma função de mapeamento de um para um, onde a *chave* seria a palavra e o *valor* seria a classe gramatical. Infelizmente essa técnica requer muitos recursos computacionais, visto que o número de entradas seria grande por ter todas as palavras possíveis de vocabulário brasileiro, caso contrário, haveria o problema de ter uma palavra fora do vocabulário, e portanto ela não teria uma classe gramatical associada. Porém, o principal revés dessa estratégia é a ambiguidade, que faz com que uma palavra tenha mais que uma classe gramatical associada, e portanto não é possível mapear com indubitabilidade de que a classe associada é a correta sem antes analisar o contexto.

Dito isso, este trabalho consiste em desenvolver um método para classificar palavras em suas respectivas classes gramaticais de modo eficiente. Uma abordagem que está sendo amplamente utilizada para resolver esse problema é aprendizado de máquina, pois ela permite treinar um modelo que aprende padrões morfológicos, sintáticos e semânticos de uma sentença. E em ordem de conseguir solucionar esse problema com eficiência, é necessário escolher um bom método computacional. Este trabalho se baseará na utilização de um método de aprendizagem profunda, que utiliza um modelo neural recursivo com múltiplas camadas. Utilizamos redes neurais pois elas oferecem um jeito alternativo de realizar aprendizado de máquina quando temos hipóteses complexas com muitas características diferentes.

1.2 Objetivo

Este trabalho irá propor um novo método de classificação de palavras em classes gramaticais e analisar sua eficiência em relação a trabalhos já publicados que utilizam métodos já consolidados. Primordialmente, isso será feito para o escopo da língua portuguesa brasileira.

Para buscar uma boa eficiência será proposto um método original, que se baseia

em classificar primeiramente palavras mais fáceis, desse modo, espera-se deixar palavras ambíguas por último. Por exemplo, para a [Figura 1](#) deixaríamos a palavra *banco* para o final. Além disso, o método se baseia numa técnica que representa palavras e classes gramaticais em vetores reais multivalorados. Essa técnica será explicada na [seção 2.3](#).

Como já mencionado, o estado da arte atual tem atualmente cerca de 97% de acurácia, tentaremos ultrapassar esse limite aplicando novas técnicas de classificação e utilizando características significantes das palavras. A acurácia da classificação não será a única medida levada em consideração, o tempo de processamento gasto no treinamento do modelo também será.

1.3 Estrutura do trabalho

Este trabalho está dividido na seguinte maneira: No [Capítulo 2](#) é mostrado os principais fundamentos necessários para entender o método proposto e seus conceitos relacionados; após passar os fundamentos, será apresentado no [Capítulo 3](#) os trabalhos relacionados que procuram resolver o problema de [POS Tagging](#) utilizando diferentes técnicas e métodos. Depois, no [Capítulo 4](#) será detalhado aspectos da metodologia a ser aplicada e a explicação do método proposto juntamente com as técnicas utilizadas. Por fim, no [Capítulo 5](#) será apresentado o cronograma de tarefas previstas até a conclusão deste trabalho.

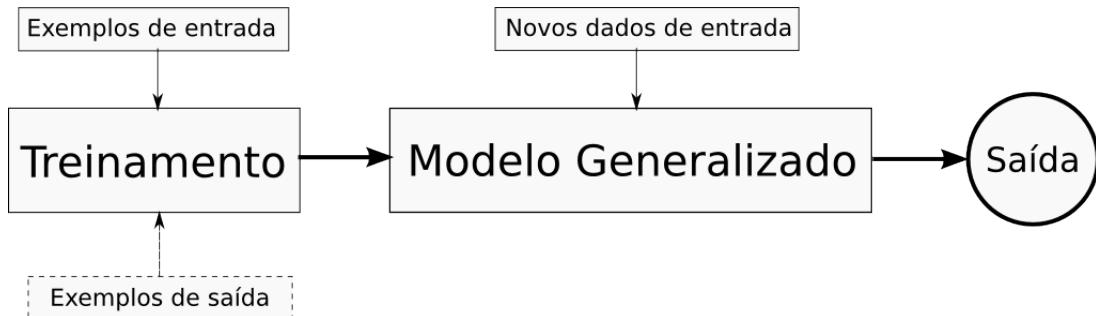
2 Fundamentos

Devido aos desafios mostrados na [seção 1.1](#), este trabalho é baseado em conceitos de aprendizado de máquina, que dedica-se na elaboração de algoritmos e técnicas que permitem a um computador aprender padrões.

2.1 Aprendizado de máquina

Para realizar a aprendizagem é necessário, a princípio, dados de entrada (*features*) que servem como exemplo para nosso modelo. Com esses dados é possível treinar o modelo para que ele possa então aprender com base nesses exemplos. Depois de realizar o treinamento, é possível generalizar sobre outros dados ainda não testados e gerar uma resposta apropriada como saída. O diagrama da [Figura 2](#) mostra os passos para obter o resultado final.

Figura 2: Diagrama para resultado final



O aprendizado de máquina pode ser dividido em duas abordagens: o **aprendizado supervisionado**, onde é dado um conjunto de dados de entrada e já se sabe como a saída deve parecer, tendo a ideia de que há uma relação entre a entrada e a saída; o **aprendizado não supervisionado**, que permite abordar problemas com pouca ou até nenhuma ideia de como os resultados devem parecer, nesse caso a caixa de exemplos de saída pode não existir.

Neste trabalho, utilizaremos o aprendizado supervisionado, pois já temos um conjunto de classes gramaticais como saída esperada.

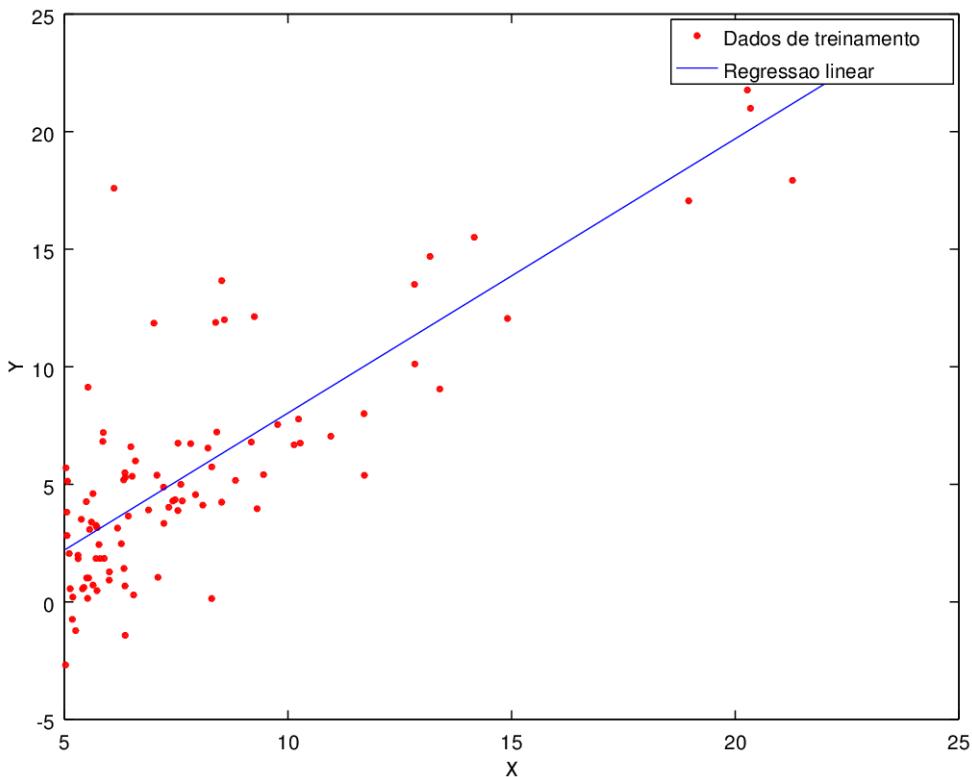
O aprendizado supervisionado permite dividir os problemas em duas categorias distintas:

- **Regressão:** Tenta-se prever resultados com uma saída contínua, significando que

deseja-se mapear variáveis de entrada em alguma função contínua. A Figura 3 demonstra esse processo. Os círculos em vermelho são dados de exemplo, já a reta azul representa a predição feita pela regressão.

- **Classificação:** Tenta-se prever resultados em uma saída discreta, ou seja, deseja-se mapear variáveis de entrada em categoriais discretas. A Figura 4 demonstra esse processo. Tem duas classes diferentes, as representadas pelos círculos amarelos e pelas cruzes vermelhas, já o contorno em verde representa o resultado da classificação sobre esses dados de exemplo.

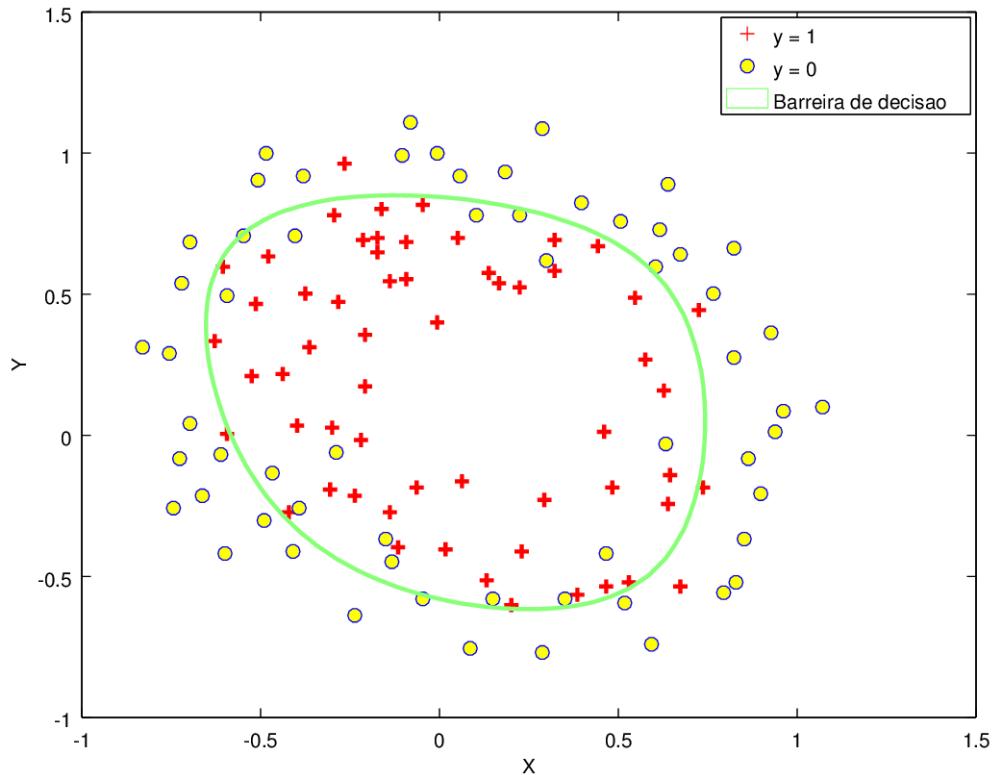
Figura 3: Demonstração de regressão



Fonte: [Ng \(2015\)](#)

Logo adiante explicaremos os fundamentos de classificação e uma técnica para resolver esse problema chamada de regressão logística (JR; LEMESHOW, 2004). E em virtude de apresentar conceitos fundamentais a serem aplicados posteriormente nas redes neurais, falaremos um pouco mais sobre regressão logística, e também sobre propriedades importantes que serão referenciadas durante todo o trabalho.

Figura 4: Demonstração de classificação

Fonte: [Ng \(2015\)](#)

2.1.1 Classificação

Em um problema de classificação, nossa saída Y vai ser um vetor com valores sendo apenas zero ou um.

$$Y \in \{0, 1\}$$

A equação acima está tratando apenas duas classes. Sendo assim, esse problema é chamado de classificação binária. Para resolver esse problema, um método que pode ser utilizado é regressão logística.

A fim de simplificar o uso das variáveis, faremos o uso de uma notação que é normalmente utilizada em textos de aprendizado de máquina, ela pode ser vista na [Tabela 1](#).

Tabela 1: Notação utilizada para classificação

X	dados de entrada ou <i>features</i>
Y	dados de saída
$X_j^{(i)}$	o valor da <i>feature</i> j no i -ésimo exemplo de treinamento
$X^{(i)}$	o vetor coluna de todas as <i>features</i> no i -ésimo exemplo de treinamento
m	número de exemplos de treinamento
n	$ X^{(i)} $, o número de <i>features</i>
(x, y)	um exemplo de treinamento
$(X^{(i)}, Y^{(i)})$	o i -ésimo exemplo de treinamento
θ	parâmetros a serem aprendidos

2.1.2 Função de hipótese

A função de hipótese tem o objetivo de mapear funções aleatórias dentro do intervalo de saída Y , para isso são atribuídos valores para os parâmetros θ s. Essa relação está definida na [Equação 2.1](#). Observe que ela tem apenas dois parâmetros. Na realidade uma hipótese pode ter vários parâmetros relacionados com os dados de entrada X , se tiver n dados de entrada, então haverá $n+1$ parâmetros. A forma geral da função de hipótese está mostrada na [Equação 2.2](#).

$$h_{\theta}(x) = \theta_0 + \theta_1 x \quad (2.1)$$

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \quad (2.2)$$

Uma maneira de simplificar a função de hipótese é trabalhar com a definição de multiplicação de matrizes. Então a função de hipótese pode ser representada de uma forma vetorizada, como mostrado na equação abaixo.

$$h_{\theta}(x) = [\theta_0 \ \theta_1 \ \dots \ \theta_n] \times \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T X$$

Para que seja possível fazer operações de matrizes com θ e X , será atribuído $X_0^{(i)} = 1$ para todos os valores de i . Isso faz com que os vetores θ e $X^{(i)}$ combinem um com o outro elementarmente.

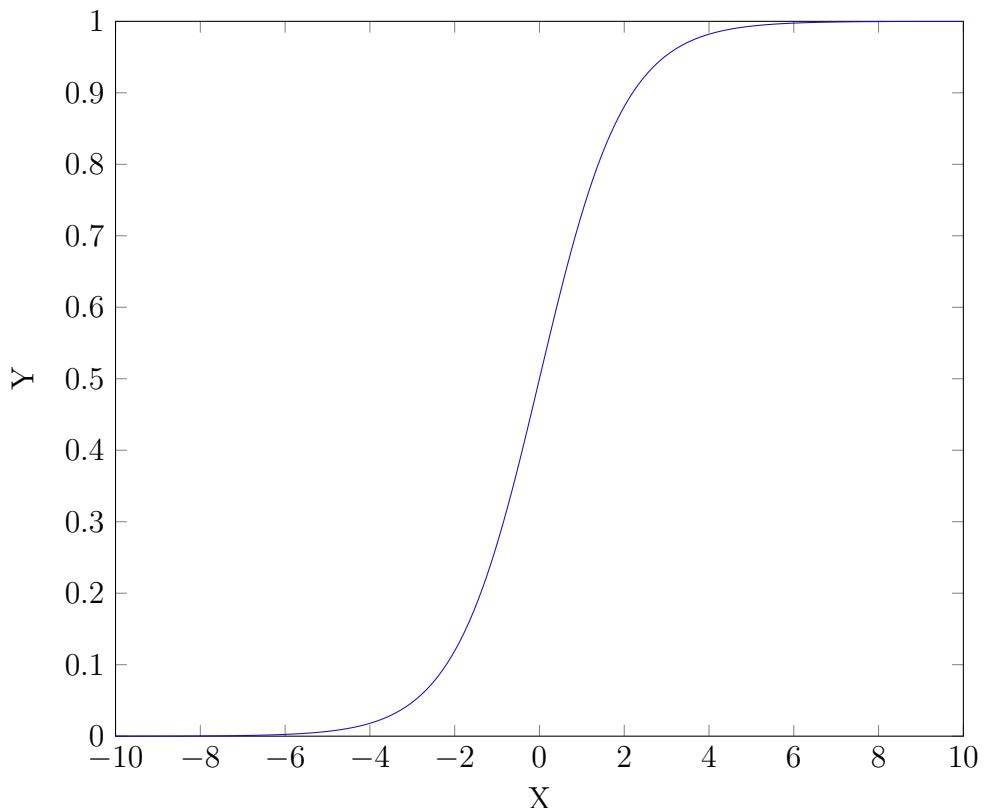
Portanto, nossa função de hipótese deve satisfazer a equação:

$$0 \leq h_{\theta}(x) \leq 1$$

Para fazer com que uma entrada contínua fique nesse intervalo, é necessário utilizar uma função linear que faça esse mapeamento de modo linear, e para isso mudamos a forma da função de hipótese. A nova forma usa a função sigmoide, também conhecida como função logística.

$$\begin{aligned} h_{\theta}(x) &= g(\theta^T X) \\ g(z) &= \frac{1}{1 + e^{-z}} \end{aligned} \tag{2.3}$$

Figura 5: Função sigmoide



A função da Equação 2.3 representada na Figura 5 mapeia qualquer número no intervalo $[0, 1]$, tornando isso útil para transformar qualquer valor arbitrário em uma função mais ideal para problemas de classificação.

Para calcular a função de hipótese, podemos colocar o $\theta^T X$ na função logística. Isso nos dará a probabilidade de que a saída é 1.

$$h_{\theta}(x) = P(y = 1|X; \theta) = 1 - P(y = 0|X; \theta)$$

Ou seja, a probabilidade de nossa predição ser 1 é o oposto da probabilidade de ser 0. Sendo assim, a soma das probabilidades para $y = 0$ e $y = 1$ deve ser 1.

Para ter uma classificação discreta podemos traduzir a saída da função de hipótese de acordo com a equação:

$$h_{\theta}(x) \geq 0.5 \Rightarrow y = 1$$

$$h_{\theta}(x) < 0.5 \Rightarrow y = 0$$

Então, se a entrada é $\theta^T X$, isso significa que:

$$h_{\theta}(x) = g(\theta^T X) \geq 0.5 \quad \text{quando} \quad \theta^T X \geq 0$$

A partir disso, podemos dizer que:

$$\theta^T X \geq 0 \Rightarrow y = 1$$

$$\theta^T X < 0 \Rightarrow y = 0$$

O **limite de decisão** (ou barreira de decisão) é a linha que separa a área quando $y = 0$ e quando $y = 1$. Isso é criado pela função de hipótese. Uma observação importante é que a função de hipótese não precisa ser linear, pode ser até mesmo um círculo ou ter qualquer forma para ajustar os dados, como mostrado na [Figura 4](#). Isto é, uma função de hipótese pode ter várias *features*. Novas *features* podem ser criadas ao combinar as *features* já existentes e modificando-as. Por exemplo, se houvesse dois dados de entrada x_1 e x_2 , seria possível criar a função de hipótese:

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \theta_6 x_1^2 x_2^2 + \dots$$

Todavia, essa abordagem pode gerar problemas de desempenho. Algumas *features* podem também acabar se tornando obsoletas e não ajudar em nada no resultado final, tornando seu cálculo inútil. Mas o principal problema dessa abordagem é o *overfitting*, que faz com que a função de hipótese não generalize sobre dados desconhecidos. Esse problema será tratado na [subseção 2.1.6](#).

2.1.3 Função de custo

Para medir a precisão da função de hipótese é necessário uma função que diga o quão precisa aquela hipótese é. Tal função é denominada de função de custo. Com ela, é feita uma média de todos os resultados das hipóteses com a entrada X comparada com o valor objetivo Y . Ela pode ser expressa como na [Equação 2.4](#).

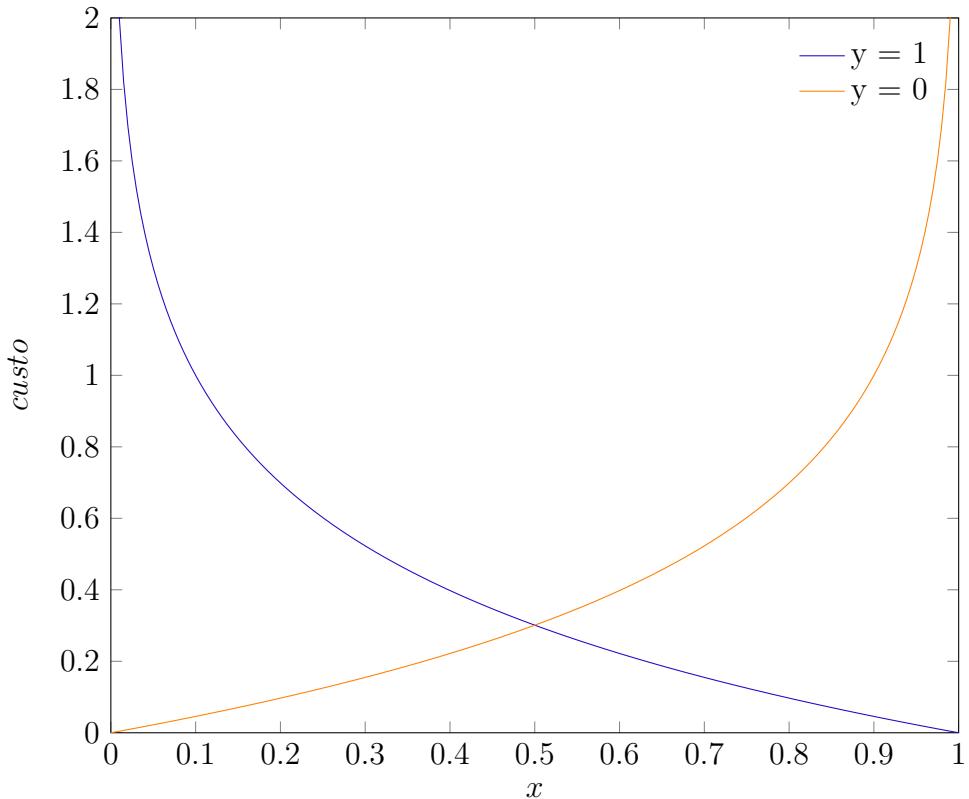
$$J(\theta) = \frac{1}{m} \sum_{i=1}^m Custo(h_{\theta}(x^{(i)}, y^{(i)})) \tag{2.4}$$

Onde,

$$Custo(h_\theta(x^{(i)}, y^{(i)})) = \begin{cases} -\log(h_\theta(x)) & \text{se } y = 1 \\ -\log(1 - h_\theta(x)) & \text{se } y = 0 \end{cases} \quad (2.5)$$

A Equação 2.5 captura a intuição de que se $h_\theta(x) = 0$, mas $y = 1$, então o algoritmo de aprendizado será penalizado por um custo muito alto. A Figura 6 demonstra os dois respectivos casos dessa equação.

Figura 6: Representação dos casos da função de custo

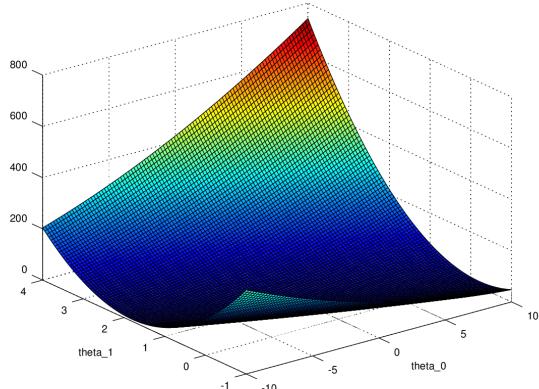


O quanto mais a função de hipótese está longe de y , maior é a saída da função de custo. Se a função de hipótese é igual a y , então o custo é 0.

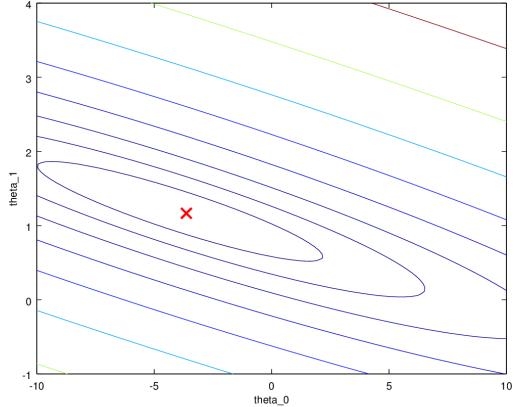
Podemos simplificar a Equação 2.5 com dois casos condicionais em apenas um caso:

$$Custo(h_\theta(x), y) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x)) \quad (2.6)$$

Nota-se que quando y é igual a 1, o segundo termo será 0 e não afetará o resultado. E quando y é igual a 0, o primeiro termo será 0 e não afetará o resultado. Aplicando a

Figura 7: Função de custo - $J(\theta_0, \theta_1)$ 

(a) Gráfico de superfície



(b) Gráfico de contorno

Fonte: Ng (2015)

Equação 2.6 na função de custo da equação Equação 2.4, podemos reescrevê-la como a equação Equação 2.7 ou na versão vetorizada da Equação 2.8.

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left(y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right) \quad (2.7)$$

$$J(\theta) = -\frac{1}{m} \left(\log(g(X\theta))^T Y + \log(1 - g(X\theta))^T (1 - Y) \right) \quad (2.8)$$

O objetivo da regressão logística é minimizar a função de custo em relação aos parâmetros θ s. Usando dois parâmetros é possível visualizar uma função de custo através do número de iterações, isso pode ser observado no exemplo da Figura 7(a) e da Figura 7(b).

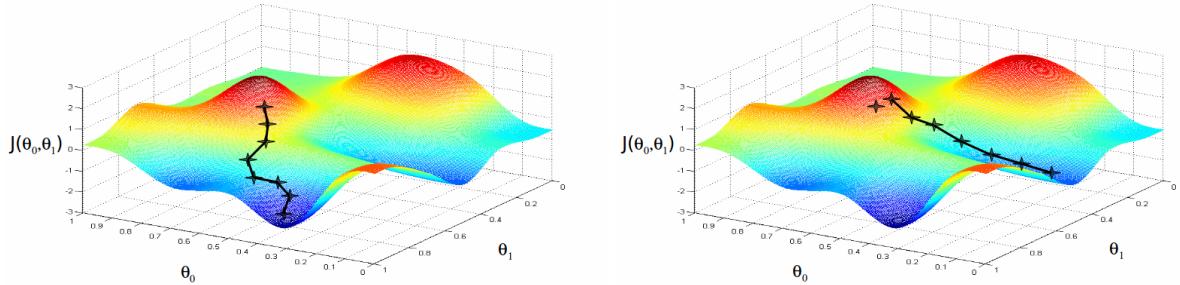
2.1.4 Minimização da função de custo

Para essa tarefa é necessário a aplicação de um algoritmo que pega a função de custo e tente minimizá-la, e por fim retorne os valores dos parâmetros θ s aprendidos. Com isso, estaremos melhorando nossa função de hipótese.

Um dos algoritmos mais utilizados para essa tarefa é o **Gradiente Descendente** (MICHALSKI; CARBONELL; MITCHELL, 2013). Seu funcionamento baseia-se em seguir a derivada da função de custo em relação aos parâmetros θ s alternadamente, com isso a encosta da tangente dará a direção para seguir adiante. Além disso, é aplicada uma taxa de aprendizagem α a cada iteração, para tentar controlar a convergência.

Seu funcionamento é descrito pelo algoritmo 2.9.

Figura 8: Funcionamento do Gradiente Descendente



(a) Convergindo para uma mínima global

(b) Convergindo para uma mínima local

Fonte: [Ng \(2015\)](#)

repita até convergir {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \quad \text{para } j = 0, 1, \dots, n \quad (2.9)$$

}

Ao trabalhar sobre a derivada parcial da função de custo, podemos chegar na [Equação 2.10](#) e adiante na versão vetorizada na [Equação 2.11](#).

$$\theta_j := \theta_j - \alpha \sum_{i=1}^m \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \quad \text{para } j = 0, 1, \dots, n \quad (2.10)$$

$$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - Y) \quad (2.11)$$

Na [Figura 8\(a\)](#) é possível visualizar esse funcionamento sobre os parâmetros. Observe que nesse caso o algoritmo consegue convergir para o mínimo global, porém em alguns casos (dependendo de nossa taxa de aprendizagem α) pode-se convergir para um mínimo local como mostrado na [Figura 8\(b\)](#). Basicamente, se α for um valor muito alto, os passos para o mínimo ótimo serão grandes e portanto o risco de divergência será maior, caso seja muito baixo, os passos para o mínimo ótimo serão pequenos e portanto irá demorar muito para convergir. [Ng \(2015\)](#) diz para testar manualmente a taxa de aprendizagem, começando com 0,001 e ir multiplicando por 10 esse valor até atingir uma divergência no resultado. Sendo assim, é possível analisar o melhor valor da taxa de aprendizagem e selecioná-la para o treinamento definitivo.

O Gradiente Descendente é apenas um dos vários algoritmos que existem para minimizar a função de custo. De acordo com [Ng \(2015\)](#), há alternativas mais sofisticadas como o Gradiente Conjugado, Broyden–Fletcher–Goldfarb–Shanno (BFGS) e Limited-memory-BFGS (L-BFGS), onde além de serem mais rápidos que o Gradiente Descendente,

não é preciso selecionar manualmente o valor de α . Mas, também é sugerido que não devemos tentar codificar esses algoritmos, visto que são mais complexos e requerem um bom conhecimento de cálculo numérico, onde ao invés podemos usar bibliotecas otimizadas que implementam esses algoritmos.

2.1.5 Classificação multiclasse

O **POS** Tagging é um problema de classificação multiclasse, onde deve-se etiquetar uma palavra em uma de várias categorias gramaticais possíveis. É possível fazer isso ao expandir nossa definição para que a saída seja $Y = \{0, 1, \dots, n\}$. Nesse caso, é dividido o problema em $n + 1$ problemas de classificação binária e em cada um será feito a predição da probabilidade de que y é membro de uma dessas classes.

$$\begin{aligned} h_{\theta}^{(0)}(X) &= P(y = 0|X; \theta) \\ h_{\theta}^{(1)}(X) &= P(y = 1|X; \theta) \\ &\vdots \\ h_{\theta}^{(n)}(X) &= P(y = n|X; \theta) \end{aligned}$$

E então escolhe-se a classe com o valor de hipótese mais alto: $\max_i(h_{\theta}^{(i)}(X))$.

2.1.6 Regularização

Regularização é uma técnica importante para resolver o problema de *overfitting*.

Quando se trabalha com abordagens de aprendizagem supervisionada, tem-se dois problemas que podem ocorrer dependendo das *features* escolhidas. O ***underfitting*** ocorre quando a forma da função de hipótese mapeia mal a tendência dos dados. Isso é causado por uma função que é muito simples ou que usa poucas *features*. Em contrapartida, ***overfitting*** é causado por uma função de hipótese que encaixa os dados avaliados mas não generaliza bem para classificar novos dados. É usualmente causado por uma função complexa que cria muitas curvas e ângulos desnecessários.

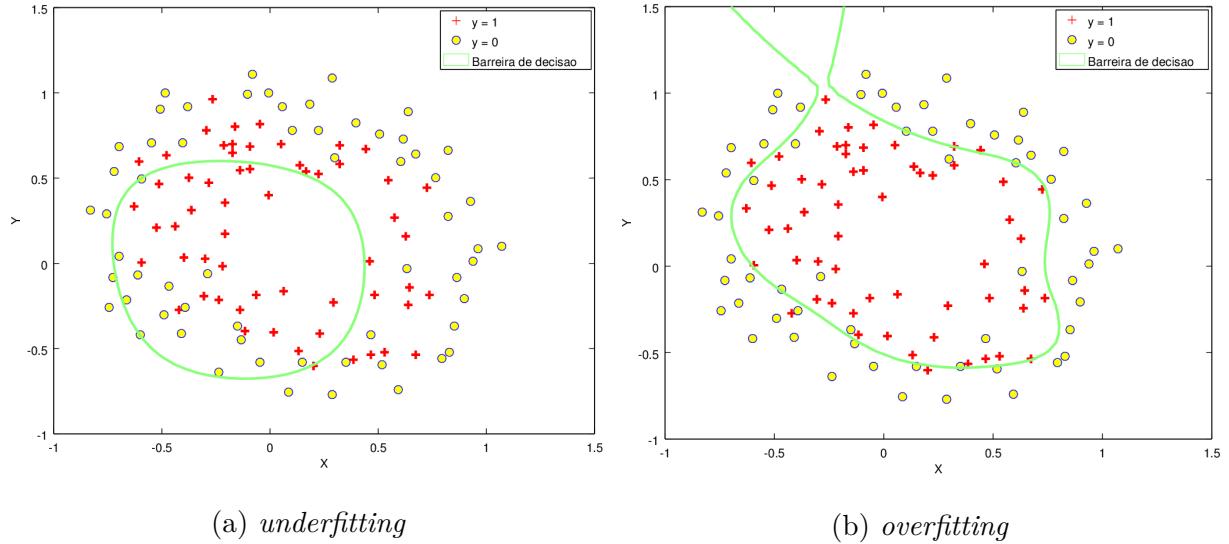
Uma boa classificação para o conjunto de dados mostrados na [Figura 9](#) é o exemplo mostrado na [Figura 4](#).

Segundo ([NG, 2015](#)), há duas opções principais para resolver o problema de *overfitting*:

a) Reduzir o número de *features*:

- Manualmente selecionar quais *features* usar;
- Usar um algoritmo de seleção de modelo.

b) Aplicar regularização:

Figura 9: Exemplo de *underfitting* e *overfitting*

Fonte: Ng (2015)

- Manter todos as *features*, mas reduzir os parâmetros θ_j .

Em geral, regularização funciona bem quando há bastante features levemente úteis. Quando há *overfitting* da função de hipótese, é possível reduzir a influência de alguns termos da função aumentando os seus custos. Por exemplo, caso quiséssemos eliminar a influência de θ_3x^3 e θ_4x^4 da Equação 2.12, poderíamos mudar a forma da função de custo. Com isso, a função de hipótese não é alterada e as *features* não são eliminadas, como mostrado na Equação 2.13.

$$h_{\theta}(x) = \theta_0 + \theta_1x^1 + \theta_2x^2 + \theta_3x^3 + \theta_4x^4 \quad (2.12)$$

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left(y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)})) \right) + \underbrace{1000\theta_3^2 + 1000\theta_4^2}_{\text{termo de regularização}} \quad (2.13)$$

Com isso, adicionamos dois termos extras no fim da função de custo para inflar o custo de θ_3 e θ_4 . Agora, para a função de custo chegar a zero, nós teremos que reduzir os valores de θ_3 e θ_4 para próximos a zero. Isso vai reduzir substancialmente o peso das *features* cuja influência queremos eliminar.

É possível regularizar todos os parâmetros em um único somatório conforme mostrado na Equação 2.14. Isso pode ser feito com a inclusão do parâmetro de regularização

λ , que determina o quanto os custos dos parâmetros θ s serão inflados.

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left(y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right) + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2 \quad (2.14)$$

Observe que o índice j do segundo somatório começa em 1, isso significa que não estamos regularizando o θ_0 .

Como foi regularizado a função de custo, há de se regularizar o algoritmo que realiza sua minimização também. Quando é trabalhado com o Gradiente Descendente isso pode ser feito como no algoritmo 2.15. Como na função de custo, não queremos regularizar o θ_0 e então precisamos separá-lo em um outro passo.

repita até convergir {

$$\begin{aligned} \theta_j &:= \theta_j - \frac{\alpha}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} && \text{se } j = 0 \\ \theta_j &:= \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} + \frac{\lambda}{m} \theta_j \right] && \text{se } j > 0 \end{aligned} \quad (2.15)$$

}

Com isso a regressão logística pode ser aplicada normalmente, pois a regularização vai cuidar do *overfitting*. Entretanto, para casos mais complexos, que envolve a criação de muitas *features* como no caso de POS Tagging, há uma melhor abordagem para realizar a etiquetagem. Essa abordagem é conhecida como redes neurais e ela será discutida na seção 2.4.

2.2 Córpus e seu conjunto de classes gramaticais

Córpus são coleções de textos agrupados. Eles são os exemplos de entrada para o treinamento dos parâmetros de um modelo. Eles podem conter informações adicionais, como classes gramaticais associadas das palavras. A anotação das palavras no *córpus* é feita manualmente por especialistas.

Esses conjuntos podem se diferenciar na sua granularidade, como por exemplo, podem ter diferentes classes gramaticais para nomes no plural e no singular, ou agrupar eles em uma única classe (FONSECA; ROSA; ALUÍSIO, 2015). No inglês, há vários conjuntos de classes gramaticais que são amplamente utilizados, são o Penn Treebank tagset (PENNSYLVANIA, 2014), CLAWS5 e CLAWS7.

Já no português, os *córpus* estão evoluindo com o tempo. Embora alguns erros são encontrados neles (FONSECA; ROSA, 2013), eles ainda são a melhor opção devido a seu tamanho e qualidade. Os principais e mais utilizados para o treinamento supervisionado

são o Mac-Morpho ([ALUÍSIO et al., 2003](#)) com cerca de um milhão de palavras, que retrata artigos publicados na Folha de São Paulo em 1994. O Tycho Brahe ([UNICAMP, 2010](#)) que tem cerca de dois milhões de palavras de 66 textos históricos entre 1380 e 1881. Há também o Bosque ([AFONSO et al., 2002](#)), um *córpus* parseado que contém cerca de 185 mil palavras. Além desses, ([FONSECA; ROSA; ALUÍSIO, 2015](#)) apresenta uma versão revisada do Mac-Morpho, com classes gramaticais novas e junção de outras. Dados referentes a cada *córpus* podem ser encontrados na [Tabela 2](#).

Tabela 2: Dados dos *córpus*

Córpus	Sentenças	Palavras	Classes gramaticais
Mac-Morpho original	53,374	1,221,465	41
Mac-Morpho revisado	49,932	945,958	26
Tycho Brahe	96,125	2.842.809	265

Infelizmente esses *córpus* não podem ser combinados em um só, já que eles se diferenciam no conjunto de classes gramaticais e também no seu uso associado. Uma possível alternativa para essa combinação seria o uso de uma aprendizagem não supervisionada, aplicando técnicas de clusterização.

A aprendizagem supervisionada, utilizando *córpus* como entrada, tem se mostrado uma estratégia atrativa, já que pode ser usado recursos criados com boa eficiência. Devido a isso, nesse trabalho vamos utilizar três *córpus*: o Mac-Morpho original; Mac-Morpho revisado e o Tycho Brahe. O conjunto de etiquetas para esses *córpus* podem ser encontrados, respectivamente, em ([ALUÍSIO et al., 2003](#)), ([FONSECA; ROSA; ALUÍSIO, 2015](#)) e ([UNICAMP, 2010](#)).

2.3 Representação das palavras

Palavras¹ podem ser representadas de várias maneiras em um modelo de aprendizagem, podendo ser até mesmo feito o uso real da palavra como uma *feature*.

No entanto, uma abordagem recente que vem obtendo sucesso é a utilização de *word embeddings*, elas são representação de palavras como vetores reais valorados em um espaço multidimensional ([TURIAN; RATINOV; BENGIO, 2010](#)). Elas podem ser geradas de maneiras diferentes dependendo da técnica utilizada. Abordagens clássicas baseiam-se na frequência e coocorrência das palavras vizinhas. A [Figura 10](#) ilustra essa um processo que baseia-se na coocorrência de palavras vizinhas, onde um processo para gerar o vetor da palavra é simplesmente buscar a linha ou coluna dessa palavra na matriz, ou utilizar Decomposição em Valores Singulares ([SOCHER, 2015](#)). Ultimamente o processo

¹ Uma palavra pode ser qualquer conjunto de caracteres, inclusive pontuações, números, etc.

de geração também tem sido feito através de redes neurais, onde é possível capturar informações sintáticas e semânticas sobre as palavras (COLLOBERT et al., 2011).

Figura 10: Exemplo de matriz de coocorrência para criação de palavras vetorizadas

EXEMPLO DE VOCABULÁRIO:

- Eu gosto de ciéncia.
- Eu gosto de PLN.
- Eu amo estudar algoritmos.

	Eu	gosto	de	ciéncia	PLN	amo	estudar	algoritmos	.
Eu	0	2	0	0	0	0	0	0	0
gosto	2	0	2	0	0	0	0	0	0
de	0	2	0	1	0	1	0	0	0
ciéncia	0	0	1	0	0	0	0	0	1
PLN	0	0	1	0	0	0	0	0	1
amo	1	0	0	0	0	0	1	0	0
estudar	0	0	0	0	0	1	0	1	0
algoritmos	0	0	0	0	0	0	1	0	1
.	0	0	0	1	1	0	0	1	0

As *word embeddings* conseguem mapear palavras em um espaço relativamente pequeno (usando algumas centenas de dimensões ou até menos) e capturam similaridades entre as palavras (o tipo de similaridade varia com o método usado para gerá-las) de acordo com sua distância euclidiana, ou outro tipo de cálculo de similaridade entre vetores (similaride do cosseno, similaridade de Jaccard, etc). Quando falamos que elas são similares, significa que elas tendem a serem usadas no mesmo contexto e usualmente pertencem a mesma classe gramatical. Em termos matemáticos significa que os vetores estão próximos um do outro. Além disso, palavras não vistas nos dados de treinamento não são completamente desconhecidas, pois elas tem uma representação vetorial. Por isso, espera-se que o impacto de palavras Fora do Vocabulário (FDV) seja menor (FONSECA; ROSA; ALUÍSIO, 2015).

As *word embeddings* podem então ser consideradas como novas *features* obtidas a partir das originais. Essa técnica tem interessado cada vez mais pesquisadores na área de PLN, e com isso novos processos de geração de *word embeddings* tem surgido.

Collobert et al. (2011) usa um modelo de rede neural (*Neural Language Model* (NLM), do inglês) para inicializar suas representações de palavras, com isso evita-se a tarefa de *engenharia de features*. Esse modelo é baseado na extração dos parâmetros aprendidos em uma rede neural treinada por *Backpropagation*, onde após o treinamento, é gerada uma tabela de busca com a palavra original como chave e o vetor de *features* como valor. Além disso, é mostrado a eficiência desse método em aplicações de análise sintática e semântica.

Collobert et al. (2011) ainda nos mostra como extender a ideia de representação

vetorial para incorporar *features* discretas. Um exemplo disso é a presença de capitalização, que traz informações importantes da palavra. Esse processo pode ser feito ao criar vetores correspondentes a *feature* criada. A Figura 11 ilustra esse processo.

Figura 11: Exemplo de vetores de *features* de cinco dimensões representando palavras

Vetores de features para palavras no vocabulário	
:	
leal:	0.41 0.54 0.49
leão:	0.19 0.33 0.01
lenha:	0.90 0.57 0.16
:	
Vetores de features para capitalização de palavras	
Todas minúsculas:	0.04 0.37
Todas maiúsculas:	0.98 0.20
Primeira maiúscula:	0.42 0.24
Outras combinações:	0.11 0.48
Nenhuma:	0.81 0.89
Representação final das palavras no modelo	
leal:	0.41 0.54 0.49 0.04 0.37
leão:	0.19 0.33 0.01 0.04 0.37
Leão:	0.19 0.33 0.01 0.42 0.24
LEÃO:	0.19 0.33 0.01 0.98 0.20

Adapatade de: Fonseca, Rosa e Aluísio (2015)

Outro método utilizado para a criação dos vetores é o *Hyperspace Analogue to Language* (HAL). Ele é baseado na contagem de coocorrência de palavras próximas umas das outras para então obter uma grande matriz de contagem. Para obter vetores a partir da matriz é utilizado um método de decomposição conhecido como Escalonamento Multidimensional (do inglês *Multidimensional scaling*) (LUND; BURGESS, 1996). Após isso, é possível verificar se duas palavras são semelhantes ao verificar a distância euclidiana entre os vetores delas.

Além dessas, outra estratégia para gerar vetores é a modelação *Skip-Gram* (SG), que tem como objetivo minimizar a complexidade computacional na geração das *word embeddings*. Essa estratégia consiste em prever palavras próximas de uma dada palavra. Isso é feito usando uma palavra como entrada para o um classificador de complexidade logarítmica com uma camada de projeção contínua. A previsão é feita com um conjunto de palavras predecessoras e sucessoras, porém quanto maior for esse conjunto, maior

será a complexidade computacional (MIKOLOV et al., 2013). Já que o classificador não tem uma camada oculta na sua arquitetura, ele é consideravelmente mais rápido que um modelo de rede neural. Essa estratégia é utilizada pela ferramenta *word2vec*, que contém a contribuição do próprio Mikolov et al. (2013).

A estratégia mais recente e indicada por Socher (2015) é a utilização de *Global Vectors* (GloVe) para representação de palavras. Essa estratégia consiste em criar os vetores de acordo com a razão das probabilidades na matriz de coocorrência em relação ao contexto de uma outra palavra no vocabulário. Além disso, ela segue a ideia da modelação SG ao utilizar um classificador de complexidade logarítmica. Essa estratégia é atualmente o estado da arte para a representação de palavras em vetores de *features* (PENNINGTON; SOCHER; MANNING, 2014).

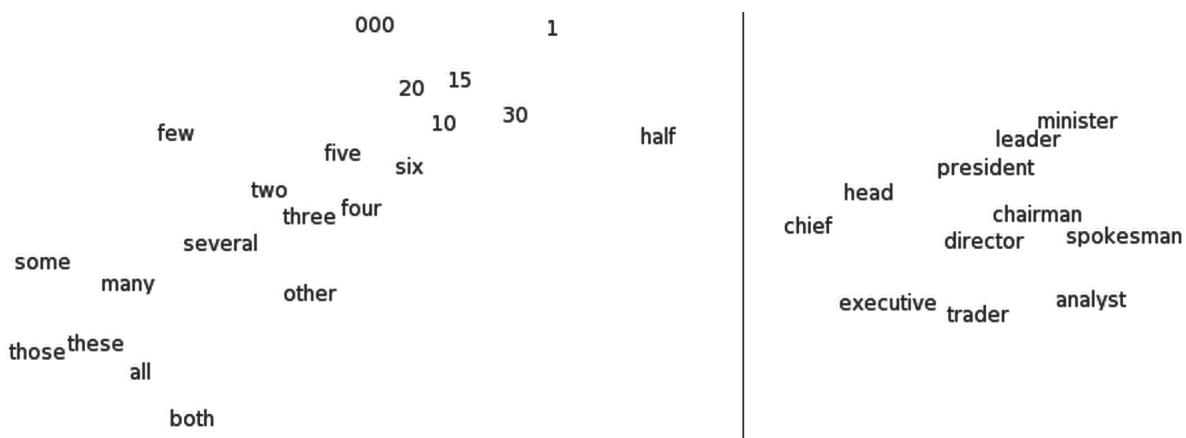
Portanto, para a criação de um vetor de *features*, temos a transformação W para uma palavra *word*:

$$W : \text{word} \rightarrow \mathbb{R}^n$$

O número de dimensões n dos vetores podem variar. Em geral, quanto mais dimensões há, melhores representações podem ser alcançadas, mas se a dimensão for muito grande, o processamento pode ser demorado.

Como já mencionado, a grande vantagem em se utilizar *word embeddings* é o fato dessa representação conseguir capturar informações sintáticas e semânticas das palavras. Isso pode ser visualizado através do t-SNE (MAATEN; HINTON, 2008), uma técnica sofisticada para visualizar dados em altas dimensões.

Figura 12: t-SNE: Visualização para *word embeddings*



Fonte: Turian, Ratinov e Bengio (2010)

Na Figura 12 podemos visualizar um tipo de mapeamento entre os sentidos intuitivos das palavras, onde na esquerda estão palavras referentes a números e na direita

palavras referentes a profissões. Ou seja, palavras similares estão pertoumas das outras.

Figura 13: Exemplo de palavras similares

FRANCE	JESUS	XBOX	REDDISH	SCRATCHED	MEGABITS
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	PSNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

Fonte: [Collobert et al. \(2011\)](#)

Na [Figura 13](#) temos mais alguns exemplos desse conceito, com isso podemos realizar tarefas semânticas mais facilmente. Como por exemplo: *homem + coroa = rei*, onde consegue-se realizar operações sobre os sentidos das palavras.

2.4 Redes neurais

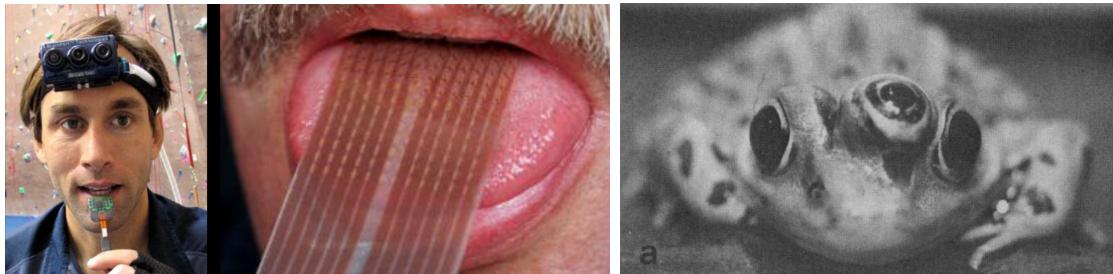
Redes neurais tem como origem algoritmos que tentam imitar o cérebro humano. Elas eram amplamente utilizadas na década de 80 e no começo da década de 90, porém sua popularidade diminuiu no fim dessa mesma década. Ressurgiram recentemente e são o estado da arte para várias aplicações de inteligência artificial ([NG, 2015](#)).

2.4.1 Neurocomputação

Há evidências de que o cérebro usa apenas um “algoritmo de aprendizagem” para todas diferentes funções. Cientistas já tentaram cortar (no cérebro de animais) a conexão entre as orelhas e o córtex auditivo e reconectar com o nervo óptico, o resultado foi que o córtex auditivo literalmente aprendeu a ver ([NG, 2015](#)).

Na [Figura 14\(a\)](#) os cientistas conseguiram fazer com que o ser humano possa enxergar através de sensores da língua conectado a uma câmera fotográfica. Já na [Figura 14\(b\)](#), foi implantado um terceiro olho em um sapo e após um tempo comprovou-se que o sapo definitivamente aprendeu a ver com aquele olho.

Figura 14: Exemplos de representação de sensores no cérebro

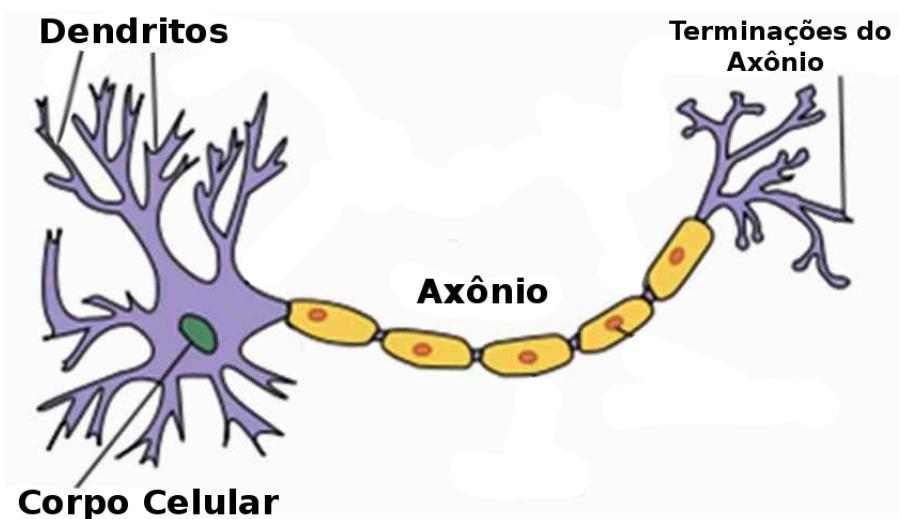


(a) Visualizando com a língua

(b) Implantando um terceiro olho

Fonte: Ng et al. (2013)

Figura 15: Representação de um neurônio



Fonte: Ng (2015)

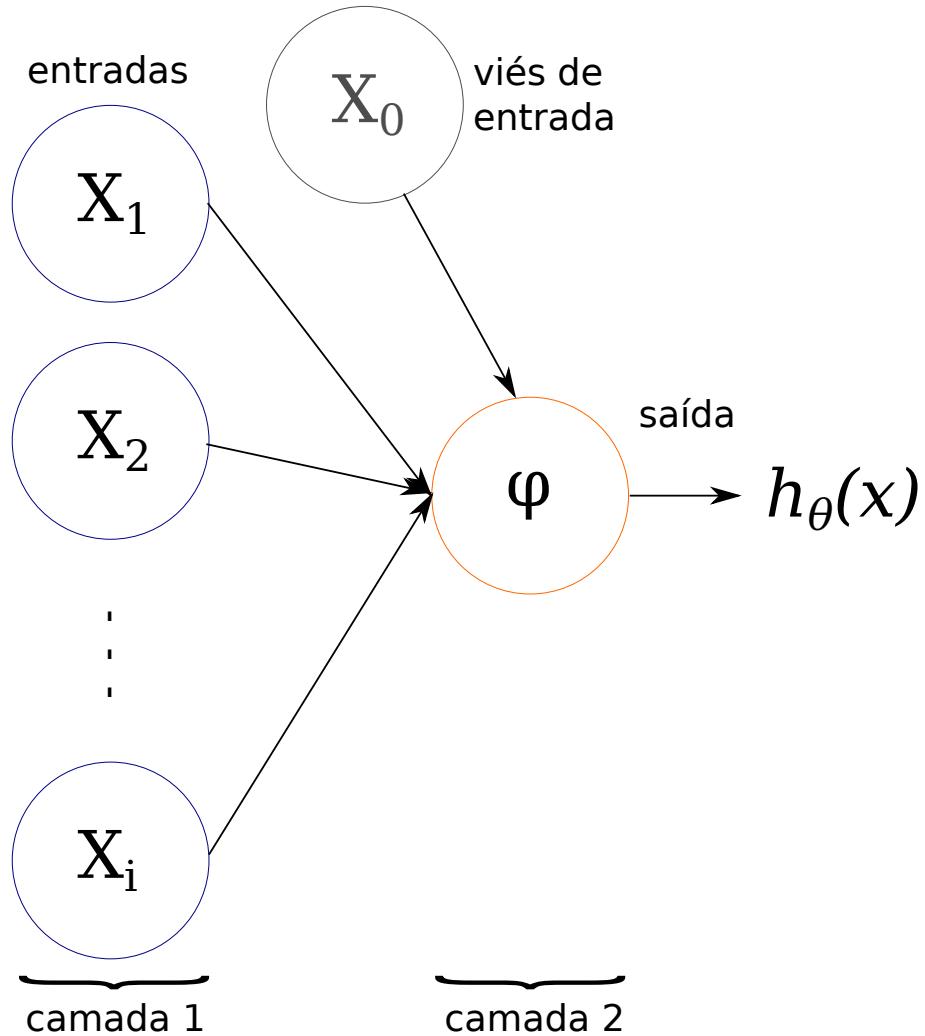
Esse princípio é conhecido como “neuroplasticidade” e tem vários exemplos e evidências de experimentos.

2.4.2 Representação do modelo

Em um nível bem simples de abstração, neurônios são basicamente unidades computacionais que recebem entradas (dendritos) como impulso elétrico que são canalizados para a saída (axônio), já o corpo celular é responsável por realizar os cálculos. Isso é ilustrado na [Figura 15](#).

A função de hipótese pode ser representada através de um modelo de redes neurais. Nesse modelo, o x_0 é usualmente chamado de viés de entrada ou unidade viés (do inglês *bias unit*) e ele sempre será igual a 1.

Figura 16: Abstração matemática de um neurônio



Em redes neurais, pode ser usada a mesma função logística como em regressão logística:

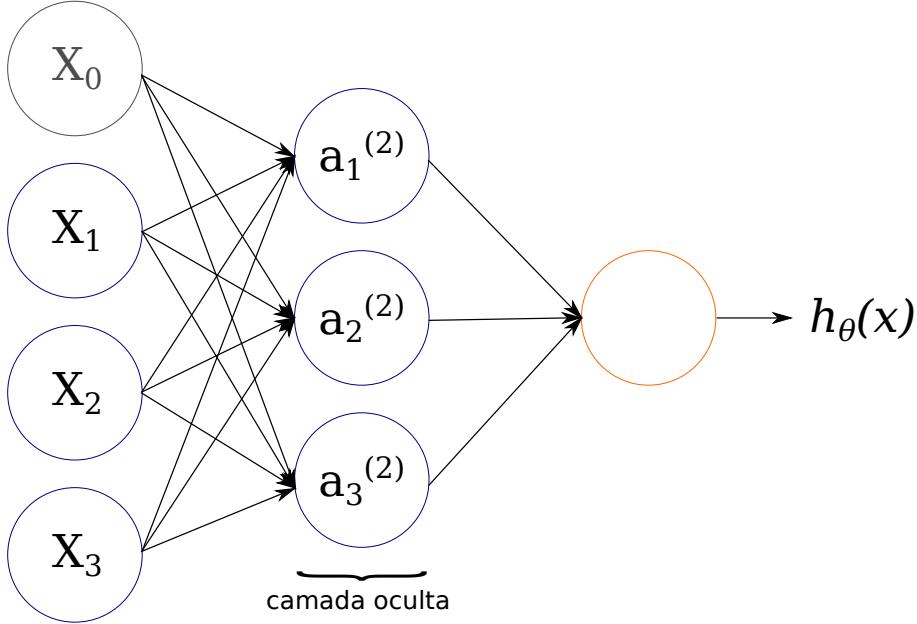
$$\frac{1}{1 + e^{\theta^T X}}$$

A diferença é que em redes neurais ela é usualmente conhecida como função de ativação sigmoide. Os parâmetros θ s a serem aprendidos são chamados de pesos. Visualmente, uma representação simples desse modelo se parece como a Figura 16. Os dados dos nodos de entrada (camada 1) vão para outro nodo (camada 2) que aplica a função de ativação, e então vão para a saída sendo a função de hipótese.

A primeira camada é chamada de “camada de entrada” e a camada final de “camada de saída”, que nos dá o valor final computado sobre a hipótese. É possível ter camadas intermediárias entre as camadas de entrada e saída chamadas de “camadas ocultas”.

Nomeamos os nodos das camadas intermediárias de $a_0^{(j)}, a_1^{(j)}, \dots, a_n^{(j)}$ e chamamo-

Figura 17: Rede neural com uma camada oculta



os de unidades de ativação. A tabela Tabela 3 reúne as notações geralmente usadas no contexto de redes neurais.

Tabela 3: Notação utilizada para representação em redes neurais

φ ou $g(z)$	função de ativação sigmoide
$a_i^{(j)}$	unidade de ativação i na camada j
$\theta^{(j)}$	matriz de pesos mapeando funções da camada j para camada $j + 1$
$z_k^{(j)}$	parâmetros da função g para a unidade de ativação k na camada j .

A Figura 17 mostra uma rede neural com uma camada oculta. Apesar do nodo viés não aparecer na camada oculta ele está presente nela, aliás, ele está presente e alimenta todos os nodos de todas camadas da rede (exceto a camada de saída), porém ele não recebe entradas.

O valor de cada nodo de ativação pode ser obtido pelo conjunto de equações 2.16:

$$\begin{aligned} a_1^{(2)} &= g(\theta_{1,0}^{(1)}x_0 + \theta_{1,1}^{(1)}x_1 + \theta_{1,2}^{(1)}x_2 + \theta_{1,3}^{(1)}x_3) \\ a_2^{(2)} &= g(\theta_{2,0}^{(1)}x_0 + \theta_{2,1}^{(1)}x_1 + \theta_{2,2}^{(1)}x_2 + \theta_{2,3}^{(1)}x_3) \\ a_3^{(2)} &= g(\theta_{3,0}^{(1)}x_0 + \theta_{3,1}^{(1)}x_1 + \theta_{3,2}^{(1)}x_2 + \theta_{3,3}^{(1)}x_3) \end{aligned} \quad (2.16)$$

E a função de hipótese é justamente a saída do único nodo de ativação na camada 3 (saída):

$$h_\theta(x) = g(\theta_{1,0}^{(2)}a_0^{(2)} + \theta_{1,1}^{(2)}a_1^{(2)} + \theta_{1,2}^{(2)}a_2^{(2)} + \theta_{1,3}^{(2)}a_3^{(2)})$$

Isso significa que estamos computando os nodos de ativação fazendo uma matriz de parâmetros com dimensão 3×4 . Aplicamos cada linha dos parâmetros para as entradas, obtendo assim o valor para um nodo de ativação. A saída da hipótese é a função logística aplicada a soma dos valores dos nodos de ativação da camada oculta, os quais foram multiplicados por outra matriz de parâmetros ($\theta^{(2)}$) contendo os pesos para a segunda camada de nodos.

Cada camada j tem sua própria matriz de pesos $\theta^{(j)}$. As dimensões dessas matrizes de pesos são determinadas pela regra 2.17.

$$\text{Se a rede tem } S_j \text{ unidades na camada } j \text{ e } S_{j+1} \text{ na camada } j+1, \text{ então } \theta^{(j)} \text{ terá a dimensão de } S_{j+1} \times (S_j + 1). \quad (2.17)$$

Isso significa que os nodos de saída não incluirão o viés de entrada, enquanto os de entrada sim. Essa regra é importante para ter uma versão vetorizada das redes neurais, pois facilita a codificação da mesma e também no entendimento de definições mais complexas.

2.4.3 Representação vetorizada do modelo

Em ordem de fazer uma implementação vetorizada das funções anteriores, será definido uma nova variável $z_k^{(j)}$ que engloba os parâmetros dentro da função g . Para o conjunto de equações 2.16 é possível simplificar a definição ao substituir os parâmetros pela variável z , como mostrado no conjunto de equações abaixo.

$$\begin{aligned} a_1^{(2)} &= g(z_1^{(2)}) \\ a_2^{(2)} &= g(z_2^{(2)}) \\ a_3^{(2)} &= g(z_3^{(2)}) \end{aligned}$$

Onde a definição de z é dada por:

$$z_k^{(j)} = \theta_{k,0}^{(j-1)}x_0 + \theta_{k,1}^{(j-1)}x_1 + \dots + \theta_{k,n}^{(j-1)}x_n$$

Levando em consideração que x é um vetor coluna com dimensão $(n+1)$, podemos calcular $z^{(j)}$ ao fazer a multiplicação matriz-vetor:

$$z^{(j)} = \theta^{(j-1)}x$$

Como sabemos que x é nosso vetor de entrada ($x = a^{(1)}$), podemos reescrever a equação como $z^{(j)} = \theta^{(j-1)}a^{(1)}$. Lembrando da regra 2.17, é possível analisar que a matriz

$\theta^{(j-1)}$ tem dimensões $S_j \times (n + 1)$, e portanto a multiplicação dessa matriz com o vetor x com altura $(n + 1)$, irá resultar no vetor z com altura S_j . Portanto, podemos obter um vetor de nodos de ativação para a camada j através da equação abaixo.

$$a^{(j)} = g(z^{(j)})$$

Onde a função g pode ser aplicada elementarmente no vetor $z^{(j)}$. Após o término da computação de $a^{(j)}$, é adicionado a unidade viés para a camada j . Onde essa unidade será o elemento $a_0^{(j)}$, que é por definição igual a 1.

Para computar as hipóteses finais, é necessário primeiro computar os outros vetores z das próximas camadas. Podemos fazer isso de acordo com a abaixo.

$$z^{(j+1)} = \theta^{(j)} a^{(j)}$$

A última matriz de pesos $\theta^{(j)}$ terá apenas uma linha, que fará com que o resultado seja apenas um escalar. Para que então seja possível obter o resultado final com:

$$h_\theta(x) = a^{(j+1)} = g(z^{(j+1)})$$

Ao adicionar todas as camadas intermediárias nas redes neurais, permite-se a produção de hipóteses não lineares mais elegantes e complexas. Ao chegar na hipótese final, conclui-se os passos que são conhecidos como *Forward Propagation*.

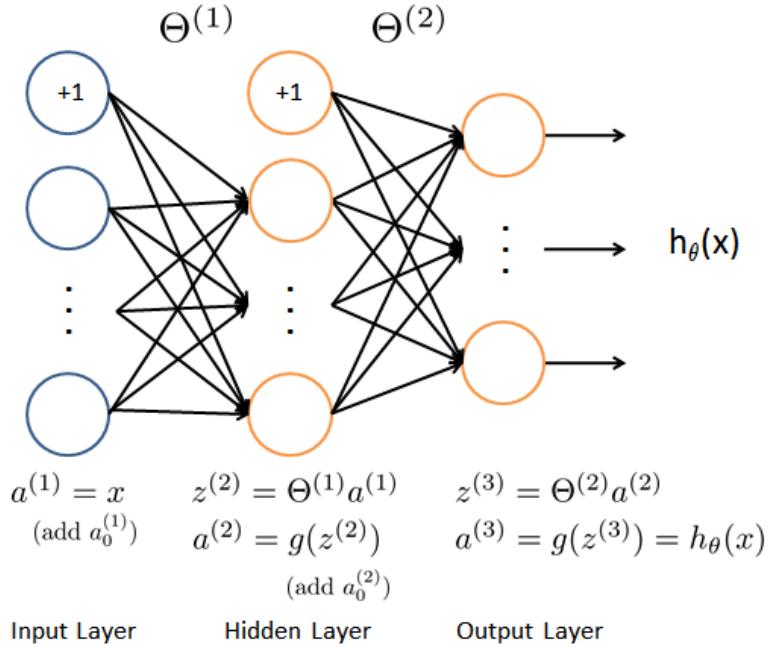
2.4.4 Classificação multiclasse

Para classificar dados em múltiplas classes, a função de hipótese deve retornar um vetor de valores. Quando a camada final for multiplicada por sua matriz θ , vai resultar em outro vetor, no qual será aplicado a função de ativação g para obter o vetor de hipóteses finais. Essa configuração pode ser vista na [Figura 18](#).

O vetor de hipóteses resultante para um conjunto de entradas pode parecer como:

$$h_\theta(x) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

No qual a classe resultante é a terceira linha, ou $h_\theta(x)_3$. É possível extender essa

Figura 18: Modelo de rede neural - Passos do *Forward Propagation*Fonte: [Ng \(2015\)](#)

definição para definir todo o conjunto de classes resultados como y' :

$$y^{(i)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

E o valor final da hipótese para um conjunto de entrada será um dos elementos em y .

2.4.5 Função de custo

As subseções anteriores mostraram como representar uma rede neural, agora estamos interessados em como realizar a aprendizagem para resolver o problema de classificação.

Antes de apresentar a função de custo para redes neurais, será definido algumas variáveis úteis nesse escopo. Elas estão na [Tabela 4](#).

Na [subseção 2.4.4](#) foi visto que é possível ter vários nodos de saída. É denotado $h_\theta(x)_k$ como sendo uma hipótese que resulta na k-ésima saída.

A função de custo para redes neurais é um pouco mais complicada que a utilizada na regressão logística, na verdade, ela é uma generalização da [Equação 2.14](#).

Tabela 4: Notação utilizada para aprendizado em redes neurais

L	número total de camadas na rede
S_l	número de unidades (sem o viés de entrada) na camada l
K	número de unidades de saída ou número de classes

Na [Equação 2.18](#), foi adicionado alguns somatórios aninhados para tratar os múltiplos nodos de saída. Na primeira parte da equação, entre os colchetes, foi adicionado um somatório aninhado que itera sobre o número de nodos de saída.

Na parte da regularização (após os colchetes), foi lidoado com as múltiplas matrizes θ s. Conforme a regra [2.17](#), o número de colunas de uma dada matriz θ é igual ao número de nodos na camada atual (incluindo o viés). Já o número de linhas da matriz θ é igual ao número de nodos na próxima camada (sem o viés).

$$J(\theta) = \frac{-1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_\theta(x^{(i)})_k) + (1 - y_k^{(i)}) \log(1 - h_\theta(x^{(i)})_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{S_l} \sum_{j=1}^{S_{l+1}} (\theta_{j,i}^l)^2 \quad (2.18)$$

2.4.6 Algoritmo *Backpropagation*

Em redes neurais, para realizar a minimização de custo um dos algoritmos amplamente utilizado é o *Backpropagation*. Seu objetivo é calcular as derivadas parciais da função de custo, para que depois seja possível realizar a operação: $\min_\theta J(\theta)$. Ou seja, o objetivo é minimizar a função de custo J usando um conjunto ótimo de parâmetros θ . E para isso é necessário o cálculo da derivada parcial da função de custo em relação aos parâmetros θ para cada nodo de ativação em uma camada l , conforme mostrado abaixo.

$$\frac{\partial}{\partial \theta_{i,j}^{(l)}} J(\theta)$$

Em *Backpropagation* é calculado o **erro** para cada nodo. Extendemos a notação usada pela [Tabela 4](#) ao adicionar uma nova variável $\delta_j^{(l)}$ que é igual ao “erro” do nodo j na camada l . Sendo assim, para a última camada, é possível calcular um vetor de valores δ com: $\delta^{(L)} = a^{(L)} - y$. Ou seja, os “valores de erro” para a última camada são simplesmente a diferença entre os resultados atuais na última camada e as saídas corretas em y .

A questão chave desse algoritmo é como obter os valores de δ nas camadas anteriores à última. Para isso é usado a [Equação 2.19](#) que nos leva para trás, da direita para

esquerda.

$$\delta^{(l)} = ((\theta^{(l)})^T \delta^{(l+1)}). * g'(z^{(l)}) \quad (2.19)$$

Os valores de δ da camada l são calculados ao multiplicar os valores de δ na próxima camada com a matriz θ na camada l . Isso é multiplicado elementarmente com uma função g' , a qual é a derivada da função de ativação g avaliada com os valores de entrada dados por $z^{(l)}$. A função g' pode ser descrita como na [Equação 2.20](#).

$$g'(z^{(l)}) = a^{(l)}. * (1 - a^{(l)}) \quad (2.20)$$

Fonte: [Ng \(2015\)](#)

Portanto, a equação completa de *Backpropagation* para os nodos interno é:

$$\delta^{(l)} = ((\theta^{(l)})^T \delta^{(l+1)}). * a^{(l)}. * (1 - a^{(l)})$$

Agora é possível computar a derivada parcial ao multiplicar os valores de ativação e os valores de erro para cada exemplo de treinamento t :

$$\frac{\partial}{\partial \theta_{i,j}^{(l)}} J(\theta) = \frac{1}{m} \sum_{t=1}^m a_j^{(t)(l)} \delta_i^{(t)(l+1)}$$

Isso porém ignora a regularização, que é feita depois.

Uma observação importante é que $\delta^{(l+1)}$ é um vetor com S_{l+1} elementos, e $a^{(l)}$ é um vetor com S_l elementos. Portanto a multiplicação deles vai gerar uma matriz com dimensão $S_{l+1} \times S_l$, que é a mesma dimensão que $\theta^{(l)}$. Ou seja, o processo irá produzir um termo gradiente para todos elementos em $\theta^{(l)}$. O algoritmo 1 junta todas essas equações.

A matriz Δ é usada apenas como um acumulador, com o intuito de passar adiante os valores enquanto é computado a derivada parcial. Os termos $D_{i,j}^{(l)}$ são as derivadas parciais finais, ou seja, os resultados esperados. A [Figura 19](#) ilustra os passos do algoritmo de *Backpropagation* para uma rede neural com uma camada oculta.

Segundo ([NG, 2015](#)), uma observação importante a ser feita é que os pesos θ não podem ser inicializados com zero, pois isso não funciona em redes neurais. Uma solução para esse problema é realizar uma **inicialização aleatória** dos pesos.

2.4.7 Treinamento

Para treinar a rede, é necessário primeiramente escolher uma arquitetura (*layout*) da rede neural, incluindo quantas unidades ocultas em cada camada e quantas camadas

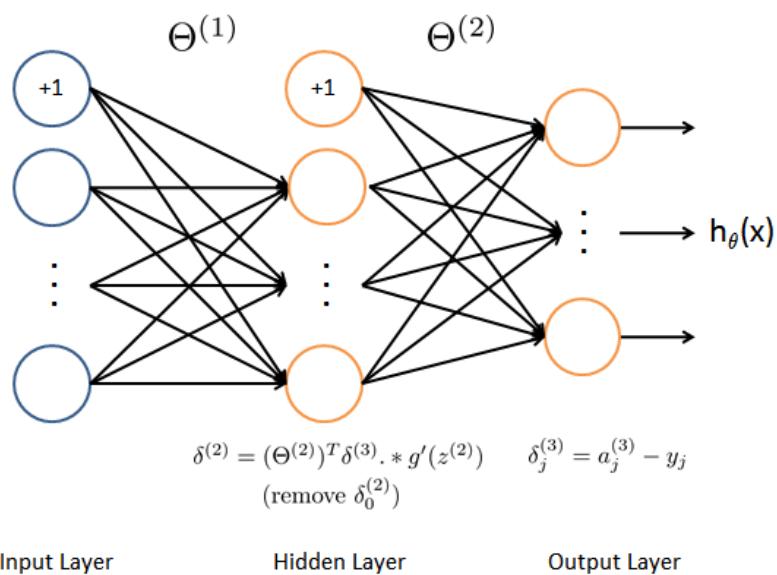
Algorithm 1 Algoritmo de Backpropagation

```

procedure BACKPROPAGATION
    dado um conjunto de treinamento  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$ 
     $\Delta_{i,j}^{(l)} = 0$  para todo  $(l, i, j)$ 
    for cada exemplo de treinamento  $t = 1$  até  $m$  do
         $a^{(1)} = x^{(t)}$ 
        Forward-Propagation $(a^{(2)})$ 
         $\delta^{(L)} = a^{(L)} - y^{(t)}$ 
        for  $l = L - 1$  até  $2$  do
             $\delta^{(l)} = ((\theta^{(l)})^T \delta^{(l+1)}). * a^{(l)}. * (1 - a^{(l)})$ 
             $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$ 
        end for
    end for
    if  $j = 0$  then
         $D_{i,j}^{(l)} = \frac{1}{m} \Delta^{(l)}$ 
    else
         $D_{i,j}^{(l)} = \frac{1}{m} (\Delta^{(l)} + \lambda \theta^{(l)})$ 
    end if
end procedure

```

Figura 19: Passos do Backpropagation



Fonte: Ng (2015)

no total.

- Número de unidades de entrada: igual ao número de dimensões das *features*;
- Número de unidades de saídas: igual ao número de classes;
- Número de camadas ocultas: por padrão é 1, caso seja mais que isso, então deve ter a mesma quantidade de unidades de ativação em cada camada oculta.

Para treinar a rede neural, é possível seguir os seguintes passos:

1. Inicializar aleatoriamente os pesos θ ;
2. Implementar o *Forward-propagation* para obter $h_\theta(x^{(i)})$;
3. Implementar a função de custo;
4. Implementar o *Backpropagation* para computar as derivadas parciais;
5. Usar o Gradiente Descendente ou outro algoritmo de otimização para minimizar a função de custo com os pesos θ .

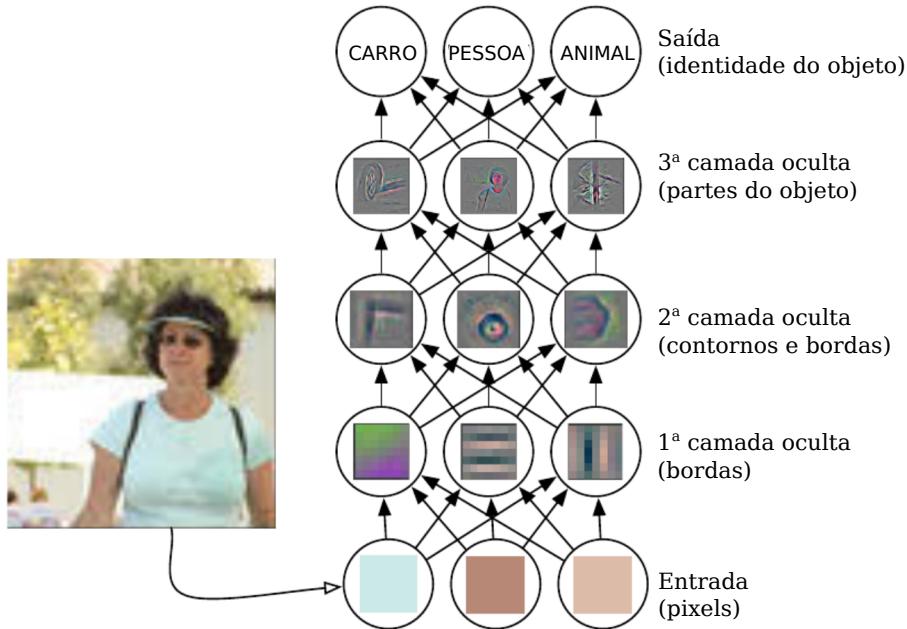
2.5 Aprendizagem profunda

A aprendizagem profunda é uma ramificação da aprendizagem de máquina, que baseia-se num conjunto de algoritmos que procuram modelar abstrações de alto nível usando modelos arquitetoriais, com estruturas complexas, compostas de múltiplas transformações não lineares (DENG; YU, 2014). Suas definições são intrínsecas, pois não há um consenso de quão profundo um modelo precisa ser para ser considerado um modelo de aprendizagem profunda. Entretanto, há a certeza de que aprendizagem profunda pode ser seguramente categorizada como o estudo de modelos que tratam uma grande quantidade de composições de aprendizagem ou conceitos de aprendizagem que não conseguem ser empregados em aprendizado de máquina (BENGIO; GOODFELLOW; COURVILLE, 2015).

Extrair *features* significantes manualmente é uma tarefa muito difícil, pois pode haver muitas variações nos dados que podem ser identificadas utilizando um nível sofisticado, quase humano, de entendimento. Quando é assim tão difícil de obter uma representação como de resolver o problema principal, a aprendizagem de representação parece não ajudar.

Aprendizagem profunda resolve esse problema de representação ao introduzir representações que são expressas em termos de representações mais simples. Permitindo o

Figura 20: Modelo de aprendizagem profunda



Adaptado de: [Bengio, Goodfellow e Courville \(2015\)](#)

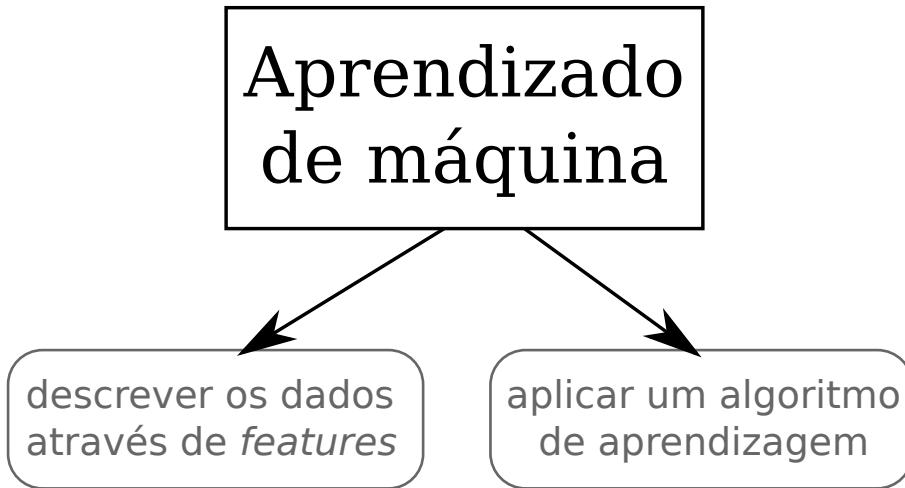
computador a construir conceitos complexos a partir de conceitos simples. A Figura 20 ilustra um modelo de aprendizagem profunda.

Na Figura 20, cada *pixel* da imagem são *features* passadas para a primeira camada oculta, que consegue identificar bordas facilmente ao comparar o brilho de *pixels* vizinhos. Ao passar a descrição dessas bordas para a segunda camada oculta, ela consegue facilmente procurar por contornos, que são reconhecidos como uma coleção de bordas. Ao chegar na terceira camada oculta, consegue-se detectar partes inteiras de objetos específicos ao procurar por específicas coleções de contornos e bordas. No final, a descrição da imagem em termos das partes dos objetos pode ser usada para reconhecer objetos inteiros presentes na imagem ([BENGIO; GOODFELLOW; COURVILLE, 2015](#)).

A Figura 21 mostra que o aprendizado de máquina se resume em otimizar pesos para fazer uma ótima predição final. E para isso, é necessário dois itens: Descrever os dados através de *features* de modo que o computador consiga entender; Aplicar um algoritmo de aprendizagem. Para o primeiro item, é necessário conhecimento do domínio específico, o que acaba gerando o chamado *engenharia de features*, que ultimamente não é bem visto pela comunidade de [PLN](#). O segundo item consiste em otimizar os pesos de acordo com as *features*, e qualquer bom algoritmo de minimização da função de custo pode ser utilizado.

A aprendizagem profunda tenta automaticamente aprender boas *features* ou representações. Na seção 2.3 foi mostrado o conceito de *word embeddings* ou vetor de

Figura 21: Sentido do aprendizado de máquina



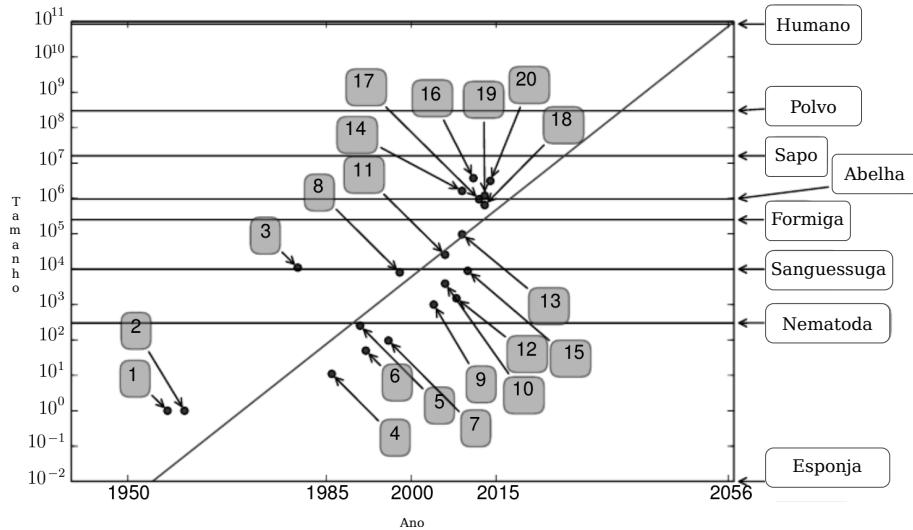
features, na verdade esse conceito está fortemente associado ao aprendizado profundo, onde tenta-se aprender múltiplos níveis da representação. Especificamente, é um tipo de aprendizado de máquina, uma técnica que permite sistemas computacionais à melhorarem com experiência e dados (BENGIO; GOODFELLOW; COURVILLE, 2015).

O modelo dominante em aprendizado profundo é redes neurais. Nesse modelo, as *features* são aprendidas rapidamente e automaticamente, adaptando-se muito bem as tarefas de PLN. Com isso, é possível aprender representações das informações linguísticas de modo não supervisionado (a partir do texto fonte) e também de modo supervisionado (usando classes específicas). Ela começou a ser usada recentemente devido a criação de novos modelos, algoritmos e ideias, e também devido ao aumento do desempenho computacional (SOCHER, 2015).

Como o processo de criação automatizada de *features* acaba gerando muitos vetores de representações, temos que criar um modelo capaz de trabalhar com essas informações com o objetivo de aprender. Por isso, geralmente se trabalha com redes neurais com múltiplas camadas. A Figura 22 demonstra o aumento do tamanho das redes neurais de acordo com o tempo.

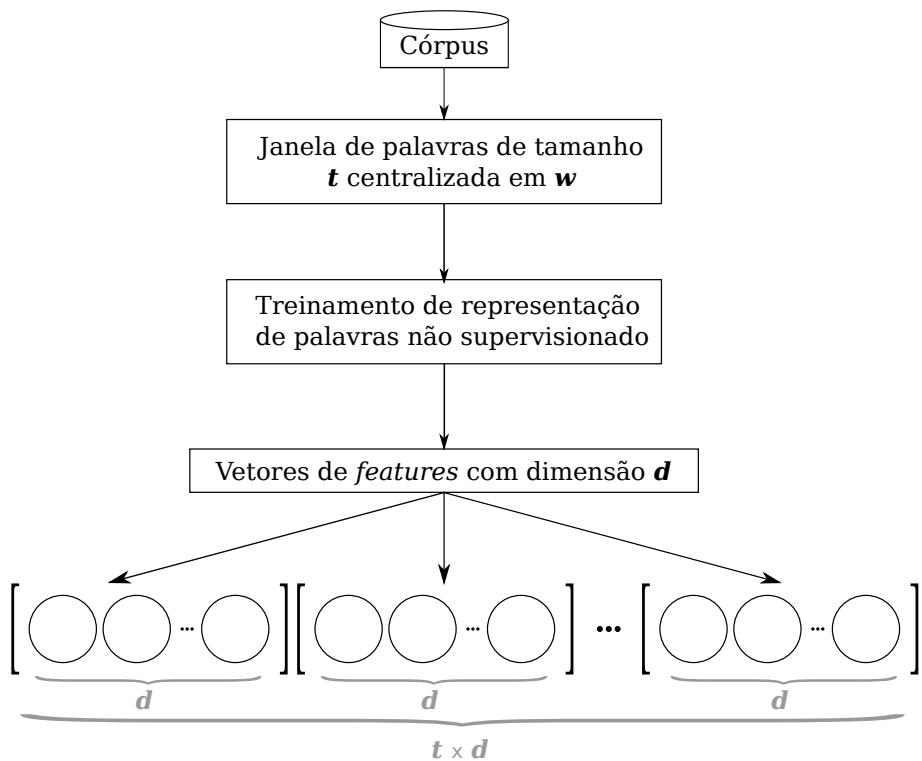
A técnica utilizada ultimamente em POS Tagging está ilustrada na Figura 23. Que consiste em primeiramente obter a representação das palavras em vetores reais com dimensões definidas pelo usuário, e após isso mesclar com *features* de formato das palavras (e.g capitalização), para então jogar esses vetores como entrada numa rede neural. O que muda em cada abordagem geralmente é a questão do treinamento da representação das palavras, as novas *features* criadas e como é organizado o modelo da rede neural.

Figura 22: Tamanho das redes neurais ao longo dos anos



Adaptado de: [Bengio, Goodfellow e Courville \(2015\)](#)

Figura 23: Funcionamento de aprendizagem profunda em POS Tagging



3 Trabalhos relacionados

Vários métodos já foram propostos para resolver esse mesmo problema em português brasileiro, apesar de nenhum deles ter um aproveitamento de 100%, vários conseguiram ótimos resultados e utilizaram variadas técnicas para isso.

É apresentado em (KEPLER, 2005) um etiquetador morfossintático baseado em cadeias de Markov. É realizado testes com dois etiquetadores diferentes, um baseado em Cadeias de Markov Ocultas (HMM, do inglês *Hidden Markov Models*), e outro baseado em Cadeias de Markov de Tamanho variável (VLMC, do inglês *Variable Length Markov Chains*). A representação das palavras é feita de forma simples, pois a etiquetagem baseia-se em modelos probabilísticos, onde a etiqueta de uma palavra depende da própria palavra e de etiquetas anteriores. Ele é testado sobre o *córpus* Tycho Brahe, e apresenta uma precisão de 95,51% com o etiquetador VLMC, essa precisão é alcançada com um tempo de aprendizagem + etiquetagem de 157 segundos.

Em (SANTOS; ZADROZNY, 2014) é apresentado um etiquetador que aprende automaticamente as features a serem usadas através de uma rede neural profunda, que emprega uma camada evolutiva capaz de aprender *embeddings* em nível de caractere e em nível de palavra. Essa rede neural profunda é conhecida como CharWNN e foi proposta originalmente por Collobert et al. (2011). Além disso, é usado um modelo em janela utilizado em (COLLOBERT et al., 2011) para atribuir classes gramaticais para cada palavra em uma sentença. Essa estratégia assume que a classe de uma palavra depende geralmente das palavras vizinhas. No fim é utilizado o algoritmo de Viterbi (VITERBI, 1967) para prever qual a sequência de classes gramaticais é a mais provável para aquela sentença. Esse trabalho usa três diferentes *córpus* para o treinamento: o Mac-Morpho original; o Mac-Morpho (versão 2) revisado em (FONSECA; ROSA, 2013); e o Tycho Brahe. Eles avaliam seu modelo sobre palavras fora do vocabulário e sobre palavras presentes no vocabulário. Com o Mac-Morpho foi obtido o melhor desempenho do trabalho, uma acurácia de 97,47%. Nesse trabalho não é mostrado estatísticas de tempo de treinamento e etiquetagem.

O mais recente etiquetador para o português brasileiro é mostrado em (FONSECA; ROSA; ALUÍSIO, 2015), onde é utilizado diferentes técnicas de representação das palavras: vetores gerados de forma aleatória, NLM, HAL, SG. É feita então uma comparação entre elas. Nele é implementado um modelo de rede neural idealizado em (COLLOBERT; WESTON, 2008), que se baseia em uma rede neural simples com múltiplas camadas, que recebe as *word embeddings* como entrada e aprende a sua classe gramatical. Para isso, eles realizam o treinamento utilizando o Tycho Brahe, a versão original do Mac-Morpho

(versão 1), a revisada pelos mesmos autores em um trabalho anterior ([FONSECA; ROSA, 2013](#)) (versão 2), e também sobre mais uma versão do Mac-Morpho (versão 3) revisado por eles nesse mesmo trabalho. Atualmente, [Fonseca, Rosa e Aluísio \(2015\)](#) dizem ter alcançado o estado da arte do POS Tagging para o português brasileiro, com uma acurácia de 97,57% sobre o Mac-Morpho original utilizando [NLM](#) como representação das palavras. Esse trabalho não apresenta estatísticas de tempo de treinamento e etiquetagem.

A [Tabela 5](#) sumarizada as técnicas encontradas pelos trabalhos relacionados. Já a [Tabela 6](#) mostra os melhores resultados para cada *córpus*.

Tabela 5: Comparativo das técnicas encontradas na literatura para POS Tagging

Autores	Modelo	Representação das palavras	Córpus
Kepler (2005)	VLMM	Sequência de caracteres	Tycho Brahe
Santos e Zadrozny (2014)	Redes neurais profundas	Vetores (CharWNN)	Tycho Brahe; Mac-Morpho (versões 1, 2)
Fonseca, Rosa e Aluísio (2015)	Redes neurais	Vetores (NLM , HAL , SG)	Tycho Brahe; Mac-Morpho (versões 1, 2, 3)
Este trabalho	Redes neurais recursivas	Vetores (NLM , SG , GloVe)	Tycho Brahe; Mac-Morpho (versões 1, 3)

Tabela 6: Comparativo dos melhores resultados encontrados na literatura para POS Tagging

Autores	Córpus	Acurácia (Todas Palavras)	Acurácia (FDV)
Kepler (2005)	Tycho Brahe	95,51%	69,53%
Santos e Zadrozny (2014)	Tycho Brahe	97,17%	86,63%
Santos e Zadrozny (2014)	Mac-Morpho versão 1	97,47%	89,74%
Santos e Zadrozny (2014)	Mac-Morpho versão 2	97,31%	92,61%
Fonseca, Rosa e Aluísio (2015) ¹	Tycho Brahe	96,91%	84,14%
Fonseca, Rosa e Aluísio (2015) ¹	Mac-Morpho versão 1	97,57%	93,38%
Fonseca, Rosa e Aluísio (2015) ¹	Mac-Morpho versão 2	97,48%	94,34%
Fonseca, Rosa e Aluísio (2015) ¹	Mac-Morpho versão 3	97,33%	93,66%

¹ Usando [NLM](#) como representação das palavras.

4 Metodologia

Neste capítulo será explicado o método proposto para resolver o problema de **POS** Tagging, onde primeiramente será definida as técnicas utilizadas para seu funcionamento.

Com a intuição de simplificar o entendimento, já definimos variáveis que serão utilizadas no método de aprendizagem. Elas podem ser encontradas na [Tabela 7](#).

Tabela 7: Notação utilizada para o treinamento do modelo

d	dimensão das palavras vetorizadas
t	tamanho da janela de palavras
ω	conjunto de palavras
γ	conjunto de classes gramaticais
c_i^t	sequência de classes gramaticais que começa em i e termina em t
w_i^t	sequência de palavras que começa em i e termina em t
c_i	i-ésima classe gramatical
z_i	i-ésima classe vetorizada com dimensão d
w_i	i-ésima palavra
v_i	i-ésima palavra vetorizada com dimensão d
V_i	vetor centrado em i com as t palavras vetorizadas concatenadas
$s_c(V_i)$	pontuação de uma sequência V_i ter a classe gramatical c
$A_{c,d,e}$	pontuação de ir de uma classe c , passar por d e chegar a e
Q	conjunto com os índices das palavras vetorizadas já classificadas

4.1 Representação das palavras

Seguimos a ideia explorada em massa pela literatura de representar as palavras através de vetores reais com uma dimensão fixa d definida pelo usuário. Isso será feito utilizando três estratégias já mencionadas: [NLM](#), [SG](#) e [GloVe](#). Ou seja, para cada $w_i \in \omega$, geramos o vetor $v_i \in \mathbb{R}^d$.

Além das *word embeddings*, utilizaremos outras duas *features* importantes no contexto de **POS** Tagging, como capitalização que consegue, na maioria das vezes, distinguir nomeações, e também prefixos, que podem ser usadas para distinguir medidas de tempo, velocidade, etc.

Em ordem de manter a rede neural homogênea, também iremos transformar cada classe grammatical em um vetor. Com isso poderemos aplicar funções que combinam características do vetor de uma palavra com o vetor de uma classe grammatical.

4.2 Pontuações para estrutura gramatical

Para classificar palavras em uma sentença, o etiquetador obtém uma janela de palavras de tamanho fixo a cada momento, e transforma as palavras em vetores de *features*, que são então passadas para uma rede neural descrita em (COLLOBERT; WESTON, 2008). A rede atribui para cada classe gramatical $c \in \gamma$ uma pontuação. A etapa de gerar pontuações para a palavra ocorre no mesmo momento que o treinamento do modelo. A saída para toda a sentença é então passada para o algoritmo de Viterbi (VITERBI, 1967), que realiza uma predição estruturada em um tempo polinomial.

Seguimos a estratégia apresentada em (SANTOS; ZADROZNY, 2014) para realizar as pontuações. Nela, a ideia é de que a classe gramatical de uma palavra depende fortemente das palavras vizinhas, o que é verdade para várias aplicações de PLN, incluindo POS Tagging.

Dada uma janela com t palavras $\{w_1, w_2, \dots, w_t\}$, que foram transformadas para sua representação vetorial $\{v_1, v_2, \dots, v_t\}$, para computar a n -ésima palavra, é centralizada a janela em n e concatena-se todas as palavras da metade à esquerda e da metade à direita em um novo vetor V_n de dimensão $t * d$. Para palavras no começo ou no fim da sentença, é usado vetores com valores fixos para preencher o espaço vazio na janela de palavras. A Equação 4.1 demonstra isso.

$$V_n = \{v_{n-(t-1)/2}, \dots, v_n, \dots, v_{n+(t-1)/2}\} \quad (4.1)$$

Seguimos (FONSECA; ROSA; ALUÍSIO, 2015) e computamos a pontuação $s_c(V_n)$ para cada classe gramatical c da palavra no meio da janela.

Além disso, usamos a ideia apresentada em (COLLOBERT et al., 2011), onde é feito um esquema de predição que leva em consideração a estrutura gramatical. O método usa uma pontuação de transição $A_{c,d}$, inicializada com 0, para ir de uma classe $c \in \gamma$ para uma classe $d \in \gamma$ conforme a sequência das palavras. Porém extendemos a notação para funcionar com trigramas: $A_{c,d,e}$. A estrutura A consegue armazenar informações importantes como “após um pronome é bastante provável que há um verbo”. Depois que a rede produz a pontuação para todas as palavras, a pontuação final de uma sequência de classes gramaticais c_1^t para uma sequência de palavras w_1^t , é dada pela Equação 4.2. Q representa um conjunto com os índices das palavras que já foram classificadas.

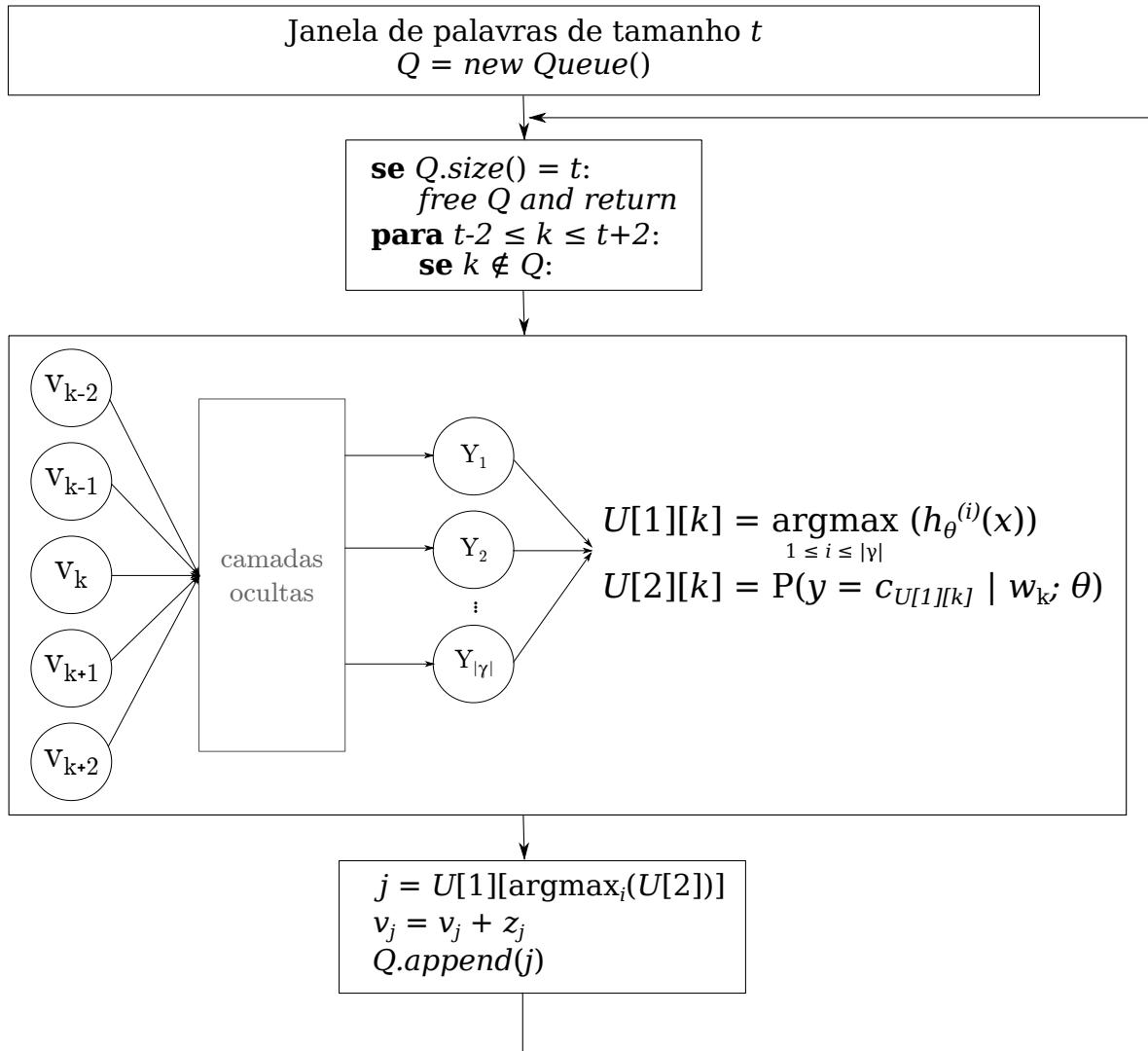
$$S(w_1^t, c_1^t) = \sum_{k=1}^t \left(\arg \max_{1 \leq i \leq t, v_i \notin Q} (s_{c_i}(V_i) + A_{c_{i-1}, c_i, c_{i+1}}) \right) \quad (4.2)$$

Após computar isso para cada palavra na sentença, a classe gramatical final é prevista através do algoritmo de Viterbi.

4.3 Treinamento

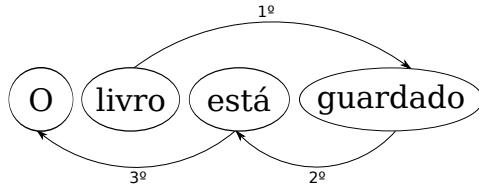
Realizamos um treinamento supervisionado utilizando uma rede neural recursiva para a classificação das palavras em suas respectivas classes gramaticais. O modelo completo da rede neural pode ser visto na [Figura 24](#).

Figura 24: Modelo da rede neural



Para treinar a rede neural, será seguido uma estratégia de aprendizagem guiada por palavras mais fáceis ([SHEN; SATTA; JOSHI, 2007](#)). A [Figura 25](#) exemplifica transições mais fáceis em uma frase, onde o peso mais baixo na aresta representa a ordem de execução. Na verdade, cada transição de um nodo para outro conta como uma nova etapa de treinamento. A [Equação 4.2](#) faz esse aprendizado guiado ao realizar a escolha da palavra mais fácil através da maximização das possibilidades que ainda não foram testadas dentro da janela de palavras.

Figura 25: Grafo de transições de palavras mais fáceis



Treinar o modelo consiste em ajustar os pesos da rede neural, os valores das *word embeddings* e as pontuações de transição. Após descobrir qual a classe gramatical mais provável para uma palavra w_n , o vetor da classe gramatical z_c é composto com o vetor da própria palavra v_n através de uma operação de soma. Essa composição é então armazenada no vetor da própria palavra, conforme mostrado abaixo.

$$v_n = v_n + z_n \quad (4.3)$$

Na rede neural da [Figura 24](#), as camadas ocultas foram removidas da imagem para simplificar a visualização. A saída da rede neural é armazenada num conjunto temporário U , que tem duas listas. A primeira lista tem os índices da classe preditada para a palavra sendo analisada w_k , já a segunda lista contém as probabilidades da classe preditada ser da palavra w_k . Após isso, recupera-se qual a palavra que tem a maior probabilidade de ser classificada. O vetor da classe da palavra recuperada z_j é composto com o vetor da palavra analisada v_j através da [Equação 4.3](#) e o processo se repete até que o conjunto Q tenha tamanho t .

Para treinar a rede neural, todos os ajustamentos são feitos em ordem de maximizar a seguinte equação:

$$\sum_{(w_1^t, c_1^t) \in \phi} P(c_1^t | w_1^t, \theta)$$

Onde ϕ denota o conjunto dos pares de palavras e classes gramaticais. Computamos a probabilidade acima utilizando uma operação *softmax* ([BENGIO; GOODFELLOW; COURVILLE, 2015](#)):

$$P(c_1^t | w_1^t, \theta) = \frac{e^{S(w_1^t, c_1^t)}}{\sum_{u_1^t \in \gamma^t} e^{S(w_1^t, u_1^t)}}$$

$$\log(P(c_1^t | w_1^t, \theta)) = S(w_1^t, c_1^t) - \log\left(\sum_{u_1^t \in \gamma^t} e^{S(w_1^t, u_1^t)}\right)$$

A função de custo pode ser definida como:

$$J(\theta) = \log \left(\sum_{u_1^t \in \gamma^t} e^{S(w_1^t, u_1^t)} \right) - S(w_1^t, c_1^t) \quad (4.4)$$

Onde, há princípio, ajustamos os pesos θ da rede utilizando o Gradiente Descendente sobre o primeiro termo da função de custo, porém vamos testar com outros algoritmos mais eficientes para essa tarefa, como o Gradiente Descendente Estocástico, Adagrad, Adadelta, etc. (BENGIO; GOODFELLOW; COURVILLE, 2015).

Também queremos maximizar o segundo termo da Equação 4.4. Para isso, seguimos (FONSECA; ROSA; ALUÍSIO, 2015) e realizamos um incremento nas pontuações de transição para cada palavra etiquetada em cada etapa do *Backpropagation*, conforme mostrado na equação abaixo:

$$\frac{\partial J(\theta)}{\partial A_{c_{i-1}, c_i, c_{i+1}}} += 1 \quad \forall i \quad (4.5)$$

Além disso, incrementamos a função $s_c(v_k) += 1$, onde v_k representa a palavra no meio da janela sendo analisada.

5 Cronograma de atividades

Nesse capítulo, é apresentado o quadro de tarefas previstas até a conclusão deste trabalho, juntamente com o período de tempo previsto para a realização das atividades.

A1 - Implementação do modelo neural recursivo;

A2 - Treinamento do modelo;

A3 - Avaliação dos resultados obtidos;

A4 - Escrita da monografia.

Tabela 8: Cronograma de atividades restantes

	Agosto	Setembro	Outubro	Novembro	Dezembro
A1	X	X	X		
A2		X	X	X	
A3			X	X	X
A4				X	X

Referências

- AFONSO, S. et al. Floresta sintá (c) tica: A treebank for portuguese. In: *LREC*. [S.l.: s.n.], 2002. Citado na página 33.
- ALUÍSIO, S. et al. An account of the challenge of tagging a reference corpus for brazilian portuguese. In: *Computational Processing of the Portuguese Language*. [S.l.]: Springer, 2003. p. 110–117. Citado na página 33.
- BENGIO, Y.; GOODFELLOW, I. J.; COURVILLE, A. Deep learning. Book in preparation for MIT Press. 2015. Disponível em: <<http://www.iro.umontreal.ca/~bengioy/dlbook>>. Citado 6 vezes nas páginas 47, 48, 49, 50, 56 e 57.
- COLLOBERT, R. Deep learning for efficient discriminative parsing. In: *International Conference on Artificial Intelligence and Statistics*. [S.l.: s.n.], 2011. Citado na página 17.
- COLLOBERT, R.; WESTON, J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In: ACM. *Proceedings of the 25th international conference on Machine learning*. [S.l.], 2008. p. 160–167. Citado 2 vezes nas páginas 51 e 54.
- COLLOBERT, R. et al. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, JMLR. org, v. 12, p. 2493–2537, 2011. Citado 4 vezes nas páginas 34, 37, 51 e 54.
- DENG, L.; YU, D. Deep learning: methods and applications. *Foundations and Trends in Signal Processing*, Now Publishers Inc., v. 7, n. 3–4, p. 197–387, 2014. Citado na página 47.
- FONSECA, E. R.; ROSA, J. L. G. Mac-morpho revisited: Towards robust part-of-speech tagging. In: *Proceedings of the 9th Brazilian Symposium in Information and Human Language Technology*. [S.l.: s.n.], 2013. p. 98–107. Citado 3 vezes nas páginas 32, 51 e 52.
- FONSECA, E. R.; ROSA, J. L. G.; ALUÍSIO, S. M. Evaluating word embeddings and a revised corpus for part-of-speech tagging in portuguese. *Journal of the Brazilian Computer Society*, Springer, v. 21, n. 1, p. 1–14, 2015. Citado 9 vezes nas páginas 17, 32, 33, 34, 35, 51, 52, 54 e 57.
- JR, D. W. H.; LEMESHOW, S. *Applied logistic regression*. [S.l.]: John Wiley & Sons, 2004. Citado na página 22.
- KEPLER, F. N. *Um etiquetador morfo-sintático baseado em cadeias de Markov de tamanho variável*. Tese (Doutorado) — Instituto de Matemática e Estatística da Universidade de São Paulo, 12/04/2005., 2005. Citado 2 vezes nas páginas 51 e 52.
- LUND, K.; BURGESS, C. Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instruments, & Computers*, Springer, v. 28, n. 2, p. 203–208, 1996. Citado na página 35.

- MAATEN, L. Van der; HINTON, G. Visualizing data using t-sne. *Journal of Machine Learning Research*, v. 9, n. 2579-2605, p. 85, 2008. Citado na página 36.
- MANNING, C. D.; SCHÜTZE, H. *Foundations of statistical natural language processing*. [S.l.]: MIT press, 1999. Citado na página 17.
- MARQUIAFÁVEL, V. S. Um processo para a geração de recursos lingüísticos aplicáveis em ferramentas de auxílio à escrita científica. Biblioteca Digital de Teses e Dissertações da Universidade Federal de São Carlos, 2010. Citado na página 17.
- MICHALSKI, R. S.; CARBONELL, J. G.; MITCHELL, T. M. *Machine learning: An artificial intelligence approach*. [S.l.]: Springer Science & Business Media, 2013. Citado na página 28.
- MIKOLOV, T. et al. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013. Citado na página 36.
- NG, A. *Course of Machine Learning*. [S.l.], 2015. Disponível em: <<https://www.coursera.org/learn/machine-learning/>>. Citado 11 vezes nas páginas 22, 23, 28, 29, 30, 31, 37, 38, 43, 45 e 46.
- NG, A. et al. *Unsupervised feature learning and deep learning*. [S.l.]: Stanford, Tutorial, 2013. Citado na página 38.
- PENNINGTON, J.; SOCHER, R.; MANNING, C. D. Glove: Global vectors for word representation. *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP 2014)*, v. 12, p. 1532–1543, 2014. Citado na página 36.
- PENNSYLVANIA, U. of. *The Penn Treebank Project (2014)*. 2014. Disponível em: <<http://www.cis.upenn.edu/~treebank/>>. Citado na página 32.
- SANTOS, C. N. dos; ZADROZNY, B. Training state-of-the-art portuguese pos taggers without handcrafted features. In: *Computational Processing of the Portuguese Language*. [S.l.]: Springer, 2014. p. 82–93. Citado 4 vezes nas páginas 17, 51, 52 e 54.
- SHEN, L.; SATTA, G.; JOSHI, A. Guided learning for bidirectional sequence classification. In: CITESEER. *ACL*. [S.l.], 2007. v. 7, p. 760–767. Citado na página 55.
- SOCHER, R. *Deep Learning for Natural Language Processing*. 2015. Disponível em: <<http://cs224d.stanford.edu/>>. Citado 3 vezes nas páginas 33, 36 e 49.
- TURIAN, J.; RATINOV, L.; BENGIO, Y. Word representations: a simple and general method for semi-supervised learning. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. *Proceedings of the 48th annual meeting of the association for computational linguistics*. [S.l.], 2010. p. 384–394. Citado 2 vezes nas páginas 33 e 36.
- UNICAMP. *Tycho Brahe Parsed Corpus of Historical Portuguese*. 2010. Disponível em: <<http://www.tycho.iel.unicamp.br/~tycho/corpus/en/index.html>>. Citado na página 33.
- VITERBI, A. J. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, IEEE, v. 13, n. 2, p. 260–269, 1967. Citado 2 vezes nas páginas 51 e 54.