

Universidade Federal do Pampa

Marcos Vinícius Treviso

**Dois modelos de aprendizagem profunda para  
análise morfossintática**

Alegrete

2015



Marcos Vinícius Treviso

## **Dois modelos de aprendizagem profunda para análise morfosintática**

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Fábio Natanael Kepler

Alegrete

2015



Marcos Vinícius Treviso

## Dois modelos de aprendizagem profunda para análise morfosintática

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

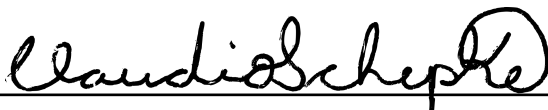
Trabalho de Conclusão de Curso defendido e aprovado em ..... de ..... de .....

Banca examinadora:



---

Fábio Natanael Kepler  
Orientador



---

Claudio Schepke  
UNIPAMPA



---

Marcelo Resende Thielo  
UNIPAMPA



# Agradecimentos

O esforço para chegar até aqui é grande, e certamente não conseguiria ter feito isso sozinho. Por isso gostaria de agradecer meu orientador Fabio por me ajudar em diversos assuntos durante a faculdade (principalmente com o TCC), com o qual aprendi muito.

Gostaria de agradecer todo o pessoal lá de casa (ainda é estranho falar casa) que moram comigo: Fabrício do SIEPE, Jair do caminhão, Jonnathan da nova geração (ou seria Wesley safadão?), Willian das cocotas, Abner dos salgadinhos, e é claro, a dona Rose que me acolheu como um filho ;-). Eles sabem como é complicado sair num sábado de manhã para ir na aula debaixo de um temporal do pata rachada (busão das 8:30 o cara tem que andar no teto), ainda bem que tinha alguém que lavava as marmitas e ia buscar a comida (valeu aí tchê!).

E por fim, não é só com estudo que conseguimos concluir uma faculdade, depende muito da nossa força de vontade, e nesse quesito gostaria de agradecer muito a minha irmã Thaís que revisou inúmeras vezes essa monografia (mais um pouco daria para colocar como co-autor), e especialmente meu pai Gilmar e minha mãe Loeiri pelo apoio financeiro e emocional.





*"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 buscando métodos modernos para o processo de Part-of-speech Tagging, pois ele pode ser usado como pré-processamento de várias aplicações. Estudamos diferentes métodos para gerar representações de palavras (*word embeddings*). Propomos dois modelos baseado em aprendizagem profunda: um modelo neural recursivo e um modelo neural recorrente bidirecional. O modelo neural recursivo é guiado por palavras mais fáceis de serem classificadas. O treinamento foi feito sobre três diferentes *corpus* etiquetados para o português brasileiro. Para cada modelo, avaliamos a acurácia sobre esses *corpus* utilizando três tipos de representações de palavras, e fizemos uma análise dos erros cometidos por eles. Além disso, comparamos os resultados de acurácia com os trabalhos relacionados e constatamos que nosso modelo recorrente bidirecional conseguiu a segunda melhor acurácia sobre o Mac-Morpho original para palavras fora do vocabulário. Nossos experimentos mostram que o modelo neural recorrente bidirecional é mais eficiente que o recursivo em termos de acurácia e tempo de treinamento. Este trabalho contribui com a definição e implementação de dois etiquetadores que foram disponibilizados à comunidade e que podem ser usados livremente.

**Palavras-chave:** Aprendizagem Profunda. Processamento de Linguagem Natural. Part-of-speech Tagging. Redes Neurais Recursivas. Redes Neurais Recorrentes Bidirecionais.



# 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). We propose two deep learning models: a recursive neural network model and a bidirectional recurrent neural network model. The recursive neural model is guided, where easy words to predict are classified first. The training was made with three different corpora for Brazilian Portuguese. For each model, we evaluate the accuracy over those corpora using three types of word embeddings, and then we analysed the mistakes made by them. Furthermore, we compare our result with related works, and we found that our bidirectional recurrent model reached the second top accuracy over original Mac-Morpho for out-of-vocabulary words. Our experiments show that the bidirectional recurrent model is more efficient than recursive model in terms of accuracy and training time. This work contributes with a definition and an implementation of two taggers that were made available to the community and can be used freely.

**Key-words:** Deep Learning. Natural Language Processing. Part-of-speech Tagging. Recursive Neural Networks. Bidirectional Recurrent Neural Networks.



# Lista de ilustrações

Figura 1 – Exemplo de classificação gramatical . . . . .	23
Figura 2 – Diagrama para resultado final . . . . .	27
Figura 3 – Demonstração de problemas de aprendizado supervisionado . . . . .	28
Figura 4 – Função sigmoide . . . . .	30
Figura 5 – Representação dos casos da função de custo . . . . .	32
Figura 6 – Função de custo - $J(\theta_0, \theta_1)$ . . . . .	33
Figura 7 – Funcionamento do Gradiente Descendente . . . . .	34
Figura 8 – Exemplo de <i>underfitting</i> e <i>overfitting</i> . . . . .	35
Figura 9 – Exemplo de matriz de coocorrência para criação de palavras vetorizadas	38
Figura 10 – Exemplo de vetores de <i>features</i> de cinco dimensões representando palavras	40
Figura 11 – t-SNE: Visualização para <i>word embeddings</i> . . . . .	41
Figura 12 – Abstração matemática de um neurônio . . . . .	42
Figura 13 – Rede neural com uma camada oculta . . . . .	43
Figura 14 – Modelo de rede neural - Passos do <i>Forward Propagation</i> . . . . .	45
Figura 15 – Passos do Backpropagation . . . . .	49
Figura 16 – Modelo de aprendizagem profunda . . . . .	50
Figura 17 – Sentido do aprendizado de máquina . . . . .	50
Figura 18 – Tamanho das redes neurais ao longo dos anos . . . . .	51
Figura 19 – Rede neural convolucional . . . . .	52
Figura 20 – Rede neural recorrente . . . . .	53
Figura 21 – Longa dependência entre termos . . . . .	53
Figura 22 – Rede neural recursiva . . . . .	54
Figura 23 – Arquitetura da rede neural recursiva . . . . .	59
Figura 24 – Exemplo de <i>Dropout</i> . . . . .	60
Figura 25 – Exemplo de predição . . . . .	66
Figura 26 – Arquitetura da rede neural recorrente bidirecional . . . . .	69
Figura 27 – Acurácia de treinamento e validação para modelo recursivo usando o Wang2Vec . . . . .	75
Figura 28 – Distribuição dos comprimentos das sentenças . . . . .	75
Figura 29 – Taxa de erros por tamanho da sentença no Tycho Brahe . . . . .	76





# Lista de tabelas

Tabela 1 – Notação utilizada para classificação . . . . .	29
Tabela 2 – Dados dos <i>córpus</i> . . . . .	38
Tabela 3 – Notação utilizada para representação em redes neurais . . . . .	43
Tabela 4 – Notação utilizada para aprendizado em redes neurais . . . . .	46
Tabela 5 – Comparativo das técnicas encontradas na literatura para POS Tagging	55
Tabela 6 – Comparativo dos melhores resultados encontrados na literatura para POS Tagging . . . . .	56
Tabela 7 – Notação utilizada para o modelo neural recursivo . . . . .	57
Tabela 8 – Exemplo de etiquetação . . . . .	61
Tabela 9 – Hiperparâmetros para os modelos . . . . .	72
Tabela 10 – Modelo neural recursivo: Acurácia sobre o Mac-Morpho original . . . .	74
Tabela 11 – Modelo neural recursivo: Acurácia sobre o Mac-Morpho revisado . . .	74
Tabela 12 – Modelo neural recursivo: Acurácia sobre o Tycho Brahe . . . . .	74
Tabela 13 – Modelo neural recorrente bidirecional: Acurácia sobre o Mac-Morpho original . . . . .	76
Tabela 14 – Modelo neural recorrente bidirecional: Acurácia sobre o Mac-Morpho revisado . . . . .	76
Tabela 15 – Modelo neural recorrente bidirecional: Acurácia sobre o Tycho Brahe .	76
Tabela 16 – Tamanho médio das sentenças . . . . .	77
Tabela 17 – Taxa de vetores não encontrados . . . . .	77
Tabela 18 – Taxa de ocorrência de vetores não encontrados . . . . .	78
Tabela 19 – Comparação da acurácia dos resultados com trabalhos relacionados . .	79



# Lista de siglas

**FDV** Fora do Vocabulário

**GloVe** *Global Vectors*

**GRU** *Gated Recurrent Unit*

**HAL** *Hyperspace Analogue to Language*

**LSTM** *Long Short Term Memory*

**NLM** *Neural Language Model*

**PLN** Processamento de Linguagem Natural

**POS** *Part-of-speech*

**ReLU** *Rectified Linear Unit*

**SG** *Skip-Gram*



# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>23</b>
<b>1.1</b>	<b>O problema</b>	<b>23</b>
<b>1.2</b>	<b>Objetivo</b>	<b>24</b>
<b>1.3</b>	<b>Estrutura do trabalho</b>	<b>25</b>
<b>2</b>	<b>FUNDAMENTOS</b>	<b>27</b>
<b>2.1</b>	<b>Aprendizado de máquina</b>	<b>27</b>
2.1.1	Classificação	28
2.1.2	Função de hipótese	29
2.1.3	Função de custo	31
2.1.4	Minimização da função de custo	33
2.1.5	Classificação multiclasse	34
2.1.6	Regularização	35
<b>2.2</b>	<b>Córpus e seu conjunto de classes gramaticais</b>	<b>37</b>
<b>2.3</b>	<b>Representação das palavras</b>	<b>38</b>
<b>2.4</b>	<b>Redes neurais</b>	<b>41</b>
2.4.1	Representação do modelo	41
2.4.2	Representação vetorizada do modelo	44
2.4.3	Classificação multiclasse	45
2.4.4	Função de custo	46
2.4.5	Algoritmo <i>Backpropagation</i>	47
<b>2.5</b>	<b>Aprendizagem profunda</b>	<b>49</b>
2.5.1	Redes neurais profundas	51
2.5.1.1	Redes neurais convolucionais	52
2.5.1.2	Redes neurais recorrentes	53
2.5.1.3	Redes neurais recursivas	54
<b>3</b>	<b>TRABALHOS RELACIONADOS</b>	<b>55</b>
<b>4</b>	<b>MODELO NEURAL RECURSIVO</b>	<b>57</b>
<b>4.1</b>	<b>Pré-processamento</b>	<b>58</b>
<b>4.2</b>	<b>Arquitetura</b>	<b>58</b>
4.2.1	Embeddings: camada de vetores de palavras e etiquetas	59
4.2.2	Merge: camada de composição	60
4.2.3	Flatten: camada de concatenação	60
4.2.4	Dense: camada totalmente conectada	60

4.2.5	Dropout: camada de regularização . . . . .	60
<b>4.3</b>	<b>Treinamento . . . . .</b>	<b>61</b>
4.3.1	Minimização da função de custo . . . . .	62
4.3.2	Análise de complexidade temporal . . . . .	63
<b>4.4</b>	<b>Predição . . . . .</b>	<b>64</b>
4.4.1	Análise de complexidade temporal . . . . .	67
<b>4.5</b>	<b>Implementação . . . . .</b>	<b>68</b>
<b>5</b>	<b>MODELO NEURAL RECORRENTE BIDIRECIONAL . . . . .</b>	<b>69</b>
<b>5.1</b>	<b>Pré-processamento . . . . .</b>	<b>69</b>
<b>5.2</b>	<b>Arquitetura . . . . .</b>	<b>69</b>
5.2.1	GRU: camada recorrente bidirecional . . . . .	70
<b>5.3</b>	<b>Treinamento . . . . .</b>	<b>70</b>
5.3.1	Minimização da função de custo . . . . .	70
<b>5.4</b>	<b>Predição . . . . .</b>	<b>70</b>
<b>5.5</b>	<b>Implementação . . . . .</b>	<b>70</b>
<b>6</b>	<b>TESTES E RESULTADOS . . . . .</b>	<b>71</b>
<b>6.1</b>	<b>Ambiente de teste . . . . .</b>	<b>71</b>
6.1.1	Bibliotecas . . . . .	71
6.1.2	Máquina . . . . .	71
<b>6.2</b>	<b>Pré-processamento . . . . .</b>	<b>71</b>
<b>6.3</b>	<b>Hiperparâmetros . . . . .</b>	<b>72</b>
<b>6.4</b>	<b>Resultados . . . . .</b>	<b>73</b>
6.4.1	Acurácia . . . . .	73
<b>6.5</b>	<b>Comparação com trabalhos relacionados . . . . .</b>	<b>78</b>
<b>7</b>	<b>CONSIDERAÇÕES FINAIS . . . . .</b>	<b>81</b>
<b>7.1</b>	<b>Conclusão . . . . .</b>	<b>81</b>
<b>7.2</b>	<b>Trabalhos futuros . . . . .</b>	<b>81</b>
	<b>Referências . . . . .</b>	<b>83</b>
	<b>APÊNDICES . . . . .</b>	<b>87</b>
	<b>APÊNDICE A – IMPLEMENTAÇÃO DO MODELO NEURAL RE- CURSIVO . . . . .</b>	<b>89</b>
<b>A.1</b>	<b>Arquitetura . . . . .</b>	<b>89</b>
<b>A.2</b>	<b>Treinamento . . . . .</b>	<b>89</b>
<b>A.3</b>	<b>Predição . . . . .</b>	<b>90</b>

	<b>APÊNDICE B – IMPLEMENTAÇÃO DO MODELO NEURAL RE- CORRENTE BIDIRECIONAL . . . . .</b>	<b>93</b>
<b>B.1</b>	<b>Arquitetura . . . . .</b>	<b>93</b>
<b>B.2</b>	<b>Treinamento . . . . .</b>	<b>94</b>
<b>B.3</b>	<b>Predição . . . . .</b>	<b>94</b>

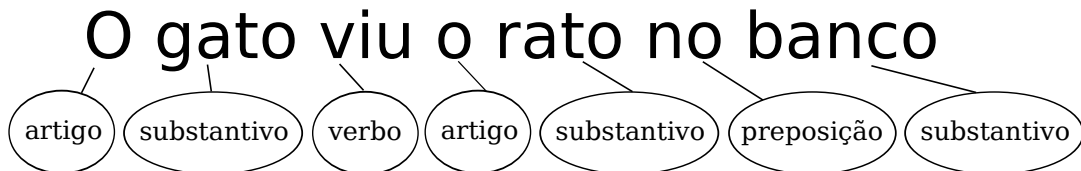




# 1 Introdução

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.

Figura 1: Exemplo de classificação gramatical



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 trinta 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.

A ambiguidade é uma propriedade que faz com que um objeto linguístico possa ser interpretado de modos diferentes. A ambiguidade morfológica ocorre quando uma palavra pode pertencer a mais de uma classe gramatical. Ao encontrar diferentes classes gramaticais possíveis de serem extraídas de uma palavra, o sistema necessita distinguir um destes significados, determinando, segundo o contexto, qual a classe a ser aplicada. Por isso, a ambiguidade é um grande problema a ser considerado no processo de POS Tagging.

## 1.1 O problema

POS Tagging é um processo difícil de ser realizado em PLN, pois linguagens na-

turais 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 dois métodos de aprendizagem profunda, um que utiliza um modelo neural recursivo com múltiplas camadas e outro que utiliza um modelo neural recorrente bidirecional. Utilizamos redes neurais pois elas oferecem um jeito alternativo de realizar aprendizado de máquina quando temos hipóteses complexas com muitas características

## 1.2 Objetivo

Este trabalho irá propor dois novos métodos 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 dois métodos originais: Um método que se baseia em classificar primeiramente palavras mais fáceis, desse modo, espera-se deixar palavras ambíguas por último; e outro método que se baseia em lembrar de longas dependências entre palavras. Além disso, será feito o uso de uma 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 significativas das palavras. Além da acurácia da classificação, será considerado outra métrica importante: o número de sentenças que foram etiquetadas corretamente ([MANNING, 2011](#)).

## 1.3 Estrutura do trabalho

Este trabalho está dividido na seguinte maneira: No [Capítulo 2](#) são mostrados 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 do modelo neural recursivo e no [Capítulo 5](#) será explicado o modelo neural recorrente bidirecional. No [Capítulo 6](#) serão mostrados os testes realizados e os resultados obtidos, e também uma discussão dos resultados. Por fim, no [Capítulo 7](#) serão apresentadas as considerações finais.



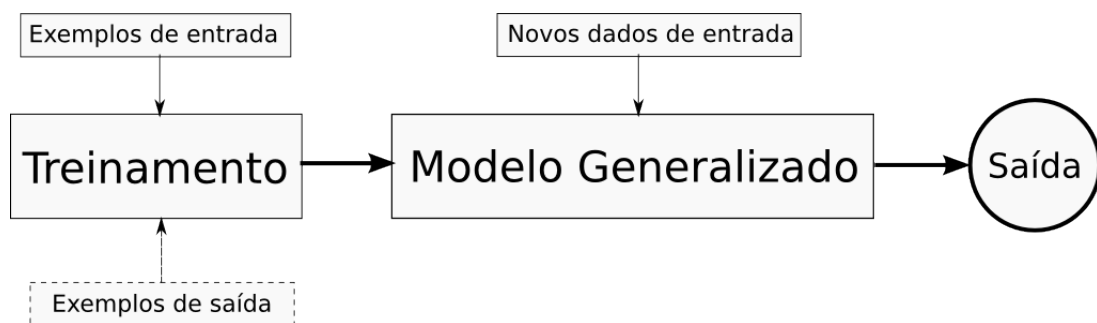
## 2 Fundamentos

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.

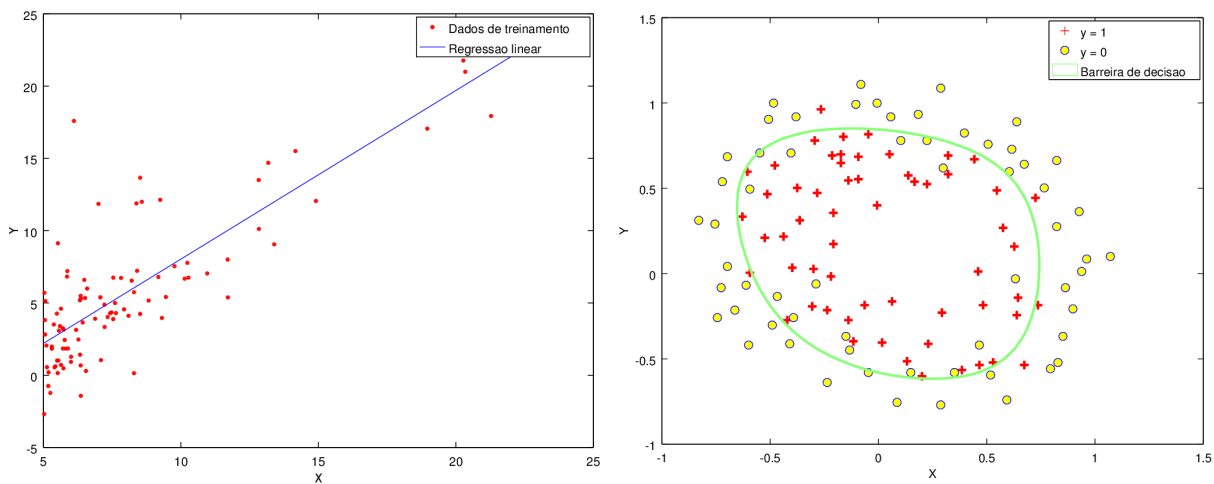
O aprendizado supervisionado permite dividir os problemas em duas categorias distintas:

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

deseja-se mapear variáveis de entrada em alguma função contínua. A Figura 3(a) 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 3(b) demonstra esse processo. Tem duas classes diferentes, as representadas pelos círculos amarelos e pelas cruces vermelhas, já o contorno em verde representa o resultado da classificação sobre esses dados de exemplo.

Figura 3: Demonstração de problemas de aprendizado supervisionado



(a) Demonstração de regressão

(b) 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
$\theta$	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  $\theta$ 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  $\theta$  e  $X$ , será atribuído  $X_0^{(i)} = 1$  para todos os valores de  $i$ . Isso faz com que os vetores  $\theta$  e  $X^{(i)}$  combinem um com o outro elementarmente.

Portanto, nossa função de hipótese deve satisfazer a equação:

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

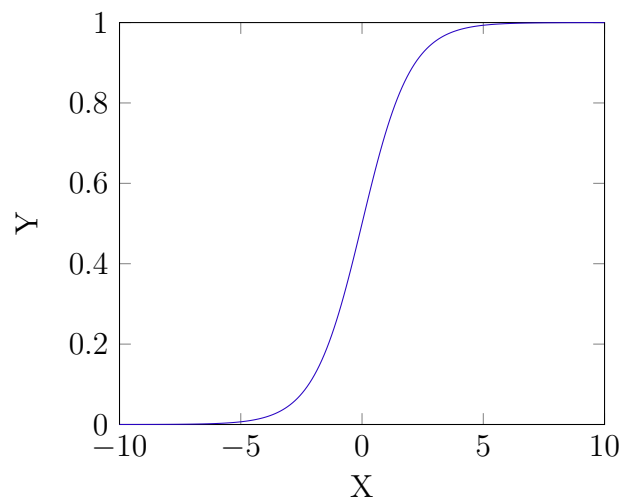
Para fazer com que uma entrada contínua fique nesse intervalo, é necessário utilizar uma função linear que faça esse mapeamento de modo linear, e para isso muda-se a forma da função de hipótese. Uma forma possível é usar a função sigmoide [Equação 2.3](#), também conhecida como função logística.

A função sigmoide representada na [Figura 4](#) mapeia qualquer número no intervalo  $[0, 1]$ , tornando isso útil para transformar qualquer valor arbitrário em uma função mais ideal para problemas de classificação.

$$h_{\theta}(x) = g(\theta^T X)$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad (2.3)$$

Figura 4: Função sigmoide



Para calcular a função de hipótese, é possível 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 é possível 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, é possível 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 3\(b\)](#). 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

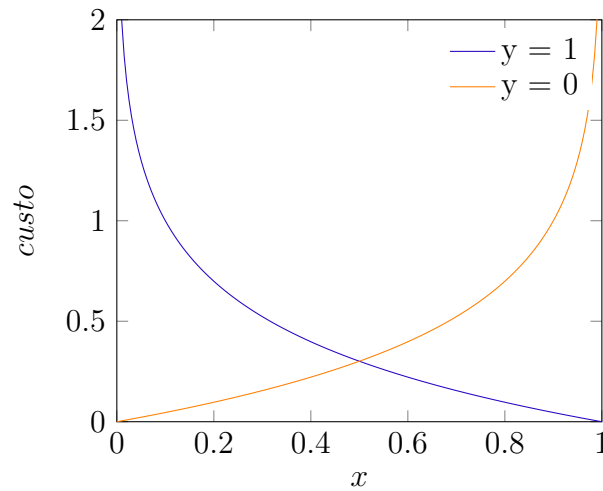
Para medir a precisão da função de hipótese é necessário uma função que diga o quão precisa aquela hipótese é. Tal função é denominada de função de custo. Com ela, é feita uma média de todos os resultados das hipóteses com a entrada  $X$  comparada com o valor objetivo  $Y$ . Ela pode ser expressa como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

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

Onde,

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

Figura 5: Representação dos casos da função de custo



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 5](#) demonstra os dois respectivos casos dessa equação.

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.

É possível simplificar a [Equação 2.5](#) com dois casos condicionais em apenas um caso:

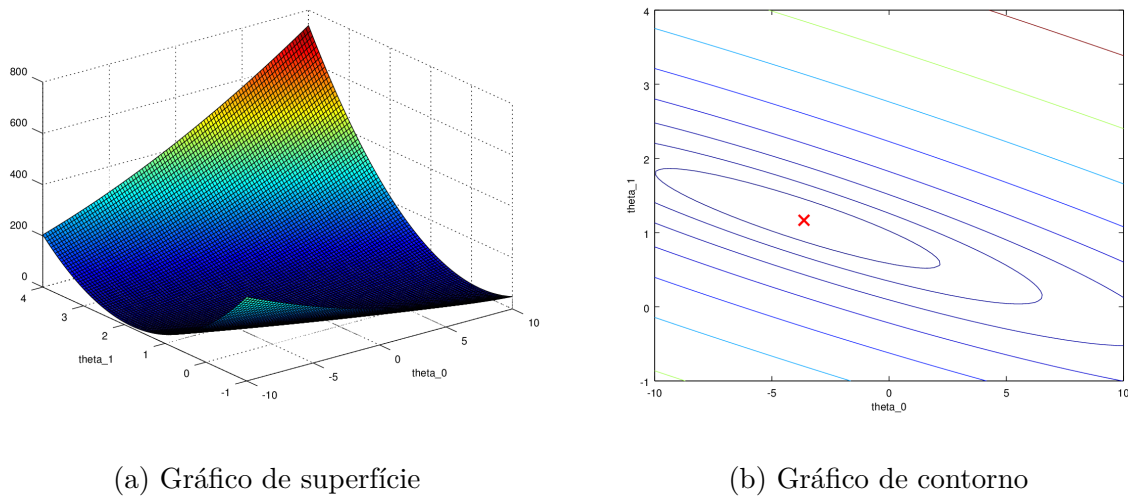
$$\text{Custo}(h_{\theta}(x), y) = -y \log(h_{\theta}(x)) - (1 - y) \log(1 - h_{\theta}(x)) \quad (2.6)$$

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 [Equação 2.6](#) na função de custo da [Equação 2.4](#), é possível reescrevê-la como a [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  $\theta$ s. Usando dois parâmetros é possível visualizar uma função de custo através do número de iterações, isso pode ser observado no exemplo da [Figura 6\(a\)](#) e da [Figura 6\(b\)](#).

Figura 6: Função de custo -  $J(\theta_0, \theta_1)$ 

Fonte: Ng (2015)

### 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  $\theta$ s aprendidos. Com isso, é possível melhorar a 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  $\theta$ s alternadamente, com isso a encosta da tangente dará a direção para seguir adiante. Além disso, é aplicada uma taxa de aprendizagem  $\alpha$  a cada iteração, para tentar controlar a convergência.

Seu funcionamento é descrito pelo algoritmo 2.9.

repita até convergir {

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

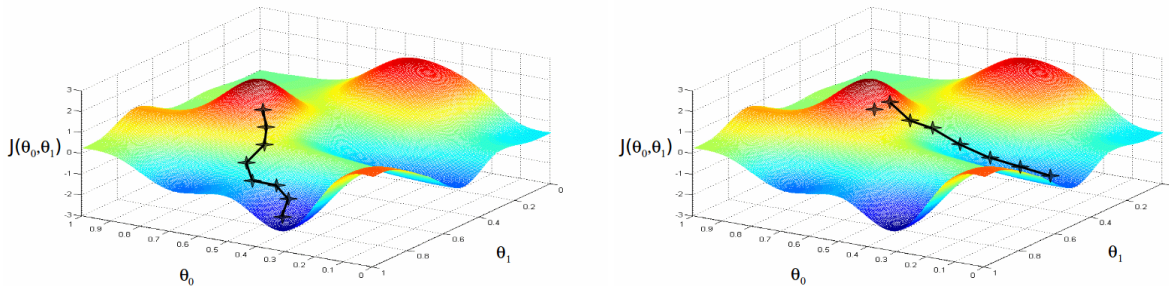
}

Ao trabalhar sobre a derivada parcial da função de custo, é possível chegar na Equação 2.10 e adiante na versão vetorizada na Equação 2.11.

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

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

Figura 7: Funcionamento do Gradiente Descendente



(a) Convergindo para um mínimo global

(b) Convergindo para um mínimo local

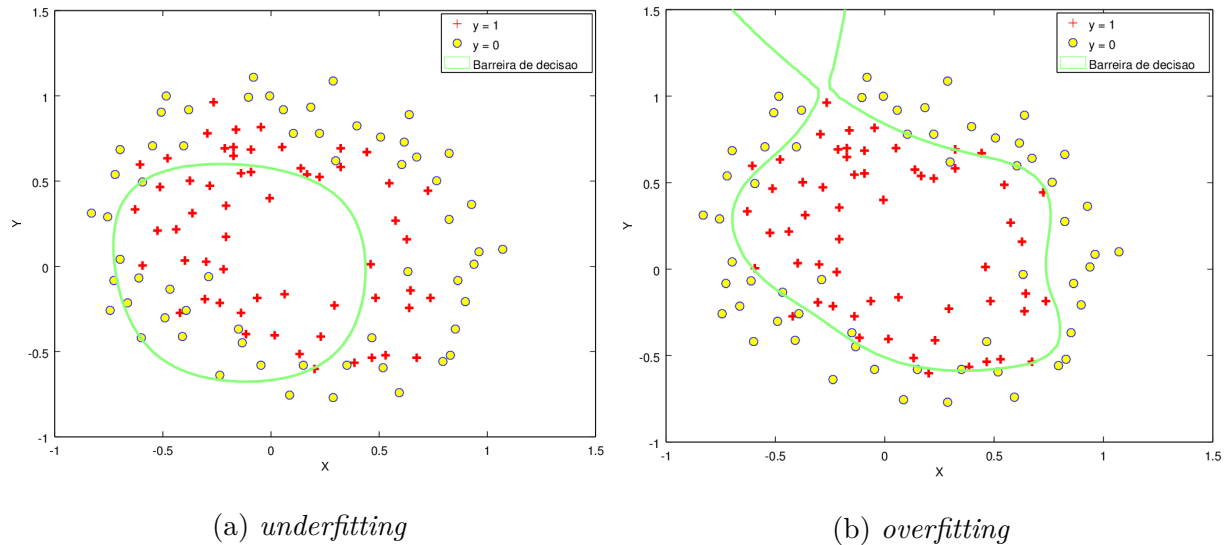
Fonte: Ng (2015)

Na Figura 7(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  $\alpha$ ) pode-se convergir para um mínimo local como mostrado na Figura 7(b). Basicamente, se  $\alpha$  for um valor muito alto, os passos para o minimal ótimo serão grandes e portanto o risco de divergência será maior, caso seja muito baixo, os passos para o minimal 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  $\alpha$ . Mas, também é sugerido que não deve-se tentar codificar esses algoritmos, visto que são mais complexos e requerem um bom conhecimento de cálculo numérico, onde ao invés é possível 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

Figura 8: Exemplo de *underfitting* e *overfitting*

Fonte: Ng (2015)

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 8 é o exemplo mostrado na Figura 3(b).

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:
- Manter todos as *features*, mas reduzir os parâmetros  $\theta_j$ .

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 deseja-se eliminar a influência de  $\theta_3x^3$  e  $\theta_4x^4$  da Equação 2.12, seria possível 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, é adicionado dois termos extras no fim da função de custo para inflar o custo de  $\theta_3$  e  $\theta_4$ . Agora, para a função de custo chegar a zero, é necessário reduzir os valores de  $\theta_3$  e  $\theta_4$  para próximos a zero. Isso vai reduzir substancialmente o peso das *features* cuja influência deseja-se 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  $\lambda$ , que determina o quanto os custos dos parâmetros  $\theta$ 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 o  $\theta_0$  não está sendo regularizado.

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 deseja-se regularizar o  $\theta_0$  e então é preciso separá-lo em um outro passo.

repite 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. Eles 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 ao 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.

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.

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	1,541,654	265

A aprendizagem supervisionada, utilizando *córpus* como entrada, tem se mostrado uma estratégia atrativa. Neste trabalho vamos utilizar três *córpus*: o Mac-Morpho original, o Mac-Morpho revisado e o Tycho Brahe. O conjunto de etiquetas para esses *córpus* podem ser encontrados, respectivamente, em (FONSECA; ROSA; ALUÍSIO, 2015) e (UNICAMP, 2010).

## 2.3 Representação das palavras

Palavras 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 9 ilustra esse processo que se baseia 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 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 9: Exemplo de matriz de coocorrência para criação de palavras vetorizadas

### EXEMPLO DE VOCABULÁRIO:

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

	Eu	gosto	de	ciência	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
ciência	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 (similaridade cosseno, similaridade de Jaccard, etc). Quando é falado 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 estender 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 10 ilustra esse processo.

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

Figura 10: Exemplo de vetores de *features* de cinco dimensões representando palavras

<b>Vetores de <i>features</i> para palavras no vocabulário</b>				
⋮				
leal:	0.41		0.54	
	0.49			
leão:	0.19		0.33	
	0.01			
lenha:	0.90		0.57	
	0.16			
⋮				
<b>Vetores de <i>features</i> para capitalização de palavras</b>				
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	
<b>Representação final das palavras no modelo</b>				
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			

Adaptade de: [Fonseca, Rosa e Aluísio \(2015\)](#)

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 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.

O *wang2vec* é a técnica de representação distribuídas de palavras mais recente. Essa técnica é uma modificação do *word2vec*, onde as *embeddings* geradas são adaptadas para problemas onde a sintaxe importa ([LING et al.,](#) ). Ou seja, nessa técnica a ordem das palavras importam para gerar o vetor da palavra sendo analisada.

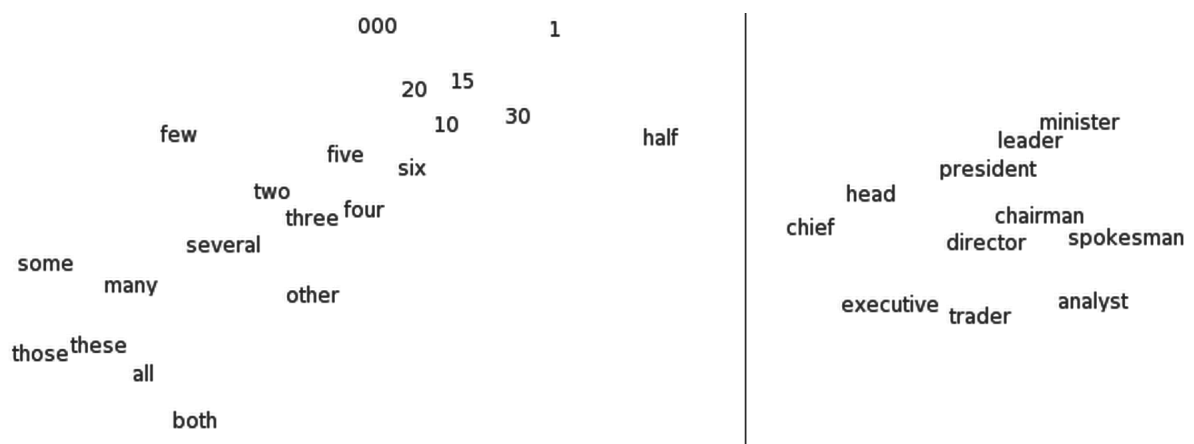
Portanto, para a criação de um vetor de *features*, há a transformação  $W$  para uma palavra *word*:

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

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 11: t-SNE: Visualização para *word embeddings*



Fonte: Turian, Ratinov e Bengio (2010)

Na Figura 11 é possível 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 próximas umas das outras.

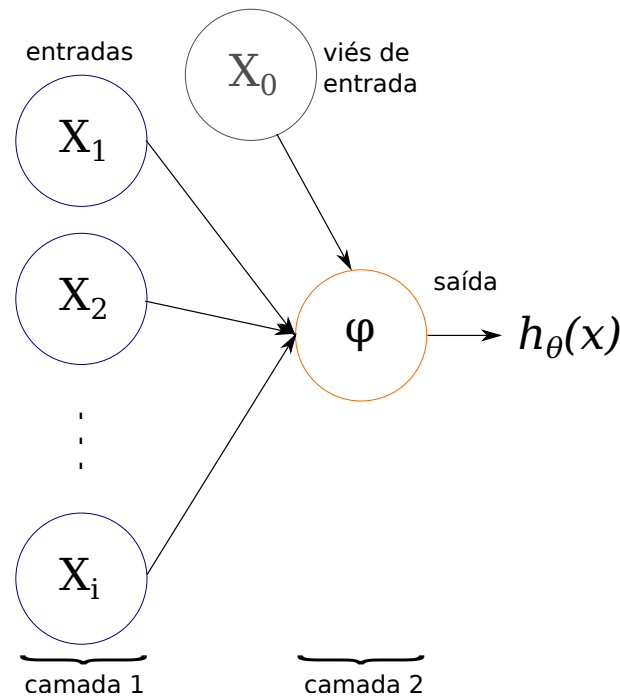
## 2.4 Redes neurais

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 Representação do modelo

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 12: 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  $\theta$ s a serem aprendidos são chamados de pesos. Visualmente, uma representação simples desse modelo se parece como a [Figura 12](#). Os dados dos nós de entrada (camada 1) vão para outro nó (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”.

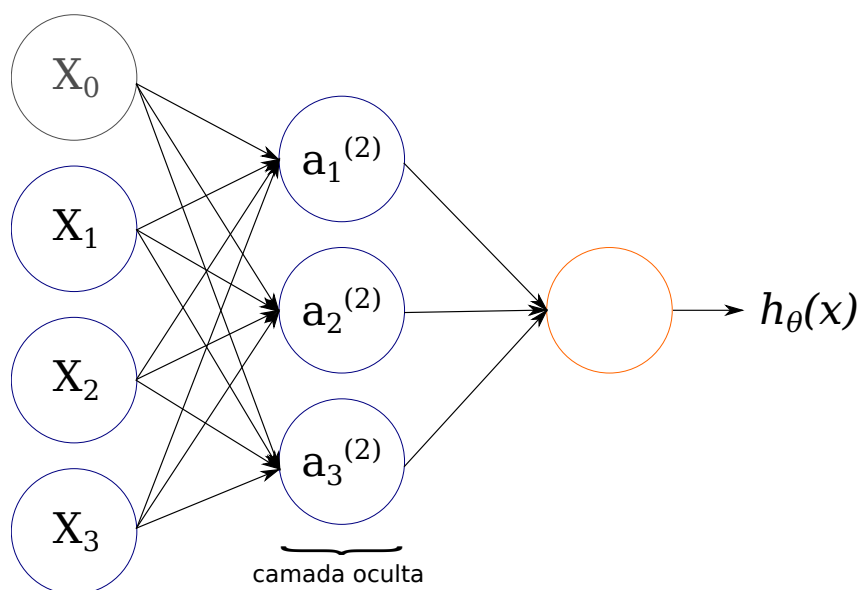
Os nós das camadas intermediárias são nomeados de  $a_0^{(j)}$ ,  $a_1^{(j)}$ , ...,  $a_n^{(j)}$  e chamados de unidades de ativação. A [Tabela 3](#) reúne as notações geralmente usadas no contexto de redes neurais.

A [Figura 13](#) mostra uma rede neural com uma camada oculta. Apesar do nó viés não aparecer na camada oculta ele está presente nela. Aliás, ele está presente e alimenta todos os nós de todas camadas da rede (exceto a camada de saída), porém ele não recebe entradas.

Tabela 3: Notação utilizada para representação em redes neurais

$\varphi$ 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 $j$ .

Figura 13: Rede neural com uma camada oculta



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

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

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 é computado os nodos de ativação fazendo uma matriz de parâmetros com dimensão  $3 \times 4$ . É aplicado 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.

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

## 2.4.2 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  $(n + 1)$ , é possível calcular  $z^{(j)}$  ao fazer a multiplicação matriz-vetor:

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

Como é conhecido que  $x$  é nosso vetor de entrada ( $x = a^{(1)}$ ), é possível 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  $S_j$ . Sendo assim, é possível 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. É possível 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:

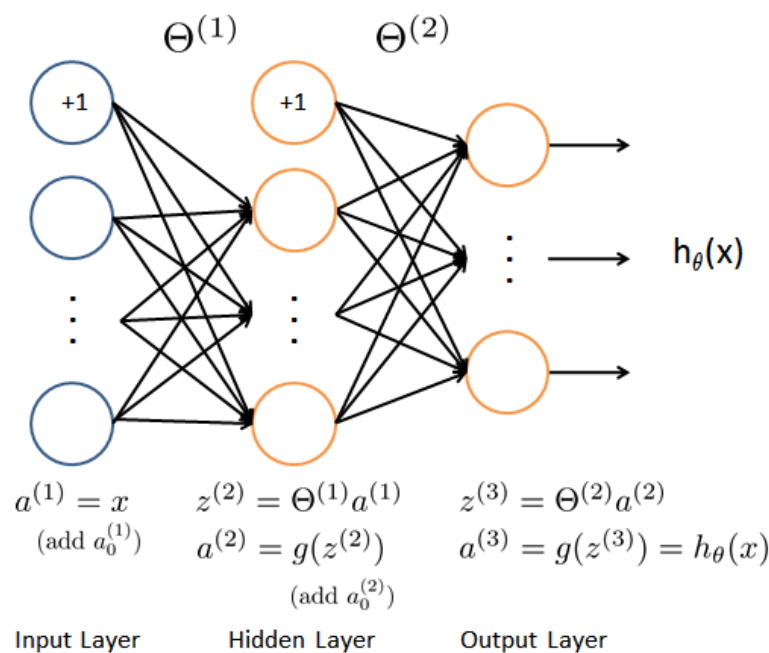
$$h_{\theta}(x) = a^{(j+1)} = g(z^{(j+1)})$$

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*.

### 2.4.3 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  $\theta$ , 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 14](#).

Figura 14: Modelo de rede neural - Passos do *Forward Propagation*



Fonte: Ng (2015)

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 estender essa 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  $y$ .

#### 2.4.4 Função de custo

As subseções anteriores mostraram como representar uma rede neural. Agora o interesse é como realizar a aprendizagem para resolver o problema de classificação.

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

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 [subseção 2.4.3](#) 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](#).

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), lidou-se com as múltiplas matrizes  $\theta$ s. Conforme a regra [2.17](#), o número de colunas de uma dada matriz  $\theta$  é igual ao número



de nodos na camada atual (incluindo o viés). Já o número de linhas da matriz  $\theta$  é igual ao número de nodos na próxima camada (sem o viés).

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

### 2.4.5 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  $\theta$ . E para isso é necessário o cálculo da derivada parcial da função de custo em relação aos parâmetros  $\theta$  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. É possível estender 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  $\delta$  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  $\delta$  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  $\delta$  da camada  $l$  são calculados ao multiplicar os valores de  $\delta$  na próxima camada com a matriz  $\theta$  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](#).

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

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

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

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.

---

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

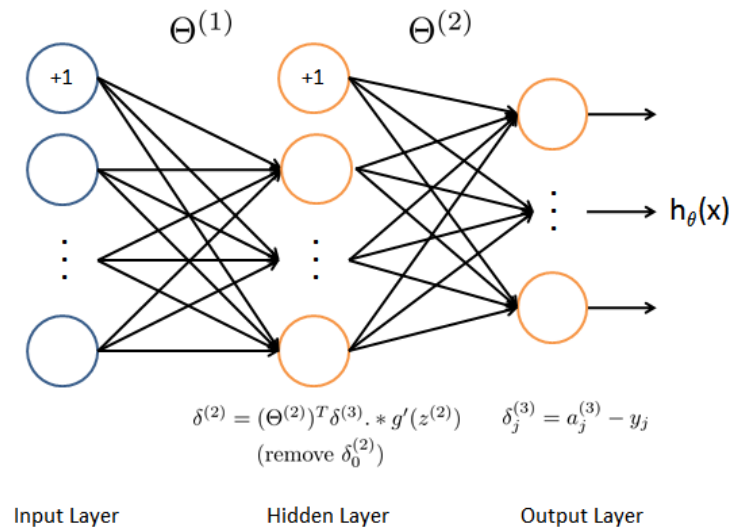
```

---

A matriz  $\Delta$  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 15 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  $\theta$  não podem ser inicializados com zero, pois isso fará com que a convergência do algoritmo se torne mais lenta. Uma solução para esse problema é realizar uma **inicialização aleatória** dos pesos.

Figura 15: Passos do Backpropagation



Fonte: Ng (2015)

## 2.5 Aprendizagem profunda

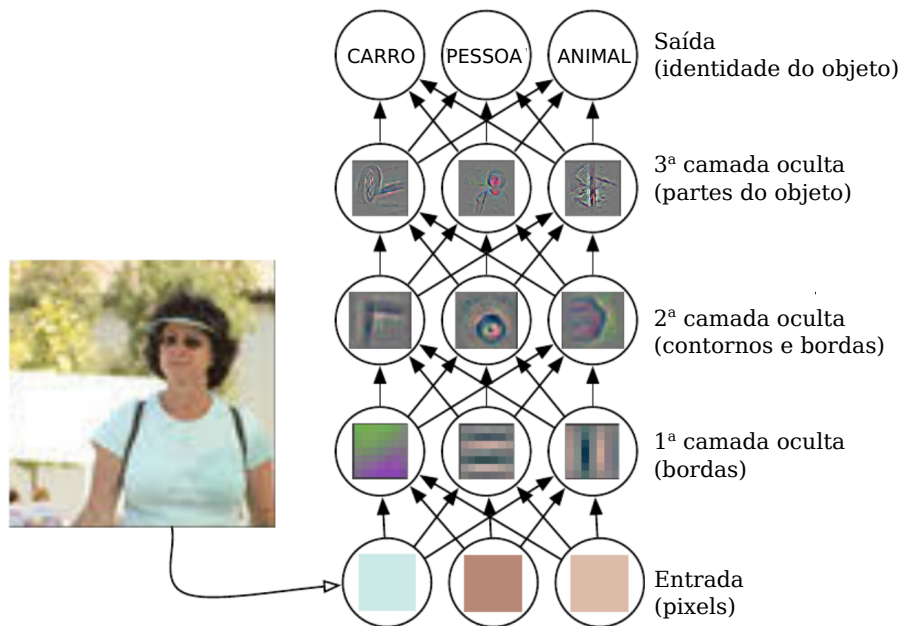
A aprendizagem profunda é uma ramificação da aprendizagem de máquina, que baseia-se num conjunto de algoritmos que procuram modelar abstrações de alto nível usando modelos arquitetoriais, com estruturas complexas, compostas de múltiplas transformações não lineares (DENG; YU, 2014). Suas definições são 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.

A aprendizagem profunda resolve esse problema de representação ao introduzir representações que são expressas em termos de representações mais simples, permitindo o computador a construir conceitos complexos a partir de conceitos simples. A Figura 16 ilustra um modelo de aprendizagem profunda.

Na Figura 16, cada *pixel* da imagem são *features* passadas para a primeira camada oculta, que consegue identificar bordas facilmente ao comparar o brilho de *pixels* vizinhos.

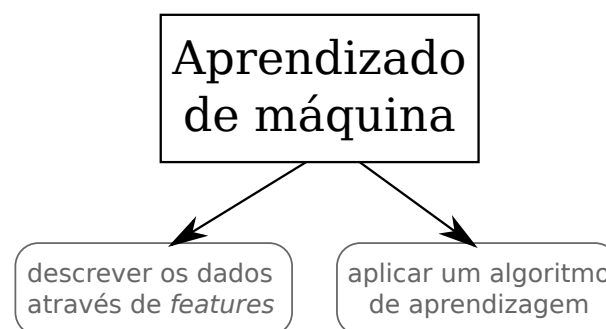
Figura 16: Modelo de aprendizagem profunda



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

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 coleções específicas 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](#)).

Figura 17: Sentido do aprendizado de máquina



A [Figura 17](#) 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.

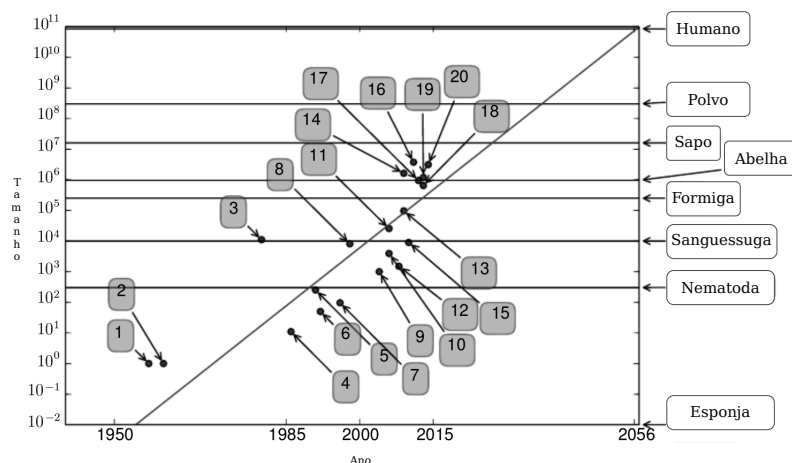
A aprendizagem profunda tenta automaticamente aprender boas *features* ou representações. Na [seção 2.3](#) foi mostrado o conceito de *word embeddings* ou vetor de *features*. Na verdade esse conceito está fortemente associado ao aprendizado profundo, onde tenta-se aprender múltiplos níveis da representação. Especificamente, é um tipo de aprendizado de máquina, uma técnica que permite sistemas computacionais a melhorarem com experiência e dados ([BENGIO; GOODFELLOW; COURVILLE, 2015](#)).

A técnica utilizada ultimamente em POS Tagging 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 profunda. Esses vetores são então ajustados ao realizar o *Backpropagation* até a camada de entrada. 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.

### 2.5.1 Redes neurais profundas

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](#)).

Figura 18: Tamanho das redes neurais ao longo dos anos



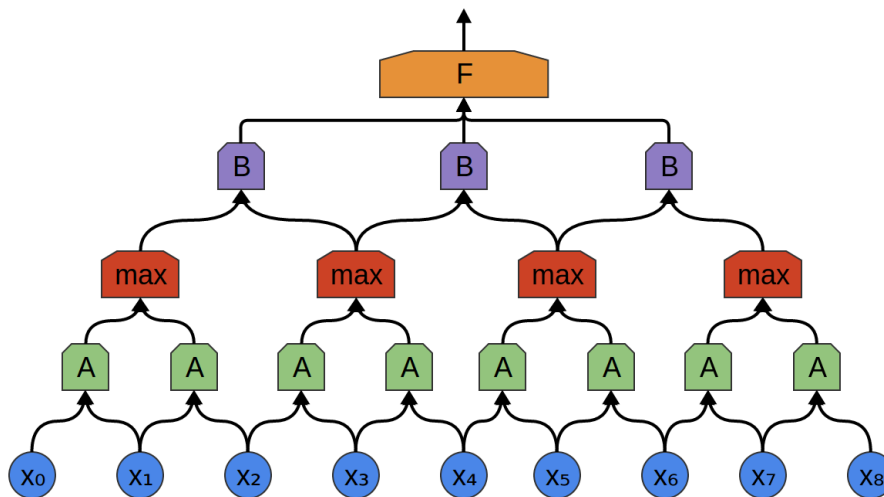
Adaptado de: [Bengio, Goodfellow e Courville \(2015\)](#)

Como o processo de criação automatizada de *features* acaba gerando muitos vetores de representações, é necessário 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 18 demonstra o aumento do tamanhos das redes neurais de acordo com o tempo.

### 2.5.1.1 Redes neurais convolucionais

Redes neurais convolucionais aprendem automaticamente *features* locais e assume que essas *features* são úteis em outros lugares. Desse modo fica mais fácil de aprender e o erro é reduzido. Elas já foram aplicadas em POS Tagging por Collobert e Weston (2008).

Figura 19: Rede neural convolucional



Fonte: Olah (2014)

Na Figura 19 é possível ver uma rede convolucional onde são extraídas *features* ao juntar entradas consecutivas duas-a-duas. Após isso, é feito um *max-pooling*, isso significa que é focalizado as *features* mais significativas do contexto sendo analisado, mas não diz qual delas são as melhores. Após selecionar as melhores *features* é então criado novas *features* em cima delas. Matematicamente, isso pode ser visto como:

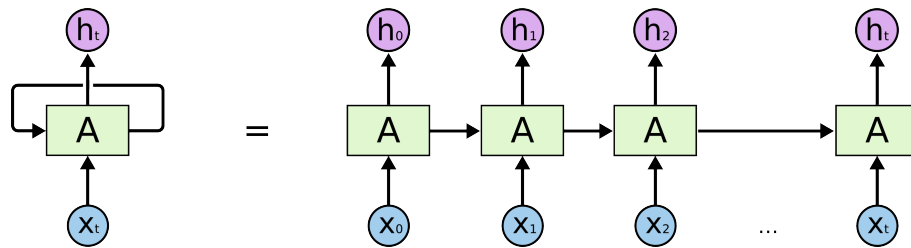
$$\begin{aligned}
 A_j &= f(\theta_1^T [1, x_j, x_{j+1}]) \\
 B_j &= f(\theta_2^T \max(A_{2*j}, A_{2*j+1})) \\
 F &= f(\theta_3^T B)
 \end{aligned}$$

Onde  $x$  são as entradas, *max* é uma camada de *max-pooling*,  $f$  é uma função de ativação,  $A$  e  $B$  são *features* aprendidas, e  $\theta_i$  é a matriz de pesos compartilhados na  $i$ -ésima camada.

## 2.5.1.2 Redes neurais recorrentes

Redes neurais recorrentes são um tipo de rede neural que tem como entrada previsões passadas. Elas são capazes de lidar com entradas de tamanhos variáveis (como sentenças). O processo de aprendizagem é feito baseando-se em informações já aprendidas no passado. Elas são usadas em tradução automática (KOMBRINK et al., 2011) e outras tarefa de PLN.

Figura 20: Rede neural recorrente

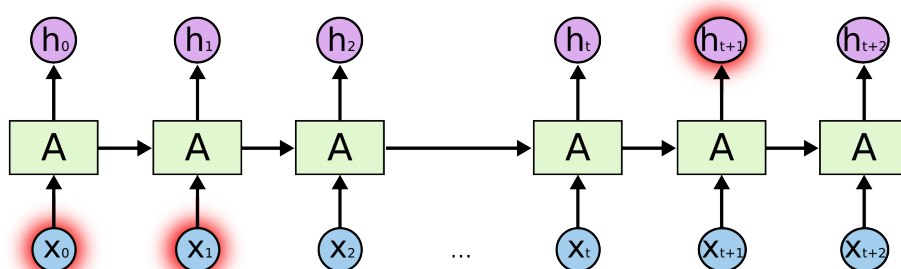


Fonte: Olah (2015)

Na Figura 20 é possível visualizar uma simples rede neural recorrente à esquerda e seu desenrolamento à direita. A rede olha uma entrada  $x_t$  e dá uma saída  $h_t$  como resposta em um momento  $t$ . Um *loop* permite que uma informação seja passada de um momento no tempo para o próximo.

Outro tipo de rede neural é a rede neural recorrente bidirecional (SCHUSTER; PALIWAL, 1997). Com ela é possível olhar informações que também estão no futuro, isto é, no momento  $t + 1, t + 2, \dots, t + n$ , onde  $n$  é o tamanho da sentença. Isso é feito através da composição da sequência de saídas da esquerda para direita (olhando o passado) e da direita para esquerda (olhando o futuro).

Figura 21: Longa dependência entre termos



Fonte: Olah (2015)

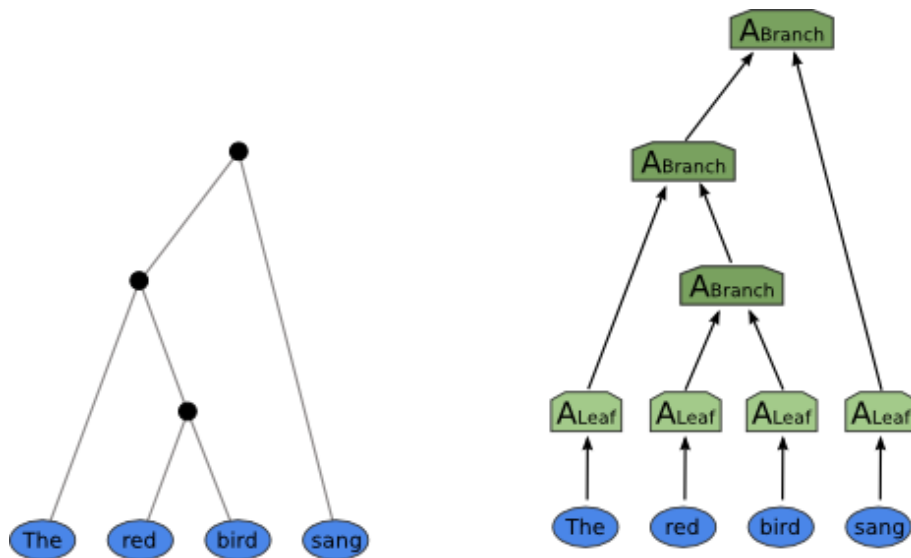
Um dos principais problemas das redes recorrentes é que elas não conseguem aprender uma longa dependência entre um termo e seus predecessores, como mostrado na Figura 21.

Para contornar esse problema, foram criadas redes neurais recorrentes com memória. As redes *Long Short Term Memory* (LSTM) foram introduzidas por Hochreiter e Schmidhuber (1997), e com o tempo foram refinadas, criando versões mais simplificadas, como a *Gated Recurrent Unit* (GRU) (CHO et al., 2014). As redes neurais com memória são utilizadas atualmente em uma grande variedade de problemas de PLN.

### 2.5.1.3 Redes neurais recursivas

Redes neurais recursivas são uma generalização de redes neurais recorrentes. Geralmente são utilizadas em tarefas de *parsing* (SOCHER et al., 2011; SOCHER; MANNING; NG, 2010), onde é possível montar uma estrutura de árvore para classificação.

Figura 22: Rede neural recursiva



Fonte: Olah (2015)

Na Figura 22 é possível visualizar uma rede neural recursiva simples. Nessa rede neural, cada folha é a entrada, e conforme sobe-se na árvore é feita a composição das *features* da entrada com nós intermediários, até que chega-se a um resultado final na raiz.



### 3 Trabalhos relacionados

Vários métodos já foram propostos para resolver o problema de POS Tagging para o português brasileiro, apesar de nenhum deles ter uma acurácia de 100%, vários conseguiram ótimos resultados e utilizaram variadas técnicas para isso. A [Tabela 5](#) sumariza as técnicas encontradas nos trabalhos relacionados.

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

Autores	Modelo	Representação das palavras	Córpus
<a href="#">Kepler (2005)</a>	VLMM	Sequência de caracteres	Tycho Brahe
<a href="#">Santos e Zadrozny (2014)</a>	Redes neurais profundas	Vetores (CharWNN)	Tycho Brahe; Mac-Morpho (versões 1, 2)
<a href="#">Fonseca, Rosa e Aluísio (2015)</a>	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)

Em ([KEPLER, 2005](#)) é apresentado um etiquetador morfossintático baseado em cadeias de Markov. São realizados 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, em que etiqueta de uma palavra depende da própria palavra e de etiquetas anteriores. Ele é testado sobre o *córpus* Tycho Brahe, e apresenta uma acurácia de 95,51% com o etiquetador VLMC, essa acurácia é alcançada com um tempo de aprendizagem + etiquetagem de 157 segundos. O VLMC é usando novamente em ([KEPLER; FINGER, 2010](#)) e consegue-se uma acurácia de 96.29%.

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 diferentes técnicas de representação das palavras são utilizadas: 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 6 mostra os melhores resultados para cada *córpus*.

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%
Kepler e Finger (2010)	Tycho Brahe	96,29%	71,60%
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) <sup>1</sup>	Tycho Brahe	96,91%	84,14%
Fonseca, Rosa e Aluísio (2015) <sup>1</sup>	Mac-Morpho versão 1	97,57%	93,38%
Fonseca, Rosa e Aluísio (2015) <sup>1</sup>	Mac-Morpho versão 2	97,48%	94,34%
Fonseca, Rosa e Aluísio (2015) <sup>1</sup>	Mac-Morpho versão 3	97,33%	93,66%

<sup>1</sup> Usando NLM como representação das palavras.

## 4 Modelo neural recursivo

Neste capítulo será explicado um dos modelos criados para resolver o problema de POS Tagging. Para isso, vamos falar primeiramente quais são os pré-processamentos feitos; depois vamos definir a arquitetura do modelo; explicar o algoritmo de treinamento e de predição; e por fim, comentar sobre a implementação do modelo.

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 modelo neural recursivo

---

$m$	número de exemplos de treinamento
$d$	dimensão das palavras e etiquetas vetorizadas
$t$	tamanho da janela de palavras e de classes
$B$	tamanho do <i>beam</i>
$J_p$	janela de palavras
$J_c$	janela de classes (ou de etiquetas)
$J_{c_b}$	$b$ -ésima janela de etiquetas, onde $0 \leq b < B$
$S_n$	$n$ -ésima sentença
$s_b$	$b$ -ésima sequência de palavras e etiquetas
$p_b$	$b$ -ésimo vetor de probabilidades
$M_b$	$b$ -ésima matriz de predição
$M_{i,j}$	elemento da linha $i$ e coluna $j$ da matriz de predição
$h_{dim}$	número de neurônios na camada oculta
$\omega$	conjunto de palavras
$\gamma$	conjunto de classes gramaticais
$c_i$	$i$ -ésima classe gramatical da sentença, onde $c \in \gamma$ e $0 \leq i <  S_n $
$w_i$	$i$ -ésima palavra da sentença, onde $w \in \omega$ e $0 \leq i <  S_n $
$i(x)$	função que retorna o índice da palavra ou classe $x$
$c_i^t$	sequência de classes que começa em $i$ e termina em $t$
$w_i^t$	sequência de palavras que começa em $i$ e termina em $t$
$V_i$	vetor centrado em $i$ com os índices das $t$ palavras concatenados
$I_t$	matriz de dimensão $t \times t$ com 1s na diagonal secundária
$Q$	conjunto com os índices das palavras já classificadas
$P$	Fila de prioridades de tamanho $B$

---

Optamos por criar um modelo neural recursivo pois acreditamos que uma rede neural recursiva pode ajudar a classificar corretamente as palavras devido ao fato de haver um contexto de etiquetas. Esse contexto é variável, e portanto, também acreditamos que a rede neural recursiva pode lidar com longa dependência.

## 4.1 Pré-processamento

Para cada sentença  $S_n$  nós criamos uma janela com  $t$  índices das palavras  $\{v(w_1), \dots, v(w_t)\}$ , para computar a  $n$ -ésima palavra, é centralizada a janela em  $n$  e concatena-se todos os índices das palavras da metade à esquerda e da metade à direita em um novo vetor  $V_n$  de dimensão  $t \times 1$ . A [Equação 4.1](#) demonstra isso.

$$V_n = \{v(w_{n-\lfloor t/2 \rfloor}), \dots, v(w_n), \dots, v(w_{n+\lfloor t/2 \rfloor})\} \quad (4.1)$$

Para palavras no começo da sentença, as  $\lfloor t/2 \rfloor$  palavras à esquerda não existem. Para contornar esse problema nós preenchemos esses espaços com um símbolo (*token*) especial. O mesmo ocorre para palavras no fim da sentença.

Nosso modelo contou com quatro *tokens* especiais listados abaixo

- `<mask>`: Para as extremidades da janela de palavras e da janela de etiquetas.
- `<unknown>`: Para palavras raras ou desconhecidas.
- `<padding_prefix>`: Para completar o vetor de prefixos.
- `<padding_suffix>`: Para completar o vetor de sufixos.

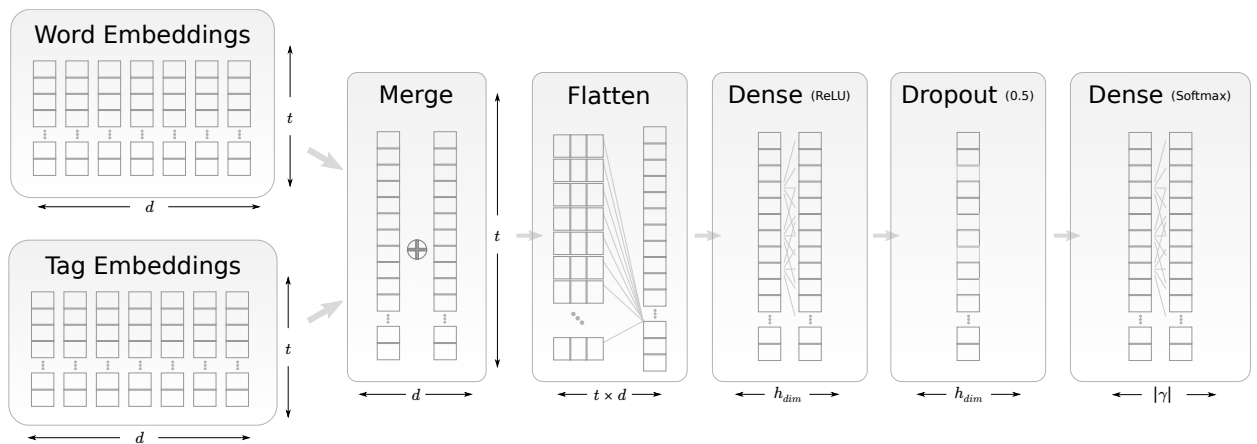
Consideramos uma palavra como sendo rara quando a quantidade de ocorrências dela no treinamento é menor que um hiperparâmetro determinado no momento da compilação.

## 4.2 Arquitetura

A arquitetura do modelo neural recursivo implementado pode ser vista na [Figura 23](#).

Como entrada da rede é passado a janela de palavras e uma janela de *tags* (etiquetas) das palavras já etiquetadas. A janela de etiquetas é passada para a camada *Tag Embeddings* e a janela de palavras é passada a camada *Word Embeddings*. Fizemos uma composição através de uma operação de soma dos vetores das *tags* com os vetores das palavras. Após calcular a composição dos vetores de palavras com os vetores das etiquetas, nós concatenamos todos os vetores dentro da janela e passamos para a próxima camada. Aplicamos esses vetores concatenados a uma camada totalmente conectada (*Fully connected* ou *Dense*) com a função de ativação *Rectified Linear Unit* (**ReLU**) [Equação 4.2](#). Aplicamos a saída da camada densa um *dropout*, que é responsável pela regularização da

Figura 23: Arquitetura da rede neural recursiva



rede. No fim adicionamos mais uma camada densa com um *softmax* (Equação 4.3) como ativação para nos dar uma probabilidade normalizada.

$$\text{ReLU}(x) = \max(0, x) \quad (4.2)$$

$$\text{softmax}(x) = \frac{e^{x_j}}{\sum_{k=1}^n e^{x_k}}, \quad \text{para } j = 1, \dots, n \quad (4.3)$$

#### 4.2.1 Embeddings: camada de vetores de palavras e etiquetas

Essa camada é responsável por treinar os vetores de palavras. Seu funcionamento é bem simples: fizemos o *Backpropagation* até a camada de entrada, desse modo, podemos transformar índices para vetores usando uma tabela de pesquisa com uma função de mapeamento de um para um.

Como por exemplo, pode ser feita a transformação de  $[[2, 3], [5, 7]]$  em  $[[[0.1, 0.2], [0.25, 0.85]], [[0.53, 0.39], [0.22, 0.84]]]$ . Ou seja, damos como entrada uma matriz com formato  $m \times t$  e é retornado uma matriz com formato  $m \times t \times d$ . Como opção, podemos atribuir os pesos de cada índice (ou seja, o vetor da palavra) no momento da compilação e ajustá-los conforme o treinamento.

A camada *Word Embeddings* foi dividida em quatro partes: Uma parte é responsável por atribuir os vetores distribuídos de palavras; outra parte trata de prefixos e sufixos; e ainda há uma parte que trata de capitalização. Para cada uma dessas partes foi atribuído os pesos no momento da compilação, desse modo o vetor de cada *feature* fica ajustado à tarefa de POS Tagging.

### 4.2.2 Merge: camada de composição

É responsável por juntar os dois submodelos: modelo de vetores de palavras e de vetores de etiquetas. Isso é feito através de uma função de composição. No nosso caso foi usado uma função de soma. Desse modo, a matriz resultante continua com o mesmo formato.

Nessa camada, estamos combinando características do vetor de uma palavra com o respectivo vetor de uma classe gramatical associada a essa palavra.

### 4.2.3 Flatten: camada de concatenação

Simplemente faz a concatenação das linhas da matriz de entrada. Ou seja, vai transformar a matriz com formato  $m \times t \times d$  em uma matriz de duas dimensões com formato  $m \times t * d$ .

### 4.2.4 Dense: camada totalmente conectada

É uma simples camada neural onde cada neurônio da camada atual é ligado a todos os neurônios da próxima camada. Portanto, o formato da matriz resultante é  $m \times h_{dim}$ .

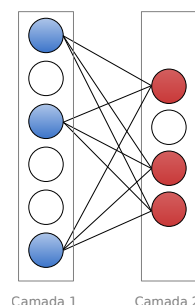
A primeira camada Dense é aplicada a função de ativação **ReLU** descrita na [Equação 4.2](#). A escolha dessa função de ativação foi feita de modo empírico.

Já na segunda camada Dense é aplicada a função de ativação *softmax* para nos dar probabilidades normalizadas.

### 4.2.5 Dropout: camada de regularização

*Dropout* é uma técnica introduzida em ([SRIVASTAVA et al., 2014](#)) para prevenir *overfitting* nas redes neurais. Quando uma rede neural é treinada, um neurônio tem probabilidade  $p$  (nesse caso  $p = 0.5$ ) de ser ignorado durante a computação do *forward propagation* e do *Backpropagation*. A [Figura 24](#) demonstra essa técnica.

Figura 24: Exemplo de *Dropout*



## 4.3 Treinamento

Nosso modelo recursivo baseia-se em escolher as palavras mais fáceis para serem etiquetadas primeiro. Essa heurística também é utilizada em (SHEN; SATTA; JOSHI, 2007) para o algoritmo de inferência.

Uma palavra é caracterizada como mais fácil quando a saída da rede neural para uma de suas etiquetas for a maior entre todas na sentença. Por exemplo, suponhamos que temos a sentença: `Computação é um curso legal!`, e que temos apenas 3 etiquetas possíveis: `substantivo`; `adjetivo`; `verbo`. Quando jogamos cada palavra na rede neural, temos a seguinte matriz mostrada na [Tabela 8](#).

Tabela 8: Exemplo de etiquetação

Palavra/Etiqueta	substantivo	adjetivo	verbo
Computação	0.6	0.2	0.2
é	0.7	0.2	0.1
um	0.1	0.6	0.3
curso	<b>0.8</b>	0.1	0.1
legal	0.4	0.4	0.2
!	0.5	0.4	0.1

Nesse caso, a palavra mais provável é `curso`, pois a etiqueta `substantivo` tem a maior probabilidade entre todas as etiquetas de todas as palavras. Formalmente, a palavra mais provável pode ser obtida através da [Equação 4.4](#).

$$\arg \max_i \left( \max_{0 \leq j \leq |\gamma|} (M_{i,j}) \right) \quad (4.4)$$

Treinamos o modelo por sentença, onde levamos em consideração duas entradas para a rede neural: A janela de palavras  $J_p$  e a janela de etiquetas  $J_c$  inicializada com um valor especial de “etiqueta desconhecida”. Além disso, inicializamos o conjunto  $Q$  de palavras já etiquetadas com o valor  $\emptyset$ . A cada passo do algoritmo, obtemos a palavra mais provável que ainda não foi etiquetada e sua respectiva etiqueta serve de contexto para a próxima palavra a ser prevista. Esse contexto é colocado em cada vetor em que essa palavra estava associada na janela de etiquetas. O processo continua até que o tamanho do conjunto  $Q$  seja igual o tamanho da sentença, ou seja, até que todas as palavras tenham sido classificadas.

Podemos representar o processo de obter a palavra mais provável que ainda não foi etiquetada através de uma modificação da [Equação 4.4](#), mostrada abaixo na [Equação 4.5](#).

$$mp(M) = \arg \max_{i \notin Q} \left( \max_{0 \leq j \leq |\gamma|} (M_{i,j}) \right) \quad (4.5)$$

E o processo de atualizar o contexto para prever a próxima palavra é mostrado na [Equação 4.6](#). Onde  $c_x$  representa a etiqueta prevista,  $x$  representa a entrada e  $I_t$  representa uma matriz com 1s na diagonal secundária com dimensão  $t \times t$ .

$$J_c[mp(x) - \lfloor t/2 \rfloor : mp(x) + \lfloor t/2 \rfloor][I_t] = c_{mp(x)} \quad (4.6)$$

O algoritmo 2 realiza todo esse processo.

---

**Algorithm 2** Algoritmo de Treinamento
 

---

```

1: procedure TREINAMENTO
2:   dado um conjunto de treinamento  $\{X, Y\}$ 
3:   for cada sentença  $S_n$  em  $\{X, Y\}$  do
4:      $J_p$  = janela de palavras da sentença  $S_n$ 
5:      $J_c[\cdot]$  = etiqueta desconhecida
6:      $Q = \{\}$ 
7:     while  $Q.size < s_t.size$  do
8:        $train\_on\_batch([J_p, J_c], Y_{S_n})$ 
9:        $M = predict\_on\_batch([J_p, J_c])$ 
10:      palavra_mais_provável =  $mp(M)$ 
11:      tag_mais_provável =  $J_c[palavra\_mais\_provável]$ 
12:       $slice = palavra\_mais\_provável - \lfloor t/2 \rfloor : palavra\_mais\_provável + \lfloor t/2 \rfloor$ 
13:       $J_c[slice][I_t] = tag\_mais\_provável$ 
14:       $Q.add(palavra\_mais\_provável)$ 
15:    end while
16:  end for
17: end procedure

```

---

### 4.3.1 Minimização da função de custo

A função de custo utilizada nesse modelo foi a *Categorical Cross-entropy* mostrada na [Equação 4.7](#), onde  $y$  é o nosso conjunto de classes gramaticais, e  $\hat{y}$  é a classe prevista pela rede.

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[ y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right] \quad (4.7)$$

Para minimizar essa função de custo, nós utilizamos um otimizador de segunda ordem baseado no Gradiente Descendente chamado Adadelta ([BENGIO; GOODFELLOW](#);



COURVILLE, 2015). O Adadelta leva em consideração informações de cada dimensão dos dados para atualizar a taxa de aprendizagem. Essa abordagem tem diversas vantagens:

- Taxa de aprendizagem dinâmica por dimensão.
- Pequena quantidade de computação por iteração.
- Hiperparâmetros escolhidos não afetam tanto o resultado.

A implementação do Adadelta utilizada foi a disponível na biblioteca Keras (CHOLLET, ).

### 4.3.2 Análise de complexidade temporal

A nossa implementação utilizou um simples vetor de *flags* de tamanho igual ao da sentença como sendo o conjunto de palavras já etiquetadas  $Q$ . Desse modo, gastamos  $O(1)$  para descobrir se uma palavra já foi classificada.

Para cada palavra  $w_i$  numa sentença  $S_n$ , treinamos a rede neural num *batch* de tamanho  $|S_n|$ . Como a rede neural depende de parâmetros para a convergência, vamos impor um limite aproximado de  $\theta((t * d * h_{dim})^3)$ , que equivale a multiplicações de matriz dos vetores de palavras concatenados com a camada oculta. Além disso, fizemos  $O(|S_n| * |\gamma|)$  iterações para buscar qual é a palavra mais provável. E por fim, atualizamos a janela de etiquetas fazendo  $O(t)$  iterações.

Supondo que todas as sentenças tenham tamanhos iguais (e portanto  $avg(S_i) = |S_n|$  para todo  $0 \leq i < m$ ):

$$O(|S_n| * ((t * d * h_{dim})^3 + (|S_n| * |\gamma|) + t))$$

Como  $t$  no pior dos casos é igual a  $|S_n|$ , temos:

$$O(|S_n| * ((|S_n| * d * h_{dim})^3 + (|S_n| * |\gamma|) + |S_n|))$$

$$O(|S_n|^4 * (d * h_{dim})^3 + |S_n|^2 * (|\gamma| + 1))$$

$$O(|S_n|^4 * (d * h_{dim})^3 + |S_n|^2 * |\gamma|)$$

$$O(|S_n|^4 * (d * h_{dim})^3)$$

Dentre essas variáveis, as únicas que não podemos controlar são  $|S_n|$  e  $|\gamma|$ , porém  $|\gamma|$  geralmente é um valor muito pequeno, e conseqüentemente, toda a computação recai sobre o tamanho da sentença. Com isso, podemos perceber que se o tamanho médio das sentenças for muito grande, então a computação vai ser demorada.

## 4.4 Predição

Para classificar uma palavra, nós seguimos a ideia apresentada por (SHEN; SATTA; JOSHI, 2007) e utilizamos um *beam search* sobre as classes mais prováveis da palavra mais provável (i.e. palavra mais fácil) em um determinado instante.

Inicializamos a janela de etiquetas com a etiqueta desconhecida e pegamos a primeira palavra mais provável e suas  $B$  tags mais prováveis. Após fazer isso, inicializamos  $B$  sequências para armazenar quais palavras foram etiquetadas e com quais tags elas foram etiquetadas. Inicializamos também a probabilidade de cada sequência  $s_b$  com o valor 1. E criamos uma lista de matrizes de predições  $M_b$  para cada  $0 \leq b < B$ .

Para cada sequência, nós utilizamos suas etiquetas já previstas como contexto na rede neural para realizar uma nova predição. Criamos uma fila de prioridade truncada com tamanho  $B$  para armazenar as próximas sequências. A prioridade da fila é a probabilidade da sequência vezes a probabilidade das etiquetas para a palavra mais provável. Após fazer isso, atualizamos nossas sequências e suas probabilidades para então voltar ao processo de predição para cada sequência. O processo termina quando todas as sequências estiverem completas, isso é, quando  $\sum_{b=0}^B s_b.size = B * |S_n|$ .

O algoritmo 3 sumariza esse processo.

Esse nosso algoritmo consegue usar contexto de palavras não vizinhas, desse modo espera-se resolver problemas de longa dependência.

Para entender melhor o funcionamento do algoritmo, sua execução está demonstrada na Figura 25.

Realizamos dois passos do algoritmo na Figura 25. Exemplificamos com uma sentença simples: “O livro está guardado”. Utilizamos  $t = 3$  e identificamos cada palavra na sentença pelo seu índice. Perceba que foi adicionado um símbolo especial de preenchimento chamado “<padding>” à esquerda e à direita da sentença. Nesse exemplo, definimos  $B = 2$  para simplificar a visualização. O círculo azul representa a rede neural já treinada de acordo com a seção anterior. O símbolo  $\cdot$  representa uma concatenação de matrizes em linha (perceba que isso não muda o objetivo da Equação 4.5).

- 1.1 Começamos inicializando as janelas de palavras  $J_p$  com os índices da sequência e a janela de etiquetas para cada  $0 \leq b < B$  com 0, que é um símbolo para “etiqueta desconhecida”, ou seja, fizemos  $J_{c_b} = 0$ .
- 1.2 Aplicamos apenas a janela com  $b = 0$  na rede neural e obtemos probabilidades normalizadas pela camada densa com ativação *softmax*.
- 1.3 Utilizamos a Equação 4.5 para obter qual é a palavra mais provável que ainda não foi etiquetada, e descobrimos que a palavra é “livro”. Após saber qual é a palavra mais

**Algorithm 3** Algoritmo de Predição

---

```

1: procedure PREDIÇÃO
2:   dado um conjunto para ser etiquetado  $\{X\}$ 
3:   dado o tamanho do beam  $B$ 
4:   for cada sentença  $S_n \in X$  do
5:      $J_p =$  janela de palavras da sentença  $S_n$ 
6:      $J_{c_b}[:] =$  etiqueta desconhecida
7:      $s_b[:] = []$ 
8:      $p_b[:] = 1$ 
9:      $M_b[:] = \text{predict\_on\_batch}([J_p, J_{c_b}])$ 
10:     $P = \text{PriorityQueue}(\text{beam\_size} = B)$ 
11:     $P.\text{push}(M_b[:])$ 
12:    while  $\sum_{b=0}^B s_b.\text{size} < B * |S_n|$  do
13:      mais_prováveis =  $P.\text{get\_items\_and\_pop}()$ 
14:      for palavra_mais_provável, tag_mais_provável,  $b \in$  mais_prováveis do
15:         $\text{slice} =$  palavra_mais_provável  $- \lfloor t/2 \rfloor : \text{palavra\_mais\_provável} + \lfloor t/2 \rfloor$ 
16:         $J_{c_b}[\text{slice}][I_t] = \text{tag\_mais\_provável}$ 
17:         $p_b^* = M_b[\text{palavra\_mais\_provável}][\text{tag\_mais\_provável}]$ 
18:         $s_b.\text{add}((\text{palavra\_mais\_provável}, \text{tag\_mais\_provável}))$ 
19:         $M_b = \text{predict\_on\_batch}([J_p, J_{c_b}])$ 
20:      end for
21:      próxima_mais_provável =  $mp(\max(M_b))$ 
22:      probs =  $zip(M_b[\text{próxima\_mais\_provável}], p_b)$ 
23:      for probabilidade_tag, probabilidade_sentença  $\in$  probs do
24:        nova_prioridade = probabilidade_tag * probabilidade_sentença
25:        novo_item =  $M_b[\text{próxima\_mais\_provável}]$ 
26:         $P.\text{push\_with\_priority}(\text{novo\_item}, \text{nova\_prioridade})$ 
27:      end for
28:      mais_prováveis =  $P.\text{get\_items}()$ 
29:       $\text{swap}(s_b, s_b[\text{mais\_prováveis}.b])$ 
30:       $\text{swap}(p_b, p_b[\text{mais\_prováveis}.b])$ 
31:    end while
32:  end for
33: end procedure

```

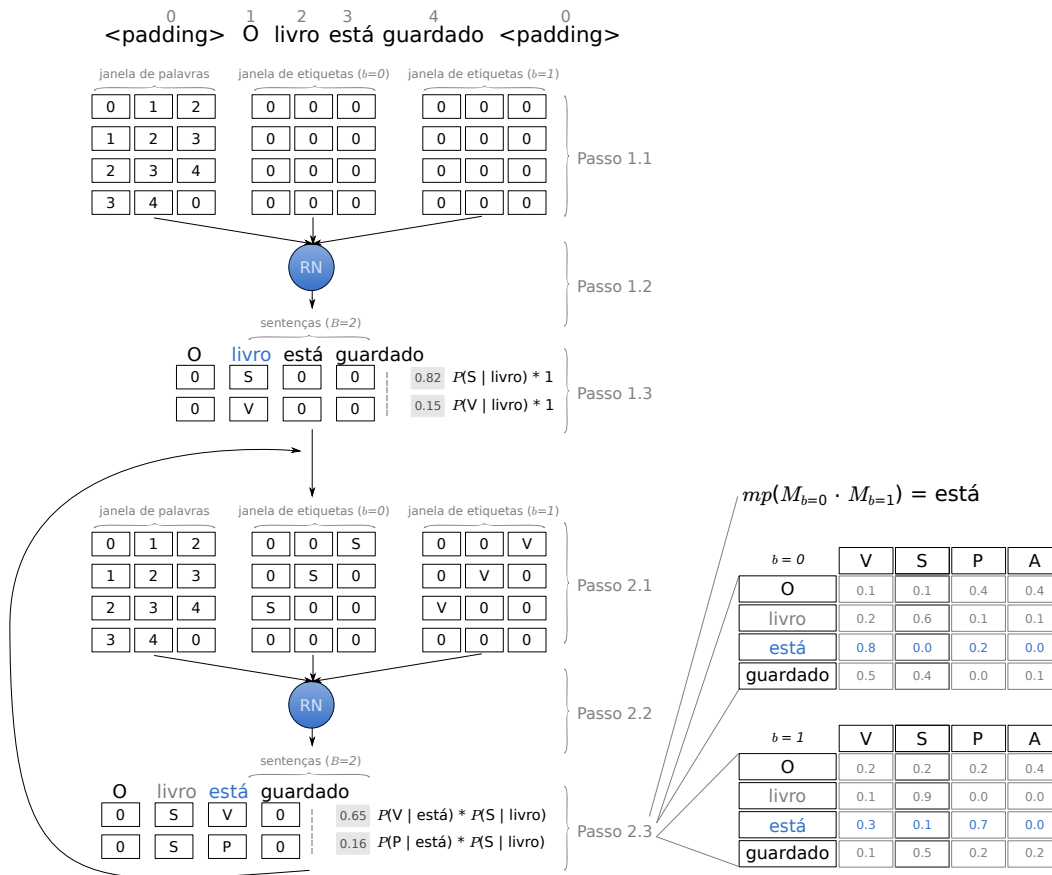
---

provável nós obtemos as  $B$  etiquetas mais prováveis e definimos a probabilidade de cada sequência  $b$  como sendo a própria probabilidade de emissão da etiqueta. Perceba que como não há sentenças anteriores, a probabilidade anterior da sentença era 100% e por isso simplesmente multiplicamos a probabilidade atual da sentença por 1.

2.1 Nesse momento já temos etiquetas para serem utilizadas como contexto. Cada  $J_{c_b}$  é atualizada com as tags previstas na  $b$ -ésima sequência.

2.2 Como agora há contexto, passamos cada  $(J_p, J_{c_b})$  para a rede neural e obtemos as predições da rede para cada  $b$ .

Figura 25: Exemplo de predição



2.3 Nesse momento é feito a busca pelas  $B$  melhores sequências. Primeiramente, obtemos qual é a palavra mais provável que ainda não foi etiquetada que está entre as duas tabelas de predições  $M_{b=0}$  e  $M_{b=1}$  utilizando a [Equação 4.5](#), e descobrimos que a palavra mais provável é “está”. Obtemos então, as  $B$  etiquetas mais prováveis de cada tabela  $M_b$  para a palavra “está”. Após isso multiplicamos as probabilidades de emissão das etiquetas com a probabilidade da sequência de onde elas vieram e selecionamos as  $B$  maiores probabilidades. Definimos as etiquetas com maiores probabilidades em ordem decrescente em cada sequência  $b$ . Perceba que no exemplo temos as seguintes  $B$  probabilidades em cada  $M_b$  para a palavra “está”:

$$P(V | \text{está}, \text{livro}, S) = P(V | \text{está}) * P(S | \text{livro}) = 0.8 * 0.82 = 0.65$$

$$P(P | \text{está}, \text{livro}, S) = P(P | \text{está}) * P(S | \text{livro}) = 0.2 * 0.82 = 0.16$$

$$P(P | \text{está}, \text{livro}, V) = P(P | \text{está}) * P(V | \text{livro}) = 0.7 * 0.15 = 0.10$$

$$P(V | \text{está}, \text{livro}, V) = P(V | \text{está}) * P(V | \text{livro}) = 0.3 * 0.15 = 0.04$$

Podemos expressar a  $P(c_i|w_i)$  como sendo a probabilidade da sequência onde  $w_i$  :  $c_i$  aparecem, obedecendo a ordem (veja que nesse caso  $P(c_3|w_3, w_2, c_2, w_1, c_1) \neq$

$P(c_3|w_3, w_1, c_1, w_2, c_2)$ ). Nesses casos, as probabilidades estarão armazenadas no vetor  $p_b$ .

Após computar as probabilidades e atualizar as  $B$  sequências mais prováveis, voltamos ao Passo 2.1 (pois temos contexto de etiquetas).

A principal diferença do nosso algoritmo para o apresentado por [Shen, Satta e Joshi \(2007\)](#) é que levamos em consideração etiquetas que estão em um contexto longe, que não são necessariamente vizinhas da palavra sendo analisada. Além disso, nós optamos por multiplicar as probabilidades das sequências com as probabilidades de emissões das etiquetas, já [Shen, Satta e Joshi \(2007\)](#) usa a operação de soma do contexto direito com o esquerdo e da emissão.

#### 4.4.1 Análise de complexidade temporal

Implementamos o algoritmo de predição utilizando um *heap* como fila de prioridades, e nesse *heap* mantemos apenas  $B$  itens.

A essência desse algoritmo é a mesma que a do algoritmo de treinamento: treinamos por sequência e sempre buscando a palavra mais provável para ser classificada primeiramente. A diferença é que agora estamos fazendo uma busca pelas  $B$  sequências de palavras/etiquetas mais prováveis, e então nossa complexidade é multiplicada pelo tamanho do *beam*. Além disso, para reduzir a complexidade do algoritmo nós trabalhamos com manipulação de índices, portanto a operação para calcular as probabilidades das sequências e de buscar o elemento na sequência são feitas em  $O(1)$ .

Fazendo as mesmas suposições que fizemos para o treinamento na [subseção 4.3.2](#):

$$O(|S_n|^4 * B * (d * h_{dim})^3 + \log(B))$$

Como  $B$  geralmente é um número pequeno (pois não queremos analisar todas as configurações possíveis),  $\log(B)$  se torna um número muito pequeno. Desse modo, temos:

$$O(|S_n|^4 * B * (d * h_{dim})^3)$$

Veja que a complexidade do algoritmo de predição seja maior que a do algoritmo de treinamento. Entretanto, o tempo de execução do treinamento é maior por dois motivos: Há um tempo extra para a atualização dos pesos via *Backpropagation* e  $m$  é muito maior para o treinamento (geralmente é 80% para treinamento e 20% para predição).

## 4.5 Implementação

Nós implementamos o modelo neural recursivo utilizando a linguagem Python 3.4.3 devido ao suporte da comunidade de aprendizagem profunda para bibliotecas dessa linguagem.

Utilizamos a biblioteca Numpy 1.10.1 para realizar eficientemente operações com matrizes. Utilizamos a biblioteca Keras 0.2.0 para a construção da rede neural descrita na [seção 4.2](#) e também para realizar a minimização da função de custo.

O código que define a arquitetura do modelo neural recursivo, e o código que realiza o treinamento e a predição podem ser vistos no [Apêndice A](#).

## 5 Modelo neural recorrente bidirecional

Neste capítulo será explicado o outro modelo proposto para resolver o problema de POS Tagging. Para isso, vamos falar primeiramente quais são os pré-processamentos feitos; depois vamos definir a arquitetura do modelo; explicar o algoritmo de treinamento e de predição; e por fim, comentar sobre a implementação do modelo.

Optamos por criar um modelo neural recorrente bidirecional, pois acreditamos que o uso de um contexto à direita pode influenciar na classificação, e que por usar GRU, a rede consegue aprender longas dependências.

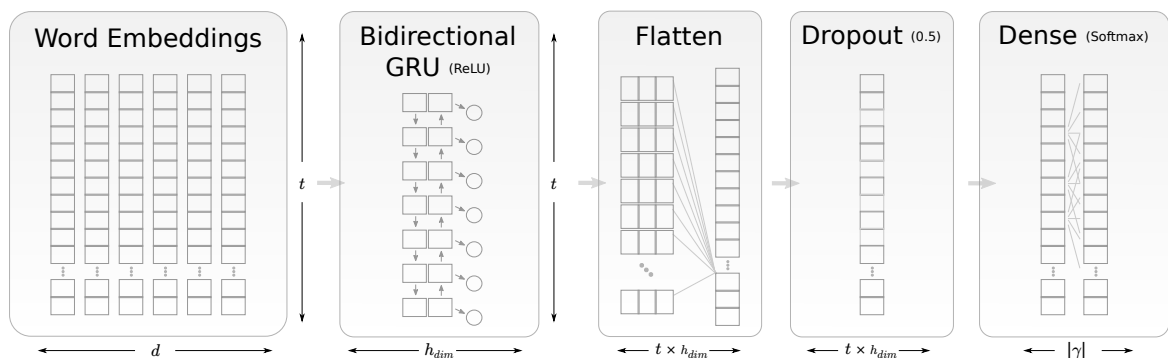
### 5.1 Pré-processamento

Fizemos o mesmo pré-processamento que no modelo neural recursivo. Porém nesse modelo nós usamos um tamanho de janela grande para que a rede possa aprender longas dependências e ainda assim obter um contexto através das palavras vizinhas.

### 5.2 Arquitetura

A arquitetura do modelo neural recorrente implementado pode ser vista na Figura 26.

Figura 26: Arquitetura da rede neural recorrente bidirecional



As diferenças na arquitetura do modelo recorrente bidirecional para o modelo recursivo é a inserção da GRU bidirecional e a retirada da camada de Embeddings para as etiquetas. Portanto, nessa seção vamos explicar apenas a camada recorrente bidirecional.

### 5.2.1 GRU: camada recorrente bidirecional

Conforme foi falado na [subseção 2.5.1](#), GRU é um tipo de rede neural que consegue aprender longas dependências entre termos numa sentença através da utilização de memória. No nosso caso, ela está recebendo uma janela de palavras  $J_p$  de tamanho  $t$ . Para cada palavra nessa janela é produzida uma representação de tamanho  $h_{dim}$ . Desse modo, o formato da matriz de saída é  $t \times h_{dim}$ .

Além disso, aplicamos a função de ativação [ReLU](#) ao final da computação da GRU. A escolha dessa função de ativação foi feita de modo empírico.

## 5.3 Treinamento

Para treinar o modelo, simplesmente tentamos minimizar a função de custo da rede neural.

### 5.3.1 Minimização da função de custo

A função de custo escolhida foi a *Categorical Cross-entropy* demonstrada na [Equação 4.7](#). Nós minimizamos essa função de custo utilizando o otimizador Adadelta, que também foi usado no modelo neural recursivo.

## 5.4 Predição

Para esse modelo pegamos a saída da rede para uma palavra como sendo a etiqueta prevista para ela.

## 5.5 Implementação

Implementamos esse modelo utilizando a mesma linguagem e as mesmas bibliotecas que foram usadas no modelo neural recursivo. O código da implementação da arquitetura, do treinamento e da predição podem ser vistos no [Apêndice B](#).



## 6 Testes e resultados

Neste capítulo será mostrado qual foi o ambiente de teste, quais foram os experimentos e os seus respectivos resultados serão discutidos. No fim do capítulo os nossos resultados serão comparados com os resultados dos trabalhos relacionados.

### 6.1 Ambiente de teste

#### 6.1.1 Bibliotecas

A implementação do modelo neural foi realizada utilizando a linguagem Python com as seguintes bibliotecas:

- Theano (0.7.0 dev): Biblioteca que define, otimiza e avalia expressões matemáticas eficientemente. Executa em GPU caso desejado.
- Keras (0.2.0): Biblioteca para aprendizagem profunda implementada sobre o Theano.
- Numpy (1.10.1): Biblioteca usada para aplicações números. Possui implementações eficientes para operações com matrizes.
- gensim (0.12.2): Biblioteca geralmente usada para aplicações de PLN. Possui um *parser* dos *dumps* da Wikipédia.

#### 6.1.2 Máquina

Todos os experimentos foram executados na máquina com as configurações abaixo:

- Sistema operacional: Ubuntu 14.04 LTS
- Processador: Intel Xeon X5690 CPU @ 3.47GHz × 24
- Memória: 64GB 1333 MHz DDR3
- Python 3.4.3

### 6.2 Pré-processamento

Nós transformamos palavras que têm uma ocorrência menor que a taxa de raridade em um símbolo especial chamado `<unknown>`. Também transformamos todos os dígitos

em “9” para diminuir a esparsidade. Para ambos os modelos, nós utilizamos *features* de capitalização, de prefixos e de sufixos.

Realizamos um treinamento não-supervisionado para gerar vetores distribuídos de palavras usando um *dump* de artigos da Wikipédia disponível em: <<https://dumps.wikimedia.org/ptwiki/latest/>>. No total haviam cerca de 44 milhões de *tokens* do *dump* e após o treino haviam 618966 vetores de palavras. Para o treino, consideramos a capitalização, acentuação e pontuação.

Com essas *features*, espera-se que o número de erros para palavras fora do vocabulário seja menor, pois desse modo temos informações relevantes da palavra mesmo que ela não esteja no conjunto de treinamento.

Para palavras que não estão no vocabulário do treinamento não-supervisionado, geramos um único vetor aleatório para elas (ou seja, as palavras compartilham o mesmo vetor).

### 6.3 Hiperparâmetros

Definimos como hiperparâmetros as variáveis que são mudadas em tempo de compilação pelo usuário e que influenciam na construção do modelo. Os hiperparâmetros escolhidos por nós se encontram na [Tabela 9](#).

Tabela 9: Hiperparâmetros para os modelos

Hiperparâmetro	Modelo recursivo	Modelo recorrente bidirecional
Tamanho da janela de palavras	5	11
Tamanho da janela de etiquetas	5	-
Tamanho dos vetores de palavras	50	50
Tamanho dos vetores de capitalização	7	7
Tamanho dos vetores de prefixos	5	5
Tamanho dos vetores de sufixos	5	5
Número de unidades ocultas	250	250
Taxa de <i>Dropout</i>	0.5	0.5
Taxa de aprendizagem	1.0	1.0
Épocas de treinamento	3	20
Tamanho do <i>beam</i>	3	-
Taxa de raridade	5	5

O tamanho de janela e de etiquetas foi escolhido de modo empírico. Seguimos (FONSECA; ROSA; ALUÍSIO, 2015) e usamos 50 para a dimensão dos vetores de palavras. O tamanho dos vetores de capitalização, de prefixos e de sufixos foram escolhidos de modo empírico. O número de unidades ocultas foi escolhido através de um *trade-off* de tempo de execução e acurácia. A taxa de *Dropout* foi escolhida de modo empírico. A taxa de aprendizagem foi deixada com o valor original do Keras. Definimos 3 épocas de

treinamento, porque o tempo de treinamento do modelo neural recursivo é alto, e além disso, não vimos diferença na diminuição do erro de validação após a terceira época. O tamanho do *beam* utilizado foi o sugerido em (SHEN; SATTA; JOSHI, 2007).

O tamanho da janela de palavras no modelo recorrente bidirecional é maior pois queremos um maior contexto para ter informações à direita. O modelo recorrente bidirecional tem mais épocas porque seu treinamento é mais rápido e com mais épocas o erro de validação decresce. Os valores “-” para o recorrente bidirecional significa que os respectivos hiperparâmetros não existem para esse modelo.

## 6.4 Resultados

Utilizamos três *corpus* para os experimentos: Mac-Morpho original (ALUÍSIO et al., 2003); Mac-Morpho revisado (FONSECA; ROSA; ALUÍSIO, 2015) (conhecido também como Mac-Morpho versão 3); Tycho Brahe (UNICAMP, 2010). Para todos esses *corpus*, dividimos eles em dois conjuntos: Um conjunto de treino (80%) e outro de validação (20%). Fizemos o uso de vetores distribuídos de palavras aprendidos de forma não-supervisionada usando o Word2Vec, Wang2Vec e os vetores treinados por Fonseca, Rosa e Aluísio (2015).

### 6.4.1 Acurácia

A acurácia mede a taxa de acertos e sua equação pode ser vista na [Equação 6.1](#), sendo  $m$  o número de exemplos de treinamento,  $\hat{y}$  a classe prevista pela rede neural,  $y$  a classe correta e a função  $equal(p, q)$  retorna 1 se  $p$  é igual a  $q$ , e 0 caso contrário.

$$\frac{1}{m} \sum_{i=1}^m equal(\hat{y}_i, y_i) \quad (6.1)$$

Foram obtidos resultados entre diferentes representações de palavras. Onde para cada representação de palavra foi calculada a acurácia entre palavras conhecidas, palavras desconhecidas (palavras que não estão no conjunto de treinamento e estão no conjunto de validação, considerando palavras raras como conhecidas), e também palavras ambíguas. Nós consideramos apenas a acurácia mais alta de uma determinada época sobre o conjunto de validação.

Calculamos também a acurácia em relação a sentença inteira, onde consideramos uma sentença como correta se todas as palavras na sentença foram etiquetadas corretamente, e incorreta caso contrário. Essa métrica se encontra na coluna chamada “Sentenças”.

Os resultados para o modelo neural recursivo podem ser vistos na [Tabela 10](#), [Tabela 11](#) e [Tabela 12](#) para o Mac-Morpho original, Mac-Morpho revisado e para o Tycho Brahe, respectivamente.

Tabela 10: Modelo neural recursivo: Acurácia sobre o Mac-Morpho original

Representação	Conhecidas (%)	Desconhecidas (%)	Ambíguas (%)	Total (%)	Sentenças (%)
Word2Vec	94.80	82.54	94.24	94.28	46.39
Wang2Vec	94.44	82.12	94.53	93.91	46.28
Fonseca	95.92	86.77	95.26	95.53	47.39

Tabela 11: Modelo neural recursivo: Acurácia sobre o Mac-Morpho revisado

Representação	Conhecidas (%)	Desconhecidas (%)	Ambíguas (%)	Total (%)	Sentenças (%)
Word2Vec	94.13	80.99	93.02	93.78	42.14
Wang2Vec	95.22	81.57	94.17	94.56	42.35
Fonseca	<b>96.12</b>	<b>88.32</b>	<b>96.44</b>	<b>95.79</b>	<b>47.28</b>

Tabela 12: Modelo neural recursivo: Acurácia sobre o Tycho Brahe

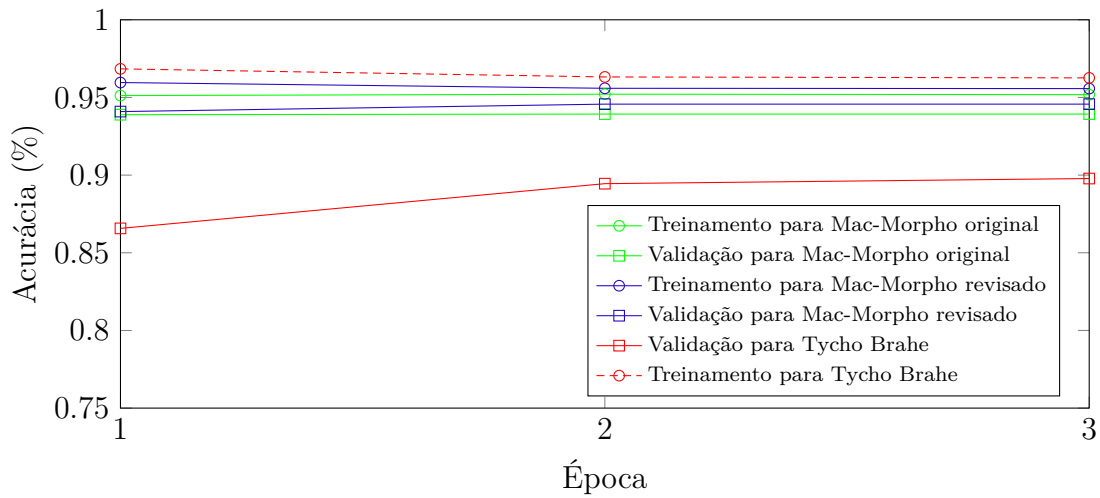
Representação	Conhecidas (%)	Desconhecidas (%)	Ambíguas (%)	Total (%)	Sentenças (%)
Word2Vec	90.27	48.12	90.81	88.82	22.93
Wang2Vec	91.23	48.77	90.86	89.78	23.47
Fonseca	89.74	47.75	89.24	88.31	19.54

O modelo recursivo teve bastante dificuldade para classificar palavras desconhecidas, o que achávamos que não deveria ter acontecido, já que há o contexto de etiquetas que ajuda no momento da classificação. Acreditamos que isso aconteceu devido a ocorrência de *underfitting*, uma vez que a acurácia de treinamento é baixa (para os padrões de POS Tagging) e a de validação é mais baixa ainda.

Através dos gráficos da acurácia de treinamento e validação pelo número de épocas para o modelo neural recursivo, mostrado na [Figura 27](#), podemos ver que ele sofre de *underfitting*, onde a acurácia de treinamento e validação são baixas. Apesar do número de épocas ser pequeno, é possível ver através da imagem que o modelo não aumenta significativamente a acurácia da segunda época para a terceira.

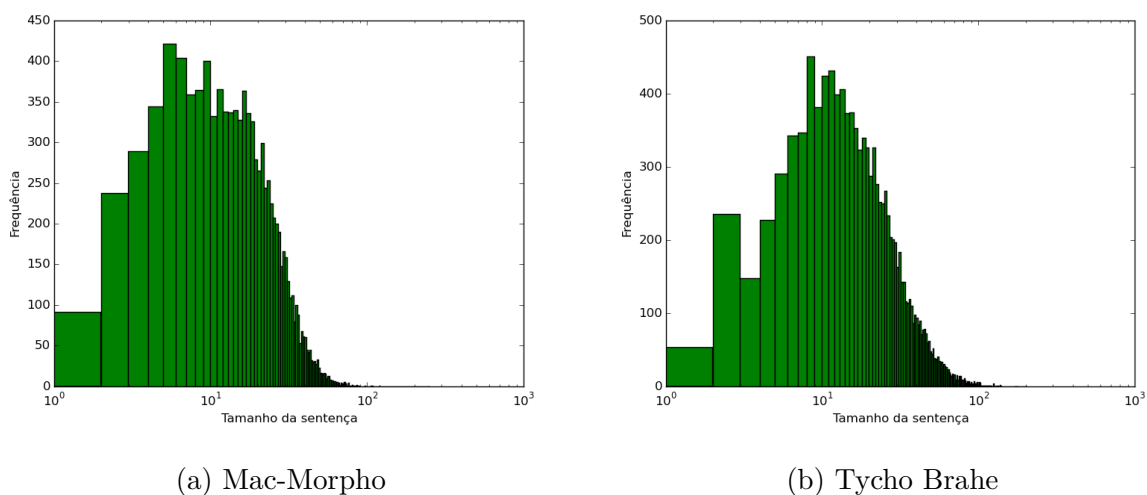
A acurácia obtida no Tycho Brahe foi baixa quando comparada aos trabalhos relacionados. Acreditamos que isso pode ser influenciado pelo fato do conjunto de etiquetas desse *corpus* ser maior, e também pelo fato do Tycho Brahe ter mais sentenças compridas do que o Mac-Morpho.

Figura 27: Acurácia de treinamento e validação para modelo recursivo usando o Wang2Vec



A Figura 28(a) e a Figura 28(b) mostram, respectivamente, a distribuição dos comprimentos das sentenças no conjunto de validação do Mac-Morpho e do Tycho Brahe. Nela, é possível perceber que o Tycho Brahe tem mais sentenças com comprimento maior que 50. Já o Mac-Morpho tem um maior número de sentenças com comprimento menor que 50. Desse modo, um erro cometido na fase inicial do algoritmo de predição é levado adiante e afeta as outras classificações

Figura 28: Distribuição dos comprimentos das sentenças



Os resultados do modelo neural recorrente bidirecional podem ser encontrados na Tabela 13, Tabela 14 e Tabela 15 para o Mac-Morpho original, Mac-Morpho revisado e para o Tycho Brahe, respectivamente.

O modelo recorrente bidirecional funciona muito bem para a tarefa de POS Tag-

Tabela 13: Modelo neural recorrente bidirecional: Acurácia sobre o Mac-Morpho original

Representação	Conhecidas (%)	Desconhecidas (%)	Ambíguas (%)	Total (%)	Sentenças (%)
Word2Vec	97.01	88.79	97.05	97.01	59.43
Wang2Vec	97.03	87.60	96.70	96.63	56.63
Fonseca	<b>97.60</b>	<b>92.63</b>	<b>97.25</b>	<b>97.37</b>	<b>66.38</b>

Tabela 14: Modelo neural recorrente bidirecional: Acurácia sobre o Mac-Morpho revisado

Representação	Conhecidas (%)	Desconhecidas (%)	Ambíguas (%)	Total (%)	Sentenças (%)
Word2Vec	96.92	87.28	96.30	96.45	<b>57.03</b>
Wang2Vec	97.33	90.03	96.50	96.99	56.04
Fonseca	<b>97.33</b>	<b>92.18</b>	<b>96.50</b>	<b>97.08</b>	56.77

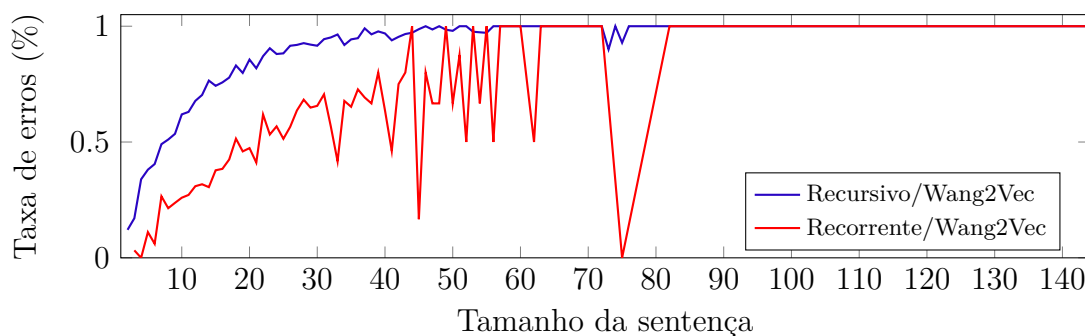
Tabela 15: Modelo neural recorrente bidirecional: Acurácia sobre o Tycho Brahe

Representação	Conhecidas (%)	Desconhecidas (%)	Ambíguas (%)	Total (%)	Sentenças (%)
Word2Vec	96.42	65.45	95.85	95.53	57.54
Wang2Vec	96.33	68.07	95.37	95.36	58.91
Fonseca	<b>96.80</b>	<b>73.39</b>	<b>96.09</b>	<b>96.00</b>	<b>64.80</b>

ging. Os resultados no Mac-Morpho original e no revisado mostram que nossas hipóteses (de longa dependência e contexto da direita) estavam corretas. Os resultados desses *córpus* ficaram muito próximos dos estados-da-arte sem requerer muito esforço para criação de *features* manualmente.

A acurácia para palavras fora do vocabulário no Tycho Brahe continuou deixando a desejar nesse modelo. Temos duas hipóteses do porquê disso ter ocorrido: A primeira é a questão do tamanho da sentença; A segunda é a questão dos vetores de palavras utilizados.

Figura 29: Taxa de erros por tamanho da sentença no Tycho Brahe



Para a primeira hipótese, a [Figura 29](#) mostra que, para ambos os modelos, a taxa de erros do Tycho Brahe tem um crescimento significativo conforme o tamanho da sentença aumenta, sendo que o recursivo comete mais erros que o bidirecional. Entretanto, a gente não vê uma causa do porquê disso ocorrer. A verdade é que vários fatores podem influenciar esse comportamento, como por exemplo os hiperparâmetros, a precisão de ponto flutuante utilizada, e até a próprio modo de treinamento/predição. A [Tabela 16](#) mostra o tamanho média das sentenças em cada *corp*us.

Tabela 16: Tamanho médio das sentenças

Conjunto	Mac-Morpho original (%)	Mac-Morpho revisado (%)	Tycho Brahe (%)
Treino	19	19	22
Teste	17	17	22

Com os dados da [Tabela 16](#) e usando as equações da análise de complexidade vistas no [Capítulo 4](#), é possível ver que o tempo de treinamento e de predição para o Tycho Brahe foi maior que para o Mac-Morpho. Na prática, o tempo de treinamento e predição do modelo recursivo foi cerca de oito vezes maior do que o do modelo bidirecional.

$$\frac{\text{número de vetores não encontrados}}{\text{número de palavras no vocabulário}} \quad (6.2)$$

Tabela 17: Taxa de vetores não encontrados

Representação	Mac-Morpho original (%)	Mac-Morpho revisado (%)	Tycho Brahe (%)
Word2Vec	33.92	27.74	78.52
Wang2Vec	33.92	27.74	78.52
Fonseca	39.33	32.12	93.98

Para a segunda hipótese, podemos levar em consideração a taxa de vetores não encontrados, descrita na [Equação 6.2](#), para cada *corp*us e representação de palavras utilizada, isso pode ser visto na [Tabela 17](#).

Perceba que a taxa de vetores não encontrados para o Tycho Brahe é muito maior que para os outros *corp*us. Além disso, é possível analisar que os vetores disponibilizados por [Fonseca, Rosa e Aluísio \(2015\)](#) obtiveram a melhor acurácia na maioria dos casos. Isso foi porque eles foram treinados num *corp*us com cerca de 240 milhões de *tokens* numa junção de artigos da Wikipédia e publicações do portal G1 (<<http://www.g1.com.br>>). Isso fez com que seus vetores ficassem mais precisos no contexto do Mac-Morpho (uma vez que o Mac-Morpho é uma compilação de textos jornalísticos publicados na Folha de São Paulo em 1994).

Podemos também analisar a taxa de ocorrência de vetores não encontrados através da [Equação 6.3](#). A [Tabela 18](#) mostra os valores dessa taxa para os três *córpus* utilizados para cada representação de palavras.

$$\frac{\text{ocorrência de vetores não encontrados}}{\text{número de palavras no conjunto de (treinamento } \cup \text{ validação)}} \quad (6.3)$$

Tabela 18: Taxa de ocorrência de vetores não encontrados

Representação	Mac-Morpho original (%)	Mac-Morpho revisado (%)	Tycho Brahe (%)
Word2Vec	21.01	17.11	18.42
Wang2Vec	21.01	17.11	18.42
Fonseca	2.14	1.72	6.22

Através desses dados, é possível perceber que, por mais que o número de vetores fora do vocabulário seja maior para a representação dos vetores disponibilizados por [Fonseca, Rosa e Aluísio \(2015\)](#), essa representação ainda tem uma taxa muito menor de ocorrência de vetores não encontrados. Isso significa que nosso *dump* da Wikipédia era bem distribuído, porém não continha palavras que ocorrem frequentemente nos *córpus* analisados. Essas tabelas fazem têm uma correlação direta com os resultados obtidos. Desse modo, acreditamos que isso foi o que mais influenciou os resultados.

Vale a pena comentar que as linhas dos Word2Vec e do Wang2Vec são iguais pois eles foram treinados sobre o mesmo *dump* da Wikipédia.

## 6.5 Comparação com trabalhos relacionados

Juntamos os melhores resultados de cada trabalho relacionado e comparamos com os nossos melhores resultados para cada *córpus*, isso pode ser visto na [Tabela 19](#). O melhor resultado para cada *córpus* está destacado em negrito.

Conforme discutido na seção anterior, um dos motivos que levaram [Fonseca, Rosa e Aluísio \(2015\)](#) conseguir bons resultados foi o tamanho do *dump* que utilizaram para treinar os vetores distribuídos, assim como o fato de terem treinados sobre textos jornalísticos beneficiou o resultado para o Mac-Morpho.

[Santos e Zadrozny \(2014\)](#) não utilizaram vetores distribuídos de palavras pré-treinados, mas criaram um modelo de palavras e de caracteres que ajudou para palavras fora do vocabulário.

Nossos melhores resultados foram atingidos usando o modelo neural recorrente bidirecional. Eles estão bem próximos dos melhores resultados para cada *córpus*, e levando



Tabela 19: Comparação da acurácia dos resultados com trabalhos relacionados

<i>Córpus</i>	Mac-Morpho original		Mac-Morpho revisado		Tycho Brahe	
	Todas(%)	FDV(%)	Todas(%)	FDV(%)	Todas(%)	FDV(%)
Kepler e Finger (2010)	-	-	-	-	96,29	71,60
Santos e Zadrozny (2014)	97.47	92.49	-	-	<b>97.17</b>	<b>86.58</b>
Fonseca, Rosa e Aluísio (2015)	<b>97.57</b>	<b>93.38</b>	<b>97.33</b>	<b>93.66</b>	96.93	84.14
Este trabalho	97.37	92.63	97.08	92.18	96.00	73.39

em consideração que nosso *dump* da Wikipédia era pequeno e não continha algumas palavras que ocorrem com frequência, o modelo bidirecional conseguiu ótimos resultados. Acreditamos que empregando um modelo de caracteres como feito por Santos e Zadrozny (2014) e aumentando o tamanho do *dump* nos ajude a levantar a acurácia.



# 7 Considerações finais

## 7.1 Conclusão

Este trabalho apresentou dois modelos neurais diferentes para solucionar o problema de POS Tagging. Esses dois modelos foram definidos e a complexidade de treinamento e predição do modelo neural recursivo foi analisada. Para os dois modelos, nós calculamos a acurácia sobre três diferentes *córpus*. Utilizamos vetores distribuídos de palavras treinados de modo não supervisionado para a representação das palavras. Isso fez com que a acurácia para palavras fora do vocabulário ficasse maior. Além disso, verificamos o impacto do tamanho da sentença para nossos piores resultados a fim de analisar os erros cometidos.

Através dos experimentos, foi possível perceber que o conjunto de treinamento utilizado para aprender as *word embeddings* tem uma grande influência nos resultados. Percebemos também que o modelo neural recursivo sofreu de *underfitting*. Os testes mostraram que o modelo neural recorrente bidirecional é mais eficiente e também mais rápido para treinar e classificar. Não foi possível alcançar o estado da arte, porém conseguimos o segundo melhor resultado para palavras fora do vocabulário no Mac-Morpho original.

Para a implementação deste trabalho, nós criamos uma ferramenta chamada DeepTagger, que está disponível publicamente no repositório do BitBucket, que pode ser acessado através do endereço: <<https://bitbucket.org/fabiokepler/deeptagger>>.

## 7.2 Trabalhos futuros

Pretendemos realizar mais testes com o modelo neural recorrente bidirecional, principalmente com um *dump* da Wikipédia maior. E também pretendemos criar novas técnicas a serem empregadas ao DeepTagger, como:

- Modelo de caracteres.
- Mecanismos de atenção.
- Suporte a outras representações distribuídas de vetores.

Para conseguir realizar mais testes no modelo neural recursivo, vamos paralelizar o código de treinamento e de predição.



# Referências

- AFONSO, S. et al. Floresta sintá (c) tica: A treebank for portuguese. In: *LREC*. [S.l.: s.n.], 2002. Citado na página 37.
- 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 2 vezes nas páginas 37 e 73.
- 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 4 vezes nas páginas 49, 50, 51 e 63.
- CHO, K. et al. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. Citado na página 54.
- CHOLLET, F. *Keras Framework*. Disponível em: <<http://keras.io/>>. Citado na página 63.
- COLLOBERT, R. Deep learning for efficient discriminative parsing. In: *International Conference on Artificial Intelligence and Statistics*. [S.l.: s.n.], 2011. Citado na página 23.
- 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 52 e 56.
- 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 3 vezes nas páginas 38, 39 e 55.
- 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 49.
- 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 37, 55 e 56.
- 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 12 vezes nas páginas 23, 37, 38, 39, 40, 55, 56, 72, 73, 77, 78 e 79.
- HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural computation*, MIT Press, v. 9, n. 8, p. 1735–1780, 1997. Citado na página 54.

- KEPLER, F. N. *Um etiquetador morfo-sintático baseado em cadeias de Markov de tamanho variável*. Dissertação (Mestrado) — Instituto de Matemática e Estatística da Universidade de São Paulo, 12/04/2005., 2005. Citado 2 vezes nas páginas 55 e 56.
- KEPLER, F. N.; FINGER, M. Variable-length markov models and ambiguous words in portuguese. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. *Proceedings of the NAACL HLT 2010 Young Investigators Workshop on Computational Approaches to Languages of the Americas*. [S.l.], 2010. p. 15–23. Citado 3 vezes nas páginas 55, 56 e 79.
- KOMBRINK, S. et al. Recurrent neural network based language modeling in meeting recognition. In: *INTERSPEECH*. [S.l.: s.n.], 2011. p. 2877–2880. Citado na página 53.
- LING, W. et al. Two/too simple adaptations of word2vec for syntax problems. Citado na página 40.
- 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 39.
- 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 41.
- MANNING, C. D. Part-of-speech tagging from 97% to 100%: is it time for some linguistics? In: *Computational Linguistics and Intelligent Text Processing*. [S.l.]: Springer, 2011. p. 171–189. Citado na página 25.
- MANNING, C. D.; SCHÜTZE, H. *Foundations of statistical natural language processing*. [S.l.]: MIT press, 1999. Citado na página 23.
- 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 23.
- 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 33.
- MIKOLOV, T. et al. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013. Citado na página 40.
- NG, A. *Course of Machine Learning*. [S.l.], 2015. Disponível em: <<https://www.coursera.org/learn/machine-learning/>>. Citado 9 vezes nas páginas 28, 33, 34, 35, 41, 45, 47, 48 e 49.
- OLAH, C. Conv nets: A modular perspective. 2014. Disponível em: <<http://colah.github.io/posts/2014-07-Conv-Nets-Modular/>>. Citado na página 52.
- OLAH, C. Understanding LSTM networks. 2015. Disponível em: <<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>>. Citado 2 vezes nas páginas 53 e 54.
- PENNSYLVANIA, U. of. *The Penn Treebank Project (2014)*. 2014. Disponível em: <<http://www.cis.upenn.edu/~treebank/>>. Citado na página 37.

- 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 5 vezes nas páginas 23, 55, 56, 78 e 79.
- SCHUSTER, M.; PALIWAL, K. K. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, IEEE, v. 45, n. 11, p. 2673–2681, 1997. Citado na página 53.
- SHEN, L.; SATTA, G.; JOSHI, A. Guided learning for bidirectional sequence classification. In: CITESEER. *ACL*. [S.l.], 2007. v. 7, p. 760–767. Citado 4 vezes nas páginas 61, 64, 67 e 73.
- SOCHER, R. *Deep Learning for Natural Language Processing*. 2015. Disponível em: <<http://cs224d.stanford.edu/>>. Citado 3 vezes nas páginas 38, 40 e 51.
- SOCHER, R. et al. Parsing natural scenes and natural language with recursive neural networks. In: *Proceedings of the 28th international conference on machine learning (ICML-11)*. [S.l.: s.n.], 2011. p. 129–136. Citado na página 54.
- SOCHER, R.; MANNING, C. D.; NG, A. Y. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In: *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*. [S.l.: s.n.], 2010. p. 1–9. Citado na página 54.
- SRIVASTAVA, N. et al. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, JMLR. org, v. 15, n. 1, p. 1929–1958, 2014. Citado na página 60.
- 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 38 e 41.
- UNICAMP. *Tycho Brahe Parsed Corpus of Historical Portuguese*. 2010. Disponível em: <<http://www.tycho.iel.unicamp.br/~tycho/corpus/en/index.html>>. Citado 3 vezes nas páginas 37, 38 e 73.
- 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 na página 55.





# Apêndices



# APÊNDICE A – Implementação do modelo neural recursivo

## A.1 Arquitetura

```

1 word_model = Sequential()
2 word_model.add(Embedding(input_dim, embeddings_size, weights=[embeddings_weights],
3                          input_length=window_size))
4
5 capitalization_model = Sequential()
6 capitalization_model.add(Embedding(capitalizations_vocab_dim, capitalization_length,
7                                   weights=[capitalization_weights], input_length=window_size))
8
9 prefix_model = Sequential()
10 prefix_model.add(Embedding(prefix_vocab_dim, prefix_size, weights=[prefixes_weights],
11                           input_length=window_size*prefixes_length))
12 prefix_model.add(Reshape((window_size, prefixes_length*prefix_size)))
13
14 suffix_model = Sequential()
15 suffix_model.add(Embedding(suffix_vocab_dim, suffix_size, weights=[suffixes_weights],
16                          input_length=window_size*suffixes_length))
17 suffix_model.add(Reshape((window_size, suffixes_length*suffix_size)))
18
19 model = Sequential()
20 model.add(Merge([word_model, capitalization_model, prefix_model, suffix_model], mode='concat'))
21 model.add(Flatten())
22 model.add(Dense(hidden_dim, activation='relu'))
23 model.add(Dropout(keep_dropout))
24 model.add(Dense(output_dim, activation='softmax'))
25
26 model.compile(loss='categorical_crossentropy', optimizer='adadelta', class_mode="categorical")

```

## A.2 Treinamento

```

1 word_samples = data_set.windowed_feature_vectors(word_features, window_size=word_window_size)
2 classes      = data_set.windowed_tag_sequences(window_size=1)
3 unvec_classes = to_vector(unvectorize(classes))
4 previous_tag_samples = np.ones((len(word_samples), tag_window_size))
5
6 for e in range(nb_epoch):
7
8     previous_tag_samples[:, :] = data_set.tokenizer.unknown_tag_index
9     j = 0
10
11     for i, sentence in enumerate(data_set.word_id_sentences):
12
13         k = j+len(sentence)
14         word_window = word_samples[j:k]

```

```

15     tag_window = previous_tag_samples[j:k]
16     classes_window = classes[j:k]
17     indexes_predicted = [-1 for _ in range(len(sentence))]
18     count = 0
19
20     while count < len(sentence):
21         samples = [word_window, tag_window]
22         model.train_on_batch(samples, classes_window)
23         predicts = model.predict_on_batch(samples)
24         bigger_id, bigger_value, bigger_index = get_bigger_not_yet_predicted(predicts, indexes_predicted)
25         update_k_window_with_value(tag_window, bigger_index, bigger_id)
26         count += 1
27
28     j += len(sentence)

```

### A.3 Predição

```

1 word_samples = data_set.windowed_feature_vectors(word_features, window_size=word_window_size)
2 classes = data_set.windowed_tag_sequences(window_size=1)
3 unvec_classes = to_vector(unvectorize(classes))
4 previous_tag_samples = np.ones((len(word_samples), beam_size, tag_window_size))
5 previous_tag_samples = previous_tag_samples * data_set.tokenizer.unknown_tag_index
6 predictions = np.ones(unvec_classes.shape) * data_set.tokenizer.unknown_tag_index
7 j = 0
8
9 for i, sentence in enumerate(data_set.word_id_sentences):
10
11     k = j+len(sentence)
12     word_window = word_samples[j:k]
13     tag_window = previous_tag_samples[j:k]
14     classes_window = classes[j:k]
15     beam_probs, beam_sequences, beam_predicts = [], [], []
16
17     for b in range(beam_size):
18         beam_probs.append(1)
19         beam_sequences.append([-1 for _ in range(len(sentence))])
20         samples = [word_window, tag_window[:, b]]
21         beam_predicts.append(model.predict_on_batch(samples))
22
23     b = 0 # all beam_predicts are equal at this time
24     word_index = beam_predicts[b].max(axis=-1).argmax()
25     bigger_window_index = word_index + j
26
27     pq = PriorityQueue() # heap
28     for tag, prob_tag in enumerate(beam_predicts[b][word_index]):
29         item = (bigger_window_index, tag, b, 1, [])
30         prior = prob_tag
31         pq.push_and_trunc(item, prior, beam_size)
32
33     for count in range(len(sentence)):
34
35         pq.remake_beans()
36         bigger_items = pq.get_items()
37         prox_index, prox_value = 0, 0
38

```

```

39     for bigger_window_index, bigger_id, b, _, _ in bigger_items:
40         word_index = bigger_window_index - j
41         bigger_value = beam_predicts[b][word_index, bigger_id]
42
43         # tag_window for b
44         for bi, wi in enumerate(beam_sequences[b]):
45             if wi != -1:
46                 for l in range(tag_window_size):
47                     t = bi + l - tag_window_size // 2
48                     if t >= 0 and t < tag_window.shape[0]:
49                         tag_window[t, b, l] = bigger_id
50
51         samples = [word_window, tag_window[:, b]]
52         beam_probs[b] *= bigger_value
53         beam_sequences[b][word_index] = bigger_id
54         beam_predicts[b] = model.predict_on_batch(samples)
55
56         for iw, prob_bigger_word in enumerate(beam_predicts[b].max(axis=-1)):
57             if not_in_list_index(beam_sequences[b], iw) and prob_bigger_word > prox_value:
58                 prox_value = prob_bigger_word
59                 prox_index = iw
60
61         bigger_window_index = prox_index + j
62         for b in range(beam_size):
63             for it, prob_tag in enumerate(beam_predicts[b][prox_index, :]):
64                 prior = prob_tag * beam_probs[b]
65                 item = (bigger_window_index, it, b, beam_probs[b], beam_sequences[b])
66                 pq.push_and_trunc(item, prior, beam_size)
67
68         items = pq.get_items_and_back() # get items in reverse order and put them back in heap
69         for b, (bigger_window_index, bigger_id, b_pred, b_prior, b_seq) in enumerate(items):
70             beam_sequences[b] = b_seq[:] # make a copy
71             beam_probs[b] = float(b_prior)
72
73         max_predict_index = np.argmax(np.sum(np.array(beam_sequences) == unvec_classes, axis=-1))
74         for p in range(j, k):
75             predictions[p] = beam_sequences[max_predict_index][p-j]
76
77         j += len(sentence)
78
79     return predictions

```



# APÊNDICE B – Implementação do modelo neural recorrente bidirecional

## B.1 Arquitetura

```

1 model = Graph()
2
3 # WORD MODEL
4 model.add_input(name='word_input', input_shape=(window_size, ), dtype=int)
5 model.add_node(Embedding(input_dim, embeddings_size,
6     weights=[embeddings_weights],
7     input_length=window_size),
8     name='word_embedding',
9     input='word_input')
10
11 # CAPITALIZATION MODEL
12 model.add_input(name='capitalization_input', input_shape=(window_size, ), dtype=int)
13 model.add_node(Embedding(capitalizations_vocab_dim, capitalization_length,
14     weights=[capitalization_weights],
15     input_length=window_size),
16     name='capitalization_embedding',
17     input='capitalization_input')
18
19 # PREFIX MODEL
20 model.add_input(name='prefix_input', input_shape=(window_size, ), dtype=int)
21 model.add_node(Embedding(prefix_vocab_dim, prefix_size,
22     weights=[prefixes_weights],
23     input_length=window_size*prefixes_length),
24     name='prefix_embedding_concatened', input='prefix_input')
25 model.add_node(Reshape((window_size, prefixes_length*prefix_size)),
26     name='prefix_embedding',
27     input='prefix_embedding_concatened')
28
29 # SUFFIX MODEL
30 model.add_input(name='suffix_input', input_shape=(window_size, ), dtype=int)
31 model.add_node(Embedding(suffix_vocab_dim, suffix_size,
32     weights=[suffixes_weights],
33     input_length=window_size*suffixes_length),
34     name='suffix_embedding_concatened',
35     input='suffix_input')
36 model.add_node(Reshape((window_size, suffixes_length*suffix_size)),
37     name='suffix_embedding',
38     input='suffix_embedding_concatened')
39
40 # FINAL MODEL
41 model.add_node(GRU(hidden_dim, return_sequences=True, activation='relu'),
42     name='forward',
43     inputs=['word_embedding', 'capitalization_embedding', 'suffix_embedding', 'prefix_embedding'],
44     merge_mode='concat')
45 model.add_node(GRU(hidden_dim, return_sequences=True, activation='relu', go_backwards=True),

```

```

46     name='backward',
47     inputs=['word_embedding', 'capitalization_embedding', 'suffix_embedding', 'prefix_embedding'],
48     merge_mode='concat')
49 model.add_node(Dropout(keep_dropout), name='dropout', inputs=['forward', 'backward'], merge_mode='sum')
50 model.add_node(Flatten(), name='flatten', input='dropout')
51 model.add_node(Dense(output_dim, activation='softmax'), name='softmax', input='flatten')
52 model.add_output(name='output', input='softmax')
53
54 model.compile('adadelta', {'output': 'categorical_crossentropy'})

```

## B.2 Treinamento

```

1 window = data_set.windowed_word_sequences(window_size=window_size)
2 classes = data_set.windowed_tag_sequences(window_size=1)
3 capitalizations, prefixes, suffixes = get_manual_features(window, data_set.tokenizer.indexes_to_words)
4
5 val_window = validation_data_set.windowed_word_sequences(window_size=window_size)
6 val_classes = validation_data_set.windowed_tag_sequences(window_size=1)
7 val_capitalizations, val_prefixes, val_suffixes = get_manual_features(val_window,
8     data_set.tokenizer.indexes_to_words)
9
10
11 callbacks.append(ModelCheckpoint(filepath=weights_file, verbose=1, save_best_only=True))
12 callbacks.append(TagAccuracy(validation_data, eval_func=calculate_accuracy))
13
14 train_data = {'word_input':window, 'capitalization_input':capitalizations,
15     'prefix_input':prefixes, 'suffix_input':suffixes, 'output':classes}
16
17 validation_data = {'word_input':val_window, 'capitalization_input':val_capitalizations,
18     'prefix_input':val_prefixes, 'suffix_input':val_suffixes, 'output':val_classes}
19
20 model.fit(train_data, nb_epoch=nb_epoch, batch_size=batch_size, validation_split=validation_split,
21     validation_data=validation_data, callbacks=callbacks)

```

## B.3 Predição

```

1 window = data_set.windowed_word_sequences(window_size=window_size)
2 capitalizations, prefixes, suffixes = get_manual_features(window, data_set.tokenizer.indexes_to_words)
3 data = {'word_input':window, 'capitalization_input':capitalizations,
4     'prefix_input':prefixes, 'suffix_input':suffixes}
5 predicted_classes = np.argmax(model.predict(data, batch_size=batch_size, verbose=1)['output'], axis=-1)
6 return predicted_classes

```