



THIAGO SOTORIVA LERMEN

CURSO DE MACHINE LEARNING

PHOTO BY PIETRO JENG ON UNSPLASH

Universidade Federal Do Rio Grande Do Sul
Instituto De Informática

CURSO DE MACHINE LEARNING

Autor: Thiago Sotoriva Lermen
PET Computação UFRGS

Baseado em:

- Machine Learning (Coursera) | Stanford University [14]
- Deep Learning Lecture Series 2020 | DeepMind & University College London [20]
- Reinforcement Learning Course | DeepMind & University College London [11]
- CS229: Machine Learning | Stanford University [13]
- CS230: Deep Learning | Stanford University [2]
- CS231n: Convolutional Neural Networks for Visual Recognition | Stanford University [7]
- CS224n: Natural Language Processing with Deep Learning | Stanford University [6]
- CS234: Reinforcement Learning | Stanford University [5]



Agradecimentos

Agradeço aos meus colegas do PET Computação UFRGS, em especial Bernardo Beneduzi Borba, Eduardo Fantini e Victoria Avelar Duarte pelo apoio, incentivo, revisões e ideias para a construção deste curso.

Agradeço à tutora do PET Computação UFRGS, Érika Fernandes Cota, pelo apoio, conhecimento e paciência, tornando possível este projeto. Agradeço, também, aos colegas do Instituto De Informática, em especial, ao Gabriel Couto Domingues e Garrenlus de Souza pelas críticas, revisão e incentivo.

Agradeço à Rosália Galiazzzi Schneider e ao Leonardo Piletti Chatain pela mentoria, conhecimento, apoio, críticas, sugestões e incentivos.

Por final, quero agradecer à minha família, à minha namorada Raya Allawy, ao meu amigo Arthur Carvalho Balejo pelo apoio e incentivo, e a todos que, com o tempo depositado, críticas e incentivo, tornaram possível a realização deste projeto o qual, com certeza, será muito importante para o desenvolvimento acadêmico de futuros estudantes com interesse no campo de Inteligência Artificial.

Sumário

I	Introdução	7
1	Visão geral	7
1.1	O que é <i>Machine Learning</i> ?	7
1.2	O que inteligência?	7
1.3	<i>Supervised Learning</i>	8
1.4	<i>Unsupervised Learning</i>	8
II	Supervised Learning	10
2	Representação de modelos	10
3	Regressão linear	10
4	Função Custo (<i>Cost Function</i>)	11
5	Gradiente Descendente (<i>Gradient Descent</i>)	13
5.1	Algoritmo Gradiente Descendente	13
5.2	Gradiente Descendente para Regressão Linear	14
6	Álgebra linear	14
6.1	Matrizes e vetores	14
6.2	Operações básicas com matrizes e vetores	15
6.3	Multiplicação entre matrizes e vetores	15
6.4	Multiplicação entre duas matrizes	15
6.5	Matriz identidade, inversa e transposta	16
7	Regressão Linear com múltiplas variáveis	16
7.1	Múltiplos parâmetros de entrada	16
7.2	Gradiente Descendente com múltiplas variáveis	17
8	Equação Normal (<i>Normal Equation</i>)	18
9	Classificação	18
9.1	Problemas de classificação	18
9.2	Representação da hipótese	19
9.3	Limite de decisão (<i>Decision Boundary</i>)	20
9.4	Classificação Multiclasse	20
10	Regressão Logística (<i>Logistic Regression</i>)	21
10.1	Função Custo (<i>Cost Function</i>)	21
10.2	Gradiente Descendente para Regressão Logística	22

11 Underfitting e overfitting	22
11.1 Função custo em casos de <i>overfitting</i>	23
11.2 Regressão linear regularizada	24
11.3 Regressão logística regularizada	25
III Redes Neurais	26
12 Redes Neurais: Representação	26
12.1 Definição básica	26
12.2 Hipótese não-linear	26
12.3 Os Neurônios e o Cérebro	26
12.4 Representação do modelo	27
12.5 Aplicações	29
12.6 Classificação Multiclasse	31
13 Redes Neurais: Aprendizado	32
13.1 Estrutura básica	32
13.1.1 Camada linear	32
13.1.2 Camada de ativação: <i>Sigmoid layer</i>	33
13.1.3 <i>Cross entropy loss</i>	33
13.1.4 Camada de ativação: <i>Softmax layer</i>	33
13.1.5 Camada de ativação: <i>Rectified Linear layer</i> (ReLU)	34
13.1.6 Camada de ativação: <i>Hyperbolic tangent</i>	34
13.2 Função Custo (<i>Cost Function</i>)	35
13.3 <i>Backpropagation Algorithm</i>	35
13.3.1 <i>Forward pass</i>	36
13.3.2 <i>Backward pass</i>	37
13.3.3 Algoritmo	39
13.4 Otimizadores	40
13.5 Verificação do gradiente	40
13.6 Inicialização aleatória	41
13.7 Organização do conhecimento	41
14 Aplicação de algoritmos de <i>machine learning</i>	41
14.1 Valoração de algoritmos de aprendizagem	41
14.2 Curvas de aprendizado	43
14.2.1 <i>High Bias</i>	43
14.2.2 <i>High Variance</i>	44
14.3 Decisões a serem tomadas	44
14.4 Diagnosticando Redes Neurais	45
14.5 <i>Support Vector Machines</i> (SVMs)	45
15 Naive Bayes	46

IV	<i>Unsupervised Learning</i>	47
16	<i>Clustering</i>	47
16.1	<i>K-Means Algorithm</i>	47
16.2	Otimização	48
16.3	Inicialização	49
17	<i>K-Nearest Neighbors</i>	49
17.1	Algoritmo	50
17.2	Escolhendo o valor correto para K	50
18	Redução de dimensionalidade	50
18.1	Compressão de dados	50
18.2	Visualização	51
18.3	Análise do componente principal (PCA)	51
19	Aprendizado por reforço (<i>Reinforcement learning</i>)	52
19.1	Visão geral	52
19.2	<i>Exploration e Exploitation</i>	53
19.3	<i>Markov Process</i>	53
19.3.1	Propriedade de Markov	53
19.3.2	Cadeia de Markov	53
19.4	<i>Markov Decision Processes (MDPs)</i>	54
19.4.1	Busca pela política ótima com MDP	55
19.5	Monte-Carlo e <i>Temporal-Difference Learning</i>	55
19.5.1	Valorização de Monte-Carlo	55
19.5.2	TD <i>Learning</i>	56
19.6	<i>Q-Learning</i>	56
V	Otimizações no Aprendizado	58
20	Detectção de anomalias	58
20.1	Motivação	58
20.2	Distribuição Gaussiana	58
20.3	Algoritmo	59
21	Aprendizado em larga escala	60
21.1	<i>Batch Normalization</i>	60
21.2	<i>Stochastic Gradient Descent (SGD)</i>	61
21.3	<i>Mini-batch Gradient Descent</i>	63
21.4	<i>SGD + Momentum</i>	64
21.5	<i>Nesterov Accelerated Gradient (NAG)</i>	65
21.6	<i>AdaGrad</i>	66
21.7	<i>Adam</i>	66

21.8 <i>Mapreduce e Paralelismo de Dados</i>	68
VI Tópicos avançados	69
22 Redes neurais convolucionais (<i>Convolutional neural networks</i>)	69
22.1 Visão geral	69
22.2 Camadas de uma ConvNet	69
22.3 Técnicas de otimização de treino	72
22.3.1 <i>Data Augmentation</i>	72
22.3.2 Modelos pré-treinados	73
22.3.3 <i>Fine tuning</i>	73
22.4 Detecção e Segmentação	73
22.4.1 Segmentação Semântica	74
22.4.2 Localização	75
22.4.3 Detecção de Objetos	75
22.4.4 Segmentação de Instâncias	77
23 Redes neurais recorrentes (<i>Recurrent neural networks</i>)	77
23.1 Visão geral	77
23.2 Vetores de palavras (<i>word embeddings</i>)	78
23.2.1 <i>One-hot encoding</i>	80
23.2.2 Word2Vec	80
23.3 Arquitetura de uma RNN	83
23.4 Aplicações	83
23.4.1 <i>One-to-one</i>	84
23.4.2 <i>One-to-many</i>	84
23.4.3 <i>Many-to-one</i>	84
23.4.4 <i>Many-to-many</i>	85
23.5 Função custo (<i>Cost function</i>)	85
23.6 <i>Backpropagation</i>	85
23.7 Funções de ativação e propriedades	86
23.8 Gradiente de desaparecimento e explosão	86
23.9 <i>Long Short-Term Memory (LSTM)</i>	87
23.10 <i>Bidirectional & Multi-layer RNNs</i>	88
23.11 <i>Attention</i>	89
23.12 Redes neurais convolucionais para NLP	92
24 Transformers	92
24.1 <i>Positional encoding</i>	94
24.2 <i>Encoder</i>	94
24.2.1 <i>Self-Attention</i>	94
24.2.2 <i>Multi-head attention</i> e geração da saída	96
24.3 <i>Decoder</i>	97

24.3.1 <i>Masked Multi-head attention</i>	98
24.3.2 <i>Encoder-Decoder attention</i>	99
25 Modelos generativos (<i>Generative models</i>)	99
25.1 Visão geral	99
25.2 Aplicações	99
25.3 <i>Auto-Regressive Models</i>	100
25.3.1 PixelRNN	100
25.3.2 PixelCNN	101
25.4 <i>Variational Autoencoders (VAE)</i>	102
25.5 <i>Generative Adversarial Networks (GANs)</i>	104
25.5.1 Treinamento: Jogo de dois jogadores	105
25.5.2 <i>Deep Convolutional GANs</i>	107
26 Deep Q-Learning	108
27 Busca de Monte Carlo	110
27.1 <i>Uninformed search</i>	110
27.2 Monte Carlo Tree Search	111

Parte I

Introdução

1 Visão geral

Nesta seção serão introduzidos os conceitos básicos para fazer uma máquina aprender usando dados, sem, necessariamente, ser explicitamente programada.

1.1 O que é *Machine Learning*?

Machine Learning (*M.L.*) é um campo de estudo da área de Inteligência Artificial (I.A.) que dá ao computador a habilidade de aprender ser explicitamente programado [18].

Em uma definição mais moderna, "Um programa de computador aprende com a experiência E com relação a alguma classe de tarefas T e medida de desempenho P, se seu desempenho nas tarefas T, conforme medido por P, melhora com a experiência E." [12].

Em geral, qualquer problema de *M.L.* pode ser atribuído a uma dessas duas classificações gerais:

- Aprendizado Supervisionado (*Supervised Learning*);
- Aprendizado Não-Supervisionado (*Unsupervised Learning*)

Antes de detalharmos cada uma dessas classificações, é importante definir como medimos a inteligência de uma máquina, quando tratamos de *machine learning*. Essa definição será especificada em detalhes na seção seguinte.

1.2 O que inteligência?

Uma dúvida que pode surgir ao definirmos *machine learning* é "O que é inteligência?". Essa dúvida se tornou muito comum em diversas áreas de pesquisa relacionadas à IA.

Recentemente, a definição usada é a seguinte: "A inteligência mede a capacidade de um agente de atingir objetivos em uma ampla variedade de ambientes" [10]. Essa definição implica na construção de um modelo matemático para a inteligência de uma máquina. Com essa definição podemos comparar máquinas inteligentes através de dados concretos - números. A seguir, está definida a medida de inteligência de uma máquina.

$$\Upsilon(\pi) := \sum_{\mu \in E} 2^{-K(\mu)} V_\mu^\pi$$

Medida de inteligência Soma sobre os ambientes Penalidade de complexidade Valor alcançado

A equação acima é a definição formal da inteligência de uma máquina, também conhecida como "inteligência universal" de um agente - máquina - π . Ela descreve a medida de inteligência, $\Upsilon(\pi)$, de um agente π como o somatório dos valores de pontuação de inteligência, V_μ^π , do agente π em cada um dos ambientes μ . Para cada um desses ambientes define-se um valor de distribuição de probabilidade universal, $2^{-K(\mu)}$, onde K é a função de complexidade de Kolmogorov [10], que penaliza a complexidade de cada um dos ambientes. .

1.3 *Supervised Learning*

No método de aprendizado supervisionado, é dado como entrada um conjunto de dados que já temos conhecimento do resultado. Então, nosso objetivo nesse tipo de problema é treinar uma máquina, a fim de fazer com que ela generalize esses resultados para entradas que ainda não foram vistas.

Problemas de aprendizado supervisionado são separados em problemas de classificação e regressão. No problema de classificação, tentamos prever um resultado em uma saída discreta, ou seja, tentamos mapear variáveis de entrada - em valores complexos - para categorias mais simples e generalizadas. Nos problemas de regressão, tentamos prever resultados em uma saída contínua, ou seja, tentamos mapear as variáveis de entrada para uma função contínua.

Podemos exemplificar um problema de regressão da seguinte forma: dado uma foto de uma pessoa, tentamos prever a sua idade com base na imagem. De outra maneira, podemos exemplificar um problema de classificação da seguinte forma: dado um paciente com um tumor, tentamos prever se o tumor é maligno ou benigno.

Indo um pouco mais fundo, um dos maiores exemplos dessa área é a base de dados MNIST [9] e processamento de linguagem natural *Natural Language Processing* através de *Generative Pre-trained Transformer 2* (GPT-2) [15] representados na Figura 1. Essa base de dados foi usada para a implementação de um algoritmo de reconhecimento de dígitos manuscritos com a utilização de redes neurais.

1.4 *Unsupervised Learning*

O aprendizado não-supervisionado nos possibilita aproximar problemas que temos pouca ou nenhuma ideia de como seus resultados devem ser. Nós podemos derivar esta estrutura utilizando o método de *clustering* nos dados baseados nos relacionamentos com as variáveis nos dados

Podemos exemplificar a técnica de *clustering* da seguinte forma: ao pegar uma coleção de um milhão de genes diferentes, encontramos uma forma de agrupá-los, de modo que os genes de cada grupo compartilhem determinada semelhança, como por exemplo, tempo de vida, localização, funções, etc.

Recentemente, foi desenvolvido pela DeepMind uma inteligência artificial que joga o jogo Go no mais alto nível de performance. AlphaGo [19] combina diferentes técnicas de redes neurais profundas para realizar o treinamento. Uma rede neural escolhe o melhor movimento a ser realizado a partir de um estado de tabuleiro e a outra realiza uma previsão do vencedor do jogo através de modelos probabilísticos. Utiliza-se de métodos de *unsupervised learning* como por exemplo *reinforcement learning* e *Monte Carlo Search* que escolhe a melhor jogada a partir de jogadas prévias. Na Figura 2 está representado um modelo de *unsupervised learning* utilizado na implementação do AlphaGo.

SYSTEM PROMPT (HUMAN-WRITTEN)	
<i>In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.</i>	
MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)	
The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.	
Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.	
Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.	
Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez.	
Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them - they were so close they could touch their horns.	

- (a) Banco de dados MNIST para reconhecimento de dígitos manuscritos. Muito usado em sistemas de processamento de imagens. Essa base de dados foi usada para a implementação de um algoritmo de reconhecimento de dígitos manuscritos com a utilização de redes neurais
- (b) Representação de uma das funções da inteligência GPT-2 desenvolvida pela OpenAI (2019), onde a partir de um texto de entrada. É uma inteligência que foi treinada para traduzir textos, responder perguntas, resumir passagens de textos e gerar um texto de saída que se similariza com o nível de escrita humano.

Figura 1

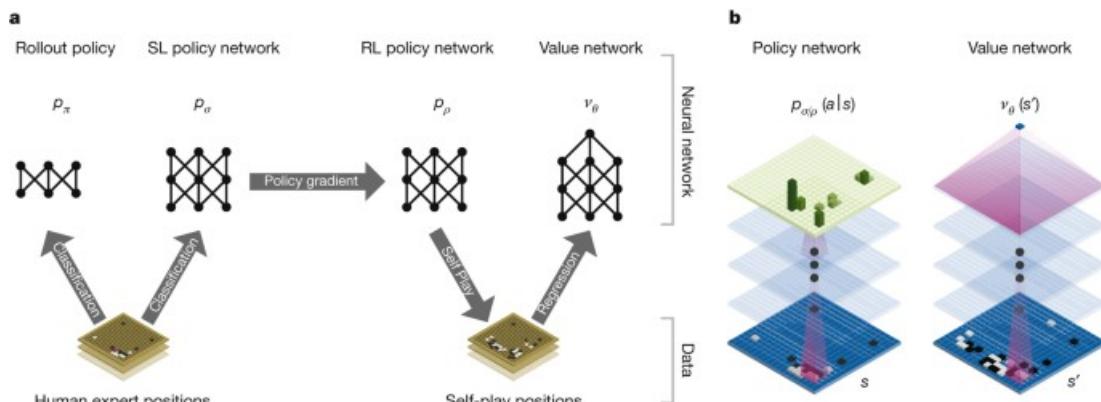


Figura 2: Representação dos da arquitetura dos métodos de aprendizado implementados para o AlphaGo. Percebe-se que foram utilizados métodos de *supervised learning* e *unsupervised learning* a fim de definir a melhor possível jogada dentre as milhões possíveis a partir de um dado estado de jogo.

Parte II

Supervised Learning

2 Representação de modelos

A fim de representação futura, podemos utilizar a notação $x^{(i)}$ para denotar as variáveis de entrada (*input*) e $y^{(i)}$ para denotar as variáveis de saída (*output*). O par $(x^{(i)}, y^{(i)})$ é chamado de exemplo de treino e o conjunto de dados (*dataset*) que está sendo utilizado para análise é uma lista com m exemplos de treino $(x^{(i)}, y^{(i)}); i = 1, \dots, m$ chamado conjunto de treino (*training set*).

Podemos usar essa notação para descrever o método de aprendizado supervisionado de uma forma mais formal, na qual, dado um *training set*, aprender uma função $h : X \rightarrow Y$ de forma que $h(x)$ seja um "bom" preditor para o valor correspondente de y . Historicamente, a função h é conhecida como hipótese (*hypothesis*). Podemos analisar a seguinte definição através da Figura 3.

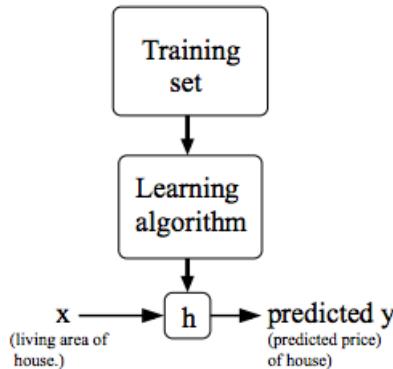


Figura 3: Processo de treinamento utilizando Supervised Learning. Dado um conjunto de treinamento como entrada, o nosso algoritmo de aprendizado tenta prever um valor y de saída que tenha relação com o valor x de entrada. Essa previsão é calculada através da função hipótese h .

3 Regressão linear

Regressão é um método que modela um valor de destino com base em preditores independentes. Este método é usado, principalmente, para prever e descobrir a relação de causa e efeito entre as variáveis.

Uma regressão linear simples é um tipo de regressão que analisa, a partir de um conjunto de variáveis independentes de entrada x , a relação entre essas variáveis com os seus respectivos valores esperados y . Na Figura 4, abaixo, a linha vermelha representa a melhor reta que aproxima melhor cada um dos pontos representados em azul. Ou seja, baseado em um conjunto de dados, tentamos gerar uma reta que modela cada um desses dados de forma ótima.

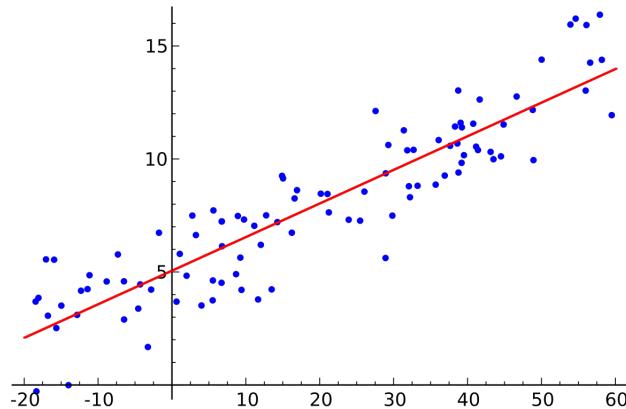


Figura 4: Representação de uma reta gerada a partir do método de regressão linear. A reta em vermelho representa a melhor aproximação de cada um dos pontos representados em azul, que são os dados de entrada.

A reta gerada pela regressão linear pode ser modelada a partir da equação linear abaixo:

$$y = \theta_0 + \theta_1 x$$

Portanto, o objetivo do algoritmo de regressão linear é encontrar os melhores valores de θ_0 e θ_1 .

Os métodos utilizados para calcularmos esses parâmetros serão apresentados nas seções seguintes.

4 Função Custo (*Cost Function*)

A função custo - também chamada de *loss function* - pode ser utilizada para medir a precisão da nossa função hipótese $h : X \rightarrow Y$. A função custo utiliza da diferença média de todos os resultados da função hipótese com todos *inputs* de x e *outputs* de y , como representado na Figura 5.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x_i) - y_i)^2$$

Através dessa expressão podemos perceber que o objetivo principal da função custo é minimizar a diferença entre o resultado esperado a função hipótese e o valor de y através das entradas θ_0 e θ_1 . Essa função também é conhecida como *Squared error function* ou *Mean squared error*.

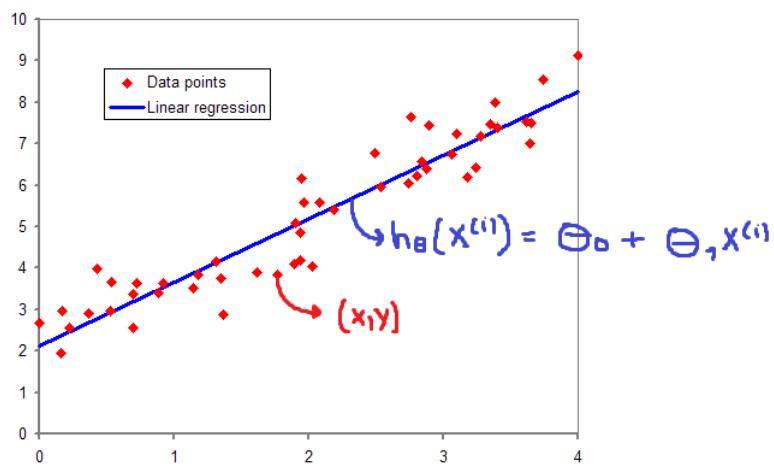


Figura 5: Representação da função custo

Dessa forma, temos quatro pré-definições básicas:

- Hipótese: $h_\theta(x) = \theta_0 + \theta_1 x$;
- Parâmetros: θ_0, θ_1 ;
- Função Custo: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x_i) - y_i)^2$;
- Objetivo: $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

Podemos utilizar linhas de contorno para representar a função de duas variáveis $J(\theta_0, \theta_1)$ em apenas duas dimensões como representado nas Figuras 6 e 7.

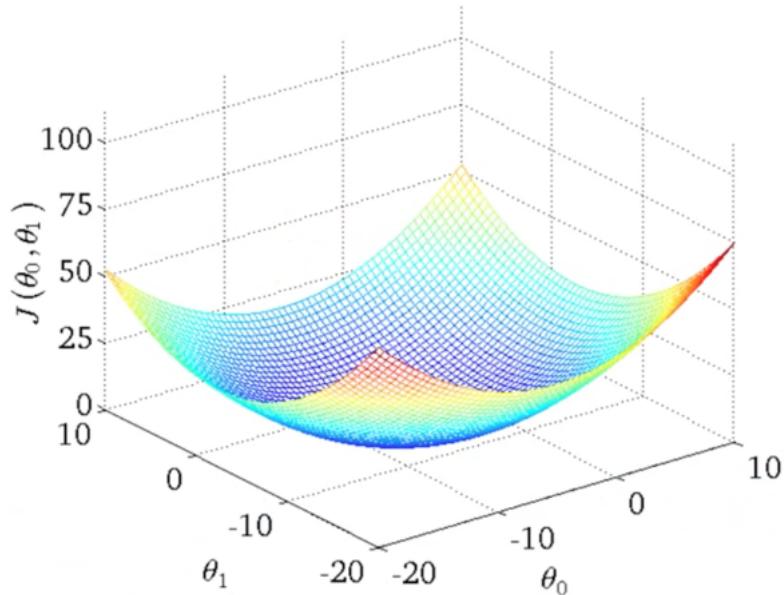


Figura 6: Representação da função custo através de visualização 3D

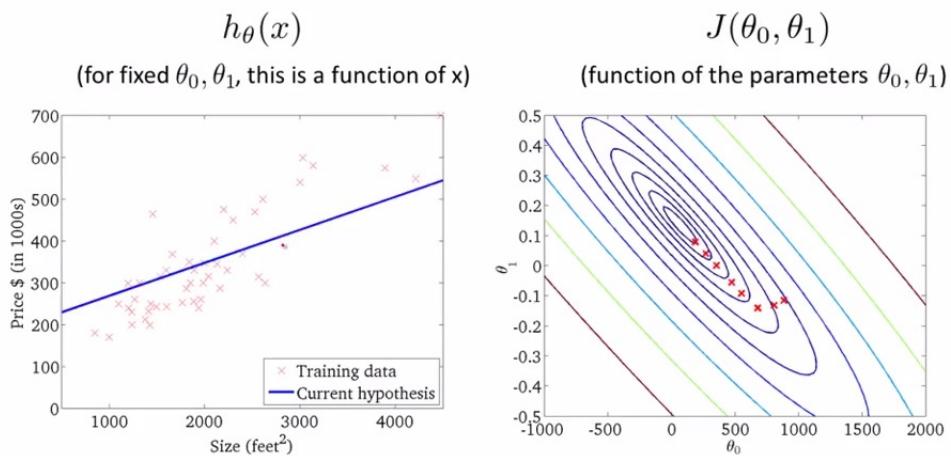


Figura 7: Representação da função custo através de linha de contorno

Os gráficos da Figura 7 minimizam a função custo ao máximo. O resultado de θ_0 e θ_1 tende a ficar em torno de 250 e 0.12, respectivamente. Em outras palavras, a melhor aproximação da função

custo está mais no centro das linhas de contorno. Chamamos o método de minimização da função custo de método do gradiente descendente que será discutido na Seção 5.

5 Gradiente Descendente (*Gradient Descent*)

O método do gradiente descendente pode ser utilizado para minimizar o valor de uma função. Utilizaremos esse método a fim de minimizar a função $J(\theta_0, \theta_1)$.

Podemos representar esse método a partir de um gráfico em três dimensões, onde θ_0 está no eixo x , θ_1 está no eixo y e o valor da função $J(\theta_0, \theta_1)$ está no eixo z , conforme está representado na Figura 8. Os pontos no gráfico são os resultados da função custo utilizando a função hipótese $h(x)$ com as entradas θ_0 e θ_1 .

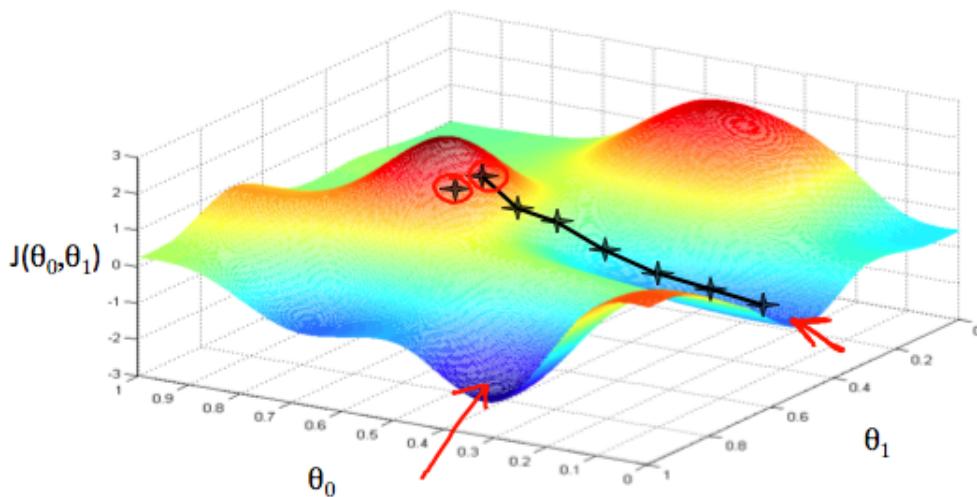


Figura 8: Representação da função gradiente descendente

O método de minimização utilizando a função gradiente descendente pode ser pensado como um algoritmo, que para cada ponto, a partir do inicial, dado como entrada, escolhe a descida mais íngrime do valor de $J(\theta_0, \theta_1)$ na função através de derivadas parciais e um valor α (*learning rate*) que determinará a distância entre cada descida. Obtemos sucesso, quando a função custo estiver em um dos mínimos do gráfico, como representado pelas setas vermelhas na Figura 8.

5.1 Algoritmo Gradiente Descendente

Algorithm 1 Algoritmo Gradiente Descendente

```

1: procedure
2:   repeat
3:      $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$                                  $\triangleright (\text{para } j = 0 \dots n)$ 
4:   until convergir
5: end procedure

```

Enquanto o método de gradiente descendente não convergir para o mínimo da função, a cada iteração, atualizamos simultaneamente os valores de $\theta_1, \theta_2, \dots, \theta_n$ fazendo com que esses valores se aproximem

cada vez mais ao mínimo da função.

Podemos perceber que o valor de α tem um certo impacto na atualização dos valores de θ . Valores de α pequenos, a convergência do método gradiente descendente é mais lenta. E para valores de α muito grandes, a convergência do método gradiente pode ultrapassar o mínimo, o que pode impossibilitar a convergência da função.

5.2 Gradiente Descendente para Regressão Linear

Podemos utilizar o método de gradiente descendente para minimizar a função *Mean squared error* utilizada no algoritmo de regressão linear, substituído a função $J(\theta_0, \theta_1)$ por nossa função hipótese. Dessa forma, o algoritmo de gradiente descendente para a minimização da função J terá a seguinte estrutura:

Algorithm 2 Algoritmo Gradiente Descendente Para Minimização Da Função J

```
1: procedure
2:   repeat
3:      $\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})$ 
4:      $\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$ 
5:   until convergir
6: end procedure
```

Onde m é o tamanho do conjunto de treino; θ_0 uma constante que será atualizada simultaneamente com θ_1 ; e $x^{(i)}, y^{(i)}$ são valores dados no conjunto de treino.

Esse algoritmo é aplicado para todos os valores dados no conjunto de treino, chamamos isso de *batch gradient descent*. Dessa forma, quando aplicamos o algoritmo, a função J possui apenas um mínimo global (sem outros mínimos locais). Portanto, a função de gradiente descendente sempre converge para regressões lineares, pois J é uma função quadrática convexa.

Nas próximas seções, veremos alguns métodos de otimização desse algoritmo utilizando álgebra linear.

6 Álgebra linear

Nesta seção, serão revisados, brevemente, alguns conceitos básicos da álgebra linear, como por exemplo operações com matrizes, inversa e transposta de matrizes e suas propriedades.

6.1 Matrizes e vetores

Matrizes são *arrays* bidimensionais.

$$M_3 = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

A matriz acima é considerada uma matrix 3x3.

Vetores são matrizes com apenas uma coluna e diversas linhas (matriz coluna).

$$v = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Como podemos perceber, vetores são um subconjunto de matrizes, e o vetor acima é uma matriz 3×1 .

Dessa forma, podemos definir algumas notações referentes à matrizes e vetores.

- A_{ij} se refere ao elemento que se encontra na i -ésima linha e na j -ésima coluna;
- Um vetor com ' n ' linhas é um vetor ' n '-dimensional;
- v_i se refere ao i -ésimo elemento do vetor;

6.2 Operações básicas com matrizes e vetores

A adição e subtração de vetores são operações unitárias referentes a cada linha do vetor.

$$v + u = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} v_1 + u_1 \\ v_2 + u_2 \\ v_3 + u_3 \end{bmatrix}$$

Para multiplicarmos ou dividirmos um valor escalar por um vetor é usada a mesma lógica.

$$v \cdot x = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \cdot x = \begin{bmatrix} v_1 \cdot x \\ v_2 \cdot x \\ v_3 \cdot x \end{bmatrix}$$

6.3 Multiplicação entre matrizes e vetores

Para realizar a multiplicação entre uma matriz e um vetor, realizamos a multiplicação no sentido "linha x coluna" conforme a expressão abaixo:

$$M_2 \times v = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} a \times v_1 + b \times v_2 \\ c \times v_1 + d \times v_2 \end{bmatrix}$$

Ao multiplicarmos uma matriz A_{ij} por um vetor v com j linhas, teremos como resultado uma matriz B_{i1} .

6.4 Multiplicação entre duas matrizes

Para realizar a multiplicação entre duas matrizes seguimos a mesma lógica apresentada na Seção 6.3 multiplicando no sentido "linha x coluna".

$$A_2 \times B_2 = \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \times \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} = \begin{bmatrix} a_1 \times a_2 + b_1 \times c_2 & a_1 \times b_2 + b_1 \times d_2 \\ c_1 \times a_2 + d_1 \times c_2 & c_1 \times b_2 + d_1 \times d_2 \end{bmatrix}$$

Ao multiplicarmos duas matrizes $A_{m \times n}$ e $B_{n \times o}$ teremos uma matriz $C_{m \times o}$. Dessa forma, podemos definir algumas propriedades relacionadas às operações com matrizes:

- Multiplicação de matrizes não são comutativas, ou seja, $A \times B \neq B \times A$
- Multiplicação de matrizes são associativas, ou seja, $(A \times B) \times C = A \times (B \times C)$

6.5 Matriz identidade, inversa e transposta

A matriz identidade é aquela que, ao ser multiplicada por uma outra matriz de mesma dimensão, resulta na matriz original. Em outras palavras, é uma matriz onde há apenas '1's' na sua diagonal principal.

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A inversa de uma matriz, denotada por A^{-1} é aquela que, ao ser multiplicada por A , resulta na matriz identidade I de A . Em outras palavras:

$$A \times A^{-1} = I$$

A matriz transposta da matriz A_{ij} é a matriz A_{ji}^T . Trata-se da matriz que vamos obter quando reescrevemos a matriz A_{ij} trocando de posição as linhas e colunas, transformando a primeira linha de A_{ij} na primeira coluna de A_{ji}^T , a segunda linha de A_{ij} na segunda coluna de A_{ji}^T , e assim sucessivamente.

$$A_{ij} = A_{ji}^T$$

$$A = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}, A^T = \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$$

7 Regressão Linear com múltiplas variáveis

Nesta seção iremos introduzir os conceitos básicos para a aplicação do algoritmo de regressão linear para multiplas entradas. Em outras palavras, iremos tentar estimar o valor da saída y a partir de diversos parâmetros de entrada.

7.1 Múltiplos parâmetros de entrada

Precisamos definir algumas notações que serão utilizadas.

- $x_j^{(i)}$ = valor do parâmetro j no i -ésimo exemplo de treino;
- $x^{(i)}$ = valor dos parâmetros no i -ésimo exemplo de treino (vetor);
- m = número de exemplos de treino;
- n = número de parâmetros;

Com isso, podemos definir a função hipótese para múltiplos parâmetros da seguinte forma:

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_n x_n$$

Sendo θ_i os parâmetros de entrada da função

Podemos utilizar as definições de matrizes anteriormente vistas na Seção 6 para definir a função hipótese de múltiplas variáveis da seguinte forma:

$$h_{\theta}(x) = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 & \dots & \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x$$

É importante mencionar que o valor de $x_0^{(i)} = 1$ para ($i \in 1, \dots, m$) por convenção.

7.2 Gradiente Descendente com múltiplas variáveis

Da mesma forma apresentada na Seção 4, teremos as seguintes pré-definições das funções que serão utilizadas a fim de minimizar o custo da função J com múltiplas variáveis de entrada.

- Hipótese: $h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_n x_n$;
- Parâmetros: $\theta_0, \theta_1, \dots, \theta_n$ (vetor $(n+1)$ -dimensional);
- Função Custo: $J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x_i) - y_i)^2$;

O algoritmo gradiente descendente, em geral, tem a mesma estrutura para os diferentes problemas. Nós apenas temos que iterar sobre os n parâmetros de entrada, atualizando-os simultaneamente.

Algorithm 3 Algoritmo Gradiente Descendente Para Múltiplas Variáveis

```

1: procedure
2:   repeat
3:      $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$                                  $\triangleright$  para  $j := 0 \dots n$ 
4:   until convergir
5: end procedure

```

Podemos otimizar o algoritmo gradiente descendente colocando todos os parâmetros no mesmo intervalo. Esse tipo de operação é muito importante para evitar que o gradiente trabalhe com valores muito grandes que possam causar "explosão", ou muito pequenos que possam ser zero, o que pode se tornar um problema para o treinamento do algoritmo de aprendizado.

Para isso, utilizaremos da técnica de normalização média (*mean normalization* ou *feature scaling*). *Feature scaling* envolve dividir os dados de entrada por um intervalo (desvio padrão, ou uma subtração entre o maior e o menor valor das variáveis de entrada) resultando em um valor próximo de 1. *Mean normalization* envolve subtrair o valor médio das variáveis de entrada de cada variável de entrada. Dessa forma, teremos a seguinte fórmula:

$$x_i = \frac{x_i - \mu_i}{s_i}$$

onde μ_i é a média de todos os parâmetros de entrada e s_i é a variação dos valores de entrada ($\max - \min$), ou o desvio padrão.

Com isso, teremos os valores de entrada em um mesmo intervalo, sem valores discrepantes para o cálculo da função custo e da otimização pelo gradiente descendente. Com os valores normalizados, evitamos diversos problemas que foram acima mencionados e o custo computacional para o cálculo das derivadas parciais é diminuído.

Para ter certeza que o método do gradiente descendente está funcionando corretamente, o valor de $J(\theta)$ deve diminuir a cada iteração e convergir para um valor próximo de zero. Caso o valor de $J(\theta)$ não esteja convergindo, podemos tentar um valor menor de α .

8 Equação Normal (*Normal Equation*)

É um outro método de minimização da função J , assim como o método de gradiente descendente. Essa forma de implementação, muitas vezes pode otimizar o tempo de processamento da função de minimização através de derivadas da função J a respeito aos θ_j 's igualando-os a zero. Isso nos permite encontrar o valor ótimo para θ sem iterações. Fórmula da equação normal é dada abaixo:

$$\theta = (X^T X)^{-1} X^T y$$

onde X é uma matriz na qual a coluna zero tem todos os elementos iguais a 1.

Dessa forma, podemos comparar as duas formas de implementação que temos: *Gradient Descent* e *Normal Equation*.

Tabela 1: Comparação entre os métodos *Gradient Descent* e *Normal Equation*

Gradient Descent	Normal Equation
É necessário definir o valor de α	Não é necessário definir o valor de α
Muitas iterações são necessárias	Não são necessárias iterações
$O(kn^2)$	$O(n^3)$, pois precisa calcular a inversa de $X^T X$
Funciona bem quando o valor de n é grande	Lento quando o valor de n é grande

9 Classificação

9.1 Problemas de classificação

O método de classificação tem como objetivo classificar um conjunto de dados entre estados ou tipos distintos. Um bom exemplo de um problema de classificação é quando queremos prever se um tumor é maligno ou benigno baseando-se apenas no tamanho do tumor.

Podemos usar o algoritmo de regressão linear para classificar uma base de dados em dois diferentes grupos, como por exemplo, para valores de $h(x) > 0.5$ e para valores $h(x) \leq 0.5$. Contudo, essa forma de implementação não funciona muito bem, pois os problemas de classificação, geralmente, não cabem em problemas de regressão linear.

De outra forma, gostaríamos de classificar nossa base de dados em saídas discretas. Como base, iremos nos focar nos problemas de classificação binários (*binary classification problems*), os quais os valores de y podem assumir apenas dois valores: zero ou um, em outras palavras $y \in \{0, 1\}$.

Podemos representar um problema de classificação através da imagem a seguir na Figura 9

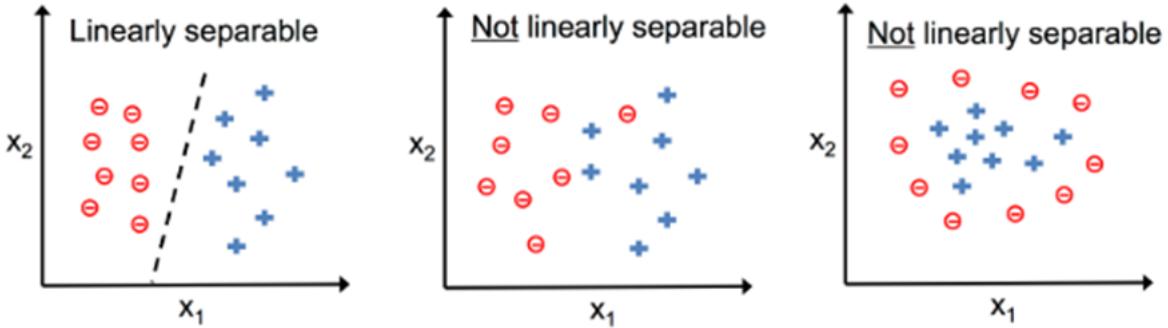


Figura 9: Representação de um problema de classificação

Podemos perceber que na Figura 9 temos dois tipos de classificações: linearmente separáveis e não linearmente separáveis. Nesta seção iremos discutir os problemas binários linearmente separáveis sem necessidade de regularização da função.

9.2 Representação da hipótese

Podemos utilizar o nosso antigo algoritmo de regressão linear para prever um valor de y discreto dado um valor de x . Mas como foi mencionado anteriormente, essa não é uma boa solução para problemas de classificação e para isso, devemos modificar a função hipótese a fim de satisfazer a saída discreta dos problemas de classificação, ou seja, $0 \leq h_\theta(x) \leq 1$.

Uma boa modificação seria basear a nossa função hipótese na função logística (*Logistic Function*) de forma que possamos nos basear na função sigmoide conforme representada na Figura 10. Em outras palavras, teremos:

$$h_\theta(x) = g(\theta^T x), \quad z = \theta^T x, \quad g(z) = \frac{1}{1+e^{-z}}$$

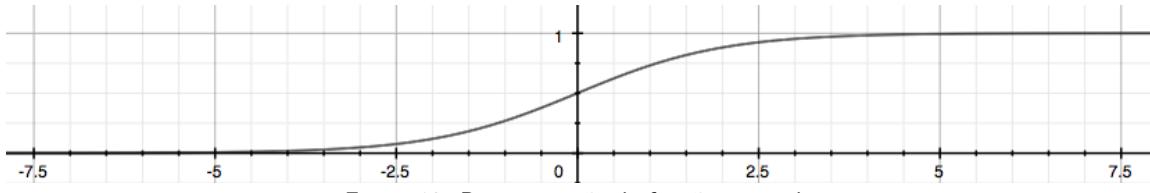


Figura 10: Representação da função sigmoide

A função sigmoide mapeia um valor real em um valor no intervalo $(0, 1)$ fazendo com que seja a melhor forma de implementação de problemas de classificação.

Dessa forma, podemos chegar a algumas conclusões e interpretações dessa nova forma de implementação da função hipótese:

- $h_\theta(x)$ nos dá a probabilidade da nossa saída ser 1.

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

$$P(y = 0|x; \theta) + P(y = 1|x; \theta) = 1$$

9.3 Limite de decisão (*Decision Boundary*)

É uma forma de encontrarmos o limite entre os valores discretos que temos na saída da função hipótese. Em outras palavras, podemos traduzir este pensamento da seguinte forma:

$$h_\theta(x) \geq 0.5 \rightarrow y = 1$$

$$h_\theta(x) < 0.5 \rightarrow y = 0$$

pois nossa função sigmoide se comporta de tal forma que quando a entrada $z(x) = \theta^T x \geq 0$ o valor da função $g(z) \geq 0.5$.

Dessa forma, o limite de decisão é a linha que separa a área entre os valores classificados, ou como $y = 0$, ou como $y = 1$. Esses valores discretos representam as duas classes do problema de classificação binário. Em outras palavras, o limite de decisão é uma curva que separa essas duas classes.

Podemos utilizar outras estruturas de funções para gerarmos diferentes limites de decisão de acordo com a nossa base de dados. Por exemplo se nossos parâmetros dividem áreas circulares, podemos utilizar uma função $z = \theta_0 + \theta_1 x_1^2 + \theta_2 x_2^2$.

9.4 Classificação Multiclasse

A classificação multiclasse é uma definição para quando temos além de apenas uma saída binária na função hipótese. Ou seja, podemos classificar os nossos dados em mais de duas classificações. A Figura 11 representa um problema de classificação multiclasse utilizando o método *one-vs-all*.

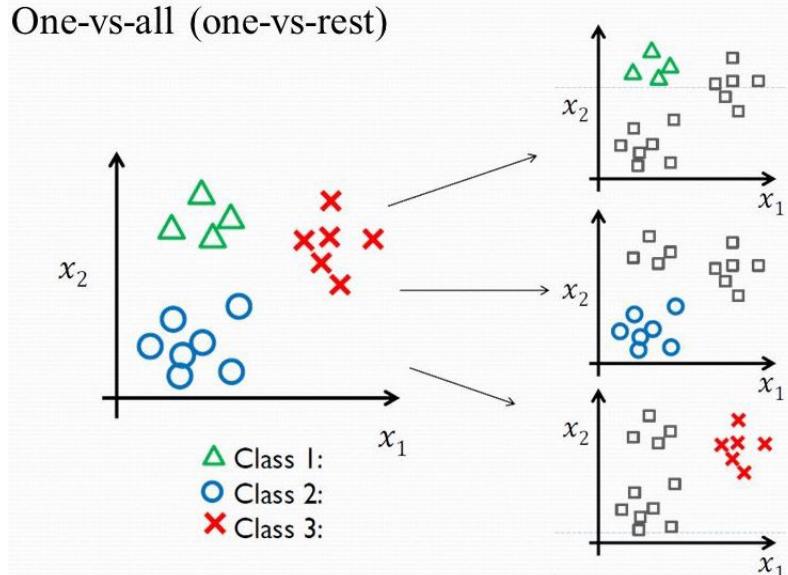


Figura 11: Representação de um problema de classificação multiclasse

No método de classificação *one-vs-all* treinamos N classificadores lineares distintos que são projetados para reconhecer cada classe. Para cada classe distinta aplicamos uma regressão logística binária e

utilizamos o valor retornado da função hipótese para classificar os dados. Em outras palavras, geramos um limite de decisão aplicando a regressão logística para cada classe de acordo com as outras. Então, se temos N classes de entrada, roda-se o algoritmo de regressão logística N vezes para cada uma dessas classes, retornando N valores distintos de h para cada uma das classes.

10 Regressão Logística (*Logistic Regression*)

10.1 Função Custo (*Cost Function*)

Nas seções anteriores, discutimos a implementação da função custo para a regressão linear. Entretanto, para a regressão logística, utilizaremos uma função hipótese voltada para os problemas de classificação. Na Figura 10 está representada a função logística que será utilizada para calcular a hipótese do problema. Essa função gera valores probabilísticos no intervalo $[0, 1]$ e, com isso, podemos calcular o valor da função custo de forma que possamos comparar a probabilidade da nossa hipótese ser igual a 1 com o resultado y esperado.

Para realizar essa comparação, devemos realizar algumas manipulações algébricas de forma que possamos definir uma função $J(\theta)$ que calcule o custo. Abaixo, está descrita a intuição dessa manipulação.

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

$$Cost(h_\theta(x^{(i)}), y) = -\log(h_\theta(x)) \quad \text{se } y = 1$$

$$Cost(h_\theta(x^{(i)}), y) = -\log(1 - h_\theta(x)) \quad \text{se } y = 0$$

Dessa forma, podemos ter duas diferentes funções para a representação da função custo para a regressão logística. Como podemos ver na Figura 12 temos duas representações, para $y = 1$ em azul e para $y = 0$ em vermelho.

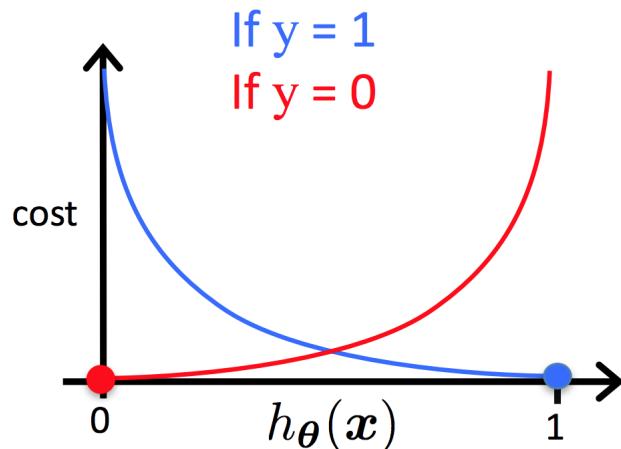


Figura 12: Representação da função custo da regressão logística

$$Cost(h_\theta(x^{(i)}), y) = 0 \quad \text{se } h_\theta(x) = y$$

$$Cost(h_\theta(x^{(i)}), y) \rightarrow \infty \quad \text{se } y = 0 \quad \text{e} \quad h_\theta(x) \rightarrow 1$$

$$Cost(h_\theta(x^{(i)}), y) \rightarrow \infty \quad se \quad y = 1 \quad e \quad h_\theta(x) \rightarrow 0$$

Podemos perceber que quando o valor da função custo é zero, então o valor da função hipótese é igual a y . Além disso, quando a função custo tende ao infinito e o valor de y é igual a zero, o valor da função hipótese tende a um e se o valor de y é igual a um, a função hipótese tende a zero.

A fim de simplificar a função custo, podemos reescrevê-la da seguinte forma sem alterar o valor do resultado:

$$Cost(h_\theta(x), y) = -y \cdot \log(h_\theta(x)) - (1 - y) \cdot \log(1 - h_\theta(x))$$

Com isso, podemos generalizar a função custo de acordo com a expressão abaixo:

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

10.2 Gradiente Descendente para Regressão Logística

Da mesma forma que o método de regressão linear, o algoritmo do gradiente descendente funciona iterando sobre os valores de θ e derivando a função custo J em relação a θ .

Assim, podemos descrever o método através do seguinte algoritmo:

Algorithm 4 Algoritmo Gradiente Descendente Para Regressão Logística

```

1: procedure
2:   repeat
3:      $\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$             $\triangleright$  Atualiza simultaneamente todos  $\theta_j$ 
4:   until convergir
5: end procedure

```

ou através da forma vetorializada:

$$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - \vec{y})$$

Dessa forma, podemos utilizar dos métodos vistos anteriormente para implementar o algoritmo de regressão logística. Contudo, ainda podemos utilizar de técnicas de computação numérica para otimizar os algoritmos. Como por exemplo *Conjugate gradient*, *BFGS* e *L-BFGS* que podemos utilizar a fim de otimizar o método de gradiente descendente.

11 Underfitting e overfitting

Quando tentamos prever um valor y a partir de um conjunto de treino, podem-se ocorrer alguns problemas relacionados a função de hipótese. Esses problemas são chamados de *underfitting* e *overfitting*. Para explicá-los, podemos nos basear na Figura 13

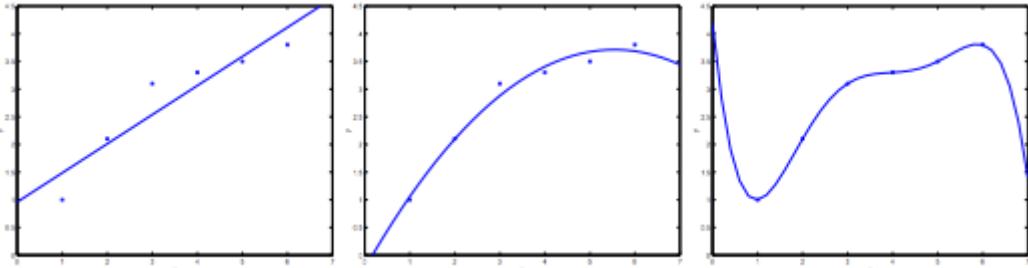


Figura 13: Representação dos problemas de *underfitting* e *overfitting*

Como podemos perceber, na figura mais à esquerda, temos uma função linear do tipo $y = \theta_0 + \theta_1x$ que não seja ajusta adequadamente com a nossa base de treino. Na figura do centro temos uma função quadrática do tipo $y = \theta_0 + \theta_1x + \theta_2x^2$, que aparentemente se adapta muito bem a nossa base de dados. E na figura mais à direita, temos um polinômio de grau cinco que atinge todos os pontos da nossa base de dados.

Com isso, podemos dizer que a figura mais à esquerda apresenta o problema de subajuste (*underfitting* ou *high bias*) e a figura mais à direita apresenta o problema de sobreajuste (*overfitting* ou *high variance*).

O problema de *underfitting* ocorre quando a função hipótese h não consegue mapear com consistência os valores da saída pois uma função muito simples é utilizada ou foram utilizados poucos parâmetros de entrada.

De outra forma, o problema de *overfitting* ocorre quando a função hipótese se adapta perfeitamente a nossa base de treino, mas não consegue generalizar os resultados das entradas. Isso ocorre pois uma função muito complexa foi utilizada ou o número de parâmetros utilizados como entrada da função é muito alto.

Assim, temos alguns métodos que podemos utilizar para evitar esse tipo de problema. Dois são principais e estão listados abaixo:

1. Reduzir o número de parâmetros:

- Manualmente selecionar os parâmetros a serem removidos;
- Usar algoritmo de modelo de seleção [16].

2. Regularização (*Regularization*):

- Manter todos os parâmetros mas reduzir a magnitude dos parâmetros de θ_j ;

11.1 Função custo em casos de *overfitting*

Em casos de *overfitting* na função hipótese, podemos reduzir o peso dos termos de maior grau, aumentando o seu custo.

Para isso, podemos utilizar do método de regularização para aumentar o custo de determinadas variáveis. Por exemplo, se tivermos uma função hipótese com quatro parâmetros, teremos uma

função de grau quatro do tipo $\theta_0 + \theta_1x + \theta_2x^2 + \theta_3x^3 + \theta_4x^4$. Através da regularização, podemos reduzir o problema diminuindo a influência dos termos de grau três e quatro.

Podemos modificar a função custo a fim de reduzir os valores de θ_3 e θ_4 e aumentar os valores de θ_1 e θ_2 da seguinte forma:

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

Chamamos a expressão acima de função custo regularizada. O valor de λ representa o parâmetro de regularização e determina o quanto os custos dos parâmetros de θ serão inflados. Caso selecionarmos um valor muito alto para λ , a função custo resultará em *underfitting*. Para isso devemos escolher estrategicamente o valor de λ .

Para exemplificar o método de regularização em casos de *overfitting*, podemos analisar a Figura 13 na seção anterior. Na imagem central, a curva representada em azul descreve a situação ótima do limite de decisão para os dados. A regularização transforma uma curva complexa em *overfitting* em uma curva ótima regularizada.

11.2 Regressão linear regularizada

Podemos utilizar o conceito de regularização para evitar problemas de *overfitting* no método de regressão linear.

É possível modificar o método gradiente descendente de forma que consigamos regularizar a atualização do valor de θ conforme visto na Seção 5:

Algorithm 5 Algoritmo Gradiente Descendente Para Regressão Linear Regularizado

```

1: procedure
2:   repeat
3:      $\theta_0 := \theta_0 - \frac{\alpha}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$ 
4:      $\theta_j := \theta_j - \left[ \left( \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right]$   $\triangleright j \in 1, 2, \dots, n$ 
5:   until convergir
6: end procedure

```

Podemos perceber que atualizamos o valor de θ_0 separadamente a fim de focarmos apenas nos termos de maior grau.

Além disso, podemos representar o mesmo algoritmo através da equação normal - descrita na Seção 8 - da seguinte forma:

$$\theta = (X^T X + \lambda \cdot L)^{-1} X^T y$$

$$onde \quad L = \begin{bmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

11.3 Regressão logística regularizada

Podemos regularizar a regressão logística da mesma forma que regulizarizamos a função J para a regressão linear. Com isso, podemos evitar casos de *overfitting* no método de regressão logística.

Assim, podemos escrever a função $J(\theta)$ da seguinte forma:

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

Com isso, da mesma forma que na regressão linear, podemos escrever o algoritmo gradiente descendente com a regularização da função J .

Algorithm 6 Algoritmo Gradiente Descendente Para Regressão Logística Regularizado

```

1: procedure
2:   repeat
3:      $\theta_0 := \theta_0 - \frac{\alpha}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$ 
4:      $\theta_j := \theta_j - \alpha \left[ \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} + \frac{\lambda}{m} \theta_j \right]$   $\triangleright j \in 1, 2, \dots, n$ 
5:   until convergir
6: end procedure

```

Parte III

Redes Neurais

12 Redes Neurais: Representação

12.1 Definição básica

Redes neurais (NNs - do inglês *Neural Networks*) são sistemas de computação com nós interconectados que funcionam como os neurônios do cérebro humano. Usando algoritmos, elas podem reconhecer padrões escondidos e correlações em dados brutos, agrupá-los e classificá-los, e – com o tempo – aprender e melhorar continuamente.

12.2 Hipótese não-linear

Nesta seção iremos discutir os principais fundamentos da hipótese não linear e as motivações para a criação de sistemas de redes neurais.

Como vimos nas seções anteriores, para um problema de classificação não linear podemos expandir a nossa função hipótese para mais termos e, para isso, devemos usar métodos de regularização para manter o treino consistente e sem problemas de *overfitting*.

Problemas de redes neurais são utilizados para modelar problemas da vida real. Um bom exemplo seria quando, a partir de um conjunto de fotos, queremos determinar se uma foto é de um carro ou não. Um algoritmo utilizando redes neurais analisa cada pixel da foto e compara com os valores adquiridos no treino. Esse tipo de problema também é um algoritmo de classificação, porém, agora, utilizando redes neurais devido à sua complexidade.

12.3 Os Neurônios e o Cérebro

Problemas utilizando redes neurais são relativamente antigos, tendo sua origem entre os anos 80 e 90. O objetivo era, basicamente, modelar o cérebro humano através de algoritmos. Por exemplo, uma função hipótese seria modelar o córtex auditivo a fim de reconhecimento de áudio, ou a área de associação visual responsável pela visão.

Biologicamente, o cérebro humano é composto por estruturas nervosas chamadas de neurônios. Neurônios são células responsáveis pela transmissão dos impulsos nervosos e constituem cerca de 10% do tecido nervoso. Eles são constituídos basicamente por três estruturas: corpo celular, dendritos e axônio, como está representado na Figura 14.

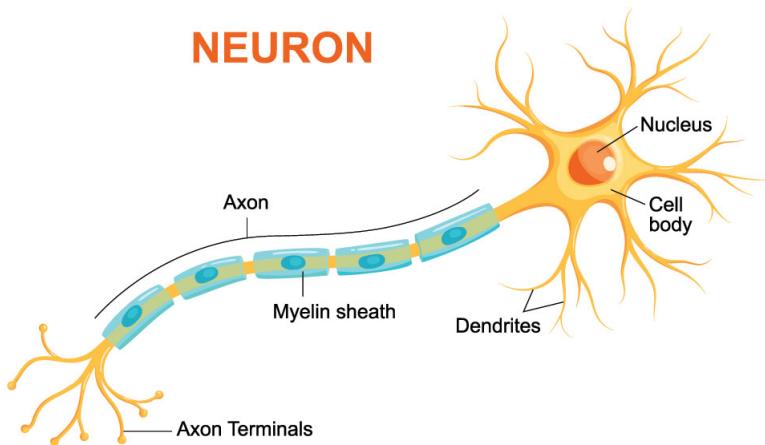


Figura 14: Representação estrutural de um neurônio humano.

Baseando-se nessa estrutura, as redes neurais tem como objetivo modelar computacionalmente as funções especificadas dos neurônios, como por exemplo, entendimento e geração de texto, análise e classificação de imagens, entre outros.

Na seção seguinte serão apresentadas as modelagens principais das redes neurais artificiais.

12.4 Representação do modelo

A partir de uma análise biológica do cérebro humano, sabemos que dois neurônios se comunicam através de impulsos nervosos chamados de sinapses. A informação recebida por um neurônio passa, primeiramente, pelos dentritos e vai em direção ao axônio. Podemos modelar essas informações a fim de gerar um modelo matemático que represente um neurônio da seguinte forma:

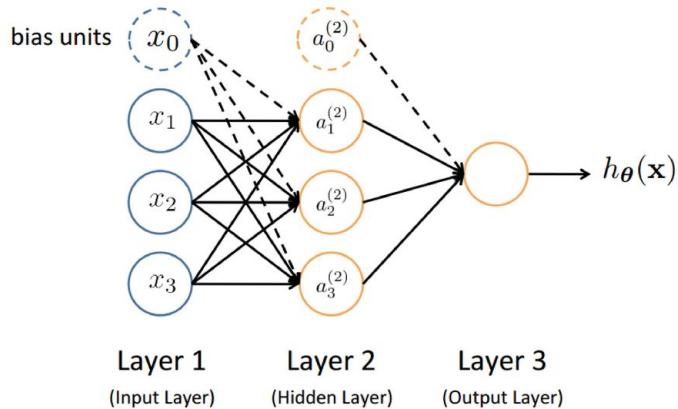
- Dendrito: entrada da função (x_1, \dots, x_n) ;
- Axônio: função hipótese $(h_\theta(x))$;

Nas redes neurais, utilizamos a mesma função utilizada na regressão logística $\frac{1}{1+e^{-z}}$, a qual chamamos de "ativação" da função e os parâmetros Θ são chamados de "pesos". Podemos perceber na Figura 15 que o neurônio possui três camadas, as quais chamamos de camada de entrada (*input layer*), camada escondida (*hidden layer*) e camada de saída (*output layer*). A camada de saída também pode ser chamada de função hipótese h_Θ .

Com isso, devemos adicionar algumas notações relacionadas às redes neurais.

- Chamamos de x_0, x_1, \dots, x_n os valores da *input layer*, onde $x_0 = 1$ (*bias unit*);
- $a_i^{(j)}$ é unidade de "ativação" i na camada j ;
- $\Theta^{(j)}$ é a matriz de pesos que controla o mapeamento da função da camada j para a camada $j + 1$.

Neural Network



Slide by Andrew Ng

12

Figura 15: Representação de uma rede neural

Na Figura 15 podemos perceber que para gerar a saída (função hipótese) passamos por uma *input layer* e uma *hidden layer*. Com isso, podemos descrever uma equação para que possamos determinar o valor da saída partindo dos parâmetros acima identificados.

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3) \\ h_{\theta}(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}) \end{aligned}$$

Percebe-se que a matriz Θ é uma matriz de tamanho 3×4 ($3 =$ número de parâmetros na camada dois e $4 =$ número de parâmetros de entrada). Quando aplicamos a função sigmoide para cada uma das camadas, nós obtemos o nodo de ativação da próxima camada. Assim, a função hipótese é a função logística aplicada na soma dos valores dos nodos de ativação os quais são multiplicados pelo parâmetro da matriz $\Theta^{(2)}$ que contém os valores dos pesos da segunda camada de nodos.

Com isso, temos a seguinte definição:

Se uma rede neural tem s_j unidades na camada j e s_{j+1} unidades na camada $j+1$, então $\Theta^{(j)}$ terá dimensões $s_{j+1} \times (s_j + 1)$.

Com as definições vistas acima, podemos introduzir uma implementação vetorizada das funções vistas. Para isso, iremos definir uma variável $z_k^{(j)}$ que engloba os parâmetros dentro da função g . Assim, teríamos:

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

Em outras palavras, para a camada $j = 2$, a função z seria:

$$z_k^{(2)} = \Theta_{k,0}^{(1)}x_0 + \Theta_{k,1}^{(1)}x_1 + \Theta_{k,2}^{(1)}x_2 + \dots + \Theta_{k,n}^{(1)}x_n$$

E a função vetorial teria a forma:

$$x = \begin{bmatrix} x_0 \\ x_2 \\ \dots \\ x_n \end{bmatrix}, \quad z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \dots \\ z_n^{(j)} \end{bmatrix}$$

Com $x = a^{(1)}$ teremos:

$$z^{(j-1)} = \Theta^{(j-1)}a^{(j-1)}$$

Com essas manipulações temos que a função hipótese pode ser definida da seguinte forma:

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

12.5 Aplicações

Mas afinal, para que uma rede neural pode ser útil? Ao longo das últimas décadas, a área de pesquisa que envolve redes neurais está se desenvolvendo de maneira significativa. Apesar desse desenvolvimento ainda estar no começo, já foram descobertas diversas aplicações de redes neurais, como podem ser vistas a seguir:

- *Computer Vision*: reconhecimento de objetos, emoções, etc;
- *Recurrent Neural Network (RNNs)*: reconhecimento de discursos e tradução de idiomas;
- *Wavenets* para tradução "texto-discurso";
- *Deep Q-Network (DQN)* e *Asynchronous Actor-Critic Agents (A3C)* para *reinforcement learning*, como por exemplo Atari;
- AlphaGo [19];
- *Neural Art* e síntese de imagens;
- *Aesthetic quality assessment*

Para o melhor entendimento do tipo de implementação que uma rede neural linear com apenas duas camadas - *input* e *output* - podemos criar um vetor para $\Theta^{(1)}$ para computar a função $x_1 AND x_2$ da seguinte forma:

$$\Theta^{(1)} = [-30 \quad 20 \quad 20]$$

Percebemos que no exemplo não temos *hidden layers*. Temos a *input layer* seguida pelo processamento da função hipótese.

Como sabemos que, pela função sigmoide, com uma entrada de duas variáveis x_1 e x_2 teremos valores binários (0 ou 1) na saída da função hipótese $h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$. Essa ideia pode ser representada na tabela a seguir:

Tabela 2: Implementação da rede neural usando portas 'and' $x_1 AND x_2$

x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Como vimos até agora, representamos redes neurais simples com duas camadas - *input* e *output* - chamadas de *Perceptrons*. Contudo, esse tipo de implementação possui diversas limitações. A primeira delas foi observada ao tentar implementar a função *XOR*. Essa limitação potencializou a criação de novas arquiteturas de redes neurais, pois a função *XOR* não é linearmente separável, ou seja, não se pode, de maneira linear - com apenas duas camadas - representá-la em uma rede neural. A Figura 16, abaixo, representa o surgimento desse problema.

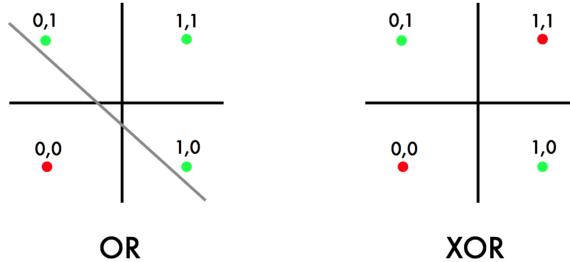


Figura 16: No gráfico da esquerda, percebemos a aplicação da função *OR*, cujos pontos verdes representam a saída 1 e os vermelhos a saída zero. No gráfico da esquerda, percebemos a aplicação da função *XOR*, cujos pontos possuem a mesma representatividade. Na função *OR*, podemos traçar uma reta que divida as classes 0 e 1 do problema, enquanto na função *XOR* isso não é possível devido ao fato das classes não sejam linearmente separáveis.

Uma solução para este problema foi a criação de novas camadas internas, chamadas de *hidden layers*, que foram discutidas na Seção 12.4, dando origem a uma nova arquitetura de rede neural chamada *Multilayer perceptron* (MLP). Na Figura 17 abaixo, está representada a arquitetura de rede neural para a resolução do problema *XOR*.

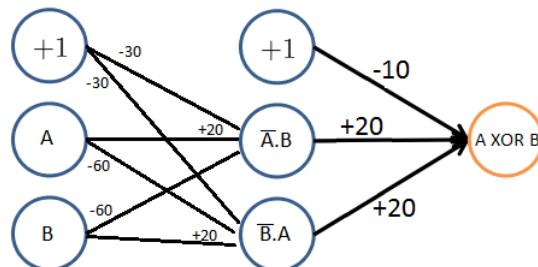


Figura 17: Representação da arquitetura que soluciona o problema *XOR*. Neste caso, temos três camadas: *input layer*, *hidden layer* e *output layer*

A seguir, iremos discutir as diferentes arquiteturas de redes neurais para diferentes tipos de classificação.

12.6 Classificação Multiclasse

Em problemas que são utilizados redes neurais, comumente são utilizados métodos de resolução desses problemas de forma multiclasse. Em outras palavras, a rede neural não terá apenas uma função hipótese como saída, $h_\Theta(x)$ nesses casos será um vetor que pertence a \mathbb{R}^n .]

Na Figura 18 abaixo podemos perceber que a rede neural foi construída com o objetivo de resolver um problema de classificação multiclasse, pois percebemos que a camada verde, abaixo ("layer 4"), que representa a função hipótese, é um vetor que pertence a \mathbb{R}^4

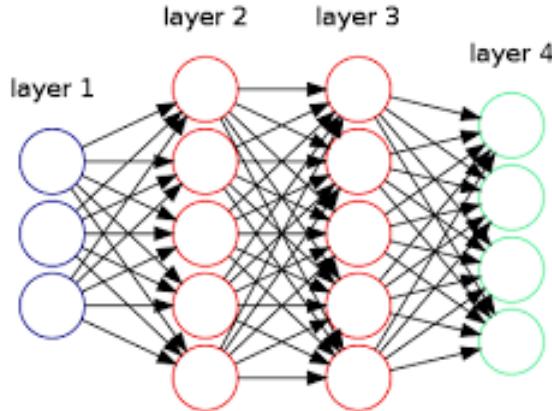


Figura 18: Representação de uma rede neural com classificação multiclasse

É possível escrever um programa de visão computacional para que possamos diferenciar pedestres, carros, motos e caminhões. Assim, teremos quatro nodos na *output layer* que representam cada um dos tipos de automóveis descritos.

$$h_\Theta(x)^1 \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad h_\Theta(x)^2 \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad h_\Theta(x)^3 \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad h_\Theta(x)^4 \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Onde $h_\Theta(x)^1$ representa um pedestre, $h_\Theta(x)^2$ representa um carro, $h_\Theta(x)^3$ representa uma moto e $h_\Theta(x)^4$ representa um caminhão. Dessa forma, a rede neural poderia ser modelada, em forma vetorial, da seguinte forma:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \\ \dots \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(3)} \\ a_1^{(3)} \\ a_2^{(3)} \\ a_3^{(3)} \\ \dots \end{bmatrix} \rightarrow \dots \rightarrow \begin{bmatrix} h_\Theta(x)_1 \\ h_\Theta(x)_2 \\ h_\Theta(x)_3 \\ h_\Theta(x)_4 \end{bmatrix}$$

13 Redes Neurais: Aprendizado

13.1 Estrutura básica

Uma rede neural consiste em uma sequência de camadas nas quais os dados são aplicados a transformações lineares e não lineares. Cada uma dessas camadas são compostas por neurônios (*neurons* ou *units*) e cada um desses neurônios estão conectados com os próximos, presentes na camada seguinte. Essas conexões são chamadas de pesos (*weights*) - um valor numérico. E, além disso, cada camada possui um valor fixo numérico, chamado de *bias*. Assim, os dados passam como entrada começam na *input layer*, passando por transformações lineares e não lineares nas camadas internas até atingir a *output layer*.

Com isso, o aprendizado de uma rede neural tem como objetivo otimizar a função custo (*loss function*) de acordo com os parâmetros, geralmente através da regra da cadeia ou através do método do gradiente descendente.

Nas próximas subseções iremos discutir os conceitos fundamentais para a estruturação básica de uma rede neural, apresentando os principais modelos e arquiteturas utilizados.

13.1.1 Camada linear

Como foi apresentado nas seções anteriores, uma rede neural possui diversas camadas e cada uma delas executa uma função sobre um dado. A primeira camada que iremos discutir é a camada mais básica, a qual executa transformações lineares sobre os dados. A camada linear - do inglês *linear layer* - realiza o treino das camadas intermediárias da rede neural baseado no método de regressão linear (Seção 3).

A camada linear realiza uma soma ponderada dos dados de entrada, ou seja, realiza uma soma dos resultados de funções afins da seguinte forma:

$$y = \sum_i w_i x_i + b$$

Onde w são as conexões de cada neurônio da camada com os pesos (*weight*), x são os valores dos neurônios conectados, b é o valor numérico *bias* de cada camada (constante), i é o número de conexões e y é o valor de saída do neurônio atual.

Os valores de y retornados irão percorrer toda a rede neural. Para isso, temos camadas de ativação as quais irão adicionar complexidade e dimensionalidade para a rede neural. Essas camadas tem como objetivo realizar transformações não-lineares dentro da rede. Com essas transformações, podemos classificar dados mais complexos e que não se comportam de maneira linear. Essas camadas de ativação podem ter diferentes formas e a mais comum delas é a chamada *dense layer*, cujos neurônios das camadas subsequentes estão, de alguma forma, conectados com os neurônios da camada precedente.

A seguir, iremos discutir diferentes implementações da camada de ativação usando diferentes modelos matemáticos

13.1.2 Camada de ativação: *Sigmoid layer*

A camada de ativação - do inglês *activation layer* ou *sigmoid layer* - realiza o cálculo da função sigmoide através de operações ponto a ponto não lineares, como foi visto na Seção 13.1.2. A função sigmoide, assim, tem a seguinte estrutura:

$$g(x) = \text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

Geralmente, aplicamos a função sigmoide a após realizarmos o cálculo da camada linear, ou seja:

$$y = g\left(\sum_i w_i x_i + b\right)$$

13.1.3 *Cross entropy loss*

Cross entropy loss tem como objetivo calcular o quanto discrepante do valor esperado está o valor produzido como saída na rede neural. Como foi visto na Seção 10.1. Essa função tem a seguinte estrutura:

$$\mathcal{L}_{CE}(y, g(z)) = -\frac{1}{m} \left[\sum_{j=1}^m y \log(g(z)) + (1-y) \log(1-g(z)) \right]$$

Essa função tem o mesmo objetivo da função custo anteriormente vista e que será detalhada na Seção 13.2.

13.1.4 Camada de ativação: *Softmax layer*

A função *softmax* é uma generalização da função logística para múltiplas dimensões. Essa função é muito utilizada em classificações multiclasse. A função *softmax* tem a seguinte estrutura:

$$y = f_{SM}(x) = \text{softmax}(x), \text{ onde } y_i = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}}$$

Elá recebe como entrada um vetor x e para cada uma das classes ela gera um valor entre -1 e 1, como se pode perceber na Figura 19.

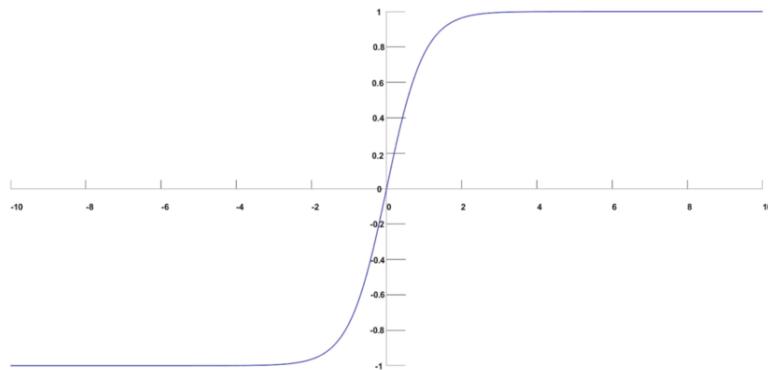


Figura 19: Representação da função *softmax*

Essa função representa a distribuição de probabilidade sobre os índices K de y .

Da mesma forma que a camada de ativação, podemos aplicar a saída dessa função na função *cross entropy loss*.

$$\mathcal{L}_{CE}(y, g(z)) = - \sum_{j=1}^k y \log(f_{SM}(x_j)) = - \sum_{j=1}^k y \left[x_j - \log \sum_{l=1}^k e^{x_l} \right]$$

13.1.5 Camada de ativação: *Rectified Linear layer (ReLU)*

A *Rectified Linear Layer* é uma função de ativação, assim como a função sigmoide que nos retorna *Rectified Linear Units* (ReLUs). É uma das funções de ativação mais utilizadas, atualmente, nas implementações de redes neurais devido ao fato de ser mais simples e mais barata que a função sigmoide.

Podemos implementar a ativação ReLU da seguinte forma:

$$y = f_{relu}(x) = \text{relu}(x), \text{ onde } y_i = \max(0, x_i)$$

Para melhor compreensão, a Figura 20 representa o comportamento da função $\text{relu}(x)$ de acordo com os parâmetros x de entrada.

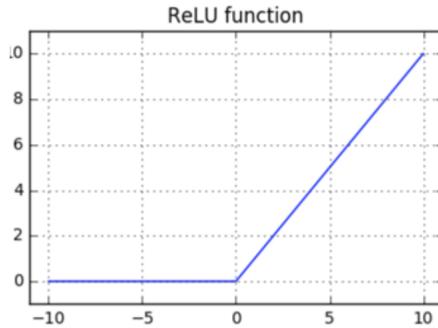


Figura 20: Representação da função ReLU usada para a camada de ativação

13.1.6 Camada de ativação: *Hyperbolic tangent*

A função *hyperbolic tangent*, ou *tanh* é muito similar a função de ativação *softmax*. Essa função é muito utilizada redes neurais recorrentes (Seção 23), especialmente nas arquiteturas GRU e LSTM. A função *tanh* tem a seguinte estrutura:

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Assim como a *softmax*, a função *tanh* também gera valores entre -1 e 1, como percebemos na Figura 21, porém não gera a distribuição para K valores de classe.

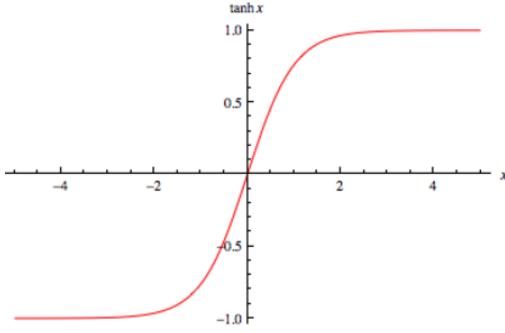


Figura 21: Representação da função \tanh usada para a camada de ativação

13.2 Função Custo (*Cost Function*)

Primeiramente, precisamos definir as variáveis que serão usadas para definir a função custo.

- L = número total de camadas (*layers*) na rede neural;
- s_l = número de unidades na camada l (sem contar a unidade *bias*);
- K = número de unidades/classes de saída (*output layer*).

Com isso podemos definir a função custo que será utilizada para calcular o custo de uma rede neural. Utilizaremos como base a função custo da regressão logística vista na Seção 10.1 a fim de determinarmos o valor da saída da função $J(\Theta)$.

A função custo para redes neurais tem o seguinte formato:

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \left[y_k^{(i)} \log((h_\Theta(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{j,i}^{(l)})^2$$

Podemos comparar esta equação com a equação da função custo da regressão logística. Assim, percebe-se que adicionamos alguns somatórios. Nos dois primeiros somatórios, de $k = 1$ até K e de $i = 1$ até m , somamos os valores do retorno da função custo da regressão logística para cada nodo da *output layer*. Nos três últimos somatórios, de $j = 1$ até s_{l+1} , de $i = 1$ até s_l e de $l = 1$ até $L - 1$, somamos os valores dos quadrados de todos os valores de Θ individuais em toda a rede neural.

13.3 *Backpropagation Algorithm*

O algoritmo de *Backpropagation* é, sem dúvida, o algoritmo mais importante para as redes neurais. É com esse algoritmo que as redes neurais aprendem, essencialmente.

Como foram vistos nas seções anteriores, existem algoritmos que são usados com o objetivo de minimizar a função custo. Para redes neurais, usa-se um algoritmo chamado *Backpropagation* que tem o mesmo objetivo: minimizar a função custo, ou seja $\min_{\Theta} J(\Theta)$. A minimização ocorre após realizarmos o processo de *forward propagation* usando o método de gradiente descendente - visto na Seção 5 -, atualizando os valores das *hidden layers* de acordo com os valores retornados da função custo.

Em geral, algoritmo de *backpropagation* acontece em duas fases principais que serão discutidas em detalhes a seguir. Essas fases são:

1. **Forward pass:** nossas entradas são passadas através da rede e as previsões de saída são obtidas. Nessa fase, calculamos a função custo e computamos as funções de ativação para cada transição de camada.
2. **Backward pass:** calculamos o gradiente da função custo na camada final (*output layer*) e usamos esse gradiente para aplicar recursivamente a regra da cadeia para atualizar os pesos da rede neural.

13.3.1 Forward pass

O principal objetivo dessa fase é calcular os valores de cada neurônio da nossa rede neural aplicando uma série de *dot products* (multiplicações entre vetores) e funções de ativação a fim de atingirmos a camada final da rede.

Por exemplo, se temos os seguintes valores de entrada x_i e a saída esperada y , de acordo com a tabela a seguir:

Tabela 3: Valores hipotéticos de entrada para a rede neural seguindo uma tupla do tipo (x, y)

x_0	x_1	x_2 (<i>bias</i>)	y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Cada um desses valores de x_i estarão presentes na *input layer* da rede neural. E para cada um desses valores, serão realizados *dot products* entre as entradas de cada camada seguido pelo cálculo da função de ativação. Esta ideia está exemplificada na Figura 22 a seguir.

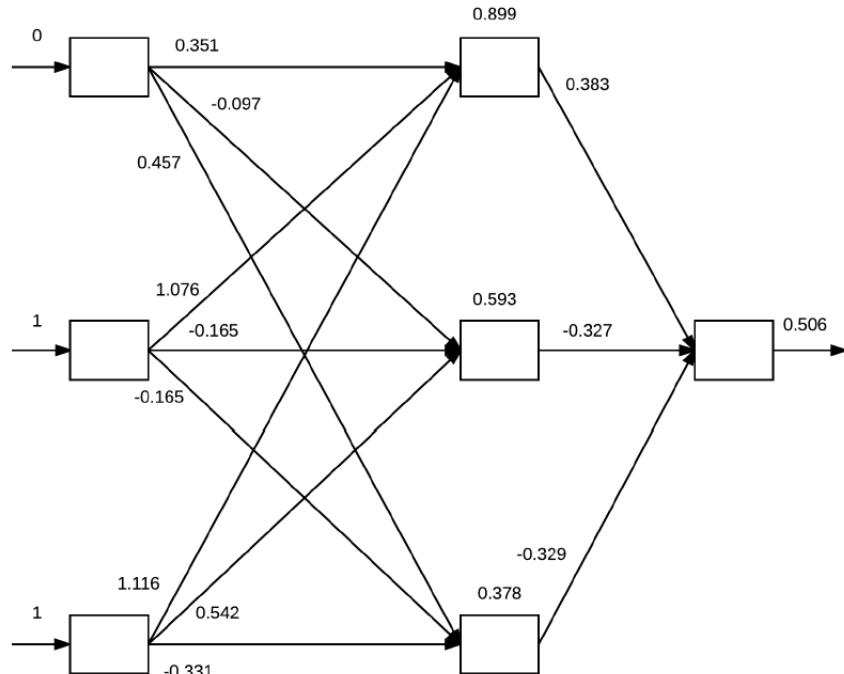


Figura 22: Exemplificação do algoritmo *backpropagation* a partir dos inputs definidos na Tabela 3. Na *input layer* estão os valores de x definidos. Os valores das arestas são inicializados aleatoriamente e representam os pesos. Na *hidden layer* estão os valores calculados a partir dos *dot products* e das aplicações das funções de ativação (neste caso, sigmoide) e na *output layer* o valor calculado pela função custo.

A partir dos valores de entrada baseados na Tabela 3 e os pesos que foram inicializados aleatoriamente, para cada um desses valores executamos as seguintes operações de *dot products* e ativações utilizando a função $g(z)$ sigmoide:

1. $g((0 \times 0.351) + (1 \times 1.076) + (1 \times 1.116)) = 0.899$
2. $g((0 \times 0.097) + (1 \times 0.165) + (1 \times 0.542)) = 0.593$
3. $g((0 \times 0.457) + (1 \times 0.165) + (1 \times 0.331)) = 0.378$

Os valores dos neurônios das *hidden layers* são atualizados de acordo com essas operações e, com eles podemos aplicar mais uma vez as mesmas operações para cada um dos valores atualizados para gerarmos o valor da *output layer*.

$$g((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506$$

A saída é, portanto, 0.506, o que representa um valor de probabilidade de 50,6%. Contudo, percebe-se que a rede neural não tem muita confiança a respeito do valor gerado, então, para que a rede neural realmente aprenda precisamos realizar a minimização da função custo através da etapa de *Backward pass*.

13.3.2 Backward pass

Para otimizar a função custo, desejamos selecionar pesos que fornecem uma estimativa ótima de uma função que modela nossos dados de treinamento. Ou seja, desejamos encontrar um conjunto de pesos Θ que minimize a saída $J(\Theta)$.

Para aplicar o algoritmo de *backpropagation* a nossa função de ativação deve ser diferenciável, de modo que possamos calcular a derivada parcial do erro em relação a um dado peso $\Theta^{(L)}$, o custo \mathcal{L} , saída do nó $a^{(L)}$ da *hidden layer* e saída da rede $z^{(L)}$.

$$\boxed{\frac{\partial \mathcal{L}}{\partial \Theta^{(L)}} = \frac{\partial z^{(L)}}{\partial \Theta^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial \mathcal{L}}{\partial a^{(L)}}}$$

A saída dessa equação é uma função composta dos pesos, entrada e funções de ativação.

Podemos, analisar cada um desses termos separadamente.

$$\frac{\partial \mathcal{L}}{\partial a^{(L)}} = 2(a^{(L)} - y), \quad \frac{\partial a^{(L)}}{\partial z^{(L)}} = g'(z^{(L)}), \quad \frac{\partial z^{(L)}}{\partial \Theta^{(L)}} = a^{(L-1)}$$

Então, seguindo o exemplo da Figura 22, a partir do valor de saída, iremos calcular a derivada da respectiva camada baseando-se em relação aos pesos que nela estão conectados. Essa derivada é uma composição de multiplicações de derivadas do custo calculado em relação a camada anterior, a camada anterior em relação a saída da rede neural e a saída da rede neural em relação aos pesos.

Com isso, podemos minimizar o erro dos pesos seguindo a seguinte ideia:

$$\text{Novo Peso} = \text{Peso Antigo} - \text{Derivada} \times \text{Taxa de aprendizado}$$

Para o caso de uma camada de uma rede neural, nós temos uma função com n entradas (número de neurônios da *input layer*) e m saídas (número de neurônios da *output layer*). Então, este é o

caso que teremos uma matriz de derivadas parciais que se refere ao Jacobiano. Um Jacobiano é uma matriz de derivadas parciais $m \times n$, em outras palavras

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \rightarrow \left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right)_{ij} = \frac{\partial f_i}{\partial x_j}$$

Podemos exemplificar o cálculo das derivadas parciais pensando em um exemplo simples com operações matemáticas. A Figura 23 a seguir representa uma rede de quatro camadas: camada de entrada, com os valores x , y e z , camada interna com operações soma (+) e \max , camada interna com a operação multiplicação (*) e a camada de saída. Percebe-se que executamos a operação $f = (x + y) * \max(y, z)$. Supondo que os valores das arestas são os valores retornados na saída de cada camada, temos que o valor da função $f = 6$.

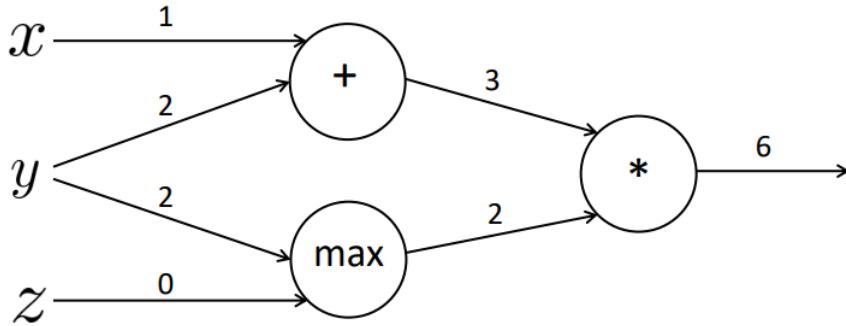


Figura 23

Com isso podemos gerar três valores internos distintos que representam a função:

$$a = x + y$$

$$b = \max(y, z)$$

$$f = ab$$

onde $x = 1$ e $y = 1$, $y = 2$ e $z = 0$.

Podemos realizar a *backpropagation* nessa rede calculando as derivadas parciais de cada um dos níveis a respeito ao nível anterior, como vimos anteriormente. Para isso, para facilitar a compreensão, calculamos as derivadas parciais locais, da seguinte forma:

$$\frac{\partial a}{\partial x} = 1, \frac{\partial a}{\partial y} = 1$$

$$\frac{\partial b}{\partial y} = 1(y > z) = 1, \frac{\partial b}{\partial z} = 1(z > y) = 0$$

$$\frac{\partial f}{\partial a} = b = 2, \frac{\partial f}{\partial b} = a = 3$$

$$\frac{\partial f}{\partial f} = 1$$

Com esses valores definidos, podemos calcular a aplicação da regra da cadeia para atualizar os valores internos dos nodos. Os valores representados em azul na Figura 24 são as multiplicações das derivadas resultantes da função em relação ao estado atual com a derivada da função gerada no estado imediatamente anterior.

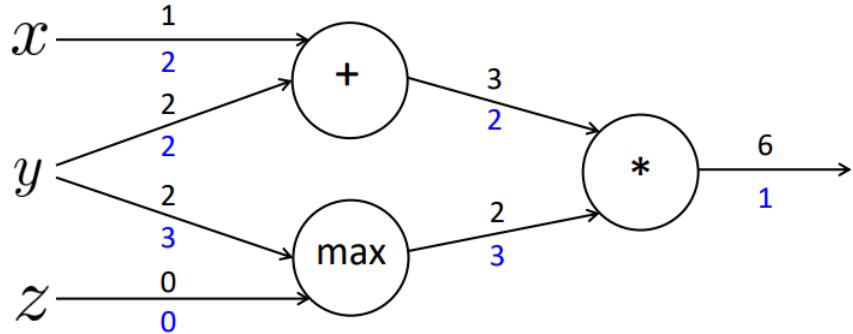


Figura 24

Assim, geramos a seguinte atualização das derivadas da função em relação a cada uma das variáveis de entrada:

$$\frac{\partial f}{\partial x} = 2$$

$$\frac{\partial f}{\partial y} = 3 + 2 = 5$$

$$\frac{\partial f}{\partial z} = 0$$

Portanto, na prática, *backpropagation* é apenas uma aplicação recursiva da regra da cadeia por toda a rede neural baseando-se nos valores gerados das camadas mais finais até as camadas mais iniciais. Em suma, *backpropagation* das redes neurais é equivalente ao algoritmo de gradiente descendente dos problemas de regressão.

13.3.3 Algoritmo

Agora, podemos verificar como funciona esse algoritmo na prática.

Realizando uma análise do algoritmo, temos que na linha 2 inicializamos $\Delta_{ij}^{(l)}$ com zeros, gerando uma matriz de zeros. No *loop for* inicializamos as variáveis $a^{(1)}$ com os valores da *input layer* e, depois realizamos *forward propagation* para computar os valores de $a^{(l)}$. Para cada valor de $a^{(l)}$ definido, computamos os valores de $\delta^{(l)}$ através de *backpropagation*, na linha 6 e 7 sabendo que

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l-1)}) \cdot a^{(l)} \cdot (1 - a^{(l)})$$

Computamos os valores de $\delta^{(l)}$ começando na camada L da rede neural até a camada 2. Utilizamos os valores de delta para armazenar o erro presente em cada um dos vetores dos nodos de ativação $a^{(l)}$.

Por fim, na linha 8, calculamos os valores de $\Theta_{ij}^{(l)}$ e determinamos o valor da derivada de $J(\Theta)$, armazenando em $D_{(ij)}^{(l)}$ nas linhas 11 e 13.

Algorithm 7 Backpropagation Algorithm

```

1: procedure BACKPROPAGATION(Training set  $[(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})]$ )
2:   Inicializar  $\Delta_{ij}^{(l)} = 0$                                  $\triangleright$  para todos  $l, i, j$  usado para computar  $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$ 
3:   for  $i = 1$  to  $m$  do
4:     Inicializar  $a^{(1)} = x^{(i)}$ 
5:     Realizar forward propagation para computar  $a^{(l)}$            $\triangleright$  para  $l = 2, 3, \dots, L$ 
6:     Usando  $y^{(i)}$ , computar  $\delta^{(L)} = a^{(L)} - y^{(i)}$ 
7:     Computar  $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ 
8:      $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ 
9:   end for
10:  if  $j \neq 0$  then
11:     $D_{(ij)}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$            $\triangleright \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{(ij)}^{(l)}$ 
12:  else
13:     $D_{(ij)}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ 
14:  end if
15: end procedure

```

13.4 Otimizadores

Em algumas implementações do gradiente descendente, podemos encontrar diferentes formas de otimizações. A seguir estão listadas algumas formas de implementações.

- Gradient Descent;
- Stochastic Gradient Descent;
- Mini-Batch Gradient Descent;
- Momentum;
- Nesterov Accelerated Gradient;
- AdaGrad;
- Adam;

Esses sistemas de otimização serão abordados e detalhados em seções seguintes.

13.5 Verificação do gradiente

Usamos o método de verificação do gradiente para assegurar que o algoritmo de *backpropagation* está funcionando correntemente.

$$\boxed{\frac{\partial}{\partial \Theta} \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}}$$

Usamos a expressão acima para computar todos os valores de Θ_j e para isso usamos valores pequenos para epsilon, como por exemplo, $\epsilon = 10^{-4}$.

Verificamos se os valores armazenados de Θ_j retornados pela expressão se aproxima dos valores de D retornados pelo algoritmo de *backpropagation*. Contudo, uma vez confirmado que o algoritmo funciona corretamente, não precisamos verificar novamente, pois o algoritmo de verificação é muito lento.

13.6 Inicialização aleatória

Para o melhor funcionamento do algoritmo de *backpropagation* devemos inicializar os vetores $\Theta_{ij}^{(l)}$ aleatoriamente de forma que $\Theta_{ij}^{(l)} \in [-\epsilon, \epsilon]$.

Nota-se que o valor de ϵ utilizado é o mesmo valor que foi utilizado no método de verificação do gradiente.

Com essa inicialização, garantimos que os valores de teta são simétricos e adaptados para o melhor funcionamento do algoritmo de *backpropagation*.

13.7 Organização do conhecimento

Como foi visto até agora, nós temos três tipos de estruturas básicas em uma arquitetura de uma rede neural:

- Número de unidades de entrada (*input units*): dimensão do vetor de entradas x ;
- Número de unidades de saída (*output units*): dimensão do vetor de saída (*classes*);
- Número de unidades intermediárias (*hidden units*) por camada: são definidas a partir da complexidade do problema e, geralmente, quando temos mais de uma camada, cada uma delas devem ter o mesmo número de unidades.

A partir dessas definições mencionadas, podemos, novamente, estruturar a lógica que deve ser seguida ao realizarmos o treino de uma rede neural. Abaixo, estão divididos em passos o algoritmo de treino utilizando *backpropagation*.

1. Aleatoriamente inicializar os pesos $\Theta_{ij}^{(l)}$ (Seção 13.6);
2. Implementar o método de *forward propagation* para computar os valores da função hipótese $h_\Theta(x^{(i)})$ para todos os valores de $x^{(i)}$ (Seção 13);
3. Implementar a função custo (Seção 13.2);
4. Implementar o método de *backpropagation* para computar os valores das derivadas parciais $\frac{\partial}{\partial \Theta}$ (Seção 13.3);
5. Utilizar o método de verificação do gradiente para confirmar que o método de *backpropagation* está funcionando corretamente. Após executar uma vez, desabilitamos a verificação (Seção 13.5);
6. Utilizar o algoritmo de gradiente descendente ou algum outro método de minimização mais otimizado para minimizarmos a função custo utilizando os valores de teta (Seção 13.4).

14 Aplicação de algoritmos de *machine learning*

14.1 Valoração de algoritmos de aprendizagem

Muitas vezes, podemos ter alguns problemas durante o teste das funções de treino. Para isso, é possível realizar alguns levantamentos a respeito do problema realizando as seguintes atividades:

- Aumentar o número de exemplos de treino;
- Diminuir o números de parâmetros;
- Adicionar parâmetros;
- Tentar parâmetros polinomiais;
- Aumentar ou diminuir o valor de λ .

Existem outros métodos que podem ser utilizados para valorar a função hipótese, ou seja, verificar a acurácia dos conjuntos de treino e teste. Assim, para cada um dos conjuntos teremos duas funções custo; uma para o conjunto de treino: $J_{treino}(\theta)$; e outra para o conjunto de teste $J_{test}(\theta)$.

Com isso, para regressão linear e para algoritmos de classificação, temos as seguintes equações para computar o erro de cada um dos dois métodos.

1. Para regressão linear:

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_{\theta}(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

2. Para classificação:

$$err(h_{\theta}(x), y) = \begin{cases} 1 & \text{se } h_{\theta}(x) \geq 0.5 \text{ e } y = 1 \text{ ou } h_{\theta}(x) < 0.5 \text{ e } y = 1 \\ 0 & \text{caso contrário} \end{cases}$$

Calculamos a média do erro do conjunto de teste:

$$TesteErro = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_{\theta}(x_{test}^{(i)}), y_{test}^{(i)})$$

Cada uma dessas formas nos retorna a proporção que os nossos dados foram erroneamente classificados.

Usualmente, dividimos o *dataset* em três diferentes conjuntos:

- Conjunto de treino (60%);
- Conjunto de *cross-validation* (20%);
- Conjunto de teste (20%).

O conjunto de treino é utilizado efetivamente para treinar o nosso modelo de rede neural. O conjunto de *cross-validation* é utilizado para testar o nosso modelo enquanto realizamos o treino, para que possamos ajudar hiperparâmetros e controlar os casos de *overfitting* e *underfitting* que podem, eventualmente vir a ocorrer. O conjunto de teste, serve, essencialmente para testarmos o modelo e verificarmos o quanto bem ele está generalizando dados que não foram previamente vistos.

Podemos calcular os erros para cada um desses três conjuntos, resultando em três diferentes valores: $J_{train}(\theta)$, $J_{cv}(\theta)$ e $J_{test}(\theta)$. Com isso, usamos os valores dos erros de $J_{cv}(\theta)$ para ajustarmos o grau do polinômio a fim de que possamos atingir o menor erro possível para $J_{test}(\theta)$.

14.2 Curvas de aprendizado

Nesta seção iremos interpretar gráficos que representam curvas de aprendizado analisando os valores de $J_{train}(\theta)$, $J_{cv}(\theta)$ e $J_{test}(\theta)$ para concluir se o treino está bem ajustado, com *overfitting* ou *underfitting*.

Em primeira análise, sabemos que para uma quantidade pequena de dados de entrada o valor do erro esperado é próximo de zero, pois o modelo aprende cerca de 100% do conjunto de treino. Porém, quando aumentamos o tamanho do conjunto de treino este erro tende a aumentar, muitas vezes, tanto para o conjunto de treino, quanto para o conjunto de teste. Para isso, podemos analisar dois tipos de situações esperadas: *high bias* e *high variance*.

Na representação a seguir que representa o erro da função J para casos de treino e teste de acordo com o grau do polinômio d . Percebe-se que quanto maior o grau do polinômio, menor é a taxa de erro no conjunto de treino e maior é a taxa de erro no conjunto de teste, e quanto menor o grau do polinômio maior é a taxa de erro do conjunto de treino e maior é a taxa de erro do conjunto de teste. Assim, devemos ajustar o grau do polinômio de forma que ele seja grande suficiente para evitar os casos de *underfitting* e *overfitting*.

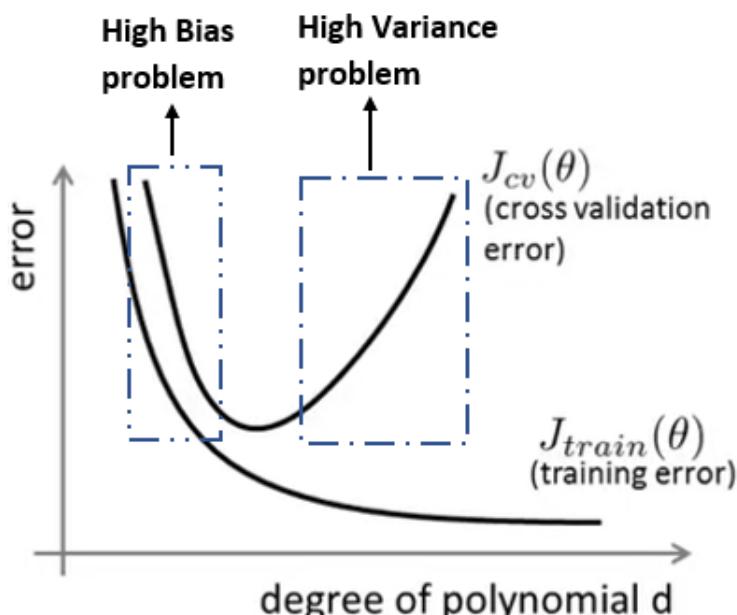


Figura 25: Representação das curvas $J_{cv}(\theta)$ e $J_{train}(\theta)$ relacionando o grau do polinômio d com a taxa de erro de cada uma dessas funções.

14.2.1 High Bias

Em casos de *underfitting* teremos a seguinte situação apresentada na Figura 26

More on Bias vs. Variance

Typical learning curve for high bias(at fixed model complexity):



Figura 26: Representação da curva de aprendizado para o caso de *high bias*

14.2.2 High Variance

Em casos de *overfitting* teremos a seguinte situação apresentada na Figura 27

More on Bias vs. Variance

Typical learning curve for high variance(at fixed model complexity):

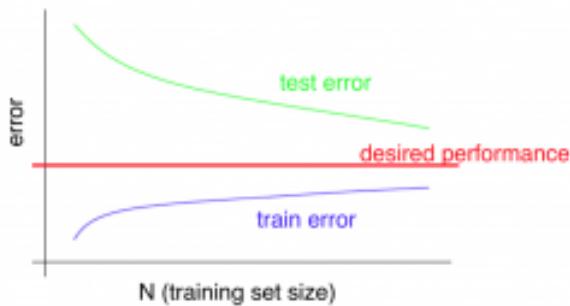


Figura 27: Representação da curva de aprendizado para o caso de *high variance*

14.3 Decisões a serem tomadas

Caso desejamos ajustar o nosso algoritmo de treino da melhor forma possível, podemos dividir a tomada de decisão a repeito de que tipo de ajuste devemos realizar da seguinte forma:

- Aumentar o número de exemplos de treino: ajusta em casos de *high variance*;
- Diminuir o números de parâmetros: ajusta em casos de *high variance*;
- Adicionar parâmetros: ajusta em casos de *high bias*;
- Tentar parâmetros polinomiais: ajusta em casos de *high bias*;
- Diminuir o valor de λ : ajusta em casos de *high bias*;
- Aumentar o valor de λ : ajusta em casos de *high variance*;

14.4 Diagnosticando Redes Neurais

Como foi visto nas seções anteriores, os problemas de *underfitting* e *overfitting* podem acontecer por diversos motivos e discutimos maneiras de podermos solucioná-los.

Em uma rede neural, esses problemas também podem ocorrer. O primeiro caso está relacionado à possibilidade de *underfitting*, e geralmente isso ocorre devido ao fato da baixa quantidade de parâmetros (computacionalmente mais barato). O segundo caso está relacionado com a possibilidade de *overfitting*, e geralmente isso ocorre devido ao fato da grande quantidade de parâmetros (computacionalmente mais caro), porém, neste caso, o problema pode ser facilmente corrigido através de regularização.

14.5 Support Vector Machines (SVMs)

Support Vector Machines, ou SVMs, podem ser usadas para tanto para problemas de regressão quanto problemas de classificação. O objetivo principal de uma SVM é encontrar um hiperplano em um espaço N -dimensional, sendo N o número de parâmetros, que classifica de forma distinta cada um dos dados.

Na Figura 28 a seguir, podemos perceber a funcionalidade de uma SVM na busca de um hiperplano ótimo (imagem à direita).

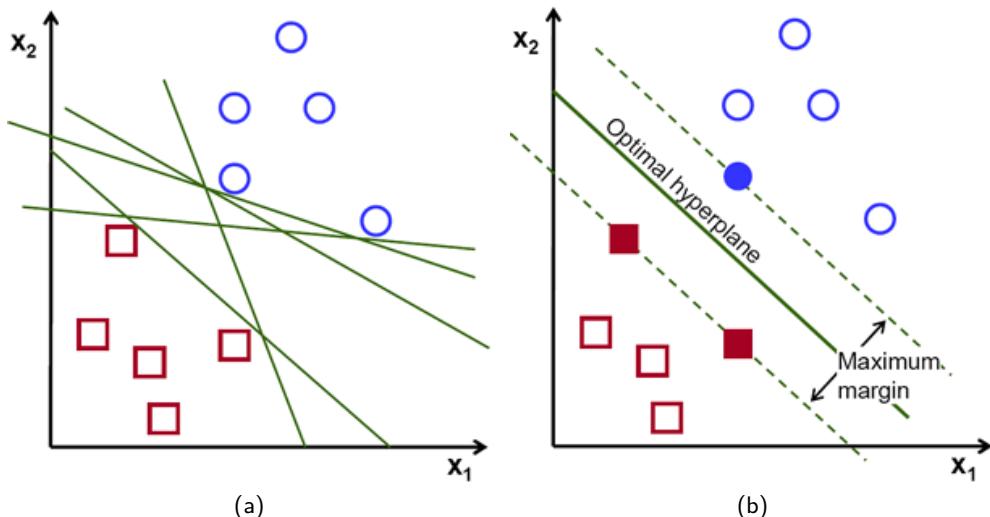


Figura 28: Exemplificação da utilização de uma SVM para encontrar o hiperplano ótimo que divide duas classes distintas de dados. Na imagem à esquerda temos todos os hiperplanos possíveis que distinguem as duas classes de dados e na imagem à direita temos o hiperplano ótimo que as distinguem.

Nosso objetivo é encontrar um plano que possui a margem máxima, ou seja, que possui a máxima distância entre os pontos de ambas as classes.

Hiperplanos são limites de decisão que ajudam a classificar as classes de dados. Além disso, a dimensão de um hiperplano depende do número de parâmetros.

Com isso, temos a definição de *support vectors*. *Support vectors* são pontos que estão mais próximos ao hiperplano e influenciam a posição e a orientação desse hiperplano. Usando-os, podemos maximizar a margem do classificador.

15 Naive Bayes

Naive Bayes é um algoritmo de classificação que utiliza aprendizado supervisionado. O algoritmo é chamado de "naive" (do inglês, ingênuo) porque ele realiza uma suposição ingênua de que cada dado é independente dos outros, o que não é verdade na vida real. Portanto, o algoritmo através do teorema de Bayes, realiza decisões que classificam os dados a partir de valores de probabilidade gerados a partir da suposição ingênua de independência dos dados.

O teorema de Bayes nos ajuda a encontrar o valor de probabilidade da hipótese dado um conhecimento prévio que pode ter interferência. Em outras palavras:

$$P(H|e) = \frac{P(e|H)P(H)}{P(e)}$$

onde:

- $P(H|e)$ (*posterior*): o quanto provável é a nossa hipótese dada a evidência observada.
- $P(e|H)$ (*likelihood*): o quanto provável é a evidência dada que a hipótese é verdadeira.
- $P(H)$ (*prior*): o quanto provável era a hipótese antes de observar a evidência.
- $P(e)$ (*marginal*): o quanto provável é a nova evidência sob todas as possíveis hipóteses.

Parte IV

Unsupervised Learning

16 ***Clustering***

Clustering é um método de aprendizado não supervisionado. A principal diferença entre *supervised learning* e *unsupervised learning* é que no método não supervisionado não passamos dados previamente classificado, em outras palavras, uma entrada para um algoritmo sem supervisão seria apenas o conjunto de treino x_1, x_2, \dots, x_n .

As principais aplicações de *clustering* são:

- Segmentação de mercado;
- Análise de redes sociais;
- Organização de *clusters* de computadores (*datacenters*);
- Análise de dados astronômicos.

O primeiro algoritmo de aprendizado não supervisionado que iremos discutir é chamado de *K-Means Algorithm* que será apresentado na seção seguinte.

16.1 ***K-Means Algorithm***

K-Means Algorithm é um dos algoritmos mais populares e são amplamente utilizados a fim de, automaticamente, agrupar dados em subgrupos denominados *clusters*.

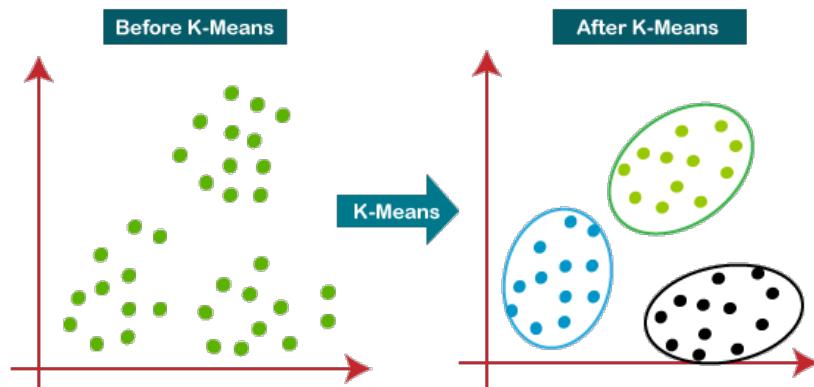


Figura 29: Representação de *clustering* através do algoritmo de *K-Means*. Percebe-se que, com a aplicação do algoritmo os dados passam a formar conjuntos diferentes, na figura, representados pelas cores azul, verde e preto.

Antes de discutir o algoritmo, podemos discutir o funcionamento desse método de *machine learning* para os três *clusters* da imagem acima.

1. Randomicamente inicializar três pontos no conjunto de dados. Esses pontos serão chamados de *cluster centroids*;

2. Atribuição do *cluster*: atribua todos os exemplos em um dos três grupos baseado em qual *centroid* os exemplos estão mais próximos;
3. Mover os *centroids*: computar as médias de todos os pontos dentro de cada um dos três *centroids*, e mover os *centroids* para os pontos que representam as médias;
4. Executar os passos (2) e (3) até convergir.

Para o algoritmo, teremos as seguintes variáveis principais:

- K : número de *clusters*;
- $x^{(1)}, x^{(2)}, \dots, x^{(m)}$: conjunto de treino, onde $x^{(i)} \in \mathbb{R}^n$,

Algorithm 8 Algoritmo *K-Means*

```

1: procedure
2:   Randomicamente inicializar  $K$  cluster centroids  $\mu_1, \mu_2, \dots, \mu_K \in \mathbb{R}^n$ 
3:   repeat
4:     for  $i = 1$  to  $m$  do
5:        $c^{(i)} :=$  índice (de 1 até  $K$ ) do cluster centroid mais perto de  $x^{(i)}$ 
6:     end for
7:     for  $k = 1$  to  $K$  do
8:        $\mu_k :=$  média dos pontos atribuídos ao cluster  $k$ 
9:     end for
10:    until convergir
11: end procedure

```

No algoritmo, percebemos que existem dois *loops*. O primeiro, realiza a etapa (2) descrita acima da seguinte forma:

$$c^{(i)} = \operatorname{argmin}_k \|x^{(i)} - \mu_k\|^2 = \| (x_1^{(i)} - \mu_{1(k)})^2 + (x_2^{(i)} - \mu_{2(k)})^2 + \dots \|$$

No segundo *loop* realiza a etapa (3) e pode ser descrita da seguinte forma:

$$\mu_k = \frac{1}{n} [x^{(k_1)} + x^{(k_2)} + \dots + x^{(k_n)}]$$

Onde cada valor de $x^{(k_1)}, x^{(k_2)}, \dots, x^{(k_n)}$ são os exemplos de treino atribuídos por μ_k

Depois de um número de iterações, o algoritmo irá convergir e as posições dos *centroids* não serão mais alteradas.

16.2 Otimização

Através dos parâmetros apresentados na seção anterior, podemos definir a função custo.

$$J(c^{(i)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k) = \frac{1}{m} \sum_{i=1}^m \|x^{(i)} - \mu_{(c^{(i)})}\|^2$$

Nosso objetivo de otimização é minimizar todos os parâmetros da função custo descrita acima, em outras palavras, desejamos

$$\min_{c, \mu} J(c, \mu)$$

16.3 Inicialização

Um método recomendado para o algoritmo $K - Means$ é inicializar aleatoriamente os *cluster centroids*. Para isso, devemos considerar os seguintes requisitos:

- $K < m$: O número de *clusters* deve ser menor que o número de exemplos de treino;
- Aleatoriamente escolher K exemplos de treinos;
- Definir μ_1, \dots, μ_K serem iguais aos K exemplos.

Para escolher o número de *clusters*, usa-se um método chamado *elbow method*, o qual se analisa a curva da função custo J e o número de *clusters* K , representado na Figura 30. A função custo deve decrescer de acordo com o aumento do número de *clusters* até tender a zero. Escolhemos um valor para K no ponto em que a função custo começa a se estabilizar.

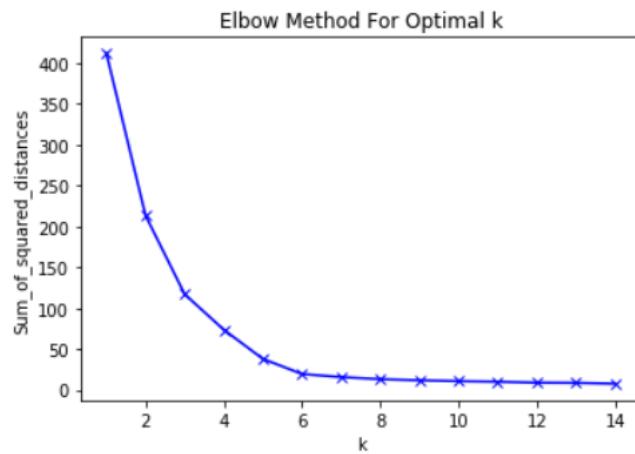


Figura 30: Representação do método de escolha do valor ótimo de K para o algoritmo KNN.

Uma outra forma de escolher o número de *clusters* é de acordo com o objetivo que desejamos atingir com o uso deles.

17 K-Nearest Neighbors

K-Nearest Neighbors (KNN) é um classificador que se baseia na similaridade dos dados próximos. Em outras palavras, classifica os dados em grupos de acordo com a similaridade entre eles e o quão próximo estão entre eles.

Na Figura 31 abaixo, podemos perceber como é realizada essa classificação. Cada uma das cores da imagem representa um conjunto de dados que possui similaridade entre si.

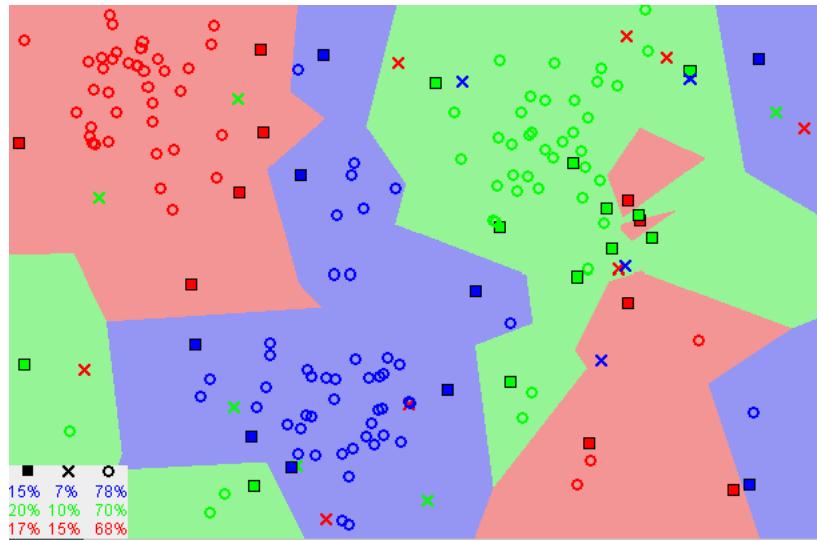


Figura 31: Representação das similaridades entre os conjuntos

A ideia de similaridade menciona é baseada na distância entre os pontos no gráfico.

17.1 Algoritmo

A seguir, será apresentado o algoritmo básico para a classificação de dados utilizando o método KNN.

1. Carregar os dados
2. Inicializar K como o número de "vizinhos" escolhidos.
3. Para cada um dos pontos: calcular a distância entre o atual ponto em relação aos outros e adicionar a distância e o índice do ponto a uma coleção ordenada
4. Ordenar a coleção ordenada de distâncias e índices de forma crescente das distâncias
5. Retornar os rótulos dos K primeiros valores da sequência

17.2 Escolhendo o valor correto para K

Para escolher o valor de K rodamos o algoritmo diversas vezes de forma a encontrar o valor de K que reduza o número de erros nos exemplos de teste.

É importante mencionar, que quanto menor o valor de K , há maior possibilidade de *overfitting* e quanto maior esse valor, maior a possibilidade de *underfitting*.

18 Redução de dimensionalidade

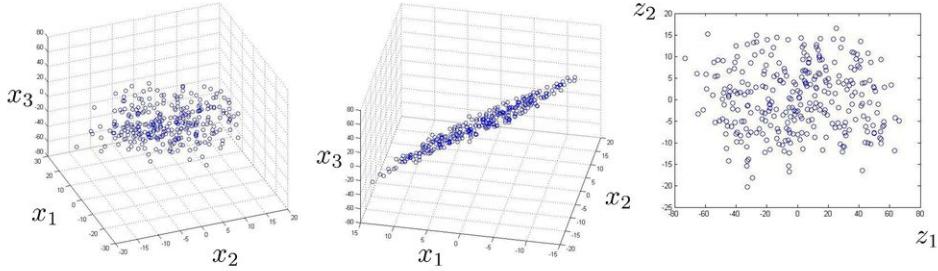
18.1 Compressão de dados

Em um conjunto de dados, muitas vezes podemos ter diversos dados redundantes e, para isso, devemos reduzir a quantidade desses dados a fim de facilitar a compreensão do conjunto o qual estamos trabalhando.

Assim, nós selecionamos dados que estão correlacionados e os colocamos em uma única linha que possa descrever o comportamento de ambos. Com redução de dimensionalidade, podemos reduzir o total de dados guardados aumentando a memória disponível e, muitas vezes, acelerando o processamento do algoritmo de aprendizagem.

Data Compression

Reduce data from 3D to 2D



Andrew Ng

Figura 32: Representação de uma compressão de dados. Na figura, percebe-se que houve a redução na dimensionalidade dos dados. Os dados previamente em três dimensões foram convertidos para duas dimensões. Essa conversão se dá através de um algoritmo que será apresentado nas seções seguintes chamado PCA.

18.2 Visualização

Quando realizamos redução de dimensionalidade dos dados, desejamos visualizá-los em duas ou, no máximo, três dimensões. Para isso, precisamos encontrar novos valores z_1, z_2 (ou z_3) para que possamos resumir esses dados efetivamente em menos dimensões.

18.3 Análise do componente principal (PCA)

O algoritmo de análise do componente principal (do inglês, *Principal Component Analysis Algorithm* (PCA)) é um algoritmo de redução de dimensionalidade de dados. Como está representado na Figura 32, o algoritmo busca comprimir os dados relacionando cada um dos dados em uma só semelhança, em outras palavras, busca reduzir a média de todas as distâncias de cada um dos valores em relação a linha projetada. Essa redução é chamada de erro de projeção.

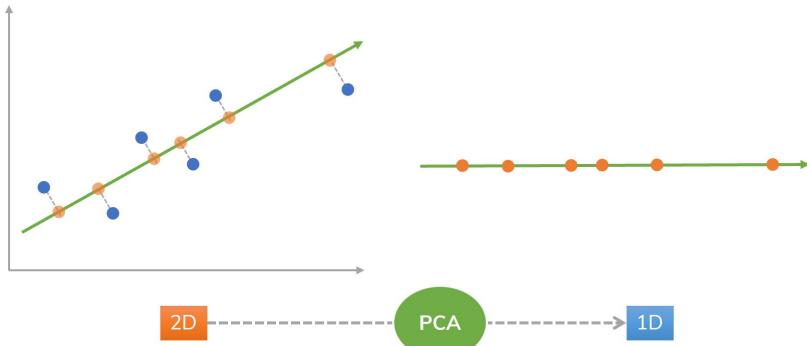


Figura 33: Representação da aplicação do algoritmo de PCA. Percebe-se que o algoritmo busca encontrar uma relação entre os pontos e fim de projetar os dados sobre essa relação. Na figura, os pontos em azuis são relacionados e projetados sobre a reta em verde, gerando os pontos em laranja sobre a reta.

19 Aprendizado por reforço (*Reinforcement learning*)

19.1 Visão geral

Nesta seção será apresentado mais um método de aprendizado de máquina chamado aprendizado por reforço (do inglês, *Reinforcement Learning* (RL)). RL ensina um agente a como escolher uma ação que faça sentido de acordo com o ambiente que ele esteja inserido (e.g. escolher uma movimentação de peça adequada em um jogo de tabuleiro) a fim de maximizar a recompensa que esse agente recebe ao longo do tempo.

Para isso, precisamos definir alguns elementos essenciais para a implementação de um algoritmo de RL (representado na Figura 34).

- Agente: o que o programa está exatamente treinando a fim de realizar alguma tarefa específica;
- Ambiente: o mundo, real ou virtual, no qual o agente realiza suas ações;
- Ação: um movimento realizado pelo agente. Essa movimentação muda o estado do ambiente;
- Recompensa: a valoração de uma ação realizada pelo agente. Essa valoração pode ser positiva ou negativa.

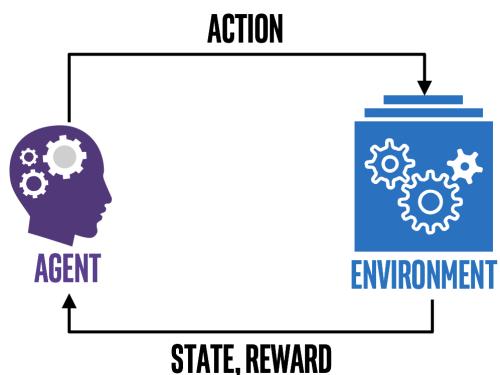


Figura 34: Representação de um esquema básico de um algoritmo de RL. Percebe-se que o agente através de ações realizadas sobre um ambiente determinado atualiza o estado do ambiente e recebe recompensas (positivas ou negativas) de acordo com a ação realizada.

Com essa definições, percebe-se que o processo de aprendizado do agente se dá por meio de tomada de decisões baseadas no ambiente e nas recompensas. Na próxima seção será apresentada as ideias fundamentais de exploração do ambiente em RL.

19.2 *Exploration e Exploitation*

Quando o agente realiza explorações no ambiente a fim de conhecer o território no qual ele está inserido, ele realiza dois tipos de ações que são determinantes para o seu aprendizado: *exploration* e *exploitation*.

Apesar de parecerem redundantes, os dois termos possuem uma grande diferença. Quando o agente toma decisões baseadas em *exploration*, o agente toma decisões puramente aleatórias, com o intuito de encontrar melhores ações para o estado no qual ele se encontra. Isso é importante para possibilitar ao agente descoberta de novas estratégias, podendo, assim, expandir o seu leque de opções de ações. Em *exploitation* o agente toma decisões baseadas em valorações que foram previamente estabelecidas de estados previamente alcançados. Isso é importante para maximizar a tomada de decisão do agente, pois ele saberá se determinado estado é "bom" ou "ruim" ao longo da compreensão do ambiente.

19.3 *Markov Process*

Nesta seção iremos discutir o princípio de formação dos métodos que envolvem *Reinforcement Learning*: *Markov process*. Contudo, antes disso, é necessário introduzir alguns conceitos chaves desse processo: a Propriedade de Markov e a Cadeia de Markov.

19.3.1 Propriedade de Markov

Na teoria das probabilidades e estatística, o termo "propriedade de Markov" se refere à propriedade sem memória de um processo estocástico, em outras palavras, em um processo aleatoriamente determinado.

Para o melhor entendimento da Propriedade de Markov, podemos descrevê-la da seguinte forma:

$$P(X(t+1) = j | X(0) = i_0, X(1) = i_1, \dots, X(t) = i) = P(X(t+1) = j | X(t) = i)$$

A equação acima representa uma situação de um estado X no tempo $t + 1$ que depende apenas do estado precedente, o estado X no tempo t . Em outras palavras, a propriedade de Markov nos diz que o próximo estado de um processo de Markov independe dos estados precedentes do anterior, porque o estado imediatamente anterior irá conter todas as informações dos estados precedentes.

19.3.2 Cadeia de Markov

A Cadeia de Markov é um modelo estocástico que descreve uma sequência de possíveis eventos os quais são dependentes de uma possibilidade de ocorrência. E essa possibilidade só depende do evento prévio a esta ação.

Quando usamos a Propriedade de Markov em um processo aleatório, chamamos esse uso de Cadeia de Markov, podendo ser definida da seguinte forma:

A Cadeia de Markov é uma tupla (S, P) onde:

- S é um conjunto de estados;
- $P(s, s')$ é uma transição de estado cujo peso é a probabilidade. A probabilidade de transição do estado s no tempo t para o estado s' no tempo $t + 1$.

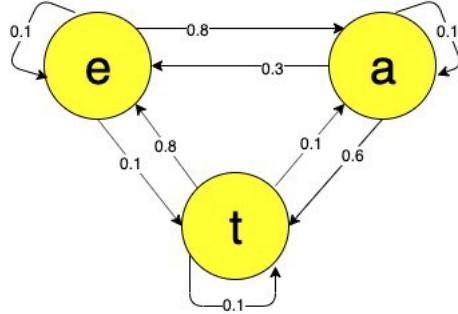


Figura 35: Representação de uma Cadeia de Markov com os estados e, a, t onde cada aresta representa uma transação de estado e seus pesos representam a probabilidade de transição.

19.4 Markov Decision Processes (MDPs)

MDP é um dos tópicos mais importantes para o entendimento de RL. MDP é um método que resolve a maioria dos problemas de RL com tomadas de decisões discretas. Com esse método, um agente atinge uma política ótima para receber o maior número de recompensas ao longo do processo de exploração do ambiente.

Como foi apresentado na seção anterior na qual discutimos as definições de Propriedade e Cadeia de Markov, uma Cadeia de Markov trabalha com as transições entre os estados e as probabilidades de transição relacionadas aos mesmos. MDP difere da Cadeia de Markov pois, agora, os estados não dependem apenas do estado imediatamente posterior, mas sim, de todas as ações que foram executadas a partir do estado atual.

Com isso, o objetivo principal do MDP é treinar um agente a fim de encontrar a melhor política possível acumulando o maior número de recompensas a partir de tomadas de decisões em um ou mais estados. Podemos definir, formalmente MDP da seguinte forma:

MDP é uma 5-upla (S, A, P, R, γ) , onde:

- S : conjunto de estados;
- A : conjunto de ações;
- $P(s, a, s')$: probabilidade de uma ação a no estado s levar ao estado s' no tempo $t + 1$;
- $R(s, a, s')$: recompensa recebida imediatamente após a transição do estado s para o s' a partir de uma ação a ;
- γ : fator de desconto o qual é usado para gerar uma recompensa relativa. Esse valor está entre 0 e 1 e quantifica a diferença de importância de recompensas imediatas e recompensas futuras.

Em outras palavras, MDP pode ser escrito como

$$\sum_{t=0}^{t=\infty} \gamma^t r(x(t), a(t))$$

19.4.1 Busca pela política ótima com MDP

Com MDP podemos fazer com que o nosso agente selecione a decisão ótima para determinado estado de ambiente. Iremos maximizar a recompensa do agente ao longo do tempo a fim de fazer com que ele atinja a política ótima, i.e. determinaremos qual é a melhor ação a ser tomada em cada estado.

Para determinar a melhor ação a ser tomada, iremos usar a equação de Bellman (*Bellman Optimality Equation* [4]) que nos possibilita estimar o valor ótimo para cada estado. A equação estima o valor de um estado computando as recompensas esperadas que cada estado pode gerar.

Abaixo está definida a equação de Bellman recursiva

$$V^*(s) = \max_a \sum_{s'} P(s, a, s') [R(s, a, s') + \gamma V(s')]$$

Onde:

- $P(s, a, s')$ é a probabilidade da transição do estado s para o s' a partir da escolha da ação a ;
- $R(s, a, s')$ é a recompensa imediata do estado s para o s' quando o agente escolhe a ação a ;
- γ é o fator de desconto (recursivo).

19.5 Monte-Carlo e *Temporal-Difference Learning*

Nesta seção serão apresentados dois métodos chaves de aprendizado de máquina através de RL: aprendizado de *Monte-Carlo* (MC) e *Temporal-Difference Learning* (TD).

19.5.1 Valoração de Monte-Carlo

O método de Monte-Carlo faz com que o agente aprenda interagindo com o ambiente e coletando diversas amostras. Essa coleção de amostras estão relacionadas as distribuições de probabilidades mencionadas $R(s, a, s')$ e $R(s, a, s')$.

Entretanto, Valoração de MC é uma forma de aprendizado baseado em testes, em outras palavras, um MDP sem a tupla P pode aprender por tentativa e erro, por meio de muitas repetições.

Nesse cenário, cada "tentativa" é chamado de episódio, e ele termina quando o estado final do MDP é atingido. No final, todos os valores de recompensa são atualizados na recompensa G_t , final de cada estado. Abaixo está representada a equação de Monte-Carlo que atualiza o valor de cada estado.

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)]$$

Onde:

- $V(S_t)$ é o valor do estado que desejamos estimar. Esse valor pode ser inicializado aleatoriamente ou baseado em alguma estratégia;
- G_t pode ser calculado da seguinte forma, onde T é o tempo de terminação:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T$$

- α é um parâmetro que influencia a convergência.

Sendo assim, temos diversas formas de atualizarmos os valores dos estados do ambiente. Essas atualizações são significantes para determinar a política do agente e otimizar a escolha da ação. Essas formas estão relacionadas ao momento de visita de cada estado, ou seja, como devemos atualizar o valor de um estado na primeira visita e em cada vez que atingimos o mesmo estado.

19.5.2 TD Learning

Como vimos na seção anterior, o método de Monte-Carlo exige que nós esperamos até o fim do episódio para determinar o valor do estado $V(S_t)$. No método de *Temporal-Difference*, esperamos até o próximo passo a ser tomado. Em outras palavras, no tempo $t + 1$, o método TD utilizado da recompensa observada R_{t+1} e, imediatamente, gera um valor, chamado *TD target* $R(t+1) + V(S_{t+1})$, atualizando o valor do estado $V(S_t)$ através do erro, chamado *TD error* $R(t+1) + V(S_{t+1}) - V(S_t)$.

Com isso, podemos definir a equação $TD(\lambda)$ na qual substituímos o valor de *TD target* por G_t da seguinte forma:

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t^{(\lambda)} - V(S_t)]$$

Onde:

$$G_t^{(\lambda)} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

Assim, com essas duas equações, Monte-Carlo(MC) e *Temporal-Difference*(TD) podemos definir um método de aprendizado por reforço que é uma combinação das duas, chamado de *Q – Learning*.

19.6 Q-Learning

Q-Learning é uma combinação entre os métodos de MC e TD vistos na seção anteriores. Nesse método, a cada passo, tomamos uma decisão gananciosa que tem como objetivo maximizar o valor da variável $Q(S_{t+1}, a)$, como está representado abaixo:

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, a)]$$

Para a implementação desse método, precisamos de dois fatores principais: a atualização do valor de *Q* - chamado de *Q-value* e um lugar para salvar esses valores, chamado de *Q-table*.

Em *Q-learning*, a cada passo tomamos a melhor decisão possível para o agente, ou seja, a decisão que terá o melhor impacto na atualização de *Q-value*. Esse tipo de tomada de decisão se chama *ϵ -greedy-policy*, onde $0 < \epsilon < 1$ é o grau de ganância do nosso agente, em outras palavras, quanto mais alto o seu valor, maior a quantidade de ações realizadas aleatoriamente.

No início da exploração, o agente toma decisões puramente aleatórias a fim de conhecer o ambiente no qual ele está inserido. E a cada tomada de decisão, atualiza-se o valor de cada estado *Q-value* na tabela *Q-table*. Portanto, é comum começar a exploração com um valor alto de ϵ , como, por exemplo, igual a 1, a fim de fazer com que o agente tome decisões 100% aleatórias.

Ao longo da exploração, o agente começa a conhecer melhor o ambiente, então podemos diminuir o valor de ϵ gradativamente ao longo do treino.

Contudo, a utilização desse tipo de implementação não é escalável, ou seja, para modelos extremamente complexos, é extremamente ineficiente a valoração dos estados e a busca pela política ótima. Para resolver esse tipo de problema, métodos derivados foram desenvolvidos utilizando redes neurais e o principal deles se chama *Deep Q-Learning* que será discutido na Seção 26.

Parte V

Otimizações no Aprendizado

20 Detecção de anomalias

20.1 Motivação

Suponha que tenhamos um conjunto de dados de treino $x^{(1)}, x^{(2)}, \dots, x^{(m)}$. E dado um novo exemplo de treino x_{test} queremos saber se esse dado pode ser considerado anormal ou anômalo.

Para isso, nós definimos uma espécie de "modelo" $p(x)$ que nos diz a probabilidade do exemplo não ser anômalo. Usamos, também uma *flag* ϵ (*epsilon*) que serve como uma divisão no nosso conjunto de dados que diferencia os dados anômalos e não anômalos.

Uma aplicação comum de detecção de anomalias é a detecção de fraudes:

- $x^{(i)}$ = dados das atividades do usuário i
- Modelo $p(x)$ dos dados
- Identifica usuários não usuais checando se $p(x) < \epsilon$

Assim, caso o exemplo $x^{(i)}$ seja considerado anômalo, o valor de $p(x) < \epsilon$, caso contrário $p(x) \geq \epsilon$.

Caso o detector de anomalias esteja detectando muitos exemplos anômalos, devemos diminuir o valor da *flag* ϵ .

20.2 Distribuição Gaussiana

A Distribuição Gaussiana, ou Distribuição Normal, é uma forma de distribuição uniforme em formato de sino que pode ser descrita através de uma função $\mathcal{N}(\mu, \sigma^2)$. Onde, dado um valor x , $x \sim \mathcal{N}(\mu, \sigma^2)$, representa a probabilidade na distribuição de x , com média μ e variância σ^2 .

A Distribuição Gaussiana, então, é parametrizada pela média e pela variância do conjunto de dados (Figura 36).

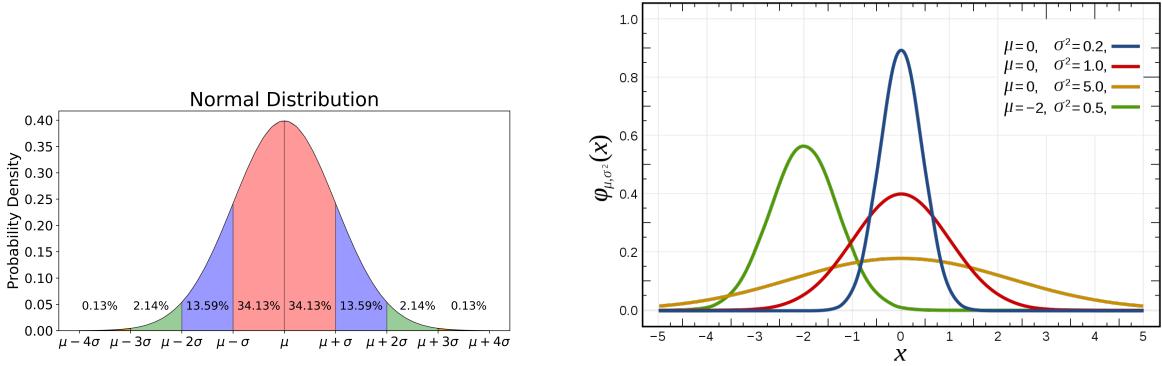
Além disso, μ representa o centro da curva (média) e a largura da distribuição é descrita por σ (desvio padrão).

A função da Distribuição Gaussiana é definida da seguinte forma:

$$p(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Podemos estimar o parâmetro μ de um dado conjunto de dados apenas calculando a média de todos os exemplos:

$$\mu = \frac{1}{m} \sum_{i=1}^m x^i$$



(a) Representação da Distribuição Gaussiana. O eixo x representa os valores dos exemplos de acordo com a média μ e com a variância σ^2 . O eixo y representa a densidade de probabilidade do exemplo.

(b) Representação de diferentes estruturas distribuições gaussianas de acordo com os valores dos parâmetros μ e σ^2 . Podemos perceber o efeito da alteração desses parâmetros nas estruturas das curvas.

Figura 36

Da mesma forma, podemos estimar o parâmetro σ^2 através da fórmula de erro quadrático a qual já estamos familiarizados.

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu)^2$$

20.3 Algoritmo

Dado um conjunto de exemplos de treino $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ onde cada exemplo é um vetor $x \in \mathbb{R}^n$.

$$p(x) = p(x_1; \mu_1, \sigma_1^2) p(x_2; \mu_2, \sigma_2^2) \dots p(x_n; \mu_n, \sigma_n^2)$$

Nesse caso, os valores das probabilidades acima são multiplicados pois os valores de x_i , de treino, são independentes entre si.

De outra forma, podemos escrever a expressão acima em forma de produtório, como segue:

$$p(x) = \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2)$$

Com isso, podemos definir o algoritmo de detecção de anomalias através desses valores de probabilidade.

1. Escolher os valores x_i que possam ser indicativos de exemplos de anomalia
2. Ajustar os parâmetros $\mu_1, \mu_2, \dots, \mu_n, \sigma_1^2, \sigma_2^2, \dots, \sigma_n^2$
3. Calcular $\mu_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)}$
4. Calcular $\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2$
5. Dado um novo exemplo x , calcular $p(x) = \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^n \frac{1}{\sigma_j \sqrt{(2\pi)}} e^{-\frac{1}{2}(\frac{x_j - \mu_j}{\sigma_j})^2}$
6. Caso o valor de $p(x)$ seja menor que ϵ , então o exemplo x é uma anomalia.

21 Aprendizado em larga escala

Na maioria das vezes, trabalhar com um grande conjunto de dados é muito melhor para qualquer tipo de problema que envolva aprendizado de máquina. Por isso, muitas vezes, iremos trabalhar com conjuntos de dados de tamanhos como $m = 100.000.000$ e, neste caso, o algoritmo de gradiente descendente terá que realizar operações com todos os 100 milhões de dados - o que torna esse método extremamente ineficiente e complexo. Queremos evitar tentar evitar isso criando algoritmos mais otimizados que serão descritos e detalhados nas seções seguintes.

21.1 *Batch Normalization*

Quando falarmos em normalização, estamos nos referindo à compactação de uma gama diversa de números em uma gama fixa, como foi visto, por exemplo, na Seção 20.2 quando falamos de Distribuição Normal.

Por exemplo, quando temos entradas com uma variância muito alta, desejamos normalizar esses valores a fim de gerarmos uma distribuição mais simples de trabalhar, colocando os valores em um mesmo alcance. Isso se chama, *input normalization*.

Redes neurais profundas, geralmente possuem uma sequência de camadas que, muitas vezes, somam milhares de neurônios. De camada a camada, esses valores podem se distinguir através das sequências operações, dificultando que as camadas da rede neural consigam fazer o trabalho de manter esses valores bem distribuídos e estáveis.

Outro problema é quando a escala de diferentes parâmetros da rede neural estão vinculados às atualizações do gradiente, o que pode levar ao problema de explosão e desaparecimento dos gradientes. Esses problemas estão relacionados, respectivamente ao crescimento descontrolado e infinito dos valores do, ou à queda descontrolada tendendo à zero dos valores do gradiente.

Geralmente, para evitar esse tipo de problema, desejamos desacoplar a escala dos parâmetros da rede neural das atualizações do gradiente e, para isso, podemos usar funções de ativação mais robustas e estáveis, como por exemplo, ReLU (Seção 13.1.5), escolhendo taxas de aprendizado ótimas e inicializando os parâmetros das redes neurais cuidadosamente. Todavia, quanto mais tarefas adicionamos, aumentamos a complexidade da rede neural, o que, muitas vezes, pode ser um problema.

Para solucionar esses problemas, utilizamos um método chamado *Batch Normalization*. Esse método tem como objetivo evitar gradientes instáveis, reduzir os efeitos da inicialização da rede neural na convergência do algoritmo de otimização e possibilitar o uso de taxas de aprendizado mais rápidas para acelerar a convergência do algoritmo de otimização do gradiente.

Para isso, devemos normalizar cada camada intermediária da rede neural de acordo com as entradas de cada camada, calculando a média e o desvio padrão de cada uma delas.

Assim, podemos, finalmente descrever o algoritmo de normalização *Batch Normalization*.

Algorithm 9 Algoritmo *Batch Normalization* (BN)

```

1: procedure (Valores de  $x$  sobre um mini-batch:  $\mathcal{B} = \{x_1, \dots, m\}$ ,
   Parâmetros a serem aprendidos:  $\gamma, \beta$ )
2:    $\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$                                       $\triangleright$  Média mini-batch
3:    $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$                     $\triangleright$  Variância mini-batch
4:    $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$                           $\triangleright$  Normalização
5:    $y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i)$                                  $\triangleright$  Escala e shift
6: end procedure

```

No algoritmo acima, as primeiras três equações calculam a média e o desvio padrão e depois normalizam as entradas, respectivamente. O valor de ϵ é um número que ajuda na estabilidade numérica, evitando que seja efetuada uma divisão por zero. A principal característica desse algoritmo é que a normalização acontece para cada um dos inputs no *batch* separadamente.

Na última equação, temos dois novos parâmetros, γ e β , que representam respectivamente o *scaling* e o *shift* dos valores. Isso limita o alcance dos valores, facilitando com que a próxima camada trabalhe com eles de forma mais simples. Esses valores são aprendidos pela rede neural e, assim, podemos ajustar esses valores de acordo com a distribuição.

21.2 Stochastic Gradient Descent (SGD)

SGD é uma alternativa mais eficiente e escalável ao método clássico de otimização gradiente descendente.

SGD pode ser escrito de maneira muito similar ao método de gradiente descendente clássico, visto.

$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_{\theta}(x^{(i)}) - y^{(i)})^2$$

A principal diferença entre o método clássico e o SGD é que o SGD considera cada exemplo de treino individualmente para calcular o custo antes de realizar a média dos custos, enquanto o método clássico calcula o custo de acordo com a média dos valores.

Com isso, podemos definir a função custo da seguinte forma:

$$J_{train}(\theta) = \frac{1}{m} \sum_{i=1}^m cost(\theta, (x^{(i)}, y^{(i)}))$$

J_{train} , agora, é apenas a média dos custos aplicado a todos os exemplos de treino.

Com esse tipo de metodologia, podemos otimizar o cálculo das derivadas parciais a fim de reduzir a função custo. Devido ao fato desse método ser estocástico, adicionamos aleatoriedade nas escolhas dos custos dos exemplos de treino. Para cada uma dessas escolhas, otimizamos os valores dos custos o que reduz de forma muito mais eficiente a complexidade da computação. Portanto, com o SGD, computamos a atualização do parâmetro θ para cada um dos exemplos de treino x_i em relação ao seu rótulo y_i , como podemos verificar abaixo:

$$\theta = \theta - \alpha \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)})$$

onde α é a taxa de aprendizado e ∇ é o gradiente do custo J .

Podemos visualizar na Figura 37 as diferenças entre as etapas de aprendizado entre o método clássico do gradiente descendente e o método estocástico. Percebemos, que SGD é mais "barulhento" em relação ao método clássico, porém, como não precisamos calcular as derivadas para cada um dos m exemplos de treino por iteração, o SGD acaba sendo mais eficiente.

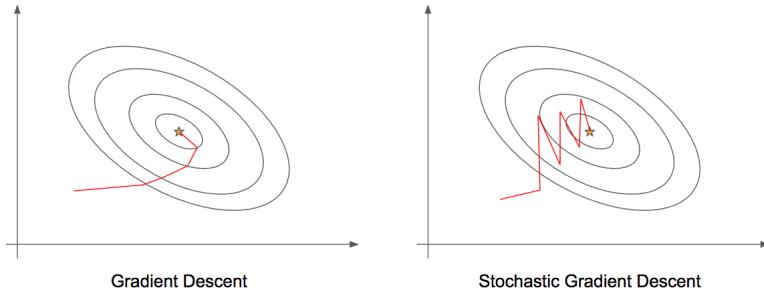


Figura 37

O algoritmo de SGD é muito similar ao algoritmo clássico, iremos atualizar os parâmetros Θ_j ao longo do aprendizado de acordo com a função custo, como segue:

Algorithm 10 Algoritmo *Stochastic Gradient Descent* (SGD)

```

1: procedure
2:   Aleatoriamente, "misturar" o conjunto de dados
3:   for  $i = 1$  to  $m$  do
4:      $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ 
5:   end for
6: end procedure

```

Esse algoritmo, diferentemente do método clássico, ajustará um exemplo de treino por vez. Com isso, podemos prosseguir com o algoritmo de gradiente descendente, sem, necessariamente examinar todos os m exemplos de treino antes.

Um dos problemas desse algoritmo é que, muitas vezes, ele não convergirá para o mínimo global do problema e, ao invés disso, irá vagar aleatoriamente ao redor desse mínimo, porém, na maioria das vezes, produz um retorno muito próximo ao esperado. Normalmente, o SGD varre o conjunto de dados entre uma ou dez vezes antes de chegar próximo ao mínimo global. Como podemos perceber na Figura 38, o SGD pode acabar travado em um mínimo local e isso resulta em dificuldade em otimizar ao máximo a função custo, pois o algoritmo não irá convergir para o mínimo global.

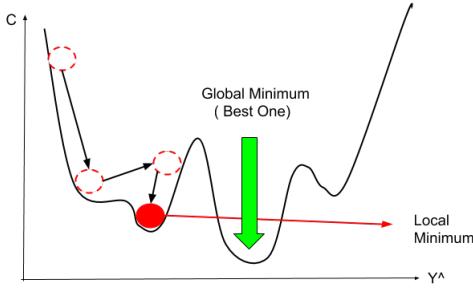


Figura 38: Representação da dificuldade de convergência do algoritmo SGD. Percebe-se que a minimização da função custo atinge um mínimo local devido às "voltas" que o algoritmo dá ao redor do mínimo global.

Podemos evitar esse tipo de problema incentivando a convergência do algoritmo. Para isso, podemos escolher valores de α diferentes a cada iteração, diminuindo o seu valor.

Com uma taxa de aprendizado menor, o algoritmo irá oscilar de maneira mais suave, sem grandes pulos em torno do mínimo global. Dessa forma, uma estratégia seria diminuir o valor de α a cada iteração, como segue:

$$\alpha = \frac{\text{const1}}{\text{iterationNumber} + \text{const2}}$$

Todavia, esse método não é muito utilizado devido a quantidade extra de parâmetros que devem ser ajustados.

21.3 Mini-batch Gradient Descent

Mini-batch Gradient Descent muitas vezes, pode ser mais eficiente que o SGD. Ao invés de usarmos todos os m exemplos de treino, como no método clássico, ou apenas um exemplo no SGD, usaremos algo entre essas duas opções. Selecionaremos b exemplos de treino. O valor de b , geralmente está entre 2-100.

Na figura a seguir, podemos verificar as diferenças entre o SGD e o *Mini-batch Gradient Descent*. Na imagem, percebe-se que o método *Mini-batch* converge mais rapidamente, principalmente devido ao fato de processar os dados em *batches* de forma vetorializada.

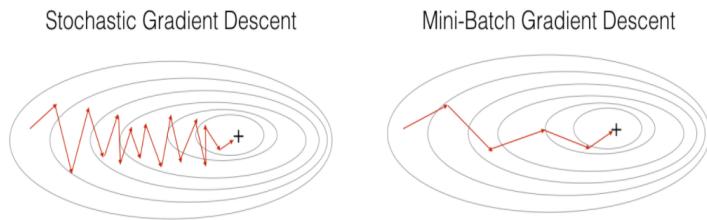


Figura 39

Por exemplo, se $b = 10$ e $m = 1000$, teremos o seguinte algoritmo:

Nesse algoritmo, estamos simplesmente somando dez exemplos por vez. A principal vantagem em

Algorithm 11 Algoritmo *Mini-batch Gradient Descent*

```
1: procedure
2:   Aleatoriamente, "misturar" o conjunto de dados
3:   for  $i = 1, 11, 21, 31, \dots, 991$  do
4:      $\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) \cdot x_j^{(k)}$ 
5:   end for
6: end procedure
```

computar mais de um exemplo por vez (em *batches*) é que podemos usar implementações vetorizadas sobre os exemplos em b .

21.4 SGD + Momentum

Como foi explicado, um dos problemas do SGD é que muitas vezes, o algoritmo pode ficar "preso" em um mínimo local próximo do global e não convergir de forma adequada para o nosso problema. Para isso, uma forma de otimização foi utilizar *momentum* a fim de incentivar a convergência.

Antes de definir esse método de otimização, podemos intuitivamente compreender como funciona o *momentum*. Para isso, podemos pensar em uma bola que rola em uma colina do ponto mais alto ao mais baixo, seguindo a inclinação. Desejamos que essa bola tenha a noção de onde ela está indo. Essa intuição pode ser relacionada aos valores dos parâmetros que desejamos atualizar, ou seja, esses parâmetros se comportariam como se fossem a tal bola.

Uma boa definição para o *momentum* é que é uma média móvel dos nossos movimentos, ou seja, como determinado corpo se comporta em determinado terreno de acordo com a velocidade e a aceleração proporcionada pelo mesmo. Dessa forma, a cada etapa do processo de otimização da função custo, temos vetores que representam o *momentum* do ponto em determinada parte do "terreno" da função custo, e esses vetores proporcionarão uma velocidade maior de convergência na direção a qual ele está apontando.

Na Figura 40 está representada a diferenciação no comportamento do SGD sem e com a utilização de *momentum*.

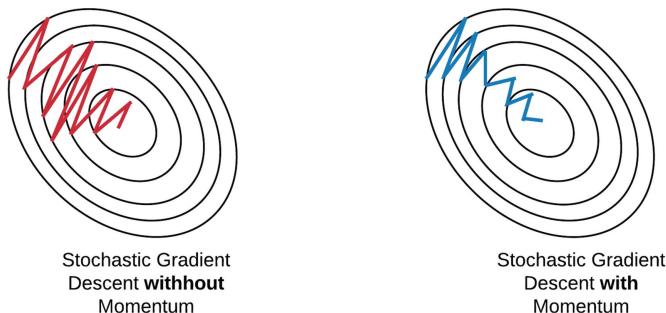


Figura 40: À esquerda representação no comportamento dos parâmetros sendo minimizados através do método SGD e à direita o mesmo método com a aplicação de *momentum*.

Podemos diferenciar os dois modelos de otimização da forma que segue:

Onde x_i representa os parâmetros a serem atualizados, v_i representa o *momentum* do ponto em

Tabela 4: Comparação entre os métodos *Gradient Descent* e *Normal Equation*

SGD	SGD + Momentum
$x_{t+1} = x_t - \alpha \nabla f(x_t)$	$v_{t+1} = \rho v_t + \nabla f(x_t)$
	$x_{t+1} = x_t - \alpha v_{t+1}$

determinada parte, $\nabla f(x_i)$ representa o gradiente da função custo e ρ representa o atrito do ponto em determinada parte (geralmente esse parâmetro é definido como 0.9 ou 0.99).

Contudo, um dos problemas que esse método de otimização proporciona é que, muitas vezes, quando os parâmetros se aproximam do mínimo, devido ao *momentum*, demoram para convergir pois ultrapassam esse mínimo. Para resolver esse problema, devemos desacelerar a atualização desses parâmetros de acordo que eles se aproximam do mínimo, o que será visto na seção seguinte.

21.5 Nesterov Accelerated Gradient (NAG)

O método NAG utiliza da mesma ideia vista na seção anterior, porém, como foi dito, desaceleramos a atualização dos parâmetros de forma a não ultrapassar de maneira muito significativa o valor de mínimo da função.

Na Figura 41, abaixo está representada a diferença no comportamento da atualização dos valores dos parâmetros.

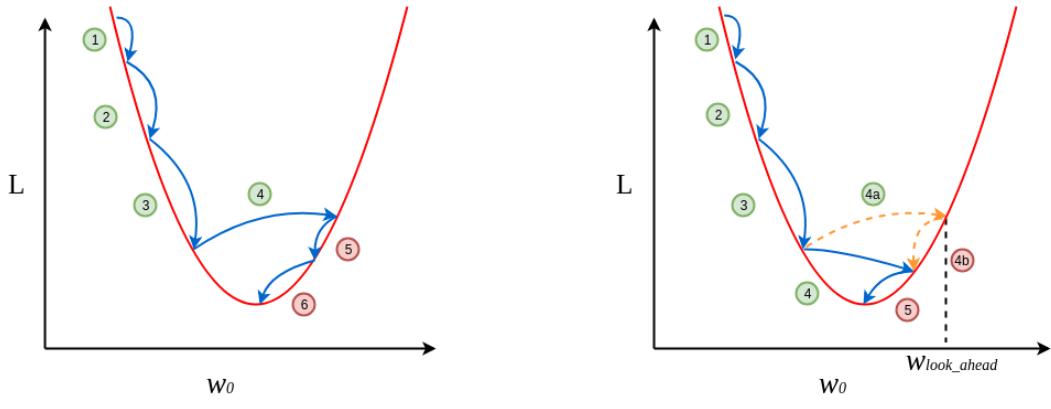


Figura 41: Exemplificação do método de Nesterov (à direita) para gradiente descendente utilizando *momentum* (à esquerda). Percebe-se que a cada passo, os valores da função custo diminuem tendendo ao mínimo. E quando esse valor se aproxima do mínimo, desacelera para facilitar a convergência.

Nesse método de otimização da função custo, primeiro olhamos para o onde o vetor atual *momentum* está apontando para, com ele, calcularmos o gradiente partindo desse ponto, como podemos verificar na Figura 42.

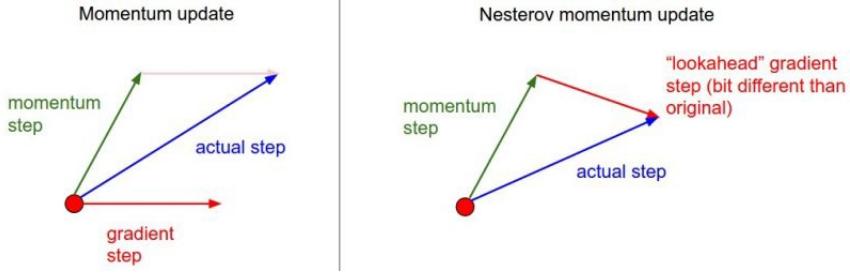


Figura 42: Representação da atualização dos vetores *moimentum* utilizando o método de Nesterov. Na imagem à esquerda, verificamos a atualização do vetor *momentum* a partir de um determinado ponto. Na figura à direita, verificamos a atualização do vetor *momentum* através do método de Nesterov, a partir do ponto onde o vetor está apontando.

O método de Nesterov utiliza o termo de *momentum* ρv_{t-1} para atualizar os parâmetros θ . Assim, se computarmos $\theta - \rho v_{t-1}$ temos a aproximação da próxima posição dos parâmetros θ . Para calcularmos a atualização dos parâmetros, primeiro devemos definir a atualização de v_t utilizando o método de Nesterov, como descrito abaixo.

$$v_t = \rho v_{t-1} + \alpha \nabla_\theta J(\theta - \rho v_{t-1})$$

$$\theta = \theta - v_t$$

onde, ρ é o termo definido como *momentum* (atrito, geralmente um valor em torno de 0.9), α é a taxa de aprendizado e ∇_θ é o gradiente de J em relação aos parâmetros θ .

Assim, com a aplicação do método de Nesterov *Momentum*, o algoritmo de otimização da função converge muito mais rapidamente, evitando o problema de aproximação ruim da convergência no SGD.

21.6 AdaGrad

Outro método de otimização é chamado de *Adaptive Gradient Algorithm* (Adagrad). Esse método adapta a taxa de aprendizado aos parâmetros, realizando atualizações menores nos parâmetros, diminuindo o "barulho" das atualizações.

Como não precisamos definir uma taxa de aprendizado α , a convergência é mais rápida e confiável. O método adapta as atualizações e a taxa de aprendizado automaticamente de acordo com os níveis da superfície, impossibilitando que as atualizações sejam muito oscilativas em superfícies mais instáveis.

Essa atualização, ocorre da seguinte forma:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

Atualizamos os parâmetros θ , multiplicando a taxa de aprendizado η pelo gradiente do ponto, e dividimos esse valor pela raiz quadrada dos somatórios dos quadrados dos gradientes para cada iteração t . Além disso, $\epsilon \approx 10^{-7}$.

21.7 Adam

Adaptative Moment Estimation (Adam) é outro método de otimização que computa taxas de aprendizado adaptativas para cada parâmetro, assim como AdaGrad e outros métodos de otimização.

Enquanto nos métodos que utilizam *momentum*, nós podemos pensar em uma bola descendo uma curva em direção ao ponto mínimo, Adam se comporta como se fosse uma bolsa pesada com atrito descendo essa superfície. Essa forma de minimização faz com que o erro de minimização seja o menor possível, ao mesmo tempo que o tempo de convergência seja pequeno.

Assim como os otimizadores que utilizam *momentum*, vistos nas seções anteriores, Adam utiliza uma média exponencial dos gradientes calculados anteriormente para definir a taxa de aprendizado. Com isso, a atualização dos parâmetros se dá de forma muito similar aos otimizadores que utilizam *momentum*.

A intuição da implementação do Adam é dividida em três etapas: *momentum*, correção de *bias* e AdaGrad.

Na primeira etapa - *momentum* - computamos as médias vetores *momentum* passados m_t e, na terceira etapa, os quadrados dos gradientes passados v_t , respectivamente, como segue:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta} J(\theta)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \nabla_{\theta} J(\theta)^2$$

Onde m_t e v_t são, respectivamente, o *momentum* acumulado e o gradiente acumulado até um período t .

Como, m_t e v_t são previamente, inicializados com 0's, foi observado que essas atualizações, principalmente nas etapas iniciais, tendem à zero e, especialmente quando a taxa de decaimento é baixa (ou seja, quando β_1 e β_2 estão próximos de 1).

Para isso, na segunda etapa neutralizamos essas tendências não previstas através de *bias-correction*. Para isso, calculamos o valor de \hat{m}_t para atualizar a direção de cada um dos parâmetros e, através dos valores dos gradientes de \hat{v}_t , atualizamos efetivamente a otimização dos parâmetros. Assim, teremos:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Agora, só basta calcular as atualizações dos parâmetros θ utilizando, por exemplo AdaGrad.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Com isso, a otimização funcionaria corretamente com os parâmetros propostos para $\beta_1 = 0.9$ e $\beta_2 = 0.999$ e taxa de aprendizado $\eta = 10^{-3}$ ou 5×10^{-4}

Na figura abaixo, podemos perceber as diferenças entre os modelos de otimizações discutidos. Adam combina cada um dos modelos vistos anteriores a fim de otimizar a função custo.

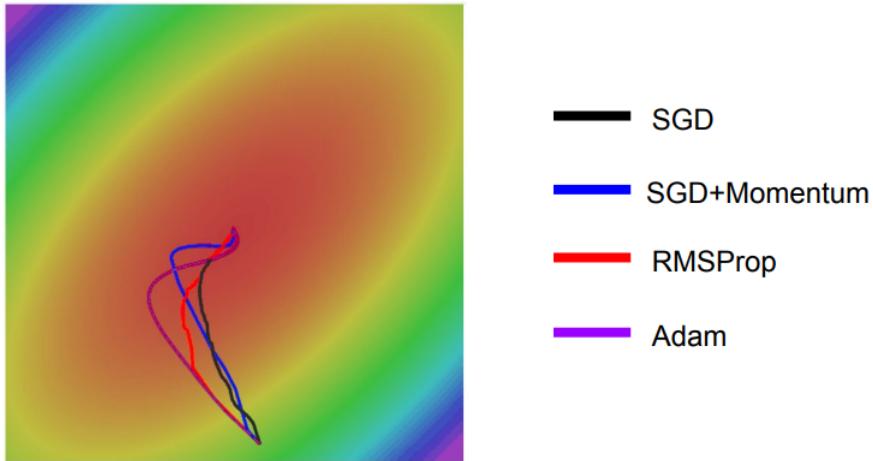


Figura 43: Comparação entre os modelos de otimização vistos. Percebe-se que Adam combina os diferentes métodos.

21.8 Mapreduce e Parallelismo de Dados

Muitas vezes, iremos trabalhar com grandes quantidades de dados e para isso, iremos dividir o conjunto de dados processado no algoritmo de otimização em subconjuntos para que possamos processar cada um deles em diferentes máquinas e, assim, treinar o nosso algoritmo em paralelo.

Podemos dividir o nosso conjunto de treino em z subconjuntos, onde z corresponde ao número de máquinas que podemos usar em paralelo. Para cada uma dessas máquinas, calculamos

$$\sum_{i=p}^q (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

Com isso, *Mapreduce* irá utilizar desses dados mapeados a cada máquina, e "reduzir" eles calculando:

Algorithm 12 Algoritmo Mapreduce

```

1: procedure
2:   for  $i = 0$  to  $n$  do
3:      $\theta_j = \theta_j - \alpha \frac{1}{z} (temp_j^{(1)} + temp_j^{(2)} + \dots + temp_j^{(z)})$ 
4:   end for
5: end procedure
```

Onde o valor de $temp_j$ representa a função custo calculada em cada uma das máquinas. *Mapreduce* calcula a média, multiplicando pela taxa de aprendizado α e atualizando os valores de θ .

Parte VI

Tópicos avançados

22 Redes neurais convolucionais (*Convolutional neural networks*)

22.1 Visão geral

Redes neurais convolucionais (CNNs ou ConvNets) são muito similares as redes neurais vistas nas Seções 12 e 13. Cada neurônio recebe *inputs* e executam funções lineares e não lineares de uma extremidade a outra da rede neural através da função custo. CNN é uma classe de rede neural artificial do tipo feed-forward, que vem sendo aplicada no processamento e análise de imagens digitais.

A principal diferença entre NNs e CNNs é que, basicamente, uma CNN pode ser pensada como uma rede neural que possui várias cópias do mesmo neurônio. Esse tipo de implementação nos permite trabalhar com modelos computacionalmente grandes enquanto mantém o número de parâmetros atuais.

22.2 Camadas de uma ConvNet

Como foi descrito acima, uma ConvNet é uma sequência de camadas que transforma um volume de ativações em outro através de uma função de diferenciação. Para isso, utilizamos camadas específicas para realizar o treinamento de uma ConvNet, dentre elas: *Convolutional layer*, *Pooling layer* e *Fully-connected layer*. Esta última, exatamente como uma rede neural regular.

A principal diferença entre uma *dense layer* e uma *convolutional layer* é que as camadas densas detectam padrões globais, enquanto camadas convolucionais detectam padrões locais.

Para a melhor compreensão será utilizado um exemplo CIFAR-10 [1] o qual consiste em uma base de dados com 60 mil imagens de tamanho 32x32 separadas em 10 classes. O conjunto de dados é separado em 50 mil para usado para treino e 10 mil para teste.

Uma ConvNet para classificar as imagens de CIFAR-10 tem a seguinte estrutura, detalhada abaixo:

- *Input* [32x32x3]: recebe um vetor dos valores dos *pixels* da imagem. Neste caso a imagem tem largura 32, altura 32 com três canais de cores (RGB);

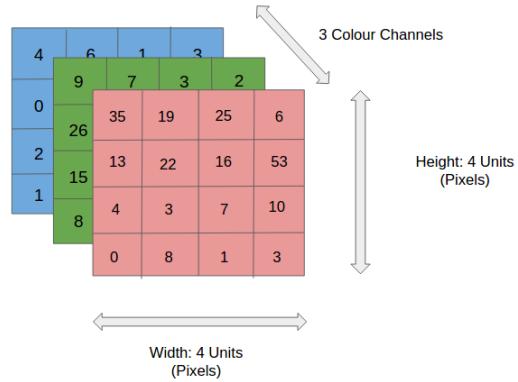


Figura 44: Representação de uma camada de *input* de uma rede neural convolucional. Podemos perceber, que na imagem representada temos uma imagem de tamanho 4×4 com três canais de cores RGB e cada valor de cada pixel corresponde a um valor em cada uma dessas escalas de cores.

- *Conv Layer*: computa a saída dos neurônios que estão conectados a regiões de entrada. Cada um computa um produto escalar entre os pesos e a uma região as quais eles estão conectados no volume de entrada. Isso pode resultar em um volume como $[32 \times 32 \times 12]$ se desejarmos utilizar 12 filtros *kernel*. Este filtro possui um tamanho que varre a imagem através de *strides* - distância entre duas varreduras do filtro - gerando uma nova imagem de mesmo tamanho, porém com esse filtro aplicado;

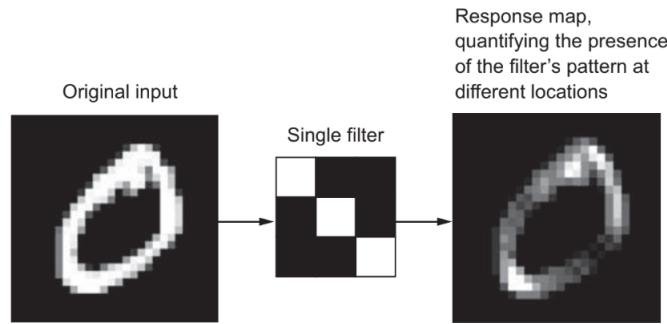


Figura 45: Representação da utilização de um filtro (*kernel*) para mapear os valores da imagem dada como entrada. Percebe-se que o filtro utilizado mantém as bordas da imagem de entrada.

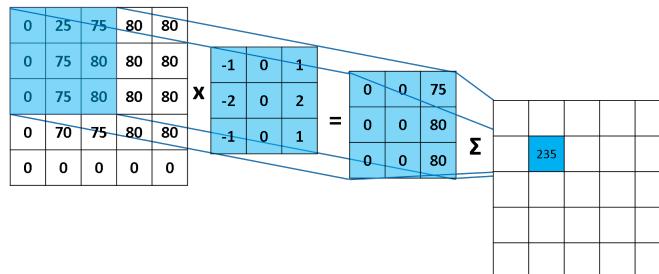


Figura 46: Representação das operações realizadas por um filtro a partir de uma imagem. O filtro (3×3) multiplica os valores da imagem (5×5) pelos seus respectivos valores até gerar uma saída correspondente de tamanho 3×3 . Esses valores retornados são somados e adicionados ao respectivo pixel da imagem (5×5) camada seguinte com o filtro aplicado.

- *ReLU Layer*: computa a função de ativação;

- *Pool Layer*: computa uma operação de redução da resolução ao longo das dimensões, resultando em um volume como [16x16x12];

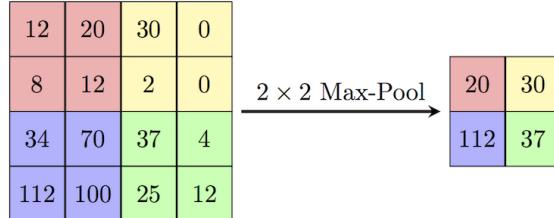


Figura 47: Representação de uma operação de *pooling*. Na figura, utiliza-se o método de *max-pooling* com um filtro de tamanho 2x2 o qual escolhe o maior valor do pixel que se encontra dentro do filtro.

- *FC Layer*: computa as pontuações das classes, resultando em um volume de tamanho [1x1x10], onde cada um dos 10 números representa uma categoria da base de dados CIFAR-10

Abaixo, na Figura 48 está representado, esquematicamente uma ConvNet voltada para a base de dados CIFAR-10. Nela, percebemos as operações que foram realizar ao longo do processamento de convolução e de aprendizado da classificação.

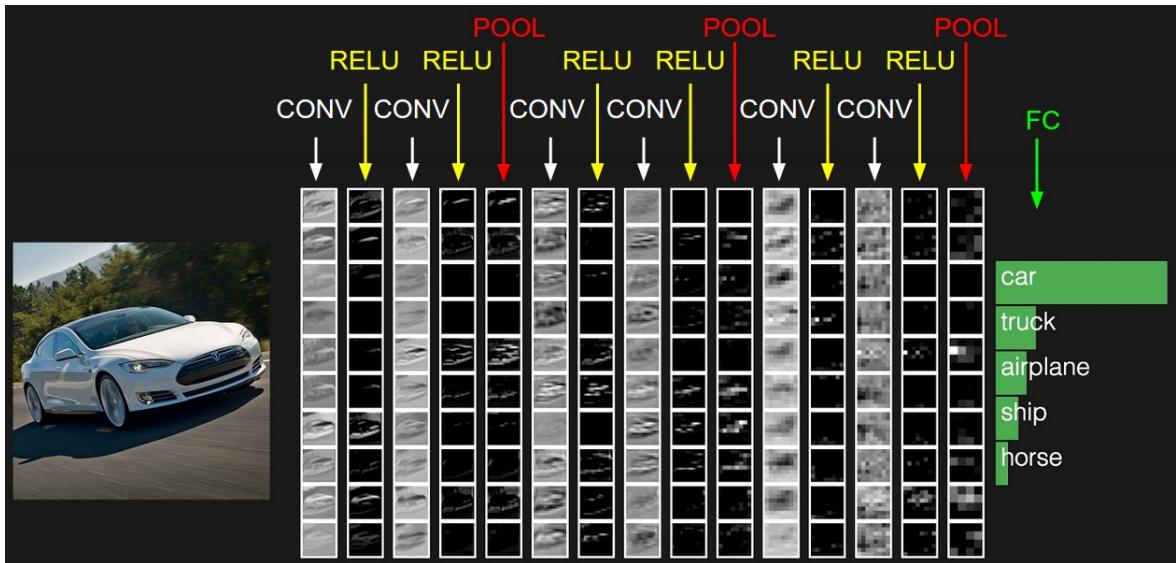
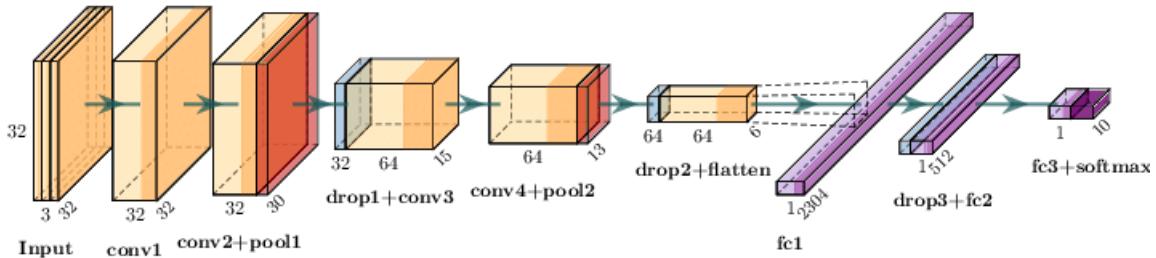


Figura 48: Representação de uma ConvNet para a base de dados CIFAR-10

Além disso, uma camada de convolução de uma ConvNet recebe alguns outros parâmetros que serão descritos a seguir:

- Aceita um volume de tamanho: $W_1 \times H_1 \times D_1$;

- Requer cinco hiperparâmetros:
 1. Número de filtros (*kernels*): K
 2. Tamanho do filtro: F
 3. *Stride* (distância entre duas posições consecutivas do filtro): S
 4. Tamanho da borda (*zero-padding*): P
- Produz um volume de: $W_2 \times H_2 \times D_2$ onde:
 - $W_2 = \frac{(W_1 - F + 2P)}{S+1}$
 - $H_2 = \frac{(H_1 - F + 2P)}{S+1}$
 - $D_2 = K$

Geralmente, uma inicialização comum para esses parâmetros é $F = 3$, $S = 1$ e $P = 1$, porém esses parâmetros podem ser modificados de acordo com a intenção de treino da ConvNet.

Da mesma forma, uma camada de *pooling* de uma ConvNet recebe alguns parâmetros a seguir descritos:

- Aceita um volume de tamanho: $W_1 \times H_1 \times D_1$;
- Requer dois hiperparâmetros:
 1. Tamanho do filtro: F
 2. *Stride*: S
- Produz um volume de: $W_2 \times H_2 \times D_2$ onde:
 - $W_2 = \frac{(W_1 - F)}{S+1}$
 - $H_2 = \frac{(H_1 - F)}{S+1}$
 - $D_2 = D_1$

Assim, percebemos que uma ConvNet é simplesmente uma lista de camadas que transformam um volume de entrada (neste caso uma imagem) e um outro volume de saída (neste caso os valores das classes).

22.3 Técnicas de otimização de treino

Em muitos casos, o conjunto de dados que desejamos utilizar para treinar o nosso modelo não é grande o suficiente para potencializar a acurácia do modelo para casos de teste. Para isso, nesta seção serão apresentadas algumas técnicas de manipulação de dados que possam ser úteis para otimização do treino utilizando milhares de imagens.

22.3.1 Data Augmentation

Essa técnica é utilizada para aumentar o tamanho do conjunto de dados e evitar casos de *overfitting*. Essa técnica realiza transformações nas imagens que serão usadas para serem treinadas a fim de

potencializar a generalização do modelo. Essas transformações podem ser compressões, rotações, alongamentos e algumas mudanças de cor.

22.3.2 Modelos pré-treinados

Nesta seção, iremos discutir sobre modelos de ConvNets previamente treinados a fim de aumentarmos a acurácia do nosso modelo. Em outras palavras, iremos utilizar um modelo que foi treinado com milhões de imagens que estão relacionadas com o nosso modelo. Isso nos garante em ter camadas de convolução extremamente eficazes.

Dessa forma, podemos treinar o nosso modelo com um conjunto de dados pequeno, pois a camada de convolução já foi treinada anteriormente com um conjunto de dados muito maior, podendo, portanto aumentar a confiabilidade dos resultados de teste do nosso modelo.

22.3.3 Fine tuning

Após utilizar um modelo pré-treinado, devemos ajustar as camadas finais para se adaptarem com o nosso modelo. Fazemos isso, pois as camadas iniciais conseguem classificar muito bem dados de baixo nível, como bordas, linhas e contornos em uma imagem. Assim, se ajustarmos as camadas finais, podemos procurar apenas recursos relevantes para o nosso problema que está relacionado ao nosso conjunto de treino.

22.4 Detecção e Segmentação

Detecção de objetivos é um problema multi-tarefa que usa algoritmos de classificação e localização a fim de saber qual objeto é e qual a sua localização. Na Figura 49 temos uma representação de uma imagem que os objetos presentes foram classificados e localizados.

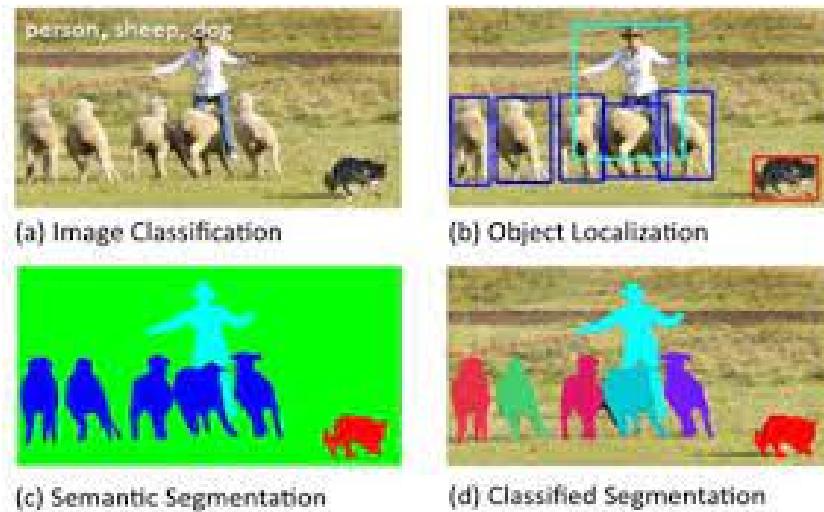


Figura 49: Representação de uma imagem que seus objetos foram devidamente classificados e localizados. Em (a) está representada a classificação da imagem, em (b) a localização dos objetos, em (c) a segmentação semântica da imagem e em (d) a segmentação classificada dos objetos e suas determinadas localizações.

Este tipo de estudo é muito recorrente na área de visão computacional, principalmente usado para a detecção de rostos através de imagens e em carros autônomos através de imagens e vídeos.

22.4.1 Segmentação Semântica

Dada uma imagem, desejamos classificar cada pixel dessa imagem em diferentes categorias. Podemos verificar na Figura 50 que dadas imagens de entradas, classificamos cada pixel em cinco diferentes categorias: céu, árvores, grama, gato ou vaca.

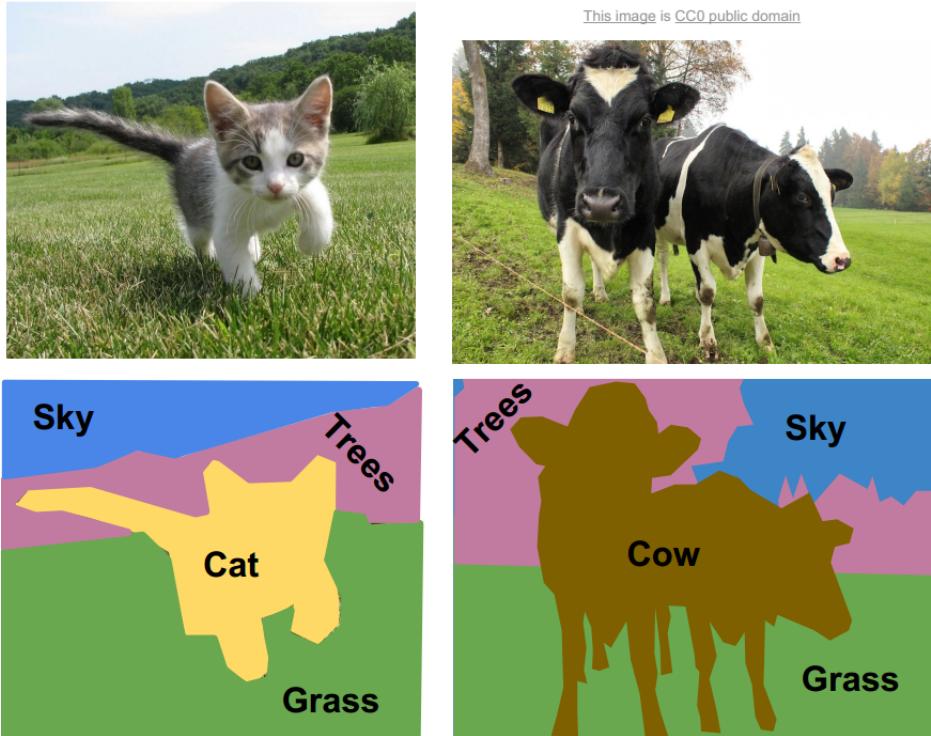


Figura 50: Representação de segmentação semântica. Na coluna da esquerda, percebemos que classificamos a imagem de um gato em quatro categorias representadas pelas cores azul, rosa, verde e amarelo. Na coluna da direita classificamos a imagem em quatro categorias representadas pelas cores azul, rosa, verde e marrom.

Para a implementação desse método, utilizamos de ConvNets que trabalham de forma muito específica: diminuindo e aumentando a resolução da imagem usando *pooling* e *upsampling* ou *unpooling*.

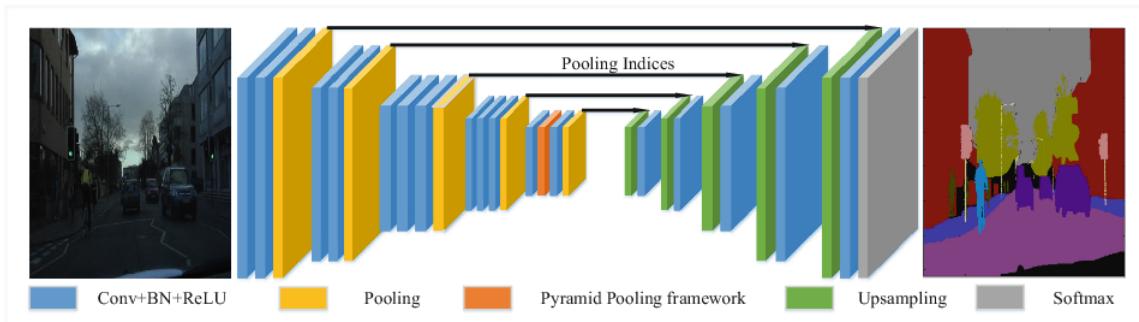


Figura 51: Representação de uma ConvNet de segmentação semântica. Percebemos que existem dois tipos de alteração da resolução da imagem: usando *pooling*, que diminui a resolução da imagem e *upsampling* que aumenta a resolução da imagem.

Com essa implementação, podemos promover o aprendizado da rede neural centrado em cada pixel da imagem e, assim, classificar cada pixel em diferentes classes.

22.4.2 Localização

Dada uma imagem, desejamos localizar onde está determinado objeto na imagem. Para isso, temos dois tipos de localização: localização com apenas um objeto na imagem e localização com mais de um objeto.

Podemos verificar como funciona a localização de objetos em imagens na Figura 52

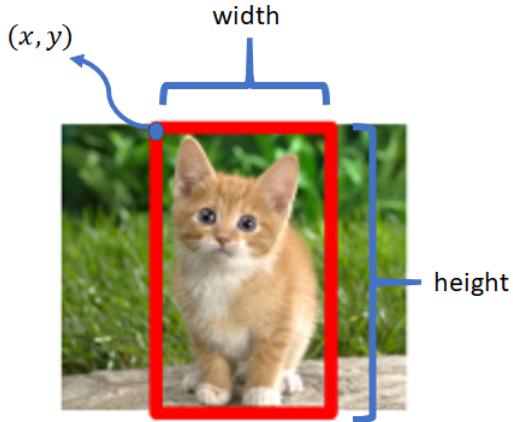


Figura 52: Representação de uma localização de objetos. Percebemos que a partir de uma coordenada (x, y) prevemos os valores da altura e da largura da *bounding box* que limita a figura do gato, neste caso.

Para o problema de localização de objetos, trataremos esse problema como um problema de regressão, que dado uma coordenada (x, y) de uma *bounding box*, desejamos prever as outras coordenadas dessa *bounding box* de acordo com a classificação gerada pela ConvNet. Assim, o resultado da função de localização será uma tupla do tipo (x, y, w, h) onde x e y são as coordenadas iniciais da *bounding box* e w e h são, respectivamente a largura e a altura da *bounding box*.

22.4.3 Detecção de Objetos

Dada uma imagem que contém múltiplos objetos, desejamos classificar e localizar cada um desses objetos presentes na imagem.

Podemos verificar na Figura 53 como funciona a detecção de objetos.

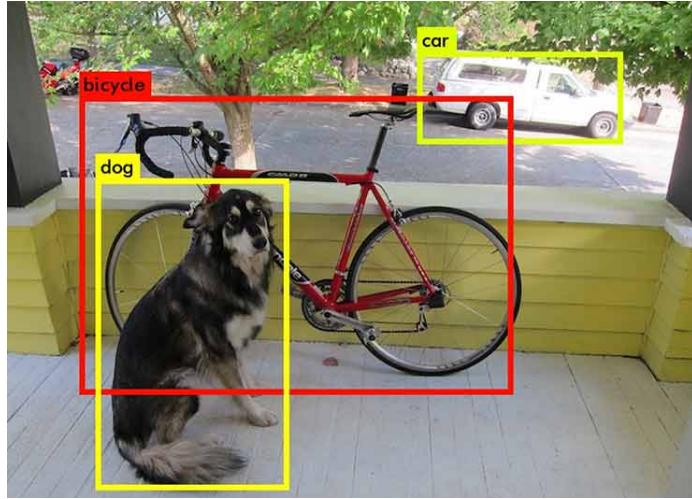


Figura 53: Representação de detecção de objetos. Percebe-se que, para cada um dos objetos na imagem, existe uma *bounding box* localizando cada um dos objetos e para cada uma dessas *bouding boxes*, sua classificação.

Para isso, foram criados dois métodos principais de detecção de objetos em imagens *You Only Look Once* (YOLO) e *Single Shot Detector* (SSD).

Para YOLO, o método é essencialmente regressão, o qual usa uma imagem para aprender as possibilidades das classes de acordo com as coordenadas das *bounding boxes*. YOLO divide cada imagem em uma rede $S \times S$ e cada rede prevê N *bounding boxes* e a confiança da precisão se cada *bounding box* realmente está contornando um objeto (Figura 54). Então, são previstas um total de $S \times S \times N$ *bounding boxes* que, em sua maioria têm pontuações de confiança baixas e, portanto, podemos nos livrar delas.

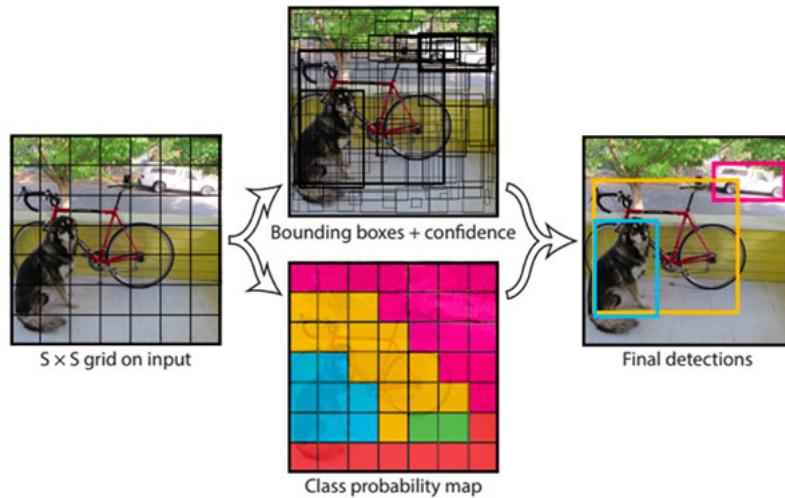


Figura 54: Representação de um algoritmo de detecção de objetos usando o método YOLO. A partir de uma imagem de entrada, a dividimos em uma rede de tamanho $S \times S$ e, através de um algoritmo de regressão, encontramos as coordenadas das *bounding boxes* e a confiança de que cada uma delas possui um objeto.

Por outro lado, SSD atinge um melhor equilíbrio entre rapidez e precisão. O SSD executa uma ConvNet na imagem de entrada apenas uma vez e calcula um mapa de recursos (*feature map*). Agora, executamos a convolução com um *kernel* 3x3 para prever as *bounding boxes* e as probabilidades de categorização.

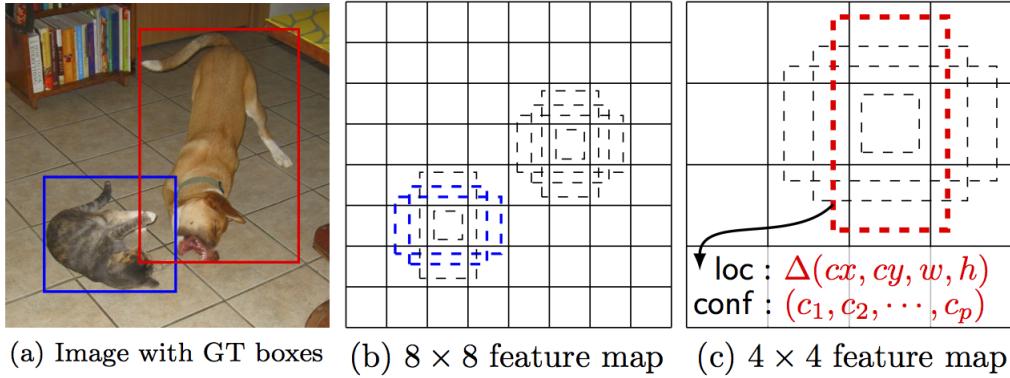


Figura 55: Representação de um algoritmo de detecção de imagens usando SSD. A partir de uma imagem de entrada, através de uma ConvNet prevemos as coordenadas das *bounding boxes* com suas respectivas probabilidades, através de apenas uma passagem pela rede para cada objeto.

22.4.4 Segmentação de Instâncias

Assim, como detecção de objetos, desejamos localizar e classificar objetos em uma dada imagem, porém, além de apenas definir uma *bounding box* para cada um dos objetos, desejamos segmentar a imagem de acordo com cada objeto encontrado e classificá-los, como pode-se perceber na Figura 56

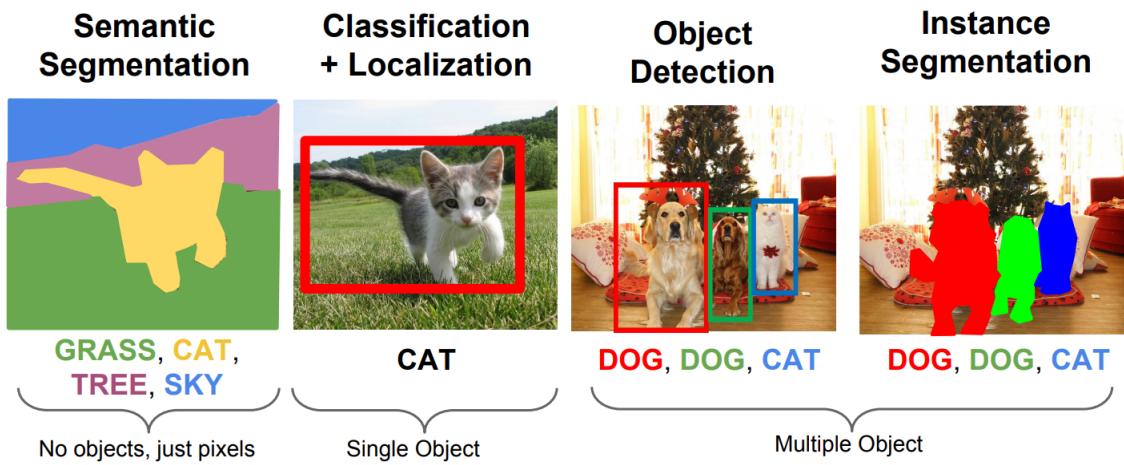


Figura 56: Comparação entre os diferentes tipos de detecção e segmentação de objetos em imagens.

23 Redes neurais recorrentes (*Recurrent neural networks*)

23.1 Visão geral

Redes neurais recorrentes, ou RNNs, são um tipo de rede neurais que usam dados sequenciais ou dados de séries temporais. Os algoritmos mais conhecidos utilizados estão relacionados a tradução de linguagens, processamento de linguagem natural (*Natural Language Processing (NLP)*), reconhecimento de discursos e legenda de imagens.

Assim como *feedforward* e redes neurais convolucionais, RNNs utilizam de dados de treino para aprender, ou seja, é um tipo de aprendizagem supervisionada. A principal diferença de uma RNN é que ela possui uma espécie de memória que guarda as informações prévias de *input* que influenciam nas próximas camadas da rede neural (Figura 57). Além disso, RNNs dependem dos elementos posteriores dentro da sequência.

Recurrent Neural Network structure

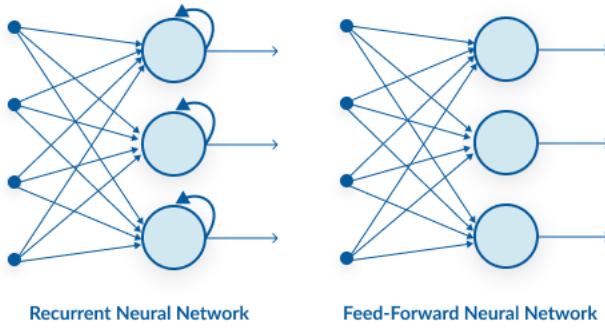


Figura 57: Representação da comparação estrutural de umas rede neural recorrente e uma rede neural tradicional do tipo *feedforward*.

Para representação de uma RNN podemos utilizar dois métodos: *rolled* e *unrolled* mostradas na Figura 58.

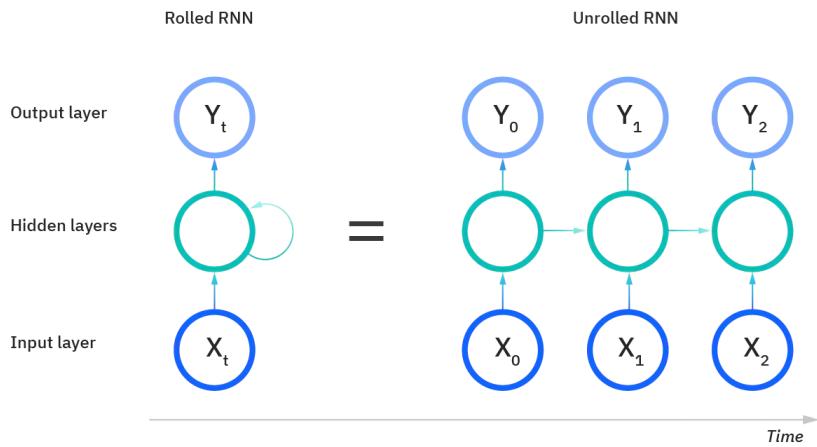


Figura 58: Duas formas básicas de representação de uma RNN. A forma *rolled* representa a rede neural inteira, focando apenas da saída. Na forma *unrolled* representa as camadas individuais com seus respectivos parâmetros.

Outra importante diferença em uma RNN em relação a uma rede neural do tipo *feedforward* é que no caso das RNNs, seus nodos compartilham dos mesmos parâmetros ao logo de cada camada da rede, enquanto em uma rede *feedforward* possui diferentes parâmetros (pesos) em cada nodo.

23.2 Vetores de palavras (*word embeddings*)

Como foi mencionado, uma das principais utilizações de RNNs é para o processamento de linguagem natural. Contudo, uma rede neural não consegue distinguir diferenças semânticas entre frases e impactos de determinadas palavras apenas pelo contexto. Sabemos que uma rede neural trabalha com valores numéricos e, para isso, devemos codificar essas palavras e frases de forma que o nosso modelo possa reconhecer e processar essas diferenças entre elas.

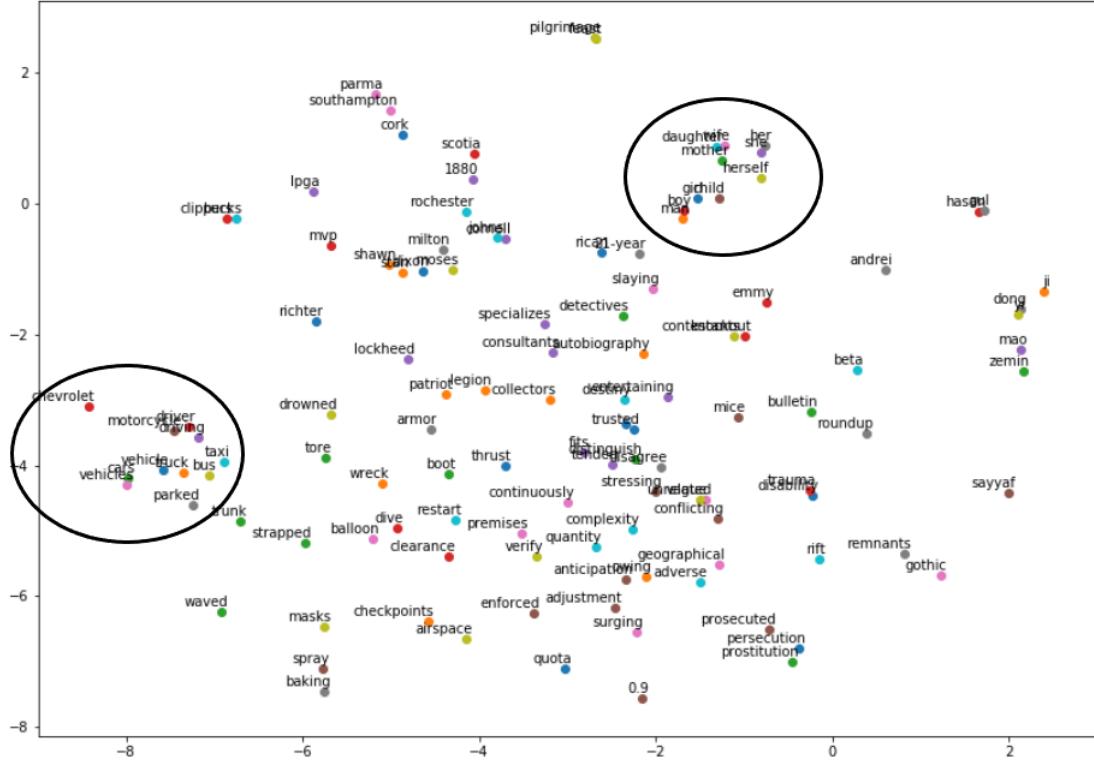


Figura 59: Representação de um espaço vetorial formado pelas *word embeddings*. Podemos verificar as similaridades das palavras dentro de um mesmo círculo e a diferenças entre elas quando não pertencem ao mesmo grupo.

Dessa forma, palavras podem ser diferenciadas em um espaço vetorial de acordo com o ângulo que geram entre elas. Por exemplo, palavras com significados opostos tendem a gerar um ângulo maior, enquanto palavras com significados similares geram um ângulo menor. Podemos verificar isso, na Figura 60, a seguir.

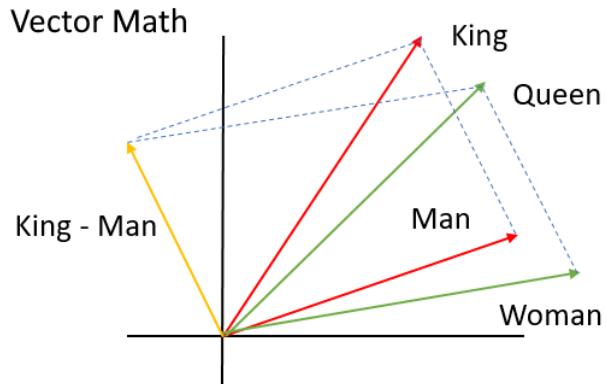


Figura 60: Representação da diferenciação de palavras através de vetores. Percebe-se que palavras com significado similar, possuem um ângulo menor, como por exemplo, as palavras 'King' e 'Queen', enquanto palavras com significados distintos como 'Woman' e 'King' possuem um ângulo maior.

Nesta seção, serão apresentadas formas de pré-processamento de dados em forma de texto a fim de alimentar a nossa rede neural. Para convertermos esse texto para valores numéricos, utilizaremos um métodos chamados *one-hot encoding* e *word vectors*.

23.2.1 One-hot encoding

One-hot encoding representa cada palavra como um vetor de tamanho V , onde V é o número de palavras únicas no nosso conjunto de dados. Cada palavra é codificada em um vetor binário no qual o valor 1 é um índice único para cada palavra e o valor 0 para o restante das palavras. Para visualizar isso melhor, podemos pensar nas duas frases: "Redes Neurais são difíceis" e "Redes Neurais são fáceis", como estão exemplificadas na Figura 61.

Redes	Neurais	são	difíceis	fáceis
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

Figura 61: Exemplo da codificação *one-hot* em um vocabulário de cinco palavras, ou seja,
 $V = 5 \rightarrow ['\text{Redes}', '\text{Neurais}', '\text{são}', '\text{difíceis}', '\text{fáceis}']$.

Essa codificação mapeia cada palavra a um vetor único em que cada posição representa uma palavra distinta do vocabulário. Esse método converte qualquer palavra em valores numéricos de uma maneira muito simples.

Todavia, com essa codificação temos dois problemas principais: uso de memória inutilizada e perda semântica da frase. Primeiramente, teremos um vetor codificado para cada uma das palavras que aumenta o tamanho de acordo com a complexidade do vocabulário, então teremos um vetor enorme, composto basicamente por zeros, para representar uma só palavra o que pode levar a um uso excessivo e desnecessário de memória. Além disso, esse tipo de codificação não guarda o significado semântico de cada palavra. Para isso, gostaríamos de representar um vetor que representasse o valor semântico de cada palavra e similaridade entre elas, por exemplo, as palavras "fáceis" e "difíceis", gostaríamos que fossem dois vetores completamente distintos, enquanto "errado" e "incorreto" gostaríamos que fossem vetores semelhantes.

Para isso, utilizamos de outro método de codificação chamado *word2Vec* que resolve os dois problemas citados acima.

23.2.2 Word2Vec

Em *word2Vec*, existem dois tipos de arquiteturas: *Continuous Bag Of Words* (CBOW) e *Skip Gram*. Primeiro iremos discutir a arquitetura *Skip Gram* para depois discutirmos CBOW.

Iremos fazer com que os dados nos digam quais palavras estão ocorrendo próximo de uma outra palavra. Usamos um método chamado "janela de contexto" para nos dizer isso.

Se utilizarmos como exemplo a seguinte frase "Deep Learning is very hard and fun". Primeiro devemos definir o tamanho da janela (*window size*), que pode ser, por exemplo igual a 2. Precisamos iterar sobre todas as palavras presentes nos nossos dados - nesse caso é apenas uma frase. Como o tamanho

da janela é dois iremos considerar duas palavras antes e duas palavras depois da palavra que estamos analisando, como percebe-se na Figura 62. E iremos repetir isso até que todas as palavras sejam coletadas em forma de pares.

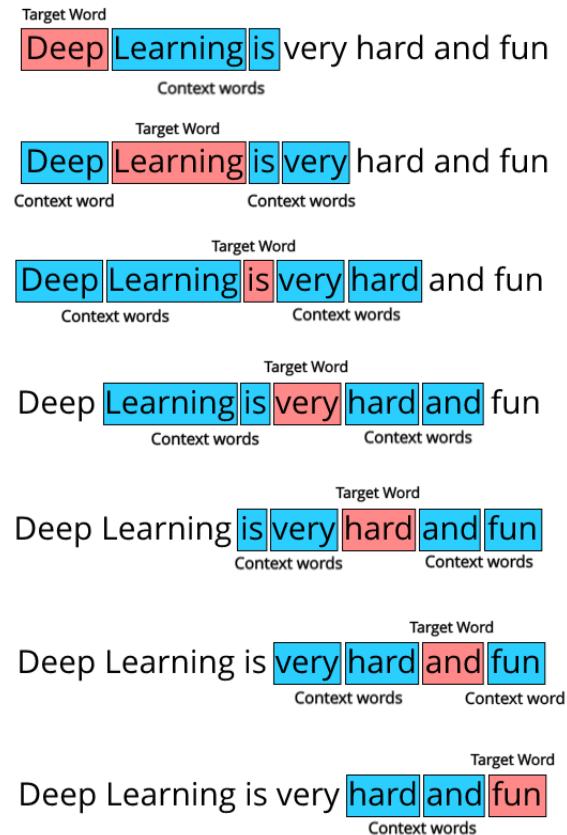


Figura 62: Representação da formação de uma "janela de contexto".

Com isso feito, podemos formar pares entre as palavras dos dados verificados de forma que a palavra que estamos verificando esteja relacionada com uma palavra que pertence ao contexto dela, ou seja, buscaremos pares ordenados do tipo (target word, context word). Para a frase do exemplo teremos os seguintes pares.

- | | |
|--|--|
| 1. (Deep, Learning), (Deep, is) | (very, hard), (very, and) |
| 2. (Learning, Deep), (Learning, is),
(Learning, very) | 5. (hard, is), (hard, very),
(hard, and), (hard, fun) |
| 3. (is, Deep), (is, Learning),
(is, very), (is, hard) | 6. (and, very), (and, hard),
(and, fun) |
| 4. (very, learning), (very, is), | 7. (fun, hard), (fun, and) |

Esses dados podem ser considerados os nossos dados de treino para *word2Vec*.

O modelo *Skip Gram* tenta prever o contexto de cada palavra dada uma palavra que irá ser focada. Usamos uma rede neural para a previsão dessa tarefa. A entrada da rede neural será a versão

codificada *one-hot* das palavras, onde o tamanho V do vetor será o tamanho do vocabulário. A arquitetura da rede está exemplificada na Figura 63, abaixo.

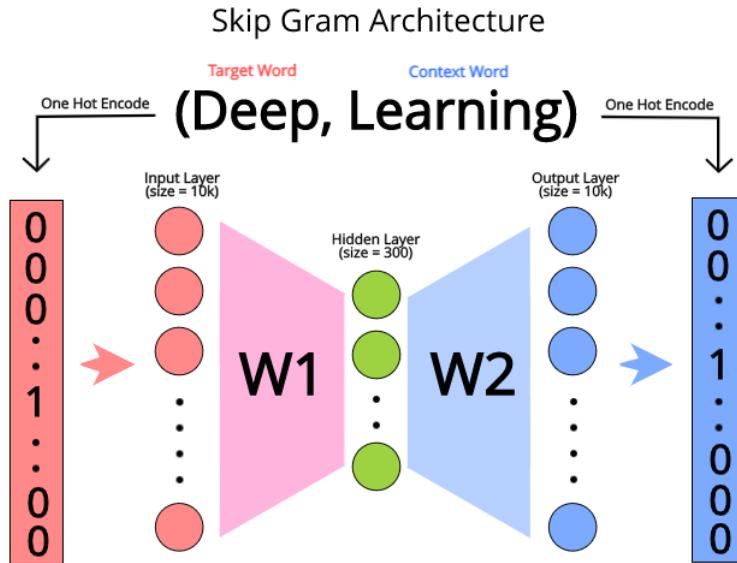


Figura 63: Representação de uma rede neural para a codificação Skip Gram a partir da entrada (Deep, Learning), onde "Deep" é a *target word* e "Learning" é a *context word*. A partir da *target word*, treinamos a rede neural para prever a *context word*, sendo que a saída e a entrada da rede estão na forma *one-hot encoding*.

Para CBOW, a única diferença é que nós tentamos prever a *target word* dada a *context word*, então, basicamente, invertemos o modelo *Skip Gram* para gerarmos o modelo CBOW, como está representado na Figura 64.

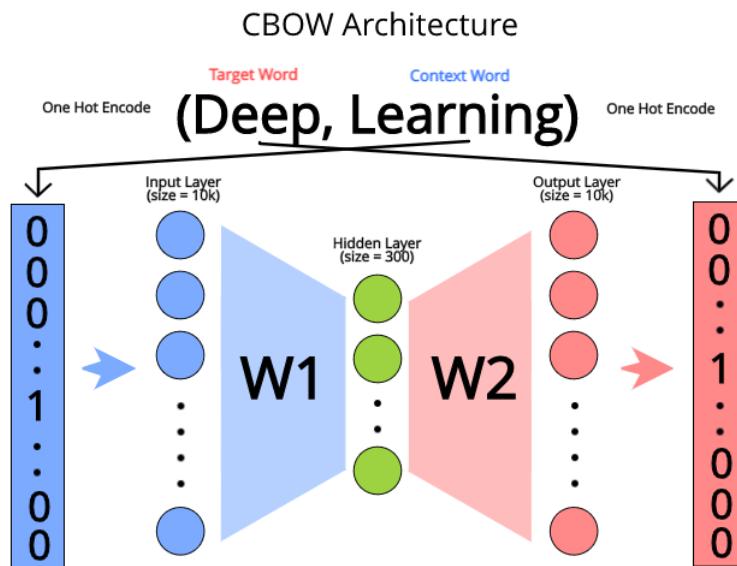


Figura 64: Representação de uma rede neural para a codificação CBOW a partir da entrada (Deep, Learning), onde "Deep" é a *target word* e "Learning" é a *context word*. A partir da *context word*, treinamos a rede neural para prever a *target word*, sendo que a saída e a entrada da rede estão na forma *one-hot encoding*.

Em geral, CBOW é mais rápido e tem melhor representações para palavras mais frequentes. Então, com isso podemos gerar uma identificação semântica de cada palavra e seus relacionamentos umas com as outras.

23.3 Arquitetura de uma RNN

Como foi explicado na Seção 23.1, uma RNN é uma classe de redes neurais que permite que as saídas anteriores sejam usadas como entradas, embora tenham estados ocultos. Com isso podemos determinar uma arquitetura básica desse tipo de rede neural e definir os seus parâmetros e suas funções de ativação que podem ser, potencialmente, utilizadas.

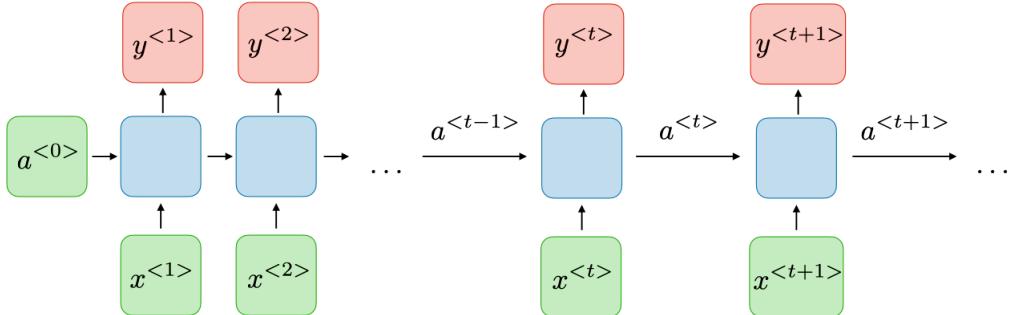


Figura 65

Na Figura 65 está representada uma estrutura básica de uma RNN. Nela, para cada pedaço de tempo t , a ativação $a^{<t>}$ e a saída $y^{<y>}$ são expressos da seguinte forma:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

Onde W_{ax} , W_{aa} , W_{ya} , b_a , b_y são coeficientes que são temporariamente compartilhados e g_1 e g_2 são as funções de ativação.

Os prós e contras de uma RNN estão descritos abaixo:

Tabela 5: Comparação entre os prós e contras de uma RNN

Prós	Contras
Possibilidade de processamento de entrada de qualquer tamanho	Computação lenta
O tamanho do modelo não aumenta de acordo com o tamanho da entrada	Dificuldade de acessar informações de tempo muito distante
A computação leva em consideração informações históricas	Não consegue levar em consideração nenhuma entrada futura para o atual estado
Os pesos são compartilhados ao longo do tempo	

23.4 Aplicações

Os modelos de RNN são muito utilizados, como foi mencionado, em áreas como processamento de linguagem natural, reconhecimento de discurso e geração de música. As diferentes aplicações serão representadas a seguir com suas respectivas arquiteturas.

23.4.1 One-to-one

O tipo de RNN *one-to-one* é uma rede neural muito simples, na qual $T_x = T_y = 1$. Esse tipo de RNN é muito utilizada em redes neurais tradicionais vistas anteriormente.

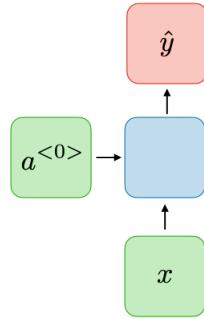


Figura 66: Representação de uma RNN do tipo *one-to-one*.

23.4.2 One-to-many

O tipo de RNN *one-to-many* é uma rede neural, na qual $T_x = 1, T_y > 1$. Esse tipo de RNN é muito utilizada em geração de música.

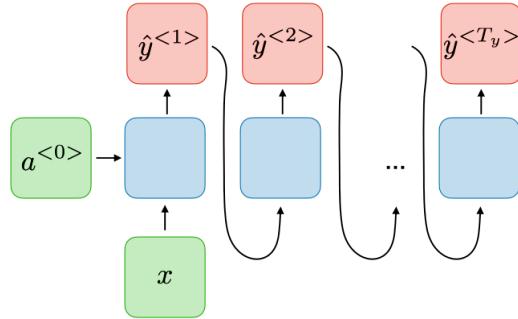


Figura 67: Representação de uma RNN do tipo *one-to-many*.

23.4.3 Many-to-one

O tipo de RNN *many-to-one* é uma rede neural, na qual $T_x > 1, T_y = 1$. Esse tipo de RNN é muito utilizada em classificação de sentimento.

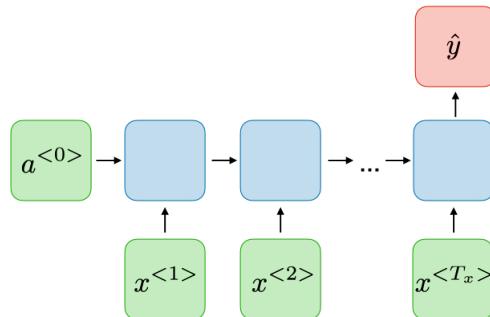


Figura 68: Representação de uma RNN do tipo *many-to-one*.

23.4.4 Many-to-many

O tipo de RNN *many-to-many* é uma rede neural, na qual $T_x = T_y$. Esse tipo de RNN é muito utilizada em nome de reconhecimento de entidade.

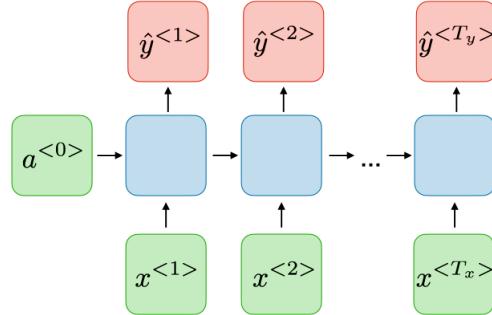


Figura 69: Representação de uma RNN do tipo *many-to-many* com saídas e entradas em números iguais.

A outra forma de representação desse tipo de RNN é na forma $T_x \neq T_y$, também chamada de *sequence-to-sequence*, que é realiza a codificação da entrada usando a arquitetura *many-to-one* e a produção da saída realizada por uma única entrada do tipo *one-to-many*, como se pode perceber na Figura 70. Esse tipo de RNN é muito utilizada em tradução de máquina.

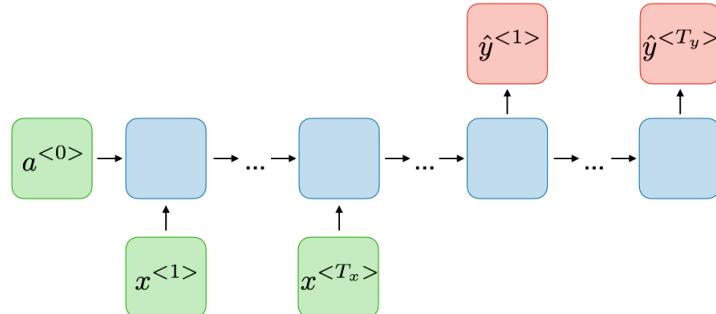


Figura 70: Representação de uma RNN do tipo *many-to-many* com saídas e entradas em números diferentes.

23.5 Função custo (*Cost function*)

Como foi visto nas seções anteriores, a função custo de uma rede neural tem como objetivo calcular o erro entre os valores de entrada a cada passo de tempo. A função custo está representada abaixo.

$$\mathcal{L}_\theta(y, \hat{y}) = - \sum_{t+1}^{T_y} \mathcal{L}_\theta(y^{<t>}, \hat{y}^{<t>})$$

onde $\mathcal{L}_\theta(y, \hat{y})_t$ é a função custo definida. Geralmente essa função custo é a *cross entropy loss*.

23.6 Backpropagation

O algoritmo de *backpropagation* é feito a cada ponto em um período de tempo. No tempo T , da derivada da função custo \mathcal{L} a respeito de uma matriz de pesos W é expressa como segue:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial W} = \sum_{t=1}^T \frac{\partial \mathcal{L}^{(T)}}{\partial W}|_{(t)}$$

Para realizar essa computação, realizamos as atualizações dos parâmetros *feedforward*, a cada passo de tempo, e ao computar a função custo, realizamos a *backpropagation*. Contudo, como esse tipo de rede neural processa os dados sequencialmente de forma a computar cada derivada para cada período de tempo. Esse algoritmo é chamado de *backpropagation through time* [22].

Realizar o cálculo das derivadas de forma sequencial pode ser uma tarefa pouco eficiente. A computação das derivadas parciais são realizadas de forma *end-to-end*, o que, muitas vezes pode causar diversos problemas como desaparecimento e explosão do gradiente que serão explicados em detalhes na Seção 23.8.

23.7 Funções de ativação e propriedades

Como vimos nas seções anteriores, para cada transição de camada, precisamos de uma função de ativação para atualizar os valores dos respectivos nodos da camada seguinte a partir dos nodos da camada anterior. Para RNNs, temos basicamente três tipos de funções de ativação:

1. *Sigmoid*:

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

2. *tanh*:

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

3. *ReLU*:

$$g(z) = \max(0, z)$$

23.8 Gradiente de desaparecimento e explosão

Esse fenômeno ocorre quando o gradiente, no contexto das RNNs, não consegue calcular valores profundos da rede neural devido às sucessivas multiplicações. Muitas vezes essas multiplicações podem ser exponenciais, decrementando ou incrementando, os valores dos parâmetros de acordo com o número de camadas da rede neural.

O principal problema gerado pelo desaparecimento/explosão do gradiente é que a RNN não consegue guardar as informações vistas em camadas muito anteriores da sequência, pois muitos valores estarão zerados ou NaNs.

A fim de evitar o problema de explosão, usa-se a técnica de "recorte do gradiente" (*gradient clipping*) na qual limita o valor do gradiente calculado a fim de evitar explosões que eventualmente podem ocorrer na *backpropagation*. Este método está representado na Figura 71.

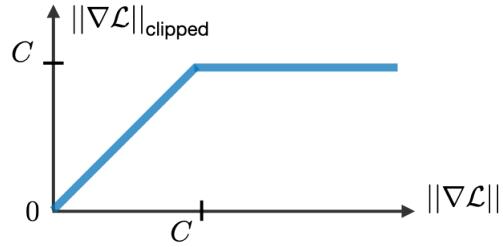


Figura 71: Representação da técnica de "recorte do gradiente". No eixo y está representado o valor do gradiente após a realização do corte de acordo com o crescimento do valor do gradiente, representado no eixo x .

Outra metodologia abordada para evitar o problema de dificuldade de memorização foi a criação de portões de memorização, utilizados em arquiteturas que serão vistas na seção a seguir, como GRUs e LSTMs.

23.9 Long Short-Term Memory (LSTM)

As arquiteturas de LSTMs são utilizadas para resolver os problemas de dificuldade de memorização de longas sequências causadas pelo desaparecimento do gradiente.

A LSTM é uma RNN que memoriza os valores em intervalos arbitrários, sendo, assim, muito adequada para processar e prever séries de valores temporais com intervalos de tempo de duração desconhecida. Como pode-se verificar na Figura 72, a LSTM possui uma estrutura de cadeia que contém quatro redes neurais e diferentes blocos de memória chamados "células".

Para cada período de tempo, a LSTM retorna dois valores: h_t e c_t que são, respectivamente, os valores da hipótese de cada período e os valores da célula de cada período. Ambos valores possuem o mesmo tamanho n . A célula guarda as informações de longo prazo da rede neural, enquanto a hipótese é o valor gerado de determinado período de tempo.

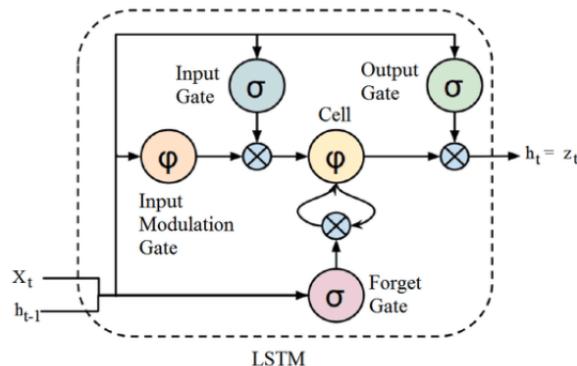


Figura 72: Representação de uma arquitetura LSTM. Percebe-se que existem três *gates* (*input*, *forget* e *output*) e uma célula, onde são realizadas as operações de memorização.

Com isso, uma LSTM consegue ler, apagar e escrever informações da célula. Essas informações são controladas através dos portões (*gates*) que, para cada período de tempo, eles podem estar fechados (0) ou abertos (1). Esses portões são dinâmicos, ou seja, os seus valores são computados de acordo com o contexto da sequência passada como entrada. A seguir, estão listados e definidos cada um dos *gates* representados na Figura 72 acima.

1. *Forget Gate*: as informações que não são mais úteis no estado da célula são removidas através do *forget gate*. Os valores de $x^{<t>}$ e $a^{<t-1>}$ são alimentadas no *gate* e multiplicadas pelas matrizes W , resultando em um valor binário após a passagem pela função de ativação. Caso a saída seja 0, o valor é esquecido e caso seja 1 o valor é mantido para uso futuro.

$$f_t^1 = \sigma(W_{f^1} \cdot [a^{<t-1>} x^{<t>}] + b_{f^1})$$

2. *Input Gate*: as informações úteis para o estado da célula é feita pelo *input gate*. Primeiro, a informação é regulada através da função sigmoide, filtrando os valores, e, depois um vetor de valores criados usando a função tanh retorna valores entre -1 a 1, que contém todos os valores possíveis para $x^{<t>}$ e $a^{<t-1>}$. Em seguida, os valores do vetor e os valores regulados são multiplicados para obter as informações úteis.

$$f_t^1 = \sigma(W_{f^2} \cdot [a^{<t-1>} x^{<t>}] + b_{f^2}) \odot \tanh(W_{f^2} \cdot [a^{<t-1>} x^{<t>}] + b_{f^2})$$

3. *Output Gate*: as informações úteis da célula atual que podem ser usadas nos estados seguintes são extraídas através do *output gate*. Um vetor é, primeiramente, gerado aplicando a função tanh na célula e, depois, essa informação é regulada através da função sigmoide filtrando os valores a serem lembrados. Os valores do vetor e dos valores regulados são multiplicados e são enviados como saída para a célula seguinte.

$$h'_t = \sigma(W_{h'_t} \cdot [a^{<t-1>} x^{<t>}] + b_{h'_t}) \odot \tanh(c_t)$$

No entanto, LSTMs possuem um problema similar aos modelos de RNNs. Quando as sentenças são muito grandes, LSTMs não funcionam muito bem. Isso acontece porque o valor de uma célula muito anterior à célula atual decresce exponencialmente e isso potencializa a perda de informação e a falta de controle do processo. Em geral, RNNs e LSTMs possuem três problemas:

1. A computação sequencial inibe a paralelização. Em outras palavras não podemos paralelizar tarefas em uma RNN ou em uma LSTM devido ao fato de que seu processamento é sequencial;
2. Sem modelagem explícita de dependências de longo e curto alcance;
3. A distância entre as posições é linear.

Contudo, mesmo com a memorização mais eficiente, LSTMs não resolvem efetivamente o problema do desaparecimento do gradiente, justamente por, ainda assim, se tratar de uma sequência. Atualmente, esse problema pode ser resolvido através da utilização de arquiteturas de redes neurais residuais, ou ResNets [8].

23.10 Bidirectional & Multi-layer RNNs

Bidirectional RNNs (BRNNs) são uma arquitetura que utilizam de duas RNNs independentes que processam a mesma sequência, porém, uma delas processa essa sequência no sentido convencional (*forward RNN*) - do início para o fim - e a outra processa no sentido contrário (*backward RNN*) - do final para o início. As saídas dessas redes, no final do processamento, em geral, são concatenadas para cada período de tempo. Na Figura 73 abaixo, está representada uma generalização da arquitetura de uma BRNN.

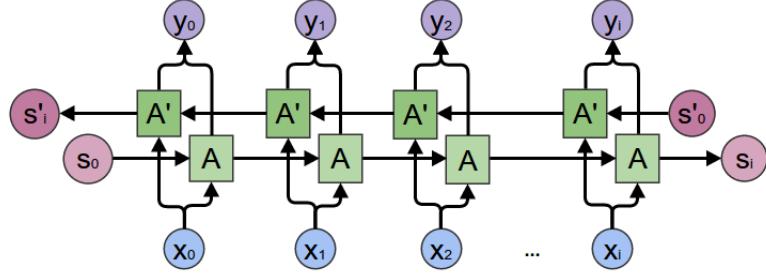


Figura 73: Representação de uma BRNN. Percebe-se que existem duas RNNs processando, independentemente duas informações da mesma sequência de entrada em sentidos opostos.

BRNNs são muito poderosas para classificações de sentimento, pois conseguem compreender o contexto das frases extremamente bem.

Multi-layer RNNs são uma arquitetura que possuem diversas camadas de RNNs empilhadas processando a mesma informação. Esse tipo de arquitetura é muito poderosa, porém o custo computacional para o processamento das informações é muito alto. São frequentemente usadas para problemas de tradução de máquina. A Figura 74 abaixo exemplifica esse tipo de arquitetura.

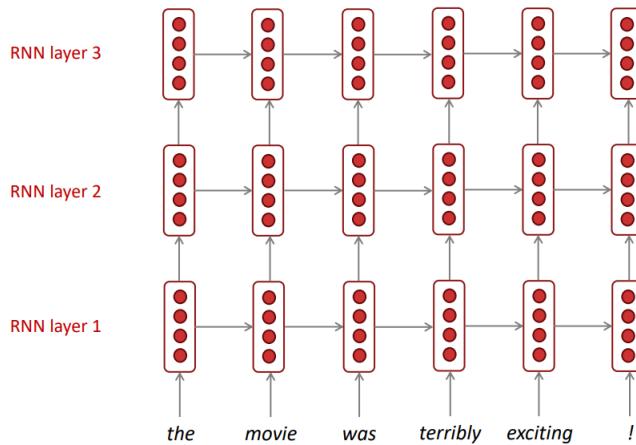


Figura 74: Exemplificação de uma arquitetura *Multi-layer* RNN.

23.11 Attention

Em problemas do tipo *seq2seq*, quando criamos um modelo de tradução de máquina que tem como objetivo traduzir uma sentença geramos um problema chamado *bottleneck* (Figura 75), o qual toda a informação da sentença que desejamos traduzir fica acumulado na camada da última palavra da sentença, devido ao processamento sequencial de uma RNN, possibilitando grande perda de informações passadas e dificultando a tradução.

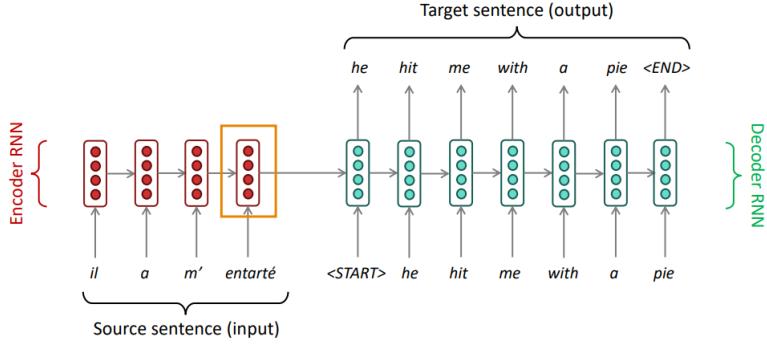


Figura 75: Exemplificação do problema de *bottleneck*, onde toda a informação da sequência que deseja ser traduzida acaba ficando acumulada na camada da última palavra da sentença (retângulo laranja). Esse tipo de problema gera impactos na tradução, principalmente para linguagens que não se relacionam linearmente.

Para resolver o problema mencionado acima, foram criadas técnicas para prestar atenção a palavras específicas e influenciar a tradução a cada período de tempo. Por exemplo, quando traduzimos uma frase, prestamos atenção na palavra que estamos querendo traduzir e validamos a tradução baseando-se nas palavras anteriores e futuras. Uma RNN pode realizar isso, através de uma técnica chamada *attention*.

Em uma RNN, ao invés de codificar toda a sentença em um único estado interno da rede neural, cada palavra corresponde a um estado interno que é passado até o estágio de decodificação, como está representado na Figura 76.

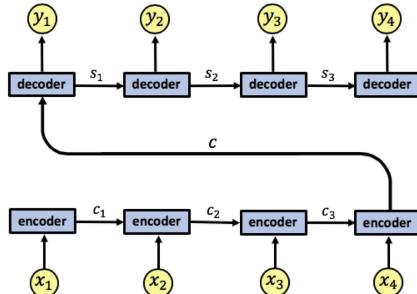


Figura 76: Representação de um processo de *encoding-decoding*. Percebe-se que os valores de x_i são os valores de *input* de cada célula. Esses valores são codificados, gerando os valores c_i , a fim de serem decodificados no final da operação, gerando s_i . Os valores gerados como saída baseado em cada um dos valores decodificados são os valores de y_i .

Contudo, um dos problemas dessa arquitetura é que, como para cada valor codificado a partir dos valores de entrada, devemos gerar um único vetor c , esse processo pode acarretar na perda de informações importantes devido ao *bottleneck*.

Para resolver esse problema, utilizamos o método de *Attention*. Uma RNN utilizando esse método está representada na Figura 77.

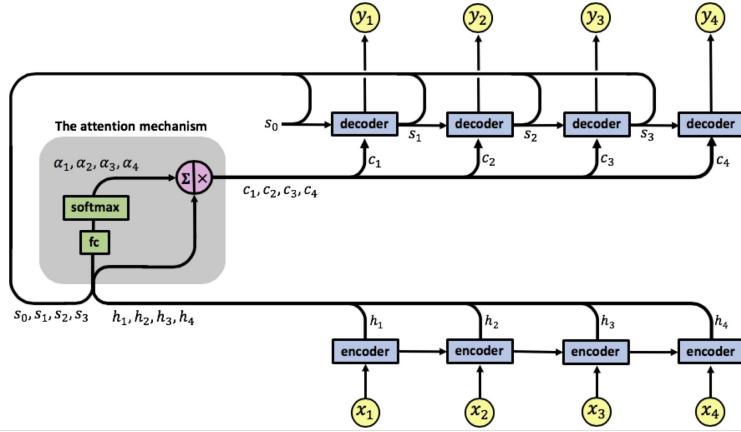


Figura 77: Representação de uma arquitetura de RNN utilizando *attention*. O módulo à esquerda representa o mecanismo de *attention*.

O modelo com *Attention* possui uma única camada de *encoding*, com 4 entradas x_i e com 4 saídas h_i . O mecanismo de *attention* está localizado entre as camadas de *encoding* e *decoding*. As entradas dessa camada são os vetores de saída h_i da camada de *encoding* e os vetores s_i dos estados do *decoder*. E sua saída é uma sequência de vetores chamados vetores de contexto c_i .

Os vetores de contexto possibilitam que o *decoder* foque em determinadas partes da entrada quando está tentando prever a saída. Para o cálculo desses vetores, realizamos uma soma ponderada dos valores h_i gerados pela camada de *encoding* com os pesos a_{ij} gerados a partir do cálculo do grau de relevância da entrada x_i sobre o output y_i no tempo i . Em outras palavras:

$$c_i = \sum_{j=1}^i a_{ij} h_j$$

Os valores de a_{ij} são aprendidos por uma rede neural densa com uma camada de ativação *softmax*.

Para exemplificar o processo de *attention* e como relaciona cada um dos estados nos processos de *encoding-decoding*, podemos pensar na tradução da frase “L'accord sur la zone économique européenne a été signé en août 1992.”, em francês para o inglês, como está representado na Figura 78.

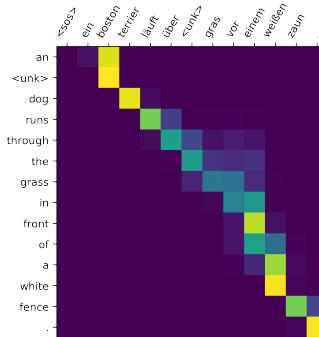


Figura 78: Representação de um processo de tradução da frase “An <unk> dog runs through the grass in front of a white fence” para o alemão. Os pixels mais amarelados representam o quanto de atenção foi prestado a cada etapa da tradução.

Como visto, o método de *attention* aumentou significantemente a performance técnicas de *Neural Machine Translation* (NMT), devido ao fato de possibilitar que o *decoder* foque apenas em determi-

nadas partes da sequência. Além disso, resolve o problema *bottleneck*, pois possibilita que o *decoder* olhe diretamente para a sentença que deve ser traduzidas.

Contudo, apesar da alta significância dessa técnica, ainda não conseguimos resolver o problema de processarmos os dados de entrada em paralelo. Para uma entrada longa, o tempo de processamento dessa técnica é aumentado significativamente.

23.12 Redes neurais convolucionais para NLP

Como foi visto na Seção 22, CNNs conseguem parallelizar atividades devido às camadas de convolução que atuam como cópias de um mesmo neurônio trabalhando em paralelo. CNNs (Seção 22) podem ajudar a resolver esse problema.

No caso das CNNs, quando utilizadas para realização de problemas que envolvem sequências de palavras, por exemplo, as entradas são palavras representadas por vetores de tamanho k , e logo após, um filtro de convolução é aplicado a fim de gerar novos valores a essas palavras. Para cada um dos vetores gerados, escolhe-se o maior valor de cada vetor, a fim de aplicar o *pooling*. Na última camada, aplica-se a função de ativação nos resultados (ver Figura 79).

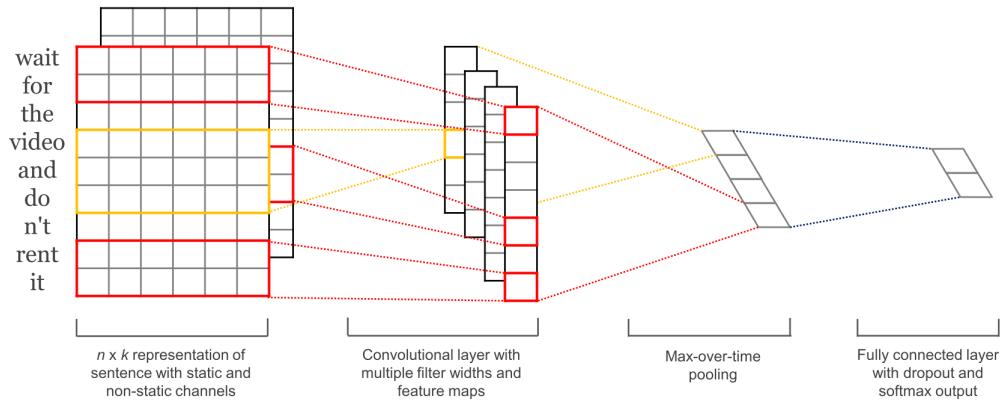


Figura 79: Representação de uma CNN voltada para a classificação de sentimento de uma sentença.

A razão pela qual uma CNN pode funcionar em paralelo é que cada palavra de entrada pode ser processada ao mesmo tempo e não depende necessariamente da palavra anterior já estar traduzida. Além disso, a distância entre o *output* e cada *input* de uma CNN está na ordem de $\log(N)$, o que é muito melhor que uma RNN cuja distância está na ordem linear N .

Com isso, seria interessante se pudéssemos realizar tarefas de forma paralela como as CNNs realizam juntamente com a efetividade da compreensão semântica do mecanismo de *attention*. Com base nisso, uma nova arquitetura foi desenvolvida chamada *transformers* que será detalhada na seção seguinte.

24 Transformers

O artigo publicado em 2017 chamado "Attention is all you need"[21] mudou a forma de como o mecanismo de *attention* era visto. *Transformers* começaram a ser aplicados em tarefas de NLP e

agora estão sendo utilizados em diversas outras áreas de pesquisa como CV.

Como visto nas seções anteriores, arquiteturas de RNNs, como por exemplo LSTMs, processam os dados sequencialmente através das suas diversas *hidden layers*, empilhando as informações. No caso dos *transformers*, as informações são processadas paralelamente, o que acarreta na perda de ordem das sequências.

Um *transformer* possui uma arquitetura similar aos modelos previamente vistos. Ele consiste num conjunto de *encoders* e *decoders*, como podemos visualizar na Figura 80, a seguir.

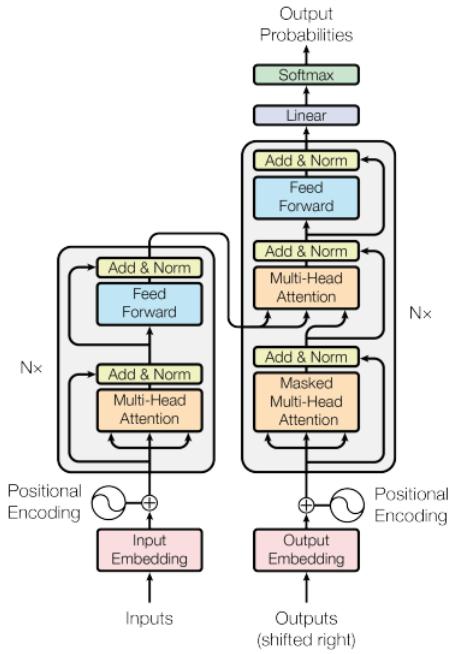


Figura 80: Representação de uma estrutura básica de um *transformer*. Percebe-se que a partir de uma dada entrada, as sequências

A revolução dos *transformers* começou com a ideia de mudar a representação da entrada dos dados a fim de alimentar toda a camada de *input* de uma vez só através de um conjunto em forma de *tokens*, exemplificado na Figura 81. Esse conjunto de entrada pode ser escrito como $X = x_1, x_2, \dots, x_N$, onde $x \in \mathbb{R}^{N \times d_{in}}$ onde os elementos da sequência x_i referem-se aos *tokens*.



Figura 81: Exemplificação do processo de tokenização. Percebe-se que dado um conjunto fixo de uma sequência de palavras, dividimos cada uma dessas palavras em *tokens* a fim de serem processados como um conjunto.

Após a tokenização, projetamos essas palavras em espaço geométrico distribuído, ou seja, como visto na Seção 23.2, *word embeddings*, capturando o sentido semântico das palavras.

24.1 *Positional encoding*

Quando processamos as palavras em forma de *tokens*, o nosso modelo perde a noção de ordem das palavras devido ao fato de processar esse conjunto de uma vez só. Com isso, antes de processarmos as informações das sequências na camada de *self-attention*, precisamos fazer com que o modelo passe a compreender ordem novamente.

Por exemplo, se duas palavras iguais aparecem em lugares diferentes na mesma sentença, provavelmente o sentido semântico delas é diferente. Para esse tipo de problema, o método de *positional encoding* é muito eficiente, pois irá tratar palavras iguais de maneiras diferentes dependendo da ordem em que elas aparecem.

Na Figura 82 abaixo está representada a camada de *positional encoding*.

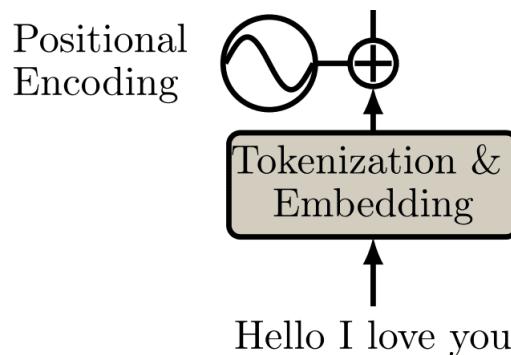


Figura 82: Representação da camada de *positional encoding*. Percebe-se que, dada uma sequência, ela é processada através de tokenização e *embedding* e enviada para uma etapa onde acontece a soma entre uma função sinusoidal e o resultado da camada anterior, gerando uma nova representação semântica para cada uma das palavras.

A função sinusoidal é utilizada para capturar informações relativas às posições das palavras e, com isso, diferenciar semanticamente palavras iguais que possuem sentidos semânticos diferentes. Abaixo, está descrita a equação de *positional encoding* e será discutida a seguir.

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

A equação de *positional encoding* recebe dois argumentos: *pos* e *i*. O primeiro argumento *pos* representa a posição que a palavra que está sendo prestada atenção ocupa. A variável *d* é o tamanho do vetor de *embedding* gerado. O segundo argumento *i* representa a *i*-ésima posição do vetor de *embedding*.

Para o mesmo propósito, pode-se utilizar a equação utilizando cosseno.

$$PE(pos, 2i + 1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

24.2 *Encoder*

24.2.1 *Self-Attention*

Segundo a definição de Ashish Vaswani et al. [21] do Google Brain, "*Self-attention*, às vezes chamado de *intra-attention*, é um mecanismo de atenção relacionado a diferentes posições de uma única sequência que tem como objetivo computar uma representação para a sequência".

Self-attention nos possibilita encontrar correlações entre diferentes palavras de entrada indicando a estrutura contextual e sintática da sentença.

Na prática, o *transformer* possui três representações de *attention* formadas por matrizes denominadas *Queries*, *Keys* e *Values* resultantes da camada de *embedding*. Essa representação é chamada de *Multi-head attention* (Figura 83).

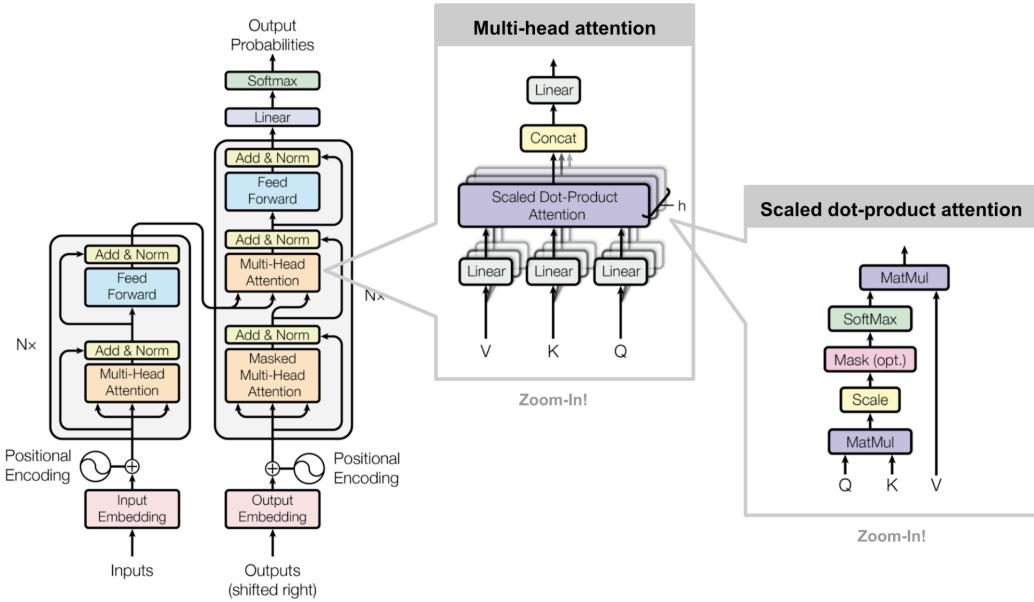


Figura 83: Representação de *multi-head attention* em um *transformer*. Os símbolos Q, K e V representam, respectivamente as matrizes *Queries*, *Keys* e *Values*.

O princípio básico da criação dessa camada é o descobrimento de relações semânticas nas sentenças. Após a camada de *embedding*, os vetores expressam a relação entre cada uma das palavras e, a partir desse contexto, a camada de *multi-head attention* tem como objetivo relacionar a palavra que estamos buscando (Q) com as palavras que possam estar relacionadas com ela (K) e, por fim, influenciar a decisão de acordo com a palavra que gostaríamos que fosse retornada (V).

Essas matrizes, inicialmente, são todas iguais quando passadas como entrada para a *linear layer* (por isso o nome *self-attention*). Essa *linear layer*, tem como objetivo diferenciar cada uma dessas três matrizes Q, K e V multiplicando cada uma delas por três matrizes de pesos da distintos W_Q , W_K , W_V , um para cada uma das *linear layers*.

Para comparar a palavra que estamos buscando Q com os seus possíveis relacionamentos K, realizamos uma operação vetorial que calcula o cosseno do ângulo entre essas palavras. Esse ângulo, como vimos na Seção 23.2 representa o quanto relacionadas semanticamente essas palavras estão. Genericamente, o cálculo do cosseno entre dois vetores se dá por

$$\cos(A, B) = \frac{A \cdot B}{|A||B|}$$

Com isso em mente, a primeira tarefa que devemos realizar é encontrar os relacionamentos entre as matrizes Q e K. Para isso realizamos o produto entre essas matrizes (numerador da equação acima) a fim de gerar um filtro de atenção e dividimos pela raiz quadrada tamanho da sentença $\sqrt{d_k}$ (denominador da equação acima). Por fim, os resultados dessa matriz é passado para uma camada

de ativação *softmax* (Seção 13.1.6) a fim de gerar valores de probabilidade dos relacionamentos entre as palavras. Essas operações irão gerar um filtro de atenção que irá prestar atenção em uma parte específica da sentença baseando-se no relacionamento entre cada uma das palavras.

Para facilitar a compreensão, podemos fazer um paralelo com a atuação de um filtro de atenção sobre uma imagem. O que acontece, neste caso é que ignoramos todos os pixels da imagem original que não interferem no resultado que desejamos alcançar. A Figura 84 abaixo, representa a aplicação do filtro de atenção gerado sobre uma imagem original. O resultado dessa aplicação é uma imagem em que somente o que realmente interessa está aparecendo.



Figura 84: Exemplificação da aplicação de um filtro de atenção sobre os valores dos pixels de uma imagem.

Nesse sentido, de acordo com a imagem acima, *attention filter* é o filtro gerado através das multiplicações de matrizes, escalonamento e *softmax* realizadas com as matrizes Q e V e *original image* é a entrada do mecanismo de *self-attention*, ou seja, a matriz V. Portanto, para gerar a imagem filtrada, realizamos, basicamente, uma multiplicação entre o filtro de atenção gerado pelas matrizes Q e K e o conteúdo de entrada V.

Com isso, temos, finalmente, a equação desse mecanismo de *self-attention*, descrita abaixo.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

24.2.2 *Multi-head attention* e geração da saída

Na seção anterior, definimos apenas uma das cabeças do mecanismo de *multi-head attention*. Agora, precisamos gerar a saída para as outras cabeças.

Essas outras cabeças são necessárias para focar em outros elementos da sentença, ou, no caso do exemplo, em outros elementos da imagem.

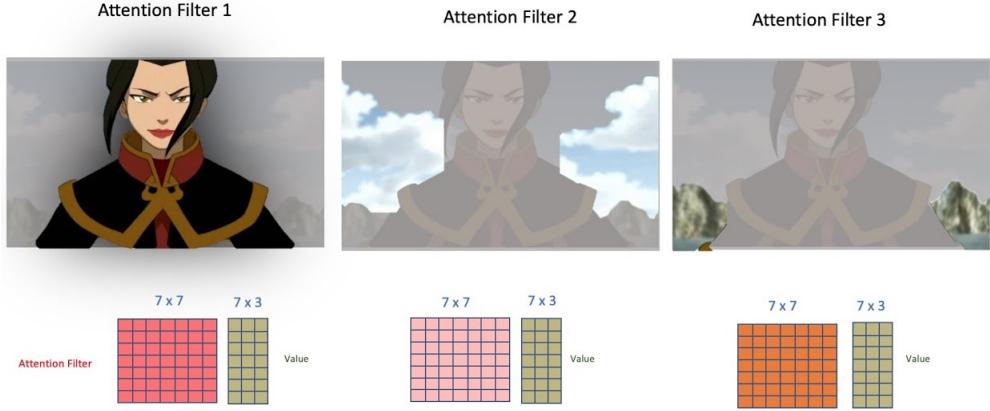


Figura 85: Exemplificação do mecanismo de *multi-head attention* com os filtros aplicados em diferentes na mesma imagem, porém, focando em diferentes elementos.

Definimos o mecanismo de *multi-head attention* da seguinte forma:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

onde W^O são os pesos da *linear layer* de saída e $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.

No final da camada de *multi-head attention* concatenamos essas três matrizes geradas com o filtro aplicado. Depois, através de conexões residuais (*residual skip connections* [8]), normalizamos a matriz de saída através do método de *layer normalization (LN)* [3] e geramos o saída através de uma *linear layer*. Na Figura 86 a seguir, podemos visualizar a estrutura de um *encoder* de um *transformer*.

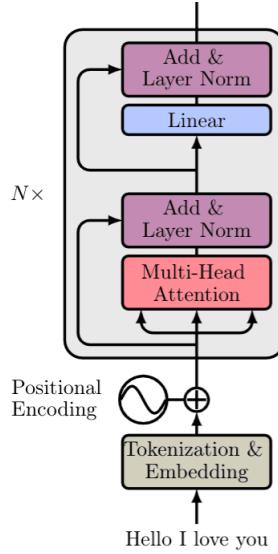


Figura 86: Representação da estrutura de um *encoder* de um *transformer*.

Por fim, para gerar o resultado da saída do modelo, devemos passar esses dados gerados na camada de *encoding* para a camada de *decoding*.

24.3 Decoder

O *decoder* consiste nos mesmos componentes do *encoder* com exceção de dois. Como visto anteriormente:

1. A sequencia de saída é computada da mesma forma através de *embedding* e *positional encoding*
 2. Esses vetores são passados para o primeiro bloco de *decoder*

Cada bloco de *decoder* contém as seguintes estruturas:

1. Uma camada de *Mask Multi-head Attention*
 2. Uma camada de normalização juntamente com uma conexão residual
 3. Uma nova camada de *multi-head attention* conhecida como *Encoder-Decoder attention*
 4. Uma nova camada de normalização seguida por conexões residuais
 5. Uma *linear layer* com uma terceira conexão residual

Assim, a estrutura do *decoder* está representada a seguir.

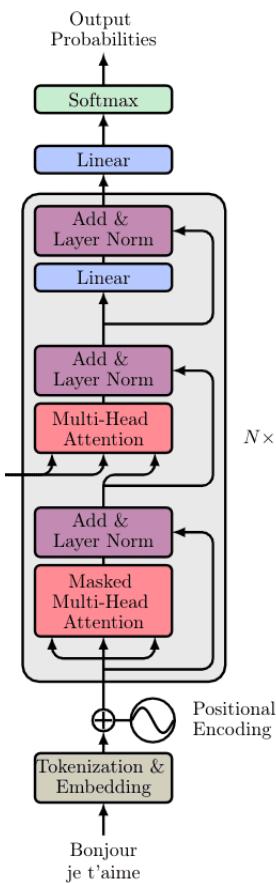


Figura 87: Representação da estrutura de um *decoder* de um *transformer*.

Na saída, "Output Probabilities" representa a probabilidade do próximo *token* da sequência.

Nas seções seguintes, serão detalhadas as subestruturas do *decoder*.

24.3.1 Masked Multi-head attention

Como o nosso objetivo, neste caso, é prever a próxima palavra da sequência, a camada de *masked multi-head attention* tem como objetivo "ensinar" o *decoder* a prever a melhor palavra a ser retornada. Para isso, atualizamos os valores da matriz de atenção de forma que o modelo preste atenção apenas

nas palavras que foram vistas anteriormente a fim de gerar a melhor que se relacione semanticamente com elas. Em outras palavras:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T + M}{\sqrt{d_k}}\right)V$$

onde a matriz M é uma matriz de zeros e $-\infty$.

Os zeros irão se tornar os valores do produto QK^T e os $-\infty$ irão se tornar zero. Esse tipo de máscara evita que o modelo preste atenção nas palavras futuras da sentença, focando nas palavras anteriores, que é o que realmente interessa.

24.3.2 Encoder-Decoder attention

Nesta etapa, a matriz gerada pela camada de *encoding* é passada como entrada para um mecanismo de *multi-head attention* juntamente com a matriz gerada pelo bloco *masked multi-head attention*.

A intuição a respeito dessa camada é que essa camada combina a sentença de entrada e saída, em outras palavras, ela irá ser treinada para associar a sentença de entrada com a sentença de saída correspondente.

25 Modelos generativos (*Generative models*)

25.1 Visão geral

Tanto o mundo real quanto o mundo digital são compostos por uma quantidade enorme de informações e tudo isso está disponível de forma acessível. Todavia, é um problema desenvolver modelos e algoritmos que possam analisar e entender essa quantidade absurda de dados.

Nesse sentido, os modelos generativos são uma das abordagens mais promissoras no subconjunto de *unsupervised learning*. Para treinar esse tipo de modelo, coletamos uma grande quantidade de dados de algum domínio (e.g. imagens, frases, sons, etc.) que chamaremos de $p_{data}(x)$ e, então, treinamos um modelo para gerar dados semelhantes aos de entrada, ou seja, geraremos $p_{model}(x)$ tal que é semelhante a $p_{data}(x)$.

A principal ideia é que as redes neurais que usamos como modelos generativos têm um número de parâmetros significativamente menor do que a quantidade de dados em que os treinamos, de modo que os modelos são forçados a descobrir e internalizar com eficiência a essência dos dados para gerá-los.

25.2 Aplicações

Com modelos generativos podemos criar amostras extremamente realistas como por exemplo modelos de arte, imagens, tarefas como *super-resolution* e colorização.



Figura 88: Exemplos de modelos gerativos. Na primeira imagem estão representados modelos de arte gerados a partir de ambientes. Na segunda imagem estão representadas imagens em alta resolução geradas a partir de identificação de rostos. Na terceira imagem estão representados exemplos de colorização de imagens a partir de uma "croqui" e imagens de treino.

Além disso, modelos gerativos podem ser aplicados em outras áreas de *unsupervised learning*, como por exemplo simulações e planejamentos usando *reinforcement learning*.

Com isso, temos três formas de modelarmos esse tipo de distribuição de modelos gerativos: através de *Auto-regressive Models*, *Auto-encoders* e GANs. Iremos discutir, separadamente, nas seções seguintes modelos como PixelRNN, PixelCNN, *Variational Autoencoder* e GANs.

25.3 *Auto-Regressive Models*

Antes mesmo de definirmos *Generative Adversarial Networks* (GANs) é importante mencionar que a principal diferença entre uma GAN e *Auto-regressive models* é que uma GAN aprende uma distribuição implícita dos dados, enquanto o último aprende uma distribuição explícita governada por uma estrutura de modelo imposta.

As principais vantagens da utilização de *Auto-regressive models* estão listadas a seguir:

1. **Fornece uma maneira de calcular a probabilidade:** esses modelos tem a vantagem de retornar probabilidades explícitas das densidades, tornando-o simples de aplicar em domínios como compressão e planejamento e explorações baseadas em probabilidade.
2. **O treino é estável:** existe um algoritmo estável que treina um *Auto-regressive model*
3. **Funciona tanto para dados discretos quanto contínuos**

Um dos problemas mais conhecidos de *unsupervised learning* é o de modelar a distribuição de imagens naturais. Para resolver esse problema, precisamos de um modelo tratável e escalonável. PixelRNN e PixelCNN fazem parte da classe de *Auto-regressive models* que atendem essas condições.

Esses tipos de modelos são, preferencialmente usados no preenchimento de imagens. A razão para isso é porque eles tem um desempenhos melhor do que outros modelos gerativos nesse tipo de problema.

25.3.1 PixelRNN

Através da utilização de modelos probabilísticos de densidade, como por exemplo Distribuição Normal, consegue gerar imagens começando através de um canto e calculando qual é o valor do próximo pixel que mais faz sentido de acordo com os pixels previamente gerados e com um valor de probabilidade.

Para processar a relação entre geração do pixel e valor de probabilidade, utilizamos de modelos que funcionam bem com sequências, como por exemplo, Redes Neurais recorrentes (RNNs).

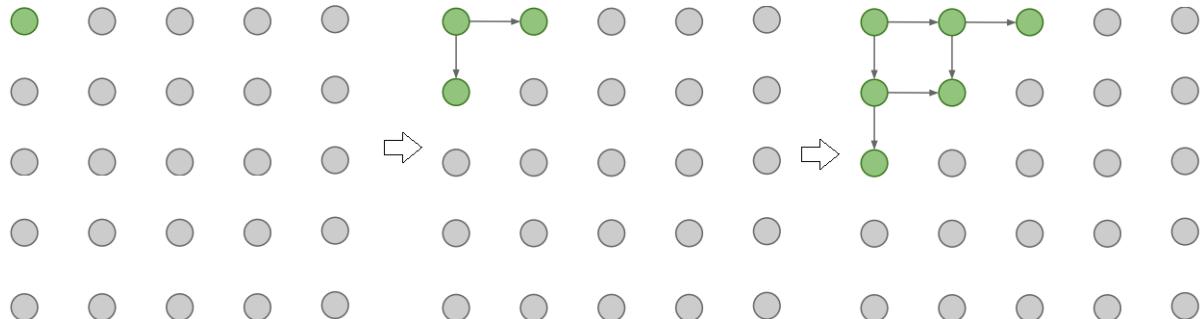


Figura 89: Geração de pixels utilizando PixelRNN. A partir de um pixel do canto da imagem, o modelo começa a gerar, sequencialmente outros pixels baseados nos valores de probabilidade e nos valores dos pixels previamente gerados.

A rede neural "varre" a imagem gerando, linha a linha e pixel a pixel a cada período de tempo, o que pode ser, muitas vezes, muito demorado. Os valores dos pixels são gerados baseando-se nos valores de probabilidade gerados através de uma distribuição que é escrita a partir do produto condicional das distribuições e os valores gerados são compartilhados através da imagem.

O objetivo, então é calcular a probabilidade $p(x)$ para cada pixel da imagem de tamanho $n \times n$. Assim, a probabilidade pode ser escrita da seguinte forma:

$$p(x) = \prod_{i=1}^{n^2} p(x_i | x_1, \dots, x_{i-1})$$

A equação acima é a probabilidade do i -ésimo pixel dada a probabilidade de todos os pixels previamente gerados. Essa geração se dá linha a linha e pixel por pixel. Além disso, cada pixel x_i é juntamente determinado por todos os três canais de cores RGB. Assim, a probabilidade condicional do i -ésimo pixel se torna:

$$p(x_{i,R}|X_{<i})p(x_{i,G}|X_{<i}, x_{i,R})p(x_{i,B}|X_{<i}, x_{i,R}, x_{i,G})$$

Portanto, cada cor é condicionada sobre as outras cores e os pixels previamente gerados.

Contudo, um dos principais problemas desse tipo de implementação é a velocidade. Como geramos cada pixel sequencialmente, isso pode ser extremamente lento de acordo com a complexidade da saída. Para resolver esse problema de otimização, podemos utilizar um método similar e paralelizável utilizando ConvNets, chamado PixelCNN.

25.3.2 PixelCNN

Como vimos, a utilização do método PixelRNN, pode, muitas vezes, ser mais lento que o desejável, para isso, podemos adicionar camadas de convolução no nosso modelo. PixelCNN usa esse tipo de camada com o objetivo de paralelizar as operações de geração de pixel, preservando a resolução espacial da imagem gerada.

Da mesma forma que PixelRNN, PixelCNN gera a imagem a partir de um canto da imagem, porém, agora, utilizando de uma ConvNet para paralelizar esse processo.

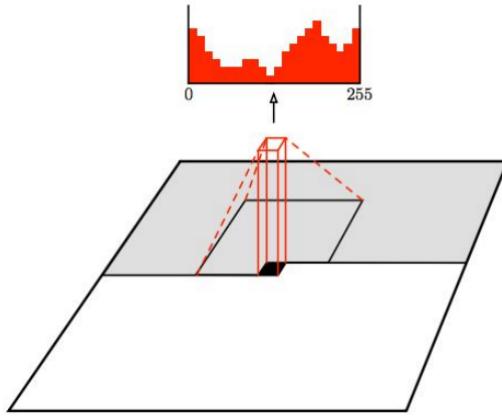


Figura 90: Geração de uma imagem através do método PixelCNN. Podemos perceber que os valores dos pixels são gerados sequencialmente através de uma camada de convolução. Os pixels previamente gerados estão representados em cinza e os valores de probabilidade são calculados paralelamente através da convolução.

Como a geração dos pixels continua sequencial, isso pode ser um ponto negativo que, muitas vezes, pode tornar a geração da imagem, ainda muito lenta. Para isso, nas próximas seções iremos discutir métodos mais eficientes para realizar esse procedimento.

25.4 Variational Autoencoders (VAE)

Como vimos até agora, PixelCNNs definem a função de densidade travável e otimiza a probabilidade de dados de treinamento. De outra forma, VAEs definem funções de probabilidade intratáveis de acordo com um valor z que é chamado de espaço latente e será definido posteriormente.

Dessa forma, um VAE é um *Autoencoder* cuja distribuição de codificações é regularizada durante o treinamento, a fim de garantir que seu espaço latente tenha boas propriedades, o que nos permite gerar novos dados.

Em geral, *Autoencoders* são redes neurais para redução de dimensionalidade. A estrutura básica de um *Autoencoder* está representada na Figura 91. Essas redes neurais têm como objetivo aprender o melhor esquema de codificação-decodificação usando um processo de otimização iterativo.

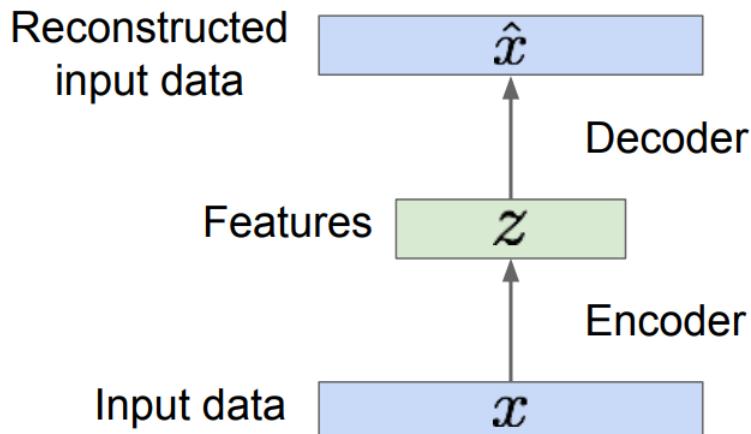


Figura 91: Através de dados de entrada x treinamos o nosso modelo de forma que possamos reconstruir os dados originais através do *Autoencoder*. Dessa forma, z são os dados de entrada codificados através de algoritmos de redução de dimensionalidade e \hat{x} é o conjunto de dados decodificado a partir de z .

Assim, a arquitetura geral de um *Autoencoder* cria um gargalo de dados que carante que apenas a parte estruturada principal da informação possa ser reconstruída. Os codificadores e decodificadores são transformações lineares simples que podem ser expressas como matrizes, da mesma forma que o algoritmo PCA funciona e no final, adicionamos não-linearidade.

Autoencoders são, portanto arquiteturas de codificador-decodificador que podem ser treinadas utilizando um algoritmo de otimização como, por exemplo, gradiente descendente.

Finalmente, podemos definir um VAE. VAE é uma arquitetura composta por um codificador e um decodificador treinada para minimizar o erro de reconstrução entre os dados decodificados e os dados iniciais. Assim, podemos treinar o modelo da seguinte forma:

1. A entrada é codificada como distribuição do espaço latente z
2. Um ponto do espaço latente é amostrado a partir dessa distribuição
3. O ponto amostrado é decodificado e o erro de reconstrução pode ser calculado
4. *Backpropagation* utilizando o erro de reconstrução

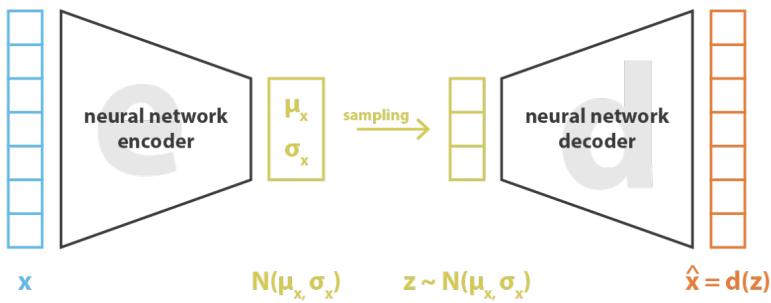
Na Figura 92 a seguir, está esquematizado o processo de treino do modelo, sabendo que x representa os dados de entrada, z o espaço latente e $d(z)$ a reconstrução da entrada.



Figura 92

Com isso, a função custo que é minimizada ao treinar um VAE é composta por um "termo de reconstrução" que torna o esquema de codificação-decodificação mais eficiente e um "termo de regularização" que regulariza a organização do espaço latente, tornando-a mais próxima da Distribuição Normal Padrão. Esse termo é expresso como a divergência de Kullback-Leibler, que sua definição não faz parte do escopo deste curso.

A seguir, na Figura 93 está esquematizada uma estrutura básica de um VAE com a definição da sua função custo, que são dependentes dos valores de entrada x , da distribuição do espaço latente $\mathcal{N}(\mu_x, \sigma_x^2)$ e da aproximação gerada a partir do espaço latente $\hat{x} = d(z)$.



$$\text{loss} = \|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)] = \|x - d(z)\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$

Figura 93

Portanto, através de VAEs podemos utilizar amostras usadas no espaço latente para gerar dados similares através do decodificador. A figura abaixo mostra os dados gerados a partir de uma rede decodificadora de um VAE treinada a partir da base de dados MNIST. Além disso, um dos pontos negativos da utilização de VAEs é que as amostras geradas são embaçadas e de baixa qualidade. Portanto para a melhoria da qualidade utilizamos GANs, que serão descritas a seguir.

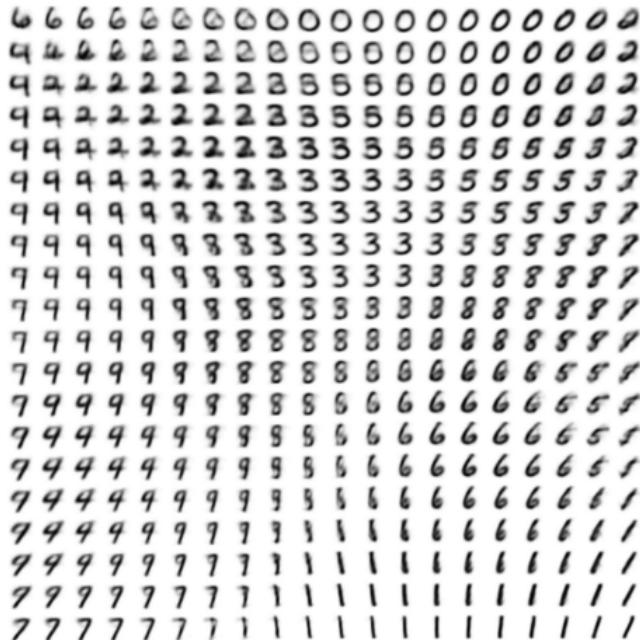


Figura 94: Nesta imagem estão representados os dígitos gerados por uma rede VAE através do decodificador. A imagem tem esse padrão devido à forma bidimensional da Distribuição Normal. Como se pode perceber, os dígitos distintos estão em diferentes regiões da imagem devido ao espaço latente.

25.5 Generative Adversarial Networks (GANs)

Até agora, trabalhamos com funções de densidade de probabilidade explícita, como foram vistas em PixelCNNs e VAEs. No caso das GANs, não trabalhamos com funções explícitas de probabilidade, usamos aproximações probabilísticas baseadas na Teoria Dos Jogos, aprendendo a gerar dados a partir da distribuição de treinamento através do jogo de dois jogadores.

Com GANs, o que desejamos é a partir de uma entrada inicializada aleatoriamente, através de uma rede neural geradora, geramos uma amostra baseada na distribuição de treino, como mostra a Figura 95.

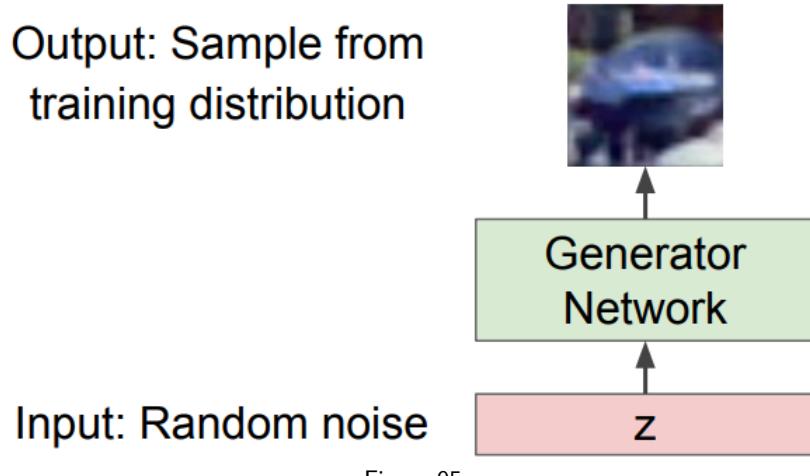


Figura 95

A seguir iremos descrever o processo de treino de GANs.

25.5.1 Treinamento: Jogo de dois jogadores

A forma que iremos treinar e fazer com que o nosso modelo aprenda a geração de dados é através da visualização desse problema como um jogo de dois jogadores. Existem dois jogadores: a rede neural geradora e a rede neural discriminadora.

A rede neural geradora (ou *Generator network*) tenta enganar o discriminador gerando imagens de aparência real. E a rede neural discriminadora (ou *Discriminator network*) tenta distinguir imagens reais e falsas. A representação desse "jogo" mostrada a seguir na Figura 96.

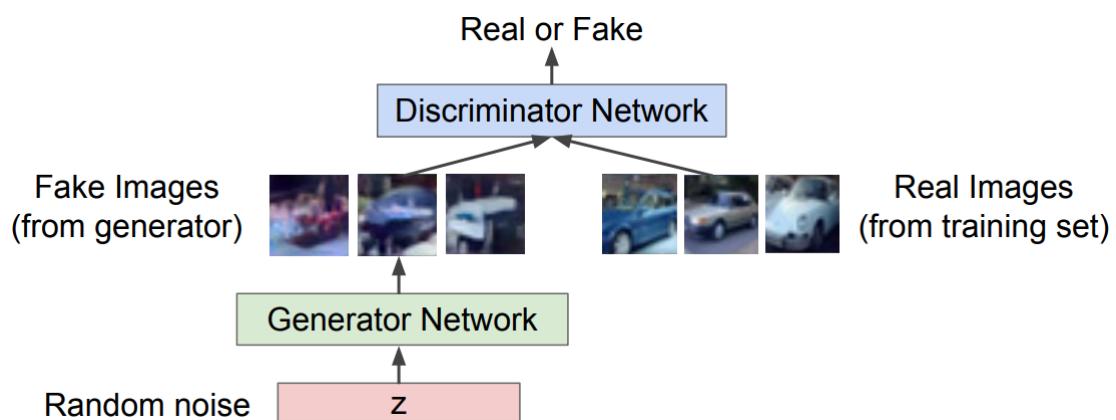


Figura 96: A partir de uma imagem inicializada aleatoriamente z , uma rede neural gerado, representada em verde, gera imagens falsas, porém muito semelhantes às reais e uma rede neural discriminadora, representada em azul, tenta distinguir se as imagens dadas como entrada para essa rede neural são falsas ou reais.

Utilizamos o algoritmo Minimax para realizar o treinamento de uma GAN, cuja função principal está descrita a seguir.

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Saída do discriminador para dados reais } x} + \mathbb{E}_{z \sim p_z} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Saída do discriminador para dados falsos gerados } G(z)}) \right]$$

Com essa equação, desejamos maximizar o objetivo do discriminador θ_d , de forma que $D(x)$ é próximo de 1 (imagem real) e $D(G(z))$ é próximo de 0 (imagem falsa). E desejamos minimizar o objetivo do gerador θ_g de forma que $D(G(z))$ é próximo de 1 (discriminador é enganado a pensar que $G(z)$ é real).

Com isso, utilizamos métodos de maximizar e minimizar os discriminadores e geradores, respectivamente. Usamos o método de gradiente ascendente para o discriminador

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p_z} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

e o método de gradiente descendente para minimizar a perda do gerador, porém, como desejamos "enganar" o discriminador, também podemos usar gradiente ascendente para o gerador

$$\max_{\theta_g} \mathbb{E}_{z \sim p_z} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Com isso, podemos descrever o seguinte algoritmo para o treinamento de GANs:

Algorithm 13 Algoritmo de treino de uma GAN

```

1: procedure
2:   for numero de iterações de treino do
3:     for  $i = 1$  to  $k$  do
4:       Crie amostras minibatch de  $m$  amostras inicializadas aleatoriamente  $\{z^{(1)}, \dots, z^{(m)}\}$  a partir de  $p_g(z)$ 
5:       Crie amostras minibatch de  $m$  exemplos  $\{x^{(1)}, \dots, x^{(m)}\}$  a partir da geração de dados pela distribuição  $p_{data}(x)$ 
6:       Atualize o discriminador pelo gradiente ascendente
7:        $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)})))]$ 
8:     end for
9:     Crie amostras minibatch de  $m$  amostras inicializadas aleatoriamente  $\{z^{(1)}, \dots, z^{(m)}\}$  a partir de  $p_g(z)$ 
10:    Atualize o gerador pelo gradiente ascendente
11:     $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$ 
12:  end for
13: end procedure

```

Após o treinamento, podemos usar a rede neural geradora para gerar novas imagens. Podemos perceber na Figura 97 alguns exemplos de imagens geradas por uma GAN.



Figura 97: Imagens geradas a partir do treinamento de uma GAN. As imagens contornadas em amarelo são imagens do conjunto de treino e, portanto as imagens da mesma linha são as vizinhas mais próximas.

25.5.2 Deep Convolutional GANs

GANs implementadas através de CNNs, ou também chamadas de *Deep Convolutional GANs* (DC-GANs) são redes neurais com filtros de convolução, como pode-se perceber na Figura 98

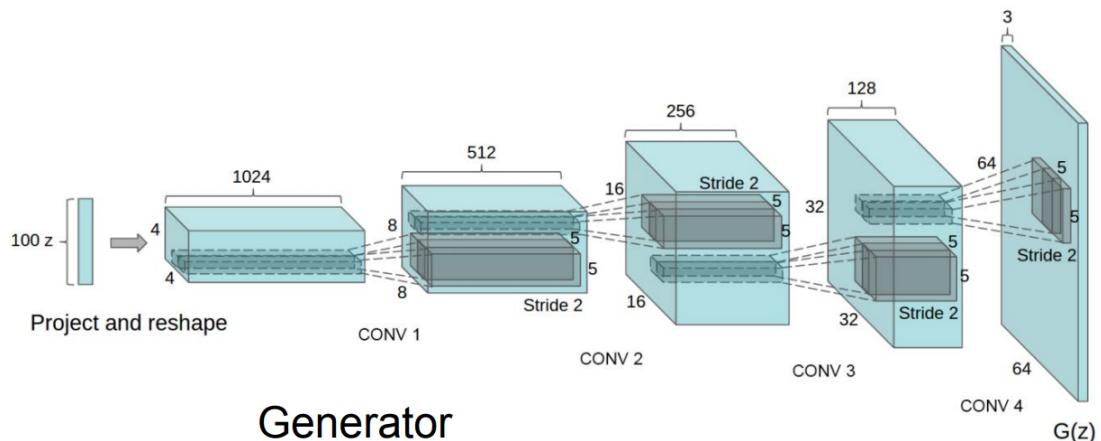


Figura 98: Representação de uma *Deep Convolutional GAN* no processo de geração de uma imagem. Percebe-se o aumento de resolução através de camadas de convolução utilizando *upsampling*.

Com esse tipo de arquitetura, podemos gerar imagens de alta qualidade, como pode-se perceber na Figura 99.

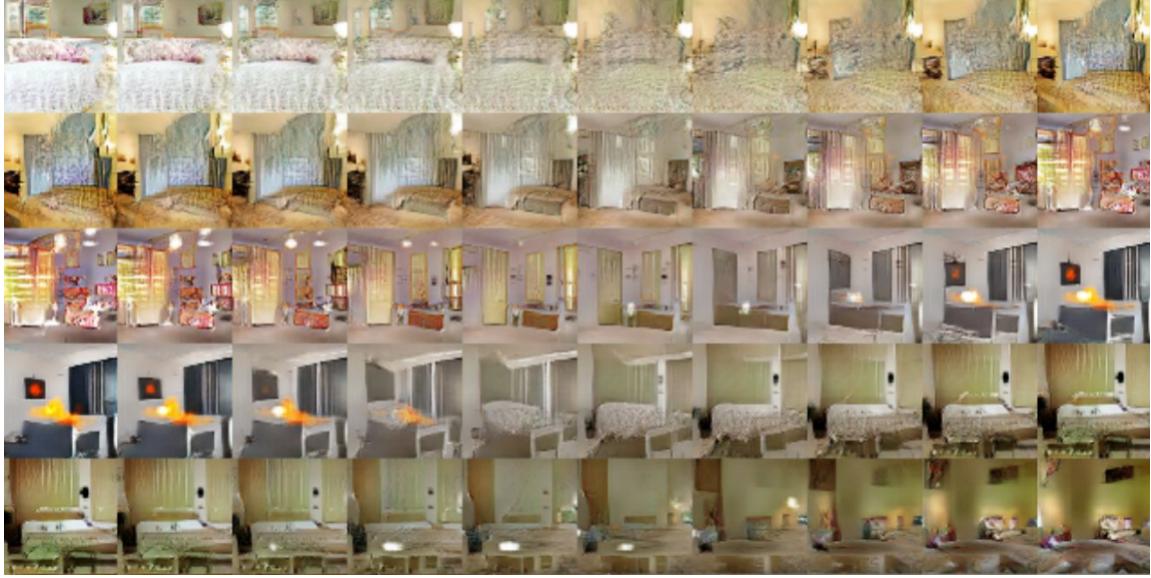


Figura 99: Exemplos de imagens geradas por *Deep Convolutional GANs*. Percebe-se que as imagens geradas são de alta resolução.

26 Deep Q-Learning

Como foi visto na Seção 19, através do método de *Q-learning*, atualizamos os valores das políticas do agente maximizando os valores de Q de acordo com o estado futuro do ambiente e da ação tomada, de maneira recursiva, usando a equação de Bellman.

Contudo, apesar do grande poder desse método, de acordo com a complexidade do ambiente, ele pode ser tornar extremamente ineficiente. Para resolver esse problema, usamos um método que engloba *Q-learning* e redes neurais profundas, chamado *Deep Q-learning* (DQN).

De maneira geral, ao invés de trabalharmos com uma tabela para atualizar os valores de Q de decisão de ação do agente, utilizamos uma rede neural. Essa rede neural funciona através de uma função de aproximação, chamada de aproximador, denotado por $Q(s, a; \theta)$, em que θ representa os pesos treináveis da rede, para aproximar os valores de Q de forma ótima.

Assim, a equação de Bellman, vista na Seção 19.6 é utilizada como a função custo, que deve ser minimizada. Em outras palavras, minimizamos a diferença entre a igualdade do valor Q que deve ser atualizado em relação a soma da recompensa com o desconto do máximo valor de Q dos estados futuros. Ou seja,

$$\boxed{Cost = [Q(s, ; \theta) - (r(s, a) + \gamma \max_a Q(s', a; \theta))]^2}$$

onde $Q(s, ; \theta)$ é chamado de *Q-target*, os parâmetros da rede neural.

O treinamento desse tipo de rede neural acontece durante o momento de *exploration* e *exploitation*, atualizando os valores de Q já vistos e que serão descobertos futuramente. Para o treino, portanto, selecionamos b estados já visitados, juntamente com os seus respectivos valores, de maneira aleatória, e usamos esses valores como *input* e *target*, respectivamente.

Diferentemente do método de *Q-learning* tradicional, DQN prevê os N possíveis valores de Q para um

determinado estado. Na Figura 100 abaixo, está representada essa diferença entre os dois métodos.

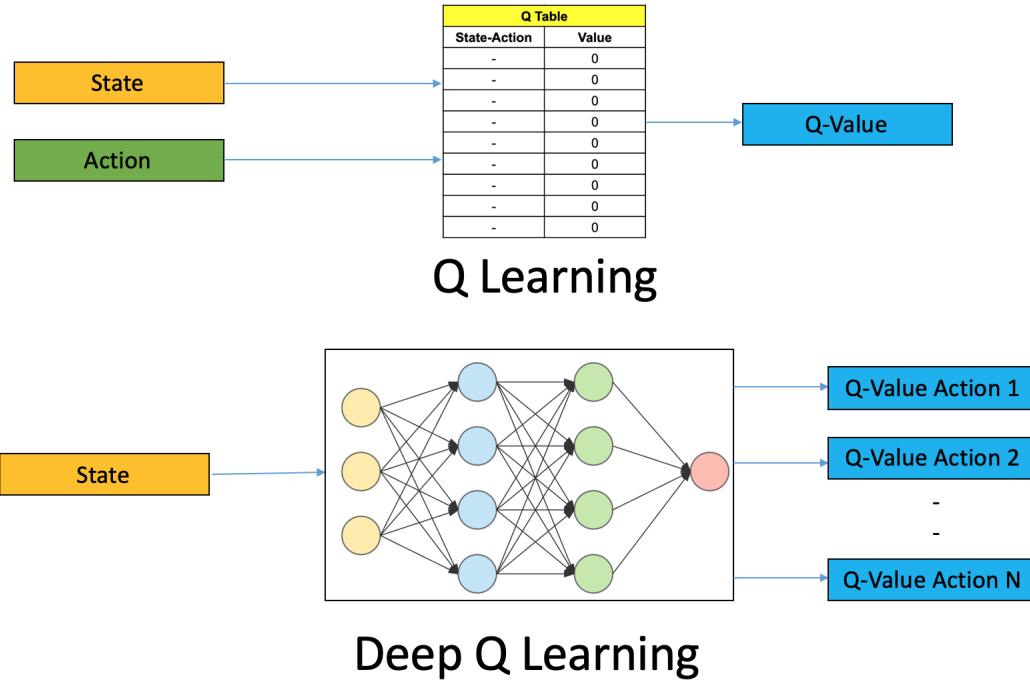


Figura 100: Representação dos dois métodos de *Q-learning*. Na figura do topo, a partir de uma determinada ação tomada em um estado de ambiente, gera-se um valor de *Q* referente a este par estado-ação. Na figura de baixo, dado um estado, a rede neural calcula todos os valores de *Q* referentes a todas as ações possíveis a serem tomadas, selecionando o melhor valor para aquele estado.

Com isso, podemos listar as etapas que envolvem o aprendizado de uma rede neural.

1. Passar como entrada estado atual s do ambiente para a rede neural. Essa rede neural irá retornar os valores Q de todas as possíveis ações a serem tomadas pelo agente no estado s .
2. Selecionar uma ação usando a política ϵ -greedy. Com a probabilidade ϵ , selecionar uma ação aleatória a (*exploration*) e com a probabilidade $1-\epsilon$ escolher a ação que corresponde o valor máximo de Q , tal que $a = \text{argmax}(Q(s, a; \theta))$
3. Realizar a ação tomada a no estado s e mover para o próximo estado s' para receber a recompensa.
4. Após, tomamos uma decisão aleatória de transição de estado e calculamos o custo
5. Minimizar o erro da função custo utilizando algum método de otimização, como por exemplo, gradiente descendente
6. A cada C iterações, copiar os pesos da rede neural atual para uma rede neural *target* a fim de salvar os pesos
7. Repetir esses passos M episódios

Com DQN, temos uma ferramenta extremamente forte para resolução de problemas de aprendizado de RL. Agora, com ambientes mais complexos e com mais interações, o agente consegue realizar uma melhor valoração das possibilidades de ações que podem ser tomadas baseadas no retorno da rede

neural. Atualmente, jogos de Atari, Go e StarCraft já atingiram níveis humanos de comportamento utilizando DQN e outros métodos de IA.

27 Busca de Monte Carlo

O método de Busca em Árvore de Monte Carlo (do inglês, Monte Carlo Tree Search (MCTS)) é extremamente poderoso para diversas aplicações que envolvem IA e teoria dos jogos. Uma das principais aplicações desse método na literatura foi, em 2016, no artigo publicado pela DeepMind [19], no qual atingem níveis humanos no jogo Go utilizando técnicas de DQN, vistas na Seção 26 e MCTS.

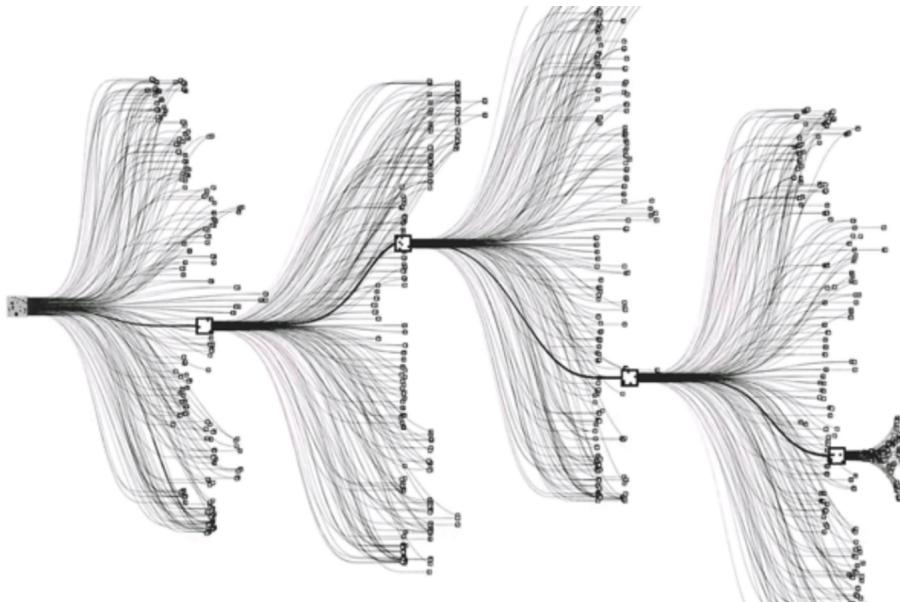


Figura 101: Representação de uma árvore de Monte Carlo gerada a partir de um estado de jogo do jogo Go. Cada nodo da árvore representa uma possibilidade de jogada baseada no estado de jogo imediatamente anterior.

O principal objetivo de MCTS é valorar estados intermediários do ambiente de forma que não seja necessário atingir estados finais para encontrar a melhor possibilidade de recompensa.

Para introduzir esse método, será definido, primeiramente um método *uninformed search*, que é mais simples e ineficiente, para depois descrever o método MCTS.

27.1 *Uninformed search*

É um algoritmo de busca genérica o qual percorre todos os nodos da árvore a fim de valorar os estados intermediários.

Para realizar essa valoração, temos buscas em profundidade e em largura, como pode ser visto na Figura 102 abaixo.

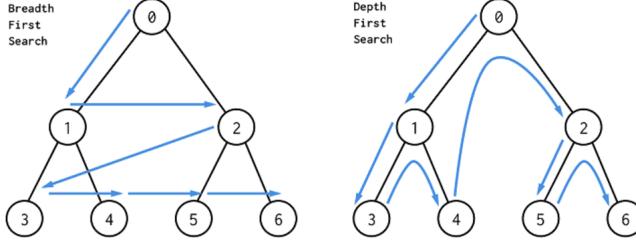


Figura 102: Representação das buscas em profundidade e largura. Percebe-se que todos os nodos da árvore são acessados durante a busca.

Percebe-se que uma árvore com b ramos por nível e profundidade d teria b^d número de nós folha. Por exemplo, no jogo Go, cujo número de ramos por nível é 250, no nível 5 da árvore, teríamos $250^5 = 976.562.500.000$ possibilidades de estados.

O maior problema desse tipo de busca é que, quanto maior a complexidade do problema, maior a profundidade e largura da árvore, o que torna o método extremamente ineficiente.

27.2 Monte Carlo Tree Search

MCTS é um método de busca baseado no método de *uninformed search*, porém, mais inteligente. MCTS usa simulação de Monte Carlo [17] para acumular estimativas de valor para orientar em qual direção ou trajetória da árvore devemos seguir para maximizar a recompensa na árvore de busca. Em outras palavras, MCTS presta mais atenção em nodos que são mais promissores, fazendo não seja necessário acessar todos os nodos da árvore para valorar um estado.

Basicamente, esse método consiste em quatro etapas bem separadas: seleção, expansão, simulação e *backup*.

No passo de simulação, utilizamos *tree policy* para construir um caminho da raiz até o nodo folha mais promissor. *Tree policy* é utilizada para selecionar a melhor ação a partir de um estado. No AlphaGo, o cálculo dessa política se dá através do valor UCB, que calcula o grau de confiança de uma determinada ação ser a mais promissora. Abaixo está representado o cálculo desse valor.

$$UCB(node_i) = V_i + 2\sqrt{\frac{\ln N}{n_i}}$$

onde, V_i é a média de recompensa de todos os valores abaixo de $node_i$. N é o número de vezes que o pai de $node_i$ foi visitado e n_i é o número de vezes que $node_i$ foi visitado.

Com isso, percebe-se que quanto mais um nodo i é visitado, menor é o valor UCB, diminuindo a probabilidade desse nodo ser selecionado novamente. Assim, usa-se técnicas de *exploration* e *exploitation*, vistas na Seção 19 para a valoração dos estados através do valor UCB.

Na etapa de expansão, apenas selecionamos um nodo de forma aleatória para ser explorado.

Na etapa de simulação, simulamos uma ou mais jogadas para verificar a recompensa acumulada de cada ação. Por exemplo, para cada simulação teríamos valores acumulados referentes à vitória, derrota ou empate.

Na etapa de *backup* usamos os valores de recompensa acumulados das ações tomadas e estados

atingidos na etapa de simulação, propagando esses valores em direção ao nodo raiz a fim de possibilitar a realização da melhor transição de estado.

A Figura 103 abaixo exemplifica de forma conjunta cada uma dessas etapas.

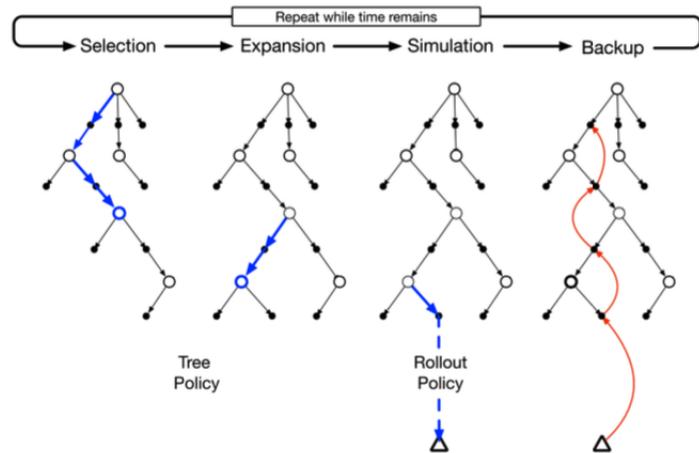


Figura 103: Exemplificação das etapas de seleção, expansão, simulação e *backup*

Com isso, percebe-se que MCTS é uma ferramenta extremamente poderosa, especialmente quando utilizada juntamente com métodos como DQN. MCTS valora as recompensas de cada estado sem necessariamente percorrer toda a árvore para decidir a melhor jogada e DQN computa o melhor valor Q de um determinado estado atingido por determinada ação.

Referências

- [1] Vinod Nair Alex Krizhevsky e Geoffrey Hinton. *THE CIFAR-10 DATABASE*. <https://www.cs.toronto.edu/~kriz/cifar.html>. 2012.
- [2] Kian Katanforoosh Andrew Ng. *Stanford University CS230 Deep Learning*. URL: <http://cs230.stanford.edu/>.
- [3] Jimmy Lei Ba, Jamie Ryan Kiros e Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].
- [4] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957. ISBN: 9780486428093.
- [5] Emma Brunskill. *Stanford University CS234 Reinforcement Learning*. URL: <http://web.stanford.edu/class/cs234/index.html>.
- [6] John Hewitt Christopher Manning. *Stanford University CS224n: Natural Language Processing with Deep Learning*. URL: <http://web.stanford.edu/class/cs224n/index.html#schedule>.
- [7] Serena Yeung Fei-Fei Li Justin Johnson. *Stanford University CS231n: Convolutional Neural Networks for Visual Recognition*. URL: <http://cs231n.stanford.edu/>.
- [8] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].
- [9] Yann LeCun. *THE MNIST DATABASE of handwritten digits*. <http://yann.lecun.com/exdb/mnist/>. 1998.
- [10] Shane Legg e Marcus Hutter. *Universal Intelligence: A Definition of Machine Intelligence*. 2007. arXiv: 0712.3329 [cs.AI].
- [11] DeepMind & University College London. *Reinforcement learning course 2020*. URL: https://www.youtube.com/playlist?list=PLqYmG7hTraZBKeNJ-JE_eyJHZ7XgBoAyb.
- [12] Tom M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.
- [13] Andrew Ng. *Stanford University CS229 Machine Learning*. URL: <http://cs229.stanford.edu/>.
- [14] Andrew Ng. *Stanford University Machine Learning*. URL: <https://www.coursera.org/learn/machine-learning?>.
- [15] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. Em: (2019).
- [16] Sebastian Raschka. *Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning*. 2020. arXiv: 1811.12808 [cs.LG].
- [17] Samik Raychaudhuri. “Introduction to Monte Carlo simulation”. Em: *2008 Winter Simulation Conference*. 2008, pp. 91–100. DOI: 10.1109/WSC.2008.4736059.
- [18] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. Em: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229. DOI: 10.1147/rd.33.0210.
- [19] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. Em: *Nature* 529 (2016), pp. 484–503. URL: <http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html>.
- [20] Sander Dieleman Viorica Patraucean James Martens Marta Garnelo Felix Hill Alex Graves Andriy Mnih Mihaela Rosca Irina Higgins Jeff Donahue Iason Gabriel Chongli Qin Thore Graepel Wojtek Czarnecki. *DeepMind & University College London Deep learning lecture series 2020*. URL: <https://www.youtube.com/playlist?list=PLqYmG7hTraZCDxZ44o4p3N5Anz31LRVZF>.

- [21] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].
- [22] P.J. Werbos. “Neural networks for control and system identification”. Em: *Proceedings of the 28th IEEE Conference on Decision and Control*, 1989, 260–265 vol.1. DOI: 10.1109/CDC.1989.70114.