Carlos Henrique Reis

# Otimização de Hiperparâmetros em Redes Neurais Profundas

Itajubá

2018

Carlos Henrique Reis

# Otimização de Hiperparâmetros em Redes Neurais Profundas

Monografia apresentada como trabalho final de graduação, requisito parcial para obtenção do título de Bacharel em Ciência da Computação, sob orientação da Prof.a Isabela Neves Drummond.

Universidade Federal de Itajubá - UNIFEI

Programa de Graduação

Orientadora: Isabela Neves Drummond

Itajubá

2018

# Agradecimentos

À DEUS por todas as bênçãos, oportunidades e pessoas que coloca em meu caminho. Gostaria de agradecer a todos aqueles que contribuíram com algo de bom para mim, durante todos os anos em que estive na graduação. O Carlos que aqui apresenta o seu trabalho final de graduação é o resultado de um somatório de muitas coisas boas, vindas de muitas pessoas diferentes.

Agradeço à minha família, sem os quais eu não teria tido condições de chegar até aqui. Todo o alicerce em que me sustenta é obra de meus queridos pais Carlos e Augusta e as minhas três irmãs Danielle, Michelle e Micaelle que sempre me compreenderam.

Agradeço aos meus amigos de graduação Vinicius, Pedro, Karen, Luana e todos os meu colegas de conheci no curso de Sistemas de Informação turma de 2014. Também agradeço aos amigos que fiz com a turma de Ciência da Computação em especial a Victor, Leandro e Francis, pessoas que levarei comigo pelo resto da vida.

Aos amigos que fiz durante a minha vida Acadêmica Tábata, George, Fábio, Felipe, Eduardo, Anthony, Thainnan, Maysa, Thayana e muitos outros que sempre estiveram dispostos a me ajudar.

Agradeço a minha querida Leticia, companheira que sempre esteve ao meu lado, sempre me ouvindo, dizendo palavras de conforto nos momentos difíceis e palavras muito divertidas nos momentos mais alegres. Com certeza teria sido tudo mais difícil sem você por perto.

Agradeço aos amigos Leonato, Samir, Any Karolyne e Bruno, que fiz na Tuuris.com empresa que me acolheu e onde fiz meu estágio obrigatório, contribui muito para minha formação. Além do Eduardo por toda compreensão e paciência

Também quero agradecer a todos os professores do Instituto de Matemática e Computação da UNIFEI. Sinto-me um grande profissional neste fim de curso, e ainda assim acho que sei apenas uma ínfima parte daquilo que cada um tem para oferecer. A todos os professores com quem tive aula, sem nenhuma exceção, o mínimo que posso oferecer de volta é o meu eterno respeito e meu muito obrigado. Mais em especial, agradeço ao professor José Arnaldo Barra Montevechi que, durante dois anos, foi meu orientador de Iniciação Científica. À professora Isabela Neves Drummond, obrigado pelos primeiros passos na área de IA, e por me acompanhar neste trabalho final de Graduação.

Por fim, agradeço ao Instituto de Matemática e Computação pela estrutura oferecida. Na minha humilde opinião, a estrutura de prédios, salas de aula, laboratórios e biblioteca, é apenas o reflexo da altíssima qualidade do curso de Ciência da Computação da UNIFEI.

*"Confia teus negócios ao Senhor e teus planos terão bom êxito"*
*Bíblia Sagrada, Provérbios 16, 3*

# Resumo

Aprendizado profundo é uma das mais recentes áreas de pesquisa em inteligência artificial, que envolve a aplicação de um subconjunto de ferramentas e técnicas de aprendizado de máquina, permitindo que modelos computacionais compostos por múltiplas camadas de processamento aprendam a representação do dado em múltiplos níveis de abstração. Um dos grandes desafios quando se trabalha com modelos de aprendizagem profunda é a dificuldade de encontrar a melhor configuração dos hiperparâmetros destes modelos, ou seja, qual a combinação de valores é a mais adequada para se obter o melhor desempenho do algoritmo de aprendizado durante a fase de treinamento. Este trabalho apresenta um breve estudo sobre métodos de otimização de hiperparâmetros em modelos de aprendizado profundo com foco em classificação de dados. Assim, o objetivo principal deste trabalho final de graduação é a análise da aplicação do aprendizado profundo para classificação de dados e a otimização de hiperparâmetros para modelos de aprendizagem profunda. Foi selecionada para análise a rede neural profunda do tipo *Deep Feedforward Network* e, em conjunto, foi aplicado um algoritmo automático de seleção dos hiperparâmetros para este modelo. Os resultados gerados foram obtidos a partir da aplicação da implementação realizada em 3 conjuntos de dados disponíveis para estudo de classificadores, buscando avaliar a implementação realizada. Os resultados dos testes demonstram grande potencial na otimização automática de hiperparâmetros para modelos de aprendizagem profunda.

**Palavras-chaves**: Aprendizado de Máquina. Aprendizado Profundo. *Tensorflow*.

# Abstract

Deep Learning is of the most recent area of research in artificial intelligence, focusing even more closely on the application of a subset of Machine Learning tools and techniques, allowing computational models composed of multiple layers of processing to learn the representation of the data in multiple levels of abstraction. One of the great challenges when working with Deep Learning models is the difficulty of finding the best configuration of the Hyperparameters of these models, that is, which combination of values is the most adequate to obtain the best performance of the learning algorithm during training. This paper presents a brief study on Hyperparameter Optimization methods in Deep Learning models focused on data classification. One of the great challenges when working with models of deep learning is the difficulty of finding the best configuration of the hyperparameters of these models, that is, which combination of values is the most adequate to obtain the best performance of the learning algorithm during training. This paper presents a brief study on methods of optimization of hyperparameters in deep learning models focused on data classification. Thus, the main objective of this final undergraduate work is the analysis of the application of deep learning for data classification and the optimization of hyperparameters for models of deep learning. The Deep Feedforward Network type neural network was selected for analysis and, together, an automatic algorithm was selected for selection of the hyperparameters of the neural network. The results obtained were begeted from the application of the implementation carried out in 3 data sets available for the study of classifiers, seeking to evaluate the implementation. The results of the tests demonstrate great potential in the automatic optimization of hyperparameters for deep learning models.

**Key-words**: Machine Learning. Deep Learning. *Tensorflow*.

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| AM | Aprendizado de Máquina |
| BA | Busca Aleatória |
| CAEs | *Convolutional Auto-Encoder* |
| CNNs | *Convolutional Neural Networks* |
| CSV | *Comma-separated-values* |
| DNNs | *Deep Neural Networks* |
| DSNs | *Deep Stacking Networks* |
| FE | *Feature Engineering* |
| GRUs | *Gated Recurrent Unit* |
| IA | Inteligência Artificial |
| MLP | *Multilayer Perceptron* |
| MDRNNs | *Multidimensional Recurrent Neural Networks* |
| RNNs | *Recurrent Neural Networks* |
| RNAs | Redes Neurais Artificiais |
| T-DSN | *Tensor Deep Stacking Network* |
| VC | Validação Cruzada |

# Sumário

# 1 Introdução

A Inteligência Artificial (IA) é considerada uma área que possui uma grande variedade de técnicas, capazes de solucionar certa gama de problemas reais considerados complexos como, por exemplo, fazer algumas análises de dados de vendas de uma loja para agrupar os clientes em grupos de forma a ter melhores resultados nas operações de marketing (CARVALHO et al., 2011). O conceito de IA não se apresenta bem definido na literatura, podendo variar entre abordagens centradas nos seres humanos e na natureza (ciência empírica) ou uma abordagem racionalista, que envolve conceitos/modelos matemáticos bem definidos (RUSSELL; NORVIG, 2014).

Para Carvalho et al. (2011) atualmente boa parte das técnicas de IA são autônomas, ou seja, reduziu-se a necessidade de intervenção humana e consequentemente limitou-se as chances de falhas. Considerando que as técnicas de IA passaram a ser capazes de criar, a partir de experiências passadas, hipóteses ou funções, que solucionam problema que se deseja resolver (indução). Essa capacidade define o que se conhece como Aprendizado de Máquina (AM). O problema de classificação pode ser considerado um assunto central no AM. Para Haykin (2007) o processo de classificação pode ser entendido como: dado um relacionamento conhecido, identifique a classe à qual os dados pertence.

A chamada aprendizagem profunda, concentra-se ainda mais estreitamente na aplicação de um subconjunto de ferramentas e técnicas de AM, permitindo que modelos computacionais compostos por múltiplas camadas de processamento aprendam a representação do dado em múltiplos níveis de abstração (LECUN; BENGIO; HINTON, 2015).

O estudo da Aprendizagem Profunda não converge somente no processo de aprendizagem, mas também, no desenvolvimento de Redes Neurais denominadas Redes Neurais Profundas, que possuem uma complexidade necessária para lidar com a classificação de grandes conjuntos de dados (SCHMIDHUBER, 2014). Neste contexto, para Haykin (2007), grande parte do esforço em pesquisa sobre Redes Neurais possui foco em classificação de padrões.

Para Deng, Yu et al. (2014) a aprendizagem profunda pode ser aplicada a qualquer tipo de dado: sinais digitais, áudio, vídeo, texto, produzindo conclusões que parecem ter sido alcançadas por humanos, porém mais rapidamente.

Segundo Goodfellow, Bengio e Courville (2016) um dos grandes desafios quando se trabalha com modelos de AM é definir qual é a melhor configuração do modelo. Essa configuração pode ser entendida como hiperparâmetros. Os hiperparâmetros são configurações de um modelo de AM que pode ser ajustado para otimizar o desempenho e

a qualidade do algoritmo de aprendizado(BERGSTRA; BENGIO, 2012).

O ajuste de hiperparâmetros depende mais de resultados experimentais do que de teoria. Um dos métodos para determinar as configurações ótimas é testar várias combinações diferentes para avaliar o desempenho de cada modelo. Tal técnica é conhecida como Busca Aleatória (BERGSTRA; BENGIO, 2012). Entretanto, para Patterson e Gibson (2017), avaliar cada modelo baseando-se apenas no conjunto de treinamento pode levar a um dos problemas mais fundamentais no aprendizado de máquina, o chamado sobreajuste (*overfitting*). O sobreajuste ocorre em um modelo de aprendizagem quando o mesmo se ajusta muito bem ao conjunto usado para seu treinamento, mas se mostra ineficaz para prever novos resultados (GOODFELLOW; BENGIO; COURVILLE, 2016). Uma das maneiras de minimizar este problema, é empregar algum mecanismo que varie os conjuntos de dados de entrada, permitindo a criação de vários modelos de classificação. Neste trabalho é empregada a técnica de validação cruzada (REFAEILZADEH; TANG; LIU, 2009).

Neste contexto, este trabalho final de graduação envolve a análise da aplicação do aprendizado profundo para classificação de dados. O objetivo principal é a implementação de um algoritmo para definição dos hiperparâmetros mais adequados para uma rede neural profunda de acordo com o conjunto de dados aplicado.

O modelo selecionado para estudo foi a rede neural profunda do tipo *Deep Feed-forward Network*, associada a um algoritmo automático de ajuste dos hiperparâmetros. Os experimentos foram realizados empregando três conjuntos de dados retirados do Repositório *UC Irvine Machine Learning*[1], com diferentes valores de dimensão: Iris, *Cardiotocography* e *Musk*.

Os resultados alcançados são comparáveis àqueles disponíveis na literatura, com relação à acurácia obtida como resultado da classificação de cada conjunto de dados: Rehman e Nawi (2011) (Conjunto de dados Iris), Huang e Hsu (2012) (*Cardiotocography Dataset*) e Zhou e Zhang (2002) (*Musk Dataset*). Além disso, foi possível analisar o desempenho da busca aleatória como algoritmo automático de ajuste de hiperparâmetros, em relação a taxa de acerto obtida pela classificação.

Este trabalho está estruturado da seguinte maneira:

1. No Capítulo 2 são apresentadas definição e características de aprendizado profundo, conceitos relacionados a aprendizagem de máquina, o processo de classificação de dados, as Redes Neurais Artificiais, sua natureza e um paralelo entre RNA e Aprendizado Profundo.

2. O Capítulo 3 aborda as técnicas de aprendizado profundo para classificação de dados

[1] https://archive.ics.uci.edu/ml/

numéricos, incluindo definições e exemplos dos métodos de Aprendizado Profundo empregados neste trabalho e a ferramenta utilizada.

3. A implementação realizada está descrita no Capítulo 4.

4. O Capítulo 5 apresenta os resultados obtidos e a análise destes resultados a partir dos conjuntos de teste.

5. Por fim, o Capítulo 6 traz a conclusão e os trabalhos futuros.

# 2 Métodos Tradicionais de Classificação e Redes Neurais Artificiais Convencionais

Para Mitchell et al. (1997) um programa de computador é dito para aprender com a experiência $E$ em relação a alguma tarefa $T$ e alguma medida de desempenho $P$ se o desempenho em $T$, conforme medido por $P$, melhora com a experiência $E$. Por exemplo, suponhamos que proveja-se a um algoritmo de aprendizagem muitos dados meteorológicos históricos, e o mesmo tenha aprendido a prever o clima. Então:

- $T$ = A tarefa de previsão do tempo.
- $P$ = A probabilidade de prever corretamente o clima de uma data futura.
- $E$ = O processo do algoritmo que examina uma grande quantidade de dados meteorológicos históricos.

O aprendizado de máquina permitiu que os computadores resolvessem problemas envolvendo o conhecimento do mundo real e tomassem decisões que parecessem subjetivas. Um algoritmo de aprendizado simples, por exemplo, pode separar os *e-mails* legítimos de e-mails de *spam* (GOODFELLOW; BENGIO; COURVILLE, 2016).

No Aprendizado de Máquina, em vez de ensinar ao computador uma lista enorme de regras para resolver um determinado problema, é fornecido a ele um modelo com o qual é possível avaliar exemplos e um pequeno conjunto de instruções para modificar o modelo quando cometer um erro. Assim é esperado que, ao longo do tempo, um modelo bem adequado seja capaz de resolver o problema com extrema precisão (BUDUMA; LOCASCIO, 2017).

## 2.1 Problema de classificação

A classificação é um dos tópicos centrais na aprendizagem de máquinas e corresponde com o ensino de máquinas para prever rótulos de classes categóricas (discretos ou nominais) que classifica os dados (constrói um modelo) com base no conjunto de treinamento e os valores (rótulos de classe) em um atributo de classificação e o utiliza na classificação de novos dados (HAN; PEI; KAMBER, 2011).

Para Han, Pei e Kamber (2011), no aprendizado de máquina e em estatística, a classificação é uma abordagem de aprendizado supervisionado, na qual o programa de computador aprende a partir da entrada de dados fornecidos a ele e, em seguida,

usa-se esse aprendizado para classificar novos dados. Este conjunto de dados pode ser simplesmente possuir duas classes, como identificar que o *e-mail* é *spam* ou *não-spam*, ou pode possuir muitas classes (DUDA; HART; STORK, 2012). Existe uma versão não supervisionada da classificação, chamada de agrupamento ou *cluster* onde os computadores encontram características compartilhadas para agrupar dados quando as categorias não são especificadas (DUDA; HART; STORK, 2012)

Ainda para Duda, Hart e Stork (2012) embora a classificação na aprendizagem por máquina requer o uso de algoritmos, por vezes, complexos, a classificação é algo que os humanos fazem naturalmente todos os dias.

Segundo Han, Pei e Kamber (2011) o processo de classificação ocorre em duas etapas:

- **Construção do modelo:** Dado um conjunto de classes predeterminadas, cada tupla/amostra é assumida como pertencente a uma classe predefinida, conforme determinado pelo atributo de rótulo de classe. O conjunto de tuplas usado para a construção do modelo é um conjunto de treinamento.
- **Uso do modelo:** Para classificar objetos futuros ou desconhecidos. O rótulo conhecido da amostra de teste é comparado com o resultado classificado do modelo, assim é possível encontrar precisão do modelo (acurácia), além de calcular a taxa de precisão, que é a porcentagem de amostras de conjuntos de teste que são classificadas corretamente pelo modelo. Se a precisão for aceitável, use o modelo para classificar novos dados.

## 2.2 Modelos Lineares

Os Modelos de AM podem ser matematicamente definidos como uma função $h(x, \theta)$, onde a entrada $x$ é um exemplo expresso em forma vetorial e $\theta$ é um vetor de parâmetros empregado pelo modelo. O algoritmo de aprendizado de máquina busca aperfeiçoar os valores dos parâmetros à medida que são apresentados mais exemplos ao modelo (GOODFELLOW; BENGIO; COURVILLE, 2016). O aprendizado de máquina é baseado em técnicas algorítmicas para minimizar o erro nesta função ($h(x, \theta)$) por meio da otimização (PATTERSON; GIBSON, 2017).

Este modelo descreve, de forma geométrica, um classificador linear, o modelo Linear, e pretende-se alterar o vetor de parâmetros $\theta$ de tal forma que o modelo faça as previsões corretas dado um exemplo de entrada $x$. Este modelo é chamado de *Linear Perceptron*, e é um modelo apresentado por McCulloch e Pitts (1943).

Um modelo linear usa uma única soma ponderada de características para fazer uma previsão. Por exemplo, se houver dados sobre idade, anos de educação e horas semanais

de trabalho para uma população (representados por $x$), um modelo poderá aprender pesos (vetor $\theta$ de parâmetros) para cada um desses números, de modo que sua soma ponderada calcule o salário de uma pessoa. Os modelos lineares também são usados também para classificação (CHENG et al., 2016).

Alguns modelos lineares transformam a soma ponderada em um formato mais conveniente. Por exemplo, a Regressão Logística conecta a soma ponderada à Função Logística para transformar a saída em um valor entre 0 e 1 (RAO; TOUTENBURG, 1995).

Estes modelos são bastante limitados em relação ao que podem aprender. À medida que os problemas apresentam-se mais complexos, como o reconhecimento de objetos e a análise de texto, a dimensão do conjunto de dados aumenta consideravelmente, e os relacionamentos que devem ser captados tornam-se altamente não-lineares, tornando o uso de modelos lineares obsoletos (BUDUMA; LOCASCIO, 2017).

Assim, Russell e Norvig (2014) destacam que, desde 1943, têm sido desenvolvidos modelos muito mais detalhados e realistas, focando-se nas propriedades mais abstratas das Redes Neurais, tais como sua capacidade de realizar computação distribuída, de tolerar entradas com ruídos e com maior foco no processo de aprendizagem.

## 2.3 Rede Neural Artificial (RNA)

As Redes Neurais Artificiais (RNAs) são modelos computacionais distribuídos e paralelos compostos de unidades de processamento, os neurônios, densamente conectadas e que têm a propensão natural para armazenar conhecimento experimental e torná-lo disponível para o uso com base em modelos matemáticos, ou seja, tais técnicas adquirem conhecimento através da experiência (HAYKIN, 2007).

Segundo Braga, Carvalho e Ludermir (2007) uma RNA é considerada grande quando possui centenas ou milhares de unidades de processamento, já o cérebro de um mamífero pode ter muitos bilhões de neurônios. Para Haykin (2007) o cérebro é um computador altamente complexo, não linear e paralelo. Ele tem a capacidade de organizar seus constituintes estruturais, conhecidos por neurônios, de forma a realizar certos processamentos. Ainda para Haykin (2007) as RNAs foram baseadas no cérebro (sistema nervoso) dos seres vivos, que é formado por um conjunto extremamente complexo de neurônios.

Os neurônios são formados pelos dendritos, que são um conjunto de terminais de entrada, pelo corpo da célula e núcleo, e pelos axônios que são longos terminais de saída (BRAGA; CARVALHO; LUDERMIR, 2007). A figura 2.3.1 mostra uma representação de como são os componentes básicos de um neurônio biológico.

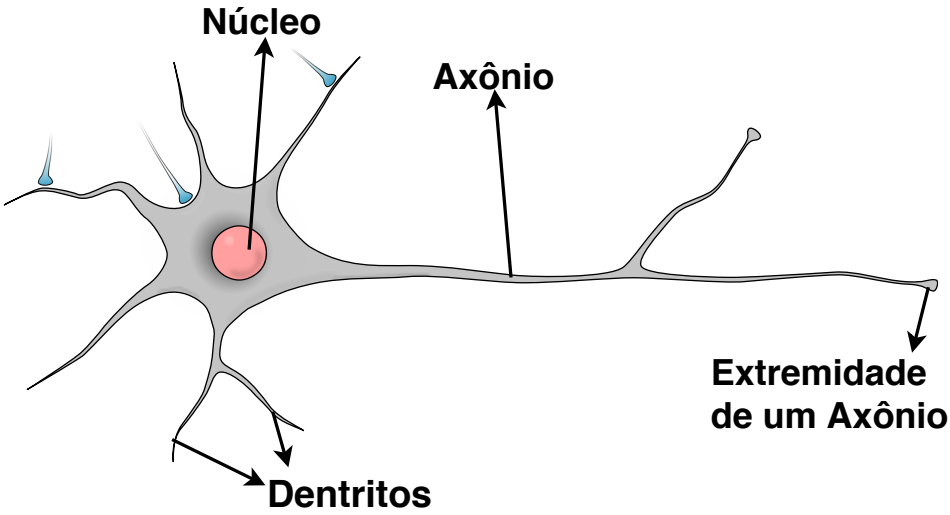



Figura 2.3.1 – Neurônio biológico. Adaptado: (BRAGA; CARVALHO; LUDERMIR, 2007)

Os neurônios se comunicam através de sinapses, definidas como unidades estruturais e funcionais elementares que intermediam as interações entre os neurônios (HAYKIN, 2007). Os impulsos recebidos por um neurônio $A$, em um determinado momento, são processados, e quando atingido um dado limiar de ação, o neurônio $A$ dispara, produzindo uma substância neurotransmissora que flui do corpo celular para o axônio, que pode estar conectado a um dendrito de outro neurônio $B$, este processo depende de vários fatores, como a geometria da sinapse e o tipo de neurotransmissor (BRAGA; CARVALHO; LUDERMIR, 2007). Em média, cada neurônio forma entre mil e dez mil sinapses e o cérebro humano possui cerca de $10^{11}$ neurônios, e o número de sinapses é de mais de $10^{14}$, possibilitando a formação de redes muito complexas (HAYKIN, 2007).

### 2.3.1 Características de uma RNA

Segundo Braga, Carvalho e Ludermir (2007) Redes Neurais Artificiais são as simulações de inspiração biológica realizadas no computador para suceder determinadas tarefas específicas, como agrupamento, classificação, reconhecimento de padrões etc.

Para Haykin (2007) as RNAs se assemelham ao cérebro humano de duas maneiras:

- Uma rede neural adquire conhecimento através da aprendizagem.
- O conhecimento de uma rede neural é armazenado dentro das forças de conexão entre neurônios conhecidas como pesos sinápticos.

É possível traduzir essa compreensão funcional dos neurônios cerebrais em um modelo artificial que pode ser representado computacionalmente (BUDUMA; LOCASCIO, 2017). Esse modelo é descrito na Figura 2.3.2:

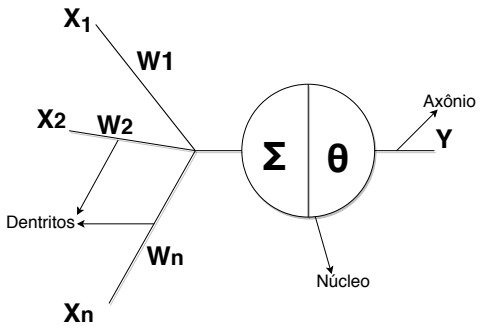



Figura 2.3.2 – Modelo Simples do Neurônio Artificial. Adaptado: (BRAGA; CARVALHO; LUDERMIR, 2007)

A estrutura de um neurônio é formada por um conjunto de conexões com outros neurônios (sinapses) que podem ser conexões de entradas (dendritos), onde há um peso ($w$) associado a cada conexão, ou saída (axônio), onde o peso da saída é dado por uma função de transferência (ativação) (RUSSELL; NORVIG, 2014).

O mecanismo de funcionamento de um neurônio é simples, onde é feita a soma dos sinais de entrada ponderados pelos pesos ($w$) das respectivas sinapses do neurônio que decide se deve ou não disparar (saída igual a 1 ou a 0) comparando a soma obtida ao limiar do neurônio (BRAGA; CARVALHO; LUDERMIR, 2007). A Equação 2.1 mostra matematicamente o mecanismo de funcionamento de um neurônio.

$$\sum_{i=1}^{n}(x_i.w_i) > \theta \tag{2.1}$$

Onde $n$ é o número de entradas do neurônio, $w_i$ é o peso associado à entrada $x_i$ e $\theta$ é o limiar do neurônio.

Outra maneira de gerar a saida do neurônio é aplicar a soma dos sinais de entrada ponderados pelos pesos ($w$) das respectivas sinapses a uma função $f$, chamada de Função de Ativação (CARVALHO et al., 2011). Segundo Haykin (2007) a Função de Ativação restringe a amplitude da saída de um neurônio. A Função de Ativação é também referida como Função Restritiva já que restringe (limita) o intervalo permissível de amplitude do sinal de saída a um valor finito. Existem várias funções que podem gerar o valor de saída do neurônio (BRAGA; CARVALHO; LUDERMIR, 2007). A figura 2.3.3 apresenta alguns exemplos.

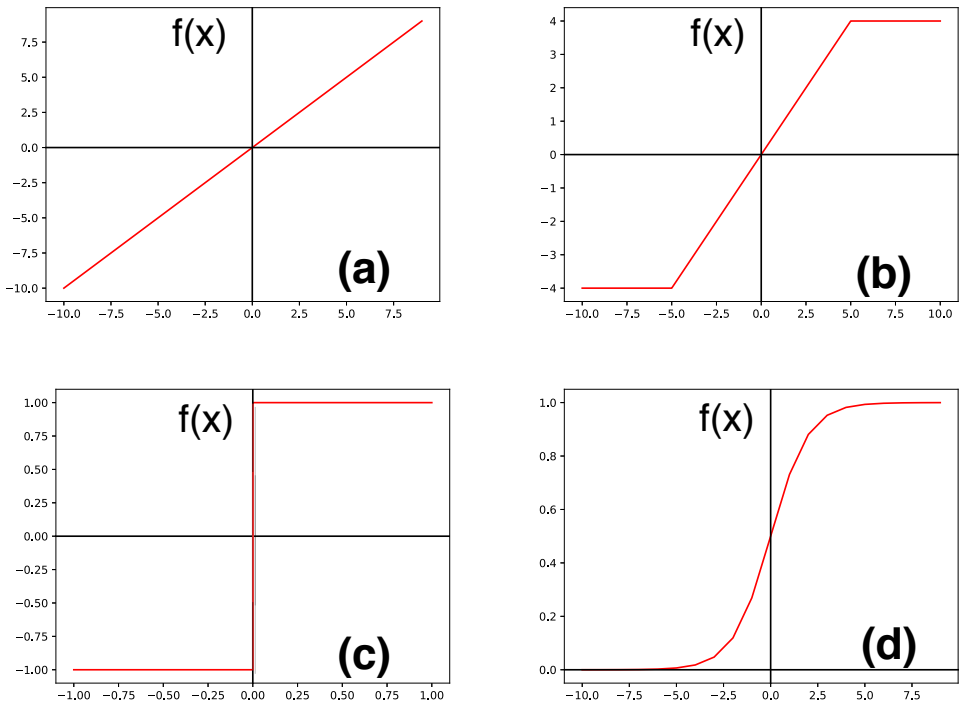



Figura 2.3.3 – Exemplos de Funções de Ativação. Adaptado: (BRAGA; CARVALHO; LUDERMIR, 2007)

A função linear (2.2) produz uma saída linear contínua, a função rampa (2.3) é usada quando se deseja restringir a função linear para produzir valores constantes em uma faixa, a função de escada (2.4) gera uma saída binária (não linear discreta) e a função sigmoide fornece uma saída não linear contínua (2.5).

$$f(x) = \alpha.x \tag{2.2}$$

$$f(x) = \begin{cases} +\lambda, & se \quad x \geq +\lambda \\ x, & se \quad |x| < +\lambda \\ -\lambda, & se \quad |x| \leq -\lambda \end{cases} \tag{2.3}$$

$$f(x) = \begin{cases} +1, & se \quad x \geq 0 \\ -1, & se \quad x \leq 0 \end{cases} \tag{2.4}$$

$$f(x) = \frac{1}{1 + e^{-1}} \tag{2.5}$$

$$f(x) = max(x, 0) \tag{2.6}$$

Atualmente, a função não linear mais popular é a unidade linear retificada (ReLU) representada pela Equação 2.6. Nas décadas passadas, as redes neurais usavam não

linearidades mais suaves, como a função 2.5, mas a ReLU tende a aprender mais rápido em redes com muitas camadas (LECUN; RANZATO, 2013).

A função de ativação *SoftMax* é uma generalização da função sigmoide e é mais adequada para problemas de classificação de várias classes. Se houver $k$ classes de saída e o vetor de peso para a classe $i$ é $w^{(i)}$, então a probabilidade prevista para a i-ésima classe dado o vetor de entrada $x \in IR^{n\times 1}$, é dada pela equação 2.7 (HOPE; RESHEFF; LIEDER, 2017).

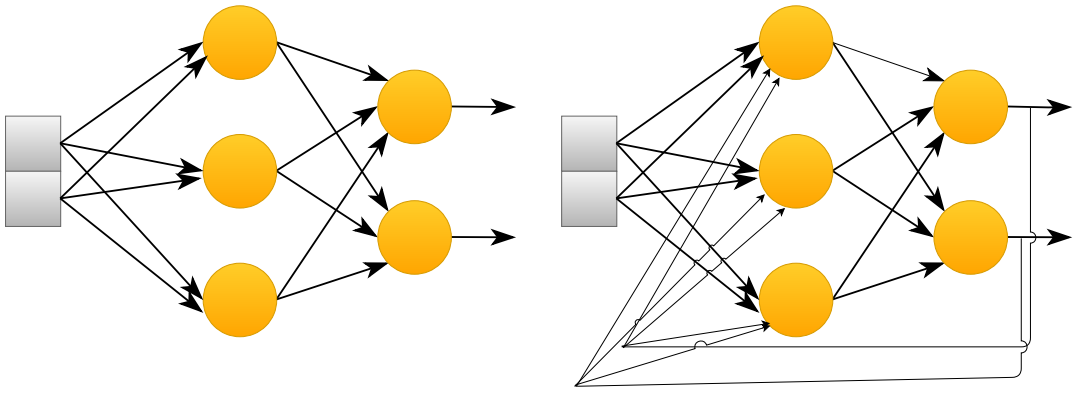
$$p(y_i = 1 \mid x) = \frac{e^{x^t w_i}}{\sum_{k=1}^{k} e^{x^t w_k}} \tag{2.7}$$

Para Braga, Carvalho e Ludermir (2007) a maioria dos modelos de Redes Neurais possui alguma regra de treinamento, onde os pesos de suas conexões são ajustados de acordo com os padrões apresentados, ou seja, aprendem através de exemplos.

Segundo Haykin (2007) a maneira pela qual os neurônios de uma rede neural estão estruturados está intimamente ligada com o algoritmo de aprendizagem usado para treinar a rede. O processo de aprendizagem será abordado na Seção 2.3.4.

### 2.3.2 Arquiteturas de RNA

A arquitetura de uma rede neural artificial é determinada por uma estrutura topológica, ou seja, a conectividade e a Função de Ativação de cada nó (neurônio) da rede (HAYKIN, 2007). Em geral, podem ser identificadas duas classes de arquiteturas de Redes Neurais, as redes com duas ou mais camadas de alimentação para frente, chamadas *feedforward*, e as redes recorrentes (RUSSELL; NORVIG, 2014). A figura 2.3.4 apresenta uma representação gráfica dos tipos de redes.



(a) Rede *feedforward* (b) Rede r*ecorrente*

Figura 2.3.4 – (a) Exemplo de RNAs feedforward e (b) Recorrente. Adaptado: (RUSSELL; NORVIG, 2014)

No caso das redes de propagação para frente (*feedforward*) o fluxo de informação é unidirecional. Neurônios que recebem a informação simultaneamente agrupam-se em camadas. Camadas que não estão ligadas às entradas e nem às saídas da rede chamam-se camadas escondidas e essa arquitetura representa uma função de sua entrada atual, portanto, não tem estado interno que não sejam os próprios pesos (RUSSELL; NORVIG, 2014).

A rede recorrente, por outro lado, alimenta suas saídas de volta às suas próprias entradas. Para Russell e Norvig (2014) isso significa que os níveis de ativação da rede formam um sistema dinâmico que pode atingir um estado estável ou apresentar oscilações ou até mesmo um comportamento caótico.

Além disso, a resposta de uma rede recorrente para determinada entrada depende do seu estado inicial, que pode depender de entradas anteriores, logo, as redes recorrentes diferente das redes com alimentação para a frente, podem suportar memória de curto prazo (RUSSELL; NORVIG, 2014).

### 2.3.3 A Álgebra das RNAs

Da mesma forma que é possível representar um neurônio, pode-se também expressar matematicamente uma RNA como uma série de operações vetoriais e matriciais (GOODFELLOW; BENGIO; COURVILLE, 2016). A figura 2.3.5 apresenta um exemplo simples de uma rede neural *feedforward* com três camadas (entrada, uma oculta e saída) e três neurônios por camada.

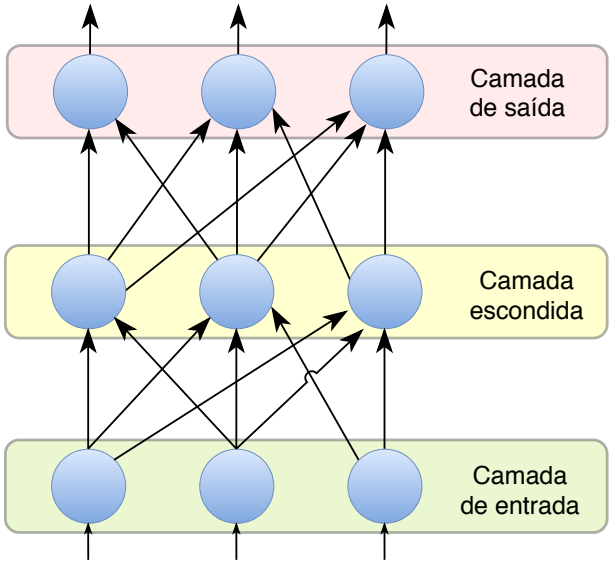



Figura 2.3.5 – Exemplo de Rede Neural de *feedforward*. Fonte: (BUDUMA; LOCASCIO, 2017)

Considerando a entrada para a camada $i$ da rede como um vetor $x = [x_1, x_2, ..., x_n]$.

É necessário encontrar o vetor $y = [y_1, y_2, ..., y_m]$ produzido pela propagação da entrada através dos neurônios. Pode-se também montar uma matriz $w$ de pesos de dimensão $n \times m$. Nesta matriz, cada coluna corresponde a um neurônio, onde o elemento da coluna corresponde ao peso da conexão. Além de um vetor $b$ contendo os bias (polarização) que é necessária para mover o limite (também conhecido como limite de decisão) para cima ou para baixo conforme necessário pela Função de Ativação. Seu valor é sempre 1, para que sua influência no resultado possa ser controlada pelo seu peso (HAYKIN, 2007). A equação 2.8 expressa a saída de uma camada de uma RNA:

$$y = f(w^{(t)}x + b) \tag{2.8}$$

Multiplica-se os dados de entrada $x$ pela matriz $w$ de pesos, adicionando os bias após a multiplicação. Isso é uma típica transformação linear, na qual usa-se operações de multiplicação e adição. Isso computa a soma ponderada de cada neurônio da camada $t$. Este valor é a entrada de alguma função não linear $f$, como a ReLU (função 2.6) para produzir a saída (GOODFELLOW; BENGIO; COURVILLE, 2016).

### 2.3.4 Processos de Aprendizado para RNA

Para Haykin (2007) a propriedade que é de importância primordial para uma rede neural é a sua habilidade de aprender a partir de seu ambiente e de melhorar o seu desempenho através da aprendizagem. Ocorre com o tempo de acordo com alguma medida preestabelecida. Uma rede neural aprende acerca do seu ambiente através de um processo iterativo de ajustes aplicados a seus pesos (RUSSELL; NORVIG, 2014).

Geralmente o processo de aprendizagem significa que os parâmetros que a rede dispõe, para solucionar a tarefa em consideração, têm que ser adaptados de uma maneira ótima. Normalmente, basta modificar os pesos $w_{ij}$ entre o neurônio $i$ e o neurônio $j$ , segundo um algoritmo de aprendizagem (BRAGA; CARVALHO; LUDERMIR, 2007). Independente do algoritmo de aprendizado é usado um conjunto finito $k$ de $n$ exemplos de treino para adaptar os pesos durante a fase de treinamento da rede (HAYKIN, 2007).

Segundo Carvalho et al. (2011), diversos autores propuseram algoritmos de treinamento para RNAs seguindo os paradigmas de aprendizado supervisionado, não supervisionado e por reforço.

Aprendizado Supervisionado: Conhecido como aprendizado supervisionado, pois, a entrada e saída desejadas para a rede são fornecidas por um supervisor (professor) externo. Este método de aprendizado é o mais comum no treinamento das RNAs e tem como objetivo ajustar os parâmetros da rede, de forma a encontrar uma ligação entre os pares de entrada e saída fornecidos (BRAGA; CARVALHO; LUDERMIR, 2007). A figura 2.3.6 mostra um fluxograma do aprendizado supervisionado.

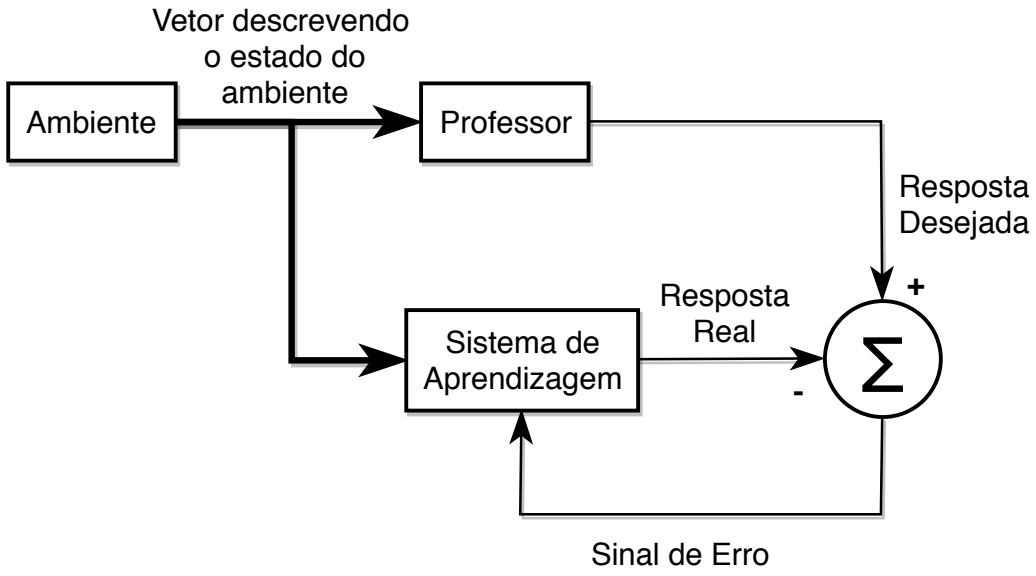



Figura 2.3.6 – fluxograma do aprendizado supervisionado. Adaptado: (HAYKIN, 2007)

O professor indica um comportamento bom ou ruim para a rede visando direcionar o processo de treinamento. O aprendizado supervisionado pode ser de duas formas: *Off-line*, os dados do conjunto de treinamento não mudam uma vez encontrada a solução, esta não muda e *On-line*, os dados mudam forçando a rede a estar em treinamento contínuo (BRAGA; CARVALHO; LUDERMIR, 2007).

Aprendizado Não Supervisionado: Neste caso, não há professor para acompanhar o processo de treinamento, por exemplo, nos seres humanos os estágios iniciais da visão e da audição ocorrem com aprendizado não supervisionado. Nos algoritmos não supervisionados somente os padrões de entrada estão disponíveis para a rede. Só é possível quando existe redundância nos dados de entrada e RNAs desse tipo podem ter ligações entre os neurônios da mesma camada (BRAGA; CARVALHO; LUDERMIR, 2007).

Reforço: Aprendizagem por reforço é um caso especial da aprendizagem supervisionada onde a saída desejada exata é desconhecida, e um crítico externo avalia a resposta fornecida pela rede (BRAGA; CARVALHO; LUDERMIR, 2007).

### 2.3.5 Algoritmos de Aprendizado para RNA

A essência de um algoritmo de aprendizado é a regra de aprendizado, isto é, uma regra de atualização dos pesos que determina como os pesos das conexões entre os neurônios são modificados. Exemplos de regras de aprendizado populares incluem a regra delta, a regra de Hebb, a regra anti-Hebb e a regra de aprendizagem competitiva (CARVALHO et al., 2011).

Basicamente essas regras de aprendizagem, também chamadas otimizadores, visam maximizar o desempenho de um modelo de aprendizado de máquina ajustando iterati-

vamente seus parâmetros até que o erro seja minimizado, existem algumas maneiras de tornar esse processo ainda mais eficiente. (BUDUMA; LOCASCIO, 2017).

Para Haykin (2007) outro ponto que merece ser analisado é a frequência de atualização dos pesos. Esta frequência influencia o desempenho obtido durante o treinamento. Segundo Buduma e Locascio (2017) existem duas abordagens diferentes:

1. Por padrão (*on-line*), nesta os pesos são atualizados após a apresentação de cada padrão de treinamento.

2. Por cliclo (*batch*), geralmente mais precisa, nesta os pesos são atualizados após um número $n$ de padrões terem sido apresentados. Esta técnica geralmente é mais estável.

#### 2.3.5.1 O Algoritmo de *Backpropagation*

Para Carvalho et al. (2011) o algoritmo de *Backpropagation* é o mais utilizado no treino de redes do tipo *feedforward*. Permite uma maneira de modificar os pesos $w_{ij}$ de modo que, sendo-lhe apresentado um padrão $\xi$, ela tenha como saída um padrão $\zeta$ pretendido (RUSSELL; NORVIG, 2014).

Segundo Haykin (2007) processo de treino de uma *feedforward* utilizando este algoritmo realiza-se em dois passos distintos:

- ***Forward***: Num primeiro passo um padrão $\xi$ é apresentado às entradas da rede dando-se a sua propagação ao longo das sinapses entre os neurônios das várias camadas até ser gerado um padrão $\zeta$ nas unidades de saída.

- ***Backward***: No segundo passo o padrão de saída $\zeta$ é comparado com padrão $\tau$ pretendido como resposta. A diferença entre os padrões (erro) é propagada para trás através da rede, modificando-se os pesos $w_{ij}$ das ligações, à medida que se dá a propagação do erro.

O algoritmo define os erros dos nodos (neurônios) das camadas intermediárias, possibilitando o ajuste de seus pesos. A função do erro a ser minimizada (otimizada) é definida pela soma dos erros quadráticos, conforme a equação 2.9 (HAYKIN, 2007).

$$E = 1/2 \sum_{p} \sum_{i=1}^{k} (d_i^p - y_i^p)^2 \tag{2.9}$$

Onde $E$ é a medida de erro total, $p$ o número de padrões ($\xi$ e $\zeta$), $k$ igual ao número de unidades de saída, $d_i$ a i-ésima saída desejada e $y_i$ igual a i-ésima saída gerada pela rede.

O algoritmo *backpropagation* procura minimizar o erro obtido pela rede ajustando pesos e limiares para que eles correspondam às coordenadas dos pontos mais baixos da superfície de erro. Para isso usa o método do Gradiente Descendente calculando a derivada parcial de E em função dos pesos (OLAH, 2015)

O gradiente de uma função é uma qualidade que a leva na direção e sentido em que a função tem taxa de variação máxima. Isso garante que a rede caminha na superfície na direção que vai reduzir mais o erro obtido. Em problemas simples, este método encontra o erro mínimo. Em problemas mais complexos, esta garantia não existe podendo levar o algoritmo a convergir para mínimos locais. *Backpropagation* fornece uma aproximação da trajetória no espaço de pesos calculado pelo método do gradiente descendente. Estes pontos podem incluir platôs ou mínimos locais (HAYKIN, 2007).

As etapas do método de gradiente descendente são as seguintes (OLAH, 2015):

1. Inicialização aleatória dos pesos
2. Cálculo da saída prevista de uma rede neural (etapa *Forward*)
3. Cálculo da função de custo/perda
4. Cálculo do gradiente da função de perda. Para a maioria das arquiteturas RNAs, o método mais comum é o *Backpropagation*
5. Atualização de peso com base no peso atual e no gradiente da função de perda (fase *Backward*)
6. Repita os passos 2 a 5, até a função perda, atinga um certo limite ou após uma certa quantidade de iteração

O procedimento de *Backpropagation* para calcular o gradiente de uma função objetivo em relação aos pesos $w_{ij}$ em múltiplas camadas pode ser interpretado como uma aplicação prática da regra da cadeia para derivadas. O principal argumento é que a derivada (ou gradiente) do objetivo em relação à entrada pode ser calculada usando a fase *Backward* a partir do gradiente em relação à saída (LECUN; BENGIO; HINTON, 2015).

#### 2.3.5.2 Dificuldade no treinamento

O algoritmo *backpropagation* oferece uma série de dificuldades ou deficiências. O principal problema refere-se a lentidão do algoritmo para problemas mais complexos (OLAH, 2015). Uma forma de minimizar esse problema é considerar efeitos de segunda ordem para o gradiente descendente. Não é raro o algoritmo convergir para mínimos locais (BRAGA; CARVALHO; LUDERMIR, 2007). Haykin (2007) sugere algumas técnicas para

acelerar o algoritmo e evitar mínimos locais: utilizar taxa de aprendizado decrescente, adicionar nós intermediários, utilizar um termo *Momentum* e Adicionar ruído aos dados.

Para Goodfellow, Bengio e Courville (2016) o Momentum é normalmente utilizado para acelerar o processo de treinamento da rede e evitar mínimos locais. Sua utilização é influenciada por ser ela uma técnica simples e efetiva, além de eliminar oscilações nos pesos.

Segundo Haykin (2007) outro problema conhecido é o *Overfitting* ou excesso de treinamento, e ocorre quando, a partir de um certo ciclo de treinamento, a rede, em vez de melhorar, começa a piorar a sua taxa de acertos para padrões diferentes daqueles utilizados para os ajustes dos pesos (treinamento).

Goodfellow, Bengio e Courville (2016) sugerem algumas alternativas como: encerrar o treinamento mais cedo, quando o erro de validação começa a subir, "podar" os pesos da rede, eliminando neurônios indesejáveis, que são aqueles com valores muito próximos de zero, nas camadas escondidas, processo conhecido como *Dropout*.

# 3 Aprendizado Profundo para classificação de Dados

Este capítulo descreve os métodos de Aprendizagem Profunda e as tecnologias utilizadas neste trabalho. O modelo de classificação utilizado é a Rede Neural Profunda e para seleção dos hiperparâmetros é empregada uma Busca Aleatória. O *Frameworks* de Aprendizado Profundo *Tensorflow*, selecionado é também detalhado neste Capítulo.

A hierarquia de conceitos permite que o computador aprenda conceitos complicados, construindo-os a partir de conceitos mais simples. Se esboçar um grafo mostrando como esses conceitos são construídos uns sobre os outros, o grafo se torna profundo, com muitas camadas. Por essa razão, chama-se essa abordagem de Aprendizagem Profunda em Inteligência Artificial (GOODFELLOW; BENGIO; COURVILLE, 2016). Este capítulo tem como objetivo detalhar o contexto de Aprendizagem Profunda e Redes Neurais Artificiais.

## 3.1 Aprendizado Profundo

O Aprendizado Profundo é uma abordagem da IA, especificamente, é um tipo de AM, uma técnica que permite que os sistemas de computadores melhorem com experiência e dados. O aprendizado de máquina é a única abordagem viável para construir sistemas de IA que podem operar em ambientes complicados do mundo real. A aprendizagem profunda é um tipo particular de aprendizado de máquina que alcança grande poder e flexibilidade aprendendo a representar o mundo como uma hierarquia aninhada de conceitos, com cada conceito definido em relação a conceitos mais simples e representações mais abstratas computadas em termos menos abstratos (GOODFELLOW; BENGIO; COURVILLE, 2016). A figura 3.1.1 ilustra um diagrama de *Venn* com as relações entre essas áreas de IA, e mostra como a Aprendizagem Profunda é um tipo de Aprendizado de Representação, que é, por sua vez, uma área de Aprendizado de Máquina.

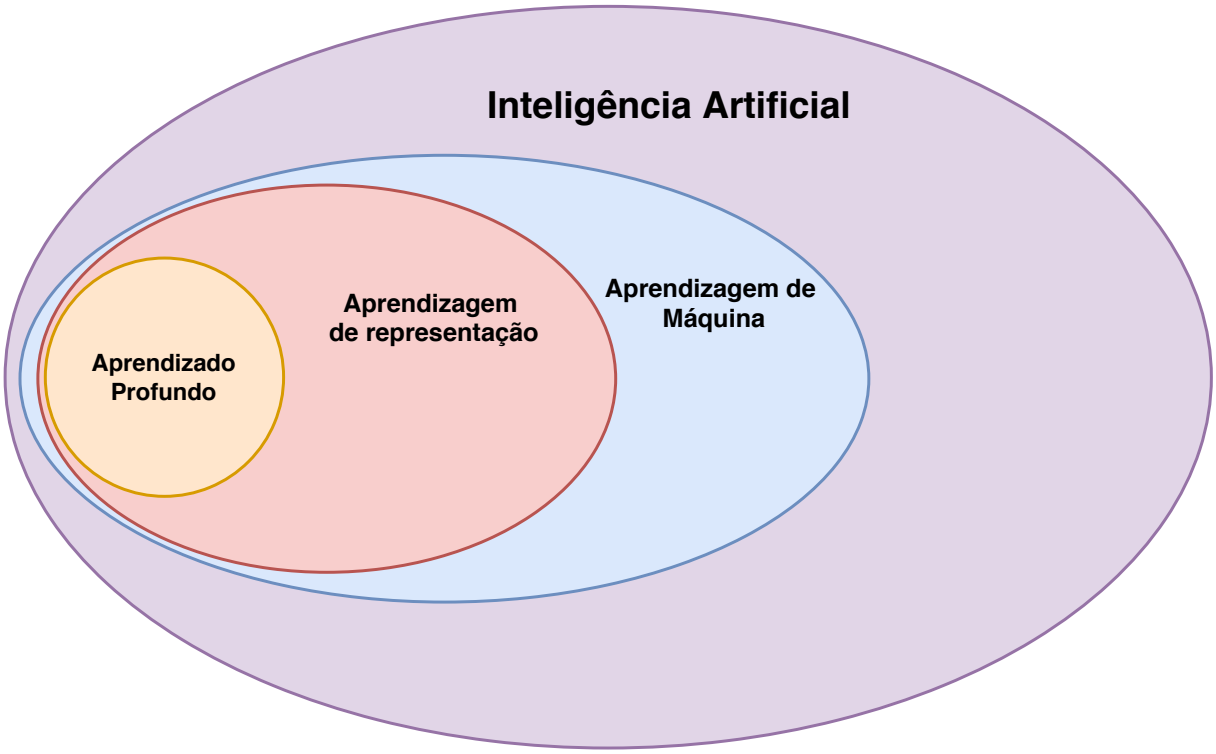



Figura 3.1.1 – Diagrama de *Venn* áreas de IA em relação ao Aprendizado Profundo. Adaptado: (GOODFELLOW; BENGIO; COURVILLE, 2016)

O Aprendizado de Representação visa usar o aprendizado de máquina para descobrir não apenas o mapeamento da representação para a saída, mas também a representação dos dados em si. Naturalmente, pode ser muito difícil extrair recursos abstratos de alto nível a partir de dados brutos. Muitos desses fatores de variação, podem ser identificados apenas com o entendimento sofisticado dos dados. Quando é quase tão difícil obter uma representação quanto resolver o problema original, o Aprendizagem de Representação se torna inviável (GOODFELLOW; BENGIO; COURVILLE, 2016).

Para LeCun, Bengio e Hinton (2015) o Aprendizado Profundo pode solucionar este problema central na Aprendizagem de Representação, introduzindo representações que são expressas em termos de outras representações mais simples. Os algoritmos de Aprendizagem Profunda são considerados uma promissora área de pesquisa na extração automatizada de representações de dados complexas (recursos) em altos níveis de abstração. Tais algoritmos desenvolvem uma arquitetura hierárquica em camadas de aprendizagem e representação de dados, onde os recursos de nível superior (mais abstrato) são definidos em termos dos recursos de níveis inferiores (menos abstratos) (LECUN; BENGIO; HINTON, 2015).

A arquitetura de aprendizagem hierárquica dos algoritmos de Aprendizagem Profunda é motivada por inteligência artificial simulando o processo de Aprendizagem Profundo e em camadas das áreas sensoriais primárias do neocórtex no cérebro humano, que extrai automaticamente recursos e abstrações dos dados subjacentes (GOODFELLOW; BENGIO;

COURVILLE, 2016).

Para Deng, Yu et al. (2014) a Aprendizagem Profunda pode ser aplicada a qualquer tipo de dado: sinais digitais, áudio, vídeo, fala, palavras e escritas, com isso produz conclusões que parecem ter sido alcançadas por humanos, porém muito mais rápido.

Embora o aprendizado profundo tenha mostrado resultados impressionantes em muitas aplicações, a fase de treinamento não é uma tarefa trivial para massivas quantidades de dados devido ao fato de que os cálculos iterativos inerentes à maioria dos algoritmos de Aprendizagem Profunda geralmente são extremamente difíceis de serem paralelizados. Assim, com o crescimento sem precedentes de conjuntos de dados comerciais e acadêmicos nos últimos anos, há um interesse crescente em algoritmos paralelos efetivos e escaláveis para treinamento de modelos profundos (LIN; KOLCZ, 2012).

### 3.1.1 Algumas Abordagens Algorítmicas para Aprendizado Profundo

Segundo Chen e Lin (2014), o aprendizado profundo em larga escala geralmente envolve tanto grandes volumes de dados como grandes modelos (Modelos Profundos). Algumas abordagens algorítmicas foram exploradas para a aprendizagem em larga escala.

Existem diversos trabalhos sobre o assunto, pode-se destacar Deng, Yu e Platt (2012) propuseram uma arquitetura profunda modificada chamada *Deep Stacking Network* (DSN), que pode ser efetivamente paralelizada. Um DSN consiste em várias Redes Neurais especializadas (denominadas módulos) com uma única camada oculta. Módulos empilhados com entradas compostas por vetor de dados brutos e a saída do módulo anterior forma um DSN. Além de Hutchinson, Deng e Yu (2013) que desenvolveram uma arquitetura profunda chamada *Tensor Deep Stacking Network* (T-DSN), que é baseada no DSN, é implementada usando *clusters* de CPU para computação paralela escalável.

Quando o volume de dados é menor, os métodos tradicionais tendem a ter um desempenho melhor do que os de aprendizagem profunda. No entanto, quando o volume de dados aumenta significativamente, o método de aprendizagem profunda ganha sobre os métodos tradicionais por uma margem enorme (HOPE; RESHEFF; LIEDER, 2017).

## 3.2 RNA e Aprendizado Profundo

O estudo da Aprendizagem Profunda não converge somente no processo de aprendizagem, mas também, no desenvolvimento de Redes Neurais denominadas Redes Neurais Profundas, as quais possuem uma complexidade necessária para lidar com a classificação de conjuntos de dados (SCHMIDHUBER, 2014).

Segundo Nielsen (2015) Redes Neurais e o Aprendizado Profundo atualmente fornecem as melhores soluções para muitos problemas no reconhecimento de imagens,

reconhecimento de fala, processamento de linguagem natural e classificação. Nielsen (2015) Classifica RNA e Aprendizado Profundo da seguinte maneira:

- Redes Neurais, um paradigma de programação biologicamente inspirado que permite que um computador aprenda com dados de observação.
- Aprendizado Profundo, um poderoso conjunto de técnicas de aprendizagem junto como desenvolvimento de redes profundas.

Basicamente as redes de Aprendizagem Profunda se distinguem das redes tradicionais por sua profundidade, ou seja, o número de camadas, através das quais os dados passam em um processo de reconhecimento de padrões em vários passos (SCHMIDHUBER, 2014). Com a implementação dos métodos de Aprendizagem Profunda, representando as Redes Neurais profundas, os novos paradigmas de aprendizagem resolveram uma série de problemas que limitavam a expansão e a implementação bem sucedida das Redes Neurais tradicionais (SCHMIDHUBER, 2014).

Segundo Deng, Yu et al. (2014) em redes de Aprendizagem Profunda, cada camada treina em um conjunto distinto de características com base na saída da camada anterior. Quanto mais se avança para a rede neural, mais complexas as características que seus nós podem reconhecer, pois agregam e recombinam recursos da camada anterior.

Este comportamento é conhecido como hierarquia de recursos, que é uma hierarquia de crescente complexidade e abstração. Com isso as redes de Aprendizagem Profunda são capazes de lidar com conjuntos de dados muito grandes e de alta dimensão com vários parâmetros (LECUN; BENGIO; HINTON, 2015).

Assim essas redes são capazes de descobrir padrões, tanto em dados estruturados quanto não estruturados. Portanto, um dos problemas que a Aprendizagem Profunda possui melhor facilidade é o processamento e classificação de dados, discernindo semelhanças e anomalias presentes no conjunto (SCHMIDHUBER, 2014).

Os algoritmos de Aprendizagem Profunda envolvem a otimização em muitos contextos. De todos os problemas de otimização envolvidos no Aprendizado Profundo o treinamento de Redes Neurais é o mais importante, pois este problema pode ser considerado como gargalo das redes profundas. Assim, mecanismos especializados em técnicas de otimização foram desenvolvido para solucioná-lo (GOODFELLOW; BENGIO; COURVILLE, 2016).

Para Schmidhuber (2014) arquiteturas de Aprendizagem Profunda são basicamente Redes Neurais Artificiais de múltiplas camadas não-lineares. Vários tipos foram propostos de acordo com as características dos dados de entrada e os objetivos da pesquisa. Min, Lee e Yoon (2017), categoriza as arquiteturas de Aprendizagem Profunda em quatro grupos:

- Redes Neurais Profundas/ *Deep Neural Networks* (DNNs),
- Redes Neurais Convolucional/ *Convolutional Neural Networks* (CNNs).
- Redes Neurais Recorrentes/ *Recurrent Neural Networks* (RNNs).
- Arquiteturas Emergentes.

Para Min, Lee e Yoon (2017) alguns trabalhos usaram "DNNs" para abrangir todas as arquiteturas de Aprendizagem Profunda, no entretanto, por convenção, usa-se "DNNs" para se referir a arquiteturas que usam redes de propagação para frente profunda, também chamadas de *Deep Feedforward Networks* (GOODFELLOW; BENGIO; COURVILLE, 2016).

O exemplo mais importante de um Modelo de Aprendizagem Profunda é a Rede Profunda *Feedforward* ou *Perceptron* Multicamadas (*Multilayer Perceptron*-MLP). A MLP é apenas uma função matemática que mapeia alguns conjuntos de valores de entrada para valores de saída. A função é formada pela composição de muitas funções mais simples. Pode-se relacionar cada aplicação de uma função matemática diferente como fornecendo uma nova representação da entrada (GOODFELLOW; BENGIO; COURVILLE, 2016).

As CNNs são arquiteturas que tiveram sucesso particularmente no reconhecimento de imagens e consistem em camadas de convolução, camadas não-lineares e camadas de agrupamento (LECUN; BENGIO; HINTON, 2015).

Os RNNs são projetados para utilizar informações sequenciais de dados de entrada com conexões cíclicas, usa-se unidades conhecidas como memórias de longo prazo ou *Gated Recurrent Unit* (GRUs) (GOODFELLOW; BENGIO; COURVILLE, 2016).

Além disso, muitas outras arquiteturas emergentes de Aprendizado Profundo foram sugeridas, como Redes Neurais recorrentes multidimensionais (*Multidimensional Recurrent Neural Networks*, MDRNNs) e auto-codificadores convolucionais (*convolutional auto-encoder*, CAEs) (MIN; LEE; YOON, 2017).

## 3.3 Redes Neurais Profundas

Redes Neurais Profundas possuem uma estrutura semelhante às Redes Neurais tradicionais vistas na sessão 2.3, mas inclui mais camadas empilhadas. É treinado de maneira puramente supervisionada que usa apenas dados rotulados. Como o método de treinamento é um processo de otimização no espaço de parâmetros de alta dimensão, estes modelos são normalmente usados quando um grande número de dados rotulados está disponível (LECUN; RANZATO, 2013).

Para Karim e Zaccone (2018), *Deep Feedforward Networks* com neurônios suficientes na camada oculta é capaz de se aproximar com precisão arbitrária e pode modelar os relacionamentos lineares e não lineares de um conjunto de dados. No entanto, não é possível determinar *a priori*, com precisão adequada, o número necessário de camadas ocultas, ou mesmo o número de neurônios que devem estar contidos dentro de cada camada oculta para computar uma função não-linear (KARIM; ZACCONE, 2018).

Se um baixo número de camadas ocultas, ou neurônios, constituem a arquitetura da rede neural, então a rede não é capaz de aproximar com precisão adequada a função desconhecida. Isso pode ocorrer porque a função é muito complexa ou porque o algoritmo de *backpropagation* está dentro de um mínimo local (ZACCONE; KARIM; MENSHAWY, 2017). Se a rede é composta de um grande número de camadas ocultas, então temos um problema de *overfitting*, ou seja, um agravamento da capacidade de generalização da rede (GOODFELLOW; BENGIO; COURVILLE, 2016).

Semelhante com as RNAs convencinais o algoritmo de *backpropagation* visa minimizar o erro entre a saída atual e a desejada. Como a rede é alimentada, o fluxo de ativação sempre avança das unidades de entrada para as unidades de saída. O gradiente da função custo é retropropagado através da modificação de pesos (GOODFELLOW; BENGIO; COURVILLE, 2016). Segundo Karim e Zaccone (2018) esse método é recursivo e pode ser aplicado a qualquer número de camadas ocultas.

Em relação as duas fases (*Forward e Backward*), a *Forward* possui o mesmo processo descrito nos capítulos anteriores. Já o *Backward* está envolvido principalmente em operações matemáticas, como criar derivadas para todas as operações diferenciáveis, isto é, métodos de diferenciação automática (OLAH, 2015). Karim e Zaccone (2018) descreve dois tipos de métodos de diferenciação automática:

- Modo inverso (*Reverse-mode*): Derivação de uma única saída em relação a todas as entradas.
- Modo de avanço (*Forward-mode*): Derivação de todas as saídas em relação a uma entrada.

Embora o aprendizado de máquina possa extrair padrões de dados, há limitações no processamento de dados brutos, que é altamente dependente de recursos projetados manualmente, ou seja, com intervenção humana. Tal processo é conhecido como *Feature Engineering* (FE), ou Engenharia de Características, em tradução livre (BRINK; RICHARDS; FETHEROLF, 2016). Para avançar da Engenharia de Características (intervenção humana) para características orientados a dados (sem intervenção humana), o aprendizado profundo tem se mostrado muito promissor. O aprendizado profundo pode descobrir características eficazes, bem como seus mapeamentos de dados para determinadas tarefas. Além disso, o

aprendizado profundo pode aprender características complexas combinando características mais simples aprendidas a partir dos dados. Em outras palavras, com Redes Neurais Artificiais de múltiplas camadas não-lineares, representações hierárquicas de dados podem ser descobertas com níveis crescentes de abstração (LECUN; RANZATO, 2013).

À medida que o número de características em um modelo linear cresce, fica cada vez mais difícil alcançar boas previsões em um processo de classificação, isso ocorre, pois conforme as características aumentam as mesmas perdem a propriedade de ser linearmente separáveis (NIELSEN, 2015). Este é um problema conhecido e uma solução particularmente eficaz é o uso de Redes Neurais Profundas (SCHMIDHUBER, 2014).

Redes Neurais Profundas são capazes de se adaptar a conjuntos de dados mais complexos e gerar melhores resultados, através de suas múltiplas camadas (LECUN; BENGIO; HINTON, 2015).

### 3.3.1 Treinamento de Redes Neurais Profundas

Como foi visto anteriormente (Seção 2.3.5) o treinamento das Redes Neurais é um processo de otimização custoso, e um conjunto especializado de técnicas de otimização foi desenvolvido para resolvê-lo. Com RNAs mais complexas e com a massividade encontrada nos dados atuais, o tempo de treinamento se tornou um gargalo nos algoritmos de Aprendizado Profundo, inclusive em RNA, já que os cálculos se tornam muito complexos (LECUN; BENGIO; HINTON, 2015).

Segundo Goodfellow, Bengio e Courville (2016), a ideia do gradiente descendente é ter uma função de custo que mostre a diferença entre as saídas previstas de alguma rede neural, com a saída real. Existem vários tipos conhecidos da função custo, como a função de erro quadrático e a função de probabilidade (*log-likelihood*) e o método de gradiente descendente otimiza o peso da rede, minimizando essa função de custo (ZACCONE; KARIM; MENSHAWY, 2017).

Ainda para Goodfellow, Bengio e Courville (2016) nos últimos anos foram desenvolvidos diversos métodos de Otimização de algortimos de treinamento para modelos profundos como: Gradiente Descendente Estocástico/ *Stochastic Gradient Descent* (SGD), *Adagrad* e *Adam*. A aplicação destes métodos proporcionam um treinamento mais rápido, já que focam na escalabilidade, viabilizando o uso de DNNs em conjuntos de dados complexo (GOODFELLOW; BENGIO; COURVILLE, 2016).

## 3.4 Selecionando Hiperparâmetros

Para Goodfellow, Bengio e Courville (2016) a maioria dos algoritmos de aprendizado de máquina possuem hiperparâmetros, configurações que pode-se usar para controlar o

comportamento do algoritmo. Em outras palavras os hiperparâmetros são as variáveis que determinam a estrutura do modelo, como por exemplo, o Número de Camadas Ocultas, Função de Ativação, etc. Além das variáveis que determinam como o mesmo é treinado, como por exemplo, Taxa de Aprendizagem, número de Épocas de treinamento. etc. Neste sentido, pode-se notar uma grande diferença entre parâmetros e hiperparâmetros de um modelo de Aprendizagem. Para Claesen e Moor (2015) um parâmetro é uma variável de configuração interna ao modelo de aprendizagem e cujo valor pode ser estimado a partir dos dados de treinamento, por exemplo, os pesos em uma RNA. Os parâmetros são estimados a partir dos dados e os hiperparâmetros podem serem usados nos processos de aprendizagem para ajudar a estimar os parâmetros do modelo (CLAESEN; MOOR, 2015).

A Tabela 3.4.1 descreve alguns hiperparâmetros que podem ser encontrados em modelos DNN.

Tabela 3.4.1 – Hiperparâmetros relacionados à estrutura DNN.

| **Hiperparâmetros** | **Características** |
|---|---|
| Número de Camadas Ocultas | Modelos computacionais compostos por múltiplas camadas de processamento aprendam a representação do dado em múltiplos níveis de abstração. |
| Número de Neurônios (unidades) | Muitas unidades ocultas dentro de uma camada com técnicas de regularização podem aumentar a precisão. Um número menor de unidades pode causar falta de adequação. |
| Taxa de Aprendizagem | Relacionado com a técnica de Otimização, a taxa de aprendizagem inadequada, seja alta ou excessiva, resulta em um modelo com baixa capacidade efetiva devido à falha de otimização. |
| Taxa *Dropout* | É provável obter um melhor desempenho quando o *Dropout* é usado, dando ao modelo mais uma oportunidade de aprender representações independentes. Geralmente, usa-se um pequeno valor de 20% -50% dos neurônios com 20%, fornecendo um bom ponto de partida. Uma probabilidade muito baixa tem efeito mínimo e um valor muito alto resulta em sub aprendizagem pela rede. |
| Função de ativação | As funções de ativação são usadas para introduzir a não-linearidade em modelos, o que permite que modelos de aprendizagem profunda aprendam limites de previsão não-lineares. Existem várias Funções de ativação, por exemplo, a Sigmoide é usado na camada de saída enquanto faz previsões binárias. O Softmax é usado na camada de saída enquanto faz previsões de várias classes. |

| | |
|---|---|
| *Momentum* | Ajuda a saber a direção do próximo passo com o conhecimento dos passos anteriores. Isso ajuda a evitar oscilações. Uma escolha típica de momento é entre 0,5 a 0,9. |
| Número de épocas | Número de épocas é o número de vezes que todos os dados de treinamento são mostrados à rede durante o treinamento. |
| Tamanho do *batch* | É o número de subamostras dadas à rede após o qual a atualização do parâmetro acontece. Um bom padrão para o tamanho do lote pode ser 32. Além disso, costuma-se a usar 32, 64, 128, 256 e assim por diante. |

Adaptado: (GOODFELLOW; BENGIO; COURVILLE, 2016)

Os valores dos hiperparâmetros não são adaptados pelo próprio modelo de aprendizado, porém é possível projetar um algoritmo aninhado ao modelo de aprendizagem que encontre os melhores hiperparâmetros (GOODFELLOW; BENGIO; COURVILLE, 2016).

### 3.4.1 Formalizando a busca de hiperparâmetros

O objetivo de muitas tarefas de aprendizado de máquina pode ser resumido como o treinamento de um modelo $\Omega$ que minimiza alguma função de perda $\rho$ predefinida $\rho(X^{(te)}, \Omega)$, aplicado a um conjunto de teste $(X^{(te)})$. Funções de perda comuns incluem erro quadrático médio e taxa de erro. O modelo $\Omega$ foi construído por um algoritmo de aprendizado $\alpha$ usando um conjunto de treinamento $X^{(tr)}$, tipicamente envolvendo a solução de algum problema de otimização. O algoritmo de aprendizagem $\alpha$ pode ser parametrizado por um conjunto de hiperparâmetros $\lambda$, isto é, $\Omega = \alpha(X^{(tr)}, \lambda)$ (CLAESEN; MOOR, 2015).

O objetivo da busca por hiperparâmetros é encontrar um conjunto de hiperparâmetros $\lambda^*$ que produza um modelo ótimo $\Omega^*$ que minimize $\rho(X^{te}, \Omega)$. Isso pode ser formalizado da seguinte forma 3.1:

$$\lambda^* = \arg\min_{\lambda} \rho(X^{(te)}, \alpha(X^{(tr)}, \lambda)) = \arg\min_{\lambda} F(\lambda, \alpha, X^{(te)}, X^{(tr)}, \rho) \tag{3.1}$$

A função objetivo $F$ seleciona uma tupla de hiperparâmetros $\lambda$ e retorna a perda associada. Os conjuntos de dados $X^{(tr)}$ e $X^{(te)}$ são fornecidos e o algoritmo de aprendizagem $\alpha$ e a função de perda $\rho$ são escolhidos. Dependendo da tarefa de aprendizado, $X^{(tr)}$ e $X^{(te)}$ podem ser rotulados e/ou iguais entre si. Na aprendizagem supervisionada, um conjunto de dados é geralmente dividido em $X^{(tr)}$ e $X^{(te)}$ usando métodos de validação cruzada (CLAESEN; MOOR, 2015).

### 3.4.2 Métodos de seleção de hiperparâmetros

Bergstra e Bengio (2012) sugerem alguns algoritmos que podem ser usados para descobrir hiperparâmetros: *Manual Search* (Busca Manual), *Grid Search* (Busca em Grade), *Random Search* (Busca Aleatória) e *Bayesian Optimization* (Otimização Bayesiana). A Busca de Grade e a Busca Manual são as estratégias mais utilizadas para a otimização de Hiperparâmetros, porém a Busca Aleatória pode ser mais eficiente para a otimização de hiperparâmetros (BERGSTRA; BENGIO, 2012). Em comparação com as Redes Neurais configuradas por uma Busca em Grade, Bergstra e Bengio (2012) concluiu que, a Busca Aleatória sobre o mesmo domínio é capaz de encontrar modelos que são tão bons ou melhores, além de ser computacionalmente mais viável. Assim, para este trabalho, se torna interessante o uso do Algoritmo de Busca Aleatória para a seleção dos hiperparâmetros para os modelos estudados.

Em qualquer método de seleção de hiperparâmetros, compara-se o modelo com base no desempenho do conjunto de validação, depois seleciona-se uma arquitetura final e, finalmente, é feita a comparação de desempenho ao conjunto de testes (BERGSTRA; BENGIO, 2012).

Entretanto, para Patterson e Gibson (2017) avaliar cada modelo apenas no conjunto de validação pode levar a um dos problemas mais fundamentais no aprendizado de máquina o *overfitting*. Mas para Bergstra e Bengio (2012) a validação cruzada é capaz de contornar este problema.

## 3.5 Validação Cruzada

Segundo Refaeilzadeh, Tang e Liu (2009) Validação Cruzada é um método estatístico de avaliação e comparação de algoritmos de Aprendizagem dividindo os dados em dois segmentos: um usado para aprender ou treinar um modelo e outro usado para validar o modelo.

Na validação cruzada típica, os conjuntos de treinamento e validação devem cruzar em execuções sucessivas, de modo que cada ponto de dados tenha uma chance de ser validado (REFAEILZADEH; TANG; LIU, 2009). A forma básica de validação cruzada é a validação cruzada em $k$ pastas ($k$-*fold*). Nesta forma, os dados de treinamento são primeiro particionados em $k$ segmentos igualmente dimensionados, ou quase igualmente. Subsequentemente, as iterações de treinamento e validação são executadas de tal forma que, dentro de cada iteração, um segmento diferente dos dados é mantido para validação, enquanto os demais $k - 1$ segmentos são usadas para treinamento Patterson e Gibson (2017). Em aprendizado de máquina, a validação cruzada de 10 segmentos ($k = 10$) é a mais comum (GÉRON, 2017).

## 3.6 Ferramentas

Trabalhos recentes indicam que o aprendizado profundo está em direção a novas aplicações em diversos domínios e setores. Para colocar em prática essas ideias de pesquisa, é necessário um *framework* de software para aprendizagem profunda (TOKUI et al., 2015).

### 3.6.1 Como os *Frameworks* de *Deep Learning* funcionam

Para Tokui et al. (2015) em *frameworks* de RNAs típicos, os modelos são construídos em duas fases, que compõem um paradigma denominado *Define-and-Run* (Definir e Executar) (Figura 3.6.1). Na fase *Define* (Definição), um grafo computacional é construído (estrutura que organiza os cálculos) e na fase *Run* (Execução), o modelo é treinado em um conjunto de dados de treinamento.

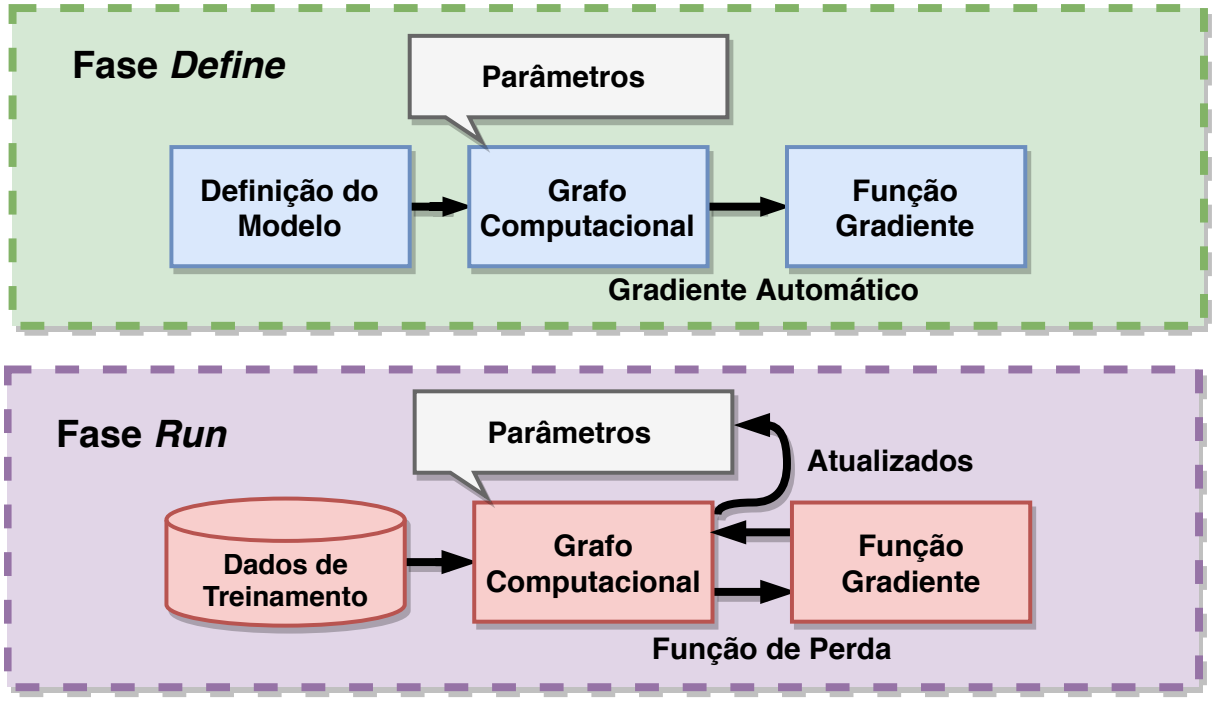



Figura 3.6.1 – Etapas do paradigma *Define-and-Run*. Adaptado: Tokui et al. (2015)

A fase *Define* consiste em instanciar um objeto de uma RNA com base em uma definição de modelo que especifica as conexões entre camadas, pesos iniciais e funções de ativação, por exemplo. Depois que o grafo é construído na memória e o cálculo do *forward* é definido, o cálculo de *backward*, correspondente ao *backpropagation*, pode ser definido pelo gradiente automático (TOKUI et al., 2015). O gradiente automático é um mecanismo que trata automaticamente da computação dos gradientes das funções definidas (OLAH, 2015). Na fase de *Run*, dado um conjunto de exemplos de treinamento, o modelo é treinado minimizando a função de perda usando algoritmos de otimização, como gradiente decrescente e ajustando os parâmetros do modelo (TOKUI et al., 2015).

Um dos problemas apresentados pelos *frameworks* projetados com base no paradigma *Define-and-Run* é sua extensibilidade limitada. Para manter a compatibilidade com versões anteriores, os desenvolvedores dos *frameworks* de aprendizado profundo não têm a liberdade de estender o esquema *Define-and-Run* para acomodar modelos mais complexos. Portanto, os usuários que desejam implementar arquiteturas de RNA originais têm duas opções: estender a estrutura ao bifurcar o repositório do *frameworks* ou impor a implementação cortando a base de código existente (TOKUI et al., 2015).

Segundo Kochura et al. (2017) na última década, inúmeras estruturas para aprendizado de máquina surgiram, principalmente as implementações de código aberto que são mais promissoras devido a várias razões: códigos-fonte disponíveis, grande comunidade de desenvolvedores e usuários finais e, consequentemente, inúmeras aplicações que demonstram e validam a maturidade dessas estruturas. A tabela 3.6.1 mostra os principais *frameworks* de código aberto.

Tabela 3.6.1 – *Frameworks* de aprendizagem profunda de código aberto

| **Framework** | **Linguagem de Programação Base** | **Multi Gpu** | **Distribuído** | **Ano** |
|---|---|---|---|---|
| Caffe | Python, C++, Matlab | Sim | Não | 2013 |
| Deeplearning4j | Java, Scala | Sim | Sim | 2014 |
| H20 | Python, R | Não | Não | 2014 |
| MXNet | Python, C++ | Sim | Sim | 2015 |
| *Tensorflow* | Python, C++ | Sim | Sim | 2015 |
| Theano | Python | Não | Não | 2010 |
| Torch | C++, Lua | Sim | Não | 2012 |

Adaptado: Géron (2017)

Quando o *Tensorflow* se tornou aberto em novembro de 2015, já havia muitas bibliotecas populares de código aberto para Aprendizagem de máquina e a maioria dos recursos de *Tensorflow* já existiam em uma biblioteca ou outra (KARIM; ZACCONE, 2018). Porém, para Abadi et al. (2016) em suma, o *Tensorflow* foi projetado para ser flexível, escalável e pronto para produção já os os *frameworks* existentes atingem apenas dois dos três destes diferenciais. Além de que segundo Géron (2017) o *Tensorflow* possui *design* limpo e possui uma ótima documentação.

### 3.6.2 *Tensorflow*

O *Tensorflow* é uma poderosa biblioteca de software de código aberto para computação numérica, particularmente adequada e ajustada para aprendizado de máquina em grande escala. Foi desenvolvido pelo Google e muitos dos seus aplicativos de AM em larga escala foram implementados usando o *Tensorflow* (ABADI et al., 2015).

Foi projetado para melhorar o desempenho otimizando a alocação de cálculos entre vários CPUs ou GPUs, além de suportar computação distribuída, para que se possa treinar RNAs profundas em conjuntos de treinamento consideravelmente grandes (*big data*) em um período de tempo razoável dividindo os cálculos em centenas de servidores (ABADI et al., 2016).

#### 3.6.2.1 Funcionamento

Para Tokui et al. (2015) o *Tensorflow* utiliza o paradigma Define-and-Run. McClure (2017) sugere um fluxo para implementação de algoritmos usando o *Framework Tensorflow*. A figura 3.6.2 apresenta o fluxograma representando, de forma genérica, como implementar algoritmos de aprendizado de máquina usando o *Tensorflow*.

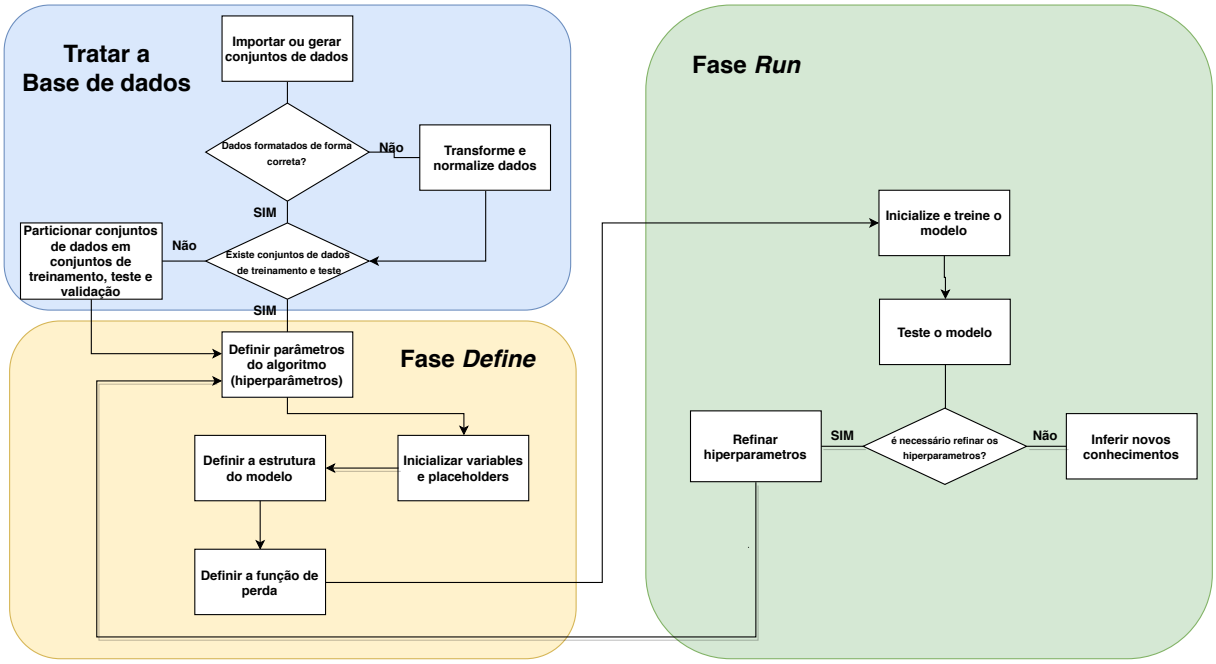



Figura 3.6.2 – Implementação de Algoritmos de AM com *Tensorflow*. Adaptado: McClure (2017)

Geralmente algoritmos de aprendizagem em máquina dependem de conjuntos de dados e casualmente podem não possuir o formato que *Tensorflow* opera, sendo necessária a transformação dos dados. Além disso, os dados geralmente não estão na dimensão ou na tipagem corretas em que os algoritmos esperam. Assim, é necessário transformar os dados antes que possam ser usados. Por exemplo, alguns algoritmos esperam dados normalizados. Além disto deve-se particionar conjuntos de dados em conjuntos de treinamento, teste e validação (MCCLURE, 2017). No fluxograma, equivale às etapas contidas no quadro azul (Tratar a base de dados).

Na maioria das vezes os modelos de AM possuem um conjunto de hiperparâmetros que define o comportamento do modelo. O *Tensorflow* precisa ter conhecimento das

variáveis que podem ser ajustadas, como por exemplo, o ajuste dos pesos de uma RNA durante a otimização para minimizar uma função de perda (erro). Para tanto, é necessário alimentar o modelo com os dados, isto é feito através dos *placeholders*. Também é primordial definir o modelo. Isso é feito através da construção de um grafo computacional, seguindo as configurações definidas através dos hiperparâmetros. Depois de definir o modelo, já é possível avaliar a saída. Neste ponto é onde declara-se a função de perda (ZACCONE; KARIM; MENSHAWY, 2017). Essas etapas são executadas na fase *Define* (contidas no quadro amarelo).

Em seguida, após a geração do modelo , cria-se uma instância do grafo (detalhado em 3.6.2.2), que é alimentado com os dados através dos *placeholders* assim permitindo que o *Tensorflow* altere as variáveis (*variables*) para prever melhor os dados de treinamento. Uma vez que o modelo foi construído e treinado, é possível avaliá-lo, observando seu desempenho quando novos dados dados são apresentados, através de alguns critérios especificados. Desta forma, é possível fazer previsões a partir de novos dados, já que o treinamento foi realizado (ABADI et al., 2015). Estas etapas são correspondentes à fase *run*, representada pelo quadro verde do fluxograma.

#### 3.6.2.2 Princípio básico

Para desenvolver modelos no *Tensorflow* define-se primeiro, em Python (ou outra linguagem), um grafo computacional de cálculos (fase *Define*) e, em seguida, o *Tensorflow* manipula esse grafo e o executa de forma eficiente usando o código C++ otimizado (fase *Run*) (ABADI et al., 2015).

Segundo Abadi et al. (2016) para compreender como o *Tensorflow* funciona, é necessário entender Tensores e Grafos. Estas são as duas estruturas básicas embutidas na estrutura de aprendizado profundo do *framework*.

Matematicamente, os tensores são funções multilineares dos espaços vetoriais aos números reais definida pela equação 3.2 (OLAH, 2015).

$$f : \underbrace{V^* \times \cdots V^*}_{p\ vezes} \times \underbrace{V \times \cdots V}_{q\ vezes} \rightarrow IR \tag{3.2}$$

Em outras palavras, para Goodfellow, Bengio e Courville (2016), tensores são generalizações de matrizes. Assim, a partir da definição de 3.2, pode-se inferir:

$$Um\ escalar\ \text{é}\ um\ tensor(f : IR \rightarrow IR,\ f(e_1) = c) \tag{3.3}$$

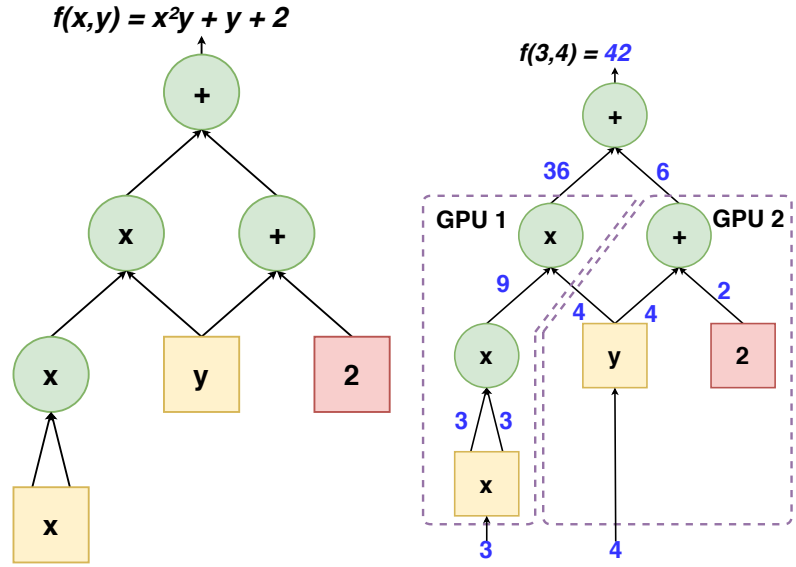
$$Um\ vetor\ \text{é}\ um\ tensor(f : IR^n \rightarrow IR,\ f(e_i) = v_i \tag{3.4}$$

$$Uma\ matriz\ é\ um\ tensor(f : IR^n \times IR^m \rightarrow IR,\ f(e_i, e_j) = v_{ij} \tag{3.5}$$

Pode-se tratar, computacionalmente, tensores como uma estrutura de dados, que são simplesmente *arrays* (vetores) ou listas com $n$ eixos (dimensões) (MCCLURE, 2017). No *Tensorflow* os tensores tem tipos estáticos e todos os elementos devem ser do mesmo tipo, porém, possui formato dinâmico. Para representar o formato do tensor, usa-se a notação entre colchetes [ ], assim, um tensor sem dimensão (um escalar) é representado pela forma [ ], um tensor de uma dimensão 1D (vetor) é representado pela forma $[k]$, um tensor 2D (matriz) é representado pela forma $[k, m]$ e assim por diante (ABADI et al., 2015).

Como um exemplo mais concreto, considere um lote de 100 imagens coloridas (RGB) e de tamanho 28x28 pixels. O tensor que armazenaria esses dados seria do formato $[n°Lote, n°Altura, n°Largura, n°Cores]$, ou, em números, $[100, 28, 28, 3]$ um tensor de 4 dimensões (ZACCONE; KARIM; MENSHAWY, 2017).

O *Tensorflow* define uma estrutura, um grafo em que os nós representam operações matemáticas e as arestas representam os tensores que fluem entre as operações, e executa cálculos envolvendo tensores que representa um modelo de aprendizagem. Também fornece vários nós de otimização avançados que encontram os parâmetros que minimizam uma função de perda (ABADI et al., 2015). A figura 3.6.3(a) mostra um exemplo de um grafo computacional responsável por resolver a equação $f(x, y) = x^2y + y + 2$. Além disso, é possível dividir o grafo em vários fragmentos e executá-los em paralelo em várias CPUs ou GPUs (conforme mostrado na Figura 3.6.3(b)) (GÉRON, 2017).




(a) Grafo Computacional simples (b) Computação paralela em várias CPUs/GPUs/servidores

Figura 3.6.3 – Exemplo de um Grafo Computacional. Adaptado: Géron (2017)

Considerando uma rede neural do tipo MLP, o *forward* será uma série de multiplicações de tensores (transformações lineares) seguidas de alguma não linearidades. Como o *Tensorflow* implementa autodiferenciação (gradiente automático ), não é necessário se preocupar com o *backward* para computar os gradientes e realizar gradiente descendente, pois isto é realizado automaticamente pela biblioteca de computação numérica (MCCLURE, 2017).

Segundo Abadi et al. (2016), o *Tensorflow* é um *Framework* para computação numérica onde os dados fluem através de um grafo, os dados são representados por *arrays N-dimensionais* chamados Tensores, e atualmente é utilizado para processamento de linguagem natural, inteligência artificial, visão computacional e análise preditiva.

#### 3.6.2.3 Escalabilidade do *Tensorflow*

Tensorflow também suporta computação distribuída, possibilitando o treinamento de redes neurais profundas e com conjuntos de treinamento gigantescos (*Big Data*) (ABADI et al., 2015). Isto em um período de tempo razoável dividindo os cálculos em centenas de núcleos, o *Tensorflow*, pode treinar uma rede com milhões de parâmetros em um conjunto de treinamento composto por bilhões de casos com milhões de recursos cada possibilitando a implementação de algoritmos de grande escala (ABADI et al., 2016).

Com o *Tensorflow*, é possível distribuir facilmente a execução de grafos em vários CPUs/GPUs e/ou várias máquinas, no desenvolvimento em alto nível o *Tensorflow* irá inserir automaticamente nós no grafo para permitir a execução distribuída do mesmo (GÉRON, 2017).

O suporte do *Tensorflow* à computação distribuída é um dos principais destaques em comparação com outros *frameworks*. Oferece controle total sobre como dividir (ou replicar) os grafos de computação entre dispositivos e servidores, e permite paralelizar e sincronizar operações de maneiras flexíveis, permitindo a escolha entre todos os tipos de abordagens de paralelização (ABADI et al., 2016).

O *Tensorflow* foi projetado com a portabilidade em mente, permitindo que os grafos computacionais sejam executados em uma ampla variedade de ambientes e plataformas de *hardware*. Com código essencialmente idêntico, o mesmo modelo implementado no *Tensorflow* pode, por exemplo, ser treinado na nuvem, distribuído em um *cluster* de várias máquinas ou em um único computador. Ele pode ser implantado para servir previsões em um servidor dedicado ou em plataformas de dispositivos móveis, como *Android* ou *iOS*, ou em sistemas embarcados (*Raspberry Pi*) (HOPE; RESHEFF; LIEDER, 2017).

#### 3.6.2.4 Componentes do *Tensorflow*

O *Tensorflow* fornece um ambiente de programação, como mostra a figura 3.6.4, o ambiente do *Framework* é dividido em várias camadas (APIs) (ABADI et al., 2015).

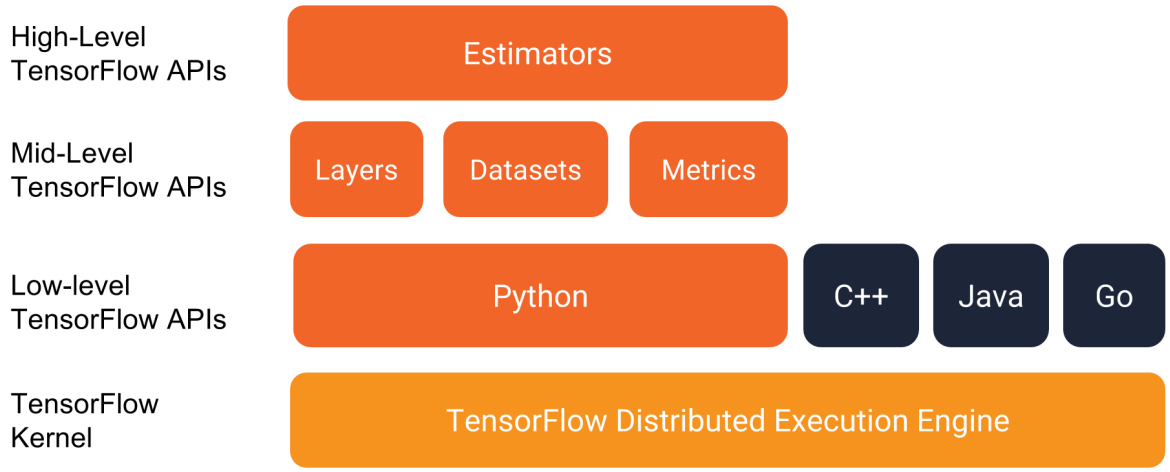



Figura 3.6.4 – ambiente de programação *Tensorflow*. Fonte. Abadi et al. (2015)

No nível mais alto (*Highy-Level*), se encontra o módulo *Estimators*, que representa um modelo completo, fornecendo métodos para treinar o modelo, julgar a precisão do modelo e gerar previsões. Os *estimators* simplificam as implementações do desenvolvimento de modelos de aprendizagem profunda. Em suma, geralmente é muito mais fácil criar modelos com *Estimators* do que com ferramentas de nível inferior do *Tensorflow*. Além de fornecerem todas questões de computação distribuída (CHENG et al., 2017).

O *Tensorflow* fornece alguns *Estimators* pré fabricados (*Premade Estimators*) que permitem concentrar os esforços em um nível conceitual muito mais alto do que as APIs básicas do *Tensorflow*. A classe *DNNClassifier*, por exemplo, é uma classe pré fabricada que treina modelos de classificação por meio de redes neurais profundas (DNNs) (CHENG et al., 2017). Fundamentalmente com a classe *DNNClassifier* é possível construir uma DNN totalmente conectada e com a função de ativação podendo ser arbitrária (sigmoide, reLu, etc) para as primeiras camadas e para última camada da função de ativação é sempre a softmax, sendo assim a saída do modelo sempre será expressa a probabilidade para um dado padrão (ZACCONE; KARIM; MENSHAWY, 2017). Por exemplo se um conjunto de dados apresentam 3 classes de padrões, a camada de saída do modelo terá três neurônios e pode ter uma saída igual a $[0.0077, 0.9923, 0.0]$, isso significa que o modelo tem 0.77% de certeza que a entrada pertence a classe 1, 99.23% de que a entrada pertence a classe 2 e 0% de pertence a classe 3.

O nível médio (*Mid-Level*) possuem três módulos. O *Datasets*, que criam um *pipeline* (catalizador) de entrada de dados, possui métodos para carregar e manipular

dados afim de alimentá-los em um modelo, o Módulo *Dataset* consiste em um conjunto de classes como pode ser visto na figura 3.6.5. *Layers* preve operações para construir camadas de RNAs. E por fim o pacote *Metrics* contém operações para avaliação de métricas e estatísticas na avaliação de modelos (ABADI et al., 2015).

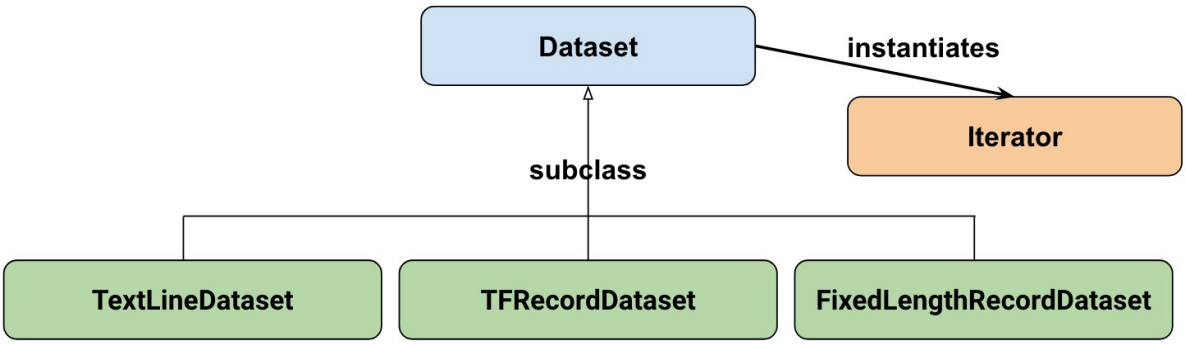



Figura 3.6.5 – Classes que constituem o Módulo *Dataset* Adaptado:(KARIM; ZACCONE, 2018)

Onde (KARIM; ZACCONE, 2018):

- *Dataset*: Classe base contendo métodos para criar e transformar conjuntos de dados. Também permite inicializar um conjunto de dados a partir de dados na memória ou de um gerador *Python*.
- *TextLineDataset*, *TFRecordDataset* e *FixedLengthRecordDataset*: Classes responsáveis pela leitura e escrita em arquivos.
- *Iterator*: Fornece uma maneira de acessar um elemento do conjunto de dados por vez.

A API de baixo nível (*low-level*) é possível criar modelos definindo uma série de operações matemáticas podendo ser implementado em várias linguagens (ABADI et al., 2015).

#### 3.6.2.5 Visualização com o *TensorBoard*

O *Tensorflow* incluiu funções para depurar e otimizar programas em uma ferramenta de visualização chamada TensorBoard (TORRES, 2016). TensorBoard é uma ferramenta de visualização completa, que permite analisar o grafo criado, resumir estatísticas de erro e ver como os parâmetros evoluem. Em se tratando de algoritmos de Aprendizado de Máquina, ver o grafo pode ser muito útil para entender o fluxo de dados no interior do modelo (ABADI et al., 2015).

Os dados exibidos com o módulo TensorBoard são gerados durante a execução do *Tensorflow* e armazenados em arquivos de rastreamento cujos dados são obtidos a partir das operações destes arquivos chamados de resumos (*Summary*) (TORRES, 2016).

Os grafos de computação para Redes Neurais Profundas podem ser bastante complexos, por exemplo, o grafo pode conter centenas de nós, devido ao tamanho e à topologia desses grafos é necessário uma forma de visualizar a organização subjacente dos grafos (ABADI et al., 2015).

Para Abadi et al. (2016) algoritmos do TensorBoard agrupam os nós em blocos de alto nível, destacando grupos com estruturas idênticas, reduzindo possíveis confusões visuais e concentra a atenção nas seções principais dos grafos computacionais.

Toda a visualização é interativa, ou seja, é possível deslocar, aplicar zoom e expandir nós agrupados para obter detalhes (ABADI et al., 2015). Um exemplo da visualização para o grafo de um modelo DNN é mostrado na figura 3.6.6:

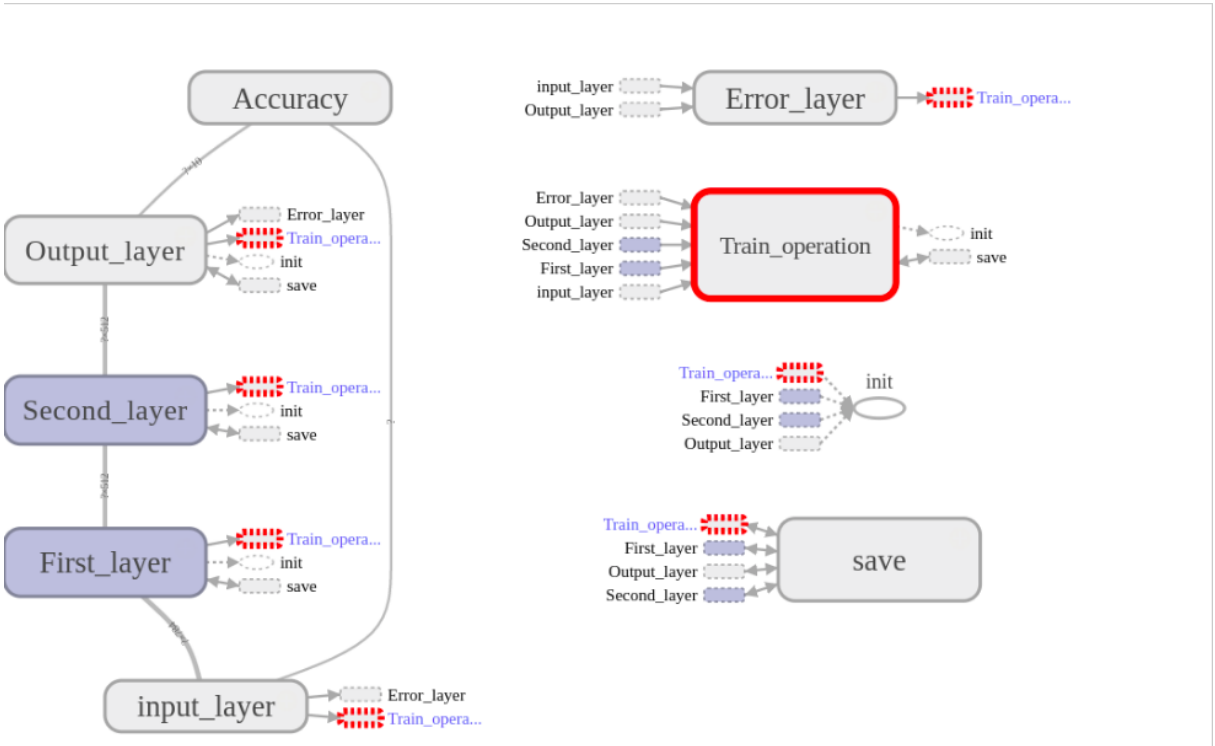



Figura 3.6.6 – Visualização de um grafo no TensorBoard de um modelo de RNA

Ao treinar modelos de aprendizado de máquina, os usuários geralmente desejam examinar o estado de vários aspectos do modelo e como esse estado muda com o tempo (ABADI et al., 2016). Para esse fim, o *Tensorflow* suporta uma coleção de diferentes operações de resumo que podem ser inseridas no grafo, por exemplo, para examinar propriedades gerais do modelo, como o valor da função de perda calculada em uma coleção de exemplos ou o tempo necessário para executar o gráfico de computação (ABADI et al., 2016). Os grafos são configurados de modo que os nós sejam incluídos para monitorar

vários valores interessantes e, de tempos em tempos, durante a execução do gráfico de treinamento, o conjunto de nós de resumo também é executado A figura 3.6.7 mostra a visualização de valores dos *Summaries* no TensorBoard.

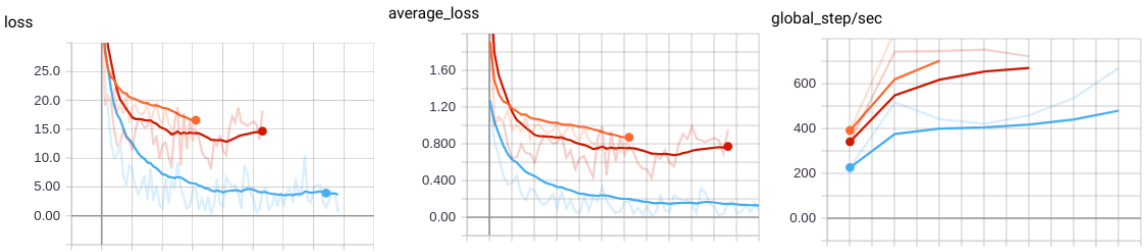



Figura 3.6.7 – Exibição gráfica do TensorBoard de algumas estatísticas

No caso são resultados gerados durante o treinamento de 3 modelos, onde o primeiro gráfico é referente a função de perda, o segundo sobre as médias das perdas e o último está relacionado a quantidade de tempo, em segundos, que cada época do treinamento levou para ser executada.

# 4 Metodologia

Para alcançar o objetivo deste trabalho, as tecnologias utilizadas foram a biblioteca de código aberto para Aprendizado de Máquina *Tensorflow*[1] e a linguagem de programação Python[2]. Como a biblioteca utilizada não implementa a funções para otimização de hiperparâmetros e técnicas de validação cruzada, houve a necessidade de implementar um algoritmo de integração. As seções que se seguem apresentam de forma detalhada todo o processo de implementação no *Tensorflow* do modelo de otimização dos hiperparâmetros de uma rede neural profunda.

Sendo assim, foi desenvolvida uma classe em Python compatível com o *Tensorflow* para encapsular as operações do modelo juntamente com a parte de otimização de hiperparâmetros. Foi empregado o modelo com aprendizado profundo DNN (*Deep Neural Network*), disponível no *Tensorflow*. A implementação realizada implementou a busca aleatória para o processo de otimização dos hiperparâmetros, com validação cruzada com $K$ pastas (vide Seção 3.5), para contornar o sobre ajuste. A figura 4.0.1 mostra, de modo simplificado, a integração do modelo implementado com o *Tensorflow* e a busca aleatória.

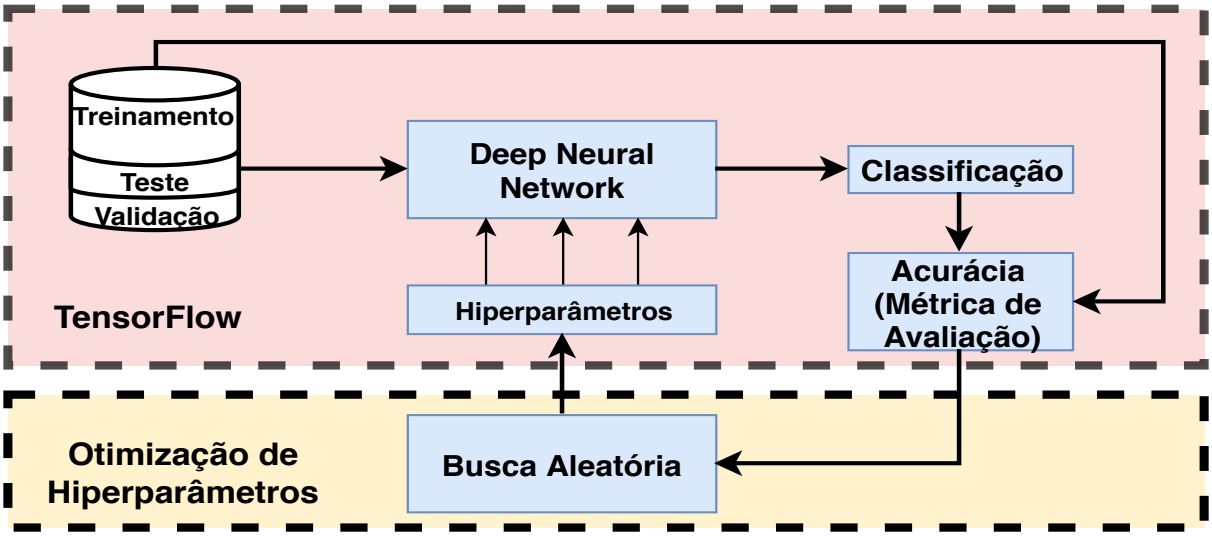



Figura 4.0.1 – Integração do Modelo e Busca Aleatória

Pode-se observar que toda implementação do modelo foi feita usando o *Tensorflow* desde a sua construção, manipulação dos conjuntos de dados, treinamento e avaliação do modelo. Estes processos se encontram dentro primeiro quadro da Figura 4.0.1 e compõem todo o processo de classificação. A otimização de hiperparâmetros utiliza a acurácia (métrica de avaliação) do classificador para alimentar a busca do melhor conjunto de parâmetros, visando sua maximização. Quanto mais próximo de 1 está a acurácia do

[1] https://www.Tensorflow.org
[2] https://www.python.org

modelo, melhor ele pode ser considerado. A busca aleatória gera os hiperparâmetros em um processo iterativo executado em um número de ciclos pré fixado.

No caso, é possível considerar o processo de classificação como uma função objetivo e deseja-se otimizar esta classificação com base em uma métrica de avaliação, que foi definida como sendo a acurácia do modelo. A otimização da função objetivo pode ocorrer através do refinamento dos hiperparâmetros ao qual o modelo é sensível. Essa Otimização ocorre através da Busca Aleatória.

## 4.1 Integração

O Algoritmo 1 descreve a implementação utilizada para o módulo de integração das fases de treinamento, refinamento dos hiperparâmetros e validação cruzada:

**Algoritmo 1** Busca Aleatória e *K-fold*

```
1: função OtimizaHiper(NumPesquisas, Modelo, Espaç)
2:     para i ← 1 até NumPesquisas faça
3:         hiperparametros = BuscaAleatória(Espaço)
4:         desempenhoValidação = 0
5:         para fold ← 1 até k faça
6:             desempenhoValidação+ =Model(fold, hiperparametros)
7:         fim para
8:         mediaModelo = desempenhoValidação/k
9:     fim para
10:    Retorne Hiperparâmetros que deram a melhor média do modelo
11: fim função
```

Como em qualquer método de busca por hiperparâmetros, é necessário comparar modelos com base no desempenho do conjunto de validação, depois escolher uma arquitetura final a partir disso e, finalmente, medir o desempenho do seu conjunto de testes. O Algoritmo 1 é responsável por comparar os modelos, usando conjunto de validação e treinamento.

Basicamente define-se uma grade de intervalos de hiperparâmetros, que em cada iteração, são selecionados aleatoriamente. Assim é possível configurar o modelo com os hiperparâmetros selecionados e em seguida é realizada a validação cruzada com $k$ pastas. São considerados três argumentos de entrada do algoritmo:

- NumPesquisas (Número de Pesquisas): Define o número de combinações geradas para o conjunto de hiperparâmetros que constituem o Modelo. Para este número foi fixado em 50, ou seja, foram feitas 50 combinações diferentes por teste.
- Modelo: Modelo de classificação de Aprendizagem Profunda, No caso o modelo estudado foi DNN (*deep neural network*)

- Espaço: É uma grade de intervalos de cada hiperparâmetro, que compõem o Modelo, uniformemente distribuídos (onde será amostrado aleatoriamente a cada iteração de NumPesquisas).

Como o Modelo utilizado foi DNN, o mesmo possui um certa gama de hiperparâmetros, que podem ser usados para a otimização da classificação. A tabela 4.1.1 mostra quais hiperparâmetros foram usados, seus intervalos e o tipo de distribuição empregada:

Tabela 4.1.1 – Conjunto de hiperparâmetros ajustados.

| **Hiperparâmetros** | **Intervalos** | Distribuição |
|---|---|---|
| Número de Camadas Ocultas | $[1 - 100]$ | Distribuídas uniformemente |
| Número de Neurônios (unidades) | $[10 - 1000]$ | Distribuídos uniformemente |
| Taxa de Aprendizagem | $[0.0001 - 0.01]$ | Distribuídos uniformemente em uma escala logaritmo |
| Taxa *Dropout* | $[0.2 - 0.5]$ | Distribuídos uniformemente |
| Função de ativação | tf.nn.relu, tf.nn.sigmoid | Distribuição discreta |
| *Momentum* | $[0.5 - 0.9]$ | Distribuído uniformemente em uma escala logaritmo |
| Número de épocas | $[1000 - 10000]$ | Distribuídos uniformemente |
| Tamanho do *batch* | $[16 - 512]$ | Distribuídos uniformemente |

Assim sendo, para hiperparâmetros que possuem efeitos multiplicativos sobre a dinâmica de treinamento é interessante usar distribuição uniforme em uma escala logarítmica, pois a probabilidade de se escolher um valor pequeno é muito maior que um valor mais alto (BERGSTRA; BENGIO, 2012). Neste caso os hiperparâmetros que possuem esta distribuição são a Taxa de Aprendizado e o *momentum*. Para demais intervalos contínuos os hiperparâmetros foram utilizados uma distribuição normal e valores discretos foram selecionados aleatoriamente.

## 4.2 Busca Aleatória

Existe uma função auxiliar chamada BuscaAleatória que é responsável por amostrar aleatoriamente os hiperparâmetros. O Algoritmo 2 apresenta o mecanismo elaborado para execução da Busca Aleatória:

**Algoritmo 2** Busca Aleatória

```
1: função BuscaAleatória(Espaço)
2:     para i ← 1 até tamanho(Espaço) faça
3:         hiperparametro[i] = Aleatório(Espaço[i])
4:     fim para
5:     Retorne hiperparametro
6: fim função
```

Para cada hiperparâmetro $i$ (tabela 4.1.1) seleciona-se de forma aleatória um valor em um intervalo uniformemente distribuído. Tal intervalo está contido no espaço de busca como parâmetro do algoritmo.

O resultado obtido pelo algoritmo de busca é empregado como hiperparâmetros do classificador, que é submetido ao treinamento considerando $k$ pastas para avaliar o modelo. Ou seja, divide-se o conjunto de treinamentos em $k$ segmentos (pastas) e, em seguida, o modelo é treinado para cada $k-1$ segmentos e avaliado para o segmento restante. O valor de $k$ pode variar de acordo com a base de dados, mas é comumente empregado um valor entre 3 e 10 segmentos (REFAEILZADEH; TANG; LIU, 2009). Para este trabalho a métrica de avaliação do modelo, é avaliada de acordo com a acurácia do teste, para cada modelo de classificação gerado. Esse processo é executado para cada segmento, ou seja, $k$ vezes. Em seguida é calculada a média das acurácias dos $k$ segmentos, considerando esta média como a métrica avaliação da função objetivo.

O modelo melhor avaliado é considerado como mais adequado e seus hiperparâmetros são retornados. Assim é possível construir o modelo a partir destes hiperparâmetros, treiná-lo com o conjunto de treinamento e analisá-lo com o conjunto de teste.

## 4.3 Implementação do Modelo

Para implementação dos modelos foram utilizadas as APIs *Highy-Level* e *Mid-Level* do *Tensorflow*. Mais especificamente foram usados os seguintes módulos:

- *Dataset*: Responsável por toda manipulação dos conjuntos de dados estudados e alimentação dos modelos implementados usando *Estimators*.
- *Estimators*: Encapsula um modelo completo, fornecendo métodos para treinar o modelo, julgar a precisão do modelo e gerar previsões.

A combinação destes módulos facilitam a implementação e alimentação de modelos de aprendizagem no *Tensorflow*, permitindo a criação um classificador com uma rede neural profunda.

Todos os conjunto de dados usados neste trabalho foram obtidos no formato de arquivo *Comma-separated-values* (CSV), onde cada linha contém $x$ valores - com $x - 1$ valores de entrada e 1 rótulo ($x$ varia de acordo com a base de dados).

A figura 4.3.1 ilustra o diagrama de classes do algoritmo implementado (integração entre o *Tensorflow*, Busca Aleatória e Validação Cruzada), ou seja, as classes que compõem o algoritmo construído com seus respectivos atributos e métodos, além de exibir como as classes se relacionam, se complementam e transmitem informações entre elas.

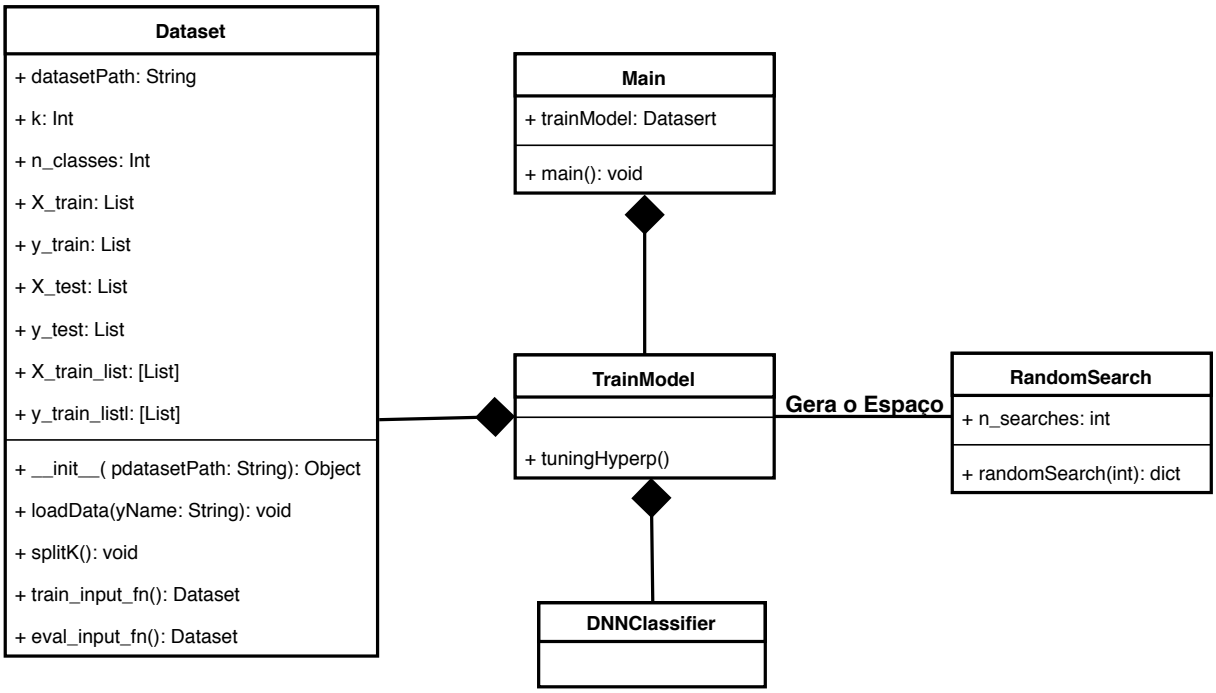



Figura 4.3.1 – Diagrama de Classes

Para tratar os conjuntos de dados foi criada uma classe (*Dataset*) responsável por encapsular os seguintes atributos:

- caminho do *csv* (*datasetPath*).
- Valor de $k$ (*k-fold*).
- Quantidade de categoria do conjunto de dados (*n_classes*).
- Conjuntos de treinamento e teste, além de uma lista que contém o conjunto dividido em $k$ segmentos.

Os métodos podem ser descritos da seguinte maneira:

- *load_data*: carrega os conjuntos de dados de um arquivo específico, definido em *datasetPath* armazenando-os nos conjuntos de treinamento e teste.

- *splitK*: divide o conjunto de treinamento em $k$ segmentos armazenando-os em uma lista.
- *train_input_fn*: função responsável por transformar os dados no formato adequado para o *Tensorflow* (alimentação dos modelos), ou seja, os conjuntos em tensores são transformados em $[q, c]$ (duas dimensões), onde $q$ é igual a quantidade de exemplos e $c$ quantidade de características.
- *eval_input_fn*: semelhante ao método anterior, porém fornece entradas para avaliação ou previsão do modelo.

A classe *TrainModel* agrega um objeto *Dataset* e um *Estimator* e usa a classe *RandomSearch* para gerar o espaço de busca e realizar a busca aleatória em cada iteração. Sendo assim o método *tuningHyperp* implementa o Algoritmo 1.

Foi utilizado um *Estimator* pré fabricado chamado *DNNClassifier*. A classe *DNNClassifier* permite construir e treinar uma rede neural profunda com qualquer número de camadas ocultas e uma camada de saída para gerar probabilidades de classe estimadas.

## 4.4 Aplicação da Metodologia

Após a implementação o algoritmo foi testado em três conjuntos de dados retirados do Repositório *UC Irvine Machine Learning Repository*, tais conjuntos se diferiam em seus formatos (dimensão e quantidade de tuplas). Para cada conjunto de dados o algoritmo implementado foi executado. Os dados coletados são as taxa de Acurácia dos modelos gerados durante a execução, os hiperparâmetros e a função de perda. Esses dados servirão de métrica para avaliação das técnicas utilizadas. Os resultados gerados estão contidos no Capítulo 5.

### 4.4.1 Ambiente de Experimentação

O algoritmo proposto foi implementado em ambiente *Linux Ubuntu 18.04 LTS (Bionic Beaver)* utilizando o *ATOM ver. 1.28.0*, para edição do código fonte, linguagem *Python ver.3.6*, e utilização das bibliotecas *Tensorflow ver. 1.7.0* (ABADI et al., 2016), *NumPy ver. 1.14.5* e *Pandas ver. 0.22.0*. Os experimentos foram realizados em um *notebook Gateway NE57007B*, com um processador *Intel®Core™i5-3337U CPU @ 1.80GHz × 4 64-bit* e 8 *GB* de memória *RAM*, executando *Linux Ubuntu 18.04 LTS (Bionic Beaver)*. O código implementado segue os algoritmos citados anteriormente.

# 5 Resultados e Análise

Este Capítulo apresenta os resultados gerados para os testes a partir de cada conjunto de dados selecionado para estudo. São descritas as principais características para cada conjunto de dados, e em seguida, são apresentados através de um conjunto de gráficos as acurácias dos modelos de classificação obtidos. São também analisados os gráficos que representam a função de perda do modelo treinado, e com a melhor configuração gerada na seleção automática de hiperparâmetros.

## 5.1 Conjuntos de dados

Para este trabalho as bases de dados estudadas foram divididas em dois conjuntos: conjunto de treinamento (80%), utilizado para o treinamento do modelo e otimização dos hiperparâmetros, conjunto de teste (20%), empregado para verificar a acurácia do modelo de classificação.

### 5.1.1 Iris *Dataset*

O conjunto de dados Iris consiste em 150 exemplos de plantas, cada uma com 4 características referentes ao comprimento da sépala (*sepal length*) e pétala (petal length), e largura da sépala (*sepal width*) e pétala (petal width), medidas em centímetros, e a classe (*class*) a qual pertence. Uma planta pode pertencer a um dos três tipos: iris setosa, iris virgínica e iris versicolor. Esta é, talvez, a base de dados mais comumente empregada na literatura sobre reconhecimento de padrões. Os padrões são igualmente divididos entre as 3 classes, 50 instâncias cada, onde cada classe se refere a um tipo de planta da íris. Uma classe é linearmente separável das outras duas, e duas não são linearmente separáveis uma da outra (DHEERU; TANISKIDOU, 2017). Como este conjunto é muito simples e rápido de ser treinado ele foi basicamente utilizado durante toda a fase de desenvolvimento, para testes e verificação da implementação realizada.

### 5.1.2 Conjunto de Dados de Cardiotocografia (*Cardiotocography Dataset*)

O conjunto de dados de cardiotocografia usado neste estudo utiliza 21 atributos, os dados podem ser classificados de acordo com a classe de padrões FHR (*fetal heart rate*/frequência cardíaca fetal) ou o código da classe de estado fetal (NSP). Neste estudo, o código de classe de estado fetal é usado como atributo alvo e cada amostra é classificada em um dos três grupos: normal, suspeito ou patológico. O conjunto de dados inclui um total de 2126 amostras, das quais 1655 pertencem à classe normal, 295 à classe suspeito e

176 amostras à classe patológica que indicam a existência de sofrimento fetal (CAMPOS et al., 2000). A tabela 5.1.1 mostra informações gerais sobre os atributos.

Tabela 5.1.1 – Atributos do conjunto de dados de cardiotocografia.

| | |
|---|---|
| LB | Linha de base (batimentos por minuto) |
| AC | Acelerações por segundo |
| FM | Movimentos fetais por segundo |
| UC | Contrações uterinas por segundo |
| DL | Desacelerações de luz por segundo |
| DS | Desacelerações severas por segundo |
| DP | Desacelerações prolongadas por segundo |
| ASTV | Porcentagem de tempo com variabilidade anormal de curto prazo |
| MSTV | Valor médio da variabilidade de curto prazo |
| ALTV | Porcentagem de tempo com variabilidade anormal a longo prazo |
| MLTV | Valor médio da variabilidade a longo prazo |
| Width | Largura do histograma da FHR (frequencia cardiaca Fetal) |
| Min | Mínimo do histograma da FHR |
| Máx | Máximo do histograma da FHR |
| Nmax | Picos do histograma |
| Nzeros | Zeros do histograma |
| Mode | Moda de histograma |
| Mean | Média do histograma |
| Median | Mediana do Histograma |
| Varuence | Variância do histograma |
| Tendency | Tendência histograma |
| CLASS | Código de classe de padrão FHR (1 a 10) |
| NSP | Código de classe de estado fetal (N = normal; S = suspeito; P = patológico). |

Adaptado: (CAMPOS et al., 2000)

As variáveis independentes desta base de dados para o problema de classificação estudado são os 21 atributos menos os atributos CLASS e NSP. O CLASS é descartado, pois representa outra classificação. A variável dependente é o Código de classe de estado fetal (NSP) que categoriza as tuplas do conjunto.

### 5.1.3 *MUSK*

Este conjunto de dados descreve um conjunto de 102 moléculas, das quais 39 são julgadas por peritos humanos como MUSK e as 63 moléculas restantes são julgadas

como não MUSK. As 166 características que descrevem essas moléculas dependem da forma exata, ou conformação da molécula. Como as ligações podem rotacionar a partir de pouca variação de energia, uma única molécula pode adotar muitas formas diferentes. Para gerar este conjunto de dados, todas as conformações de baixa energia das moléculas foram geradas para produzir 6998 conformações. Então, um vetor de características foi extraído que descreve cada conformação. É possível classificar tal base de acordo com a conformação em relação às 102 moléculas ou se a conformação é ou não MUSK (DHEERU; TANISKIDOU, 2017). O objetivo é treinar o Modelo de aprendizagem a prever se novas conformações de novas moléculas serão MUSK ou não MUSK. A tabela 5.1.2 mostra informações gerais sobre sobre os atributos.

Tabela 5.1.2 – Atributos conjunto de dados MUSK.

| | |
|---|---|
| molecule-name | Nome simbólico de cada molécula. Musks têm nomes como MUSK-188. Os não-musks têm nomes como NON-MUSK-jp13 |
| conformation-name | Nome simbólico de cada conformação. Estes têm o formato MOL-ISO + CONF. Onde MOL é o número da molécula. ISO é o número do estereoisômero (geralmente 1) e o CONF é o número de conformação. |
| f1 a f166 | São "atributos de distâncias" entre os elementos que compõem a molécula. As distâncias podem ser negativas ou positivas |

Adaptado: (DHEERU; TANISKIDOU, 2017)

As variáveis independentes desta base de dados para o problema de classificação estudado são os atributos de distâncias (f1 a f11). A variável dependente é o nome da molécula que define se é ou não Musk,

### 5.1.4 Considerações Iniciais

Para os conjuntos de dados de Cardiotocografia e Musk foram utilizados os intervalos citados no capítulo anterior (a tabela 4.1.1) e na validação cruzada foram usadas k = 10 pastas. Em relação ao conjunto de dados Iris, por sua simplicidade, principalmente se comparado com os outros modelos estudados, houve a necessidade em reduzir os intervalos dos espaços de busca e a redução da quantidade de segmentos da validação cruzada, empregando 3 pastas. A redução do espaço de busca ocorreu em quatro hiperparâmetros: Número de Camadas Ocultas $[1-5]$, Número de Neurônios $[10-100]$, a quantidade de épocas de treinamento $[200-1000]$ e o tamanho do *Batch* (lote) $[2-64]$. Os três primeiros foram necessários, pois, com valores desproporcionais em relação a base de dados, essas configurações podem prejudicar o desempenho do modelo e gerar resultados não satisfatório. Já o tamanho do lote, houve a redução pois o intervalo usado continha valores

maiores que a própria quantidade de amostras contidas no conjunto. De forma análoga também houve a necessidade em reduzir a quantidade de segmentos.

## 5.2 Resultados

### 5.2.1 Testando a implementação com o Conjunto de Dados Iris

Como mostra a Figura 5.2.1, a execução 33 apresentou a melhor média das $k$ acurácias obtidas a partir do treinamento do Modelo com a configuração dos hiperparâmetros gerados nesta execução, aproximadamente 0,9905 ou 99,05%, já a pior média foi obtida na execução 11 com apenas 0,4190 (41,90%).

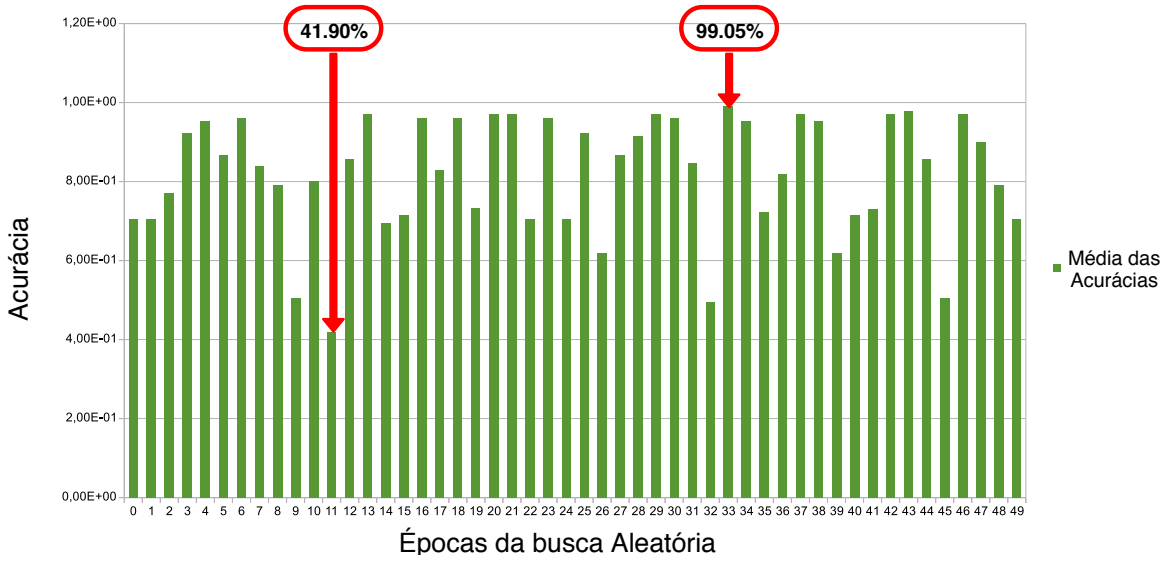



Figura 5.2.1 – Média das Acurácias do conjunto de dados Iris

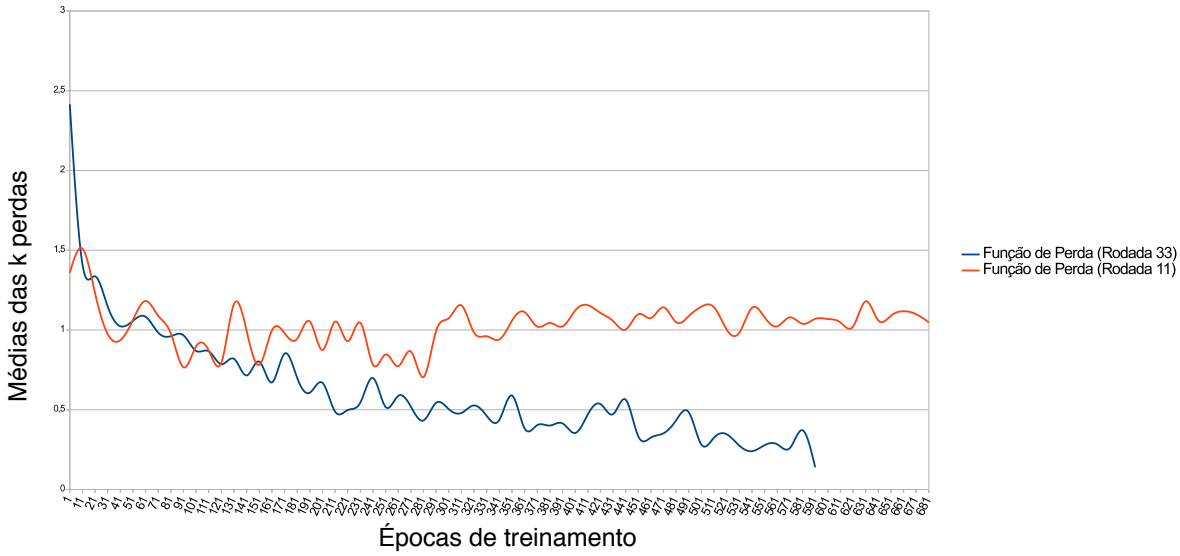



Figura 5.2.2 – Médias das Funções de Perda dos k Segmentos para as Execuções 11 e 33 (Iris)

A Figura 5.2.2 exibe a média das funções de perda nas execuções 33 e 11. Além disso a função de Perda da execução 11 variou entre 0.5 e 1.5, diferentemente da função de Perda da execução 33 que começou mais elevada porém possui um decaimento mais acentuado.

A tabela 5.2.1 exibe as configurações retornadas pelo algoritmo do pior (11) e melhor (33) modelos avaliados.

Tabela 5.2.1 – Hiperparâmetros gerados na execução 11 e 33 (Iris)

| execução | Nº Neurônios | Nº Camadas | T. Aprendizagem | Tam *batch* | T. *dropout* | Momentum | Nº épocas |
|---|---|---|---|---|---|---|---|
| 11 | 91 | 5 | 0,02024 | 6 | 0,45 | 0,6 | 684 |
| 33 | 37 | 4 | 0,00373 | 15 | 0,5 | 0,6 | 600 |

A função de ativação selecionada aleatoriamente para ambas execuções foi a ReLu. Os hiperparâmetros não possuem uma discrepância muito elevada, somente o valor da taxa de aprendizagem da execução 11 que aumentou mais de 5 vezes se comparada com com a execução 33.

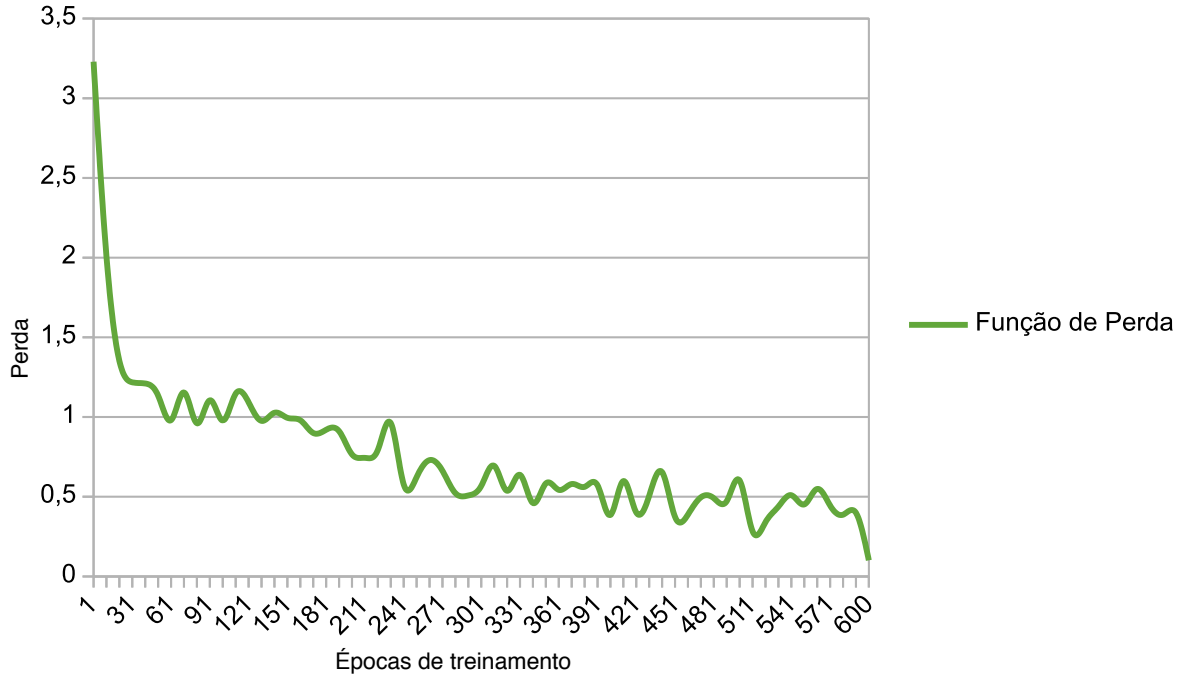



Figura 5.2.3 – Função de perda do modelo treinado e testado com os hiperparâmetros retornados (Iris)

Após a busca aleatória, o modelo (DNN) foi construído usando os hiperparâmetros da melhor execução (execução 33). A figura 5.2.3 mostra a função de perda gerada durante este novo treinamento, semelhante às médias das funções de perda da execução 33, que apresentou um decaimento significativo durante as iterações de treinamento, e também obteve um valor final levemente melhor de 0,1009 contra 0,1305. Já a acurácia foi de 98.6%, um pouco abaixo da execução 33 (99,05%).

### 5.2.2 Testando a implementação com o Conjunto de Dados de Cardiotocografia

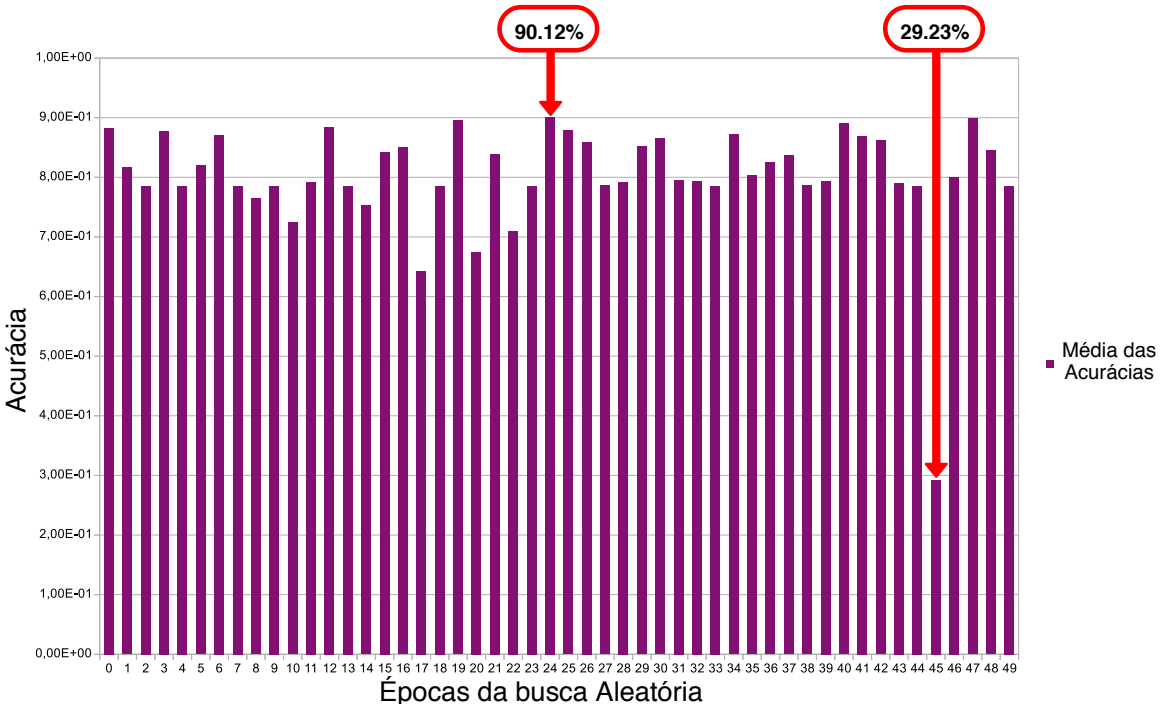



Figura 5.2.4 – Média das Acurácias do conjunto de dados Cardiotocografia

A Figura 5.2.4 exibe a média das acurácias das 50 pesquisas feitas usando o conjunto de dados Cardiotocografia. A execução 24 obteve a melhor média das $k$ acurácias obtidas a partir do treinamento do modelo com a configuração dos hiperparâmetros gerados nesta execução, aproximadamente 0,9012 ou 90,12%. Já a pior média foi obtida na execução 45 com apenas 0,2923 (29,23%).

Como mostra a Figura 5.2.5, a execução 24 (função em azul) apresentou uma perda menor no final do seu treinamento, aproximadamente 0,1879, já a execução 45 (função em laranja) apresentou uma perda mais elevada no fim de seu treinamento cerca de 0,6659. Ainda a função de perda da execução 45 variou muito pouco a partir da época 121 no treinamento, diferentemente da função de perda da execução 24 que sempre tendeu a minimizar o erro durante todo o treinamento, ou seja, obteve sempre um bom decaimento durante as épocas de treinamento.

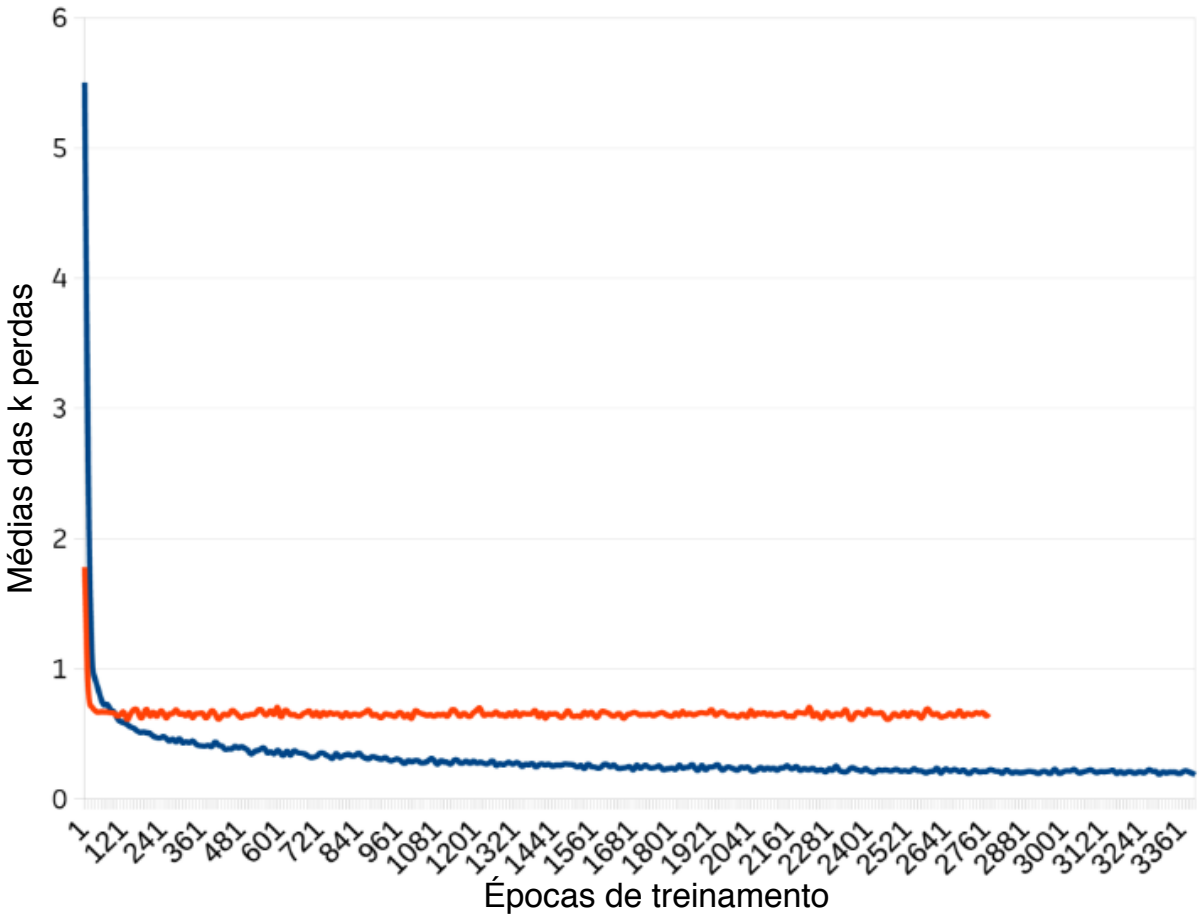



Figura 5.2.5 – Médias das Funções de Perda dos k Segmentos para as Execuções 24 e 45 (Cardiotocografia)

A tabela 5.2.2 compara as configurações retornadas pelo algoritmo do pior (45) e melhor (24) modelos durante a execução da BA.

Tabela 5.2.2 – Hiperparâmetros gerados na execução 24 e 45 (Cardiotocografia)

| Execução | Nº Neurônios | Nº Camadas | T. Aprendizagem | Tam *batch* | T. *dropout* | Momentum | Nº épocas |
|---|---|---|---|---|---|---|---|
| 45 | 16 | 4 | 0,02947 | 218 | 0,43 | 0,77 | 2778 |
| 24 | 184 | 6 | 0,00121 | 342 | 0,32 | 0,86 | 3405 |

A função de Ativação selecionada Aleatoriamente para a execução 45 execuções foi a sigmoide e a execução 24 utilizou a função ReLu. Alguns Hiperparâmetros tiveram grandes diferenças como o Número de Neurônios a execução 24 apresenta mais que 12 vezes que a execução 45, novamente a taxa de Aprendizagem na qual a melhor execução era mais de 24 vezes menor que a pior execução, além de possuir um pouco menos iterações de treinamento.

Com as configurações geradas na execução 24 o Modelo foi construído e treinado. A figura 5.2.6 mostra graficamente a função de perda gerada durante o treinamento, análogo ao gráfico das médias das funções de perda da execução 24 obteve um bom decaimento durante todas as épocas de treinamento e gerou um resultado final de 0,0763 bem abaixo

se comparado com o resultado final da execução 24 (0,1879). Em relação a Acurácia o Modelo retreinado foi levemente melhor se comparado com as médias da Acurácias geradas pelo algoritmo de otimização dos hiperparâmetros, 92% (0.920) contra 90,12%.

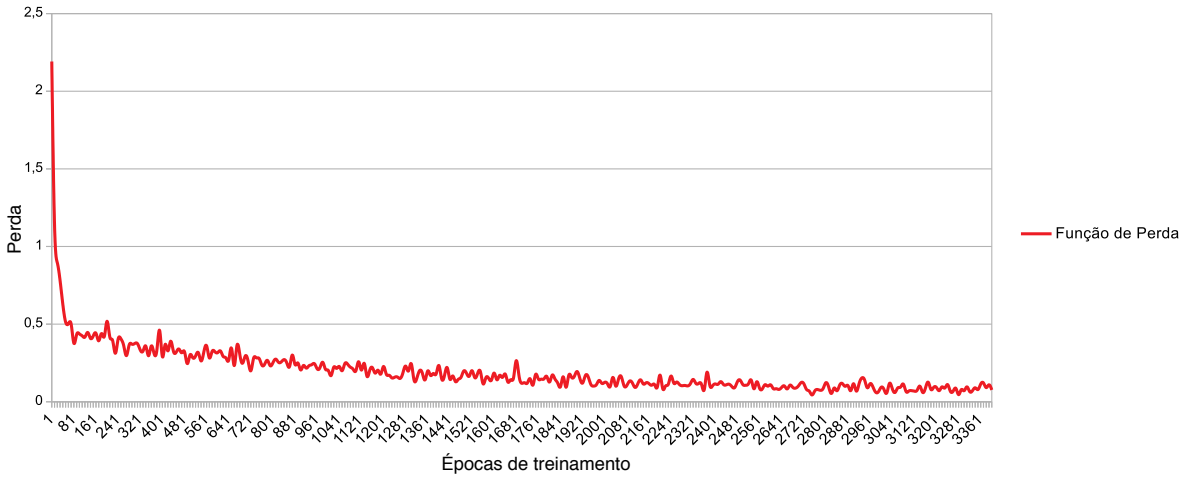



Figura 5.2.6 – Função de Perda do Modelo Treinado e Testado com os Hiperparâmetros retornados (Cardiotocografia)

### 5.2.3 Testando a implementação com o Conjunto de Dados Musk

Como mostra a Figura 5.2.7, a execução 6 apresentou a melhor média das $k$ acurácias obtidas a partir do treinamento do Modelo com a configuração dos hiperparâmetros gerados nesta execução, aproximadamente 0,9948 ou 99,48%, já a pior média foi obtida na execução 36 com 0,7020 (70,20%).

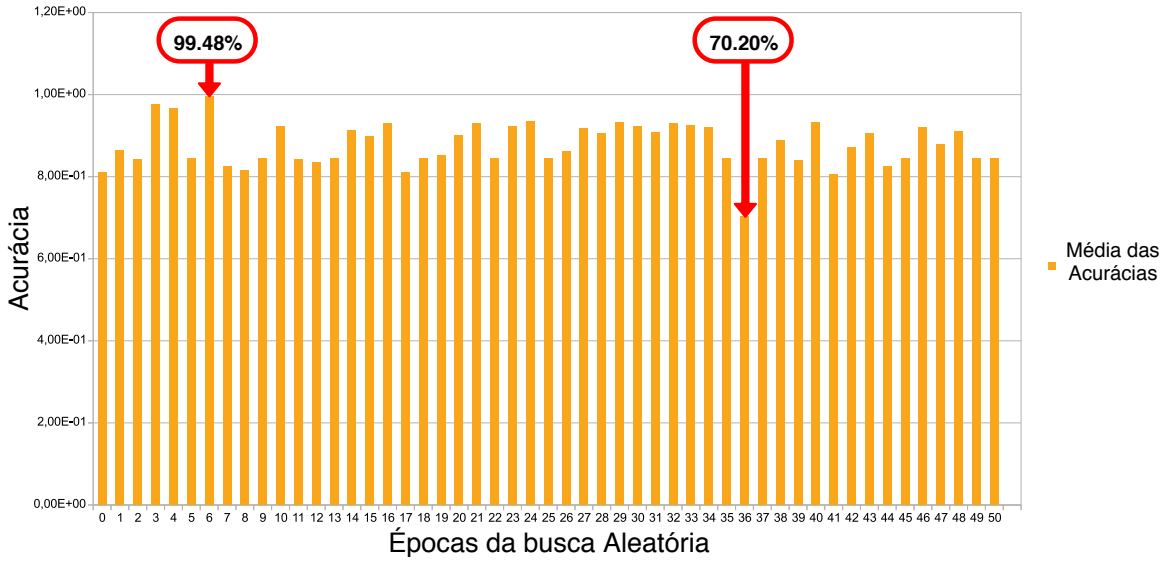



Figura 5.2.7 – Média das Acurácias do conjunto de dados Musk

Já a figura 5.2.8 expressa as médias das funções de perda das execuções 6 e 36. A execução 6 (função em azul) apresentou uma perda menor no final do seu treinamento,

aproximadamente 0,00502, já a execução 36 (função em vermelho) apresentou uma perda mais elevada no fim de seu treinamento cerca de 0,4122. Além disso, houve uma estabilidade da função de perda próximo a época 71. Outro fato é que no final a execução 6 apresentou uma média em suas perdas cerda de 82 vezes menor que as médias das funções de perda da execução 36.

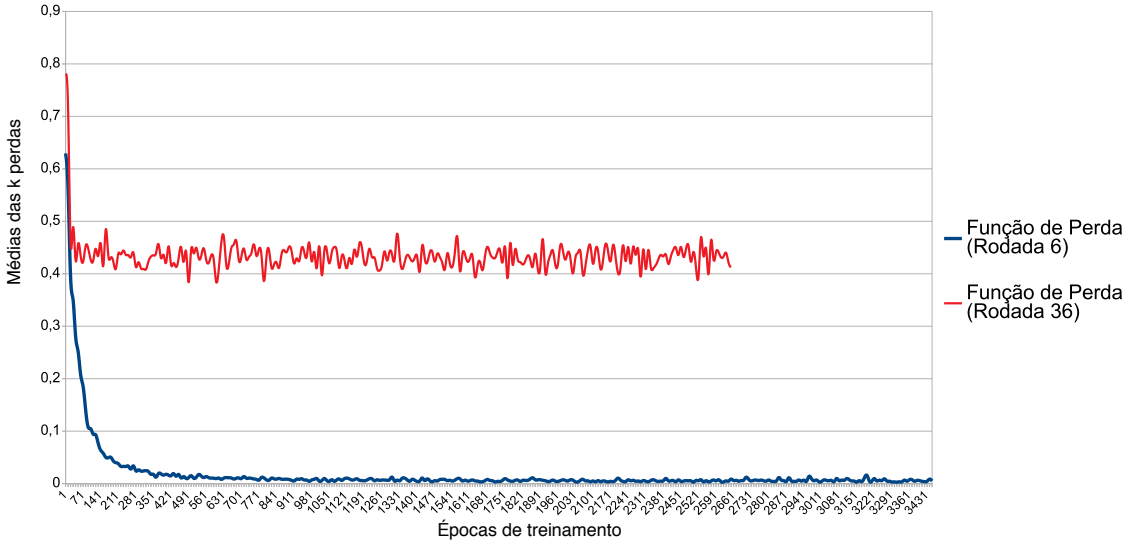



Figura 5.2.8 – Médias das Funções de Perda dos k Segmentos para as Execuções 06 e 36 (MUSK)

A tabela 5.2.3 compara as configurações retornadas pelo algoritmo do pior (45) e melhor (24) modelos durante a execução da BA.

Tabela 5.2.3 – Hiperparâmetros gerados na execução 6 e 36 (MUSK)

| **execução** | **Nº Neurônios** | **Nº Camadas** | **T. Aprendizagem** | **Tam *batch*** | **T. *dropout*** | **Momentum** | **Nº épocas** |
|---|---|---|---|---|---|---|---|
| 6 | 880 | 6 | 0,00212 | 767 | 0,25 | 0,51 | 3459 |
| 36 | 11 | 5 | 0.03907 | 111 | 0,44 | 0.87 | 2657 |

A função de ativação selecionada aleatoriamente para a execução 6 foi a reLu e a execução 36 utilizou a função Sigmoide. Alguns hiperparâmetros tiveram grandes diferenças como o Número de Neurônios a execução 6 tem 80 vezes mais neurônios que a execução 36, a taxa de aprendizagem obteve uma discrepância mais de 18 vezes entre as execuções semelhante o tamanho do *batch* que no caso foi mais de 6 vezes.

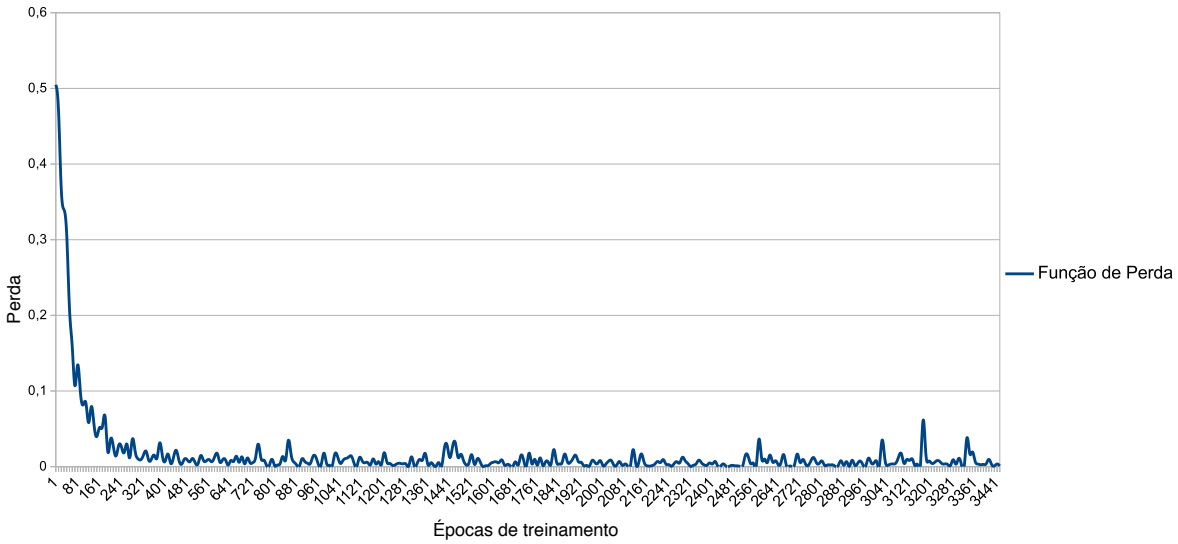



Figura 5.2.9 – Função de Perda do Modelo Treinado e Testado com os Hiperparâmetros retornados (MUSK)

A figura 5.2.9 mostra a função de perda gerada durante o treinamento do modelo com os hiperparâmetros retornados pela busca aleatória e gerou um resultado final de 0,0008747 quase seis vezes menor que o resultado final da execução 6 (0,00502). A Acurácia do Modelo retreinado e avaliado com o conjunto de teste foi de 99.94% (0.9994) contra 99,48% das médias das Acurácias obtidas pela validação cruzada durante a execução 6.

### 5.2.4 Tempos gerados durante as execuções

A figura 5.2.10 mostra as funções de tempo geradas durante a execução do algoritmo para cada base de dados testadas.

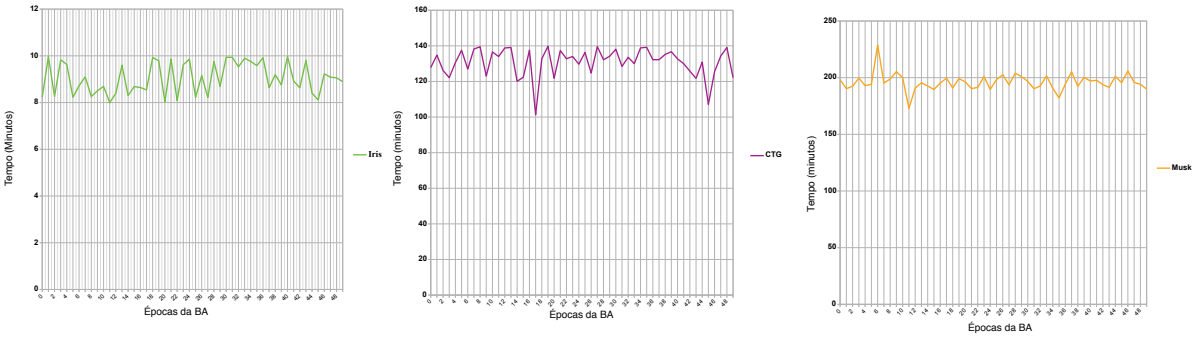



Figura 5.2.10 – Tempos de cada iteração da BA durante a execução do algoritmo nas bases de dados

Apesar da execução dos testes não ocorrerem em um ambiente totalmente controlado, para exemplificação, foram coletados os tempos para cada iteração da busca aleatória para cada base de dados. A tabela 5.2.4 exibe os tempos totais, em minutos, de cada base de dados.

Tabela 5.2.4 – Tempo total de execução do algoritmo proposto para cada base dedos

| **Base de dados** | **Tempo total (minutos)** |
|---|---|
| Iris | 449.450 minutos $\sim$ 7 horas e 30 minutos |
| CTG | $6415,9$ minutos $\sim$ 4 dias e 11 horas |
| MUSK | $9861,1$ minutos $\sim$ 6 dias e 20 horas |

### 5.2.5 Resultados Gerais

O esforço realizado convergiu na implementação de um mecanismo de seleção de hiperparâmetros em modelos de aprendizagem profunda, integrado ao *Tensorflow*, que consiste em uma poderosa ferramenta para desenvolvimento de modelos de AM em grande escala, amplamente utilizada, tanto no mercado de trabalho, quanto no meio acadêmico. Neste âmbito, a pesquisa associada ao desenvolvimento de algoritmos de seleção de hiperparâmetros é, sem dúvida, de extrema valia na solução de problemas de classificação.

Este estudo teve como intuito beneficiar pesquisadores presentes no meio acadêmico e profissionais no mercado trabalho que desenvolvem soluções com o auxilio da Aprendizagem Profunda, além de contribuições para comunidade que ajuda a desenvolver o *Tensorflow*, já que este trabalho fornece uma maneira de proceder a integração entre Validação Cruzada, Otimização de Hiperparâmetros e Aprendizagem Profunda, permitido que demais entusiastas possam criar e aplicar esse método de integração com outras técnicas de Otimização de Hiperparâmetros e outros modelos de Aprendizado Profundo.

## 5.3 Análise dos resultados obtidos

Através dos resultados dos testes é possível observar que a implementação de um método de validação cruzada junto ao algoritmo proposto contornou o problema de sobre ajuste. Como é evidenciado através dos valores da acurácia e função de perda obtidos quando o modelo foi retreinado com os hiperparâmetros do melhor resultado gerados pela busca aleatória e avaliado com os conjuntos de teste. Além disso, pode-se notar que variações pequenas nos valores dos hiperparâmetros geram resultados diferentes.

Apesar de não ser o foco principal deste trabalho, existe um ponto que merece ser levantado: o esforço computacional para realizar otimização de hiperparâmetros usando busca aleatória. Leva-se muito tempo para realizar a busca aleatória, pois deve-se executar o processo de treinamento várias vezes e como foi visto na seção 3.3.1 o treinamento é um dos gargalos quando se trabalha com modelos de aprendizagem profunda. Se o algoritmo proposto tivesse sido implementado de forma paralela e os experimentos ocorressem em um ambiente adequando, provavelmente, obteria-se resultados mais satisfatórios para os testes em relação ao tempo.

Outro fator é que como o modelo usando (DNN) é um modelo profundo teve-se mais flexibilidade em decidir como os dados serão usados para gerar o melhor resultado possível. Em nenhum momento houve a necessidade de aplicar métodos para decidir quais variáveis independentes devem ser incluídas, pois o próprio Modelo considera todos os parâmetros e "automaticamente" determina a melhor combinação dos valores de entrada. Diferentemente de trabalhos que não usam modelos profundos, como (HUANG; HSU, 2012), em que houve a necessidade de executar uma fase de pré processamento dos dados antes do processo de classificação.

Em questões do uso do *Tensorflow* para implementação de modelos de Aprendizagem Profunda, nota-se que a utilização deste *framework* se torna muito interessante devido a sua ótima documentação além de flexibilidade de implementação provida de suas camadas (APIs) contidas no ambiente de programação. Mesmo usando o paradigma *Define-and-Run* o *Tensorflow* é totalmente extensível e permite facilmente a incorporação de modelos e algoritmos de Aprendizagem Profunda (que não estão implementados).

## 5.4 Comparação com resultados apresentados na literatura

É possível comparar os resultados finais obtidos com os resultados de outros trabalhos disponíveis na literatura que empregam as mesmas bases de dados selecionadas para estudo neste trabalho.

Em Rehman e Nawi (2011) foi empregada uma MLP tradicional para o processo de classificação do conjunto de dados Iris. A tabela 5.4.1 exibe os valores dos hiperparâmetros encontrados pela busca aleatória e a configuração da MLP do trabalho de (REHMAN; NAWI, 2011).

Tabela 5.4.1 – Valores dos hiperparâmetros

| **Trabalho** | **Nº Neurônios** | **Nº Camadas** | **T. Aprendizagem** | **Tam *batch*** | **T. *dropout*** | **Momentum** | **Nº épocas** |
|---|---|---|---|---|---|---|---|
| Neste Trabalho | 37 | 4 | 0,00373 | 15 | 0,5 | 0,6 | 600 |
| (REHMAN; NAWI, 2011) | 5 | 1 | 0,4 | – | – | 0.2 | 3000 |

Adaptado: (REHMAN; NAWI, 2011)

Com relação à acurácia obtida neste trabalho, a taxa foi de 98.6% contra 93.85% no trabalho de (REHMAN; NAWI, 2011).

Acerca do Conjunto de Dados de Cardiotocografia, Huang e Hsu (2012) empregou três modelos de aprendizado de máquina para classificação: Análise Discriminante, Árvore de decisão e Rede Neural Artificial. Apesar deste trabalho não detalhar bem as configurações dos hiperparâmetros utilizados pode-se extrair os seguintes dados sobre a RNA (modelo que mais se assemelha a uma DNN): a quantidade de camadas ocultas é igual a 2, a variação do número de neurônio é $[4 - 7]$, e foi aplicada uma aplicada uma busca manual, resultando numa acurácia de 82,1% para Análise Discriminante, 86,36% pela Árvore de

decisão e entre 78,6% e 97,78% com a busca manual para a RNA. No presente trabalho a acurácia alcançada para a rede neural profunda foi de 92%.

Zhou e Zhang (2002) aplicou o conjunto de dados MUSK em seu trabalho. Apesar de considerar uma relação de muitos para um (problema de múltiplas instâncias), ou seja, tinha como objetivo classificar uma molécula como "MUSK" se qualquer uma de suas conformações for classificada como MUSK e como "não MUSK" se nenhuma das suas conformações for classificada como MUSK. Diferentemente deste trabalho que apenas classifica se uma conformação pode ser ou não considerada como MUSK. Pode-se considerar essa falta de algoritmos implementados no *Tensorflow* que tratam este problema de múltiplas instâncias como uma limitação do *Framework*. Apenas para exemplificar (ZHOU; ZHANG, 2002) obteve como resultado uma Acurácia de 80.4% considerando o problema de múltiplas instâncias contra 99.94% obtidas no retreinado do DNN e avaliado com o conjunto de teste, somente considerando se uma conformação é ou não MUSK.

# 6 Conclusão

O objetivo principal deste trabalho esteve concentrado na implementação de uma integração entre a otimização de Hiperparâmetros com os modelos de Aprendizagem Profunda implementados junto ao *Tensorflow*, buscando contribuir com interessados da área de Aprendizagem Profunda, além de analisar a aplicabilidade da integração sugerida em algumas bases de dados.

Com este trabalho é possível concluir que a utilização de técnicas de otimização de hiperparâmetros em modelos de Aprendizagem Profunda é uma abordagem aplicável. Os resultados obtidos se apresentam promissores, e estão associados ainda ao emprego da validação cruzada, no intuito de minimizar os efeitos dos conjuntos de teste e treinamento empregados para avaliação de um processo de classificação.

Apesar do *Tensorflow* ser totalmente escalável, assim como o algoritmo de busca aleatória, em nenhum momento esta característica foi tratada neste trabalho. Entretanto, a questão de escalabilidade deve ser totalmente levada em conta, já que a aplicação de técnicas de aprendizagem Profunda para classificação de dados se torna interessante quando aplicada a conjunto de dados extraordinariamente grandes, o que torna essencial o uso dos algoritmos escaláveis para obtenção de resultados.

Outras técnicas de otimização de hiperparâmetros são definidas na literatura, que podem ser implementadas seguindo o procedimento proposto por este trabalho como a Busca em Grade e a Otimização Bayesiana. Também é totalmente viável analisar a aplicabilidade do método em outros modelos de Aprendizagem Profunda como as Redes Convolucionais e as Redes Recorrentes. Consequentemente aplicar em outras áreas como reconhecimento de imagens, tratamento de linguagem natural etc.

Além disso, já que não existe nenhuma implementação em alto nível sobre a seleção (otimização) automática de Hiperparâmetros no *Framework* usado neste estudo, aplicar tal técnica pode ser extremamente complicado. Portanto, o refinamento da implementação proposta para a API de alto nível do *Tensorflow*, ou seja contribuir com a geração de um modelo oficial[1], seria uma grande contribuição, e pode ser traçado como mais um trabalho futuro.

[1] https://github.com/Tensorflow/models/tree/master/official

# Referências

ABADI, M. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Software available from tensorflow.org. Disponível em: <https://www.tensorflow.org/>. Citado 8 vezes nas páginas 6, 39, 41, 42, 43, 44, 45 e 46.

ABADI, M. et al. Tensorflow: A system for large-scale machine learning. In: *OSDI*. [S.l.: s.n.], 2016. v. 16, p. 265–283. Citado 6 vezes nas páginas 39, 40, 41, 43, 46 e 53.

BERGSTRA, J.; BENGIO, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, v. 13, n. Feb, p. 281–305, 2012. Citado 3 vezes nas páginas 13, 37 e 50.

BRAGA, A. d. P.; CARVALHO, A. C. P. d. L. F.; LUDERMIR, T. B. *Redes neurais artificiais*. 2. ed. [S.l.]: LTC Editora, 2007. Citado 9 vezes nas páginas 6, 17, 18, 19, 20, 21, 23, 24 e 26.

BRINK, H.; RICHARDS, J.; FETHEROLF, M. *Real-world machine learning*. [S.l.]: Manning Publications Co., 2016. Citado na página 33.

BUDUMA, N.; LOCASCIO, N. *Fundamentals of deep learning: designing next-generation machine intelligence algorithms*. [S.l.]: "O'Reilly Media, Inc.", 2017. Citado 6 vezes nas páginas 6, 15, 17, 18, 22 e 25.

CAMPOS, D. Ayres-de et al. Sisporto 2.0: a program for automated analysis of cardiotocograms. *Journal of Maternal-Fetal Medicine*, Taylor & Francis, v. 9, n. 5, p. 311–318, 2000. Citado na página 55.

CARVALHO, A. et al. Inteligência artificial–uma abordagem de aprendizado de máquina. *Rio de Janeiro: LTC*, 2011. Citado 5 vezes nas páginas 12, 19, 23, 24 e 25.

CHEN, X.-W.; LIN, X. Big data deep learning: challenges and perspectives. *IEEE access*, Ieee, v. 2, p. 514–525, 2014. Citado na página 30.

CHENG, H.-T. et al. Tensorflow estimators: Managing simplicity vs. flexibility in high-level machine learning frameworks. In: ACM. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. [S.l.], 2017. p. 1763–1771. Citado na página 44.

CHENG, H.-T. et al. Wide & deep learning for recommender systems. In: ACM. *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. [S.l.], 2016. p. 7–10. Citado na página 17.

CLAESEN, M.; MOOR, B. D. Hyperparameter search in machine learning. *arXiv preprint arXiv:1502.02127*, 2015. Citado 2 vezes nas páginas 35 e 36.

DENG, L.; YU, D. et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, Now Publishers, Inc., v. 7, n. 3–4, p. 197–387, 2014. Citado 3 vezes nas páginas 12, 30 e 31.

DENG, L.; YU, D.; PLATT, J. Scalable stacking and learning for building deep architectures. In: IEEE. *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. [S.l.], 2012. p. 2133–2136. Citado na página 30.

DHEERU, D.; TANISKIDOU, E. K. *UCI Machine Learning Repository*. 2017. Disponível em: <http://archive.ics.uci.edu/ml>. Citado 2 vezes nas páginas 54 e 56.

DUDA, R. O.; HART, P. E.; STORK, D. G. *Pattern classification*. [S.l.]: John Wiley & Sons, 2012. Citado na página 16.

GÉRON, A. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. [S.l.]: "O'Reilly Media, Inc.", 2017. Citado 5 vezes nas páginas 6, 37, 39, 42 e 43.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. Deep learning.(2016). *Book in preparation for MIT Press. URL: http://www. deeplearningbook. org*, 2016. Citado 17 vezes nas páginas 6, 12, 13, 15, 16, 22, 23, 27, 28, 29, 30, 31, 32, 33, 34, 36 e 41.

HAN, J.; PEI, J.; KAMBER, M. *Data mining: concepts and techniques*. [S.l.]: Elsevier, 2011. Citado 2 vezes nas páginas 15 e 16.

HAYKIN, S. *Redes neurais: princípios e prática*. [S.l.]: Bookman Editora, 2007. Citado 11 vezes nas páginas 6, 12, 17, 18, 19, 21, 23, 24, 25, 26 e 27.

HOPE, T.; RESHEFF, Y. S.; LIEDER, I. *Learning TensorFlow: A Guide to Building Deep Learning Systems*. [S.l.]: O'Reilly Media, Inc., 2017. Citado 3 vezes nas páginas 21, 30 e 43.

HUANG, M.-L.; HSU, Y.-Y. Fetal distress prediction using discriminant analysis, decision tree, and artificial neural network. *Journal of Biomedical Science and Engineering*, Scientific Research Publishing, v. 5, n. 09, p. 526, 2012. Citado 2 vezes nas páginas 13 e 65.

HUTCHINSON, B.; DENG, L.; YU, D. Tensor deep stacking networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, v. 35, n. 8, p. 1944–1957, 2013. Citado na página 30.

KARIM, M. R.; ZACCONE, G. *Deep Learning with TensorFlow - Second Edition*. [S.l.]: Packt Publishing Ltd, 2018. Citado 4 vezes nas páginas 6, 33, 39 e 45.

KOCHURA, Y. et al. Performance analysis of open source machine learning frameworks for various parameters in single-threaded and multi-threaded modes. In: SPRINGER. *Conference on Computer Science and Information Technologies*. [S.l.], 2017. p. 243–256. Citado na página 39.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. *nature*, Nature Publishing Group, v. 521, n. 7553, p. 436, 2015. Citado 6 vezes nas páginas 12, 26, 29, 31, 32 e 34.

LECUN, Y.; RANZATO, M. Deep learning tutorial. In: CITESEER. *Tutorials in International Conference on Machine Learning (ICML'13)*. [S.l.], 2013. p. 1–29. Citado 3 vezes nas páginas 21, 32 e 34.

LIN, J. J.; KOLCZ, A. Large-scale machine learning at twitter. In: *SIGMOD Conference*. [S.l.: s.n.], 2012. Citado na página 30.

MCCLURE, N. *TensorFlow Machine Learning Cookbook*. [S.l.]: Packt Publishing Ltd, 2017. Citado 4 vezes nas páginas 6, 40, 42 e 43.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943. Citado na página 16.

MIN, S.; LEE, B.; YOON, S. Deep learning in bioinformatics. *Briefings in bioinformatics*, Oxford University Press, v. 18, n. 5, p. 851–869, 2017. Citado 2 vezes nas páginas 31 e 32.

MITCHELL, T. M. et al. *Machine learning. WCB*. [S.l.]: McGraw-Hill Boston, MA:, 1997. Citado na página 15.

NIELSEN, M. A. *Neural networks and deep learning*. [S.l.]: Determination Press, 2015. Citado 3 vezes nas páginas 30, 31 e 34.

OLAH, C. Calculus on computational graphs: Backpropagation. *Colah's Blog*, 2015. Citado 4 vezes nas páginas 26, 33, 38 e 41.

PATTERSON, J.; GIBSON, A. *Deep Learning: A Practitioner's Approach*. [S.l.]: "O'Reilly Media, Inc.", 2017. Citado 3 vezes nas páginas 13, 16 e 37.

RAO, C. R.; TOUTENBURG, H. Linear models. In: *Linear models*. [S.l.]: Springer, 1995. p. 3–18. Citado na página 17.

REFAEILZADEH, P.; TANG, L.; LIU, H. Cross-validation. In: _____. *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009. p. 532–538. ISBN 978-0-387-39940-9. Disponível em: <https://doi.org/10.1007/978-0-387-39940-9_565>. Citado 3 vezes nas páginas 13, 37 e 51.

REHMAN, M. Z.; NAWI, N. M. The effect of adaptive momentum in improving the accuracy of gradient descent back propagation algorithm on classification problems. In: SPRINGER. *International Conference on Software Engineering and Computer Systems*. [S.l.], 2011. p. 380–390. Citado 2 vezes nas páginas 13 e 65.

RUSSELL, S.; NORVIG, P. Inteligência artificial: uma abordagem moderna. ed. *Prentice-Hall, 3ª Edição. São Paulo, Brazil*, 2014. Citado 8 vezes nas páginas 6, 12, 17, 19, 21, 22, 23 e 25.

SCHMIDHUBER, J. *Deep learning in neural networks: An overview. CoRR abs/1404.7828*. 2014. Citado 4 vezes nas páginas 12, 30, 31 e 34.

TOKUI, S. et al. Chainer: a next-generation open source framework for deep learning. In: *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*. [S.l.: s.n.], 2015. v. 5. Citado 4 vezes nas páginas 6, 38, 39 e 40.

TORRES, J. *First Contact with tensorflow*. [S.l.]: Lulu. com, 2016. Citado 2 vezes nas páginas 45 e 46.

ZACCONE, G.; KARIM, M. R.; MENSHAWY, A. *Deep Learning with TensorFlow*. [S.l.]: Packt Publishing Ltd, 2017. Citado 5 vezes nas páginas 33, 34, 41, 42 e 44.

ZHOU, Z.-H.; ZHANG, M.-L. Neural networks for multi-instance learning. In: *Proceedings of the International Conference on Intelligent Information Technology, Beijing, China.* [S.l.: s.n.], 2002. p. 455–459. Citado 2 vezes nas páginas 13 e 66.