Universidade de Brasília
Departamento de Estatística

Estudo sobre Redes Neurais de Aprendizado Profundo com Aplicações em Classificação de Imagens

por

Eduarda Almeida Leão Marques

Relatório Final de Monografia apresentado para o Departamento de Estatística, Instituto de Ciências Exatas, Universidade de Brasília, como parte dos requisitos necessários para o grau de Bacharel em Estatística.

Brasília
2016

Eduarda Almeida Leão Marques

Orientador:
Prof. Dr. **Donald Matthew Pianto**

Relatório Final de Monografia apresentado para o Departamento de Estatística, Instituto de Ciências Exatas, Universidade de Brasília, como parte dos requisitos necessários para o grau de Bacharel em Estatística.

Brasília
2016

Sumário

Resumo

Em meio a tantos avanços tecnológicos, o estudo e a procura de desenvolvimento de técnicas de inteligência artificial tem ganhado um grande destaque na comunidade acadêmica e científica, entre elas estão as redes neurais. As redes neurais se incluem no universo de técnicas de aprendizado de máquina em que uma rede é treinada para identificar padrões e, assim como o cérebro, após o aprendizado, tornar o conteúdo adquirido disponível para uso. Uma aplicação de grande importância das redes neurais é o reconhecimento de imagem. Esse tipo de aplicação exige uma arquitetura conhecida como redes neurais de aprendizado profundo, ou *deep learning*, onde a ideia é adicionar mais camadas para ampliar sua capacidade de aprendizado e de generalização. Ou seja, aprender os padrões desejados em imagens durante o processo de treinamento e ser capaz de generalizar esse conhecimento para observações que nunca foram vistas antes pela rede. Nos últimos anos, as redes neurais convolucionais têm ganhado grande importância em tarefas de classificação, devido a sua arquitetura, que fazem essas redes serem modelos seletivos e invariantes. Este estudo apresenta um exercício de avaliação comparativa entre diversas arquiteturas de redes neurais e redes neurais de aprendizado profundo em tarefas de classificação de imagens.

Palavras-chave: aprendizado de máquina; aprendizado profundo; redes neurais; redes neurais convolucionais

Abstract

Among many technological advances, the study and demand of techniques development of artificial intelligence has progressively gained recognition in academic and science community, as for example neural networks. Neural networks are included in a universe of machine learning techniques in a net that is trained to, after learning, identify patterns and turn the content available for use, as the brain. One big important use of neural networks is in image recognition. This type of application requires and architecture known as deep learning neural networks, in which the main idea is to add more layers to amplify your learning and generalizing capacity. In another words, to learn desired patterns in images during the training process and being capable to generalize observation knowledge that was never seen before by the net. The convolutional neural networks has attracted more attention in classification tasks over the last years because of your structure, that makes this nets to be selective and invariant models. This study outlines a comparison evaluation exercise between a variety of neural networks architectures and deep learning neural networks in classification images tasks.

Keywords: machine learning; deep learning; neural network; convolutional neural network

Lista de Figuras

Lista de Tabelas

1 Introdução

A sociedade atual está mais tecnologicamente avançada do que nunca. Pessoas foram mandadas para a Lua, os telefones interagem com pessoas, é possível personalizar estações de rádio para tocar somente músicas que agradam ao ouvinte, entre muitos outros avanços. No entanto, as máquinas mais avançadas e computadores ainda lutam para realizar uma das tarefas mais básicas e importantes do ser humano: dar sentido ao que se vê.

Um dos grandes desejos do homem e das grandes empresas é a criação de máquinas capazes de operar sem a necessidade do controle humano, traduzindo todos os dados disponíveis em informação útil para a realidade trabalhada. Sendo assim, as redes neurais têm sido muito exploradas pela comunidade científica e o aprendizado profundo se tornou uma poderosa ferramenta para reconhecer padrão em dados de alta-dimensão, sendo muito aplicável em muitos domínios da ciência, empresas e governos.

As Redes Neurais Artificiais (RNA's) consistem em uma gama de ferramentas que possuem a capacidade de aprender e se adaptar para a realização de certas tarefas através de um processo de aprendizagem. Na estatística esse processo é conhecido como estimação. As habilidades de aprendizagem e adaptação indicam que os modelos de redes neurais podem lidar com dados imprecisos e situações que não foram totalmente definidas no início do treinamento. Por isso, o uso da inteligência artificial se torna cada vez mais atrativa em técnicas de processamento de imagem, reconhecimento de padrão, problemas de classificação, controle de processos, entre outros.

As RNA's tiveram início na década de 40, pelo neurofisiologista MacCulloch, do Instituto de Tecnologia de Massachussetts (MIT), e pelo matemático Waltter Pitts, da Universidade de Illinois, em que o trabalho realizado consistia em um modelo de resistores variáveis e amplificadores representando conexões sinápticas de um neurônio biológico. Desde então, diversos modelos de redes neurais têm sido desenvolvidos para aperfeiçoar e aumentar a aplicabilidade desta técnica em diversas áreas do conhecimento. (BENGIO; GOODFELLOW; COURVILLE, 2016)

Com o avanço da tecnologia e com o desenvolvimento dos bancos de dados, sentiu-se a necessidade de desenvolver técnicas cada vez mais robustas para as redes neurais. Técnicas de aprendizado, como aquelas por meio do gradiente ganharam cada vez mais destaque graças ao desenvolvimento do algoritmo de *back-propagation*, que tem sido de extrema importância para o desenvolvimento das redes neurais de aprendizagem profunda. Essas técnicas permitiram também o desenvolvimento das redes recorrentes e convolucionais.

As redes convolucionais (ConvNets) são utilizadas no processamento de imagens, vídeos, voz e áudio, enquanto que as redes recorrentes são mais eficientes para análise de

dados sequenciais, como análise de texto e discurso.

De acordo com LeCun (2015), a aprendizagem profunda, ou *Deep Learning*, permite que modelos computacionais compostos de várias camadas de processamento possam aprender apresentações de dados com múltiplos níveis de abstração. Sendo assim, o aprendizado profundo é representado por funções de grande complexidade, onde adicionando mais camadas e unidades entre aquelas chega-se ao resultado o qual as redes profundas se tornam muito eficientes em tarefas que consistem em mapear o vetor de entradas em vetor de saídas.

As redes neurais de aprendizado profundo são fundamentais para o desenvolvimento de diversas ferramentas que, hoje, são fundamentais no contexto diário da sociedade, como ferramentas de criação automática de textos no momento da digitação de uma mensagem em um smartphone, por exemplo. Além disso, no quesito segurança, uma máquina pode ser treinada com diversos exemplos de caligrafia da assinatura de uma determinada pessoa e ser capaz de classificar uma nova assinatura como verdadeira ou não verdadeira. Facilitadores como reconhecimento de um sorriso como um sinal de captura de uma foto, reconhecimento de um determinado rosto em uma imagem, que são muito comuns em redes sociais, também são instrumentos desenvolvidos a partir das redes neurais de aprendizado profundo.

1.1 Justificativa

As redes de aprendizado profundo são muito utilizadas em problemas de classificação de imagens, reconhecimentos de fala e escritas. Esses problemas requerem um modelo que seja seletivo e invariante, ou seja, uma função que produz representações que são seletivas com relação aos detalhes que são importantes para a discriminação de fatores presentes na imagem, mas que são invariantes a aspectos irrelevantes, tais como posição, tamanho, iluminação, entre outros. Para resolver esse problema, uma rede pode ser treinada por um processo de aprendizagem de uso geral onde boas características podem ser aprendidas pela rede. Esta é exatamente a principal vantagem de se usar redes de aprendizado profundo, ou *Deep Learning*.

As redes convolucionais integram uma parte importante das redes profundas. São um exemplo chave de uma aplicação bem sucedida de *insights* obtidos por estudar o cérebro em aplicação de aprendizado de máquina. Essas foram as primeiras redes profundas a serem treinadas utilizando o algoritmo de *back-propagation*. Aplicações de redes convolucionais ganharam uma série de concursos, incluindo o concurso promovido pelo ImageNet para reconhecimento de objetos. As redes convolucionais tem sido mais eficientes do que as redes que são totalmente conectadas, por serem redes especializadas para

trabalhar com dados de topologia semelhante a grades, como uma imagem 2D que possui grades de pixels.

1.2 Objetivos

O objetivo geral deste trabalho é estudar e entender as redes neurais de aprendizado profundo. E, o objetivo específico deste trabalho é aplicar as redes neurais e redes neurais de aprendizado profundo em um problema de classificação de imagem.

2 Revisão de Literatura

2.1 Aprendizado de Máquina

As técnicas de aprendizado de máquina são apresentadas como técnicas e algoritmos em que o modelo é capaz de analisar e aprender rapidamente tarefas que os seres humanos demorariam muito tempo. Alguns modelos de aprendizado de máquina são iterativos, ou seja, conforme os modelos são expostos a novos dados, eles são capazes de se adaptar de forma independente, sem a intervenção de humanos.

Um algoritmo de aprendizagem é aquele capaz de aprender com os dados. Segundo Mitchell (2007), uma definição de aprendizagem é: "Um certo programa aprende a partir de sua experiência, com respeito a uma classe de tarefas que são mensuradas e que melhora com sua experiência". Os detalhes de cada uma dessas etapas, tarefa, mensuração e experiência, serão descritos a seguir no contexto de aprendizado de máquina.

Primeiramente, o modelo ganha experiência a partir do seu tipo de treinamento, que pode ser supervisionado ou não supervisionado.

A ideia do processo de treinamento supervisionado é ajustar um modelo com o objetivo de compreender a relação entre as variáveis respostas e preditoras para, então, prever com precisão a resposta de observações futuras, ou seja, que não foram apresentadas ao modelo anteriormente.Este tipo de aprendizado acontece quando é fornecido ao vetor de treinamento a resposta que se deseja obter, através de exemplos de entrada e saída. Assim, se o problema de classificação for discreto, pode-se entender que para cada valor de entrada, $x_i,\ i = 1, ..., n$ existe um rótulo associado y_i. Se o problema de classificação for contínuo, pode-se usar uma regressão. Posteriormente a resposta desejável é confrontada com a saída apresentada pela rede, fornecendo um sinal de erro.

Um grande conjunto de técnicas usam o treinamento supervisionado, como regressão, SVM, rede neural, entre outras.

Já o treinamento não supervisionado acontece quando para cada valor de entrada $x_i,\ i = 1.., n$ não existe um valor de resposta y_i associado. Esta situação é referida como não supervisionada exatamente pelo fato de que não existe uma resposta para supervisionar o treinamento da análise. Então, procura-se entender e aprender a relação entre as variáveis das observações e identificar quais são semelhantes. Algumas técnicas muito usadas neste tipo de treinamento são as análises de Cluster e Componentes principais.

As tarefas de aprendizado normalmente descrevem o tipo de processamento de observações por meio do sistema de aprendizagem, sendo que uma observação $x \in \mathbb{R}^n$ representa um conjunto de características de um determinado objeto ou evento de inte-

resse e cada entrada x_i representa uma característica específica deste objeto. Por exemplo, as características de uma imagem são representadas pelos seus pixels.

Existem muitos tipos de processamento para cada tipo de "tarefa"em aprendizado de máquina. Por exemplo, classificação, regressão, transcrição, tradução, entre outros.

Em problemas de classificação, o sistema tem a tarefa de designar para cada observação de entrada, uma das k classes disponíveis. Então, para resolver esse problema, o algoritmo de aprendizagem produz uma função $f : \mathbb{R}^n \rightarrow \{1, ..., k\}$. Para os problemas de regressão, o sistema tem a tarefa de prever um valor numérico para cada observação de entrada. Então, o algoritmo de entrada produz uma função $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Este tipo de tarefa é similar à classificação, diferindo apenas no formato da saída.

Existem infinitas tarefas que podem ser realizadas por meio do uso de aprendizado de máquina e para cada uma delas é necessária a análise do tipo de processamento mais adequado.

Para avaliar a qualidade de ajustamento do método de aprendizado de máquina, é necessário o uso de uma medida quantitativa de desempenho, ou seja, medir quão bem suas previsões realmente correspondem aos dados observados. Por isso é necessário quantificar o quanto a resposta prevista pelo modelo está distante do valor real de cada observação. Cada tipo de tarefa recebe uma medida adequada para avaliação de desempenho. Por exemplo, para tarefas de classificação usa-se a acurácia ou a taxa de erro e, já para problemas de regressão, pode ser usado o erro quadrático médio (MSE).

A acurácia se refere à capacidade do modelo de classificar corretamente as observações e a taxa de erro mostra a proporção de classificações incorretas.

Como o problema central em aprendizado de máquina é estimar os parâmetros de um modelo que seja capaz de ter um desempenho ótimo em observações em que não foram fornecidas durante o treinamento, é possível admitir que a capacidade do modelo de ajustar-se bem a novas observações é chamada de generalização.

Dois fatores principais podem contribuir para a redução na capacidade de generalização do modelo, o *underfitting* e *overfitting* (sub-ajuste e o sobre-ajuste).

O sub-ajuste ocorre quando os parâmetros do modelo não se ajustam aos dados. Por outro lado, o sobre-ajuste ocorre quando o modelo se ajusta muito bem especificamente aos dados de treinamento, perde a capacidade de ajuste a dados que não foram vistos durante o treinamento.

2.2 Rede Neural e Neurônio Artificial

Define-se uma Rede Neural Artificial como sendo um processador paralelamente distribuído e constituído de unidades de processamento simples, que tem a propensão natural para armazenar conhecimento experimental e torná-lo disponível para uso. Ela se assemelha ao cérebro pelo fato de que o conhecimento é adquirido pela rede a partir de seu ambiente por meio de um processo de aprendizagem, onde forças de conexões entre neurônios são utilizadas para armazenar o conhecimento adquirido, estes são conhecidos como pesos sinápticos (HAYKIN, 2004).

Na Figura 1 é apresentado um modelo simplificado de uma Rede Neural Artificial simples e de um neurônio artificial. A estrutura de uma RNA é composta por 3 camadas principais: a camada de entrada, as camadas escondidas e a camada de saída. A estrutura de um neurônio é composta pelos sinais de entrada, os pesos sinápticos, uma função de ativação e as saídas do neurônio.

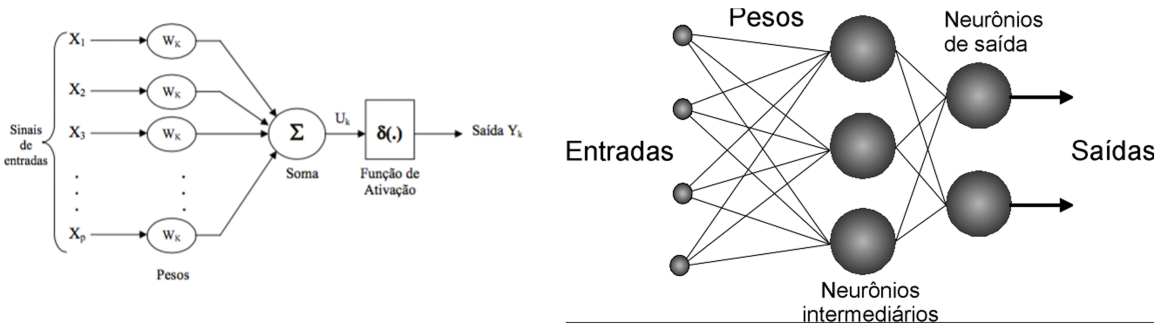


Figura 1: Descrição de Rede Neural e Neurônio Artificial

A camada de entrada é a camada onde os dados são apresentados para a rede. A camada escondida se refere àquela onde são definidos os pesos sinápticos. E ainda, a camada de saída é a camada onde são apresentados os resultados da rede.

Os sinais de entrada de um neurônio se referem aos sinais que serão utilizados como entrada da Rede Neural. Já os pesos sinápticos são os valores que serão ajustados para a rede durante o processo de aprendizagem. Esses valores são multiplicados pelos valores de entrada que determinam a influência do sinal do neurônio. Assim, cada neurônio pode tratar o impulso recebido de maneira diferente.

Outro fator importante é que os pesos podem assumir valores positivos ou negativos. O somador realiza o somatório de todos os resultados da multiplicação dos sinais pelos pesos. Por fim, outra característica do neurônio artificial é a função de ativação, tendo em vista que essa é a função que define se um neurônio será ativado ou não. E a saída do

neurônio são os resultados para cada camada em que o mesmo é exposto.

2.3 Treinamento de uma Rede Neural

Como já foi mencionado, uma importante propriedade das redes neurais é a habilidade de aprender de seu ambiente e, com isso, melhorar seu desempenho. Isso é feito através do treinamento, onde acontece todo o processo de ajustes aplicado a seus pesos. Então, o aprendizado ocorre quando a rede neural é capaz de promover uma solução generalizada para uma classe de problemas.

Sendo assim, uma importante tarefa da rede neural consiste em aprender um modelo no ambiente em que está inserida, ou com os dados apresentados a ela. Então, a representação da aprendizagem é um conjunto de métodos que permite uma máquina ser alimentada com dados e então estimar os parâmetros de um modelo para detectar as representações necessárias para a classificação. O desempenho da rede é testado com dados não apresentados anteriormente a ela. Se o aprendizado for supervisionado, o desempenho é estimado comparando-se a saída da rede com o verdadeiro valor. Portanto, primeiramente uma RNA passa por um processo de aprendizagem, onde o conhecimento é adquirido e, então, acontece um processo de generalização onde o conhecimento adquirido torna-se disponível para uso em novas observações.

O treinamento de uma rede neural consiste na estimação de um modelo que minimize o erro entre a saída do sistema e a resposta desejada. Para a estimação desse modelo, precisa-se ajustar uma função de custo e o termo de regularização, escolher a função de ativação para calcular os pesos e, para atualização desses pesos, usa-se um método de aprendizagem baseado no gradiente. Basicamente, o treinamento da rede neural é feito utilizando otimizadores iterativos baseados em gradientes que conduzem a função de custo para um valor muito baixo.

2.3.1 Função de Custo

Muitas vezes, simplesmente, reduzir o número de classificações erradas não torna o algoritmo eficiente. Por isso tem-se as funções de custo, ou funções de perda.

Segundo Bishop (2006), a função de custo é "uma medida global de perdas ocorridas para se tomar qualquer decisão ou ação. Sendo assim, o objetivo é minimizar a perda total do evento ocorrido".

Suponha que para um novo valor de x, a verdadeira classe é y_k, mas x foi classificado como y_j (j pode ser ou não igual a k). Assim, em cada classificação tem-se um nível de perda denotado por um escalar L_{kj} e a matriz de perda é construída com os $k \times j$ elementos.

A solução ótima é aquela que minimiza a função de perda. No entanto, essa função depende da verdadeira classe de x, o que é, muitas vezes, desconhecida. Então, para um vetor de entradas x, sua verdadeira classe é dada pela distribuição $p(x, y_k)$ e a perda esperada, chamada de risco, é dada por:

$$\mathbb{E}[L] = \sum_k \sum_j \int_{R_j} L_{kj} p(x, y_k) dx$$

Ao invés de minimizar o risco, o objetivo é escolher regiões R_j, que são partições do espaço amostral, tal que minimizam o risco, o que implica que, para cada x tem-se que minimizar:

$$\sum_k L_{kj} p(x, y_k)$$

Pode-se usar a regra do produto $p(x, y_k) = p(y_k|x)p(x)$ para eliminar o fator comum $p(x)$.

Assim, a regra de decisão que minimiza a perda esperada é aquela que atribui cada nova observação de x para a classe j que minimiza:

$$\sum_k L_{kj} p(y_k|x)$$

Pode-se, também, estimar a função de custo a partir da Máxima Verossimilhança. Considerando que a probabilidade de classificar corretamente uma observação segue uma distribuição Bernoulli(p). Como cada $x_i = (x_1, ..., x_n)$ é independente e identicamente distribuído, então o custo pode ser estimado a partir da probabilidade de se classificar n observações corretamente, que é dada pela distribuição Binomial a seguir:

$$l_i(p) = p^s(1-p)^f$$

Onde s representa o número de sucessos, ou seja, classificações corretas, e f o número de fracassos.

Aplicando o *log*:

$$-l(\prod_i p) = \sum_s \left(-log\left(p(x)\right)\right) + \sum_f \left(-log\left(1-p(x)\right)\right)$$

Então, a estimação log-verossimilhança é dada por:

$$-l_i(\prod_i p) = \sum_i -log(p(y = y_i|x_i))$$

Para minimizar a função log-verossimilhança, deve-se escolher y_i tal que a probabilidade dada por $p(y = y_i|x_i)$ é máxima. Assim, y receberá a classificação dada por y_i.

A vantagem dessa abordagem de derivar a função de custo a partir da Máxima Verossimilhança é que isto remove a necessidade de se estimar uma função custo para cada modelo. Especificar o modelo $p(y|x)$ é suficiente para definir a função de custo $-log(p(y|x))$.

As funções de custo mais comuns são as funções de perda 0-1, L^{0-1} e o erro quadrático, L^{SE}. A perda 0-1 é comumente utilizada em tarefas de classificação, onde a perda contabiliza a quantidade de classificações erradas e é dada por:

$$L^{0-1}(y, f(x)) = \begin{cases} 0 & f(x) = y \\ 1 & f(x) \neq y \end{cases}$$

A função de perda erro quadrático é comum em tarefas de regressão, onde a perda é representada pela distância entre o valor real e o predito. Portanto, quanto maior a distância entre o valor predito do valor real, maior é o custo.

$$L^{SE}(y, f(x)) = \|y - f(x)\|_2^2$$

A função de custo utilizada para treinar redes neurais profundas, normalmente combina o risco com um termo de regularização.

2.3.2 Regularização

Em aprendizado de máquina, existem diversas técnicas para reduzir o erro de teste e o sobre-ajuste. Essas técnicas são chamadas de regularização.

Define-se regularização como "qualquer modificação realizada no algoritmo de aprendizagem com o objetivo de reduzir o erro de generalização, mas não o erro de treinamento". (GOODFELLOW; BENGIO; COURVILLE, 2016). Existem muitas técnicas de regularização, como adicionar restrições ou penalidades ao modelo ou aos valores dos parâmetros. No contexto de redes neurais com aprendizado profundo, o método que será utilizado é de penalidade na norma dos parâmetros (*Parameter Norm Penalties*).

Esse método é baseado em limitar a capacidade do modelo adicionando um parâmetro de penalidade $\Omega(\theta)$ à função de custo C. Então, uma função regularizada é dada por $\tilde{C}$, tal que:

$$\tilde{C}(\theta, X, y) = C(\theta; X, y) + \alpha\Omega(\theta)$$

Onde $\alpha \in [0, \infty)$ é o hiperparêmetro de contribuição relativa do termo de penalidade Ω com relação à função de custo $C(x, \theta)$. Se $\alpha = 0$ então não existe penalidade e quanto maior o valor de α, maior é a regularização. Os dois tipos mais comuns de parâmetros de penalidade são *Ridge Regression* (L^2) e o LASSO (L^1).

O termo de regularização L^2 é conhecido como *Ridge Regression* e tem como estratégia de regularização diminuir o valor dos pesos, levando-os até a origem adicionando um termo de regularização $\Omega(\theta)$ à função de custo, onde $\Omega(\theta) = \frac{1}{2}\|w\|_2^2$. Então o modelo com o termo de regularização é dado por:

$$\tilde{C}(w, X, y) = \frac{\alpha}{2} w^T w + C(w; X, y)$$

O objetivo do termo de regularização é encontrar $w^* := arg \min_w \tilde{C}(w; X, y)$.

Neste sentido, pode-se começar a compreender o comportamento do termo de regularização estudando o gradiente da função de custo regularizada. O gradiente da função de custo regularizada com relação aos pesos w é dado por:

$$\nabla_w \tilde{C}(w, X, y) = \alpha w + \nabla_w C(w; X, y)$$

A atualização dos pesos com o termo de regularização é dada por:

$$w \leftarrow w - \epsilon(\alpha w + \nabla_w C(w; X, y))$$

$$w \leftarrow (1 - \epsilon\alpha)w - \epsilon\nabla_w C(w; X, y)$$

Uma forma alternativa ao *Rigde Regression* para regularização é dada pelo decaimento dos pesos (*weight decay*). A ideia básica dessa abordagem é reduzir a influência dos pesos de cada conexão, evitando o aumento desnecessário dos valores dos mesmos nas unidades intermediárias. Segundo com Goodfellow, Yoshua, Courville (2016) o processo de decaimento dos pesos é realizado minimizando a soma que compreende tanto o MSE no treinamento, quanto um critério $J(w)$, que expressa a preferência para um certo peso ter a menor norma quadrada L^2. Especificamente, temos que:

$$J(w) = MSE_{treinamento} + \lambda w^T w$$

Onde λ representa o parâmetro de decaimento dos pesos, ou seja, o parâmetro que controla a força imposta para a redução nos valores dos pesos. Então quando $\lambda = 0$, não existe imposição de que os pesos devem ser reduzidos e quanto maior o valor de λ, maior a imposição de pesos menores.

Sendo assim, minimizar $J(w)$ implica em uma escolha de pesos que são pequenos o suficiente para se ajustarem aos dados de treinamento. Ou seja, o objetivo é encontrar um valor intermediário de λ que controla a habilidade de aprendizagem no modelo em que os pesos são definidos em apenas um conjunto de características e, assim aumentar sua capacidade de generalização. Dessa forma, se os pesos são definidos em mais características pode acontecer o sobre-ajuste e, em menos características, o sub-ajuste.

O LASSO é uma outra forma alternativa ao *Ridge Regression* para penalização. Formalmente, a regularização L^1 representa a soma dos valores absolutos dos pesos e é dado por:

$$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|$$

No caso da regularização L^1, o decaimento do peso controla a resistência da regularização direcionada para a penalidade Ω usando um hiperparâmetro positivo α. A função regularizada é dada por $\tilde{C}$, tal que:

$$\tilde{C}(w; X, y) = \alpha \|w\|_1 + C(w; X, y)$$

E o gradiente correspondente é dado por:

$$\nabla_w \tilde{C}(w; X, y) = \alpha \; sign(w) + \nabla_w C(w, X, y)$$

Onde *sign* representa a função sinal de w. O objetivo também é encontrar uma solução em que: $w^* := arg \min_w \tilde{C}(w; X, y)$.

2.3.3 Algoritmo de *Back-propagation*

Quando uma entrada x é apresentada à rede, a informação inicial das observações é propagada camada por camada até que a informação processada chegue na camada de saída $\hat{y}$. Esse processo de propagação da informação é chamado de propagação para frente. Durante o treinamento, a propagação para frente continua até que a rede produza um custo escalar $L(\theta)$. Então, o algoritmo de *back-propagation* permite que a informação do custo passe por toda a rede, mas agora no sentido de trás para frente, com a intenção atualizar os pesos de cada entrada de cada neurônio de acordo com o gradiente.

Para definir o vetor gradiente, é importante lembrar que a derivada de uma função indica a taxa de variação em torno de uma pequena região perto de um ponto específico. Sendo assim, o vetor gradiente indica o sentido que representa a maior taxa de crescimento (ou decrescimento) em um determinado ponto. Portanto:

Definição 2.1 *Seja $f(x_1, ..., x_n)$ uma função diferenciável, então o vetor gradiente de f é dado por:*

$$\nabla f = \left\langle \frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_n} \right\rangle$$

Sendo que o ∇f representa o crescimento máximo da função, $-\nabla f$ é o decrescimento máximo e a taxa de variação é dada pela $\|\nabla f\|_2$[1].

O cálculo do gradiente necessita do valor das derivadas das funções. Como essas funções muitas vezes são composições de outras funções, é muito mais fácil utilizar a regra da cadeia para computar tais derivadas. Essa regra parte do princípio de que a derivada da composta de duas funções deriváveis é o produto de suas derivadas. A vantagem de se usar essa técnica é transformar o cálculo da derivada de uma função composta na derivada de duas funções conhecidas, ou seja, calcular a derivada de $f(g(x))$ no ponto x equivale a derivada de f no ponto $g(x)$ multiplicada pela derivada de g no ponto x, onde f e g são duas funções conhecidas.

Definição 2.2 *Seja $x \in \mathbb{R}$ e $f, g : \mathbb{R} \to \mathbb{R}$. Supondo $y = g(x)$ e $z = f(g(x)) = f(y)$, então os estados da regra da cadeia são dados por:*

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Para exemplificar como o *back-propagation* funciona para atualizar os pesos utilizando a regra da cadeia para calcular as derivadas de uma função, considere a função $f(x, y, z) = (x + y)z$, onde $g = (x + y)$ e $f = gz$. Considerando as entradas, $x = -2$, $y = 5$ e $z = -4$, tem-se o seguinte grafo na Figura 2 na fase de propagação a frente da informação, ou seja, considerando a composição da função $f(x, y, z) = (x+y)z$, na fase de propagação para frente, a informação tem inicio nos pontos x,y e z e chega em f a partir da aplicação da função em cada ponto.

Para computar o *back-propagation*, a propagação da informação tem início em f e precisa chegar nos estados iniciais x, y, e z. Para isto, usa-se a regra da cadeia para

[1] A norma representa o comprimento de um vetor, onde:

$$\|x\|_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}$$

Para $p \in \mathbb{R}$, $p \geq 1$

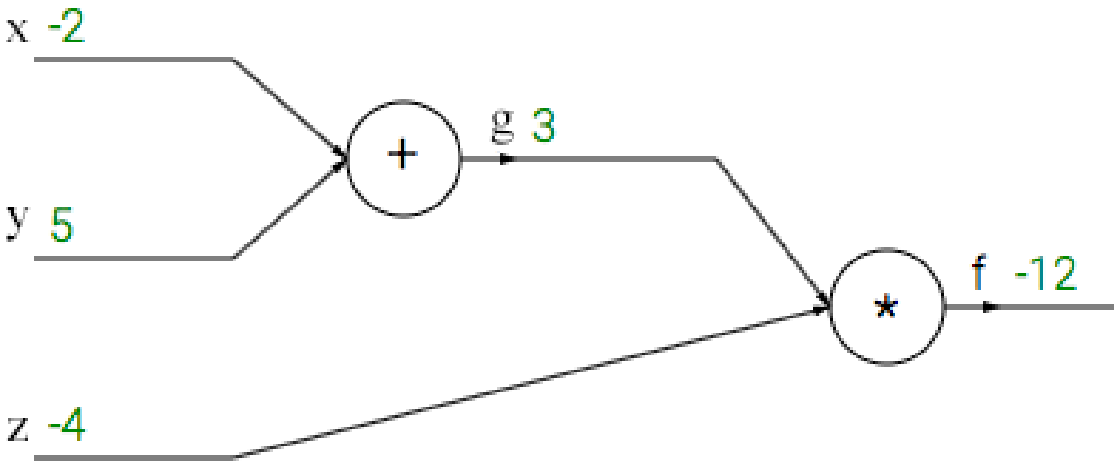


Figura 2: Descrição da propagação para frente

calcular as derivadas das funções composta, permitindo que a informação se propague com mais facilidade.

- Para a informação chegar em x:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\partial g}{\partial x}$$

$$\frac{\mathrm{d}f}{\mathrm{d}x} = z * 1 = -4$$

- Para a informação chegar em y:

$$\frac{\mathrm{d}f}{\mathrm{d}y} = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\partial g}{\partial y}$$

$$\frac{\mathrm{d}f}{\mathrm{d}y} = z * 1 = -4$$

- Para a informação chegar em z:

$$\frac{\mathrm{d}f}{\mathrm{d}z}$$

$$\frac{\mathrm{d}f}{\mathrm{d}z} = q = 3$$

O grafo representado na Figura 3 mostra a propagação da informação para frente, representada pelos valores em verde, e para trás, valores em vermelho. Quando o passe para frente acaba, durante o *back-propagation*, cada neurônio irá aprender sobre o gradiente de cada saída. A regra da cadeia diz que cada nó deve pegar seu gradiente e computá-los normalmente para todas as entradas. Como a ideia do gradiente é escolher

o ponto de maior crescimento, então se x, y diminuem (devido ao seu gradiente negativo $\nabla = -4$, então o nó de adição $x + y$ diminui, o que torna o nó da multiplicação maior.

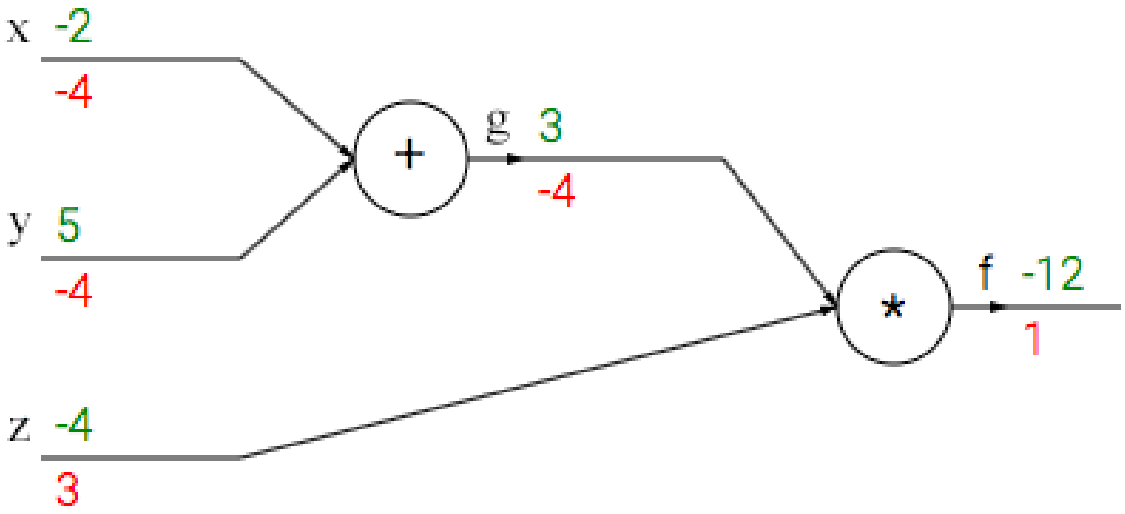


Figura 3: Descrição da propagação para trás

Para um problema de minimização de uma função com múltiplas entradas $f : \mathbb{R}^n \rightarrow \mathbb{R}$ tem-se que utilizar o conceito de derivadas parciais. As derivadas parciais $\frac{\partial}{\partial x_i} f(x)$ medem o quanto f varia com uma pequena variação em x_i no ponto x. Então, o gradiente $\nabla_x f(x)$, contém todas as derivadas parciais de f, sendo que o i-ésimo elemento representa a derivada parcial de f com respeito a x_i. Portanto, a matriz que contém todas as derivadas parciais de uma função é conhecida como matriz Jacobiana.

Definição 2.3 *Seja* $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$*, então a matriz Jacobiana* $J \in \mathbb{R}^{n \times m}$ *de* f *é dada por:*

$$J_{i,j} = \frac{\partial}{\partial x_j} f_i(x), \ i = 1, ..., n \ e \ j = 1, ..., m$$

Generalizando essa regra da cadeia para um caso em múltiplas entradas. Seja $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $g : \mathbb{R}^m \rightarrow \mathbb{R}^n$ e $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Se $y = g(x)$ e $z = f(y)$, então:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Então, o gradiente $\nabla_x z$ é equivalente a seguinte notação vetorial:

$$\nabla_x z = \left(\frac{\partial y}{\partial x} \right)^T \nabla_y z = (J_{xy})^T \nabla_y z$$

Pode-se notar que o gradiente da variável x pode ser obtida pela multiplicação da matriz Jacobiana de y com relação a x pelo gradiente de z com respeito a y. A atua-

lização via *back-propagation* acontece exatamente multiplicando a Jacobiana da função de ativação pelo gradiente do custo total.

Para exemplificar o algoritmo de *back-propagation* passando pela propagação para frente e para trás, considere uma função com uma única variável de entrada com l camadas e os parâmetros do modelos são dados pelo peso, W, e o bias, b.

Na propagação para frente, o objetivo principal é calcular a função custo. Como a perda $L(\hat{y}, y)$ depende do resultado $\hat{y}$ e do valor desejado y, então o algoritmo irá computar o valor do gradiente de C para cada parâmetro W e b, onde C representa o custo total e a perda pode ter um termo de regularização $\Omega(\theta)$, se necessário, e θ contém todos os parâmetros. Então o algoritmo se da pelas seguintes etapas:

1. Os dados são apresentados para a rede na camada de entrada:

$$h^{(0)} = x$$

2. Para cada camada $k = 1, ..., l$:

$$\alpha^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$$

$$h^{(k)} = f(\alpha^{(k)})$$

   Onde f é a função de ativação da camada.

3. Após completar esta iteração para todas as camadas, tem-se um sinal de saída:

$$\hat{y} = h^{(l)}$$

4. O custo total então é dado pela perda somada do termo de regularização:

$$C = L(\hat{y}, y) + \lambda \Omega(\theta)$$

Após computar o custo, o objetivo do algoritmo é propagar $\nabla_{\hat{y}} C$ para atualização dos parâmetros W e b. Essa propagação ocorre no sentido inverso, ou seja, começa na última camada e passa por todas as camadas até chegar na primeira.

5. Calcular o gradiente do custo total na camada de saída. Esse valor será atualizado a cada camada para que seja propagado para as próximas camadas o valor atual do custo.

$$g \leftarrow \nabla_{\hat{y}} C = \nabla_{\hat{y}} L(\hat{y}, y)$$

   Para cada camada $k = l, l-1, ..., 1$:

6. Converter $g = \nabla_{\hat{y}} C$ em um gradiente do custo com relação a cada $\alpha^{(k)}$ através do gradiente do custo e da derivada da função de ativação. A função g é atualizada ao final de cada iteração.

$$\nabla_{\alpha^{(k)}} C = g \odot f'(\alpha^{(k)})$$

Onde $f'(\alpha^{(k)})$ é o Jacobiano de f em relação a $\alpha^{(k)}$.

7. Atualizar os parâmetros W e b, incluindo o termo de regularização, quando necessário.

$$\nabla_{b^{(k)}} C = \nabla_{\alpha^{(k)}} C + \lambda \nabla_{b^{(k)}} \Omega(\theta)$$

$$\nabla_{W^{(k)}} C = h^{(k-1)T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$$

8. Após a atualização dos parâmetros, tem-se que atualizar o gradiente do custo representado por g para a propagação da informação para as próximas camadas até que chegue na primeira camada:

$$g \leftarrow \nabla_{h^{(k-1)}} C = W^{(k)T} \nabla_{\alpha^{(k)}} C$$

Um aspecto importante é que o *back-propagation* não precisa ter acesso aos valores numéricos específicos de todos os gradientes. Ao invés disso, apenas descreve como calcular as derivadas através do grafo de operações, tais como nas figuras 2 e 3, e a máquina é capaz de calcular as derivadas de qualquer valor específico. Isso é útil porque o algoritmo não precisa calcular os gradientes com relação a todos os pontos. O gradiente é definido uma vez e o algoritmo calcula o valor específico quando necessário, o que torna o algoritmo mais rápido e eficiente.

2.4 Inicialização dos Parâmetros

Um aspecto muito importante para a propagação e para o aprendizado são os valores de θ, ou seja, pesos, W, e bias, b. Como os pesos são os valores que representam a intensidade de um sinal, o processo de inicialização desses parâmetros é essencial para definir a velocidade do processo de aprendizagem.

Pesos iniciais com valores grandes ajudam a evitar unidades redundantes, ou seja, com a mesma informação e ajudam também a evitar a perda de sinal durante a propagação da informação para frente e para trás. Porém, pesos grandes podem resultar em valores extremos que saturam a função de ativação, causando uma perda completa do gradiente nessas unidades e, consequentemente a diminuição na velocidade do aprendizado. Por outro lado, pesos iniciais com valores pequenos, perto de 0, podem acarretar em perda de sinais durante as propagações. O ideal é que esteja em um meio termo e que o valor

dos pesos iniciais, que serão atualizados durante o algoritmo de aprendizagem, não seja muito diferente dos pesos finais.

Pensando sob o ponto de vista da otimização do gradiente, os pesos devem ser suficientemente grandes para propagar a informação, mas o termo de regularização faz com que os pesos sejam menores. Por isso, existem alguns métodos para gerar os parâmetros iniciais, como, por exemplo, gerando-os através de uma distribuição Gaussiana ou uniforme. Normalmente a distribuição geradora dos valores dos pesos não faz muita diferença. O que realmente importa é a extensão dessas distribuições.

Considerando que θ_0 são os parâmetros iniciais de θ, e que θ segue uma distribuição Gaussiana com média θ_0, distribuição a priori $p(\theta) \sim N(0, \sigma)$. Assim, como já foi mencionado, outra distribuição muito comum para inicialização dos pesos é a distribuição uniforme, então a priori $p(\theta) \sim U[-a, a]$.

2.5 Função de Ativação

As funções de ativação de uma rede dependem do tipo de situação a ser tratada. Normalmente é difícil determinar quando usar cada função de ativação, sendo que o melhor modelo é definido a partir de tentativa e erro, e, nesse caso, a intuição e o conhecimento sobre o problema podem ser usados como tentativa a priori. Por exemplo, para uma rede que irá tratar de um problema de classificação binária com uma única variável, uma alternativa adequada seria utilizar o modelo logístico. Porém, para um problema em que as entradas são contínuas, pode-se usar um modelo de regressão para classificação das observações. Podem surgir diversos problemas de classificação de rede, então deve-se procurar a função de ativação mais adequada para cada situação apresentada para a rede.

As funções de ativação são definidas dentro das camadas escondidas. Então, supondo um vetor de entradas x, dentro da camada escondida é aplicada uma transformação affine $z = W^T x + b$, e depois é aplicada uma função não linear $g(z)$, chamada de função de ativação. Algumas funções comuns são: sigmóide, tanh, ReLU, que serão detalhadas a seguir.

É desejável que as funções de ativação sejam diferenciáveis [2] em todos os pontos de entrada para passar pelo processo de aprendizagem da rede que é realizado via gradiente.

Algumas funções de ativação não são diferenciáveis em todos os pontos, por exemplo a função ReLU, dada por $g(z) = max\{0, z\}$, não é diferenciável no ponto $z = 0$:

$$\lim_{z \to 0^-} = \frac{g(z) \ - \ g(0)}{z \ - \ 0} = 0$$

[2] Uma função é diferenciável se as derivadas laterais são definidas e iguais.

$$\lim_{z \to 0^+} = \frac{g(z) \ - \ g(0)}{z \ - \ 0} = 1$$

Na prática, o algoritmo de aprendizagem da rede neural reduz significativamente o valor da função de custo, mas normalmente não atinge o seu mínimo global. Sendo assim, as funções que não são diferenciáveis em todos os pontos possuem apenas um pequeno conjunto de pontos não diferenciáveis. Então, em geral, a função $g(z)$ tem a derivada esquerda definida pela inclinação da função no valor imediatamente à esquerda de z e à derivada direita é definida pela inclinação da função no valor imediatamente a direita de z. Portanto, funções que possuem poucos pontos de descontinuidade não interferem significativamente no processo de aprendizado da rede.

2.5.1 ReLU

A função ReLU é uma função de unidade linear retificado que ganhou grande popularidade nos últimos anos devido a sua facilidade de implementação, principalmente, em aplicações de reconhecimento de imagem. Esta é uma função estritamente positiva dada por:

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

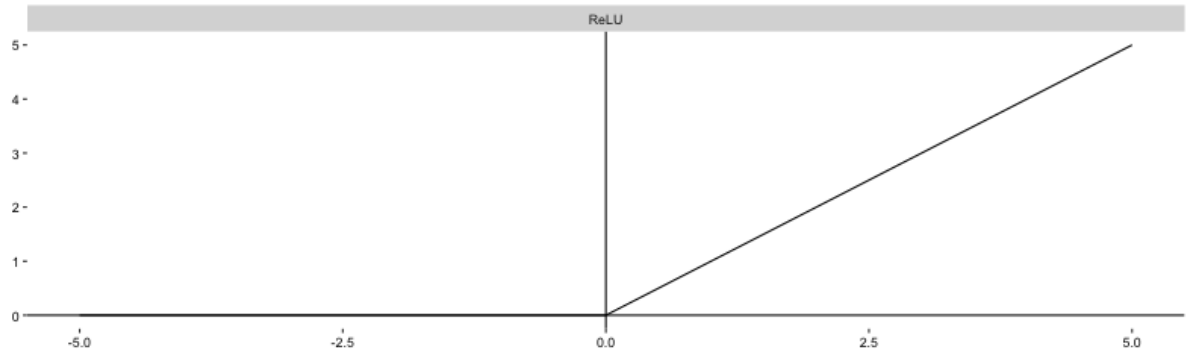


Figura 4: Função Unidade Linear Retificado

A grande vantagem é que esta é uma função fácil de ser otimizada devido à sua semelhança com a função linear. Esta semelhança faz com que a derivada em toda função permaneça grande sempre que a função está ativa, ou seja, o ∇g é grande e constante. Quando uma unidade está ativa, a derivada primeira da função é 1 e, 0, caso contrário . Então, neste sentido, é muito mais útil para o processo de aprendizagem utilizar a direção do vetor ∇g do que os efeitos de segunda ordem.

A ReLU é aplicada sobre a transformação affine:

$$h = g(W^T x + b)$$

Então, o processo de aprendizagem acontece quando os parâmetros da transformação affine são inicializados. Se o parâmetro b for tiver um valor inicial pequeno, como 0,1, a função ReLU será inicialmente ativada para boa parte dos valores de entrada durante o treinamento.

Se a função de ativação de um elemento é 0, o ReLU não é capaz de submeter este elemento ao processo de aprendizagem via método de aprendizagem do gradiente. Então, para contornar esse problema, é necessário usar generalizações dessa função para garantir que o ∇ seja recebido de todas as direções. O generalizador mais utilizado é o unidades*maxout*.

A ideia do unidades *maxout* é dividir z em i grupos de k valores. Então, cada unidade *maxout* é representada pelo elemento máximo de cada grupo. Esta função é dada por:

$$g(z)_i = \max_{j \in \mathbb{G}^{(i)}} z_j$$

Onde $\mathbb{G}^{(i)}$ indica a entrada do i-ésimo grupo, $\{(i-1)k+1, ..., ik\}$.

O regularizador unidade *maxout* faz com que as divisões em grupos se aproximem de uma função linear, facilitando o aprendizado da função e garantindo que o gradiente responda a múltiplas direções do espaço de entrada x. Com k suficientemente grande, uma unidade *maxout* pode aprender a aproximar qualquer função convexa [3] com uma fidelidade arbitrária. Uma unidade *maxout* pode também ser interpretada como uma função de ativação.

2.5.2 Sigmoide logística

Esta é uma função não linear que transforma um valor de entrada real no intervalo entre [0,1] através da forma:

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

Pode-se notar por esta função que, quanto menor o valor de z, mais a função se aproximará de 0. E, quanto maior o valor de z, mais a função se aproximará de 1.

[3] Uma função $f : I \rightarrow \mathbb{R}$ é uma função convexa se para todo $x, y \in I$ e todo $\lambda \in [0, 1]$ valer:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda) f(y)$$

A função sigmoide foi muito utilizada devido a sua facilidade de interpretação como resultado da taxa de ativação de um neurônio: Se nenhuma unidade está ativa, ($\sigma(z) = 0$). E se todas as unidades estão ativas ($\sigma(z) = 1$). Porém existe algumas propriedades indesejáveis que deixaram essa função cair em desuso, como o fato de que esta não é uma função centrada em zero, o que dificulta o cálculo do gradiente no algoritmo de *back-propagation*.

2.5.3 Tangente Hiperbólica

A função de tangente hiperbólica (tanh) é não linear, em que transforma um valor real de entrada no intervalo [-1,1]. Esta função é descendente da função sigmoide, na forma:

$$tanh(z) = 2\sigma(2z) - 1$$

$$tanh(z) = \frac{2}{(1 + e^{-2z})} - 1$$

Apesar da função tanh ser não linear, ela possui a vantagem de ser centrada em zero, o que torna o aprendizado através de métodos baseados em gradientes mais fácil do que a sigmóide logística.

A Figura 5 mostra as funções Sigmóide e Tanh.

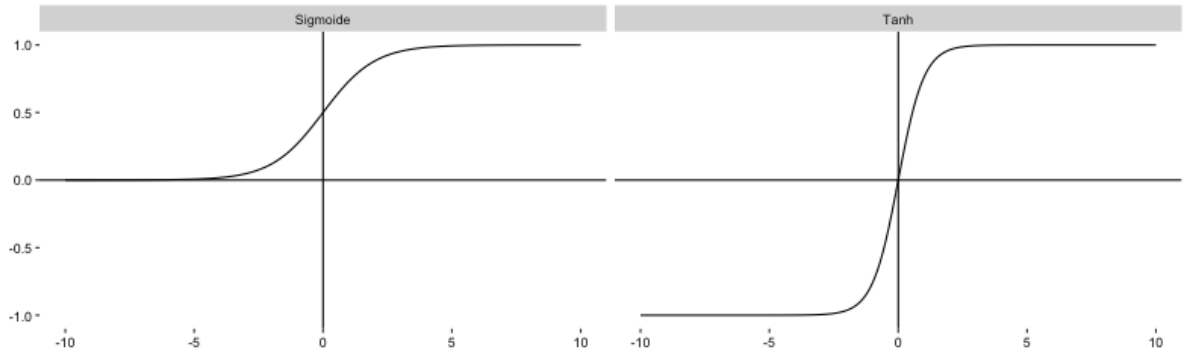


Figura 5: Função Sigmóide à esquerda e Função Tangente Hiperbólica à direita

2.6 Validação Cruzada e K-folhas

Um dos principais objetivos das técnicas de validação cruzada é selecionar o melhor modelo dentro de um conjunto de modelos. Em um conjunto de modelos com diferentes valores de um parâmetro, busca-se aquele com a melhor qualidade, ou seja, com a menor taxa de erro. A busca do melhor modelo é feita através de técnicas de reamostragem, que inclui a retirada de amostras, treinamento e validação, em um conjunto de treinamento.

A validação cruzada pode ser usada para estimar a taxa de erro associado a um método de estimação para avaliar a performance do ajuste do mesmo ou selecionar os melhores parâmetros a serem utilizados.

A abordagem de validação envolve, basicamente, dividir o conjunto de treinamento em dois conjuntos: um de treinamento e outro de validação. Então, o modelo é ajustado no conjunto de treinamento e o ajuste do modelo nos dados de validação são usados para avaliar a qualidade do mesmo através da comparação dos valores preditos com a resposta desejada.

Um dos métodos de validação cruzada é o método de K-folhas. Segundo Tibshirani et al. (2013), essa técnica consiste em dividir aleatoriamente o banco de dados em k grupos, ou folhas, de tamanhos aproximadamente iguais. Então, a i-ésima folha é tratada como o conjunto de validação e as outras $k-1$ folhas como treinamento, sendo que a cada iteração, um grupo diferente é tratado como validação e o $k-1$ grupos restantes como treinamento. Então, a cada iteração o modelo é estimado com os dados de treinamento e os valores preditos são calculados com os dados de validação. A medida de avaliação desse processo é dado pela média da validação cruzada sobre todas as k combinações.

A estimativa para o risco da função de classificação f_k de x para o parâmetro γ_j através do método de k-folhas é dado por:

$$\hat{R_{CV}} := \frac{1}{K}\sum_{k=1}^{K}\frac{1}{|bloco\ k|}\sum_{(\tilde{x}\tilde{y}\in block\ \ k)} L(\tilde{y}, f_k(\tilde{x}, \gamma_j))$$

E esse processo é repetido para todos os valores do parâmetro $\gamma_1, ..., \gamma_m$. Então o modelo selecionado será aquele que possui o valor γ^* que minimiza a estimativa do risco.

2.7 Arquitetura de uma rede

A arquitetura de uma rede se refere à organização da estrutura da rede, ou seja, quantidade de camadas, conexões, parâmetros e unidades de aprendizado. As redes são organizadas em camadas, que ganham um formato de cadeia. Em cada camada é aplicada a função de ativação. Normalmente, os valores de saída das camadas servem de entrada para a camada seguinte e cada camada recebe uma função de ativação, por exemplo, para a primeira camada:

$$h^{(1)} \ = \ g^{(1)}\left(W^{(1)T}x_0 + b^{(1)}\right)$$

Para a i-ésima camada tem-se:

$$h^{(i+1)} \ = \ g^{(i+1)}\left(W^{(i+1)T}x_i + b^{(i+1)}\right)$$

Uma das questões que deve ser considerada é a quantidade de camadas que uma rede deve possuir e a quantidade de nós em cada uma delas. Em se tratando de aprendizado, até uma rede com apenas uma camada escondida é capaz de se ajustar a um treinamento. Mas redes mais profundas são capazes de usar menos unidades e parâmetros por camada e possuem uma capacidade de generalização melhor para observações que não participaram do treinamento da rede. Porém, quanto mais camadas, maior a dificuldade em otimizar o algoritmo de treinamento. A arquitetura ideal de uma rede deve ser encontrada através do monitoramento do modelo treinado e o conjunto de validação.

Além da profundidade da rede, uma questão importante a ser tratada são as conexões entre as camadas. Uma conexão padrão é realizada por meio da transformação linear descrita por W, onde cada unidade de entrada está ligada a uma unidade de saída. Porém, muitas redes especializadas estão sendo desenvolvidas para diminuir a quantidade de conexões entre as camadas, de modo que cada unidade na camada de entrada é ligada apenas a um pequeno subconjunto de unidades na camada de saída. Essas estratégias para reduzir o número de ligações, reduzem o número de parâmetros e, consequentemente, o poder computacional requerido para treinar uma rede neural. As redes convolucionais são um exemplo de arquitetura que utilizam dessa estratégia de redução do número de conexões.

Os principais tipos de arquitetura de rede neural são as redes recorrentes e *feedfoward*. Mas arquiteturas específicas têm sido desenvolvidas para diferentes tarefas. Por exemplo, as redes convolucionais são derivadas da arquitetura *feedfoward* e são muito eficientes em reconhecimento de imagem e tarefas de visão computacional (*computer vision*). Arquiteturas *feedfoward* podem ser generalizadas para redes recorrentes e serem usadas para dados sequenciais.

2.7.1 Redes Recorrentes

A arquitetura recorrente se refere às redes onde existe pelo menos um *loop* de realimentação da rede. Essas redes podem processar como entrada uma sequência de elementos, $x^{(1)}, ..., x^{(\tau)}$, mantendo em suas unidades escondidas um vetor que, implicitamente, contém informações sobre a história de todos os últimos elementos da sequência. Esse tipo de rede usa uma técnica de compartilhamento de parâmetros em diferentes parcelas do modelo. O compartilhamento de parâmetro permite ajustar e aplicar o modelo para observações de diferentes formas e, assim, aumentar a capacidade de generalização do mesmo.

Para entender melhor, se a rede tivesse parâmetros separados para cada valor no intervalo de tempo, não seria possível generalizar uma sequência que não fosse vista durante o treinamento e nem generalizar para caso a mesma sequência ocorresse em

posições diferentes no vetor de entrada, por exemplo, considerando as seguintes frases: "estamos no ano de 2016"e "2016 é o ano que estamos"e colocando como objetivo da rede recorrente identificar o ano presente em cada frase, "2016"será a informação relevante que a rede terá que reconhecer. Mas essa informação aparece em diferentes posições em cada uma das frases. O treinamento de uma rede *feedfoward* para realizar tal tarefa ocorreria em diferentes pesos para cada uma das entradas. Com o compartilhamento de pesos, essa tarefa se torna muito mais simples. Então, tal partilha é particularmente importante quando um pedaço específico de informação pode ocorrer em várias posições no interior da sequência.

Portanto, as redes recorrentes são muito eficientes em trabalhos que envolvem entradas sequenciais, como reconhecimento de fala e texto. As redes recorrentes são muito poderosas e dinâmicas, mas são redes complicadas de serem treinadas com o algoritmo de *back-propagation*. Graças aos avanços nos estudos de sua arquitetura e forma de treiná-las, essas redes são muito eficazes em problemas como prever o próximo caractere no texto ou uma palavra seguinte em uma sequência.

2.7.2 Redes *Feedfoward*

A arquitetura *Feedfoward* se refere à uma rede onde uma informação é apresentada através de um vetor x na camada de entrada, então a informação é avaliada por meio de cálculos intermediários utilizados para definir a função f e, finalmente, tem-se a saída y. Este tipo de arquitetura é acíclico e não existem *loops*, ou seja, todas as saídas de uma camada servem de entrada apenas para a camada seguinte.

O objetivo de uma rede *feedfoward* é aproximar uma função f^*, tal que o classificador $y = f^*(x)$ relaciona uma entrada x_i para uma categoria y_i. Sendo assim, uma rede *feedfoward* define mapas $y = f(x;\theta)$ e estima os valores do parâmetro θ para a melhor aproximação da função f.

As redes *feedfoward* compõem diferentes funções em uma mesma estrutura de camadas. Por exemplo, um modelo com 3 funções $f^{(1)}$, $f^{(2)}$ e $f^{(3)}$ conectados em cadeia são representados na forma $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$, onde $f^{(1)}$ representa a primeira camada, $f^{(2)}$ a segunda camada e $f^{(3)}$ a terceira camada. A quantidade de camadas da rede se refere à profundidade da mesma.

A ultima camada da rede é chamada de camada de saída. Durante o treinamento, a função $f(x)$ é estimada por $f^*(x)$. Então os dados de treinamento fornecem o erro aproximado por $f^*(x)$ e avaliado em diferentes observações de treinamento. Para o treinamento, cada observação x é acompanhada de uma classe $y = f^*(x)$ então, a ideia dos dados de treinamento é que eles dizem exatamente o que a camada de saída deve fazer em cada

observação x para que o valor da classificação seja o mais próximo possível do valor de y.

O comportamento das outras camadas, as intermediárias, não são especificados diretamente pelos dados de treinamento, como acontece na camada de saída. O algoritmo de aprendizagem que decide o comportamento de cada camada para implementar uma aproximação de f^* e produzir a saída desejada. Então, as camadas intermediárias são as chamadas camadas escondidas.

De acordo com Goodfellow, Yoshua, Courville (2016), o princípio geral de uma rede *feedfoward* é melhorar os modelos aprendendo características. Uma das maneiras de se justificar o uso da rede *feedfoward* é iniciando o pensamento com um modelo linear. Os modelos lineares são de fácil compreensão e implementação, mas possui limitações como não ser capaz de entender interações entre variáveis, por exemplo.

Para resolver esse problema, as funções lineares podem ser representadas por funções não lineares de x. Pode-se aplicar o modelo linear em uma transformação de x dada por $\phi(x)$, onde ϕ é uma transformação não linear de x (essa transformação pode ser aplicada via kernel também) e pode ser interpretado como uma transformação que fornece características de x.

A estratégia do aprendizado profundo é de estimar ϕ para que se possa estimar:

$$y \;=\; f(x;\theta,w) \;=\; \phi(x,\theta)^T w$$

Agora, o parâmetro θ é usado para estimar ϕ a partir de uma determinada **classe de funções** e os parâmetros w, que são os chamados pesos, direcionam $\phi(x)$ para a saída desejada. Então, para treinar uma rede *feedfoward* requer a definição da arquitetura da rede, função de custo, forma das unidades de saída. Para as camadas escondidas, tem-se a definição da função de ativação que será usada para computar os valores nas camadas escondidas e, qual método de aprendizagem via gradiente será usado.

Quanto mais camadas escondidas uma rede possui, mais complexo é a otimização dos métodos de aprendizagem e computação dos valores para mapear as entradas e saídas de uma rede. As redes *feedfoward* com muitas camadas escondidas são chamadas de rede de aprendizado profundo.

Dentro das redes *feedfoward* de aprendizado profundo, ou *deep learning*, arquiteturas específicas foram desenvolvidas para realizar tarefas de alta complexidade, como reconhecimento de imagem. Essas são as chamadas redes convolucionais, que possui uma arquitetura própria, com tipos de camadas e conexões diferenciadas das redes tradicionais. Essas redes serão estudadas com mais detalhes abaixo.

2.7.3 Redes Convolucionais

Segundo Goodfellow, Yoshua, Courville (2016), as redes convolucionais, ou CNN (*Convolutional Neural Network*), são "redes especializadas em processamento de dados com topologia semelhante a grades. Por exemplo, dados de imagens possuem sua representação em uma grade 2D de pixels ou dados de séries temporais que podem ser tomados como amostras de grade 1D de um período regular de tempo. Além disso, as CNN's podem ser aplicadas em diversos tipos de dados com diferentes dimensões".

O nome convolucional se refere a compilação matemática realizada neste tipo de rede, ou seja, redes convolucionais são simplesmente redes neurais que usam a convolução no lugar da multiplicação de matrizes em pelo menos uma de suas camadas. Sendo assim, pode-se definir convolução como:

Definição 2.4 *Convolução é definido como um operador linear aplicado a duas funções cujo objetivo é medir uma terceira área subentendida pela sobreposição das mesmas em função do deslocamento existente entre elas. O operador de convolução pode ser contínuo ou discreto e é denotado por* $*$*:*

$$s(t) = (x * w)(t) = \int x(a)w(t-a)d(a)$$

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Nas redes convolucionais, o primeiro argumento (x) se refere as entradas da rede, o segundo argumento (w) se refere ao kernel e as saídas se referem aos mapas de características.

Normalmente se usa a convolução sobre mais de um eixo por vez. Por exemplo, uma imagem bidimensional com as entradas denotadas por I, é melhor se usar um kernel K que também seja bidimensional. Então:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Como a convolução é comutativa, pode-se reescrever $S(i,j)$ como:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Ainda segundo Goodfellow, Yoshua, Courville (2016), as camadas convolucionais possuem 3 propriedades que tornam o processamento dos dados de entrada mais efici-

entes, sendo elas: interações esparsas, compartilhamento de parâmetros e representação equivariante.

Tradicionalmente, as camadas das redes neurais utilizam multiplicação de uma matriz por uma matriz de parâmetros com um parâmetro separado descrevendo a interação entre cada unidade de entrada e saída, ou seja, cada unidade de saída interage com cada unidade de entrada. Entretanto, as redes convolucionais utilizam as interações esparsas, o que torna o kernel menor do que as entradas. Supondo, por exemplo, o processamento de uma imagem com m entradas e n saídas, então a multiplicação de matrizes é dada por $m \times n$ parâmetros. Se existe uma limitação no número de conexões de cada saída para k, onde $k < m$, então a abordagem das conexões esparsas requerem apenas $k \times n$ parâmetros, o que implica em uma redução considerável no número de conexões.

Em uma rede neural tradicional, cada elemento de uma matriz de pesos é usada exatamente uma vez até a camada de saída, ou seja, esse peso é multiplicado uma vez por um elemento de entrada e nunca mais é revisitado. O compartilhamento de parâmetros se refere a utilizar o mesmo parâmetro para mais de uma função no modelo. A ideia de compartilhamento de parâmetros é chamada também de pesos atados, o que significa que o valor de um peso aplicado em uma entrada está amarrado ao valor do mesmo peso aplicado em outro lugar. Nas redes convolucionais, cada elemento do kernel é usado em todas as posições de entrada e o compartilhamento de pesos usado pela operação de convolução significa que ao invés de o modelo aprender um conjunto separado de parâmetros para cada local, aprende-se apenas um conjunto. Essa propriedade não interfere no tempo de execução da propagação da rede, mas reduz os requisitos de armazenamento do modelo de m para k parâmetros. Considerando que k é muito menor que m e que n normalmente tem o mesmo tamanho que m, então k é praticamente insignificante quando comparado a $m \times n$. Por isso as redes convolucionais são significativamente mais eficientes com relação a requerimento de memória e aprendizado estatístico.

O compartilhamento de parâmetros é a causa da terceira propriedade das redes convolucionais, a equivariância. Uma função é equivariante se uma mudança na entrada modifica a saída da mesma forma, ou seja, uma função $f(x)$ é equivariante a uma função g se $f(x) = f(g(x))$. No caso convolucional, se g é qualquer função que traduz a entrada, então a função de convolução é equivariante a g. Essa propriedade é especificamente eficiente para que a rede seja invariante a aspectos irrelevantes da imagem, tais como posição, tamanho, rotação, iluminação, entre outros.

Por ser uma rede diferenciada, normalmente a arquitetura de uma rede convolucional pode ter tipos de camadas diferentes, como as camadas convolucionais, que foram vistas acima, as camadas totalmente conectadas, que são exatamente iguais as das redes neurais comuns e as camadas de agrupamento que serão especificadas a seguir.

As camadas de agrupamento, ou as chamadas *pooling layers*, possuem a função de reduzir progressivamente o tamanho espacial da representação para reduzir a quantidade de parâmetros e computação da rede e, portanto, para também controlar o super ajuste. As funções de agrupamento, ou de *pooling*, são funções que substituem as saídas da rede em um certo local por um valor que seja representativo daquele subconjunto local. Por exemplo, a operação de *max pooling* que representa o valor máximo dentre uma vizinhança retangular. Em todas as operações, o *pooling* também ajuda para que a representação seja aproximadamente invariante a pequenas traduções na entrada, ou seja, pequenas variações não alteram a maioria das saídas, já que estas são baseadas em valores representativos. A propriedade de invariância pode ser muito interessante se o objetivo é saber se existe uma certa característica na imagem do que onde ela está. Por exemplo saber se existe um rosto sem dar importância a localização desse rosto.

Como o *pooling* sumariza a resposta em uma janela de elementos, é possível usar menos unidades de agrupamento por meio de estatísticas descritivas para o *pooling* em janelas com distâncias de tamanho k pixels, ao invés de 1 pixel. Essa redução melhora a eficiência computacional da rede, porque a próxima camada possui k entradas a menos para processar. Além de aumentar a eficiência computacional, diminui, também, a quantidade de memória necessária para armazenar os parâmetros da rede.

3 Estudo de Caso

3.1 Metodologia

Para este trabalho, foram criadas 3 estruturas de redes neurais, sendo uma com apenas uma camada e duas redes neurais profundas, para aplicação em classificação de imagens. A criação do banco de dados e os métodos utilizados estão descritos nas próximas seções.

3.1.1 Material

O banco de dados utilizado neste trabalho se refere as imagens que foram classificadas como espirais à direita e à esquerda. Para o treinamento da rede, foram geradas 1.262 imagens, sendo 630 como espirais à direita e 632 à esquerda. Além disso, foram geradas mais 254 imagens diferentes para compor o banco de teste. Esse banco de dados possui 217 imagens de cada classe, espirais à esquerda e à direita.

As imagens foram geradas através do software R e foram empregadas duas funções principais para gerar as imagens espirais. Para gerar uma série de imagens, os parâmetros b e k se tornaram variáveis, onde k=[1,10] e b se tornou uma sequência de 1 até 76 com intervalos de tamanho 5. Para cada combinação entre os valores de k e b, tem-se os seguintes resultados:

θ se refere ao ângulo das espirais e este varia de 0 até $k \times 2\pi$ com intervalos de tamanho $b \times k$. Assim, R também é um vetor de tamanho $(b \times k) - 1$, onde $R = 0, 98^{(1)}, ..., 0, 98^{(b \times k)-1}$. Então para cada valor de R eθ:

$$X = R \times cos(\theta)$$

$$Y = R \times sen(\theta)$$

Para criar espirais na direção oposta, os ângulos considerados são dados por $-\theta$.

Outro método utilizado para gerar imagens em forma de espirais para compor o banco de teste é dado por:

Considerando θ como uma sequência que varia de 0 até $b \times 2\pi$ com intervalos de tamanho $2\pi/72$, sendo que $b = [1, 30]$, então para cada valor de θ, tem-se que:

$$R = \theta^{(\frac{1}{2})}$$

$$X = R \times cos(\theta)$$

$$Y = R \times sen(\theta)$$

Após gerar essas imagens, foram selecionadas 1.262 que de fato possuíam o formato de espiral para compor o banco de espirais, tanto à esquerda quanto à direita. As figuras 6 e 7 mostram alguns exemplos de espirais geradas a partir desses métodos.

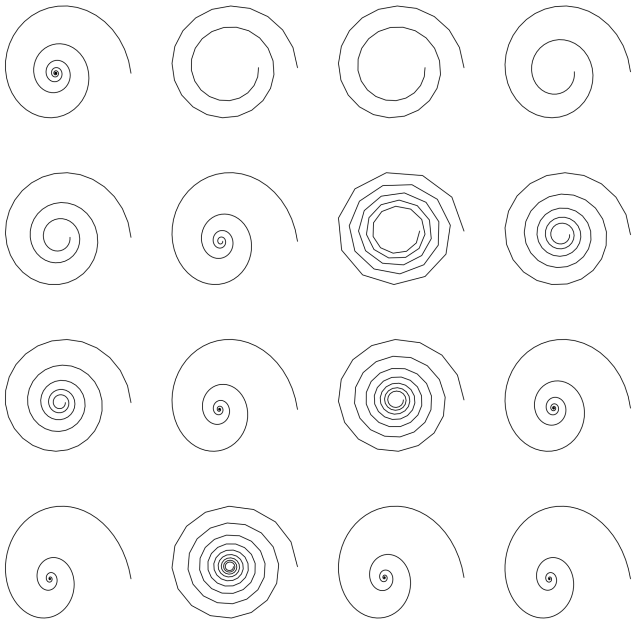


Figura 6: Exemplo de imagens de espiral à esquerda

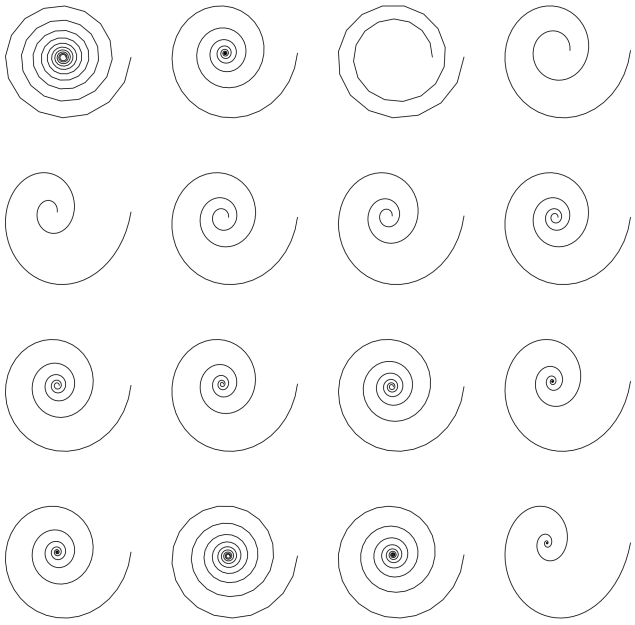
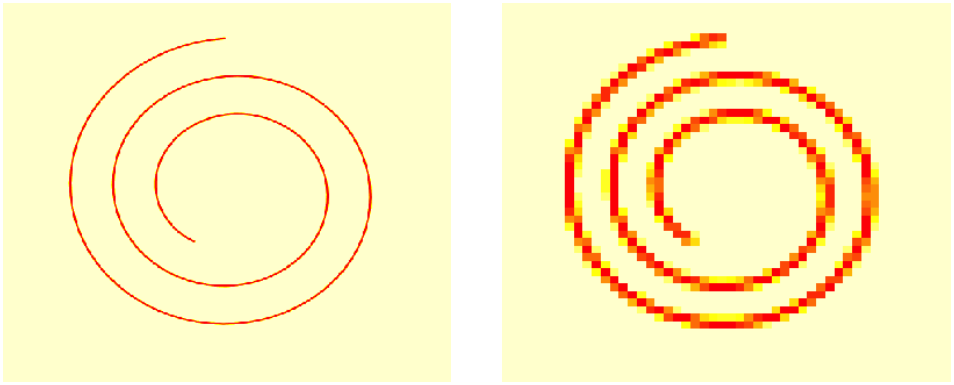


Figura 7: Exemplo de imagens de espiral à esquerda

Como já foi dito, as imagens que servem de entrada para a rede possuem um formato de matriz de pixels. A qualidade das imagens geradas possuem um formato de uma matriz com dimensão 480x480 pixels. Na prática, uma única imagem de 480x480 pixels seria representada por uma entrada de vetor de 230.400 pixels, sendo que o processamento de 4156 imagens, cada uma com dimensão 480x480, seria inviável devido aos recursos computacionais limitados. Por isso, as imagens foram redimensionadas para uma matriz de dimensão 50x50, o que levaria a um vetor de pixels de tamanho 2.500. A imagem a seguir ilustra uma mesma imagem processada pelo software R em ambas as dimensões.



(a) (b)

Figura 8: Representação da imagem de acordo com da qualidade definida. (a): À esquerda tem-se uma imagem com qualidade original representada por uma matriz de dimensão 480x480. (b): à direta apresenta-se uma imagem com qualidade reduzida, representada por uma matriz de dimensão 50x50.

Apesar da redução da qualidade da imagem apresentar uma redução significativa do detalhamento da mesma, não espera-se que isto seja um empecilho para um treinamento eficiente da rede neural.

3.1.2 Métodos

Antes da rede neural de aprendizado profundo, foram ajustadas três redes neurais MLP, ou seja, redes com uma única camada escondida. Essas redes receberam 126, 256 e 504 neurônios na sua camada escondida e 2 neurônios na camada de saída. Além disso, foi utilizada a função de ativação ReLU com o parâmetro de *dropout* e taxa de aprendizado de 0,1. Por se tratar de um problema de classificação, a métrica utilizada para avaliação do modelo foi a acurácia e a função de custo padrão é a função L^{0-1}.

A arquitetura utilizada para o treinamento da rede neural profunda foi a feedfoward com 5 camadas, sendo uma camada de entrada, três camadas escondidas do tipo totalmente conectada e uma cada de saída. A primeira camada totalmente conectada possui 128 neurônios, a segunda 64 neurônios e a ultima apenas 2, cada uma referente a uma

classe possível para classificação das imagens. Nas duas primeiras camadas totalmente conectadas, foi utilizada a função de ativação ReLU. O diagrama a seguir representa a arquitetura da rede.

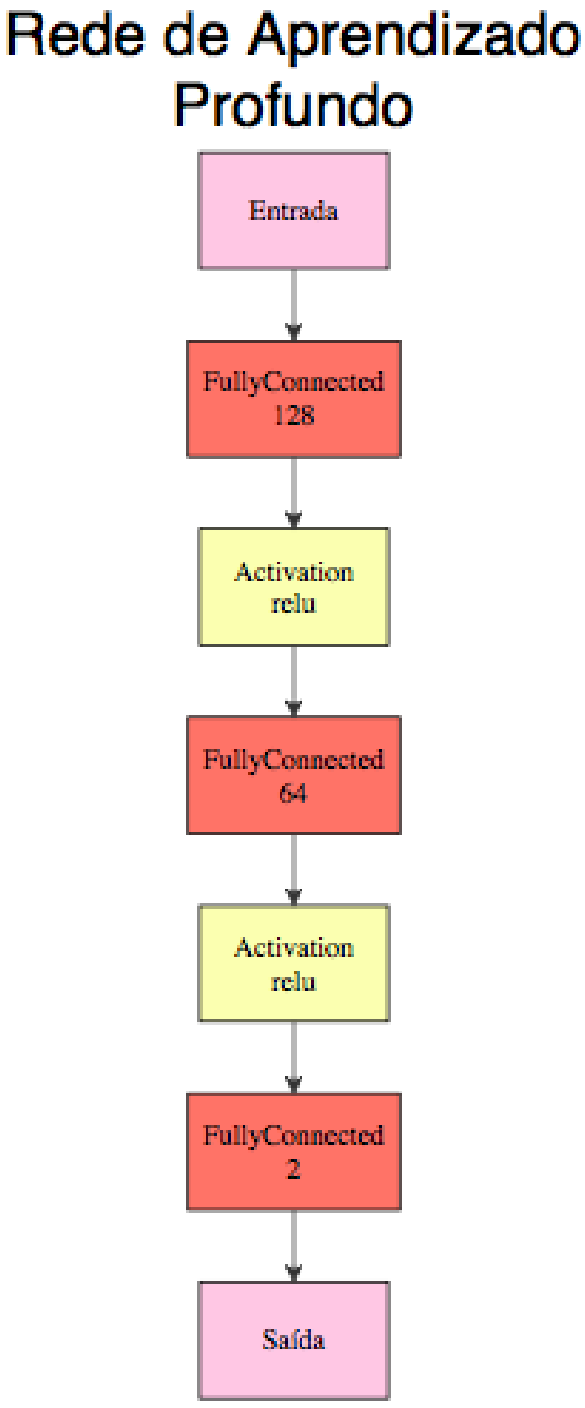


Figura 9: Arquitetura da rede neural com três camadas

Como já foi mencionado, a inicialização dos parâmetros pode influenciar na velocidade do treinamento da rede. Então, para o processo de treinamento da rede, foram

consideradas a inicialização dos pesos através das seguintes distribuições:

$$\theta_0 \sim N(0, 0, 1)$$

$$\theta_0 \sim N(0, 1)$$

$$\theta_0 \sim U[-0, 1, 0, 1]$$

$$\theta_0 \sim U[-1, 1]$$

Outro detalhe referente a aprendizagem é que foi utilizado o processo de aprendizado via gradiente, onde a taxa de aprendizagem foi de 0,1 e o decaimento de pesos em 0,0005, onde uma pequena quantidade de decaimento de pesos ajuda na aceleração do processo de aprendizagem.

Por ser um problema de classificação, a métrica de avaliação de construção do modelo nos dados de treinamento foi a acurácia. Por ser um modelo de convergência rápida, foram considerados uma quantidade fixa iterações para comparação entre os modelos, sendo o valor fixo de 20 iterações. Para avaliar a performance do modelo nos dados de teste, a métrica utilizada foi taxa de erro e o modelo mais eficiente foi aquele com a melhor capacidade de generalização em novos dados, ou seja, aquele que possui a menor taxa de erro.

Após selecionar o modelo mais eficiente, foi realizada uma validação cruzada pelo método de k-folhas com 3 grupos, para ajustar os parâmetros de regularização de decaimento dos pesos, wd. Então novamente foi testada a eficiência do modelo com relação a taxa de erro.

Além do modelo *feedfoward* e MPL detalhados acima, foi ajustada uma CNN com duas camadas convolucionais e duas camadas totalmente conectadas. Como detalhado na Figura 10, tem-se uma estrutura onde a primeira camada convolucional possui 20 kernels de tamanho 5x5 e janelas com uma 1 unidade de distância, isto é, a distância entre dois mapas de kernels vizinhos. Essa camada possui uma camada de *pooling* pelo máximo, a função de ativação ReLU e uma camada de *dropout*. A segunda camada possui 50 kernels de 5x5 e janelas com uma unidade de distância. Além disso, também possui a função de ativação ReLU, uma camada de *pooling* pelo máximo e uma camada *dropout*. A primeira camada totalmente conectada possui 500 neurônios, acompanhada da função de ativação ReLU e a última camada conectada possui 2 neurônios, onde cada um se refere a uma possível classe dentro do conjunto de classificação.

Os resultados foram gerados por meio do pacote MXNet, R *package* versão 0.7. O processador utilizado foi um CPU 1,8 GHz Intel Core i5 com memória 4 GB 1600 MHz DDR3 e gráficos intel HD Graphics 4000 1536 MB.

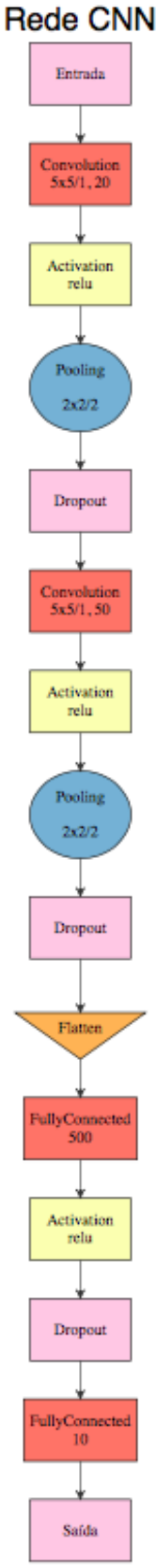


Figura 10: Arquitetura da rede convolucional com 2 camadas convolucionais e duas camadas totalmente conectadas

3.2 Resultados

Nesta seção são apresentados os resultados para as duas arquiteturas de redes neurais que foram ajustadas, a MLP, que possui apenas uma camada escondida e a rede *feedfoward* com 3 camadas escondidas.

3.2.1 Rede *Multilayer Perceptron*

Uma justificativa de se usar as redes de aprendizado profundo é que essas tem a capacidade de processar melhor dados em alta dimensão, além de ser mais eficientes para mapear entradas em saídas. Assim, as redes neurais com uma única camada escondida não possuem o desempenho desejado quando se trata em identificar características em dados de alta dimensão, no caso de reconhecimento de imagens. Os resultados estão sumarizados na tabela 1 a seguir:

Tabela 1: Desempenho das redes MLP

Modelo	Acurácia	Taxa de erro	Tempo
MLP 126	35,23%	50%	15s
MLP 256	35,07%	50%	22s
MLP 504	36,16%	50%	102s

Como pode-se perceber através da Tabela 1, a acurácia do modelo, independente do número de neurônios no modelo, se deu em torno de 36%. Ou seja, a capacidade do modelo MLP de classificar corretamente as observações foi de 36%. Nesse caso em específico, como a rede não conseguiu aprender os padrões dos dados de entrada, as três redes MLP classificaram todas as novas observações como rótulo "1"(espirais à direita), consequentemente a taxa de erro foi de 50%.

A Figura 11 representa o gráfico de desempenho das 3 redes ao longo das 50 iterações.

Pode-se perceber as três trajetórias atingem os maiores valores antes das 10 primeiras iterações, sendo que a MLP256 obteve a acurácia mais alta com relação às outras duas redes em análise, sendo o maior valor encontrado 50,16%. O valor mínimo encontrado foi de 30,86% na rede MLP504. Com um desvio padrão de 0,0539 e coeficiente de variação em 14,92%, a rede MLP504 obteve a menor variação em torno da acurácia média, ou seja, os valores da acurácia da rede durante as 50 iterações tiveram uma variação de aproximadamente 15%. os valores do coeficiente de variação das outras duas redes foram muito próximos, 15,05% e 16,27%, para as redes MLP256 e MLP504, respectivamente.

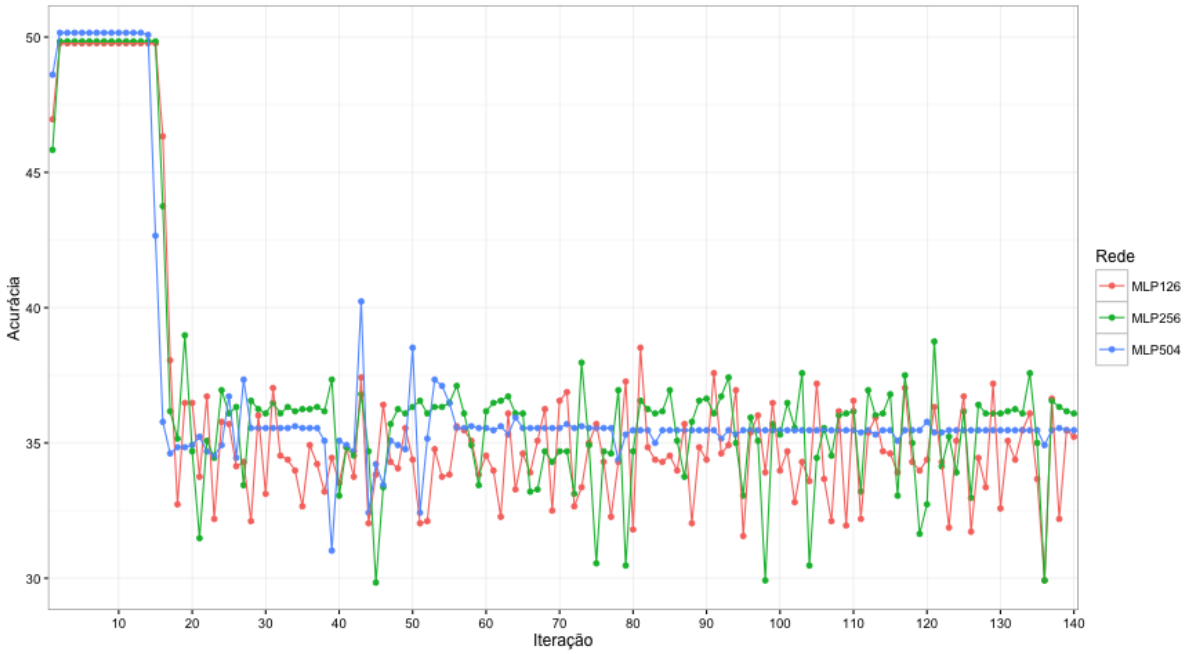


Figura 11: Desempenho das redes MLP ajustadas

3.2.2 Rede *Feedfoward*

Com relação aos modelos ajustados com a arquitetura de 5 camadas, sendo elas 3 camadas totalmente conectadas, espera-se que esses modelos obtenham um desempenho melhor do que o encontrado na estrutura MLP. Os resultados encontrados para essas redes estão sumarizados na tabela 2.

Tabela 2: Desempenho das redes *Feedfoward*

Modelo	Inicialização dos parâmetros	Acurácia	Taxa de erro	Tempo
Feedfoward-3	N(0, 0,1)	100%	46,42%	6s
Feedfoward-3	N(0, 1)	100%	47,61%	3s
Feedfoward-3	U[-0,1, 0,1]	100%	41,66%	5s
Feedfoward-3	U[-1, 1]	100%	27,77%	3s

Como pode-se perceber através da Tabela 2 acima, o desempenho das redes *feedfoward* com 3 camadas escondidas foi muito melhor do que o desempenho nas redes MLP, que por sua vez foi capaz de aprender os padrões das imagens de entrada e tornar o conhecimento adquirido disponível para uso, ou seja, para classificar novas imagens. A rede com a melhor capacidade de classificação foi com a inicialização dos parâmetros através de uma distribuição U[-1, 1]. Essa rede obteve uma taxa de erro de 27,77%, ou seja, esse modelo classifica, aproximadamente 28% das observações no banco de teste incorretamente. A rede com os parâmetros inicializados através de uma distribuição N(0,1) teve a maior

taxa de erro, classificando de forma incorreta 47,61% das observações no banco de teste. As redes com os parâmetros de inicializados através das distribuições N(0, 0,1) e U[-0,1, 0,1] também tiveram desempenhos extremamente altos com relação ao treinamento da rede, porém com taxas de erro relativamente altas 46,42% e 41,66%.

A Figura 12 representa a trajetória do desempenho das 4 redes ao longo das 20 iterações.

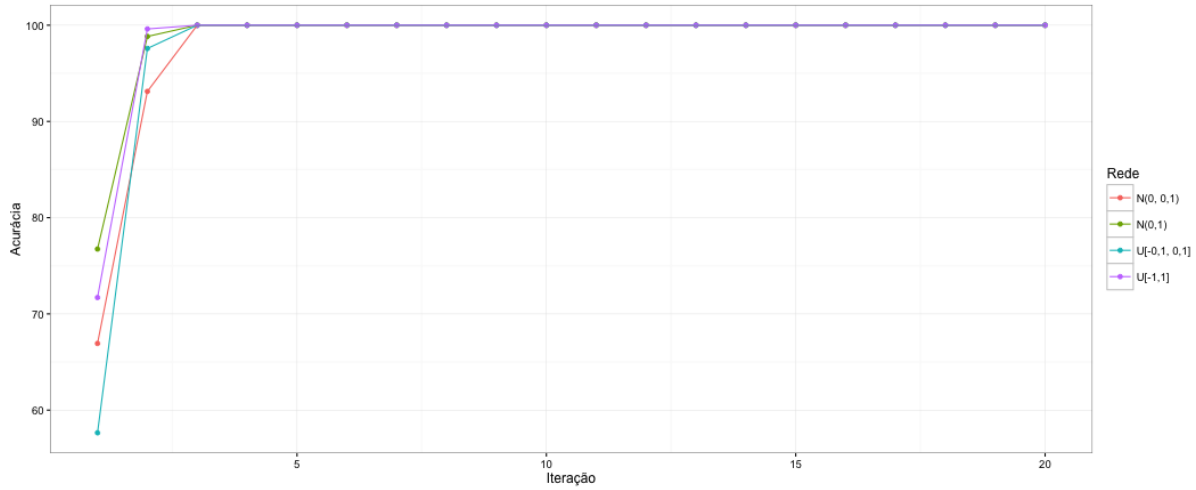


Figura 12: Desempenho das redes *feedfoward* ajustadas

É possível perceber através da Figura 12 que as 4 trajetórias se comportam de forma muito semelhante e todos os 4 modelos possuem uma convergência rápida em direção a acurácia 1. Com menos de 3 iterações, todos os modelos convergem para o valor máximo, conseguindo classificar corretamente todas as observações do banco de treinamento. Apesar desses modelos serem extremamente eficientes para o treinamento da rede, esses deixaram a desejar com relação a capacidade de generalização dos mesmos, ou seja, os modelos não foram capazes de se ajustar bem em novas observações. Esse fenômeno se da por causa do sobre-ajuste, ou seja, sem um parâmetro de regularização, o modelo se ajustou muito bem especificamente aos dados de treinamento.

Para tentar diminuir o sobre-ajuste e aumentar a capacidade de generalização desses modelos, foi acrescentado um parâmetro de regularização de decaimento dos pesos. Então, para selecionar o melhor valor do parâmetros, foi realizada uma validação cruzada por meio do k-folhas, com $k = 3$. Esse método foi aplicado no modelo mais eficiente encontrado anteriormente, ou seja, o modelo *feedfoward* com três camadas escondidas e parâmetros gerados a partir de uma distribuição U[-1,1]. Os resultados estão dispostos na Tabela 3.

Pela Tabela 3, nota-se que durante o processo de validação cruzada, praticamente em todos os modelos ajustados os erros de treinamento e predito são 0. Como a acurácia dos modelos converge para 1, naturalmente o mesmo será capaz de classificar corretamente

Tabela 3: Seleção de modelos através da validação cruzada com 3-folhas para os valores do parâmetro de regularização wd

wd	Erro de treinamento	Erro predito	Erro de teste
0,0001	0	0	34,12%
0,001	0	0	35,71%
0,002	0	0	35,31
0,003	0	0	33,73%
0,004	0	0	33,33%
0,005	0	0	33,33%
0,006	0	0	15,87%
0,007	32,64%	17,27%	26,58%
0,008	0	0	27,77%
0,009	0	0	27,38%
0,01	0	0	27,38%

todas as observações no banco de treinamento. Com relação ao erro predito, como as imagens do banco inicial de treinamento foram geradas por uma mesma função, existem muitas imagens que são semelhantes. Como o banco de validação é uma reamostragem a partir do banco de treinamento, a rede não possui grandes desafios para classificar as imagens no banco de validação pelo fato de que a rede viu imagens semelhantes durante o treinamento. O mesmo não acontece com o grupo de teste pelo fato de que as imagens que compõem esse banco de dados são espirais geradas por funções diferentes, ou seja, as semelhanças com o grupo de treinamento são mais superficiais. Por isso rede encontra uma dificuldade maior em classificar essas imagens, e, consequentemente, a taxa de erro para o banco de testes é maior.

Como já foi dito, o objetivo da regularização do decaimento dos pesos é encontrar um valor intermediário de wd, que minimize a taxa de erro no banco de teste. Pode-se notar que valores da regularização wd muito pequenos ainda possuem uma alta taxa de erro com relação ao banco de teste, isso acontece porque valores pequenos levam o modelo ao sobre-ajuste e grandes valores de wd implicam em sub-ajuste. De acordo com os resultados gerados e que são observados na tabela 3 acima, o valor ótimo de wd é $0,006$, onde a taxa de erro é reduzida para 15,87%. Pode-se notar que pequenos valores de wd possuem um erro de teste relativamente alto e, conforme o valor desse parâmetro aumenta, o modelo aumenta sua capacidade de generalização. Essa redução acontece até o valor ótimo de wd. Após esse ponto, o modelo começa a perder sua capacidade de treinamento e generalização, já que mais pesos são levados a zero e, consequentemente, acontece o sub-ajuste.

3.2.3 Rede Neural Convolucional

O modelo ajustado com a arquitetura de uma rede convolucional, com 2 camadas convolucionais e duas camadas totalmente conectadas teve um desempenho com a acurácia média em 49,23%. Com 300 iterações, levou 7 minutos de processamento. Além disso, essa estrutura obteve uma taxa de erro de 50%. Com apenas duas camadas convolucionais e a quantidade de kernels em cada uma das camadas não foi suficiente para a rede convolucional aprender as características de uma imagem de espiral à direita e à esquerda durante o treinamento, já que a partir da 16ª iteração a acurácia se mantém constante no valor de 49,52%. A Figura 13 representa a trajetória da acurácia dessa rede durante as 20 iterações do treinamento.

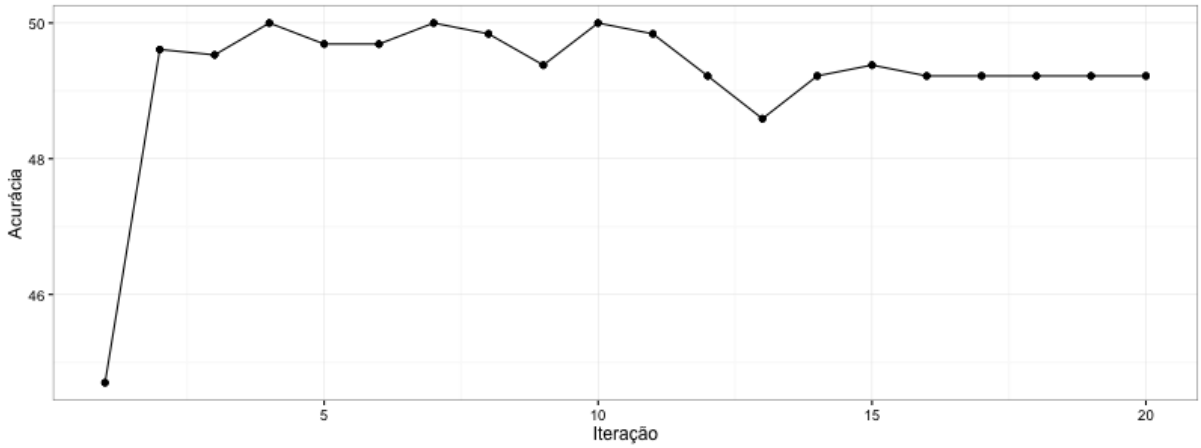


Figura 13: Desempenho da rede neural convolucional ajustadas

4 Considerações Finais

Após as análises das 3 estruturas de redes neurais ajustadas, ou seja, as redes MLP, *feedfoward* e CNN, a rede considerada mais eficiente foi a rede *feedfoward* com inicialização dos parâmetros através de uma distribuição U[-1,1]. Considerando como eficiente aquela rede que possui a maior capacidade de generalização, ou seja, que possui a maior capacidade classificar corretamente observações que não foram apresentadas para a rede durante o treinamento. Mesmo essa rede sendo considerada eficiente, nota-se que ela teve uma taxa de erro ainda alta, sendo que essa rede classificou de forma incorreta aproximadamente 28% das observações no banco de teste. Após o ajuste de um regularizador de decaimento dos pesos, onde $wd = 0,06$, a rede reduziu sua taxa de erro para 15,87%, aumentando sua performance em novas observações de forma significativa, ou seja, o parâmetro de regularização reduziu o sobre-ajuste do modelo.

Com relação as redes convolucionais, durante o seu treinamento, essa rede teve um desempenho maior do que as redes MPL, com acurácia média de 49,23%. Porém, para um computador pessoal, o seu processamento é árduo e demorado.

Um dos fatores relevantes para o desempenho eficiente de uma rede neural com camadas convolucionais se refere às imagens de entrada. Uma estrutura desse tipo espera identificar os padrões em diferentes posições, tamanhos e ângulos nas imagens de entrada. Pelo fato de que o banco de dados foi gerado de forma muito regular e estável, no sentido de que todas as imagens possuem um mesmo tamanho e estão todas centralizadas, a rede encontrou dificuldade para identificação das imagens. Além disso, outro fator importante para aumentar sua performance é que uma estrutura convolucional precisa ser muito mais complexa, ou seja, possuir mais camadas e mais neurônios para obter uma maior capacidade de identificação de características, e assim se tornar invariante e seletiva. A grande desvantagem de uma estrutura convolucional é que esta é computacionalmente complexa. Por isso, para o treinamento eficiente de uma rede desse tipo é necessário dispor de recursos computacionais maiores, como processadores do tipo GPU e realizar processamento de dados em paralelo.

5 Referências

LECUN, Y., BENGIO, Y., HINTON, G. (2015) Deep Learning. *Nature 521.*

BISHOP, C. (2001) *Pattern Recognition and Machine Learning.* Editora SPRINGER, 1ªed.

HAYKIN, S. (1999) *Redes Neurais: Princípios e Prática.* Artmed Editora S.A., 2ªed

BENGIO, Y., COURVILLE, A., GOODFELLOW, I. (2016) *Deep Learning.* MIT Press, 1ªed.

TIBISHIRANI, R., JAMES, G., WITTEN, D., and HASTIE, T. (2013) *An Introduction to Statistical Learning.* 1ªed.

A Anexos

```
############################
# 01_GERAR_BANCO_DE_DADOS #
############################

library("png")
library("colorspace")
setwd("/Users/Arthur/Desktop/banco_teste/")

## BANCO DE TREINAMENTO ##
#Direita
for(k in 1:10){
  for(b in seq(1,76, 5)){
    par(mar=c(0,0,0,0), xpd=TRUE)
    n=k*b
    theta=seq(0,k*2*pi, length=n)
    R=0.98^(1:n-1)
    x=R*cos(theta)
    y=R*sin(theta)
    png(paste("teste1_",k,"_",b,".png",sep=""))
    plot(x,y, type="l", axes=FALSE, xlab="", ylab="", lwd=3 )
    dev.off()
  }}


for(c in c(1,4,5,6)){
  for(k in 1:10){
    for(b in seq(1,76, 5)){
      par(mar=c(0,0,0,0), xpd=TRUE)
      n=k*b
      theta=seq(0,k*2*pi, length=n)
      R=0.98^(1:n-1)
      x=R*cos(theta)
      y=R*sin(theta)
      png(paste("teste1_",k,"_",b,"_",c,"_2.png",sep=""))
      plot(x,y, type="l", axes=FALSE, xlab="", ylab="", lwd=3, lty=c)
      dev.off()
    }}}


# Esquerda

for(k in 1:10){
  for(b in seq(1,76, 5)){
    par(mar=c(0,0,0,0), xpd=TRUE)
    n=k*b
```

```
    theta=seq(0,k*2*pi, length=n)
    R=0.98^(1:n-1)
    x=R*cos(-theta)
    y=R*sin(-theta)
    png(paste("teste3_",k,"_",b,".png",sep=""))
    plot(x,y, type="l", axes=FALSE, xlab="", ylab="", lwd=3)
    dev.off()
  }}

for(c in c(1,4,5,6)){
  for(k in 1:10){
    for(b in seq(1,76, 5)){
      par(mar=c(0,0,0,0), xpd=TRUE)
      n=k*b
      theta=seq(0,k*2*pi, length=n)
      R=0.98^(1:n-1)
      x=R*cos(-theta)
      y=R*sin(-theta)
      png(paste("teste3_",k,"_",b,"_",c,".png",sep=""))
      plot(x,y, type="l", axes=FALSE, xlab="", ylab="", lwd=3 , lty=c)
      dev.off()
    }}}


## BANCO DE TESTE ##
#Esquerda
for(b in seq(10,76, 5)){
  for(k in 1:10){
    par(mar=c(0,0,0,0), xpd=TRUE)
    theta = seq(0, k* 2 * pi, by = 2 * pi/b)
    x = cos(theta)
    y = sin(theta)
    R =theta^(1/2)
    png(paste("teste4_",a,"_",b,"_",k,".png",sep=""))
    plot(x * R, y * R, type="l", axes=FALSE, xlab="", ylab="", lwd=3)
    dev.off()
  }}

#Direita
for(b in seq(10,76, 5)){
  for(k in 1:10){
    par(mar=c(0,0,0,0), xpd=TRUE)
    theta = seq(0, k* 2 * pi, by = 2 * pi/b)
    x = cos(-theta)
    y = sin(-theta)
    R =theta^(1/2)
    png(paste("teste5_",a,"_",b,"_",k,".png",sep=""))
    plot(x * R, y * R, type="l", axes=FALSE, xlab="", ylab="", lwd=3)
```

```
    dev.off()
  }}

###################################
#02_REDUZIR_QUALIDADE_DAS_IMAGENS #
###################################

#RODAR PELO TERMINAL (LINUX OU OSX)

#Instalar o pacote ImageMagick

\#Abrir o compilador de convert. Comando do ImageMagick
type convert
\#Definir diretorio das pastas
\#Para converter todas as imagens em um formato 50x50
convert -resize 50x50 -type grayscale *.png \%02d.png


#######################################
# 03_LER_IMAGENS_CRIAR_BANCO_DE_DADOS #
#######################################

library("png")
library("colorspace")

#Importacao das imagens e transformacao em matriz de pixels
#Direita: 1
setwd("/Users/Arthur/Desktop/banco_teste/Direita/Convertidos")

imagens<-list.files(getwd())
treinamento<-NULL

for(i in 1:length(imagens)){
  x<-readPNG(imagens[i])
  if(length(dim(x))==3){
    y <- rgb(x[,,1], x[,,2], x[,,3])
    yn<- col2rgb(y)[1, ]/255
    #dim(yn) <- dim(x)[1:2]
    treinamento<-rbind(treinamento, yn)
  }
  else{
    y<-as.vector(x)
    treinamento<-rbind(treinamento, y)
  }
  setTxtProgressBar(txtProgressBar(min=0, max=length(imagens), style = 3), i)
}
#Rotulando as imagens direita como 1
banco.direita<-cbind(label=rep(1, nrow(treinamento)), treinamento)
```

```
#Esquerda:2
setwd("/Users/Arthur/Desktop/banco_teste/Esquerda/Convertidos")

imagens<-list.files(getwd())
treinamento<-NULL

for(i in 1:length(imagens)){
  x<-readPNG(imagens[i])
  if(length(dim(x))==3){
    y <- rgb(x[,,1], x[,,2], x[,,3])
    yn<- col2rgb(y)[1, ]/255
    #dim(yn) <- dim(x)[1:2]
    treinamento<-rbind(treinamento, yn)
  }
  else{
    y<-as.vector(x)
    treinamento<-rbind(treinamento, y)
  }
  setTxtProgressBar(txtProgressBar(min=0, max=length(imagens), style = 3), i)
}

#Rotulando as imagens esquerda como 2
banco.esquerda<-cbind(label=rep(2, nrow(treinamento)), treinamento)

final.treinamento<-rbind(banco.direita, banco.esquerda)

#Trocando a escala de 1 para 0
final.treinamento[,2:ncol(final.treinamento)]<-
abs(final.treinamento[,2:ncol(final.treinamento)]-1)

write.csv2(final.treinamento, file=
"/Users/Arthur/Desktop/banco_teste/banco.final.treinamento2.csv", row.names = FALSE


#Banco de teste
#Direita: 1
setwd("/Users/Arthur/Desktop/banco_teste/Teste_Direita/Convertidos")

imagens<-list.files(getwd())
teste<-NULL

for(i in 1:length(imagens)){
  x<-readPNG(imagens[i])
  if(length(dim(x))==3){
    y <- rgb(x[,,1], x[,,2], x[,,3])
    yn<- col2rgb(y)[1, ]/255
    #dim(yn) <- dim(x)[1:2]
```

```
    teste<-rbind(teste, yn)
  }
  else{
    y<-as.vector(x)
    teste<-rbind(teste, y)
  }
  setTxtProgressBar(txtProgressBar(min=0, max=length(imagens), style = 3), i)
}
#Rotulando as imagens direita como 1
banco.direita<-cbind(label=rep(1, nrow(teste)), teste)

#Esquerda: 2
setwd("/Users/Arthur/Desktop/banco_teste/Teste_Esquerda/Convertidos")

imagens<-list.files(getwd())
teste<-NULL

for(i in 1:length(imagens)){
  x<-readPNG(imagens[i])
  if(length(dim(x))==3){
    y <- rgb(x[,,1], x[,,2], x[,,3])
    yn<- col2rgb(y)[1, ]/255
    #dim(yn) <- dim(x)[1:2]
    teste<-rbind(teste, yn)
  }
  else{
    y<-as.vector(x)
    teste<-rbind(teste, y)
  }
  setTxtProgressBar(txtProgressBar(min=0, max=length(imagens), style = 3), i)
}
#Rotulando as imagens direita como 1
banco.esquerda<-cbind(label=rep(2, nrow(teste)), teste)


final.teste<-rbind(banco.direita, banco.esquerda)

#Trocando a escala de 1 para 0
final.teste[,2:ncol(final.teste)]<-abs(final.teste[,2:ncol(final.teste)]-1)

write.csv2(final.teste, file=
"/Users/Arthur/Desktop/banco_teste/banco.final.teste2.csv", row.names = FALSE)
```