UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA E INFORMÁTICA

Estudo sobre arquiteturas, regularização e otimização de sistemas de aprendizado profundo

Vítor Leão Caminha

Campina Grande, PB

Março de 2020

Vítor Leão Caminha

Estudo sobre arquiteturas, regularização e otimização de sistemas de aprendizado profundo

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Elétrica.

Área de Concentração: Controle e Automação

Orientador: **Professor George Acioli Júnior, D. Sc.**

Campina Grande, PB

Março de 2020

Vítor Leão Caminha

Estudo sobre arquiteturas, regularização e otimização de sistemas de aprendizado profundo

Aprovado em ___/___ /___

Professor Danilo Freire de Souza Santos, D. Sc.

Universidade Federal de Campina Grande

Avaliador

Professor George Acioli Júnior, D. Sc.

Universidade Federal de Campina Grande

Orientador

Dedico este trabalho a toda a minha família pelo apoio recebido,

onde forças foram somadas para continuar essa jornada.

Agradecimentos

Inicialmente, a Deus, por estar sempre presente em minha vida e me guiar pelos melhores caminhos possíveis.

Aos meus pais, Valério e Jane, que sempre me apoiaram eu tudo que puderam, e por me oferecerem a educação e os seus exemplos como principal herança que eles poderiam me proporcionar.

A Bruno, meu irmão, pelo exemplo constante do que é lutar com todas as forças pelos seus sonhos.

À Cynthia, minha noiva, pela dedicação, apoio e parceria sempre disponíveis para mim.

A meus amigos, por estarem sempre presentes nos momentos difíceis com descontração e leveza necessária.

A todos os professores, que me ensinaram todo o essencial necessário para eu me tornar um excelente profissional. Em especial, agradeço ao meu orientador George Acioli, pela disponibilidade e suporte na orientação deste trabalho.

Por fim, gratidão a todos que passaram pela minha vida e que contribuíram, direta ou indiretamente, para a construção deste momento que para mim é tão importante.

Resumo

Devido ao recente desenvolvimento da área do aprendizado profundo no campo da inteligência artificial, muitos conceitos teóricos e práticos ainda se encontram obscuros e com pouca documentação para pesquisadores iniciantes na área. Nesse contexto, esse trabalho de conclusão de curso visa explanar alguns conceitos básicos e suas influências na performance e resultados de sistemas de aprendizado profundo e redes neurais. Além da teoria, implementaram-se experimentos didáticos para exemplificar construções práticas de sistemas com diferentes arquiteturas, para diferentes problemas, comparando-se funções de ativação, algoritmos de otimização e regularização, valores iniciais de parâmetros, entre outros, otimizando-se as redes com escolhas de projeto melhores para os problemas específicos.

Palavras chave: Aprendizado profundo, Redes neurais, Inteligência artificial, Função de ativação, Otimização, Regularização

Abstract

Due to the recent development of the area of deep learning in the field of artificial intelligence, many theoretical and practical concepts are still obscure and with little documentation for beginning researchers in the area. In this context, this undergraduate thesis aims to explain some basic concepts and their influences on the performance and results of deep learning systems and neural networks. In addition to the theory, didactic experiments were implemented to exemplify practical constructions of systems with different architectures, for different problems, comparing the activation functions, optimization and regularization algorithms, initial parameter values, among others, optimizing the networks with better design choices for the specific problems.

Keywords: Deep learning, Neural network, Artificial intelligence, Activation function, Optimization, Regularization

Lista de Figuras

Lista de Abreviaturas e Siglas

η	Taxa de aprendizado
θ	Limiar de ativação
$J(W)$	Função de custo
W	Matriz de parâmetros
w_i	Peso da entrada i
x_i	Entrada i
IA	Inteligência Artificial

Conteúdo

1 Introdução

Quando os primeiros computadores programáveis foram concebidos, a ideia de maquinas inteligentes também começou a ser imaginada, até que o primeiro algorítimo inteligente foi inventado em 1842, por Ada Lovelace. Hoje, a Inteligencia Artificial (IA) é um campo em amplo desenvolvimento, com diversas aplicações práticas, como automatização de rotinas de trabalho, interpretação de textos, voz e imagens, diagnósticos médicos, suporte à pesquisas científicas, tomadas de decisões por robôs, entre outros.

Inicialmente, a IA se preocupava em solucionar problemas que eram difíceis de resolver manualmente, mas que eram simples de descrever em uma lista de regras formais. Porém, o verdadeiro desafio da IA se provou ser solucionar tarefas que são fáceis de pessoas realizarem, mas difíceis de serem descritas formalmente, problemas que nós resolvemos intuitivamente e automaticamente, como reconhecer voz, letras ou objetos, que são tarefas essenciais para os mais variados dispositivos e sistemas atuais.

A solução para esses problemas mais intuitivos é permitir que as máquinas aprendam pela experiência e que compreendam o ambiente por meio de conceitos hierárquicos, onde cada conceito é definido por uma relação com um conceito mais simples. Isso permite que o aprendizado de conceitos complexos seja construído a partir dos mais simples, formando camadas de conceitos que, se empilhadas, formariam blocos bastante profundos. Por essa razão, essa abordagem da IA é chamada de *deep learning*. A figura 1.2 mostra, de forma simplificada, como sistemas *deep learning* podem combinar conceitos simples para representar conceitos complexos de uma imagem, e, assim, poder classificá-la.

1.1 Aprendizado de máquina e aprendizado profundo

As dificuldades encontradas por sistemas que dependem de conhecimento codificado manualmente, citadas anteriormente, sugerem que sistemas de IA precisam da habilidade de adquirir seu próprio conhecimento extraindo padrões de dados brutos. Essa capacidade é conhecida como

aprendizado de máquina, e a sua introdução permite que computadores abordem problemas do mundo real e tomem decisões que aparentam ser subjetivas.

Porém, o desempenho de simples algoritmos de aprendizado de máquinas são altamente dependentes da representação dos dados aos quais são alimentados. Algumas tarefas podem ser resolvidas pela extração de um conjunto de características bem definidas, mas para outras tarefas, a dificuldade de saber quais características devem ser extraídas se torna muito alta. A figura 1.1 mostra a importância das representações de características na performance do algoritmo. Suponha que queira-se dividir duas categorias de dados de um problema traçando uma linha no seu gráfico. Para a representação da esquerda, que é representada em coordenadas cartesianas, seria uma tarefa impossível. Já para a da direita, representada em coordenadas polares, seria uma tarefa bastante simples.

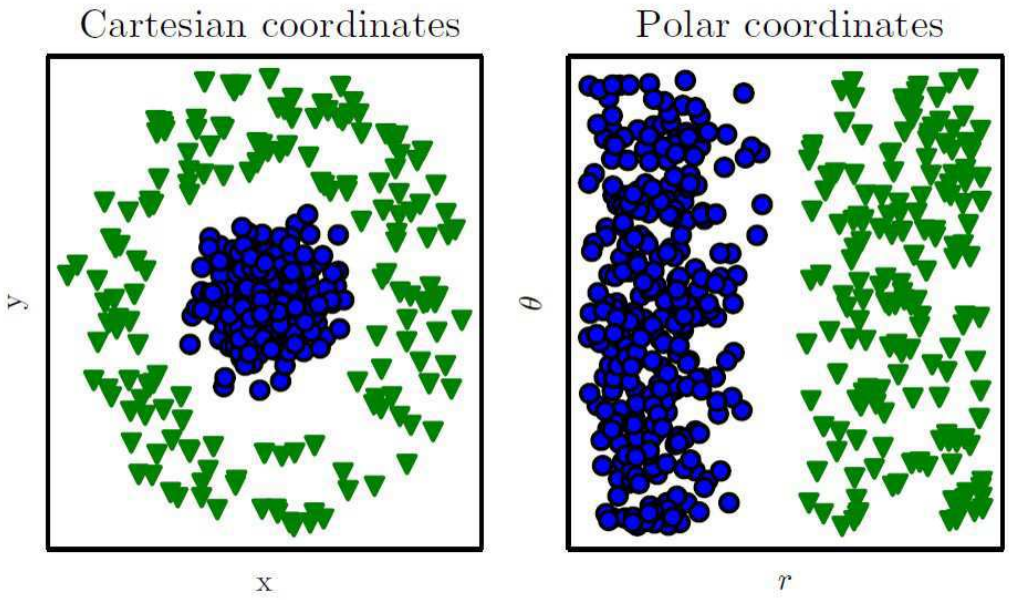

Figura 1.1: Exemplo de duas diferentes representações para um mesmo problema de classificação

Uma solução para esse problema é usar o aprendizado de máquinas para descobrir não somente a relação da representação com a saída, mas também a representação em si. Essa abordagem é conhecida como aprendizado de representação. Representações aprendidas geralmente resultam em uma performance muito melhor do que representações definidas manualmente. Elas também permitem a sistemas de IA se adaptarem a novas tarefas com uma intervenção humana

mínima.

Um algoritmo de aprendizado de representação pode descobrir um bom conjunto de características para uma simples tarefa em minutos, mas para uma tarefa complexa, pode durar de horas a meses, e defini-las manualmente requer uma grande quantidade de tempo e esforço humano, podendo levar décadas com uma comunidade inteira de pesquisadores dedicados.

O aprendizado profundo, ou *deep learning*, resolve esse problema central do aprendizado de representação ao introduzir representações que são expressas em termos de outras mais simples. A figura 1.2 mostra como esse sistema pode representar uma imagem a partir da combinação de conceitos simples, e a figura 1.3 mostra um fluxograma com os diferentes tipos de sistemas de IA e como elas se comportam em cada etapa do seu processo de aprendizado.

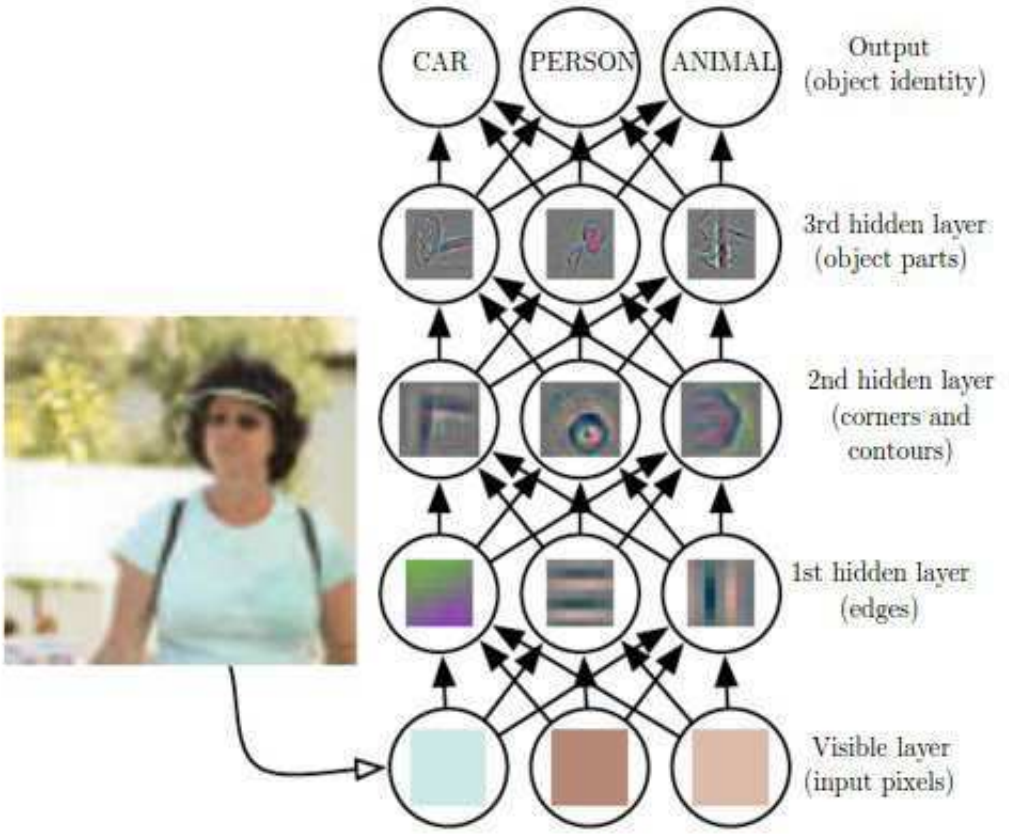

Figura 1.2: Exemplo de um simples sistema *deep learning* representando conceitos complexos a partir de conceitos mais simples

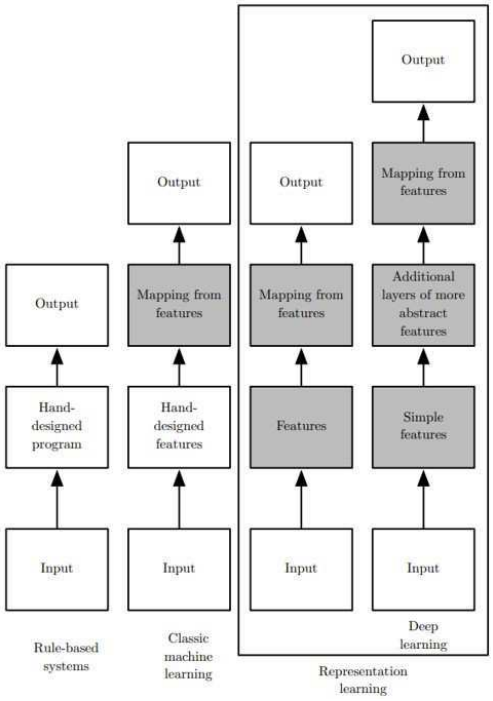

Figura 1.3: Fluxograma mostrando os diferentes tipos de sistemas de IA e seus comportamentos em cada etapa do seu processo de aprendizado

Mais formalmente, *deep learning* é uma classe de técnicas de aprendizado de máquinas que explora muitas camadas de processamento de informações não-lineares, para extração e transformação de características, e para análises de padrões e classificações, de forma supervisionada ou não. Os tipos de sistemas são:

- Supervisionado: Utilizam de dados rotulados. São os mais usados. Incluem as Redes Neurais feedfoward, as convolucionais, recorrentes, entre outros.
- Semi-supervisionados: Utilizam base de dados parcialmente rotulados. Incluem principalmente os sistemas de aprendizado profundo por reforço.
- Não-supervisionados: Não utilizam nenhum tipo de rotulação. Por serem os sistemas mais recentes, ainda são pouco utilizados, porém é a forma que se tem maior expectativa, devido a menor interferência humana e tamanho do conjunto de dados. Incluem principalmente os Autoencoders e alguns casos de aprendizado profundo por reforço.

1.2 Objetivos

Com o aumento cada vez maior das aplicações de sistemas de aprendizado profundo nos mais diversos contextos, existe uma crescente procura por pesquisas e estudos na área, porém ainda com poucas documentações e explicações na linguagem. Com a intenção de ajudar e incentivar novos cientistas e engenheiros a adentrar na área de sistemas de aprendizado profundo, o presente trabalho possui o objetivo de apresentar e explicar, de forma simplificada e didática, conceitos básicos e suas influências no desempenho dos sistemas, principais arquiteturas, parâmetros e métodos de otimização e regularização dos modelos, para que o novo pesquisador entenda da melhor forma as ferramentas disponíveis e o porquê de utilizá-las no seu sistema de aprendizado profundo para resolver determinado problema específico.

Além da teoria mostrada, são exemplificadas de forma prática, para as principais arquiteturas atuais, implementações de modelos e o processo de refatoração de um sistema, com melhores escolhas de parâmetros, otimização e regularização para o determinado problema, para obter níveis desejados de desempenho. Dessa forma, tanto os modelos quanto o processo, podem ser usados como ponto de partida pelo leitor para a construção e adaptação de um sistema que atenda suas necessidades de resolução de um problema de forma satisfatória.

1.3 Organização do trabalho

O presente trabalho foi organizado e estruturado da seguinte forma:

- No capítulo 1, foi dada uma breve introdução e motivação para a realização de sistemas de aprendizado profundo e deste trabalho;
- No capítulo 2, foi apresentada a arquitetura mais simples de uma rede neural, conhecida como perceptron, para entender os primeiros conceitos básicos de um sistema de aprendizado profundo;
- No capítulo 3, apresentou-se a arquitetura mais utilizada atualmente, a FeedForward, com

suas principais escolhas de parâmetros e métodos possíveis. Além disso, é explicado de que forma a arquitetura consegue aprender a partir dos dados a qual ela foi alimentada;

- Nos capítulos 4, 5 e 6, foram apresentados os principais métodos de otimização, regularização e escolhas de topologias. Explicou-se como cada um deles funcionam, suas vantagens e desvantagens, e o porque deve-se escolher determinado método para cada caso;

- No capítulo 7, apresentou-se uma metodologia prática de criação de uma rede neural com arquitetura FeedForward para resolver um problema simples, exemplificando-se o processo de implementação, otimização, regularização e refatoração de uma rede, fazendo com que o exemplo possa ser usado como base para um passo-a-passo da tomada de decisões e análises de parâmetros de performance em um processo de criação de um sistema de aprendizado profundo prático em um contexto real;

- Nos capítulos 8 e 9, foram apresentadas mais duas arquiteturas muito usadas atualmente, a convolucional e a recorrente, apresentando os seus conceitos e funcionalidades específicas, e os casos e aplicações onde cada arquitetura se sobressai. Ambas também apresentaram exemplificações de implementações práticas, para que, além de servir como base para a construção de modelos com essas arquiteturas, pudessem ser comparadas as vantagens e desvantagens de cada arquitetura, analisando as facilidades de implementação e treinamento, as eficiências e as performances atingidas;

- No capítulo 10, foi feita uma conclusão do trabalho e são apresentadas outras arquiteturas que surgiram recentemente e que, apesar de ainda estarem sendo pouco utilizadas, prometem ter grandes e bem sucedidos futuros;

- No capítulo 11, são mostradas as referências bibliográficas usadas durante a realização deste trabalho;

- E, por fim, foram anexados quatro códigos utilizados para as implementações e exemplificações práticas citadas neste trabalho.

2 Perceptron

Perceptron é tido como a arquitetura mais simples de uma rede neural artificial, que é composto por somente uma camada de entrada, um neurônio, ou unidade oculta (*hidden unit*), formando apenas uma camada oculta, ou *hidden layer*, e uma única saída, como representado na figura 2.1. Essa arquitetura se comporta basicamente como um classificador binário, ou seja, o resultado da saída será dado por uma função que compara o somatório dos valores de entrada, multiplicados por seus respectivos pesos, com um valor limiar, chamado de limiar de ativação. Se for maior, a função retornará o valor 1 para a saída, ou o valor 0 caso contrário. De forma algébrica:

$$f(x) = \begin{cases} 0, & se \quad \sum_{i=1}^{n} w_i.x_i < \theta \\ 1, & se \quad \sum_{i=1}^{n} w_i.x_i \geq \theta \end{cases} \tag{1}$$

Onde,

w_i é o peso da entrada i;

x_i é a entrada i;

θ é o limiar de ativação.

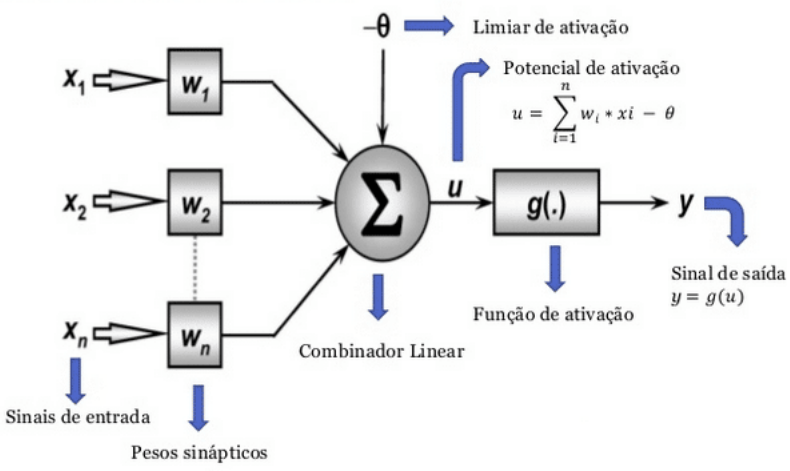

Figura 2.1: Representação da arquitetura perceptron

Ainda, ao passar θ para a primeira parte da equação 1, podemos fazer com que o limiar de ativação se torne um valor de entrada do neurônio, se tornando uma característica visível, ajustável

e própria de cada neurônio. Logo, temos a função

$$f(x) = \begin{cases} 0, & se \quad \sum_{i=1}^{n} w_i.x_i - \theta < 0 \\ 1, & se \quad \sum_{i=1}^{n} w_i.x_i - \theta \geq 0 \end{cases} \tag{2}$$

Portanto, é possível observar que essa função do perceptron, chamada de função de ativação, nada mais é do que uma função degrau. Dada essa simplicidade, o perceptron pode ser usado para classificações lineares, porém, é notável a sua incapacidade de solucionar problemas não-lineares, fazendo com que seu uso seja muito restrito.

3 Redes neurais feedfoward

Redes neurais feedfoward, ou *multi layers perceptrons* (MLPs), são modelos de redes neurais que, a exemplo do perceptron, não possuem nenhum tipo de *feedback* para correção de erros do sistema. Porém, ao ser composto por diversas camadas ocultas, cada uma com sua função de ativação, é capaz de solucionar um número muito mais abrangente de casos, incluindo os não-lineares. Por exemplo, para uma arquitetura com três camadas ocultas, sua função de ativação é dada por $f(x) = f^{(3)}(f^2(f^{(1)}(x)))$, onde $f^{(1)}$ é chamada de **primeira camada** da rede, $f^{(2)}$ é chamada de **segunda camada** da rede, e assim por diante. Se estender o modelo linear ao problema não-linear, então $f^{(3)}$, que é a função de ativação da saída, deverá aplicar o modelo linear a seu vetor de entrada, que não corresponde mais à camada de entrada **x**, mas sim a uma entrada **h(x)** que é a representação não-linear de **x**, gerada pelas camadas ocultas anteriores da rede. Devemos, obviamente, usar funções não-lineares para descrever essas representações. Na maior parte das redes neurais, a função de ativação mais recomendada é a *Rectified Linear Unit*, ou ReLU, definida como $g(z) = max(0, z)$, muito usada pela simplicidade, pois, apesar de ser uma função não linear, se assemelha bastante a uma função linear em metade do seu domínio, e tem valor 0 na outra metade, diminuindo substancialmente o esforço computacional necessário ao modelo e abrangendo grande parte das aplicações. Porém, existem outras funções de ativação muito usadas em casos específicos, como a Sigmoid, definida por $g(z) = \frac{1}{1+e^{-z}}$, usada em aplicações probabilísticas por ter sua saída entre 0 e 1; Tanh, definida por $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, usada em casos que necessitam de sua saída entre -1

e 1; ELU, derivada da ReLU, mantém um comportamento linear em metade do domínio e constante na outra metade, permanecendo uma função simples que não exige muito esforço computacional. Porém, permite a configuração da angulação da curva e a utilização de valores negativos com a definição de uma assíntota diferente de zero, abrangendo mais aplicações; Softmax, muito usada como função de ativação da camada de saída para problemas de classificação em múltiplas classes. Ela calcula a distribuição de probabilidades de um evento sobre todos os outros eventos. Por exemplo, ela vai calcular as probabilidades de uma determinada entrada pertencer a um tipo dentre vários tipos possíveis; entre outras. Os comportamentos das três primeiras funções são demonstrados na figura 3.1. Contudo, como os parâmetros da representação não-linear são definidos pelo próprio aprendizado do sistema, é necessário o uso de algorítimos de otimização, e há um custo que deve ser considerado.

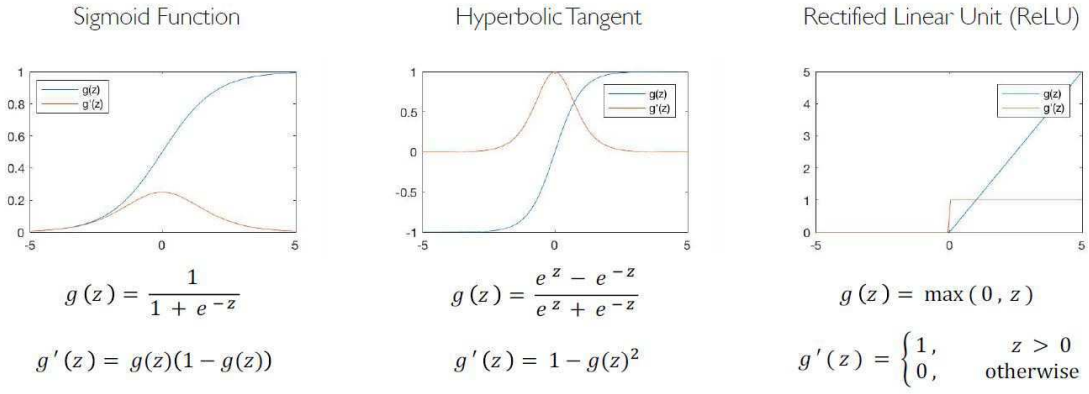

Figura 3.1: Comportamento das principais funções de ativação e suas derivadas

Ao desenhar uma arquitetura do modelo feedfoward para resolver um determinado problema, deve-se considerar alguns outros quesitos além das funções de ativação, como quantas camadas a rede deve conter, como essas camadas devem ser conectadas entre si, e quantas unidades cada camada deve conter; como deve ser medido os erros e qual o custo tolerável; como dever ser feita a otimização e a regularização para que não ocorra *overfitting*, nem *underfitting*, entre outros que serão abordados.

3.1 Medição de performance

Para medir as capacidades de um algoritmo de aprendizado de máquina, deve-se projetar uma medição quantitativa da sua performance. Normalmente, essa medição de performance é específica a uma determinada tarefa sendo realizada pelo sistema. Para tarefas como classificação ou transcrição, geralmente se mede a precisão do modelo.

Precisão é a proporção de exemplos para os quais o sistema produz saídas corretas. Pode-se também obter uma informação equivalente ao medir uma taxa de erro, que é a proporção de exemplos para os quais o sistema produz saídas incorretas. Podemos nos referir a taxa de erro também como o custo 0-1 esperado. O custo 0-1 em um exemplo particular é 0 se for corretamente classificado ou 1 se não for.

Para tarefas como estimativa de densidade de probabilidade, não faz sentido medir a precisão, a taxa de erro ou qualquer outro tipo de custo 0-1. Como alternativa, deve-se utilizar uma métrica de performance diferente que possibilita ao modelo fornecer um valor contínuo para cada exemplo. A abordagem mais comum é expor o valor médio do log da probabilidade que o modelo determina para alguns exemplos.

Usualmente, o interesse está em conhecer a performance do algorítimo de aprendizado de máquina em um dado não conhecido anteriormente, já que isso determina quão o sistema irá funcionar em aplicações práticas reais. Para isso, utiliza-se um conjunto de dados de teste, que é separado dos dados usados para treinar o algorítimo.

Apesar de parecer uma medição direta e objetiva, muitas vezes se encontra dificuldades em escolher uma medição de performance que corresponda bem ao comportamento desejado do sistema. Em alguns casos, isso se dá pela dificuldade de se decidir o que deve ser medido. Por exemplo, quando realizada uma tarefa de transcrição de texto, deve-se medir a precisão do sistema em transcrever sequências inteiras, ou deve-se usar uma medição de performance mais refinada que reconhece o mérito parcial de conseguir corretamente alguns elementos da sequência? Quando realizada uma tarefa de regressão, deve-se penalizar mais o sistema se ele comete pequenos erros frequentemente ou se ele comete grandes erros raramente? Esses tipos de decisões de projeto

dependem da aplicação e da tarefa que o sistema irá realizar.

Em outros casos, sabe-se exatamente o que deve ser medido, mas a medição é impraticável. Isso acontece frequentemente no contexto de estimativa de densidade de probabilidade. Muitos dos melhores modelos probabilísticos representam a densidade de probabilidade somente de forma implícita. Computar o real valor da probabilidade atribuído a um ponto no espaço nesses modelos é intratável. Nesses casos, deve-se encontrar outros critérios que se aproximem do desejado, ou que tenham o mesmo objetivo. Essas escolhas são o que tornam a medição da performance de vários sistemas algo tão complexo e subjetivo e, por isso, deve-se sempre atentar para quais critérios estão sendo medidos, e se eles de fato atendem ao objetivo desejado.

3.2 Funções de custos

Como dito, um importante aspecto da criação de uma rede neural profunda é a escolha da função de custo, que nada mais é do que a quantificação do custo total médio incorrido dos erros de predição por todos os valores do conjunto de dados de entrada da rede. Ou seja, se o erro gerado por uma entrada é dada por $\mathcal{L}(y^*, y)$, então a função de custo será

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i^*, y_i) \tag{3}$$

Onde,

W é a matriz de parâmetros da rede

y_i^* é a saída predita para uma entrada x_i e a matriz de pesos W;

y_i é a saída correta para uma entrada x_i;

n é o número de entradas no conjunto de dados.

Existem diversas equações para quantificar esse erro $\mathcal{L}(y^*, y)$. Entre os principais, estão o erro médio quadrático e a entropia cruzada binária. No erro médio quadrático, usado em medições

de características que resultem em números reais contínuos, a função de custo fica na forma:

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^*)^2 \tag{4}$$

Muito usada para modelos probabilísticos ou que tenham saída entre 0 e 1, na entropia cruzada binária teremos a seguinte função de custo:

$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} y_i log(y_i^*) + (1 - y_i) log(1 - y_i^*) \tag{5}$$

A maior diferença entre os modelos lineares e os não-lineares das redes neurais é que a última faz com que a maioria das melhores funções de custo se tornem não-convexas. Isso significa que as redes neurais são geralmente treinadas utilizando otimizadores iterativos baseado em gradientes que levam a função de custo para um valor muito baixo, e não solucionando equações lineares, pois, diferente dos modelos lineares usados para treinar sistemas de regressão lineares ou SVMs, suas convergências muitas vezes não são alcançadas.

Otimizações convexas iniciando a partir de qualquer parâmetro inicial irão convergir em quase todos os casos. Algoritmos baseados em gradiente descendentes aplicados a funções de custo não convexos não possuem essa convergência garantida e é sensível aos valores dos parâmetros iniciais. Para redes neurais feedfoward, é importante inicializar todos os pesos com pequenos valores randômicos. Os limiares de ativação devem ser inicializados com valor zero ou valores positivos pequenos. Algoritmos de otimização iterativos baseados em gradientes são quase sempre os usados para treinar redes feedfoward e quase todos os outros modelos profundos, e veremos mais sobre eles a seguir.

3.3 Back-propagation

Quando utiliza-se uma rede neural feedfoward para aceitar uma entrada x e produzir uma saída y*, a informação flui através da rede. A entrada x provê a informação inicial que é propagada pelas unidades ocultas de cada camada até produzir a saída y*. Isso é chamado de forward propagation, ou propagação direta.

Durante o treinamento, a propagação direta é realizada até produzir um custo escalar $J(W)$. Então, o algoritmo back-propagation, ou backprop, permite que a informação da função de custo flua para trás através da rede, calculando o gradiente da função de custo em relação a cada parâmetro da rede, de forma a atualizar seus valores e otimizar a função de custo.

Calcular uma expressão analítica para o gradiente é algo bem direto, mas calcular a expressão numericamente pode ter um alto custo computacional. O algoritmo back-propagation realiza essa tarefa utilizando um procedimento bastante simples e com quase nenhum custo.

O termo back-propagation é entendido erroneamente como abrangendo todo o algoritmo de aprendizado para redes neurais. Na verdade, back-propagation se refere somente ao método de calcular o gradiente, enquanto outros algoritmos, como o gradiente descendente, é utilizado para o aprendizado usando o gradiente encontrado.

Muitas tarefas de aprendizado de máquina envolvem calcular outras derivadas, seja como parte do processo de aprendizado ou para analisar o modelo aprendido. O algoritmo back-propagation pode ser aplicado a essas tarefas também e não é restrito ao cálculo do gradiente da função de custo com relação aos seus parâmetros.

3.4 Algoritmo de otimização

Deve-se calcular o gradiente $\nabla_w J(W)$, necessário para os algoritmos de aprendizado, da seguinte forma: inicialmente, se escolhe valores iniciais para os parâmetros, geralmente randomicamente com distribuição normal centrado em 0 e com um desvio padrão σ^2, e se calcula o gradiente da função de custo para esse ponto. Então, os valores dos parâmetros são atualizados em um passo no sentido contrário ao gradiente, definido como

$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W} \tag{6}$$

Onde,

W é a matriz de parâmetros da rede;

η é a taxa de aprendizado do algoritmo;

$J(W)$ é a função de custo calculado anteriormente.

Com os novos valores de W, encontra-se o novo valor da função de custo, e repete-se o cálculo do gradiente para o novo valor. Depois, reatualiza-se o valor da matriz W e repete-se os mesmos passos até que o gradiente convirja para um valor mínimo ótimo.

Mas como devem ser calculados os gradientes para cada parâmetro? No exemplo a seguir, é mostrado como o algoritmo back-propagation utiliza a regra da cadeia para identificar a influência e calcular o valor do gradiente de cada parâmetro.

Suponha a simples rede neural mostrada na figura 3.2 e que se queira descobrir como a atualização de um parâmetro específico w2 irá influenciar no valor final da função de custo.

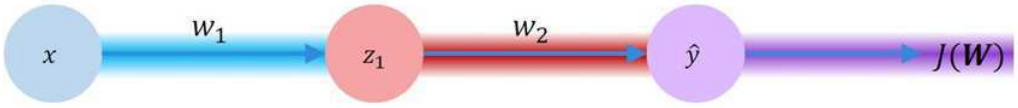

Figura 3.2: Simples rede neural com os seus parâmetros

Para isso, calcula-se o gradiente da função de custo para a variável w2, que, pela regra da cadeia, fica na seguinte forma:

$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2} \tag{7}$$

Como todos os valores são conhecidos, o cálculo se torna algo simples. E, se quisermos saber o gradiente para qualquer outro parâmetro do sistema, utiliza-se a regra da cadeia da mesma forma, fazendo com que a informação da saída de propague por todos os parâmetros até a entrada do sistema, mantendo a computação simples, como mostrado na equação 8.

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1} \tag{8}$$

Agora, suponha uma rede neural um pouco mais complexa, com 12 parâmetros, e para

cada parâmetro, são necessárias 1000 iterações para de obter o gradiente mínimo. Então, para cada dado de entrada, teria 12 mil cálculos. Supondo que existam 10 mil dados de entrada, seriam 120 milhões de cálculos para o conjunto de dados inteiro. Se aumentarmos ainda mais a complexidade da rede neural, fica claro que o custo computacional vai deixando o sistema cada vez mais impraticável, fazendo com que exista a grande necessidade de otimizar também os algoritmos de aprendizado do sistema, e que deve-se escolher com muita atenção como essa otimização deverá ser feita.

4 Regularização

Regularização é definido como "qualquer modificação feita ao algorítimo de aprendizado com a intenção de minimizar seu erro de generalização, mas não seu erro de treinamento"[1]. Existem várias estratégias de regularização que, em sua maioria, se propõem a limitar e penalizar os parâmetros do modelo, com o objetivo de melhorar sua performance durante o conjunto de teste, e são baseados em estimadores regularizantes.

Regularização de estimadores funcionam pela troca de maiores limiares de ativação por uma variância reduzida. Um regularizador efetivo é um que faça uma troca vantajosa, reduzindo a variância significantemente enquanto o limiar não é aumentado demais. Na prática, Algoritmos de aprendizado profundo são tipicamente aplicados a domínios extremamente complexos, como imagens, sequência de áudio e texto, entre outros, que no processo de geração de dados, envolve simular todo um universo. Isso significa que controlar a complexidade do modelo não é uma simples questão de encontrar o modelo do tamanho certo, com o número certo de parâmetros, mas encontrar um modelo com o menor erro de generalização possível, que seja robusto e regularizado apropriadamente.

Serão mostradas três das principais estratégias para criar esses modelos regularizados, que sejam profundos e robustos.

4.1 Aumento do conjunto de dados

A melhor forma de fazer um modelo generalizar melhor é treiná-lo com mais dados. Na prática, a quantidade de dados que se tem é limitada e uma forma de contornar esse problema é criando dados falsos e os adicionando ao conjunto de treinamento. Para algumas tarefas, criar novos dados falsos é algo razoavelmente direto.

Essa abordagem é mais fácil para tarefas de classificação. Um classificador precisa pegar uma complicada entrada x, de alta dimensão, e resumi-la em uma única categoria y. Isso significa que a tarefa principal enfrentada por um classificador é ser invariante para uma vasta variedade de transformações. Pode-se gerar facilmente um novo par (x, y) somente ao transformar uma entrada x em nosso conjunto de treinamento. Essa abordagem não é tão aplicável para outras tarefas. Por exemplo, é difícil gerar um novo dado falso para uma tarefa de estimativa de densidade de probabilidade, a menos que o problema da estimativa já esteja resolvido.

Operações como translacionar as imagens de treinamento alguns pixels em cada direção podem melhorar fortemente a generalização. Muitas outras operações como rotacionar, escalar ou adicionar ruídos a imagens ou áudios também se mostram muito efetivas, não somente à generalização como à robustez do modelo.

Deve-se ter cuidado para não aplicar essas transformações a entradas de forma a mudar suas classes, de fato. Por exemplo, alguns sistemas devem reconhecer a diferença entre “b” e “d” ou “6” e “9”. Então, espelhar ou rotacionar essas entradas não seriam formas apropriadas de aumentar o conjunto de dados para essa tarefa.

4.2 *Early Stopping*

Ao treinar modelos robustos com capacidade representacional suficiente para sobre-ajustar uma tarefa, é possível observar que o erro de treinamento diminui constantemente com o passar do tempo, mas o erro do conjunto de validação volta a subir. A figura 4.1 mostra um exemplo desse comportamento que ocorre frequentemente. Isso significa que pode-se obter um modelo com

um melhor erro no conjunto de validação (e, portanto, um melhor erro no conjunto de teste) pela configuração dos parâmetros que resultou no menor erro no conjunto de validação.

Toda vez que o erro de validação melhora, uma cópia dos parâmetros do modelo pode ser guardada, e, no fim do treinamento, esses são os valores retornados, e não os últimos valores. O algorítimo de treinamento termina quando nenhum parâmetro tem melhoras em um número predeterminado de épocas em relação aos valores guardados do melhor erro de validação. Essa estratégia é conhecida como *early stopping* e é, provavelmente, a forma de regularização mais comumente usada em aprendizado profundo, devido a sua efetividade e simplicidade.

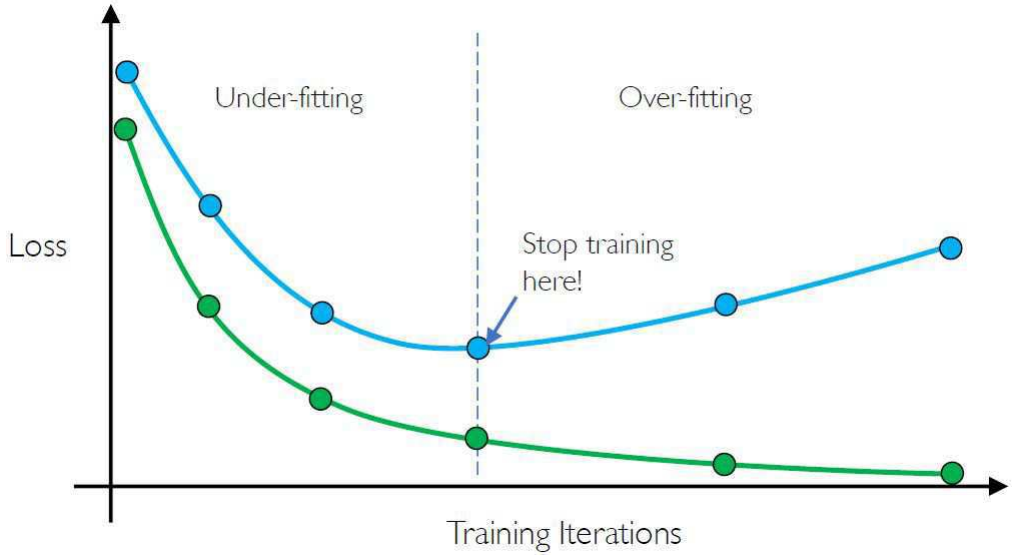

Figura 4.1: Comportamento da curva da função de custo para o conjunto de validação, em azul, e de treinamento, em verde.

Outra forma de pensar sobre *early stopping* é que é um algoritmo muito eficiente de seleção de hiper-parâmetro e, nessa ótica, o número de iterações de treinamento é mais um hiper-parâmetro, que passa a ser chamado de época. Controla-se a capacidade efetiva do modelo pela determinação de quantas épocas são feitas para ajustar o conjunto de treinamento.

A maioria dos hiper-parâmetros deve ser escolhida através de um custoso processo de tentativa e erro, onde se define o hiper-parâmetro no começo do treinamento, e treina-se o modelo por algumas iterações para observar seu efeito. Mas o hiper-parâmetro ”tempo de treinamento”é

capaz de, em um único treinamento, testar vários valores.

O único custo relevante de escolher esse hiper-parâmetro automaticamente por *early stopping* é ter que estimar o conjunto de validação periodicamente durante o treinamento. O custo dessas estimações pode ser reduzido pelo uso de um conjunto de validação que seja pequeno comparado ao conjunto de treinamento, ou estimar o erro do conjunto de validação menos frequentemente, obtendo uma estimativa de baixa resolução do tempo de treinamento ótimo. Um custo adicional é a necessidade de manter uma cópia dos melhores valores para os parâmetros, porém é um custo insignificante por poder guardar esses valores em qualquer tipo de memória.

Early stopping é uma forma discreta de regularização que requer quase nenhuma modificação no processo de treinamento, função de custo ou o conjunto de valores de parâmetros permitidos. Isso significa que é fácil de usar essa estratégia sem danificar as dinâmicas de aprendizado.

Early stopping pode ser usado sozinho ou em conjunção com outras estratégias de regularização. Ele também é útil por reduzir o custo computacional do processo de treinamento, pois, além da óbvia redução devido a limitação do número de iterações de treinamento, ele também provê regularização sem adicionar termos que penalizem a função de custo ou o cálculo do gradiente desses termos adicionais.

4.3 *Dropout*

Dropout é um método de regularização computacionalmente barato, mas poderoso. Especificamente, ele treina o conjunto de todas as sub-redes que podem ser formadas ao remover unidades ocultas ou de entrada da rede que se pretende treinar, como exemplificado na figura 4.2. Na maior parte das redes neurais modernas, pode-se efetivamente remover uma unidade da rede ao multiplicar seu valor de saída por zero.

Para treinar um rede com *dropout*, usa-se algorítimos baseados em mini lotes que realizem pequenos deslocamentos, como o gradiente descendente estocástico. A cada vez que dados são carregados em mini lotes, uma máscara randômica binária é aplicada a todas as unidades de en-

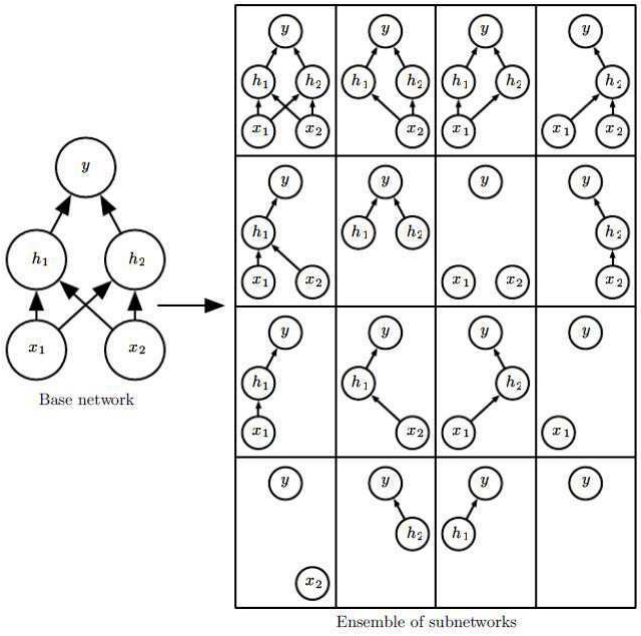

Figura 4.2: Exemplificação das possíveis sub-redes geradas a partir da exclusão de unidades da rede neural original

trada e ocultas da rede. Cada máscara é escolhida independentemente das outras, e a probabilidade de um valor da máscara ter valor 1 (fazendo com que a unidade seja incluída) é um hiper-parâmetro escolhido antes do treinamento começar. Tipicamente, uma unidade de entrada é incluída com probabilidade de 0.8, e uma unidade oculta com probabilidade de 0.5.

É importante entender que boa parte do poder do *dropout* vem do fato que o ruído da máscara é aplicado às unidades ocultas. Isso pode ser visto como uma forma altamente inteligente e adaptativa de destruição da informação contida na entrada. Por exemplo, se um modelo aprende que uma unidade oculta detecta o rosto através da identificação do nariz, então excluir essa unidade corresponde a apagar a informação que existe um nariz na imagem. Logo, o modelo terá que aprender como detectar o rosto através das outras unidades, seja redundantemente identificando o nariz, ou aprendendo a identificar outras características, como a boca, por exemplo.

5 Otimização

De todos os problemas de otimização em aprendizado profundo, o mais complicado é o treinamento da rede neural. É comum passar dias a meses, e o uso de várias máquinas, tentando resolver até um único caso de problema de treinamento de rede neural. Por esse problema ser tão importante e caro, um conjunto especializado de técnicas de otimização está sendo desenvolvido para resolvê-los.

Um caso particular, e muito usual, é encontrar os parâmetros w e θ da rede neural que reduza significantemente a função de custo $J(W)$, o que tipicamente inclui o cálculo da medição da performance em todo o conjunto de treinamento, assim como termos de regularização adicionais.

O processo de treinamento baseado em minimizar a média do erro de treinamento é conhecido como minimização do risco empírico, porém ele está sujeito a um sobre-ajuste. Modelos com alta capacidade representacional podem simplesmente memorizar o conjunto de treinamento. Em muitos casos, a minimização do risco empírico não é realmente possível. Os algorítimos de otimização modernos mais eficientes são os baseados em gradiente descendente, porém muitas funções de custo eficientes não possuem derivadas úteis (as derivadas tem valor 0 ou é indefinida em qualquer lugar). Esses dois problemas fazem com que se tenha uma abordagem um pouco diferente, com o uso dos métodos de substituição das funções de custo e do *early stopping*.

5.1 Taxa de aprendizado

A taxa de aprendizado regula o tamanho dos passos da atualização dos parâmetros obtidos pelo back-propagation. Uma taxa de aprendizado pequena converge de forma muito devagar e pode ficar preso em um mínimo local. Uma taxa de aprendizado grande pode ultrapassar o mínimo global, se tornando instável e divergindo. Uma taxa de aprendizado adequado faz com que o algoritmo de aprendizado convirja suavemente e evite os mínimos locais.

Essa taxa de aprendizado adequado pode ser obtido de forma manual, por tentativa e erro. Porém, para cenários mais complexos, alguns algoritmos de aprendizado utilizam taxas de

aprendizado adaptáveis, que se adequam a cada cenário utilizando diversos métodos, seja uma média das taxas anteriores, o tamanho do gradiente calculado, quão rápido o aprendizado está acontecendo, o tamanho dos pesos específicos, entre outros.

5.2 Lotes e mini-lotes

Algoritmos de otimização que usam conjuntos de treinamento inteiro são chamados de métodos de gradiente em lotes, ou *batch*, pois são processados todos os exemplos de treinamento simultaneamente em um grande lote. Os Algoritmos que usam somente um único exemplo de cada vez são chamados de métodos de gradientes estocásticos.

A maioria dos algoritmos usados em aprendizado profundo se localizam entre os dois, usando mais de um exemplo de treinamento, mas menos do que todos ao mesmo tempo. Eles são tradicionalmente chamados de métodos de gradiente estocásticos em mini-lotes, mas, devido ao seu grande uso, passaram a ser chamados simplesmente de métodos estocásticos.

O tamanho dos mini-lotes são genericamente guiados pelos seguintes fatores:

- Lotes maiores provêm uma estimativa mais precisa do gradiente, porém exigem um maior esforço computacional.
- Dispositivos com arquiteturas mais robustas ou de multi núcleos são subutilizados por lotes extremamente pequenos, motivando o uso de um tamanho mínimo absoluto do lote, onde não há uma redução do tempo de processamento de lotes menores do que o mínimo definido.
- Pequenos lotes podem oferecer um efeito regularizante, talvez devido ao ruído que eles adicionam ao processo de aprendizado. Erro de generalização geralmente é melhor para lotes de tamanho 1. Treinar com um lote tão pequeno pode necessitar de uma pequena taxa de aprendizado para manter a estabilidade, devido a alta variância do gradiente estimado. Porém, pela taxa de aprendizado e maior número de passos para observar todo o conjunto de treinamento, o tempo de execução se torna mais alto.

5.3 Gradiente Descendente Estocástico ou SGD

Gradiente descendente estocástico e seus variantes são os algoritmos de otimização mais usados. É possível obter uma estimativa não tendenciosa do gradiente pela media dos gradientes obtidos em cada mini-lote.

Um parâmetro crucial para o algoritmo SGD é a taxa de aprendizado. O SGD pode ser usado com uma taxa de aprendizado fixa, porém, na prática, geralmente é necessário diminuir essa taxa gradualmente pelas iterações. Isso é devido ao estimador de gradiente do SGD, que introduz um fonte de ruído (a amostragem aleatória dos exemplos de treinamento) que não desaparece nem ao atingir um mínimo.

5.4 AdaGrad

O algoritmo AdaGrad adapta a taxa de aprendizado de todos os parâmetros do modelo ao escalá-los inversamente proporcional a raiz quadrada da soma de todos os valores dos gradientes ao quadrado. Os parâmetros com derivadas parciais da função de custo maiores tem uma diminuição da taxa mais rápida, enquanto parâmetros com derivadas parciais menores tem uma diminuição menor da sua taxa de aprendizado.

No contexto de otimização convexa, o algoritmo AdaGrad usufrui algumas propriedades teóricas desejáveis. Porém, empiricamente, ao treinar modelos de redes profundas, a acumulação dos gradientes ao quadrado do começo do treinamento pode resultar em um prematura e excessiva diminuição na taxa de aprendizado efetiva. AdaGrad performa bem em alguns casos, mas não em todos os modelos de aprendizado profundo.

5.5 RMSProp

O algoritmo RMSProp modificou o AdaGrad para performar melhor em uma configuração não convexa ao alterar a acumulação do gradiente. A AdaGrad é projetada para convergir

rapidamente quando aplicada a uma função convexa. Quando aplicada uma função não-convexa para treinar a rede, a trajetória de aprendizado pode passar por muitas estruturas diferentes até eventualmente atingir uma região localmente convexa. Ele encolhe a taxa de aprendizado de acordo com o histórico inteiro do gradiente ao quadrado e pode fazer com que a taxa se torne muito pequena até atingir tal região convexa.

RMSProp utiliza uma média exponencial com decaimento para descartar o histórico extremamente antigo, podendo convergir mais rapidamente a região convexa, como se fosse uma instância do algoritmo AdaGrad inicializada nessa região. Empiricamente, RMSProp tem se mostrado um algoritmo de otimização efetivo e prático para redes neurais profundas, e vem sendo muito utilizado também.

5.6 Adam

O algoritmo Adam é outro algoritmo de otimização com taxa de aprendizado adaptativo. Ele é considerado como um variante do RMSProp com a utilização do momentum da taxa. Adam é genericamente considerado como uma escolha de parâmetros robusta, apesar da taxa de aprendizado algumas vezes precisar ser modificado para o padrão sugerido.

Foi discutido uma série de algoritmos de otimização, com cada um buscando endereçar um desafio diferente para otimizar modelos profundos ao adaptar a taxa de aprendizado para cada parâmetro do modelo. Mas qual algoritmo devemos usar? Infelizmente, atualmente não há um consenso nesse ponto. Apesar de existir diversos estudos e pesquisas com comparações, nenhum algoritmo emergiu como o melhor, principalmente por ser dependente de muitos fatores pequenos.

Atualmente, os algoritmos de otimização mais populares são o SGD, o RMSProp e suas respectivas variantes com momentum. Porém, essa popularidade se deve mais a familiaridade do usuário com tal algoritmo (facilitando a sintonização dos hiper-parâmetros), do que de fato por performance.

6 Topologia

Outro aspecto chave para a criação de uma rede neural é determinar sua topologia. A topologia se refere a estrutura geral da rede: quantas camadas ela deve possuir, quantas unidades cada camada deve conter e como essas unidades devem ser conectadas entre si. De modo geral, enquanto redes mais largas possuem uma maior capacidade representacional do modelo, ao mesmo tempo em que exigem um maior esforço computacional para ser treinada como desejado, redes mais profundas são capazes de utilizar um menor numero de unidades por camada, muito menos parâmetros e geralmente possuem uma melhor generalização, porém, tendem a ter uma otimização mais complicada.

Em uma camada de rede neural padrão, descrita por uma transformação linear via a matriz de pesos **W**, cada unidade da camada de entrada é conectada a cada unidade da camada de saída. Porém, muitas redes especializadas possuem menos conexões, de modo que cada unidade da camada de entrada é conectada a apenas um subconjunto de unidades da camada de saída. Essa estratégia também diminui os números de parâmetros e o esforço computacional necessário para a rede, mas é altamente dependente do problema a que se propõe solucionar.

Ainda é possível fazer com que as camadas de uma rede não estejam conectadas em cadeia. Apesar de essa ser a prática mais comum, muitas topologias constroem uma cadeia principal e adicionam conexões que saltam uma camada ou mais. Essas conexões fazem com que seja mais fácil do gradiente fluir da camada de saída para as camadas mais próximas da entrada.

A topologia ideal de uma rede para determinada tarefa deve ser encontrada por tentativas e experimentos, com escolhas guiadas através de uma monitoração do erro no conjunto de validação e de sua eficiência computacional. Além das redes MLPs, existem outros tipos de arquiteturas, como as redes neurais convolucionais, as redes neurais recorrentes, autoencoders, *Deep Neural Network Capsules*, entre outros.

7 Metodologia Prática

Para resolução de um problema didático, foi desenvolvida uma rede neural MLP completamente conectada para realizar a simples tarefa de identificar números de 0 a 9 escritos à mão, com o objetivo de comparar a eficiência das funções de ativação, e verificar o efeito de mudanças na regularização e otimização em cada uma delas. Foi usada a linguagem de programação Python como ferramenta de desenvolvimento, utilizando a plataforma do Tensorflow, junto com a Keras, na construção, obtenção do conjunto de dados, e no cálculo e visualização da performance da rede. Todo o código escrito para esse problema encontra-se no anexo 1.

7.1 Conjunto de dados

O conjunto de dados utilizado foi o MNIST, que consiste em 70 mil imagens com 28x28 pixels em escala de cinza de dígitos de 0 a 9 escritos à mão, onde 60 mil são usadas no treinamento e 10 mil usados como conjunto de teste. Todo o conjunto de dados pode ser importado diretamente da Keras, necessitando apenas de um tratamento dos dados, onde foi feita uma transformação da matriz de pixel em um vetor unidimensional, e uma normalização dos seus valores para que fiquem entre 0 e 1. O código da importação e do tratamento do conjunto de dados é mostrado a seguir.

```
import tensorflow as tf

(x_input, y_input),(x_test, y_test)=tf.keras.datasets.mnist.load_data()
x_input = x_input.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255
```

7.2 Topologia e funções de ativação

Foi escolhida uma topologia bastante utilizada e validada por sua simplicidade e eficiência. Apesar de simples, possui bastante capacidade representacional tornando possível observar os

efeitos de sub e sobre-ajuste e o efeito de cada mudança na rede com um baixo número de épocas. Foram alocadas 784 unidades de entrada (uma para cada pixel de uma imagem), duas camadas completamente conectadas com 512 unidades ocultas, e uma camada de saída com 10 unidades com a função de ativação softmax, uma para cada classe (dígitos de 0 a 9), para que pudesse ser feita uma classificação.

Foram treinadas 5 redes semelhantes, cada uma usando uma função de ativação diferente nas camadas ocultas. As funções de ativação usadas para comparação foram a ”Sigmoid”, ”Tanh”, ”Softmax”, ”ReLU”e a ”ELU”. A função que define a topologia da rede é mostrada a seguir.

```
def net(x_train, y_train, x_test, y_test, epoch=5, activation='relu'):
    x_input = tf.keras.layers.Input(shape=(784,))
    l1 = tf.keras.layers.Dense(512, activation=activation)(x_input)
    l2 = tf.keras.layers.Dense(512, activation=activation)(l1)
    logit = tf.keras.layers.Dense(10, activation='softmax')(l2)

    model = tf.keras.Model(inputs=x_input, outputs=logit)
```

As 5 redes foram treinadas durante 50 épocas cada uma, número alto para a simplicidade do problema e a capacidade representacional da rede, para que fosse possível observar todo o seu comportamento durante o aprendizado. Suas 5 chamadas (uma para cada função de ativação) fica na seguinte forma:

```
epoch = 50

# cria uma rede neural para cada função de ativação
sig_train_score, sig_test_score = net(x_input, y_input,
                                      x_test, y_test,
                                      epoch, activation='sigmoid')
tanh_train_score, tanh_test_score = net(x_input, y_input,
                                        x_test, y_test,
```

```
                                        epoch, activation='tanh')
relu_train_score, relu_test_score = net(x_input, y_input,
                                        x_test, y_test,
                                        epoch, activation='relu')
softmax_train_score, softmax_test_score = net(x_input, y_input,
                                              x_test, y_test,
                                            epoch, activation='softmax')
elu_train_score, elu_test_score = net(x_input, y_input,
                                      x_test, y_test,
                                      epoch, activation='elu')
```

7.3 Regularização e otimização

Com a intenção de observar todos os comportamentos e reações desejadas e indesejadas da rede neural, a única regularização feita foi a separação do conjunto de validação, com o tamanho de 20% do conjunto de treinamento, ou seja, 1200 imagens. O custo e a precisão das predições da rede no conjunto de validação foram calculados a cada época.

Inicialmente, a otimização foi feita a partir do algorítimo ”SGD”, que é baseado em gradiente descente estocástico com a taxa de aprendizado, com valor inicial padrão de 0.01. O tamanho do mini-lote foi de 64 imagens para cada passo do gradiente. A função de custo a ser minimizada foi a ”sparse categorical crossentropy”, que é um algorítimo de entropia cruzada para classes, onde o índice do vetor de saída da rede com maior valor é comparado com o valor numérico correto fornecido pelo conjunto de dados.

Os códigos para a regularização e otimização das redes foram adicionados à função ”net”que as define, mostrada anteriormente. O código adicionado é exibido a seguir.

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              metrics=['accuracy'])
```

```
fit = model.fit(x_train, y_train,
                batch_size=64,
                epochs=epoch,
                validation_split=0.2)
```

7.4 Resultados

Por fim, foi adicionado à mesma função o código para guardar as informações de performance das redes durante o treinamento. Além disso, foi adicionado também um método para executar os dados de teste e obter a performance também no conjunto de teste, como mostrado abaixo.

```
history = fit.history

test_scores = model.evaluate(x_test, y_test, verbose=2)

return history, test_scores
```

Ambas as informações foram retornadas da função, e passaram por um tratamento para que pudessem ser plotadas para visualização da performance e comportamento, mostrado no código completo no anexo 1. Os custos e as precisões para cada função de ativação das camadas ocultas são mostradas na figura 7.1. O custo e precisão de cada função de ativação durante o conjunto de treinamento e de validação também são mostrados na figura 7.2. O resultado da execução do conjunto de teste é mostrado na tabela 1.

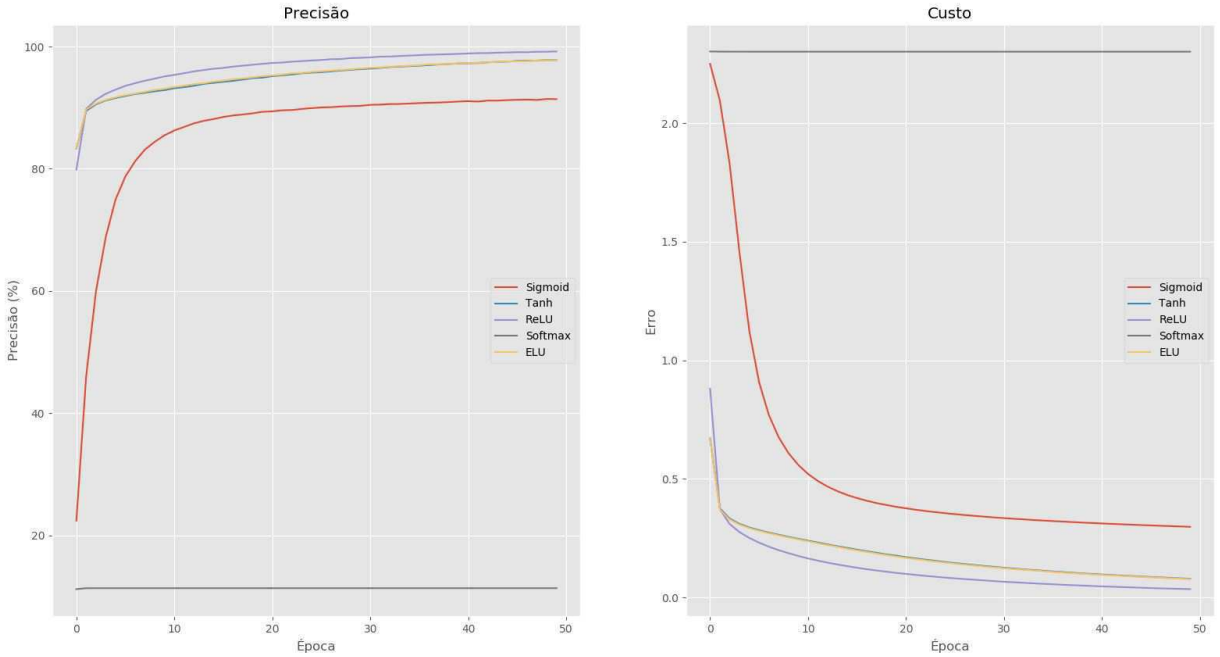

Figura 7.1: Custo e precisão das cinco funções de ativação na primeira configuração da rede

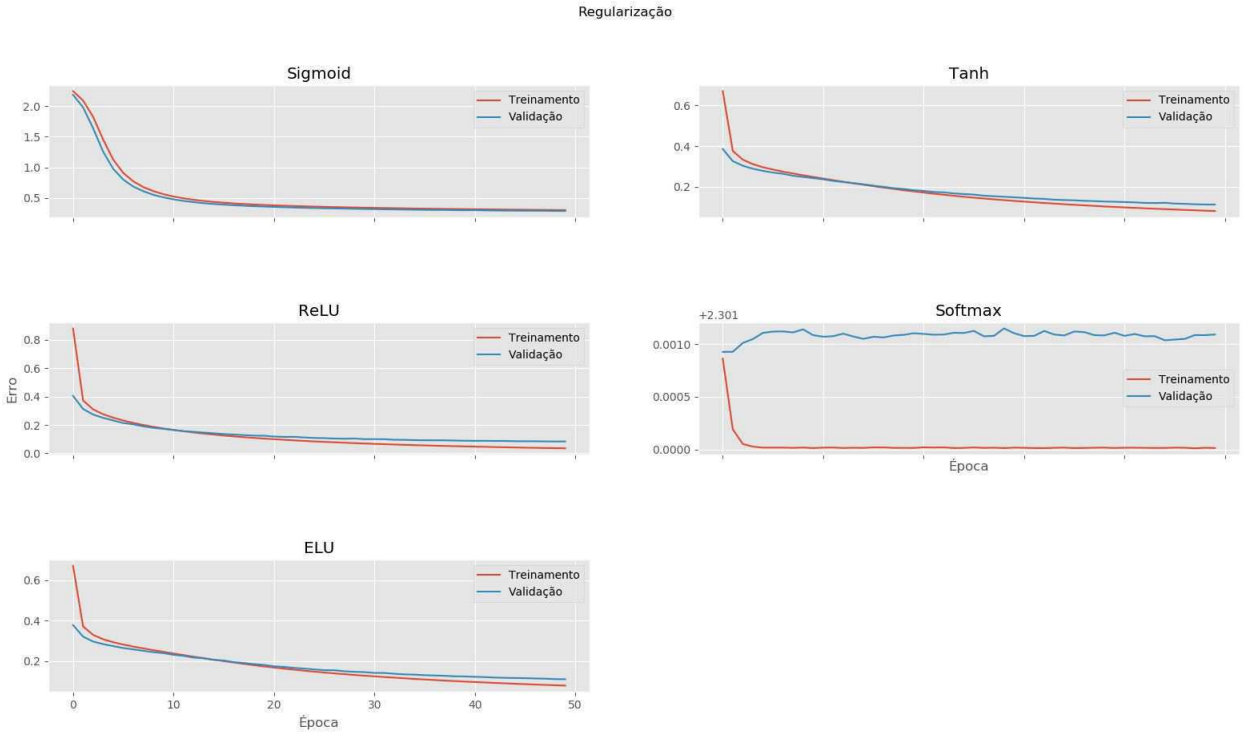

Figura 7.2: Custo das cinco funções de ativação no conjunto de treinamento e de validação

Função de ativação	Precisão	Custo
Sigmoid	91.67%	0.2886
Tanh	96.96%	0.1017
ReLU	97.73%	0.0745
ELU	97.04%	0.1018
Softmax	11.35%	2.3010

Tabela 1: Resultado da execução do conjunto de teste nas 5 redes treinadas

Com esses resultados, é possível observar uma performance semelhantemente boa para as funções de ativação ReLU, ELU e Tanh. É possível observar também a ineficácia esperada da função Softmax em camadas ocultas, apesar da sua ótima aplicação como função de ativação da camada de saída. A função Sigmoid, também de forma esperada, tem uma performance um pouco abaixo das três primeiras, já que é um problema de classificação, e não um problema probabilístico, que é onde a função Sigmoid se sobressai.

Cada rede demorou em torno de 5 minutos para ser treinado por 50 épocas em um computador de poder de processamento médio, um baixo tempo para o resultado obtido em uma sistema de aprendizado profundo com 669,706 parâmetros treináveis. Isso mostra o poder de redes como essa em solucionar problemas que, de outra forma, seriam complexos e de difícil resolução.

7.5 Refatoração

Apesar alta precisão e baixo custo, decidiu-se tentar otimizar ainda mais a rede, pois foi possível observar que, mesmo com 50 épocas, a rede não chegou ao seu valor mínimo do custo do conjunto de validação. Para a o processo de otimização, só foi treinada uma única rede, com função de ativação Tanh, porém poderia ter sido feita com qualquer outra das três funções com melhor desempenho para o problema.

Ao aumentar o valor inicial para 0.1, o treinamento se torna mais rápido, porém a rede não consegue convergir para um valor mínimo, e fica com o custo oscilando, o que não é algo

desejado. Então, alterou-se o algoritmo de otimização para um algoritmo que possuísse uma taxa de aprendizado adaptativo. O algoritmo utilizado foi o RMSProp, com a taxa de aprendizado com valor inicial 0.001. O gráfico do custo de validação e treinamento é mostrado na figura 7.3.

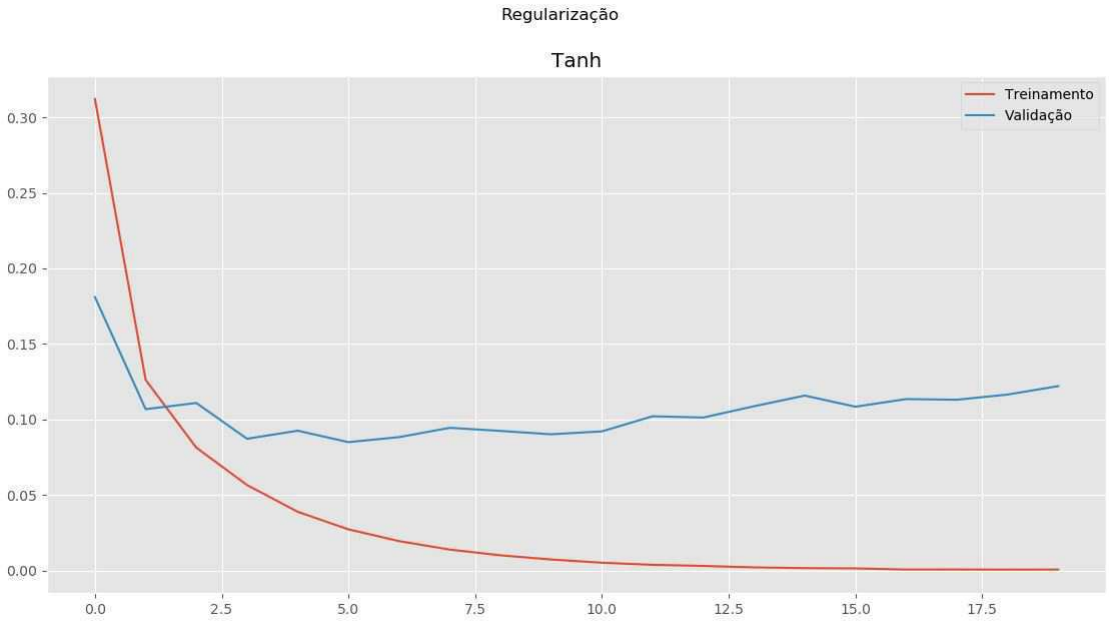

Figura 7.3: Custo da conjunto de validação e treinamento para a nova configuração da rede

O resultado do custo e precisão do conjunto de teste foram, respectivamente, 0.1041 e 98.24%. Possivelmente, apesar dos bons resultados, existiu um efeito de sobre-ajuste, onde os melhores parâmetros da rede seriam em torno da época 5. Logo, mesmo com cada época sendo treinada com o dobro do tempo, o tempo total de treinamento ainda se torna bem menor.

Para tentar otimizar ainda mais a rede, foram adicionados dropouts a cada camada oculta da rede como forma de regularização, com probabilidade de inclusão de 80% para unidades de entrada e 50% para unidades ocultas, na tentativa de diminuir ainda mais o erro de generalização da rede. A arquitetura passa a ser definida como no código exibido abaixo, e os novos resultados são mostrados na figura 7.4. O código final inteiro pode ser encontrado no anexo 2.

```
x_input = tf.keras.layers.Input(shape=(784,))
dropout = tf.keras.layers.Dropout(0.2)(x_input)
```

```
l1 = tf.keras.layers.Dense(512, activation=activation)(dropout)
dropout2 =tf.keras.layers.Dropout(0.5)(l1)

l2 = tf.keras.layers.Dense(512, activation=activation)(dropout2)
logit = tf.keras.layers.Dense(10, activation='softmax')(l2)
```

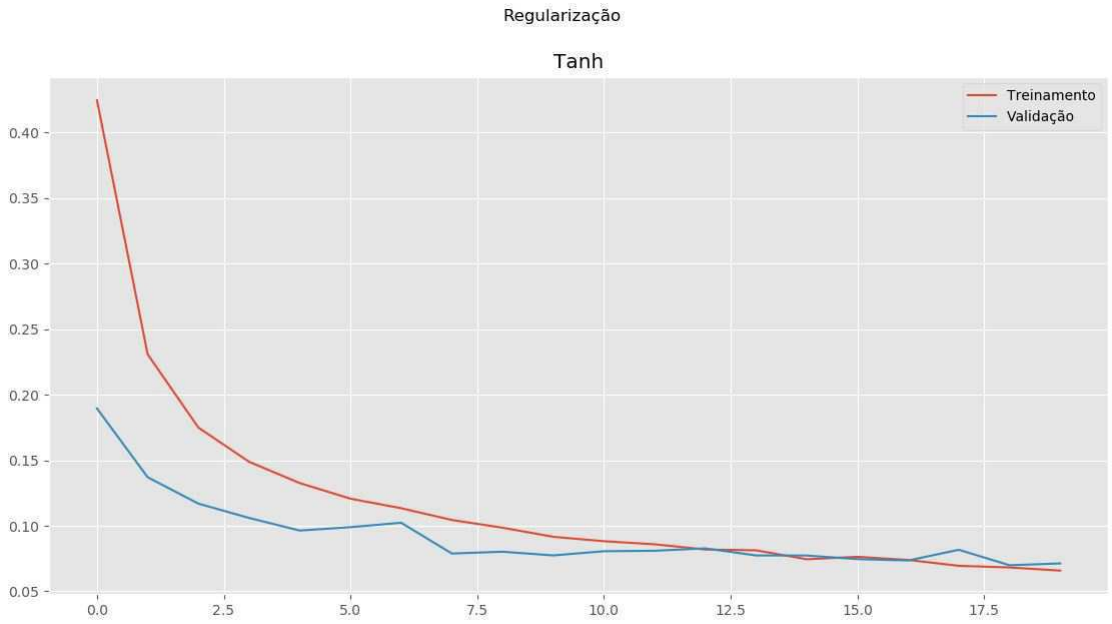

Figura 7.4: Custo da conjunto de validação e treinamento para a configuração da rede com dropout

A rede teve um custo de 0.0625 e uma precisão 97.97% durante o conjunto de teste. É possível notar que os dropouts conseguiram evitar um efeito de sobre-ajuste e realmente obtiveram um custo ainda menor. Como o sobre-ajuste foi evitado, seria possível aumentar o número de épocas e adicionar mais um método de regularização, o *early stopping*, para poder salvar os melhores valores dos parâmetros, obtendo uma rede com uma ótima eficiência.

Porém, é uma decisão de projeto opcional, devido aos valores de performance já estarem bastante satisfatórios sem esse método, e por precisar de um número maior de épocas, seria necessário um maior tempo de treinamento e poder computacional para adicioná-lo. O *early stopping* pode ser adicionado e todo o modelo pode ser salvo para implementação futura sem precisar ser treinado novamente com os seguintes códigos à rede:

```
tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                 patience=3,
                                 restore_best_weights=True)
model.save('model.h5')
```

8 Redes neurais convolucionais

Redes neurais convolucionais, ou CNNs, são um tipo de redes especializadas em processamento de dados que possuem uma topologia semelhante a uma grade. Exemplos incluem dados temporais, que podem ser imaginados como uma grade 1D tomando amostras em intervalos regulares de tempo, e dados de imagens, que podem ser pensados como uma grade 2D de pixels. Redes convolucionais vem tendo grandes sucessos nessas e em outras aplicações práticas. Redes convolucionais são simplesmente redes neurais que usam a convolução no lugar da multiplicação matricial em pelo menos uma camada da sua arquitetura. A convolução alavanca três importantes ideias que podem melhorar o sistema de aprendizado profundo: interações esparsadas, compartilhamento de parâmetros e representações equivariantes. Além disso, convolução provê um meio de trabalhar com entradas de diferentes tamanhos.

Camadas de redes neurais tradicionais usam multiplicação de matrizes de parâmetros, com parâmetros separados descrevendo a interação entre cada unidade de entrada com cada unidade de saída. Isso significa que cada unidade de saída interage com todas as unidades de entrada. No entanto, as redes convolucionais geralmente têm interações esparsadas (também conhecidas como conectividade esparsada ou pesos esparsados). Isso é feito tornando o núcleo, ou kernel, menor que a entrada. Por exemplo, ao processar imagens, a imagem de entrada pode ter milhares ou milhões de pixels, mas podemos detectar recursos pequenos e significativos, como arestas, com núcleos que ocupam apenas dezenas ou centenas de pixels. Isso significa que precisamos armazenar menos parâmetros, o que reduz os requisitos de memória do modelo e melhora sua eficiência. Isso também significa que calcular a saída requer menos operações, como mostrado na figura 8.1. Essas melhorias na eficiência geralmente são bastante grandes.

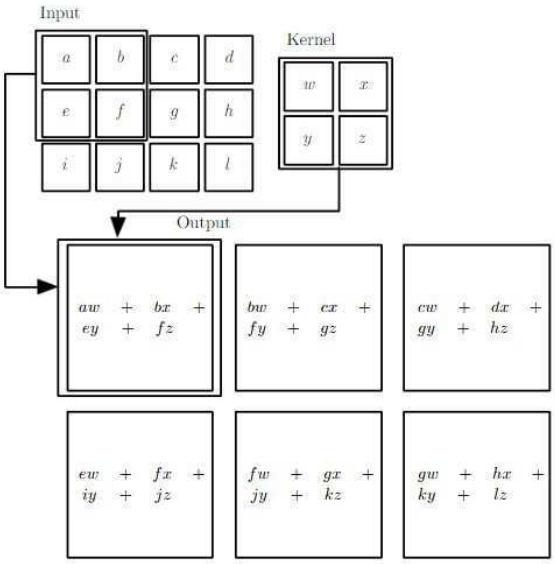

Figura 8.1: Exemplificação dos cálculos realizados na camada convolucional de um rede por seu núcleo.

O compartilhamento de parâmetros refere-se ao uso do mesmo parâmetro para mais de uma função em um modelo. Em uma rede neural tradicional, cada elemento da matriz de peso é usado exatamente uma vez ao calcular a saída de uma camada. É multiplicado por um elemento da entrada e nunca mais é reusado. Como sinônimo de compartilhamento de parâmetros, pode-se dizer que a rede possui pesos vinculados, porque o valor do peso aplicado a uma entrada está vinculado ao valor de um peso aplicado em outro local. Em uma rede neural convolucional, cada elemento do núcleo é usado em todas as posições da entrada (exceto talvez alguns dos pixels da borda, dependendo das decisões de design em relação às bordas). O compartilhamento de parâmetros usado pela operação de convolução significa que, em vez de aprender um conjunto separado de parâmetros para cada conexão, aprende-se apenas um conjunto.

No caso de convolução, a forma específica de compartilhamento de parâmetros faz com que a camada tenha uma propriedade chamada equivariância à translação. Dizer que uma função é equivariante significa que, se a entrada mudar, a saída mudará da mesma maneira. Ao processar dados temporais, isso significa que a convolução produz um tipo de linha do tempo que mostra quando diferentes características aparecem na entrada. Se movermos um evento para um tempo futuro na entrada, a mesma exata representação aparecerá na saída, logo depois. Semelhantemente

nas imagens, a convolução cria um mapa em 2D onde certas características aparecem na entrada. Se movermos o objeto na entrada, sua representação moverá a mesma quantidade na saída. Isso é útil quando sabe-se que alguma função utilizada em um pequeno número de pixels vizinhos é eficientemente aplicada a vários locais da entrada.

Por exemplo, ao processar imagens, é útil detectar arestas na primeira camada de uma rede convolucional. As mesmas arestas aparecem mais ou menos em toda parte da imagem; portanto, é prático compartilhar parâmetros em toda a imagem. Em alguns casos, talvez não deseja-se compartilhar parâmetros em toda a imagem. Por exemplo, se estiver processando imagens que são cortadas para serem centralizadas no rosto de um indivíduo, provavelmente deseja-se extrair diferentes características em diferentes locais - a parte da rede que processa a parte superior do rosto precisa procurar sobrancelhas, enquanto a parte da rede que processa a parte inferior do rosto precisa procurar um queixo.

8.1 Pooling

Uma camada típica de uma rede convolucional consiste em três estágios. No primeiro estágio, a camada executa várias convoluções em paralelo para produzir um conjunto de ativações lineares. No segundo estágio, cada ativação linear é executada através de uma função de ativação não linear, como a função de ativação ReLU. Às vezes, esse estágio é chamado de estágio detector. No terceiro estágio, usamos uma função de pooling para modificar ainda mais a saída da camada. Pode-se ainda adicionar camadas de regularização com *dropout* em cada um desses estágios se desejar, de forma semelhante a redes MLPs.

Uma função de pooling substitui a saída da rede em um determinado local por uma estatística resumida das saídas próximas. Por exemplo, a operação de pooling máximo retorna o valor de saída máxima dentro de uma vizinhança retangular, como mostrado na figura 8.2. Outras funções populares de pooling incluem a média de uma vizinhança retangular ou uma média ponderada com base na distância do pixel central.

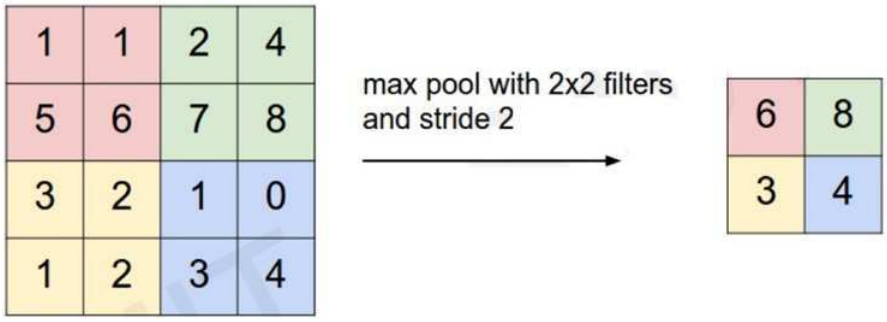

Figura 8.2: Exemplificação da ação de uma camada de polling com a operação de pooling máximo.

8.2 Eficiência

As aplicações das redes convolucionais modernas geralmente envolvem redes que contêm mais de um milhão de unidades. Implementações poderosas que exploram recursos de computação paralela são essenciais para um baixo tempo de treinamento, por permitir o cálculo da convolução com diversos núcleos ao mesmo tempo, cada um filtrando uma característica diferente em toda a imagem. Em muitos casos, no entanto, também é possível acelerar o treinamento selecionando um algoritmo de convolução apropriado.

Convolução é equivalente a converter a entrada e o núcleo no domínio da frequência usando uma transformada de Fourier, realizando a multiplicação pontual dos dois sinais e convertendo de volta ao domínio do tempo usando uma transformada de Fourier inversa. Para alguns tamanhos de problemas, isso pode ser mais rápido que a implementação direta de convolução discreta.

8.3 Exemplo prático

Para exemplificação, foi desenvolvido o código para resolver o mesmo problema do tópico anterior, porém utilizando um rede convolucional, com uma arquitetura já validada para o problema e o conjunto de dados. A otimização e regularização foram exatamente as mesmas, e a função de ativação usada na camada oculta densa também foi a Tanh. Em relação as camadas

convolucionais, foram adicionadas duas, cada uma com um núcleo de tamanho 3x3 e uma função de ativação ReLU, porém a primeira fazendo a filtragem de 24 características diferentes em cada imagem, e a segunda filtrando por 36 características.

Foram adicionadas também duas camadas de polling, com o método de polling máximo, e uma janela retangular de 2x2. Vale ressaltar que na rede convolucional, não se faz necessário a transformação das entradas do conjunto de dados para um vetor unidimensional, pois a rede convolucional aceita uma matriz bidimensional em sua entrada. Porém, ao fluir a informação para a camada densa, se faz necessária essa transformação, que é feita por uma camada implementada no Keras, chamada de *Flatten*. O código de definição da arquitetura que foi modificada se encontra destacado abaixo, e o código inteiro pode ser visualizado no anexo 3.

```
x_input = tf.keras.layers.Input(shape=(28, 28, 1))

l1 = tf.keras.layers.Conv2D(24,
                            kernel_size=(3, 3),
                            activation='relu')(x_input)
l1 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(l1)



l2 = tf.keras.layers.Conv2D(36, (3, 3), activation='relu')(l1)
l2 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(l2)

l3 = tf.keras.layers.Flatten()(l2)
l4 = tf.keras.layers.Dense(128, activation=activation)(l3)
logit = tf.keras.layers.Dense(10, activation='softmax')(l4)

model = tf.keras.Model(inputs=x_input, outputs=logit)
```

Como o objetivo foi somente exemplificar uma rede convolucional, é possível observar a falta de dropouts nas camadas ocultas que poderiam ter sido adicionados e não se fez novas

tentativas de otimizar e regularizar a rede. Apesar de ter obtido uma performance semelhante a rede MLP otimizada, o tempo de execução de cada época se tornou ainda maior, devido a falta de uso do paralelismo computacional, que é essencial para os cálculos simultâneos das convoluções para filtrar as diferentes características em redes convolucionais.

9 Redes neurais recorrentes

Redes neurais recorrentes, ou RNNs, são uma família de redes neurais para o processamento de dados sequenciais, ou seja, uma família de redes neurais que possuem algum tipo de memória. Assim como uma rede convolucional é uma rede neural especializada para processar uma grade de valores X, como uma imagem, uma rede neural recorrente é uma rede neural especializada para processar uma sequência de valores $x^{(1)}, ..., x^{(\tau)}$.

Assim como as redes convolucionais podem ser facilmente escalonadas para imagens com grande largura e altura, e algumas redes convolucionais podem processar imagens de tamanho variável, as redes recorrentes podem ser escaladas para sequências muito mais longas do que seria prático para redes sem especialização baseada em sequências. A maioria das redes recorrentes também pode processar sequências de comprimento variável. Além disso, redes recorrentes também tiram proveito de uma das primeiras ideias encontradas no aprendizado de máquina e nos modelos estatísticos da década de 1980: compartilhamento de parâmetros em diferentes partes de um modelo.

O compartilhamento de parâmetros possibilita estender e aplicar o modelo a exemplos de diferentes formas (diferentes comprimentos, no caso) e generalizá-las. Se a rede possuísse parâmetros separados para cada valor de entrada da sequência, ou etapa de tempo como é chamada em RNNs, não poderia ser generalizada para sequências de comprimentos não vistos durante o treinamento, nem compartilhar a força estatística em diferentes comprimentos de sequência e em diferentes posições no tempo. Esse compartilhamento é particularmente importante quando uma informação específica pode ocorrer em várias posições dentro da sequência.

Uma ideia relacionada é o uso da convolução através de uma sequência temporal 1D. A operação de convolução permite que uma rede compartilhe parâmetros ao longo do tempo, mas é superficial. A saída da convolução é uma sequência em que cada membro da saída é uma função de um pequeno número de membros vizinhos da entrada. A ideia do compartilhamento de parâmetros se manifesta na aplicação do mesmo núcleo de convolução a cada índice de tempo.

As redes recorrentes compartilham parâmetros de maneira diferente. Cada membro da saída é uma função dos membros anteriores da saída. Ele é produzido usando a mesma regra de atualização aplicada às saídas anteriores. Essa formulação recorrente resulta no compartilhamento de parâmetros através de um grafo computacional muito profundo.

Para simplificar a exposição, nos referimos às RNNs como uma rede operando em uma sequência que contém vetores x(t) com o índice de intervalo de tempo t variando de 1 a τ. Na prática, as redes recorrentes geralmente operam em mini lotes dessas sequências, com um comprimento de sequência τ diferente para cada membro dele. Omite-se os índices de mini lote para simplificar a notação. Além disso, o índice de etapas do tempo não precisa se referir literalmente à passagem do tempo no mundo real. Às vezes, refere-se apenas à posição na sequência. As RNNs também podem ser aplicadas em duas dimensões em dados espaciais, como imagens, e mesmo quando aplicadas a dados que envolvem tempo, a rede pode ter conexões que retrocedem no tempo, desde que toda a sequência seja observada antes de ser fornecida à rede.

Dependendo do critério de treinamento, pode-se manter seletivamente alguns aspectos da sequência passada com mais precisão do que outros. Por exemplo, se a RNN for usada na modelagem de linguagem estatística, normalmente para prever a próxima palavra dada as palavras anteriores, armazenar todas as informações na sequência de entrada até o tempo t pode não ser necessário; armazenar apenas informações suficientes para prever o restante da frase é suficiente. A situação mais exigente é quando solicitamos que h(t) seja rico o suficiente para permitir recuperar aproximadamente a sequência de entrada, como nas estruturas de autoencoder.

O processo de recorrência apresenta duas grandes vantagens:

- Independentemente do tamanho da sequência, o modelo aprendido sempre tem o mesmo

tamanho de entrada, porque é especificado em termos de transição de um estado para outro estado, em vez de ser especificado em termos de um histórico de comprimento variável de estados.

- É possível usar a mesma função de transição f com os mesmos parâmetros a cada etapa de tempo.

Esses dois fatores tornam possível aprender um único modelo que opera em todas as etapas de tempo e para todos os comprimentos de sequência, em vez de precisar aprender um modelo separado g(t) para todas as etapas de tempo possíveis. O aprendizado de um único modelo compartilhado permite que a generalização sequencie comprimentos que não apareceram no conjunto de treinamento e permite que o modelo seja estimado com muito menos exemplos de treinamento do que seria necessário sem o compartilhamento de parâmetros.

Alguns exemplos de padrões de design importantes para redes neurais recorrentes incluem:

- Redes recorrentes que produzem uma saída a cada etapa do tempo e têm conexões recorrentes entre unidades ocultas.
- Redes recorrentes que produzem uma saída a cada etapa do tempo e têm conexões recorrentes somente a partir da saída de uma etapa para a unidade oculta da próxima etapa.
- Redes recorrentes com conexões recorrentes entre unidades ocultas, que leiam uma sequência inteira e produzam uma única saída.

Essas opções permitem uma troca entre performance e tempo de execução para treinamento.

Todas as redes recorrentes que foram consideradas até agora têm uma estrutura "causal", o que significa que o estado no momento t captura apenas informações do passado, $x(1), ..., x(t-1)$ e a entrada atual $x(t)$. Alguns dos modelos que foram discutidos também permitem que as informações dos valores y anteriores afetem o estado atual quando esses valores de y estiverem disponíveis.

Em muitas aplicações, no entanto, deseja-se gerar uma previsão de y(t) que pode depender de toda a sequência de entrada. Por exemplo, no reconhecimento de fala, a interpretação correta do som atual como fonema pode depender dos próximos fonemas: se houver duas interpretações da palavra atual que sejam ambos acusticamente plausíveis, talvez tenha-se que olhar para o futuro (e o passado) para desambiguá-los. Isso também se aplica ao reconhecimento de texto manuscrito e a muitas outras tarefas de aprendizado de sequência a sequência.

9.1 Rede recorrente bidirecional

Redes neurais recorrentes bidirecionais (ou RNNs bidirecionais) foram inventadas para atender a essa necessidade. Elas têm sido extremamente bem-sucedidas em aplicações que possuem essa necessidade, como reconhecimento de texto manuscrito, reconhecimento de fala e bioinformática. Como o nome sugere, as RNNs bidirecionais combinam uma camada recorrente que avança no tempo, começando no início da sequência, com outra camada recorrente que retrocede no tempo, começando no final da sequência até a etapa de tempo x(t).

Essa ideia pode ser estendida naturalmente a entradas bidimensionais, como imagens, tendo quatro camadas recorrentes, cada uma seguindo uma das quatro direções: cima, baixo, esquerda, direita. Em cada ponto (i, j) de uma grade 2D, teria uma saída $O_{i,j}$ que poderia então calcular uma representação que capturaria principalmente informações locais, mas também poderia depender de entradas de longo alcance, se a RNN puder aprender a transportar essas informações. Comparados a uma rede convolucional, as RNNs aplicadas às imagens geralmente são computacionalmente mais caros, mas permitem interações laterais de longo alcance entre características no mesmo mapeamento.

9.2 Gradiente

Apesar do algoritmo back-propagation ser eficiente no cálculo do gradiente durante o treinamento de uma rede recorrente, existe uma dificuldade para dependências de longo prazo. O problema básico é que os gradientes propagados por vários estágios tendem a desaparecer (na

maioria das vezes) ou a explodir (raramente, mas com muitos danos à otimização). Mesmo que se assuma que os parâmetros são tais que a rede recorrente é estável (pode armazenar memórias, com gradientes não explodindo), a dificuldade com dependências de longo prazo decorre dos pesos exponencialmente menores para as interações de longo prazo em comparação com as de curto prazo.

As redes recorrentes envolvem a composição da mesma função várias vezes, uma vez por etapa do tempo. Essas composições podem resultar em comportamento extremamente não linear. Esse problema é específico para redes recorrentes. No caso escalar, imagine multiplicar um peso w muitas vezes. O produto w^t desaparecerá ou explodirá dependendo da magnitude de w. Na prática, a medida que aumentamos o alcance das dependências que precisam ser capturadas, a otimização baseada em gradiente se torna cada vez mais difícil, com a probabilidade de um treinamento bem-sucedido de uma RNN tradicional via SGD rapidamente atingindo 0 para sequências com comprimentos de apenas 10 ou 20.

9.3 LSTM e RNNs fechadas

Os modelos de sequência mais eficazes usados em aplicações práticas são chamados RNNs fechadas. Isso inclui a longa memória de curto prazo (LSTM) e as redes baseadas em unidades recorrentes fechadas. As RNNs fechadas são baseadas na ideia de criar caminhos ao longo do tempo que permitem calcular derivadas que não desaparecem nem explodem. Eles generalizam parâmetros que podem ter seu valor alterado a cada etapa do tempo.

A ideia inteligente de introduzir auto loops para produzir caminhos onde o gradiente possa fluir por longos períodos de tempo é uma contribuição fundamental do modelo LSTM. Uma adição crucial tem sido condicionar o peso desse auto loop ao contexto, em vez de ficar fixo. Ao aumentar o peso desse auto loop (controlado por outra unidade oculta), a escala de tempo da integração pode ser alterada dinamicamente. Nesse caso, quer dizer que, mesmo para um LSTM com parâmetros fixos, a escala de tempo da integração pode mudar com base na sequência de entrada, porque as constantes de tempo são produzidas pelo próprio modelo.

9.4 Exemplo prático

Para exemplificar uma rede recorrente, foi desenvolvido um código para resolução de um novo problema. O problema proposto é um problema de classificação sentimental, e o conjunto de dados foi o conjunto de avaliações de filmes no IMDB. Foi necessário um tratamento dos elementos de entrada para que cada um deles fosse limitado a 20 mil palavras. Além disso, cada elemento foi dividido em sequências com tamanho fixo de 80 palavras, adicionando valor 0 para sequências menores que o tamanho fixado. O código para importação e tratamento do conjunto de dados é mostrado a seguir.

```
max_features = 20000
dataset = tf.keras.datasets.imdb.load_data(num_words=max_features)

(x_input, y_input), (x_test, y_test) = dataset
x_input = tf.keras.preprocessing.sequence.pad_sequences(x_input,
                                                        maxlen=80)
x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test,
                                                       maxlen=80)
```

Por fim, foi definida a arquitetura da rede com uma camada LSTM com 128 neurônios, com dropout com probabilidade de 80% de inclusão, tanto na etapa *feedfoward* como na etapa recorrente, e gerando um saída para cada etapa de tempo. A camada LSTM é conectada diretamente a camada de saída, com uma função de ativação Sigmoid, pois o problema de classificação sentimental possui uma saída probabilística, neste caso.

Foi adicionada uma camada *Embedding* à entrada da camada LSTM, que é uma camada implementada pelo Keras com um algoritmo de transformação do vetor de entrada em um vetor denso, fazendo com que o número de características determinado, no caso 128, flua para a camada LSTM, e não as 80 palavras em si. Porém, essa camada pode ser substituída por outros algoritmos que realizem a mesma função, podendo inclusive ser algoritmos pré-treinados, onde a performance e adaptação ao problema específico é trocado por um menor tempo de execução e custo computa-

cional.

A função de custo utilizada foi a entropia cruzada binária, pois o problema é uma classificação em apenas duas classes (avaliação positiva e negativa), e o algoritmo de otimização usado foi o RMSProp, com o valor inicial da taxa de aprendizado de 0.001. O tamanho do lote foi de 128 elementos e o conjunto de validação foi o próprio conjunto de teste. O código para implementação da rede e dos algoritmos é mostrado a seguir. O código inteiro pode ser visualizado no anexo 4.

```
def net(x_train, y_train, x_test, y_test, epoch=5, max_features=20000):

    x_input = tf.keras.layers.Input(shape=(80))

    l1 = tf.keras.layers.Embedding(max_features, 128)(x_input)

    l2 = tf.keras.layers.LSTM(128,
                              dropout=0.2,
                              recurrent_dropout=0.2,
                              return_sequences=False)(l1)

    logit = tf.keras.layers.Dense(1, activation='sigmoid')(l2)

    model = tf.keras.Model(inputs=x_input, outputs=logit)

    model.compile(loss='binary_crossentropy',
              optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              metrics=['accuracy'])

    fit = model.fit(x_train, y_train,
                    batch_size=128,
                    epochs=epoch,
                    validation_data=(x_test, y_test))
```

```
    print(model.summary())

    history = fit.history

    test_scores = model.evaluate(x_test, y_test, verbose=2)


    return history, test_scores
```

Apesar da maior complexidade do problema comparado ao problema solucionado pelas redes anteriores, foi possível obter uma performance relativamente boa com a rede implementada, com um custo de 0.4986 e 82.50% de precisão no conjunto de teste. A rede foi executada durante 10 épocas, cada uma com uma duração de treinamento em torno de 45 segundos, com um número de parâmetros treináveis de 2.691.713, sendo 2.560.000 parâmetros da camada de *Embedding*, responsável pela transformação do vetor de entrada, que, por opção de projeto, pode ser substituído por um algoritmo pré-treinado.

A rede recorrente implementada, de forma semelhante à rede convolucional, não passou por tentativas de melhoras na sua otimização e regularização, devido ao objetivo de exemplificação, a uma boa performance já obtida, e um custo computacional muito alto para que sejam feitas várias tentativas.

10 Conclusão

Com a explanação da teoria e conceitos sobre as principais arquiteturas, funções de ativação, funções de custo e métodos de regularização e otimização, com uma abordagem um pouco mais prática, com uso de uma metodologia e exemplificações para clarear o aprendizado, este trabalho espera ter alcançado o objetivo de poder situar e facilitar a vida de novos entusiastas, pesquisadores e engenheiros de sistemas de aprendizado profundo.

A implementação das redes durante a realização deste trabalho, desde a concepção da arquitetura e definição dos métodos, até a análise dos resultados, possibilitou a visualização da importância das escolhas que devem ser feitas durante o desenvolvimento do projeto do sistema. É importante levar em consideração durante o desenvolvimento de um sistema de aprendizado profundo, os requisitos do projeto almejados durante sua concepção, e a precisão mínima necessária do sistema para atendê-los, para que o sistema tenha a performance necessária com o menor custo possível.

Além disso, os resultados e conclusões obtidos neste trabalho podem ser usados como ponto inicial para que novos parâmetros e o comportamentos de novas arquiteturas, para outras finalidades, sejam estudados e testados. Algumas sugestões de arquiteturas de redes recentes e menos exploradas, mas que prometem grandes sucessos, são:

- **Autoencoders**: Um autoencoder é um tipo de rede neural artificial usada para aprender codificações de dados eficientes de maneira não supervisionada. Tem o objetivo de aprender uma representação de um conjunto de dados, tipicamente para reduções dimensionais.

- **GAN**: Uma rede contraditória generativa, ou generative adversarial network, é uma classe de sistemas de aprendizado de máquina inventados por Ian Goodfellow e seus colegas em 2014. Dado um conjunto de treinamento, essa técnica aprende a gerar novos dados com as mesmas estatísticas que o conjunto de treinamento. Por exemplo, um GAN treinado em fotografias pode gerar novas fotografias que parecem, pelo menos superficialmente, autênticas para os observadores humanos, tendo muitas características realistas.

- **CapsNet**: A Capsule Neural Network adiciona estruturas chamadas ”cápsulas”a uma rede neural convolucional (CNN) e reutiliza a saída de várias dessas cápsulas para formar representações mais estáveis (com relação a várias perturbações) para cápsulas mais altas. Um mecanismo de roteamento dinâmico para redes de cápsulas foi introduzido por Geoffrey Hinton e sua equipe em 2017. A abordagem afirmou reduzir as taxas de erro no MNIST e reduzir o tamanho dos conjuntos de treinamento. Os resultados foram consideravelmente melhores do que uma CNN em dígitos altamente sobrepostos.

Apesar deste trabalho poder ser usado futuramente nas mais diversas áreas, sugere-se a sua aplicação em trabalhos futuros nas áreas de controle e automação, utilizando modelos salvos em nuvem. Controle e automação são áreas que podem se beneficiar fortemente das vantagens trazidas por sistemas de aprendizado profundo, mas, apesar disso, não possuem muitas pesquisas e implementações já realizadas, se tornando áreas que devem ser melhor atentadas ao se pensar em projetos e pesquisas futuras.

Finalmente, com a construção de uma base teórica e prática sólida suficiente para construir sistemas escaláveis e aplicáveis em situações que resolvam problemas reais, além dos acadêmicos, este trabalho espera ter incentivado a realização de novos projetos e pesquisas futuras que validem todos os resultados, dados e conhecimentos obtidos neste e nos futuros trabalhos.

11 Referências

[1] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[2] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. (2015). *Deep learning*. Nature 521, no. 7553

[3] Schmidhuber, Jürgen. (2015). *Deep Learning in Neural Networks: An Overview*. Neural Networks, no. 61

[4] Matthew D Zeiler and Rob Fergus. (2013). *Visualizing and Understanding Convolutional Networks*.

[5] James Bergstra, Rémy Bardenet, Yoshua Bengio and Balázs Kég. (2011). *Algorithms for Hyper-Parameter Optimization*. NIPS'2011

[6] Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. (2010). *Understanding the difficulty of training deep feedforward neural networks*. AISTATS'10

[7] Bengio, Yoshua. (2009). *Learning deep architectures for* $\{AI\}$. Foundations and Trends in Machine Learning, vol. 2

12 Anexo 1

```
import numpy as np
import pandas as pd
import tensorflow as tf
import os # para criar pastas
from matplotlib import pyplot as plt # para mostrar imagens


def net(x_train, y_train, x_test, y_test, epoch=5, activation='relu'):

    # Monta uma rede neural simples, com duas camadas e 512 neurônios cada
    x_input = tf.keras.layers.Input(shape=(784,))
    l1 = tf.keras.layers.Dense(512, activation=activation)(x_input)
    l2 = tf.keras.layers.Dense(512, activation=activation)(l1)
    logit = tf.keras.layers.Dense(10, activation='softmax')(l2)

    model = tf.keras.Model(inputs=x_input, outputs=logit)

    model.compile(loss='sparse_categorical_crossentropy',
              optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              metrics=['accuracy'])


    fit = model.fit(x_train, y_train,
                    batch_size=64,
                    epochs=epoch,
                    validation_split=0.2)

    history = fit.history

    test_scores = model.evaluate(x_test, y_test, verbose=2)

    return history, test_scores

```

```
34
35  if __name__ == '__main__':
36
37      (x_input, y_input), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
38      x_input = x_input.reshape(60000, 784).astype('float32') / 255
39      x_test = x_test.reshape(10000, 784).astype('float32') / 255
40
41      epoch = 50
42
43      # cria uma rede neural para cada função de ativação
44      sig_train_score, sig_test_score = net(x_input, y_input,
45                                            x_test, y_test,
46                                            epoch, activation='sigmoid')
47      tanh_train_score, tanh_test_score = net(x_input, y_input,
48                                              x_test, y_test,
49                                              epoch, activation='tanh')
50      relu_train_score, relu_test_score = net(x_input, y_input,
51                                              x_test, y_test,
52                                              epoch, activation='relu')
53      softmax_train_score, softmax_test_score = net(x_input, y_input,
54                                                    x_test, y_test,
55                                                    epoch, activation='softmax')
56      elu_train_score, elu_test_score = net(x_input, y_input,
57                                            x_test, y_test,
58                                            epoch, activation='elu')
59
60      losses = [sig_train_score['loss'],
61                tanh_train_score['loss'],
62                relu_train_score['loss'],
63                softmax_train_score['loss'],
64                elu_train_score['loss']]
65
66      acc = 'accuracy'
67      accuracies = [sig_train_score[acc],
68                    tanh_train_score[acc],
69                    relu_train_score[acc],
```

```
                softmax_train_score[acc],
                elu_train_score[acc]]

val_losses = [sig_train_score['val_loss'],
                tanh_train_score['val_loss'],
                relu_train_score['val_loss'],
                softmax_train_score['val_loss'],
                elu_train_score['val_loss']]

acc = 'val_accuracy'
val_accuracies = [sig_train_score[acc],
                  tanh_train_score[acc],
                  relu_train_score[acc],
                  softmax_train_score[acc],
                  elu_train_score[acc]]

test_score = {'Sigmoid': sig_test_score,
                'Tanh': tanh_test_score,
                'ReLU': relu_test_score,
                'Softmax': softmax_test_score,
                'ELU': elu_test_score}

print(test_score)

plt.style.use('ggplot')

df = pd.DataFrame(np.array(accuracies) * 100)
df = df.T

df2 = pd.DataFrame(np.array(losses))
df2 = df2.T

df3 = pd.DataFrame([np.array(losses[0]), np.array(val_losses[0])])
df3 = df3.T

df4 = pd.DataFrame([np.array(losses[1]), np.array(val_losses[1])])
```

```
106     df4 = df4.T
107
108     df5 = pd.DataFrame([np.array(losses[2]), np.array(val_losses[2])])
109     df5 = df5.T
110
111     df6 = pd.DataFrame([np.array(losses[3]), np.array(val_losses[3])])
112     df6 = df6.T
113
114     df7 = pd.DataFrame([np.array(losses[4]), np.array(val_losses[4])])
115     df7 = df7.T
116
117     fig1, (ax1, ax2) = plt.subplots(1, 2)
118     ax1.plot(df)
119     ax1.set_title('Precisão')
120     ax1.legend(['Sigmoid', 'Tanh', 'ReLU', 'Softmax', 'ELU'])
121     ax1.set(xlabel='Época', ylabel='Precisão (%)')
122
123     ax2.plot(df2)
124     ax2.set_title('Custo')
125     ax2.legend(['Sigmoid', 'Tanh', 'ReLU', 'Softmax', 'ELU'])
126     ax2.set(xlabel='Época', ylabel='Erro')
127
128     fig2, ((ax3, ax4), (ax5, ax6), (ax7,ax8)) = plt.subplots(3, 2)
129     fig2.suptitle('Regularização')
130     ax3.plot(df3)
131     ax3.set_title('Sigmoid')
132     ax3.legend(['Treinamento', 'Validação'])
133
134     ax4.plot(df4)
135     ax4.set_title('Tanh')
136     ax4.legend(['Treinamento', 'Validação'])
137
138     ax5.plot(df5)
139     ax5.set_title('ReLU')
140     ax5.legend(['Treinamento', 'Validação'])
141     ax5.set(ylabel='Erro')
```

```
142
143     ax6.plot(df6)
144     ax6.set_title('Softmax')
145     ax6.legend(['Treinamento', 'Validação'])
146     ax6.set(xlabel='Época')
147
148     ax7.plot(df7)
149     ax7.set_title('ELU')
150     ax7.legend(['Treinamento', 'Validação'])
151     ax7.set(xlabel='Época')
152
153     fig2.delaxes(ax8)
154
155     plt.show()
```

13 Anexo 2

```
import numpy as np
import pandas as pd
import tensorflow as tf
import os # para criar pastas
from matplotlib import pyplot as plt # para mostrar imagens


def net(x_train, y_train, x_test, y_test, epoch=5, activation='relu'):

    # Monta uma rede neural simples, com duas camadas e 512 neurônios cada
    x_input = tf.keras.layers.Input(shape=(784,))
    dropout = tf.keras.layers.Dropout(0.2)(x_input)
    l1 = tf.keras.layers.Dense(512, activation=activation)(dropout)
    dropout2 =tf.keras.layers.Dropout(0.5)(l1)
    l2 = tf.keras.layers.Dense(512, activation=activation)(dropout2)
    logit = tf.keras.layers.Dense(10, activation='softmax')(l2)

    model = tf.keras.Model(inputs=x_input, outputs=logit)

    model.compile(loss='sparse_categorical_crossentropy',
              optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              metrics=['sparse_categorical_accuracy'])

    fit = model.fit(x_train, y_train,
                    batch_size=64,
                    epochs=epoch,
                    validation_split=0.2)

    print(model.summary())

    history = fit.history

    test_scores = model.evaluate(x_test, y_test, verbose=2)
```

```
34
35      return history, test_scores
36
37
38  if __name__ == '__main__':
39
40      (x_input, y_input), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
41      x_input = x_input.reshape(60000, 784).astype('float32') / 255
42      x_test = x_test.reshape(10000, 784).astype('float32') / 255
43
44      epoch = 20
45
46      # cria uma rede neural para cada função de ativação
47      train_score, test_score = net(x_input, y_input,
48                                    x_test, y_test,
49                                    epoch, activation='tanh')
50
51      loss = train_score['loss']
52      accuracy = train_score['sparse_categorical_accuracy']
53
54      val_loss = train_score['val_loss']
55      val_accuracy = train_score['val_sparse_categorical_accuracy']
56
57      test_score = test_score
58      print(test_score)
59
60      plt.style.use('ggplot')
61
62      df = pd.DataFrame([np.array(accuracy) * 100])
63      df = df.T
64
65      df2 = pd.DataFrame([np.array(loss)])
66      df2 = df2.T
67
68      df3 = pd.DataFrame([np.array(loss), np.array(val_loss)])
69      df3 = df3.T
```

```
70
71      fig1, (ax1, ax2) = plt.subplots(1, 2)
72      ax1.plot(df)
73      ax1.set_title('Precisão')
74      ax1.legend(['Tanh'])
75      ax1.set(xlabel='Época', ylabel='Precisão (%)')
76
77      ax2.plot(df2)
78      ax2.set_title('Custo')
79      ax2.legend(['Tanh'])
80      ax2.set(xlabel='Época', ylabel='Erro')
81
82      fig2, ax3 = plt.subplots(1, 1)
83      fig2.suptitle('Regularização')
84      ax3.plot(df3)
85      ax3.set_title('Tanh')
86      ax3.legend(['Treinamento', 'Validação'])
87
88      plt.show()
```

14 Anexo 3

```
import numpy as np
import pandas as pd
import tensorflow as tf
import os # para criar pastas
from matplotlib import pyplot as plt # para mostrar imagens


def net(x_train, y_train, x_test, y_test, epoch=5, activation='relu'):

    x_input = tf.keras.layers.Input(shape=(28, 28, 1))

    l1 = tf.keras.layers.Conv2D(24, kernel_size=(3, 3), activation='relu')(x_input)
    l1 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(l1)


    l2 = tf.keras.layers.Conv2D(36, (3, 3), activation='relu')(l1)
    l2 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(l2)

    l3 = tf.keras.layers.Flatten()(l2)
    l4 = tf.keras.layers.Dense(128, activation=activation)(l3)
    logit = tf.keras.layers.Dense(10, activation='softmax')(l4)

    model = tf.keras.Model(inputs=x_input, outputs=logit)

    model.compile(loss='sparse_categorical_crossentropy',
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              metrics=['sparse_categorical_accuracy'])

    fit = model.fit(x_train, y_train,
                    batch_size=128,
                    epochs=epoch,
                    validation_split=0.2)

```

```
34      print(model.summary())
35
36      history = fit.history
37
38      test_scores = model.evaluate(x_test, y_test, verbose=2)
39
40      return history, test_scores
41
42
43  if __name__ == '__main__':
44
45      (x_input, y_input), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
46      x_input = x_input.reshape(x_input.shape[0], 28, 28, 1).astype('float32') / 255
47      x_test = x_test.reshape(x_test.shape[0], 28, 28, 1).astype('float32') / 255
48
49      epoch = 12
50
51      train_score, test_score = net(x_input, y_input, x_test, y_test, epoch)
52
53      loss = train_score['loss']
54      accuracy = train_score['sparse_categorical_accuracy']
55
56      val_loss = train_score['val_loss']
57      val_accuracy = train_score['val_sparse_categorical_accuracy']
58
59      test_score = test_score
60      print(test_score)
61
62      plt.style.use('ggplot')
63
64      df = pd.DataFrame([np.array(accuracy) * 100])
65      df = df.T
66
67      df2 = pd.DataFrame([np.array(loss)])
68      df2 = df2.T
69
```

```
70      df3 = pd.DataFrame([np.array(loss), np.array(val_loss)])
71      df3 = df3.T
72
73      fig1, (ax1, ax2) = plt.subplots(1, 2)
74      ax1.plot(df)
75      ax1.set_title('Precisão')
76      ax1.legend(['CNN'])
77      ax1.set(xlabel='Época', ylabel='Precisão (%)')
78
79      ax2.plot(df2)
80      ax2.set_title('Custo')
81      ax2.legend(['CNN'])
82      ax2.set(xlabel='Época', ylabel='Erro')
83
84      fig2, ax3 = plt.subplots(1, 1)
85      fig2.suptitle('Regularização')
86      ax3.plot(df3)
87      ax3.set_title('CNN')
88      ax3.legend(['Treinamento', 'Validação'])
89
90      plt.show()
```

15 Anexo 4

```
import numpy as np
import pandas as pd
import tensorflow as tf
import os # para criar pastas
from matplotlib import pyplot as plt # para mostrar imagens


def net(x_train, y_train, x_test, y_test, epoch=5, max_features=20000):

    x_input = tf.keras.layers.Input(shape=(80))

    l1 = tf.keras.layers.Embedding(max_features, 128)(x_input)

    l2 = tf.keras.layers.LSTM(128,
                              dropout=0.2,
                              recurrent_dropout=0.2,
                              return_sequences=False)(l1)

    logit = tf.keras.layers.Dense(1, activation='sigmoid')(l2)

    model = tf.keras.Model(inputs=x_input, outputs=logit)

    model.compile(loss='binary_crossentropy',
              optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              metrics=['accuracy'])

    fit = model.fit(x_train, y_train,
                    batch_size=128,
                    epochs=epoch,
                    validation_data=(x_test, y_test))

    print(model.summary())

```

```
34      history = fit.history
35
36      test_scores = model.evaluate(x_test, y_test, verbose=2)
37
38      return history, test_scores
39
40
41  if __name__ == '__main__':
42
43      max_feats = 20000
44      dataset = tf.keras.datasets.imdb.load_data(num_words=max_feats)
45
46      (x_input, y_input), (x_test, y_test) = dataset
47      x_input = tf.keras.preprocessing.sequence.pad_sequences(x_input, maxlen=80)
48      x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test, maxlen=80)
49
50      epoch = 10
51
52      train_score, test_score = net(x_input, y_input, x_test, y_test, epoch, max_feats)
53
54      loss = train_score['loss']
55      accuracy = train_score['accuracy']
56
57      val_loss = train_score['val_loss']
58      val_accuracy = train_score['accuracy']
59
60      test_score = test_score
61      print(test_score)
62
63      plt.style.use('ggplot')
64
65      df = pd.DataFrame([np.array(accuracy) * 100])
66      df = df.T
67
68      df2 = pd.DataFrame([np.array(loss)])
69      df2 = df2.T
```

```
70
71      df3 = pd.DataFrame([np.array(loss), np.array(val_loss)])
72      df3 = df3.T
73
74      fig1, (ax1, ax2) = plt.subplots(1, 2)
75      ax1.plot(df)
76      ax1.set_title('Precisão')
77      ax1.legend(['RNN'])
78      ax1.set(xlabel='Época', ylabel='Precisão (%)')
79
80      ax2.plot(df2)
81      ax2.set_title('Custo')
82      ax2.legend(['RNN'])
83      ax2.set(xlabel='Época', ylabel='Erro')
84
85      fig2, ax3 = plt.subplots(1, 1)
86      fig2.suptitle('Regularização')
87      ax3.plot(df3)
88      ax3.set_title('RNN')
89      ax3.legend(['Treinamento', 'Validação'])
90
91      plt.show()
```