Chapter
6

Deep Learning - Teoria e Prática

Cristina Nader Vasconcelos, Esteban Walter Gonzalez Clua

Abstract

Deep Learning is bringing a major IT revolution in recent years, opening up new horizons and possibilities for many different areas and applications. Thanks to the investment of large companies such as NVIDIA, Google, Microsoft, IBM, among others, many tools and frameworks are becoming accessible for the friendly application of Deep Learning in solving many different problems. This short course will present basic concepts of neural networks and deep neural networks, an overview of some libraries and tools and a brief introduction to GPU architectures and how they support the area.

Resumo

Redes Neurais Profundas vêm provocando uma grande revolução na indústria de TI nos últimos anos, abrindo diversos horizontes e possibilidades para as mais variadas áreas e aplicações. Graças ao investimento de grandes empresas, como NVIDIA, Google, Microsoft, IBM, dentre outras, inúmeras ferramentas e plataformas vêm se tornando acessíveis para aplicar Deep Learning em soluções de diversos problemas. Neste mini-curso serão apresentados conceitos básicos de redes neurais profundas, algumas bibliotecas e ferramentas, arquiteturas de GPUs e como as mesmas são capazes de viabilizar a área.

6.1. Introdução

A Inteligência Artificial (IA) tem sido há longo tempo almejada e buscada pelos humanos. O HAL (o computador de bordo da *Discovery One*, do filme "2001 - Uma Odisséia no Espaço"), a Skynet (sistema de IA do filme Exterminador do Futuro) ou a Matrix (do filme homônimo) nos fascinam e ao mesmo tempo nos assustam com uma visão do que pode ser um futuro controlado pelas máquinas. Embora pesquisas na área de IA venham sendo desenvolvidas há décadas, nos últimos anos experimentamos uma verdadeira revolução, levada adiante por grandes empresas e com

investimentos bilionários: NVIDIA, Google, IBM, Microsoft, Amazon, Facebook, Baidu, entre diversas outras. Em alguns casos, o tema de IA não se trata apenas de uma das muitas áreas de atuação destas empresas, mas vem se tornando a mais importante em seu direcionamento.

Apesar de "tarefas inteligentes" exigirem diversos graus de atuação e de não nos aprofundarmos na comparação com a inteligência humana neste texto, estamos aqui interessados em mecanismos que possibilitem a resolução de tarefas pelo aprendizado dos padrões que compõem o problema a ser resolvido: se queremos preparar uma receita, precisamos reconhecer os ingredientes, as suas quantidades e inclusive onde se encontram na dispensa; se queremos atravessar uma rua, temos que reconhecer a sinalização do local, os carros que estão nas vias e o local correto para a travessia. Podemos observar que em diferentes tarefas faz-se necessário como estágio fundamental o reconhecimento de padrões que regem a tomada de decisão entre a observação do contexto em que queremos resolvê-la (fornecido por um sinal de entrada) até a produção da saída desejada.

Em alto nível, podemos dizer que a revolução da IA que estamos vivendo se dá pelo desenvolvimento de sistemas com capacidade de aprender padrões intrínsecos que regem uma determinada tarefa pela observação de grandes volumes de dados e da sua capacidade de generalização dos modelos aprendidos nesses sistemas para inferir a resposta esperada em novos casos.

Através da análise de padrões, os sistemas baseados em técnicas de *Deep Learning* são capazes de reconhecer, traduzir, sintetizar e até prever sinais das mais diferentes naturezas. Têm sido usadas na analise de sinais visuais como imagens e vídeos, sinais de áudio, de linguagem natural, entre outros tipos de sinais e além da combinação de sinais de natureza distintas.

Inúmeras são as aplicações nas quais as técnicas de *Deep Learning* já são utilizadas com sucesso, obtendo resultados impensáveis de serem obtidos em curto prazo até recentemente. A listagem a seguir inclui apenas algumas delas, como forma de inspiração ao leitor. Técnicas de *Deep Learning* têm sido usadas para:

- interpretação de caracteres em diferentes alfabetos e tipografias a partir de imagens [LeCun et al. 1989, Ciresan et al. 2011a, Ciresan et al. 2012];
- reconhecimento de fala [Chan et al. 2016], verificação e identificação do orador [Heigold et al. 2016];
- tradução entre diferentes línguas [Britz et al. 2017] e geração de linguagem natural modelando explicitamente propriedades como estilo e assunto [Bowman et al. 2016];
- produzir respostas a e-mails [Kannan et al. 2016] e ainda construir agentes capazes de interagir em linguagem natural [Shao et al. 2017];
- detecção de pedestres em imagens [Sermanet et al. 2013], classificação de poses das mãos [Tompson et al. 2014], e ainda estimar pose do corpo em imagens com múltiplas pessoas [Papandreou et al. 2017];

- segmentação e rotulação em tempo real de cenas pela análise de imagens de profundidade [Couprie et al. 2013] e também análises especializadas em diversos outros tipos de objetos como reconhecimento de sinais de trânsito [Ciresan et al. 2011b] e de números de imóveis [Sermanet et al. 2012] além de poderem identificar múltiplos objetos, contá-los e localizá-los [Eslami et al. 2016];

- controlar veículos autônomos e robôs seja sua navegação [Gupta et al. 2017, Giusti et al. 2016] ou seus mecanismos de interação com o ambiente [Levine et al. 2016];

- detecção de mitose em imagens histológicas de câncer de mama [Ciresan et al. 2013], detecção de retinopatia diabética em fotografias de fundo de retina [Gulshan et al. 2016], classificação de lesões de pele [Esteva et al. 2017], entre diversas outras aplicações biomédicas;

- produzir imagens com super resolução [Dahl et al. 2017] ou ainda sintetizar imagens foto-realistas [Odena et al. 2016];

- simular em tempo real fluidos e fumaça [Tompson et al. 2016] bem como criar uma reconstrução em 3D a partir de uma visualização 2D de objetos [Yan et al. 2016];

- prever a continuidade em sequências de vídeo [Villegas et al. 2017];

- aprender e aplicar estilos artísticos seja na criação de imagens [Dumoulin et al. 2017] ou na geração de música [Jaques et al. 2016];

- jogar videogames [Mnih et al. 2013] e jogos de tabuleiro [Silver et al. 2016];

- encontrar provas de teoremas de primeira ordem [Loos et al. 2017];

- produzir computadores capazes de aprender sua própria programação, como uma Máquina de Turing Neural [Kaiser and Sutskever 2016].

As redes neurais são as protagonistas nesta revolução produzida pela aprendizagem de máquina. A modelagem tradicional de sistemas com aprendizagem de máquina consiste em duas fases. Primeiramente são extraídos a partir dos dados brutos (uma imagem, por exemplo) que compõe o sinal de entrada um conjunto de medições que caracterizem as propriedades que se deseja avaliar do problema (chamada etapa de extração de características). A partir das medições uma amostra que se deseja avaliar passa a ser representada tipicamente por um vetor descritor que representa a assinatura daquela amostra segundo as medições estabelecidas. Na sequência é aplicado um algoritmo de aprendizagem de máquina para a tomada de decisão sobre o conjunto de medições levantado.

O grande desafio nesta abordagem clássica é que a etapa de extração de características não é trivial, já que nela estão subentendidos conhecimentos especialistas. O que devemos observar na análise de uma imagem médica para fazer um diagnóstico? O que devemos observar de um áudio para identificar o orador? O que devemos observar de um texto para verificar a sua autoria?

Além disso, mesmo em casos em que são conhecidos os critérios especialistas adotados em uma determinada análise, encontrar uma formalização matemática que suporte as medições desejadas sobre o elemento em questão com todas as suas possíveis variações válidas também não é trivial. Em grande parte dos problemas de análise de sinais biológicos é comum que existam uma série de variações válidas e frequentes que o tornam ainda mais difíceis de serem modelados. Supondo que conseguimos modelar a forma 3D de um determinado cachorro para responder a pergunta: essa foto contém um cachorro? E se alterarmos a iluminação, continua sendo um cachorro? E o cenário, pose, escala com que o cachorro aparece na imagem, influenciam nessa decisão? Como deve ainda ser considerado um cachorro, tais invariâncias devem ser incorporadas ao modelo matemático.

Esse problema se estende pra outros tipos de sinais e tarefas. Ao analisar áudio poderíamos indagar sobre sotaque, intonação, velocidade da fala, ruído ambiente, taxa de captura, entre outras variações. De acordo com o sinal observado e o problema que se deseja resolver existem outras tantas variações a serem incorporadas ao modelo matemático de suporte a sistemas para tratamento de sinais biológicos, podendo até por vezes não serem explicitamente conhecidas dada a complexidade da tarefa ou do sinal.

Nas abordagens conhecidas como *Deep Learning* para solução dos mais diferenciados problemas, a etapa de descrição formal do modelo pela formulação matemática e implementação do conhecimento especialista para extração de características é substituída por um processo de treinamento de redes neurais de maneira integrada a etapa de tomada de decisão. O treinamento permite observar as correlações existentes em um grande volume de dados para aprender diretamente do dado bruto quais padrões são relevantes a tarefa e assim ajustar os parâmetros da rede para que uma vez treinada passe a fornecer um mapeamento entre um novo dado de entrada e sua saída no problema modelado.

Esta possibilidade de solução integrada vem destacando um conjunto de técnicas que atendem ao nome *Deep Learning* (DL) como uma das abordagens mais populares e bem sucedidas para tarefas envolvendo aprendizado de padrões. Existem 3 fatores que se alinharam para que o DL se tornasse viável atualmente: (1) uma grande quantidade de dados disponíveis (Big Data); (2) Desenvolvimento de novas técnicas de DL; (3) Supercomputação de forma acessível. Neste último quesito é onde as GPUS se tornam importantes protagonistas nesta revolução: além de serem processadores altamente paralelos (arquiteturas como a NVIDIA Pascal podem chegar a 11 TFlops de processamento numa única placa, mais do que o dobro do supercomputador Laurence, maior computador do mundo no ano 2000), o "DNA" de suas arquiteturas permitem que diversas tarefas típicas de DL sejam beneficiadas de forma direta pelo alto paralelismo disponível. Para se ter uma ideia, em 2012 a Google apresentou um sistema de DL para análise de imagens no qual usavam 1000 servidores de CPUs, com um custo de 5 milhões de dólares e consumo de energia de 600KW. No ano seguinte, resolveram o mesmo problema usando 3 servidores, com 12 GPUs, a um custo de 33 mil dólares e com uma energia de 4KW.

Sobre o primeiro dos 3 fatores listados, também é notável que as comunidades

de pesquisa em reconhecimento de padrões nos diferentes tipos de sinais (áudio, texto e imagem) tenham se organizado para promover desafios com grandes bancos de dados anotados. A anotação desses bancos consiste em criar pares em que cada exemplo de entrada é associado a resposta esperada, muitas vezes fornecida por um especialista. A partir desses bancos é possível padronizar a avaliação de diferentes algoritmos de maneira a convergir esforços de pesquisa em uma plataforma de comparação em comum. É o caso da Imagenet Large Scale Visual Recognition Challenge [ima], que em 2012 destacou a capacidade das Redes Neurais de Convolução frente as abordagens de modelagem de conhecimento especialista na tarefa de classificação de um conjunto de 1000 categorias de objetos fornecendo para treinamento uma base de imagens com 1.2 milhão de imagens anotadas.

Outra frente importante tem sido as bibliotecas disponibilizadas para a comunidade. Sejam as desenvolvidas por Universidades ou por empresas, viabilizaram acelerar a pesquisa na área e impulsionaram o crescente número de aplicações desenvolvidas.

Este capítulo de livro, embora de maneira resumida, busca cobrir os temas mais abordados nas soluções vigentes com o uso de *Deep Learning*. Não tem a pretensão de cobrir toda a área, mas de servir como convite para um primeiro contato do leitor e de instigar sua busca pelas fronteiras do DL. São apresentados os fundamentos necessários para compreensão do que constitui as chamadas Redes Neurais na Seção 6.2, aprofundando duas de suas possíveis modelagens ditas Redes Neurais Alimentadas para Frente (na Seção 6.2.2.1) e Redes Neurais Recorrentes (na Seção 6.2.2.3). Em seguida, a Seção 6.3 apresenta as chamadas Redes Neurais de Convolução como ilustração do primeiro modelo, enquanto que a Seção 6.4 ilustra o segundo modelo apresentando as chamadas redes *Long-Short Term Mermory*. Buscando cobrir os bastidores de sua implementação eficiente, a Seção 6.5 apresenta a arquitetura das GPUs. Por fim, uma breve apresentação sobre as ferramentas de desenvolvimento é apresentada na Seção 6.6.

6.2. Fundamentos

As diferentes soluções baseadas em aprendizado profundo são construídas em cima de variações das chamadas redes neurais artificiais. Seus fundamentos são apresentados nesta seção sem a pretensão de cobrir os múltiplos aspectos teóricos da área.

Diferentemente de algoritmos programados para traduzir um modelo de co/-nhe/-ci/-men/-to especialista na solução de um problema, redes neurais simulam o cérebro humano no sentido de que aprendem a realizar uma tarefa.

Em um contexto mais amplo do que as redes neurais artificiais, a área de aprendizado de máquina classifica os algoritmos de aprendizado em três categorias:

- **Aprendizado supervisionado**: aplicável quando são conhecidos pares associando entradas às respectivas respostas desejadas. Algoritmos nesta categoria buscam aprender uma função que mapeia as entradas nas saídas desejadas. Uma vez aprendida, tal função de mapeamento pode ser aplicada a novas entradas. O processo de treinamento continua até que o modelo atinja um nível

de precisão desejado nos dados de teste ou que uma quantidade de iterações seja atingida.

- **Aprendizado não-supervisionado:** aplicável quando não são fornecidas saídas desejadas, mas se deseja encontrar ou fazer inferências sobre padrões inerentes ao comportamento das amostras de entrada. Os algoritmos mais conhecidos nesta categoria são os de clusterização, responsáveis pela organização das amostras de entrada em um conjunto de grupos, ditos *clusters*.

- **Aprendizado por reforço:** aplicável quando se deseja um algoritmo que interage com um ambiente para tomar decisões. Nesta categoria o aprendizado se dá expondo o algoritmo a um ambiente no qual ele pode praticar por tentativa e erro, recebendo recompensa ou punições como *feedback* pelas decisões tomadas. Com o passar do tempo, passa a acumular conhecimento com a experiência passada, ou seja, aprende a preferir o tipo certo de ação e evitar as que induzem ao erro, tentando capturar o melhor conhecimento/estratégia para tomar decisões futuras.

Existem abordagens de redes neurais profundas para tratamento de problemas nas três categorias citadas. Sua construção parte de elementos simples, mas que em grande quantidade e propriamente dispostos e adaptados interagem entre si modelando diferentes comportamentos.

6.2.1. O Perceptron

Diferentes topologias de Redes Neurais Artificiais (RNAs) podem ser construídas utilizando-se variações e combinações da mesma peça básica: um neurônio artificial.

Em alto nível, podemos pensar que um neurônio artificial é um mecanismo capaz de processar um conjunto de sinais que recebe como entrada na forma do vetor $X = \{x_1, x_2, ..., x_N\}$ para produzir um sinal de saída (y).

Assim como neurônios biológicos possuem dendritos por onde recebem estímulos, um corpo no qual tais sinais são processados, e um axônio responsável por emitir para fora do neurônio um sinal de saída, também os neurônios artificiais possuem um conjunto de canais de entrada, etapas de processamento e uma saída que pode ser ligada a outros neurônios (Figura 6.1).

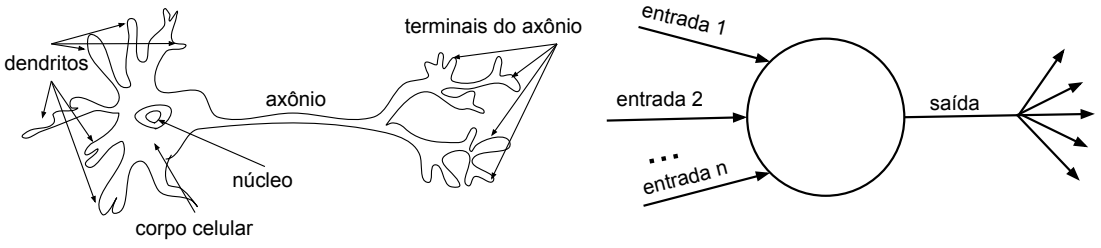

Figura 6.1: Neurônio Biológico e Neurônio Artificial

Ainda hoje utilizamos os fundamentos do modelo básico de neurônio artificial proposto em 1957 por Frank Rosenblatt, denominado Perceptron [Rosenblatt 1961]. Partindo de conceitos propostos pela neurociência, Rosenblatt introduziu a ideia de associar um peso a cada conexão de entrada do neurônio artificial de maneira a ponderar sua influência para uma determinada tarefa (Figura 6.2).

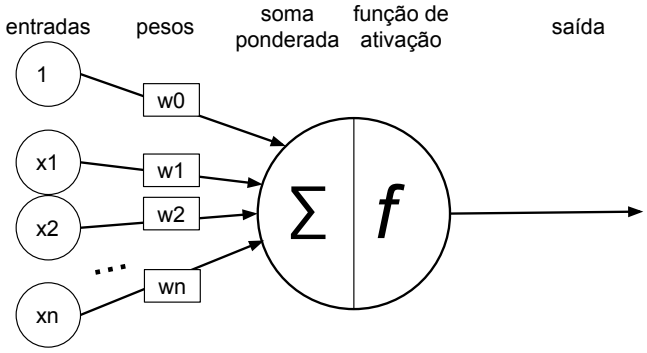

Figura 6.2: Perceptron: a combinação linear das entradas x_i ponderadas pelos pesos w_i é transformada pela função de ativação f na saída y emitida pelo neurônio

Assim, um neurônio que possui N conexões de entrada, associadas respectivamente aos pesos $W = \{w_i\}, 1 < i <= N$ produz uma transformação linear do sinal de entrada $X = \{x_i\}, 1 < i <= N$ descrita pela equação:

$$z = W \cdot X + b = \sum_{i=1}^{N} w_i x_i + b = w_1 x_1 + w_2 x_2 + ... + w_N x_N + b \quad (1)$$

Observe que além do vetor X de sinais de entrada e do vetor W de seus respectivos pesos é comum acrescentar ao neurônio um termo b chamado de bias. O bias não depende do sinal de entrada, mas aumenta os graus de liberdade da fronteira de decisão representada pelo neurônio, uma vez que permite que ela não necessariamente passe pela origem do sistema de coordenadas do sinal de entrada.

Para uniformizar a nomenclatura, é comum encontrarmos na literatura a inclusão do valor constante 1 como primeiro elemento do conjunto de entrada de maneira que b passa a ser representado pelo peso associado a tal entrada constante e passa a ser representado como o peso w_0. Com isso, a Equação 1 passa a ser reescrita como:

$$z = W \cdot X = \sum_{i=0}^{N} w_i x_i = w_0 1 + w_1 x_1 + w_2 x_2 + ... + w_N x_N \quad (2)$$

O neurônio artificial aplica uma nova transformação ao resultado da combinação linear dos sinais de entrada. Tal transformação é tipicamente não linear de maneira a aumentar o poder de expressão do neurônio e realizada pela função de ativação f.

Diferentes funções de ativação são utilizadas na formulação dos neurônios artificiais tais como a identidade, a tangente hiperbólica, a sigmoide hiperbólica, e a retificada linear (ReLU) (Figura 6.3).

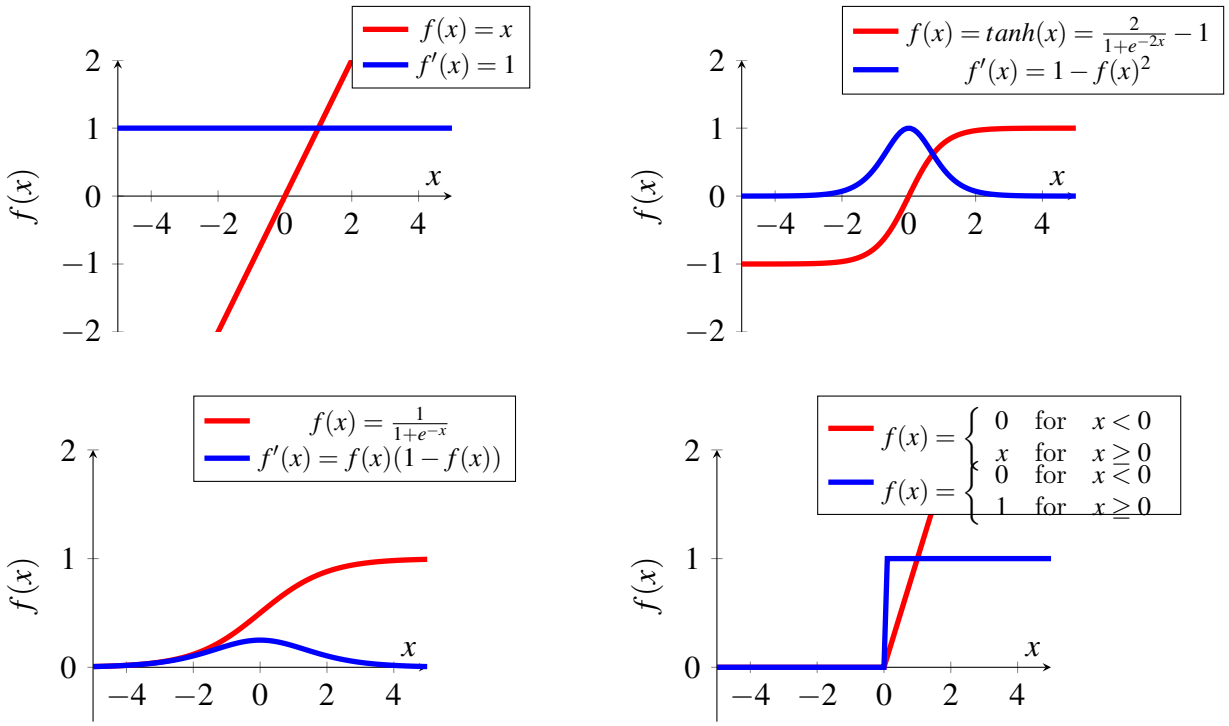

Figura 6.3: Exemplos de diferentes funções de ativação: função identidade, tangente hiperbólica, sigmoide hiperbólica (função logística), e a retificação linear (ReLU)

O perceptron de Rosenblatt foi formulado visando a construção de um classificador binário, ou seja, capaz de produzir como saída os valores 0 e 1. Nele a função de ativação desempenha o papel de aplicação de um limiar. A função de ativação adotada no perceptron é chamada de Função Degrau de Heaviside e definida como (Figura 6.4):

$$f(z) = \begin{cases} 1, & se\ W \cdot X + b > 0 \\ 0, & cc. \end{cases} \tag{3}$$

O perceptron desta forma definido consegue resolver problemas binários, no qual as possíveis saídas são linearmente separáveis. Portanto, é capaz de aprender as funções booleanas de negação (NOT), "e"(AND) e "ou"(OR). Entretanto, utilizando-se apenas um perceptron não é possível modelar a função booleana "ou exclusivo" (XOR), nem sua negação (NXOR). A Figura 6.5 ilustra fronteiras de decisão no espaço de entrada 2D estabelecidas por perceptrons recebendo duas entradas booleanas x_1 e x_2 e considerando $x_0 = 1$ como entrada constante de ativação do bias. Observe que são necessárias duas retas para modelar a operação XOR (portanto não sendo possível de representar com um único perceptron), enquanto que basta uma reta para definir as operações AND, OR e NOT. Tal limitação é

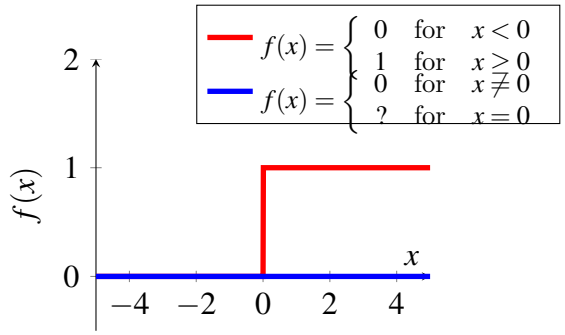

Figura 6.4: Função de ativação dos perceptrons: Função Degrau de Heaviside

desfeita pela combinação de mais perceptrons em camadas, conforme apresentado nas próximas seções.

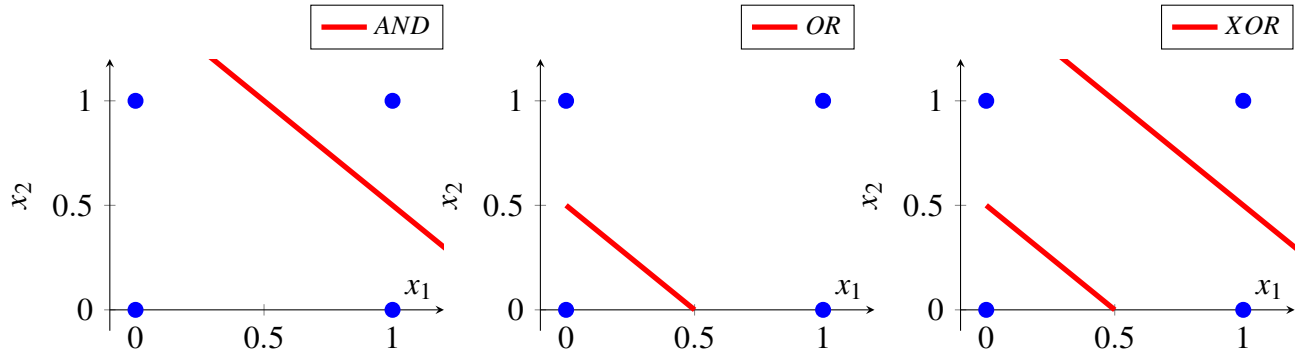

Figura 6.5: AND e OR: Operações booleanas modeladas por um único perceptron; XOR: duas retas são necessárias

6.2.1.1. Treinando o Perceptron

Em uma abordagem diferente da modelagem e programação de algoritmos específicos para resolver um problema, a mágica dos neurônios artificiais acontece quando os neurônios aprendem a resolvê-los.

No caso dos perceptrons, o processo de aprendizado é capaz de encontrar retas que separam entradas 2D, planos no caso de entradas 3D, ou ainda hiperplanos em dimensões maiores. Ao fim do aprendizado, podem ser utilizados para, dada um nova amostra, determinarem se esta se encontra de um lado ou de outro da fronteira de decisão aprendida.

O aprendizado parte de um conjunto de amostras de entrada sobre as quais são conhecidas as saídas desejadas. Cada amostra de treinamento é composta por um par (X,Y), no qual X é um vetor n-dimensional descrevendo um sinal de entrada, e Y é a resposta correspondente desejada.

Os parâmetros W e b configuram a fronteira de decisão modelada pelo neurônio, portanto, são os parâmetros a serem aprendidos durante o treinamento. Para treinar

um perceptron é comum inicializar seus n pesos do conjunto W com valores aleatórios (tipicamente entre -1 e 1) e deve-se associar também um valor inicial para seu bias b (tipicamente zero).

Uma vez inicializados, o treinamento em si consiste de duas etapas a serem alternadas repetidas vezes. A cada iteração t: (i) a primeira etapa processa um sinal de entrada X, produzindo um valor de saída $o(t)$; (ii) a segunda etapa ajusta os parâmetros do neurônio de acordo com a comparação entre a saída obtida $o(t)$ e o resultado desejado Y;

Ao executar a primeira etapa os valores dos parâmetros W e b são mantidos fixos e usados para ponderar os elementos de uma amostra de treinamento, que são então somados ao bias e aplicados à função degrau, obtendo-se uma saída $o(t) = f(z(X(t)))$ (vide equações 2 e 3).

Na segunda etapa acontece o aprendizado propriamente dito. Nela, o perceptron ajusta os pesos e bias observando quão diferente é o resultado $o(t)$ obtido com seus parâmetros atuais da resposta desejada Y fornecida no par de treinamento. O erro obtido na iteração t é calculado como $e(t) = Y - o(t)$ e é utilizado para atualizar os parâmetros por:

$$W(t+1) = W(t) + \alpha(Y - o(t))X \tag{4}$$

onde α representa a taxa de aprendizado (*learning rate*), t representa o passo da iteração, X representa o vetor com os canais da amostra de entrada acrescido de $x_0 = 1$ e W representa o vetor de pesos acrescido de $w_0 = b$.

O valor de α é um parâmetro escolhido previamente ao treinamento utilizando diferentes políticas e de alta influência ao processo de aprendizado. Ele estabelece o fator com que uma amostra contribui em uma iteração para a atualização dos parâmetros do perceptron. Logo, a Equação 4 pode ser interpretada como uma atualização proporcional ao erro da amostra avaliada (termo $y - o(t)$) e a quanto cada conexão de entrada x_i contribuiu para o erro uma vez que está relacionada a um w_i correspondente.

O processo de treinamento do perceptron é repetido até que por uma época o erro de treinamento seja inferior do que um limiar pré-estabelecido, ou ainda, pode ser repetido por uma quantidade pré-determinada de épocas. Denomina-se época, uma passagem do treinamento sob o conjunto de amostras completo.

6.2.2. Redes Neurais Artificiais

Existem diferentes maneiras de se conectar um conjunto de neurônios para formação de uma Rede Neural Artificial - RNA. A arquitetura de uma RNA define o padrão de disposição e conexão de seus neurônios. Focaremos nossa abordagem em dois tipos de arquiteturas diferenciadas pelo direcionamento de suas conexões.

A Subseção 6.2.2.1 introduz as arquiteturas nas quais o sinal é transmitido em uma única direção e por este motivo são chamadas de Redes Neurais Alimentadas para Frente (do inglês *Feedforward Neural Networks*).

Em seguida, a Subseção 6.2.2.3 introduz as arquiteturas nas quais há neurônios cujas saídas realimentam a si próprios e a outros neurônios de maneira a criar ciclos. Por esse motivo são chamadas Redes Neurais de Retroalimentação ou ainda Redes Neurais Recorrentes (do inglês *Recurrent Neural Networks*).

6.2.2.1. Redes Neurais Alimentadas para Frente

As Redes Neurais Alimentadas para Frente são também chamadas de Redes Neurais de Múltiplas Camadas ou ainda de Perceptron Múltiplas Camadas (MLP do inglês *multi-layer perceptron*). Adotaremos a sigla MLP embora não necessariamente utilizam neurônios idênticos aos perceptrons originais por substituir a função de ativação original.

Nas MLPs os neurônios são organizados em camadas sequenciais de maneira que o sinal fornecido como entrada é transmitido em uma direção apenas (Figura 6.6). Nessas arquiteturas os neurônios de uma mesma camada não são conectados entre si. Cada neurônio recebe sinais de entrada vindos de neurônios de camadas anteriores e por sua vez transmite o sinal por ele produzido para neurônios de camadas seguintes.

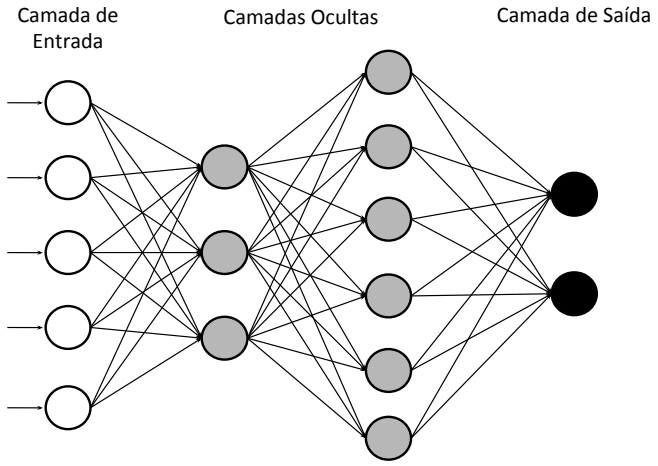

Figura 6.6: Estrutura de uma rede neural de múltiplas camadas. Os neurônios estão representados por círculos e as conexões por setas, todas em uma mesma direção, sem a formação de ciclos.

A estrutura em camadas é composta por: (i) uma camada inicial dita camada de entrada, responsável pela leitura dos dados a serem processados; (ii) uma ou mais camadas intermediárias, chamadas camadas escondidas, as quais não são nem de entrada nem de saída e são responsáveis pelo processamento propriamente dito; (iii) uma última camada responsável por emitir o resultado do processamento, ou seja, a saída da rede.

Deste ponto em diante do texto vamos incluir um novo índice j para di/-fe/-ren/-ciar de qual neurônio nos referimos. O conjunto formado pelos pesos associados às conexões (indexadas por i) de entrada do neurônio j passa a ser definido como

$W_j = \{w_{ji}\}$ (Figura 6.7) e o conjunto das ativações que alimentam tais conexões como $X_j = \{x_{ji}\}$.

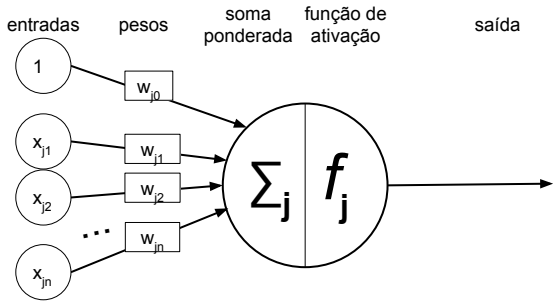

Figura 6.7: Neurônio j e suas conexões de entrada de dados, cada uma associada a um índice i

Nesta notação, a combinação linear das N entradas de um neurônio j passa a ser escrita como:

$$z_j = W_j \cdot X_j = \sum_{i=1}^{N} w_{ji} x_{ji} \tag{5}$$

Enquanto que a função de ativação f, a qual produz a saída o_j emitida pelo neurônio, passa a ser escrita como:

$$f(z_j) = f(W_j \cdot X_j) = f(\sum_{i=1}^{N} w_{ji} x_{ji}) = o_j \tag{6}$$

O resultado obtido pelo neurônio j passa a alimentar outros neurônios de camadas seguintes. Supondo existir uma conexão de índice i a partir do neurônio j para o neurônio k, tal conexão é ativada com valor o_j. Logo, temos $o_j = x_{ki}$ e portanto o_j passa a fazer parte do conjunto X_k. Uma vez que a saída de um neurônio j pode estar conectada a diferentes neurônios em camadas seguintes, os elementos dos conjuntos de ativações de diferentes neurônios não são necessariamente únicos.

Podemos descrever o algoritmo de propagação de um sinal X por uma rede neural alimentada para frente de L camadas ocultas, parametrizada por $L+1$ matrizes de pesos W como:

Algorithm 1 Propagação MLP

procedure MLPFORWARD(X, W)
  $x(0) \leftarrow X$.
  $c \leftarrow 1$.
  for $c <= L+1$ **do**
    $z(c) \leftarrow W(c-1)x(c-1)$ ▷ combinação linear
    $x(c) \leftarrow f(z(c))$ ▷ transformação não-linear
  return $o \leftarrow x(L+1)$

Partindo de um modelo de arquitetura, a topologia de uma RNA descreve sua composição estrutural específica, tal como o número de neurônios utilizados, a(s) função(ões) de ativação escolhida(s).

Sobre a formulação de uma MLP para um determinado problema, a definição da topologia das suas camadas de entrada e saída é normalmente simples e intuitiva de ser definida. Isso porque essas camadas estão diretamente relacionadas respectivamente a dimensionalidade dos dados que queremos fornecer para a rede e a dimensionalidade da solução desejada.

As mais diferentes arquiteturas com alimentação para frente exploram variações nas camadas ocultas, uma vez que ao modificá-la altera-se de fato o mapeamento modelado pela rede. A Seção 6.3 apresenta um tipo de arquitetura de alimentação para frente inspirada no processamento biológico de dados visuais.

Uma vez definida uma arquitetura, a escolha do número de camadas ocultas e do número e disposição de seus neurônios e conexões é muitas vezes realizada por experimentação.

6.2.2.2. Treinando Redes Alimentadas para Frente: Algoritmo de Retropropagação

Vimos na Seção 6.2.1.1 o algoritmo para treinamento de perceptrons, os quais também podem ser pensados como redes neurais simples de uma única camada oculta. Nesta seção apresentamos o algoritmo de Retropropagação (do inglês *Backpropagation*) para treinamento de redes neurais de múltiplas camadas.

Para realizar o treinamento de tais redes acrescenta-se uma extensão após a sua camada de saída (ilustrada na Figura 6.8), responsável por calcular uma função E de erro ou perda (do inglês *loss function*). É essa função de erro que se deseja minimizar ao longo do processo de treinamento.

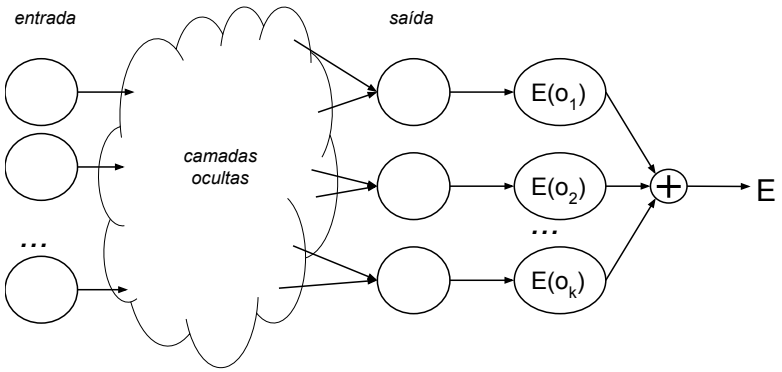

Figura 6.8: Extensão da rede criada para o cálculo do erro.

Uma vez definida uma rede MLP, os parâmetros a serem ajustados para diminuir o erro segundo a função E adotada são os pesos associados as suas conexões. Portanto, são eles os parâmetros que se deseja aprender com o processo de treinamento.

O primeiro passo do algoritmo de retropropagação é a inicialização dos valores dos pesos da rede. É comum o uso de pesos iniciais aleatórios em torno de zero (como

por exemplo amostrados de uma distribuição normal uniforme de média zero e desvio padrão um).

Uma vez concluída a inicialização, o algoritmo passa a repetir as seguintes duas etapas até que o critério de parada escolhido seja atingido (Figura 6.9):

- **Etapa de propagação do sinal**: fixando-se os pesos, um sinal de entrada $X(t)$ é processado na direção estabelecida pelas conexões, até produzir a saída $o(t)$ correspondente a tal amostra de treinamento. O erro de treinamento da iteração t é então calculado comparando a resposta da rede $o(t)$ com a saída desejada $y(t)$ fornecida no par de treinamento utilizando-se a função de erro $E(t)$ previamente escolhida.

- **Etapa de retropropagação do erro**: partindo-se da camada de saída da rede é feita uma retropropagação do erro na direção oposta à estabelecida pelas conexões até atingir os neurônios da camada de entrada. Uma vez estimado quanto cada conexão influenciou o erro obtido, os pesos são atualizados.

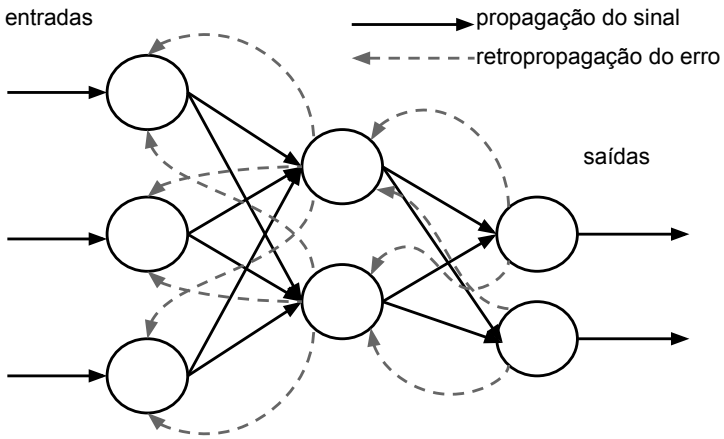

Figura 6.9: Fluxograma do algoritmo de retro-propagação: propagação do sinal na direção das conexões e retro-propagação do erro na direção oposta

Na descrição em alto nível apresentada, falta detalhar como são atualizados os pesos. O algoritmo de retropropagação realiza um processo de otimização da função E, ou seja, busca minimizá-la, sendo os pesos as variáveis a serem manipuladas com esse objetivo. Assumiremos uma otimização de gradiente descendente estocástico online no qual os pesos são atualizados baseados na observação de uma amostra de treinamento por vez, apresentadas em ordem aleatória.

Para ajustar um determinado w_{ji} segundo a otimização de gradiente descendente, o algoritmo estima como este peso influenciou o erro obtido na iteração atual t. Para isso calcula localmente a inclinação de $E(t)$ em relação a $w_{ji}(t)$ enquanto assume fixos os demais pesos. Matematicamente, a tangente da função de erro avaliada em $E(t)$ em relação a um determinado peso $w_{ji}(t)$ é obtida pela derivada parcial de $E(t)$ em relação a tal peso: $\partial E(t)/\partial w_{ji}(t)$. Por sua vez, o vetor gradiente da

função de erro em relação aos pesos da rede é formado pelas derivadas parciais de $E(t)$ por cada um dos elementos de $W(t)$:

$$\nabla E(t) = \left\langle \frac{\partial E(t)}{\partial w_{1,1}(t)}, \frac{\partial E(t)}{\partial w_{1,2}(t)}, \frac{\partial E(t)}{\partial w_{1,\dots}}, \frac{\partial E(t)}{\partial w_{2,1}(t)}, \frac{\partial E(t)}{\partial w_{2,2}(t)}, \frac{\partial E(t)}{\partial w_{2,\dots}}, \cdots, \frac{\partial E(t)}{\partial w_{n,1}(t)}, \frac{\partial E(t)}{\partial w_{n,2}(t)}, \frac{\partial E(t)}{\partial w_{n,\dots}}, \right\rangle \quad (7)$$

O vetor gradiente indica a direção e o sentido de maior inclinação de $E(t)$ em relação a um conjunto de parâmetros. Por conta dessa propriedade, os métodos conhecidos como gradiente descendente utilizam $-\nabla E(t)$ para caminhar em E de maneira a minimizá-la. Operam alternando entre o cálculo do erro, o respectivo vetor gradiente em relação aos parâmetros observados, e a atualização de tais parâmetros por um pequeno passo na direção negativa a encontrada.

Com isso, no treinamento de redes MLP seus pesos são atualizados somando-se a eles o negativo do vetor $\nabla E(t)$ ponderado pela taxa de aprendizado α a qual regula o tamanho do passo a ser dado na direção do gradiente:

$$W(t+1) = W(t) - \alpha \nabla E(t) \quad (8)$$

Em relação a um determinado peso $w_{ij}(t)$ podemos escrever:

$$w_{ji}(t+1) = w_{ji}(t) - \alpha \frac{\partial E(t)}{\partial w_{ji}(t)} \quad (9)$$

Falta ainda detalhar como o algoritmo de retropropagação encontra as derivadas parciais $\partial E(t)/\partial w_{ji}(t)$ em uma determinada iteração. O algoritmo de retropropagação propõe o uso da regra da cadeia para abrir $\partial E(t)/\partial w_{ji}(t)$ nos passos de processamento realizados pela rede entre a ativação da conexão i de alimentação do neurônio j até a saída e cálculo da função de erro.

Supondo j um neurônio da camada de saída, z_j a combinação linear de suas entradas (Equação 5), f_j a função de ativação aplicada ao resultado de z_t (Equação 6) e supondo uma função de erro atuando sobre as saídas dos neurônios da última camada, podemos reescrever $\partial E(t)/\partial w_{ji}(t)$ como (Figura 6.10):

$$\frac{\partial E(t)}{\partial w_{ji}(t)} = \frac{\partial E(t)}{\partial f_j(t)} \frac{\partial f_j(t)}{\partial z_j(t)} \frac{\partial z_j(t)}{\partial w_{ji}(t)} \quad (10)$$

onde:

- o termo $\partial E(t)/\partial f_j(t)$ pode ser obtido diretamente da derivação da função de erro em relação a saída produzida pelo neurônio j, avaliada em $f_z(t)$;
- o termo $\partial f_j(t)/\partial z_j(t)$ pode ser obtido pela derivação da função de ativação adotada, avaliada em $z_j(t)$;
- e por sua vez o termo $\partial z_j(t)/\partial w_{ji}(t)$, por representar a derivação da combinação linear das entradas de j em relação ao peso $w_{ji}(t)$, assume o valor fornecido como entrada para tal conexão, portanto $x_{ji}(t)$.

Com esses valores, pode-se aplicar a regra definida pela Equação 9 para atualizar os pesos da última camada. É comum reescrevê-la de maneira a isolar os termos que não dependem de $w_{ji}(t)$ usando:

$$\delta_j = \frac{\partial E(t)}{\partial z_j(t)} \tag{11}$$

e reescrevemos as Equações 10 e 8 respectivamente como:

$$\frac{\partial E(t)}{\partial w_{ji}(t)} = \delta_j(t) x_{ji}(t) \tag{12}$$

$$W(t+1) = W(t) - \alpha \delta_j(t) * X_j \tag{13}$$

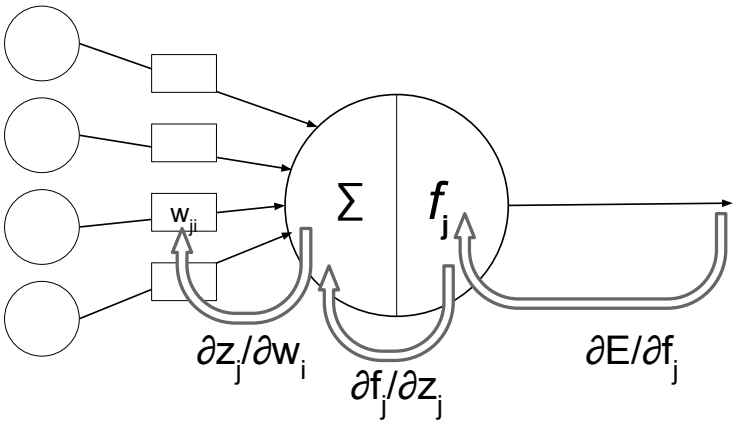

Figura 6.10: Reescrita da derivada parcial do erro em relação aos neurônios da última camada

Pesos associados a conexões que alimentam neurônios de outras camadas continuam esse processo de retropropagação. Tomando como ponto de partida a Equação 10 para avaliar o que ocorre em neurônios das demais camadas, observamos que o termo $\frac{\partial f_j(t)}{\partial z_j(t)}$ pode ser calculado uma vez conhecida a função de ativação do neurônio, e $\frac{\partial z_j(t)}{\partial w_{ji}(t)}$ é obtido como o valor do sinal que ativou a conexão, portanto $x_{ji}(t)$.

Entretanto, o termo $\frac{\partial E(t)}{\partial f_j(t)}$ não é mais obtido diretamente da derivação de E, uma vez que neurônios das camadas ocultas não produzem diretamente a saída da rede. O sinal produzido como sua saída em f_j se propaga por n conexões para neurônios da(s) camada(s) seguinte(s), na forma de ativações $x_k i$. Portanto $\frac{\partial E(t)}{\partial f_j(t)}$ precisa ser reescrito como a soma das derivadas parciais de E em relação a essas n conexões por onde $f_j(t)$ é propagado.

$$\frac{\partial E(t)}{\partial f_j(t)} = \sum_{k=1}^{n} \frac{\partial E(t)}{\partial x_{ki}(t)} = \sum_{k=1}^{n} \frac{\partial E(t)}{\partial z_k(t)} \frac{\partial z_k(t)}{\partial x_{ki}(t)} = \sum_{k=1}^{n} \frac{\partial E(t)}{\partial z_k(t)} w_{ki}(t) = \sum_{k=1}^{n} \delta_k(t) w_{ki}(t) \tag{14}$$

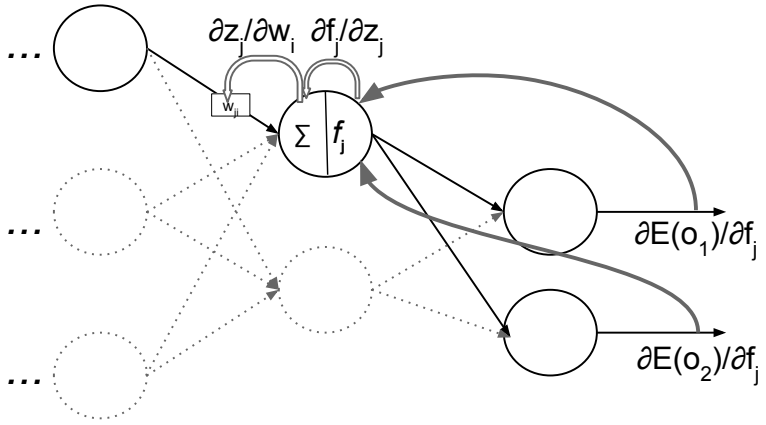

Figura 6.11: Reescrita da derivada parcial do erro em relação aos neurônios de camadas escondidas

Usando a definição de $\delta_j(t)$ (Equação 11) podemos ainda escrever:

$$\delta_j(t) = \frac{\partial f_j(t)}{\partial z_j(t)} \sum_{k=1}^{n} \delta_k(t) w_{ki}(t) \tag{15}$$

De onde observamos que o algoritmo de retropropagação do erro requer que as funções de ativação adotadas para os neurônios sejam diferenciáveis. Com a Equação 15 é possível retropropagar os δ de camadas mais profundas para anteriores, e atualizar os pesos aplicando-as na Equação 13.

Logo, o algoritmo de retropropagação pode ser descrito como:

- Aplica-se uma amostra de treinamento pela rede propagando-a usando as equações 5 e 6 para encontrar as ativações de todos os neurônios das camadas escondidas e de saída.
- Os valores de δ_k para todos os neurônios de saída são calculados usando as equações 11 e 10.
- Os δs encontrados são retropropagados usando a Equação 15 para obter δ_j de cada neurônio de camadas escondidas.
- Para aplicar a atualização dos pesos, usa-se a Equação 14 e os valores precomputados dos δs na Equação 9.

6.2.2.3. Redes Neurais Recorrentes

Nesta seção apresentamos uma introdução ao modelo de redes com retroalimentação chamadas Redes Neurais Recorrentes (do inglês *Recurrent Neural Networks*-RNN). Diferentemente das redes alimentadas para frente, em RNNs a saída de alguns neurônios retro-alimentam a si próprios e a outros neurônios formando ciclos

Algorithm 2 Retro-propagação MLP

```
1: procedure MLPBACK(X, W0)
2:     dx(L+1) ← dE(x(L+1), y)/dx(L+1)
3:     for c from L+1 downto 1 do
4:         dz(c) ← df/dz(z(c)) · dx(c)                    ▷ δ(c)
5:         dx(c−1) ← W^T(c−1) dz(c)
6:         dW(c−1) ← dz(c) x^T(c−1)
       return dW[0 ··· L]
```

no percurso de processamento. Tais arranjos formadores de ciclos permitem que haja informação persistente entre computações, ou seja, valores que se propagam ao longo de uma sequência de análises na forma de ativações das conexões de retro-alimentação. Esses valores são considerados estados escondidos ou ocultos por não terem sidos fornecidos como sinais de entrada, nem serem emitidos como de saída da rede.

Por permitirem que informações sejam passadas de uma etapa de processamento para próxima, RNNs são capazes de lidar com sinais na forma de sequências. Seja X o vetor com o sinal de entrada da rede composto por uma sequência de N_s elementos, onde cada elemento é indexado como $X(t)$ e é descrito por N_c valores ou canais. Logo, a sequência completa é composta por um vetor de tamanho $N_s * N_c$.

Uma mesma RNN pode lidar com sequências de tamanhos variados, não exigindo que N_s seja fixo. Isso porque a RNN é modelada para receber um dos elementos $X(t)$ da sequência por vez (de onde sua camada de entrada tem N_c neurônios), e para ser replicada ao longo da sequência.

Como ilustração de uma topologia de RNN, a Figura 6.12 apresenta uma rede capaz de processar sequências de tamanho qualquer contendo elementos descritos cada um por 2 valores. A RNN ilustrada possui uma camada de entrada de dois neurônios (em correspondência ao número de canais do sinal de entrada), ligados a uma camada escondida de 3 neurônios interconectados entre si e a uma camada de saída formada por 2 neurônios. Observe na Figura 6.13 que a rede é replicada para processar os elementos da sequência de entrada, indexados na ilustração como $t-1$, t, $t+1$.

Embora outras construções possam ser elaboradas, por exemplo aumentando o número de camadas escondidas, assumiremos por simplicidade um modelo geral de uma RNN com uma única camada escondida, com uma quantidade de neurônios qualquer, na qual seus neurônios possuem uma conexão para si próprios, para os outros neurônios da mesma camada e para os neurônios de saída da rede.

Os neurônios recursivos de uma RNN seguem em alto nível o modelo geral de neurônios artificiais no sentido de aplicarem uma combinação linear de suas entradas, seguida de uma transformação não-linear. Entretanto, não somente utilizam de tal combinação de transformações para produzir seu valor de saída, mas também para atualizar os estados escondidos, que irão realimentar a si próprios e a outros neurônios da rede no tratamento do próximo elemento da sequência (Figura 6.14).

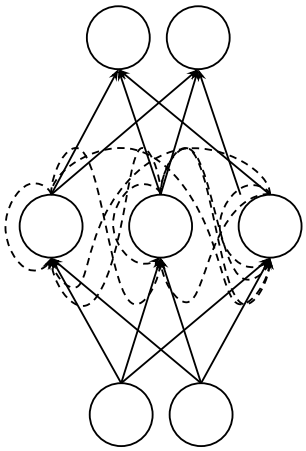

Figura 6.12: Ilustração de topologia RNN com 2 neurônios na camada de entrada, 3 na camada escondida, responsáveis pela recorrência, e 2 na camada de saída

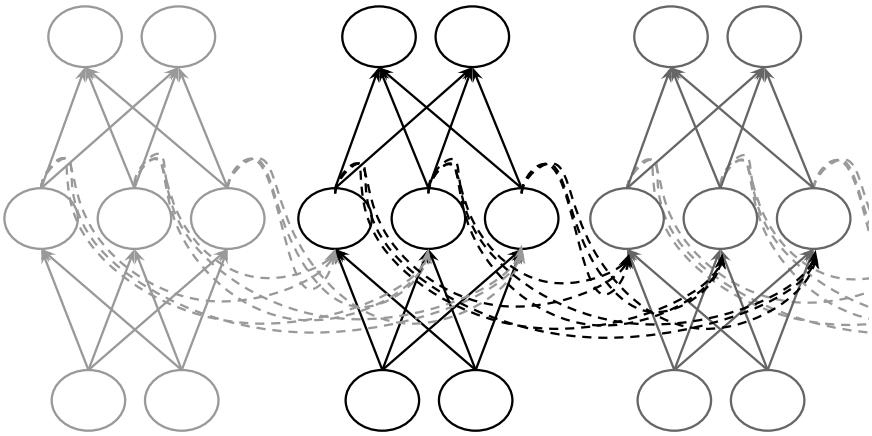

Figura 6.13: RNN da Figura 6.12, replicada em $t-1$, t e $t+1$.

O estado $h(t)$ produzido por um neurônio recursivo de função de ativação f_h, na iteração t pode ser encontrado como:

$$
\begin{aligned}
h(1) &= f_h(W_h h(0) + W_e X(1) + b_e) \\
h(2) &= f_h(W_h h(1) + W_e X(2) + b_e) \\
&= f_h(W_h(f_h(W_h h(0) + W_e X(1) + b_e)) + W_e X(2) + b_e) \\
h(3) &= f_h(W_h h(2) + W_e X(3) + b_e) \\
&= f_h(W_h(f_h(W_h(f_h(W_h h(0) + W_e X(1) + b_e)) + W_e X(2) + b_e)) + W_e X(3) + b_e) \\
h(t) &= f_h(W_h h(t-1) + W_e X(t) + b_e) \qquad (16) \\
&= f_h(W_h(f_h(W_h \cdots (f_h(W_h h(0) + W_e X(1) + b_e)) \cdots + W_e X(t-1) + b_e)) + W_e X(t) + b_e) \qquad (17)
\end{aligned}
$$

Onde,

- $h0$ é o estado inicial dos estados escondidos, o qual pode ser inicializado com zeros, um valor fornecido ou aprendido ;

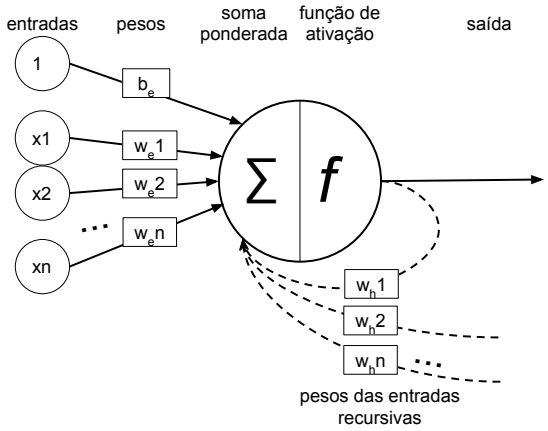

Figura 6.14: Neurônio recursivo

- W_e, b_e: representam respectivamente a matriz de pesos e o vetor de bias das conexões alimentadas com o sinal de entrada;
- W_h: representa a matriz de pesos das conexões recursivas;
- f_h: representa a função de transformação não-linear dos neurônios recursivos

Já a saída produzida pela rede na iteração t, a partir do valor produzido pelo estado escondido é encontrada por:

$$o(t) = f_s(W_s h(t) + b_s) \tag{18}$$

onde,

- W_s, b_s: representam respectivamente a matriz de pesos e o vetor de bias das conexões partindo dos neurônios recursivos para os neurônios da camada de saída;
- f_s: representa a função de transformação não-linear dos neurônios de saída.

Segundo este modelo, dada uma sequência X, o processamento realizado por uma RNN de uma única camada escondida pode ser descrito como:

Portanto, RNNs são parametrizadas por 3 matrizes de pesos (W_e,W_h e W_s), 2 vetores bias (b_e e b_s) e, opcionalmente, um vetor de valores para definir seu estado escondido inicial ($h0$). São esses os parâmetros a serem aprendidos por um processo de treinamento.

6.2.2.4. Treinando Redes Neurais Recorrentes

Assim como no treinamento de redes alimentadas para frente, também em RNNs o treinamento parte de uma função de perda ou erro E. Como a saída esperada neste caso é uma sequência, a função de perda tipicamente é formulada como a soma

Algorithm 3 Processamento realizado por uma RNN

1: **procedure** RNNRUN(X, N_s, $h0$)
2: $h(0) \leftarrow h0$. ▷ inicializa estado oculto
3: $t \leftarrow 1$.
4: **for** $t <= N_s$ **do**
5: $z_h(t) \leftarrow W_e X(t) + W_h h(t-1) + b_e$ ▷ combinação linear do sinal de entrada e estado oculto
6: $h(t) = f_h(z_h(t))$ ▷ atualiza estado oculto
7: $z_s(t) \leftarrow W_s h(t) + b_s$ ▷ combinação linear dos resultados da camada escondida
8: $o(t) = f_s(z_s(t))$ ▷ calcula saída
9: $t \leftarrow t+1$

dos erros de cada instante t comparando elemento a elemento a sequência desejada $y = \{y(1), y(2), ..., y(N_s)\}$ com a produzida pela rede $o = \{o(1), o(2), ..., o(N_s)\}$. Mais formalmente:

$$E(o,y) = \sum_{t=1}^{N_s} E(o(t), y(t)) \tag{19}$$

O treinamento da RNN pode ser realizado como uma otimização por gradiente descendente, de maneira semelhante ao que foi apresentado na Seção 6.2.2.2. Neste caso o vetor gradiente da função de erro é calculado em relação aos 6 parâmetros de configuração da RNN que se deseja aprender.

De maneira semelhante ao treinamento apresentado para MLP, o algoritmo chamado Retropropagação Através do Tempo (do inglês *backpropagation through time*) aplica a regra da cadeia para decompor as derivadas parciais da função de erro da Equação 19 nas etapas de processamento da RNN da saída da rede até os respectivos parâmetros. Para isso, desenrola a RNN em uma rede sem laços. Isso é possível porque a RNN aplicada a uma sequência pode ser pensada como cópias de uma mesma rede, cada uma passando uma mensagem a cópia sucessora. O desenrolar é feito replicando a rede RNN e seus parâmetros em tantas cópias quantos forem os elementos da sequência que se deseja processar, e transformando as conexões de retroalimentação em conexões de alimentação para frente entre o neurônio de uma determinada cópias e os neurônios da cópia seguinte ao longo da sequência de redes criada. Assim, 'desenrola-se' a RNN ao longo do tempo em uma rede de alimentação para frente, mas com parâmetros comuns compartilhados entre as cópias (Figura 6.16).

Enquanto as ativações que trafegam na rede são dinâmicas e estão relacionadas a um determinado instante de processamento, por outro lado os parâmetros de configuração da rede são fixos durante todo o processamento de uma sequência e portanto são compartilhados nas N_s cópias da rede. Por esse motivo, no cálculo das derivadas do erro parcial em cada um deles, tais erros são acumulados ao longo da

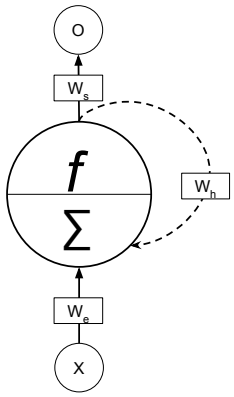

Figura 6.15: Ilustração simplificada da RNN agrupando conexões da mesma natureza

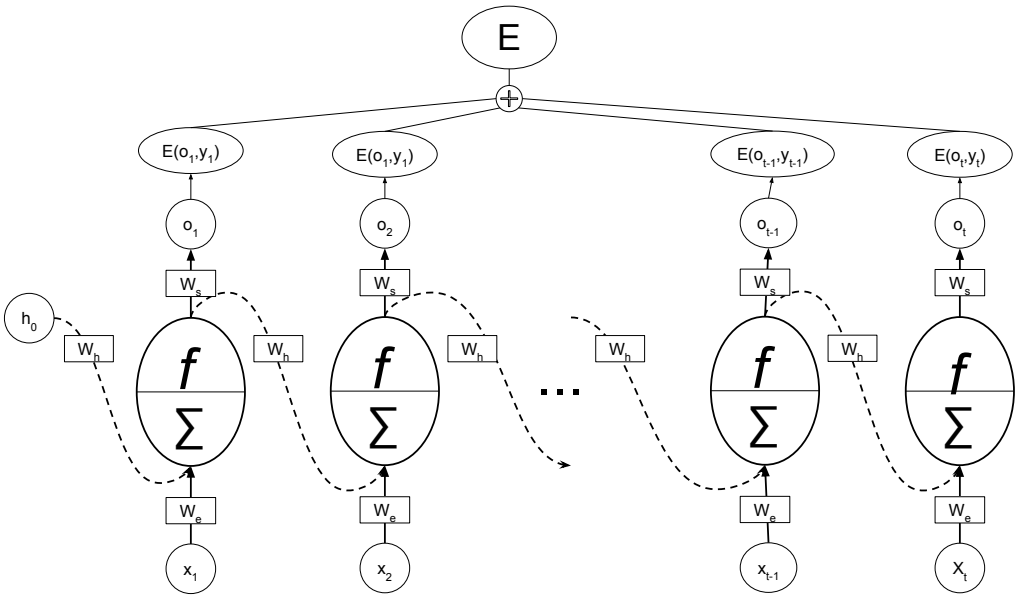

Figura 6.16: Erro aplicado ao modelo de RNN simplificado 'desenrolado'

sequência:

$$\frac{\partial E(o,y)}{\partial W_s} = \sum_{i=1}^{T} \frac{\partial E(o(t),y(t))}{\partial z_s(t)} h(t)^\intercal = \sum_{i=1}^{T} \delta z_s(t) h(t)^\intercal \tag{20}$$

$$\frac{\partial E(o,y)}{\partial b_s} = \sum_{i=1}^{T} \frac{\partial E(o(t),y(t))}{\partial z_s(t)} = \sum_{i=1}^{T} \delta z_s(t) \tag{21}$$

$$\frac{\partial E(o,y)}{\partial W_h} = \sum_{i=1}^{T} \frac{\partial E(o(t),y(t))}{\partial z_h(t)} h(t-1)^\intercal = \sum_{i=1}^{T} \delta z_h(t) h(t-1)^\intercal \tag{22}$$

$$\frac{\partial E(o,y)}{\partial W_e} = \sum_{i=1}^{T} \frac{\partial E(o(t),y(t))}{\partial z_h(t)} X(t)^\intercal = \sum_{i=1}^{T} \delta z_h(t) X(t)^\intercal \tag{23}$$

$$\frac{\partial E(o,y)}{\partial b_e} = \sum_{i=1}^{T} \frac{\partial E(o(t),y(t))}{\partial z_s(t)} = \sum_{i=1}^{T} \delta z_h(t) \tag{24}$$

Assim, utilizando a regra da cadeia, o algoritmo de retropropagação do erro através do tempo é descrito em pseudo-código como:

Algorithm 4 Retropropagação através do tempo

1: **procedure** RNNBACK(X, N_s, $h0$)
2: o = RNNrun(X, N_s, $h0$)
3: $t \leftarrow N_s$.
4: **for** $t > 0$ **do**
5: $dz_s(t) \leftarrow df_s(z_s(t)) \cdot dL(o(t), y(t))/do(t)$ ▷ δz_s
6: $db_s \leftarrow db_s + dz_s(t)$
7: $dW_s \leftarrow dW_s + dz_s(t)h(t)^\intercal$
8: $dh(t) \leftarrow dh(t) + W_s^\intercal dz_s(t)$ ▷ parte 1
9: $dz_h(t) \leftarrow df_h(z_h(t))dh(t)$ ▷ δz_h
10: $dW_e \leftarrow dW_e + dz_h(t)X(t)^\intercal$
11: $db_e \leftarrow db_e + dz_h(t)$
12: $dW_h \leftarrow dW_h + dz_h(t)h(t-1)^\intercal$
13: $dh(t-1) \leftarrow W_h^\intercal dz_h(t)$ ▷ parte 2
14: $t \leftarrow t-1$
return $[dW_e, dW_h, dW_s, db_e, db_s, dh(0)]$

Uma observação é que, assim como o sinal percorre a rede em duas direções (direção de emissão da saída $o(t)$ e direção de atualização do estado oculto $h(t)$), também o erro em relação a h deve ser retropropagado por esses dois caminhos. Eles podem ser identificados respectivamente no pseudo código como a parte 1 e parte 2 de contribuição para a dh.

Na prática, o treinamento de RNNs apresenta dois problemas, inicialmente apontados por Bengio et al. [Bengio et al. 1994], decorrentes da propagação do erro através de múltiplas iterações não-lineares: explosão do gradiente e fuga/diluição do gradiente. Seu detalhamento formal pode ser consultado em [Pascanu et al. 2013].

O problema dito explosão do gradiente refere-se a observação de um grande aumento da norma do vetor gradiente durante o treinamento de relações de longo prazo. Neste efeito, as derivadas da função de perda de um determinado instante em relação às ativações de muitos passos atrás podem ser exponencialmente grandes. Tal aumento é causado por um efeito borboleta na retropropagação do erro por longas cadeias, onde uma pequena alteração resulta em um grande efeito muitas iterações depois.

O problema da diluição/desaparecimento do gradiente refere-se ao comportamento oposto e mais frequentemente observado, quando componentes da retropropagação do gradiente de longo prazo vão exponencialmente rápido para a norma 0, impossibilitando o modelo de aprender correlação entre eventos temporalmente distantes.

Mesmo quando os parâmetros assumem valores que estabilizam o gradiente (sem sumir ou explodir), a dificuldade do aprendizado de dependências de longo alcance permanece. Isso pelo fato de que a multiplicação repetida das Matrizes

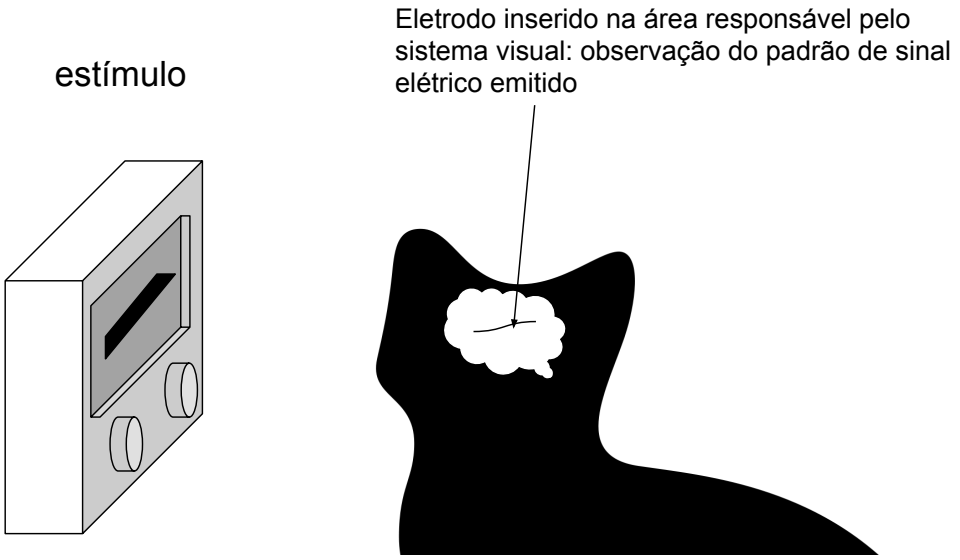

Figura 6.17: Observação dos campos receptivos de mamíferos com estímulo de diferentes padrões visuais [Hubel and Wiesel 1959, Hubel and Wiesel 1968, Hubel and Wiesel 2005]

Jacobianas [1], por conta da retro-propagação do gradiente por diversas iterações, produz pesos exponencialmente menores para aprendizado de dependências a longo prazo quando comparados a dependências de curto prazo.

Tais motivos fazem com que seja difícil treinar RNNs em sua formulação original em sequências com dependências temporais de longo alcance. Felizmente, há arquiteturas que buscam reduzir tais problemas, conforme apresentado na Seção 6.4.

6.3. Redes Neurais de Convolução

Esta seção apresenta as chamadas Redes Neurais de Convolução (do inglês *Convolutional Neural Networks – CNN ou ConvNets*), que são redes com arquiteturas de alimentação para frente especialmente projetadas para processamento de sinais visuais.

CNNs foram inspiradas pelas descobertas dos neurocientistas Hubel e Wiesel nas décadas de 1950 e 1960 sobre a organização do córtex visual de animais (Figura 6.17) [Hubel and Wiesel 1959, Hubel and Wiesel 1968, Hubel and Wiesel 2005]. Nele há neurônios que individualmente respondem a pequenas regiões do campo visual. A subregião do campo visual observado que dispara um determinado neurônio é chamada seu campo receptivo.

Observaram também um padrão relacionando a distribuição espacial dos

[1] A Matriz Jacobiana é composta pelas derivadas parciais de um ponto específico x de uma função de múltiplas variáveis. No algoritmo de retropropagação ela indica como uma pequena mudança no estado h se propaga (e se retro-propaga) e é multiplicada por si mesma por conta da propagação do erro pelas conexões recorrentes.

neurônios com a de seus respectivos campos receptivos segundo o qual neurônios vizinhos apresentam campos receptivos semelhantes e com sobreposição entre si, de maneira a formar em conjunto um mapa completo do campo visual observado. Diferentes mapas são formados com variação do padrão de tamanho e localização dos campos receptivos e em níveis crescentes de abstração semântica. Essas observações guiaram a base da modelagem das arquiteturas de CNNs, conforme descrito a seguir (Figura 6.18).

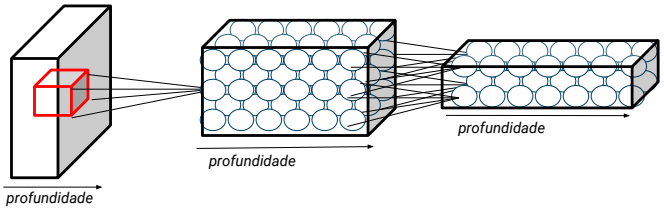

Figura 6.18: Organização espacial dos neurônios em CNNs em grades de altura e largura coerentes com o sinal de recebido como entrada por cada camada e profundidade correspondente ao número de mapas produzidos pela camada

Com exceção da primeira e das últimas camadas de uma CNN, os neurônios de cada camada são organizados em uma grade com altura e largura suficientes para cobertura do sinal de alimentação correspondente e profundidade representando diferentes processamentos sobre tal sinal. Nesta organização, uma fatia da grade é obtida fixando-se uma profundidade. Neurônios vizinhos no interior de uma fatia possuem sobreposição entre suas conexões de entrada, simulando a sobreposição de campos receptivos do córtex. Já os neurônios em uma determinada altura e largura fixas, possuem o mesmo campo receptivo, mas realizam diferentes operações sobre o campo observado (Figura 6.19).

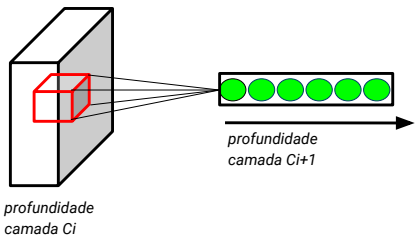

Figura 6.19: Neurônios de uma mesma altura e largura em uma determinada camada da CNN compartilham campos receptivos, mas realizam diferentes operações sobre o sinal de entrada

Os campos receptivos, por sua vez, são modelados limitando as conexões de entrada em cada neurônio apenas a elementos no interior de uma determinada vizinhança estabelecida sobre os sinais de camadas anteriores. Observe que essa modelagem das conexões apenas sobre uma vizinhança difere do modelo mais geral de redes alimentadas para frente no qual um neurônios de uma determinada camada é conectado a todos os neurônios da camada anterior e por esse motivo são ditas camadas completamente conectadas.

Sobre as atividades desempenhadas pelas células do córtex visual, Hubel e Wiesel observaram dois tipos de comportamento: o das células simples, que têm seu disparo maximizado quando seu campo receptivo apresenta arestas em orientações particulares, e o das células complexas, cujo disparo é insensível a posição exata das arestas no campo observado, e que cobrem campos receptivos maiores do que os das células simples. Esses dois tipos de células inspiram respectivamente as chamadas camadas de convolução e camadas de agrupamento (do inglês *pooling*) presentes de maneira intercalada em redes CNN.

As camadas de convolução de uma CNN realizam a extração de padrões visuais tais como arestas, texturas, motivos e ainda outros padrões visuais de maior significado semântico. Quando a aplicação da CNN não se tratar de imagens, as camadas de convolução são responsáveis pela extração de características relevantes ao tipo de sinal sendo processado. Para esse fim, desempenham operações semelhantes a aplicação de filtros pelo operador de convolução adotado na área de Processamento de Imagens e de Sinais em geral (Figura 6.23). A convolução de um sinal 1D de domínio discreto F por um filtro W de tamanho $s = (2*k+1)$ é o operador linear definido como:

$$(F*W)[x] = \sum_{i=-k}^{k} F[x-i]W[i] \tag{25}$$

A convolução é replicada por toda a extensão de f variando-se x para obter o sinal de saída 1D. A Figura 6.20 ilustra a aplicação de um filtro onde $s = 3$.

Figura 6.20: Convolução de um sinal 1D por um filtro de tamanho 3

De maneira semelhante, a convolução pode ser definida para operar em sinais com mais dimensões. Seja F um sinal 2D e W um filtro de tamanho $(2*k+1)\times(2*k+1)$, a convolução $(F*W)$ é o operador linear definido como:

$$(F*W)[x,y] = \sum_{i=-k}^{k}\sum_{j=-k}^{k} F[x-i][y-j]W[i][j] \tag{26}$$

A convolução é replicada por toda a extensão de F variando-se x,y para obter o sinal de saída 2D (Figura 6.21).

Enquanto na área de Processamento de Sinais os valores dos filtros são cuidadosamente definidos por especialistas de maneira a representar uma transformação desejada, as CNNs realizam convoluções por filtros cujos pesos (W) são aprendidos pelo processo de treinamento.

O conjunto de neurônios de uma fatia replica o mesmo operador de convolução a diferentes regiões do sinal de entrada de maneira a representar a aplicação de um determinado filtro sobre todo o sinal (Figura 6.22). Para isso, os neurônios de uma

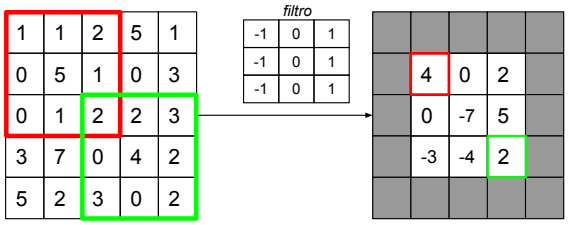

Figura 6.21: Convolução de um sinal 2D por um filtro de tamanho 3x3. Elementos em cinza descartados pelo filtro ultrapassar a borda. De maneira alternativa, podem ser calculados supondo zero elementos fora do mapa de características (*zero padding*)

fatia compartilham entre si seus pesos, fazendo com que a união de suas saídas formem um mapa de uma determinada característica extraída do sinal observado.

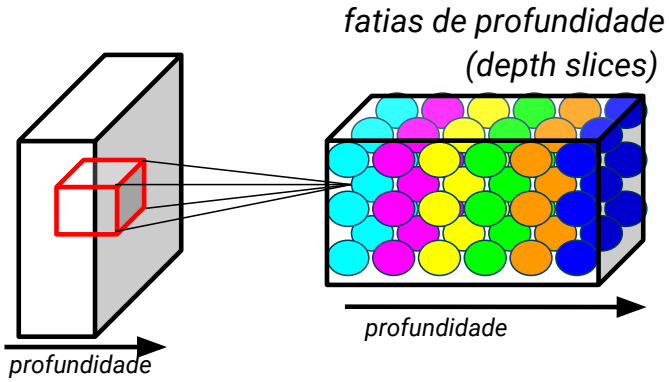

Figura 6.22: Fatiamento dos neurônios de uma CNN: neurônios de uma mesma fatia compartilham pesos, mas aplicam a convolução a diferentes campos receptivos de maneira a cobrir toda o sinal de entrada

Como consequência, juntas as respostas às fatias de uma grade produzem um conjunto de diferentes mapas de características sobre o sinal observado. A Figura 6.23 ilustra um exemplo de conjunto de filtros aprendidos em uma primeira camada de convolução de filtros de $7x7$ elementos.

Sobre o compartilhamento de pesos, cabe ressaltar que além de simular convoluções, é a propriedade responsável por viabilizar a aplicação de redes neurais em dados espacialmente densos, tal como imagens, pela significativa redução de parâmetros na rede. Essa redução acontece em comparação ao modelo padrão de redes alimentadas para frente, já que ao invés de cada neurônio receber como entrada todos os elementos produzidos na camada anterior, em CNNs para a produção de cada mapa de características o número de pesos é limitado ao tamanho do filtro aplicado (altura do filtro$\times$ largura do filtro $\times$ profundidade da camada anterior).

Sobre a profundidade dos filtros, tipicamente a primeira camada de convolução processa uma quantidade pequena de canais de entrada (um para imagens em tons de cinza, três para canais em cores, entre outros), enquanto que as demais camadas passam a processar a quantidade de fatias geradas como mapas de carac-

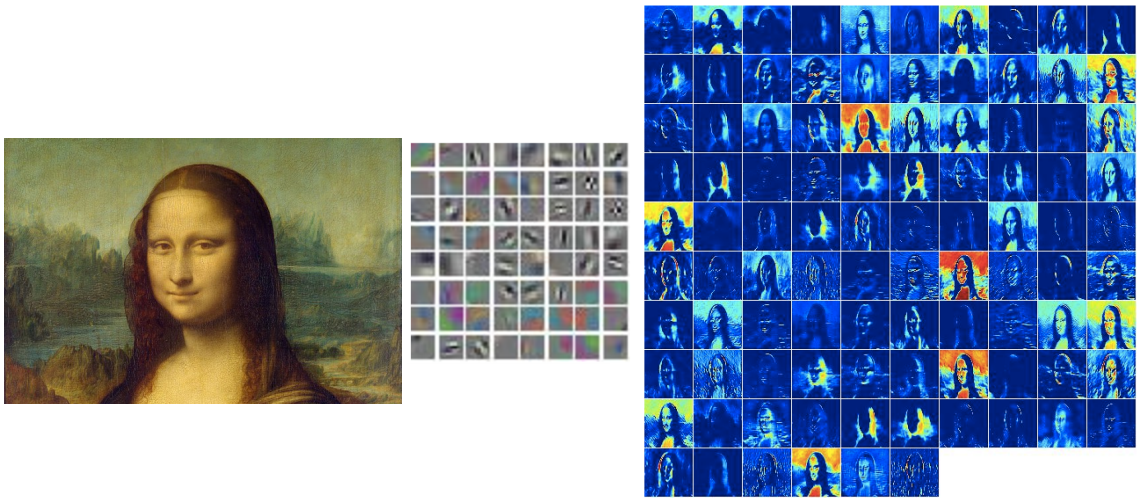

Figura 6.23: Exemplo de filtros aprendidos pela primeira camada de convolução e saída da ativação desses filtros ao apresentar uma face para a rede

terísticas distintos na camada que as alimenta. Ressaltando que a convolução opera ponderando elementos no interior mas também entre fatias.

Ao longo da CNN, é construída uma hierarquia de características, na qual as camadas iniciais extraem características elementares, como bordas e cantos, que são combinadas sucessivamente para a extração de características de mais alto nível semântico nas camadas seguintes.

Os neurônios da camada de convolução se completam com a escolha de uma função de ativação, sendo a rectificação linear (do inglês *Rectified Linear Unit* – ReLU) a mais popular em CNNs recentes. Pautada em observações biológicas, foi proposta por Hahnloser et al. [Hahnloser et al. 2000] e é definida como:

$$f(x) = max(0,x) \tag{27}$$

A ReLU é diferenciável para todo x diferente de zero, sendo sua derivada constante em 1 para $x > 0$, e zero nos demais casos. Essa função tem importantes propriedades tais como não saturar, não apresentar problemas de explosão nem desaparecimento do gradiente, produzir ativações esparsas, além de simplificar os passos de propagação e retro-propagação do gradiente ao diminuir a complexidade dos cálculos envolvidos. Por esses motivos é observado que o uso da ReLU acelera o aprendizado em CNNs profundas, e tem permitido o treinamento de topologias cada vez mais profundas (ou seja, com crescente número de camadas).

Neurônios com ativação ReLU podem vir a 'morrer' e passar a não aprender durante os passos seguintes do treinamento. Esse fenômeno ocorre quando todas as amostras apresentadas a rede são classificadas por tal neurônio com zero. Esse caso induz gradientes nulos e portanto, a parada do aprendizado. Para que um neurônio seja considerado vivo durante o treinamento, basta que ao longo de uma época pelo menos algumas das amostras sejam por ele classificadas com saídas não-zero, pois em consequência induzem sua atualização.

Nas CNNs, seguindo a observação do que acontece em neurônios biológicos, as camadas de convolução são intercaladas com camadas com maior campo receptivo. Para isso, as camadas de agrupamento (do inglês *pooling*) de CNNs realizam uma operação de re-escalonamento do sinal (do inglês *downsampling*) ao longo das dimensões de altura e largura da grade, mantendo o número de fatias inalterado.

O agrupamento de ativações vizinhas nos mapas de características é feito em sub-regiões de tamanho pré-determinado na topologia da rede (como um hiperparâmetro), com ou sem sobreposição entre tais sub-regiões de acordo com o passo entre regiões. Dado um sinal de entrada de dimensões $w_i \times h_i \times d_i$, um passo s e um campo receptivo de tamanho f, após o agrupamento obtêm-se uma saída de dimensão $w_o \times h_o \times d_o$ por:

$$w_o = (w_i - f) \div s + 1 \tag{28}$$

$$h_o = (h_i - f) \div s + 1 \tag{29}$$

$$d_o = d_i \tag{30}$$

Ao mesmo tempo que o agrupamento produz neurônios com maior campo receptivo, impõe uma invariância na localização das características no interior das subregiões agrupadas.

Sobre a operação realizada pelos neurônios das camadas de agrupamento, em CNNs recentes a operação de máximo local tem sido a mais popular (Figura 6.24) embora historicamente outras operações tenham sido usadas, como por exemplo agrupamento *L2-norm* ou média local [LeCun et al. 1998]. Neste caso são chamadas camadas de *max-pooling*.

Ainda que tenham sido inspiradas pela organização do córtex visual de organismos vivos, as CNNs têm se mostrado robustas no processamento de uma variedade de outros sinais descritos na forma matricial, não necessariamente imagens, abrindo seu leque de aplicações. Como exemplos de sinais analisados com sucesso por CNNs podemos citar entre outros: sequências de linguagem natural na forma de vetores 1D; imagens em tons de cinza e espectogramas de áudio na forma de vetores 2D; imagens com profundidade, vídeos e dados volumétricos na forma de vetores 3D, entre outros.

Esse sucesso se dá por conta de quatro ideias chaves da arquitetura CNN descritas nesta seção que tomam proveito das características de sinais naturais: adoção de conexões locais, compartilhamento de pesos, agrupamento (*pooling*) e a construção em múltiplas camadas provendo uma hierarquia de características.

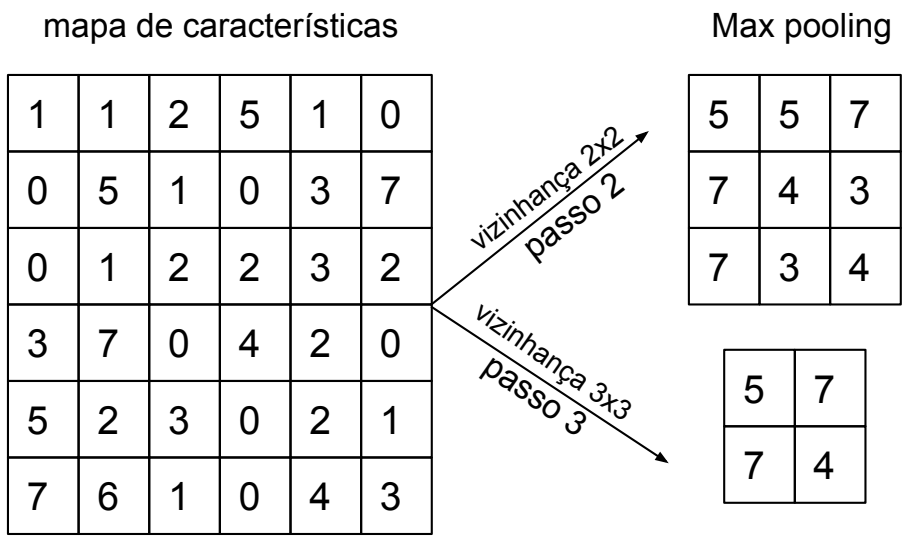

Figura 6.24: Ilustração aplicação de agrupamentos sobre mapa de características de $6x6$, com agrupamento por filtro de $2x2$ e passo de agrupamento 2, e agrupamento por filtro de $3x3$ e passo de agrupamento 3

A última camada de CNNs está associada ao problema que se deseja resolver. Em problemas de regressão, em que as saídas são valores reais, tipicamente é adotada função de perda Euclidiana nos valores emitidos pelos neurônios desta camada.

Em problemas de classificação, a última camada é normalmente definida como uma camada com k neurônios, onde cada um representa uma determinada classe, completamente conectada à anterior. Para predição de K classes mutualmente exclusivas, adota-se tipicamente nessa camada a função de ativação *Softmax*, também chamada de função exponencial normalizada que é definida como:

$$\sigma(Z_c) = \frac{e^{Z_c}}{\sum_{j=1}^{K} e^{Z_j}} \ \texttt{para c} = 1, \cdots, k. \tag{31}$$

onde Z_c representa a transformação linear do neurônio c, ou seja, a combinação linear do vetor de ativações X produzidas pela camada anterior pelo pesos W_c de suas conexões.

A função *Softmax* escala o vetor de saída da última camada da CNN de maneira que seus elementos pertencem ao intervalo $[0,1]$ e que somados sejam 1. Com isso, modela uma distribuição de probabilidade discreta da saída da rede Y pertencer a uma classe c entre as k possíveis saídas da rede, observadas as ativações X recebidas pela última camada:

$$P(X = c|\mathbf{X}) = \frac{e^{\mathbf{X}^\intercal \mathbf{W}_c}}{\sum_{j=1}^{K} e^{\mathbf{X}^\intercal \mathbf{W}_j}} = \tag{32}$$

É importante ainda mencionar que, com a intenção de melhorar a performance das CNNs, durante seu treinamento é comum a adoção do método de *dropout* [Srivastava et al. 2014]. O método de *dropout* busca impedir que um neurônio dependa fortemente de algum outro para fazer suas previsões em um fenômeno de coadaptação.

O *dropout* consiste do desligamento aleatório de neurônios antes de cada passo de propagação de amostras durante o treinamento. Os neurônios desligados não contribuem para o passo de propagação do sinal e desta forma não contribuem durante a retropropagação do erro. Com isso, a cada iteração do treinamento, o mecanismo de sorteio simula uma rede com topologia diferente, e força a quebra de dependências. Assim, faz com que os neurônios aprendam características mais robustas e relevantes a partir das diferentes combinações geradas.

6.3.1. Estudo de caso

Existem diversas arquiteturas de CNNs em um enorme leque de aplicações. Estão a seguir descritas em alto nível algumas delas, selecionadas por questões históricas do desenvolvimento de CNNs ou ainda por sua relevância frente a desafios de classificação de imagens em grandes bases. Mais especificamente, estão descritas as arquiteturas vencedoras das edições 2012 a 2015 do desafio de classificação do *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* [Russakovsky et al. 2015]). São elas:

- LeNet [LeCun et al. 2001]: arquitetura responsável por realizar as primeiras aplicações de sucesso de CNNs como o reconhecimento de dígitos em processamento de cheques. O uso de camadas alternadas simulando células simples e complexas já tinha sido usado anteriormente no modelo denominado neocognitron [Fukushima 1980], que é treinado por uma heurística. A LeNet introduz o treinamento de CNNs pelo algoritmo de retropropagação, e o compartilhamento dos pesos conforme adotado até hoje [LeCun et al. 1989].

  A arquitetura LeNet-5 consiste de uma rede de 7-camadas mais a entrada (duas de convolução alternadas com duas de agrupamento, três completamente conectadas sendo a última a da camada de saída) sendo usada para classificar dígitos em imagens monocromáticas de 32×32 pixels (Figura 6.25).

- AlexNet [Krizhevsky et al. 2012]: arquitetura vencedora do desafio ILSVRC em 2012, no qual obteve 16% de erro em comparação ao segundo lugar com 26% error na classificação *top-5*, chamando a atenção para o potencial das CNNs profundas. Em comparação com a LeNet, a Alexnet é maior, mais profunda, trabalhando com imagens coloridas e em maior resolução, possui maior número de mapas de características empilhados por camada e se diferencia também por adotar a função de ativação ReLU, agrupamento por *max-pooling* além de ser treinada em GPU. A arquitetura AlexNet consiste de cinco camadas de convolução alternadas com agrupamento, seguidas de três camadas completamente conectadas seguidas da camada de saída com função de ativação *Softmax*(Figura 6.26).

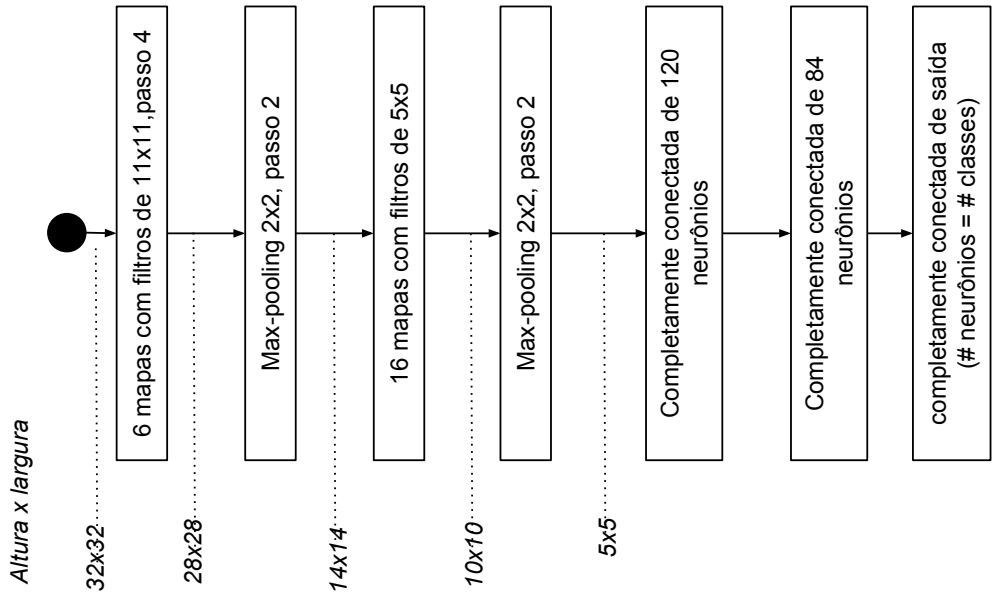

Figura 6.25: Ilustração da arquitetura LeNet-5 [LeCun et al. 2001]

- ZF Net[Zeiler and Fergus 2013]: arquitetura vencedora do ILSVRC 2013 fez importantes melhorias sobre os hiperparâmetros que definem a AlexNet expandindo o tamanho das camadas de convolução intermediárias e reduzindo a passada e o tamanho dos filtros da primeira camada de convolução. Este trabalho também propõe camadas de reversão da rede, na estrutura chamada deconvnet, permitindo a visualização das ativações produzidas pelos mapas de características ao longo da hierarquia de camadas da CNN.

- GoogLeNet [Szegedy et al. 2015a]: arquitetura de 22 camadas vencedora da edição de 2014 do ILSVRC, desenvolvida por pesquisadores do Google (Figura 6.27). Propõe um módulo denominado *Inception* o qual reduziu consideravelmente o número de parâmetros da rede em comparação com a arquitetura AlexNet (de $60M$ para $4M$). O módulo inception combina filtros de diferentes tamanhos criando camadas mistas e mais largas inspirados na proposta de rede dentro da rede [Lin et al. 2013]. Além disso, elimina as camadas completamente conectadas tipicamente usadas no topo de CNNs, e coloca em seu lugar uma camada de agrupamento por média (*average pooling*), com isso eliminando mais parâmetros de treinamento. Melhorias do modelo original da GoogLeNet foram propostas por [Szegedy et al. 2015b, Szegedy et al. 2016]

- VGGNet [Simonyan and Zisserman 2014]: arquitetura vencedora do desafio de localização e segundo lugar no desafio de classificação da edição do ILSVRC 2014. Apontou a profundidade da rede como componente crucial para melhoria de performance. A VGGNet é uma arquitetura extremamente homogênea em dois modelos, com 16 ou 19 camadas de convolução alternadas com agrupamento, seguidas por 3 camadas completamente conectadas no final da rede. Utiliza filtros de 3×3 por toda a rede com agrupamentos em janelas 2×2. Em comparação a GoogLeNet, possui maior demanda por memória e parâmetros

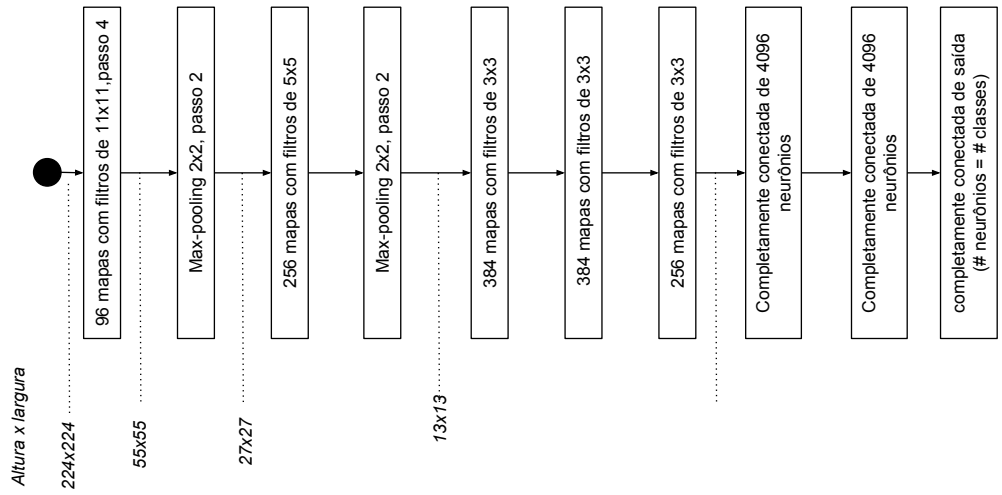

Figura 6.26: Ilustração da arquitetura AlexNet proposta em [Krizhevsky et al. 2012]

(140M), mas muitos provindos das camadas completamente conectadas, o que pode ser substituido sem perda de performance.

- ResNet [He et al. 2015]: arquitetura vencedora da edição ILSVRC 2015. Suas principais contribuições são a inclusão de conexões especiais chamadas de *skip connections* além do uso de normalização de lotes. Juntas permitiram o treinamento de redes consideravelmente mais profundas chegando a 1K camadas. A ResNet é composta por múltiplas de camadas de convolução com filtros 3×3 (com exceção da primeira) e agrupamento 2×2, combinadas com *skip connections*. A Figura 6.28 ilustra a arquitetura ResNet com 34 camadas, sendo a vencedora do ILSVRC composta por 152 camadas. Seguindo a linha do estado da arte até então, as redes ResNet não possuem camadas completamente conectadas no final da rede. Sendo sua versão melhorada descrita no artigo [He et al. 2016].

6.4. Redes Neurais Recorrentes: LSTMs

Conforme apresentado na Seção 6.2.2.3, o aprendizado de dependências de longo prazo em redes neurais recorrentes é afetado pelo fato de que a repetida multiplicação das matrizes Jacobianas para a retro-propagação dos gradientes ao longo das conexões recursivas tendem numericamente a explodir ou a desaparecer.

Diferentes abordagens foram desenvolvidas buscando contornar os problemas no aprendizado de dependências de longo prazo, embora ainda seja considerado um dos principais desafios de pesquisa na área. Entre as abordagens desenvolvidas podemos citar: as *Echo State Networks – ESN* [Jaeger and Haas 2004]; as *Liquid State Machines – LSM* [Maass et al. 2002]; redes com conexões entre instantes de tempo não consecutivos (maiores do que entre t e $t+1$) como o uso de *skip connections* [Lin et al. 1995]; as redes com barreira (*gates*) como nos modelos *long-shot term memory – LSTM* [Hochreiter and Schmidhuber 1997] e *Gated Recurrent Units* (GRU) [Cho et al. 2014]; entre outros. O modelo proposto pelas redes LSTM tem

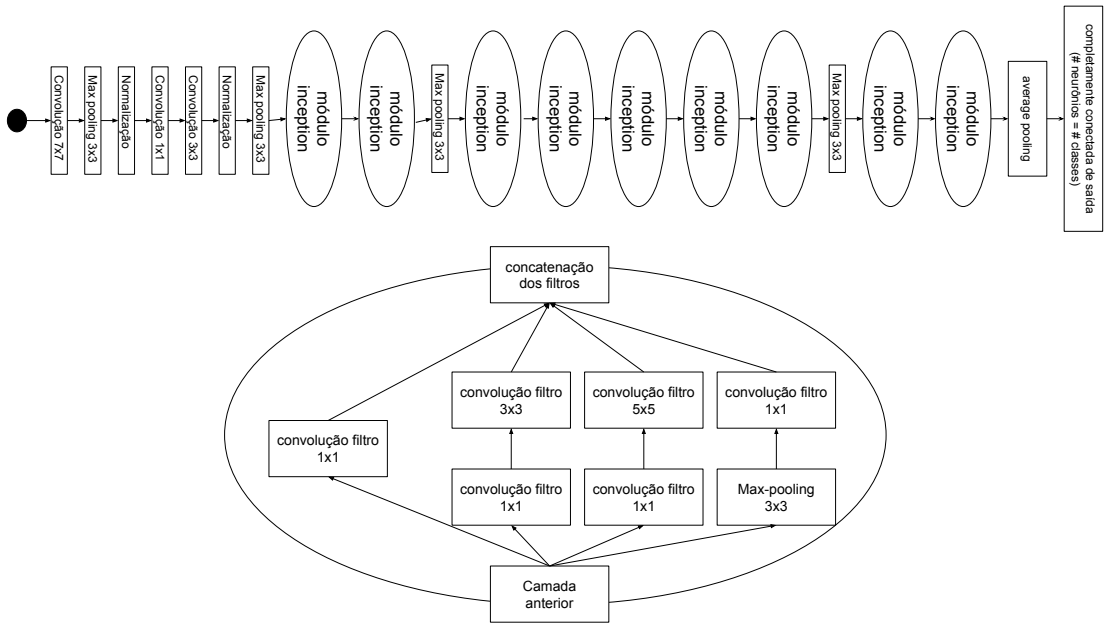

Figura 6.27: Ilustração da Arquitetura GoogLeNet e seu módulo Inception propostos em [Szegedy et al. 2015a]

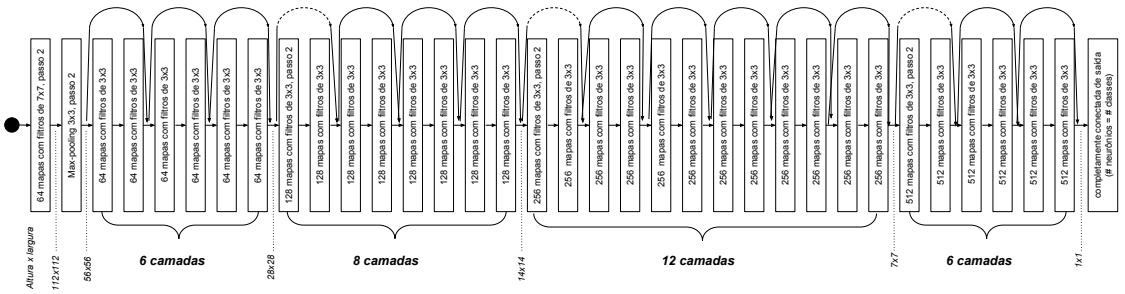

Figura 6.28: Ilustração da arquitetura ResNet de 34 camadas proposta em [He et al. 2015]. Camadas de convolução com filtros em sua maioria de tamanho 3x3, e inclusão das *skip connections* a cada par de camadas de convolução

sido a proposta com maior adoção e em diferentes aplicações e por esse motivo é apresentado nesta seção.

A ideia básica das LSTMs para tratar os problemas de treinamento de dependências de longo prazo de RNNs é baseada na criação de caminhos pelos quais a retropropagação do gradiente possa fluir.

Sua arquitetura é baseada no conceito de células que externamente funcionam como neurônios de uma RNN no sentido que recebem as mesmas entradas (diretas ou recorrentes) e emitem as mesmas saídas de um neurônio da RNN.

A diferença está no funcionamento interno de tais células. Os neurônios de uma RNN tradicional aplicam uma função de ativação não linear sobre a combinação linear de suas entradas (diretas e recorrentes), conforme apresentado na Seção 6.2.2.3. Já nas células da rede LSTM, além da recorrência original de RNN (que

passa a ser considerada uma recorrência externa a célula) possuem também conexões de recorrência internas a própria célula. Essas novas conexões possuem o papel de controlar a leitura e escrita do estado atual, criando um mecanismo dinâmico de controle do fluxo da informação, simulando barreiras (do inglês *gates*) no caminho da retro-alimentação dos estados.

As barreiras em uma LSTM criam conexões de maneira que a rede permite guardar informações durante um longo período de tempo. Além de poder guardar, uma vez que a informação for utilizada, também criam um mecanismo para que a rede possa esquecer uma determinada informação (associando zero ao estado correspondente). A LSTM é capaz de aprender quando o estado pode ser esquecido, portanto reagindo ao contexto da sequência sendo analisada. Assim, a LSTM consegue controlar dinamicamente de acordo com o contexto quando guardar ou esquecer informações mantidas em seus estados.

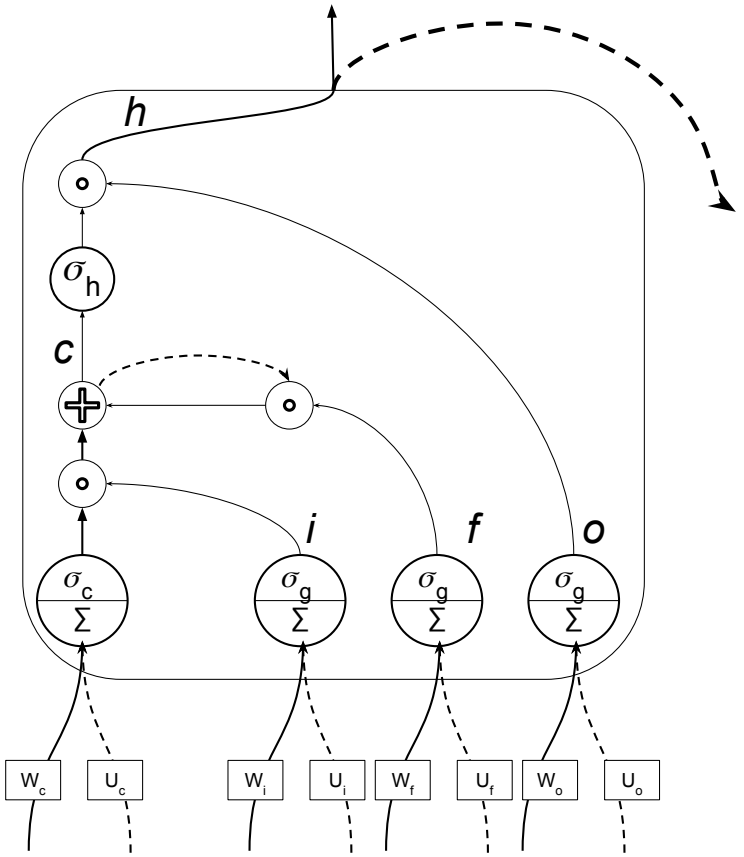

Figura 6.29: Ilustração de uma célula LSTM que mantém as entradas e saídas originais de uma RNN tradicional, mas cria internamente mecanismos de barreira (*gates*) que controlam o fluxo da informação i, f e o, para controle de escrita, limpeza e leitura da célula c

O ponto central da célula LSTM é seu estado $S(t)$, o qual pode fluir através da célula para permitir que a informação trafegue tanto diretamente sem ser alterada, ou ainda permitindo ser alterada de acordo com as conexões de barreira.

Mais especificamente, as conexões de barreira de uma célula LSTM são de três tipos definidas a seguir. Uma barreira i que limita a entrada de informação para a célula, controlando portanto a operação de escrita na célula (do inglês *input gate*). Uma barreira o que limita a saída da célula, controlando a leitura/consulta da informação contida na célula (do inglês *output gate*). Uma barreira f que gerencia a manutenção ou esquecimento de valores na célula, controlando a limpeza da informação contida na célula (do inglês *forget gate*). Sendo cada uma das três controladas por seus próprios pesos aprendidos durante o treinamento, de maneira a reagirem com o contexto (informação de entrada) para controle de suas funções. São definidas por:

$$f(t) = \sigma_g(W_f x(t) + U_f h(t-1) + b_f) \tag{33}$$
$$i(t) = \sigma_g(W_i x(t) + U_i h(t-1) + b_i) \tag{34}$$
$$o(t) = \sigma_g(W_o x(t) + U_o h(t-1) + b_o) \tag{35}$$

onde W_f,W_i,W_o são os pesos e b_f,b_i,b_o seus bias associados as conexões de entrada para as respectivas barreiras f, i e o; enquanto que U_f,U_i,U_o representam os pesos associados a retroalimentação da saída da célula para suas barreiras. As funções de ativação σ_g tipicamente usadas nas barreiras são do tipo função sigmóide (portanto, emitem saída no intervalo $[0,1]$). Com essa formulação, as três barreiras reagem dinamicamente a cada instante t em reação ao sinal recebido como entrada $x(t)$ e também a saída produzida pela célula no instante anterior $h(t-1)$.

O funcionamento da célula c por sua vez é definido por:

$$c(t) = f(t) \circ c(t-1) + i(t) \circ \sigma_c(W_c x(t) + U_c h(t-1) + b_c) \tag{36}$$

onde σ_c é tipicamente uma tangente hiperbólica e isoladamente realiza um processamento equivalente ao de neurônios de camadas escondidas em RNNs tradicionais uma vez que combina as ativações de entradas diretas $x(t)$ ponderadas por pesos W_c e bias b_c com as recorrentes ponderadas pelos pesos U_c. A diferença da LSTM está no fato que o valor produzido como saída de $\sigma_c(t)$ pode ou não ser acumulado ao estado corrente, de acordo com a barreira de entrada $i(t)$ e a manutenção os não do estado da célula ao longo de repetidas iterações é controlada pela barreira f.

Por fim, a saída da rede é controlada pela barreira o usando:

$$h(t) = o(t) \circ \sigma_h(c(t)) \tag{37}$$

onde $h(t)$ representa o vetor de saída e a função de ativação σ_h tanto é definida como uma tangente hiperbólica, como também por $\sigma_h(x) = x$ na variação das LSTM denominadas *peephole LSTM*.

O acréscimo das conexões de retroalimentação que exercem o papel de barreiras faz com que a célula LSTM possua mais parâmetros do que o modelo original RNN (apresentado na Seção 6.2.2.3). Além disso, enquanto no modelo original os pesos de atualização dos estados ocultos são fixos ao longo das iterações, ao aplicar-se a LSTM em uma sequência, a atualização dos estados ocultos ocorre dinamicamente. Isso porque ao longo do tempo tal atualização é condicionada ao contexto por seus

pesos estarem controlados por outras unidades escondidas que são as próprias barreiras.

Com essa modelagem, a LSTM consegue aprender dependências de longo prazo mais facilmente ao produzir o chamado Carrossel de Erro Constante (CEC) pois cria um mecanismo capaz de induzir derivadas de valor constante próximo de 1 ao longo de diversas iterações quando as próprias barreiras são ativadas com valores próximos de 1 fazendo com que a informação seja propagada por mais tempo que em RNNs.

6.5. GPUs

As GPUs foram parcialmente responsáveis pela revolução em Deep Learning. Como apresentado anteriormente, grande parte da matemática envolvida em redes neurais profundas corresponde a cálculos de matrizes. As arquiteturas das GPUs permitem em alguns casos acelerar algumas centenas de vezes a velocidade destes cálculos, tornando viável a resolução de problemas no que se refere ao tempo.

Neste capítulo iremos apresentar resumidamente a arquitetura das GPUs, bem como uma breve introdução em como programa-las no nível mais baixo de sua arquitetura. Embora na maioria das vezes os usuários de deep learning não venham a precisar entrar neste nível, uma compreensão básica da arquitetura pode ajudar nas buscas de otimizações ou na manipulação das bibliotecas, bem como entender de onde vem a aceleração das GPUs.

6.5.1. Breve História

O termo GPU surgiu em 1999, quando a NVIDIA lançou a GeForce 256 e a ATI a Radeon 7500. Estas duas placas gráficas passaram a se denominar processadores ao invés de simples aceleradores, pois implementavam tarefas complexas do pipeline gráfico, incluindo estágios de iluminação, processamento de vértices e pixel. O pipeline gráfico introduzido pelas GPUs recebeu o nome de pipeline de função fixa, pois todos os algoritmos de transformação geométrica, iluminação e rasterização eram implementados diretamente no hardware e manipulados via APIs gráficas. A natureza dos métodos numéricos do pipeline gráfico implicam uma resolução de cálculos de vetores e matrizes.

Em 2001 surge o pipeline gráfico programável, possibilitando que o desenvolvedor possa programar os estágios de vértice e de pixel do pipeline gráfico. Esta programação passou a ser feita por programas denominados de shaders, que eram funções invocadas pelas APIs gráficas em algum dos estágios do pipeline, possibilitando que os mesmos pudessem ser customizados e alterados. O hardware das placas desta época eram compostos por dois conjuntos de processadores, um conjunto de processadores dedicados ao processamento dos vértices e o outro conjunto de processadores de pixel. Neste momento percebeu-se que as GPUs eram poderosos processadores para resolver métodos numéricos não atrelados ao pipeline gráfico, especialmente em casos onde operações matriciais eram intensas e grandes. Em 2006 finalmente foram lançadas as primeiras GPUs com arquiteturas unificadas: os processadores de vértice e pixel passaram a ser um único processador mais genérico e

com capacidade computacional mais abrangente. Desde então, a arquitetura CUDA vem possibilitando que as GPUs possam encapsular milhares de processadores num único dispositivo.

6.5.2. Arquitetura da GPU

A arquitetura da GPU é classificada como um modelo do tipo Single Instruction, Multiple Threads (SIMT). Isto quer dizer que diversas threads executam a mesma instrução a cada ciclo de clock. Além disso, no modelo de programação do CUDA, as thread são organizadas em blocos, formando uma organização lógica das instâncias do kernel. Na parte de hardware, as GPUs são organizadas em multiprocessadores (SM - symmetric multiprocessors) idênticos [4], onde cada um dos SMs é composto por centenas de cores, memórias e registradores. Estes valores dependem da arquitetura e geração da GPU.

A execução das threads envolve o escalonamento das mesmas em dois níveis: no primeiro nível, os blocos de threads são escalonados nos SMs; e, no segundo nível, as threads de cada bloco são escalonadas nos núcleos (cores). O número de SMs pode variar de acordo com o modelo e geração da GPU. Muitas mudanças no projeto de um SM foram feitas desde a primeira versão da GPU a fim de melhorar a performance e o consumo de energia destes processadores.

Há um limite de threads que podem ser alocadas em uma GPU. Atualmente o limite de threads por blocos é de 2048. Embora haja um limite de quantidade de blocos, por ser um número muito grande (2.147.483.647), considera-se este como infinito. Por fim, conceitualmente os blocos são agrupados formando uma grid.

A GPU tem diferentes níveis hierárquicos de memórias, cada um com capacidade e velocidade de acesso diferente e são organizadas da seguinte forma:

- A memória global, que é a memória principal e pode chegar a uma capacidade de 12GB na arquitetura Maxwell. Os dados contidos nesta memória podem ser acessados por qualquer thread de qualquer bloco. Tem uma latência maior que as demais memórias e os dados armazenados podem ser vistos por diferentes kernels . A gerência desta memória, como alocação e desalocação, são feitas pela CPU.

- A memória compartilhada é uma memória de baixa latência e uma velocidade de acesso muito rápida. Cada SM tem seu próprio espaço de memória compartilhada e seu escopo é o do bloco, isto é, quando um bloco é desalocado de um SM, seus dados são apagados.

- A memória local é chamada assim porque o seu âmbito é local para a thread e não por causa de sua localização física. A memória local é off-chip, tornando o acesso a ela tão caro como o acesso à memória global. Esta memória é acessível somente para uma thread específica e os dados persistem apenas durante a execução desta thread.

- A Memória de textura corresponde a uma memória só de leitura e em cache. Ela é otimizada para localidade espacial 2D, permitindo que threads que estão próximas possam usar a mesma operação de leitura para os dados correspondentes, no caso de haver coalescência. Essa memória também é capaz de realizar interpolação

de dados, operação típica na resolução de textura anti- aliasing.

- A memória constante também é armazenada temporariamente em cache e seu acesso custa uma operação de leitura a partir do cache, caso seja evitado um cache miss. É uma pequena memória, com capacidade de 64 KB na Kepler, por exemplo.

A CPU é capaz de ler e escrever na memória global da GPU, sendo que este processo ocorre através do barramento de dados PCI-EXPRESS. Toda a comunicação entre o host (CPU) e o device (GPU) é feita através deste canal.

O escalonamento dentro de cada SM é feito em grupos de 32 threads. Assim, 32 threads consecutivas executam a mesma instrução. Ao fim da execução da última instrução, outras 32 threads consecutivas são escalonadas. Este procedimento de escalonamento é conhecido como warp. Neste sentido recomenda-se instanciar a quantidade de threads em um bloco como sendo múltiplo de 32. O melhor padrão de acesso à memória ocorre quando os dados da memória estão alinhados com as threads de um warp, ou seja, o acesso é coalescente. No procedimento de escalonamento são criados dois índices: um índice é referente ao bloco ao qual uma determinada thread pertence e o outro é o índice da thread dentro do bloco que ela pertence. Como 32 threads de um mesmo warp executam as mesma instruções, uma importante otimização é garantir que estas threads tenham o mesmo caminho de dados. Neste sentido, se há divergência no fluxo de execução do código em uma ou mais thread de um mesmo warp , algumas threads podem ficar ociosas, serializando a execução. Chama-se a este fenômeno de divergência.

Em versões mais antigas do CUDA, só era permitido criar grids com o mesmo kernel , ou seja, não existia a possibilidade de executar kernels concorrentes. Com o lançamento da arquitetura FERMI, passou a ser possível executar mais de um kernel ao mesmo tempo. Contudo, apenas a arquitetura Kepler realmente implementa a execução de kernels concorrentes a nível de hardware. Kernels concorrentes são escalonados internamente pela GPU e um SM pode executar apenas um mesmo kernel. Na prática, isto significa que cada bloco pode ter apenas threads instanciadas de um mesmo kernel e o kernel concorrente é escalonado em outro SM.

No lançamento da arquitetura Kepler foi introduzida a tecnologia Hyper-Q, que permite diferentes programas ou threads de CPU dispararem diferentes kernels em uma mesma GPU. Até então, quando um kernel era disparado, a GPU ficava ocupada executando este kernel e não recebia outros kernels de outros programas ou threads de CPU, sendo necessário um sincronismo por barreira para que uma grid fosse finalizada antes de inicializar outra.

6.5.3. CUDA

Pode-se utilizar a GPU de diversas maneiras diferentes. Em alguns casos serão usadas bibliotecas que se utilizam intensivamente da GPU para executar as suas funções, como é o caso do cuDNN. Em outros casos, o nível de abstração pode ser maior ainda, onde serão usados frontends que por sua vez usam bibliotecas e que por sua vez usam as GPUs, como é o caso do DIGITS. Entretanto, sempre que

for necessário programar diretamente na GPU, é necessário utilizar sua linguagem nativa, portanto CUDA ou OpenCL. Nesta seção estaremos focando em arquiteturas NVIDIA, portanto arquiteturas baseadas em CUDA.

Aplicações desenvolvidas para GPUs envolvem partes de código em CPU e partes de código para GPU, uma vez que as CPUs são responsáveis por gerenciamento das GPUs. Este gerenciamento consiste em alocação de memória, cópia de dados e chamada do kernel. Após a computação da GPU, a CPU copia de volta os dados da memória da GPU para a CPU. A comunicação (processo de cópia) ocorre entre a memória principal da CPU e a memória global da GPU.

A arquitetura atual da GPU PASCAL é capaz de alcançar 11 TFlops em processamento e possui largura de banda capaz de copiar até 720 GB por segundo de dados da CPU para a GPU e vice-versa. Em outras palavras, é possível copiar até 56 bilhões de números ponto flutuante por segundo, o que é uma quantidade de 65 vezes menor que a capacidade de processamento de pontos flutuantes por segundo. Por esta razão, a otimização em minimizar o tráfego de dados de/para a GPU é uma obrigação a fim de atingir o máximo de desempenho. O código de GPU é basicamente uma função, que é chamada pela CPU. Esta função é chamada de kernel e o código da figura 6.30 é um exemplo de “Hello world” em CUDA.

```
2 // GPU code
3 __global__ void the_kernel(void){
4    }
5
6 // CPU Code
7 int main(void) {
8       the_kernel<<<1,1>>>();
9       printf("Hello World!");
10 }
```

Figura 6.30: Código de um programa Hello World em CUDA.

A declaração que indica a função kernel é a declaração ”global”. Esta declaração indica que a função é executada na GPU, sendo chamada pela CPU. Neste exemplo, nenhuma computação está sendo realizada, pois não há sentido em imprimir de forma paralela várias mensagens de Hello world, o exemplo é para efeitos didáticos.

Ao chamar um kernel é necessário indicar quantas threads serão instanciadas. Como já apresentado, há uma organização das threads em blocos e o total de threads instanciadas é dado por: total de threads = quantidade de blocos x threads por blocos. No exemplo do Código da figura , as threads são instanciadas na linha 7 através da sintaxe: KERNEL <1,1>, onde o primeiro parâmetro é o número de blocos e o segundo indica o número de threads por bloco. Estes mesmos parâmetros podem também serem interpretados como: o primeiro parâmetro indica a dimensão do bloco, enquanto o segundo parâmetro descreve a dimensão do bloco.

O Código da figura 6.31 é um exemplo de aplicação que tem o objetivo de somar dois valores, onde o kernel executa uma simples operação de soma na GPU. Embora este exemplo ainda não aborde as questões inerentes ao paralelismo da GPU, pois apenas instancia uma única thread, é um exemplo que apresenta um importante estágio das aplicações de GPU que é a comunicação entre GPU e CPU.

O Código da figura 6.31, em linhas gerais, ilustra todos os estágios que

```
1 // GPU code
2 __global__ void sumKernel (int *a, int *b, int *c) {
3    *c = *a + *b;
4 }
5
6 // CPU Code
7 int main(void) {
8    int a, b, c;
9    int *d_a, *d_b, *d_c;
10   int size = sizeof(int);
11
12   // Allocate space for device
13   cudaMalloc((void **)&d_a, size);
14   cudaMalloc((void **)&d_b, size);
15   cudaMalloc((void **)&d_c, size);
16
17   // Give some initial values
18   a = 10; b = 20;
19
20   // CPU -> GPU
21   cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
22   cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
23
24   // kernel execution: 1 thread
25   sumKernel<<<1,1>>>(d_a, d_b, d_c);
26   // GPU -> CPU
27   cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
28   // Clean memory
29   cudaFree(d_a);cudaFree(d_b); cudaFree(d_c);
30 }
```

Figura 6.31: código completo de um programa simples para somar dois valores na GPU

envolvem uma aplicação:

a. Definição do kernel (linhas 2 a 4);

b. Declaração das variáveis de GPU (linha 9). Como a GPU tem espaços de memória diferente é necessário criar e alocar variáveis em espaços diferentes da CPU;

c. Alocação da memória da GPU utilizada no kernel. A gerência da memória da GPU é de responsabilidade da CPU. Portanto, tanto neste passo (alocação) quanto no passo de desalocação, os comandos são executados pela CPU, conforme linhas 13 à 15.O cudaMalloc tomará conta da referência para as variáveis declaradas em (b);

d. Enviar dados da CPU para a GPU (linhas 21 e 22): função cudaMemcpy irá transferir os dados da memória de CPU pra a GPU. Note que o último argumento é usado para dizer em que direção ocorre a cópia, neste caso DeviceToHost;

e. Execução do Kernel (linha 25): o kernel será executado criando um número total de threads equivalente a blocos x números de threads por bloco;

f. Enviar os dados de volta da GPU para a CPU (linha 29), usando novamente a função cudaMemcpy;

g. Liberar a memória da GPU (linha 29);

Como mencionado anteriormente, GPU computing é uma computação híbrida e deve tratar com pelo menos duas arquiteturas de hardware. Neste sentido, cada hardware tem sua própria memória e sempre que se requer enviar uma tarefa de um dispositivo para outro é necessário fazer a transferência de dados pelas memórias. Transferir dados da CPU para a GPU é normalmente um dos gargalos dos programas e deve ser minimizado. Num futuro próximo pretende-se que os sistemas tenham apenas uma única memória, evitando a tarefa de transferência.

Os kernels de CUDA são lançados de forma assíncrona em relação ao host. Isto significa que uma vez chamado o kernel, a CPU está livre para continuar processando o que vier na sequência. Entretanto, se logo após chamar o kernel a CPU requerer resultados desta chamada, deve-se inserir um ponto de sincronismo, de forma a esperar que a GPU entregue os resultados antes de usá-los. Esta tarefa é feita usando o comando cudaDeviceSynchronize. Note que no código da figura 6.31 não há este ponto de sincronismo. Isto ocorre porque a função cudaMemcpy já inclui uma sincronização dentro de sua implementação.

Finalmente, vamos incluir paralelismo no exemplo. Obviamente somar dois números não é uma tarefa paralelizável, portanto vamos agora somar 2 vetores de tamanho N. O código da Figura 6.32 mostra um kernel que faz esta tarefa.

```
1 __global__ void Vecadd(int *d_a, int *d_b, int *d_c) {
2    int i = threadIdx.x;
3    d_c[i] = d_a[i] + d_b[i]
```

Figura 6.32: Kernel para somar dois vetores de forma paralela.

Este kernel mostra uma palavra reservada importante em CUDA, chamada threadIdx. Cada thread de um bloco irá receber seu índice individual. A atribuição deste índice é feita pelo escalonador de warps e independentemente do número de threads criados, tem tempo constante. Neste sentido, suponhamos que se desejem criar N threads. De forma paralela, teremos a seguinte execução:

Thread 0: dc[0] = da[0] + db[0]

Thread 1: dc[1] = da[1] + db[1]

...

Thread N: dc[N] = da[N] + db[N]

Se o número de threads é menor que o número de núcleos disponíveis em cada bloco em execução, podemos dizer que a soma será feita numa única passada na GPU. Entretanto, se o N for maior, uma fila de threads será criada e haverá listas de threads para serem chamadas em warps distintos. Para somar o vetor de N, uma maneira possível de chamar o kernel seria:

VecAdd < < <1, > > >(da, db, dc);

Esta chamada cria um bloco, sendo que este possui N threads dentro. Enquanto aqui já há um paralelismo ocorrendo, ainda estamos usando um número

limitado de recursos, já que há apenas um bloco sendo usado, o que significa que tudo está recaindo num mesmo SM. Há também outra limitação, que consiste no número máximo de threads que se pode criar por bloco, sendo em algumas arquiteturas 1024 e em outras (mais modernas) 2048. Isto significa que se o vetor tiver mais do que esta quantidade de elementos, este kernel não poderá tratar o vetor. Uma possibilidade seria em cada thread somar mais do que um elemento do vetor. Mesmo assim, ainda estaríamos usando um só SM, portanto não usaríamos todos os recursos da GPU.

Para um uso completo da GPU, é necessário criar também diversos blocos, conforme a chamada de kernel abaixo:

VecAdd < < <K,L> > >(da, db, dc);

Neste caso estamos criando K blocos, cada um composto de L threads. Assim sendo, o tamanho do Grid corresponde a N=L.L, que é o número de elementos do vetor. Para fazer isto, é necessário realizar um pequeno ajuste no kernel, de forma que os blocos sejam usados adequadamente para mapear partes distintas do vetor. O código da Figura 6.32 mostra este kernel.

```
__global__ void Vecadd(int *d_a, int *d_b, int *d_c) {
     int i= threadIdx.x + blockIdx.x * blockDim.x;
     d_c[i] = d_a[i] + d_b[i];
}
```

Figura 6.33: Kernel para somar dois vetores usando mais de um bloco.

O escalonador de warps cria L threads para cada bloco, de forma que cada bloco tem o mesmo conjunto de números threadIdx para suas threads. Entretanto, o escalonador atribui diferentes índices de blocos para cada bloco. Estes índices podem ser compostos com os índices das threads para criar índices únicos para cada elemento do vetor. Enquanto este kernel é mais eficiente que o apresentado anteriormente, ainda há um problema a ser tratado, que é o caso de N não ser múltiplo da tupla K,L. Suponha, por exemplo, que N=101. Neste caso, poderíamos atribuir K=5 e L=21. Porém seriam criados 105 threads e para o caso do blocoIdx.x=4 e threadIdx.x=20 teriamos um índice i=104, que levaria a um elemento do vetor que não existe. Para evitar este problema, é comum colocar um condicional referente ao índice calculado, conforme pode ser visto na figura 6.34

```
1 __global__ void Vecadd(int *d_a, int *d_b, int *d_c) {
2 int i= threadIdx.x + blockIdx.x * blockDim.x;
3    if (i < N)
4       d_c[i] = d_a[i] + d_b[i];
5 }
```

Figura 6.34: Kernel para somar dois vetores usando mais de um bloco.

É importante mencionar que nestes exemplos estamos usando a memória global para armazenar os vetores. Enquanto esta memória é grande e capaz de

ser acessada por qualquer thread de qualquer bloco, a mesma tem uma latência grande e pode demandar bastante tempo para o código. A memória compartilhada é uma alternativa eficiente para acesso a memória. Enquanto uma leitura de um valor na memória global pode levar até 400 ciclos de máquina, um acesso memória compartilhada pode requerer apenas 4 ciclos. A maior restrição presente na memória compartilhada são o seu tamanho pequeno (96Kb na arquitetura Pascal) e pouca persistência (quando um bloco termina de ser executado, os dados da memória compartilhada são apagados, pois um nobo bloco irá entrar no SM). Entender como funciona e como desenvolver programas com esta memória está além do propósito deste texto, mas o leitor interessado pode ler mais em [Clua and Zamith 2015]

6.6. Ferramentas

Avanços em ferramentas e bibliotecas para o desenvolvimento de redes neurais profundas permitem a engenheiros, cientistas e entusiastas explorar soluções para diferentes aplicações na área de Aprendizado de Máquina, tais como classificação de imagens e vídeo, processamento natural de linguagem e reconhecimento de áudio. Essas ferramentas permitem que usuários treinem, desenvolvam e testem redes neurais profundas utilizando todo poder computacional proporcionado pelas GPUs. As bibliotecas de redes neurais profundas mais recentes apresentam uma forma de interação em alto nível onde o usuário deve se preocupar apenas com a modelagem da rede, sendo transparentes ao usuário as etapas de mais baixo nível de programação e otimização computacional. Entre as bibliotecas mais populares encontram-se: Caffe, CNTK, Tensor Flow, Torch e Theano.

Durante o minicurso será apresentada uma introdução à biblioteca Caffe [caf] e um estudo de caso será desenvolvido usando a ferramenta Digits NVidia DIGITS [dig]. O material da parte experimental está disponível em:

`http://www2.ic.uff.br/~gpu/learn-gpu-computing/deep-learning/`

6.7. Outras considerações

Este capítulo procurou apresentar conceitos fundamentais ao aprendizado profundo, sem a pretensão de cobrir toda a área. Alguns temas importantes não foram apresentados ou detalhados, tais como técnicas de aprendizado não-supervisionado, aprendizado por reforço, transferência de conhecimento, mecanismos de atenção, redes generativas adversárias, dentre outros. Tal diversidade ilustra a abrangência de inúmeras pesquisas na área e a importância que a mesma possui na indústria da tecnologia. Embora haja uma grande proliferação de aplicações de redes profundas nos últimos anos, boa parte da fundamentação vem sendo desenvolvida já há alguns anos e deve ser consultada como embasamento e fonte de inspiração para novas soluções. Dentro de sua limitação, este texto buscou ser um convite inicial ao leitor que deseja se iniciar por tais investigações.

References

[caf] Caffenet. `http://caffe.berkeleyvision.org/`. Último acesso em 10/4/2017.

[ima] Imagenet. `http://www.image-net.org/`. Último acesso em 10/4/2017.

[dig] Página oficial do projeto digits da nvidia. `https://developer.nvidia.com`. Digits, 2016. Último acesso em 10/7/2016.

[Bengio et al. 1994] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166.

[Bowman et al. 2016] Bowman, S. R., Vilnis, L., Vinyals, O., Dai, A. M., Jozefowicz, R., and Bengio, S. (2016). Generating sentences from a continuous space.

[Britz et al. 2017] Britz, D., Goldie, A., Luong, T., and Le, Q. (2017). Massive exploration of neural machine translation architectures. *arXiv*.

[Chan et al. 2016] Chan, W., Jaitly, N., Le, Q. V., and Vinyals, O. (2016). Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *ICASSP*.

[Cho et al. 2014] Cho, K., van Merrienboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259.

[Ciresan et al. 2013] Ciresan, D. C., Giusti, A., Gambardella, L. M., and Schmidhuber, J. (2013). Mitosis detection in breast cancer histology images with deep neural networks. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2013 - 16th International Conference, Nagoya, Japan, September 22-26, 2013, Proceedings, Part II*, pages 411–418.

[Ciresan et al. 2011a] Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2011a). Convolutional neural network committees for handwritten character classification. In *2011 International Conference on Document Analysis and Recognition, ICDAR 2011, Beijing, China, September 18-21, 2011*, pages 1135–1139.

[Ciresan et al. 2011b] Ciresan, D. C., Meier, U., Masci, J., and Schmidhuber, J. (2011b). A committee of neural networks for traffic sign classification. In *The 2011 International Joint Conference on Neural Networks, IJCNN 2011, San Jose, California, USA, July 31 - August 5, 2011*, pages 1918–1921.

[Ciresan et al. 2012] Ciresan, D. C., Meier, U., and Schmidhuber, J. (2012). Transfer learning for latin and chinese characters with deep neural networks. In *The 2012 International Joint Conference on Neural Networks (IJCNN), Brisbane, Australia, June 10-15, 2012*, pages 1–6.

[Clua and Zamith 2015] Clua, E. and Zamith, M. (2015). Programming in cuda for kepler and maxwell architecture. In *Revista de Informática Teórica e Aplicada*, volume 22, pages 34–42.

[Couprie et al. 2013] Couprie, C., Farabet, C., Najman, L., and LeCun, Y. (2013). Indoor semantic segmentation using depth information. In *International Conference on Learning Representations (ICLR2013)*.

[Dahl et al. 2017] Dahl, R., Norouzi, M., and Shlens, J. (2017). Pixel recursive super resolution. *arXiv*.

[Dumoulin et al. 2017] Dumoulin, V., Shlens, J., and Kudlur, M. (2017). A learned representation for artistic style. *ICLR*.

[Eslami et al. 2016] Eslami, S. M. A., Heess, N., Weber, T., Tassa, Y., Szepesvari, D., Kavukcuoglu, K., and Hinton, G. E. (2016). Attend, infer, repeat: Fast scene understanding with generative models.

[Esteva et al. 2017] Esteva, A., Kuprel, B., Novoa, R. A., Ko, J., Swetter, S. M., Blau, H. M., and Thrun, S. (2017). Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115–118.

[Fukushima 1980] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202.

[Giusti et al. 2016] Giusti, A., Guzzi, J., Ciresan, D. C., He, F., Rodriguez, J. P., Fontana, F., Faessler, M., Forster, C., Schmidhuber, J., Caro, G. D., Scaramuzza, D., and Gambardella, L. M. (2016). A machine learning approach to visual perception of forest trails for mobile robots. *IEEE Robotics and Automation Letters*, 1(2):661–667.

[Gulshan et al. 2016] Gulshan, V., Peng, L., Coram, M., Stumpe, M. C., Wu, D., Narayanaswamy, A., Venugopalan, S., Widner, K., Madams, T., Cuadros, J., Kim, R., Raman, R., Nelson, P. Q., Mega, J., and Webster, D. (2016). Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *JAMA*.

[Gupta et al. 2017] Gupta, S., Davidson, J., Levine, S., Sukthankar, R., and Malik, J. (2017). Cognitive mapping and planning for visual navigation.

[Hahnloser et al. 2000] Hahnloser, R. H., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J., and Seung, H. S. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951.

[He et al. 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

[He et al. 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Identity mappings in deep residual networks. *CoRR*, abs/1603.05027.

[Heigold et al. 2016] Heigold, G., Moreno, I., Bengio, S., and Shazeer, N. M. (2016). End-to-end text-dependent speaker verification. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

[Hochreiter and Schmidhuber 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[Hubel and Wiesel 1959] Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, 148:574–591.

[Hubel and Wiesel 1968] Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, 195:215–243.

[Hubel and Wiesel 2005] Hubel, M. and Wiesel, T. N. (2005). *Brain and Visual Perception*. Oxford Univeristy Press.

[Jaeger and Haas 2004] Jaeger, H. and Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, pages 78–80.

[Jaques et al. 2016] Jaques, N., Gu, S., Turner, R. E., and Eck, D. (2016). Generating music by fine-tuning recurrent neural networks with reinforcement learning. In *Deep Reinforcement Learning Workshop, NIPS*.

[Kaiser and Sutskever 2016] Kaiser, L. and Sutskever, I. (2016). Neural gpus learn algorithms. In *International Conference on Learning Representations*.

[Kannan et al. 2016] Kannan, A., Kurach, K., Ravi, S., Kaufman, T., Miklos, B., Corrado, G., Tomkins, A., Lukacs, L., Ganea, M., Young, P., and Ramavajjala, V. (2016). Smart reply: Automated response suggestion for email. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD) (2016)*.

[Krizhevsky et al. 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[LeCun et al. 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.

[LeCun et al. 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[LeCun et al. 2001] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (2001). Gradient-based learning applied to document recognition. In *Intelligent Signal Processing*, pages 306–351. IEEE Press.

[Levine et al. 2016] Levine, S., Sampedro, P. P., Krizhevsky, A., and Quillen, D. (2016). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection.

[Lin et al. 2013] Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *CoRR*, abs/1312.4400.

[Lin et al. 1995] Lin, T., Horne, B. G., Tiňo, P., and Giles, C. L. (1995). Learning long-term dependencies is not as difficult with narx recurrent neural networks. Technical report, College Park, MD, USA.

[Loos et al. 2017] Loos, S., Irving, G., Szegedy, C., and Kaliszyk, C. (2017). Deep network guided proof search.

[Maass et al. 2002] Maass, W., Natschläger, T., and Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Comput.*, 14(11):2531–2560.

[Mnih et al. 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*.

[Odena et al. 2016] Odena, A., Olah, C., and Shlens, J. (2016). Conditional image synthesis with auxiliary classifier gans. *arXiv*.

[Papandreou et al. 2017] Papandreou, G., Zhu, T., Kanazawa, N., Toshev, A., Tompson, J., Bregler, C., and Murphy, K. (2017). Towards accurate multi-person pose estimation in the wild.

[Pascanu et al. 2013] Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1310–1318.

[Rosenblatt 1961] Rosenblatt, F. (1961). Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, DTIC Document.

[Russakovsky et al. 2015] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.

[Sermanet et al. 2012] Sermanet, P., Chintala, S., and LeCun, Y. (2012). Convolutional neural networks applied to house numbers digit classification. In *International Conference on Pattern Recognition (ICPR 2012)*.

[Sermanet et al. 2013] Sermanet, P., Kavukcuoglu, K., Chintala, S., and LeCun, Y. (2013). Pedestrian detection with unsupervised multi-stage feature learning. In *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR'13)*. IEEE. Video part 1; Video part 2.

[Shao et al. 2017] Shao, L., Gouws, S., Britz, D., Goldie, A., Strope, B., and Kurzweil, R. (2017). Generating long and diverse responses with neural conversation models. *arXiv*.

[Silver et al. 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503.

[Simonyan and Zisserman 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

[Srivastava et al. 2014] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

[Szegedy et al. 2016] Szegedy, C., Ioffe, S., and Vanhoucke, V. (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261.

[Szegedy et al. 2015a] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015a). Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*.

[Szegedy et al. 2015b] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015b). Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567.

[Tompson et al. 2016] Tompson, J., Schlachter, K., Sprechmann, P., and Perlin, K. (2016). Accelerating eulerian fluid simulation with convolutional networks. *arXiv*.

[Tompson et al. 2014] Tompson, J., Stein, M., Lecun, Y., and Perlin, K. (2014). Real-time continuous pose recovery of human hands using convolutional networks. *ACM Trans. Graph.*, 33(5):169:1–169:10.

[Villegas et al. 2017] Villegas, R., Yang, J., Hong, S., Lin, X., and Lee, H. (2017). Decomposing motion and content for natural video sequence prediction.

[Yan et al. 2016] Yan, X., Yang, J., Yumer, E., Guo, Y., and Lee, H. (2016). Perspective transformer nets: Learning single-view 3d object reconstruction without 3d supervision.

[Zeiler and Fergus 2013] Zeiler, M. D. and Fergus, R. (2013). Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901.