Capítulo 3

Introdução à Aprendizagem Profunda

Eduardo Bezerra (CEFET/RJ)

Abstract

Deep Learning is a subfield of Machine Learning investigating techniques to simulate the behavior of the human brain in tasks such as visual recognition, speech recognition and natural language processing. Deep learning algorithms aim at producing high-level hierarchical representations of the input data, by sequential processing layers in an artificial neural network. This Chapter presents an introduction to deep learning techniques, algorithms, and main network architectures, as well as examples of applications in areas such as computational vision and natural language processing.

Resumo

Aprendizagem Profunda é uma subárea de Aprendizagem de Máquina que investiga técnicas para simular o comportamento do cérebro humano em tarefas como reconhecimento visual, reconhecimento de fala e processamento de linguagem natural. Algoritmos de aprendizagem profunda objetivam produzir representações hierárquicas de alto nível dos dados de entrada, por meio de camadas de processamento sequencial em uma rede neural artificial. Esse Capítulo apresenta uma introdução às técnicas, algoritmos e principais arquiteturas de rede para aprendizagem profunda, assim como exemplos de aplicações em áreas como visão computacional e processamento de linguagem natural.

3.1. Considerações Iniciais

Em aprendizado de máquina, uma característica (*feature*) é uma propriedade mensurável de um fenômeno de interesse [1]. Nesse contexto, são úteis transformações de uma representação de dados para outra em um nível conceitual mais alto e que possa ser efetivamente aproveitada por algoritmos de aprendizado. Tradicionalmente, algoritmos de aprendizado de máquina pressupõem que o conjunto de dados a ser usado no treinamento foi previamente construído por meio de um processo de engenharia de características. Nesse processo, as características (i.e., atributos) componentes do conjunto de dados são

manualmente selecionadas por um ser humano. A escolha de características ao mesmo tempo relevantes e discriminativas é fundamental para a geração de modelos de aprendizado adequados. Entretanto, esse processo está sujeito a erros quando depende da existência de um especialista do domínio em questão, que possa selecionar adequadamente as características para compor o conjunto de dados de treinamento. Por meio de técnicas de *aprendizado de representação* [2], é possível que o próprio algoritmo de aprendizado identifique automaticamente um subconjunto de *características* adequado.

Aprendizagem profunda é o termo usado para denotar o problema de treinar redes neurais artificiais que realizam o aprendizado de características de forma hierárquica, de tal forma que características nos níveis mais altos da hierarquia sejam formadas pela combinação de características de mais baixo nível [3, 4, 5, 6, 7, 8]. Muitos problemas práticos têm sido resolvidos por redes neurais compostas de uma única camada intermediária [9]. De fato, o *Teorema da Aproximação Universal* [10] garante que redes neurais com uma única camada oculta são *aproximadores universais*, i.e., são estruturas capazes de aproximar qualquer função contínua, contanto que se possa definir a quantidade suficiente de neurônios nessa única camada intermediária.

3.1.1. Motivações para Redes Profundas

Uma das motivações para a investigação de redes neurais artificiais compostas de muitas camadas veio da neurociência, mais especificamente do estudo da parte do cérebro denominada *córtex visual* [11]. Sabe-se que, quando algum estímulo visual chega à retina, o sinal correspondente percorre uma sequência de regiões do cérebro, e que cada uma dessas regiões é responsável por identificar características específicas na imagem correspondente ao estímulo visual. Em particular, neurônios das regiões iniciais são responsáveis por detectar formas geométricas simples na imagem, tais como cantos e bordas, enquanto que neurônios nas regiões finais têm a atribuição de detectar formas gráficas mais complexas, compostas das formas gráficas simples detectadas por regiões anteriores. Dessa forma, cada região de neurônios (que é análoga ao conceito de *camada* em redes neurais artificiais) combina padrões detectados pela região imediatamente anterior para formar características mais complexas. Esse processo se repete através das regiões até que os neurônios da região final têm a atribuição de detectar características de mais alto nível de abstração, tais como faces ou objetos específicos [2].

Há outra motivação, dessa vez teórica, para o estudo de redes profundas. Em [12], usando funções lógicas (*boolean functions*) como base para seu estudo, os autores demonstram que é possível representar qualquer função lógica por meio de um circuito lógico de uma única camada, embora essa representação possa requerer uma quantidade de unidades na camada intermediária que cresce exponencialmente com o tamanho da entrada. Eles também demonstraram que, ao permitir que o circuito cresça em quantidade de camadas intermediárias, essas mesmas funções lógicas problemáticas poderiam ser representadas por uma quantidade de unidades ocultas que cresce polinomialmente com o tamanho da entrada.

3.1.2. Visão Geral do Capítulo

A Seção 3.2 apresenta (de forma sucinta) diversos conceitos introdutórios envolvidos na construção de redes neurais e necessários ao entendimento do restante do conteúdo. A seguir, descrevemos três classes de redes neurais populares em aprendizagem profunda, a saber, autocodificadoras (Seção 3.3), redes convolucionais (Seção 3.4) e redes recorrentes (Seção 3.5). Essas classes de redes estão na vanguarda das aplicações mais recentes e do sucesso da aprendizagem profunda. Na Seção 3.6, descrevemos procedimentos utilizados durante o treinamento redes neurais profundas. Finalmente, na Seção 3.7, apresentamos as considerações finais.

3.2. Conceitos Básicos de Redes Neurais Artificiais

A arquitetura de uma rede neural diz respeito ao modo pelo qual suas unidades de processamento estão conectadas. Essa arquitetura influencia diretamente nos problemas que uma Rede Neural Artificial (RNA) pode resolver e na forma de realizar seu treinamento. Nesta Seção, apresentamos detalhes acerca da arquitetura básica de uma RNA, assim como descrevemos sucintamente aspectos do treinamento.

3.2.1. Arquitetura Básica

A transmissão do sinal de um neurônio a outro no cérebro é um processo químico complexo, no qual substâncias específicas são liberadas pelo neurônio transmissor. O efeito é um aumento ou uma queda no potencial elétrico no corpo da célula receptora. Se este potencial alcançar o limite de ativação da célula, um pulso ou uma ação de potência e duração fixa é enviado para outros neurônios. Diz-se então que o neurônio está *ativo*. De forma semelhante à sua contrapartida biológica, uma RNA possui um sistema de *neurônios artificiais* (também conhecidas como *unidades de processamento* ou simplesmente *unidades*). Cada unidade realiza uma computação baseada nas demais unidades com as quais está conectada.

Os neurônios de uma RNA são organizados em camadas, com conexões entre neurônios de camadas consecutivas. As conexões entre unidades são ponderadas por valores reais denominados *pesos*. A RNA mais simples possível contém uma única camada, composta por um único neurônio. Redes dessa natureza são bastante limitadas, porque resolvem apenas processos de decisão binários (i.e., nos quais há duas saídas possíveis) e linearmente separáveis (e.g., funções booleanas AND e OR). Por outro lado, é possível construir redes mais complexas (i.e., capazes de modelar processos de decisão não linearmente separáveis) por meio de um procedimento de composição de blocos de computação mais simples organizados em camadas.

A Figura 3.1 ilustra esquematicamente os elementos da arquitetura mais comum de RNA, uma *Rede Alimentada Adiante* (*Feedforward Neural Network*), ao mesmo tempo em que apresenta parte da notação usada neste Capítulo. Nessa arquitetura, a rede forma um grafo direcionado, no qual os vértices representam os neurônios, e as arestas representam os pesos das conexões. A camada que recebe os dados é chamada *camada de entrada*, e a que produz o resultado final da computação é chamada *camada de saída*. Entre as camadas de entrada e saída, pode haver uma sequência de L camadas, onde ocorre o processamento interno da rede. Cada elemento dessa sequência é chamado de *camada*

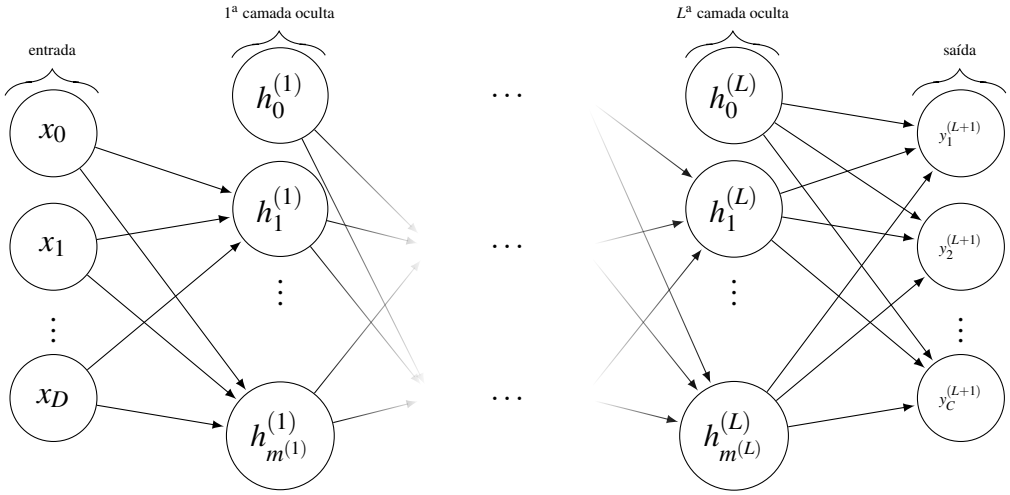


Figura 3.1. Esquema de uma RNA com $(L+1)$ camadas. Há D unidades na camada de entrada e C unidades na saída. A l-ésima camada oculta contém $m^{(l)}$ unidades.

oculta. Após o seu treinamento (Seção 3.2.2), a rede é capaz de realizar um mapeamento de vetores no espaço D-dimensional para vetores no espaço C-dimensional.

Embora existam muitas convenções diferentes para declarar o número real de camadas em uma rede multicamada, neste Capítulo vamos considerar a quantidade de conjuntos distintos de pesos treináveis como o número de camadas de uma RNA. Por exemplo, considerando a Figura 3.1, quando $L = 2$, temos uma RNA de duas camadas de pesos: aqueles que ligam as entradas à camada oculta, e os que ligam a saída da camada oculta com as unidades da camada de saída.

Cada neurônio de uma camada oculta recebe um vetor $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ como entrada e computa uma média ponderada pelos componentes de outro vetor de *pesos* associados $\mathbf{w} = (w_1, w_2, \ldots, w_N)$, em que cada componente pode ser interpretado como a força da conexão correspondente. É comum também considerar uma parcela adicional nessa média ponderada, correspondente ao denominado *viés* (*bias*), com peso constante e igual a $+1$ (correspondente a cada unidade h_0^l, $1 \leq l \leq L$ na Figura 3.1), cujo propósito é permitir que a rede neural melhor se adapte à função subjacente aos dados de entrada. Ao valor resultante dessa computação dá-se o nome de *pré-ativação* do neurônio (ver Eq. 1).

$$a(\mathbf{x}) = b + \sum_i w_i x_i = \mathbf{b} + \mathbf{w}^T \mathbf{x} \tag{1}$$

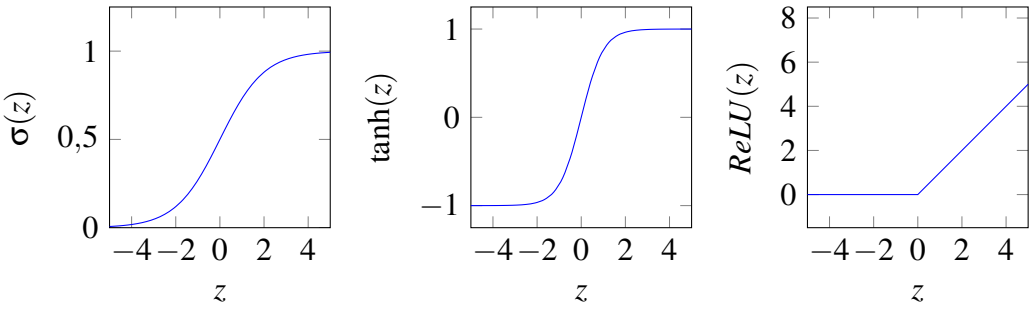
Após computar $a(\mathbf{x})$, o neurônio aplica uma transformação não linear sobre esse valor por meio de uma *função de ativação*, $g(\cdot)$. A função composta $h = g \circ a$ produz o que se chama de *ativação do neurônio* (ver Eq. 2).

$$y = h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i) \tag{2}$$

Há diferentes alternativas para a função de ativação g, dependendo da tarefa em questão. Três dessas funções são a *sigmoide logística* (Eq. 3), a *sigmoide hiperbólica tangente* (Eq. 4) e a *retificada linear* (Eq. 5).

$$g(z) = \frac{1}{1+e^{-z}} \quad (3) \qquad g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \quad (4) \qquad g(z) = \max(0,z) \quad (5)$$

A Figura 3.2 apresenta os gráficos dessas funções. Repare que, com exceção da ReLU em $z = 0$, todas são diferenciáveis e estritamente monotônicas. Repare também que as funções sigmoides possuem assíntotas que as limitam inferior e superiormente.



(a) Logística. (b) Hiperbólica tangente. (c) Retificada linear.

Figura 3.2. Funções de ativação comumente utilizadas.

As equações previamente apresentadas para a pré-ativação e para a ativação de uma única unidade podem ser estendidas para o caso em que há L camadas ocultas, com vários neurônios em cada camada. Em primeiro lugar, em vez de pensar na pré-ativação de um único neurônio, podemos considerar a pré-ativação de toda uma camada oculta, conforme a Eq. 6. Nessa equação, que vale para todo $k > 0$, o sobrescrito k denota a k-ésima camada intermediária, e o uso do negrito indica que a Eq. 1 é aplicada a cada neurônio dessa camada. Explicação análoga pode ser apresentada para a Eq. 7. Além disso, $\mathbf{W}^{(k)}$ é a matriz de pesos das conexões entre os neurônios das camadas $k-1$ e k. Finalmente, considere que, se $k = 0$, então $\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{x}$.

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x}) \quad (6)$$

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \quad (7)$$

Como um todo, podemos considerar que, dado um vetor $\mathbf{x}$ de entrada, a RNA computa uma função $f(\mathbf{x})$ tal que $f(\mathbf{x}) : \mathbb{R}^D \rightarrow \mathbb{R}^C$, conforme a Eq. 8.

$$f(\mathbf{x}) = \mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) \quad (8)$$

Na Eq. 8, $\mathbf{o}(\cdot)$ representa a função de ativação usada para as unidades da camada de saída. Quando o propósito de uma RNA é realizar a tarefa de classificação, é comum selecionar para $\mathbf{o}(\cdot)$ a função denominada `softmax`, que permite interpretar os valores da camada de saída como probabilidades posteriores. O valor produzido pela função softmax para a i-ésima unidade da camada de saída é dada pela Eq. 9.

$$o\left(a_i^{(L+1)}\right) = \frac{e^{a_i^{(L+1)}}}{\sum_{j=1}^{C} e^{a_j^{(L+1)}}} \tag{9}$$

As saídas das unidades $a_i^{(L+1)}$, $1 \leq i \leq C$, podem ser realmente interpretadas como valores de probabilidade, posto que pertencem ao intervalo $[0,1]$ e sua soma é igual a 1.

3.2.2. Treinamento Supervisionado

No aprendizado supervisionado, considera-se a disponibilidade de um conjunto de treinamento da forma $\{(\mathbf{x}^{(t)}, \mathbf{y}^{(t)}) : 1 \leq t \leq T\}$, em que $\mathbf{x}^{(t)}$ é o vetor de características está associado ao componente $\mathbf{y}^{(t)}$. Considera-se que os elementos do conjunto de treinamento são independentes e identicamente distribuídos. No contexto de redes neurais, esse aprendizado (ou treinamento) supervisionado corresponde a utilizar o conjunto de treinamento para ajustar os valores dos *parâmetros* da rede, de tal forma que o *erro de treinamento* (*training error*) seja minimizado. Dessa forma, o treinamento de uma rede neural é convertido em um *problema de otimização*, cujo objetivo é minimizar o erro cometido pela rede, quando considerados todos os exemplos de treinamento.

Com o propósito de medir o quanto a função inferida pela rede (Eq. 8) se ajusta aos exemplos de treinamento, o problema de otimização é definido para minimizar uma *função de custo* (*cost function*), $J(f(\mathbf{x}^{(t)};\theta), \mathbf{y}^{(t)})$, que mede a diferença (ou *erro*) que a rede comete relativamente ao exemplo $\mathbf{x}^{(t)}$. O uso de θ na definição dessa função denota a dependência de J com relação ao vetor de parâmetros θ da RNA. Duas alternativas para a escolha da função J são a *soma dos erros quadrados* (*mean squared sum*) e a *entropia cruzada* (*cross entropy*). A escolha da função de custo adequada depende da tarefa em particular. Detalhes acerca das propriedades de cada uma dessas funções podem ser encontrados em [1]. A Eq. 10 corresponde ao custo relativo ao conjunto de treinamento como um todo.

$$\mathbf{J}(\theta) = \underbrace{\frac{1}{T}\sum_{t=1}^{T} J(f(\mathbf{x}^{(t)};\theta), \mathbf{y}^{(t)})}_{\text{custo dos dados}} + \underbrace{\frac{\lambda}{2T}\sum_{k}\sum_{l} W_{k,l}^2}_{\text{custo de regularização}} \tag{10}$$

Repare que a segunda parcela da Eq. 10 contém a soma dos quadrados de todos os pesos da RNA. Nessa parcela, o parâmetro λ, outro hiperparâmetro da RNA, controla a importância relativa entre as duas parcelas da função $\mathbf{J}(\theta)$. Visto que o propósito da otimização é minimizar $\mathbf{J}(\theta)$, podemos concluir que combinações de valores grandes para os pesos não são desejadas, ao menos que eles façam com que o valor da primeira parcela seja muito pequeno. A intuição aqui é que se o vetor de pesos ficar muito grande, um método de otimização que aplica pequenas alterações gradativas a esse vetor não terá condições de alterá-lo significativamente, resultando na convergência para uma solução inadequada. Além disso, incentivar o processo de otimização a manter os pesos pequenos resulta em um RNA menos suscetível a peculiaridades nos dados de entrada.

Quando θ é um vetor de muitas componentes (i.e., quando existem muitas funções candidatas) ou o conjunto de treinamento não é suficientemente grande, o algoritmo de

treinamento é capaz de memorizar todos os exemplos de conjunto, sem produzir um modelo com boa capacidade de generalização. Esse fenômeno é conhecido como *sobreajuste* (overfitting). A segunda parcela da Eq. 10 é apenas uma forma, de diversas possíveis que podem ser usadas com o propósito de evitar que o processo de otimização fique preso em uma solução sub-ótima. Na Seção 3.6, apresentamos outras técnicas com esse mesmo propósito, que são genericamente conhecidas como técnicas de *regularização*[1].

Qualquer algoritmo de otimização pode ser utilizado para o treinamento de uma RNA. Entretanto, por razões práticas, um método de otimização comumente usado é o *gradiente descendente* (*gradient descent*, GD), ou uma de suas diversas variações (*stochastic gradient descent*, *batch gradient descent*, etc.). O GD considera a superfície multidimensional representada pela função de custo J. A ideia básica desse procedimento é realizar (de forma iterativa) pequenas modificações no vetor de parâmetros θ que levem esse vetor na direção da maior descida nessa superfície, o que corresponde a seguir na direção do gradiente[2] negativo da função, $-\nabla J$. Essa ideia pode ser ilustrada graficamente se considerarmos que J é uma função quadrática de apenas dois parâmetros, w_1 e w_2. Nesse caso, a Figura 3.3 apresenta uma curva de níveis de J (regiões mais escuras correspondem a valores cada vez menores de J). A sequência de segmentos de reta nessa figura representa o caminho formado pelas atualizações gradativas nos parâmetros w_1 e w_2 com o propósito de minimizar J.

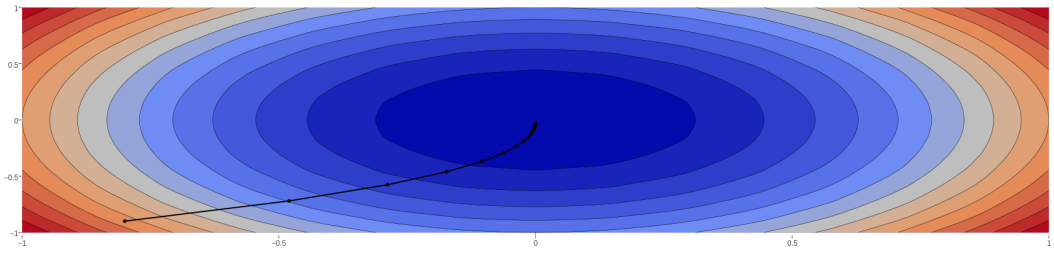


Figura 3.3. O algoritmo dos gradientes descendentes permite percorrer o espaço de busca dos parâmetros de uma rede neural.

Considere que $w[t]$ corresponde ao vetor de pesos de uma RNA na t-ésima iteração do GD. É possível provar que a atualização nos parâmetros realizada pelo GD é dada conforme a Eq. 11. Nessa equação, γ é a *taxa de aprendizagem* (*learning rate*), um hiperparâmetro da RNA cujo valor adequado precisa ser encontrado. De acordo com o discutido em [1], esta abordagem tem várias dificuldades. Uma delas é a escolha do valor da taxa de aprendizagem que propicie o aprendizado rápido, mas ao mesmo tempo, evite o fenômeno da *oscilação*[3].

---

[1] Em Aprendizagem de Máquinas, *regularização* é alguma técnica usada para evitar o sobreajuste durante o treinamento de um modelo.

[2] O gradiente de uma função f mede o quando f varia uma vez que seus argumentos são alterados. Se f for uma função multivariada de n variáveis, então ∇f é um vetor n-dimensional cujas componentes são as derivadas parciais de f.

[3] A oscilação ocorre quando a taxa de aprendizagem escolhida é muito grande, de tal forma que o algoritmo sucessivamente ultrapassa o mínimo da função.

$$\Delta w[t] = -\gamma \frac{\partial J}{\partial w[t]} = -\gamma \nabla J(w[t]) \tag{11}$$

Um aspecto importante na Eq. 11 diz respeito ao modo de calcular as derivadas parciais $\frac{\partial l}{\partial w[t]}$. Nesse contexto, um componente fundamental dos algoritmos para treinamento de redes neurais, incluindo as arquiteturas profundas modernas descritas adiante neste Capítulo, é a retropropagação de erros através das camadas ocultas, a fim de atualizar os parâmetros (pesos e viés) utilizados pelos neurônios [13]. Esse algoritmo, denominado *retropropagação de erros* (*backward propagation of errors* ou *backpropagation*) é combinado com algum método de otimização (e.g., gradiente descendente), que atua para minimizar a função de custo da RNA.

A ideia básica subjacente ao algoritmo de retropropagação é como segue: se a saída produzida pela rede para uma determinada observação $\mathbf{x}^{(t)}$ é diferente da esperada, então é necessário determinar o grau de *responsabilidade* de cada parâmetro da rede nesse erro para, em seguida, alterar esses parâmetros com o propósito de reduzir o erro produzido. Para isso, dois passos são realizados, conforme descrição a seguir.

Para determinar o erro associado à observação $\mathbf{x}^{(t)}$, inicialmente o algoritmo de retropropagação apresenta $\mathbf{x}^{(t)}$ como entrada para a rede e propaga esse sinal através de todas as camadas ocultas até a camada de saída. Esse primeiro passo é denominado *propagação adiante* (*forward step*). Ao realizar esse passo, o algoritmo mantém o registro das pré-ativações $\mathbf{a}^{(k)}$ e ativações $\mathbf{h}^{(k)}$ para todas as camadas intermediárias, uma vez que estes valores serão utilizados para atualização dos parâmetros. O segundo passo (denominado *passo de retropropagação*) consiste em atualizar cada um dos pesos na rede de modo a fazer com que a saída produzida pela rede se aproxime da saída correta, minimizando assim o erro para cada neurônio de saída e para a rede como um todo.

O procedimento composto dos dois passos acima descritos é resumido no Algoritmo 1. O algoritmo de retropropagação é uma aplicação da *regra da cadeia*[4] para computar os gradientes de cada camada de uma RNA de forma iterativa.

3.3. Redes Autocodificadoras

Redes autocodificadoras, também conhecidas como *autocodificadoras* ou *redes autoassociativas*, são uma classe de rede neural que pode ser treinada para aprender *de forma não supervisionada características* (*features*) a partir de um conjunto de dados. Estas características são úteis para uso posterior em tarefas de aprendizado supervisionado, tais como reconhecimento de objetos e outras tarefas relacionadas à visão computacional. Curiosamente, uma autocodificadora é normalmente treinada não para predizer alguma classe, mas para reproduzir na sua saída a própria entrada. Essa família de redes permite o aprendizado de representações mais concisas de um conjunto de dados. Nesta Seção, apresentamos a arquitetura básica das autocodificadoras, além de algumas variações. Descrevemos também algumas aplicações dessa família de redes.

[4]No Cálculo Diferencial, a regra da cadeia é uma expressão que permite calcular as derivadas de primeira ordem de uma função composta: se z é uma função de y, e y é uma função de x, então $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$.

Algorithm 1 Algoritmo de aprendizado por retropropagação.

for $t \leftarrow 1, 2, \ldots, T$ **do**

PASSO ADIANTE

Com início na camada de entrada, usar a Eq. (6) e a Eq. (7) para propagar o exemplo $\mathbf{x}^{(t)}$ através da rede.

PASSO DE RETROPROPAGAÇÃO

Computar as derivadas da função J com relação às ativações da camada $L+1$

for $l \leftarrow L, L-1, \ldots, 1$ **do**

Computar as derivadas da função J com relação às entradas das unidades da camada $l+1$

Computar as derivadas da função J com relação aos pesos entre a camada l e a camada $l-1$

Computar as derivadas da função J com relação às ativações da camada $l-1$

end for

Atualizar os pesos.

end for

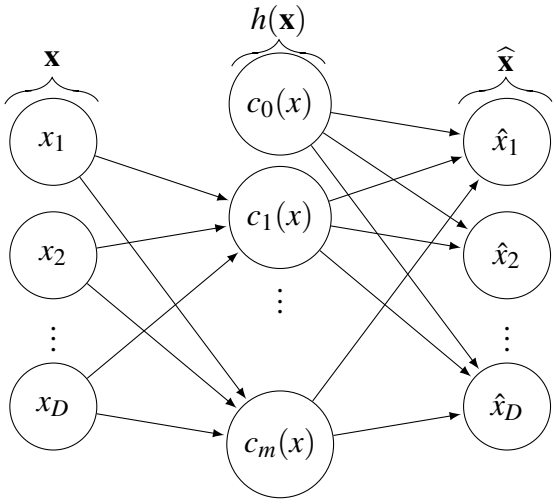


Figura 3.4. Esquema de uma rede autocodificadora

3.3.1. Arquitetura básica e variações

Uma autocodificadora possui pelo menos uma camada intermediária, e as camadas de entrada e de saída possuem igual quantidade de unidades, D. Essa rede é treinada para tentar produzir na saída um padrão $\widehat{\mathbf{x}}$ que seja o mais próximo possível do padrão de entrada $\mathbf{x}$. Na Figura 3.4, que apresenta um esquema de uma rede autocodificadora com uma única camada intermediária, repare que $\mathbf{x}, \widehat{\mathbf{x}} \in \mathbb{R}^D$.

Uma rede autocodificadora realiza transformações na entrada de acordo com dois tipos de funções, conforme descrito a seguir.

- Função de extração de características, $h : \mathbb{R}^D \rightarrow \mathbb{R}^K$. Essa função, conhecida como `codificadora` (*encoder*) mapeia as variáveis do conjunto de dados de treinamento para uma representação latente.

- Função de reconstrução, $r : \mathbb{R}^K \rightarrow \mathbb{R}^D$. Conhecida como `decodificadora` (*decoder*), essa função mapeia a representação produzida por h de volta para o espaço original.

Uma escolha comum para as funções h e r (principalmente quando o vetor de entrada $\mathbf{x}$ é binário) é utilizar a função sigmoide (Seção 3.2.1) aplicada à transformação linear correspondente à multiplicação de $\mathbf{x}$ pela respectiva matriz de pesos: $h(\mathbf{x}) = \sigma(\mathbf{b}_1 + \mathbf{W}_1 \cdot \mathbf{x})$ e $\widehat{\mathbf{x}} = r(h(\mathbf{x})) = \sigma(\mathbf{b}_2 + \mathbf{W}_2 \cdot \mathbf{h(x)})$. Nessas equações, $\mathbf{W}_1$ e $\mathbf{W}_2$ são as matrizes de pesos das camadas intermediária e de saída, respectivamente.

Outra decisão de projeto comum quando da definição da arquitetura de uma rede autocodificadora é usar o mecanismo de *pesos atados* (*tied weights*), o que significa que $\mathbf{W}_2 = \mathbf{W}_1'$ (i.e., uma matriz é a transposta da outra). Essa decisão tem um efeito regularizador (Seção 3.2.2), porque resulta em menos parâmetros para otimizar (quando comparado ao caso em que se permite que $\mathbf{W}_2 \neq \mathbf{W}_1'$, além de prevenir que soluções degeneradas resultem do processo de treinamento. De todo modo, a decisão de usar ou não pesos atados depende do conjunto de dados a ser usado no treinamento [].

Durante o treinamento de uma rede autocodificadora, o objetivo é definir um conjunto de parâmetros no par de funções h e r tal que o *erro de reconstrução* (i.e., a diferença entre $\mathbf{x}$ e $\widehat{\mathbf{x}}$) seja minimizado. Esse erro é representado pela função de custo L apresentada na Eq. 12. Dependendo da arquitetura da rede e da natureza do conjunto de treinamento, podem ser feitas diferentes escolhas para $L(\cdot)$. Para dados binários, pode ser usada a entropia cruzada (*cross-entropy*), enquanto que a média de erros quadrados (*mean squared error*) é popular para dados contínuos. Porque a saída de uma autocodificadora deve ser similar a sua entrada, ela pode ser treinada conforme discutido na Seção 3.2.2 (i.e., usando o algoritmo de retropropagação combinado com algum procedimento de otimização), com o cuidado de substituir os valores-alvo desejados (que não são conhecidos neste caso) pela própria entrada $\mathbf{x}$.

$$L(\mathbf{X}) = \sum_{\mathbf{x}^{(i)} \in \mathbf{X}} \ell(\mathbf{x}^{(i)}, r(h(\mathbf{x}^{(i)}))) \tag{12}$$

Idealmente, os parâmetros aprendidos durante o treinamento devem resultar em uma autocodificadora que identifique uma *representação latente* dos dados de entrada por meio de uma redução de dimensionalidade, ao mesmo tempo em que mantém o máximo de informação acerca da distribuição de entrada. Considere que a representação latente descoberta por uma autocodificadora tenha dimensão D'. No caso em que $D' < D$, temos uma *autocodificadora subcompleta* (*undercomplete autoencoder*). Por outro lado, em uma *autocodificadora supercompleta* (*overcomplete autoencoder*), temos que $D' > D$. A escolha do valor de D' determina a quantidade de unidades da camada intermediária, assim como que tipo de informação a autocodificadora pode aprender acerca da distribuição de entrada.

No caso subcompleto, a autocodificadora deve aprender uma representação compacta da entrada. Como ilustração desse caso, considere um conjunto de dados de treinamento fictício contendo T exemplos. Considere ainda que, em cada exemplo, o esquema

Figura 3.5. Nove exemplos passados para uma autocodificadora com filtragem de ruído e as saídas respectivas reconstruídas por ela.

de codificação usado seja *um bit por estado*[5]. Nesse caso, uma autocodificadora deveria aprender (de forma não supervisionada) a representar os T exemplos da entrada por meio de uma codificação binária compactada, com $\log_2 T$ bits para cada exemplo.

Por outro lado, se considerarmos o aspecto de redução de dimensionalidade, o caso supercompleto parece não ter utilidade alguma. Entretanto, se o objetivo é encontrar características da entrada para posteriormente serem apresentadas a um classificador linear, quanto mais características houver, maior a probabilidade de que esse classificador tenha sucesso em identificar a classe correta de cada objeto. Entretanto, ao treinar uma autocodificadora supercompleta, é preciso tomar certas precauções para assegurar que a representação identificada seja útil. Um exemplo desse tipo de representação é a situação em que a autocodificadora apenas copia os D bits da entrada para D unidades na camada intermediária (deixando de usar $D' - D$ unidades nessa camada).

Variações da autocodificadora simples descrita acima foram propostas com o fim de evitar a produção de soluções indesejadas. Três dessas variações, descritas a seguir, são *autocodificadora com filtragem de ruído* (*denoising autoencoder*), *autocodificadora contrativa* (*contractive autoencoder*) e *autocodificadora esparsa* (*sparse autoencoder*).

Em uma autocodificadora com filtragem de ruído, o objetivo é fazer com que a representação aprendida seja robusta a ruídos nos dados de entrada. Isso pode ser realizado por meio da aplicação de um processo probabilístico em cada exemplo de treinamento antes de apresentá-lo à rede. Uma alternativa de processo probabilístico aplicável neste caso é atribuir zero em cada entrada de um exemplo, com probabilidade p, que passa a ser um hiperparâmetro da rede, denominado *fator de ruído* (*noise factor*). Outra possibilidade é perturbar cada componente da entrada $\mathbf{x}$ por meio de um ruído gaussiano aditivo, com hiperparâmetros μ e σ (média e desvio padrão, respectivamente).

Um aspecto importante é que, se $\tilde{\mathbf{x}}$ for o resultado da aplicação do processo probabilístico ao exemplo $\mathbf{x}$, a função de perda compara $\mathbf{x}$ e $\widehat{\mathbf{x}}$, ou seja, compara a saída com a entrada original, e não com $\tilde{\mathbf{x}}$). A justificativa disso é fazer com que a rede treinada reconstrua a entrada original, ao mesmo tempo em que tenta inferir dependências entre as componentes que foram mantidas intactas e as que foram alteradas (zeradas) pela aplicação do processo probabilístico [14]. A Figura 3.5 apresenta exemplos passados para uma autocodificadora com filtragem de ruído treinada com 100 épocas, com ruído gaussiano, com uso de mini-lotes de tamanho igual a 128, e com fator de ruído $p = 0,5$.

Outra abordagem para coibir a produção de representações indesejadas é por meio do treinamento de uma *autocodificadora contrativa* [15]. Nessa abordagem, um termo é adicionado à função de perda para penalizar as representações indesejadas do espaço latente. Em particular, é adicionada uma parcela correspondente à norma de Frobenius

[5]Na codificação *um bit por estado* (*one-hot encoding*), cada exemplo é representado como uma sequência de b bits na qual apenas um bit é igual a 1, e os bits restantes são todos iguais a 0.

da Jacobiana[6] correspondente à função h com relação às unidades da camada de entrada. Veja a Eq. 13, composta de duas parcelas. A primeira incentiva o processo de otimização a encontrar soluções que sejam adequadas do ponto de vista do erro de reconstrução. Já a segunda parcela penaliza qualquer reconstrução, posto que se trata de uma função quadrada (cuja primeira derivada é zero). Sendo assim, a combinação dessas duas parcelas direciona o processo de otimização para manter apenas representações adequadas com relação ao erro de reconstrução.

$$-\sum_{k=1}^{d}[x_k \log z_k + (1-x_k)\log(1-z_k)] + \|\frac{\partial h(x)}{\partial x}\|^2 \tag{13}$$

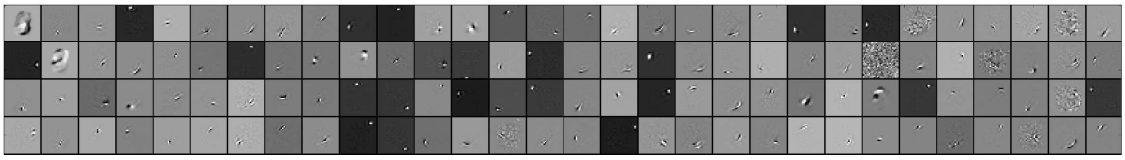
Uma terceira alternativa para evitar que o algoritmo de treinamento aprenda uma representação inadequada (mesmo com o uso de mais unidades ocultas do que as existentes na camada de entrada) é por meio do uso das autocodificadoras esparsas, que procuram fazer com que apenas uma pequena quantidade de unidades da camada oculta seja ativada para cada padrão de entrada. Nesse tipo de rede, a esparsidade pode ser obtida por meio da adição de termos adicionais na função de perda durante o treinamento, ou mantendo apenas as k unidades mais ativas e tornando todas as demais unidades iguais a zero manualmente. Este último caso corresponde às denominadas autocodificadoras k-esparsas (*k-sparse autoencoders*), redes com funções de ativação lineares e pesos atados [16]. O valor de k, um hiperparâmetro denominado *fator de esparsidade*, corresponde à quantidade de unidades ocultas que devem mantidas.

Uma vez que uma autocodificadora é treinada, pode ser útil visualizar (por meio de uma imagem bidimensional) a função codificadora **h** aprendida pela rede, com o propósito de entender que característica cada unidade oculta tenta detectar. Como exemplo desse processo, considere a situação de treinar uma autocodificadora k-esparsa de uma única camada intermediária de 1000 unidades sobre o conjunto de dados MNIST[7]. A Figura 3.6 apresenta a visualização dos pesos após o treinamento para diferentes valores de k. Os pesos estão mapeados para valores de pixels de tal forma que um valor de peso negativo é preto, um valor de peso próximo de zero é cinza, e um valor de peso positivo é branco. Cada quadrado nessa figura apresenta o (norma limitada) da imagem de entrada x que ativa maximamente uma das 1000 unidades ocultas. É possível perceber que as unidades ocultas aprenderam a detectar bordas em diferentes posições e orientações na imagem. É possível também notar que, para $k = 25$ e $k = 40$, a saída é reconstruída usando um menor número de unidades ocultas do que em $k = 70$ e, portanto, as características tendem a ser melhores. Por outro lado, se o fator de esparsidade for muito pequeno (e.g., $k = 10$), as características detectadas são muito globais e inadequadas para fatorar a entrada em partes que façam sentido.

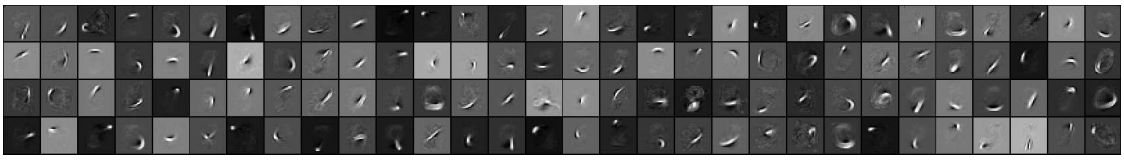
Outro tipo especial de autocodificadora, denominada *autocodificadora linear* (*linear autoencoder*), é obtida conforme descrito a seguir. Em primeiro lugar, deixamos de aplicar uma não linearidade para produzir a saída da camada intermediária, i.e., usamos

[6] A Jacobiana de uma função é uma matriz cujos elementos são derivadas parciais de primeira ordem dessa função. Já a norma de Frobenius de uma matriz $A_{n\times m}$ é obtida por $||A||^2 = \sum_{i=1}^{n}(\sum_{j=1}^{m}|a_{ij}|^2) = \sqrt{\text{traço}(AA^T)}$.

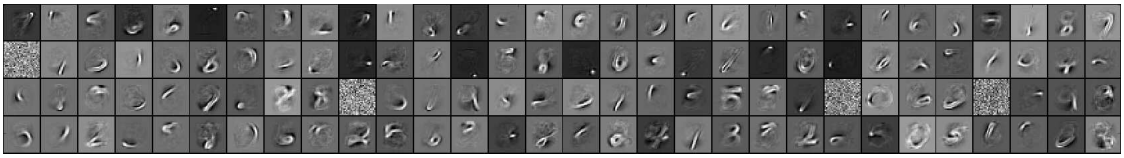
[7] `http://yann.lecun.com/exdb/mnist/`



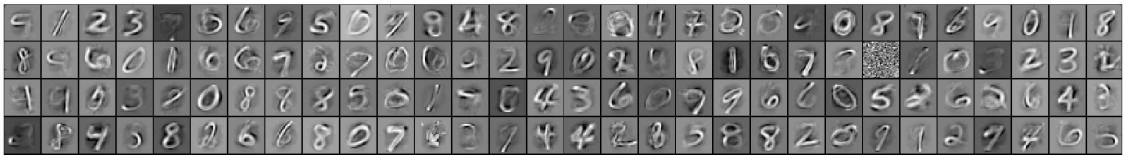
(a) $k = 70$



(b) $k = 40$



(c) $k = 25$



(d) $k = 10$

Figura 3.6. Filtros de uma autocodificadora *k*-esparsa para diferentes níveis de esparsidade *k*, treinada sobre o conjunto de dados MNIST com 1000 unidades ocultas.

$h(\mathbf{x}) = \mathbf{b}_1 + \mathbf{W}_1 \cdot \mathbf{x}$. Além disso, usamos a média de erros quadrados como função de perda L (ver Eq. 12). Pode ser provado [17, 18] que uma autocodificadora linear treinada com um conjunto de dados especialmente normalizado produz resultados equivalente aos obtidos por meio da técnica de Análise de Componentes Principais [19] e, por conta disso, é ótima no sentido de gerar o menor erro de reconstrução. Esse tipo de autocodificadora pode ser aplicado apenas no caso em que os dados de entrada residem em uma superfície multidimensional linear.

3.3.2. Aplicações

Autocodificadoras são úteis no contexto do aprendizado de codificações mais compactas de um conjunto de dados, seja no sentido de menor dimensionalidade, ou no sentido de a representação resultante ser mais esparsa [20].

Autocodificadoras podem também ser usadas como uma espécie de bloco de construção para abordagens de aprendizagem profunda. Em particular, autocodificadoras podem ser usadas para iniciar os pesos de uma rede profunda para uso posterior em uma tarefa de aprendizado supervisionado. De fato, das duas abordagens originais de pré-treinamento não supervisionado (Seção 3.6.1) para iniciar os pesos antes do treinamento supervisionado de uma rede profunda, uma delas envolvia o uso de autocodificadoras [21]. Em [22], os autores apresentam um estudo sobre o uso de autocodificadoras para

pré-treinamento não supervisionado. De todo modo, autocodificadoras com esse propósito têm se tornado menos importantes nos últimos anos, muito pelos avanços obtidos em técnicas alternativas para inicialização de pesos e para controlar a regularização de redes profundas (veja a Seção 3.6), assim como pela maior disponibilidade de quantidades suficientes de dados rotulados.

Em [23], os autores apresentam o resultado do treinamento de uma autocodificadora esparsa de 9 camadas sobre um conjunto de 10 milhões de imagens. Eles conseguem demonstrar que é possível treinar um detector de faces sem a necessidade de usar um conjunto de treinamento no qual os exemplos estão rotulados.

3.4. Redes Convolucionais

Redes neurais convolucionais (*convolutional neural networks*, CNN) se inspiram no funcionamento do córtex visual [24, 25]. Nesta Seção, apresentamos detalhes acerca dessa classe de redes. Descrevemos suas características particulares, tais como o compartilhamento de pesos e a conectividade esparsa. Apresentamos também as operações primitivas principais utilizadas em camadas intermediárias de redes dessa família, a saber, subamostragem e convolução.

3.4.1. Conceitos e Operações

Redes de convolução se baseiam em algumas ideias básicas, a saber, *campos receptivos locais* (*local receptive fields*), *compartilhamento de pesos* (*shared weights*), convolução (*convolution*) e *subamostragem* (*subsampling* ou *pooling*). A seguir, apresentamos detalhes acerca desses conceitos.

Redes neurais tradicionalmente são *completamente conectadas*, i.e., cada unidade na camada L está conectada a todas as unidades da camada $L-1$. Suponha que fôssemos modelar uma rede completamente conectadas com apenas uma camada intermediária para o conjunto de dados MNIST. Suponha ainda que essa única camada intermediária tenha o dobro de unidades da camada de entrada. Sendo assim, teríamos da ordem de 10^6 parâmetros para ajustar durante o treinamento dessa rede, e isso para imagens representadas em escala de cinza. (Para imagens coloridas, esse valor dever ser triplicado para considerar os 3 canais de cores, RGB.)

Além de possuir uma quantidade grande de parâmetros, a arquitetura de rede descrita no parágrafo anterior não leva em consideração a estrutura espacial tipicamente existente em dados correspondentes a imagens. Isso quer dizer que tanto pixels que estão próximos quanto os localizados em duas regiões muitos distantes da imagem (e.g., cantos superior esquerdo e inferior direito) são tratados indistintamente. Nessa arquitetura, a própria rede teria que detectar as dependências existentes na estrutura espacial da distribuição subjacente às imagens de entrada. Além disso, devido à conectividade muito grande, esse tipo de arquitetura sofre da denominada *maldição da dimensionalidade* (*curse of dimensionality*) e portanto não é adequado a imagens de alta resolução, por conta do alto potencial de sobreajuste (Seção 3.2.2) aos dados. Isso sem considerar o tempo para computar as pré-ativações de todas as unidades em cada camada.

Em vez da conectividade global, uma rede convolucional utiliza conectividade

local. Por exemplo, considere que $L^{(1)}$ é a primeira camada intermediária de uma CNN. Cada unidade de $L^{(1)}$ está conectada a uma quantidade restrita de unidades localizada em uma região contígua da camada de entrada $L^{(0)}$, em vez de estar conectada a todas as unidades. As unidades de $L^{(0)}$ conectadas a uma unidade de $L^{(1)}$ formam o *campo receptivo local* dessa unidade. Por meio de seu campo receptivo local, cada unidade de $L^{(1)}$ pode detectar características visuais elementares, tais como arestas orientadas, extremidades, cantos. Essas características podem então ser combinadas pelas camadas subsequentes para detecção de características mais complexas (e.g., olhos, bicos, rodas, etc.).

Por outro lado, é provável que um determinado detector de alguma característica elementar seja útil em diferentes regiões da imagem de entrada. Para levar isso em consideração, em uma CNN as unidades de uma determinada camada são organizadas em conjuntos disjuntos, cada um dos quais é denominado um *mapa de característica* (*feature map*), também conhecido como *filtro*. As unidades contidas em um mapa de características são únicas na medida em que cada um delas está ligada a um conjunto de unidades (i.e., ao seu campo receptivo) diferente na camada anterior. Além disso, todas as unidades de um mapa compartilham os mesmos parâmetros (i.e., a mesma matriz de pesos e viés). O resultado disso é que essas unidades dentro de um mapa servem como detectores de uma mesma característica, mas cada uma delas está conectada a uma região diferente da imagem. Portanto, em uma CNN, uma camada oculta é segmentada em diversos mapas de características (os planos ilustrados na Figura 3.8), em que cada unidade de um mapa tem o objetivo realizar a mesma operação sobre a imagem de entrada, com cada unidade aplicando essa operação em uma região específica dessa imagem.

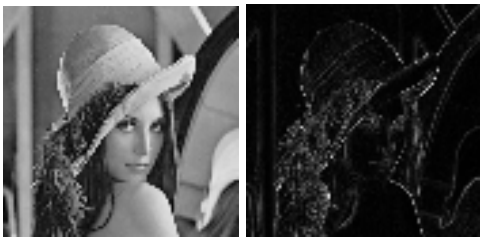
Cada unidade em um mapa de característica realiza uma operação denominada *convolução*. Para entender essa operação, considere que a imagem de entrada contém 28×28 pixels. Por simplicidade, vamos considerar ainda que essas imagens de entrada são em escala de cinza. Em processamento de imagens, uma convolução sobre uma imagem I corresponde a aplicar o produto de Hadamard[8] entre a matriz de pixels de I e outra matriz, denominada *núcleo da convolução* (*convolution kernel*). Por exemplo, a Figura 3.7(a) apresenta o resultado da aplicação de um tipo particular de convolução sobre uma imagem.

Outra operação importante utilizada em uma CNN é a *subamostragem*. Em processamento de imagens, a subamostragem de uma imagem envolve reduzir a sua resolução, sem no entanto alterar significativamente o seu aspecto. No contexto de uma CNN, a subamostragem reduz a dimensionalidade de um mapa de característica fornecido como entrada e produz outro mapa de característica, uma espécie de resumo do primeiro. Há várias formas de subamostragem aplicáveis a um mapa de característica: selecionar o valor máximo (*max pooling*), a média (*average pooling*) ou a norma do conjunto (*L2-pooling*), entre outras. Como exemplo, considere que a ilustração à esquerda da Figura 3.7(b)[9] corresponde às ativações das unidades de um mapa de característica de tamanho 4x4. A ilustração à direita apresenta outro mapa de característica, resultante da operação de subamostragem com o uso de filtros de tamanho igual a 2x2 e tamanho do passo igual

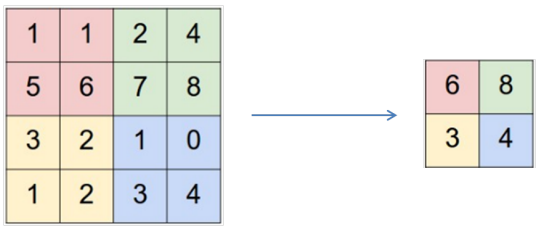
[8] O produto de Hadamard de duas matrizes $A_{m\times n}$ e $B_{m\times n}$ resulta em outra matriz $C_{m\times n}$ tal que $c_{ij} = a_{ij}b_{ij}$.

[9] Figura adaptada do material disponibilizado na disciplina CS231n `http://cs231n.stanford.edu/`.

a 2. O uso da subamostragem faz com que uma CNN seja robusta em relação à localizações exatas das características na imagem, uma propriedade que permite a esse tipo de rede aprender representações invariantes com relação a pequenas translações.



(a) Resultado de uma convolução (direita) aplicada a uma imagem (esquerda).



(b) Subamostragem com filtro de tamanho 2x2 e tamanho do passo igual a 2.

Figura 3.7. Exemplos das operações de subamostragem e de convolução.

3.4.2. Arquitetura

Os conceitos apresentados na Seção 3.4.1 servem de blocos de construção para montar as camadas de uma CNN, conforme descrevemos nesta Seção. Em geral, uma CNN possui diversos tipos de camadas: *camadas de convolução*, *camadas de subamostragem*, *camadas de normalização de contraste* e *camadas completamente conectadas*. Na forma mais comum de arquitetar uma CNN, a rede é organizada em *estágios*. Cada estágio é composto por uma ou mais camadas de convolução em sequência, seguidas por uma *camada de subamostragem*, que por sua vez é seguida (opcionalmente) por uma camada de normalização. Uma CNN pode conter vários estágios empilhados após a camada de entrada (que corresponde à imagem). Após o estágio final da rede, são adicionadas uma ou mais camadas completamente conectadas.

A Figura 3.8 apresenta um exemplo esquemático de uma CNN na qual, após a camada de entrada (que corresponde aos pixels da imagem), temos uma *camada de convolução* composta de 6 mapas de características (representados como planos na figura), seguida de uma camada de subamostragem, completando o primeiro estágio. Essa figura ainda ilustra um segundo estágio antes das duas camadas completamente conectadas.

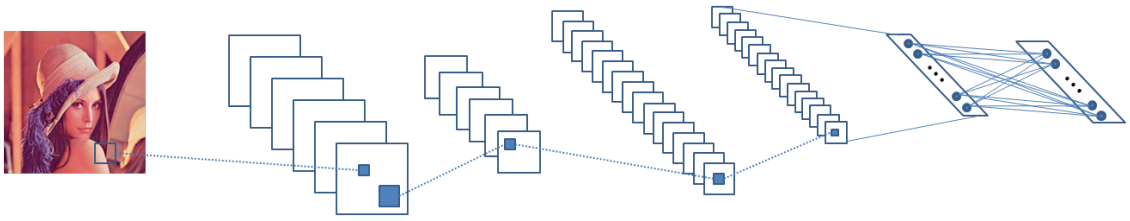


Figura 3.8. Arquitetura típica de uma rede convolucional. Figura adaptada de [24].

No contexto de uma CNN, a computação realizada em cada mapa de característica M de uma camada de convolução C envolve a aplicação de várias operações de convolução, uma para cada unidade contida em M, conforme descrito a seguir. Cada unidade u de M realiza uma convolução usando a matriz de pesos de M. (Lembre-se de que todas as unidades em M compartilham a mesma matriz de pesos.) A matriz de pesos corresponde

ao núcleo dessa convolução, e a matriz sobre a qual a convolução é aplicada correspondente ao campo receptivo local de u. A quantidade de unidades é a mesma em cada mapa de característica de C e depende (1) do tamanho do campo receptivo das unidades e (2) do denominado *tamanho do passo* (*stride size*). Esse segundo hiperparâmetro é um inteiro positivo (normalmente igual a 1) que define o quanto de sobreposição há entre os campos receptivos locais de duas unidades vizinha em um mapa L. Por exemplo, considere que as imagens usadas como entrada para C são de tamanho 28x28 pixels, que o campo receptivo local de cada unidade tem tamanho 5x5, e que o tamanho do passo seja igual a 1. Então, cada mapa de características em C deve conter 24x24 unidades. (A melhor maneira de entender esse resultado é imaginar que uma janela (de mesmo tamanho dos campos receptivos) deslize da esquerda para a direita e de cima para baixo na imagem de entrada.)

Embora a descrição aqui fornecida sobre o funcionamento de uma camada de convolução tenha sido apresentada considerando que C recebe uma imagem como entrada, o mesmo procedimento de funcionamento se aplica na computação realizada pelas demais camadas de convolução da CNN.

Outro tipo de camada em uma CNN é a denominada *camada de subamostragem* (*subsampling layer* ou *pooling layer*). O objetivo dessa, que também é composta por um conjunto de mapas de características, é realizar a agregação das saídas (ativações) de um conjunto de unidades da camada anterior. É possível mostrar que camadas de subamostragem resultam em uma rede mais robusta a transformações espaciais (e.g., mais invariante a eventuais translações dos objetos em uma imagem).

Redes neurais convolucionais também podem conter as denominadas *camadas de normalização de contraste local* (*Local Contrast Normalization*, LCN) [26]. Quando utilizada, essa camada é posicionada na saída da camada de subamostragem. Uma camada LCN normaliza o contraste de uma imagem de forma não linear. Em vez de realizar uma normalização global (i.e., considerando a imagem como um todo), a camada LCN aplica a normalização sobre regiões locais da imagem, considerando cada pixel por vez. A normalização pode corresponder a subtrair a média da vizinhança de um pixel particular e dividir pela variância dos valores de pixel dessa vizinhança. Esta transformação equipa a CNN com a *invariância de brilho*, propriedade útil no contexto de reconhecimento de imagens.

O outro tipo de camada em uma CNN é a completamente conectada. Em arquiteturas de CNNs modernas, é comum encontrar uma ou duas camadas desse tipo antes da camada de saída. Juntamente com as camadas de convolução e subamostragem, as camadas totalmente conectadas geram descritores de características da imagem que podem ser mais facilmente classificados pela camada de saída (que normalmente usa a função softmax; ver Seção 3.2.1).

É importante notar que os pesos dos mapas de características em cada camada de convolução são aprendidos durante o treinamento. Por conta de as operações de convolução e subamostragem serem diferenciáveis, uma CNN pode ser também treinada por meio do algoritmo de retropropagação (Seção 3.2.2), com a diferença de que os pesos são atualizados considerando a média dos gradientes dos pesos compartilhados. Durante esse treinamento, é comum aplicar a técnica de *desligamento* (Seção 3.6.3) às camadas

completamente conectadas para evitar o sobreajuste.

3.4.3. Aplicações

Por meio das redes neurais convolucionais, é possível realizar a detecção de padrões em imagens de forma mais adequada, no sentido de que características dos dados dessa natureza podem ser exploradas para obter melhores resultados. Desde sua origem, esse tipo de rede tem se mostrado adequado em tarefas que envolvem reconhecimento visual, tais como reconhecimento de caracteres manuscritos e detecção de faces [27, 7, 24]. Com o sucesso resultante do uso de arquiteturas profundas para o processamento visual e o surgimento de bases de dados de imagem com milhões de exemplos rotulados (por exemplo, ImageNet[10], Places[11]), o estado da arte em visão computacional por meio de CNNs tem avançado rapidamente.

Essas redes ganharam popularidade em 2012, quando foram usadas para reduzir substancialmente a taxa de erro em uma conhecida competição de reconhecimento de objetos, a ImageNet[12] [28, 29]. Conforme apresentado nesta Seção, o objetivo de cada mapa de característica em uma CNN é extrair características da imagem fornecida. Portanto, uma CNN também desempenha o papel de extrator *automático* de características da imagem. De fato, desde 2012, a área de Visão Computacional tem sido invadida por abordagens que usam CNNs para tarefas de classificação e detecção de objetos em imagens, em substituição às abordagens anteriores que envolviam a extração manual das características das imagens para posteriormente aplicar algum método de classificação, como por exemplo as máquinas de vetores suporte (*support vector machines*, SVM). Aplicações mais recentes de CNNs envolvem a detecção de pedestres [30] e de placas de sinais de trânsito [31].

3.5. Redes Recorrentes

Redes Neurais Recorrentes (*Recurrent Neural Networks*, RNN) constituem uma ampla classe de redes cuja evolução do estado depende tanto da entrada corrente quanto do estado atual. Essa propriedade (ter estado dinâmico) proporciona a possibilidade de realizar computação dependente do contexto e aprender dependências de longo prazo: um sinal que é fornecido a uma rede recorrente em um instante de tempo t pode alterar o comportamento dessa rede em um momento $t+k$, $k > 0$.

3.5.1. Arquitetura

A arquitetura usada em RNNs é adequada para permitir o processamento de *informação sequencial* (e.g., textos, áudio e vídeo). Um exemplo de tarefa em Processamento de Linguagem Natural que envolve esse tipo de informação é a *Modelagem de Linguagem*, que corresponde a criar um modelo preditivo da próxima palavra em uma frase [32, 33]. Nesse tipo de tarefa, é fácil perceber que conhecer a sequência de palavras anteriores é importante para a predição. Uma particularidade das RNNs é que esse tipo de rede possui uma memória, que permite registrar que informações foram processadas até o momento

[10] http://image-net.org/
[11] http://places.csail.mit.edu/
[12] http://image-net.org/

atual.

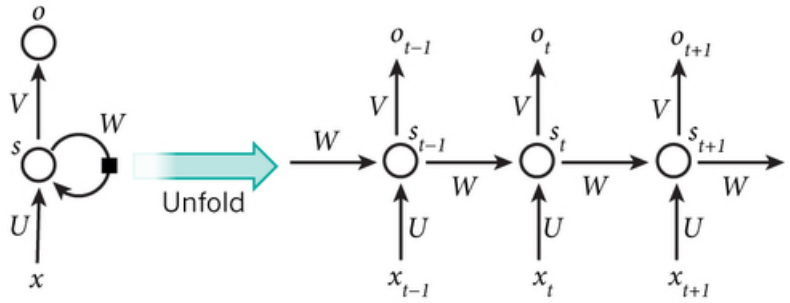


Figura 3.9. Esquema do desdobramento da computação em uma rede recorrente. Fonte: [7]

Há muitas maneiras pelas quais as unidades de uma RNN podem estar conectadas entre si. A ilustração à esquerda da Figura 3.9 apresenta a situação mais simples possível, uma RNN que possui apenas uma camada oculta, composta de uma única unidade, denotada por *s*. Apresentada dessa forma, a rede é aparentemente simples. Entretanto, repare que a unidade oculta contém uma conexão que a liga consigo mesma. Dessa forma, *s* recebe sinais que foram gerados por ela própria um passo de tempo no passado.

O *desdobramento no tempo* (*time unfolding*) de uma RNN corresponde à situação em que essa rede recebe uma sequência de sinais da entrada, um em cada passo de tempo. Na ilustração da direita da Figura 3.9, é apresentada a mesma rede da esquerda *desdobrada* (*unfolded*) no tempo. Note que, vista dessa forma, uma RNN que recebe uma sequência de *n* vetores de entrada pode ser vista como uma rede alimentada adiante (Seção 3.2.1) de *n* camadas.

Repare também que, embora na Figura 3.9 sejam apresentadas entradas e saídas em cada passo de tempo, dependendo da tarefa específica para qual a RNN deve ser projetada, alguns dos passos de tempo podem não conter nem entradas nem saídas. Isso permite que RNNs aprendam mapeamentos entre sequências de entrada e de saída com diferentes tamanhos: um-para-*n* (e.g., produzir a legenda correspondente a uma imagem), *n*-para-um (e.g., produzir a carga de sentimento, positiva ou, negativa, de uma frase de entrada), *n*-para-*m* (e.g., traduzir uma frase do inglês para o português).

Considerando ainda a Figura 3.9, os vários componentes de uma RNN representados nessa segunda ilustração são descritos a seguir.

- x_t é o vetor que representa a entrada fornecida à RNN no passo de tempo t. Por exemplo, se a RNN processa sequências de palavras, então x_0 é a representação da primeira palavra, x_1 representa a segunda palavra, e assim por diante.
- s_t é o estado (i.e., o valor de ativação) da unidade oculta no passo de tempo t. Essa unidade representa a *memória* da rede, porque sua finalidade é registrar que informação foi processada desde alguns passos de tempo no passado até o presente momento. A ativação dessa unidade é computada como uma função recursiva, que depende do estado anterior e da entrada no passo atual: $s_t = \sigma(Ux_t + Ws_{t-1} + b_s)$. σ é a função de ativação escolhida (Seção 3.2.1). Repare que s_{-1}, que corresponde ao valor da base da recursão, é normalmente definido como o vetor nulo.

- o_t é a saída no passo de tempo t. Por exemplo, se a tarefa corresponde a predizer a próxima palavra em uma frase, então o_t é um vetor de probabilidades sobre todas as palavras do vocabulário, i.e., $o_t = \text{softmax}(Vs_t + b_o)$.

- As matrizes U (conexões entre estados ocultos), V (conexões dos estados ocultos para a saída) e W (conexões da entrada para os estados ocultas) são os parâmetros que a rede deve aprender durante o treinamento. De forma similar ao que acontece com as redes convolucionais (Seção 3.4), esses parâmetros são compartilhados por todos os passos de tempo. Esse compartilhamento permite que a rede processe entradas de tamanho variável.

Uma vez definida a arquitetura da RNN, é possível iniciar o treinamento por meio da minimização de uma função de custo J. Nesse caso, uma variante do algoritmo de retropropagação do erro (Seção 3.2.2) é utilizada, a denominada *retropropagação através do tempo* (*Backpropagation Through Time*, BPTT). O BPTT pode ser entendido se considerarmos que uma RNN corresponde a uma rede alimentada adiante com uma camada para cada passo de tempo e na qual a matriz de pesos de cada camada é a mesma, i.e., se considerarmos o desdobramento da rede através do tempo. (Aliás, é dessa perspectiva que se originou o nome do algoritmo BPTT.) Dessa forma, após o fornecimento da sequência de entrada, os gradientes são calculados (da mesma maneira que nas redes alimentadas adiante) e posteriormente alterados para manter a restrição de igualdade da matriz de pesos em cada camada oculta. Por exemplo, se w_1 e w_2 são os valores de conexões correspondentes em duas camadas ocultas distintas (i.e., em dois passos de tempo distintos), então $w_1 = w_2$. Para fazer com que esses pesos continuem sendo iguais no próximo passo de tempo, o BPTT calcula $\frac{\partial J}{\partial w_1}$ e $\frac{\partial J}{\partial w_2}$ e usa a média simples (ou a soma) dessas quantidades para atualizar tanto w_1 quanto w_2.

3.5.2. Aplicações

Devido a sua flexibilidade no que concerne ao processamento de entradas de tamanho variável, RNNs têm sido aplicadas recentemente em uma variedade de problemas. De fato, a maioria das aplicações recentes de RNNs envolve o uso de um tipo mais complexo dessa classe de redes proposto em 1997, as denominadas redes LSTM (*Long Short-Term Memory*) [34]. Basicamente, uma rede LSTM é formada de blocos (ou células) que, por possuírem uma estrutura interna, podem armazenar um valor por uma quantidade arbitrária de passos de tempo. As redes LSTM e sua contrapartida mais moderna (e mais simples), as redes GRU (*Gated Recurrent Unit*) [35], são especialmente apropriadas para evitar o problema da dissipação dos gradientes (Seção 3.6).

Algumas aplicações das RNNs são reconhecimento de fala [36, 37], geração de textos [38], geração de textos manuscritos (*handwriting generation*) [39], tradução automática [40] e predição em vídeo [41].

Visto que RNNs são máquinas de Turing completas, em teoria, elas podem realizar qualquer computação. Nesse contexto, tem surgido recentemente aplicações de RNNs baseadas nos *modelos de atenção* (*atention models*). No contexto de uma tarefa de visão computacional, um modelo de atenção permite que essa rede direcione sequencialmente seu foco para um subconjunto da entrada por vez. Exemplos de aplicações desses modelos

de atenção são [42] (para geração de imagens) e geração legendas para imagens [43].

3.6. Técnicas para Treinamento de Redes Profundas

Apesar das vantagens computacionais e de ser um modelo mais aproximado do cérebro, o treinamento de redes neurais profundas era uma tarefa de difícil execução até uma década atrás, devido a dois complicadores principais. O primeiro deles é conhecido como a *dissipação dos gradientes* (*vanishing gradients*) [44], que é causado por unidades que estão *saturadas*, conforme descrição a seguir. Durante a aplicação do algoritmo de retropropagação, o gradiente da ativação de cada unidade oculta é calculado e utilizado como um termo multiplicativo para atualização dos pesos correspondentes. Entretanto se a unidade em questão está próxima da saturação, então a derivada parcial resultante será um valor próximo de zero. Isso faz com que o gradiente com relação às pré-ativações seja também próximo de zero, porque seu cálculo envolve multiplicar pela derivada parcial. Isso significa que o gradiente retropropagado para as camadas anteriores será cada vez mais próximo de zero, de tal forma que quanto mais próximo à camada de entrada, maior a quantidade de gradientes próximos de zero. Como resultado, à medida que o gradiente do erro é retropropagado seu valor decresce exponencialmente, de tal forma que o aprendizado nas camadas mais próximas à entrada se torna muito lento. Esse problema afeta não apenas o aprendizado em redes profundas alimentadas adiante (Seção 3.2.1), mas também em redes recorrentes (Seção 3.5). O segundo complicador é consequência do número maior de parâmetros para adequar (os parâmetros de uma rede neural são seus pesos, e quanto mais camadas, mais pesos). De fato, quanto maior a quantidade de parâmetros a ajustar em um modelo, maior o risco de *sobreajuste* (Seção 3.2.2).

O aprendizado de modelos por meio do treinamento de redes neurais profundas tem sido tradicionalmente tratado como um processo empírico. De fato, soluções para os problemas mencionados acima envolvem o uso de diversos procedimentos que foram surgindo ao longo dos anos e que em conjunto passaram a contar como fatores importantes no sucesso da aprendizagem profunda (além do aumento do poder computacional, da disponibilidade de grandes conjuntos de dados e de melhores arquiteturas de rede). Nessa Seção, descrevemos procedimentos cujo propósito é treinar redes neurais profundas de forma adequada, seja diminuindo o tempo de treinamento, seja diminuindo o erro de generalização dos modelos resultantes.

3.6.1. Pré-treinamento Não Supervisionado

A ideia subjacente ao pré-treinamento não supervisionado (*unsupervised pre-training*) é iniciar os pesos de uma rede por meio de uma abordagem não supervisionada [45, 46, 21]. Esse procedimento permite posicionar os pesos da rede em uma região do espaço de parâmetros tal que seja mais fácil para métodos de otimização baseados em gradientes encontrarem um valor ótimo para o objetivo durante o treinamento supervisionado [22].

Durante a aplicação desse procedimento, camadas intermediárias são adicionadas à rede iterativamente. Em cada iteração, uma nova camada intermediária é adicionada à rede. A nova camada intermediária adicionada deve capturar regularidades nos padrões de ativação das unidades contidas na camada anterior. Esse procedimento pode ser aplicado para a construção de redes com um número arbitrário de camadas intermediárias.

Após a fase de pré-treinamento não supervisionado (que substitui a iniciação aleatória dos pesos), a camada de saída (i.e., a que produz uma distribuição de probabilidades sobre os rótulos da tarefa de classificação em questão) é então adicionada à rede. (Os pesos que conectam essa camada com a camada anterior podem ser iniciados de forma aleatória.) Finalmente, a rede completa é treinada da forma supervisionada usual (i.e., com sequências de propagações de sinais adiante intercaladas por retropropagações do erro). Essa segunda fase é denominada sintonia fina (*fine-tuning*).

Uma vantagem do pré-treinamento é que é possível tirar proveito da existência de dados não rotulados, que em geral são mais comuns do que dados rotulados. Outra vantagem é que normalmente o aprendizado com pré-treinamento converge mais rapidamente. nos trabalhos que propuseram originalmente o uso de pré-treinamento [45, 46], os autores realizaram o treinamento não supervisionado por meio de máquinas restrita de Boltzmann (*restricted Boltzmann machines*). Posteriormente, outros autores propuseram usar pré-treinamento por meio de redes autocodificadoras [47, 21] (Seção 3.3).

3.6.2. Uso de Unidades Lineares Retificadas

Durante anos, foi um consenso na comunidade de redes neurais usar funções sigmoides (Seção 3.2.1) para implementar a ativação de unidades ocultas, de modo a induzir não linearidades. Entretanto, esse consenso foi questionado em 2010 por Nair e Hinton [48], que propuseram o uso de uma alternativa mais simples, a *função de ativação retificada linear* (Eq. 5). Neurônios que usam essa função de ativação são chamados de *unidades retificadoras lineares* (*rectified linear units*, ReLU).

Foi demonstrado por meio de experimentos [28] que em geral a função ReLU produz um erro de treinamento menor, por conta de não ser tão suscetível ao problema da *dissipação dos gradientes* quanto as funções sigmoide e tangente hiperbólica. Além disso, essa função de ativação reduz o tempo de convergência dos parâmetros (por conta de sua computação não envolver exponenciações). Por exemplo, em [28], os autores apresentam uma comparação entre as curvas de convergência em uma rede de convolução com 4 camadas em duas variações, uma com neurônios ReLU e outra com neurônios tanh no conjunto de dados CIFAR-10[13]. Nos experimentos apresentados, a primeira variação converge 6 vezes mais rápido.

Em 2011, o grupo de Yoshua Bengio [49] demonstrou que ReLUs permitem o treinamento supervisionado de redes profundas sem a necessidade de utilizar pré-treinamento não supervisionado (Seção 3.6.1). Em 2015, uma rede convolucional treinada no conjunto de dados ImageNet com a função ReLU pela primeira vez atingiu precisão de classificação maior do que a alcançada por um ser humano [50].

3.6.3. Desligamento (*Dropout*)

Essa é outra técnica também aplicável ao treinamento de uma rede neural para reduzir o risco de sobreajuste [51, 52]. Na fase de propagação adiante (*forward propagation*), antes de um padrão ser apresentado à camada intermediária de índice k, cada entrada de um vetor binário $\mathbf{m}^{(k)}$ é gerado por meio de uma distribuição uniforme com parâmetro p

[13] O conjunto de dados CIFAR-10 (`https://www.cs.toronto.edu/~kriz/cifar.html`) consiste de 60.000 imagens coloridas de 32x32 pixels, divididas em 10 classes, com 6000 imagens por classe.

($p = 0{,}5$ é um valor usual). Esse vetor é então usado como uma máscara para *desativar* algumas das unidades da camada k. Mais especificamente, a pré-ativação da camada k ocorre de forma usual (conforme a Eq. 1), mas a sua ativação é modificada ligeiramente: $h(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \cdot \mathbf{m}^{(k)}$ (compare com a Eq. 2). Efetivamente, essa modificação faz com que a ativação das unidades correspondentes a valores iguais a zero na máscara $\mathbf{m}^{(k)}$ seja também igual a zero [53]. A Figura 3.10 o efeito da aplicação do desligamento: a imagem da esquerda é a configuração original da rede, e a da direita corresponde à configuração após o desligamento de algumas unidades.

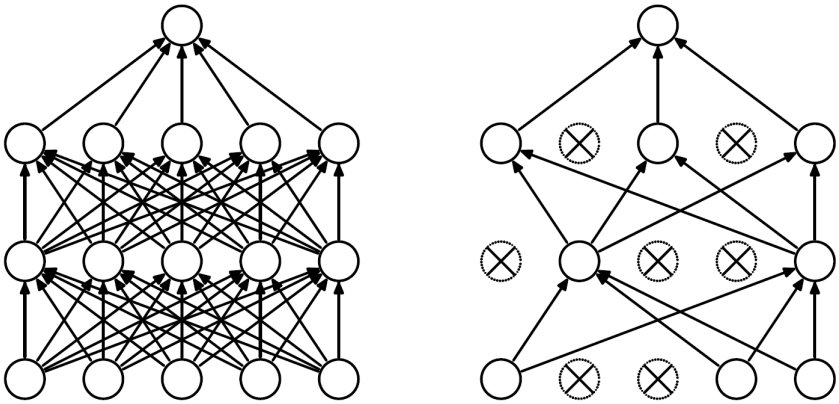


Figura 3.10. Efeito da aplicação do desligamento (dropout) em uma rede neural. Fonte: [52]

A aplicação de desligamento requer a definição de um hiperparâmetro, a probabilidade p que controla a intensidade de desligamento. Se $p = 1$ implica nenhum desligamento, e quanto menor o valor de p, maior a taxa de desligamento. Valores típicos de p para as unidades ocultas estão na faixa de 0,5 a 0,8 [52].

De acordo com [54], o procedimento de desligamento pode ser interpretado (e mesmo justificado) como uma forma adaptativa de *acréscimo de dados* (*data augmentation*) [55]. De fato, esses autores apontam que definir a mudança de ativação de algumas unidades ocultas para 0 é equivalente a fornecer na entrada da rede um exemplo moldado para produzir ativações iguais a 0 para aquelas unidades, ao mesmo tempo em que se mantém a mesma ativação para todas as demais unidades daquela camada. Uma vez que (1) cada exemplo moldado é muito provavelmente diferente do exemplo original correspondente (por conta da aplicação da máscara estocástica), e que (2) cada máscara diferente corresponde a um exemplo moldado de forma diferente, então é intuitivo concluir que o procedimento de desligamento produz resultado semelhante a aumentar a quantidade de exemplos de treinamento.

3.6.4. Normalização em Lote

Uma transformação usual durante o treinamento de uma RNA é normalizar o conjunto de dados de treinamento de acordo com a distribuição normal padrão (i.e., média igual a zero e variância igual a 1) para evitar problemas de comparação devido às diferentes escalas usadas nos dados. Ocorre que durante a passagem dos exemplos normalizados através das camadas da rede, esses valores são novamente transformados (por meio das pré-ativações e ativações), podendo fazer com que os dados de entrada de algumas camadas ocultas fiquem desnormalizados novamente. Esse problema, conhecido como *mudança de co-*

variável interna (*internal covariate shift*), é tanto mais grave quanto mais profunda for a rede a ser treinada. Esse problema aumenta o tempo de treinamento porque implica na definição de uma taxa de aprendizagem pequena, além de propiciar a dissipação dos gradientes.

A *Normalização em Lote* (*Batch Normalization*) [56] é um mecanismo proposto recentemente para resolver o problema acima, e que consiste em normalizar os dados fornecidos a cada camada oculta. A normalização é aplicada em cada *mini-lote*.[14], para aumentar a eficiência durante a aplicação da transformação.

De acordo com os experimentos realizados pelos autores da técnica [56], ela também produz um efeito de regularização sobre o treinamento, em alguns casos eliminando a necessidade de aplicar o *desligamento* (Seção 3.6.3). Uma aceleração significativa do tempo de treinamento também foi observada, resultante da diminuição de 14 vezes na quantidade de passos de treinamento necessários, quando comparada ao tempo de treinamento sem o uso da normalização.

3.7. Considerações Finais

Um aspecto importante das técnicas de aprendizagem profunda que não foi abordado neste Capítulo diz respeito à eficiência computacional do processo de treinamento. Embora a aplicação de técnicas de paralelismo para agilizar o processamento, tais como o uso de unidades gráficas de processamento (*graphics processing units*, GPU), não seja recente (ver, e.g., [57]), o tamanho dos conjuntos de treinamento atuais e a quantidade imensa de parâmetros envolvidos no treinamento de arquiteturas profundas tornou esse uso uma necessidade. Por exemplo, em 2012, Le et al [23] realizaram uma série de experimentos que envolvem a aplicação de várias técnicas de paralelismo e de processamento assíncrono para treinar redes neurais que geral modelos com 1 bilhão de conexões, a partir de um conjunto de dados com 10 milhões de imagens, cada uma das quais com 200×200 pixels. Em [58], os autores apresentam experimentos sobre um sistema paralelo capaz treinar redes com 1 bilhão de parâmetros em 3 máquinas em um período de dois dias. Os autores também apresentam evidências de que é possível realizar o treinamento de redes com mais de 11 bilhões de parâmetros em 16 máquinas. O uso de bibliotecas especializadas que realizem de forma eficiente operações comuns em aprendizagem profunda (tais como operações sobre arranjos multidimensionais) e que tirem proveito de hardware para paralelismo é também uma tendência. Ferramentas como Torch[15], Caffe[16], TensorFlow[17] e MXNet[18] são alternativas populares.

Outra área promissora também não abordada neste Capítulo é a *Aprendizagem Profunda por Reforço* (*Deep Reinforcement Learning*). Por meio de técnicas dessa área de pesquisa, robôs podem ser treinados para navegar por um ambiente recebendo como estímulo apenas pixels de imagens desse ambiente [59]. Agentes de software também

[14] Um *mini-lote* (*mini-batch*) de treinamento é uma amostra aleatória do conjunto de treinamento. Valores comumente escolhidos para o tamanho de um mini-lote estão entre 50 e 256.

[15] http://torch.ch/

[16] http://caffe.berkeleyvision.org/

[17] https://www.tensorflow.org/

[18] https://mxnet.readthedocs.io/

foram treinados para realizar diversas tarefas complexas: jogar Go no nível de um mestre mundial, um desafio antigo da IA apenas recentemente sobrepujado [60]; aprender a jogar um jogo clássico (*arcade*) por meio da análise de imagens capturadas desse jogo e do resultado de ações [61], a princípio, aleatórias.

Outra tendência de pesquisa na área que também não abordamos neste Capítulo é o *aprendizado multimodal* (e.g., redes que são treinadas com texto mais imagem, ou áudio mais vídeo, etc.). Dois trabalhos representativos dessa área são [62] e [63].

Classes de redes como as convolucionais e as recorrentes já existem há vários anos. Entretanto, apenas recentemente modelos construídos com essas arquiteturas de redes começaram a mostrar seu valor. Isso se deve a vários fatores. Em primeiro lugar, é cada vez maior a disponibilidade de coleções de dados suficientemente grandes para treinamento dessas redes. Outro fator importante é a disponibilidade de poder computacional para manipular essas quantidades imensas de dados. Finalmente, a pesquisa constante de técnicas melhores para mitigar o problema do sobreajuste (algumas das quais descritas na Seção 3.6) tem gerado frutos para o treinamento de arquiteturas profundas.

Para assegurar a continuidade desse progresso, é importante aumentar a compreensão das representações que são aprendidas pelas camadas internas dessas arquiteturas profundas. Os trabalhos de Andrej Karpathy [64] e Matthew Zeiler [25] são pesquisas nessa direção. Na mesma linha de continuidade do progresso, é importante que a comunidade continue a investigar melhores técnicas para aliviar o sobreajuste, mas que também analise as interdependências entres essas técnicas. Conforme mencionado na Seção 3.6, algumas das técnicas atualmente existentes eliminam a necessidade de usar outras, assim como há técnicas que fornecem resultados melhores quando aplicadas em conjunto. A interação entre os diversos procedimentos de treinamento ainda não é bem entendida pela comunidade. Também são necessárias mais investigações na linha de [65], no qual os autores investigam o comportamento de métodos de otimização em espaços de busca de grandes dimensionalidades.

Possivelmente, a grande fronteira a ser alcançada na pesquisa em aprendizagem profunda (e em aprendizagem de máquinas em geral) é projetar algoritmos que possam treinar sistemas com poucos dados e principalmente aproveitar de forma eficiente a existência de dados não supervisionados. A técnica de pré-treinamento não supervisionado (Seção 3.6.1) é um exemplo de uso de dados não supervisionados que em 2006 alavancou a pesquisa em aprendizagem profunda [45, 46, 21] mas que, de lá para cá, foi suplantada pela disponibilidade de cada vez mais dados supervisionados e pelo fato de abordagens supervisionadas funcionarem tão bem na presença de uma grande quantidade de dados dessa natureza. Uma criança não precisa ser apresentada a uma quantidade imensa de exemplos de um de determinado conceito para capturar a essência dele. É justamente essa capacidade de abstração extrema que falta aos métodos atuais usados para aprendizado não supervisionado. As redes convolucionais (Seção 3.4) e as autocodificadoras (Seção 3.3) são inspiradas no funcionamento do córtex cerebral [66]. Nos próximos anos, devemos esperar o aumento da pesquisa sobre métodos inspirados no funcionamento do cérebro para tirar proveito de dados não supervisionados de forma mais adequada.

Referências

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[3] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

[4] Li Deng and Dong Yu. Deep learning: Methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[5] Itamar Arel, Derek C. Rose, and Thomas P. Karnowski. Research frontier: Deep machine learning–a new frontier in artificial intelligence research. *Comp. Intell. Mag.*, 5(4):13–18, November 2010.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[8] Yoshua Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, January 2009.

[9] Simon Haykin. *Neural Networks: A Comprehensive Foundation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2007.

[10] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[11] James J. DiCarlo, Davide Zoccolan, and Nicole C. Rust. How does the brain solve visual object recognition? *Neuron*, 73(3):415 – 434, 2012.

[12] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *J. Mach. Learn. Res.*, 10:1–40, June 2009.

[13] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[14] Pascal et al Vincent. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *J. Mach. Learn. Res.*, 11:3371–3408, December 2010.

[15] Salah et al Rifai. Contractive Auto-Encoders: Explicit Invariance During Feature Extraction. In *ICML-11*, 2011.

[16] Alireza Makhzani and Brendan Frey. k-sparse autoencoders. In *ICLR 2014*, 2014.

[17] P. Baldi and K. Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural Netw.*, 2(1):53–58, January 1989.

[18] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In Isabelle Guyon, Gideon Dror, Vincent Lemaire, Graham W. Taylor, and Daniel L. Silver, editors, *ICML Unsupervised and Transfer Learning 2011*, volume 27 of *JMLR Proceedings*, pages 37–50. JMLR.org, 2012.

[19] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000.

[20] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006.

[21] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *NIPS 19*, pages 153–160. MIT Press, Cambridge, MA, 2007.

[22] Dumitru Erhan and et al. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11:625–660, 2010.

[23] Marc'aurelio Ranzato et al. Building high-level features using large scale unsupervised learning. In John Langford and Joelle Pineau, editors, *ICML-12*, pages 81–88, New York, NY, USA, 2012. ACM.

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

[25] Matthew D. Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*, pages 818–833. Springer International Publishing, Cham, 2014.

[26] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *ICCV 2009*, pages 2146–2153. IEEE, 2009.

[27] Y. et al LeCun. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, December 1989.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *NIPS 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[29] Olga Russakovsky et al. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

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

[31] P. Sermanet and Y. LeCun. Traffic sign recognition with multi-scale convolutional networks. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 2809–2813, July 2011.

[32] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *NIPS 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[33] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura, editors, *INTERSPEECH 2010*, pages 1045–1048. ISCA, 2010.

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

[35] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. *Empirical evaluation of gated recurrent neural networks on sequence modeling*. 2014.

[36] Alex Graves, Abdel-Rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6645–6649. IEEE, 2013.

[37] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *ICML-14*, pages 1764–1772, 2014.

[38] I. Sutskever, J. Martens, and G. Hinton. Generating text with recurrent neural networks. In L. Getoor and T. Scheffer, editors, *ICML-11*, ICML '11, pages 1017–1024, New York, NY, USA, June 2011. ACM.

[39] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[40] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[41] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhudinov. Unsupervised learning of video representations using lstms. In David Blei and Francis Bach, editors, *ICML-15*, pages 843–852. JMLR Workshop and Conference Proceedings, 2015.

[42] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. In David Blei and Francis Bach, editors, *ICML-15*, pages 1462–1471. JMLR Workshop and Conference Proceedings, 2015.

[43] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In David Blei and Francis Bach, editors, *ICML-15*, pages 2048–2057. JMLR Workshop and Conference Proceedings, 2015.

[44] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In Kremer and Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

[45] Geoffrey E. Hinton. To recognize shapes first learn to generate images. In *Computational Neuroscience: Theoretical Insights into Brain Function*. Elsevier, 2007.

[46] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[47] Marc'Aurelio Ranzato, Christopher S. Poultney, Sumit Chopra, and Yann LeCun. Efficient learning of sparse representations with an energy-based model. In Bernhard Schölkopf, John C. Platt, and Thomas Hofmann, editors, *NIPS 19*, pages 1137–1144. MIT Press, 2006.

[48] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *ICML-10*, pages 807–814. Omnipress, 2010.

[49] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon and David B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 1026–1034, 2015.

[51] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[52] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[53] Pierre Baldi and Peter J Sadowski. Understanding dropout. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *NIPS 26*, pages 2814–2822. Curran Associates, Inc., 2013.

[54] Kishore Reddy Konda, Xavier Bouthillier, Roland Memisevic, and Pascal Vincent. Dropout as data augmentation. *CoRR*, abs/1506.08700, 2015.

[55] D. van Dyk and X. L. Meng. The art of data augmentation (with discussion). *Journal of Computational and Graphical Statistics*, **10**:1–111, 2001.

[56] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In David Blei and Francis Bach, editors, *ICML-15*, pages 448–456. JMLR Workshop and Conference Proceedings, 2015.

[57] Kyoung-Su Oh and Keechul Jung. {GPU} implementation of neural networks. *Pattern Recognition*, 37(6):1311 – 1314, 2004.

[58] Adam Coates et al. Deep learning with cots hpc systems. In Sanjoy Dasgupta and David Mcallester, editors, *ICML-13*, volume 28, pages 1337–1345. JMLR Workshop and Conference Proceedings, May 2013.

[59] Fangyi Zhang, Jürgen Leitner, Michael Milford, Ben Upcroft, and Peter Corke. Towards vision-based deep reinforcement learning for robotic motion control. In *Australasian Conference on Robotics and Automation 2015*, Canberra, A.C.T, September 2015.

[60] David et al Silver. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[61] Volodymyr et al Mnih. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[62] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y. Ng. Multimodal deep learning. In Lise Getoor and Tobias Scheffer, editors, *ICML-11*, pages 689–696. Omnipress, 2011.

[63] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *CVPR*, 2014.

[64] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[65] Yann N. Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Z. Ghahramani, M. Welling, C. Cortes, N.d. Lawrence, and K.q. Weinberger, editors, *NIPS 27*, pages 2933–2941. Curran Associates, Inc., 2014.

[66] Bruno A. Olshausen and David J. Field. Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision Research*, 37(23):3311 – 3325, 1997.