DEEP LEARNING

UNIDADE II
FUNDAMENTOS DE *DEEP LEARNING*

Elaboração

Natasha Sophie Pereira

Produção

Equipe Técnica de Avaliação, Revisão Linguística e Editoração

SUMÁRIO

FUNDAMENTOS DE *DEEP LEARNING*

UNIDADE II

O termo **Aprendizado Profundo**, ou *Deep Learning* (DL) como é mais conhecido, foi introduzido no campo do Aprendizado de Máquina, em 1986, por Rina Dechter, em seu artigo intitulado "*Learning while Searching in Constraint-Satisfaction-Problems*" (DECHTER, 1986), porém, a expressão só começou a se destacar a partir da década de 2000, quando o termo começou a ser destaque em trabalhos científicos importantes. Nos últimos anos, as técnicas desenvolvidas a partir de pesquisas com *Deep Learning* já estão presentes em muitos trabalhos de processamento de sinais (imagens, fala, áudio) e de informações (texto), incluindo aspectos importantes de aprendizado de máquina e inteligência artificial.

Veja alguns importantes trabalhos na área de *Deep Learning*:

- ***Learning while Searching in Constraint-Satisfaction-Problems*** (DECHTER, 1986).
- ***A Fast Learning Algorithm for Deep Belief Nets*** (HINTON; OSINDERO; TEH, 2006).
- ***Learning Multiple Layers of Representation*** (HINTON, 2007).
- ***Learning deep architectures for AI*** (BENGIO, 2009).
- ***Deep Machine Learning – a New Frontier in Artificial Intelligence Research*** (AREL; ROSE; KARNOWSKI, 2010).
- ***An Overview of Deep-Structured Learning for Information Processing*** (DENG, 2011).
- ***Deep Learning and Its Applications to Signal and Information Processing*** (YU; DENG, 2011).
- ***Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks*** (LEE *et al.*, 2011).
- ***Deep Neural Networks for Acoustic Modeling in Speech Recognition*** (HINTON *et al.*, 2012).
- ***Recent Advances in Deep Learning for Speech Research at Microsoft*** (DENG *et al.*, 2013).

- ***Representation Learning:*** *a Review and New Perspectives* (BENGIO; COURVILLE; VINCENT, 2013).
- ***ImageNet Classification with Deep Convolutional Neural Networks*** (KRIZHEVSKY; SUTSKEVER; HINTON, 2017).

Deep Learning trata de arquiteturas que processam informações e sinais através de várias camadas, e não do profundo entendimento de informações ou sinais. Também é necessário observar que a Psicologia Educacional utiliza o termo para descrever uma abordagem de aprendizado caracterizada pelo engajamento ativo, motivação intrínseca e busca pessoal por significado (OHLSSON, 2011). Devemos tomar cuidado para não confundir o termo computacional com a definição da psicologia em nossas pesquisas.

Nesta unidade será abordada uma visão geral sobre *Deep Learning*, apresentando seu funcionamento, definições e tendências para essa tecnologia.

CAPÍTULO 1

O QUE É *DEEP LEARNING*

Desde a concepção dos primeiros computadores, as pessoas já se perguntavam se eles seriam capazes de pensar. Atualmente, a Inteligência Artificial é um campo de pesquisa muito amplo e em constante desenvolvimento, que pode ser aplicado nas mais diversas áreas e problemas. Já estamos acostumados com sistemas inteligentes que nos auxiliam nas rotinas do dia a dia, que reconhecem fala ou imagens, que fazem diagnósticos médicos, e principalmente, que nos auxiliam em nossa vida acadêmica e científica.

Os computadores realizam facilmente tarefas que consideramos muito complexas, como a realização de cálculos ou problemas de lógica, porém, o maior desafio na AI é resolver tarefas que são fáceis e rotineiras para os humanos, e, por serem tão intuitivas, são difíceis de serem descritas formalmente, por exemplo, reconhecer um objeto em uma imagem, ou entender frases de uma pessoa com muito sotaque.

As primeiras aplicações de sucesso da AI não requeriam do computador nenhum conhecimento do mundo real, era apenas um conjunto muito restrito de regras matemáticas (como é o caso de *Deep Blue*, o primeiro computador a vencer um ser humano em uma partida de xadrez). Ao contrário da implementação manual de regras a serem seguidas pelo computador, as tarefas rotineiras do ser humano requerem uma enorme quantidade de conhecimentos gerais do universo em que estamos inseridos, sendo que grande parte deste conhecimento é adquirido de forma subjetiva e intuitiva, o que torna muito difícil de se explicar de maneira formal. Então, pesquisadores entenderam que os computadores deveriam adquirir estes conhecimentos para se comportar de maneira inteligente. O principal desafio da Inteligência Artificial é descobrir formas de fazer com que o computador consiga adquirir este conhecimento geral do mundo real.

As dificuldades em codificar o conhecimento do mundo real para gerar sistemas inteligentes levaram os pesquisadores a definirem que os sistemas precisavam adquirir seu próprio conhecimento, de forma automática, ao extrair informações de dados brutos. A capacidade dos sistemas de adquirir conhecimento é conhecida como Aprendizado de Máquina. A inserção de conceitos de ML no desenvolvimento de sistemas levou o computador a conseguir resolver problemas que envolvem conhecimento do mundo real, sendo capazes de tomar decisões aparentemente subjetivas. Porém, a performance destes algoritmos depende inteiramente da representação dos dados de entrada.

Muitas tarefas envolvendo AI podem ser resolvidas apenas definindo o conjunto correto de características que devem ser extraídas para aquela tarefa e, então, aplicando em um algoritmo simples de ML. Porém, para algumas tarefas, é difícil definir este conjunto de características a serem extraídas. Uma solução seria utilizar os algoritmos de Aprendizado

de Máquina para descobrir não apenas a resposta baseada em características predefinidas, mas descobrir, também, as próprias características que levam ao resultado esperado. Essa abordagem é conhecida como Aprendizado de Representação (do inglês, *Representation Learning*). Aprender a representar as características que levam a um resultado geralmente leva a uma performance melhor que a definição manual, além disso, permite que o algoritmo se adapte mais facilmente para solução de novas tarefas e agiliza o processo de seleção das características apropriadas.

O principal exemplo de algoritmos para Aprendizado de Representação são os chamados autocodificáveis (do inglês *Autoencoders*), que transformam os dados brutos de entrada em características que os representam (função codificadora, chamada *encoder*), e então, voltam os dados para o formato original (função decodificadora, chamada *decoder*). Eles são programados para guardar a maior quantidade possível de informação durante este processo. Quando desenvolvemos algoritmos para aprender características de dados, é muito importante definir os fatores de variação, que influenciam na definição das características, ou seja, são informações que auxiliam na abstração das características dos dados e suas variações. Infelizmente, da mesma forma que os fatores de variação ajudam a determinar uma solução, eles podem influenciar em todas as características, o que torna o processo complicado.

Como o Aprendizado de Representação nem sempre consegue solucionar uma tarefa devido à grande quantidade de características e fatores de variação presentes, surgiu o *Deep Learning* para solucionar esse problema. DL quebra a representação de características em diversas partes menores, que representam características mais simples, permitindo ao computador definir conceitos complexos a partir de conceitos mais simples.

Deng e Yu (2013) definem *Deep Learning* como um conjunto de técnicas de Aprendizado de Máquina que explora o processamento de diversas camadas de informação não linear utilizando técnicas supervisionadas e não supervisionadas para extração e transformação, e para análise de padrões e classificação.

Então, é possível definir *Deep Learning* a partir de duas características principais:

- Os modelos computacionais que representam *Deep Learning* consistem em múltiplas camadas (ou estágios) de processamento de informações não lineares.
- Os métodos de aprendizado supervisionado ou não supervisionado de representação de características agem sucessivamente em camadas mais altas e mais abstratas.

Pode-se dizer que as técnicas de *Deep Learning* estão na intersecção das áreas de pesquisa de Inteligência Artificial, Aprendizado de Máquina, Redes Neurais, Modelagem Gráfica, Otimização, Reconhecimento de Padrões e Processamento de Sinais.

O exemplo clássico de modelo *Deep Learning* são as redes *Feedforward* ou *Multilayer Perceptron* (MLP), que nada mais é que uma função matemática que mapeia dados de entrada em saídas, e é composta por diversas funções mais simples. Além de aprender a melhor representação para os dados, a quantidade de camadas em *Deep Learning* permite ao computador aprender passo a passo, de modo que, a cada nova camada, o aprendizado adquirido na camada anterior é levado em consideração.

Não existe uma quantidade exata de camadas que determina se uma rede é ou não de aprendizado profundo, porém, *Deep Learning* pode ser considerado como o estudo de modelos que envolvem uma grande quantidade de funções de aprendizado que levam ao aprendizado de máquina.

Os estudos voltados para *Deep Learning* vêm aumentando muito nos últimos anos devido, principalmente, ao aumento do poder de processamento computacional (com o uso de GPGPUs, do inglês *General-Purpose Graphical Processing Units*), ao aumento significante da quantidade de dados disponíveis para treinamento das redes e aos avanços em pesquisas nas áreas de Aprendizado de Máquina e Processamento de Sinais e Informações. Estes avanços permitiram que os métodos de DL explorassem o funcionamento das funções não lineares, possibilitando o aprendizado de representação de características de forma distribuída e hierárquica, possibilitando, inclusive, o uso efetivo de dados rotulados e não rotulados.

As principais entidades que estudam efetivamente *Deep Learning* são as Universidades de Toronto, Nova Iorque, Montreal, Stanford, além de universidades brasileiras, como a USP e a UnB, entre outras; *Microsoft Research* (desde 2009); *Google* (Desde 2011); *IBM Research* (desde 2011); *Facebook* (desde 2013), e outras centenas de laboratórios espalhados pelo mundo[1].

As principais áreas de pesquisa utilizando *Deep Learning* são:

- Visão Computacional.
- Reconhecimento e classificação de Fala (reconhecimento fonético, busca por voz, reconhecimento de conversa, classificação de semântica etc.).
- Decodificação de Imagens e Fala.

1 Veja uma lista com vários Grupos de Pesquisa em Deep Learning no site Deep Learning.net: http://deeplearning.net/deep-learning-research-groups-and-labs/.

» Entendimento de Linguagem Natural.

» Robótica.

» Análise de moléculas e descoberta de novos medicamentos.

Alguns dos principais *sites* que trazem informações sobre *Deep Learning*, com disponibilização de materiais e dicas:

» http://deeplearning.net/: Infelizmente, parou de ser atualizado em 2006, porém, traz uma lista grande de tutoriais, softwares e bancos de dados na área.

» http://www.deeplearningbook.org/: Disponibilização do livro "*Deep Learning*" (GOODFELLOW; BENGIO; COURVILLE, 2016) pela MIT Press, além do livro, o site traz listas de exercícios e indicação de materiais.

» http://deeplearningbook.com.br/: *Site* mantido pela Data Science Academy, traz um livro em português, disponibilizando capítulos semanais. O *site* ainda traz indicação de cursos on-line, e indicações de leituras complementares a cada capítulo.

» http://ufldl.stanford.edu/: *Site* mantido pelo Departamento de Ciência da Computação da Universidade de Stanford, traz um amplo tutorial sobre DL, além de indicações de materiais complementares e exercícios a cada conteúdo.

Deep Learning é uma área da Inteligência Artificial, mais especificamente de Aprendizado de Máquina, que é uma técnica que permite que os sistemas evoluam com sua própria experiência e com a análise de dados. Segundo Goodfellow, Bengio e Courville (2016), o Aprendizado de Máquina é a única abordagem que permite que os computadores atuem em tarefas complexas do mundo real. *Deep Learning*, por sua vez, é um tipo especial de Aprendizado de Máquina que alcança flexibilidade e grande performance com o aprendizado das representações do mundo real através de uma sequência de camadas hierárquicas de conceitos, onde cada conceito pode ser entendido como a relação entre conceitos mais simples, e representações mais abstratas processadas a partir de representações menos abstratas. Para compreender melhor a relação entre os diferentes tipos de Inteligência Artificial e suas etapas, veja o fluxograma apresentado na Figura 10.

Os conceitos de *Deep Learning* foram introduzidos no campo da AI e ML para suprir algumas necessidades, e evoluíram devido a fatores cruciais de *Hardware* e disponibilização de dados. Alguns dos pontos-chave para o *Deep Learning* ser tão popular atualmente são:

» *Deep Learning* vem de uma longa história desde a concepção dos primeiros computadores, já foi chamado por vários nomes que refletiam diferentes pontos de

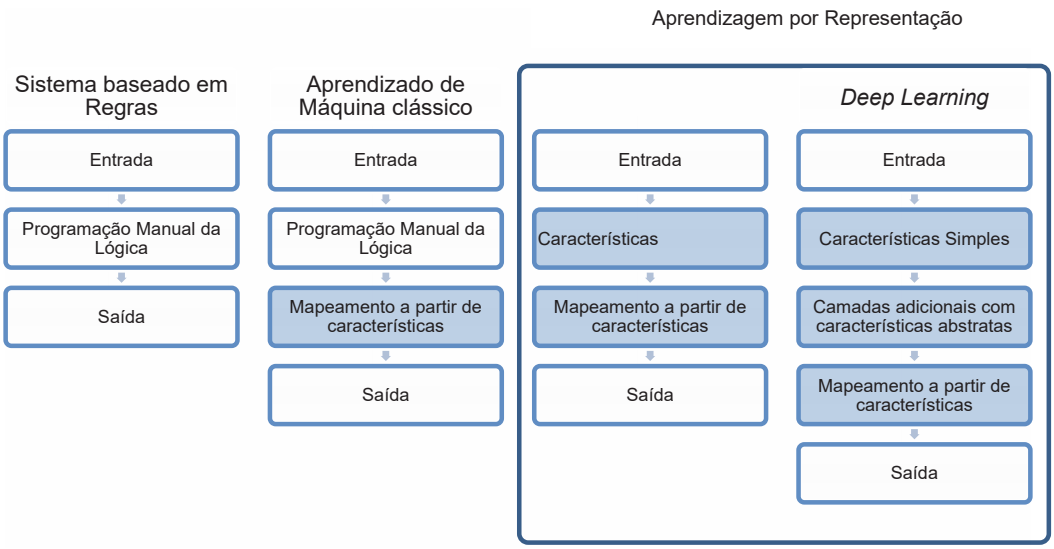
vista sobre sua utilização, passou por momentos de muita popularidade, e outros de desprezo pela ciência.

» As técnicas de *Deep Learning* se tornaram mais usuais a partir do aumento significativo da quantidade de dados disponíveis.

» Os modelos de *Deep Learning* aumentaram de tamanho (quantidade de camadas e de neurônios por camadas) à medida que o poder de processamento computacional aumentou, permitindo a melhoria de sua infraestrutura.

» A cada dia, técnicas de *Deep Learning* vêm sendo utilizadas para solucionar problemas rotineiros mais complexos.

Figura 10. Gráfico apresentando as etapas de cada um dos diferentes tipos de sistemas de Inteligência Artificial. As caixas sombreadas indicam etapas com capacidade de aprender com os dados.



Fonte: Adaptado de Goodfellow; Bengio; Courville, 2016.

Há alguns anos, as arquiteturas rasas, também conhecidas como aprendizado raso ou *Shallow Learning*, dominavam as pesquisas de Aprendizado de Máquina e Processamento de Sinais, elas são chamadas rasas pois trabalham com apenas uma ou duas camadas para o processamento de características não lineares. Alguns dos principais métodos de *Shallow Learning* são:

» Modelo de Misturas Gausianas, ou *Gaussian Mixture Model* (GMM).

» *Conditional Random Fields* (CRF).

» Modelo de Máxima Entropia, ou *Maximum Entropy Model* (MaxEnt).

» Máquinas de Vetor de Suporte, ou *Support Vector Machine* (SVM).

- Regressão Logística (*Logistic Regression*).
- Perceptrons Multicamadas, ou *Multilayer Perceptrons* (MLP) com uma única camada oculta.

O *Shallow Learning* se mostra eficiente para a solução de problemas simples, com escopo bem definido, mas essa metodologia possui baixa capacidade representacional e de modelagem, o que pode gerar muitas dificuldades no manuseio de problemas mais complexos do mundo real, em especial se estes problemas envolvem sinais naturais como fala, linguagem, visão, que são características humanas. A necessidade de manuseio dessas características naturalmente humanas abriu espaço para a utilização das arquiteturas profundas, conhecidas como *Deep Learning*.

Deep Learning é uma técnica que foi desenvolvida na década de 1940, porém, muitos pensam que é uma tecnologia recente. Isso é devido à sua baixa popularidade até alguns anos atrás, e aos muitos nomes que já assumiu no decorrer da história da evolução da Inteligência Artificial. De acordo com Goodfellow, Bengio e Courville (2016), *Deep Learning* passou por três grandes fases, quando assumiu os seguintes nomes: I) Cibernética, entre as décadas de 1940 e 1960; II) conexionismo entre as décadas de 1960 a 1990; III) *Deep Learning*, a partir da década de 2000.

Os conceitos de *Deep Learning* se originaram no campo de pesquisa das Redes Neurais Artificiais (ANN, do inglês *Artificial Neural Networks*), e são comumente chamados de "Nova geração das Redes Neurais". Redes neurais *feedforward* ou MLPs com várias camadas ocultas são exemplos de Redes Neurais Profundas, ou *Deep Neural Network* (DNN). As técnicas de *Backpropagation* (BP) também podem ser consideradas como DNN, apesar de não ter funcionado bem sozinho no aprendizado de redes com muitas camadas ocultas. Porém, apesar deste princípio voltado para os modelos neurais, atualmente *Deep Learning* vai além da perspectiva de replicar o modelo biológico do cérebro, e busca um modelo mais genérico para aprendizado em vários níveis de composição (GOODFELLOW; BENGIO; COURVILLE, 2016) baseado em modelos não necessariamente inspirados nos modelos neurais.

Componentes básicos

Antes de estudarmos o funcionamento de uma rede profunda, vamos revisar alguns componentes básicos. Patterson e Gibson (2017) apresentam estes conceitos já com um viés de aplicação em *Deep Learning*:

Parâmetros

Em Aprendizado de Máquina simples, os parâmetros estão relacionados com o vetor x de parametros na equação $Ax = b$. Em Redes Neurais, os parâmetros estão diretamente relacionados com os pesos nas conexões da rede. Nestes casos, o vetor x de parâmetros é multiplicado pontualmente pela matriz A a fim de obter o vetor b de saída. Quanto mais próximo o vetor b é dos valores de treinamento, maior a acurácia do modelo. A fim de minimizar a perda, utilizamos métodos de otimização (como o Gradiente Descendente, por exemplo) para encontrar os valores mais adequados para o vetor x.

Em *Deep Learning* ainda existem os vetores de parâmetros que representam os pesos nas conexões e que precisamos otimizar, porém, a grande diferença é a forma como as camadas estão conectadas, que difere de acordo com a arquitetura escolhida. Essa diferença nas arquiteturas das redes determina como os parâmetros serão selecionados.

Camadas

Já vimos anteriormente como funcionam as camadas de entrada, camadas ocultas e camadas de saída em uma rede neural. No caso de *Deep Learning*, as redes possuem um número maior de camadas ocultas. É possível ajustar as camadas alterando as funções de ativação que são utilizadas por elas, e cada tipo de camada necessita de hiperparâmetros específicos de acordo com a arquitetura escolhida, para possibilitar que a rede consiga aprender.

A arquitetura da rede define como os diversos tipos de camadas são combinados para atingir determinado objetivo. Dependendo da arquitetura da rede, essas camadas se relacionam entre si, os neurônios presentes em uma camada podem se relacionar com outros neurônios da mesma camada, ou, ainda, em algumas arquiteturas, uma camada pode ser uma sub-rede. Enfim, as camadas são componentes essenciais quando se estuda *Deep Learning*.

Funções de ativação

As funções de ativação devem ser escolhidas de acordo com a Arquitetura da rede para extrair características específicas dos dados de entrada. Em *Deep Learning*, as características aprendidas a partir dos dados são transformações não lineares aplicadas à saída da camada anterior (que, geralmente, se torna a entrada da camada corrente). Este fato permite que a rede aprenda padrões dentro de um contexto. Patterson e Gibson (2017) trazem exemplos de funções de ativação para cada tipo de arquitetura:

» **Função de Ativação para Arquiteturas Gerais:** as funções de ativação, geralmente, são aplicadas em dois tipos de camadas, as camadas ocultas e as

camadas de saída. Não é usual utilizar funções de ativação nas camadas de entrada. As camadas ocultas têm a função de extrair de forma progressiva as características predominantes nos dados de entrada, e o conjunto de funções de ativação escolhido depende da arquitetura.

- **Funções de Ativação para Camadas Ocultas:** as mais comumente utilizadas são a Sigmoid, a Tanh, a Hard Tanh e a Unidade linear restrita (ReLU) e suas variantes. Já estudamos as características destas funções anteriormente no caderno. Para dados de entrada que são apresentados de forma contínua, a função ReLU é a mais indicada. Porém, caso a rede não tenha muitas camadas, também é possível utilizar a Tanh, quando a ReLU não estiver obtendo resultados muito satisfatórios.
- **Camadas de Saída para Regressão:** neste caso, é necessário observar qual tipo de resposta esperamos que nossa rede gere. Se a saída esperada é um único valor numérico, basta utilizar uma função linear de ativação na camada de saída.
- **Camadas de Saída para Classificação Binária:** neste caso, o indicado é usar uma camada de saída com um único neurônio e função Sigmoid, a fim de obter valores entre 0.0 e 1.0 (exceto os própios valores 0 e 1). Os valores obtidos são a probabilidade de os dados pertencerem a uma classe (Distribuição Probabilística).
- **Camadas de Saída para Classificação Multiclasse:** no caso de um problema com várias classes possíveis para enquadramento da entrada, desejamos obter a probabilidade de esta entrada pertencer a cada classe. Neste caso, a função Softmax é a mais indicada com uma função arg_max() para obter a maior pontuação para todas as classes. A função Softmax fornece a distribuição probabilística para todas as classes.
- **Camada de Saída para Classificações Múltiplas:** se a necessidade é obter como saída uma classificação múltipla (por exemplo, pessoa + carro), não se deve utilizar Softmax, mas, sim, uma camada com n neurônios e função Sigmoid, que fornecerá a distribuição probabilística (0.0 a 1.0) para cada classe de forma independente.

Funções de perda

As funções de perda quantificam a diferença entre a saída predita pela rede e a saída esperada (correta), também chamada de verdade. As funções de perda são utilizadas para penalizar a rede pela classificação incorreta de determinado vetor de entrada. As funções de perda geralmente são utilizadas em Regressão, Classificação e Reconstrução. Já vimos o que é regressão e o que é classificação. A Reconstrução é utilizada na extração de características de forma não supervisionada, e auxilia *Deep Learning* a atingir

excelentes valores de acurácia. Em algumas arquiteturas, a reconstrução, juntamente com a função de ativação apropriada, auxilia a rede na extração de características. Um exemplo seria usar entropia-cruzada multiclasse como função de perda junto com uma função de ativação softmax.

Métodos de otimização

Como já vimos anteriormente, treinar uma rede significa encontrar o melhor conjunto de valores para o vetor de parâmetros do modelo. Este conjunto de valores é obtido através da aplicação das Funções de Perda e punição da rede até que os valores preditos pela rede sejam próximos o suficiente da verdade, ou seja, cujas Funções de Perda retornam valores muito baixos. Já conhecemos o algoritmo de Gradiente Descendente, que é um algoritmo de otimização básico. Outros algoritmos mais indicados para *Deep Learning* são a Matriz Jacobiana (Otimização de primeira ordem) e a Matriz Hessiana (Otimização de segunda ordem).

- **Matriz Jacobiana:** é uma matriz de derivadas parciais dos valores da Função de Perda com relação a cada parâmetro. Ela possui uma derivada parcial para cada parâmetro, e o algoritmo assume a direção especificada pela Matriz Jacobiana.
- **Matriz Hessiana:** é calculada a derivada da matriz Jacobiana. Neste caso, é levada em consideração a interdependência entre os parâmetros ao escolher o quanto modificar cada um deles.

Hiperparâmetros

Os Hiperparâmetros são as configurações que o desenvolvedor da rede pode selecionar, porém, influenciam na performance da rede. Podem ser: o tamanho da camada, a magnitude, as regularizações, as ativações e funções de ativação, a estratégia para inicialização dos pesos, as funções de perda, as configurações para cada interação durante o treinamento (Tamanho dos minilotes), o esquema de Normalização para os dados de entrada (Vetorização) etc.

- **Tamanho da Camada:** o tamanho da camada é definido pela quantidade de neurônios que ela possui. O tamanho da camada de entrada está diretamente relacionado com a quantidade de características do vetor de entrada. O tamanho da camada de saída será 1 neurônio, n neurônios, onde n é a quantidade de classes possíveis para predição. O problema é decidir quantos neurônios as camadas ocultas devem possuir. Não há uma regra para definir um número máximo ou mínimo de neurônios que uma camada oculta deve possuir, porém, a complexidade

do problema deve estar diretamente relacionada à quantidade de neurônios das camadas ocultas.

Quanto maior a quantidade de neurônios, maior o custo de processamento. O peso das conexões são os parâmetros que devem ser treinados. À medida que incluímos mais parâmetros no modelo, aumentamos o esforço necessário para treinar a rede. Uma grande quantidade de parâmetros pode levar um longo tempo para treinamento e modelos que não conseguem convergir a um resultado. Outro problema é que uma camada com muitos neurônios pode convergir muito rapidamente devido ao *Overfitting*, ou seja, ela memoriza o conjunto de treinamento.

- **Magnitude:** são o Gradiente, a Taxa de Aprendizado e o *Momentum*, que é uma melhoria sobre o Gradiente Descendente simples.
  - **Taxa de Aprendizado:** é a velocidade com que alteramos o vetor de parâmetros à medida que nos movemos pela rede. Uma taxa de aprendizado muito alta permite atingir o objetivo mais rapidamente, porém, a possibilidade de a acurácia não ser boa é grande. Em contrapartida, uma taxa de aprendizado pequena demais pode demorar demais para completar o processo de treinamento, e pode tornar o algoritmo ineficiente. Sendo assim, é necessário analisar o conjunto de dados para escolher o tamanho certo para a taxa de aprendizado. É possível configurar algumas regras que reduzem a taxa de aprendizado com o tempo.

    - **AdaGrad:** é uma técnica desenvolvida para auxiliar o modelo a encontrar a taxa de aprendizado ideal. Ela reduz constantemente e nunca aumenta a taxa de aprendizado acima do que foi estipulado inicialmente. AdaGrad acelera o treinamento no começo e diminui a velocidade de acordo com a convergência, tornando o processo de treinamento mais suave.
    - **AdaDelta:** é uma variação do AdaGrad, porém, ao invés de acumular os valores, mantém apenas o histórico mais recente.
    - **ADAM:** é uma técnica de atualização que define a taxa de aprendizado a partir de uma estimativa do primeiro e segundo momentos do gradiente.

- ***Momentum*:** em alguns casos, o uso do Gradiente Descendente Estocástico (SGD) pode tornar o gradiente próximo a zero para alguns parâmetros fazendo com que os passos do SGD sejam muito pequenos, ou tornar o gradiente muito grande definindo passos muito grandes, dependendo da situação. Algumas técnicas como Nesterov's *momentum*, Adam ou AdaDelta, podem ser utilizadas a fim de minimizar estes problemas. É possível acelerar o treinamento aumentando o *momentum*, porém, é necessário reduzir a probabilidade de o modelo atingir o erro mínimo sem obter os valores ideais para os parâmetros. O *Momentum* é um fator entre 0.0 e 1.0 aplicado aos pesos ao longo das interações.

» **Regularização:** é uma forma de evitar o *Overfitting*. O *overfitting* acontece quando um modelo se torna especialista no conjunto de treinamento, mas não consegue generalizar estes conceitos para novas entradas. Sendo assim, os modelos que sofrem *overfitting* não conseguem fazer previsões para dados que não conhecem previamente. A regularização modifica o gradiente de modo que o modelo não fique tendencioso ao *overfitting*. As principais funções de regularização são o Dropout, DropConnect, L1 Penalty e L2 Penalty. As duas primeiras ocultam partes da entrada para cada camada, fazendo com que a rede neural possa aprender novas partes da entrada, aprendendo representações mais genéricas. Já as duas últimas evitam que os parâmetros da rede sejam muito grandes, através da redução dos pesos. A regularização adiciona novas variáveis ao cálculo normal do Gradiente.

  › **Dropout:** mecanismo usado para melhorar o treinamento de redes neurais ao omitir uma unidade oculta. Acelera o processo de treinamento, pois pula alguns neurônios aleatoriamente, então, estes neurônios não farão parte do treinamento nem do *Backpropagation*.

  › **DropConnect:** faz o mesmo que o Dropout, porém, ao invés de omitir unidades ocultas, ele ignora a conexão entre dois neurônios.

  › **L1:** multiplica o valor absoluto dos pesos, e não seus quadrados. Esta função transforma muitos pesos em zero e permite que outros aumentem, facilitando a interpretação dos pesos. É computacionalmente ineficiente em casos não esparsos, pois tem uma saída esparsa, e faz a seleção automática de características.

  › **L2:** multiplica metade da soma dos quadrados dos pesos por um coeficiente (chamado custo do peso). Melhora a generalização da rede e suaviza a saída do modelo a partir da alteração das entradas; também auxilia a rede a ignorar os pesos que não são utilizados. É computacionalmente eficiente, pois trabalha com soluções analíticas e não possui saída esparsa, mas não faz seleção automática de características.

» **Minilotes:** ao invés de a entrada ser um grande vetor, permite que o vetor de entrada seja dividido em vetores menores que são apresentados ao sistema na fase de treinamento. Isso otimiza a utilização dos recursos de *hardware*, e permite a utilização de operações de álgebra linear nos dados vetorizados, por exemplo, a multiplicação entre matrizes.

Com estes conceitos definidos, é possível compreender melhor como funciona uma rede profunda.

CAPÍTULO 2

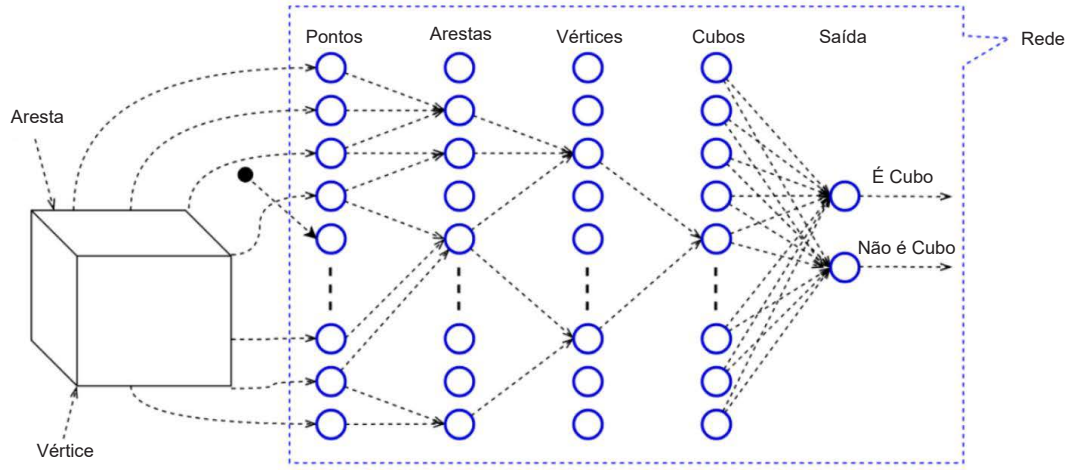
FUNCIONAMENTO

Uma rede profunda não aprende apenas a reconhecer determinado objeto, mas aprende quais características estão presentes naquele objeto e aprende a representar o objeto a partir dessas características. Quando a rede neural aprende as características básicas dos objetos, ela passa a ser capaz de classificar estes objetos baseando-se nas suposições que fez a partir deste aprendizado.

Aprendizado de características

Para entender o funcionamento dos algoritmos de *Deep Learning*, vamos observar uma figura geométrica simples, um cubo, por exemplo (Figura 11). O cubo é composto por arestas (ou linhas) que se cruzam nos vértices. Imagine que cada ponto da imagem está relacionado a um neurônio da rede. Então, todos os neurônios estariam representados na primeira camada, ou seja, a camada de entrada de uma rede *Feedforward* multicamadas. Um neurônio da camada de pontos é ativado se em alguma linha houver um ponto correspondente ao representado pelo neurônio. Os diversos pontos que estão na mesma linha formam uma aresta, então, eles são representados pelo mesmo neurônio da camada de arestas (próxima camada), consequentemente, não podem ser representados pelos outros neurônios dessa camada. A não ser pontos que se encontram nos vértices, estes são representados por três neurônios diferentes na camada de arestas. Dessa forma, temos duas camadas com diferentes níveis de abstração, a primeira define os pontos, e a segunda as arestas. Porém, apenas pontos e arestas sozinhos não definem um cubo completo.

Figura 11. Abstração de uma Rede Neural representando um cubo. Diferentes camadas codificam recursos com diferentes níveis de abstração.



Fonte: Adaptado de Vasilev *et al.*, 2019, p. 72.

Vamos adicionar mais uma camada para representar os vértices (terceira camada). Nesta camada, três neurônios ativados na camada de arestas, que formam um vértice, serão representados pelo mesmo neurônio na camada de vértices. Como uma aresta em um cubo está ligada a dois vértices, cada neurônio ativado na camada de arestas será representado por dois neurônios na camada de vértices. Ainda assim, será necessária uma última camada, que definirá os cubos (quarta camada). Nesta camada, todos os neurônios ativados da camada de vértices que formam o mesmo cubo serão representados por um mesmo neurônio na camada cubos.

Este exemplo simples nos possibilita tirar algumas conclusões. Uma delas é que as redes neurais profundas são muito eficientes para dados hierarquicamente organizados, como imagens (*pixels*, linhas, bordas, regiões etc.) e texto (caracteres, palavras, frases etc.), por exemplo. Ao contrário do que fizemos no exemplo, onde cada camada especificamente representa uma característica do objeto, as redes *Deep Learning* aprendem quais características devem ser representadas por quais camadas de forma automática durante o treinamento. Isso torna as camadas mais abstratas e menos sensíveis a ruídos.

Redes profundas

Podemos definir *Deep Learning* como um tipo de Aprendizado de Máquina que processa informações em camadas hierárquicas a fim de aprender suas representações e características e níveis crescentes de complexidade. Na prática, todos os algoritmos de *Deep Learning* são Redes Neurais que compartilham algumas propriedades básicas, visto que todos consistem em vários neurônios interconectados e organizados em camadas. A diferença está na forma como os neurônios estão organizados (Arquitetura da rede) e na forma como são treinados. Vasilev *et al.* (2019) apresenta alguns dos algoritmos mais utilizados na atualidade:

- **Multilayer Perceptrons (MLPs):** é uma rede com *feedforward propagation*, com camadas totalmente conectadas, e pelo menos uma camada oculta.
- **Redes Neurais Convolucionais (CNNs):** do inglês *Convolutional Neural Networks* é uma rede neural *feedforward* com vários tipos de camadas especiais. Atualmente, é muito utilizada na visão computacional e no processamento de linguagem natural.
- **Redes Recorrentes:** este tipo de rede tem uma memória interna que é baseada em parte ou em todos os dados de entrada já fornecidos à rede. A saída é uma combinação do conhecimento que ela já possui em sua memória, ou seja, seu estado atual, com a nova entrada fornecida. A cada nova entrada fornecida, a memória da

rede é ajustada, e seu estado é alterado. Devido a essas características, esse tipo de rede é adequado a informações que funcionam de forma sequencial, como textos ou séries temporais.

» **Autoencoders:** é um tipo de algoritmo não supervisionado, onde o formato de saída é o mesmo formato de entrada, o que permite à rede aprender representações básicas.

Treinando redes profundas

É possível utilizar diferentes algoritmos para treinar uma rede neural. Porém, os mais utilizados são o Gradiente Descendente Estocástico (do inglês *Stochastic Gradient Descent*, SGD) juntamente com o *Backpropagation*. O *momentum* é uma melhoria sobre o Gradiente Descendente simples. A regra para o ajuste dos pesos em redes neurais é:

$$w \rightarrow w - \lambda \nabla \left(J(w) \right)$$

Onde λ é a taxa de aprendizado. Para inserir o *momentum*, deve-se adicionar outro parâmetro a essa equação. Primeiro, é calculado o valor de ajuste do peso:

$$\Delta w \rightarrow \mu \Delta w - \lambda \left(\nabla J(w) \right)$$

Então, atualiza-se o peso:

$$w \rightarrow w + \Delta w$$

A partir das equações acima, é possível perceber que $\mu \Delta w$ é o *momentum*. Δw representa o valor anterior da atualização de peso e μ é o coeficiente que define a relação entre o novo valor de peso e o anterior.

Bibliotecas populares de código aberto

Existem diversas bibliotecas de código aberto que permitem a criação de redes neurais profundas com Python sem a necessidade de escrever todo o código à mão. De acordo com Vasilev *et al.* (2019), algumas das mais populares são *TensorFlow*, *Keras* e *PyTorch*. Todas possuem algumas características semelhantes, como:

» A unidade básica de armazenamento de dados é o *Tensor*. Considere o tensor como uma generalização de uma matriz para dimensões mais altas. Matematicamente, a definição de um tensor é mais complexa, mas no contexto de bibliotecas de

aprendizagem profunda, elas são matrizes multidimensionais de valores de base. Um tensor é semelhante a um array NumPy e é composto do seguinte:

- Um tipo de dados básico de elementos tensores. Eles podem variar entre bibliotecas, mas geralmente incluem números reais de 16, 32 e 64 bits e números inteiros de 8, 16, 32 e 64 bits.
- Um número arbitrário de eixos (também conhecido como hierarquia, ordem ou grau do tensor). Um tensor 0D é apenas um valor escalar, 1D é um vetor, 2D é uma matriz, e assim por diante. Em redes profundas, os dados são propagados em lotes de n amostras. Isso é feito por motivos de desempenho, mas, também, se encaixa nos conceitos de Gradiente Descendente Estocástico. Nas bibliotecas de *Deep Learning*, o primeiro eixo de um tensor representa as diferentes amostras.
- Uma forma que é o tamanho (o número de valores) de cada eixo do tensor.

» Redes neurais são representadas como um gráfico computacional de operações. Os nós do gráfico representam as operações (soma ponderada, função de ativação, e assim por diante). As bordas representam o fluxo de dados, ou seja, a forma como a saída de uma operação serve como entrada para a próxima. As entradas e saídas das operações (incluindo as entradas e saídas de rede) são tensores.

» Todas as bibliotecas incluem diferenciação automática. Só é necessário definir a arquitetura da rede e as funções de ativação e a biblioteca automaticamente definirá quais os parâmetros para treinamento por *backpropagation*.

» Todas as bibliotecas utilizam linguagem Python.

» A grande maioria dos projetos de *Deep Learning* rodam sobre GPUs NVIDIA, visto que essas placas dão maior suporte a essas arquiteturas. Essas bibliotecas também. Para implementar operações da GPU, elas contam com o *toolkit* CUDA[2] juntamente com a biblioteca cuDNN[3]. A biblioteca cuDNN é uma extensão de CUDA específica para trabalhar aplicações *Deep Learning*.

TensorFlow

TensorFlow é a biblioteca mais popular em *Deep Learning*, que foi desenvolvida e é mantida pela Google. Não é necessário especificar para a biblioteca que ela deve usar uma GPU, se você tiver uma, a biblioteca tentará utilizá-la automaticamente. Porém, se você possui mais de uma GPU, é necessário definir especificamente quais operações devem ser feitas em qual GPU, ou apenas a principal será utilizada. A biblioteca TensorFlow tem

2 https://developer.nvidia.com/cuda-zone.

3 https://developer.nvidia.com/cudnn.

uma curva de aprendizado mais acentuada em comparação com as outras bibliotecas. Para definir isso, basta utilizar o seguinte código:

```
with tensorflow.device("/gpu:1"):
  # definição do modelo aqui
```

Veja um exemplo:

```
"/cpu:0": CPU principal de seu computador
"/gpu:0": GPU principal de seu computador, se houver alguma
"/gpu:1": GPU secundária de seu computador, se houver uma segunda
"/gpu:2": GPU terciária de seu computador, se houver uma terceira
```

Keras

Keras é uma biblioteca Python para rede neural de alto nível que é executada sobre o TensorFlow, o CNTK[4] ou o Theano. Neste material, utilizaremos Keras sobre TensorFlow. Com a biblioteca Keras é possível realizar experimentações rápidas, e ela é relativamente mais fácil de utilizar que o TensorFlow. Da mesma forma que o TensorFlow, a Keras detectará automaticamente a presença de uma GPU e tentar utilizá-la, se não houver GPU, ela utilizará a CPU. Para especificar qual processador utilizar, você pode importar o TensorFlow e usar o mesmo código:

```
with tensorflow.device("/gpu:1"):
  # definição do modelo aqui
```

PyTorch

PyTorch é uma biblioteca para *Deep Learning* baseada em Torch e desenvolvida pelo *Facebook*. Seu uso é relativamente fácil, e vem ganhando popularidade entre os desenvolvedores. Da mesma forma que as demais, ela detecta automaticamente a presença de uma GPU e a seleciona, ou executa em CPU se não houver GPU. Para selecionar manualmente o dispositivo, utilize o seguinte código:

```
# No começo do Script
device = torch.device("cuda:0" if torch.cuda.is_available()
  else "cpu")
...
# Então, sempre que você tiver um novo Tensor ou módulo, ele
# não será copiado se já estiver no dispositivo correto
```

4 https://github.com/Microsoft/CNTK.

```
input = data.to(device)
model = MyModule(...).to(device)
```

Veja onde consultar as bibliotecas apresentadas neste capítulo para obter informações adicionais:

TensorFlow: https://www.tensorflow.org/.

Keras: http://keras.io/.

PyThorch: https://pytorch.org/.

CAPÍTULO 3

EXEMPLOS DE USO

Processamento de texto: utilizando Keras para classificar dígitos escritos manualmente

Seguindo o exemplo apresentado por Vasilev *et al.* (2019) em seu livro, vamos utilizar a biblioteca Keras para classificar as imagens do conjunto de dados MNIST.

A base de dados MNIST, de dígitos escritos manualmente, foi desenvolvida pelos pesquisadores Yann LeCun (*Courant Institute*, NYU), Corinna Cortes (*Google Labs*, New York) e Christopher J. C. Burges (*Microsoft Research*, Redmond), para auxiliar pessoas que têm interesse em aprender técnicas de aprendizado de máquinas e métodos de reconhecimento de padrões em dados reais.

No *site*, além do conjunto de dados, os pesquisadores ainda apresentam um conjunto de diversos tipos de classificadores e trabalhos correlacionados, que podem auxiliar em muito seu estudo. Disponível em: http://yann.lecun.com/exdb/mnist/.

Este conjunto de dados é composto por 70.000 exemplos de dígitos escritos manualmente por pessoas diferentes. Como sugere o *site* da base de dados, os primeiros 60.000 exemplos serão utilizados para treinamento e os 10.000 restantes para testes. Alguns dos dígitos que a base MNIST fornece são apresentados na Figura 12.

Figura 12. Exemplo de dígitos escritos manualmente presentes na base de dados MNIST.

Fonte: Vasilev *et al.*, 2019, p. 85.

A biblioteca Keras importa a base de dados MNIST sem necessidade de que você faça o *download* diretamente no *site*. Então, o primeiro passo é obter a base de dados:

```
from keras.datasets import mnist
```

Em seguida, é necessário importar algumas classes que permitem a utilização da rede *Feedforward*.

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.utils import np_utils
```

Após obter a base de dados e importar as classes, é possível carregar o conjunto de treinamento e de teste. Vamos definir como `(X_train, Y_train)` as imagens e rótulos para treinamento e como `(X_test, Y_test)` as imagens e rótulos para teste:

```
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

É necessário que os dados sejam ajustados a fim de permitir sua utilização. X_train contém 60.000 imagens de 28x28 pixels e X_test contém 10.000. Para apresentar esses dados à rede, é necessário reorganizá-los individualmente em long arrays de 784 *pixels* cada, ao invés de matrizes bidimensionais de 28x28 *pixels*. Utilize o seguinte código para a conversão:

```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
```

Os rótulos indicam qual é o valor retratado na imagem. Vamos transformar o conjunto de rótulos individuais para cada entrada em um vetor codificado onde todos os elementos são zeros, exceto o elemento que representa o valor da imagem, estes vetores são chamados *one-hot-encoded*. Por exemplo, se a imagem representa o número 4, o vetor seria [0, 0, 0, 0, 1, 0, 0, 0, 0, 0], e assim por diante. Sendo assim, nossa rede terá 10 neurônios de possíveis saídas:

```
classes = 10
Y_train = np_utils.to_categorical(Y_train, classes)
Y_test = np_utils.to_categorical(Y_test, classes)
```

É necessário especificar o tamanho da camada de entrada (`input_size`), ou seja, o tamanho das imagens MNIST, o número de camadas de neurônios ocultas (`hidden_neurons`), o número de ciclos para treinamento da rede (`epochs`) e o tamanho dos pacotes (`batch_size`). Todas essas definições devem ser feitas antes da execução da função principal:

```
input_size = 784
batch_size = 100
hidden_neurons = 100
epochs = 100
```

Após a realização de todas as etapas acima, é possível definir a rede. Vamos utilizar um modelo sequencial, onde cada camada servirá de entrada para a próxima camada. Na biblioteca Keras, Dense significa que todas as camadas são totalmente conectadas (*fully-connected*). Para este exemplo, vamos utilizar uma rede com uma camada oculta, ativação *sigmoid* e saída *softmax*:

```
model = Sequential([
  Dense(hidden_neurons, input_dim=input_size),
  Activation('sigmoid'),
  Dense(classes),
  Activation('softmax')
])
```

A biblioteca Keras permite especificar de forma simples a função de custo (*loss function*) e sua otimização, através da entropia cruzada (*cross-entropy*) e do Gradiente Descendente Estocástico (SGD). Vamos manter os valores-padrão para a taxa de aprendizado, *momentum* etc:

```
model.compile(loss='categorical_crossentropy',
  metrics=['accuracy'], optimizer='sgd')
```

Função *Softmax* e Entropia cruzada

Softmax é um tipo de regressão logística para aplicação em múltiplas classes. Sua fórmula é representada por:

$$F(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

onde, i,j = 0, 1, 2, ..., n e x_i representa cada um dos n valores reais arbitrários, correspondentes às n classes mutuamente exclusivas.

A função *Softmax* encaixa o valor de entrada em um intervalo entre 1 e 0, de modo similar à função de logística, porém, no caso da *Softmax*, a soma de todos os valores encaixados neste intervalo deve resultar em 1. Dessa forma, a saída da função *Softmax* pode ser definida como uma distribuição de probabilidade normalizada das classes. A partir disso, é possível utilizar uma função de custo que compara a diferença entre a probabilidade estimada da classe e a distribuição real das classes, essa diferença é

conhecida como entropia cruzada. A distribuição real das classes geralmente é um vetor codificado, de modo que todos os elementos têm probabilidade 0, exceto o elemento que representa a classe, que é 1, ou seja, probabilidade de 100% de ser aquela classe. A função de custo para isso é chamada de custo de entropia cruzada:

$$H(p,q) = -\sum_{i=1}^{n} p_i(x)\log\log(q_i(x))$$

onde, $q_i(x)$ é a probabilidade estimada de uma saída pertencer a uma classe i (entre um total n de classes possíveis) e $p_i(x)$ é a probabilidade real. Quando são utilizados valores-alvos do tipo *one-hot-encoded* para $p_i(x)$, apenas a classe-alvo possui valor 1 e todas as demais possuem valor 0. Sendo assim, o custo de entropia cruzada calculará o erro apenas para a classe-alvo e descartará todos os demais erros.

Após especificar a função de custo, já é possível treinar a rede. Ao utilizar Keras, é possível realizar o treinamento de uma forma fácil com o método de ajuste (*fit method*):

```
model.fit(X_train, Y_train, batch_size = batch_size,
  nb_epoch=epochs, verbose=1)
```

Agora, basta adicionar um código para verificar a acurácia dos dados de teste:

```
score = model.evaluate(X_test, Y_test, verbose=1)
  print('Test accuracy:', score[1])
```

A acurácia do conjunto de teste será de aproximadamente 96%. Este não é um desempenho perfeito, mas é necessário considerar a rapidez de execução (aproximadamente 30s) em uma CPU. Algumas melhorias que poderiam ser feitas a fim de melhorar a acurácia seria aumentar a quantidade de camadas de neurônios ocultas ou aumentar a quantidade de ciclos para treinamento da rede.

Tente implementar essas melhorias, e veja se a acurácia será melhorada!

A fim de verificar o que a rede aprendeu, é possível visualizar os pesos das camadas ocultas. Para obter estes pesos, basta utilizar o seguinte código:

```
weights = model.layers[0].get_weights()
```

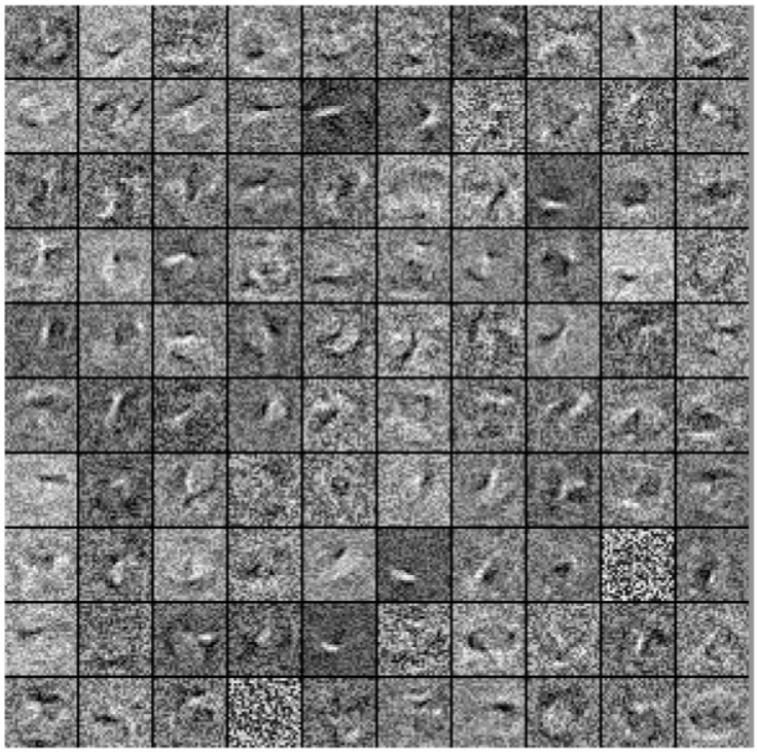
Para visualizar, é necessário reorganizar os neurônios novamente em matrizes bidimensionais de 28x28 *pixels*:

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy
fig = plt.figure()
w = weights[0].T
```

```
for neuron in range(hidden_neurons):
  ax = fig.add_subplot(10, 10, neuron + 1)
  ax.axis("off")
  ax.imshow(numpy.reshape(w[neuron], (28, 28)),
    cmap=cm.Greys_r)
plt.savefig("neuron_images.png", dpi=300)
plt.show()
```

Após executar estes comandos, será possível visualizar a imagem presente na Figura 13. Note que as imagens de todos os neurônios das camadas ocultas foram agrupadas em uma única imagem para facilitar a visualização. É perceptível, ao analisar a Figura 13, que cada neurônio aprendeu uma característica diferente do conjunto de dados apresentado à rede.

Figura 13. Apresentação do que foi aprendido em cada camada oculta da rede para classificação dos dados disponibilizados pela MNIST.



Fonte: Vasilev *et al.*, 2019, p. 89.

Processamento de imagem: utilizando Keras para classificar imagens de objetos

Vamos verificar outro exemplo apresentado por Vasilev *et al.* (2019) em seu livro. Neste caso, utilizaremos a biblioteca Keras para classificar imagens de objetos. Keras é uma

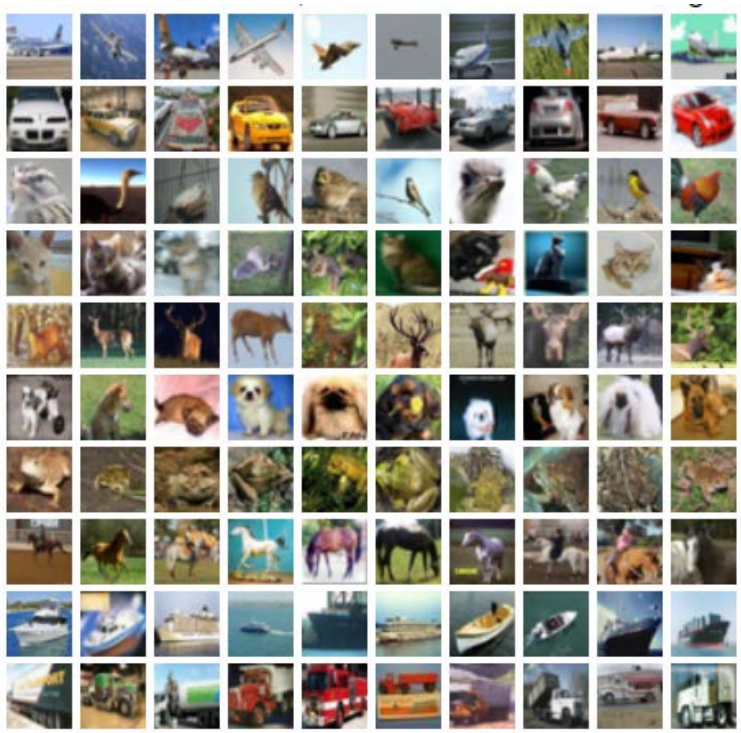
biblioteca que facilita a criação de redes neurais, mas também facilita o acesso aos dados de teste. Nesta seção, vamos utilizar os dados do CIFAR-10 (*Canadian Institute For Advanced Research*).

A base de dados CIFAR-10 é mantida pela Universidade de Toronto, no Canadá, e possui 60.000 imagens coloridas de 32x32 *pixels*, divididas em 10 classes com 6.000 imagens cada.

O *site* também possui a base de dados CIFAR-100, comporta por 100 classes com 600 imagens cada. Caso queira testar novas imagens. Disponível em: https://www.cs.toronto.edu/~kriz/cifar.html.

As classes do CIFAR-10 são: aviões, carros, pássaros, gatos, cervos, cachorros, sapos, cavalos, navios, caminhões. A Figura 14 nos mostra exemplos dessas classes, respectivamente, de cima para baixo.

Figura 14. Exemplo de imagens presentes na base de dados CIFAR-10.



Fonte: The CIFAR-10 dataset <https://www.cs.toronto.edu/~kriz/cifar.html>. Acesso em: 27/05/2019.

Assim como no MNIST, a base de dados CIFAR-10 é dividida em imagens para treinamento e imagens para teste. Neste caso, são 50.000 para treinamento e 10.000 para teste. Inicialmente, vamos importar as imagens do CIFAR-10, assim como fizemos com os dados do MNIST:

```
from keras.datasets import cifar10
from keras.layers.core import Dense, Activation
```

```
from keras.models import Sequential
from keras.utils import np_utils
```

Em seguida, vamos dividir os dados importados em 50.000 imagens para treinamento e 10.000 para teste. Da mesma forma, vamos remodelar os dados para um vetor unidimensional:

```
X_train, Y_train), (X_test, Y_test) = cifar10.load_data()

X_train = X_train.reshape(50000, 3072)
X_test = X_test.reshape(10000, 3072)

classes = 10
Y_train = np_utils.to_categorical(Y_train, classes)
Y_test = np_utils.to_categorical(Y_test, classes)

input_size = 3072
batch_size = 100
epochs = 100
```

Os dados contidos no MNIST são matrizes bidimensionais, pois são imagens em escala de cinza de dígitos manuscritos. Então, suas dimensões são linha e coluna, como vimos anteriormente, uma imagem do MNIST possui tamanho 28 (linhas) x 28 (colunas), totalizando 784 *pixels* por imagem.

Já os dados disponibilizados pelo CIFAR-10 são matrizes tridimensionais, pois, além de possuírem as dimensões linha x coluna, ainda possuem as bandas, que são constituídas por valores em tons de cinza para os canais R, G e B, formando as cores da imagem. Sendo assim, essas imagens possuem 32 (linhas) x 32 (colunas) x 3 (bandas), totalizando 3.072 *pixels* por imagem.

Outros tipos de imagens podem possuir diferentes quantidades de bandas.

Note que os dados disponibilizados pelo CIFAR-10 são mais complexos que os dados do MNIST, dessa forma, a rede deve refletir essa complexidade. Por este motivo, usaremos três camadas de neurônios ocultos para esse experimento, também usaremos mais neurônios:

```
model = Sequential([
  Dense(1024, input_dim=input_size),
  Activation('relu'),
```

```
    Dense(512),
    Activation('relu'),
    Dense(512),
    Activation('sigmoid'),
    Dense(classes),
    Activation('softmax')
])
```

Efetuaremos o treinamento da rede com mais um parâmetro, o `validation_data=(X_test, Y_test)`, este comando fará com que a rede utilize os dados de teste para validação da rede:

```
model.compile(loss='categorical_crossentropy',
metrics=['accuracy'], optimizer='sgd')
model.fit(X_train, Y_train, batch_size=batch_size,
    epochs=epochs, validation_data=(X_test, Y_test),
    verbose=1)
```

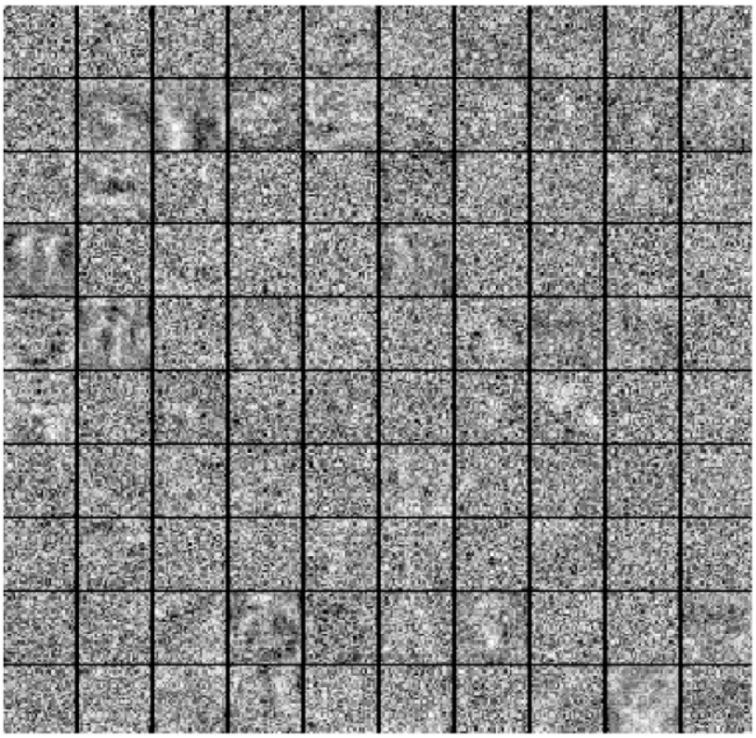
Para verificar o que a rede aprendeu, vamos visualizar os pesos dos 100 neurônios da primeira camada. Lembre-se de que é necessário reajustar os pesos para matrizes de tamanho 32x32, calculando a média dos valores presentes nos três canais para gerar uma imagem em tons de cinza (onde a tonalidade é a média entre os valores dos canais R, G e B):

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.gridspec as gridspec
import numpy
import random

fig = plt.figure()
outer_grid = gridspec.GridSpec(10, 10, wspace=0.0,
    hspace=0.0)

weights = model.layers[0].get_weights()

w = weights[0].T
```

```
for i, neuron in enumerate(random.sample(range(0, 1023),
    100)):
  ax = plt.Subplot(fig, outer_grid[i])
  ax.imshow(numpy.mean(numpy.reshape(w[i], (32, 32, 3)),
      axis=2), cmap=cm.Greys_r)
  ax.set_xticks([])
  ax.set_yticks([])
  fig.add_subplot(ax)

plt.show()
```

A partir destes comandos, é possível visualizar os pesos, conforme apresentado na Figura 15.

Figura 15. Apresentação do que foi aprendido em cada camada oculta da rede para classificação dos dados disponibilizados pela CIFAR-10.



Fonte: Vasilev *et al.*, 2019, p. 91.

Foi possível perceber que a fase de treinamento demorou mais do que quando treinamos a rede para classificar os dados do MNIST. Ao final deste experimento, foi possível obter uma acurácia de 60% durante o treinamento e cerca de 51% na fase de teste, apesar de termos desenvolvido uma rede maior. Isso se deu por conta da maior complexidade dos dados.

Note que são apresentados apenas os neurônios da primeira camada, e não é possível perceber o que eles aprenderam.

Tente visualizar os neurônios das demais camadas ocultas e, se possível, perceber o que eles aprenderam.

Neste capítulo, trabalhamos com duas bases de dados, a MNIST, que disponibiliza dígitos manuscritos e a CIFAR-100, que apresenta imagens coloridas. Outra base mencionada no capítulo foi a CIFAR-100, que possui um número ainda maior de classes de imagens que a CIFAR-10.

Outra base de dados para você verificar é a ImageNet <http://image-net.org/>, que disponibiliza milhares de imagens em diferentes categorias.

Coloque em prática o que você aprendeu, e tente replicar para as outras bases de dados.

REFERÊNCIAS

AREL, I.; ROSE, D. C.; KARNOWSKI, T. P. Deep machine learning – a new frontier in artificial intelligence research. **IEEE Computational intelligence magazine**, v. 5, n. 4, pp. 13-18, 2010.

BENGIO, Y. Learning deep architectures for AI. **Foundations and trends® in machine learning**, v. 2, n.1, p. 1-127, 2009.

BENGIO, Y.; COURVILLE, A.; VINCENT, P. representation learning: a review and new perspectives. **IEEE Transactions on pattern analysis and machine intelligence**, v. 35, n.8, pp. 1.798-1.828, 2013.

BRYNJOLFSSON, E.; MITCHELL, T. M. What can machine learning do? Workforce Implications. **Science**, v. 358, n. 6.370, pp. 1.530-1.534, 2017.

BUDUMA, N. **Fundamentals of deep learning:** designing next-generation machine intelligence algorithms. Sebastopol, CA: O'Reilly Media, 2017.

CHO, K.; MERRIËNBOER, B. van; GULCEHRE, C.; BAHDANAU, D.; BOUGARES, F.; SCHWENK, H.; BENGIO, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *In*: Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar. **Anais**... Doha, Qatar: 2014.

CHOLLET, F. **Deep learning with python**. New York: Manning Publications, 2018.

DATA SCIENCE ACADEMY. **Deep learning book**. [s.l.] Data Science Academy, 2017.

DECHTER, R. Learning while searching in constraint-satisfaction-problems. *In*: Conference Proceedings of the Association for the Advancement of Artificial Intelligence 1986, **Anais**...1986.

DENG, L. An overview of deep-structured learning for information processing. *In*: Proceedings of Asian-Pacific Signal & Information Processing Annual Summit and Conference (APSIPA-ASC), Xi'an. **Anais**... Xi'an: 2011. Disponível em: https://www.microsoft.com/en-us/research/publication/an-overview-of-deep-structured-learning-for-information-processing/. Acesso em: 27/05/2019.

DENG, L. *et al*. Recent advances in deep learning for speech research at Microsoft. *In*: ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing – Proceedings, Vancouver, BC, Canadá. **Anais**... Vancouver, BC, Canadá: 2013.

DENG, L.; YU, D. Deep Learning: Methods and Applications. **Foundations and trends® in signal processing**, v. 7, n. 3-4, pp. 197-387, 2013.

DENTON, E.; CHINTALA, S.; SZLAM, A.; FERGUS, R. Deep generative image models using a laplacian pyramid of adversarial networks. *In*: Neural Information Processing Systems (NIPS), Montreal, Canadá. **Anais**... Montreal, Canadá: 2015. Disponível em: http://arxiv.org/abs/1506.05751.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep learning**. [s.l.] MIT Press, 2016.

GOODFELLOW, I.; POUGET-ABADIE, J.; MIRZA, M.; XU, B.; WARDE-FARLEY, D.; OZAIR, S.; COURVILLE, A.; BENGIO, Y. Generative adversarial nets. In: Advances in Neural Information Processing Systems 27 (NIPS 2014), **Anais**...2014. Disponível em: https://papers.nips.cc/paper/5423-generative-adversarial-nets. Acesso em: 27/5/2019.

HINTON, G. E. Learning Multiple Layers of Representation. **Trends in cognitive sciences**, v. 11, n. 10, pp. 428-434, 2007.

HINTON, G. E. *et al*. Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. **IEEE Signal Processing Magazine**, v. 29, n. 6, pp. 82-97, 2012.

HINTON, G. E.; OSINDERO, S.; TEH, Y. W. A fast learning algorithm for deep belief nets. **Neural computation**, v. 18, n. 7, pp. 1.527–1.554, 2006.

JAIN, S. **An Overview of Regularization Techniques in Deep Learning (with Python code)**. 2018. Disponível em: https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/. Acesso em: 27/05/2019.

JOSHI, P. **Artificial intelligence with python:** build real-world artificial intelligence applications with python to intelligently interact with the world around you. Birmingham, UK: Packt Publishing, 2017.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Convolutional neural networks imagenet classification with deep convolutional neural network. **Communications of the Acm**, v. 60, n. 6, pp. 84-90, 2017.

LECUN, Y. **Generalization and network design strategiestechnical report CRG-TR-89-4**. [s.l: s.n.].

LEE, H.; GROSSE, R.; RANGANATH, R.; NG, A. Y. Unsupervised learning of hierarchical representations with convolutional deep belief networks. **Communications of the ACM**, v. 54, n. 10, pp. 95-103, 2011.

LIMA, I.; PINHEIRO, C. A. M.; SANTOS, F. A. O. **Inteligência artificial**. Rio de Janeiro: Elsevier, 2014.

MCCULLOCH, W. S.; PITTS, W. A Logical Calculus of the Ideas Immanent in Nervous Activity. **The bulletin of mathematical biophysics**, v. 5, n. 4, pp. 115-133, 1943.

MIRZA, M.; OSINDERO, S. **Conditional generative adversarial nets**. [s.l: s.n.]. Disponível em: http://arxiv.org/abs/1411.1784. Acesso em: 27/05/2019.

MITCHELL, T. M. **Machine learning**. [s.l: s.n.]

MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. **Foundations of machine learning**. Cambridge, MA: The MIT Press, 2012.

OHLSSON, S. **Deep learning:** how the mind overrides experience. Cambridge: Cambridge University Press, 2011.

PATTERSON, J.; GIBSON, A. **Deep Learning:** A practitioner's approach. Sebastopol, CA: O'Reilly Media, 2017.

PÉREZ CASTAÑO, A. **Practical artificial intelligence:** machine learning, bots, and agent solutions using C#. New York, NY: Apress, 2018.

RADFORD, A.; METZ, L.; CHINTALA, S. **Unsupervised representation learning with deep convolutional generative adversarial networks**. [s.l: s.n.]. Disponível em: http://arxiv.org/abs/1511.06434. Acesso em: 27/05/2019.

RUSSELL, S.; NORVIG, P. **Artificial intelligence:** a modern approach. 3ª ed. New Jersey: Pearson Education, 2010.

SIMEONE, O. **A brief introduction to machine learning for engineers**. London, England: King's College London, 2018.

SUTSKEVER, I.; VINYALS, O.; LE, Q. V. Sequence to Sequence Learning with Neural Networks. *In*: NIPS'14 Proceedings of the 27th International Conference on Neural Information Processing Systems, Montreal, Canadá. **Anais**... Montreal, Canadá: 2014. Disponível em: http://arxiv.org/abs/1409.3215. Acesso em: 27/05/2019.

THE ASIMOV INSTITUTE. **The neural network zoo**. Disponível em: http://www.asimovinstitute.org/neural-network-zoo/. Acesso em: 27/5/2019.

THRUN, S. *et al*. Stanley: The robot that won the DARPA grand challenge. **Journal of Field Robotics**, v. 23, n. 9, pp. 661-692, 2006.

TURING, A. M. Computing Machinery and Intelligence. **MIND**, v. 59, n. 236, pp. 433-460, 1950.

VASILEV, I.; SLATER, D.; SPACAGNA, G.; ROELANTS, P.; ZOCCA, V. **Python deep learning:** exploring deep learning techniques and neural network architectures with pytorch, keras and tensorflow. 2ª ed. [s.l.] Packt Publishing, 2019.

VINYALS, O.; TOSHEV, A.; BENGIO, S.; ERHAN, D. **Show and tell:** a neural image caption generator. *In*: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA. **Anais**... Boston, MA: 2015.

WERBOS, P. J. **Backpropagation Through time:** what it does and how to do it. *In*: Proceedings of IEEE, **Anais**...1990.

YU, D.; DENG, L. Deep Learning and its applications to signal and information processing. **IEEE Signal Processing Magazine**, v. 28, n. 1, pp. 145-150, 2011.