Capítulo 4

Introdução às Redes Neurais Profundas com Python

Josenildo C. da Silva, Raimundo Osvaldo Vieira

Abstract

Deep network are one of the most important subfield of machine learning. Everyday we learn about new applications or new architectures attracting a lot of interest from industry and academy as well. This lecture is a gentle introduction on deep neural networks for developers who want to give the first steps in this field. This text contains the fundamental concepts, main architectures and libraries. It is complemented by code examples covering each concept discussed.

Resumo

As redes neurais profundas representam uma das mais importantes subáreas da aprendizagem de máquina. Todos os dias surgem novas aplicações ou novas arquiteturas tornando esta tecnologia um tópico de muito interesse tanto para o mercado como para academia. Este minicurso é uma introdução às redes neurais profundas para desenvolvedores que desejam dar os primeiros passos nesta área. O texto apresenta os conceitos básicos, principais arquiteturas e bibliotecas e é complementado com exemplos de código para demonstrar na prática cada tópico.

4.1. Introdução

O desenvolvimento de algoritmos tradicionais é ponto central de estudo da ciência da computação. O programador tem a tarefa de compreender um dado problema e criar um programa que produz a saída requerida, de acordo com as restrições do problema. Isto funciona sempre que o problema é matematicamente bem definido, tais como programas para área comercial, aplicações bancárias, jogos, editores de texto, sistemas operacionais, etc. Entretanto, há algumas situações onde as regras não são claras ou o objetivo é descobri-las. Neste caso, a abordagem tradicional não se aplica e o desenvolvimento de um algoritmo torna-se inviável. Nestes casos, utilizamos a aprendizagem de máquina.

Uma das aplicações mais tradicionais de aprendizagem de máquina é a análise de crédito para solicitação de empréstimos ou cartão de crédito. Alguns exemplos mais recentes são aplicativos ou websites recomendadores automáticos (de livros, filmes, músicas, etc), detectores de *spam* nos aplicativos de email, reconhecedores de voz nos telefones celulares, identificadores de rosto no aplicativo de fotos Picasa, ou no Facebook. O maior exemplo, no entanto, talvez seja o mecanismo de busca da Google. Na realidade, todos os produtos da Google observam atentamente nossos movimentos e utilizam aprendizagem para fazer as propagandas do Google AdWords, ou manter o ranking das páginas. Atualmente, somos usuários de aprendizagem de máquina diariamente, mesmo sem nos darmos conta disto.

Uma importante subárea da aprendizagem de máquina é chamada de redes neurais profundas. A partir dos primeiros trabalhos sobre neurônios artificiais, na década de 1940, esta linha de pesquisa tem produzido muitos avanços nas ultimas décadas. Mas foi somente na década de 2000 que as redes neurais profundas começaram a chamar a atenção. Dois fatores contribuíram para o sucesso destas redes: o surgimento das unidades de processamento gráfico (GPUs) e o aumento da quantidade de dados disponíveis. Os primeiros sucessos das redes neurais profundas foram na área de processamento de imagem. Problemas tais como detecção de objetos, segmentação de imagem e até mesmo geração de legenda para imagens são exemplos de aplicação mais conhecidos. As redes profundas também aceitam outros tipos de dados, tais como áudio, vídeo ou texto. Os resultados mais recentes são de redes geradoras de imagens a partir de texto, DALL-E[1] e Imagen[2].

Este é um minicurso introdutório, voltado para desenvolvedores que desejam dar os primeiros passos na área de deep learning. Assumimos conhecimento prévio de álgebra básica e algoritmos clássicos de aprendizagem de máquina. Também é desejável que você tenha conhecimento básico da linguagem Python, o que pode facilitar o entendimento dos exemplos apresentados. Nosso objetivo é dar uma visão geral sobre redes neurais profundas, apresentando os conceitos fundamentais bem como exemplos em linguagem Python. Ao final deste minicurso, o aluno deve ter uma clara ideia sobre os componentes básicos de redes neurais profundas, seu potencial e indicações de como iniciar uma trilha de estudos nessa área.

Este texto faz parte do material do minicurso, juntamente com os códigos em Python que serão disponibilizados aos participantes. O objetivo é apresentar os conceitos básicos no texto e apresentar os aspectos avançados através de discussão com código. Nas seções seguintes, discutiremos os fundamentos da aprendizagem de máquina, redes neurais clássicas, redes neurais profundas e redes neurais recorrentes.

4.2. Fundamentos de Aprendizagem de Máquina

A visão mais atual de aprendizagem de máquina é a de construção de modelos. A tarefa de criar um software capaz de fazer predições é equivalente a criar um modelo de uma função desconhecida a partir dos dados disponíveis. A primeira etapa desta tarefa é definir uma família de modelos, como, por exemplo, funções lineares, árvores de decisão, ou mesmo redes neurais. O trabalho do algoritmo de aprendizagem pode ser visto como uma

[1] `https://openai.com/blog/dall-e/`
[2] `https://imagen.research.google/`

busca por uma instância de modelo específico que melhor represente o conjunto de dados [Hastie et al. 2009].

De modo geral, pode-se definir este problema como uma busca em espaço de estados, onde cada estado é um modelo possível. Para guiar o processo de navegação no espaço de estados, normalmente os algoritmos utilizam alguma **função de custo** além de **função de avaliação** do modelo, para decidir se o modelo atual é bom o suficiente para representar a função desconhecida ou se é necessário continuar a busca. Como se trata de uma busca local, ou seja, não importa o caminho percorrido até o modelo escolhido, tem-se um problema de otimização. Se a função de avaliação é convexa, pode ser possível encontrar uma **formula fechada** para produzir o modelo. Se a função de avaliação não for bem comportada, com vários máximos locais, é necessário utilizar alguma técnica de **otimização** avançada.

Para exemplificar, considere a regressão linear simples. O modelo produzido é uma combinação linear de polinômios. Cada estado é uma combinação em particular definida pelos valores dos coeficientes. A função de avaliação mais utilizada é a soma dos quadrados das diferenças entre previsão e valor correto de y (SSE, sum of squared errors). O algoritmo *least square regression* (LSR) utiliza uma fórmula fechada para calcular os coeficientes do modelo a partir dos dados, sem ter que navegar pelo espaço de estados.

Um outro exemplo é o algoritmo ID3. O modelo produzido pelo ID3 é simbólico, consiste em uma árvore formada pelo nome dos atributos e valores de decisão a cada nível. Nas folhas há o valor de cada classe. A construção é iterativa e a navegação é guiada por uma função heurística que define qual atributo escolher para a raiz da próxima subárvore.

4.2.1. Tipos de Aprendizagem

Um pressuposto importante na aprendizagem de máquina é que existe uma relação (desconhecida) entre os conjuntos de dados que estamos interessados. O objetivo da aprendizagem é construir um modelo que se comporte de modo similar à relação desconhecida.

Há três configurações básicas para aprendizagem automática, de acordo com o formato das informações disponíveis: aprendizagem supervisionada, aprendizagem não-supervisionada e aprendizagem por reforço [Géron 2017].

Na **aprendizagem supervisionada** há um conjunto X representando observações e um conjunto Y representando valores associados a cada elemento de X. Assim, pode-se dizer que, na aprendizagem supervisionada, para cada $x \in X$ sabe-se qual é o $y \in Y$ correto. Em outras palavras, existe uma coleção de pares $\{(x,y) \mid x \in X, y \in Y\}$, ou seja, uma amostra da relação verdadeira de modo enumerado. O objetivo do algoritmo de aprendizagem é encontrar uma aproximação mais concisa para esta relação.

Na **aprendizagem não-supervisionada** há um conjunto X representando as observações, mas não há um conjunto Y. Deste modo, não há valores corretos a priori e, portanto, não há relações para se descobrir. Entretanto, há várias propriedades dos dados que são de interesse. Por exemplo, pode-se investigar as estatísticas básicas das observações (média, desvio padrão, etc.), saber quantas modas existem nos dados, se há regiões mais densas que outras, etc.

Na **aprendizado por reforço** o objetivo é encontrar a melhor ação a, a ser escolhida

em um dada situação x, com a recompensa y. Além disso, não há conjuntos X, Y ou A à disposição, mas temos acesso a cada par (a,x,y) um por vez. O algoritmo constrói um modelo de modo incremental, à medida que chegam novas observações, que representa uma política de escolhas de ações de modo a maximizar a recompensa.

4.2.2. Utilização do Conhecimento

Uma vez que se tenha um modelo construído, há pelo menos duas coisas que se pode fazer: *prever o futuro* ou *explicar o passado*. De modo mais técnico, as principais tarefas de aprendizagem, de acordo com o uso do modelo, são: predição e descrição.

Predição é a tarefa de calcular valores Y para tuplas futuras. Por exemplo, se temos um modelo $f(\cdot)$ que foi treinado com os pares $\{(2,4),(9,81),(3,9),(10,100),(4,16)\}$, podemos utilizá-lo para predizer qual o valor y quando $x = 7$, ou seja, $y = f(7)$, ou ainda $(7,y)$. Duas formas de predição mais conhecidas são a regressão e a classificação. Fala-se de *regressão* quando o conjunto Y é continuo. Por outro lado, quando o conjunto Y é discreto, fala-se de *classificação*. A predição, pressupõe que haja um conjunto Y com os valores corretos ou categorias. Portanto, a predição é aprendizagem supervisionada.

Descrição é a tarefa de construir modelos que sintetizam as principais características do conjunto de observações. O objetivos é tentar descobrir se há alguma estrutura subjacente no conjunto de observações ou se é possível descrever o processo de geração dos dados. Duas formas de descrição mais conhecidas são a estimação de densidade e a análise de grupos. Como não há indicação de grupos a priori, trata-se de uma forma de aprendizagem não-supervisionada.

4.3. Redes Neurais Artificiais Clássicas

Redes Neurais Artificiais representam uma técnica supervisionada de predição que gera um modelo não linear representando um grafo de unidades neuronais e os pesos de suas conexões [Coppin and Valério 2015]. Os algoritmos mais conhecidos são Perceptron, Adaline e Multilayer Perceptron (MLPs). As redes neurais profundas são o desenvolvimento mais recente desta família de modelos.

4.3.1. Neurônio biológico, Perceptrons e MLPs

As redes neurais naturais são formadas por um intricado grafo de conexões entre os neurônios biológicos. Um neurônio recebe impulsos elétricos de outros neurônios e se a soma de todos os impulsos de entrada ultrapassam um determinado limiar, ele transfere o impulso para outros neurônios a ele conectado (veja Figura 4.1).

Um **perceptron** é uma descrição matemática de um neurônio biológico. Os perceptrons realizam uma soma ponderada dos valores de entrada, seguido de uma função de ativação, que se comporta de modo similar ao neurônio natural (Figura 4.2).

A saída será ativada se a soma atingir um valor de limiar.

$$y = \begin{cases} 1, \text{se } \mathbf{W}\mathbf{x} + b > 0 \\ 0, \text{caso contrário.} \end{cases} \tag{1}$$

Uma rede neural artificial clássica é uma estrutura de camadas onde vários percep-

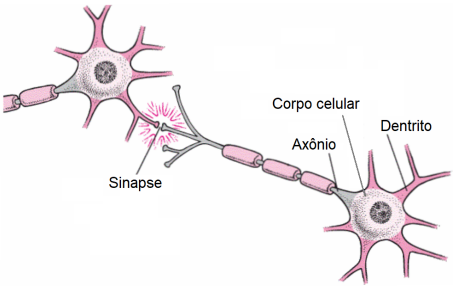

Figura 4.1. Neurônio biológico.

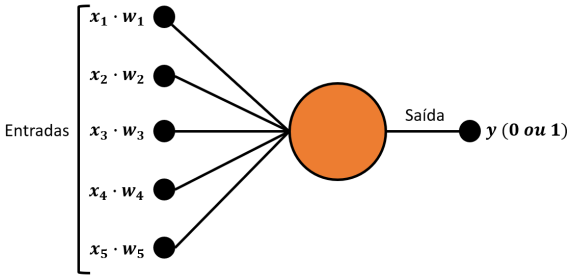

Figura 4.2. Perceptron.

trons são interconectados. Uma rede com esta arquitetura é denominada **perceptron de múltiplas camadas**, ou MLPs (da sigla em inglês). Nesta formato, as camadas internas da rede, que não sejam entrada ou saída, são chamadas de camadas escondidas (veja Figura 4.3).

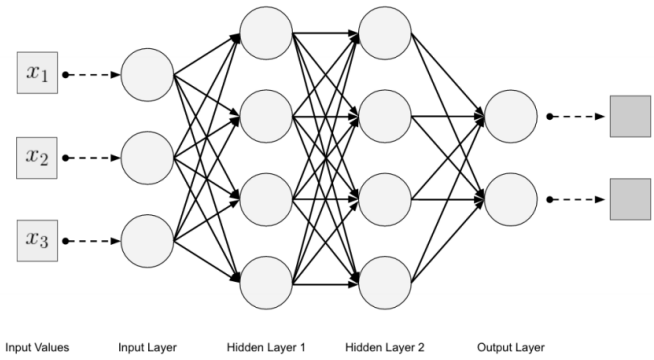

Figura 4.3. Exemplo de uma MLP com duas camadas escondidas.

As MLPs são chamadas de redes de alimentação adiante, ou *feedforward networks*, porque a informação é processada desde a camada de entrada até a camada de saída sem que haja retroalimentação entre as camadas ou unidades [Goodfellow et al. 2016].

4.4. Python, Tensorflow, Keras

4.4.1. Python

Python é uma das linguagens de programação mais populares atualmente. Aparece em terceiro lugar no TIOBE Index em setembro de 2019[3] tendo crescido muito o interesse por ela nos três últimos anos. Python tem se tornado uma das linguagens mais utilizadas para análise de dados junto com R e Júlia.

Python é uma linguagem de sintaxe simples, código aberto, que conta com uma grande comunidade de colaboradores. Isto torna a linguagem particularmente acessível e adequada para atacar vários tipos de problemas em domínios diferentes. As bibliotecas (packages) mais relevantes para a área de aprendizagem de máquina são: numpy, pandas, matplotlib, jupyter, entre outros.

A linguagem Python foi criada por Guido van Rossum em 1991 com o objetivo de ser clara, concisa e explícita. Para programadores com alguma experiência em outras linguagens, alguns aspectos do Python são muito estranhos, como a utilização de espaços em branco.

A tipagem em Python é dinâmica, o que significa que não precisamos declarar tipos ao criar uma variável. Entretanto, todo valor atribuído a uma variável possui um tipo associado. Recentemente, foi acrescentado a possibilidade de indicar o tipo de variáveis, parâmetros e retornos de funções, chamado de *type hints*, ou dicas de tipo (em tradução livre).

Python permite trabalhar com vários paradigmas. Oferece a possibilidade de se trabalhar orientado à objetos, mas também aceita que um programa seja baseado em funções. Além disso, por não ser compilado, permite prototipagem rápida ideal para área de análise de dados e aprendizagem de máquina.

4.4.2. Tensorflow

Tensorflow (TF) é uma plataforma de desenvolvimento de modelos de aprendizagem de máquina em larga escala [Abadi et al. 2015]. TF é desenvolvido em python pela Google para permitir operações com tensores de modo eficiente. De certo modo, o TF é muito similar ao Numpy. Entretanto, o TF consegue trabalhar com CPU e GPU, permite a distribuição de operações entre várias máquinas, além de poder ser exportado para outras plataformas tais como C++, Javascript ou Tensoflow Lite.

Tensorflow representa internamente a execução de uma função através de um grafo dirigido. Cada nodo representa uma operação, por exemplo multiplicação, ou soma. A informação flui através do grafo de computação na forma de vetores de dimensão arbitrária chamados de tensores.

O TF realiza diferenciação automática de funções utilizando um grafo específico para isso (veja Figura 4.5). Portanto, se um determinado nó do grafo representa uma operação $f(x)$, o TF calcula também sua derivada df/dx no gráfico de derivadas. Essa abordagem facilita a implementação de algoritmos tais como *backpropagation*

[3] `https://www.tiobe.com/tiobe-index/`

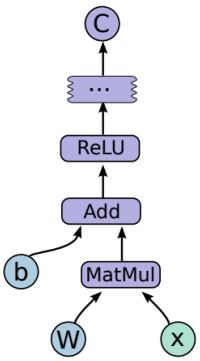

Figura 4.4. Grafo computacional do TensorFlow. Cada nó representa uma operação [Abadi et al. 2015].

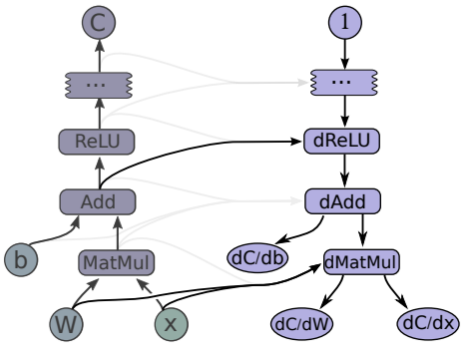

Figura 4.5. Grafo para derivação automática [Abadi et al. 2015].

A implementação do TF inclui vários pacotes com funções específicas tais como processamento de E/S, otimização, deployment, visualização com tensorboard, além de uma extensa lista de operações matemáticas (veja Figura 4.6). TF possui ainda pacotes para processamento de sinais, tratamento de imagens, e datasets de exemplos prontos para uso.

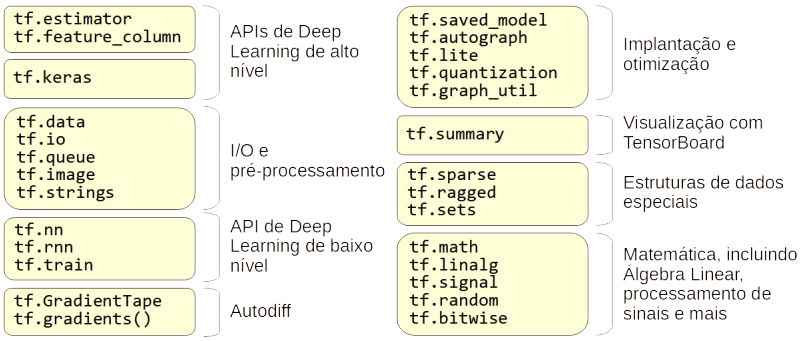

Figura 4.6. API do Tensorflow em Python [Géron 2017].

4.4.3. Keras

Keras é uma API de alto nível desenvolvido para facilitar o uso de tensorflow (veja figura 4.7). Keras foi lançado em 2015, e inicialmente foi desenvolvido sobre o Theano, outra biblioteca para manipulação de tensores que também tem diferenciação automática de funções. Em 2018 o Keras foi escolhido pelo Tensoflow como sua API de alto nível oficial.

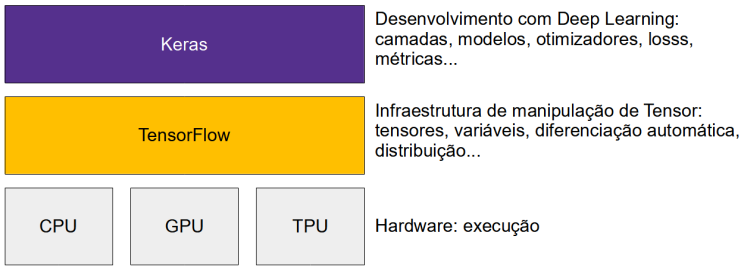

Figura 4.7. Keras é uma API desenvolvida a partir do Tensorflow [Chollet 2021].

Os principais componentes e etapas do Keras são: camadas, modelos, compilação e treinamento.

Camadas (Layers). Camadas são unidades de processamento no Keras que recebem um tensor de entrada e produzem outro tensor de saída. Os pesos de uma camada são aprendidos durante o treinamento da rede.

Modelos (Models). Uma rede neural em Keras é um grafo de camadas represendado pela classe Model. O modelo mais simples é chamado de Sequential, mas o Keras permite a definição de modelos com topologia complexa, não-sequencial.

Compilação. O método `compile()` configura o processo de treinamento. Geralmente são informados o otimizador, a função de custo e as métricas a serem utilizados no treinamento.

Treinamento. O método `fit()` executa o treinamento. Seus principais parâmetros são os dados de treinamento, o número de épocas e o tamanho do subset de treino (batch).

4.4.4. Exemplo: rede MLP simples

No código a seguir, mostramos como implementar uma MLP em tensorflow. Para isso vamos importar o pacote `tensorflow`.

```
import tensorflow as tf
```

Usaremos o dataset de dígitos manuscritos MNIST, que é distribuído no tensorflow. Há uma versão mais recente deste dataset, chamado fashion MNIST, que possui imagens de itens de moda. Ele pode ser obtido com `fashion_mnist` no lugar de `mnist`.

```
mnist = tf.keras.datasets.mnist
```

Separamos os dados de treino e de teste e também normalizamos os valores para que estejam no intervalo de 0 a 1.

```
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

O modelo tem como entrada um tensor de 784 neurônios definido pela camada Flatten, que recebe um tensor de rank-1. O valor 784 é o número de pixeis em cada imagem de dimensões 28x28. Nesta MLP há uma camada escondida de 128 neurônios e uma camada de saída com 10 neurônios.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

O modelo definido é compilado com um otimizador `adam` e uma função de custo `entropia cruzada categórica`, que é a função de custo mais utilizada para treinar um classificador. Além disso, o modelo será avaliado com a métrica `acurácia`.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Finalmente, iniciamos o treinamento chamando o método `fit()` do modelo compilado. Neste exemplo são definidos apenas 5 épocas de treinamento. Na prática, pode ser necessário um número muito maior de épocas para treinar um modelo.

```
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

4.5. Redes Neurais Convolucionais

Uma rede neural profunda [Goodfellow et al. 2016] é uma MLP, ou seja, uma rede de alimentação para frente (feedforward network) que possui várias camadas escondidas, tais como discutidas na seção 4.3. Estas redes são inspiradas no cortex visual do cérebro, onde os sinais exteriores são processados por várias camadas. Atualmente, o número de camadas pode chegar a centenas, dependendo da arquitetura. A AlexNet [Krizhevsky et al. 2012] por exemplo, possui 8 camadas. A VGGNet possui versões com 16 a 19 camadas [Simonyan and Zisserman 2014]. Já a ResNet pode chegar a 152 camadas [He et al. 2016]. As redes neurais profundas utilizam muitas operações matriciais e portanto só ganharam atenção com o desenvolvimento de GPUs (*graphic processor units*), que permitiram a execução paralela de um grande número de operações.

Talvez o ponto mais importantes das redes neurais profundas seja não apenas o número de camadas, mas a introdução de vários operadores que não eram utilizados em MLPs. Nas seções seguintes, vamos discutir os principais destes operadores: convolução, pooling e flattening. Devido à utilização de camadas de convolução, estas arquiteturas são geralmente chamadas de **redes neurais convolucionais**, ou **convolutional neural networks** (CNN) no original em inglês.

4.5.1. Principais Componentes das CNNs

Convolução Convolução é uma operação que aplica repetidamente um filtro (kernel) sobre os dados de entrada para gerar um mapa de atributos (features). O kernel geralmente tem dimensões pequenas e representa padrões locais a serem detectados. Matematicamente, a operação consiste em uma multiplicação do kernel com uma fatia dos dados de entrada. A operação utilizada em redes profundas é uma correlação cruzada (cross-correlation), mas o nome convolução continua sendo amplamente utilizado.

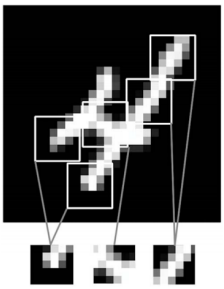

Figura 4.8. Kernels representam padrões locais aprendidos [Chollet 2021].

Ao contrário de um filtro utilizado em outros domínios, por exemplo tratamento de imagens, os valores do kernel em redes profundas são aprendidos durante o treinamento. O kernel representa um padrão local um atributo aprendido e é similar aos filtros de borda em imagens, linhas, ou outros padrões mais complexos (cf. Figura 4.8). Quando a característica representada pelo kernel está presente em uma região dos dados de entrada, o resultado da operação indica um valor maior que das regiões onde nada foi detectado. A utilização de operação tem como saída um mapa de resposta ou mapa de características (*feature map*), que geralmente é menor que os dados originais e que serve de entrada para as camadas seguintes da rede.

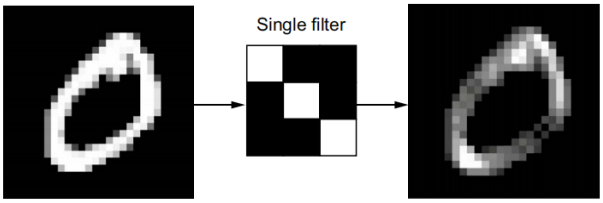

Figura 4.9. Mapa de resposta (*feature map*) resultante da aplicação de um filtro sobre uma imagem [Chollet 2021].

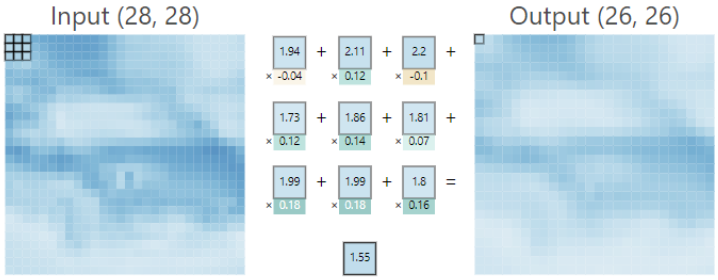

Figura 4.10. Convolução: multiplica-se cada elemento do filtro por um elemento equivalente na fatia da imagem e depois todos os termos são somados [Wang et al. 2020].

Há duas característica importantes das redes baseadas em convolução. Primeiro, os padrões aprendidos por convolução são independentes de localização. Desse modo, um padrão aprendido pode ser detectado em qualquer posição da imagem. Portanto, diz-se que os padrões aprendidos são *invariante de translação*. Segundo, os padrões aprendidos nas primeiras camadas são utilizados como elementos para compor padrões mais complexos em camadas subsequentes. Isto forma uma hierarquia de padrões[4].

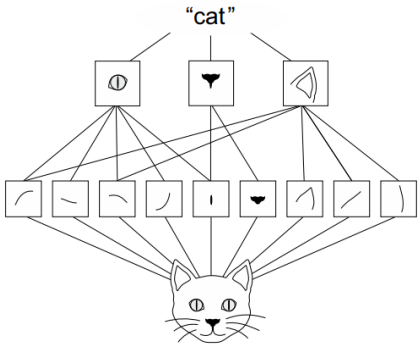

Figura 4.11. Hierarquia de padrões aprendidos em uma rede neural profunda [Chollet 2021].

A operação de convolução possui três hiperparâmetros: padding, kernel size e stride. Vamos discutir cada um deles.

[4]Um dos primeiros trabalhos a incluir todas estas características foi a rede Neurocognitron em 1980. O trabalho do Prof. Fukushima é reconhecido como o precursor das Redes Neurais Convolucionais.

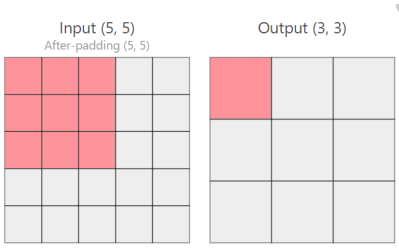

Figura 4.12. Operação de covolução com um kernel de 3x3 em uma imagem de 5x5, com padding = 0 e stride = 1.

kenel size: Define o tanho do filtro que será utilizado. Tamanhos típicos são 2x2, 3x3, ou 5x5.

padding: Ao aplicar o kernel, pode-se preencher a entrada com zeros para que o filtro seja aplicado em mais áreas limites. Um valor típico de padding é 0. Em Keras, usa-se o valor ”valid”para indicar padding = 0, and ”same”para usar um valor de padding automático.

stride: O filtro é aplicado em forma de janela deslizante. O stride define o passo em que o filtro será aplicado. Valor típico é 1.

Pooling. A operação de pooling aplica uma função matemática em uma subárea do input e substitui todos os valores pelo resultado da função. Geralmente usa-se a função max, mas também é possível utilizar a função média. Um tamanho típico de pooling é 2x2. Neste caso, uma região de 2x2 será substituída pelo valor máximo encontrado na região 4.13.

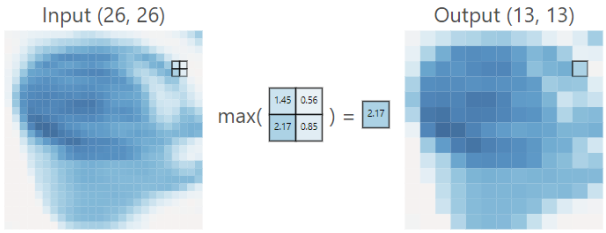

Figura 4.13. Max pooling de 2x2 aplicado a um input de 26x26. A Saída será 13x13.

Um dos efeitos do pooling é a redução das dimensões da entrada. Quanto maior o tamanho da camada de pooling, maior será a redução.

Flattening. Quando as operações de convolução e de pooling são realizadas em um tensor de rank-2 ou maior, ou seja, em uma entrada de duas ou mais dimensões, é necessário mudar o formato da camada antes de passar para etapa de classificação, geralmente um MLP. Por isso, a operação de flattening transforma os dados para um tensor de rank-1 (veja Figura 4.14).

Normalmente, a operação de flattening é aplicada sobre o mapa de características produzidas pelas camadas convolutiva anteriores. O tensor rank-1 resultante é utilizado

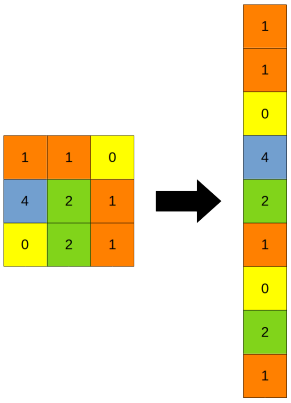

Figura 4.14. Operação de flattening.

como entrada em uma camada totalmente conectada, geralmente chamada de camada densa na literatura de redes profundas.

Funções de ativação. As funções de ativação mais conhecidas ao se construir uma MLP são a sigmoide e a tangente hiperbólica. Entretanto, a maioria das funções de

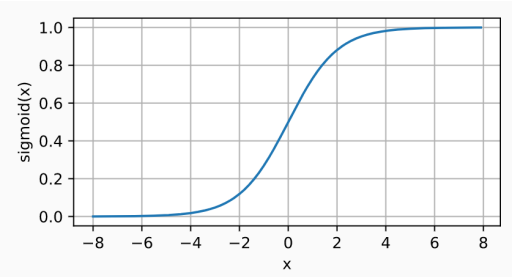

Figura 4.15. Função sigmoide [Zhang et al. 2021].

ativação podem fazer com que o gradiente fique próximo de zero em redes profundas, devido à grande quantidade de camadas onde o gradiente tem que ser multiplicado durante o processo de *backpropagation*. Devido à isso, várias outras funções de ativação são utilizadas em redes neurais profundas, dentre elas a *rectified linear unit*, ou ReLU.

$$ReLU(x) = \max(0, x) \tag{2}$$

A função ReLU é utilizada após a operação de convolução. O resultado é que apenas os valores positivos são preservados (veja Figura 4.16. Além disso os valores positivos não são restridos, como na sigmoide (veja Figura 4.15). Logo, não há saturação quando x é positivo. A ReLU pode ser utilizada em qualquer camada escondida logo após uma camada de convolução.

Existem várias outras alternativas à função ReLU, tais como Leaky ReLU, GeLU, e assim por diante. Todas buscam corrigir algum problema da ReLU. Ainda assim, ela continua sendo largamente utilizada na maioria das arquiteturas de redes profundas.

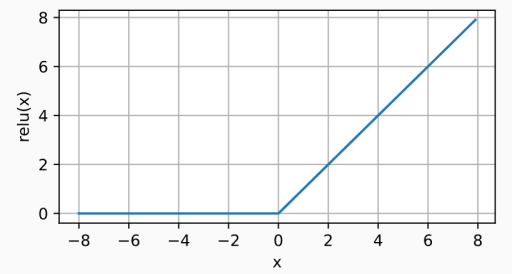

Figura 4.16. Função ReLU [Zhang et al. 2021].

Os valores de saída de uma ReLU não são restritos a nenhum valor superior. Para saber qual a classe indicada por uma rede, seria suficiente tomar o valor máximo na saída da rede. Contudo, geralmente é útil expressar a saída da rede em forma de probabilidade, onde todos os valores são normalizados para somarem 1. A função softmax faz isso. Além disso, ela é derivável, portanto pode ser utilizada diretamente no algoritmo de backpropagation.

A função softmax é definida como:

$$Softmax(\mathbf{z}) = \frac{\exp(z_i)}{\sum_{j=0}^{K} \exp(z_j)} \tag{3}$$

Por exemplo, considere que uma determinada classe tem o valor final, antes da softmax, de 8.65. A softmax vai somar todos os valores para outras classes e somar, e usar esse total para normalizar o valor 8.65. Veja a imagem na Figura 4.17.

A função softmax é utilizada como ativação na ultima camada de uma rede neural.

$$\frac{\exp(8.65)}{\begin{array}{l}(\exp(-4.26) + \exp(2.97) + \exp(-0.38) + \exp(5.24) + \\ \exp(-7.58) + \exp(-3.43) + \exp(8.65) + \exp(2.63) + \\ \exp(6.31) + \exp(0.69))\end{array}} = 0.8808$$

Figura 4.17. Exemplo numérico da aplicação de softmax.

4.5.2. Exemplo: arquitetura LeNet

Uma das primeiras redes que utilizam operação de convolução foi proposta por Yann LeCun [Lecun et al. 1998]. A LeNet foi aplicada para reconhecimento de dígidos manuscritos e ajudou a despertar o interesse para as arquiteturas profundas.

LeNet apresenta sete camadas, todas elas com parâmetros modificáveis durante o treino. As quatro primeira camadas apresentam blocos de Convolução seguida de Pooling. Em seguida é aplicada uma operação de Flattening que produz um vetor de entrada para três camadas Densas finais. A última camada tem como saída uma função de ativação softmax que produz a probabilidade da imagem de entrada ser um dos dígitos de 0 a 9.

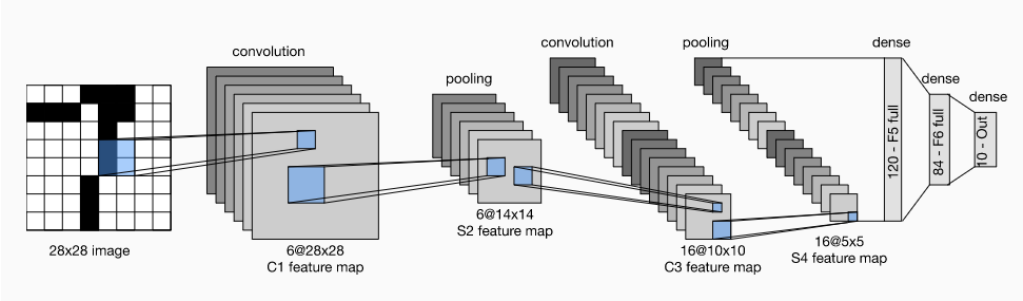

Figura 4.18. Arquitetura da rede LeNet (reproduzido de [Zhang et al. 2021]).

4.5.3. Exemplo: classificador de dígitos manuscritos

O exemplo a seguir foi extraído da documentação oficial do Keras. A arquitetura produzida é muito similar a uma rede LeNet. O objetivo é apresentar um exemplo mínimo mas funcional em Keras.

Os pacotes necessários para este exemplo são numpy, keras e layers.

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
```

O dataset MNIST já vem disponível no Keras. As imagens são em escala de cinza, apenas um canal, com dimensões 28x28. No Keras o dataset já está dividido em treino e teste. Na prática, quando você utilizar um outro dataset, esta divisão terá que ser realizada explicitamente pelo programador.

```
# Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)
# Load the data and split it between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

Os valores das imagens precisam ser normalizadas para valores entre 0 e 1. Originalmente estão em escala de cinza com valores de 0 a 255.

```
# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")
```

As classes são convertidas para uma codificação vetorial que facilita a saída de uma rede com 10 neurônios. Essa representação algumas vezes é chamada de *one-hot encoding*. Como exemplo, a classe 0 pode ser representada como [1,0,0,0,0,0,0,0,0,0]. A classe 1 pode ser representada como [0,1,0,0,0,0,0,0,0,0]. E assim por diante. Cada posição do vetor representa a classe da imagem.

```
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

Esta rede possui uma entrada de 28x28. Em seguida, uma camada de convolução com 32 kernels de 3x3 é aplicada sobre a entrada, gerando os mapas de características. Uma função ReLU é utilizada para retificar os mapas de características. Em seguida uma camada de pooling de 2x2 e aplicada tendo como efeito a redução das camadas de features para metade do seu tamanho. A mesma sequencia é repetida, desta vez com 64 kernels 3x3, ReLU e pooling.

Finalmente, os mapas de caracteristicas são tranformados em um tensor rank-1 e passados como entrada para uma camada MLP (densa) com saída de 10 neurônios. A função de ativação na saída é uma softmax.

```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)
```

O Keras produz um resumo da arquitetura construída com o método `summary()` do modelo.

```
model.summary()
```

Este exemplo utiliza subconjuntos de 128 imagens para cada etapa do treinamento (batchs). O treinamento total será feito em 15 épocas.

```
batch_size = 128
epochs = 15
```

O modelo vai ser treinado com a função de custo ”categorical crossentropy”, muito utilizada para problemas de classificação. O otimizador utilizado é o Adam e a qualidade do modelo final será determinada utilizando acurácia.

```
model.compile(loss="categorical_crossentropy", optimizer="adam",
                                    metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
                                    validation_split=0.1)
```

O método `evaluate()` permite examinar os valores finais da função de custo e da função de qualidade, neste caso a acurácia.

```
score = model.evaluate(x_test, y_test, verbose=0)
```

```
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

4.5.4. Aplicações de redes convolucionais

As redes convolucionais tem sido aplicadas em vários domínios diferentes e com uma vasta gama de tipos de dados. Para reconhecimento de objetos em imagens, detecção de objetos, segmentação de imagens. As CNNs tem sido ainda utilizadas para detecção de padrões em aúdio, reconhecimento de voz, tratamento de linguagem natural, análise de vídeo, detecção de padrões de sono em sinais de EEG.

4.6. Redes Recorrentes

Sequências são um tipo de dados onde a ordem é importante, o tamanho pode variar e os elementos podem se repetir. Por exemplo, letras em uma palavra pode ser visto como uma sequência. Palavras em um texto também são sequências. Notas musicais, frames de vídeos, valores de demandas de produtos, preços de ações em bolsa de valores, registro de consumo de energia.

As redes neurais convolucionais (CNNs) vistas na Seção 4.5 não conseguem representar dados sequenciais. Uma CNN recebe um tensor X de entrada e calcula uma saída y. Entretanto, a arquitetura CNN não consegue manter a ordem dos dados, ou lidar com sequências de tamanho variado ou muito grandes.

Mas, então, como modelar dados em sequência? A seguir, vamos discutir algumas abordagens, limitações e introduzir as redes neurais recorrentes e suas variações.

4.6.1. Redes Neurais Recorrentes (RNNs)

Uma primeira forma de modelar uma sequência $\mathbf{X} = (x_1, x_2, \ldots, x_t)$ poderia ser calcular a **probabilidade da sequência completa**, a partir da probabilidade de cada elemento. Assim, teremos:

$$p(\mathbf{X}) = \prod_{t=1}^{T} p(x_t) \tag{4}$$

Mas esse modelo assume independência entre o elementos da sequência, o que não é verdadeiro para sequências.

Um melhor modelo seria **condicionar a probabilidade** do próximo elemento aos elemento passados, ou seja, $p(x_t|x_{t-1}, x_{t-2}, \ldots, x_0)$. Entretanto, essa abordagem não é escalável. Mesmo considerando apenas um elemento anterior, para modelar a palavras em uma língua, teríamos que calcular um número gigante de probabilidades $p(x_t|x_{t-1})$, da ordem de $|\Sigma|^2$, onde Σ é o vocabulário da língua em questão.

Vetorizar o contexto é uma técnica que recebe $x_{t-1}, x_{t-2}, \ldots, x_0$ e devolve um valor h que representa um sumário dessa sequência de contexto. Desse modo:

$$p(x_t|x_{t-1}, x_{t-2}, \ldots, x_0) \approx p(x_t|h) \tag{5}$$

A **redes neuronais recorrentes** (*recurrents neural networks*, RNNs) codificam o contexto h através da multiplicação de uma matriz $\mathbf{W}$ de pesos aprendidos com o contexto anterior. Há outra matriz de pesos aprendidos $\mathbf{U}$ que é aplicada a entrada atual no cálculo do novo contexto h [Goodfellow et al. 2016]:

$$h_t = \tanh\left(\mathbf{W}^\top \mathbf{h}_{t-1} + \mathbf{U}^\top x_t\right) \tag{6}$$

Na literatura de RNNs, h é geralmente chamado de *hidden state*.

Assim, pode-se calcular $p(x_{t+1}|h_t)$ como:

$$p(x_{t+1}|h_t) = softmax(\mathbf{V}^\top \mathbf{h}_t) \tag{7}$$

4.6.2. LSTMs e GRUs

As RNNs sofrem de alguns problemas graves. Primeiro, não são capazes de trabalhar com sequências de tamanho variado. Além disso, devido ao número de matrizes intermediárias no cálculo de h, o gradiente pode tender a zero, tornando o aprendizado ineficiente. Este problema é chamado de vanishing gradients. As RNNs também não conseguem manter um contexto muito longo, pois o contexto é transformado por várias multiplicações de matrizes ao longo da construção do contexto.

As **long short term memory** (LSTMs) resolvem estes problemas através da introdução dos chamados portões (*gates*) e também do contexto de longo prazo c. O *forget gate* controla quais informações serão removidas do contexto de longo prazo. O *input gate* controla o que será adicionado ao contexto de longo prazo. O *output gate* utiliza a entrada x_t, o contexto de curto prazo h_{t-1} e o contexto de longo prazo c_{t-1} para produzir a saída h_t.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t]) \tag{8}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t]) \tag{9}$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t]) \tag{10}$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_{t-1} \tag{11}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t]) \tag{12}$$
$$h_t = o_t * \tanh(C_t) \tag{13}$$

As **gated recurrent unit** (GRUs) são uma versão simplificada das LSTMs, que trabalha com somente dois *gates*. Ela combina o forget e o input gate em um só e também combina contextos de longo e curto prazo.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \tag{14}$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \tag{15}$$
$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t]) \tag{16}$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_{t-1} \tag{17}$$

4.6.3. Aplicações de Redes Recorrentes

As redes recorrentes são utilizadas em vários cenários distintos. Os principais tipos de tarefas quando se modela sequências são:

muitos-para-um: o contexto h representa vários elementos da sequência e a predição será para apenas um elemento x_{t+1}. Exemplo: texto de um comentário é o contexto e o sentimento é a saída calculada.

um-para-muitos: o contexto h representa um elemento da sequência e a predição será para vários elementos $x_{t+1}, x_{t+2}, \ldots$. Exemplo: uma imagem é a entrada e um texto de legenda é a saída gerada.

muitos-para-muitos: o contexto h representa vários elementos da sequência e a predição será para vários elementos $x_{t+1}, x_{t+2}, \ldots$. Exemplo: tradução automática de uma língua para outra.

4.7. Conclusão e Próximos Passos

Este minicurso apresentou uma breve introdução às redes neurais profundas. Nosso foco foi as arquiteturas voltadas para aprendizagem supervisionada, uma vez que isso já representa muito material a ser discutido. Todo o código usado está disponibilizado como material suplementar que acompanha esse texto.

Esperamos que este minicurso sirva para despertar o interesse do leitor para a área de redes neurais profundas. A partir desta introdução, o aluno interessado pode escolher uma trilha de estudos sobre aspectos fundamentais das redes, tais como funções de ativação ou arquiteturas avançadas, ou ainda uma trilha de aplicações.

Mesmo com o grande número de trabalhos que são publicados na área a cada dia, acreditamos que as bases são as mesmas e este minicurso pode servir de arcabouço para os estudos futuros. Recomendamos ao estudante os livros listados na seção de referência bibliográfica. Alguns são mais téoricos (ex.: [Goodfellow et al. 2016]) e outros mais práticos (ex.: [Géron 2017] ou [Zhang et al. 2021]). Também recomendamos fortemente as playlists de vídeos produzidas pela UCL e pelo MIT sobre o mesmo tema.

Referências

[Abadi et al. 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Chollet 2021] Chollet, F. (2021). *Deep learning with Python*. Simon and Schuster.

[Coppin and Valério 2015] Coppin, B. and Valério, J. (2015). *Inteligência Artificial*. Grupo Gen – LTC.

[Géron 2017] Géron, A. (2017). *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media.

[Goodfellow et al. 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Hastie et al. 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer.

[He et al. 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

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

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

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

[Wang et al. 2020] Wang, Z. J., Turko, R., Shaikh, O., et al. (2020). Cnn explainer: Learning convolutional neural networks with interactive visualization. https://poloclub.github.io/cnn-explainer/.

[Zhang et al. 2021] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2021). Dive into deep learning. *arXiv preprint arXiv:2106.11342*.