



Universidade Estadual de Campinas



Faculdade de Engenharia Elétrica e de Computação

EA006: Trabalho de Fim de Curso

Introdução às redes neurais: reconhecimento de dígitos
numéricos escritos à mão

Natália Ganiku Dini, RA 184903

Orientador: Romis Ribeiro de Faissol Attux

Fevereiro de 2023
Campinas/SP

Sumário

1.	Introdução.....	3
2.	Conceitos básicos.....	4
2.1.	Perceptrons.....	4
2.2.	Neurônios com função de ativação sigmoidal.....	8
2.3.	Arquitetura de redes neurais.....	10
3.	Fundamentos para aprendizado de máquina.....	11
3.1.	Construção de uma rede neural.....	11
3.2.	Conjunto de dados e função de custo.....	12
3.3.	Gradiente descendente.....	12
3.4.	Gradiente descendente aplicado a redes neurais.....	15
3.4.1.	Gradiente descendente estocástico com <i>mini-batches</i>	15
3.5.	<i>Backpropagation</i>	15
3.6.	<i>Underfitting</i> e <i>overfitting</i>	16
3.7.	Como evitar <i>overfitting</i>	17
3.7.1.	<i>Early stopping</i>	17
3.7.2.	Regularização L1 e L2.....	17
3.7.3.	<i>Dropout</i>	18
4.	Rede neural para classificar dígitos numéricos escritos à mão.....	18
4.1.	Funções.....	18
4.2.	Carregamento do conjunto de dados.....	23
4.3.	Treinamento da rede neural e resultados.....	24
5.	Aprimoramento de performance.....	25
5.1.	Entropia cruzada como função de custo.....	26
5.2.	Função de ativação.....	27
5.3.	Inicialização de parâmetros.....	28
5.4.	Escolha de hiperparâmetros.....	29
5.4.1.	Taxa de aprendizado.....	29
5.4.2.	Parâmetro de regularização λ	30
5.4.3.	Número de camadas ocultas e de neurônios por camada oculta.....	30
5.5.	Redes neurais convolucionais.....	30
5.6.	Biblioteca: <i>Keras</i>	31
6.	Conclusão.....	34
7.	Bibliografia.....	35

1. Introdução

Redes neurais fornecem soluções de elevado desempenho para diversos problemas em áreas como reconhecimento de imagem, reconhecimento de fala e processamento de linguagem natural [1]. Quando abordamos a programação para solução de problemas de maneira convencional, “dizemos ao computador o que fazer”, quebrando grandes problemas em pequenas tarefas bem definidas que o computador pode facilmente executar. Já em uma rede neural, a máquina deve, a partir de dados, encontrar uma solução.

Neste trabalho, apresentaremos o uso de redes neurais para reconhecer dígitos numéricos escritos à mão. O problema de reconhecimento de dígitos é uma tarefa de classificação, que é típica do aprendizado supervisionado, ou seja, o conjunto de dados para treinar o algoritmo oferece as respostas desejadas (como um gabarito).



Figura 1: Números escritos à mão.

A maioria das pessoas facilmente reconhece os dígitos da Fig. 1 como ‘7’, ‘0’, ‘1’, ‘6’, ‘3’ e ‘2’. Tal facilidade é, de certa forma, ilusória: há milhões de neurônios, com bilhões de conexões entre si, que foram ajustados pela evolução por centenas de milhares de anos, trabalhando para processar o mundo visual. Essa tarefa aparentemente simples se mostra bastante complicada quando tentamos criar um programa para realizá-la. Como fazer o computador entender que o número ‘6’ tem uma parte circular embaixo e uma “perninha” que se estende para cima, pelo lado esquerdo? É difícil definir regras precisas.

Com redes neurais, abordaremos esse problema de outra maneira: vamos utilizar uma grande quantidade de dígitos escritos à mão como exemplos para treinamento e desenvolver um sistema que possa aprender a partir desses exemplos.

A rede neural usa os exemplos para automaticamente deduzir regras a fim de reconhecer os dígitos escritos à mão. Quanto maior for a quantidade de exemplos para treinamento, mais a rede poderá aprender sobre caligrafia e, conseqüentemente, melhorar sua precisão no reconhecimento.

O foco deste trabalho é no reconhecimento de dígitos escritos à mão, um problema bastante usado no aprendizado de redes neurais. Vamos apresentar, na seção 2, os conceitos centrais, a fim de construir uma base teórica sobre o tema.

2. Conceitos básicos

2.1. Perceptrons

Perceptron é um modelo de neurônio artificial desenvolvido por Frank Rosenblatt [1]. Ele recebe várias entradas reais e produz uma única saída, que, originalmente, era binária. A Fig. 2 apresenta uma ilustração.

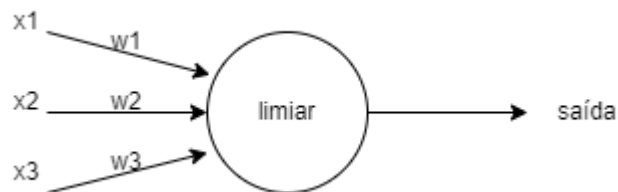


Figura 2: Representação de um perceptron.

No exemplo da Fig. 2, temos três entradas, x_1 , x_2 , x_3 . Para determinar a saída, utilizaremos pesos (w), ou seja, números reais que expressam a influência de cada entrada sobre a saída. O valor da ativação é determinado pela combinação linear $\sum_i w_i x_i$. Na formulação de Rosenblatt, caso a ativação esteja acima de um valor limite (limiar), o resultado da saída é 1. Caso contrário, é 0 (ou -1). O caráter binário da saída pode ser compreendido como sendo gerado por uma *função de ativação* não-linear do tipo degrau.

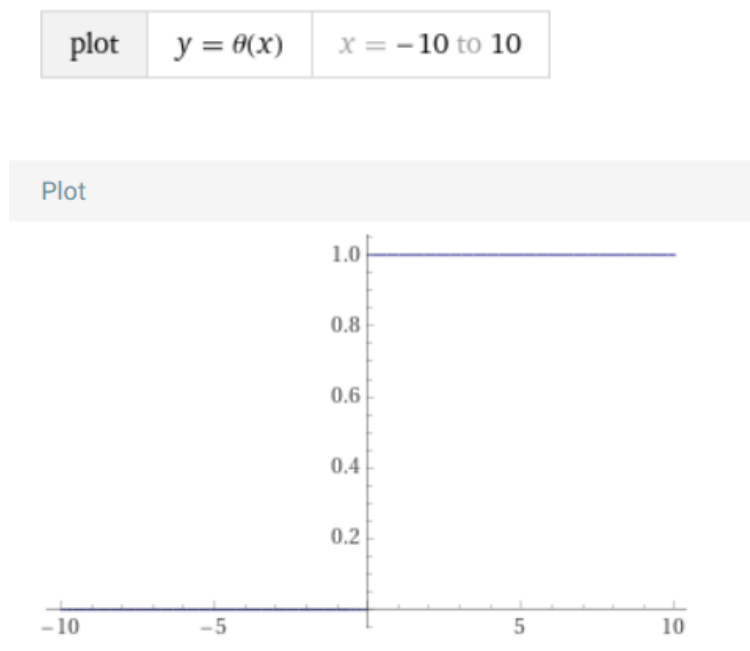


Figura 3: A saída de um perceptron é descrita pela função degrau.

Para facilitar a visualização do funcionamento do modelo apresentado, pensemos no seguinte caso hipotético: um colega não muito próximo convidou você para uma festa de

aniversário ao ar livre, que estará repleta de doces e guloseimas. Para decidir se irá ou não à festa, tomará sua decisão a partir de alguns critérios:

- x_1 : o tempo está bom?
- x_2 : seu/sua companheiro(a) vai com você?
- x_3 : há estação de metrô próxima ao local da festa?

Atribuiremos 1 para resposta positiva e 0 para resposta negativa. Suponha que você goste bastante de doces: então estaria feliz de ir mesmo que seu/sua companheiro(a) não fosse e que a estação de metrô ficasse um pouco longe. Por outro lado, você realmente detesta sair na chuva. Assim, poderíamos atribuir os seguintes valores para os pesos: $w_1 = 6$, $w_2 = 2$, $w_3 = 2$. Isso mostra que a condição do tempo é muito importante para você, bem mais do que ter companhia ou estação próxima. Suponhamos que o valor limite do perceptron seja 5. Com os pesos escolhidos, o único critério que importa é o do tempo, pois se ele for 0, automaticamente a soma estará abaixo do valor limite, e a saída será 0.

Ao variar os pesos e o valor limite, teremos diferentes modelos para a tomada de decisões. Se alterarmos o valor limite para 3, o perceptron decidirá que devemos ir à festa caso o tempo esteja bom ou caso haja conjuntamente companhia e metrô próximo. Abaixar o valor limite significa que você está mais disposto(a) a ir à festa. É claro que um perceptron não modela completamente a tomada de decisão humana, mas o caso hipotético foi apenas uma ilustração de como o perceptron pode considerar diferentes tipos de critérios a fim de tomar decisões.

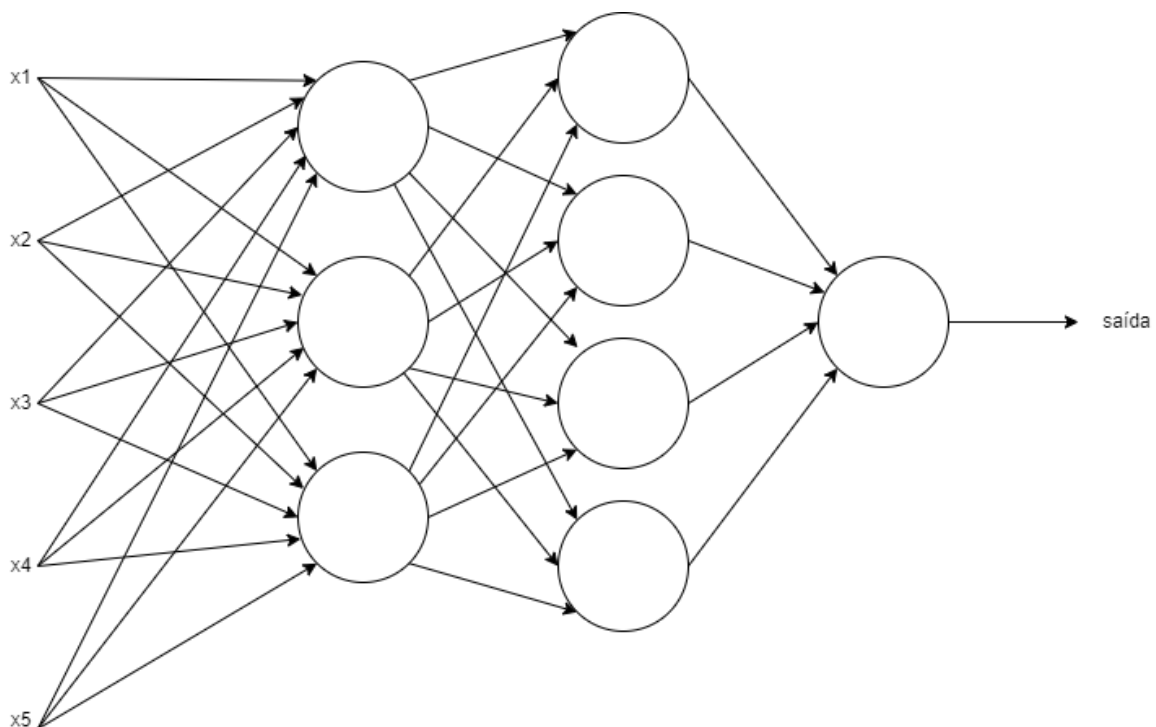


Figura 4: Rede neural com três camadas. É importante lembrar que cada perceptron possui apenas uma saída. Quando há mais de uma seta na saída, é para indicar que a mesma saída serve de entrada para mais de um perceptron.

Parece razoável que uma rede de perceptrons mais complexa possa tomar decisões mais refinadas. No exemplo da Fig. 4, a primeira camada faz três decisões simples, apenas ponderando as cinco entradas. Já os perceptrons da segunda camada tomam decisões a partir dos resultados da primeira camada. Dessa forma, pode-se dizer que esses perceptrons podem tomar decisões em um nível mais complexo e abstrato que os da primeira camada. Então, uma rede com várias camadas pode ser aplicada em uma tomada de decisões bem mais sofisticada.

Voltemos à notação. Determinamos que a saída vale 1 caso $\sum_i w_i x_i > \text{limiar}$. Vamos representar o somatório por um produto escalar $\mathbf{w} \cdot \mathbf{x}$, onde \mathbf{w} e \mathbf{x} são vetores cujos componentes são pesos e entradas, respectivamente. Se passarmos o valor do limite para o outro lado da inequação, temos que a saída é 1 se $\mathbf{w} \cdot \mathbf{x} - \text{limite} > 0$. Seja $b = -\text{limite}$. Podemos considerar que b é o *bias*, o que nos leva a:

$$saída = \begin{cases} 1 & \text{se } w \cdot x + b > 0 \\ 0 & \text{caso contrário} \end{cases}$$

Perceptrons também podem ser usados para computar funções lógicas básicas, como AND, OR, NAND etc. Suponha um perceptron com duas entradas, cujos pesos são -2 e cujo *bias* é 3, como mostrado na Fig. 5.

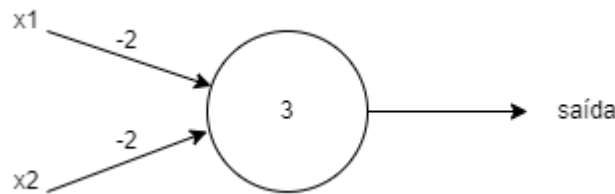


Figura 5: Perceptron que computa uma porta NAND.

Com ambas as entradas em 0, temos:

$$(-2) * 0 + (-2) * 0 + 3 = 3 > 0 \Rightarrow saída = 1$$

Fazendo as contas para todos os valores de entradas, obtemos:

x_1	x_2	$w \cdot x + b$	<i>saída</i>
0	0	3	1
0	1	1	1
1	0	1	1
1	1	-1	0

Tabela 1: Valores das entradas e saída do perceptron que implementa porta NAND.

O exemplo acima mostra que perceptrons podem ser utilizados para computar funções lógicas básicas. A partir da porta NAND, é possível construir as portas lógicas NOT, AND e OR, conforme ilustrado na Fig. 6.

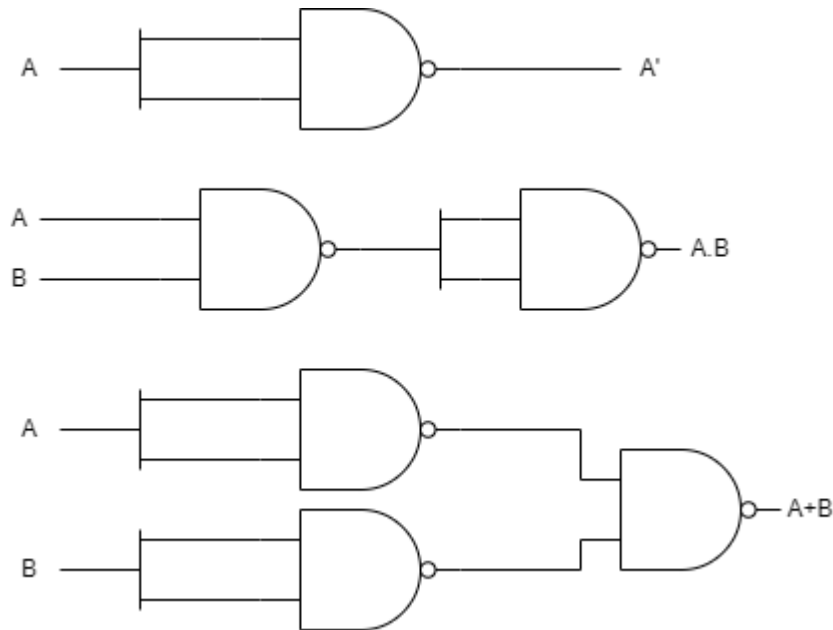


Figura 6: Portas lógicas NOT, AND e OR construídas a partir de NANDs.

Dessa forma, também podemos implementar circuitos lógicos mais complexos, como o somador de 2 bits, com carry, mostrado na Fig. 7.

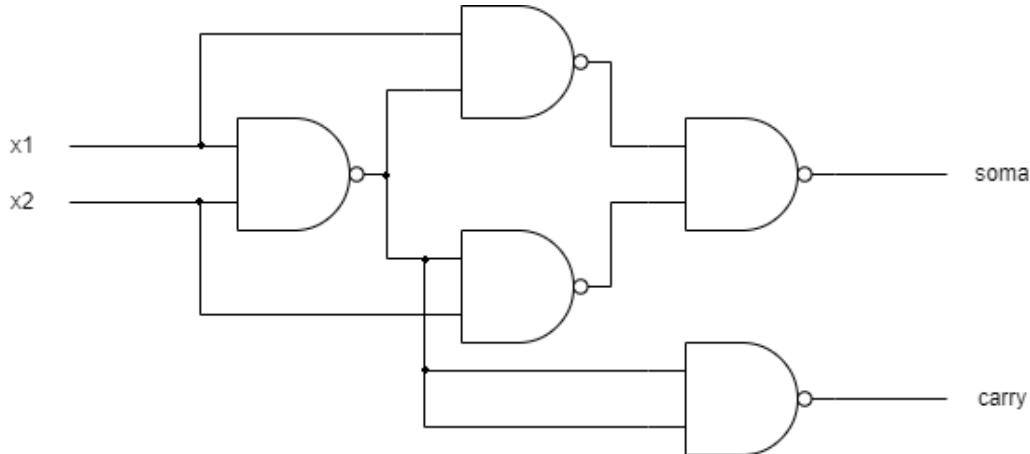


Figura 7: Somador de 2 bits construído a partir de portas NAND.

Para obter o equivalente com uma rede de perceptrons, basta substituir todas as portas NAND por perceptrons com duas entradas de peso -2 cada e *bias* 3. O exemplo do somador mostra como uma rede de perceptrons pode ser usada para simular um circuito que contém várias portas NAND, que são universais na computação - implicando que perceptrons também o são [2]. Porém, com perceptrons, podemos criar algoritmos de aprendizagem que automaticamente regulam os pesos e *biases* de uma rede de neurônios artificiais. Essa regulação acontece como resposta a estímulos externos, sem intervenção direta do(a) programador(a).

Vamos supor que queremos uma rede de perceptrons para resolver o seguinte problema: as entradas da rede são pixels de uma imagem escaneada de um dígito escrito à mão, e desejamos que a rede descubra os pesos e *bias* para que sua saída seja a classificação correta do dígito. Para entendermos o funcionamento da rede, vamos então alterar de forma mínima os valores de algum peso ou *bias* dela. O que esperamos é que esta pequena mudança cause uma alteração também pequena na saída da rede, como mostrado na Fig. 8..

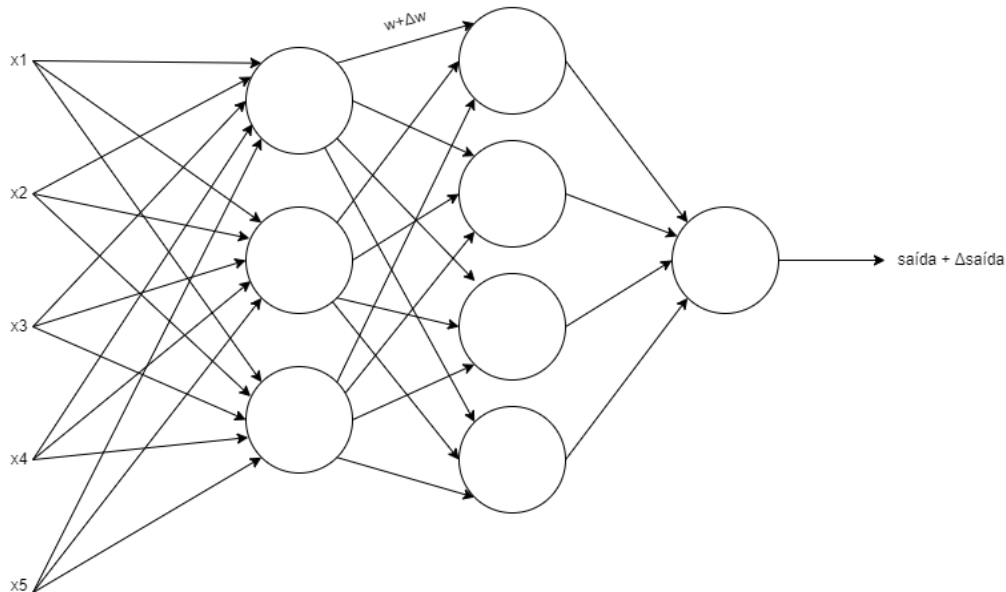


Figura 8: Pequena mudança em um peso causa pequena mudança na saída.

Se é verdade que uma pequena mudança em um peso ou *bias* causa apenas uma pequena mudança na saída, então podemos usar isso para modificar pesos e *bias* a fim de fazer a rede se comportar de forma mais próxima daquilo desejamos. Por exemplo, se uma rede classifica erroneamente uma imagem como “7” quando deveria ser “9”, podemos descobrir uma forma de alterar minimamente os pesos e *bias* para que a rede chegue mais perto de classificar a imagem como um “9”. E repetiremos o processo, alterando devagar e em pequenas doses, para produzir saídas cada vez mais acuradas: a rede estaria aprendendo!

O problema é que isso não acontece em uma rede de perceptrons clássicos. Uma pequena alteração em um peso ou *bias* em qualquer perceptron da rede pode causar uma saída completamente diferente, afetando todo o resto da rede. Se antes o problema era só classificar o “9” incorretamente, agora pode ser que outros dígitos - antes corretos - passem a ser classificados erroneamente. Isso tudo torna difícil ver como modificar gradualmente os pesos e *bias* para ajustar a rede conforme desejado.

Note que não é possível determinar o valor da função na transição de 0 para 1. Isso é uma das motivações que levaram à proposta que veremos a seguir.

2.2. Neurônios com função de ativação sigmoidal

Podemos contornar o problema descrito na seção anterior com neurônios com funções de ativação sigmoidais (*sigmoid*): são extensões do perceptron clássico, porém pequenas alterações em seus pesos e *bias* causam apenas pequenas mudanças em suas saídas. A

diferença do neurônio *sigmoid* para o perceptron é sua função de ativação é uma extensão da função degrau no sentido de possuir continuidade. A saída é determinada por:

$$saída = \frac{1}{1 + \exp(-w \cdot x - b)}$$

A função $\sigma(z) = \frac{1}{1 + e^{-z}}$ é conhecida como função *sigmoid*, mostrada na Fig. 9.

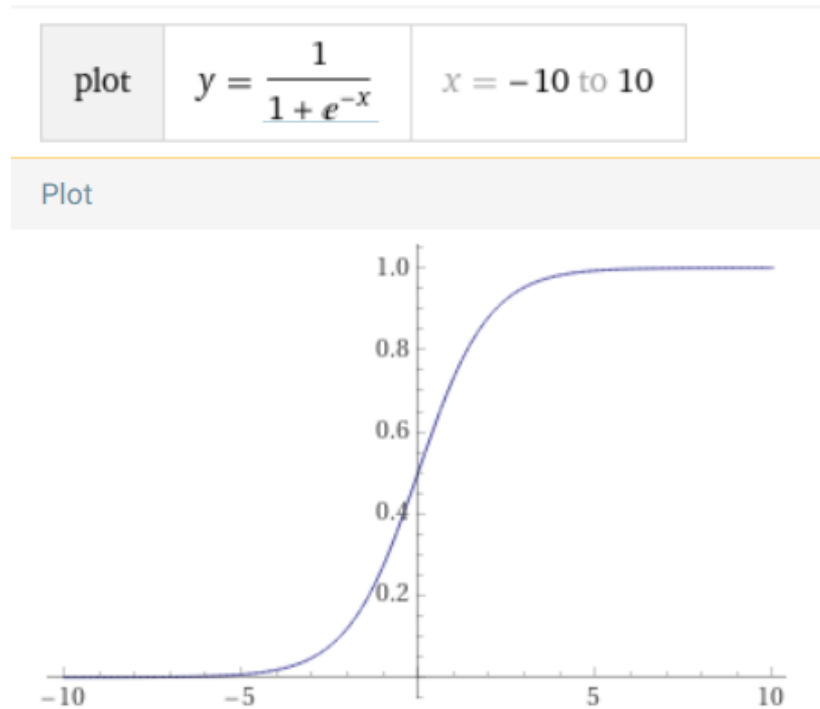


Figura 9: Função *sigmoid*.

Dado que $\sigma(z) = \frac{1}{1 + e^{-z}}$, onde $z = w \cdot x + b$, note que: a saída σ se aproxima de 1 para valores grandes e positivos de z ; enquanto a saída σ se aproxima de 0 para valores grandes e negativos de z . É o comportamento esperado para um perceptron também. A diferença se encontra justamente nos valores intermediários.

A função *sigmoid* suaviza a curva de transição entre 0 e 1, o que é essencial para que pequenas alterações em pesos e *biases* produzam mudanças proporcionalmente pequenas nas saídas dos neurônios. De fato, temos:

$$\Delta saída \simeq \sum_i \frac{\partial saída}{\partial w_i} \Delta w_i + \frac{\partial saída}{\partial b} \Delta b$$

onde $\frac{\partial saída}{\partial w_i}$ e $\frac{\partial saída}{\partial b}$ são as derivadas parciais da saída em relação a w_i e b , respectivamente.

A expressão acima nos diz que a alteração $\Delta saída$ é uma função que depende linearmente das mudanças nos pesos e *biases*, Δw_i e Δb . Isso significa que é relativamente fácil escolher pequenas alterações a fim de obter a saída desejada.

2.3. Arquitetura de redes neurais

Vimos um exemplo de rede neural na Fig. 4. Para que as entradas x_1, x_2, x_3, x_4, x_5 não fiquem “flutuando no diagrama”, vamos representá-las por círculos na Fig. 10.

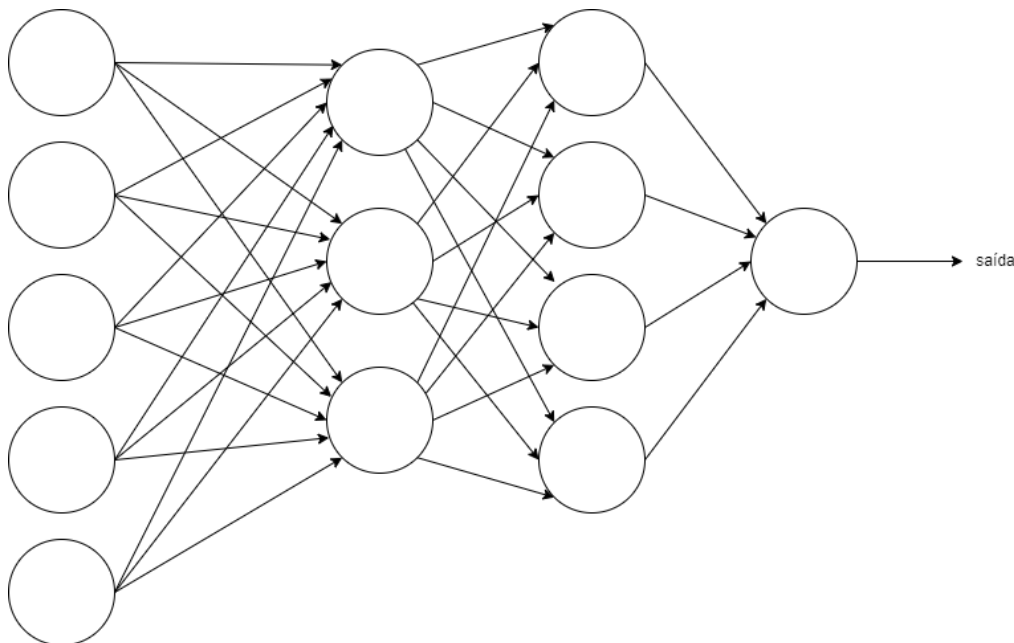


Figura 10: Exemplo de neural. A camada mais à esquerda representa os neurônios de entrada, enquanto a camada mais à direita representa os neurônios de saída (aqui há apenas um neurônio). A camada intermediária é chamada de camada oculta ou escondida, já que seus neurônios não são entradas nem saídas. Aqui existem duas camadas ocultas.

O desenho das camadas de entradas e saídas é bastante direto e simples. Se desejamos fazer uma rede que determina se uma imagem de número escrito à mão é ou não um “4”, uma maneira de projetar a rede é codificar nos neurônios de entrada as intensidades dos pixels da imagem. Se esta é uma imagem de 64x64 pixels numa escala de cinza, então teremos 4096 neurônios de entrada, com as intensidades adequadamente dimensionadas entre 0 e 1. A camada de saída possui apenas um neurônio, que indicará que a imagem é um “4” se o valor produzido for maior ou igual a 0.5, ou indicará que não é um “4” se o valor produzido for menor do que 0.5.

É importante ressaltar que as saídas dos neurônios das redes neurais apresentadas neste trabalho vão sempre para a camada seguinte. Por exemplo, os resultados gerados na segunda camada necessariamente vão para a terceira, não voltam para a primeira. É possível criar redes que se retroalimentam, porém, aqui, não focaremos nisso. Redes cujos valores de saídas só são passadas adiante se chamam *feedforward*.

Enquanto projetar as camadas de entrada e saída é relativamente fácil, o mesmo não pode ser dito para as camadas ocultas. Não há uma regra que indique quantas camadas devem existir, apenas heurísticas para conseguirmos obter o comportamento desejado.

3. Fundamentos para aprendizado de máquina

3.1. Construção de uma rede neural

Primeiramente, vamos definir o tipo de problema a ser resolvido: queremos uma rede neural que identifique o desenho contido em uma imagem como números entre 0 e 9. Cada imagem tem 28x28 pixels, o que implica uma camada de entrada de 784 neurônios. O valor que chega em cada entrada varia entre 0 (branco) e 1 (preto). Os valores intermediários estão numa escala de cinza.

Teremos apenas uma camada oculta, com n neurônios. Serão testados diferentes valores para n .

A camada de saída terá 10 neurônios. Se o primeiro neurônio acende, ou seja, tem saída 1, significa que a rede classificou o dígito como “0”. Se o segundo neurônio acende, a rede classificou o dígito como “1”. E assim sucessivamente até o décimo neurônio, que indica que a rede classificou o dígito como “9”. Para facilitar, vamos chamar de neurônio 0 para o dígito 0, neurônio 1 para o dígito 1, e assim por diante.

Alguém que está acostumado a criar circuitos com portas lógicas poderia dizer que é possível fazer uma rede neural com apenas 4 saídas, afinal, com 4 bits podemos representar números de 0 a 15. Precisaríamos testar ambas as formas para saber qual método é mais eficiente. A resposta é: utilizar 10 neurônios de saída é melhor. Será que existe alguma heurística que nos ajude a prever qual método é melhor?

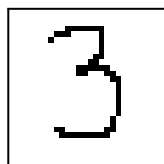


Figura 11: Imagem de 28x28 pixels.

Se a rede com 10 neurônios funciona corretamente, o neurônio 3 da saída deve acender quando classificar a imagem da Fig. 11, enquanto todos os outros se mantêm apagados. Vamos supor que na camada oculta existam neurônios que detectem os seguintes traços:

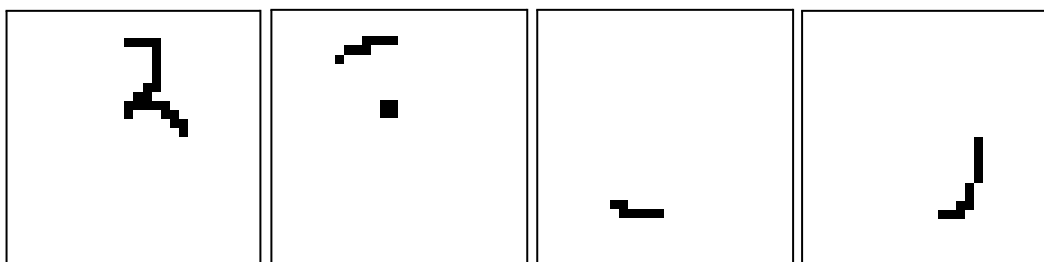


Figura 12: Traços do número “3”, separados em quadrantes.

Vamos considerar que:

- na camada oculta existem neurônios que pesam fortemente traços que se sobrepõem aos demonstrados na Fig. 12, ao passo que pesam levemente traços que não estão presentes;

- as saídas desses 4 neurônios da camada oculta vão para o neurônio 3 na camada de saída; ou seja, se esses 4 neurônios estão acesos, logo o neurônio 3 também acende.

Agora, se tomarmos a rede com 4 neurônios de saída, a rede precisa decidir, por exemplo, se o primeiro neurônio, o que representa o bit mais significativo (*most significant bit*, ou MSB), está aceso: isso engloba os números 8, 9 e 10. O bit menos significativo (*least significant bit*, ou LSB) está aceso se o número é ímpar. É difícil sabermos essas características a partir dos traços das imagens. Talvez isso nos dê uma ideia do motivo pelo qual um *design* é melhor do que o outro. Mas vale lembrar que isso é heurística. Nada garante que a rede de fato funciona conforme foi descrito acima, nem que não exista algum algoritmo que encontre uma solução mais eficiente para 4 neurônios de saída.

3.2. Conjunto de dados e função de custo

Temos um projeto para nossa rede neural. E agora, como fazê-la aprender a reconhecer os dígitos? Precisaremos de um conjunto de dados! Vamos utilizar um *dataset* bastante conhecido na literatura, o MNIST (Modified National Institute of Standards and Technology). Ele contém milhares de imagens (de 28x28 pixels) de dígitos escritos à mão, juntamente com seus rótulos corretos. Está separado em conjunto de treinamento (60 mil imagens) e conjunto de teste (10 mil imagens). O MNIST é considerado um “*Hello World*” do aprendizado de máquina, pois é muitas vezes utilizado como primeiro teste para verificar o desempenho de algoritmos de classificação novos.

Vamos à notação agora: x representa uma entrada de treinamento, que será um vetor de dimensão 784 (pois cada imagem contém 28x28 pixels). Cada componente no vetor indica o valor na escala de cinza de cada pixel. Vamos representar a saída desejada correspondente por $y = y(x)$, que será um vetor de dimensão 10. Em outras palavras: se uma imagem contém uma imagem do número “7”, a saída deve ser $y(x) = (0, 0, 0, 0, 0, 0, 0, 1, 0, 0)^T$. Buscamos um algoritmo que descubra os pesos e *biases* para que a saída da rede aproxime corretamente $y(x)$ para todos os valores de entrada de treinamento x . Para medirmos quão boa está a aproximação, utilizamos a função de custo ou perda:

$$C(w, b) = \frac{1}{2n} \sum^n (y(x) - a)^2$$

onde w são os pesos da rede, b são os *biases*, n é a quantidade de entradas de treinamento, a é o vetor de saídas da rede ($a = w \cdot x + b$). A função de custo também é conhecida por erro quadrático médio, ou MSE (*mean squared error*).

Veja que a função custo é sempre maior ou igual a zero. Se $C(w, b)$ for pequeno (próximo de zero), significa que $a \simeq y(x)$, ou seja, o valor da saída está próximo ao esperado. Indica que o algoritmo encontrou valores adequados de pesos e *biases* para todas as entradas de treinamento. Por outro lado, se o valor de $C(w, b)$ for grande, quer dizer que a saída a está bem longe do esperado para muitas entradas. Queremos encontrar um conjunto de pesos e *biases* que minimize a função custo.

3.3. Gradiente descendente

Nesta seção vamos desenvolver o conceito de gradiente descendente, que será utilizado para resolver o problema de minimização citado anteriormente. Seja $F(v)$ uma

função real multivariável ($v = v_1, v_2, \dots$). Queremos encontrar o valor de v que minimize F . Para ilustrar o problema, consideremos a imagem abaixo:

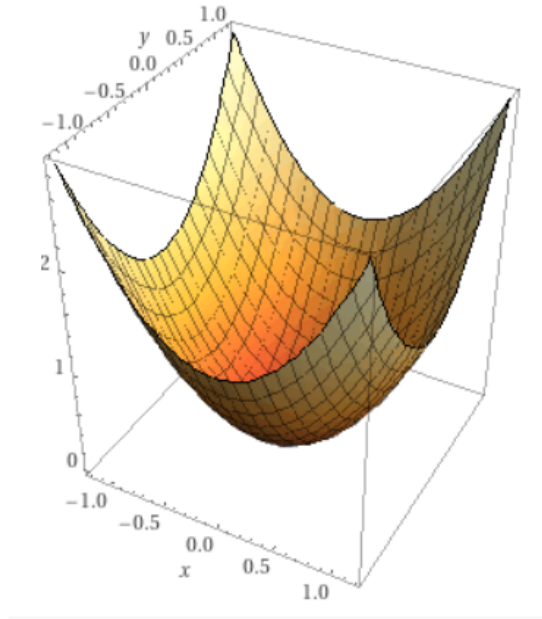


Figura 13: Paraboloide infinito: $F(v) = v_1^2 + v_2^2$.

Precisamos encontrar onde F atinge o mínimo global. O exemplo da Fig. 13 é simples, podemos facilmente identificar onde a função atinge seu valor mínimo. Porém, F pode assumir formas bastante complicadas, e é difícil detectar o ponto de mínimo.

Imaginemos que a Fig. 13 representa um campo por onde podemos andar. Queremos encontrar o ponto mais baixo desse campo. Se um passo é dado em determinada direção, sabemos se subimos ou descemos. Podemos dizer que um pequeno passo é determinado por um Δv_1 na direção v_1 e um Δv_2 na direção v_2 . Então sabemos que a alteração no relevo ΔF é dada por:

$$\Delta F = \frac{\partial F}{\partial v_1} \Delta v_1 + \frac{\partial F}{\partial v_2} \Delta v_2$$

Se $\Delta F < 0$, significa que estamos descendo. Precisamos encontrar Δv_1 e Δv_2 de forma a tornar ΔF negativo. Tomemos $\Delta v = (\Delta v_1, \Delta v_2)^T$. Definimos o gradiente de F como o vetor de derivadas parciais:

$$\nabla F = \left(\frac{\partial F}{\partial v_1}, \frac{\partial F}{\partial v_2} \right)^T$$

Vamos interpretar o que o gradiente de F , representado por ∇F , significa: ele relaciona mudanças em v_1 e v_2 com as mudanças em F . Reescrevendo ΔF em termos de Δv e ∇F , temos:

$$\Delta F = \nabla F \cdot \Delta v$$

Com a equação acima, podemos escolher Δv de modo a fazer com que ΔF seja negativo. Se escolhermos $\Delta v = -\eta \nabla F$, onde η é um número pequeno e positivo (conhecido como taxa de aprendizado), então temos que:

$$\Delta F = \nabla F \cdot \Delta v = -\eta |\nabla F|^2$$

Como $|\nabla F|^2$ é sempre maior ou igual a zero, significa que ΔF sempre será menor ou igual a zero se escolhermos $\Delta v = -\eta \nabla F$. Era o que precisávamos: encontrar Δv_1 e Δv_2 , que posteriormente denotamos $\Delta v = (\Delta v_1, \Delta v_2)^T$, de forma a tornar ΔF negativo.

Seja v_o o valor inicial de v . O próximo passo seria $v = v_o - \eta \nabla F$. Depois repetimos o processo - o que decresce o valor de F - até encontrarmos o mínimo global (é o que se espera!). É necessário ter cuidado: devemos escolher a taxa de aprendizado η suficientemente pequena para que $\Delta F = \nabla F \cdot \Delta v$ se mantenha uma boa aproximação e para que não aconteça de ΔF se tornar positivo. Como?

Reveja a Fig. 13, que ilustra o paraboloide infinito. Observe suas curvas de nível na Fig. 14 e imagine o ponto atual representado pela bolinha azul.

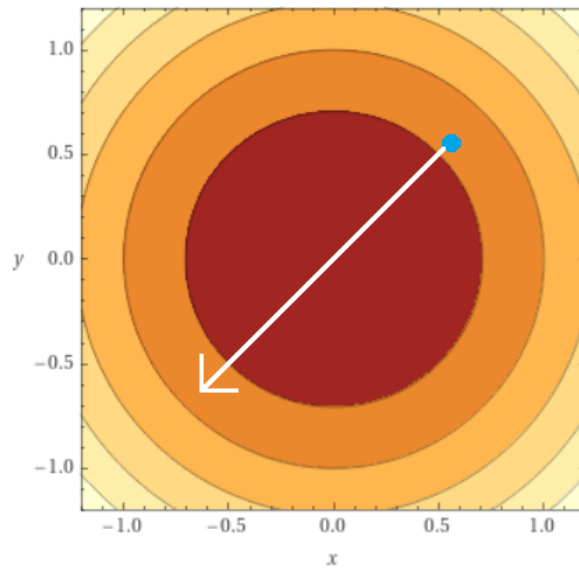


Figura 14: Curvas de nível de um paraboloide infinito $F(v) = v_1^2 + v_2^2$.

Se a taxa de aprendizagem η for grande demais, pode ser que o próximo passo não faça F decrescer. A imagem é um exemplo exagerado, mas ilustra bem o problema de escolher um valor inadequado para η .

Por outro lado, tampouco queremos η pequeno demais, pois assim o algoritmo seria muito lento. Precisamos do suficiente para manter $\Delta F = \nabla F \cdot \Delta v$ como uma boa aproximação.

É possível provar que o valor de Δv que faz F decrescer ao máximo é $\Delta v = -\eta \nabla F$, onde $\eta = \frac{\epsilon}{|\nabla F|}$ é determinado pela restrição $|\Delta v| = \epsilon$, mas não entraremos nesse escopo neste trabalho.

Resumindo: temos que $\Delta v = -\eta \nabla F$ é tamanho do passo em v para caminharmos na direção em que F decresce.

Vimos o gradiente decrescente utilizando uma função de duas variáveis como exemplo, mas $F(v)$ pode ter o tamanho que for!

3.4. Gradiente descendente aplicado a redes neurais

Utilizaremos o conceito de gradiente descendente para encontrar os pesos e *biases* que minimizam a função de custo $C(w, b) = \frac{1}{2n} \sum (y(x) - a)^2$. Vamos reescrever o que vimos na seção anterior, aplicando agora as variáveis da função de custo:

- Para os pesos: $w_{k+1} = w_k - \eta \frac{\partial C}{\partial w_k}$
- Para os *biases*: $b_{l+1} = b_l - \eta \frac{\partial C}{\partial w_l}$

Repetindo o processo acima, esperamos encontrar o valor mínimo da função de custo. Lembre-se: minimizar a função custo $C(w, b)$ significa que o algoritmo está aprendendo, pois ele encontra os valores de pesos e *biases* que aproximam a saída da rede com o valor do “gabarito” (conjunto de treinamento).

3.4.1. Gradiente descendente estocástico com *mini-batches*

A palavra “estocástico” se refere, a grosso modo, à aleatoriedade. Dessa forma, calcular o gradiente descendente estocástico é determinar seu valor a partir de entradas de treinamento aleatoriamente selecionadas, em vez de utilizar o conjunto de dados de treinamento inteiro.

Criam-se pequenos lotes, ou *mini-batches*, com entradas de treinamento aleatórias e, a partir deles, calcula-se o gradiente descendente. A quantidade de amostras deve ser suficientemente grande para que a média de ∇C encontrada a partir delas seja próxima do valor de ∇C encontrado quando se usa o *batch* inteiro.

A vantagem de utilizar *mini-batches* é que a estimativa se torna mais rápida.

3.5. Backpropagation

Backpropagation é, como o nome sugere, propagação para trás. Não entraremos em detalhes matemáticos de seu funcionamento, mas oferecemos uma intuição.

Backpropagation leva em conta a derivada parcial da função de custo em relação ao peso ou ao *bias*, o que nos diz quão rapidamente o custo varia quando w e b mudam, ao considerar erros de cada neurônio. Ou seja, é possível calcular a dependência do erro com cada peso da rede.

Após encontrar os erros para cada camada, examina-se o erro final (da última camada) propagando-o para trás. Considere a Fig. 15.

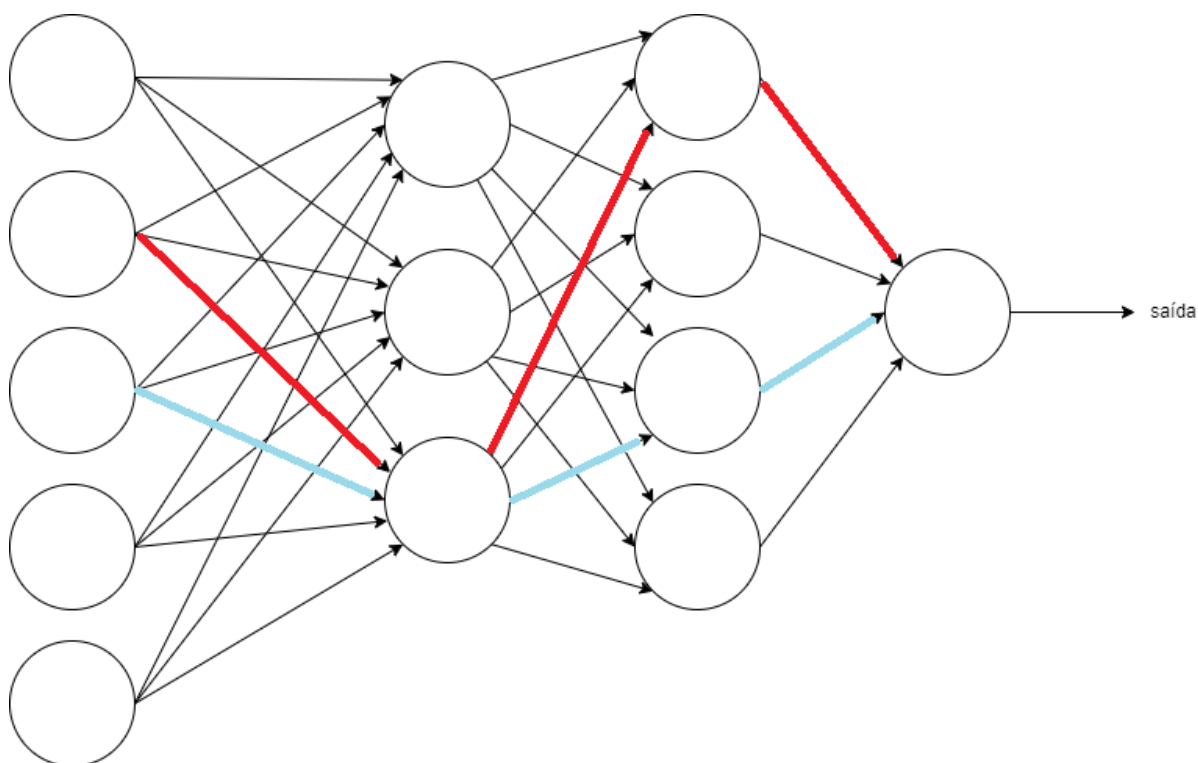


Figura 15: Exemplo de rede neural. Suponha que o caminho em vermelho possui pesos ou *biases* que afetam fortemente a saída, enquanto o caminho em azul não tanto. As derivadas parciais da função de custo indicarão isso.

Se algum neurônio influencia fortemente o resultado final, significa que também tem impacto grande no erro, pois pode amplificá-lo. Ter conhecimento de quais são esses neurônios ajuda a otimizar o processo da escolha de pesos que devem ser alterados.

O custo computacional para fazer isso é praticamente o mesmo de alimentar a rede para frente (o *feedforward*), o que é bastante razoável: bem melhor do que calcular regra da cadeia inúmeras vezes!

3.6. *Underfitting* e *overfitting*

Sub-ajuste e sobre-ajuste, ou *underfitting* e *overfitting*, são problemas que podem acontecer em aprendizado de máquina.

Suponha um conjunto de dados que, quando plotados, claramente formam algo próximo a uma parábola. Se o seu modelo utiliza uma reta $y(x) = ax + b$, ele nunca encontrará a e b adequados para se aproximar do comportamento apresentado pelos dados. Isso se chama *underfitting*.

Por outro lado, no *overfitting* ocorre o oposto: é quando o modelo se ajusta tão bem aos dados de treinamento a ponto de encontrar uma curva que descreve seu comportamento perfeitamente. Vamos novamente ao exemplo polinomial: se os dados apresentam comportamento de parábola, porém há pontos que possuem ruído e se encontram fora da curva, um modelo que utiliza um polinômio de quinto grau $y(x) = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$ pode encontrar uma curva que engloba até mesmo os dados com ruído como parte do comportamento do conjunto. Isso significa que o

modelo dará respostas erradas para o conjunto de dados de teste (e qualquer outro que não seja o de treinamento), pois é como se ele tivesse decorado a resolver o problema apenas para o conjunto de dados de treinamento.

No contexto de redes neurais, esses problemas podem acontecer caso a quantidade de neurônios ou os valores de pesos e *biases* sejam inadequados. O *overfitting* é mais comum, já que redes neurais podem chegar a ter centenas ou até milhares de parâmetros, o que lhes dá bastante flexibilidade.

3.7. Como evitar *overfitting*

3.7.1. *Early stopping*

Um método bastante popular para evitar o *overfitting* é a parada antecipada, ou *early stopping*. Trata-se de interromper o treinamento do algoritmo quando o desempenho dos resultados para o conjunto de testes piora. Existem outras técnicas para impedir o *overfitting*, que podem inclusive ser combinadas com o *early stopping*.

3.7.2. Regularização L1 e L2

Há uma técnica de regularização, dentre diversas existentes, conhecida por decaimento de peso (*weight decay*), ou regularização L2. Consiste em adicionar um termo extra à função de custo e ajustá-lo por um fator $\frac{\lambda}{2n}$:

$$C(w, b) = \frac{1}{2n} \sum^n (y(x) - a)^2 + \frac{\lambda}{2n} \sum w^2$$

onde λ é o parâmetro de regularização, n é o tamanho do conjunto de treinamento. Note que o termo extra é a soma dos quadrados dos pesos da rede. Na seção 5 discutiremos brevemente como o valor de λ pode ser escolhido.

A ideia da regularização L2 é fazer com que a rede dê preferência a pesos de valores baixos, a menos que um valor alto melhore consideravelmente a função de custo original. O parâmetro λ é quem dita essa regra: se seu valor for pequeno, significa que minimizar a função de custo original é mais importante; caso contrário, significa que utilizar valores baixos de w é mais importante.

Isso reduz o *overfitting* porque valores baixos para os pesos significa que a rede não mudará drasticamente se alterarmos algumas entradas. Vale comparar com o exemplo polinomial visto na seção 3.6. Quanto maior o grau do polinômio, mais flexibilidade tem o algoritmo, e portanto mais facilmente se ajusta a ruídos - o que é indesejável. Com valores baixos para os pesos, ocorre algo semelhante: a rede se torna menos flexível a ruído.

Outra técnica famosa, a regularização L1, é bem parecida com a apresentada anteriormente: o termo extra é $\frac{\lambda}{n} \sum |w|$, ou seja, somar os pesos absolutos da rede. A diferença é que, na regularização L1, os pesos são diminuídos por um tamanho fixo, enquanto na L2 eles são diminuídos proporcionalmente dependendo do valor de w . Ou seja, se um peso tem um valor absoluto grande, a diminuição proporcionada pela regularização L1 é relativamente bem menor do que aquela proporcionada pela regularização L2. Se um peso

tem um valor absoluto pequeno, a regularização L1 penaliza muito mais, quando comparada à regularização L2.

3.7.3. Dropout

Outro método de regularização conhecido é o de descarte, ou *dropout*. Não altera a função de custo, mas sim a própria rede neural.

Imagine que treinamos, com o mesmo conjunto de dados, dez redes neurais, das quais seis classificam uma dada entrada como “2”. Entendemos que provavelmente a saída é de fato um “2”, e quatro redes neurais erraram a classificação. É como se houvesse uma votação de qual é o resultado correto, e a partir disso é feita uma média.

O *dropout* apresenta uma ideia semelhante: consiste em eliminar temporariamente parte dos neurônios da camada oculta (geralmente metade) e treinar a rede. Depois voltamos à rede original e eliminamos outro conjunto de neurônios da camada oculta para treinar a rede. Repetimos o processo diversas vezes. Eventualmente executamos a rede inteira, com todos os seus neurônios. Para compensar o fato de que ela veio sendo treinada com metade de seus neurônios, agora que ela está com todos eles (ou seja, o dobro), dividimos por dois os valores dos pesos das camadas ocultas.

É como se tivéssemos treinado diversas redes diferentes e calculássemos uma média para os pesos depois. Isso diminui o *overfitting* por tornar a rede mais robusta, já que um neurônio não pode contar com a presença de outros neurônios específicos. O *dropout* é uma técnica computacionalmente cara, porém extremamente poderosa, bastante utilizada para treinar redes neurais com diversas camadas.

4. Rede neural para classificar dígitos numéricos escritos à mão

Vamos escrever o programa em *python*, no *Google Colab*. A pasta com os dados e códigos pode ser acessada em: [Google Drive: EA006](https://drive.google.com/drive/folders/1sGBZ4yiEG0wX2RRkqMbJ0FZ6LKkkfiwM?usp=sharing)¹

Para que o código funcione adequadamente, crie uma pasta chamada **ea006** em seu próprio *Google Drive* e copie os arquivos contidos no *link* acima para ela. Isso é necessário porque o código buscará os dados em `/content/drive/MyDrive/ea006`.

4.1. Funções

Vamos desenvolver algumas funções básicas que executam as tarefas apresentadas nas seções de conceitos e fundamentos. O código foi retirado do livro “*Neural Networks and Deep Learning*”, de Michael Nielsen, porém contém pequenas alterações.

```
Python
import random
import numpy as np

class Network (object):
    def __init__(self, sizes):
```

¹ <https://drive.google.com/drive/folders/1sGBZ4yiEG0wX2RRkqMbJ0FZ6LKkkfiwM?usp=sharing>

```

"""
    sizes é uma lista cujos elementos representam o número de neurônios em
    cada camada da rede. Se a lista é [3,5,1], então a primeira camada tem 3
    neurônios, a segunda tem 5 e a terceira tem 1. Os pesos e biases são
    inicializados aleatoriamente com uma distribuição gaussiana de média 0 e
    variância 1.
"""

self.num_camadas = len(sizes)
self.sizes = sizes
self.biases = [np.random.randn(y,1) for y in sizes[1:]]
self.pesos = [np.random.randn(y,x) for x,y in zip(sizes[:-1], sizes[1:])]

```

Aqui guardamos o tamanho da lista na variável *num_camadas*, *sizes* é a lista que armazena a quantidade de neurônios em cada camada, *biases* é a lista que armazena os valores de *bias* aleatoriamente gerados para cada neurônio (exceto os da primeira camada, pois neurônios de entrada não possuem *bias*) e *pesos* é a lista que armazena os valores de pesos aleatoriamente gerados para conexão de neurônio a neurônio.

```

sizes=[2,3,1]
biases=[np.random.randn(y,1) for y in sizes[1:]]
pesos=[np.random.randn(y,x) for x,y in zip(sizes[:-1], sizes[1:])]
print(biases)
print(pesos)

[array([[ -1.94183469],
        [ -1.22201819],
        [  0.41447496]]), array([[0.53334608]])]
[array([[ 0.59155031, -1.66855393],
        [ 1.19129959,  0.45571547],
        [-0.60396162,  0.23364823]]), array([[ 1.32871529, -1.54046394, -1.50680854]])]

```

Figura 16: Exemplo que ilustra funcionamento do trecho de código apresentado acima.

Vemos que há 3 camadas: a primeira possui 2 neurônios, a segunda possui 3 neurônios e a terceira possui 1 neurônio. Note que foram gerados 4 valores para *biases* (3 valores para a 2ª camada e 1 valor para a 3ª camada) e 9 valores para *pesos* (6 valores para as conexões entre 1ª e 2ª camadas e 3 valores para as conexões entre 2ª e 3ª camadas).

Agora que já temos número de camadas e de neurônios, bem como valores iniciais para pesos e *biases*, vamos escrever a função de ativação dos neurônios. Conforme visto na seção 2.2, utilizaremos a função *sigmoid*, $\sigma(z) = \frac{1}{1+e^{-z}}$.

Python

```
def sigmoid(z):  
    return 1.0/(1.0+np.exp(-z))
```

Precisamos também criar uma função que retorne a saída da rede neural. Para isso, é necessário calcular a saída de cada neurônio da rede, ou seja, realizar o cálculo $saída = \frac{1}{1+exp(-w \cdot x - b)}$ para cada um deles. Lembre-se que $w \cdot x$ é um produto escalar entre os vetores w e x . Para uma rede que não se retroalimenta, teremos:

Python

```
def feedforward(self, a):  
    for b,w in zip(self.biases, self.pesos):  
        a = sigmoid(np.dot(w,a)+b)  
    return a
```

Agora queremos que a rede aprenda! Vimos que, para isso, ela utiliza o gradiente descendente (aqui, será o estocástico com *mini-batches*). Em inglês, a sigla é SGD, de *stochastic gradient descent*.

Python

```
def SGD(self, dados_treinamento, epochs, mini_batch_size, eta,  
        dados_teste=None):  
    """  
    Treina a rede neural utilizando o gradiente descendente estocástico com  
    mini-batches. Os dados de treinamento é uma lista de tuplas (x,y) que  
    representam as entradas de treinamento e as saídas desejadas. A variável  
    epochs é a quantidade de epochs de treinamento. A variável mini_batch_size é  
    o tamanho dos mini-batches que serão usados na amostragem. A variável eta é  
    a taxa de aprendizado. Se dados_teste forem fornecidos, a rede será avaliada  
    com os dados de teste após cada epoch e mostrará o progresso parcial - o que  
    pode ser útil, mas torna o processo consideravelmente mais lento.  
    """  
  
    if dados_teste:  
        n_test = len(dados_teste)  
  
        n = len(dados_treinamento) #(entradas,resultados)  
  
    for j in range(epochs):  
        random.shuffle(dados_treinamento)
```

```

mini_batches = [dados_treinamento[k:k+mini_batch_size] for k in
range(0,n,mini_batch_size)]

for mini_batch in mini_batches:
    self.atualiza_mini_batch(mini_batch, eta)
if dados_teste:
    print("Epoch {0}: {1} / {2}".format(j,self.evaluate(dados_teste),
n_test))
else:
    print("Epoch {0} completa".format(j))

def evaluate(self, dados_teste):
    """
    Retorna o número de entradas de teste para os quais a rede neural dá a
    resposta correta.
    """
    resultados_teste = [(np.argmax(self.feedforward(x)), y) for (x, y) in
test_data]
    return sum(int(x == y) for (x, y) in resultados_teste)

```

O trecho do código acima embaralha a cada *epoch* os dados de treinamento para criar os *mini-batches* e depois, para cada *mini-batch*, atualiza os valores dando um passo apontado pelo gradiente descendente, através da função `atualiza_mini_batch()`. Em seguida, é avaliado se os dados de teste foram fornecidos para avaliar o progresso do aprendizado.

```

Python
def atualiza_mini_batch(self, mini_batch, eta):
    """
    Atualiza os pesos e biases da rede: aplica o gradiente descendente com
    backpropagation em um mini-batch. A variável mini_batch é uma lista de tuplas
    (x,y). A variável eta é a taxa de aprendizado.
    """

    # Cria lista de zeros para biases e pesos de todos os neurônios
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.pesos]

    for x,y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

    self.pesos = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.pesos,
nabla_w)]

```

```
self.biases = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.biases,
nabla_b)]
```

A função acima calcula o gradiente da função custo para cada dado de treinamento dentro do *mini-batch*, chamando outra função, `backprop()`, e depois atualiza os pesos e *biases* (conforme equações vistas nas seções 3.3 e 3.4).

Python

```
def backprop(self, x, y):
    """
    Retorna uma tupla (nabla_b, nabla_w) que representa o gradiente da função
    de custo.
    """
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.pesos]

    #feedforward
    ativacao = x #entradas de treinamento que passam pela função sigmoid
    ativacoes = [x] #armazena cada ativacao (todas são sigmoid)
    zs = [] #armazena cada ativacao

    for b, w in zip(self.biases, self.pesos):
        z = np.dot(w, ativacao)+b
        zs.append(z)
        ativacao = sigmoid(z)
        ativacoes.append(ativacao)

    #backward
    delta = self.custo_derivada(ativacoes[-1], y) * sigmoid_derivada(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, ativacoes[-2].transpose())

    for l in range(2, self.num_camadas):
        z = zs[-l] #pega o último valor armazenado
        ds = sigmoid_derivada(z)
        delta = np.dot(self.pesos[-l+1].transpose(), delta) * ds
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, ativacoes[-l-1].transpose())

    return (nabla_b, nabla_w)

def custo_derivada(self, saida_ativacoes, y):
    """
    Retorna o vetor de derivadas parciais d(custo)/d(ativacao).
    """
```

```
return (saida_ativacoes - y) #é mais um erro entre saída e valor esperado

def sigmoid_derivada(z):
    return sigmoid(z) * (1 - sigmoid(z))
```

Na seção 3.5, que se tratava de *backpropagation*, foi apresentada uma intuição sobre seu funcionamento. Acima está o registro do código que realiza os cálculos. Ele retorna o tamanho do passo que deve ser dado, posteriormente utilizado pela função que determina o gradiente descendente.

Um adendo: todas as funções, exceto por `sigmoid(z)` e `sigmoid_derivada(z)` estão dentro da classe `Network`.

4.2. Carregamento do conjunto de dados

Agora que temos um algoritmo que implementa a rede neural, precisamos dos dados para treiná-la. Conforme já mencionado anteriormente, utilizaremos a base de dados MNIST. Os dados podem ser baixados em formato adequado e conveniente para o uso em:

- [GitHub: Michael Nielsen²](#)
- [IRO Université de Montréal: MNIST³](#)

No arquivo baixado através do primeiro *link*, além do *dataset*, há também o código original que deu base ao que foi apresentado na seção anterior. Em <http://yann.lecun.com/exdb/mnist/> é possível obter mais informações sobre os *datasets*.

Porém esses arquivos também já se encontram na pasta do *Google Drive* fornecida no início da seção 4, com as alterações já feitas para que tudo funcione no *Google Colab* e com a versão 3.8.10 do *python*.

O arquivo “`mnist_loader.py`” no *Google Colab* tem comentários que explicam seu funcionamento. Contém funções que carregam os dados nas variáveis que serão utilizadas no código apresentado na seção 4.1.

Vamos usar `load_data_wrapper()` para carregar os dados, da seguinte forma:

```
Python
dados_treinamento, dados_teste = load_data_wrapper()
```

Serão sorteados 10 dados do conjunto de testes para serem exibidos, com suas imagens e respectivos rótulos.

² <https://github.com/mnielsen/neural-networks-and-deep-learning/archive/master.zip>

³ <http://www.iro.umontreal.ca/~lisa/deep/data/mnist/>

```
dados_treinamento, dados_teste = load_data_wrapper()

Mounted at /content/drive
quantidade de imagens de treinamento: 60000
quantidade de imagens de teste: 10000

- - - - - 10 amostras de rótulos e suas respectivas imagens - - - - -

RÓTULOS:
[6, 7, 0, 5, 7, 3, 5, 3, 6, 2]

IMAGENS:
```

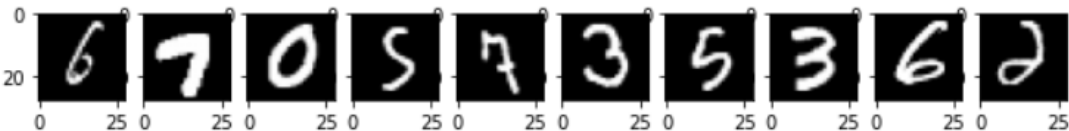


Figura 17: Utilização da função para carregar os dados. São sorteados 10 dados para serem exibidos, com suas imagens e respectivos rótulos.

4.3. Treinamento da rede neural e resultados

Agora que já temos os dados, é hora de treinar a rede! Invariavelmente, serão 784 neurônios de entrada e 10 de saída. Vamos começar com uma camada oculta de 30 neurônios.

Em seguida, vamos treinar a rede neural com 10 *epochs*, *mini-batches* de tamanho 100 e $\eta = 30$.

```
net = Network([784, 30, 10])
net.SGD(dados_treinamento, 10, 100, 30, dados_teste=dados_teste)
```

```
Epoch 0: 2485 / 10000
Epoch 1: 3140 / 10000
Epoch 2: 3863 / 10000
Epoch 3: 4525 / 10000
Epoch 4: 4843 / 10000
Epoch 5: 5143 / 10000
Epoch 6: 5268 / 10000
Epoch 7: 5417 / 10000
Epoch 8: 5468 / 10000
Epoch 9: 5595 / 10000
```

Figura 18: Criação da rede com 784 neurônios de entrada, 10 neurônios de saída e uma camada oculta de 30 neurônios. Com *mini-batches* de tamanho 100 e $\eta = 30$, a acurácia da rede foi de 55.95% após as 10 *epochs*.

Os valores impressos a cada *epoch* indicam o número de imagens de teste que foram classificadas corretamente (são 10 mil imagens no total). Abaixo temos outra tentativa:


```
net = Network([784, 30, 10])
net.SGD(dados_treinamento, 40, 60, 10, dados_teste=dados_teste)
```

```
Epoch 0: 2968 / 10000
Epoch 1: 3579 / 10000
Epoch 2: 3995 / 10000
Epoch 3: 4763 / 10000
Epoch 4: 5673 / 10000
Epoch 5: 6280 / 10000
Epoch 6: 6627 / 10000
Epoch 7: 6921 / 10000
Epoch 8: 7234 / 10000
Epoch 9: 7413 / 10000
Epoch 10: 7587 / 10000
```

...

```
Epoch 30: 8718 / 10000
Epoch 31: 8715 / 10000
Epoch 32: 8730 / 10000
Epoch 33: 8764 / 10000
Epoch 34: 8786 / 10000
Epoch 35: 8811 / 10000
Epoch 36: 8813 / 10000
Epoch 37: 8794 / 10000
Epoch 38: 8820 / 10000
Epoch 39: 8815 / 10000
```

Figura 19: Treinamento da mesma rede neural, mas agora com 40 *epochs*, *mini-batches* de tamanho 60 e $\eta = 10$. A acurácia foi de 88.15%.

Com uma camada oculta de 100 neurônios (mantendo *epochs*, *mini-batches* e η do caso anterior), obteve-se acurácia de 62.58%. Aumentar a quantidade de neurônios piorou o desempenho para este caso. A rede não generalizou o modelo tão bem quanto na configuração anterior.

Com uma camada oculta de 30 neurônios, 40 *epochs*, *mini-batches* de tamanho 10 e $\eta = 3$, obteve-se acurácia de 93.54%.

É importante lembrar que pode haver flutuações na porcentagem de acerto cada vez que se treina uma mesma rede neural, afinal os pontos de partida para pesos e *biases* são gerados aleatoriamente. Além disso, a escolha do número de neurônios, da taxa de aprendizagem, da quantidade de *epochs*, etc, é uma tarefa desafiadora. Pode ser que os valores de inicialização dos pesos e *biases* sejam inadequados, a ponto de dificultar consideravelmente o aprendizado da rede. Por isso, faz-se essencial o desenvolvimento de técnicas heurísticas para tratar desses problemas e obter melhores resultados.

5. Aprimoramento de performance

Na seção anterior, construímos uma rede neural que reconhece dígitos numéricos. Podemos gastar muito tempo fazendo diversas tentativas a fim de encontrar bons

hiperparâmetros (quantidade de neurônios na camada oculta, taxa de aprendizado, número de *epochs*, etc) e ainda assim não chegar a um bom resultado.

E como avaliamos o resultado? O modo mais simples de avaliá-lo é comparar a taxa de acerto com um palpite aleatório: seria de aproximadamente 10% caso simplesmente sorteássemos um valor para a imagem.

Como descobrir um bom ponto de partida para os parâmetros da rede neural? Como encontrar bons hiperparâmetros? Existe uma opção melhor para a função de custo ou para a função de ativação? Discutiremos a seguir.

5.1. Entropia cruzada como função de custo

Uma opção para substituir o erro quadrático médio (MSE) é a entropia cruzada, ou *cross entropy*, definida a seguir:

$$C(w, b) = -\frac{1}{n} \sum [y \cdot \ln(a) + (1 - y) \cdot \ln(1 - a)]$$

onde n é a quantidade de dados de treinamento, $a = \sigma(w \cdot x + b)$, x são as entradas de treinamento e y é a saída correspondente desejada (“gabarito”).

A entropia cruzada é similar ao MSE nos seguintes aspectos:

- Seu valor C é sempre maior ou igual a zero;
- Se o valor de saída encontrado para determinada entrada for próximo ao valor esperado, C será quase nulo.

A vantagem de se usar a entropia cruzada como função de custo está em suas derivadas parciais em relação a w e b . Relembremos o formato de uma função *sigmoid*:

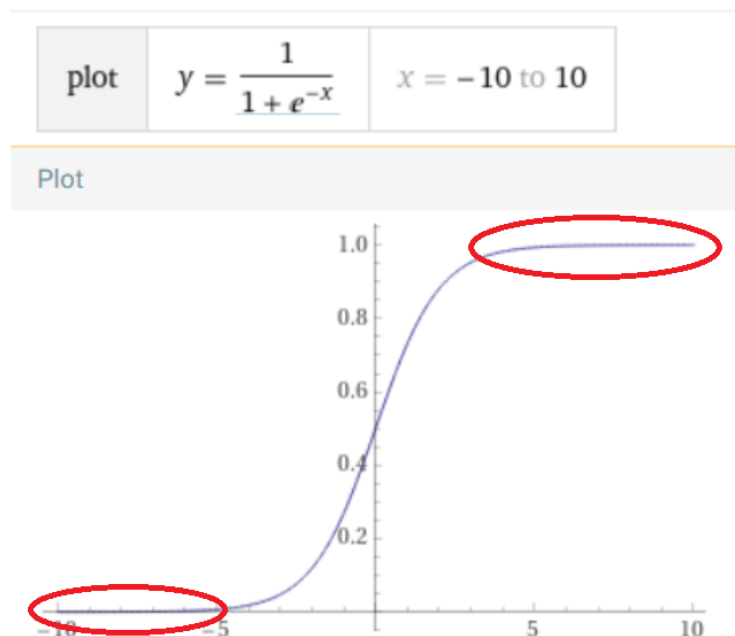


Figura 20: Função *sigmoid*.

A fim de adquirirmos uma intuição, para um único exemplo de treinamento, consideremos as seguintes derivadas parciais de cada função de custo:

MSE:

- $\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x$
- $\frac{\partial C}{\partial b} = (a - y)\sigma'(z)$

Entropia cruzada:

- $\frac{\partial C}{\partial w} = x(\sigma(z) - y)$
- $\frac{\partial C}{\partial b} = \sigma(z) - y$

Podemos ver que, para MSE, a derivada $\sigma'(z)$ se torna muito pequena quando a saída do neurônio está em alguma região praticamente plana da curva (regiões destacadas na Fig. 20). Isso torna o aprendizado lento se nos encontrarmos nesses lugares. Por exemplo, se a resposta esperada é 0 e no momento foi calculado o valor 1 para a saída, os passos serão pequenos enquanto caminharmos pela parte plana do *sigmoid*. Quando atingirmos a região íngreme da curva, as derivadas serão grandes, e consequentemente o aprendizado será mais rápido enquanto o algoritmo passa por esses valores.

Já para a entropia cruzada, dependemos da diferença entre o valor encontrado e o valor esperado: se a saída encontrada estiver muito longe do esperado, o aprendizado será rápido. Se no momento nos encontramos na saída de valor 1 quando o esperado é 0, o próximo passo será grande, pois é grande o erro entre o que foi calculado e o esperado.

Como a inicialização dos parâmetros é feita aleatoriamente, é provável que várias saídas terão valores bem errados quando comparadas com o esperado. Por isso, o uso da entropia cruzada como função de custo geralmente se mostra preferível.

5.2. Função de ativação

Na seção 2.2, apresentamos o *sigmoid* como função de ativação. Um dos problemas dessa escolha foi visto no tópico anterior: a função satura a partir de certos valores - tanto para 0 quanto para 1 -, o que potencialmente torna o aprendizado lento. Mostraremos a seguir uma alternativa: a unidade linear retificada, ou ReLU (*rectified linear unit*).

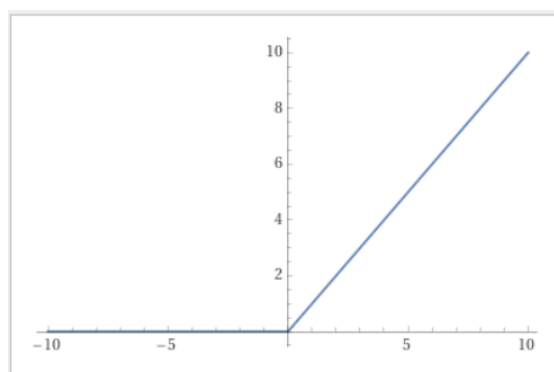


Figura 21: Formato da ReLU.

A função ReLU é definida:

$$saída = \max(0, w \cdot x + b)$$

Assim como o *sigmoid*, ReLU também é capaz de computar qualquer função. Uma diferença fundamental, porém, é que não há saturação enquanto $z = w \cdot x + b$ for positivo - portanto não há desaceleração no aprendizado (pelo contrário, este costuma ser mais rápido!).

Por outro lado, se a entrada x resultar em valor negativo na saída, o neurônio simplesmente para de aprender, pois o gradiente também se anula. Mas há vantagens para isso também: a esparsidade (presença de zeros), quando não ocorre para a maioria dos neurônios, aparenta ser benéfica para a convergência de resultados. Além disso, o custo computacional para executar a função ReLU é mais baixo.

Novamente, vale sempre lembrar que, em redes neurais, há bastante heurística envolvida. Muitas vezes o embasamento teórico matemático ainda não foi desenvolvido, então não se sabe exatamente por que uma escolha é mais eficiente do que a outra.

5.3. Inicialização de parâmetros

A inicialização dos pesos e *biases* no código apresentado na seção 4.1 foi feita da seguinte maneira:

```
Python
self.biases = [np.random.randn(y,1) for y in sizes[1:]]
self.pesos = [np.random.randn(y,x) for x,y in zip(sizes[:-1], sizes[1:])]
```

A função retorna um valor aleatório com distribuição gaussiana de média 0 e variância 1. Vamos ilustrar um problema que pode acontecer com um exemplo, para facilitar a visualização.

Imagine o primeiro neurônio de uma camada oculta, que recebeu o valor 1 como saída de 500 neurônios de entrada, sendo que existem 1000 entradas. A soma dos pesos do neurônio será $z = \sum_i^n w_i \cdot x_i + b$. Sabemos que w e b foram inicializados aleatoriamente com distribuição gaussiana de média 0 e variância 1. Então z também será uma distribuição gaussiana de média 0, porém de variância 501 (desvio padrão = $\sqrt{501} = 22.38$), pois são 500 valores de pesos e um valor de *bias*.

Podemos ver, na Fig. 22, que a distribuição não tem um pico concentrado. Isso significa que z provavelmente vai ser muito grande em módulo, logo o valor de saída do neurônio (cujas função de ativação é o *sigmoid*) estará próxima de 0 ou de 1 - ou seja, em alguma das regiões de saturação, indicadas na Fig. 20. Conforme vimos na seção anterior, isso desacelera o aprendizado. Podemos estender o problema que esse neurônio enfrenta para todos os outros na camada oculta.

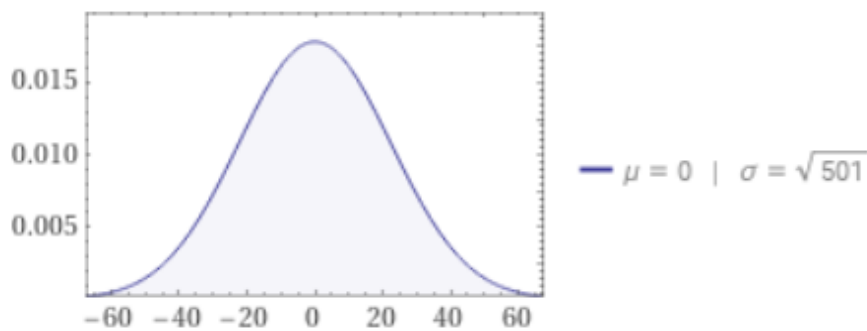


Figura 22: Distribuição gaussiana de média 0 e variância 501 (desvio padrão 22.38).

Uma forma de contornar é escalar o desvio padrão dos pesos por um fator de $\frac{1}{\sqrt{n_{in}}}$, onde n_{in} é a quantidade de pesos de entrada de um neurônio. Isso vai achatar a distribuição, deixando-a mais concentrada na média, o que conseqüentemente torna mais provável que o valor caia na região não saturada do *sigmoid*. Uma explicação mais detalhada pode ser encontrada [aqui](#)⁴.

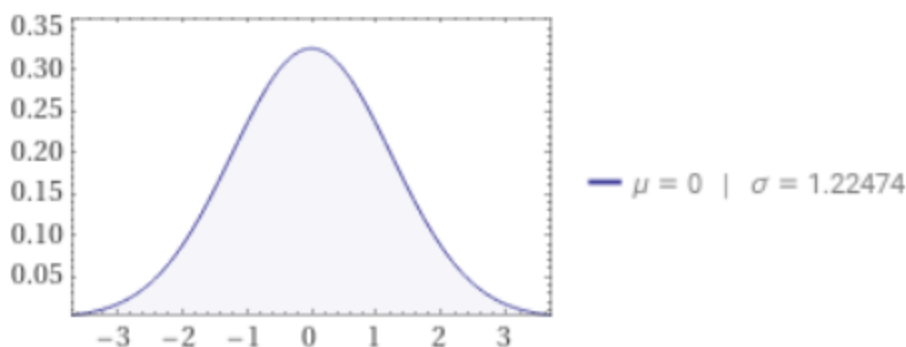


Figura 23: Distribuição gaussiana após aplicar o novo desvio padrão para os pesos. Note como a base é bem menor do que a base da Fig. 22 e o valor da probabilidade está acima de 0.3 (contra menos de 0.02 no caso anterior).

A biblioteca *Keras*, que será apresentada na seção 5.6, tem diversas opções para inicialização dos pesos.

5.4. Escolha de hiperparâmetros

Nesta seção veremos algumas heurísticas que podem ser utilizadas para escolher os hiperparâmetros da rede neural.

5.4.1. Taxa de aprendizado

Na seção 3.3 vimos como a taxa de aprendizado funciona. Não queremos que seja grande a ponto de ultrapassar o ponto de mínimo da função de custo. Porém, a escolha pode ser tão ruim que os resultados pioram logo após a primeira *epoch*. Pode-se iniciar com

⁴ <https://compsci682-fa19.github.io/notes/neural-networks-2/#init>

$\eta = 0.01$. Se os resultados forem ruins (se o custo aumenta ou oscila muito), então é melhor tentar com valores menores de η (ou seja, $\eta = 0.001, 0.0001\dots$), até encontrar um valor para o qual a função custo diminua nas primeiras *epochs*. Se, ao contrário, para $\eta = 0.01$, a função custo já diminuir nas primeiras *epochs*, então pode-se aumentar o valor de η sucessivamente ($\eta = 0.1, 1.0\dots$) até que ele passe a oscilar ou aumentar. Encontrar esse valor nos oferece uma estimativa do limite para η .

5.4.2. Parâmetro de regularização λ

Uma forma de ajustar os hiperparâmetros é utilizando um conjunto de dados para validação. No caso do MNIST, por exemplo, podemos separar os conjuntos em 50 mil imagens para treinamento, 10 mil imagens para validação e 10 mil imagens para teste.

Então, para estabelecer um valor para λ , podemos seguir a mesma sequência apresentada em 5.4.1. Após obter um valor para η , começamos com um valor de λ e depois o ajustamos (aumentando ou diminuindo em potências de 10) até obter a melhor performance. Em seguida, pode-se otimizar η novamente, com um valor de λ fixo.

5.4.3. Número de camadas ocultas e de neurônios por camada oculta

Não há uma regra para definir a quantidade de camadas ocultas em uma rede neural. Sabe-se que camadas iniciais capturam informações de baixo nível, enquanto as camadas mais profundas conseguem modelar padrões mais complexos. Por exemplo, as primeiras camadas podem reconhecer curvas em diferentes direções e formatos, enquanto as intermediárias são capazes de identificar formas geométricas e as últimas camadas já modelam objetos ou rostos.

Para conjuntos de dados complexos, a presença de mais camadas ocultas ajuda a rede a convergir mais rapidamente, além de aumentar sua capacidade de generalização. Uma possível abordagem é aumentar gradualmente a quantidade de camadas ocultas e analisar os resultados.

A mesma ideia se aplica ao número de neurônios. Dificilmente encontraremos a quantidade ideal de camadas e neurônios, então a forma mais simples de enfrentar o problema é escolher um modelo com mais camadas e neurônios do que o necessário e evitar o *overfitting* com as técnicas apresentadas na seção 3.7.

5.5. Redes neurais convolucionais

Vamos apresentar agora as redes neurais convolucionais, cujo nome deriva de uma operação matemática chamada convolução. Esse tipo de rede neural é profunda, diferentemente do que foi visto até agora. O que caracteriza uma rede neural profunda (*deep neural network*) é o fato haver duas ou mais camadas ocultas.

Em redes neurais profundas, é como se as camadas iniciais respondessem a perguntas bastante simples e diretas a respeito da imagem, e as camadas seguintes pudessem abstrair conceitos mais sutis e complexos.

A definição formal de convolução é:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$$

Vamos convoluir o núcleo (*kernel*) $f(t)$ com o sinal de entrada $g(t)$. A convolução é uma operação linear utilizada no lugar da multiplicação de matrizes (método usado na seção 4), pois o processo se torna menos custoso computacionalmente devido às propriedades da Transformada Rápida de Fourier (*Fast Fourier Transform*, FFT) [3]. Quando se trata de redes profundas, essa propriedade é uma enorme vantagem.

5.6. Biblioteca: *Keras*

A biblioteca *Keras* é voltada para aplicações de redes neurais e apresenta diversas ferramentas para criar os modelos, definir as funções de custo e de ativação, etc.

Na aba “Reconstruir o que foi feito até agora, mas com a biblioteca” do *Google Colab*, há o código comentado de uma rede neural semelhante à que foi construída ao longo da seção 4, com 30 neurônios na camada oculta. Foi treinada com 40 *epochs*, *mini-batches* de tamanho 10 e $\eta = 3$. Anteriormente, obtivemos uma acurácia de 93.54%.

Model: "modelo"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 784)]	0
dense (Dense)	(None, 30)	23550
dense_1 (Dense)	(None, 10)	310
Total params: 23,860		
Trainable params: 23,860		
Non-trainable params: 0		

Figura 24: Modelo de rede neural com 784 neurônios de entrada, 30 neurônios na camada oculta e 10 neurônios de saída.

O modelo construído com uso das ferramentas da biblioteca atingiu uma acurácia de 94.88%. Está próximo do que foi encontrado anteriormente. Em todas as vezes, obteve resultados levemente melhores do que o modelo apresentado na seção 4, por volta de 1% a mais. Para comparar resultados, também foram feitos testes com a seguinte configuração: 40 *epochs*, *mini-batches* de tamanho 60 e $\eta = 10$ para modelos com camada oculta de 30 e de 100 neurônios, assim como na seção 4. Porém, os resultados foram bem diferentes: para 30 neurônios, obtivemos 96.78% (contra 88.15%); para 100 neurônios, obtivemos 96.20% (contra 62.58%). Uma possível e provável explicação para essas diferenças é a presença de otimizadores e outros argumentos cujos valores *default* diferem das configurações da rede construída anteriormente.

Agora vamos explorar as ferramentas para construir redes neurais convolucionais. O código utilizado nesta seção foi baseado no [código de exemplo](#)⁵ encontrado na própria página do *Keras*. Na documentação do código há todos os *links* para as funções utilizadas da biblioteca. A rede neural convolucional criada é mostrada na Fig. 25.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 32)	9248
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 32)	0
flatten (Flatten)	(None, 1568)	0
dropout (Dropout)	(None, 1568)	0
dense (Dense)	(None, 10)	15690
Total params: 25,258		
Trainable params: 25,258		
Non-trainable params: 0		

Figura 25: Rede neural convolucional criada a partir das ferramentas do *Keras*.

A rede neural da Fig. 25 foi criada a partir do código abaixo:

```
Python
model = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
        layers.Conv2D(filters=32, kernel_size=(3, 3),
            activation="relu", padding='same'),
        layers.MaxPooling2D(pool_size=(2, 2), strides=(2,2)),
```

⁵ https://keras.io/examples/vision/mnist_convnet/


```

        layers.Conv2D(filters=32, kernel_size=(3, 3),
activation="relu", padding='same'),
        layers.MaxPooling2D(pool_size=(2, 2), strides=(2,2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

```

Note que a entrada deixou de ser um vetor de tamanho 784, mas sim passou a ser uma matriz de tamanho 28x28. Isso será necessário para realizarmos a convolução espacial em duas dimensões. Uma representação do que acontece está na Fig. 26, imagem retirada do livro “*Neural Networks and Deep Learning*”, de Michael Nielsen.

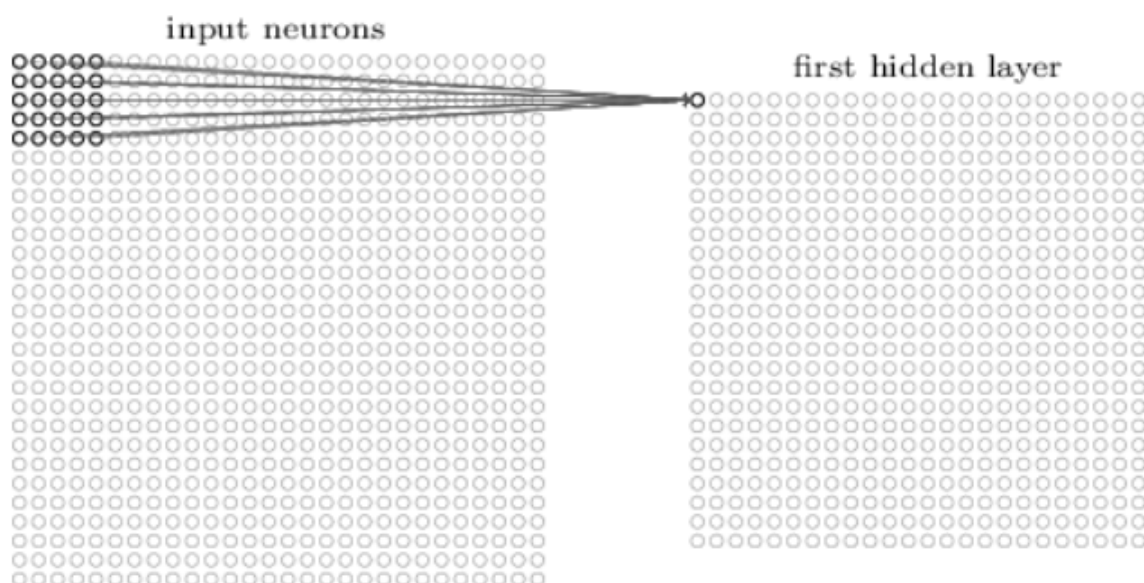


Figura 26: Uma “janela” de pixels serve de entrada para um neurônio na camada oculta.

É como se cada neurônio da camada oculta aprendesse as características de uma dada região. O argumento `kernel_size` contém o tamanho dessa “janela”, enquanto o argumento `filters` diz o número de filtros de saída na convolução.

Depois, deslizamos a “janela” para alguma outra região e a conectamos a outro neurônio da camada oculta. A Fig. 27, também retirada do livro de Michael Nielsen, representa a ideia.

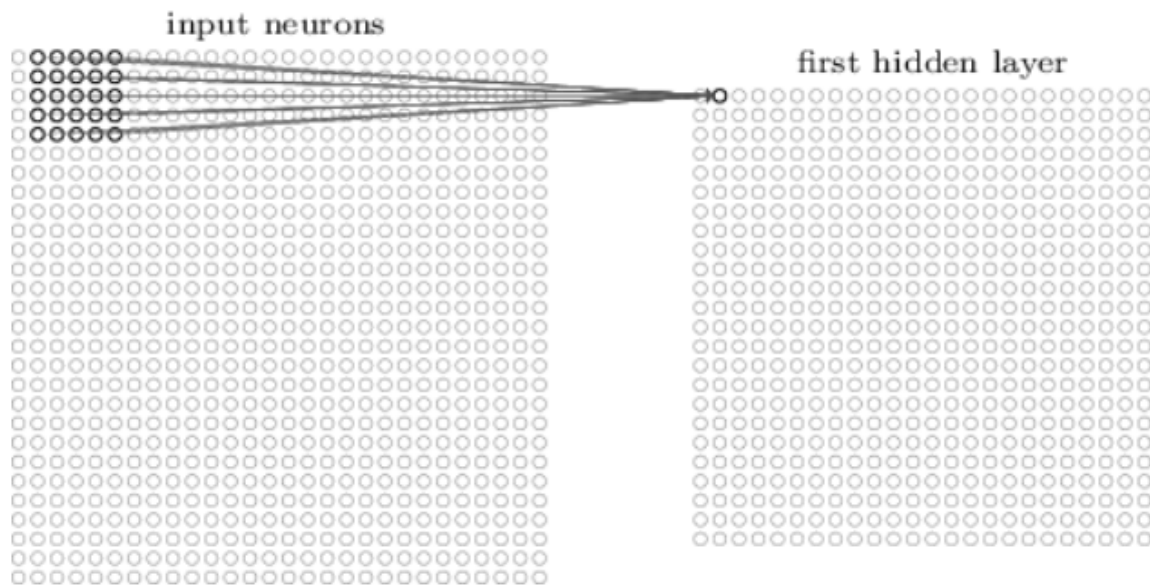


Figura 27: A “janela” se deslocou para outra região, agora associada a outro neurônio da camada oculta.

Vemos que a próxima camada da rede se chama “MaxPooling2D”. A ideia por trás do *max pooling* é reduzir a quantidade de dados ao escolher o valor máximo dentro da “janela” de tamanho 2x2 (argumento `pool_size`).

Em seguida temos mais uma camada “Conv2D” e outra “MaxPooling2D”. Depois vem uma camada que se chama “Flatten”, logo antes da última camada “Dense” (desconsiderando o “Dropout” - que serve simplesmente para aplicar o método explicado na seção 3.7.3). “Flatten” transforma o que chega nela em um vetor (de dimensão 1) para passar os dados adiante, pois é assim que eles devem chegar a uma camada “Dense” [4].

A acurácia da rede neural convolucional (*convolutional neural network*, CNN) apresentada foi de 98.93%. Para um conjunto de dados relativamente simples como o MNIST - o “Hello World” da classificação de imagens -, uma rede profunda pode parecer desnecessária, mas, quando trabalhamos com conjuntos de dados grandes e complexos, CNNs são poderosas ferramentas, bem mais eficientes do que as redes neurais rasas [1].

6. Conclusão

Neste trabalho apresentamos as bases e fundamentos para a construção de uma rede neural, atacando um problema famoso de reconhecimento de imagem: a classificação de dígitos numéricos escritos à mão. Ao acompanhar passo a passo a implementação realizada por Michael Nielsen, pudemos verificar a aplicação de cada conceito exposto e observar uma performance com taxa de acerto maior que 90%.

Em seguida, discutimos algumas limitações da abordagem adotada e apresentamos possíveis substituições. Devemos nos lembrar de que não há método intrinsecamente melhor; a escolha depende do problema a ser encarado. Por isso também exploramos brevemente as redes neurais convolucionais, com auxílio da biblioteca *Keras*, que oferece inúmeras

ferramentas para *deep learning*. O desempenho da CNN foi ainda mais impressionante, ultrapassando os 98% de acurácia.

O presente trabalho é apenas uma introdução às redes neurais artificiais, que são o alicerce do *deep learning* e ideais para realizar tarefas de alta complexidade, como reconhecimento de imagens (há aplicações na medicina, agricultura, tráfego, segurança, etc), reconhecimento de fala (há aplicativos que transformam fala em texto escrito, em comandos para dispositivos, etc), processamento de linguagem natural (tradução, assistentes inteligentes, filtros de e-mail, etc), e muito mais.

7. Bibliografia

- [1] GÉRON, Aurélien. *Mãos à Obra: Aprendizado de Máquina com Scikit-Learn & TensorFlow*. AltaBooks. Copyright 2019 de Aurélien Géron, 978-85-508-0381-4.
- [2] NIELSEN, Michael. [Neural Networks and Deep Learning](http://neuralnetworksanddeeplearning.com/)⁶.
- [3] BetterExplained. [Intuitive Guide to Convolution](https://betterexplained.com/articles/intuitive-convolution/)⁷.
- [4] Towards Data Science. [The Most Intuitive and Easiest Guide for Convolutional Neural Network](https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480)⁸.
- [5] Data Science Academy. [Deep Learning Book](https://www.deeplearningbook.com.br/)⁹.

⁶ <http://neuralnetworksanddeeplearning.com/>

⁷ <https://betterexplained.com/articles/intuitive-convolution/>

⁸ <https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480>

⁹ <https://www.deeplearningbook.com.br/>