



Redes Neurais e aprendizado profundo

Apresentação de conceitos e aplicações de redes neurais e aprendizado profundo, com abordagem das arquiteturas das redes, e de problemas relacionados ao processo de aprendizado; de aplicações práticas de redes neurais de convolução (CNNs) e recorrentes (RNNs), utilizando a linguagem de programação Python.

Prof. Sérgio Assunção Monteiro

Propósito

Compreender os conceitos de aprendizado de máquina em redes neurais profundas e sua aplicação em problemas práticos usando a linguagem Python é importante para a formação do cientista de dados.

Preparação

Para executar nossos exemplos sobre o TensorFlow, você vai precisar criar uma conta no Google Colab.

Objetivos

- Reconhecer as etapas necessárias para fazer o treinamento de redes neurais profundas.
- Reconhecer os aspectos relacionados ao aprendizado por reforço.
- Identificar as redes neurais de convolução (CNN) e recorrentes (RNN).
- Esquematizar simulação com TensorFlow e AWS SageMaker.

Introdução

Atualmente, é muito comum ouvirmos falar sobre inteligência artificial (IA) e, em especial, sobre aprendizado de máquina. De fato, os recursos da IA estão cada vez mais próximos da nossa realidade. Alguns exemplos práticos são:

- Quando fazemos uma pesquisa na internet e um provedor nos dá sugestões baseadas na análise do nosso perfil.
- Em aplicações de reconhecimento de imagem e voz.
- Na análise de textos para fazer classificação etc.

Certamente, existem muitos outros exemplos práticos que poderíamos apresentar. O fato é que, desde que o termo **aprendizado de máquina** foi usado pela primeira vez, em 1959, por Arthur Samuel, um cientista da IBM pioneiro no campo de jogos de computador e inteligência artificial, houve intensa evolução ao longo dos anos, desde os aspectos teóricos até os práticos.

O avanço tecnológico e dos processos de desenvolvimento e a implantação de software tornou o uso dos algoritmos de aprendizado de máquina não apenas possível, mas necessário para interagir com um mundo complexo, caracterizado pela diversidade de aplicações, redução de custos e velocidade na resolução de problemas.

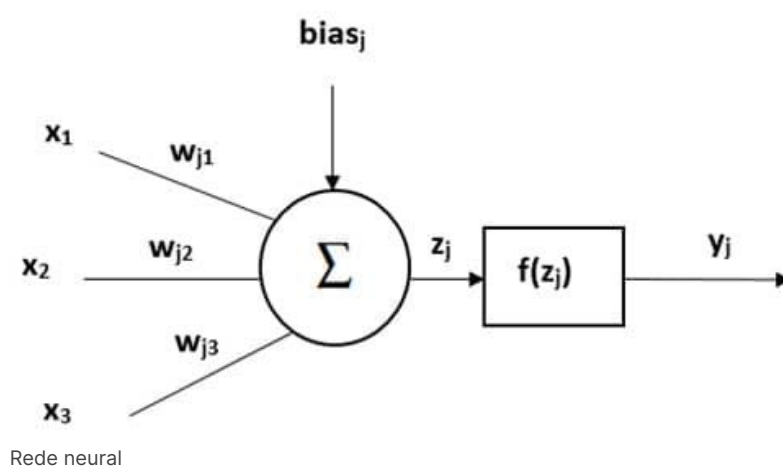
Ao longo do texto, apresentaremos conceitos relacionados aos modelos de aprendizado profundo, em especial os relacionados às arquiteturas cujas características tornaram possíveis aplicações de classificação de imagens e de processamento de linguagem natural, por exemplo. Na parte final, veremos aplicações práticas com o uso de bibliotecas que facilitam o desenvolvimento de algoritmos de aprendizado de máquina com o uso da linguagem de programação Python.

Problemas dos gradientes

Uma rede neural é formada por neurônios interconectados por meio de ligações chamadas de sinapses. Na imagem a seguir, podemos ver os elementos principais de uma rede neural.

Sinapses

As sinapses são junções entre a terminação de um neurônio e a membrana de outro neurônio. São elas que fazem a conexão entre células vizinhas, dando continuidade à propagação do impulso nervoso por toda a rede neuronal.



Nas sinapses, estão associadas entradas - representadas na imagem pelas variáveis x_1, x_2 e x_3 - e pesos - representados pelas variáveis w_{j1}, w_{j2} e w_{j3} - combinados por meio de uma função que faz a multiplicação das entradas com os respectivos pesos e soma com o **bias**, obtendo, assim, z_j conforme podemos ver na equação:

$$z_j = \sum_{i=1}^N x_i w_{ji} + \text{bias}_j$$

A variável z_j também é chamada de função de propagação. O índice "j" refere-se ao "j-ésimo" neurônio da rede. A partir da variável z_j , calculamos a chamada função de ativação, representada por $f(z_j)$ na imagem. Essa função de ativação vai produzir y_j .



Comentário

Bias é uma constante que ajuda a Rede Neural, para que ela se ajuste melhor aos dados fornecidos. Ou seja, é uma constante que dá flexibilidade para um melhor desempenho da Rede Neural.

Funções de ativação

Existem alguns tipos de funções de ativação. A seguir, apresentamos as mais comuns:

Função linear

A função linear e sua derivada são dadas, respectivamente, por:

$$f(z) = \alpha z \quad f'(z) = \alpha$$

Na imagem, apresentamos o gráfico da função linear.

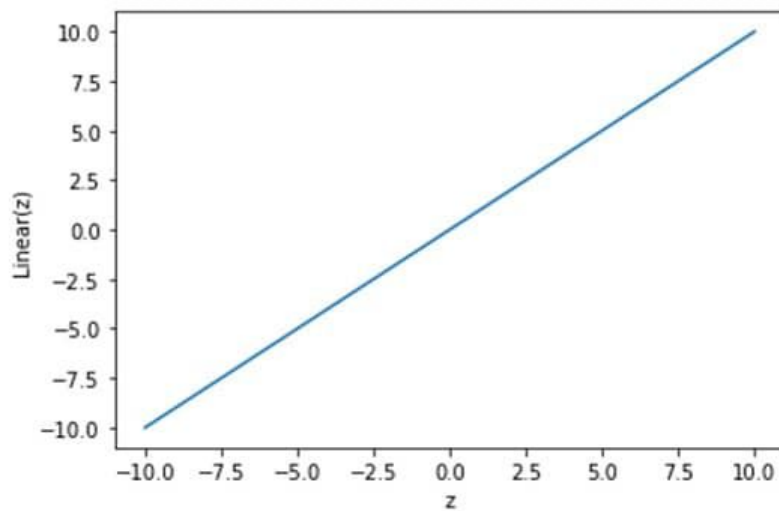


Gráfico da função linear

Função sigmoide

A função sigmoide ou logística e sua derivada são dadas, respectivamente, por:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$
$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Na imagem, apresentamos o gráfico da função sigmoide.

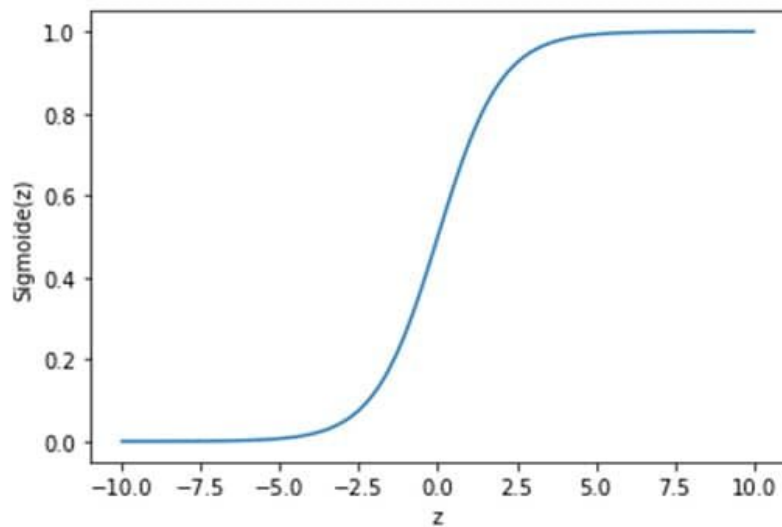


Gráfico da função sigmoide.

Função Tanh (tangente hiperbólica)

A função tangente hiperbólica e sua derivada são dadas, respectivamente, por:

$$\begin{aligned}\tanh(z) &= 2\sigma(2z) - 1 \\ \tanh'(z) &= 1 - \tanh^2(z)\end{aligned}$$

Na imagem, apresentamos o gráfico da função tangente hiperbólica.

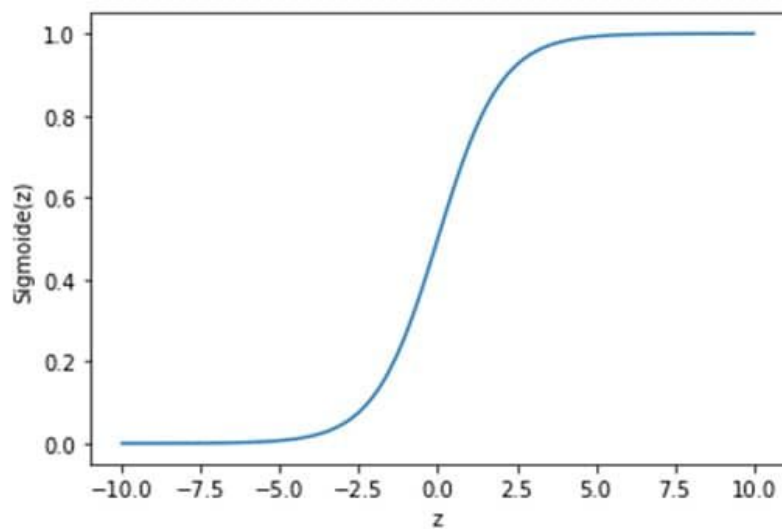


Gráfico da função tangente hiperbólica.

Função ReLU (unidade linear retificada)

A função ReLU e sua derivada são dadas, respectivamente, por:

$$\text{ReLU}(s) = \max\{0, s\} \quad \text{ReLU}'(s) = \begin{cases} 1 & \text{se } s \geq 0 \\ 0 & \text{caso contrário} \end{cases}$$

Na imagem, apresentamos o gráfico da função ReLU.

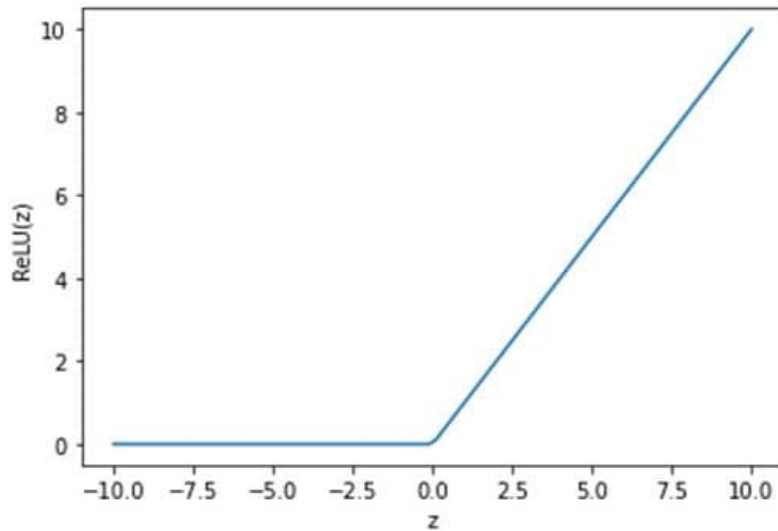


Gráfico da função ReLU.

A ideia central de uma rede neural é mapear uma entrada por meio de uma função de ativação. Dessa forma, a rede passa por um processo de treinamento que tem como objetivo determinar um conjunto de parâmetros – os **pesos sinápticos** – que produzem as melhores respostas.

Isso é obtido com a minimização da diferença entre o valor esperado e a saída do modelo. Por exemplo, vamos representar por " \hat{y} " o valor que esperamos que a rede neural obtenha para classificar determinada entrada e vamos representar por " y " a variável que a rede neural vai obter. A ideia, então, é minimizar o erro entre o valor obtido e o valor real. Essa função é dada por:

$$\text{Erro}(y) = \sum_{j=1}^N \frac{(y_j - \hat{y}_j)^2}{2}$$

Onde N representa a quantidade de unidades de saída.

Um dos algoritmos mais conhecidos para treinamento de redes neurais é o backpropagation, ou **algoritmo de retropropagação**. Esse algoritmo utiliza o método de gradiente de descida, que, por definição, compreende duas etapas:

- Calcular os gradientes da função de erro.
- Atualizar os parâmetros existentes em relação aos gradientes.

As etapas 1 e 2 são repetidas até que o algoritmo obtenha o valor mínimo da função de erro.

Com um pouco mais de detalhes, o algoritmo de retropropagação pelo método de gradiente descendente pode ser formalizado como:

Entrada de dados

w , bias, x , \hat{y} , η

Repita

Calcule y_j

Calcule $\text{Erro}(y_j)$

Calcule os gradientes Δw_{ji}

Faça $w_{ji} = w_{ji} - \eta \Delta w_{ji}$

Faça $w_{ji} = w_{ji} - \eta \Delta w_{ji}$

Até que $\text{Erro}(y_j) < \text{TOLERÂNCIA}$

Onde η é chamada de taxa de aprendizado e Δw_{ji} é o gradiente da função de erro em relação ao peso w_{ji} .

O gradiente de uma função corresponde à sua inclinação. O processo de treinamento da rede neural com o uso de gradiente de descida reduz a função erro, aumentando, assim, o acerto da rede neural.

Infelizmente, existem alguns problemas quando aplicamos esse método. São eles (BENGIO; SIMARD; FRASCONI, 1994):

Dissipação do gradiente (vanishing gradient)

Trata-se de um cenário em que o modelo não aprende nada. Isso ocorre quando o gradiente se torna muito pequeno; implica que os pesos não mudarão para minimizar o erro.

Explosão do gradiente

Trata-se exatamente do oposto do desaparecimento do gradiente. O gradiente cresce excessivamente e o modelo nunca converge.

Ponto de Sela (ponto minimax)

O algoritmo fica estacionado em um ponto da função de erro que não permite sua minimização.

Reutilização de camadas pré-treinadas

Na abordagem clássica de aprendizado de máquina, existem diferentes estratégias e técnicas de transferência de aprendizagem, que podem ser aplicadas com base no domínio, na tarefa em mãos e na disponibilidade de dados. Os métodos de aprendizagem por transferência podem ser categorizados das seguintes maneiras (PAN; YANG, 2010):

Aprendizagem por transferência indutiva

Nesse cenário, tanto o domínio da origem como o do destino são os mesmos, apesar de as tarefas de origem e destino serem diferentes umas das outras. O objetivo do método por indução é aprender com as instâncias de treinamento observadas e com um conjunto de suposições sobre a verdadeira distribuição dos casos de teste.

Aprendizagem por transferência não supervisionada

É semelhante à transferência indutiva com foco em tarefas não supervisionadas no domínio de destino. Os domínios de origem e destino são semelhantes, mas as tarefas são diferentes. Nesse cenário, os dados rotulados não estão disponíveis em nenhum dos domínios.

Aprendizagem por transferência transdutiva

Nesse cenário, existem semelhanças entre as tarefas de origem e de destino, mas os domínios correspondentes são diferentes. Nessa configuração, o domínio de origem tem muitos dados rotulados, enquanto o domínio de destino não tem nenhum. Isso pode ser classificado em subcategorias, referindo-se a configurações em que os espaços de características são diferentes ou a probabilidades marginais.

Uma evolução das redes neurais artificiais são as redes neurais profundas (DNN - Deep Neural Networks), que incluem várias camadas entre as camadas de entrada e saída. Elas são usadas para diversas aplicações em que a abordagem tradicional não funciona bem, como processamento de imagens (visão computacional) e processamento de linguagem natural. Então, além da abordagem tradicional, temos os modelos de transferência de aprendizado profundo (DTL - Deep Transfer Learning). A DTL oferece uma flexibilidade muito maior em extrair recursos de alto nível e transferi-los de um problema de origem para um problema de destino. Entre as vantagens da DTL, temos:

- Ajuda a resolver problemas complexos do mundo real com várias restrições.
- Resolve problemas com poucos dados rotulados disponíveis.
- Há transferência de conhecimento de um modelo para outro com base em domínios e tarefas.

Apesar das vantagens da transferência de aprendizado, existem questões importantes que devem ser tratadas como:

Transferência negativa

Há situações em que o processo de aprendizagem por transferência pode levar à queda no desempenho da rede neural. Isso ocorre em cenários em que a transferência de conhecimento da fonte para o destino não leva a nenhuma melhoria, mas causa uma queda no desempenho geral da tarefa de destino. Os motivos para a ocorrência desses problemas estão relacionados a casos em que a tarefa de origem não está suficientemente relacionada à tarefa de destino.

Limites de transferência

A qualidade da transferência deve ser quantificada para analisar qual foi o ganho real com esse processo.

Otimizadores velozes

Treinar uma rede neural profunda de forma eficiente não é uma tarefa simples. Existem diversos algoritmos que são aplicados para o processo de otimização dos pesos de uma rede neural de aprendizado profundo. Os mais comuns são:

- Gradiente de descida (gradient descent).
- Gradiente de descida estocástico.
- Gradiente acelerado de Nesterov.

Gradiente de descida (gradient descent)

O método calcula o gradiente da função de custo associada em relação a cada peso w_k e obtém o vetor gradiente $\Delta J(w_k)$ usando a equação:

$$w_{k+1} = w_k - \eta \Delta J(w_k)$$

Portanto, a velocidade do método depende exclusivamente do parâmetro de taxa de aprendizagem η .

Gradiente de descida estocástico

Em vez de calcular os gradientes para todos os seus exemplos de treinamento em cada passagem do método de gradiente de descida, ele usa apenas um subconjunto dos exemplos de treinamento de cada vez.

Gradiente acelerado de Nesterov

O método dá um "salto" com $w_k + \beta m_k$ na direção do gradiente acumulado anterior e depois mede o gradiente $\Delta J(w_k + \beta m_k)$, onde termina e faz uma correção. A ideia é que é melhor corrigir um erro depois de cometê-lo. O esquema do algoritmo é dado pelas equações:

$$m_{k+1} = \beta m_k - \eta \Delta J(w_k + \beta m_k)$$

$$w_{k+1} = w_k + m_{k+1}$$

Além desses métodos, também existem muitos outros, como AdaGrad, Momentum Optimization, RMSProp (root mean square prop), Adam e Nadam. O principal problema de usar um método de otimização inadequado é que o modelo leva muito tempo para convergir para um mínimo global ou fica preso em um mínimo local.

A escolha de um otimizador vai acelerar a velocidade de treinamento e ajudar a melhorar o desempenho do modelo.

Uso de regularização para evitar sobreajuste

Overfitting é um problema comum para todos os algoritmos de aprendizado de máquina. Ele ocorre quando o modelo se ajusta muito bem ao conjunto de treinamento, mas não tem um desempenho tão bom no conjunto de teste. Os modelos também podem apresentar o problema de ajuste insuficiente, que ocorre quando escolhemos um modelo muito simples que não funciona bem no conjunto de treinamento e teste. As formas mais comuns de reduzir o overfitting são por meio dos seguintes métodos:

Regularizações L1 e L2

Trata-se de adicionar um elemento extra à função de custo que penaliza o modelo por usar valores muito altos na matriz de peso. Desse modo, tentamos limitar sua flexibilidade, mas também incentivá-lo a construir soluções baseadas em vários recursos. Duas versões populares desse método são L1 — least absolute deviations (LAD) — e L2 — least square errors (LSE). As equações que descrevem essas regularizações são dadas por:

$$J_{L1}(W, b) = \frac{1}{m} \sum_{i=1}^m L(\bar{y}^{(i)}, y^{(i)}) + \lambda \|w\|_1, \|w\|_1 = \sum_{j=1}^n |w_j|$$

$$J_{L2}(W, b) = \frac{1}{m} \sum_{i=1}^m L(\bar{y}^{(i)}, y^{(i)}) + \lambda \|w\|_2, \|w\|_2 = \sum_{j=1}^n w_j^2$$

Na maioria dos casos, o uso de **L1** é preferível, pois reduz a zero os valores de peso menos significativos. Além disso, **L2** não tem um desempenho muito bom em conjuntos de dados com grande número de outliers (pontos que diferem muito dos demais dados).

Dropout

Cada unidade de nossa rede neural (exceto aquelas pertencentes à camada de saída) tem a probabilidade **p** de ser temporariamente ignorada nos cálculos. O hiperparâmetro **p** é chamado de taxa de dropout e, muitas vezes, seu valor-padrão é definido como 0,5. Então, em cada iteração, selecionamos aleatoriamente os neurônios que descartamos de acordo com a probabilidade atribuída. Como resultado, cada vez trabalhamos com uma rede neural menor.

Parada antecipada ou early stopping

Trata-se do algoritmo para quando o modelo de validação atinge um platô. A ideia é que o modelo tente seguir a função de custo durante a fase de treinamento, ajustando os parâmetros. Então, analisamos outro conjunto de dados como o conjunto de validação e, à medida que avançamos no treinamento, mantemos um registro da função de custo nos dados de validação; quando vemos que não há nenhuma melhoria no conjunto de validação, paramos, em vez de aplicarmos todas as iterações (épocas) do algoritmo de otimização.

Treinamento de redes neurais profundas

O especialista Sérgio Assunção Monteiro fala sobre os principais tópicos a respeito do treinamento de redes neurais profundas abordadas neste módulo.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

O processo de aprendizado profundo é caro. Isso significa dizer que é necessário ter à disposição recursos computacionais para fazer o processamento de diversas operações. Nesse sentido, os algoritmos de transferência de aprendizado se mostram uma técnica muito interessante e promissora. A respeito dos algoritmos de transferência de aprendizado, selecione a opção correta.

A

A aprendizagem por transferência pode ser indutiva, o que significa que os domínios de origem e de destino podem ser distintos.

B

Uma das formas de aprendizagem por transferência é a não supervisionada, em que os domínios de origem e destino são semelhantes, mas as tarefas são diferentes.

C

A aprendizagem por transferência pode ser transdutiva, o que significa que os domínios origem e destino são iguais.

D

Apesar de haver muitas pesquisas na aprendizagem por transferência, esses modelos ainda não possuem aplicação prática.

E

A aprendizagem por transferência é um processo que consome muitos recursos computacionais, por isso a relação custo-benefício precisa ser analisada antes de sua aplicação.



A alternativa B está correta.

Uma das formas de aprendizagem por transferência é a não supervisionada. Ela tem foco em tarefas não supervisionadas no domínio de destino. Nesse cenário, os domínios de origem e destino são semelhantes, mas as tarefas são diferentes.

Questão 2

O método de gradientes é fundamental para os modelos de aprendizado de máquina. Por outro lado, existem alguns problemas conhecidos quando esse método é utilizado. Em relação aos problemas conhecidos da aplicação do método de gradientes para os modelos de aprendizado de máquina, selecione a opção correta.

A

Um dos problemas mais importantes é o de dissipação do gradiente, que ocorre quando os valores do gradiente aumentam muito, levando à dispersão da convergência do modelo.

B

A explosão do gradiente afeta diretamente o resultado do modelo de aprendizado, pois os valores dos elementos do gradiente ficam cada vez menores, levando à explosão do número de iterações.

C

Um problema importante é o de ponto de sela, que ocorre quando o algoritmo converge para um ponto que não é o ótimo.

D

Um dos problemas mais sérios que pode ocorrer é o da implosão do gradiente, que ocorre quando o modelo ocasiona um número excessivo de erros numéricos, tornando inviável a sua convergência.

E

Apesar de existirem diversas categorias de problemas do algoritmo de gradiente, raramente elas ocorrem. Na prática, se ocorrerem, basta reiniciar o algoritmo com outro conjunto de pesos.



A alternativa C está correta.

Um dos problemas relacionados ao método dos gradientes é o ponto de sela, em que o algoritmo fica “estacionado” em um ponto da função de erro que não permite sua minimização.

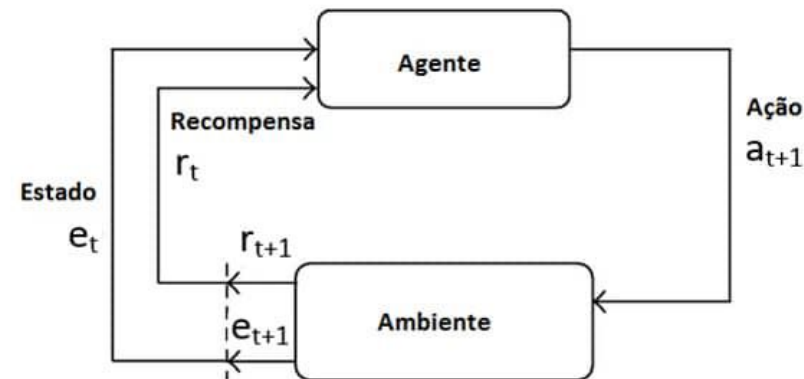
Aprendendo a otimizar recompensas

O aprendizado por reforço é um método de aprendizado de máquina voltado para tratar como os agentes de software devem realizar ações em um ambiente (LUGER, 2013). Ele integra os métodos de aprendizado profundo, dando suporte para maximizar recompensas cumulativas.

Nesse método, o agente não é informado por exemplos positivos e negativos.

O aprendizado é baseado em um método de tentativa e erro para encontrar as políticas ideais.

Todo o processo de aprendizagem é desenvolvido a partir de ideias em psicologia. O agente obtém recompensas a partir da interação com seu ambiente; assim, ele pode melhorar ou enfraquecer a execução de ações para obter as políticas ideais. Portanto, o objetivo final da aprendizagem por reforço é maximizar a recompensa cumulativa. A estrutura de aprendizagem é mostrada na imagem a seguir.



Estrutura de aprendizagem por reforço.

Vamos explicar um pouco mais sobre os principais elementos da aprendizagem por reforço que estão representados na imagem.

Agente

É uma entidade que realiza ações em um ambiente para ganhar alguma recompensa.

Ambiente

É o cenário em que um agente está contextualizado, ou seja, onde realiza ações e aprende.

Estado (e)

É uma situação concreta e imediata em que o agente se encontra. Por exemplo, o agente pode estar em um local e em momento específicos. Trata-se de uma configuração instantânea que coloca o agente em relação a outras coisas significativas, como ferramentas, obstáculos, inimigos ou prêmios.

Recompensa (r)

É o retorno dado para o agente como modo de premiar o seu sucesso ou penalizar o seu fracasso, ao realizar ações em determinado estado. Por exemplo, em um jogo, quando um personagem atinge um objetivo, ele ganha pontos. A partir de qualquer estado, um agente realiza ações no ambiente e, por sua vez, o ambiente retorna o novo estado do agente, bem como sua recompensa. As recompensas podem ser imediatas ou, ainda, atrasadas. Elas são o modo como as ações do agente são avaliadas efetivamente.

Política (π)

É uma estratégia aplicada pelo agente para decidir a próxima ação com base no estado atual. Ela faz o mapeamento dos estados nas ações com maiores recompensas.

Valor (V)

É definido como o retorno esperado a longo prazo do estado atual em relação à política π . Quanto mais longo for o tempo para se obter uma recompensa, maior será o desconto do seu valor aplicado, ou seja, o seu valor estimado é reduzido.

Função valor

Ela mede qual é o valor de um estado, isto é, calcula o valor total da recompensa. É essa função que o agente tem como objetivo maximizar.

Valor Q

É bastante semelhante ao valor. A diferença entre os dois é que o valor Q leva em consideração o estado, a ação e a política π adotada pelo agente, ou seja, trata do valor da recompensa de um agente em relação aos pares de estado-ação para determinada política. Normalmente, ela é representada por $Q^\pi(e, a)$.

A ideia da otimização de recompensas, portanto, consiste no mapeamento de ações a valores associadas a políticas contextualizadas no ambiente em que o agente vai atuar.

Políticas de rede neural

Como vimos, uma política de rede neural é uma regra usada pelo agente para decidir o que fazer, dado o conhecimento do estado atual do ambiente.

A política é definida matematicamente (HAYKIN, 2008) por:

$$\pi = \{\mu_0, \mu_1, \mu_2, \dots\}$$

Sendo que μ_t é uma função que mapeia o estado $E_t = e$ em uma ação $A_t = a$ no instante t .

Esse mapeamento é tal que:

$$\mu_t(e) \in A_e, \forall e \in E$$

Onde A_e representa o conjunto de todas as ações possíveis tomadas pelo agente no estado e .

As políticas organizadas de forma a serem utilizadas pelas redes neurais são chamadas de políticas admissíveis. Uma política pode ser **não estacionária** ou **estacionária**:

Política não estacionária

Varia com o tempo. Em termos matemáticos, isso é dado exatamente como representamos o conjunto de elementos de uma política, com μ_t variando ao longo do tempo.



Política estacionária

É independente do tempo. Em outras palavras, uma política estacionária especifica exatamente a mesma ação cada vez que ocorre determinado estado.

Basicamente, existem duas categorias principais para a otimização da política na aprendizagem por reforço: os **métodos baseados em valor** e os **métodos baseados em políticas**. Existe ainda uma classe de algoritmos conhecida como **ator-crítico**, que é uma combinação dessas duas categorias, o que potencializa a função valor para atualizar a política.

Métodos baseados em valor

Geralmente, implicam otimização da função valor $Q^\pi(e, a)$. A otimização da função vai encontrar a política que gera o melhor retorno. Isso é representado pela notação:

$$Q^{\pi^*}(e, a) = \max_a Q^\pi(e, a)$$

Onde o "*" representa a solução ótima. Como nesses algoritmos trabalhamos com soluções aproximadas, a política ótima pode ser representada por:

$$\pi^* \approx \arg \max_{\pi} Q^\pi$$

Onde o símbolo " \approx " é usado para representar o erro de aproximação.

Os algoritmos mais comuns baseados em valores são:

- Q-learning.
- DQN.

As vantagens dessa categoria de algoritmos é que normalmente eles são mais eficientes em termos de amostra do que os algoritmos baseados em políticas. Isso ocorre porque eles têm menor variância e fazem melhor uso dos dados coletados do ambiente.

No entanto, não há garantias de que esses algoritmos irão convergir para uma solução ótima. Em sua formulação-padrão, eles também são aplicáveis apenas a ambientes com espaços de ação discretos.

Método baseado em política

Ele faz a otimização através de um processo de atualização da política de modo iterativo até que o retorno cumulativo seja maximizado. A ideia é que, se um agente precisa agir em um ambiente, faz sentido aprender uma política. O que constitui uma boa ação em determinado momento depende do estado; portanto, uma função de política toma um estado e como entrada para produzir uma ação a .

Vantagem

Fazem parte de uma classe muito geral de métodos de otimização e podem ser aplicados a problemas com qualquer tipo de ação, discreta, contínua ou mista. Além disso, essa classe de métodos converge localmente para uma política ótima.



Desvantagem

Têm alta variância e são ineficientes para a amostra.

Alguns algoritmos baseados em políticas comuns incluem gradiente de política, região de confiança e evolução.

Aprendizado por diferenças temporais e Q-learning

Para podermos entender o método de aprendizado por diferença temporal, precisamos compreender as funções valor (ou funções de valor). Elas são funções que recebem como entrada o par “estado e ação” e, assim, estimam quão boa uma ação particular será em determinado estado, ou seja, qual será o retorno para aquela ação. Elas são representadas matematicamente por:

É o valor esperado de um estado e em relação a política π .

É o valor esperado ao iniciar a ação a no estado e seguindo a política π .

O problema em questão é estimar essas funções valor para uma política específica. O nosso objetivo é estimar essas funções de valor para que possam ser usadas para escolher uma ação que fornecerá a melhor recompensa total possível, depois de estar naquele determinado estado.

Os métodos de aprendizado por diferença temporal são usados para estimar essas funções valor. Se não fosse feita a estimativa das funções valor, o agente precisaria esperar até que a recompensa final fosse recebida antes que qualquer valor do par estado-ação pudesse ser atualizado. Assim que a recompensa final fosse recebida, o caminho percorrido para chegar ao estado final precisaria ser rastreado e cada valor atualizado de acordo. Isso pode ser expresso formalmente como (HAYKIN, 2008):

$$V(E_t) \leftarrow V(E_t) + \alpha [R_t - V(E_t)]$$

Onde E_t é o estado no instante t , R_t é a recompensa no instante t e α é um parâmetro constante.

Nos métodos de aprendizado por diferença temporal, uma estimativa da recompensa final é calculada em cada estado, e o valor da ação do estado é atualizado para cada etapa do caminho. A fórmula, portanto, é expressa como:

$$V(E_t) \leftarrow V(E_t) + \alpha [R_{t+1} + \gamma V(E_{t+1}) - V(E_t)]$$

Onde E_{t+1} é a recompensa observada no instante $t + 1$.

Uma importante distinção entre algoritmos de aprendizado por reforço é se eles estão dentro ou fora da política. Essa característica determina como as iterações de treinamento usam os dados.

Algoritmo dentro da política

Um algoritmo está **dentro da política** se aprender sobre a política – ou seja, o treinamento só pode utilizar dados gerados a partir da política atual. Isso significa que, conforme o treinamento avança por meio de versões de políticas, $\pi_1, \pi_2, \pi_3, \dots$, cada iteração de treinamento usa apenas a política atual naquele momento para gerar dados de treinamento. A consequência é que todos os dados devem ser descartados após o treinamento, uma vez que se tornam inutilizáveis. Isso torna os métodos dentro da política ineficientes para a amostragem – eles exigem mais dados de treinamento.

Algoritmo fora da política

No caso dos algoritmos que estão **fora da política**, quaisquer dados coletados podem ser reutilizados no treinamento. Consequentemente, os métodos fora da política são mais eficientes na amostragem, mas isso pode exigir muito mais memória para armazenar os dados.



Atenção

Os métodos de diferença temporal estão dentro da política, ou seja, aprendem o valor da política que é usada para tomar decisões.

As funções de valor são atualizadas usando-se os resultados da execução de ações determinadas por alguma política. Essas políticas são geralmente suaves e não determinísticas. O significado de suave, nesse sentido, é que ele garante que sempre haja um elemento de exploração na política. A política não é tão rígida a ponto de sempre escolher a ação que mais recompensa. As políticas mais comuns são:

- ϵ -soft.
- ϵ -greedy.
- Softmax.

Os métodos fora da política podem aprender diferentes políticas de comportamento e estimativa. Neles, a política de comportamento é geralmente suave. Algoritmos fora da política podem atualizar as funções de valor estimado usando ações hipotéticas, aquelas que não foram realmente tentadas. Isso contrasta com os métodos dentro da política, que atualizam funções de valor com base estritamente na experiência. Isso significa que algoritmos fora da política podem separar a exploração do controle, e os algoritmos dentro da política não. Em outras palavras, um agente treinado usando um método fora da política pode acabar aprendendo táticas que não necessariamente exibiu durante a fase de aprendizado.

O Q-learning é um método de aprendizado por reforço que usa Q-valores (também chamado de valores de ação) para melhorar iterativamente o comportamento do agente no processo de aprendizado. Ele foi apresentado pela primeira vez em Watkins (1989).



Curiosidade

A letra Q no Q-learning não significa nada de especial; é apenas uma notação que foi adotada por Watkins em sua derivação original do algoritmo. Trata-se de um método fora da política.

A tarefa comportamental desse sistema é encontrar uma política ótima (ou seja, custo mínimo) depois de tentar várias sequências de ações e observar os custos obtidos e as transições de estado que ocorrem. Esse algoritmo trabalha com duas políticas: a de comportamento e a de estimativa. A primeira é voltada para o comportamento do agente, enquanto a segunda tem como objetivo estimar o valor de uma política.

A função valor do Q-learning pode ser representada da seguinte forma (HAYKIN, 2008):

$$Q(E_t, A_t) \leftarrow Q(E_t, A_t) + \alpha [R_t + \gamma \max_a Q(E_{t+1}, a) - Q(E_t, A_t)]$$

Essa regra de atualização para estimar o valor de Q é aplicada em cada etapa de iteração dos agentes com o ambiente. Vamos detalhar um pouco mais sobre os termos usados na função:

Representa o estado do agente no instante t .

Representa o próximo estado do agente.

É a próxima melhor ação a ser escolhida usando-se a estimativa do valor atual de Q , ou seja, é a ação com o valor Q máximo do próximo estado.

Recompensa atual observada do ambiente em resposta à ação atual A_t .

É o fator de desconto para recompensas futuras. É um valor real entre 0 e 1.

É o comprimento do passo levado para atualizar a estimativa de $Q(E_t, A_t)$.

A seguir, apresentamos o algoritmo Q-learning:

Iniciar $Q(E_0, A_0)$

Para $k = 1, 2, \dots$

Iniciar o estado e

Repetir

Escolher uma ação a do estado e usando uma política derivada de Q

Calcular $(E_t, A_t) \leftarrow Q(E_t, A_t) + \alpha [R_t + \gamma \max_a Q(E_{t+1}, a) - Q(E_t, A_t)]$

Atualizar o estado $E_t \leftarrow E_{t+1}$

Até que E_t seja um estado terminal

A política a a que se refere o algoritmo Q-learning pode ser a política ϵ -gananciosa (Greedy policy), que é uma política muito simples de escolha de ações usando as estimativas de valor Q atuais. Basicamente, ela funciona assim:

Com probabilidade

O algoritmo escolhe a ação que tem o valor Q mais alto.

Com probabilidade

O algoritmo escolhe qualquer ação aleatoriamente.

Utilizando o algoritmo DQN

Para a maioria das aplicações, não é viável representar a função valor Q como uma tabela contendo os valores para todas as possíveis combinações de ações e estados. Para tratar desses casos, foi desenvolvido o método Deep Q-Network (DQN). O DQN é um método de aprendizagem de reforço e fora da política. Ele é uma variante do Q-learning. Algumas das técnicas mais comuns que o DQN utiliza para resolver problemas relacionados à instabilidade do aprendizado são as seguintes:

Repetir a experiência (experience replay)

A ideia básica é que, ao armazenar as experiências de um agente e, em seguida, projetar lotes aleatoriamente delas para treinar a rede, podemos aprender de forma mais robusta e ter um bom desempenho na tarefa. Ao manter as experiências que extraímos aleatoriamente, evitamos que a rede apenas aprenda sobre o que está fazendo imediatamente no ambiente e permitimos que ela aprenda com uma gama mais variada de experiências anteriores. Cada uma dessas experiências é armazenada como uma tupla de <estado, ação, recompensa, próximo estado>. O buffer do experience replay armazena um número fixo de memórias recentes e, conforme as novas entram, as antigas são removidas. Quando chega a hora de treinar, simplesmente extraímos um lote de memórias aleatórias do buffer e treinamos nossa rede com elas.

Rede alvo (target network)

O DQN usa outra rede neural chamada rede de destino para obter um estimador do erro quadrático médio usado no treinamento da rede Q. A rede de destino é sincronizada com a rede Q após certo período de iterações.

Agora, apresentamos o algoritmo DQN (HAYKIN, 2008):

Iniciar $Q(E_0, A_0; \theta)$ com valores aleatórios para os pesos θ

Iniciar a memória B com o método Experience Replay

Iniciar o procedimento de exploração $\text{Eexplorar}(\cdot)$

Para $i = 1, 2, \dots, N$ faça

$$y_{e,a}^i = E_B[r + \max_a Q, (e_{t+1}, a_{t+1}; \theta_{i-1}) \mid e, a]$$

$$\theta_i \approx \arg \min_{\theta} E_B[(y_{e,a}^i - Q(e_t, a_t; \theta))^2]$$

Explorar(\cdot), Atualizar B

Fim

Saída $Q^{DQN}(e, a; \theta_N)$

Na linha 5, os valores alvo $y_{e,a}^i$ são construídos usando uma rede-alvo designada $Q(e_{t+1}, a_{t+1}; \theta_{i-1})$ com o uso dos parâmetros de iteração anteriores θ_{i-1} , em que o retorno esperado E_B é obtido a partir de uma amostra da distribuição de transições de experiência no buffer $(e_t, a_t; \theta) B$. Na linha 6, o algoritmo DQN resolve um problema de aprendizado supervisionado para aproximar a função valor $Q(e, a; \theta)$. A perda do DQN é minimizada usando-se uma variação do algoritmo de gradiente de descida estocástica. Além disso, o DQN utiliza um procedimento de exploração para interagir com o ambiente - por exemplo, um procedimento de exploração gananciosa que chamamos de **Explorar** (\cdot).

Aprendizado por reforço

O especialista Sérgio Assunção Monteiro fala sobre os principais tópicos a respeito do aprendizado por reforço abordados neste módulo.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

A aprendizagem por reforço é um modelo muito importante aplicado no aprendizado de máquina. Existem diversos elementos no modelo com responsabilidades bem específicas. Em relação aos principais elementos da aprendizagem por reforço, selecione a opção correta.

A

O agente é quem, de fato, executa ações em um ambiente com o objetivo de ganhar alguma recompensa.

B

O ambiente corresponde aos recursos computacionais disponíveis para o programa.

C

O estado se refere ao contexto em que a execução do modelo ocorre.

D

A recompensa corresponde à penalização dada a um agente por não ter obtido uma resposta positiva dentro de determinado período.

E

A política se refere aos aspectos de interação entre o agente e a recompensa de acordo com o estado em que esteja.



A alternativa A está correta.

O agente é a entidade responsável por realizar ações contextualizadas em um ambiente para ganhar alguma recompensa; assim, ele pode melhorar ou enfraquecer a execução de ações para obter as políticas ideais.

Questão 2

A política é um elemento fundamental em um modelo de aprendizado por reforço. É nela que são estabelecidas as estratégias, para que o modelo faça escolhas que melhorem seu desempenho rumo a um aprendizado mais preciso. Em relação à política de aprendizado por reforço, selecione a opção correta.

A

A política de um modelo de aprendizado por reforço nunca varia ao longo das iterações do algoritmo.

B

As políticas de um modelo de aprendizado por reforço podem ser tanto variantes como invariantes ao longo das iterações do modelo.

C

As políticas não estacionárias garantem que o algoritmo não vai convergir para um ponto de sela.

D

As políticas estacionárias não garantem que ocorram problemas com os gradientes dos modelos; no entanto, se esses problemas não ocorrerem, sempre vai se obter a solução ótima.

E

As políticas dos modelos de aprendizado por reforço não podem ser combinadas, pois isso pode gerar comportamentos não previsíveis.



A alternativa B está correta.

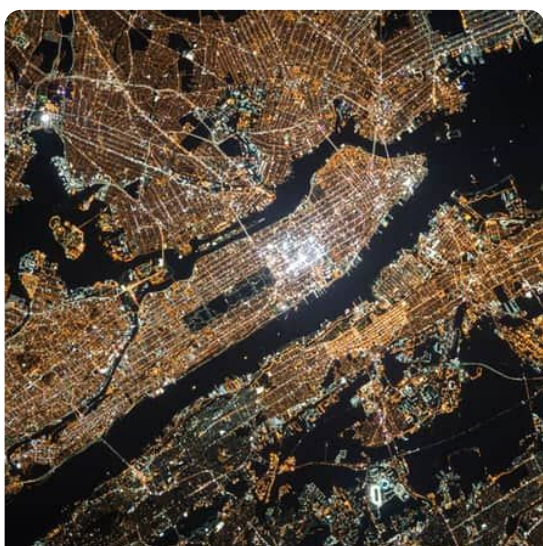
As políticas dos modelos de aprendizado por reforço podem ser estacionárias e não estacionárias. Existe até a possibilidade de combinar ambas as políticas.

Camadas e arquiteturas

O aprendizado de máquina tem diversas aplicações: sistemas de tomada de decisão, sistemas de recomendação, sistemas para identificar objetos a partir das imagens, entre tantas outras.

Os algoritmos de aprendizado de máquina usam exemplos de treinamento para descobrir padrões, construir um modelo e, em seguida, construir previsões sobre as novas informações com base no modelo.

Existem vários algoritmos de aprendizado de máquina usados para classificação. Um deles é o das redes neurais profundas, **deep neural network (DNN)**, que extraem características por si mesmas com várias camadas ocultas. Entre as principais aplicações das redes neurais profundas, estão:



Reconhecimento de imagem

No **reconhecimento de imagem**, a rede aprende a reconhecer pixels, bordas e texturas, por exemplo.



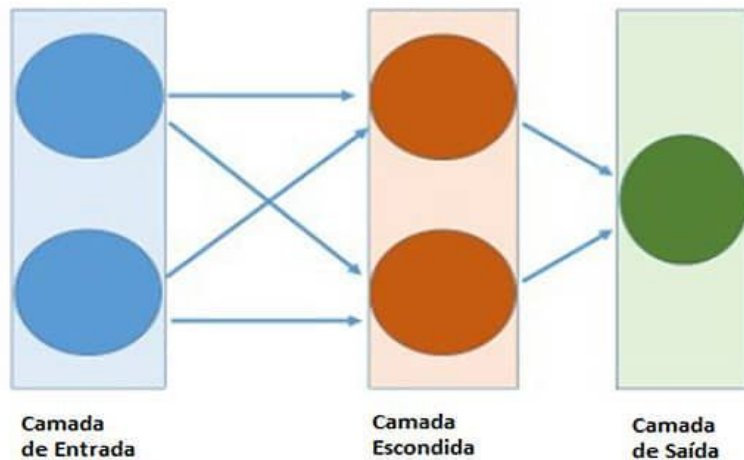
Reconhecimento de texto

No **reconhecimento de texto**, os recursos que a rede aprende são personagens, palavras, grupos de palavras e frases, por exemplo.

Existem três categorias de arquiteturas de aprendizado profundo: rede neural profunda, rede neural de convolução e rede neural recorrente. Vamos detalhar um pouco mais cada uma dessas arquiteturas:

Rede neural profunda (DNN)

São redes do tipo alimentação direta, chamadas de feed forward. É bastante comum que sejam usadas com esse nome em português, ou seja, **redes do tipo feed forward**, nas quais os dados fluem da camada de entrada para a camada de saída sem retroalimentação. Na imagem a seguir, apresentamos uma representação dessa rede:

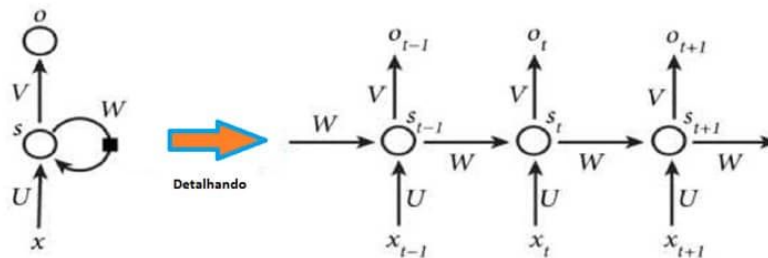


DNN com feed forward.

As saídas de uma DNN são obtidas por aprendizagem supervisionada.

Rede neural recorrente (RNN)

É uma classe de rede neural em que as conexões entre as unidades formam um ciclo direcionado, que pode processar sequências arbitrárias de entradas. O estado interno da rede permite que ela tenha **memória interna** e explore o comportamento dinâmico (temporal). Na imagem a seguir, apresentamos um exemplo de uma RNN.



Exemplo conceitual de uma RNN.

Os principais elementos que se destacam na imagem são:

U

Representa os pesos entre a camada de entrada e oculta.

W

Representa os pesos entre a camada oculta anterior e a atual.

V

Representa os pesos entre a camada de entrada e saída.

É a função de camada oculta tanh ou ReLU.

É o estado oculto no tempo t . Ele é obtido pela expressão: $S_t = f(Ux_t + WS_{t-1})$.

A RNN compartilha os mesmos valores de parâmetros (U, W, V) em cada camada. O treinamento de RNN tem o problema de aprender dependências de longo prazo, conforme a profundidade aumenta, o que pode resultar no problema de desaparecimento do gradiente. Para lidar com esse problema, vários métodos foram propostos, entre eles, o método denominado **long-short-term-memory (LSTM)**.

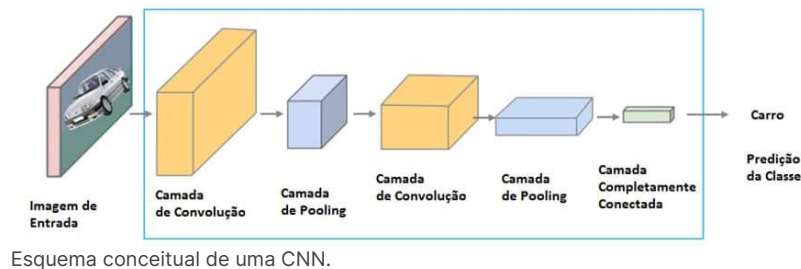
Rede neural de convolução (CNN)

Também são conhecidas como ConvNet. Elas normalmente são aplicadas para análise de imagens. Outras aplicações incluem compreensão de vídeos, reconhecimento de voz e compreensão de processamento de linguagem natural. Além disso, a CNN combinada com o LSTM tem sido aplicada com sucesso para legendagem automática de imagens. Os conceitos básicos por trás da CNN são: conectividade local, compartilhamento de parâmetros, pooling e o uso de muitas camadas.

A arquitetura básica contém três camadas:

- Camada de convolução.
- Camada de pooling.
- Camada totalmente conectada.

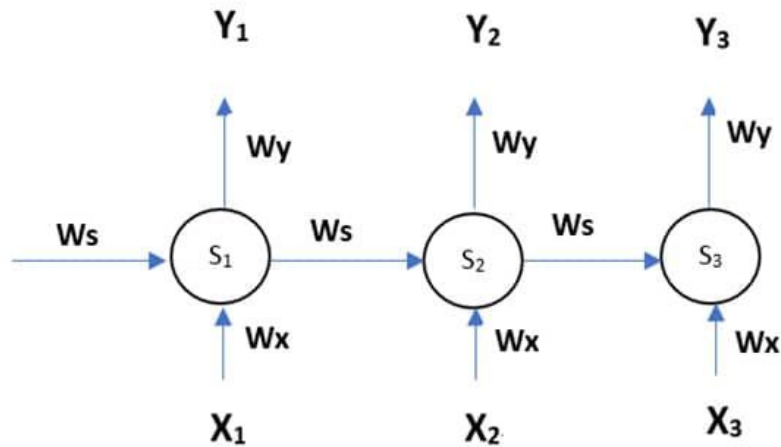
Na imagem, apresentamos um exemplo de uma CNN.



Treinando RNN

O treinamento de uma RNN é semelhante ao de uma rede neural artificial tradicional (ANN). Para que possam obter suas saídas, elas usam as entradas atuais e levam as que ocorreram antes delas em consideração. Isso significa que, na prática, a saída atual depende da entrada atual e das entradas anteriores, que é o que denominamos de memória.

Para treinar essas redes, utilizamos os algoritmos de retropropagação – mais conhecidos pela expressão *backpropagation*. A diferença do treinamento da RNN em relação ao de uma ANN é que, no caso da RNN, os parâmetros são compartilhados por todas as etapas de tempo na rede, ou seja, o gradiente em cada saída depende das etapas anteriores e não apenas da atual. Por exemplo, para calcular o gradiente no instante de tempo $t=3$, precisaríamos retropropagar 2 etapas e somar os gradientes. Isso é chamado de retropropagação ao longo do tempo (BTPP - *Backpropagation through time*). Na imagem, apresentamos a representação de uma RNN.



Arquitetura de RNN.

Na imagem, temos os seguintes elementos que se destacam:

- S_1, S_2 e S_3 são os estados ocultos, ou, ainda, as unidades de memória no tempo t_1, t_2, t_3 , respectivamente, e W_s é a matriz de peso associada aos estados.
- X_1, X_2 e X_3 são as entradas no tempo t_1, t_2, t_3 , respectivamente, e W_x é a matriz de peso associada às entradas.
- Y_1, Y_2 e Y_3 são as saídas no tempo t_1, t_2, t_3 , respectivamente, e W_y é a matriz de peso associada às saídas.

Para qualquer instante t , temos que a equação para obter o estado atual S_t é dada por:

$$S_t = f_1(W_x x_t + W_s S_{t-1})$$

A equação para obter a saída atual Y_t é dada por:

$$Y_t = f_2(W_y S_t)$$

Onde f_1 e f_2 são funções de ativação.

Como exemplo, vamos mostrar como o algoritmo realiza a retropropagação no instante de tempo $t = 3$.

Vamos tomar a função de erro quadrático E_t como:

$$E_t = (d_t - Y_t)^2$$

Onde:

- d_t : representa a saída desejada.
- Y_t : é a variável saída.

No caso do nosso exemplo, $t = 3$, portanto obtemos:

$$E_3 = (d_3 - Y_3)^2$$

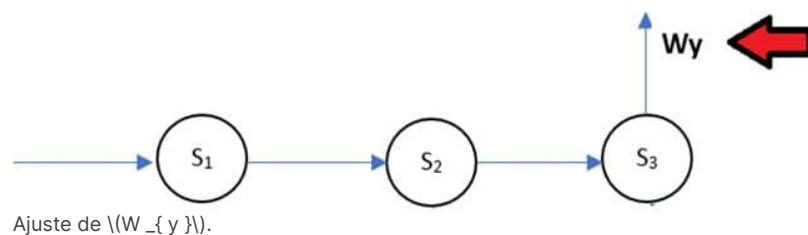
Para aplicar a retropropagação, temos de ajustar os pesos associados para:

- Entradas;
- Unidades de memória;
- Saídas.

Portanto, vamos ter de ajustar W_y, W_s e W_x .

Ajuste de

Na imagem a seguir, destacamos o trecho da imagem anterior de Arquitetura de RNN que se refere a W_y .



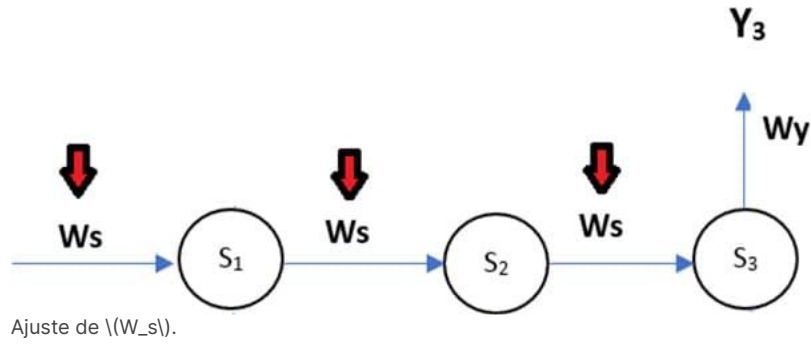
Como E_3 é uma função de Y_3 , fazemos a diferenciação de E_3 em relação à Y_3 . Além disso, Y_3 é uma função de W_y , então, aplicamos a diferenciação de Y_3 em relação à W_y . Por fim, obtemos a seguinte expressão:

$$\frac{\partial E_3}{\partial W_x} = \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial W_x} \right) + \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial S_2} \cdot \frac{\partial S_2}{\partial W_x} \right) + \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial S_1} \cdot \frac{\partial S_1}{\partial W_x} \right)$$

Os processos de ajuste de W_s e W_x são bem semelhantes, como veremos.

Ajuste de

Na imagem a seguir, destacamos o trecho da imagem de Arquitetura de RNN que se refere a W_s .

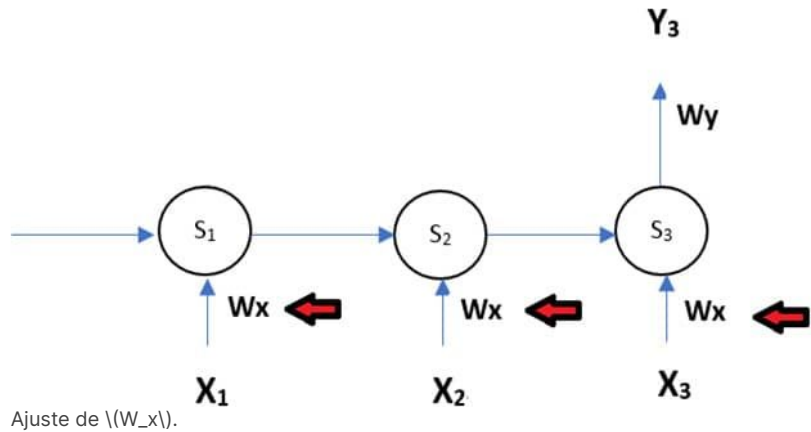


Como E_3 é uma função de Y_3 , fazemos a diferenciação de E_3 em relação a Y_3 . Além disso, Y_3 é uma função de S_3 , então, aplicamos a diferenciação de Y_3 em relação a S_3 . Agora, temos que S_3 é uma função de W_s , então, aplicamos a diferenciação de S_3 em relação a W_s . Ainda precisamos considerar os passos anteriores; isso significa que temos de diferenciar a função erro em relação às unidades de memória S_2 e S_1 em relação a W_s , portanto, diferenciamos S_3 em relação a S_2 e derivamos S_2 em relação a S_1 . Por fim, obtemos a seguinte expressão:

$$\frac{\partial E_3}{\partial W_s} = \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial W_s} \right) + \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial S_2} \cdot \frac{\partial S_2}{\partial W_s} \right) + \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial S_2} \cdot \frac{\partial S_2}{\partial S_1} \cdot \frac{\partial S_1}{\partial W_s} \right)$$

Ajuste de

Na imagem a seguir, destacamos o trecho da imagem de Arquitetura de RNN que se refere a W_x .



Como E_3 é uma função de Y_3 , fazemos a diferenciação de E_3 em relação a Y_3 . Além disso, Y_3 é uma função de S_3 , então, aplicamos a diferenciação de Y_3 em relação a S_3 . Agora, temos que S_3 é uma função de W_x , então, aplicamos a diferenciação de S_3 em relação a W_x . Ainda precisamos considerar os passos anteriores; isso significa que temos de diferenciar a função erro em relação às unidades de memória S_2 e S_1 em relação a W_x , portanto, diferenciamos S_3 em relação a S_2 e derivamos S_2 em relação a S_1 . Por fim, obtemos a seguinte expressão:

$$\frac{\partial E_3}{\partial W_x} = \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial W_x} \right) + \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial S_2} \cdot \frac{\partial S_2}{\partial W_x} \right) + \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial S_3} \cdot \frac{\partial S_3}{\partial S_2} \cdot \frac{\partial S_2}{\partial S_1} \cdot \frac{\partial S_1}{\partial W_x} \right)$$

O método de retropropagação BPTT tem uma limitação de camadas (etapas de tempo) que podem ser levadas em consideração nos ajustes dos pesos. Isso ocorre porque, quando se trabalha com muitas

camadas, o gradiente se torna muito pequeno. Esse problema é chamado de problema do **desaparecimento de gradiente**, conhecido em inglês como vanishing.

Uma das soluções para esse problema é usar o método LSTM. No entanto, o LSTM também tem um problema: o do gradiente explosivo, em que o gradiente se torna muito grande. A solução encontrada foi desenvolver um método chamado **recorte do gradiente**, conhecido em inglês como clipping, que pode ser usado em cada etapa de tempo.

A ideia básica é: se a norma do gradiente for superior a determinado valor, então, o gradiente é cortado, ou seja, ele passa por um processo para limitar os seus valores.

RNN profundas

Como vimos anteriormente, as RNNs são muito importantes para diversas aplicações que envolvem processamento de dados sequenciais, mas possuem alguns problemas que são tratados por outros métodos. A evolução das RNNs levou às redes neurais recorrentes com profundidade (DRNN - Deep recurrent neural network). As DRNNs têm duas características:

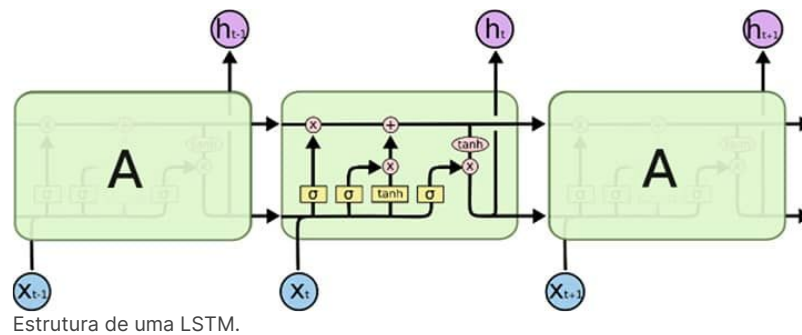
- Elas consistem em várias camadas empilhadas. Essa abordagem é conhecida como aprendizado profundo. As arquiteturas profundas têm se mostrado altamente eficazes em várias aplicações, devido à sua capacidade de aprender a representação hierárquica dos dados. As camadas iniciais aprendem recursos simples, enquanto as camadas seguintes aprendem conceitos mais complicados.
- Têm capacidade de preservar seu estado interno ao longo do tempo. Isso ocorre pela introdução de conexões recorrentes na rede que retornam à saída anterior do neurônio para si mesmo, ou para outras unidades na mesma camada.

Uma arquitetura bastante usada de DRNNs é a de blocos de construção, chamados unidades de memórias longas de curto prazo (LSTM - Long short-term memory).

As LSTMs se lembram de seu estado interno por longo ou curto períodos. Normalmente, são compostas por quatro componentes:

- **Célula de memória.**
- **Porta de entrada:** Controla a entrada de informações em cada intervalo de tempo.
- **Porta de saída:** Controla quanta informação é enviada para a próxima célula, ou camada superior.
- **Porta de esquecimento:** Controla a quantidade de dados a perder em cada etapa de tempo.

Cada porta está conectada a outra por meio de sua entrada e saída, e várias conexões são recorrentes. Na imagem a seguir, apresentamos a estrutura de uma LSTM.

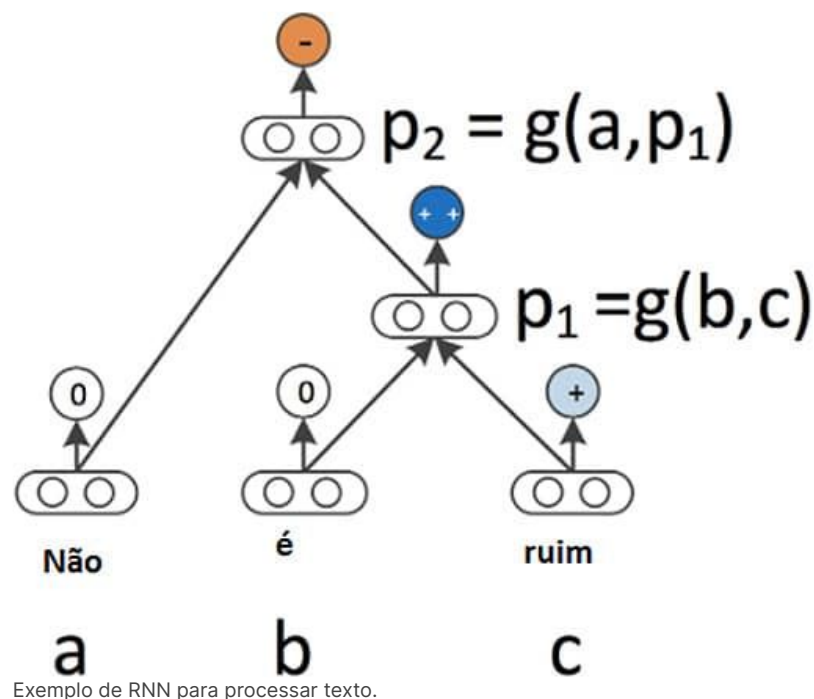


Na imagem, podemos ver que a LSTM é composta de blocos de memória chamados células, sendo referenciadas por C_t . Dois estados são transferidos para a próxima célula: o estado da célula C_t e o estado oculto h_t . O estado da célula é a cadeia principal do fluxo de dados, o que permite que os dados fluam para a frente sem que sejam alterados. Os dados podem ser adicionados, ou removidos do estado da célula por meio de portas sigmóides. Uma porta é semelhante a uma camada, ou série de operações de matriz, que contém pesos individuais diferentes.

Os LSTMs são projetados para evitar o problema de dependência de longo prazo porque usam portas para controlar o processo de memorização.

Processamento de linguagem natural

Uma aplicação muito importante de aprendizado profundo e que tem sido aperfeiçoada ao longo dos anos é o processamento de linguagem natural (PLN). Tanto as RNNs como as CNNs tratam a linguagem como uma sequência de dados. Temos de observar que a linguagem em si tem uma estrutura hierárquica: as palavras são compostas em frases e orações que podem ser combinadas recursivamente de acordo com um conjunto de regras de produção. A ideia inspirada linguisticamente de tratar sentenças como árvores em vez de uma sequência dá origem a redes neurais recursivas (SOCHER *et al.*, 2013), que podem ser vistas na imagem a seguir.



Na imagem, podemos ver que, a cada nó da árvore da RNN, uma nova representação é calculada, compondo as representações dos nós filhos. Por exemplo, a frase de entrada foi: "PLN é incrível!". A primeira composição,

no caso, p_1 , é composta pelas palavras "é" e "incrível". A composição p_2 é composta com o resultado p_1 e com a palavra "PLN".

Quando é feita uma classificação de texto, o objetivo da previsão da rede é classificar a que grupo ou grupos o texto pertence. Uma das aplicações mais comuns nessa área é determinar se o sentimento de um trecho de texto é positivo, ou negativo. Então, se uma RNN for treinada para prever texto de um corpus – grandes conjuntos de dados textuais – dentro de determinado domínio, então, essa rede é uma forte candidata para ser reaproveitada para classificação de texto dentro desse domínio, que é a situação de reutilização de camadas pré-treinadas, como discutimos anteriormente. A ideia é reaproveitar os pesos dentro da estrutura interna da rede.

A PLN tem diversas aplicações, sendo as mais comuns:

Tradução automática

É um processo que usa técnicas para traduzir automaticamente o texto de um idioma para outro, sem a necessidade de intervenção humana.

Reconhecimento de entidades nomeadas (REN)

Tem como objetivo identificar entidades (nomes próprios) contidas em um texto, como, por exemplo, identificar pessoas, locais, organizações contextualizadas em domínios específicos.

Classificação automática de textos

Tem como objetivo encontrar categorias preestabelecidas para determinado texto. Uma possível aplicação é a de analisar textos não estruturados e associá-los a categorias que, posteriormente, poderão ser analisadas por métodos estatísticos.

Análise de sentimentos

Tem como objetivo classificar as opiniões relacionadas a determinado assunto em categorias. Essas categorias, normalmente, são do tipo positivo, negativo ou neutro. Uma das aplicações atuais é na pesquisa de opinião do histórico de bate-papo e tópicos de mídias sociais.

Sistemas de perguntas e respostas

Tem como objetivo dar respostas a partir da análise de perguntas elaboradas em linguagem humana. Uma aplicação cada vez mais importante nessa área são os assistentes virtuais, chamados, normalmente, de **chat bots**.

Extração da informação

Tem como objetivo obter conteúdo relevante e estruturado restrito a determinado domínio.

Sumarização automática

A partir da análise de um conteúdo textual, tem como objetivo elencar os trechos mais significativos de modo a obter um resumo legível.

Busca semântica

Tem como objetivo aplicar mecanismos de buscas mais elaborados para compreender os critérios de pesquisa por determinada informação. É fundamental para quase todas as tarefas de pesquisa na internet e PNL, como, por exemplo, para a tradução automática, em que busca semelhança entre frases em diferentes idiomas, como para realizar pesquisas na web, em que busca por semelhança entre consultas e documentos.

Redes neurais de convolução e recorrentes

O especialista Sérgio Assunção Monteiro fala sobre os principais tópicos a respeito de redes neurais de convolução e recorrentes abordados neste módulo.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Existem três categorias de arquiteturas de aprendizado profundo: rede neural profunda, rede neural de convolução e rede neural recorrente. Cada uma delas se aplica melhor a um contexto. Em relação às categorias de arquiteturas de aprendizado profundo, selecione a opção correta.

A

Rede neural profunda é caracterizada pela retroalimentação.

B

As saídas de uma rede neural profunda são obtidas por aprendizagem por reforço.

C

A rede neural recorrente é caracterizada por ter conexões entre as suas unidades, de modo a formar um ciclo direcionado.

D

A rede neural recorrente é adaptada para aprender dependências de longo prazo, conforme a profundidade aumenta.

E

A rede neural de convolução é aplicada com bastante sucesso em processos de tomada de decisão.



A alternativa C está correta.

A classe da rede neural recorrente possui conexões entre as suas unidades, de modo que formam um ciclo direcionado. São aplicadas com bastante sucesso para problemas de processamento de linguagem natural.

Questão 2

O treinamento de uma rede neural recorrente é semelhante ao de uma rede neural artificial tradicional. A rede neural recorrente é usada com sucesso em diversas aplicações. Em relação à rede neural recorrente, selecione a opção correta.

A

Elas usam as entradas individuais para serem processadas pelas camadas escondidas e, finalmente, obter a saída.

B

São denominadas como redes sem memória.

C

Usam algoritmos de alimentação direta (feed forward) para realizar o treinamento do modelo.

D

Os parâmetros são compartilhados por todas as etapas de tempo na rede.

E

Utilizam outras estratégias de otimização para evitar os problemas do algoritmo de gradientes.



A alternativa D está correta.

As redes neurais recorrentes compartilham os parâmetros por todas as etapas de tempo na rede. Isso significa que o gradiente em cada saída depende das etapas anteriores, e não apenas da atual.

Simulação com TensorFlow

O TensorFlow é uma plataforma de código aberto criada pelo Google para computação numérica e é bastante usada para implementar modelos de machine learning. Uma das vantagens de se trabalhar com essa biblioteca é que ela se consolidou ao longo dos anos e, atualmente, é a mais popular aplicada para aprendizado de máquina. Isso significa que encontrar documentação, exemplos e fóruns de discussão que tratam de dúvidas e problemas relacionados à biblioteca é bastante facilitado.

O TensorFlow utiliza algumas APIs com objetivos específicos que facilitam o trabalho de desenvolvimento. As principais APIs são:

tf.data

Permite manipular grandes quantidades de dados, fazer a leitura de diferentes formatos e realizar transformações complexas.

tf.keras

É a API do TensorFlow para criar e treinar modelos de aprendizado profundo.

tf.estimators

Fornecer métodos para treinar e avaliar a precisão do modelo e para gerar previsões.

API Distribution Strategy

É usada para distribuir treinamento em várias GPUs, várias máquinas, ou TPUs.

O TensorFlow não é a única biblioteca de aprendizado profundo. É interessante conhecer outras plataformas, que podem ser úteis para situações específicas e, em muitos casos, podem ser usadas concomitantemente ao TensorFlow. Entre as principais plataformas de aprendizado profundo, estão:

Keras

É uma biblioteca de rede neural de código aberto escrita em Python, que pode ser executada em TensorFlow, Microsoft Cognitive Toolkit (CNTK), Theano, R e PlaidML. Normalmente, ela é usada como um complemento ao TensorFlow.

PyTorch

É uma biblioteca de rede neural de código aberto desenvolvida e mantida principalmente pelo AI Research Lab do Facebook (FAIR). Ela é a principal concorrente do TensorFlow.

MXNet

É um framework de aprendizado profundo de código aberto desenvolvido pela Apache Foundation. Ele tem suporte em várias linguagens de programação, tais como C++, Python, Java, Julia, MATLAB, JavaScript, Go, R, Scala, Perl e Wolfram Language.

CNTK

Foi lançado pela Microsoft como seu framework de código aberto para aprendizado profundo. Também é conhecido como Microsoft Cognitive Toolkit. É suportado pelas linguagens de programação Python, C++, C# e Java.

Executando alguns exemplos no TensorFlow

A forma mais fácil de usar o TensorFlow é com o Google Colab. No Google Colab, o TensorFlow já está pré-instalado.

Vamos apresentar dois exemplos que usam o TensorFlow com as seguintes características:

Exemplo CNN

Uma aplicação de uma rede neural de convolução (CNN) para classificação de imagens. Para isso, vamos usar o conjunto de dados MNIST, que é um banco de dados de dígitos manuscritos usado com bastante frequência para treinar vários sistemas de processamento de imagem.

Exemplo RNN

Uma aplicação de uma rede neural recorrente (RNN) para prever se uma revisão é positiva ou negativa. Vamos utilizar IMDB, que é um grande conjunto de dados de revisão de filmes bastante popular.

Exemplo CNN

Para facilitar a compreensão da sequência de execução do programa, vamos dividi-lo em etapas.

CNN 1: Fazer download dos dados do MNIST

No código a seguir, mostramos como baixar os dados do MNIST.

plain-text

```
import tensorflow as tf
import tensorflow_datasets as tfds
(x_treino,y_treino),(x_teste,y_teste)=tfds.as_numpy(
    tfds.load('mnist',
              split=['train', 'test'],
              batch_size=-1,
              as_supervised=True,
              shuffle_files=True
    ))
```

O MNIST contém 60 mil imagens de treinamento e 10 mil imagens de teste.

Nas linhas 1 e 2, importamos as bibliotecas tensorflow e tensorflow_datasets.

Na linha 3, separamos os dados para treinamento e teste e também separamos os rótulos e as imagens. As variáveis x_treino e x_teste contêm códigos RGB em tons de cinza (de 0 a 255), enquanto as partes y_treino e y_teste contêm rótulos de 0 a 9.

CNN 2: Visualização dos dados

No código a seguir, mostramos como visualizar os dados do MNIST.

```
plain-text

import matplotlib.pyplot as plt
img_index = 8888 #You may pick a number up to 60,000
print("O dígito na imagem:", y_treino[img_index])
plt.imshow(x_treino[img_index].reshape(28,28), cmap='Greys')
```

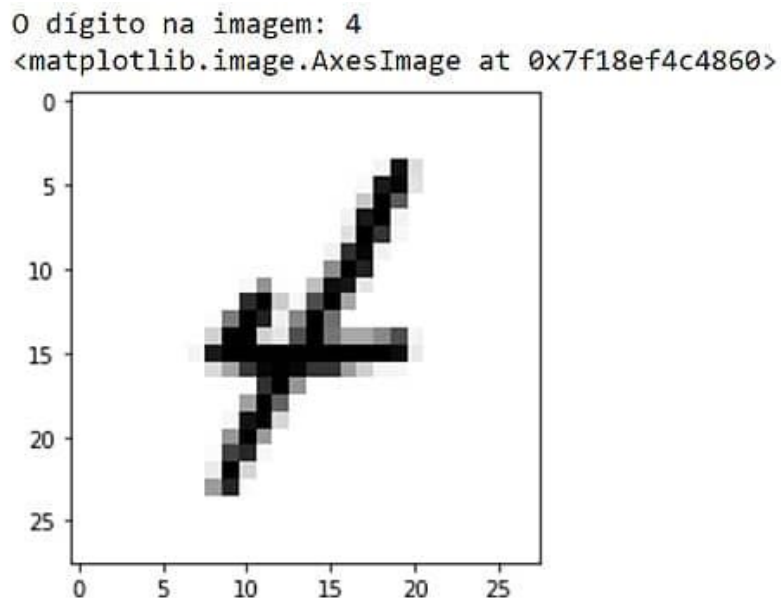
Fazemos a importação da biblioteca matplotlib na linha 1. Ela será necessária para visualizarmos as imagens.

Na linha 2, selecionamos o dado de índice 8888. Podemos selecionar valores que vão de 0 a 64.999.

Na linha 3, exibimos o rótulo associado ao índice 8888.

Na linha 4, exibimos a imagem associada ao índice 8888.

A imagem a seguir apresenta o resultado dessa etapa.



Resultado da visualização do rótulo e da respectiva imagem. Captura de tela do TensorFlow.

Também precisamos saber a forma do conjunto de dados para entrar com ele na CNN. Portanto, usamos o atributo de forma da matriz NumPy, conforme o código a seguir.

```
plain-text  
x_treino.shape
```

Este produz a saída que é exibida na imagem a seguir.



Shape da imagem. Captura de tela do TensorFlow.

CNN 3: Remodelar e normalizar as Imagens

No código a seguir, mostramos como remodelar e normalizar as imagens do MNIST.

```
plain-text  
  
x_treino = x_treino.astype('float32')  
x_teste = x_teste.astype('float32')  
x_treino /= 255  
x_teste /= 255  
print('Forma de x_treino:', x_treino.shape)  
print('Quantidade de imagens em x_treino', x_treino.shape[0])  
print('Quantidade de imagens em x_teste', x_teste.shape[0])
```

Nas linhas 1 e 2, para que possamos obter pontos decimais após a divisão, precisamos que os valores sejam de ponto flutuante.

Nas linhas 3 e 4, fazemos a normalização dos códigos RGB na escala de cinza.

CNN 4: Construir a rede neural de convolução (CNN)

No código a seguir, mostramos como construir a CNN (YALÇIN, 2021, p. 156).

```
plain-text  
  
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten,  
    MaxPooling2D  
model = Sequential()  
model.add(Conv2D(28, kernel_size=(3,3), input_shape=(28,28,1)))  
model.add(MaxPooling2D(pool_size=(2,2)))  
model.add(Flatten())  
model.add(Dense(128, activation=tf.nn.relu))  
model.add(Dropout(0.2))  
model.add(Dense(10, activation=tf.nn.softmax))
```

Nas linhas 1 e 2, importamos os programas necessários do Keras que contêm modelo e camadas.

Das linhas 4 a 6, criamos um modelo sequencial e adicionamos camadas.

Das linhas 7 a 10, achatamos os vetores 2D para camadas totalmente conectadas.

CNN 5: Compilar e ajustar (fitting) o modelo

No código a seguir, mostramos como compilar e ajustar o modelo.

```
plain-text

model.compile(optimizer='adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
model.fit(x=x_treino, y=y_treino, epochs=10)
```

Na linha 1, definimos um otimizador com determinada função de perda que usa uma métrica.

Na linha 4, fazemos o ajuste do modelo usando nossos dados de treino e estabelecemos 10 épocas para o algoritmo aprender.

CNN 6: Avaliar o modelo

No código a seguir, mostramos como avaliar o modelo.

```
plain-text

model.evaluate(x_teste, y_teste)
```

Que produz a saída da a seguir.

```
313/313 [=====] - 2s 7ms/step - loss: 0.0580 - accuracy: 0.9864
[0.058032721281051636, 0.9864000082015991]
```

Qualidade do modelo. Captura de tela do TensorFlow.

CNN 7: Predição de dados

Agora que já entramos com os dados e já treinamos a CNN, queremos usá-la para fazer a predição de dados. No código a seguir, mostramos como fazer a predição.

```
plain-text

img_pred_index = 1000
plt.imshow(x_teste[img_pred_index].reshape(28,28), cmap='Greys')
pred = model.predict(x_teste[img_pred_index].reshape(1,28,28,1))
print("Nosso modelo CNN prevê que o dígito na imagem é:", pred.argmax())
```

Na linha 1, entramos com o índice de uma imagem do banco.

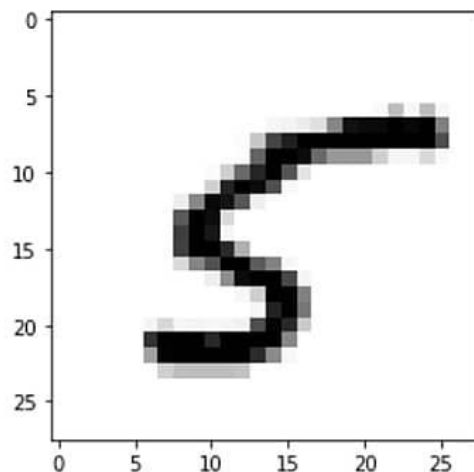
Na linha 2, exibimos a imagem apenas para visualizarmos.

Na linha 3, o modelo faz a predição, ou seja, ele vai associar, conforme o aprendizado dele, a imagem a um rótulo.

Na linha 4, apresentamos o rótulo que o modelo obteve por sua predição.

Na imagem a seguir, apresentamos o resultado da saída do programa.

Nosso modelo CNN prevê que o dígito na imagem é: 5



Resultado da predição do modelo. Captura de tela do TensorFlow.

CNN 8: Salvar modelo

Agora que construímos a CNN para classificar dígitos com a API sequencial Keras do Tensorflow e obtivemos um nível de precisão de mais de 98%, vamos salvar este modelo. A seguir, apresentamos o código.

```
plain-text\n\n!mkdir -p modelo_salvo\nmodel.save('modelo_salvo/classificador_digitos')
```

Na linha 1, criamos uma pasta 'modelo_salvo'.

Na linha 2, salvamos o modelo completo com suas variáveis, pesos e bias (vieses).

Exemplo RNN

Do mesmo modo que apresentamos o exemplo da CNN, vamos apresentar o código do RNN em etapas.

RNN 1: Fazer as importações das bibliotecas

No código a seguir, mostramos como importar as bibliotecas do tensorflow e tensorflow_datasets.

```
plain-text\n\nimport tensorflow as tf\nimport tensorflow_datasets as tfds
```

A biblioteca tensorflow_datasets será necessária para importar o conjunto de dados que vamos trabalhar.

RNN 2: Carregando o conjunto de dados do TensorFlow

No código a seguir, mostramos como fazer a importação dos dados do IMDB por meio da biblioteca tensorflow_datasets.

```
plain-text

dataset, info = tfds.load('imdb_reviews/subwords8k',
                          with_info=True,
                          as_supervised=True)
```

Os dados do IMDB já estão em um formato que facilita o trabalho com o tensorflow.

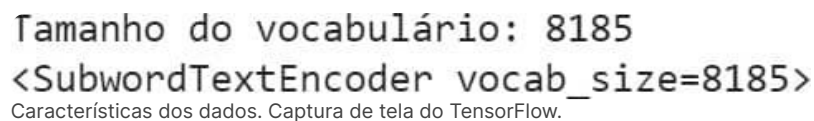
RNN 3: Compreendendo o conceito o vocabulário

No código a seguir, mostramos como obter informações sobre o vocabulário de palavras.

```
plain-text

encoder = info.features['text'].encoder
print('Tamanho do vocabulário: {}'.format(encoder.vocab_size))
print(encoder)
```

A saída do programa pode ser vista na imagem a seguir.



Tamanho do vocabulário: 8185
<SubwordTextEncoder vocab_size=8185>

Características dos dados. Captura de tela do TensorFlow.

RNN 4: Preparar o Dataset

No código a seguir, mostramos como dividir o conjunto de dados para treinamento e teste.

```
plain-text

dataset_treino, dataset_teste = dataset['train'], dataset['test']
```

No código a seguir, mostramos como fazer o embaralhamento dos dados para evitar qualquer viés.

```
plain-text

BUFFER_SIZE = 10000
BATCH_SIZE = 64
dataset_treino = dataset_treino.shuffle(BUFFER_SIZE)
dataset_treino = dataset_treino.padded_batch(BATCH_SIZE)
dataset_teste = dataset_teste.padded_batch(BATCH_SIZE)
```

Nas linhas 1 e 2, estabelecemos os tamanhos do buffer para que possamos embaralhar o conjunto de dados de treino e limitamos o comprimento de sequência a 64.

Na linha 3, fazemos o embaralhamento dos dados do dataset de treino.

Nas linhas 4 e 5, aplicamos o método `padded_batch` nos datasets de treino e teste, que é um redimensionamento dos conjuntos de dados de treino e de teste.

RNN 5: Construindo a rede neural recorrente (RNN)

No código a seguir, mostramos como criar a RNN (YALÇIN, 2021, p. 176).

```
plain-text

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Dense,

                                     Embedding,
                                     Bidirectional,
                                     Dropout,
                                     LSTM)

from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.optimizers import Adam
```

No código, usamos a API Keras Sequential API para construir os modelos.

Nas linhas 2 a 6, configuramos as camadas da RNN com as opções dense, embedding, bidirecional, LSTM e dropout.

Na linha 7, importamos o binary crossentropy como nossa função de perda, pois usamos a classificação binária para prever se um comentário é negativo ou positivo.

Finalmente, na linha 8, usamos o otimizador Adam para otimizar nossos pesos com retropropagação.

RNN 6: Criação do modelo e preenchimento das camadas

No código a seguir, mostramos como criar o modelo e preenchê-lo com camadas e configurá-las. O modelo tem como características: uma camada de codificação, duas camadas LSTM envoltas em camadas bidirecionais, duas camadas densas e uma camada de exclusão (YALÇIN, 2021, p. 177).

```
plain-text

model = Sequential([Embedding(encoder.vocab_size, 64),
                    Bidirectional(LSTM(64, return_sequences=True)),
                    Bidirectional(LSTM(32)),
                    Dense(64, activation='relu'),
                    Dropout(0.5),
                    Dense(1)])

model.summary()
```

Na linha 1, a camada de incorporação converte as sequências de índices de palavras em sequências de vetores. Uma camada de incorporação armazena um vetor por palavra.

Nas linhas 2 e 3, temos camadas LSTM envolvidas em camadas bidirecionais. As camadas bidirecionais propagam a entrada e para trás por meio das camadas LSTM e, em seguida, concatenam a saída, o que é útil para aprender dependências de longo alcance.

Na linha 4, temos a camada densa com 64 neurônios para aumentar a complexidade.

Na linha 5, temos uma camada de dropout para combater o overfitting.

Finalmente, na linha 6, adicionamos uma camada densa final para fazer uma previsão binária.

Podemos ver o resultado da execução do código na imagem a seguir.

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, None, 64)	523840
bidirectional_2 (Bidirection	(None, None, 128)	66048
bidirectional_3 (Bidirection	(None, 64)	41216
dense_4 (Dense)	(None, 64)	4160
dropout_2 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 1)	65

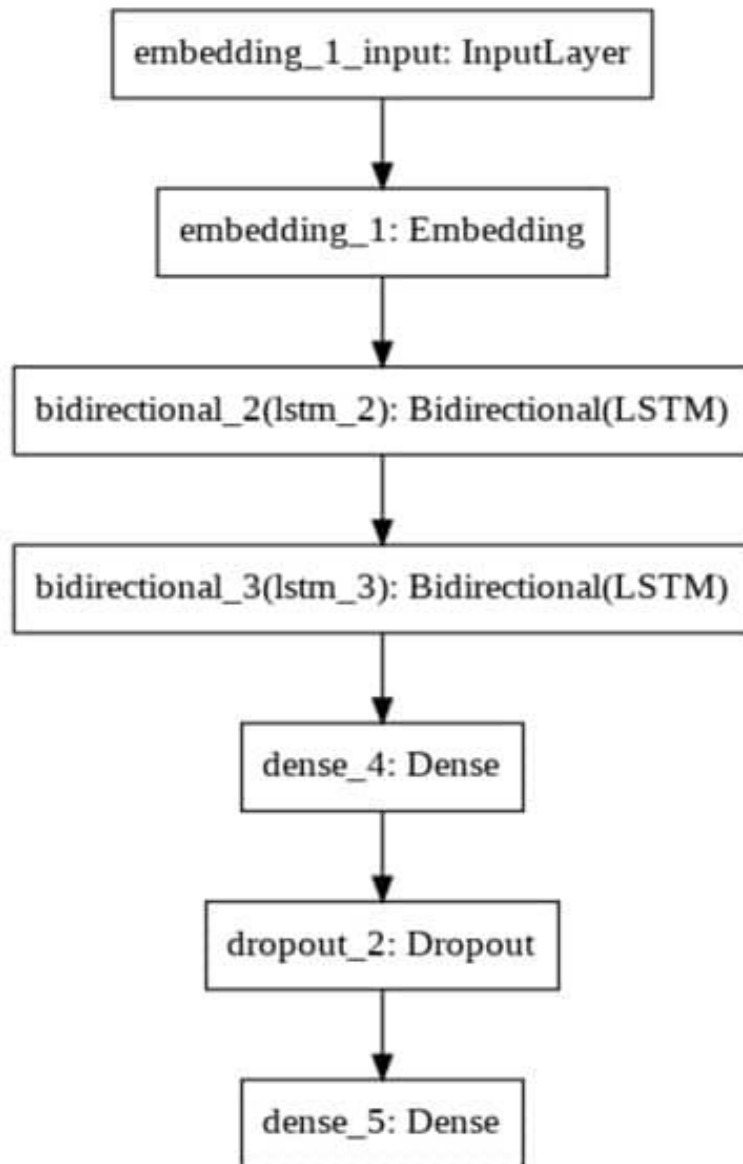
=====
Total params: 635,329
Trainable params: 635,329
Non-trainable params: 0

Resumo da configuração das camadas. Captura de tela do TensorFlow.

O TensorFlow oferece uma forma bastante interessante de visualizar a RNN. Apresentamos o código de como fazer a visualização:

```
plain-text  
tf.keras.utils.plot_model(model)
```

Podemos ver o resultado do código na imagem a seguir.



Visualização do RNN. Captura de tela do TensorFlow.

RNN 7: Compilar e ajustar o modelo

Apresentamos o código para compilar o modelo (YALÇIN, 2021, p. 179):

```
plain-text
model.compile(loss=BinaryCrossentropy(from_logits=True),
              optimizer=Adam(1e-4),
              metrics=['accuracy'])
```

Em especial, observe que, na linha 2, usamos o algoritmo otimizador Adam.

A seguir, apresentamos o código para fazer o ajuste do modelo.

```
plain-text

history = model.fit(dataset_treino,
                    epochs=3,
                    validation_data=dataset_teste,
                    validation_steps=30)
```

Usamos o dataset_treino para o processo de aprendizado da RNN e o dataset_teste para validar o resultado.

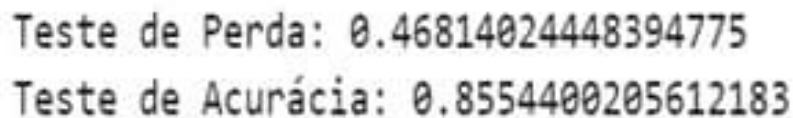
RNN 8: Avaliação do modelo

Apresentamos o código para avaliar o modelo:

```
plain-text

test_loss, test_acc = model.evaluate(dataset_teste)
print('Teste de Perda: {}'.format(test_loss))
print('Teste de Acurácia: {}'.format(test_acc))
```

Na imagem a seguir, o resultado da execução do código:



The image shows a screenshot of a terminal window with a black background and green text. It displays the results of the evaluation code: 'Teste de Perda: 0.46814024448394775' and 'Teste de Acurácia: 0.8554400205612183'.

Perda e precisão do modelo. Captura de tela do TensorFlow.

RNN 9: Predições

Agora que a nossa RNN está treinada, vamos fazer novas predições de sentimento a partir das análises que ainda não foram examinadas por ela.

Precisamos construir uma função que receba um comentário, faça o padding, um ajuste do tamanho da entrada, e o codifique.

A seguir, apresentamos o código para fazer o padding do comentário.

```
plain-text

def padding_comentario(comentario_codificado, tamanho_padding):
    zeros = [0] * (tamanho_padding -
                    len(comentario_codificado))comentario_codificado.extend(zeros)
    return comentario_codificado
```

Agora, apresentamos a função que faz a codificação do comentário.

plain-text

```
def comentario_encoder(comentario):  
    comentario_codificado =  
        padding_comentario(encoder.encode(comentario), 64)  
    comentario_codificado =  
        tf.cast(comentario_codificado, tf.float32)  
    return tf.expand_dims(comentario_codificado, 0)
```

Perceba que, na linha 2, a função faz uma chamada para `padding_comentario` para ajustar o comprimento da entrada.

Agora, vamos entrar com o nosso comentário. O filme que escolhemos para comentar é Matrix. Como os comentários no IMDb estão todos em inglês, o nosso comentário também está em inglês, como podemos ver a seguir:

plain-text

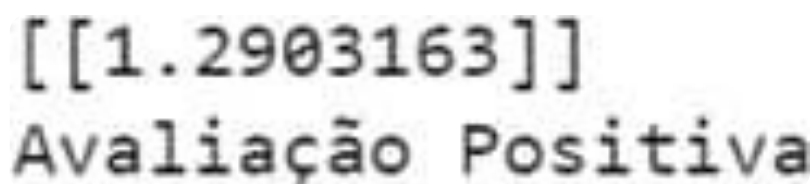
```
comentario_matrix = 'The movie is very good. It made me wonder how advances in machine  
learning will influence the lives of humanity. I want to watch other films of the same  
type.'
```

Finalmente, podemos verificar como a nossa RNN vai classificar o comentário. O código de predição está no código a seguir.

plain-text

```
r = model.predict(comentario_encoder(comentario_matrix))  
print(r)  
resultado = lambda x: 'Avaliação Positiva' if x>0.5 else 'Avaliação Negativa'  
print(resultado(r[0]))
```

Podemos ver o resultado da classificação na imagem.



Resultado da predição. Captura de tela do TensorFlow.

Quando o valor do modelo for maior do que 0.5, o comentário é classificado positivo; caso contrário, é classificado como negativo. No caso do nosso exemplo, o comentário teve uma avaliação positiva.

Simulação com AWS SageMaker

O Amazon SageMaker é um serviço de aprendizado de máquina disponibilizado pela Amazon. A ideia do serviço é oferecer um ambiente integrado de desenvolvimento, testes e implantação hospedado na nuvem. É um serviço pago em que o treinamento e a hospedagem são cobrados por minutos de uso, mas existe uma opção com limitação de recursos que pode ser usada gratuitamente.

Depois de criar uma conta em que será necessário informar dados pessoais, o desenvolvedor terá acesso ao ambiente de desenvolvimento integrado SageMaker Studio. Nele, é possível implementar as diversas etapas de um projeto de machine learning, tais como:

- Preparar os dados;
- Construir modelos;
- Treinar e ajustar modelos;
- Fazer o deploy do projeto.

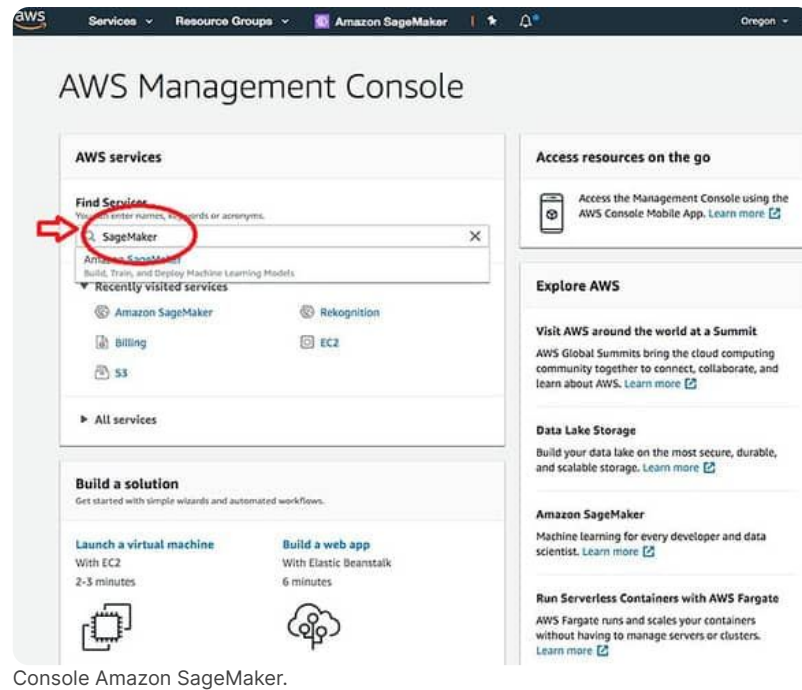
No *site* oficial do SageMaker, também são apresentados interessantes casos de uso que cobrem os tópicos:

- Manutenção preditiva;
- Visão computacional;
- Direção autônoma;
- Detecção de fraudes;
- Previsão de risco de crédito;
- Extração e análise dados de documentos;
- Previsão de rotatividade de clientes;
- Previsão sob demanda;
- Recomendações personalizadas.

A Amazon também apresenta no site um tutorial para criar, treinar e implantar um modelo de aprendizado de máquina. A sequência para criar um projeto completo possui sete etapas, que são:

Etapas 1

Acessar o console do Amazon SageMaker. É por meio desse console que o usuário seleciona o serviço do SageMaker.



Console Amazon SageMaker.

Etapa 2

Criar uma instância de bloco de anotações do Amazon SageMaker.

Etapa 3

Preparar os dados por meio do uso do bloco de anotações do Amazon SageMaker para pré-processar os dados necessários para treinar o modelo de machine learning.

Etapa 4

Treinar o modelo de machine learning a partir do conjunto de dados de treinamento.

Etapa 5

Implantar o modelo treinado em um endpoint para criar previsões.

Etapa 6

Avaliar o desempenho e a precisão do modelo de machine learning.

Etapa 7

Encerrar o uso dos recursos relacionados ao Amazon SageMaker. Essa etapa merece atenção especial, pois pode haver cobranças, caso não haja encerramento dos recursos.

Simulação de aprendizado profundo

O especialista Sérgio Assunção Monteiro fala sobre os principais tópicos a respeito de simulação de aprendizado profundo abordados neste módulo.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

O TensorFlow é uma biblioteca muito utilizada para desenvolver projetos de aprendizado de máquina. Ele utiliza algumas APIs com objetivos específicos que facilitam o trabalho de desenvolvimento. Em relação às APIs do TensorFlow, selecione a opção correta.

A

A API `tf.data` é usada para manipular os dados dos gradientes de modo a otimizar o desempenho da rede.

B

A API `tf.keras` é aplicada nos modelos de aprendizado profundo para criar e treinar as redes.

C

A API `tf.estimators` é utilizada para fazer estimações sobre os valores dos gradientes de modo a antecipar possíveis problemas e, portanto, viabilizar que o algoritmo faça ajustes.

D

A API `Distribution Strategy` é usada para determinar a política que a rede vai aplicar no processo de treinamento.

E

A API `tf.tdfs` é usada para fazer a leitura de diferentes formatos e realizar transformações complexas.



A alternativa B está correta.

A API `tf.keras` está entre as principais API do TensorFlow e é utilizada para criar e treinar modelos de aprendizado profundo.

Questão 2

O TensorFlow é, atualmente, a biblioteca mais conhecida e usada para desenvolver aplicações de aprendizado profundo. No entanto, existem outras bibliotecas e frameworks disponíveis para serem utilizados e que, em muitos casos, têm melhor desempenho para aplicações específicas. Em relação às bibliotecas e aos frameworks concorrentes do TensorFlow, selecione a opção correta.

A

Keras é uma biblioteca de rede neural de código aberto escrita em Python que pode ser executada apenas com o TensorFlow.

B

PyTorch é uma biblioteca de código aberto e é considerada a principal concorrente do TensorFlow.

C

MXNet é um framework de aprendizado profundo de código aberto desenvolvido pela Microsoft e, portanto, tem suporte para as linguagens de programação C# e Visual Basic.

D

CNTK é framework de código aberto desenvolvido pelo Facebook e tem suporte pelas linguagens de programação Python, C++, C# e Java.

E

Microsoft Cognitive Toolkit foi desenvolvido pela Microsoft e é uma biblioteca de código proprietário, porém dá suporte para diversas linguagens de programação R, Go, Java Script, Python, C++, C# e Java.



A alternativa B está correta.

O PyTorch é uma biblioteca de rede neural de código aberto desenvolvida e mantida principalmente pelo AI Research Lab do Facebook. Ela é a principal concorrente do TensorFlow.

Considerações finais

No decorrer do conteúdo, apresentamos aspectos teóricos e práticos sobre redes neurais e aprendizado de máquina, como arquitetura, problemas relacionados aos cálculos dos gradientes, otimização de recompensas, aprendizados por diferenças temporais e Q-learning, além de abordarmos as redes neurais de convolução (CNNs) e recorrentes (RNNs). Vimos, ainda, a aplicação de CNNs e RNNs para classificação de imagens e processamento de linguagem natural, respectivamente, com o desenvolvimento de uma aplicação utilizando a linguagem de programação Python.

A compreensão dos aspectos teóricos nos ajuda a extrair melhores resultados quando partimos para o desenvolvimento de aplicações práticas, como o TensorFlow, por exemplo. É interessante perceber a evolução dos recursos e ambientes disponíveis para desenvolver aplicações de aprendizado de máquina que se integram a outros componentes de sistemas complexos.

Os benefícios da compreensão e aplicação do aprendizado de máquina ajuda a potencializar resultados dos sistemas com a melhoria da qualidade relacionada à precisão e com tempos de resposta competitivos. A escolha da linguagem Python para desenvolver os projetos de aprendizado de máquina é natural devido à vasta disponibilidade de documentação e de pacotes que facilitam o desenvolvimento, porém essas aplicações podem ser desenvolvidas em outras linguagens de programação, como Java, C# e R, por exemplo.

Podcast

Neste podcast, o especialista Sérgio Assunção Monteiro abordará a importância do aprendizado profundo, o emprego em diversas situações práticas e quais os desafios no estudo do aprendizado profundo.



Conteúdo interativo

Acesse a versão digital para ouvir o áudio.

Explore+

- Acesse o *site* do TensorFlow para aprender mais detalhes sobre instalação, configuração e desenvolvimento em diferentes plataformas, além de ver mais exemplos de projetos de aprendizado profundo.
- Explore o *site* de documentação do TensorFlow para obter mais detalhes sobre a biblioteca e praticar mais exemplos.
- Acesse o *site* do Python e aprenda mais sobre essa linguagem de programação que será muito útil, para que você desenvolva projetos de aprendizado profundo.
- Visite a página do Amazon SageMaker para conhecer os recursos disponibilizados no serviço.
- A Amazon apresenta um tutorial para criar, treinar e implantar um modelo de aprendizado de máquina. Pesquise no *site* e crie, treine e implante um modelo de machine learning com o Amazon SageMaker.

- Conheça mais sobre as APIs do TensorFlow, pesquisando API Documentation na página do TensorFlow.
- O Colaboratory ou Colab permite escrever código Python no seu navegador. Acesse o Google Colab para executar outros códigos, além dos exemplos do tema.
- Para ver outros tipos de instalação do TensorFlow além do Google Colab, veja Instalar o TensorFlow com PIP no *site* do TensorFlow.

Referências

BENGIO, Y.; SIMARD, P.; FRASCONI, P. **Learning long-term dependencies with gradient descent is difficult**. *In: IEEE Transactions on neural networks*, vol. 5, n. 2, p. 157–166, mar. 1994. Consultado em meio eletrônico em: 8 fev. 2021.

CAMACHO, C. **Convolutional Neural Networks**. 2018. Consultado em meio eletrônico em: 8 fev. 2021.

HAYKIN, S. **Neural networks and learning machines**. New Jersey: Pearson Education, 2008.

LUGER, G. F. **Inteligência Artificial**. 6. ed. São Paulo: Pearson Education do Brasil, 2013.

OLAH, Christopher. **Understanding LSTM Networks**. 2015. Consultado em meio eletrônico em: 8 fev. 2021.

PAN, S. J.; YANG, Q. **A survey on transfer learning**. *In: IEEE Transactions on Knowledge and Data Engineering*, vol. 22, n. 10, p.1345–1359, 2010.

SOCHER, R.; PERELYGIN, A.; WU, J. Y.; CHUANG, J. *et al.* **Recursive deep models for semantic compositionality over a sentiment treebank**. *In: Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, p. 1631–1642, Seattle, Washington, USA, 18-21 Oct. 2013.

WATKINS, C. **Learning from delayed rewards**. 1989. Tese (Doutorado) – Universidade de Cambridge, Cambridge, Inglaterra.

YALÇIN, O. G. **Applied Neural Networks with TensorFlow 2: API Oriented Deep Learning with Python**. Nova York: Apress, 2020.