FERRAMENTAS UTILIZADAS

Python

https://www.python.org/

TensorFlow

https://www.tensorflow.org/

Numpy

https://numpy.org/

Pandas

https://pandas.pydata.org/

Scikit Learn

https://scikit-learn.org/

--//--

Estes tópicos foram replicados do livro REDES NEURAIS ARTIFICIAIS – Fernando Feltrin.

--//--

TEORIA E PRÁTICA EM CIÊNCIA DE DADOS

O que são Redes Neurais Artificiais

Desde os primeiros estudos por parte biológico e psicológico dos mecanismos que temos de aprendizado, descobrindo nossa própria biologia (no sentido literal da palavra),

criamos modelos lógicos estruturados buscando abstrair tais mecanismos para que pudéssemos fazer ciência sobre os mesmos.

À medida que a neurociência evoluía, assim como áreas abstratas como psicologia também, pudemos cada vez melhor entender o funcionamento de nosso cérebro, desde os mecanismos físicos e biológicos até os que nos tornam autoconscientes e cientes de como aprendemos a medida que experenciamos esse mundo.

Não por acaso, as primeiras tentativas de abstrairmos tais mecanismos de forma lógico-computacional se deram literalmente, recriando tais estruturas anatômicas e mecanismos de aprendizado de forma artificial, por meio das chamadas redes neurais artificiais, com seu sistema de neuronal sináptico.

Existe uma grande variedade de tipos/modelos de redes neurais artificiais de acordo com sua estrutura e propósito, em comum a todas elas estão por exemplo conceitos de perceptrons, onde literalmente criamos neurônios artificiais e interação entre os mesmos para que possam encontrar padrões em dados, usando de poder computacional como vantagem no que diz respeito ao número de informações a serem processadas e o tempo de processamento muito superior aos limites humanos.

Dessa forma, partindo de preceitos de problemas computacionais a serem resolvidos (ou problemas reais abstraídos de forma computacional), temos modelos de redes neurais artificiais que conseguem ler e interpretar dados a partir de praticamente todo tipo de arquivo (inclusive imagens e sons), realizar classificações para os mais diversos fins, assim como realizar previsões com base em histórico de dados, simular algoritmos genéticos, também realizar processamento de linguagem natural, geração de novos dados a partir de amostras, reconhecimento por conversão, simulação de sentidos biológicos como visão, audição e outros com base em leitura de sensores, aprendizado com base em experiência, tomadas de decisão com base em memória, etc... Uma verdadeira infinidade de possibilidades.

Raciocine que nesse processo de criarmos arquiteturas de redes neurais artificiais, onde podemos simular praticamente qualquer tipo de inteligência de um ser vivo biológico de menor complexidade. Ainda assim estamos engatinhando nessa área, mesmo com décadas de evolução, porém procure ver à frente, todo o potencial que temos para as próximas décadas à medida que aperfeiçoamos nossos modelos.

Teoria Sobre Redes Neurais Artificiais

Um dos métodos de processamento de dados que iremos usar no decorrer de todos exemplos deste livro, uma vez que sua abordagem será prática, é o de redes neurais artificiais. Como o nome já sugere, estamos criando um modelo computacional para abstrair uma estrutura anatômica real, mais especificamente um ou mais neurônios, células de nosso sistema nervoso central e suas interconexões.

Nosso sistema nervoso, a nível celular, é composto por neurônios e suas conexões chamadas sinapses. Basicamente um neurônio é uma estrutura que a partir de reações bioquímicas que o estimular geram um potencial elétrico para ser transmitido para suas conexões, de forma a criar padrões de conexão entre os mesmos para ativar algum processo de

outro subsistema ou tecido ou simplesmente guardar padrões de sinapses no que convencionalmente entendemos como nossa memória.

Na eletrônica usamos de um princípio parecido quando temos componentes eletrônicos que se comunicam em circuitos por meio de cargas de pulso elétrico gerado, convertido e propagado de forma controlada e com diferentes intensidades a fim de desencadear algum processo ou ação.

Na computação digital propriamente dita temos modelos que a nível de linguagem de máquina (ou próxima a ela) realizam chaveamento traduzindo informações de código binário para pulso ou ausência de pulso elétrico sobre portas lógicas para ativação das mesmas dentro de um circuito, independentemente de sua função e robustez.

Sendo assim, não diferentemente de outras áreas, aqui criaremos modelos computacionais apenas como uma forma de abstrair e simplificar a codificação dos mesmos, uma vez que estaremos criando estruturas puramente lógicas onde a partir delas teremos condições de interpretar dados de entrada e saídas de ativação, além de realizar aprendizagem de máquina por meio de reconhecimento de padrões e até mesmo desenvolver mecanismos de interação homem-máquina ou de automação.

A nível de programação, um dos primeiros conceitos que precisamos entender é o de um Perceptron. Nome este para representar um neurônio artificial, estrutura que trabalhará como modelo lógico para interpretar dados matemáticos de entrada, pesos aplicados sobre os mesmos e funções de ativação para ao final do processo, assim como em um neurônio real, podermos saber se o mesmo é ativado ou não em meio a um processo.

Aproveitando o tópico, vamos introduzir alguns conceitos que serão trabalhados posteriormente de forma prática, mas que por hora não nos aprofundaremos, apenas começaremos a construir aos poucos uma bagagem de conhecimento sobre tais pontos.

Quando estamos falando em redes neurais artificiais, temos de ter de forma clara que este conceito é usado quando estamos falando de toda uma classe de algoritmos usados para encontrar e processar padrões complexos a partir de bases de dados usando estruturas lógicas chamadas perceptrons. Uma rede neural possui uma estrutura básica de entradas, camadas de processamento, funções de ativação e saídas.

De forma geral é composta de neurônios artificiais, estruturas que podem ser alimentadas pelo usuário ou retroalimentadas dentro da rede por dados de suas conexões com outros neurônios, aplicar alguma função sobre esses dados e por fim gerar saídas que por sua vez podem representar uma tomada de decisão, uma classificação, uma ativação ou desativação, etc...

Outro ponto fundamental é entender que assim como eu uma rede neural biológica, em uma rede neural artificial haverão meios de comunicação entre esses neurônios e podemos inclusive usar esses processos, chamados sinapses, para atribuir valores e funções sobre as mesmas. Uma sinapse normalmente possui uma sub estrutura lógica chamada Bias, onde podem ser realizadas alterações intermediárias aos neurônios num processo que posteriormente será parte essencial do processo de aprendizado de máquina já que é nessa “camada” que ocorrem atualizações de valores para aprendizado da rede.

Em meio a neurônios artificiais e suas conexões estaremos os separando virtualmente por camadas de processamento, dependendo da aplicação da rede essa pode ter apenas uma camada, mais de uma camada ou até mesmo camadas sobre camadas (deep learning). Em suma um modelo básico de rede neural conterá uma camada de entrada, que pode ser alimentada manualmente pelo usuário ou a partir de uma base de dados.

Em alguns modelos haverão o que são chamadas camadas ocultas, que são intermediárias, aplicando funções de ativação ou funcionando como uma espécie de filtros sobre os dados processados. Por fim desde os moldes mais básicos até os mais avançados sempre haverá uma ou mais camadas de saída, que representam o fim do processamento dos dados dentro da rede após aplicação de todas suas funções.

A interação entre camadas de neurônios artificiais normalmente é definida na literatura como o uso de pesos e função soma. Raciocine que a estrutura mais básica terá sempre esta característica, um neurônio de entrada tem um valor atribuído a si mesmo, em sua sinapse, em sua conexão com o neurônio subsequente haverá um peso de valor gerado aleatoriamente e aplicado sobre o valor inicial desse neurônio em forma de multiplicação. O resultado dessa multiplicação, do valor inicial do neurônio pelo seu peso é o valor que irá definir o valor do próximo neurônio, ou em outras palavras, do(s) que estiver(em) na próxima camada. Seguindo esse raciocínio lógico, quando houverem múltiplos neurônios por camada além dessa fase mencionada anteriormente haverá a soma dessas multiplicações.

1 neurônio Z = entradas x pesos

n neurônios Z = (entrada1 x peso1) + (entrada2 + peso2) + etc…

Por fim um dos conceitos introdutórios que é interessante abordarmos agora é o de que mesmo nos modelos mais básicos de redes neurais haverão funções de ativação. Ao final do processamento das camadas é executada uma última função, chamada de função de ativação, onde o valor final poderá definir uma tomada de decisão, uma ativação (ou desativação) ou classificação. Esta etapa pode estar ligada a decisão de problemas lineares (0 ou 1, True ou False, Ligado ou Desligado) ou não lineares (probabilidade de 0 ou probabilidade de 1). Posteriormente ainda trabalharemos com conceitos mais complexos de funções de ativação.

Linear - ReLU (Rectified Linear Units) - Onde se X for maior que 0 é feita a ativação do neurônio, caso contrário, não. Também é bastante comum se usar a Step Function (Função Degrau) onde é definido um valor como parâmetro e se o valor de X for igual ou maior a esse valor é feita a ativação do neurônio, caso contrário, não.

Não Linear - Sigmoid - Probabilidade de ativação ou de classificação de acordo com a proximidade do valor de X a 0 ou a 1. Apresenta e considera os valores intermediários entre 0 e 1 de forma probabilística.

Também estaremos trabalhando com outras formas de processamento quando estivermos realizando o treinamento supervisionado de nossa rede neural artificial. Raciocine que os tópicos apresentados anteriormente apenas se trata da estrutura lógica ao qual sempre iremos trabalhar, o processo de aprendizado de máquina envolverá mais etapas onde

realizaremos diferentes processos em cima de nossa rede para que ela “aprenda” os padrões os quais queremos encontrar em nossos dados e gerar saídas a partir dos mesmos.

O que é Aprendizado de Máquina

Entendendo de forma simples, e ao mesmo tempo separando o joio do trigo, precisamos de fato contextualizar o que é uma rede neural artificial e como se dá o mecanismo de aprendizado de máquina. Raciocine que, como mencionado anteriormente, temos um problema computacional a ser resolvido. Supondo que o mesmo é um problema de classificação, onde temos amostras de dois ou mais tipos de objetos e precisamos os classificar conforme seu tipo.

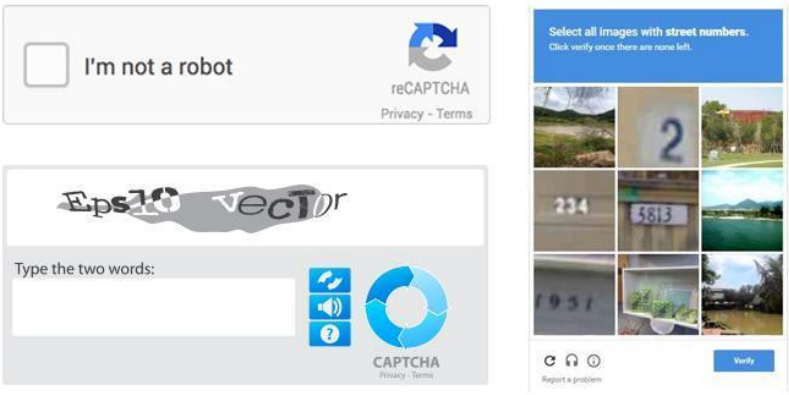
Existe literalmente uma infinidade de algoritmos que poderiam realizar essa tarefa, mas o grande pulo do gato aqui é que por meio de um modelo de rede neural, podemos treinar a mesma para que reconheça quais são os objetos, quais são suas características relevantes e assim treinar e testar a mesma para que com algumas configurações consiga realizar essa tarefa de forma rápida e precisa, além de que uma vez aprendida essa classificação, essa rede neural pode realizar novas classificações com base nos seus padrões aprendidos previamente.

Para esse processo estaremos criando toda uma estrutura de código onde a rede terá dados de entrada, oriundos das amostras a serem classificadas, processamento via camadas de neurônios artificiais encontrando padrões e aplicando funções matemáticas sobre os dados, para ao final do processo separar tais dados das amostras conforme o seu tipo.

Neste processo existe o que é chamado de aprendizado de máquina supervisionado, onde literalmente temos que mostrar o caminho para a rede, mostrar o que é o resultado final correto para que ela aprenda os padrões intermediários que levaram até aquela resolução. Assim como também teremos modelos de aprendizado não supervisionado, onde a rede com base em alguns algoritmos conseguirá sozinha identificar e reconhecer os padrões necessários.

Aprendizado de Máquina

Quando estamos falando em Ciência e Dados, mais especificamente de Aprendizado de Máquina, é interessante começarmos a de fato entender o que essas nomenclaturas significam, para desmistificarmos muita informação errônea que existe a respeito desse termo. Na computação, apesar do nome “máquina”, basicamente estaremos criando modelos lógicos onde se é possível simular uma inteligência computacional, de forma que nosso software possa ser alimentado com dados de entrada e a partir destes, reconhecer padrões, realizar diversos tipos de tarefas sobre estes, aprender padrões de experiência e extrair informações ou desencadear tomadas de decisões a partir dos mesmos. Sendo assim, ao final temos um algoritmo funcional que foi treinado e aprendeu a atingir um objetivo com base nos dados ao qual foi alimentado.



Você pode não ter percebido, mas uma das primeiras mudanças na web quando começamos a ter sites dinâmicos foi o aparecimento de ferramentas de validação como os famosos captcha. Estas aplicações por sua vez tem uma ideia genial por trás, a de que pedindo ao usuário que realize simples testes para provar que ele “não é um robô”, ele está literalmente treinando um robô a reconhecer padrões, criando dessa forma uma espécie de inteligência artificial. Toda vez que você responde um captcha, identifica ou sinaliza um objeto ou uma pessoa em uma imagem, você está colaborando para o aprendizado de uma máquina.

Quando criamos estes tipos de modelos de algoritmos para análise e interpretação de dados (independe do tipo), inicialmente os fazemos de forma manual, codificando um modelo a ser usado e reutilizado posteriormente para problemas computacionais de mesmo tipo (identificação, classificação, regressão, previsão, etc...).

Tipos de Aprendizado de Máquina

Supervisionado

Basicamente esse é o tipo de aprendizado de máquina mais comum, onde temos um determinado problema computacional a ser resolvido (independentemente do tipo), sabemos de forma lógica como solucionaremos tal problema, logo, treinaremos uma rede neural para que identifique qual a maneira melhor, mais eficiente ou mais correta de atingir o seu objetivo.

Em outras palavras, é o tipo de aprendizado onde teremos de criar e treinar toda uma estrutura de rede neural já lhe mostrando o caminho a ser seguido, a rede neural por sua vez, apenas irá reconhecer quais foram os padrões adotados para que aquele objetivo fosse alcançado com melhor margem de acerto ou precisão. Modelo muito utilizado em problemas de classificação, regressão e previsão a partir de uma base de dados.

Como exemplo, imagine uma rede neural artificial que será treinada para avaliar o perfil de crédito de um correntista de um banco, a fim de conceder ou negar um empréstimo. A rede neural que irá classificar o novo cliente como apto ou não apto a receber crédito, fará essa escolha com base em milhares de outros clientes com as mesmas características, de forma a encontrar padrões de quais são bons ou maus pagadores. Dessa forma, o novo cliente irá fornecer dados que serão filtrados conforme os padrões da base de dados dos clientes antigos

para que se possa presumir se ele por sua vez será um bom cliente (se o mesmo possui as mesmas características de todos os outros bons pagadores), obtendo seu direito ao crédito.

Não Supervisionado

Sabendo que é possível mostrar o caminho para uma rede neural, podemos presumir que não mostrar o caminho significa um aprendizado não supervisionado, correto? Não necessariamente! Uma rede neural realizando processamento de forma não supervisionada até pode encontrar padrões, porém sem saber qual o objetivo, a mesma ficará estagnada após algumas camadas de processamento ou terá baixíssima eficiência já que ela não possui capacidade de deduzir se seus padrões estão certos ou errados para aquele fim.

Porém, esse conceito pode justamente ser utilizado para a resolução de outros tipos de problemas computacionais. Dessa forma, modelos de redes neurais artificiais para agrupamento tendem a funcionar muito bem e de forma não supervisionada. Raciocine que neste tipo de modelo em particular estamos apenas separando amostras conforme suas características, sem um objetivo em comum a não ser o de simplesmente separar tais amostras quanto ao seu tipo.

Como exemplo, imagine que você tem uma caixa com 3 tipos de frutas diferentes, a rede neural irá realizar o mapeamento de todas essas frutas, reconhecer todas suas características e principalmente quais características as fazem diferentes umas das outras para por fim realizar a separação das mesmas quanto ao seu tipo, sem a necessidade de um supervisor.

Por Reforço

Será o tipo de aprendizado onde um determinado agente busca alcançar um determinado objetivo, mas por sua vez, ele aprenderá como alcançar tal objetivo com base na leitura e interpretação do ambiente ao seu redor, interagindo com cada ponto desse ambiente tentando alcançar seu objetivo no que chamamos de tentativa e erro. Toda vez que o mesmo realizar uma ação coerente ou correta dentro daquele contexto, será recompensado de forma que grave aquela informação como algo válido a ser repetido, assim como toda vez que o mesmo realizar uma ação errada para aquele contexto, receberá uma penalidade para que aprenda que aquela forma ou meio usada na tentativa não está correta, devendo ser evitada.

Como exemplo, imagine um carro autônomo, que tem a missão de sair de um ponto A até um ponto B, estacionando o carro, porém o mesmo parte apenas do pressuposto de que pode andar para frente e para trás, assim como dobrar para esquerda ou direita. Nas primeiras tentativas o mesmo irá cometer uma série de erros até que começará a identificar quais ações foram tomadas nas tentativas que deram certo e o mesmo avançou em seu caminho. Em um processo que as vezes, dependendo da complexidade, pode levar literalmente milhões de tentativas até que se atinja o objetivo, a inteligência artificial irá aprender quais foram os melhores padrões que obtiveram os melhores resultados para replicá-los conforme a necessidade. Uma vez aprendido o caminho nesse ambiente, a IA deste veículo autônomo fará esse processo automaticamente com uma margem de acertos quase perfeita, assim como terá memorizado como foram realizadas suas ações para que possam ser replicadas.

Aprendizagem por reforço natural vs artificial

Enquanto não colocamos em prática esses conceitos, ou ao menos não visualizamos os mesmos em forma de código, tudo pode estar parecendo um tanto quanto abstrato, porém, talvez você já tenha identificado que o que estamos tratando aqui como um tipo de aprendizagem, é justamente o tipo de aprendizagem mais básica e orgânica que indiretamente conhecemos por toda nossa vida.

Todos os mecanismos cognitivos de aprendizado que possuímos em nós é, de certa forma, um algoritmo de aprendizagem por reforço. Raciocine que a maior parte das coisas que aprendemos ao longo de nossa vida foi visualizando e identificando padrões e os imitando, ou instruções que seguimos passo-a-passo e as repetimos inúmeras vezes para criar uma memória cognitiva e muscular sobre tal processo ou com base em nossas experiências, diretamente ou indiretamente em cima de tentativa e erro.

Apenas exemplificando de forma simples, quando aprendemos um instrumento musical, temos uma abordagem teórica e uma prática. A teórica é onde associamos determinados conceitos de forma lógica em sinapses nervosas, enquanto a prática aprendemos que ações realizamos e as repetimos por diversas vezes, ao final de muito estudo e prática dominamos um determinado instrumento musical em sua teoria (como extrair diferentes sons do mesmo formando frases de uma música) e em sua prática (onde e como tocar para extrair tais sons).

Outro exemplo bastante simples de nosso cotidiano, agora focando no sistema de recompensa e penalidade, é quando temos algum animal de estimação e o treinamos para realizar determinadas tarefas simples. Dentre a simplicidade cognitiva de um cachorro podemos por exemplo, ensinar o mesmo que a um determinado comando de voz ou gesto ele se sente, para isto, lhe damos como recompensa algum biscoitinho ou simplesmente um carinho, indiretamente o mesmo está associando que quando ele ouvir aquele comando de voz deve se sentar, já que assim ele foi recompensado. Da mesma forma, quando ele faz uma ação indesejada o chamamos a atenção para que associe que não é para repetir aquela ação naquele contexto.

Sendo assim, podemos presumir que esta mesma lógica de aprendizado pode ser contextualizada para diversos fins, inclusive, treinando uma máquina a agir e se comportar de determinadas maneiras conforme seu contexto, aplicação ou finalidade, simulando de forma artificial (simples, porém convincente) uma inteligência.

No futuro, talvez em um futuro próximo, teremos desenvolvido e evoluído nossos modelos de redes neurais artificiais intuitivas de forma a não somente simular com perfeição os mecanismos de inteligência quanto os de comportamento humano, tornando assim uma máquina indistinguível de um ser humano em sua aplicação. Vide teste de Turing.

Aprendizado por Reforço em Detalhes

Avançando um pouco em nossos conceitos, se entendidos os tópicos anteriores, podemos começar a nos focar no que será nosso objetivo ao final deste livro, os tipos de redes neurais artificiais que de uma forma mais elaborada simularão uma inteligência artificial.

Ainda falando diretamente sobre modelos de redes neurais artificiais, temos que começar a entender que, como dito anteriormente, existirão modelos específicos para cada tipo de problema computacional, e repare que até agora, os modelos citados conseguem ser modelados de forma parecida a uma rede neural biológica, para abstrair problemas reais de forma computacional, realizando processamento supervisionado ou não, para que se encontrem padrões que podem gerar diferentes tipos de saídas conforma o problema em questão. Porém, note que tudo o que foi falado até então, conota um modelo de rede neural artificial estático, no sentido de que você cria o modelo, o alimenta, treina e testa e gera previsões sobre si mas ao final do processo ele é encerrado assim como qualquer outro tipo de programa.

Já no âmbito da inteligência artificial de verdade, estaremos vendo modelos de redes neurais que irão além disso, uma vez que além de todas características herdadas dos outros modelos, as mesmas terão execução contínua inclusive com tomadas de decisões.

Os modelos de redes neurais artificiais intuitivas serão aqueles que, em sua fase de aprendizado de máquina, estarão simulando uma memória de curta ou longa duração (igual nossa memória como pessoa), assim como estarão em tempo real realizando a leitura do ambiente ao qual estão inseridos, usando esses dados como dados de entrada que retroalimentam a rede neural continuamente, e por fim, com base no chamado aprendizado por reforço, a mesma será capaz de tomar decisões arbitrárias simulando de fato uma inteligência artificial.

Diferentemente de um modelo de rede neural artificial “tradicional” onde a fase supervisionada se dá por literalmente programar do início ao fim de um processo, uma rede neural artificial “intuitiva” funciona de forma autosupervisionada, onde ela realizará determinadas ações e, quando a experiência for positiva, a mesma será recompensada, assim como quando a experiência for negativa, penalizada.

Existirão, é claro, uma série de algoritmos que veremos na sequência onde estaremos criando os mecanismos de recompensa e penalização, e a forma como a rede neural aprende diante dessas circunstâncias. Por hora, raciocine que nossa rede neural, em sua inteligência simulada, será assim como um bebê descobrindo o mundo, com muita tentativa, erro e observações, naturalmente a mesma aprenderá de forma contínua e com base em sua experiência será capaz de tomar decisões cada vez mais corretas para o seu propósito final.

Apenas realizando um adendo, não confundir esse modelo de rede neural artificial com os chamados algoritmos genéticos, estes por sua vez, adotam o conceito de evolução baseada em gerações de processamento, mas não trabalham de forma autônoma e contínua. Veremos o referencial teórico para esse modelo com mais detalhes nos capítulos subsequentes.

Características Específicas do Aprendizado por Reforço

À medida que vamos construindo nossa bagagem de conhecimento sobre o assunto, vamos cada vez mais contextualizando ou direcionando nosso foco para as redes neurais

artificiais intuitivas. Nesse processo será fundamental começarmos a entender outros conceitos das mesmas, entre eles, o de aprendizado por reforço utilizado neste tipo de rede neural.

Como mencionado anteriormente de forma rápida, o chamado aprendizado por reforço implicitamente é a abstração da forma cognitiva de como nós seres vivos aprendemos de acordo com a forma com que experenciamos esse mundo. Por meio de nossos sentidos enxergamos, ouvimos, saboreamos e tocamos deferentes objetos e criando memórias de como interagimos e que experiência tivemos com os mesmos.

Da mesma forma, porém de forma artificial, teremos um agente que simulará um ser vivo ou ao menos um objeto inteligente que por sua vez, via sensores (equivalendo aos seus sentidos), ele fará a leitura do ambiente e com o chamado aprendizado por reforço, experenciará o mesmo realizando ações com os objetos e aprendendo com base em tentativa erro ou acerto.

Para isso, existirão uma série de algoritmos que veremos nos capítulos subsequentes em maiores detalhes que literalmente simularão como esse agente usará de seus recursos de forma inteligente. Desde o mapeamento do ambiente, dos estados e das possíveis ações até a realização das mesmas em busca de objetivos a serem cumpridos gerando recompensas ou penalidades, por fim guardando os melhores estados em uma memória.

Então, apenas definindo algumas características deste tipo de algoritmo, podemos destacar:

- A partir do seu start, toda execução posterior é autônoma, podendo ser em tempo real via backpropagation ou em alguns casos via aprendizado por gerações quando numa tarefa repetitiva.

- Pode ter ou não um objetivo definido, como ir de um ponto A até o B ou aprender a se movimentar dentro de um ambiente fechado, respectivamente.

- Realiza leitura de sensores que simulam sentidos, mapeando o ambiente ao seu redor e o seu estado atual (que tipo de ação está realizando).

- Busca alcançar objetivos da forma mais direta, de menor tempo de execução ou mais eficiente, trabalhando com base em sua memória de experiência, estado e recompensas.

- Usa de redes neurais realizando processamento retroalimentado em tempo real, convertendo as saídas encontradas em tomadas de decisão.

- Possui um tempo considerável de processamento até gerar suas memórias de curta e longa duração, dependendo da complexidade de sua aplicação, precisando que uma determinada ação seja repetida milhares ou milhões de vezes repetidamente até que se encontre e aprenda o padrão correto.

Principais Algoritmos para Inteligência Artificial

Quando estamos falando em ciência de dados, aprendizado de máquina, rede neurais artificias, etc... existe uma grande gama de nomenclatura que em suma faz parte da mesma área/nicho da computação, porém com importantes características que as diferenciam entre si que precisamos entender para acabar separando as coisas.

Machine learning é a nomenclatura usual da área da computação onde se criam algoritmos de redes neurais artificiais abstraindo problemas reais em computacionais, de forma que a "máquina" consiga encontrar padrões e aprender a partir destes. De forma bastante reduzia, podemos entender que as redes neurais artificiais são estruturas de processamento de dados onde processamos amostras como dados de entrada, a rede neural reconhece padrões e aplica funções sobre esses dados, por fim gerando saídas em forma de classificação, previsão, etc..

Para alguns autores e cientistas da área, a chamada data science (ciência de dados em tradução livre) é o nicho onde, com ajuda de redes neurais artificiais, conseguimos processar enormes volumes de dados estatísticos quantitativos ou qualitativos de forma a se extrair informação dos mesmos para se realizem estudos retroativos ou prospectivos. Porém é importante entender que independentemente do tipo de problema computacional, teremos diferentes modelos ou aplicações de redes neurais para que façam o trabalho pesado por nós.

Sendo assim, não é incorreto dizer que toda aplicação que use deste tipo de arquitetura de dados como ferramenta de processamento, é de certa forma inteligência artificial simulada.

Partindo do pressuposto que temos diferentes tipos de redes neurais, moldáveis a praticamente qualquer situação, podemos entender rapidamente os exemplos mais comumente usados.

- Redes neurais artificiais – Modelo de processamento de dados via perceptrons (estrutura de neurônios interconectados que processam dados de entrada gerando saídas) estruturadas para problemas de classificação, regressão e previsão.

- Redes neurais artificiais profundas (deep learning) – Estruturas de rede neural com as mesmas características do exemplo anterior, porém mais robustas, com maior quantidade de neurônios assim como maior número de camadas de neurônios interconectados entre si ativos realizando processamento de dados.

- Redes neurais artificiais convolucionais – Modelo de rede neural com capacidade de usar de imagens como dados de entrada, convertendo as informações de cada pixel da matriz da imagem para arrays (matrizes numéricas), realizando processamento sobre este tipo de dado.

- Redes neurais artificiais recorrentes – Modelos de rede neurais adaptados a realizar o processamento de dados correlacionados com uma série temporal, para que a partir dos mesmos se realizem previsões.

- Redes neurais artificiais para mapas auto-organizáveis – Modelo de rede neural com funcionamento não supervisionado e com processamento via algoritmos de aglomeração, interpretando amostras e classificando as mesmas quanto sua similaridade.

- Redes neurais artificiais em Boltzmann Machines – Modelo não sequencial (sem entradas e saídas definidas) que processa leitura dos perceptrons reforçando conexões entre os mesmos de acordo com sua similaridade, porém podendo também ser adaptadas para sistemas de recomendação

- Redes neurais artificiais com algoritmos genéticos – Modelo baseado em estudo de gerações de processamento sequencial, simulando o mecanismo de evolução reforçando os melhores dados e descartando os piores geração após geração de processamento.

- Redes neurais artificiais adversariais generativas – Modelo capaz de identificar características de similaridade a partir de dados de entrada de imagens, sendo capaz de a partir desses dados gerar novas imagens a partir do zero com as mesmas características.

- Redes neurais artificiais para processamento de linguagem natural – Modelo de nome autoexplicativo, onde a rede tem capacidade de encontrar padrões de linguagem seja ela falada ou escrita, aprendendo a linguagem.

- Redes neurais artificiais intuitivas – Modelo autônomo que via sensores é capaz de realizar a leitura do ambiente ao seu entorno, interagir com o mesmo usando os resultados de tentativa e erro como aprendizado e a partir disso tomar decisões por conta própria.

- Redes neurais artificiais baseadas em Augumented Random Search (ARS) – Modelo que pode ser construído em paralelo a uma rede neural artificial intuitiva, seu foco se dá em criar mecanismos de aprendizado por reforço para um Agente que possui um corpo, dessa forma uma rede neural realizará aprendizado por reforço para o controle de seu corpo, enquanto outra realizará aprendizado por reforço para seu aprendizado lógico convencional.

Estes modelos citados acima são apenas uma pequena amostra do que se existe de redes neurais artificiais quando as classificamos quanto ao seu propósito. Raciocine que uma rede neural pode ser adaptada para resolução de praticamente qualquer problema computacional, e nesse sentido são infinitas as possibilidades. Outro ponto a destacar também é que é perfeitamente normal alguns tipos/modelos de redes neurais artificiais herdarem características de outros modelos, desde que o propósito final seja atendido.

Adendo – Augumented Random Search

O algoritmo Augmented Random Search, idealizado por Horia Mania e Aurelia Guy, ambos da universidade de Berkeley, é um modelo de estrutura de código voltada para inteligência artificial, principalmente no âmbito de abstrair certas estruturas de nosso Agente tornando-o um simulacro de ser humano ou de algum animal.

Diferente de outros modelos onde nosso agente é uma forma genérica com alguns sensores, por meio do ARS temos, entre outros modelos, o chamado MiJoCo (Multi-joint dynamics with contact), que por sua vez tenta simular para nosso agente uma anatomia com todas suas características fundamentais, como tamanho e proporção, articulações, integração com a física simulada para o ambiente, etc...

Se você já está familiarizado com os conceitos de Agente vistos nos capítulos anteriores, deve lembrar que o mesmo, via rede neural artificial intuitiva, trabalha com estados e ações, partindo do princípio de sua interação com o ambiente. Pois bem, quando estamos trabalhando com ARS estamos dando um passo além pois aqui o próprio corpo de nosso agente passa a ser um ambiente. Raciocine que o mesmo deverá mapear, entender e aprender sobre o seu corpo antes mesmo de explorar o ambiente ao seu entorno.

O conceito de nosso Agente se manterá da mesma forma como conhecemos, com sua tarefa de ir de um ponto A até um ponto B ou realizar uma determinada tarefa, realizando leituras do ambiente, testando possibilidades, aprendendo com recompensas e penalidades, etc... nos moldes de aprendizado por reforço. Porém dessa vez, de uma forma mais complexa, terá de realizar suas determinadas tarefas controlando um corpo.

Raciocine, apenas como exemplo, um Agente que deve explorar um ambiente e em um determinado ponto o mesmo deve subir um lance de escadas, raciocine que o processo que envolve o mesmo aprender o que é um lance de escadas e como progredir sobre esse "obstáculo" é feito considerando o que são suas pernas, como deverá mover suas pernas, qual movimento, em qual direção, em que período de tempo, com contração de quais grupos musculares, etc...

Se você já jogou algum jogo recente da série GTA, deve lembrar que os NPCs são dotados de uma tecnologia chamada Ragdoll que simula seu corpo e a física sobre o mesmo de forma realista, reagindo a qualquer estímulo do ambiente sobre si, aqui o conceito será semelhante, porém nosso Agente terá um corpo e uma inteligência, ao contrário do jogo onde há um corpo e um simples script de que ações o NPC fica realizando aleatoriamente para naquele cenário simplesmente parecer uma pessoa vivendo sua vida normalmente.

Note que novamente estaremos abstraindo algo de nossa própria natureza, todo animal em seus primeiros dias/meses de vida, à medida que descobre o seu próprio corpo e explora o ambiente, aprende sobre o mesmo. Ao tentar desenvolver uma habilidade nova, como por exemplo tocar um instrumento musical, parte do processo é entender a lógica envolvida e parte do processo é saber o que e como fazer com seu corpo.

Se você um dia, apenas como exemplo, fez aulas de violão, certamente se lembrará que para extrair diferentes sons do instrumento existe uma enorme variedade de posições dos seus dedos nas casas do violão para que se contraiam as cordas do mesmo para ressoar uma determinada frequência. Entender o que é um *ré menor* é lógica, a posição dos dedos nas cordas

é física, a prática de repetição feita nesse caso é para desenvolver uma memória muscular, onde sempre que você for tocar um ré menor, já saiba como e onde posicionar seus dedos para essa nota.

Ao programar este tipo de característica para um Agente, não teremos como de costume um algoritmo conceituando cada ação a ser realizada e em que ordem a mesma deve ser executada, mas fazendo o uso de q-learning, aprenderá o que deve ser feito em tentativa e erro fazendo uso de aprendizado por reforço.

Apenas concluindo esta linha de raciocínio, lembre-se que em nossos modelos de inteligência artificial tínhamos o conceito de continuidade, onde nosso Agente sempre estava em busca de algo a se fazer, uma vez que o "cérebro" do mesmo não para nunca seu processamento. Agora iremos somar a este modelo muito de física vetorial, pois estaremos ensinando o controle e movimento do corpo de nosso Agente em um ambiente simulado de 2 ou 3 dimensões, dependendo o caso.

Método de Diferenças Finitas

Um ponto que deve ser entendido neste momento é o chamado Método de Diferenças Finitas, pois esse trata algumas particularidades que diferenciará o ARS de outros modelos lógicos de inteligência artificial.

Como de costume, para nossos modelos voltados a inteligência artificial, temos aquele conceito básico onde nosso Agente realizará uma determinada ação por tentativa e erro, recebendo recompensas quando as ações foram dadas como corretas e penalidades quando executadas determinadas ações que diretamente ou indiretamente acabaram o distanciando de alcançar seu objetivo. Lembre-se que nesse processo, conforme nosso agente, via aprendizado por reforço, aprendia o jeito certo de realizar determinada tarefa, a medida que os padrões dos pesos eram reajustados para guardar os melhores padrões, a arquitetura do código era desenvolvida de forma a aos poucos descartar totalmente os processos das tentativas que deram errado.

O grande problema nisso é que, ao contextualizar este tipo de conceito para algoritmos em ARS, os mesmos irão contra a própria lógica e proposta de ARS. Vamos entender melhor, em uma rede neural artificial intuitiva comum, onde nosso Agente é genérico e possui apenas sensores para ler, mapear e explorar o ambiente, e um objetivo de, por exemplo, se deslocar de um ponto A até um ponto B, o mesmo simplesmente iria considerando qual ação de deslocamento o deixou mais afastado do ponto A e mais próximo do ponto B, sem a necessidade de considerar nada mais além disso. Agora raciocine que ao considerar um corpo para nosso Agente, o mesmo inicialmente iria entender o que são suas pernas, qual tipo de movimento é possível realizar com elas, e começar a fazer o uso das mesmas em seu deslocamento, porém, a cada tentativa dada como errada, a nível dos ajustes dos pesos em busca dos padrões corretos o mesmo iria "desaprender" um pouco a caminhar a cada erro, tendo que reaprender a andar de forma compensatória em busca do acerto.

Sendo assim, nossa estrutura de código deverá levar em consideração esse problema e ser adaptada de forma que nosso Agente não tenha de aprender a caminhar novamente a cada processo. Teremos de separar as coisas no sentido de que, aprender a caminhar e caminhar de

forma correta deve ser aprendida e não mais esquecida, e por parte de lógica, aprender quais ações foram tomada de forma errada deverão ser compensadas por ações tomadas de forma correta, sem interferir no aprendizado de caminhar.

Novamente estamos abstraindo coisas de nossa própria natureza, tentando simular nossos mecanismos internos que nos fazem animais racionais de forma simples e dentro de uma série de diversas limitações computacionais. Apenas tenha em mente que, por exemplo, você aprendeu a digitar textos em algum momento de sua vida, todas as funções físicas e cognitivas que você usa ao digitar não precisam ser reaprendidas caso você digite algo errado, mas simplesmente deve continuar usando a mesma habilidade para corrigir o erro lógico encontrado.

Dessa forma, naquele processo de retroalimentação das redes neurais utilizadas no processo, teremos algumas particularidades para que não se percam informações relativas a estrutura de nosso Agente e ao mesmo tempo sejam corrigidas apenas as variáveis que estão diretamente ligadas a sua lógica. É como se houvesse aprendizado simultâneo para “corpo e mente” de nosso Agente, porém nenhum aprendizado referente ao seu corpo pode ser perdido enquanto todo e qualquer aprendizado para sua mente pode sofrer alterações.

Em uma rede neural artificial intuitiva convencional o aprendizado é/pode ser feito ao final de cada ciclo, ao final de cada ação, agora, para conseguirmos separar os aprendizados de corpo e de mente de nosso Agente, esse processo é realizado ao final de todo o processo, tendo informações de todos os ciclos envolvidos desde o gatilho inicial até a conclusão da tarefa, de modo que possamos separar o que deu certo e errado no processo de aprendizado apenas por parte da mente de nosso Agente.

Se você entendeu a lógica até aqui, deve ter percebido que internamente não teremos mais aquele conceito clássico de descida do gradiente pois diferente de uma rede neural convencional, aqui teremos estados específicos para o corpo de nosso Agente e dinâmicos para sua mente. Ainda no exemplo de caminhar de um ponto A até um ponto B, o processo de caminhar sempre será o mesmo, para onde, de que forma, em que sentido, em qual velocidade será reaprendido a cada ciclo. Ao final dessa tarefa inicial, caso delegamos que nosso Agente agora caminhe de um ponto B até um ponto C, o mesmo não terá de aprender a caminhar novamente, mas se focará apenas em caminhar de encontro ao ponto C.

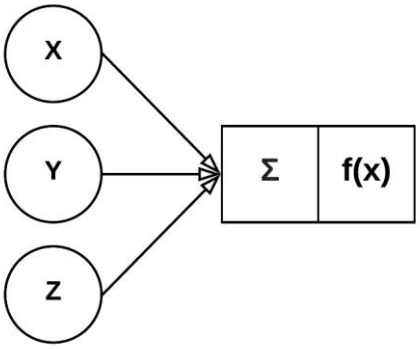
Finalizando nossa linha de raciocínio, agora o nome Augumented Random Search deve fazer mais sentido, uma vez que os estados e ações de nosso Agente serão realizados por uma pesquisa aleatória da interação de nosso Agente com o ambiente ao seu entorno, de forma extracorpórea, ou seja, a simulação de sua inteligência artificial em sua realidade aumentada leva a consideração de um corpo e o constante aprendizado de sua mente.

Perceptrons

Para finalmente darmos início a codificação de tais conceitos, começaremos com a devida codificação de um perceptron de uma camada, estrutura mais básica que teremos e que ao mesmo tempo será nosso ponto inicial sempre que começarmos a programar uma rede neural artificial para problemas de menor complexidade, chamados linearmente separáveis.

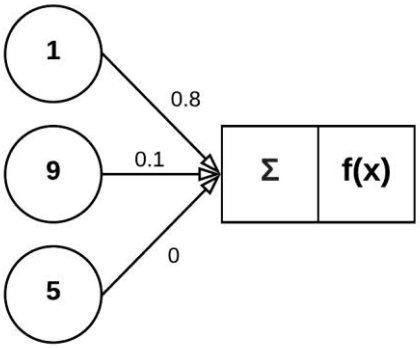
O modelo mais básico usado na literatura para fins didáticos é o modelo abaixo, onde temos a representação de 3 espaços alocados para entradas (X, Y e Z), note que estas 3 figuras se conectam com uma estrutura central com o símbolo Sigma Σ, normalmente atribuído a uma Função Soma, operação aritmética bastante comum e por fim, esta se comunica com uma última estrutura lógica onde há o símbolo de uma Função de Ativação f(x).

Em suma, estaremos sempre trabalhando com no mínimo essas três estruturas lógicas, haverão modelos muito mais complexos ou realmente diferentes em sua estrutura, mas por hora o que deve ficar bem entendido é que todo perceptron terá entradas, operações realizadas sobre estas e uma função que resultará na ativação ou não deste neurônio em seu contexto.



A partir deste modelo podemos resolver de forma computacional os chamados problemas linearmente separáveis. De forma bastante geral podemos entender tal conceito como o tipo de problema computacional onde se resulta apenas um valor a ser usado para uma tomada de decisão. Imagine que a partir desse modelo podemos criar uma pequena rede neural que pode processar dados para que se ative ou não um neurônio, podemos abstrair essa condição também como fazíamos com operadores lógicos, onde uma tomada de decisão resultava em 0 ou 1, True ou False, return x ou return y, etc… tomadas de decisão onde o que importa é uma opção ou outra.

Partindo para prática, agora atribuindo valores e funções para essa estrutura, podemos finalmente começar a entender de forma objetiva o processamento da mesma:



Sendo assim, usando o modelo acima, aplicamos valores aos nós da camada de entrada, pesos atribuídos a elas, uma função de soma e uma função de ativação. Aqui, por hora, estamos atribuindo manualmente esses valores de pesos apenas para exemplo, em uma aplicação real

eles podem ser iniciados zerados, com valores aleatórios gerados automaticamente ou com valores temporários a serem modificados no processo de aprendizagem.

A maneira que faremos a interpretação inicial desse perceptron, passo-a-passo, é da seguinte forma:

Temos três entradas com valores 1, 9 e 5, e seus respectivos pesos 0.8, 0.1 e 0.

Inicialmente podemos considerar que o primeiro neurônio tem um valor baixo mas um impacto relativo devido ao seu peso, da mesma forma o segundo neurônio de entrada possui um valor alto, 9, porém de acordo com seu peso ele gera menor impacto sobre a função, por fim o terceiro neurônio de entrada possuir valor 5, porém devido ao seu peso ser 0 significa que ele não causará nenhum efeito sobre a função.

Em seguida teremos de executar uma simples função de soma entre cada entrada e seu respectivo peso e posteriormente a soma dos valores obtidos a partir deles.

Neste exemplo essa função se dará da seguinte forma:

Soma = (1 * 0.8) + (9 * 0.1) + (5 * 0)

Soma = 0.8 + 0.9 + 0

Soma = 1.7

Por fim a função de ativação neste caso, chamada de Step Function (em tradução livre Função Degrau) possui uma regra bastante simples, se o valor resultado da função de soma for 1 ou maior que 1, o neurônio será ativado, se for um valor abaixo de 1 (mesmo aproximado, mas menor que 1) este neurônio em questão não será ativado.

Step Function = Soma => 1 (Ativação)

Step Function = Soma < 1 (Não Ativação)

Note que esta é uma fase manual, onde inicialmente você terá de realmente fazer a alimentação dos valores de entrada e o cálculo destes valores manualmente, posteriormente entraremos em exemplos onde para treinar uma rede neural teremos inclusive que manualmente corrigir erros quando houverem para que a rede possa “aprender” com essas modificações.

Outro ponto importante é que aqui estamos apenas trabalhando com funções básicas que resultarão na ativação ou não de um neurônio, haverão situações que teremos múltiplas entradas, múltiplas camadas de funções e de ativação e até mesmo múltiplas saídas…

Última observação que vale citar aqui é que o valor de nossa Step Function também pode ser alterado manualmente de acordo com a necessidade.

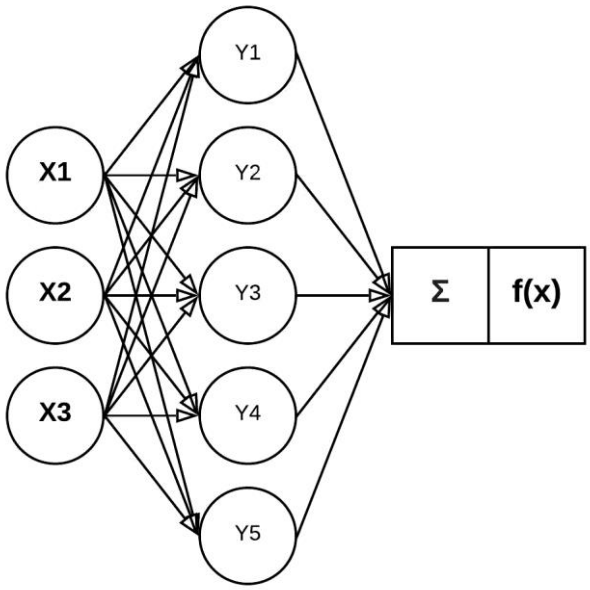
Com base nesse processamento podemos concluir que de acordo com o parâmetro setado em nossa Step Function, este perceptron em sua aplicação seria ativado, desencadeando uma tomada de decisão ou classificação de acordo com o seu propósito.

Perceptron Multicamada

Uma vez entendida a lógica de um perceptron, e exemplificado em cima de um modelo de uma camada, hora de aumentarmos um pouco a complexidade, partindo para o conceito de perceptron multicamada.

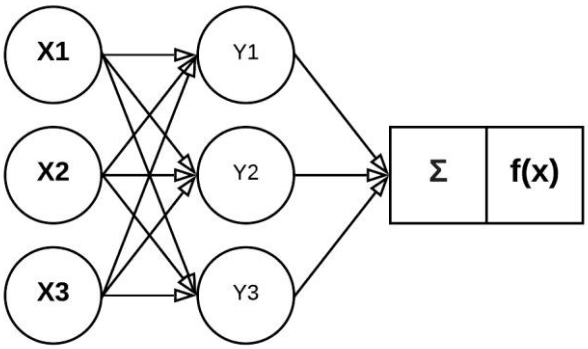
O nome perceptron multicamada por si só é bastante sugestivo, porém é importante que fique bem claro o que seriam essa(s) camada(s) adicionais. Como entendido anteriormente, problemas de menor complexidade que podem ser abstraídos sobre o modelo de um perceptron tendem a seguir sempre a mesma lógica, processar entradas e seus respectivos pesos, executar uma função (em perceptron de uma camada normalmente uma função de soma) e por fim a aplicação de uma função de ativação (em perceptron de uma camada normalmente uma função degrau, multicamada podemos fazer uso de outras como função sigmoide por exemplo).

Agora, podemos adicionar o que por convenção é chamado de camada oculta em nosso perceptron, de forma que o processamento dos dados de entrada com seus pesos, passarão por uma segunda camada para ajuste fino dos mesmos, para posteriormente passar por função e ativação.



Repare no modelo acima, temos entradas representadas por X1, X2 e X3 e suas conexões com todos neurônios de uma segunda camada, chamada camada oculta, que se conecta com o neurônio onde é aplicada a função de soma e por fim com a fase final, a fase de ativação.

Além desse diferencial na sua representatividade, é importante ter em mente que assim como o perceptron de uma camada, cada neurônio de entrada terá seus respectivos pesos, e posteriormente, cada nó de cada neurônio da camada oculta também terá um peso associado. Dessa forma, a camada oculta funciona basicamente como um filtro onde serão feitos mais processamentos para o ajuste dos pesos dentro desta rede.



Apenas para fins de exemplo, note que a entrada X1 tem 3 pesos associados que se conectam aos neurônios Y1, Y2 e Y3, que por sua vez possuem pesos associados à sua conexão com a função final a ser aplicada.

Outro ponto importante de se salientar é que, aqui estamos trabalhando de forma procedural, ou seja, começando dos meios e modelos mais simples para os mais avançados. Por hora, apenas para fins de explicação, estamos trabalhando com poucos neurônios, uma camada oculta, um tipo de função e um tipo de ativação. Porém com uma rápida pesquisada você verá que existem diversos tipos de função de ativação que posteriormente se adequarão melhor a outros modelos conforme nossa necessidade. Nos capítulos subsequentes estaremos trabalhando com modelos diferentes de redes neurais que de acordo com suas particularidades possuem estrutura e número de neurônios e seus respectivos nós diferentes dos exemplos tradicionais mencionados anteriormente.

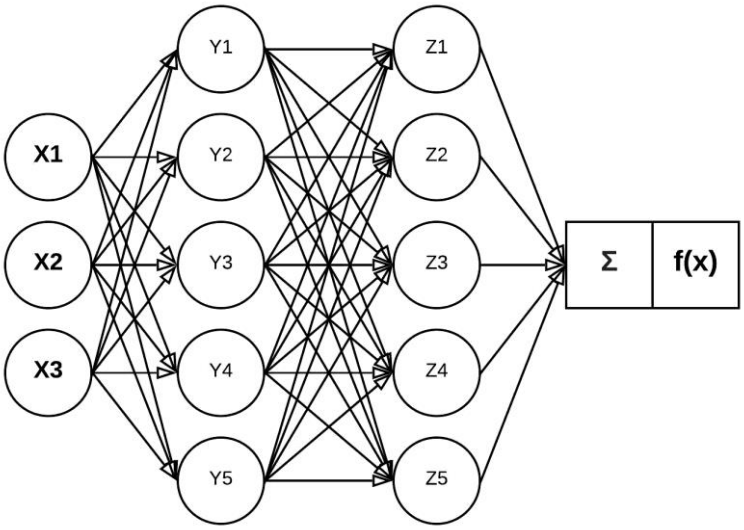
Porém tudo ao seu tempo, inicialmente trabalhando com problemas computacionais de menor complexidade os modelos acima são suficientes, posteriormente para problemas mais robustos estaremos entendendo os diferentes modelos e formatos de estrutura de uma rede neural artificial.

Deep Learning

Um dos conceitos mais distorcidos dentro dessa área de redes neurais é o de Deep Learning (em tradução livre Aprendizado Profundo). Como vimos anteriormente, uma rede neural pode ser algo básico ou complexo de acordo com a necessidade. Para problemas de lógica simples, tomadas de decisão simples ou classificações linearmente separáveis, modelos de

perceptron com uma camada oculta junto a uma função de soma e de ativação correta normalmente é o suficiente para solucionar tal problema computacional.

Porém haverão situações onde, para resolução de problemas de maior complexidade teremos de fazer diferentes tratamentos para nossos dados de entrada, uso de múltiplas camadas ocultas, além de funções de retroalimentação e ativação específicas para determinados fins bastante específicos de acordo com o problema abstraído. O que acarreta em um volume de processamento de dados muito maior por parte da rede neural artificial, o que finalmente é comumente caracterizado como deep learning.



A “aprendizagem” profunda se dá quando criamos modelos de rede neural onde múltiplas entradas com seus respectivos pesos passam por múltiplas camadas ocultas, fases supervisionadas ou não, feedforward e backpropagation e por fim literalmente algumas horas de processamento. Na literatura é bastante comum também se encontrar este ponto referenciado como redes neurais densas ou redes neurais profundas.

Raciocine que para criarmos uma arquitetura de rede neural que aprende por exemplo, a lógica de uma tabela verdade AND (que inclusive criaremos logo a seguir), estaremos programando poucas linhas de código e o processamento do código se dará em poucos segundos. Agora imagine uma rede neural que identifica e classifica um padrão de imagem de uma mamografia digital em uma base de dados gigante, além de diversos testes para análise pixel a pixel da imagem para no fim determinar se uma determinada característica na imagem é ou não câncer, e se é ou não benigno ou maligno (criaremos tal modelo posteriormente). Isto sim resultará em uma rede mais robusta onde criaremos mecanismos lógicos que farão a leitura, análise e processamento desses dados, requerendo um nível de processamento computacional alto. Ao final do livro estaremos trabalhando com redes dedicadas a processamento puro de imagens, assim como a geração das mesmas, por hora imagine que este processo envolve múltiplos neurônios, múltiplos nós de comunicação, com múltiplas camadas, etc…, sempre proporcional a complexidade do problema computacional a ser resolvido.

Então a fim de desmistificar, quando falamos em deep learning, a literatura nos aponta diferentes nortes e ao leigo isso acaba sendo bastante confuso. De forma geral, deep learning

será uma arquitetura de rede neural onde teremos múltiplos parâmetros assim como um grande volume de processamento sobre esses dados por meio de múltiplas camadas de processamento.

Q-Learning e Deep Q-Learning

Uma vez entendidos os princípios básicos de um modelo de rede neural artificial, assim como as principais características comuns aos modelos de redes neurais artificiais, podemos avançar nossos estudos entendendo os mecanismos internos de um algoritmo intuitivo e o que faz dele um modelo a ser usado para simular uma inteligência artificial.

Raciocine que o que visto até então nos capítulos anteriores é como o núcleo do que usaremos em redes neurais artificiais, independentemente do seu tipo ou propósito. Ao longo dos exemplos estaremos entendendo de forma prática desde o básico (perceptrons) até o avançado (inteligência artificial), e para isso temos que identificar cada uma dessas particularidades em seu contexto.

Por hora, também é interessante introduzir os conceitos de inteligência artificial, porém os mesmos serão estudados com maior profundidade em seus respectivos capítulos.

Para separarmos, sem que haja confusão, conceitos teóricos de machine learning, deep learning e q-learning, estaremos sempre que possível realizando a comparação entre esses modelos.

Dando sequência, vamos entender as particularidades em Q-Learning.

Equação de Bellman

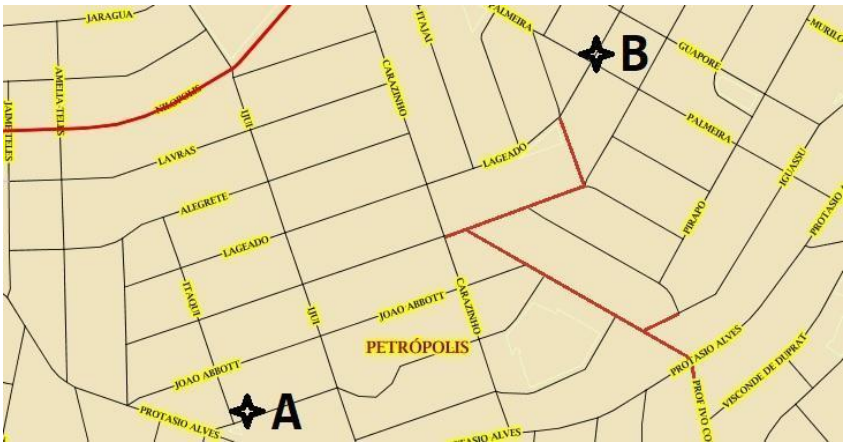
Richard Bellman foi um matemático estadunidense referenciado no meio da ciência da computação pela “invenção” da programação dinâmica em 1953, modelo esse de equações e algoritmos aplicados para estados de programação em tempo real. Em outras palavras, por meio de suas fórmulas é possível criar arquiteturas de código onde camadas de processamento podem ser sobrepostas e reprocessadas em tempo real. Como mencionado anteriormente, um dos grandes diferenciais do modelo de rede neural intuitiva é a capacidade da mesma de trabalhar continuamente, realizando processamento de seus estados e memória em tempo real.

Por parte da lógica desse modelo, raciocine que teremos um agente, um ambiente e um objetivo a ser alcançado. O agente por si só terá uma programação básica de quais ações podem ser tomadas (em alguns casos uma programação básica adicional de seu tamanho e formado) e com base nisso o mesmo realizará uma série de testes fazendo o mapeamento do ambiente até que atinja seu objetivo.

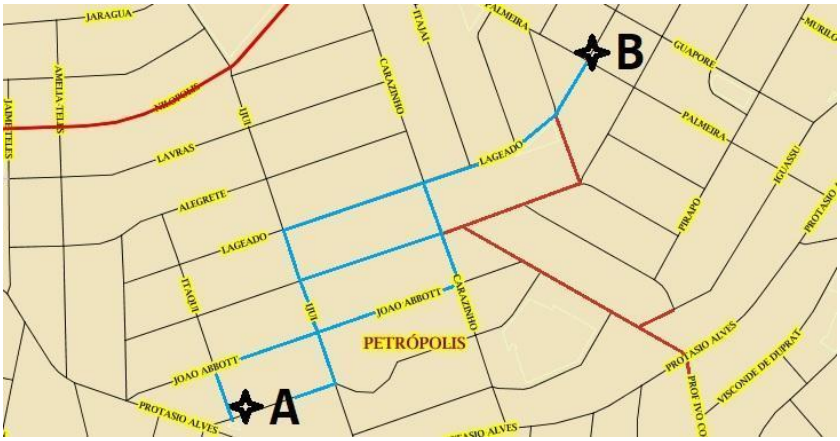
Quando atingido o objetivo esperado, o agente irá verificar qual caminho percorreu, quais sequências de ações foram tomadas do início ao final do processo e irá salvar essa

configuração em uma espécie de memória. A experiência desse agente com o ambiente se dará de forma sequencial, uma ação após a outra, de forma que caso ele atinja algum obstáculo receberá uma penalidade, assim como quando ele alcançar seu objetivo receberá uma recompensa.

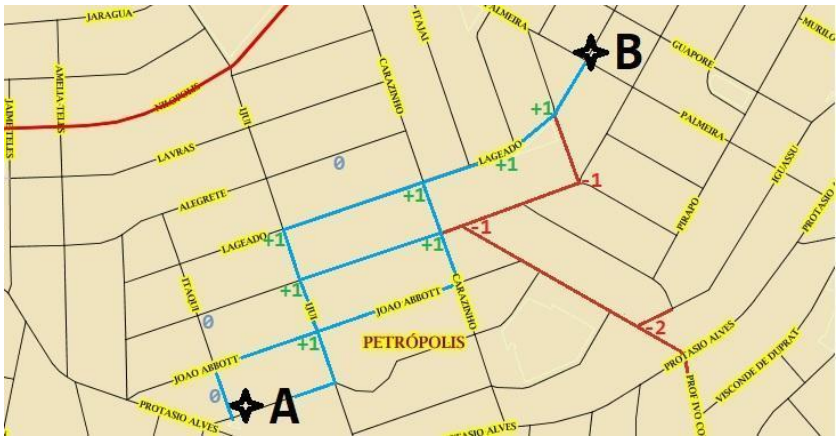
Toda trajetória será remapeada, passo a passo realizado, atribuindo valores de experiência positiva quando correto, de forma que numa situação de repetir o mesmo percurso, o agente irá buscar e replicar as ações que ele já conhece e sabe que deram certo ao invés de tentar novas experiências.



Apenas como exemplo, imagine que você possui um agente (com suas características de possíveis ações a serem tomadas bem definidas), um ambiente que nesse caso trata-se de um mapa de uma determinada cidade e um objetivo de ir do ponto A até o B. Realizando uma leitura rápida desse mapa podemos ver duas características importantes, primeira delas que existem diversas possíveis rotas que fazem o trajeto do ponto A até o ponto B. Note que destacado em linhas vermelhas temos alguns segmentos dessas trajetórias que por algum motivo estão fechadas, sendo assim, o mapeamento de todas possíveis rotas é feito.



Agora delimitados como linhas azuis estão traçadas algumas rotas diretas entre o ponto A e o ponto B. O que nosso agente irá fazer é percorrer cada uma dessas rotas identificando sempre o seu nível de avanço conforme a distância de seu objetivo diminui, assim como demarcar cada etapa concluída com uma pontuação positiva para cada possibilidade válida executada com sucesso, assim como uma pontuação negativa quando atingido qualquer estado onde seu progresso for interrompido. Lembrando que cada passo em busca de seu objetivo é feito uma nova leitura do seu estado atual e sua posição no ambiente.



Apenas exemplificando, note que pontuamos alguns trajetos com uma numeração entre -2, -1, 0 e +1 (podendo usar qualquer métrica ao qual se sinta mais confortável adotar). Possíveis trajetos classificados como 0 são aqueles que não necessariamente estão errados, porém não são os mais eficientes, no sentido de ter uma distância maior do objetivo sem necessidade. Em vermelho, nos entroncamentos das ruas representadas por linhas vermelhas temos pontuação -2 por exemplo, onde seria um trajeto que nosso agente testou e considerou muito fora da rota esperada.

Da mesma forma os entroncamentos pontuados em -1 por sua vez foram testados pelo agente e considerados ineficientes. Por fim, sobre as linhas azuis que representam os melhores caminhos, existem marcações +1 sinalizando que tais caminhos são válidos e eficientes em seu propósito de traçar a menor distância entre o ponto A e o ponto B.

Como mencionado anteriormente, o agente arbitrariamente irá testar todos os possíveis trajetos deste ambiente, gerando com base em sua experiência um mapa dessas rotas corretas. Desse modo, caso essa função tenha de ser repetida o agente irá se basear puramente nos resultados corretos que ele conhece de sua experiência passada.

$$V(s) = \max_{a}(R(s,a) + \gamma V(s'))$$

Traduzindo este conceito lógico em forma de equação chegamos na chamada equação de Bellman, equação esta que integrará o algoritmo de nossa rede neural de forma a realizar cálculos para cada estado do agente, podendo assim prever qual a próxima ação a ser tomada de acordo com os melhores resultados, já que sempre estamos buscando melhor eficiência.

Sem nos aprofundarmos muito no quesito de cálculo, podemos entender a lógica dessa equação para entendermos como a mesma é alimentada de dados e realiza seu processamento.

Na equação descrita acima, temos alguns pontos a serem entendidos, como:

V – Possíveis valores de cada estado a cada etapa.

s – Estado atual de nosso agente.

s’ – Próximo estado de nosso agente.

A – Ação a ser tomada.

R – Recompensa gerada para o resultado da ação tomada por nosso agente.

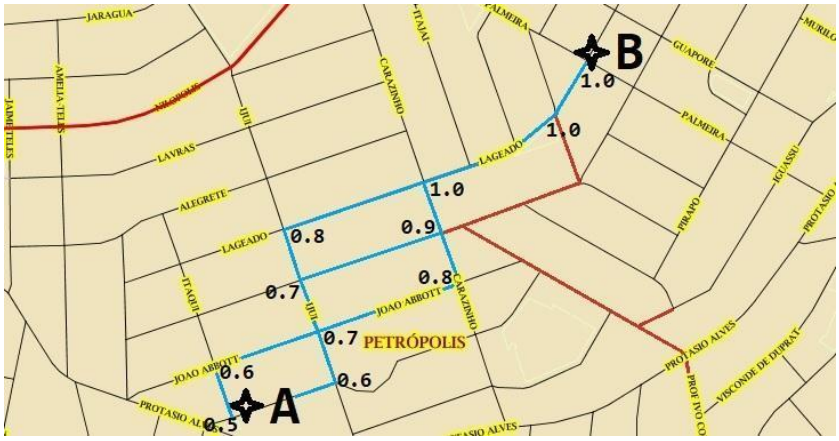
y – Fator de desconto.

V (s) – Qual é o valor do último estado de nosso agente.

R (s, a) – Lendo inversamente, de acordo com a última ação o agente recebe um valor de estado para que seja recompensado ou penalizado.

* A equação de Bellman é calculada a cada estado de nosso agente, atualizando assim os seus dados de estado.

* Mesmo que numericamente os estados se repitam quando calculados manualmente, vale lembrar que a cada ação executada um novo estado é alcançado.



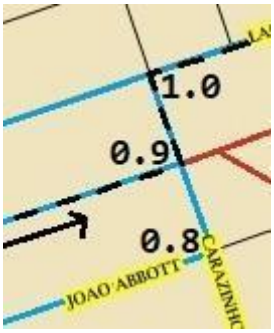
Com base na aplicação da equação de Bellman para cada estado é retroativamente pontuados os valores dos melhores caminhos e tais dados passam a compor o equivalente a memória de longa duração de nosso agente.

Se o mesmo objetivo com os mesmos parâmetros for necessário ser realizado novamente, nosso agente já possui todos os dados de experiência, apenas necessitando replicá-los.

Plano ou Política de Ação

Note que em nosso exemplo a pontuação aplicada para cada estado segue a simples lógica de que quanto maior for o número mais próximo do objetivo estará o nosso agente. Como normalmente acontece na prática, existirão diversos caminhos corretos, o que nosso agente fará para descobrir a melhor ou mais eficiente rota simplesmente será fazer a sua leitura do seu estado atual, assim como as pontuações dos possíveis estados e ações a serem tomadas, optando automaticamente para avançar para o próximo estado mapeado de maior pontuação.

Essa ação tomada de forma mais direta normalmente é denominada como plano de ação, para alguns autores, este tipo de lógica usada para tomada de decisão é como se fosse uma modelo de aprendizado de máquina auto supervisionado, onde o agente sabe com margem de precisão muito alta o que deve ser feito, porém em situações de maior complexidade teremos inúmeros fatores que poderão prejudicar essa precisão, fazendo com que o agente tenha que considerar esses fatores impactando diretamente seu desempenho, ou a probabilidade de sucesso de suas tomadas de decisão.



Apenas como exemplo, seguindo a linha tracejada indicada pela seta, ao chegar no entroncamento nosso agente terá de fazer a leitura de seu estado atual, do ambiente (bloqueado à frente, com caminhos possíveis para a direita e para a esquerda), e fará sua tomada de decisão, nesse caso aplicando seu plano de ação para 1 de apenas 2 direções, e com base no valor mais alto do próximo estado, nesse caso, seguindo o caminho à esquerda.

Haverá situações onde o agente terá muitos fatores que prejudicam sua leitura de estado ou de ambiente, assim como pontos de estado alcançado onde será necessário realizar múltiplas escolhas. Para essas situações devemos considerar todos os fatores inclusive dados de ações imediatamente realizadas nos passos anteriores ou a leitura da possível modificação do ambiente entre um estado e outro para que se determine direções com melhor probabilidade de acertos.

Como exemplo, imagine a complexidade do sistema de tomada de decisão de um carro autônomo, sua IA faz a leitura do ambiente dezenas de vezes por segundo, também levando em consideração que dependendo da velocidade do carro percorrendo um determinado trajeto, em uma fração de segundo o ambiente pode se modificar à sua frente caso algum animal atravesse a pista ou um objeto seja jogado nela como exemplo. Raciocine que existe uma série de ações a serem tomadas seja tentando frear o carro ou desviar do obstáculo (ou as duas coisas) ainda tendo que levar em consideração qual tipo de ação terá maior probabilidade de sucesso, tudo em uma fração de segundo.

Sistema de Recompensas e Penalidades

Como vimos nos tópicos anteriores, nosso agente de acordo com suas ações recebe recompensas ou penalidades como consequência de seus atos, porém é necessário entender um pouco mais a fundo como esse mecanismo de fato funciona em nosso algoritmo.

Anteriormente também comentamos que uma das principais características de nosso modelo de rede neural artificial intuitiva é a de que nosso agente a partir de um gatilho inicial se torna "vivo", agindo de forma constante e independente. Isso se dá porque devemos considerar que todo e qualquer tipo de processamento constante funciona de forma cíclica, podendo ter uma série de particularidades em seu ciclo, mas basicamente um ciclo de processamento tem um estágio inicial, camadas de processamento de entradas gerando saídas que retroalimentam as entradas gerando um loop de processamento.

Contextualizando para nosso agente, todo e qualquer agente sempre fará o ciclo de leitura de seu estado atual (inicial), tomará uma série de ações para com o ambiente ao seu redor, recebendo recompensas ou penalidades de acordo com as consequências de suas ações, retroalimentando com estes dados o seu novo estado que por sua vez é o seu "novo" estado atual (inicial).

Para alguns autores, pontuar essas recompensas e penalidades facilita a interpretação dos estados de nosso agente, porém devemos tomar muito cuidado com esse tipo de interpretação, uma vez que se você raciocinar a lógica deste modelo, na realidade teremos uma leitura diferencial, com muito mais penalidades do que recompensas uma vez que independentemente da tarefa a ser realizada, boa parte das vezes existem poucos meios de realizar tal função corretamente, em contraponto a inúmeras maneiras erradas de se tentar realizar a mesma.

Logo, devemos ter esse tipo de discernimento para que possamos de fato ler o estado de nosso agente e entender suas tomadas de decisão. Internamente, por parte estruturada do algoritmo, lembre-se que a aprendizagem por reforço justamente visa dar mais ênfase aos acertos, dando mais importância a eles dentro do código, enquanto alguns modelos até mesmo descartam totalmente grandes amostragens de erros, uma vez que eles são maioria e podem ocupar muito da capacidade de memória de nosso agente.

Na prática, nos capítulos onde estaremos implementando tais conceitos em código, você notará que temos controle da taxa de aprendizado de nosso agente, assim como temos controle dos pesos das penalidades no processo de aprendizado de máquina. Teremos meios

para dar mais pesos aos erros enfatizando que os mesmos tenham seus padrões facilmente identificados pela rede neural de forma a reforçar o aprendizado da rede pelo viés correto.

Aplicação sob Diferença Temporal

A essa altura, ao menos por parte teórica, você já deve estar entendendo o funcionamento lógico de uma rede neural artificial intuitiva, assim como o papel de nosso agente, a maneira como o mesmo se comporta em relação ao ambiente e as suas ações.

Hora de aprofundar um pouco o conceito de diferença temporal, haja visto que já temos uma boa noção sobre o mesmo.

Em suma, quando estamos falando que neste tipo específico de rede neural artificial temos processamento em tempo real, aprendizado auto supervisionada, tempo de ação, tomada de ação autônoma, etc... estamos falando de estruturas lógicas que irão simular todo processamento necessário dessa inteligência artificial em decorrência de um período de tempo real. Caso você procure por artigos científicos que tratam sobre esse assunto verá inúmeros exemplos de como moldar uma estrutura lógica e de código de forma que são aproveitados resultados de amostras de processamento passados assim como pré-programados já estarão uma série de novas ações a serem tomadas, repetindo esses passos em ciclo.

Todo e qualquer código tem o que chamamos de interpretação léxica, uma forma sequencial de ler linhas e blocos de código. A partir dos mesmos, temos de criar estruturas que fiquem carregadas na memória realizando processamento de dados de entrara e saída de forma sequencial e cíclica. Um computador não tem (ainda) discernimento sobre coisas tão abstratas como fisicamente se dá a passagem do tempo, o que temos são dados de tempo de processamento contínuo, que será usado para simular tempo real.

Tenha em mente que diferente dos modelos de redes neurais artificiais convencionais, onde o processamento tem um começo, meio e fim bem definidos, nosso agente terá para si um modelo de rede neural artificial baseado em um começo (um gatilho inicial), meio e um recomeço. Dessa maneira simulamos continuidade em decorrência do tempo.

Outro ponto importante a ser observado é que este modelo, quando comparado aos modelos de redes neurais artificiais comuns, não possui explicitamente dados base, dados para treino e teste, no lugar destas estruturas estaremos criando o que por alguns autores é a chamada memória de replay, onde a base se constrói gradualmente por dados de experiência se auto retroalimentando.

Apenas finalizando essa linha de raciocínio, você terá de moldar seu agente (e isso será visto na prática nos capítulos de implementação) de forma que o mesmo interpretará estados passados (últimas ações executadas), estados presentes (estado atual + mapeamento do ambiente + recompensa obtida) e estados futuros (próximas ações a serem tomadas) de forma que essa sequência se repetirá de forma intermitente.

$$V(s) = \max_{a}(R(s,a) + \gamma V(s'))$$

Anteriormente, vimos que de acordo com a equação de Bellman, havia uma variável V que por sua vez representava os possíveis valores de cada estado a cada etapa.

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a')$$

Agora temos uma variável Q (s, a) que representa valores que já foram computados em uma fase de tomada de ação anterior e que está armazenado em memória, inclusive levando em consideração o valor de recompensa obtido anteriormente.

Repare que desta maneira, estamos sempre trabalhando levando em consideração o estado anterior para que seja previsto o próximo estado.

$$TD(a,s) = R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)$$

Assim chegamos na fórmula da equação final, onde temos uma variável TD representando a diferença temporal, que por sua vez realizará a atualização dos valores de Q. Novamente, a cada etapa de execução, a aplicação dessa fórmula será realizada, com o diferencial que agora ela não se inicia aleatoriamente, mas a partir do último valor de estado obtido, gerando um ciclo de processamento retroalimentado e capaz de simular continuidade, já que sempre será retroalimentado e sempre será tomada uma nova decisão.

Transfer Learning

Revisando alguns conceitos e desmistificando outros, quando estamos falando de modelos de redes neurais artificiais o convencional é que, para certos tipos de problemas computacionais, desenvolvemos nossos próprios modelos (não do absoluto erro, uma vez que usamos de estrutura de códigos de bibliotecas dedicadas a tais fins, mas compondo manualmente camada por camada de nossos modelos), treinando os mesmos e os implementando para alguma aplicação real.

Porém, existem certos contextos onde, dada a alta complexidade de um problema, podemos recorrer a estruturas pré-moldadas para tal propósito assim como fazer uso de modelos pré-treinados para tal fim, poupando todo ou parte do processo de treinamento de uma rede neural que, como bem sabemos, proporcional a sua complexidade e ao hardware onde a qual é executada pode ser um processo maçante.

Dentro dessa linha de raciocínio, a prática de uso de transferência de aprendizagem vem sendo cada vez mais adotada por estudantes e profissionais da área.

Em outras palavras, basicamente raciocine que sempre que estivermos falando de transfer learning, estamos falando de estruturas inteiras de código já desenvolvidas, treinadas, testadas e aprimoradas por terceiros, normalmente de livre acesso ou com pequenas restrições

legais, onde reaproveitamos quase totalmente o modelo, apenas o adaptando para um novo propósito.

No âmbito de redes neurais artificiais convolucionais (dedicadas a processamento diretamente sobre imagens) não é diferente, pois temos diversos modelos e abordagens para que possamos de fato aplicar mecanismos de aprendizado de máquina para que se extraiam características de imagens, reconhecendo padrões a partir das mesmas.

Nesse contexto, como dito anteriormente, o método convencional é que a partir de uma base de dados de imagens previamente tratada, criemos uma arquitetura de rede neural especializada (usando de camadas que aplicam mecanismos de convolução além, é claro, das camadas convencionais da rede) de modo que o processo de aprendizado de máquina é realizado a partir do zero e exclusivamente para aquela base de dados e para um determinado propósito bastante específico.

Porém, por mais contraditório que possa inicialmente parecer, uma técnica possível de aplicação em redes neurais artificiais convolucionais, que inclusive tem se mostrado bastante eficiente, é a chamada transferência de aprendizado diferencial. Nesta técnica, usamos de uma arquitetura de rede neural artificial já pronta e treinada para outros fins (existe um leque consideravelmente grande de modelos prontos e distribuídos por pesquisadores da área, como AlexNet, Inception, entre outros que recomendo fortemente sua pesquisa) em nossa nova base de dados, para que assim possamos usar da experiência previamente treinada de um modelo como referência para os mecanismos de aprendizado de máquina de nosso modelo.

Em outras palavras, é como se usássemos uma rede neural artificial convolucional pronta e treinada para reconhecimento de um tipo de objeto para classificar outro tipo de objeto completamente distinto, por exemplo, um modelo treinado com todas as características possíveis de um garfo para que se detecte e classifique uma faca (uma vez que o modelo sabe "tudo" sobre um garfo, é relativamente fácil para o mesmo deduzir que quando um objeto não é um garfo, será uma faca conforme dados repassados via aprendizado supervisionado).

O interessante deste processo é que, novamente ressaltando, por mais contraditório que possa parecer, se descobriu que usar de um modelo muito treinado para um propósito pode gerar bons resultados quando usamos do aprendizado de máquina do mesmo para novos propósitos, desde que próximos ou dentro de um mesmo contexto.

Muito desse processo se faz, como dito anteriormente, por meio de algoritmos internos de diagnóstico diferencial, onde uma rede neural artificial usada como referência sabe "tudo" sobre um determinado padrão de objeto, e quando uma nova amostra é sujeitada a testes pela mesma, ela busca todas as características que conhece, destacando as que não conhece para gerar novos padrões ou simplesmente agregando os novos padrões como novas categorias em sua bagagem de conhecimento.

Apenas como exemplo e já contextualizando para o viés de nosso projeto neste capítulo, na área da Radiologia médica existe o profissional Médico Radiologista, especializado em realizar a leitura e diagnóstico a partir de exames de imagem como raios-x, ultrassom, tomografia computadorizada, ressonância magnética, entre outras modalidades.

Dentro desse contexto, é simplesmente humanamente impossível tal profissional saber todas as características de todas as possíveis patologias que podem ser identificadas em um exame de imagem. Em seu período de especialização, tal médico estuda a fundo todas as

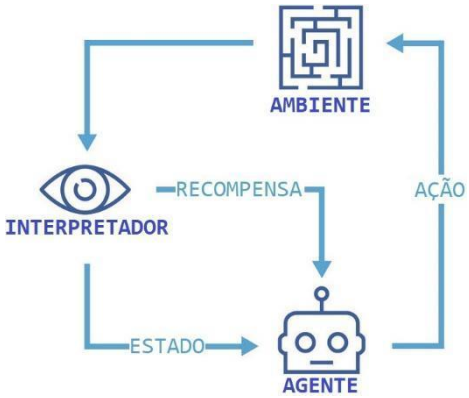
características de uma anatomia normal em um exame de imagem, para quando identificar algo fora dessa normalidade classificar tal exame como patológico e a partir disto buscar entender que patologia se encontra em evolução, além é claro, de aprender nesse processo sobre a nova característica variante.

No processo de transfer leaning realizamos algo muito próximo a isso, entendendo em nossa base de dados novos padrões a partir de padrões já bem estabelecidos em dados passados, usando mecanismos que testam se as características de uma determinada amostra é diferente de tudo o que se conhece sobre a mesma.

Logicamente, existem vantagens e desvantagens nesta modalidade, sendo (ao menos a meu ver) a maior vantagem o tempo poupado para o desenvolvimento e treinamento do modelo, tendo como desvantagem uma carência de modelos facilmente adaptáveis para certos contextos bastante específicos. Raciocine que de nada adianta esperarmos grandes resultados se usarmos um modelo pré treinado com características de veículos para tentar classificar massas tumorais cerebrais, porém, usando de modelos de propósitos aproximados, adaptando e configurando os mesmos para nosso problema computacional atual, podemos conseguir resultados bastante relevantes.

Modelo de Rede Neural Artificial Intuitiva

Apenas encerrando essa etapa, com o conhecimento acumulado até então, podemos finalmente estruturar um modelo lógico a ser transformado para código.



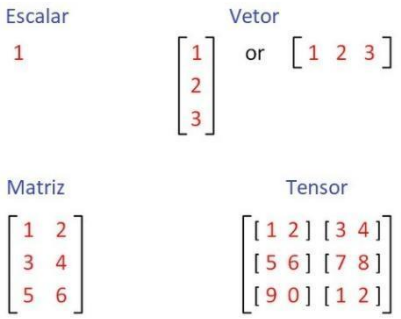
Em suma, teremos um agente, que por sua vez pode ser qualquer tipo de objeto ou ser vivo; Da mesma forma teremos um ambiente a ser explorado, com suas características particulares inicialmente desconhecidas por nosso agente; Sendo assim, nosso agente após ser ativado uma vez irá reconhecer seu formato e possíveis funções e a partir disto começará a explorar o ambiente, literalmente por tentativa e erro assim como milhares (ou até mesmo milhões) de execuções dependendo da complexidade de seu objetivo. À medida que o mesmo realiza determinadas ações, é recompensado ou penalizado para que dessa forma encontre o padrão correto de alcançar seu objetivo.

Tensores

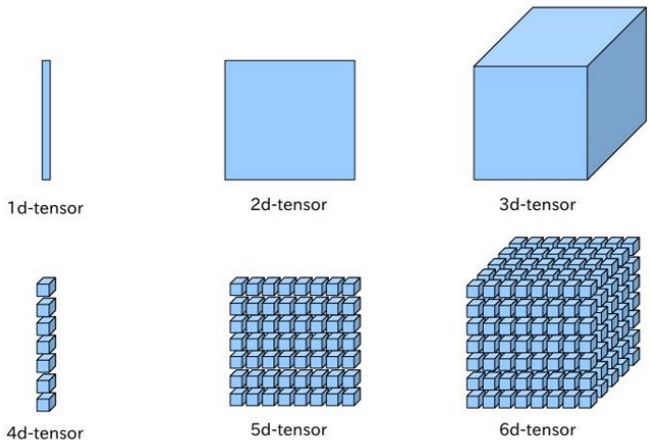
Quando estamos falando de redes neurais artificias temos uma vasta gama de arquiteturas desenvolvidas para os mais diversos propósitos, cada uma com suas particularidades no que diz respeito aos meios e métodos usados para resolver certos problemas computacionais.

Os modelos criados a partir do TensorFlow não fogem a essa regra, uma vez que sua arquitetura é única se comparado com outras bibliotecas de matemática computacional. Usando de arquiteturas de código desenvolvidas pelos engenheiros da Google Brain baseados em tensores para fluxos de informação, sendo que nesses moldes a biblioteca TensorFlow se sobressaiu em praticidade e performance quando comparada a outras.

Sendo assim, um dos primeiros conceitos que devemos explorar é justamente esse sistema de tensores, estruturas lógicas para processamento de dados de forma modularizada.



Bibliotecas científicas normalmente usam de tipos e dados em padrão escalar, vetorial e/ou matricial dependendo de outras ferramentas para padronizar e permitir o cruzamento de dados. Já a biblioteca TensorFlow usa destes tipos de dados internamente sem distinções, apenas os tratando como matrizes multidimensionais.



Dessa forma o mapeamento de certos dados para dimensões diretamente aplicada aos tensores do modelo acaba por realizar o processamento de tais dados em busca de seus padrões e todo aprendizado de máquina de modo mais eficiente tanto em performance quanto em margem de precisão em processos de classificação e/ou regressão.

Apenas fazendo uma ressalva, o modelo teórico de um processamento de dados em camadas de matrizes não é algo novo, exames de tomografia computadorizada, por exemplo, trabalham desde a década de 60 com imagens geradas em pixels e voxels para renderizar volume em imagem. O que outras bibliotecas não haviam feito até então era usar deste modelo teórico para transformar tal estrutura em camadas de processamento em uma rede neural artificial.

Sendo assim, em resumo podemos inicialmente entender que o sistema de tensores nada mais é do que uma organização de dados modulares executados em sessões (inclusive permitindo computação paralela) onde um fluxo progressivo de informações é processado por meio de tensores, invólucros independentes que podem conter diversas estruturas de dados e funções a serem executadas.

Os tipos de dados que serão apresentados em seguida, variáveis e constantes, guardarão dados/valores atribuídos para si e toda e qualquer operação com estes tipos de dados serão feitas por meio de tensores. Este conceito pode parecer um tanto quanto confuso inicialmente, porém na prática você entenderá facilmente estas distinções.

Como nosso foco neste livro é uma abordagem prática, muitos dos conceitos envolvidos também serão devidamente explicados à medida que aplicamos os mesmos em nossos exemplos.

Constantes e Variáveis

Uma vez que temos nossos contêineres de processamento de dados em fluxo chamados tensores, outras estruturas de dados que estarão interagindo diretamente com nossos tensores são as chamadas constantes e variáveis.

Por hora, raciocine que no processo de criação de nossa rede neural artificial via tensorflow teremos sempre alguns dados de entrada, estes por sua vez podem ter valores predefinidos fixos (constantes) assim como valores iniciais dinâmicos (variáveis), que sofrerão alterações à medida que são instanciados por algum tensor.

Existe uma terceira estrutura relacionada a variáveis chamada placeholder, que por sua vez nada mais é do que um marcador que reserva um espaço o qual será alimentado com novos dados no decorrer da execução da rede neural.

Sendo assim, nas estruturas de código teremos tais tipos de dados demarcados em nós que irão interagir com/em tensores conforme a rede demande o uso destes dados.

Podemos fazer um paralelo com uma arquitetura de rede neural convencional apenas para que fique claro tais nomenclaturas. Raciocine que em um modelo convencional, equivalente aos neurônios da camada de entrada de uma rede neural artificial teremos nossas constantes.

Da mesma forma, entre camadas de neurônios, onde eram realizados os cálculos e ajustes dos pesos, no tensorflow teremos nossas variáveis.

Por fim, em redes neurais que guardam um estado inicial zerado/nulo, por meio de mecanismos de aprendizado de máquina como aprendizado por reforço, a rede retroalimenta essa memória a cada fase de execução da rede guardando o último estado apenas atualizando o mesmo, inicialmente vazio, com novos dados conforme a rede neural aprende.

Sessões

Encerrando o básico do conceitual teórico inicial do TensorFlow, uma das estruturas de processamento de dados mais básicas desta biblioteca é a chamada sessão.

Novamente, este é apenas um conceito inicial para um primeiro contato, porém, na prática será muito mais interessante de se entender de fato o que é uma sessão em TensorFlow.

Basicamente podemos dizer que uma sessão, desculpe a redundância, é de fato uma sessão criada para que neste espaço sejam de fato executadas as ações de nossos blocos de código.

As estruturas entendidas até então como tensores, variáveis e constantes, são estruturas de código onde abstraímos tudo o que é necessário em forma de código e internamente o interpretador do TensorFlow organiza tais estruturas de dados como elementos de uma equação a ser executada, porém o processamento de tais dados efetivamente acontece ao abrirmos uma sessão que instancia os mesmos.

Em outras palavras, graças a estrutura modular do TensorFlow, uma determinada arquitetura pode ser desmembrada em pequenos blocos que podem inclusive serem processados separadamente e paralelamente por um ou mais hardwares dedicados.

Neste processo de se modularizar tudo, cada estrutura de dado, cada método, cada interação entre dados pode ser separada não somente em blocos de código, mas módulos de processamento individual, de forma que apenas quando necessário, os módulos são ativados e postos para interagir uns com os outros.

Imagine um daqueles brinquedos modulares muito usados para o aprendizado básico em robótica, você tem um núcleo básico e ele funciona sozinho por si só, a medida que quer implementar maiores funcionalidades ao robô basta adicionar as devidas peças e elas imediatamente se integram ao sistema e começam a executar suas ações.

A arquitetura interna desenvolvida pelos engenheiros da Google Brain funciona muito pautada neste tipo de modelo, onde podemos ter um modelo de rede neural artificial ao mesmo tempo muito simples como muito robusta, apenas adicionando mais funcionalidade modularizadas a arquitetura da rede neural.

Apenas encerrando essa linha de raciocínio, tenha em mente que para as versões a partir da 2.X da biblioteca TensorFlow todo esse processo é realizado de forma implícita, versões anteriores a esta demandavam que o usuário criasse manualmente cada sessão para cada fluxo de dado.

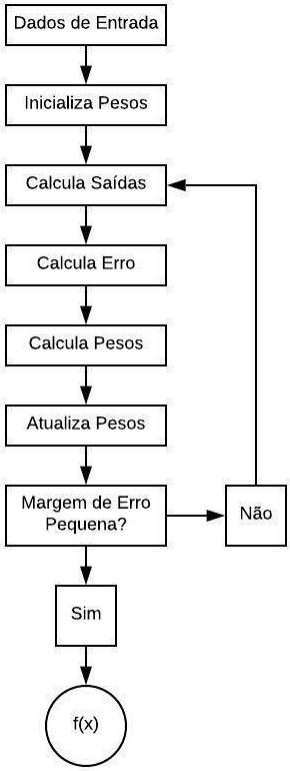
Rotinas de uma Rede Neural Artificial

Independentemente da área que estivermos falando sempre costumamos dividir nossas atribuições em rotinas que executaremos ao longo de um período de tempo, é assim em nossa rotina de trabalho, de estudos, até mesmo de tentar estabelecer padrões de como aproveitar melhor o pouco tempo que temos longe destes… Sendo assim, uma rotina basicamente será aquela sequência de processos que estaremos executando sempre dentro de uma determinada atividade. Em machine learning, no âmbito profissional, teremos diversos tipos de problemas a serem contextualizados para nossas redes neurais e sendo assim teremos uma rotina a ser aplicada sobre esses dados, independente de qual seja o resultado esperado.

No primeiro dataset que usaremos de exemplo para apresentar uma rede neural mais robusta, estaremos executando uma rotina bastante completa que visa extrair o máximo de informações a partir de um banco de dados pré-estabelecido, assim como estaremos criando nossa rede neural em um modelo onde determinados processos serão divididos em blocos que poderão ser reutilizáveis se adaptados para outros contextos posteriormente. Tenha em mente que uma vez criadas nossas ferramentas não há necessidade de cada vez criá-las a partir do zero, estaremos elucidando quais rotinas de processos melhor se adaptam de acordo com os tipos de problema computacional apresentado e dessa forma, uma vez entendido os conceitos e codificadas suas ferramentas, posteriormente você poderá simplesmente as reajustar de acordo com a necessidade.

Trabalhando sobre a Breast Cancer Dataset, primeira rede neural mais robusta que estaremos criando e entendendo seus conceitos passo a passo a seguir, estaremos executando uma determinada rotina que não necessariamente se aplica a todos os demais problemas computacionais, mas já servirá de exemplo para criar uma boa bagagem de conhecimento sobre o processo de importação e tratamento dos dados, criação e aplicação de uma rede neural artificial assim como a configuração da mesma a fim de obter melhores resultados. Raciocine que posteriormente em outros datasets estaremos trabalhando e usando outros modelos e ferramentas que não se aplicariam no contexto da Breast Dataset, porém começar por ela promoverá um entendimento bom o suficiente para que você entenda também de acordo com o problema, a rotina de processamento a ser aplicada.

Toda rede neural pode começar a ser contextualizada por um algoritmo bastante simples, onde entre nossas entradas e saídas temos:



Em algumas literaturas ainda podemos encontrar como rotina: Fase de criação e modelagem da rede > Fase supervisionada de reajustes da mesma > Fase de aprendizado de máquina > Fase de testes de performance da rede e dos resultados > Saídas.

Como mencionado nos capítulos anteriores, aqui usaremos de uma abordagem progressiva e bastante didática. Estaremos sim executando esses modelos de rotinas mencionados acima, porém de uma forma mais natural, seguindo a lógica a qual estaremos criando e aplicando linha a linha de código em nossa rede neural.

Sendo assim, partindo para prática estaremos com base inicial no Breast Cancer Dataset entendendo e aplicando cada um dos passos abaixo:

Rotina Breast Cancer Dataset

Rede neural simples:

Importação da base de dados de exemplo

Criação de uma função sigmoide de forma manual

Criação de uma função Sigmoide Derivada de forma manual

Tratamento dos dados, separando em atributos previsores e suas saídas

Criação de uma rede neural simples, camada a camada manualmente com aplicação de pesos aleatórios, seus reajustes em processo de aprendizado de máquina para treino do algoritmo.

Rede neural densa:

Importação da base de dados a partir de arquivo .csv

Tratamento dos dados atribuindo os mesmos em variáveis a serem usadas para rede de forma geral, assim como partes para treino e teste da rede neural.

Criação de uma rede neural densa multicamada que irá inicialmente classificar os dados a partir de uma série de camadas que aplicarão funções pré-definidas e parametrizadas por ferramentas de bibliotecas externas e posteriormente gerar previsões a partir dos mesmos.

Teste de precisão nos resultados e sobre a eficiência do algoritmo de rede neural.

Parametrização manual das ferramentas a fim de obter melhores resultados

Realização de técnicas de validação cruzada, e tuning assim como seus respectivos testes de eficiência.

Teste de viés de confirmação a partir de uma amostra.

Salvar o modelo para reutilização.

Posteriormente iremos trabalhar de forma mais aprofundada em outros modelos técnicas de tratamento e processamento de dados que não se aplicariam corretamente na Breast Cancer Dataset mas a outros tipos de bancos de dados.

Tenha em mente que posteriormente estaremos passo-a-passo trabalhando em modelos onde serão necessários um polimento dos dados no sentido de a partir de um banco de dados bruto remover todos dados faltantes, errôneos ou de alguma forma irrelevantes para o processo.

Estaremos trabalhando com um modelo de abstração chamado de variáveis do tipo Dummy onde teremos um modelo de classificação com devidas particularidades para classificação de muitas saídas.

Também estaremos entendendo como é feito o processamento de imagens por uma rede neural, neste processo, iremos realizar uma série de processos para conversão das informações de pixel a pixel da imagem para dados de uma matriz a ser processado pela rede.

Enfim, sob a sintaxe da linguagem Python e com o auxílio de algumas funções, módulos e bibliotecas estaremos abstraindo e contextualizando problemas da vida real de forma a serem solucionados de forma computacional por uma rede neural artificial.

APRENDIZADO DE MÁQUINA

Perceptron de Uma Camada – Modelo Simples

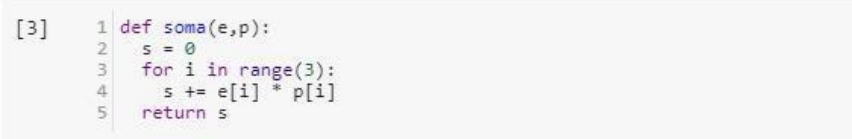
Anteriormente entendemos a lógica de um Perceptron, assim como as particularidades que estes possuem quanto a seu número de camadas, deixando um pouco de lado o referencial teórico e finalmente partindo para prática, a seguir vamos codificar alguns modelos de perceptron, já no capítulo seguinte entraremos de cabeça na codificação de redes neurais artificiais.

Partindo para o Código:

Inicialmente vamos fazer a codificação do algoritmo acima, pondo em prática o mesmo, através do Google Colaboratory (sinta-se livre para usar o Colaboratory ou o Jupyter da suíte Anaconda), ferramentas que nos permitirão tanto criar este perceptron quanto o executar em simultâneo. Posteriormente em problemas que denotarão maior complexidade estaremos usando outras ferramentas.

Abrindo nosso Colaboratory inicialmente renomeamos nosso notebook para Perceptron 1 Camada.ipynb apenas por convenção. *Ao criar um notebook Python 3 em seu Colaboratory automaticamente será criada uma pasta em seu Drive de nome Colab Notebooks, onde será salva uma cópia deste arquivo para posterior utilização.

Por parte de código apenas foi criado na primeira célula um comentário, na segunda célula foram declaradas duas variáveis de nomes entradas e pesos, respectivamente. Como estamos atribuindo vários valores para cada uma passamos esses valores em forma de lista, pela sintaxe Python, entre chaves e separados por vírgula. Aqui estamos usando os mesmos valores usados no modelo de perceptron anteriormente descrito, logo, entradas recebe os valores 1, 9 e 5 referentes aos nós de entrada X, Y e Z de nosso modelo assim como pesos recebe os valores 0.8, 0.1 e 0 como no modelo.



Em seguida criamos uma função de nome soma que recebe como parâmetro as variáveis temporárias e e p. Dentro dessa função inicialmente criamos uma nova variável de nome s que inicializa com valor 0. Em seguida criamos um laço de repetição que irá percorrer todos valores de e e p, realizar a sua multiplicação e atribuir esse valor a variável s. Por fim apenas deixamos um comando para retornar s, agora com valor atualizado.

```
[4]  1 s = soma(entradas,pesos)

[5]  1 print(s)

     1.7000000000000002
```

Criamos uma variável de nome s que recebe como atributo a função soma, passando como parâmetro as variáveis entradas e pesos. Executando uma função print( ) passando como parâmetro a variável s finalmente podemos ver que foram feitas as devidas operações sobre as variáveis, retornando o valor de 1.7, confirmando o modelo anterior.

```
[6]  1 def stepFunction(s):
     2   if (s >= 1):
     3     return 1
     4   return 0
```

Assim como criamos a função de soma entre nossas entradas e seus respectivos pesos, o processo final desse perceptron é a criação de nossa função degrau. Função essa que simplesmente irá pegar o valor retornado de soma e estimar com base nesse valor se esse neurônio será ativado ou não.

Para isso simplesmente criamos outra função, agora de nome stepFunction( ) que recebe como parâmetro s. Dentro da função simplesmente criamos uma estrutura condicional onde, se o valor de s for igual ou maior que 1, retornará 1 (referência para ativação), caso contrário, retornará 0 (referência para não ativação).

```
[7]  1 saida = stepFunction(s)

[8]  1 print(saida)

     1
```

Em seguida criamos uma variável de nome saida que recebe como atributo a função stepFunction( ) que por sua vez tem como parâmetro s. Por fim executamos uma função print( ) agora com saida como parâmetro, retornando finalmente o valor 1, resultando como previsto, na ativação deste perceptron.

Aprimorando o Código:

Como mencionado nos capítulos iniciais, uma das particularidades por qual escolhemos desenvolver nossos códigos em Python é a questão de termos diversas bibliotecas, módulos e extensões que irão facilitar nossa vida oferecendo ferramentas que não só automatizam, mas melhoram processos no que diz respeito a performance. Não existe problema algum em mantermos o código acima usando apenas os recursos do interpretador do Python, porém se podemos realizar a mesma tarefa de forma mais reduzida e com maior performance por quê não o fazer.

Sendo assim, usaremos aplicado ao exemplo atual alguns recursos da biblioteca Numpy, de forma a usar seus recursos internos para tornar nosso código mais enxuto e eficiente. Aqui trabalharemos pressupondo que você já instalou tal biblioteca como foi orientado em um capítulo anterior.

```
[1]  1 import numpy as np
```

Sempre que formos trabalhar com bibliotecas que nativamente não são carregadas e pré-alocadas por nossa IDE precisamos fazer a importação das mesmas. No caso da Numpy, uma vez que essa já está instalada no sistema, basta executarmos o código import numpy para que a mesma seja carregada e possamos usufruir de seus recursos. Por convenção quando importamos alguns tipos de biblioteca ou módulos podemos as referenciar por alguma abreviação, simplesmente para facilitar sua chamada durante o código. Logo, o comando import numpy as np importa a biblioteca Numpy e sempre que a formos usar basta iniciar o código com np.

```
[2]  1 entradas = np.array([1, 9, 5])
     2 pesos = np.array([0.8, 0.1, 0])
```

Da mesma forma que fizemos anteriormente, criamos duas variáveis que recebem os respectivos valores de entradas e pesos. Porém, note a primeira grande diferença de código, agora não passamos uma simples lista de valores, agora criamos uma array Numpy para que esses dados sejam vetorizados e trabalhados internamente pelo Numpy. Ao usarmos o comando np.array( ) estamos criando um vetor ou matriz, dependendo da situação, para que as ferramentas internas desta biblioteca possam trabalhar com os mesmos.

```
[3]  1 def Soma(e,p):
     2   return e.dot(p)
```

Segunda grande diferença agora é dada em nossa função de Soma, repare que agora ela simplesmente contém como parâmetros e e p, e ela executa uma única linha de função onde ela retornará o produto escalar de e sobre p. Em outras palavras, o comando e.dot(p) fará o mesmo processo aritmético que fizemos manualmente, de realizar as multiplicações e somas dos valores de e com p, porém de forma muito mais eficiente.

```
[4]  1 s = Soma(entradas, pesos)
     2
     3 def stepFunction(Soma):
     4   if (s >= 1):
     5     return 1
     6   return 0
     7
     8 saida = stepFunction(s)
     9
    10 print(s)
    11 print(saida)
```

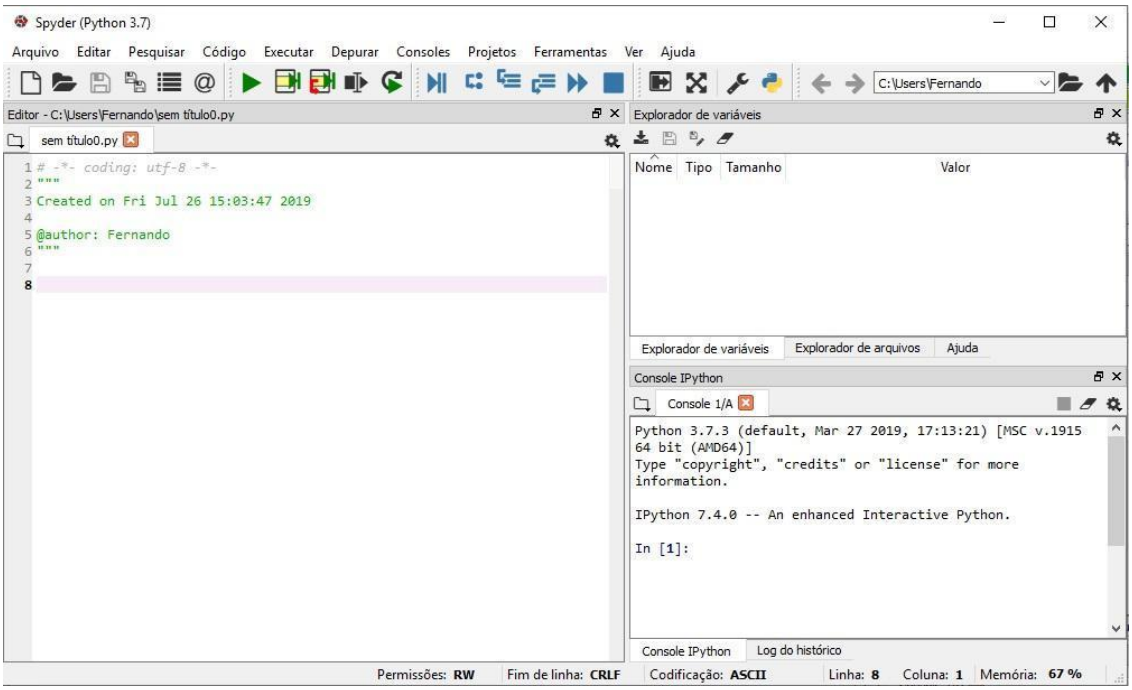
```
1.7000000000000002
1
```

Todo o resto do código é reaproveitado e executado da mesma forma, obtendo inclusive como retorno os mesmos resultados (o que é esperado), a diferença de trocar uma função básica, baseada em listas e condicionais, por um produto escalar realizado por módulo de uma biblioteca dedicada a isto torna o processo mais eficiente. Raciocine que à medida que formos implementando mais linhas de código com mais funções essas pequenas diferenças de performance realmente irão impactar o desempenho de nosso código final.

Usando o Spyder 3:

Os códigos apresentados anteriormente foram rodados diretamente no notebook Colab do Google, porém haverão situações de código mais complexo onde não iremos conseguir executar os códigos normalmente a partir de um notebook. Outra ferramenta bastante usada para manipulação de dados em geral é o Spyder. Dentro dele podemos criar e executar os mesmos códigos de uma forma um pouco diferente graças aos seus recursos. Por hora, apenas entenda que será normal você ter de se familiarizar com diferentes ferramentas uma vez que algumas situações irão requerer mais ferramentas específicas.

Tratando-se do Spyder, este será o IDE que usaremos para praticamente tudo a partir daqui, graças a versatilidade de por meio dele podermos escrever nossas linhas de código, executá-las individualmente e em tempo real e visualizar os dados de forma mais intuitiva.



Abrindo o Spyder diretamente pelo seu atalho ou por meio da suíte Anaconda nos deparamos com sua tela inicial, basicamente o Spyder já vem pré-configurado de forma que possamos simplesmente nos dedicar ao código, talvez a única configuração que você deva fazer é para sua própria comodidade modificar o caminho onde serão salvos os arquivos.

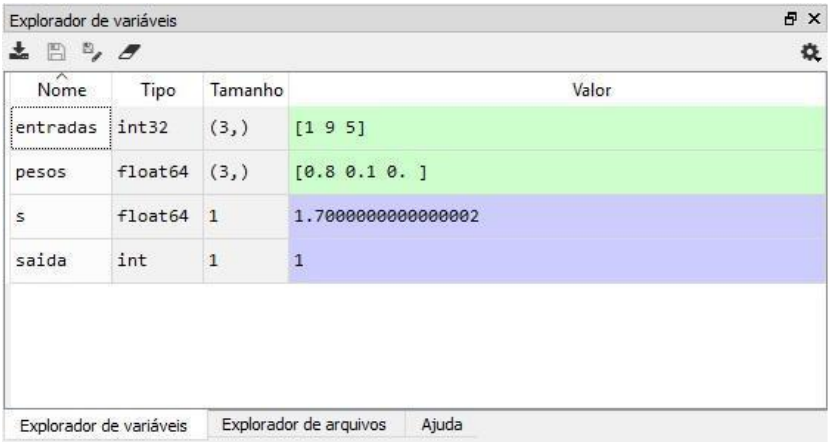
A tela inicial possui a esquerda um espaço dedicado ao código, e a direita um visualizador de variáveis assim como um console que mostra em tempo real certas execuções de blocos de código.

Apenas como exemplo, rodando este mesmo código criado anteriormente, no Spyder podemos em tempo real fazer a análise das operações sobre as variáveis, assim como os resultados das mesmas via terminal.

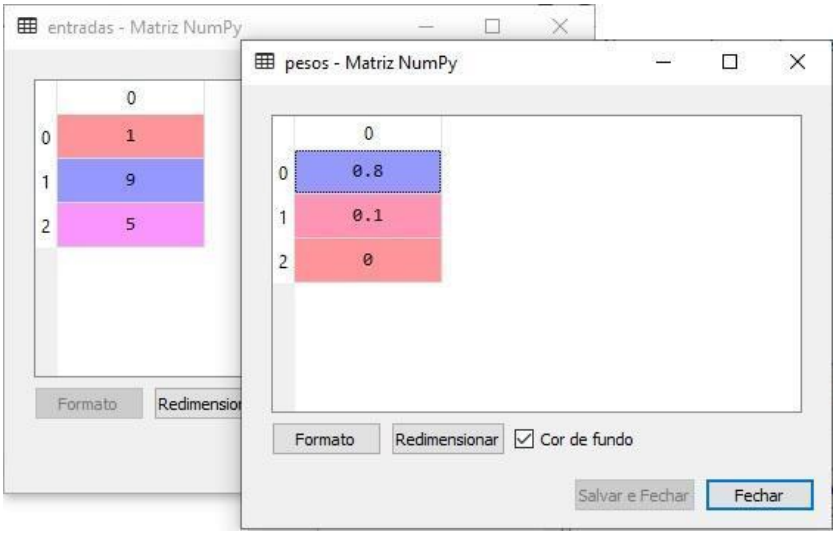
Perceptron 1 Camada Otimizado.py*

```
1 import numpy as np
2
3 entradas = np.array([1, 9, 5])
4 pesos = np.array([0.8, 0.1, 0])
5
6 def Soma(e,p):
7   return e.dot(p)
8
9 s = Soma(entradas, pesos)
10
11 def stepFunction(Soma):
12   if (s >= 1):
13     return 1
14   return 0
15
16 saida = stepFunction(s)
17
18 print(s)
19 print(saida)
20
```

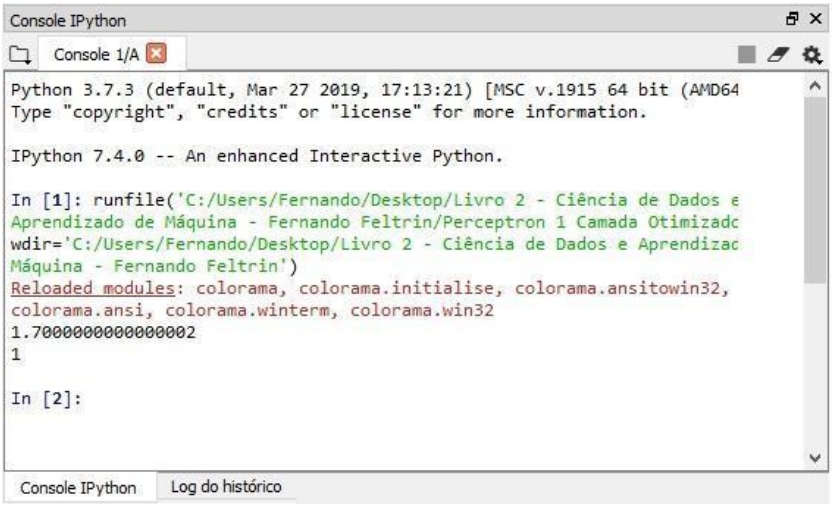
Explorador de Variáveis:



Visualizando Variáveis:



Terminal:



Perceptron de Uma Camada – Tabela AND

Entendidos os conceitos lógicos de o que é um perceptron, como os mesmos são um modelo para geração de processamento de dados para por fim ser o parâmetro de uma função de ativação. Hora de, também de forma prática, entender de fato o que é aprendizagem de máquina.

Novamente se tratando de redes neurais serem uma abstração as redes neurais biológicas, o mecanismo de aprendizado que faremos também é baseado em tal modelo. Uma das características de nosso sistema nervoso central é a de aprender criando padrões de conexões neurais para ativação de alguma função ou processamento de funções já realizadas (memória). Também é válido dizer que temos a habilidade de aprender e melhorar nosso desempenho com base em repetição de um determinado tipo de ação.

Da mesma forma, criaremos modelos computacionais de estruturas neurais onde, programaremos determinadas funções de forma manual (fase chamada supervisionada) e posteriormente iremos treinar tal rede para que identifique o que é certo, errado, e memorize este processo. Em suma, a aprendizagem de máquina ocorre quando uma rede neural atinge uma solução generalizada para uma classe de problemas.

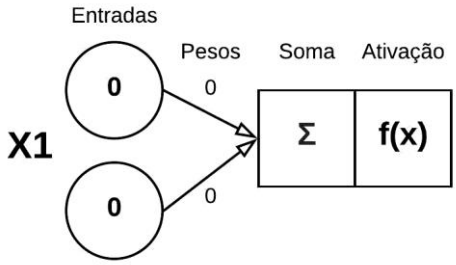
Partindo para prática, vamos criar do zero uma rede neural que aprenderá o que é o mecanismo lógico de uma tabela AND. Operador lógico muito usado em problemas que envolvem simples tomada de decisão. Inicialmente, como é de se esperar, iremos criar uma estrutura lógica que irá processar os valores mas de forma errada, sendo assim, na chamada fase supervisionada, iremos treinar nossa rede para que ela aprenda o padrão referencial correto e de fato solucione o que é o processamento das entradas e as respectivas saídas de uma tabela AND.

Tabela AND:

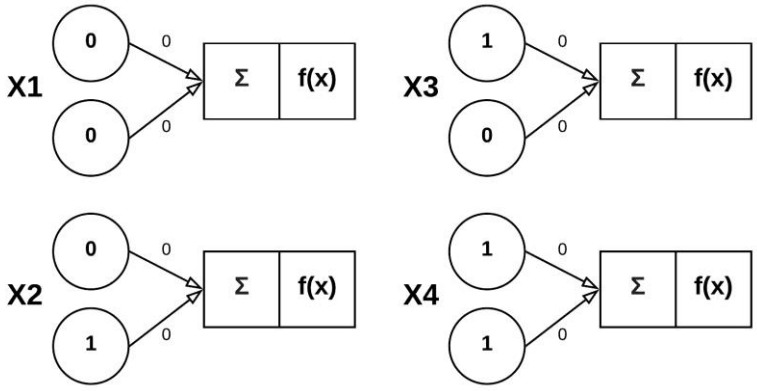
X1	X2	
0	0	0
0	1	0
1	0	0
1	1	1

Todo mundo em algum momento de seu curso de computação teve de estudar operadores lógicos e o mais básico deles é o operador AND. Basicamente ele faz a relação entre duas proposições e apenas retorna VERDADEIRO caso os dois valores de entrada forem verdadeiros. A tabela acima nada mais é do que a tabela AND em operadores aritméticos, mas o mesmo conceito vale para True e False. Sendo 1 / True e 0 / False, apenas teremos como retorno 1 caso os dois operandos forem 1, assim como só será True se as duas proposições forem True.

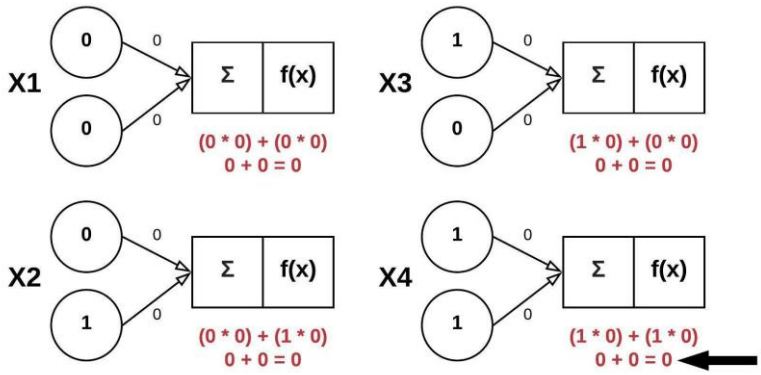
Dessa forma, temos 4 operadores a primeira coluna, mais 4 operadores na segunda camada e na última coluna os resultados das relações entre os mesmos. Logo, temos o suficiente para criar um modelo de rede neural que aprenderá essa lógica.



Basicamente o que faremos é a criação de 4 perceptrons onde cada um terá 2 neurônios de entrada, pesos que iremos atribuir manualmente, uma função soma e uma função de ativação.



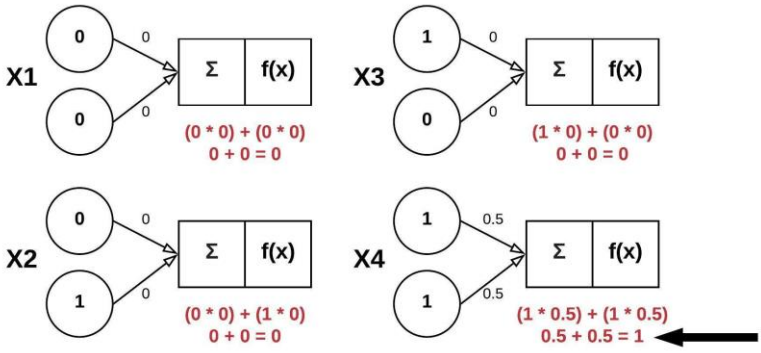
Montada a estrutura visual dos perceptrons, por hora, apenas para fins didáticos, hora de realizar as devidas operações de multiplicação entre os valores de entrada e seus respectivos pesos. Novamente, vale salientar que neste caso em particular, para fins de aprendizado, estamos iniciando essas operações com pesos zerados e definidos manualmente, esta condição não se repetirá em outros exemplos futuros.



Vamos ao modelo, criamos 4 estruturas onde X1 representa a primeira linha de nossa tabela AND (0 e 0), X2 que representa a segunda linha (0 e 1), X3 que representa a terceira linha (1 e 0) e por fim X4 que representa a quarta linha do operador (1 e 1).

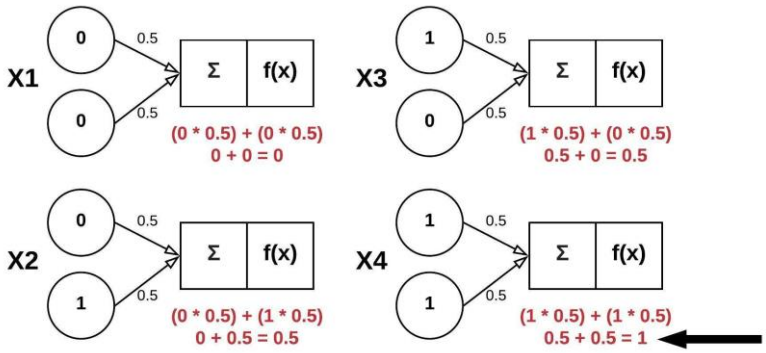
Executando a função de soma dos mesmos, repare que para X1 a multiplicação das entradas pelos respectivos pesos e a soma destes resultou no valor 0, que era esperado nesse caso (Tabela AND - 0 e 0 = 0). O processo se repete exatamente igual aos perceptrons X2 e X3 onde a função soma resulta em 0. Porém, repare em X4, a multiplicação das entradas pelos respectivos pesos como manda a regra resultou em 0, mas sabemos que de acordo com a tabela verdade este resultado deveria ser 1. Isso se deu porque simplesmente os pesos 0 implicaram em uma multiplicação falha, qualquer valor multiplicado por 0 resulta em 0.

Precisamos então treinar nossa rede para que ao final do processo se chegue ao valor esperado (X4 = 1). Em redes neurais básicas como estas, o que faremos é o simples reajuste dos valores dos pesos e a repetição do processo para ver se o mesmo é corrigido.



Revisando apenas o perceptron X4, alterando os valores dos pesos de 0 para 0.5 e repetindo a função de soma, agora sim temos o valor esperado 1. (Tabela AND 1 e 1 = 1.

Uma vez encontrado o valor de pesos que solucionou o problema a nível de X4, hora de aplicar este mesmo valor de pesos a toda rede neural e verificar se este valor funciona para tudo.



Aplicando o valor de peso 0.5 a todos neurônios e refeitos os devidos cálculos dentro de nossa função soma, finalmente temos os valores corrigidos. Agora seguindo a lógica explicada anteriormente, como os valores de X2 e X3 são menores que 1, na step function os mesmos serão reduzidos a 0, sendo apenas X4 o perceptron ativado nessa rede. o que era esperado de acordo com a tabela AND.

Em X1 0 e 0 = 0, em X2 0 e 1 = 0, em X3 1 e 0 = 0 e em X4 1 e 1 = 1.

Partindo para o Código:

```
[1]  1 import numpy as np
```

Por parte do código, todo processo sempre se inicia com a importação das bibliotecas e módulos necessários para que possamos fazer o uso deles. Nesse caso todas as operações que faremos serão operações nativas do Python ou funções internas da biblioteca Numpy. Para sua importação basta executar o comando import numpy, por convenção também as referenciaremos como np por meio do comando as, dessa forma, sempre que quisermos "chamar" alguma função sua basta usar np.

```
[2]  1 entradas = np.array([[0,0],[0,1],[1,0],[1,1]])
     2 saidas = np.array([0,0,0,1])
     3 pesos = np.array([0.0,0.0])
```

Em seguida criamos três variáveis que ficarão responsáveis por guardar os valores das entradas, saídas e dos pesos. Para isso declaramos uma nova variável de nome entradas que recebe como atributo uma array numpy através do comando np.array que por sua vez recebe como parâmetros os valores de entrada. Repare na sintaxe e também que para cada item dessa lista estão contidos os valores X1 e X2, ou seja, cada linha de nossa tabela AND. Da mesma forma criamos uma variável saidas que recebe uma array numpy com uma lista dos valores de saída. Por fim criamos uma variável pesos que recebe uma array numpy de valores iniciais zerados. Uma vez que aqui neste exemplo estamos explorando o que de fato é o aprendizado de máquina, usaremos pesos inicialmente zerados que serão corrigidos posteriormente.

```
[3]  1 taxaAprendizado = 0.5
```

Logo após criamos uma variável de nome taxaAprendizado que como próprio nome sugere, será a taxa com que os valores serão atualizados e testados novamente nesse perceptron. Você pode testar valores diferentes, menores, e posteriormente acompanhar o processo de aprendizagem.

```
[4]  1 def Soma(e,p):
     2     return e.dot(p)
```

Na sequência criamos a nossa função Soma, que tem como parâmetro as variáveis temporárias e e p, internamente ela simplesmente tem o comando de retornar o produto escalar de e sobre p, em outras palavras, a soma das multiplicações dos valores atribuídos as entradas com seus respectivos pesos.

```
[5]  1 s = Soma(entradas, pesos)
```

Criamos uma variável de nome s que como atributo recebe a função Soma que agora finalmente possui como parâmetro as variáveis entradas e pesos. Dessa forma, podemos instanciar a variável s em qualquer bloco de código que ela imediatamente executará a soma das entradas pelos pesos, guardando seus dados internamente.

```
[6]  1 def stepFunction(soma):
     2   if (soma >= 1):
     3     return 1
     4   return 0
```

Seguindo com o código, criamos nossa stepFunction, em tradução livre, nossa função degrau, aquela que pegará os valores processados para consequentemente determinar a ativação deste perceptron ou não em meio a rede. Para isso definimos uma função stepFunction que tem como parâmetro a variável temporária soma. Internamente criamos uma estrutura condicional básica onde, se soma (o valor atribuído a ela) for igual ou maior que 1, retorna 1 (resultando na ativação), caso contrário retornará 0 (resultando na não ativação do mesmo).

```
[7]  1 def calculoSaida(reg):
     2   s = reg.dot(pesos)
     3   return stepFunction(s)
```

Assim como as outras funções, é preciso criar também uma função que basicamente pegará o valor processado e atribuído como saída e fará a comparação com o valor que já sabemos de saída da tabela AND, para que assim se possa identificar onde está o erro, subsequentemente realizar as devidas modificações para se se aprenda qual os valores corretos de saída. Simplesmente criamos uma função, agora de nome calculoSaida que tem como parâmetro a variável temporária reg, internamente ela tem uma variável temporária s que recebe como atributo o produto escalar de reg sobre pesos, retornando esse valor processado pela função degrau.

```
[8]  1 def aprendeAtualiza():
     2   erroTotal = 1
     3   while (erroTotal != 0):
     4     erroTotal = 0
     5     for i in range (len(saidas)):
     6       calcSaida = calculoSaida(np.array(entradas[i]))
     7       erro = abs(saidas[i] - calcSaida)
     8       erroTotal += erro
     9       for j in range(len(pesos)):
    10         pesos[j] = pesos[j] + (taxaAprendizado * entradas[i][j] * erro)
    11         print('Pesos Atualizados> ' + str(pesos[j]))
    12     print('Total de Erros: ' +str(erroTotal))
```

Muita atenção a este bloco de código, aqui será realizado o processo de aprendizado de máquina propriamente dito.

Inicialmente criamos uma função de nome aprendeAtualiza, sem parâmetros mesmo. Internamente inicialmente criamos uma variável de nome erroTotal, com valor 1 atribuído simbolizando que existe um erro a ser corrigido.

Em seguida criamos um laço de repetição que inicialmente tem como parâmetro a condição de que se erroTotal for diferente de 0 será executado o bloco de código abaixo. Dentro desse laço erroTotal tem seu valor zerado, na sequência uma variável i percorre todos valores de saidas, assim como uma nova variável calcSaida recebe os valores de entrada em forma de array numpy, como atributo de calculoSaida. É criada também uma variável de nome erro que de forma absoluta (sem considerar o sinal negativo) recebe como valor os dados de saidas menos os de calcSaida, em outras palavras, estamos fazendo nesta linha de código a diferença entre os valores de saída reais que já conhecemos na tabela AND e os que foram encontrados inicialmente pelo perpeptron. Para finalizar esse bloco de código erroTotal soma a si próprio o valor encontrado e atribuído a erro.

Na sequência é criado um novo laço de repetição onde uma variável j irá percorrer todos valores de pesos, em sequida pesos recebe como atributo seus próprios valores multiplicados pelos de taxaAprendizado (definido manualmente anteriormente como 0.5), multiplicado pelos valores de entradas e de erro. Para finalizar ele recebe dois comandos print( ) para que no console/terminal sejam exibidos os valores dos pesos atualizados assim como o total de erros encontrados.

```
[9]  1 aprendeAtualiza()

Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.0
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Total de Erros: 1
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Total de Erros: 0
```

Por fim simplesmente executando a função aprendeAtualiza( ) podemos acompanhar via console/terminal o processo de atualização dos pesos (aprendizado de máquina) e o log de

quando não existem mais erros, indicando que tal perceptron finalmente foi treinado e aprendeu de fato a interpretar uma tabela AND, ou em outras palavras, reconhecer o padrão correto de suas entradas e saídas.

Usando Spyder:

Perceptron 1 Camada Tabela AND PRONTO.py*

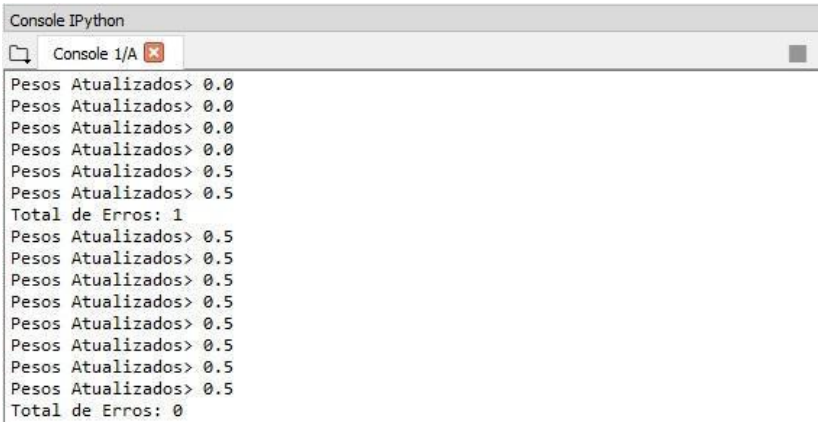
```
1 import numpy as np
2
3 entradas = np.array([[0,0],[0,1],[1,0],[1,1]])
4 saidas = np.array([0,0,0,1])
5 pesos = np.array([0.0,0.0])
6
7 taxaAprendizado = 0.5
8
9 def Soma(e,p):
10     return e.dot(p)
11
12 s = Soma(entradas, pesos)
13
14 def stepFunction(soma):
15   if (soma >= 1):
16     return 1
17   return 0
18
19 def calculoSaida(reg):
20   s = reg.dot(pesos)
21   return stepFunction(s)
22
23 def aprendeAtualiza():
24   erroTotal = 1
25   while (erroTotal != 0):
26     erroTotal = 0
27     for i in range (len(saidas)):
28       calcSaida = calculoSaida(np.array(entradas[i]))
29       erro = abs(saidas[i] - calcSaida)
30       erroTotal += erro
31       for j in range(len(pesos)):
32         pesos[j] = pesos[j] + (taxaAprendizado * entradas[i][j] * erro)
33         print('Pesos Atualizados> ' + str(pesos[j]))
34     print('Total de Erros: ' +str(erroTotal))
35
36 aprendeAtualiza()
```

Explorador de Variáveis:

Explorador de variáveis

Nome	Tipo	Tamanho	Valor
entradas	int32	(4, 2)	[[0 0] [0 1]
pesos	float64	(2,)	[0.5 0.5]
s	float64	(4,)	[0. 0. 0. 0.]
saidas	int32	(4,)	[0 0 0 1]
taxaAprendizado	float	1	0.5

Console:



Perceptron Multicamada – Tabela XOR

Entendida a lógica do processo de aprendizado de máquina na prática, treinando um perceptron para que aprenda um operador lógico AND, hora de aumentar levemente a complexidade. Como dito anteriormente, gradualmente vamos construindo essa bagagem de conhecimento. O principal diferencial deste capítulo em relação ao anterior é que, em uma tabela AND temos um problema linearmente separável, onde a resposta (saída) era basicamente 1 ou 0, pegando o exemplo de uma tabela XOR temos o diferencial de que este operador lógico realiza a operação lógica entre dois operandos, que resulta em um valor lógico verdadeiro se e somente se o número de operandos com valor verdadeiro for ímpar, dessa forma temos um problema computacional onde, a nível de perceptron, teremos de treinar o mesmo para descobrir a probabilidade desta operação ser verdadeira.

Em outras palavras, teremos de identificar e treinar nosso perceptron para que realize mais de uma camada de correção de pesos, afim de identificar a probabilidade correta da relação entre as entradas e as saídas da tabela XOR que por sua vez possui mais de um parâmetro de ativação. O fato de trabalharmos a partir daqui com uma camada a mais de processamento para rede, camada essa normalmente chamada de camada oculta, permite que sejam aplicados “filtros” que nos ajudam a identificar, treinar e aprender sob nuances de valores muito menores do que os valores brutos do exemplo anterior.

Tabela XOR:

X1	X2	
0	0	0
0	1	1
1	0	1
1	1	0

Estrutura Lógica:

Diretamente ao código:

Perceptron Multicamada XOR.py

```
1 import numpy as np
2
```

Como de costume, todo processo se inicia com a importação das bibliotecas e módulos que utilizaremos ao longo de nossa rede. Agora trabalhando diretamente com o Spyder, podemos em tempo real executar linhas ou blocos de código simplesmente os selecionando com o mouse e aplicando o comando Ctrl + ENTER. Para importação da biblioteca Numpy basta executar o comando import como exemplificado acima.

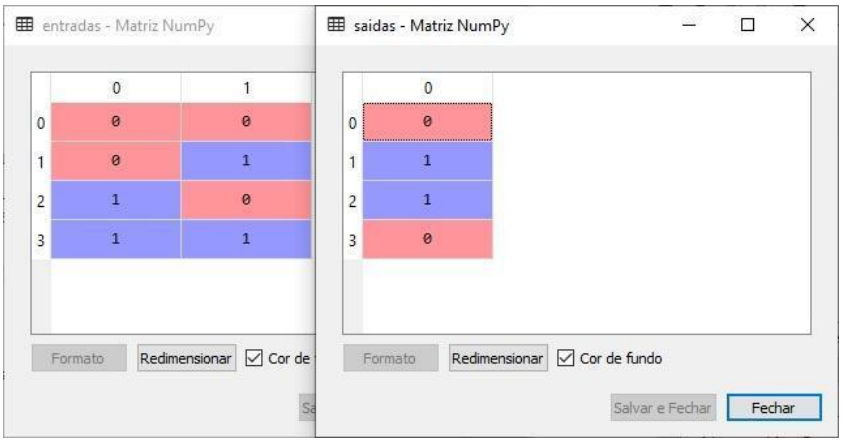
```
3 entradas = np.array([[0,0],
4                      [0,1],
5                      [1,0],
6                      [1,1]])
7 saidas = np.array([[0],[1],[1],[0]])
```

Em seguida criamos as variáveis entradas e saidas que como seus nomes já sugerem, recebem como atributos os respectivos dados da tabela XOR, em forma de array numpy, assim, uma vez vetorizados, poderemos posteriormente aplicar funções internas desta biblioteca.

Explorador de variáveis

Nome	Tipo	Tamanho	Valor
entradas	int32	(4, 2)	[[0 0] [0 1]
saidas	int32	(4, 1)	[[0] [1]

Executando esse bloco de código (via Ctrl + ENTER) podemos ver que ao lado direito da interface, no Explorador de variáveis são criadas as respectivas variáveis.



Clicando duas vezes sobre as mesmas podemos as explorar de forma visual, em forma de matrizes como previsto, lado a lado compondo a tabela XOR.

```
pesos0 = np.array([[-0.424, -0.740, -0.961],
                   [0.358, -0.577, -0.469]])
pesos1 = np.array([[-0.017], [-0.893], [0.148]])
```

Em seguida criamos as variáveis dedicadas a guardar os valores dos pesos, aqui, novamente apenas para fins de exemplo, estamos iniciando essas variáveis com valores aleatórios.

Nome	Tipo	Tamanho	Valor
entradas	int32	(4, 2)	[[0 0] [0 1] ...
pesos0	float64	(2, 3)	[[-0.424 -0.74 -0.961] [0.358 -0.577 -0.469]]
pesos1	float64	(3, 1)	[[-0.017] [-0.893] ...
saidas	int32	(4, 1)	[[0] [1] ...

Se a sintaxe estiver correta, ao selecionar e executar este bloco de código podemos ver no explorador de variáveis que tais variáveis foram corretamente criadas.

```
ntreinos = 100
taxaAprendizado = 0.3
momentum = 1
```

Em seguida criamos estas três variáveis auxiliares que nos serão úteis posteriormente. A variável ntreinos, que aqui recebe como atributo o valor 100, diz respeito ao parâmetro de quantas vezes a rede será executada para que haja o devido ajuste dos pesos. Da mesma forma a variável taxaAprendizado aqui com valor 0.3 atribuído é o parâmetro para de quantos em quantos números os pesos serão modificados durante o processo de aprendizado, esse parâmetro, entendido em outras literaturas como a velocidade em que o algoritmo irá aprender, conforme o valor pode inclusive fazer com que o algoritmo piore sua eficiência ou entre em um loop onde fica estagnado em um ponto, logo, é um parâmetro a ser testado com outros valores e comparar a eficiência dos resultados dos mesmos. Por fim momentum é um parâmetro padrão, uma constante na fórmula de correção da margem de erro, nesse processo de atualização de pesos, aqui, para este exemplo, segue com o valor padrão 1 atribuído, mas pode ser testado com outros valores para verificar se as últimas alterações surtiram melhoria da eficiência do algoritmo.

```
def sigmoid(soma):
    return 1 / (1 + np.exp(-soma))
```

Logo após criamos nossa Função Sigmoide, que neste modelo, substitui a Função Degrau que utilizamos anteriormente. Existem diferentes funções de ativação, as mais comuns, Degrau e Sigmoide utilizaremos com frequência ao longo dos demais exemplos. Em suma, a grande diferença entre elas é que a Função Degrau retorna valores absolutos 0 ou 1 (ou valores definidos pelo usuário, mas sempre neste padrão binário, um ou outro), enquanto a Função Sigmoide leva em consideração todos valores intermediários entre 0 e 1, dessa forma, conseguimos verificar a probabilidade de um dado valor se aproximar de 0 ou se aproximar de 1, de acordo com suas características.

Para criar a função Sigmoide, simplesmente definimos sigmoid( ) que recebe como parâmetro a variável temporária soma, internamente ela simplesmente retorna o valor 1 dividido pela soma de 1 pela exponenciação de soma, o que na prática nos retornará um valor float entre 0 e 1 (ou seja, um número com casas decimais).

```
20 for i in range(ntreinos):
21     camadaEntrada = entradas
22     somaSinapse0 = np.dot(camadaEntrada, pesos0)
23     camadaOculta = sigmoid(somaSinapse0)
24 
25     somaSinapse1 = np.dot(camadaOculta, pesos1)
26     camadaSaida = sigmoid(somaSinapse1)
27 
28     erroCamadaSaida = saidas - camadaSaida
29     mediaAbsoluta = np.mean(np.abs(erroCamadaSaida))
```

Prosseguindo criamos o primeiro bloco de código onde de fato ocorre a interação entre as variáveis. Criamos um laço de repetição que de acordo com o valor setado em ntreinos executa as seguintes linhas de código. Inicialmente é criada uma variável temporária camadaEntrada que recebe como atributo o conteúdo de entradas, em seguida é criada uma variável somaSinapse0, que recebe como atributo o produto escalar (soma das multiplicações) de camadaEntrada por pesos0. Apenas relembrando, uma sinapse é a termologia da conexão entre neurônios, aqui, trata-se de uma variável que faz a interligação entre os nós da camada de entrada e da camada oculta. Em seguida é criada uma variável camadaOculta que recebe como atributo a função sigmoid( ) que por sua vez tem como parâmetro o valor resultante dessa operação sobre somaSinapse0.

Da mesma forma é necessário criar a estrutura de interação entre a camada oculta e a camada de saída. Para isso criamos a variável somaSinapse1 que recebe como atributo o produto escalar entre camadaOculta e pesos1, assim como anteriormente fizemos, é criada uma variável camadaSaida que por sua vez recebe como atributo o valor gerado pela função sigmoid( ) sobre somaSinapse1.

Por fim são criados por convenção duas variáveis que nos mostrarão parâmetros para avaliar a eficiência do processamento de nossa rede. erroCamadaSaida aplica a fórmula de erro, fazendo a simples diferença entre os valores de saidas (valores que já conhecemos) e camadaSaida (valores encontrados pela rede). Posteriormente criamos a variável mediaAbsoluta que simplesmente, por meio da função .mean( ) transforma os dados de erroCamadaSaida em um valor percentual (Quantos % de erro existe sobre nosso processamento).

Nome	Tipo	Tamanho	Valor
camadaEntrada	int32	(4, 2)	[[0 0] [0 1]
camadaOculta	float64	(4, 3)	[[0.5 0.5 0.5] [0.5885562 0.35962319 0.38485296 ...
camadaSaida	float64	(4, 1)	[[0.40588573] [0.43187857]
entradas	int32	(4, 2)	[[0 0] [0 1]
erroCamadaSaida	float64	(4, 1)	[[-0.40588573] [0.56812143]
i	int	1	99
mediaAbsoluta	float64	1	0.49880848923713045

Selecionando e executando esse bloco de código são criadas as referentes variáveis, das camadas de processamento assim como as variáveis auxiliares. Podemos clicando duas vezes sobre as mesmas visualizar seu conteúdo. Por hora, repare que neste primeiro processamento é criada a variável mediaAbsoluta que possui valor 0.49, ou seja 49% de erro em nosso processamento, o que é uma margem muito grande. Pode ficar tranquilo que valores altos nesta etapa de execução são perfeitamente normais, o que faremos na sequência é justamente trabalhar a realizar o aprendizado de máquina, fazer com que nossa rede identifique os padrões corretos e reduza esta margem de erro ao menor valor possível.

```
31 def sigmoideDerivada(sig):
32     return sig * (1-sig)
33 sigDerivada = sigmoid(0.5)
34 sigDerivada1 = sigmoideDerivada(sigDerivada)
```

Dando sequência criamos nossa função sigmoideDerivada( ) que como próprio nome sugere, faz o cálculo da derivada, que também nos será útil na fase de aprendizado de máquina. Basicamente o que fazemos é definir uma função sigmoideDerivada( ) que tem como parâmetro a variavel temporária sig. Internamente é chamada a função que retorna o valor obtido pela multiplicação de sig (do valor que estiver atribuído a esta variável) pela multiplicação de 1 menos o próprio valor de sig.

Em seguida, aproveitando o mesmo bloco de código dedicado a esta etapa, criamos duas variáveis sigDerivada, uma basicamente aplica a função sigmoid( ) com o valor de 0.5, a outra, aplica a própria função sigmoideDerivada( ) sobre sigDerivada.

```
36 derivadaSaida = sigmoideDerivada(camadaSaida)
37 deltaSaida = erroCamadaSaida * derivadaSaida
```

Da mesma forma, seguimos criando mais variáveis que nos auxiliarão a acompanhar o processo de aprendizado de máquina, dessa vez criamos uma variável derivadaSaida que aplica a função sigmoideDerivada( ) para a camadaSaida, por fim criamos a variável deltaSaida que recebe como atributo o valor da multiplicação entre erroCamadaSaida e derivadaSaida.

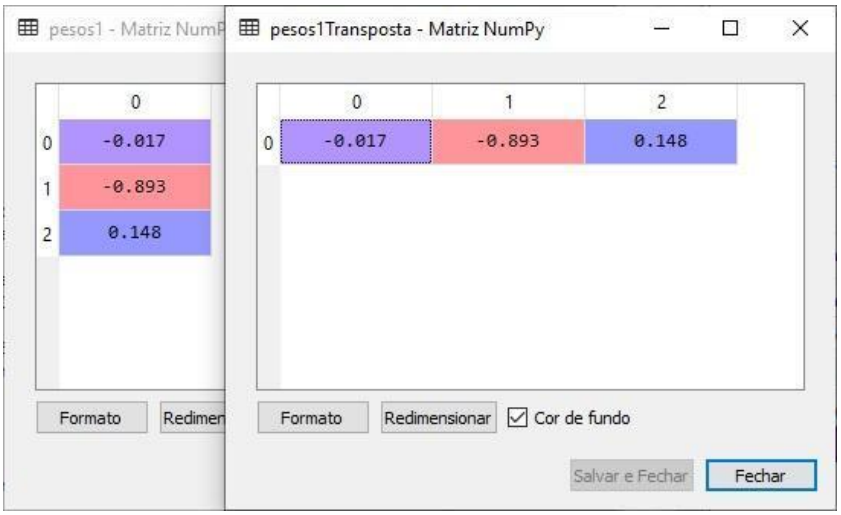
Vale lembrar que o que nomeamos de delta em uma rede neura normalmente se refere a um parâmetro usado como referência para correto ajuste da descida do gradiente. Em outras palavras, para se realizar um ajuste fino dos pesos e consequentemente conseguir minimizar a margem de erro dos mesmos, são feitos internamente uma série de cálculos para que esses reajustes sejam feitos da maneira correta, parâmetros errados podem prejudicar o reconhecimento dos padrões corretos por parte da rede neural.

Podemos sempre que quisermos, fazer a visualização das variáveis por meio do explorador, aqui, os dados encontrados executando a fórmula do cálculo da derivada sobre os valores de saída assim como os valores encontrados para o delta, em outras palavras, pelo sinal do delta podemos entender como (em que direção no grafo) serão feitos os reajustes dos pesos, uma vez que individualmente eles poder ter seus valores aumentados ou diminuídos de acordo com o processo.

```
39 pesos1Transposta = pesos1.T
40 deltaSaidaXpesos = deltaSaida.dot(pesos1Transposta)
41 deltaCamadaOculta = deltaSaidaXpesos * sigmoideDerivada(camadaOculta)
```

Dando sequência, teremos de fazer um reajuste de formado de nossos dados em suas devidas matrizes para que as operações aritméticas sobre as mesmas possam ser realizadas corretamente. Raciocine que quando estamos trabalhando com vetores e matrizes temos um padrão de linhas e colunas que pode ou não ser multiplicável. Aqui nesta fase do processo teremos inconsistência de formatos de nossas matrizes de pesos, sendo necessário realizar um processo chamado Matriz Transposta, onde transformaremos linhas em colunas e vice versa de forma que possamos realizar as operações e obter valores corretos.

Inicialmente criamos uma variável pesos1Transposta que recebe como atributo pesos1 com aplicação da função .T( ), função interna da biblioteca Numpy que realizará essa conversão. Em seguida criamos uma variável de nome deltaSaidaXpesos que recebe como atributo o produto escalar de deltaSaida por pesos1Transposta. Por fim aplicamos novamente a fórmula do delta, dessa vez para camada oculta, multiplicando deltaSaidaXpesos pelo resultado da função sigmoide sob o valor de camadaOculta.



Apenas visualizando a diferença entre pesos1 e pesos1Transposta (colunas transformadas em linhas).

Se você se lembra do algoritmo explicado em capítulos anteriores verá que até este momento estamos trabalhando na construção da estrutura desse perceptron, assim como sua alimentação com valores para seus nós e pesos. A esta altura, obtidos todos valores iniciais, inclusive identificando que tais valores resultam em erro se comparado ao esperado na tabela XOR, é hora de darmos início ao que para alguns autores é chamado de backpropagation, ou seja, realizar de fato os reajustes dos pesos e o reprocessamento do perceptron, repetindo esse processo até o treinar de fato para a resposta correta.

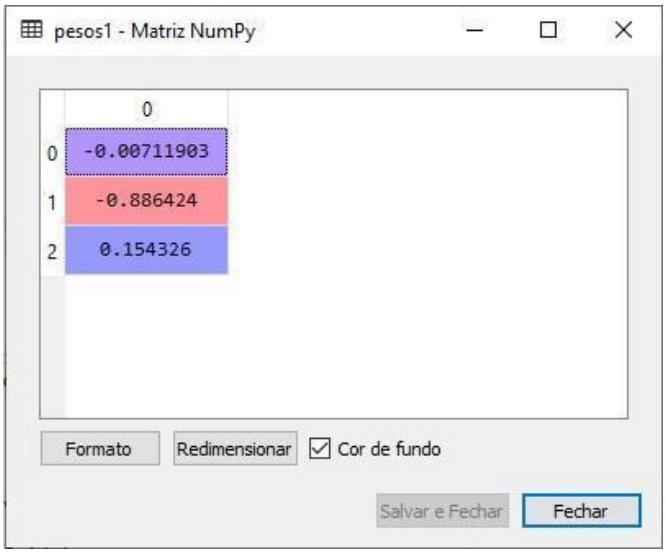
Todo processo realizado até o momento é normalmente tratado na literatura como feed forward, ou seja, partindo dos nós de entrada fomos alimentando e processando em sentido a saída, a ativação desse perceptron, a etapa de backpropagation como o nome já sugere, retroalimenta o perceptron, faz o caminho inverso ao feed forward, ou em certos casos, retorna ao ponto inicial e refaz o processamento a partir deste.

O processo de backpropagation inclusive tem uma fórmula a ser aplicada para que possamos entender a lógica desse processo como operações sobre nossas variáveis.

peso(n + 1) = (peso(n) + momento) + (entrada * delta * taxa de aprendizagem)

```
43 camadaOcultaTransposta = camadaOculta.T
44 pesos3 = camadaOcultaTransposta.dot(deltaSaida)
45 pesos1 = (pesos1 * momentum) + (pesos3 * taxaAprendizado)
```

Da mesma forma como fizemos anteriormente, criamos uma variável camadaOcultaTransposta que simplesmente recebe a camadaOculta de forma transposta por meio da função .T( ). Logo após usamos a variável pesos3 que recebe como atributo o produto escalar de camadaOcultaTransposta pelo deltaSaida. Finalmente, fazemos o processo de atualização dos valores de pesos1, atribuindo ao mesmo os valores atualizados de pesos1 multiplicado pelo momentum somado com os valores de pesos3 multiplicados pelo parâmetro de taxaAprendizado.



Executando esse bloco de código você pode ver que de fato houve a atualização dos valores de pesos1. Se nesse processo não houver nenhuma inconsistência ou erro de sintaxe, após a execução desses blocos de código temos a devida atualização dos pesos da camada oculta para a camada de saída.

```
47 camadaEntradaTransposta = camadaEntrada.T
48 pesos4 = camadaEntradaTransposta.dot(deltaCamadaOculta)
49 pesos0 = (pesos0 * momentum) + (pesos4 * taxaAprendizado)
```

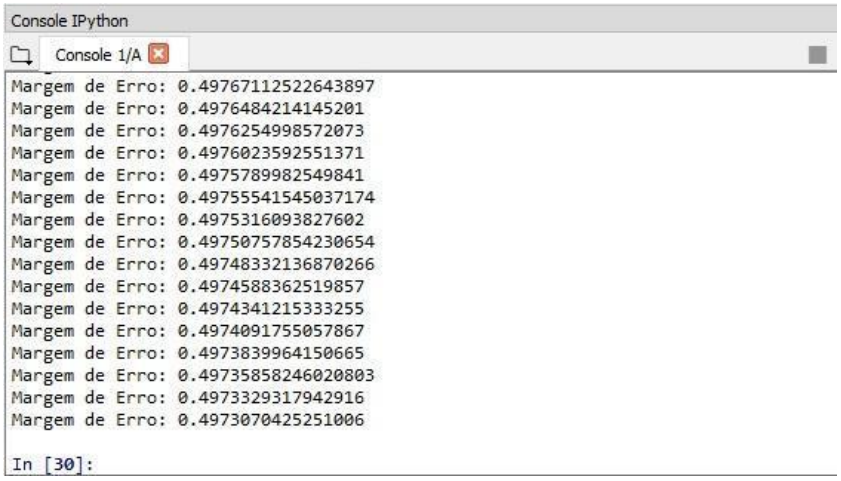
O mesmo processo é realizado para atualização dos valores de pesos0. Ou seja, agora pela lógica backpropagation voltamos mais um passo ao início do perceptron, para que possamos fazer as devidas atualizações a partir da camada de entrada.

```
25 for i in range(ntreinos):
26     camadaEntrada = entradas
27     somaSinapse0 = np.dot(camadaEntrada, pesos0)
28     camadaOculta = sigmoid(somaSinapse0)
29 
30     somaSinapse1 = np.dot(camadaOculta, pesos1)
31     camadaSaida = sigmoid(somaSinapse1)
32 
33     erroCamadaSaida = saidas - camadaSaida
34     mediaAbsoluta = np.mean(np.abs(erroCamadaSaida))
35 
36     derivadaSaida = sigmoideDerivada(camadaSaida)
37     deltaSaida = erroCamadaSaida * derivadaSaida
38 
39     pesos1Transposta = pesos1.T
40     deltaSaidaXpesos = deltaSaida.dot(pesos1Transposta)
41     deltaCamadaOculta = deltaSaidaXpesos * sigmoideDerivada(camadaOculta)
42 
43     camadaOcultaTransposta = camadaOculta.T
44     pesos3 = camadaOcultaTransposta.dot(deltaSaida)
45     pesos1 = (pesos1 * momentum) + (pesos3 * taxaAprendizado)
46 
47     camadaEntradaTransposta = camadaEntrada.T
48     pesos4 = camadaEntradaTransposta.dot(deltaCamadaOculta)
49     pesos0 = (pesos0 * momentum) + (pesos4 * taxaAprendizado)
```

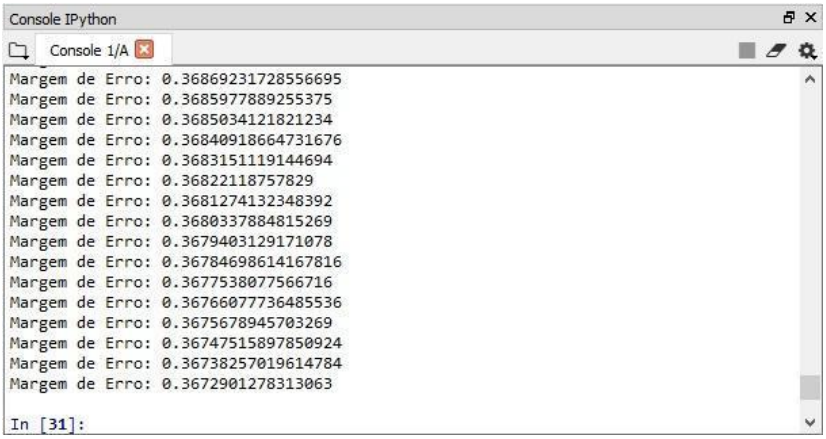
Lembrando que Python é uma linguagem interpretada e de forte indentação, o que em outras palavras significa que o interpretador lê e executa linha após linha em sua sequência normal, e executa blocos de código quando estão uns dentro dos outros de acordo com sua indentação. Sendo assim, todo código praticamente pronto, apenas reorganizamos o mesmo para que não haja problema de interpretação. Dessa forma, todos blocos dedicados aos ajustes dos pesos ficam dentro daquele nosso laço de repetição, para que possamos executá-los de acordo com os parâmetros que iremos definir para as épocas e taxa de aprendizado.

```
51    print('Margem de Erro: ' +str(mediaAbsoluta))
```

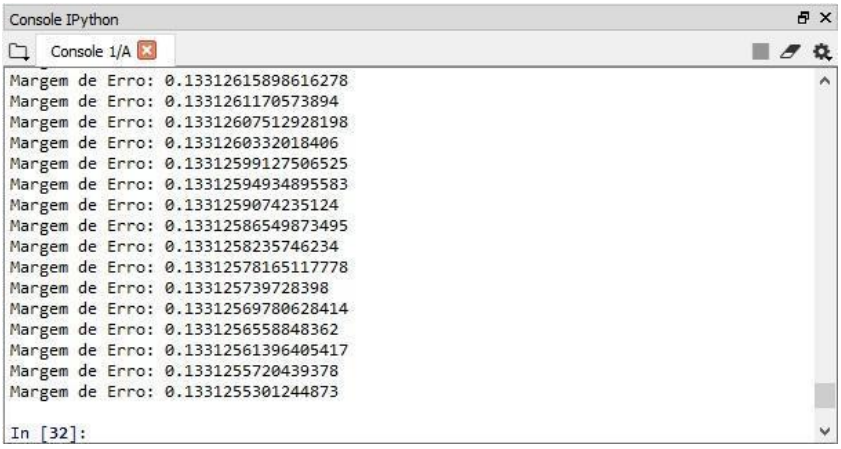
Finalmente criamos uma função print( ) dedicada a nos mostrar via console a margem de erro deste processamento. Agora selecionando e executando todo o código, podemos acompanhar via console o aprendizado de máquina acontecendo. Lembrando que inicialmente havíamos definido o número de vezes a serem executado o código inicialmente como 100. Esse parâmetro pode ser livremente modificado na variável ntreinos.



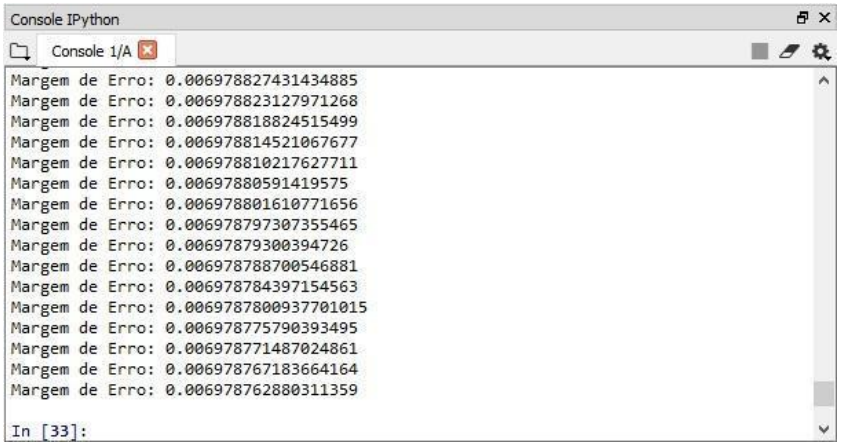
Executando 100 vezes a margem de erro cai para 0.497.



Executando 1000 vezes, a margem de erro cai para 0.367.



Executando 100000 vezes, a margem de erro cai para 0.133.



Por fim, rede executada 1000000 de vezes, a margem de erro cai para 0.006, em outras palavras, agora a rede está treinada para identificar os padrões de entrada e saída de uma tabela XOR, com 0,006% de margem de erro (ou 99,994% de precisão). Lembrando que em nossa amostragem inicial tínhamos 51% de acerto, agora quase 100% de acerto é um resultado excelente.

--//--

CONTATO

fernando2rad@gmail.com

danki.code

PYTHON NEXUS