IPOG

Redes Neurais

EDITORA IPOG

IP5p Instituto de Pós-Graduação e Graduação – IPOG

Redes Neurais. / Autor: Joelma de Moura Ferreira.

240f. :il.

ISBN:

1. Redes Neurais. Perceptron 3. Forward Propagation 4. Backpropagation 5. PyTorch.

CDU: 005

Bibliotecário responsável: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

IPOG

Instituto de Pós Graduação e Graduação

http://www.ipog.edu.br

Sede

Av. T-1 esquina com Av. T-55 N. 2.390 - Setor Bueno - Goiânia-GO. Telefone (0xx62) 3945-5050

SUMÁRIO

APRESENTAÇÃO

Seja bem-vindo ou bem-vinda a disciplina de Redes Neurais. Nesta disciplina, abordaremos os principais conceitos e técnicas que fundamentam o estudo e a aplicação de redes neurais, essenciais para o desenvolvimento de soluções inteligentes em diversos contextos.

Iniciaremos com uma introdução aos conceitos básicos, passando pelas arquiteturas mais utilizadas, como Redes Neurais Feedforward, Convolucionais, Recorrentes, LSTM e Transformers. Aprofundaremos o estudo no Perceptron Multicamadas, para que você possa entender a importância das camadas e suas funções em redes mais complexas. Utilizaremos o PyTorch como ferramenta para implementação prática, cobrindo aspectos como propagação, inicialização de pesos e funções de ativação. Na fase final do curso, veremos a retropropagação e as técnicas de otimização, concluindo com métricas de avaliação de modelos de redes neurais.

Estou confiante de que esta disciplina lhe proporcionará uma base técnica e um entendimento da área. Esteja preparado(a) para ampliar seu conhecimento e desenvolver habilidades fundamentais para sua carreira no universo das redes neurais.

Desejo a você uma ótima experiência de aprendizado e sucesso nos estudos!

Profa. Dra. Joelma de Moura Ferreira

OBJETIVOS

OBJETIVO GERAL

O aluno será capaz de implementar solução usando conceitos de redes neurais artificiais com múltiplas camadas.

OBJETIVOS ESPECÍFICOS

- Entender os conceitos fundamentais de redes neurais artificiais.
- Montar estrutura de uma rede neural definindo estrutura de camadas
- Treinar uma rede neural em projeto prática utilizando o framework Pytorch.

Conheça como esse conteúdo foi organizado!

Unidade 1: Conceitos Iniciais

Unidade 2: Rede Multilayer Perceptron

Unidade 3: Retropropagação (Backpropagation)

Unidade 4: Métricas De Avaliação De Redes Neurais

Unidade 5: Projeto Prática de Treinamento de uma Rede Neural

UNIDADE 1 CONCEITOS INICIAIS

Esta unidade aborda os conceitos iniciais sobre redes neurais. Define o que é um neurônio artificial e a sua estrutura básica. Além disso, são apresentadas as principais arquiteturas de redes neurais.

OBJETIVOS DA UNIDADE 1

Ao final dos estudos, você deverá ser capaz de:

- Entender o que é um neurônio artificial
- Compreender a estrutura de um perceptron
- Diferenciar as principais arquiteturas de rede

1.1 MACHINE LEARNING X DEEP LEARNING

Machine Learning (ML) é uma área da Inteligência Artificial que utiliza algoritmos para analisar dados, aprender com eles e tomar decisões com base em padrões identificados. Ele pode ser dividido em dois principais tipos de aprendizado:

- **Aprendizado supervisionado:** quando o algoritmo é treinado com dados rotulados, onde as entradas e saídas são conhecidas. Exemplo: um modelo de ML pode ser treinado para classificar e-mails como "spam" ou "não spam" com base em características pré-definidas.
- **Aprendizado não supervisionado:** Não há rótulos nos dados, e o algoritmo busca padrões ou grupos nos dados por conta própria. Exemplo: segmentação de clientes em clusters com base no comportamento de compra.

No Machine Learning, os algoritmos, como regressão linear, árvores de decisão e máquinas de vetores de suporte (SVM), processam os dados de forma mais direta, exigindo que os engenheiros de dados extraiam e selecionem manualmente as características mais relevantes. Por exemplo, na previsão de preços de imóveis, em ML, seria necessário selecionar características como o tamanho da casa, localização, número de quartos, etc., e alimentar esses dados no modelo.

Os modelos de Machine Learning funcionam bem com quantidades moderadas de dados e geralmente requerem menos poder computacional. Eles são ideais para problemas mais simples, como previsões baseadas em dados tabulares ou classificações binárias, como prever se um cliente vai ou não cancelar uma assinatura com base em dados históricos.

Deep Learning (DL), por sua vez, é uma abordagem que utiliza redes neurais profundas para realizar essas mesmas tarefas, mas de maneira muito mais eficiente em cenários de grande escala e alta complexidade. As redes neurais usadas no DL têm múltiplas camadas de neurônios artificiais, permitindo que o modelo extraia automaticamente características complexas dos dados.

Em Deep Learning, as redes neurais artificiais aprendem automaticamente as características relevantes, dispensando grande parte da intervenção humana. Um exemplo de DL é uma rede neural convolucional (CNN), amplamente utilizada em reconhecimento de imagens. Nesse caso, o modelo pode identificar características visuais como bordas, formas e padrões sem a necessidade de programação explícita dessas características.

O Deep Learning se destaca em tarefas que envolvem grandes volumes de dados, como processamento de linguagem natural (NLP) ou reconhecimento de

imagem. Ele se torna mais eficaz à medida que a quantidade de dados cresce, enquanto os algoritmos tradicionais de ML podem perder desempenho com grandes volumes de dados devido à dificuldade de identificar padrões complexos.

Exemplo: Em uma aplicação de reconhecimento de voz, o DL utiliza redes neurais recorrentes (RNN) para analisar e interpretar padrões de fala em tempo real, enquanto o ML tradicional teria dificuldades para capturar todas as nuances de variações na voz e contexto.

Os modelos de Machine Learning geralmente podem ser treinados em computadores convencionais com CPUs, enquanto os modelos de Deep Learning requerem maior poder computacional, frequentemente utilizando GPUs (unidades de processamento gráfico) ou até mesmo clusters em nuvem para processar grandes volumes de dados em um tempo razoável.

Essa diferença de poder de processamento é evidente em aplicações como veículos autônomos, onde o deep learning é usado para processar imagens e sensores em tempo real, enquanto métodos tradicionais de machine learning não conseguiriam lidar com a quantidade de dados necessária para uma condução segura e eficiente.

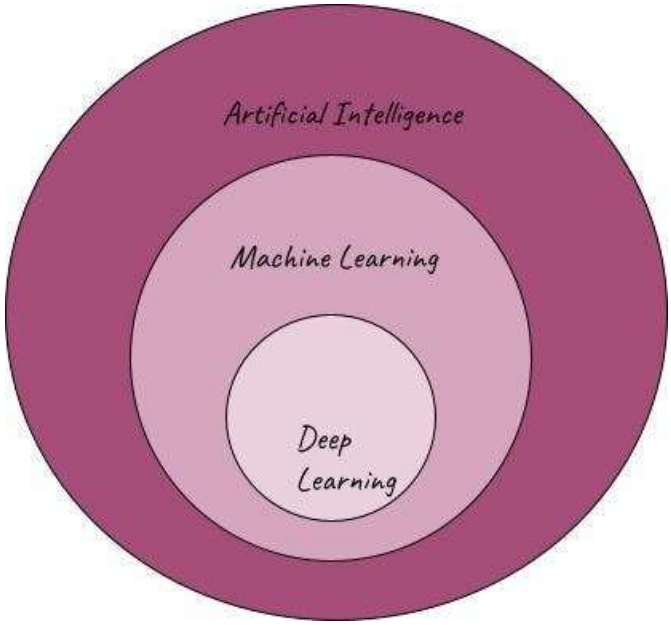


Figura 1: Relação de Machine Learning e Deep Learning

Fonte: https://www.turing.com/kb/ultimate-battle-between-deep-learning-and-machine-learning

1.1.2 REDES NEURAIS ARTIFICIAIS

As redes neurais artificiais (ANNs) são a estrutura fundamental do deep learning. Inspiradas na estrutura biológica do cérebro humano, são sistemas computacionais inspirados no funcionamento do cérebro humano, capazes de aprender padrões a partir de dados. Imagine que você tem um conjunto de imagens e deseja classificá-las, por exemplo, identificar se uma imagem contém um gato ou um cachorro. Uma rede neural realiza essa tarefa aprendendo uma função matemática que mapeia as entradas (as imagens) para as saídas (as classificações) a partir de exemplos fornecidos durante o treinamento.

Simplificando, uma rede neural é composta por neurônios artificiais, que são pequenas unidades de processamento que realizam cálculos simples. Esses neurônios podem estar interconectados de várias maneiras para formar uma estrutura capaz de resolver problemas complexos, como reconhecimento de voz, tradução automática, e até mesmo jogos de estratégia. O conceito de "deep learning" surge quando essas redes neurais são profundas, ou seja, possuem várias camadas de neurônios, permitindo a modelagem de padrões muito complexos.

1.2. CONCEITOS CHAVE EM REDES NEURAIS

1.2.1 NEURÔNIO ARTIFICIAL

Um neurônio artificial é a unidade básica de uma rede neural artificial que simula o comportamento de um neurônio biológico. Ele recebe diversas entradas, as processa por meio de uma soma ponderada, e aplica uma função de ativação para gerar uma saída.

O funcionamento do neurônio pode ser descrito por quatro componentes principais: pesos, viés (bias), função de somatório e função de ativação. O modelo de um neurônio artificial é mostrado na Figura 2.

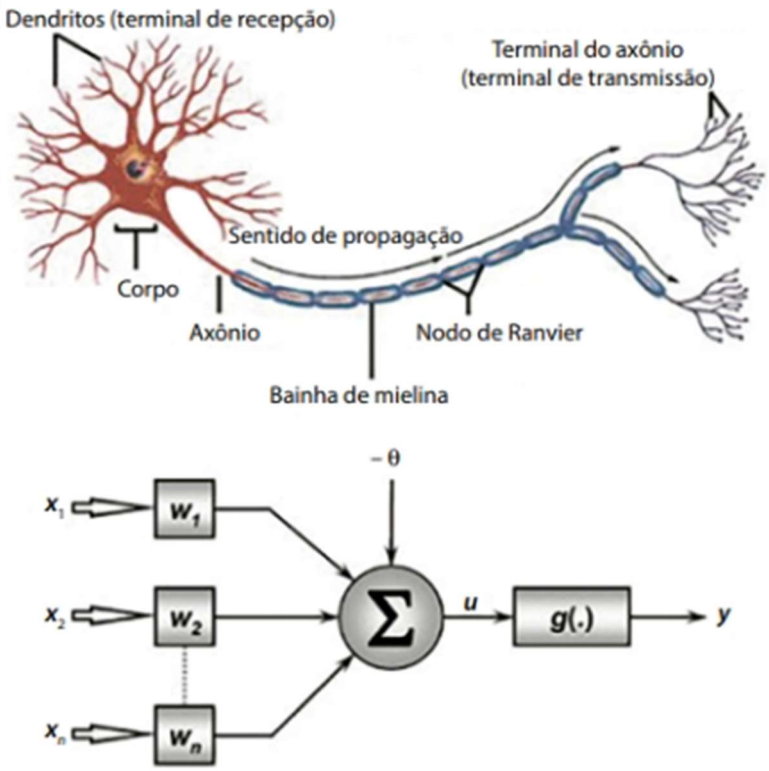


Figura 2: Modelo de um neurônio artificial.
Fonte: SILVA. et. al. **Inteligência artificial** SAGAH, 2019.

Exemplo:

Suponha que um neurônio artificial receba três entradas x_1, x_2 e x_3 com valores 0.5, 0.8 e 0.2, respectivamente, e que os pesos associados a essas entradas sejam w_1= 0.3, w_2 = -0.2, w_3 = 0.7. O viés do neurônio θ é 0.1. O somatório dessas entradas ponderadas, mais o viés, seria:

$$y = (0.5 \times 0.3) + (0.8 \times -0.2) + (0.2 \times 0.7) + 0.1$$

O que resulta em:

$$y = 0.15 - 0.16 + 0.14 + 0.1 = 0.23$$

Esse valor y=0.23 seria então passado para a função de ativação, que determinará se o neurônio será ativado ou não.

1.2.1.1 PESO DE CONEXÃO

Os pesos, na Figura 2 representados por w_{ki}, determinam a força ou a importância de cada entrada. Durante o treinamento da rede, esses pesos são

ajustados de acordo com o erro cometido na saída, utilizando algoritmos como o gradiente descendente para minimizar a diferença entre a saída predita e o valor real.

Se uma entrada x_i está associada a um peso muito alto, como w_i=10, o impacto dessa entrada na saída do neurônio será muito maior. Se essa entrada for irrelevante para o problema, o treinamento ajustará o peso para valores mais baixos, próximo de zero, reduzindo seu impacto. Esse ajuste contínuo dos pesos é o que permite que as redes neurais aprendam padrões complexos nos dados.

1.2.1.2 VIÉS

O viés ou bias, na Figura 2 representados por θ, é um valor constante adicionado ao resultado ponderado. O viés é um valor que permite ao modelo fazer ajustes mais flexíveis. Ele desloca a função de ativação, permitindo que o neurônio gere uma saída diferente de zero, mesmo quando todas as entradas forem zero. Isso dá ao modelo a capacidade de aprender com mais eficácia.

Por exemplo, Se o viés b for negativo, ele torna mais difícil a ativação do neurônio (um valor de θ =−1 exigiria entradas mais fortes para ativar o neurônio). Se θ =1, o neurônio será mais propenso a ser ativado, mesmo com entradas fracas. O ajuste correto do viés ajuda a calibrar a rede neural de forma mais precisa durante o treinamento.

1.2.1.3 FUNÇÃO SOMATÓRIO

Em um neurônio artificial, o somatório (ou função de soma), na Figura 2 representados por ∑, tem o objetivo de calcular o valor de ativação do neurônio com base nas entradas recebidas e nos pesos associados a cada entrada.

Ele funciona da seguinte maneira:

1. **Entrada de Dados:** Cada neurônio recebe várias entradas, cada uma multiplicada por um peso. Esses pesos indicam a importância relativa de cada entrada.
2. **Somatório:** O somatório calcula a soma ponderada dessas entradas, ou seja, a multiplicação de cada valor de entrada pelo seu respectivo peso, somada com um termo de viés (bias). O viés é adicionado para ajustar a função de ativação, permitindo que o modelo se ajuste melhor aos dados.

$$y = \sum_{i}^{n} (x_i \times w_i) + \theta$$

Onde:

- x_i é o valor de entrada,

- w_i é o peso da conexão,
- θ é o viés,
- y é o valor resultante do somatório.

Um exemplo de código em Python que calcula o somatório de entradas, pesos e viés de dados fixos.

```python
import numpy as np

# Entradas e pesos
entradas = np.array([0.5, 0.8, 0.2])
pesos = np.array([0.3, -0.2, 0.7])
viés = 0.1

# Função de somatório
somatório = np.dot(entradas, pesos) + viés
print(somatório)
```

Nesse código, utilizamos a função np.dot para calcular o produto escalar entre as entradas e os pesos, somando o viés ao resultado. Isso reproduz a equação descrita anteriormente:

$$y = \sum_{i}^{n} (x_i \times w_i) + \theta$$

Um exemplo de um problema simples que poderia ser mapeado para um neurônio artificial seria a soma de dois números. Esse neurônio receberia todas as entradas ponderadas e retornaria o valor (aproximado).

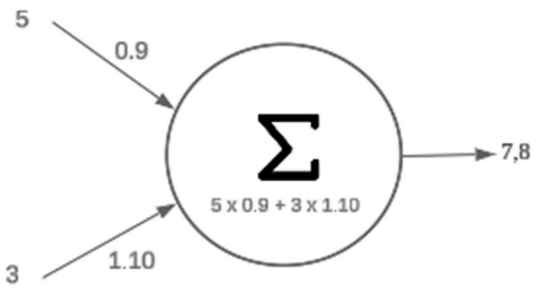


Figura 3: Modelo do neurônio do somatório

Na Figura 3 vemos a ilustração desse exemplo. As entradas x_1 e x_2, são respectivamente 5 e 3. Já os pesos w_1 e w_2, são respectivamente, 0.9, 1.1. Observe que quando a soma 5 x 0.9 + 3 x 1.1 é executada o resultado é 7.8, que é um valor próximo da resposta real que seria 8. Os pesos dizem ao neurônio para responder mais a uma entrada e menos a outra.

Abaixo um exemplo em código Python de implementação desse neurônio

```python
import numpy as np

def perceptron(entradas, pesos):
    soma_ponderada = np.dot(entradas, pesos)
    return soma_ponderada

entradas = [ [5, 3] ]
pesos = np.array([0.9, 1.1])

for entrada in entradas:
    saida = perceptron(entrada, pesos)
    print(f'Entrada: {entrada} -> Saída: {saida}')
```

Mas como saber que os pesos 0.9 e 1.1 são os corretos, neste caso eles estão fixos? Inicialmente eles são escolhidos aleatoriamente, ou seguindo algum critério. Durante o processo de treinamento, esses pesos são ajustados para minimizar o erro entre a saída esperada e a saída real, permitindo que a rede "aprenda" a tarefa para a qual foi projetada. Os pesos são ajustados durante o treinamento — é assim que a rede aprende. Mas como fazer isso veremos mais a frente.

SAIBA MAIS:

Pesquise sobre a biblioteca numpy e descubra o que os comandos np.dot e np.array fazem. Fonte: https://numpy.org/doc/stable/

1.2.1.4 FUNÇÃO DE ATIVAÇÃO

O resultado do somatório passa para uma função de ativação, que poderia ser uma função linear, sigmoid, ReLU, entre outras. A função de ativação, na Figura

2 representados por g (.), decide se o neurônio deve ser ativado ou não. Ela introduz não-linearidade na rede, permitindo que a rede aprenda padrões complexos. Exemplos comuns de funções de ativação incluem a função sigmoide, ReLU (Rectified Linear Unit), e tanh.

Existem várias funções de ativação comuns utilizadas em redes neurais:

- **ReLU (Rectified Linear Unit):** Uma das funções de ativação mais populares, especialmente em redes neurais profundas. A ReLU retorna a entrada diretamente se for positiva, e zero caso contrário. Sua fórmula é ReLU(x)=max(0,x). A ReLU é amplamente utilizada devido à sua simplicidade e eficiência em redes profundas, pois ela ajuda a resolver o problema do desaparecimento do gradiente, que pode ocorrer com outras funções de ativação.
- **Sigmoid:** A função sigmoide mapeia a entrada em um valor entre 0 e 1, usando a fórmula $\sigma(z)=1/1+e^{-z}$. É frequentemente usada em problemas de classificação binária, onde a saída deve representar uma probabilidade. No entanto, a função sigmoide pode sofrer com problemas de saturação, onde os gradientes se tornam muito pequenos para valores extremos de entrada.

Cada função de ativação tem suas próprias vantagens e desvantagens, e a escolha da função apropriada depende do problema específico que se está tentando resolver.

O código abaixo é a modificação do anterior aplicando a função de ativação ReLU na saída. Não vai haver nenhuma alteração no resultado, pela característica do problema e por termos apenas um neurônio.

```
import numpy as np

# Função de ativação (ReLU)
def relu(x):
    return max(0, x)

def perceptron(entradas, pesos):
    soma_ponderada = np.dot(entradas, pesos)
    saida_ativacao = relu(soma_ponderada)  # Aplicando a função de ativação ReLU
    return soma_ponderada, saida_ativacao

entradas = [[5, 3]]
pesos = np.array([0.9, 1.1])

for entrada in entradas:
    soma, saida = perceptron(entrada, pesos)
    print(f'Entrada: {entrada} -> Soma ponderada: {soma} -> Saída após ativação (ReLU): {saida}')
```

1.2.3.4 PERCEPTRON

O **Perceptron** é um tipo específico de neurônio artificial. Ele é o primeiro modelo formal de um neurônio em uma rede neural, criado por Frank Rosenblatt em 1958. O Perceptron foi projetado para resolver problemas de classificação binária, ou seja, decidir entre duas classes distintas, e utiliza uma função de ativação do tipo degrau, que gera uma saída discreta (0 ou 1).

A estrutura básica de um Perceptron consiste em entradas, pesos, um somatório e uma função de ativação. Assim como no neurônio artificial descrito na seção anterior, o Perceptron calcula a soma ponderada das entradas, adiciona um viés e aplica uma função de ativação para gerar a saída. O Perceptron realiza a seguinte operação matemática:

$$y = \emptyset\left(\sum_{i=1}^{n}(x_i \times w_i) + b\right)$$

Onde:

- x_i é o valor de entrada,
- w_i é o peso da conexão,
- b é o viés,
- ϕ é a função de ativação, tipicamente uma função degrau que decide se a saída será 0 ou 1.
- y é o valor resultante do somatório.

Observe que um neurônio artificial pode ter várias funções de ativação, como ReLU, sigmoide ou tangente hiperbólica. O Perceptron utiliza apenas a função de ativação degrau, o que limita sua aplicação a problemas de classificação simples. A função de ativação degrau retornará 1 se o resultado do somatório for maior ou igual a 0, e retornará 1 se for menor que 0. A função de ativação degrau faz com que o Perceptron seja adequado apenas para problemas de **classificação linear**.

$$y = \begin{cases} 1, & \text{se } \sum_{i=1}^{n}(x_i \times w_i) + b \geq 0 \\ 0, & \text{se } \sum_{i=1}^{n}(x_i \times w_i) + b < 0 \end{cases}$$

Dessa forma, uma rede perceptron é usada apenas para classificar casos linearmente separáveis com saída binária (0 ou 1), não conseguindo resolver problemas não lineares, sendo assim, não é utilizado em arquiteturas modernas de Deep Learning.

1.3. ARQUITETURAS E TIPOS DE REDES NEURAIS

Na seção anterior vimos um exemplo com um único neurônio, mas uma rede neural tem centenas, milhares ou bilhões de neurônios a depender do problema, todos estes conectados, sendo a saída de um a entrada de outro. A Figura 4 ilustra uma rede neural com diversos neurônios distribuídos em camadas.

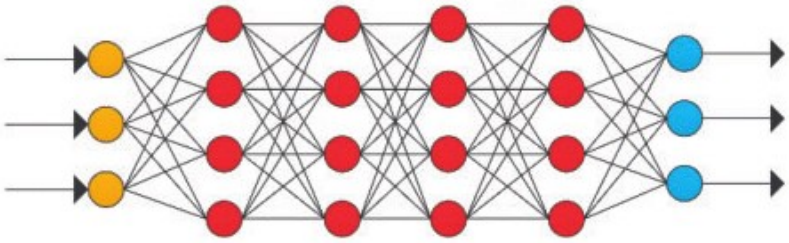


Figura 4: Rede Neural com diversos neurônios.

Essas redes neurais mais densas nos dão a oportunidade de definir mais parâmetros nos pesos e vieses da rede, e assim reconhecer tarefas mais complicadas.

Existem várias arquiteturas de redes neurais, compostas por diversos neurônios conectados, cada uma projetada para lidar com diferentes tipos de dados e tarefas de aprendizado. A seguir, abordaremos os tipos mais amplamente reconhecidos:

1.3.1 REDES NEURAIS FEEDFORWARD (FNN)

As Redes Neurais Feedforward (FNNs) são o tipo mais simples de redes neurais artificiais (ANNs). Nessas redes, a informação flui em uma única direção, da camada de entrada para a camada de saída, sem loops de retroalimentação. As FNNs são frequentemente usadas para tarefas básicas de classificação, como reconhecer dígitos escritos à mão, e problemas de regressão, como prever valores contínuos a partir de entradas numéricas.

As FNNs têm uma arquitetura chamada de totalmente conectada (fully connected), onde cada neurônio em uma camada está conectado a todos os neurônios da próxima camada.

Exemplo de uma FNN:

- **Camada de Entrada:** Contém o vetor de entrada, que representa os dados de entrada.
- **Camadas Ocultas:** Estas são compostas por neurônios que recebem entradas ponderadas da camada anterior e aplicam uma função de ativação. Em uma FNN totalmente conectada, cada neurônio em uma camada oculta está conectado a todos os neurônios da camada anterior e seguinte.
- **Camada de Saída:** É a última camada de neurônios, que produz o resultado final da rede, como uma classificação ou uma previsão. O número de neurônios na camada de saída corresponde ao número de classes ou ao formato dos dados de saída.

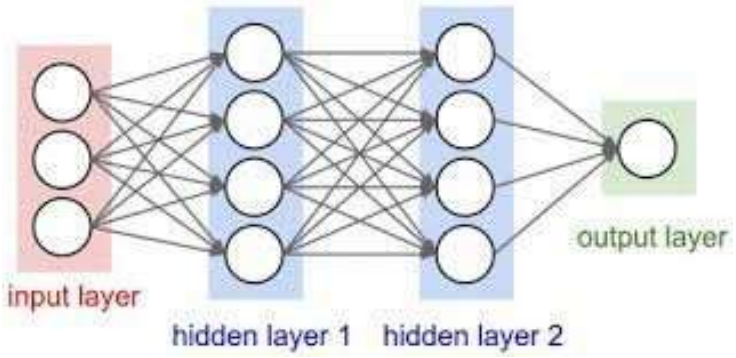


Figura 5: Exemplo de redes Neurais Feedforward (FNNs)

As camadas ocultas são chamadas assim porque não são diretamente observáveis pelo "mundo exterior" – elas processam internamente as entradas para produzir as saídas desejadas.

1.3.2 REDES NEURAIS CONVOLUCIONAIS (CNN)

As Redes Neurais Convolucionais (CNNs) são especializadas no processamento de dados em forma de grade, como imagens, onde os dados possuem uma estrutura espacial significativa. As CNNs usam camadas convolucionais que aplicam filtros ou kernels às entradas para extrair características relevantes, como bordas, texturas e formas.

As CNNs são amplamente utilizadas em tarefas de visão computacional, como:

- **Reconhecimento de Imagens:** a rede identifica objetos em imagens.
- **Detecção de Objetos:** a rede localiza e classifica múltiplos objetos em uma imagem.
- **Reconhecimento de Vídeos:** a rede analisa sequências de quadros para entender o conteúdo visual.

A arquitetura típica de uma CNN inclui camadas convolucionais, seguidas de camadas de pooling (que reduzem a dimensionalidade), e finalmente camadas totalmente conectadas que produzem a saída final.

1.3.3 REDES NEURAIS RECORRENTES (RNN)

As Redes Neurais Recorrentes (RNNs) são projetadas para o processamento de dados sequenciais, como séries temporais ou texto. Ao contrário das FNNs, as RNNs possuem loops que permitem que a saída de um neurônio seja usada como entrada para o mesmo neurônio em passos subsequentes. Isso cria uma espécie de "memória" interna, permitindo que a rede capture dependências temporais.

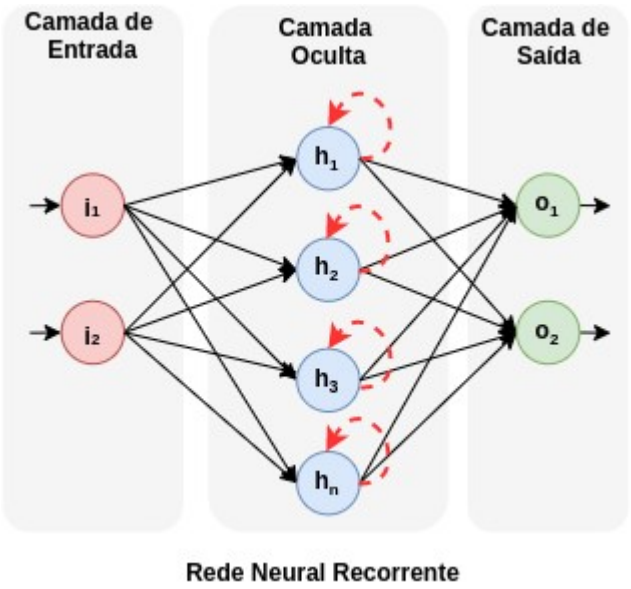


Figura 6: Exemplo de redes Neurais Recorrentes (RNNs)

As RNNs são usadas em diversas aplicações, incluindo:

- **Modelagem de Linguagem Natural:** a rede prevê a próxima palavra em uma frase.
- **Reconhecimento de Fala:** a rede converte fala em texto.
- **Previsão de Séries Temporais:** prever preços de ações ou demandas de energia.

Apesar de sua capacidade de capturar dependências temporais, as RNNs simples podem ter dificuldade em lembrar informações por longos períodos, o que levou ao desenvolvimento de variantes como as LSTMs.

1.3.4 REDES DE MEMÓRIA DE LONGO CURTO PRAZO (LSTM)

As Redes de Memória de Longo Curto Prazo (LSTMs) são um tipo especial de RNN projetado para lidar com a limitação das RNNs padrão em capturar dependências de longo prazo. As LSTMs introduzem células de memória e mecanismos de portas que controlam o fluxo de informações, permitindo que a rede aprenda e lembre-se de sequências mais longas.

As LSTMs são particularmente eficazes em tarefas como:

- **Geração de Texto:** a rede gera texto coerente ao aprender padrões em grandes corpora.
- **Tradução de Idiomas:** a rede traduz texto de um idioma para outro.
- **Previsão de Séries Temporais:** entender padrões de longo prazo é essencial.

Essas redes têm sido essenciais em avanços em processamento de linguagem natural e outras áreas onde a dependência temporal é crítica.

1.3.5 REDES ADVERSÁRIAS GENERATIVAS (GAN)

As Redes Adversárias Generativas (GANs) consistem em duas redes neurais que competem entre si: uma rede geradora e uma rede discriminadora. A rede geradora cria dados sintéticos (como imagens) a partir de ruído aleatório, enquanto a rede discriminadora tenta distinguir entre dados reais e falsos.

As GANs têm revolucionado áreas como:

- **Geração de Imagens e Vídeos:** podem criar imagens realistas de pessoas, animais, e cenários que não existem.
- **Criação de Conteúdo Criativo:** gerar obras de arte, música, ou mesmo designs de produtos.
- **Aprimoramento de Imagens:** aumentar a resolução de imagens ou remover ruídos.

As GANs são um exemplo poderoso de como redes neurais podem ser usadas de maneiras criativas e inovadoras.

1.3.6 AUTOENCODERS (AES)

Os Autoencoders são redes neurais utilizadas para aprendizado não supervisionado, onde a tarefa principal é aprender uma representação compacta dos dados (codificação) e depois reconstruir os dados originais a partir dessa representação. A estrutura básica de um autoencoder inclui uma camada de codificação que reduz a dimensionalidade dos dados e uma camada de decodificação que tenta reconstruir os dados a partir da codificação.

Os Autoencoders são usados em:

- **Redução de Dimensionalidade:** podem ajudar a reduzir o número de variáveis em um conjunto de dados, preservando a maior quantidade de informação possível.
- **Redução de Ruído:** podem ser treinados para remover ruído de imagens ou sinais.
- **Detecção de Anomalias:** aprendem a reconstruir dados normais, e qualquer dado que não possa ser bem reconstruído é considerado uma anomalia.

1.3.7 REDES TRANSFORMER

As Redes Transformer representam uma arquitetura de rede neural que se baseia inteiramente em mecanismos de autoatenção, evitando a necessidade de recorrência ou convoluções. Os Transformers foram introduzidos para melhorar a eficiência do treinamento de redes neurais, especialmente em tarefas de processamento de linguagem natural.

Os Transformers são a base de modelos como o GPT (Generative Pre-trained Transformer) e BERT (Bidirectional Encoder Representations from Transformers), e têm sido usados em:

- **Tradução de Idiomas:** são treinados para traduzir textos entre diferentes línguas com alta precisão.
- **Resumo de Textos:** podem condensar longos documentos em resumos curtos e precisos.
- **Geração de Linguagem Natural:** geram texto fluente e coerente, como histórias, artigos, ou respostas a perguntas.

A arquitetura Transformer tem se mostrado eficaz em capturar relações contextuais em dados sequenciais de maneira mais eficiente do que as RNNs, o que levou a sua ampla adoção em diversas aplicações de inteligência artificial.

Nesta disciplina focaremos na arquitetura mais simples: Redes Neurais Feedforward (FNN).

Aprenda Mais:

O que é uma rede neural? Disponível em: https://www.elastic.co/pt/what-is/neural-network

Testando seu conhecimento

Questão 1: Qual das seguintes opções descreve o principal objetivo de uma rede neural?
a) Reproduzir exatamente o comportamento do cérebro humano.
b) Resolver problemas utilizando algoritmos baseados em lógica booleana.
c) Aprender padrões a partir de dados fornecidos.
d) Classificar imagens apenas em casos de problemas lineares.

Questão 2: Um neurônio artificial realiza o seguinte cálculo básico:
a) Multiplica as entradas pelos pesos, aplica uma função de ativação e gera a saída.
b) Subtrai os valores das entradas pelos pesos e gera a saída diretamente.
c) Soma todas as entradas e retorna o valor diretamente, sem função de ativação.
d) Classifica as entradas com base em uma função de ativação fixa, sem ajuste de pesos.

Questão 3: Em uma rede neural, qual é a função dos pesos de conexão?
a) Ajustar o impacto de cada entrada no valor de saída do neurônio.
b) Regular a frequência de ativação do neurônio.
c) Definir o tipo de função de ativação usada pelo neurônio.
d) Controlar o número de camadas da rede neural.

Questão 4: O viés (bias) em um neurônio artificial tem a função de:
a) Definir a frequência de atualização dos pesos durante o treinamento.
b) Ajustar a função de ativação, permitindo ativação do neurônio mesmo com entradas zeradas.
c) Impedir que o neurônio seja ativado caso todas as entradas sejam positivas.
d) Limitar a quantidade de dados que o neurônio pode processar.

Questão 5: Qual é a característica que distingue o Perceptron de outras arquiteturas de rede neural?
a) Utiliza uma função de ativação contínua como a ReLU.
b) Só consegue classificar problemas linearmente separáveis.
c) É capaz de resolver problemas não lineares complexos.

d) Não utiliza pesos ou viés em sua estrutura.

Questão 6: A função do somatório em um neurônio artificial é:
a) Aplicar uma função de ativação não linear às entradas.
b) Somar as entradas ponderadas pelos pesos e adicionar o viés.
c) Multiplicar todas as entradas e retorná-las diretamente.
d) Ajustar os pesos de conexão durante o treinamento.

Questão 7: Durante o treinamento de uma rede neural, os pesos são ajustados para:
a) Maximizar a saída de cada neurônio em relação às entradas.
b) Minimizar o erro entre a saída prevista e a saída esperada.
c) Garantir que a rede gere sempre as mesmas previsões para entradas diferentes.
d) Aumentar a complexidade do modelo, tornando-o mais difícil de ser interpretado.

Respostas:

Questão 1: c)
Questão 2: a)
Questão 3: a)
Questão 4: b)
Questão 5: b)
Questão 6: b)
Questão 7: b)

Desafio: Altere o código do Perceptron da imagem para devolver, ao invés da soma, o valor 0 se a soma for menor que um threshold de 10. E retornar 1 se for maior ou igual a 10.

```python
import numpy as np

def perceptron(entradas, pesos):
    soma_ponderada = np.dot(entradas, pesos)
    return soma_ponderada

entradas = [ [5, 3] ]
pesos = np.array([0.9, 1.1])

for entrada in entradas:
    saida = perceptron(entrada, pesos)
    print(f'Entrada: {entrada} -> Saída: {saida}')
```

UNIDADE 2 REDE MULTILAYER PERCEPTRON

Nesta unidade será abordado o processo de treinamento de uma rede neural, em especial a Rede Neural MultiLayer Perceptron.

.

OBJETIVOS DA UNIDADE 2

Ao final dos estudos, você deverá ser capaz de:

- Compreender o que é uma Rede Neural MultiLayer Perceptron
- Entender o passo de criar camadas em uma rede neural
- Entender a propagação direta (forward propagation).
- Implementar uma arquitetura de rede simples utilizando PyTorch

2.1 A IMPORTÂNCIA DAS CAMADAS EM UMA REDE NEURAL

As redes Neural Feedforward (FNN) são um tipo de rede neural onde a informação se propaga apenas em uma direção, dos neurônios de entrada para os de saída, sem ciclos ou loops. A sua arquitetura pode incluir uma única camada oculta ou várias camadas ocultas, mas o ponto chave é que não há realimentação ou conexões recorrentes.

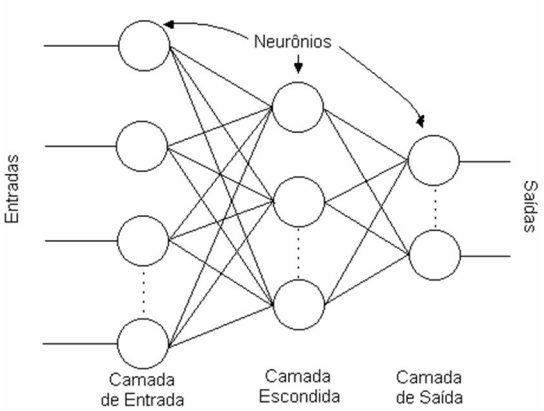


Figura 7: Rede Neural com diversas camadas

Fonte: https://www.researchgate.net/figure/Rede-Neural-Artificial-com-3-camadas-A-arquitetura-de-rede-adotada-neste-trabalho-possui_fig1_305212719

Em problemas reais e complexos, os neurônios são organizados em camadas, e cada uma desempenha um papel específico no processamento dos dados dentro da rede neural. As redes neurais multicamadas, como é o caso das FNN, permitem que padrões complexos sejam aprendidos e identificados, algo que redes de camada única não conseguem realizar. Na arquitetura de uma rede neural, essas camadas podem ser descritas da seguinte forma.

- **Camada de Entrada:** Recebe os dados iniciais e os transmite para as camadas subsequentes. Cada neurônio na camada de entrada representa uma característica do dado de entrada, como os pixels de uma imagem.
- **Camadas Ocultas (escondidas):** Realizam cálculos intermediários e transformações nos dados. Cada camada oculta captura diferentes níveis de abstração dos dados. Em uma rede neural profunda, essas camadas ocultas permitem a detecção de padrões muito complexos. Por exemplo, em uma rede que reconhece rostos, as primeiras camadas podem detectar bordas e texturas, enquanto as camadas mais profundas podem reconhecer formas como olhos e boca.
- **Camada de Saída:** Produz o resultado final, como uma classificação ou uma previsão. O número de neurônios na camada de saída depende do

problema em questão. Por exemplo, em uma tarefa de classificação binária, a camada de saída pode ter um único neurônio com uma função de ativação sigmoide para retornar a probabilidade de uma classe.

Sendo assim, podemos resumir que, uma rede de camada única possui as entradas ligadas diretamente nos neurônios de saída, já uma rede de múltiplas camadas possui camadas intermediárias e permite que os dados passem por várias transformações antes de chegar à saída, o que aumenta significativamente a capacidade da rede de modelar relações não lineares, conforme ilustrado na Figura 8.

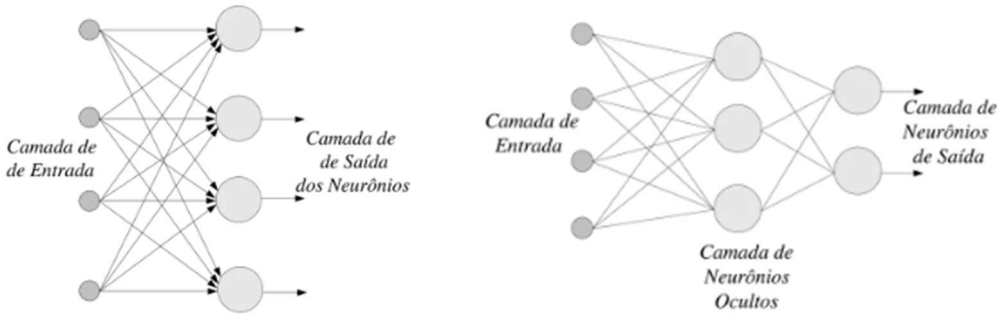


Figura 8: Rede Neural com camada única (esquerda) e com múltiplas camadas (direita)

2.2 PERCEPTRON MULTICAMADA

A FNN é um termo genérico que pode englobar diversos tipos de redes neurais, inclusive as redes Perceptron (com uma camada) e MultiLayer Perceptron (MLP) (com várias camadas).

MultiLayer Perceptron (MLP), ou perceptron multicamadas em português, é um dos modelos mais fundamentais de redes neurais. Ele é composto por uma camada de entrada, uma ou mais camadas ocultas e uma camada de saída.

O MLP utiliza funções de ativação não lineares, como a função ReLU (Rectified Linear Unit), para permitir que a rede aprenda padrões complexos, o que o torna mais poderoso do que o perceptron simples, que só resolve problemas linearmente separáveis.

O principal motivo pelo qual o MLP é poderoso é sua habilidade de ajustar os pesos e vieses dos neurônios para modelar funções não lineares. No processo de treinamento o MLP utiliza algoritmos de retropropagação e gradiente descendente para ajustar os pesos, o que é uma técnica comum em redes neurais mais complexas.

Dessa forma, para definir uma arquitetura de MLP, é importante especificar os seguintes elementos:

- O número de camadas ocultas
- A quantidade de neurônios em cada camada
- A função de ativação utilizada
- O algoritmo de aprendizado, como o gradiente descendente

2.3 INTRODUÇÃO AO PYTORCH

Antes de aprender a treinar uma rede MLP é preciso falar sobre o framework que vamos usar: O PyTorch. O PyTorch é uma biblioteca de aprendizado profundo amplamente utilizada para a construção e treinamento de redes neurais. Desenvolvida pelo laboratório de pesquisa em inteligência artificial do Facebook (Meta AI), ela é conhecida por sua simplicidade e flexibilidade, sendo ideal tanto para pesquisas quanto para a implementação de modelos de aprendizado profundo em ambientes de produção.

O PyTorch oferece diversas vantagens no treinamento de redes neurais, entre elas se desrtacam:

- **Diferenciação Automática (Autograd):** Um dos aspectos mais importantes do PyTorch é o suporte à diferenciação automática, através do módulo autograd. Isso significa que, ao definir o modelo, o PyTorch automaticamente calcula os gradientes necessários para ajustar os pesos durante o processo de retropropagação, simplificando o processo de treinamento de redes neurais.
- **Tensores Otimizados para GPU:** O PyTorch permite o uso de tensores, estruturas de dados que são semelhantes a arrays do NumPy, mas otimizadas para serem usadas em GPUs. Isso acelera o treinamento de redes neurais, especialmente para modelos grandes e complexos, já que GPUs são projetadas para lidar com operações em paralelo de maneira eficiente.

2.3.1 FRAMEWORKS DE TREINAMENTO DE REDES NEURAIS

Embora o PyTorch seja amplamente utilizado, há outras bibliotecas que desempenham papéis importantes no treinamento de redes neurais. Aqui estão três outras opções populares:

- TensorFlow: Desenvolvido pela Google, oferece grande flexibilidade e poder, mas sua curva de aprendizado pode ser um pouco mais íngreme que a do PyTorch. TensorFlow é frequentemente utilizado em grandes projetos de produção.
- Keras: inicialmente um projeto independente, agora integrado ao TensorFlow. O Keras é uma API de alto nível para redes neurais, que facilita a construção de modelos com menos código e abstrações simplificadas. Envolve camadas e operações comumente usadas em aprendizado profundo, organizando-as em blocos de construção simples. No entanto, para projetos que exigem maior controle e otimização, pode ser necessário acessar o backend subjacente, como TensorFlow ou Theano.
- MXNet: Suportado pela Amazon Web Services (AWS), o MXNet é outra biblioteca que suporta o treinamento distribuído em larga escala, tornando-o uma escolha popular para grandes corporações. Embora menos popular que PyTorch e TensorFlow, ele tem suporte robusto para computação em nuvem, especialmente no AWS.

Comparando o Keras e o PyTorch que são bibliotecas muito usadas, Piotr Migdal e Rafał Jakubanis do site deepsense.ai, afirmam que:

> “Keras é um framework de nível mais alto que envolve camadas e operações comumente usadas em aprendizado profundo, organizando-as em blocos de construção simples, como peças de lego, abstraindo as complexidades do aprendizado profundo dos olhos preciosos de um cientista de dados. PyTorch oferece um ambiente de nível mais baixo para experimentação, dando ao usuário mais liberdade para escrever camadas personalizadas e explorar os detalhes das tarefas de otimização numérica”.

Piotr Migdal e Rafał Jakubanis inclusive fazem uma comparação direta de uma rede convolucional simples definida em Keras e PyTorch:

Keras

```keras
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPool2D())
model.add(Conv2D(16, (3, 3), activation='relu'))
model.add(MaxPool2D())
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

PyTorch

```pytorch
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3)
        self.conv2 = nn.Conv2d(32, 16, 3)
        self.fc1 = nn.Linear(16 * 6 * 6, 10)
        self.pool = nn.MaxPool2d(2, 2)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 6 * 6)
        x = F.log_softmax(self.fc1(x), dim=-1)
        return x
model = Net()
```

Fonte: https://deepsense.ai/keras-or-pytorch/

Aprenda Mais:

Tutorial PyTorch: um guia rápido para você entender agora os fundamentos do PyTorch Disponível em: https://www.insightlab.ufc.br/tutorial-pytorch-um-guia-rapido-para-voce-entender-agora-os-fundamentos-do-pytorch/

2.3.2 ESTRUTURA BÁSICA DO PYTORCH

A estrutura básica do PyTorch gira em torno do conceito de tensores. Um tensor no PyTorch é uma estrutura de dados que pode ser vista como uma generalização de matrizes e arrays. Tensores são fundamentais para representar dados de entrada e saída, bem como para armazenar os pesos e gradientes das redes neurais.

Um dos grandes benefícios do PyTorch é que esses tensores podem ser manipulados tanto em CPUs quanto em GPUs, o que acelera o processamento e o treinamento de grandes modelos.

Aqui está um exemplo básico de criação de um tensor no PyTorch:

```python
import torch

# Criando um tensor 2x2
tensor = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
print(tensor)
```

O código a seguir mostra formas diferenciadas de criar um tensor usando o PyTorch.

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

```
[2]: #Criação de uma matriz de dados
     data = [[1, 2],[3, 4]]
     data
```

```
[2]: [[1, 2], [3, 4]]
```

```
[3]: # Tensor criado diretamente dos dados: tipo inferido
     tensor_data = torch.tensor(data)
     tensor_data
```

```
[3]: tensor([[1, 2],
            [3, 4]])
```

```
[4]: #Criação de um array de dados numpy
     array_data = np.array(data)
     array_data
```

```
[4]: array([[1, 2],
           [3, 4]])
```

```
[5]: # Tensor criado a partir de um array numpy
     tensor_data = torch.from_numpy(array_data)
     tensor_data
```

```
[5]: tensor([[1, 2],
            [3, 4]], dtype=torch.int32)
```

```
[6]: # Tensor criado a partir de outro tensor
     # torch.ones_like: Tensor criado apenas com 1, mas mantendo
     # as propriedades (shape, datatype) do tensor de origem
     tensor_data2 = torch.ones_like(tensor_data)
     tensor_data2
```

```
[6]: tensor([[1, 1],
            [1, 1]], dtype=torch.int32)
```

```
[7]: # Tensor criado a partir de outro tensor
     # torch.rand_like: Tensor criado com valores aleatórios, mas mantendo
     # as propriedades (shape, datatype) do tensor de origem
     tensor_data3 = torch.rand_like(tensor_data, dtype=torch.float)
     tensor_data3
```

```
[7]: tensor([[0.9281, 0.1553],
            [0.5049, 0.7257]])
```

```
[8]: # Tensor criado com valores aleatórios
     tensor = torch.rand(3,4)
     tensor
```

```
[8]: tensor([[0.9548, 0.5378, 0.9561, 0.2762],
            [0.8587, 0.9297, 0.6753, 0.7459],
            [0.3279, 0.6952, 0.5605, 0.7285]])
```

```
[10]: #Mostrando as propriedades do tensor
      print(f"Shape do tensor: {tensor.shape}")
      print(f"Datatype do tensor: {tensor.dtype}")
      print(f"Dispositivo que o tensor está armazenado: {tensor.device}")
```

```
Shape do tensor: torch.Size([3, 4])
Datatype do tensor: torch.float32
Dispositivo que o tensor está armazenado: cpu
```

2.4 TREINAMENTO DE REDES NEURAIS

O treinamento de redes neurais é um processo iterativo cujo objetivo é ajustar os pesos e vieses dos neurônios de forma a minimizar o erro entre as previsões da rede e os valores reais (esperados). A função de erro, também chamada de função de perda, é uma métrica que mede a discrepância entre a previsão da rede e a verdade, e o objetivo do treinamento é reduzir essa perda ao máximo.

Para minimizar a função de perda, são utilizados algoritmos de otimização, sendo o mais comum o gradiente descendente. Este algoritmo ajusta os pesos da rede iterativamente, seguindo a direção oposta ao gradiente da função de perda. O gradiente indica a inclinação da função de perda em relação aos pesos, ou seja, mostra como modificar os pesos para reduzir a perda.

No treinamento, o processo se baseia nos seguintes passos:

- **Propagação direta (Forward Propagation):** Os dados de entrada são passados pela rede, e uma previsão é gerada na saída.
- **Cálculo da perda (Erro):** Com base na previsão gerada, a função de perda é calculada, comparando a previsão com o valor real esperado.
- **Retropropagação (Backpropagation):** Utilizando a regra da cadeia e o gradiente descendente, os erros são propagados de volta pela rede, ajustando os pesos de cada camada.
- **Atualização dos Pesos:** Os pesos e vieses são ajustados para reduzir a função de perda. Dependendo do algoritmo de otimização escolhido (como Adam, SGD, RMSprop), a forma como os pesos são ajustados pode variar.

Esse processo é repetido várias vezes (chamadas épocas) até que a função de perda atinja um valor mínimo aceitável ou os pesos se estabilizem. A cada época, a rede aprende mais sobre os padrões presentes nos dados de treinamento, ajustando seus parâmetros para melhorar a acurácia.

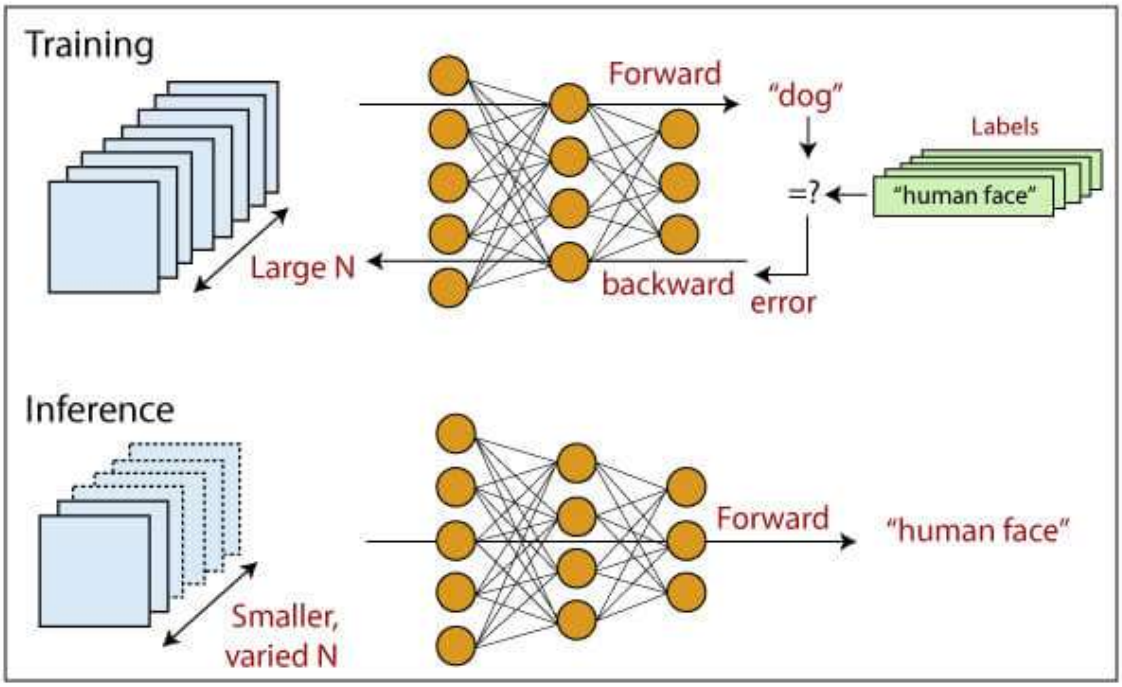


Figura 9: Etapas de Treinamento de uma rede MLP

Vamos agora explicar cada uma dessas etapas e mostrar como implementar no PyTorch.

2.5 PROPAGAÇÃO DIRETA (FORWARD PROPAGATION)

A propagação direta (forward propagation) é a primeira etapa no ciclo de treinamento de uma rede neural. Nesse processo, os dados de entrada são passados pela rede, camada por camada, até chegar à camada de saída, onde uma previsão é feita. As ativações de cada camada são calculadas com base nos pesos e nas funções de ativação, gerando uma previsão final na saída.

2.5.1 CRIANDO A ESTRUTURA DE CAMADAS

A construção de uma rede neural MLP no PyTorch segue uma estrutura modular, onde criamos classes herdando de torch.nn.Module, e definimos as camadas da

rede no método __init__. A propagação de dados (forward pass) é implementada no método forward.

Exemplo simples de uma rede neural totalmente conectada:

```
import torch
import torch.nn as nn

# Exemplo de Forward Propagation em uma rede simples
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.layer1 = nn.Linear(10, 5)  # 10 entradas, 5 neurônios
        self.layer2 = nn.Linear(5, 1)   # 5 entradas, 1 saída

    def forward(self, x):
        x = torch.relu(self.layer1(x))  # ReLU como função de ativação
        x = self.layer2(x)  # Saída sem função de ativação final
        return x

# Exemplo de uso
model = SimpleNN()
input_data = torch.randn(1, 10)  # Exemplo de um lote randômico com 1 exemplo e 10 features
output = model(input data)
```

Nesse exemplo, a rede possui:

- uma camada de entrada totalmente conectada (nn.Linear) com 10 neurônios, seguida de uma camada oculta com 5 neurônios.
- a função de ativação utilizada é a ReLU, que introduz não linearidades no modelo, permitindo que ele aprenda padrões mais complexos.
- o método forward define a passagem dos dados através das camadas da rede.

2.5.1 INICIALIZAÇÃO DOS PESOS

A inicialização dos pesos é um aspecto crítico no desempenho e treinamento eficiente de uma rede neural. Pesos mal inicializados podem resultar em diversos problemas, como convergência lenta ou a rede ficar presa em mínimos locais durante o processo de otimização. Se os pesos forem muito grandes, podem causar gradientes explosivos, e se forem muito pequenos, podem levar ao problema de gradientes desaparecendo (vanishing gradients), tornando o aprendizado extremamente ineficiente.

A inicialização de pesos aleatórios, mas com um controle adequado, é uma prática padrão para garantir que a rede comece o treinamento de maneira

eficiente. Estratégias modernas de inicialização de pesos, como a inicialização Xavier (também conhecida como Glorot initialization) ou a inicialização He, foram projetadas para lidar com esses desafios, promovendo uma distribuição de pesos que ajuda a manter os gradientes estáveis conforme eles são propagados para trás através da rede.

No PyTorch, você pode aplicar essa inicialização manualmente aos parâmetros do modelo, como demonstrado no código abaixo:

```
# Inicializando pesos manualmente
def inicializa_pesos(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)  # Inicialização Xavier
        nn.init.zeros_(m.bias)  # Bias inicializado como zero

# Aplicando a inicialização aos parâmetros do modelo
model.apply(inicializa_pesos)
```

Neste exemplo de trecho de código, a função inicializa_pesos poderia ser usada para inicializar os pesos de um modelo de uma rede neural. A inicialização empregada é a Xavier_uniform para os pesos e o bias está sendo inicializado com zero. A inicialização do bias como zero é uma prática comum, pois o bias é responsável por ajustar a ativação da rede e, ao começar com zero, evita que a rede tenha um comportamento enviesado antes do início do treinamento.

A inicialização Xavier (Glorot) é uma técnica que define a distribuição dos pesos de forma que a variância dos valores na saída de cada camada seja mantida aproximadamente constante. Isso ajuda a mitigar o problema dos gradientes desaparecendo ou explodindo, especialmente em redes com muitas camadas. Na inicialização Xavier, os pesos são amostrados de uma distribuição uniforme ou normal, com limites calculados a partir do número de unidades (neurônios) na camada anterior e na camada atual.

Para uma inicialização uniforme, a fórmula usada para calcular os limites da distribuição uniforme é:

$$\text{limite} = \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}$$

Onde:

- n_{in} é o número de entradas na camada.
- n_{out} é o número de saídas da camada.

2.5.2 FUNÇÃO DE ATIVAÇÃO

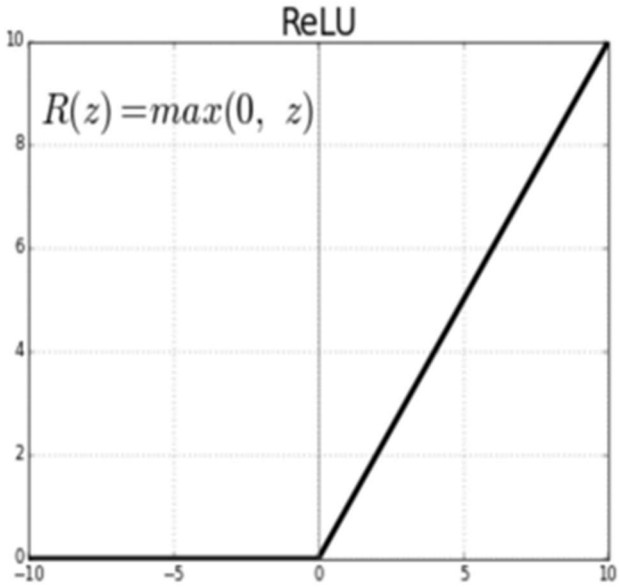
As funções de ativação em redes neurais desempenham um papel importante ao decidir se um neurônio deve ser "ativado" ou não, isto é, se sua saída será propagada para a próxima camada da rede. A principal função dessas ativações é introduzir não-linearidade ao sistema, o que permite que a rede modele relações complexas entre as variáveis de entrada e saída. Sem essa não-linearidade, as redes neurais seriam apenas combinações lineares das entradas, independentemente do número de camadas, e perderiam a capacidade de capturar padrões mais sofisticados nos dados.

As funções de ativação transformam a soma ponderada das entradas do neurônio em uma saída que será usada pela próxima camada da rede. A escolha da função de ativação pode influenciar significativamente o desempenho do modelo, e cada função é mais adequada para tipos específicos de tarefas

Como visto anteriormente, as funções mais comuns são:

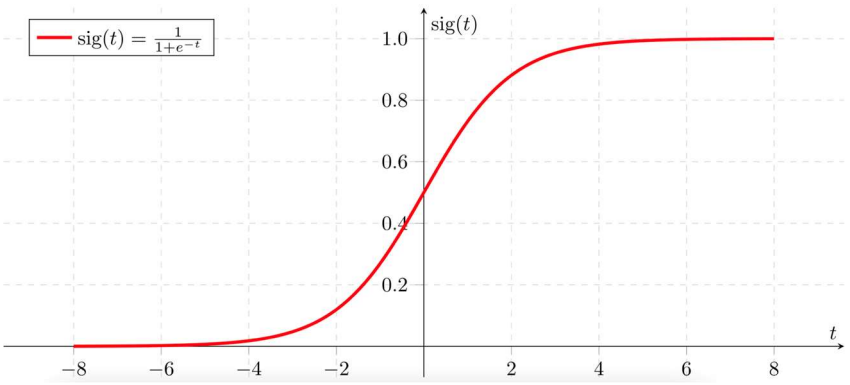
- **ReLU (Rectified Linear Unit)**: Retorna zero para valores negativos e o próprio valor para positivos. ReLU ajuda a evitar o problema de gradientes desaparecendo, comum em funções como a sigmoide e a tangente hiperbólica, tornando o treinamento mais rápido. Produz resultados no intervalo [0, ∞[. É uma função não linear.

$$f(x) = \max(0, x)$$



- **Sigmoide**: Transforma a saída em um valor entre 0 e 1, ideal para problemas de classificação binária. Boa para modelos que precisam de saídas probabilísticas, especialmente na última camada em problemas de classificação binária. Função não linear suave que é capaz de lidar com apenas duas classes.

$$f(x) = \frac{1}{1 + e^{-x}}$$



- **TanH (Tangente Hiperbólica)**: Produz valores entre -1 e 1, útil para redes que necessitam de saídas negativas e positivas.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Aqui está um exemplo de como as funções de ativação podem ser usadas em uma rede neural implementada em PyTorch:

```
# Exemplo de diferentes funções de ativação
sigmoid = nn.Sigmoid()
tanh = nn.Tanh()

# Aplicando as funções de ativação
input_data = torch.randn(1, 5)  # Exemplo de input
output_sigmoid = sigmoid(input_data)  # Função sigmoide
output_tanh = tanh(input_data)  # Função tanh
```

- nn.Sigmoid(), nn.Tanh(), e nn.ReLU(): Essas são as funções de ativação disponíveis no PyTorch, e elas podem ser aplicadas diretamente às entradas.

2.5.3 DROPOUT

Dropout é uma técnica utilizada em redes neurais para prevenir o overfitting—o fenômeno em que a rede se ajusta muito bem aos dados de treinamento, mas tem dificuldade em generalizar para novos dados : Neurônios co-adaptam seus aprendizados em relação a outros neurônios. O dropout evita que os neurônios fiquem altamente correlacionados, aumentando a capacidade da rede de generalizar melhor.

Durante o treinamento, o dropout "desativa" aleatoriamente uma fração dos neurônios em uma camada de uma rede neural, geralmente com uma probabilidade definida, como 50% ou 20%. Isso força a rede a distribuir o aprendizado entre todos os neurônios, em vez de depender excessivamente de neurônios específicos.

O efeito do dropout é similar ao de treinar diversas redes menores e independentes, e essa diversificação de aprendizado é recomposta durante a inferência, quando todos os neurônios estão ativos, mas com pesos escalados.

Durante a fase de inferência (avaliação/teste), o dropout não é aplicado. No entanto, para compensar a ausência do dropout durante a inferência, os pesos das conexões são escalados proporcionalmente à probabilidade de dropout utilizada durante o treinamento. Por exemplo, se o dropout com p=0.5 (50%) foi utilizado no treinamento, os pesos da rede são escalados por um fator de 1/(1-p) durante a inferência para manter a consistência do aprendizado.

No PyTorch, o dropout pode ser aplicado de maneira muito simples utilizando o módulo nn.Dropout. Você pode especificar a probabilidade p de "desativar" os neurônios durante o treinamento. No exemplo abaixo, a probabilidade de dropout

foi definida como 50% (p=0.5), o que significa que, em média, metade dos neurônios será desativada a cada iteração de treinamento.

```
# Aplicando dropout
class DropoutNN(nn.Module):
    def __init__(self):
        super(DropoutNN, self).__init__()
        self.layer1 = nn.Linear(10, 5)
        self.dropout = nn.Dropout(p=0.5)  # Dropout com probabilidade de 50%
        self.layer2 = nn.Linear(5, 1)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = self.dropout(x)  # Aplicando dropout após a camada 1
        x = self.layer2(x)
        return x

model = DropoutNN()
output = model(input data)
```

2.5.4 CÁLCULO DA PERDA

O cálculo da perda em redes neurais é um dos passos mais importantes durante o processo de treinamento. A função de perda (também chamada de função de custo) mede o quanto as previsões feitas pela rede neural diferem dos valores reais, ou seja, o quão longe o modelo está da solução correta. O objetivo do treinamento é minimizar essa função de perda, ajustando os pesos da rede de forma que as previsões se tornem cada vez mais precisas.

Existem diferentes funções de perda que podem ser usadas, dependendo do tipo de problema que se está tentando resolver. As mais comuns são:

- **Erro Quadrático Médio (Mean Squared Error - MSE):** Muito utilizada em problemas de regressão, o MSE calcula a média dos quadrados das diferenças entre as previsões da rede e os valores reais. A fórmula do MSE é:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  Onde y_i são os valores reais, $\hat{y}_i$ são os valores previstos pela rede e n é o número de exemplos. O MSE penaliza grandes erros de previsão, já que eleva ao quadrado as diferenças entre as previsões e os valores reais.

- **Entropia Cruzada (Cross-Entropy Loss):** Usada em problemas de classificação, especialmente classificação binária e multiclasse. A entropia cruzada mede a divergência entre a distribuição prevista pela rede (as probabilidades atribuídas às diferentes classes) e a distribuição real. Em uma tarefa de classificação binária, a fórmula da entropia cruzada é:

$$L = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Onde y_i é o rótulo real (0 ou 1), $\hat{y}_i$ é a probabilidade prevista pela rede de que a observação pertença à classe 1, e n é o número de exemplos. A entropia cruzada penaliza fortemente previsões com baixa confiança em relação à classe correta.

Esse código a seguir mostra como definir e calcular a função de perda em um modelo de rede neural usando PyTorch. Neste caso, duas funções de perda são apresentadas: Erro Quadrático Médio (MSE) e Entropia Cruzada Binária (BCE), cada uma apropriada para diferentes tipos de problemas.

```
# Função de perda MSE (Erro Quadrático Médio)
loss_fn_mse = nn.MSELoss()

# Função de perda Entropia Cruzada
loss_fn_cross_entropy = nn.BCELoss()  # Para classificação binária

# Exemplo de cálculo da perda
predictions = torch.sigmoid(model(input_data))  # Supondo classificação binária
target = torch.tensor([1.0])  # Rótulo verdadeiro

# Cálculo da perda usando entropia cruzada
loss = loss_fn_cross_entropy(predictions, target)
```

Aprenda Mais:

Redes Neurais com PyTorch.

Disponível em: https://colab.research.google.com/github/storopoli/ciencia-de-dados/blob/main/notebooks/Aula_18_b_Redes_Neurais_com_PyTorch.ipynb

Testando seu conhecimento

Questão 1: Qual o principal objetivo do processo de treinamento de uma rede neural?
a) Maximizar a função de perda.
b) Minimizar o erro entre as previsões da rede e os valores reais.
c) Aumentar o número de camadas da rede neural.
d) Eliminar a função de ativação para tornar o modelo linear.

Questão 2: O que é o PyTorch?
a) Um framework de otimização de redes neurais com foco em redes convolucionais.
b) Um framework de aprendizado profundo conhecido pela simplicidade e flexibilidade.
c) Uma biblioteca de matemática discreta aplicada em aprendizado de máquina.
d) Um framework especializado em redes neurais adversárias.

Questão 3: Qual a função dos tensores no PyTorch?
a) Representar funções de ativação em redes neurais.
b) Substituir as camadas ocultas nas redes neurais.
c) Representar os dados de entrada e saída, com suporte para GPUs.
d) Calcular automaticamente o gradiente descendente.

Questão 4: O que ocorre durante a propagação direta (forward propagation)?
a) O cálculo do gradiente de cada camada da rede neural.
b) Os dados são passados pela rede, camada por camada, gerando uma previsão final.
c) A rede ajusta os pesos dos neurônios para minimizar o erro.
d) O cálculo da perda e a retropropagação.

Questão 5: Por que a inicialização correta dos pesos é importante no treinamento de uma rede neural?
a) Porque uma má inicialização pode causar instabilidade no processo de retropropagação.
b) Porque impede a rede de aprender padrões complexos.
c) Porque garante que a rede tenha uma saída constante.
d) Porque determina a função de ativação utilizada pela rede.

Questão 6: O que a função de ativação ReLU (Rectified Linear Unit) faz?
a) Garante que a saída seja sempre entre 0 e 1.
b) Ativa o neurônio apenas se a entrada for negativa.
c) Retorna zero para valores negativos e o próprio valor para positivos.
d) Mapeia a entrada para um valor entre -1 e 1.

Questão 7: Quando a função sigmoide é mais adequada?
a) Em problemas de regressão contínua.
b) Quando a saída da rede deve ser entre -1 e 1.
c) Em problemas de classificação binária, onde a saída deve ser entre 0 e 1.
d) Para redes neurais recorrentes.

Questão 8: Qual é o principal objetivo do Dropout em uma rede neural?
a) Aumentar o número de neurônios em cada camada.
b) Desativar aleatoriamente uma fração de neurônios durante o treinamento para evitar overfitting.
c) Reduzir o tempo de treinamento da rede.
d) Aumentar a precisão da rede em problemas de classificação.

Questão 9: Qual das opções a seguir descreve corretamente o cálculo de perda usando o Erro Quadrático Médio (MSE)?
a) Penaliza pequenas diferenças entre as previsões e os valores reais.
b) Calcula a diferença média entre as previsões e os valores reais sem penalizar grandes erros.
c) Calcula a média dos quadrados das diferenças entre as previsões e os valores reais.
d) Gera a perda com base na probabilidade de uma classe específica.

Questão 10: Qual a função da Entropia Cruzada (Cross-Entropy Loss) em redes neurais?
a) Maximizar a probabilidade de todas as classes em problemas de classificação.
b) Medir a divergência entre a distribuição prevista e a distribuição real.
c) Gerar previsões de valores contínuos em problemas de regressão.
d) Reduzir o número de camadas da rede neural para evitar overfitting.

Questão 11: Qual das funções de ativação a seguir é mais adequada para redes neurais que necessitam de saídas tanto positivas quanto negativas?
a) Sigmoide
b) ReLU
c) TanH
d) Função degrau

Questão 12: Por que o uso de GPUs é vantajoso no PyTorch?
a) Porque permite a utilização de mais funções de ativação não-lineares.
b) Porque otimiza a execução de operações matemáticas envolvendo tensores de forma paralela.
c) Porque reduz o número de parâmetros treináveis na rede neural.
d) Porque permite que o PyTorch use menos memória durante o treinamento.

Respostas:

Questão 1: b)
Questão 2: b)

Questão 3: c)
Questão 4: b)
Questão 5: a)
Questão 6: c)
Questão 7: c)
Questão 8: b)
Questão 9: c)
Questão 10: b)
Questão 11: c)
Questão 12: b)

Desafio: Monte a arquitetura simples de uma rede neural totalmente conectada com três camadas ocultas densas com 100, 50 e 20 neurônios, respectivamente e nessa ordem. Coloque uma camada de dropout entre a segunda e a terceira e uma função de ativação ReLU depois da primeira camada oculta.

UNIDADE 3 RETROPROPAGAÇÃO (BACKPROPAGATION)

Nesta unidade será abordado a segunda etapa do processo de treinamento de uma Rede Neural MultiLayer Perceptron. Será apresentado o algoritmo retropropagação (ou backpropagation).

.

OBJETIVOS DA UNIDADE 3

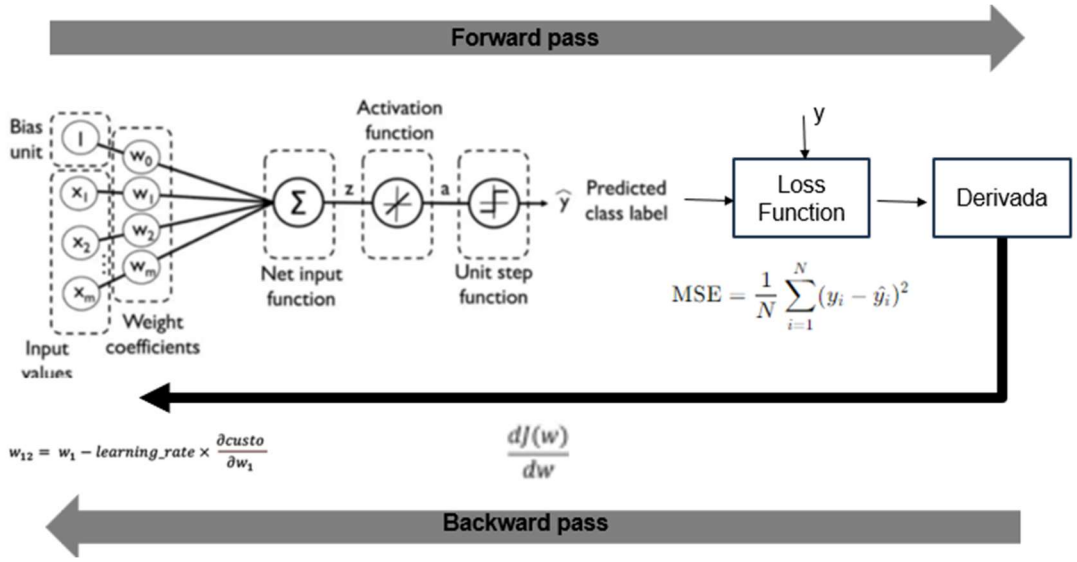
Ao final dos estudos, você deverá ser capaz de:

- Entender a retropropagação (ou backpropagation)
- Entender o processo de atualização dos pesos com base no erro
- Treinar uma rede neural simples utilizando PyTorch

A retropropagação (ou backpropagation) é o algoritmo central utilizado no treinamento de redes neurais artificiais. Ela permite que a rede ajuste seus pesos com base no erro entre a previsão do modelo e os valores reais esperados (rótulos). Esse ajuste é feito para minimizar a função de perda, que quantifica a diferença entre a previsão da rede e o valor correto. O objetivo principal é que, ao final do treinamento, a rede aprenda padrões nos dados de entrada que possam ser aplicados a novos dados, generalizando o aprendizado.

Como Funciona a Retropropagação

- **Propagação Direta (Forward Pass):** Como visto na Unidade 2, na fase de propagação direta, os dados de entrada passam pelas camadas da rede e produzem uma previsão na camada de saída.
- **Cálculo da Perda**: A diferença entre a previsão do modelo e o valor real é calculada usando uma função de perda. Quanto maior a perda, maior a diferença entre a previsão e o valor correto.
- **Retropropagação do Erro:** O erro calculado na função de perda é propagado para trás, desde a camada de saída até as camadas anteriores, usando a regra da cadeia (derivadas parciais). O objetivo é calcular o gradiente da função de perda em relação a cada peso da rede.
- **Ajuste dos Pesos:** Utilizando esses gradientes, os pesos da rede são ajustados em pequenas quantidades, de forma a minimizar a função de perda. Esse processo se repete a cada iteração, fazendo com que a rede fique melhor a cada passo.



3.1 GRADIENTE DESCENDENTE

O gradiente descendente é o método de otimização utilizado para ajustar os pesos da rede durante a retropropagação. O gradiente descendente procura a

direção que minimiza a função de perda, seguindo o gradiente negativo da função. Em outras palavras, ele ajusta os pesos na direção oposta ao gradiente, pois essa é a direção em que a perda diminui mais rapidamente.

Existem várias variantes do gradiente descendente, que podem ser aplicadas dependendo da natureza do problema e dos recursos disponíveis:

- **Gradiente Descendente Estocástico (SGD):** Neste método, os pesos são atualizados após cada exemplo de treinamento. Embora introduza variação nas atualizações (devido à aleatoriedade dos exemplos), o SGD pode acelerar o processo de convergência, especialmente em grandes conjuntos de dados.
- **Gradiente Descendente com Mini-batches:** Aqui, os dados de treinamento são divididos em pequenos lotes (mini-batches). Cada lote é usado para calcular o gradiente e atualizar os pesos. Isso combina a eficiência do SGD com a estabilidade do gradiente descendente em lote completo, garantindo um treinamento mais suave e eficiente.
- **Gradiente Descendente com Momento:** Introduz um termo de aceleração, chamado de "momento", que permite que o algoritmo evite mínimos locais e acelere a convergência. O momento atua como uma força que "empurra" a atualização dos pesos para direções que apresentam um gradiente consistente, acumulando o movimento de iterações anteriores.

O PyTorch oferece uma variedade de otimizadores prontos para uso, que implementam essas variantes do gradiente descendente. Alguns dos otimizadores mais comuns incluem:

- **SGD:** O otimizador de gradiente descendente estocástico básico.
- **Adam (Adaptive Moment Estimation):** Uma variante popular que combina o conceito de momentum com a adaptação de taxas de aprendizado para cada parâmetro. Ele ajusta a taxa de aprendizado com base na magnitude dos gradientes, o que o torna muito eficiente e amplamente utilizado.
- **RMSprop**: Otimizador que ajusta a taxa de aprendizado com base na média dos gradientes ao longo do tempo. É ideal para problemas onde os gradientes variam bastante em magnitude.

Aqui está um exemplo de como inicializar um otimizador no PyTorch:

```
# Otimizador SGD
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Loop de treinamento completo
for epoch in range(100):
    model.train()  # Modo treinamento
    optimizer.zero_grad()  # Zerar os gradientes

    output = model(input_data)  # Forward propagation
    loss = loss_fn(output, target)  # Cálculo da perda
    loss.backward()  # Backpropagation: cálculo do gradiente
```

Outro fatores que influenciam o treinamento é a taxa de aprendizado (learning rate). Um hiperparâmetro que define o tamanho do passo que o algoritmo de otimização dá ao ajustar os pesos. Uma taxa muito alta pode fazer o modelo "pular" o ponto ótimo, enquanto uma taxa muito baixa pode resultar em um treinamento muito lento.

Estratégias comuns para lidar com a escolha da taxa de aprendizado incluem:

- **Taxas de aprendizado adaptativas:** Algoritmos como o Adagrad, RMSprop e Adam ajustam a taxa de aprendizado dinamicamente durante o treinamento, de acordo com o comportamento do gradiente. Isso permite que o modelo faça ajustes mais eficazes, evitando problemas com taxas de aprendizado mal calibradas.
- **Decaimento da taxa de aprendizado:** Outro método é usar uma taxa de aprendizado que diminui gradualmente ao longo do treinamento, permitindo grandes ajustes no início e refinamentos menores à medida que o treinamento progride.

```
optimizer_adam = optim.Adam(model.parameters(), lr=0.001)
```

No processo de **atualização dos pesos** esses são ajustados com base no gradiente descendente para minimizar o erro na próxima iteração.

```
optimizer.step()  # Atualização dos pesos
```

Este ciclo se repete até que a função de perda atinja um valor mínimo aceitável ou até que o número máximo de iterações seja atingido. O objetivo é que a rede

seja capaz de generalizar seu aprendizado para novos dados, evitando o problema de overfitting (superajuste), onde o modelo "memoriza" os dados de treinamento e não consegue lidar com dados nunca antes vistos.

O código abaixo implementa o treinamento de um modelo de rede neural simples usando o PyTorch que segue todas essas etapas, considerando que a arquitetura SimpleNN já foi definida na Unidade 2.

```
#Exemplo usando Gradiente Descendente Estocástico (SGD)
import torch
import torch.optim as optim
# Modelo e dados de exemplo
model = SimpleNN()
input_data = torch.randn(10, 10)  # 10 exemplos, 10 features
target = torch.randn(10, 1)  # Saída esperada

# Função de perda
loss_fn = nn.MSELoss()

# Otimizador SGD (Gradiente Descendente Estocástico)
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Loop de treinamento
for epoch in range(100):
    model.train()
    optimizer.zero_grad()  # Zera os gradientes acumulados
    output = model(input_data)  # Propagação direta
    loss = loss_fn(output, target)  # Cálculo da perda
    loss.backward()  # Retropropagação do erro
    optimizer.step()  # Atualização dos pesos com base no gradiente
```

Importando bibliotecas

- torch: Biblioteca principal do PyTorch, usada para operações de tensores, definição de modelos e funções de perda.
- torch.optim: Contém otimizadores, como o SGD, usados para atualizar os pesos do modelo durante o treinamento.

Modelo e Dados de Exemplo

- model = SimpleNN(): Inicializa um modelo criado chamado SimpleNN. O modelo é definido separadamente e é uma rede neural simples com camadas totalmente conectadas.

- input_data: Gera um tensor aleatório com 10 exemplos de entrada, cada um com 10 características (features). Este é o conjunto de dados de entrada para o modelo.

- target: Gera a saída esperada (valores-alvo) para o treinamento. Este tensor tem 10 exemplos, com uma única saída por exemplo. O valor que o modelo tenta prever.

Função de Perda

- nn.MSELoss(): Define a função de perda como Erro Quadrático Médio (MSE). Esta função calcula a diferença entre a saída prevista pelo modelo e o valor esperado (target) ao quadrar o erro. O objetivo do treinamento é minimizar essa perda.

Otimizador

- optim.SGD: Cria o otimizador SGD (Gradiente Descendente Estocástico). Ele ajusta os pesos da rede com base nos gradientes calculados durante a retropropagação. O argumento model.parameters() indica que o otimizador deve atuar nos parâmetros do modelo (os pesos das camadas da rede neural).

- lr=0.01: A taxa de aprendizado é configurada para 0.01. Isso controla o tamanho dos passos que o otimizador dá ao ajustar os pesos.

Loop de Treinamento

- for epoch in range(100): O treinamento será realizado por 100 épocas, ou seja, o modelo passará 100 vezes por todo o conjunto de dados.

- model.train(): Coloca o modelo em modo de treinamento, o que é importante para certas operações que ocorrem de maneira diferente em comparação ao modo de inferência (como o dropout).

- optimizer.zero_grad(): Antes de cada iteração de treinamento, os gradientes acumulados dos parâmetros são zerados. Isso é necessário porque, no PyTorch, os gradientes são acumulados a cada chamada de backward().

Propagação Direta e Cálculo da Perda

- output = model(input_data): Passa os dados de entrada pelo modelo, realizando a propagação direta (forward pass), para gerar a saída prevista.

- loss = loss_fn(output, target): Calcula a perda (erro) entre a saída prevista pelo modelo (output) e os valores-alvo (target). A função de perda escolhida é o MSE.

Retropropagação e Atualização dos Pesos

- loss.backward(): Realiza a retropropagação. Isso calcula os gradientes da perda em relação a cada um dos pesos do modelo. Esses gradientes são usados para ajustar os pesos e minimizar a perda.

- optimizer.step(): Usa os gradientes calculados para atualizar os pesos da rede, conforme definido pelo algoritmo SGD. Cada passo move os pesos na direção que minimiza a função de perda.

Aprenda Mais:

Redes Neurais com PyTorch.

Disponível em: https://colab.research.google.com/github/storopoli/ciencia-de-dados/blob/main/notebooks/Aula_18_b_Redes_Neurais_com_PyTorch.ipynb

Testando seu conhecimento

Questão 1. Qual é o principal objetivo da retropropagação em redes neurais?
A) Melhorar a velocidade do treinamento
B) Minimizar a função de perda ajustando os pesos
C) Aumentar o número de camadas na rede
D) Reduzir o número de neurônios na camada de saída

Questão 2. Durante o processo de retropropagação, em qual etapa ocorre a atualização dos pesos?
A) Na propagação direta
B) No cálculo da perda
C) Na retropropagação do erro
D) Após o cálculo do gradiente da função de perda

Questão 3. Qual das seguintes variantes do gradiente descendente pode ser mais eficiente para grandes conjuntos de dados?
A) Gradiente descendente em lote completo
B) Gradiente descendente estocástico (SGD)
C) Gradiente com momento
D) Gradiente descendente com taxa de aprendizado fixa

Questão 4. Qual das opções abaixo descreve corretamente o papel do momento no gradiente descendente?
A) Introduz aleatoriedade nas atualizações dos pesos
B) Controla a taxa de aprendizado automaticamente
C) Ajuda a evitar mínimos locais e acelera a convergência
D) Calcula o erro da função de perda em tempo real

Questão 5. Uma taxa de aprendizado muito alta pode causar:
A) Convergência rápida e estável
B) Grandes saltos nos pesos, tornando o treinamento instável
C) A rede a "memorizar" os dados de treinamento
D) Um erro maior na função de perda a cada iteração

Questão 6. O que diferencia o gradiente descendente com mini-batches do gradiente descendente estocástico (SGD)?
A) Mini-batches atualiza os pesos após cada exemplo de treinamento
B) Mini-batches combina a eficiência do SGD com a estabilidade do gradiente descendente em lote completo
C) Mini-batches utiliza toda a base de dados para calcular o gradiente
D) Mini-batches ajusta automaticamente a taxa de aprendizado

Questão 7. O que pode ocorrer se a taxa de aprendizado for muito baixa durante o treinamento?
A) A rede convergirá rapidamente, mas com erro alto
B) O treinamento pode ser muito lento, com ajustes pequenos nos pesos
C) O modelo será incapaz de generalizar para novos dados
D) A função de perda não será calculada corretamente

Questão 8. Algoritmos como RMSprop e Adam são exemplos de:
A) Taxas de aprendizado fixas
B) Algoritmos que utilizam momento
C) Taxas de aprendizado adaptativas
D) Algoritmos de regularização

Questão 9. Qual é a função da propagação direta em redes neurais?
A) Calcular o gradiente da função de perda
B) Passar a entrada pela rede e produzir uma previsão
C) Ajustar os pesos após cada iteração
D) Minimizar a função de perda durante o treinamento

Questão 10. A função de perda em uma rede neural tem o papel de:
A) Ajustar os pesos automaticamente
B) Calcular o erro entre a previsão e o valor real
C) Controlar a taxa de aprendizado
D) Estabilizar as atualizações dos pesos

Questão 11. Qual das seguintes estratégias pode ajudar a evitar problemas com taxas de aprendizado mal calibradas?
A) Aumentar o número de camadas na rede
B) Usar decaimento da taxa de aprendizado
C) Remover neurônios das camadas ocultas
D) Aumentar o número de iterações no treinamento

Questão 12. Qual é o objetivo final do treinamento de uma rede neural?
A) Memorizar os dados de treinamento
B) Minimizar o erro apenas nos dados de teste
C) Generalizar bem para novos dados nunca antes vistos
D) Reduzir o número de neurônios para aumentar a eficiência

Questão 13. O termo "overfitting" em redes neurais refere-se a:
A) A capacidade de uma rede aprender rapidamente
B) O ajuste excessivo do modelo aos dados de treinamento, prejudicando a generalização
C) O uso excessivo de gradiente descendente
D) A minimização eficiente da função de perda

Questão 14. O que ocorre durante a retropropagação do erro em uma rede neural?
A) O gradiente da função de perda é calculado em relação aos pesos
B) A entrada é passada pela rede para gerar uma previsão
C) O número de iterações é aumentado
D) A função de ativação é ajustada para cada neurônio

Questão 15. O que diferencia o gradiente descendente com mini-batches do gradiente descendente em lote completo?
A) Lote completo usa apenas um exemplo de treinamento por vez
B) Mini-batches divide os dados em pequenos lotes, enquanto o lote completo usa todos os dados
C) Mini-batches ajusta a taxa de aprendizado automaticamente
D) Lote completo introduz variação nas atualizações

Respostas
Questão 1: Resposta correta: B
Questão 2: Resposta correta: D
Questão 3: Resposta correta: B
Questão 4: Resposta correta: C
Questão 5: Resposta correta: B
Questão 6: Resposta correta: B
Questão 7: Resposta correta: B
Questão 8: Resposta correta: C
Questão 9: Resposta correta: B
Questão 10: Resposta correta: B
Questão 11: Resposta correta: B
Questão 12: Resposta correta: C

Questão 13: Resposta correta: B
Questão 14: Resposta correta: A
Questão 15: Resposta correta: B

UNIDADE 4 MÉTRICAS DE AVALIAÇÃO DE REDES NEURAIS

Nesta unidade são apresentadas as principais métricas de avaliação de modelos de redes neurais para o problema de classificação.

OBJETIVOS DA UNIDADE 4

Ao final dos estudos, você deverá ser capaz de:

- Entender a utilização da principais métricas de classificação para validar um modelo treinado

A avaliação de redes neurais é que não pode ser deixada de lado pois é utilizada para verificar se o modelo treinado está desempenhando bem sua tarefa e generalizando adequadamente para novos dados. Nesta unidade, discutiremos as principais métricas utilizadas para avaliar redes neurais em problemas de classificação.

4.1 IMPORTÂNCIA DAS MÉTRICAS DE AVALIAÇÃO

As métricas de avaliação são utilizadas para medir a eficácia de um modelo. Elas fornecem uma forma objetiva de comparar o desempenho entre diferentes modelos e ajustar hiperparâmetros, como a taxa de aprendizado e o número de camadas ocultas.

Ao longo do treinamento, as métricas de desempenho ajudam a monitorar o progresso da rede, identificar overfitting e underfitting, e tomar decisões sobre a necessidade de ajustes no modelo.

As métricas mais comuns dependem do tipo de problema que o modelo está resolvendo, sendo que cada tarefa exige uma métrica específica ou um conjunto delas para avaliação adequada.

4.2 AVALIAÇÃO DE MODELOS DE CLASSIFICAÇÃO BINÁRIA

Nos problemas de classificação binária, o objetivo é atribuir um rótulo de uma entre duas classes, como "positivo" ou "negativo". As principais métricas de avaliação incluem:

4.2.1 ACURÁCIA

A Acurácia mede a proporção de previsões corretas em relação ao número total de amostras. Embora seja uma métrica amplamente utilizada, ela pode ser enganosa quando há desbalanceamento entre as classes.

$$Acurácia = \frac{\text{Número de previsões corretas}}{\text{Número total de amostras}}$$

4.2.2 MATRIZ DE CONFUSÃO

A Matriz de Confusão é uma tabela que resume as previsões feitas pelo modelo em comparação com os valores reais. Ela é composta por quatro valores:

- Verdadeiros Positivos (VP): A quantidade de previsões corretas da classe positiva.

- Falsos Positivos (FP): A quantidade de previsões incorretas da classe positiva.
- Verdadeiros Negativos (VN): A quantidade de previsões corretas da classe negativa.
- Falsos Negativos (FN): A quantidade de previsões incorretas da classe negativa.

Essa matriz permite o cálculo de métricas importantes como precisão e revocação.

4.2.3 PRECISÃO E REVOCAÇÃO

A Precisão mede a proporção de previsões positivas que são realmente positivas, ou seja, a capacidade do modelo de evitar falsos positivos.

$$Precisão = \frac{VP}{VP + FP}$$

A Revocação mede a proporção de verdadeiros positivos capturados pelo modelo, ou seja, sua capacidade de encontrar todas as instâncias da classe positiva.

$$Revocação = \frac{VP}{VP + FN}$$

4.2.4 F1-SCORE

O F1-Score é a média harmônica entre a precisão e a revocação. Ele oferece uma métrica equilibrada que é útil quando há desbalanceamento entre as classes.

$$F1 = 2 \times \frac{\text{Precisão} \times \text{Revocação}}{\text{Precisão} + \text{Revocação}}$$

Testando seu conhecimento

Questão 1. A retropropagação é uma técnica amplamente utilizada no treinamento de redes neurais artificiais. Qual é o principal objetivo deste algoritmo?
A) Maximizar o número de camadas ocultas na rede.
B) Minimizar a função de perda ajustando os pesos da rede.
C) Aumentar a taxa de aprendizado ao longo do treinamento.
D) Garantir que a rede generalize melhor ajustando os dados de entrada.

Questão 2. Durante o processo de retropropagação, qual etapa é responsável por ajustar os pesos da rede?
A) Cálculo da função de ativação.
B) Propagação direta.
C) Cálculo do erro.
D) Atualização dos pesos usando o gradiente descendente.

Questão 3. A retropropagação utiliza o método do gradiente descendente para ajustar os pesos da rede. Qual problema essa técnica visa resolver?
A) Reduzir o número de iterações no processo de treinamento.
B) Minimizar o erro acumulado ao longo de todas as amostras.
C) Aumentar o número de neurônios na camada de saída.
D) Garantir que os dados sejam pré-processados corretamente.

Questão 4. Qual das opções a seguir é uma métrica adequada para avaliar o desempenho de um modelo de classificação binária, em um cenário com classes desbalanceadas?
A) Acurácia.
B) Precisão.
C) Taxa de aprendizado.
D) Número de épocas.

Questão 5. A matriz de confusão é uma ferramenta importante para avaliar o desempenho de modelos de classificação binária. Qual dos elementos abaixo mede a quantidade de previsões incorretas da classe positiva?
A) Verdadeiros Positivos (VP).
B) Verdadeiros Negativos (VN).
C) Falsos Positivos (FP).
D) Falsos Negativos (FN).

Questão 6. Qual métrica é mais apropriada para medir a capacidade de um modelo em evitar falsos positivos em um problema de classificação binária?
A) Acurácia.
B) Precisão.
C) Revocação.

D) F1-Score.

Questão 7. Se um modelo de classificação binária apresenta alta revocação, o que isso indica sobre seu desempenho?
A) O modelo faz muitas previsões corretas da classe negativa.
B) O modelo identifica corretamente a maior parte dos exemplos positivos.
C) O modelo está evitando a maior parte dos falsos negativos.
D) O modelo tem alta acurácia.

Questão 8. O F1-Score é uma métrica que combina precisão e revocação. Por que ele é especialmente útil em problemas de classificação binária com classes desbalanceadas?
A) Porque leva em consideração apenas os verdadeiros positivos.
B) Porque ignora os falsos negativos.
C) Porque oferece um equilíbrio entre precisão e revocação.
D) Porque é equivalente à acurácia em problemas desbalanceados.

Questão 9. Em uma rede neural treinada com retropropagação, o que pode indicar que o modelo está sofrendo de "overfitting"?
A) Alta acurácia nos dados de treinamento e baixa acurácia nos dados de teste.
B) Baixa acurácia nos dados de treinamento e alta acurácia nos dados de teste.
C) Alto F1-Score em ambos os conjuntos de dados.
D) Alta revocação e baixa precisão nos dados de teste.

Questão 10. Qual das alternativas a seguir descreve corretamente o processo de retropropagação em uma rede neural?
A) Ajuste dos pesos baseado no erro calculado na camada de entrada.
B) Atualização dos pesos usando gradiente descendente a partir da camada de saída até a camada de entrada.
C) Ajuste dos pesos com base nas saídas corretas geradas pela rede.
D) Atualização das funções de ativação em cada neurônio.

Questão 11. Suponha que um modelo de rede neural está superestimando sistematicamente a classe positiva em um problema de classificação binária. Qual métrica seria mais adequada para entender esse comportamento?
A) Acurácia.
B) Precisão.
C) Revocação.
D) F1-Score.

Questão 12. No treinamento de redes neurais, a divisão da base de dados em conjuntos de treino e teste é fundamental para:
A) Aumentar o número de amostras disponíveis para o modelo aprender.

B) Avaliar a capacidade de generalização do modelo em dados novos.
C) Garantir que o modelo memorize todas as amostras de treinamento.
D) Reduzir a complexidade computacional durante o treinamento.

Questão 13. Em um problema de classificação binária, quando o modelo é muito conservador e raramente classifica uma amostra como positiva, é provável que a precisão seja:
A) Baixa, e a revocação alta.
B) Alta, mas a revocação baixa.
C) Alta, e a revocação também alta.
D) Baixa, e o F1-Score alto.

Questão 14. Durante o ajuste dos hiperparâmetros de uma rede neural, qual métrica deve ser monitorada para evitar "overfitting"?
A) Acurácia nos dados de treinamento.
B) Acurácia nos dados de validação.
C) Precisão nos dados de treinamento.
D) Taxa de aprendizado durante o treino.

Questão 15. No contexto de classificação binária, o que significa uma alta taxa de falsos negativos?
A) O modelo está errando frequentemente ao prever a classe negativa como positiva.
B) O modelo está errando frequentemente ao prever a classe positiva como negativa.
C) O modelo está classificando corretamente a classe positiva.
D) O modelo tem baixa acurácia nos dados de treinamento.

Respostas:

Questão 1. B)
Questão 2. D)
Questão 3. B)
Questão 4. B)
Questão 5. C)
Questão 6. B)
Questão 7. B)
Questão 8. C)
Questão 9. A)
Questão 10. B)
Questão 11. B)
Questão 12. B)
Questão 13. B)

Questão 14. B)
Questão 15. B)

Exemplo de código completo

UNIDADE 5 PROJETO DE TREINAMENTO DE UMA REDE NEURAL

Esta unidade apresenta um projeto completo de criação de uma rede neural MLP, treinamento e teste.

OBJETIVOS DA UNIDADE 5

Ao final dos estudos, você deverá ser capaz de:

- Entender o processo completo de treinamento de uma rede através de um estudo de caso

Nesta unidade você encontra o código que implementa uma rede neural simples usando PyTorch para resolver um problema de classificação binária. Ele também utiliza as funções de métrica do Scikit-learn para avaliar o desempenho do modelo.

O código está dividido em partes mas pode ser digitado na sequencia para ser executado.

5.1 IMPORTANDO OS PACOTES

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

- **torch**: Biblioteca PyTorch, usada para criar tensores, redes neurais e funções de otimização.
- **torch.nn:** Módulo que contém os blocos básicos para definir redes neurais, como camadas totalmente conectadas.
- **torch.optim:** Módulo de otimizadores, usado para atualizar os pesos da rede durante o treinamento.
- **sklearn.metrics:** Contém as funções de métrica usadas para avaliar o desempenho do modelo, como acurácia, precisão, revocação e F1-Score.

5.2 DEFININDO A ESTRUTURA DA REDE

```
# Exemplo simples de uma rede MLP com uma camada oculta
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 50)  # 10 features de entrada, 50 neurônios ocultos
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(50, 2)   # Saída binária (2 classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

- **SimpleNN:** Define uma rede MLP (MultiLayer Perceptron) simples com uma camada oculta e uma camada de saída.
- **nn.Linear(10, 50):** Primeira camada totalmente conectada (fully connected) com 10 neurônios de entrada e 50 de saída.

- **nn.ReLU():** Função de ativação ReLU é aplicada após a primeira camada para introduzir não-linearidade.
- **nn.Linear(50, 2):** A segunda camada tem 50 neurônios de entrada e 2 saídas (binárias, para classificação entre duas classes).

5.3 INSTANCIANDO A REDE

```
# Instanciando a rede neural
model = SimpleNN()
```

5.4 GERAÇÃO DOS DADOS SINTÉTICOS

```
# Criando um dataset sintético
X_train = torch.randn(100, 10)  # 100 exemplos de treino com 10 features cada
y_train = torch.randint(0, 2, (100,))  # Classes 0 ou 1

X_test = torch.randn(20, 10)  # 20 exemplos de teste
y_test = torch.randint(0, 2, (20,))   # Classes 0 ou 1
```

- **torch.randn(100, 10):** Gera um conjunto de dados de treino com 100 exemplos, cada um com 10 características aleatórias (normalmente distribuídas).
- **torch.randint(0, 2, (100**,)): Gera rótulos de classe binária (0 ou 1) para os exemplos de treino.
- **X_test e y_test:** Dados de teste com 20 exemplos e rótulos de classes binárias.

5.5 DEFININDO A FUNÇÃO DE PERDA E O OTIMIZADOR

```
# Definindo a função de perda (CrossEntropy para classificação binária) e o otimizador
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

- **nn.CrossEntropyLoss():** Função de perda para problemas de classificação.
- **optim.SGD():** Otimizador Stochastic Gradient Descent (Gradiente Descendente Estocástico) com taxa de aprendizado de 0.01.

5.6 DEFININDO A QUANTIDADE DE ÉPOCA

```
# Treinando a rede neural por 100 épocas
for epoch in range(100):
    model.train()
```

- **model.train():** Coloca o modelo no modo de treinamento, o que é importante se houver dropout ou camadas específicas que se comportam de maneira diferente entre treinamento e inferência.

5.7 EXECUTANDO O FORWARD

```
# Forward pass
outputs = model(X_train)
loss = criterion(outputs, y_train)
```

- **Forward pass:** Os dados de treinamento são passados pela rede, produzindo saídas (outputs), que são comparadas com os rótulos verdadeiros (y_train) para calcular a perda usando criterion.

5.8 EXECUTANDO O BACKPROPAGATION

```
# Backward pass e otimização
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

- **Backward pass**: O erro (perda) é retropropagado através do modelo, calculando os gradientes em relação aos pesos. O otimizador então ajusta os pesos para minimizar a perda.

5.9 MOSTRANDO A PERDA NA TELA

```
if (epoch+1) % 10 == 0:
    print(f'Época [{epoch+1}/100], Loss: {loss.item():.4f}')
```

- Print da perda a cada 10 épocas: Monitora o valor da perda durante o treinamento.

5.10 EXECUTANDO O MODELO COM OS DADOS DE TESTE

```
# Avaliação no conjunto de teste
model.eval()  # Colocando o modelo em modo de avaliação (desativa dropout, etc)
with torch.no_grad():  # Não precisamos calcular gradientes durante a inferência
    outputs = model(X_test)
    _, predicted = torch.max(outputs.data, 1)  # Pegando a classe com maior probabilidade
```

- **model.eval():** Coloca o modelo em modo de avaliação, o que desativa camadas como dropout ou batch normalization.
- **torch.no_grad():** Desativa o cálculo de gradientes durante a fase de inferência para economizar memória e acelerar o processo.
- **torch.max(outputs.data, 1):** Retorna o índice da classe com a maior probabilidade para cada exemplo de entrada.

5.11 TRANSFORMANDO OS TENSORES PARA ARRAY NUMPY

```
# Convertendo para numpy para usar as métricas do sklearn
y_test_np = y_test.numpy()
predicted_np = predicted.numpy()
```

- **Conversão para NumPy:** Converte os tensores PyTorch para arrays NumPy para usar as funções de métrica do Scikit-learn.

5.12 CALCULANDO AS MÉTRICAS DE CLASSIFICAÇÃO

```
# Calculando acurácia, precisão, revocação e F1-Score
accuracy = accuracy_score(y_test_np, predicted_np)
precision = precision_score(y_test_np, predicted_np)
recall = recall_score(y_test_np, predicted_np)
f1 = f1_score(y_test_np, predicted_np)

print(f"Acurácia: {accuracy:.2f}")
print(f"Precisão: {precision:.2f}")
print(f"Revocação: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")
```

- **Conversão para NumPy:** Converte os tensores PyTorch para arrays NumPy para usar as funções de métrica do Scikit-learn.
- **Acurácia (accuracy_score):** Mede a proporção de previsões corretas.

- **Precisão (precision_score):** Mede a proporção de verdadeiros positivos sobre todas as previsões positivas.
- **Revocação (recall_score):** Mede a proporção de verdadeiros positivos sobre todos os exemplos positivos reais.
- **F1-Score:** Média harmônica da precisão e revocação, útil para balancear as duas métricas.

FINALIZAR

Finalizamos o presente material sobre Redes Neurais, cobrindo conceitos essenciais e técnicas fundamentais para o entendimento e aplicação deste campo em crescente desenvolvimento. Iniciamos com uma introdução aos conceitos básicos de redes neurais, passando pela arquitetura de neurônios artificiais e suas variações, como Redes Neurais Feedforward, Convolucionais, Recorrentes, LSTM, Autoencoders e Transformers.

Avançamos para a compreensão das Redes Multilayer Perceptron, explorando a importância das camadas em redes neurais e o funcionamento do Perceptron multicamadas, consolidando as bases para redes mais complexas. Posteriormente, exemplificamos o uso do PyTorch, abordando sua estrutura básica, o processo de propagação direta, inicialização de pesos, funções de ativação, dropout e cálculo da função de perda.

Na sequência, discutimos a retropropagação, destacando o uso do gradiente descendente como técnica de otimização. Finalizamos com as métricas de avaliação de redes neurais, essenciais para a análise de desempenho de modelos, incluindo acurácia, matriz de confusão, precisão, revocação e F1-score, com foco na avaliação de modelos de classificação binária. Além disso, apresentamos um projeto completo de treinamento de uma rede neural artificial

Este material oferece um panorama técnico e fundamentado, que pode servir como base para a aplicação prática de redes neurais em diferentes contextos. Encorajo a continuidade nos estudos e a prática constante para aprimorar a compreensão e o domínio dessas técnicas. O campo das redes neurais está em constante evolução, e o aprofundamento contínuo é essencial para acompanhar as novas tendências e inovações tecnológicas.

Profa. Dra. Joelma de Moura Ferreira

Sobre a autora

Joelma de Moura Ferreira é doutora em Ciência da Computação pela Universidade Federal de Goiás, com mestrado em Ciência da Computação pela Universidade Federal de Goiás, MBA em Gerenciamento de Projetos pela Fundação Getúlio Vargas, especialização em Redes de Computadores pela Universidade Salgado de Oliveira, MBA em Tecnologia para Negócios: AI, Data Science e Big Data pela Pontifícia Universidade Católica do Rio Grande do Sul e graduação em Ciência da Computação pela Universidade Católica de Goiás. Tendo atuado por mais de 20 anos como docente de graduação e pós-graduação em diversas instituições de ensino superior, incluindo Faculdade Sul-Americana, Universidade Paulista, Faculdade Estácio de Sá de Goiás, Pontifícia Universidade Católica, Centro Universitário Alves Farias. Desempenhou a função de coordenadora do curso de graduação de Sistemas de Informação e dos cursos de pós-graduação em Gestão de Projetos, Gestão de Tecnologia da Informação e Arquitetura e Engenharia de Software no Centro Universitário Alves Faria, onde também exerceu a atividade de pesquisadora no Mestrado em Desenvolvimento R Fora do domínio acadêmico, exerce a função de Cientista de Dados no Tribunal de Justiça do Distrito Federal e Territórios.

Referências Bibliográficas

SILVA. et. al. **Inteligência artificial** SAGAH, 2019.

SICSÚ, A. L. et. al. **Técnicas de Machine Learning,** Blucher, 2023.

HAYKIN, Simon. **Redes Neurais: Princípios e Prática**. Bookman, 2007

FACELI, K. et. al **Inteligência Artificial: uma abordagem de aprendizado de máquina**, LTC, 2021

RUSSEL, Stuart J., NORVIG, P. **Inteligência Artificial - Uma Abordagem Moderna**, LTC, 2022.

GÉRON, Aurélien . **Mãos à Obra Aprendizado de Máquina com Scikit-Learn**

GRUS, Joel. **Data Science do Zero**, Alta Books, 2019.

KAUFMAN, D**., Desmistificando a inteligência artificial,** Autentica, 2022.