

Relatório 16 - Prática: Redes Neurais Convolucionais 1 (Deep Learning) (II)

Lucas Augusto Nunes de Barros

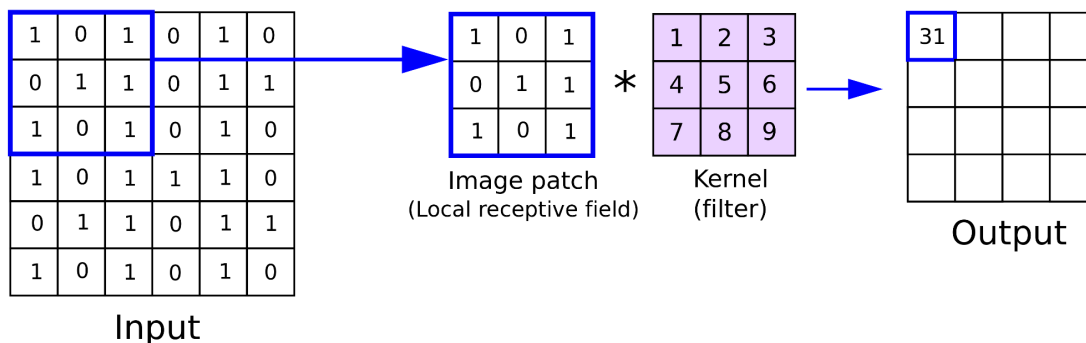
Descrição das atividades

1. Redes neurais convolucionais

1.1 Convolução

A convolução é uma operação matemática fundamental no contexto de redes neurais convolucionais (CNN - *Convolutional Neural Networks*), amplamente utilizadas em tarefas de visão computacional, como reconhecimento de imagens e detecção de objetos.

Na prática, a convolução em redes neurais envolve a aplicação de um filtro (também chamado de kernel) sobre uma imagem ou outro tipo de dado de entrada. O filtro é uma matriz pequena (por exemplo, 3x3 ou 5x5) que desliza sobre a imagem, realizando cálculos em cada região local.



1.2 Por que usar indexação 0?

Iniciar índices com zero na computação não é incomum e em redes neurais também é uma convenção amplamente adotada, por razões práticas e matemáticas. A indexação em zero facilita, por exemplo, o cálculo de endereços de memória, utilizados em linguagens menos abstratas, como o C no qual o Python é escrito.

Em redes neurais, os dados são frequentemente representados como tensores. A indexação iniciada em zero é consistente com a forma na qual os tensores são manipulados em bibliotecas como NumPy, TensorFlow e PyTorch, evitando erros e complicações durante a manipulação desses dados. Uma razão adicional é que em muitas operações matemáticas, como as convoluções, o uso de índices baseados em zero simplifica fórmulas, tal como é mostrado durante as aulas.

1.3 Convolução em Imagens Coloridas

A convolução de imagens coloridas é uma extensão da convolução aplicada a imagens em tons de cinza, mas com uma dimensão adicional para representar as cores.

$$(A * w)_{ij} = \sum_{i'=0}^{K-1} \sum_{j'=0}^{K-1} A(i + i', j + j') w(i', j')$$

Como as imagens coloridas precisam de uma terceira dimensão para representar as cores de cada pixel, sua fórmula precisa de um terceiro índice que irá percorrer a dimensão das cores.

$$(A * w)_{ij} = \sum_{c=1}^3 \sum_{i'=0}^{K-1} \sum_{j'=0}^{K-1} A(i + i', j + j', c) w(i', j', c)$$

O processo de convolução em uma imagem colorida, com dimensões HxWx3, Ao aplicar filtros diferentes de modo a preservar as dimensões espaciais da imagem, obtemos dois mapas de características, cada um deles com dimensões HxW. Esses mapas podem ser empilhados para formar um tensor HxWx2, onde o último valor representa a quantidade de filtros aplicados. Esse processo pode ser estendido para qualquer número de características, permitindo que a rede neural extraia uma variedade de padrões e informações da imagem.

Inicialmente, a convolução era aplicada apenas em imagens convertidas para escala de cinza, utilizando filtros 2D. Para imagens coloridas, a abordagem foi estendida, utilizando filtros 3D, porém ao realizar o produto escalar ao longo das três dimensões a entrada que era 3D, era convertida em uma saída 2D, impedindo a aplicação sequencial de convoluções e dificultando a operação.

Uma solução foi permitir que cada camada faça a extração de múltiplas características usando n filtros. Isso resulta em n saídas 2D que podem ser empilhadas formando uma saída 3D novamente. Isso permite convoluções em cascata que podem ter sua saída aplicada nas camadas densas de uma rede neural. No entanto, após convoluções subsequentes, a terceira dimensão irá representar as n características (uma para cada filtro), e não mais as 3 dimensões referente às cores. Essas camadas da terceira dimensão são chamadas de mapas de características (*feature maps*), onde cada imagem 2D indica a localização de uma característica específica.

A camada de convolução em uma rede neural opera de forma similar a uma camada densa, aplicando a convolução e incluindo um termo de *BIAS* juntamente com uma função de ativação, o que a torna eficiente para processar dados espaciais. Essa estrutura permite que a camada de convolução extraia características locais das imagens de maneira eficaz.

A convolução possui algumas vantagens em relação a simples multiplicação matricial, pois utiliza filtros que compartilham pesos ao longo da imagem, o que resulta em uma quantidade significativamente menor de parâmetros necessários para a operação. Essa abordagem é particularmente vantajosa ao processar dados, principalmente os de maior dimensionalidade, como imagens, em que padrões locais devem ser detectados da maneira mais eficiente possível.

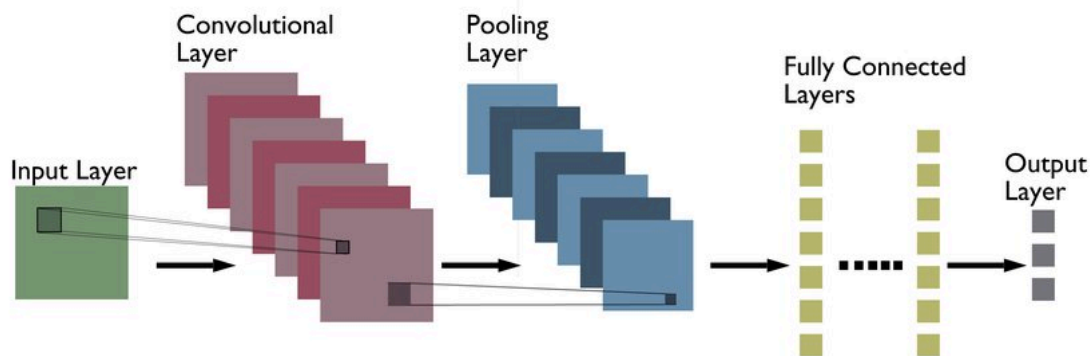
A diferença é facilmente notada quando comparamos a camada convolucional e a camada densa de uma rede neural, a camada densa exigirá uma quantidade substancialmente maior de parâmetros, pois não aproveita a localidade e a repetição de padrões presentes na imagem. A convolução, ao compartilhar pesos, permite que a rede aprenda a detectar padrões em diferentes locais sem a necessidade de parâmetros adicionais para cada posição da *feature*, tornando-a mais eficiente e adequada para tarefas como reconhecimento de padrões em imagens, ou seja, a convolução consegue extrair características independentemente de translações que essa característica possa sofrer dentro da imagem.

1.4 Arquitetura CNN

A estrutura básica de uma CNN (*Convolutional Neural Network*) é composta por camadas que recebem os dados de entrada, para extrair características e realizar tarefas como classificação ou detecção. As principais camadas são:

- **Entrada:** Recebe os dados pré-processados e no formato esperado pelo modelo.
- **Convolução:** Aplica filtros (*kernel's*) nos dados de entrada para detectar características. Os filtros compartilham pesos, o que reduz a quantidade de parâmetros e permite a detecção de padrões independentemente de translações.
- **Pooling:** Reduz a dimensão da saída da camada de convolução, preservando as características mais importantes.
- **Flatten:** Faz a conversão dos tensores de saída da camada de *pooling* em vetores de uma dimensão que serão recebidos como entrada pela camada densa.
- **Densa (*Fully Connected*):** Conecta todos os neurônios de uma camada à próxima, geralmente usada no final da rede para combinar as características extraídas e realizar a classificação.
- **Saída:** Produz as previsões finais.

Essas camadas são organizadas de forma sequencial, como mostrado a seguir, permitindo que a CNN aprenda de forma hierárquica.



O *pooling* é uma técnica de redução de dimensionalidade usada em redes neurais convolucionais para diminuir o tamanho espacial das imagens, mantendo as características mais importantes. Essa redução ajuda a diminuir o custo computacional e o número de parâmetros necessários, além de tornar a rede mais robusta a pequenas variações na entrada, como translações ou distorções. Em termos práticos, o *pooling* ajuda a criar invariância na posição, o que significa que a rede se torna menos sensível à posição exata de uma determinada característica, focando mais na presença da característica do que em sua localização.

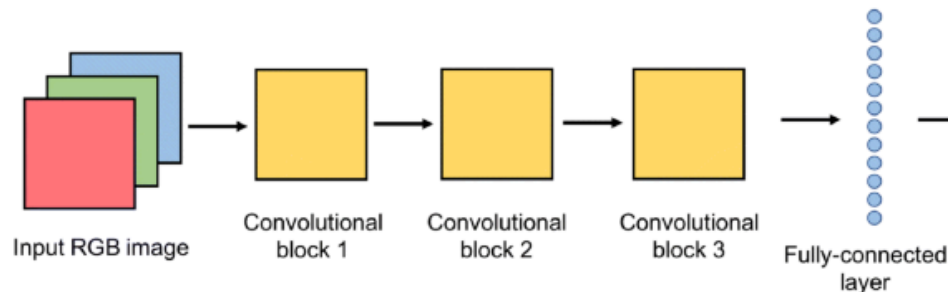
A combinação de convolução e *pooling* é uma estratégia comum em CNN's utilizada para reduzir o tamanho das imagens enquanto mantém o tamanho dos filtros constante. Após cada etapa de convolução e *pooling*, a imagem é reduzida, já os filtros são mantidos com as mesmas dimensões. Dessa forma, os filtros ocupam um espaço cada vez maior dentro da imagem, o que permite a extração das características seja hierárquicas, aprendendo primeiro padrões mais simples e posteriormente as características mais complexas.

Durante a redução de dimensionalidade da imagem por meio de técnicas como o *pooling*, parte da informação é de fato perdida, já que algumas localizações da imagem não são preservadas. No entanto, essa perda é compensada pelo aumento no número de mapas de características. Esses mapas capturam diferentes aspectos e padrões da imagem, permitindo que a rede aprenda representações mais abstratas, mesmo com a redução de resolução, o que ajuda o modelo a generalizar melhor.

Em redes neurais convolucionais, a escolha de hiperparâmetros é crucial e algumas convenções foram estabelecidas com o intuito de nortear os valores desses parâmetros. Usar filtros menores que as imagens de entrada, aumentar progressivamente o número de mapas de características ao longo das camadas (por exemplo, de 32 para 64, de 64 para 128), bem como repetir camadas subsequentes de convolução e *pooling* são práticas recomendadas durante o teste dos hiperparâmetros. Essas convenções ajudam a garantir que a rede aprenda de maneira eficiente.

Uma alternativa ao *pooling* em CNN's é o uso de convolução com *stride* (passo). Essa convolução permite diminuir a dimensionalidade da saída diretamente durante a operação de convolução, sem a necessidade de adicionar uma camada de *pooling*. Nessa técnica o filtro se desloca em passos maiores, resultando em uma redução no tamanho da saída.

Nessa técnica um *stride* igual a 1 significa que o filtro move-se apenas um pixel por vez, enquanto um *stride* de 2 faz com que o filtro pule dois *pixels* a cada movimento. Ao usar um *stride* maior que 1, a imagem de saída será reduzida. Com um *stride* maior, o filtro cobre menos posições, resultando em uma saída de menor tamanho. Dependendo da aplicação pode preservar mais informações do que o *pooling*.



Prosseguindo para a próxima camada do modelo atingimos a camada densa, ou totalmente conectada, devido a ligação entre todos os neurônios de uma camada com a próxima. Enquanto a saída de uma camada de convolução é tridimensional, uma camada densa espera um vetor unidimensional. Para resolver isso, é comum usar a operação de *flatten*, que transforma a saída tridimensional em um vetor unidimensional, permitindo que a rede processe a imagem de forma adequada.

Quando são aplicadas operações para reduzir o tamanho da imagem, o vetor resultante após o *flatten* pode ter dimensões muito diferentes dependendo do tamanho da imagem de entrada. Isso cria um problema, pois redes neurais densas não podem lidar com vetores de entrada de tamanhos variáveis, o que limita a flexibilidade do modelo.

1.5 Preparação do Código CNN

O Keras é uma especificação de API que permite a criação de bibliotecas de *Deep Learning* utilizando Python. Isso significa que qualquer pessoa pode desenvolver uma biblioteca e integrá-la à API do Keras, permitindo que outros usem Keras com essa nova biblioteca como *backend*. O criador do Keras passou a trabalhar no Google, o que aproximou a API do *TensorFlow*.

O Keras como biblioteca independente é instalado separadamente de seu *backend*, o que pode levar a incompatibilidades de versão. Por outro lado, o Keras como módulo interno do TensorFlow é integrado diretamente à biblioteca, garantindo que não haja incompatibilidades entre as versões do Keras e do TensorFlow. Essa integração facilita o desenvolvimento e a manutenção de modelos de *Deep Learning*, especialmente quando se utiliza o TensorFlow como *backend*.

Exemplos de formas diferentes de criar um modelo de IA usando o Keras/TensorFlow.

```
model = Sequential([
    Input(shape=(D,)),
    Dense(128, activation='relu'),
    Dense(K, activation='softmax'),
])

...
model.fit(...)
model.predict(...)
```

```
i = Input(shape=(D,))
x = Dense(128, activation='relu')(i)
x = Dense(K, activation='softmax')(x)

model = Model(i, x)

...
model.fit(...)
model.predict(...)
```

Exemplo de definição de uma camada de convolução.

`Conv2D(32, (3, 3), strides=2, activation='relu', padding='same')`



1.6 Rede Neural Convolutacional na base de dados *Fashion MNIST*

A primeira rede neural implementada foi treinada com a base de dados *Fashion MNIST*, que é um exemplo clássico de aplicação de *Deep Learning* para classificação de imagens. O *Fashion MNIST* é um conjunto de dados que contém 70.000 imagens em tons de cinza, divididas em 10 categorias de roupas e acessórios (camisetas, calças, sapatos, bolsas e etc). Cada imagem tem um tamanho de 28x28 pixels, semelhante ao famoso conjunto de dados MNIST de dígitos manuscritos, mas com a vantagem de representar um problema mais desafiador e realista.

Durante o treinamento, a CNN aprende a extrair características hierárquicas das imagens, começando com padrões simples, como bordas, e evoluindo para características mais complexas, como formas e detalhes específicos de cada categoria.

1.7 Rede Neural Convolutacional na base de dados CIFAR-10

A segunda implementação de rede neural foi treinada com a base de dados CIFAR-10, que é um exemplo clássico de aprendizado profundo para classificação de imagens. O CIFAR-10 consiste em 60.000 imagens coloridas de 32x32 pixels, divididas em 10 categorias, como aviões, carros, pássaros, gatos e cachorros. Diferente do *Fashion MNIST*, que utiliza imagens em tons de cinza, o CIFAR-10 apresenta o desafio adicional de trabalhar com as três dimensões de cores.

1.8 Data Augmentation

Os modelos de *Deep Learning*, que dependem de grandes volumes de dados, geralmente superam algoritmos tradicionais em tarefas complexas, como reconhecimento de imagens ou processamento de linguagem natural. No entanto, isso só é possível quando há dados suficientes e de alta qualidade para o treinamento.

Uma solução para aumentar a variabilidade dos dados é usar o conceito de *Data Augmentation*. É possível implementar manualmente transformações em imagens, porém o Keras oferece ferramentas que podem automatizar essas transformações de forma automática e randômica. Isso é particularmente útil ao trabalhar com grandes conjuntos de dados, onde o pré-processamento manual é impraticável.

1.9 Batch Normalization

A padronização dos dados antes de passá-los para algoritmos de *Deep Learning* é crucial para o desempenho do modelo, porém há um problema, apenas a primeira camada da rede neural recebe os dados normalizados. Após a transformação pela camada densa, os dados não permanecem normalizados. Isso ocorre porque as camadas densas alteram a distribuição dos dados. Para resolver isso, técnicas como *Batch Normalization* devem ser aplicadas entre as camadas densas da rede neural, garantindo que os dados permaneçam normalizados ao longo de todo o processo de treinamento.

A *Batch Normalization* (Normalização em Lote) pode atuar como uma forma de regularização, ajudando a prevenir o *overfitting* em redes neurais. Durante o treinamento, cada lote de dados tem uma média e um desvio padrão ligeiramente diferentes, já que os lotes são amostras do conjunto de dados. Essas variações introduzem um certo nível de "ruído" no processo de normalização. Esse ruído, por sua vez, tem um efeito benéfico, tornando a rede neural mais robusta, impedindo o *overfitting* e isso permite o modelo aprender a generalizar melhor, tornando-se mais resistente a flutuações nos dados.

Embora inicialmente discutida no contexto de camadas densas, a *Batch Normalization* é mais comumente aplicada após camadas convolucionais. Isso ocorre porque a normalização em lote ajuda a estabilizar e acelerar o treinamento, reduzindo a variação na distribuição das entradas das camadas durante o treinamento.

1.10 Melhorando a Rede Neural CIFAR-10

Para otimizar a rede neural foram alteradas algumas camadas do modelo, adicionando camadas de *BatchNormalization* e de *MaxPooling2D*. Após as alteração a rede ficou com a seguinte estrutura


```
i = Input(shape=x_train[0].shape)

x = Conv2D(32, (3, 3), activation="relu", padding="same")(i)
x = BatchNormalization()(x)
x = Conv2D(32, (3, 3), activation="relu", padding="same")(x)
x = BatchNormalization()(x)
x = MaxPooling2D(2, 2)(x)

x = Conv2D(64, (3, 3), activation="relu", padding="same")(x)
x = BatchNormalization()(x)
x = Conv2D(64, (3, 3), activation="relu", padding="same")(x)
x = BatchNormalization()(x)
x = MaxPooling2D(2, 2)(x)

x = Conv2D(128, (3, 3), activation="relu", padding="same")(x)
x = BatchNormalization()(x)
x = Conv2D(128, (3, 3), activation="relu", padding="same")(x)
x = BatchNormalization()(x)
x = MaxPooling2D(2, 2)(x)

x = Flatten()(x)
x = Dropout(0.2)(x)
x = Dense(1024, activation="relu")(x)
x = Dropout(0.2)(x)
x = Dense(num_classes, activation="softmax")(x)

model = Model(i, x)
```

2. Processamento de Linguagem Natural

2.1 *Embeddings*

Uma técnica comum em processamento de linguagem natural para representar textos e outros dados categóricos na forma numérica é o *One-Hot Encoding*, que cria um vetor de tamanho igual ao número de categorias. Cada palavra é mapeada para um índice específico nesse vetor, onde o valor 1 indica a presença da palavra, e os demais valores são 0. Essa abordagem é simples, mas pode se tornar ineficiente para vocabulários grandes, pois resulta em vetores de alta dimensionalidade e esparsos.

Esse conceito também pode ser aplicado para sequências de palavras. Se uma frase contém T palavras e cada palavra é representada por um vetor *One-Hot* de tamanho V , a frase inteira pode ser representada como uma matriz $T \times V$. Por exemplo, a frase "I like cats" pode ser codificada como `[[0, 0, 0, 1], [0, 1, 0, 0], [1, 0, 0, 0]]` em um vocabulário de 4 palavras. Essa representação mantém a estrutura da frase, mas ainda sofre com os problemas de esparsidade e alta dimensionalidade do One-Hot Encoding.

Uma alternativa mais eficiente ao *One-Hot Encoding* é o uso de *embeddings* que são representações mais densas e de menor dimensionalidade das palavras, onde cada informação é mapeada em um vetor contínuo em um espaço de N dimensões. Ao contrário do *One-Hot Encoding*, os *embeddings* capturam relações semânticas entre as palavras, como similaridades e diferenças, pois palavras com significados semelhantes tendem a ter vetores próximos no espaço de *embeddings*.

Um problema comum em processamento de linguagem natural é o tamanho do vocabulário V que possui facilmente tamanhos com 1 milhão ou mais de palavras ou tokens. Isso resulta em vetores de entrada de alta dimensionalidade, o que aumenta o custo computacional, além de tornar a matriz de pesos e a camada oculta consideravelmente grande, isso torna o processamento ineficiente.

Além de codificar as palavras é importante levar em consideração a estrutura geométrica dos dados. Baseado na ideia de que os vetores de dados pertencentes à mesma classe devem necessariamente estar próximos uns dos outros no espaço de características. Essa estrutura geométrica é essencial para que algoritmos de aprendizado de máquina possam generalizar e fazer previsões corretas. Por isso o uso de *One-Hot Encoding* é menos comum, pois resulta em vetores esparsos e sem estrutura geométrica, dificultando a aprendizagem do modelo em relação à semântica entre as palavras.

Para capturar relações semânticas entre palavras (por exemplo, "king" e "queen") é preciso que os dados tenham uma estrutura geométrica útil, que possa ser analisada e compreendida pelo modelo. Esse processo consiste em calcular pesos para cada, e esses pesos são aprendidos automaticamente durante o treinamento do modelo, usando algoritmos como o gradiente descendente.

Uma outra abordagem, alternativa ao treinamento comum, é o uso de vetores de palavras pré-treinados, gerados por outros algoritmos. Esses vetores são treinados em grandes volumes de texto e capturam relações entre palavras. Essa abordagem é útil pois aproveita o conhecimento já aprendido por outros modelos.


2.2 Preparação do Código para Processamento de Linguagem Natural

Primeiramente, é preciso converter as palavras em dados numéricos para então organizá-los em vetores. Esse processo envolve a criação de um mapeamento das palavras com índices inteiros, que são usados para acessar a matriz de *embeddings*.

No TensorFlow, sequências de texto são padronizadas para um comprimento fixo e qualquer sequência mais curta é preenchida com zeros. Como o valor zero é reservado para *padding*, ele não deve ser usado como índice de palavras, por essa razão os índices começam em um, pois assim se evita conflitos com o *padding*.

Vale lembrar que o *padding* é uma técnica usada em processamento de linguagem natural e outras áreas de aprendizado de máquina para garantir que todas as sequências de dados tenham comprimento igual. Isso é necessário pois modelos de redes neurais exigem entradas de tamanho fixo.

No TensorFlow o texto é tratado como uma string e não como uma lista de palavras. Para processar esse texto, é necessário primeiro transformá-lo em *tokens*. Após a "*tokenização*", os *tokens* são convertidos em números inteiros e, finalmente, em vetores de *embeddings*.

sentences = ["I like eggs and ham.", "I love chocolate and bunnies.", "I hate onions."]		sequences = [[1, 2, 3, 4, 5], [1, 6, 7, 4, 8], [1, 9, 10]]
---	---	--

Em processamento de linguagem natural, onde frases podem variar em comprimento, o *padding* preenche as sequências mais curtas com valores pré-determinados, geralmente zeros, até que todas as sequências tenham o mesmo comprimento. É possível controlar se o *padding* é adicionado no início ou no final da sequência.

Por outro lado, a truncagem é outra técnica usada para lidar com sequências de comprimentos variados, porém cortando as sequências mais longas para que se ajustem ao comprimento especificado. Também é possível controlar se a truncagem removerá dados do início ou do final da sequência, sendo que essa escolha depende da aplicação.

Ambas as técnicas são fundamentais em preparar os dados para modelos *Deep Learning*, garantindo que as entradas tenham tamanho consistente.

2.3 Processamento de Texto com TensorFlow

Nessa etapa do curso foi demonstrada a implementação em python, utilizando a biblioteca TensorFlow, de algumas das técnicas apresentadas durante as aulas teóricas. Durante a aula foi exemplificado os diversos tipos de *padding* e *truncate* que são aplicáveis em sequências de dados.

2.4 CNN para Texto

CNNs podem aprender textos ao tratar palavras ou tokens como uma imagem unidimensional, onde os *kernels* capturam padrões lineares, como combinações de palavras. Essa abordagem é eficaz para tarefas que dependem da identificação de padrões e ainda pode ser combinada com outras técnicas, como *embeddings* pré-treinados. Embora as CNNs (*Convolutional Neural Network*) não sejam tão boas quanto as RNNs (*Recurrent Neural Network*) para capturar relações ao longo prazo, elas são uma ferramenta poderosa e eficiente para muitas tarefas de processamento de linguagem natural.

A aplicação de convoluções em sequências de dados possui uma única dimensão não relacionada a características: o tempo. Assim como os pixels de uma imagem, dados próximos no tempo tendem a estar correlacionados de alguma forma. A convolução unidimensional é usada para capturar padrões nessas sequências

2.5 Classificação de Texto com CNN's

A classificação de texto com Redes Neurais Convolucionais (CNNs) é uma abordagem eficaz para tarefas como detecção de spam ou categorização de documentos. Embora originalmente projetadas para processamento de imagens, as CNNs podem ser adaptadas para textos ao representar palavras como vetores de *embeddings* e tratar sequências como "imagens" unidimensionais. Nessas sequências é possível aplicar os *kernels* da CNN, filtros convolucionais que capturam padrões locais, como letras e sequências, ou seja, palavras.

3. Convolução e Suas Aplicações

3.1 Exemplos de Convolução da Vida Real

São abordados também exemplos reais de convolução, como a aplicação em processamento de áudio. Efeito como *eco*, *reverb* e *wahwah* são exemplos de operações de convolução aplicadas em filtros que modificam o sinal de entrada para criar efeitos sonoros. Esses filtros demonstram como a convolução pode ser usada para transformar dados de maneira controlada, mantendo a essência do sinal original enquanto adiciona características específicas. Esses exemplos ilustram que a convolução não se limita a imagens, mas é uma técnica versátil aplicável a diversos tipos de dados, como áudio, vídeo e até séries temporais.

A convolução pode ser comparada com uma "caixa preta" que aplica um efeito desejável à sua entrada. No contexto de processamento de imagens, por exemplo, essa caixa preta pode ser vista como um filtro que transforma a imagem de entrada, destacando ou suprimindo certas características. Essa analogia ajuda a entender a convolução como uma operação que modifica os dados de maneira controlada, seja para realçar bordas, suavizar texturas ou extrair padrões específicos.

A convolução pode ser vista de várias perspectivas. Em aplicações práticas, como efeitos especiais em áudio ou vídeo, a convolução é usada para modificar diretamente a entrada. No entanto, no contexto de *Deep Learning*, o objetivo não é adicionar efeitos, mas transformar a entrada em uma representação útil. Nesse caso, a convolução atua como uma ferramenta para extrair características, permitindo que a rede neural aprenda padrões complexos e generalize bem para novos dados.

3.2 Guia para Iniciantes em Convolução

A primeira etapa de aula apresenta um guia introdutório sobre convolução. O conceito de *padding* é brevemente abordado, que é usado para controlar o tamanho da saída de uma operação de convolução. Sem *padding*, o tamanho da saída é reduzido para $(N - k + 1)$, onde N é o tamanho da entrada e k é o tamanho do filtro. No entanto, com o uso de *same padding*, zeros são adicionados ao redor da entrada para manter o tamanho da saída igual ao da entrada. Já o *full padding* adiciona ainda mais zeros, permitindo que o filtro cubra todos os *pixels* da entrada, resultando em uma saída maior $(N + k - 1)$. Essas técnicas são amplamente utilizadas para ajustar a dimensionalidade dos dados em redes neurais convolucionais.

3.3 Visões Alternativas da Convolução

Existem diferentes perspectivas para entender a convolução, cada uma oferecendo diferentes *insights*. Compreender essas alternativas ajudam a construir uma compreensão mais sólida da convolução. O professor do curso inclusive sugere que, para um entendimento mais aprofundado, um curso de processamento de sinais pode ser benéfico, já que a convolução é um conceito fundamental.

Uma dessas alternativas mostra como a convolução pode ser vista como uma forma de multiplicação matricial. Nessa representação, o sinal de entrada é multiplicado por uma matriz gerada a partir do filtro. A matriz resultante é construída de forma que cada linha represente uma versão deslocada do filtro, permitindo que a operação de convolução seja realizada como uma multiplicação de matrizes.

No contexto de deep learning, a função de correlação cruzada é uma ferramenta essencial para analisar a relação entre diferentes séries temporais ou

seqüências de dados. Já em redes neurais, especialmente em modelos de processamento de sinais e visão computacional, a correlação cruzada é frequentemente utilizada para identificar padrões e dependências entre conjuntos de dados. Um exemplo desse uso em redes convolucionais (CNNs) é a operação de convolução, que é fundamental para extrair características de imagens, podendo ser vista como uma forma de correlação cruzada.

A correlação cruzada também é útil em tarefas de previsão, onde o objetivo é prever valores futuros com base em dados históricos. Em modelos de *Deep Learning* para previsão de séries, a correlação cruzada pode ajudar a entender como diferentes variáveis temporais influenciam umas às outras. Essa é uma tarefa crucial para melhorar as previsões de modelos complexos, especialmente em aplicações que possuem muitas variáveis envolvidas.

A autocorrelação avalia a correlação de uma série temporal com uma versão deslocada de si mesma ao longo do tempo. No contexto de *Deep Learning*, a autocorrelação é particularmente útil para identificar padrões repetitivos e dependências temporais em dados sequenciais. Isso é crucial para melhorar a capacidade do modelo de prever valores futuros com base em observações passadas.

4.1 Convolução em Imagens 3D

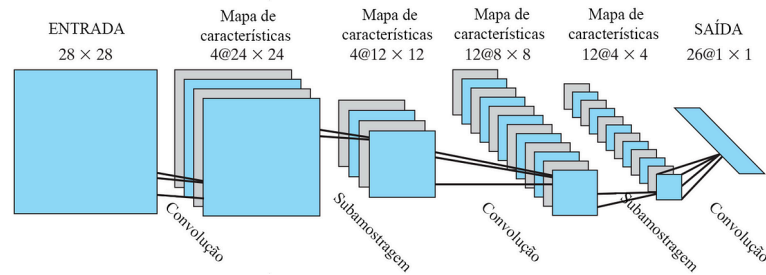
A detecção de bordas é uma técnica fundamental em visão computacional, utilizada para identificar transições significativas de intensidade em uma imagem, que geralmente correspondem a bordas de objetos. No contexto de *Deep Learning*, filtros são aplicados para detectar bordas nas imagens, que são mudanças repentinas nos valores dos *pixels*.

Uma questão importante surge ao aplicar múltiplos filtros: como combinar os mapas de características resultantes? Nas redes convolucionais, cada *kernel* gera um mapa de características, esses mapas formam a camada de convolução. Isso permite que a rede aprenda a integrar informações de diferentes orientações de bordas e outras características espaciais. A combinação desses mapas de características é crucial para o bom desempenho do modelo.

A operação de convolução para imagens tridimensionais é essencial para processar dados complexos, como imagens coloridas. Originalmente, assumimos que uma imagem é bidimensional, mas imagens coloridas têm três canais (vermelho, verde e azul), resultando em uma imagem tridimensional. Isso significa que cada pixel agora possui três valores, correspondentes às intensidades de cada canal de cor.

4.2 Maneiras de Rastreamento em CNN's

Em uma CNN típica, os dados de entrada, como uma imagem de dimensões, são processados por uma série de camadas convolucionais de *Pooling*, seguidas por camadas totalmente densas. Cada camada convolucional gera mapas de características que capturam padrões, enquanto as camadas de pooling reduzem a dimensionalidade, preservando as características mais relevantes. Ao final, as camadas totalmente conectadas integram essas características para realizar a classificação.



Uma rede feedforward, ou rede neural feedforward, é um tipo fundamental de arquitetura de rede neural onde a informação flui em uma única direção, da entrada para a saída, sem ciclos ou loops.

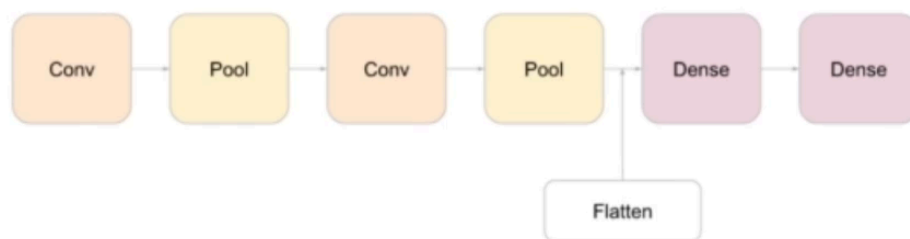
Uma rede feedforward é como uma linha de montagem para processar informações. Imagine que você tem uma máquina que recebe uma foto e precisa decidir se é um gato ou um cachorro. A rede feedforward faz isso passando a foto por várias etapas (camadas) de processamento.

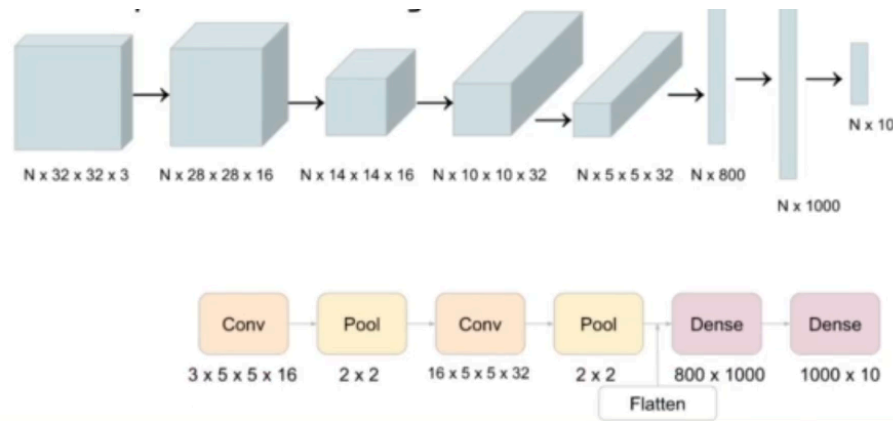
1º Entrada: A foto entra na rede. Cada pixel da foto é pré-processado para ser usado como entrada da rede.

2º Camadas Escondidas: A foto passa pelas camadas onde a máquina faz cálculos para entender padrões e extraí-los. Cada camada executa uma operação matemática que ajusta o peso das *features*.

3º Saída: No final, o modelo gera uma resposta: gato ou cachorro.

Full CNN





A escolha de hiperparâmetros como o tamanho do filtro não são aprendidos durante o treinamento, mas sim ajustados com base em experimentação e técnicas como *grid search* ou busca aleatória. À medida que a rede avança, camadas de *pooling* reduzem o tamanho da imagem enquanto o número de mapas de características aumenta, isso permite o aprendizado de uma maior variedade de características.

5. CNNs avançadas e como projetar a sua própria

Uma estrutura comum em redes neurais convolucionais consiste em duas partes principais: uma série de camadas convolucionais seguidas por uma série de camadas totalmente conectadas. As camadas convolucionais são responsáveis por extrair características espaciais da entrada enquanto as camadas totalmente conectadas integram essas características para realizar tarefas como classificação.

Entendidos alguns padrões sobre como as redes neurais podem ser organizadas, foram exploradas as estruturas de algumas CNN's famosas, como a ImageNet, VGG, AlexNet, GoogleNet.

6. 1 Erro Médio ao Quadrado (Mean Squared Error)

O Erro Quadrático Médio (MSE - *Mean Squared Error*) é uma métrica comum quando se precisa medir a diferença entre os valores previstos e reais. Do ponto de vista probabilístico, o MSE é útil porque penaliza erros maiores de forma mais significativa, o que ajuda a preparar o modelo para entender outras funções de perda, como a entropia cruzada.

A razão pela qual o erro é elevado ao quadrado no MSE é para garantir que os erros sejam sempre positivos. Se os erros não fossem positivos, os sinais opostos poderiam se cancelar, levando a uma interpretação enganosa de que a taxa de erros do modelo é baixa.

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

O Erro Absoluto Médio (MAE - *Mean Absolute Error*) também é uma opção válida, principalmente em contextos onde valores discrepantes devem ter menos influência. De toda forma, o MSE é mais prático em muitos casos, pois é diferenciável em todos os pontos, o que facilita a otimização por gradiente descendente. Além de que ele considera erros maiores como sendo mais relevantes.

Maximum Likelihood Estimation é um método estatístico utilizado para estimar os parâmetros de um modelo probabilístico. O objetivo é encontrar valores de parâmetros que maximizem a probabilidade de que os dados observados tenham sido gerados pelo modelo. Esse conceito é frequentemente ensinado em cursos introdutórios de probabilidade e estatística.

O processo consiste em definir uma função que representa a probabilidade dos dados observados em um conjunto de parâmetros. Para uma distribuição Gaussiana, a função de *likelihood* (verossimilhança) é baseada na fórmula da distribuição normal. Após definida a função utilizada, a próxima etapa é encontrar valores parâmetros que maximizam essa função. Isso geralmente é feito tomando o logaritmo da função escolhida e derivando em relação aos parâmetros para encontrar o ponto máximo.

$$L = \prod_{i=1}^N p(x)$$

$$L = \prod_{i=1}^N \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_i - \mu)^2}{2\sigma^2}\right)$$

Por exemplo, se for necessário modelar as alturas dos alunos de uma turma usando uma distribuição Gaussiana, a distribuição teria dois parâmetros: a média (μ) e a variância (σ^2). Usando a função MLE é possível estimar os parâmetros de forma que a probabilidade observada nas alturas dos alunos seja a maior possível. Nas equações descritas acima, X_i são constantes (dados coletados). Esses valores são fixos e não variam durante o processo de estimação.

A média (μ) é o parâmetro estimado (a média da distribuição Gaussiana, no caso das alturas). Esse é o valor que varia durante o processo de maximização. Já a variância (σ^2) é outro parâmetro da distribuição Gaussiana, que dependendo da situação pode ter seu valor fixo ou estimado. A ideia fundamental é estimar quais valores de μ e σ^2 que maximizam o valor de L .

Para maximizar a função de *likelihood* L em relação a μ , podemos usar cálculo, mas maximizar diretamente a função L pode ser complicado devido à sua complexidade. Em vez disso, uma abordagem mais inteligente é usar o logaritmo da função, o que simplifica os cálculos. Portanto, maximizar $\ln(L)$ é equivalente a maximizar a própria função L .

Derivando a função de verossimilhança em relação a μ , obtém-se a equação:

$$\frac{dl}{d\mu} = \left(\sum_{i=1}^N \frac{1}{\sigma^2} (x_i - \mu) \right)$$

Igualando a derivada a zero e resolvendo para μ , chega-se a estimativa de máximo da função de *likelihood*. Como a função em questão pode ser descrita na forma:

$$l = \left(\sum_{i=1}^N \left\{ \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} (x_i - \mu)^2 \right\} \right)$$

Ela pode ser reescrita, de forma simplificada, da seguinte maneira:

$$l = C1 - C2 \sum_{i=1}^N (x_i - \mu)^2$$

Onde C1 e C2 são constantes positivas e não influenciam a maximização. Portanto, podemos simplificar a expressão.

$$C1 = 0$$

$$C2 = \frac{1}{N}$$

Dessa forma, a equação que deve ser maximizada é:

$$l = - \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2$$

6. 2 Função de Perda Binária : Binary Cross Entropy

A Entropia Cruzada Binária (Binary Cross-Entropy, BCE) é a função de perda padrão para problemas de classificação binária. Ela mede a diferença entre as probabilidades previstas pelo modelo e os rótulos reais. Sua fórmula é:

$$Loss = -\frac{1}{N} \sum_{i=1}^N \{y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)\}$$

A Entropia Cruzada Binária (BCE) é a função de perda ideal para classificação binária porque está de acordo com a distribuição de Bernoulli, que é responsável por modelar eventos binários discretos, como lançamentos de moeda (cara=1, coroa=0).

O log da função de verossimilhança para uma distribuição de Bernoulli (como lançamentos de moeda) é dada por:

$$l = \log L = \sum_{i=1}^N \{x_i \cdot \log(\mu) + (1 - x_i) \cdot \log(1 - \mu)\}$$

Onde:

- x_i são os dados observados
- μ é a probabilidade de sucesso

Derivando e igualando a zero, encontra-se a estimativa de máxima verossimilhança:

$$\frac{\partial l}{\partial \mu} = \sum_{i=1}^N \frac{x_i}{\mu} - \frac{1-x_i}{1-\mu} = 0$$

Obtendo a estimativa de sucesso do evento analisado por:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i$$

A conexão entre essas funções justifica o uso da função de perda usada para classificação binária: com ela é possível encapsular a teoria estatística e utilizar de técnicas de otimização via gradiente descendente.

6.3 Função de Perda Multiclasse : Categorical Cross Entropy

A *Categorical Cross-Entropy* é uma função de perda oriunda da extensão natural da *Binary Cross-Entropy* para problemas com múltiplas classes. Assim como a entropia binária deriva da distribuição de Bernoulli, a versão categórica surge da distribuição categórica, que é uma generalização da distribuição de Bernoulli para K classes.

	<i>Binary Cross-Entropy</i>	<i>Categorical Cross-Entropy</i>
Classes	2	$K > 2$
Distribuição	Bernoulli	Categórica
Função de Saída	Sigmoide	Softmax

A entropia cruzada categórica é fundamental para classificação multiclasse, como já foi visto, o uso da codificação *one-hot encode* é uma forma de ajustar os dados para o modelo. Para classificação multiclasse com muitos rótulos ou grandes volumes de dados, esse tipo de codificação revela várias limitações críticas como alto consumo de memória (devido a quantidade de zeros armazenados), cálculos redundantes, além do tempo gasto na conversão de dados categóricos para *one-hot*.

Uma alternativa é a dupla indexação, essa abordagem melhora significativamente a eficiência computacional. Essa otimização é especialmente crítica em problemas com centenas de classes. Implementações modernas de frameworks de deep learning já adotam essa abordagem por padrão. A Sparse Categorical Cross-Entropy é o nome dessa variação projetada para eliminar a ineficiência do *one-hot encode*.

7.1 Gradiente Descendente

O Gradiente Descendente (*Gradient Descent*) é um algoritmo fundamental para o treinamento de modelos de *machine learning*. Sua importância vem do fato de ser uma ferramenta útil para resolver o problema central do aprendizado de máquina, que é encontrar os pesos dos parâmetros que minimizam uma função.

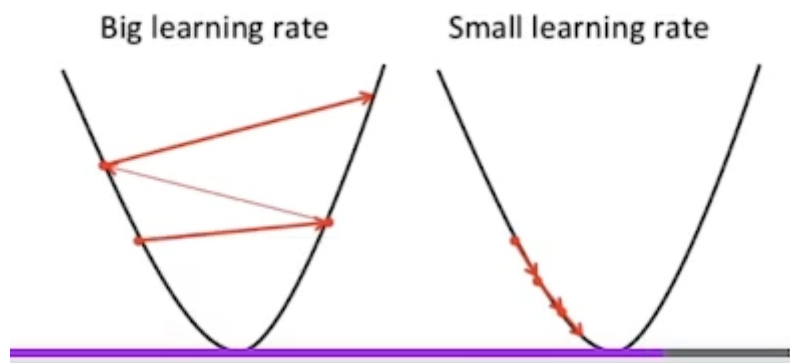
Matematicamente, o gradiente é um vetor N -dimensional, onde N é o número de parâmetros (variáveis) da função analisada. Cada parâmetro da função é uma dimensão do espaço N -dimensional que o vetor Gradiente ocupa, e cada uma das componentes deste vetor é dada pela derivada parcial da função estudada em relação a variável daquela dimensão. As derivadas parciais indicam a direção da maior taxa de variação da função em relação a determinada variável.

A definição do valor de cada uma das variáveis é feita de forma aleatória, inicialmente, e após o início do cálculo do Gradiente Descendente os valores são

atualizados com base na Taxa de Aprendizagem, um hiperparâmetro que controla o tamanho do passo, ou seja, que determina a variação no valor do parâmetro.

Esse cálculo é feito derivando a função original de maneira a obter as derivadas parciais em relação a todas as variáveis. Como a derivada pode ser interpretada graficamente como a inclinação da reta tangente em um ponto da função, é fácil compreender porque as derivadas devem ser iguais a zero, pois uma vez que não há inclinação da reta tangente, significa que atingimos um vértice da função, seja local ou global.

Um problema comum em *Machine Learning* envolve a otimização de equações complexas ou de alta dimensionalidade que não podem ser resolvidas analiticamente. Em vez de tentar resolvê-las diretamente, algo geralmente impossível ou muito trabalhoso, é mais fácil utilizar métodos numéricos que aproximam a solução através de iterações. Através de um método numérico é possível calcular o gradiente (vetor de derivadas parciais) e ajustar os parâmetros na direção oposta (afinal o gradiente é descendente), reduzindo progressivamente (seguindo a métrica da Taxa de Aprendizagem) o valor da função. A abordagem iterativa permite encontrar mínimos locais mesmo quando não há uma solução analítica.



Dois hiperparâmetros controlam esse processo: o número de épocas (iterações completas sobre os dados) e a taxa de aprendizado. O número de épocas deve ser suficiente para garantir a convergência do custo, enquanto a taxa de aprendizagem deve ser cuidadosamente equilibrada, pois uma taxa grande demais causa oscilações infinitas, enquanto valores muito pequenos tornam o treinamento excessivamente lento. A escolha desses valores geralmente envolve tentativa e erro, embora existam métodos automatizados que podem ser utilizados.

7.2 Gradiente Descendente Estocástico (*Stochastic Gradient Descent*)

O *Stochastic Gradient Descent* (SGD) é uma variação do algoritmo clássico de *Gradient Descent* (GD), amplamente utilizado em *Machine Learning* e *Deep Learning* para treinar modelos. A principal diferença está na forma como o gradiente é calculado e como os parâmetros são atualizados.

Essa variação funciona atualizando os parâmetros do modelo usando pequenas amostras (lotes ou *batch*) do dataset a cada iteração. Ele calcula o gradiente da função para esse *batch* e ajusta os parâmetros, multiplicado pela taxa de aprendizado. Esse processo é repetido várias vezes, permitindo que o modelo aprenda de forma incremental.

As vantagens do SGD incluem eficiência computacional e facilidade em escapar de mínimos locais de funções complexas. No entanto, o SGD também apresenta desafios, como alta variância nas atualizações devido ao uso de amostras, o que pode levar a oscilações durante o treinamento. A escolha adequada da taxa de aprendizado é crucial para garantir convergência sem tornar o processo excessivamente lento ou instável.

7.3 Momentum

Iniciando a aula com uma analogia à física clássica, o Momentum é enunciado inicialmente como a quantidade de movimento linear, que é facilmente percebido ao mover um objeto sobre uma superfície de atrito baixo ou desprezível, nessa situação o movimento permanece sem que haja necessidade de exercer força novamente a cada interação. Dentro do contexto de *Machine Learning*, o conceito de Momentum é menos prático, sendo definido como uma técnica matemática para otimizar o cálculo do gradiente descendente.

De forma semelhante ao conceito da física clássica, a técnica de Momentum matematicamente adiciona um termo inercial ao cálculo do gradiente, criando uma certa “memória” do histórico de valores. Esse comportamento permite ao algoritmo manter a direção do gradiente mais consistente mesmo em locais da função com certa variabilidade.

É adicionado ao cálculo do gradiente descendente um vetor velocidade, que irá receber os valores anteriores para cada um dos parâmetros, esse vetor agirá como a “memória” do sistema. Para cada iteração, um novo gradiente é calculado e seus valores atualizados através do produto desse gradiente com uma fração da velocidade anterior (termo inercial).

$$V_t = \beta \cdot V_{t-1} - \eta \cdot g_t$$

Onde:

V_t - vetor “velocidade” atualizado

V_{t-1} - vetor “velocidade” com valores da última iteração

β - coeficiente de momentum (adiciona inércia ao vetor)

η - taxa de aprendizado

g_t - vetor gradiente

7.4 Taxas de Aprendizado Variáveis e Adaptativas

A taxa de aprendizado é um dos hiperparâmetros mais importantes no treinamento de modelos de *machine learning*, especialmente em algoritmos baseados em gradiente descendente. Quando fixas, podem levar a problemas de convergência e instabilidade.

Para resolver essa demanda, foram desenvolvidas estratégias de taxas variáveis e adaptativas, que ajustam automaticamente o tamanho do passo durante o treinamento de acordo com regras pré-definidas. O objetivo é equilibrar velocidade e estabilidade. Alguns métodos são:

- *Step decay*: Reduz a taxa em fatores fixos após um número específico de épocas. Por exemplo, a cada 100 épocas a taxa de aprendizado é dividida por 2.
- *Exponencial decay*: Reduz a taxa de aprendizado exponencialmente a cada época, segue a lei de formação a seguir.

$$\eta(t) = A * e^{-K.t}$$

- *Inverse decay*: Reduz a taxa de acordo com o inverso de t .

$$\eta(t) = \frac{A}{(K.t+1)}$$

Nesta etapa do curso é abordado o conceito de *babysitting*, que é uma técnica interativa e manual, útil para ganhar compreender melhor os problemas no treinamento e fazer ajustes rápidos. Embora não seja escalável, é valioso em estágios iniciais de projetos, quando se deseja entender como os dados se comportam.

Técnicas de taxa de aprendizado adaptativas são algoritmos que ajustam a taxa automaticamente para cada parâmetro, baseando-se no comportamento dos gradientes durante o treinamento. Alguns dos principais algoritmos são apresentados, como o AdaGrad, que ajusta a taxa para cada parâmetro independentemente com base no histórico de gradientes do próprio treinamento. Apesar de suas limitações, inspirou técnicas mais avançadas.

$$\theta = \theta - \eta \frac{\Delta J}{\sqrt{cache + \epsilon}}$$

Outro algoritmo abordado foi o RMSProp (*Root Mean Square Propagation*), é uma técnica de otimização adaptativa que corrige os principais problemas do AdaGrad, especialmente o decaimento excessivo da taxa de aprendizado.

O RMSProp introduz uma taxa de decaimento para calcular uma média móvel exponencial dos gradientes ao quadrado, evitando que a taxa de aprendizado diminua drasticamente. Isso o torna mais robusto em problemas não esparsos e em paisagens de erro irregulares. O RMSProp é especialmente útil em redes neurais profundas. Embora tenha sido parcialmente substituído pelo Adam em muitas aplicações, ainda é uma opção para alguns problemas específicos.

7.5 Adam

Por fim, o curso traz o otimizador mais famoso, o Adam (*Adaptive Moment Estimation*), criado por Jimmy Ba, que na época era estudante de pós-doutorado na Universidade de Toronto. Esse algoritmo é tido por vezes como a evolução do RMSProp.

Para compreender bem o Adam é preciso estar bem ciente dos conceitos de momentum e taxas de aprendizados adaptativas, como o RMSProp. A problemática exposta traz uma situação em que é necessário calcular a média atual em tempo de processamento, sem ter que refazer os cálculos todos novamente, incrementando apenas a parte que falta.

Matematicamente, a essência do otimizados Adam é a equação

$$\theta_t = \theta_{t-1} - \eta \frac{m_t}{\sqrt{V_t + \epsilon}}$$

Onde:

- $V_t = \beta_2 V_t + (1 - \beta_2)g_t^2$
- $m_t = \beta_1 m_t + (1 - \beta_1)g_t$

Uma das características do Adam é ser não estacionário, ou seja, dados mais recentes recebem mais importância do que dados antigos, isso é feito através da média móvel ponderada exponencial, do inglês *Exponentially Weighted Moving Average* (EWMA) é uma técnica usada para suavizar séries temporais, atribuindo pesos decrescentes exponencialmente a observações passadas.

Dada uma série temporal $x(t)$ a EWMA $y(t)$ é calculada como:

$$y(t) = \beta y(t - 1) + (1 - \beta)x(t)$$

Apesar de o Adam ser a escolha padrão para a maioria dos problemas, a verdade é que não existe um otimizador universalmente superior, o desempenho varia conforme a necessidade do problema, a arquitetura do modelo e até a distribuição dos dados. A área do *Machine Learning* é fundamentada em experimentação, a teoria é importante na pesquisa, no desenvolvimento e na implementação, mas a prática continua sendo a maior professora. Por isso, a única maneira de saber exatamente qual otimizador funcionará melhor para cada caso é fazendo testes práticos exaustivamente, executando o código e analisando os resultados.

Conclusão

O estudo teórico e a compreensão de conceitos matemáticos são fundamentais para o verdadeiro domínio dos algoritmos de *deep learning*, pois fornecem as bases necessárias para interpretar seu comportamento. No entanto, a teoria por si só não é suficiente, a experimentação é insubstituível. Ajustar hiperparâmetros, testar diferentes otimizadores e validar dados por diferentes métodos são etapas cruciais para transformar conhecimento teórico em expertise. Portanto, equilibrar fundamentação teórica (matemática/lógica) e prática experimental é essencial para desenvolver modelos eficientes.

Referências

[1] card16 - Deep Learning: Convolutional Neural Networks in Python (seções 5 - 11)