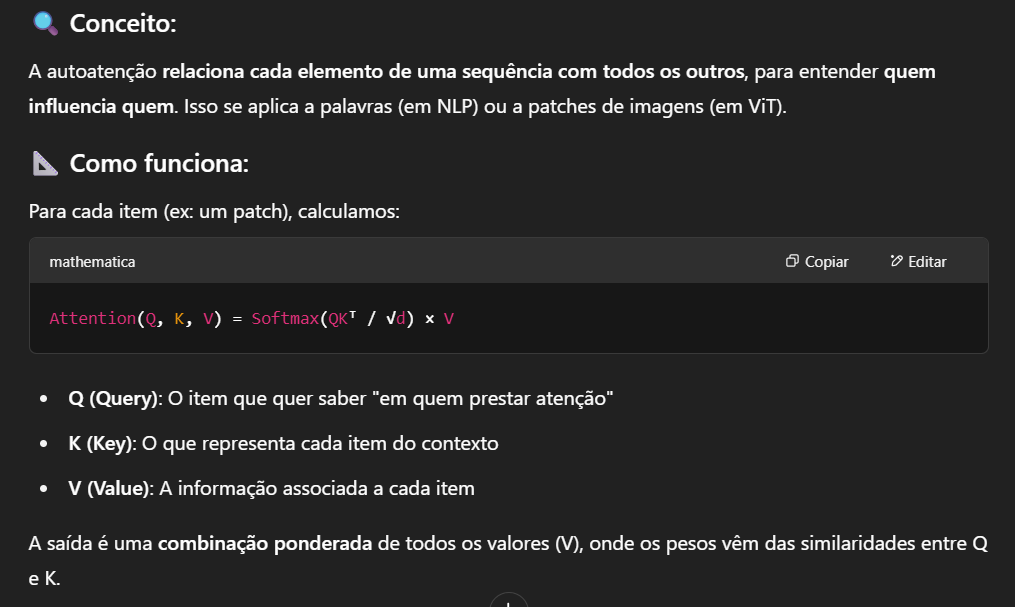
Roteiro de estudos – Vision Transformers Architecture

1. VISION TRANSFORMER (INTRODUÇÃO)

O Vision Transformer (ViT) é um modelo de deep learning baseado na arquitetura de *Transformers*, originalmente criada para NLP (como o BERT). A inovação do ViT foi aplicar esse conceito à visão computacional, tratando uma imagem como uma **sequência de patches**, como se fosse uma frase com palavras.

Essa abordagem substitui convoluções por **autoatenção**, o que permite que o modelo capture **relações globais** entre partes da imagem — algo mais difícil com CNNs tradicionais.

* Convolução – A **convolução** em uma CNN é uma operação matemática entre um filtro (ou kernel) e uma pequena região da imagem (janela local). Essa operação é usada para **extrair padrões locais**, como bordas, texturas, curvas, etc.
* Autoatenção – A autoatenção **relaciona cada elemento de uma sequência com todos os outros**, para entender **quem influencia quem**. Isso se aplica a palavras (em NLP) ou a patches de imagens (em ViT).



1. TRANSFORMER E SUA ORIGEM

O Transformer é uma arquitetura de deep learning introduzida por Vaswani et al. (2017) no artigo “Attention is All You Need”, que revolucionou o campo de NLP ao substituir redes recorrentes (RNN/LSTM) por atenção. O principal problema na NLP era traduzir, classificar e responder questionamento com base em frases. Antes, usava-se RNNs, mas elas processam palavra por palavra de forma sequencial (lento e difícil de aprender relações de longo prazo).  
Transformers processam **toda a sequência de uma vez**, e usam **atenção** para aprender o contexto de cada palavra em relação às outras. No caso das ViT’s a ideia central é tratar a imagem como uma frase, onde cada trecho da imagem (patch) seria tratado como uma palavra na frase

Componentes principais da estrutura de um transformer em ViT’s:

1. **Tokenização**: Nesta etapa a frase é dividida em tokens (ex: ["O", "cachorro", ...]). No caso das ViT’s, as imagens serão divididas em diferentes patchs (trechos da imagem), ou seja, considerando uma imagem colorida 224x224 pixels com 3 canais (RGB) temos:

Divisão horizontal -> 224/16 = 14

Divisão vertical -> 224/16 = 14

Patches -> 14x14 = 196 (trechos de imagens)

Valores = [16 pixels \* 16 pixels \* 3 canais] = 768 valores (dimensões)

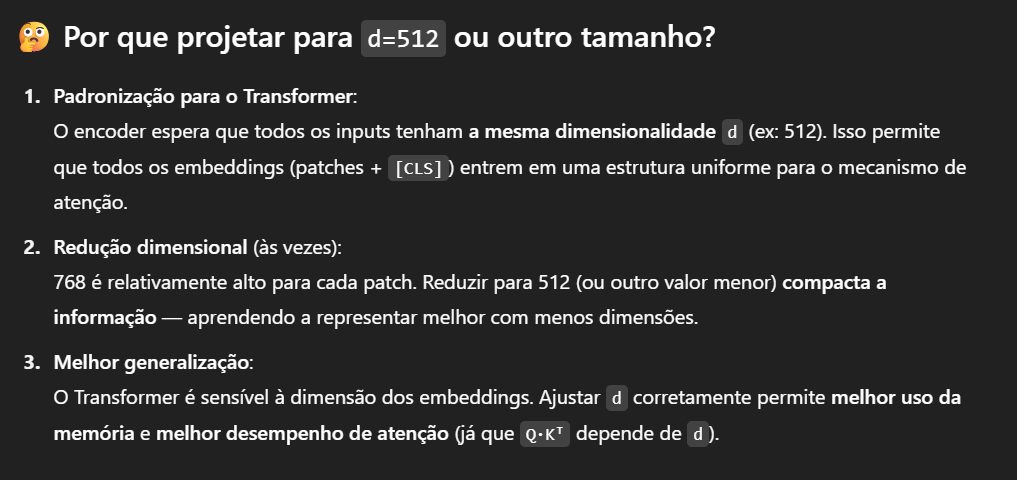
Após este processo, é como se tivéssemos uma frase contendo 196 palavras, cada uma contendo 768 dimensões

1. **Embedding**: Cada token é transformado em um vetor numérico (ex: 512 dimensões)

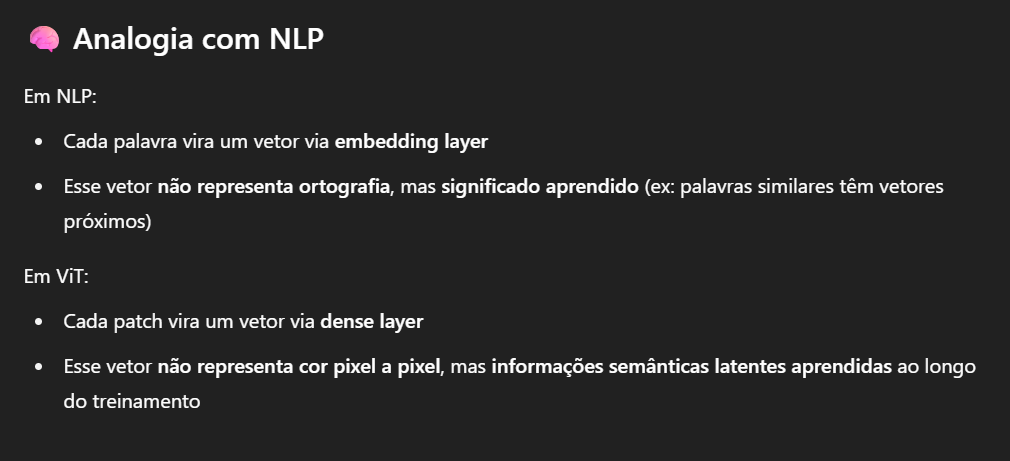
Esses 768 valores **são apenas dados brutos** (como os bytes de uma imagem). Eles **ainda não carregam semântica aprendida** — ou seja, não têm estrutura vetorial adequada para serem processados em um transformer (que espera vetores embedding com significado).

Nesta etapa, os dados dos patches são processados por uma camada densa de machine learning, a fim de **aprender uma nova representação vetorial de tamanho fixo** (definida por um valor **d**), com propriedades úteis para atenção e classificação.

**Observação**:

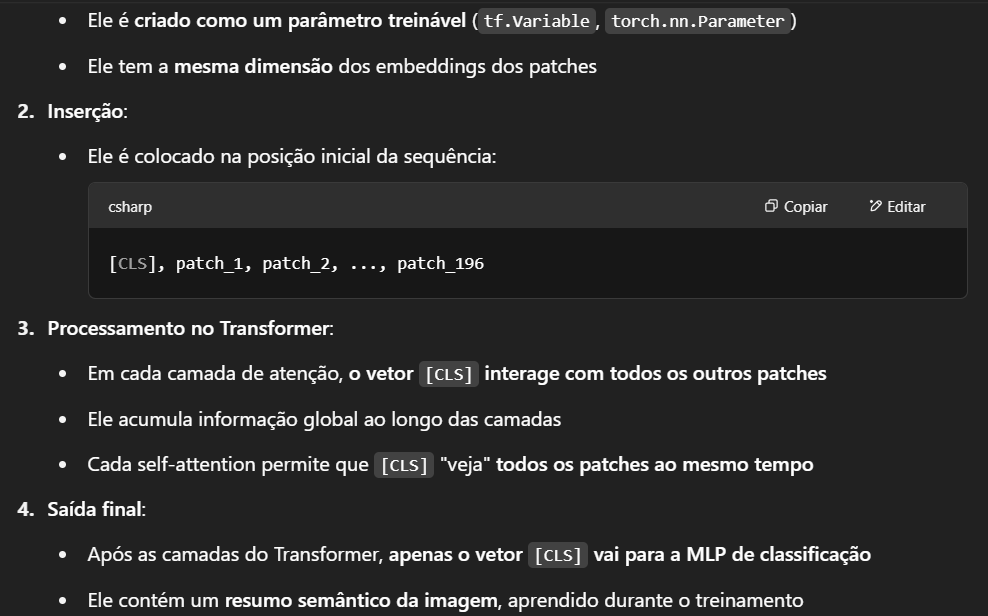


**Observação 2:**



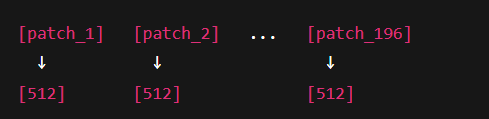
1. **Classification Token:** Inicialmente um vetor denominado ‘classification token’ [CLS] é inserido no início da sequência de embeddings. Este vetor **não representa um patch da imagem**, mas sim uma posição especial. Seu objetivo é **aprender a representar toda a imagem**, pois é a **única posição usada para prever a classe final.**

Importante salientar que o [CLS] **NÃO** resume a imagem no início, ele começa como um vetor aprendível aleatório, igual a qualquer outro vetor de embedding. Ele não contém informação da imagem no início. Mas ao passar pelas camadas do Transformer, ele vai "ouvir" todos os patches via autoatenção — e ao final, ele terá aprendido a carregar as informações globais mais relevantes da imagem.

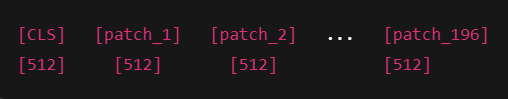


Representação Prática:

* Após a etapa de patching e linear embedding, temos uma sequência de N vetores (ex: 196 patches, cada vetor tem dimensão d = 512)



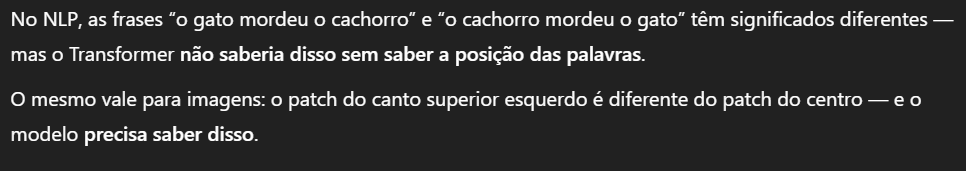
* Adicionamos um vetor [CLS], com a mesma dimensão d = 512, na **posição zero**:



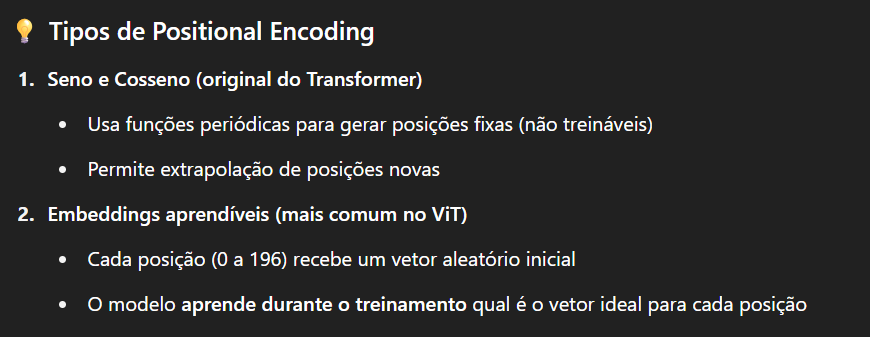
***Analogia didática:*** *Imagine que cada patch é uma frase de um parágrafo, e você precisa entender o contexto completo. O [CLS] é como uma caixa de resumo no topo do parágrafo, onde você vai anotar tudo de importante que ler.*

*O [CLS] funcional porque o* ***self-attention permite que ele veja todos os patches****. E como esse vetor é* ***otimizado com o mesmo objetivo da tarefa (classificar corretamente)****, ele* ***aprende a se moldar*** *ao conteúdo da imagem inteira. Em outras palavras: O [CLS] é como um "ouvinte central" que presta atenção em todos os patches. Ao final do processo, ele "entende" o suficiente para tomar uma decisão sobre a classe da imagem.*

1. **Positional** **Encoding**: Como o modelo vê tudo de uma vez, ele precisa saber a ordem das palavras. Eles processam a entrada como uma **sequência de vetores** e tratam todos da mesma forma, diferentemente das CNNs (que exploram a posição dos pixels diretamente). **O ViT precisa que a ordem dos patches seja aprendida.**

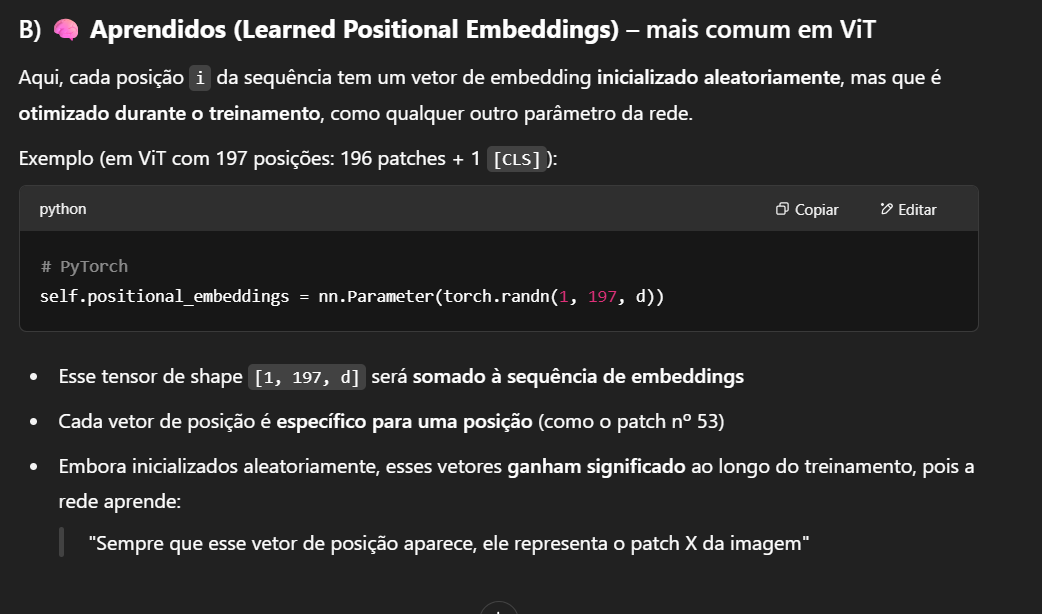
****

Funcionamento do Positional Encoding se dá por meio de uma soma entre os vetores definidos pela etapa de embedding e um vetor de posição específico, criando uma **sequência enriquecida** com noção de "onde está cada patch" na imagem.



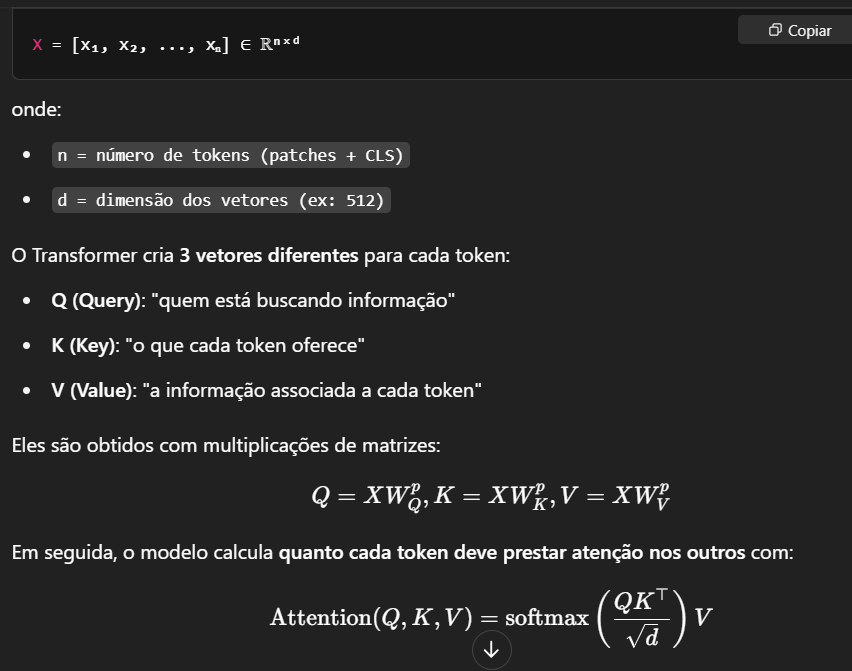
* É importante salientar que o vetor de posição é somado aos embeddings uma vez que esta operação mantém a **mesma dimensão d,** sendo necessário para que o encoder possa processar a sequência com shape fixo. Além disso, a soma é simples, eficiente e empiricamente eficaz.

Funcionamento do positional encoding para o caso das ViT’s:



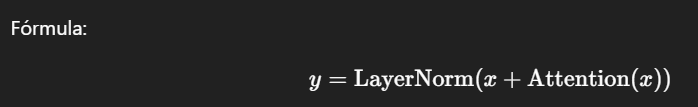
1. **Transform encoder:** Esse bloco é o responsável por aprender relações entre patches da imagem, tratando-os como tokens (palavras). Esse bloco é repetido **várias vezes** para refinar os vetores progressivamente e pode ser subdivido em 3 etapas principais:
2. Self-Attention (Auto-atenção) – Permitir que cada patch “preste atenção” nos outros patches da imagem, decidindo quais são importantes para compor sua própria representação.

Dado uma sequência de embeddings, a operação desta sub-etapa se dá por:



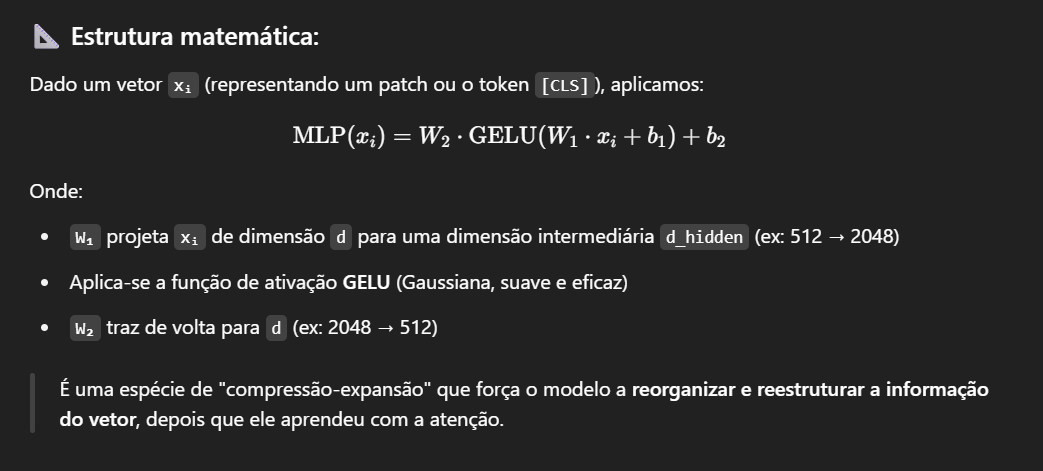
De modo mais didático, podemos dizer que para cada vetor da sequência, ele cria um **mapa de atenção**, que é uma **distribuição de pesos** (soma = 1) dizendo **em quem ele deve prestar atenção**. Isso permite que, por exemplo, o patch do canto superior "olhe" para o do centro se for relevante.

1. Add & Norm (Soma com conexão residual + normalização): Uma vez que as redes densas sofrem com **vanishing gradients** e instabilidade, a saída da atenção é **somada com a entrada original (residual connection)** e depois normalizada com **Layer Normalization**, para manter a estabilidade.



Essa soma ajuda o modelo a **preservar a informação original**, enquanto ajusta com o que aprendeu com a atenção.

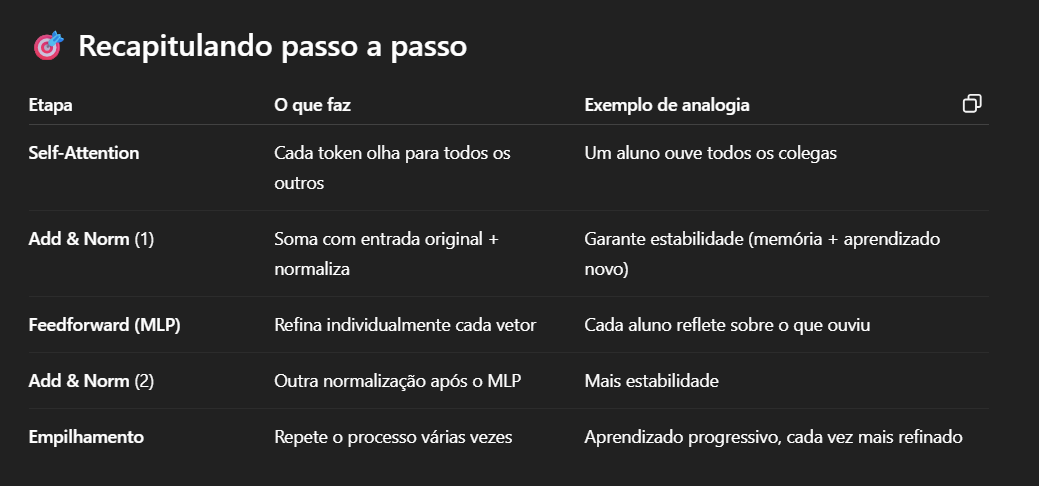
1. Feedforward (MLP): Um mini MLP (duas camadas densas) que processa cada vetor (vetor de contexto gerado após a etapa de atenção) de forma independente (posição por posição), aumentando sua capacidade de representação. Com isso, espera-se ajustar e refinar individualmente o conteúdo de cada vetor, depois que ele já está "contaminado" com informações globais via atenção.



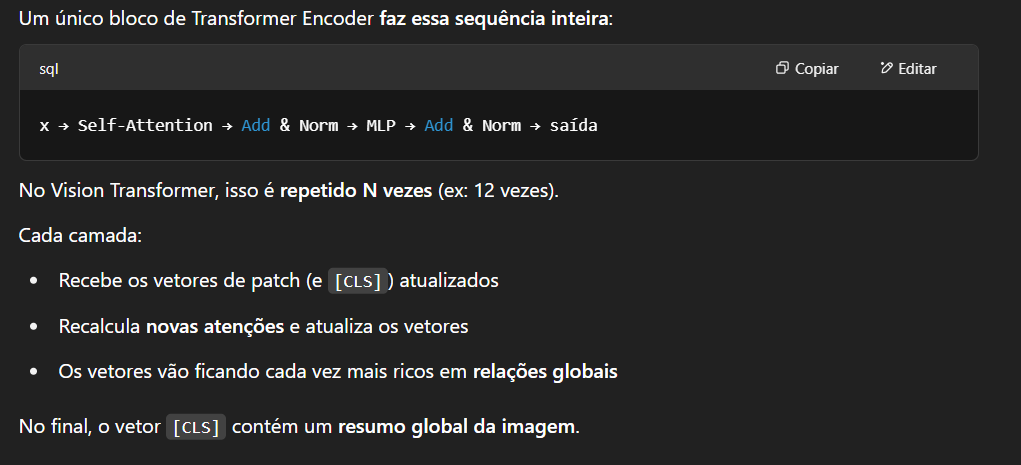
Na prática essa etapa se torna responsável por:

1. **Reorganiza o vetor:** A atenção "mistura" todos os tokens entre si. O MLP agora reorganiza internamente os valores do vetor individualmente.
2. **Aumenta a capacidade expressiva**: Transformações não-lineares (via GELU) permitem que o vetor represente conceitos mais complexos — como padrões texturais, alinhamentos de bordas, ou sintomas visuais em folhas
3. **Remove redundância e reforça padrões**: Como há um aumento seguido de redução na dimensão (ex: 512 → 2048 → 512), o MLP age como um filtro ou refinador: descarta coisas inúteis e reforça padrões importantes.

Resumo das operações do bloco ‘transform encoder’



1. **Empilhamento e Camada final:** O bloco de transform encoder pode ser repetido múltiplas vezes a fim de refinar os resultados, e ao final de tais execuções, é retornado um vetor especial, sendo geralmente o [CLS] para gerar a saída (ex: classe da frase). A imagem a seguir define o passo-a-passo



1. Implementação prática da arquitetura ViT (FEITA NO PYCHARM)
2. Considerações e Conceitos relevantes:
3. Cabeça de atenção (Multi-head attention)

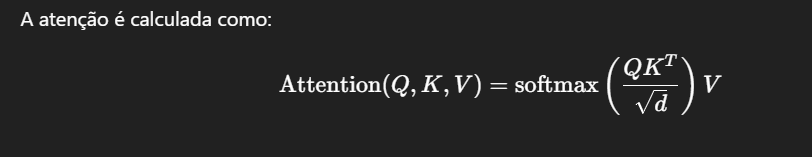
No Transformer, uma "cabeça de atenção" é uma instância independente do mecanismo de self-attention. Em multi-head attention, você tem várias cabeças trabalhando em paralelo, cada uma aprendendo a focar em partes diferentes da entrada.

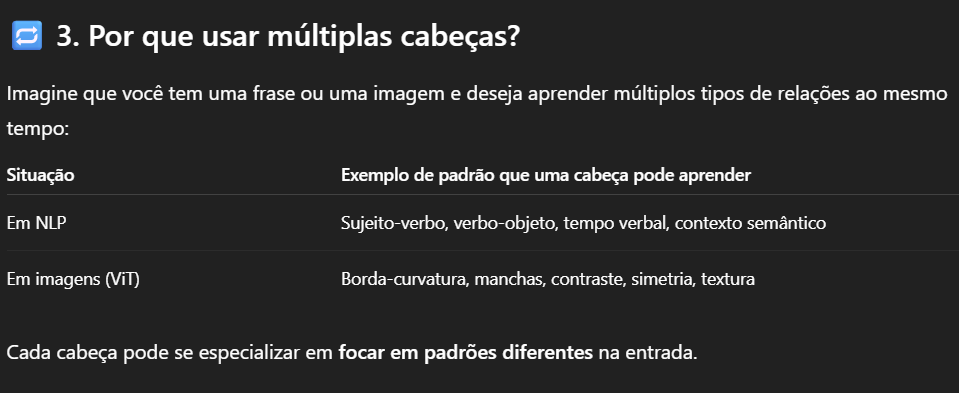
Uma cabeça de atenção é uma versão isolada do mecanismo de self-attention, contendo seus próprios pesos e projeções Q, K, V e tem como objetivo aprender a focar em certos padrões da sequência.

O mecanismo de atenção própria (self-attention) permite que cada elemento de uma sequência (ex: patches da imagem ou palavras de uma frase) “olhe” para os outros elementos e combine informações relevantes.

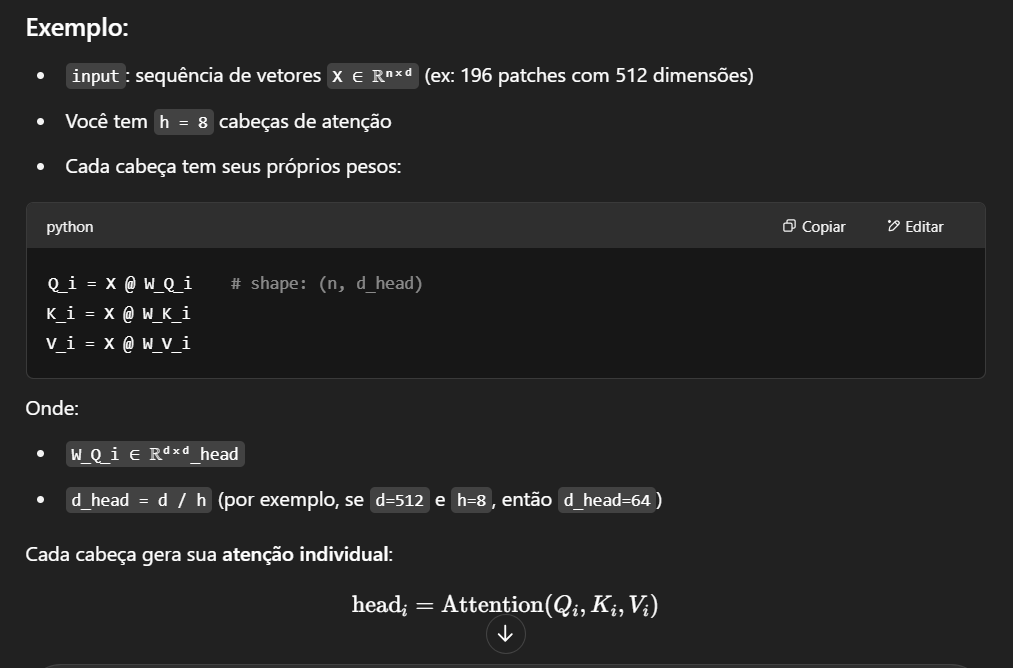
Cada vetor da sequência é transformado em três vetores:

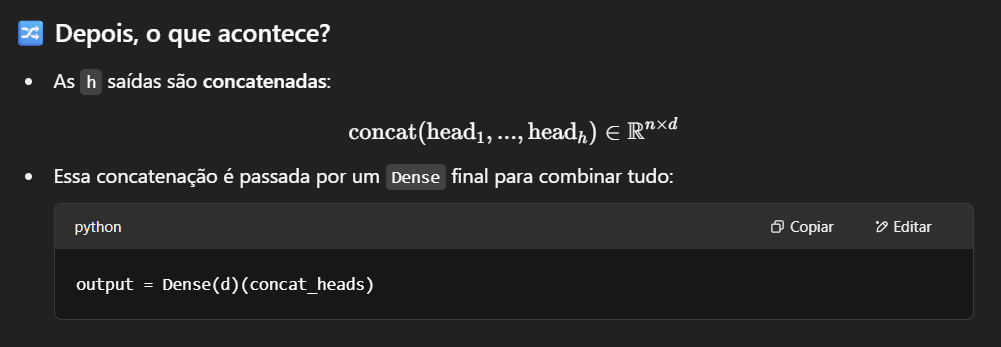
* Query (Q): o que estou procurando?
* Key (K): o que cada item representa?
* Value (V): qual informação será usada?





Exemplo matemático:





SLN

1. Conceitos básicos para recordar:

Processo de compilação de uma rede neural. Buscar entender otimizadores, funções de custo, métricas de avaliação. Preciso compreender melhor conceitos sobre acurácia, perda, falsos negativos e positivos, matriz de confusão, funções de custo, etc. Eu consigo entender como a rede é estruturada mas tenho muita dificuldade de entender a forma de operação da rede bem como seus hiperparâmetros e métodos de avaliação.