[PSI5790 aula 9 parte 1. Início]

Vision Transformers (ViTs) Aplicados ao Dataset MNIST

[Esta apostila foi escrita com auxílio de IA]

1. Introdução e Motivação

Historicamente, as Redes Neurais Convolucionais (CNNs) dominaram o campo da Visão Computacional devido ao seu viés indutivo de invariância por translação e localidade.

Nota: Viés indutivo é a estratégia que o algoritmo adota quando aparece uma amostra de teste que não foi visto entre as amostras de treino.
Nota: Invariância por translação é a propriedade de reconhecer um objeto independentemente da sua localização na imagem. Talvez seria mais correto falar de "equivariância".
Nota: Localidade é a suposição que um algoritmo faz de que pixels próximos estão mais relacionados entre si do que com pixels distantes.

No entanto, em 2020, o artigo "An Image is Worth 16×16 Words" introduziu o Vision Transformer (ViT), provando que a arquitetura Transformer — originalmente concebida para Processamento de Linguagem Natural — poderia ser aplicada diretamente a imagens com o mínimo de modificações.

Ao estudarmos o ViT no dataset MNIST, não estamos buscando o estado da arte em acurácia (já que CNNs simples resolvem o MNIST com quase 100% de precisão), mas sim de compreender o mecanismo de uma arquitetura que carece de vieses indutivos locais e precisa aprender a geometria do espaço do zero. O debate acadêmico entre CNNs e ViTs reside na escalabilidade e na dependência do volume de dados.

Nota: Escalabilidade é a propriedade do algoritmo gerar um modelo melhor quando se aumenta a quantidade de dados de treino ou o número de parâmetros do modelo.
Nota: Dependência de volume de dados diz respeito a quantidade de dados de treino necessária para o algoritmo gerar um modelo que atinja um certo desempenho.

Em datasets pequenos (como o MNIST ou o CIFAR-10 com dezenas de milhares de imagens), as CNNs superam consistentemente os ViTs devido aos seus fortes vieses indutivos de localidade e invariância por translação, que funcionam como uma excelente regularização natural. Em contrapartida, os ViTs tendem severamente ao overfitting por precisarem aprender a própria geometria espacial do zero.

Nota: Regularização é a técnica usada para evitar overfitting.

Em datasets médios (como o ImageNet-1k, com 1000 categorias e aproximadamente 1,3 milhões de imagens de treino), a disputa se equilibra: os ViTs conseguem atingir ou superar as CNNs, mas frequentemente exigem arquiteturas híbridas ou técnicas agressivas de data augmentation e regularização para convergir adequadamente.

Por fim, em datasets massivos (como o ImageNet-21k com 21000 categorias e aproximadamente 14 milhões de imagens de treino), a ausência de amarras estruturais torna-se a maior força do ViT; ele supera o teto de desempenho limitante das CNNs tradicionais ao mapear relações globais e contextos complexos que extrapolam o campo receptivo local e rígido das operações convolucionais.

Assim, em praticamente todas as aplicações "práticas" (imagens e sinais de um certo domínio específico com até dezenas de milhares de dados de treino) CNN supera ViT. Porém, existem alguns problemas específicos onde uma abordagem híbrida ViT/CNN consegue superar CNN puro. São problemas onde é necessário associar atributos distantes da imagem (como detectar assimetria global ou distorção arquitetural em mamografias).

Além disso, fazendo transfer learning com modelos pré-treinados em datasets massivos, ViT pode gerar bons resultados mesmo usando poucas imagens de treino.

ViT compara cada patch da imagem com todas as outras (enquanto que CNN procura achar padrões locais). Esta operação poderia ser lenta fazendo processamento sequencial, mas é executada de forma massivamente paralela em GPU ou TPU, usando a operação "multiplicação matricial em lote" que já é muito usada em outras operações de redes neurais (tanto MLP como CNN).

Considere o problema que consiste em dizer se numa imagem há duas formas geométricas iguais ou diferentes. Por exemplo, se numa imagem houver dois quadrados o modelo deve responder "iguais"; se houver um círculo e um triângulo o modelo deve responder "diferentes". Provavelmente, ViT teria mais facilidade para resolver este problema do que CNN, pois é necessário olhar para regiões distantes da imagem.

Exercício: Crie um dataset de imagens 96×96 com fundo ruidoso e exatamente duas formas geométricas (triângulo, quadrado ou círculo), com rotação e escala aleatórias, em regiões aleatórias da imagem. Depois, crie um modelo CNN que classifica se as duas formas presentes na imagem são de tipos iguais ou diferentes, treinando a rede a partir do zero. O mesmo usando ViT.

2. Implementação de ViT para classificar MNIST

O programa tr02c.py abaixo implementa ViT para classificar MNIST, sem fazer data augmentation.

```
# tr01.py -> tr02.py -> tr02c.py (imprime curvas e roda 60 epocas)
import os

os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt

# Configurações e Hiperparâmetros
num_classes = 10
input_shape = (28, 28, 1)
learning_rate = 0.001
weight_decay = 0.0001
batch_size = 256
num_epochs = 60
image_size = 28  # Redimensionar se necessário, mas 28 é nativo do MNIST
patch_size = 7   # Tamanho de cada patch (7x7)
num_patches = (image_size // patch_size) ** 2
projection_dim = 64
num_heads = 4
transformer_units = [
    projection_dim * 2,
    projection_dim,
]  # Tamanho das camadas MLP no Transformer
transformer_layers = 4
mlp_head_units = [2048, 1024]  # Tamanho das camadas densas do classificador final

# Carregar MNIST
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

x_train = x_train.reshape((-1, 28, 28, 1)).astype("float32") / 255.0
x_test = x_test.reshape((-1, 28, 28, 1)).astype("float32") / 255.0

# Camada para extrair patches da imagem
class Patches(layers.Layer):
    def __init__(self, patch_size):
        super().__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID",
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])
        return patches

# Camada para projetar os patches e adicionar embeddings posicionais
class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super().__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )

    def call(self, patch):
```

```python
        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(patch) + self.position_embedding(positions)
        return encoded

def impHistoria(history):
  print(history.history.keys())
  plt.plot(history.history['accuracy'])
  plt.plot(history.history['val_accuracy'])
  plt.title('model accuracy'); plt.ylabel('accuracy'); plt.xlabel('epoch')
  plt.legend(['train', 'test'], loc='upper left')
  plt.savefig("tr02c_acc.png")
  plt.close()
  # plt.show()
  plt.plot(history.history['loss'])
  plt.plot(history.history['val_loss'])
  plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
  plt.legend(['train', 'test'], loc='upper left')
  plt.savefig("tr02c_loss.png")
  plt.close()
  # plt.show()

def create_vit_classifier():
    inputs = layers.Input(shape=input_shape)

    # Criar patches
    patches = Patches(patch_size)(inputs)
    # Codificar patches
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    # Criar múltiplas camadas do bloco Transformer
    for _ in range(transformer_layers):
        # Layer normalization 1
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        # Multi-head attention
        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)
        # Skip connection 1
        x2 = layers.Add()([attention_output, encoded_patches])
        # Layer normalization 2
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)'
        # MLP
        x3 = layers.Dense(transformer_units[0], activation=tf.nn.gelu)(x3)
        x3 = layers.Dropout(0.1)(x3)
        x3 = layers.Dense(transformer_units[1], activation=tf.nn.gelu)(x3)
        x3 = layers.Dropout(0.1)(x3)
        # Skip connection 2
        encoded_patches = layers.Add()([x3, x2])

    # Criar um [batch_size, projection_dim] tensor
    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)

    # Camadas densas finais para classificação
    for units in mlp_head_units:
        representation = layers.Dense(units, activation=tf.nn.gelu)(representation)
        representation = layers.Dropout(0.5)(representation)

    logits = layers.Dense(num_classes)(representation)

    model = keras.Model(inputs=inputs, outputs=logits)
    return model

# Compilar e treinar
model = create_vit_classifier()

# Imprimir a estrutura da rede
from tensorflow.keras.utils import plot_model
plot_model(model, to_file='tr02c.png', show_shapes=True)
model.summary()

optimizer = tf.keras.optimizers.AdamW(
    learning_rate=learning_rate, weight_decay=weight_decay
)

model.compile(
    optimizer=optimizer,
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[
        keras.metrics.SparseCategoricalAccuracy(name="accuracy"),
    ],
)

print("Iniciando treinamento do Vision Transformer no MNIST...")
history=model.fit(
    x=x_train, y=y_train, batch_size=batch_size, epochs=num_epochs,
    validation_split=0.1,
)
impHistoria(history)

# Avaliação final
_, accuracy = model.evaluate(x_test, y_test)
print(f"Test accuracy: {round(accuracy * 100, 2)}%")
```

Programa P: Classifica MNIST usando ViT. [~/deep/keras/transformer/tr02c.py]

Train accuracy: 99.32%
Val accuracy: 99.02%
Test accuracy: 99.10%.

Nota: Tínhamos conseguido acuracidade de teste de 99.52% usando CNN sem data augmentation. Portanto, ViT “puro” tem acuracidade um pouco menor que CNN para classificar MNIST.

Diagrama de blocos simplificado do programa P:

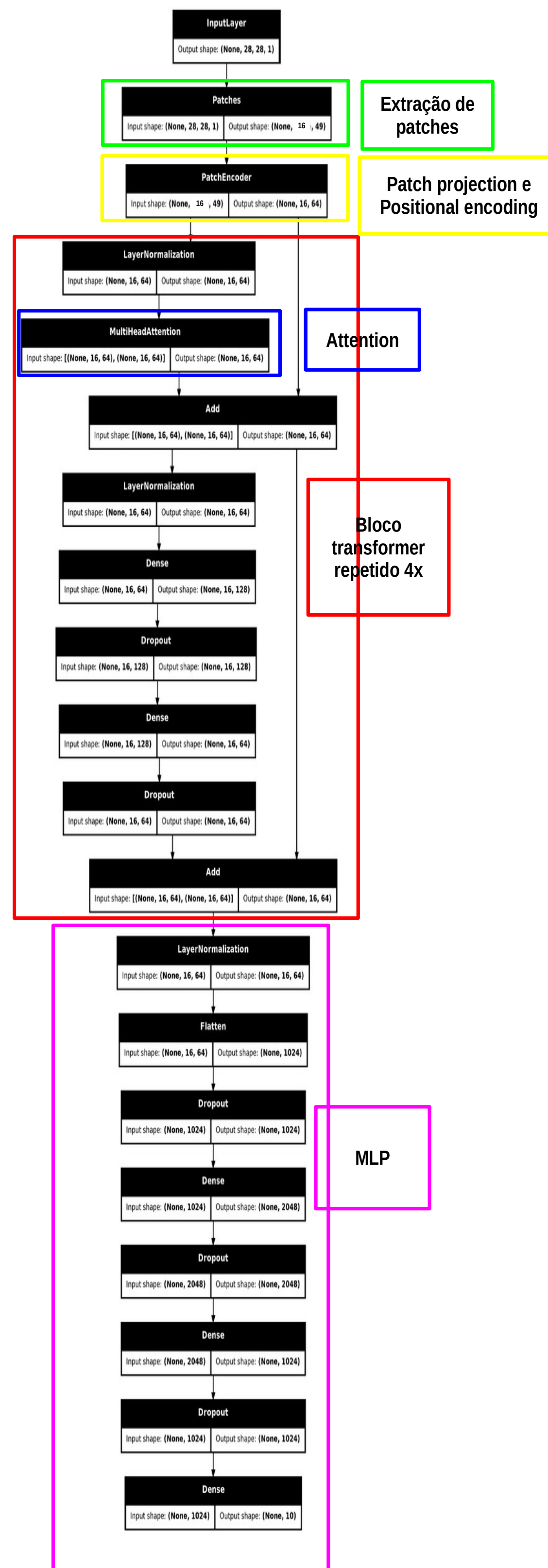

Figura F: Para facilitar a visualização, aqui é mostrado um único bloco transformer, marcado em vermelho. No programa P, esse bloco é repetido 4 vezes em sequência.

3. Visão Geral da Arquitetura ViT

Transformer foi originalmente criada para processar linguagem natural, que consiste em uma sequência de tokens (cada palavra é convertida em número chamado de token). O desafio fundamental para aplicar Transformer a imagens é que um Transformer espera uma sequência de vetores (tokens), enquanto que imagens são tensores tridimensionais (altura, largura, canais). O ViT resolve isso quebrando a imagem em uma grade de pequenos retalhos (patches, figura G).

3.1. Extração de Patches e Projeção Linear

Abaixo, detalhamos o fluxo de dados considerando uma imagem do MNIST (28×28×1), dividido em N=4×4=16 patches com P×P=7×7 pixels cada (figura G, trecho verde do programa P e da figura F).

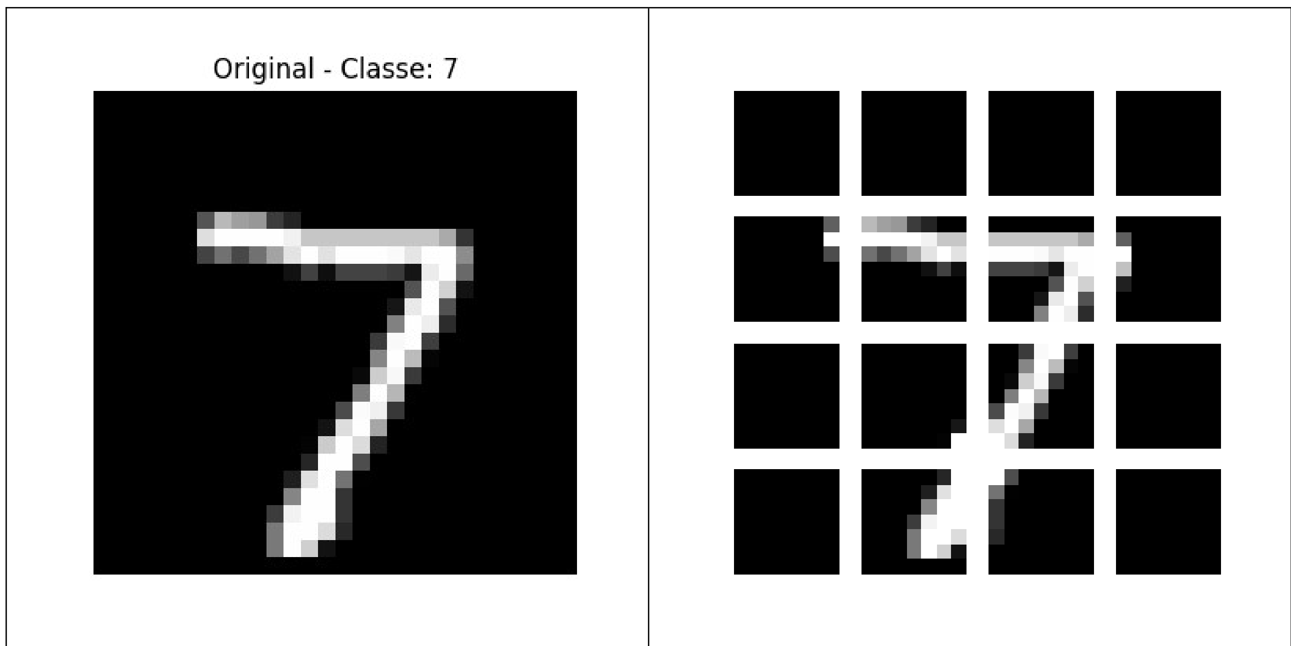

Figura G: Divisão de uma imagem de MNIST em 4×4 patches com 7×7 pixels cada.

Cada patch é convertido em um vetor com P×P=49 elementos que são projetados num token de dimensão D (D=64 no nosso caso) ao passarem por uma camada densa. Normalmente, o vetor é "achatado" (D<P×P×1), mas no nosso exemplo o vetor é "esticado" (D>P×P×1) (trecho amarelo do programa P e figura F). Em processamento de linguagem natural, esta etapa converte palavras em tokens.

3.2. Codificação Posicional (Positional Embedding)

Como a operação de atenção não conhece a ordem dos patches na imagem, precisamos injetar a informação espacial. Adicionamos um vetor de parâmetros aprendíveis de dimensão D=64 a cada um dos 16 patches, representando sua posição linear (de 0 a 15, parte amarelo do programa). Note que ViT não conhece qual patch está em cima ou embaixo de qual patch.

Etapa	Dimensão de Entrada	Dimensão de Saída	Descrição
Imagem Original	(28, 28, 1)	(28, 28, 1)	Matriz bruta de pixels.
Patches	(28, 28, 1)	(16, 49)	Divisão em 16 blocos lineares com P×P=49.
Patch Projection	(16, 49)	(16, 64)	Mapeamento linear para o espaço latente.
Positional Encoding	(16, 64)	(16, 64)	Soma da informação de ordem sequencial.

a. Positional embedding

A camada *layers.Embedding* do Keras insere a informação posicional. Ela pode ser resumida em uma metáfora: é semelhante a um *lookup table*. A função dela é transformar números inteiros discretos (índices 0 a 15 dos patches) em vetores densos. A camada Embedding cria uma matriz **E (**16×64) de parâmetros treináveis. No nosso código:

```
layers.Embedding(input_dim=16, output_dim=64)
```

cria internamente uma matriz **E** de tamanho 16×64 de números aleatórios (inicialmente no intervalo [-0.05, 0.05]):

- 16 (input_dim): É o número N=4×4=16 de posições de patches.
- 64 (output_dim): É o tamanho do vetor que vai representar cada uma dessas 16 categorias (ou seja, o tamanho D do espaço de embedding).

b. Como funciona a consulta

Quando você passa o número 0 para essa camada, ela simplesmente vai até a linha 0 da sua matriz **E** e puxa o vetor de 64 números que está guardado lá. Se você passar o número 5, ela puxa a linha 5.

c. Matriz treinável

A matriz **E** é treinada pela backpropagation. No final do treino, a matriz **E** terá números geralmente flutuando em [-2.0 e +2.0].

d. Por que não usar One-Hot Encoding?

Você poderia se perguntar: *"Por que não transformar o número 5 num vetor cheio de zeros com o número 1 na quinta posição (One-Hot Encoding)?"*

- **Espaço:** O One-Hot encoding gera vetores esparsos (cheios de zeros), o que desperdiça muita memória quando o número de categorias (patches) é grande (pense num dicionário de 50.000 palavras).
- **Semântica:** No One-Hot encoding, todos os vetores estão à mesma distância uns dos outros. Na camada Embedding, como os números da matriz são treináveis, o modelo aprende a colocar categorias parecidas com vetores parecidos.

No nosso caso, o modelo aprende a colocar vetores de posições vizinhas (como o patch 0 e o patch 1) com valores numéricos próximos na tabela de embedding.

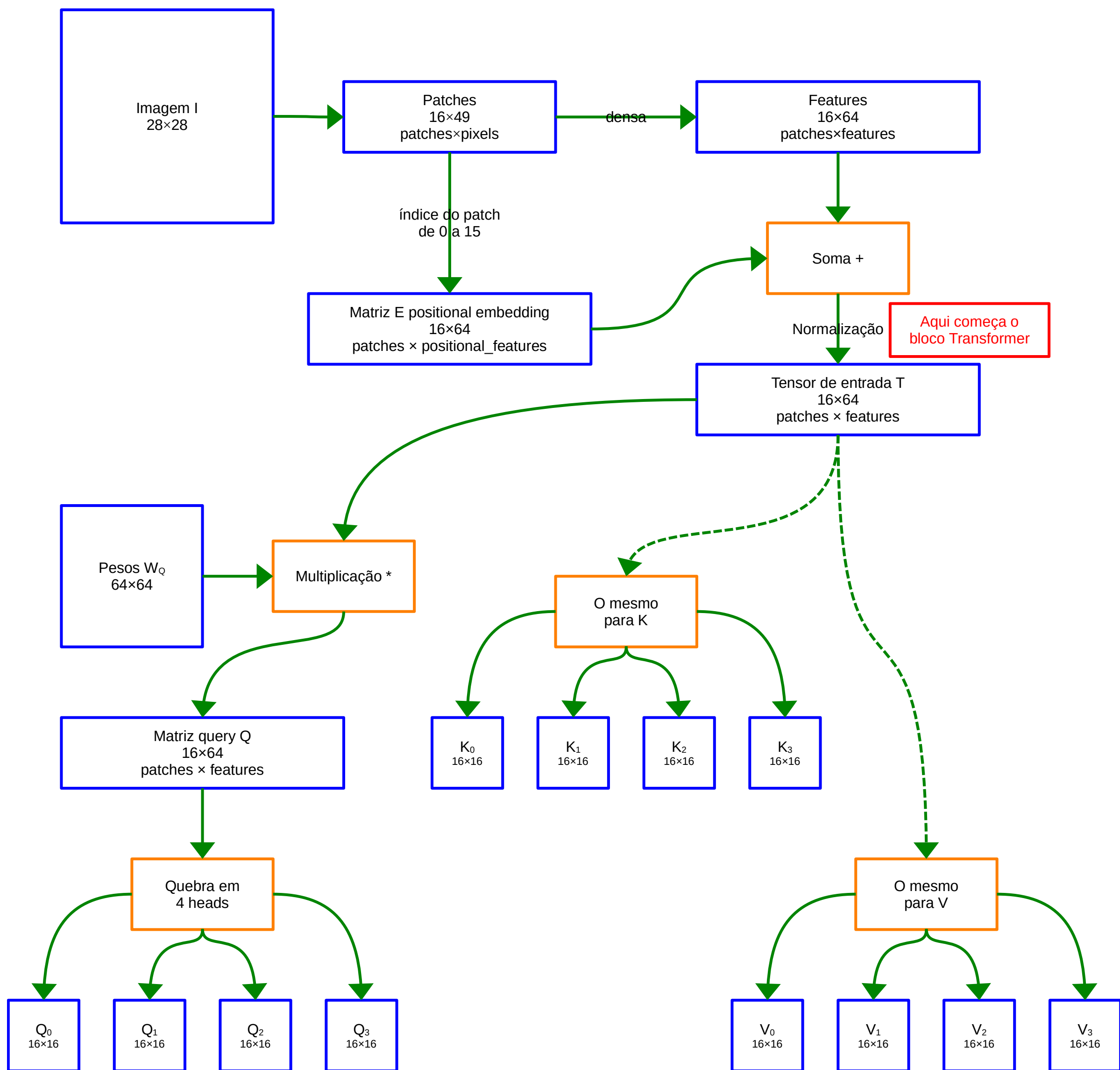

Figura G: Os passos da imagem de entrada I (28×28) até se converterem em matrizes 16×64 Q, K e V, divididas em h=4 heads.

4. O Bloco Transformer e o Mecanismo de Atenção

O núcleo do ViT é o bloco Transformer (caixa em vermelho na figura F e programa P), composto por duas partes principais aplicadas sequencialmente, sempre precedidas por uma Camada de Normalização (LayerNorm – normaliza todas as características de uma amostra por vez, sem depender de lote). No nosso programa, o bloco Transformer recebe tensores 16×64 e gera tensores na saída também de dimensão 16×64. No programa P, 4 blocos transformer são aplicados em sequência.

4.1. Multi-Head Self-Attention (MSA)

Diferente das CNNs que olham apenas para os vizinhos locais, a auto-atenção permite que cada patch interaja com todos os outros. A auto-atenção descobre (por exemplo) que, para que a imagem represente o número "7" escrito com 2 traços e não-cortado (figura H), precisa ter na imagem uma relação entre pares de patches marcados em cores. Isto é, para ser "7" (com 2 traços e não-cortado), é preciso ter reta horizontal no patch 5 casando com uma esquina no patch 6; uma esquina no patch 6 casando com reta vertical no patch 10; etc. Isto é feito comparando cada um dos patches com todos os outros, através de produto escalar. No nosso caso, temos self-attention, pois cada patch da imagem é comparado com todos os outros patches da mesma imagem (repare nos dois parâmetros iguais (x1, x1) da camada MultiHeadAttention).

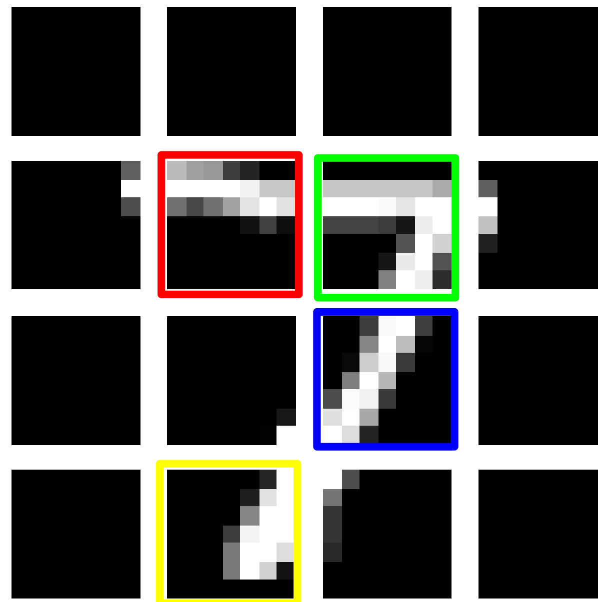

Figura H: No ViT, todos os pares de patches são comparados entre si.

Os 4 patches considerados importantes na última camada de Transformer	Cor “quente” indica as regiões da imagem importantes para a classificação.
Mapas de atenção indicam os pares de patches importantes nas 4 camadas de atenção (última matriz da figura J).	

Figura I: No ViT, todos os pares de patches são comparados entre si.

Figura I2: Diferentes formas de escrever “7”.

Dependendo da forma de escrever “7” (figura I2), as relações entre pares de patches pode mudar. Por exemplo, os pares de patches que identificam 7 cortado escrito com 4 traços podem não ser adequados para reconhecer para reconhecer “7” não cortado escrito com 2 traços. Isto pode ser resolvido usando múltiplos “heads”. Cada head irá detectar relações entre pares de patches de diferentes formas de escrever “7”.

O programa P não mostra o que acontece dentro da camada *MultiHeadAttention*. A classe *MyMultiHeadAttention* abaixo implementa explicitamente o conteúdo da classe *MultiHeadAttention*.

Nota: Usando a nova classe, atingiu-se acuracidade de teste 98.86% (um pouco menos que 99.10% do programa original) – pode ser que a implementação de MultiHeadAttention seja ligeiramente diferente de *MyMultiHeadAttention*.

```python
# Classe MyMultiHeadAttention explicitando Q, K, V
class MyMultiHeadAttention(layers.Layer):
    def __init__(self, num_heads, key_dim, dropout=0.1, **kwargs):
        super().__init__(**kwargs)
        self.num_heads = num_heads
        self.key_dim = key_dim

        # Camadas densas para projetar as matrizes Query, Key e Value
        self.q_dense = layers.Dense(num_heads * key_dim)
        self.k_dense = layers.Dense(num_heads * key_dim)
        self.v_dense = layers.Dense(num_heads * key_dim)

        self.dropout = layers.Dropout(dropout)
        self.out_dense = None

    def build(self, input_shape):
        # A saída deve ter a mesma dimensão da entrada para a skip connection
        self.out_dense = layers.Dense(input_shape[-1])
        super().build(input_shape)

    def split_heads(self, x, batch_size):
        # Redimensiona para (batch_size, seq_len, num_heads, key_dim)
        # Usamos -1 para seq_len para manter flexibilidade, mas as outras dimensões são fixas
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.key_dim))
        # Transpõe para (batch_size, num_heads, seq_len, key_dim)
        return tf.transpose(x, perm=[0, 2, 1, 3])

    def call(self, query, value, attention_mask=None, training=False):
        # Em chamadas self-attention, query e value (que faz papel de key também) são o mesmo tensor
        key = value
        batch_size = tf.shape(query)[0]
        seq_len = tf.shape(query)[1]

        # 1. Geração das matrizes Q, K, V através de projeções lineares
        Q = self.q_dense(query) # (batch, seq_len, num_heads * key_dim)
        K = self.k_dense(key)   # (batch, seq_len, num_heads * key_dim)
        V = self.v_dense(value) # (batch, seq_len, num_heads * key_dim)

        # 2. Divisão em múltiplas "cabeças" (heads)
        Q = self.split_heads(Q, batch_size) # (batch, num_heads, seq_len, key_dim)
        K = self.split_heads(K, batch_size) # (batch, num_heads, seq_len, key_dim)
        V = self.split_heads(V, batch_size) # (batch, num_heads, seq_len, key_dim)

        # 3. Scaled Dot-Product Attention
        # Cálculo de Q * K^T (Score de atenção)
        matmul_qk = tf.matmul(Q, K, transpose_b=True) # (batch, num_heads, seq_len_q, seq_len_k)

        # Escalonamento (Scaling) para evitar gradientes pequenos
        dk = tf.cast(self.key_dim, tf.float32)
        scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

        if attention_mask is not None:
            scaled_attention_logits += (attention_mask * -1e9)

        # Softmax para obter os pesos de atenção (soma 1 na última dimensão)
        attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
        attention_weights = self.dropout(attention_weights, training=training)

        # Multiplicação dos pesos pela matriz V (Value)
        output = tf.matmul(attention_weights, V) # (batch, num_heads, seq_len_q, key_dim)

        # 4. Re-concatenação das cabeças
        output = tf.transpose(output, perm=[0, 2, 1, 3]) # (batch, seq_len, num_heads, key_dim)
        concat_attention = tf.reshape(output, (batch_size, seq_len, self.num_heads * self.key_dim))

        # 5. Projeção linear final (Dense)
        output = self.out_dense(concat_attention)

        # Tenta restaurar o shape estático se conhecido
        if not tf.executing_eagerly():
            output.set_shape(query.shape)
        return output

    def compute_output_shape(self, input_shape):
        return input_shape
```

Classe C: Classe *MyMultiHeadAttention* permite entender o que acontece dentro da camada *MultiHeadAttention*.

Para cada patch, geramos três matrizes multiplicando o vetor de patches T (16×64) por matrizes de pesos aprendíveis $W_Q, W_K, W_V \in \boldsymbol{R}^{D \times D}$: Query (Q), Key (K) e Value (V). No nosso caso, D=64.

As matrizes W_Q, W_K e W_V são inicializadas aleatoriamente. Isso é fundamental: Se inicializássemos todos os pesos com zero ou um único valor constante, todas as 4 cabeças de atenção fariam exatamente os mesmos cálculos.
Nota: Keras usa inicializador Glorot Uniform (Xavier), que no nosso exemplo gera números aleatórios no intervalo [-0.2165, +0.2165].

À medida que o treinamento começa usando ‘model.fit()’, o otimizador (AdamW) passa a atualizar esses valores livremente para buscar minimizar o erro de classificação.

Multiplicando o tensor de entrada T (16×64 que representa os 16 patches) pelas matrizes W_Q, W_K e W_V (64×64), obtemos as matrizes Q, K e V (query, key e value) de dimensões 16×64.

$$\begin{cases} Q = T \bullet W_Q \\ K = T \bullet W_K \\ V = T \bullet W_V \end{cases}$$

Nota: Os termos Query (Consulta), Key (Chave) e Value (Valor) foram emprestados de sistemas de Recuperação de Informação. Imagine que você está usando o Google:

- **Query (Q):** É o que você digita na barra de busca.
- **Key (K):** O sistema compara sua *Query* com todas as *Keys* disponíveis para ver quais são relevantes para a sua busca.
- **Value (V):** Uma vez que o sistema decide que uma *Key* é relevante para a sua *Query*, ele entrega a informação contida no *Value*.

No Transformer, esses três elementos são vetores gerados a partir da mesma entrada (no caso do *Self-Attention*).

1. **Comparação (Q×K):** Calculamos o produto escalar entre a *Query* de um elemento e as *Keys* de todos os outros. Isso gera um "score" de afinidade (quem deve prestar atenção em quem).
2. **Filtro (Softmax):** Esses scores são normalizados para somarem 1, linha a linha (pesos de atenção).
3. **Extração (Score×V):** O resultado final é uma soma ponderada dos *Values*. Se a *Query* de um patch de um nódulo na mamografia “combina” com a *Key* de outro patch suspeito, o valor desse segundo patch terá um peso alto na representação final.

Ao utilizarmos múltiplas cabeças (*num_heads* = 4), dividimos as 64 dimensões em subespaços de dimensão 16, resultando em matrizes Q, K e V 16×16 para cada cabeça (haverá 4 matrizes desses, uma para cada cabeça, não usaremos índices para simplificar a notação). Cada cabeça pode se especializar numa relação diferente entre pares de patches. Por exemplo, cada cabeça poderia detectar uma forma diferente de escrever dígito “7” (com 2, 3 ou 4 traços, cortado ou não).

A matemática efetuada em cada cabeça de atenção é a *Scaled Dot-Product Attention*:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

QK^T : Calcula a matriz de afinidade/similaridade entre os patches. Cada patch está sendo comparado com todos os outros. O resultado tem dimensão 16×16.
$\sqrt{d_k}$: Fator de escala (onde d_k = D / *num_heads* = 16/4 = 4) para evitar saturação do gradiente.
Softmax: Normaliza as afinidades em probabilidades que somam 1, linha a linha.
V: O conteúdo visual a ser propagado.
O resultado final será um mapa de atenção 16×16, mostrando qual patch está relacionado com qual.
Repetindo esse processo para 4 cabeças, teremos 4 mapas de atenção 16×16.

O mecanismo de atenção compara cada patch com todos os outros através do produto escalar. A matriz Q (16×64) é multiplicada pela matriz K^T (64×16), resultando numa matriz 16×16. Essa multiplicação matricial compara cada patch com todos os outros. Depois do fator de escala, *softmax* e

multiplicação por V, resulta no mapa de atenção 16×16. Esta operação é repetida para h=4 cabeças para comparar diferentes aspectos dos patches, resultando em 4 mapas de atenção 16×16.

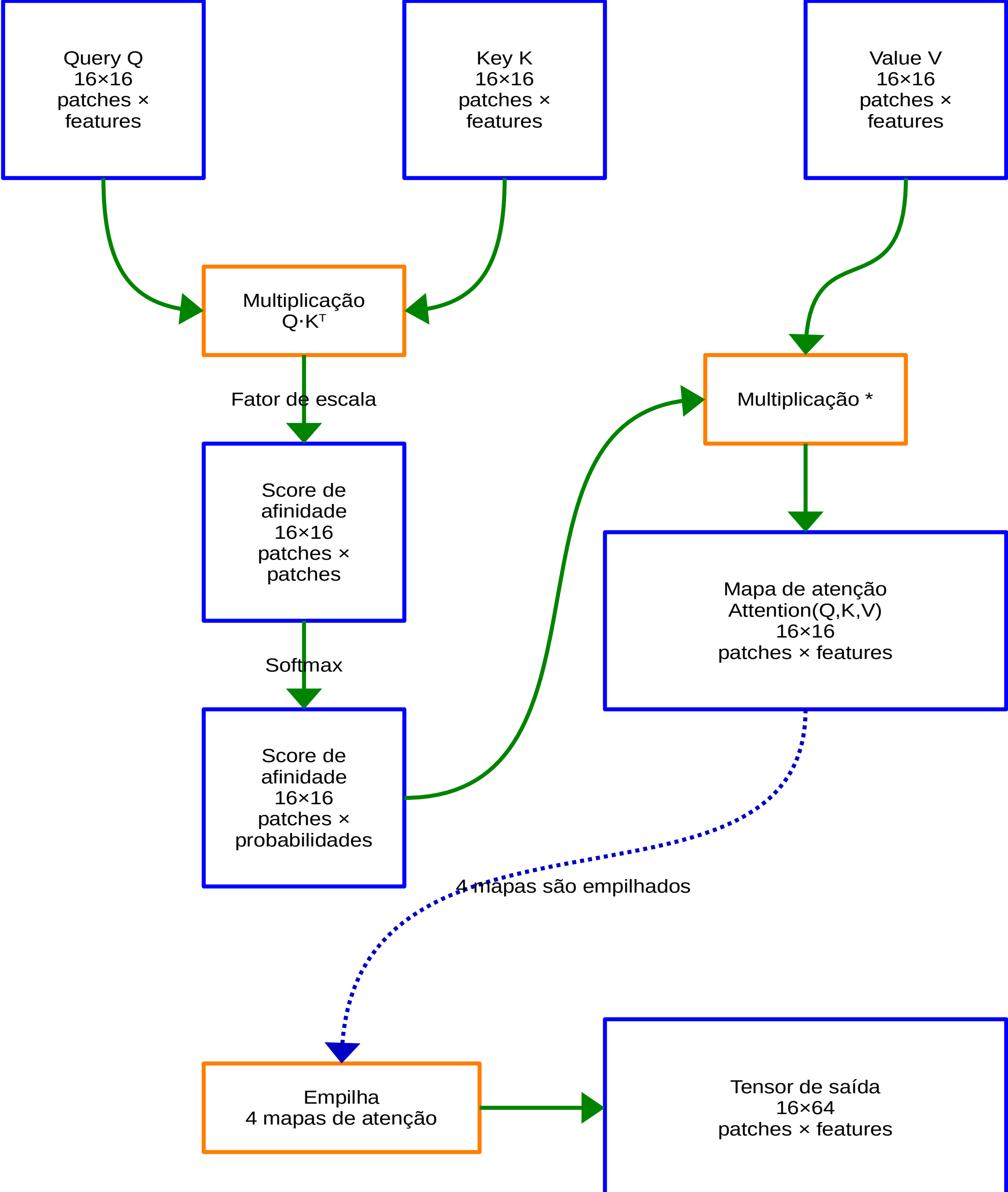

Figura J: Cálculo de mapa de atenção *Attention*(Q, K, V) e empilhamento das h=4 mapas, para gerar o tensor final do bloco transformer 16×64. O bloco transformer é aplicado 4 vezes.

A **Multi-Head Attention (MHA)** é como ter **vários especialistas** olhando para a mesma imagem ao mesmo tempo, cada um procurando por algo diferente. Imagine que você está analisando uma mamografia.

- Um especialista foca apenas na **textura** das microcalcificações.
- Outro foca na **simetria** entre a mama esquerda e a direita.
- Um terceiro foca nas **bordas** de uma possível massa.

O processo segue três etapas principais:

1. **Divisão (Split):** O vetor de entrada (com dimensão *D*, por exemplo, 64) é projetado linearmente em Q, K e V e dividido em *h* partes (por exemplo, *h*=4).
2. **Atenção em Paralelo:** Cada cabeça aplica o mecanismo de *Self-Attention* (Query, Key, Value) de forma independente. Como os pesos iniciais de cada cabeça são diferentes, cada uma "aprende" a focar em relações diferentes.
3. **Concatenação e Projeção:** Os resultados de todas as cabeças são colados uns nos outros (concatenados) e passados por uma última camada linear para voltarem ao tamanho original.

Em implementações de alto desempenho (como Keras ou PyTorch), não se usa loop para as *h* cabeças. Em vez disso, usamos um truque de redimensionamento de tensores para que todas as cabeças sejam processadas em uma única multiplicação de matrizes.

1. O Truque das Dimensões

Imagine a matriz de entrada com dimensão (16, 64) - 16 patches com 64 atributos cada um. Se queremos 4 cabeças, o código não cria 4 matrizes. Ele apenas "enxerga" a matriz de um jeito diferente:

1) Projeção: Primeiro, ele multiplica a entrada por pesos para gerar Q, K, V ainda no tamanho (16, 64).
2) Reshape: Ele transforma a matriz de (16, 64) para (16, 4, 16).
   a) **16** continua sendo o número de patches (tokens).
   b) **4** é o número de cabeças.
   c) **16** é a dimensão interna de cada cabeça (64 / 4 = 16).
3) Transpose (Permutação): Ele troca as posições para ficar (4, 16, 16).

2. Multiplicação Matricial em Lote (Batch Matrix Multiplication)

Agora que os dados estão no formato (4, 16, 16), o framework de Deep Learning trata o número de cabeças (*h*=4) como se fosse um **batch** (um lote). Quando você faz Q·K, o computador executa uma operação chamada **Batch MatMul**:

- Ele multiplica a "fatia" da cabeça 1 de Q pela "fatia" da cabeça 1 de K.
- Faz o mesmo para a cabeça 2, 3 e 4.
- **Tudo isso acontece simultaneamente na GPU**, aproveitando os milhares de núcleos de processamento paralelo dela.

3. Como o código "volta" ao normal?

Depois da multiplicação e do Softmax, o resultado ainda está "fatiado" em 4 cabeças. O código então:

- Faz o caminho inverso: transforma de (4, 16, 16) para (16, 4, 16).
- Dá um **Concatenate** (ou reshape final) para voltar a ser (16, 64).
- Passa por uma última matriz de pesos (Dense) para misturar as informações que cada cabeça aprendeu.

Como é possível detectar, usando produto escalar que, para representar dígito "7", reta horizontal no patch #5 deve "casar" com esquina no patch #6 (figura H)? Parece que o produto escalar entre o vetor que representa reta horizontal com vetor que representa esquina não deve resultar em valor muito alto.

Repare que o produto escalar não é calculado entre os valores dos pixels dos patches, mas entre esses valores transformados. Os conteúdo de um patch (49 valores) passa por 3 transformações aprendíveis por backpropagation antes de calcular a atenção usando produto escalar:

1. Projeção linear (embedding) que transforma patch com 49 valores em vetor com *D*=64 elementos.
2. Positional embedding que soma informação de localização espacial do patch.
3. Multiplicação matricial por pesos $W_Q, W_K, W_V \in \boldsymbol{R}^{\boldsymbol{D \times D}}$ para gerar matrizes *Q*, *K* e *V* (query, key e value) de dimensões 16×64. Estas matrizes são quebradas em 4 cabeças.

As 3 transformações aprendíveis acima são atualizadas pela backpropagation para extrair features que permitem comparar pares de patches com a finalidade de classificar os 10 dígitos. Pares de patches que devem aparecer juntos para formar dígito "7" são mapeados em vetores de features semelhantes, para que o produto escalar entre eles dê um resultado grande.

4.2. Perceptron Multicamadas (MLP)

Após o bloco transformer (bloco vermelho da figura F), os dados passam por uma rede feed-forward clássica (bloco magenta da figura F) que expande a dimensão temporariamente (para $2 \times D = 128$) e a reduz de volta para D=64, aplicando a ativação não-linear GELU (Gaussian Error Linear Unit).

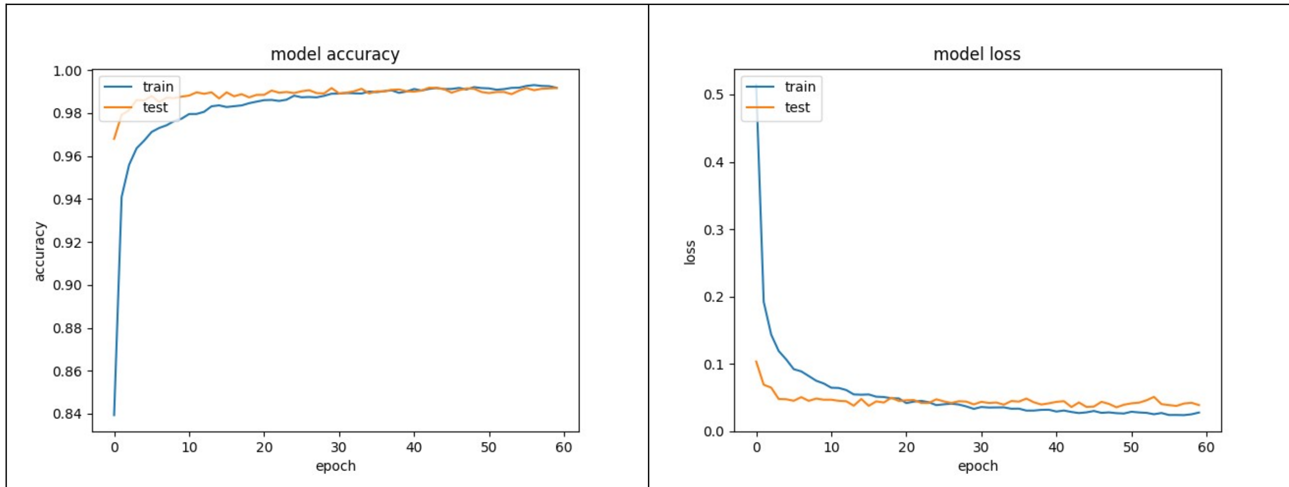

Figura: Acuracidade e loss ao longo das épocas.

5. Tópicos Avançados para Discussão

Para além da implementação, os seguintes tópicos devem pautar as análises empíricas e teóricas de estudantes avançados:

1. **A ausência de Viés Indutivo:** Ao contrário das CNNs, o ViT não “sabe” que pixels (patches) adjacentes estão relacionados. Ele precisa aprender as relações espaciais através do dataset. Por isso, ViTs puros costumam performar pior que CNNs em datasets pequenos, a menos que passem por forte regularização ou pré-treinamento massivo.

2. **Visualização de Atenção:** Uma das maiores vantagens do ViT é a interpretabilidade. Ao extrairmos os pesos da matriz $QK^T/\sqrt{d_k}$, podemos plotar mapas de calor que mostram exatamente em quais partes da imagem a rede “prestou atenção” para classificar o dígito.

3. **Complexidade Quadrática:** O custo computacional da auto-atenção cresce em relação a $O(N^2)$, onde N é o número de patches. Ao aumentar a resolução de uma imagem, o número de patches cresce e o custo explode, abrindo espaço para pesquisas em Efficient Transformers. Porém, para N razoável, auto-atenção pode ser calculada em paralelo.

Nota: Os programas estão em ~/deep/keras/transformer
Veja arquivos ~/haepi/apostila/transformer.odt e ~/haeweb/apostila/transformer.odt

Boa explicação da diferença entre transformer e CNN:
https://www.youtube.com/watch?v=KnCRTP11p5U
https://www.youtube.com/watch?v=jNPmSjrxGtY

Outro vídeo interessante:
https://www.youtube.com/watch?v=j3VNqtJUoz0&t=19s

Livro dive into deep learning:
https://d2l.ai/chapter_attention-mechanisms-and-transformers/vision-transformer.html

[PSI5790 aula 9, lição de casa extra #1. Vale +4.] Aumente a acuracidade do programa *P* acima de 99.50%, fazendo data augmentation e outros “truques”. Descreva quais foram as alterações introduzidas. A única técnica que não pode aqui é fazer transfer learning.

ViT pré-treinado em ImageNet.

Assim como existem vários CNNs pré-treinados em ImageNet, existem vários modelos ViT pré-treinados. Uma forma de acessá-los é através do pacote keras_hub.

O código abaixo baixa algumas imagens exemplos.

```
url='http://www.lps.usp.br/hae/apostila/cifar_pretreinado.zip'
import os; nomeArq=os.path.split(url)[1]
if not os.path.exists(nomeArq):
  print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
  os.system("wget -U 'Firefox/50.0' "+url)
else:
  print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
print("Descompactando arquivos novos de",nomeArq)
os.system("unzip -u "+nomeArq)
```

O programa abaixo classifica uma imagem usando ViT-B/16:

```
#~/algpi/ep/2026-pos-ViT/ViT-B16/ViT-B16b.py
import os
os.environ["KERAS_BACKEND"] = "tensorflow"

import tensorflow.keras as keras
import keras_hub
import numpy as np
from PIL import Image

# 1. Selecionar o modelo (Presets nativos funcionam sem autenticação)
# Opções Transformer: "vit_base_patch16_224_imagenet", "deit_base_distilled_patch16_224_imagenet"
# Opções CNN: "resnet_50_imagenet", "efficientnet_b0_ra_imagenet", "vgg_16_imagenet"

preset = "deit_base_distilled_patch16_224_imagenet"
print(f"Carregando o modelo: {preset}...")

try:
    classifier = keras_hub.models.ImageClassifier.from_preset(preset)
except Exception as e:
    print(f"Erro ao carregar preset {preset}: {e}")
    print("Tentando o modelo padrão vit_base_patch16_224_imagenet...")
    classifier = keras_hub.models.ImageClassifier.from_preset("vit_base_patch16_224_imagenet")

# 2. Carregar o mapeamento de classes ImageNet
def load_classes(file_path):
    classes = {}
    if not os.path.exists(file_path):
        return {i: f"Classe {i}" for i in range(1000)}
    with open(file_path, 'r') as f:
        for line in f:
            parts = line.strip().split(', ')
            if len(parts) >= 2:
                classes[int(parts[0])] = parts[1]
    return classes

classes = load_classes("imagenet_classes.txt")

def classify_image(img_path):
    if not os.path.exists(img_path):
        print(f"Erro: Arquivo {img_path} não encontrado.")
        return

    # 3. Carregar e preparar a imagem
    img = Image.open(img_path).convert("RGB")
    img = img.resize((224, 224))

    img_array = np.array(img)
    img_array = np.expand_dims(img_array, axis=0)

    # 4. Predição
    predictions = classifier.predict(img_array, verbose=0)

    # 5. Aplicar Softmax
    probs = keras.activations.softmax(predictions).numpy()[0]

    # 6. Obter Top 5
    top_indices = np.argsort(probs)[-5:][::-1]

    print(f"\nResultado para: {img_path}")
    for i, idx in enumerate(top_indices):
        class_name = classes.get(idx, f"ID {idx}")
        prob = probs[idx]
        print(f"{i+1}. {class_name} ({idx}): {prob:.4f}")

    return top_indices[0]

if __name__ == "__main__":
    import sys
    img_to_test = sys.argv[1] if len(sys.argv) > 1 else "chimpanzee.jpg"
```

```
classify_image(img_to_test)
```

~/algpi/ep/2026-pos-ViT/ViT-B16/ViT-B16b.py

Modelos preset prontos para serem usados obtidos com comando:
print(list(keras_hub.models.ImageClassifier.presets.keys()))
['efficientnet_b0_ra_imagenet', 'efficientnet_b0_ra4_e3600_r224_imagenet', 'efficientnet_b1_ft_imagenet', 'efficientnet_b1_ra4_e3600_r240_imagenet', 'efficientnet_b2_ra_imagenet', 'efficientnet_b3_ra2_imagenet', 'efficientnet_b4_ra2_imagenet', 'efficientnet_b5_sw_imagenet', 'efficientnet_b5_sw_ft_imagenet', 'efficientnet_el_ra_imagenet', 'efficientnet_em_ra2_imagenet', 'efficientnet_es_ra_imagenet', 'efficientnet2_rw_m_agc_imagenet', 'efficientnet2_rw_s_ra2_imagenet', 'efficientnet2_rw_t_ra2_imagenet', 'efficientnet_lite0_ra_imagenet', 'vit_base_patch16_224_imagenet', 'vit_base_patch16_384_imagenet', 'vit_large_patch16_224_imagenet', 'vit_large_patch16_384_imagenet', 'vit_base_patch32_384_imagenet', 'vit_large_patch32_384_imagenet', 'vit_base_patch16_224_imagenet21k', 'vit_base_patch32_224_imagenet21k', 'vit_huge_patch14_224_imagenet21k', 'vit_large_patch16_224_imagenet21k', 'vit_large_patch32_224_imagenet21k', 'mobilenetv5_300m_enc_gemma3n', 'deit_base_distilled_patch16_384_imagenet', 'deit_base_distilled_patch16_224_imagenet', 'deit_tiny_distilled_patch16_224_imagenet', 'deit_small_distilled_patch16_224_imagenet', 'mit_b0_ade20k_512', 'mit_b1_ade20k_512', 'mit_b2_ade20k_512', 'mit_b3_ade20k_512', 'mit_b4_ade20k_512', 'mit_b5_ade20k_640', 'mit_b0_cityscapes_1024', 'mit_b1_cityscapes_1024', 'mit_b2_cityscapes_1024', 'mit_b3_cityscapes_1024', 'mit_b4_cityscapes_1024', 'mit_b5_cityscapes_1024', 'csp_darknet_53_ra_imagenet', 'csp_resnext_50_ra_imagenet', 'csp_resnet_50_ra_imagenet', 'darknet_53_imagenet', 'densenet_121_imagenet', 'densenet_169_imagenet', 'densenet_201_imagenet', 'vgg_11_imagenet', 'vgg_13_imagenet', 'vgg_16_imagenet', 'vgg_19_imagenet', 'hgnetv2_b4_ssld_stage2_ft_in1k', 'hgnetv2_b5_ssld_stage1_in22k_in1k', 'hgnetv2_b5_ssld_stage2_ft_in1k', 'hgnetv2_b6_ssld_stage1_in22k_in1k', 'hgnetv2_b6_ssld_stage2_ft_in1k', 'xception_41_imagenet', 'mobilenet_v3_small_050_imagenet', 'mobilenet_v3_small_100_imagenet', 'mobilenet_v3_large_100_imagenet', 'mobilenet_v3_large_100_imagenet_21k', 'resnet_18_imagenet', 'resnet_50_imagenet', 'resnet_101_imagenet', 'resnet_152_imagenet', 'resnet_v2_50_imagenet', 'resnet_v2_101_imagenet', 'resnet_vd_18_imagenet', 'resnet_vd_34_imagenet', 'resnet_vd_50_imagenet', 'resnet_vd_50_ssld_imagenet', 'resnet_vd_50_ssld_v2_imagenet', 'resnet_vd_50_ssld_v2_fix_imagenet', 'resnet_vd_101_imagenet', 'resnet_vd_101_ssld_imagenet', 'resnet_vd_152_imagenet', 'resnet_vd_200_imagenet']

Outros modelos (como Swin Transformer e ConvNeXt) podem ser baixados de Kaggle ou Hugging face.

chimpanzee.jpg 1. chimpanzee (367): 0.7641 2. patas (371): 0.0130 3. siamang (369): 0.0095 4. guenon (370): 0.0084 5. gorilla (366): 0.0045	coelho.jpg 1. wood_rabbit (330): 0.4072 2. hare (331): 0.2612 3. Angora (332): 0.0710 4. Persian_cat (283): 0.0039 5. mink (357): 0.0036	eiffel.jpg 1. stupa (832): 0.5107 2. palace (698): 0.0695 3. fountain (562): 0.0419 4. flagpole (557): 0.0232 5. church (497): 0.0141

[PSI5790 aula 9 parte 1. Fim.]