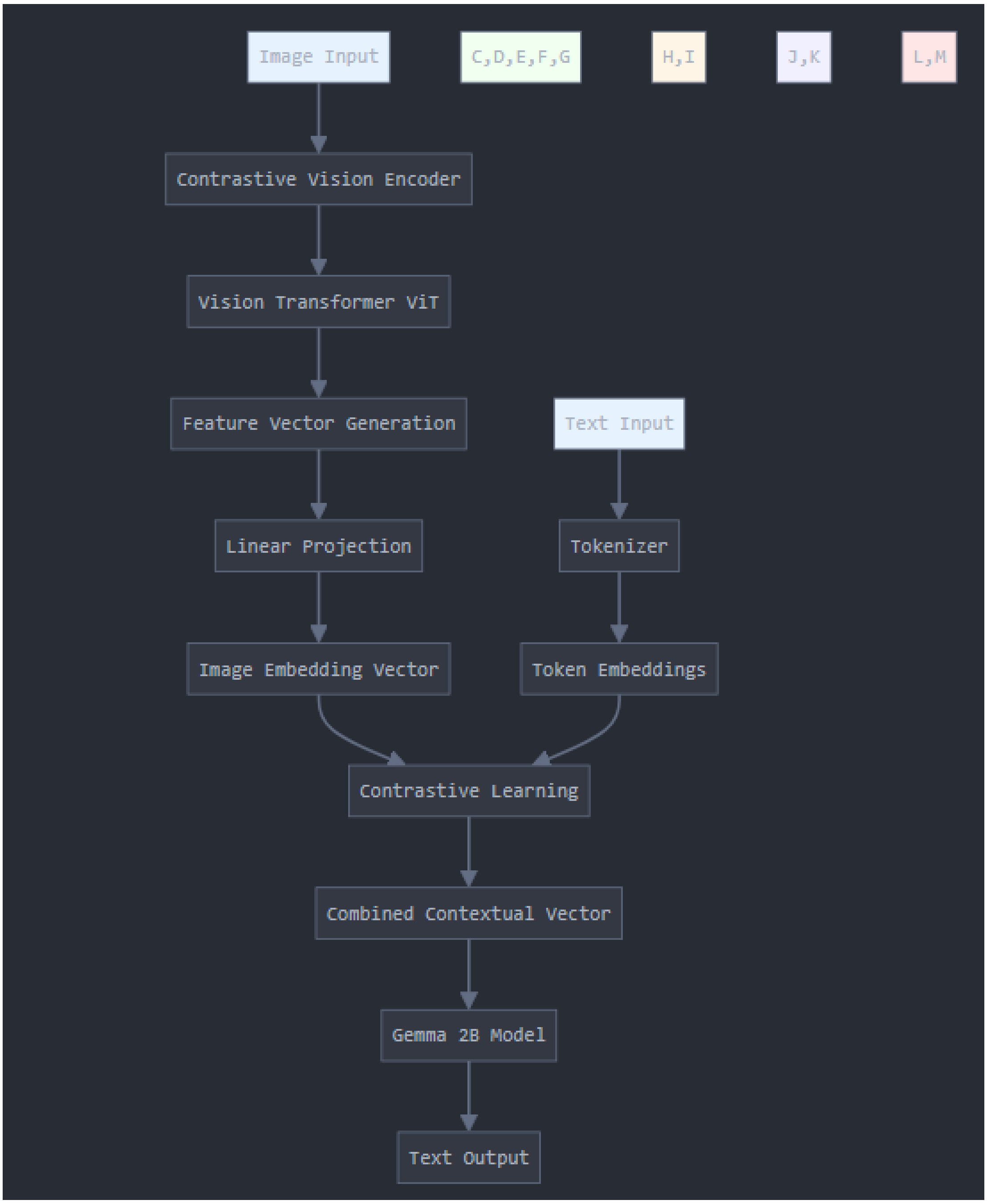
1ST of all we design the **PaliGemma Vision language model** so in that model target to generate the final contexiual text on the base of images and texts as input.

Let see how the whole process is going on.

1st we give the input to the model (texts and Images) like in Images we give the images for somethings and in text we ask the questions related to that,so the model target to generate the contexiual output in form of text on the basis of given information(images and texts). Now we discuss how the working in between the input and final output is processd, what function we used and what algorthim we used so its give best result in their final output on the basis of given inputs.

So 1st of all we give the overview how complete things is going on and after this we explain all the things step wise.



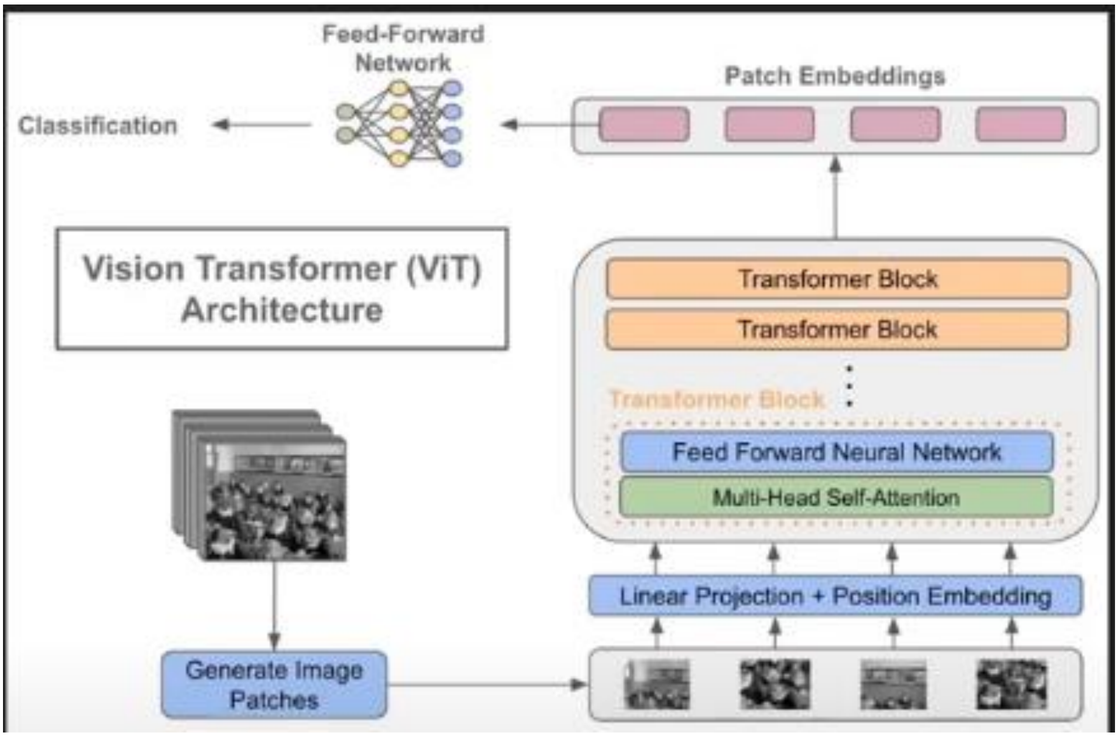
The above process are the whole process , now we explain the all components step by step and the things which we use in it. After that we define that , we use the **SigLIP Vision Model.** The whole process till generate the combined contextual vector is the part of that Vision model. Because its task to combined the embedding vectors of both modalities and then apply the contrastive learning approach which generate the best contextual vector of both modalities , which define the relationship in btw text and image. So we explain the components which used under **SigLIP Vision Model.**

1st we take the image process:

So from the image side 1st take the image as a input and then apply contrastive Vision encoder. Now the ViT is the one of the type of contrastive vision encoder to encode the image, means to extract the image features and represent this features in numerical format called as embedding vector. We also used it ResNet instead of ViT , but use because for complex task or large dataset we use ViT. Now we discuss ViT.

Vision Transformer(ViT):
Let me 1st explain about it, so the vision transformer is simply we explain vision and transformer , so it means just like it work for NLP task(text base) similarly it work for vision(images).

The target of this transformer to encode the image in embedding vectors. So let see how they do that , we define step wise how ViT work on backend.



So let me explain you step wise:

Step-1: In that we take the image as input and then we divide this image in patches by applying the convolution(nn.**Conv2d).**

```python
class SiglipVisionEmbeddings(nn.Module):
    def __init__(self, config: SiglipVisionConfig):
        super().__init__() #_init_ constructor which accept the config parameters
        self.config= config #Store the config inside the class
        self.embed_dim= config.hidden_size #this define that what is the dimension of embedding vector
        self.image_size = config.image_size
        self.patch_size= config.patch_size

        self.patch_embedding = nn.Conv2d( # patch_embedding are the convolutional layer which use to divide the image to the patches.
            in_channels=config.num_channels,
            out_channels=self.embed_dim,#which are equal to the embedding dimension of all patch.
            kernel_size=self.patch_size,
            stride=self.patch_size,
            padding="Valid",#this indicates no padding is added
        )
```

In the above code 1st we initialize the patch_embedding layer, it's the convolution layer which are used to divide the image in patches. Its divide the image in small chunks which is called as patches. The same thing we do in the above code we assign the same size to the patch_embedding layer (**Kernel_size=self.patch_size).** And in **stride=self.patch_size** we remove the collsion issue btw two patches. Means after extracting the 1 patch we directly jump to the other patch, because stride size equal to the patch size . if its size is equal to the size of pixels then its collide.

So In the above image the all box are represent the patches, now in every box have the pixels, let our image size if 224 X 224 and the patches size is 16 X 16, Now How many patches in that feature is 224 X 224/ 16 X 16=196. And let every pixel size is 768. Now the image are coloured so the coloured image have 3 channels R(Red),G(Green),B(Blue) which we define (**in_channels=config.hidden_size)**. So we multiply the every parameter by 3. Note- The value of R,G,B is define the intensity of that channel in particular part of the image.

After divide the image, the **Patch_embedding** layer convert this patches in numerical values in vector representation.

```
patch_embeds = self.patch_embedding(pixel_values)
```

Step-2: After that convert this vector representation patches in 1D form.

```
embeddings = patch_embeds.flatten(2)
```

after that the shape of initial vector representation [Batch_Size, Embed_Dim, Num_Patches_H, Num_Patches_W] convert in [Batch_Size, Embed_Dim, Num_Patches].

```
embeddings = embeddings.transpose(1, 2)
```

In that Batch_Size is total number of patches in the batch, Embed_Dim is total number of dimension of each patch's embedding.
Num_patches_H is number of patches along the hight of the image(horizontally).
Num_patches_W is number of patches along the width of the image(vertically).

We flatten(2) mean's we flatten the vector who's dimension is >=2.
so in that we combine both Num_patches_H and Num_patches_W in 1D num_patches. Like …._H=4 and ….._W=4 so Num_patches=16. After that we perform positional embedding and Linear projection.

Step-3: Linear Projection: What is the need??
so after generating the 1D embedding vectors of every patch's , we provide that vector's to the transformer as input but transformer are design to work with fixed-size vectors but image's are 2D grid which have may be multiple channles, so every patches have different number of pixels as per channels intensity. So to manage this disturbance in the size of vector, we use linear projection.
So using the Linear projection map the 1D vector in higher dimension space(size Embed_Dim). So its easy to extract the contextual information.

Now the benifits of the Linear Projection is that to reduce the size of pixels pr patch. So its perform the Linear projection on the basis of
Y=XW+b , where X is the 1D vector who's size is 768 and Y is output(projected vector) and W weight matrix and B is bias.

Now the benefit to reduce the size of pixels is that.
I-We have required a less resource, like memory to store the pixels, let we reduce their size 768 to 512.

In the given code the portion perform the linear projection Is

```
patch_embeds = self.patch_embedding(pixel_values)
```

Means the convolution itself perform the linear projection.

II- Another benefit is ,when the linear projection is reduce the size of pixels , it
always remove those portion which are noisy in all over pixels or we can uninformative. So that after that only those pixels are their which are much informative and valuable for training point of view.

Step-4: Position Embedding:
Let we provide the input like 196 patches(1D) and every patch have 512 pixels to the transformer encoder so the encoder are not know the original sequence.

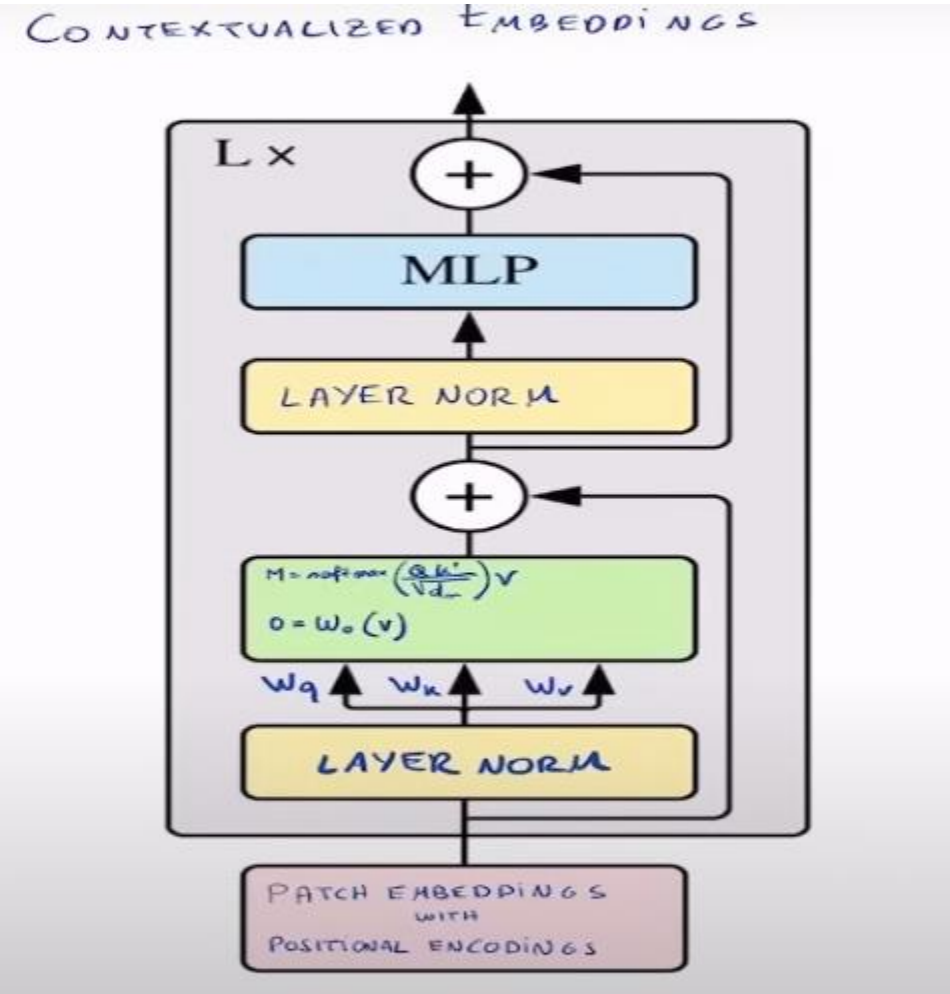
Means as simple we can say that its define the position of the specific patch in the original images. Because with the help of this model understood the structure of the original image or the spatial relationship in btw the patches.

```
# Add position embeddings to each patch. Each positional encoding is a vector of size [Embed_Dim]
embeddings = embeddings + self.position_embedding(self.position_ids)
# [Batch_Size, Num_Patches, Embed_Dim]
return embeddings
```

After that return the final embedding vectors which ready to provide as input to the transformer.

```
def forward(self, pixel_values: torch.FloatTensor) -> torch.Tensor:
    _, _, height, width = pixel_values.shape # [Batch_Size, Channels, Height, Width]
    # Convolve the `patch_size` kernel over the image, with no overlapping patches since the stride is equal to the kernel size
    # The output of the convolution will have shape [Batch_Size, Embed_Dim, Num_Patches_H, Num_Patches_W]
    # where Num_Patches_H = height // patch_size and Num_Patches_W = width // patch_size
    patch_embeds = self.patch_embedding(pixel_values)
    # [Batch_Size, Embed_Dim, Num_Patches_H, Num_Patches_W] -> [Batch_Size, Embed_Dim, Num_Patches]
    # where Num_Patches = Num_Patches_H * Num_Patches_W
    embeddings = patch_embeds.flatten(2)
    # [Batch_Size, Embed_Dim, Num_Patches] -> [Batch_Size, Num_Patches, Embed_Dim]
    embeddings = embeddings.transpose(1, 2)
    # Add position embeddings to each patch. Each positional encoding is a vector of size [Embed_Dim]
    embeddings = embeddings + self.position_embedding(self.position_ids)
    # [Batch_Size, Num_Patches, Embed_Dim]
    return embeddings
```

Architecture of Transformer encoder:



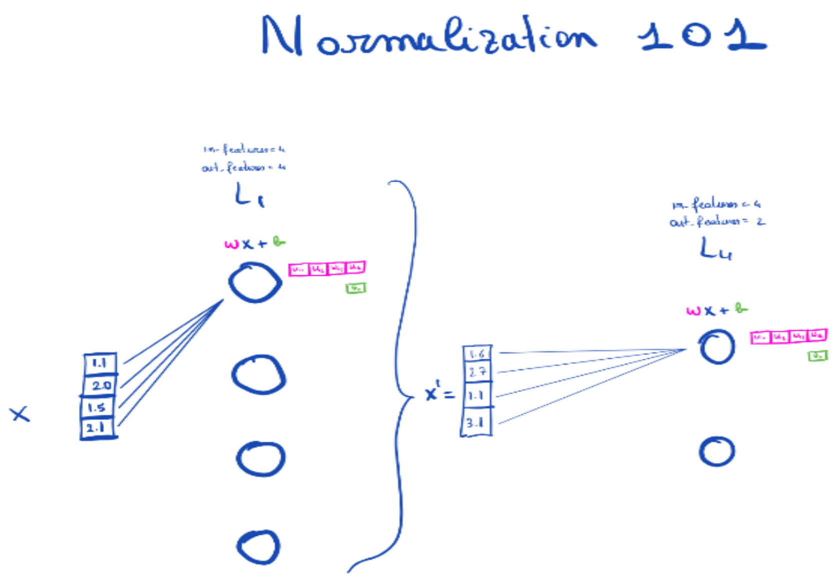
Now we discuss the process after the after, when we generate the embedding vectors for all patches with their positional embedding. After that we perform we send as input to the Transformer encoder which generate the contextualizes embeddings vectors. Which are define the relation in btw the patches. So let see how they generate it.

1st of all when we give as a input to the encoder , the encoder perform the layer normalization. Now what is the need?
1st of all we explain why normalization is required, so when we train the model ,output of current layer work as a input to the next layer (layer is encoder's which use in transformer). If the output of the any layer will fluctuate largely , so it difficult to process by the second layer. So that the normalization ensure the output of every layer is consistent range. So it easy to process by the layer.

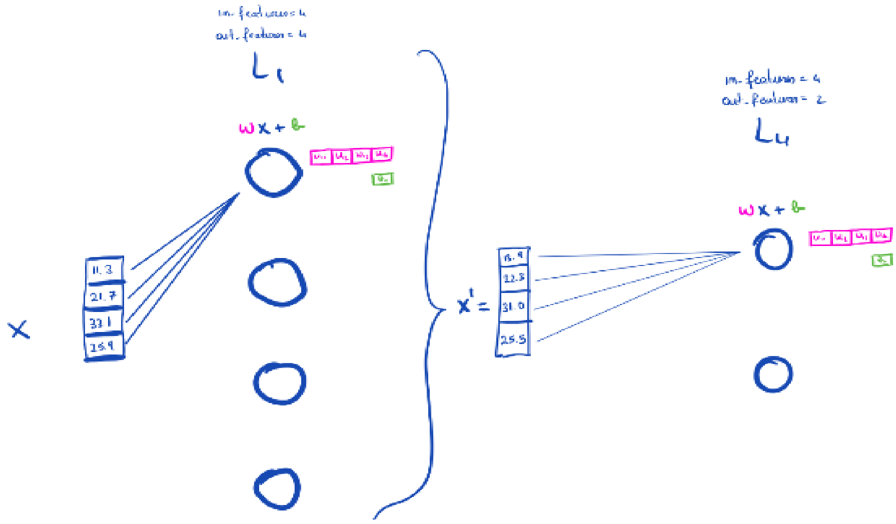
Now why layer normalization not other's.
So initially when we perform the simple normalization which normalise the

input data and set in fixed range. But in deep network , there are the shift in the layer, called as **covariate shift**.

Covariate Shift: This is problem which occur if the distribution of input data are change in training and testing time . let we provide the input on training time of different senerio and on testing is different. So for that its directly effected to the model performance. Similarly in deep network output of current layer work as input of the next layer so that's why input output distribution are very important because it changeable , when their distribution is change so its effect the model training time , difficult to proceed. So that the reason we use layer normalization ,which manage this distribution range so that its easy to the model.
Now why we not used the batch normalization , so we are working with the transformer which work on the sequence2sequence data, and some time only one input(1 sentence) in the sequence , and batch normalization are not work with single inputs. Because of batch size is lower ,so to calculate the reliable mean and variance is difficult. so we use the layer normalization which are not depend on the batch size , its directly normalise each embeddings independently.



Problem: covariate shift



Why is it bad?

Big change in input of a layer ⟹ Big change in output of a layer ⟹ Big change in loss ⟹

⟹ Big change in gradient ⟹ Big change in the weights of the network

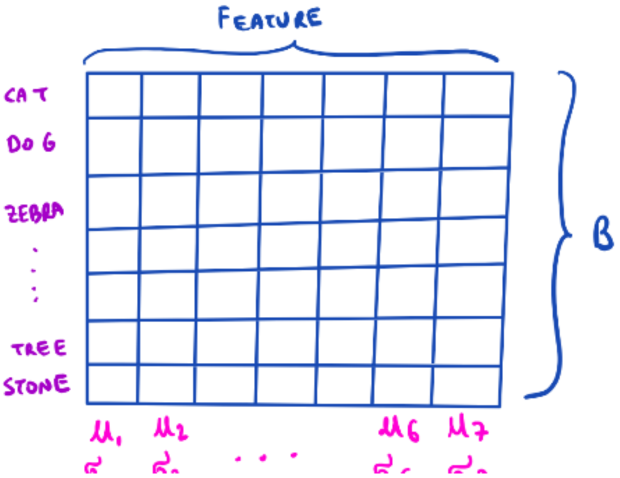
⟹ **Network learns slowly!**

[redacted]

3 Normalization via Mini-Batch Statistics

Since the full whitening of each layer's inputs is costly and not everywhere differentiable, we make two necessary simplifications. The first is that instead of whitening the features in layer inputs and outputs jointly, we will normalize each scalar feature independently, by making it have the mean of zero and the variance of 1. For a layer with d-dimensional input $\mathrm{x} = (x^{(1)} \ldots x^{(d)})$, we will normalize each dimension

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$



we introduce, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}, \beta^{(k)}$, which scale and shift the normalized value:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}.$$

These parameters are learned along with the original model parameters, and restore the representation power of the network. Indeed, by setting $\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$ and $\beta^{(k)} = \mathrm{E}[x^{(k)}]$, we could recover the original activations, if that were the optimal thing to do.

In the batch setting where each training step is based on the entire training set, we would use the whole set to normalize activations. However, this is impractical when using stochastic optimization. Therefore, we make the second simplification: since we use mini-batches in stochastic gradient training, *each mini-batch produces estimates of the mean and variance* of each activation. This way, the

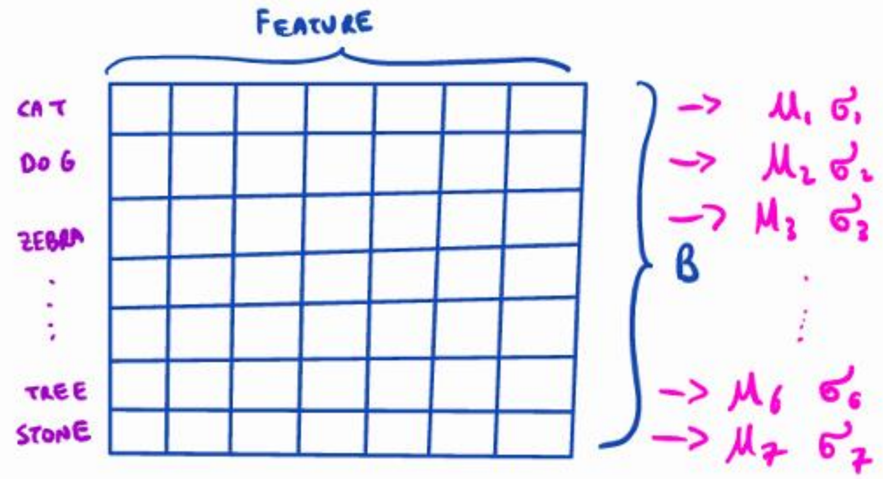
The problem with Batch Norm is that each statistic depends on what other items are in the batch. To get good results, we must use a big batch size

Layer Normalization:-

We now consider the layer normalization method which is designed to overcome the drawbacks of batch normalization.

Notice that changes in the output of one layer will tend to cause highly correlated changes in the summed inputs to the next layer, especially with ReLU units whose outputs can change by a lot. This suggests the "covariate shift" problem can be reduced by fixing the mean and the variance of the summed inputs within each layer. We, thus, compute the layer normalization statistics over all the hidden units in the same layer as follows:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2} \tag{3}$$



In this case each item is treated independently.

After that In apply Multi-Head attention Mechanism.

```python
class SigLipEncoderLayer(nn.Module): # nn.module yha pr pytorch ki base class ko inherit kr rha for neural networks.
  def __init__(self, config: SiglipVisionConfig):
    super().__init__()
    self.embed_dim=config.hidden_size
    self.self_attn = SiglipAttention(config)
    self.layer_norm1=nn.LayerNorm(self.embed_dim, eps=config.layer_norm_eps) # in that the layernorm1 and 2 for the layer normalization means layernorm1 perform
    self.mlp=SiglipMLP(config)# after that normalization we apply MLP as per architecture after that we again use
    self.layer_norm2=nn.LayerNorm(self.embed_dim,eps=config.layer_norm_eps)# layernorm2 to normalise the input which we provide the MLP and the output of MLP.
```

```python
def forward(self,hidden_states:torch.Tensor)->torch.Tensor:
  # residual: [Batch_Size, Num_Patches, Embed_Dim] batch size define how many images we take in one time.
    residual = hidden_states
    # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
    hidden_states = self.layer_norm1(hidden_states)
    # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
    hidden_states, _ = self.self_attn(hidden_states=hidden_states)
    # [Batch_Size, Num_Patches, Embed_Dim]
    hidden_states = residual + hidden_states
    # residual: [Batch_Size, Num_Patches, Embed_Dim]
    residual = hidden_states
    # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
    hidden_states = self.layer_norm2(hidden_states)
    # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
    hidden_states = self.mlp(hidden_states)
    # [Batch_Size, Num_Patches, Embed_Dim]
    hidden_states = residual + hidden_states

    return hidden_states
```

In that code we give the normalise input to the **Multi-Head attention mechanism.** Which fucntion we mention it **SiglipAttention(config).**

After that we again perform the layer normalization, then we perform the MLP(Multilayer prceptron) and then we perform the 2nd layer normalization.

How the hidden_states vector flow in whole algorthim as seen in above code **Forward(…………………………)->torch.Tensor:** and return the result till mlp outputs which gernate the contexual embedding vectors.

Mutli-Head attention:

Multi-Head Attention

$$X = \begin{bmatrix} [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \end{bmatrix}$$

Sequence of 4 items where each item is represented as a vector with 1024 dimensions.
Suppose number of heads $h = 8$

Our goal with MHA is to transform the initial sequence of <u>uncontextualized</u> embeddings into a sequence of contextualized embeddings.

VISION TRANSFORMER

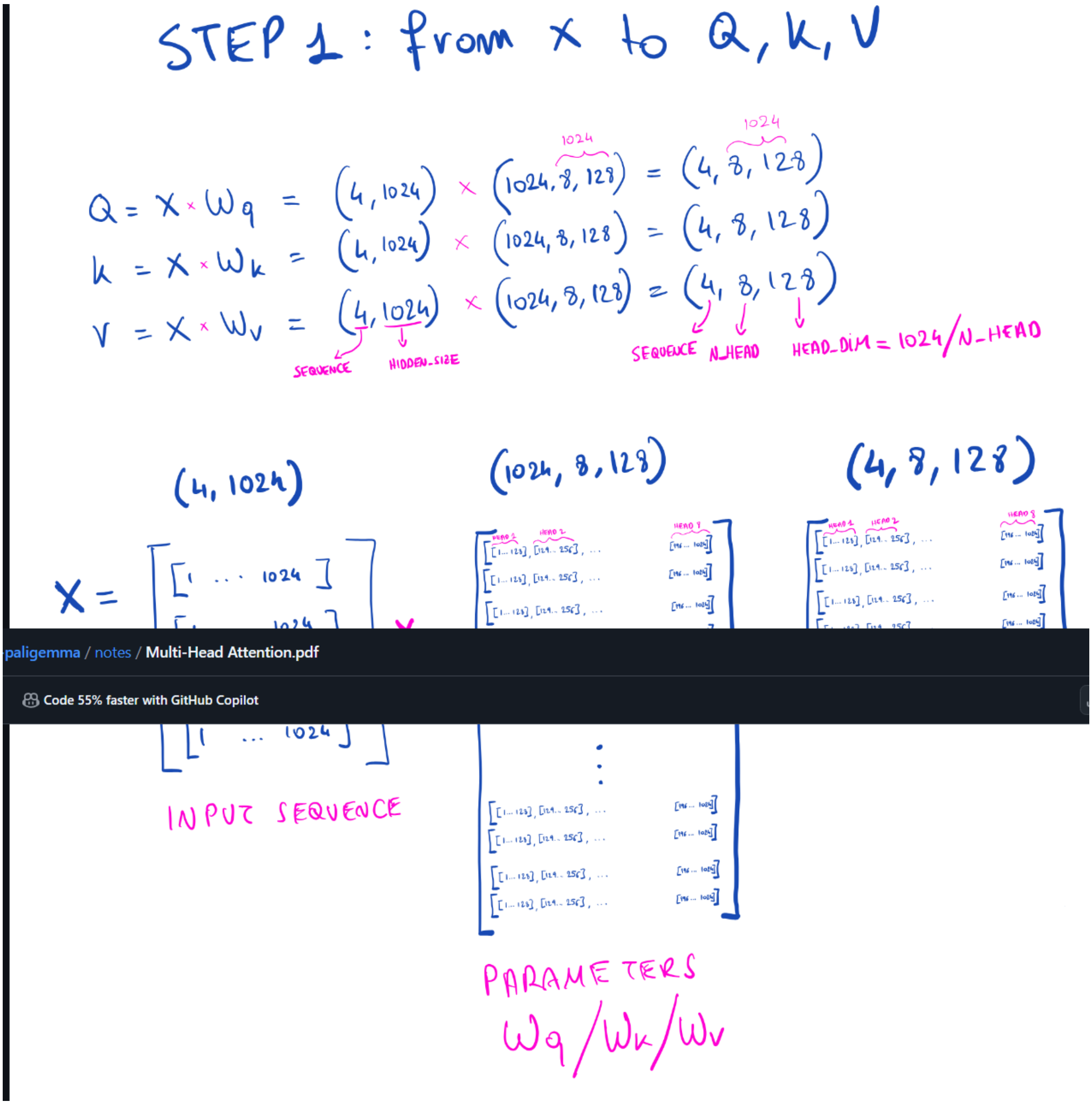
$$X = \begin{array}{cl} \begin{bmatrix} [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \end{bmatrix} & \begin{array}{l} \text{PATCH 1} \\ \text{PATCH 2} \\ \text{PATCH 3} \\ \text{PATCH 4} \end{array} \end{array} \Rightarrow X = \begin{array}{cl} \begin{bmatrix} [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \end{bmatrix} & \begin{array}{l} \text{PATCH 1, 2, 3, 4} \\ \text{PATCH 1, 2, 3, 4} \\ \text{PATCH 1, 2, 3, 4} \\ \text{PATCH 1, 2, 3, 4} \end{array} \end{array}$$

paligemma / notes / Multi-Head Attention.pdf

Code 55% faster with GitHub Copilot

$$X = \begin{array}{cl} \begin{bmatrix} [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \end{bmatrix} & \begin{array}{l} \text{I} \\ \text{LOVE} \\ \text{PEPPERONI} \\ \text{PIZZA} \end{array} \end{array} \Rightarrow X = \begin{array}{cl} \begin{bmatrix} [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \\ [1 & \cdots & 1024] \end{bmatrix} & \begin{array}{l} \text{I} \\ \text{I LOVE} \\ \text{I LOVE PEPPERONI} \\ \text{I LOVE PEPPERONI PIZZA} \end{array} \end{array}$$

In that "items" represent the patches and their vector whose dimension is 1024 means every patch have 1024 pixels. Now we understand the target of the Multi-head attention is that initially on left side the single image divide in 4 patches where every patch of size 1024. Then after apply the self attention mechanism so the senerio of multi head is create means, now we mix the information of the all patches like ,during calculating the similarity score so it calculate the multi-head in parllal so let on 1st we calulate the similarity score head 1 with repsect to other head ,same for other head , so this will help to figure out the relationship in btw the heads. And help to create the contextual vector which is the output of Multi-head attention.



STEP 2: TREAT EACH HEAD INDEPENDENTLY:

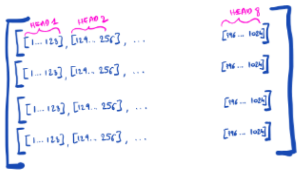
Q : (4, 8, 128)　　TRANSPOSE　　(8, 4, 128)

K : (4, 8, 128)　　$\Longrightarrow$　　(8, 4, 128)

V : (4, 8, 128)　　　　　　　　(8, 4, 128)

each head... ... will compute the attention scores independently from other heads by using a part of the entire embedding.

(4, 8, 128)



Q/K/V

$\Longrightarrow$

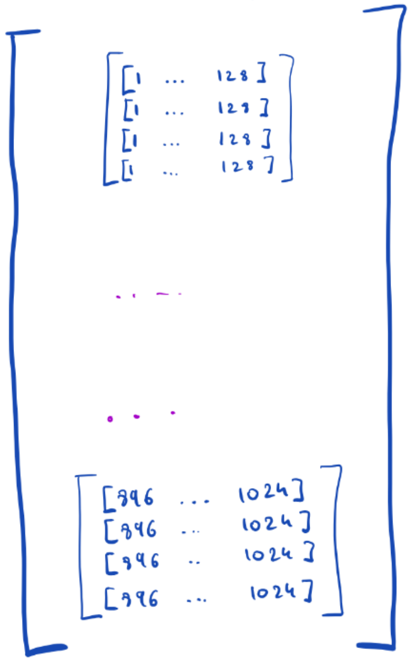
(8, 4, 128)

HEAD 1

HEAD 2

HEAD 7

HEAD 8



1) We want to parallelize the comput...

2) Each head should learn to relate tokens (or patches) differently

STEP 3: CALCULATE THE ATTENTION FOR EACH HEAD IN PARALLEL

$$Q_{HEAD_1} = \overset{(4,128)}{\begin{bmatrix} [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \end{bmatrix}} \begin{array}{l} \to \text{dimension } 1\dots128 \text{ of token } 1 \\ \to \text{dimension } 1\dots128 \text{ of token } 2 \\ \\ \to \text{dimension } 1\dots128 \text{ of token } 4 \end{array}$$

$$K^T_{HEAD_1} = \overset{(128,4)}{\begin{bmatrix} 1 & 1 & 1 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 128 & 128 & 128 & 128 \end{bmatrix}}$$

$\downarrow$ dimension $1\dots128$ of token 1 ... $\downarrow$ dimension $1\dots128$ of token 4

$$\frac{Q \times K^T}{\sqrt{d_{head}}} = Q\overbrace{}^{K}$$

	I	LOVE	PEPPERONI	PIZZA
I	13.9	21.1	-100.3	17.5
LOVE	-5.0	3.14	1.2	75.3
PEPPERONI	...	...	...	...
PIZZA	...	...	...	...

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_{head}}}\right) = Q\overbrace{}^{K}$$

	I	LOVE	PEPPERONI	PIZZA
I	0.1	0.2	0.5	0.3
LOVE	0.4	0.1	0.3	0.2
PEPPERONI	...	...	...	...
PIZZA	...	...	...	...

BRO, WHERE IS YOUR MASK?

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_{head}}} + \text{MASK}\right) = Q\overbrace{}^{K}$$

	I	LOVE	PEPPERONI	PIZZA
I	1.0	0	0	0
LOVE	0.6	0.4	0	0
PEPPERONI	0.2	0.4	0.4	0
PIZZA	0.4	0.2	0.3	0.1

STEP 4: MULTIPLY BY THE V SEQUENCE

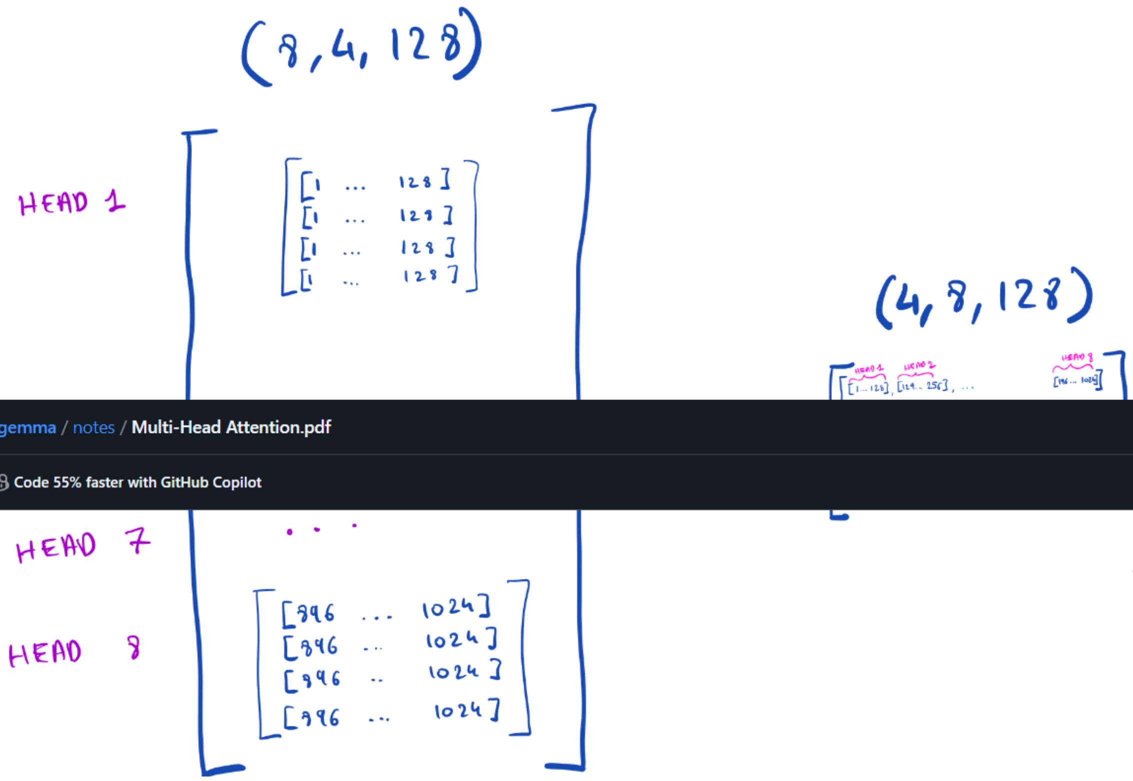
$$
\begin{array}{c}
\overbrace{\hphantom{xxxxxxxxxxxxxxx}}^{K} \\
Q\left\{\begin{array}{|c|c|c|c|}\hline 1.0 & 0 & 0 & 0 \\ \hline 0.6 & 0.4 & 0 & 0 \\ \hline 0.2 & 0.4 & 0.4 & 0 \\ \hline 0.4 & 0.2 & 0.3 & 0.1 \\ \hline \end{array}\right.
\end{array}
\begin{array}{l} \text{I} \\ \text{LOVE} \\ \text{PEPPERONI} \\ \text{PIZZA} \end{array}
\times
\begin{bmatrix} [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \end{bmatrix}
\begin{array}{l} \text{I} \\ \text{LOVE} \\ \text{PEPPERONI} \\ \text{PIZZA} \end{array}
$$

EACH ROW REPRESENTS A WEIGHTED SUM OF:

$$
= \begin{bmatrix} [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \end{bmatrix}
\begin{array}{l} \to \text{I} \\ \to \text{I LOVE} \\ \to \text{I LOVE PEPPERONI} \\ \to \text{I LOVE PEPPERONI PIZZA} \end{array}
$$

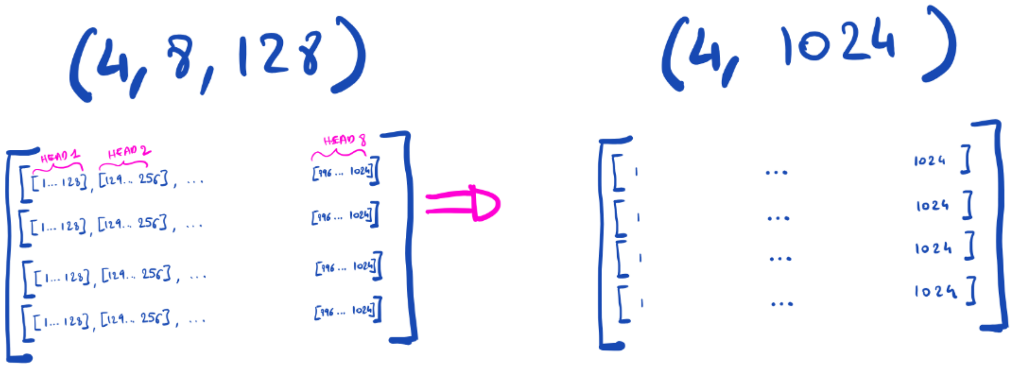
$(4, 128)$

STEP 5: TRANSPOSE BACK

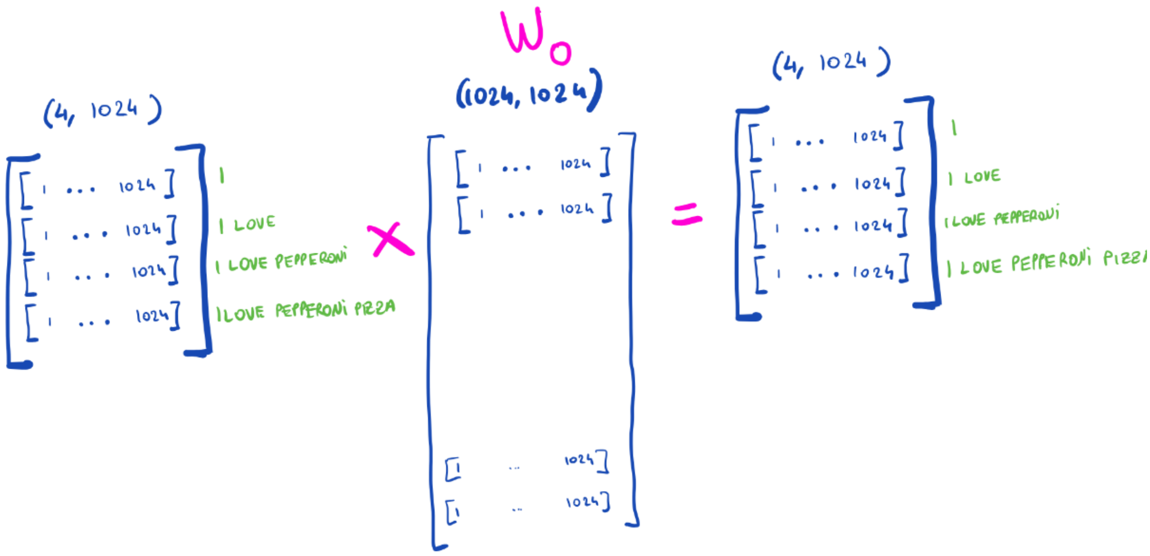


STEP 6: CONCATENATE ALL THE HEADS

Given that each head is computing the contextualized embeddings using a "part" of each token we can concatenate all the result of all the heads back together



STEP 7: MULTIPLY BY W_o




```python
class SiglipAttention(nn.Module):
    """Multi-headed attention from 'Attention Is All You Need' paper"""

    def __init__(self, config):
        super().__init__()
        self.config = config
        self.embed_dim = config.hidden_size # its define the dimension on which input data is represent.
        self.num_heads = config.num_attention_heads # its define ki how many head's in multihead attention mechanism.
        self.head_dim = self.embed_dim // self.num_heads
        self.scale = self.head_dim**-0.5 # Equivalent to 1 / sqrt(self.head_dim) this is the scale factore which use to scale the dot produ
        self.dropout = config.attention_dropout# because when we calculate the value of thier dot product so its may be too large,so that's

        self.k_proj = nn.Linear(self.embed_dim, self.embed_dim)
        self.v_proj = nn.Linear(self.embed_dim, self.embed_dim)
        self.q_proj = nn.Linear(self.embed_dim, self.embed_dim)
        self.out_proj = nn.Linear(self.embed_dim, self.embed_dim)# this is final output porjection which store the output of the attention.
# this forward function process the attention machanism.
    def forward(
        self,
        hidden_states: torch.Tensor,
    ) -> Tuple[torch.Tensor, Optional[torch.Tensor]]:

        # hidden_states: [Batch_Size, Num_Patches, Embed_Dim]
        batch_size, seq_len, _ = hidden_states.size()
        # query_states: [Batch_Size, Num_Patches, Embed_Dim]
        query_states = self.q_proj(hidden_states)
        # key_states: [Batch_Size, Num_Patches, Embed_Dim]
        key_states = self.k_proj(hidden_states)
        # value_states: [Batch_Size, Num_Patches, Embed_Dim]
        value_states = self.v_proj(hidden_states)
```

```
# value_states: [Batch_Size, Num_Patches, Embed_Dim]
value_states = self.v_proj(hidden_states)
# query_states: [Batch_Size, Num_Heads, Num_Patches, Head_Dim]
query_states = query_states.view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)

key_states = key_states.view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)

value_states = value_states.view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
# After take the transpose of all the projections it will calulate the similarity score , and then attention score.
# Calculate the attention using the formula Q * K^T / sqrt(d_k). attn_weights: [Batch_Size, Num_Heads, Num_Patches, Num_Patches]
attn_weights = (torch.matmul(query_states, key_states.transpose(2, 3)) * self.scale)

if attn_weights.size() != (batch_size, self.num_heads, seq_len, seq_len):
    raise ValueError(
        f"Attention weights should be of size {(batch_size, self.num_heads, seq_len, seq_len)}, but is"
        f" {attn_weights.size()}"
    )

# Apply the softmax row-wise. attn_weights: [Batch_Size, Num_Heads, Num_Patches, Num_Patches]
attn_weights = nn.functional.softmax(attn_weights, dim=-1, dtype=torch.float32).to(query_states.dtype)
# Apply dropout only during training
attn_weights = nn.functional.dropout(attn_weights, p=self.dropout, training=self.training)
# Multiply the attention weights by the value states. attn_output: [Batch_Size, Num_Heads, Num_Patches, Head_Dim]
attn_output = torch.matmul(attn_weights, value_states)
```

```
        if attn_output.size() != (batch_size, self.num_heads, seq_len, self.head_dim):
            raise ValueError(
                f"`attn_output` should be of size {(batch_size, self.num_heads, seq_len, self.head_dim)}, but is"
                f" {attn_output.size()}"
            )
        # [Batch_Size, Num_Heads, Num_Patches, Head_Dim] -> [Batch_Size, Num_Patches, Num_Heads, Head_Dim]
        attn_output = attn_output.transpose(1, 2).contiguous()
        # [Batch_Size, Num_Patches, Num_Heads, Head_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
        attn_output = attn_output.reshape(batch_size, seq_len, self.embed_dim)
        # [Batch_Size, Num_Patches, Embed_Dim]
        attn_output = self.out_proj(attn_output)

        return attn_output, attn_weights
#in that the final output of this mechanism are return(attn_output) and with that attn_weights are also return
# This weight define that which patches pairs get the higher attention.
```

The above we explain the Whole working of the Multi-Head attention mechanism for code point of view and theory point of view.

After generating that attn_output by the attention layer,we 1st add their output with the input which we provide as input before layer normalization. Why we add, because we two type information which are initial and after attention mechanism so its easy for MLP layer to extract the best features.

After that we again perform the layer normalization of whole hidden_state,and then provide as input to the MLP layer.

MLP(Multi-Layer Perceptron):
This is layer after the attention mechanism, Its kind of feed-forward neural network. Why we use this , because after the attention mechanism its refine the feature information as more purity or clarity, so that its useful to the other layer's. now we discuss how it work with this code.

```python
class SiglipMLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.fc1 = nn.Linear(config.hidden_size, config.intermediate_size)
        self.fc2 = nn.Linear(config.intermediate_size, config.hidden_size)
#In that 1st fc1 are the fully connected(linear) layer which tranfrom the input data.
# hidden_size which are the output of the attention layer, which we transform in intermediate_size
# which are capture the large information.
#after fc2 which are the 2nd layer of MLP, its transform the intermediate size into original size
# so that after the attention mechanism the output is compatible to the orginal format.
    def forward(self, hidden_states: torch.Tensor) -> torch.Tensor:
        # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Intermediate_Size]
        hidden_states = self.fc1(hidden_states)
        # hidden_states: [Batch_Size, Num_Patches, Intermediate_Size]
        hidden_states = nn.functional.gelu(hidden_states, approximate="tanh")
        # In that we use gelu(Gaussian error linear unit) which are introduce the non-linarity,
        # means its understand the complex pattern as well with the simple linear relationship.
        # this will calculate why the approximate forumla of tanh.
        # [Batch_Size, Num_Patches, Intermediate_Size] -> [Batch_Size, Num_Patches, Embed_Dim]
        hidden_states = self.fc2(hidden_states)

        return hidden_states
    # final hidden_states which are reprsent the orginal contexual embedding vectors of all the patches.
```

Now we Discuss the text processing:

After that we pick the text data which are provide in the input , then tokenise that text using PaliGemma tokenizer which perform the tokenizing and then convert every tokens in embedding vectors. Let see how thay do that…

```python
inputs = self.tokenizer(
            input_strings,
            return_tensors="pt",
            padding=padding,
            truncation=truncation,
        )
```

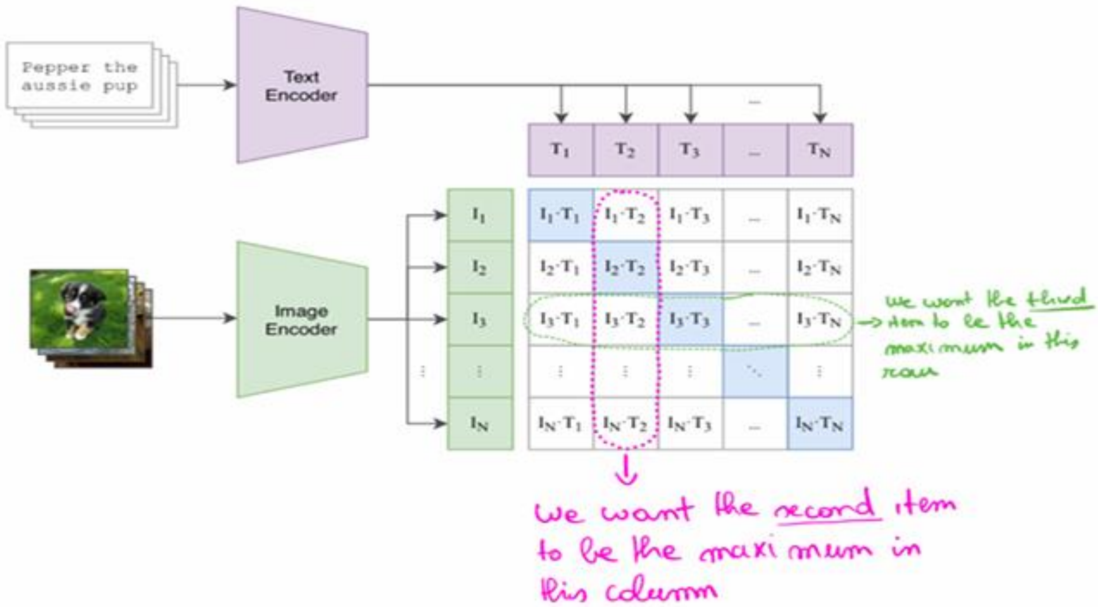
After that we apply the Contrastive pre-training(learning) approach to align the image and text embedding vector's and generate the contextual vector which provide as input to the Gemma. Let 1st we see how contrastive learning approach is work and how they align the text and image embedding vectors. But contrastive learning approach are not separately process the tokens and patches embedding vectors its process the complete image and sentence. After that we provide the image aggregate embedding vector and text aggregate embedding vector and then process for every sentence and every

image, and then generate the contextual representation. Let see how they work…

Contrastive learning :

The CLIP(Contrastive language-image Pre-Training) is to align the embedding vector of the image and text which are generate by the both encoders. So the model learn that for a specific image, what is the possible textual description. Or the learn what is the possible image for the specific text.

Let see how they work:



Problem : how do we tell the model we want one item in each row/column to be maximized while minimizing all the others?

Hint : this is very similar to language modeling in which we want a single token to be the next one given the prompt...

Solution : We use the Cross-Entropy Loss!

In the given image 1st we encode the both image and text after that we get the embedding vector which represent the feature of the text and image in vector form. After that we apply the joint multimodal embedding to both the embedding vectors.

Joint Multimodal Embedding:

I- 1st we normalize the both embedding vectors, to get the same Dimensions and size.

II- After that we calculate the cosine similarity in btw both embedding. In above case there are N images and N text sentence , in btw it check the cosine similarity.

```
# scaled pairwise cosine similarities [n, n]
logits = np.dot(I_e, T_e.T) * np.exp(t)
```
→ Compute all the possible dot products.

Logits Matrix(n x n):

1st make the score matrix for each image and each text.

Row: Image embedding, Columns: Text embedding.

In every row , score of single image is calculate with all texts. Similarly for every column score of single text calculate with other images.

Now the target of this Score matrix is that to maximise score of those pair which are correct pair.

And minimise for all which are incorrect pair.

Loss Function(Cross-Entropy Loss):

This loss function or we can say Cross-Entropy loss are used to maximise the correct pairs(Positive pair) and minimise the incorrect pairs(Negative Pair).

Now 1st we define how model decide the positive pair and negative , so as per the given labels in dataset we those are more related to the images' which are positive and those are not related to the image are negative pairs. An the target of cross entropy loss is to maximise the positive pairs and minimise the negative pairs.

Now let see how it can happen:

Let we have 2 images I1 and I2 and their corresponding texts T1 and T2. Now the score matrix is like

Similarity Scores Before Training:

Images/Text	Text 1 ("Dog")	Text 2 ("Cat")
Image 1	**0.6**	0.4
Image 2	0.3	**0.5**

Now let the correct pair is (Image 1 ,Text 1) and (Image 2, text 2) for that case similarity score is increase

Incorrect pairs:

(image 1,text 2) and (Image 2, text 1) this pair are incorrect so their similarity score are decreese.

For this we use loss function (cross entropy loss) ,let we discuss how they work.

Step 1: Normalize Scores(Softmax)

```
# t               - learned temperature parameter

# extract feature representations of each modality
I_f = image_encoder(I) #[n, d_i]
T_f = text_encoder(T)  #[n, d_t]

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)

# scaled pairwise cosine similarities [n, n]
```

→ convert a list of images into a list of embeddings

→ convert a list of prompts into a list of embeddings

Make sure both image and text embeddings have the same number of dimensions, and then normalize the vectors

In that I_f and T_f are the embedding vector of image's and text sentences respectively .

I_e and T_e is represent the normalised form of these embedding vector.

After that cross entropy loss use softmax to convert scores into probabilities.

Figure 3. Numpy-like pseudocode for the core of an implementation of CLIP.

Numerical stability of the softmax

$\forall_{i \in 1 \dots N} \quad S_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}$ The softmax makes all the elements of a vector in such a way that they're in the real range [0,1] and they sum up to 1.

Problem: the softmax is numerically unstable, as the exp function can grow fast and may not fit in a 32 bit floating-point number.

Solution: do <u>not</u> make the exp grow to infinity.

$$S_i = \frac{c \cdot e^{a_i}}{c \cdot \sum_{k=1}^{N} e^{a_k}} = \frac{e^{\log(c)} e^{a_i}}{e^{\log(c)} \sum_{k=1}^{N} e^{a_k}} = \frac{e^{a_i + \log(c)}}{\sum_{k=1}^{N} e^{a_k + \log(c)}}$$

We normally choose $\log(c) = -\max_i (a_i)$

This will push the arguments of the exp towards negative numbers and the exp itself towards zero.

In genral when we use Numerical Stability so it means we stable the our data with in a 32 bits. But the e^ai is the term which are not fit in their bound

,because it increase exponentially so it touch the infinity if ai-> infinity ,so for the numerical stability in that divide their summation of all possible value of k. so it fit within a range of 0 to 1 because softmax are convert the embedding into the probability distribution ,and probability>=0 and Sumition of Pi=1. So we chose log(c) = -max(ai).

The normalization factor in the softmax

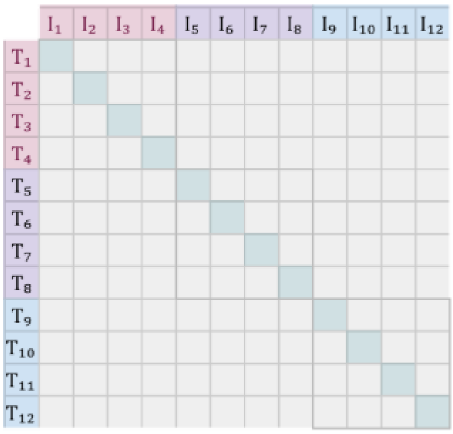
To calculate the normalization factor, we must go through all the elements of each row and each column.

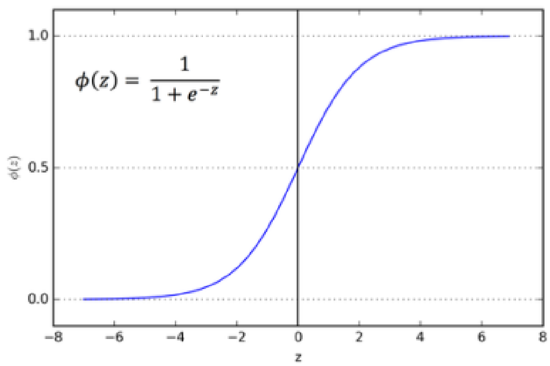
[redacted]. Note that due to the asymmetry of the softmax loss, the normalization is independently performed two times: across images and across texts [36]. [redacted]

$$-\frac{1}{2|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|}\left(\overbrace{\log\frac{e^{t\mathbf{x}_i\cdot\mathbf{y}_i}}{\sum_{j=1}^{|\mathcal{B}|}e^{t\mathbf{x}_i\cdot\mathbf{y}_j}}}^{\text{image}\rightarrow\text{text softmax}}+\overbrace{\log\frac{e^{t\mathbf{x}_i\cdot\mathbf{y}_i}}{\sum_{j=1}^{|\mathcal{B}|}e^{t\mathbf{x}_j\cdot\mathbf{y}_i}}}^{\text{text}\rightarrow\text{image softmax}}\right)$$

The solution is to use ... a Sigmoid!

$$-\frac{1}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|}\sum_{j=1}^{|\mathcal{B}|}\underbrace{\log\frac{1}{1+e^{z_{ij}(-t\mathbf{x}_i\cdot\mathbf{y}_j+b)}}}_{\mathcal{L}_{ij}}$$





Using this sigmoid function we perform the binary classification like those are correct pair we give the score 1 and those are incorrect pair give 0 for all.

After the we apply loss function the score matrix is like for above eg:

Images/Text	Text 1 ("Dog")	Text 2 ("Cat")
Image 1	**0.95**	0.05
Image 2	0.10	**0.90**

In that correct pair have increase value of the score and for incorrect pair has decrease value. At the end calculate the symmetric loss.

Symmetric Loss:

```
# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

Teach the model which item in each row/column needs to be maximized

The Symmetric loss is a variation of the **Cross-Entropy Loss**, 1^{st} we talk by why its Symmetric , because the loss is calculated in both the directions.

Image-to-text: How well an image embedding can predict its corresponding text embedding.

```
loss_i = cross_entropy_loss(logits, labels, axis=0)
```

Text-to-image: How well a text embedding can predict its corresponding image embedding.

```
loss_t = cross_entropy_loss(logits, labels, axis=1)
```

Then we take the average of both the losses to ensure symmetry.

```
loss = (loss_i + loss_t) / 2
```

Now why its used, so as we mention in above image its teach the model which item in each row/column needs to be maximise, so its help the model to find the positive pairs. So that we using the cross entropy loss we maximise the positive pairs.

So with the help of this we generate the final Contextual vector of both the Modalities, Which are define the best relationship in btw text and image. And pass this vector as a input to the Gemma:2B Language Model.

NOTE: CLIP is the technique which perform zero-shot classification , let see what it's. zero shot classification means when we give the some images and some textual labels so its easily match the labels with their corresponding without any additional training.

GEMMA:(Generalized Estimating Equation Model-based Analysis)

Let me explain the Whole process how the Gemma work in their backend. We particularly explain in our case so we provide the image and text as input ,so how they work.

Step 1-Image and text Inputs: 1st of all we processed the image and text as input to the Gemma modal.

After that it perform the text embedding to using the input_ids which we provide as input to the model. For the case of image it processed the image to pass pixels of the image through the vision_tower so that it extract the features of the image.

```
def forward(
    self,
    input_ids: torch.LongTensor = None,
    pixel_values: torch.FloatTensor = None,
    attention_mask: Optional[torch.Tensor] = None,
    kv_cache: Optional[KVCache] = None,
) -> Tuple:

    # Make sure the input is right-padded
    assert torch.all(attention_mask == 1), "The input cannot be padded"

    # 1. Extra the input embeddings
    # shape: (Batch_Size, Seq_Len, Hidden_Size)
    inputs_embeds = self.language_model.get_input_embeddings()(input_ids)

    # 2. Merge text and images
    # [Batch_Size, Channels, Height, Width] -> [Batch_Size, Num_Patches, Embed_Dim]
    selected_image_feature = self.vision_tower(pixel_values.to(inputs_embeds.dtype))
    # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Hidden_Size]
    image_features = self.multi_modal_projector(selected_image_feature)
```

Step 2- Fusion of information: In that case it merge the text and image embeddings. let see how it will do that

- Scaling Image features:
  1st of all normalize the image so that match the scale of image and text.
- Mask Creation:
  The working of mask creation is that it will analyse the type of input and define that which tokens are important or not let see how they do that
  # Shape: [Batch_Size, Seq_Len]. True for text tokens text_mask = (input_ids != self.config.image_token_index) & (input_ids != self.pad_token_id)
  This code explain that which tokens are text. In that (input_ids != self.config.image_token_index) this portion ensure that the given token are not image token.
  (input_ids != self.pad_token_id) and this portion ensure that the given tokens are not padding.
  padding tokens(little bit): The tokens are nothing , its just fixed the text and image data at standardize the length. Means when we provide the text and image as a input so it manage the length of the inputs on same scale. Because in batch processing the size of all the sequence are same.

  Now where the mask creation is used so let see.
  final_embedding = torch.where(text_mask_expanded, inputs_embeds, final_embedding) , the embedding of text tokens add in the final embedding with the help of text_mask.
  Similary for image.

- Merge creation: In that we combine the image and text using torch.where and masked_scatter.

```python
# Merge the embeddings of the text tokens and the image tokens
inputs_embeds, attention_mask, position_ids = self._merge_input_ids_with_image_features(image_features, inputs_embeds, input_ids, attention_mask, kv_cache)
```

```python
def _merge_input_ids_with_image_features(
    self, image_features: torch.Tensor, inputs_embeds: torch.Tensor, input_ids: torch.Tensor, attention_mask: torch.Tenso
):
    _, _, embed_dim = image_features.shape
    batch_size, sequence_length = input_ids.shape
    dtype, device = inputs_embeds.dtype, inputs_embeds.device
    # Shape: [Batch_Size, Seq_Len, Hidden_Size]
    scaled_image_features = image_features / (self.config.hidden_size**0.5)

    # Combine the embeddings of the image tokens, the text tokens and mask out all the padding tokens.
    final_embedding = torch.zeros(batch_size, sequence_length, embed_dim, dtype=inputs_embeds.dtype, device=inputs_embeds
    # Shape: [Batch_Size, Seq_Len]. True for text tokens
    text_mask = (input_ids != self.config.image_token_index) & (input_ids != self.pad_token_id)
    # Shape: [Batch_Size, Seq_Len]. True for image tokens
    image_mask = input_ids == self.config.image_token_index
    # Shape: [Batch_Size, Seq_Len]. True for padding tokens
    pad_mask = input_ids == self.pad_token_id

    # We need to expand the masks to the embedding dimension otherwise we can't use them in torch.where
    text_mask_expanded = text_mask.unsqueeze(-1).expand(-1, -1, embed_dim)
    pad_mask_expanded = pad_mask.unsqueeze(-1).expand(-1, -1, embed_dim)
    image_mask_expanded = image_mask.unsqueeze(-1).expand(-1, -1, embed_dim)

    # Add the text embeddings
    final_embedding = torch.where(text_mask_expanded, inputs_embeds, final_embedding)
    # Insert image embeddings. We can't use torch.where because the sequence length of scaled_image_features is not equal
    final_embedding = final_embedding.masked_scatter(image_mask_expanded, scaled_image_features)
    # Zero out padding tokens
    final_embedding = torch.where(pad_mask_expanded, torch.zeros_like(final_embedding), final_embedding)
```

Step 3- Positional Encoding:

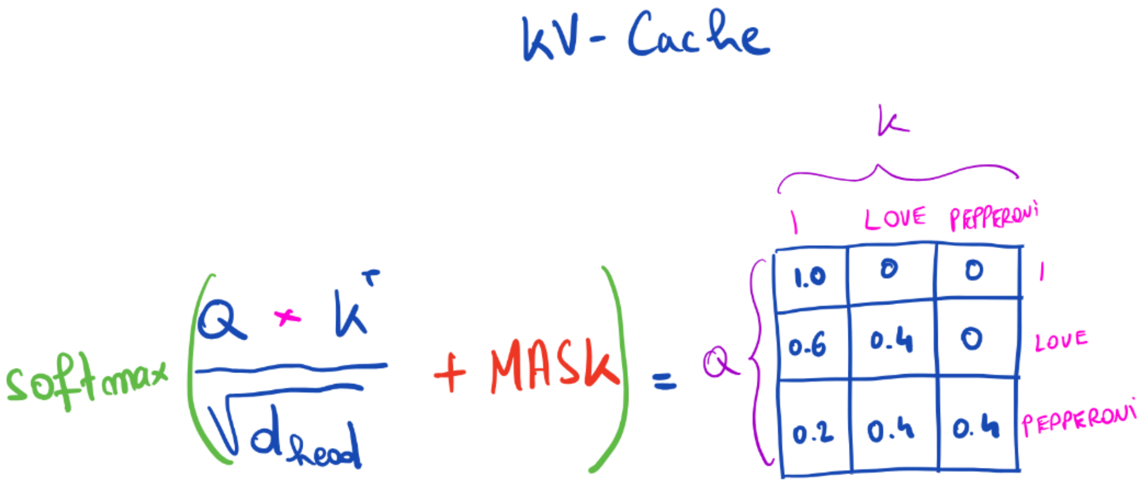
In that tokens encode the positions so that we preserve the information of the sequence.

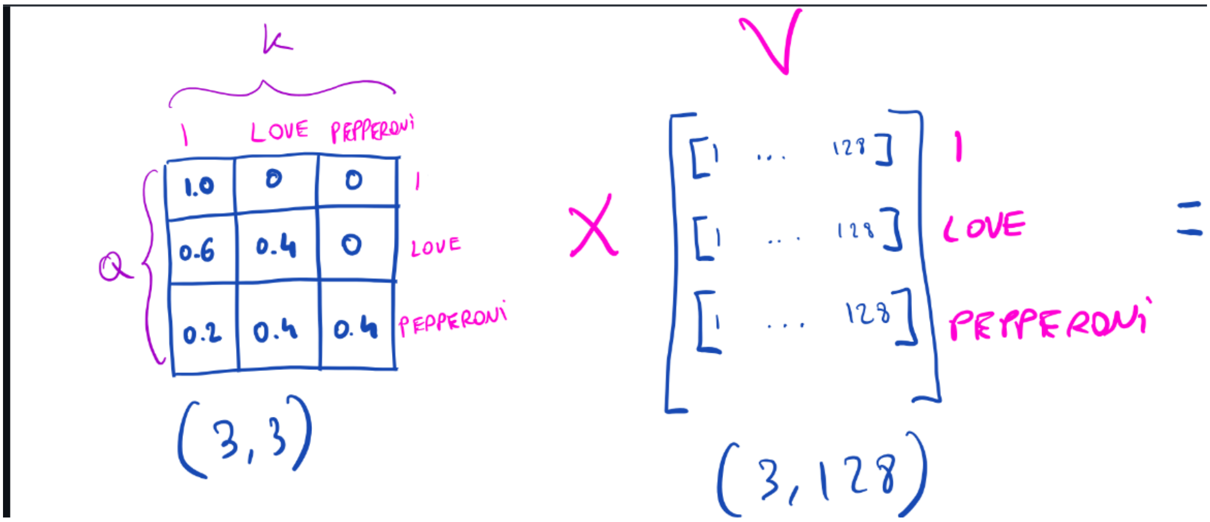
```python
class GemmaRotaryEmbedding(nn.Module):
    def __init__(self, dim, max_position_embeddings=2048, base=10000, device=None):
        super().__init__()

        self.dim = dim # it is set to the head_dim
        self.max_position_embeddings = max_position_embeddings
        self.base = base

        # Calculate the theta according to the formula theta_i = base^(-2i/dim) where i = 0, 1, 2, ..., dim // 2
        inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2, dtype=torch.int64).float() / self.dim))
        self.register_buffer("inv_freq", tensor=inv_freq, persistent=False)

    @torch.no_grad()
    def forward(self, x, position_ids, seq_len=None):
        # x: [bs, num_attention_heads, seq_len, head_size]
        self.inv_freq.to(x.device)
        # Copy the inv_freq tensor for batch in the sequence
        # inv_freq_expanded: [Batch_Size, Head_Dim // 2, 1]
        inv_freq_expanded = self.inv_freq[None, :, None].float().expand(position_ids.shape[0], -1, 1)
        # position_ids_expanded: [Batch_Size, 1, Seq_Len]
        position_ids_expanded = position_ids[:, None, :].float()
        device_type = x.device.type
        device_type = device_type if isinstance(device_type, str) and device_type != "mps" else "cpu"
        with torch.autocast(device_type=device_type, enabled=False):
            # Multiply each theta by the position (which is the argument of the sin and cos functions)
            # freqs: [Batch_Size, Head_Dim // 2, 1] @ [Batch_Size, 1, Seq_Len] --> [Batch_Size, Seq_Len, Head_Dim /
            freqs = (inv_freq_expanded.float() @ position_ids_expanded.float()).transpose(1, 2)
            # emb: [Batch_Size, Seq_Len, Head_Dim]
            # cos, sin: [Batch_Size, Seq_Len, Head_Dim]
            cos = emb.cos()
            sin = emb.sin()
        return cos.to(dtype=x.dtype), sin.to(dtype=x.dtype)


def rotate_half(x):
    # Build the [-x2, x1, -x4, x3, ...] tensor for the sin part of the positional encoding.
    x1 = x[..., : x.shape[-1] // 2] # Takes the first half of the last dimension
    x2 = x[..., x.shape[-1] // 2 :] # Takes the second half of the last dimension
    return torch.cat((-x2, x1), dim=-1)


def apply_rotary_pos_emb(q, k, cos, sin, unsqueeze_dim=1):
    cos = cos.unsqueeze(unsqueeze_dim) # Add the head dimension
    sin = sin.unsqueeze(unsqueeze_dim) # Add the head dimension
    # Apply the formula (34) of the Rotary Positional Encoding paper.
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed
```

This is combined embedding vector of the text and image feature with the positional encoding we provide to the gemma decoder as a input.

After that Gemma decoder is process.

Step-4: KV Cache: Key-value pair cache are used to store the input which provide by the user like "What is colour of " so to prediction of next possible required the previous whole context so this context store in KV cache in key, value pairs.





```python
class KVCache():

    def __init__(self) -> None:
        self.key_cache: List[torch.Tensor] = []
        self.value_cache: List[torch.Tensor] = []

    def num_items(self) -> int:
        if len(self.key_cache) == 0:
            return 0
        else:
            # The shape of the key_cache is [Batch_Size, Num_Heads_KV, Seq_Len, Head_Dim]
            return self.key_cache[0].shape[-2]

    def update(
        self,
        key_states: torch.Tensor,
        value_states: torch.Tensor,
        layer_idx: int,
    ) -> Tuple[torch.Tensor, torch.Tensor]:
        if len(self.key_cache) <= layer_idx:
            # If we never added anything to the KV-Cache of this layer, let's create it.
            self.key_cache.append(key_states)
            self.value_cache.append(value_states)
        else:
            # ... otherwise we concatenate the new keys with the existing ones.
            # each tensor has shape: [Batch_Size, Num_Heads_KV, Seq_Len, Head_Dim]
            self.key_cache[layer_idx] = torch.cat([self.key_cache[layer_idx], key_states], dim=-2)
            self.value_cache[layer_idx] = torch.cat([self.value_cache[layer_idx], value_states], dim=-2)

        # ... and then we return all the existing keys + the new ones.
        return self.key_cache[layer_idx], self.value_cache[layer_idx]
```

Step 4- Attention Mechanism:

This working we define above , but it target to calculate the attention weights and focus on relevant features

```python
class GemmaAttention(nn.Module):

    def __init__(self, config: GemmaConfig, layer_idx: Optional[int] = None):
        super().__init__()
        self.config = config
        self.layer_idx = layer_idx

        self.attention_dropout = config.attention_dropout
        self.hidden_size = config.hidden_size
        self.num_heads = config.num_attention_heads
        self.head_dim = config.head_dim
        self.num_key_value_heads = config.num_key_value_heads
        self.num_key_value_groups = self.num_heads // self.num_key_value_heads
        self.max_position_embeddings = config.max_position_embeddings
        self.rope_theta = config.rope_theta
        self.is_causal = True

        assert self.hidden_size % self.num_heads == 0

        self.q_proj = nn.Linear(self.hidden_size, self.num_heads * self.head_dim, bias=config.attention_bias)
        self.k_proj = nn.Linear(self.hidden_size, self.num_key_value_heads * self.head_dim, bias=config.attention_bias)
        self.v_proj = nn.Linear(self.hidden_size, self.num_key_value_heads * self.head_dim, bias=config.attention_bias)
        self.o_proj = nn.Linear(self.num_heads * self.head_dim, self.hidden_size, bias=config.attention_bias)
        self.rotary_emb = GemmaRotaryEmbedding(
            self.head_dim,
            max_position_embeddings=self.max_position_embeddings,
            base=self.rope_theta,
        )
```

```python
def forward(
    self,
    hidden_states: torch.Tensor,
    attention_mask: Optional[torch.Tensor] = None,
    position_ids: Optional[torch.LongTensor] = None,
    kv_cache: Optional[KVCache] = None,
    **kwargs,
) -> Tuple[torch.Tensor, Optional[torch.Tensor], Optional[Tuple[torch.Tensor]]]:
    bsz, q_len, _ = hidden_states.size() # [Batch_Size, Seq_Len, Hidden_Size]
    # [Batch_Size, Seq_Len, Num_Heads_Q * Head_Dim]
    query_states = self.q_proj(hidden_states)
    # [Batch_Size, Seq_Len, Num_Heads_KV * Head_Dim]
    key_states = self.k_proj(hidden_states)
    # [Batch_Size, Seq_Len, Num_Heads_KV * Head_Dim]
    value_states = self.v_proj(hidden_states)
    # [Batch_Size, Num_Heads_Q, Seq_Len, Head_Dim]
    query_states = query_states.view(bsz, q_len, self.num_heads, self.head_dim).transpose(1, 2)
    # [Batch_Size, Num_Heads_KV, Seq_Len, Head_Dim]
    key_states = key_states.view(bsz, q_len, self.num_key_value_heads, self.head_dim).transpose(1, 2)
    # [Batch_Size, Num_Heads_KV, Seq_Len, Head_Dim]
    value_states = value_states.view(bsz, q_len, self.num_key_value_heads, self.head_dim).transpose(1, 2)

    # [Batch_Size, Seq_Len, Head_Dim], [Batch_Size, Seq_Len, Head_Dim]
    cos, sin = self.rotary_emb(value_states, position_ids, seq_len=None)
    # [Batch_Size, Num_Heads_Q, Seq_Len, Head_Dim], [Batch_Size, Num_Heads_KV, Seq_Len, Head_Dim]
    query_states, key_states = apply_rotary_pos_emb(query_states, key_states, cos, sin)
```

```python
if kv_cache is not None:
    key_states, value_states = kv_cache.update(key_states, value_states, self.layer_idx)

# Repeat the key and values to match the number of heads of the query
key_states = repeat_kv(key_states, self.num_key_value_groups)
value_states = repeat_kv(value_states, self.num_key_value_groups)
# Perform the calculation as usual, Q * K^T / sqrt(head_dim). Shape: [Batch_Size, Num_Heads_Q, Seq_Len_Q, Seq_Len
attn_weights = torch.matmul(query_states, key_states.transpose(2, 3)) / math.sqrt(self.head_dim)

assert attention_mask is not None
attn_weights = attn_weights + attention_mask

# Apply the softmax
# [Batch_Size, Num_Heads_Q, Seq_Len_Q, Seq_Len_KV]
attn_weights = nn.functional.softmax(attn_weights, dim=-1, dtype=torch.float32).to(query_states.dtype)
# Apply the dropout
attn_weights = nn.functional.dropout(attn_weights, p=self.attention_dropout, training=self.training)
# Multiply by the values. [Batch_Size, Num_Heads_Q, Seq_Len_Q, Seq_Len_KV] x [Batch_Size, Num_Heads_KV, Seq_Len_K
attn_output = torch.matmul(attn_weights, value_states)

if attn_output.size() != (bsz, self.num_heads, q_len, self.head_dim):
    raise ValueError(
        f"`attn_output` should be of size {(bsz, self.num_heads, q_len, self.head_dim)}, but is"
        f" {attn_output.size()}"
    )
# Make sure the sequence length is the second dimension. # [Batch_Size, Num_Heads_Q, Seq_Len_Q, Head_Dim] -> [Bat
attn_output = attn_output.transpose(1, 2).contiguous()
# Concatenate all the heads together. [Batch_Size, Seq_Len_Q, Num_Heads_Q, Head_Dim] -> [Batch_Size, Seq_Len_Q, N
attn_output = attn_output.view(bsz, q_len, -1)
# Multiply by W_o. [Batch_Size, Seq_Len_Q, Hidden_Size]
attn_output = self.o_proj(attn_output)

return attn_output, attn_weights
```

Its generate the attn_output and attn-weight which define that which image text pair are higher atten_score.

Step-6: Transformer layers: This layer refine the output through the multiple layers of the transformer , as per [2407.07726] PaliGemma: A versatile 3B VLM for transfer this research paper it was use 6 decoding layer in transformer decoder.

```python
class GemmaDecoderLayer(nn.Module):

    def __init__(self, config: GemmaConfig, layer_idx: int):
        super().__init__()
        self.hidden_size = config.hidden_size

        self.self_attn = GemmaAttention(config=config, layer_idx=layer_idx)

        self.mlp = GemmaMLP(config)
        self.input_layernorm = GemmaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)
        self.post_attention_layernorm = GemmaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)

    def forward(
        self,
        hidden_states: torch.Tensor,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        kv_cache: Optional[KVCache] = None,
    ) -> Tuple[torch.FloatTensor, Optional[Tuple[torch.FloatTensor, torch.FloatTensor]]]:
        residual = hidden_states
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = self.input_layernorm(hidden_states)

        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states, _, = self.self_attn(
            hidden_states=hidden_states,
            attention_mask=attention_mask,
            position_ids=position_ids,
            kv_cache=kv_cache,
        )
        hidden_states = residual + hidden_states

        # [Batch_Size, Seq_Len, Hidden_Size]
        residual = hidden_states
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = self.post_attention_layernorm(hidden_states)
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = self.mlp(hidden_states)
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = residual + hidden_states

        return hidden_states
```

Now in that the function like GemmaMLP(multi-layer perceptron) which are the feed-forward network, which are used after the attention mechanism of the transformer to learn the complex relationship in btw text and image, and transform the data non-linear form means, because when we learn the linear model so its may chance of overfiting , because model not understand the complex relationship. So when we use the activation function (GELU) to the linear ouput so this output transform to the non-linear outpu00t. And in the Gemma model we use RMS normalization for providing the stability and faster convergence to the to model.

Step -7: Final output of the model:
So after pass this contextual vector to all the 6 decoder layer it will give the best contextual output at the end of 6th layer. And this final contextual vector which are generated by the gemma decoder is pass to the language model,
Final logits are generated through the lm_head, which are used for text generation.

```
outputs = self.language_model(
    attention_mask=attention_mask,
    position_ids=position_ids,
    inputs_embeds=inputs_embeds,
    kv_cache=kv_cache,
)

return outputs
```