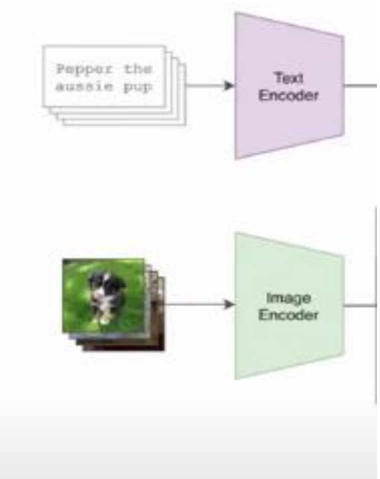
1ST of all we design the **PaliGemma Vision language model** so in that model target to generate the final contexiual text on the base of images and texts as input.

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.



Text Encoder:
In that encoder ,it follow the architecture of transformer encoder.
Transformer encoder: In a transformer encoder have 6 encoders(Reference: 1706.03762) are used, in that initially break the complete sentence in form of tokens, after that its 3 components is find Query vector ,key vector and value vector. Where we calculate the attention score using the dot product of the query vector and corresponding key vector. Which define that which text is more important with the given query vector. Initially passing through the embedding layer which generate the embedding vector , after that this embedding vector pass through the self attention mechanism so after that it will generate the contexiual vector, and this vector define that ,how the one token is related to others and so on.

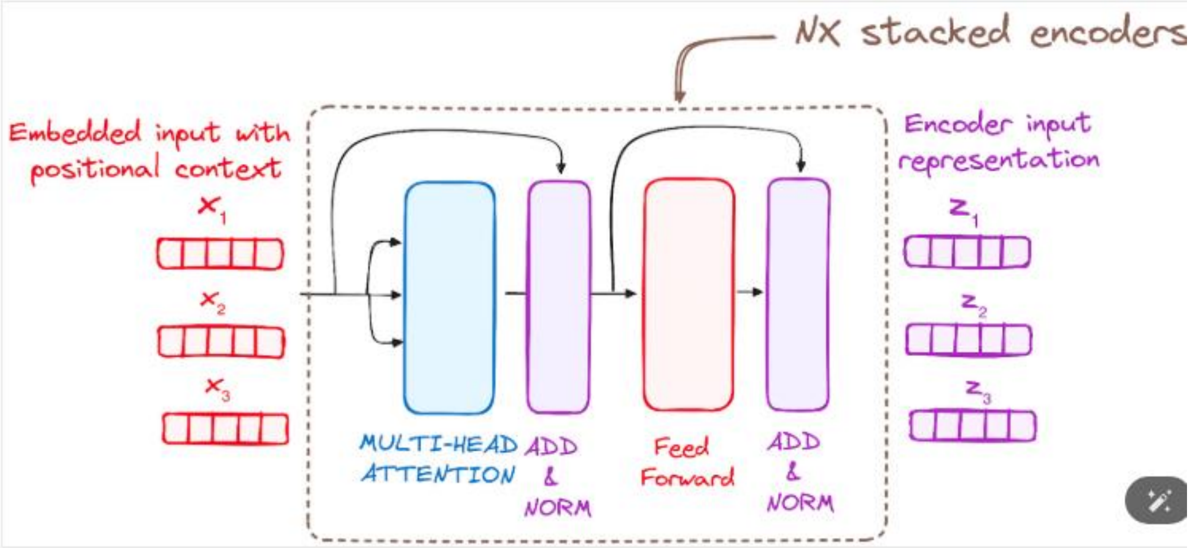


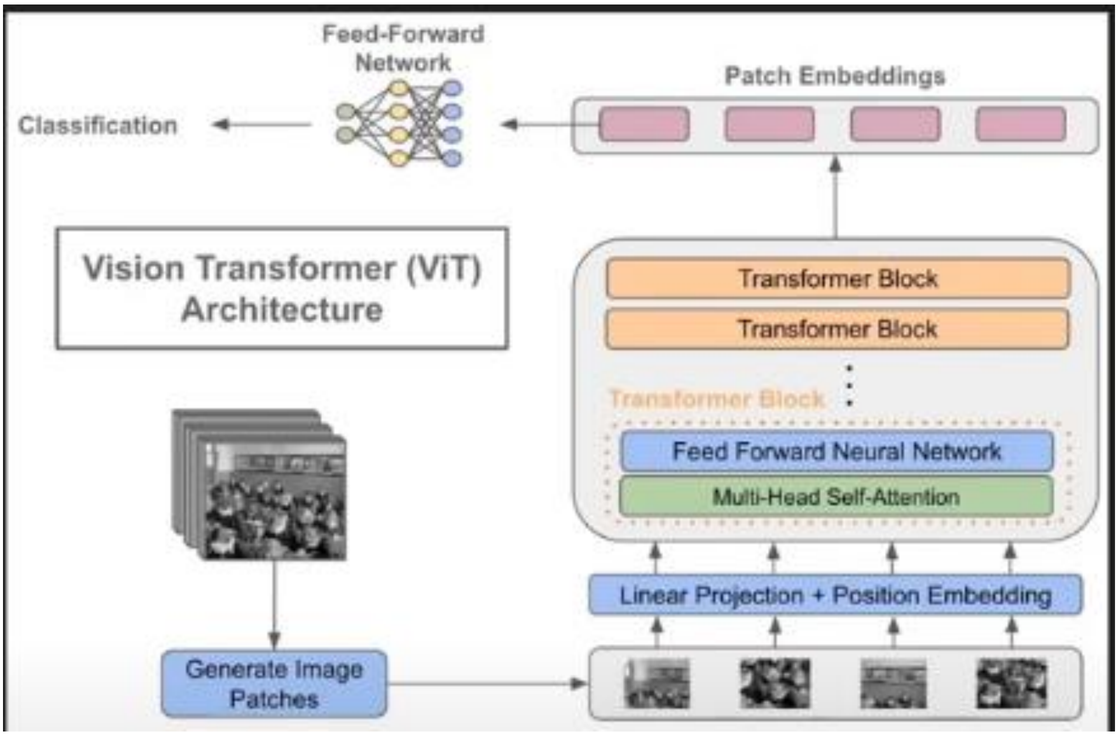
Image Encode:
This encoder to perform the encoding using Vision transformer, how they work or we also used the ResNet, but we prefer transformer over the ResNet because transformer work on the large dataset or complex dataset efficiently as compare to ResNet.

In the case we use SigLIP vision encoder,

SigLIP(Sign Language Interpretation and Processing) Vision encoder: This encoder are used for Visual Data,like image's ,video.now if we talk about ,how they work for Image and text , so for that its 1st understood the semantic relationship in between image and text, like how much similarity in btw text and image or what is the contextual relation in btw them.

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 contextual vector. 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, now the patches like:-

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). So we multiply the every parameter by 3.
Note- The value of R,G,B is define the intensity of that channel is that field.

Step-2: After that convert this patches in 1D form , means their vector represention 1D me hogi and its embedding vectors for every patch.

Step-3: Linear Projection:
Now the target 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.

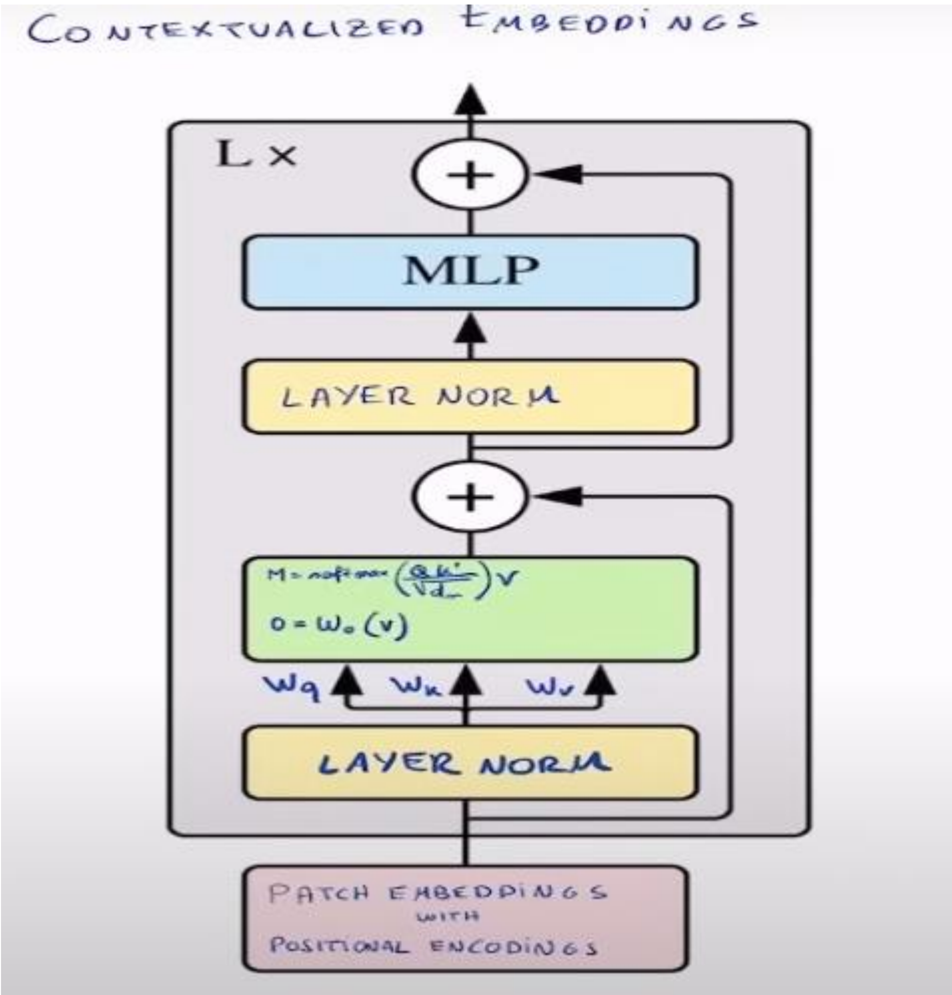
Now the benefit to reduce the size of pixels is that.
I-We have required a less resource, like memory to store the pixels, let we reduce their size 768 to 512.

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. if it change the patches order on which location of original image,so that the final output is inconsistent. So that why we add the position of the patch with their embedding information. After that we give as a input to the encoder.

After that it will used Transformer encoder which perform the feature extraction. Like complex pattern and context.

Architecture of Transformer encoder:



In that Embedded patches represent the input which we provide in it like 196 patches wit 512 pixels per patch with add on the position embedding.

After that In apply Multi-Head attention Mechanism.

Mutli-Head attention:

Multi-Head Attention

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

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 uncontextualized embeddings into a sequence of contextualized embeddings.

VISION TRANSFORMER

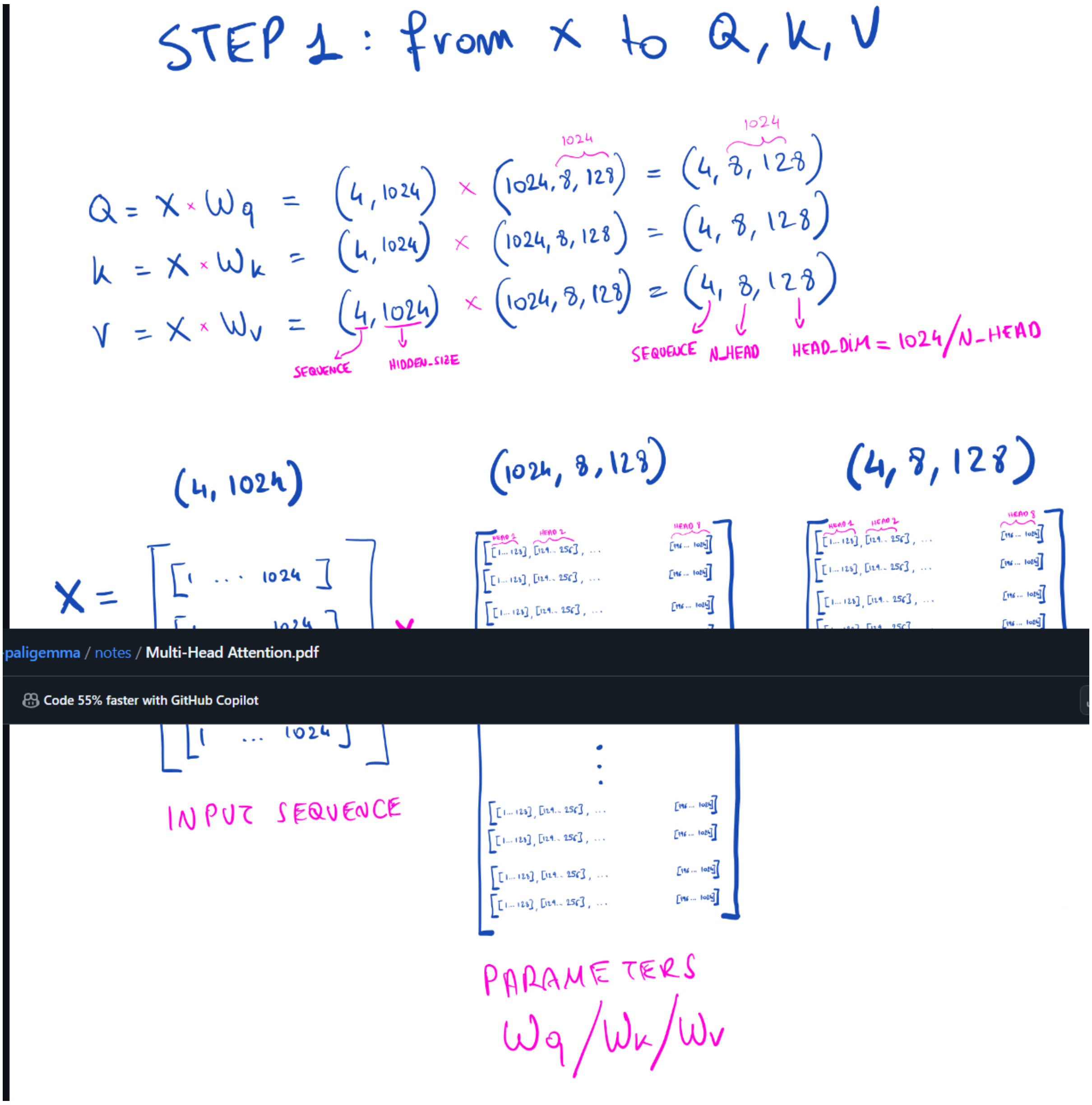
$$X = \begin{array}{c} \begin{bmatrix} [1 & \dots & 1024] \\ [1 & \dots & 1024] \\ [1 & \dots & 1024] \\ [1 & \dots & 1024] \end{bmatrix} \end{array} \begin{array}{l} \text{PATCH 1} \\ \text{PATCH 2} \\ \text{PATCH 3} \\ \text{PATCH 4} \end{array} \Rightarrow X = \begin{array}{c} \begin{bmatrix} [1 &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

paligemma / notes / Multi-Head Attention.pdf

Code 55% faster with GitHub Copilot

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

In that “items” represent the patches and their vector whose dimension is 1024 means every patch have 1024 pixels.



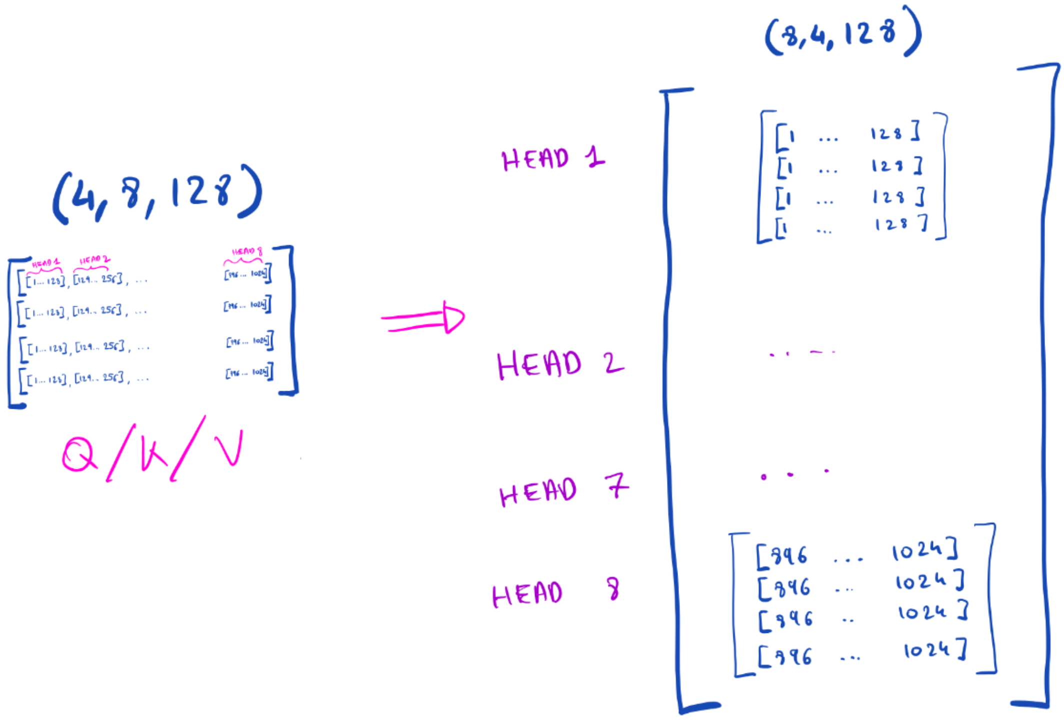
STEP 2: TREAT EACH HEAD INDEPENDENTLY:

Q : (4, 8, 128) $\xrightarrow{\text{TRANSPOSE}}$ (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.



paligemma / notes / **Multi-Head Attention.pdf**

Code 55% faster with GitHub Copilot

1) We want to parallelize the computation

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

STEP 3: CALCULATE THE ATTENTION FOR EACH HEAD IN PARALLEL

$$Q_{HEAD_1} = \underbrace{\begin{bmatrix} [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \end{bmatrix}}_{(4,128)}$$

→ dimension 1...128 of token 1
→ dimension 1...128 of token 2
→ dimension 1...128 of token 4

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

dimension 1...128 of token 1 — dimension 1...128 of token 4

$$\frac{Q \times K^T}{\sqrt{d_{head}}} = Q\left\{ \overbrace{\begin{array}{cccc|l} 13.9 & 21.1 & -100.3 & 17.5 & \text{I} \\ -5.0 & 3.14 & 1.2 & 75.3 & \text{LOVE} \\ \dots & \dots & \dots & \dots & \text{PEPPERONI} \\ \dots & \dots & \dots & \dots & \text{PIZZA} \\ \hline \text{I} & \text{LOVE} & \text{PEPPERONI} & \text{PIZZA} & \end{array}}^{K} \right.$$

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_{head}}}\right) = Q\left\{ \overbrace{\begin{array}{cccc|l} 0.1 & 0.2 & 0.5 & 0.3 & \text{I} \\ 0.4 & 0.1 & 0.3 & 0.2 & \text{LOVE} \\ \dots & \dots & \dots & \dots & \text{PEPPERONI} \\ \dots & \dots & \dots & \dots & \text{PIZZA} \\ \hline \text{I} & \text{LOVE} & \text{PEPPERONI} & \text{PIZZA} & \end{array}}^{K} \right.$$

BRO, WHERE IS YOUR MASK?

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_{head}}} + \text{MASK}\right) = Q\left\{ \overbrace{\begin{array}{cccc|l} 1.0 & 0 & 0 & 0 & \text{I} \\ 0.6 & 0.4 & 0 & 0 & \text{LOVE} \\ 0.2 & 0.4 & 0.4 & 0 & \text{PEPPERONI} \\ 0.4 & 0.2 & 0.3 & 0.1 & \text{PIZZA} \\ \hline \text{I} & \text{LOVE} & \text{PEPPERONI} & \text{PIZZA} & \end{array}}^{K} \right.$$

STEP 4: MULTIPLY BY THE V SEQUENCE

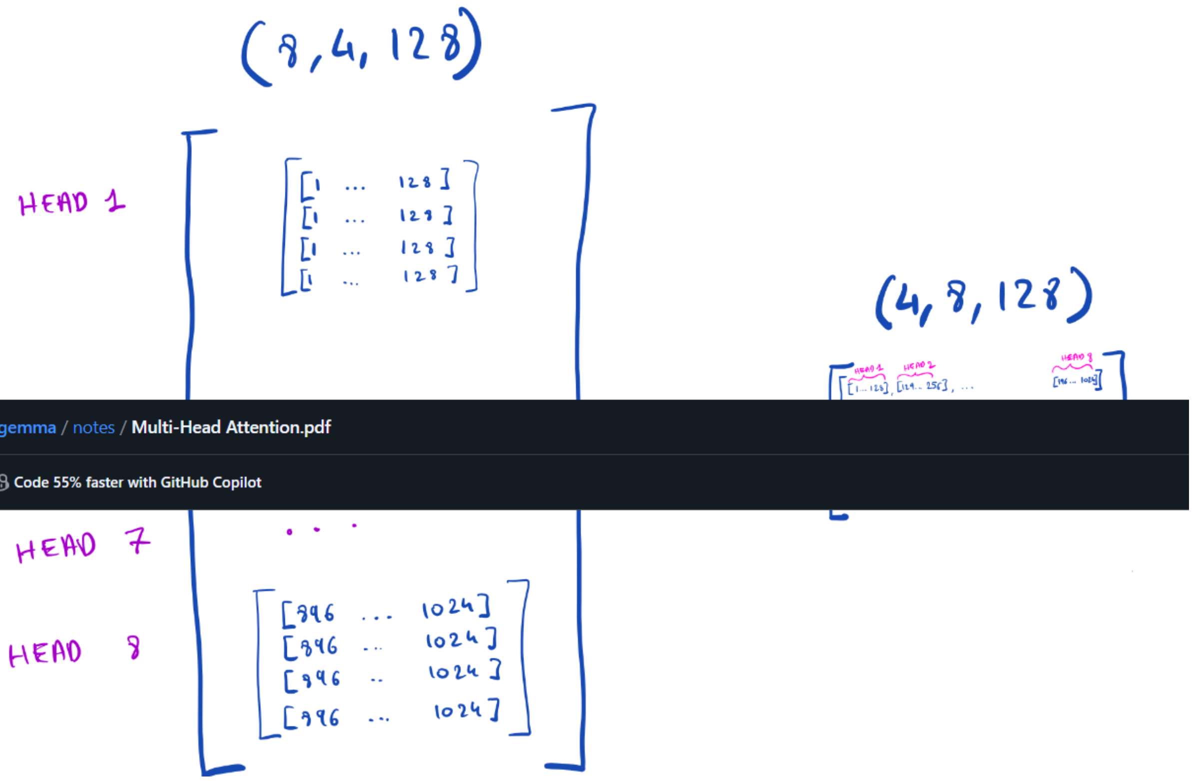
$$
\begin{array}{c}
 \overbrace{\hphantom{1.0 \quad 0 \quad 0 \quad 0}}^{K} \\
Q\left\{ \begin{array}{|c|c|c|c|}
\hline
1.0 & 0 & 0 & 0 \\
\hline
0.6 & 0.4 & 0 & 0 \\
\hline
0.2 & 0.4 & 0.4 & 0 \\
\hline
0.4 & 0.2 & 0.3 & 0.1 \\
\hline
\end{array} \right.
\end{array}
\begin{array}{l}
\text{I} \\ \text{LOVE}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

EACH ROW REPRESENTS A WEIGHTED SUM OF:

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

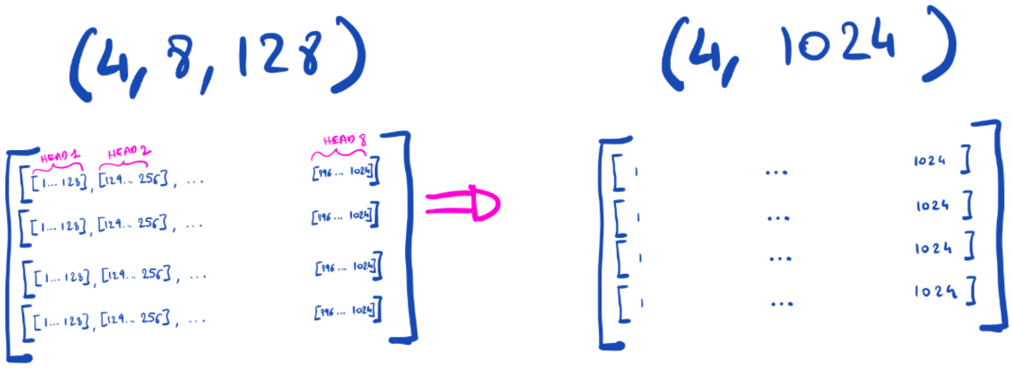
$(4, 128)$

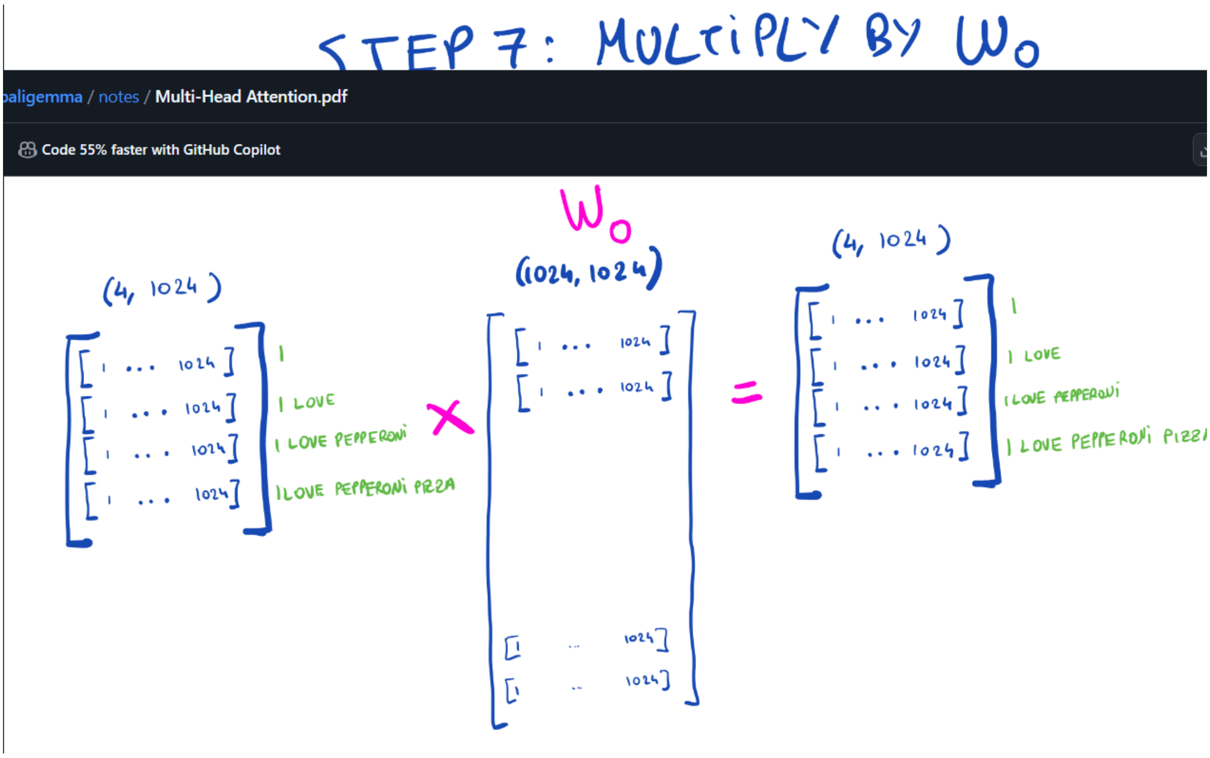
STEP 5: TRANSPOSE BACK



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



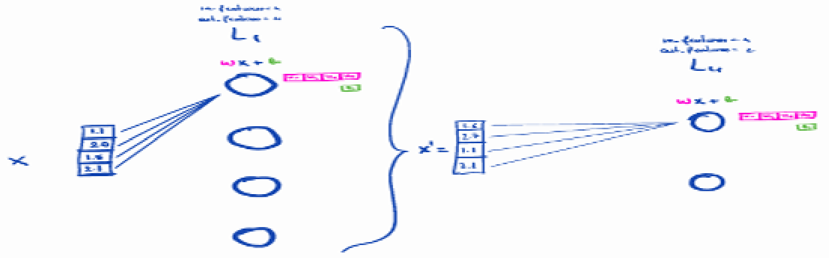


After the this mechanism we add the results of the Multi-Head Attention, which define that which patch have give the more attention to the another patch on the bases of attention score which are calculate after finding the similarity score of all with respect to the every head. And the original result which we provide as a input of the encoder.

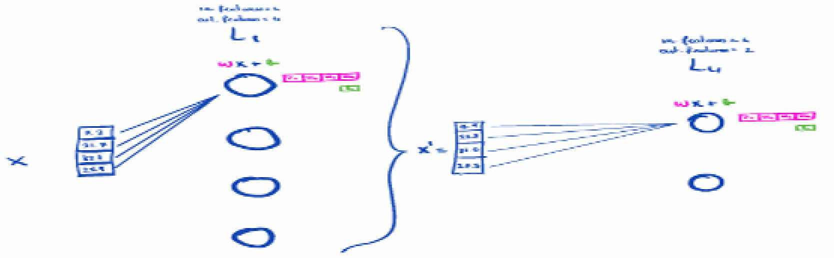
Normalization:

We check the different different type of normalization and their drawback:

Normalization 101



Problem: covariate shift



Why is it bad?

Big change in input of a layer ⇒ Big chang in output of a layer ⇒ Big chang in loss ⇒

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

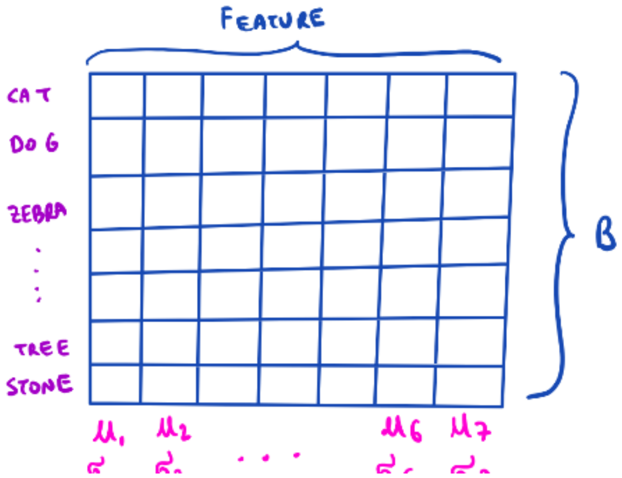
⇒ **Network learns slowly!**

[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

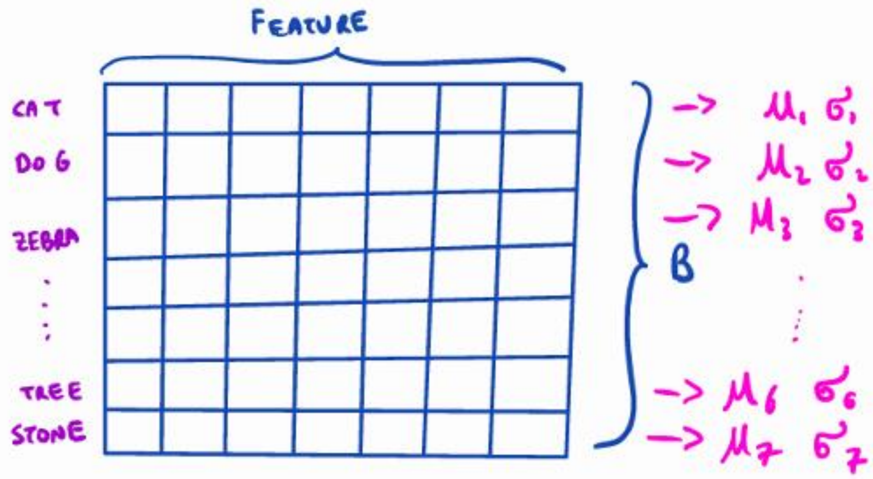
The problem with Batch Norm is that each statistic depends on what other items are in the batch. To get good results, we must use a big batch size

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:

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



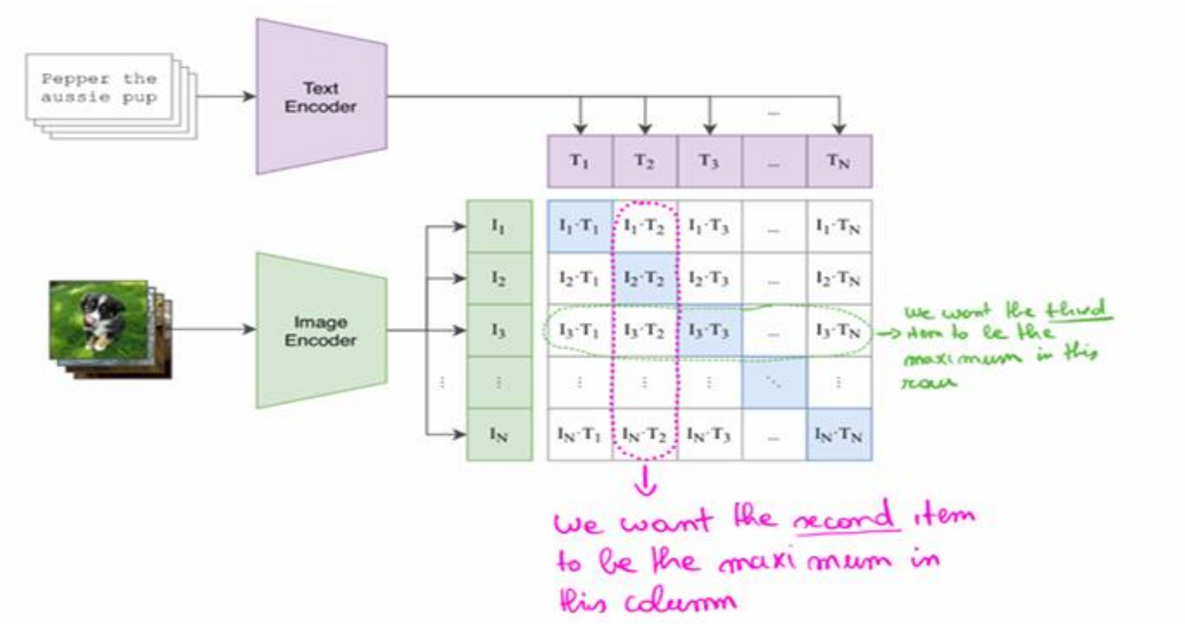
In this core each item is treated mdependently.

MLP(Multi-Layer Perceptron):

Contrastive Language-Image Pre-training(CLIP):

The CLIP is to align the embedding vectors 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.

Join 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 is are N images and N text sentence , in btw it check the cosine similarity.

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 and minimise the incorrect 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] -> convert a list of images into a list of embeddings
T_f = text_encoder(T)  #[n, d_t] -> convert a list of prompts into a list of embeddings

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1) } Make sure both image and text
T_e = l2_normalize(np.dot(T_f, W_t), axis=1) } embeddings have the same number
                                               of dimensions, and then normalize
# scaled pairwise cosine similarities [n, n]     the vectors
```

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

I_e and T_e is represent the normalised form of the 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.

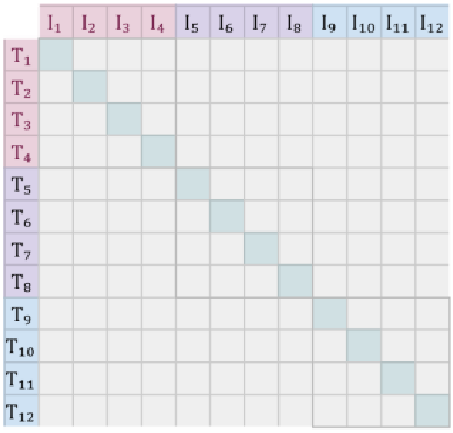
The normalization factor in the softmax

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

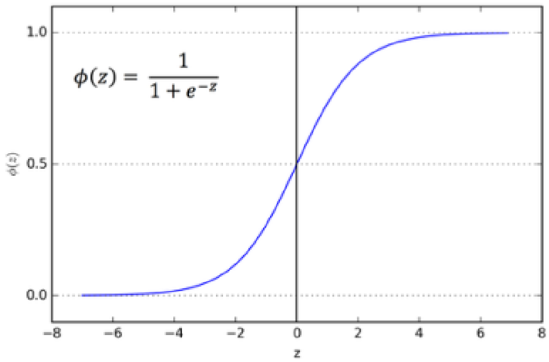
[redacted]. Note that due to the asymmetry of the softmax loss, the normalization is independently performed two times: across images and across texts [36]. [redacted]

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

The solution is to use ... a Sigmoid !



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



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:

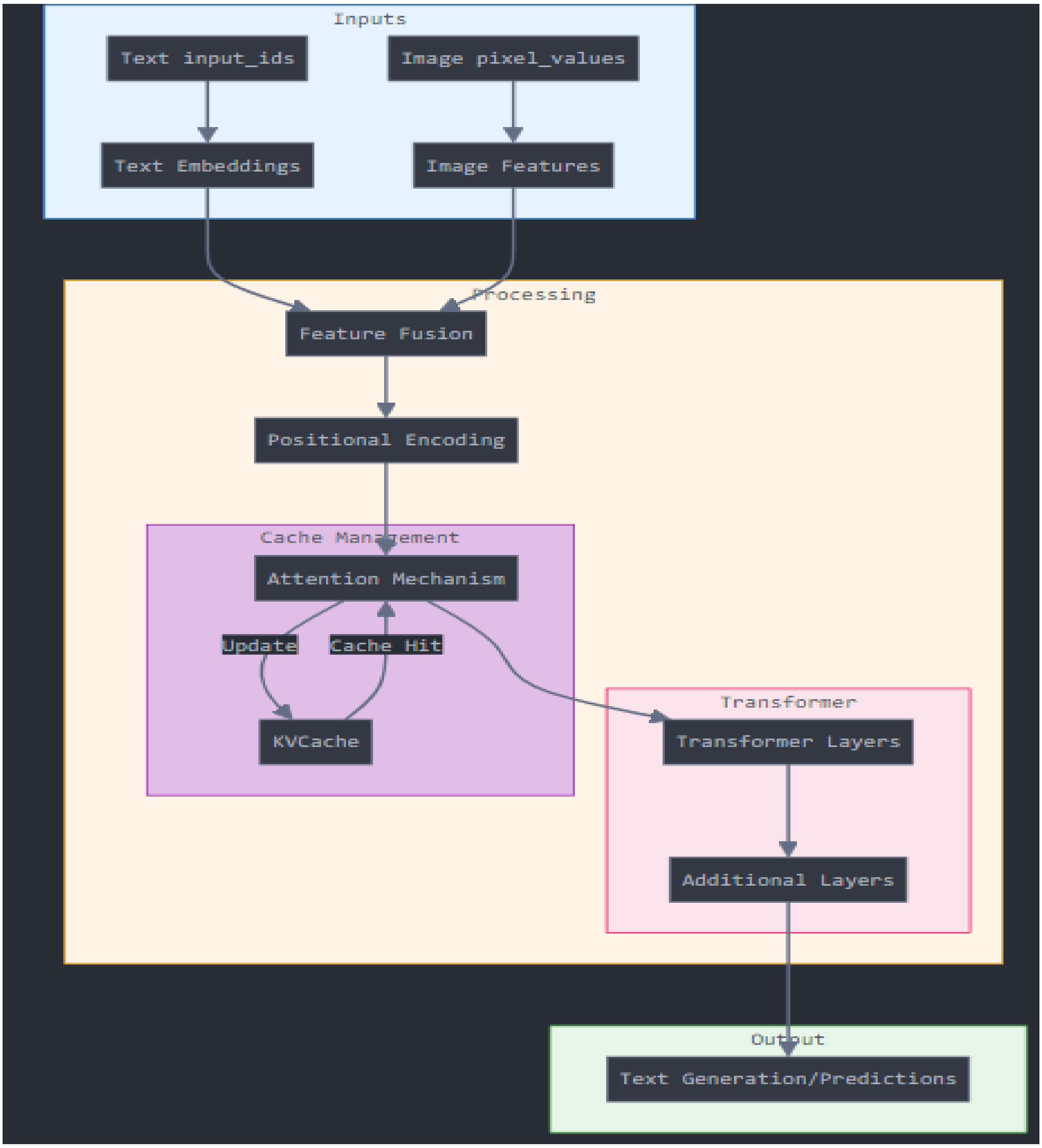
```
# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

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

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:

Let me explain the Whole process how the Gemma work in their backend by flow chart, then explain all.



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.

```
# 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)
```

```
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
```

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.

KV- Cache

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_{head}}} + \text{MASK}\right) = Q \begin{array}{c} \overbrace{\begin{array}{ccc} I & LOVE & PEPPERONI \end{array}}^{K} \\ \begin{bmatrix} 1.0 & 0 & 0 \\ 0.6 & 0.4 & 0 \\ 0.2 & 0.4 & 0.4 \end{bmatrix} \end{array} \begin{array}{l} I \\ LOVE \\ PEPPERONI \end{array}$$

$$Q \begin{array}{c} \overbrace{\begin{array}{ccc} I & LOVE & PEPPERONI \end{array}}^{K} \\ \begin{bmatrix} 1.0 & 0 & 0 \\ 0.6 & 0.4 & 0 \\ 0.2 & 0.4 & 0.4 \end{bmatrix} \end{array} \begin{array}{l} I \\ LOVE \\ PEPPERONI \end{array} \times \overset{V}{\begin{bmatrix} [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \end{bmatrix}} \begin{array}{l} I \\ LOVE \\ PEPPERONI \end{array} =$$

$$(3,3) \qquad\qquad (3,128)$$

$$= \begin{bmatrix} [1 & \dots & 128] \\ [1 & \dots & 128] \\ [1 & \dots & 128] \end{bmatrix} \left.\begin{array}{l} I \\ I\ LOVE \\ I\ LOVE\ PEPPERONI \end{array}\right\} \text{CONTEXTUALIZED EMBEDDINGS}$$

$$(3,128)$$

```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
```

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,
        )
        # [Batch_Size, Seq_Len, Hidden_Size]
        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
```

Step -7: Final output of the model:

Pass the combined embedding through 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
```