‘All’ you need to know about transformer

Victoria Zhang

Match 13 2023

- Background
- Q, K, V and attention
- Code Snippets
- Visualization of attention maps
- GPT

Background

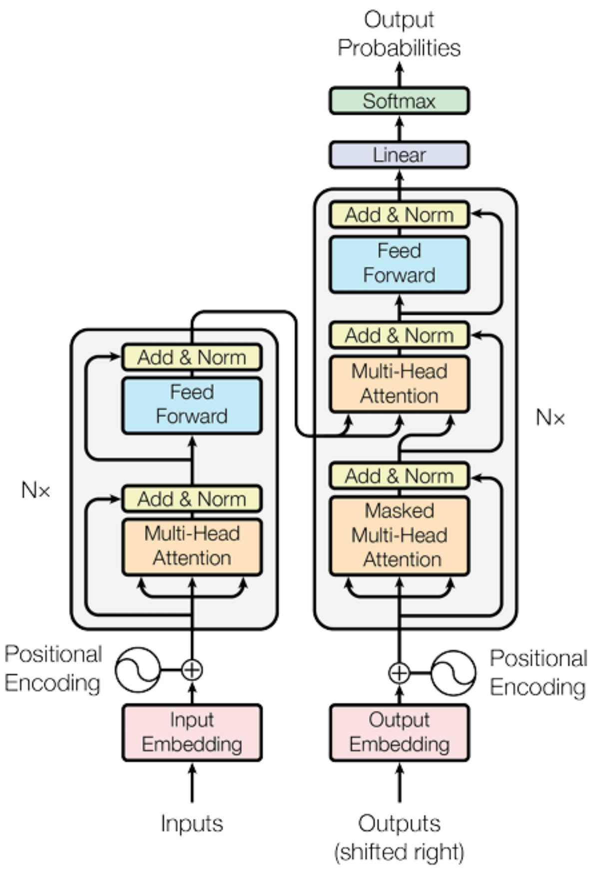
Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

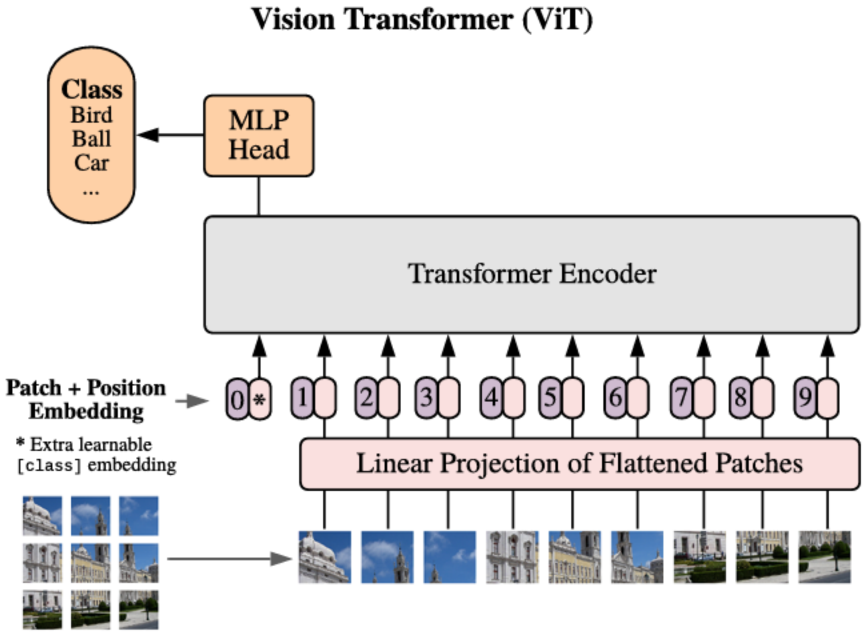
AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE

Alexey Dosovitskiy*,†, Lucas Beyer*, Alexander Kolesnikov*, Dirk Weissenborn*, Xiaohua Zhai*, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby*,†
*equal technical contribution, †equal advising
Google Research, Brain Team
{adosovitskiy, neilhoulsby}@google.com

Background

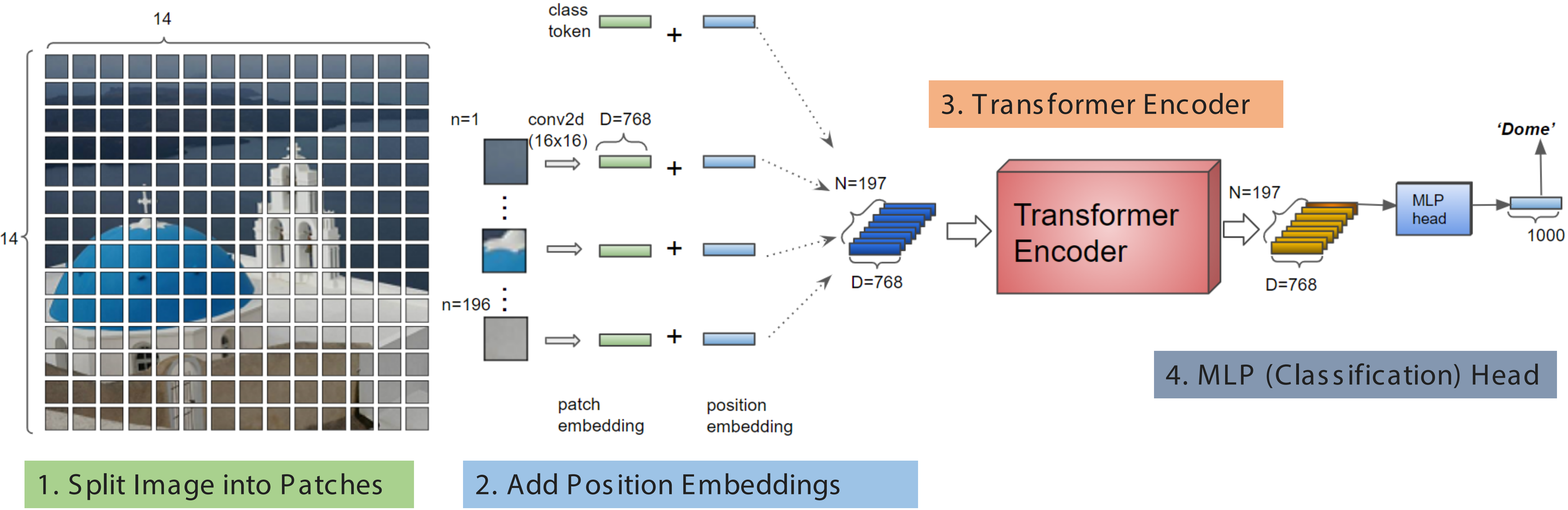


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

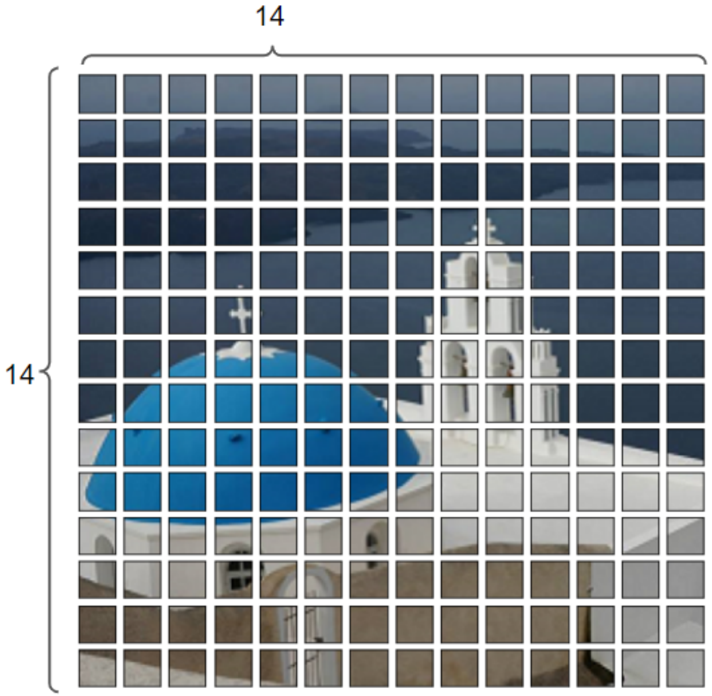


Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0 *
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches

Background


14
14
class
token
n=1
conv2d
(16x16)
D=768
n=196
patch
embedding
position
embedding
N=197
D=768
3. Transformer Encoder
Transformer
Encoder
N=197
D=768
MLP
head
'Dome'
1000
4. MLP (Classification) Head
1. Split Image into Patches
2. Add Position Embeddings

1. Split Image into Patches

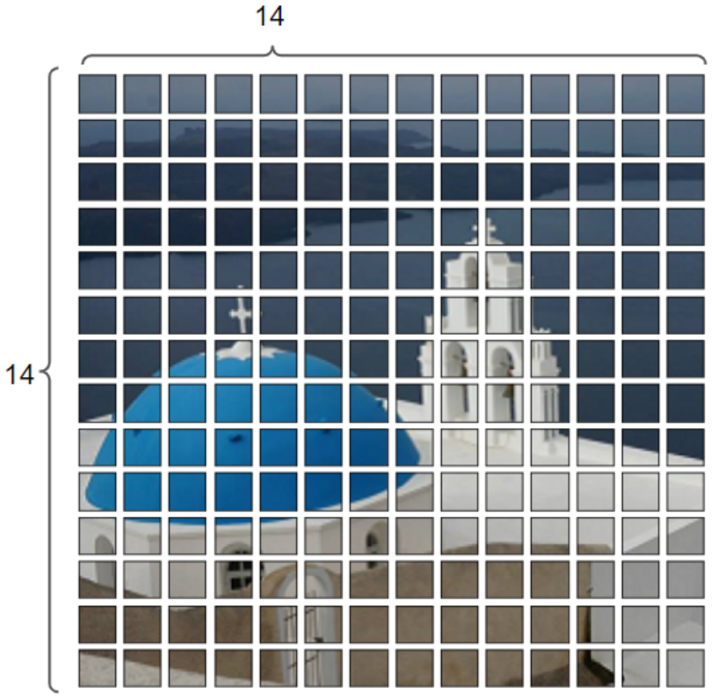


224x224

- 14x14 different tokens
- Each token: 16 x 16 patch

```python
class VisionTransformer(nn.Module):
    """ Vision Transformer """
    def __init__(self, img_size=[224], patch_size=16, in_chans=3,
                 num_classes=0,
                 embed_dim=768,
                 depth=12,
                 num_heads=12,
                 mlp_ratio=4.,
                 **kwargs):
    super().__init__()
    self.num_features = self.embed_dim = embed_dim

    self.patch_embed = PatchEmbed(img_size=img_size[0],
                                  patch_size=patch_size,
                                  in_chans=in_chans,
                                  embed_dim=embed_dim)
```

1. Split Image into Patches



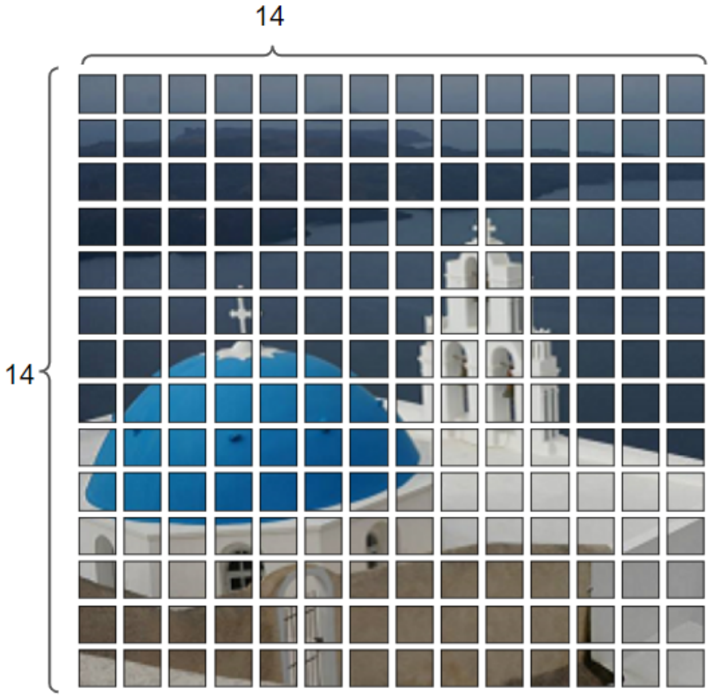
224x224

Conv2d (k=16x16) with stride=(16, 16)

- 14x14 different tokens
- Each token: 16 x 16 patch

```
class VisionTransformer(nn.Module):
    """ Vision Transformer """
    def __init__(self, img_size=[224], patch_size=16, in_chans=3,
                 num_classes=0,
                 embed_dim=768,
                 depth=12,
                 num_heads=12,
                 mlp_ratio=4.,
                 **kwargs):
    super().__init__()
    self.num_features = self.embed_dim = embed_dim

    self.patch_embed = PatchEmbed(img_size=img_size[0],
                                  patch_size=patch_size,
                                  in_chans=in_chans,
                                  embed_dim=embed_dim)
```

1. Split Image into Patches



224x224

```python
class PatchEmbed(nn.Module):
    """ Image to Patch Embedding
    """

    def __init__(self, img_size=224, patch_size=16, in_chans=3, embed_dim=768):
        super().__init__()
        num_patches = (img_size // patch_size) * (img_size // patch_size)
        self.img_size = img_size
        self.patch_size = patch_size
        self.num_patches = num_patches
        self.proj = nn.Conv2d(in_chans, embed_dim,
                              kernel_size=patch_size,
                              stride=patch_size)

    def forward(self, x):
        B, C, H, W = x.shape
        x = self.proj(x).flatten(2).transpose(1, 2)
        return x
```

Conv2d (k=16x16) with stride=(16, 16)

- 14x14 different tokens
- Each token: 16 x 16 = 196 patch

1. Split Image into Patches

```
img = nn.rand(1, 3, 224, 224)
```

```
PatchEmbed(img_size=224,
           patch_size=16,
           in_chans=3,
           embed_dim=768)
```

```
Conv2d(in_chans=3,
       embed_dim=768,
       kernel_size=16,
       stride=16)
```

Conv2d _out.shape = 1 x 768 x 14 x 14

PatchEmbed _out.shape = 1 x 196 x 768

```
class PatchEmbed(nn.Module):
    """ Image to Patch Embedding
    """
```

m=768):

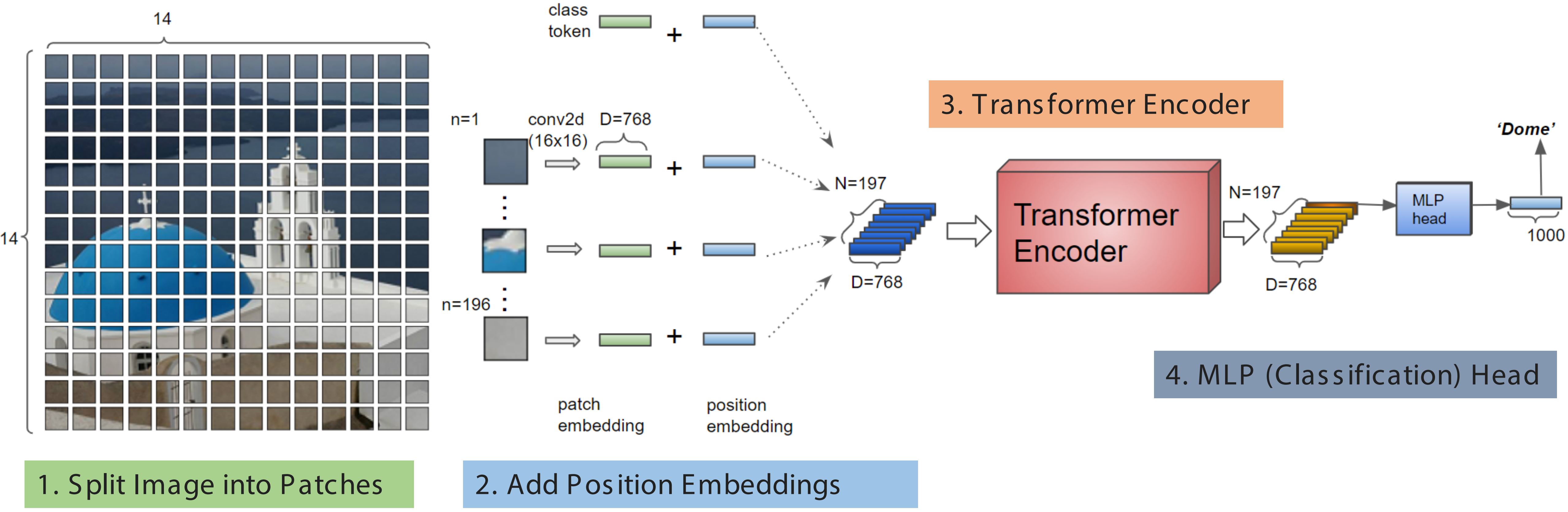
Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in})
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

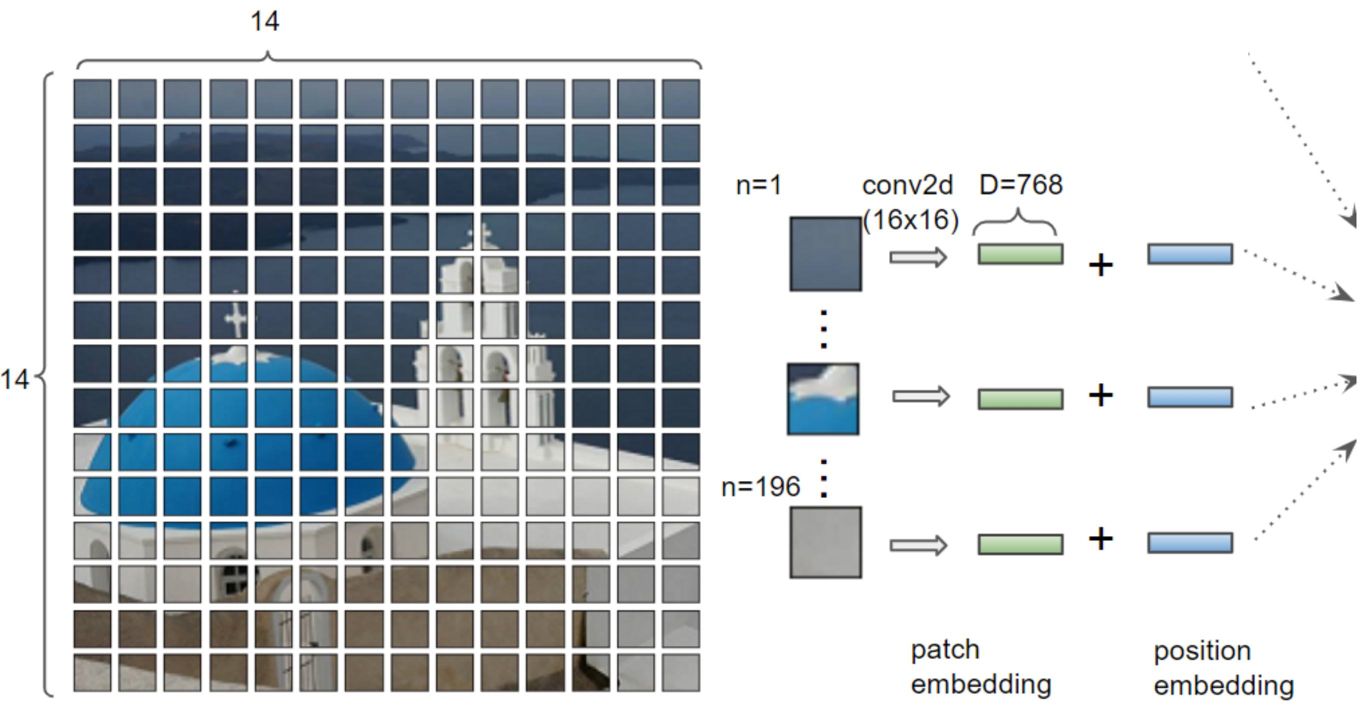
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

```
    def forward(self, x):
        B, C, H, W = x.shape
        x = self.proj(x).flatten(2).transpose(1, 2)
        return x
```


14
14
class
token
n=1
conv2d
(16x16)
D=768
n=196
patch
embedding
position
embedding
N=197
D=768
3. Transformer Encoder
Transformer
Encoder
N=197
D=768
MLP
head
'Dome'
1000
4. MLP (Classification) Head
1. Split Image into Patches
2. Add Position Embeddings

2. Add Position Embeddings

```python
class VisionTransformer(nn.Module):
    """ Vision Transformer """
    def __init__(self, img_size=[224], patch_size=16, in_chans=3,
                 num_classes=0,
                 embed_dim=768,
                 depth=12,
                 num_heads=12,
                 mlp_ratio=4.,
                 **kwargs):
    super().__init__()
    self.num_features = self.embed_dim = embed_dim

    self.patch_embed = PatchEmbed(img_size=img_size[0],
                                  patch_size=patch_size,
                                  in_chans=in_chans,
                                  embed_dim=embed_dim)
    num_patches = self.patch_embed.num_patches
```



1 head

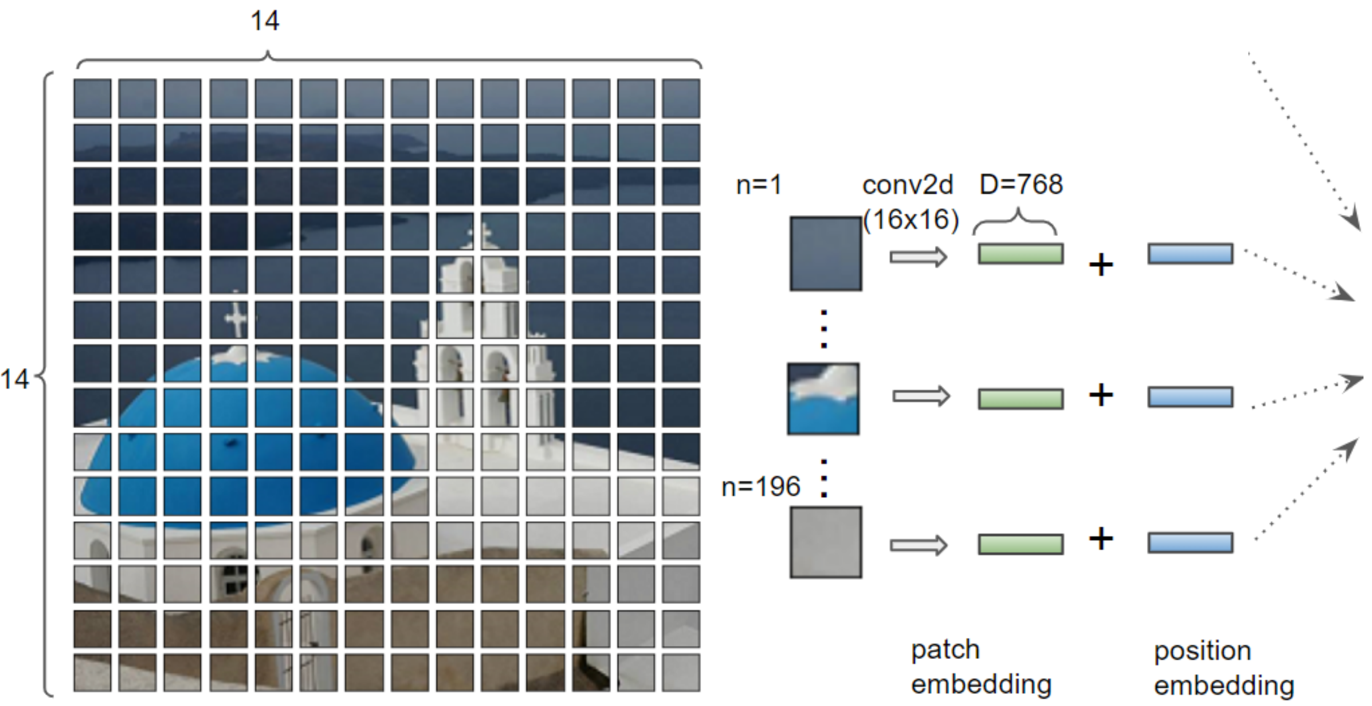
768: embedding dimension

PatchEmbed _out.shape = 1 x 196 x 768

14x14=196 image patches

2. Add Position Embeddings

```python
class VisionTransformer(nn.Module):
    """ Vision Transformer """
    def __init__(self, img_size=[224], patch_size=16, in_chans=3,
            num_classes=0,
            embed_dim=768,
            depth=12,
            num_heads=12,
            mlp_ratio=4.,
            **kwargs):
    super().__init__()
    self.num_features = self.embed_dim = embed_dim

    self.patch_embed = PatchEmbed(img_size=img_size[0],
                                  patch_size=patch_size,
                                  in_chans=in_chans,
                                  embed_dim=embed_dim)
    num_patches = self.patch_embed.num_patches
    self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
```



12 heads

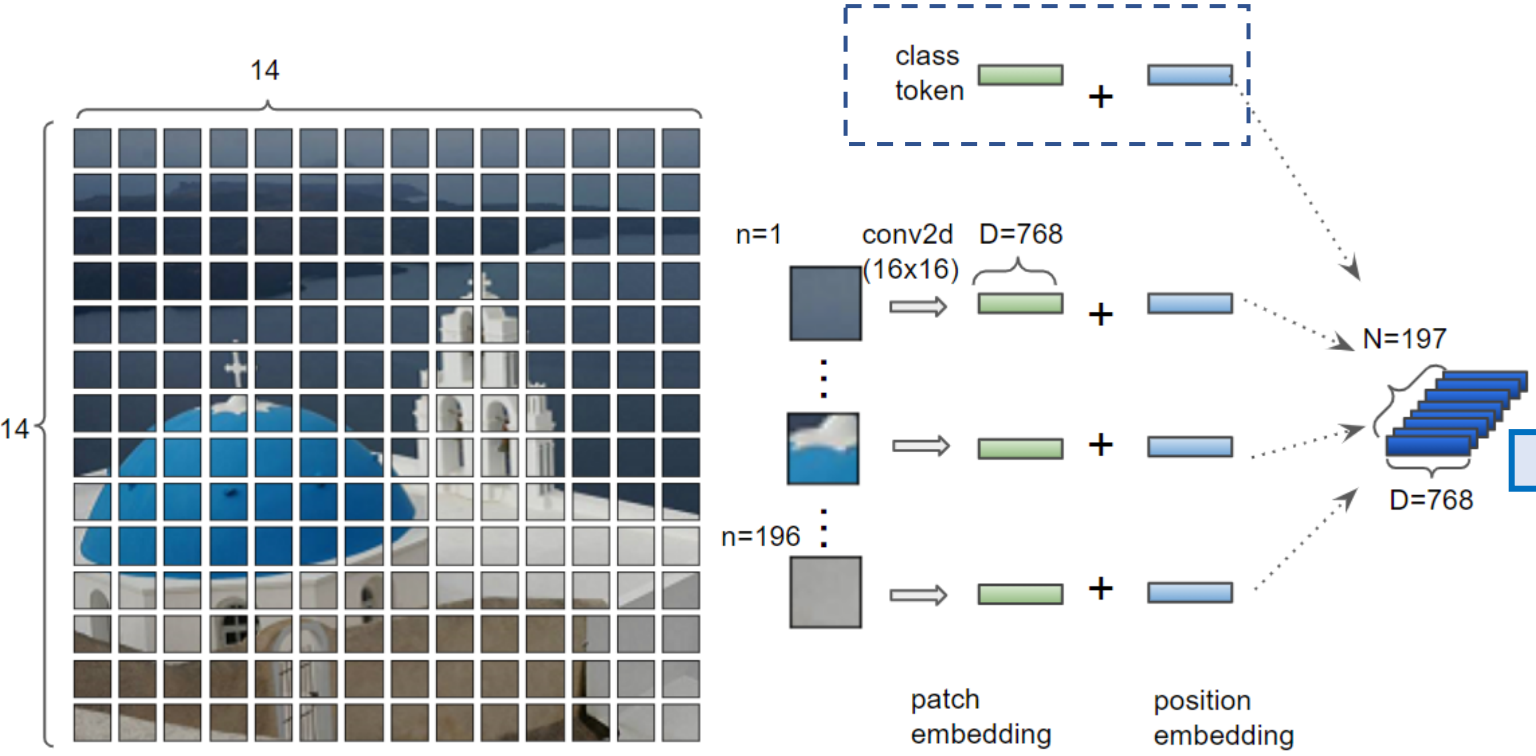
768/12 = 64: feature representation

PatchEmbed _out.shape = 12 x 196 x 64

14x14=196 image patches

2. Add Position Embeddings

```python
class VisionTransformer(nn.Module):
    """ Vision Transformer """
    def __init__(self, img_size=[224], patch_size=16, in_chans=3,
            num_classes=0,
            embed_dim=768,
            depth=12,
            num_heads=12,
            mlp_ratio=4.,
            **kwargs):
        super().__init__()
        self.num_features = self.embed_dim = embed_dim

        self.patch_embed = PatchEmbed(img_size=img_size[0],
                            patch_size=patch_size,
                            in_chans=in_chans,
                            embed_dim=embed_dim)
        num_patches = self.patch_embed.num_patches
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
```



12 heads

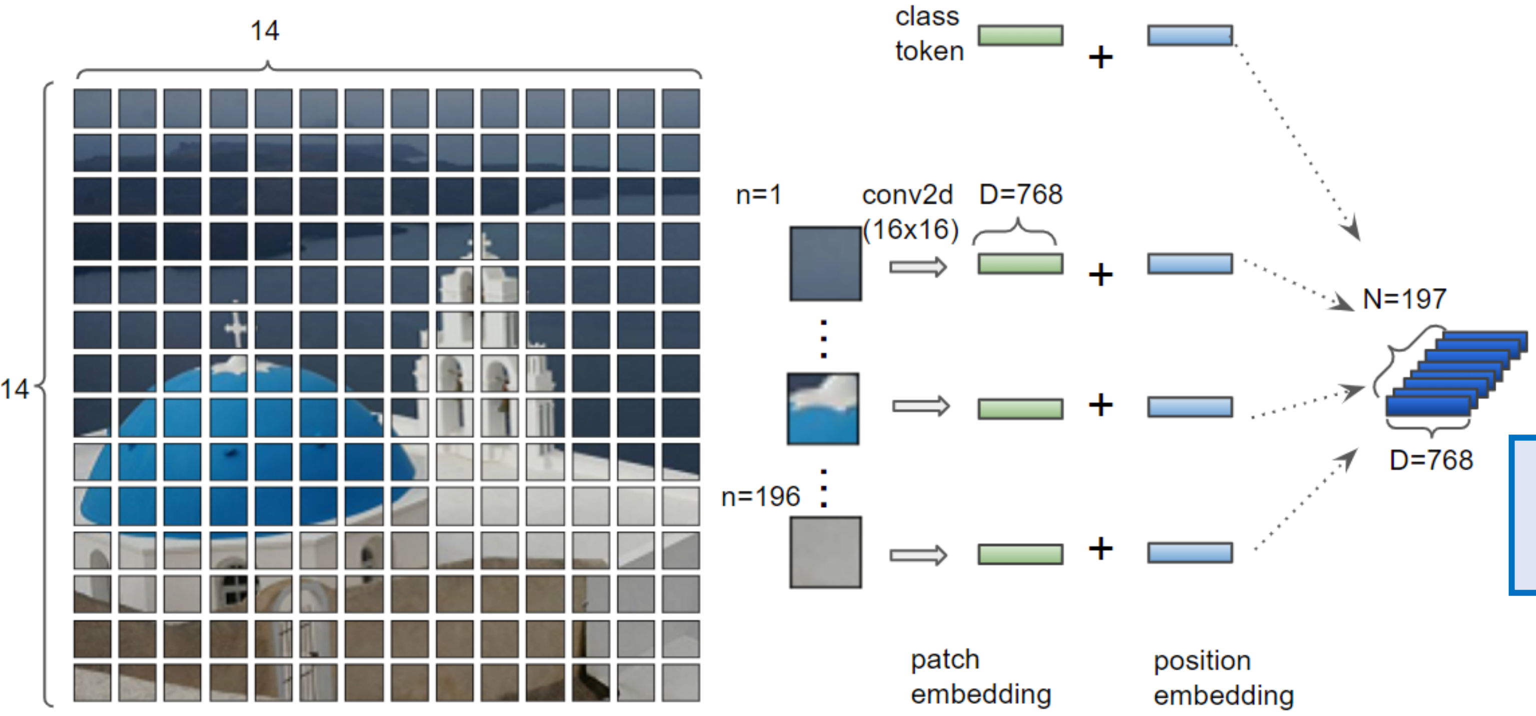
64: feature representation

PatchEmbed _out.shape = 12 x 197 x 64

14x14=196 image patches

+ 1 class token that flows through the Transformer

2. Add Position Embeddings




```python
class VisionTransformer(nn.Module):
    """ Vision Transformer """
    def __init__(self, img_size=[224], patch_size=16, in_chans=3,
            num_classes=0,
            embed_dim=768,
            depth=12,
            num_heads=12,
            mlp_ratio=4.,
            **kwargs):
    super().__init__()
    self.num_features = self.embed_dim = embed_dim

    self.patch_embed = PatchEmbed(img_size=img_size[0],
                                  patch_size=patch_size,
                                  in_chans=in_chans,
                                  embed_dim=embed_dim)
    num_patches = self.patch_embed.num_patches
    self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
    self.pos_embed = nn.Parameter(
            torch.zeros(1, num_patches + 1, embed_dim))
    self.pos_drop = nn.Dropout(p=drop_rate)
```

12 heads

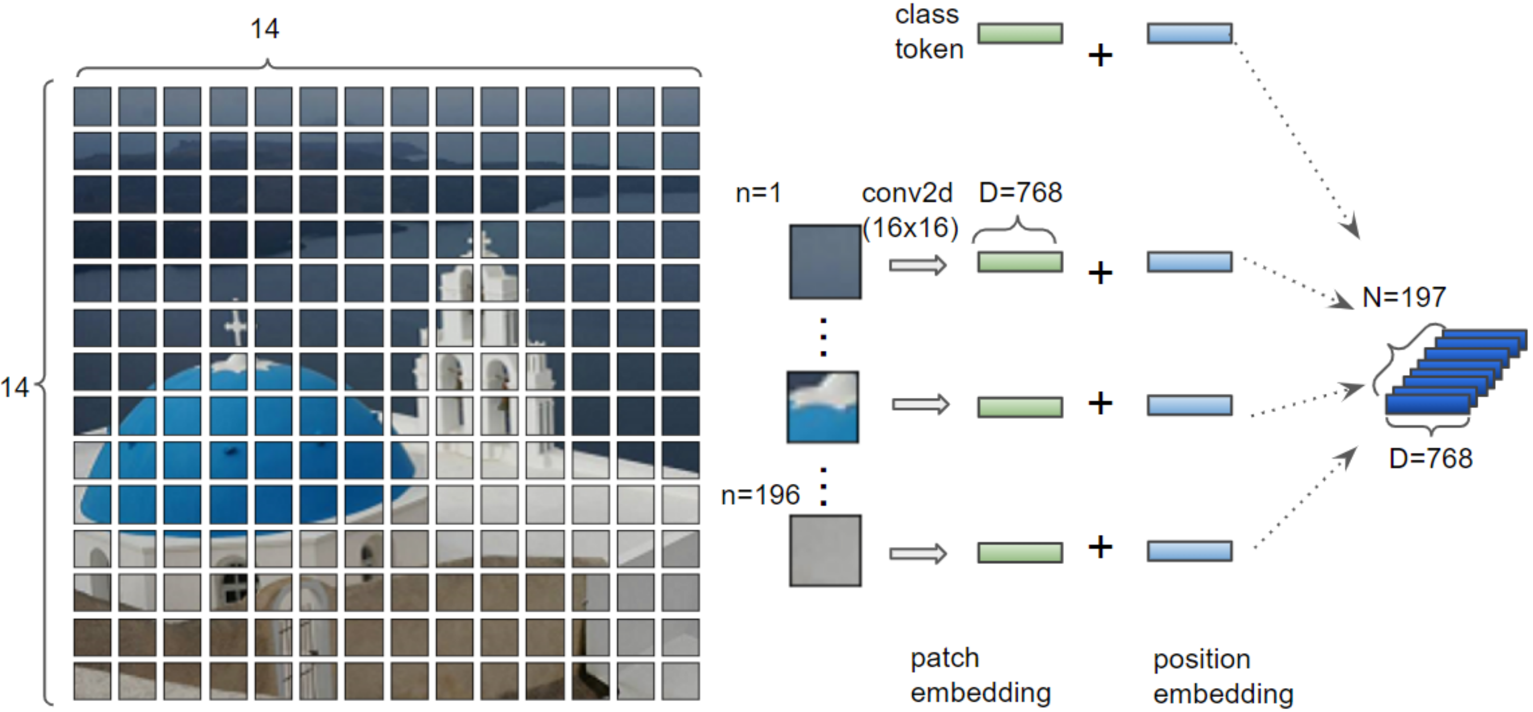
64: feature representation

PatchEmbed _out.shape = 12 x 197 x 64

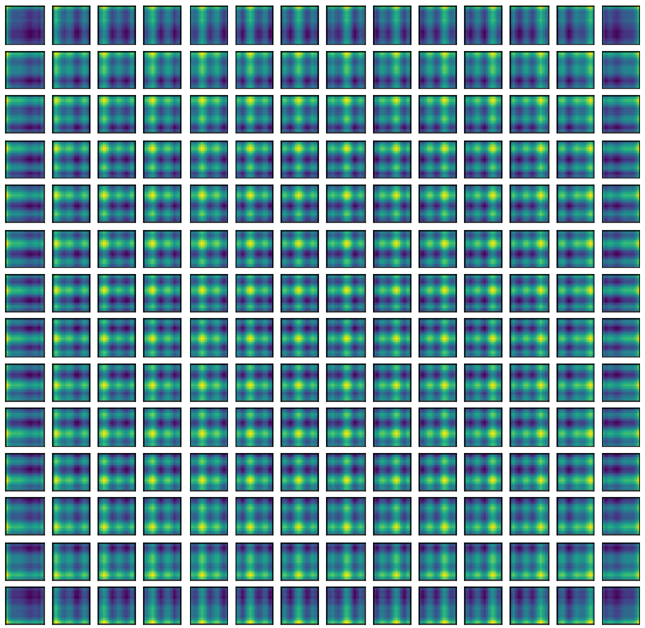
14x14=196 image patches

+ 1 class token that flows through the Transformer

2. Add Position Embeddings



Visualization of position embedding similarities



12 heads

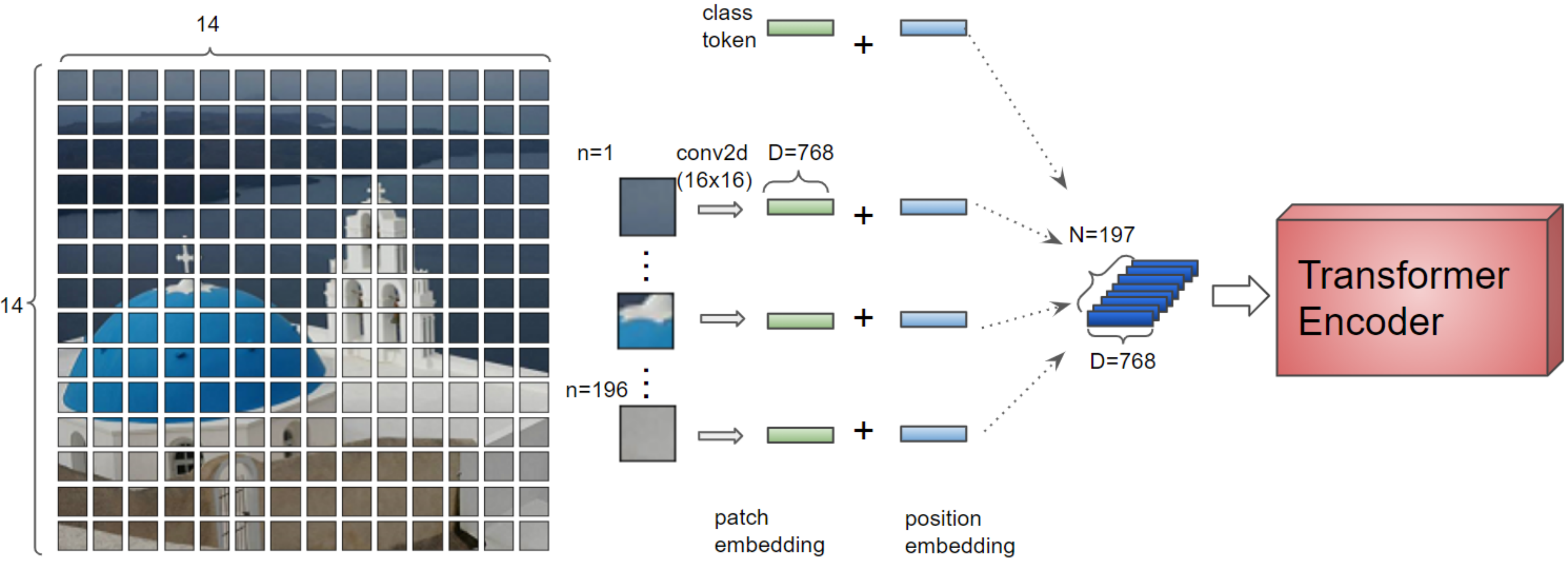
64: feature representation

PatchEmbed _out.shape = 12 x 197 x 64

14x14=196 image patches
+ 1 class token that flows through the Transformer

3. Transformer Encoder

Q, K, V and attention



12 heads

Every token has a feature representation of length 64.

PatchEmbed _out.shape = 12 x 197 x 64

Each attention heads sees 197 tokens

3. Transformer Encoder

Q, K, V and attention

PatchEmbed_out.shape = 1 x 197 x 768

1 x 197 x 768

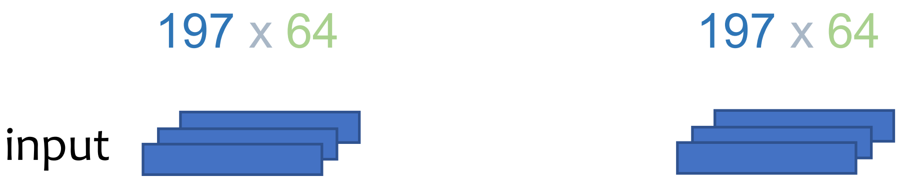
input

```
class Attention(nn.Module):

    def __init__(self, dim, num_heads=8,**.):
        super().__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim**-0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        # B = 1
        # N = 197
        # C = 768
```

3. Transformer Encoder

PatchEmbed_out.shape = 1 x 197 x 768

PatchEmbed_out.shape = 12 x 197 x 64




```
class Attention(nn.Module):

    def __init__(self, dim, num_heads=8,**.):
        super().__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim**-0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        # B = 1
        # N = 197
        # C = 768
        qkv = self.qkv(x)
                .reshape(B, N, 3, self.num_heads, C // self.num_heads)
                .permute(2, 0, 3, 1, 4)
        # qkv: 1 x 197 x 3 x 12 x 64.permute(2, 0, 3, 1, 4)
        # qkv: 3 x 1 x 12 x 197 x 64
```

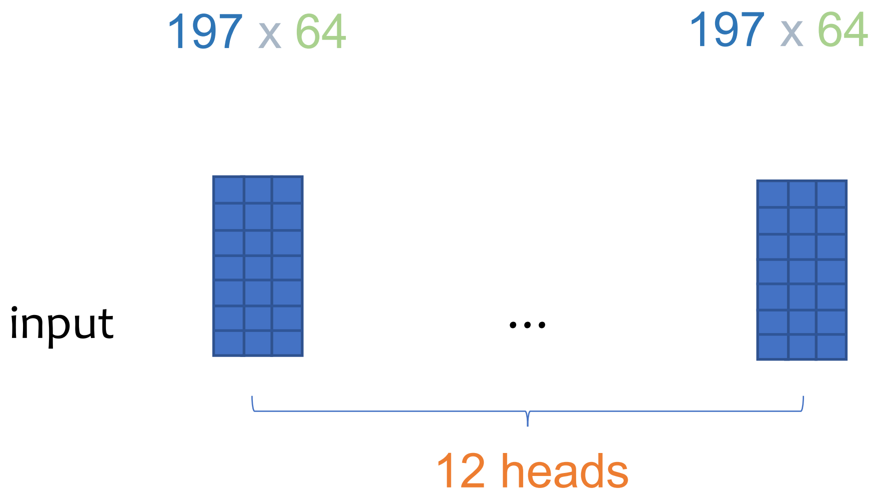
3. Transformer Encoder

PatchEmbed_out.shape = 1 x 197 x 768

PatchEmbed_out.shape = 12 x 197 x 64

Q, K, V.shape = 1 x 12 x 197 x 64

single head Q, K, V.shape = 1 x 197 x 64




```
class Attention(nn.Module):

    def __init__(self, dim, num_heads=8,**.):
        super().__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim**-0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        # B = 1
        # N = 197
        # C = 768
        qkv = self.qkv(x)
                .reshape(B, N, 3, self.num_heads, C // self.num_heads)
                .permute(2, 0, 3, 1, 4)
        # qkv: 1 x 197 x 3 x 12 x 64.permute(2, 0, 3, 1, 4)
        # qkv: 3 x 1 x 12 x 197 x 64
```

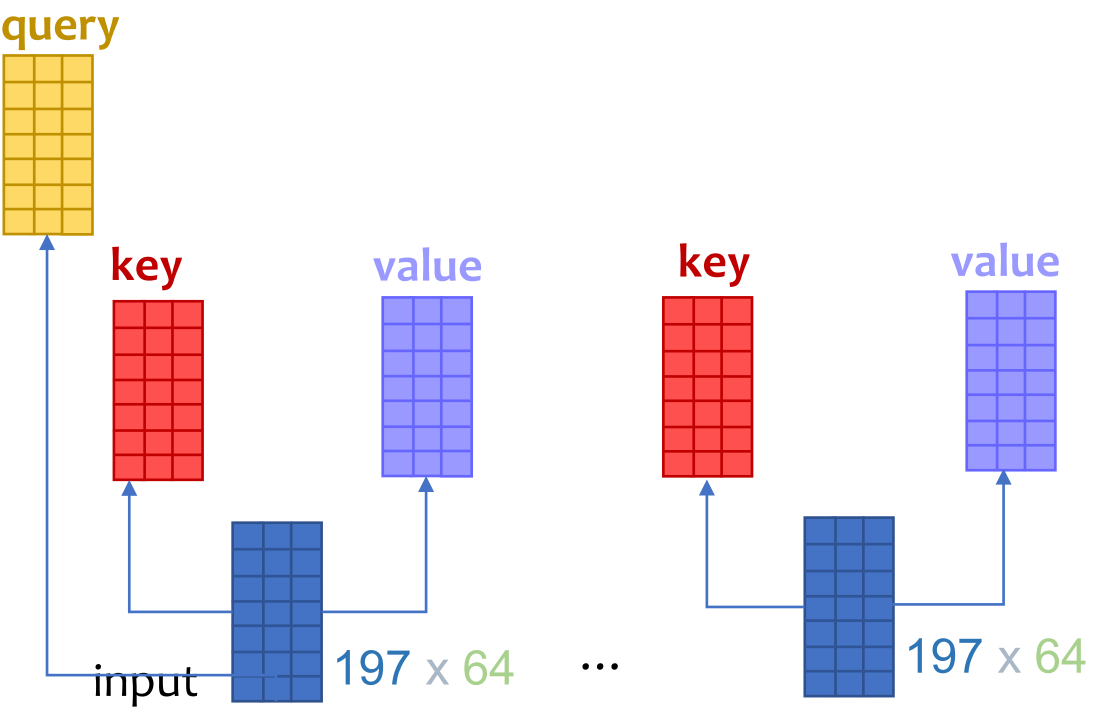
3. Transformer Encoder

PatchEmbed_out.shape = 1 x 197 x 768

PatchEmbed_out.shape = 12 x 197 x 64

Q, K, V.shape = 1 x 12 x 197 x 64

single head Q, K, V.shape = 1 x 197 x 64




```
class Attention(nn.Module):

    def __init__(self, dim, num_heads=8,**.):
        super().__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim**-0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        qkv = self.qkv(x)
                .reshape(B, N, 3, self.num_heads, C // self.num_heads)
                .permute(2, 0, 3, 1, 4)
```

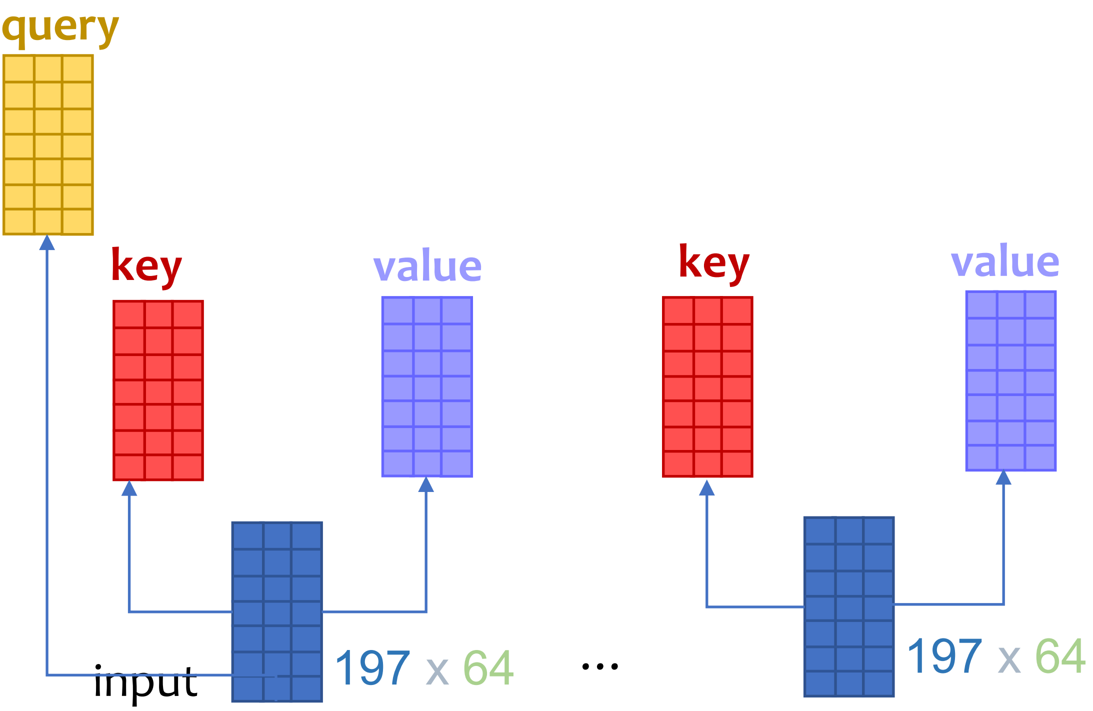
3. Transformer Encoder

PatchEmbed_out.shape = 1 x 197 x 768

PatchEmbed_out.shape = 12 x 197 x 64

Q, K, V.shape = 1 x 12 x 197 x 64

single head Q, K, V.shape = 1 x 197 x 64

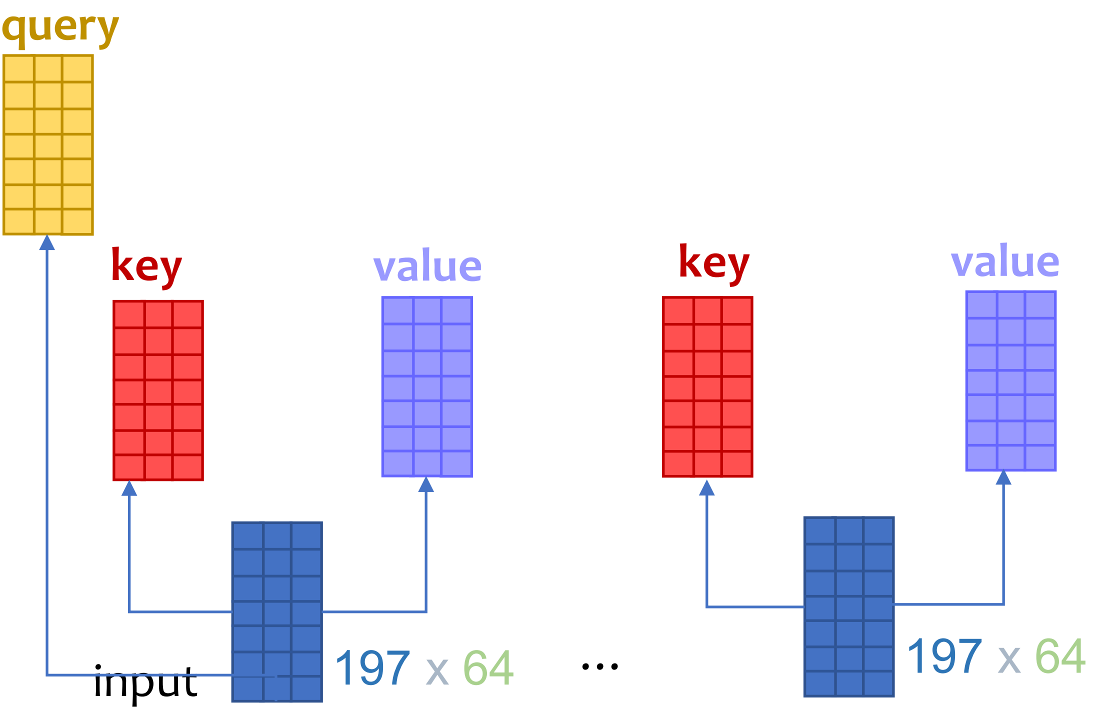


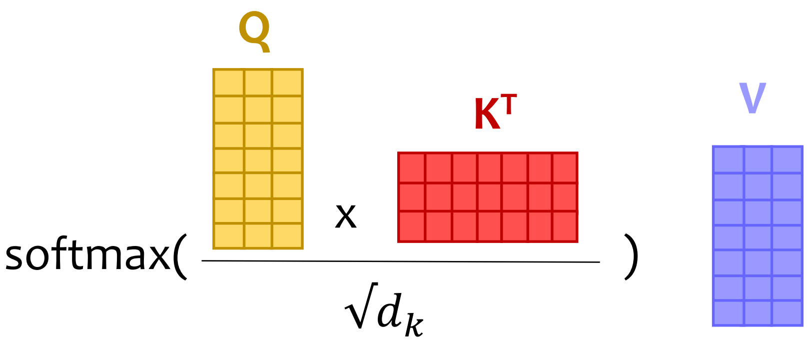

```
class Attention(nn.Module):

    def __init__(self, dim, num_heads=8,**.):
        super().__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim**-0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        qkv = self.qkv(x)
                .reshape(B, N, 3, self.num_heads, C // self.num_heads)
                .permute(2, 0, 3, 1, 4)
```

3. Transformer Encoder

Q, K, V.shape = 1 x 12 x 197 x 64

Q K^T = 1 x 12 x 197 x 197

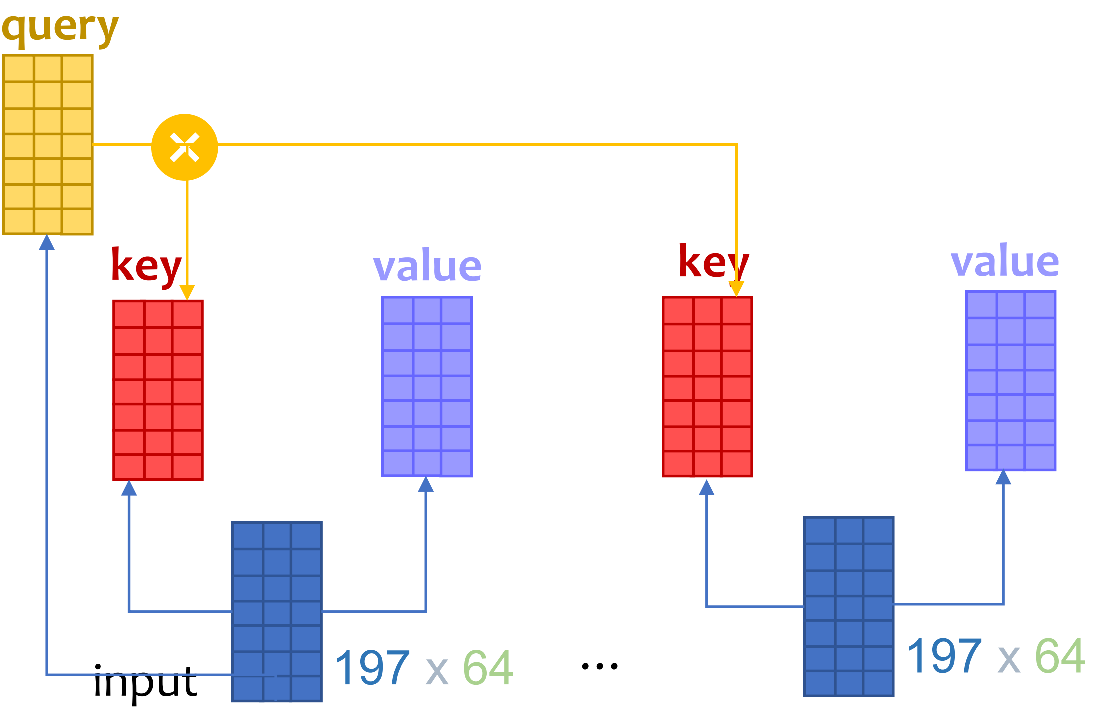


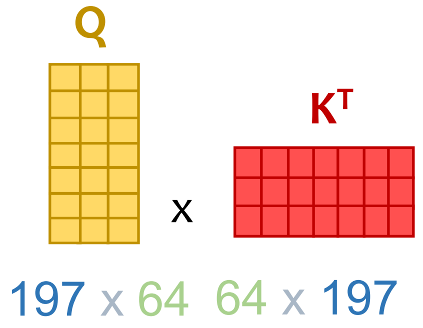


3. Transformer Encoder

$Q, K, V.\text{shape} = 1 \times 12 \times 197 \times 64$

$QK^T = 1 \times 12 \times 197 \times 197$

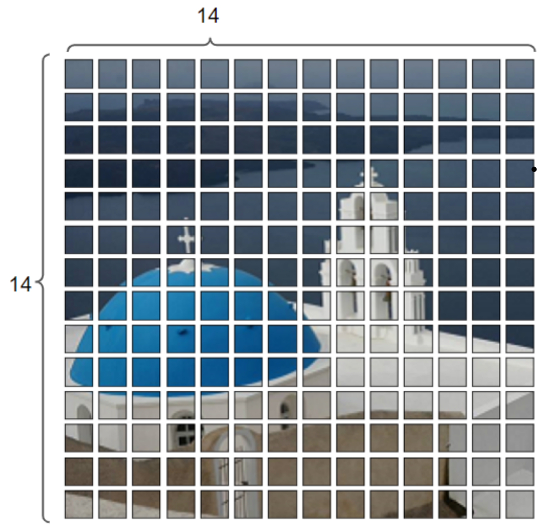


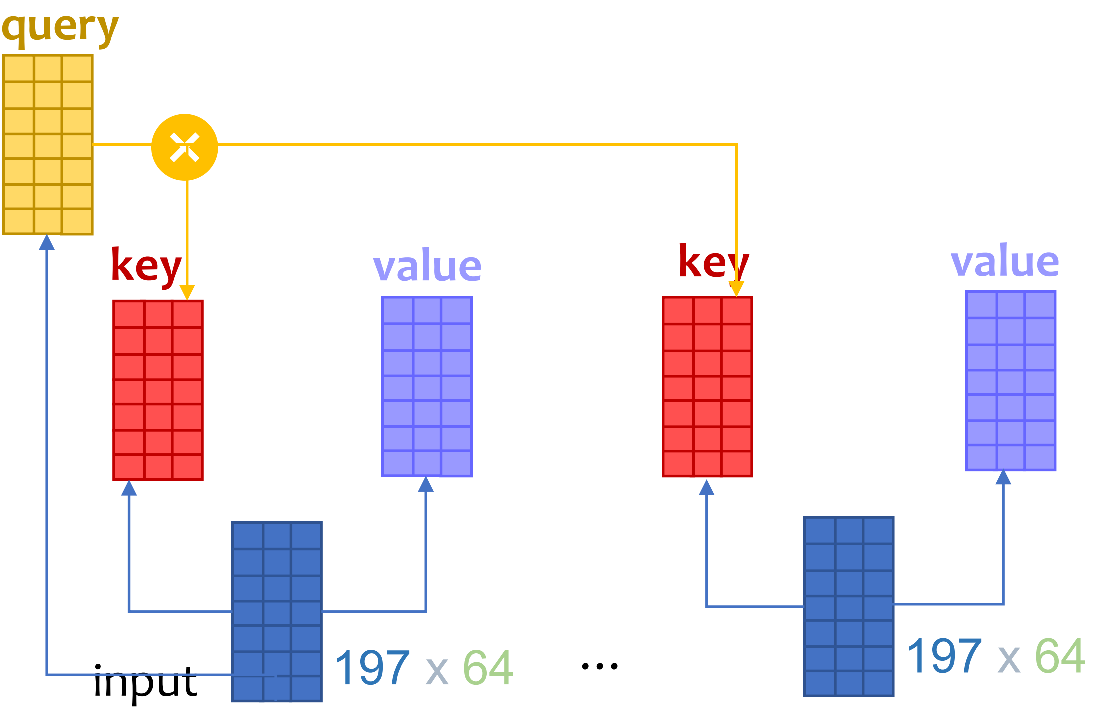


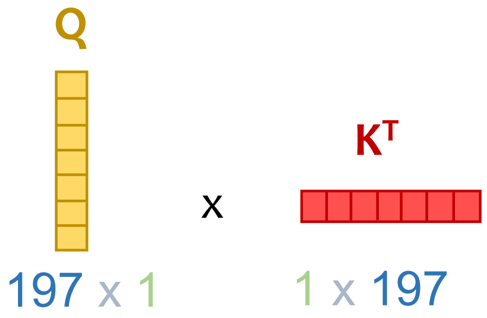
3. Transformer Encoder

single Q, K, V.shape = 1 x 197 x 64

single Q K^T = 1 x 197 x 197



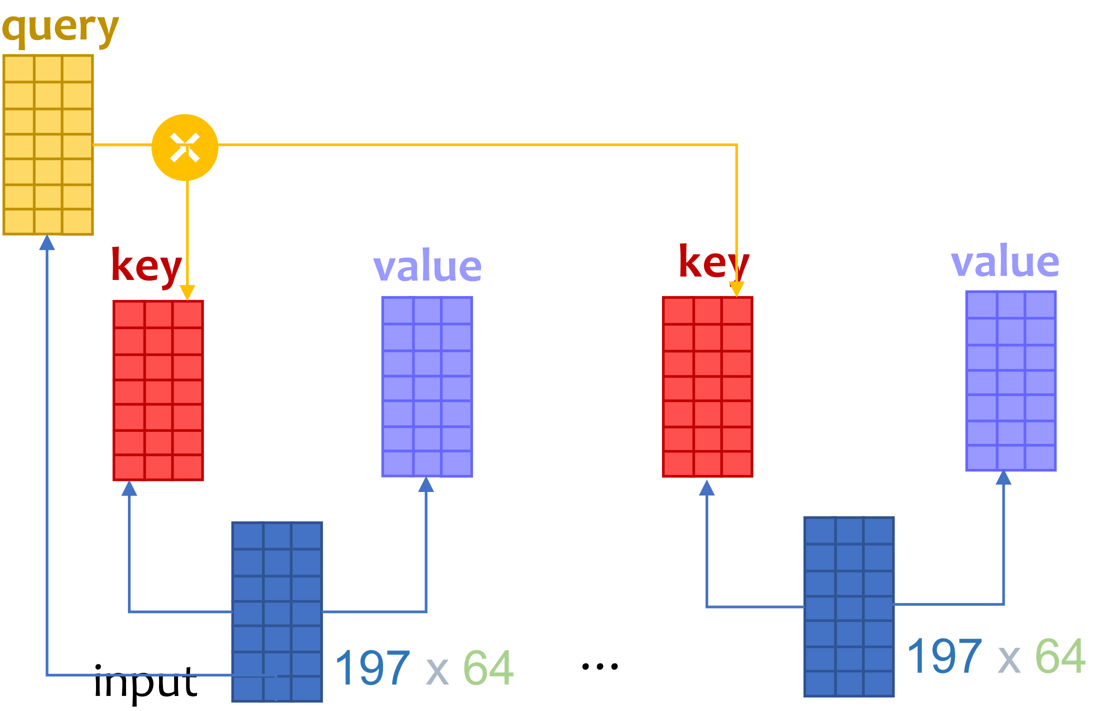


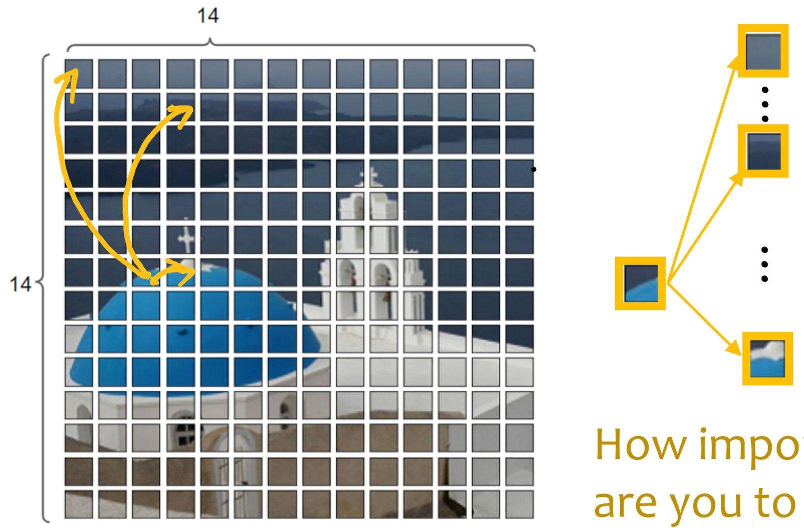


3. Transformer Encoder

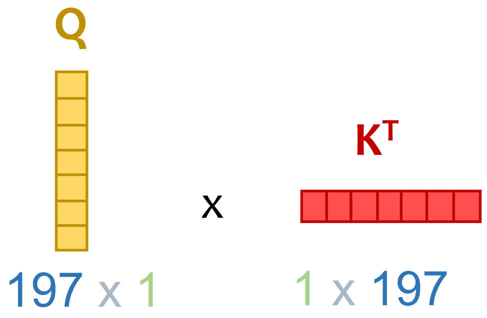
single Q, K, V.shape = 1 x 197 x 64

single Q K^T = 1 x 197 x 197





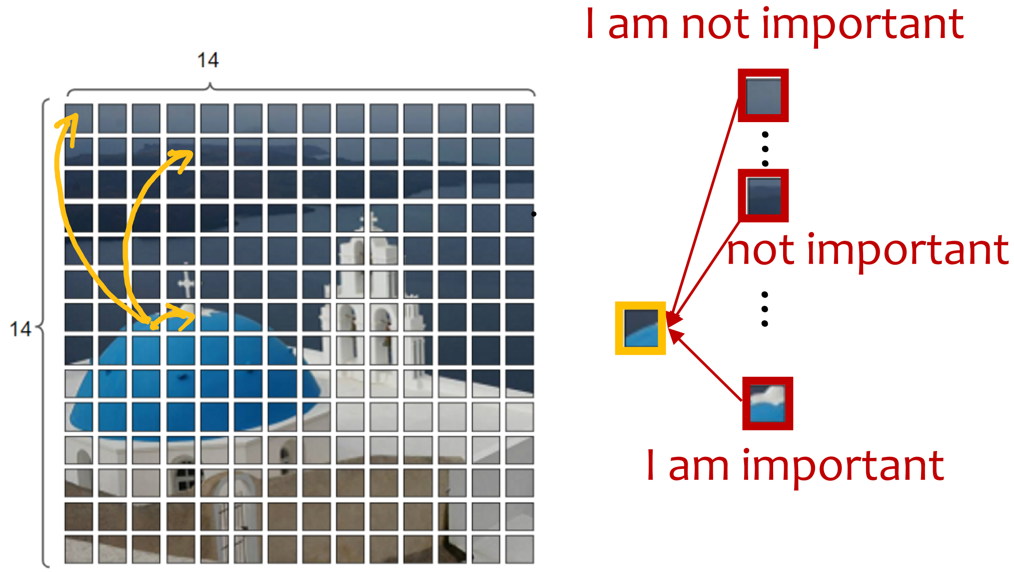
How important are you to me?

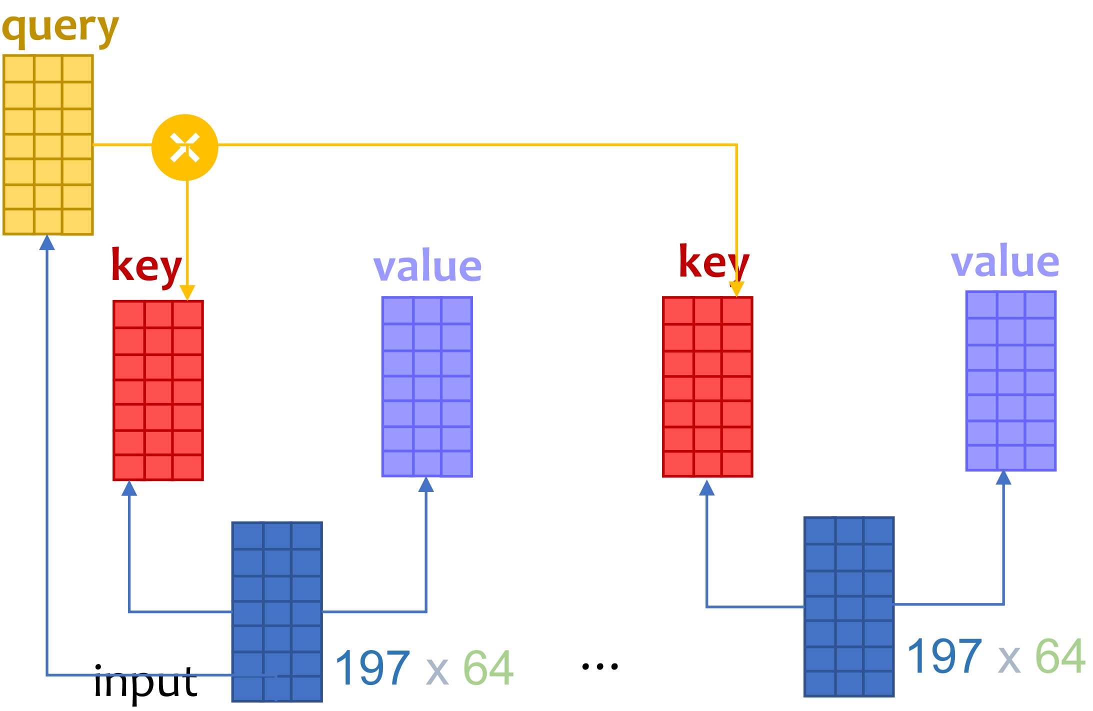


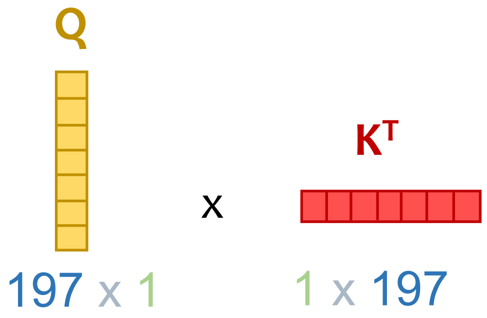
3. Transformer Encoder

single Q, K, V.shape = 1 x 197 x 64

single Q K^T = 1 x 197 x 197



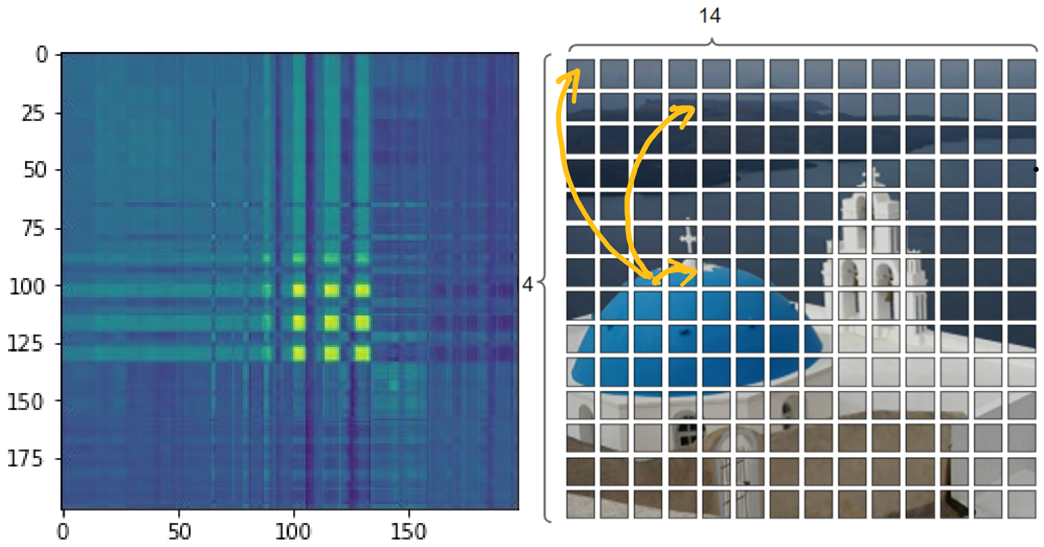


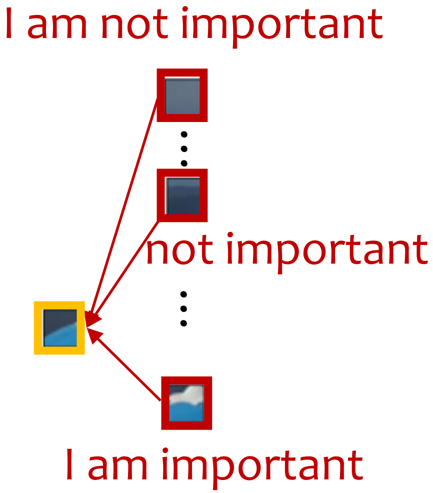


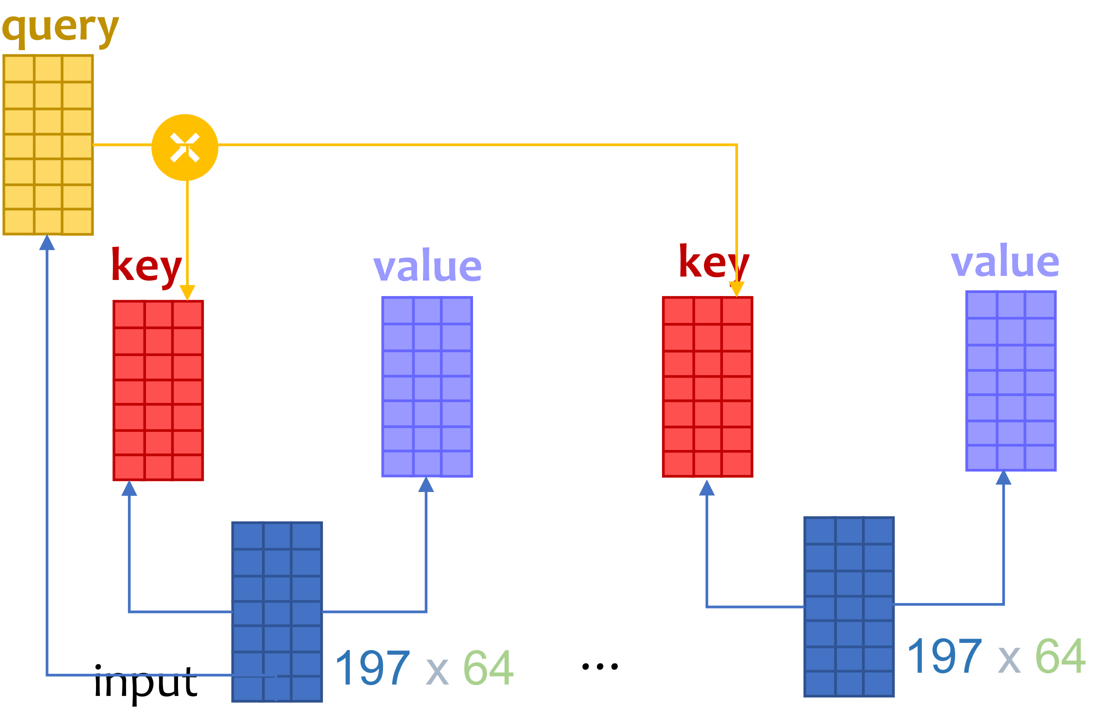
3. Transformer Encoder

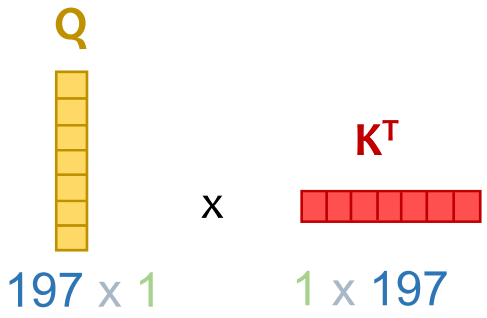
single Q, K, V.shape = 1 x 197 x 64

single Q K^T = 1 x 197 x 197







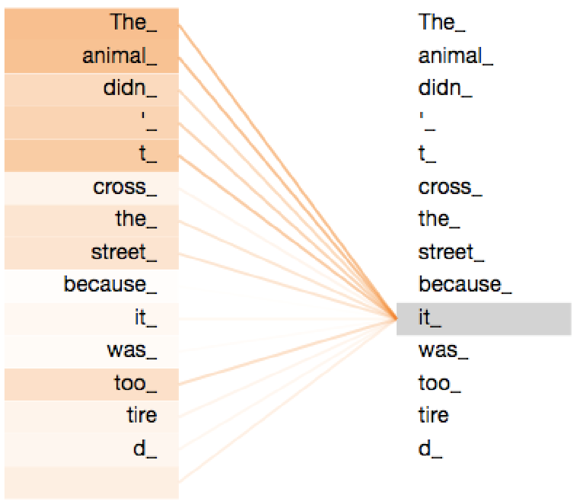


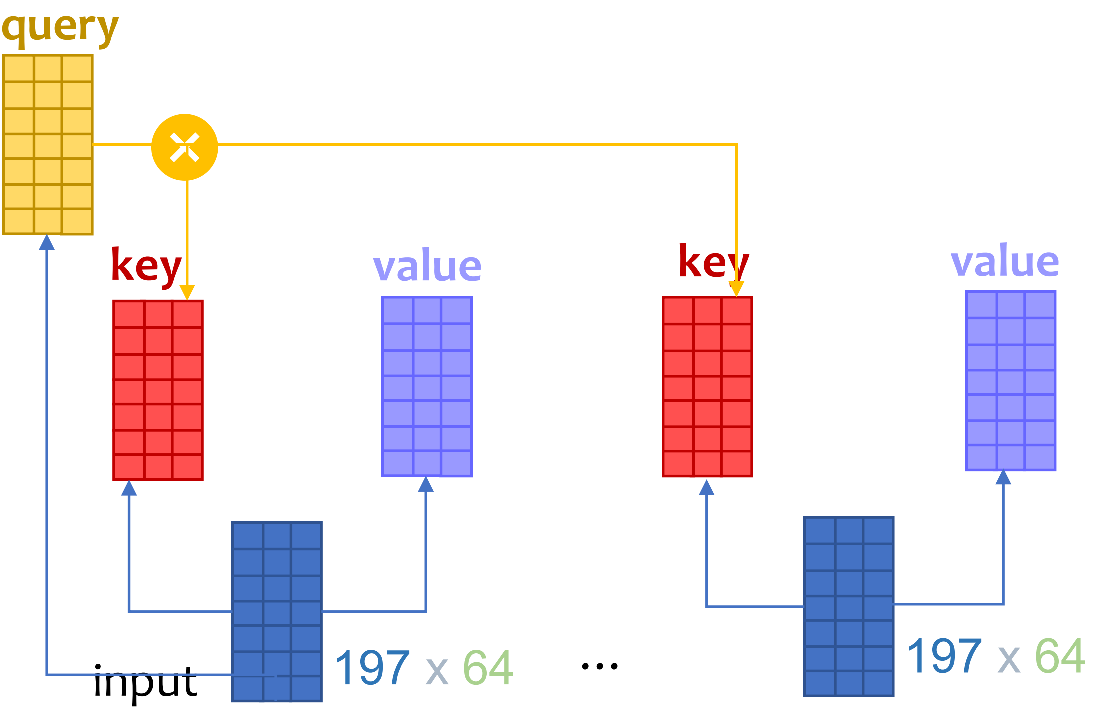
3. Transformer Encoder

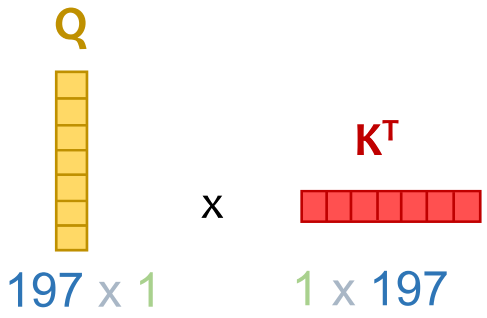
Q, K, V and attention

single Q, K, V.shape = 1 x 197 x 64

single Q K^T = 1 x 197 x 197



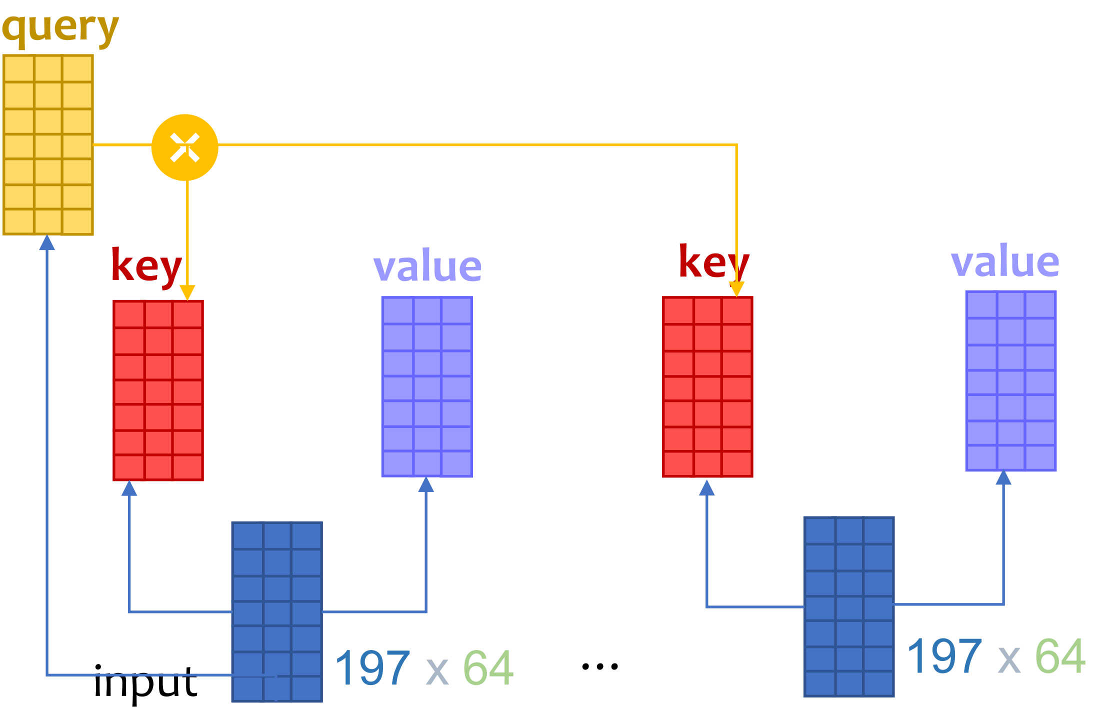




Q, K, V and attention

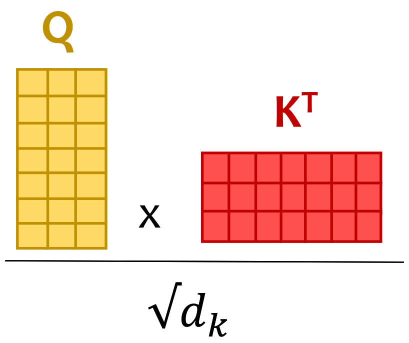
Q, K, V.shape = 1 x 12 x 197 x 64

dot_prod_attention.shape = 1 x 12 x 197 x 197



dot product attention

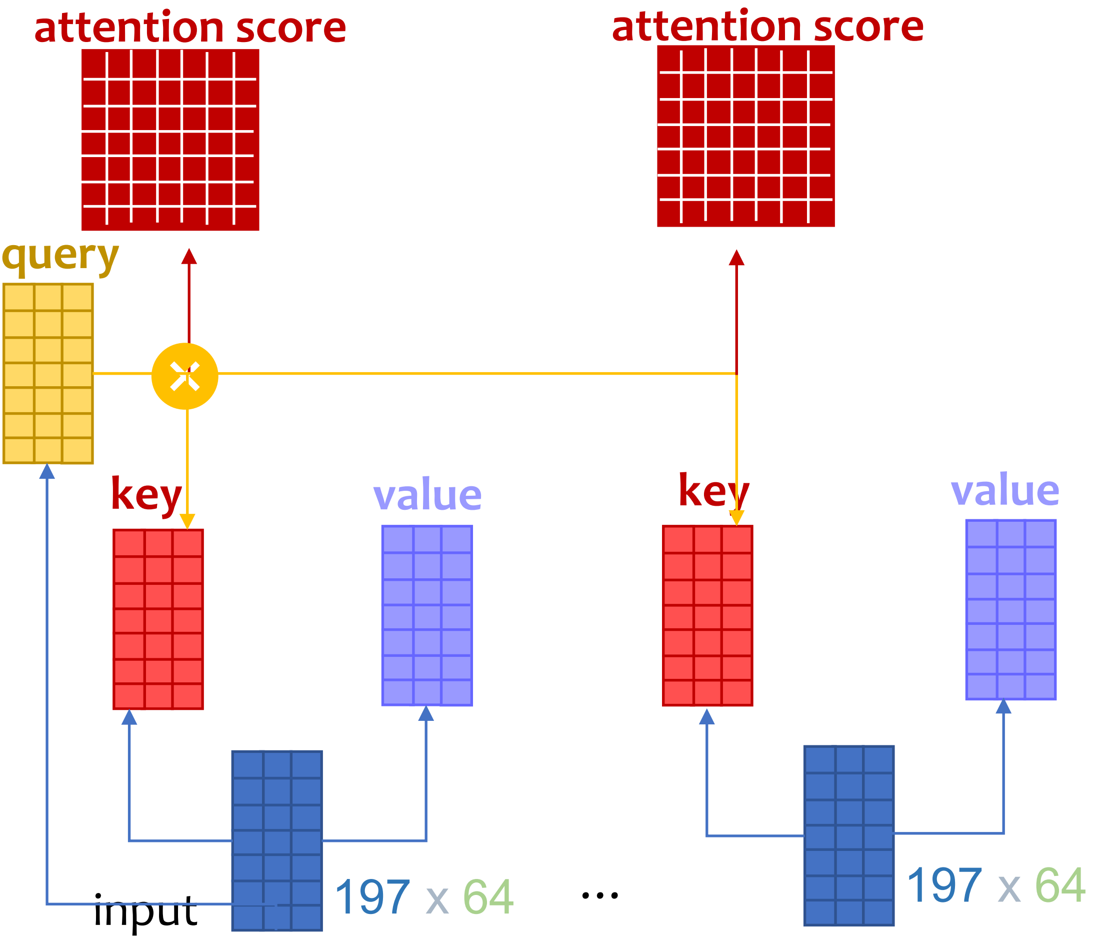
normalization



Q, K, V and attention

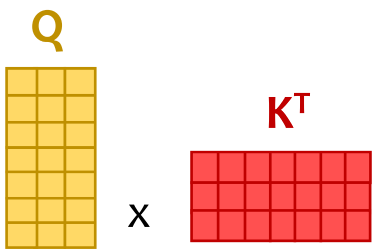
Q, K, V.shape = 1 x 12 x 197 x 64

attention_score.shape = 1 x 12 x 197 x 197



attention scores $\mathrm{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right)$

3. Transformer Encoder



$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

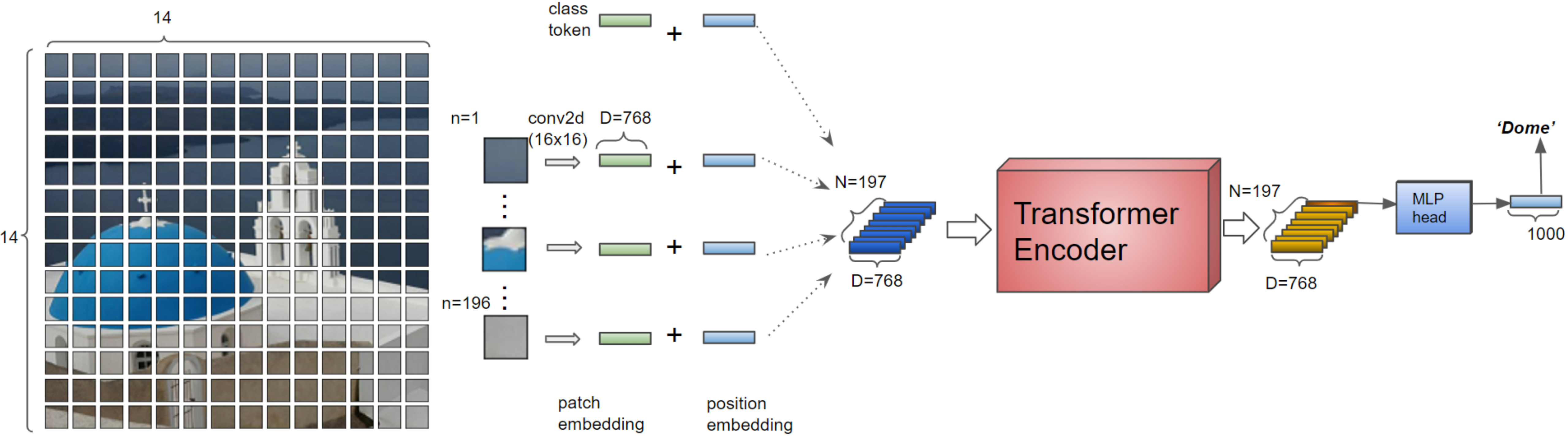
PatchEmbed_out.shape = 1 x 197 x 768

Q, K, V.shape = 1 x 12 x 197 x 64

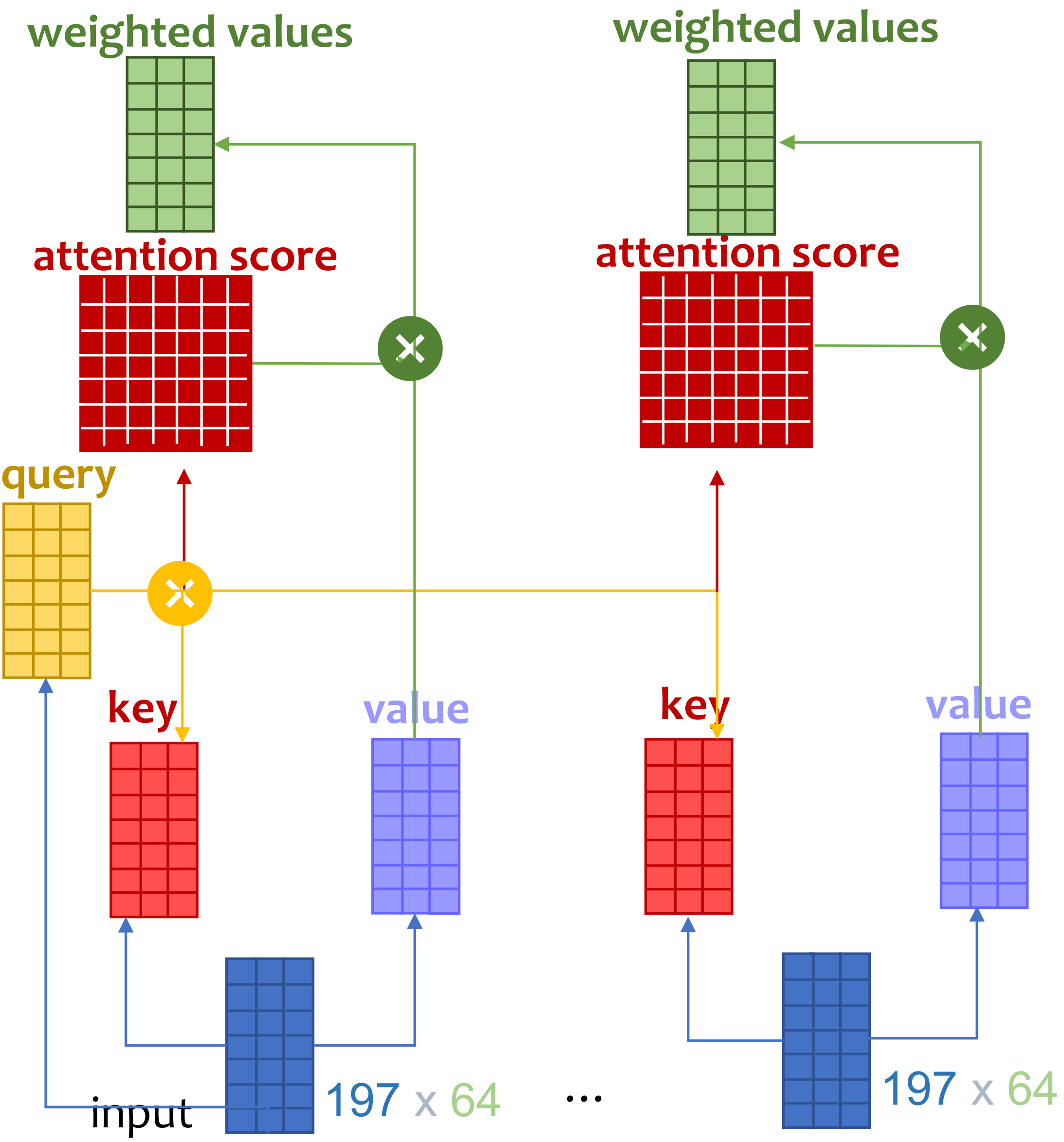
attention_score.shape = 1 x 12 x 197 x 197

```
class Attention(nn.Module):

    def __init__(self, dim, num_heads=8,**.):
        super().__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim**-0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        qkv = self.qkv(x)
                .reshape(B, N, 3, self.num_heads, C // self.num_heads)
                .permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]

        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)
```

4. MLP (Classification) Head



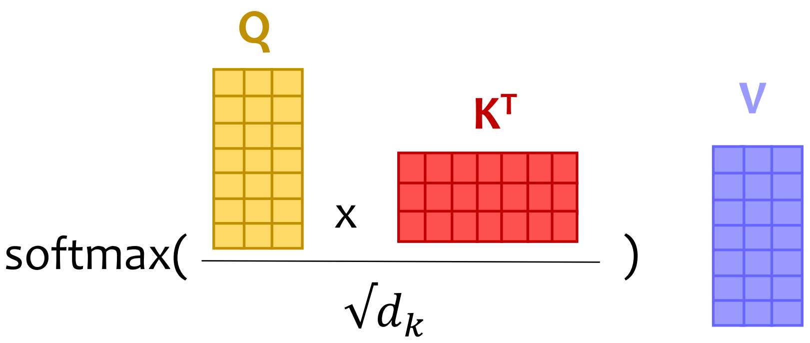
4. MLP (Classification) Head



Q, K, V.shape = 1 x 12 x 197 x 64

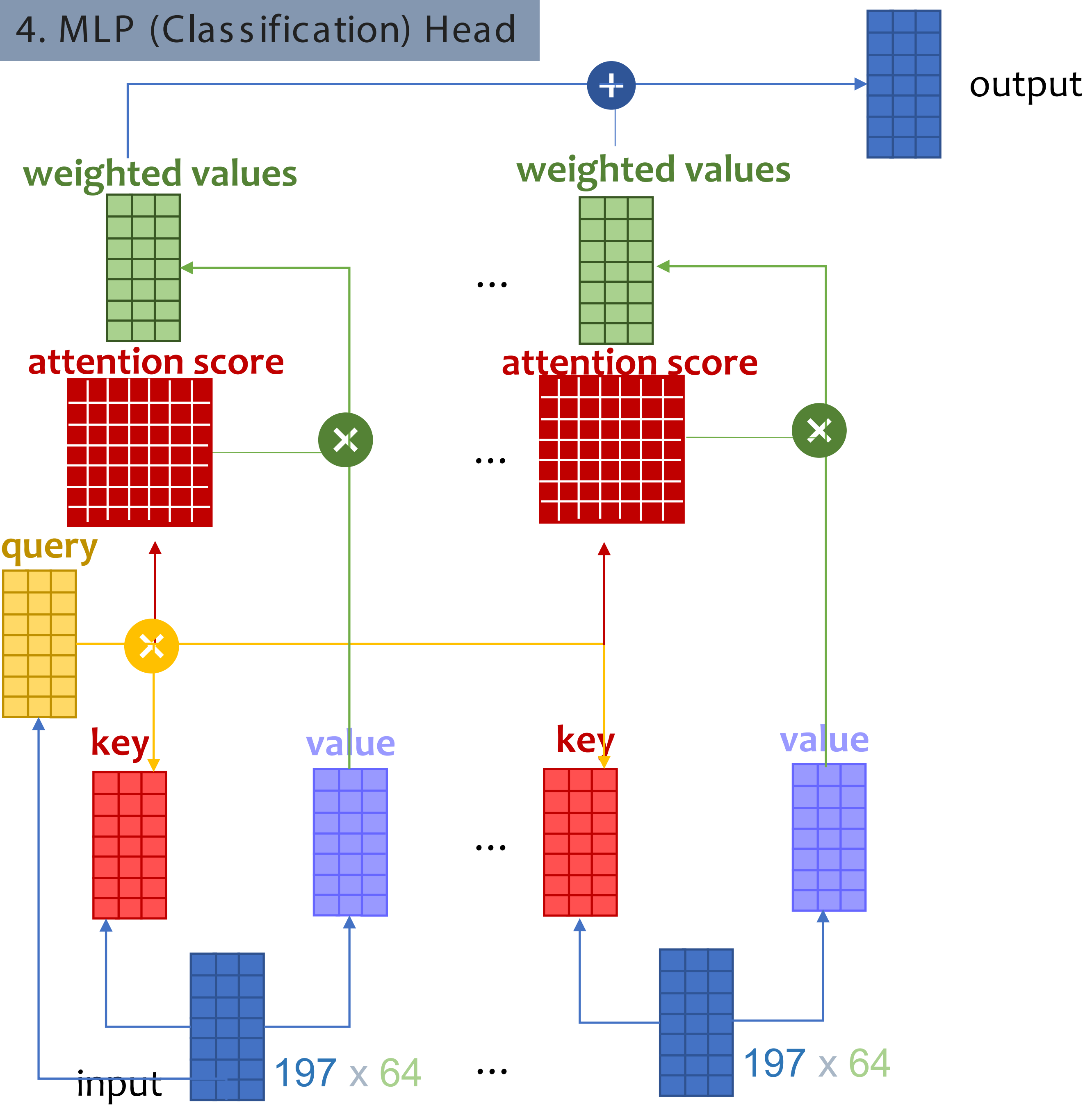
attention_score.shape = 1 x 12 x 197 x 197

out.shape = 1 x 12 x 197 x 64



weighted values

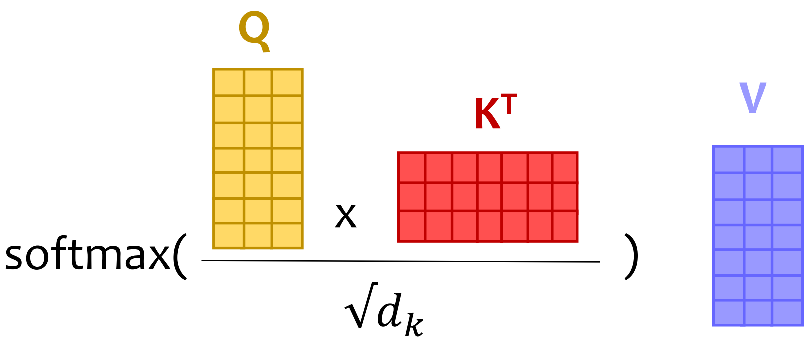
4. MLP (Classification) Head



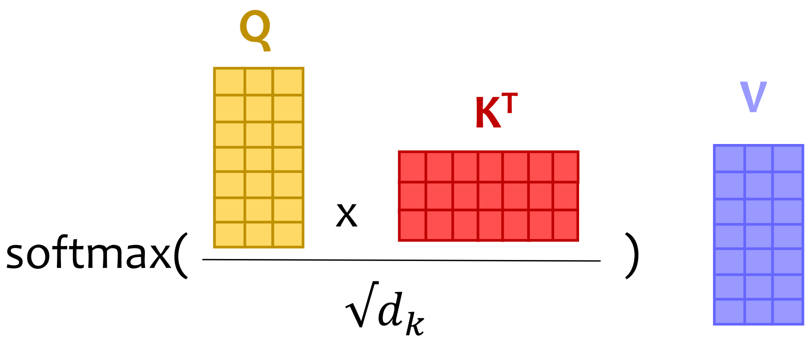
Q, K, V.shape = 1 x 12 x 197 x 64

attention_score.shape = 1 x 12 x 197 x 197

concat_out.shape = 1 x 197 x 768



4. MLP (Classification) Head



PatchEmbed_out.shape = 1 x 197 x 768

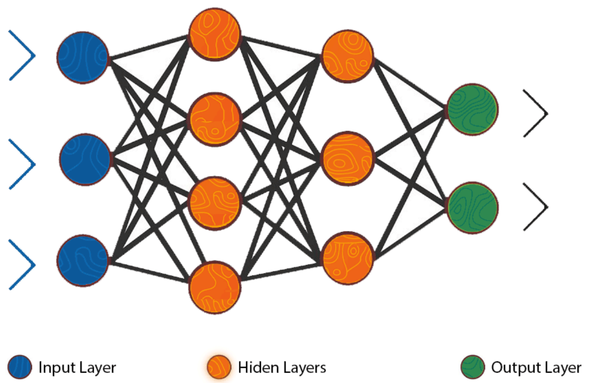
Q, K, V.shape = 1 x 12 x 197 x 64

attention_score.shape = 1 x 12 x 197 x 197

concat_out.shape = 1 x 197 x 768

```
class Attention(nn.Module):

    def __init__(self, dim, num_heads=8,**.):
        super().__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim**-0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        qkv = self.qkv(x)
                .reshape(B, N, 3, self.num_heads, C // self.num_heads)
                .permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]

        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)

        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
        x = self.proj(x)
        x = self.proj_drop(x)
        return x
```

4. MLP (Classification) Head



PatchEmbed_out.shape = 1 x 197 x 768

Q, K, V.shape = 1 x 12 x 197 x 64

attention_score.shape = 1 x 12 x 197 x 197

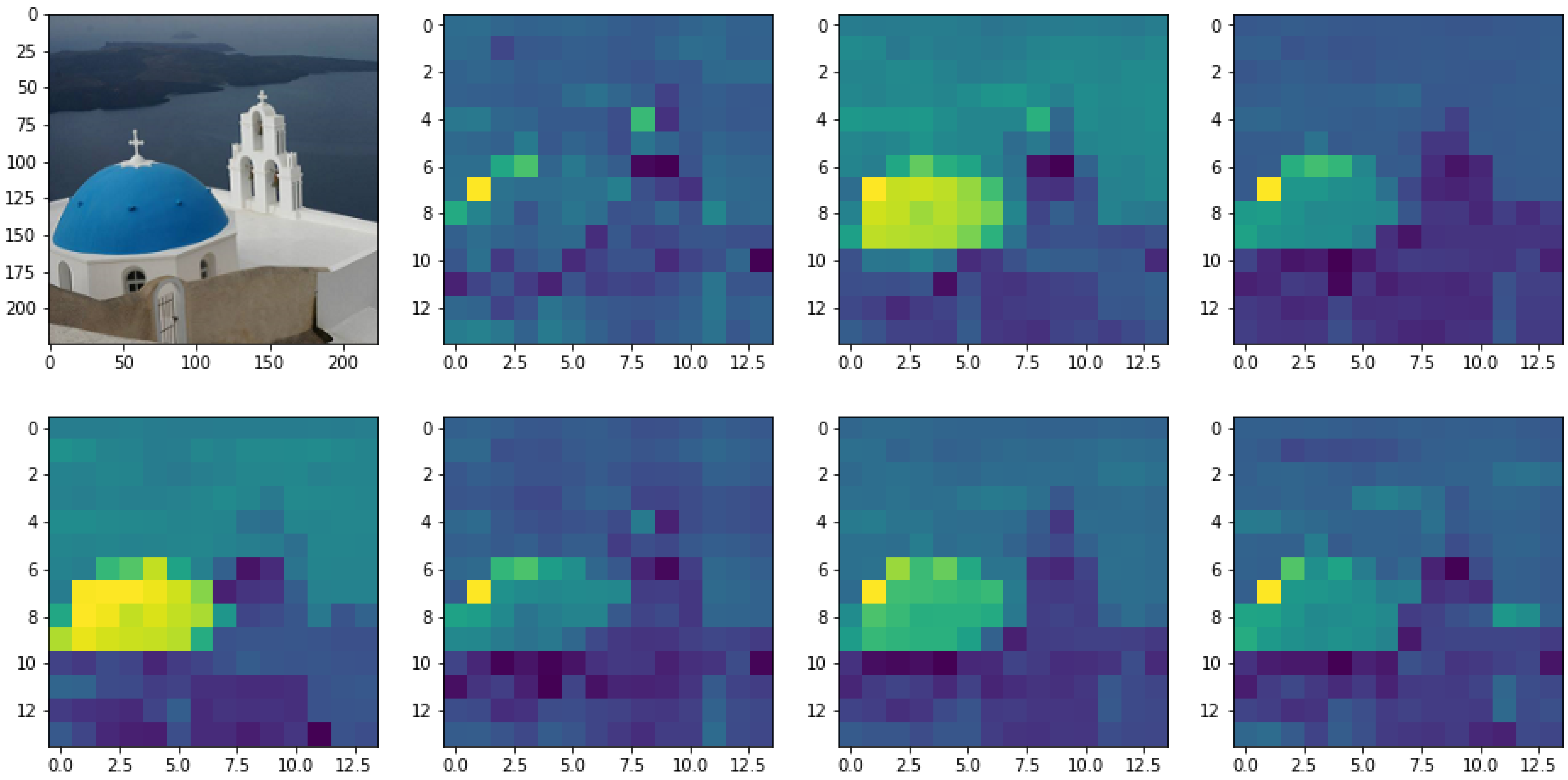
concat_out.shape = 1 x 197 x 768

out.shape = 1 x 768

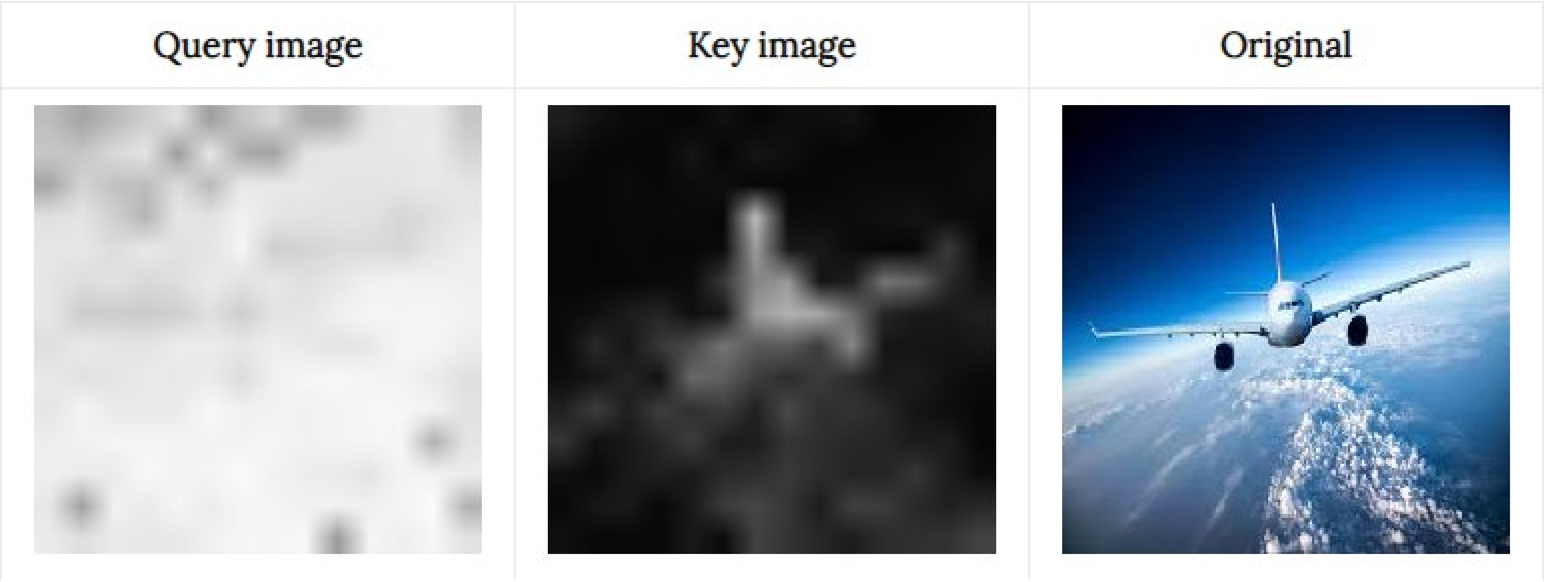
```
class VisionTransformer(nn.Module):
    """ Vision Transformer """
    def __init__():
    super().__init__()

    # Split into Patches
    self.num_features = self.embed_dim = embed_dim
    self.patch_embed = PatchEmbed()

    # Positional Encoding
    self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
    self.pos_embed = nn.Parameter(
                torch.zeros(1, num_patches + 1, embed_dim))

   # Multi-head Attention
    self.norm
    self.attention
    self.mlp
    self.norm

    # Classifier head
    self.head = nn.Linear(embed_dim, num_classes) if num_classes > 0 else nn.Identity()
    trunc_normal_(self.pos_embed, std=.02)
    trunc_normal_(self.cls_token, std=.02)
    self.apply(self._init_weights)
```

Attention maps

Visualization of Attention



Attention maps



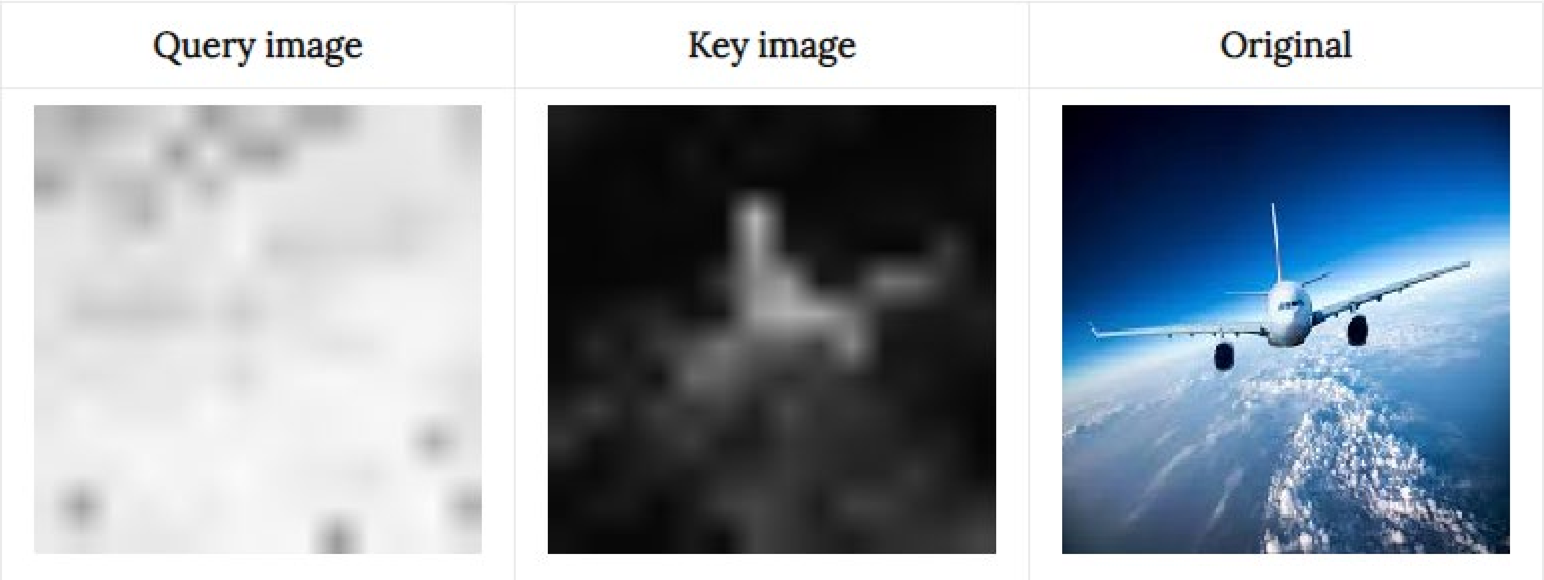
q_{ic}+ k_{jc}+ $q_{ic}\, k_{jc}^{T}$+
q_{ic}- k_{jc}- $q_{ic}\, k_{jc}^{T}$+
This means that the image location j and channel c - k_{jc} - is going to contribute to flowing information into that image location q_i

q_{ic}+ k_{ic}- $q_{ic}\, k_{ic}^{T}$-

This means that the image location j and channel c - k_{jc} - is NOT going to contribute to flowing information into that image location q_i

Figure: https://jacobgil.github.io/deeplearning/vision-transformer-explainability

Attention maps



- The key image highlights the Airplane.
- The query image highlights all the image.

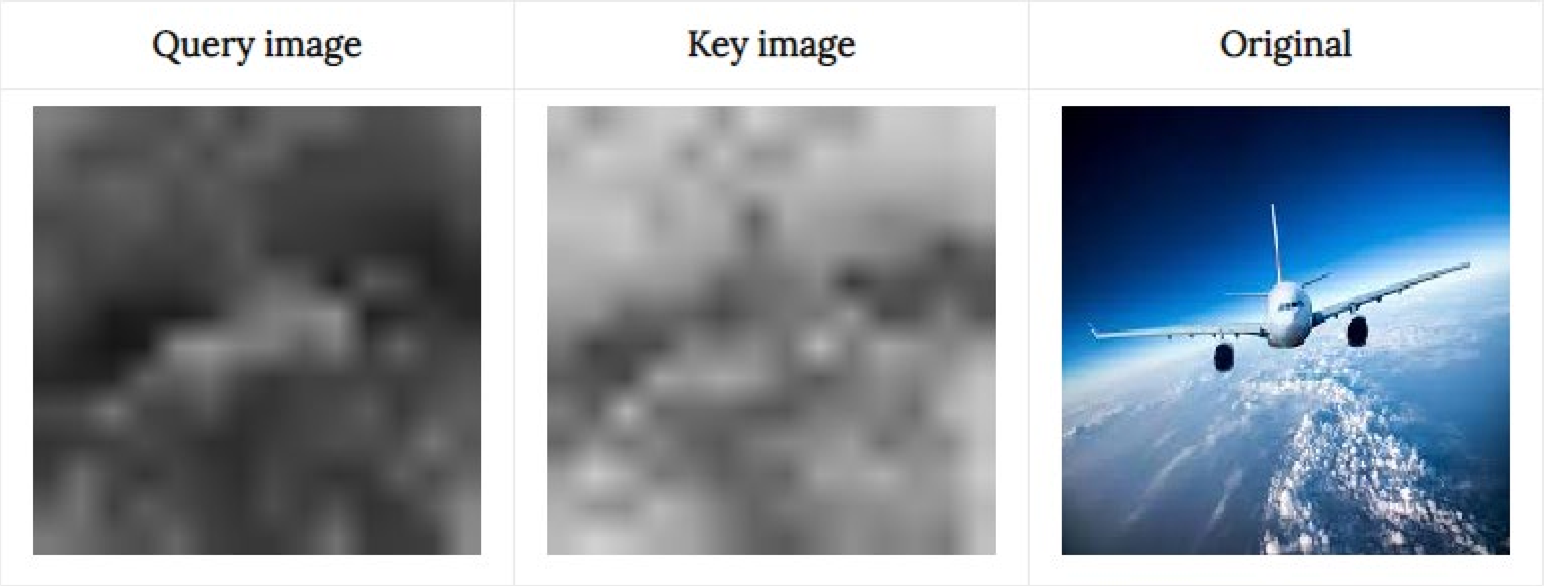
For most locations in the Query image, since they are positive, information is going to flow to them only from the positive locations in the Key image - that come from the Airplane.

Q, K here are telling us -

We found an airplane, and we want all the locations in the image to know about this!

Figure: https://jacobgil.github.io/deeplearning/vision-transformer-explainability

Attention maps



- The Query image highlights mainly the bottom part of the Airplane.
- The Key image is negative in the top part of the Airplane.

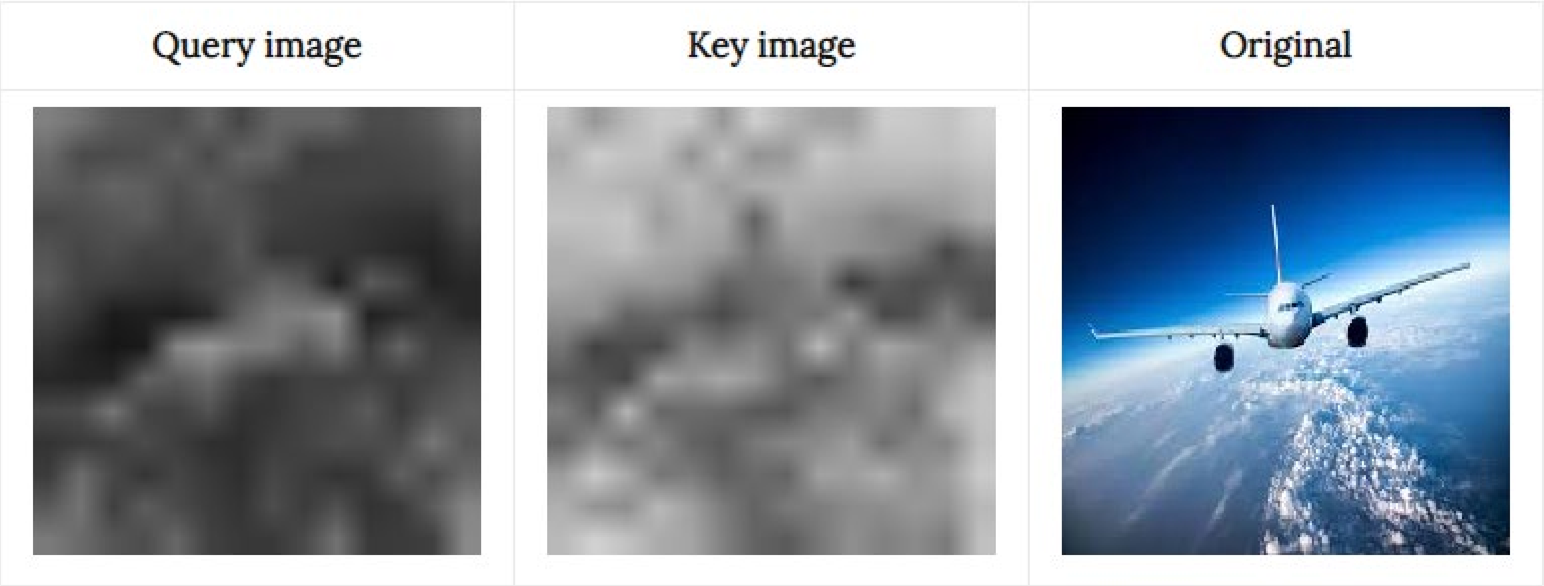
The information flows in two directions here: (1/2)

The top part of the plane (negative values in the Key) is going to spread into all the image (negative values in the Query).

Hey we found this plane, lets tell the rest of the image about it.

Figure: https://jacobgil.github.io/deeplearning/vision-transformer-explainability

Attention maps



- The Query image highlights mainly the bottom part of the Airplane.
- The Key image is negative in the top part of the Airplane.

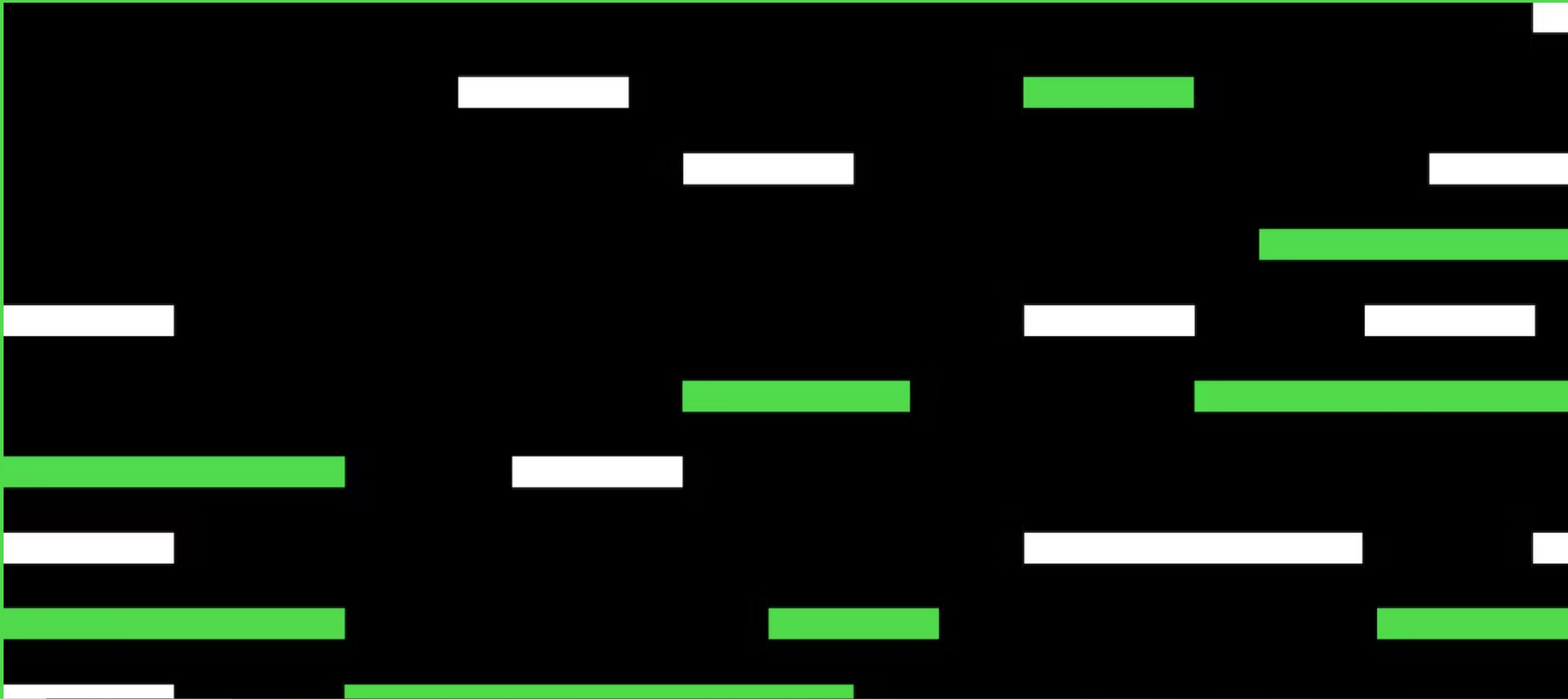
The information flows in two directions here: (2/2)

Information from the “Non Plane” parts of the image (positive values in the Key) is going to flow into the bottom part of the Plane (positive values in the Query).

Lets tell the plane more about what's around it.

Figure: https://jacobgil.github.io/deeplearning/vision-transformer-explainability

GPT-4



GPT

Improving Language Understanding by Generative Pre-Training

Alec Radford, OpenAI, alec@openai.com
Karthik Narasimhan, OpenAI, karthikn@openai.com
Tim Salimans, OpenAI, tim@openai.com
Ilya Sutskever, OpenAI, ilyasu@openai.com

Language Models are Unsupervised Multitask Learners

Alec Radford [*,1] Jeffrey Wu [*,1] Rewon Child [1] David Luan [1] Dario Amodei [**,1] Ilya Sutskever [**,1]

Language Models are Few-Shot Learners

Tom B. Brown[*] Benjamin Mann[*] Nick Ryder[*] Melanie Subbiah[*]

Jared Kaplan[†] Prafulla Dhariwal Arvind Neelakantan Pranav Shyam Girish Sastry

Amanda Askell Sandhini Agarwal Ariel Herbert-Voss Gretchen Krueger Tom Henighan

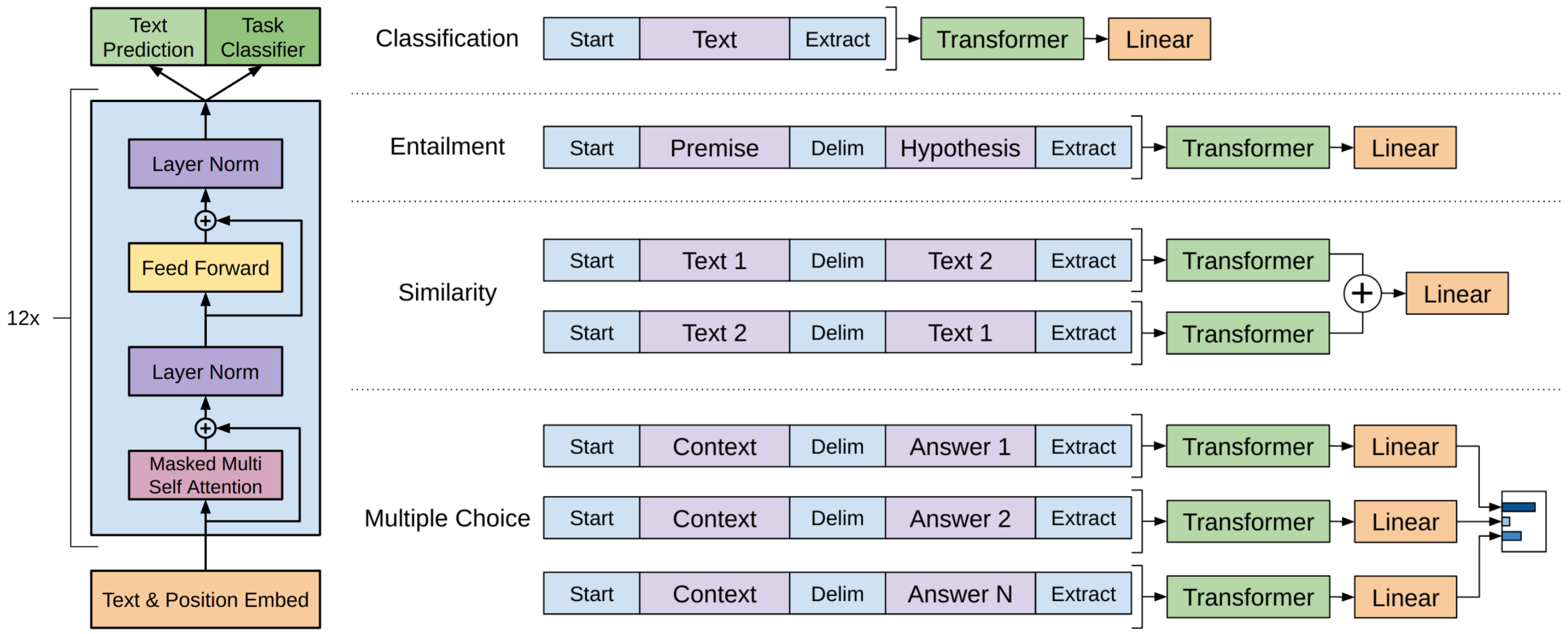
Rewon Child Aditya Ramesh Daniel M. Ziegler Jeffrey Wu Clemens Winter

Christopher Hesse Mark Chen Eric Sigler Mateusz Litwin Scott Gray

Benjamin Chess Jack Clark Christopher Berner

Sam McCandlish Alec Radford Ilya Sutskever Dario Amodei

OpenAI



Text Prediction
Task Classifier
12x
Layer Norm
Feed Forward
Layer Norm
Masked Multi Self Attention
Text & Position Embed
Classification
Start
Text
Extract
Transformer
Linear
Entailment
Start
Premise
Delim
Hypothesis
Extract
Transformer
Linear
Similarity
Start
Text 1
Delim
Text 2
Extract
Transformer
Start
Text 2
Delim
Text 1
Extract
Transformer
Linear
Multiple Choice
Start
Context
Delim
Answer 1
Extract
Transformer
Linear
Start
Context
Delim
Answer 2
Extract
Transformer
Linear
Start
Context
Delim
Answer N
Extract
Transformer
Linear

Unsupervised learning served as pre-training objective for supervised fine-tuned models, hence named Generative Pre-training

a. **Unsupervised Language Modelling** (Pre-training): For unsupervised learning, standard language model objective was used.

$$L_1(T) = \sum_i log\, P(t_i | t_{i-k}, \ldots, t_{i-1}; \theta) \qquad (i)$$

where T was the set of tokens in unsupervised data {t_1,…,t_n}, k was size of context window, θ were the parameters of neural network trained using stochastic gradient descent.

Unsupervised learning served as pre-training objective for supervised fine-tuned models, hence named Generative Pre-training

b. **Supervised Fine-Tuning:** This part aimed at maximising the likelihood of observing label y, given features or tokens x_1,…,x_n.

$$L_2(C) = \sum_{x,y} log\, P(y|x_1, \ldots, x_n) \qquad (ii)$$

$$L_3(C) = L_2(C) + \lambda L_1(C) \qquad (iii)$$

Unsupervised learning served as pre-training objective for supervised fine-tuned models, hence named Generative Pre-training

c. **Task Specific Input Transformations**: In order to make minimal changes to the architecture of the model during fine tuning, inputs to the specific downstream tasks were transformed into ordered sequences. The tokens were rearranged in following manner:

— Start and end tokens were added to the input sequences.

— A delimiter token was added between different parts of example so that input could be sent as ordered sequence. For tasks like question answering, multiple choice questions etc. multiple sequences were sent for each example. E.g. a training example comprised of sequences for context, question and answer for question answering task.

Reference

- https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a
- https://jacobgil.github.io/deeplearning/vision-transformer-explainability