

Computer Vision in Python

Day 2, Part 4/4

Matthieu De Mari



SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

About this lecture

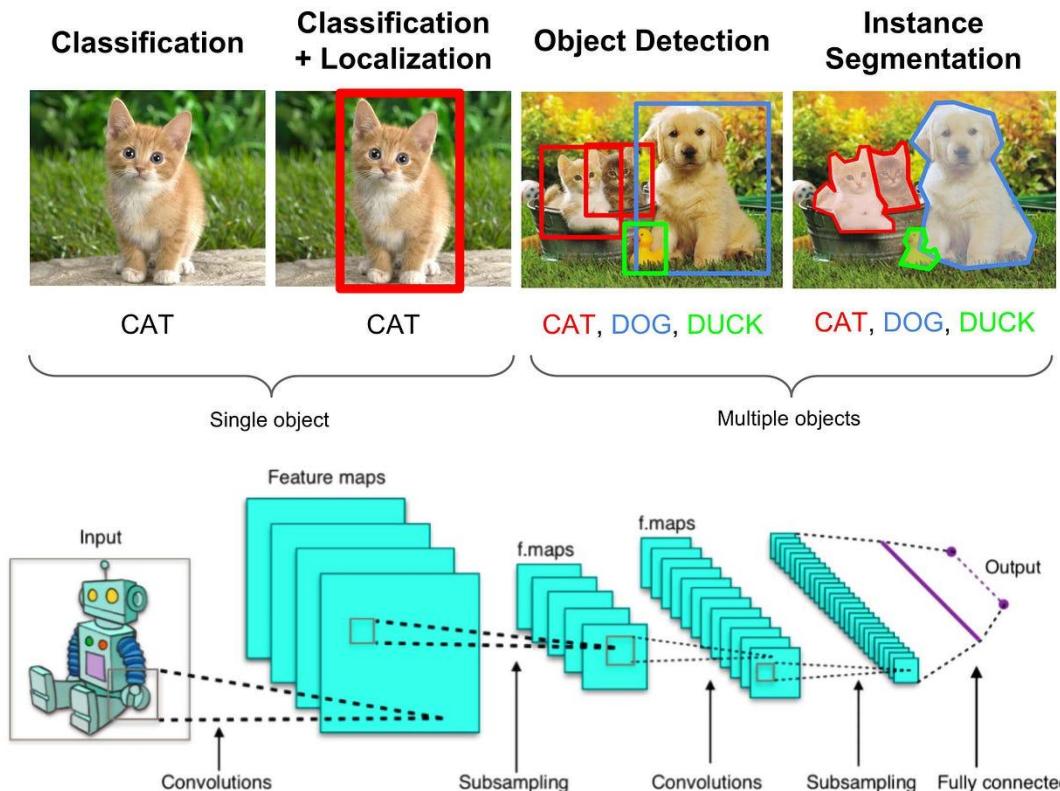
1. What are **Vision Transformers (ViTs)**?
2. How are **ViTs related to Convolutional Neural Networks** (previous day) and **(Natural Language Processing) Transformers** (briefly mentioned in the previous lecture)?
3. What is the **Swin model**?
4. What are **Hybrid ViT models**?

Reminders on Computer Vision and CNNs

As seen on Week 4, **CNNs can be used to recognize patterns in visual data**, e.g. image classification, segmentation, etc.

- CNNs do so by using local pattern recognition through convolutions.
- Hierarchical feature learning: edges → textures → objects.

CNN have produced milestone models for computer vision CV (e.g., AlexNet, ResNet, etc.) that are efficient, fast, and adaptable for smaller datasets.



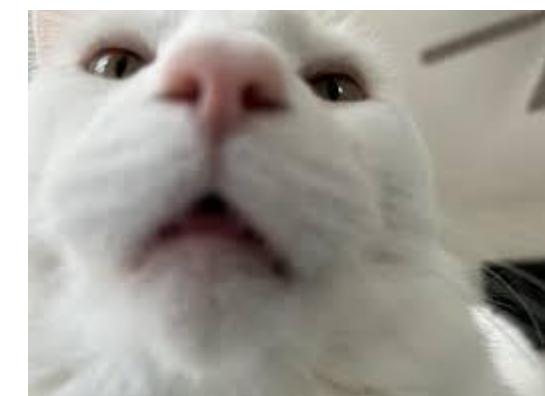
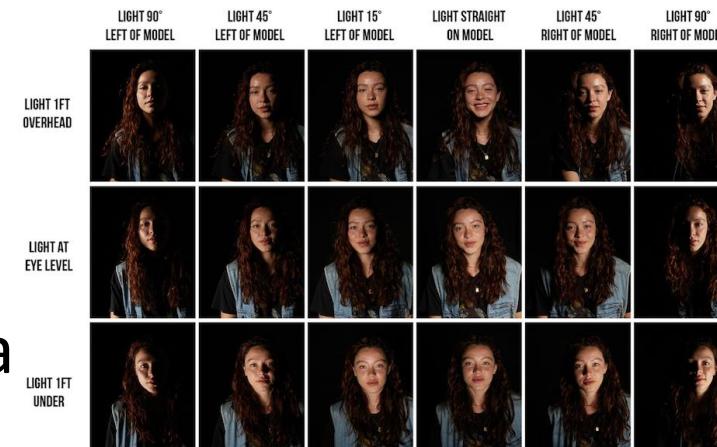
→ What is the issue then?

Limitations of Convolutional Neural Networks

Issue #1: Inductive Biases, i.e.
**CNNs assume patterns always
look the same.**

- What if the image is tilted, distorted, blurred, obstructed zoomed out/in, uses different lighting conditions, etc.?
- Might not work, even with extra tricks like data augmentation.

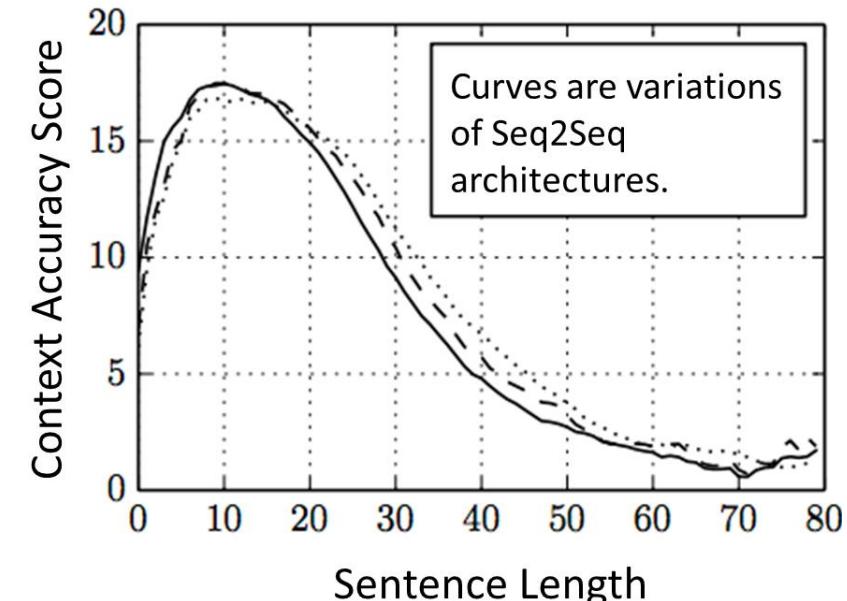
CNNs are not robust to these minor image variations!



Limitations of Convolutional Neural Networks

Issue #2: Global Context

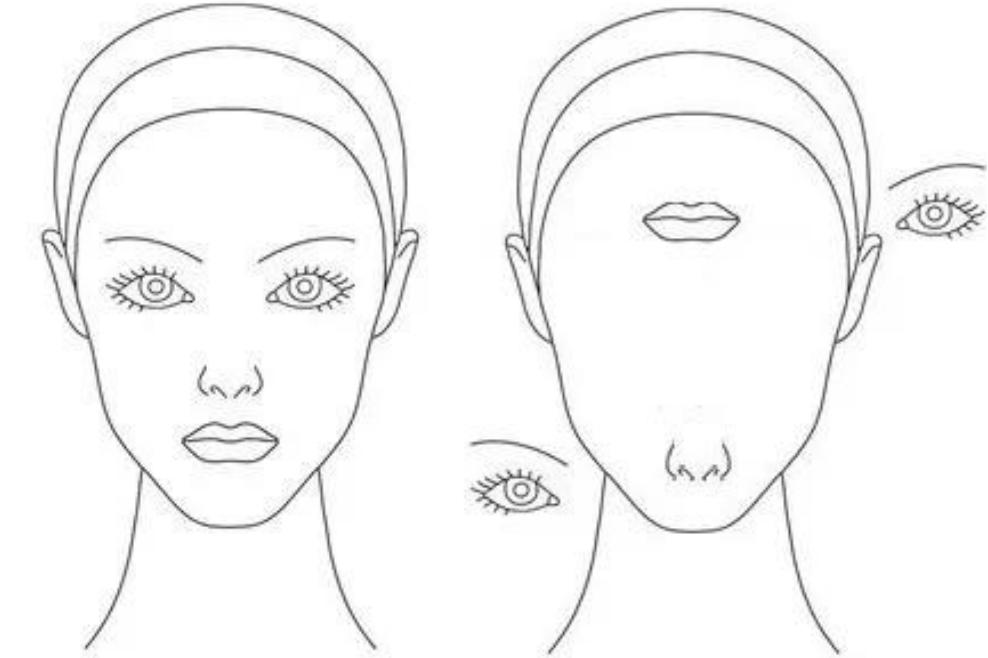
- **CNNs are great at spotting small local patterns (like edges or textures) but can miss the big picture** when features are large or far apart.
- Like with RNNs (previous lecture), there is a **limited “range” for which contextual information can be exchanged** using CNNs in an image.
- Convolution is too local to understand the big picture?



Limitations of Convolutional Neural Networks

Issue #3: No positional information

- High level neurons check if all features of a certain class of object are present. The process of checking whether features are present is done by striding the image.
- During this process, the CNN model completely loses all the information about the composition and position of the components.



CNNs will classify both images as a human face, but is the person on the right really human?

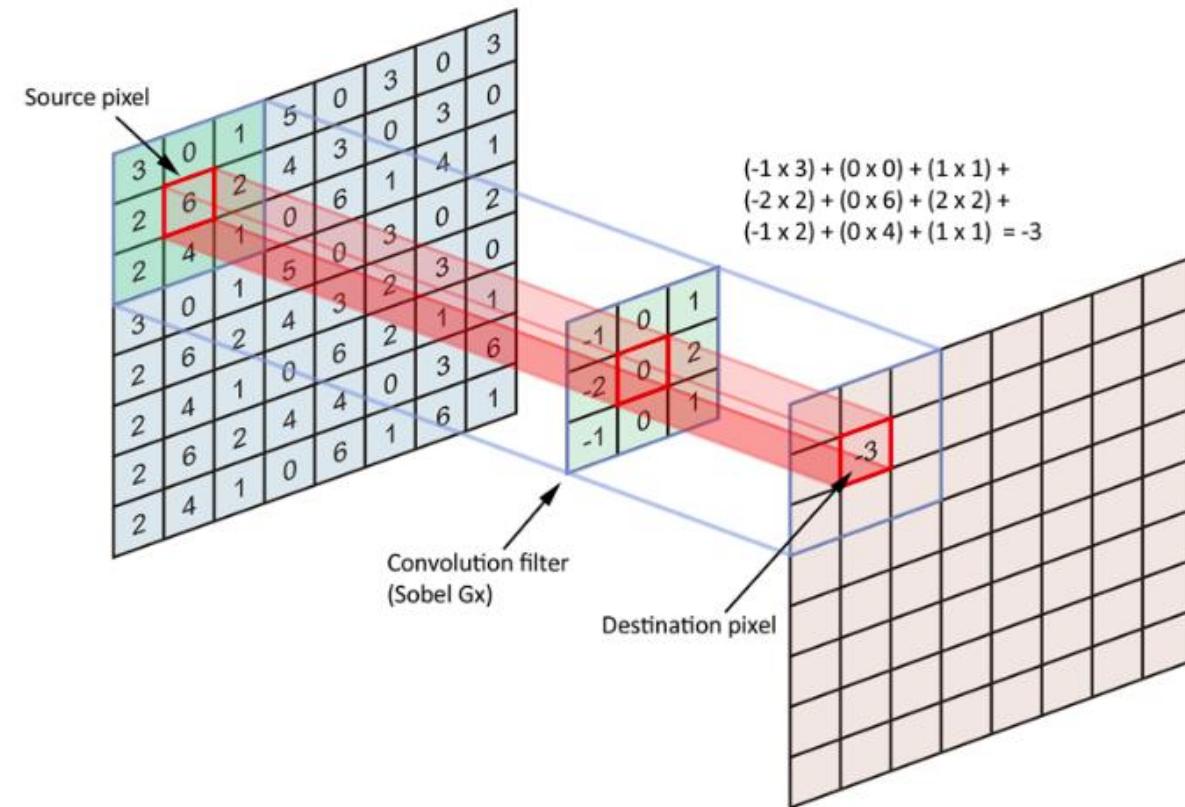
Limitations of Convolutional Neural Networks

Issue #4: Fixed Kernels

- Filters are fixed in size and structure when creating the model.
- Less flexibility for diverse tasks like object relationships.

It is time for us to investigate models that are able to resolve these issues!

→ Enter Vision Transformers



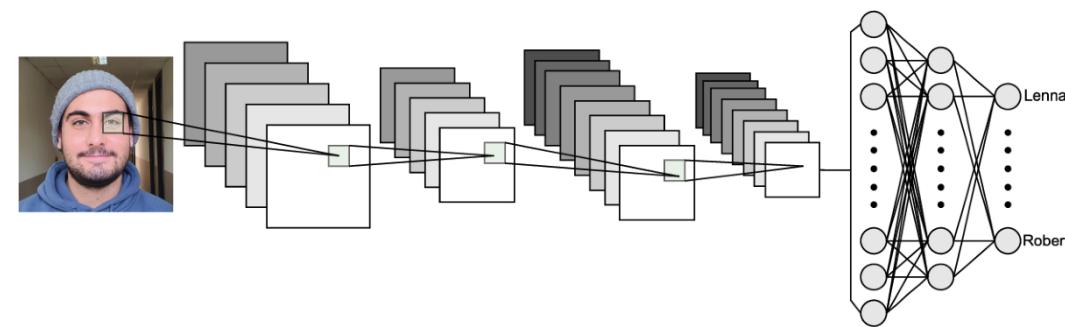
Why use Transformers in Vision?

As seen on W8S3, transformers excel in global context modelling:

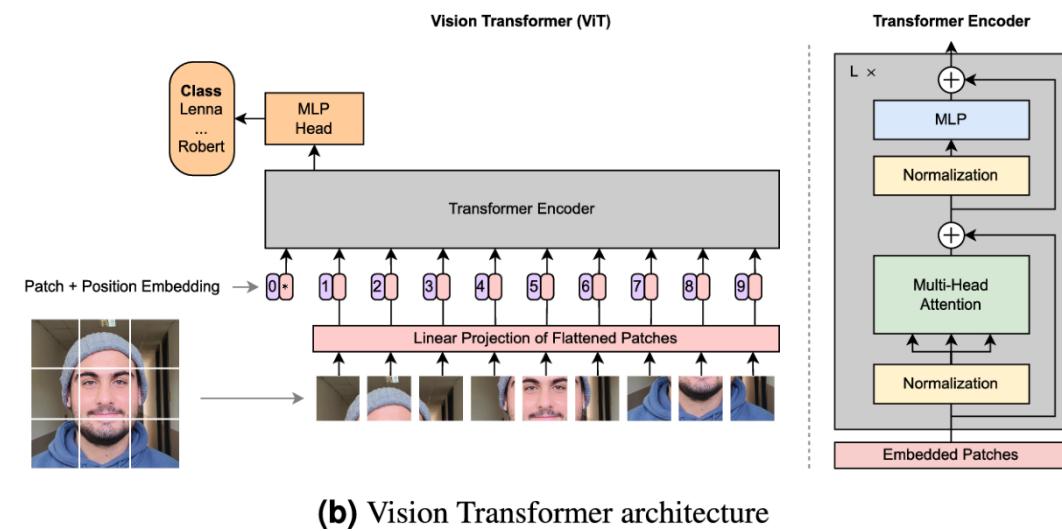
- Self-attention mechanisms can be used to connect every pixel to every other pixel.
- No locality bias as relationships between distant features are captured.

Benefits of self-attention:

- Adaptive focus across the entire input.
- Better understanding of complex object relationships and positional information.
- Proven success in language tasks (e.g., BERT, GPT) paved the way for visual tasks.



(a) Common CNN architecture

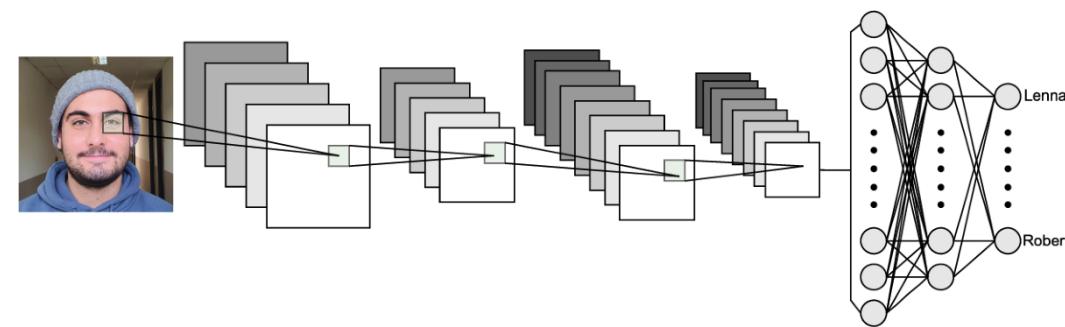


Historical Development of ViTs

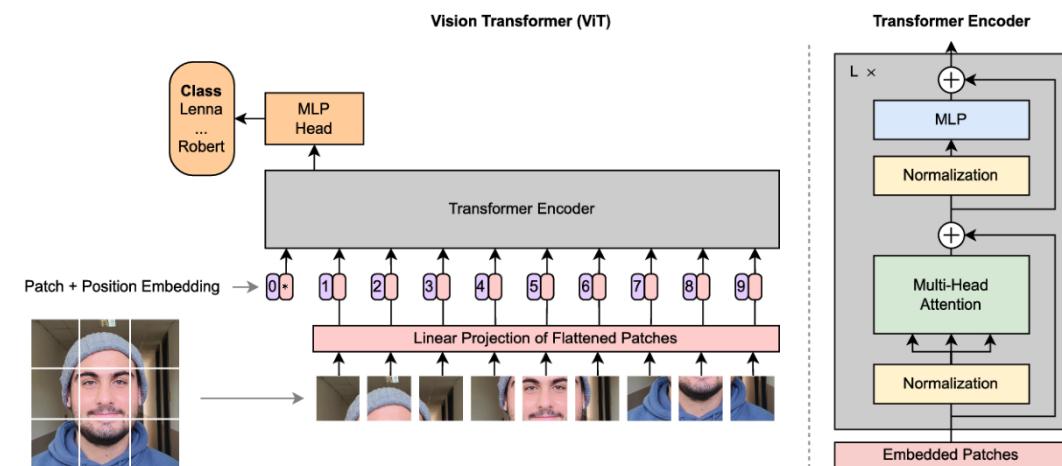
ViTs are a typical example of a certain AI field (here, NLP) producing a good idea (transformer), and the idea being adapted by another field (here, CV).

- 2017: Attention Is All You Need
→ Transformers dominate NLP.
- 2020: ViT were later introduced by Dosovitskiy et al., see [ViT2020].

Nowadays, **ViTs achieve state-of-the-art performance** in image classification (e.g., ImageNet), object detection and segmentation.



(a) Common CNN architecture

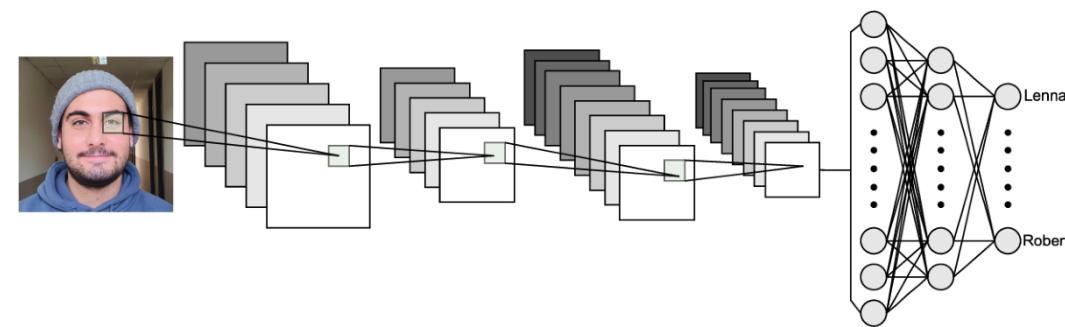


(b) Vision Transformer architecture

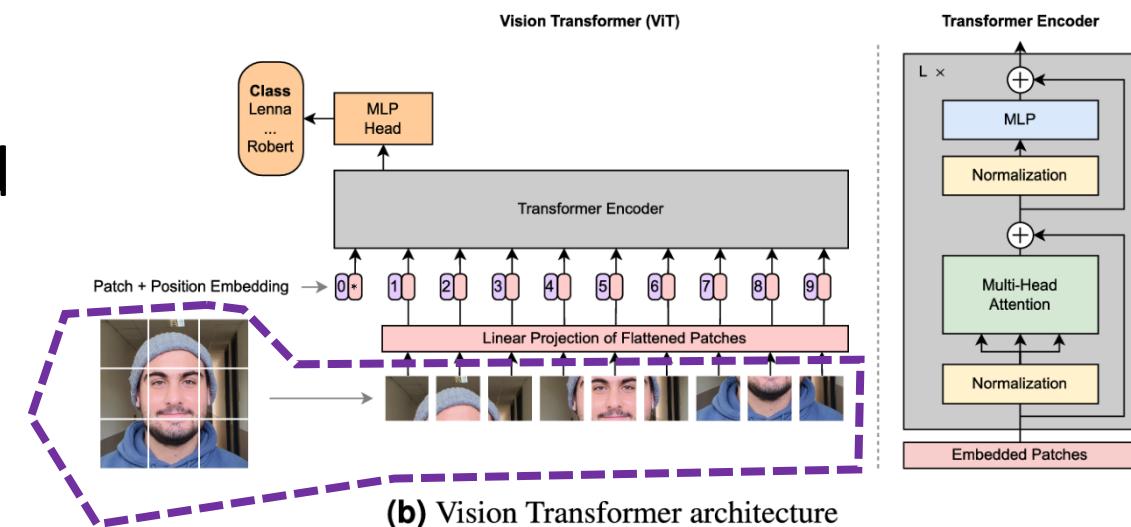
Implementation of a basic ViT

Basic steps taken by ViTs

- **Step 1:** Images split into patches, and each patch treated as tokens.
- **Step 2:** Positional encoding to retain positional information about the different patches of the image
- **Step 3:** Global dependencies modelled via transformers and attention layers instead of simple CNNs.



(a) Common CNN architecture

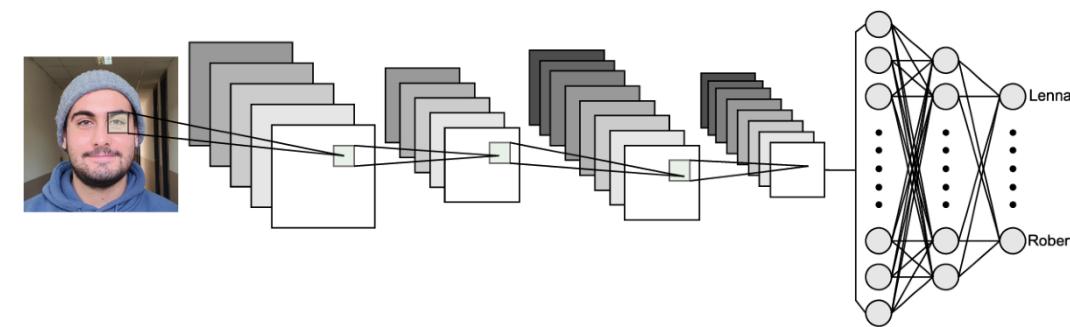


(b) Vision Transformer architecture

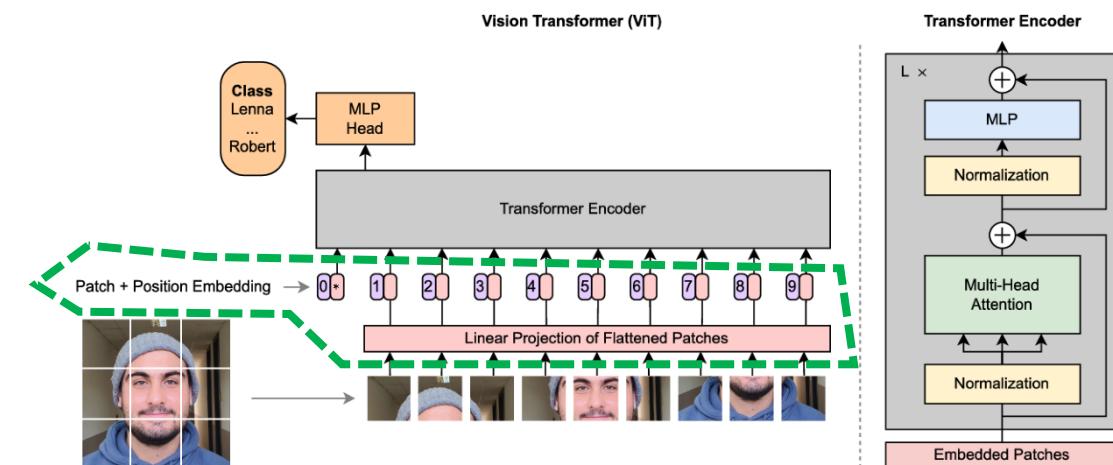
Implementation of a basic ViT

Basic steps taken by ViTs

- **Step 1:** Images split into patches, and each patch treated as tokens.
- **Step 2:** Positional encoding to retain positional information about the different patches of the image
- **Step 3:** Global dependencies modelled via transformers and attention layers instead of simple CNNs.



(a) Common CNN architecture

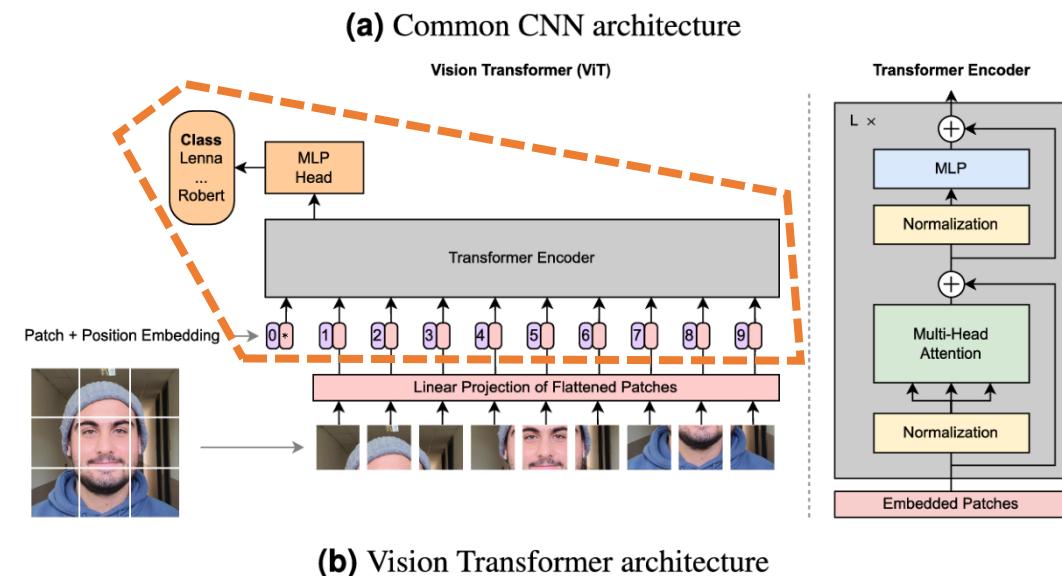
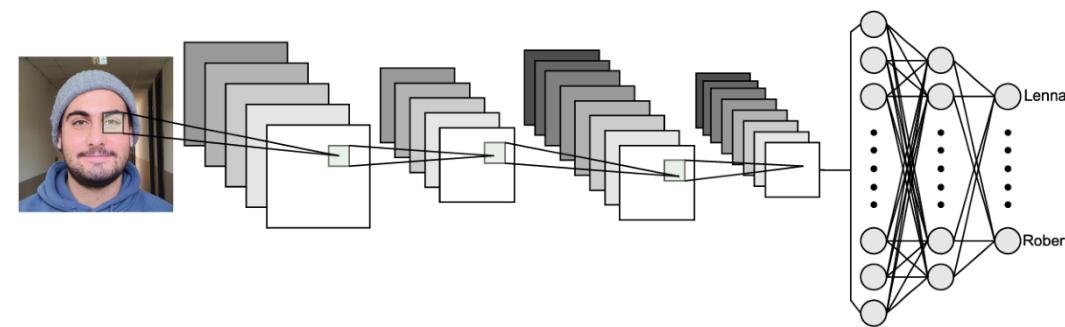


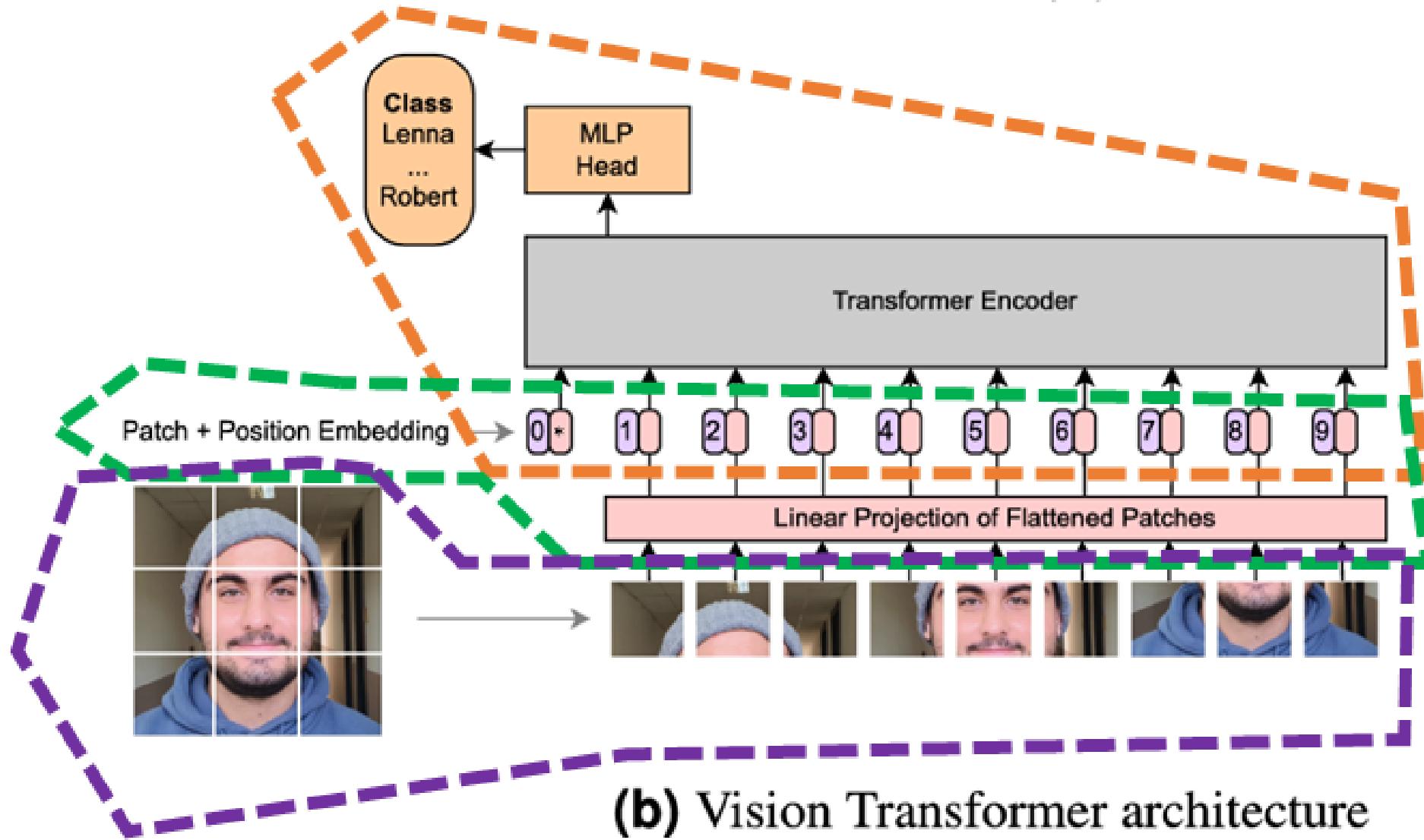
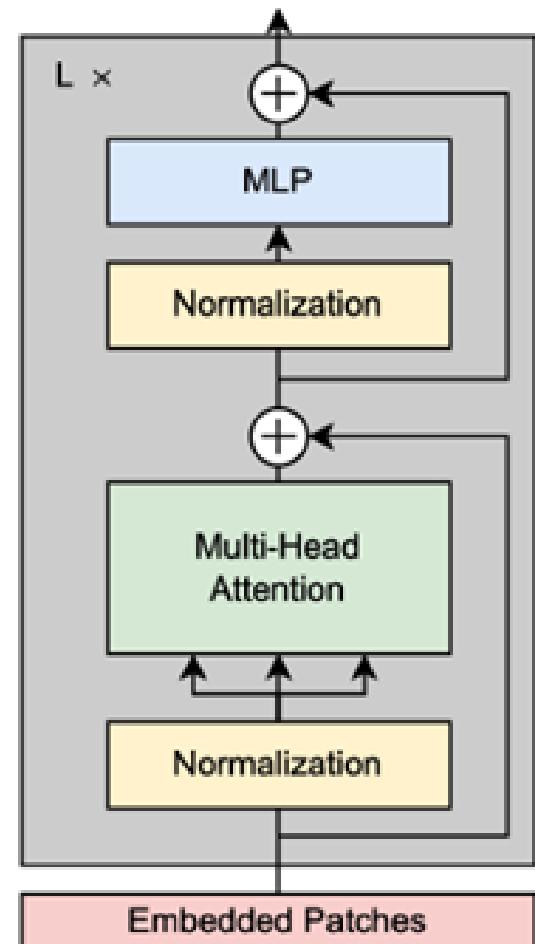
(b) Vision Transformer architecture

Implementation of a basic ViT

Basic steps taken by ViTs

- **Step 1:** Images split into patches, and each patch treated as tokens.
- **Step 2:** Positional encoding to retain positional information about the different patches of the image
- **Step 3:** Global dependencies modelled via transformers and attention layers instead of simple CNNs.



Vision Transformer (ViT)**Transformer Encoder****(b)** Vision Transformer architecture

Benefits of ViTs vs. CNNs

What are the benefits of ViTs vs. CNNs?

- **Global Context:** CNNs dominated computer vision but have limitations with global context. Transformers overcome these by leveraging self-attention. Vision Transformers treat images as sequences of patches, enabling global dependency modelling.
- **Improved Scalability:** ViTs perform better as data size increases.
- **State-of-the-art Performance:** Similar or superior performance to CNNs on large-scale datasets.
- **Model Flexibility:** Unified framework for text and vision tasks. Can later be expanded to video, 3D vision, and multi-modal tasks.

Splitting images into patches

Step 1: Images split into patches, and each patch treated as tokens.

Definition (image patch): The first step requires to cut the image into smaller, fixed-sized square pieces (e.g., 16x16 pixels). These pieces are called **patches**. Each patch holds a portion of the image.

In our demo notebook, we use CIFAR-10 (32 by 32 images) and split it into 16 patches of size 8 by 8.



Original Image

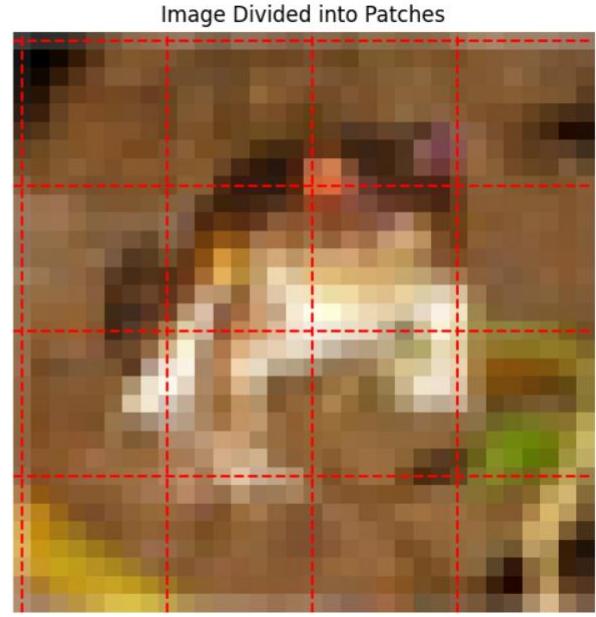


Image Divided into Patches

Linear Embeddings

Step 1: Images split into patches, and each patch treated as tokens.

Then, we **transform the patches into linear embeddings**, by:

- Using a Conv2d on each patch,
- And then flatten to obtain a 1D vector of size 64, this is the **linear embedding** for the patch.

```
# Define Patch Embedding function
class PatchEmbedding(nn.Module):
    def __init__(self, img_size = 32, patch_size = 8, \
                 in_channels = 3, embed_dim = 64):
        super().__init__()
        self.patch_size = patch_size
        self.projection = nn.Conv2d(in_channels, embed_dim, \
                                   kernel_size = patch_size, \
                                   stride = patch_size)

    def forward(self, x):
        # Convert image to patches
        x = self.projection(x)
        # Flatten and rearrange
        patches = x.flatten(2).transpose(1, 2)
        return patches
```

```
# Select the first image of the dataset
# We use unsqueeze, to add the batch dimension,
# our image is then a tensor of size (1, 3, 32, 32)
image, label = dataset[0]
image = image.unsqueeze(0)

# Create Patch Embedding module
patch_embedding = PatchEmbedding(img_size = 32, patch_size = 8, \
                                  in_channels = 3, embed_dim = 64)

# Apply the patch embedding
patches = patch_embedding(image)
# Patches are tensors of size (batch_size = 1, num_patches = 16, embed_dim = 64)
print(f"Patches Shape: {patches.shape}")

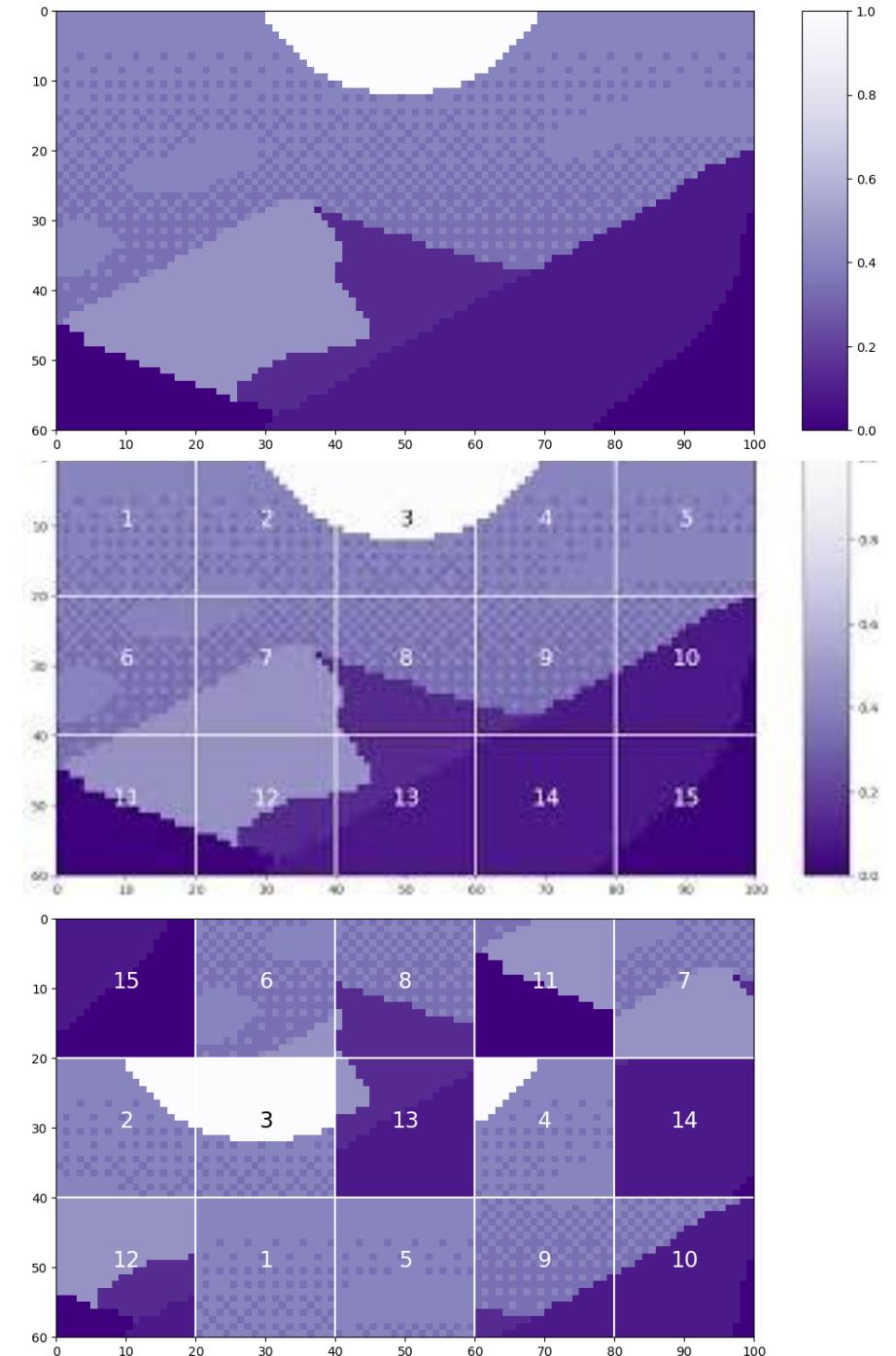
Patches Shape: torch.Size([1, 16, 64])
```

Positional Encoding

- **Step 2:** Positional encoding to retain positional information about the different patches of the image

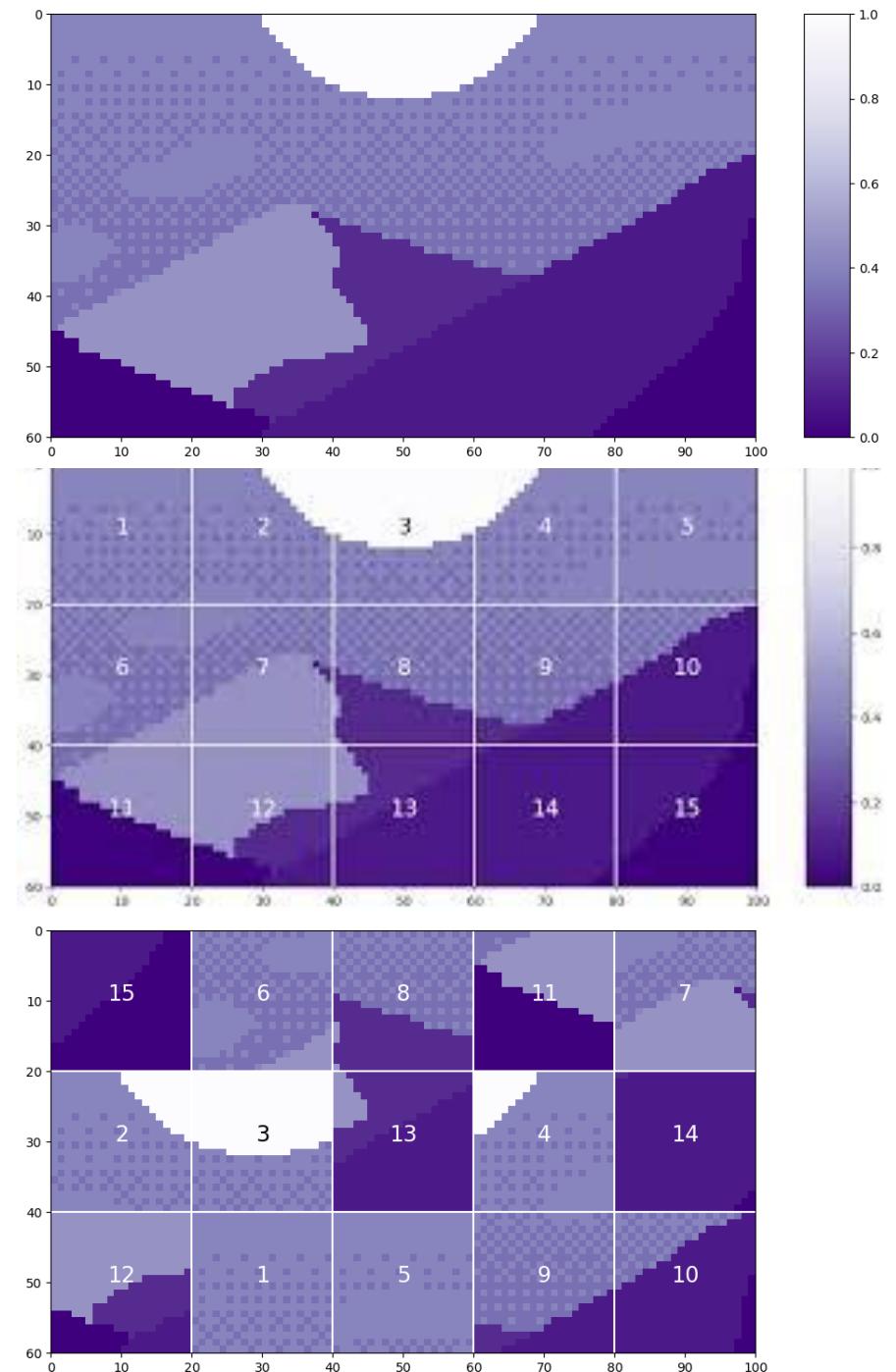
Why is position important?

- In images, the arrangement of pixels defines the structure (e.g., a cat's ears are at the top, not in the middle of their body).



Positional Encoding

- Transformers do not naturally understand positioning for the different patches
- They might treat patches as an unordered sequence.
- Without positional encoding, transformers would not know the difference between a cat and a scrambled cat puzzle, which could prove problematic to understand the global context of the image.



Positional Encoding

Definition (positional encoding**):**

Positional encoding is a piece of information that is added to each patch embedding to give it a unique location identity.

This helps the transformer know where the patch originally comes from in the image.

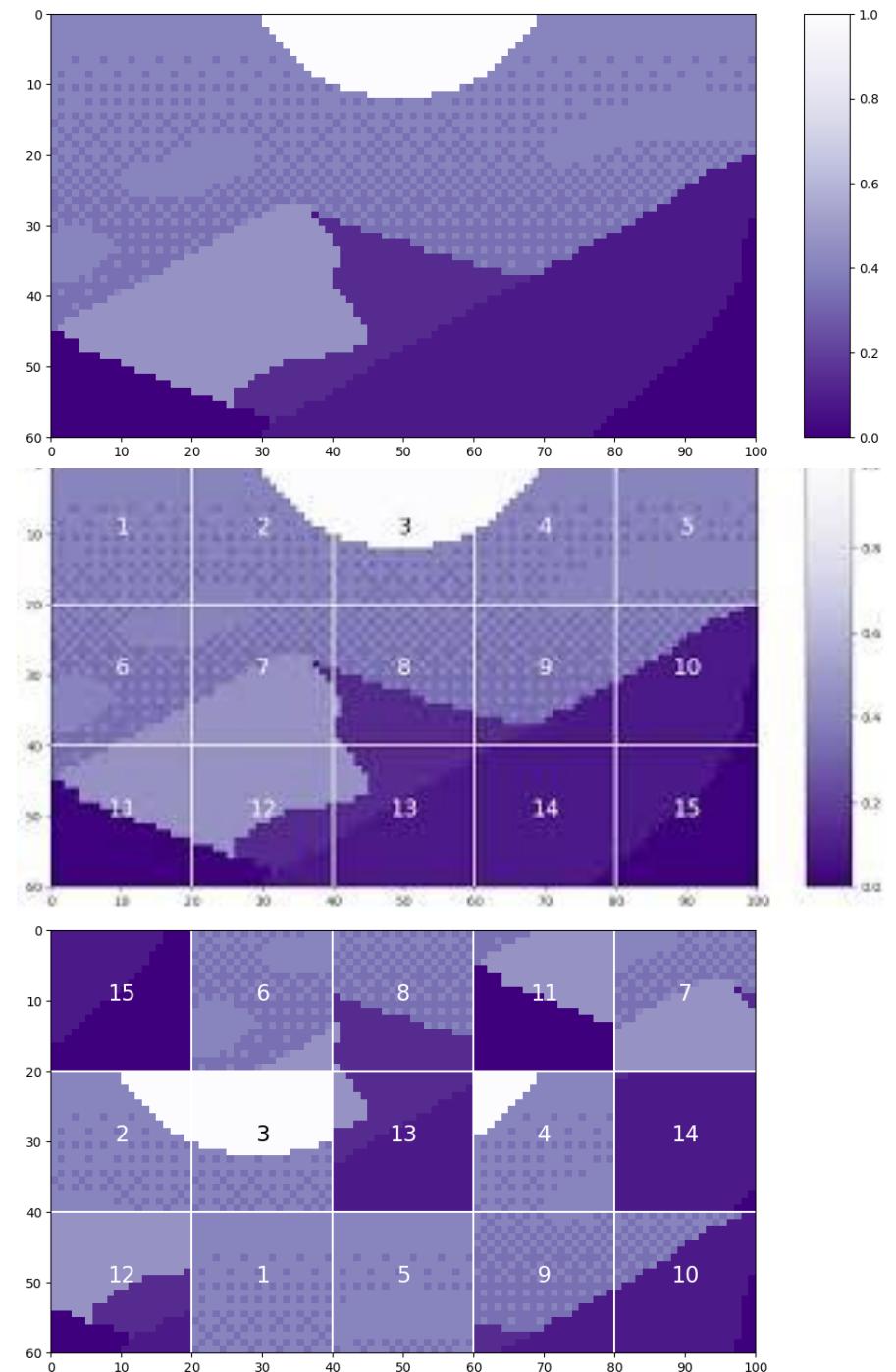
It then prevents the “scrambled puzzle” effect mentioned earlier.

Positional encodings are usually vectors added to the patch embeddings. They are often defined as sinusoidal functions (similar to waves) or as learned embeddings.

Positional Encoding

Comparison between CNNs and ViTs:

- **CNNs:** Position is “hard-coded” by the sliding convolutional filters, which inherently respect spatial structure, but is often “too local”.
- **ViTs:** Explicitly add positional encoding information to patches because they lack built-in spatial understanding, attention layers will then allow for a global context understanding.



Sinusoidal Positional Encoding

What do we want for a positional encoding?

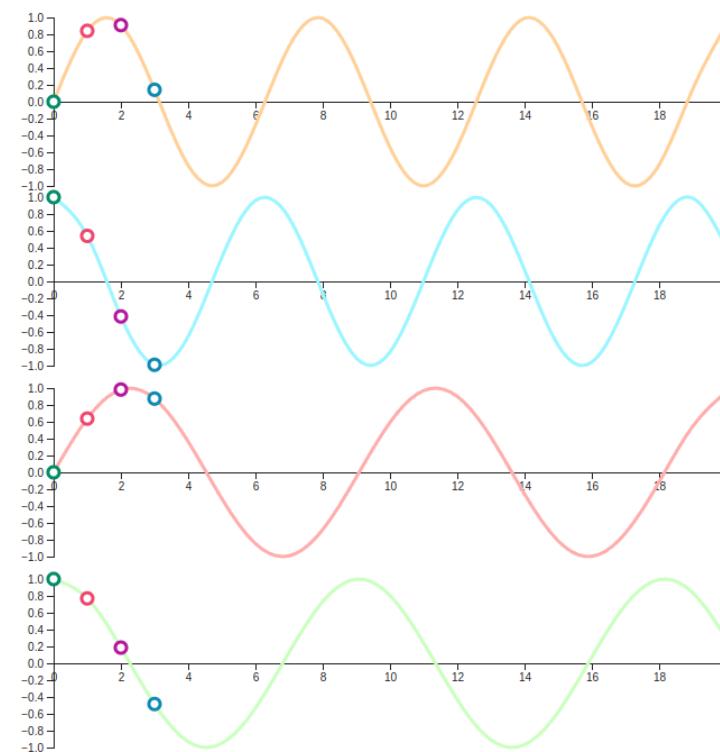
- **“Kind-of” Orthogonality:** Patches should have “orthogonal” encodings.
- **Absolute Information:** The encoding should provide absolute positional information (e.g., "this is position 5").
- **Relative Information:** The encoding should help capture relative relationships between positions (e.g., "position 5 is close to 6").
- **Similarity Between Close Positions:** Positions close to each other should have encodings with small differences, allowing the model to understand local relationships.

Sinusoidal Positional Encoding

Definition (Sinusoidal functions for Positional Encoding):

Sinusoidal positional encodings are a way to inject **positional information** into a sequence of embeddings so that a transformer model can distinguish the order or spatial arrangement of the data.

Sinusoidal positional encodings **satisfy to many properties we want for positional encodings.**



	p0	p1	p2	p3	i=0
0.000	0.841	0.909	0.141		i=1
1.000	0.540	-0.416	-0.990		i=2
0.000	0.638	0.983	0.875		i=3
1.000	0.770	0.186	-0.484		

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Settings: $d = 50$
The value of each positional encoding depends on the *position (pos)* and *dimension (d)*. We calculate result for every *index (i)* to get the whole vector.

Sinusoidal Positional Encoding

How does it work?

$$\phi(i, k) = \begin{cases} \sin\left(\frac{i}{10000^{\frac{k}{D}}}\right) & \text{if } k \text{ is even} \\ \cos\left(\frac{i}{10000^{\frac{k}{D}}}\right) & \text{if } k \text{ is odd} \end{cases}$$

Where i is the patch index (in our case, $i \in [0, 15]$ as have 16 patches)

And D is the size of the positional encoding vector (we set it to be the same as our linear embeddings earlier, i.e. 64)

And $k \in [0, D] = [0, 63]$.

Sinusoidal Positional Encoding

How does it work?

Later on, we add this positional encoding to our embedding vectors.

$$\psi(i, k) = \theta(i, k) + \phi(i, k)$$

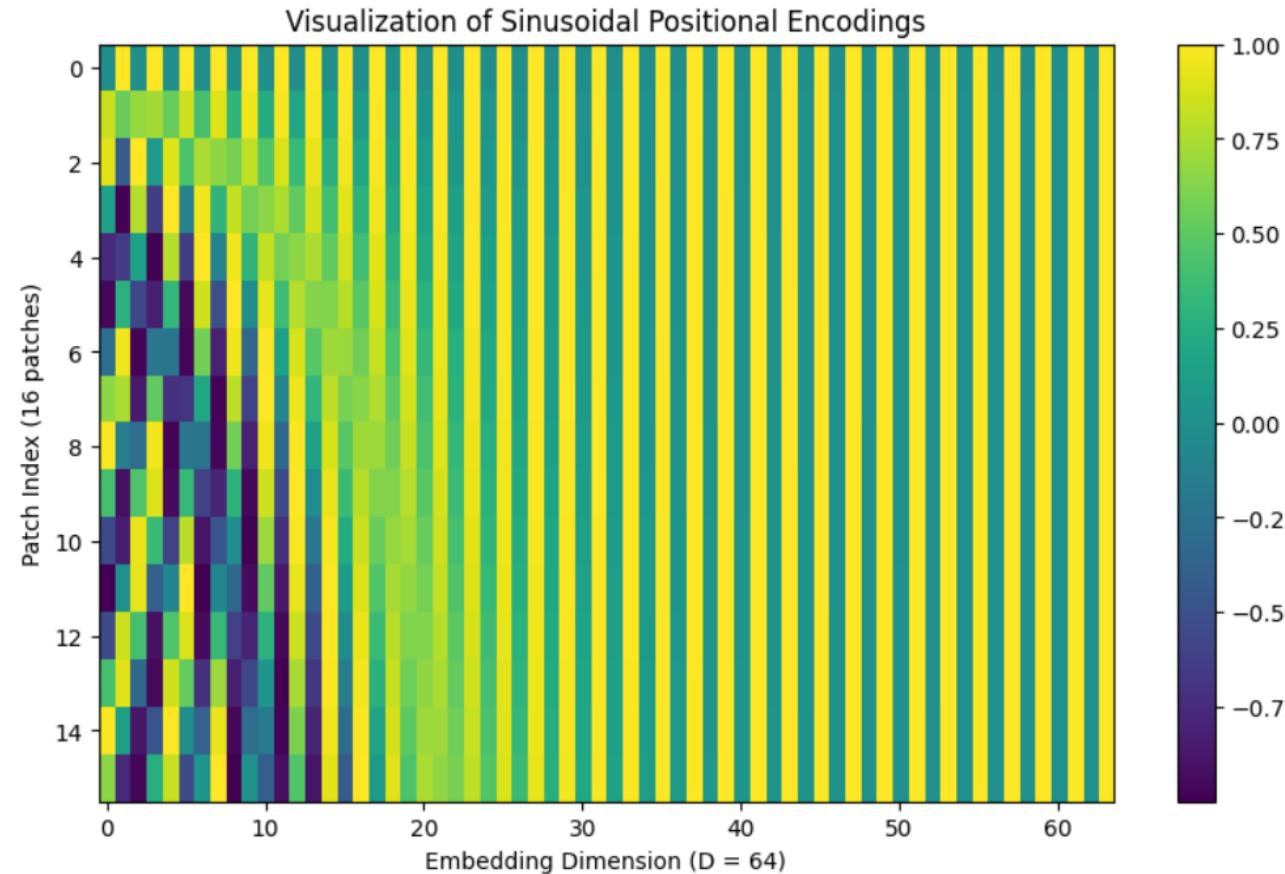
- Where $\theta(i, k)$ is the k -th element of the embedding vector produced by our PatchEmbedding class from earlier for patch i .
- And $\phi(i, k)$ is the k -th element of the positional encoding vector for patch i from the previous slide.
- And $\psi(i, k)$ is the k -th element of the vector combining embeddings and positional encoding for patch i .

Why use this sinusoidal formula? See [KN].

Sinusoidal Positional Encoding

Remember, we want our positional encodings to have certain properties.

- **“Kind-of” Orthogonality:** Patches should have different encodings.
- **Absolute Information:** The encoding should provide absolute positional information (e.g., "this is position 5").



A very common pattern for sinusoidal embeddings

Sinusoidal Positional Encoding

- **Relative Information:** The encoding should help capture relative relationships between positions (e.g., "position 5 is close to 6").
- **Similarity Between Close Positions:** Positions close to each other should have encodings with small differences, allowing the model to understand local relationships.

```
def cosine_similarity(vec1, vec2):  
    # Helper function to calculate cosine similarity  
    dot_product = np.dot(vec1, vec2)  
    norm1 = np.linalg.norm(vec1)  
    norm2 = np.linalg.norm(vec2)  
    return dot_product/(norm1*norm2)  
  
# Calculate cosine similarity between specified patches  
similarity_5_6 = cosine_similarity(positions[5], positions[6])  
similarity_5_7 = cosine_similarity(positions[5], positions[7])  
similarity_5_8 = cosine_similarity(positions[5], positions[8])  
similarity_6_7 = cosine_similarity(positions[6], positions[7])  
similarity_7_8 = cosine_similarity(positions[7], positions[8])  
  
print(f"Cosine Similarity between patch 5 and 6: {similarity_5_6:.4f}")  
print(f"Cosine Similarity between patch 5 and 7: {similarity_5_7:.4f}")  
print(f"Cosine Similarity between patch 5 and 8: {similarity_5_8:.4f}")  
print(f"Cosine Similarity between patch 6 and 7: {similarity_6_7:.4f}")  
print(f"Cosine Similarity between patch 7 and 8: {similarity_7_8:.4f}")  
  
Cosine Similarity between patch 5 and 6: 0.9662  
Cosine Similarity between patch 5 and 7: 0.8845  
Cosine Similarity between patch 5 and 8: 0.7996  
Cosine Similarity between patch 6 and 7: 0.9662  
Cosine Similarity between patch 7 and 8: 0.9662
```

From 1D to 2D sinusoidal encodings

- In practice, it appears that the positional encodings for patches k and $k + 1$ are close to each other, which is nice.
 - But in practice, our image has a 2D layout of patches.
 - This means that patches 3 and 4 should not be close to each other, but 0 and 4 should.
 - Therefore, we need 2D positional encodings
- Lucky for us, it relies on the same intuition as before with sinusoidal encodings.



From 1D to 2D sinusoidal encodings

Use separate row and column representations for patches:

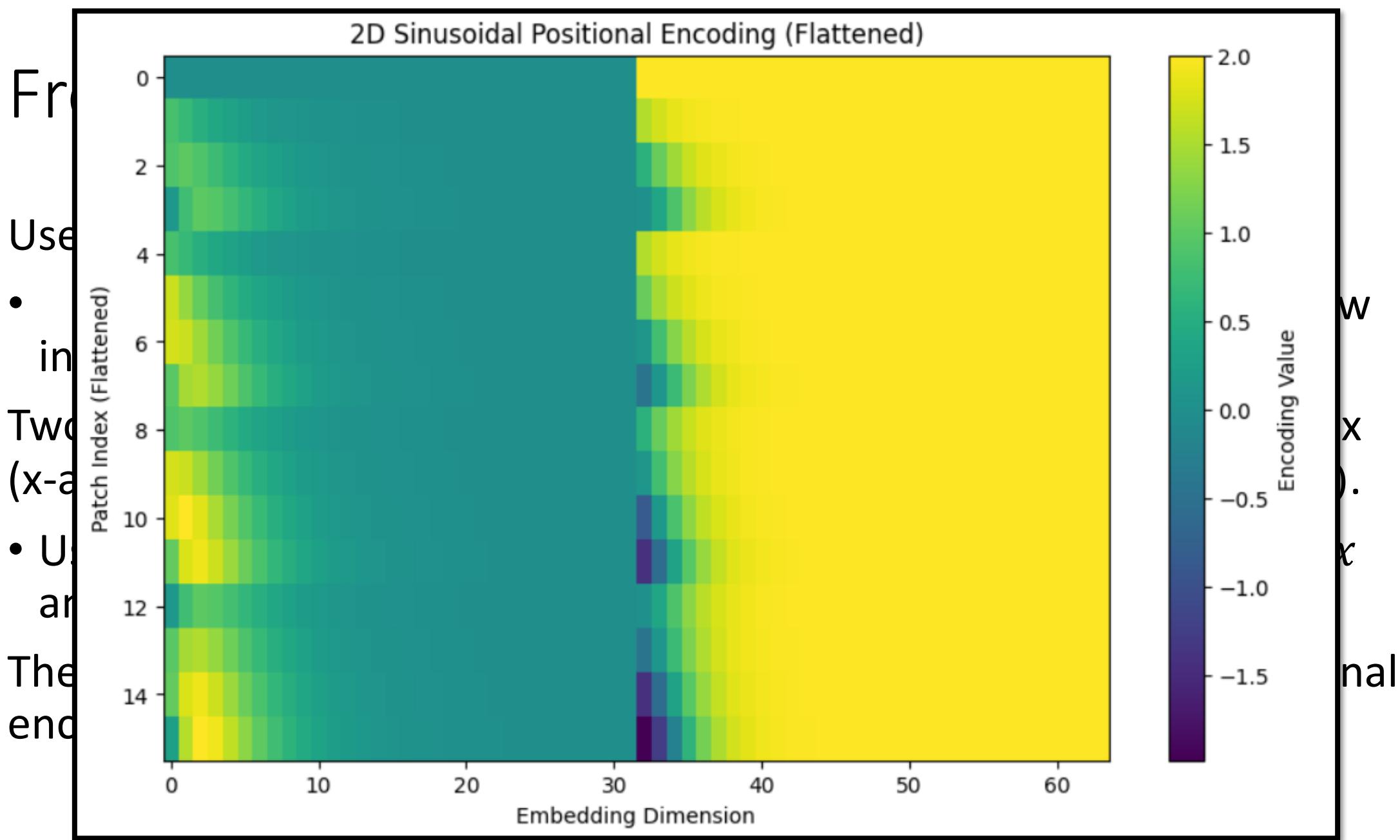
- Each patch i in a grid can now be equivalently identified by its row index and column index (x, y) .

Two separate positional encodings are created: One for the row index (x-axis) denoted $\phi_x(k)$, and one for the column index (y-axis), $\phi_y(k)$.

- Use sinusoidal positional encodings, as before, but this time using x and y instead of i .

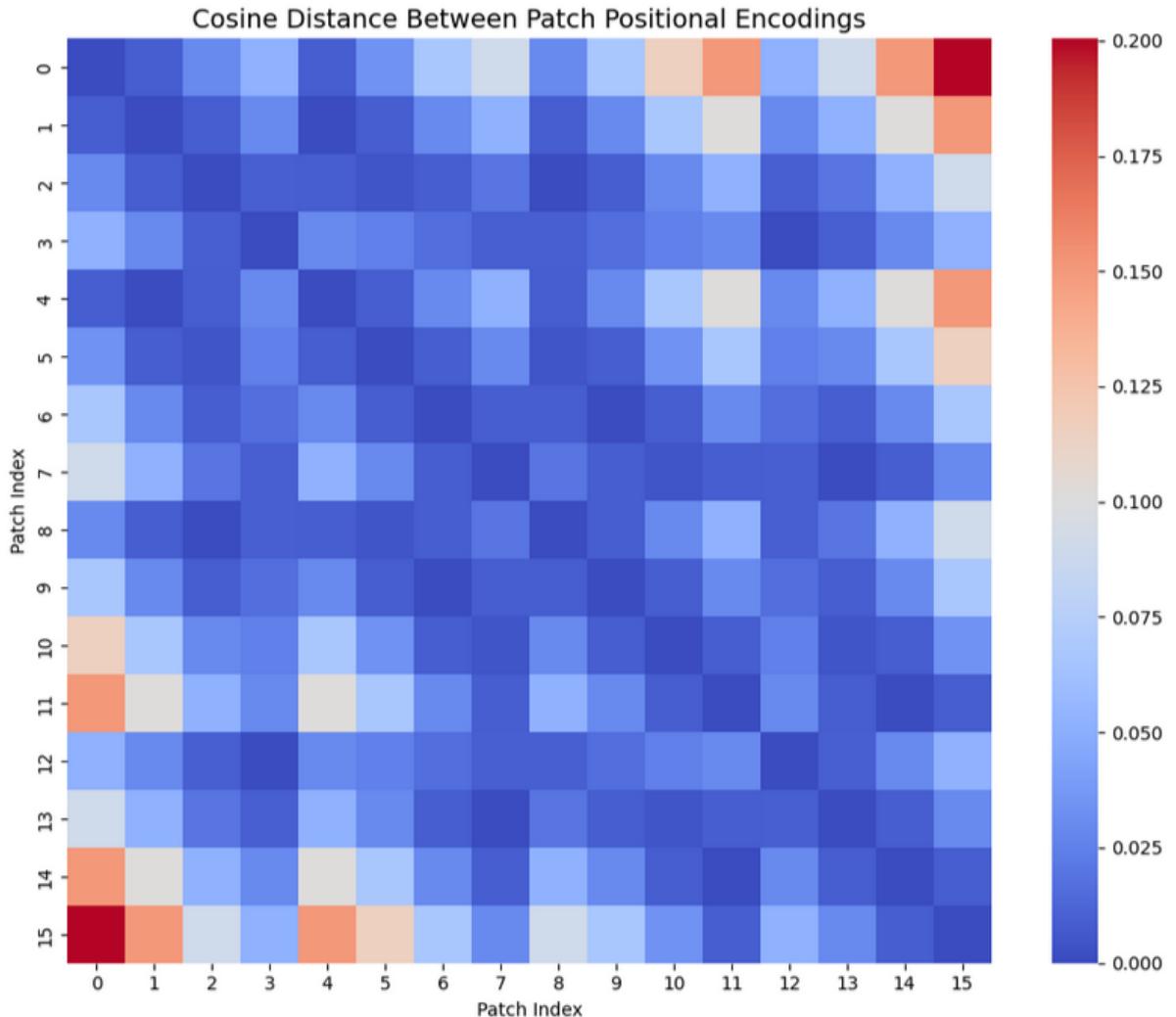
These encodings are summed, as before, to form the final 2D positional encoding $\psi(i, k)$ for patch i .

$$\psi(i, k) = \theta(i, k) + \phi_x(k) + \phi_y(k)$$



From 1D to 2D sinusoidal encodings

- It is then possible to verify that the 2D positional encodings now preserve the relative distance in 2D as expected.
- This can be observed using the heatmap on the side.
- For instance, patch 0 is similar to patches 1 and 4, as expected.
- *(It comes from the demo notebook 1).*



A simple ViT model

- **Step 3:** Global dependencies modelled via transformers and attention layers instead of simple CNNs.

We do so by creating a model

- That receives the patches embeddings, along with the 2D sinusoidal pos. encodings,
- And implements some simple attention layers

This time, we are not coding the transformer from scratch:

- Using the PyTorch transformer objects for attention layers,
- Finish with a single Linear layer for predictions of classes.
- Using GELU, dropout, etc.

```
class ViT(nn.Module):
    def __init__(self, embed_dim, num_patches, num_classes, num_heads, num_layers, mlp_dim, dropout = 0.1):
        super(ViT, self).__init__()
        self.num_patches = num_patches
        # Class token
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        # Transformer Encoder, using PyTorch prototypes for transformers
        self.encoder = nn.TransformerEncoder(nn.TransformerEncoderLayer(d_model = embed_dim,
                                                                        nhead = num_heads,
                                                                        dim_feedforward = mlp_dim,
                                                                        dropout = dropout,
                                                                        activation = 'gelu',
                                                                        batch_first=True),
                                            num_layers=num_layers)
        # Classification head
        self.mlp_head = nn.Sequential(nn.LayerNorm(embed_dim),
                                     nn.Linear(embed_dim, num_classes))

    def forward(self, x):
        batch_size, num_patches, embed_dim = x.shape
        cls_tokens = self.cls_token.expand(batch_size, -1, -1)
        x = torch.cat((cls_tokens, x), dim = 1)
        x = self.encoder(x)
        cls_output = x[:, 0]
        logits = self.mlp_head(cls_output)
        return logits
```

Preparing the model for training

- We will combine the processor (patches, positional encoding) with the model. Feel free to play with parameters.

```
# Parameters for model
img_size = 32
patch_size = 8
embed_dim = 64
num_patches = (img_size//patch_size)**2
num_classes = 10
num_heads = 2
num_layers = 2
mlp_dim = 128

# Vision Transformer Processor for Patches and Encodings
vi_processor = VisionTransformerProcessor(img_size = img_size, patch_size = patch_size, embed_dim = embed_dim, device = device)

# Vision Transformer Model
model = ViT(embed_dim = embed_dim, num_patches = num_patches, num_classes = num_classes,
            num_heads = num_heads, num_layers = num_layers, mlp_dim = mlp_dim).to(device)
```

Training loop

Finally, prepare a training loop, very similar to the ones we have used so far

- CrossEntropyLoss for the 10 classes of CIFAR-10
- Adam Optimizer, basic hyperparameters
- The losses are coming down, indicating there might be training. Need evaluation.

```
R # Hyperparameters
epochs = 100
lr = 1e-3

# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr = lr)

# Training Loop
def train_model(model, dataloader, criterion, optimizer, epochs):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            # Generate patch embeddings with positional encodings
            patch_embeddings = vi_processor.process_images(images)
            # Forward pass
            outputs = model(patch_embeddings)
            loss = criterion(outputs, labels)
            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch [{epoch + 1}/{epochs}], Loss: {running_loss / len(dataloader):.4f}")

# Train the model
# (Takes a while as transformers have Lots of trainable parameters!)
train_model(model, train_loader, criterion, optimizer, epochs)

Epoch [1/100], Loss: 2.0053
Epoch [2/100], Loss: 1.8314
Epoch [3/100], Loss: 1.7352
Epoch [4/100], Loss: 1.6735
Epoch [5/100], Loss: 1.6252
Epoch [6/100], Loss: 1.5944
Restructured [7/100] 'loss' to 'lss'
```

Training loop

In practice, ViTs need many more iterations than CNNs to train, which is normal given that they have way more parameters!

- Post-training accuracy after 100 epochs: 59% (meh...).

What could be the issue here?

```
# Evaluation
def evaluate_model(model, dataloader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            # Generate patch embeddings with positional encodings
            patch_embeddings = vi_processor.process_images(images)
            # Forward pass
            outputs = model(patch_embeddings)
            preds = torch.argmax(outputs, dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
    # calculate accuracy
    accuracy = accuracy_score(all_labels, all_preds)
    return accuracy
```

```
# Evaluate the model
test_accuracy = evaluate_model(model, test_loader)
print(f"Test Accuracy: {test_accuracy*100:.2f}%")
```

Test Accuracy: 56.76%

Training loop

What could be the issue here?

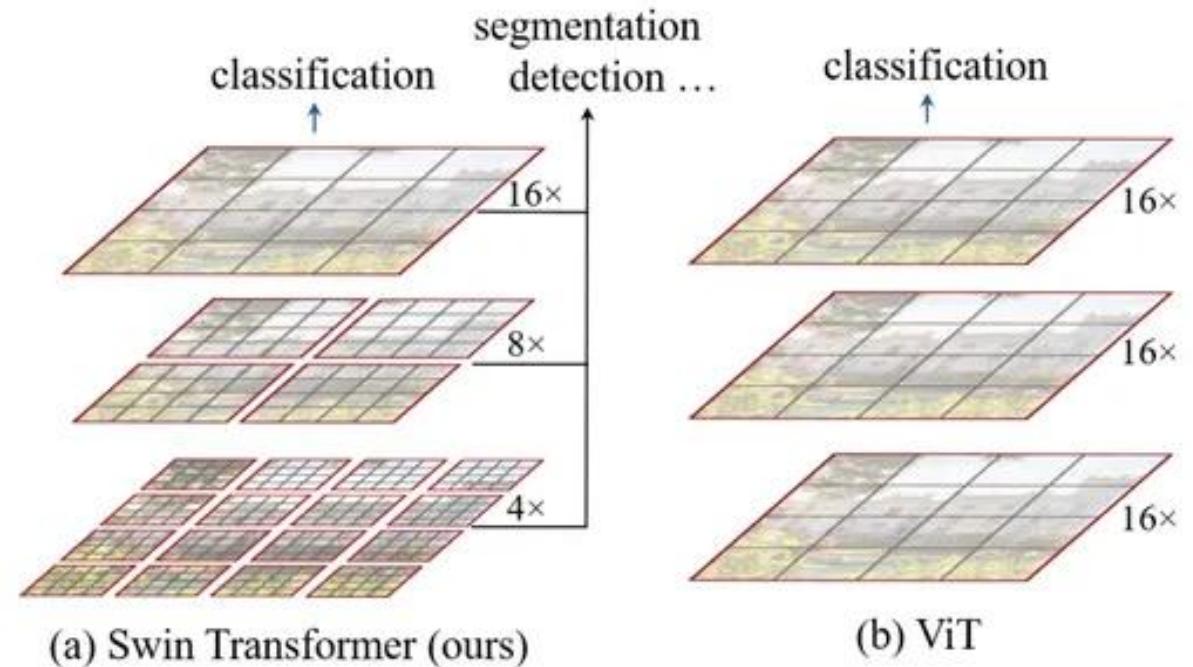
- ViTs need patches to be relevant information in order to train properly. This typically means that they often work better on high-resolution images.
- CIFAR-10 is likely too low in terms of resolution (32 by 32), in order for patches to be useful (8 by 8 patches will probably not contain much info).
- Try on a larger dataset to see if it works better?
(Be ready for the computational cost though!)
- Also, a good idea to bring all the training tools (LR schedulers, validation set, etc.) to improve the training of our model.

Advanced ViTs: The Swin Transformer (Implementation is out-of-scope)

Definition (**Swin Transformer**):

The **Swin Transformer** (short for "Shifted Window Transformer") is an advanced Vision Transformer, addressing scalability and efficiency challenges when applied to larger images and high-resolution tasks like object detection and segmentation.

It relies on **hierarchical representations, local self-attention and shifted windows**.

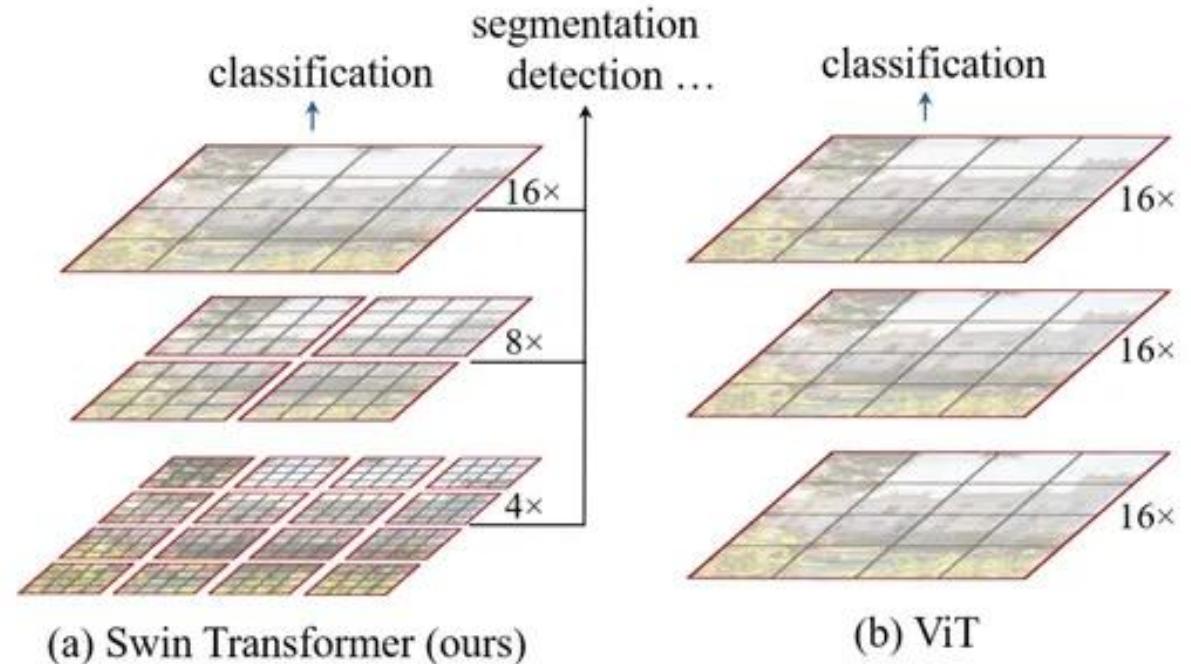


Intuition behind the Swin Transformer, taken from [SWIN2021]

Advanced ViTs: The Swin Transformer (Implementation is out-of-scope)

Hierarchical representations

- Standard ViT: Fixed-size grid of patches.
- Swin: Builds a hierarchical feature pyramid similar to CNNs.
- Starts with small patches and progressively merges them to form larger receptive fields.
- Enables multi-scale feature extraction, tasks involving spatial hierarchies (e.g., detecting objects of different sizes in a given image).



Intuition behind the Swin Transformer, taken from [SWIN2021]

Advanced ViTs: The Swin Transformer

(Implementation is out-of-scope)

Local Self-Attention Windows:

- Instead of computing self-attention globally for all patches (which has quadratic complexity in the number of patches), Swin processes patches **within local windows**.
- **Key Advantage:** Reduces the computational complexity from $O(N^2)$ to $O(N)$, where N is the number of patches.

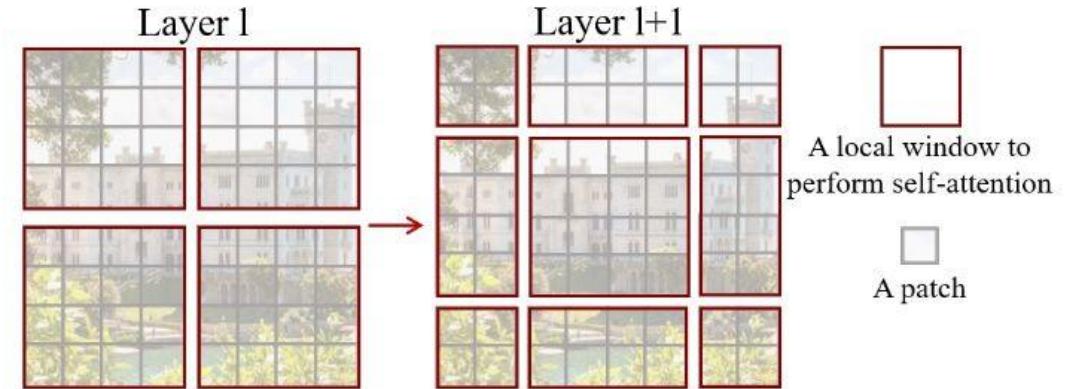


Figure 2. An illustration of the *shifted window* approach for computing self-attention in the proposed Swin Transformer architecture. In layer l (left), a regular window partitioning scheme is adopted, and self-attention is computed within each window. In the next layer $l + 1$ (right), the window partitioning is shifted, resulting in new windows. The self-attention computation in the new windows crosses the boundaries of the previous windows in layer l , providing connections among them.

知乎 @谷溢

Intuition behind the Swin Transformer, taken from [SWIN2021]

Advanced ViTs: The Swin Transformer

(Implementation is out-of-scope)

Shifted Window Mechanism:

- To overcome the limitation of local windows (which lack global context), Swin **shifts the windows** between successive layers.
- This allows **cross-window interactions** and captures long-range dependencies efficiently.
- In a sense, it allows to “propagate” information across windows over successive layers.

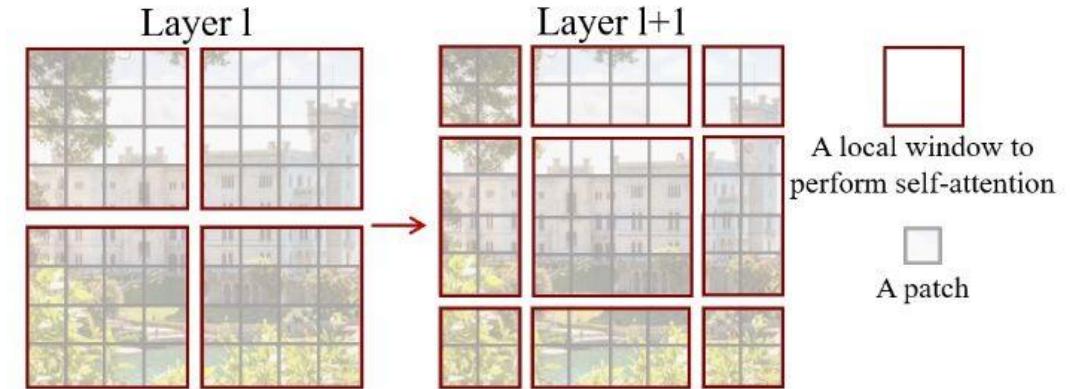


Figure 2. An illustration of the *shifted window* approach for computing self-attention in the proposed Swin Transformer architecture. In layer l (left), a regular window partitioning scheme is adopted, and self-attention is computed within each window. In the next layer $l + 1$ (right), the window partitioning is shifted, resulting in new windows. The self-attention computation in the new windows crosses the boundaries of the previous windows in layer l , providing connections among them.

知乎 @谷溢

Intuition behind the Swin Transformer, taken from [SWIN2021]

Advanced ViTs: Hybrid models (Implementation is out-of-scope)

Strengths of CNNs:

- CNNs excel at extracting local features (e.g., edges, textures) due to their inductive biases like locality and translation invariance.
- Efficient for tasks requiring fine-grained feature extraction and low computational cost.

Strengths of Vision Transformers:

- Transformers excel at capturing global context using self-attention mechanisms.
- Particularly useful for tasks requiring long-range dependencies and relationships.

Challenges When Used Independently:

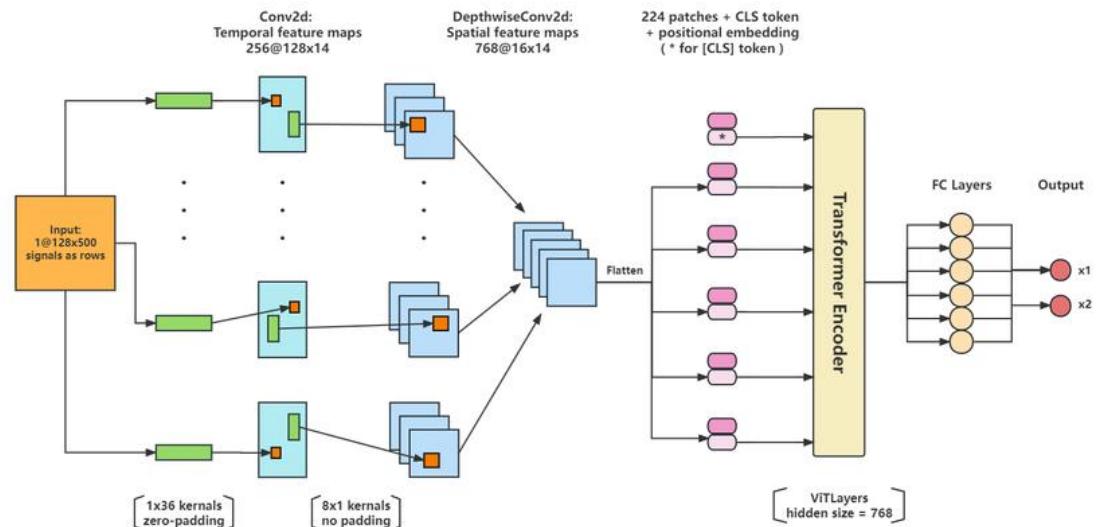
- CNNs struggle to model global dependencies due to their local receptive fields.
- Transformers are data-hungry and computationally expensive, making them challenging to train on small or mid-sized datasets.

Advanced ViTs: Hybrid models (Implementation is out-of-scope)

Definition (**Hybrid ViT models**):

Hybrid ViT models attempt to combine the best of both the CNN and ViT worlds.

- For instance, these hybrid models might use CNNs as a front-end feature extractor, efficiently capturing local spatial information.
- Then, the features extracted by CNNs might be passed to the Transformer layers for global self-attention and context modeling.



Intuition behind a possible Hybrid model, taken from [DEIT2021]

Implementing a self-attention layer (short)

In Notebook 3, we show an even simpler implementation of a hybrid ViT model and run it on MNIST. It starts with attention layers.

```
1 # Define a self-attention layer implementation
2 class SelfAttentionLayer(nn.Module):
3     def __init__(self, in_features):
4         super(SelfAttentionLayer, self).__init__()
5         self.in_features = in_features
6         self.query = nn.Linear(in_features, in_features)
7         self.key = nn.Linear(in_features, in_features)
8         self.value = nn.Linear(in_features, in_features)
9
10    def forward(self, x):
11        batch_size = x.size(0)
12        query = self.query(x).view(batch_size, -1, self.in_features)
13        key = self.key(x).view(batch_size, -1, self.in_features)
14        value = self.value(x).view(batch_size, -1, self.in_features)
15        attention_weights = F.softmax(torch.bmm(query, key.transpose(1, 2))/(self.in_features**0.5), dim = 2)
16        out = torch.bmm(attention_weights, value).view(batch_size, -1)
17        return out
```

A simple vision transformer on MNIST?

Then, define our MNIST Hybird Vision Transformer as:

- Linear(28×28 , 128) to be used on flattened images,
- Self-attention,
- Linear(128, 64),
- Self-attention again,
- Linear(64, 10), to get the right number of classes,
- ReLU activations after each Linear except the last one.

```
1 # Neural network definition using self-attention
2 class Transformer(nn.Module):
3     def __init__(self):
4         super(Transformer, self).__init__()
5         self.fc1 = nn.Linear(28*28, 128)
6         self.attention1 = SelfAttentionLayer(128)
7         self.fc2 = nn.Linear(128, 64)
8         self.attention2 = SelfAttentionLayer(64)
9         self.fc3 = nn.Linear(64, 10)
10
11    def forward(self, x):
12        x = x.view(-1, 28*28)
13        x = F.relu(self.fc1(x))
14        x = self.attention1(x)
15        x = F.relu(self.fc2(x))
16        x = self.attention2(x)
17        x = self.fc3(x)
18        return x
```

Will our simple attention model train?

```
1 # Create model
2 model = Transformer()
3 # Define the loss function and optimizer
4 criterion = nn.CrossEntropyLoss()
5 optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
6
7 # Train the model
8 n_epochs = 5
9 for epoch in range(n_epochs):
10     for i, (images, labels) in enumerate(train_loader):
11         # Flatten image
12         images = images.reshape(-1, 28*28)
13         # Forward pass
14         outputs = model(images)
15         loss = criterion(outputs, labels)
16         # Backprop
17         optimizer.zero_grad()
18         loss.backward()
19         optimizer.step()
20         # Display
21         if (i + 1) % 100 == 0:
22             print("Epoch [{}/{}], step [{}/{}], Loss: {:.4f}".format(epoch + 1, \
23                                         n_epochs, \
24                                         i + 1, \
25                                         len(train_loader), \
26                                         loss.item()))
```

Yes, and it seems to do ok!

```
Epoch [1/10], Step [100/1875], Loss: 0.6941
Epoch [1/10], Step [200/1875], Loss: 0.2044
Epoch [1/10], Step [300/1875], Loss: 0.7688
Epoch [1/10], Step [400/1875], Loss: 0.4087
Epoch [1/10], Step [500/1875], Loss: 0.3414
Epoch [1/10], Step [600/1875], Loss: 0.0827
Epoch [1/10], Step [700/1875], Loss: 0.1491
Epoch [1/10], Step [800/1875], Loss: 0.5172
Epoch [1/10], Step [900/1875], Loss: 0.1956
Epoch [1/10], Step [1000/1875], Loss: 0.2963
Epoch [1/10], Step [1100/1875], Loss: 0.1300
Epoch [1/10], Step [1200/1875], Loss: 0.4159
Epoch [1/10], Step [1300/1875], Loss: 0.0367
Epoch [1/10], Step [1400/1875], Loss: 0.3448
Epoch [1/10], Step [1500/1875], Loss: 0.2060
Epoch [1/10], Step [1600/1875], Loss: 0.0879
```

```
1 # Test the model
2 with torch.no_grad():
3     correct = 0
4     total = 0
5     for images, labels in test_loader:
6         # Flatten images
7         images = images.reshape(-1, 28 * 28)
8         # Forward pass and accuracy calculation
9         outputs = model(images)
10        _, predicted = torch.max(outputs.data, 1)
11        total += labels.size(0)
12        correct += (predicted == labels).sum().item()
13    # Final display
14    print("Test Accuracy: {} %".format(100*correct/total))
```

Test Accuracy: 97.04 %

Seems to train nicely!

Open question about another possible hybrid

```
# Define a convolutional attention layer implementation
class ConvAttentionLayer(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3):
        super(ConvAttentionLayer, self).__init__()
        self.query_conv = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, padding=kernel_size // 2)
        self.key_conv = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, padding=kernel_size // 2)
        self.value_conv = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, padding=kernel_size // 2)

    def forward(self, x):
        query = self.query_conv(x)
        key = self.key_conv(x)
        value = self.value_conv(x)
        batch_size, channels, height, width = query.size()
        query = query.view(batch_size, channels, -1)
        key = key.view(batch_size, channels, -1)
        value = value.view(batch_size, channels, -1)
        attention_weights = F.softmax(torch.bmm(query.transpose(1, 2), key), dim=2)
        out = torch.bmm(value, attention_weights).view(batch_size, channels, height, width)
        return out
```

Question: Can we get better performance by combining convolutions and attentions operations, to get the best of both worlds?

Possible applications for ViTs: Image captioning + Local Explanation



A woman is throwing a frisbee in a park.



A dog is standing on a hardwood floor.



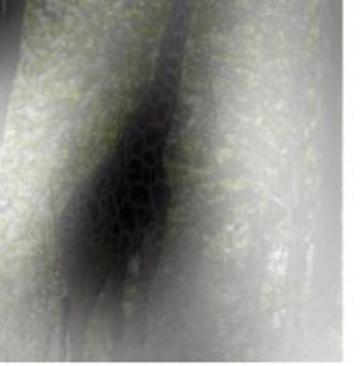
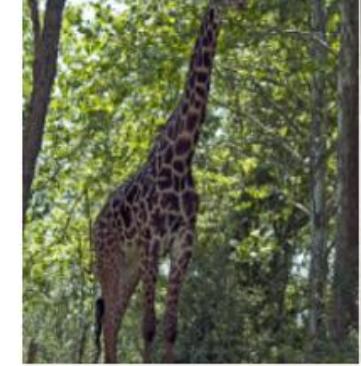
A stop sign is on a road with a mountain in the background.



A little girl sitting on a bed with a teddy bear.



A group of people sitting on a boat in the water.

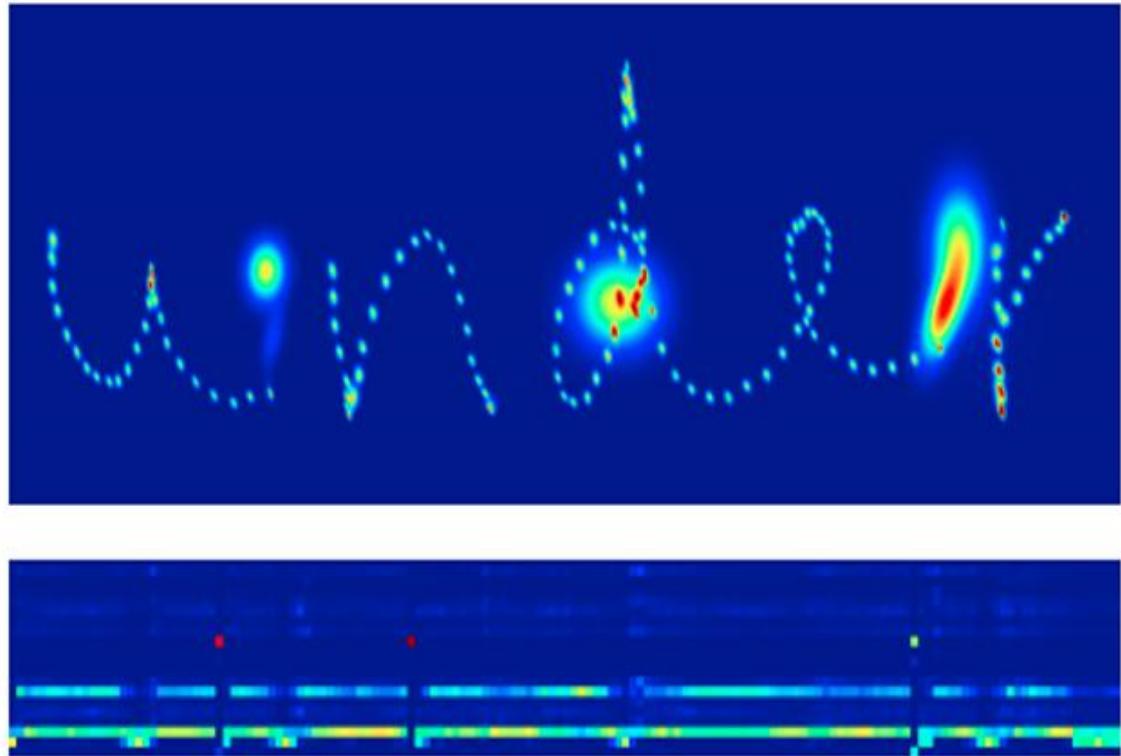


A giraffe standing in a forest with trees in the background.

Possible applications for ViTs: Video captioning + Frame Explanation



Possible applications for ViTs: Handwritten Recognition + Other Languages?



零 壹 贰 叁
肆 伍 六 柒
捌 玖 拾 佰
仟 万 亿 桑

Learn more about these topics

Out of class, for those of you who are curious

- [VIT2020] Dosovitskiy et al., “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”, 2020.
- [SWIN2021] Z. Liu, et al. “Swin transformer: Hierarchical vision transformer using shifted windows”, 2021.
- [DEIT2021] Touvron et al., “Training Data-Efficient Image Transformers & Distillation through Attention ”, 2021.
- [KN] A good article to explain the mathematical intuition behind the sinusoidal positional encodings
https://kazemnejad.com/blog/transformer_architecture_positional_encoding/