

04 Vision Transformer new approach to vision processing

1. From CNNs to ViTs

****CNNs**

Local pattern recognition

- Edges, textures, small objects
- Built-in translation equivariance

Parameter efficiency

- Weight sharing across spatial locations
- Small models can work well

Small dataset performance

- Strong inductive biases help with limited data
- Work well with < 100K images

Global context

- Need many layers to see full image
- Receptive field grows slowly: $O(\sqrt{L})$

Long-range dependencies

- Difficult to relate distant pixels
- Information must flow through many layers

Adaptive computation

- Same computation everywhere
- Can't focus on important regions

Scalability

- Performance saturates with more parameters
- Diminishing returns from depth

The Vision Transformer Motivation

Key Question:

If Transformers revolutionised NLP by treating text as sequences, can we treat images as sequences too?

The Parallel:

NLP:

"The cat sat on the mat" → [The] [cat] [sat] [on] [the] [mat]

↓ ↓ ↓ ↓ ↓ ↓

Self-Attention → Understand relationships

Computer Vision:

Image (224×224) → [Patch₁] [Patch₂] ... [Patch₁₉₆]

↓ ↓ ↓

Self-Attention → Understand spatial relationships

CNN Limitations → ViT Solutions

CNN Limitation	Root Cause	ViT Solution
Slow receptive field growth	Local convolutions	Global attention from layer 1
Fixed computation pattern	Sliding window	Dynamic, content-based routing
Poor long-range modeling	Information bottleneck	Direct all-to-all connections
Architecture complexity	Manual design (ResNet, etc.)	Minimal architecture
Texture bias	Local pattern focus	More shape-aware

Architectural Comparison

CNN Processing:

Input → Conv(3×3) → Conv(3×3) → Conv(3×3) → ... → Global Pool → FC

↓ ↓ ↓

Local Local Local

(receptive field grows slowly)

ViT Processing:

```
Input → Patches → Self-Attention → Self-Attention → ... → [CLS] Token → FC
                ↓                ↓
                Global            Global
                (immediate full-image context)
```

The Trade-off: Inductive Bias vs Data

CNNs: Strong Priors, Data Efficient

```
# CNN priors (built into architecture)
priors = {
    'locality': True,          # Nearby pixels related
    'translation_eq': True,   # Same pattern anywhere
    'hierarchy': True,        # Simple → Complex
    'weight_sharing': True,   # Same filter everywhere
}

result = {
    'data_needed': '10K - 100K images',
    'performance_ceiling': 'Good but limited',
    'flexibility': 'Low (priors may not fit data)'
}
```

ViT: Weak Priors, Data Hungry

```
# ViT priors (minimal architecture constraints)
priors = {
    'locality': False,        # Learn if needed
    'translation_eq': False,  # Learn if needed
    'hierarchy': False,       # Flat processing
    'weight_sharing': 'Only in attention',
}

result = {
    'data_needed': '1M - 300M images',
    'performance_ceiling': 'State-of-the-art',
    'flexibility': 'High (can learn any pattern)'
}
```

CNNs Win: When Priors Match Reality

✅ **Small datasets** (< 100K images)

- Strong priors compensate for limited data
- Faster convergence

✓ Truly local tasks

- Texture classification
- Edge detection
- Small object recognition

✓ Deployment constraints

- Mobile devices (100x fewer FLOPs)
 - Real-time requirements (lower latency)
-

ViT Wins: When Flexibility Needed

✓ Large datasets (> 1M images)

- Can learn optimal features
- Weak priors don't constrain learning

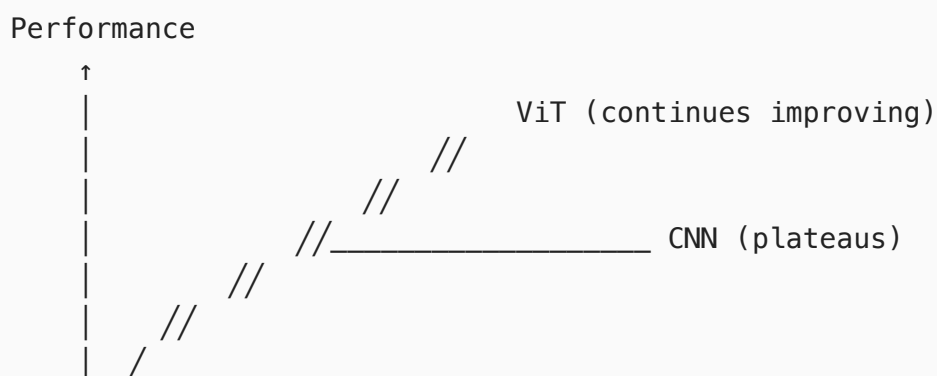
✓ Global understanding

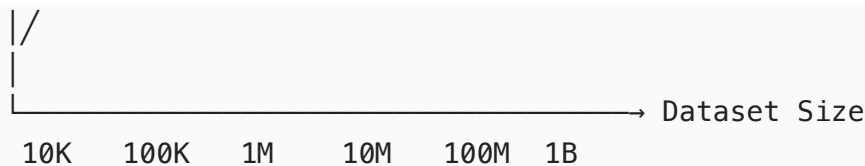
- Scene recognition
- Object relationships
- Spatial reasoning

✓ Out-of-distribution data

- More robust to domain shift
 - Better transfer learning
-

The Data-Performance Curve





Crossover point: ~1M images

Before: CNNs better

After: ViT better

Practical Decision Framework

Use CNNs When:

```
if (dataset_size < 100_000 or
    deployment_target == 'mobile' or
    inference_latency_ms < 10 or
    task_requires_fine_details):

    model = 'ResNet-50' # or EfficientNet, ConvNeXt
```

Use ViT When:

```
if (dataset_size > 1_000_000 or
    transfer_learning_available or
    global_context_crucial or
    robustness_priority):

    model = 'ViT-B/16' # with pretrained weights
```

Use Hybrid When:

```
if (want_best_of_both):
    model = 'Swin-Transformer' # or ConvNeXt, CvT
    # CNN-like locality + Transformer flexibility
```

ViT

Architecture Details

1. How to tokenise images (patching)
2. Position encodings (absolute vs relative)

3. Attention mechanism in vision context
4. Multi-head attention for visual features

Training Considerations

1. Why ViT is hard to train from scratch
2. Critical hyperparameters (warmup, weight decay)
3. Data augmentation strategies
4. Transfer learning best practices

Practical Insights

1. When ViT actually helps vs hype
2. Computational costs and optimizations
3. How to debug attention patterns
4. Deployment considerations

The Bigger Picture: Convergence of Architectures

Evolution Timeline

2012: AlexNet → CNNs dominate vision

2017: Transformer → Attention dominate NLP

2020: ViT → Can we use Transformers for vision?

2021–2023: Convergence

- └ CNNs adopt attention (CBAM, CoordConv)
- └ ViT adopt locality (Swin, LocalViT)
- └ Hybrid models (CvT, ConvNeXt)

2024+: Architecture agnostic

- Meta-learning optimal structures
- Task-specific architectures
- Unified multi-modal models

Key Mindset Shift from CNN to Transformer

"How can I design layers to extract hierarchical features?"

↓

Engineer locality → Build hierarchy → Hope for global understanding

"What relationships matter in this image?"

↓

Let model discover structure → Learn which patterns to attend to

The Philosophical Difference:

CNN Philosophy	ViT Philosophy
"Impose structure"	"Discover structure"
"Build hierarchy"	"Learn connections"
"Local first, then global"	"Global always, refine locally"
"Architecture = prior knowledge"	"Architecture = flexible framework"

What Makes ViT Different (Preview)

1. Patch-based Processing

- Image → Non-overlapping patches
- Each patch = token (like word in NLP)
- Position encoding adds spatial info

2. Self-Attention

- Every patch attends to every other patch
- Content-based, not position-based
- Dynamic weights, not fixed kernels

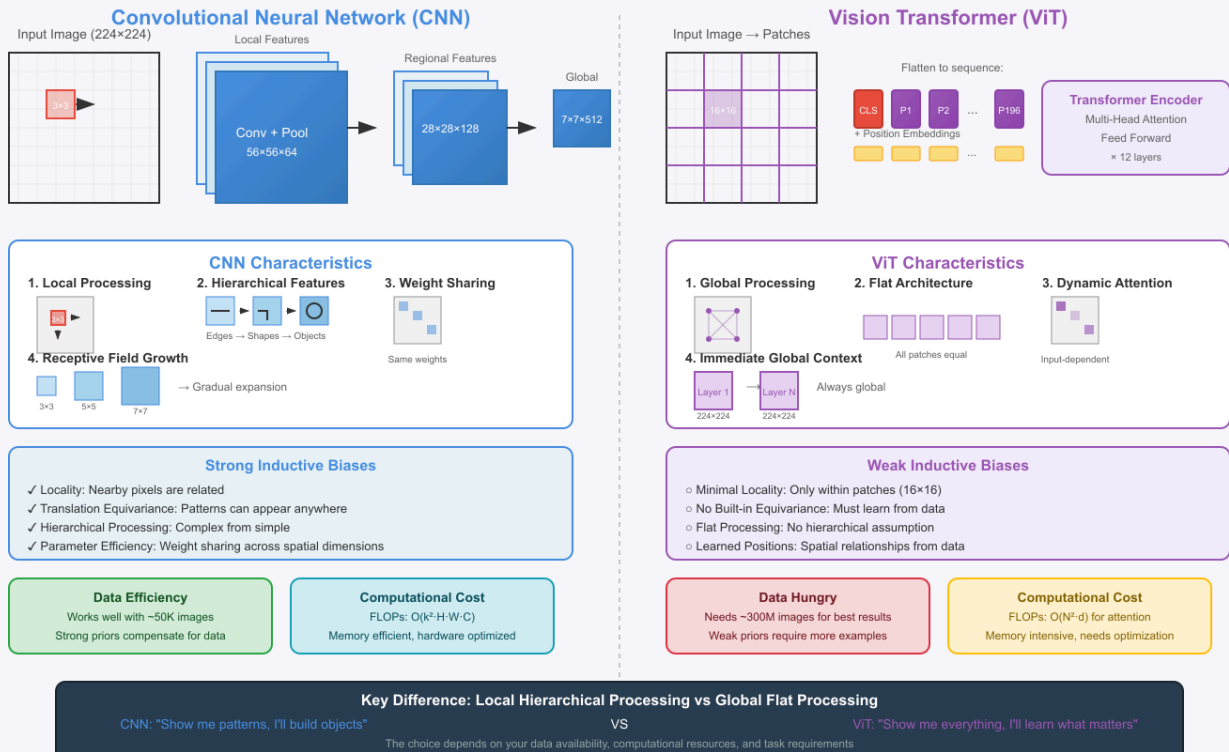
3. Global Receptive Field

- Layer 1 sees entire image (!)
- No need for depth to build context
- Different from CNN's gradual expansion

4. Data Requirements

- Needs massive pre-training (300M images)
- Or strong data augmentation
- Transfer learning is crucial

CNN vs Vision Transformer: Processing Paradigms



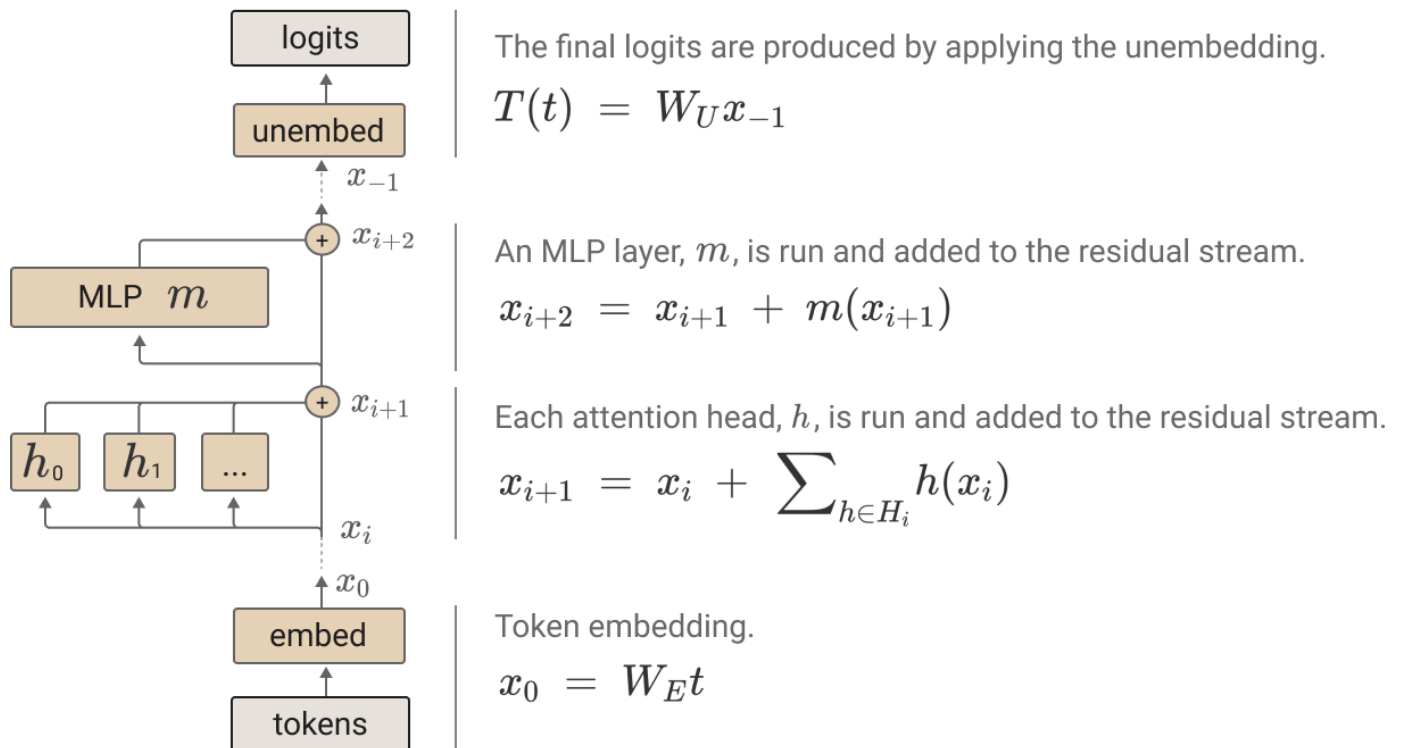
2. The Transformer Revolution

As aliens entered our planet

Original idea of attention comes from [Neural machine translation by jointly learning to align and translate, Bahdanau et al., 2014 \(ICLR 2015\)](#).

Image ($H \times W \times C$) \rightarrow Patches ($N \times P^2 \times C$) \rightarrow Tokens \rightarrow Transformer \rightarrow Classification

- this results in
 - no convolutions
 - no image-specific inductive biases
 - just attention (and ffn), all the way down



(from an Anthropic blog page) a Transformer block

Transformer is *just* a linear combination of nonlinear functions

- an attention,
- and a feed-forward network
- added together to a residual stream (remember the ResNet architecture?)

Key Question: Can we treat images as sequences and leverage Transformers' success?

3. Tokenisation: From Pixels to Patches

The Fundamental Problem

1. pixels as tokens

```
# Direct approach (impractical)
Image: 224×224×3 = 150,528 pixels
```

Attention complexity: $O(N^2) = O(150,528^2) \approx 22.6$ billion operations!

- computationally prohibitive
- Memory requirements impossible
- Most pixel-to-pixel attention unnecessary

ViT solution: Use patches

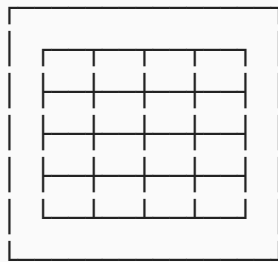
Patches: $(224/16) \times (224/16) = 14 \times 14 = 196$ patches

Attention complexity: $O(196^2) = 38,416$ operations (manageable!)

- manageable

Patch Embedding Process

Step 1: Divide image into non-overlapping patches



224×224 image
→ 14×14 grid
→ 196 patches of 16×16

Step 2: Flatten each patch

Each 16×16×3 patch → 768-dimensional vector

Step 3: Linear projection

768 → D (embedding dimension, typically 768 or 1024)

Mathematical Formulation

Given image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$:

1. **Reshape into patches** where

$$N = HW/P^2$$

2. Flatten each patch

$$\text{flatten} : \mathbb{R}^{P \times P \times C} \rightarrow \mathbb{R}^{P^2 C}$$

3. Linear projection to embedding

$$\mathbf{z}_0 = \mathbf{x}_p \mathbf{E} + \mathbf{x}_{pos}$$

- where $\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}$ is the patch embedding matrix
- and $\mathbf{x}_{pos} \in \mathbb{R}^{N \times D}$ is the positional embedding

[CLS] token (classification token): global representation

```
# Special learnable token for classification
self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))

# Prepend to patch tokens
cls_tokens = self.cls_token.expand(B, -1, -1) # (B, 1, embed_dim)
x = torch.cat((cls_tokens, x), dim=1) # (B, num_patches + 1, embed_dim)
```

Why [CLS] token?

- [CLS] is prepended to the input sequence and serves as a global representation
- Aggregates global information through attention
- No positional bias (unlike using first/last patch)
- Used for final classification
- is learnable

4. Embeddings & Positional Encoding

What is an Embedding?

An embedding maps discrete objects to continuous vector representations:

Discrete Space		Continuous Space
Patch (16×16×3)	→	Vector $\in \mathbb{R}^D$
Raw pixels		Semantic representation
High-dimensional		Lower-dimensional
Sparse		Dense
Patch (16×16×3 raw pixels)	→	Embedding (D-dimensional vector)
Learned representation captures semantic information		

Are the embeddings learnable?

- embeddings are just linear transformations

```
nn.Embedding(max_length, d_length)
```

- embeddings are trained along with all other parameters
- why?
 - same inputs need different embeddings in different contexts, i.e. are task-specific
 - embeddings work in cooperation with attention
 - learned embeddings can help (optimise) distribution of information
 - can learn token relations
 - allow for flexibility
 - fixed embeddings can be used for very small datasets and for pre-trained embeddings as initialisations
- embeddings
 - provide model's vocabulary
 - optimise representations in connection with attention
 - develop task-specific coding, adapt to these tasks and datasets
 - can help optimise gradient flow
 - adapt to specific
 - learns optimal representations
 - capture patterns emerging from the data
 - enable true end-to-end learning

Why Positional Encoding is Critical

$$\text{Attention}(\pi(\mathbf{X})) = \text{Attention}(\mathbf{X})$$
for permutation π

Transformers are permutation invariant!

Without positional information

[Patch_1] [Patch_2] [Patch_3] \equiv [Patch_3] [Patch_1] [Patch_2]

This would destroy spatial structure!

Positional Encoding Methods

1. Learnable Position Embeddings (ViT default)

```
```python
```

```
self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1, embed_dim))
nn.init.trunc_normal_(self.pos_embed, std=0.02)
```

```
Add to patch embeddings
x = x + self.pos_embed
```

- Should it be **added** or **concatenated**?
  - concatenation extends the dimension but is reversible
  - addition is not reversible
  - but addition carries the position information
    - the same information token in **different** places would convey that info
  - so add or concatenate

### Positional Encoding in Transformers: Addition in Embedding Space

Example: "A husband of a queen is a king"

#### Step 1: Tokens



#### Step 2: Token Embeddings (4 dimensions shown)

Token	dim 0	dim 1	dim 2	dim 3	... (d=512)
"queen"	0.42	-0.31	0.85	-0.12	...

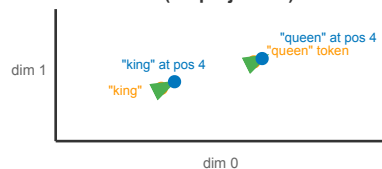
#### Step 3: Positional Encoding (sin/cos patterns)

Position	dim 0	dim 1	dim 2	dim 3	
pos = 4	0.05	0.02	-0.03	0.08	← Small values!

#### Step 4: Element-wise Addition

$$[0.42, -0.31, 0.85, -0.12, \dots] + [0.05, 0.02, -0.03, 0.08, \dots] = [0.47, -0.29, 0.82, -0.04, \dots]$$

#### Step 5: Geometric View (2D projection)



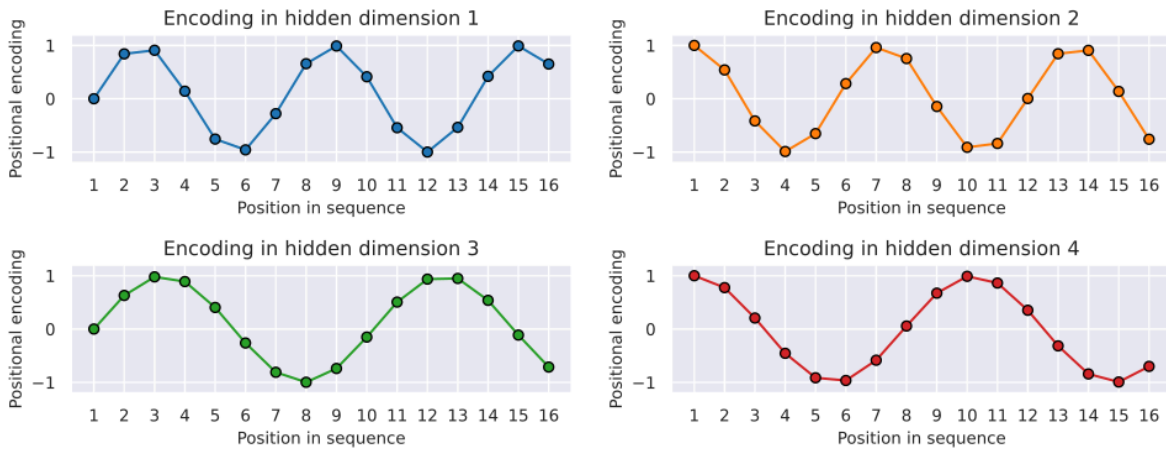
Final embedding = TokenEmbed(word) + PosEncode(position)  
 where ||PosEncode|| << ||TokenEmbed|| (typically 10-20% magnitude)

#### Why This is Optimal:

1. Small translations: Position values are ~10x smaller than token embeddings
2. Unique per dimension: Each dim has different frequency (sin/cos patterns)
3. Preserves semantic info: "queen" and "king" stay semantically close
4. Position distinguishable: Same word at different positions are separate
5. Smooth interpolation: Nearby positions have similar encodings

- **Advantages:**
  - Flexible, can learn any pattern
  - Simple to implement
  - Empirically effective
- **Disadvantages:**
  - Fixed resolution at training
  - Doesn't naturally generalise to different image sizes

## 2. Sinusoidal Encoding (from original Transformer)



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

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

### 3. 2D Positional Encoding

Encode row and column positions separately:

- encode separately for rows and columns
- add using concatenation

```
pos_embed = pos_embed_row + pos_embed_col
```

### 4. Relative Position Encoding

Instead of absolute positions, encode relative distances:

$$\text{Attention}_{ij} = \text{softmax}\left(\frac{q_i \cdot k_j + r_{ij}}{\sqrt{d}}\right)$$

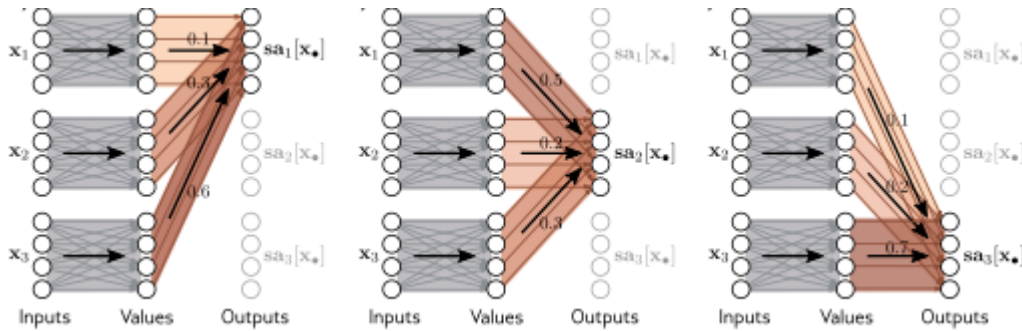
where  $r_{ij}$  encodes the relative position between patches  $i$  and  $j$

### 5. Interpolation for different resolutions

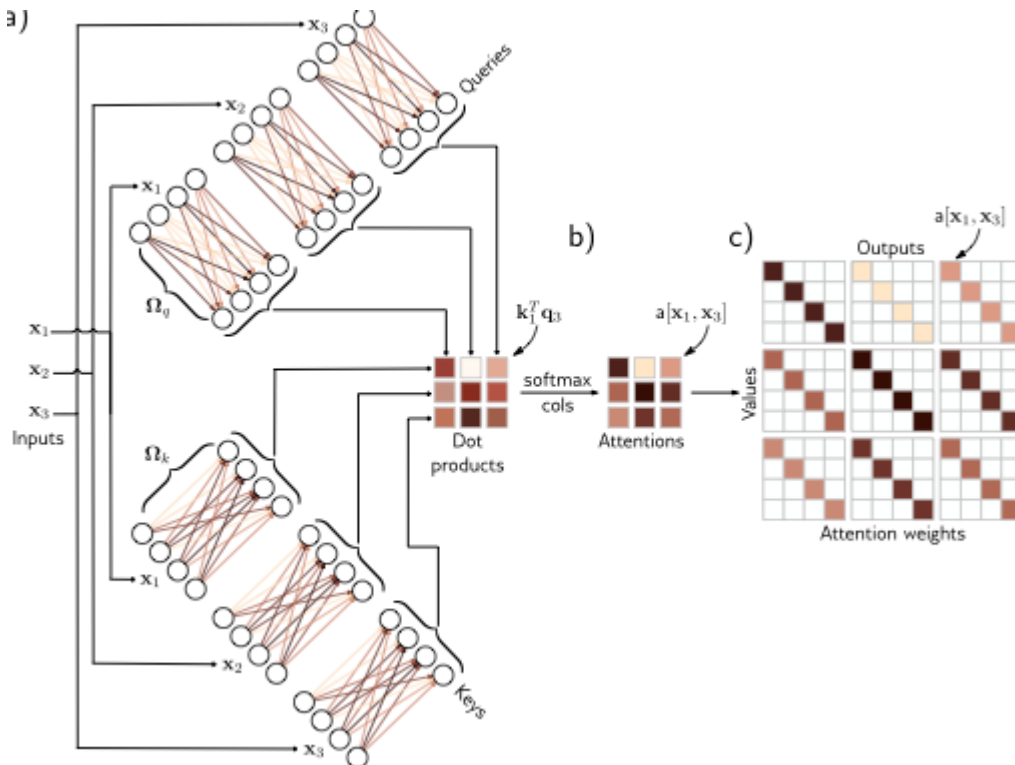
- **different** resolutions can be used to use coarse - to fine attention
- the positional tokens may be interpolated, e.g. bilinearly

## 5. Attention Mechanism in Vision

### Self-Attention: The Core Operation



- a **value** is computed from each token
  - may be done in parallel
  - value  $v_i$  is some linear combination of all elements of vector representation of token  $x_i$
- a scalar  $a[x_i, x_j]$  is the *attention* that token  $x_j$  pays to token  $x_i$ 
  - attentions  $a[\cdot, x_j]$  sum up to 1
- *self attention*  $sa_j(x_1, \dots, x_N) = \sum_{i=1}^N a[x_i, x_j] \cdot v_i$ 
  - each self-attention  $sa_i[x_\cdot]$  can be thought as a different routing of the original  $N$  tokens  $x_i$  for the current task
- all can be computed in parallel per token



- *query* vectors are computed as  $q_n = \beta_q + \Omega_q x_n$
- *key* vectors are computed as  $k_n = \beta_k + \Omega_k x_n$



- dot products are passed to a softmax giving attention values

Given input  $\mathbf{X} \in \mathbb{R}^{N \times D}$ :

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

Where:

- $Q = XW_Q$  (Queries): "What am I looking for?"
- $K = XW_K$  (Keys): "What information do I have?"
- $V = XW_V$  (Values): "What information to aggregate?"

## Visual Interpretation

For each patch:

1. Query: "I'm a patch containing an eye"
2. Keys: All patches respond with what they contain
3. Attention weights: High for other eye, nose, face patches
4. Values: Aggregate features from attended patches
5. Output: Enriched representation with context

## Multi-Head Attention

### Key Idea

Different heads learn different types of relationships

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O$$

where each head:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

```
Instead of single attention:
MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W_O

Where each head attends to different aspects:
head_i = Attention(QW_Q^i, KW_K^i, VW_V^i)
```

### Intuition

Different heads should capture different relationships

```
Empirical observations from trained ViTs:
```

Head 1-3: Local patterns (similar to small convolutions)

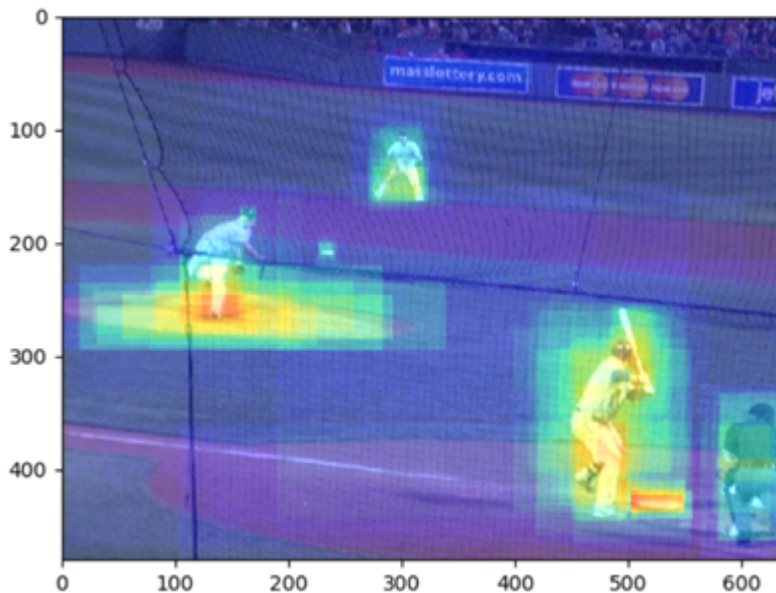
Head 4-6: Texture and colour similarity

Head 7-9: Semantic grouping (objects)

Head 10-12: Global context and spatial relationships

## Visualising Attention

- to show "important" places in a grid



```
def visualize_attention(model, image, patch_size=16, layer_idx=11):
 """Visualize attention from CLS token to patches"""
 # Get attention weights
 _, attention_weights = model.get_attention(image, layer_idx)

 # attention_weights: (batch, num_heads, num_patches+1, num_patches+1)
 # We want CLS token attention (index 0) to all patches
 cls_attention = attention_weights[0, :, 0, 1:] # (num_heads,
num_patches)

 # Average across heads
 cls_attention = cls_attention.mean(dim=0)

 # Reshape to image grid
 h = w = int(math.sqrt(cls_attention.shape[0]))
 attention_map = cls_attention.reshape(h, w)

 # Interpolate to original image size
 attention_map = F.interpolate(
 attention_map.unsqueeze(0).unsqueeze(0),
 size=image.shape[-2:],
```

```
mode='bilinear'
) .squeeze()

return attention_map
```

## Computational Complexity of Attention

### Standard Self-Attention:

- Complexity:  $O(N^2 \cdot d)$  where  $N$  is sequence length
- Memory:  $O(N^2)$  for attention matrix

### For Vision (196 patches):

```
ViT-B/16 on 224x224 image
num_patches = 196
embed_dim = 768
num_heads = 12

Per layer:
attention_flops = 2 * num_patches**2 * embed_dim # ~59M
projection_flops = 4 * num_patches * embed_dim**2 # ~462M
total_flops_per_layer = ~521M

For 12 layers:
total_flops = ~6.3B
```

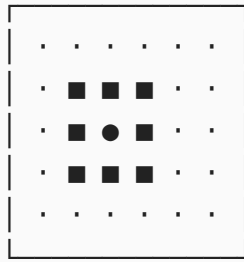
---

## 6. Attention vs Convolution: Fundamental Differences

- **convolutions**
  - **Locality** Only sees  $k \times k$  neighborhood
  - **Weight sharing** Same kernel everywhere
  - **Translation equivariance** Built-in
  - **No content adaptation** Same operation regardless of input
- **attention**
  - **Global reach** Every patch sees every other patch
  - **Dynamic weights** Attention weights depend on input
  - **No built-in equivariance** Must be learned or added
  - **Content adaptation** Different weights for different inputs

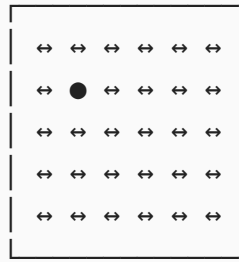
## Receptive Field Comparison

Convolution (3×3 kernel):



Local  
Fixed size  
Same for all

Self-Attention:



Global  
Dynamic  
Content-dependent

## Key Differences

Aspect	Convolution	Self-Attention
Receptive Field	Local (kernel size)	Global (all patches)
Weight Sharing	Same kernel everywhere	Dynamic, input-dependent
Computational Pattern	Sliding window	All-to-all comparison
Inductive Bias	Locality, translation equivariance	Minimal (only architecture)
Parameters	$O(k^2CD)$	$O(D^2)$
Compute	$O(k^2HWCD)$	$O(HW)^2D$

## Mathematical Comparison

Convolution:

$$y_{ij} = \sum_{m,n} w_{mn} \cdot x_{i+m,j+n}$$

- Static weights  $w_{mn}$
- Local neighborhood

Self-Attention:

$$y_i = \sum_{j=1}^N \alpha_{ij} \cdot v_j$$

- Dynamic weights  $\alpha_{ij} = \text{softmax}(q_i \cdot k_j / \sqrt{d})$
- Global context

## Receptive Field Evolution

# CNN: Receptive field grows slowly

```

Layer 1: 3×3
Layer 2: 5×5
Layer 3: 7×7
...
Layer L: (2L+1)×(2L+1) # Linear growth

ViT: Immediate global receptive field
Layer 1: 224×224 # Full image
Layer 2: 224×224 # Full image
...
Layer L: 224×224 # Always global

```

## Inductive Bias Comparison

Property	Convolution	Self-Attention
Locality	Strong (kernel size)	None (learns if needed)
Translation Equivariance	Built-in	Must learn
Weight Sharing	Enforced	Across positions only
Scale Invariance	Partial (via architecture)	None
Permutation Invariance	No	Yes (without pos. encoding)
Long-range Dependencies	Difficult (need depth)	Natural (one layer)

## When Each Excels

### Convolution Wins:

```

Small datasets – strong priors help
if dataset_size < 100_000:
 use_cnn()

Mobile/edge deployment
if deployment == "mobile":
 use_mobilenet() # 100x fewer FLOPs

Known local patterns
if task == "edge_detection":
 use_cnn()

```

### Attention Wins:

```

Large datasets – can learn priors
if dataset_size > 10_000_000:
 use_vit()

```

```
Global relationships crucial
if task in ["scene_understanding", "object_relationships"]:
 use_transformer()

Variable input sizes (with adaptation)
if input_size == "variable":
 use_transformer_with_interpolation()
```

## 7. Inductive Biases: What Priors Do ViTs Encode?

### What is an Inductive Bias?

**Inductive bias:** Assumptions built into the model architecture that guide learning

Strong Bias

↓

Less flexible

Less data needed

Faster convergence

May miss patterns

Weak Bias

↓

More flexible

More data needed

Slower convergence

Can learn any pattern

### CNN Priors (Strong)

1. Locality: Nearby pixels are related
2. Translation Equivariance:  $f(\text{shift}(x)) = \text{shift}(f(x))$
3. Hierarchical Processing: Simple → Complex features
4. Weight Sharing: Same pattern detector everywhere

#### 1. Locality Bias

$$y_{i,j} = \sum_{m=-k}^k \sum_{n=-k}^k w_{m,n} \cdot x_{i+m,j+n}$$

implies the need of many layers for global context

#### 2. Translation Equivariance

```
If pattern appears at position (x, y) or (x+dx, y+dy)
CNN responds similarly (shifted)
```

```
Conv(Shift(image)) \approx Shift(Conv(image))
```

**But NOT rotation/scale equivariant:**

```
Conv(Rotate(image)) \neq Rotate(Conv(image))
Conv(Scale(image)) \neq Scale(Conv(image))
```

### 3. Spatial Hierarchy

```
CNNs build features hierarchically
Layer 1: Edges (3×3 receptive field)
Layer 2: Textures (5×5 receptive field)
Layer 3: Parts (11×11 receptive field)
...
Layer N: Objects (full image)
```

### 4. Weight Sharing

```
Same filter applied everywhere
parameters_conv = kernel_size2 × in_channels × out_channels
NOT: image_size2 × in_channels × out_channels
```

## ViT Priors (Weak)

1. Patch Structure: Minimal spatial assumption
2. Position Encoding: Weak spatial prior
3. Architecture: Transformer structure only

### 1. Patch Structure (Minimal Locality)

```
Only assumes pixels within patch might be related
patch_size = 16 # Weak locality assumption
```

### 2. Position Encoding (Learned Spatial Prior)

```
Positions are learned, not hard-coded
self.pos_embed = nn.Parameter(torch.randn(1, num_patches, embed_dim))
```

### 3. Attention Structure

```
Only architectural constraint: attention mechanism
No assumption about what to attend to
```

```
attention = softmax(Q @ K.T / sqrt(d)) @ V
```

## Measuring Inductive Bias Strength

### Data Efficiency Test

**ViTs** need 10x more data for ImageNet processing to achieve 75% accuracy

### Generalisation to Different Domains

```
Train on natural images, test on:
 CNN ViT
Medical images: 65% 71% # ViT better (less bias)
Sketches: 45% 42% # CNN better (edge bias helps)
Satellite: 55% 68% # ViT better (different structure)
```

## The Data Efficiency Trade-off

```
CNNs: Strong priors → Less data needed
CNN_ImageNet_Top1 = 76.2% # with 1.3M images

ViT: Weak priors → More data needed
ViT_ImageNet_Top1 = 77.9% # Needs pre-training on 300M images!
ViT_ImageNet_Top1 = 74.5% # Without pre-training (worse than CNN)
```

## The Bias-Variance Tradeoff in Deep Learning

### Classical View:

- High bias → Underfitting
- High variance → Overfitting

### Modern Deep Learning View:

#### CNNs: High bias, lower variance

- Strong assumptions may prevent learning optimal solution
- But more stable with limited data

#### ViTs: Low bias, higher variance

- Can learn optimal solution given enough data
- But may overfit or fail to converge with limited data

## When Each Approach Wins



## CNNs Excel When:

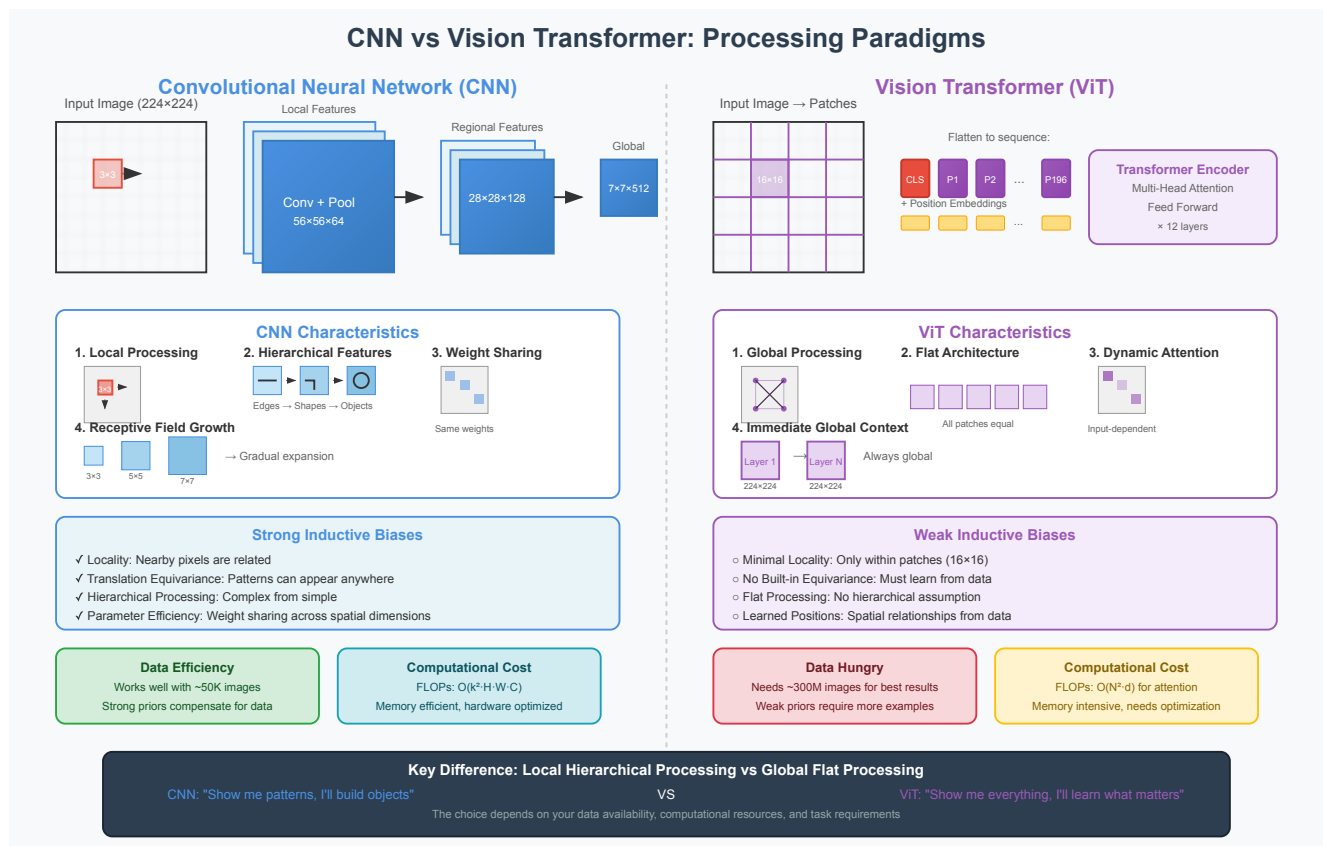
- Limited training data
- Strong locality matters
- Need parameter efficiency
- Edge devices/mobile deployment

## ViTs Excel When:

- Massive datasets available
- Global context crucial
- Long-range dependencies
- Transfer learning scenarios

# 8. Comparative Analysis: ViT vs CNN

## Performance Comparison



## ImageNet Classification Results

```
Model comparisons (ImageNet validation accuracy)
results = {
 # CNNs
 "ResNet-50": {"params": "25.6M", "FLOPs": "4.1G", "Top-1": 76.2},
```

```

"ResNet-152": {"params": "60.2M", "FLOPs": "11.3G", "Top-1": 77.8},
"EfficientNet-B0": {"params": "5.3M", "FLOPs": "0.39G", "Top-1":
77.1},
"EfficientNet-B7": {"params": "66M", "FLOPs": "37G", "Top-1": 84.3},
"ConvNeXt-B": {"params": "89M", "FLOPs": "15.4G", "Top-1": 85.8},

ViTs (without pre-training)
"ViT-B/16 (scratch)": {"params": "86.6M", "FLOPs": "17.6G", "Top-1":
74.5},
"ViT-L/16 (scratch)": {"params": "307M", "FLOPs": "63.6G", "Top-1":
76.5},

ViTs (with JFT-300M pre-training)
"ViT-B/16": {"params": "86.6M", "FLOPs": "17.6G", "Top-1": 84.0},
"ViT-L/16": {"params": "307M", "FLOPs": "63.6G", "Top-1": 87.5},
"ViT-H/14": {"params": "632M", "FLOPs": "167G", "Top-1": 88.6},
}

```

## Scaling Behavior

### Data Scaling

```

def plot_data_scaling():
 """How performance improves with more data"""

 data_points = [1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9]

 # CNN saturates quickly
 cnn_perf = [45, 65, 75, 80, 82, 83, 83.5]

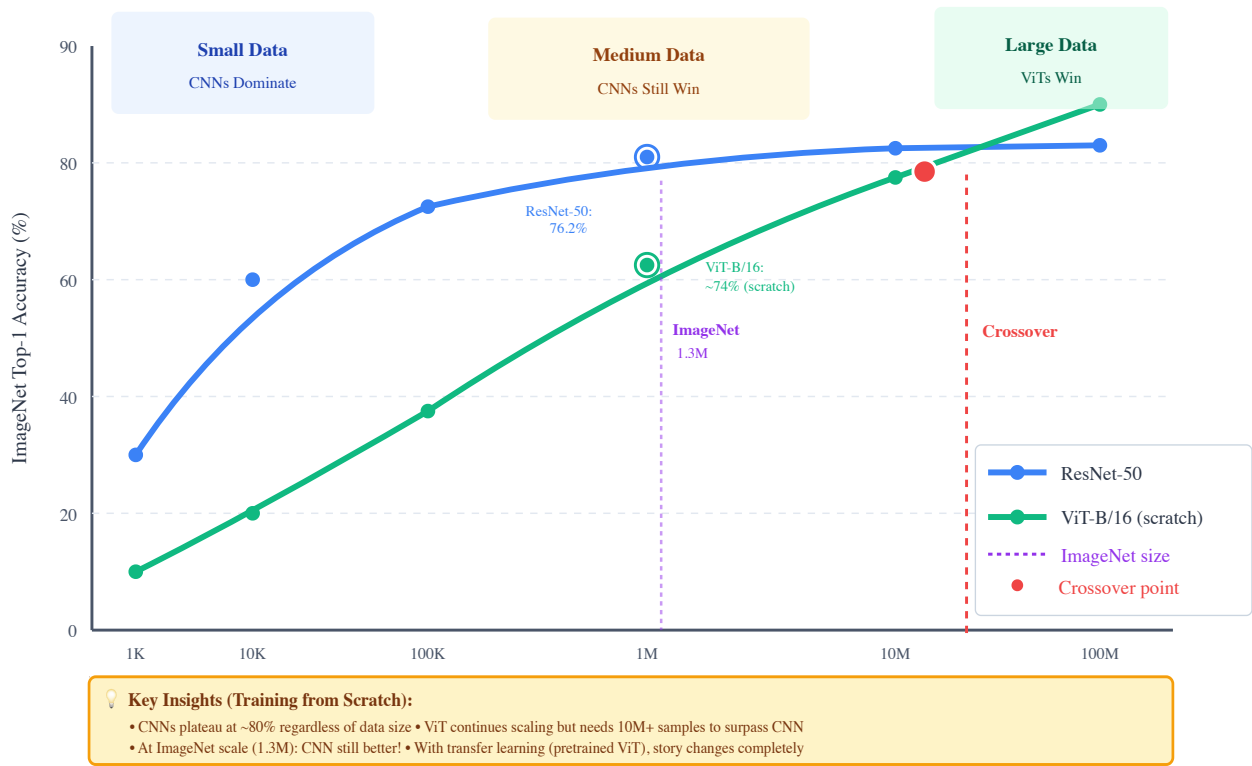
 # ViT keeps improving
 vit_perf = [20, 35, 55, 75, 83, 87, 90]

 # ViT surpasses CNN at ~1M samples
 crossover_point = 1e7

```

# Data Efficiency: CNN vs Vision Transformer

How performance scales with dataset size (training from scratch)



## Model Scaling

### # Parameters vs Performance

	Params	ImageNet	Scaling Efficiency
ViT-S/16:	22M	81.4%	<div><div></div></div>
ViT-B/16:	86M	84.0%	<div><div></div></div>
ViT-L/16:	307M	87.5%	<div><div></div></div>
ViT-H/14:	632M	88.6%	<div><div></div></div>
ResNet-50:	25M	76.2%	<div><div></div></div>
ResNet-152:	60M	77.8%	<div><div></div></div>
ResNet-200:	65M	78.3%	<div><div></div></div> (Saturating!)

## Computational Complexity Analysis

### Theoretical Complexity

Component	CNN	ViT
Convolution/Attention	$O(k^2 \cdot HW \cdot C_{in} \cdot C_{out})$	$O(N^2 \cdot d)$
Projection	-	$O(N \cdot d^2)$
Memory	$O(k^2 \cdot C_{in} \cdot C_{out})$	$O(N^2 + Nd)$
Parameters	$\sum_l k^2 \cdot C_{in}^l \cdot C_{out}^l$	$12 \cdot (3d^2 + d \cdot d_{mlp})$

## Practical Measurements

```
import time
import torch

def benchmark_models(batch_size=32, image_size=224):
 """Compare actual inference time"""

 x = torch.randn(batch_size, 3, image_size, image_size).cuda()

 models = {
 'ResNet-50': torchvision.models.resnet50(),
 'ViT-B/16': timm.create_model('vit_base_patch16_224'),
 'EfficientNet-B0': timm.create_model('efficientnet_b0'),
 }

 results = {}
 for name, model in models.items():
 model = model.cuda().eval()

 # Warmup
 for _ in range(10):
 _ = model(x)

 # Benchmark
 torch.cuda.synchronize()
 start = time.time()
 for _ in range(100):
 with torch.no_grad():
 _ = model(x)
 torch.cuda.synchronize()
 end = time.time()

 results[name] = (end - start) / 100

 return results

Typical results (V100 GPU):
ResNet-50: 8.2ms
EfficientNet-B0: 5.1ms
ViT-B/16: 15.3ms
```

## Training Dynamics

### Convergence Behavior

```
Training characteristics
```

```
CNNs: Fast initial convergence
epoch 1 5 10 20 50 100
CNN_acc: 45% 68% 74% 78% 80% 81%

ViTs: Slower start, better final performance
ViT_acc: 20% 45% 62% 75% 83% 86%

Key differences:
- ViTs need careful warmup (learning rate schedule)
- ViTs benefit more from longer training
- ViTs more sensitive to optimization hyperparameters
```

## Optimization Challenges

```
ViT-specific training techniques

class ViTTrainer:
 def __init__(self):
 # 1. Learning rate warmup (critical!)
 self.warmup_steps = 10000

 # 2. Large batch size (helps stability)
 self.batch_size = 4096

 # 3. Strong augmentation
 self.augmentation = RandAugment(n=2, m=10)

 # 4. Stochastic depth (drop paths)
 self.drop_path_rate = 0.1

 # 5. Weight decay (but not on all params)
 self.weight_decay = 0.05 # Higher than CNN typical 1e-4

 # 6. AdamW optimizer (not SGD)
 self.optimizer = torch.optim.AdamW

 # 7. Gradient clipping
 self.grad_clip = 1.0
```

## Robustness and Generalization

### Out-of-Distribution Performance

```
Robustness benchmarks
```

	Natural	Adversarial	Corruption	Sketch
ResNet-50:	76.2%	29.3%	39.2%	24.1%
ViT-B/16:	84.0%	61.2%	52.3%	32.4%

Difference:                    +7.8%    +31.9%                    +13.1%                    +8.3%

# ViTs are significantly more robust!

## Shape vs Texture Bias

```
def measure_shape_texture_bias(model):
 """Test if model relies on shape or texture"""

 # Create images with conflicting shape/texture
 # e.g., elephant texture on cat shape

 results = {
 "ResNet-50": {"shape": 0.23, "texture": 0.77}, # Texture biased
 "ViT-B/16": {"shape": 0.45, "texture": 0.55}, # More balanced
 "Human": {"shape": 0.96, "texture": 0.04} # Shape
 }

 biased = {}

 # ViTs are closer to human vision!
```

## Memory and Storage Requirements

```
Model sizes and memory usage

def calculate_memory_usage(model_name, batch_size=32):
 """Calculate GPU memory requirements"""

 memory = {
 # Storage (parameters)
 "ResNet-50": {"params_mb": 98, "gradients_mb": 98},
 "ViT-B/16": {"params_mb": 346, "gradients_mb": 346},

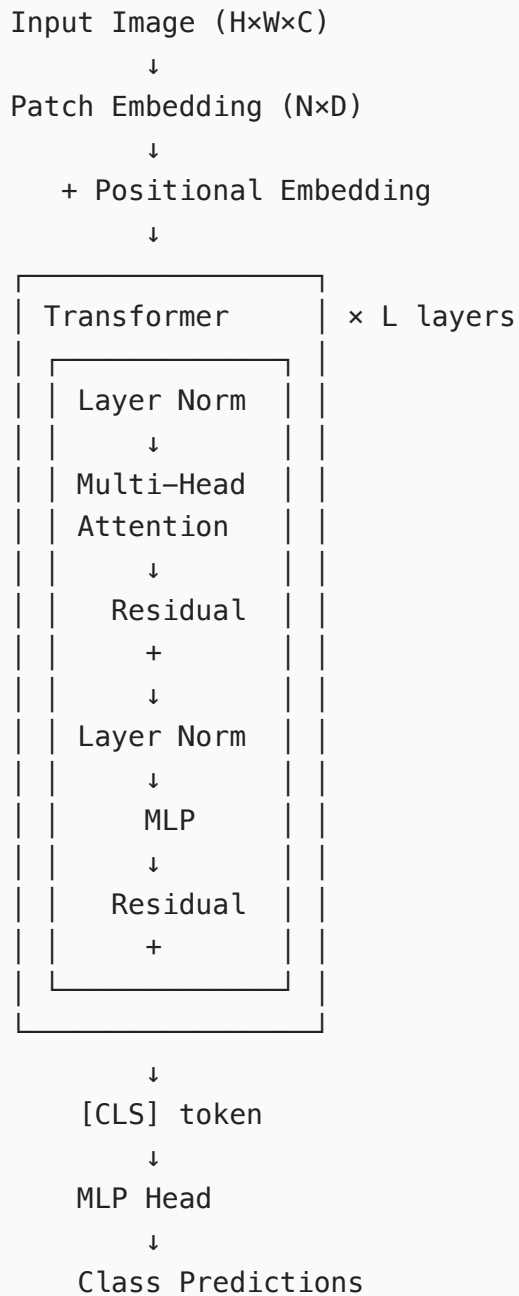
 # Activations (depends on batch size)
 # For batch_size=32, image_size=224
 "ResNet-50": {"activations_mb": 412},
 "ViT-B/16": {"activations_mb": 856},

 # Attention matrices (ViT only)
 "ViT-B/16": {"attention_mb": batch_size * 12 * 196 * 196 * 4 /
1e6}
 }

 # Total for training (params + grads + activations + optimizer state)
 resnet_total = 98 + 98 + 412 + 196 # ~804MB
 vit_total = 346 + 346 + 856 + 692 + 147 # ~2387MB
```

## 9. ViT Architecture: Complete Picture

### Standard ViT Architecture



### Key Components

```
class VisionTransformer(nn.Module):
 def __init__(self, image_size=224, patch_size=16, num_classes=1000,
 dim=768, depth=12, heads=12, mlp_dim=3072):
 super().__init__()
 num_patches = (image_size // patch_size) ** 2
```

```

self.patch_embed = PatchEmbed(patch_size, dim)
self.cls_token = nn.Parameter(torch.zeros(1, 1, dim))
self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1,
dim))

self.transformer = Transformer(dim, depth, heads, mlp_dim)
self.mlp_head = nn.Linear(dim, num_classes)

def forward(self, x):
 # Embed patches
 x = self.patch_embed(x) # (B, N, D)

 # Add [CLS] token
 cls_tokens = self.cls_token.expand(x.shape[0], -1, -1)
 x = torch.cat((cls_tokens, x), dim=1)

 # Add positional embedding
 x = x + self.pos_embed

 # Transformer blocks
 x = self.transformer(x)

 # Classification head
 x = x[:, 0] # [CLS] token
 x = self.mlp_head(x)
 return x

```

---

## Fine-tuning Pre-trained ViT

```

def finetune_vit(pretrained_model, num_classes, image_size=384):
 """Fine-tune pre-trained ViT for new task"""

 # Load pre-trained model
 model = timm.create_model(pretrained_model, pretrained=True)

 # Handle different image size
 if image_size != 224:

 # Interpolate position embeddings
 pos_embed = interpolate_pos_encoding(
 model.pos_embed,
 image_size // model.patch_size
)
 model.pos_embed = nn.Parameter(pos_embed)

 # Replace classification head

```



```

model.head = nn.Linear(model.embed_dim, num_classes)

Different learning rates for different parts
param_groups = [
 {'params': model.patch_embed.parameters(), 'lr': 1e-5}, # Lower
LR
 {'params': model.blocks[:-2].parameters(), 'lr': 1e-5}, # Lower
LR
 {'params': model.blocks[-2:].parameters(), 'lr': 1e-4}, # Medium
LR
 {'params': model.head.parameters(), 'lr': 1e-3}, # Higher LR
]

optimizer = torch.optim.AdamW(param_groups, weight_decay=0.05)

return model, optimizer

```

## Efficient ViT Variants

### 1. DeiT: Data-Efficient Image Transformers

```

class DeiT(VisionTransformer):
 """ViT with distillation token for knowledge transfer"""

 def __init__(self, *args, **kwargs):
 super().__init__(*args, **kwargs)
 self.dist_token = nn.Parameter(torch.randn(1, 1, self.dim))
 # Update pos_embed for extra token
 self.pos_embed = nn.Parameter(
 torch.randn(1, self.num_patches + 2, self.dim)
)

 def forward(self, x):
 # Add both cls and distillation tokens
 cls_tokens = self.cls_token.expand(x.shape[0], -1, -1)
 dist_tokens = self.dist_token.expand(x.shape[0], -1, -1)
 x = torch.cat((cls_tokens, dist_tokens, x), dim=1)
 # ... rest of forward pass

```

### 2. Swin Transformer: Hierarchical with Shifted Windows

```

class SwinTransformerBlock(nn.Module):
 """Swin Transformer block with shifted window attention"""

 def __init__(self, dim, num_heads, window_size=7, shift_size=0):
 super().__init__()
 self.dim = dim

```

```

self.num_heads = num_heads
self.window_size = window_size
self.shift_size = shift_size

self.norm1 = nn.LayerNorm(dim)
self.attn = WindowAttention(dim, window_size, num_heads)
self.norm2 = nn.LayerNorm(dim)
self.mlp = FeedForward(dim, dim * 4)

def forward(self, x, H, W):
 # Save shape
 B, L, C = x.shape

 # Reshape to image
 x = x.view(B, H, W, C)

 # Cyclic shift
 if self.shift_size > 0:
 x = torch.roll(x, shifts=(-self.shift_size, -self.shift_size),
dims=(1, 2))

 # Partition windows
 x_windows = window_partition(x, self.window_size)

 # Window attention
 attn_windows = self.attn(x_windows)

 # Merge windows
 x = window_reverse(attn_windows, self.window_size, H, W)

 # Reverse cyclic shift
 if self.shift_size > 0:
 x = torch.roll(x, shifts=(self.shift_size, self.shift_size),
dims=(1, 2))

 # Reshape back
 x = x.view(B, H * W, C)

 # Residual connections
 x = x + self.norm1(x)
 x = x + self.mlp(self.norm2(x))

 return x

```

## 10. Computational Complexity Analysis

### Complexity Comparison

Operation	CNN	ViT
Per Layer	$O(k^2 \cdot H \cdot W \cdot C_{\text{in}} \cdot C_{\text{out}})$	$O(N^2 \cdot D + N \cdot D^2)$
Memory	$O(k^2 \cdot C_{\text{in}} \cdot C_{\text{out}})$	$O(N^2 + N \cdot D)$
Receptive Field Growth	$O(\sqrt{L})$	$O(1)$ - immediate global

## Concrete Example: ImageNet

# ResNet-50

Parameters: 25.6M

FLOPs: 4.1G

Memory: ~100MB

# ViT-B/16

Parameters: 86.6M

FLOPs: 17.6G

Memory: ~350MB

# But ViT achieves better accuracy with sufficient pre-training!

## Scaling Behaviour

As image resolution increases:

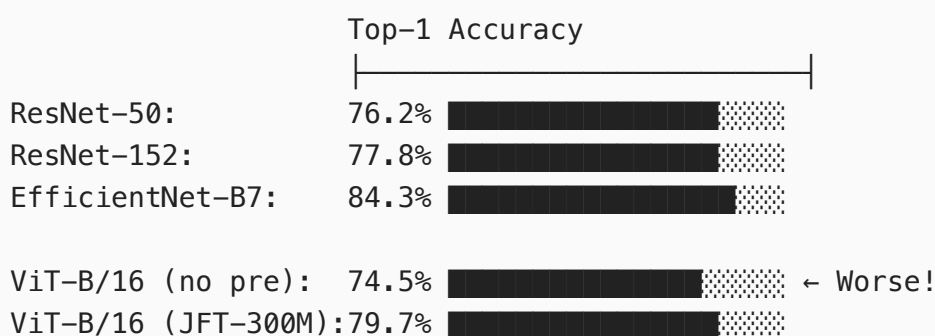
CNN:  $O(N)$  - Linear in pixels

ViT:  $O(N^2)$  - Quadratic in patches

Solution: Hierarchical ViTs (Swin Transformer)

## 11. Performance: The Complete Picture

### ImageNet Results



ViT-L/16 (JFT-300M): 87.5% ██████████ ← Best!  
ViT-H/14 (JFT-3B): 88.6% ██████████

## Key Observations

1. **Data Requirements:** ViT needs ~100× more data than CNNs
2. **Scaling:** ViTs scale better with data and compute
3. **Transfer Learning:** Pre-trained ViTs transfer exceptionally well
4. **Robustness:** ViTs more robust to occlusions, perturbations

## 12. Practical Insights & Trade-offs

### When to Use What?

Error parsing Mermaid diagram!

Cannot read properties of null (reading 'getBoundingClientRect')

## Hybrid Approaches

### 1. CNN + Transformer

Early layers: CNN (extract local features efficiently)  
Later layers: Transformer (model global relationships)  
Example: LeViT, CvT

### 2. Hierarchical ViT

Swin Transformer: Windowed attention + shifting

- Reduces  $O(N^2)$  to  $O(N)$
- Maintains global receptive field
- Better for dense tasks

### 3. Convolutional Stem

```
Replace patch embedding with Conv layers
self.conv_stem = nn.Sequential(
 nn.Conv2d(3, 64, 3, 2, 1),
 nn.Conv2d(64, 128, 3, 2, 1),
 nn.Conv2d(128, dim, 3, 2, 1)
)
Reduces patches, adds inductive bias
```

## 13. Recent Advances & Future Directions

### Efficient Attention Mechanisms

```
Linear Attention: $O(N)$ instead of $O(N^2)$
attention = (Q @ K.T).softmax(dim=-1) @ V # Standard $O(N^2)$
attention = Q @ (K.T @ V) # Linear $O(N)$

Sparse Attention: Attend to subset
Local + Global attention patterns
```

### Key Innovations

1. **DeiT** (Data-efficient ViT): Knowledge distillation from CNNs
2. **Swin Transformer**: Hierarchical architecture with shifted windows
3. **MLP-Mixer**: Replace attention with MLPs
4. **ConvNeXt**: Modernized CNN inspired by ViT designs

### Open Questions

- Can we reduce ViT data requirements?
- Optimal patch size?
- Better positional encodings?
- Unified architecture for all vision tasks?

### Efficient Attention Mechanisms

#### 1. Linear Attention (Performer)

```
class LinearAttention(nn.Module):
 """ $O(N)$ complexity instead of $O(N^2)$ """

 def __init__(self, dim, heads=8):
 super().__init__()
 self.heads = heads
 self.dim_head = dim // heads
 self.to_qkv = nn.Linear(dim, dim * 3)

 def forward(self, x):
 b, n, _ = x.shape
 qkv = self.to_qkv(x).chunk(3, dim=-1)
 q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d',
 h=self.heads), qkv)
```

```

Apply random features for linear attention
q = self.kernel_function(q)
k = self.kernel_function(k)

O(N) attention: (Q @ K.T) @ V becomes Q @ (K.T @ V)
kv = torch.einsum('bhnd,bhne->bhde', k, v)
out = torch.einsum('bhnd,bhde->bhne', q, kv)

Normalize
q_sum = q.sum(dim=-1, keepdim=True)
out = out / q_sum

return rearrange(out, 'b h n d -> b n (h d)')

```

## 2. Flash Attention (Memory Efficient)

```

Flash Attention: Fused kernel for memory efficiency
Reduces memory from O(N²) to O(N)
from flash_attn import flash_attn_func

class FlashAttention(nn.Module):
 def __init__(self, dim, heads=8):
 super().__init__()
 self.heads = heads
 self.to_qkv = nn.Linear(dim, dim * 3)

 def forward(self, x):
 b, n, _ = x.shape
 qkv = self.to_qkv(x)
 qkv = rearrange(qkv, 'b n (three h d) -> b n three h d',
 three=3, h=self.heads)

 # Use flash attention (fused CUDA kernel)
 out = flash_attn_func(qkv[:, :, 0], qkv[:, :, 1], qkv[:, :, 2])
 return rearrange(out, 'b n h d -> b n (h d)')

```

## Multi-Scale Vision Transformers

### Pyramid Vision Transformer (PVT)

```

class PVT(nn.Module):
 """Multi-scale feature pyramid with transformers"""

 def __init__(self, img_size=224, num_classes=1000):
 super().__init__()

```

```

4 stages with decreasing resolution
self.stages = nn.ModuleList([
 # Stage 1: H/4 × W/4
 PVTStage(dim=64, patches_resolution=56, depth=3, num_heads=1),
 # Stage 2: H/8 × W/8
 PVTStage(dim=128, patches_resolution=28, depth=4,
num_heads=2),
 # Stage 3: H/16 × W/16
 PVTStage(dim=320, patches_resolution=14, depth=6,
num_heads=5),
 # Stage 4: H/32 × W/32
 PVTStage(dim=512, patches_resolution=7, depth=3, num_heads=8),
])

self.head = nn.Linear(512, num_classes)

def forward(self, x):
 features = []
 for stage in self.stages:
 x = stage(x)
 features.append(x)

 # Use final stage for classification
 x = x.mean(dim=1)
 return self.head(x), features # Return multi-scale features

```

## Cross-Modal Vision Transformers

### CLIP-style Vision-Language Model

```

class CLIPVisionTransformer(nn.Module):
 """Vision transformer for multi-modal learning"""

 def __init__(self, image_size=224, patch_size=16, dim=512):
 super().__init__()
 self.vit = VisionTransformer(
 image_size=image_size,
 patch_size=patch_size,
 dim=dim,
 pool='cls'
)

 # Project to shared embedding space
 self.projection = nn.Linear(dim, 512)

 def forward(self, images, return_features=False):
 # Get image features

```

```

 features = self.vit(images)

 # Project to shared space
 embeddings = self.projection(features)
 embeddings = F.normalize(embeddings, dim=-1)

 if return_features:
 return embeddings, features
 return embeddings

def contrastive_loss(image_embeddings, text_embeddings, temperature=0.07):
 """CLIP-style contrastive loss"""
 # Normalized embeddings
 image_embeddings = F.normalize(image_embeddings, dim=-1)
 text_embeddings = F.normalize(text_embeddings, dim=-1)

 # Cosine similarity as logits
 logits = torch.matmul(image_embeddings, text_embeddings.T) /
temperature

 # Symmetric loss
 labels = torch.arange(len(logits)).to(logits.device)
 loss_i2t = F.cross_entropy(logits, labels)
 loss_t2i = F.cross_entropy(logits.T, labels)

 return (loss_i2t + loss_t2i) / 2

```

## Self-Supervised Learning with ViT

### Masked Autoencoder (MAE)

```

class MAE(nn.Module):
 """Masked Autoencoder for self-supervised learning"""

 def __init__(self, encoder, decoder_dim=512, mask_ratio=0.75):
 super().__init__()
 self.encoder = encoder
 self.mask_ratio = mask_ratio

 # Decoder (lighter than encoder)
 self.decoder = nn.TransformerDecoder(
 nn.TransformerDecoderLayer(decoder_dim, 8),
 num_layers=4
)

 # Prediction head
 self.prediction_head = nn.Linear(decoder_dim, patch_dim)

```



```

def forward(self, x):
 # Get patches
 patches = self.encoder.to_patch_embedding(x)
 B, N, D = patches.shape

 # Random masking
 num_mask = int(N * self.mask_ratio)
 indices = torch.randperm(N)
 mask_indices = indices[:num_mask]
 keep_indices = indices[num_mask:]

 # Keep only visible patches for encoder
 visible_patches = patches[:, keep_indices]

 # Encode
 encoded = self.encoder.transformer(visible_patches)

 # Decode (with mask tokens)
 mask_tokens = self.mask_token.repeat(B, num_mask, 1)
 full_sequence = torch.cat([encoded, mask_tokens], dim=1)

 # Restore original order
 full_sequence = restore_order(full_sequence, indices)

 # Decode
 decoded = self.decoder(full_sequence)

 # Predict masked patches
 predictions = self.prediction_head(decoded)

 # Compute reconstruction loss (only on masked patches)
 loss = F.mse_loss(predictions[:, mask_indices], patches[:,
mask_indices])

 return loss, predictions

```

## Future Directions

### 1. Unified Architecture for All Vision Tasks

```

class UniversalViT(nn.Module):
 """Single model for classification, detection, segmentation"""

 def __init__(self):
 super().__init__()
 self.backbone = VisionTransformer(...)

 # Task-specific heads

```

```

 self.heads = nn.ModuleDict({
 'classification': ClassificationHead(),
 'detection': DetectionHead(),
 'segmentation': SegmentationHead(),
 'depth': DepthHead(),
 })

 def forward(self, x, task='classification'):
 features = self.backbone(x, return_all_patches=True)
 return self.heads[task](features)

```

## 2. Neural Architecture Search for ViTs

```

AutoFormer: Automated transformer architecture search
search_space = {
 'depth': [12, 14, 16, 18],
 'embed_dim': [384, 512, 640, 768],
 'num_heads': [6, 8, 10, 12],
 'mlp_ratio': [3.0, 3.5, 4.0],
 'patch_size': [14, 16, 18],
}

```

## 3. Efficient Deployment

```

Quantization for edge devices
quantized_vit = torch.quantization.quantize_dynamic(
 model, {nn.Linear}, dtype=torch.qint8
)

Knowledge distillation to smaller models
tiny_vit = distill_from_teacher(
 teacher=vit_huge,
 student=vit_tiny,
 temperature=3.0
)

Token pruning during inference
def adaptive_token_pruning(x, keep_ratio=0.5):
 """Keep only important tokens"""
 importance_scores = compute_token_importance(x)
 keep_indices = torch.topk(importance_scores, int(len(x) * keep_ratio))
 return x[keep_indices]

```

---



---

# 14. Summary: The Paradigm Shift

## From Local to Global

CNNs:

Local → Hierarchical → Global

Strong priors

Data efficient

Hardware optimized

ViTs:

Global from the start

Minimal priors

Compute efficient (at scale)

Emerging optimizations

## Key Takeaways

1. **ViT treats images as sequences** of patches (tokens)
2. **Attention enables global interactions** from layer 1
3. **Weak inductive bias** requires more data but enables flexibility
4. **Superior scaling** with data and compute
5. **Not always better** - depends on your constraints

## The Future is Hybrid

The best models increasingly combine:

- CNN efficiency for local patterns
- Transformer flexibility for global reasoning
- Task-specific architectural biases

---

## Practical Recommendations

### For Your Projects

```
Small dataset (<50K images)
model = torchvision.models.resnet50(pretrained=True)

Medium dataset with compute
model = timm.create_model('convnext_base', pretrained=True)

Large dataset or transfer learning
model = timm.create_model('vit_base_patch16_224', pretrained=True)

Dense prediction tasks
model = timm.create_model('swin_base_patch4_window7_224')
```

```
Mobile/edge deployment
model = timm.create_model('efficientnet_b0')
```

## Key Hyperparameters to Tune

1. **Patch size:** Smaller → more patches → more compute but finer details
  2. **Embedding dim:** Match model size to data availability
  3. **Number of heads:** 12 often optimal
  4. **Depth vs width:** Deeper better for complex tasks
  5. **Drop path rate:** Critical for training deep ViTs
- 

## Extension Points for Advanced Topics

### Potential Deep Dives

1. **Mathematical Foundations**
    - Attention as kernel regression
    - Connection to graph neural networks
    - Fourier analysis of positional encodings
  2. **Advanced Architectures**
    - Cross-attention for multi-modal (CLIP)
    - Hierarchical vision transformers (Swin, PVT)
    - Efficient attention mechanisms (Performer, Linformer)
  3. **Training Dynamics**
    - Why ViTs are hard to optimize
    - Role of LayerNorm placement
    - Attention pattern evolution during training
  4. **Interpretability**
    - Attention visualization techniques
    - What do ViTs learn vs CNNs?
    - Probing intermediate representations
  5. **Applications Beyond Classification**
    - Object detection (DETR)
    - Segmentation (SegFormer)
    - Video understanding (TimeSformer)
- 

## Code Example: Minimal ViT Implementation

```

import torch
import torch.nn as nn

class SimpleViT(nn.Module):
 def __init__(self, image_size=224, patch_size=16, num_classes=10,
 dim=512, depth=6, heads=8, mlp_ratio=4):
 super().__init__()
 assert image_size % patch_size == 0
 num_patches = (image_size // patch_size) ** 2
 patch_dim = 3 * patch_size ** 2

 self.patch_size = patch_size
 self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1,
dim))
 self.patch_to_embedding = nn.Linear(patch_dim, dim)
 self.cls_token = nn.Parameter(torch.randn(1, 1, dim))

 self.transformer = nn.TransformerEncoder(
 nn.TransformerEncoderLayer(
 d_model=dim,
 nhead=heads,
 dim_feedforward=dim * mlp_ratio,
 batch_first=True
),
 num_layers=depth
)

 self.to_cls_token = nn.Identity()
 self.mlp_head = nn.Sequential(
 nn.LayerNorm(dim),
 nn.Linear(dim, num_classes)
)

 def forward(self, img):
 B, C, H, W = img.shape
 P = self.patch_size

 # Create patches
 patches = img.unfold(2, P, P).unfold(3, P, P) # (B, C, H/P, W/P,
P, P)
 patches = patches.contiguous().view(B, C, -1, P*P) # (B, C,
num_patches, P*P)
 patches = patches.permute(0, 2, 1, 3) # (B, num_patches, C, P*P)
 patches = patches.flatten(2) # (B, num_patches, C*P*P)

 # Embed patches
 x = self.patch_to_embedding(patches)

 # Add cls token

```

```
cls_tokens = self.cls_token.expand(B, -1, -1)
x = torch.cat((cls_tokens, x), dim=1)

Add positional embedding
x += self.pos_embedding

Transformer
x = self.transformer(x)

Classification
return self.mlp_head(x[:, 0])

Test
model = SimpleViT(image_size=32, patch_size=8, num_classes=10, dim=128,
depth=4)
x = torch.randn(2, 3, 32, 32)
out = model(x)
print(f"Output shape: {out.shape}") # [2, 10]
```

---

## References & Further Reading

### Essential Papers

- [Attention Is All You Need](#) - Original Transformer
- [An Image is Worth 16x16 Words](#) - Vision Transformer
- [Training data-efficient image transformers](#) - DeiT
- [Swin Transformer](#) - Hierarchical Vision Transformer
- [Masked Autoencoders Are Scalable Vision Learners](#) - MAE
- [A ConvNet for the 2020s](#) - ConvNeXt

### Practical Resources

- [timm library](#) - Pre-trained ViTs, best implementations
  - [Hugging Face Transformers](#) - Easy to use
  - [Annotated ViT](#) - Clear implementation
  - [ViT Tutorial](#) - Hugging Face guide
-