Building Vision Transformers (ViT) from Scratch

Maninder Singh Follow 40 min read · 5 days ago

260 3

I still remember the day I first read *An Image is Worth 16x16 Words* and felt my brain do a little flip. Transformers had already turned NLP on its head — so what happens if you hand that same machinery an image and tell it to "read" it like a sentence? The paper's central dare was deceptively simple: **what if we stopped using convolutions altogether?** Chop the image into patches, treat each patch like a word, and let a Transformer learn the relationships between them.

That idea sounded almost too clever to be real. But the results convinced me: with enough data and compute, Vision Transformers (ViT) can out-perform CNNs on large benchmarks. I wanted to understand *why* — and more importantly, I wanted to stop passively nodding at equations and actually build one myself. That idea fascinated me. I was already curious about how Transformers might be applied beyond text, and I also wanted to push myself to not just *read papers*, but actually *convert them into code*. For me, this

was about going deeper — really understanding how models and algorithms work under the hood.

So I did what felt natural: I took the paper, opened PyTorch, and started implementing each piece by hand — no pretrained black boxes, no convenience wrappers — just the paper as my map and a stubborn willingness to debug until things made sense.

What a ViT really is ?

At a high level, ViT replaces the sliding-window, kernel-based scan of a CNN with a very different approach:

- You chop an image into fixed-size patches (think of breaking a chocolate bar into squares).
- Each patch is flattened and projected into an embedding vector — now each patch behaves like a token or a “word” in a sentence.

- Because transformers are order-agnostic, you add positional information so the model knows *where* each patch came from.
- Those patch tokens go through Transformer encoder blocks, which let every patch talk to every other patch — capturing both local texture and long-range relationships.
- Finally, a classification head reads the transformed tokens and says, "That's Tom" or "That's Jerry."

The tradeoff that made me cautious (and curious)

This design flips some core inductive biases on their head:

- **CNNs** bake in locality and translation equivariance. Those assumptions help a lot when your dataset is small or medium — the model already "knows" some useful things about images.
- **ViTs** have much weaker inductive biases. That's powerful because they can learn long-range dependencies naturally, but the flip side is they usually need **a lot more data** (or heavier regularization) to reach the same reliability.

That tradeoff is the theme of this whole project: can we build a clean, paper-aligned ViT that's small and disciplined enough to actually run on a laptop — and still learn something meaningful on a modest dataset?

I wanted to make this project hands-on and fun — something I could actually run on my laptop while understanding every moving part. So I built a **simplified Vision Transformer in PyTorch**, line by line, guided only by the original paper. No pretrained shortcuts, no high-level wrappers — just raw tensors, attention scores, and positional embeddings that I had to wrestle into shape.

To test it, I picked a small **Tom & Jerry dataset from Kaggle** 🐭🐱. It's nowhere near ImageNet or JFT-300M dataset scale, but that's what made it perfect — a lightweight playground to see how ViTs behave with limited data and what tricks are needed to make them learn.

In this blog, I'll walk you through the exact process — the working parts, the bugs that broke everything, and the "aha" moments that tied it all together.

Picture an image being split into small patches, each turned into an embedding, sprinkled with positional info, passed through Transformer layers, and finally classified as "Tom" or "Jerry."

By the end, you'll have a **fully working ViT implementation**, a solid intuition for each component, and a few practical lessons for running ViTs on your own datasets.

Now, let's start by tearing an image into patches and see how the model reacts.

🔗 ***ALL THE CODE CAN BE FOUND IN MY GITHUB REPOSITORY FROM HERE***

GitHub - manindersingh120996/vision-transformer-from-scratch

Contribute to manindersingh120996/vision-transformer-from-scratch development by creating an account on GitHub.

github.com

Table of Contents

Project Snapshot & Reproducibility

Before we start building our Vision Transformer architecture, let's pause and answer a very practical question:
"Can you actually run this project on your own machine?"

That's important because a lot of papers and tutorials sound great in theory but fall apart when you try to reproduce them. My goal with this build was to

make it dead simple to install, train, and experiment — without needing to hack around with hidden internals.

The project is organized into a few clean modules:

```
vit-project/
│
├── configs/        # Hydra config files (model, training, dataset)
├── src/            # Model and dataset code (modular PyTorch classes)
├── train.py        # Training loop, powered by Hydra
├── inference.py    # Script to run inference with a trained model
├── app.py          # Streamlit app for training & inference (UI)
├── outputs/        # Logs, checkpoints, plots from single runs
├── multirun/       # Logs from multiple experiments
└── requirements.txt
```

Getting started

- **Clone & install**

```
git clone https://github.com/manindersingh120996/vision-transformer-from-scratch
cd vit-project
pip install -r requirements.txt
```

- **Train a model (Hydra single-run)**
  Hydra keeps all hyperparameters in YAML configs but also allows overrides from the command line:

```
# Run with default config
```

```
python train.py
```

```
# Override config from CLI
python train.py optimizer.lr=0.0005 model.num_layers=8
```

- **Run multiple experiments (Hydra multirun)**

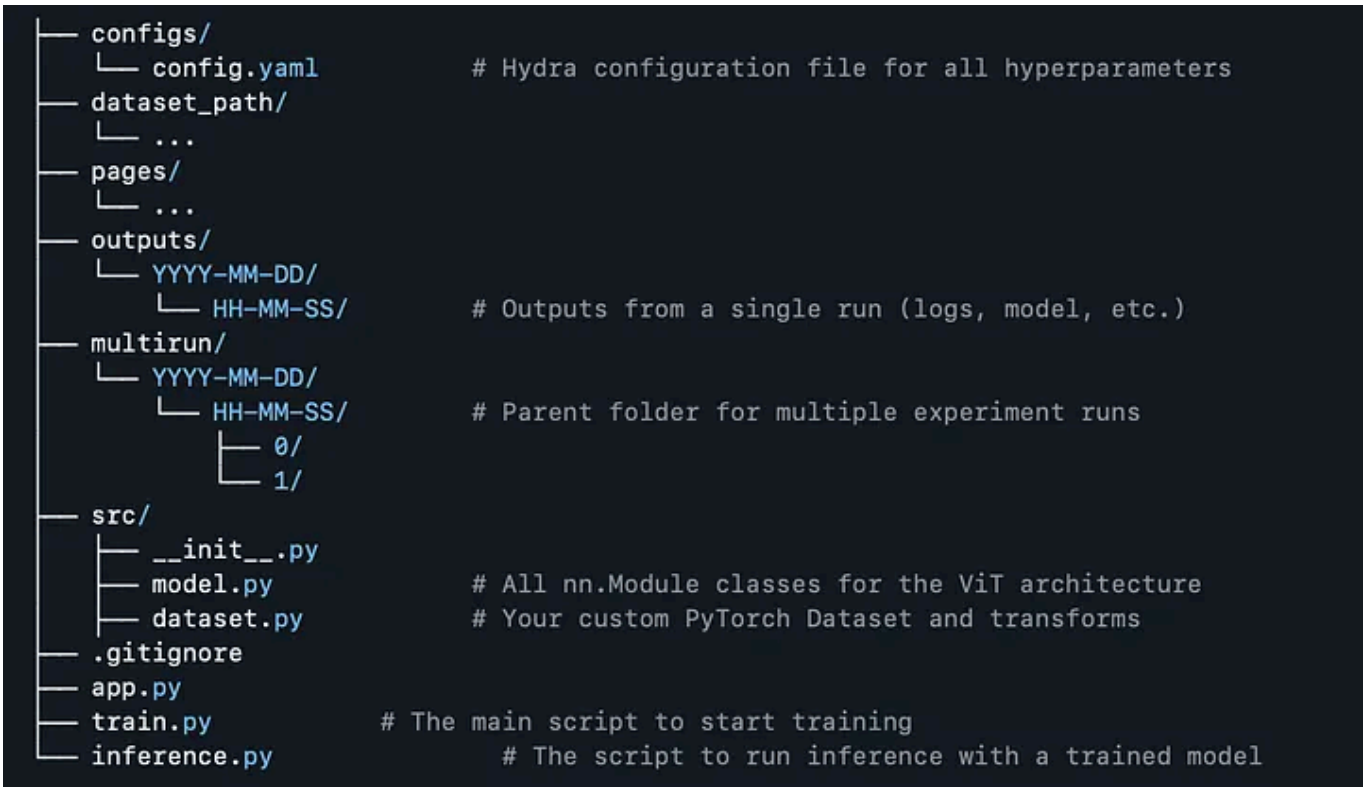
```
python train.py --multirun model.num_layers=4,6,8 optimizer.lr=0.001,0.0005
```

This launches several experiments back-to-back and stores results neatly in `multirun/`.

- **Optional: Streamlit front end**
  If you prefer a simple UI instead of configs and command lines, launch the Streamlit app:

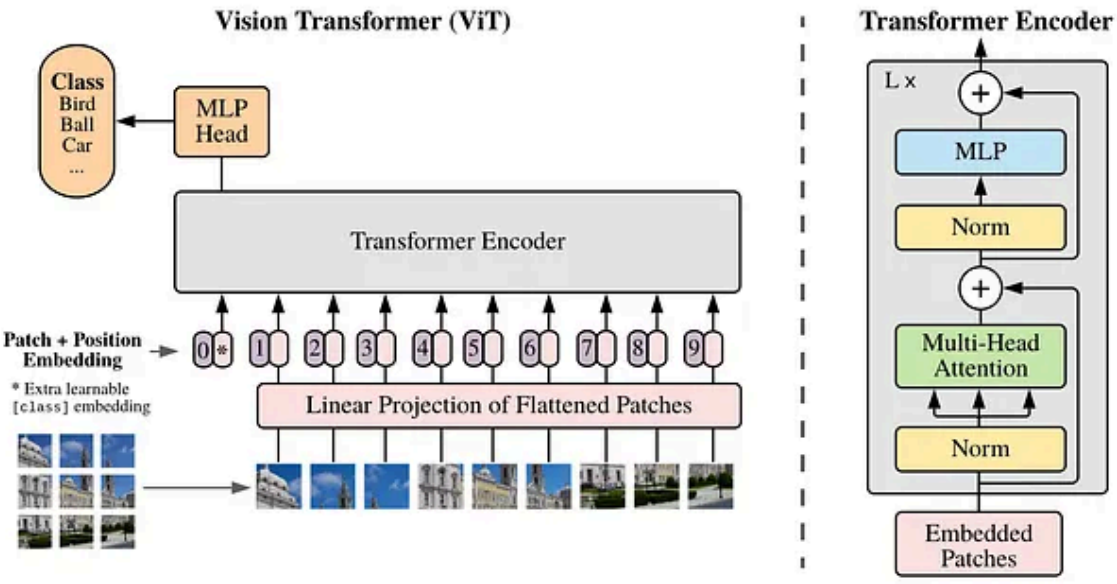
```
streamlit run app.py
```

Here you can pick hyperparameters from dropdowns, watch real-time loss curves, and test inference with drag-and-drop images — no terminal required.



Model Architecture (`model.py`)

Let’s start building our Vision Transformer (ViT) model. To keep things modular and easy to follow, I have broken the implementation into several sub-classes, each handling one key component of the architecture.



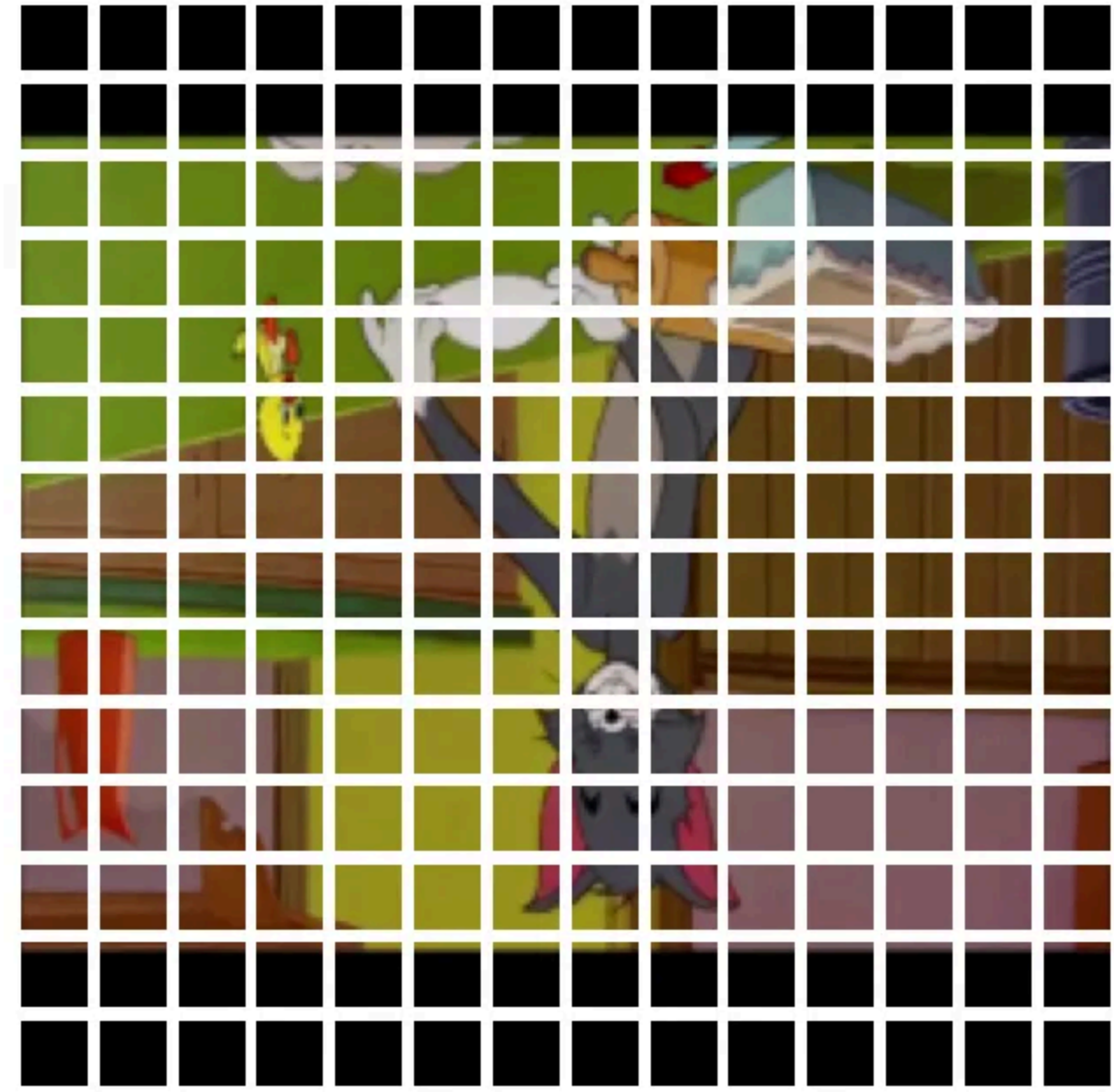
ViT Image from the Paper

We begin with patch creation, where the input image is divided into smaller fixed-size patches. From there, we construct the input layer that combines patch embeddings, positional encodings, and the special classification (CLS) token.

After that, we'll move on to layer normalization, self-attention from scratch, feed-forward layers, and so on — exactly as described in the original ViT paper. By doing this step by step, we not only implement the model but also develop an intuition for why each part exists and how it fits into the bigger picture.

Patch Embeddings: Turning Images into Tokens

One of the most surprising and elegant ideas in the Vision Transformer paper (*"An Image is Worth 16×16 Words"*, Dosovitskiy et al., 2020) is that images can be treated almost like text. Instead of feeding raw pixels into a convolutional neural network, the image is split into small, fixed-size patches, and each patch is treated as if it were a "word" in a sentence.



Think about it this way:

- In **NLP**, a sentence is a sequence of words.
- In **ViT**, an image is a sequence of patches.

This reframing allows the transformer — originally designed for sequential text data — to directly process visual inputs.

Mathematical Formulation (as per the paper, Section 3.1):

If we have an image of resolution **H × W** with **C** channels (e.g., 3 for RGB), and we split it into patches of size **P × P**, then:

- Each patch is a vector in **R^(P²·C)** after flattening.
- The total number of patches is:

- Each patch is a vector in **R^(P²·C)** after flattening.
- The total number of patches is:

$$N = \frac{HW}{P^2}$$

So the set of patches can be written as:

$$x_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$$

where:

- **P** = patch size (e.g., 16×16),
- **C** = number of channels (3 for RGB),
- **N** = number of patches in the image,
- Each flattened patch has dimension **P²·C**.

For example, if we take an image of **224 × 224 × 3** and choose a patch size **P = 16**, then:

- Each patch is of size **16 × 16 × 3 = 768** values.
- The total number of patches is **(224 / 16) × (224 / 16) = 14 × 14 = 196 patches**.

So this image is represented as a sequence of **196 tokens**, each of length **768-> (now this 768 is through calclulation but as per paper we can have it any value set in output paramter of Conv2d layer as learnable paramter.)**, ready to be projected into the embedding space of the transformer

Important: 768 is the flattened patch size; the transformer's embedding dimension `D` is a separate, learnable size — we project each 768-length vector into `D`.

Now let's see its' implementation in code :

```
class PatchCreation(nn.Module):
    def __init__(self,
                 input_color_channel: int,
                 patch_size: int,
                 embedding_dimensions : int):
        super().__init__()
        self.patch_size = patch_size
        # use Conv2d to simultaneously extract and project patches
        self.patching_conv = nn.Conv2d(
            input_color_channel,
            embedding_dimensions,
            kernel_size=patch_size,
```



```
    def forward(self, x):
        image_dimension = x.shape[-1]
        assert image_dimension % self.patch_size == 0, \
            f"Given image dimension {image_dimension} is not divisible by patch
        if len(x.shape) == 3:  # allow input in (C, H, W) by adding batch dim
            x = x.unsqueeze(0)
        return self.patching_conv(x).flatten(2).permute(0, 2, 1)
```

> **Quick F*avor* 🥲*: If you found this article helpful, consider following me, clapping* 👏*, or sharing it with others who might benefit. Thank you! . Let's get back to the article***

So what we basicallhy do in this code is that :

- **Split the image into non-overlapping patches** of size `patch_size × patch_size`.
- **Project each patch to an embedding vector** of length `embedding_dimensions`.
- **Return a sequence of token embeddings** shaped `(batch_size, num_patches, embedding_dim)`, ready to be summed with positional embeddings and fed into the Transformer encoder.

You might be having a question that why are we using Conv2d in here?,

the reason is simple that the paper describes flattening each patch and applying a learned linear projection `E` to go from $P^2 \cdot C \rightarrow D$. We can implement that with `unfold + Linear`, which is explicit and instructive. But I found a much cleaner option: a `Conv2d` with `kernel_size=P` and `stride=P`. Each convolution window exactly covers one patch, and the conv filters act as the linear projection into the embedding space.

Why I like the `Conv2d` approach:

- One layer does both **extracting** and **projecting** patches.
- It uses optimized convolution kernels (fast).
- The code is shorter and easier to read.

Let's also visit the sape transformation once as well :

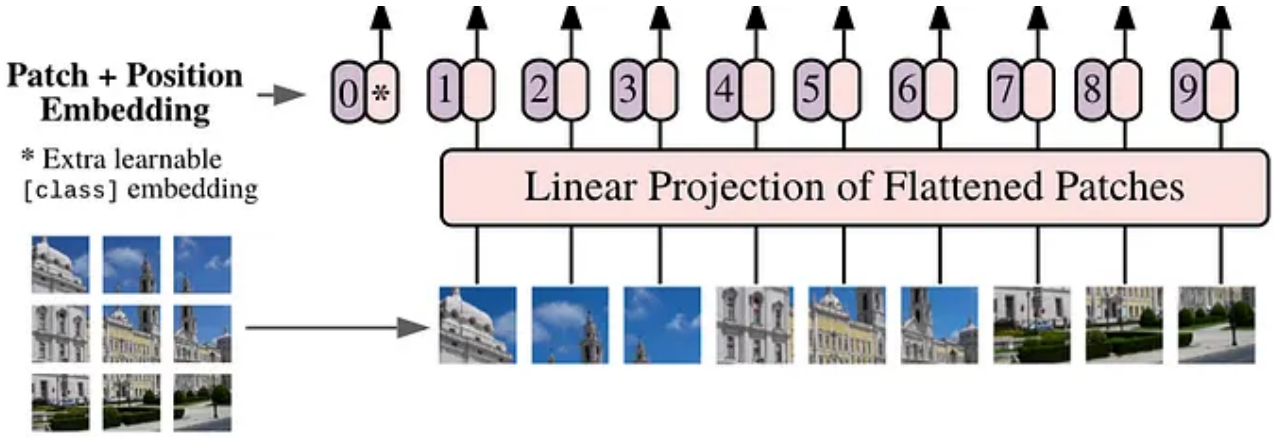
- **Input image:** `(B, C, H, W)`
- If you pass a single image `(C, H, W)`, the code adds a batch dimension → `(1, C, H, W)`.
- **After `patching_conv`:** `(B, D, H/P, W/P)`
- **D** = embedding dimension.
- **H/P × W/P** gives the grid of patches.
- **After `flatten(2)`:** `(B, D, N)`
- **N = (H/P) × (W/P)** = total number of patches.
- **After `permute(0, 2, 1)`:** `(B, N, D)`
- Sequence of patch embeddings in the exact format expected by a transformer.

This completes the first step of our ViT construction: turning images into tokens through patch embeddings. From here, we are ready to move on to the input layer, where embeddings, positional information, and the classification token are combined to form the sequence that will be processed by the transformer.

Input Layer: CLS Token and Positional Embeddings

In the previous section, we saw how an image can be split into fixed-size patches and projected into embedding vectors. This gave us a sequence of tokens, much like words in a sentence. However, before we feed this sequence into the transformer encoder, we need to make two crucial

modifications as it still had no idea **where** each piece came from. It's like handing someone 196 cropped tiles of a puzzle and asking them to describe the whole picture.



1. The Classification Token (CLS)

Borrowing from BERT (Devlin et al., 2018), the ViT model introduces a learnable **[CLS] token.** This is a special embedding vector that is prepended to the sequence of patch embeddings.

- During training, the transformer learns to use this token as an aggregate representation of the entire image.
- At inference time, we just take the final hidden state of the CLS token and send it to a classification head (usually a small MLP) — and boom, that's our predicted class.

In other words:

- **Patch embeddings** → carry local information from image patches.

- **CLS token** → gathers and summarizes global information across the entire image sequence.

The paper explicitly notes this in Section 3.1:

> *"Similar to BERT's [class] token, we prepend a learnable embedding to the sequence of embedded patches, whose state at the output of the Transformer encoder serves as the image representation."*

2. Positional Embeddings

The Transformer has no notion of order by default — it just sees a set of embeddings. But images **do** have a natural spatial structure, and we don't want the model to treat them as if patches were independent and unordered.

To inject spatial information, we add **positional embeddings**.

In the original NLP Transformer paper (*Vaswani et al., 2017*), these were sinusoidal functions designed to allow the model to generalize to longer sequences. But in ViT, the authors went for something simpler and surprisingly effective: **learnable positional embeddings**.

Why this change?

- Images are **fixed-size** during training (say, 224×224). The number of patches doesn't vary. So we don't need generalization to variable sequence lengths.
- We don't need generalization to longer sequences, just accurate relative positioning.

- So instead of fixed sinusoids, we let the model *learn* where each patch belongs.

as per eqation in orginal paper:

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \tag{1}$$

The paper emphasizes this design choice:

> *"We add a learnable positional embedding to the patch embeddings to retain positional information."*(Section 3.1)

```
class ViTInputLayer(nn.Module):
    def __init__(self, in_channels: int,
                 patch_size: int,
                 image_size: int,
                 embedding_dimensions: int,
                 input_dropout_rate: float = 0.0):
        super().__init__()
        # Patch embedding module
        self.patch_embeddings = PatchCreation(in_channels,
                                              patch_size,
                                              embedding_dimensions)

        # Learnable CLS token
        self.cls_token = nn.Parameter(
            torch.randn(1, 1, embedding_dimensions),
            requires_grad=True
        )

        # Number of patches
        num_patches = (image_size // patch_size) ** 2
        num_positions = num_patches + 1  # +1 for CLS token

        # Learnable positional embeddings
        self.positional_embeddings = nn.Parameter(
            torch.randn(1, num_positions, embedding_dimensions),
            requires_grad=True
```

```
        )

        # Optional dropout (helps regularization)
        self.dropout = nn.Dropout(p=input_dropout_rate)

    def forward(self, x):
        batch_size = x.shape[0]

        # Step 1: Get patch embeddings
        patch_embeddings = self.patch_embeddings(x)

        # Step 2: Expand CLS token for batch size
        cls_token = self.cls_token.expand(batch_size, -1, -1)

        # Step 3: Concatenate CLS token + patch embeddings
        tokens = torch.concat((cls_token, patch_embeddings), dim=1)

        # Step 4: Add positional embeddings + apply dropout
        return self.dropout(tokens + self.positional_embeddings)
```

SO what actualy happens in here is that :

1. **We first generate the Patch embeddings** from the input image.
2. Then a **learnable CLS token** is prepended to the sequence.
3. **Then further we add the Positional embeddings** to it to retain spatial structure.
4. **Further Dropout** is applied to improve generalization, which is important.

Finally the result which we achieive is a tensor of shape `(batch_size, num_patches+1, embedding_dim)` — the complete input sequence expected by the transformer encoder.

Shape Transformations Recap

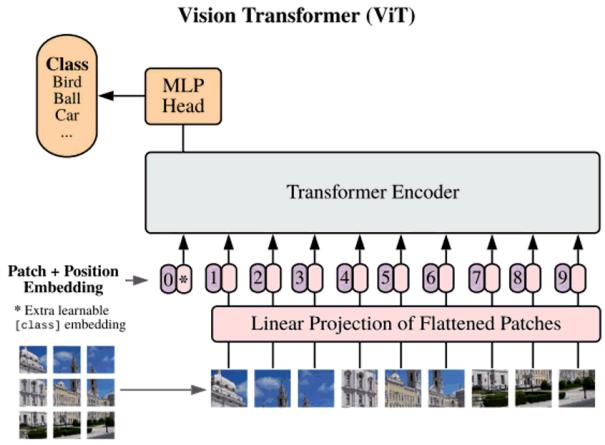
- Input image: `(B, C, H, W) IN PREVIOUS STEP`

- After patching (from previous step): `(B, N, D)` (sequence of patch embeddings)
- After adding CLS token: `(B, N+1, D)`
- After adding positional embeddings: `(B, N+1, D)` (final input to encoder)

Layer Normalization and Multi-Head Attention

With the input sequence ready (patch embeddings + CLS token + positional embeddings), the next step in the Vision Transformer is to process these tokens through the **Transformer encoder blocks.** Two foundational components here are **Layer Normalization** and **Multi-Head Self-Attention**, both of which are critical for stable training and learning complex dependencies across patches.

$$\mathbf{z'}_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \dots L \tag{2}$$



1. Layer Normalization

Unlike batch normalization, which normalizes across a batch, **Layer Normalization (LayerNorm)** normalizes across the feature dimension for each individual example.

In ViT (*Dosovitskiy et al., 2020*, Section 3.2), LayerNorm is applied **before the attention and MLP sub-layers**, also called *pre-norm*, which helps stabilize training for deep Transformers.

Implementation Insight:

```
class LayerNormalisation(nn.Module):
    def __init__(self, embed_dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.alpha = nn.Parameter(torch.ones(embed_dim))
        self.bias = nn.Parameter(torch.zeros(embed_dim))

    def forward(self, x):
```

```
        mean = x.mean(dim=-1, keepdim=True)
        std = x.std(dim=-1, keepdim=True)
        return self.alpha * (x - mean) / (std + self.eps) + self.bias
```

Shape walkthrough:

- Input `x` has shape `(B, N, D)` → batch size, number of tokens, embedding dimension.
- LayerNorm computes mean and standard deviation along the last axis (`D`) independently for each token.
- Output retains the same shape `(B, N, D)`.

> ***LayerNorm standardizes values along the last axis (D), keeping output shape the same.*** *This ensures that gradients flow smoothly through the network — a small but crucial detail that prevents exploding or vanishing updates as we go deeper.*

2. Multi-Head Self-Attention

The **attention mechanism** is the core of the Vision Transformer. Unlike CNNs, which rely on local receptive fields, attention allows each patch token to "see" every other patch in the image. This is crucial for capturing **long-range dependencies**, such as recognizing that the tail of a cat belongs to the same object as its head, even if they are far apart spatially.

Conceptual Intuition

Think about a sentence in NLP:

- Each word(token) in a sentence has a relationship to other words.

- For example, in “The cat chased the mouse,” the word “cat” is strongly related to “chased” and “mouse.”

Similarly, in images:

- Each patch is like a “word” describing part of the image.
- Multi-head attention allows each patch to figure out **how much it should pay attention to every other patch** when forming its representation.

This is exactly what the **Vision Transformer paper (Dosovitskiy et al., 2020, Section 3.2)** leverages: treating image patches as sequential tokens, just like words, and applying self-attention to learn their relationships.

The detialed explanintion of attenion mechanism can be found in the **The Detailed Explanation of Self-Attention in Simple Words** blog written by me, but still to give over view :

The Detailed Explanation of Self-Attention in Simple Words

The attention mechanism is one of the most groundbreaking concepts in deep learning. It fundamentally changes how...

medium.com

Step-by-Step Mechanics

1. **Linear Projections: Queries, Keys, and Values (Q, K, V)** through learnable matrices:

For a sequence of patch embeddings `X` of shape `(B, N, D)`:

- `B` = batch size
- `N` = number of tokens (patches + CLS)
- `D` = embedding dimension

Each embedding is projected into three vectors via learnable matrices:

- **Query (Q):** What does this token want to find in other tokens?
- **Key (K):** How does this token present itself to others?
- **Value (V):** What information does this token carry to be shared?

Mathematically (as in Vaswani et al., 2017):

$Q = V \,.\, W\_q,$

$K = V \,.\, W_k,$

$V = V \,.\, W_v$

Here, W_q, W_k, and W_v are the learnable parameters that the model optimizes during training.

> In ViT, these projections are implemented using simple linear layers (`nn.Linear`), applied to each token.

- **Similarity scores:**
  Compute how much each token should attend to others using:

$$Attention_i(Q_i, K_i, V_i) = Softmax(\frac{Q_i K_I^T}{\sqrt{d_k}})V_i$$

Where, the scaling factor $\frac{1}{\sqrt{d_k}}$ stabilisez training, as explained in previous section

- **Scaling:**
  - Division by √dk keeps values numerically stable for the softmax.
- **Multiple heads:**
  Splitting into multiple heads allows the model to capture different types of relationships (e.g., texture, color, or position).
- **Concatenation + Projection:**
  Outputs from all heads are concatenated and projected back into the embedding dimension.

Shape walkthrough:

- `Q, K, V` → `(B, head, N, D_head)` after splitting heads
- Attention scores → `(B, head, N, N)` (each token attends to all others)
- Output per head → `(B, head, N, D_head)`

Multi-Head Attention: Why multiple heads?

1. One attention head can only focus on **one type of relationship**. For instance:

- Head 1 may learn foreground-background interactions

- Head 2 may focus on edges and textures
- Head 3 may capture object co-occurrences

By splitting the embedding dimension into multiple heads (`D_head = D / num_heads`), each head learns **complementary contextual relationships.**

After attention:

- Heads are concatenated back into `(B, N, D)`
- Linear projection (`W_O`) is applied to mix the information from all heads.

Dropout Regularization

- Dropout is applied on attention weights (`attention_dropout`) to prevent overfitting.
- Another dropout is applied on the final output (`proj_dropout`) for additional regularization.

Code Implementaiton :

```
class MultiHeadAttention(nn.Module):
    def __init__(self, embedding_dimension: int, head: int, dropout_rate: float
        super().__init__()
        assert embedding_dimension % head == 0, "Embedding dimension must be div
        self.w_q = nn.Linear(embedding_dimension, embedding_dimension)
        self.w_k = nn.Linear(embedding_dimension, embedding_dimension)
        self.w_v = nn.Linear(embedding_dimension, embedding_dimension)
        self.head = head
        self.d_k = embedding_dimension // head
        self.w_o = nn.Linear(embedding_dimension, embedding_dimension)

        self.attention_dropout = nn.Dropout(p=dropout_rate)
```

```
        self.proj_dropout = nn.Dropout(p=dropout_rate)

    @staticmethod
    def attention(q, k, v, dropout: nn.Dropout = None):
        d_k = q.shape[-1]
        scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)
        scores = scores.softmax(dim=-1)
        if dropout is not None:
            scores = dropout(scores)
        return torch.matmul(scores, v), scores

    def forward(self, q, k, v):
        batch_size, num_tokens, _ = q.shape

        # Linear projections
        query = self.w_q(q).view(batch_size, num_tokens, self.head, self.d_k).tr
        key = self.w_k(k).view(batch_size, num_tokens, self.head, self.d_k).tran
        value = self.w_v(v).view(batch_size, num_tokens, self.head, self.d_k).tr

        # Attention
        x, self.attention_score = MultiHeadAttention.attention(query, key, value

        # Concatenate heads
        x = x.transpose(1, 2).contiguous().view(batch_size, num_tokens, self.hea

        # Final linear projection + dropout
        return self.proj_dropout(self.w_o(x))
```

Shape Transformations:

Step	Shape	Explanation
Input embeddings	`(B, N, D)`	Batch size B, N tokens, D embedding dimension
After linear Q/K/V	`(B, N, D)`	Still same shape, now projected
Reshape for heads	`(B, head, N, D_head)`	D_head = D / head
Attention scores	`(B, head, N, N)`	Each token attends to all others
Contextualized output	`(B, head, N, D_head)`	Weighted sum of values per head
Concatenate heads	`(B, N, D)`	Reconstruct original embedding dimension
Final linear projection	`(B, N, D)`	Ready for residual connection + MLP

At this stage, the model:

- Has normalized token embeddings via **LayerNorm.**
- Built **contextualized representations** for each patch using **Multi-Head Attention.**

Next, these outputs will go through the **MLP/Feed-Forward block**, residual connections, and finally stack into the Transformer encoder to create deep, hierarchical representations.

> ***Quick Favor 🥲: If you found this article helpful, consider following me, clapping 👏, or sharing it with others who might benefit. Thank you! . Let's get back to the article***

Now that we have our multi-head self-attention (MHSA) mechanism working, the next logical step is to give each token the ability to transform its own representation — independently, without talking to others. This is where the **Feed-Forward Network (FFN)**, or **position-wise MLP**, comes in.

The Feed Forward Layer (Position-wise MLP)

After attention layers mix information across all patches, the FFN acts like a small, private processor for each token. It helps the model learn non-linear transformations and refine what each patch has learned from the rest of the image.

In the original ViT paper , the FFN is defined as:

$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1 \ldots L \tag{3}$$

The ViT paper (*Dosovitskiy et al., 2020*) replaces ReLU with **GELU** (Gaussian Error Linear Unit), a smoother and more stable activation:

$$GELU(x) = x\Phi(x)$$

where $\Phi(x)$ is the cumulative distribution function of a standard normal distribution. This subtle change improves training stability and helps the model learn softer activation boundaries — a small but meaningful shift from NLP to vision.

```
class FeedForwardLayer(nn.Module):
    def __init__(self, d_model: int, d_ff_scale: int = 2, dropout_rate: float =
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff_scale * d_model),
            nn.GELU(),
            nn.Dropout(dropout_rate),
            nn.Linear(d_ff_scale * d_model, d_model),
            nn.Dropout(dropout_rate)
        )

    def forward(self, x):
        return self.mlp(x)
```

What actually happens inside is fairly straightforward:

- Each patch embedding (shape **B × N × D**) passes through two linear layers with a GELU activation and dropout in between.

- The first layer expands the representation (from D → D_{ff}), allowing the model to explore a higher-dimensional feature space.
- The second layer projects it back down (D_ff → D).
- Every patch goes through the same transformation independently — hence the name *position-wise*.

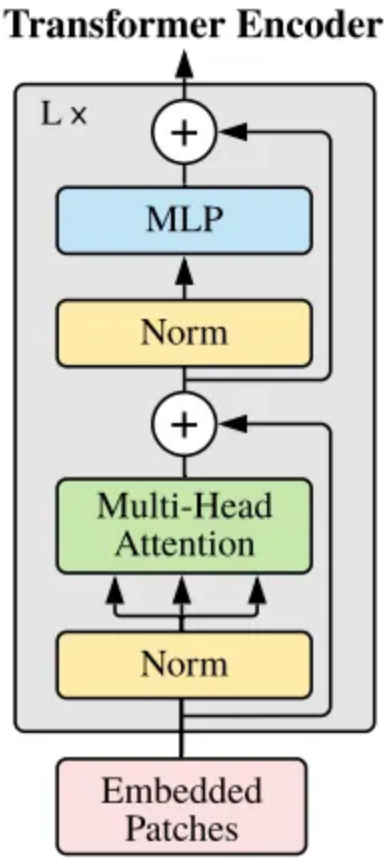
While the attention block helps patches interact and share context, the FFN lets each patch evolve internally, deepening its representation before the next encoder block processes it. It's a quiet but crucial step — the moment where attention's global reasoning and the MLP's local refinement meet.

Putting It Together: The Encoder Block

With all the core building blocks in place — layer normalization, multi-head self-attention, and the feed-forward network — we can now assemble the complete **Transformer Encoder Block**, the true engine of the Vision Transformer. Each encoder block follows the same elegant structure used in both the original Transformer and ViT:

$$x' = x + MHSA(LayerNorm(x))$$

$$y = x' + FFN(LayerNorm(x'))$$



a. Layer Normalization

Before applying attention or feed-forward layers, we normalize each token's embedding using **Layer Normalization**. This stabilizes training by keeping feature statistics consistent across layers.

b. Residual Connections

Residual (skip) connections, introduced in *ResNet* and adopted in *Transformers*, are crucial for gradient flow. Each sublayer's output is added back to its input, forming a shortcut path that helps prevent vanishing gradients and allows the model to learn identity mappings when necessary.

The flow inside each block is simple but powerful.

1. First, the input sequence of patch embeddings is normalized and passed into the multi-head self-attention module, where tokens interact globally —

each patch attending to all others.

2. The output of this attention layer is then added back to the original input through a residual connection.

3. Next, this updated sequence undergoes another layer normalization, followed by the feed-forward network that processes each token independently.

4. Finally, another residual connection merges the MLP's output with its input, preserving dimensional consistency and stabilizing learning through depth.

```
class EncoderBlock(nn.Module):
    def __init__(self, embedding_dimensions, heads, attention_dropout_rate, feed
        super().__init__()
        self.normalisation_stage1 = LayerNormalisation(embedding_dimensions)
        self.mhsa = MultiHeadAttention(embedding_dimensions, heads, attention_dr
        self.normalisation_stage2 = LayerNormalisation(embedding_dimensions)
        self.feed_forward_layer = FeedForwardLayer(embedding_dimensions, d_ff_sc

    def forward(self, x):
        residual1 = x
        x = self.normalisation_stage1(x)
        x = self.mhsa(x, x, x) + residual1
        residual2 = x
        return self.feed_forward_layer(self.normalisation_stage2(x)) + residual2
```

Notice how the shapes remain perfectly consistent throughout:

input (B, N, D) flows through normalization, attention, feed-forward, and residual steps — always returning to the same (B, N, D) format.

This shape invariance is essential because it allows multiple encoder blocks to be stacked seamlessly, forming deep hierarchical layers that progressively refine the image representation.

Stacking Multiple Encoder Blocks

The Vision Transformer doesn't use just one encoder block — it stacks many (typically 12 for ViT-Base). Each layer refines the representations further.In the code we implement these encoder blocks using the ModuleList in pytorch

```
class Encoder(nn.Module):
    def __init__(self, num_of_encoders, embeddings, dff_scale, heads, attention_
        super().__init__()
        self.encoder_stack = nn.ModuleList(
            EncoderBlock(
                embedding_dimensions=embeddings,
                heads=heads,
                dff_scale=dff_scale,
                attention_dropout_rate=attention_dropout_rate,
                feed_forward_dropout_rate=feed_forward_dropout_rate
            ) for _ in range(num_of_encoders)
        )

    def forward(self, x):
        for module in self.encoder_stack:
            x = module(x)
        return x
```

Each encoder layer passes its output as input to the next. As we go deeper:

- Lower layers capture **local patch relationships.**

- Middle layers capture **object-level features.**
- Higher layers capture **semantic understanding** across the entire image.

This hierarchical depth is what gives ViT its power — despite having **no convolutions**, it learns **rich multi-scale features**through global attention.

Final Vision Transformer Architecture

All components — patch creation, input embedding, encoder stack, and classification head — come together in the final `VisionTransformer` class:

```
class VisionTransformer(nn.Module):
    def __init__(self, in_channels, image_size, patch_size, number_of_encoder,
                 embeddings, d_ff_scale, heads, input_dropout_rate,
                 attention_dropout_rate, feed_forward_dropout_rate, number_of_cl
        super().__init__()
        self.input_layer = ViTInputLayer(in_channels, patch_size, image_size, em
        self.encoder_stack = Encoder(number_of_encoder, embeddings, d_ff_scale,
                                     attention_dropout_rate, feed_forward_dropou
        self.classification_head = nn.Sequential(
            nn.LayerNorm([embeddings]),
            nn.Linear(embeddings, number_of_classes)
        )

    def forward(self, x):
        return self.classification_head(self.encoder_stack(self.input_layer(x))[
```

At this point, our ViT isn't just a theoretical construct — it's a **complete, modular, and fully functional model** that mirrors the original *"An Image is Worth 16×16 Words"* paper by Dosovitskiy et al. (Google Research, 2020).

But to bring this architecture to life, we need one final piece — **data.**
Our freshly built ViT is like a brain waiting to learn, and now it needs

something to observe and understand.

The next section focuses on preparing that data — from **loading and transforming raw images** to **structuring them into patches** that our model can process.
To make things a bit fun (and nostalgic), we'll be using a **Tom and Jerry dataset** 🐭🐱 for training and testing.

Here's what's coming up next:

- **Dataset Creation:** How the Tom and Jerry dataset is organized, downloaded, and prepared for training.
- **Image Transformations:** Why resizing and normalization are crucial for patch-based models.
- **Data Augmentation Strategies:** Simple tricks to help the model generalize better.
- **DatasetLoader Class:** Our custom PyTorch Dataset that ties seamlessly into the ViT input pipeline.

Dataset Pipeline (`dataset.py`)

Well, we've built the model — all the building blocks are now in place. **Now it's time to put data into the machine.**

In the **original Vision Transformer paper**, the authors trained their model on **massive datasets** like **ImageNet-21k (14 million images)** and **JFT-300M (300 million images).** These large-scale datasets are what gave ViTs their remarkable ability to learn global visual representations.

However, **training on such enormous datasets requires high-end hardware** — multi-GPU clusters or TPU pods — far beyond what most personal computers can handle.

So, for this project, I've used a **smaller, open-source dataset** from Kaggle: the **Tom & Jerry image classification dataset** 🐭🐱. It's light enough to train on a single machine yet rich enough to demonstrate the ViT pipeline in action.

The dataset defines a clear 4-class problem — distinguishing between **Tom (the cat) , Jerry (the mouse) , Both Tom and Jerry Present and Neither Tom nor Jerry Present** — making it perfect for experimentation and learning.

> *Quick Favor 🙂: If you found this article helpful, consider following me, clapping 👏, or sharing it with others who might benefit. Thank you! . Let's get back to the article*

In the following section, I'll walk you through:

- How I **downloaded and organized** the dataset.
- The **custom PyTorch Dataset** class I wrote to load images efficiently.
- The **image transforms** used — resizing, normalization, and augmentation — and why each step matters.
- A few **practical tips** for working with **small or toy datasets**, to help you make the most of limited data while still getting meaningful results.

Downloading the basic organisation dataset

I downloaded the dataset with a small Jupyter helper (you can find the full code in the notebook **base_testing.ipynb** notebook in the first few starting cells). The snippet below shows the quick download and a script I used to split the raw dataset into `train/` and `test/` folders and also create a small `subset/` for fast local experiments:

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("balabaskar/tom-and-jerry-image-classification
print("Path to dataset files:", path)
```

> ***NOTE :*** *If you are not bale to see your downloaded dataset in the base directory then check it once in the '**.cache**' directory where usually these files from kaggle or hugging face gets downloaded, and can move it manulally to the base directory.*

After extracting, I inspected the folders and then randomly split images per class into train / test (90% train / 10% test). For faster iteration, I also create a small `subset` with 100 images per class (90 train / 10 test) so I can debug training quickly without waiting for full runs.

Key points in the splitting script:

- We loop through each class folder.
- Pick 90% files → move to `train/<class>`.
- Remaining 10% → `test/<class>`.
- Copy 100 images into `subset/` for very fast smoke tests.

This is the exact kind of setup that makes experimentation painless: a full dataset for real runs and a tiny subset for debugging.

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
```

```
import glob
from PIL import Image
from torchvision.transforms import v2
import os

import random
import shutil
from pathlib import Path

import numpy as np
class TomAndJerryDataset(Dataset):
    def __init__(self,dataset_path,transform=None, target_transform=None):
        super().__init__()
        self.transformation = transform

        self.target_transform = target_transform
        self.classes = os.listdir(dataset_path)
        self.class_to_index = {self.classes[ix]:ix for ix in range(len(self.clas
        self.index_to_class = {ix:self.classes[ix] for ix in range(len(self.clas
        self.images = glob.glob(dataset_path+'/*/*.jpg')
        self.dataset_path = dataset_path
        self.to_tensor = v2.ToTensor()
        # print(self.images[0])


    def __len__(self,):
        return len(self.images)

    def __getitem__(self, index):
        image = self.images[index]
        image = Image.open(image).convert('RGB')

        # label = str(Path(self.images[index]).parent).split('/')[-1]
        label = Path(self.images[index]).parent.name
        # print(label)
        # print(np.array(image).shape)
        # plt.imshow(image)

        if self.transformation is not None:
            image = self.transformation(image)
        else:
            image = self.to_tensor(image)


        label = self.class_to_index[label]
        if self.target_transform:
            label = self.target_transform(label)
        # plt.imshow(image)

        return image,label
```

```
# transform = v2.Compose([
#     v2.Resize((224,224)),
#     v2.RandomVerticalFlip(p=0.34),
#     v2.ToTensor()

# ])

train_transform = v2.Compose([

    v2.RandomResizedCrop(size=(224, 224), scale=(0.8, 1.0)),
    v2.TrivialAugmentWide(),
    v2.RandomHorizontalFlip(p=0.5),
    v2.ToTensor(),
    v2.Normalize(mean=[0.485, 0.456, 0.406],
                 std=[0.229, 0.224, 0.225])
])

val_transform = v2.Compose([
    v2.Resize((224, 224)),
    v2.ToTensor(),
    v2.Normalize(mean=[0.485, 0.456, 0.406],
                 std=[0.229, 0.224, 0.225])
])
```

The Dataset class

The I creatred a custom `TomAndJerryDataset` that wraps raw image files and returns `(image_tensor, label_index)` pairs. This keeps the model code clean and the data loading flexible.

Although, in this code i have created a custom dataset class for Tom and jery but it still can be easily used for any other dataset just by providing the folder.

and also we have the custom transfomration defined for each input image as per the original paper,

Transforms and preprocessing (what I actually used)

Vision Transformers expect **fixed-size inputs** (we used **224×224**) and benefit greatly from consistent **normalization**. I built the preprocessing pipeline using **PyTorch's** `torchvision.transforms.v2` , which offers a cleaner and faster interface for data augmentation.

Why these choices?

- `RandomResizedCrop` replaces the older `CenterCrop+Resize` flow and encourages robustness to scale and composition.
- `TrivialAugmentWide` is a cheap, effective automatic augmentation policy — good baseline augmentation without hand-tuning. It's similar in spirit to policy-based augmentations used in robust ViT training (see DeiT for augmentation discussions).
- `RandomHorizontalFlip` handles left-right invariance (Tom on the left vs right).
- `Normalize` uses ImageNet mean/std — common practice when models or pretraining rely on ImageNet statistics. It stabilizes the optimization and usually improves transferability.
- **Validation** uses simple `Resize + Normalize` to keep evaluation stable.

Augmentations applied on-the-fly (as done here) do not change the number of samples per epoch, but they **increase the effective diversity** the model sees during training. That's usually the goal: more variety per epoch without inflating dataset storage.

If you want to literally increase dataset size (e.g., for offline training constrained by random seeds), you can augment offline and save to disk — but that costs space and often isn't necessary.

Now our dataset class is created successfully and now we can load our entire datasedt for training jsut few line of codes as below for training purposes as pewr used in the original train.py file:

```
train_dataset = dataset.TomAndJerryDataset(dataset_path=train_path,
                                           transform=train_transform)
val_dataset = dataset.TomAndJerryDataset(dataset_path=val_path,
                                         transform=val_transform)
train_data_loader = DataLoader(train_dataset,
                               batch_size = cfg.batch_size,
                               shuffle = True)
val_data_loader = DataLoader(val_dataset,
                             batch_size = cfg.batch_size,
                             shuffle = False)
```

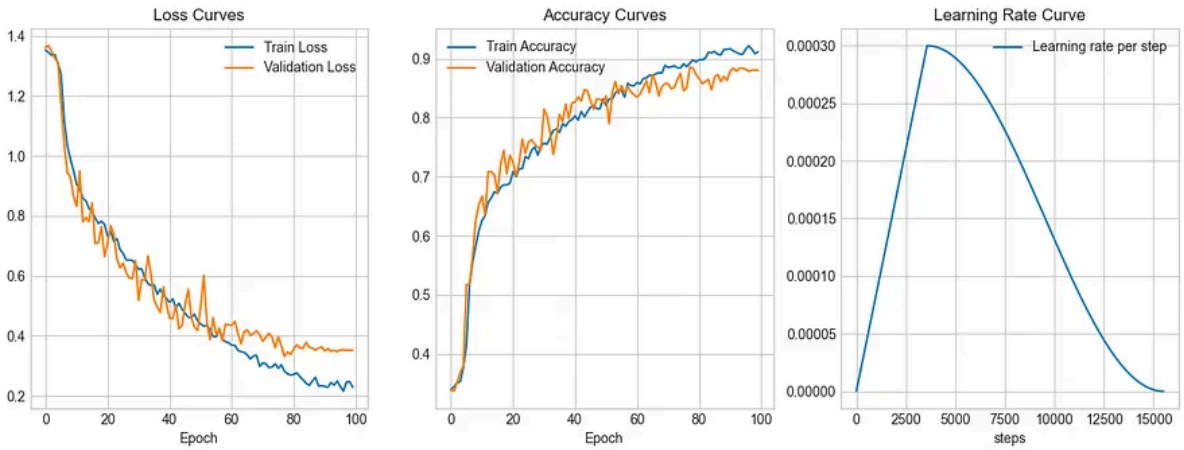
Training Logic (`train.py`)

Now that we have completed our `model.py` to define the skeleton of our Vision Transformer and built the `dataset.py` to handle custom dataset creation, we finally have all the essential components in place to move toward the most exciting part — **training our Vision Transformer.**

In the `train.py` file, I'll bring everything together —

1. loading the dataset,
2. constructing the ViT model,
3. and setting up the full training loop.

But before diving into the loop itself, there's one subtle yet important step I always take at the very beginning — **configuring my environment and device setup.**



Why Hydra?

As experiments grow more complex — tweaking learning rates, trying different optimizers, or testing new model depths — manually updating these in code quickly becomes messy.

That's why I use **Hydra**, a powerful configuration management library.
It lets you define all experiment settings (paths, model parameters, training hyperparameters, etc.) neatly in **YAML** files. You can override any setting directly from the command line — for example:

```
python train.py optimizer.lr=3e-4 model.heads=8
```

> *Quick Favor 🥲: If you found this article helpful, consider following me, clapping 👏, or sharing it with others who might benefit. Thank you! . Let's get*

> *back to the article*

Now, coming to the device setup — I wanted my training code to be device-agnostic. This means whether you're training on a powerful NVIDIA GPU, Apple's MPS backend, or even just your CPU, the same code should run without a single line change. So, I added a small but essential snippet right at the top of `train.py`:

```
if torch.cuda.is_available():
    device = torch.device('cuda')
    print("✅ Setting Device as CUDA...")
elif torch.backends.mps.is_available() and torch.backends.mps.is_built():
    print("🥲  Device is set to MPS...")
    device = torch.device('mps')
else:
    print("No accelerator available 🥺 ...using CPU for this task...")
log = logging.getLogger(__name__)
```

Building the Dataset Creation Function

Once the base setup for device configuration was done, my next focus in the `train.py` file was on constructing the **dataset creation pipeline**.

This is where our `dataset.py` file comes into play — the one where we designed the `TomAndJerryDataset` class to handle image loading and transformations.

```
def dataset_creation(cfg: DictConfig):

    train_path = cfg.train_dataset_path
    val_path = cfg.test_dataset_path
    val_transform = dataset.val_transform
```

```
    train_transform = dataset.train_transform
    train_dataset = dataset.TomAndJerryDataset(dataset_path=train_path,
                                               transform=train_transform)
    val_dataset = dataset.TomAndJerryDataset(dataset_path=val_path,
                                             transform=val_transform)
    train_data_loader = DataLoader(train_dataset,
                                   batch_size = cfg.batch_size,
                                   shuffle = True)
    val_data_loader = DataLoader(val_dataset,
                                 batch_size = cfg.batch_size,
                                 shuffle = False)
    return train_dataset,train_data_loader,val_dataset,val_data_loader
```

I wrapped the entire dataset setup into a clean, Hydra-compatible function called `dataset_creation(cfg: DictConfig)`.
The reason I prefer writing this as a separate function rather than just directly inside the main training loop is simple — **cleaner modularity**. I want each logical step in my training pipeline to be encapsulated and reusable.

Also, The dataset paths (`train_dataset_path`, `test_dataset_path`) are pulled directly from the Hydra config. This means switching datasets (say from Tom & Jerry to CIFAR-10) is as easy as editing a YAML file — no code changes needed.

Once executed, this function returns both datasets and their loaders — neatly packed and ready to be consumed by the training loop.
What I like most about this approach is how self-contained it feels. No matter what dataset I plug in — CIFAR, custom cartoons, or any new image collection — the same function works effortlessly by just adjusting the config file.

Saving and Visualizing Training Progress

After the dataset setup, the next function I added was something I consider **non-negotiable in any serious deep learning project** — a visualization function to track the model's progress across epochs.

```
def saving_training_plots(history_df,lr, output_dir):
    """
    Function to store train vs. val loss and train vs. val accuracy.

    Input : history df with columns
                'train loss','train acc','val loss','val acc','learning rate'
    Output : NONe

    """
    plt.style.use("seaborn-v0_8-whitegrid")
    fig, ax = plt.subplots(1,3, figsize=(15,5))
    ax[0].plot(history_df["train loss"], label="Train Loss")
    ax[0].plot(history_df["val loss"], label="Validation Loss")
    ax[0].set_title("Loss Curves")
    ax[0].set_xlabel("Epoch")
    ax[0].legend()

    ax[1].plot(history_df["train acc"], label="Train Accuracy")
    ax[1].plot(history_df["val acc"], label="Validation Accuracy")
    ax[1].set_title("Accuracy Curves")
    ax[1].set_xlabel("Epoch")
    ax[1].legend()

    ax[2].plot(lr, label="Learning rate per step")
    ax[2].set_title("Learning Rate Curve")
    ax[2].set_xlabel("steps")
    ax[2].legend()

    plot_path = os.path.join(output_dir, "training_curves.png")
    fig.savefig(plot_path)
    log.info(f"Saved training plot to {plot_path}")
```

This function creates three side-by-side plots as shown in the image above:

1. **Training vs Validation Loss** — to see how the model's error evolves and whether it's overfitting.
2. **Training vs Validation Accuracy** — to gauge improvement in predictive capability.
3. **Learning Rate per Step** — especially important when using **learning rate schedulers** like cosine decay or warmup strategies (which we'll implement shortly).

The output figure is automatically saved inside the experiment's output directory — again, easily configurable through Hydra. This small automation makes it easy to revisit past experiments and visually compare performance trends without rerunning anything.

The Training Loop: Bringing Everything Together

With the model implemented, the dataset prepared, and my helpers in place, it's finally time to train. This is the moment when all the small design decisions start to matter: how you schedule the learning rate, whether you clip gradients, how you checkpoint, and how you log metrics.

Below I'll walk through the `main_loop` (Hydra-decorated) that I use for actual training. I'll explain each block, why it exists, and what I watch for while the model is running.

Hydra and run setup

The entry point to the training script is decorated with Hydra:

```
@hydra.main(config_path='configs', config_name='config', version_base=None)
def main_loop(cfg: DictConfig):
```

```
...
```

Hydra does a few invaluable things for me:

- It centralizes **all** hyperparameters and paths in YAML files, so I don't have magic numbers scattered across code.
- It creates an **output run directory** for every run (HydraConfig.get().runtime.output_dir/ multi-run), which I use to store checkpoints, plots, and the exact config YAML used for that experiment.
- It makes experimenting trivial: to try a new LR or model size I just pass `optimizer.lr=0.0005` on the command line — no code edits.

Right after Hydra hands me the `cfg`, I print and log the active config and the Hydra output directory:

```
output_dir = HydraConfig.get().runtime.output_dir
log.info(f"All artifacts will be saved to: {output_dir}")
log.info(f"\n{OmegaConf.to_yaml(cfg)}")
```

Dataset verification & saving label mapping

Next I build datasets via the helper we discussed earlier:

```
train_dataset, train_data_loader, val_dataset, val_data_loader = dataset_creatio
num_of_classes_in_dataset = len(train_dataset.classes)
```

I also check that the dataset's number of classes matches what the config expects. If they don't match, that usually signals a path mistake (wrong train folder) or mislabeled folders — better to fail fast:

```
assert num_of_classes_in_dataset == cfg.model.number_of_classes, \
    f"Mismatch: config expects {cfg.model.number_of_classes} classes, but datase
```

If the assertion fails, I log a clear error and exit. If it passes, I save an index→label JSON mapping into the run folder:

```
index_to_class = train_dataset.index_to_class
mapping_save_path = os.path.join(output_dir,"class_mapping.json")
with open(mapping_save_path,'w+') as f:
    json.dump(index_to_class, f, indent=4)
```

This small JSON is gold when you do inference later — the model outputs indices, and this file maps them back to human-readable class names.

Creating the model

Then we further create the model, using the hyper-parameters defined in hydra configuration file, along with creating the model we will be passing the dummy data point to pass it to the model , in order to visualise the model being created using torchsummary module.

```
    log.info("Model Creation Begin")
```

```
    vit_model = model.VisionTransformer(in_channels=cfg.model.in_channels,
                                        image_size=cfg.model.image_size,
                                        patch_size=cfg.model.patch_size,
                                        number_of_encoder=cfg.model.number_of_en
                                        embeddings = cfg.model.embedding_dims,
                                        d_ff_scale=cfg.model.d_ff_scale_factor,
                                        heads = cfg.model.heads,
                                        input_dropout_rate=cfg.model.input_dropo
                                        attention_dropout_rate=cfg.model.attenti
                                        feed_forward_dropout_rate=cfg.model.feed
                                        number_of_classes=cfg.model.number_of_cl
    test_input = torch.randn((cfg.train.batch_size,
                              cfg.model.in_channels,
                              cfg.model.image_size,
                              cfg.model.image_size)).to(device)
    log.info("Model Creation Completed.")
    # print(summary(vit_model,input_data= test_input))
    log.info("--- Model Summary ---")
    log.info(summary(vit_model, input_data=test_input))
    log.info("---------------------")
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
EncoderBlock                             [32, 197, 768]            --
├─LayerNormalisation: 1-1                [32, 197, 768]            1,536
├─MultiHeadAttention: 1-2                [32, 197, 768]            --
│    └─Linear: 2-1                       [32, 197, 768]            590,592
│    └─Linear: 2-2                       [32, 197, 768]            590,592
│    └─Linear: 2-3                       [32, 197, 768]            590,592
│    └─Linear: 2-4                       [32, 197, 768]            590,592
├─LayerNormalisation: 1-3                [32, 197, 768]            1,536
├─FeedForwardLayer: 1-4                  [32, 197, 768]            --
│    └─Sequential: 2-5                   [32, 197, 768]            --
│    │    └─Linear: 3-1                  [32, 197, 1536]           1,181,184
│    │    └─GELU: 3-2                    [32, 197, 1536]           --
│    │    └─Dropout: 3-3                 [32, 197, 1536]           --
│    │    └─Linear: 3-4                  [32, 197, 768]            1,180,416
│    │    └─Dropout: 3-5                 [32, 197, 768]            --
==========================================================================================
Total params: 4,727,040
Trainable params: 4,727,040
Non-trainable params: 0
Total mult-adds (M): 151.22
==========================================================================================
Input size (MB): 19.37
Forward/backward pass size (MB): 348.59
Params size (MB): 18.91
Estimated Total Size (MB): 386.86
==========================================================================================
```

Model Summary using torchinfo

Loss, Optimizer, and Learning Rate Schedule

Once the model and dataset are ready, the next important piece is deciding how the model would actually **learn** — that means defining the **loss function**, the **optimizer**, and the **learning rate schedule.**

For this setup, I went with **CrossEntropyLoss** as per the original paper . It's a straightforward and widely used choice for classification tasks, especially when the final layer outputs class logits. Since my Vision Transformer's output head produces class probabilities for each image, CrossEntropyLoss made perfect sense.

Now, for the optimizer, I didn't go with the default Adam — I chose **AdamW** instead. This isn't random; there's a reason behind it. The original **Vision Transformer paper by Dosovitskiy et al. (2020)** emphasized the importance of **weight decay** when training ViTs. The "W" in AdamW literally refers to that — it decouples weight decay from the gradient update, leading to better regularization and preventing overfitting, especially in large models like ViTs.

Here's the part from my code inside the training loop:

```
total_training_steps = cfg.train.epochs * len(train_data_loader)
num_warmup_steps = int(0.23 * total_training_steps)
```

```
optimizer = optim.AdamW(
    vit_model.parameters(),
    lr=cfg.optimizer.lr,
    betas=(cfg.optimizer.beta1, cfg.optimizer.beta2),
    weight_decay=cfg.optimizer.weight_decay
)
criterion = nn.CrossEntropyLoss()
```

I calculate `total_training_steps` simply as the number of epochs multiplied by the number of batches per epoch. Then I choose `num_warmup_steps` as roughly 23% of that total. This fraction isn't set in stone — it's tunable, but I found this value worked quite well in stabilizing the training phase.

Now, the **learning rate schedule** is where things get interesting. Instead of sticking to a fixed learning rate or a step decay, I followed the strategy proposed in the **original Transformer paper by Vaswani et al. (2017)** and later adapted by the ViT paper.

The idea is simple but powerful:

- Start with a **linear warmup** phase, where the learning rate gradually increases from 0 up to the base learning rate over the first `num_warmup_steps`.
- Then, after the warmup phase, **cosine decay** kicks in — smoothly decreasing the learning rate from the peak back down to near zero as training progresses.

This combination — `warmup + cosine` — has become almost a *standard practice* for transformer-based architectures. The warmup helps avoid the common issue where training becomes unstable early on because the optimizer takes too large steps when weights are still uncalibrated. The cosine decay, on the other hand, helps the model converge gracefully, avoiding abrupt drops that could hurt convergence stability.

I implemented this using PyTorch's `LambdaLR`, where the `lr_lambda` function computes the current learning rate multiplier based on the global step count. I call `scheduler.step()` at every optimizer step (i.e., after every batch) so that the learning rate updates smoothly across steps, rather than jumping only once per epoch.

In practice, this schedule gave me noticeably smoother training curves and more stable convergence — which is consistent with what both the Transformer and ViT papers observed. It's one of those techniques that quietly does a lot of heavy lifting in keeping training stable, especially for models that rely heavily on self-attention and layer normalization.

Accuracy helper & gradient clipping

A tiny helper computes batch accuracy:

```
def accuracy_fn(y_pred, y_true):
    preds = torch.argmax(y_pred, dim=1)
    correct = (preds == y_true).sum().item()
    return correct / y_true.size(0)
```

I also clip gradients after `loss.backward()` and before `optimizer.step()`:

```
torch.nn.utils.clip_grad_norm_(vit_model.parameters(), max_norm=1.0)
```

Transformers (especially large ones) sometimes produce large gradient norms. Clipping keeps gradients from exploding and often improves stability — I use `max_norm=1.0` as a conservative default.

The training + validation loop (batch by batch, epoch by epoch)

Here's the core loop in prose form — the code maps directly to this narrative:

For each epoch:

1. **Set the model to training mode** using `model.train()`.
2. **Initialize tracking variables** for training loss and accuracy.
3. **Iterate over batches** in the training DataLoader:

- Reset gradients with `optimizer.zero_grad()`.
- Transfer the input images and labels to the target device (e.g., GPU).
- Perform a **forward pass**:

  `pred_logits = vit_model(images)`

- Compute the **loss** using the chosen criterion:

  ```
  loss = criterion(pred_logits, labels)
  ```

- Perform **backpropagation:**

  ```
  loss.backward()
  ```

- Optionally apply **gradient clipping** using `clip_grad_norm_`.
- **Update model parameters:**

  ```
  optimizer.step()
  ```

- **Update the learning rate** (per batch) via `scheduler.step()`.
- Accumulate **loss and batch-level accuracy** for tracking.

4. After processing all training batches, compute the **average training loss and accuracy** for the epoch and store them in tracking lists.

5. Switch to evaluation mode with `model.eval()` and disable gradient calculations using `torch.inference_mode()`:

- Iterate over the validation DataLoader to compute **validation loss and accuracy.**
- Accumulate metrics across batches, average them, and append to the history.

6. Log the epoch summary with a concise, informative line displaying the key metrics.

```
log.info(f"Epoch {epoch+1} | train_loss: {epoch_avg_loss:.4f} | train_accuracy:
```

A couple of practical notes here:

- I record the learning rate at each step with `scheduler.get_last_lr()`. If you use a single parameter group, `get_last_lr()[0]` is fine. If you had multiple param groups, you might want to record each one separately.
- Averaging accuracy: I sum per-batch accuracies and divide by the number of batches. That works well when all batches are the same size — if your last batch is smaller it slightly biases the average, but for typical experiments it's negligible. If you want exact sample-level averages, accumulate `correct` and `total` counts and compute `correct/total` at the end.

Checkpointing best model

I save the model whenever validation accuracy improves:

```
if val_epoch_avg_accuracy > best_val_acc:
    best_val_acc = val_epoch_avg_accuracy
    model_path = os.path.join(output_dir, "best_model.pt")
    torch.save(vit_model.state_dict(), model_path)
```

Saving `state_dict()` is compact and portable. One improvement you may want is to save a full checkpoint including optimizer and scheduler states, so you can resume training precisely:

```
torch.save({
    "model_state": vit_model.state_dict(),
    "optimizer_state": optimizer.state_dict(),
    "scheduler_state": scheduler.state_dict(),
```

```
        "epoch": epoch,
    }, checkpoint_path)
```

But for many classification experiments saving the best model weights is sufficient for inference.

Finishing up: CSVs, plots, and artifacts

Once training completes, I bundle history into a DataFrame and save it:

```
data = list(zip(train_loss, train_acc, val_loss, val_acc))
df = pd.DataFrame(data, columns=['train loss','train acc','val loss','val acc'])
csv_path = os.path.join(output_dir, "training_history.csv")
df.to_csv(csv_path, index_label="epoch")
```

I also save the step-wise learning rates to a separate CSV and generate the training curves plot using the `saving_training_plots()` helper described earlier. That plot gives a quick visual cue whether things went well: smooth decay in loss, stable gap between train/val, LR shape matches expected `warmup + cosine`, etc.

Finally I log a completion message including the run output directory so I can quickly find artifacts.

> *Quick Favor 🙂: If you found this article helpful, consider following me, clapping 👏, or sharing it with others who might benefit. Thank you! . Let's get back to the article*

Debugging Tips & Best Practices

Even with careful setup, training can fail to learn. Here's my checklist when the metrics look wrong:

- **Can the model overfit a tiny subset?** If it can't overfit 100 images, something's wrong in the code — data pipeline, loss, or labels.
- **Is the learning rate appropriate?** Too small → no learning. Too large → diverging loss. Watch the LR plot.
- **Is weight decay too strong?** For small datasets, high weight decay can kill learning.
- **Are labels correct?** I always peek at `index_to_class` and a few image→label samples.
- **Is the model on the right device?** Mixing CPU tensors with GPU model silently fails — ensure `image.to(device)` and `label.to(device)` are present.
- **Gradients zero / explode?** If loss is NaN or huge, try gradient clipping or reduce LR.
- **Check augmentations:** Overly aggressive augmentations may destroy signals for small datasets.

Inference Setup (`inference.py`)

After the training loop was complete and my Vision Transformer was producing good validation accuracy, the next natural step was to test how

well it performs in the real world — on unseen images. That's where the **inference.py** script comes in.

This file is designed not just to load a model and make a prediction, but to do so in a **structured, repeatable, and configuration-aware** way. The whole idea was to make inference as smooth as possible — no more hardcoding model paths, class names, or hyperparameters. Everything gets reconstructed automatically using **Hydra** and the experiment logs.

Setting up the device

Just like in the training script, the first thing I do is set the appropriate compute device. Whether I'm running on a high-end NVIDIA GPU, Apple's MPS backend, or just a CPU, I want the code to be device-agnostic.

```
if torch.cuda.is_available():
    device = torch.device('cuda')
    print("✅ Setting Device as CUDA...")
elif torch.backends.mps.is_available() and torch.backends.mps.is_built():
    print("🥲  Device is set to MPS...")
    device = torch.device('mps')
else:
    print("No accelerator available 🥺 ...using CPU for this task...")
```

This small check ensures that wherever the model is deployed, it automatically picks the best hardware available. When I was experimenting with Apple Silicon, this MPS check came in handy — it allowed me to test the model locally before scaling up to CUDA on a proper GPU machine.

Finding the right experiment

Once the environment is ready, the next question becomes:
"Which trained model should I use for inference?"

When you've run multiple experiments — especially with Hydra's `multirun` feature — it can quickly get messy to track which model gave the best performance. So, I wrote a small utility called `find_experiment()`.

This function scans through my **outputs/** and **multirun/** directories (Hydra automatically organizes them that way). For each experiment folder, it looks for three essential files:

- `best_model.pt` — the saved model weights
- `training_history.csv` — for tracking the accuracy and loss over epochs
- `.hydra/config.yaml` — Hydra's saved configuration snapshot

Only when all three exist, the folder is treated as a valid experiment. Then, I read the validation accuracy from the `training_history.csv` file and keep track of it.

This way, I don't just load *a* model — I can pick the **best** model based on validation performance.

Here's how it looks in code:

```
def find_experiment(search_dirs=["outputs", "multirun"]):
    ...
    for exp_dir in exp_dirs:
        if (os.path.exists(os.path.join(exp_dir, "best_model.pt")) and
            os.path.exists(os.path.join(exp_dir, "training_history.csv")) and
            os.path.exists(os.path.join(exp_dir, ".hydra/config.yaml"))):

            history = pd.read_csv(os.path.join(exp_dir, "training_history.csv"))
            best_acc = history["val acc"].max()

            all_valid_exp.append({
```

```
            "path": exp_dir,
            "best_val_acc": best_acc
        })
```

In practice, this small automation saved me from a ton of confusion. Instead of manually copying model paths or guessing which experiment worked best, I can now just run the script — and it neatly lists all my trained models, sorted by performance.

Reconstructing the model with Hydra

Once I pick the experiment, I load its saved Hydra configuration file:

```
cfg_path = os.path.join(selected_exp['path'], '.hydra', 'config.yaml')
cfg = OmegaConf.load(cfg_path)
```

This single line is one of my favorites.
Because Hydra saves **every parameter** — from model hyperparameters to optimizer settings — I can now perfectly reconstruct the model exactly as it was during training.

That means the same `patch_size`, `embedding_dims`, number of encoder layers, dropout rates — everything comes straight from the saved config.
This is one of Hydra's biggest strengths: **reproducibility**.
No matter how long after training you come back, you can always rebuild the exact same model setup.

Here's how I reinitialize the model:

```
model = VisionTransformer(
    in_channels=cfg.model.in_channels,
    image_size=cfg.model.image_size,
    patch_size=cfg.model.patch_size,
    number_of_encoder=cfg.model.number_of_encoder,
    embeddings=cfg.model.embedding_dims,
    d_ff_scale=cfg.model.d_ff_scale_factor,
    heads=cfg.model.heads,
    input_dropout_rate=cfg.model.input_dropout_rate,
    attention_dropout_rate=cfg.model.attention_dropout_rate,
    feed_forward_dropout_rate=cfg.model.feed_forward_dropout_rate,
    number_of_classes=cfg.model.number_of_classes
).to(device)
```

After reconstructing the model, I load the weights using `torch.load(model_path, map_location=device)` and switch to evaluation mode using `model.eval()`.
I also like to print a quick model summary using `torchinfo.summary()` — it's a great sanity check to confirm that the loaded model's architecture matches what I trained earlier.

Performing inference on an image

Once the model is loaded, I can pass in any image for classification.
The script asks for an image path, loads it with **PIL**, applies the **same validation transforms** (`val_transform`) that were used during training, and then sends it to the model for prediction.

```
image = Image.open(image_path).convert("RGB")
transformed_image = val_transform(image).unsqueeze(0).to(device)
```

```
with torch.inference_mode():
    logits = model(transformed_image)
```

```
probs = torch.softmax(logits, dim=1)
pred_label_idx = torch.argmax(probs, dim=1).item()
```

Using `torch.inference_mode()` instead of `torch.no_grad()` gives a small performance boost, especially on CUDA — it tells PyTorch we're only doing forward passes, no gradient tracking required.

Finally, I read the **class mapping** JSON file (`class_mapping.json`) that was saved during dataset preparation. That helps map the numeric index back to the actual class name, like *"Jerry"* or *"Tom" or "neither_tom_nor_jerry" or "bnoth_tom_and_jerry"*.

The final printout shows both the predicted label and the model's confidence:

```
Predicted Class Label: Jerry
Predicted Class Index: 1
Confidence: 98.23%
```

Why I love this setup

This inference pipeline turned out to be incredibly practical for experimentation.
Because of **Hydra**, I don't need to remember which hyperparameters I used — I just pick the experiment folder, and everything else is reconstructed automatically.
Because of **PyTorch's device management**, it runs seamlessly across CPU, GPU, or even Apple Silicon.
And because of **pandas** and **json**, I can transparently see which experiment performed best and what each label means.

It's clean, reproducible, and extensible — if I ever train new models or change architectures, I don't need to rewrite anything. The same inference pipeline will still work, which was exactly the kind of robust design I wanted for this project.

What's Next — Future Plans 🚀

As much as I've loved reproducing this Vision Transformer pipeline from scratch — from recreating original paper's ViT model and dataset setup to model training to inference — I'm far from done.
This project has become a playground for experimenting with modern deep learning workflows, and there are several exciting directions I'm planning to take next.

1. TensorBoard Integration

While I've already been logging training details into CSVs and visualizing them through Matplotlib, I plan to integrate **TensorBoard** for richer, real-time insights.
This would allow me to track loss curves, learning rates, attention visualizations, and more — all while the model is still training.
Having TensorBoard support means I can catch training anomalies early, monitor overfitting visually, and even compare multiple runs interactively without manually saving plots.

2. Distributed Data Parallel (DDP) Training

Currently, my training pipeline runs on a single GPU, which is fine for smaller datasets or quick experiments.
But for larger datasets or deeper transformer architectures, scaling up is crucial.

I'm planning to integrate **PyTorch Distributed Data Parallel (DDP)** next — which allows the model to train across multiple GPUs (and even across multiple machines) simultaneously.

DDP will help me:

- Greatly **reduce training time**
- **Increase batch size** without memory overflow
- Improve model convergence through better gradient averaging

This will bring the project much closer to how industrial-scale Vision Transformers are actually trained — especially in setups like Google's TPU pods or NVIDIA DGX clusters.

3. Web Frontend — it is Completed ✅

I've already taken a big step forward on this front.
To make this entire project more interactive and user-friendly, I built a **Streamlit-based frontend** that lets anyone — even non-programmers — use the model directly from their browser.

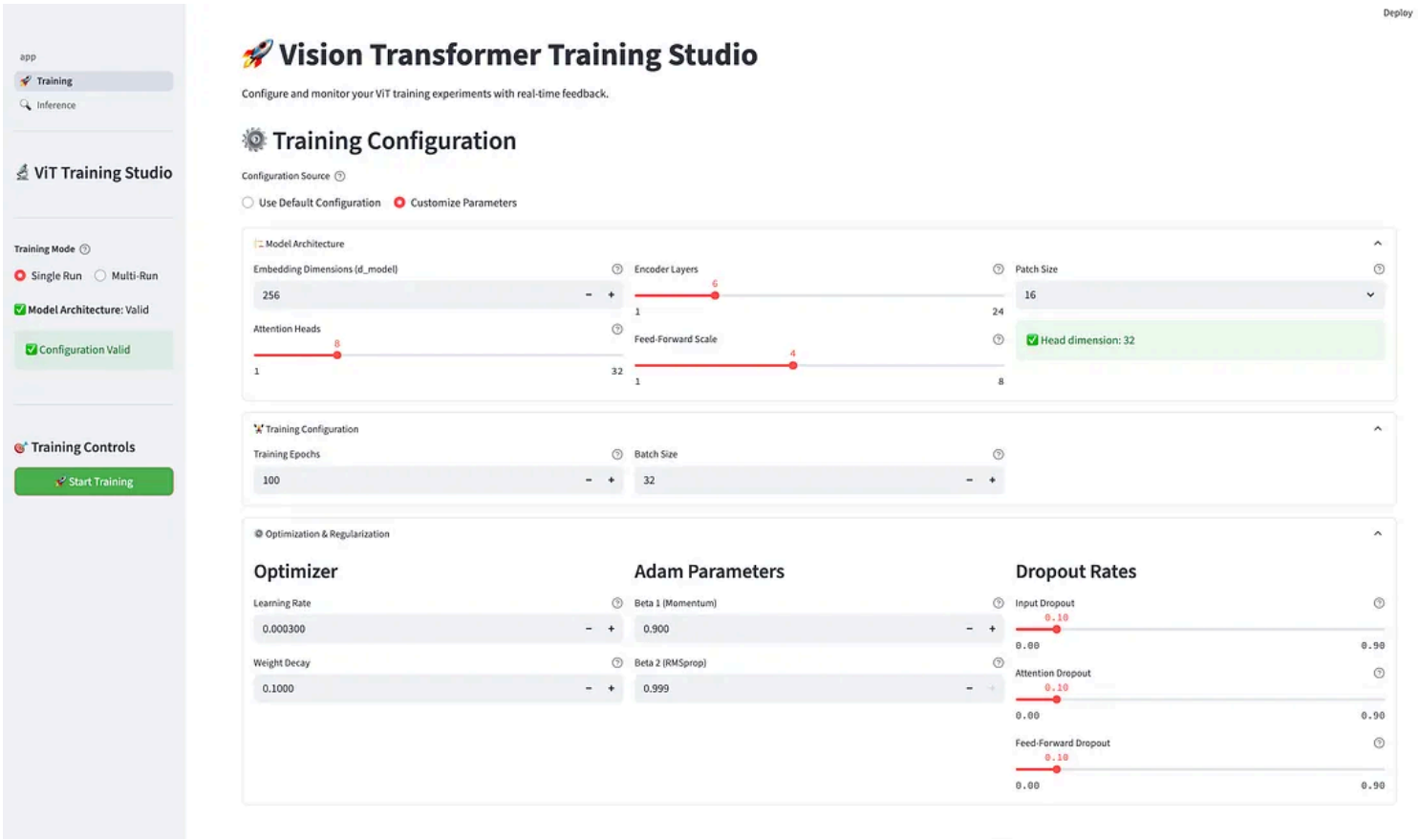
Instead of running commands or editing YAML files, users can now just **click, upload, and run** experiments in a fully visual interface.

Training Studio

The **Training Studio** allows users to:

- Select hyperparameters interactively
- Launch single or multi-run Hydra experiments

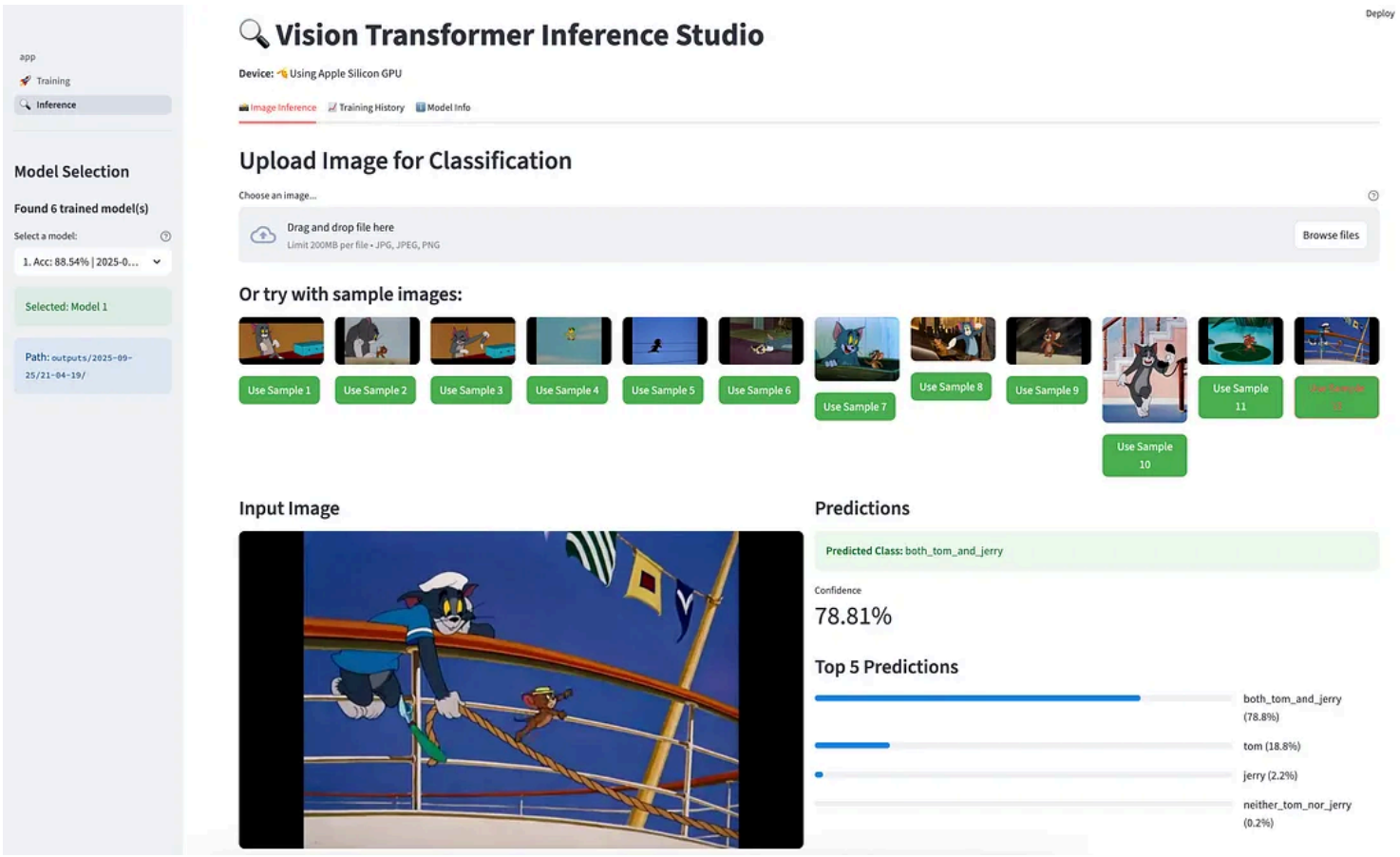
- Watch training metrics update live
- Start or stop training without touching the terminal



Inference Studio

The **Inference Studio** is equally powerful — it automatically detects all trained models in `outputs/` and `multirun/`, lets you browse through training histories, and instantly run inference on any image.

You can simply drag and drop an image, see predictions along with their confidence scores, and inspect the model configuration and architecture that produced those results.



How to Run

Launching the interface is as simple as:

```
streamlit run app.py
```

This frontend has already transformed the project into something more than just code — it's now an **interactive deep learning workspace**, perfect for both learning and experimentation.

4. Cloud Deployment and MLOps Pipeline

The next step is to take this setup to the cloud.
I plan to **containerize** the entire pipeline using **Docker**, making it portable and reproducible across different environments.

Once containerized, the training and inference workflows can be deployed on platforms like **AWS**, **Azure**, or **GCP**, with automated pipelines for:

- Continuous Training (CT)
- Continuous Evaluation (CE)
- Model versioning and artifact tracking
- Auto-deployment of the best-performing model

In essence, this would evolve the project from a research setup into a **production-grade Vision Transformer system** — capable of scaling, updating, and serving predictions reliably in real-world applications.

> *Quick Favor 🙂: If you found this article helpful, consider following me, clapping 👏, or sharing it with others who might benefit. Thank you! . Let's get back to the article*

Final Thoughts & Conclusion 💭

Looking back, this Vision Transformer project turned out to be far more than just a code implementation — it became a true exploration for me on how to translate research ideas into working, end-to-end systems.

When I started writing the `model.py`, I remember feeling the weight of the original **"Attention Is All You Need"** paper — trying to adapt that powerful concept from NLP to vision felt both exciting and intimidating.
Understanding how patches could play the same role as tokens did in text was a fascinating insight in itself. Once that clicked, building the block-by-block architecture — the patch embeddings, the linear projections, and the multi-head attention — felt almost like reconstructing a Transformer for NLP.

This entire Vision Transformer project has been one of those learning journeys that bridges theory and real-world application. From crafting the **core architecture in `model.py`** to building a flexible **custom dataset pipeline in `dataset.py`**, and finally tying it all together with **Hydra-powered configuration and training logic in `train.py`**, the experience has been deeply rewarding.

Adding the **Streamlit frontend** was the cherry on top — turning what started as a research-focused implementation into something practically usable and visually engaging.

But more than the code, what I really gained here was a deeper understanding of how design choices flow through a project — from model layers to configuration management, visualization, and deployment. Each piece now feels like part of a cohesive system, not isolated scripts.

And as I continue improving this — integrating DDP, TensorBoard, and cloud deployment — my goal remains the same:
to make machine learning systems that are **transparent, scalable, and truly enjoyable to work with.**

Thank you for taking the time to read and engage with this article. Your support in the form of following me and clapping on the article is highly valued and appreciated. If you have any queries or doubts about the content of this article, please do not hesitate to reach out to me via email at ***manindersingh120996@gmail.com****. You can also connect with me on* 🔗*LinkedIn*.

Computer Vision | Computer Science | Data Science | Machine Learning | Software Engineering

Written by Maninder Singh

260 followers · 10 following

Data Scientist and NLP Specialist | Passionate to share my knowledge

Follow

Responses (3)

Alex Mylnikov

What are your thoughts?

D. Chakraborty
4 days ago

For reader's context: I went through his codes thoroughly and found out that it's ChatGPT generated and so is this article. As expected, people with no experience writing tech articles from "that" specific region.

16 1 reply Reply

Yeo Yong Hui
1 day ago

Clean and well-structured code, thorough and accessible article. Insightful with explanation of design intentions and choices. Good read for anyone looking to dabble in Deep Learning.

By the way, any consideration of using **MLflow** for Experiment tracking, instead of your custom find_experiment utility?

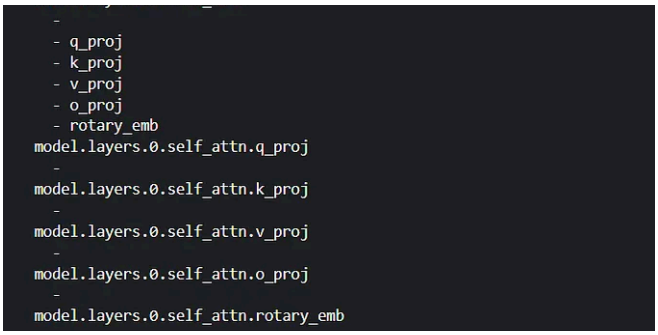
3 1 reply Reply

Goossens
3 days ago

No way to install the dataset properly. The "base_experimentations.ipynb" has no rule 5. I have the dataset in my .cache but no ready subset. How should the config.yaml be adapted to the train/test paths ?

2 replies Reply

More from Maninder Singh

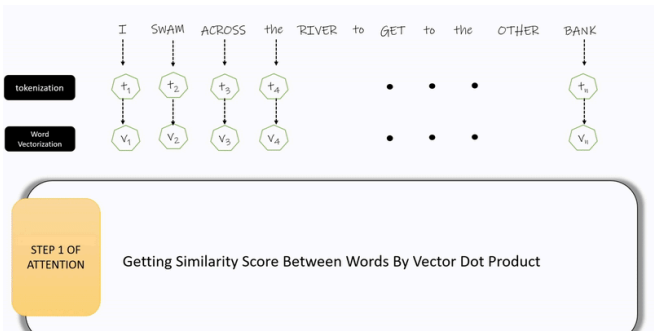


Maninder Singh

Practical Guide to Fine-tune LLMs with LoRA

LoRA(PeFT) makes Fine-Tuning Open-Source LLMs like LLama,Mistral,Gemma,etc on a...

Oct 13, 2024 285 3

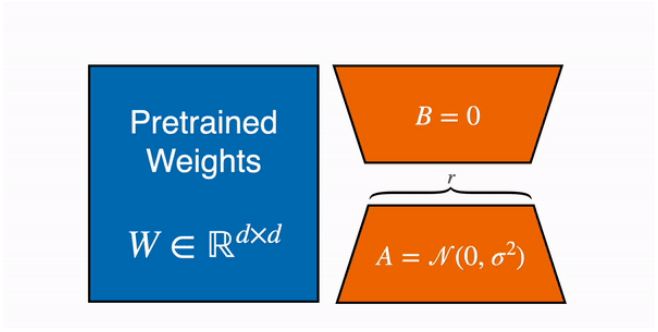


Maninder Singh

The Detailed Explanation of Self-Attention in Simple Words

The attention mechanism is one of the most groundbreaking concepts in deep learning. I...

Apr 1 240

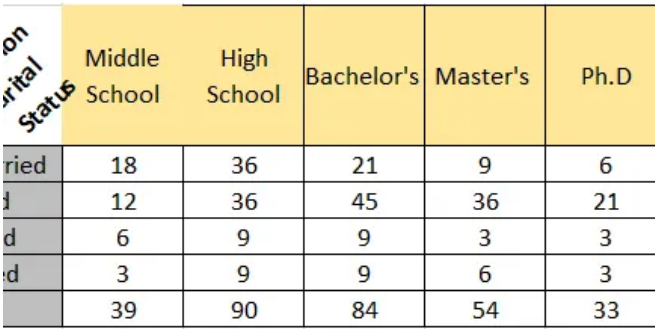


In Artificial Intelligence in Plain E... by Maninder Si...

Understanding Low-Rank Adaptation (LoRA) for Efficient...

LoRA, an efficient adaptation strategy that maintains high model quality without...

Jun 23, 2024 222 1



Status	Middle School	High School	Bachelor's	Master's	Ph.D
ried	18	36	21	9	6
d	12	36	45	36	21
d	6	9	9	3	3
d	3	9	9	6	3
	39	90	84	54	33

Maninder Singh

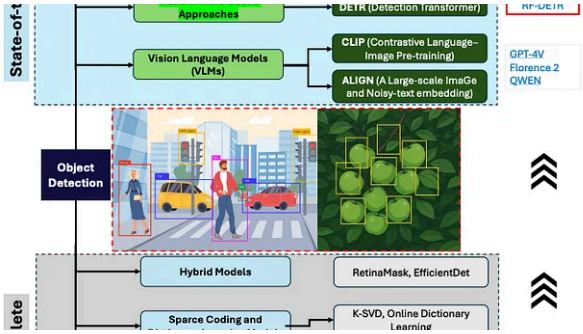
Understanding Categorical Correlations with Chi-Square Test...

In this, I explained about correlation(code) between categorical features which I Learne...

Jun 18, 2023 308 4

See all from Maninder Singh

Recommended from Medium

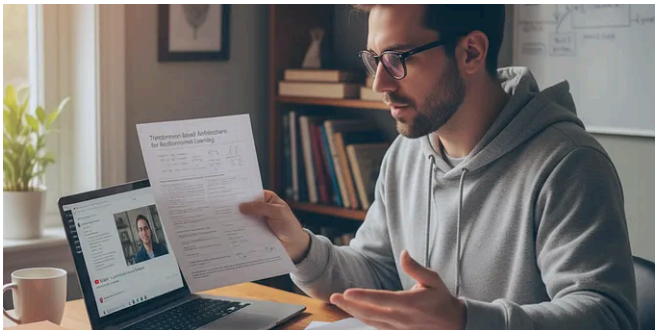


Md Faruk Alam

RF-DETR Object Detection vs YOLOv12: A Study of Transformer...

The object detection landscape is experiencing a fundamental shift. For years,...

Oct 14 38

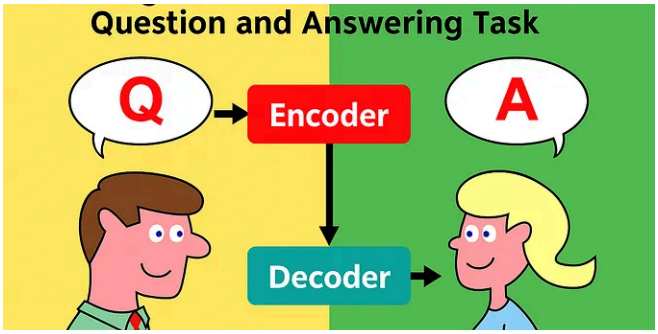


Ashanvi Yadav

How to Effectively Read Machine Learning Papers: A Beginner's...

From "What Does This Even Mean?" to "Actually, This Makes Sense"

Oct 16 227 7



In Toward Humanoids by Nickolaus Jackoski

Building An Encoder-Decoder For A Question and Answering Task

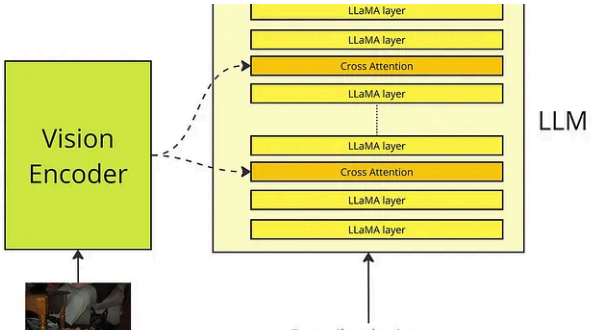
This article explores the architecture of Transformers which is one of the leading...

Simranjeet Singh

Top 15 GenAI/ML Interview Questions asked in FAANG 2025

Top 15 FAANG interview questions for GenAI, LLM, RAG, ML, explained with practical...

Oct 6 197 2



Saptarshi MT

Training a Vision Language Model from scratch (VLM multi-modal)

So after my previous blog on LLM from scratch i thought about training a model for...

Jun 11 5

5d ago 124 3

Logan Sritharan

The Rise of Self-Supervised Learning in Computer Vision

Exploring the DINO models and the Future of Label-Free Learning

Oct 16 25

See more recommendations