Computer Vision Notes

Ana Maria Sousa

Hugging Face

(Part III)

Additional notes material from

DeepLearning.AI

comet

Content

1. Video

An image is a binary, two-dimensional (2D) representation of visual data. A video is a multimedia format that sequentially displays these frames or images. Technically speaking, the frames are separate pictures. As a result, storing and playing these frames sequentially at a conventional speed giving the illusion of motion, thus resulting in the creation of a video.

1.1. Aspects of a Video

- **Resolution:** This refers to the number of pixels in each frame of the video, indicating its size. Common resolutions include HD (1280x720), Full HD (1920x1080), and 4K (3840x2160). Higher resolutions offer more detail but require more storage and processing power.
- **Frame Rate:** This is the number of frames displayed per second (fps) to create the illusion of motion. Common frame rates are 24, 30, and 60 fps. Higher frame rates produce smoother motion.
- **Bitrate:** The quantity of data needed to describe audio and video is called bitrate. This measures the amount of data used to encode the video and audio per second. Higher bitrates provide better quality but need more storage and bandwidth for streaming. Commonly expressed in megabytes per second (mbps) or kilobytes per second (kbps).

1.1.1. Codecs

Codecs (short for "compressor-decompressor") are tools used to compress and decompress digital media files. They reduce file size, making storage and transmission more efficient while preserving an acceptable level of quality. There are two main types of codecs:

- **Lossless Codecs:** Compress data without any loss of quality.
- **Lossy Codecs:** Compress data by removing some information, resulting in a loss of quality.

1.2. Applications of Video Processing

A video is a dynamic multimedia format that combines a sequence of individual frames, audio, and often additional metadata. It serves a wide range of applications:

- Human Action Recognition (HAR)
- Motion Detection
- Object Tracking
- Video Classification
- Object Detection (detecting moving objects)
- Behavior Analysis
- Gait Analysis
- Action Segmentation
- Scene Understanding

1.3. Deep Learning for Video Processing

Deep learning techniques have revolutionized video processing by enabling systems to extract intricate spatial and temporal patterns from video data.

Key approaches:

1. **CNN Based Approach**:
    - CNN models, initially designed for image recognition, have been extended for video processing. Techniques include:
        - Multi-resolution architectures for capturing local motion information.
        - **Context streams** for low-resolution image modeling.
        - **Fusion techniques** like early, late, and slow fusion for learning spatial-temporal features.
    - **Two-stream CNN** architectures integrate spatial and temporal information using separate streams for image and optical flow data.
    - Specialized models like MultiD-CNN for multimodal gesture recognition and DeepSORT for object tracking showcase the versatility of CNNs in various video processing tasks.

2. **DNN Based Approach**:
    - Deep neural networks (DNNs) with multiple layers excel in handling complex, high-dimensional video data.
    - Models like Multivariate Gaussian Fully Convolution Adversarial Autoencoder (MGFC-AAE) demonstrate robustness in video anomaly detection and localization.
3. **RNN Based Approach**:
    - Recurrent neural networks (RNNs) are tailored for sequential or time-series data, making them ideal for video processing.
    - Variants like LSTM and GRU are popular choices for modeling temporal dynamics in videos.
    - Applications range from group activity recognition to facial expression recognition and video manipulation detection.
4. **Hybrid Approach**:
    - Combining multiple deep learning methods offers enhanced capabilities in video processing.
    - Hybrid models leverage the strengths of different techniques, such as combining CNNs and LSTMs for human action recognition.
    - Transformer networks, integrating attention mechanisms, show promising results in tasks like gesture recognition and scene understanding.
    - Specialized architectures like Convolutional Recurrent Neural Networks (CRNNs) are adept at salient object detection and intelligent monitoring in dynamic environments.

These approaches highlight the diverse strategies researchers employ to tackle the challenges of video processing, from activity recognition to anomaly detection and beyond. Each method offers unique advantages and applications, contributing to the advancement of this field.

1.4. Video Captioning

Videos are temporal sequences and for description generation making it important to manage their varying lengths and dynamic content for generating descriptions. To do this effectively, several techniques can be used:

1. **Recurrent Networks and Attention Mechanisms:** These are used during the encoding stage to handle the sequence of frames and focus on important parts of the video.

2. **Multi-Stream Architectures:** Can accurately exploit spatio-temporal features by having different (specialized) streams to capture objects, object actions, or interactions between different objects. These use multiple specialized streams to capture different aspects of the video:
   - Static Features: Extracted from single frames to detect objects in the video.
   - Dynamic Features: Extracted using 3D Convolutional Neural Networks (3D-CNNs) to understand movement, such as in optical flow frames.

3. **Combining Visual and Semantic Features**: Both static (objects in single frames) and dynamic (movement across frames) features are considered. These features can be visual (what you see) and semantic (the meaning behind what you see).

Likewise, an image captioning static and dynamic features can be both visual and semantic features. The combined information from all streams is then used by the decoder to generate a coherent description of the video.

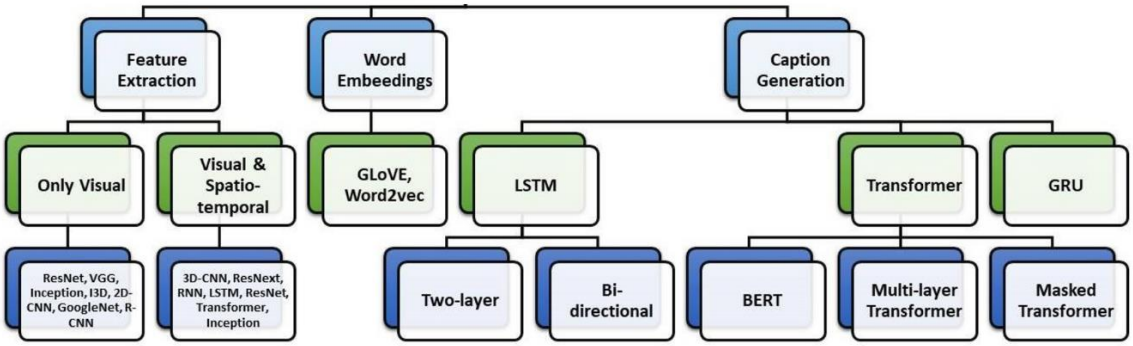


FIGURE 1- OVERVIEW OF DIFFERENT DEEP LEARNING METHODS IN VIDEO CAPTIONING

1.5. Transformers leading with Image Vs. Video

With the rise of transformers, vision transformers have become essential for various computer vision tasks involving images and videos.

However, these models process images and videos differently, and it is crucial to understand these differences for optimal performance and accurate results.

1.5.1. Understanding Image Processing for Vision Transformers

Vision transformers handle image data by breaking it down into smaller, non-overlapping patches. For example, a 224x224 image can be split into 16x16 patches, with each patch being 14x14 pixels. This approach reduces computation and helps the model capture **local features effectively.**

Each patch is processed through **self-attention layers and feed-forward neural networks to extract semantic information**. This hierarchical method allows vision transformers to capture both high-level and low-level features in the image.

1.5.1.1. Spatial Context challenge

Since transformers don't inherently track the position of inputs, the spatial context of the image can be lost. Thus, vision transformers use positional encodings to retain the spatial context of the image. These encodings help the model understand the relative positions of patches, improving its ability to recognize objects and patterns.

Unlike CNNs, which focus on learning spatial features, vision transformers are designed to learn both spatial and contextual features.

1.5.2. Understanding the Key Differences, Video Processing for Vision Transformers

Videos are sequences of frames and processing them requires **capturing motion information.** While transformers for images focus only on spatial details within a single frame, video processing needs to consider **the temporal (time-based)** relationships between frames.

Understanding these temporal relations is essential for interpreting video content accurately. For instance, for tasks such as action recognition (aims to classify actions in a video), and video captioning (generates textual descriptions of video content).

Temporal information helps vision transformers produce more contextually accurate captions by understanding the sequence of actions. This requires a different algorithm that includes the **time dimension in addition to the spatial dimensions.**

To handle this, there are two main approaches for extracting tokens or embedding video clips, which effectively incorporate the temporal aspect: Uniform Frame sampling and Tubelet Embedding

1.5.2.1. Uniform Frame Sampling

This method tokenizes a video by uniformly sampling frames from the clip.

Each 2D frame is processed independently, just like in image processing, and the tokens from all frames are combined.

If we extract $n_h * n_w$ non-overlapping patches from each frame, we get $n_h * n_h * n_w'$ tokens in total for the transformer encoder. Uniform frame sampling involves selecting frames from the video and applying standard Vision Transformer (ViT) tokenization to each frame.

- *Sample frames, extract 2D patches and linearly project (as in ViT)*
- *Effectively consider a video as a "big image"*

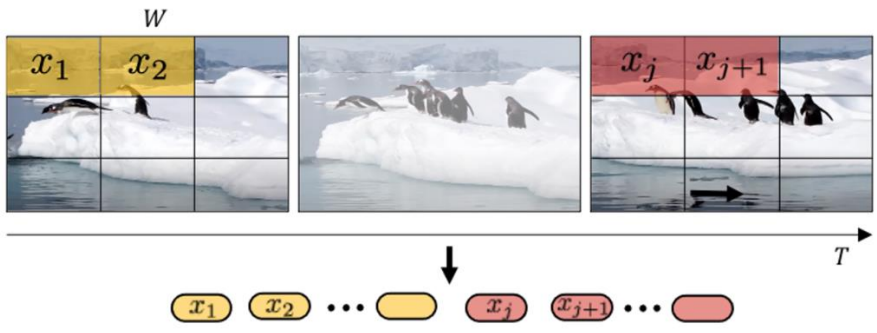


FIGURE 2- UNIFORM FRAME SAMPLING (GOOGLE RESEARCH)

1. Tubelet Embedding

This method adapts the vision transformer's image embedding to 3D, similar to a 3D convolution. Instead of processing frames independently, it extracts **non-overlapping spatiotemporal "tubes"** from the video.

First, tubes that include both spatial patches and temporal information are extracted from the video. These tubes are then flattened to create video tokens.

This method integrates **spatiotemporal information** during tokenization, unlike "uniform frame sampling," where the transformer combines temporal information from different frames later on.

- *Extract 3D tubelets to encode spatio-temporal "tubes" into tokens.*
- *Temporal information included from the initial tokenisation stage.*
- *Works better when initialised appropriately.*

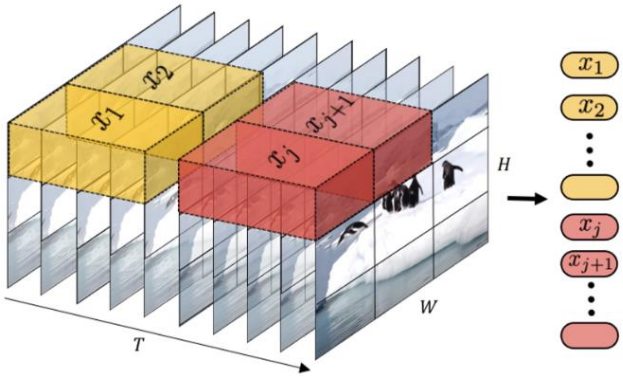


FIGURE 3- TUBELET EMBEDDING (GOOGLE RESEARCH)

Additional refs:

- *https://ieeexplore.ieee.org/document/9563948*
- *Large-Scale Video Understanding with Transformers*
- *Journal on Image and Video Processing*

2. 3D Computer Vision

Recent advancements in AI, particularly in language and vision, have brought forth new models like Llama 3, showcasing their utility across various tasks. However, the field of 3D technology has yet to experience a similar breakthrough.

While there is a surge of new research emerging, it often lacks clarity and consistency regarding the definition and representation of 3D. Common outputs include pre-rendered videos from diverse techniques like **meshes, splats, NeRFs**, and **multi-view diffusion**, but the methods and meanings behind these representations can be confusing.

The significance of 3D in advancing AI towards general intelligence is gaining recognition, as a grounded understanding of the 3D world is essential for developing intelligent systems.

2.1. Representations for 3D Data

Depending on the application, one of several different representations for 3D data might be used:

1. Point Clouds: Represent objects through a collection of 3D points, typically generated via 3D scanning (e.g., LIDAR). These lack connectivity information, making it hard to define surfaces and topology.

2. Meshes: Commonly used in computer graphics, they represent surfaces using interconnected triangles. They can include additional data like normals, colors, and textures, making them efficient for rendering solid objects in applications like games.
(The Python trimesh package aids in handling mesh data)

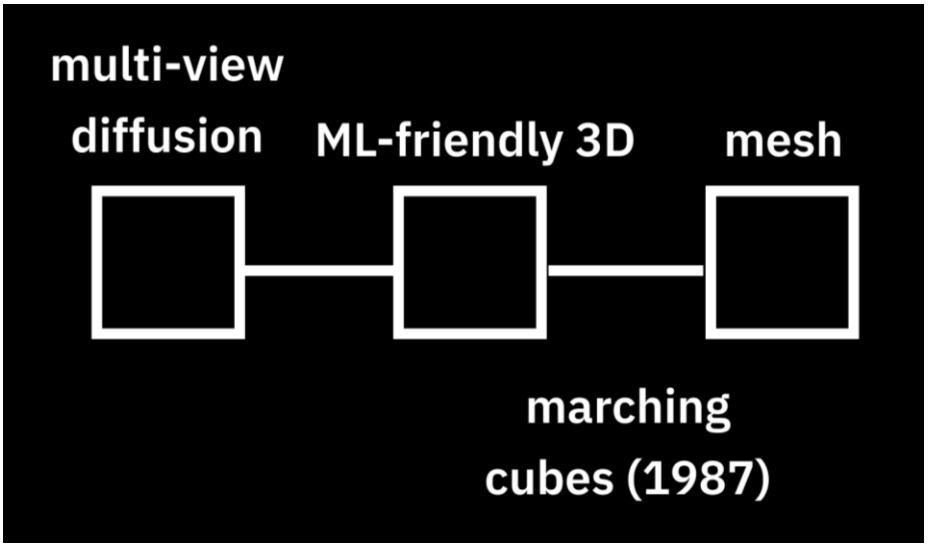
3. Volumetric Data: Used for transparent objects (e.g., clouds or fire), it maps positions in 3D space to attributes like density or color. Volumetric grids are one way to represent this data, but more advanced methods (e.g., neural networks) can improve accuracy.

4. Implicit Surfaces: Like volumetric data but focused on the surface of objects. The surface is defined as the zero point of a function, often a **signed distance function (SDF),** which measures the distance to the surface. Sphere tracing can quickly compute intersections with surfaces.

2.2. 3D Pipeline

A 3D pipeline consists of three key steps:

- **Multi-View Diffusion:** A diffusion model, like Stable Diffusion, generates novel views of an object from either source images or text descriptions. This part of the pipeline is highly technical and rapidly evolving, focusing more on diffusion models than traditional 3D techniques.
- **ML-Friendly 3D Representation:** The generated multi-view images are used to create a non-mesh, "ML-friendly" 3D representation. This can take the form of a voxel grid, sign distance field (SDF), or point cloud, which is suitable for machine learning purposes.
- **Mesh Conversion (Marching Cubes)**: Finally, the ML-friendly 3D representation is converted into a mesh using the Marching Cubes algorithm. This step produces a surface mesh composed of polygons (typically triangles) that can be used in applications like rendering or simulation.



2.3. Mesh and non-Mesh representations.

Meshes is the approach used to represent 3D in real-world application nowadays. A Mesh is a collection of vertices, edges and faces that define a 3D-object.

Problem: **Meshes are difficult for machine learning models**.

The Challenges

- **Why Meshes are Not Suitable for ML**: Meshes are inherently difficult for machine learning models to process. Meshes consist of vertices, edges, and faces that define the surface geometry of an object. Unlike grid-based data structures, meshes have irregular topologies, varying connectivity between points, and non-uniform resolution. This irregularity makes them unsuitable for standard ML models, which typically expect structured input data like grids (used for images) or sequences (used for text).
- **Progress in Multi-image diffusion:** Step 1 has seen rapid advancements with techniques like diffusion models, which generate novel 3D representations from multi-view images or even text prompts. These representations, such as voxel grids or point clouds, fit well within the framework of modern machine learning because they are structured and easier to handle.

- **Lack of Progress in conversion to Mesh methods:** While there is significant progress in generating "ML-friendly" 3D representations, converting these into meshes (using Marching Cubes) hasn't evolved much since the 1980s. This creates a bottleneck because while machine learning research advances, practical applications still depend on meshes for final outputs. However, meshes are less compatible with ML models, which limits the potential integration of these cutting-edge advancements into real-world applications.

The **gap** between **3D machine learning research and practical applications stems** from the **difficulty** in processing meshes with ML models. **Non-mesh 3D representations** work well for ML but need to be converted to meshes for real-world use, a step that hasn't seen much innovation. This disconnect highlights the challenge in bridging 3D research and application.

2.2.1 Non-Meshes

3D machine learning research often uses non-mesh representations. Common non-mesh formats include:

i) ***Triplanes*** (e.g., InstantMesh)
ii) ***NeRFs*** (Neural Radiance Fields, e.g., NeRFiller)
iii) ***Splats*** (e.g., LGM)

Gaussian splatting is a unique non-mesh representation where "splats" are small, Gaussian-shaped particles. Unlike other non-mesh formats, splats can be rendered in real-time, making them suitable for interactive applications like animation, physics simulations, and dynamic lighting.

Splats vs. Meshes

While splats have potential for real-time generative 3D content, they are unlikely to fully replace meshes. The entire 3D ecosystem (software, hardware, and workflows) is deeply built around meshes. Instead, splats are more likely to complement meshes, especially in real-time applications where fast rendering and dynamic scenes are important.

2.4. Multi-view Diffusion

Multi-view diffusion is a type of diffusion model, like **Stable Diffusion**, but instead of generating regular 2D images, it is trained on multiple views of the same object from different perspectives. The goal is to generate a consistent 3D representation by synthesizing these views.

Problems: The Janus Problem

This problem refers to a lack of consistency across different views of the same object. For example, an object might appear correct from one angle, but when viewed from another, it could have **extra faces or mismatched features** (like the Roman god Janus with two faces).

This **inconsistency happens** because the **model struggles to link the different views into a coherent**, unified 3D structure. As a result, the generated object may have errors, making multi-view diffusion difficult to use "out-of-the-box" for accurate 3D modeling. State-of-the-art multi-view-diffusion models like ***MVDream*** address this problem using specialized techniques.

2.5. ML-Friendly 3D / Gaussian splatting

Gaussian Splatting is a differentiable rasterization technique, which means it's designed to be **AI-compatible** while converting 3D data into 2D pixels for display.

In traditional **triangle rasterization**, 3D meshes are converted to 2D pixel data by making discrete decisions, like determining whether a pixel is inside a triangle. While this method is commonly used to render meshes, it's not ideal for neural networks, which prefer continuous, differentiable processes.

Differentiable rasterization, like Gaussian Splatting, avoids these hard, discrete decisions, allowing neural networks to work with smoother, more continuous data, making it better suited for AI-driven 3D tasks.

How do Gaussian Splatting work?

Gaussian Splatting is a **differentiable rasterization technique** that works by projecting millions of 3D points (splats) onto a 2D image. Each splat point has four key parameters:

1. **Position**: 3D location (XYZ).
2. **Covariance**: Describes how the point is stretched or shaped (3x3 matrix).
3. **Color**: RGB color values.
4. **Alpha**: Transparency level (α).

How It Works:

1. **Projection**: The 3D splat points are projected into 2D space and sorted for rendering (back-to-front blending).
2. **Contribution Calculation**: Each splat point contributes to the color of every pixel in the image. The farther a point is from a pixel, the smaller its contribution.
    - In theory, every point contributes to every pixel, but this is optimized using **tile-based rasterization** to improve efficiency.

Differentiability:

- This process is **differentiable**, meaning it works well with machine learning models, allowing smooth adjustments to point parameters (position, color, etc.) during training.

Training:

- The points are initialized using **Structure-from-Motion**, a traditional 3D reconstruction method. The system then compares the rasterized image to the ground truth and adjusts the point parameters using gradient descent to reduce the error.
- **Automated Densification and Pruning**: The system adds or removes points dynamically during training to optimize the representation.

Inference:

- For inference (rendering without training), Gaussian splatting can be simplified by treating each point as a small 2D quad, which is efficient and used in open-source tools like **gsplat**.
- This technique is also scalable to more complex 3D models like **generative 3D networks**, as seen in tools like **LGM** for generating dynamic 3D scenes.

2.6. Meshes

As mentioned previously, Marching Cubes can produce dense and rough meshes unsuitable for production.

However, recent advancements have introduced methods that enable the conversion of **dense meshes to low-poly meshes in a differentiable manner,** meaning the process can be smoothly integrated into optimization pipelines. This allows for automatic refinement of the mesh, striking a balance between model detail and computational efficiency, with applications in areas such as computer graphics and 3D modeling.

Key Concepts:

- **Marching Cubes:** An algorithm used for generating a 3D surface mesh from volumetric data (like medical imaging or voxel grids). While efficient, it often results in meshes that are overly complex.

- **Mesh Generation:** The process of creating 3D surface models from different types of input data, such as point clouds or voxel grids. This step is critical for applications in simulations, video games, CAD, etc.

2.6.1. Marching Cubes

The **Marching Cubes** algorithm is used to generate 3D surface meshes from volumetric data (like a point cloud or voxel grid). It converts a volume into a mesh by dividing the space into small **cubes (voxels)** and figuring out how these cubes should be connected with triangles to form a surface.

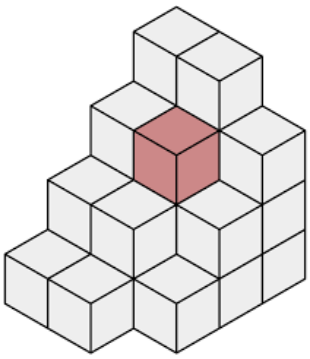
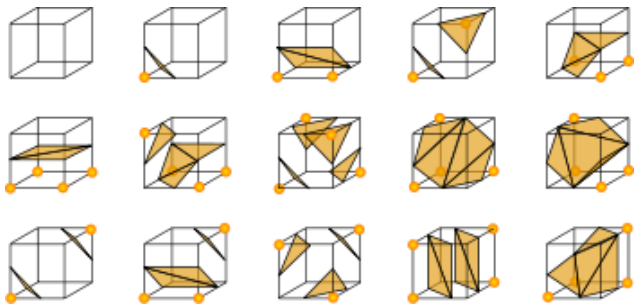


FIGURE 4 – VOXEL.

How it works step-by-step:

1. **Divide the Space into Voxels**: The 3D space is split into small cubes (voxels). The smaller the cubes, the higher the mesh detail/resolution.
2. **Sample the Eight Vertex Positions**: For each cube (voxel), the algorithm checks whether the eight corners of the cube are inside or outside the surface.
3. **Determine the Triangle Configuration**: Based on which corners are inside or outside, the algorithm chooses one of 256 possible patterns of how triangles should connect the corners to form part of the surface.



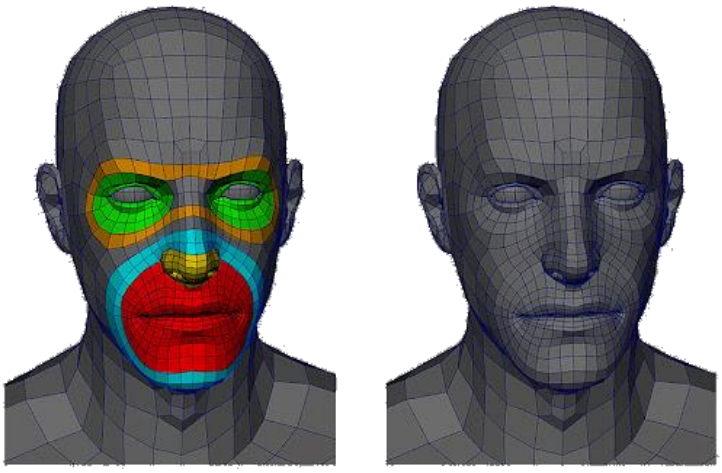
4. **Generate the Mesh**: The algorithm processes each voxel and applies the correct triangle configuration to create the overall surface.

Limitations:

- **High Polygon Count**: The mesh produced by Marching Cubes is very dense, with many triangles, which can be a problem for real-time applications, where fewer polygons are needed for fast rendering.

- **Edge Flow**: The triangles are not well-organized for smooth animation, leading to problems like unwanted deformations when the mesh moves.
- **Texturing Issues**: Since the mesh is dense and irregular, it becomes hard to apply textures cleanly, leading to visible texture distortions.

In simple terms, while **Marching Cubes** is great for quickly visualizing 3D surfaces, it creates very complex and **messy meshes** that aren't ideal for real-time use.



Improvements:

Techniques like **FlexiCubes** attempt to improve the **Marching Cubes** algorithm. Unlike the rigid, fixed positioning of vertices in Marching Cubes.

FlexiCubes allows the vertices of the mesh to **move or "flex"** to better match the shape of the object being represented. This flexibility helps create smoother surfaces and improves the quality of the mesh without significantly increasing its complexity. **FlexiCubes** generates cleaner and less "jagged" surfaces, which addresses some of the roughness that is common in standard **Marching Cubes meshes.**

This is used by tools like **InstantMesh**, a popular open-source 3D pipeline. **Instant Meshes** is a tool designed to produce high-quality **quad-dominant** (four-sided polygon) meshes from existing 3D models, like those generated by algorithms such as Marching Cubes. Its primary goal is to **remesh** a dense, irregular mesh into a cleaner, more organized structure that is better suited for production tasks like animation, rendering, or 3D printing.

Despite these improvements, the resulting meshes are still too dense and impractical for use in production environments.

SUMMARY:

Cleaning up the topology of meshes generated by **Marching Cubes** can be more time-consuming and resource intensive than creating a new mesh from the ground up, posing a significant bottleneck for machine learning applications in 3D modeling.

While **Gaussian Splatting** provides a potential workaround for this issue, recent advancements have introduced differentiable techniques that directly tackle the problem, enabling the creation of l**ow poly meshes with improved topology quality**. Further details on these innovations will be explored in the next section.

2.6.2. Mesh Generation

Recent advancements have introduced new solutions to overcome the limitations of the Marching Cubes algorithm, particularly in **converting dense meshes into low-polygon (low-poly)** meshes.

Traditional mesh rendering processes involve discrete decisions that are not differentiable, such as determining if a pixel is within a triangle. However, recently a differentiable approach to mesh generation was proposed, **treating mesh triangles as discrete symbols, similar to how words are treated in language models.**

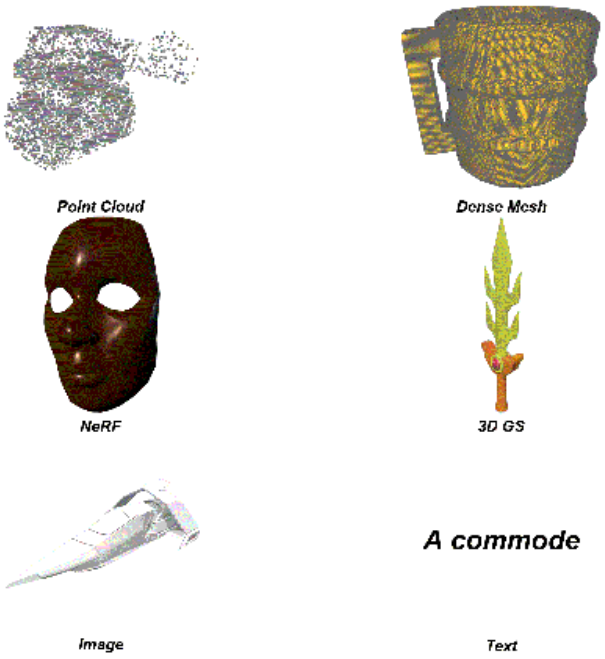
MeshAnything

"MeshAnything: Artist-Created Mesh Generation with Autoregressive Transformers." is designed to convert dense meshes into low-poly meshes. It employs techniques from **MeshGPT**, focusing on two main components:

1. **VQ-VAE Encoder**: This part encodes dense 3D data into a more compact, discrete representation using a **Vector Quantization Variational Autoencoder** (VQ-VAE). Essentially, it simplifies the complex mesh data into a manageable format.
2. **Autoregressive Transformer Decoder**: This component generates the mesh triangles in a sequential manner, like how text is generated in natural language processing, using an autoregressive transformer model.

Limitations: MeshAnything is a major advancement in 3D mesh generation, but its current results are often similar to or **worse than traditional methods** like Decimate, still requiring significant manual refinement.

The introduction of differentiable mesh generation allows for more context-aware topology reduction, meaning that it can take into account the shape and potential deformations of the mesh. While this area is still under development, successfully tackling these challenges could lead to the creation of highly effective 3D modeling tools.



2.7. Novel View Synthesis

Novel View Synthesis (NVS) is a field in computer vision that focuses on **creating new perspectives of a scene from unseen camera angles**, based on a set of input images. These new views are generated in a way that ensures consistency with the original images.

To achieve this, NVS methods are typically grouped into two main categories:

I. Methods using Intermediate 3D representation.
II. Direct Image Synthesis Methods

2.7.1. Methods Using Intermediate 3D Representations.

These approaches create a **3D model** of the scene as an intermediate step. Once the 3D structure is built, it can be "rendered" (converted back into 2D images) from any desired viewing angle.

- **How It Works**:
  - Use input images to infer a 3D structure or geometry of the scene.
  - Techniques like NeRF (Neural Radiance Fields) are popular in this space. NeRF represents the scene as a volumetric function, which predicts color and density for every 3D point when queried from a specific angle.
  - This 3D representation ensures that all generated views are consistent with the physical constraints of the scene.
- **Pros**:
  - Excellent for generating highly accurate and physically consistent views.
  - Enables further use in applications requiring explicit 3D models, such as virtual reality or simulations.
- **Cons**:
  - Computationally expensive to create the 3D model.

- Often requires significant processing for rendering.

2.7.2. Direct Image Synthesis Methods

These methods bypass the creation of a 3D model and directly generate a new image from a desired viewpoint.

- **How It Works**:
  - Use machine learning (often neural networks like **GANs or diffusion models**) to learn the relationships between the input images and the desired new view.
  - The model learns how changes in viewpoint should translate into changes in the image, without explicitly building the underlying 3D structure.
- **Pros**:
  - Faster and less resource-intensive compared to methods that generate 3D representations.
  - Can produce visually appealing results, especially in scenarios where perfect physical accuracy isn't necessary.
- **Cons**:
  - Lacks physical consistency in some cases, as it doesn't understand the actual 3D structure.
  - Struggles with complex geometries or occlusions (hidden parts of objects in the scene).

2.7.3. Challenges

Creating new views in Novel View Synthesis (NVS) is **challenging** because the **task is often underdetermined**—there isn't enough information in the input images to fully predict what unseen parts of the scene should look like. For example:

- If you see the back of a sign, the front could have any number of designs.
- Parts of objects may be occluded (hidden) by other parts, making their appearance ambiguous.

Challenges in Prediction

When models are trained to **directly predict unseen parts** (using **a loss function penalizing errors i**n reconstructing views), they tend to **predict blurry, uncertain regions** (e.g., grey areas) where the true appearance is ambiguous.

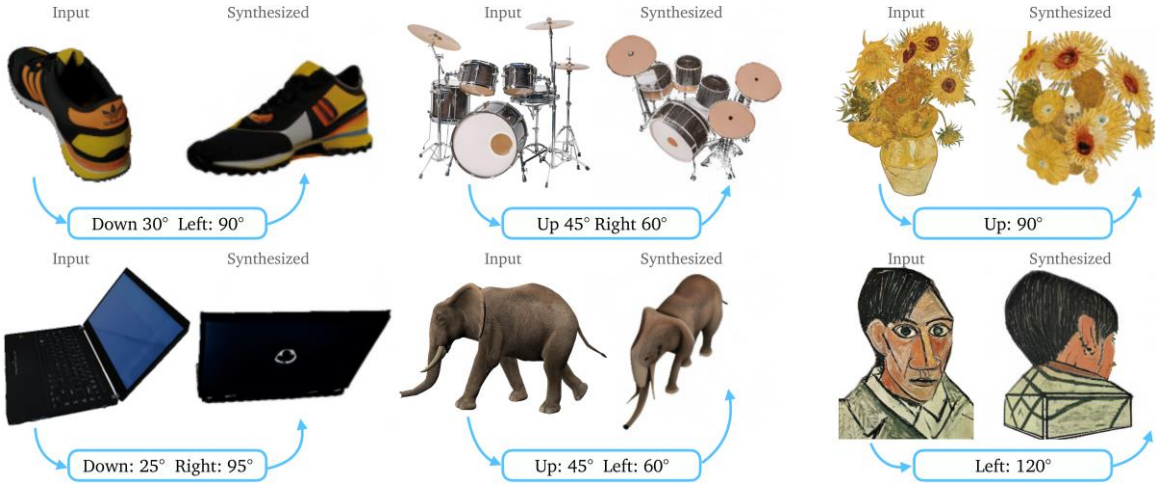
This happens because the model tries to average all possibilities. This problem has motivated the use of **generative models**, like diffusion-based models, which can produce multiple plausible guesses for the unseen parts instead of averaging them.

There we are going to briefly explore two approaches ***PixelNeRF*** (method that uses an intermediate 3D representation**)** and ***Zero123*** (methods that directly generate new images).

2.7.2.1. Zero123

Zero123 works by taking an input image and the relative viewpoint transformation (described in spherical coordinates) to produce a plausible, 3D-consistent image from the new angle.

The model builds on the **Stable Diffusion architecture**, specifically leveraging weights from Stable Diffusion Image Variations.



Instead of using a text prompt for guidance, it uses **CLIP image embeddings**—a representation of the input image. These embeddings are combined with the viewpoint transformation to condition the diffusion process. The key adaptation is that these embeddings, along with the latent representation of the input image, are concatenated channel-wise with the noisy latents before being passed to the denoising U-Net.

While much of the architecture remains similar to Stable Diffusion, these adjustments allow Zero123 to **handle viewpoint transformations effectively** and generate visually consistent outputs without relying on an intermediate 3D model.

Related methods

- 3DiM - X-UNet architecture, with cross-attention between input and noisy frames.
- Zero123-XL - Trained on the larger objaverseXL dataset.
- Stable Zero 123
- Zero123++ - Generates 6 new fixed views, at fixed relative positions to the input view, with reference attention between input and generated images.

2.7.1.1. PixelNeRF

PixelNeRF is a method that generates a **NeRF (Neural Radiance Field)** directly from one or more input images by conditioning the NeRF on features extracted from these images.

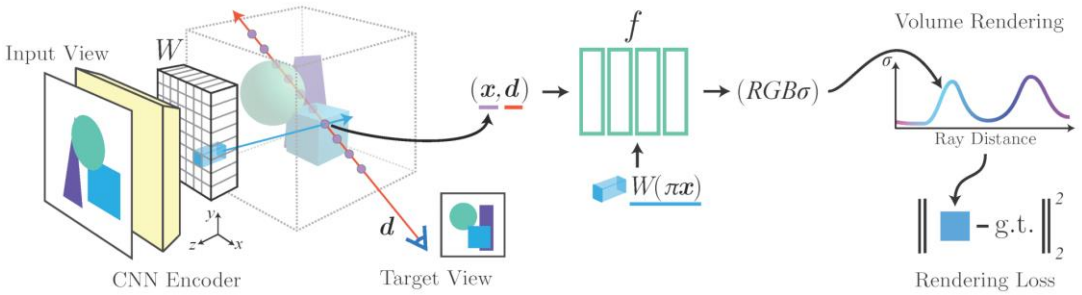
Unlike the original NeRF, which trains a multi-layer perceptron (MLP) from scratch to map spatial points to density and color, PixelNeRF **uses image-based spatial features to inform the NeRF.**

1. The process starts by passing the input images through a **convolutional neural network (ResNet34).**
2. Features from multiple layers of the network are **upsampled** to match the input image resolution.
3. For each query point in the 3D scene, PixelNeRF **identifies the corresponding location** in the input images (using the camera transformation). The features at these locations are extracted via bilinear interpolation.

4 These image features are then combined with the positional encoding of the query point and the viewing direction.

The NeRF network itself is structured with ResNet blocks:

- In the first three blocks, the interpolated image features are added as inputs.
- Two additional blocks process the features further before producing the final output: RGB values and density.



When multiple input views are provided, the features are processed independently in the early layers and averaged in later layers.

PixelNeRF was originally trained on the ShapeNet dataset, using mean-squared error loss to compare the rendered and expected views. Training was performed separately for different object categories (e.g., cars, planes), with models typically trained on one or two input images to predict a single novel view.

This method allows PixelNeRF to synthesize new views of a scene by leveraging both the spatial information from the input images and the volumetric rendering approach of NeRF.

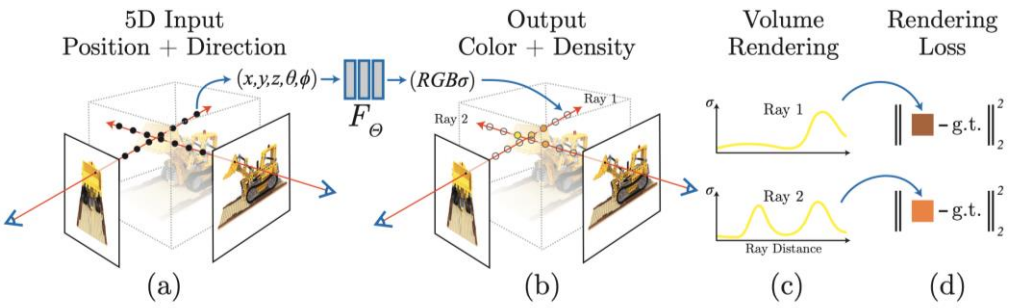
Related methods

- In the ObjaverseXL paper, PixelNeRF was trained on a *much* larger dataset allenai/objaverse-xl.
- Generative Query Networks
- Scene Representation Networks
- LRM

2.8. Neural Radial Fields (NeRFs)

Neural Radiance Fields (NeRFs) represent 3D scenes using a neural network, making them an **implicit representation** of a scene.

Instead of explicitly storing color or density values in 3D grids (as in voxel-based methods), NeRFs encode this information in the weights of a **Multi-Layer Perceptron (MLP)**.



This allows for high-quality **novel view synthesis** while using much less memory, as the MLP's size is independent of the scene's complexity.

How NeRFs Work

NeRFs model a scene as a continuous function that maps:

- A **3D position** *X* and a **2D viewing direction θ.**
- To a **color** *C* and **density σ.**

This function, approximated by an MLP, outputs the scene information needed to render images from novel viewpoints. The rendering process involves:

1. Sampling 3D points along rays cast from the camera.
2. Passing the sampled points and directions through the MLP to get color and density values.
3. Using **volumetric rendering** to combine these samples into a 2D image.

The model is trained using a **reconstruction loss** that compares the rendered pixels to the ground truth, typically using a pixel-wise mean squared error.

Volumetric Rendering

NeRF uses a differentiable **volumetric rendering** equation derived from classical computer graphics. It computes the expected color of a ray by integrating the contributions of sampled points along the ray:

- **Color contributions** depend on **density σ, color C,** and accumulated transmittance **T(t)** (how much light reaches a point without being blocked).
- In practice, this integration is approximated as a weighted sum over discrete samples, making it computationally feasible and suitable for backpropagation.

$$\mathbf{C}(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt$$

In the equation above, $\mathbf{C}(\mathbf{r})$ is the expected colour of a camera ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, where $\mathbf{o} \in \mathbb{R}^3$ is the origin of the camera, $\boldsymbol{d} \in \mathbb{R}^3$ is the viewing direction as a 3D unit vector and $t \in \mathbb{R}_+$ is the distance along the ray. t_n and t_f stand for the near and far bounds of the ray, respectively. $T(t)$ denotes the accumulated transmittance along ray $\mathbf{r}(t)$ from t_n to t.

After discretization, the equation above can be computed as the following sum:

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))\mathbf{c}_i\,, \text{ where } T_i = \exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right)$$

Loss formulation

As the discretized volumetric rendeing equation is fully differentiable, the weights of the underlying neural network can then be trained using a reconstruction loss on the rendered pixels. Many NeRF approaches use a pixel-wise error term that can be written as follows:

$$\mathcal{L}_{\text{recon}}(\hat{\boldsymbol{C}}, \boldsymbol{C}^*) = \left\|\hat{\boldsymbol{C}} - \boldsymbol{C}^*\right\|^2$$

,where $\hat{\boldsymbol{C}}$ is the rendered pixel colour and $\boldsymbol{C}^*$ is the ground truth pixel colour.

NeRFs have rapidly evolved, with numerous papers and implementations since their introduction. They remain at the forefront of 3D scene representation and rendering, with continued efforts to improve speed, scalability, and real-world applications.

Key Advancements

NeRF's initial formulation in 2020 required long training times. Recent innovations include:

- **Instant-ngp (2022)**: Uses trainable hash-based encoding to drastically reduce training times while maintaining quality.
- **MipNeRF-360 (2022)**: Introduced techniques for handling unbounded real-world scenes.
- **Zip-NeRF (2023)**: Combines advancements like hash-based encoding and scene contraction, reducing training times to under an hour.

Current Research Directions

The field of NeRFs is advancing rapidly, with numerous publications pushing the boundaries of what NeRFs can achieve. Recent efforts focus on:

1. *Training and Rendering Speed:*
   - Methods like **VR-NeRF** and **SMERF** aim to achieve real-time streaming of real-world scenes, even on edge devices.
2. *Expanding Capabilities:*
   Research extends beyond speed optimization to explore:
   - **Generative NeRFs**: For creating entirely new 3D content.
   - **Pose Estimation**: Using NeRFs to estimate camera and object poses.
   - **Deformable NeRFs**: Handling dynamic scenes with moving objects.

- **Compositionality**: Representing scenes with modular components that can be manipulated independently.

3. Model Optimization

Model optimization is the process of modifying a trained model to improve its efficiency, especially for deployment on hardware with lower specifications than what was used during training.

While **training** may occur **on high-performance GPUs**, **inference** often takes place on smaller, **less powerful devices** like microcomputers, mobile phones, or IoT devices. Without optimization, issues like large model size, slow prediction times, and memory limitations can arise. Optimization ensures the model can run smoothly on these lower-spec devices by reducing resource usage without sacrificing much accuracy.

3.1. Why is Model Optimization important?

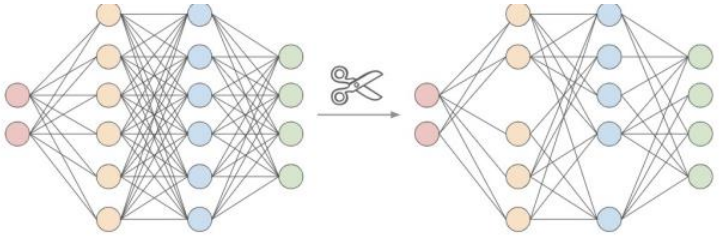
Model optimization is crucial for deploying computer vision models because of several key factors:

- **Resource limitations**: Computer vision models often need a lot of memory, CPU, or GPU power. Devices like mobile phones or embedded systems may not have these resources, so optimization reduces model size and computation to make them deployable.
- **Latency requirements**: Some applications, like self-driving cars or augmented reality, need real-time responses. Optimization helps increase inference speed, ensuring the model meets strict timing needs.
- **Power consumption**: Devices running on batteries, such as drones or wearables, need models that use less energy. Optimized models consume less power, extending battery life.
- **Hardware compatibility**: Different hardware has different capabilities. Optimizing the model for specific hardware ensures it runs efficiently on the target platform.

3.2. Types of model Optimization Approaches

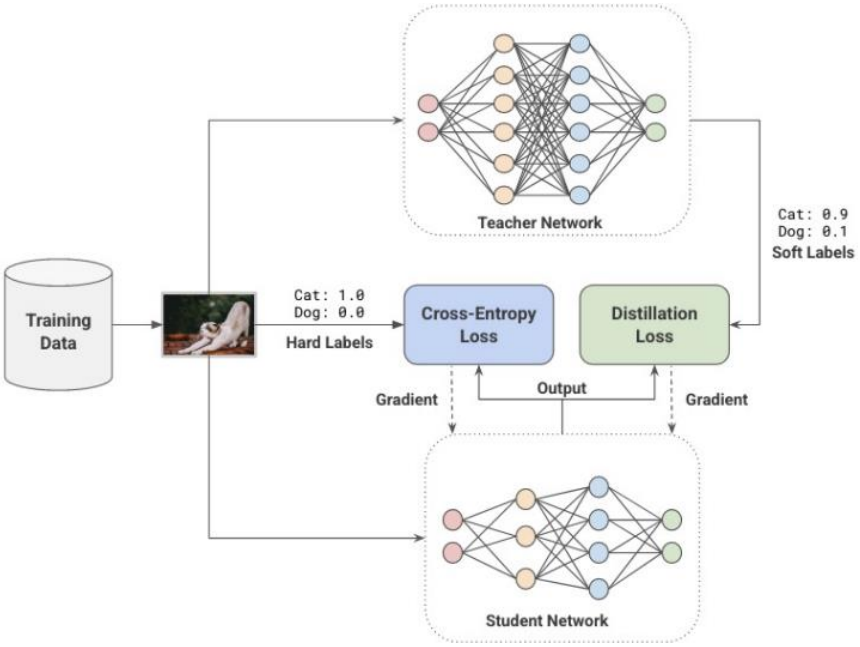
There are several key techniques for model optimization:

- **Pruning**: Removes unnecessary or redundant connections in the model to reduce its size and complexity.



- **Quantization**: Converts high-precision model weights (like 32-bit floats) to lower-precision formats (such as 16-bit or 8-bit integers) to decrease memory usage and speed up inference.

- **Knowledge Distillation**: Transfers knowledge from a larger, complex model (teacher model) to a smaller model (student model) by training the student to mimic the teacher's behavior.



- **Low-rank approximation**: Simplifies large matrices by approximating them with smaller ones, cutting down memory and computation costs.

- **Model compression with hardware accelerators**: Combines techniques like pruning and quantization but optimizes models specifically for certain hardware, such as NVIDIA GPUs or Intel devices.

3.3. Trade-offs

When deploying a model, there's a trade-off between **accuracy**, **performance**, and **resource usage**, and prioritizing one often affects the others.

- **Accuracy** refers to how well the model predicts correctly. Higher accuracy is important but typically demands more resources and slower performance, especially on devices with limited memory.
- **Performance** measures the model's speed and efficiency, which is critical for real-time applications. However, improving performance usually decreases accuracy.
- **Resource usage** is the computational cost (CPU, memory, storage) required for inference. Efficient use of resources is essential for deployment on resource-constrained devices like smartphones or IoT devices.

In optimizing models, we aim to balance these three aspects. Focusing on high accuracy can slow down inference or increase resource needs, so applying optimization techniques like pruning, quantization, or knowledge distillation helps achieve a balance between accuracy, speed, and resource efficiency for specific use cases.

4. Model Deployment

When deploying machine learning models, several key aspects and platforms must be considered. The deployment landscape involves diverse platforms, important practices like serialization and packaging, and strategies for serving models effectively.

4.1. Deployment Platforms

4.1.1 Cloud

Platforms like AWS, Google Cloud, and Azure provide scalable infrastructure for deploying AI models with managed services.

- ***Advantages:***
  - High scalability with powerful computing and memory resources.
  - Seamless integration with other cloud services.
- ***Considerations:***
  - Costs related to infrastructure usage.
  - Addressing data privacy and network latency for real-time applications.

4.1.2 Edge

Deploying on edge devices (e.g., IoT devices, edge servers) allows models to run locally, reducing the need for cloud dependency.

- ***Advantages:***
  - Low latency and real-time processing.
  - Enhanced privacy by minimizing data transmission to the cloud.
- ***Challenges/Considerations:***
  - Limited computing and memory resources.
  - Optimization is needed for constrained hardware environments.
  - Edge deployments involve training models elsewhere (like in the cloud) and optimizing them for smaller, resource-limited devices.

4.1.3 Mobile

Mobile deployment requires optimizing models for performance and resource efficiency.

- ***Frameworks:***
  Tools like Core ML (for iOS) and TensorFlow Mobile (for Android) facilitate this process.

4.2. Model Serialization and Packaging

4.2.1. Serialization

Serialization converts a model into a format that can be easily stored or transmitted.

ONNX is a universal format that facilitates interoperability across frameworks like TensorFlow, PyTorch, and scikit-learn.

- PyTorch can export models using `torch.onnx.export()`
- TensorFlow can convert to ONNX using tools like `tf2onnx`.

4.2.2. Packaging

Packaging involves bundling the serialized model, along with dependencies and any pre/post-processing code, into a deployable unit.

- ***Cloud Deployment:*** Typically, serialized models are packaged into containers (e.g., Docker) and deployed as web services, with support for auto-scaling and load balancing.
- ***Managed Services:*** Services like 🤗 Inference Endpoints simplify deployment, allowing you to deploy models without managing containers or GPUs.

4.3. Model Serving and Inference

4.3.1. Model Serving

Serving makes the deployed model available for inference requests.

Methods:

- ***HTTP REST API:*** Models can be served via HTTP endpoints using frameworks like Flask, FastAPI, or TensorFlow Serving.
- ***gRPC:*** A high-performance communication protocol, language-agnostic and efficient for serving models.
- ***Cloud Services:*** Managed services on platforms like AWS, Azure, and GCP offer scalable options for model serving.

4.3.2. Inference

Inference is the process of generating predictions based on incoming data using the deployed model.

Client Interaction: Clients send input data to the model through the serving infrastructure and receive predictions as outputs.

4.3.3. Kubernetes

Kubernetes is a popular container orchestration platform that helps manage model deployment at scale, offering reliability and flexibility.

4.4. Best Practices for Deployment in Production

Deploying machine learning models in production requires best practices to ensure smooth performance, reliability, and scalability. Key considerations:

1. **MLOps**: This approach adapts DevOps principles for machine learning projects, covering version control, continuous integration and deployment (CI/CD), automation, and monitoring.
2. **Load Testing**: Test the model under various workloads to ensure responsiveness and stability in different conditions.
3. **Anomaly Detection**: Implement systems to detect changes in model behavior, such as a distribution shift, where incoming data deviates from the data the model was trained on, leading to reduced accuracy or performance.
4. **Real-time Monitoring**: Use monitoring tools to identify issues immediately, such as spikes in prediction errors or abnormal data patterns. These tools help detect and address problems quickly.
5. **Security and Privacy**: Ensure secure data transmission and model access by using encryption and setting up strict access controls to protect sensitive data.
6. **A/B Testing**: Test new model versions alongside the current one by splitting traffic between them. Compare performance metrics to determine which version performs better before full deployment.
7. **Continuous Evaluation**: Regularly assess the deployed model's performance and be ready to roll back to previous versions if problems arise.
8. **Documentation**: Keep detailed records of the model's architecture, dependencies, and performance metrics to ensure transparency and traceability.

5. Zero-Shot Computer Vision

6.1. Generalization & Domain Adaptation

Generalization is a model's ability to apply what it learned from training data to new, unseen data from the same distribution. It's essential for creating models that can recognize new examples (e.g., new cat pictures) without overfitting to the training data.

Domain adaptation deals with applying a model trained on one type of data (e.g., real cat photos) to a different type (e.g., cartoon cats), where the data distributions are different. This process is necessary when data changes significantly across different environments.

Both generalization and domain adaptation are challenging because models often overfit or struggle with differences in data distributions. They are crucial for building models that perform well in real-world, unpredictable scenarios without needing endless retraining.

6.1.1. Why Is This a Challenge?

The main challenges in generalization and domain adaptation arise from differences in **data distributions**. If the training data is too narrow or specific, the model struggles to generalize to new

data that varies in characteristics, like color or breed in cat images. **Overfitting** is another issue, where models learn overly specific patterns that don't transfer to new examples. In domain adaptation, the challenge is even greater due to the significant **shift** between training and target domains, such as recognizing cartoon cats after training on real photos, requiring additional adjustments.

6.1.2. Why Is This Needed?

Generalization and domain adaptation are crucial for creating models that can **handle real-world applications**, where the data they encounter is often more varied than the training set. They **eliminate the need for constant retraining,** allowing models to work effectively across a broad range of scenarios. Moreover, domain adaptation enables models to remain useful when the **types of data shift** over time, such as recognizing products in different styles or formats, ensuring they stay relevant without starting from scratch.

6.2. What is Zero-shot Learning?

Zero-shot learning (ZSL) is a machine learning approach where a model is tested on classes (categories) of data it has never encountered during training. This means that when the model is evaluated, all test examples belong to classes that were completely absent in the training set. It's akin to a student being tested on material they never studied, making the task particularly challenging.

There's an extension known as **generalized zero-shot learning (GZSL)**. In GZSL, the test set can include both classes the model has seen during training and new classes it hasn't encountered before. This reflects real-world situations more accurately, where models often need to deal with new, unseen categories while still performing on familiar ones.

Key Points/summary:

- **ZSL**: Tests on entirely new classes, no training data overlap.
- **GZSL**: Allows a mix of seen and unseen classes in testing, making it more applicable to real-world scenarios.

6.2.1. How Does Zero-shot Learning Work in CV?

Zero-shot learning (ZSL) in computer vision involves enabling a model to recognize and classify objects it has never encountered during training. While this concept is straightforward in natural language processing (NLP)—where models learn from large text datasets to understand patterns and semantic relationships—ZSL in computer vision is more complex.

Humans can recognize unfamiliar objects by leveraging knowledge from related objects. The ability to draw on contextual information is key to zero-shot computer vision, which relies on **multi-modal learning**—integrating different types of information (like visual features and semantic descriptions).

Unlike unsupervised learning, where models learn from unlabeled data, zero-shot learning remains a form of supervised learning. It operates on **dateless labels**, meaning the model learns to classify objects based on descriptions or attributes without having direct training examples.

Essentially, ZSL in computer vision allows models to leverage conceptual knowledge to recognize new classes, much like humans do when encountering unfamiliar items.