Object Detection/Segmentation

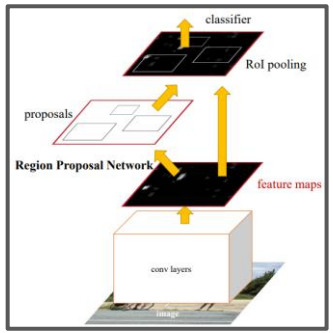
Describe Faster R-CNN. At a high level, how did it improve on Fast R-CNN and R-CNN?

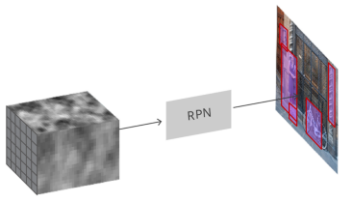
These three methods address object detection. Faster R-CNN works with 4 stages:

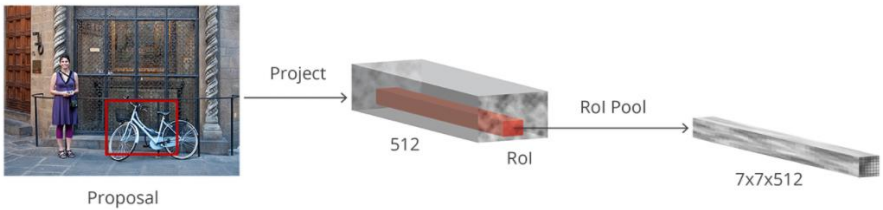
1) **Region Proposal Network (RPN)** takes a CNN feature map as input. Then for a large, discrete set of predefined anchors and bounding boxes on the feature map, it outputs
   a) A "objectness" score (representing foreground/background), formulated as binary classification
   b) Slight transformations on that bounding box, formulated as regression.
   The regular grid of anchor points naturally arises due to the downscaling of the FEN, it doesn't have to be implemented explicitly.
   This network is implemented in a fully convolutional way using 1x1 convolutions on the feature map. Only fg objects have a regression loss.
1) **Non-maximum suppression (NMS)** tries to remove duplicates, by iterating over the list of proposals sorted by score, and discarding those which have an IoU larger than some threshold with a proposal that has a higher score.
2) A **classification head,** which:
   a) Applies **ROI pooling** to randomly selected fg and bg bboxed feature proposals from stage 1, by projecting from image to feature map, and standardizing the size by splitting the ROI into a grid and max pooling (better than avg pooling in practice).
   b) Performs a final classification, to label it as one of the target classes, or background. This uses FC layers.
   c) Performs a final bbox regression loss based on class. This uses FC layers.
3) Loss is computed as a classification loss (both fg/bg bboxes) and bbox loss based on L1 dist (on only fg)

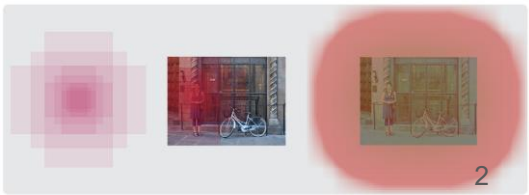
Some notes:

- RPN needs to have high recall, since if we don't get proposals here there's no chance they will be classified
- For some applications you can just use RPN to save time (eg if you are only recognizing a single class)
- Comparing the 3 methods:
  - R-CNN does not use a CNN to predict region proposals; it uses a selective search algorithm. Then, it uses a SVM to classify.
  - Fast R-CNN, still uses selective search, but uses ROI pooling and classifies with a CNN.
  - Faster R-CNN adds the RPN, and is thus end-to-end.
  - The time it takes to process an image are approx 50 secs for R-CNN, 2 secs for Fast R-CNN, and 0.2 secs for Faster R-CNN.









Left: Anchors, Center: Anchor for a single point, Right: All anchors

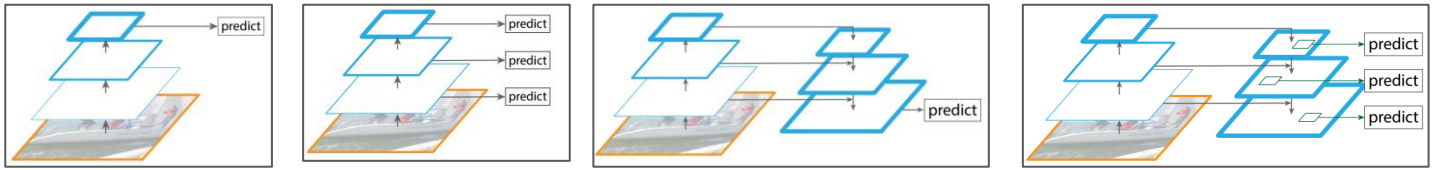
Explain the 1 stage detectors: YOLO, SSD, FPN, and RetinaNet. How do they compare to 2 stage detectors?

YOLO directly regresses bboxes with a CNN + FC layers:

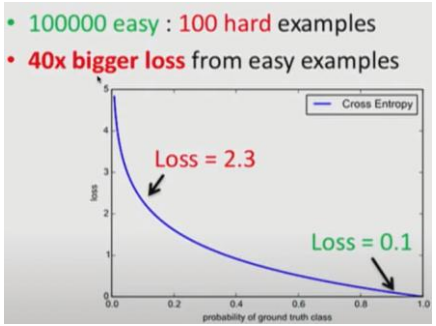
- Input is split to a grid, and the network outputs a fixed number of bboxes per square. For each bbox, there are parameters for (x,y) relative to center of obj, width/height of bbox, confidence, and probabilities for each class.

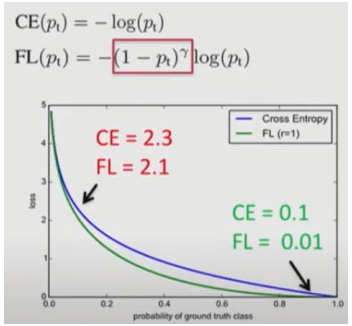
SSD is similar to YOLO, but predicts independent object detections from multiple feature maps of different resolutions. This helps with detecting small objects.

Feature Pyramid Networks (FPN) implements a U-shaped architecture which enables even better detection of small objects. As in U-Net, the intuition is to combine low-resolution, semantically strong features with high-resolution, semantically weak features via a top-down pathway and lateral connections to get rich semantics at all levels. It uses nearest neighbor upsampling (to "hallucinate" high level semantics, yet at high resolutions) and conv layers for upsampling, while U-Net uses transpose convolutions. From left to right, we show YOLO style, SSD style, UNet style, and FPN style.



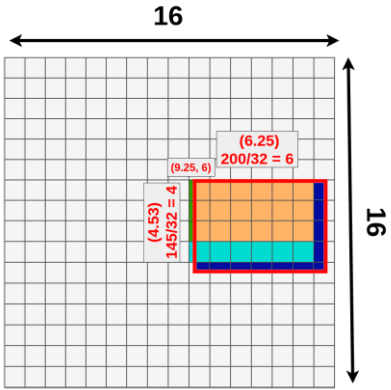
RetinaNet is essentially a combination of a ResNet backbone, FPN neck, and a Focal loss which helps with the fg/bg imbalance issue. Even with Cross Entropy loss's asymptotic behavior, the many easy bg examples can dominate the few hard examples. So, the focal loss adds a simple term to exacerbate the asymptotic behavior.

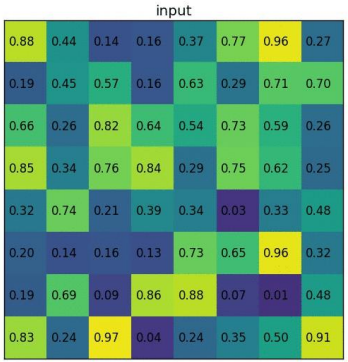




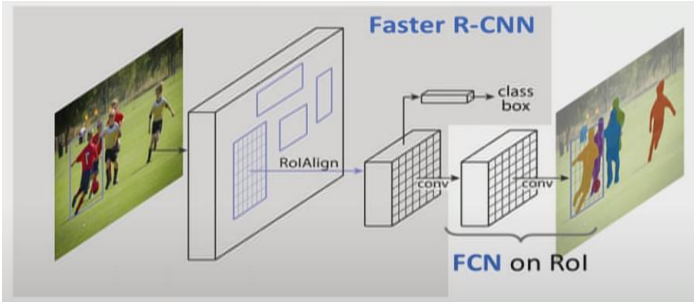
Explain how Mask-RCNN works.

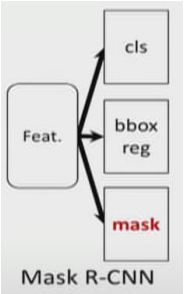
Essentially a Faster R-CNN (ResNet+FPN backbone), with a Fully Convolutional Network (FCN) on each ROI for segmentation.
Uses **RoI Align** instead of RoI (max) Pooling, which does not use any quantization and does not break pixel-to-pixel alignment, by using bilinear interpolation before the max pooling operation.

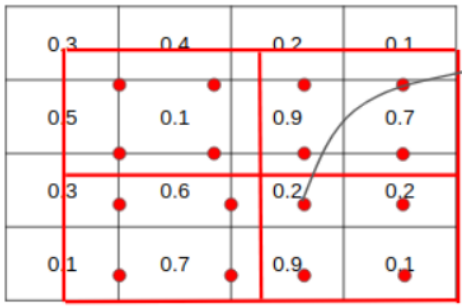


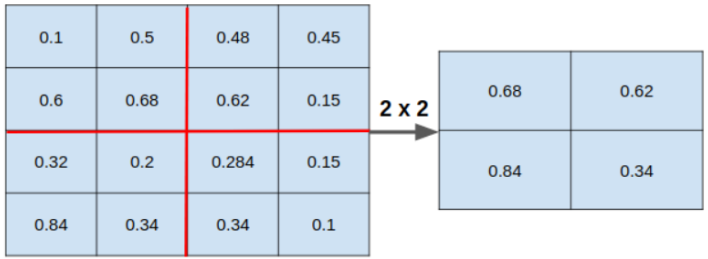


RoI (Max) Pooling









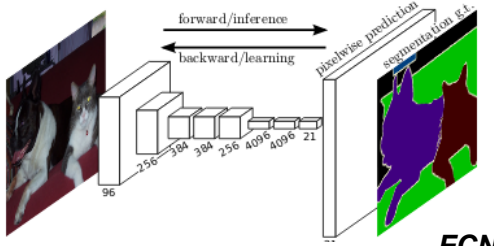
RoI Align

Explain FCN, U-Net, and DeepLab

All three are for semantic segmentation. Here, the key challenge is upsampling, while preserving detail.

Fully Convolutional Network (FCN) stacks a bunch of conv layers in an encoder-decoder fashion.

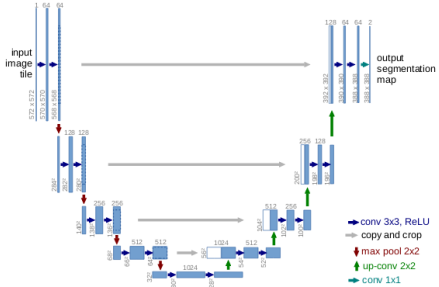
- No linear layers at the end for classification; only 1x1 convolutions. This allows different size img inputs.
- Upsamples only once in the decoder (using a deconvolution)
- Skip connections are added. The intuition is that this can recover fine-grained spatial information lost in the pooling/downsampling layers for making the final predictions



FCN

The **U-Net** is similar, but has some modifications:

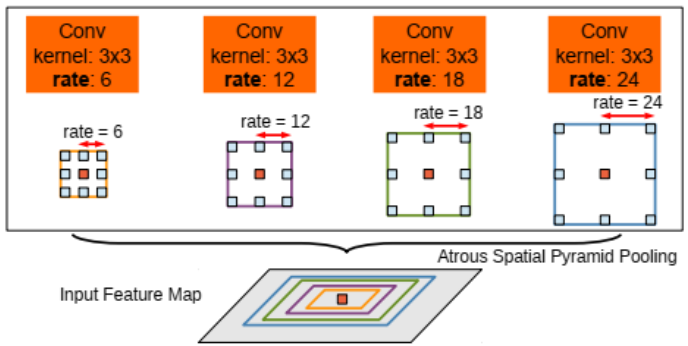
- Multiple upsampling layers, so that the encoder-decoder is symmetric
- Skip connections are concatenated

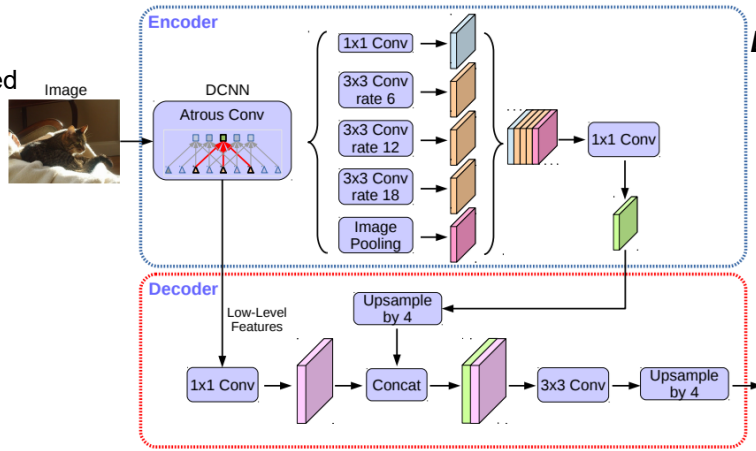


U-Net

The **DeepLabv3+** (from Google) is the best performing of the 3:

- Uses multiple Atrous convolutions, in a depthwise separable manner, at different dilation rates (creating a pyramid) in Encoder
- Upsamples 4x twice using bilinear interpolation, in Decoder
- Original DeepLab used conditional random fields (CRFs), but this was dropped





DeepLab

What losses are used for semantic segmentation?

- Cross entropy
- Dice loss
  - I.e., IoU loss
  - In practice a “soft dice loss” is used, where A is the probability map mask and the g.t. Mask
  - In general, this is conceptually more “direct”, but cross entropy can be better in implementation because it has nicer gradient properties which do not explode as easily

Dice loss = $1 - \frac{2|A \cap B|}{|A| + |B|}$

$$|A \cap B| = \begin{bmatrix} 0.01 & 0.03 & 0.02 & 0.02 \\ 0.05 & 0.12 & 0.09 & 0.07 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \xrightarrow{\text{element-wise multiply}} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix} \xrightarrow{\text{sum}} 7.41$$

prediction target

Explain how the Detection Transformer (DETR) works.

- DETR (from facebook) leverages a **transformer encoder-decoder** architecture for object detection
  - Moves away from traditional object detection like Faster-RCNN, which rely on complex pipelines with region proposals, anchor boxes, and NMS
- Transformer decoder attends to encoded features and object queries (fixed learned embeddings), to predict all bounding boxes and classes in parallel (not autoregressive)
  - Final decoder outputs passed to a FFN to predict a detection (class & bbox) or a “no object class”
- **Bipartite Matching Loss (Hungarian Loss)** used to enforce one-to-one mapping between predictions & ground truth
  - Since this method effectively outputs an unordered set, we need to assign GT to predictions before backprop
  - A cost function is computed for all pairings
  - Optimal matching is determined via the Hungarian algorithm (not differentiable, but doesn’t need to be; only a preprocessing step to assign GT)
  - Final loss includes classification loss and Generalized IoU + L1 loss for bounding boxes.

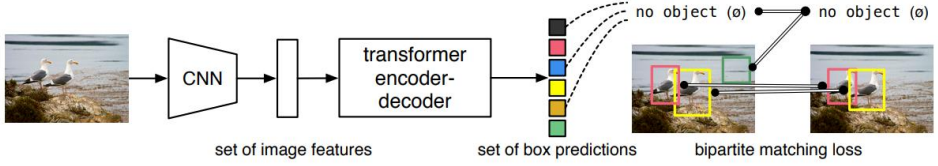


Fig. 1: DETR directly predicts (in parallel) the final set of detections by combining a common CNN with a transformer architecture. During training, bipartite matching uniquely assigns predictions with ground truth boxes. Prediction with no match should yield a “`no object`” (∅) class prediction.

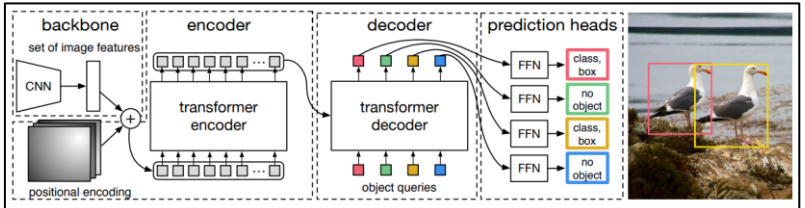


Fig. 2: DETR uses a conventional CNN backbone to learn a 2D representation of an input image. The model flattens it and supplements it with a positional encoding before passing it into a transformer encoder. A transformer decoder then takes as input a small fixed number of learned positional embeddings, which we call *object queries*, and additionally attends to the encoder output. We pass each output embedding of the decoder to a shared feed forward network (FFN) that predicts either a detection (class and bounding box) or a “`no object`” class.

Explain how Mask2Former works.

- General / “universal” formulation that address any segmentation task (panoptic, instance, & semantic)
  - Predicts a set of binary masks and assigns a single class to each mask. Each mask represents the segmentation of one instance
- Main MaskFormer components for K classes, up to N detections
  - **Backbone**: extracts low-resolution image features
  - **Pixel Decoder**: upsamples encoded features from backbone to high-resolution per-pixel embeddings
  - **Transformer Decoder**: object queries attend to image features. Produces N mask embeddings and class predictions.
  - Final N mask predictions computed by dot product of image features and mask embeddings
  - Matching between set of predictions and ground truth segments done using bipartite Hungarian matching
- Mask2Former improvements:
  - **Masked Attention**: Variant of cross attention that only attends within the predicted foreground mask region for each query
    - Leads to more effective learning/convergence; hypothesizes that local features can update query features, and context info can be gathered in self-attention later
    - Implemented by adjusting attention probabilities so that only positive mask regions have nonzero probability
  - Now, different transformer decoder layers can cross-attend to different sizes/resolutions of features
  - Switch order of self and cross/masked attention; now, cross attention goes first
    - Query features to the first self-attention layer are image-independent and do not have signals from the image, thus applying self- attention is unlikely to enrich information
  - **Improved training efficiency**
    - Uniformly randomly samples points to calculate bipartite matching step instead using whole mask.
    - In the final loss, sample points in mask for supervision using **importance sampling** (choosing regions with high uncertainty)
  - Remove dropout in residual connections and attention maps
  - Make query features learnable (previously, only query positional embeddings were learnable)

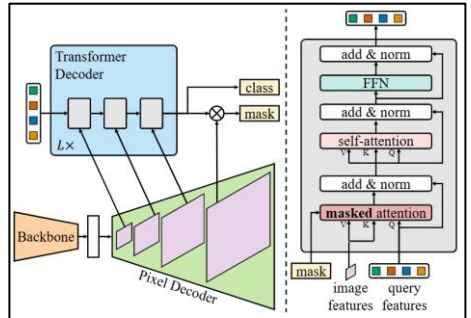


Figure 2. **Mask2Former overview.** Mask2Former adopts the same meta architecture as MaskFormer [14] with a backbone, a pixel decoder and a Transformer decoder. We propose a new Transformer decoder with *masked attention* instead of the standard cross-attention (Section 3.2.1). To deal with small objects, we propose an efficient way of utilizing high-resolution features from a pixel decoder by feeding one scale of the multi-scale feature to one Transformer decoder layer at a time (Section 3.2.2). In addition, we switch the order of self and cross-attention (*i.e.*, our masked attention), make query features learnable, and remove dropout to make computation more effective (Section 3.2.3). Note that positional embeddings and predictions from intermediate Transformer decoder layers are omitted in this figure for readability.

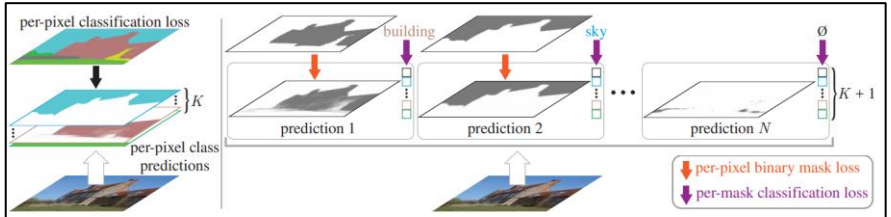


Figure 1: **Per-pixel classification *vs*. mask classification. (left)** Semantic segmentation with per-pixel classification applies the same classification loss to each location. **(right)** Mask classification predicts a set of binary masks and assigns a single class to each mask. Each prediction is supervised with a per-pixel binary mask loss and a classification loss. Matching between the set of predictions and ground truth segments can be done either via *bipartite matching* similarly to DETR [4] or by *fixed matching* via direct indexing if the number of predictions and classes match, *i.e.*, if $N = K$.

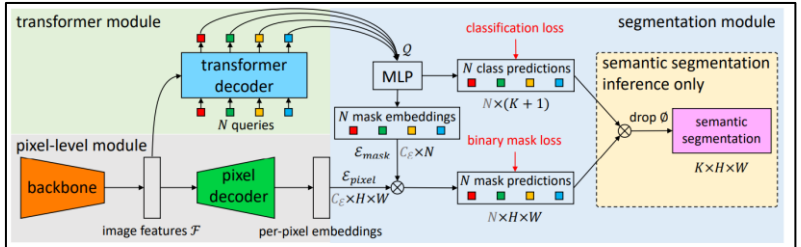
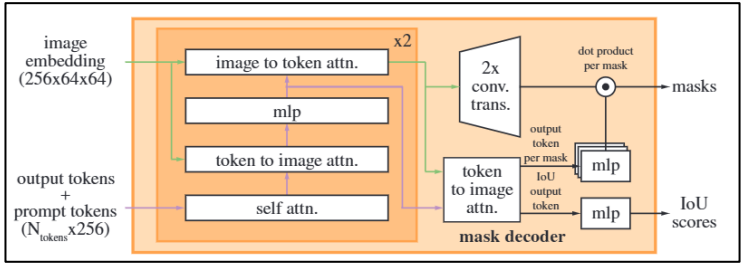


Figure 2: **MaskFormer overview.** We use a backbone to extract image features $\mathcal{F}$. A pixel decoder gradually upsamples image features to extract per-pixel embeddings $\mathcal{E}_{\text{pixel}}$. A transformer decoder attends to image features and produces N per-segment embeddings $\mathcal{Q}$. The embeddings independently generate N class predictions with N corresponding mask embeddings $\mathcal{E}_{\text{mask}}$. Then, the model predicts N possibly overlapping binary mask predictions via a dot product between pixel embeddings $\mathcal{E}_{\text{pixel}}$ and mask embeddings $\mathcal{E}_{\text{mask}}$ followed by a sigmoid activation. For semantic segmentation task we can get the final prediction by combining N binary masks with their class predictions using a simple matrix multiplication (see Section 3.4). Note, the dimensions for multiplication $\otimes$ are shown in gray.

How does Segment Anything (SAM) work?

- Given an image and prompt (eg positive/negative points, box, mask, or text), output a valid segmentation mask
  - "Valid" means that even when the prompt is ambiguous, it should still be reasonable.
- Marketed as a **foundation model** for image segmentation
  - Important qualities: **prompt engineerable** and generalizable to solve a range of downstream segmentation problems; easy to fit as a piece in a larger ML system, eg with VR using gaze detection.
- Architecture consists of different types of prompt encoders (outputting 256 dimensional vectors), an image encoder (Masked Autoencoder ViT outputting 256x64x64 feature map), and a Mask decoder similar to MaskFormer
  - Multiple (3) output tokens, to naturally allow for ambiguity. We only backprop on the one with the lowest loss.
  - Mask decoder block has cross-attention in both directions
  - Also predicts IoU scores, supervised by true IoU score
  - Text embedding can be used with clip embeddings
- Training procedure simulates interactive 11-round setups
  - First, either randomly select an initial point or bbox from the GT mask
  - Iteratively make predictions and adding new query point prompts based on the error region between prediction and GT; each new point is foreground or background if the error point is a false negative or false positive respectively. Additionally re-feed mask into network.
  - Only backprop after the 11-rounds.
- Comes with a 1.1B mask dataset
  - Collected with extensive data engine starting with assisted manual annotations
  - Labelers had simple instructions: to just label objects they could name/describe; did not collect names/descriptions
  - Continually bootstrapped SAM model to improve automation; final dataset is composed of all psuedolabels by prompting model with regular grid and filtering for confident masks.
  - All images come from photographers, but is able to zero-shot generalize decently to other domains / image styles
- SAM2 adds streaming memory-based transformer for video segmentation & occlusion prediction head; trained on new video segmentation dataset



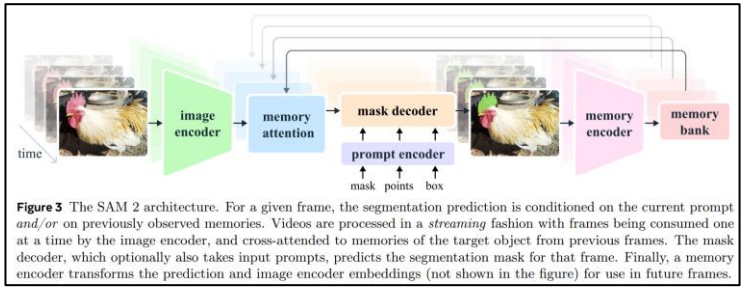
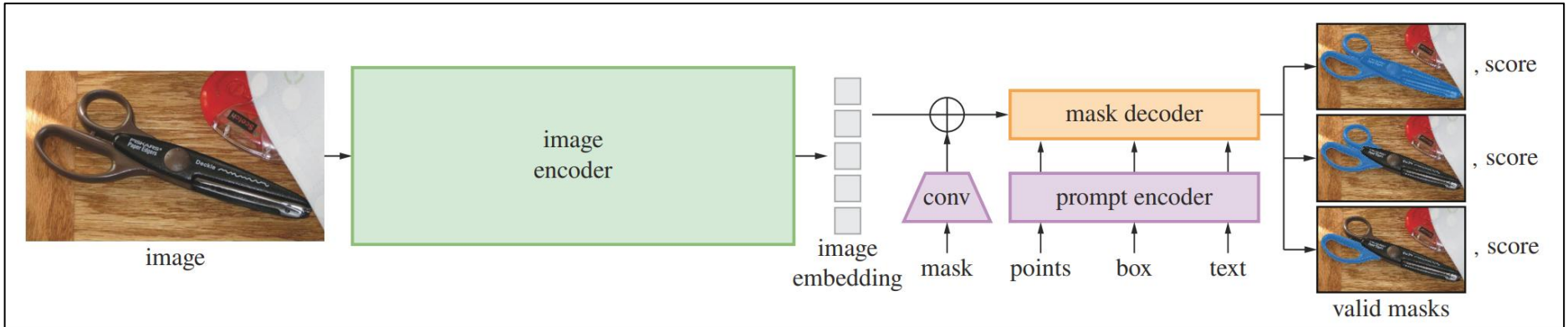


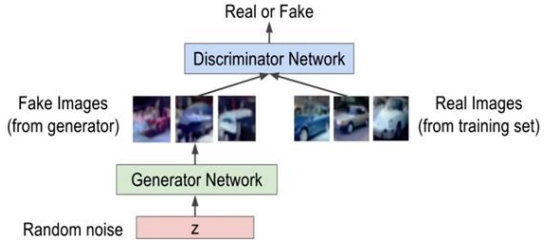
Figure 3 The SAM 2 architecture. For a given frame, the segmentation prediction is conditioned on the current prompt *and/or* on previously observed memories. Videos are processed in a *streaming* fashion with frames being consumed one at a time by the image encoder, and cross-attended to memories of the target object from previous frames. The mask decoder, which optionally also takes input prompts, predicts the segmentation mask for that frame. Finally, a memory encoder transforms the prediction and image encoder embeddings (not shown in the figure) for use in future frames.

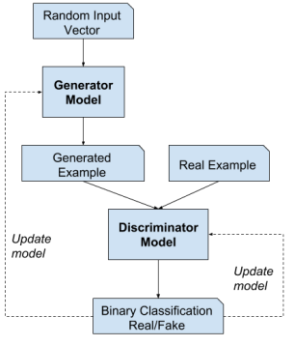


Generative Modeling

How do (W)GANs work, and how are they optimized? Can they be conditioned on?

GANs implicity model probability density by focusing on the ability to sample from an implicit distribution, in an unsupervised way. This is done through a two-player game between a generator and discriminator net. Convergence is reached at “nash equilibrium”, where which each player cannot reduce their cost without changing the other players’ parameters.





1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. **Instead: Gradient ascent** on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

The **Wasserstein GAN (WGAN)** is an alternative formulation which may have better theoretical properties.

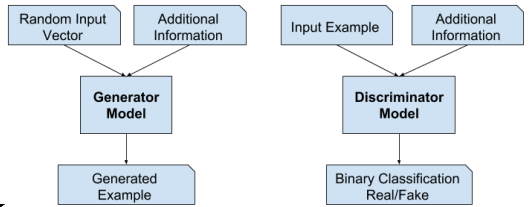
- Discriminator becomes a “critic”, which optimizes towards an approximation of the Wasserstein distance between the real and generated distribution.
- Unlike standard GANs, we want to train critic to convergence. In principle, WGAN loss should be informative (unlike standard gans losses, which have no immediate meaning for how training is progressing).
- The critic/generator are supposed to be K-Lipschitz, so gradient clipping or penalty is needed.
- The critic is optimized to maximize the distance between average real and generated outputs; this approximates the wasserstein loss (lines 2-8)
  - Critic outputs “realness score”; lower critic score= fake, higher critic score= real
- Then, the generator is trained to maximize the critic’s score

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

1: **while** θ has not converged **do**
2: **for** $t = 0, ..., n_{\text{critic}}$ **do**
3: Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
4: Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
5: $g_w \leftarrow \nabla_w \left[\frac{1}{m} \sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^{m} f_w(g_\theta(z^{(i)})) \right]$
6: $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
7: $w \leftarrow \text{clip}(w, -c, c)$
8: **end for**
9: Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
10: $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
11: $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
12: **end while**

(W)GANs can be conditioned (cGAN) on by providing both the generator and discriminator with some additional input.



Explain how Pix2Pix, CycleGAN, and conditional GANs work, and what they're useful for.

Both methods perform the task of image-to-image translation. Pix2Pix requires paired data, while CycleGAN does not.

Pix2Pix is a conditional GAN (cGAN); the discriminator is fed both domains to discriminate.

CycleGAN for domains X and Y, learns a mapping G:X ->Y and inverse mapping F:Y->X. The outputs of both have their own discriminator. It's enforced that cycles F(G(x))=x and g(F(y))=y hold.

Conditional GAN: both the generator and discriminator are conditioned (fed as input) with some additional information, like class labels or data from other modalities. Since discriminator now has to also ensure that input images belong to the correct class/criteria, these gradients guide the generator to create more specific images of the appropriate class/criteria

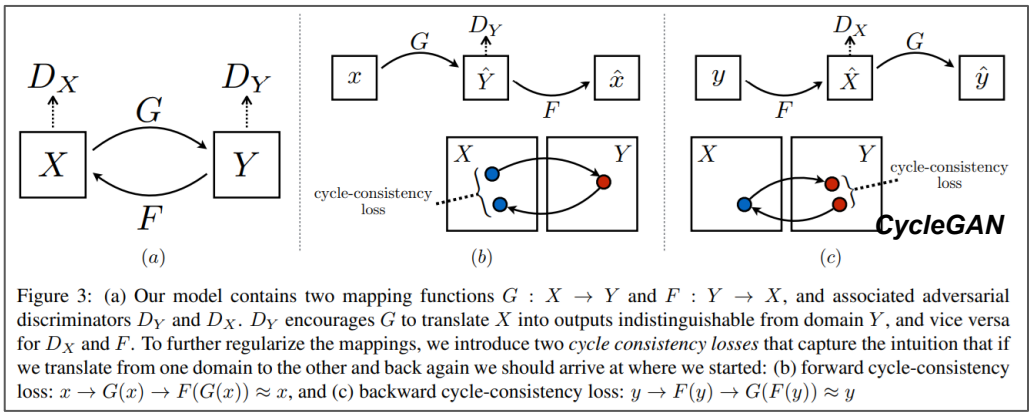
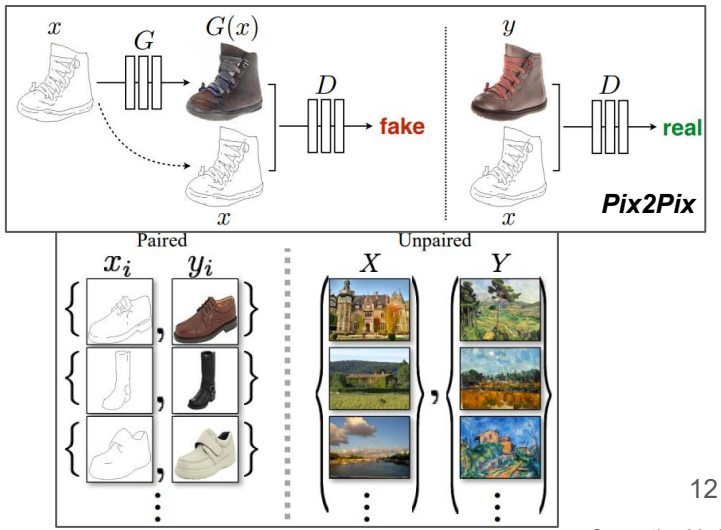
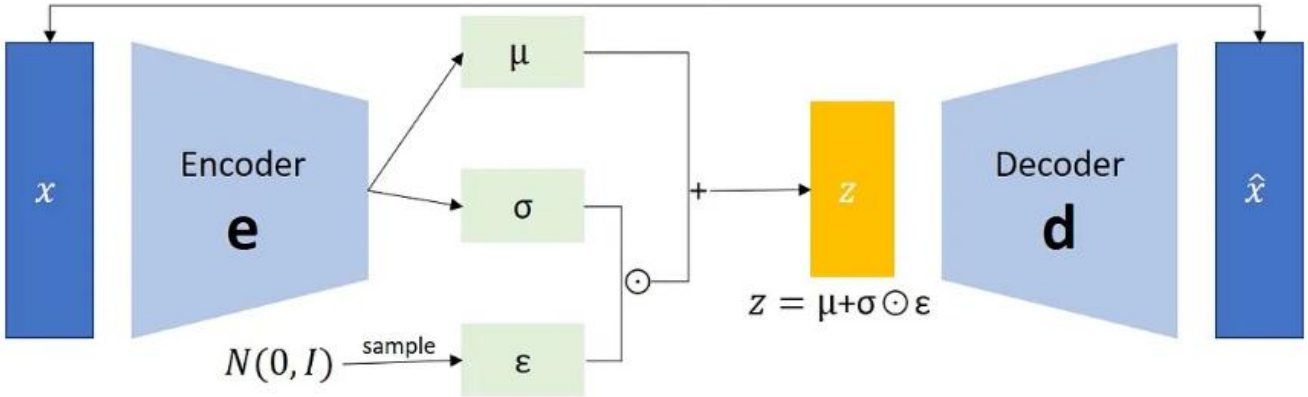


Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$



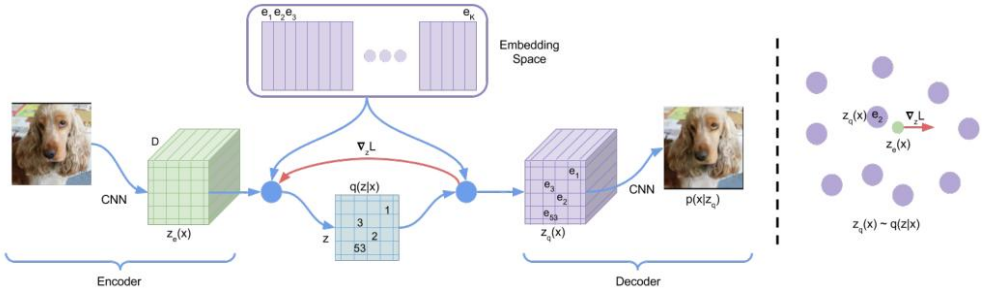
How do VAEs work?

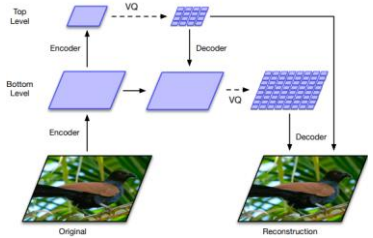
- Encoder $q_\phi(z|x)$ ie posterior, decoder $p_\theta(x|z)$, prior distribution $p(z) = N(0, I)$
- Encoder outputs parameters μ, σ of a gaussian distribution
- Loss involves reconstruction term and KL divergence to regularize to standard normal:
  $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)\|p(z))$
- Conducive to compact, & semantically organized latent space for stochastic sampling
- Reparameterization trick where $\epsilon \sim N(0, I)$ allows gradients to flow through μ, σ
- Can generate new samples from test time by sampling from $N(0, I)$ & passing through decoder
- Weaknesses: produces blurry images due to averaging nature of the reconstruction loss & normal regularization
  - In limit, can suffer to posterior collapse; encoder collapses to the prior distribution, causing latent variables to be uninformative and leading to poor diversity in generated samples



How does VQ-VAE work?

- Durning training an encoder maps the image to a continuous feature volume $z_e(x)$; each channel is then quantized to the nearest entry in a learned, finite-sized codebook to obtain $z_q(x)$; this is then decoded to reconstruct the original image.
  - The argmin “snapping” operator here is non-differentiable, but in practice everything seems to work fine if you just approximate & pass the decoder gradient directly through to the encoder
- Loss function contains 1) reconstruction loss, 2) “codebook alignment loss” to incentive learned codebook embeddings to be close to the encoder output, and 3) “codebook commitment loss” to encourage encoder outputs to be close to codebook embeddings
  - $\log(p(x|q(x))) + ||\text{sg}[z_e(x)] - e||_2^2 + \beta||z_e(x) - \text{sg}[e]||_2^2$
  - Here, sg stands for “stop gradient” and prevents the gradient from flowing through that part
- The use of discrete latents facilitates the use of autoregressive token-by-token latent priors for speech/audio/images
- At test time, we can learn a new prior $p(z)$ by 1) taking a dataset of images, 2) obtaining their corresponding sequences of codebook tokens with the encoder & codebook, 3) training an autoregressive generative model (eg pixelCNN or transformer) model on those sequences, 4) generating new novel sequences, and 4) feeding the sequences into the decoder for image generation
  - In this paradigm, intuitively pixelCNN would be the “creative director” generating a sequence of semantically meaningful latents and the decoder would carry out the work of realizing those ideas in pixel space
- Some later works improve this by 1) using vision transformers (ViT) for the encoder/decoder, 2) using multiple scales w/ corresponding different scale codebooks, 3) adding styleGAN discriminator losses, 4) adding perceptual loss





How does DALL-E 1 implement text to image generation?

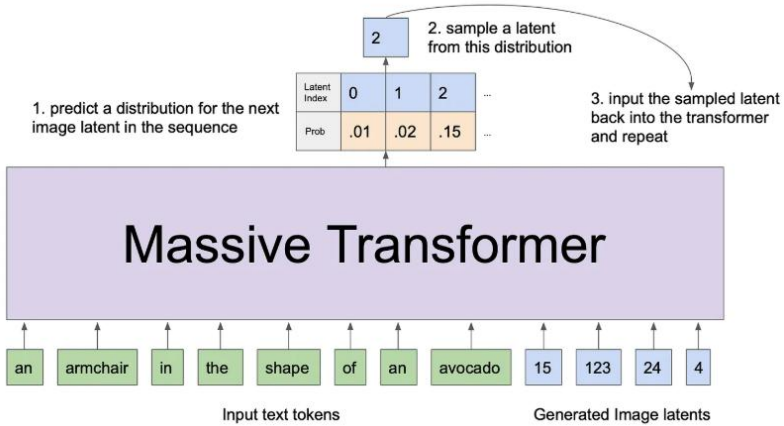
- The first step involves using a Discrete VAE (dVAE), which is similar to VQ-VAEs, but instead of the encoder trying to match codebook values, it outputs a distribution over codebook vectors for each latent, which is more expressive. (Note, this is not a fundamental change, and in principle DALL-E could have used vanilla VQ-VAEs.)
- To backpropogate through sampling from this categorical distribution, it uses a gumbel softmax relaxation:
  - First, we use a "gumbel" reparameterization trick to allow us to disentangle nn parameters from the source of randomness when categorically sampling.
    - It can be shown that if g_i are iid samples from the Gumbel distribution, then the following allows for random categorical sampling, where $q(e_i|x)$ is the probability prediction of the e_i^{th} codebook vector

$$z = \text{codebook}[\text{argmax}_i[g_i + log(q(e_i|x))]]$$

  - Second, we use a softmax relaxation with a temperature τ that approaches 0 throughout training, instead of argmax. This allows us to "softly" choose the max via weighted sum in a backpropagatable way
    - When τ is large, this resembles a uniform distribution
    - When $\tau = 1$, this is a softmax over the categorical distribution
    - As $\tau \to 0$, the differences become exaggerated and it approaches a max

$$y_i = \frac{e^{\frac{g_i + log(q(e_i|x))}{\tau}}}{\sum_{j=1}^{k} e^{\frac{g_j + log(q(e_j|x))}{\tau}}}$$

- After the dVAE is trained, we can train a transformer to predict sequences of text, followed by image latents (which we can subsequently look up in codebook & feed to the dVAE decoder)
  - DALL-E was trained on 250 million text-image pairs scraped from the internet



How does a Masked Generative Image Transformer (MaskGIT) work?

- MaskGIT uses VQVAE-style tokenization, but modifies the token generation from sequential AR to an iterative parallel approach by using a bidirectional transformer decoder similar to BERT.
- During training, we learn to predict randomly masked tokens by attending to tokens in all directions
- During inference, we iteratively predict all tokens simultaneously, but only keep the most confident ones; remaining tokens are masked out and re-predicted in the next iteration.
  - In principle the model is able to infer all tokens and generate the entire image in a single pass, but this doesn't work well due to the training task's formulation
- Shown to also natively work well for zero-shot tasks such as class-conditioned image manipulation, inpainting, and outpainting
- Much faster, intuitively like solving a jigsaw puzzle: predicting masked parts of an image in parallel, guided by confidence and contextual understanding

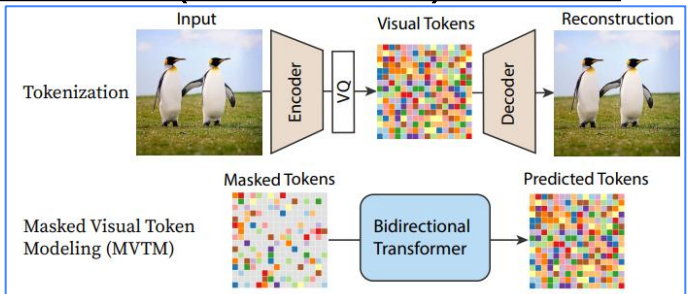
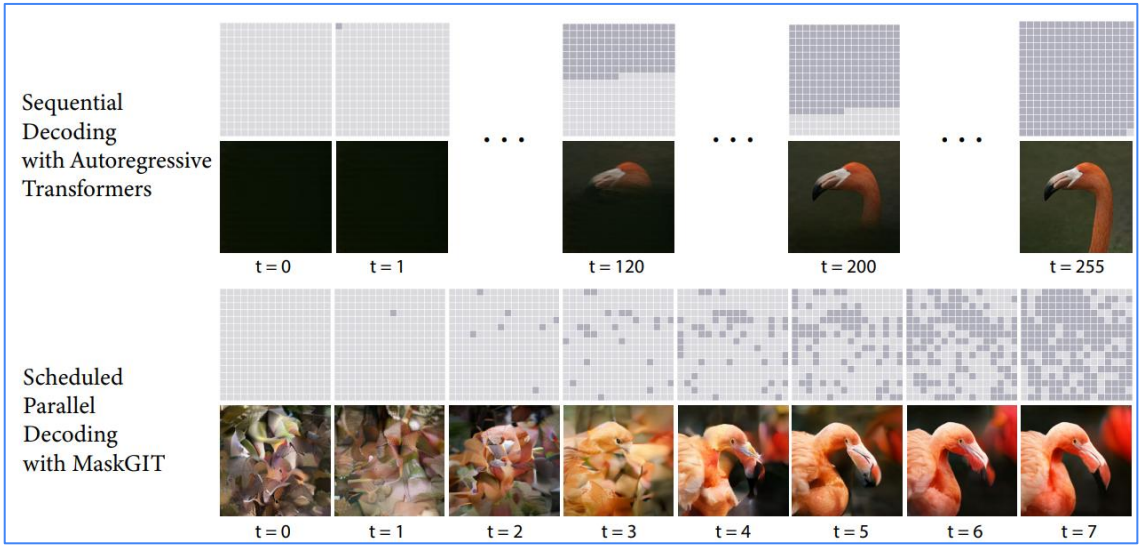
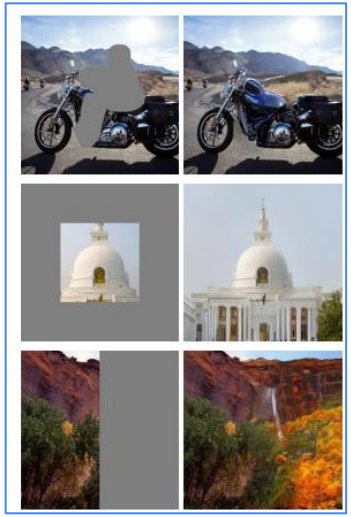
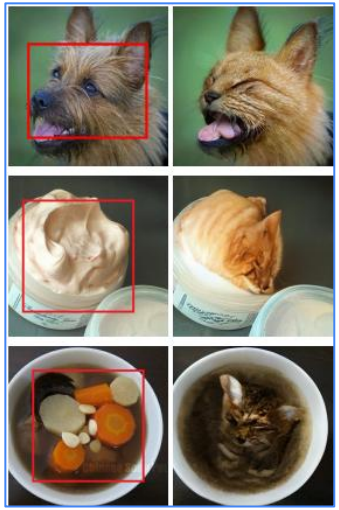


Figure 3. **Pipeline Overview.** MaskGIT follows a two-stage design, with 1) a tokenizer that tokenizes images into visual tokens, and 2) a bidirectional tranformer model that performs MVTM, i.e. learns to predict visual tokens masked at random.







Explain how Visual AutoRegressive (VAR) models work.

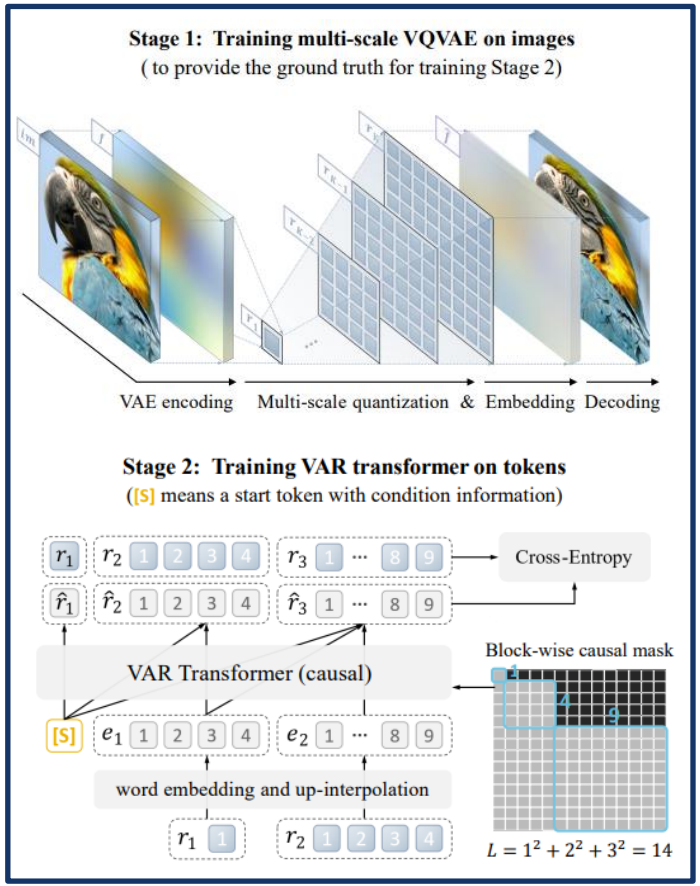
- Motivation: VQVAE & similar works perform flattening & process tokens unidirectionally (eg raster-scan or spiral), even though on images there are bidirectional spatial correlations
- VAR reformulates the task by shifting from “next token” to “next scale” prediction; autoregressive unit is an entire token map, instead of just a single token. Leads to SoTA performance and speed
- Multi-scale VQVAE Encoding (algorithm 1 below):
  - Start with an initial feature map f. For K iterations, we successively go from low resolution to high resolution, storing quantized feature map interpolations (ie token maps) & residually updating f by subtracting the codebook vectors interpolated to the original size (ϕ_k conv layers follow the interpolation)
- Multi-scale VQVAE Decoding (algorithm 2 below):
  - Given the encoded list of multi-scale token maps, sum them all by converting them to codebook vectors and interpolating them to the original size. Then, pass that feature volume to the decoder to reconstruct
- Compound loss used, with reconstruction loss + codebook quantization error loss + perceptual loss + styleGAN discriminator loss
- Standard decoder-only transformer used for generation, where in the k^{th} autoregressive step we treat the sequence of tokens $(r_1, \dots r_{k-1})$ as the conditional prefix and generate $h_k \times w_k$ tokens in

Algorithm 1: Multi-scale VQVAE Encoding

Inputs: raw image im;
Hyperparameters: steps K, resolutions $(h_k, w_k)_{k=1}^{K}$;
$f = \mathcal{E}(im)$, $R = []$;
for $k = 1, \cdots, K$ **do**
  $r_k = \mathcal{Q}(\text{interpolate}(f, h_k, w_k))$;
  $R = \text{queue\_push}(R, r_k)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $f = f - \phi_k(z_k)$;
Return: multi-scale tokens R;

Algorithm 2: Multi-scale VQVAE Reconstruction

Inputs: multi-scale token maps R;
Hyperparameters: steps K, resolutions $(h_k, w_k)_{k=1}^{K}$;
$\hat{f} = 0$;
for $k = 1, \cdots, K$ **do**
  $r_k = \text{queue\_pop}(R)$;
  $z_k = \text{lookup}(Z, r_k)$;
  $z_k = \text{interpolate}(z_k, h_K, w_K)$;
  $\hat{f} = \hat{f} + \phi_k(z_k)$;
$\hat{im} = \mathcal{D}(\hat{f})$;
Return: reconstructed image $\hat{im}$;



How does diffusion work?

Forward Diffusion:

$0 < \beta_1 < \beta_2 < \ldots < \beta_T < 1$ ***(Known variance schedule)***

$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$

$\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ ***(Variance towards 1 while x towards 0; approaches unit normal)***

$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$ ***(shortcut)***

$\alpha_t := 1 - \beta_t \qquad \bar{\alpha}_t := \Pi_{s=1}^{t}\alpha_s$

Reverse Diffusion (Denoising) via Bayes Rule:

$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t\mathbf{I})$

$$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} * \beta_t$$

(Derived result; can show that the reverse itself is close to Gaussian as well)

$$\tilde{\mu}_t = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_t\right)$$

Objective Function Formulation:

$\|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|^2 = \|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon, t)\|^2 \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

- *Take batch of images*
- *Forward diffuse to randomly sampled noisy timesteps*
- *NN predicts the added gaussian noise relative to their respective previous timestep*
- *NN recieves a mixture of image & noise; it's its job to understand the distribution of images and be able to separate out the noise*

Test-Time Sampling:

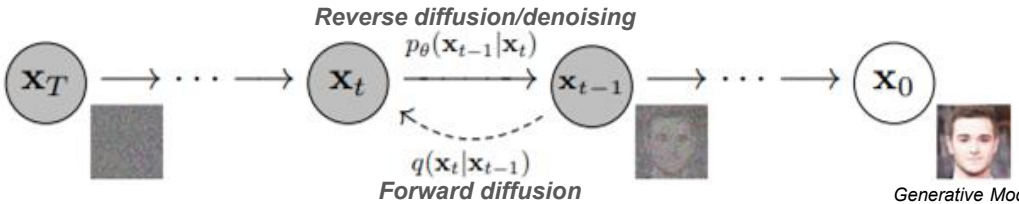
Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \ldots, 1$ **do**
3: $\quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\quad \mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\mathbf{z}_\theta(\mathbf{x}_t, t)\right) + \sigma_t\mathbf{z} \quad (\sigma_t = \tilde{\beta}_t)$
5: **end for**
6: **return** $\mathbf{x}_0$

- *Note: mathematically, in principle we can directly jump from x_T to x_0 based on the objective function. But this isn't done in practice b/c the network is trained to work on many timesteps, as it empirically outperforms.*

Other Details:

- Original arch is U-Net w/ transformer layers & groupnorm
- t encoded as sinusoidal positional embedding & added to feature maps multiple times throughout, where position embeddings dimensions = # of channels
- One of the keys to making this work is the concept of iterative refinement: rather than generating a sample in a single pass through a neural network, the model is applied repeatedly to refine a canvas, and hence the unrolled sampling procedure corresponds to a much "deeper" computation graph (similar to LLMs). The paradigm is train-efficient since we don't need to backprop through everything; generative process not trained end-to-end



Explain how diffusion models can be conditioned, with and without a classifier.

Classifier Guidance:

- If you train a classifier $p_\phi(y|x_t)$ on noisy images, we can use its gradients to guide the diffusion sampling process towards a class-semantic correct image
- Classifier tells us which direction in image space increases probability of the class, and we follow that direction during generation
- No need to retrain the diffusion model, but training classifier to be accurate on noisy images can be challenging

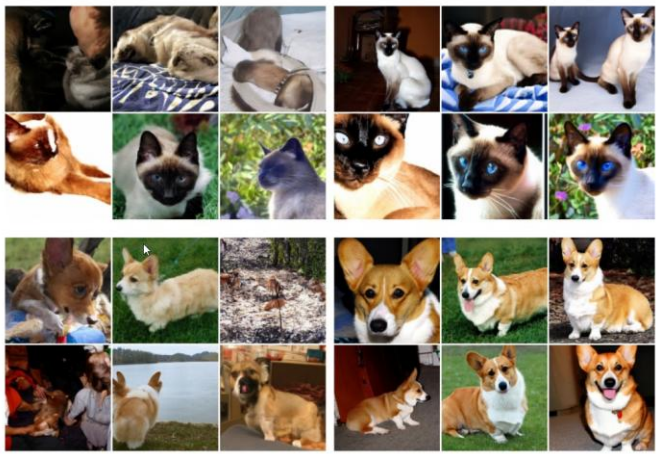
Classifier-Free Guidance

- Involves training a diffusion model to be exposed to both conditioned & unconditioned inputs (via null labels)
- Then, at inference use a linear combination of conditional and unconditional score estimates (w > 1)
- Works better than classifier guidance b/c we've constructed the "classifier" from a generative model, which provides much more robust gradients

$$\tilde{\epsilon}_\theta(\mathbf{z}_\lambda, \mathbf{c}) = (1+w)\epsilon_\theta(\mathbf{z}_\lambda, \mathbf{c}) - w\epsilon_\theta(\mathbf{z}_\lambda)$$

Algorithm 1 Classifier guided diffusion sampling, given a diffusion model $(\mu_\theta(x_t), \Sigma_\theta(x_t))$, classifier $p_\phi(y|x_t)$, and gradient scale s.

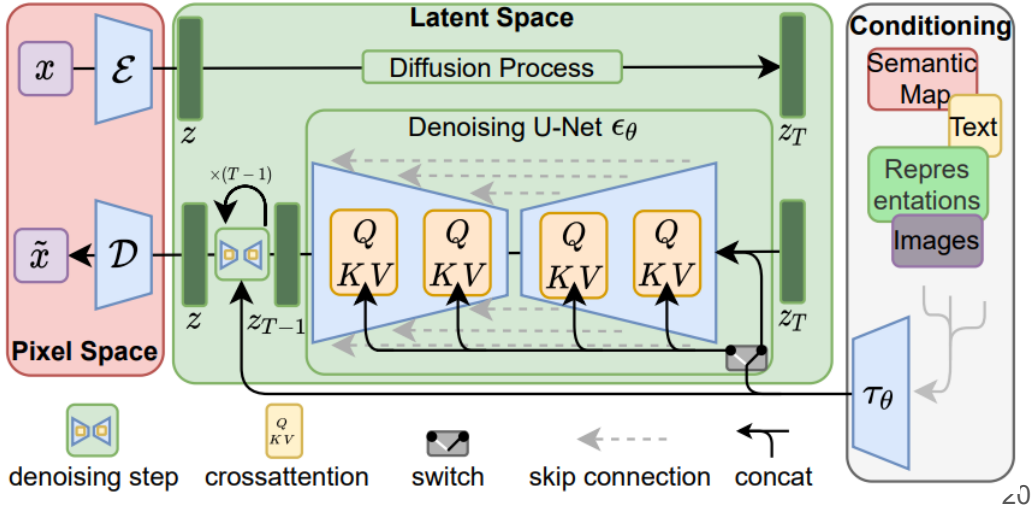
Input: class label y, gradient scale s
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
for all t from T to 1 **do**
  $\mu, \Sigma \leftarrow \mu_\theta(x_t), \Sigma_\theta(x_t)$
  $x_{t-1} \leftarrow$ sample from $\mathcal{N}(\mu + s\Sigma \nabla_{x_t} \log p_\phi(y|x_t), \Sigma)$
end for
return x_0



Unconditional (left) vs conditional (right)

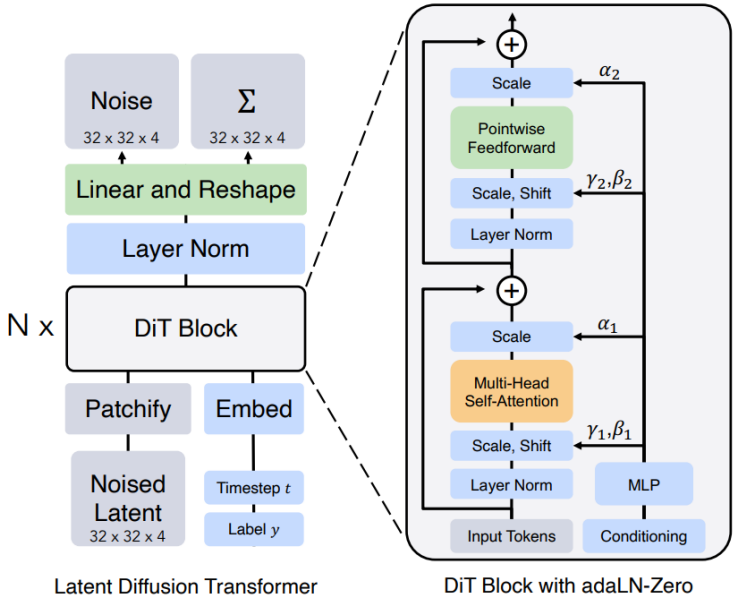
How can diffusion be done in the latent space?

- Two stage recipe:
  - First, train autoencoder on input signals
  - Second, train generative diffusion model on latent representations. In this stage, encoder parameters are frozen (and we don't need the decoder until inference)
- Intuition: by extracting a more compact representation that focuses on the perceptually relevant fraction of signal content, and modelling that instead of the original representation, we are able to make relatively modestly sized generative models punch above their weight.



How do Diffusion Transformers Work?

- Transformer-based diffusion model operating on latent representations
  - Latents are from a frozen pretrained VAE encoder, which is generally convolution / U-Net based
- Uses AdaLN-Zero for conditioning, which was found to perform better than cross-attention and in-context tokens
  - Embeds timestamp and class label with separate MLPs into e_t, e_y, then adds to get e
  - Layernorm residual scaled with separate MLPs: $h_{out} = h + \alpha(e) * Block\big(\gamma(e) * LayerNorm(h) + \beta(e)\big)$



How can diffusion sampling be made deterministic?

- Forward process (noising) is the same as before
- Reverse process involves first obtaining $\widehat{x_0}$, a "one-shot" estimate of x_0
- Then, the update is a weighted combination of the clean image $\widehat{x_0}$ and the noise direction
- Allows for a deterministic mapping from noise -> image, and image back to noise

$$\hat{x}_0 = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \sqrt{1 - \alpha_t}\, \epsilon_\theta(x_t, t) \right)$$

$$x_{t-1} = \sqrt{\alpha_{t-1}} \cdot \hat{x}_0 + \sqrt{1 - \alpha_{t-1}} \cdot \epsilon_\theta(x_t, t)$$

How can diffusion be sped up using distillation?

- Train a standard teacher DDIM model
- Perform inference on a dataset of images, and cache some intermediate partially-denoised results
- Train student to directly map those intermediate results

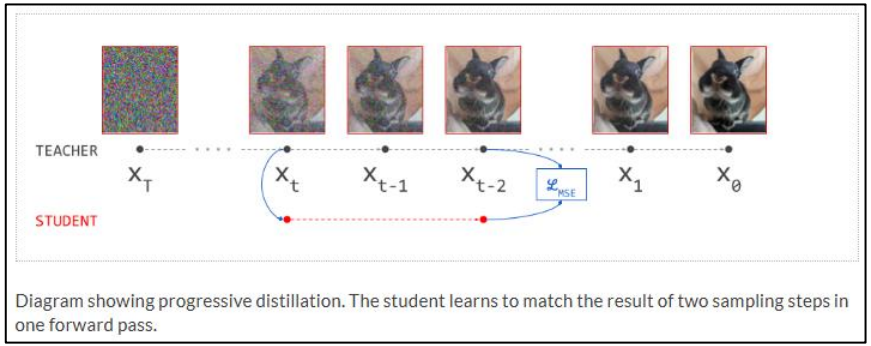


Diagram showing progressive distillation. The student learns to match the result of two sampling steps in one forward pass.

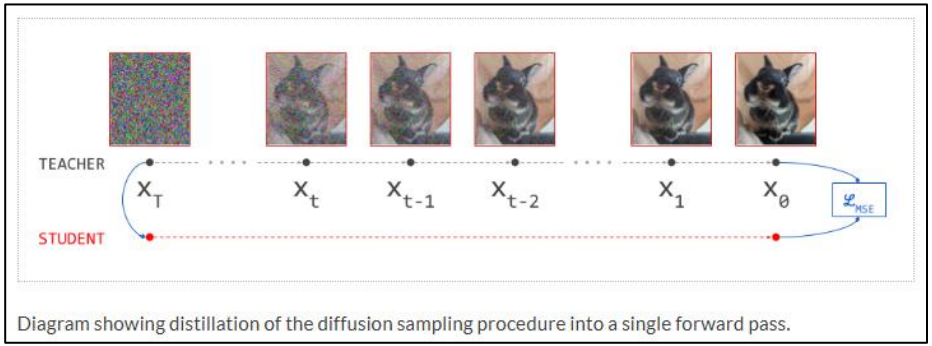
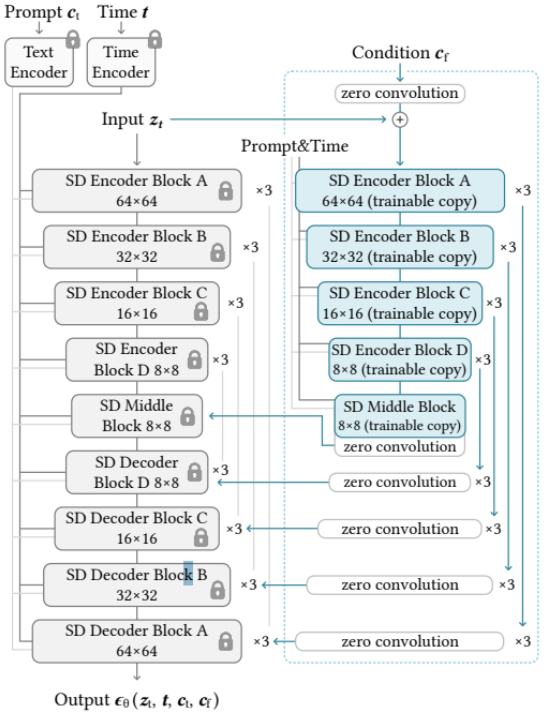


Diagram showing distillation of the diffusion sampling procedure into a single forward pass.

What is ControlNet?

- Method to enable precise spatial control/conditioning in large text-to-image diffusion models, without modifying the original pretrained model
  - Inputs include edge maps, human poses, depth maps, segmentation maps, sketches, etc
- Freezes pretrained model weights and connects it to zero-initialized 1x1 convolutions (so that it avoids injecting noise too early, and promote gradually learning how to modulate network)
- Requires paired data with conditioning, and optionally paired text captions

Figure 1: Controlling Stable Diffusion with learned conditions. ControlNet allows users to add conditions like Canny edges (top), human pose (bottom), *etc*., to control the image generation of large pretrained diffusion models. The default results use the prompt "a high-quality, detailed, and professional image". Users can optionally give prompts like the "chef in kitchen".



(a) Stable Diffusion (b) ControlNet

Figure 3: Stable Diffusion's U-net architecture connected with a ControlNet on the encoder blocks and middle block. The locked, gray blocks show the structure of Stable Diffusion V1.5 (or V2.1, as they use the same U-net architecture). The trainable blue blocks and the white zero convolution layers are added to build a ControlNet.

How can diffusion be used to do inpainting?

- Can be done without retraining the diffusion model; during inference timesteps, noise known (unmasked) regions to the appropriate corresponding noise level, while unknown (masked) regions are initialized with random noise and proceed as usual
  - Has additional benefit to not restrict training to a specific distribution of mask, which can limit generalization
  - The RePaint paper additionally applies strategy to include periodic jumps forward/back in diffusion steps to refine results
- Alternatively, you can train the model to directly do inpainting by training on masked images
  - Network trained to see partially noised images, in the masked regions

$$x_t = M \odot x_{\text{noisy}} + (1 - M) \odot x_0$$

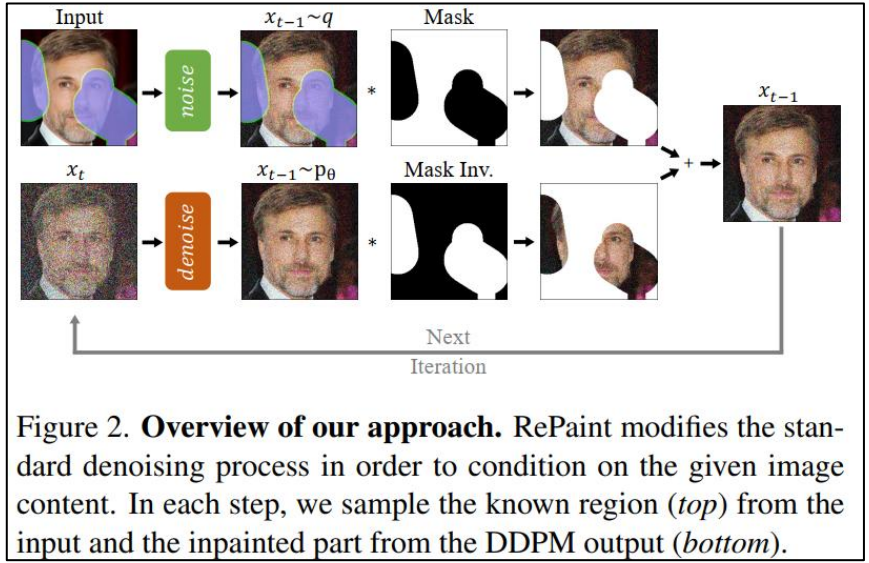
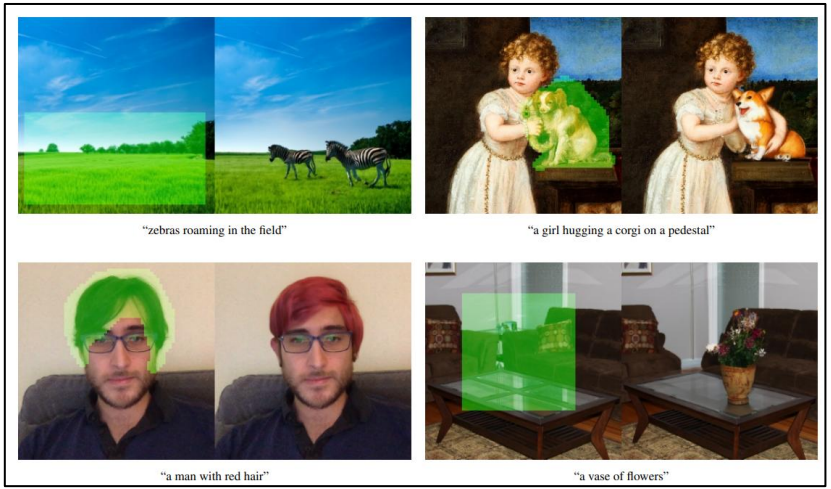


Figure 2. **Overview of our approach.** RePaint modifies the standard denoising process in order to condition on the given image content. In each step, we sample the known region (*top*) from the input and the inpainted part from the DDPM output (*bottom*).

What is GLIDE and how can it perform text-guided inpainting?

- Text-guided diffusion (using classifier-free guidance); outperforms CLIP-guided diffusion where you train a clip model on noisy images
- Can do text-guided inpainting by finetuning model to inpaint with text conditioning
  - Masked regions don't necessarily correspond to text captions during training – suboptimal, but in practice still seems to work
  - The model is able to see a fair amount of examples where the masked region does correspond, and over time learns to provide desired behavior



What is cascaded diffusion?

- Framework for generating high-resolution images using a cascade of separate diffusion models
  - Consists of a base diffusion model that generates a low res image (eg 32 x 32), then several superresolution models that upsample to higher resolutions
  - Models not trained end-to-end; they're separate models
- Superresolution model turns high-res gaussian noise into a high-res image, conditioned on a low-resolution image
- Key trick is augmentation to superresolution model to minimize train-test mismatch (train is natural images, test is high-res images; different distributions)
  - Variants include truncated diffusion results (adds noise), or gaussian noise/blur to bridge the gap

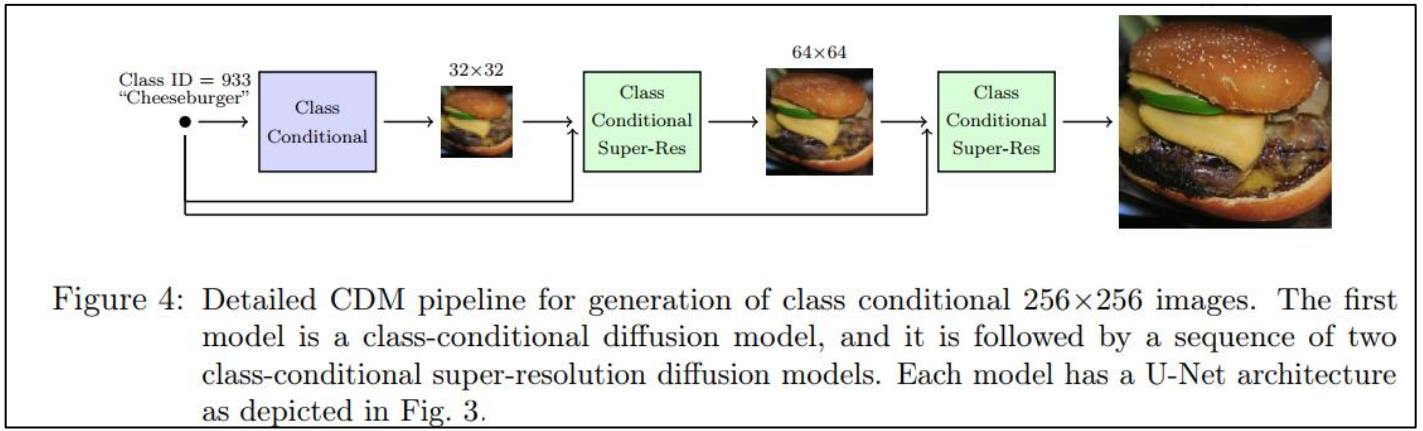


Figure 4: Detailed CDM pipeline for generation of class conditional 256×256 images. The first model is a class-conditional diffusion model, and it is followed by a sequence of two class-conditional super-resolution diffusion models. Each model has a U-Net architecture as depicted in Fig. 3.

How does DreamBooth work?

- Tackles subject-driven generation: method to personalize large text-to-image diffusion models so they can generate novel, high-fidelity images of a specific subject (like a pet or object) using only 3–5 reference images
- Fine-tunes a pretrained diffusion model to bind a **unique textual token** to a specific subject instance (eg a particular dog). Enables model to recognize and regenerate the object, placed in various contexts
- Method:
  - Finetune model using few images, paired with prompts like “a [V] dog”, where [V] is a rare token. The prompt should include the class so the model can leverage the class’s prior
  - It’s also important to have a “class-specific prior preservation loss”, where we have the model continue training on general images that involve the class of the specific subject. This significantly reduces overfitting; notably, authors found that training in generated images (ie, psuedolabels) works best. This encourages model to continue to generate diverse samples in the subject’s class
- Early stopping also used to mitigate overfitting; training is short, with a small learning rate

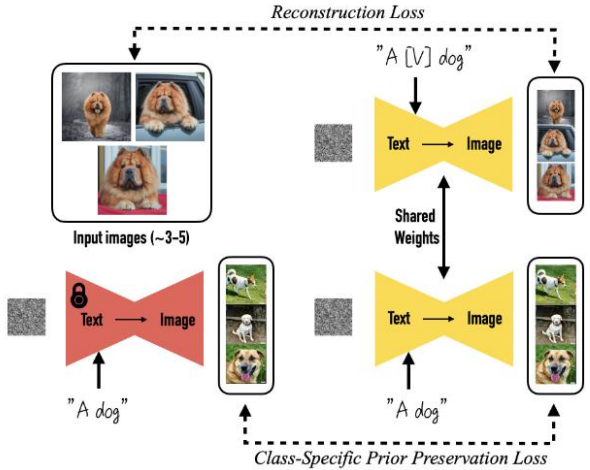
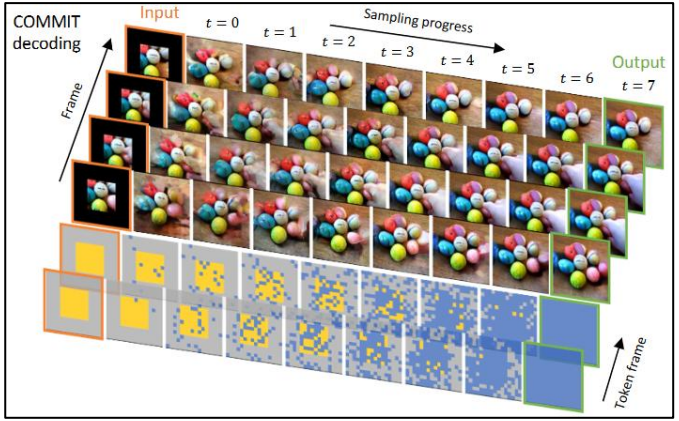


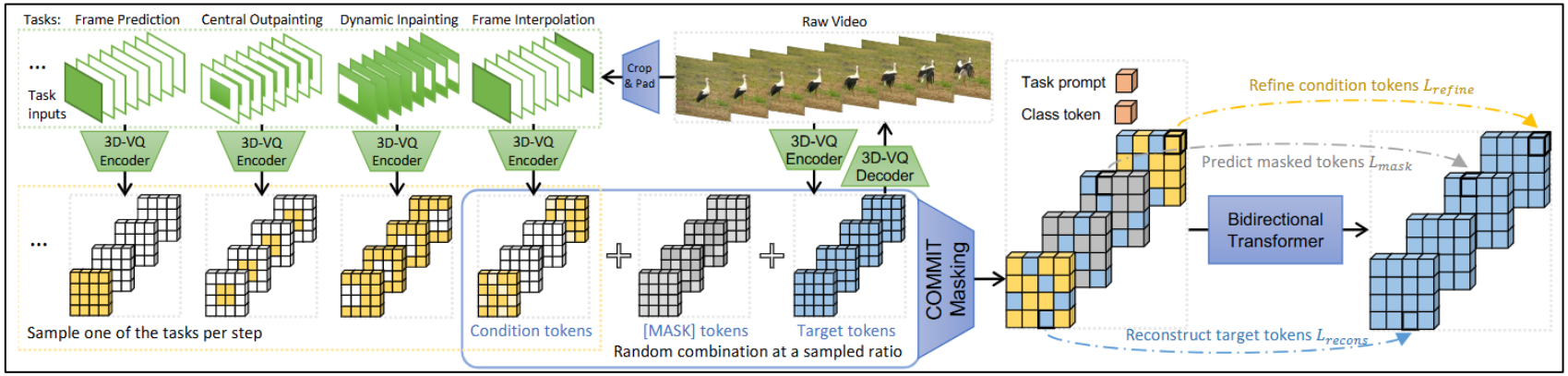
Figure 3. **Fine-tuning.** Given $\sim 3-5$ images of a subject we fine-tune a text-to-image diffusion model with the input images paired with a text prompt containing a unique identifier and the name of the class the subject belongs to (e.g., “A [V] dog”), in parallel, we apply a class-specific prior preservation loss, which leverages the semantic prior that the model has on the class and encourages it to generate diverse instances belong to the subject’s class using the class name in a text prompt (e.g., “A dog”).

How does MAGVIT work?

- First, trains 3D VQ-GAN using 3D conv based architecture
  - Encodes videos into codebook, enabling space-time compression (quantizes 16 frame, 128x218 video into 4x16x16 tokens, where each token is a spatiotemporal supervoxel)
  - Results in a 3D-VQ encoder to obtain video tokens, and a 3D-VQ decoder to convert tokens back to pixel space
- Second, trains a BERT-like bidirectional transformer to do masked token modeling
  - Learns to predict masked tokens based on context, class, and task specific prompts
  - Input to BERT composed of three parts, all with the same reconstruction loss.
    - **Condition Tokens**: Partially masked video is fed into 3D-VQ encoder to obtain tokens, and these are propagated to BERT. Allows us to model the train-test mismatch, since the encoder now has to deal with sparse video inputs as conditioning. Intuitively, refining known condition tokens ensures that it can process noisy/incomplete inputs well.
    - **Mask Tokens**: Want BERT to predict these regions.
    - **Target Tokens**: We feed some target tokens; by feeding in GT this helps ensure all tokens are well modeled.
  - Tasks includes frame prediction, interpolation, inpainting, etc

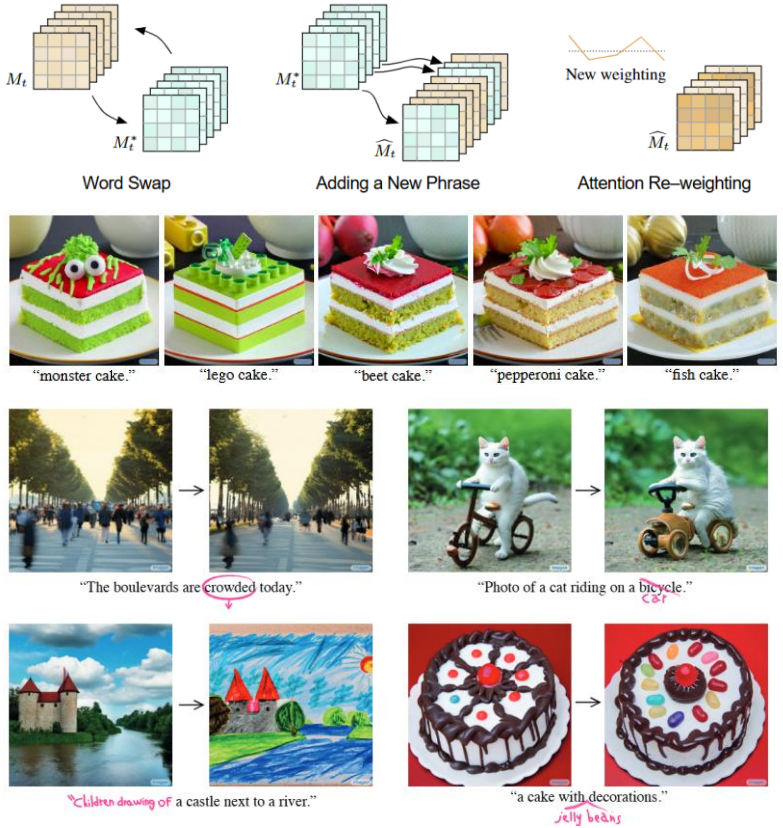


- Decoding/inference is done non-autoregressively
  - we start with token sequence with condition tokens embedded and others masked. Then, iteratively: predict masked tokens, replace a fraction of them based on high confidence.
  - Entire video generated in the first iteration, and then in subsequent iterations we continue to refine every frame in the video simultaneously in parallel.
- Note: no guarantee that conditioned pixels will remain the same after generation, but this can be a feature; allows liberty for consistency and flexibility



How does prompt-to-prompt editing work for diffusion models?

- Allows fine-grained, targeted editing of images generated by text-to-image diffusion models by using only textual prompts; no masks
  - Crucially, properly maintains the image structure and content of unchanged words
- Based on the Imagen diffusion architecture, which is U-Shaped and conditions on entire sequences of text embeddings via cross attention
- Does not require any retraining; an inference-only modification that modifies/injects the cross-attention map. First seed is fixed, then procedure is to at each diffusion timestep:
  1. Run denoising through original prompt, and save cross-attention map
  2. Run denoising through target prompt, and save cross-attention map
  3. Composite a new cross attention map that preserves the structure of the original prompt, but selectively changes it using the target cross-attention map
  4. Run denoising with the composite cross-attention map, and use this for the next iteration
- Tasks enabled:
  - **Word Swap**: Replaces a word in the prompt
  - **Adding New Phrase**: Append tokens to the prompt
  - **Attention Re-Weighing**: Scale influence of a token



How does DALL-E 2 work?

- **First, we train a CLIP model on noisy (diffusion gaussian noise) images with their text pairs**
  - This helps distill images into their base semantic meaning
  - This multimodal embedding space also allows use to do interpolation & vector arithmetic
- **Second, we train a "diffusion prior model": predict CLIP image embedding given CLIP text embeddings**
  - This is a decoder-only transformer, that operates with a casual attention mask on the tokenized text, CLIP text embedding, diffusion timestep, and noised image CLIP embedding
  - Intuitively, the image embedding can be more appropriate compared to directly using the text embedding, because the text/image "shared" embedding space is not perfectly aligned
- **Third, train a classifier-free conditional U-Net diffusion model using the diffusion prior**
- At inference, given a text prompt we embed it, predict the image embedding, and then decode using diffusion
- Can do many things with this setup:
  - Generate images from text
  - Encode an image, then decode with diffusion to generate variations; 2 image embeddings can be interpolated as well
  - Encode 2 text samples and diffuse their interpolation

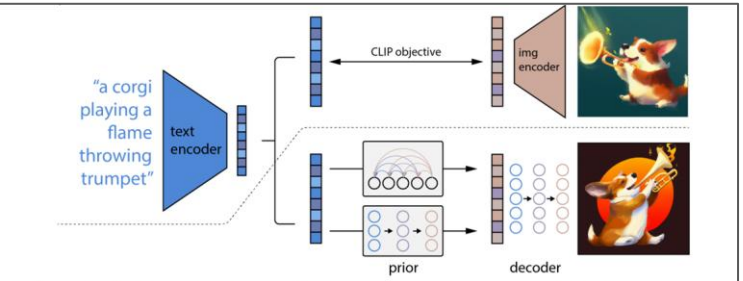


Figure 2: A high-level overview of unCLIP. Above the dotted line, we depict the CLIP training process, through which we learn a joint representation space for text and images. Below the dotted line, we depict our text-to-image generation process: a CLIP text embedding is first fed to an autoregressive or diffusion prior to produce an image embedding, and then this embedding is used to condition a diffusion decoder which produces a final image. Note that the CLIP model is frozen during training of the prior and decoder.

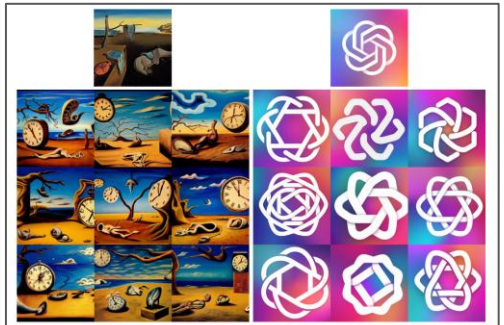


Figure 3: Variations of an input image by encoding with CLIP and then decoding with a diffusion model. The variations preserve both semantic information like presence of a clock in the painting and the overlapping strokes in the logo, as well as stylistic elements like the surrealism in the painting and the color gradients in the logo, while varying the non-essential details.

a photo of a landscape in winter → a photo of a landscape in fall

How does Lumiere work?

- Lumiere is a space-time aware diffusion model that generates entire video clips in a single pass (and then iteratively refines it)
- Uses a Space-Time U-Net for diffusion
- Loads weights from a pretrained Imagen network (U-net based diffusion model trained for images; conditions on text using cross-attention & does classifier-free guidance)
  - We “inflate” the imagen weights by adapting them to video & incorporating temporal components
  - We add 1D convolutions applied across the time dimension; spatial attention becomes spatiotemporal attention by attending temporally
  - The weights of the pretrained network are fixed; only added temporal components are trained.
- Additionally can do image-to-video (by conditioning on the first frame) or text-to-video (with text conditioning)

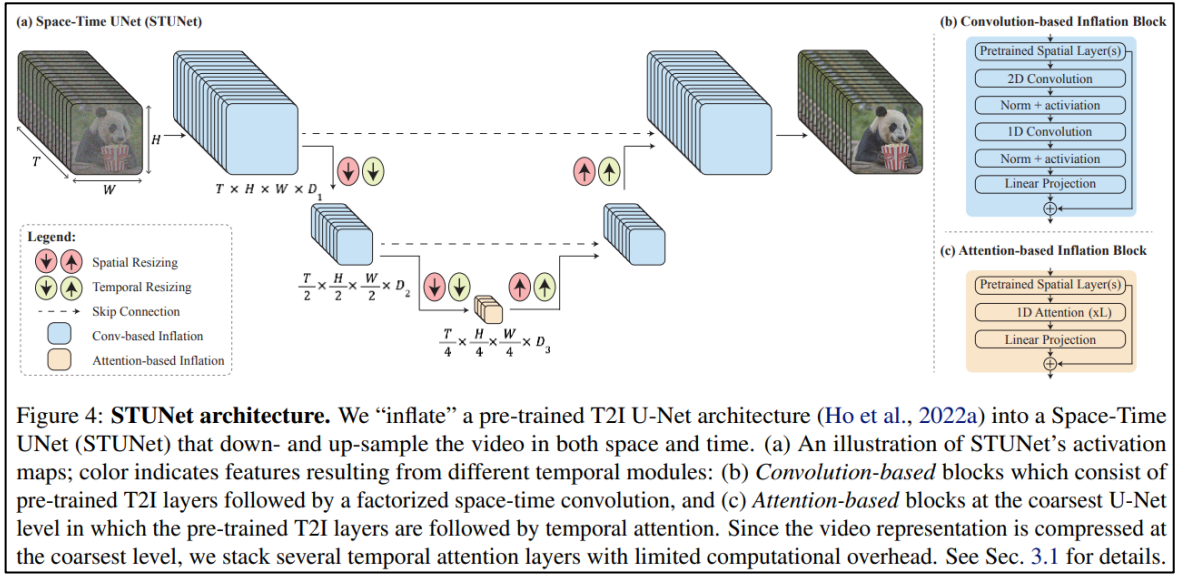


Figure 4: **STUNet architecture.** We “inflate” a pre-trained T2I U-Net architecture (Ho et al., 2022a) into a Space-Time UNet (STUNet) that down- and up-sample the video in both space and time. (a) An illustration of STUNet’s activation maps; color indicates features resulting from different temporal modules: (b) *Convolution-based* blocks which consist of pre-trained T2I layers followed by a factorized space-time convolution, and (c) *Attention-based* blocks at the coarsest U-Net level in which the pre-trained T2I layers are followed by temporal attention. Since the video representation is compressed at the coarsest level, we stack several temporal attention layers with limited computational overhead. See Sec. 3.1 for details.

How does Zero123 work?

- Novel view synthesis by finetuning a pretrained Stable Diffusion model to take in camera pose conditioning
  - Stable diffusion is just a latent-space diffusion method with a U-Net architecture
- Trained on synthetic objaverse dataset (10M 3D objects)
- Able to generalize well on real images, due to the internet-scale trained stable diffusion training, as well as high-quality & diverse synthetic data
- Zero123++ improves method further via tiling and other tricks
  - Tiling involves generating 6 predefined views of an object simultaneously, all within one image to improve 3D consistency

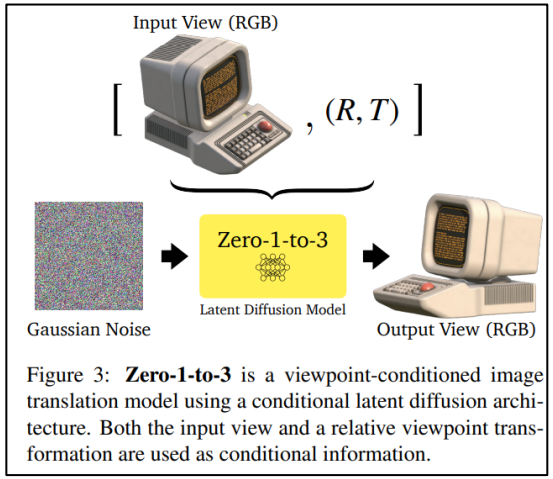


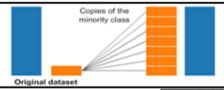
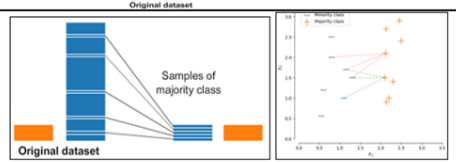
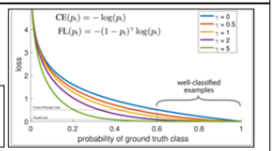
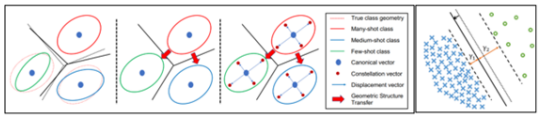
Figure 3: **Zero-1-to-3** is a viewpoint-conditioned image translation model using a conditional latent diffusion architecture. Both the input view and a relative viewpoint transformation are used as conditional information.

At a high level, what are the trade-offs between diffusion vs VQ for generation?

- Diffusion models (eg Stable Diffusion, Imagen):
  - are generally more detailed/photorealistic
  - slower & need to take many steps
  - Very good editability & controllability
- VQ based models (eg DALLE-1, VQ-VQE, VQGAN, MAGVIT):
  - Faster
  - Very naturally fits LLM-framework / transformers

Data Imbalance

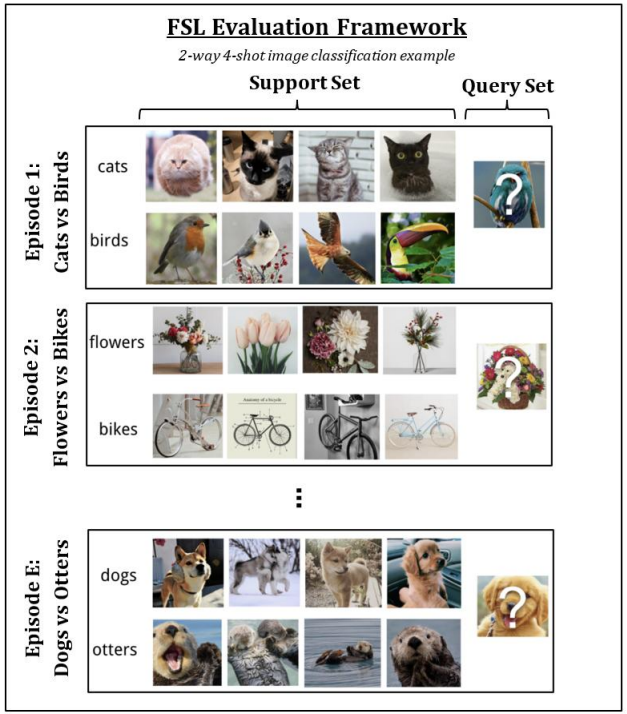
What are the methods to deal with data imbalance?

Summary of Methods to Improve Learning with Data Imbalance			
Name	**Description**	**Remarks**	**Figure**
Data Collection	• Collect more data to improve class balance	Best performance, but costs the most.	
Oversampling	• **Duplicate data copies** of the minority class(es), to improve balance.	Can cause overfitting and poor generalization.	
Undersampling	• **Remove some data** from the majority class(es), to improve balance. For example, this can be random, or you can choose the "furthest neighbor points". • One technique is **NearMiss**, which tries to only keep points from majority class which are close to the decision boundary (i.e. avg distance to the minority class are smallest)	A good choice when you have a ton of data from majority class. Obviously, also improves training time. May discard valuable data. One option is to use an ensemble (similar to bagging).	 *Left: Undersampling. Right: NearMiss.*
Data Augmentation	• **Create synthetic** or augmented data for minority class(es). This can be through data augmentation or even GANs. • In **SMOTE**, you randomly select a close neighbor of a minority class point, and create a random point within their line segment. This requires a well-defined semantic space. • In **Mixup**, new images are created by interpolating pairs images & their labels.		 *Left: SMOTE. Right: Mixup.*
Reweighing	• Provide a weight for each class in the cost function (e.g. cross entropy), to provide more emphasis on minority classes. • **Focal Loss** modifies the cross-entropy function curvature so easy, confident predictions have less loss weight. Widely used for detection, where there are many "easy" BG negative bboxes and only a few "hard" FG positive, to pay attention to.	A ICLR 21 paper states that although mathematically equivalent, resampling is generally better than reweighing because of factors from SGD optimization.	$\mathcal{L}_{\text{CS-CE}}(\mathbf{z}, c) = -\frac{n_{min}}{n_c} \log\left(\frac{\exp(z_c)}{\sum_{i=1}^{C} \exp(z_i)}\right)$  *Left: Class-weighed CE loss. Right: CE (Blue) vs Focal Loss (green)*
Two-Stage Training	• Consists of an **imbalanced training stage** first (to learn good representations) and a **balanced fine-tuning stage** (using another method, to re-balance).		
Metric Learning	• Want to enforce good margins in an embedding space (especially for few-shot examples), in a class-balanced sensitive way. • **GistNet** tries to transfer the decision boundary shape of many-shot classes to the medium/few-shot classes. • **Large-margin softmax** tries to make margins larger, useful for generalizability of few/medium shot. **Label Distribution Aware Margin (LDAM)** is an improvement on this to consider class data size.		 *Left: GistNet. Right: LDAM.*

Few-Shot Learning

At a high level, what is the few-shot learning problem?

- In **few-shot learning (FSL)**, the goal is to learn tasks with only a few training examples.
- We usually evaluate a FSL algorithm's capability for data-efficient learning with multiple **N-way, K-shot episodes**.
- Here, **"way" = "classes"** and **"shot" = "training examples per class"**. An **episode** can be thought of as an individually defined "classification task". In detail, each episode will have:
  - A **support set** containing K training samples & labels per class (so, samples total), which the FSL algorithm trains on
    - Commonly, K is 1 or 5; N is usually 5 but can be over 100
  - A **query set**, which we test the FSL algorithm on after it trains on the support set
    - Generally at least NK in size
    - For testing purposes we have labels, but the FSL algorithm doesn't train on them; it's purely a forward pass
- In general, FSL episodes are easier with fewer classes (lower way) and more examples per class (higher shot).
- FSL episodes are harder with more classes and less examples per class.
- A variant of few-shot learning is **zero-shot learning**, which learns solely on the category name, textual descriptions, or attributes.
- Note that many few-shot methods are more tailored to that problem, and may be insufficient to tackle the more general **long-tailed problem**.



What is Meta Learning?

- In meta learning, the goal is to optimize a model to perform well across a distribution of tasks, instead of just one task. It should quickly adapt to new tasks using only a small amount of data – goal is to "learn to learn"
- Optimization-Based
  - Here, the goal is to learn an initialization that allows for fast & effective adaptation via fine-tuning on a few examples
  - Involves training a model so it can adapt quickly with only a few gradient steps
  - Examples include MAML (Model Agnostic Meta Learning) and REPTILE
    - In MAML, we compute preliminary updated parameters on multiple tasks, then make a meta-update by evaluating performance on new/held-out tasks (requiring second order derivatives)
    - Intuitively, we gather what a small update looks like for multiple tasks, then adjust the parameters so that future updates work even better.
    - You want to "learn how to learn" efficiently, rather than learning for a specific downstream task; the goal is to learn something of a higher (meta) order, not necessarily the task directly
    - For example, humans learn certain universally applicable mental tricks/techniques that allow us to learn novel tasks/concepts rapidly
- Model-Based
  - HyperNetworks directly regress/predict the parameters of another network
  - Have been shown to work for very large networks (hundreds of millions)
  - Note, HyperNetworks are conceptually similar to LoRA
- Applications:
  - Few-Shot Learning
  - Quick generation of NeRFs
  - Personalized/dynamic networks
  - Neural architecture search
- Limitations:
  - Can be prone to overfitting and bad generalization
  - Can be hard to train/optimize

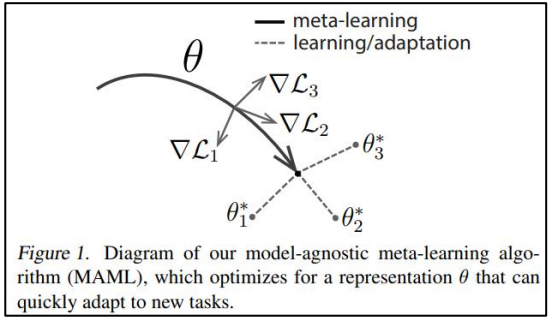


Figure 1. Diagram of our model-agnostic meta-learning algorithm (MAML), which optimizes for a representation θ that can quickly adapt to new tasks.

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

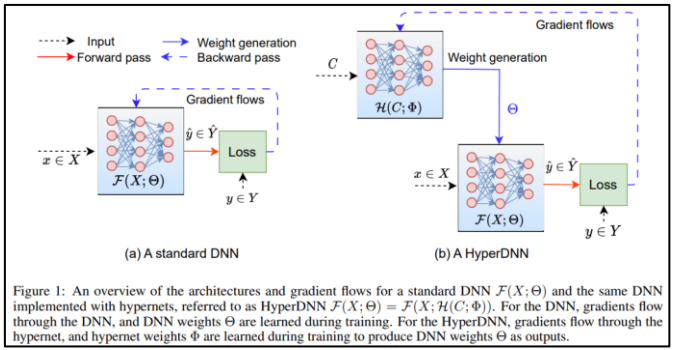
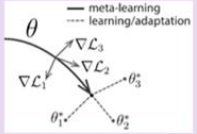
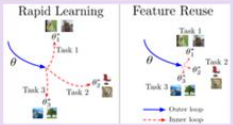


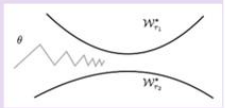
Figure 1: An overview of the architectures and gradient flows for a standard DNN $\mathcal{F}(X;\Theta)$ and the same DNN implemented with hypernets, referred to as HyperDNN $\mathcal{F}(X;\Theta) = \mathcal{F}(X;\mathcal{H}(C;\Phi))$. For the DNN, gradients flow through the DNN, and DNN weights Θ are learned during training. For the HyperDNN, gradients flow through the hypernet, and hypernet weights Φ are learned during training to produce DNN weights Θ as outputs.

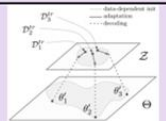
Summarize the following optimization meta learning approaches for few-shot: MAML, Reptile, LEO.

Summary of Methods for Few-Shot Learning

Metric Meta Learning | Optimization Meta Learning | Transfer Learning

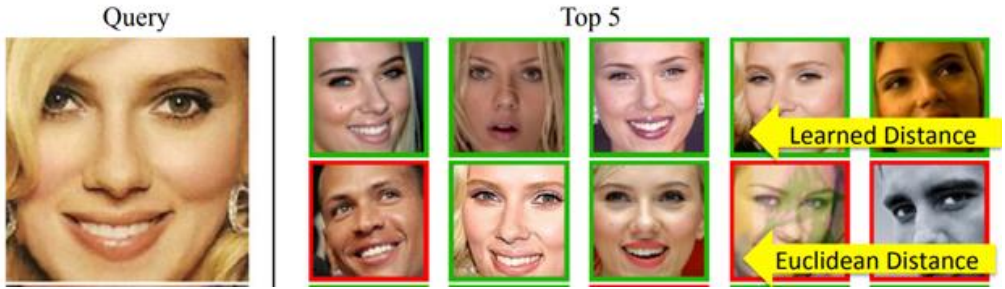
Name	Category	Description	Figure
MAML: Model Agnostic Meta Learning (ICML 17, > 5000 cits)	Optimization Meta Learning	• Goal: **Learn good NN initialization that's easy to fine-tune** • Parameters of NN explicitly trained so a few gradient steps with a few training data from a new task will produce good generalization performance • Very literal approach. Repeatedly 1) **samples tasks** 2) for each task, takes a **gradient step on individual copy of weights**, and 3) meta-updates weights, **backpropagating** through step 2 so that **weights would minimize loss after step 2's gradient step**: $\min_\theta \sum_{T_i \sim p(T)} \mathcal{L}_{T_i}(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{T_i}(f_\theta)})$ • The 3rd step involves taking the gradient of a gradient (**2nd order derivative**), unrolling through the computational graph. However, it's shown that **First-Order MAML is almost as good**, by treating the weights from step 2 as constants. • There's evidence MAML mostly reuses/learns high-quality features, rather than "learning how to learn rapidly".	 **Algorithm 1** Model-Agnostic Meta-Learning **Require:** $p(\mathcal{T})$: distribution over tasks **Require:** α, β: step size hyperparameters 1: randomly initialize θ 2: **while** not done **do** 3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$ 4: **for all** $\mathcal{T}_i$ **do** 5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples 6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ 7: **end for** Note: the meta-update is using different set of data. 8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ 9: **end while** 

Reptile (OpenAI, 18, >1000 citations)	Optimization Meta Learning	•Like MAML, goal is to find a parameter configuration which is close to the optimal parameter manifolds of all tasks
		•Repeatedly 1) **samples tasks**, 2) separately trains parameters on each **task for multiple gradient steps**, and 3) **meta-updates the original parameters** towards the averaged new parameters
		•Multiple SGD steps makes it different than joint, mixture training
		•They theoretically show that this generalizes MAML and also **implicitly involves 2nd derivatives**, but is **more efficient**
			
			Algorithm 2 Reptile, batched version
			Initialize θ
			for iteration $= 1, 2, \ldots$ **do**
			Sample tasks $\tau_1, \tau_2, \ldots, \tau_n$
			for $i = 1, 2, \ldots, n$ **do**
			Compute $W_i = \mathrm{SGD}(L_{\tau_i}, \theta, k)$
			end for
			Update $\theta \leftarrow \theta + \beta \frac{1}{n} \sum_{i=1}^{n} (W_i - \theta)$
			end for

LEO: Meta-Learning w/ Latent Embedding Optimization (DeepMind,ICLR 19,>700 cits)	Optimization Meta Learning	• While MAML operates directly in a high dimensional parameter space, LEO performs **meta-learning within a low-dimensional latent space** • Thus, **parameters are generated/predicted**	

At a high level, what is metric learning? How can it be applied to few-shot?

- **Metric Learning** (i.e. **similarity learning**) aims to measure how related two objects are, in a lower-dimensional, semantic manifold
- It can be learned with a **regression** or **classification** based framework, using labeled pairs of similar or dissimilar images
- Alternatively, **triplets** of can be used where is known to be more similar to than .
    - This is **weaker supervision** than regression or classification, since instead of providing an exact measure of similarity (usually one-hot), one only has to provide the relative order of similarity. This also allows the use of data augmentation, for self supervised learning (e.g. simCLR)
    - However, mining for the “hard” examples for to train on is important, and nontrivial
- The embeddings can **readily be used for one-shot learning tasks** (e.g. face verification databases), and **support easy additions** at test-time without retraining. You just need to embed the support elements, and use **K nearest neighbors to classify**.
- Besides FSL, other applications for metric learning include **ranking** (for **retrieval** or **recommender systems**), large-scale **classification**, or **clustering**. It is also common to use the learned representation from **self-supervised metric learning** for a wide range of **downstream tasks via transfer learning**.



Explain how few-shot can tackled with transfer learning, optimization meta learning, and metric meta learning.

FSL via Transfer Learning

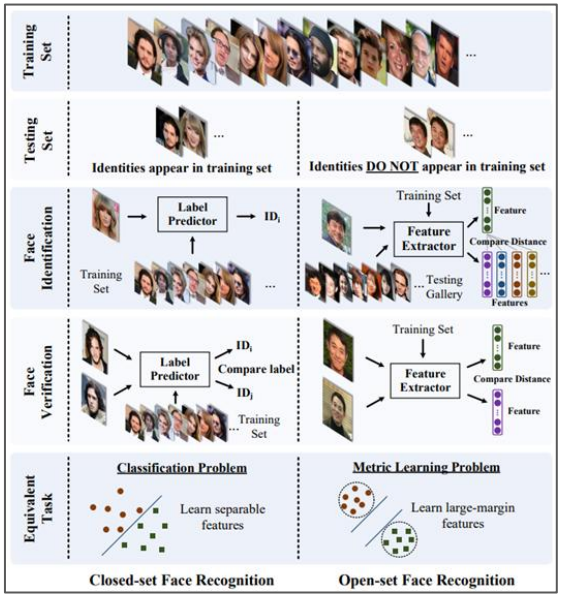
- **Pre-train on a large, separate dataset** with many classes and instances per class (where classes distinct from those seen during evaluation), e.g. ImageNet
- Training can be fully supervised or self-supervised
- During evaluation, **transfer the learned representations**, for example by
  - **Fine-tuning** on the NK episode training examples, or
  - Using it as a **fixed feature extractor** and learning a linear classifier (e.g. softmax) on top
- Intuitively, at the transfer learning step, you're not "teaching it to reason", just "**recontextualizing**" with the few examples that you have
- Most basic approach, but surprisingly, **very competitive** with more complicated meta learning based methods

FSL via Optimization-Based Meta Learning

- Insight: **Replicate the FSL episode-based evaluation procedure during training**
- Intuitively, optimize for "fast-weights/parameters" in network which can be adapted in a data-efficient manner to episodes without overfitting, using only a few gradient steps
- However, some believe that this is more less due to **reusing/learning high-quality features**, rather than "rapid learning of each task".

FSL via Metric-Based Meta Learning

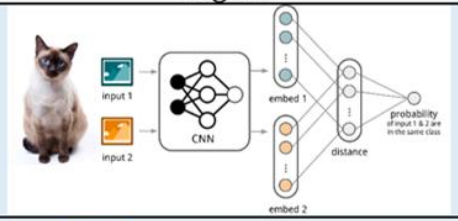
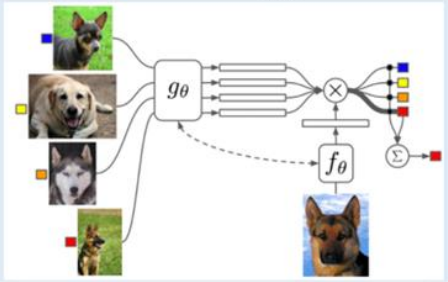
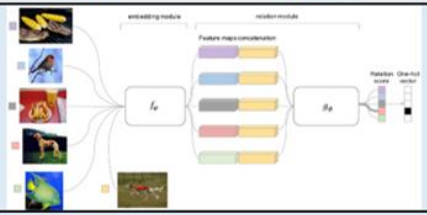
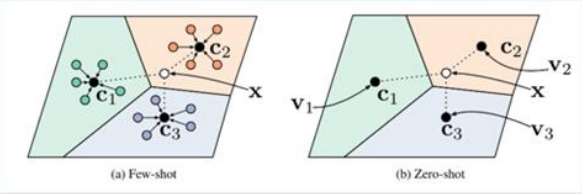
- Based on metric learning; want to learn a mapping from images to an **embedding space**, and use a (possibly learned) **similarity function** between points
  - Goal: images from the same category are closed together and from different categories are far apart.
  - Ideally, the mapping will hold true for unseen categories; during evaluation, we embed the support set and use nearest-neighbors to classify
  - Essentially assumes that can be modeled by the form , where k is a similarity function (e.g. embedding Euclidian distance)
- "Older", more classical approach to FSL
- Especially useful for tasks like **face verification**, where you have a **frequently changing database** of objects (pictures of people) that you want to recognize, but only a few data samples for each object. Unlike Meta-learning approaches, you would **not need to retrain** when you add or remove people from the database.
- Can involve contrastive methods (where you have positive and negative sampling), or methods based on softmax (which is a generally easier to apply)



Summarize the following metric meta learning approaches for few-shot: Siamese NNs, Matching Networks, Relation Networks, Prototypical Networks.

Summary of Methods for Few-Shot Learning

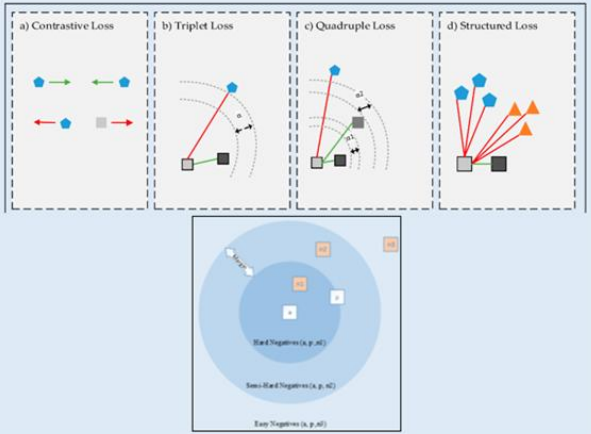
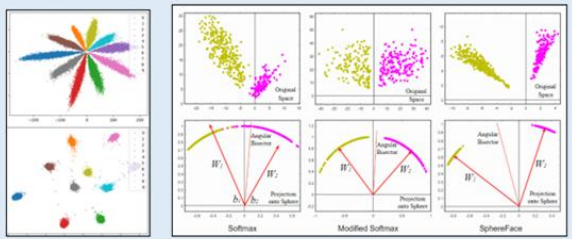
Metric Meta Learning | Optimization Meta Learning | Transfer Learning

Name	Category	Description	Figure
Siamese NNs (ICML 15, >2500 cits)	**Metric Meta Learning**	• Feeds **pairs of images into a CNN** to get two **embeddings** • The **L1 distance** between embeddings fed into a linear layer to predict **similarity probability score** of if they are same class • Trained with binary cross entropy loss • At test time, a query image is classified as the class of the support image with highest class-similarity probability	
Matching Networks (Google, NIPS 16, >3700 cits)	**Metric Meta Learning**	• Given support $S = \{x_i y_i\}_{i=1}^k$ where y_i are one-hot vectors, learns embedding functions f, g such that for query x we have $P(y\|x, S) = \sum_{i=1}^k a(x, x_i) y_i$ • a is an "**attention kernel**" **weighing one-hot vectors**, based on **cosine similarity softmax**: $a(x, x_i) = \frac{\exp\left(cossim\left(f(x), g(x_i)\right)\right)}{\sum_{j=1}^k \exp\left(cossim\left(f(x), g(x_j)\right)\right)}$ • **Training mirrors evaluation**: during training, loss explicitly minimized over small episodic samples of support & queries • Potentially, f, g can be the same feature extractor, but they instead propose a more complicated **bidirectional LSTM**	
Relation Networks (CVPR 18, >2000 cits)	**Metric Meta Learning**	• Similar to Siamese NNs gets pairs of **embeddings**, but **concatenates them** and feeds into a NN to **predict relation score** instead of using L1 distance • Also, instead of cross entropy, takes a regression approach and uses MSE with a one-hot vector	
Prototypical Networks (NIPS 17, >3500 cits)	**Metric Meta Learning**	• Embeds all support images, and defines a **prototype embedding for each class mean** • Given query x, **distribution over classes given as softmax over negative squared Euclidean distance** • It's argued that this **inductive bias** reduces overfitting • **Training mirrors evaluation**: episodes sampled during training • Trained with cross entropy	

Summarize contrastive and non-contrastive-margin metric learning approaches for few-shot. What are their advantages and disadvantages?

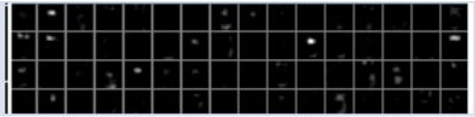
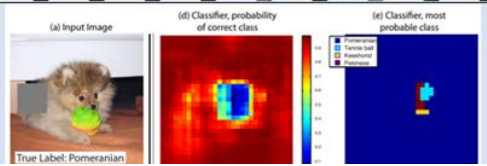
Summary of Methods for Few-Shot Learning

Metric Meta Learning | Optimization Meta Learning | Transfer Learning

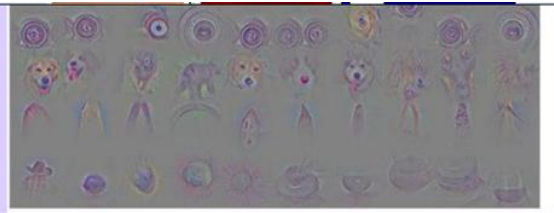
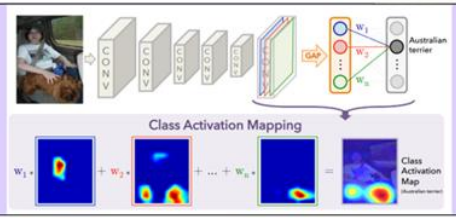
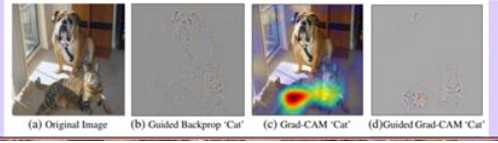
Name	Category	Description	Figure
Contrastive Metric Learning: Triplet Loss (FaceNet,CVPR 15,>10K cits), Quadruplet Loss (CVPR 17,1K), Structured Loss (CVPR 16, 1.2K), N-Pair Loss, Magnet Loss, Clustering loss, MoCO (CVPR20, 21 K)	**Metric Meta Learning**	• Goal: pull together embeddings with same label, and push away those with dissimilar labels, with **explicit pos/neg samples** • Classical **contrastive loss** uses pairs of samples and margin α: $\mathcal{L} = 1_{y_1=y_2} D^2(f(x_1), f(x_2)) + 1_{y_1 \neq y_2} \max(0, D^2(\alpha - f(x_1), f(x_2))$ • **Triplet loss** uses A and P with same class, and N of different class: $\mathcal{L}(A,P,N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$ • **Structured** and **N-Pair** Losses tries to improve sampling with all samples in a batch, instead of ignoring them and only using one • **Magnet** and **Clustering** loss are based on the dataset as a whole, but is complex and hard to scale. • Often paired with heavy data aug, large batch size, and/or memory bank to efficiently reuse embeddings from the prev. iteration • Main **difficulty of contrastive learning is sampling comprehensively**, mining the "**hard negatives**". Also, batch-sample based learning enforces local margins, but a global margin for all labels is hard to achieve in practice. Thus, non-contrastive losses are often easier to use. • Useful for **unsupervised** representation learning (e.g. simCLR)	
Margin Non-Contrastive Metric Learning: Center Loss (ECCV16,27K cit), LargeMargin Softmax(ICML16,1K), SphereFace (CVPR 17, 2K cit), CosFace (CVPR 18, 1300 cit), ArcFace (CVPR 19, 2300 cit)	**Metric Meta Learning**	• Goal: pull together embeddings with same label, and push away those with dissimilar labels, **without explicit pos/neg samples** • **Center loss** adds an additional term to softmax loss, to pull each batch of N features towards its (jointly trained) class center • **Large-Margin Softmax** notes that for sample (x_i, y_i), the softmax loss for a FC layer exponentiates $W_{y_i}^T x_i = \|W_{y_i}\| \|x_i\| \cos(\theta_{y_i})$ in the numerator (selects the g.t. class vector in W and takes dot prod). Then, a larger margin is explicitly enforced by enforcing larger θ. • **SphereFace** improves margin by enforcing the centers to be on a hypersphere w/ normalization, improving the decision boundary • Generally **only applicable when supervised**. For unsupervised settings or when you have a lot of out-of-distribution samples, contrastive learning may be better. • General **tradeoff** with **within-domain accuracy** (**smaller margins**, standard classification) vs **out-of-distribution generalization** (**larger margins**, longtail/fewshot/openset verif.)	 Left: Softmax vs Center Loss. Right: Softmax vs SphereFace

Explainable Artificial AI (XAI)

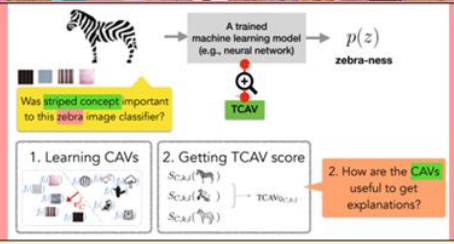
At a high level, describe the following basic XAI methods: Layer weight/activation visualization, maximally activating retrievals, and occlusion maps.

Show Layer Activations	Show forward pass activations (as an image) for an input	Check for localized regions in later layers. If an activation map is zero for many different inputs, this can indicate dead filters	
Show Layer Weights	Show convolutional filter weights (as an image) at layer of trained network	Usually, well-trained networks should display nice and smooth filters without noisy patterns.	
Maximally Activating Retrievals	Take large dataset of images, feed them through the network. Keep images maximally activate some neuron	Gives an understanding of what specific neurons are looking for in its receptive field.	
Occlusion Maps	Plot the probability of the correct class as a function of the position of an occluder object (e.g. a gray square), as a 2D heatmap.	Helps understand where a CNN is looking for its prediction, e.g. , checks if it's looking a context clues/backgrounds rather than the main object	

At a high level, describe the following backprop-based XAI methods: Guided backprop, DeepDream, CAM, Grad-CAM.

Guided Backpropagation	Backpropagate prediction all way back to the input while zeroing-out negative gradients (only accounts for positive contributions, reducing noise)	Fine-grained visualization, in input space to help understand the important parts of the input in making prediction.	
DeepDream	Apply gradient ascent on an image or noise, freezing network weights, to optimize the probability classification of a certain class or (classes).	Shows the salient features used for the task, through generation. Can reveal biases/correlations in network.	
Class Activation Mappings (CAM)	In architecture, number of feature maps in the last conv layer equals the number of classes. Maps are global averaged pooled and fed directly into softmax. This forces a heatmap-like representation. CAM is the weighted average of those feature maps.	Shows the most important areas in input image used for the prediction.	
Grad-CAM	Weighted sum of feature maps activations; weights are from the gradients with respect to a target class, averaged per feature map.	Shows the most important areas in input image used for the prediction.	

At a high level, describe the following numerical-based XAI methods: T-SNE, TCAV.

T-SNE	A method for non-linear dimensionality reduction, based on neighbor distances. Can be applied to feature embeddings for network inputs.	Can check for clusters within data.	
Testing with Concept Activation Vectors (TCAV)	Given images of concepts and random images, trains linear classifier for vector orthogonal to hyperplane decision boundary.	Flexible way to quantitively check how much abstract concepts (e.g. gender, race) are related to a trained-CNN prediction. Concept need not be seen during training.	

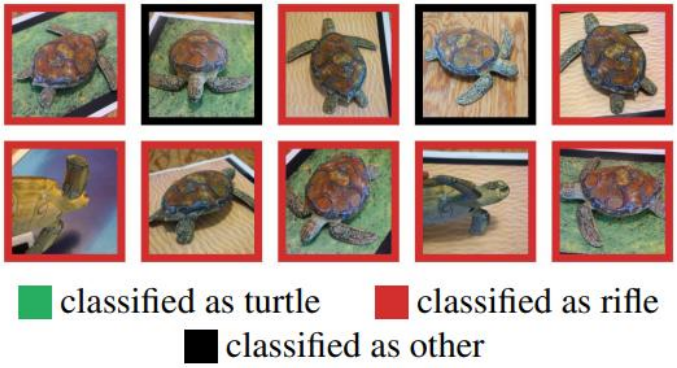
Security / Adversarial Attacks

Explain what white box and black box attacks are, and how they're typically implemented.

- **White box** attacks have access to neural network weights
  - You can use **gradient ascent** to backpropagate back to the original input image
- **Black box** attacks do not have access to the weights of the neural network.
  - A **substitute model** can be used to emulate the original model, using a transferable attack strategy
  - A **query feedback** mechanism can be used, along with random/grid/local search. You just keep modifying the input until you break the CNN and produce incorrect results. Modifications can be things like affine transform, changing hue, etc.
- Most attacks that are designed to fool a particular CNN also fool many other CNNs
- There are two theories on why adversarial attacks exist:
  - Consequence of SGD-based approximate optimization
  - Reliance on human-invisible non-robust feature patterns (i.e. overfitting to the training dataset)

Explain what real-world adversarial attacks are, and give some examples.

- Real world attacks face some additional challenges:
  - Needs to have viewpoint invariance and robustness to lighting, occlusion, etc
  - If it's on something like a street sign, cannot be on the whole image, only on the foreground obj
  - Cannot just be imperceptible; needs to be perceptible by camera
  - Needs to be robust to some fabrication error (eg by printer)
- The paper "Synthesizing Robust Adversarial Examples" shows that attacks can be 3d printed and work in the real world.
  - A 3D turtle texture is optimized to be an adversarial attack from many viewpoints and lighting conditions with differentiable rendering
- In the paper "Robust Physical-World Attacks on Deep Learning Visual Classification", real-world attacks are performed on stop signs
  - The adversarial optimization procedure is performed over a distribution (dataset) of "realistic" stop signs, by using a hybrid real & synthetic augmentation approach.
  - A mask is used to ensure the attack pattern is on the foreground stop sign
  - There is also a loss that encourages colors/patterns to be easily printable



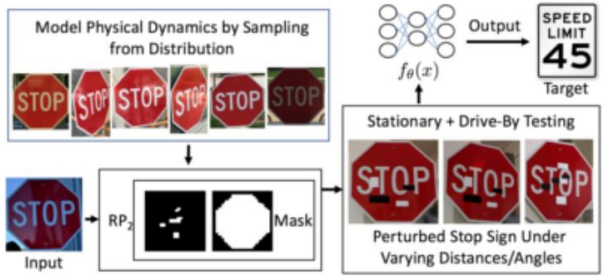


Figure 2: RP_2 pipeline overview. The input is the target Stop sign. RP_2 samples from a distribution that models physical dynamics (in this case, varying distances and angles), and uses a mask to project computed perturbations to a shape that resembles graffiti. The adversary prints out the resulting perturbations and sticks them to the target Stop sign.

What are some typical defenses for adversarial attacks?

- Adversarial training (eg training from FGSM or iterative FGSM attacks)
  - In general, any attack can be transformed into a defense in this way
- Ensemble voting
- Adversarial classifier

$$\boldsymbol{X}^{adv} = \boldsymbol{X} + \epsilon \operatorname{sign}\left(\nabla_X J(\boldsymbol{X}, y_{true})\right)$$

FGSM procedure, where J is the loss function

Efficient Deep Learning

At a high level, what is the goal of efficient DL?

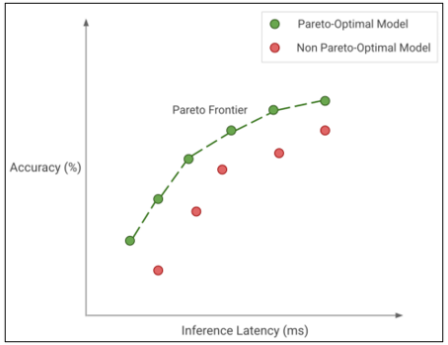
- The goal of **efficient deep learning** is to improve the **quality/footprint ratio** of models:

$$\text{Model Efficiency} = \frac{\text{Quality}}{\text{Footprint}}$$

Quality: Measured by metric, e.g. Accuracy, precision, recall, IoU, MSE, etc.

Footprint: In terms of runtime, computational hardware needed, or data.

- While usually not a priority for general ML research, it's important for:
  - Very large models (which cost substantial amounts of money)
  - Deployment at the edge (e.g. inference of NNs on smartphones)
- In general, there is a **Pareto optimal frontier** that describes optimal efficiency for intrinsic trade-offs between quality and footprint. The goal of efficient DL is to approach the frontier.
- Methods can be broken down into 5 categories:
  - **Compression**: Can we compress parts of the given model graph?
  - **Learning Techniques**: Can we train the model better? Techniques may involve the data and/or loss functions.
  - **Architectural**: Can we use/design layers and architectures which are fundamentally more efficient?
  - **Automation**: Can we search for more efficient models?
  - **CS-Based:**
    - **Caching**: if some inputs are more common than others. Before calling model, check cache
    - **Batching**: Gather prediction requests until you have a batch, then perform them in parallel as a batch. Batch size tradeoffs between throughput and latency

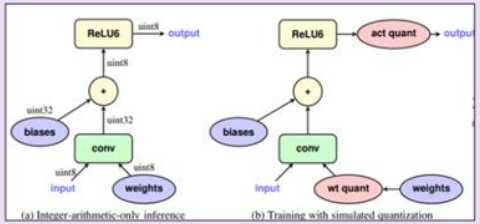
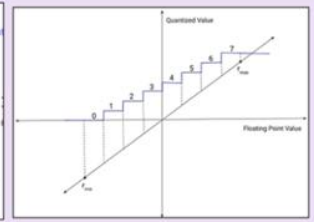


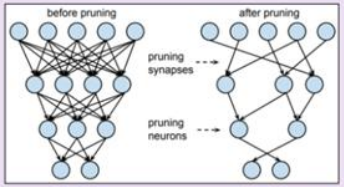
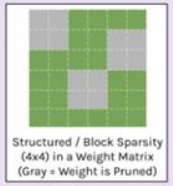
Explain some compression and architectural based methods for efficient DL.

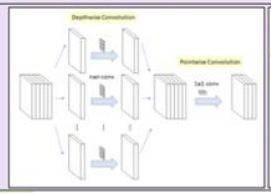
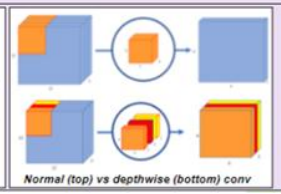
Overview of Methods for Efficient Deep Learning

Footprint Reduction | Quality Improvement | Both

Method Name	Category	Description	Figure

Numerical Quantization	**Compression**	• **Replaces high-bit signed floats with lower-bit signed ints**, weights and/or activations
		• Conversions to/from ints are performed using **scale mapping** (range-based normalization)
		  ○ E.g. suppose float range is [0,1] and we want to convert 0.5 to int8 which can represent ints in [-128,128]; the result is 64
		  ○ At the extreme end, can even be **binarized (BNNs)**
		• Generally, multiplications reduced precision; additions lighter and can be full precision
		• Simplest method is to apply **quantization post-training**, but has more accuracy loss
		• Can also make **training quantization-aware** (e.g. by "simulating" it, allowing the network to adapt and achieve better performance)
		• Can lead to significant (~4x) reductions in model size and inference latency
			 

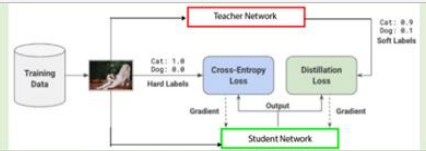
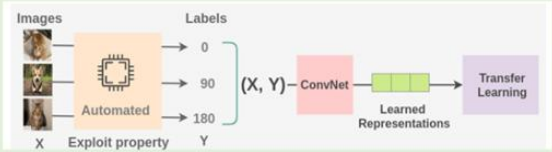
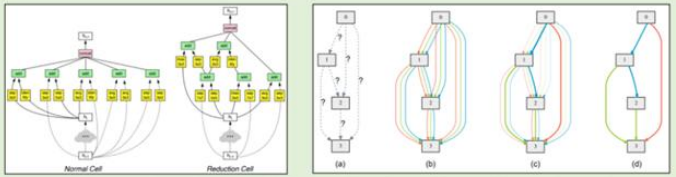
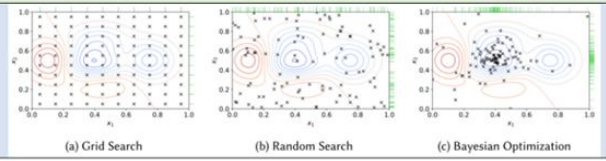
Model Pruning	**Compression**	• **Removes weights** (i.e. set to 0, making applicable for sparsity optimization) or **remove nodes/filters** from a trained model (more easily supported, but can hurt effectiveness more) • Goal: reduce memory and make calculations faster • Several heuristics to remove weights or modes, e.g. based on weight magnitude (replace those close to 0 with 0), activation outputs (replace "dead" neurons which always output small values), or derivatives wrt loss • Network can be retrained after pruning to restore accuracy (iteratively)	 

Depthwise Separable Convolutions	**Architectural**	• **Decomposes the regular convolution**, for increased efficiency:
		  ○ **Depthwise convolution** operation preserves depth C of input. C many "groups" of n x n x 1 kernels are used, each separately applied to each channel of the input to get C channels
		  ○ **1 x 1 convolutions** provide desired depth
		• While fewer in parameters, performance lags behind large ResNet variants
		• Originally from **MobileNet** paper
			 

Explain some learning and automation based methods for efficient DL.

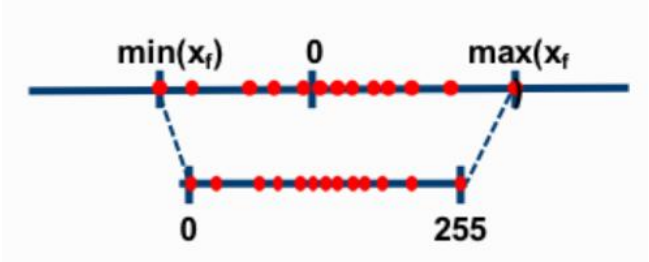
Overview of Methods for Efficient Deep Learning

Footprint Reduction | Quality Improvement | Both

Method Name	Category	Description	Figure
Student-Teacher Distillation	Learning Techniques	• First train a **larger "teacher" model**, then train a more **lightweight "student" model** with both the original training data and the softmax classification scores of the teacher (with KL divergence) • The availability of "**dark knowledge**" non-one-hot information from the teacher can improve performance for the student	
Data Augmentation	Learning Techniques	• Useful to improve performance and data efficiency • Can use techniques like **RandAugment** and **Mixup**, or even **GANs** to synthesize new examples • Synthetic sampling like **SMOTE** can allow for re-balancing to make up for dataset skewness	
Self-Supervised Learning Transfer	Learning Techniques	• Goal: utilize **auxiliary tasks** in a way where we can generate virtually **unlimited labels** from our existing images, to **learn useful representations** • Resulting network can then be used for **transfer learning on a domain-specific downstream task** • Useful to improve performance and data efficiency • Examples: **colorization, superresolution, inpainting, jigsaw, rotation, clustering, data-aug based contrastive learning**	
Neural Architecture Search (NAS)	Automation	• **Automates NN design** by searching space of filter sizes, layer types, depth, etc for best performance and/or efficiency ○ Have **normal cells** (in/out have same spatial size) & **reduction cells** • Several **black-box optimization** approaches can be taken including grid search, random search, RL, and evolutionary algorithms • **Differentiable NAS** (e.g. DARTS) is also possible by a continuous relaxation of the search space allowing gradient-based optimization ○ Discrete operation choices replaced by mixture operations in DAG graph • Bilevel optimization problem: need to optimize the parameters & architecture weights. Can approximately solve by iteratively optimizing	 *Left: Normal vs Reduction cells (NasNet). Right: Differentiable NAS (DARTS)*
Hyperparameter Search	Automation	• **Automates tuning NN hyperparameters** for better performance, without changing footprint • Several **black-box optimization** approaches can be taken including grid search, random search, Bayesian optimization, RL, and evolutionary algorithms	

How does QAT work? What types of errors can be incurred?

2. **Insertion of Quantization Nodes**: Modify the model by inserting "fake" quantization nodes in the network. These nodes simulate the effect of quantization (like rounding and clipping) in the forward pass during training, but the backward pass is still done using high-precision calculations. This step is crucial as it helps the model to 'learn' the quantization noise and adapt to it.
3. **Calibration**: Run a few batches of data through the modified model to calibrate the quantization parameters. This step involves determining the range (minimum and maximum values) for activations and setting the scale and zero-point for quantization. The calibration can be done using either a subset of the training data or a separate calibration dataset.
4. **Fine-tuning**: Retrain (fine-tune) the network with these quantization nodes in place. During this process, the model adjusts its weights to compensate for the quantization errors introduced. The goal is to minimize the impact of quantization on the model's accuracy. The training can be done for a few epochs using the same hyperparameters as the original training.

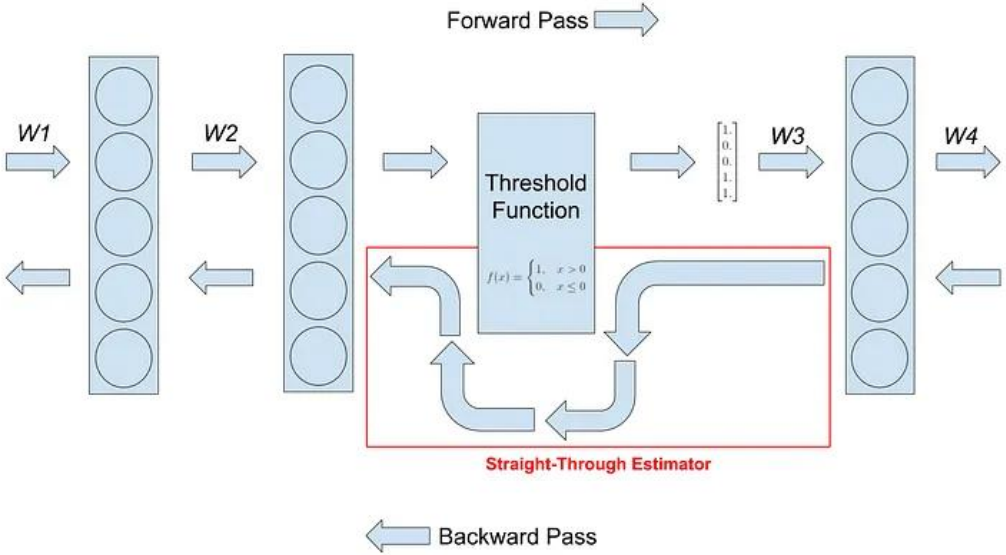


$$x_q = round\left((x_f - min_{x_f})\underbrace{\frac{2^n - 1}{max_{x_f} - min_{x_f}}}_{q_x}\right) = round(q_x x_f - \underbrace{min_{x_f} q_x}_{zp_x}) = round(q_x x_f - zp_x)$$

1. **Rounding Error**: This is the most common type of error in quantization. When converting from a high-precision format (like 32-bit floating-point) to a lower precision (like 8-bit integers), values are rounded to the nearest representable value in the lower precision format. This rounding can introduce small errors in the weights and activations of the network.
2. **Clipping Error**: When quantizing, the range of values that can be represented in the lower precision format is often smaller than in the original format. If a value falls outside this representable range, it is clipped to the nearest bound (either the minimum or maximum representable value). This clipping can lead to significant errors, especially if the original value distribution has a long tail.

How does the backwards pass work during QAT?

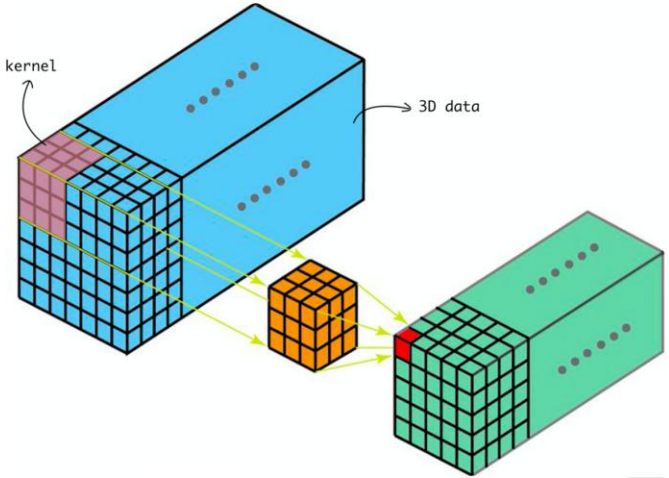
- Since a QAT model is a piecewise constant function where gradients vanishes almost everywhere (as quantization is non-differentiable), we cannot perform vanilla backprop
- Instead, we use a “straight-through estimator” (STE), ignoring the fakequant node and bypassing it as if it was an identity function.
- This “gradient” is not the actual gradient of the loss function, yet empirically/heuristically seems to work (this is still not fully understood)



3D Deep Learning

How do 3D convolutions, and deconv/transpose convs work?

- Instead of 3D n x n x C kernels/filters, we use 4D n x n x n x C kernels/filters to slide over the 4D input of W x H x D x C
  - Sliding dot product aggregates local features in 3D; width, height, and depth
- Deconvolution is similar, but uses padding in between to increase spatial size of output feature map



How do graph convolutions work?

At a high level, the idea is as follows. Suppose we have graph $G = (V, E)$, containing:

- N many D-dimensional vertices $V = \{v_i | v_i \in \mathbb{R}^D\}$, $|V| = N$
  - This can be summarized as a feature matrix $X_{N \times D}$
- Edges E
  - This can be summarized as an adjacency matrix $A_{N \times N}$

We wish to have a neural network with layers:

$H^{l+1} = f(H^l, A)$

The first input is $H^0 = X$, and $H^L = Z_{N \times F}$ are the output features (of dimension F) for each vertex.
A very simple way to have propagation of features from connected nodes on the graph is to have $f(H^l, A) = \sigma(AH^lW^l)$.
This is kind of like a FC layer (HW), where edges are "selected" using the adjacency matrix.

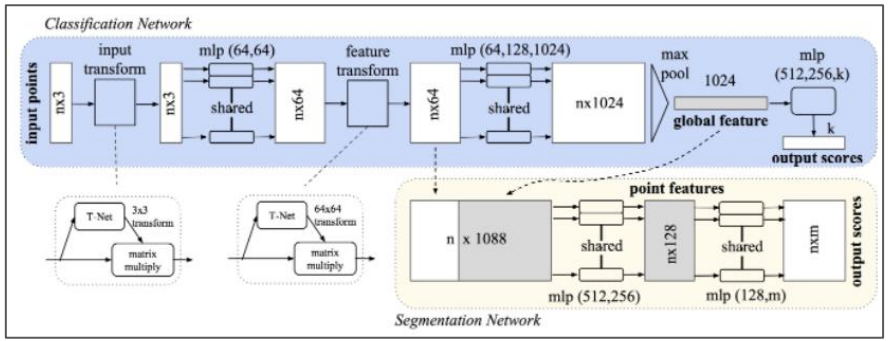
In practice, we actually use the following, which incorporates symmetric normalization and self-loops on A.

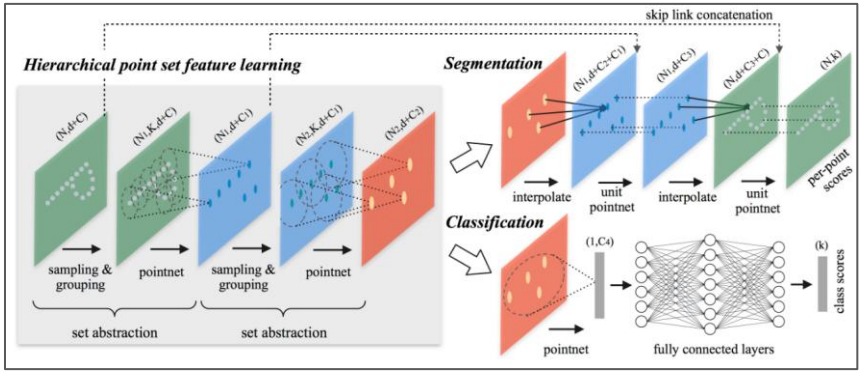
$$f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

There are some more complicated aspects to this, involving spectral graph theory and Fourier transforms.

How do PointNets and PointNet++ work?

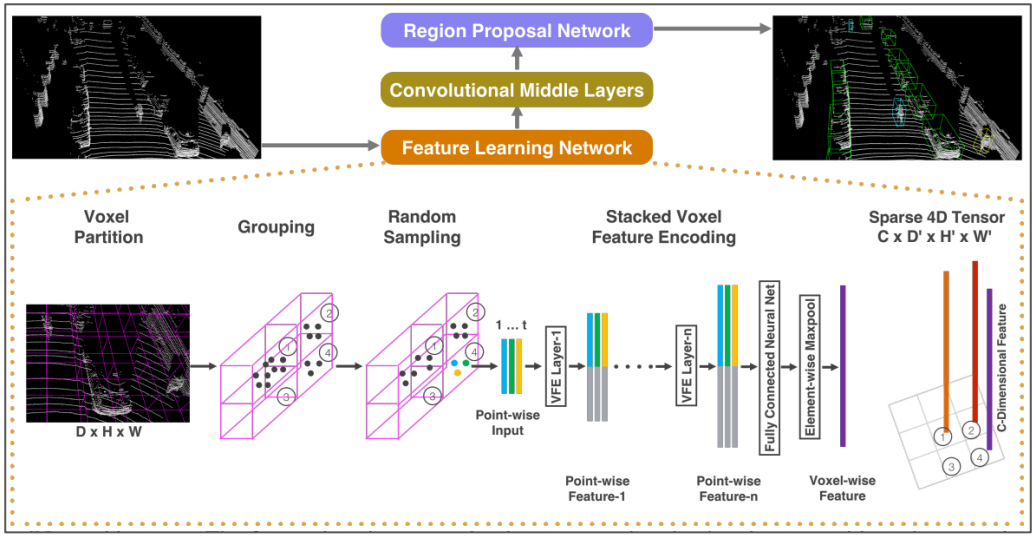
- PointNet and PointNet++ consume raw point clouds; no need to voxelize or render
- PointNet:
  - Learns both local and global point features
    - Per-point local feature vectors by shared MLPs on n individual points
      - T-Nets try to make it invariant to rotation by learning a 3x3 transformation matrix
    - Global feature vector obtained by max pooling n x 1024 -> 1024
  - Applications:
    - Classify point cloud: global feature vector fed into MLP + softmax + CE loss
    - Segment point cloud: global feature vector concat with local feature vectors, per point classification
      - Useful for object part segmentation or scene semantic segmentation
  - Notes:
    - Max pooling is an example of a **symmetric function, which is order invariant**
- PointNet++:
  - Repeatedly uses PointNets to aggregrate sets of nearby points
  - For segmentation, need to interpolate to obtain the original input dimensions





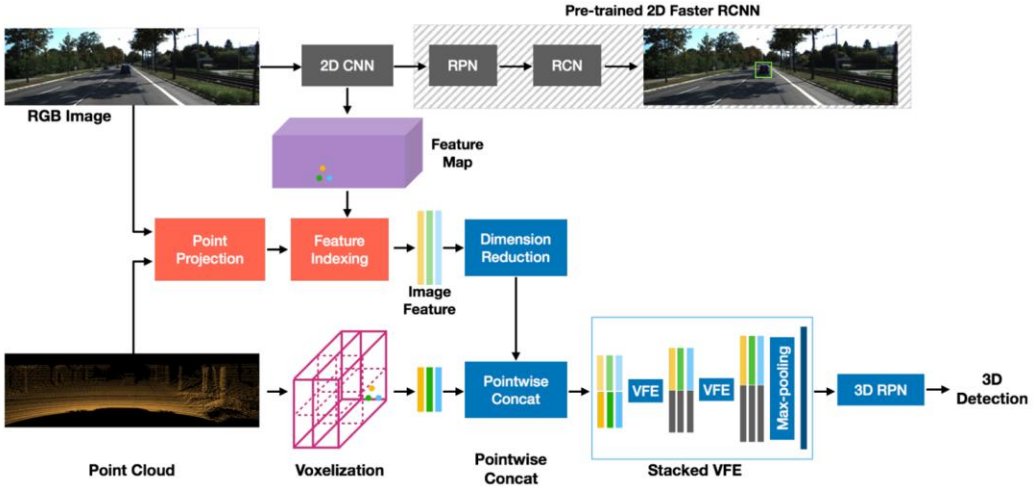
How do voxelnets work?

- Performs 3D object detection on point clouds
- Steps:
  - Partitions pointcloud into 3D voxel grid, and then applies a PointNet w/ global concat to each cell to get a sparse 3D feature map of dimensions H x W x D x C
  - Applies 3D convolutions onto 3D feature map (still in form H x W x D x C)
  - Merge depth and channels; flatten to H x W x C
  - RPN finds bboxes in 2D feature map, which also regress bbox
    - 7 Bbox params: xyz center, length, width, height, degree yaw about z axis



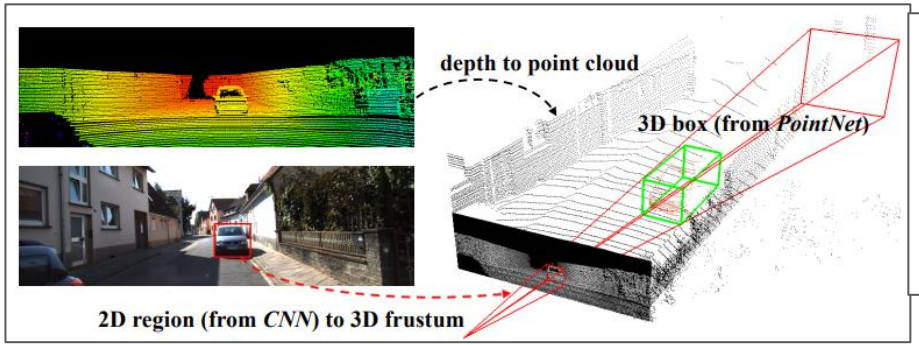
How does the MVX-Net extend voxelnets?

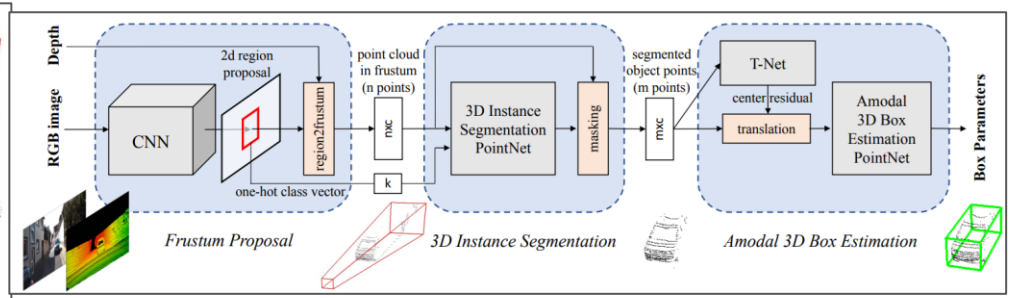
- Instead of 3D detection only using pointcloud data, also incorporates image data
- 3D points are projected to the image using the calibration information and learned corresponding image feature map features are appended to the 3D points using interpolation



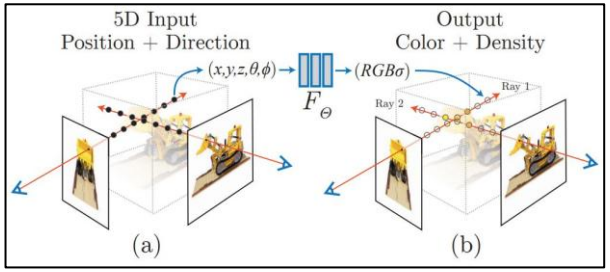
How do Frustrum PointNets work?

- Instead of just using a 3D-lidar based pointcloud (e.g. the voxelnet approach), uses RGBD + image setup
- Steps:
  - Make 2D detection on image
  - Project 2D BBox to a frustrum of points, similar to Psuedo-LiDAR
  - Segment points w/ pointnet to remove background clutter
  - Feed 3D instance segmentation of points to regress amodal (shape-completed) 3D BBox parameters
- For maximum effectiveness, combine with BEV methods to still detect 2D occluded objects





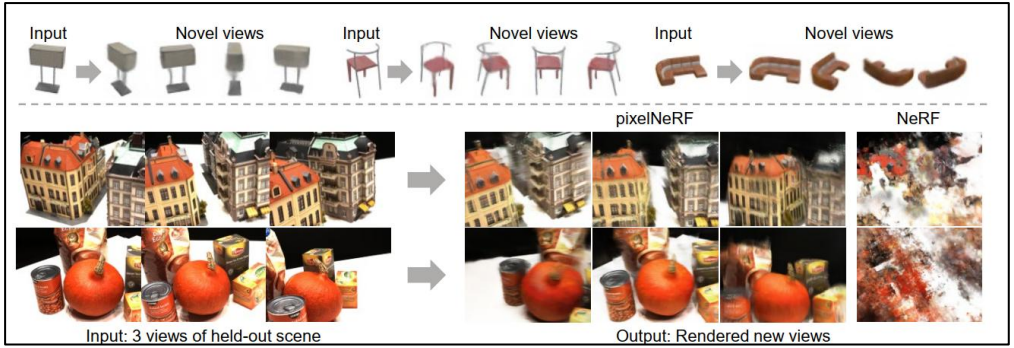
How do NeRFs work?



- Method for novel view synthesis & 3D reconstruction based on an MLP-based implicit volumetric scene
  - Given set of **images from a static scene with known camera poses**, render the scene from novel viewpoints
  - Requires good coverage eg for small object (20-100 imgs), indoor room (100-1000+ imgs)
- First points x_i are obtained by ray casting; for a pixel in a novel view, $x_i = r(t_i) = o + t_i d$
  - o is camera origin, d is ray direction, t is depth along ray
- MLP f_θ takes in 3D points x_i and predicts:
  - **Radiance (color)** $c_i = f_\theta\big(\gamma(x_i), \gamma(d)\big) \in \mathbb{R}^3$
    - Feeding in view direction allows us to learn specular reflections
  - **Volume Density** $\sigma_i = f_\theta(\gamma(x_i)) \in \mathbb{R}_+$
    - $\sigma_i \approx 0$ mostly empty air, transparent
    - $\sigma_i \to \infty$ opaque, eg solid objects or heavy fog
    - Intermediate values of σ_i represent semi-transparent things (eg dirty glass, light fog)
- **Positional Encoding** function $\gamma(\boldsymbol{x}) = [\gamma(x), \gamma(y), \gamma(z)]$ is used which empirically allows the MLP to learn better sharp color transitions and fine details
  - $\gamma(x) = [\sin(2^0\pi x), \cos(2^0\pi x), \ldots, \sin(2^{L-1}\pi x), \cos(2^{L-1}\pi x)]$
- **Volume Rendering Equation** is $\hat{C}(r) = \sum_{i=1}^{N} T_i * \alpha_i * c_i$; accumulates light along a camera ray through a semi-transparent 3D scene through weighted avg
  - $\alpha_i = 1 - \frac{1}{e^{\sigma_i \delta_i}} \in [0,1]$: **Opacity** (contribution strength) of point i
    - δ_i is the step size
    - For air $\alpha_i \approx 0$; for solid surfaces $\alpha_i \approx 1$
  - $T_i = \exp\big(-\sum_j^i \sigma_j \delta_j\big) \in [0,1]$: **Transmittance**, telling us how much light makes it to point i, without being blocked by earlier points
    - If earlier parts of ray are dense $T_i \approx 0$, if nothing in the way $T_i \approx 1$
- Training objective: minimize pixel error between predicted and GT RGB: $L = \sum_{rays} \big|\big|\hat{C}(r) - C_{gt}(r)\big|\big|^2$
- Can extract explicit 3D representation by volume grid sampling + marching cubes, or rendering depthmaps + TSDF fusion
- Advantages over traditional MVS:
  - Great at modeling soft shadows, fine details (eg hair), translucency, view-dependent specularities (eg glossy reflections)
  - Avoids artifacts of meshes, textures, etc
- Limitations:
  - Can be slow to render at inference (this has been largely mitigated in recent works)
  - Works best for static scenes; difficulties with dynamic scenes
  - Sensitive to coverage
  - Limited controllability in implicit form
  - Vanilla formulation optimized per-scene, does not generalize to new scenes

How does PixelNeRF work?

- Learns a NeRF from one or view images in a generalized feed-forward way, eliminating per-scene test-time optimization
- Projects 3D query points onto image feature map using intrinsics and uses bilinear interpolation to sample the image feature
  - Sample features then passed through linear layer and added to NeRF network at multiple intermediate activations
  - Local feature may be ambiguous/incorrect due to occlusions; model needs to learn priors to hallucinate in those cases
- Trained on only 2D multi-view image supervision; no GT 3D data or masks



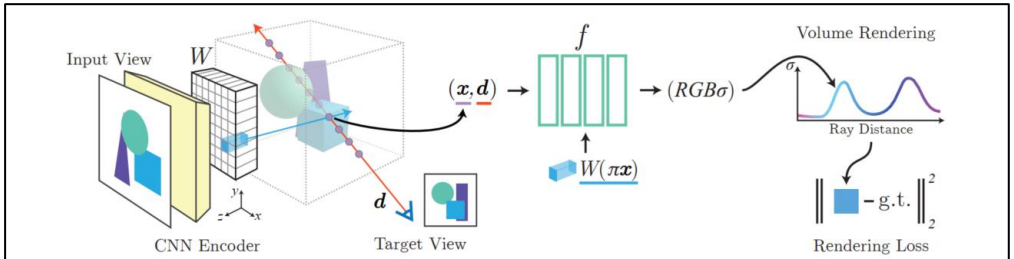


Figure 2: **Proposed architecture in the single-view case.** For a query point $\mathbf{x}$ along a target camera ray with view direction $\mathbf{d}$, a corresponding image feature is extracted from the feature volume $\mathbf{W}$ via projection and interpolation. This feature is then passed into the NeRF network f along with the spatial coordinates. The output RGB and density value is volume-rendered and compared with the target pixel value. The coordinates $\mathbf{x}$ and $\mathbf{d}$ are in the camera coordinate system of the input view.

How does DreamFusion work?

- Text-to-3D generation using pretrained & frozen 2D text-to-image diffusion model (Imagen); no 3D training data
- Training:
  - First, instantiates a NeRF per prompt, differentiably rendering random viewpoints
  - Then, perturb render with noise
  - Imagen predicts added noise
  - Noise defines a gradient direction to update NeRF parameters; backprop into NeRF weights
- Intuitively, we are updating the weights of the NeRF so that the rendered images are more likely under the prompt-conditional distribution defined by Imagen

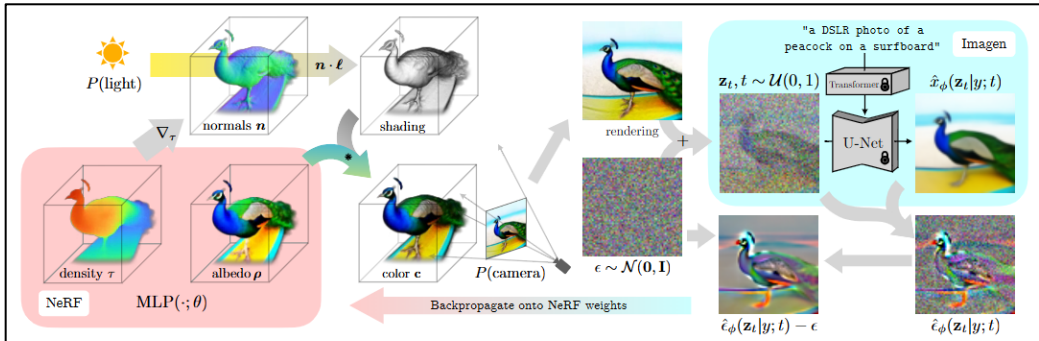
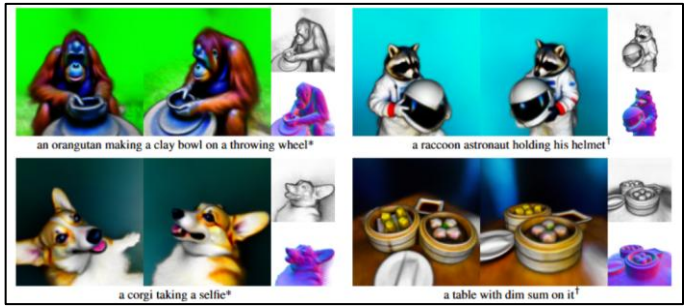
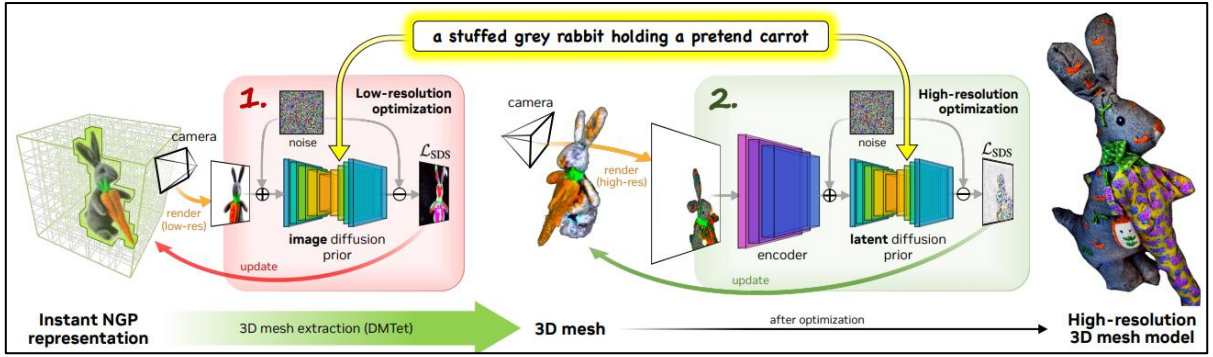


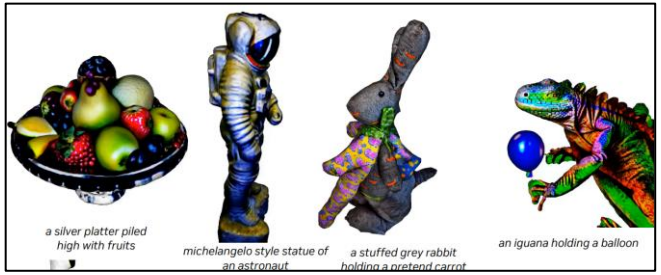
Figure 3: DreamFusion generates 3D objects from a natural language caption such as *"a DSLR photo of a peacock on a surfboard."* The scene is represented by a Neural Radiance Field that is randomly initialized and trained from scratch for each caption. Our NeRF parameterizes volumetric density and albedo (color) with an MLP. We render the NeRF from a random camera, using normals computed from gradients of the density to shade the scene with a random lighting direction. Shading reveals geometric details that are ambiguous from a single viewpoint. To compute parameter updates, DreamFusion diffuses the rendering and reconstructs it with a (frozen) conditional Imagen model to predict the injected noise $\hat{\epsilon}_\phi(\mathbf{z}_t|y;t)$. This contains structure that should improve fidelity, but is high variance. Subtracting the injected noise produces a low variance update direction $\text{stopgrad}[\hat{\epsilon}_\phi - \epsilon]$ that is backpropagated through the rendering process to update the NeRF MLP parameters.



How does Magic3D improve DreamFusion?

- DreamFusion is slow & low-res; Magic3D improves via 2 stages
- Stage One: Low-Res 3D Implicit Generation
  - Use grid-based NeRF (Instant-NGP), which is a popular option for more efficiency
  - Optimize the same way DeepFusion does, with a low-res (64x64) diffusion network
- Stage Two: High Res 3D Mesh Generation
  - First convert Instant-NGP into an SDF
  - Differentiably turn SDF into textured mesh using differentiable marching cubes
  - Differentiably render mesh at random viewpoints
  - Low res network no longer used; instead, optimize using a high-res (512x512) diffusion network
  - By doing differentiable mesh extraction instead of non-differentiable, there's more flexibility with the mesh topology





What is Gaussian Splatting?

- Explicit 3D representation that models objects as a set of Anisotropic 3D Gaussians
  - Each gaussian has: a **Mean** $\mu_i \in \mathbb{R}^3$, **Covariance Matrix** $\Sigma_i \in \mathbb{R}^{3\times3}$, **Color** $c_i \in \mathbb{R}^3$, **Opacity** $o_i \in [0,1]$
  - Anisotropic: can have different variances in different directions (can be ellipsoidal)
- Optionally, also add **spherical harmonics** to allow for view-depending effects like specular highlights
  - Common choice is to use first 9 basis functions $Y_0(v), \dots, Y_8(v)$ where $v = (x, y, z)$ is viewing direction
  - Each gaussian would have 27 new parameters representing color; $c^R \in \mathbb{R}^9$, $c^G \in \mathbb{R}^9$, $c^B \in \mathbb{R}^9$
  - Then, calculate RGB as $\left[\sum_{i=0}^{8} c_i^R * Y_i(v), \sum_{i=0}^{8} c_i^G * Y_i(v), \sum_{i=0}^{8} c_i^B * Y_i(v)\right]$
  - Intuitively, spherical harmonics are orthogonal functions defined on a sphere that can approximate any function of a sphere. So we're learning the weights of these basis functions to provide view-dependent color.
- Can be rendered / "splatted" onto 2D pixel grid using alpha blending. First we sort gaussians by depth, and then contributions are accumulated with $C_{final} = \sum_i T_i \alpha_i c_i$
  - $T_i = \prod_{j=1}^{i-1}(1 - \alpha_j)$ is the transmittance up to the i-th Gaussian
  - $\alpha_i(p) = o_i \cdot \exp\left(-\frac{1}{2}(p - x_i)^T(\Sigma_i)^{-1}(p - x_i)\right)$ is the alpha value
- Original, vanilla formulation doesn't use a neural network; parameters optimized directly using a set of images with known poses, with pixel-space loss
- Generally, point-cloud via some other algorithm (eg COLMAP) is used to initialize position
- With some heuristic scheduling, **prune** invisible gaussians (eg low alpha, or outside bbox) and **densify** for new gaussians (if gaussian has large gradient)
- Common regularizations:
  - Opacity regularization to prevent too many opaque layers / over saturation
  - Covariance regularization to incentivize compact & sharp gaussians

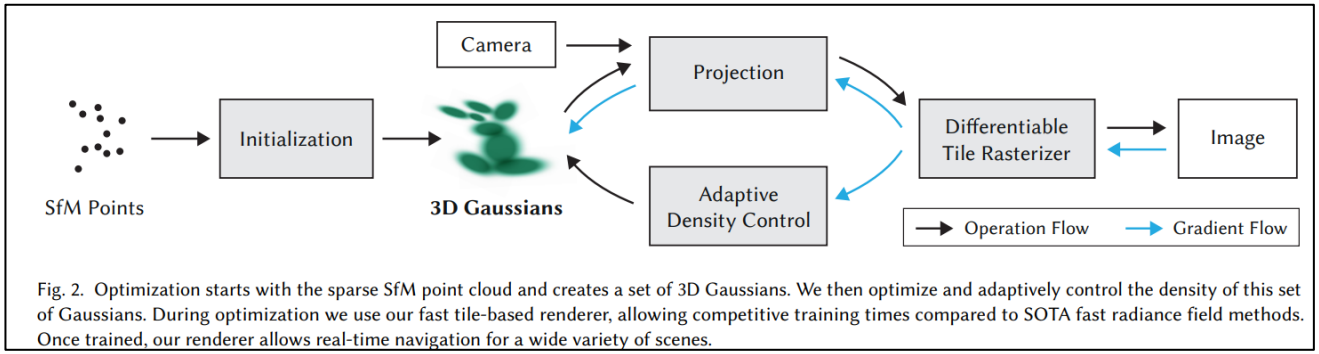


Fig. 2. Optimization starts with the sparse SfM point cloud and creates a set of 3D Gaussians. We then optimize and adaptively control the density of this set of Gaussians. During optimization we use our fast tile-based renderer, allowing competitive training times compared to SOTA fast radiance field methods. Once trained, our renderer allows real-time navigation for a wide variety of scenes.

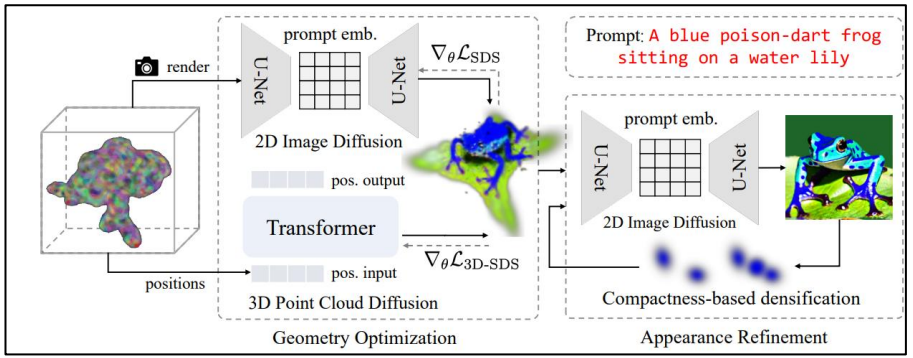
Left: Gaussian Splat. Right: Same splat with all gaussians semi-transparent

What's the connection between Gaussian Splatting & NeRFs?

- Similarities
  - Both are radiance fields; Function that map 3D locations & directions to color.
  - optimized using photometric loss
  - Require substantial coverage of scene
  - Focus is photorealistic rendering from novel viewpoints
- Differences
  - NeRF is an implicit representation, while GS is explicit
  - NeRFs are generally slower to train & render; GS allows for real-time rendering
  - NeRFs are harder to interprete
- Converting NeRF → GS
  - Initialize gaussians by querying NeRF MLP
  - Optimize/estimate shape of gaussian by analyzing local point structure.
  - Further optimize by using rendered NeRF images
- GS → NeRF
  - Render images from GS and train NeRF with that supervision

How does GSGEN work?

- Text-to-3D method by optimizing a per-instance Gaussian Splat
- First stage: Geometry Optimization
  - Given prompt, first use Point-E (a text-to-pointcloud diffusion model) to initialize gaussian centers
  - Optimize by randomly differentiably rendering views, adding noise, and updating gaussians with diffusion gradient
  - Additional loss term comes from the same Point-E 3D network, to continue updating positions of gaussians
- Second Stage: Appearance Refinement
  - Continue optimizing using 2D image diffusion network
  - Add gaussian densification algorithm based on close neighbors to improve smoothness
  - Prune gaussians with low alpha



How does LGM work?

- Generates 3D gaussian splats from text or single-view images, in a feed-forward manner
- First, obtain 4 images from a text prompt (Mvdream, text to MV image diffusion) or image prompt (ImageDream, image to MV image diffusion)
- Given the 4 images as input, regresses per-image pixel-aligned gaussian splats (128*128*4=65536 gaussians)
  - Input images are always in a canonical pose (0,90,180,270 at fixed elevation 0)
- After fusing the 4 gaussians, render novel views and supervise in rgb space
- Contends from domain gap – training using 3D asset renders, but inference using diffusion generated multi-view images

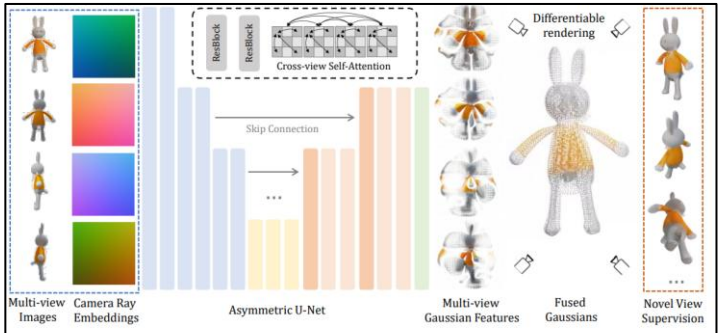


Fig. 3: Architecture of LGM. Our network adopts an asymmetric U-Net based architecture with cross-view self-attentions. We take four images with camera ray embeddings as the input, and output four feature maps which are interpreted and fused into 3D Gaussians. The Gaussians are then rendered at novel views and supervised with ground truth images.

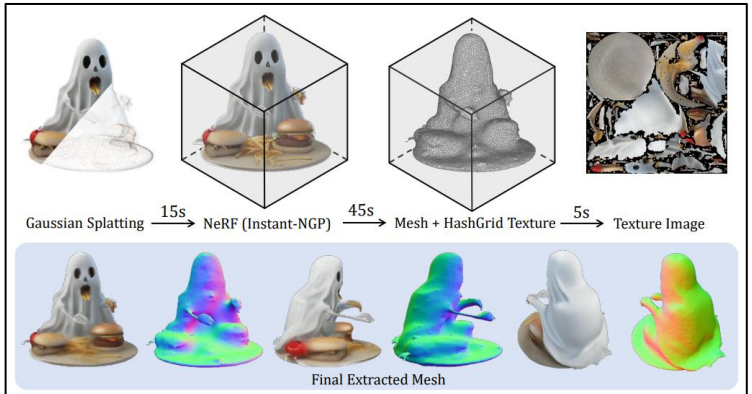


Fig. 4: Mesh Extraction Pipeline. We implement an efficient pipeline to convert the 3D Gaussians into smooth and textured meshes.

How does Point-E work?

- Text-to-3dPointCloud method with two stages
- Text-to-Image Stage:
  - First, use a fine-tuned GLIDE model to generate a synthetic 2D view from a text prompt
- Image-to-3D Stage:
  - A point cloud diffusion model generates a 3D RGB point cloud
  - Model is transformer-based, invariant to point order
  - Points represented as tensor of shape $n \times 6$, where n is the number of points and we have 3D position + RGB values, normalized to [-1,1]
  - Points passed through linear layer to get embedding; in total, input sequence has point embeddings, timestep, and image embeddings
  - Encoder processes sequence and the last n tokens are projected as ϵ and Σ for the diffusion step
- Trained on millions of 3D models

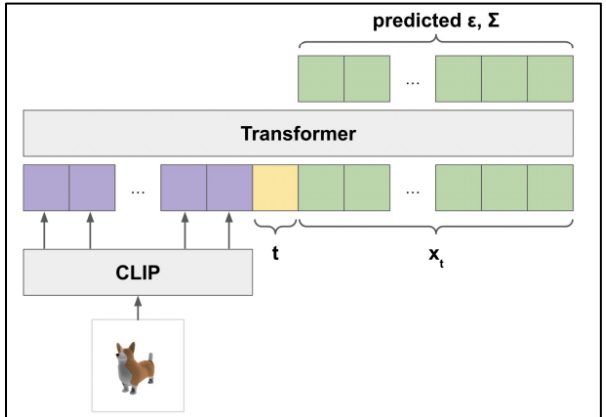
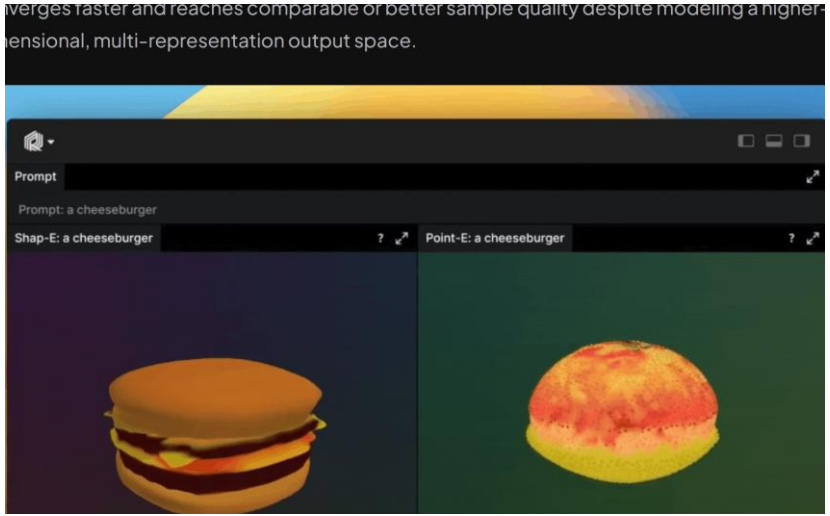


Figure 3. Our point cloud diffusion model architecture. Images are fed through a frozen, pre-trained CLIP model, and the output grid is fed as tokens into the transformer. Both the timestep t and noised input x_t are also fed in as tokens. The output tokens corresponding to x_t are used to predict ϵ and Σ.

How does Shape-E work?

- Text-to-NeRF model, with two stages
- Encoder Stage
  - Train encoder to predict parameters of a NeRF given a point cloud and its RGB views. NeRF supervised with rendering loss
- Diffusion Training
  - Learn to generate latent MLP parameters conditioned on text/image prompts



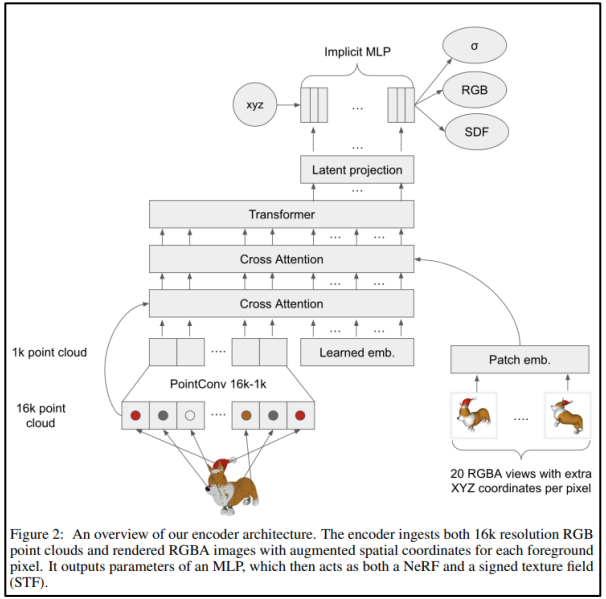


Figure 2: An overview of our encoder architecture. The encoder ingests both 16k resolution RGB point clouds and rendered RGBA images with augmented spatial coordinates for each foreground pixel. It outputs parameters of an MLP, which then acts as both a NeRF and a signed texture field (STF).

What are triplanes & k-planes?

- 3D representation that provides a more efficient alternative to 3D voxel grids
- Comprised of 3 orthogonal 2D feature planes aligned with principal axes:
  - F_{xy} aligned with XY plane, F_{xz} aligned with XZ plane, F_{yz} aligned with YZ plane
  - Each plane is a 2D grid of high dimensional feature vectors, shape $N \times N \times C$
- To sample feature at point $p = (x, y, z)$
  - Project onto F_{xy} (x,y), F_{xz} (x,z), & F_{yz} (y,z)
  - For each projection, apply bilinear interpolation to get the corresponding feature vectors f_{xy}, f_{xz}, f_{yz}
  - Aggregate features by concatenation/summation
- Final feature can be used for downstream applications, eg into a MLP to decode NeRF parameters
- Space usage is $O(N^2C)$ instead of $O(N^3C)$ for voxel grids

- Can be further generalized to K-Planes, where a d-dimensional space is turned into "d choose 2" 2D feature planes
- One key application is for 4D NeRFs, with 6 planes: xy, yz, xz, xt, yt, zt
  - NeRF predicts rgb & density given x,y,z,t,d (viewing direction)
  - Needs a multiview dynamic video dataset as supervision (Multiview video, multiple cameras capturing a scene over time)
  - A 4D point (x,y,z,t) is projected onto all 6 planes; features bilinearly interpolated and multiplied (Hadamard product)
  - This is done repeatedly through volumetric ray rendering & then supervised with photometric loss
  - K-plane formulation allows for regularizing in certain dimensions (eg Laplacian smoothness in time)

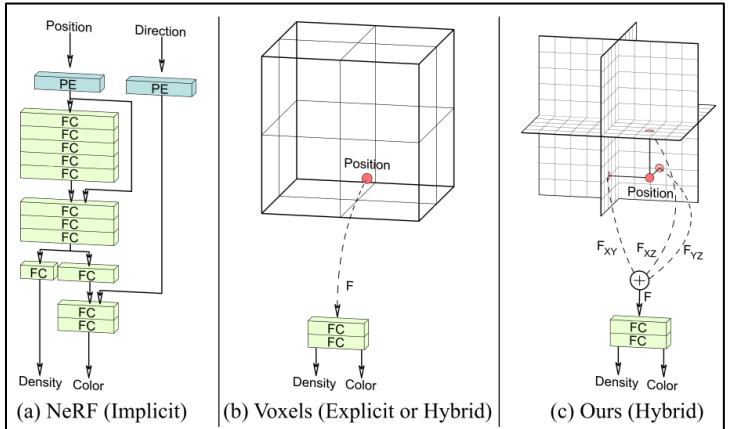
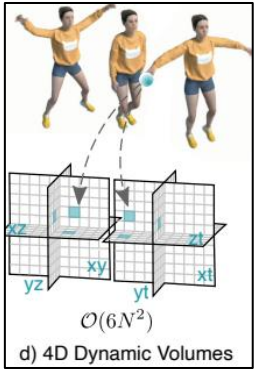


Figure 2. Neural implicit representations use fully connected layers (FC) with positional encoding (PE) to represent a scene, which can be slow to query (a). Explicit voxel grids or hybrid variants using small implicit decoders are fast to query, but scale poorly with resolution (b). Our hybrid explicit–implicit tri-plane representation (c) is fast and scales efficiently with resolution, enabling greater detail for equal capacity.



How are 4D Gaussian Splats formulated & optimized?

- Introduces a deformation field that warps a canonical, initial set of 3D gaussians over time
- First stage: a fixed set of initial gaussians are learned (eg over the first few frames)
- Second stage: Learn gaussian deformation field network
  - Hex-plane based, where 6 multi-resolution voxel planes are learned and fed into MLP to regress time-dependent changes on the canonical gaussians
  - Changes include position, rotation, scale, and optionally alpha & color
  - Deformed gaussians differentiably splatted & supervised using photometric loss
- Two kinds of data setup, requiring camera poses
  - Multiple cameras capturing the same scene simultaneously from different viewpoints -- preferred
  - Monocular video (single camera) – more challenging

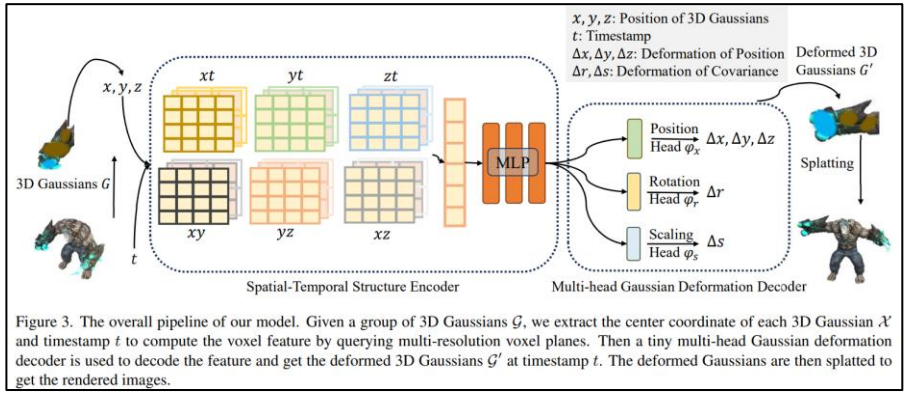
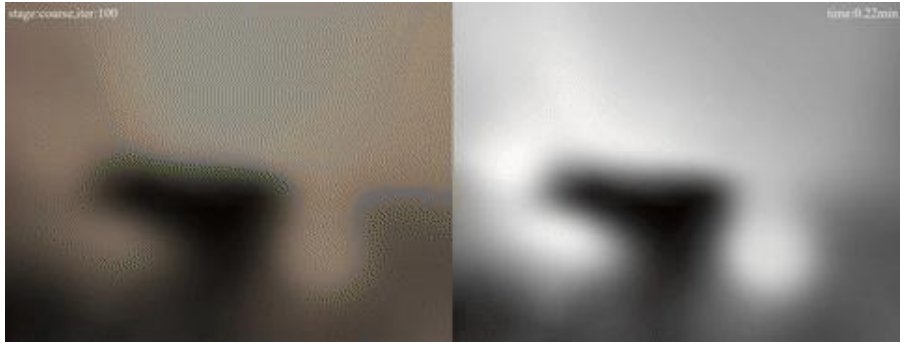
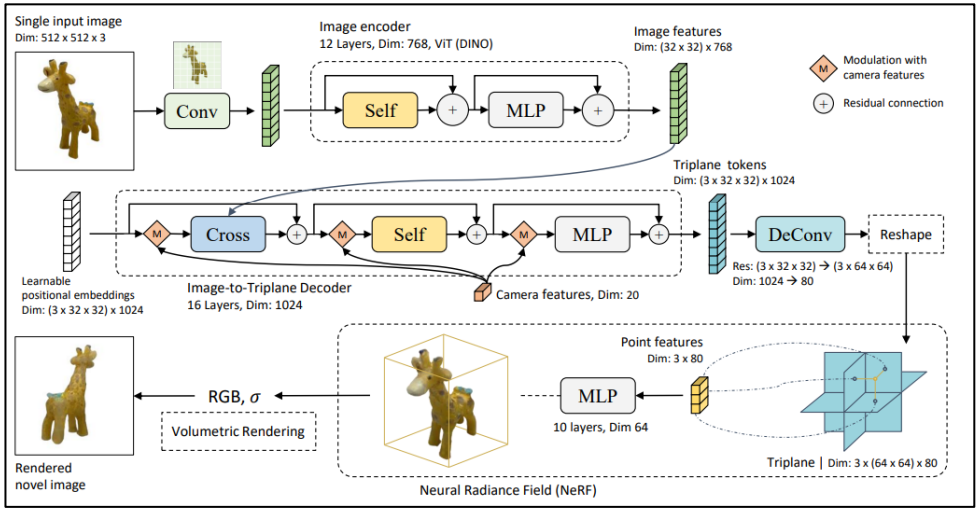


Figure 3. The overall pipeline of our model. Given a group of 3D Gaussians $\mathcal{G}$, we extract the center coordinate of each 3D Gaussian $\mathcal{X}$ and timestamp t to compute the voxel feature by querying multi-resolution voxel planes. Then a tiny multi-head Gaussian deformation decoder is used to decode the feature and get the deformed 3D Gaussians $\mathcal{G}'$ at timestamp t. The deformed Gaussians are then splatted to get the rendered images.



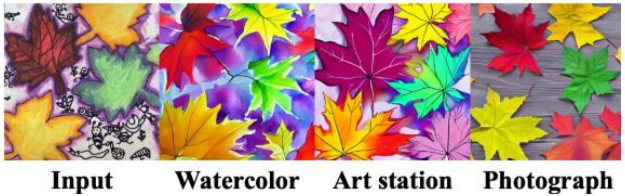
How does LRM work?

- Single-view 3D reconstruction at scale
  - Trained end-to-end on a million 3D models, mix of real & synthetic data
- Three main steps:
  - **Pre-trained DINO ViT** processes images into patches
  - **Image-To-Triplane decoder** converts 2D features into a triplane representation using cross-attention & camera-conditioned modulation using extrinsics & intrinsics
  - **Triplane NerF**: MLP predicts RGB & density from queried triplane features. Random novel views rendered from this and supervised using RGB loss
- Training feeds in camera embedded features (from intrinsics/extrinsics); at test time, we feed a fixed vector
  - Feeding this in during training led to better convergence
  - A trade-off; sometimes leads to degradation in performance



How does SDEdit work?

- Framework for guided image generation/editing, derived using the score-based continuous-time SDE formulation of diffusion
- Can do stroke-based editing as well as style transfer
- Method: by “hijacking”/exploiting the fact that diffusion models can be run from arbitrary noise levels (not just from pure noise), we can intentionally corrupt the guide and denoise towards a sample that lies both near the guide and on the natural image manifold
  - Eg, the colors used in the stroke painting matters – influences types of objects generated
  - Similarly, for style transfer, we destroy low-frequency noise (style) but keep high-frequency noise (structure). Similar intuition to how diffusion models incrementally add details.
- Uses a standard pretrained model & does not change weights



How does CAT4D work?

Given a monocular video (real or generated), generates a 4D gaussian using 3 steps:

Step 1: Multi-view Video Diffusion Model Training (monocular video to multi-view videos)

- Takes in M input conditional set of images I^{cond}, poses P^{cond}, & times T^{cond}; trained to learn distribution of N target images I^{tgt} given their camera parameters P^{tgt} & times T^{tgt} : $p(I^{tgt} \mid I^{cond}, P^{cond}, T^{cond}, P^{tgt}, T^{tgt})$
- Based off of CAT3D: multi-view latent diffusion model, U-Net based with 3D self-attention to connect multi-view latents
- Uses classifier-free guidance with two components to disentangle camera & time

Step 2: Generating Consistent Multi-View Videos

- Uses an **Alternating Sampling Strategy** to generate large consistent multiview video grids (many cameras x timesteps)
- Uses a sliding-window approach to gradually build full grid to enforce viewpoint & time consistency
- First, Multi-View sampling is done (fix timestep, generate multiple consistent views w/ sliding window)
- Then, Temporal Sampling is done (fix camera, generate timestamps w/ sliding window)
- Above procedure can be iterative refined using SDEdit – partial refinement & consistency corrections (instead of regenerating from scratch every time)

Step 3: 4DGS generation

- Use the multi-view videos and learn a gaussian splat along with deformation field

Data-driven 4D prior is difficult to build due to lack of diverse data; approach here is to use combinations of synthetic 4D data, fixed camera videos, and object-centric video captures. The latter two is augmented through CAT3D (to provide camera motion) and Lumiere (to provide object motion) respectively

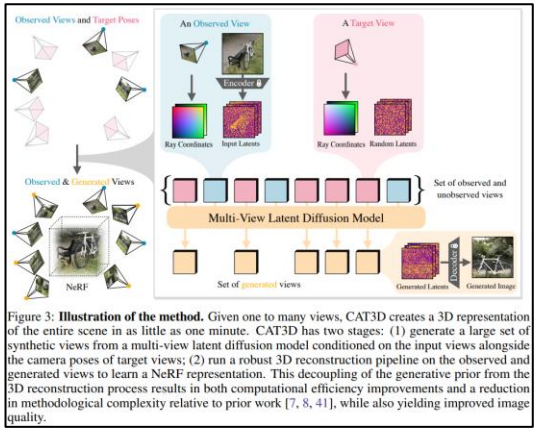


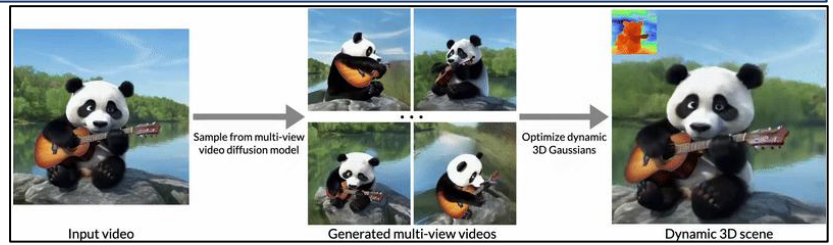
Figure 3: **Illustration of the method.** Given one to many views, CAT3D creates a 3D representation of the entire scene in as little as one minute. CAT3D has two stages: (1) generate a large set of synthetic views from a multi-view latent diffusion model conditioned on the input views alongside the camera poses of target views; (2) run a robust 3D reconstruction pipeline on the observed and generated views to learn a NeRF representation. This decoupling of the generative prior from the 3D reconstruction process results in both computational efficiency improvements and a reduction in methodological complexity relative to prior work [7, 8, 41], while also yielding improved image quality.

$$\hat{\epsilon}_\theta(z, P_{\text{tgt}}, c_I, c_T) = \underbrace{\epsilon_\theta(z, P_{\text{tgt}}, \emptyset, \emptyset)}_{\text{unconditioned}} + s_I \cdot \left[\underbrace{\epsilon_\theta(z, P_{\text{tgt}}, c_I, \emptyset)}_{\text{view-guided}} - \epsilon_\theta(z, P_{\text{tgt}}, \emptyset, \emptyset) \right] + s_T \cdot \left[\underbrace{\epsilon_\theta(z, P_{\text{tgt}}, c_I, c_T)}_{\text{full-guided}} - \epsilon_\theta(z, P_{\text{tgt}}, c_I, \emptyset) \right]$$

- First term: **unconditional prediction**
- Second term: encourages the sample to align with the **camera/view input**
- Third term: encourages the sample to align with the **temporal input**

$$c_I = \{I_{\text{cond}}, P_{\text{cond}}\}$$

$$c_T = \{T_{\text{cond}}, T_{\text{tgt}}\}$$



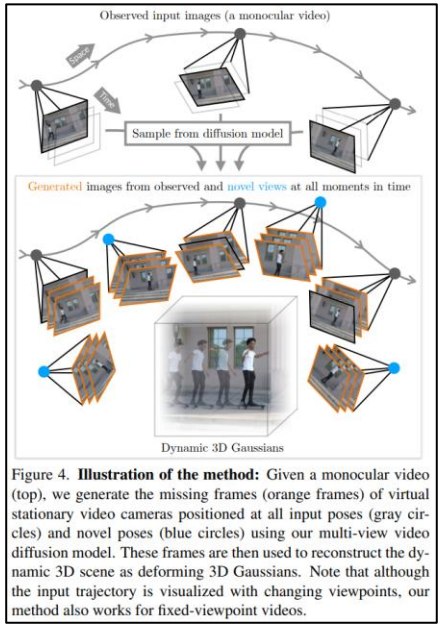
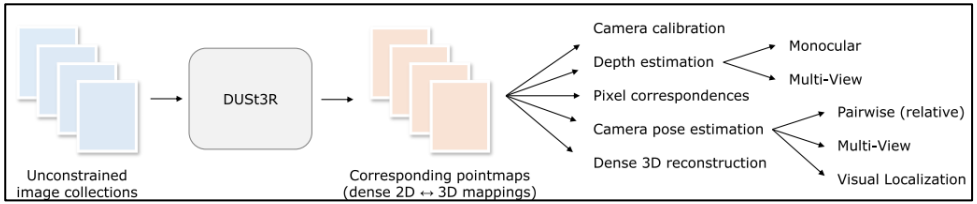


Figure 4. **Illustration of the method:** Given a monocular video (top), we generate the missing frames (orange frames) of virtual stationary video cameras positioned at all input poses (gray circles) and novel poses (blue circles) using our multi-view video diffusion model. These frames are then used to reconstruct the dynamic 3D scene as deforming 3D Gaussians. Note that although the input trajectory is visualized with changing viewpoints, our method also works for fixed-viewpoint videos.

How does Dust3r work?

- Unified, general, simple model that tackles a lot of 3D vision tasks which were historically fragmented
- Predicts dense pointmaps (2D grid where each pixel maps to a 3D point) directly from image pairs, w/o intrinsics/poses
  - Pointmap $\in \mathbb{R}^{W \times H \times 3}$
  - Crucially, both pointmaps from a stereo pair are expressed/aligned in the same coordinate frame, for easy geometric operations
  - Similar to a dense 3D pointcloud, but with one point per pixel; point corresponding to pixel approximately lies along that pixel's viewing ray
- Also predicts confidence map (for where point is not well defined, like the sky)
- Downstream tasks using pointmaps:
  - **Camera calibration**
    - If we assume principal point is centered & pixels are squares, we can easily solve for focal length
  - **Camera pose estimation**
    - Compare pointmaps $X^{1,1}$ & $X^{1,2}$ and solve optimization problem to find extrinsics that aligns them (pointmap for the same image, but in different coordinate frames by swapping the input order)
    - Alternatively, use PnP + RANSAC
  - **Depth estimation**
  - **Pixel correspondences**
  - **Dense 3D reconstruction**
- Trained using a combination of real & synthetic RGBD data



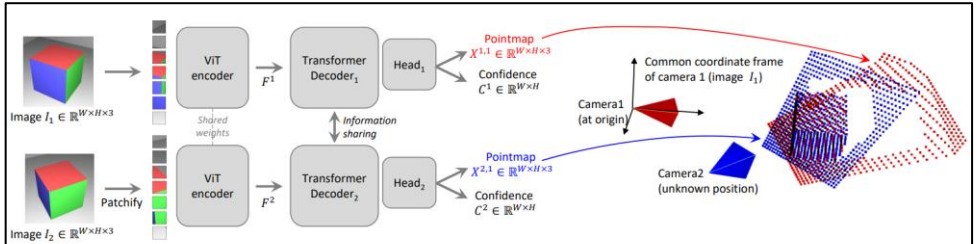


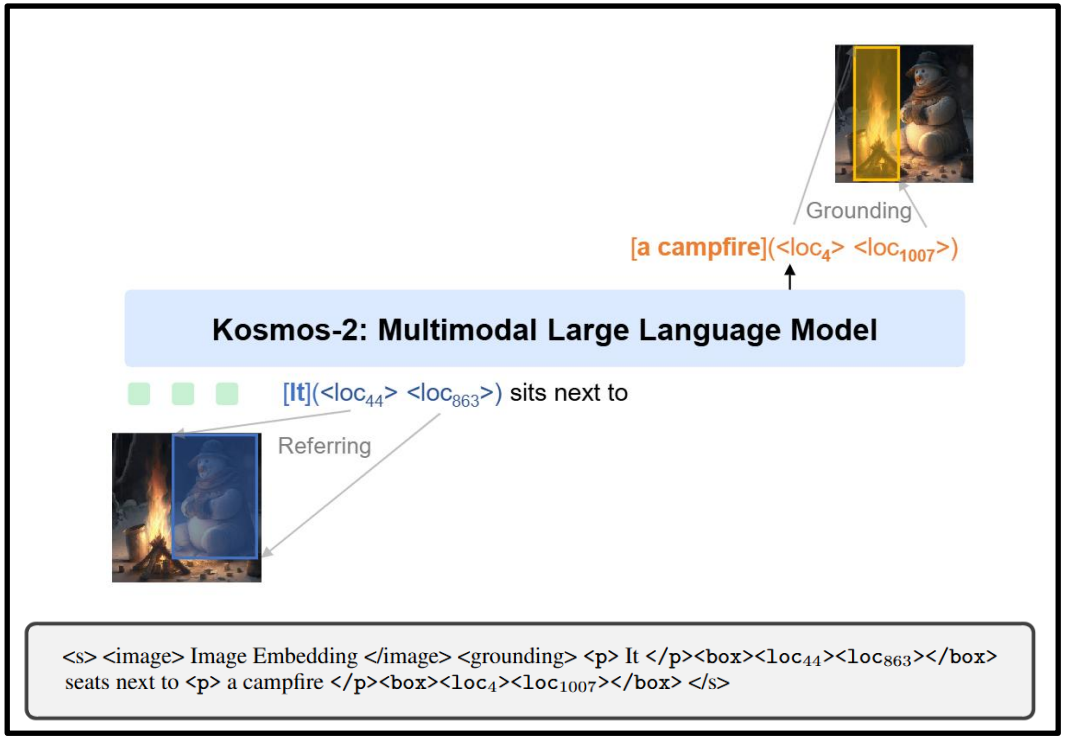
Figure 2. **Architecture of the network $\mathcal{F}$.** Two views of a scene (I^1, I^2) are first encoded in a Siamese manner with a shared ViT encoder. The resulting token representations F^1 and F^2 are then passed to two transformer decoders that constantly exchange information via cross-attention. Finally, two regression heads output the two corresponding pointmaps and associated confidence maps. Importantly, the two pointmaps are expressed in the same coordinate frame of the first image I^1. The network $\mathcal{F}$ is trained using a simple regression loss (Eq. (4))

Figure 3. **Reconstruction examples** on two scenes never seen during training. From left to right: RGB, depth map, confidence map, reconstruction. The left scene shows the raw result output from $\mathcal{F}(I^1, I^2)$. The right scene shows the outcome of global alignment (Sec. 3.4).

Vision Language Models

Describe the KOSMOS-2 model.

- Multimodal LLM designed to ground text to images by linking phrases to bounding boxes
  - Here, grounding refers to explicitly establishing connections to real-world natural language concepts; can mitigate hallunications & provide interpretability
- Trained on new GRIT dataset, with 90M+ image-text-bbox pairs
- Uses special tokens to relate text and image regions; uses prompt templates to train models to use these tokens
  - Overall trained on image-text pairs, monomodal text corpora, image-caption pairs, and interleaved image-text data
  - Relatively small dataset ; posttrain takes 1 day.



- "What is it <box><loc1><loc2></box>? It is {*expression*}."
- "What is this <box><loc1><loc2></box>? This is {*expression*}."
- "Describe this object <box><loc1><loc2></box>. This object is {*expression*}."
- " It <box><loc1><loc2></box> is {*expression*}."
- " This <box><loc1><loc2></box> is {*expression*}."
- " The object <box><loc1><loc2></box> is {*expression*}."

Table 9: Instruction templates used for expression generation.

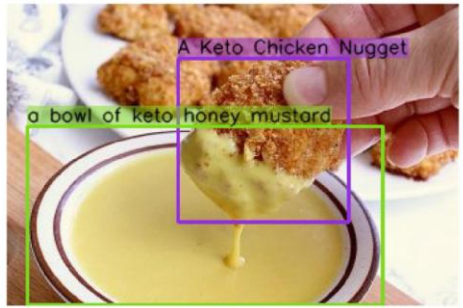
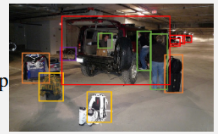


Figure 7: Example from GRIT. Caption: "*A Keto Chicken Nugget being dipped into a bowl of keto honey mustard.*".

Describe the LLaVA model.

- Endows a text-only LLM with image reasoning abilities via instruction tuning, to build a general-purpose visual assistant
- Takes the text-only Vicuna model (a fine-tuned LLaMA), and adds a CLIP ViT visual encoder & projection layer
- Multi-turn VLM dataset generated by combining captioned image data with text-only GPT-4
  - Feeds in captions & bboxes as input
  - Asks conversation, description, and reasoning questions to get answers
  - These then become GT for the LLaVA dataset

Context type 1: Captions
A group of people standing outside of a black vehicle with various luggage.
Luggage surrounds a vehicle in an underground parking area
People try to fit all of their luggage in an SUV.
The sport utility vehicle is parked in the public garage, being packed for a trip
Some people with luggage near a van that is transporting it.



Context type 2: Boxes
person: [0.681, 0.242, 0.774, 0.694], backpack: [0.384, 0.696, 0.485, 0.914], suitcase: ...<omitted>

Response type 1: conversation
Question: What type of vehicle is featured in the image?
Answer: The image features a black sport utility vehicle (SUV) ...<omitted>
Response type 2: detailed description
The image is an underground parking area with a black sport utility vehicle (SUV) parked. There are three people in the scene, with one person standing closer to the left side of the vehicle, another person in the middle, and the third person on the right side. They are all working together to pack their luggage into the SUV for a trip. ...<omitted>
Response type 3: complex reasoning
Question: What challenges do these people face?
Answer: In the image, a group of people is standing outside a black SUV in a parking area, surrounded by various pieces of luggage, including suitcases and backpacks. They are facing the challenge of fitting all their luggage into the black SUV. There are multiple suitcases and backpacks to be packed, which suggests that the group has a significant amount of belongings ...<omitted>

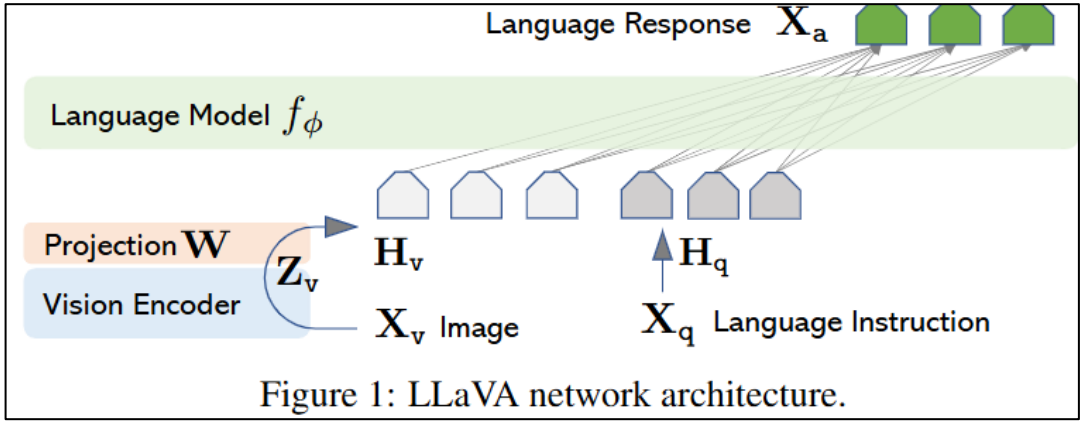
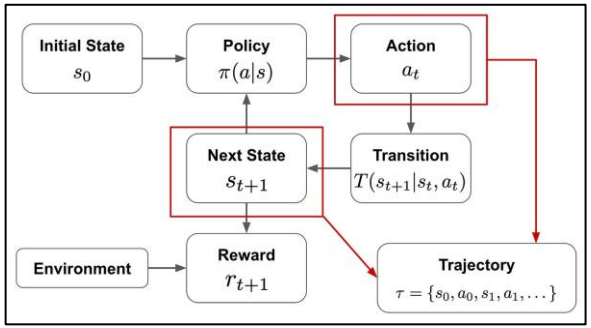


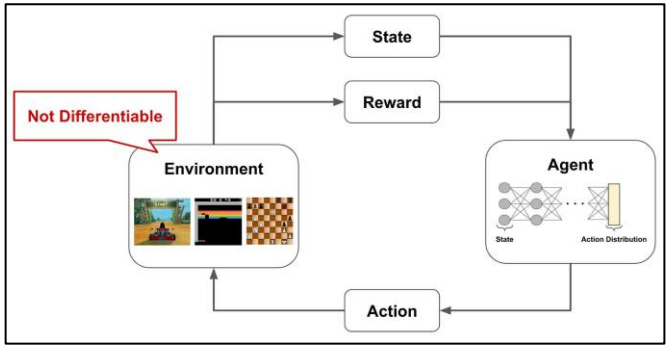
Figure 1: LLaVA network architecture.

Reinforcement Learning

At a high level, what is the standard setup for training RL models?

- Similar to how humans learn, RL trains neural networks through trial and error
  - Producing outputs, receiving feedback, and adjusting weights
  - Agent learns to make decisions by interacting with an environment in order to maximize a reward signal over time
- RL generally the agent takes actions in an environment and indirectly receives delayed and sparse feedback in the form of rewards, without being told the correct action.
  - It's told how good the action was, but not the exact correct action; model has to discover this through exploration and learning
  - Generally environment is a black box and is not differentiable
  - In contrast, in supervised learning the feedback is direct & correct, and everything is typically backpropagable
  - Rewards is not associated with every action; they can have a long horizion and require several correct actions to generate any positive reward
- Formally modeled by a Markov Decision Process (MDP)
  - $s \in S$, **State**
    - Set of all possible situations the agent can find itself in
    - Eg in robotics the position/velocity/orientation; in video game, the frame pixels
  - $a \in A$, **Action**
    - Eg {up, down, left, right} or {accelerate, brake, turn left, etc}
  - $r_t \in \mathbb{R}$, **Reward from reward function** $R(s_t, a_t, s_{t+1})$
    - Gives agent a reward when transitioning from s_t to s_{t+1} via action a_t
  - $T(s_{t+1}|s_t, a|t)$, **Transition**
    - Defines dynamics of the environment
  - $\pi(a|s)$, **Policy**
  - $\tau \in \{s_0, a_0, s_1, a_1, \ldots, s_t, a_t\}$, **Trajectory**
  - $R(\tau) = \sum_t \gamma^t r_t$, **Return**
    - $\gamma^t \in [0,1]$ is the discount factor, which can prefer near-term results





What are some strengths and weaknesses of RL?

- Strengths
  - **Learns by interaction:**
    - Through trial and error, ideal for sequential decision-making tasks
  - **No need for labeled data:**
    - Only need a reward signal, ideal for cases where labeling is impractical or not well-defined
  - **Long term planning:**
    - Considers delayed rewards and can optimize actions over time, enabling strategic decision-making
  - **Has had success in complex tasks, eg AlphaGo**

- Weaknesses
  - **Sample inefficiency:**
    - Requires many interactions with environment to learn effectively, which is usually only possible in sim. Due to large action/state spaces.
  - **Exploration vs. exploitation:**
    - Balancing trying new actions (exploration) vs. sticking with known good ones (exploitation) is non-trivial and can hinder learning.
  - **Reward design is hard:**
    - Poorly shaped or sparse rewards can lead to undesired behaviors or very slow learning.
  - **General instability and sensitivity**

What's the difference between on-policy & off-policy RL?

On-policy:

- Same policy used to generate behavior (collect data) and to improve itself
- Learns from data it generates by following its current policy
- Generally more stable training, but requires continual data collection as policy changes and less sample efficient since old data becomes stale

Off-policy:

- Agent learns about one policy (target policy) using data collected from a different policy (behavior policy)
- More sample-efficient (can reuse past experiences, eg experience replay)
- Enables learning from humans/expert demonstrations
- More complex / tricky, can need corrections like importance sampling

Derive the vanilla/basic policy gradient, and intuitively explain what it means.

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim(\pi_\theta,T)}\left[R(\tau)\right]$$

Trajectory, Return, Policy, Transition Function

Using chain rule of probability and log-derivative trick:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathbb{E}_{\tau\sim(\pi_\theta,T)}\left[R(\tau)\right] \\ &= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) \\ &= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau)\end{aligned}$$

$$P(\tau|\theta_t) = \rho(s_0)\prod_{t=0}^{T} T(s_{t+1}|s_t,a_t)\pi_{\theta_t}(a_t|s_t)$$

Probability of Initial State, Policy, Trajectory Probability, Transition Function

Basic Policy Gradient

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim(\pi_\theta,T)}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right]$$

$$\mathcal{D} = \{\tau_0,\tau_1,\dots\} \quad \text{(Sampled Trajectories)}$$

$$\overline{\nabla_\theta J(\pi_\theta)} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\sum_{t=0}^{T}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right]$$

$$\theta_{t+1} = \theta_t + \alpha\overline{\nabla_\theta J(\pi_\theta)|_{\theta_t}}$$

Gradient of objective, Learning Rate

- Tells us how to adjust the policy's parameters to make actions that led to good outcomes more likely in the future
- ∇_θ tells us how the probability of particular actions a_t change with respect to the model parameters θ
- Combined effect depends on $R(\tau)$
  - If positive, then we increase chance of choosing that action
  - If negative, then we decrease chance of choosing that action
- Gradient shifts probability mass to push agent to repeat lucky successes, based on observed outcomes
- Note, VPG is very data inefficient

Define On-Policy/Optimal Value/Action-Value functions, and how they lead to advantage functions.

- ***On-Policy Value Function***: $V^{\pi}(s) = \mathbb{E}_{\tau \sim (\pi, T)} [R(\tau) | s_0 = s]$
- Expected return if you start in state s and act according to policy π afterwards.

- ***On-Policy Action-Value Function***: $Q^{\pi}(s, a) = \mathbb{E}_{\tau \sim (\pi, T)} [R(\tau) | s_0 = s, a_0 = a]$
- Expected return if you start in state s, take some action a (may not come from the current policy), and act according to policy π afterwards.

- ***Optimal Value Function***: $V^{\star}(s) = \max_{\pi} \mathbb{E}_{\tau \sim (\pi, T)} [R(\tau) | s_0 = s]$
- Expected return if you start in state s and always act according to the optimal policy afterwards.

- ***Optimal Action-Value Function***: $Q^{\star}(s, a) = \max_{\pi} \mathbb{E}_{\tau \sim (\pi, T)} [R(\tau) | s_0 = s, a_0 = a]$
- Expected return if you start in state s, take some action a (may not come from the current policy), and act according to the optimal policy afterwards.

- The advantage function measures how much better/worse an action is compared to the average action in a given state

$$A^{\pi}(s, a) = \underbrace{Q^{\pi}(s, a)}_{\text{On-Policy Action-Value Function}} - \underbrace{V^{\pi}(s)}_{\text{On-Policy Value Function}} \quad \text{(Advantage Function)}$$

- Used in the bellman equations, which say that the value of a state is just the immediate reward plus the value of the future
- Encodes the recursive nature of reward accumulation and form the basis for how most RL algorithms incremental improve estimates.

$$V^{\pi}(s) = \mathop{\mathrm{E}}_{\substack{a \sim \pi \\ s' \sim P}} [r(s, a) + \gamma V^{\pi}(s')],$$

$$Q^{\pi}(s, a) = \mathop{\mathrm{E}}_{s' \sim P} \left[r(s, a) + \gamma \mathop{\mathrm{E}}_{a' \sim \pi} [Q^{\pi}(s', a')] \right],$$

$$V^{*}(s) = \max_{a} \mathop{\mathrm{E}}_{s' \sim P} [r(s, a) + \gamma V^{*}(s')],$$

$$Q^{*}(s, a) = \mathop{\mathrm{E}}_{s' \sim P} \left[r(s, a) + \gamma \max_{a'} Q^{*}(s', a') \right].$$

What is Generalized Advantage Estimation (GAE)?

$$A^{\pi}(s,a) = \underbrace{Q^{\pi}(s,a)}_{\text{On-Policy Action-Value Function}} - \underbrace{V^{\pi}(s)}_{\text{On-Policy Value Function}} \quad \text{(Advantage Function)}$$

- In general, we can't directly compute the advantage function – it's intractable
- Temporal distances (TD) only involve a one-step lookahead estimate, and everything is available at time t to get δ_t
- GAE takes this a step further & blends multiple estimates using decaying weights, to place more weigh on short-term signals

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

- In practice, a neural network V_ϕ regresses the value function using MSE loss
- Data source is sampled trajectories; tuples of $(s_t, a_t, r_{t+1}, s_{t+1})$
- The supervision target is $\hat{y} = r_{t+1} + \gamma V_\phi(s_{t+1})$
  - Bootstrapping technique: over time, will converge to a fixed point that satisfies the bellman equation
  - Not chicken-and-egg, even though V_ϕ appears on both sides of the target

$$\mathcal{L}(\phi) = \left(V_\phi(s_t) - \hat{y}_t\right)^2$$

What is Trust Region Policy Optimization (TRPO)?

- Motivated by how data inefficient VPG is, TRPO updates the policy under a constraint based on the KL divergence. Some notes:
  - Instead of performing gradient descent, we solve a constrained maximization problem to generate each new policy. Performs larger updates
  - KL divergence in parameter space prevents overly large steps
  - Term in expectation modified to express probability of an action as the ratio between old and updated policy
  - Advantage function is used instead of just the award function, to focus updates only on better-than-expected actions; helps policy learn to favor relatively good actions w/o overacting to randomness
- Follows an actor-critic model with 2 different neural networks
  - **Actor**
    - Represents policy $\pi_\theta(a|s)$, optimized via an estimate of the advantage
  - **Critic**
    - Approximates value function V, which estimates how good a state is under the current policy
    - Trained using regression (using temporal difference TD error / generalized advantage estimation GAE)
    - Helps train actor by providing advantage estimates

Advantage Function

$$\theta_{k+1} = \underset{\theta}{\text{argmax}}\ \mathbb{E}_{(s,a)\sim(\pi_{\theta_k},T)}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a)\right]$$

$$\text{such that } \overline{D}_{\text{KL}}(\theta||\theta_k) < \delta$$

KL Divergence

What is Proximal Policy Optimization (PPO)?

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min(r_t(\theta)A_t, \text{CLIP}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right] \qquad r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$$

- Improvement upon TRPO that is simpler & converges faster
- Like TRPO, uses ratio r_t; >1 when action is more probable under currently policy than old policy, and in [0,1) when less probable
- However, this ratio can lead to overly large gradient steps. One way to mitigate is to build stabilizing properties into a surrogate objective function
  - Clip limits ratio to a range, eg (0.8, 1.2)
  - We build an asymmetric feature via the clip & min components -- that if we do the right thing, don't take to big of a step, but if we do the wrong thing, we are free to take a larger step
- Because of the designed objective, we can run multiple epochs of gradient ascent on your samples. As we run more, we'll start hitting clipping limits, so training will gradually stop until we move onto the next iteration to collect new samples

Algorithm 1 PPO, Actor-Critic Style

```
for iteration=1, 2, ... do
    for actor=1, 2, ..., N do
        Run policy π_θold in environment for T timesteps
        Compute advantage estimates Â_1, ..., Â_T
    end for
    Optimize surrogate L wrt θ, with K epochs and minibatch size M ≤ NT
    θ_old ← θ
end for
```

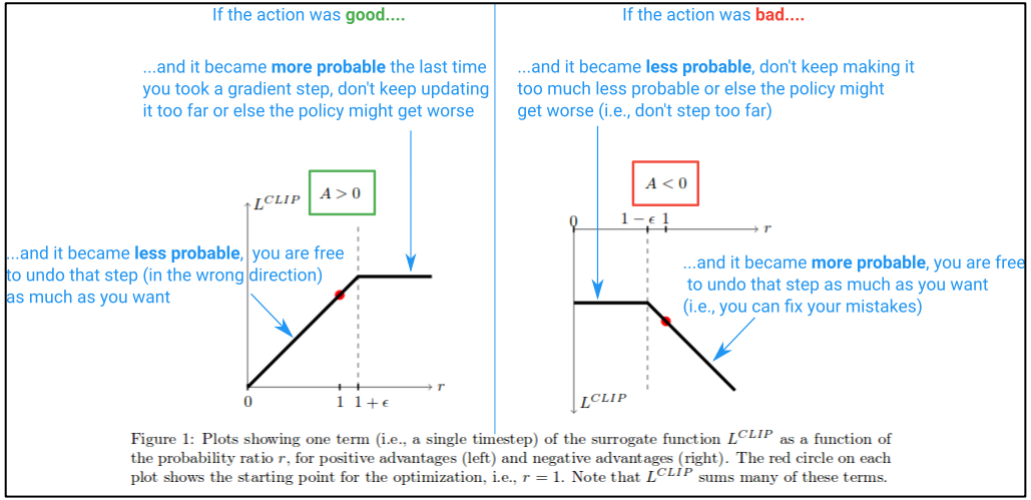
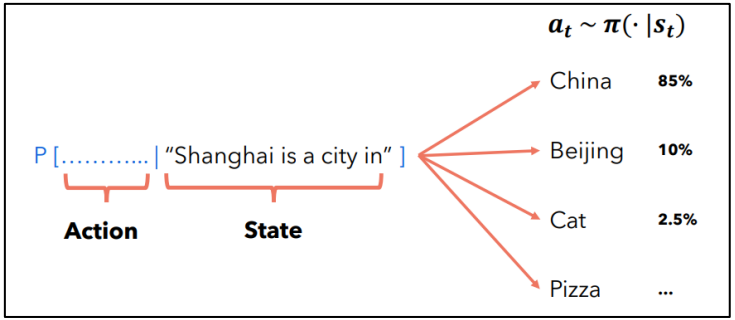


Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{CLIP} sums many of these terms.

How can LLM alignment be formulated as an RL problem?

- RL allows us to align LLM outputs to human preferences and generate more helpful responses
  - In this final stage, we don't have concrete GT targets; there can be multiple plausible outputs, so the RL framework makes sense
  - Also, the autoregressive sampling process (eg greedy, beam search, top-k, etc) is not differentiable; output text is discrete
- Data: Instead of annotating raw reward scores of independent prompts/answers (which is not well-defined and flakey), annotate relative pairwise preferences with two answers
  - Tuples of $(prompt, x^+, x^-)$
- With data, train a transformer reward model R_ϕ to regress reward
  - Loss: $L_{RM} = -log\sigma\left(R_\phi(x^+) - R_\phi(x^-)\right)$
- Formulation:
  - **States**: The language model prompts
  - **Action**: The next token chosen autoregressively
  - **Policy/Agent**: The LLM itself; $\pi_\theta(a|s)$
  - **Episode**: prompt → sampled response → award
- Optimization:
  - Sample prompts, and generate multiple responses per prompt
  - Compute their rewards using reward model
  - Compute advantage estimates: how much better a response was compared to expected using a learned critic model $V_\phi(s_t)$
    - Advantages $\hat{A}_t$ are per-response-timestep, and tell us how much better or worse that token choice was than expected
    - Estimates which tokens contributed more/less to the final award & a good/bad response, enabling credit assignment
    - Gives us a way to assign credit across timestamps even when only the full sequence is awarded
  - Update policy using PPO's surrogate objective (allows for small steps, less chance for reward hacking)
    - By treating the reward model as a black-box function & not backpropagating through it, we also help mitigate reward model hacking



$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l}$$

What is Direct Preference Optimization (DPO)?

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta\log\frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\right)\right]$$

- PPO can be difficult to use when aligning LLM models
  - Requires training a reward model
  - Training takes longer, need to generate/sample in-the-loop during training
  - Involves more complexity, RL machinery, tuning
- DPO allows us to align LLMs without RL
  - Directly use pairwise preference annotations and optimize using gradient descent
- In loss function, we have a negative log sigmoid, so we want the winning ratio to be high, and the losing ratio to be low
  - Probabilities $\pi(y|x)$ computed by multiplying per-token probabilities when running through LLM
  - Loss derived from the Bradley-Terry model, where $y_w > y_l$ reads as "y_w preferred to y_l"
- Some weaknesses of DPO compared to PPO:
  - Cannot train on arbitrary unannotated prompts (no reward models)
  - DPO is off-policy training (model not exposed to its own policy during training)
  - Limited to pairwise comparisons, cannot optimize scalar rewards

$$P(y_w > y_l) = \frac{e^{r^*(x,y_w)}}{e^{r^*(x,y_w)} + e^{r^*(x,y_l)}}$$

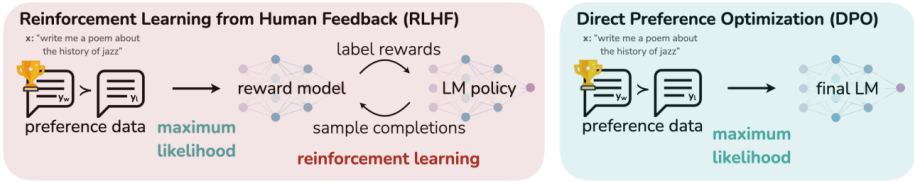
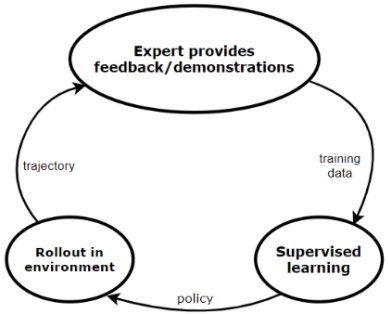
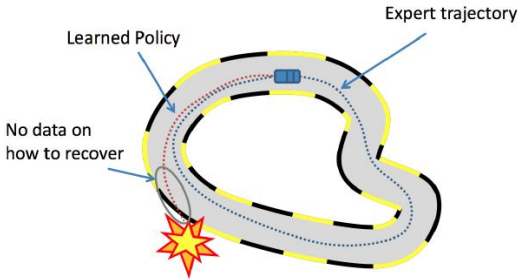


Figure 1: **DPO optimizes for human preferences while avoiding reinforcement learning.** Existing methods for fine-tuning language models with human feedback first fit a reward model to a dataset of prompts and human preferences over pairs of responses, and then use RL to find a policy that maximizes the learned reward. In contrast, DPO directly optimizes for the policy best satisfying the preferences with a simple classification objective, fitting an *implicit* reward model whose corresponding optimal policy can be extracted in closed form.

What is imitation learning?

- In imitation learning, an agent learns by observing & mimicking expert behavior
- Simplest form is **behavior cloning**, which uses supervised learning over a dataset of expert demonstrations (state-action pairs)
  - There are no explicit rewards, unlike RL. Treats the expert to learn the implicit reward function
  - Not trial-and-error based; does not have the same potential for superhuman performance that RL agents can discover
- Extensions use techniques from RL to improve generalizability
  - **Dagger (Dataset Aggregation)** iteratively gathers data, trains a policy, and deploys the policy in the environment to collect new states for the expert to annotate
  - **Inverse RL** learns a reward function to explain the expert's behavior, and then uses RL to find a policy
- Applicability
  - IL: Scenarios where solution is complex but can be demonstrated by expert. Reward is sparse and not well defined
  - RL: Good for problems where reward structure is clearly defined but solution is complex/unknown





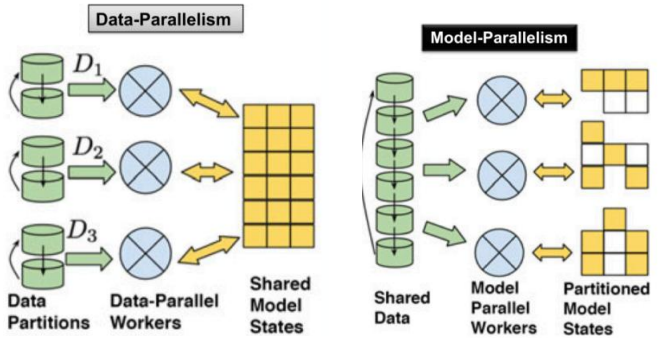
Full Stack Deep Learning

https://fullstackdeeplearning.com/

[Training & Debugging Phase] Compare data vs model parallelism when doing distributed training.

Distributed training is a must for large datasets and large models, and involves multiple GPUs and/or machines to train a single model. Note that forward computations and backprop are inherently serial in nature, so one can think of these methods as “workarounds”.

- **Data parallelism**
  - Helpful for large datasets
  - Splits batch into multiple GPUs (each gpu must fit a copy of the model on it), evaluate/backprop on each, and combine to get overall gradient to update across all GPUs
  - Generally simple to do
  - Hogwild paper (NIPS 11, 2000+ cit) showed that you don’t even need locks; SGD works even if the shared model sometimes receives older gradients.
- **Model Parallelism**
  - Needed when a model does not fit on a GPU
  - Splits the model parameters on separate GPUs
  - Introduces a lot of complexity, so see if this can be avoided by getting larger GPUs
  - One way this is implemented (AWS Sagemaker) is by splitting consecutive layers into different GPUs, and having an execution pipeline where different devices can work on forward and backward passes for different data samples at the same time



[Testing Phase] Draw a diagram of an ML system, and how you can test it.

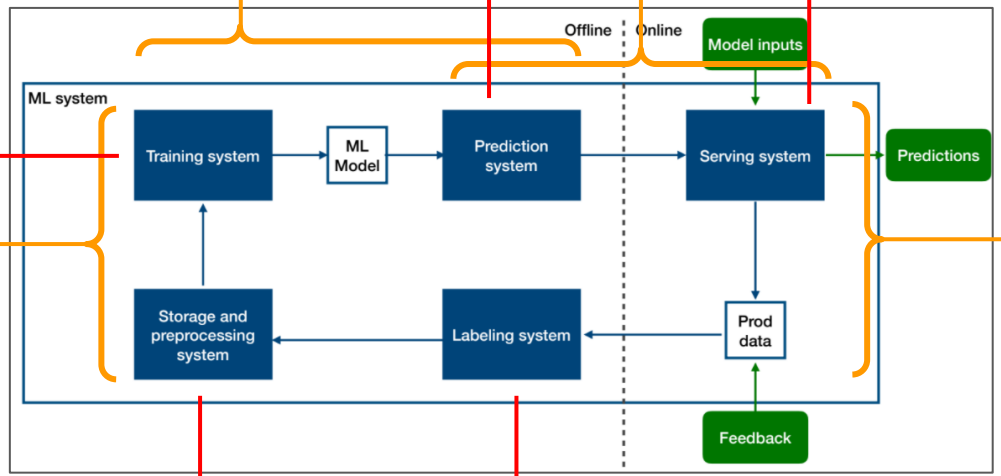
Evaluation Testing
- Fully understand performance characteristics; see if guardrails are necessary
- Eval on all metrics, datasets, and **slices**
- Invariance robustness testing
- Compare to previous models, using thresholds. Avoid performance "degradation leaks".
- Simulation testing,if possible (needs rendering)
- Test for privacy, fairness, bias

Basic Pred. Tests
- Checks basic prediction functionality
- Load pretrained model, test on few key examples

Shadow Tests
- Checks prod. bugs and compare performance w/ old model
- Run new model in parallel with old model in production, but in "shadow mode"
- Unlike dev/sandbox env, more realistic

Training Infrastructure Tests
- Avoid bugs in training pipeline
- Train on a single batch or epoch, to make sure it can overfit

A/B Tests
- Understand how new model affects user experience and business metrics
- Canary model on a tiny fraction of incoming requests
- Compare metrics on control and variation cohort

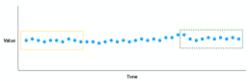


Training Tests
- Makes sure training is reproducible with current architecture
- Pull a fixed dataset and train it
- Check to make sure performance is consistent or better than before

Production Monitoring
- Want to keep model healthy in deployment
- Extremely important; what's used to produce next model and decide what new data to train on
- Often hard to do directly if you don't have labels; can instead check **business metrics** and **model inputs/predictions** for irregularities
- Need to check for bugs, outliers, class imbalance, **concept drift** (dependent var. changes)
- **Data drift** (independent var. changes) can be detected by comparing a sample of previous data and current prod. data using distance metric. The data can be features, or something more basic like HSV.
  - Don't use KL-Divergence, too sensitive and range dependent (div by zero)
  - Use **Kolmogorov-Smirnov Stat** (max dist between histogram CDF) or **D1 distance** (sum of histogram CDF distances)
  - Can experiment with more complex metrics like wasserstein dist or earth mover's dist.
  - Or, use **rule-based metrics** (w/ great expectations, e.g. min, max, mean in acceptable range, Col A > col B, etc)

Data Expectation Tests
- Catch data quality issues before training
- Check for null values, numerical ranges, and formatting using rules and thresholds
- Example package: "Great Expectations"

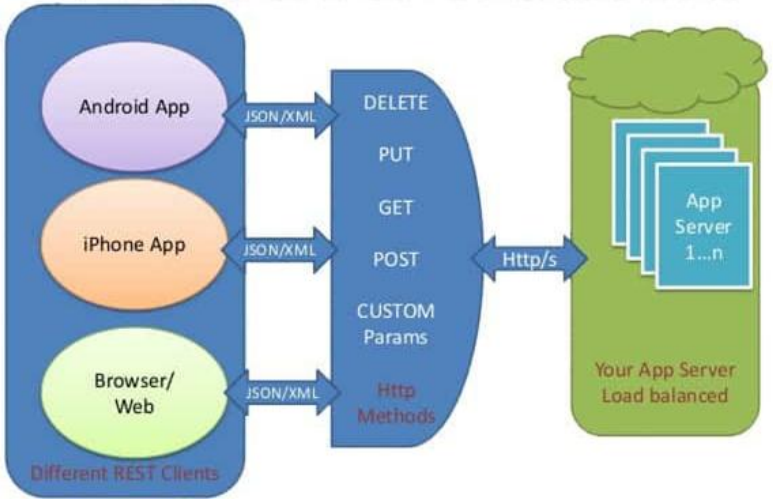
Labeling QA
- Catch poor quality labels
- Train labelers well
- Assign them a "trust score"; promote best performers to be "QA experts"
- Aggregate labels by voting



[Deployment Phase] Explain what a REST APIs is, and its advantages.

- A widely used, standardized, simple format for the access of resources and communicating over the internet, when it comes to client-server models
- Works over HTTPs and uses GET (read), POST (create), PUT (update), and DELETE, a server destination URI/URL, along with simple data formats like JSON and XML. Can also contain headers with auth data.
    - Client sends request, server sends response
- Crucially, is **stateless** on the server side; each request processed independently
    - If notions of a session are required, then this is managed on the client-side, and passed onto other services as necessary by the client.
    - This creates a clear distinction between front-end (client) and back-end (server)
- The stateless feature allows easy **horizontal scaling** (no state syncing required) and **caching**.
- Can have an actual server running 24/7, or use something like AWS Lambda (function as a service FaaS, pay-as-used but can have cold-starts sometimes. Considered “serverless” since you don’t have to manage them)
    - Typically uses a dependency management system like docker
- Can be implemented in python using packages like Flask or FastAPI

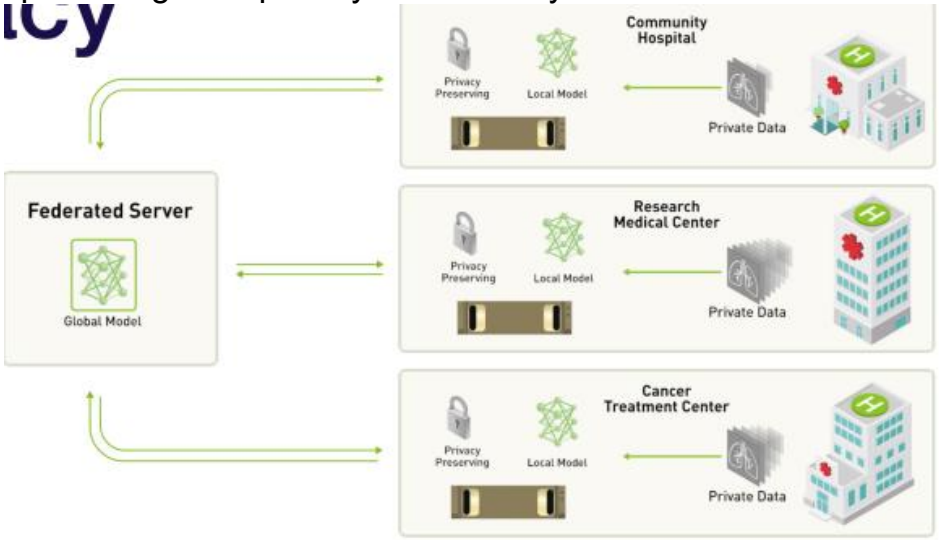
TLDR: API format over HTTP that is stateless & thus easily scalable/cacheable



[Deployment Phase] What is Federated Learning, at a high level?

- Federated Learning: training a global model from data on local devices, without ever having access to the data
- approach for collaborative machine learning without compromising data privacy and security

2. Local Training: Local devices or servers, such as smartphones or edge devices, perform training on their own data locally. This training typically consists of updating the model based on the data available on each device.
3. Model Updates: Instead of sending raw data to the central server, local devices or servers send model updates (such as gradients or weight updates) to the central server. These updates represent the changes needed to improve the global model.
4. Aggregation: The central server collects these model updates from various devices and aggregates them to update the global model. This can be done using techniques like federated averaging, where the central server averages the updates from multiple devices to update the global model.



Machine Learning Implementation

What is the general skeleton for optimizing a neural network?

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5, stride=1, padding=0) # input, output, kernel size
        self.pool = nn.MaxPool2d(2, 2) # kernel size, stride
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    # x should be of a batch of 3 x 32 x 32 images
    # 32 -> 28 -> 14-> 10 -> 5
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

torch.manual_seed(0)

# setting up data
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5))])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)
```

```
device = torch.device('cuda:0')
net = Net()
net.to(device)
net.train()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss()
loss_record = []

# Training
for epoch in range(2):
    for i, data in enumerate(tqdm(trainloader)):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs.to(device))
        loss = criterion(outputs, labels.to(device))
        loss.backward()
        optimizer.step()
        loss_record.append(loss.item())
torch.save(net.state_dict(), './cifar_net.pth')
# net.load_state_dict(torch.load('./cifar_net.pth'))

# Testing
correct = 0
total = 0
net.eval()
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images.to(device))
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels.to(device)).sum().item()
print('Accuracy of the network on the 10000 test images: {}'.format({100 * correct // total}))
```

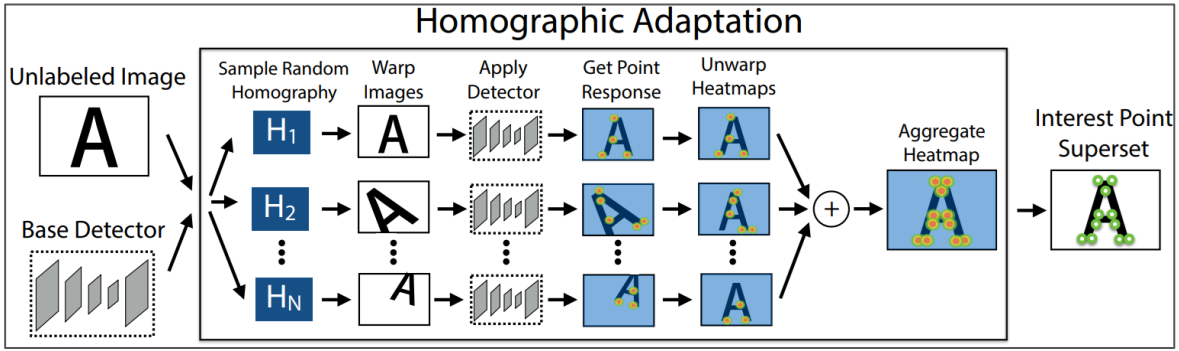
How does pytorch's backpropagation system (autograd) work?

- Recall that there are two steps:
  - **Forward Propagation**: NN makes its best guess about the correct output, by running the input data through each of its functions
  - **Backwards propagation**: NN adjusts its parameters proportionate to the error in its guess, by collecting derivatives and optimizing parameters through SGD
- Backprop is started when you call loss.backward()
  - Autograd calculates and stores gradients for each model in their .grad attribute, if their requires_grad attribute is true.
- optimizer.step() initiates gradient descent to actually update the weights
- All this is managed through a computational graph

Multi-View Geometry

How does SuperPoint work?

- Unlike tasks like human keypoint estimation which can be tackled with strong annotated supervision, general interest point prediction is semantically ill-defined
- SuperPoint instead takes an approach using synthetic data + pseudo GT
  - a) Train a **base detector on a large synthetic dataset** of simple geometric shapes
  - b) Warp target domain, unlabeled input images multiple times to help the base model see the image from many different viewpoints & scales, and save those **pseudo-GT** results. The warping is called “Homographic Adaptation” & helps bridge the domain gap. In the paper, this is iterative done twice to bootstrap.
  - c) Using pseudo-GT, train 2 head model: **per-pixel interest point binary classification** (CE loss) & **feature description** (hinge loss)
    - Training occurs with a pair of synthetically warped image with pseudo-GT & known correspondence from a randomly sampled homography
    - Hinge loss is applied to all pairs of descriptor cells, where the dot product of matching pairs are incentivized to be larger than a margin, and mismatching pairs smaller than a margin



$$\mathcal{L}_d(\mathcal{D}, \mathcal{D}', S) = \frac{1}{(H_c W_c)^2} \sum_{h=1}^{H_c, W_c} \sum_{h'=1}^{H_c, W_c} l_d(\mathbf{d}_{hw}, \mathbf{d}'_{h'w'}; s_{hwh'w'}),$$

where

$$l_d(\mathbf{d}, \mathbf{d}'; s) = \lambda_d * s * \max(0, m_p - \mathbf{d}^T \mathbf{d}') + (1 - s) * \max(0, \mathbf{d}^T \mathbf{d}' - m_n).$$