Supplementary Material

A.1 Neural Radiance Fields (NeRFs)

Neural radiance fields [9] model a scene as a 6D, vector-valued continuous function that maps from a position $\mathbf{x} = (x, y, z)$ and a normalized viewing direction $\mathbf{d} = (d_x, d_y, d_z)$, to the differential density σ and emitted color (r, g, b). In practice, this is achieved via two neural networks which partially share parameters: 1) the density network $\sigma(\mathbf{x})$ which depends only on the position $\mathbf{x}$; and 2) the color network $\mathbf{c}(\mathbf{x}, \mathbf{d})$ which depends on both the position $\mathbf{x}$ and viewing direction $\mathbf{d}$.

Novel-View Synthesis. For a scene, we are given a dataset of N RGB images $\{\mathbf{I}_i\}_{i=1}^N$ with camera poses $\{\mathbf{T}\}_{i=1}^N$. At each iteration, we sample a batch of rays $\mathcal{R} \sim \{\mathbf{T}\}_{i=1}^N$ and optimize σ and $\mathbf{c}$ by minimizing the photometric loss $\mathcal{L}_{\text{rgb}} = \sum_{\mathbf{r}\in\mathcal{R}} \| \hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})\|_2^2$, where $\mathbf{C}(\mathbf{r})$ is the RGB value of the pixel corresponding to ray $\mathbf{r} \in \mathcal{R}$, and $\hat{\mathbf{C}}(\mathbf{r})$ is the color estimated by the model using a discrete approximation of Eq.5 [9, 36].

NeRF synthesizes an image by casting a ray $\mathbf{r}$ from the camera origin $\mathbf{o}$ through the center of each pixel. Points along the ray are parameterized as $\mathbf{r}_t = \mathbf{o} + t\mathbf{d}$, where t is the distance of the point to the camera origin $\mathbf{o}$. The color $\mathbf{C}(\mathbf{r})$ of the ray $\mathbf{r}$ between the near and far scene bounds t_n and t_f is given by the volume rendering integral [37]

$$\mathbf{C}(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}_t)\mathbf{c}(\mathbf{r}_t, \mathbf{d})\,\mathrm{d}t, \quad T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}_s)\,\mathrm{d}s\right), \tag{5}$$

where $T(t)$ is the accumulated transmittance along the ray from $\mathbf{r}_{t_n}$ to $\mathbf{r}_t$.

A.2 Fast Feature Distillation

A.2.1 Distillation Procedure

We extend the Nerfacto method from Nerfstudio [8], and implement our feature field using a hierarchical hash grid [6].

Algorithm 1 Parallel Distillation Pseudocode

```
def parallel_distillation(images, num_steps, lambda_feat):
    nerf = NeRF()
    feat_field = FeatureField()
    optimizer = Adam(nerf, feat_field)

    feats = extract_features(images)

    for i in range(num_steps):
        rays_o, rays_d, target_rgb, target_feats =
    sample_rays(rgb_images, feats)

        # Forward pass
        pred_rgb = nerf(rays_o, rays_d)
        pred_feats = feature_field(rays_o, rays_d)

        # Compute loss
        rgb_loss = MSELoss(pred_rgb, target_rgb)
        feat_loss = MSELoss(pred_feats, target_feats

        loss = rgb_loss + lambda_feat * feat_loss
        loss.backward()
        optimizer.step()
```

We offer two distillation procedures both implemented in our codebase. The first approach trains the NeRF and distills the features in parallel (Algorithm 1). This can require tuning the weight λ associated with the feature loss. Our second approach trains the NeRF, then distills features sequentially (Algorithm 2).

Algorithm 2 Sequential Distillation Pseudocode

```
def sequential_distillation(images, rgb_num_steps,
    feat_num_steps):
    nerf = NeRF()
    feat_field = FeatureField()
    rgb_optimizer = Adam(nerf)
    feat_optimizer = Adam(feat_field)

    feats = extract_features(images)

    for i in range(rgb_num_steps):
        target_rgb = sample_rays(rgb_images)

        # Forward pass
        pred_rgb = nerf(rays_o, rays_d)
        rgb_loss = MSELoss(pred_rgb, target_rgb)

        rgb_loss.backward()
        rgb_optimizer.step()

    for i in range(rgb_num_steps):
        # Distill features at their original resolution
        rays_o, rays_d, target_feats = sample_rays(feats)

        # Forward pass
        pred_feats = nerf(rays_o, rays_d)
        feat_loss = MSELoss(pred_feats, target_feats)

        feat_loss.backward()
        feat_optimizer.step()
```

A.2.2 Extracting Dense Visual Features from CLIP

We present an improved dense feature extraction scheme for vision-language models that preserves their semantic grounding. For clarity, we include Python code in Alg.3 for the original feature output in a CLIP ViT (Line 2: `forward`), and our improved dense feature output (Line 9: `forward_v`). We remove the identity and the residual output from the small MLP in Lines 5-6, which improves the signal-to-noise for language-guided object retrieval tasks. We apply the same technique to the final attention pooling layer in the ResNet version of CLIP, and observe that it provides better contrast than the ViT backbone but is less localized.

Another detail is that the image model in CLIP first center-crops the input image, and then divides it into 14×14 patches which are very low in resolution. We alleviate this issue by interpolating the positional encoding of each patch [3], so that our CLIP model can consume images with arbitrary aspect ratio and divide them into arbitrary numbers of patches. These changes enable us to extract higher-resolution visual features from CLIP and use all the information in an image by removing the need for cropping.

Algorithm 3 Extracting Dense Features from CLIP

```
class Attn(nn.ResidualBlock):
   def forward(self, x):                          # regular output
        q, k, v = W_qkv @ self.ln_1(x)
        v = F.softmax(dot(q[:1], k), dim= -1) * v
        x = x + W_out @ v
        x = x + self.mlp(self.ln_2(x))
        return x                        # take x[:1] for CLS token

   def forward_v(self, x):                           # dense output
        v = W_v @ self.ln_1(x)
        z = W_out @ v
        return z                     # take z[1:] remove CLS token
```

A.3 Experimental Setup

We provide details about our experimental setup used across our experiments for learning to grasp from demonstrations and language-guided object manipulation.

Physical Setup. We collect RGB images with a RealSense D415 camera (the depth sensor is not used) mounted on a selfie stick. The selfie stick is used to increase the coverage of the workspace, as a wrist-mounted camera can only capture a small area of the workspace due to kinematic limitations. We program a Franka Panda arm to pick up the selfie stick from a magnetic mount, scan $50 \times 1280 \times 720$ RGB images of the scene following a fixed trajectory of three helical passes at different heights, and place the selfie stick back on the mount.

To calibrate the camera poses, we run COLMAP [38, 39] on a scan of a dedicated calibration scene with objects at known poses placed by the robot. We use these objects to compute the transformation from the COLMAP coordinate system to the world coordinate system. These camera poses are reused on subsequent scans. Given that the true camera poses vary due to small differences in how the robot grasps the selfie-stick, we optimize them as part of NeRF modeling to minimize any errors and improve reconstruction quality [8, 40, 41].

Labeling Demonstrations. We label demonstrations in virtual reality (VR) using a web-based 3D viewer based on Three.js we developed that supports the real-time rendering of NeRFs, point clouds, and meshes. Given a NeRF of the demonstration scene, we sample a point cloud and export it into the viewer. We open the viewer in a Meta Quest 2 headset to visualize the scene, and move a gripper to the desired 6-DOF pose using the hand controllers (see Fig.2c).

NeRF and Feature Field Modeling. We downscale the images to 640×480 to speed up modeling of the RGB NeRF, and use the original 1280×720 images as input to the vision model for dense feature extraction. We optimize the NeRF and feature field sequentially for 2000 steps each, which takes at most 90s (average is 80s) on a NVIDIA RTX 3090, including the time to load the vision model into memory and extract features.

Feature Type	Resolution
DINO ViT	98×55
CLIP ViT	42×24
CLIP ResNet	24×14

Table 3: **Feature Map Resolutions.** Resolutions of the features output by the vision models given a 1280×720 RGB image.

In our experiments, we distill the features at their original feature map resolution which is significantly smaller than the RGB images (see Table 3). We achieve this by transforming the camera intrinsics to match the feature map resolutions, and sampling rays based on this updated camera model. The specific models we used were `dino_vits8` for DINO ViT, `ViT-L14@336px` for CLIP ViT, and `RN50x64` for CLIP ResNet.

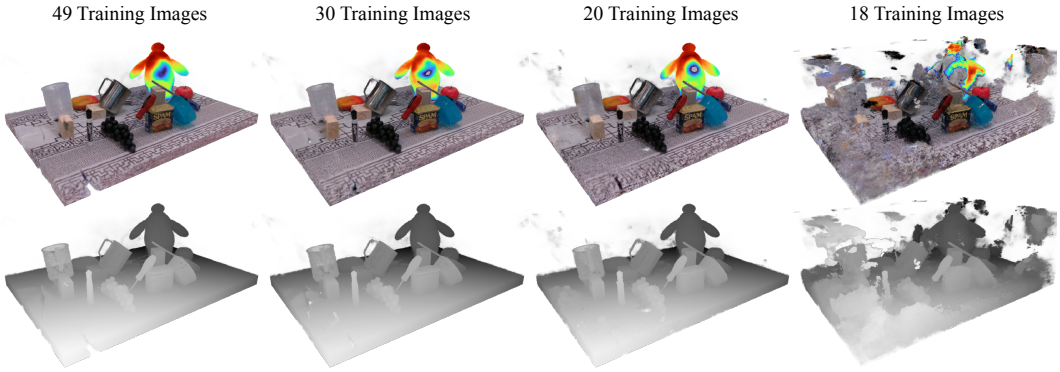


Figure A10: Qualitative comparison of feature fields trained on different numbers of views. (Top Row) The segmentation heatmap for "Baymax" from a CLIP feature field overlaid on the RGB image from NeRF. (Bottom Row) The depth map rendered from NeRF.

A.4 Ablation on Number of Training Views

Although our robot scans 50 images per scene in our experiments, we demonstrate that it is possible to use a significantly smaller number of views for NeRF and feature field modeling without a significant loss in quality. To investigate this, we ablate the number of training images by evenly subsampling from the 50 scanned images and modeling a NeRF and feature field.

Fig. A10 qualitatively compares the RGB, depth, and segmentation heatmaps. We observe an increase in floaters as we reduce the number of training images, with approximately 20 images being the lower bound before a drastic decline in quality.

A.5 Learning to Grasp from Demonstrations

Sampling Query Points. We use $N_q = 100$ query points across all the tasks. As other works have observed [17], the downstream performance can vary significantly across different samples of the query points. To address this issue, we sample five sets of query points over different seeds for each task, and run the grasp optimization procedure across a set of test scenes used for method development. We select the query points that achieved the highest success rate. The covariance of the Gaussian is manually tuned to fit the task.

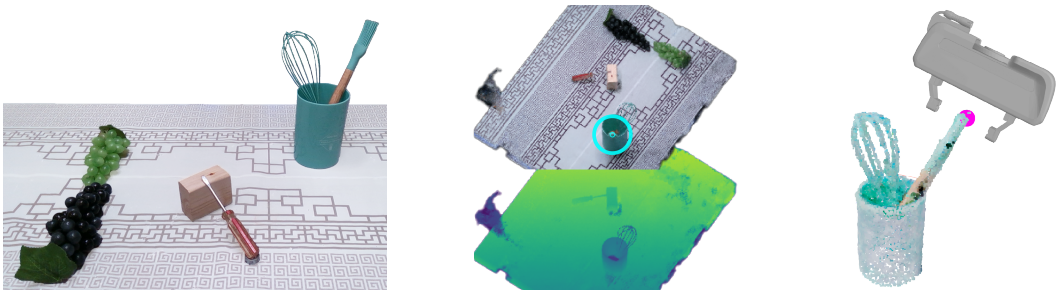
Grasp Pose Optimization. We first discuss how we initialize the grasp poses. We consider a tabletop workspace of size $0.7 \times 0.8 \times 0.35$ meters, and sample a dense voxel grid over the workspace with voxels of size $\delta = 0.0075m$ (we use 0.005m for the cup on racks experiment), where each voxel $\mathbf{v} = (x, y, z)$ represents the translation for a grasp pose.

Next, we compute the alpha value $\alpha(\mathbf{v})$ for each voxel using the NeRF density network σ, and filter out voxels with $\alpha(\mathbf{v}) < 0.1$. This removes approximately 98% of voxels by ignoring free space.

The cosine similarity of the voxel features $\mathbf{f}_\alpha(\mathbf{v})$ is thresholded with the task embedding to further filter out voxels. This threshold is adjusted depending on the task and type of feature distilled, and typically cuts down 80% of the remaining voxels. Finally, we uniformly sample $N_r = 8$ rotations for each voxel to get the initial grasp proposals $\mathcal{T}$.

We minimize Eq. 3 to find the grasp pose that best matches the demonstrations using the Adam optimizer [19] for 50 steps with a learning rate of 5e-3. This entire procedure takes 15s on average.

Grasp Execution. We reject grasp poses which cause collisions by checking the overlap between a voxelized mesh of the Panda gripper and NeRF geometry. We input the ranked list of grasp poses into an inverse kinematics solver and BiRRT motion planner in PyBullet [20, 21], and execute the highest-ranked grasp with a feasible motion plan.



(a) Test Scene with a Screwdriver (b) Affordance Prediction (c) Predicted Grasp

Figure A11: **MIRA Failure Case.** (a) Test scene with a screwdriver and other distractors for the screwdriver task (Fig.4b). (b) The orthographic render of the view selected by MIRA, we show the RGB (top) and depth (bottom) renders. The pixel circled in cyan indicates the action with the highest pixel-wise affordance across all views. (c) The predicted 6-DOF grasp incorrectly targets the silicone brush, as it shares resemblance to a screwdriver from a top-down perspective.

A.5.1 Baselines

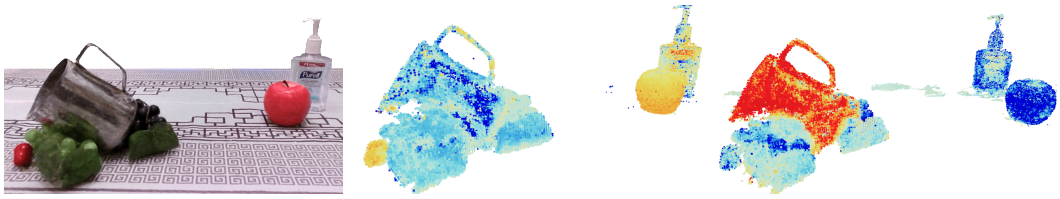
We provide implementation details of the four baselines used in our few-shot imitation learning experiments. The first three baselines use NeRF-based outputs as features for the query point-based pose optimization:

1. Density: we use the alpha $\alpha \in (0, 1)$ values for NeRF density to ensure the values are scaled consistently throughout different scenes, as the density output by the density field σ is unbounded.
2. Intermediate Features: we use the features output by the intermediate density embedding MLP in Nerfacto [8], which have a dimensionality of 15.
3. RGB: we use $[r, g, b, \alpha]$ as the feature for this baseline. α is used to ensure that this baseline pays attention to both the color and geometry, as we found that using RGB only with the alpha-weighted feature field (Eq. 2) collapsed RGB values to $(0, 0, 0)$ for free space, which corresponds to the color black.

MIRA Baseline. The fourth baseline we consider is Mental Imagery for Robotic Affordances (MIRA) [25], a NeRF-based framework for 6-DOF pick-and-place from demonstrations that renders orthographic views for pixel-wise affordance prediction. MIRA formulates each pixel in a rendered orthographic view as a 6-DOF action $\mathbf{T} = (\mathbf{R}, \mathbf{t})$, with the orientation of the view defining the rotation $\mathbf{R}$ and the estimated depth from NeRF defining the translation $\mathbf{t}$. The FCN is trained to predict the pixels in the rendered views corresponding to the demonstrated 6-DOF actions, and reject pixels sampled from a set of negative views. During inference, MIRA renders several orthographic views of the scene and selects the pixel that has the maximum affordance across all views.

In our experiments, we train a separate FCN for 20000 steps using the two demonstrations for each task in Fig.4, and sample negative pixels from datasets containing distractor objects. We use data augmentation following Yen-Chen et al. [25]'s provided implementation and apply random SE(2) transforms to the training views. Given a test scene, we scan 50 RGB images as described in Appendix A.3, render 360 orthographic viewpoints randomly sampled over an upper hemisphere looking towards the center of the workspace, and infer a 6-DOF action. MIRA was designed for suction cup grippers and does not predict end-effector rotations. We attempted to learn this rotation, but found that the policy failed to generalize. To address this issue and give MIRA the best chance of success, we manually select the best end-effector rotation to achieve the task. We additionally find that MIRA often selects floater artifacts from NeRF, and manually filter these predictions out along with other unreasonable grasps (e.g., grasping the table itself).

Given that MIRA is trained from scratch given just two demonstrations, we find that it struggles to generalize and is easily confused by floaters and distractors despite data augmentations and negative



(a) Novel Scene (b) DINO ViT Heatmap (c) CLIP ViT Heatmap

Figure A12: Comparing DINO and CLIP feature fields. We depict the cosine similarity for the task of grasping a mug by the handle. Two demos are provided on a red and a white mug (cf. Fig.3b). (b) DINO overfits to the red color of the apple, while (c) CLIP captures higher-level semantics, and identifies the metal mug.

samples. MIRA additionally reasons over 2.5D by using the rendered RGB and depth from NeRF as inputs to the FCN, while our query point-based formulation reasons optimizes explicitly over 3D. Because of this, we observe that MIRA can fail when there are occlusions or distractor objects that look like the demonstration objects from certain viewpoints. For example, one of the demonstrations for the screwdriver task was a top-down grasp on a screwdriver standing vertical in a rack (Fig. 4b). Fig. A11 depicts an example scene for the screwdriver grasping task where MIRA incorrectly selects a silicone brush as it looks similar to a screwdriver from a top-down 2.5D view. DINO and CLIP ResNet feature fields successfully grasp the screwdriver in this scene, highlighting the benefits of using pretrained features and reasoning explicitly over 3D geometry.

A.5.2 DINO Failure Cases

Our experiments show that DINO struggles with distractor objects which have high feature similarity to the demonstrations, despite not representing the objects and their parts we care about (Fig.A12b). We observe that DINO has the tendency to overfit to color. On the other hand, CLIP struggles far less with distractors due to its stronger semantic understanding (Fig.A12c).

A.6 Language-Guided Manipulation

For the language-guided experiments, we distilled CLIP ViT features from `ViT-L14@336px`. We use the $N_q = 100$ query points sampled for the demonstrations in the learning to grasp from demonstrations section, and similarly use $N_r = 8$ rotations per voxel. We minimize Eq. 4 for 200 steps with Adam using a learning rate of 2e-3.

Failure Cases for Retrieving Demonstrations via Text We find that retrieving demonstrations via text can require prompt engineering, with queries like "mug rack" and "wooden spoon" undesirably matching to the mug handle and cup on racks task (the demo rack is made out of wood), respectively. Since the CLIP feature space is very high dimensional, just one language query could underspecify the desired problem. We believe that task matching could be robustified by incorporating a language description for each demonstration.