Supplemental Materials for TAPIR: Tracking Any Point with per-frame Initialization and temporal Refinement

Carl Doersch[*] **Yi Yang**[*] **Mel Vecerik**[*†] **Dilara Gokay**[*] **Ankush Gupta**[*]
Yusuf Aytar[*] **Joao Carreira**[*] **Andrew Zisserman**[*‡]

[*]Google DeepMind [†] University College London
[‡]VGG, Department of Engineering Science, University of Oxford

Time (s)	# points=10			# points=20			# points=50		
	TAP-Net	PIPs	TAPIR	TAP-Net	PIPs	TAPIR	TAP-Net	PIPs	TAPIR
# frames = 8	0.02	1.3	0.08	0.02	2.4	0.08	0.02	6.2	0.09
# frames = 25	0.05	3.4	0.12	0.05	7.2	0.13	0.05	17.9	0.15
# frames = 50	0.09	6.9	0.20	0.09	14.0	0.21	0.09	34.5	0.25

Table 1. **Computational time for model inference.** We conducted a comparison of computational time (in seconds) for model inference on the DAVIS video *horsejump-high* with 256×256 resolution.

Method	Kinetics			DAVIS		
	AJ	$< \delta^x_{avg}$	OA	AJ	$< \delta^x_{avg}$	OA
TAPIR 256×256	57.2	70.1	87.8	61.3	73.6	88.8
TAPIR Hi-Res	60.0	72.1	86.7	65.7	77.6	86.7

Table 2. **TAPIR at high resolution on TAP-Vid**. Each video is resized so it is at most 1080 pixels tall and 1920 pixels wide for DAVIS, and 720 pixels tall and 1280 pixels wide for Kinetics.

1. Runtime Analysis

A fast runtime is of critical importance to widespread adoption of TAP. Dense reconstruction, for example, may require tracking densely.

We ran TAP-Net, PIPs, and TAPIR on the DAVIS video *horsejump-high*, which has a resolution of 256×256. All models are evaluated on a single V100 GPU using 5 runs on average. Query points are randomly sampled on the first frame with 10, 20, and 50 points, and the video input was truncated to 8, 25, and 50 frames.

Table 1 shows our results. Both TAP-Net and TAPIR are capable of conducting fast inference due to their parallelization techniques; for the number of points we evaluated, the runtime was dominated by feature computation and GPU overhead, and so the runtime is independent of the number of points, and appears to be scaled better than linear across different number of frames (though parts of both algorithms do still scale linearly). In contrast, PIPs exhibits a linear increase in computational time with respect to the number of points and frames processed. Overall, TAP-Net and TAPIR are more efficient in terms of computational time than PIPs, particularly for longer videos and a larger number of points.

2. Extension to High-Resolution Videos

Although TAPIR is trained only on 256×256 videos, it depends mostly on comparisons between features, meaning that it might extend trivially to higher-resolution videos by simply using a larger convolutional grid, similar to ConvNets. However, we find that applying the per-frame initialization to the entire image is likely to lead to false positives, as the model is tuned to be able to discriminate between a 32×32 grid of features, and may lack the specificity to deal with a far larger grid. Therefore, we instead create an image pyramid, by resizing the image to K different resolutions. The lowest resolution is 256×256, while the highest resolution is the original resolution, with logarithmically-spaced resolutions in between that resize at most by a factor of 2 at each level. We run TAPIR to get an initial estimate at 256×256, and then repeat iterative refinement at every level, with the same number of iterations at every level. When running at a given level, we use the position estimate from the prior level, but we directly re-initialize the occlusion and uncertainty estimates from the track initialization, as we find the model otherwise tends to become overconfident. The final output is then the average output across all refinement resolutions.

For running TAPIR on high-resolution videos, we partitioned the model across 8 TPU-v3 devices, where each device received a subset of frames. This allows us to fit DAVIS videos at 1080p and Kinetics at 720p without careful optimization of JAX code. Results are shown in Table 2. Unsurprisingly, having more detail in the image helps the model localize the points better. However, we see that occlusion prediction becomes less accurate, possibly because large context helps with occlusion. Properly combining multiple resolutions is an interesting area for future works.

3. Open-Source Version

In the previous sections, our goal was to align our decisions with those made by prior models TAP-Net and PIPs, allowing for easier comparisons and identification of crucial high-level architectural choices. However, in the interest of open-sourcing the most powerful model possible to the community, we now conduct more detailed hyperparameter tuning. This comprehensive model, inclusive of training and inference code, is openly accessible at `https://github.com/deepmind/tapnet`. The open-source TAPIR model introduces some extra model modifications including (1) the backbone, (2) several constants utilized in the model and training process, and (3) the training setup.

Regarding the backbone, we employ a ResNet-style architecture based on a single-frame design, structurally resembling the TSM-ResNet used in other instances but with an additional ResNet layer. Specifically, we utilize a PreResNet18 backbone with the removal of the max-pooling layer. The four ResNet layers consist of feature dimensions of $[64, 128, 256, 256]$, with 2 ResNet blocks per layer. The convolution strides are set as $[1, 2, 2, 1]$. Apart from the change in the backbone, the TAP features in the pyramid originate from ResLayer2 and ResLayer4, corresponding to resolutions of 64×64 and 32×32, respectively, on a 256×256 image. It is noteworthy that the dimensionality of high-resolution feature map has been altered from 64 to 128. Another significant modification involves transitioning from batch normalization to instance normalization. Our experiments reveal that instance normalization, in conjunction with the absence of a max-pooling layer, yields optimal results across the TAP-Vid benchmark datasets. Collectively, these alterations result in an overall improvement of 1% on DAVIS and Kinetics datasets.

Regarding the constants, we have discovered that employing a softmax temperature of 20.0 performs marginally better than 10.0. Additionally, we have chosen to use only 3 pyramid layers instead of the 5 employed elsewhere. Consequently, a single pyramid layer consists of averaging the ResNet output. For our publicly released version, we have opted for 4 iterative refinements. Furthermore, the expected distance threshold has been set to 6.

In terms of optimization, we train the model for 50,000 steps, which we have determined to be sufficient for convergence to a robust model. Each TPU device is assigned a batch size of 8. We use 1000 warm-up steps and a base learning rate of 0.001. To manage the learning rate, we employ a cosine learning rate scheduler with a weight decay of 0.1. The AdamW optimizer is utilized throughout the training process.

We train TAPIR using both our modified panning dataset and the publicly available Kubric MOVi-E dataset. Our findings indicate that the model trained on the public Kubric dataset performs well when the camera remains static. However, it may produce suboptimal results on videos with moving cameras, particularly for points that leave the image frame. Consequently, we have made both versions of the model available, allowing users to select the version that best suits their specific application requirements.

Average Jaccard (AJ)	Kinetics	DAVIS	RGB-Stacking	Kubric
TAPIR tuned (MOVi-E)	60.2	62.9	73.3	88.3
TAPIR tuned (Panning Kubric)	58.2	62.4	69.1	85.8

Table 3. **Open Sourced TAPIR results on TAP-Vid Benchmark.** Comparing to our major reported model, the open sourced version improves substantially, particularly on RGB-Stacking dataset.

Average Jaccard (AJ)	Kinetics	DAVIS	RGB-Stacking	Kubric
TAPIR Model	57.2	**61.3**	**62.7**	**84.7**
+ With RNN	57.1	61.1	59.6	84.6
+ With RNN with gating	**57.6**	61.1	59.5	84.5

Table 4. **Comparison of the TAPIR model with and without an RNN.** We see relatively little benefit for this kind of temporal integration, though we do not see a detriment either. This suggests an area for future research.

4. More Ablations

Although our model does not have many extra components, both TAP-Net and PIPs have a fair number of their own design decisions. In this section, we examine the importance of some of these design decisions quantitatively. We consider four open questions. First, are there simple and fast alternatives to 'chaining' that allow the model to process the full video between the query point and every output frame? Second, is within-channel computation (i.e. depthwise conv or the within-channel layers of the Mixer) important for overall performance, or do standard dense convolutions work better? Third, do we need as many pyramid levels as proposed in the PIPs model? Fourth, do we still need the time shifting that was proposed in TAP-Net? In this section, we address each question in turn.

4.1. Recurrent Neural Networks

In theory, the 'chaining' initialization of PIPs has an advantage which TAPIR does not reproduce: when the model has a query on frame t_q and makes a prediction at a later frame t_o, the prediction at frame t_o depends on *all* frames between t_q and t_o. This may be advantageous if there are no occlusions between t_q and t_o, and the point's appearance changes over time so that it becomes difficult to match based on appearance alone. PIPs can, in these cases, use temporal continuity to avoid losing track of the target point, in a way that TAPIR cannot do easily.

Of course, it is unclear how important this is in practice: relying on temporal continuity can backfire under occlusions. To begin to investigate this question, we made a simple addition to the model: an RNN which operates across

time, which has similar properties to the 'chaining,' which we apply during the TAP-Net initialization after computing the cost volume.

A convolutional recurrent neural network (Conv RNN) is a classic architecture for spatiotemporal reasoning, and could integrate information across the entire video. However, if given direct access to the features, it could overfit to the object categories; furthermore, if given too much latent state, it could easily memorize the stereotypical motion patterns of the Kubric dataset. Global scene motion, on the other hand, can give cues for the motion of points that are weakly textured or occluded. Our RNN architecture aims to capture both temporal continuity and global motion while avoiding overfitting.

The first step is to compute global motion statistics. Starting with the original feature map $F \in \mathbb{R}^{T\times H//8\times W//8\times D}$, we compute local correlations $D \in \mathbb{R}^{T\times H\times W\times 9}$ for each feature in frame t with nearby features in frame $t+1$:

$$S_{t,x,y} = \{F_{t,x,y} \cdot F_{t-1,i,j} | x-1 \le i \le x+1; \\ y-1 \le j \le y+1\} \quad (1)$$

Note that, for the first frame, we wrap $t-1$ to the final frame. We pass S into two Conv+ReLU layers (256 channels) before global average pooling and projecting the result to $P_t \in \mathbb{R}^{32}$ dimensions. Note that this computation does not depend on the query point, and thus can be computed just once. The resulting 32-dimensional vector can be seen as a summary of the motion statistics, which will be used as a gating mechanism for propagating matches.

The RNN is applied separately for every query point. The state $R_t \in \mathbb{R}^{H\times W\times 1}$ of the RNN itself is a simple spatial softmax, except for the first frame which is initialized to 0. The cost volume C_t at time t is first passed through a conv layer (which increases the dimensionality to 32 channels), and then concatenated with the RNN state R_{t-1} from time $t-1$. This is passed through a Conv layer with 32 channels, which is then multiplied by the (spatially broadcasted) motion statistics P^t. Then a final Conv (1 channel) followed by a Softmax computes R_t. The final output of the RNN is the simple concatenation of the state R across all times. The initial track estimate for each frame is a soft argmax of this heatmap, as in TAPIR. We do not modify the occlusion or uncertainty estimate part of TAPIR.

Table 4 shows our results. We also show a simplified version, where we drop the 'gating', and simply compute R_t from R_{t-1} concatenated with C_t, two Conv layers, and a softmax. Unfortunately, the performance improvement is negligible: with gating we see just a 0.4% improvement on Kinetics and a 0.2% loss on DAVIS. We were not able to find any RNN which improves over this, and informal experiments adding more powerful features to the RNN harmed performance. While the recurrent processing makes intuitive sense, actually using it in a model appears to be non-trivial, and is an interesting area for future research.

Average Jaccard (AJ)	Kinetics	DAVIS	RGB-Stacking	Kubric
MLP Mixer	54.9	53.8	61.9	79.7
Conv1D	56.9	60.9	61.3	84.6
Depthwise Conv	**57.2**	**61.3**	**62.7**	**84.7**

Table 5. **Comparison between the layer kernel type in iterative updates.** We find that depthwise conv works slightly better than a dense 1D convolution, even though the latter has more parameters.

Average Jaccard (AJ)	Kinetics	DAVIS	RGB-Stacking	Kubric
pyramid_level=5 (TAPIR)	57.2	61.3	62.7	84.7
pyramid_level=4	57.5	61.3	61.6	**84.9**
pyramid_level=3	**57.9**	**61.5**	**63.0**	84.8
pyramid_level=2	57.2	61.0	61.2	84.4

Table 6. **Comparison on number of feature pyramid levels.** We find that the number of pyramid levels makes relatively little difference in performance; in fact, 3 pyramid levels seems to be all that is required, and even 2 levels gives competitive performance.

4.2. Depthwise versus Dense Convolution

PIPs uses an MLP-Mixer for its iterative updates to the tracks, and the within-channel layers of the mixer inspired the depthwise conv layers of TAPIR. How critical is this design choice? It is desirable as it saves some compute, but at the same time, it reduces the expressive power relative to a dense 1D convolution. To answer this question, we replaced all depthwise conv layers in TAPIR with dense 1D convolutions of the same shape, a decision which almost quadruples the number of parameters in our refinement network. The results in Table 5, however, show that this is actually slightly harmful to performance, although the differences are negligible. This suggests that, despite adding more expressivity to the network, in practice it might result in overfitting to the domain, possibly by allowing the model to memorize motion patterns that it sees in Kubric. Using models with more parameters is an interesting direction for future work.

4.3. Number of Pyramid Layers

In PIPs, a key motivation for using a large number of pyramid layers is to provide more spatial context. This is important if the initialization is poor: in such cases, the model should consider the query point similarity to other points over a wide spatial area. TAPIR reproduces this decision, extracting a full feature pyramid for the video by max pooling, and then comparing the query feature to local neighborhoods in each layer of the pyramid. However, TAPIR's initialization involves comparing the query to all other features in the entire frame. Given this relatively stronger initialization, is the full pyramid necessary?

To explore this, we applied the same TAPIR architecture,

Average Jaccard (AJ)	Kinetics	DAVIS	RGB-Stacking	Kubric
TAPIR Model	**57.2**	**61.3**	**62.7**	**84.7**
- No TSM	57.1	61.0	59.4	84.3

Table 7. Comparing the TAPIR model with a version without the TSM (temporal shift module).

but successively removed the highest levels of the pyramid, and Table 6 gives the results. We see that fewer pyramid levels than were used in the full TAPIR model are sufficient. In fact, 2 pyramid levels saves computation while providing competitive performance.

4.4. Time-Shifting

One slight complexity that TAPIR inherits from TAP-Net is its use of a TSM-ResNet [3] rather than a simple ResNet. TAP-Net's TSM-ResNet is actually modified from the original version, in that only the lowest layers of the network use the temporal shifting. Arguably, the reason for this choice is to perform some minor temporal aggregation, but it comes with the disadvantage that it makes the model more difficult to apply in online settings, as the features for a given frame cannot be computed without seeing future frames. However, our model uses a refinement step that is aware of time. This raises the question: how important is the TSM module?

To answer this question, we replaced the TSM-ResNet with a regular ResNet–i.e., we simply removed the time shifting from its earliest layers, and kept all other architectural details the same. In Table 7, we see that this actually makes little difference for TAPIR on real data, losing a negligible 0.1% performance on Kinetics and a similarly negligible 0.3% on DAVIS. Only for RGB-Stacking does it seem to make a difference. One possible explanation is that the model struggles to segment the low-texture RGB-Stacking objects, so the model uses motion cues to do this. Regardless, it seems that for real data, the time-shifting layers can be safely removed.

5. Implementation Details

In this section, we provide implementation details to enable reproducibility, first for TAPIR, then for our new synthetic dataset.

5.1. TAPIR

The TSM-ResNet model used in the bulk of experiments (but not the open-source model) follows the one in TAP-Net: i.e., it has time-shifting in the first two blocks, and replaces the striding in later blocks with dilated convolutions to achieve a stride-8 convolutional feature grid. We use the output features of unit 2 (stride 8) and unit 0 (stride 4), and normalize both across channels before further processing.

We use the unit 2 features to compute the cost volume, using bilinear interpolation to obtain a query feature, before computing dot products with all other features in the video. The TAP-Net-style post-processing first apply an embedding convolution (16 channels followed by ReLU) to obtain an embedding e, followed by a convolution to a single channel, to which we apply a spatial softmax followed by a soft argmax operation to obtain a position estimate p_t^0 for time t. To predict occlusion, we apply a strided convolution to the embedding e (32 channels) followed by a ReLU and then a spatial global average pool. Then we apply an MLP (256 units, ReLU, and finally a projection to 2 units) to produce the logits o_t^0 and u_t^0 for occlusion and uncertainty estimate, respectively.

The above serves as the initialization for the refinement, along with the raw features. Each iteration i produces an update $(\Delta p_t^i, \Delta o_t^i, \Delta u_t^i, \Delta F_{q,t,i})$. To construct the inputs for the depthwise convolutional network, o_t^i and u_t^i are passed in the form of raw logits; $F_{q,t,i}$ is passed unmodified. Note that, for versions of the network with a higher resolution, $F_{q,t,i}$ comprises both the high-resolution (64-channel) feature as well as the low-resolution (256-dimensional) feature, which are concatenated along the channel axis. The dot products that make up the score maps are passed in spatially unraveled. The final input for p_t^i needs a slight modification. For p_t^i, PIPs subtracts the initial estimate for each chunk before processing with an MLP mixer. We cannot follow this exactly because our model has no chunks; therefore, we instead subtract the temporal mean x and y value across the entire segment, ignoring occlusion. This choice means that, like it is for PIPs, the input to the depthwise-convolutional network input is roughly invariant to spatial shifts of the entire video, at least up to truncation of the score maps.

Following PIPs, our depthwise convolutional network first increases the number of channels to 512 per frame with a linear projection. Then we have 12 blocks, each consisting of one 1×1 convolutional residual unit, and one depthwise residual unit. The 12 blocks are followed by a projection back down to the output dimension, which is split into $(\Delta p_t^i, \Delta o_t^i, \Delta u_t^i, \Delta F_{q,t,i})$. Both types of residual unit consist of a single convolution which increases the per-frame dimension to 2048, followed by a GeLU, followed by another convolution that reduces the dimension back to 512 before adding the input. Note that increasing the dimensionality is non-trivial within a depthwise convolutional network: in principle, each output channel in a depthwise convolution should correspond to exactly one input channel. Therefore, we accomplish this by running *four* depthwise convolutions in parallel, each on the same input. We apply a GeLU, and then run a second depthwise convolution on each of these outputs, and sum all four output layers. This has the effect of increasing the per-frame dimension

to 2048 before reducing it back to 512, while ensuring that each output channel receives only input from the same input channel.

Our losses are described in the main text; we use the same relative scaling of the Huber and BCE as in TAP-Net, and the uncertainty loss is weighted the same as the occlusion. Like TAP-Net, we train on 256 query points per video, sampled using the defaults from Kubric. However, we find that running refinement on all query points with a batch size of 4 tends to run out of memory. We found it was effective to run iterative refinement on only 32 query points per batch, randomly sampled at every iteration.

5.2. Training Dataset

Our primary motivation for generating a new dataset is to address a particular degenerate behavior that we noticed when running the model densely: videos with panning often caused TAPIR to fail on background points that went offscreen early in the video. Therefore, we made a minor modification to the public MOVi-E script by changing the camera rotation. Specifically, we set the 'look at' point to follow a linear trajectory near the bottom of the workspace, traveling through the center of the workspace to ensure that the camera pans but still remains mostly looking at the objects in the scene.

To accomplish this, we first sample a 'start point' a within a medium-sized sphere (4 unit radius; the camera center itself starts between 6 and 12 units from the workspace center). We also constrain that the start point a is above the ground plane, but close to the ground (max 1 unit height). Then we sample a 'travel through' point b in a small sphere in the center of the workspace (1 unit radius), ensuring that b is also above the ground plane; the final 'look at' path will travel through this point. Finally, we sample an end point by extending the line from the start to the 'travel through' point by as much as 50% of the distance between them: i.e., the 'end point' $c = b + \gamma * (b - a)$, where γ is randomly sampled between 0 and 0.5. The final 'look at' point travels on a linear trajectory, either from a to c or from c to a with a 50% probability. We sample 100K videos to use as a training set, and keep all other parameters consistent with Kubric.

5.3. Diffusion Models for Animating Still Images

Recent pipelines using diffusion for video generation have typically used image pretraining, as well as multiple levels model 'chaining', i.e., one model may be trained to produce low-resolution, low-frame rate videos, another may be used to fill in gaps between frames, a third may be used to upsample, and so on. Make-a-video [5], for example, had four different models, and was pretrained on 2.3B images. It also uses the same 'noise' vector on every image to encourage temporal coherence, although this in practice can limit the variability of background textures across the video. For simplicity and for computational reasons, in this project we chain together just two models, and use no image pretraining. We expect that further model chaining, pretraining, and larger video datasets would improve performance, but that is beyond the scope of the current work.

As described in the main paper, our diffusion model consists of two components: a *trajectory* model, and a *video* model. To train both, we first run TAPIR on a large database of videos, center-cropped to 256×256 and clipped to 24 frames. We query TAPIR with a dense, 64×64 grid of query points on the first frame. This provides training data for both models.

These two models are trained independently: the trajectory model is conditioned on the first frame and is trained to reproduce the associated trajectories. The video model is conditioned on both the first frame and the associated trajectories, and is trained to reproduce the later frames of the video. We describe the network architecture for each in turn.

5.3.1 Trajectory Model

The trajectory model first processes the input image with a modified NFNet [1] F0, modified to output a high-resolution grid: all layers after the first have striding removed and, to prevent a resulting explosion in memory, the number of channels in the backbone is capped at 512. This results in a feature at stride 8, with relatively small receptive fields; to enable more global reasoning, we apply a multi-headed self-attention layer with 4 heads and 128 attention channels, resulting in a new feature map with 512 channels at stride 8. The tracks, meanwhile, are at stride 4: therefore, we upsample the above feature map with a transposed convolution with 256 output channels. To recover high-resolution information, we also apply a convolution to the NFNet's block 0 output (a stride-4 tensor with 256 channels) and add this to the output of the transposed convolution. Thus, the final image representation G is a stride 4 tensor with 256 channels.

Our overall diffusion pipeline follows DDPM [2] with a cosine rule. The input array to be denoised has shape $24 \times 64 \times 64 \times 3$, where 24 is the number of frames, and 3 corresponds to x, y, and occlusion. This encodes the (x, y) positions relative to the first frame, scaled to the range $[-1, 1]$ (i.e., 1 corresponds to a positive motion of 256 pixels, the max possible). Occlusion is unfortunately a binary value, so we apply a smoothing operation to make it continuous. Let $\tilde{o}_t$ be an occlusion indicator: i.e., 1 if the point is occluded at time t and -1 otherwise. For each point t, let $\hat{t}$ be the nearest time such that $\tilde{o}_{\hat{t}} \neq \tilde{o}_t$. We compute $\bar{o}_t = \tilde{o}_t * (1 - (2/3)^{|t-\hat{t}|})$. Thus, this value decays exponentially toward the extreme values 1 and -1 as distance from

a 'transition' increases, but it preserves the sign of $\tilde{o}_t$, making decoding straightforward. We use $\bar{o}_t$ as the occlusion estimate, rescaled to the range $(0, .2)$. This rescaling encourages the model to reconstruct the motion first, and then reconstruct the occlusion value based on the reconstructed motion.

For each training example, we randomly sample a 'time' value τ uniformly between 0 and 1, and we sample a noise vector $\hat{\epsilon}$ which is the same size as the input array. The loss is then:

$$\|x_0 - f_\theta(x_0 * cos(\tau * \pi/2) + \hat{\epsilon} * sin(\tau * \pi/2)|G)$$

Here, f_θ is the neural network parameterized by θ. f_θ is a U-Net with self attention, following [6], although we use only 3 residual blocks per resolution. To apply the conditioning, we follow the conditional group norm formulation [2, 4]. That is, after group norm does mean subtraction and variance normalization within each group to produce a normalized output Z, the layer would typically apply a scale and shift operation. These are replaced with linear projections of the conditioning G. Prior work, however, assumes that the conditioning is a single vector, i.e., $G \in \mathbb{R}^C$, whereas we have a spatial tensor $G \in \mathbb{R}^{H//4 \times W//4 \times C}$. Therefore, we first resize G so that its spatial dimensions are the same size as Z, and then we apply two 1×1 convolution layers to create a scale and multiplier that are the same size as Z.

To visualize the output of this model, we use a patch-based warping. For a given frame t, the trajectories at time t tell us where each 4×4 patch in the input image should appear. Therefore, it is straightforward to construct a new image where each local patch is placed at its correct location, using bilinear interpolation to get subpixel accuracy. However, in practice this will look bad if objects get larger, e.g., as they approach the camera, as there will be gaps between patches. To deal with this, we actually warp a larger patch around each point (8 pixels on a side). When multiple patches appear covering the same output pixel, we weight them inversely proportional to their distance from the track center. This results in a smooth blend between patches that reveals motion without producing too many artifacts. However, note that this simple method means that there tend to be artifacts around edges; e.g., points near object edges will tend to capture part of the background as well and drag it along with the object.

5.3.2 Video Model

The second stage of our model produces pixels given trajectories and an initial image. At training time, we use pseudo-ground-truth trajectories from TAPIR extracted from our 24-frame videos, as well as the initial frame at 256×256, and train the model to reproduce the original 24-frame clip at full 256×256 resolution.

Training the model to produce a full clip at the input resolution, however, would be prohibitive in terms of memory. Therefore, at training time, we train our model to reconstruct a single frame at a time. We rely mostly on the trajectory model to provide temporal coherence, at least for the image contents that are visible in the first frame. For the rest of the pixels, we find it is beneficial to also include a small amount of temporal context from the noisy video.

Therefore, at training time, the model must reconstruct some frame t. The input is threefold. We include two forms of 'conditioning' computed by warping the input frame, first in feature space and second in image space. Third, we input the noisy inputs for frames $t-1$ to $t+1$. The target output is the noise residual, i.e., the noise that has been added to the image.

To compute the feature-space-warped image, we use the same image encoder that we used for the tracks model, which produces a feature grid at stride 4. Therefore, for each position in the feature grid, we have the position where that feature appears at time t. We use bilinear interpolation to place the feature at the appropriate location. We keep track of the number of features that have been placed within any particular grid cell (specifically, the sum of bilinear interpolation weights) and normalize by this sum, unless the sum is less than 0.5, in which case we simply divide by 0.5. This serves as a grid of conditioning features, which we use in exactly the same way that we used it in the trajectory prediction network: i.e., it is the conditioning signal for the conditional group norm.

To compute the image-space warped image, we use almost exactly the same warping algorithm that we use for our visualizations, and concatenate the result to the noisy frame. However, we find that the model struggles to deal with the aliasing that occurs when multiple patches overlap; with just a single warp, the model cannot tell what is the contribution of each patch in turn. Therefore, we actually warp the same patch 5 times. The only difference between the warps is the way that the blending weight is calculated. Let $p_{i,j,t}$ be the position of the trajectory beginning at (i, j) in the original image at time t. In the original warping, the weight for the patch carried by trajectory to any particular pixel p'_t (assuming p'_t is close enough to be within the patch) is proportional to $1/d(p_{i,j,t}, p'_t)$, where d is Euclidean distance. The weighting that we use for the five warps is instead $1/d(p_{i,j,t} + \eta, p'_t)$, where $\eta \in \{(0,0), (-2,0), (0,-2), (2,0), (0,2)\}$. Therefore, the first warp is identical to the warp used in the visualization; however, the model can use the differences in intensities between the different warpings to infer the original values for different patches, and can use this information to 'undo' the aliasing.

We also make a few modifications to the U-Net architecture. Notably, we reduce the number of residual blocks per resolution from 3 to 2, and we use self attention only at the lowest-resolution layers (which are at stride 8). We find that the memory usage is otherwise prohibitive. We also find it is beneficial to introduce a gating layer which allows the lower-resolution pathways of the U-Net to preferentially make predictions for parts of the image that contain 'holes' in the warped image; these are the parts of the image where higher-level information is most critical to 'fill in' the missing data. Recall that U-Nets combine the lower-resolution, higher-level feature tensors with 'skip connection' tensors by first upsampling the lower-resolution features via a transposed strided convolution, and then concatenating it with the 'skip connection' tensor. We take the output of this concatenation and apply a projection down a single channel; we interpret this as a logit and apply a sigmoid to obtain a 'gating' value between 0 and 1, with a single channel but the same spatial shape as the input tensors. We then repeat the concatenation, but this time, multiply the skip connection by the 'gating' value before concatenating. Therefore, the model can override the low-level information with higher-level information if it decides that the lower-level input is uninformative.

At test time, we use different noise vectors for every frame. For every iteration of diffusion, we break the noisy video into the above, per-frame inputs and make a prediction for a single frame. Then we concatenate all outputs and repeat the process.

We trained both models for roughly 200K iterations with a cosine learning rate schedule. We use a batch size of 4096 for the trajectory model and a batch size of 256 for the video model, each time training on a $2 \times 2 \times 2$ TPU-v3 pod. In practice we performed relatively little tuning of this model due to its high computational cost; we suspect that further architectural tuning, as well as larger datasets, will lead to improved performance.

References

[1] Andy Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 1059–1071. PMLR, 2021.

[2] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Proceedings of Neural Information Processing Systems (NeurIPS)*, 33:6840–6851, 2020.

[3] Ji Lin, Chuang Gan, Kuan Wang, and Song Han. Tsm: Temporal shift module for efficient and scalable video understanding on edge devices. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[4] Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer. In *Proceedings of AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[5] Uriel Singer, Adam Polyak, Thomas Hayes, Xi Yin, Jie An, Songyang Zhang, Qiyuan Hu, Harry Yang, Oron Ashual, Oran Gafni, et al. Make-a-video: Text-to-video generation without text-video data. *arXiv preprint arXiv:2209.14792*, 2022.

[6] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *Proceedings of International Conference on Learning Representations (ICLR)*, 2021.