A Implementation Details

A.1 Pseudocode of NeRV-Enc and NeRV-Dec

We firstly provide pseudocode for NeRV-Enc and NeRV-Dec in Algorithm 1.

Algorithm 1 Pseudocode in a PyTorch style.

```
############ 1) Video encoding: NeRV-Enc ############
# Input: video x, initial weights θ_0
# Output: video-specific weights θ̂'

# Video tokenization
x = FC_1(x.tokennize())                                          # d \times M

# Concat video patches and initial weights as input
x = x.concat(θ_0)                                                # d \times (M+N)

# Hypernetwork g_ϕ output video-specific weights θ̂'
θ̂' = g_ϕ.forward(x)[-N:].                                        # d \times N
θ̂' = FC_2(θ̂')                                                   # C_out \times N

############ 2) Video decoding: NeRV-Dec ############
θ̂' = θ̂'.expand_as(θ_1)                                           # broadcast into needed shape
θ' = θ_1 * θ̂'                                                    # C_out \times C_in \times K \times K

# Initial NeRV model f_θ with generated weights
f_θ.reset_parameter(θ')

# Input frame index t, and output video frame x̂_t
x̂_t = f_θ.forward(t)

############### 3) Model optimization ###############
# Compute loss and backward gradients
loss = MSELoss(x̂_t, x_t)
loss.backward()

# update all learnable parameters
update([ϕ, θ_0, θ_1])
```

FC: fully connected layer;
MSELoss: mean square error loss.

A.2 Scaling the training of NeRV-Enc

We explore factors such as the number of training videos, training epochs, encoder size, and weight token distributions, as outlined in Tab. 7. Generally, NeRV-Enc's reconstruction performance improves with increased computational resources, although the gains tend to plateau as training duration becomes adequate. Higher dropout ratios are essential for achieving improved generalization in longer training epochs or with larger hyper-networks. These ablations underscore that the reconstruction quality of NeRV-Enc can be significantly enhanced by scaling the training with additional resources, including more training videos, longer training epochs, and larger hyper-networks.

	PSNR ↑				SSIM ↑			
	Train	K400	SthV2	UCF101	Train	K400	SthV2	UCF101
Epoch ablation								
150	25.8	25.8	28.5	25.2	0.732	0.727	0.795	0.723
300	26.8	26.7	29.5	26.2	0.763	0.757	0.819	0.757
600	27.2	27.1	30	26.6	0.774	0.768	0.827	0.766
1200	27.6	27.3	30.2	26.7	0.787	0.776	0.832	0.774
1800	27.8	27.4	30.4	26.9	0.791	0.782	0.837	0.781
Encoder size ablation								
47.6M	25.8	25.8	28.5	25.2	0.732	0.727	0.795	0.723
125M	26.4	26.2	28.9	25.6	0.751	0.743	0.806	0.737
251M	26.5	26.7	29.5	26.2	0.756	0.762	0.822	0.759
404M	26.5	26.5	29.3	25.9	0.753	0.755	0.816	0.751
Video number ablation								
10k	25.8	25.8	28.5	25.2	0.732	0.727	0.795	0.723
20k	26.7	26.6	29.4	26	0.757	0.752	0.814	0.751
40k	27	26.9	29.7	26.4	0.764	0.761	0.821	0.762
80k	27.6	27.7	30.6	27.2	0.791	0.791	0.845	0.794
240k	27.8	27.9	30.9	27.4	0.792	0.799	0.852	0.802

Table 7: NeRV-Enc ablations. For ablation of weight tokens, we compare uniform tokens (TransINR [9]), layer-specific tokens (GINR [27] at the 2nd layer), and our proposed layer-adaptive weight tokens. Please refer to the main paper for their distinction.

A.3 NeRV-Enc for video restoration tasks.

Our NeRV-Enc framework is versatile across various downstream tasks, and shows robust restoration quality for various degradations. Results in **??** and Tab. 8 demonstrate that the reconstruction quality for downsampled, blurred, and masked input videos is comparable to that achieved through conventional video regression. *This underscores the framework's effectiveness in restoring common pixel degradations within the implicit space.*

Input degradation	Input PSNR				Output PSNR			
	Train	k400	sth-v2	ucf101	Train	k400	sth-v2	ucf101
Downsample	20.1	20.3	22.9	19.5	24.5	24.3	26.8	23.9
Gaussian blur	23.1	23.3	26.0	22.3	24.8	24.7	27.3	24.0
Inpainting	19.0	18.6	18.1	18.4	25.5	25.2	27.9	24.7
No	-	-	-	-	25.8	25.8	28.5	25.2

Table 8: Results for downstream tasks with NeRV-Enc.

A.4 Weight quantization for efficient storage.

In this quantization procedure, each element of a vector μ is mapped to the nearest integer using the linear transformation defined by the formula:

$$\mu_i = \text{Round}\left(\frac{\mu_i - \mu_{\min}}{\text{scale}}\right) * \text{scale} + \mu_{\min}, \text{where}$$
$$\text{scale} = \frac{\mu_{\max} - \mu_{\min}}{2^b - 1}, \tag{2}$$

Here, μ_i represents a vector element, *Round* is a rounding function, b is the quantization bit length, $\mu_{\max}$ and $\mu_{\min}$ are the maximum and minimum values of vector μ, and 'scale' is the scaling factor. Additionally, we use Huffman encoding to further reduce the disk storage.

Results for Model Quantization We extend our analysis on model quantization in Tab. 9, assessing performance on three datasets: K400, Something-V2, and UCF-101.

Bits	PSNR ↑			SSIM ↑		
	K400	STH-V2	UCF101	K400	STH-V2	UCF101
32	28.4	31.6	28.1	0.808	0.862	0.817
8	28.4	31.5	28.1	0.808	0.861	0.816
7	28.3	31.5	28	0.807	0.86	0.815
6	28.1	31.2	27.9	0.802	0.855	0.811
5	27.5	30.2	27.3	0.784	0.836	0.794
4	25.6	27.7	25.6	0.712	0.759	0.725

Table 9: Ablation study on **model quantization**.

A.5 Implementation details

Video Encoding. We firstly provide training details of NeRV-Enc below.

– NeRV-Enc:
- Video: 8 frames, frame stride evenly sample from the whole video, 256×256 resolution.
- Batch size: 32
- Patch size: 64
- Position embedding dimension for NeRV: 16
- Activation layer in NeRV: GeLU
- Kernel size for convolution layers in NeRV: 1, 3, 3, 3
- Upscale factor for NeRV blocks: 4, 4, 4, 4
- Token number for NeRV layers: 4, 128, 64, 0
- Token dimensions for NeRV layers: 256, 144, 288, 0

- Model dimension and feed-forward dimension for transformer encoder layers: 720 and 2800 for NeRV-Enc of 47.6M, 1600 and 6400 for NeRV-Enc of larger NeRV-Enc (251M)
- Dropout ratio in transformer encoder layers: 0 for default training, 0.15 for larger NeRV-Enc and long training
- Optimizer: AdamW
- Learning rate: 0.0001

Video Decoding. To assess the decoding speed of NeRV-Dec, H.264, RAM, and AV1, we employ a PyTorch dataloader to facilitate parallel decoding. We initially stack the NeRV model weights before inputting them into NeRV-Dec. Regarding video compression, we utilize the 'torchvision.io.write_video' function to store videos, applying various CRF (Constant Rate Factor) settings. For video loading, we experiment with two backends: 'decord' and 'torchvision.io.read_video', selecting the one that offers superior performance for H.264 and AV1.

B Hurdles When Converting MLP to CNN

We address the difficulties encountered when converting MLP to CNN for NeRV-Enc, as detailed in Tab. 10. The final model weights θ' result from the element-wise multiplication of $\hat{\theta}' \in \mathbb{R}^{d_{\text{out}} \times N}$ and video-agnostic weights $\theta_1 \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K \times K}$, as illustrated in Figs. 2 and 4 of the main paper. This is expressed as

$$\theta' := \theta_1 * \hat{\theta}'.\text{expand\_as}(\theta_1), \tag{3}$$

and is followed by L2 normalization. **a)** Omitting the ***final normalization*** leads to a significant performance drop, from 25.8 to 22.9 for the K400 test PSNR. **b)** ***Convolution initialization of*** θ_1 is crucial, as it increases the test PSNR by approximately 1.8. **c)** The choice of ***expansion dimension*** when expanding $\hat{\theta}'$ to match θ_1's shape is pivotal. Expanding along dimension -2 (the default choice) produces the best results compared to dimensions -1 and -3.

Method	Train	K400	SthV2	UCF101
NeRV-Enc	25.8	25.8	28.5	25.2
w/o Normalize θ'	23.3	22.9	25.7	22.9
w/o ConvInit θ_1	24.1	24	26.7	23.5
Expand $\hat{\theta}'$ at dim = -1	24.6	24.3	26.9	23.8
Expand $\hat{\theta}'$ at dim = -3	24.2	23.9	26.5	23.4

Table 10: Challenges in converting MLPs to ConvNets, showcasing normalized final weights θ', convolution initialization of θ_1, and expansion dimension for $\hat{\theta}'$. PSNR↑ showed, the higher the better.