A Pre-Training Details

We adapt the implementation of the original MAE in two main aspects to make it compatible with salient patch selection and downstream RL tasks: architecture and datasets.

A.1 Architecture and Training Schedule

The standard architectures designed in the original ViT and MAE papers [6,14] have tens of millions of parameters, which are too huge to be applied directly in the RL domain efficiently (i.e., with respect to computational and memory costs). Accordingly, we use a much smaller encoder with about 162K parameters as shown in Table 4. Moreover, to enhance the ability for background reconstruction to select moving objects, we use a larger decoder than encoder, which is contrary to the original MAE design.

Table 4. Comparison of architecture settings between `SPIRL` (for Atari game frames) and original MAE (for real image dataset like ImageNet) [14]. Note that to count the total number of parameters for MAE, we need to sum up the encoder and decoder's number of parameters, as well as the size of trainable [cls] token and [mask] token.

Hyper-parameter		`SPIRL`	Original MAE		
			Base	Large	Huge
Image shape		(96, 96, 3)	(224, 224, 3)		
Patch size		8	16	16	14
# Patches		12×12	14×14	14×14	16×16
Encoder	Embed dim	64	768	1024	1280
	Depth	3	12	24	32
	#Heads	4	12	16	16
	#Parameters	162,432	85,646,592	303,098,880	630,434,560
Decoder	Embed dim	128	512		
	Depth	3	8		
	#Heads	8	16		
	#Parameters	628,160	26,007,808	26,138,880	26,177,612
Total #Parameters		790,784	111,655,680	329,239,296	656,613,964

We keep the same training schedule as the original MAE in general (Table 5), except that we use less training epochs and smaller number of batch size since our training datasets is much simpler and smaller than real image datasets like ImageNet [5].

A.2 Compare Pre-training Requirements with Baselines

RGB frames are collected from a random uniform policy that taking actions from a uniform distribution with the same environment wrapper as RL training

Table 5. Hyper-parameters for MAE pre-training on Atari games.

Hyper-parameter	Value
Mask ratio	75%
Batch size	64
Epochs	50
Warm up epochs	5
Weight decay	0.05
Base learning rate	0.001
Learning rate	base_learning_rate * batch_size / 256
Optimizer	AdamW(β_1 =0.9, β_2 =0.95)

(see Table 9). Pixel values are pre-processed by normalizing from [0,255] to [0,1]. Table 6 shows the number of frames to pre-train each game. Dataset will be randomly shuffled before training. Table 6 illustrates the length of trajectories for per-training.

Table 6. Total number of frames for pre-training.

Game	# Frames
Frostbite	5K
MsPacman	50K
Seaquest	50K
BattleZone	50K

As demonstrated in Table 7, our method has the least requirement for pre-training in terms of data quality and quantity[2] compare with other baselines. We could expect better performance if datasets with better quality (e.g., more frames from better rollouts) are used or online trained with RL.

A.3 Configuration to Compare Pre-training Performance

To visualize and compare the performance of salient part selection between `SPIRL` and baselines, we pre-train Transoprter and PermaKey from scratch with the same environment and machine configuration. Pre-training experiments are run on a single GPU card from a server with 4×GeForce RTX 2070 Super 4×7.8Gi GPU and 4×64 GB memory. The configuration to produce Fig. 4 is listed in Table 8. We keep the same setting as in the published code from PermaKey[3]

[2] About the length of trajectories needed for Transporter pre-training, the authors only said that 100K pairs of frames are needed, but their did not give the exact value of $len(\tau)$ to generate them. According to their description that a diverse dataset is needed and the re-implementation code from PermaKey, we think the value should be larger than `SPIRL`.

[3] The authors have code for both PermaKey and Transporter

Table 7. Comparison of the training settings between SPIRL and other methods. $len(\tau)$ denotes the length of trajectories for pre-training. The 3^{rd} row compares use which kind of policy to collect trajectories, where *Rand* refers to uniform random policy and *Trained* refers to pre-trained policies with existing RL algorithms from Atari Model Zoo [36]. Moreover, since Transporter's pre-training data needs diversity, we mark it as *Rand-Diverse*. Fine-tune set to T means online training with RL policy is required and F means not.

Config		SPIRL	PermaKey	Transporter	MOREL
τ	$len(\tau)$	<50K	85K	——	100K
	Policy	*Rand*	*Trained*	*Rand-Diverse*	*Rand*
Fine-tune		F	T	F	T

except that we use 16 as the pre-training batch size for PermaKey since it is memory costly and larger number of batch size can not be satisfied on our test machine.

Table 8. Pre-training configuration and time cost.

	Transporter	PermaKey	SPIRL
$len(\tau)$	85K	85K	≤50K
Training epochs	100	50	50
Batch size	64	16	64
Time(hours)	25.9	50.7	1.6

A.4 Visualization to Compare Salient-patches / Keypoints Selection

See Fig. 7.

B RL Training Details

B.1 Environment Wrapper

Our environment configurations follow standard requirements [28], which is also adopted in baselines we compared with, except that we use RGB frames down-sampled from shape $210 \times 160 \times 3$ to $96 \times 96 \times 3$. Table 9 lists the related details.

B.2 Rainbow Configurations

We base our code on the original Rainbow implementation, but with modified hyperparameters introduced by DE-Rainbow as shown in Table 10. Compared with original Rainbow with CNN features of the whole frame as the input to its

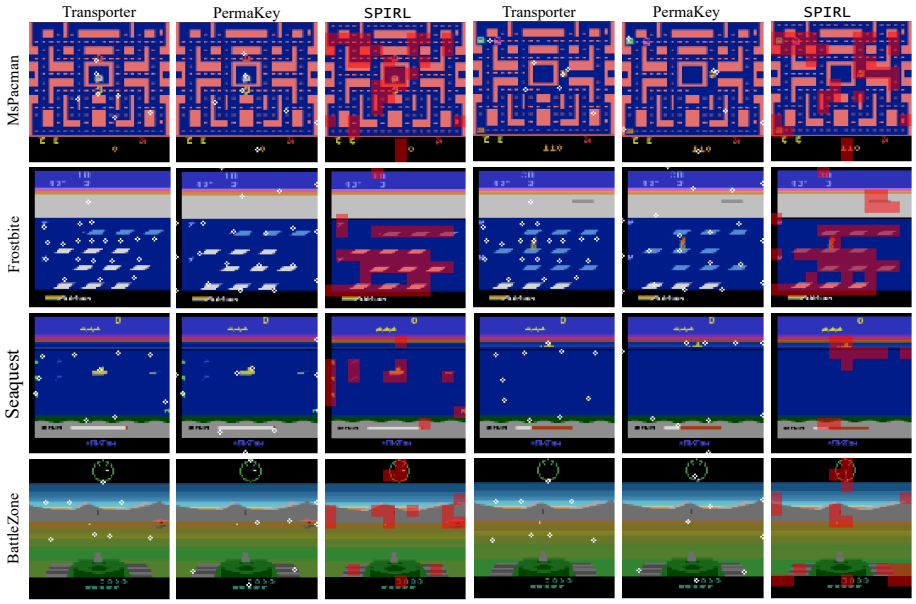

Fig. 7. More visualization to compare SPIRL salient patch selection with baselines. The left 3 columns and right 3 columns corresponding to 2 different frames.

policy network, we use selected patch embeddings from the frozen pre-trained MAE encoder. We do not fine-tune the pre-trained encoder, therefore we can save the selected patch embeddings to replay buffer directly to reduce computational cost for off-policy RL training.

B.3 Evaluation Details

For Table 2 and Table 3, numbers without (resp. with) parentheses are average values (resp. standard deviation). Best results for each game are marked in bold. For each trial, average score and median score is tested with 50 different environments using different seeds. The experimental results are averaged over 5 trials.

B.4 Source of Experimental Results for Table 2

The results of SimPLe, Transporter, and PermaKey come from their original papers. For PermaKey, we take the best results among their CNN and GNN versions.

Since the original DE-Rainbow's [39] code is not published, its results are evaluated with our re-implementation by replacing SPIRL's feature extractor

Table 9. Hyper-parameters for environment wrapper.

Hyper-parameter	Value
Grey-scaling	False
Observation down-sampling	(96, 96)
Frame stacked	4
Frame skipped	4
Action repetitions	4
Max start no ops	30
Reward clipping	[-1, 1]
Terminal on loss of life	True
Max frames per episode	108K

Table 10. Hyper-parameters that differs from the original Rainbow setting [16] used in our implementation.

Hyper-parameter	For both *100K* and *400K* experiments	
Minimal steps to replay/update	1600	
N-step	20	
DQN hidden size	256	
Learning rate	0.0001	
#Training updates	100K	
Hyper-parameter	*100K*	*400K*
Priority β increase steps	100K	400K
Buffer size	100K	400K
Steps per training update	1	4

with the data-efficient CNN as introduced in their paper. Note that `SPIRL` already uses hyper-parameters introduced in [39] to configure the other RL part as explained in Appendix B.2. Table 11 compares the scores between our re-implemented Rainbow and the reported scores. To make the evaluation more reliable, we use 10 trials while the original paper only uses 5.

Table 11. Comparison of our evaluation (averaged over 10 seeds) of DE-Rainbow against the performance reported in [39].

Game	Reported	Ours
Frostbite	**866.8**	341.4(277.8)
MsPacman	**1204.1**	1015.2(124.3)
Seaquest	354.1	**396.5(124.4)**
BattleZone	10124.6	**10602.2(2299.6)**

As for DE-Rainbow-P, Table 12 compares the different hyper-parameter settings: (1) we set the minimal steps to replay for each game as the size of our pre-training dataset, which is a much larger value than the default (50K or 5K v.s. 1.6K); (2) we keep the total number of learning steps and replay buffer size the same with default. Since policy will use a total random policy to interact with environments before reaching minimal steps to replay, DE-Rainbow-P equals to fill the replay buffer with the same data as our pre-training datasets.

Table 12. Compare hyper-parameter settings for DE-Rainbow-P against DE-Rainbow and `SPIRL`.

Configuration	DE-Rainbow-P		DE-Rainbow/`SPIRL`
	MsPacman/BattleZone/Seaquest	Frostbite	All Games
Minimal steps to replay	50K	5K	1.6K
#Env steps	148.4K	103.4K	100K
Buffer size	100K	100K	100K

B.5 Attention RL Implementation

Architecture As shown in Table 13, our attention module includes a single transformer layer with pre-layer normalization, without residual connection inside the transformer layer, and use a trainable $[cls]$ as the pooling method. We kept the same positional embedding setting as MAE, which are pre-defined 2-D non-trainable sinusoid embeddings.

Table 13. Configurations about our Transformer-based model architecture.

Configuration		Value
Attention Depth		1
Number of Attention Head		8
Projected Embedding Dim		32
Add Residual inside Attention Block		False
Pooling Method		Trainable $[cls]$ token
Positional Embedding		2-D non-trainable sinusoid embeddings
Configuration	Game	Value
Maximal Ratio	Frostbite	35%
	MsPacman	30%
	Seaquest	20%
	BattleZone	30%

Guidance to Choose the Maximal Ratio *mr* We exploit the pre-training dataset to provide a guidance to determine a suitable mr by counting the number of selected salient-patches of frames in datasets. Since we don't want to lose information about salient information, we choose an ideal maximal ratio from a discrete set $\{5\%, 10\%, \ldots, 90\%, 95\%\}$ as mr^* that can guarantee salient maintenance for more than 99.9% of frames in pre-training datasets. Practically we try $\{mr^* - 5\%, mr^*, mr^* + 5\%\}$ and select the best one as the adopted mr. As shown in Fig. 8, the ideal ratio mr^* is selected by guaranteeing more than 99.9% will not lose salient information, while the adopted ratio is selected from $\{mr^* - 5\%, mr^*, mr^* + 5\%\}$ according to the actual performance.

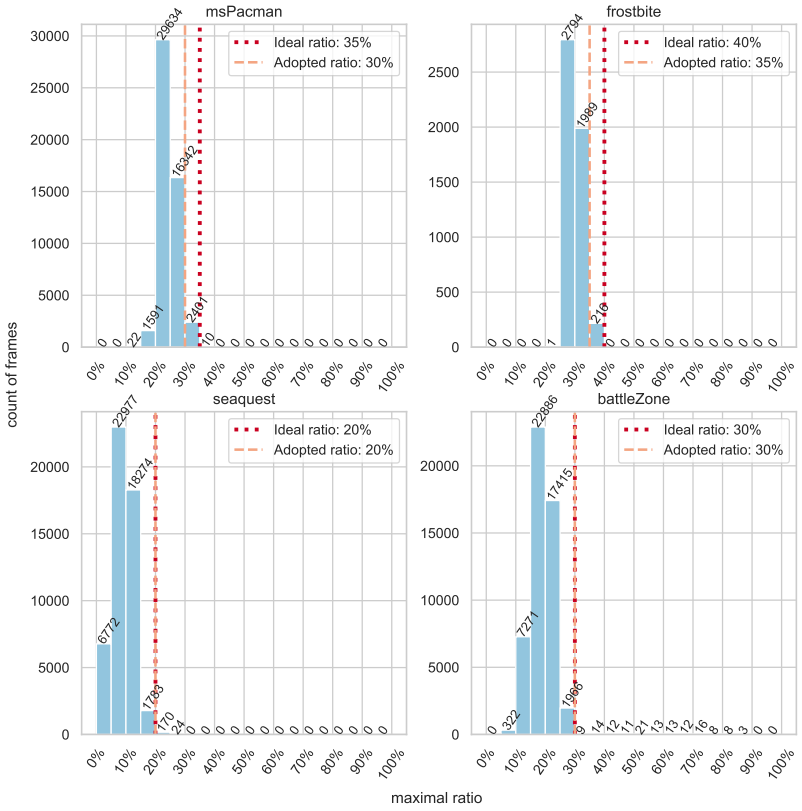

Fig. 8. Histograms based on the count of salient patches in frames from the pre-training datasets of 4 games.

C More Experimental Results

C.1 Different Solutions to Deal with Dynamic Number of Embeddings for Transformer-based RL

We mainly test 3 different solutions on *Seaquest*: (1) Zero-Padding, which add dummy patches with zero padding when the number of selected patches is less than the maximal number; (2) Trainable-[*pad*], which use a trainable [*pad*] token to replace non-salient-patches' embedding before inputing RL part; (3) Masked-Attention, which use zeros to replace the softmaxed attention score for non-salient-patches inside the policy's Transformer blocks. Table 14 shows the efficiency of Zero-Padding solution, which is also the easiest to implement one.

Table 14. Comparison of different solutions to deal with varying number of embedding inputs for RL. Tested on *Seaquest* under *100K* setting.

Method	Zero-Padding	Trainable-[*pad*]	Masked-Attention
Score	**557.9(148.1)**	491.6(139.6)	494.40(154.8)

C.2 Ablation study about Transformer-based RL Architecture

Instead of using a trainable class token [*cls*] in the Transformer-based aggregation in the RL part, we also try a simple average pooling. Table 15 shows the ablation study about the pooling methods. Interestingly, the average pooling performs very well in the *100K* setting, but [*cls*] pooling reveals its strength in the *400K* setting.

Table 15. Aggregation methods in Transformer-based RL. We use a unified maximal ratio (50%) for fair ablation.

Games	*100K*, average		*400K*, median	
	average	[*cls*]	average	[*cls*]
Frostbite	**604.3(788.7)**	425.7(563.3)	689.0(950.8)	**1407.1(1312.4)**
MsPacman	916.3(124.1)	**957.6(236.2)**	1113.0(187.5)	**1176.0(209.3)**
Seaquest	507.8(82.4)	**541.1(89.1)**	550.0(178.6)	**552.0(97.8)**
BattleZone	**10412.0(2535.1)**	8447.1(1491.3)	12900.0(2012.4)	**13500.0(1870.8)**