Supplementary Material

A. Implementation Details

In this section, we first describe the architectures of each network component in our proposed model, followed by details of the training and fine-tuning processes. Finally, we present a table to summarize our hyperparameters.

A.1. Model Architecture

For clarification, while our proposed model contains all five components detailed below, the baseline models only contain the feature extractor f_e and the classifier f_c. We also describe the difference in the two model variants: one with the fully connected classifier and the other with the cosine similarity classifier, which we denote as FC and CS, respectively.

Feature extractor f_e. (Fig. 1) The feature extractor f_e is implemented with a Conv-4 structure, which takes either the entire image x (the baseline settings) or a patch p_i (our proposed model) as input and produces the feature embedding (e or e_i, respectively). Each convolutional block consists of the Conv-BN-ReLU-MaxPool structure, with the convolutional layer having a 64 channel output and the pooling layer with a 2×2 max-pooling operation. We utilize "same" padding technique for the max-pooling layers. Four of these convolutional blocks are stacked together to form the Conv-4 backbone. For the CS variants, we removed the ReLU operation in the final convolution block leaving only the Conv, BN, and the MaxPool layer.

Action context encoder f_a. (Fig. 2) The action context encoder f_a takes the global context g (i.e., features produced by the policies π_θ) and the action a_i, producing an action context vector c_i. a_i is first encoded with a fully connected layer without any activation functions, then is multiplied with g and passed through a ReLU activation. Finally, a fully connected layer with a ReLU activation is used to produce c_i. For the CS variants, we removed the ReLU operation in the final fully connected layer.

State encoder f_s. The state encoder f_s is implemented with a GRU and takes the previous state s_{i-1} and the concatenation of both e_i and c_{i-1} as input, producing the current state s_i. For the CS variants, we remove the tanh nonlinearity before the output.

Maximum entropy sampler f_Q and π_θ. (Fig. 3 and Fig. 4) f_Q and π_θ has an identical architecture, except for minor difference in the input and output. We experimented with both Leaky ReLUs and ReLUs but observed no significant difference. Here we report what we used in our final implementation. Note that while the positive and negative policy have different weights, they share the same network architecture.

f_Q takes the image x, the state s_j, and the action a_j as input. Note that the f_Q is not used during the feed-forward pass. The inputs are sampled from stored transitions stored in the replay buffer during training (backprop). The image x is encoded with 3 convolutional layers consisting of 64, 64, and 1 output channels, followed by a fully connected layer. The state s_j is encoded with two fully connected layers. The action a_j is also encoded with two fully connected layers. Leaky ReLUs are used as an activation function for all the layers except for the final fully connected layers in each submodule mentioned above. The three embeddings are added together, followed by a ReLU activation and is fed to a final connected layer without any activations to produce the final Q-value for the given state-action pair.

π_θ takes the image x, the state s_i, and a noise vector n_i as input. x and s_i are encoded in the same way as in f_Q. Instead of the action a_i, we now encode n_i with two fully connected layers. Like in f_Q, Leaky ReLUs are used as an activation function for all the layers except for the final fully connected layers in each sub-module. The three embeddings are added together, followed by a ReLU activation, producing an aggregated feature g. This feature g is fed into f_a to provide some global context. Finally, g is also fed into a final connected layer with a tanh activation to produce a 2D vector ranging from -1 to $+1$, which corresponds to the action a_i.

Classifier f_c. The classier f_c takes the features of the whole image e (the baseline settings) or the final state s_N (our proposed model) as input and predicts a label for the input image x.

For the FC variant, f_c is a fully-connected layer with

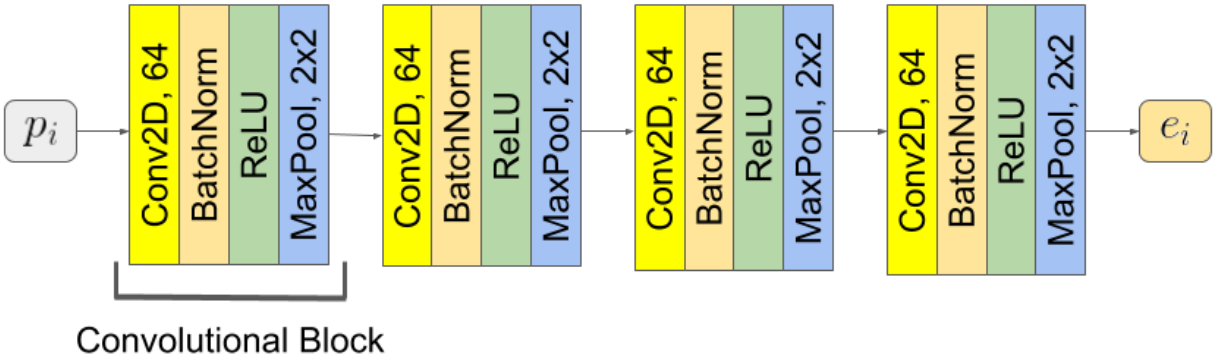


Figure 1: Schematic diagram for the feature extractor f_e that shares the same architecture among all models. For the CS variant, the ReLU activation in the final block is removed leaving only Conv, BN, and MaxPool layer.

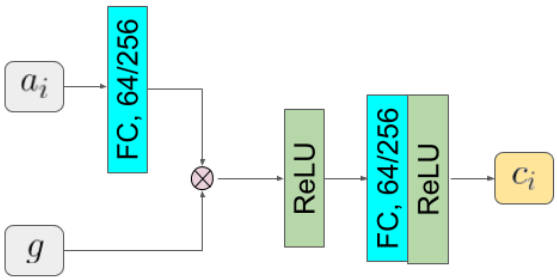


Figure 2: Schematic diagram for the action context encoder f_a. The two numbers (X_1/X_2) proceeding the FC layers indicate the number of output neurons for Omniglot (X_1) and miniImagenet (X_2), respectively. For the CS variant, the final ReLU activation is removed.

the number of output logits equal to the number of classes present followed by a softmax activation. For the CS variant, f_c is a cosine similarity layer. The output is then multiplied by a learnable scaling factor k before being fed into the softmax activation, that is:

$$\hat{y}_i = \frac{exp(k\overline{w}_i^T \bar{s}_N)}{\sum exp(k\overline{w}_i^T \bar{s}_N)}, \quad (1)$$

where $\bar{s}_N$ is the normalized final state vector, i.e., normalized input, and $\overline{w}_i$ is the normalized columns of the weight matrix.

A.2. Training and Fine-tuning

For training and fine-tuning, we use separate classifiers, i.e., f_{c1} and f_{c2}, as the number of classes are different. When we fine-tune on the novel classes, the weights of all other components are frozen and we only update the weights in the "novel" classifier f_{c2}. We used the Huber loss to update f_Q.

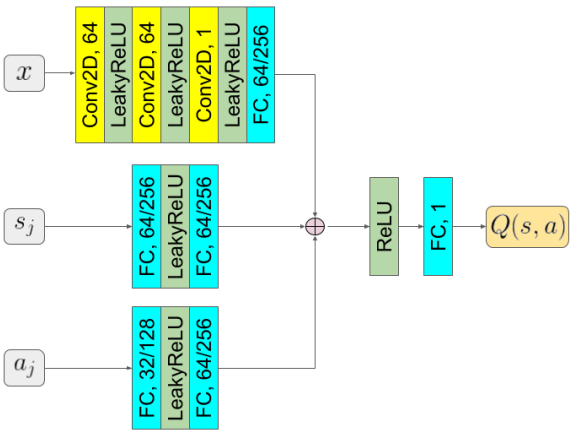


Figure 3: Schematic diagram for the Q-function f_Q. The two numbers (X_1/X_2) proceeding the FC layers indicate the number of output neurons for Omniglot (X_1) and miniImagenet (X_2), respectively. We use the subscript j to indicate that the inputs are sampled from stored transitions instead of the current feed-froward pass.

We train our baseline models with a learning rate of 10^{-3} via the ADAM optimizer. For our proposed model, we found that pre-training with a lower learning rate of 10^{-4} via the ADAM optimizer is more stable. During fine-tuning, we modified the learning rate to 10^{-3} and fine-tune on the "novel" classifier f_{c2}.

A list of the hyperparameters we used for training our proposed model can be seen in Table 1, Table 2, and Table 3.

B. Additional Ablation Studies

Due to the limited space in the main paper, here we present the results of additional ablation studies. We per-

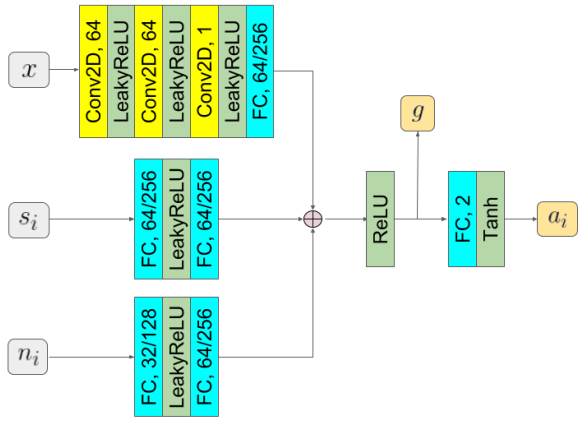


Figure 4: Schematic diagram for the policy π_θ. Note that the two policies π_+ and π_- share the same architecture but have separate weights. The two numbers (X/Y) proceeding the FC layers indicate the number of output neurons for Omniglot and miniImagenet, respectively.

Table 1: Input and output dimensions for the Omniglot and miniImagenet dataset. Here we list the dimensions of the inputs (denoted as [I]) and outputs (denoted as [O]) in our modules for clarity. Note that the input image x size in miniImagenet is 64×64 (instead of 84×84) after performing standard data augmentation techniques (i.e., random cropping). B denotes the batch size.

		Omniglot (20-way)	miniImagenet (5-way)
f_e	p_i [I]	$B \times 16 \times 16 \times 1$	$B \times 24 \times 24 \times 3$
	e_i [O]	$B \times 64$	$B \times 256$
f_a	a_i [I]	$B \times 2$	$B \times 2$
	g [I]	$B \times 64$	$B \times 256$
	c_i [O]	$B \times 64$	$B \times 256$
f_s	e_i [I]	$B \times 64$	$B \times 256$
	c_{i-1} [I]	$B \times 64$	$B \times 256$
	s_{i-1} [I]	$B \times 128$	$B \times 512$
	s_i [O]	$B \times 128$	$B \times 512$
f_Q	x [I]	$B \times 28 \times 28 \times 1$	$B \times 64 \times 64 \times 3$
	s_j [I]	$B \times 128$	$B \times 512$
	a_j [I]	$B \times 2$	$B \times 2$
	$Q(s, a)$ [O]	$B \times 1$	$B \times 1$
π_θ	x [I]	$B \times 28 \times 28 \times 1$	$B \times 64 \times 64 \times 3$
	s_i [I]	$B \times 128$	$B \times 512$
	n_i [I]	$B \times 2$	$B \times 2$
	a_i [O]	$B \times 2$	$B \times 2$
f_{c1}	s_N [I]	$B \times 128$	$B \times 512$
	$\hat{y}$ [O]	$B \times (4800 + 1)$	$B \times (64 + 1)$
f_{c2}	s_N [I]	$B \times 128$	$B \times 512$
	$\hat{y}$ [O]	$B \times (20 + 1)$	$B \times (5 + 1)$

form inference with the soft voting scheme, which has been shown in the main text to be favorable over the hard voting scheme. For completeness, we include the results for $N = 1, 3, 5, 7$ votes.

Table 2: Hyperparameters for training the model. For the two final hyperparameters, the two numbers indicate the chosen hyperparameter value for the Omniglot and mini-Imagenet dataset, respectively.

Batch size B	16
Number of Patches N	4
Learning Rate (training)	$1 \times e^{-4}$
Learning Rate (finetuning)	$1 \times e^{-3}$
L_{class} Weighting	$\frac{1}{4800}/\frac{1}{64}$
Prob. of Selecting π_+ (β)	$0.9/0.1$

Table 3: Hyperparameters for performing Soft Q-Learning.

Replay Buffer Size	100000
Discount Factor γ	0.99
Value Samples	16
Kernel Samples	32
Fixed Kernel Sample Ratio	0.5
Entropy Objective Weighting α	1.0

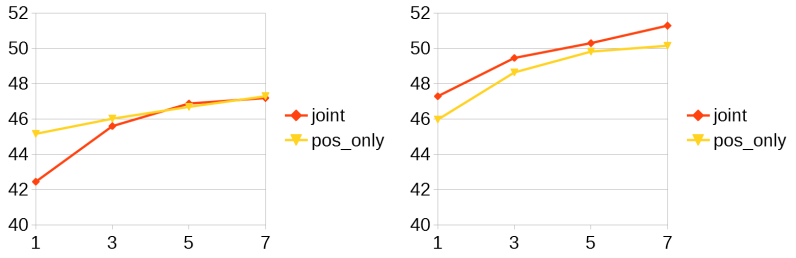
B.1. Impact of Negative Sampling Policy π_-

Here we compare the performance of the model with and without jointly using the positive and negative sampling policies in Fig. 5 and Fig. 6. Note that the reward function is not augmented with the $R_i = -1$ case as the negative sampling policy π_- does not exist. We see that the incorporation of the negative policy can lead to improvements: nearly 1% in the 1-shot setting for the CS classifier and nearly 0.5% in the 5-shot setting for the FC classifier. For the remaining two settings, we observe a difference no bigger than 0.2% and we attribute this to different random seeds.

B.2. Impact of Augmented Reward Function

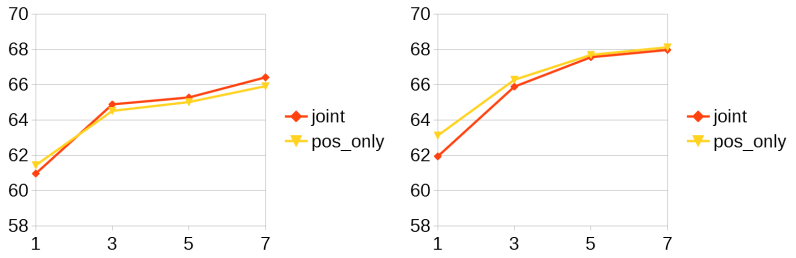
Here we compare the performance of the model with and without the augmented reward function (i.e., with and without the $R_i = -1$ case). Without the reward augmentation, we assign a reward value of 0 for a "background" prediction.

Without the reward augmentation, receiving a "background" label is as "bad" as receiving an incorrect classified label for the positive policy π_+, and vice versa for the negative policy π_-. Note that there is a potential issue with this formulation: if the encoder and classifier are not well trained to extract promising features and perform classification (i.e., classifier is unable to reach 100% accuracy), the positive policy π_+ could be sampling at the locations of interest and still result in a misclassification. However, to the sampler, this is an equally rewarding situation (resulting in a reward of 0) compared with the situation when



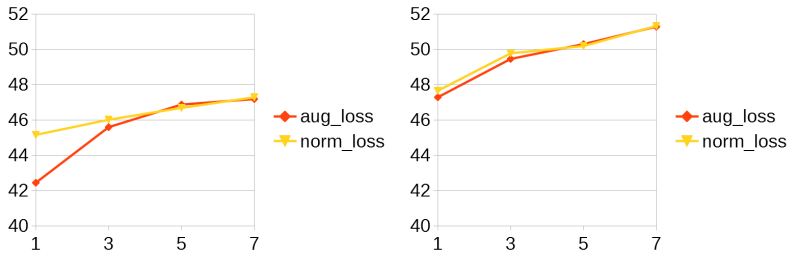
(a) FC Classifier (b) CS classifier

Figure 5: Performance with and without **joint policies** on miniImagenet for (a) FC and (b) CS variants for **1-shot** settings. We observe a noticeable improvement (nearly 1%) for the CS variant with 7 votes.



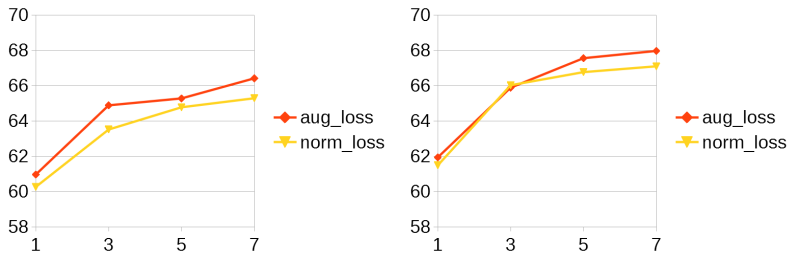
(a) FC Classifier (b) CS classifier

Figure 6: Performance with and without **joint policies** on miniImagenet for (a) FC and (b) CS variants for **5-shot** settings. We notice a slight improvement (around 0.5%) for the FC variant with 7 votes.



(a) FC Classifier (b) CS classifier

Figure 7: Performance with and without an **augmented reward function** on miniImagenet for (a) FC and (b) CS variants for **1-shot** settings. We observe no major differences for the 1-shot settings (within 0.2%).



(a) FC Classifier (b) CS classifier

Figure 8: Performance with and without an **augmented reward function** on miniImagenet for (a) FC and (b) CS variants for **5-shot** settings. We observe a noticeable improvement for both classifiers (around 0.8%) with 7 votes.

the sampler samples at the "irrelevant" locations (leading to a "background" prediction). This could in turn confuse the maximum entropy sampler, which motivates us to introduce an augmented reward function to address such an issue.

To be specific, the three reward terms now corresponds to these scenarios: 1) Reward $+1$: "Relevant" patches, correct classification. 2) Reward 0: Misclassification. 3) Reward -1: "Irrelevant" patches, "background" classification. Note how the introduction of the third term helps avoid the aforementioned ambiguity by accounting for the imperfection of the encoder and classifier. The augmented reward function also complies with the intention of the policies. For the positive policy π_+, it is preferable to sample at the "relevant" patches that leads to the correct classification, less preferable to sample at the patches that leads to a misclassification, and the "irrelevant" patches that leads to a "background" classification should be avoided. For the negative π_-, this order of preference is inverted.

We plot the results in Fig. 7 and Fig. 8. We observed that the augmented reward function provides an improvement of around 0.8% for both classifiers in the 5-shot setting. For the 1-shot setting, we see differences no bigger than 0.2% and we attribute this to different random seeds.

C. Additional Sampling Trajectories

Due to the spaces limitation in main paper, here we present more sampling trajectory visualizations produced by our sampling policy on miniImagenet for the input images from base classes (Fig. 9) and from novel classes (Fig. 10). Note that the samplers were **not** fine-tuned on the novel classes. The order of the sampled patches is: blue, green, red, and white, indicated by the color of the frames. At first glance, we can see that the sampling policy learns to sample on the regions of interest and may also sometimes choose to sample on background patches. We would like to clarify that this is, in fact, the intended behavior, and is the results of our main objective function, where we aim to maximize the action variety of the sampling policy.

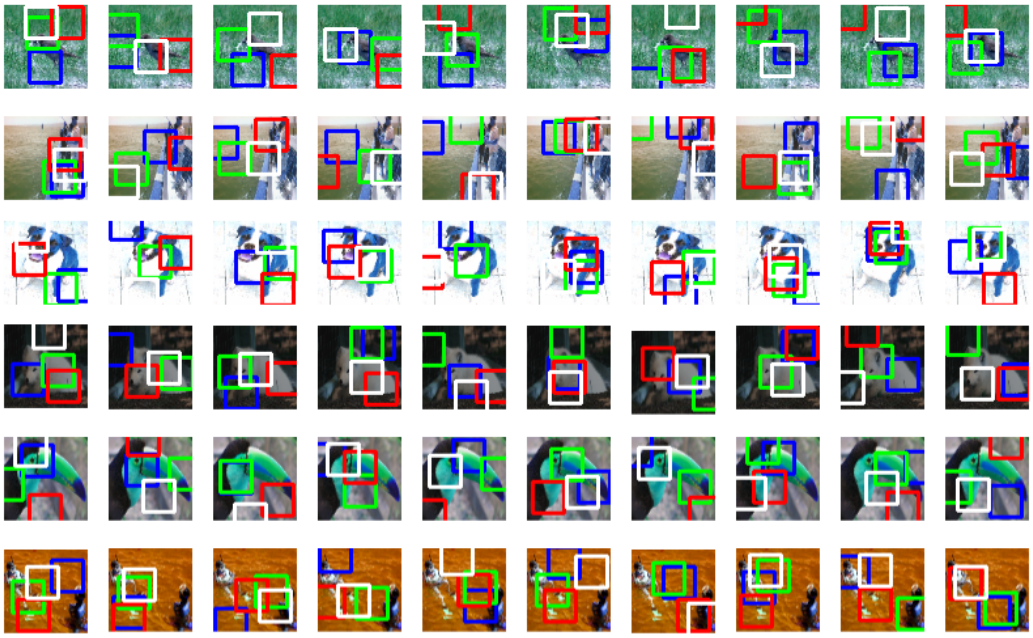


Figure 9: Sample trajectories for different input images from **base classes** in miniImagenet. We use the same input image and run multiple feed-forward passes to extract the different sampling trajectories.

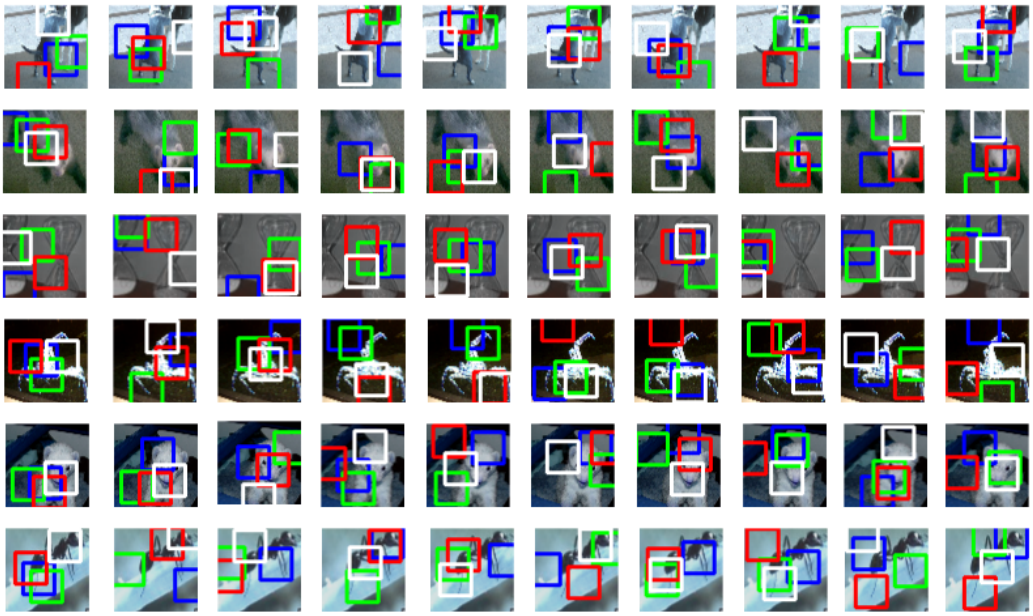


Figure 10: Sample trajectories for different input images from **novel classes** in miniImagenet. We use the same input image and run multiple feed-forward passes to extract the different sampling trajectories. Note that the sampling policy must be able to generalize to unknown classes as the samplers are **not** fine-tuned on the novel classes.