A APPENDIX

A.1 IMPLEMENTATION DETAILS

RainbowMNIST: All the images from Rainbow-MNIST are in 28×28 pixel resolution and with 3 channels. We used a 4 layer convolutional neural network for these experiments, with each layer having [32,32,64,64] number of filters. After every convolution ReLU activation and max pooling is applied to the features. At the last layer we take an average pooling. Finally, 64 dimensional features are passed through fully connected layer with a sigmoid activation. Since the Rainbow-MNIST uses MNIST data to classify, the final layer has 10 neurons, and the objective is to correctly classify the digits of different scales and colors into 10 classes (actual numbers irrespective of the rotation and background color). All the models were trained with 50 gradient updates.

ONLINE-CIFAR100: All the images from CIFAR100 are in 32×32 pixel resolution and with 3 channels. We used a similar 7 layer convolutional neural network in a siamese network architecture for these experiments, with each layer having [32,32,32,64,64,64,128] number of filters. After every convolution ReLU activation and batchnorm is applied to the features. Every other layers have max-pooling. At the last layer we take an average pooling. The L2 distance between two images are calculated and a fully connected layer is used to classify the two images as same class or not.

ONLINE-CELEB: We resized the images from CELEB to 64×64 pixel resolution. We used a similar 7 layer convolutional neural network as in Online-CIFAR100 experiments, with each layer having ([64, 64, 128, 128, 512] number of filters. After every convolution ReLU activation and batchnorm is applied to the features. Every other layers have max-pooling. At the last layer we take an average pooling. The L2 distance between two images are calculated and a fully connected layer is used to classify the two images as same class or not.

A.2 BASELINE METHODS

TFS: Every time this model sees a new task (This needs to know the task boundary), the model resets to new sets of random weights. After that the model is trained only using the data from the new task. At the same time, we also rest the optimizer state to remove any effect of momentum from previous updates. The model is trained with Adam optimizer with a learning rate of 0.001.

TOE: Although TOE optimize the parameters on the all available task, training the model using all the available data is practically impossible. Therefore, we sample datapoints from the buffer and update the model parameters. However, since the number of data is increasing over time, the model also needs be updated on a good representative data of the online stream. Therefore, we increase the number of gradient updates over time. For ONLINE-CIFAR100 experiment, the number of gradients updates are increased by 10 for every 100 tasks. The model is trained with Adam optimizer with a learning rate of 0.001.

FTL: This method, first pretrains a model on the past data, and then fine-tune it on the very recent samples. Similar to TOE, we first train a set of weights using the datapoints in the data buffer. After that, the model is fine-tuned on the correct task data. However, after this adaptation and evaluation the fine-tuned weights are simply discarded and the pretraining starts from the initial weights of the adaptation process. Adam optimizer with same configuration is used to train this model.

FTML: The FTML updates the meta-parameters using MAML style update. For that, we trained the FTML with 5 inner-loop updates (each inner-loop have task-specific data for the inner-loop adaptation). After the meta-update, the meta-parameters are used as the initialization for the inner-loop adaptation, and similar to FTL the adapted weights are simply discarded after evaluation. The inner-loop is trained with SGD with a learning rate of 0.001 and the outer-loop is trained with a learning rate of 0.0005.

FOML: Our method also have two sets of parameter vectors, thus two different optimizers. We use Adam for both online and meta updates with a learning rate of 0.001 for both cases. Since, our online updates share the meta-knowledge via the regularization term, the β_1 controls how much pull is applied to the meta parameters by the online parameters. We use 0.01 for β_1. Similarly the pull on meta-parameters by the online parameters is controlled by β_2, and we use 0.001 in our experiments.

A.3 ADDITIONAL ABLATIONS

Memory size: The main experiments in the paper has access to an unbounded memory to train the meta parameters, however this is not feasible in real world settings. Therefore, we evaluate our method on two sampling strategies. **a)** we limit the memory to store only the last 50 tasks, providing a fixed memory size (like iCaRL) that does not grow over time (FOML-mem-a in Table 1). **b)** We use an exemplar-based memory where we store a part of the data stream (only 10%) into the memory. These samples serve as a set of exemplars for this task (FOML-mem-b in Table 1). In both experiments, FOML performs similarly to a model with access to full memory. We report the test errors of the baseline methods and ours for every 200 tasks on OnlineCIFAR100 dataset.

Variable classes: We also evaluate our method on harder tasks to handle variable numbers of classes. We ran FOML on CIFAR100 with each task containing a random variable number of classes (randomly chosen between 5 and 10) without any modification in the algorithm. This experiment is in Table 1, row "FOML-c." It shows no substantive degradation compared to the standard CIFAR100 experiment, indicating that FOML can handle variable numbers of classes without modification. Although the number of classes are varying, we evaluate this method on a fixed samples to get average error, therefore these errors are comparable.

	Test Error					
Method	200	400	600	800	1000	1200
TFS	47.3	47.8	47.4	47.9	47.5	47.5
TOE	38.3	33.3	28.1	25.1	22.2	20.6
FTL	33.9	27.9	22.8	20.5	18.0	16.2
FTML	30.4	22.8	18.2	16.7	15.4	15.7
LWF	35.4	28.6	22.6	18.4	15.4	14.6
FOML (ours)	**22.8**	**16.5**	**14.8**	**15.5**	**14.3**	**14.5**
FOML-mem-a	24.1	18.2	16.3	16.5	15.4	15.4
FOML-mem-b	23.8	17.1	16.8	16.1	15.1	15.3
FOML-c	27.6	19.9	17.2	16.1	15.1	14.5

Table 1: Test error after every 200 tasks of online-CIFAR100.

Runtime analysis: To evaluate the computational complexity of different approaches, we report the runtime (training) of various methods in different settings. We measure the time required to train a single task and also measure the time required to achieve a certain percentage of error in the prediction. As shown in Tabel 2, although per task training time is a bit slower, FOML requires less time to train on the full sequence of tasks compared to many other methods.

Time (mins)	ICARL	LWF	TOE	FTL	FTML	FOML
per task	2.5	3.3	3.3	**2.3**	**2.3**	5.9
25% error	1324.5	639.3	1109.3	330.4	559.0	**157.2**
20% error	-	1352.9	-	714.8	887.3	**240.7**
15% error	-	3666.2	-	-	1970.1	**566.0**

Table 2: Run time: Train time (in mins) on single NVIDIA 2080 GPU.

Effect of noise: In this experiment we study the effect that additive noise corruption of the images has on our method. We first test the condition where the online stream of images is corrupted with Gaussian noise with different strength, while the images used in the meta-update are uncorrupted. The noise is zero mean with 3 different variances (0.05, 0.10, 0.15). In the second setting, we add noise only to the images stored in memory and used for meta-update, while the online streaming images are uncorrupted. The results in Table 3 shows that in both cases noise have only a small effect in the performance of our method.

	Test Error					
β_1, β_2	200	400	600	800	1000	1200
online images (5%)	25.5	22.1	19.9	20.1	19.2	19.4
online images (10%)	25.7	21.4	20.1	20.6	19.3	19.4
online images (15%)	25.7	22.0	19.8	21.6	20.8	19.9
memory images (5%)	25.2	21.2	19.9	20.9	19.4	19.4
memory images (10%)	26.1	21.7	20.3	20.8	19.2	19.4
memory images (15%)	25.0	20.7	19.9	19.7	18.9	18.9

Table 3: Test error after every 200 tasks, on online-CIFAR100 after corrupting images with Gaussian noise in the inner-loop (online adoptation) and meta update.

Hyper-parameter sensitivity: To study the sensitivity of various hyper-parameters in our algorithm, we vary the regularization coefficients β_1 and β_2 and measure the effect on the performance of FOML on the online-CIFAR dataset. Table 4 shows the results of this experiment. The variance in the test error with different choices of β_1 and β_2 is small, suggesting FOML is not very sensitive to the choice of these hyper-parameters.

	Test Error					
β_1, β_2	200	400	600	800	1000	1200
$\beta_1 = 1e-3, \beta_2 = 1e-3$	24.8	19.7	18.0	18.3	17.5	16.3
$\beta_1 = 2e-3, \beta_2 = 1e-3$	24.8	21.1	20.1	21.6	20.3	18.8
$\beta_1 = 5e-3, \beta_2 = 1e-3$	26.7	22.2	20.3	21.0	19.3	19.2
$\beta_1 = 2e-4, \beta_2 = 1e-3$	25.0	21.6	19.4	20.2	18.9	19.0
$\beta_1 = 5e-4, \beta_2 = 1e-3$	24.6	20.9	19.5	19.9	19.0	18.7
$\beta_1 = 1e-3, \beta_2 = 1e-3$	24.8	19.7	18.0	18.3	17.5	16.3
$\beta_1 = 1e-3, \beta_2 = 2e-3$	25.3	21.5	19.5	20.0	19.2	18.4
$\beta_1 = 1e-3, \beta_2 = 5e-3$	24.8	21.1	20.1	21.6	20.3	18.8

Table 4: Test error after every 200 tasks, on online-CIFAR100 for various choices of β_1 and β_2.

FOML with task boundaries : One might ask a question what would be the optimal performance if the task boundaries are accessible. We believe FOML with tasks boundaries (FOML+T) should perform better than FOML. To test this hypothesis we train our FOML model, but at the meta-update stage, we allowed it to access task boundaries and sample efficiently from various tasks. However, our experiments shows that FOML+T performs slower than FOML. We will investigate this further and report the conclusion in the final version.

FOML with Reptile : In our FOML formulation, the meta updates requires computing second order deviates of the inner-loop updates. We could make a simple approximation to the problem by taking the first order approximation to this meta updates. This is equivalent to apply a similar version reptile for the meta update in our FOML algorithm. Here, we study the effect of 2nd order meta gradients and the first order approximation for fast online learning. In FOML+R, where "R" corresponds to reptile style update, we directly compute the meta update direction towards the last parameter of the online updates.

$$\theta = \theta - \alpha_2(\phi^j - \theta)\} \tag{11}$$

Here, ϕ^j is the lastly updated online parameters. As shown in the Fig 5, FOML performs much faster than FOML+R. This shows that, while the first order approximation of our FOML algorithm can learn new tasks, this approximation hurts the performance over all. We will add this experiment in the main paper with other datasets in the final version.

A.4 Discussion

Slow-Fast wights: There are few works which use two set of parameters (slow and fast) to learn good representations and achieve generalization. Zhang et al. Zhang et al. (2019) proposed an look-ahead

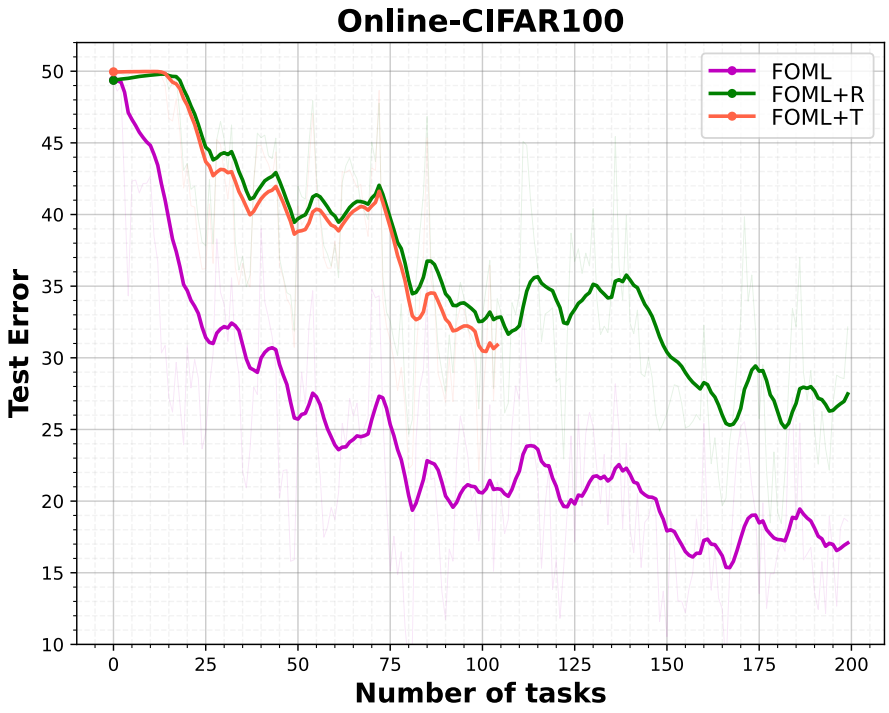

Figure 5: Additional Ablations :

optimizer which first trains the model for k steps in the inner-loop, and then updates the weights towards the direction of k^{th} inner-loop weights. Caron et al. Caron et al. (2021) also keeps two sets of weights, for self-supervised contrastive learning. In their work, they use exponential moving average to update the slow weights from fast weights. While these two works share some similarities with ours by having slow and fast weights, there are various difference between these works and ours. First, we solve a different problem compared to these line of works, our work focus on online learning with fast adoption, while these works aim for better generalization across all tasks/classes. Second difference is that, we can formulate these works as first order meta-learning methods. For example, Zhang et al. Zhang et al. (2019) only computes the direction of the inner-loop to update the outer-loop parameters. However, our method computes the 2nd order gradients to update the meta-parameters. Similarly, for Caron et al. Caron et al. (2021), they update their slow-weights as the exponential moving average of the inner-loop parameters. This is a very simple first order approximation of the meta-updates. However, computing the 2nd order derivative is computationally very complex, therefore for large models it might be hard to scale. In these cases, as shown by Zhang et al. and Caron et al. first order approximation is probably good enough.

A.5 Limitations

Memory: One of our main limitations is that our standard FOML method assumes that the memory buffer is infinite and can store all the past observations. In this work we assume this large memory is available during learning and it will grow over time without bounds. This is not feasible in a real-world setting. However, FOML can utilize more restricted memory buffers to overcome this limitations. We explore two such schemes in A.3, which examines both a fixed-size memory (a) and a memory buffer that stores only 10% of the observed datapoints (b). In both regimes, FOML can attain performance that is only somewhat worse than the standard (unlimited memory) condition, and still better than baselines but the full unlimited memory version of FOML still attains better results.

Non-mutually exclusive tasks: Another limitation of our method is that our experiments are designed such that the transition between each task is very smooth. This design choice makes each tasks to be non-mutually exclusive. While we believe this setting is similar to the tasks that exist in the real world (e.g., weather patterns, recognizing street signs as illumination and background change over time etc.), some settings would consist of mutually exclusive tasks (e.g., settings where the same class

label has different meaning in different tasks). However, any mutually exclusive tasks can be made non-mutually exclusive by mixing some old observation data into the new task. It is also possible to simply formulate the problem as a Siamese classification task, as in our online-CIFAR experiment, where the goal is to classify whether a given input image is of the same class as a given prototype. This does not require any additional assumption, since at least one example image of each class is always necessary to even define the problem.

Shallow backbone: Most of our experiments use a shallow 7-layer ConvNet architecture. One of the bottleneck of training FOML with larger models is that it requires a large GPU memory to compute second order gradients. There are several works which solve this problem by Hessian-Free Meta Learning Song et al. (2019), or having wrap layers Flennerhag et al. (2019). However, in this work we used the vanilla version of meta gradients to update the meta parameters. We have tried to train a larger ResNet model on the same datasets, however it is not easy to train it without any additional tricks to manage the memory use, and tuning the hyper-parameters is challenging. We believe this limitation can be solved by using previously mentioned works to compute the second-order deviates, which we will leave for the future work.

Connection to Distillation: There are some similarities and differences with knowledge distillation and online adaptation in FOML. During the online adaptation we do not perform any knowledge distillation explicitly via minimizing KL divergence of the logits of online model and meta model. However, we having a regularization term allows some “knowledge" to pass to the online model implicitly.