Fast Context Adaptation via Meta-Learning

Supplementary Material

A. Pseudocode

Algorithm 1 CAVIA for Supervised Learning

Require: Distribution over tasks $p(\mathcal{T})$
Require: Step sizes α and β
Require: Initial model $f_{\phi_0,\theta}$ with θ initialised randomly and $\phi_0 = 0$
while not done **do**
  Sample batch of tasks $\mathbf{T} = \{\mathcal{T}_i\}_{i=1}^{N}$ where $\mathcal{T}_i \sim p$
  for all $\mathcal{T}_i \in \mathbf{T}$ **do**
    $\mathcal{D}_i^{\text{train}}, \mathcal{D}_i^{\text{test}} \sim q_{\mathcal{T}_i}$
    $\phi_0 = 0$
    $\phi_i = \phi_0 - \alpha \nabla_\phi \frac{1}{M_i^{\text{train}}} \sum_{(x,y)\in\mathcal{D}_i^{\text{train}}} \mathcal{L}_{\mathcal{T}_i}(f_{\phi_0,\theta}(x), y)$
  end for
  $\theta \leftarrow \theta - \beta \nabla_\theta \frac{1}{N} \sum_{\mathcal{T}_i\in\mathbf{T}} \frac{1}{M_i^{\text{test}}} \sum_{(x,y)\in\mathcal{D}_i^{\text{test}}} \mathcal{L}_{\mathcal{T}_i}(f_{\phi_i,\theta}(x,y))$
end while

Algorithm 2 CAVIA for RL

Require: Distribution over tasks $p(\mathcal{T})$
Require: Step sizes α and β
Require: Initial policy $\pi_{\phi_0,\theta}$ with θ initialised randomly and $\phi_0 = 0$
while not done **do**
  Sample batch of tasks $\mathbf{T} = \{\mathcal{T}_i\}_{i=1}^{N}$ where $\mathcal{T}_i \sim p$
  for all $\mathcal{T}_i \in \mathbf{T}$ **do**
    Collect rollout τ_i^{train} using $\pi_{\phi_0,\theta}$
    $\phi_i = \phi_0 + \alpha \nabla_\phi \tilde{\mathcal{J}}_{\mathcal{T}_i}(\tau_i^{\text{train}}, \pi_{\phi_0,\theta})$
    Collect rollout τ_i^{test} using $\pi_{\phi_i,\theta}$
  end for
  $\theta \leftarrow \theta + \beta \nabla_\theta \frac{1}{N} \sum_{\mathcal{T}_i\in\mathbf{T}} \tilde{\mathcal{J}}_{\mathcal{T}_i}(\tau_i^{\text{test}}, \pi_{\phi_i,\theta})$
end while

B. Practical Tips

B.1. Implementation

The context parameters ϕ can be added to any network, and do not require direct access to the rest of the network weights like MAML. In PyTorch this can be done as follows. To add CAVIA parameters to a network, it is necessary to first initialise them to zero when the model is initialised:

```
self.context_params =
torch.zeros(size=[self.num_context_params],
requires_grad=True)
```

Add a way to reset the context parameters to zero (e.g., a method that just does the above). During the forward pass, add the context parameters to the input by concatenating it (when using a fully connected network):

```
x = torch.cat((x,
self.context_params.expand(x.shape[0],
-1)), dim=1)
```

(This is for fully connected networks. We refer the reader to our implementation for how to use FiLM to condition CNNs.) To correctly set the computation graph for the outer loop, it is necessary to assign the context parameters manually with their gradient. In the inner loop, compute the gradient:

```
grad = torch.autograd.grad(task_loss,
model.context_params,
create_graph=True)[0]
```

The option *create_graph* will make sure that you can take the gradient of *grad* again. Then, update the context parameters using one gradient descent step

```
model.context_params = model.context_params
- lr_inner * grad
```

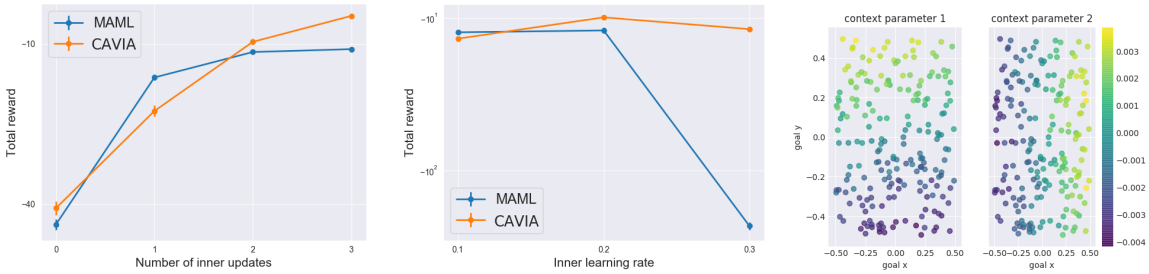
If you now do another forward pass and compute the gradient of the model parameters θ (for the outer loop), these will include higher order gradients because *grad* above includes gradients of θ, and because we kept the computation graph via the option *grad*. To see how to train CAVIA and aggregate the meta-gradient over several tasks, see our implementation at [blinded; see supplementary material].

B.2. Hyperparameter Selection

The choice of network architecture/size and context parameters can be guided by domain knowledge. E.g., for the few-shot image classification problem, an appropriate model is a deep convolutional model. For the context parameters, it is important to make sure they are not underparameterised. CAVIA can deal with larger than necessary context parameters (see Table 1), although it might start overfitting in the inner loop at some point (we have not experiences this in practise). Regarding learning rates, we always started with an inner loop learning rate of 1 and the Adam optimiser with the standard learning rate of 0.001 for the outer loop.

For CNNs, we found that adding the context parameters not at the input layer, but after several (in our case after the third out of four) convolutions works best. We believe this is because the lower-level features that the first convolutions extract are useful for any image classification task, and we only want our task embedding to influence the activations at the deeper layers. In our experiments we used a FiLM network with no hidden layers. We tried deeper versions, but this resulted in inferior performance.

We also tested to add context parameters at several layers instead of only one. However, in our experience this resulted in similar (regression and RL) or worse (in the case of CNNs) performance.



(a) Multiple gradient updates. (b) Learning rate comparison. (c) Goal position embedding.

Figure 6. Results for the 2D navigation reinforcement learning problem.

C. Experiments

C.1. Classification: Details

For Mini-Imagenet, our model takes as input images of size $84 \times 84 \times 3$ and has 5 outputs, one for each class. The model has four modules that each consist of: a $2D$ convolution with a 3×3 kernel, padding 1 and 128 filters, a batch normalisation layer, a max-pooling operation with kernel size 2, if applicable a FiLM transformation (only at the third convolution, details below), and a ReLU activation function. The output size of these four blocks is $5 \times 5 \times 128$, which we flatten to a vector and feed into one fully connected layer. The FiLM layer itself is a fully connected layer with inputs ϕ and a 256-dimensional output and the identity function at the output. The output is divided into γ and β, each of dimension 128, which are used to transform the filters that the convolutional operation outputs. The context vector is of size 100 (other sizes tested: 50, 200) and is added after the third convolution (other versions tested: at the first, second or fourth convolution).

The network weights are initialised using He et al. (2015), the bias parameters are initialised to zero (except at the FiLM layer). We use the Adam optimiser for the meta-update step with an initial learning rate of 0.001. This learning rate is annealed every $5,000$ steps by multiplying it by 0.9. The inner learning rate is set to 0.1 (others tested: 1.0, 0.01). We use a meta batchsize of 4 and 2 tasks for 1-shot and 5-shot classification respectively. For the batch norm statistics, we always use the current batch – also during testing. I.e., for 5-way 1-shot classification the batch size at test time is 5, and we use this batch for normalisation.

C.2. Reinforcement Learning: Additional Experiments

We also perform reinforcement learning experiments on the simple 2D Navigation task of Finn et al. (2017a). The agent moves in a 2D world using continuous actions and at each timestep is given a negative reward proportional to its distance from a pre-defined goal position. Each task has a new unknown goal position.

We follow the same procedure as Finn et al. (2017a). Goals are sampled from an interval of $(x, y) = [-0.5, 0.5]$. At each step we sample 20 tasks for both the inner and outer loops and testing is performed on 40 new unseen tasks. We learn for 500 iterations and optimise for one gradient update in the inner loop. The best performing policy during training is then presented with new test tasks and allowed two gradient updates. For each update, the total reward over 20 rollouts per task is measured. We use a two-layer network with 100 units per layer and ReLU nonlinearities to represent the policy and a linear value function approximator. For CAVIA we use five context parameters at the input layer.

Figure 6a shows that the two methods are highly competitive. We think that the similar performance is mostly due to a ceiling effect, since the domain is relatively simple. Notably, CAVIA adapts only five parameters at test time, whereas MAML adapts around $10,000$. Figure 6b, which plots performance for several learning rates (at test time, after two gradient updates), shows that CAVIA is again less sensitive to the inner loop learning rate. Only when using a learning rate of 0.1 is MAML competitive in performance.[3]

As with regression, the optimal task embedding is low dimensional enough to plot. We therefore apply CAVIA with two context parameters and plot how these correlate with the actual position of the goal for 200 test tasks. Figure 6c shows that the context parameters obtained after two policy gradient updates represent a disentangled embedding of the actual task. Specifically, context parameter 1 encodes the y position of the goal, while context parameter 2 encodes the x position. Hence, CAVIA can learn compact interpretable task embeddings via backpropagation through the inner loss.

C.3. Additional CelebA Image Completion Results

The following images show additional results for the CelebA image completion task.

[3] For MAML we halve the learning rate after the first gradient update, following Finn et al. (2017a).

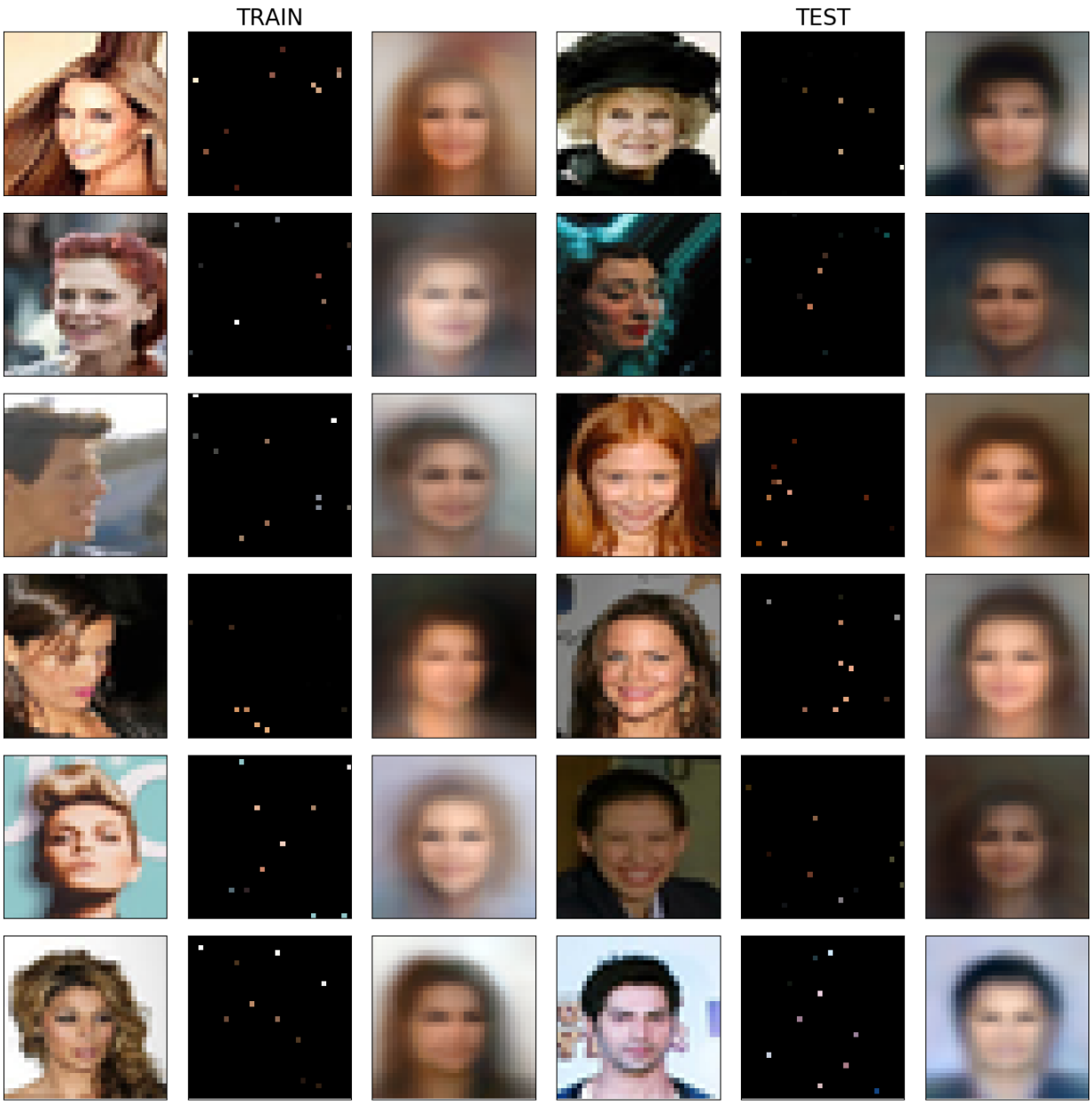


Figure 7. Additional image completion results for the CelebA image completion problem, when $k = 10$ pixels are given.

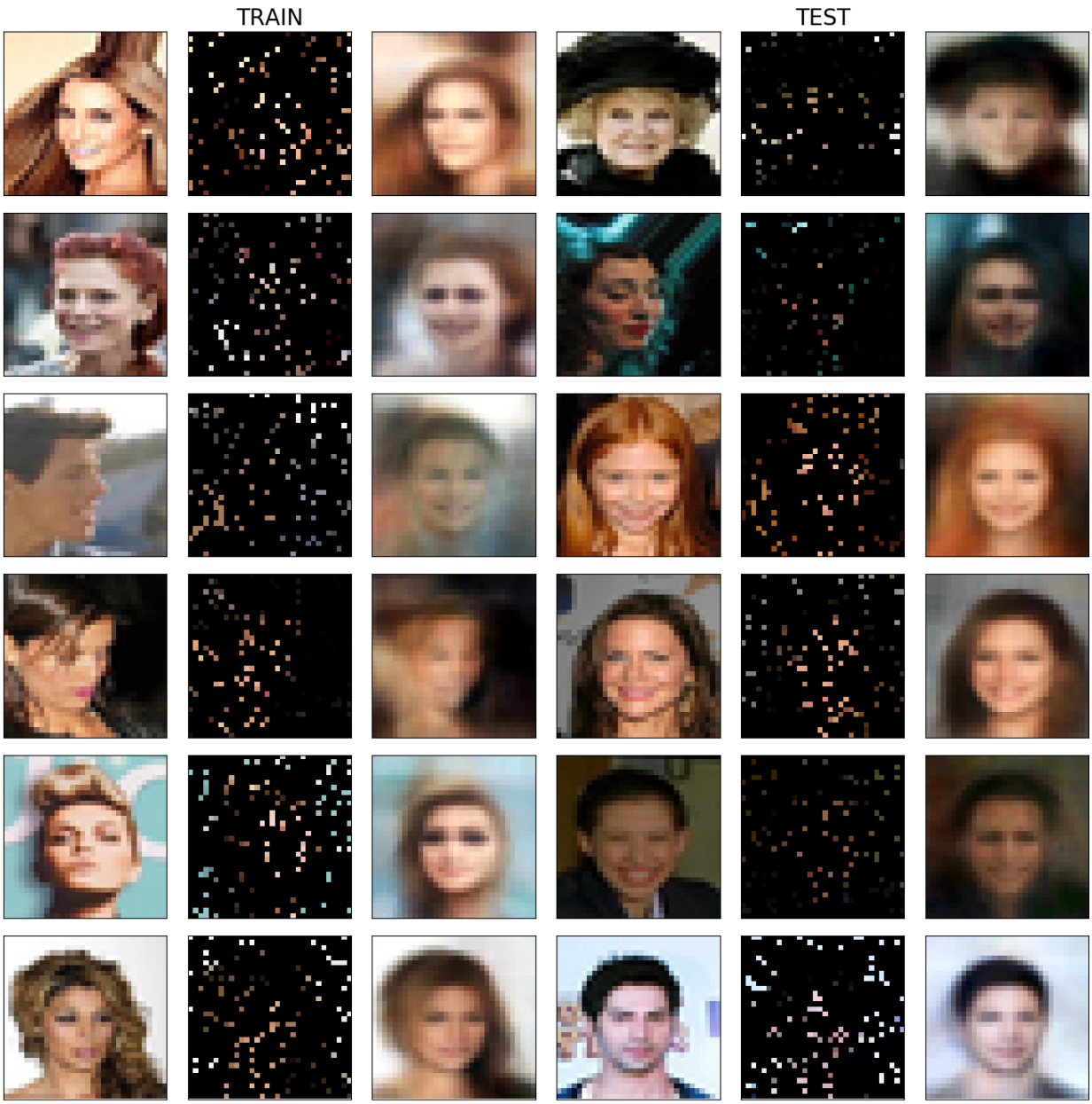


Figure 8. Additional image completion results for the CelebA image completion problem, when $k = 10$ pixels are given.

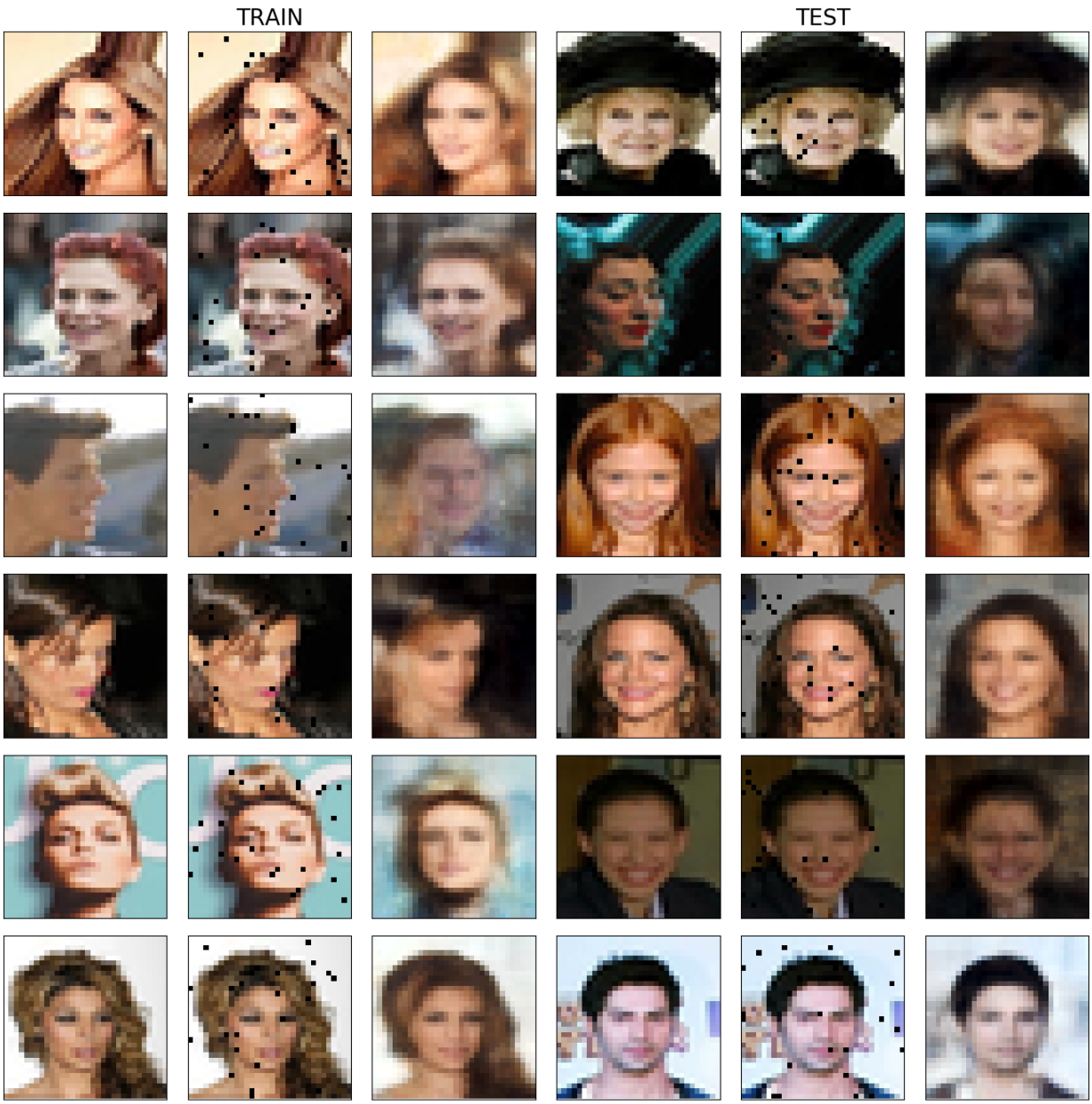


Figure 9. Additional image completion results for the CelebA image completion problem, when $k = 10$ pixels are given.