EECE695D: Efficient ML Systems

Meta-Learning

(part 1)

Meta-Learning

"... an alternative paradigm where

*(1) an ML model gains **experience** over multiple learning episodes —often covering a distribution of related tasks—and*

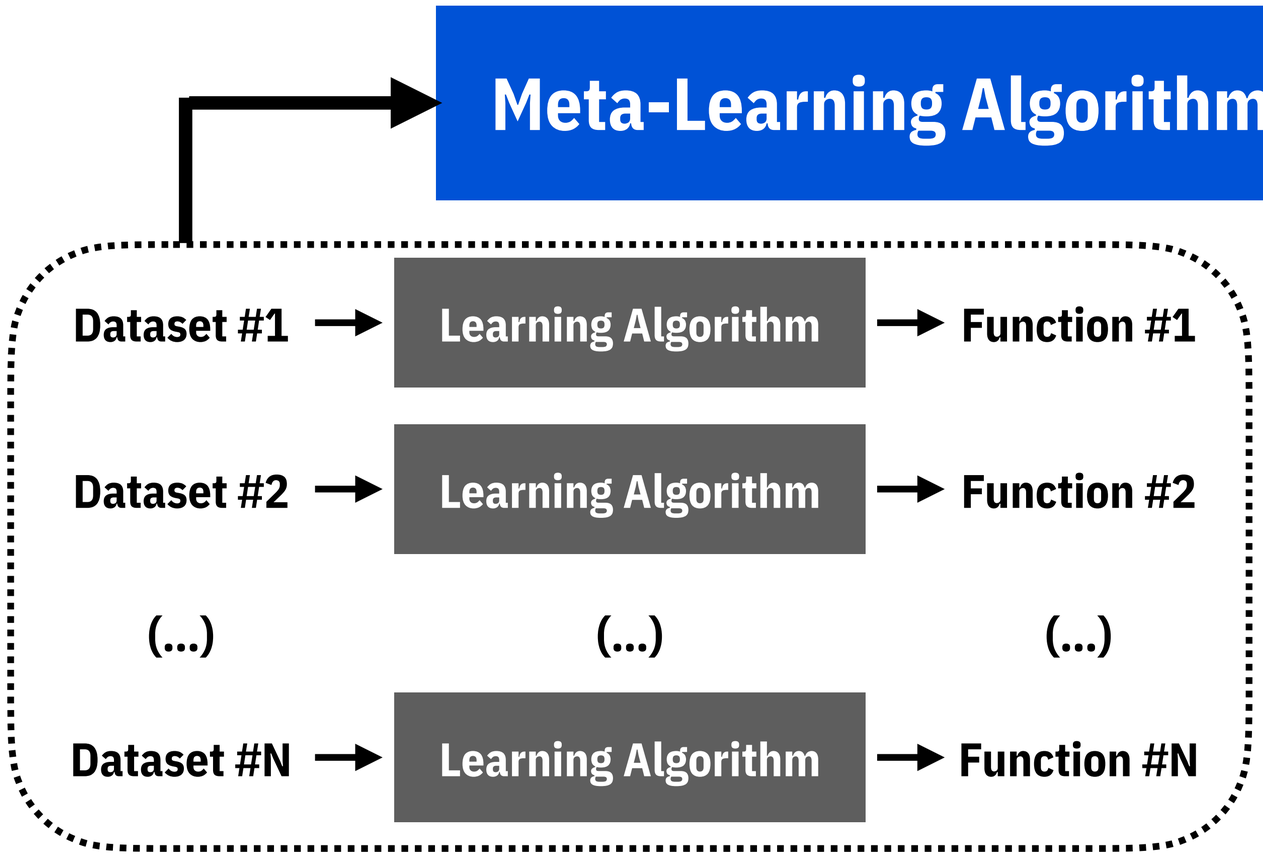
*(2) uses this **experience** to improve its future learning performance."*

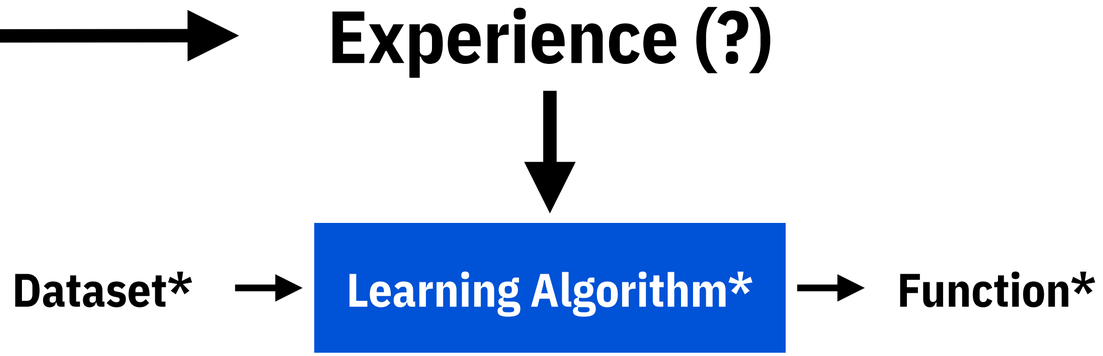
Hospidales et al., "Meta-Learning in Neural Networks: A Survey," 2022.

e.g., a good table-tennis/golf player is also good at tennis

Traditional Paradigm.
In standard ML scenarios, we are given many data instances (training data).
The goal is to find a nice function that works well on this set of data,
so that we expect the function to work well on a new data (test data)

Meta-Learning (a.k.a., learning-to-learn).
Meta-learning assumes that we have many learning tasks at hand (called support set).
The goal is to find a nice "experience/prior/knowledge" that works well on this set of tasks,
so that we expect it to work well also on new tasks (called query set).





Sample-efficient Few-shot Learning
Fast Many-shot Optimization (Less compute)
Robust to Domain Shift
Robust to Label Noise
Robust to Adversarial Attack
...

Formally. Let us define a task as a dataset and loss pair $\mathsf{T} = (\mathsf{D}, \mathsf{L})$.
We posit that there exists some distribution of tasks $p(\mathsf{T})$.

Meta-learning is about finding some meta-knowledge ω that solves

$$\min_{\omega} \mathbb{E}_{\mathsf{T} \sim p(\mathsf{T})} \, \mathsf{L}(\mathsf{D}; \omega)$$

where $\mathsf{L}(\mathsf{D}; \omega)$ measures the performance of the model trained on dataset D
utilizing the meta-knowledge ω.

That is, if we assume that the dataset D can be divided into train-val sets $(\mathsf{D}_{\text{train}}, \mathsf{D}_{\text{val}})$

$$\mathsf{L}(\mathsf{D}; \omega) = \mathbb{E}_{(X,Y) \in \mathsf{D}_{\text{val}}} \ell(f_{\theta^*(\omega)}(X), Y)$$

where $\theta^*(\omega)$ is the model parameter learned by minimizing the loss over $\mathsf{D}_{\text{train}}$
utilizing the meta-knowledge ω during the learning process.

Question. What is ω, and what do we mean by utilizing ω during the learning process?

There are so many meta-learning algorithms, differing in what ω we use (Meta-Representation)
and how to optimize such ω (Meta-Optimizer)

Today, we focus on describing three most popular algorithms, with focus on compute-efficiency.

Meta-Representation	Meta-Optimizer		
	Gradient	RL	Evolution
Initial Condition	[16], [79], [88], [102], [166], [166]–[168]	[169]–[171] [16], [63], [64]	[172], [173]
Optimizer	[19], [94] [21], [39], [79], [106], [107], [174]	[81], [93]	
Hyperparam	[17], [69] [71]	[175], [176]	[173] [177]
Feed-Forward model	[38], [45], [86], [110], [178], [179] [180]–[182]	[22], [114], [116]	
Metric	[20], [90], [91]		
Loss/Reward	[42], [95] [127] [124]	[126] [121], [183] [124]	[123] [23] [177]
Architecture	[18] [135]	[26]	[25]
Exploration Policy		[24], [184]–[188]	
Dataset/Environment	[156] [159]	[162]	[163]
Instance Weights	[151], [152], [155]		
Feature/Metric	[20], [90]–[92]		
Data Augmentation/Noise	[145] [119] [189]	[144]	[146]
Modules	[140], [141]		
Annotation Policy	[190], [191]	[192]	

TABLE 1

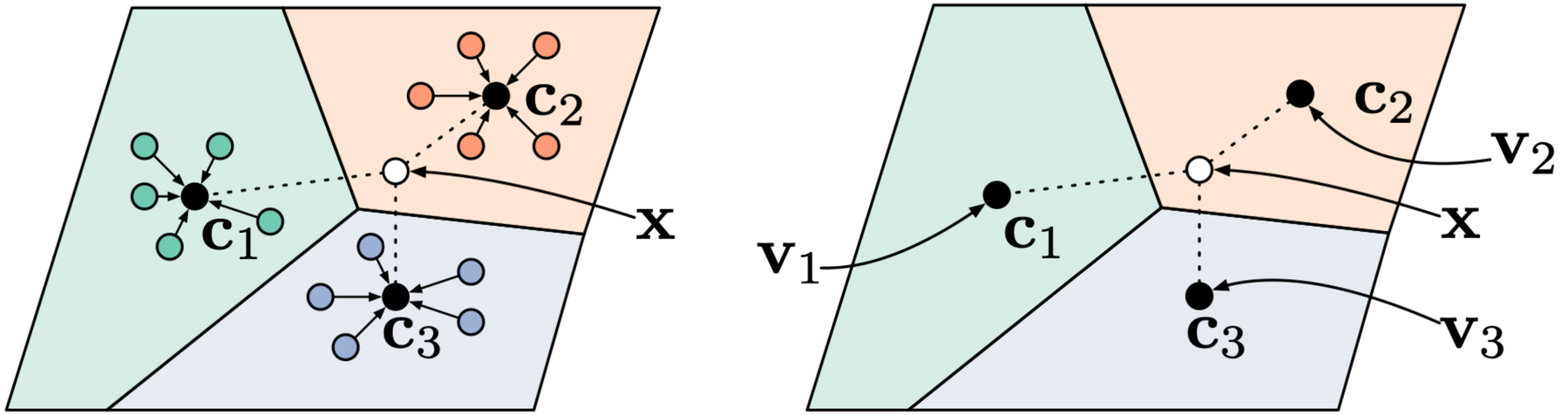
Research papers according to our taxonomy. We use color to indicate salient meta-objective or application goal. We focus on the main goal of each paper for simplicity. The color code is: sample efficiency (red), learning speed (green), asymptotic performance (purple), cross-domain (blue).

Hospidales et al. (2022), "Meta-Learning in NNs: A Survey" IEEE Transactions on PAMI, 2022

ProtoNet. A method proposed by Snell et al. (2017).

inherits a rich line of work including Koch et al. (2015), Vinyals et al. (2016), Ravi and Larochelle (2017)

The key idea is to use "metric learning"—
We use support set data to learn a nice embedding space of the data,
where a nearest-neighbor classification on the embedding space works well.



(a) Few-shot (b) Zero-shot

Figure 1: Prototypical Networks in the few-shot and zero-shot scenarios. **Left**: Few-shot prototypes $\mathbf{c}_k$ are computed as the mean of embedded support examples for each class. **Right**: Zero-shot prototypes $\mathbf{c}_k$ are produced by embedding class meta-data $\mathbf{v}_k$. In either case, embedded query points are classified via a softmax over distances to class prototypes: $p_{\boldsymbol{\phi}}(y = k|\mathbf{x}) \propto \exp(-d(f_{\boldsymbol{\phi}}(\mathbf{x}), \mathbf{c}_k))$.

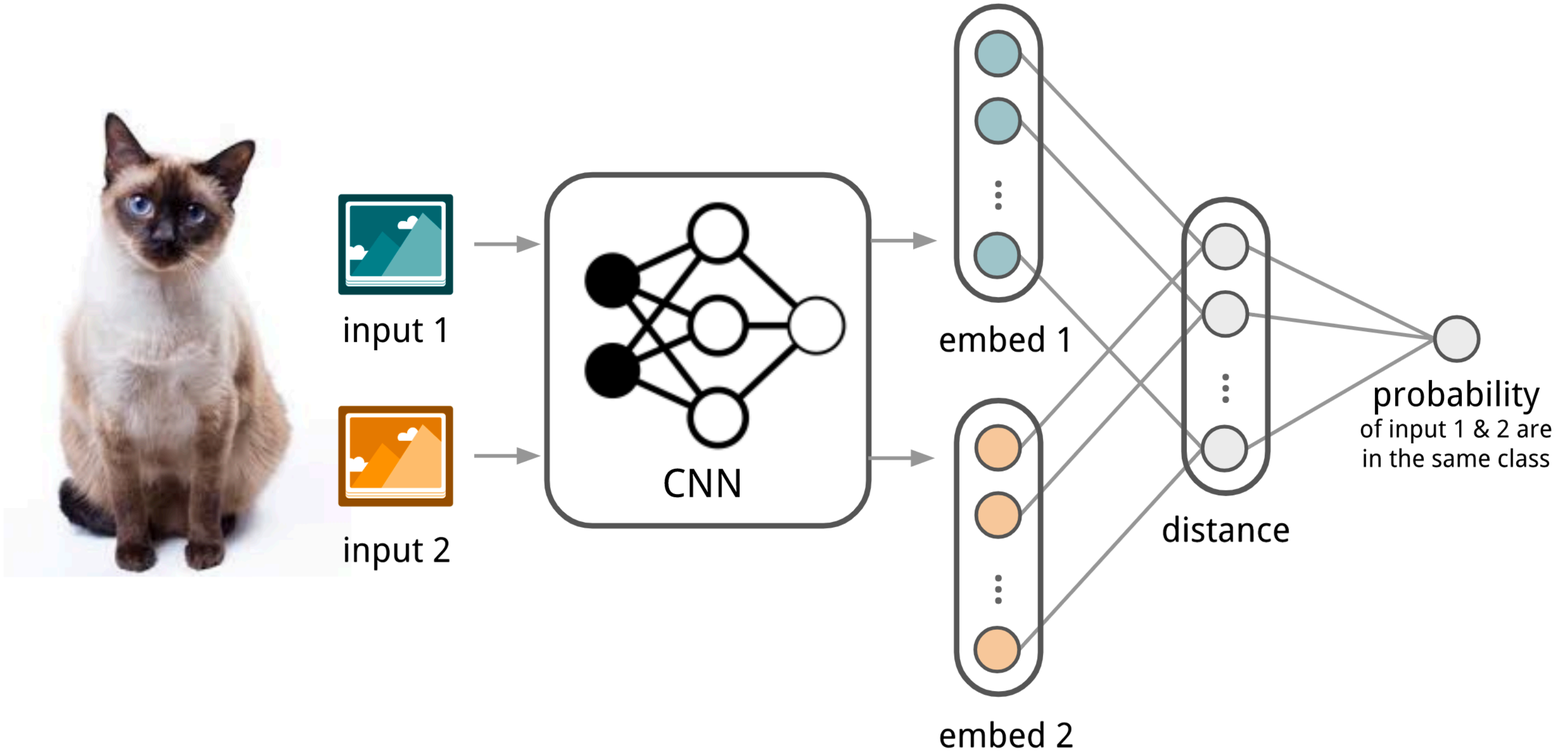
Snell et al., "Prototypical networks for few-shot learning," NeurIPS 2017
Koch et al., "Siamese NNs for one-shot image recognition," ICML 2015
Vinyals et al., "Matching networks for one-shot learning," NeurIPS 2016
Ravi and Larochelle, "Optimization as a model for few-shot learning," ICLR 2017

Here, the meta-knowledge ω is simply the parameter of the embedding CNN.
(We don't have the model parameter θ, as we are using nearest-neighbor!)

Given the embedding map $f_\omega(\cdot)$ the final prediction for each task can be written as

$$p_\omega(y \mid x) = \frac{\exp(-\|f_\omega(x) - \mathbf{c}_y\|^2)}{\sum_{k=1}^{K} \exp(-\|f_\omega(x) - \mathbf{c}_k\|^2)}$$

($\mathbf{c}_k$ is the centroid of training data)



To train the embedding map, we take an "episode-based approach."

More specifically, we iterate over many episodes, where in each episode we:

(1) Randomly draw a task (or several tasks, if RAM is large) from the support set.

(2) Compute prototypes (i.e., centroids) using the training split data of the task.

(3) Update ω for several SGD steps, with loss computed on validation split data of the task.

Training Cost (for query set tasks). We have zero training cost for the new task!
This is a great benefit, but comes at the cost of zero capacity for adaptation (low test performance).

Inference Cost. Inference cost for CNN + nearest-neighbor cost (essentially a linear classifier).

Meta-Training Cost. Large, as we perform many cycles of mini-trainings.

MAML. The model-agnostic meta-learning algorithm proposed by Finn et al. (2017).

Idea. A key difference from ProtoNet is that MAML allows for a per-task adaptation of embedding maps.
That is, we let ω to be a "learned initialization,"
from which perform small number of SGD to fit the target task (thus $\theta = \omega + \Delta\theta$).

(also, no longer need to do nearest-neighbor; allow the final fully-connected layer to adapt!)

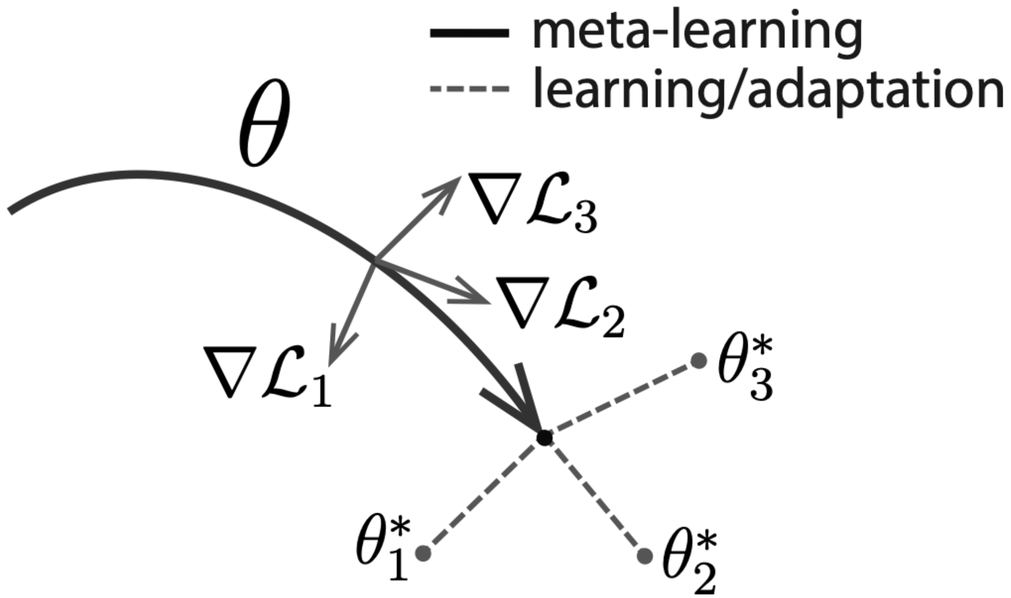


Figure 1. Diagram of our model-agnostic meta-learning algorithm (MAML), which optimizes for a representation θ that can quickly adapt to new tasks.

Finn et al. "Model-agnostic meta-learning for fast adaptation of deep networks" ICML 2017

In a nutshell, we repeat over the “outer loops” that consists of:

(1) Sample a batch of tasks $\mathsf{T}_1, \ldots, \mathsf{T}_k$

(2) For each task, generate an adapted version by running SGD on the training split of each task

$$\theta_i = \omega - \alpha \cdot \nabla_\omega \mathbb{E}_{(X,Y)\sim \mathsf{D}_{\text{train}}} \ell(f_\omega(X), Y)$$

(here, note that θ_i is a function of ω)

(3) Update ω by using a gradient descent, where the gradient is computed based on the training split of each task:

$$\omega \leftarrow \omega - \beta \cdot \nabla_\omega \mathbb{E}_{(X,Y)\sim \mathsf{D}_{\text{val}}} \ell(f_{\theta_i(\omega)}(X), Y)$$

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

Note#1. Keeping the gradient flowing from each θ_i to ω is a great memory burden!
Note#2. We typically do multiple steps of (2), called inner loops, which signifies the memory problem.
Note#3. Similar concept is “pre-training”..... who’s the winner?

Finn et al. “Model-agnostic meta-learning for fast adaptation of deep networks” ICML 2017

MiniImagenet (Ravi & Larochelle, 2017)	5-way Accuracy	
	1-shot	5-shot
fine-tuning baseline	$28.86 \pm 0.54\%$	$49.79 \pm 0.79\%$
nearest neighbor baseline	$41.08 \pm 0.70\%$	$51.04 \pm 0.65\%$
matching nets (Vinyals et al., 2016)	$43.56 \pm 0.84\%$	$55.31 \pm 0.73\%$
meta-learner LSTM (Ravi & Larochelle, 2017)	$43.44 \pm 0.77\%$	$60.60 \pm 0.71\%$
MAML, first order approx. (ours)	$\mathbf{48.07 \pm 1.75\%}$	$\mathbf{63.15 \pm 0.91\%}$
MAML (ours)	$\mathbf{48.70 \pm 1.84\%}$	$\mathbf{63.11 \pm 0.92\%}$

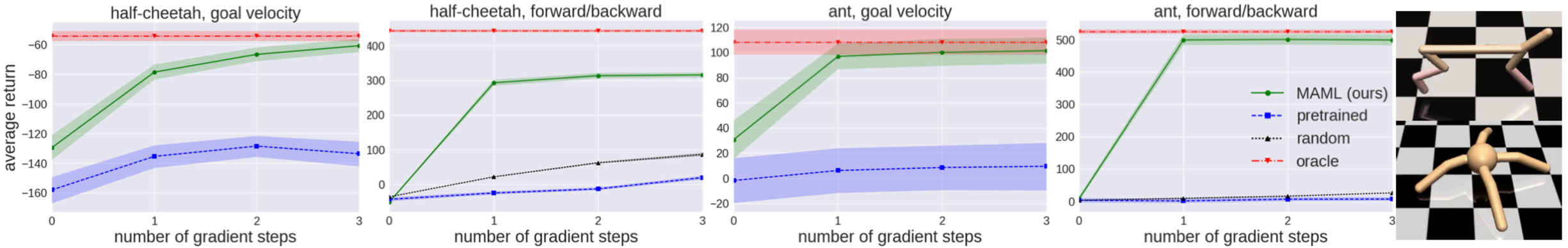


Figure 5. Reinforcement learning results for the half-cheetah and ant locomotion tasks, with the tasks shown on the far right. Each gradient step requires additional samples from the environment, unlike the supervised learning tasks. The results show that MAML can adapt to new goal velocities and directions substantially faster than conventional pretraining or random initialization, achieving good performs in just two or three gradient steps. We exclude the goal velocity, random baseline curves, since the returns are much worse (< -200 for cheetah and < -25 for ant).

Learned Optimizer. Instead of transferring the model parameters, one can learn a nice "optimizer" to replace SGD (Andrychowicz et al., 2016)

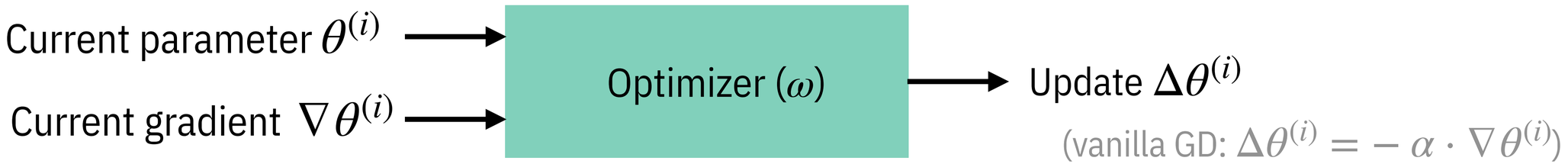
By this time, people have been stunned by the huge success of the Adam optimizer, which is an SGD-ish algorithm with many optimizer hyperparameters.

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Andrychowicz et al., "Learning to learn by gradient descent by gradient descent," NeurIPS 2016

Why don't we parameterize the optimizer itself with a neural network, and train the optimizer over many workloads using the meta-learning approach?



The motivation is twofold:

(1) Maybe we can find a better optimizer than Adam

(2) We may not need to tune the learning rate, momentum, … via extensive search.

Andrychowicz et al., "Learning to learn by gradient descent by gradient descent," NeurIPS 2016

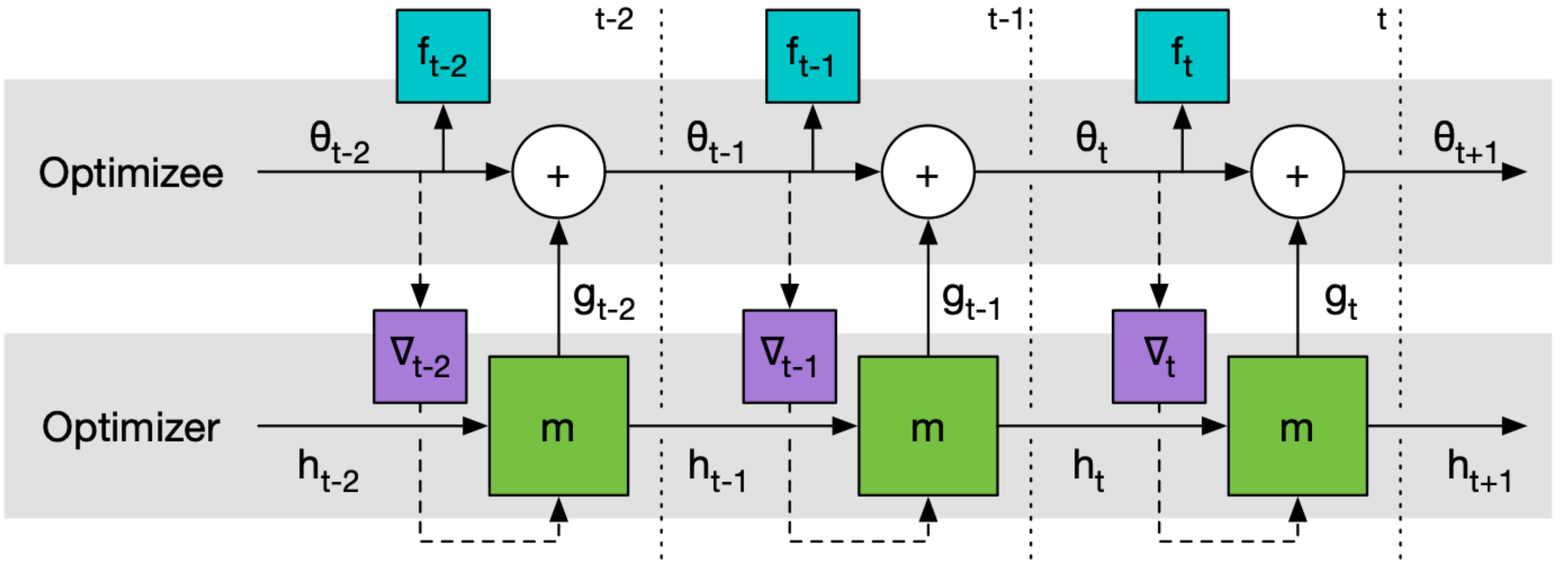


Figure 2: Computational graph used for computing the gradient of the optimizer.

As one may expect, the optimizer is usually implemented as an LSTM-based RNNs.

One problem is that having “all gradients” requires a too big model—
Andrychowicz et al. avoids this problem by training an optimizer per-coordinate (i.e., takes one param as input)

This gives an additional benefit of architecture-agnosticity!

Andrychowicz et al., “Learning to learn by gradient descent by gradient descent,” NeurIPS 2016

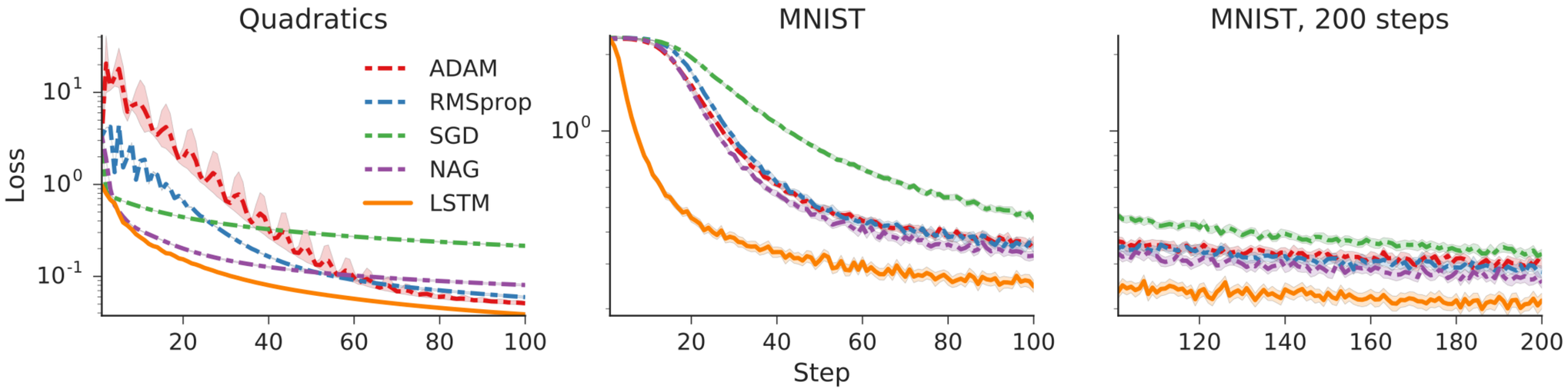


Figure 4: Comparisons between learned and hand-crafted optimizers performance. Learned optimizers are shown with solid lines and hand-crafted optimizers are shown with dashed lines. Units for the y axis in the MNIST plots are logits. **Left:** Performance of different optimizers on randomly sampled 10-dimensional quadratic functions. **Center:** the LSTM optimizer outperforms standard methods training the base network on MNIST. **Right:** Learning curves for steps 100-200 by an optimizer trained to optimize for 100 steps (continuation of center plot).

Andrychowicz et al., "Learning to learn by gradient descent by gradient descent," NeurIPS 2016

Next Up. Efficient MAMLs
Test-time adaptation
VeLO — a learned optimizer arXived on Nov 17, 2022