Meta-Learning

EECE695D: Efficient ML Systems

Spring 2025

Recap

- **Goal.** Efficient Training

- **How?** Use "experience" gained from previous training episodes

- **Last Class.** Continual Learning
    - Multiple tasks, shown sequentially
    - Goal. Preserve knowledge on seen tasks, to perform well on seen tasks
- **Today.** Use it for unseen tasks?

Basic idea

Idea

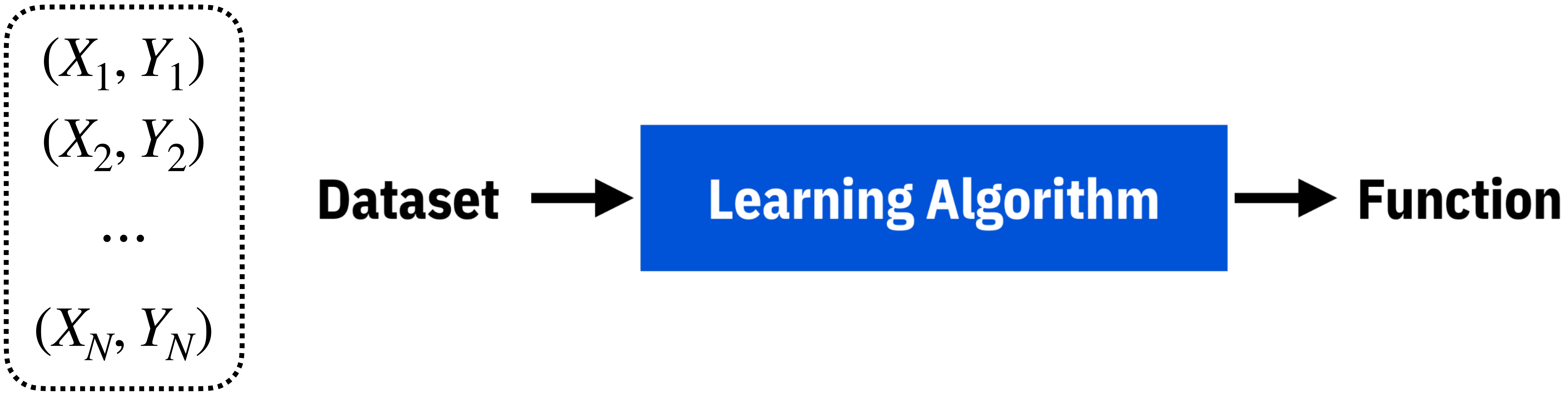
- Gains **experience** over multiple learning episodes
    - Covering a distribution of related tasks

- **Goal.** Improve its performance on **future learning tasks**
    - Has two names
        - "Learning to learn"
        - "Meta-learning"

Hospidales et al., "Meta-Learning in Neural networks: A Survey," IEEE TPAMI 2022

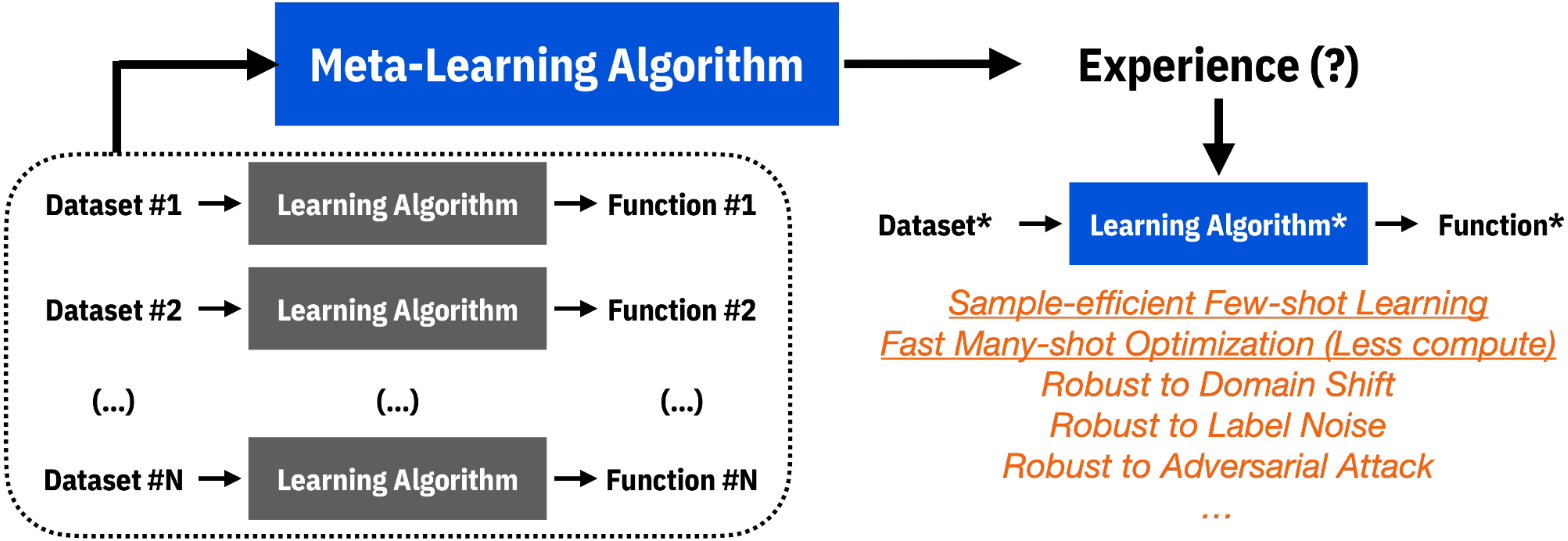
Learning

- **Given.** A dataset drawn from a distribution (i.e., training data)
- **Goal.** Find a **model** (function) that works well on the dataset
  - Should work well on new data drawn from the distribution (i.e., test data)



“Meta”-Learning

- **Given.** A “task” set drawn from a distribution
- **Goal.** Find a **”meta-model”** (experience) that works well on the task set
  - Should work well on new “task” drawn from the distribution



Formalism

- We have a set of tasks drawn from an unknown distribution

$$T_1, \ldots, T_m \sim P_{\text{task}}$$

  - Each task consist of a triplet

$$T_i = (D_i^t, D_i^v, L_i)$$

    - D_i^t, D_i^v: Training (support) / Validation (query) set of task i
    - L_i: Loss function
      - $L_i(\theta, \omega, D_i^v)$ is the loss of model param θ on dataset D_i^v, when we have transferred the **meta-knowledge** ω

Formalism

- **Training.** Fit the model parameter θ on each task:

$$\theta_i^*(\omega) = \arg\min_{\theta} L_i(\theta, \omega, D_i^t)$$

- **Meta-Training.** Minimize the average task-wise losses:

$$\min_{\omega} \sum_{i=1}^{m} L_i(\theta_i^*(\omega), \omega, D_i^v)$$

  - Note. We care about the validation loss, evaluated after per-task fitting

Formalism

- **Question.** Which **meta-knowledge** ω can we transfer?
  - Initial Parameters, Optimizer, Hyperparameters, Black-box Model, Embedding (Metric), Modules, Instance Weights, Exploration Policy, Attention, Architecture, Noise Generator, Curriculum, Dataset, Environment, Loss/Reward, Data Augmentation, (…)

- Today we'll cover the most popular ideas

Hospidales et al., "Meta-Learning in Neural networks: A Survey," IEEE TPAMI 2022

Example task

- As a running example, we consider **few-shot classification**
  - Each task is a k-class classification
    - For each class, we have few samples (e.g., n samples)
    - Classes differ from task to task
  - At (meta-)test, we receive another k-class classification problem with n training samples for each class.

(called k-way, n-shot classification)

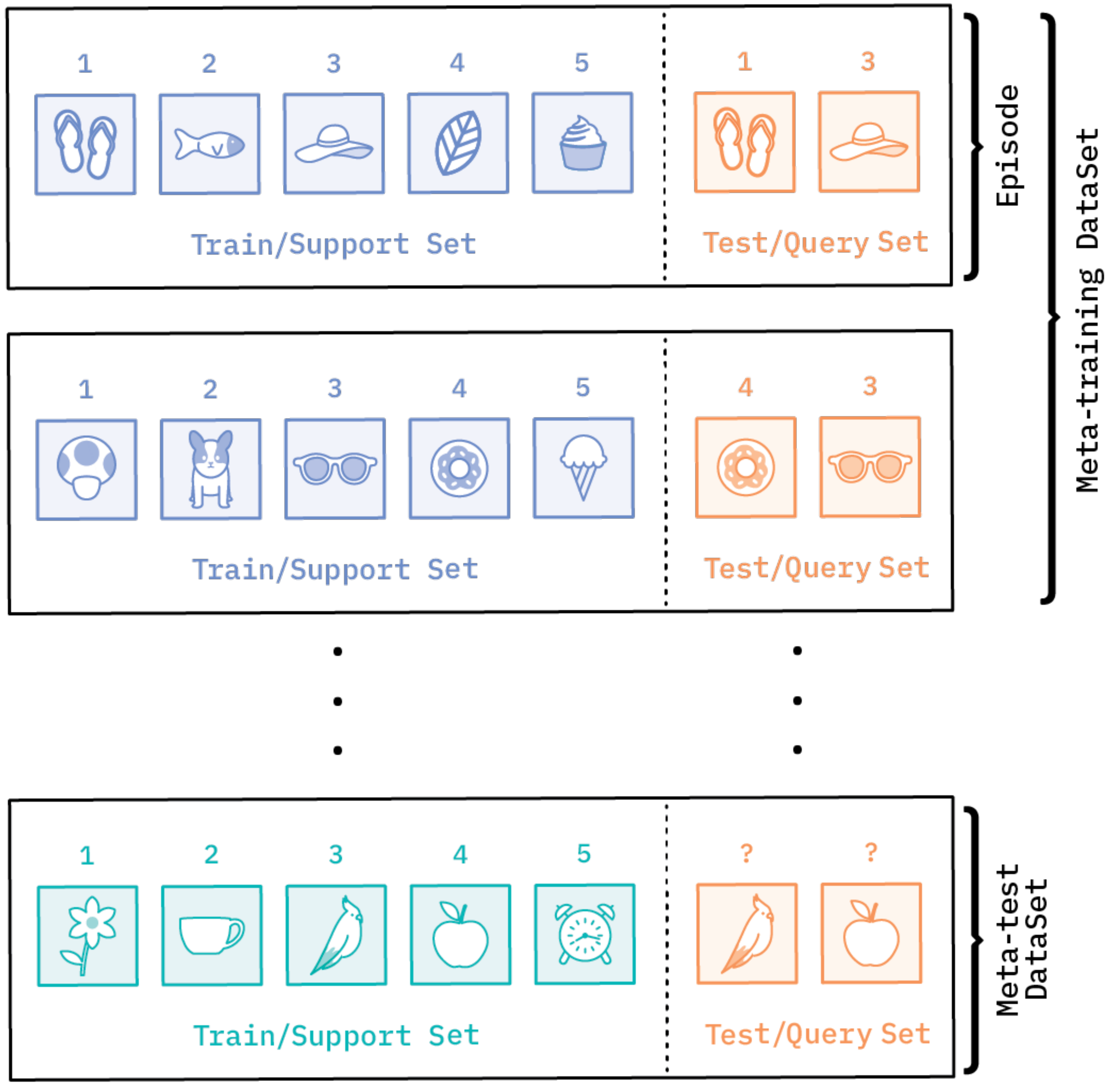
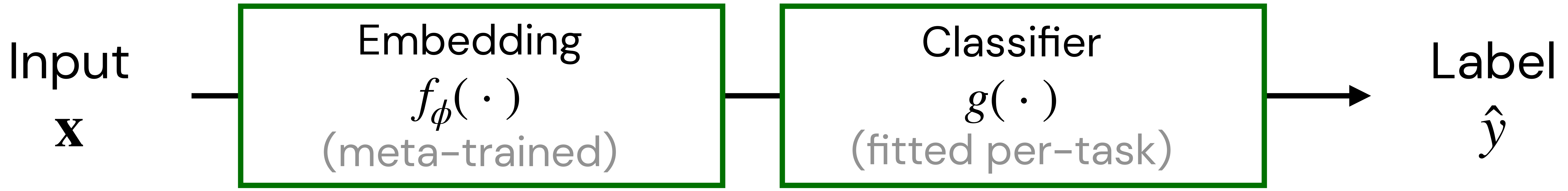


Image source: https://meta-learning.fastforwardlabs.com/

Algorithms

Metric-based: ProtoNet

- **Idea.** Learn a feature-space metric that works well for future tasks
  - That is, train an embedding function $f_\phi(\cdot)$ so that classification based on the latent features $f_\phi(\mathbf{x})$ can be done accurately
    - Meta-knowledge ω. Embedding function $f_\phi(\cdot)$
    - Model parameter θ. Metric-based classifier $g(\cdot)$ (will be explained shortly)



Snell et al., "Prototypical networks for few-shot learning," NeurIPS 2017

Metric-based: ProtoNet

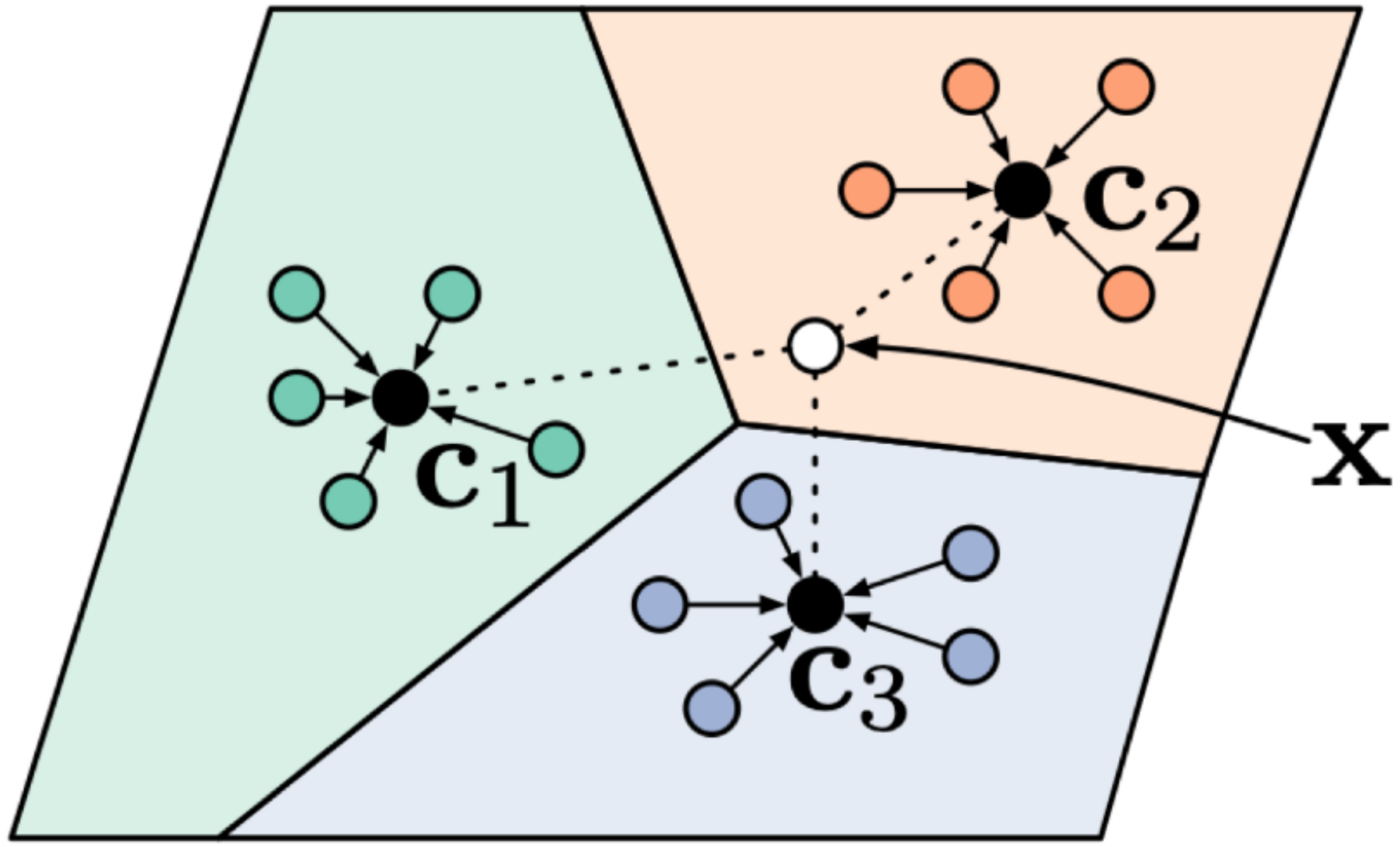
- **Classifier.** Prototype Classifiers
  - Prototype features are defined for each class, as the mean embedding

$$\mathbf{c}_k = \frac{1}{|S_k|} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_\phi(\mathbf{x}_i)$$

  - Perform the softmax classification

$$p_\phi(y = k \mid \mathbf{x}) = \frac{\exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k))}{\sum_{k'} \exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_{k'}))}$$

    - No training needed; not many samples needed



Snell et al., "Prototypical networks for few-shot learning," NeurIPS 2017

Metric-based: ProtoNet

- **Meta-Training.** Find ϕ which minimizes classification loss on each task:
  - i.e.,, average of the per-task losses, where the loss for task j is:

$$\sum_{(\mathbf{x}_i, y_i) \in D_j^v} -\log p_\phi(y = y_i \,|\, \mathbf{x}_i)$$

    - Note. We use validation samples
    - Note. Prototypes $\mathbf{c}_k$ also depend on ϕ

Snell et al., "Prototypical networks for few-shot learning," NeurIPS 2017

Metric-based: ProtoNet

- **Algorithm.** Take an episode-based approach:
  - Iterate over:
    - Randomly draw a task (or tasks, if RAM permits)
    - Compute prototypes with the training split
    - Compute loss on validation split
    - Update features for several SGD steps
      - Gradients through both prototypes & validation samples

Snell et al., “Prototypical networks for few-shot learning,” NeurIPS 2017

Metric-based: ProtoNet

- **Pros.** Zero adaptation cost
- **Cons.**
  - No flexibility
    - Given f_ϕ, we cannot improve much even with many test samples
  - Meta-training cost is large
    - Feature map is usually large
    - Gradients flow through both support & query samples

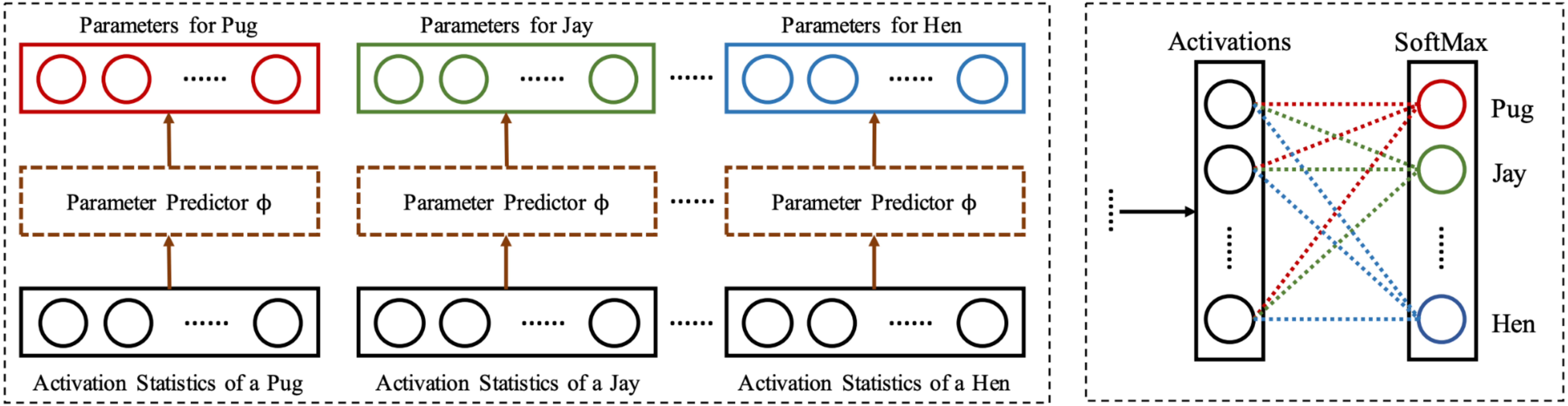
Snell et al., “Prototypical networks for few-shot learning,” NeurIPS 2017

Model-based: Parameter Prediction

- **Idea.** Train a model which predict classifier weights for each class,
  based on the activation statistics of a pre-trained feature map

  - <u>Meta-knowledge ω</u>. Weight prediction model

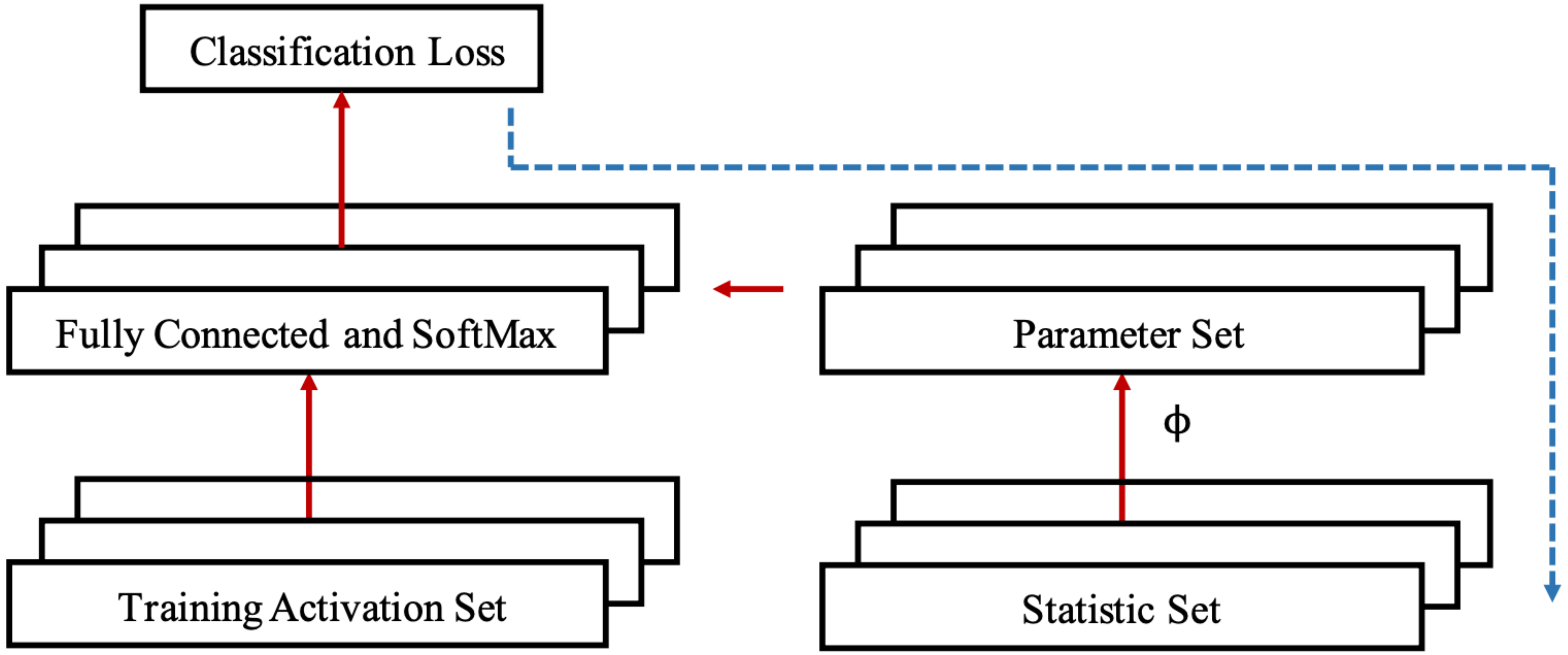
  - <u>Model parameter θ</u>. The predicted weights



Qiao et al., "Few-Shot Image Recognition by Predicting Parameters from Activations," CVPR 2017

Model-based: Parameter Prediction

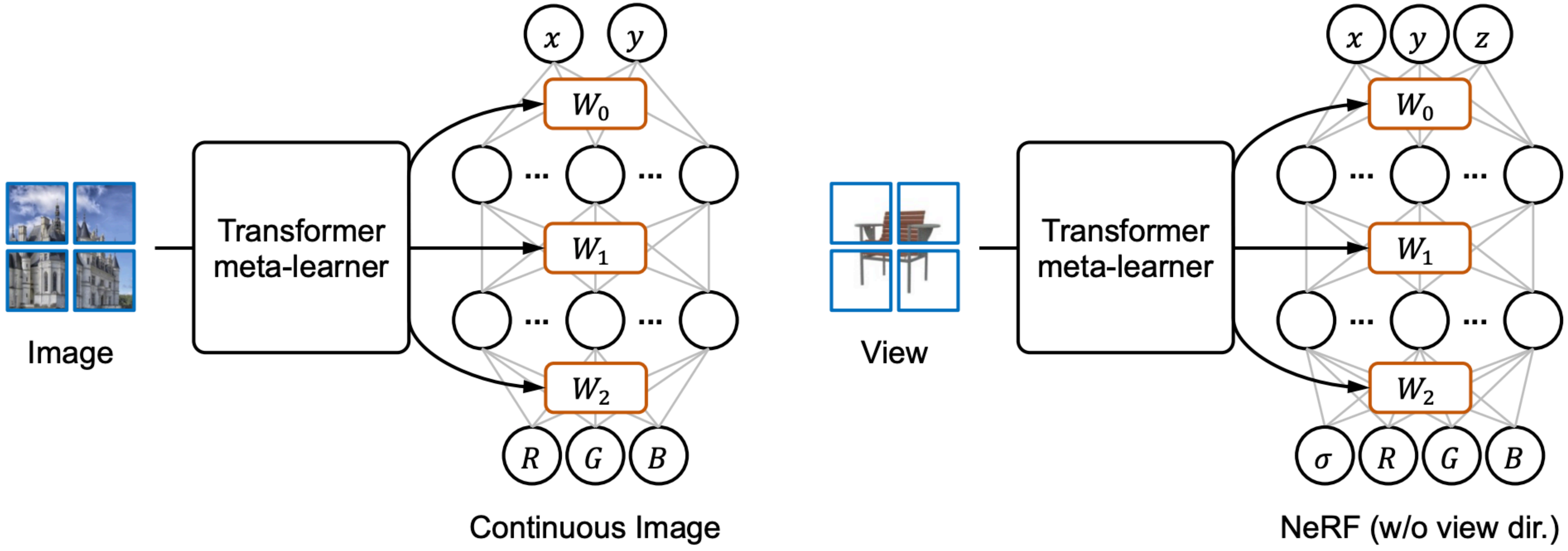
- **Meta-Training.** Similar to ProtoNet, but update the weight predictors not feature maps
  - Gradient on parameter predictors flows through the support samples only
    - Small-scale, and query samples do not affect the parameter predictor



Qiao et al., "Few-Shot Image Recognition by Predicting Parameters from Activations," CVPR 2017

Model-based: Parameter Prediction

- This approach is quite popular in NeRF / 3DGS literature
  - All layer weights are predicted, from the given image/views
    - Sometimes a “modulation” added or multiplied to the base model
    - Requires a very large meta-learner, sometimes



Chen and Wang, “Transformers as Meta-Learners for Implicit Neural Representations,” ECCV 2022

Model-based: Parameter Prediction

- **Pros.** Potentially reduced computational cost
    - Can play with the model size
- **Cons.** Still, suffers from restricted expressive power
    - On unseen data, limited capacity to adapt further

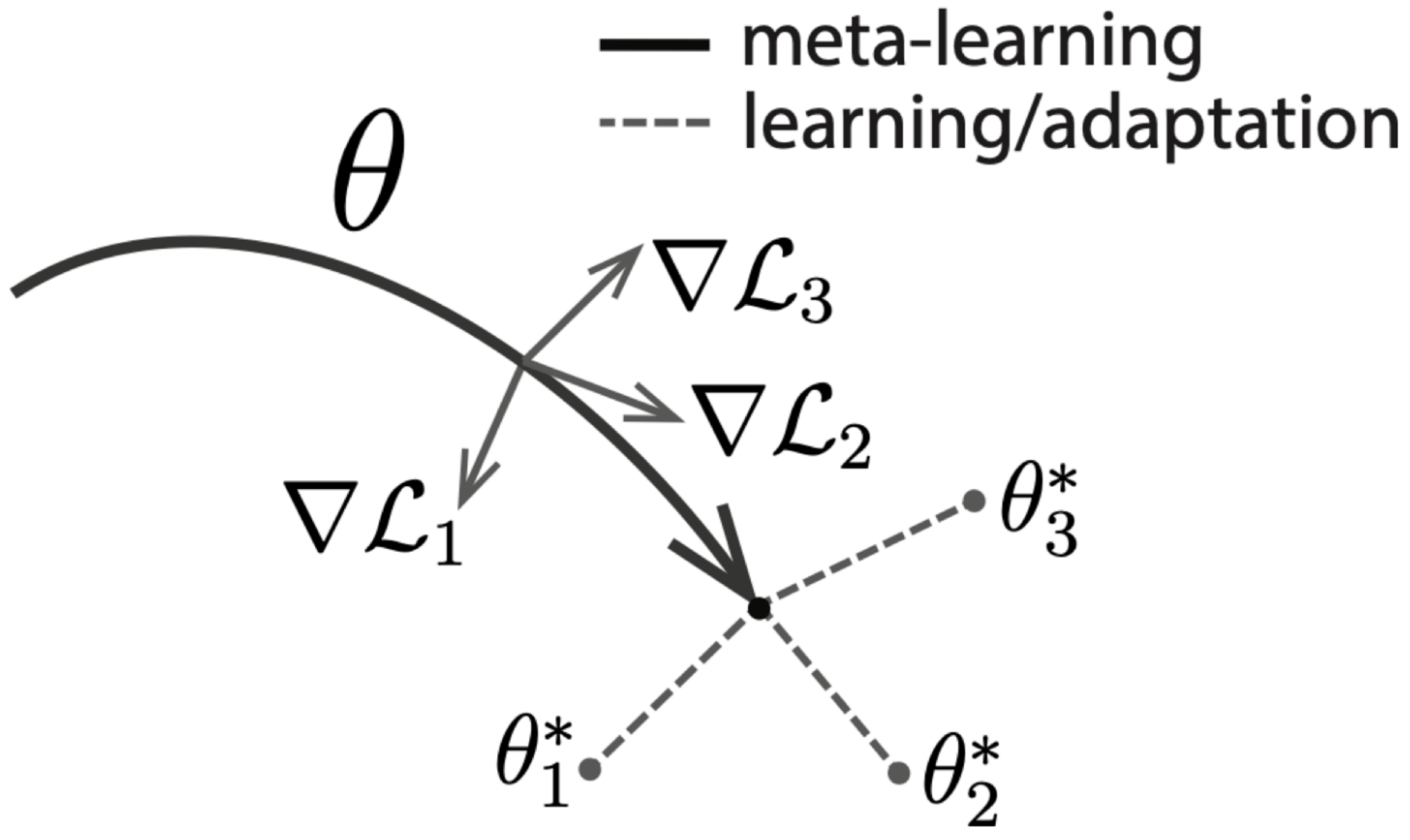
Chen and Wang, “Transformers as Meta-Learners for Implicit Neural Representations,” ECCV 2022

Optimization-based: MAML

- **Idea.** Train a good initialization from which the model can adapt rapidly to each task within a small number of SGD steps

  - Meta-knowledge ω. Initial parameters θ_0

  - Model parameter θ. Model weights $\theta_i = \theta_0 + \Delta\theta_i$



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

Optimization-based: MAML

- **Meta-Training.** We iterate over a double loop:
  - Initialize θ
  - **OUTER LOOP:**
    - Sample a batch of task $1, \ldots, t$
    - **INNER LOOP:** For each $i \in \{1, \ldots, t\}$
      - Generate task-adapted parameters with SGD $\theta_i'(\theta) = \theta - \alpha \nabla_\theta L(\theta, D_i^t)$
    - Update θ (pre-adaption) to minimize val loss $\theta \leftarrow \theta - \beta \nabla_\theta \sum L_i(\theta_i'(\theta), D_i^v)$
  - Return the converged parameter

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

Optimization-based: MAML

- **Pros.** Improved adaptivity — just train further!
- **Cons.** Much memory required (need to track multiple versions of model)
  - Many memory-light variants: iMAML, 1st-order MAML, Reptile
  - Still, long-horizon meta-learning is not satisfactory with these (i.e., many steps in the inner loop)

Algorithm 2 Reptile, batched version

```
Initialize θ
for iteration = 1, 2, ... do
    Sample tasks τ_1, τ_2, ..., τ_n
    for i = 1, 2, ..., n do
        Compute W_i = SGD(L_{τ_i}, θ, k)
    end for
    Update θ ← θ + β (1/n) Σ_{i=1}^{n} (W_i − θ)
end for
```

Nichol et al., "On First-order meta-learning algorithms," arXiv 2018

Learned Optimizers

- **Idea.** Learn an optimizer to replace SGD
  - Motivation. Adam works extremely well
    - Is it optimal?
    - How do we remove the need for hyperparameter tuning?

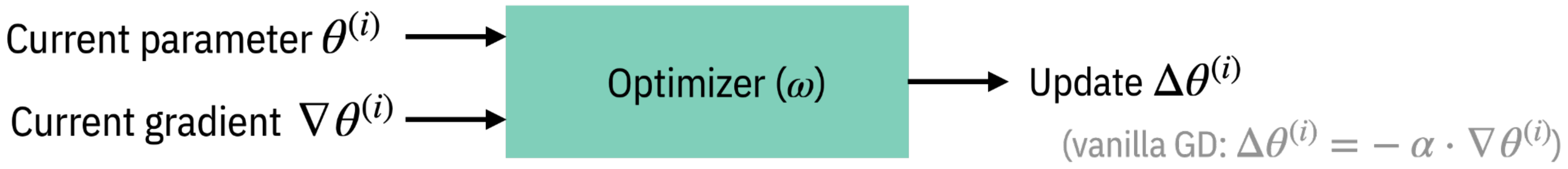
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ 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 1$^{\text{st}}$ moment vector)
  $v_0 \leftarrow 0$ (Initialize 2$^{\text{nd}}$ 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

Learned Optimizers

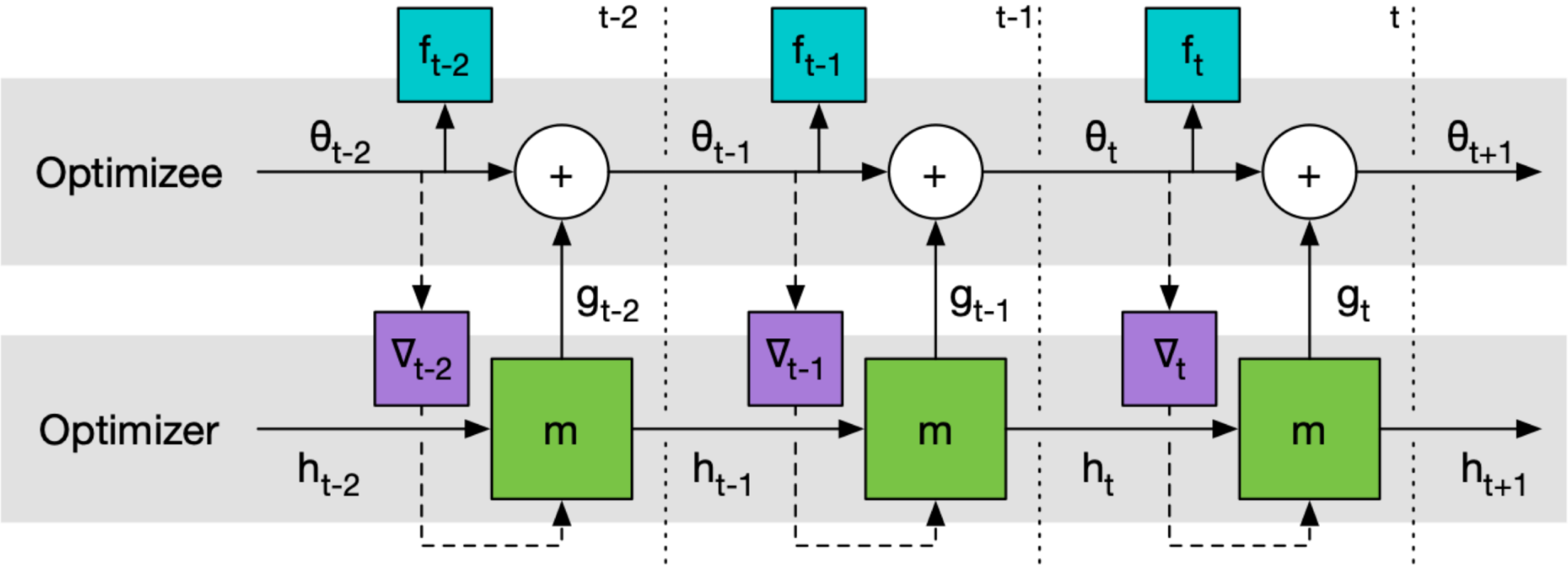
- **Question.** How do we parameterize the optimizer?
- **Answer.** View it as a black box that takes current param & gradient as input, and the actual update as an output
  - Challenge. Need to be able to express the momentum
  - Challenge. Need to be able to optimize various-sized tensors / models



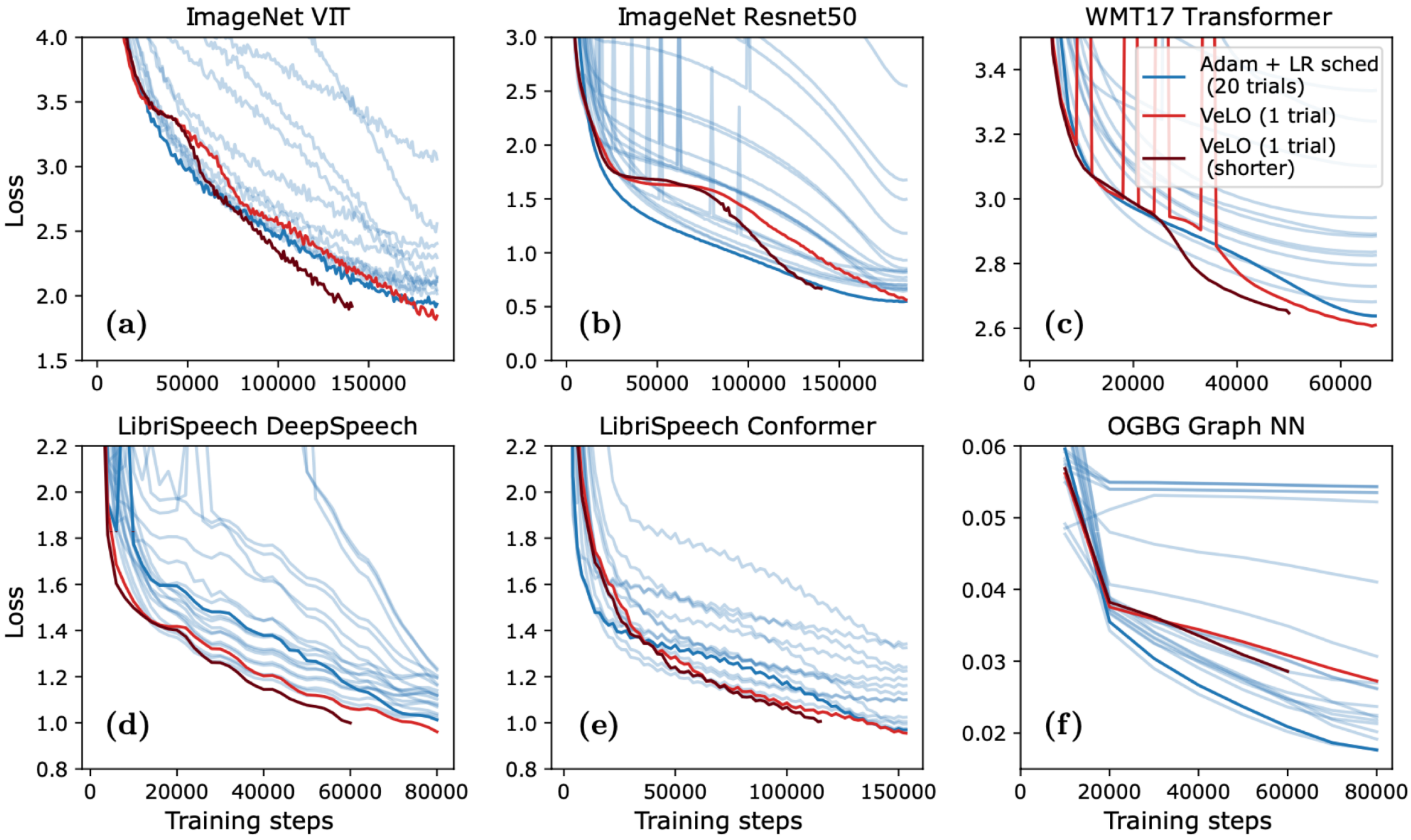
Andrychowicz et al., "Learning to learn by gradient descent by gradient descent," NeurIPS 2016

Learned Optimizers

- We use LSTM-based models
  - **Momentum.** “States” can keep track of past gradients
  - **Tensor size.** Sequential prediction, coordinate-by-coordinate



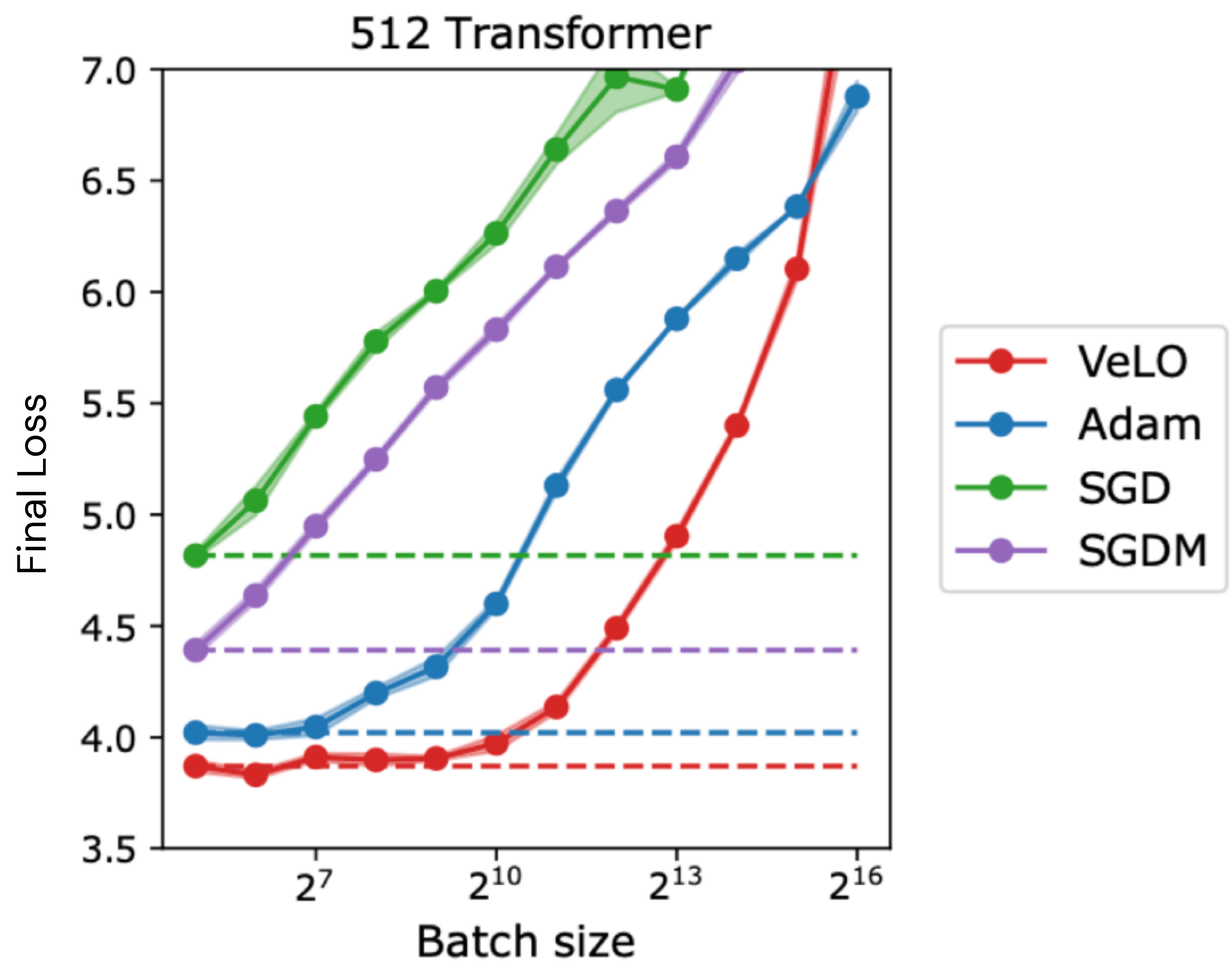
Andrychowicz et al., “Learning to learn by gradient descent by gradient descent,” NeurIPS 2016



Metz et al., "VeLO: Training Versatile Learned Optimizers by Scaling Up," arXiv 2022

Learned Optimizers

- **Pros.**
  - Less need to tune optimizers
  - Can handle larger batch sizes
    - Accelerate training!
- **Cons.**
  - Does not scale up to large models / long training / RL
  - No actual speedup (more compute)



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

Other topics:
Test-time adaptation

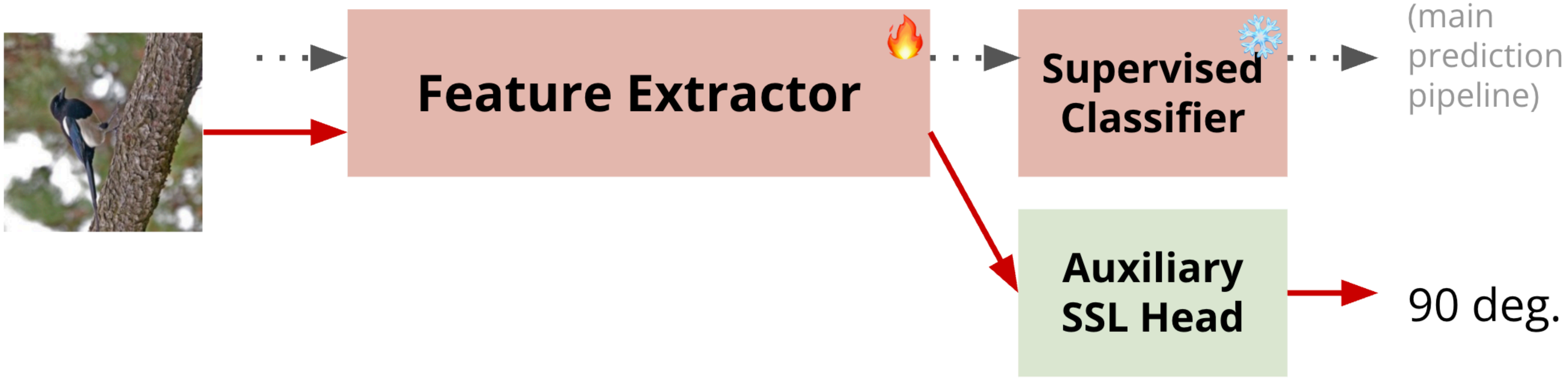
Test-Time Training / Adaptation

- **Idea.** Perform additional adaptation on given task at test time
  - Unlike meta-learning, use only the (a batch of) unlabeled data
  - Roughly two categories:
    - Test-Time Training. Can utilize some source data
    - Fully Test-Time Adaptation. No access to source data

setting	**source data**	**target data**	**train loss**	**test loss**
fine-tuning	-	x^t, y^t	$L(x^t, y^t)$	-
domain adaptation	x^s, y^s	x^t	$L(x^s, y^s) + L(x^s, x^t)$	-
test-time training	x^s, y^s	x^t	$L(x^s, y^s) + L(x^s)$	$L(x^t)$
fully test-time adaptation	-	x^t	-	$L(x^t)$

Test-Time Training / Adaptation

- Example. Test-Time Training (2019)
  - Fine-tune the feature map using a self-supervised learning task
    - Uses rotation-prediction task
    - Needs altering the orig. model to be trained using SL + SSL loss jointly



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

Test-Time Training / Adaptation

- Example. TENT (2021)
  - If we have a good model, maybe our predictor is mostly correct:
  - Thus, reinforce current predictions:
    - Use a batch of data to minimize prediction entropy
    - Tunes only scaling&shifting in BatchNorm layers

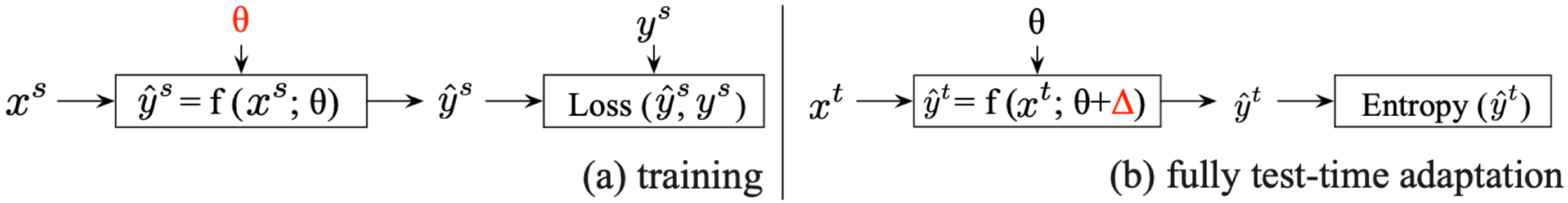


Figure 3: Method overview. Tent does not alter training (a), but minimizes the entropy of predictions during testing (b) over a constrained modulation Δ, given the parameters θ and target data x^t.

Wang et al., "TENT: Fully test-time adaptation by entropy minimization," ICLR 2021

Wrapping up

- Transferring knowledge from a task to task:
  - Continual Learning
  - Meta-Learning
  - Test-time Adaptation

- **Next week.** A bit more on training efficiency

That’s it for today 🙌