Meta-Learning

DSCC/LING 251/451: Machine Learning with Limited Data

C.M. Downey

Spring 2026

Roadmap

- What is meta-learning? The "learning to learn" intuition
- Episodic training: the N-way K-shot protocol
- The meta-learning landscape: metric, model, optimization
- **Deep dive on MAML**: the algorithm, the math, the variants
- MAML vs. Prototypical Networks: when to use which
- Connection back to in-context learning
- Honest assessment: where the field is now

What is Meta-Learning?

The human analogy

A student who has learned French finds it easier to learn Spanish

Not (just) because the languages are similar — that's **transfer**

But because they've learned **how to learn a language**:

- Listen for cognates
- Look for verb conjugation patterns
- Build vocabulary through context
- Expect noun genders, case systems, etc.

Meta-learning tries to capture this: train on a *distribution of tasks* so the model acquires a **learning procedure** that works well on new tasks

The formal framing

Standard learning:

- Given dataset D for task $\mathcal{T}$, learn parameters θ that minimize loss on $\mathcal{T}$

Meta-learning:

- Given a *distribution of tasks* $p(\mathcal{T})$, learn something that enables **fast learning** on any new $\mathcal{T}_{\text{new}} \sim p(\mathcal{T})$

The "something" is an **inductive bias** — meta-learning *learns* the inductive bias rather than hand-designing it (Lecture 3)

What does meta-learning learn?

The "something" differs across methods — each family learns a different kind of inductive bias:

Family	What it learns	Key method
Optimization-based	An **initialization** for fast fine-tuning	MAML
Metric-based	An **embedding space** for comparison	Prototypical Networks
Model-based	A **memory system** for storage/retrieval	MANN

Two levels of learning

	Inner loop (task-level)	Outer loop (meta-level)
Goal	Solve one specific task	Learn to solve tasks quickly
Data	K examples (support set)	Many tasks
Speed	Fast (few gradient steps)	Slow (standard training)
Analogy	Student taking an exam	Designing the curriculum

The outer loop optimizes: *"after the inner loop adapts to a new task, how well does it perform?"*

Meta-learning vs. multi-task learning

- **Multi-task learning**: train one model on all tasks simultaneously
  - Goal: good performance on the *training* tasks
  - "Be good at everything"
- **Meta-learning**: train across tasks for fast adaptation
  - Goal: good performance on *new, unseen* tasks
  - "Be quick at anything new"

The traditional distinction: meta-learning is evaluated on **tasks it has never seen**

But modern instruction tuning blurs this line — FLAN uses standard multi-task training (no episodes) yet holds out entire tasks for evaluation, exactly like meta-learning. The *training mechanism* is multi-task; the *evaluation protocol* is meta-learning.

When does meta-learning make sense?

Three requirements:

1. You face **many related few-shot tasks** — not just one
2. You have a **task distribution** — enough distinct tasks to meta-train on
3. New tasks share structure with training tasks

When it does NOT make sense:

- You have one task with limited data → just use transfer + fine-tuning
- Your tasks are wildly dissimilar → the task distribution assumption breaks

Natural applications

- **Drug discovery**: each new compound is a classification task (active/inactive), but you have thousands of prior compounds
- **Personalization**: each new user is a new task, but you have many existing users
- **Robotics**: each new object to manipulate is a new task
- **Low-resource languages**: each language is a "task," meta-train across high-resource ones

The common pattern: a **natural task distribution** where individual tasks have little data, but the *collection* of tasks is large

Episodic Training

The N-Way K-Shot Protocol

Constructing one episode

An **episode** simulates one few-shot problem during training:

1. Sample N classes from the full class set
2. For each class, sample K examples → **support set** S ($N \times K$ total)
3. For each class, sample additional examples → **query set** Q
4. Give the model S, evaluate on Q
5. Compute loss on Q, update the meta-learner

Constructing one episode

Example meta-learning set-up for few-shot image classification, visual adapted from Ravi & Larochelle '17.

Why episodes?

Standard training:

- Each batch has many examples from all classes
- The model learns to classify the training classes

Episodic training:

- Each batch *simulates the test-time scenario*
- The model learns to learn from few examples

The training distribution matches the test distribution

> *"One-shot learning is much easier if you train the network to do one-shot learning"*
> — The key insight from Vinyals et al. (2016)

The meta-split: classes, not examples

	Standard ML	Meta-learning
What you split	Examples	Classes
Splits share classes?	Yes	**No** — disjoint
Generalize to…	New examples of known categories	**Entirely new categories**

Example — Mini-ImageNet (100 classes total):

Split	Classes	Purpose
Meta-train	64	Used during meta-training episodes
Meta-val	16	Hyperparameter tuning
Meta-test	20	Final evaluation

A subtlety: "new classes" vs. "new tasks"

We defined meta-learning as adapting to **new tasks** from a distribution

But the N-way K-shot protocol tests something narrower: **new classes within one task** (image classification)

- Classifying 5 new bird species is still image classification — the *task* is the same, only the *classes* are new
- Switching from image classification to medical diagnosis is a genuinely new *task*

The benchmark literature often conflates these — keep the distinction in mind as we survey methods

What kind of shift is this? (Lecture 11 callback)

The "new classes" setting is **not** covariate shift — $p(y \mid x)$ isn't staying fixed while $p(x)$ shifts

We're *extending the label space* to y values never seen in training. The labeling function $p(y \mid x)$ is **partially known** (the embedding transfers), but the new class assignments are genuinely new.

This matters for method choice:

- **New classes, same task**: metric-based methods (ProtoNets) excel — the learned embedding captures transferable structure of $p(y \mid x)$
- **Genuinely new tasks**: $p(x)$ and $p(y \mid x)$ can both change entirely — need a more flexible mechanism (MAML)

The Meta-Learning Landscape

Family 1: Metric-based methods

Core idea: learn an embedding space where same-class examples are close, different-class examples are far. Classify by nearest-neighbor.

Prototypical Networks (Snell et al., 2017):

- Compute a **prototype** (mean embedding) for each class from the support set:

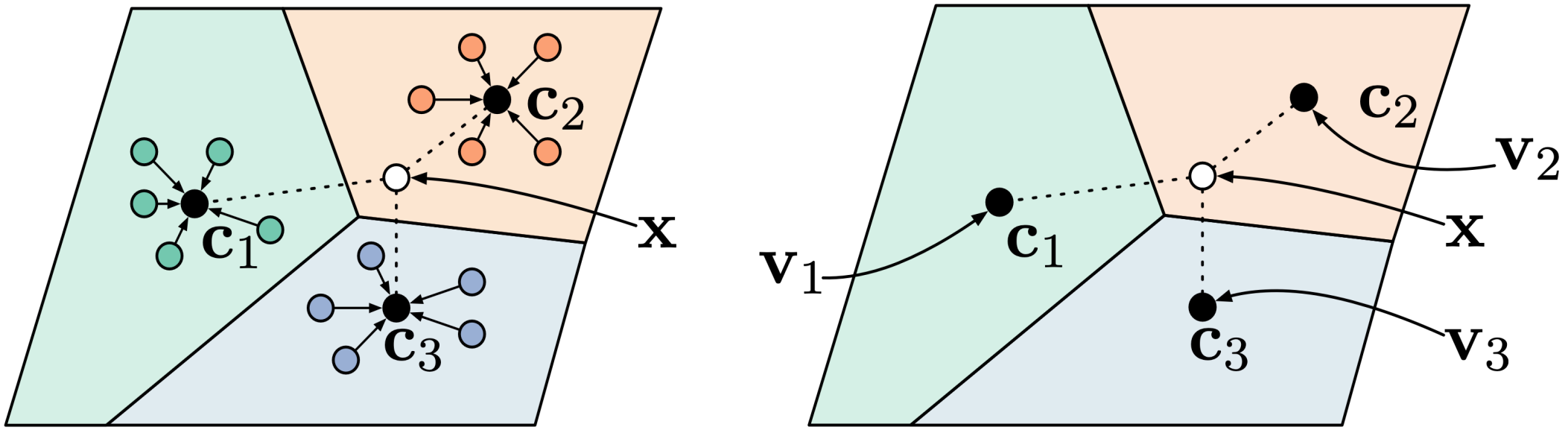
$$c_k = \frac{1}{K} \sum_{i=1}^{K} f_\theta(x_i^{(k)})$$

- Classify query x by distance to prototypes:

$$p(y = k \mid x) \propto \exp\left(-d\left(f_\theta(x),\ c_k\right)\right)$$

- Beautifully simple — and surprisingly competitive

Prototypical Networks



(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: $p_{\boldsymbol{\phi}}(y = k|\mathbf{x}) \propto \exp(-d(f_{\boldsymbol{\phi}}(\mathbf{x}), \mathbf{c}_k))$.

Why simplicity works here

With $K = 5$ examples per class, you can't reliably estimate:

- A class covariance matrix ($d \times d$ parameters)
- A class-specific decision boundary or subnetwork

You *can* reliably estimate:

- A class mean in embedding space (d parameters)

Prototypical Networks lean into this constraint rather than fighting it

Other metric-based methods:

- **Matching Networks** (Vinyals et al., 2016): attention-weighted comparison to all support examples. Introduced episodic training.
- **Relation Networks** (Sung et al., 2018): learn the distance function itself

Prototypical Networks: learned embeddings

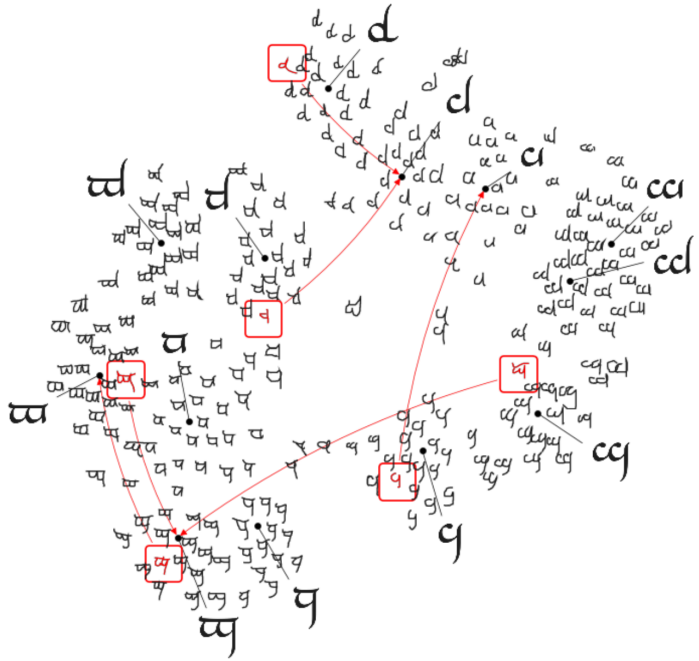


Figure 2: A t-SNE visualization of the embeddings learned by Prototypical networks on the Omniglot dataset. A subset of the Tengwar script is shown (an alphabet in the test set). Class prototypes are indicated in black. Several misclassified characters are highlighted in red along with arrows pointing to the correct prototype.

Matching Networks

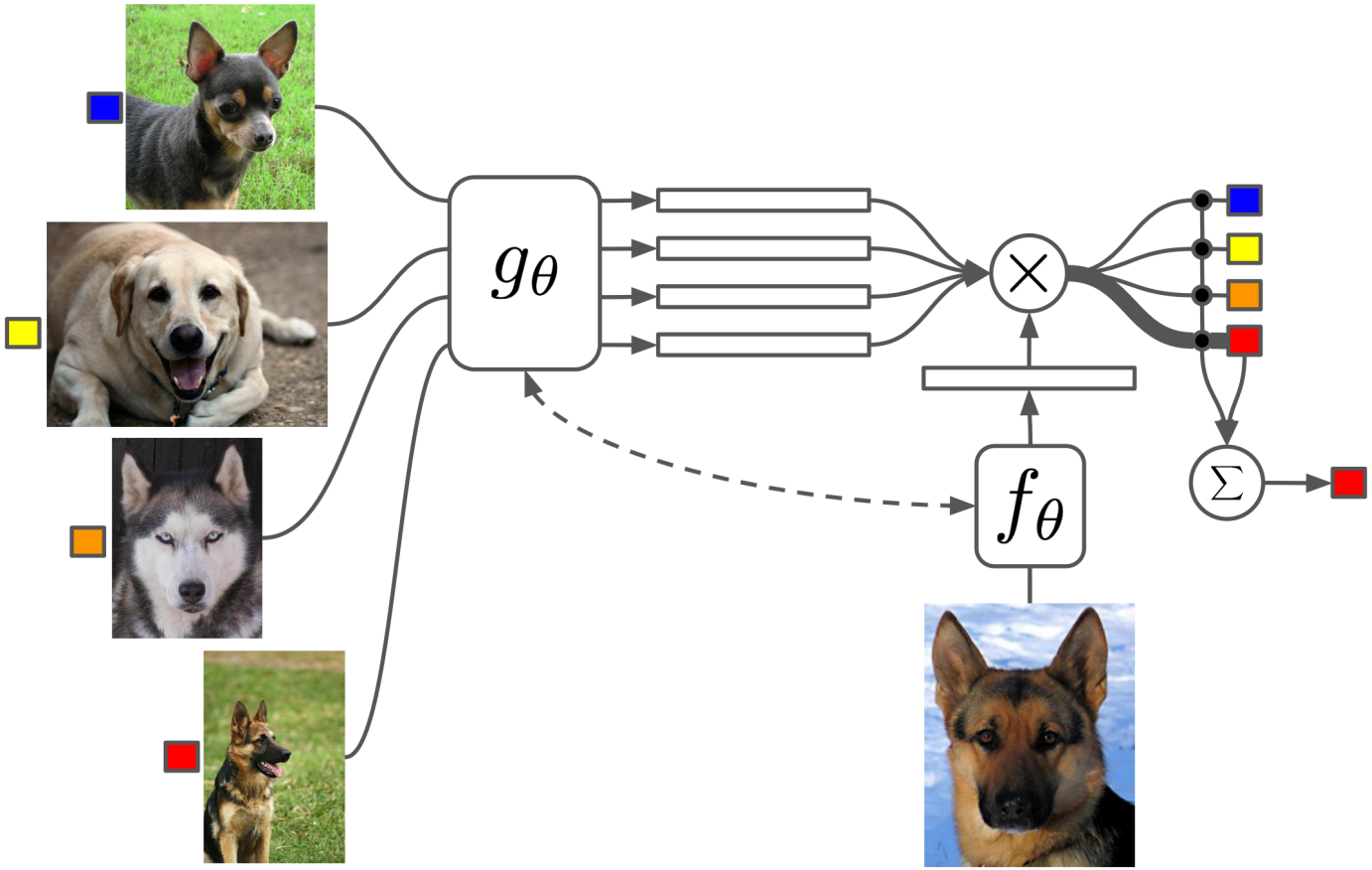


Figure 1: Matching Networks architecture

Family 2: Model-based methods (brief)

Core idea: use an architecture with external memory to store and retrieve support examples

- **MANN** (Santoro et al., 2016): neural net with external memory bank
- **MetaNet** (Munkhdalai & Yu, 2017): generates fast weights from support set

These are less popular now — the architectural overhead is hard to justify when simpler methods work comparably

Family 3: Optimization-based methods

Core idea: learn an optimization procedure that works well with few steps on few examples

Instead of learning a fixed embedding (metric) or a fixed memory system (model), learn how the model **updates itself**

The canonical method: **MAML** (Finn et al., 2017)

Learn an initialization θ such that a few gradient steps on the support set produce good performance on the query set

This is where we'll spend the next several slides

MAML

Model-Agnostic Meta-Learning

Finn et al. (2017) — see also the BAIR blog post

The question MAML asks

Is there an initialization θ of a neural network such that **a few gradient steps** on *any* new task produce good performance?

If yes: find that initialization by training across many tasks

The MAML intuition

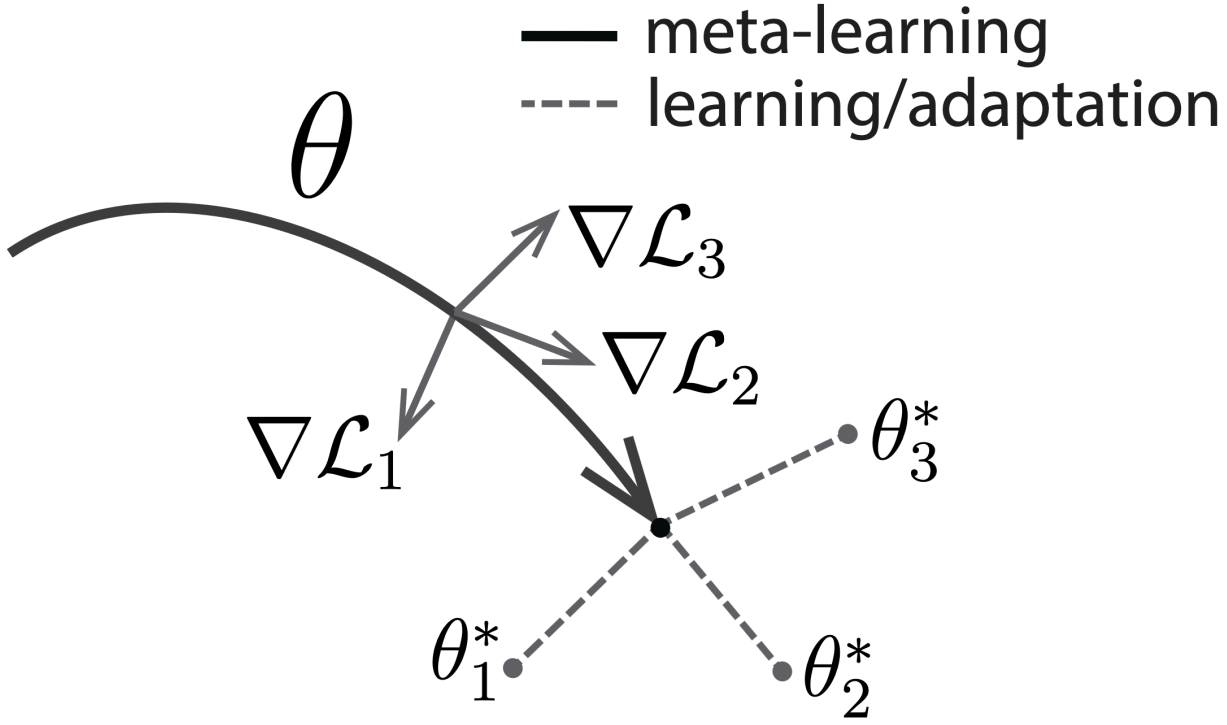


Figure 1. Diagram of our model-agnostic meta-learning algorithm (MAML), which optimizes for a representation θ that can quickly adapt to new tasks.

Why MAML is elegant

Three properties that set it apart:

1. **Model-agnostic**: works with any model trained by gradient descent
    - CNNs, RNNs, transformers — anything differentiable
2. **No extra parameters**: doesn't change the architecture — just finds a better starting point
3. **Composes with existing pipelines**: take your model, use MAML to find θ, then fine-tune normally

The learned initialization *is* the inductive bias:

- A random θ encodes no preference over tasks
- MAML's θ encodes: "I'm close to many task-specific solutions"

The MAML algorithm: inner loop

Setup: model f_θ, task distribution $p(\mathcal{T})$, inner LR α, outer LR β

1. Sample a batch of tasks $\{\mathcal{T}_i\}$ from $p(\mathcal{T})$

2. For each task $\mathcal{T}_i$, sample support set S_i and query set Q_i

3. Inner loop — adapt to each task's support set:

$$\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{S_i}(f_\theta)$$

This is the same thing that happens at test time: take a few gradient steps on K examples

The MAML algorithm: outer loop

4. Evaluate each adapted model $f_{\theta'_i}$ on its query set Q_i

5. Outer loop — update θ to minimize query losses across all tasks:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_i \mathcal{L}_{Q_i}(f_{\theta'_i})$$

The outer loop asks: *"starting from θ, after the inner loop adapts, how good is the result?"*

It optimizes the **starting point** so that fast adaptation works well on any task

The MAML algorithm

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**

MAML for supervised learning

Algorithm 2 MAML for Few-Shot Supervised 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: Sample K datapoints $\mathcal{D} = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation (2) or (3)
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample datapoints $\mathcal{D}_i' = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$ for the meta-update
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 2 or 3
11: **end while**

The critical subtlety

The outer gradient is a **gradient through a gradient**:

$$\nabla_\theta \, \mathcal{L}_{Q_i} \left(f_{\, \theta \, - \, \alpha \, \nabla_\theta \mathcal{L}_{S_i}(f_\theta)} \right)$$

- θ_i' depends on θ through the inner gradient step
- The outer gradient must **backpropagate through the inner optimization**
- This requires second-order derivatives (Hessian-vector products)

Why the second-order term matters

Without it (pretend θ_i' doesn't depend on θ):

- You're just asking: "what initialization gets low loss *after* fine-tuning?"
- You ignore *how* the initialization affects the fine-tuning trajectory

With it (full MAML):

- You're asking: "what initialization leads to the best *adaptation*?"
- The outer gradient accounts for the curvature of the inner optimization
- It rewards initializations that are in "easy-to-adapt-from" regions of the loss landscape

The second-order term is what makes MAML actually optimize for **fast adaptation**, not just **proximity to solutions**

FOMAML: the first-order shortcut

The problem: second-order gradients are expensive (~2× compute and memory)

FOMAML (First-Order MAML):

- Simply ignore the second-order term
- Treat θ_i' as if it doesn't depend on θ through the gradient
- Outer update: $\theta \leftarrow \theta - \beta \nabla_{\theta_i'} \sum_i \mathcal{L}_{Q_i}(f_{\theta_i'})$

Surprisingly, it works almost as well — on Mini-ImageNet, FOMAML and full MAML are essentially identical; on Omniglot, full MAML has a small edge depending on N and K (Nichol et al., 2018)

Reptile: even simpler

Reptile (Nichol et al., 2018):

1. Sample a batch of tasks $\{\mathcal{T}_i\}$
2. For each task, take several gradient steps: $\theta \rightarrow \theta_i'$
3. Move θ toward the average result: $\theta \leftarrow \theta + \epsilon \frac{1}{n} \sum_i (\theta_i' - \theta)$

No second-order gradients. No support/query split needed. Works comparably to FOMAML.

The MAML family

Method	Second-order?	Support/query split?	Performance
Full MAML	Yes	Yes	Best (small edge on some benchmarks)
FOMAML	No	Yes	Nearly identical to MAML
Reptile	No	No	Comparable to FOMAML

Practical advice: start with FOMAML or Reptile — the full second-order version is rarely worth the cost

The `learn2learn` PyTorch library provides implementations of MAML, ProtoNets, and standard benchmarks

MAML in practice

What makes it work well:

- Tasks are related enough that a single good initialization exists
- Small models or few inner steps
- Inner learning rate α is tuned carefully (or learned jointly)

What makes it struggle:

- **Computational cost**: even FOMAML requires simulating the inner loop
- **Training instability**: bi-level optimization is finicky
- **Task diversity**: too-diverse tasks → no single good initialization
- **Scaling**: designed for small models (4-layer CNNs); expensive for large ones

What does MAML actually learn?

Raghu et al. (2020) asked: does MAML do "rapid learning" (significantly changing all layers) or "feature reuse" (mostly reusing body features, only changing the head)?

Finding: MAML's inner loop mostly reuses features from the body and only significantly changes the **last layer**

This motivated **ANIL** ("Almost No Inner Loop"):

- Only update the final layer in the inner loop; freeze the body
- Much cheaper, works comparably

Implication: the meta-learned initialization provides a good *feature extractor*, and adaptation is mostly about learning a new *classifier head* on top of it

Sound familiar? This is very similar to the transfer learning picture from Lecture 10.

Rapid learning vs. feature reuse

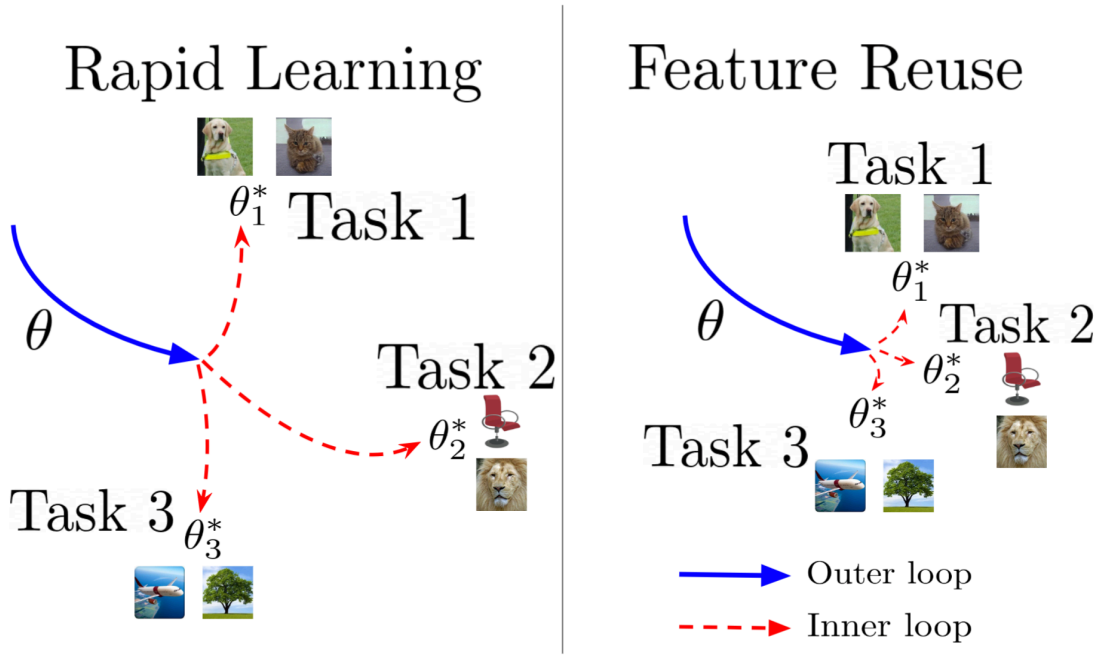


Figure 1: **Rapid learning and feature reuse paradigms.** In Rapid Learning, outer loop training leads to a parameter setting that is well-conditioned for fast learning, and inner loop updates result in significant task specialization. In Feature Reuse, the outer loop leads to parameter values corresponding to reusable features, from which the parameters do not move significantly in the inner loop.

Choosing a Method

MAML vs. Prototypical Networks

	MAML	Prototypical Networks
Learns	An initialization	An embedding space
Adaptation	Gradient descent (inner loop)	Nearest-neighbor to prototypes
Cost at test time	Higher (gradient steps needed)	Lower (just a forward pass)
Model-agnostic?	Yes	No (needs meaningful embeddings)
Scales to large models?	Poorly	Better
When it wins	Complex tasks, diverse structures	Standard classification

In practice: ProtoNets or their variants are often the first thing to try. MAML is more interesting as a *framework* — it reveals what meta-learning is actually doing.

Other notable variants (pointers)

MAML extensions:

- **Meta-SGD** (Li et al., 2017): also learn per-parameter learning rates
- **ANIL** (Raghu et al., 2020): only adapt the head — much cheaper
- **LEO** (Rusu et al., 2019): MAML in a low-dimensional latent space

Hybrid approaches:

- **MetaOptNet** (Lee et al., 2019): differentiable SVM as the inner loop's classifier
- **CNAPS** (Requeima et al., 2019): task-conditioned feature extractors

Meta-learning beyond classification

MAML was originally demonstrated on **regression** and **reinforcement learning** too — it's truly model-agnostic

Active application areas:

- **Drug discovery**: each molecule property is a task — natural few-shot structure
- **Robotics**: each new object or environment is a task requiring fast adaptation
- **NLP**: adapting to new tasks/languages, though fine-tuning + PEFT has largely overtaken dedicated meta-learning here

Meta-Learning and ICL

Connecting the Threads

The structural parallel

	Meta-Learning	In-Context Learning
Meta-training	Train across many episodes	Pre-train on diverse text
Support set	K labeled examples	K demonstrations in the prompt
Adaptation	Gradient steps (MAML) or distance (ProtoNets)	Forward pass through transformer
Query	Evaluate on new examples	Generate output for new input

The hypothesis from Lecture 12:

- If attention layers implement implicit gradient descent...
- ...then pre-training on diverse text is **meta-training** across tasks
- ...and ICL is the **meta-test** phase

ICL may be "MAML where the inner loop is implicit in the architecture"

The key differences

	Meta-Learning	ICL
Task distribution	Designed explicitly	Inherited from pre-training
Modality	Any	Sequential (text, code)
What adapts	Model *parameters*	Model *activations*
Control	You design the tasks	You get what pre-training contained

When to use which:

- Non-text data + natural task distribution → **meta-learning**
- Text tasks + access to a large LM → **ICL might suffice**

Honest Assessment

The baseline problem

Chen et al. (2019), "A Closer Look at Few-Shot Classification":

- A well-tuned baseline — **pre-train, then fine-tune the last layer** — is competitive with meta-learning on standard benchmarks
- The gap is often within error bars

This does NOT mean meta-learning is useless — it means the standard benchmarks are saturated

When meta-learning still wins

Meta-learning outperforms the "pre-train and fine-tune" baseline when:

- Tasks require **structural adaptation**, not just feature reuse
- You have a **genuine task distribution** (not just subsampled ImageNet classes)
- You're in **non-text modalities** where ICL isn't available

The ideas — episodic training, learned initialization, fast adaptation — remain powerful. The easy benchmarks just aren't the right test for them.

Do you even need episodes?

Recall the FLAN observation from earlier: standard multi-task training + held-out task evaluation — no episodes, no inner/outer loop — and it generalizes to new tasks

Two challenges to the meta-learning narrative, from different angles:

Challenge	Source	Implication
Simple baselines match meta-learning	Chen et al. (2019)	Good features may be enough
Multi-task training generalizes to new tasks	FLAN (Wei et al., 2022)	Episodic training may not be necessary

Both suggest that **scale + task diversity** can substitute for the meta-learning training protocol

Episodic training likely matters most when scale is limited and the task distribution is narrow — exactly the **low-resource regime** this course is about

Where meta-learning fits in the course

- **Lecture 3** (Inductive bias): meta-learning *learns* the inductive bias
- **Lectures 5-7** (Self/unsupervised): these provide features that meta-learning builds on
- **Lecture 10** (Transfer): meta-learning is multi-source transfer optimized for adaptation speed
- **Lecture 12** (Few-shot): meta-learning is the purpose-built mechanism; ICL is the emergent one
- **Next time**: data augmentation and synthetic data — another way to fight data scarcity

For Tuesday's discussion

When reading meta-learning papers:

- Which **family** does the method belong to? (metric, model, optimization)
- Does it assume a standard N-way K-shot setup, or something more realistic?
- How does it compare to the **"pre-train then fine-tune" baseline**?
- Is the task distribution **natural** or **manufactured**?

For your term projects:

- Do you have a genuine task distribution?
- Would meta-learning help, or is transfer + fine-tuning sufficient?

References

- Finn et al. (2017), *Model-Agnostic Meta-Learning (MAML)* (ICML)
- Snell et al. (2017), *Prototypical Networks for Few-Shot Learning* (NeurIPS)
- Vinyals et al. (2016), *Matching Networks for One Shot Learning* (NeurIPS)
- Nichol et al. (2018), *On First-Order Meta-Learning Algorithms (Reptile)*
- Raghu et al. (2020), *Rapid Learning or Feature Reuse? (ANIL)* (ICLR)
- Chen et al. (2019), *A Closer Look at Few-Shot Classification* (ICLR)

References (cont.)

- Li et al. (2017), *Meta-SGD*
- Sung et al. (2018), *Relation Networks* (CVPR)
- Lee et al. (2019), *MetaOptNet* (CVPR)
- Rusu et al. (2019), *LEO: Latent Embedding Optimization* (ICLR)
- Von Oswald et al. (2023), *Transformers Learn In-Context by Gradient Descent* (ICML)

Survey:

- Hospedales et al. (2022), *Meta-Learning in Neural Networks: A Survey* (IEEE TPAMI)

Blog:

- Lilian Weng (2018), *Meta-Learning: Learning to Learn Fast*