Meta Learning

‘Learning to Learn’

Normal Machine Learning

- In normal ML, we train a model on a dataset (ex., MNIST digits).
- The model learns patterns (shapes of “0”, “1”, ... “9”).
- At test time, it works well on new images of the **same classes** it trained on.
- But if we give it new classes (say letters A, B, C), the model will fail, because it never learned how to handle new tasks.

Normal ML = learn one task well

In **real-world** situations, we need models that adapt quickly to new tasks with **very little data.**
This is where **meta-learning** comes in.

What is Meta-Learning?

- Instead of learning only one task, the model is trained on **many small tasks**.
- The goal: learn the *process of learning itself*.

Meta-learning = learning to learn.

***(Later for presentation)**

Meta-learning is about training a model to *learn how to learn*, rather than just memorizing specific classes.
When the model encounters new classes or tasks it has never seen before, it can adapt quickly using only a few examples — thanks to the experience it gained from solving many different tasks during training.
This mimics human intelligence: just like we don’t need to see millions of examples to recognize a new object, a meta-learning model can generalize and learn efficiently from limited data.

Why don't we use Transfer learning?

- Transfer learning: pre-train on a big dataset → fine-tune on a new one.
- Problem: fine-tuning often still needs thousands of samples.

While Meta-learning,

- Adapt to a new task with very few samples (1–5 samples per class).
- Save time (no long retraining).
- Save data collection and labelling costs.
- mimics human intelligence

Quick confusion breakdown with other approaches of learning

Batch Learning: Train on the entire dataset offline; entire retraining is needed when new data comes.

Online Learning: Continuously update the model as new data arrives (streaming).

Incremental Learning: Gradually add new tasks/data while preserving old knowledge.

Continual Learning: Broad setting where the model learns tasks sequentially without forgetting (includes incremental).

Transfer Learning: Reuse knowledge from one domain/task to improve performance in another.

Meta-Learning: Learn the *process of learning*, enabling fast adaptation to new tasks with few examples.

Multitask Learning: Train one model to handle multiple tasks simultaneously by sharing representations.

Zero-shot Learning: Solve new tasks/classes without training examples, often via semantic info (e.g., text embeddings).

- Normal ML: Train one model on one dataset with one set of classes.
- Meta-Learning: We simulate tasks during training.
    - Each task = a mini learning problem.
    - This mimics the real-world situation of facing new tasks with very few examples.

N-way K-shot task

- N-way → Number of classes in the task (ex., 5 classes).
- K-shot → Number of examples per class available for learning (ex., 1 or 5).
- Support set → The K examples per class (used for "learning").
- Query set → Additional examples from those classes (used for evaluation).

Training in meta-learning = repeatedly sample many of these tasks (episodes).

Categories of Meta-Learning Approaches

Meta-learning methods fall into **3 main families(approaches)**:

1. Metric-based (learn a good embedding space)
    - Ex: Prototypical Networks, Matching Networks, Siamese Networks.
    - Idea: Learn embeddings → compare new examples using distances.
2. Optimization-based (learn fast adaptation rules)
    - Ex: MAML, Reptile.
    - Idea: Train initial parameters so that only a few gradient steps are needed to adapt.
3. Model-based (meta-learner as an RNN/Memory module)
    - Ex: Neural Turing Machines, LSTM meta-learners.
    - Idea: Architecture itself is designed to adapt quickly.

🔴 Prototypical Networks (Metric-based Meta-Learning)

Goal: Represent each class with a "prototype" vector in an embedding space, and classify new samples based on their distance to these prototypes.

How it works (step by step):

1. Embedding Network
   - A CNN converts each input image into a vector (its embedding).
2. Prototype Creation
   - For each class, take the average of the embeddings of its support examples.
   - This average is the class's prototype.
3. Classification
   - For a new query image, compute its embedding.
   - Compare it to all class prototypes (usually using Euclidean distance).
   - Predict the class of the closest prototype.
4. Training
   - Use cross-entropy loss based on distances.
   - Train over many "episodes" (mini tasks) so embeddings naturally cluster by class.

Why does it work?

- Because the model doesn't memorize specific classes, it learns an embedding space where any new class can be represented by its prototype, even if it was unseen during training.

🔴 MAML (Model-Agnostic Meta-Learning, Optimization-based)

Goal: Train a model so it can adapt to new tasks with just a few gradient updates.

How it works (step by step):

1. Initialization
    - Start with shared model parameters (θ).
2. Task Adaptation (Inner Loop)
    - For each sampled task:
        - Copy θ.
        - Update the copy using the task's support examples (a few gradient steps).
        - This produces adapted parameters (θ').
3. Meta-Update (Outer Loop)
    - Evaluate θ' on the query set of that task.
    - Adjust the original θ so that, after adaptation, it performs well across many tasks.

Why does it work?

- The model is optimized not for one dataset, but to be a good starting point for learning *any* new task with minimal data.

✨ In short:

- Prototypical Networks → Learn a smart distance metric in embedding space.
- MAML → Learn model parameters that can adapt quickly with just a few training steps.

  Ex., prototypes = "average face" for recognizing family members
  MAML = "good study habits to quickly learn any subject"

Evaluation (Few-Shot Testing)

- After training, test on completely unseen classes.
- Example: train on Omniglot characters A–M, test on characters N–Z.
- Give only 1–5 support examples per class, and see if the model classifies new queries correctly.
- Metric: few-shot classification accuracy.

Technical Details:

Episode Generation (N-way K-shot)

- Instead of standard train/test split → create *episodes*:
    - Randomly pick **N classes** (e.g., 5).
    - From each class, pick **K support samples** (e.g., 1 or 5).
    - Also pick some **query samples** to test.
- Each episode is a mini classification task (like "classify between 5 unseen characters given 1 example each").

Model Architecture (Embedding Network)

A simple CNN works well:

- 4 convolutional blocks:
    - Conv (64 filters, 3×3) → BatchNorm → ReLU → MaxPool
- Flatten to a vector (embedding dimension ~64 or 128).
- Output = embedding .

Prototypical Networks Mechanism

1. Pass all support set images through CNN → get embeddings.
2. For each class, average embeddings = **prototype**.
3. For each query embedding:
    - Compute distance (Euclidean) to each prototype.
    - Softmax over negative distances = probability distribution.
4. Loss = cross-entropy comparing prediction to true class.

🛑 Important Insights:

In our encoder, we kept the **same number of filters (hidden_size = 64)** across all layers. Let's unpack why this choice is common in **few-shot learning / prototypical networks**:

Usual CNN design vs. ProtoNet design

- **In normal CNNs (e.g., ResNet, VGG):**
  - We often **increase the number of filters** as we go deeper (e.g., 32 → 64 → 128 → 256).
  - This makes sense for large datasets (ImageNet) because deeper layers need more channels to capture high-level patterns.
- **In Prototypical Networks for Omniglot / Mini-ImageNet:**
  - We usually keep **filters constant (e.g., 64)** across all conv layers.
  - Reason: we don't need extremely deep/complex features — Omniglot images are **small (28×28)** and **simple (strokes, shapes)**.
  - A consistent filter size is:
    - **Lightweight** → trains faster, less overfitting.
    - **Balanced** → ensures all layers produce embeddings in the same scale.
    - **Enough capacity** → 64 filters are already sufficient to learn discriminative features.

✨ In short:

We use the **same number of filters (64)** because Omniglot is small/simple, and meta-learning benefits from a lightweight encoder that generalizes well instead of memorizing.