Chapter 13 – Meta Learning

Author: Gianmarco Scarano

gianmarcoscarano@gmail.com

1. Introduction

In Meta Learning, we want machine learning models to mimic the way the human learns things (in a quick way substantially).

We expect, then, a good meta-learning model to be able of adapt and generalize to new tasks (*not data!*) that have never been encountered during training time. This adaptation is called mini-learning session.

One might think that Meta Learning is actually Fine Tuning, but it's not. Fine Tuning means that when we have a pre-trained model and we want to adapt it to a new task, we need training data for running SGD and tune our model to this new training data.

In Meta Learning we do not modify the weights of the model we have learned, because they are fixed. We'll see later the proper difference.

A quick overview of Meta Learning is enclosed within this formula:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{\mathcal{D} \sim p(\mathcal{D})}[\mathcal{L}_\theta(D)]$$

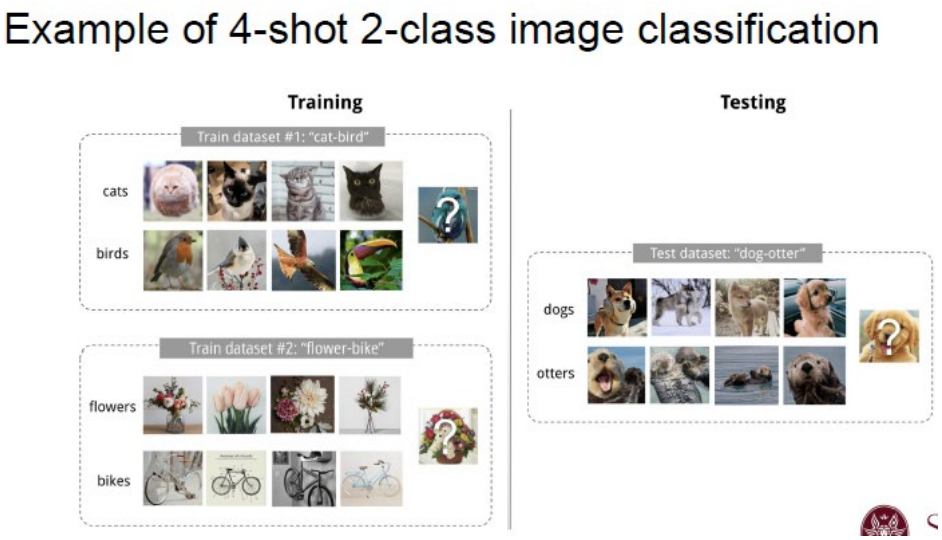
We want to find parameters θ such that we want to minimize the loss function $\mathcal{L}$. The difference from standard training is that we do not train on samples, but we train on tasks where one data sample here is consider as a whole full dataset.

1.1 Few Shot Learning

Few-shot classification is an instantiation of Meta-Learning in the field of Supervised Learning. The dataset is often composed as follows:

- S = Support set for Learning
- B = Prediction set for training or testing
- $D = < S, B >$

Imagine that for this Dataset D we have N possible classes. For each class, then, we have K samples. This is called the K-shot N-class classification task. Like, for example, if we speak about 4-Shot Binary Classification, we mean that we have 2 classes where each class has 4 samples.



2. Training

We know that a standard Training procedure is explained in this way (*let's remember maximizing the expectation is the same as minimizing the loss, so it's basically the same thing*):

- A dataset contains pairs of feature vectors and labels
  - $\mathscr{D}=\{(\mathbf{x}_i, y_i)\}$
  - each label belongs to a known set.
- Let's consider a classifier f_θ with parameter θ that outputs a probability of a data point belonging to the class y given the feature vector x, $P_\theta(y|\mathbf{x})$.
- The optimal parameters should maximize the probability of true labels across multiple training batches $B \subset \mathscr{D}$

$$\theta^* = \arg\max_\theta \mathbb{E}_{B \subset \mathcal{D}}[\sum_{(\mathbf{x},y)\in B} P_\theta(y|\mathbf{x})] \quad ; \text{trained with mini-batches.}$$

As we can see the set of parameters are the parameters of the Meta Learning Model, but now before sampling batches, we need to sample Tasks. In fact, we sample a Support Set S and a Prediction Set B (also called Batch) from a Dataset.

- The model is trained such that it can generalize to other datasets.

$$\theta = \arg\max_\theta E_{L \subset \mathcal{L}}[E_{S^L \subset \mathcal{D}, B^L \subset \mathcal{D}}[\sum_{(x,y)\in B^L} P_\theta(x, y, S^L)]]$$

Symbols in red are added for meta-learning in addition to the supervised learning objective.

- The idea is to some extent similar to using a pre-trained model in image classification (ImageNet) or language modeling (big text corpora) when only a limited set of task-specific data samples are available.
- Meta-learning takes this idea one step further, rather than fine-tuning according to one down-stream task, it optimizes the model to be good at many, if not all.

Additionally, we add a term S^L tells us that we can compute the probability of y given x, using the support set S^L as well. And this is how the model learns new tasks (because we give them to it).

Also, we could divide learning in 2 parts (Learner and Meta-Learner) where we first train a classifier (*learner*) on a given task and, in the meantime, we update the its weights through an optimizer g_ϕ through the support set S. Of course, in the optimization step, we need to update both θ and ϕ.

3. Approaches

There are 3 main approaches to Meta Learning, which involve:

- **Metric-based**
- **Model-based**
- **Optimization-based**

3.1 Metric-Based

The Metric Based approach is the most natural one as it tries to learn how to consider elements in the support set in order to make a decision. When we have a collection of pictures featuring flowers, parrots, and more, and we encounter a new picture, we compare it to the existing ones. If the new picture resembles parrots the most, for instance, we conclude that the new one is equivalent to parrots. In simpler terms, we identify the most similar category from our collection to classify the new image.

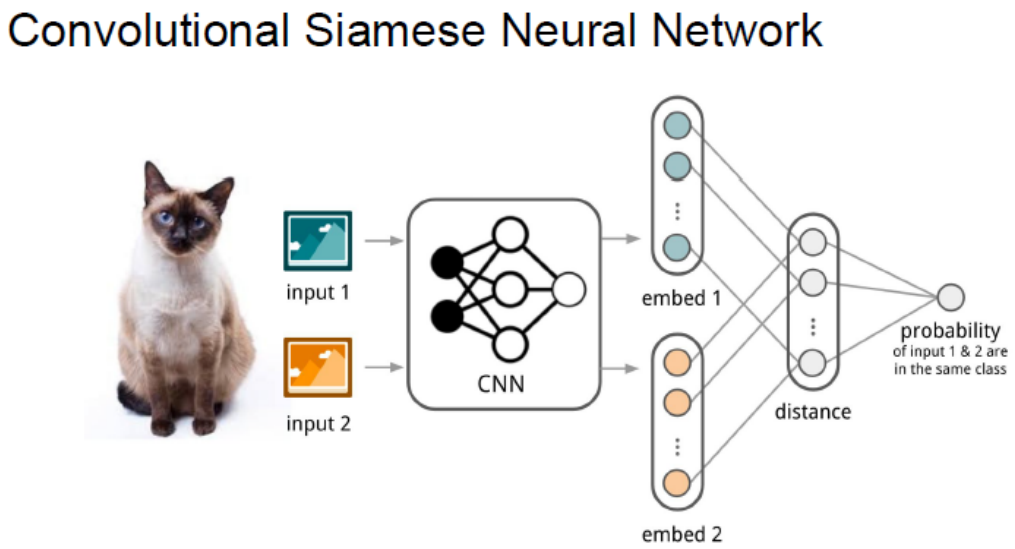
The **Metric-Based** approach follows, then, a **KNN** style. It uses the samples in the support set as training data for a general KNN classifier.

The weight is generated by a kernel function k_θ, measuring the similarity between two data samples.

$$P_\theta(y|\mathbf{x}, S) = \sum_{(\mathbf{x}_i, y_i) \in S} k_\theta(\mathbf{x}, \mathbf{x}_i) y_i$$

This is Meta-Learning as my model doesn't care about what x and x_i are. It only cares which are the closest points in the support set S.
In this case, from the formula above, we have to learn the parameters of a kernel which is involved in the process of measuring the similarity between two data points.

3.2 Convolutional Siamese Neural Network



First, the siamese network is trained for a verification task for telling whether two input images are in the same class.

- It outputs the probability of two images belonging to the same class.

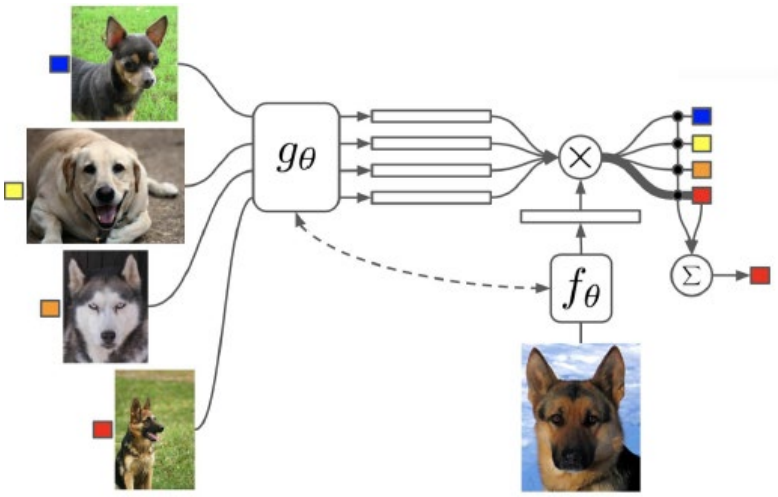
Then, during test time, the siamese network processes all the image pairs between a test image and every image in the support set.

- The final prediction is the class of the support image with the highest probability.
    - Given a support set S and a test image **x**, the final predicted class is:

$$\hat{c}_S(\mathbf{x}) = c(\arg\max_{\mathbf{x}_i \in S} P(\mathbf{x}, \mathbf{x}_i))$$

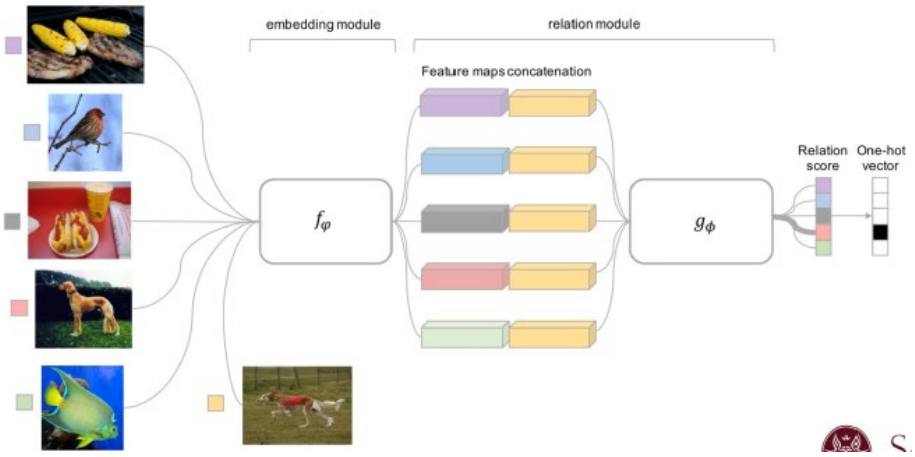
where c() is the class label of an image **x** and ĉ() is the predicted label.

3.3 Matching Networks



In the **Matching Network**, unlike in the CSNN, we don't train a distance function. Instead, we train an attention mask over the support set. In the Siamese approach, we evaluate the similarity of each support set sample with the input separately. However, in the Matching Network approach, we consider the entire support set together with the new image for classification. We assess the new image in the context of the entire support set, making a decision by looking at the support samples as a collective entity, rather than individually. The kernel itself is an attention mechanism that we want to learn during training.

3.4 Relation Network

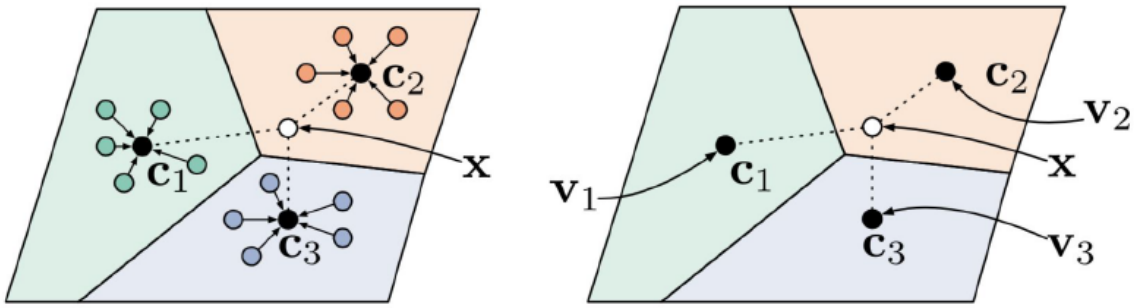


In this case, f_ρ embeds all of the elements including the ones in the prediction set. We take each embedding of the support set S (colored ones) and we concatenate them with the embedding of the prediction set. Then, through g_ϕ, we output similarity scores between the embeddings. The scores serve as a form of a SoftMax, like a multi-class classifier, resulting in the usual typical loss. Training involves an end-to-end process, simultaneously optimizing both the input encoder and the similarity computation component. In contrast to the Siamese networks where separate encoders handle distinct parts, the Matching Network's encoder (f_ρ) and similarity component (g_ϕ) consider the same elements during training.

3.5 Prototypical Network

Prototypical Networks use a function, denoted as f_θ, to turn each input into a feature vector. For each class, a prototype is defined as the average feature vector of all examples in that class. To decide the class for a new input, they calculate distances between the input and prototype vectors and use a SoftMax to get class probabilities. The loss function is a negative log-likelihood. Surprisingly, this method works well even when there are no examples in the support set S during training.

In a few-shot learning scenario, then, prototypes are learned for each class, similar to how centroids are computed in K-Means. However, in a Zero Shot learning scenario, where there is no support set available during training, Prototypical Networks face the challenge of making predictions for classes that have never been seen before.



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

In the model-based approach we had an optimizer and a learner where we had to optimize both parameters. This task is called "Learning to Learn by Gradient Descent by Gradient Descent".

The main idea is can we actually learn an algorithm to do Gradient Descent?

Gradient Descent update rule is of the form

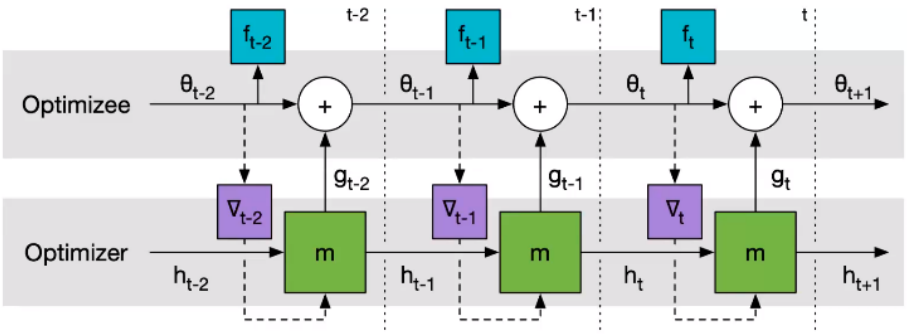
$$\theta_{t+1} = \theta_t - \alpha_t \nabla f(\theta_t)$$

that can be rewritten as

$$\theta_{t+1} = \theta_t + g_t(\nabla f(\theta_t), \phi)$$

where g_t is a function that determines how much the parameters should be updated w.r.t. gradient of f, and a set of parameters ϕ.

In this case, we learn ϕ simply by gradient descent. g is a function which takes in input the gradients. In the paper, they used LSTM which takes as input a sequence of gradients and predicts the next set of parameters, as in this architecture:



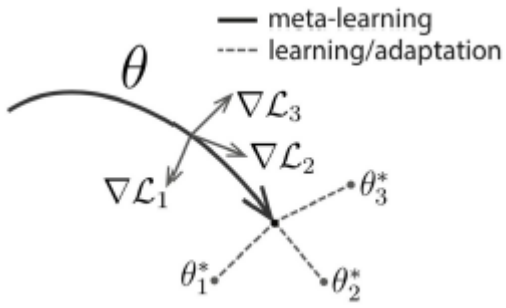
We have the Optimizer which is the LSTM that optimizes the optimizee which is the function g. In simple terms, it works by updating itself based on the history of gradients. This optimizer is then used to calculate new values for the parameters. This process repeats: the optimizer gets updated, provides new values to add to the parameters, and then the cycle

continues. It's like a loop where you keep refining and improving the parameters using the optimizer's calculations. For training, the errors made in the optimizer are not back propagated through the gradient computation. So, the gradient computation is a **non-differentiable** operation. The optimizer will, then, learn to minimize the distance between the output of the optimizer and the output of $-\alpha \cdot \nabla$ which is the standard SGD process. During testing, instead, you don't use gradients or optimize because you're not actively training the model. Instead, you already have known weights. You input data into the system and it implicitly computes a kind of gradient. This information is then used to shape the output through an optimization process, even though you're not explicitly training the model.

5. Model-Agnostic Meta-Learning (MAML)

Imagine that we want to learn a meta learner which is f_θ for a task which is called τ_i that has its associated dataset ($D_{train}^{(i)}$, $D_{test}^{(i)}$).

A meta-learning model-agnostic says that to achieve a good generalization across a variety of tasks, we would like to find the optimal θ^* so that the task-specific fine-tuning is more efficient.



Let's look at this picture: Imagine that the solid arrows ($\nabla\mathcal{L}_3, etc$) are the trajectories of your parameter set. The big θ is the set of parameters of the Meta Learner which means that, given a new task and by using that θ, my model should be able to work well on the new task.

But how do I exactly update the parameters?

First, I compute 3 different sets of new parameters, meaning that I update θ differently per task. For example, Task θ_1^* will give me $\nabla\mathcal{L}_1$, Task θ_2^* will give me $\nabla\mathcal{L}_2$ etc. Eventually the final update of θ will be the sum of all these updates. In a way we are not moving just in one direction given by a single dataset but we are moving in the direction that is common to all of the datasets, making the model more robust to different tasks.

Concluding, there are situations where we want to learn a generic model that, given a few data in input, adapts well to a new different task. The first choice that one could make is the Siamese Network, then the Relation Network and in case we want to not rely only on the support set but also optimize the learning, then one would use MAML.