

# Meta learning for variable number of classes

Dipankar Ghosh

UIN: 652811050

Graduation: Spring 2019

Advisor: Prof. Bing Liu

Second committee member: Prof. Bhaskar DasGupta



## **Table of contents**

<b>#</b>	<b>Contents</b>	<b>Page</b>
1	Abstract	4
2	Introduction	5
3	Few shot classification	6
4	Survey on Meta learning for few shot classification	7
5	-- Metric based	7
6	-- -- Convolutional siamese networks	7
7	-- -- Matching networks	8
8	-- Model based	9
7	-- -- Memory augmented neural networks	9
8	-- Optimization based	10
9	-- -- LSTM meta learner	10
10	-- -- Model agnostic meta learning	12
11	-- -- Reptile	14
12	Our task	16
13	Inspired & related work	16
14	Approach	16
15	Evaluation	17
16	Conclusion	18
17	Acknowledgement	19
18	References	19

## **Abstract**

There has been unprecedented growth in the field of supervised classification tasks where large data is present for each class. People have also tried to solve tasks where very few examples of each class are present by using several techniques of meta learning. However, all of these methods expect all the tasks to have a fixed number of classes in meta-training, as well as, meta-testing. This might lead to issues when we do not have required tasks with equal number of classes. We try to tackle this problem by constructing a generalized meta learning model which can handle any number of classes during meta-testing. We show that our accuracy during meta-test is quite competitive with the accuracies obtained with existing meta learning methods, which use fixed number of classes throughout.

## **Introduction**

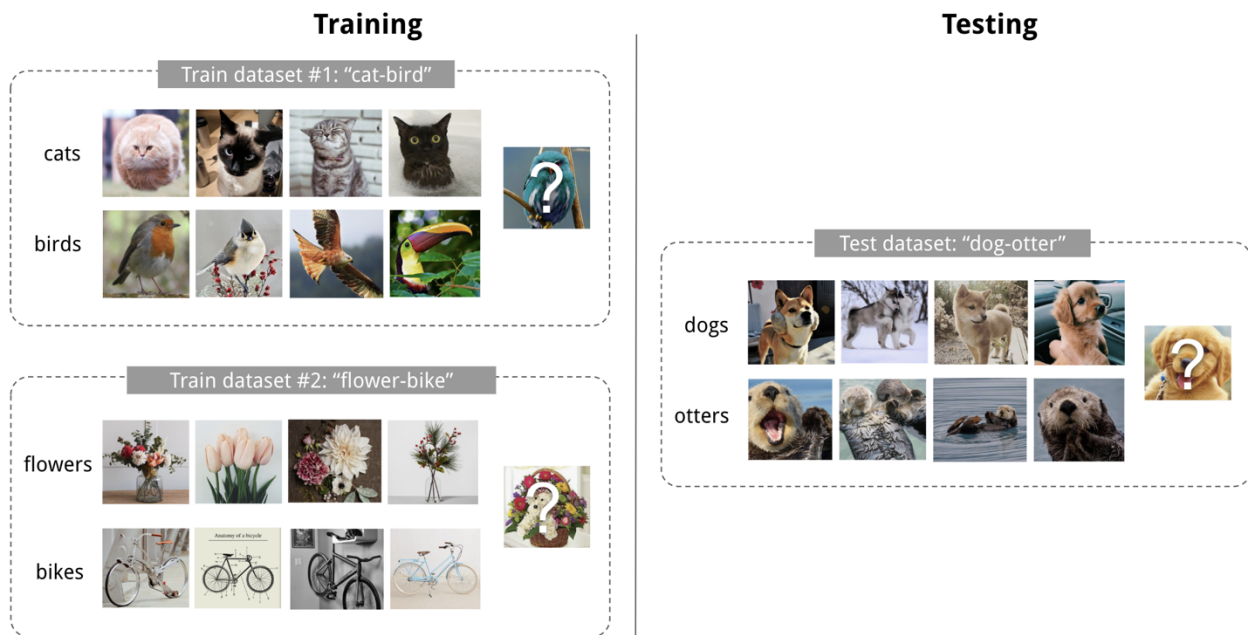
A good machine learning model is often built by training with a large number of samples. On the contrary, humans, learn new skills much faster and more efficiently. Kids easily learn to distinguish between cats and birds after seeing them only a few times. Is it possible to design a machine learning model with similar properties — learning new concepts and skills fast with just a few training examples? That's essentially what meta-learning aims to solve.

We expect a meta-learning model to be capable of adapting and generalizing well to new tasks and new environments which were not encountered during training time. The adaptation process, essentially a mini learning session, happens during test but with a limited exposure to the new task configurations. Eventually, the adapted model can complete new tasks. This is why meta-learning is also known as learning to learn. Building this kind of meta learning models became necessary since many times we do not have thousands of labelled examples per class, and also, there can be very useful applications of such models which can generalize after seeing just one example from each class to provide decent test performance.

The tasks can be any well-defined family of machine learning problems: supervised learning, reinforcement learning, etc. A good meta-learning model should be trained over a variety of learning tasks and optimized for the best performance on a distribution of tasks, including potentially unseen tasks. Each task is associated with a dataset  $D$ , containing both feature vectors and true labels. It solves the learning problem at two levels. Firstly, given several similar tasks, it tries to grab knowledge within each task separately. Along with that, it also extracts knowledge learned across all the tasks & tries to help the first with that acquired knowledge. As a result, when the system has already seen several tasks & learned from them, it can apply that knowledge to learn how a new task works very quickly.

## Few shot classification

Few-shot classification is an instantiation of meta-learning in the field of supervised learning. The tasks in few-shot learning problems are generally referred to as  $k$ -shot  $N$ -class tasks, where for each task, the dataset constitutes of ' $k$ ' labelled examples of each class, and in total there are ' $N$ ' classes. We have meta sets  $M_{\text{meta-train}}$ ,  $M_{\text{meta-validation}}$  &  $M_{\text{meta-test}}$ . Each of these meta sets consists of multiple tasks. Each task will have a training dataset  $D_{\text{train}}$ , on which we optimize a parameter  $\theta$ , and then evaluate its performance on the test set  $D_{\text{test}}$ . Here,  $D_{\text{train}}$  will have  $k*N$  labelled examples, and  $D_{\text{test}}$  will have a set number of examples consisting at most ' $N$ ' number of classes.



An example of 4-shot 2-class image classification

This technique is mostly utilized in the field of computer vision, where employing an object categorization model still gives appropriate results even without having several training samples. If we have only one image of a bird, this would be a one-shot machine learning problem.

## Survey on Meta learning for few shot classification

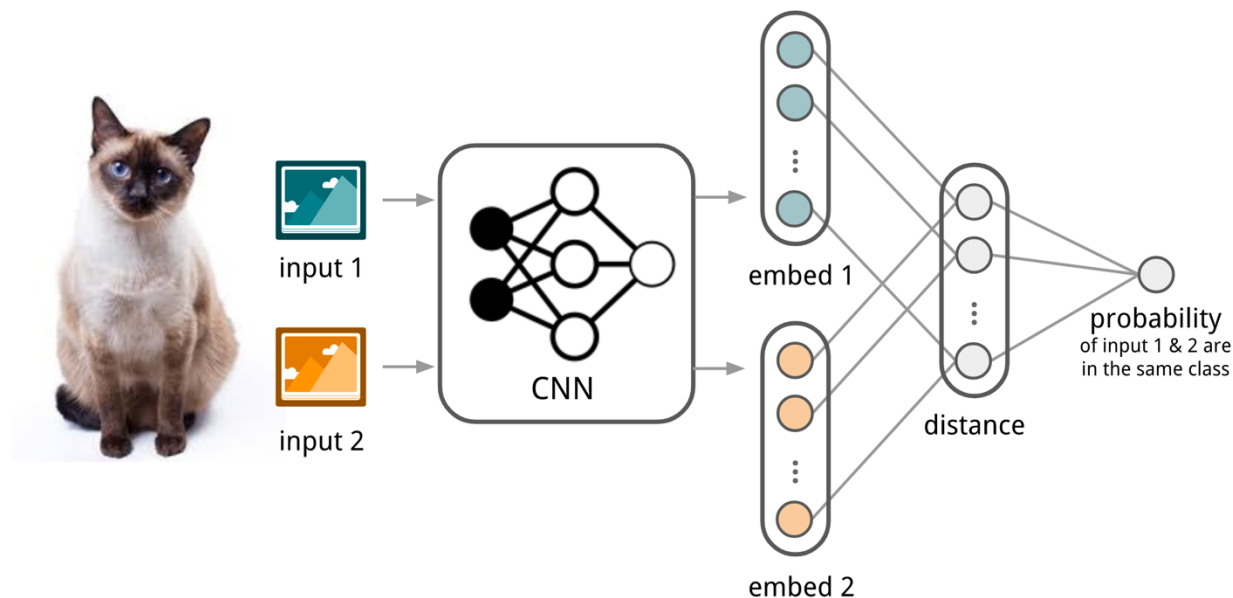
There are three broad categories of designing models that can learn new skills or adapt to new environments rapidly with a few examples: metric-based, model-based, and optimization-based.

### Metric based

The core idea in metric-based meta-learning is similar to nearest neighbors algorithms (i.e., k-NN classifier and k-means clustering) and kernel density estimation. The predicted probability over a set of known labels  $y$  is a weighted sum of labels of support set samples. The weight is generated by a kernel function  $k_\theta$ , measuring the similarity between two data samples.

$$P_\theta(y|x,S) = \sum_i (k_n(x, x_i) y_i)$$

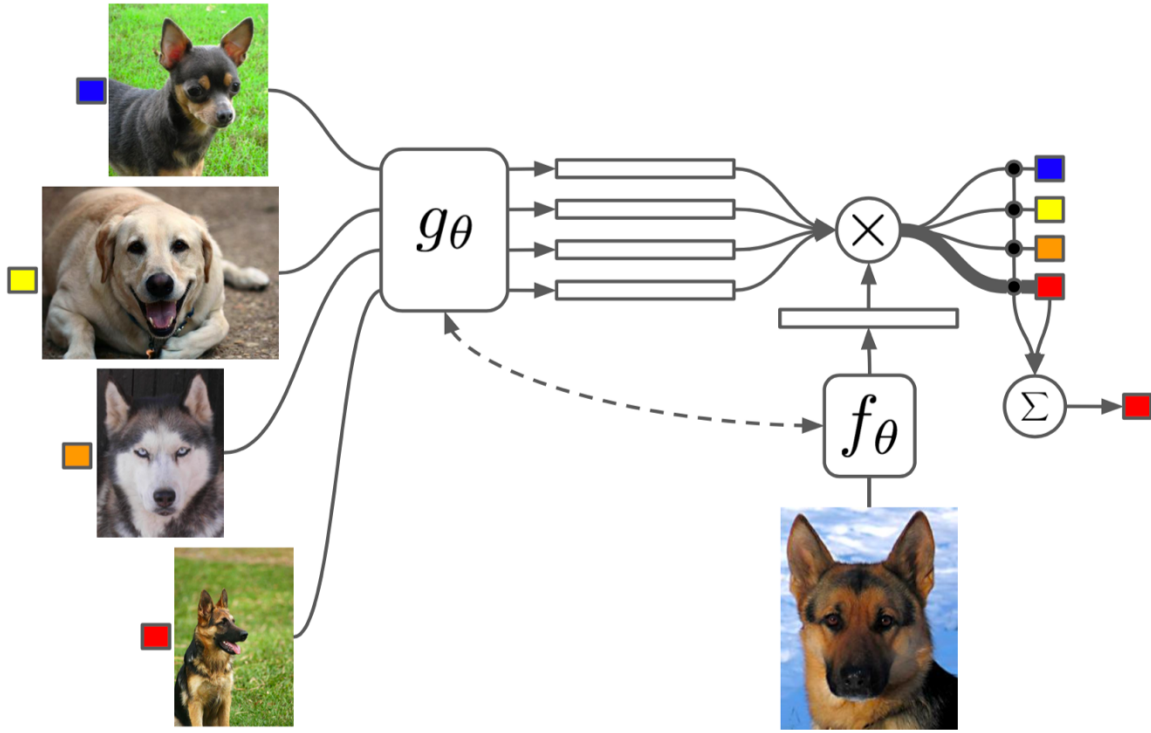
Convolutional Siamese Networks: A Siamese network ([Koch, Zemel & Salakhutdinov, 2015](#)) is composed of two twin networks and their outputs are jointly trained on top with a function to learn the relationship between pairs of input data samples. The twin networks are identical, sharing the same weights and network parameters. 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.



### Siamese networks architecture for few-shot image classification

First, the convolutional siamese network learns to encode two images into feature vectors via an embedding function which contains a couple of convolutional layers. The L1-distance between two embeddings is  $|f_\theta(x_i) - f_\theta(x_j)|$ . The distance is converted to a probability by a linear feedforward layer and sigmoid function. It is the probability of whether two images are drawn from the same class. The loss is cross entropy since the label is binary.

Matching Networks: A Matching Network (Vinyals et al., 2016) learns a classifier  $c_S$  for any given (small) support set  $S = \{x_i, y_i\}_{i=1}^k$  (k-shot classification). This classifier defines a probability distribution over output labels  $y$  given a test example  $x$ . Similar to other metric-based models, the classifier output is defined as a sum of labels of support samples weighted by attention kernel  $a(x, x_i)$  - which should be proportional to the similarity between  $x$  and  $x_i$ .



Matching Networks architecture

$$c_S(\mathbf{x}) = P(y|\mathbf{x}, S) = \sum_{i=1}^k a(\mathbf{x}, \mathbf{x}_i) y_i, \text{ where } S = \{(\mathbf{x}_i, y_i)\}_{i=1}^k$$

The attention kernel depends on two embedding functions,  $f$  and  $g$ , for encoding the test sample and the support set samples respectively. The attention weight between two data points is the cosine similarity,  $\text{cosine}(\cdot)$ , between their embedding vectors, normalized by softmax:

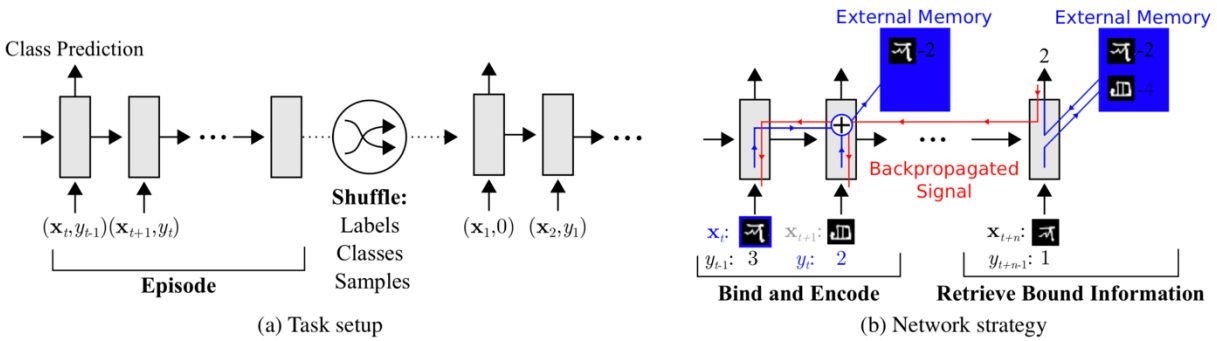
$$a(\mathbf{x}, \mathbf{x}_i) = \frac{\exp(\text{cosine}(f(\mathbf{x}), g(\mathbf{x}_i)))}{\sum_{j=1}^k \exp(\text{cosine}(f(\mathbf{x}), g(\mathbf{x}_j)))}$$



## Model based

Model-based meta-learning models make no assumption on the form of  $P_\theta(y|x)$ . Rather it depends on a model designed specifically for fast learning — a model that updates its parameters rapidly with a few training steps. This rapid parameter update can be achieved by its internal architecture or controlled by another meta-learner model.

Memory-Augmented Neural Networks: A family of model architectures use external memory storage to facilitate the learning process of neural networks, including Neural Turing Machines and Memory Networks. With an explicit storage buffer, it is easier for the network to rapidly incorporate new information and not to forget in the future. Such a model is known as MANN, short for “Memory-Augmented Neural Network” ([Santoro et al., 2016](#)). Note that recurrent neural networks with only internal memory such as vanilla RNN or LSTM are not MANNs.



MANN task setup

To use MANN for meta-learning tasks, we need to train it in a way that the memory can encode and capture information of new tasks fast and, in the meantime, any stored representation is easily and stably accessible. The training happens in an interesting way so that the memory is forced to hold information for longer until the appropriate labels are presented later. In each training episode, the truth label  $y_t$  is presented with one step offset,  $(x_{t+1}, y_t)$ : it is the true label for the input at the previous time step  $t$ , but presented as part of the input at time step  $t+1$ . In this way, MANN is motivated to memorize the information of a new dataset, because the memory has to hold the current input until the label is present later and then retrieve the old information to make a prediction accordingly.

The addressing mechanism for writing newly received information into memory operates a lot like the cache replacement policy. The Least Recently Used Access (LRUA) writer is designed for MANN to better work in the scenario of meta-learning. A LRUA write head prefers to write new content to either the least used memory location or the most recently used memory location.

There are many cache replacement algorithms and each of them could potentially replace the design here with better performance in different use cases. Furthermore, it would be a good idea to learn the memory usage pattern and addressing strategies rather than arbitrarily set it.

## Optimization based

Deep learning models learn through backpropagation of gradients. However, the gradient-based optimization is neither designed to cope with a small number of training samples, nor to converge within a small number of optimization steps. Is there a way to adjust the optimization algorithm so that the model can be good at learning with a few examples? This is what optimization-based approach meta-learning algorithms intend for.

*LSTM Meta-Learner*: The optimization algorithm can be explicitly modeled. [Ravi & Larochelle \(2017\)](#) did so and named it “meta-learner”, while the original model for handling the task is called “learner”. The goal of the meta-learner is to efficiently update the learner’s parameters using a small support set so that the learner can adapt to the new task quickly.

Let’s denote the learner model as  $M_\theta$  parameterized by  $\theta$ , the meta-learner as  $R_\Theta$  with parameters  $\Theta$ , and the loss function  $L$ .

Why LSTM?

The meta-learner is modeled as a LSTM, because:

1. There is similarity between the gradient-based update in backpropagation and the cell-state update in LSTM.
2. Knowing a history of gradients benefits the gradient update; think about how momentum works.

The update for the learner’s parameters at time step  $t$  with a learning rate  $\alpha_t$  is:

$$\theta_t = \theta_{t-1} - \alpha_t \nabla_{\theta(t-1)} L_t$$

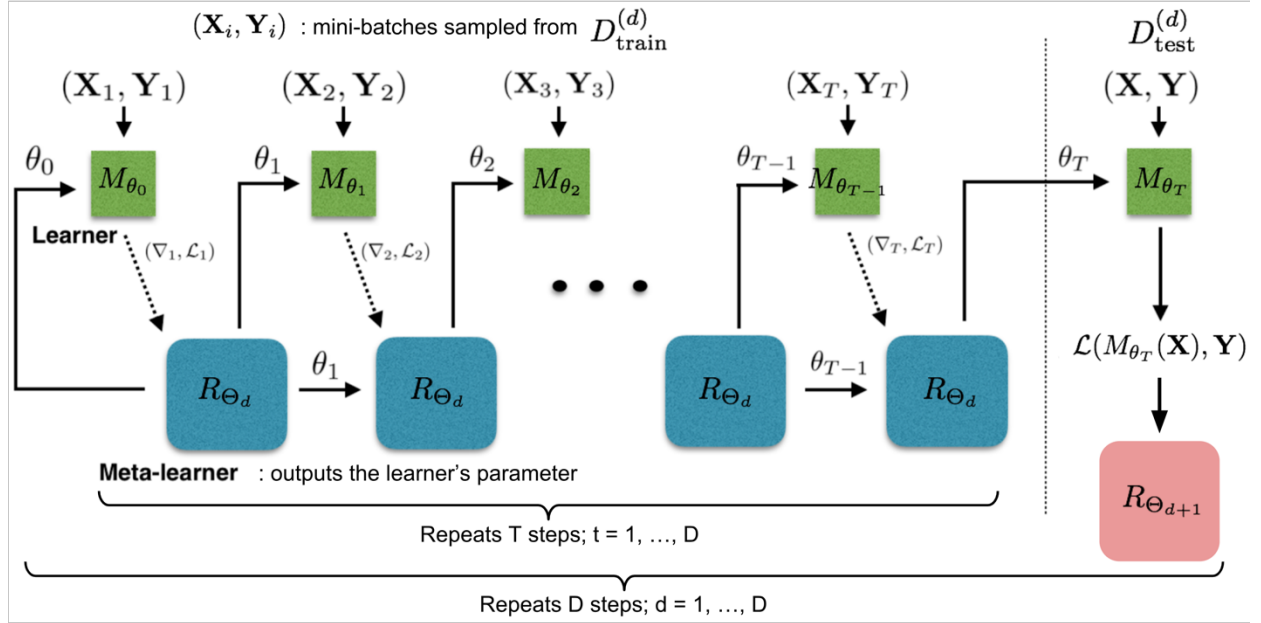
where  $\theta_{t-1}$  are the parameters of the learner after  $t-1$  updates,  $\alpha_t$  is the learning rate at time  $t$ ,  $L_t$  is the loss optimized by the learner for its  $t^{\text{th}}$  update,  $\nabla_{\theta(t-1)} L_t$  is the gradient of the loss with respect to parameters  $\theta_{t-1}$  and  $\theta_t$  is the updated parameters of the learner.

The above update rule resembles the update rule for the cell state in an LSTM

$$c_t = f_t \bullet c_{t-1} + i_t \bullet c'_t$$

if  $f_t = 1$ ,  $c_{t-1} = \theta_{t-1}$ ,  $i_t = \alpha_t$ , and  $c'_t = -\nabla_{\theta(t-1)} L_t$

Here, we have the cell state of the LSTM to be the parameters of the learner, or  $c_t = \theta_t$ , and the candidate cell state  $c'_t = -\nabla_{\theta(t-1)} L_t$ . Parametric forms have been defined for  $i_t$  &  $f_t$  so that the meta learner can determine optimal values through the course of the updates. Additionally, notice that we can also learn the initial value of the cell state  $c_0$  for the LSTM, treating it as a parameter of the meta-learner. This corresponds to the initial weights of the classifier (that the meta-learner is training). Learning this initial value lets the meta-learner determine the optimal initial weights of the learner so that training begins from a beneficial starting point that allows optimization to proceed rapidly.



The meta training conditions should match the meta test conditions in order to obtain good results. Therefore, when considering each dataset  $D \in M_{\text{meta-train}}$ , the training objective is the loss  $L_{\text{test}}$  of the produced classifier on  $D$ 's test set  $D_{\text{test}}$ .

While iterating over the examples in  $D$ 's training  $D_{\text{train}}$ , at each time step  $t$ , the LSTM meta learner receives  $(\nabla_{\theta(t-1)}L_t, L_t)$  from the classifier and proposes the new set of parameters  $\theta_t$ . This process repeats for several steps, after which the classifier and its final parameters are evaluated on the test set to produce the loss that is then used to train the meta learner.

Few implementation details to pay extra attention to:

1. How to compress the parameter space in LSTM meta-learner? As the meta-learner is modeling parameters of another neural network, it would have hundreds of thousands of variables to learn. Following the idea of sharing parameters across coordinates.
2. To simplify the training process, the meta-learner assumes that the loss  $L_t$  and the gradient  $\nabla_{\theta(t-1)}L_t$  are independent.

For the learner, we use a simple CNN containing 4 convolutional layers, each of which is a  $3 \times 3$  convolution with 32 filters, followed by batch normalization, a ReLU non-linearity, and lastly a  $2 \times 2$  max-pooling. The network then has a final linear layer followed by a softmax for the number of classes being considered. The loss function  $L$  is the average negative log-probability assigned by the learner to the correct class. For the meta-learner, we use a 2-layer LSTM, where the first layer is a normal LSTM and the second layer is the modified LSTM meta-learner. The gradients and losses are preprocessed and fed into the first layer LSTM, and the regular gradient coordinates are also used by the second layer LSTM to implement the state update rule. At each time step, the learner's loss and gradient is computed on a batch consisting of the entire training set  $D_{\text{train}}$ , because we consider training sets with only a total of 5 or 25 examples. We train our LSTM with ADAM using a learning rate of 0.001 and with gradient clipping using a value of 0.25.

---

**Algorithm 1** Train Meta-Learner

---

**Input:** Meta-training set  $\mathcal{D}_{meta-train}$ , Learner  $M$  with parameters  $\theta$ , Meta-Learner  $R$  with parameters  $\Theta$ .

```
1:  $\Theta_0 \leftarrow$  random initialization
2:
3: for  $d = 1, n$  do
4:    $D_{train}, D_{test} \leftarrow$  random dataset from  $\mathcal{D}_{meta-train}$ 
5:    $\theta_0 \leftarrow c_0$  ▷ Initialize learner parameters
6:
7:   for  $t = 1, T$  do
8:      $\mathbf{X}_t, \mathbf{Y}_t \leftarrow$  random batch from  $D_{train}$ 
9:      $\mathcal{L}_t \leftarrow \mathcal{L}(M(\mathbf{X}_t; \theta_{t-1}), \mathbf{Y}_t)$  ▷ Get loss of learner on train batch
10:     $c_t \leftarrow R((\nabla_{\theta_{t-1}} \mathcal{L}_t, \mathcal{L}_t); \Theta_{d-1})$  ▷ Get output of meta-learner using Equation 2
11:     $\theta_t \leftarrow c_t$  ▷ Update learner parameters
12:  end for
13:
14:   $\mathbf{X}, \mathbf{Y} \leftarrow D_{test}$ 
15:   $\mathcal{L}_{test} \leftarrow \mathcal{L}(M(\mathbf{X}; \theta_T), \mathbf{Y})$  ▷ Get loss of learner on test batch
16:  Update  $\Theta_d$  using  $\nabla_{\Theta_{d-1}} \mathcal{L}_{test}$  ▷ Update meta-learner parameters
17:
18: end for
```

---

LSTM meta learner algorithm

MAML: short for Model-Agnostic Meta-Learning (Finn, et al. 2017) is a fairly general optimization algorithm, compatible with any model that learns through gradient descent.

Let's say our model is  $f_\theta$  with parameters  $\theta$ . Given a task  $\tau_i$  and its associated dataset ( $D_{train}^{(i)}$ ,  $D_{test}^{(i)}$ ), we can update the model parameters by one or more gradient descent steps (the following example only contains one step):

$$\theta'_i = \theta - \alpha \nabla_{\theta} L_{\tau_i}^{(0)}(f_\theta)$$

where  $L^{(0)}$  is the loss computed using the mini data batch with id (0).

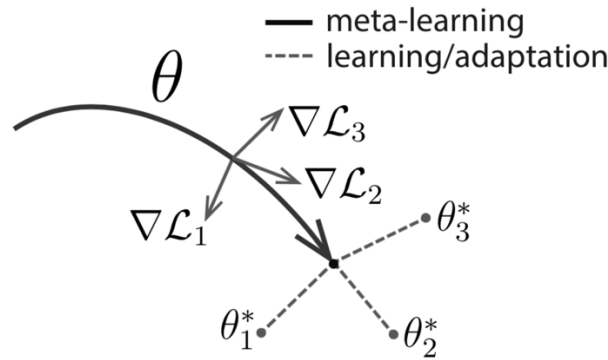


Diagram of MAML

Well, the above formula only optimizes for one task. To achieve a good generalization across a variety of tasks, we would like to find the optimal  $\theta^*$  so that the task-specific fine-tuning is more efficient. Now, we sample a new data batch with id (1) for updating the meta-objective. The loss, denoted as  $L^{(1)}$ , depends on the mini batch (1). The superscripts in  $L^{(0)}$  and  $L^{(1)}$  only indicate different data batches, and they refer to the same loss objective for the same task.

$$\theta^* = \arg \min_{\theta} \sum_{\tau_i \sim p(\tau)} \mathcal{L}_{\tau_i}^{(1)}(f_{\theta'_i}) = \arg \min_{\theta} \sum_{\tau_i \sim p(\tau)} \mathcal{L}_{\tau_i}^{(1)}(f_{\theta - \alpha \nabla_{\theta} \mathcal{L}_{\tau_i}^{(0)}(f_{\theta})})$$

$$\theta \leftarrow \theta - \beta \nabla_{\theta} \sum_{\tau_i \sim p(\tau)} \mathcal{L}_{\tau_i}^{(1)}(f_{\theta - \alpha \nabla_{\theta} \mathcal{L}_{\tau_i}^{(0)}(f_{\theta})}) \quad ; \text{ updating rule}$$

---

**Algorithm 1** Model-Agnostic Meta-Learning

---

**Require:**  $p(\mathcal{T})$ : distribution over tasks

**Require:**  $\alpha, \beta$ : step size hyperparameters

- 1: randomly initialize  $\theta$
  - 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** Note: the meta-update is using different set of data.
  - 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**
- 

First-Order MAML: The meta-optimization step above relies on second derivatives. To make the computation less expensive, a modified version of MAML omits second derivatives, resulting in a simplified and cheaper implementation, known as First-Order MAML (FOMAML). Let's consider the case of performing  $k$  inner gradient steps,  $k \geq 1$ . Starting with the initial model parameter  $\theta_{\text{meta}}$ :

$$\begin{aligned} \theta_0 &= \theta_{\text{meta}} \\ \theta_1 &= \theta_0 - \alpha \nabla_{\theta} \mathcal{L}^{(0)}(\theta_0) \\ \theta_2 &= \theta_1 - \alpha \nabla_{\theta} \mathcal{L}^{(0)}(\theta_1) \\ &\dots \\ \theta_k &= \theta_{k-1} - \alpha \nabla_{\theta} \mathcal{L}^{(0)}(\theta_{k-1}) \end{aligned}$$

Then in the outer loop, we sample a new data batch for updating the meta-objective.

$$\theta_{\text{meta}} \leftarrow \theta_{\text{meta}} - \beta g_{\text{MAML}} \quad ; \text{ update for meta-objective}$$

where  $g_{\text{MAML}} = \nabla_{\theta} \mathcal{L}^{(1)}(\theta_k)$

$$\begin{aligned}
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot (\nabla_{\theta_{k-1}} \theta_k) \dots (\nabla_{\theta_0} \theta_1) \cdot (\nabla_{\theta} \theta_0) && \text{; following the chain rule} \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \prod_{i=1}^k \nabla_{\theta_{i-1}} \theta_i \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \prod_{i=1}^k \nabla_{\theta_{i-1}} (\theta_{i-1} - \alpha \nabla_{\theta} \mathcal{L}^{(0)}(\theta_{i-1})) \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \prod_{i=1}^k (I - \alpha \nabla_{\theta_{i-1}} (\nabla_{\theta} \mathcal{L}^{(0)}(\theta_{i-1})))
\end{aligned}$$

The MAML gradient is:

$$g_{\text{MAML}} = \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \prod_{i=1}^k (I - \alpha \nabla_{\theta_{i-1}} (\nabla_{\theta} \mathcal{L}^{(0)}(\theta_{i-1})))$$

The First-Order MAML ignores the second derivative part in red. It is simplified as follows, equivalent to the derivative of the last inner gradient update result.

$$g_{\text{FOMAML}} = \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k)$$

*Reptile*: Reptile (Nichol, Achiam & Schulman, 2018) is a remarkably simple meta-learning optimization algorithm. It is similar to MAML in many ways, given that both rely on meta-optimization through gradient descent and both are model-agnostic.

The Reptile works by repeatedly:

1. sampling a task,
2. training on it by multiple gradient descent steps,
3. and then moving the model weights towards the new parameters.

See the algorithm below:  $\text{SGD}(L_{\tau_i}, \theta, k)$  performs stochastic gradient update for  $k$  steps on the loss  $L_{\tau_i}$  starting with initial parameter  $\theta$  and returns the final parameter vector. The batch version samples multiple tasks instead of one within each iteration. The reptile gradient is defined as  $(\theta - W)/\alpha$ , where  $\alpha$  is the stepsize used by the SGD operation.

---

**Algorithm 2** Reptile, batched version

---

Initialize  $\theta$

**for** iteration = 1, 2, ... **do**

    Sample tasks  $\tau_1, \tau_2, \dots, \tau_n$

**for**  $i = 1, 2, \dots, n$  **do**

        Compute  $W_i = \text{SGD}(L_{\tau_i}, \theta, k)$

**end for**

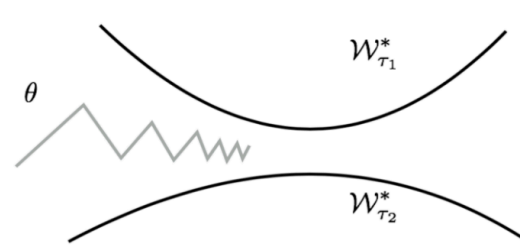
    Update  $\theta \leftarrow \theta + \beta \frac{1}{n} \sum_{i=1}^n (W_i - \theta)$

**end for**

---

At a glance, the algorithm looks a lot like an ordinary SGD. However, because the task-specific optimization can take more than one step. it eventually makes  $\text{SGD}(\mathbb{E}_\tau[L_\tau], \theta, k)$  diverge from  $\mathbb{E}_\tau[\text{SGD}(L_\tau, \theta, k)]$  when  $k > 1$ .

The Optimization Assumption: Assuming that a task  $\tau \sim p(\tau)$  has a manifold of optimal network configuration,  $\mathcal{W}_\tau^*$ . The model  $f_\theta$  achieves the best performance for task  $\tau_i$  when  $\theta$  lays on the surface of  $\mathcal{W}_\tau^*$ . To find a solution that is good across tasks, we would like to find a parameter close to all the optimal manifolds of all tasks:

$$\theta^* = \arg \min_{\theta} \mathbb{E}_{\tau \sim p(\tau)} \left[ \frac{1}{2} \text{dist}(\theta, \mathcal{W}_\tau^*)^2 \right]$$


The Reptile algorithm updates the parameter alternatively to be closer to the optimal manifolds of different tasks.

Let's use the L2 distance as  $\text{dist}(\cdot)$  and the distance between a point  $\theta$  and a set  $\mathcal{W}_\tau^*$  equals to the distance between  $\theta$  and a point  $\mathcal{W}_\tau^*(\theta)$  on the manifold that is closest to  $\theta$ :

$$\text{dist}(\theta, \mathcal{W}_\tau^*) = \text{dist}(\theta, \mathcal{W}_\tau^*(\theta)), \text{ where } \mathcal{W}_\tau^*(\theta) = \arg \min_{W \in \mathcal{W}_\tau^*} \text{dist}(\theta, W)$$

The gradient of the squared euclidean distance is:

$$\begin{aligned} \nabla_{\theta} \left[ \frac{1}{2} \text{dist}(\theta, \mathcal{W}_{\tau_i}^*)^2 \right] &= \nabla_{\theta} \left[ \frac{1}{2} \text{dist}(\theta, \mathcal{W}_{\tau_i}^*(\theta))^2 \right] \\ &= \nabla_{\theta} \left[ \frac{1}{2} (\theta - \mathcal{W}_{\tau_i}^*(\theta))^2 \right] \\ &= \theta - \mathcal{W}_{\tau_i}^*(\theta) \end{aligned} \quad ; \text{ See notes.}$$

Notes: According to the Reptile paper, “the gradient of the squared euclidean distance between a point  $\Theta$  and a set  $S$  is the vector  $2(\Theta - p)$ , where  $p$  is the closest point in  $S$  to  $\Theta$ ”. Technically the closest point in  $S$  is also a function of  $\Theta$ .

Thus, the update rule for one stochastic gradient step is:

$$\theta = \theta - \alpha \nabla_{\theta} \left[ \frac{1}{2} \text{dist}(\theta, \mathcal{W}_{\tau_i}^*)^2 \right] = \theta - \alpha (\theta - \mathcal{W}_{\tau_i}^*(\theta)) = (1 - \alpha) \theta + \alpha \mathcal{W}_{\tau_i}^*(\theta)$$

The closest point on the optimal task manifold  $\mathcal{W}_\tau^*(\theta)$  cannot be computed exactly, but Reptile approximates it using  $\text{SGD}(L_\tau, \theta, k)$ .

## Our task

We extend the above ideas to tasks with a variety in the number of classes, resembling more of a real-life scenario. The meta learning model will be trained to generalize well among tasks with any number of classes in them. Hence, during testing, the meta learner can efficiently converge the learner network, where a new task is thrown with an arbitrary number of classes. Such an approach will help alleviate forming tasks with a defined number of classes. All the classes, no matter how many are there, will be given equal importance, and all of them will be used during classification.

## Inspired & related work

Our work is on top of Optimization as a Model for Few-Shot Learning ([Ravi & Larochelle, 2017](#)), and Reptile (Nichol, Achiam & Schulman, 2018). We are using similar meta learning models & overall approach to do few-shot learning. We have also taken ideas from Learning to learn by Gradient Descent by Gradient Descent and Siamese Networks.

After observing how the models are applied to achieve good performance on few-shot learning problems, we thought what if the number of classes vary across tasks that are used to build few-shot learning problems?

## Approach

Like mentioned before, the tasks in few-shot learning problems are generally referred to as k-shot N-class tasks, where for each task, the dataset constitutes of 'k' labelled examples of each class, and in total there are 'N' classes. In our case, 'N' will vary across different tasks. Now in the meta learning system, we have meta sets  $M_{\text{meta-train}}$ ,  $M_{\text{meta-validation}}$  &  $M_{\text{meta-test}}$ . Each of these meta sets consists of multiple tasks. Each task will have a training dataset  $D_{\text{train}}$ , on which we optimize a parameter  $\theta$ , and then evaluate its performance on the test set  $D_{\text{test}}$ . Here,  $D_{\text{train}}$  will have  $k*N$  labelled examples, and  $D_{\text{test}}$  will have a set number of examples consisting at most 'N' number of classes.

During meta training phase, 'N' will be assigned a value at random from  $\{2, 4, 6, 8, 10\}$  and the corresponding task with  $D_{\text{train}}$  &  $D_{\text{test}}$  will be defined. 'k' will have a fixed value throughout. Over the course of training & evaluation on various tasks, the meta learner network tries to learn a procedure by which it can take any  $D_{\text{train}}$  as input and produce a classifier which can perform well on the corresponding  $D_{\text{test}}$ . After a set number of iterations through tasks in  $M_{\text{meta-train}}$ , we perform hyper-parameter selection of the meta learner using  $M_{\text{meta-validation}}$ . Then finally, we evaluate its generalization performance on  $M_{\text{meta-test}}$ .



## Evaluation

We consider the k-shot, mixedN-class classification setting where a meta-learner trains on many related but small training sets of k examples for each of mixedN (a random value from {2, 4, 6, 8, 10} classes. We first split the list of all classes in the data into disjoint sets and assign them to each meta-set of meta-training, meta-validation, and meta-testing. To generate each instance of a k-shot, mixedN-class task dataset  $D = (D_{\text{train}}, D_{\text{test}}) \in M$ , we do the following: we first sample classes from the list of classes corresponding to the meta-set we consider. We then sample k examples from each of those classes. These k examples together compose the training set  $D_{\text{train}}$ . Then, an additional fixed amount of the rest of the examples are sampled to yield a test set  $D_{\text{test}}$ . We generally have 15 examples per class in the test sets. When training the meta-learner, we iterate by sampling these datasets (episodes) repeatedly. For meta-validation and meta-testing, however, we produce a fixed number of these datasets to evaluate each method. We produce enough datasets to ensure that the confidence interval of the mean accuracy is small.

We performed meta-training & meta-testing on N-class setting as given in the original paper, with values of  $N = 2, 4, 6, 8, 10$ , each separately. Then we performed our own meta-train setting, where for every iteration, we randomly picked a value from {2, 4, 6, 8, 10} and formed a task with that many classes. So basically, on every iteration, we are working on a task which has a random number of classes in it. Next, while meta-testing we observed that it performs quite as good as the fixed number of classes setting.

Here is a comparison of our results along with the original results:

#classes	LSTM Accuracy (%)				Reptile Accuracy (%)			
	1-shot		5-shot		1-shot		5-shot	
	Fixed	Mixed	Fixed	Mixed	Fixed	Mixed	Fixed	Mixed
2	66.87	66.28	83.92	83.29	74.32	73.11	83.53	83.03
3	55.12	54.29	72.56	72.2	59.91	58.23	75.11	73.97
4	48.05	46.84	65.42	65.62	52.62	51.25	68.74	65.96
5	43.14	40.23	60.21	59.15	46.88	45.95	62.76	61.01
6	36.67	35.48	55.61	54.48	43.08	41.48	57.53	57.12
7	33.86	33.39	50.08	49.36	38.91	37.92	53.48	53.1
8	31.06	30.42	47.33	46.03	36.51	35.49	51.09	49.79
9	30.22	29.15	45.1	44.21	34.18	33.43	49.01	47.21
10	26.49	26.02	43.62	41.03	31.74	31.08	45.8	45.46

## **Conclusion**

In this few-shot learning setting we tried to generalize the LSTM-based meta learner ([Ravi & Larochelle, 2017](#)) & Reptile ([Nichol, Achiam & Schulman, 2018](#)) on a mixed number of classes. That way, we won't have to build tasks with specific (fixed) number of classes throughout the train & test phase. Our intuition was that the meta learner should try to learn the optimization algorithm of the classifier irrespective of the number of classes in the tasks fed to it.

We have demonstrated and verified that meta learners can learn the learning algorithm of another network even if the tasks are diverse, which would help a lot in the few-shot regime. However, we have done all our experiments on the ImageNet dataset. We believe that our idea should work similarly in case of textual data, and our future research will be on textual tasks.

## **Acknowledgement**

I whole-heartedly thank Professor Bing Liu for his extraordinary support and innovative ideas.

## **References**

Koch, Gregory, Richard Zemel, and Ruslan Salakhutdinov. "Siamese neural networks for one-shot image recognition." In ICML deep learning workshop, vol. 2. 2015.

Vinyals, Oriol, Charles Blundell, Timothy Lillicrap, and Daan Wierstra. "Matching networks for one shot learning." In Advances in neural information processing systems, pp. 3630-3638. 2016.

Santoro, Adam, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. "Meta-learning with memory-augmented neural networks." In International conference on machine learning, pp. 1842-1850. 2016.

Ravi, Sachin, and Hugo Larochelle. "Optimization as a model for few-shot learning." (2016).  
Finn, Chelsea, Pieter Abbeel, and Sergey Levine. "Model-agnostic meta-learning for fast adaptation of deep networks." In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 1126-1135. JMLR. org, 2017.

Nichol, Alex, and John Schulman. "Reptile: a scalable metalearning algorithm." arXiv preprint arXiv:1803.02999 2 (2018).

Andrychowicz, Marcin, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. "Learning to learn by gradient descent by gradient descent." In Advances in Neural Information Processing Systems, pp. 3981-3989. 2016.