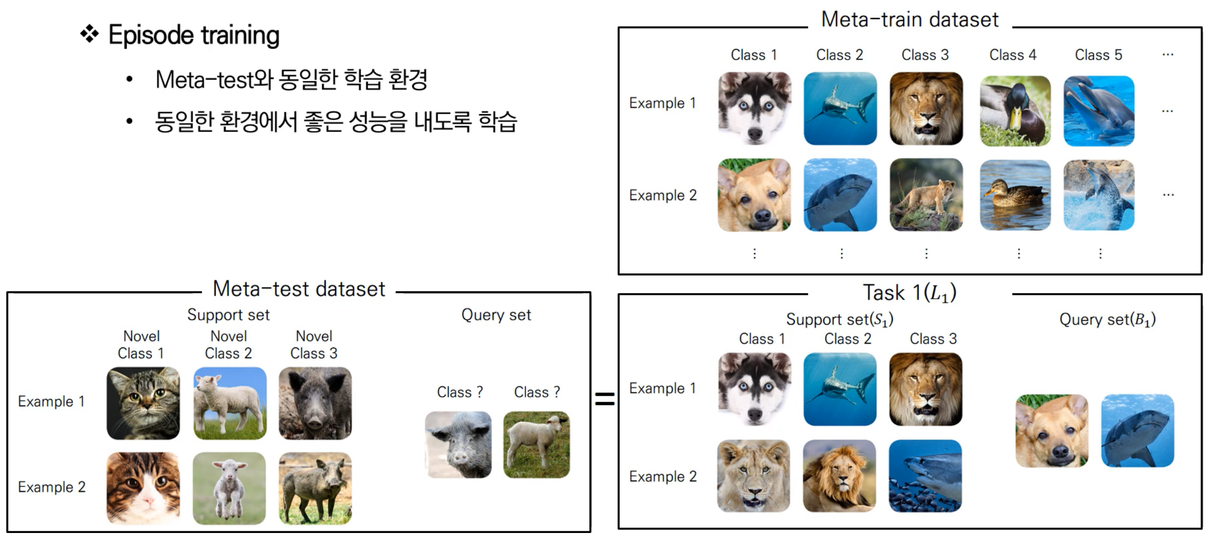
Few-shot Learning; Self-supervised Learning

[Seminar Video] Metric-based Approaches to Meta-learning

Common Terminology regarding a Dataset for the Meta-learning



❖ Episode training

- Meta-test와 동일한 학습 환경
- 동일한 환경에서 좋은 성능을 내도록 학습

What is a Metric-based Approach to Meta-learning?

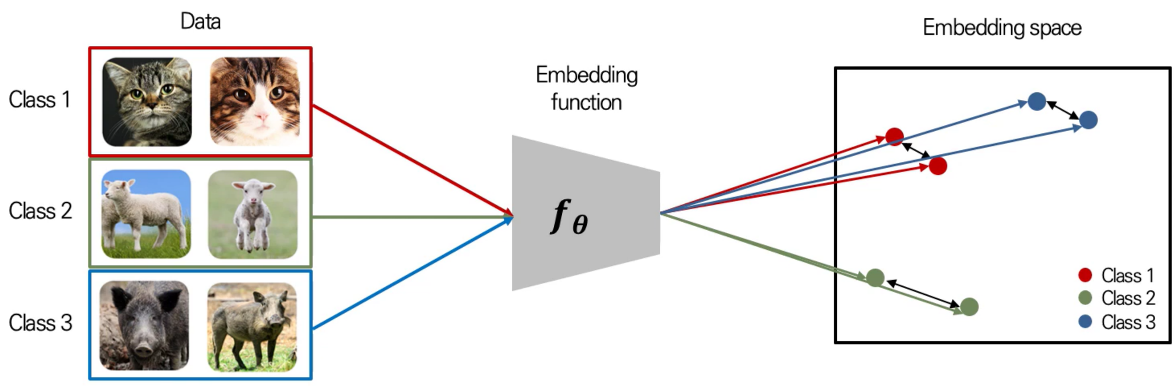
Metric-learning의 개념을 이용해서 meta-learning에 적용시킨 그런 방법론들을 지칭: Deep siamese network; Matching network; Protypical network; Relation network;

❖ Embedding function

- 데이터를 저차원으로 임베딩하는 함수

❖ Distance

- 임베딩 공간에서 데이터간 거리



Metric-learning에 있어서 가장 중요한 개념 두가지: Embedd∈gfunction and Distance. 위의 그림에서의 metric-learning은 embedding space상에서 같은 class끼리의 embedding vectors들은 가깝게, 다른 class끼리의 embedding vectors들은 서로 멀게 학습한다.

Reference: http://dmqm.korea.ac.kr/activity/seminar/301

G. Koch, et al., 2015, "Siamese Neural Networks for One-Shot Image Recognition"

- *Metric-learning approach*

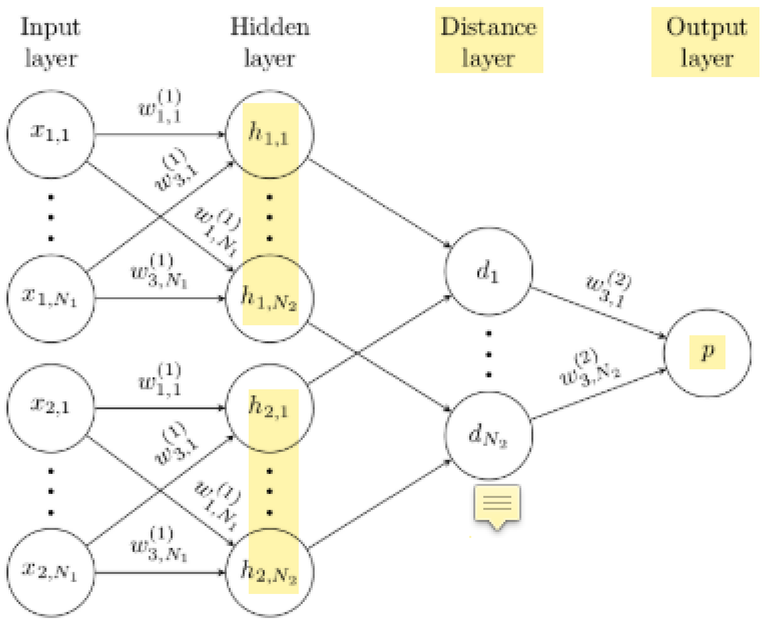


Figure 3. A simple 2 hidden layer siamese network for binary classification with logistic prediction p. The structure of the network is replicated across the top and bottom sections to form twin networks, with shared weight matrices at each layer.

The twin networks (entire weights are shared) take two different input (images) and output two *representation vectors* - h_1

and h_2. The two vectors are compared (merged) by a L1 distance metric, and the output probability p is 1 if the two input images belong to the same class, if not, p is zero. The training is conducted by minimizing the binary cross entropy of p

. During the training, the representation vectors are learned. The detailed training procedure can be found in this PyTorch Siamese implementation on Github.

F. Sung, et al., 2017, "Matching Networks for One Shot Learning"

- *Metric-learning approach*

The previous version of the Prototypical network.

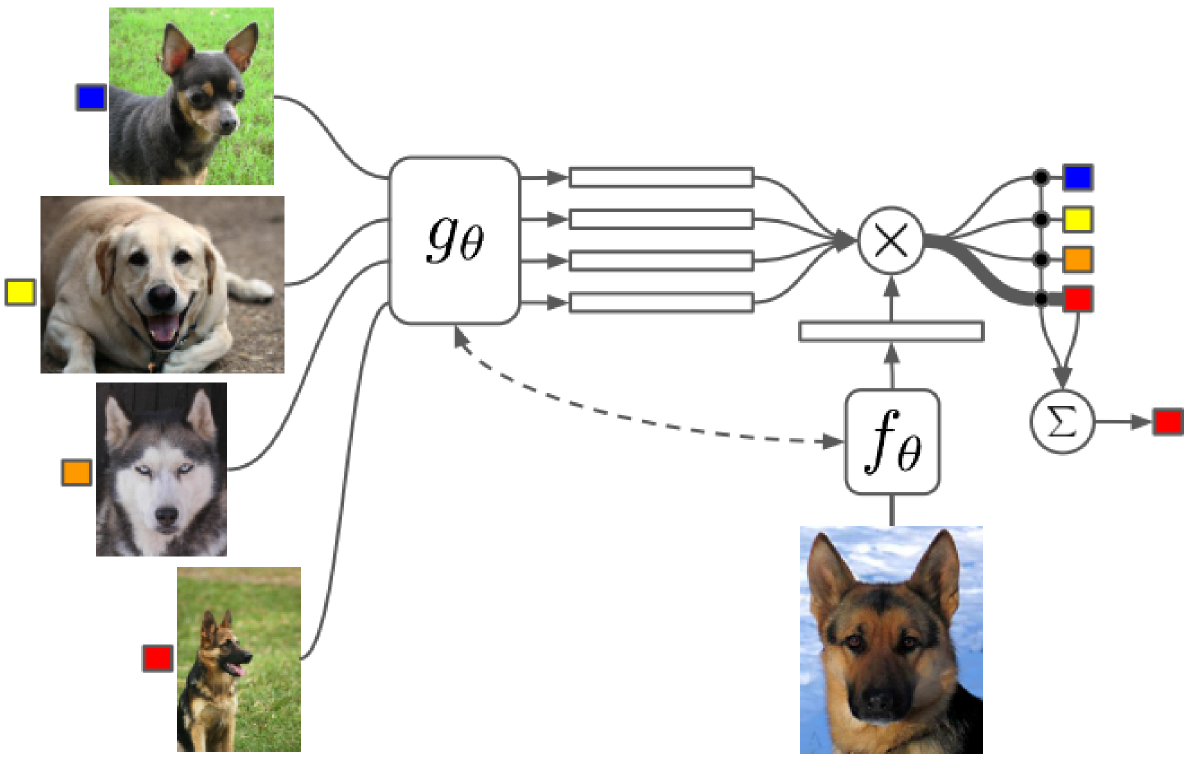


Figure 1: Matching Networks architecture

The model in its simplest form computes a predicted label ^y

as follows:

$\text{^}y={}_{k}\sum_{i=1}att(\text{^}x,x_i)y_i(1)$

where ^x

, x_i, and y_i are a test example, and samples and labels from the support set $S=\{(x_i,y_i)\}_{ki=1}$, respectively, and att is an attention mechanism. Note that Eq.(1) essentially describes the output for a new class as a linear combination of the labels in the support set. The attention value would be zero if x_i and ^x

are far away from each other and an appropriate constant otherwise.

Eq.(1)

relies on choosing att(.,.), which fully specifies the classifier. The simplest form is to use the softmax over the cosine distance c

. That is,

$att(\text{^}x,x_i)=exp(c(f(\text{^}x),g(x_i)))\sum_{kj=1}exp(c(f(\text{^}x),g(x_i)))$

where f

and g are appropriate nueral networks (potentially $f=g$) to embed $\hat{x}$ and x_i

.

The *episodic training (episode-based training)* was first proposed in this paper, which is well utilized for the Relation Network (F. Sung, 2018).

Episodic Training (Referred to from the Relation Network paper)

We consider the task of few-shot classifier learning. Formally, we have three datasets: a *training set*, a *support set*, and a *testing set*. The support set and testing set share the same label space, but the training set has its own label space that is disjoint with support/testing set. If the support set contains K

labelled examples for each of C unique classes, the target few-shot problem is called C-way K

-shot.

With the support set only, we can in principle train a classifier to assign a class label $\hat{y}$

to each sample $\hat{x}$

in the test set. However, due to the lack of labelled samples in the support set, the performance of such a classifier is usually not satisfactory. Therefore, we aim to perform meta-learning on the training set, in order to extract transferrable knowledge that will allow us to perform better few-shot learning on the support set and thus classify the test set more successfully.

An effective way to exploit the training set is to mimic the few-shot learning setting via *episode*-based training as proposed in this paper. In each training iteration, an episode is formed by randomly selecting C

classes from the training set with K labelled samples from each of the C classes to act as the *sample* set $S=\{(x_i,y_i)\}_{i=1}^{m}$ $(m=K\times C)$, as well as a fraction of the remainder of those C classes' samples to serve as the *query* set $Q=\{(x_j,y_j)\}_{j=1}^{n}$

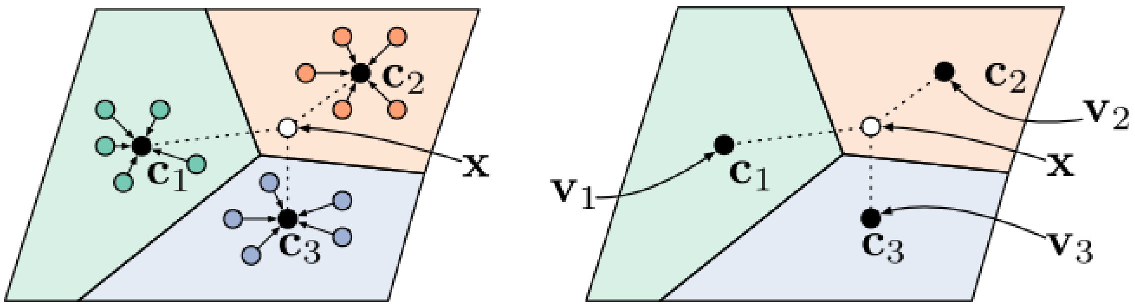
This sample/query set split is designed to simulate the support/test set that will be encountered at test time.

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

- Metric-learning approach

This network is based on a matching *network,* which uses an attention mechanism over a learned embedding of the labelled set of examples (the support set) to predict classes for the unlabeled points (the query set). The matching network can be interpreted as a weighted nearest-neighbour classifier applied within an embedding space.

The prototypical network is based on the idea that there exists an embedding in which points cluster around a single *prototype* representation for each class. In order to do this, we learn a non-linear mapping of the input into an embedding space using a neural network and take a class's prototype to be the mean of its support set in the embedding space. Classification is then performed for an embedded query point by simply finding the nearest class prototype. We follow the same approach to tackle zero-shot learning.



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

Learning Process

In few-shot classification, we are given a small support set of N

labelled examples $S=\{(x_1,y_1),\cdots,(x_N,y_N)\}$ where each $x_i \in R_D$ is the D-dimensional feature vector of an example and $y_i \in \{1,\cdots,K\}$ is the corresponding label. S_k denotes the set of examples labelled with class k

.

Prototypical networks compute an M

-dimensional representation $c_k \in R_M$, or *prototype*, of each class through an embedding function $f_\phi:R_D \rightarrow R_M$ with learnable parameters ϕ

. Each prototype is the mean vector of the embedded support points belonging to its class:

$c_k=1|S_k|\sum_{(x_i,y_i)\in S_k} f_\phi(x_i)$

Given a distance function d

, prototypical networks produce a distribution over classes for a query point x

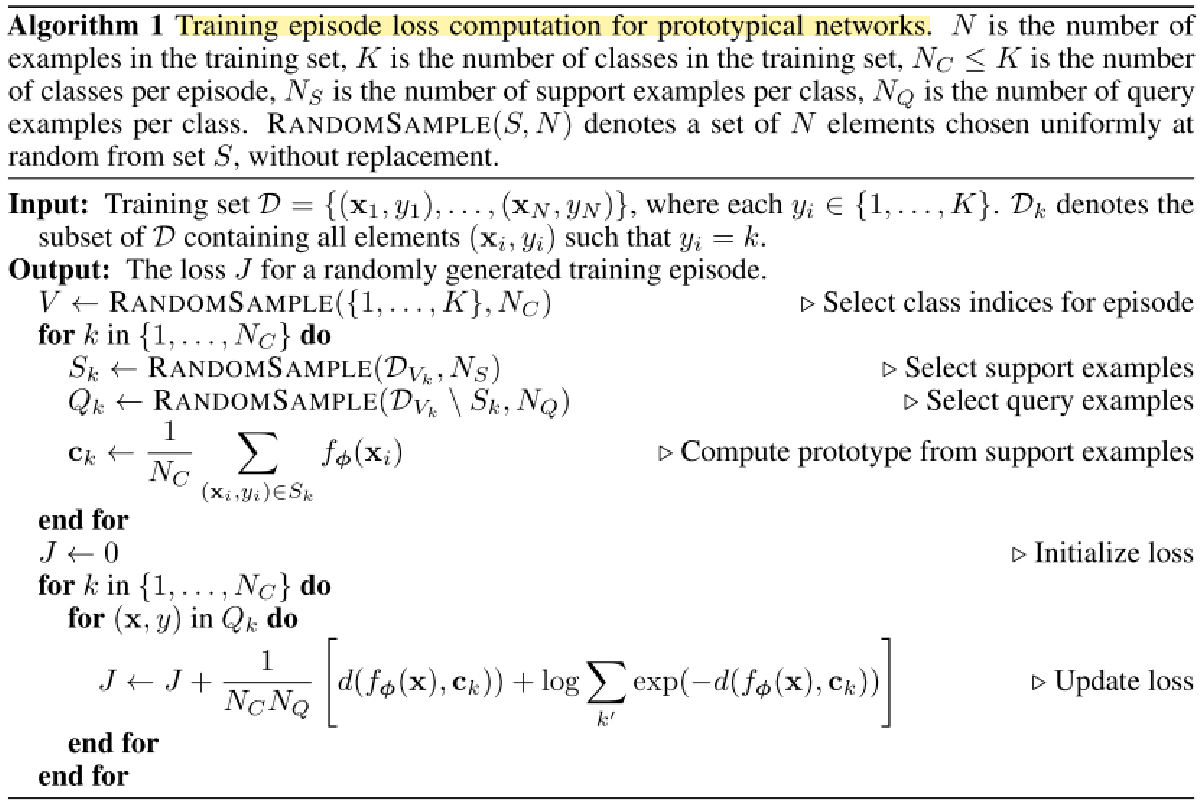
based on a softmax over distances to the prototypes in the embedding space:

$p_\phi(y=k|x)=\exp(-d(f_\phi(x),c_k))\sum_k \exp(-d(f_\phi(x),c_k))$

Learning proceeds by minimising the negative log-probability $J(\phi)=-\log p_\phi(y=k|x)$

of the true class k via SGD. Training episodes are formed by randomly selecting a subset of classes from the training set, then choosing a subset of examples within each class to act as the *support* set and a subset of the remainder to server as *query* points. Pseudocode to compute the loss $J(\phi)$

for a training episode is provided below:



Algorithm 1 Training episode loss computation for prototypical networks. N is the number of examples in the training set, K is the number of classes in the training set, $N_C \leq K$ is the number of classes per episode, N_S is the number of support examples per class, N_Q is the number of query examples per class. RANDOMSAMPLE(S, N) denotes a set of N elements chosen uniformly at random from set S, without replacement.

```
Input: Training set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, where each $y_i \in \{1, \ldots, K\}$. $\mathcal{D}_k$ denotes the
   subset of $\mathcal{D}$ containing all elements $(\mathbf{x}_i, y_i)$ such that $y_i = k$.
Output: The loss $J$ for a randomly generated training episode.
  $V \leftarrow$ RANDOMSAMPLE$(\{1, \ldots, K\}, N_C)$                    ▷ Select class indices for episode
  for $k$ in $\{1, \ldots, N_C\}$ do
    $S_k \leftarrow$ RANDOMSAMPLE$(\mathcal{D}_{V_k}, N_S)$                      ▷ Select support examples
    $Q_k \leftarrow$ RANDOMSAMPLE$(\mathcal{D}_{V_k} \setminus S_k, N_Q)$                   ▷ Select query examples
    $\mathbf{c}_k \leftarrow \frac{1}{N_C} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_\phi(\mathbf{x}_i)$          ▷ Compute prototype from support examples
  end for
  $J \leftarrow 0$                                                        ▷ Initialize loss
  for $k$ in $\{1, \ldots, N_C\}$ do
    for $(\mathbf{x}, y)$ in $Q_k$ do
      $J \leftarrow J + \frac{1}{N_C N_Q}\left[d(f_\phi(\mathbf{x}), \mathbf{c}_k)) + \log \sum_{k'} \exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k))\right]$      ▷ Update loss
    end for
  end for
```

It feels quite similar to the 'episodic training' as in Matching Network and Relation Network

S. Ravi, et al., 2017, "Optimization as a Model for Few-Shot Learning"

- *Initialization-based Methods* ∋ *Learning an optimizer*

In this paper, the authors proposed an LSTM-based *meta-learner* optimizer that is trained to optimise a *learner* neural network classifier. The meta-learner captures both short-term knowledge within a task and long-term knowledge common among all the tasks. By using an objective that directly captures an optimization algorithm's ability to have good generalisation performance given only a set number of updates, the meta-learner model is trained to converge a learner classifier to a good solution quickly on each task. Additionally, the formulation of our meta-learner model allows it to learn a task-common initialization for the learner classifier, which captures fundamental knowledge shared among all the tasks.

In this paper, a configuration of the meta-learning dataset is similar to the prior papers:

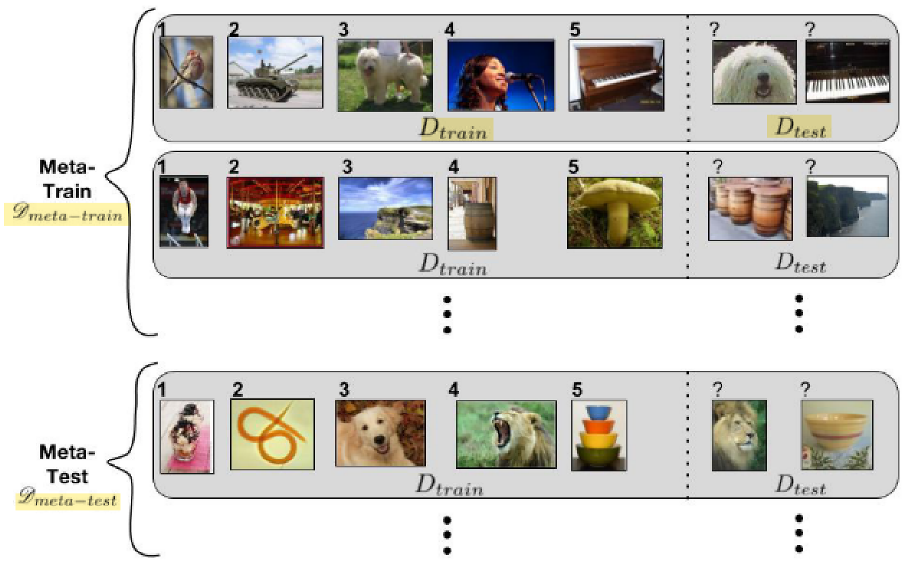


Figure 1: Example of meta-learning setup. The top represents the meta-training set $\mathscr{D}_{meta-train}$, where inside each gray box is a separate dataset that consists of the training set D_{train} (left side of dashed line) and the test set D_{test} (right side of dashed line). In this illustration, we are considering the 1-shot, 5-class classification task where for each dataset, we have one example from each of 5 classes (each given a label 1-5) in the training set and 2 examples for evaluation in the test set. The meta-test set $\mathscr{D}_{meta-test}$ is defined in the same way, but with a different set of datasets that cover classes not present in any of the datasets in $\mathscr{D}_{meta-train}$ (similarly, we additionally have a meta-validation set that is used to determine hyper-parameters).

The key idea of the LSTM-based meta learner is mimicking a process of gradient descent update with a *parametric learning rate (with the input gate)* and the *forget gate for the learned weight* introduced. The cell state in an LSTM is updated as:

$c_t=f_t\odot c_{t-1}+i_t\odot\tilde{c}_t(1)$

The Eq.(1)

is manipulated for the LSTM-based meta learner as in the following equation:

$c_t=f_t\odot\theta_{t-1}+i_t\odot(-\nabla_{\theta_{t-1}}L_t)$

where the f_t

and i_t denotes the forget gate and input gate, respectively, and i_t acts as a parametric learning rate. θ_t denotes the parameters of the learner. The meta-learner gets to determine optimal values of i_t and f_t through the course of the updates. The configurations of i_t and f_t

are as follows:

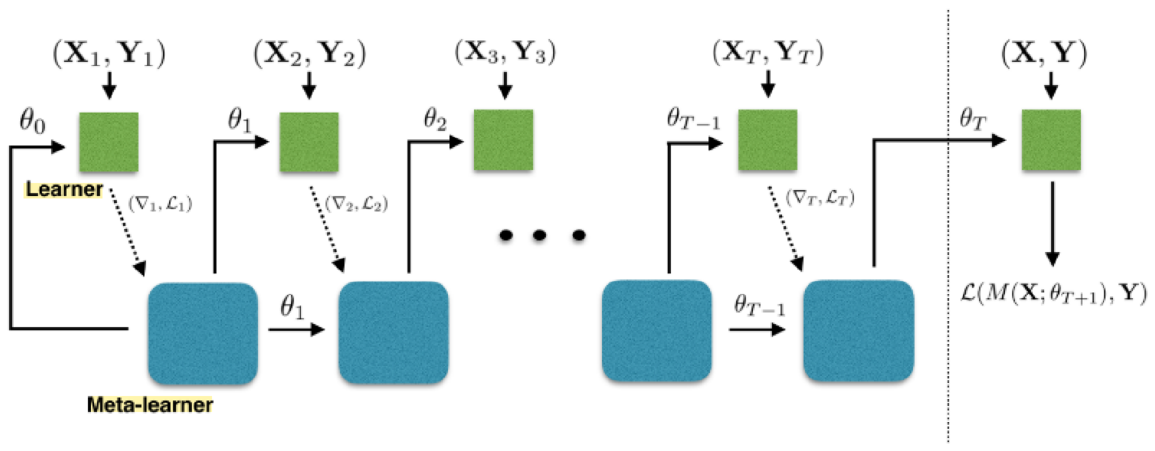
$i_t = \sigma(W_I \cdot [\nabla_{\theta_{t-1}} L_t, L_t, \theta_{t-1}, i_{t-1}] + b_I)$

$f_t = \sigma(W_F \cdot [\nabla_{\theta_{t-1}} L_t, L_t, \theta_{t-1}, f_{t-1}] + b_F)$

With i_t

, the meta-learner should be able to finely control the learning rate so as to train the learner quickly while avoiding divergence. As for f_t, what would justify shrinking the parameters of the learner and forgetting part of its previous value would be if the learner is currently in a bad local optima and needs a large change to escape. This would correspond to a situation where the loss is high but the gradient is close to zero. 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. 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.



$(\mathbf{X}_1, \mathbf{Y}_1)$
$(\mathbf{X}_2, \mathbf{Y}_2)$
$(\mathbf{X}_3, \mathbf{Y}_3)$
$(\mathbf{X}_T, \mathbf{Y}_T)$
$(\mathbf{X}, \mathbf{Y})$
θ_0
θ_1
θ_2
θ_{T-1}
θ_T
Learner
$(\nabla_1, \mathcal{L}_1)$
$(\nabla_2, \mathcal{L}_2)$
$(\nabla_T, \mathcal{L}_T)$
$\mathcal{L}(M(\mathbf{X}; \theta_{T+1}), \mathbf{Y})$
Meta-learner

Figure 2: Computational graph for the forward pass of the meta-learner. The dashed line divides examples from the training set D_{train} and test set D_{test}. Each $(\mathbf{X}_i, \mathbf{Y}_i)$ is the i^{th} batch from the training set whereas $(\mathbf{X}, \mathbf{Y})$ is all the elements from the test set. The dashed arrows indicate that we do not back-propagate through that step when training the meta-learner. We refer to the learner as M, where $M(\mathbf{X}; \theta)$ is the output of learner M using parameters θ for inputs $\mathbf{X}$. We also use ∇_t as a shorthand for $\nabla_{\theta_{t-1}} \mathcal{L}_t$.

F. Sung et al., 2018, "Learning to compare: Relation network for few-shot learning"

- *Metric-learning approach*

It is one of the metric-learning approaches to meta-learning. The authors further aim to learn a *transferrable deep metric* for the few-shot learning or zero-shot learning instead of pred-defined fixed metric such as Euclidean. This Relation Net is most related to the prototypical network and siamese network, but it is an upgraded version.

The episodic training (episode-based training) is explicitly utilised, which was first proposed in the Matching Network paper.

The model of the Relation Network is illustrated in the following figure:

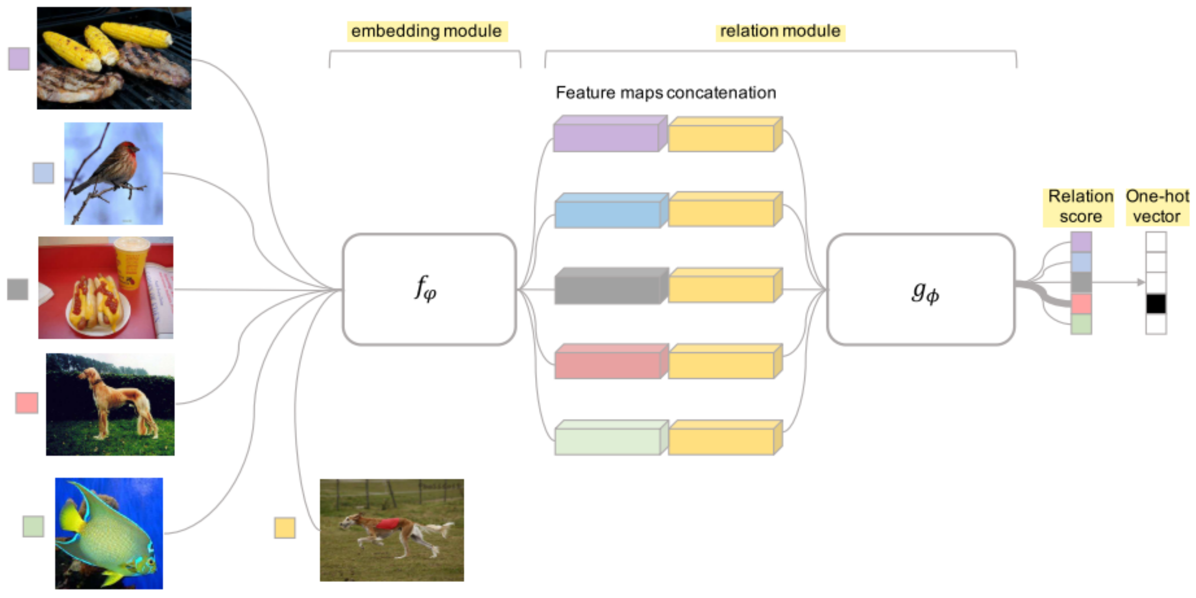


Figure 1: Relation Network architecture for a 5-way 1-shot problem with one query example.

One-shot

The relation network consists of two modules: an *embedding module* f_φ

and a *relation module* g_ϕ. Samples x_j in the query set Q, and samples x_i in the sample set S are fed through the embedding module f_φ, which produces feature maps $f_\varphi(x_i)$ and $f_\varphi(x_j)$. The feature maps $f_\varphi(x_i)$ and $f_\varphi(x_j)$ are combined with operator C. In this work, we assume $C(\cdot,\cdot)$

to be concatenation of feature maps in depth, although other choices are possible.

The combined feature map of the sample and query are fed into the relation module g_ϕ

, which eventually produces a scalar in range of 0 to 1 representing the similarity between x_i and x_j, which is called *relation score*, $r_{i,j}$. $r_{i,j}$ is for the relation between one query input x_j and training sample set examples x_i

.

$r_{i,j}=g_\phi(C(f_\varphi(x_i),f_\varphi(x_j)));i=1,2,\cdots,C$

K-shot

For K-shot where $K>1$

, we element-wise sum over the embedding module outputs of all samples from each training class to form this class' feature map.

Objective Function

We use the MSE loss to train the model, regressing the relation score $r_{i,j}$

to the ground truth: matched pairs have similarity 1 and the mismatched pair have similarity 0.

$$\varphi,\phi \leftarrow \operatorname{argmin}_{\varphi,\phi} \sum_{i=1}^{m} \sum_{j=1}^{n} (r_{i,j} - 1(y_i == y_j))^2$$

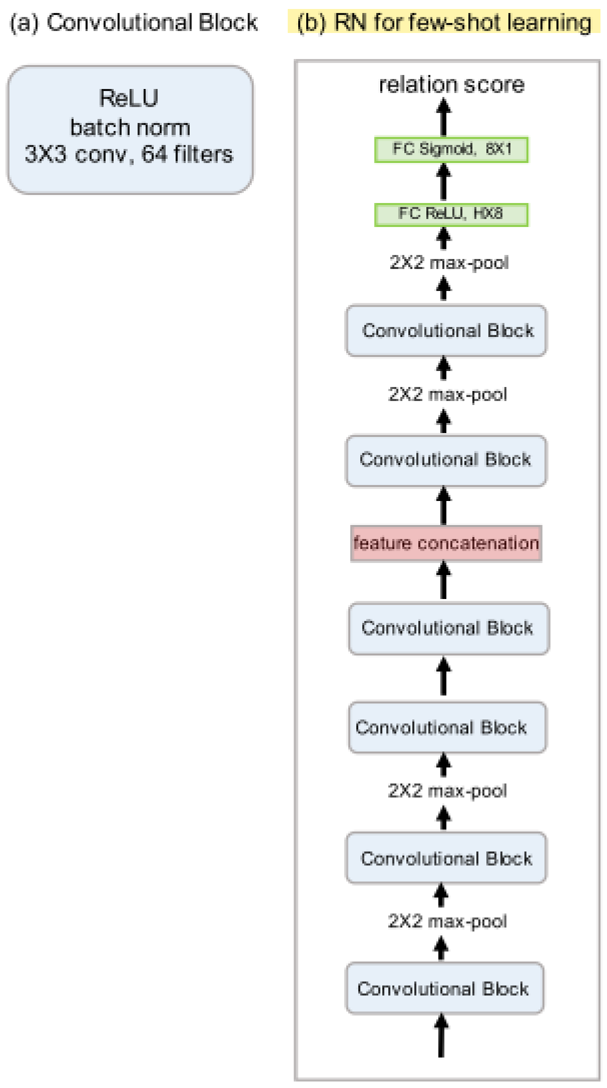


Figure 2: Relation Network architecture for few-shot learning (b) which is composed of elements including convolutional block (a).

C. Finn et al., 2017, "Model-agnostic meta-learning for fast adaptation of deep networks" (MAML)

- *Initialization-based Methods ∋ Good model initialization*

It finds good initial weights by the meta-learner such that the weights can be adapted to the new task easily, fast, and efficiently:

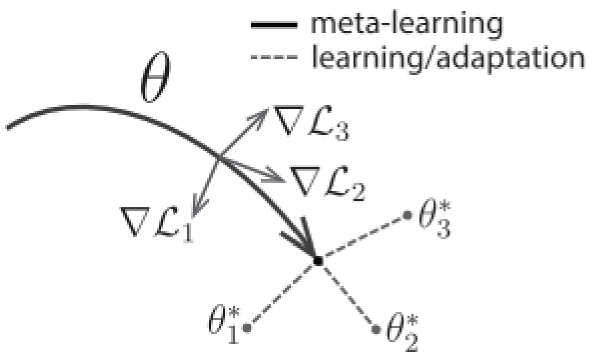
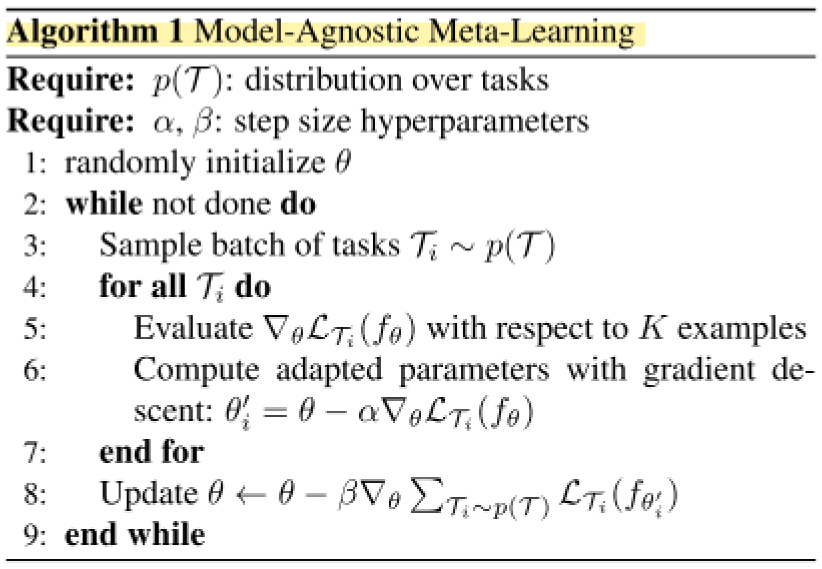


Figure 1. Diagram of our model-agnostic meta-learning algorithm (MAML), which optimizes for a representation θ that can quickly adapt to new tasks.



```
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
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
```

A. Nicohl, 2018, "On first-order meta-learning algorithms" (Reptile)

- *Initialization-based Methods ∋ Good model initialization*

This paper proposes *Reptile*, a simplified version of the MAML, by considering only first-order derivatives for the meta-learning updates. The results show that the performance of the Reptile is very competitive to the MAML although it's the simplified version.

Algorithm 1 Reptile (serial version)

```
Initialize $\phi$, the vector of initial parameters
for iteration $= 1, 2, \ldots$ do
    Sample task $\tau$, corresponding to loss $L_\tau$ on weight vectors $\widetilde{\phi}$
    Compute $\widetilde{\phi} = U^k_\tau(\phi)$, denoting $k$ steps of SGD or Adam
    Update $\phi \leftarrow \phi + \epsilon(\widetilde{\phi} - \phi)$
end for
```

A. Rusu et al., 2019, "Meta-Learning with Latent Embedding Optimization"

- *Initialization-based Methods* ∋ *Good model initialization*

This paper proposes *Latent Embedding Optimization (LEO)*. The LEO can be seen as an upgraded version of the MAML such that the LEO enables us to perform the MAML gradient-based adaptation steps in the learned, low-dimensional embedding space.

Algorithm 1 Latent Embedding Optimization

```
Require: Training meta-set $\mathcal{S}^{tr} \in \mathcal{T}$
Require: Learning rates $\alpha, \eta$
 1: Randomly initialize $\phi_e, \phi_r, \phi_d$
 2: Let $\phi = \{\phi_e, \phi_r, \phi_d, \alpha\}$
 3: while not converged do
 4:     for number of tasks in batch do
 5:         Sample task instance $\mathcal{T}_i \sim \mathcal{S}^{tr}$
 6:         Let $\left(\mathcal{D}^{tr}, \mathcal{D}^{val}\right) = \mathcal{T}_i$
 7:         Encode $\mathcal{D}^{tr}$ to $\mathbf{z}$ using $g_{\phi_e}$ and $g_{\phi_r}$
 8:         Decode $\mathbf{z}$ to initial params $\theta_i$ using $g_{\phi_d}$
 9:         Initialize $\mathbf{z}' = \mathbf{z}$, $\theta'_i = \theta_i$
10:         for number of adaptation steps do
11:             Compute training loss $\mathcal{L}^{tr}_{\mathcal{T}_i}\left(f_{\theta'_i}\right)$
12:             Perform gradient step w.r.t. $\mathbf{z}'$:
                $\mathbf{z}' \leftarrow \mathbf{z}' - \alpha \nabla_{\mathbf{z}'} \mathcal{L}^{tr}_{\mathcal{T}_i}\left(f_{\theta'_i}\right)$
13:             Decode $\mathbf{z}'$ to obtain $\theta'_i$ using $g_{\phi_d}$
14:         end for
15:         Compute validation loss $\mathcal{L}^{val}_{\mathcal{T}_i}\left(f_{\theta'_i}\right)$
16:     end for
17:     Perform gradient step w.r.t $\phi$:
        $\phi \leftarrow \phi - \eta \nabla_\phi \sum_{\mathcal{T}_i} \mathcal{L}^{val}_{\mathcal{T}_i}\left(f_{\theta'_i}\right)$
18: end while
```

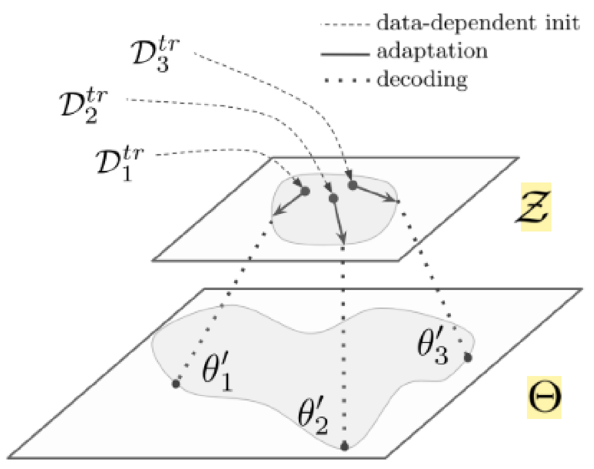


Figure 1: High-level intuition for LEO. While MAML operates directly in a high dimensional parameter space Θ, LEO performs meta-learning within a low-dimensional latent space $\mathcal{Z}$, from which the parameters are generated.

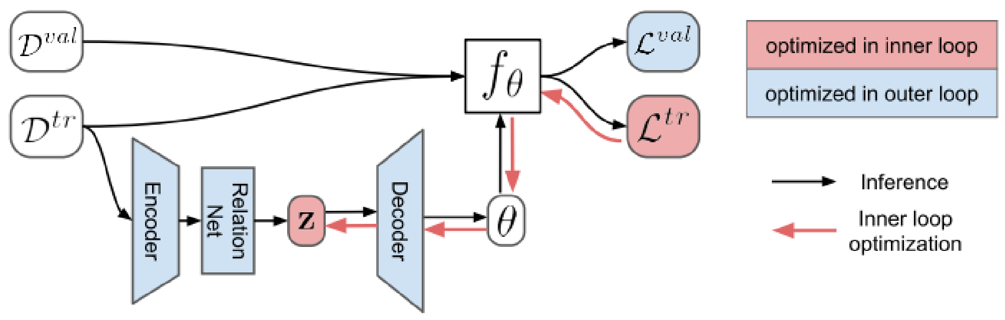


Figure 2: Overview of the architecture of LEO.

T. Munkhdalai et al., 2017, "Meta network"

The *MetaNet* consists of two main learning components: abase≤ar≠r

and amη≤ar≠r

equipped with an external memory. The weights of the meta-learner are updated using the external memory.

The paper is not so comprehensible. Some notations are kinda confusing. I found no significant idea from this paper.

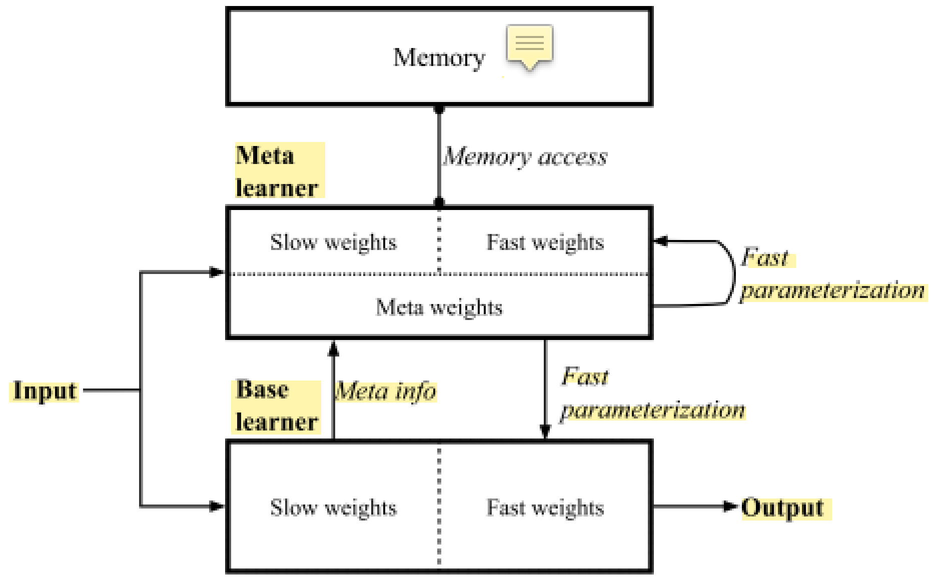


Figure 1. Overall architecture of Meta Networks. weight-update mechanism with the external memory