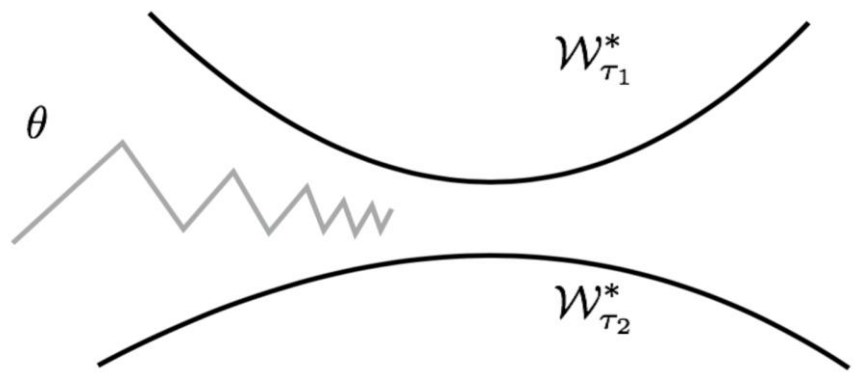
Meta Learning


$\mathcal{W}^*_{\tau_1}$
θ
$\mathcal{W}^*_{\tau_2}$

- Meta-learning is most commonly understood as learning to learn; the process of improving a learning algorithm over multiple learning tasks/episodes.
- In contrast, conventional ML improves model predictions over multiple data instances.

- During base learning, an inner (or lower/base) learning algorithm solves a task such as image classification, defined by a dataset and objective.
- During meta-learning, an outer (or upper/meta) algorithm updates the inner learning algorithm such that the model it learns improves an outer objective. For instance this objective could be generalization performance or learning speed of the inner algorithm.

Meta supervised learning

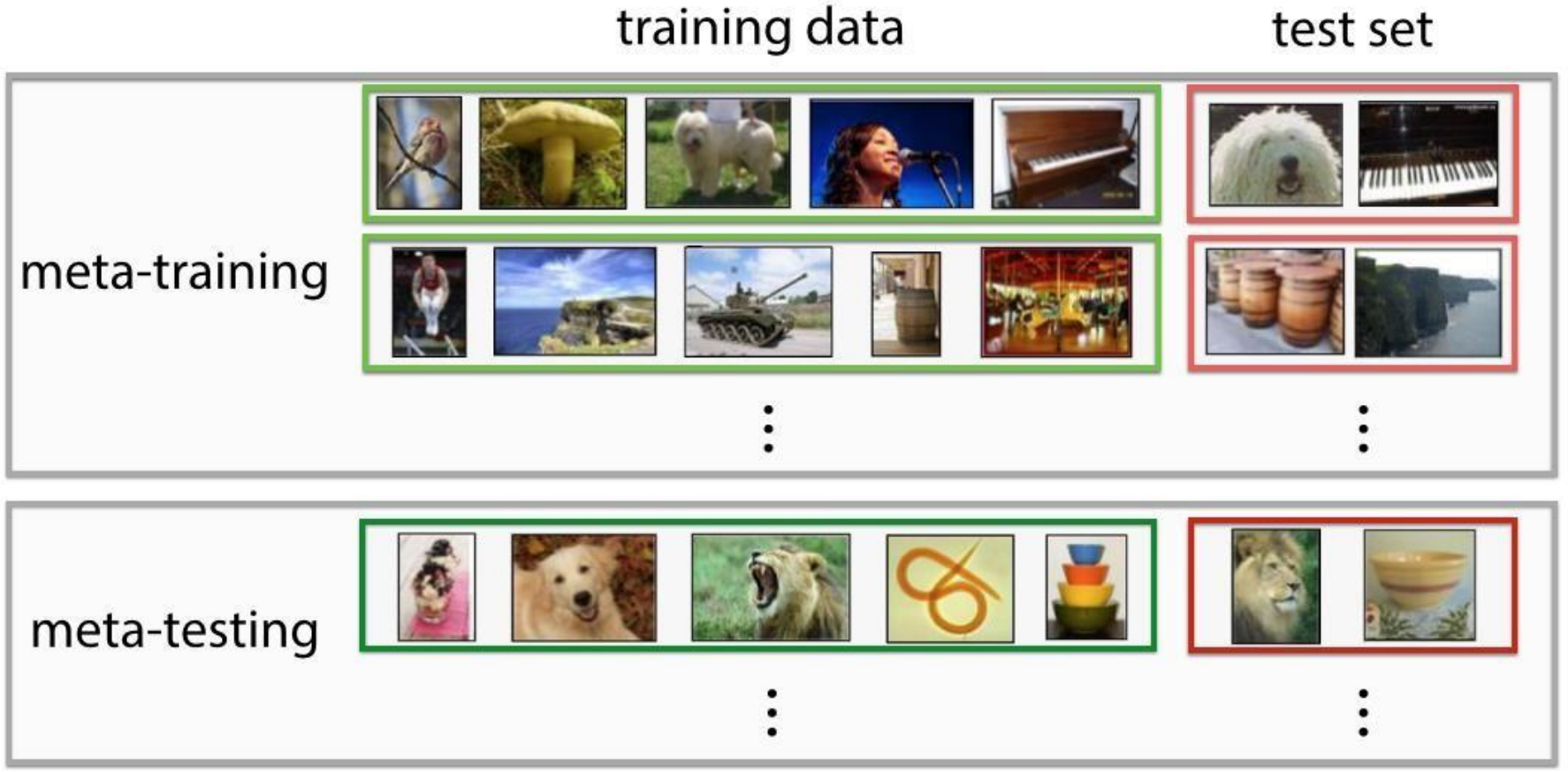


image credit: Ravi & Larochelle
'17

Setting

- In few-shot classification tasks, we have a meta-dataset D containing many classes C, where each class is itself a set of example instances {c1, c2, ..., cn}.
- If we are doing K-shot, N-way classification, then we sample tasks by selecting N classes from C and then selecting K + 1 examples for each class.
- We split these examples into a training set and a test set, where the test set contains a single (or more) example for each class.
- The model gets to see the entire training set, and then it must classify a randomly chosen sample from the test set.
- For example, if you trained a model for 5-shot, 5-way classification, then you would show it 25 examples (5 per class) and ask it to classify a 26th example.

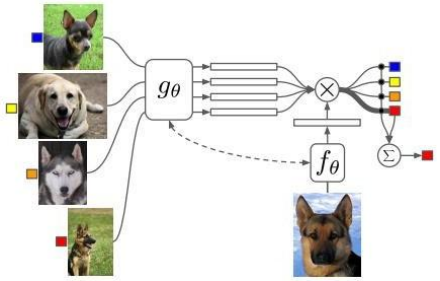
Testing

- A support set and query set is given. The classes are novel compared to meta-train set.

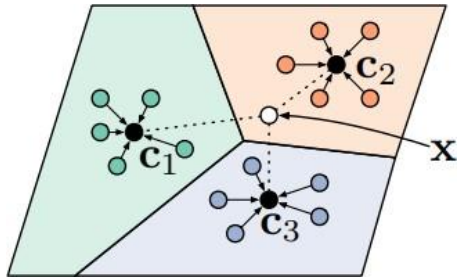
- With the support set only, we can in principle train a classifier to assign a class label y to each sample x in the test set.
- However, due to the lack of labelled samples in the sup- port set, the performance of such a classifier is usually not satisfactory.
- Therefore we aim to perform meta-learning 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.

Meta-learning methods

non-parametric meta-learning

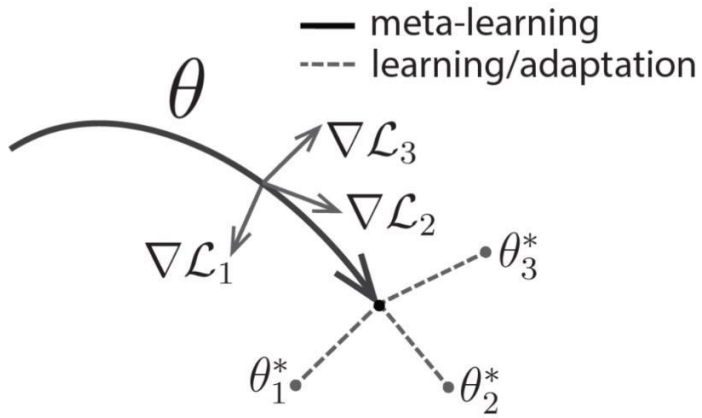


Vinyals et al. **Matching Networks for One Shot Learning**. 2017.



Snell et al. **Prototypical Networks for Few-shot Learning**. 2018.

gradient-based meta-learning



Finn et al. **Model-Agnostic Meta-Learning**. 2018.

MAML

Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks, ICM2017

Chelsea Finn, Pieter Abbeel, Sergey Levine

Meta-learning as an optimization problem

$$\theta^\star = \arg\min_\theta \sum_{i=1}^{n} \mathcal{L}(\phi_i, \mathcal{D}_i^{\text{ts}})$$

$$\text{where } \phi_i = f_\theta(\mathcal{D}_i^{\text{tr}})$$

$$\text{what if } f_\theta(\mathcal{D}_i^{\text{tr}}) \text{ is just a } \textit{finetuning} \text{ algorithm?}$$

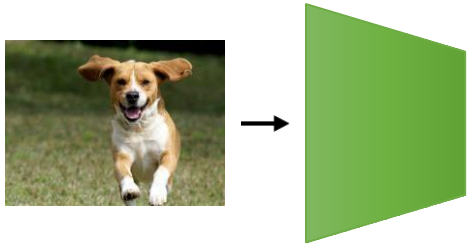
$$f_\theta(\mathcal{D}_i^{\text{tr}}) = \theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i^{\text{tr}})$$

(could take a few gradient steps in general)

This can be trained the same way as any other neural network, by implementing gradient descent as a computation graph and then running backpropagation *through* gradient descent!

Finn, Abbeel, Levine. **Model-Agnostic Meta-Learning for Fast Adaptation of Deep**

MAML in pictures



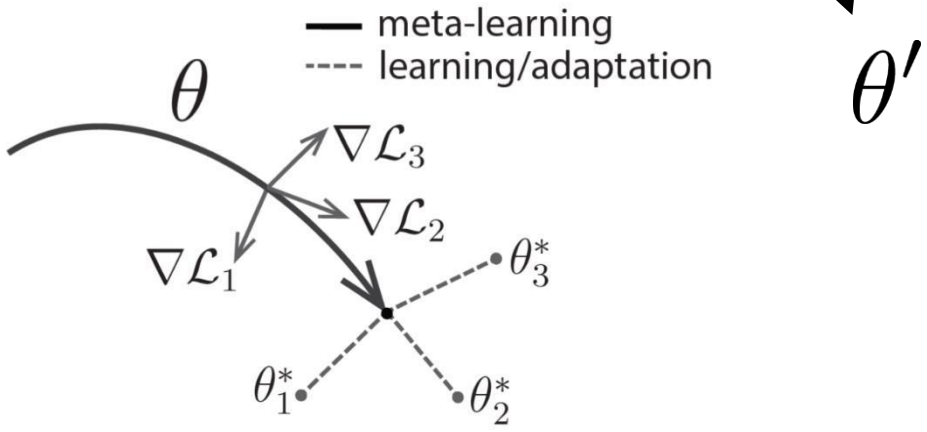
[Label]

θ

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}^{\text{tr}})$$

$$\theta \leftarrow \theta - \beta \sum_i \nabla_\theta \mathcal{L}(\theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i^{\text{tr}}), \mathcal{D}_i^{\text{ts}})$$

θ'



What did we just do??

supervised learning: $f(x) \rightarrow y$

supervised meta-learning: $f(\mathcal{D}^{\text{tr}}, x) \rightarrow y$

model-agnostic meta-learning: $f_{\text{MAML}}(\mathcal{D}^{\text{tr}}, x) \rightarrow y$

$$f_{\text{MAML}}(\mathcal{D}^{\text{tr}}, x) = f_{\theta'}(x)$$

$$\theta' = \theta - \alpha \sum_{(x,y) \in \mathcal{D}^{\text{tr}}} \nabla_\theta \mathcal{L}(f_\theta(x), y)$$

Just another computation graph…

Can implement with any autodiff package (e.g., TensorFlow)

But has favorable inductive bias…

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 $\mathcal{T}_i \sim p(\mathcal{T})$

Say class 1, 2, 3 are one task, and
Class 4, 5, 6 are 2nd,
7, 8, 9, 10 are third and so on

Sample tasks 1, 2, and 3

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 $\mathcal{T}_i$

5-way, 5-shot
K = 5, tasks can be 5.

Sample from training data

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)

5-way, 5-shot
K = 5, tasks can be 5.

Run sgd for each task.

Initialize model and copy the parameters such that
We have as many models as tasks.

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

5-way, 5-shot
K = 5, tasks can be 5.

Sample from test set and compute loss using θ_i'

5-way, 5-shot
K = 5, tasks can be 5.

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**

First Order MAML

- $\theta = \theta_{\text{meta}}$

Inner loop for a task

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

First Order MAML

- $\theta = \theta_{\text{meta}}$

Outer loop update

$$\theta = \theta - \beta g$$

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$$

$$\begin{aligned}
g &= \nabla_\theta \mathcal{L}^{(1)}(\theta_k) \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot (\nabla_{\theta_{k-1}} \theta_k) \ldots (\nabla_{\theta_0} \theta_1) \cdot (\nabla_\theta \theta_0) \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \left(\prod_{i=1}^{k} \nabla_{\theta_{i-1}} \theta_i \right) \cdot I \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \prod_{i=1}^{k} \nabla_{\theta_{i-1}} (\theta_{i-1} - \alpha \nabla_\theta \mathcal{L}^{(0)}(\theta_{i-1})) \\
&= \nabla_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \prod_{i=1}^{k} (I - \alpha \nabla_{\theta_{i-1}} (\nabla_\theta \mathcal{L}^{(0)}(\theta_{i-1})))
\end{aligned}$$

Treated as identity in FOMAML

Reptile, OpenAI

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**

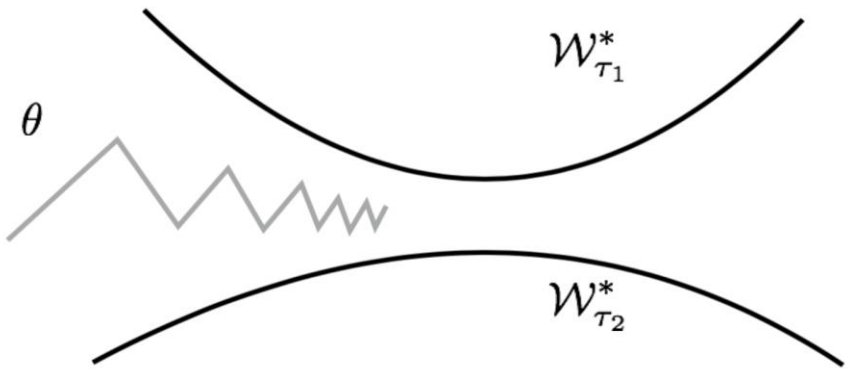
Algorithm 2 Reptile, batched version

Initialize θ
for iteration $= 1, 2, \ldots$ **do**
  Sample tasks $\tau_1, \tau_2, \ldots, \tau_n$
  for $i = 1, 2, \ldots, n$ **do**
    Compute $W_i = \mathrm{SGD}(L_{\tau_i}, \theta, k)$
  end for
  Update $\theta \leftarrow \theta + \beta \frac{1}{n} \sum_{i=1}^{n} (W_i - \theta)$
end for

$$\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$$

Reptile

- Assuming that a task T has a manifold of solution, then distance of meta parameter wrt task parameters



$$\begin{aligned}\nabla_\theta[\frac{1}{2}\mathrm{dist}(\theta, \mathcal{W}^*_{\tau_i})^2] &= \nabla_\theta[\frac{1}{2}\mathrm{dist}(\theta, W^*_{\tau_i}(\theta))^2] \\ &= \nabla_\theta[\frac{1}{2}(\theta - W^*_{\tau_i}(\theta))^2] \\ &= \theta - W^*_{\tau_i}(\theta)\end{aligned}$$

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

- In addition to the above setup, we also experimented with the transductive setting, where the model classifies the entire test set at once.
- In our transductive experiments, information was shared between the test samples via batch normalization.
- In our non-transductive experiments, batch normalization statistics were computed using all of the training samples and a single test sample.

Omniglot

Algorithm	1-shot 5-way	5-shot 5-way	1-shot 20-way	5-shot 20-way
MAML + Transduction	$98.7 \pm 0.4\%$	$99.9 \pm 0.1\%$	$95.8 \pm 0.3\%$	$98.9 \pm 0.2\%$
1^{st}-order MAML + Transduction	$98.3 \pm 0.5\%$	$99.2 \pm 0.2\%$	$89.4 \pm 0.5\%$	$97.9 \pm 0.1\%$
Reptile	$95.32 \pm 0.05\%$	$98.87 \pm 0.02\%$	$88.27 \pm 0.30\%$	$97.07 \pm 0.12\%$
Reptile + Transduction	$97.97 \pm 0.08\%$	$99.47 \pm 0.04\%$	$89.36 \pm 0.20\%$	$97.47 \pm 0.10\%$

```
import torch

class MAML:
    def __init__(self, model, lr_inner=0.01, lr_outer=0.001, num_inner_updates=1):
        self.model = model
        self.lr_inner = lr_inner  # learning rate for inner loop
        self.lr_outer = lr_outer  # learning rate for outer loop
        self.num_inner_updates = num_inner_updates  # number of updates for inner loop
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=self.lr_outer)

    def inner_update(self, x, y):
        # Compute loss with respect to task
        loss = self.model.loss(x, y)

        # Compute gradients
        grad = torch.autograd.grad(loss, self.model.parameters())

        # Update model parameters using gradients
        fast_weights = list(map(lambda p: p[1] - self.lr_inner * p[0], zip(grad, self.model.parameters())))

        return fast_weights
```

```
def outer_update(self, x, y, x_val, y_val):
    # Initialize fast weights with model parameters
    fast_weights = self.model.parameters()

    # Perform inner loop updates
    for _ in range(self.num_inner_updates):
        fast_weights = self.inner_update(x, y)

    # Compute validation loss using fast weights
    self.model.copy_weights(fast_weights)
    loss_val = self.model.loss(x_val, y_val)

    # Update model parameters using validation loss
    self.optimizer.zero_grad()
    loss_val.backward()
    self.optimizer.step()

def train(self, dataloader):
    for batch in dataloader:
        x, y, x_val, y_val = batch
        self.outer_update(x, y, x_val, y_val)
```

Architecture

- Consisting of a stack of modules, each of which is a 3×3 convolution with 64 filters followed by batch normalization, a Relu non-linearity and 2×2 max-pooling.

- resized all the images to 28×28 so that the resulting feature map is $1 \times 1 \times 64$.

- A fully connected layer followed by a softmax non-linearity is used to define the Baseline Classifier.

Non-parametric

- Matching Net, ProtoNet

Matching Networks for One Shot Learning, NIPS16

Oriol Vinyals
Google DeepMind
vinyals@google.com

Charles Blundell
Google DeepMind
cblundell@google.com

Timothy Lillicrap
Google DeepMind
countzero@google.com

Koray Kavukcuoglu
Google DeepMind
korayk@google.com

Daan Wierstra
Google DeepMind
wierstra@google.com

Matching networks

Vinyals et al. 2016

- Classifier: probability distribution based on similarity between x_{test} and $x_i \in D_{train}$

$$p(y|x_{test}, D_{train}) = \sum_{i=1}^{k} a(x_{test}, x_i) y_i$$

- Similarity: attention kernel based on cosine distance between embedded data points

$$a(x, x_i) = softmax(cos(f(x), g(x_i)))$$

$$a(\hat{x}, x_i) = e^{c(f(\hat{x}), g(x_i))} / \sum_{j=1}^{k} e^{c(f(\hat{x}), g(x_j))}$$

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(y, \hat{y})$$

- Simple embedding: $\theta = \theta'$, so $g_\theta = f_{\theta'}$
- Contextual embedding: g is a bidirectional LSTM and f is an attention LSTM

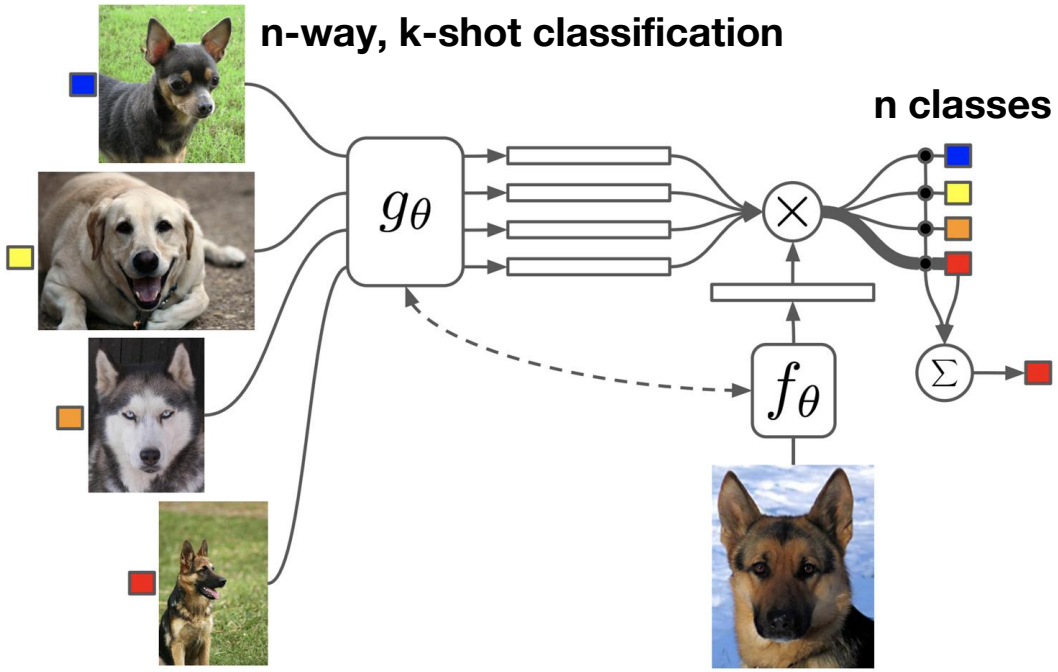


Figure source: Vinyals et al. 2016

The key point is that when trained, Matching Networks are able to produce sensible test labels for unobserved classes *without any changes to the network*. More precisely, we wish to map from a (small) support set of k examples of image-label pairs $S = \{(x_i, y_i)\}_{i=1}^{k}$ to a classifier $c_S(\hat{x})$ which, given a test example $\hat{x}$, defines a probability distribution over outputs $\hat{y}$. We define the mapping $S \rightarrow c_S(\hat{x})$ to be $P(\hat{y}|\hat{x}, S)$ where P is parameterised by a neural network. Thus, when given a

$$\theta = \arg\max_{\theta} E_{L\sim T}\left[E_{S\sim L, B\sim L}\left[\sum_{(x,y)\in B} \log P_\theta\left(y|x, S\right)\right]\right]$$

This could be read as min CE loss

P(y|x,S) is the prediction and y' would be the true label

y'^{T} log P(y|x,S)

Experimental set-up

- Omniglot dataset consists of 1623 characters from 50 different alphabets.
- Each of these was hand drawn by 20 different people.
- The large number of classes (characters) with relatively few data per class (20), makes this an ideal data set for testing small-scale one-shot classification.

- The N-way Omniglot task setup is as follows: pick N unseen character classes, independent of alphabet, as L.
- Provide the model with one drawing of each of the N characters as S $\sim$ L and a batch B $\sim$ L.
- 1200 characters for training, and the remaining character classes for evaluation

$$\theta = \arg\max_{\theta} E_{L\sim T}\left[E_{S\sim L, B\sim L}\left[\sum_{(x,y)\in B} \log P_\theta\left(y|x, S\right)\right]\right]$$

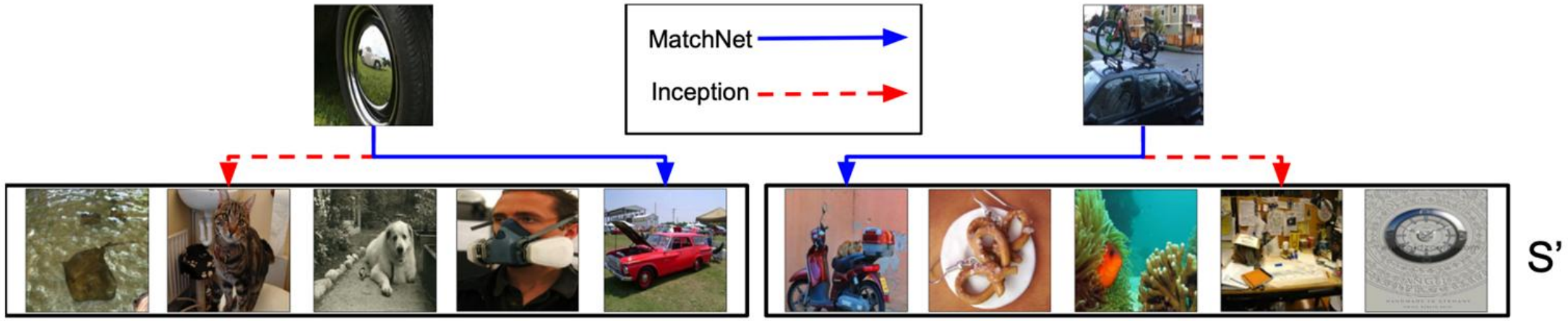
Note: Meta-test is always on novel classes

Model	Matching Fn	Fine Tune	5-way Acc		20-way Acc	
			1-shot	5-shot	1-shot	5-shot
PIXELS	Cosine	N	41.7%	63.2%	26.7%	42.6%
BASELINE CLASSIFIER	Cosine	N	80.0%	95.0%	69.5%	89.1%
BASELINE CLASSIFIER	Cosine	Y	82.3%	98.4%	70.6%	92.0%
BASELINE CLASSIFIER	Softmax	Y	86.0%	97.6%	72.9%	92.3%
MANN (NO CONV) [21]	Cosine	N	82.8%	94.9%	–	–
CONVOLUTIONAL SIAMESE NET [11]	Cosine	N	96.7%	98.4%	88.0%	96.5%
CONVOLUTIONAL SIAMESE NET [11]	Cosine	Y	97.3%	98.4%	88.1%	97.0%
MATCHING NETS (OURS)	Cosine	N	**98.1%**	**98.9%**	**93.8%**	98.5%
MATCHING NETS (OURS)	Cosine	Y	97.9%	98.7%	93.5%	**98.7%**

Table 1: Results on the Omniglot dataset.

Training

- The baseline classifier was trained to classify an image into one of the original classes present in the training data set, but excluding the N classes so as not to give it an unfair advantage (i.e., trained to classify classes in $\not\models \mathrm{L}'$).
- We then took this network and used the features from the last layer (before the softmax) for nearest neighbour matching, a strategy commonly used in computer vision [3] which has achieved excellent results across many tasks.
- We also tried further fine tuning the features using only the support set S' sampled from L'. This yields massive overfitting, but given that our networks are highly regularized, can yield extra gains. Note that, even when fine tuning, the setup is still one-shot, as only a single example per class from L' is used.



Example of two 5-way problem instance on ImageNet. The images in the set S′contain
classes never seen during training. Our model makes far less mistakes than the Inception baseline.

- Prototypical Networks for Few-shot Learning, NIPS17

Jake Snell
University of Toronto*

Kevin Swersky
Twitter

Richard S. Zemel
University of Toronto, Vector Institute

Prototypical networks

Snell et al. 2017

- Use an embedding function f_θ to encode each data point
- Define a prototype v_c for every class c based on the examples of that class $D_{train}^{(y)}$

$$\mathbf{v}_c = \frac{1}{|D_{train}^{(y)}|} \sum_{(x_i, y_i) \in D_{train}^{(y)}} f_\theta(x_i)$$

- Class distribution for input x is based on inverse distance between x and prototypes

$$p(y = c|x_{test}) = softmax(-d_\phi(f_\theta(x), v_c))$$

- Distance function can be any differentiable distance
  - E.g. squared Euclidean
- Loss function to learn the embedding:

$$\mathcal{L}(\theta) = -log p_\theta(y = c|x)$$

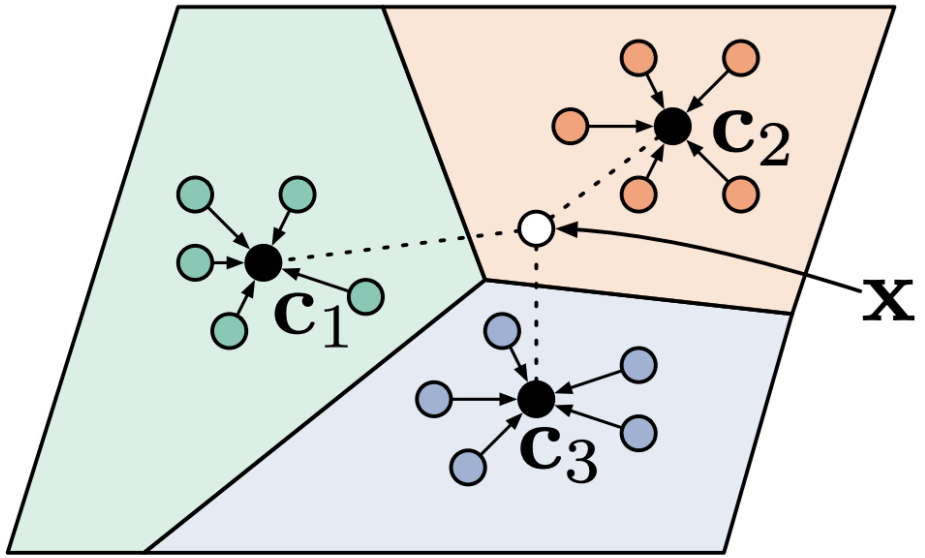


Figure source: Snell et al. 2017

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. $\textsc{RandomSample}(S, N)$ 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.

$V \leftarrow \textsc{RandomSample}(\{1, \ldots, K\}, N_C)$ ▷ Select class indices for episode

for k in $\{1, \ldots, N_C\}$ **do**

$S_k \leftarrow \textsc{RandomSample}(\mathcal{D}_{V_k}, N_S)$ ▷ Select support examples

$Q_k \leftarrow \textsc{RandomSample}(\mathcal{D}_{V_k} \setminus S_k, N_Q)$ ▷ Select query examples

$$\mathbf{c}_k \leftarrow \frac{1}{N_C} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_{\boldsymbol{\phi}}(\mathbf{x}_i)$$ ▷ 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**

$$J \leftarrow J + \frac{1}{N_C N_Q} \left[d(f_{\boldsymbol{\phi}}(\mathbf{x}), \mathbf{c}_k)) + \log \sum_{k'} \exp(-d(f_{\boldsymbol{\phi}}(\mathbf{x}), \mathbf{c}_k)) \right]$$ ▷ Update loss

end for

end for

- In Pytorch, we can pass the negative Euclidean distances as logits and one-hot encoded class labels to cross entropy loss.

Table 1: Few-shot classification accuracies on Omniglot.

Model	Dist.	Fine Tune	5-way Acc.		20-way Acc.	
			1-shot	5-shot	1-shot	5-shot
MATCHING NETWORKS [29]	Cosine	N	98.1%	98.9%	93.8%	98.5%
MATCHING NETWORKS [29]	Cosine	Y	97.9%	98.7%	93.5%	98.7%
NEURAL STATISTICIAN [6]	-	N	98.1%	99.5%	93.2%	98.1%
PROTOTYPICAL NETWORKS (OURS)	Euclid.	N	**98.8%**	**99.7%**	**96.0%**	**98.9%**

- Learning to Compare: Relation Network for Few-Shot Learning, CVPR18

Flood Sung Yongxin Yang[3] Li Zhang[2] Tao Xiang[1] Philip H.S. Torr[2] Timothy M. Hospedales[3]
[1]Queen Mary University of London [2]University of Oxford [3]The University of Edinburgh

Relation networks

Sung et al. 2018

- Similar to matching networks, but with a trainable similarity metric
  - Learns a non-linear relationship between the data points
- Learn an embedding network f_θ and a relation network $g_{\theta'}$

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

- The relationship/similarity between a pair of inputs:

$$r_{ij} = g_{\theta'}(concat(f_\theta(x_i), f_\theta(x_j)))$$

- Predictions based on the examples most related to x_{test}
- More expressive power, often beats other metric learners

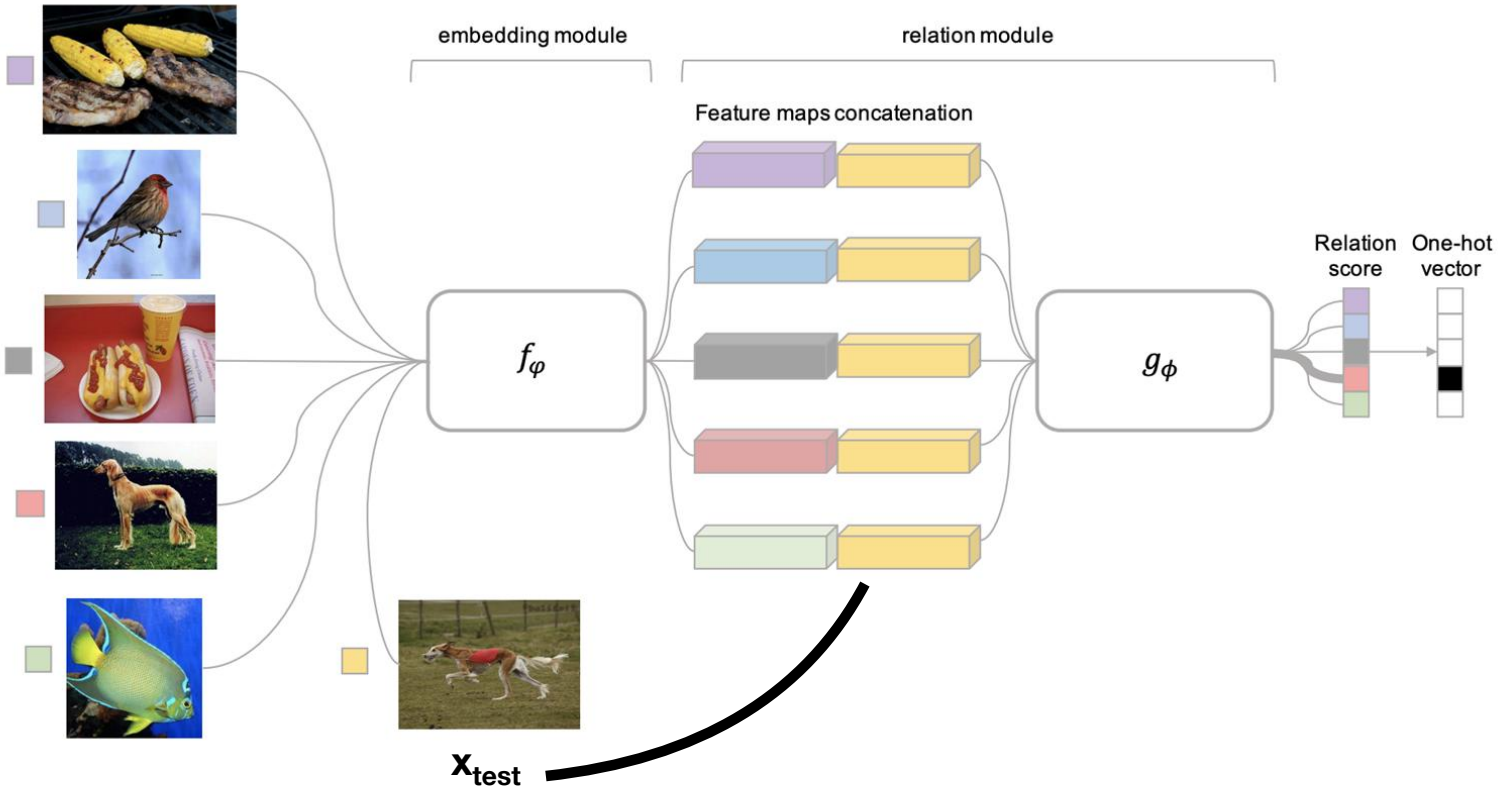
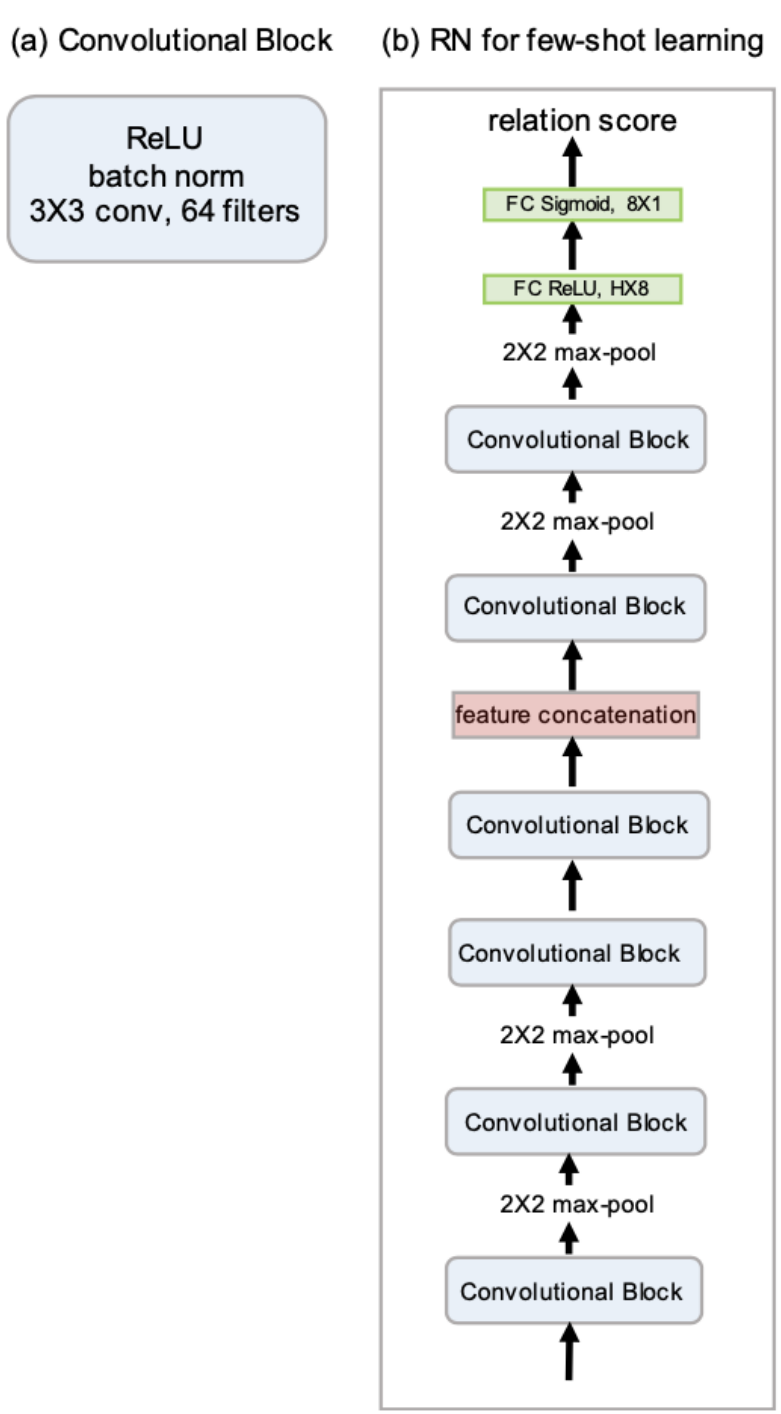


Figure source: Sung et al. 2018

An effective way to exploit the training set is to mimic the few-shot learning setting via *episode* based training, as proposed in [39]. 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 $\mathcal{S} = \{(x_i, y_i)\}_{i=1}^{m}$ ($m = K \times C$), as well as a fraction of the remainder of those C classes' samples to serve as the *query* set $\mathcal{Q} = \{(x_j, y_j)\}_{j=1}^{n}$. This sam-

- For K-shot where $\mathrm{K} > 1$, element-wise sum over the embedding module outputs of all samples from each training class is taken to form this class' feature map.
- This pooled class-level feature map is combined with the query image feature map as above.
- Thus, the number of relation scores for one query is always C in both one-shot or few- shot setting.

Model


(a) Convolutional Block
ReLU
batch norm
3X3 conv, 64 filters
(b) RN for few-shot learning
relation score
FC Sigmoid, 8X1
FC ReLU, HX8
2X2 max-pool
Convolutional Block
2X2 max-pool
Convolutional Block
feature concatenation
Convolutional Block
Convolutional Block
2X2 max-pool
Convolutional Block
2X2 max-pool
Convolutional Block

Omniglot

Model	Fine Tune	5-way Acc.		20-way Acc.	
		1-shot	5-shot	1-shot	5-shot
MANN [32]	N	82.8%	94.9%	-	-
CONVOLUTIONAL SIAMESE NETS [20]	N	96.7%	98.4%	88.0%	96.5%
CONVOLUTIONAL SIAMESE NETS [20]	Y	97.3%	98.4%	88.1%	97.0%
MATCHING NETS [39]	N	98.1%	98.9%	93.8%	98.5%
MATCHING NETS [39]	Y	97.9%	98.7%	93.5%	98.7%
SIAMESE NETS WITH MEMORY [18]	N	98.4%	99.6%	95.0%	98.6%
NEURAL STATISTICIAN [8]	N	98.1%	99.5%	93.2%	98.1%
META NETS [27]	N	99.0%	-	97.0%	-
PROTOTYPICAL NETS [36]	N	98.8%	99.7%	96.0%	98.9%
MAML [10]	Y	98.7 ± 0.4%	**99.9 ± 0.1%**	95.8 ± 0.3%	98.9 ± 0.2%
RELATION NET	N	**99.6 ± 0.2%**	**99.8± 0.1%**	**97.6 ± 0.2%**	**99.1± 0.1%**