Machine Learning (2023) 112:4635–4662
https://doi.org/10.1007/s10994-023-06394-x

Are LSTMs good few-shot learners?

Mike Huisman[1] · **Thomas M. Moerland**[1] · **Aske Plaat**[1] · **Jan N. van Rijn**[1]

Received: 8 February 2023 / Revised: 4 July 2023 / Accepted: 16 August 2023 /
Published online: 7 September 2023

Abstract
Deep learning requires large amounts of data to learn new tasks well, limiting its applicability to domains where such data is available. Meta-learning overcomes this limitation by learning how to learn. Hochreiter et al. (International conference on artificial neural networks, Springer, 2001) showed that an LSTM trained with backpropagation across different tasks is capable of meta-learning. Despite promising results of this approach on small problems, and more recently, also on reinforcement learning problems, the approach has received little attention in the supervised few-shot learning setting. We revisit this approach and test it on modern few-shot learning benchmarks. We find that LSTM, surprisingly, outperform the popular meta-learning technique MAML on a simple few-shot sine wave regression benchmark, but that LSTM, expectedly, fall short on more complex few-shot image classification benchmarks. We identify two potential causes and propose a new method called *Outer Product LSTM (OP-LSTM)* that resolves these issues and displays substantial performance gains over the plain LSTM. Compared to popular meta-learning baselines, OP-LSTM yields competitive performance on within-domain few-shot image classification, and performs better in cross-domain settings by 0.5–1.9% in accuracy score. While these results alone do not set a new state-of-the-art, the advances of OP-LSTM are orthogonal to other advances in the field of meta-learning, yield new insights in how LSTM work in image classification, allowing for a whole range of new research directions. For reproducibility purposes, we publish all our research code publicly.

Keywords Meta-learning · Few-shot learning · Deep learning · Transfer learning

Editors: Fabio Vitale, Tania Cerquitelli, Marcello Restelli, Charalampos Tsourakakis.

✉ Mike Huisman
m.huisman@liacs.leidenuniv.nl

Thomas M. Moerland
t.m.moerland@liacs.leidenuniv.nl

Aske Plaat
a.plaat@liacs.leidenuniv.nl

Jan N. van Rijn
j.n.van.rijn@liacs.leidenuniv.nl

[1] Leiden Institute of Advanced Computer Science, Leiden University, Niels Bohrweg 1, 2333CA Leiden, The Netherlands

1 Introduction

Deep neural networks have demonstrated human or even super-human performance on various tasks in different areas (Krizhevsky et al., 2012; He et al., 2015; Mnih et al., 2015; Silver et al., 2016). However, they often fail to learn new tasks well from limited amounts of data (LeCun et al., 2015), limiting their applicability to domains where abundant data is available. *Meta-learning* (Brazdil et al., 2022; Huisman et al., 2021; Naik and Mammone, 1992; Schmidhuber, 1987; Thrun, 1998) is one approach to overcome this limitation. The idea is to learn an efficient learning algorithm over a large number of different tasks so that new tasks can be learned from a few data points. Meta-learning involves learning at two different levels: the *inner-level* learning algorithm produces a predictor for the given task at hand, whereas the *outer-level* learning algorithm is adjusted to improve the learning ability across tasks.

Hochreiter et al. (2001) and Younger et al. (2001) have shown that LSTMs trained with gradient descent are capable of meta-learning. At the inner level—when presented with a new task—the LSTM ingests training examples with corresponding ground-truth outputs and conditions its predictions for new inputs on the resulting hidden state (the general idea for using recurrent neural networks for meta-learning has been visualized in Fig. 1). The idea is that the training examples that are fed into the LSTM can be remembered or stored by the LSTM in its internal states, allowing predictions for new unseen inputs to be based on the training examples. This way, the LSTM can implement a learning algorithm in the recurrent dynamics, whilst the weights of the LSTM are kept frozen. During meta-training, the weights of the LSTM are only adjusted at the outer level (across tasks) by backpropagation, which corresponds to updating the inner-level learning program. By exposing the LSTM to different tasks which it cannot solve without learning, the LSTM is stimulated to learn tasks by ingesting the training examples which it is fed. The initial experiments of Hochreiter et al. (2001) and Younger et al. (2001) have shown promising results on simple and low-dimensional toy problems. Meta-learning with LSTMs has also been successfully extended to reinforcement

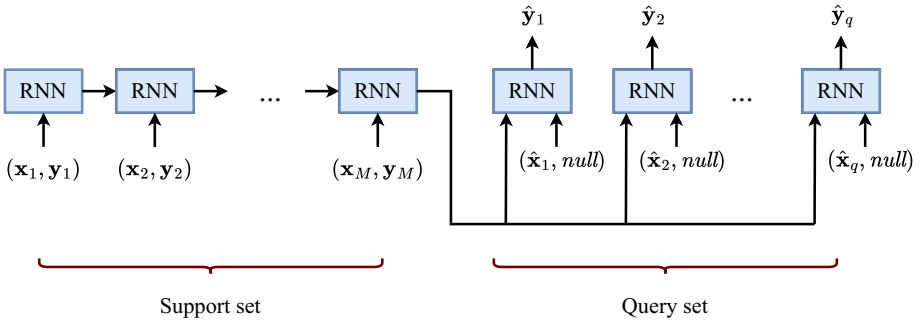


Fig. 1 The use of a recurrent neural network for few-shot learning. The support set $D^{tr}_{\mathcal{T}_j} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_M, \mathbf{y}_M)\}$ is fed as a sequence into the RNN. The predictions $\hat{\mathbf{y}}_j$ for new query points $\hat{\mathbf{x}}_j$ are conditioned on the resulting state. We note that feeding the tuples $(\mathbf{x}_i, \mathbf{y}_i)$ does not lead to the RNN directly outputting the presented labels (drastic overfitting) as the goal is to make predictions for query inputs, for which the ground-truth outputs are unknown. Alternatively, the support set could be fed into the RNN in a temporally offset manner (e.g., feed support tuples $(\mathbf{x}_i, \mathbf{y}_{i-1})$ into the RNN) as in Santoro et al. (2016) or in different ways (for example feed the error instead of the ground-truth target) (Hochreiter et al., 2001)

learning settings (Duan et al., 2016; Wang et al., 2016), and demonstrates promising learning speed on new tasks.

To the best of our knowledge, the LSTM approach has, in contrast, not been studied on more complex and modern supervised few-shot learning benchmarks by the research community, which has already shifted its attention to more developing new and more complex methods (Finn et al., 2017; Snell et al., 2017; Flennerhag et al., 2020; Park and Oliva, 2019). In our work, we revisit the idea of meta-learning with LSTMs and study the ability of the learning programs embedded in the weights of the LSTM to perform few-shot learning on modern benchmarks. We find that an LSTM outperforms the popular meta-learning technique MAML (Finn and Levine, 2017) on a simple few-shot sine wave regression benchmark, but that it falls short on more complex few-shot image classification benchmarks.

By studying the LSTM architecture in the context of meta-learning, we identify two potential causes for this underperformance, namely (1) the fact that it is not invariant to permutations of the training data and (2) that the input representation and learning procedures are intertwined. We propose a general solution to the first problem and propose a new meta-learning technique, *Outer Product LSTM (OP-LSTM)*, where we solve the second issue by learning the weight update rule for a base-learner network using an LSTM, in addition to good initialization parameters for the base-learner. This approach is similar to that of Ravi and Larochelle (2017), but differs in how the weights are updated with the LSTM and that in our approach, the LSTM does not use hand-crafted gradients as inputs in order to produce weight updates. Our experiments demonstrate that OP-LSTM yields substantial performance gains over the plain LSTM.

Our contributions are the following.

- We study the ability of a plain LSTM to perform few-shot learning on modern few-shot learning benchmarks and show that it yields surprisingly good performance on simple regression problems (outperforming MAML (Finn et al., 2017)), but is outperformed on more complex classification problems.
- We identify two problems with the plain LSTM for meta-learning, namely (1) the fact that it is not invariant to permutations of the training data and (2) that the input representation and learning procedures are intertwined, and propose solutions to overcome them by 1) an average pooling strategy and 2) decoupling the input representation from the learning procedure.
- We propose a novel LSTM architecture called *Outer Product LSTM (OP-LSTM)* that overcomes the limitations of the classical LSTM architecture and yields substantial performance gains on few-shot learning benchmarks.
- We discuss that OP-LSTM can approximate MAML (Finn et al., 2017) as well as Prototypical network (Snell et al., 2017) as it can learn to perform the same weight matrix updates. Since OP-LSTM does not update the biases, it can only approximate these two methods.

Compared to popular meta-learning baselines, including MAML (Finn et al., 2017), Prototypical network (Snell et al., 2017), and Warp-MAML (Flennerhag et al., 2020), OP-LSTM yields competitive performance on within-domain few-shot image classification, and outperforms them in cross-domain settings by 0.5–1.9% in raw accuracy score. While these results alone do not set a new state-of-the-art, the advances of OP-LSTM are orthogonal to other advances in the field of meta-learning, allowing for a whole range of new research directions, such as using OP-LSTM to update the weights in gradient-based meta-learning

techniques (Flennerhag et al., 2020; Park and Oliva, 2019; Lee and Choi, 2018) rather than regular gradient descent. For reproducibility and verifyability purposes, we make all our research code publicly available.[1]

2 Related work

Earlier work with LSTMs Meta-learning with recurrent neural networks was originally proposed by Hochreiter et al. (2001) and Younger et al. (2001). In their pioneering work, Hochreiter et al. (2001) also investigated other recurrent neural network architectures for the task of meta-learning, but it was observed that Elman networks and vanilla recurrent neural networks failed to meta-learn simple Boolean functions. Only the LSTM was found to be successful at learning simple functions. For this reason, we solely focus on LSTM in our work.

The idea of meta-learning with an LSTM at the data level has also been investigated and shown to achieve promising results in the context of reinforcement learning (Duan et al., 2016; Wang et al., 2016; Alver and Precup, 2021). In the supervised meta-learning community, however, the idea of meta-learning with an LSTM at the data level (Hochreiter et al., 2001; Younger et al., 2001) has not gained much attention. A possible explanation for this is that Santoro et al. (2016) compared their proposed memory-augmented neural network (MANN) to an LSTM and found that the latter was outperformed on few-shot Omniglot (Lake et al., 2015) classification. However, it was not reported how the hyperparameters of the LSTM were tuned and whether it was a single-layer LSTM or a multilayer LSTM. In addition, the LSTM was fed the input data as a sequence which is not permutation invariant, which can hinder its performance. We propose a permutation-invariant method of feeding training examples into recurrent neural networks and perform a detailed study of the performance of LSTM on few-shot learning benchmarks.

In concurrent work, Kirsch et al. (2022) investigates the ability of transformer architectures to implement learning algorithms, a baseline with a similar name as our proposed method was proposed ("Outer product LSTM"). We emphasize, however, that their method is different from ours (OP-LSTM) as it is a model-based approach that ingests the entire training set and query input into a slightly modified LSTM architecture (with an outer product update and inner product read-out) to make predictions, whereas in our OP-LSTM, the LSTM acts on a meta-level to update the weights of a base-learner network.

In concurrent works done by Kirsch et al. (2022) and Chan et al. (2022), the ability of the classical LSTM architecture to implement a learning algorithm was also investigated. They observed that it was unable to embed a learning algorithm into its recurrent dynamics on image classification tasks. However, the focus was not on few-shot learning, and no potential explanation for this phenomenon was given. In our work, we investigate the LSTM's ability to learn a learning algorithm in settings where only one or five examples are present per class, dive into the inner working mechanics to formulate two hypotheses as to why the LSTM architecture is incapable of learning a good learning algorithm, and as a result, propose OP-LSTM which overcomes the limitations and performs significantly better than the classical LSTM architecture.

*Different LSTM architectures for meta-learning*Santoro et al. (2016) used an LSTM as a read/writing mechanism to an external memory in their MANN technique. Kirsch and

[1] See: https://github.com/mikehuisman/lstm-fewshotlearning-oplstm.

Schmidhuber (2021) proposed to replace every weight in a neural network with a recurrent neural network that communicates through forward and backward messages. The system was shown able to learn backpropagation and can be used to improve upon it. Our proposed method OP-LSTM can also learn to implement backpropagation (see Sect. 6). Other works (Ravi and Larochelle, 2017; Andrychowicz et al., 2016) have also used an LSTM for meta-learning the weight update procedure. Instead of feeding the training examples into the LSTM, as done by the plain LSTM (Hochreiter et al., 2001; Younger et al., 2001), the LSTM was fed gradients so that it could propose weight updates for a separate base-learner network. Our proposed method OP-LSTM is similar to these two approaches that meta-learn the weight update rules as we use an LSTM to update the weights (2D hidden states) of a base-learner network. Note that this strategy thus also deviates from the plain LSTM approach, which is fed raw input data. In our approach, the LSTM acts on predictions and ground-truth targets or messages. In addition, we use a coordinate-wise architecture where the same LSTM is applied to different nodes in the network. A difference with other learning-to-optimize works (Ravi and Larochelle, 2017; Andrychowicz et al., 2016) is that we do not feed gradients into the LSTM and that we update the weights (2D hidden states) through outer product update rules.

3 Meta-learning with LSTM

In this section, we briefly review the LSTM architecture (Hochreiter and Schmidhuber, 1997), explain the idea of meta-learning with an LSTM through backpropagation as proposed by Hochreiter et al. (2001) and Younger et al. (2001), discuss two problems with this approach in the context of meta-learning and propose solutions to solve them.

Additionally, we propose solutions to this problem. We prove that a single-layer RNN followed by a linear layer is incapable of embedding a classification learning algorithm in its recurrent dynamics and show by example that an LSTM adding a single linear layer is sufficient to achieve this type of learning behavior in a simple setting.

3.1 LSTM architecture

LSTM (Hochreiter and Schmidhuber, 1997) is a recurrent neural network architecture suitable for processing sequences of data. The architecture of an LSTM cell is displayed in Fig. 2. It maintains an internal state and uses four gates to regulate the information flow within the network

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f), \tag{1}$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i), \tag{2}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o), \tag{3}$$

$$\bar{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c). \tag{4}$$

Here, $\theta = \{\mathbf{W}_f, \mathbf{W}_c, \mathbf{W}_i, \mathbf{W}_o, \mathbf{b}_f, \mathbf{b}_c, \mathbf{b}_i, \mathbf{b}_o\}$ are the parameters of the LSTM, $[\mathbf{a}, \mathbf{b}]$ represents the concatenation of $\mathbf{a}$ and $\mathbf{b}$, σ is the sigmoid function (applied element-wise) and $\mathbf{f}_t, \mathbf{i}_t, \mathbf{o}_t, \bar{\mathbf{c}}_t \in \mathbb{R}^{d_h}$ are the forget, input, output, and cell gates, respectively. These gates regulate the information flow within the network to produce the next cell and hidden states

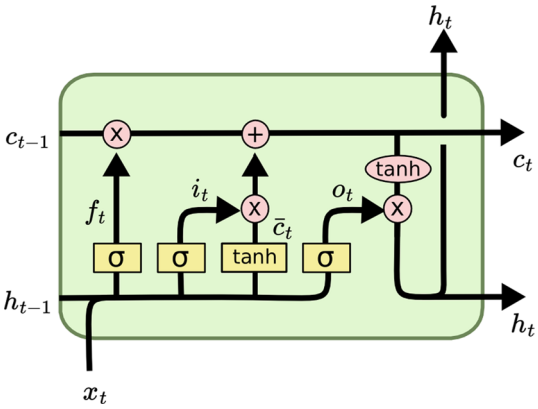


Fig. 2 The architecture of an LSTM cell. The LSTM maintains an inner cell state $\mathbf{c}_t$ and hidden state $\mathbf{h}_t$ over time that are updated with new incoming data $\mathbf{x}_t$. The forget $\mathbf{f}_t$, input $\mathbf{i}_t$, output $\mathbf{o}_t$, and cell $\bar{\mathbf{c}}_t$ gates regulate how these states are updated. Image adapted from Olah (2015)

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \bar{\mathbf{c}}_t, \tag{5}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \tag{6}$$

The hidden state and cell state are obtained by applying LSTM g_θ to inputs $(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1)$, i.e.,

$$[\mathbf{h}_t, \mathbf{c}_t] = g_\theta(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1) \tag{7}$$

$$= m_\theta(\mathbf{x}_t; \mathbf{h}_{t-1}, \mathbf{c}_{t-1}). \tag{8}$$

3.2 Meta-learning with LSTM

Hochreiter et al. (2001) and Younger et al. (2001) show that the LSTM can perform learning purely through unrolling its hidden state over time with fixed weights. When presented with a new task $\mathcal{T}_j$—denoting the concatenation of an input and its target as $\mathbf{x}'_t = (\mathbf{x}_t, \mathbf{y}_t)$—the support set $D^{tr}_{\mathcal{T}_j} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_M, \mathbf{y}_M)\} = \{\mathbf{x}'_1, \mathbf{x}'_2, \ldots, \mathbf{x}'_M\}$, is fed as a sequence, e.g., $(\mathbf{x}_1, null), (\mathbf{x}_2, \mathbf{y}_1), \ldots, (\mathbf{x}_M, \mathbf{y}_{M-1})$, into the LSTM to produce a hidden state $\mathbf{h}_M(D^{tr}_{\mathcal{T}_j})$. Predictions for unseen inputs (queries) $\hat{\mathbf{x}}$ are then conditioned on the hidden state $\mathbf{h}_M(D^{tr}_{\mathcal{T}_j})$ and cell state $\mathbf{c}_M(D^{tr}_{\mathcal{T}_j})$, where we have made it explicit that $\mathbf{h}_M$ and $\mathbf{c}_M$ are functions of the support data. More specifically, the hidden state of the query input $\hat{\mathbf{x}} = [\mathbf{x}, \mathbf{y}_M]$ is computed as $[\hat{\mathbf{h}}, \hat{\mathbf{c}}] = m_\theta(\hat{\mathbf{x}}; \mathbf{h}_M(D^{tr}_{\mathcal{T}_j}), \mathbf{c}_M(D^{tr}_{\mathcal{T}_j}))$, and this hidden state is used either directly for prediction or can be fed into a classifier function (which also uses fixed weights). Since the weights of the LSTM are fixed when presented with a new task, the learning takes place in the recurrent dynamics, and the hidden state $\mathbf{h}_M(D^{tr}_{\mathcal{T}_j})$ is responsible for guiding predictions on unseen inputs $\hat{\mathbf{x}}$. Note that there are different ways to feed the support data into the LSTM, as one can also use additional data such as the error on the previous input or feed the

current input together with its target tuples $(\mathbf{x}_t, \mathbf{y}_t)$ (as done in Fig. 1 and our implementation). We use the latter strategy in our experiments as we found it to be most effective.

This recurrent learning algorithm can be obtained by performing meta-training on various tasks which require the LSTM to perform learning through its recurrent dynamics. Given a task, we feed the training data into the LSTM, and then feed in query inputs to make predictions. The loss on these query predictions can be backpropagated through the LSTM to update the weights across different tasks. Note, however, that during the unrolling of the LSTM over the training data, the weights of the LSTM are held fixed. The weights are thus only updated across different tasks (not during adaptation to individual tasks) to improve the recurrent learning algorithm. By adjusting the weights of the LSTM using backpropagation across different tasks, we are essentially changing the learning program of the LSTM and hence performing meta-learning.

3.3 Problems with the classical LSTM architecture

The classical LSTM architecture suffers from two issues that may limit its ability to implement recurrent learning algorithms.

Non-temporal training data LSTMs work with sequences of data. When using an LSTM in the meta-learning context, the recurrent dynamics should implement a learning algorithm and process the support dataset. This support dataset $D^{tr}_{\mathcal{T}_j} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_M, \mathbf{y}_M)\} = \{\mathbf{x}'_1, \mathbf{x}'_2, \ldots, \mathbf{x}'_M\}$, however, is a *set* rather than a sequence. This means that we would want the hidden embedding after processing the support data to be invariant with respect to the order in which the examples are fed into the LSTM. Put more precisely, given any two permutations of the M training examples $\pi = (\pi_1, \pi_2, \ldots, \pi_M)$ and $\pi' = (\pi'_1, \pi'_2, \ldots, \pi'_M)$, we want to enforce

$$g_\theta(\mathbf{x}'_{\pi_1}, \mathbf{x}'_{\pi_2}, \ldots, \mathbf{x}'_{\pi_M}) = g_\theta(\mathbf{x}'_{\pi'_1}, \mathbf{x}'_{\pi'_2}, \ldots, \mathbf{x}'_{\pi'_M}), \tag{9}$$

where $\mathbf{x}'_{\pi_i}$ is the i-th input (possibly containing target or error information) under permutation π and $\mathbf{x}'_{\pi'_i}$ the input under permutation π'.

Intertwinement of embedding and learning In the LSTM approach proposed by Hochreiter et al. (2001) and Younger et al. (2001), the recurrent dynamics implement a learning algorithm. At the same time, however, the hidden state also serves as an input embedding. Thus, in this approach, the input embedding and learning procedures are intertwined. This may be problematic because a learning procedure may be highly complex and nonlinear, whilst the optimal input embedding may be simple and linear. For example, suppose that we feed convolutional features into a plain LSTM. Normally, we often compute predictions using a linear output layer. Thus, a simple single-layer LSTM may be the best in terms of input representation. However, the learning ability of a single-layer LSTM may be too limited, leading to bad performance. In other words, stacking multiple LSTM layers may be beneficial for finding a better learning algorithm, but the resulting input embedding may be too complex, which can lead to overfitting. On the other hand, a good but simple input embedding may overly restrict the search space of learning algorithms, resulting in a bad learning algorithm.

An LSTM with sufficiently large hidden dimensionality may be able to separate the learning from the input representation by using the first N dimensions of the hidden

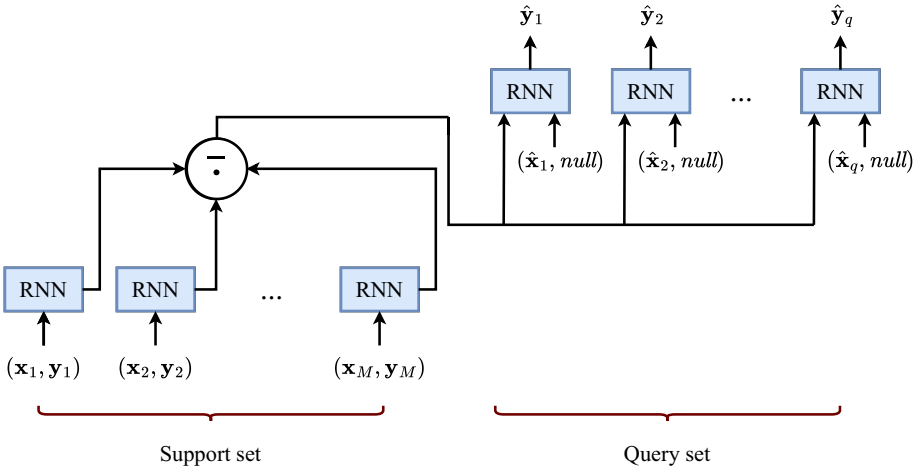


Fig. 3 Our proposed batch processing of the support data, resulting in a state that is permutation invariant. Every support example $(\mathbf{x}_i, \mathbf{y}_i)$ is processed in parallel, and the resulting hidden states are aggregated with mean-pooling (denoted by the symbol $\bar{\cdot}$). The predictions $\hat{\mathbf{y}}_j$ for new query points $\hat{x}_j$ are conditioned on the resulting permutation-invariant state. Note that the support data is only fed once into the RNN (a single time step t), although it is possible to make multiple passes over the data, by feeding the mean-pooled state into the RNN at the next time step

representations to perform learning and to preserve important information for the next time step, and using the remaining dimensions to represent the input. However, this poses a challenging optimization problem due to the risk of overfitting and the large number of parameters that would be needed.

3.4 Towards an improved architecture

These potential issues of the classical LSTM architecture inspire us to develop an architecture that is better suited for meta-learning.

Non-temporal data → average pooling In order to enforce invariance of the hidden state and cell state with respect to the order of the support data, we can *pool* the individual embeddings. That is, given an initial state of the LSTM $\mathbf{s}_t = [\mathbf{h}_t, \mathbf{c}_t]$, we update the state by processing the support data as a batch and by average pooling, i.e.,

$$\mathbf{s}_{t+1} = \left[\mathbf{h}_{t+1}, \mathbf{c}_{t+1}\right] = \frac{1}{M} \sum_{i=1}^{M} m_\theta(\mathbf{x}_i'; \mathbf{h}_t, \mathbf{c}_t). \tag{10}$$

Note that one time step now corresponds to processing the entire support dataset once, since $\mathbf{s}_{t+1}$ is a function thereof. Our proposed batch processing for a single time step (during which we ingest the support data) has been visualized in Fig. 3.

Intertwinement of embedding and learning The problems associated with the intertwinement of the embedding and learning procedures can be solved by decoupling them. In this way, we create two separate procedures: (1) the embedding procedure, and (2) the learning procedure. The embedding procedure can be implemented by a base-learner neural network, and the learning procedure by a meta-network that updates the weights of the base-learner network.

In the plain LSTM approach, where the learning procedure is intertwined with the input representation mechanism, predictions would be conditioned on the hidden state $\mathbf{h}(\mathbf{x}, \mathbf{h}(D^{tr}_{\mathcal{T}_j}), \mathbf{c}(D^{tr}_{\mathcal{T}_j}))$. Instead, we choose to use the inner product between the hidden state (acting as weight vector) and the embedding of current input $\mathbf{a}^{(L)}(\mathbf{x})$, i.e.,

$$\hat{y}(\mathbf{x}) = a^{(L)}(\mathbf{x}) = \underbrace{\mathbf{h}^{(L)}(D^{tr}_{\mathcal{T}_j})^T}_{\text{learning}} \underbrace{\mathbf{a}^{(L-1)}(\mathbf{x})}_{\text{embedding}}, \tag{11}$$

where $\mathbf{a}^{(L-1)}(\mathbf{x})$ is the representation of input $\mathbf{x}$ in layer $L-1$ of some base-learner network (consisting of L layers), whose weights are updated by a meta-network. We use the inner product to force interactions between the learning and embedding components, so that the predictions can not rely on either of the two separately. Note that by computing predictions in this way, we effectively decouple the learning algorithm implemented by hidden state dynamics from the input representation. A problem with this approach is that the output is a single scalar. In order to obtain an arbitrary output dimension $d_{out} > 1$, we should multiply the input representation $\mathbf{a}^{(L-1)}(\mathbf{x})$ with a matrix $\mathbf{H} \in \mathbb{R}^{d_{out} \times d_{in}}$, i.e., $\hat{\mathbf{y}}(\mathbf{x}) = \mathbf{H}^{(L)}\mathbf{a}^{(L-1)}(\mathbf{x})$. In order to obtain $\mathbf{H}^{(L)}$, one could use a separate LSTM with a hidden dimension of d_{in} per output dimension, but the number of required LSTMs would grow linearly with the output dimensionality. Instead, we use the outer product, which requires only one hidden vector of size d_{in} that can be outer-multiplied with a vector of size d_{out}. We detail the computation of 2D weight matrices $\mathbf{H}$ (hidden states) with the outer product rule in the next section.

Note that the 2D hidden state $\mathbf{H}$ can be seen as a weight matrix of a regular feed-forward neural network, which allows us to generalize this approach to networks with an arbitrary number of layers, where we have a 2D hidden state $\mathbf{H}^{(\ell)}$ for every layer $\ell \in \{1, 2, \ldots, L\}$ in a network with L layers. Our approach can then be seen as meta-learning an outer product weight update rule for the base-learner network such that it can quickly adapt to new tasks.

4 Outer product LSTM (OP-LSTM)

Here, we propose a new technique, called *Outer Product LSTM (OP-LSTM)*, based on the problems of the classical LSTM architecture for meta-learning and our suggested solutions. We begin by discussing the architecture, then cover the learning objective and algorithm, and end by studying the relationship between OP-LSTM and other methods.

4.1 The architecture

Since we can view the 2D hidden states $\mathbf{H}^{(\ell)}$ in OP-LSTM as weight matrices that act on the input, the OP-LSTM can be interpreted as a regular fully-connected neural network. The output of the OP-LSTM for a given input $\mathbf{x}$ is given by

$$f_\theta(\mathbf{x}, D^{tr}_{\mathcal{T}_j}, T) = \sigma^{(L)}(\mathbf{H}^{(L)}_T \mathbf{a}^{(L-1)}_T(\mathbf{x}) + \mathbf{b}^{(L)}), \tag{12}$$

where $D^{tr}_{\mathcal{T}_j}$ is the support dataset of the task, L the number of layers of the base-learner network, and T the number of time steps that the network unrolls (trains) over the entire support set. Here, $\sigma^{(L)}$ is the activation function used in layer L, $\mathbf{b}^{(L)}$ the bias vector in the output layer, and $\mathbf{a}^{(L-1)}_T(\mathbf{x})$ the input to layer L after making T passes over the support set and having received the query input.

Put more precisely, the activation in layer ℓ at time step t, as a function of an input $\mathbf{x}$, is denoted $\mathbf{a}^{(\ell)}_t(\mathbf{x})$ and defined as follows

$$\mathbf{a}^{(\ell)}_t(\mathbf{x}) = \begin{cases} \mathbf{x} & \text{if } \ell = 0 \text{ (input layer)}, \\ \sigma^{(\ell)}(\mathbf{H}^{(\ell)}_t \mathbf{a}^{(\ell-1)}_t(\mathbf{x}) + \mathbf{b}^{(\ell)}) & \text{otherwise.} \end{cases} \tag{13}$$

Note that this defines the forward dynamics of the architecture. Here, the $\mathbf{H}^{\ell}_t \in \mathbb{R}^{d^{(\ell)}_{out} \times d^{(\ell)}_{in}}$ is the 2D hidden state that is updated by pooling over the normalized 2D *outer product* hidden states $\mathbf{h}^{(\ell)}_{t+1}(\mathbf{x}'_i)\mathbf{a}^{(\ell-1)}_t(\mathbf{x}'_i)^T$ associated with individual training examples $\mathbf{x}'_i = (\mathbf{x}_i, \mathbf{y}_i)$, i.e.,

$$\mathbf{H}^{(\ell)}_{t+1} = \mathbf{H}^{(\ell)}_t + \frac{\gamma}{M} \sum_{i=1}^{M} \frac{\mathbf{h}^{(\ell)}_{t+1}(\mathbf{x}'_i)\mathbf{a}^{(\ell-1)}_t(\mathbf{x}_i)^T}{\|\mathbf{h}^{(\ell)}_{t+1}(\mathbf{x}'_i)\mathbf{a}^{(\ell-1)}_t(\mathbf{x}_i)^T\|_F}, \tag{14}$$

where γ is the step size of the updates and $\|\cdot\|_F$ is the Frobenius norm. We perform this normalization for numerical stability. Note that this update using average pooling ensures that the resulting hidden states $\mathbf{H}^{(\ell)}_t$ are invariant to permutations of the support data. Moreover, we observe that this equation defines the backward dynamics of the architecture (updating the 2D hidden states). However, this equation does not yet tell us how the hidden states $\mathbf{h}^{(\ell)}_{t+1}(\mathbf{x}'_i)$ are computed.

We use a coordinate-wise LSTM so that the same LSTM can be used in layers of arbitrary dimensions, in similar fashion as Ravi and Larochelle (2017); Andrychowicz et al. (2016). This means that we maintain a state $s^{(\ell)}_{t,j} = [h^{(\ell)}_{t,j}, c^{(\ell)}_{t,j}]$ for every individual node j in the state vector and every layer $\ell \in \{1, 2, \ldots, L\}$ over time steps t. In order to obtain the hidden state vector for a given layer ℓ and time step t, we simply concatenate the individual hidden states computed by the coordinate-wise LSTM, i.e., $\mathbf{h}^{(\ell)}_t = [h^{(\ell)}_{t,1}, h^{(\ell)}_{t,2}, \ldots, h^{(\ell)}_{t,d^{(\ell)}}]^T$, where $d^{(\ell)}$ is the number of neurons in layer ℓ. The LSTM weights to update these states are shared across all layers and nodes with the same activation function. For classification experiments, we often have two LSTMs: one for the final layer which uses a softmax activation function, and one for the body of the network, which uses the ReLU activation. This allows OP-LSTM to learn weight updates akin to gradient descent, where the backward computation is tied to each nonlinearity in the base-learner network (as this can not be done by a single LSTM, due to the different non-linearities of the softmax and the RELU

activation). We use pooling over the support data in order to update the states using a coordinate-wise approach, where every element of the hidden state $\mathbf{h}_t^{(\ell)}$ of a given layer ℓ is updated independently by a single LSTM.

In order to compute the next state $s_{t+1,j}^{(\ell)}$ of node j in layer ℓ, we need to have the previous state consisting of the previous hidden state $h_{t,j}^{(\ell)}$ and cell state $c_{t,j}^{(\ell)}$ of that node. Moreover, we need to feed the LSTM an input, which we define as $z_{t,j}^{(\ell)}$. In OP-LSTM, we define this input as

$$z_{t,j}^{(\ell)}(\mathbf{x}_i') = \begin{cases} [(\mathbf{a}_t^{(\ell)}(\mathbf{x}_i))_j, (\mathbf{y}_i)_j] & \text{if } \ell = L(\text{output layer}), \\ [(\mathbf{a}_t^{(\ell)}(\mathbf{x}_i))_j, ((\mathbf{H}_t^{(\ell+1)})^T \mathbf{h}_t^{(\ell+1)}(\mathbf{x}_i))_j] & \text{otherwise.} \end{cases} \tag{15}$$

Note that for the output layer L, the input to the LSTM corresponds to the current prediction and the ground-truth output, which share the same dimensionality. For the earlier layers in the network, we do not have access to the ground-truth signals. Instead, we view the hidden state of the output layer LSTM as errors and propagate them backward through the 2D hidden states $\mathbf{H}_t^{(\ell+1)}$, hence the expression $(\mathbf{H}_t^{(\ell+1)})^T \mathbf{h}_t^{(\ell+1)}(\mathbf{x}_i)$ for earlier layers. We note that this is akin to backpropagation, where error messages $\delta^{(\ell+1)}$ are passed backward through the weights of the network.

Given an input $\mathbf{x}_i'$, the next state $s_{t+1,j}^{(\ell)}$ can then be computed by applying the LSTM m_θ to the input $z_{t,j}^{(\ell)}(\mathbf{x}_i')$, conditioned on the previous hidden state $h_{t,j}^{(\ell)}$ and cell state $c_{t,j}^{(\ell)}$.

$$s_{t+1,j}^{(\ell)}(\mathbf{x}_i') = [h_{t+1,j}^{(\ell)}(\mathbf{x}_i'), c_{t+1,j}^{(\ell)}(\mathbf{x}_i')] = m_\theta(z_{t,j}^{(\ell)}(\mathbf{x}_i'); h_{t,j}^{(\ell)}, c_{t,j}^{(\ell)}), \tag{16}$$

where $z_{t,j}^{(\ell)}(\mathbf{x}_i')$ is the input to the LSTM used to update the state. These individual states are averaged over all training inputs to obtain

$$s_{t+1,j}^{(\ell)} = [h_{t+1,j}^{(\ell)}, c_{t+1,j}^{(\ell)}] = \frac{1}{M} \sum_{i=1}^{M} s_{t+1,j}^{(\ell)}(\mathbf{x}_i'). \tag{17}$$

Note that we can obtain a state vector, hidden vector, and cell state vector, by concatenation, i.e., $\mathbf{s}_{t+1,j}^{(\ell)}(\mathbf{x}_i') = [s_{t+1,1}^{(\ell)}(\mathbf{x}_i'), s_{t+1,2}^{(\ell)}(\mathbf{x}_i'), \ldots, s_{t+1,d^{(\ell)}}^{(\ell)}(\mathbf{x}_i')]$, $\mathbf{h}_{t+1,j}^{(\ell)}(\mathbf{x}_i') = [h_{t+1,1}^{(\ell)}(\mathbf{x}_i'), h_{t+1,2}^{(\ell)}(\mathbf{x}_i'), \ldots, h_{t+1,d^{(\ell)}}^{(\ell)}(\mathbf{x}_i')]$, and $\mathbf{c}_{t+1,j}^{(\ell)}(\mathbf{x}_i') = [c_{t+1,1}^{(\ell)}(\mathbf{x}_i'), c_{t+1,2}^{(\ell)}(\mathbf{x}_i'), \ldots, c_{t+1,d^{(\ell)}}^{(\ell)}(\mathbf{x}_i')]$.

4.2 The algorithm

OP-LSTM is trained to minimize the expected loss on the query sets conditioned on the support sets, where the expectation is with respect to a distribution of tasks. Put more precisely, we wish to minimize $\mathop{\mathbb{E}}_{\mathcal{T}_j \sim p(\mathcal{T})} \left[\mathcal{L}_{D_{\mathcal{T}_j}^{te}}(\Theta) \right]$, where $\Theta = \{\boldsymbol{\theta}, \mathbf{H}_0^{(1)}, \mathbf{H}_0^{(2)}, \ldots, \mathbf{H}_0^{(L)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \ldots, \mathbf{b}^{(L)}\}$. This objective can be approximated by sampling batches of tasks, updating the weights using the learned outer product rules, and evaluating the loss on the query sets. Across tasks, we update Θ using gradient descent. In practice, we use the cross-entropy loss for classification tasks and the MSE loss for regression tasks.

The pseudocode for OP-LSTM is displayed in Algorithm 1. First, we randomly initialize the initial 2D hidden states $\mathbf{H}_0^{(\ell)}$ and the LSTM parameters $\boldsymbol{\theta}$. We group these parameters as $\Theta = \{\boldsymbol{\theta}, \mathbf{H}_0^{(1)}, \mathbf{H}_0^{(2)}, \ldots, \mathbf{H}_0^{(L)}\}$, which will be meta-learned across different tasks. Given a task

$\mathcal{T}_j$, we make T updates on the entire support set $D^{tr}_{\mathcal{T}_j}$ by processing the examples individually, updating the 2D hidden states $\mathbf{H}^{(\ell)}_t$, and computing the new hidden states of the coordinate-wise LSTM for every layer $\mathbf{s}^{(\ell)}_t$. After having made T updates on the support data, we compute the loss of the model on the query set $D^{te}_{\mathcal{T}_j}$. The gradient of this loss with respect to all parameters Θ is added to the gradient buffer. Once a batch of tasks B has been processed in this way, we perform a gradient update on Θ and repeat this process until convergence or a maximum number of iterations has been reached.

Algorithm 1 Meta-learning with outer product LSTM (OP-LSTM)

Randomly initialize $\mathbf{H}^{(\ell)}_0$ and biases $\mathbf{b}^{(\ell)}$ for all $1 \leq \ell \leq L$
Randomly initialize LSTM parameters θ, set $\mathbf{h}^{(\ell)}_o = \mathbf{0}$, $\mathbf{c}^{(\ell)}_0 = \mathbf{0}$
repeat
  Initialize gradient buffer $\zeta = \mathbf{0}$
  Sample batch of J tasks $B = \{\mathcal{T}_j \backsim p(\mathcal{T})\}^J_{j=1}$
  for $\mathcal{T}_j = (D^{tr}_{\mathcal{T}_j}, D^{te}_{\mathcal{T}_j})$ in B **do**
    set $\mathbf{h}^{(\ell)}_o = \mathbf{0}$, $\mathbf{c}^{(\ell)}_0 = \mathbf{0}$ for all $1 \leq \ell \leq L$
    for $t = 1, ..., T$ **do**
      for $\mathbf{x}'_i = (\mathbf{x}_i, \mathbf{y}_i) \in D^{tr}_{\mathcal{T}_j}$ **do**
        Compute predictions $\mathbf{a}^{(L)}_{t-1}(\mathbf{x}_i)$ (Equation 13)
        Compute $\mathbf{z}^{(\ell)}_t(\mathbf{x}'_i)$ and $\mathbf{h}^{(\ell)}_t(\mathbf{x}_i)$ for $1 \leq \ell \leq L$ with backward message passing (see Equation 15 and Equation 16)
        Update $\mathbf{H}^{(\ell)}_t$ for $1 \leq \ell \leq L$ (Equation 14)
      end for
      Compute $\mathbf{s}^{(\ell)}_t$ for $1 \leq \ell \leq L$ (Equation 17) through concatenation
    end for
    Compute query predictions $\mathcal{L}_{D^{te}_{\mathcal{T}_j}}(\{\mathbf{H}^{(\ell)}_T\}^L_{\ell=1})$
    Update gradient buffer $\zeta = \zeta + \frac{1}{J}\nabla_\Theta \mathcal{L}_{D^{te}_{\mathcal{T}_j}}(\{\mathbf{H}^{(\ell)}_T\}^L_{\ell=1})$
  end for
  Update $\mathbf{\Theta} = \Theta - \beta\zeta$
until convergence

5 Experiments

In this section, we aim to answer the following research questions:

- How do the performance and training stability of a plain LSTM compare when processing the support data as a sequence versus as a set with average pooling? (See Sect. 5.1).
- How well does the plain LSTM perform at few-shot sine wave regression and within- and cross-domain image classification problems compared with popular meta-learning

methods such as MAML (Finn et al., 2017) and Prototypical network (Snell et al., 2017)? (See Sects. 5.2 and 5.3).
- Does OP-LSTM yield a performance improvement over the simple LSTM and the related approaches MAML and Prototypical network in few-shot sine wave regression and within- and cross-domain image classification problems? (See Sects. 5.2 and 5.3).
- How does OP-LSTM adjust the weights of the base-learner network? (See Sect. 5.4).

For our experiments, we use few-shot sine wave regression (Finn et al., 2017) as an illustrative task, and popular few-shot image classification benchmarks, namely Omniglot (Lake et al., 2015), miniImageNet (Ravi and Larochelle, 2017; Vinyals et al., 2016), and CUB (Wah et al., 2011). We use MAML (Finn et al., 2017), prototypical network (Snell et al., 2017), SAP (Huisman et al., 2023) and Warp-MAML (Flennerhag et al., 2020) as baselines. The former two are popular meta-learning methods and can both be approximated by the OP-LSTM (see Sect. 6), allowing us to investigate the benefit of OP-LSTM's expressive power. The last two baselines are used to investigate how OP-LSTM compares to state-of-the-art gradient-based meta-learning methods in terms of performance, although it has to be noted that that OP-LSTM is orthogonal to that method, in the sense that OP-LSTM could be used on top of Warp-MAML. However, this is a nontrivial extension and we leave this for future work. We run every technique on a single GPU (PNY GeForce RTX 2080TI) with a computation budget of 2 days (for detailed running times, please see Sect. 1). Each experiment is performed with 3 different random seeds, where the random seed affects the random weight initialization of the neural networks as well as the used training tasks, validation tasks, and testing tasks. Below, we describe the different experimental settings that we use. Note that we do not aim to achieve state-of-the-art performance, but rather investigate whether the plain LSTM is a competitive method for few-shot learning on modern benchmarks and whether OP-LSTM yields improvements over the plain LSTM, MAML, and Prototypical network.

Sine wave regression This toy problem was originally proposed by Finn et al. (2017) to study meta-learning methods. In this setting, every task $\mathcal{T}_j$ corresponds to a sine wave $s_j = A_j \cdot \sin(x - p_j)$, where A_j and p_j are the amplitude and phase of the task, sampled uniformly at random from the intervals $[0.1, 5.0]$ and $[0, \pi]$, respectively. The goal is to predict for a given task the correct output y given an input x after training on the support set, consisting of k examples. The performance of learning is measured in the query set, consisting of 50 input–output. For the plain LSTM approach, we use a multi-layer LSTM trained with Backpropagation through Time (BPTT) using Adam (Kingma and Ba, 2015). During meta-training, the LSTM is shown 70,000 training tasks. Every 2500 tasks, we perform meta-validation on 1000 tasks, and after having selected the best validated model, we evaluate the performance on 2000 meta-test tasks.

Few-shot image classification In case of few-shot image classification, all methods are trained for 80,000 episodes on training tasks and we perform meta-validation every 2500 episodes. The best learner is then evaluated on 600 hold-out test tasks, each task having a number of examples per class in the support set as indicated by the experiment (ranging from 1 to 10) as well as a query set of 15 examples per class. We repeat every experiment 3 times with different random seeds, meaning the that weight initializations and tasks are different across runs, although the class splits for sampling training/validation/testing tasks are kept fixed. For the Omniglot image classification dataset, we used a fully-connected neural network as base-learner for MAML and OP-LSTM, following Santoro et al. (2016) and Finn et al. (2017). The network consists of 4 fully-connected blocks with dimensions 256-128-64-64. Every block consists of a linear layer, followed by BatchNorm and ReLU

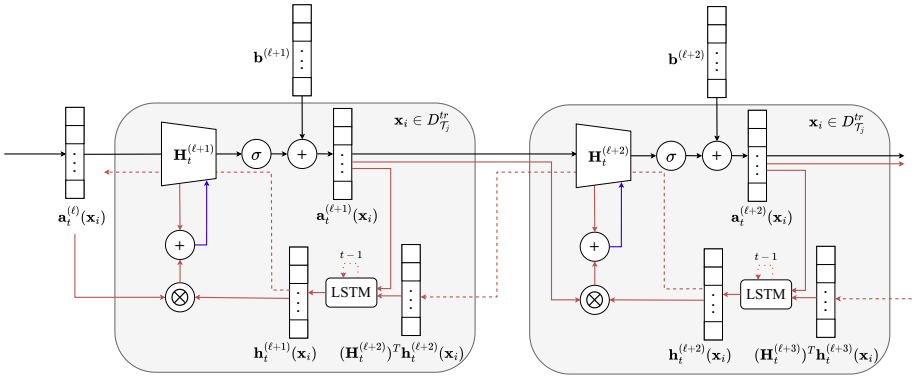
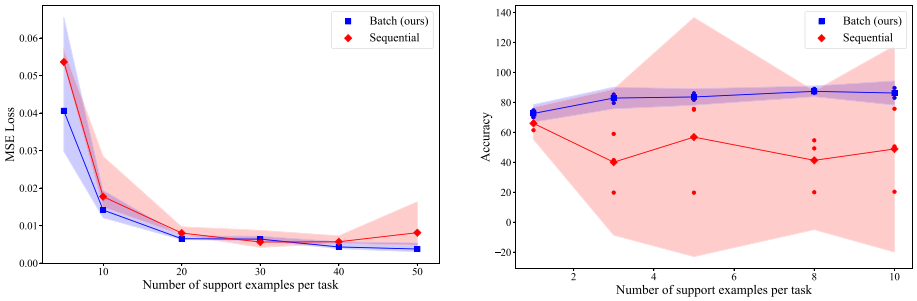


Fig. 4 The workflow of OP-LSTM. We have visualized two layers of the base-learner network. During the forward pass, the 2D hidden states $\mathbf{H}_t^{(\ell)}$ act as weight matrices of a feed-forward neural network that act on the input of that layer $\mathbf{a}_t^{(\ell-1)}$. This linear combination $\mathbf{H}_t^{(\ell)}\mathbf{a}_t^{(\ell-1)}(\mathbf{x}_i)$ is passed through a nonlinearity σ and added with a bias vector $\mathbf{b}^{(\ell)}$ to produce the activation $\mathbf{a}_t^{(\ell)}(\mathbf{x}_i)$. The entire forward pass is displayed by the black arrows. The red arrows, on the other hand, indicate the backward pass using the coordinate-wise LSTM. The outer product ($\otimes$) of the resulting hidden state $\mathbf{h}_t^{(\ell+1)}$ and the inputs from the previous layer $\mathbf{a}_t^{(\ell)}$ are added to the 2D hidden state $\mathbf{H}_t^{(\ell+1)}$ to produce $\mathbf{H}_{t+1}^{(\ell+1)}$ (blue arrow), which can be interpreted as the updated weight matrix

activation. Every layer of the base-learner network is an OP-LSTM block. The plain LSTM approach uses an LSTM as base-learner. For MAML, we use the best reported hyperparameters by Finn et al. (2017). We performed hyperparameter tuning for LSTM and OP-LSTM using random search and grid search, respectively (details can be found in Appendix B). Note that as such, the comparison against MAML and Prototypical networks is only for illustrative purposes, as the hyperparameter optimization procedure on these methods has, due to computational restrictions, not been executed under the same conditions.

For the miniImageNet and CUB image classification datasets, we use the Conv-4 base-learner network for all methods, following Snell et al. (2017); Finn et al. (2017). This base-learner consists of 4 blocks, where every block consists of 64 feature maps created with 3×3 kernels, BatchNorm, and ReLU nonlinearity. MAML uses a linear output layer to compute predictions, the plain LSTM operates on the flattened features extracted by the convolutional layers (as an LSTM taking image data as input does not scale well), whereas OP-LSTM uses an OP-LSTM block (see Fig. 4) on these flattened features. Importantly, OP-LSTM is only used in the final layer as it does currently not support propagating messages backward through max pooling layers.

We first study the within-domain performance of the meta-learning methods, where test tasks are sampled from the same dataset as the one used for training (albeit with unseen classes). Afterward, we also study the cross-domain performance, where the techniques train on tasks from a given dataset and are evaluated on test tasks from another dataset. More specifically, we use the scenarios miniImageNet → CUB (train on miniImageNet and evaluate on CUB) and vice versa.



(a) Sinewave - Average MSE loss (b) Omniglot - Average accuracy (%)

Fig. 5 The average accuracy score of a plain LSTM with sequential and batch support data processing on few-shot sine wave regression (left) and Omniglot classfication (right) for different numbers of training examples per task. Note that a lower MSE (left) or a higher accuracy (right) corresponds to better performance. The results are averaged over 3 runs (each measured over 600 meta-test tasks) with different random seeds and the 95% confidence intervals over the mean performances of the runs are shown as shaded regions. Additionally, in the right plot, we have added scatter marks to indicate the average performances per run (dots, unconnected, 3 per setting). Batch processing performs on par or outperforms sequential processing and improves the training stability over different runs

5.1 Permutation invariance for the plain LSTM

First, we investigate the difference in performance of the plain LSTM approach when processing the support data as a sequence $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_k, \mathbf{y}_k)$ or as a set $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_k, \mathbf{y}_k)\}$ (see Sect. 3.4) on few-shot sine wave regression and few-shot Omniglot classification. For the former, every task consists of 50 query examples, whereas for the latter, we have 10 query examples per class. We tuned the LSTM that processes the support data sequentially with random search (details in appendix). We compare the performance of this tuned sequential model to that of an LSTM with batching (with the same hyperparameter configuration) to see whether the resulting permutation invariance is helpful for the performance and training stability of the LSTM. To measure the stability of the training process, we compute the confidence interval over the mean performances obtained over 3 different runs rather than over all performances concatenated for the different runs, as done in later experiments for consistency with the literature.

The results of this experiment are shown in Fig. 5. In the case of few-shot sine wave regression (left subfigure), the performance of the LSTM with batching is on par or better compared with the sequential LSTM as the MSE score of the former is smaller or equal. We also note that the performance tends to improve with the amount of available training data. A similar, although more convincing, pattern can be seen in the case of few-shot Omniglot classification (right subfigure), where the LSTM with batching significantly outperforms the sequential LSTM across the different numbers of training examples per class. Surprisingly, in this case the performance of the LSTM does not improve as the number of examples per class increases. We found that this is due to training stability issues of the plain LSTM (as shown by the confidence intervals): for some runs, the LSTM does not learn and yields random performance, and in other runs the learning starts only after a certain period of burn-in iterations and fails to reach convergence within 80K meta-iterations (see appendix Sect. B.2 for detailed learning curves for every run). For the LSTM with batching, we do not observe such training

Table 1 Average test MSE on few-shot sine wave regression

	Parameters	5-Shot	10-Shot	20-Shot
MAML	17,018	0.18 ± 0.009	0.033 ± 0.003	0.005 ± 0.001
LSTM	20,201	$\mathbf{0.04} \pm 0.002$	0.010 ± 0.001	0.007 ± 0.000
OP-LSTM	18,107	0.11 ± 0.009	$\mathbf{0.008} \pm 0.001$	$\mathbf{0.003} \pm 0.000$

The 95% confidence intervals are displayed as $\pm x$, and calculated over all meta-test tasks. We used batch processing for the LSTM and OP-LSTM. The best performances are displayed in bold font

stability issues. This shows that batching not only helps improve the performance, but also greatly increases the training stability. Note that the fact that the shaded confidence interval of the sequential LSTM goes above the performance obtained by the batching LSTM is an artifact of using symmetrical confidence intervals above and below the mean trend: the sequential LSTM never outperforms the batching LSTM. As we can see, the MSE loss for both approaches decreases as the size of the support set increases, as more training data is available for learning. Furthermore, we see that the performance of the LSTM with batching improves with the number of available training data, whereas this is not the case for the sequential LSTM, which struggles to yield competitive performance. Overall, the results imply that the permutation invariance is a helpful inductive bias to improve the few-shot learning performance. Consequently, we will use the LSTM with batching henceforth.

5.2 Performance comparison on few-shot sine wave regression

Next, we compare the performance of the plain LSTM with batching, our proposed OP-LSTM, as well as MAML (Finn et al., 2017). To ensure a fair comparison with MAML, we tuned the hyperparameters in the same way as for the plain LSTM as done in the previous subsection on 5-shot sine wave regression. For this tuning, we used the default base-learner architecture consisting of two hidden layers with 40 ReLU nodes, followed by an output layer of 1 node. Afterward, we searched over different architectures with different numbers of parameters such that the expressivity in terms of the number of parameters does not limit the performance of MAML. We used the same base-learner architecture for the OP-LSTM as MAML without additional tuning.

The test performances on the sine wave regression taska are displayed in Table 1. We note MAML, despite having a comparable number of parameters (models with more parameters than LSTM and OP-LSTM performed worse), is outperformed by LSTM and OP-LSTM, indicating that LSTM and OP-LSTM have discovered more efficient learning algorithms for sine wave tasks. Comparing LSTM with OP-LSTM, we see that the former yields the best performance in the 5-shot setting, whereas OP-LSTM outperforms LSTM in the 10-shot and 20-shot settings.

5.3 Performance comparison on few-shot image classification

Within-domain Next, we investigate the within-domain performance of OP-LSTM and LSTM on few-shot image classification problems, namely, Omniglot, miniImageNet, and

Table 2 The mean test accuracy (%) on 5-way Omniglot classification across 3 different runs

Technique	Parameters	1-Shot	5-Shot
MAML	247, 621	84.1 ± 0.90	**93.5** ± 0.30
ProtoNet	247, 621	83.6 ± 0.88	93.4 ± 0.29
LSTM	13, 30, 097	72.6 ± 0.90	84.8 ± 0.50
OP-LSTM (ours)	249, 167	**84.3** ± 0.90	91.8 ± 0.30

The 95% confidence intervals are displayed as $\pm x$, and calculated over all runs and meta-test tasks (600 per run). The plain LSTM is outperformed by MAML. All methods (except LSTM) used a fully-connected feed-forward classifier. The best performances are displayed in bold font

Table 3 Meta-test accuracy scores on 5-way miniImageNet and CUB classification over 3 runs

Technique	Parameters	MiniImageNet		CUB	
		1-Shot	5-Shot	1-Shot	5-Shot
MAML	121, 093	48.6 ± 1.04	63.0 ± 0.54	57.5 ± 1.04	**74.8** ± 0.51
Warp-MAML	231, 877	50.4 ± 1.04	65.6 ± 0.53	59.6 ± 1.00	74.2 ± 0.51
SAP	412, 852	**53.0** ± 1.08	67.6 ± 0.51	**63.5** ± 1.00	73.9 ± 0.51
ProtoNet	121, 093	50.1 ± 1.04	65.4 ± 0.53	50.9 ± 1.01	63.7 ± 0.55
LSTM	55, 879, 349	20.2 ± 0.20	19.4 ± 0.20	–	–
OP-LSTM (ours)	141, 187	51.9 ± 1.04	**67.9** ± 0.50	60.2 ± 1.04	73.1 ± 0.52

The 95% confidence intervals are displayed as ± x, and calculated over all runs and meta-test tasks (600 per run). All methods used a Conv-4 backbone as a feature extractor. The "–" indicates that the method did not finish within 2 days of running time. The best performances are displayed in bold font

CUB. The results for the Omniglot dataset are displayed in Table 2. Note that the LSTM has many more parameters than the other methods as it consists of multiple fully-connected layers with large hidden dimensions, which were found to give the best validation performance. As we can see, the plain LSTM (with batching) does not yield competitive performance compared with the other methods, in spite of the fact that it has many more parameters and, in theory, could learn any learning algorithm. This shows that the LSTM is hard to optimize and struggles to find a good solution in more complex few-shot learning settings, i.e., image classification. OP-LSTM, on the other hand, which separates the learning procedure from the input representation, yields competitive performance compared with MAML and ProtoNet in both the 1-shot and 5-shot settings, whilst using fewer parameters than the plain LSTM.

The results for miniImageNet and CUB are displayed in Table 3. Note that again, the LSTM uses more parameters than other methods as it consists of multiple large fully-connected layers which were found to yield the best validation performance. Nonetheless, it is applied on top of representations computed with the Conv-4 backbone, which is also used by all other methods. As we can see, the plain LSTM approach performs at chance level, again suggesting that the optimization problem of finding a good learning algorithm is too complex for this problem. The OP-LSTM, on the other hand, yields competitive or superior performance compared with all tested baselines on both miniImageNet and CUB,

Table 4 Average cross-domain meta-test accuracy scores over 5 runs using a Conv-4 backbone

	MIN → CUB		CUB → MIN	
	1-Shot	5-Shot	1-Shot	5-Shot
MAML	37.9 ± 0.40	53.6 ± 0.40	31.1 ± 0.36	45.8 ± 0.39
Warp-MAML	42.0 ± 0.43	56.9 ± 0.42	31.1 ± 0.35	41.3 ± 0.36
SAP	41.5 ± 0.44	58.0 ± 0.41	33.3 ± 0.39	47.1 ± 0.39
ProtoNet	39.7 ± 0.41	56.0 ± 0.41	31.7 ± 0.34	45.3 ± 0.38
LSTM	20.1 ± 0.28	20.0 ± 0.25	–	–
OP-LSTM (ours)	**42.3** ± 0.42	**58.5** ± 0.41	**35.8** ± 0.40	**49.0** ± 0.40

Techniques trained on tasks from one data set and were evaluated on tasks from another data set. The 95% confidence intervals are displayed as ± x, and calculated over all runs and meta-test tasks. The "–" indicates that the method did not finish within 2 days of running time. The best performances are displayed in bold font

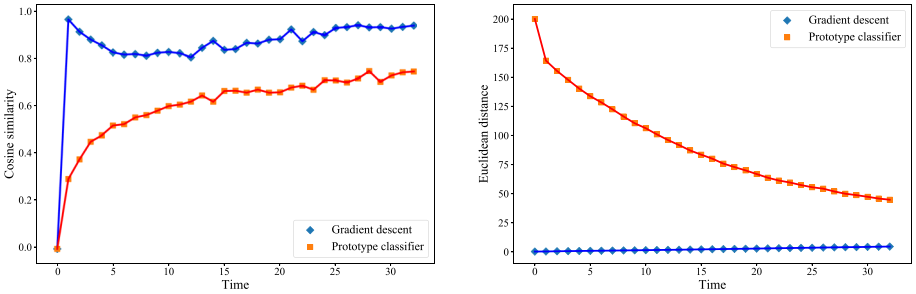
regardless of the number of shots, which shows the advantage of decoupling the input representation from the learning procedure.

Cross-domain Next, we investigate the cross-domain performance of the LSTM and OP-LSTM, where the test tasks come from a different a different dataset than the training tasks. We test this in two scenarios: train on miniImageNet and evaluate on CUB (MIN → CUB) and vice verse (CUB → MIN). The results of this experiment are displayed in Table 4. Again, the plain LSTM does not outperform a random classifier, whilst the OP-LSTM yields superior performance in every tested scenario, showing its versatility in this challenging setting.

5.4 Analysis of the learned weight updates

Lastly, we investigate how OP-LSTM updates the weights of the base-learner network. More specifically, we measure the cosine similarity and Euclidean distance between the OP-LSTM updates and updates made by gradient descent or Prototypical network. Denoting the initial final classifier weight matrix as $\mathbf{H}_0^{(L)}$, the OP-LSTM update direction after T updates is $\Delta_{OP} = \vec{\mathbf{H}}_T^{(L)} - \vec{\mathbf{H}}_0^{(L)}$, where $\vec{\mathbf{H}}$ means that we vectorize the matrix by flattening it. Similarly, we can measure the update compared with the initial weight matrix and those obtained by employing nearest-prototype classification ($\mathbf{H}_{Proto}^{(L)}$) as done in Prototypical network or gradient descent $\mathbf{H}_{GD}^{(L)}$, where the latter is obtained by performing T gradient update steps (with a learning rate of 0.01). These updates are associated with the update direction vectors $\Delta_{Proto} = \vec{\mathbf{H}}_{Proto}^{(L)} - \vec{\mathbf{H}}_0^{(L)}$ and $\Delta_{GD} = \vec{\mathbf{H}}_{GD}^{(L)} - \vec{\mathbf{H}}_0^{(L)}$. We can then measure the distance between the update direction Δ_{OP} of the OP-LSTM and Δ_{Proto} and Δ_{GD}. As a distance measure, we use the Euclidean distance. In addition, we also measure the cosine similarity between the update directions as an inverse distance measure that is invariant to the scale and magnitudes of the vectors. After every 2, 500 episodes, we measure these Euclidean distances and cosine similarity scores on the validation tasks, and average the results over 3 runs.

The results of this experiment are displayed in Fig. 6. As we can see, the cosine similarity between the weight update directions of OP-LSTM and gradient descent and



(a) Cosine similarity (b) Euclidean distance

Fig. 6 The average cosine similarity (left) and Euclidean distance (right) between the weight update directions of the OP-LSTM and a prototype-based and gradient-based classifier as a function of time on 5-way 1-shot miniImageNet classification. Each point on the x-axis indicates a validation step, which is performed after every 2, 500 episodes. The results are averaged over 3 runs with different random seeds and the 95% confidence intervals are shown as shaded regions. The confidence intervals are within the size of the symbols and imperceptible. As time progresses, the updates performed by OP-LSTM become more similar to those of gradient descent and prototype-based classifiers (increasing cosine similarity)

prototype-based classifiers increases with training time. OP-LSTM very quickly learns to update the weights in a similar direction as gradient descent, followed by a gradual decline in similarity, which is later followed by a gradual increase. This gradual decline may be to incorporate more prototype-based updates. Looking at the Euclidean distance, we observe the same pattern for the similarity compared with the prototype-based classifier, as the distance between the updates decreases (indicating a higher similarity). The Euclidean distance between OP-LSTM updates and gradient updates slightly increase over time, which may be a side effect of the sensitivity to scale and magnitude of this distance measure. Thus, even if both would perform gradient descent, but with different learning rates, The cosine similarity gives a better idea of directional similarity as it abstracts away from the magnitude of the vectors.

6 Relation to other methods

Here, we study the relationship of OP-LSTM to other existing meta-learning methods. More specifically, we aim to show that OP-LSTM is a general meta-learning approach, which can *approximate* the behaviour of different classes of meta-learning, such as optimization-based meta-learners [e.g., MAML (Finn et al., 2017)] and metric-based methods [e.g., Prototypical network (Snell et al., 2017)].

Model-agnostic meta-learning (MAML) MAML (Finn et al., 2017) aims to learn good initialization parameters for a base-learner network $\boldsymbol{\theta} = \{\mathbf{W}_0^{(1)}, \mathbf{W}_0^{(2)}, \ldots, \mathbf{W}_0^{(L)}, \mathbf{b}_0^{(1)}, \mathbf{b}_0^{(2)}, \ldots, \mathbf{b}_0^{(L)}\}$ such that new tasks can be learned efficiently using a few gradient update steps. Here, $\mathbf{W}_0^{(\ell)}$ is the initial weight matrix of layer ℓ and $\mathbf{b}_0^{(\ell)}$ the initial bias vector of layer ℓ when presented with a new task.

The initial 2D hidden states $\mathbf{H}_0^{(\ell)}$ in OP-LSTM can be viewed as the initial weights $\mathbf{W}_0^{(\ell)}$ of the neural network in MAML. In MAML, the weights in layer ℓ for a given input are updated as $\mathbf{W}_{t+1}^{(\ell)} = \mathbf{W}_t^{(\ell)} - \eta\delta^{(\ell)}(\mathbf{p}^{(\ell-1)}(\mathbf{x}))^T$, where $\mathbf{a}^{(i)}(\mathbf{x})$ (with $1 \leq i \leq L$) is the vector of

post-activation values in layer i as a result of the input $\mathbf{x}$, and $\delta^{(i)} = \nabla_{\mathbf{a}^{(i)}} \mathcal{L}(\mathbf{x})$, where $\mathcal{L}(\mathbf{x}, \mathbf{y})$ is the loss on input $\mathbf{x}$ given the ground-truth target $\mathbf{y}$, and η is the learning rate.

Instead of using this hand-crafted weight update rule, OP-LSTM learns the update rule using the outer product of LSTM hidden states and the input activation. From Eq. (14) it follows that OP-LSTM is capable of updating the weights $\mathbf{H}_t^{(\ell)}$ with gradient descent by setting $\mathbf{h}_{t+1}^{(\ell)}(\mathbf{x}) = -\eta \delta^{(\ell)} = -\eta \nabla_{\mathbf{a}^{(\ell)}} \mathcal{L}(\mathbf{x}, \mathbf{y})$. [Note that in Eq. (14) the gradient is also normalized by the Frobenius norm, which is formally not part of MAML.] We note that the inputs to the coordinate-wise LSTM contain the necessary information to compute the errors $\delta^{(\ell)}$ in every layer. That is, for the output layer, the LSTM receives the ground-truth output and prediction in the output layer. For earlier layers, the LSTM receives the back-propagated messages (the errors), as well as the activations. Consequently, OP-LSTM can update the 2D hidden states $\mathbf{H}^{(\ell)}$ with gradient descent, as MAML. OP-LSTM is thus an approximate generalization of MAML as it could learn to perform the same weight matrix updates, although OP-LSTM does not update the bias vectors given a task.

Prototypical network Prototypical network (Snell et al., 2017) aims to learn good initial weights $\boldsymbol{\theta} = \{\mathbf{W}_0^{(1)}, \mathbf{W}_0^{(2)}, \ldots, \mathbf{W}_0^{(L-1)}, \mathbf{b}_0^{(1)}, \mathbf{b}_0^{(2)}, \ldots, \mathbf{b}_0^{(L-1)}\}$ for all parameters except for the final layer, such that a nearest-prototype classifier yields good performance. Let $f_\theta(\mathbf{x}_i)$ be the embeddings produced by this $(L-1)$-layered network for a given observation $\mathbf{x}_i$ (from the support set). Note that the network has $(L-1)$ layers as this is the feature embedding module without the output layer. Prototypical network computes centroids $\mathbf{c}_n = \frac{1}{|X_n|} \sum_{\mathbf{x}_i \in X_n} f_\theta(\mathbf{x}_i)$ for every class n, where X_n is the set of all support inputs with ground-truth class n, and $f_\theta(\mathbf{x})$ is the embedding of input $\mathbf{x}$. Then, the predicted score of a new input $\hat{\mathbf{x}}$ for class n is then given by $\hat{y}_n(\hat{\mathbf{x}}) = \frac{\exp(-d(f_\theta(\hat{\mathbf{x}}), \mathbf{c}_n))}{\sum_{n'} \exp(-d(f_\theta(\hat{\mathbf{x}}), \mathbf{c}_{n'}))}$, where $d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\|_2^2$ is the squared Euclidean distance, and n' is a variable iterating over all classes.

This nearest-prototype classifier can be seen as a regular linear output layer (Triantafillou et al., 2020). To see this, note that the prediction score for class j is given by

$$\hat{y}_j(\hat{\mathbf{x}}) = \|f_\theta(\hat{\mathbf{x}}) - \mathbf{c}_n\|_2^2 = (f_\theta(\hat{\mathbf{x}}) - \mathbf{c}_n)^T (f_\theta(\hat{\mathbf{x}}) - \mathbf{c}_n) \tag{18}$$

$$= f_\theta(\hat{\mathbf{x}})^T f_\theta(\hat{\mathbf{x}}) - 2 f_\theta(\hat{\mathbf{x}})^T \mathbf{c}_n + \mathbf{c}_n^T \mathbf{c}_n \tag{19}$$

$$\propto -2 f_\theta(\hat{\mathbf{x}})^T \mathbf{c}_n + \mathbf{c}_n^T \mathbf{c}_n, \tag{20}$$

where we ignored the first term $(f_\theta(\hat{\mathbf{x}})^T f_\theta(\hat{\mathbf{x}}))$ as it is constant across all classes n. The prediction score for class j is thus obtained by taking the dot product between the input embedding $f_\theta(\hat{\mathbf{x}})$ and $-2\mathbf{c}_n$ and by adding a bias term $b_n = \mathbf{c}_n^T \mathbf{c}_n$. Thus, the prototype-based classifier is equivalent to a linear output layer, i.e., $\hat{\mathbf{x}} = \mathbf{W}^{(L)} f_\theta(\hat{\mathbf{x}}) + \mathbf{b}^{(L)}$ where the n-th row of $\mathbf{W}^{(L)}$ corresponds to $-2\mathbf{c}_n$ and the n-th element of $\mathbf{b}^{(L)}$ is equal to $\mathbf{c}_n^T \mathbf{c}_n$. OP-LSTM can approximate the behavior of Prototypical network with $T = 1$ steps per task as follows. First, assume that the underlying base-learner network is the same for Prototypical network and OP-LSTM, i.e., the initialization of the hidden state is equivalent to the initial weights of the base-learner used by Prototypical network $\mathbf{H}_0^{(\ell)} = \mathbf{W}_0^{(\ell)}$ for $\ell \in \{1, 2, \ldots, L-1\}$, and that the hidden state of the output layer in OP-LSTM is a matrix of zeros, i.e., $\mathbf{H}_0^{(L)} = \mathbf{0}$. Second, let the hidden states of the LSTM in OP-LSTM be be a vector of zeros $\mathbf{h}^{(\ell)} = \mathbf{0}$ for every layer $\ell < L$, and let the hidden state of the output layer given the example $\mathbf{x}_i' = (\mathbf{x}_i, \mathbf{y}_i)$ be the label identity function $\mathbf{h}^{(L)}(\mathbf{x}_i') = \mathbf{y}_i$ (which can be learned by an LSTM). Then,

OP-LSTM will update the hidden states as follows using Eq. (14). The n-th row of $\mathbf{H}^{(L)}$ will equal $\gamma \frac{1}{M} \sum_{\mathbf{x}_i \in X_n} \frac{\mathbf{a}^{(L-1)}(\mathbf{x}_i)}{\|\mathbf{a}^{(L-1)}(\mathbf{x}_i)\|_F}$, where $X_n = \{\mathbf{x}_i \in D^{tr}_{\mathcal{T}_j} | \mathbf{y}_i = \mathbf{e}_n\}$ is the set of training inputs with class n, and γ and M are the learning rate of OP-LSTM and number of examples respectively. Note that this expression corresponds to the scaled prototype (mean of the embeddings) of class n, that is, $\gamma \bar{\mathbf{c}}_n$, where $\bar{\mathbf{c}}_n = \frac{1}{M} \sum_{\mathbf{x}_i \in X_n} \frac{\mathbf{a}^{(L-1)}(\mathbf{x}_i)}{\|\mathbf{a}^{(L-1)}(\mathbf{x}_i)\|_F}$. The prediction for the n-th class for a given input $\hat{\mathbf{x}}$ is thus given by $\gamma \bar{\mathbf{c}}_n^T \mathbf{a}^{(L-1)}(\mathbf{x}) + b_n^{(L)}$, where we omitted the time step for $\mathbf{a}^{(L-1)}$ and b_n is a fixed bias in the output layer. Note that for $\gamma = -2$, the first term $(-2\bar{\mathbf{c}}_n^T \mathbf{a}^{(L-1)}(\mathbf{x}))$ resembles the first term in the prediction made by Prototypical network for class n, which is given by $-2\mathbf{c}_n^T \mathbf{a}^{(L-1)}(\mathbf{x})$, where $\mathbf{a}^{(L-1)}(\mathbf{x}) = f_\theta(\mathbf{x})$. Hence, OP-LSTM can learn to approximate (up to the bias term) a normalized Prototypical network classifier.

We have thus shown that OP-LSTM can learn to implement a parametric learning algorithm (gradient descent) as well as a non-parametric learning algorithm (prototype-based classifier), demonstrating the flexibility of the approach.

7 Conclusions

Meta-learning is a strategy to enable deep neural networks to learn from small amounts of data. The field has witnessed an increase in popularity in recent years, and many new techniques are being developed. However, the potential of some of the earlier techniques have not been studied thoroughly, despite promising initial results. In our work, we revisited the plain LSTM approach proposed by Hochreiter et al. (2001) and Younger et al. (2001). This approach simply ingests the training data for a given task, and conditions the predictions of new query inputs on the resulting hidden state.

We analysed this approach from a few-shot learning perspective and uncovered two potential issues for embedding a learning algorithm into the weights of the LSTM: (1) the hidden embeddings of the support set are not permutation invariant, and (2) the learning algorithm and the input embedding mechanism are intertwined, which leads to a challenging optimization problem and an increased risk of overfitting. In our work, we proposed to overcome issue (1) by mean pooling the embeddings of individual training examples, rendering the obtained embedding permutation invariant. We found that this method is highly effective and increased the performance of the plain LSTM on both few-shot sine wave regression and image classification. Moreover, with this first solution, the plain LSTM approach already outperformed the popular meta-learning method MAML (Finn et al., 2017) on the former problem. It struggled, however, to yield good performance on few-shot image classification problems, highlighting the difficulty of optimizing this approach.

In order to resolve this difficulty, we proposed a new technique, Outer Product LSTM (OP-LSTM), that uses an LSTM to update the weights of a base-learner network. By doing this, we effectively decouple the learning algorithm (the weight updates) from the input representation mechanism (the base-learner), solving issue (2), as done in previous works (Ravi and Larochelle, 2017; Andrychowicz et al., 2016). Compared with previous works, OP-LSTM does not receive gradients as inputs. Our theoretical analysis shows that OP-LSTM is capable of performing an approximate form of gradient descent (as done in MAML (Finn et al., 2017)) as well as a nearest prototype based approach (as done in Prototypical network (Snell et al., 2017)), showing the flexibility and expressiveness of the

method. Empirically, we found that OP-LSTM overcomes the optimization issues associated with the plain LSTM approach on few-shot image classification benchmarks, whilst using fewer parameters. It yields competitive or superior performance compared with MAML (Finn et al., 2017) and Prototypical network (Snell et al., 2017), both of which it can approximate.

Future work When the base-learner is a convolutional neural network, we applied OP-LSTM on top of the convolutional feature embeddings. A fruitful direction for future research would be to propose a more general form of OP-LSTM that can update also the convolutional layers. This would require new backward message passing protocols to go through pooling layers often encountered in convolutional neural networks.

Moreover, we note that OP-LSTM is one way to overcome the two issues associated with the plain LSTM approach, but other approaches could also be investigated. For example, one could try to implement a convLSTM (Shi et al., 2015) such that the LSTM can be applied directly to raw inputs, instead of only after the convolutional backbone in case of image classification problems.

Another fruitful direction for future work would be to investigate different recurrent neural architectures and their ability to perform meta-learning. In the pioneering work of Hochreiter et al. (2001) and Younger et al. (2001), it was shown that only LSTMs were successful whereas vanilla recurrent neural networks and Elman networks failed to meta-learn simple functions. It would be interesting to explore how architectural design choices influence the ability of recurrent networks to perform meta-learning.

Lastly, OP-LSTM is a method to learn the weight update rule for a base-learner network, and is thus orthogonal to many advances and new methods in the field of meta-learning, such as Warp-MAML (Flennerhag et al., 2020) and SAP (Huisman et al., 2023). Since this is a nontrivial extension of these methods, we leave this for future work. We think that combining these methods could yield new state-of-the-art performance.

Appendix A Sine wave regression: additional results

We also performed an experiment to investigate the effect of the input representation on the performance of the plain LSTM approach [proposed by Younger et al. (2001) and Hochreiter et al. (2001)] on the 5-shot sine wave regression performance. The experimental setting follows the setup described in Sect. 5.1. For every input format, we performed hyperparameter tuning with the same randomly sampled hyperparameter configurations using Table 6. The performances of the best validated models per input format are displayed in Table 5. The best

Table 5 The influence of different input information on the performance of the LSTM on 5-shot sine wave regression

Input $\mathbf{x}_t$	Prev target y_{t-1}	Prev pred $\hat{y}_{t-1}$	Prev error e_{t-1}	5-Shot MSE
✓	✓			0.04 ± 0.002
✓	✓	✓		$\mathbf{0.03} \pm 0.002$
✓	✓		✓	0.05 ± 0.004
✓	✓	✓	✓	0.06 ± 0.011

95% confidence intervals are displayed as $\pm x$. The best performances are displayed in bold font

performance is obtained by feeding the current input, previous target, and the previous prediction into the LSTM, although the differences with other inputs are small.

Appendix B Hyperparameter tuning

B.1 Permutation invariance experiments

For the permutation invariance experiments on few-shot sine wave regression, we sampled 20 random configurations for the plain LSTM from the distributions displayed in Table 6 and validated their performance on 5-shot ($k = 5$) sine-wave regression. We selected the best configuration and evaluated it on the meta-test tasks,

For Omniglot, we performed random search with a function evaluation budget of 100, with a fixed learning rate of 0.001. The architecture of the plain LSTM with sequential data processing was sampled uniformly at random from {1024-512-256-128-64, 2048-1024-512-128-64, 2048-1024-512-256-128, 1024-600-400-200-92, 1024-512-512-256-128-64, 1024-512-512-256-256-128-64, 612-400-256-128-64, 1024-1024-1024-512-256-128-64, 2048-1024-512-180-100, 1024-580-280-160-80, 256-128-64, 512-256-128-64, 128-64-64-64, 256-128-64, 512-256-64, 256-128-100, 128-64-64-64-64, 64-64-64-64, 50-50}, the number of passes over the support data T was sampled uniformly at random from $\{1, 2, \ldots, 10\}$, and the meta-batch size from $\{1, 2, \ldots, 32\}$. We used the best hyperparameter configuration of the sequential plain LSTM for the plain LSTM with batching to compare the differences in performance.

B.2 Omniglot

For the *plain LSTM* approach, we used the best hyperparameter configuration found for the permutation invariance experiments.

For *OP-LSTM*, we performed a grid search, varying the meta-batch size within {1, 4, 8, 16, 32}, the architecture of the coordinate-wise LSTM within {20-1, 10-10-1, 40-5, 40-20-1, 20-20-20-5} (note that the last element is always 1 because it operates per coordinate), and the number of passes over the support set within {1, 3, 5, 10}.

Detailed learning curves for the plain LSTM on Omniglot Here, we show the validation learning curves of the sequential LSTM and the LSTM which uses batching to complement the results displayed in Sect. 5.1. Figure 7 displays the validation learning curves of the LSTM with batch data ingestion (top row) and the LSTM with sequential data processing (bottom row). As we can see, batching increases the stability of the training process and

Table 6 The used ranges and distributions for tuning the hyperparameters with random search for sine wave regression

Hyperparameter	Range
Number of layers	Uniform ($\{1, 2, 3, 4\}$)
Hidden dimensions	Uniform ($\{1, 3, 8, 20, 40\}$)
Meta-batch size	Uniform ($\{1, 2, 3, 4\}$)
Learning rate	LogUniform (1e−5, 4e−2)
Unroll steps	Uniform ($\{1, 2, \ldots, 14\}$)

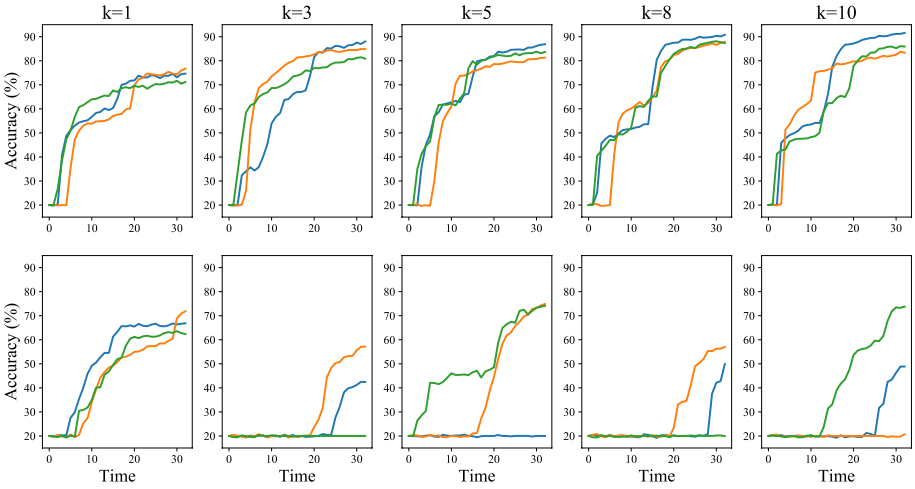


Fig. 7 The mean validation accuracy of the LSTM over time on Omniglot for every of the three different runs, for different numbers of examples per class *k*. *Top row:* LSTM with batching (mean-pooling). *Bottom row:* LSTM with sequential data ingestion. As we can see, batching improves the stability of the training process

Table 7 Mean running times on 5-way miniImageNet and CUB classification over 3 runs

Technique	Parameters	miniImageNet		CUB	
		1-Shot	5-Shot	1-Shot	5-Shot
MAML	121, 093	13 h 9 min	12 h 1 min	26 h 57 min	17 h 39 min
Warp-MAML	231, 877	12 h 25 min	12 h 30 min	13 h 6 min	12 h 48 min
SAP	412, 852	5 h 40 min	11 h 14 min	7 h 11 min	11 h 17 min
ProtoNet	121, 093	4 h 14 min	5 h 6 min	31 h 18 min	38 h 46 min
LSTM	55, 879, 349	40 h 14 min	46 h 47 min	–	–
OP-LSTM (ours)	141, 187	4 h 50 min	5 h 31 min	31 h 58 min	40 h 8 min

All methods used a Conv-4 backbone as a feature extractor. "*x* h *y* min" means *x* h and *y* min. The "–" indicates that the method did not finish within 2 days of running time

makes the LSTM less sensitive to the random initialization, as every run succeeds to reach convergence in contrast to the sequential LSTM.

B.3 miniImageNet and CUB

For *plain LSTM*, we used random search with a budget of 130 function evaluations, the meta-batch size was sampled uniformly between 1 and 48, the number of layers between 1 and 4, the hidden size log-uniformly between 32 and 3200, and the number of passes T over the support dataset uniformly between 2 and 9.

For *OP-LSTM*, we performed the same grid search as for Omniglot. We use the best found hyperparameters for both methods on miniImageNet also on CUB.

We also measured the running times of the techniques on miniImageNet and CUB, as shown in Table 7. We note that the running times may be affected by the server's load and thus can only give a rough estimation of the required amount of compute time. As we can see, the plain LSTM is the slowest method, despite achieving random performance on miniImageNet. OP-LSTM, in contrast, is more efficient.

B.4 Robustness to random seeds

Here, we investigate the robustness of the investigated methods to the random seed for the few-shot image classification experiments performed in Sect. 5.3. We perform th Instead of computing the confidence intervals over the performances of all test tasks for all seeds, we now compute the confidence interval over the mean test performance per run. As we perform three runs per method, we compute the confidence intervals over three observations per method. Note that the mean performance does not change as taking the mean of the three means will be equivalent (as the means are based on an equal number of task performances).

B.4.1 Within-domain

Here, we present additional results for the conducted within-domain image classification experiments.

Omniglot The mean test performance and confidence intervals over the random seeds for Omniglot image classification are shown in Table 8. As we can see, the confidence intervals are higher than in previous experiments because the intervals are computed over 3 observations instead of 1800 individual test task performances (600 per run). As we can see, the LSTM is unstable, supporting the hypothesis that the optimization problem is difficult. OP-LSTM, on the other hand, is less sensitive to the chosen random seed and has a stability that is comparable to that of MAML.

MiniImageNet and CUB The mean test performance and confidence intervals over the random seeds for miniImageNet and CUB image classification are shown in Table 9. In contrast to what we observed on Omniglot, the LSTM is now more stable. This is caused

Table 8 The mean test accuracy (%) on 5-way Omniglot classification across 3 different runs

Technique	Parameters	1-Shot	5-Shot
MAML	247, 621	84.1 ± 3.10	**93.5** ± 0.70
ProtoNet	247, 621	83.6 ± 0.52	93.4 ± 1.48
LSTM	13, 530, 097	72.6 ± 3.87	84.8 ± 6.12
OP-LSTM (ours)	249, 167	**84.3** ± 3.18	91.8 ± 0.70

The 95% confidence intervals, computed over the mean performances of the 3 different random seeds, are displayed as $\pm x$. The plain LSTM is outperformed by MAML. All methods (except LSTM) used a fully-connected feed-forward classifier. The best performances are dislpayed in bold font

Table 9 Meta-test accuracy scores on 5-way miniImageNet and CUB classification over 3 runs

Technique	Parameters	miniImageNet		CUB	
		1-Shot	5-Shot	1-Shot	5-Shot
MAML	121, 093	48.6 ± 4.00	63.0 ± 0.33	57.5 ± 0.83	**74.8** ± 2.10
Warp-MAML	231, 877	50.4 ± 2.58	65.6 ± 0.98	59.6 ± 2.15	74.2 ± 2.51
SAP	412, 852	**53.0** ± 3.71	67.6 ± 0.47	**63.5** ± 6.24	73.9 ± 1.57
ProtoNet	121, 093	50.1 ± 4.06	65.4 ± 2.84	50.9 ± 2.35	63.7 ± 0.47
LSTM	55, 879, 349	20.2 ± 0.60	19.4 ± 0.47	–	–
OP-LSTM (ours)	141, 187	51.9 ± 2.52	**67.9** ± 2.40	60.2 ± 1.58	73.1 ± 1.57

The 95% confidence intervals, computed over the mean performances of the 3 different random seeds, are displayed as ± x. All methods used a Conv-4 backbone as a feature extractor. The "–" indicates that the method did not finish within 2 days of running time. The best performances are displayed in bold font

Table 10 Average cross-domain meta-test accuracy scores over 5 runs using a Conv-4 backbone

	MIN → CUB		CUB → MIN	
	1-Shot	5-Shot	1-Shot	5-Shot
MAML	37.9 ± 2.22	53.6 ± 0.67	31.1 ± 1.19	45.8 ± 2.06
Warp-MAML	42.0 ± 0.85	56.9 ± 4.16	31.1 ± 1.59	41.3 ± 1.37
SAP	41.5 ± 3.72	58.0 ± 1.79	33.3 ± 2.33	47.1 ± 1.28
ProtoNet	39.7 ± 4.11	56.0 ± 4.89	31.7 ± 0.20	45.3 ± 1.84
LSTM	20.1 ± 0.77	20.0 ± 0.40	–	–
OP-LSTM (ours)	**42.3** ± 1.90	**58.5** ± 1.49	**35.8** ± 2.98	**49.0** ± 0.80

Techniques trained on tasks from one data set and were evaluated on tasks from another data set. The 95% confidence intervals, computed over the mean performances of the 3 different random seeds, are displayed as ± x. The "–" indicates that the method did not finish within 2 days of running time. The best performances are displayed in bold font

by the fact that it consistently fails to learn a learning algorithm that performs better than random guessing, and thus performs stably at chance level.

B.4.2 Cross-domain

Lastly, we compute the confidence intervals in cross-domain settings and display the results in Table 10. Again, the LSTM is a stable random guesser. The other algorithms are less stable, but do yield a better performance. We cannot observe a general pattern of stability in the sense that one algorithm is consistently more stable than others.

Acknowledgements This work was performed using the compute resources from the Academic Leiden Interdisciplinary Cluster Environment (ALICE) provided by Leiden University.

Author Contributions MH has conducted the research presented in this manuscript. TM, AP, and JvR have regularly provided feedback on the work, contributed towards the interpretation of results, and have critically revised the whole. All authors approve the current version to be published and agree to be accountable for all aspects of the work in ensuring that questions related to the accuracy or integrity of any part of the work are appropriately investigated and resolved.

Availability of data and materials All data that was used in this research have been published as benchmarks by Deng et al. (2009); Vinyals et al. (2016) (miniImageNet) and Wah et al. (2011) (CUB), and is publicly available. The data generator for sine wave regression experiments can be found in the provided code (see below).

Code availability All code that was used for this research is made publicly available at https://github.com/mikehuisman/lstm-fewshotlearning-oplstm.

Declarations

Conflict of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Consent for publication Not applicable: this research does not involve personal data, and publishing of this manuscript will not result in the disruption of any individual's privacy.

Employment All authors declare that there is no recent, present, or anticipated employment by any organization that may gain or lose financially through the publication of this manuscript.

References

Alver, S., & Precup, D. (2021). What is going on inside recurrent meta reinforcement learning agents? arXiv preprint arXiv:2104.14644.

Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., & De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems* (Vol. 29, pp. 3988–3996). Curran Associates Inc.

Brazdil, P., van Rijn, J. N., Soares, C., & Vanschoren, J. (2022). *Metalearning: Applications to automated machine learning and data mining* (2nd ed.). Springer.

Chan, S., Santoro, A., Lampinen, A., Wang, J., Singh, A., Richemond, P., McClelland, J., & Hill, F. (2022). Data distributional properties drive emergent in-context learning in transformers. In *Advances in neural information processing systems*.

Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 248–255). IEEE.

Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., & Abbeel, P. (2016). RL^2: Fast reinforcement learning via slow reinforcement learning. arXiv preprint arXiv:1611.02779.

Finn, C., & Levine, S. (2017). Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. arXiv preprint arXiv:1710.11622.

Finn, C., Abbeel, P., & Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th international conference on machine learning (ICML'17)* (pp. 1126–1135). PMLR.

Flennerhag, S., Rusu, A. A., Pascanu, R., Visin, F., Yin, H., & Hadsell, R. (2020). Meta-learning with warped gradient descent. In *International conference on learning representations (ICLR'20)*.

He, K., Zhang, X., Ren, S., & Sun, J., (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (pp. 1026–1034).

Hochreiter, S., Younger, A. S., & Conwell, P. R. (2001). Learning to learn using gradient descent. In *International conference on artificial neural networks* (pp. 87–94). Springer.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780.
Huisman, M., Plaat, A., & van Rijn, J. N. (2023). Subspace adaptation prior for few-shot learning (forthcoming).
Huisman, M., Van Rijn, J. N., & Plaat, A. (2021). A survey of deep meta-learning. *Artificial Intelligence Review, 54*(6), 4483–4541.
Kingma, D. P., & Ba, J. L. (2015). Adam: A method for stochastic gradient descent. In *International conference on learning representations (ICLR'15)*.
Kirsch, L., Harrison, J., Sohl-Dickstein, J., & Metz, L. (2022). General-purpose in-context learning by meta-learning transformers. arXiv preprint arXiv:2212.04458.
Kirsch, L., & Schmidhuber, J. (2021). Meta learning backpropagation and improving it. *Advances in Neural Information Processing Systems, 34*, 14122–14134.
Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems, 25*, 1097–1105.
Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science, 350*(6266), 1332–1338.
LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521*(7553), 436–444.
Lee, Y., & Choi, S. (2018). Gradient-based meta-learning with learned layerwise metric and subspace. In *Proceedings of the 35th international conference on machine learning (ICML'18)* (pp. 2927–2936). PMLR.
Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529–533.
Naik, D. K., & Mammone, R. J. (1992). Meta-neural networks that learn by learning. In *International joint conference on neural networks (IJCNN'92)* (pp. 437–442). IEEE.
Olah, C. (2015). Understanding LSTM networks. Retrieved January 23, 2023, from https://colah.github.io/posts/2015-08-Understanding-LSTMs/
Park, E., & Oliva, J. B. (2019). Meta-curvature. *Advances in Neural Information Processing Systems, 32*, 3309–3319.
Ravi, S., & Larochelle, H. (2017). Optimization as a Model for Few-Shot Learning. In *International conference on learning representations (ICLR'17)*.
Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., & Lillicrap, T. (2016). Meta-learning with memory-augmented neural networks. In *Proceedings of the 33rd international conference on international conference on machine learning (ICML'16)* (pp. 1842–1850).
Schmidhuber, J. (1987). *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook* (Master's thesis, Technische Universität München).
Shi, X., Chen, Z., Wang, H., Yeung, D. Y., Wong, W. K., & Woo, W. C. (2015). Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems* (Vol. 28).
Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., & Dieleman, S. (2016). Mastering the game of go with deep neural networks and tree search. *Nature, 529*(7587), 484–489.
Snell, J., Swersky, K., & Zemel, R. (2017). Prototypical networks for few-shot learning. In *Advances in neural information processing systems* (Vol. 30, pp. 4077–4087). Curran Associates Inc.
Thrun, S. (1998). Lifelong learning algorithms. In *Learning to learn* (pp. 181–209). Springer.
Triantafillou, E., Zhu, T., Dumoulin, V., Lamblin, P., Evci, U., Xu, K., Goroshin, R., Gelada, C., Swersky, K., Manzagol, P. A., Larochelle, H. (2020). Meta-dataset: A dataset of datasets for learning to learn from few examples. In *International conference on learning representations (ICLR'20)*.
Vinyals, O., Blundell, C., Lillicrap, T., & Wierstra, D. (2016). Matching networks for one shot learning. *Advances in Neural Information Processing Systems, 29*, 3637–3645.
Wah, C., Branson, S., Welinder, P., Perona, P., & Belongie, S. (2011). *The Caltech-UCSD Birds-200-2011 dataset* (Tech. Rep. CNS-TR-2011-001, California Institute of Technology).
Wang, J.X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., & Botvinick, M. (2016). Learning to reinforcement learn. arXiv preprint arXiv:1611.05763.
Younger, A. S., Hochreiter, S., & Conwell, P. R. (2001). Meta-learning with backpropagation. In *International joint conference on neural networks (IJCNN'01)*. IEEE.