A Code illustration

A.1 EvoGrad code

EvoGrad update is simple to implement if we use *higher* library [2] for the perturbed parameters of different model copies. We show only the part that is relevant to the meta-update.

```
model_parameter = [i.detach() for i in get_func_params(model)]
theta_list = [[j + sigma * torch.sign(torch.randn_like(j))
               for j in model_parameter]
               for i in range(n_model_candidates)]
pred_list = [model_patched(feature_transformer(inputs), params=theta)
             for theta in theta_list]
loss_list = [criterion(pred, targets) for pred in pred_list]
weights = torch.softmax(-torch.stack(loss_list)/temperature, 0)
theta_updated = [sum(map(mul, theta, weights))
                 for theta in zip(*theta_list)]
preds_meta = model_patched(inputs_meta, params=theta_updated)
loss_meta = criterion(preds_meta, targets_meta)

meta_opt.zero_grad()
loss_meta.backward()
meta_opt.step()
```

A.2 $T_1 - T_2$ code – for comparison

For comparison with EvoGrad, we also show how online $T_1 - T_2$-style meta-learning is often implemented using so-called *fast weights*. This approach has been, for example, used in [1, 20]. The meta-update itself is concise, but it requires us to implement layers that support fast weights, which is a far more lengthy part.

```
preds = model(feature_transformer(inputs))
loss = criterion(preds, targets)
optimizer.zero_grad()
grads = torch.autograd.grad(
    loss, model.parameters(), create_graph=True)
for k, weight in enumerate(model.parameters()):
    weight.fast = weight - meta_lr * grads[k]

preds_meta = model(inputs_meta)
loss_meta = criterion(preds_meta, targets_meta)

meta_opt.zero_grad()
loss_meta.backward()
meta_opt.step()
```

We also show the definition of a linear layer that supports fast weights:

```
class Linear_fw(nn.Linear):
    def __init__(self, in_features, out_features, bias=True):
        super(Linear_fw, self).__init__(in_features, out_features,
                                        bias=bias)
        self.weight.fast = None
        self.bias.fast = None

    def forward(self, x):
        if self.weight.fast is not None and self.bias.fast is not None:
            out = F.linear(x, self.weight.fast, self.bias.fast)
        else:
```

```
            out = super(Linear_fw, self).forward(x)
        return out
```

Normally we would use simple `nn.Linear(in_features, out_features)`.

B How to select EvoGrad hyperparameters

EvoGrad as an algorithm has a few hyperparameters common to most evolutionary approaches: perturbation value σ, temperature τ and the number of model copies K. In practice we use only 2 models as it is enough and improves the efficiency. The other hyperparameter values can be selected relatively easily by looking at the training loss of the unperturbed model and the training loss of the perturbed models. The losses should be similar to each other, but not the same – we want to make sure the perturbed weights can still be successfully used. We have found that in practice value $\sigma = 0.001$ is reasonable. Once we have selected the value of σ, we can select the value of temperature τ which leads to reasonably different weights for the two (or more) model copies. In practice we have found $\tau = 0.05$ to be a value which leads to suitable weights. For example, 0.48 and 0.52 for two model copies could be considered reasonable, while 0.5001 and 0.4999 would be too similar. Note that in the special case of a 1-dimensional toy problem, suitable EvoGrad hyperparameters are different than what is useful for practical problems.

C Additional details

We include an algorithmic description of the details as well as additional description of the experimental settings for all five problems that we discuss in the paper.

C.1 Illustration using a 1-dimensional problem

We provide more detailed descriptions of how we perform both analyses. In the first analysis, we calculate the EvoGrad hypergradient estimate for 100 values of λ between 0 and 2, starting with 0.1 and ending with 2.0. In each case we perform 100 repetitions to obtain an estimate of the mean and standard deviation of the hypergradient, considering the stochastic nature of EvoGrad. Given a value of λ, the process of EvoGrad estimate can be summarized using Algorithm 1. As a reminder, we use training loss function $f_T(x, \lambda) = (x-1)^2 + \lambda\|x\|_2^2$ that includes a meta-parameter λ and validation loss function $f_V(x) = (x-0.5)^2$ that does not include the meta-parameter. The value of temperature is 0.5 and the number of model candidates varies between 2, 10 and 100.

Algorithm 1 EvoGrad hypergradient estimate for the 1D problem

1: **Input:** λ: target hyperparameter; K: number of model candidates; τ: temperature; f_T, f_V: training and validation loss functions
2: **Output:** g: hypergradient estimate
3: Sample $x \sim \mathcal{N}(0, 1)$
4: Sample K noise parameters $\epsilon_k \sim \mathcal{N}(0, 1)$ and use them to create $x_k = x + \epsilon_k$
5: Calculate losses $\ell_k = f_T(x_k, \lambda)$ for k between 1 and K
6: Calculate weights $w_1, w_2, \ldots, w_K = \mathrm{softmax}([-\ell_1, -\ell_2, \ldots, -\ell_K]/\tau)$
7: Calculate $x^* = w_1 x_1 + w_2 x_2 + \cdots + w_K x_K$
8: Calculate $\ell_V = f_V(x^*)$
9: Calculate hypergradient $g = \frac{\partial \ell_V}{\partial \lambda}$ by backpropagating through x^* computation

The second analysis evaluates the trajectories that values of x, λ take if we update them with SGD with the hypergradient estimated by EvoGrad compared to the ground-truth. We can summarize the process using Algorithm 2. When using the ground-truth hypergradient, we simply replace lines 6 to 10 by directly updating the value of λ using the closed-form formula for the hypergradient: $g(\lambda) = (\lambda - 1)/(\lambda + 1)^3$. We use 5 steps, learning rate of 0.1 and temperature 0.5.

Algorithm 2 Training with EvoGrad – 1D problem

Input: x_0, λ_0: initial values of x, λ; N: number of steps; α: learning rate; K: number of model candidates; τ: temperature; f_T, f_V: training and validation loss functions
Output: Optimized values of x, λ
Initialize $x = x_0$ and $\lambda = \lambda_0$
for i between 1 and N **do**
  Update $x \leftarrow x - \alpha \frac{\partial f_T(x,\lambda)}{\partial x}$
  Sample K noise parameters $\epsilon_k \sim \mathcal{N}(0, 1)$ and use them to create $x_k = x + \epsilon_k$
  Calculate losses $\ell_k = f_T(x_k, \lambda)$ for k between 1 and K
  Calculate weights $w_1, w_2, \ldots, w_K = \mathrm{softmax}([-\ell_1, -\ell_2, \ldots, -\ell_K]/\tau)$
  Calculate $x^* = w_1 x_1 + w_2 x_2 + \cdots + w_K x_K$
  Update $\lambda \leftarrow \lambda - \alpha \frac{\partial f_V(x^*)}{\partial \lambda}$
end for

C.2 Rotation transformation

As part of the rotation transformation problem, we try to prepare a model for the classification of rotated images. We use MNIST images [8] and train the base model with unrotated training images, while testing is done with images rotated by $30°$. We split the original training set to create a meta-validation set of size 10,000 with images rotated by $30°$.

To prepare the model for the target problem, we meta-learn a rotation transformation alongside training the base model – which we apply to the unrotated images. Our base model is LeNet [9] that has two CNN layers followed by three fully-connected layers. We use ReLU non-linearity and max-pooling. The base model is trained with Adam optimizer [5] with 0.001 learning rate, while the meta-parameter is optimized with Adam optimizer with learning rate of 0.01. We use a batch size of 128 and cross-entropy loss ℓ. EvoGrad parameters are $\tau = 0.05, \sigma = 0.001, K = 2$. We sample the noise parameters as $\boldsymbol{\epsilon}_k \sim \sigma\mathrm{sign}(\mathcal{N}(\mathbf{0}, \boldsymbol{I}))$, and we use this formulation also in the further practical meta-learning problems – it is a better-controlled version of simple $\mathcal{N}(\mathbf{0}, \sigma\boldsymbol{I})$. We train the models for 5 epochs and repeat the experiments 5 times. The algorithm is summarized in Algorithm 3.

Rotations are performed using a model with one learnable parameter λ (angle). The input that the model receives is rotated using matrix:

$$\begin{pmatrix} \cos(\lambda) & -\sin(\lambda) \\ \sin(\lambda) & \cos(\lambda) \end{pmatrix}.$$

Algorithm 3 Training with EvoGrad hypergradient – rotation transformation

Input: α: learning rate; β: meta-learning rate; σ: noise parameter; K: number of model candidates; τ: temperature
Output: $\boldsymbol{\theta}$: trained model; λ: rotation parameter
Initialize $\boldsymbol{\theta} \sim p(\boldsymbol{\theta})$ and $\lambda = 0$
while *training* **do**
  Sample minibatch of training x_t, y_t (standard) and validation x_v, y_v (rotated) examples
  Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \ell(f_{\boldsymbol{\theta}}(f_\lambda(x_t)), y_t)$
  Sample K noise parameters $\boldsymbol{\epsilon}_k \sim \sigma\mathrm{sign}(\mathcal{N}(\mathbf{0}, \boldsymbol{I}))$ and use them to create $\boldsymbol{\theta}_k = \boldsymbol{\theta} + \boldsymbol{\epsilon}_k$
  Calculate losses $\ell_k = \ell(f_{\boldsymbol{\theta}_k}(f_\lambda(x_t)), y_t)$ for k between 1 and K
  Calculate weights $w_1, w_2, \ldots, w_K = \mathrm{softmax}([-\ell_1, -\ell_2, \ldots, -\ell_K]/\tau)$
  Calculate $\boldsymbol{\theta}^* = w_1 \boldsymbol{\theta}_1 + w_2 \boldsymbol{\theta}_2 + \cdots + w_K \boldsymbol{\theta}_K$
  Update $\lambda \leftarrow \lambda - \beta \nabla_\lambda \ell(f_{\boldsymbol{\theta}^*}(x_v), y_v)$
end while

We compare our meta-learning approach to simple standard training that does not use the rotation transformer. In such case we keep the same settings as before and update the model simply as $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \ell(f_{\boldsymbol{\theta}}(x_t), y_t)$. The results prove EvoGrad is capable of meta-learning suitable values.

C.3 Cross-domain few-shot classification via learned feature-wise transformation

We extend the *Learning-to-Learn Feature-Wise Transformation* method from [20] to show the practical impact that EvoGrad can make. The goal of the LFT method is to make metric-based few-shot learners robust to domain shift. A detailed description of the LFT method is provided in [20], and here we describe the main changes that are needed to use EvoGrad for LFT. The key difference is that we do not backpropagate via standard model update that leads to higher memory and time consumption (we measure maximum allocated memory and time per epoch).

We summarize how EvoGrad is applied to LFT in Algorithm 4. A metric based model (we choose RelationNet [19]) includes feature encoder $E_{\boldsymbol{\theta}_e}$ and metric function $M_{\boldsymbol{\theta}_m}$. Feature transformation layers parameterized by $\boldsymbol{\theta}_f = \{\boldsymbol{\theta}_\gamma, \boldsymbol{\theta}_\beta\}$ are integrated into the feature encoder to form $E_{\boldsymbol{\theta}_e, \boldsymbol{\theta}_f}$. Similarly as [20], we sample pseudo-seen $\mathcal{T}^{\text{ps}}$ and pseudo-unseen $\mathcal{T}^{\text{pu}}$ domains from the seen domains $\{\mathcal{T}_1^{\text{seen}}, \mathcal{T}_2^{\text{seen}}, \cdots, \mathcal{T}_n^{\text{seen}}\}$. In each step, we sample pseudo-seen and pseudo-unseen few-shot learning tasks that both include support and query examples. The pseudo-seen task is described as $T^{\text{ps}} = \left\{\left(\mathcal{X}_s^{\text{ps}}, \mathcal{Y}_s^{\text{ps}}\right), \left(\mathcal{X}_q^{\text{ps}}, \mathcal{Y}_q^{\text{ps}}\right)\right\} \in \mathcal{T}^{\text{ps}}$ and the pseudo-unseen task is $T^{\text{pu}} = \left\{\left(\mathcal{X}_s^{\text{pu}}, \mathcal{Y}_s^{\text{pu}}\right), \left(\mathcal{X}_q^{\text{pu}}, \mathcal{Y}_q^{\text{pu}}\right)\right\} \in \mathcal{T}^{\text{pu}}$, for task examples $\mathcal{X}$ with labels $\mathcal{Y}$.

We have used the exact same set-up as [20] with their official implementation (for RelationNet), so we only describe the additional settings that are unique to us. In particular, EvoGrad-specific parameters are $\tau = 0.05, K = 2, \sigma = 0.001$ (we have used σ equal to the learning rate). We have used ResNet-10 [3] backbone for direct comparison with [20]. The datasets that we use are processed in the same way as done by [20], and they are MiniImagenet [16], CUB [21], Cars [6], Places [23] and Plantae [4].

In order to use ResNet-34, we have trained a new ResNet-34 baseline model on MiniImagenet [16] per [20] instructions. We use the same hyperparameters as were used for ResNet-10, which also means that when using fixed feature transformation layers, we use $\theta_\gamma = 0.3, \theta_\beta = 0.5$. Note that ResNet-34 ran out of memory for the original second-order LFT approach on 5-way 5-shot task with 16 query examples when using standard GPU with 12 GB GPU memory. If we wanted to use this model also for the second-order approach, we would need to decrease the number of examples in the task appropriately. However, with EvoGrad we do not need to make this compromise and overall it means that EvoGrad scales also to problems where the original second-order approach does not scale because of GPU memory limitations.

Algorithm 4 Learning-to-learn feature-wise transformation – with EvoGrad

Input: $\{\mathcal{T}_1^{\text{seen}}, \mathcal{T}_2^{\text{seen}}, \cdots, \mathcal{T}_n^{\text{seen}}\}$: seen domains; α: learning rate; σ: noise parameter; K: number of model candidates; τ: temperature
Output: $\boldsymbol{\theta}_e$: feature extractor; $\boldsymbol{\theta}_m$: metric learner; $\boldsymbol{\theta}_f$: feature transformation layers
Initialize $\boldsymbol{\theta}_e, \boldsymbol{\theta}_m, \boldsymbol{\theta}_f \sim p(\boldsymbol{\theta}_e), p(\boldsymbol{\theta}_m), p(\boldsymbol{\theta}_f)$
while *training* **do**
  Randomly sample non-overlapping pseudo-seen $\mathcal{T}^{\text{ps}}$ and pseudo-unseen $\mathcal{T}^{\text{pu}}$ domains from the seen domains
  Sample a pseudo-seen task $T^{\text{ps}} \in \mathcal{T}^{\text{ps}}$ and a pseudo-unseen task $T^{\text{pu}} \in \mathcal{T}^{\text{pu}}$
  // Standard update of the metric-based model with pseudo-seen task:
  Update $\boldsymbol{\theta}_e, \boldsymbol{\theta}_m \leftarrow \boldsymbol{\theta}_e, \boldsymbol{\theta}_m - \alpha \nabla_{(\boldsymbol{\theta}_e, \boldsymbol{\theta}_m)} \ell\left(M_{\boldsymbol{\theta}_m}\left(\mathcal{Y}_s^{\text{ps}}, E_{\boldsymbol{\theta}_e, \boldsymbol{\theta}_f}\left(\mathcal{X}_s^{\text{ps}}\right), E_{\boldsymbol{\theta}_e, \boldsymbol{\theta}_f}\left(\mathcal{X}_q^{\text{ps}}\right)\right), \mathcal{Y}_q^{\text{ps}}\right)$
  // EvoGrad computations:
  Sample K noise parameters $\left\{\boldsymbol{\epsilon}_e^{(k)}, \boldsymbol{\epsilon}_m^{(k)}\right\}_{k=1}^{K} \sim \sigma \text{sign}(\mathcal{N}(\mathbf{0}, \boldsymbol{I}))$
  Create $\boldsymbol{\theta}_e^{(k)} = \boldsymbol{\theta}_e + \boldsymbol{\epsilon}_e^{(k)}$ and $\boldsymbol{\theta}_m^{(k)} = \boldsymbol{\theta}_m + \boldsymbol{\epsilon}_m^{(k)}$ for k between 1 and K
  Calculate losses $\ell_k = \ell\left(M_{\boldsymbol{\theta}_m^{(k)}}\left(\mathcal{Y}_s^{\text{ps}}, E_{\boldsymbol{\theta}_e^{(k)}, \boldsymbol{\theta}_f}\left(\mathcal{X}_s^{\text{ps}}\right), E_{\boldsymbol{\theta}_e^{(k)}, \boldsymbol{\theta}_f}\left(\mathcal{X}_q^{\text{ps}}\right)\right), \mathcal{Y}_q^{\text{ps}}\right)$
  Calculate weights $w_1, w_2, \ldots, w_K = \text{softmax}([-\ell_1, -\ell_2, \ldots, -\ell_K]/\tau)$
  Calculate $\boldsymbol{\theta}_e^* = w_1 \boldsymbol{\theta}_e^{(1)} + w_2 \boldsymbol{\theta}_e^{(2)} + \cdots + w_K \boldsymbol{\theta}_e^{(K)}$
  Calculate $\boldsymbol{\theta}_m^* = w_1 \boldsymbol{\theta}_m^{(1)} + w_2 \boldsymbol{\theta}_m^{(2)} + \cdots + w_K \boldsymbol{\theta}_m^{(K)}$
  // Update feature-wise transformation layers with pseudo-unseen task:
  Update $\boldsymbol{\theta}_f \leftarrow \boldsymbol{\theta}_f - \alpha \nabla_{\boldsymbol{\theta}_f} \ell\left(M_{\boldsymbol{\theta}_m^*}\left(\mathcal{Y}_s^{\text{pu}}, E_{\boldsymbol{\theta}_e^*}\left(\mathcal{X}_s^{\text{pu}}\right), E_{\boldsymbol{\theta}_e^*}\left(\mathcal{X}_q^{\text{pu}}\right)\right), \mathcal{Y}_q^{\text{pu}}\right)$
end while

C.4 Label noise with Meta-Weight-Net

We use the experimental set-up from [18] for the label noise experiments, together with their official implementation. The label noise experiments use ResNet-32 model and 60 epochs, each of which has 500 iterations. CIFAR-10 and CIFAR-100 [7] datasets are used. Meta-Weight-Net is represented by a neural network with two linear layers with hidden size of 300 units, ReLU nonlinearity in between and sigmoid output unit. Meta-Weight-Net weights instance-wise losses for each example in the minibatch, which are then combined together by taking their sum. EvoGrad specific parameters are $\tau = 0.05, K = 2, \sigma = 0.001$. The level of label noise depends on the specific scenario considered – 40%, 20% or 0%.

We provide an overview of the EvoGrad approach applied to the label noise with Meta-Weight-Net problem in Algorithm 5. Even though we do the standard update using noisy examples after the meta-update, the order could be swapped and we simply follow the order chosen by [18]. Detailed explanations are provided in [18], we only explain how we modify the method to use EvoGrad. Note that we do not rerun the baseline experiments and we directly take the reported values from the Meta-Weight-Net paper [18]. However, we do our own rerun of standard second-order Meta-Weight-Net to get memory and runtime statistics.

Algorithm 5 Meta-Weight-Net for label noise – with EvoGrad

Input: α: learning rate; σ: noise parameter; K: number of model candidates; τ: temperature
Output: $\boldsymbol{\theta}$: trained model; $\boldsymbol{\omega}$: Meta-Weight-Net parameters
Initialize $\boldsymbol{\theta}, \boldsymbol{\omega} \sim p(\boldsymbol{\theta}), p(\boldsymbol{\omega})$
while *training* **do**
  Sample minibatch of training x_t, y_t (noisy) and validation x_v, y_v (clean) examples
  // EvoGrad update:
  Sample K noise parameters $\boldsymbol{\epsilon}_k \sim \sigma \text{sign}(\mathcal{N}(\mathbf{0}, \boldsymbol{I}))$ and use them to create $\boldsymbol{\theta}_k = \boldsymbol{\theta} + \boldsymbol{\epsilon}_k$
  Calculate losses $\ell_k = f_{\boldsymbol{\omega}}\left(\ell(f_{\boldsymbol{\theta}_k}(x_t), y_t)\right)$ for k between 1 and K
  Calculate weights $w_1, w_2, \ldots, w_K = \text{softmax}([-\ell_1, -\ell_2, \ldots, -\ell_K]/\tau)$
  Calculate $\boldsymbol{\theta}^* = w_1\boldsymbol{\theta}_1 + w_2\boldsymbol{\theta}_2 + \cdots + w_K\boldsymbol{\theta}_K$
  Update $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \alpha\nabla_{\boldsymbol{\omega}}\ell(f_{\boldsymbol{\theta}^*}(x_v), y_v)$
  // Standard update using noisy examples and MWN:
  Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla_{\boldsymbol{\theta}} f_{\boldsymbol{\omega}}\left(\ell(f_{\boldsymbol{\theta}}(x_t), y_t)\right)$
end while

In addition, we provide further details about Meta-Weight-Net scalability analyses. We have chosen MWN to conduct these analyses because it represents a real problem where meta-learning is helpful, yet the memory consumption and time requirements are small enough to allow us to easily evaluate scaling up of the numbers of parameters. All Meta-Weight-Net scalability experiments are repeated 5 times, but we do not run them fully – we only do 10 epochs to get estimates of the time per epoch.

We have provided the main results that evaluate the impact of using a model with significantly more parameters in the main part of the paper. Here we provide additional figures. Figure 1 shows the impact of variable number of meta-parameters (number of hidden units in MWN). We can see the number of meta-parameters does not significantly impact the memory usage or runtime. This is likely because we use reverse-mode backpropagation that becomes more expensive with more model parameters and not hyperparameters [14]. Further, the number of meta-parameters still remains small compared to the size of the model. Figure 2 shows the number of model copies does not lead to increased memory consumption, perhaps because we only keep the model weights in memory and not also many intermediate variables like activations that are needed for backpropagation – backpropagation is significantly more expensive in terms of memory than forward propagation [15]. The runtime increases slightly with additional model copies, which comes from the need to calculate additional forward propagations.

C.5 Low-resource cross-lingual learning with MetaXL

MetaXL [22] is an approach that meta-learns meta representation transformation to improve transfer in low-resource cross-lingual learning. We show how EvoGrad is applied to MetaXL in Algorithm 6

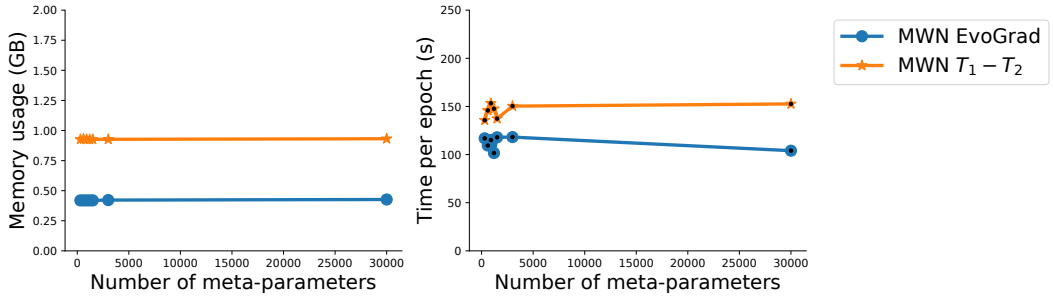

Figure 1: Memory and time scaling of MWN EvoGrad vs original second-order Meta-Weight-Net – when changing the number of learnable hyperparameters (meta-parameters). The number of meta-parameters does not noticeably influence the memory usage and time per epoch in this case.

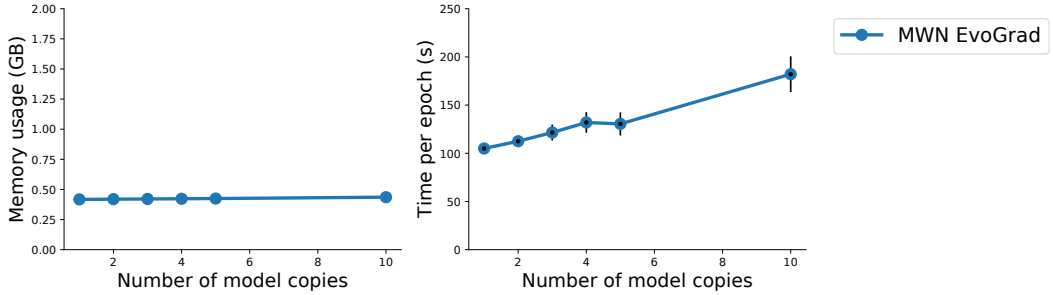

Figure 2: Memory and time scaling of MWN EvoGrad – when using different numbers of model copies. A larger number of model copies does not increase the memory usage in this case, but it leads to a larger time per epoch.

In order to do experiments, we have taken the official code provided by [22] and tried to replicate their experiments as closely as possible. We used the named entity recognition (NER) task with English source language. The only change we made is a smaller batch size: 12 instead of 16 to fit into the memory of the largest GPUs that we have currently available. All details are described in [22]. For EvoGrad we have selected the same hyperparameters as for the other tasks in our paper (two model candidates, $\sigma = 0.001$ and $\tau = 0.05$). In order to make the implementation of EvoGrad on MetaXL simple, we have only applied noise perturbation on the top layer of the model. It is likely that in practice it is enough to only apply the noise to the top layer, which can make using EvoGrad very simple in most cases.

C.6 Datasets availability

All datasets that we use are freely available and their details are described in [20], [18] and [22] – including how to download them.

C.7 Computational resources

Illustration using a 1-dimensional problem and rotation transformation can be easily run on a laptop GPU. For cross-domain few-shot learning with LFT and label noise with MWN, we have used an internal cluster with NVIDIA GPUs - Titan X or P100 (all with 12GB GPU memory). For MetaXL we have used NVIDIA 3090 Ti GPUs with 24GB memory. When reporting the time and memory statistics we made sure to use the same model of GPU so that the comparisons are accurate. The experiments were allocated 14 GB RAM memory and 6 CPUs to allow for faster data loading (fewer resources would also be suitable).

Algorithm 6 MetaXL for cross-lingual learning – with EvoGrad

Input: α, β: learning rates; σ: noise parameter; K: number of model candidates; τ: temperature; D_t, D_s: input data from the target and source language
Output: $\boldsymbol{\theta}$: trained model; $\boldsymbol{\omega}$: representation transformation network
Initialize base model parameters $\boldsymbol{\theta}$ with pretrained XLM-R weights, initialize parameters of the representation transformation network $\boldsymbol{\omega}$ randomly
while *training* **do**
  Sample a source batch (x_s, y_s) from D_s and a target batch (x_t, y_t) from D_t
  // EvoGrad update:
  Sample K noise parameters $\boldsymbol{\epsilon}_k \sim \sigma\text{sign}(\mathcal{N}(\mathbf{0}, \boldsymbol{I}))$ and use them to create $\boldsymbol{\theta}_k = \boldsymbol{\theta} + \boldsymbol{\epsilon}_k$
  Calculate losses $\ell_k = \ell(f_{\boldsymbol{\omega} \circ \boldsymbol{\theta}_k}(x_s), y_s)$ for k between 1 and K
  Calculate weights $w_1, w_2, \ldots, w_K = \text{softmax}([-\ell_1, -\ell_2, \ldots, -\ell_K]/\tau)$
  Calculate $\boldsymbol{\theta}^* = w_1\boldsymbol{\theta}_1 + w_2\boldsymbol{\theta}_2 + \cdots + w_K\boldsymbol{\theta}_K$
  Update $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \beta\nabla_{\boldsymbol{\omega}}\ell(f_{\boldsymbol{\theta}^*}(x_t), y_t)$
  // Standard update using representation transformation network:
  Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\omega} \circ \boldsymbol{\theta}}(x_s), y_s)$
end while

D Evaluation of hypernetworks

We have evaluated hypernetworks [10] for cross-domain few-shot classification via learned feature-wise transformation, to find if the approach can be useful for recent meta-learning applications. To make the approach computationally viable, we have used hypernetworks with a bottleneck. For H hyperparameters, P model parameters and bottleneck size of B, our hypernetwork $\boldsymbol{\phi}$ consists of two layers, one with a weight matrix of $H \times B$, followed by $B \times P$ weight matrix, with sigmoid non-linearity in between. Note that B needs to be relatively small and directly using one layer with a weight matrix of $H \times P$ would require far more memory than normally available – for the considered problem. Following [10], we have used bottleneck size $B = 10$. We have used the exact same experimental set-up as in our other experiments.

Our results in Table 1 show hypernetworks fail to discover a good solution within the standard number of iterations used throughout, and their performance is poor. The results highlight that generating model parameters based on the hyperparameters may not be sufficient in more challenging and more realistic meta-learning problems. It also explains why hypernetworks are not commonly used in meta-learning applications.

Table 1: RelationNet test accuracies (%) and 95% confidence intervals across test tasks on various unseen datasets. 5-way 1-shot learning at the top and 5-way 5-shot learning at the bottom. Hypernetworks lead to significantly worse accuracies than $T_1 - T_2$ and EvoGrad, showing they fail to generate well-performing model parameters.

Scenario	CUB	Cars	Places	Plantae
LFT with hypernetworks	38.94 ± 0.57	30.10 ± 0.48	38.07 ± 0.58	33.83 ± 0.58
LFT with $T_1 - T_2$	46.03 ± 0.60	31.50 ± 0.49	49.29 ± 0.65	36.34 ± 0.59
LFT with EvoGrad	47.39 ± 0.61	32.51 ± 0.56	50.70 ± 0.66	36.00 ± 0.56
LFT with hypernetworks	56.91 ± 0.57	40.64 ± 0.56	56.08 ± 0.58	44.73 ± 0.57
LFT with $T_1 - T_2$	65.94 ± 0.56	43.88 ± 0.56	65.57 ± 0.57	51.43 ± 0.55
LFT with EvoGrad	64.63 ± 0.56	42.64 ± 0.58	66.54 ± 0.57	52.92 ± 0.57

E Comparison to more meta-learning approaches

In this section we provide an extended comparison of hypergradient approximations by various gradient-based meta-learners, similar to the analysis done in [11]. The approximations themselves are provided in Table 2, while the time and memory requirements are given in Table 3.

Table 2: Comparison of hypergradient approximations of different gradient-based meta-learning methods. Number of inner-loop steps is denoted by i. Note that also one-step approximation methods can be used once per i steps. $\boldsymbol{\theta}^*$ describes the optimal model parameters given $\boldsymbol{\lambda}$, while $\widehat{\boldsymbol{\theta}^*}$ represents their approximation.

Method	Hypergradient approximation
Unrolled diff. [13]	$\frac{\partial \ell_V}{\partial \boldsymbol{\lambda}} - \frac{\partial \ell_V}{\partial \boldsymbol{\theta}} \times \sum_{j \leq i} \left[\prod_{k<j} I - \frac{\partial^2 \ell_T}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}^T} \Big\vert_{\boldsymbol{\theta}_{i-k}} \right] \frac{\partial^2 \ell_T}{\partial \boldsymbol{\theta} \partial \boldsymbol{\lambda}^T} \Big\vert_{\boldsymbol{\theta}_{i-j}}$
K-step truncated unrolled diff. [17]	$\frac{\partial \ell_V}{\partial \boldsymbol{\lambda}} - \frac{\partial \ell_V}{\partial \boldsymbol{\theta}} \times \sum_{K \leq j \leq i} \left[\prod_{k<j} I - \frac{\partial^2 \ell_T}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}^T} \Big\vert_{\boldsymbol{\theta}_{i-k}} \right] \frac{\partial^2 \ell_T}{\partial \boldsymbol{\theta} \partial \boldsymbol{\lambda}^T} \Big\vert_{\boldsymbol{\theta}_{i-j}}$
$T_1 - T_2$ [12]	$\frac{\partial \ell_V}{\partial \boldsymbol{\lambda}} - \frac{\partial \ell_V}{\partial \boldsymbol{\theta}} \times [I]^{-1} \frac{\partial^2 \ell_T}{\partial \boldsymbol{\theta} \partial \boldsymbol{\lambda}^T} \Big\vert_{\widehat{\boldsymbol{\theta}^*}(\boldsymbol{\lambda})}$
Hypernetworks [10]	$\frac{\partial \ell_V}{\partial \boldsymbol{\lambda}} + \frac{\partial \ell_V}{\partial \boldsymbol{\theta}} \times \frac{\partial \boldsymbol{\theta}^*_{\boldsymbol{\phi}}}{\partial \boldsymbol{\lambda}}$ where $\boldsymbol{\theta}^*_{\boldsymbol{\phi}}(\boldsymbol{\lambda}) = \arg\min_{\boldsymbol{\phi}} \ell_T \left(\boldsymbol{\lambda}, \boldsymbol{\theta}_{\boldsymbol{\phi}}(\boldsymbol{\lambda}) \right)$
Exact IFT [11]	$\frac{\partial \ell_V}{\partial \boldsymbol{\lambda}} - \frac{\partial \ell_V}{\partial \boldsymbol{\theta}} \times \left[\frac{\partial^2 \ell_T}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}^T} \right]^{-1} \frac{\partial^2 \ell_T}{\partial \boldsymbol{\theta} \partial \boldsymbol{\lambda}^T} \Big\vert_{\boldsymbol{\theta}^*(\boldsymbol{\lambda})}$
Neumann IFT [11]	$\frac{\partial \ell_V}{\partial \boldsymbol{\lambda}} - \frac{\partial \ell_V}{\partial \boldsymbol{\theta}} \times \left(\sum_{j<i} \left[I - \frac{\partial^2 \ell_T}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}^T} \right]^j \right) \frac{\partial^2 \ell_T}{\partial \boldsymbol{\theta} \partial \boldsymbol{\lambda}^T} \Big\vert_{\widehat{\boldsymbol{\theta}^*}(\boldsymbol{\lambda})}$
EvoGrad (ours)	$\frac{\partial \ell_V}{\partial \boldsymbol{\lambda}} + \frac{\partial \ell_V}{\partial \boldsymbol{\theta}} \times \mathcal{E} \frac{\partial \boldsymbol{w}}{\partial \boldsymbol{\ell}} \frac{\partial \boldsymbol{\ell}}{\partial \boldsymbol{\lambda}} = \frac{\partial \ell_V}{\partial \boldsymbol{\lambda}} + \frac{\partial \ell_V}{\partial \boldsymbol{\theta}} \times \mathcal{E} \frac{\partial \operatorname{softmax}(-\boldsymbol{\ell})}{\partial \boldsymbol{\lambda}} \Big\vert_{\widehat{\boldsymbol{\theta}^*}(\boldsymbol{\lambda})}$

Table 3: Comparison of asymptotic time and memory requirements of EvoGrad and other gradient-based meta-learners. P is the number of model parameters, H is the number of hyperparameters, I is the number of inner-loop steps, N is the number of model copies in EvoGrad. Note this is a first-principles analysis, so the time requirements are different when using e.g. reverse-mode backpropagation that uses parallelization.

Method	Time requirements	Memory requirements
Unrolled diff. [13]	$\mathcal{O}(IP^2 + PH)$	$\mathcal{O}(PI + H)$
K-step truncated unrolled diff. [17]	$\mathcal{O}(KP^2 + PH)$	$\mathcal{O}(PK + H)$
$T_1 - T_2$ [12]	$\mathcal{O}(PH)$	$\mathcal{O}(P + H)$
Linear hypernetworks [10]	$\mathcal{O}(PH)$	$\mathcal{O}(PH)$
Neumann IFT [11]	$\mathcal{O}(P^2 + PH)$	$\mathcal{O}(P + H)$
EvoGrad (ours)	$\mathcal{O}(NP + H)$	$\mathcal{O}(P + H)$

References

[1] Chen, W.-Y., Liu, Y.-C., Kira, Z., Tech, G., Wang, Y.-C. F., Huang, J.-B., and Tech, V. (2019). A closer look at few-shot classification. In *ICLR*.

[2] Grefenstette, E., Amos, B., Yarats, D., Htut, P. M., Molchanov, A., Meier, F., Kiela, D., Cho, K., and Chintala, S. (2019). Generalized inner loop meta-learning. In *arXiv*.

[3] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. In *CVPR*.

[4] Horn, G. V., Aodha, O. M., Song, Y., Cui, Y., Sun, C., Shepard, A., Adam, H., Perona, P., and Belongie, S. (2018). The iNaturalist species classification and detection dataset. In *CVPR*.

[5] Kingma, D. P. and Ba, J. (2015). Adam: a method for stochastic optimization. In *ICLR*.

[6] Krause, J., Stark, M., Deng, J., and Fei-Fei, L. (2013). 3D object representations for fine-grained categorization. In *ICCV Workshops*.

[7] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report.

[8] LeCun, Y., Cortes, C., and Burges, C. (1998). MNIST handwritten digit database.

[9] LeCun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E., and Hubbard, W. (1989). Handwritten digit recognition: applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46.

[10] Lorraine, J. and Duvenaud, D. (2018). Stochastic hyperparameter optimization through hypernetworks. In *arXiv*.

[11] Lorraine, J., Vicol, P., and Duvenaud, D. (2020). Optimizing millions of hyperparameters by implicit differentiation. In *AISTATS*.

[12] Luketina, J., Berglund, M., Klaus Greff, A., and Raiko, T. (2016). Scalable gradient-based tuning of continuous regularization hyperparameters. In *ICML*.

[13] Maclaurin, D., Duvenaud, D., and Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. In *ICML*.

[14] Micaelli, P. and Storkey, A. (2019). Zero-shot knowledge transfer via adversarial belief matching. In *NeurIPS*.

[15] Rajeswaran, A., Finn, C., Kakade, S., and Levine, S. (2019). Meta-learning with implicit gradients. In *NeurIPS*.

[16] Ravi, S. and Larochelle, H. (2017). Optimization as a model for few-shot learning. In *ICLR*.

[17] Shaban, A., Cheng, C.-A., Hatch, N., and Boots, B. (2019). Truncated back-propagation for bilevel optimization. In *AISTATS*.

[18] Shu, J., Xie, Q., Yi, L., Zhao, Q., Zhou, S., Xu, Z., and Meng, D. (2019). Meta-Weight-Net: learning an explicit mapping for sample weighting. In *NeurIPS*.

[19] Sung, F., Yang, Y., Zhang, L., Xiang, T., Torr, P. H., and Hospedales, T. M. (2018). Learning to compare: relation network for few-shot learning. In *CVPR*.

[20] Tseng, H.-Y., Lee, H.-Y., Huang, J.-B., and Yang, M.-H. (2020). Cross-domain few-shot classification via learned feature-wise transformation. In *ICLR*.

[21] Welinder, P., Branson, S., Mita, T., Wah, C., Schroff, F., Belongie, S., and Perona, P. (2010). Caltech-UCSD Birds 200. Technical report, California Institute of Technology.

[22] Xia, M., Zheng, G., Mukherjee, S., Shokouhi, M., Neubig, G., and Awadallah, A. H. (2021). MetaXL: meta representation transformation for low-resource cross-lingual learning. In *NAACL*.

[23] Zhou, B., Lapedriza, A., Khosla, A., Oliva, A., and Torralba, A. (2018). Places: a 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(6):1452–1464.