LEARNING UNSUPERVISED LEARNING RULES

Anonymous authors
Paper under double-blind review

ABSTRACT

A major goal of unsupervised learning is to discover data representations that are useful for subsequent tasks, *without* access to supervised labels during training. Typically, this goal is approached by minimizing a surrogate objective, such as the negative log likelihood of a generative model, with the hope that representations useful for subsequent tasks will arise incidentally. In this work, we propose instead to directly target a later desired task by meta-learning an unsupervised learning rule, which leads to representations useful for that task. Here, our desired task (meta-objective) is the performance of the representation on semi-supervised classification, and we meta-learn an algorithm – an unsupervised weight update rule – that produces representations that perform well under this meta-objective. Additionally, we constrain our unsupervised update rule to a be a biologically-motivated, neuron-local function, which enables it to generalize to novel neural network architectures. We show that the meta-learned update rule produces useful features and sometimes outperforms existing unsupervised learning techniques. We further show that the meta-learned unsupervised update rule generalizes to train networks with different widths, depths, and nonlinearities. It also generalizes to train on data with randomly permuted input dimensions and even generalizes from image datasets to a text task.

1 INTRODUCTION

Supervised learning has proven extremely effective for many problems where large amounts of labeled training data are available. There is a common hope that unsupervised learning will prove similarly powerful in situations where labels are expensive, impractical to collect, or where the prediction target is unknown during training. Unsupervised learning however has yet to fulfill this promise. One explanation for this failure is that unsupervised representation learning algorithms are typically mismatched to the target task. Ideally, learned representations should linearly expose high level attributes of data (e.g. object identity) and perform well in semi-supervised settings. Many current unsupervised objectives, however, optimize for objectives such as log-likelihood of a generative model or reconstruction error and produce useful representations only as a side effect.

Unsupervised representation learning seems uniquely suited for meta-learning (Hochreiter et al., 2001; Schmidhuber, 1995). Unlike most tasks where meta-learning is applied, unsupervised learning does not define an explicit objective, which makes it impossible to phrase the task as a standard optimization problem. It is possible, however, to directly express a *meta-objective* that captures the quality of representations produced by an unsupervised update rule by evaluating the usefulness of the representation for candidate tasks, e.g. semi-supervised classification. In this work, we propose to *meta-learn* an unsupervised update rule by meta-training on a meta-objective that directly optimizes the utility of the unsupervised representation. Unlike hand-designed unsupervised learning rules, this meta-objective directly targets the usefulness of a representation generated from unlabeled data for later supervised tasks.

By recasting unsupervised representation learning as meta-learning, we treat the creation of the unsupervised update rule as a transfer learning problem. Instead of learning transferable features, we learn a transferable learning rule which does not require access to labels and generalizes across both data domains and neural network architectures. Although we focus on the meta-objective of semi-supervised classification here, in principle a learning rule could be optimized to generate representations for *any* subsequent task.

2 RELATED WORK

2.1 UNSUPERVISED REPRESENTATION LEARNING

There is no single accepted definition of unsupervised learning, and it is a topic of broad and diverse interest. Here we briefly present several techniques that can lead to a useful latent representation of a dataset. In contrast to our work, each method imposes a manually defined training algorithm or loss function to optimize whereas we *learn* the algorithm that creates useful representations as determined by a meta-objective.

Autoencoders (Hinton and Salakhutdinov, 2006) work by first compressing and optimizing reconstruction loss. Extensions have been made to de-noise data (Vincent et al., 2008; 2010), as well as compress information in an information theoretic way (Kingma and Welling, 2013). Le et al. (2011) further explored scaling up these unsupervised methods to large image datasets.

Generative adversarial networks (Goodfellow et al., 2014) take another approach to unsupervised feature learning. Instead of a loss function, an explicit min-max optimization is defined to learn a generative model of a data distribution. Recent work has shown that this training procedure can learn unsupervised features useful for few shot learning (Radford et al., 2015; Donahue et al., 2016; Dumoulin et al., 2016).

Other techniques rely on self-supervision where labels are easily generated to create a non-trivial 'supervised' loss. Domain knowledge of the input is often necessary to define these losses. Noroozi and Favaro (2016) use unscrambling jigsaw-like crops of an image. Techniques like Misra et al. (2016) and Sermanet et al. (2017) rely on using temporal ordering from videos.

Another approach to unsupervised learning relies on feature space design such as clustering. Coates and Ng (2012) showed that k-means can be used for feature learning. Xie et al. (2016) jointly learn features and cluster assignments. Bojanowski and Joulin (2017) develop a scalable technique to cluster by predicting noise. Other techniques such as Schmidhuber (1992), Hochreiter and Schmidhuber (1999), and Olshausen and Field (1997) define various desirable properties about the latent representation of the input, such as predictability, complexity of encoding mapping, independence, or sparsity, and optimize to achieve these properties.

2.2 META LEARNING

Most meta-learning algorithms consist of two levels of learning, or 'loops' of computation: an *inner loop*, where some form of learning occurs (e.g. an optimization process), and an *outer loop* or *meta-training* loop, which optimizes some aspect of the inner loop. We call the aspects of the inner loop modified by the outer loop *meta-parameters*. The performance of the inner loop computation is quantified by a *meta-objective*. *Meta-optimization* is then the process of adjusting the meta-parameters so that the inner loop performs well on this meta-objective. Meta-learning approaches differ by the computation performed in the inner loop, the domain, the choice of meta-parameters, and the method of optimizing the outer loop.

Some of the earliest work in meta-learning includes (Schmidhuber, 1987) which explores a variety of meta-learning and self-referential algorithms. Similarly to our algorithm Bengio et al. (1990; 1992) propose to learn a neuron local learning rule, though their approach differs in task and problem formulation. Additionally, Runarsson and Jonsson (2000) meta-learn supervised learning rules which mix local and global network information. A number of papers propose meta-learning for few shot learning (Vinyals et al., 2016; Ravi and Larochelle, 2016; Mishra et al., 2017; Finn et al., 2017; Snell et al., 2017), though these do not take advantage of unlabeled data. In (Garg and Kalai, 2017) meta-learning is used for unsupervised learning, primarily in the context of clustering and with a small number of meta-parameters. To allow easy comparison against other existing approaches, we present a more extensive survey of previous work in meta-learning in table form in Appendix A, highlighting differences in choice of task, structure of the meta-learning problem, choice of meta-architecture, and choice of domain.

To our knowledge, we are the first meta-learning approach to tackle the problem of unsupervised representation learning, where the inner loop consists of unsupervised learning. This contrasts with transfer learning, where a neural network is instead trained on a similar dataset, and then fine tuned or

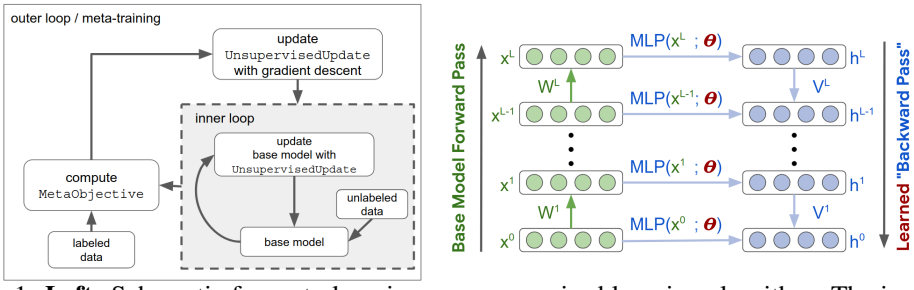

Figure 1: **Left:** Schematic for meta-learning an unsupervised learning algorithm. The inner loop computation consists of iteratively applying the UnsupervisedUpdate to a base model. During meta-training the UnsupervisedUpdate (parameterized by θ) is itself updated by gradient descent on the MetaObjective. **Right:** Schematic of the base model and UnsupervisedUpdate. Unlabeled input data, x_0, is passed through the base model, which is parameterised by W and colored green. The goal of the UnsupervisedUpdate is to modify W to achieve a top layer representation x^L which performs well at few-shot learning. In order to train the base model, information is propagated backwards by the UnsupervisedUpdate in a manner analogous to backprop. Unlike in backprop however, the backward weights V are decoupled from the forward weights W, and there is no explicit error signal. Instead at each layer, and for each neuron, a learning signal is injected by a meta-learned MLP parameterized by θ, with hidden state h. Weight updates are again analogous to those in backprop, and depend on the hidden state of the pre- and post- synaptic neurons for each weight.

otherwise post-processed on the target dataset. We additionally believe we are the first representation meta-learning approach to generalize across input data *modalities* as well as datasets, the first to generalize across permutation of the input dimensions, and the first to generalize across neural network architectures (e.g. layer width, network depth, activation function).

3 Model Design

We consider a multilayer perceptron (MLP) $f(\cdot;\phi_t)$, with parameters ϕ_t, as the *base model*. The inner loop of our meta-learning process trains this base model via iterative application of our learned optimizer. See Figure 1 for a schematic illustration and Appendix B for a more detailed diagram.

In standard supervised learning, the 'learned' optimizer is stochastic gradient descent (SGD). A supervised loss $l(x, y)$ is associated with this model, where x is a minibatch of inputs, and y are the corresponding labels. The parameters ϕ_t of the base model are then updated iteratively by performing SGD using the gradient $\frac{\partial l(x,y)}{\partial \phi_t}$. This supervised update rule can be written as $\phi_{t+1} = \text{SupervisedUpdate}(\phi_t, x_t, y_t; \theta)$. Here θ are the meta-parameters of the optimizer, which consist of hyper-parameters such as learning rate and momentum.

In this work, our learned optimizer is a parametric update process which does not depend on label information, $\phi_{t+1} = \text{UnsupervisedUpdate}(\phi_t, x_t; \theta)$. This form of the update rule is general. It encompasses many unsupervised learning algorithms and all methods in Section 2.1.

In traditional unsupervised learning algorithms, expert knowledge or a simple hyper-parameter search determines θ, which consists of a handful of meta-parameters such as learning rate and regularization constants. In contrast, our update rule will have orders of magnitude more meta-parameters, including the weights of a neural network. We train these meta-parameters by performing SGD on the sum of the MetaObjective over the course of (inner loop) training in order to find optimal parameters θ^*,

$$\theta^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \mathbb{E}_{\text{task}} \left[\sum_t \text{MetaObjective}(\phi_t(\theta)) \right], \tag{1}$$

that minimize the meta-objective over a set of training tasks.

In the following sections, we briefly review the main functional pieces to this model: the base model $f(\cdot, \phi)$, the UnsupervisedUpdate, and the MetaObjective. See the Appendix for a complete specification. Additionally, code and meta-trained parameters θ for our meta-learned UnsupervisedUpdate will be available.

3.1 BASE MODEL: $f(\cdot;\phi)$

Our base model consists of a standard fully connected multi-layer perceptron (MLP), with batch normalization (Ioffe and Szegedy, 2015), and ReLU nonlinearities. We chose this as opposed to a convolutional model to limit the inductive bias hard-coded in favor of learned behavior from the UnsupervisedUpdate. We call the pre-nonlinearity activations $z^1, \cdots, z^L$, and post-nonlinearity activations $x^0, \cdots, x^L$, where L is the total number of layers, and $x^0 \equiv x$ is the network input (raw data). The parameters are $\phi = \left\{W^1, b^1, V^1, \cdots, W^L, b^L, V^L\right\}$, where W^l and b^l are the weights and biases (applied after batch norm) for layer l, and V^l are the corresponding weights used in the backward pass.

3.2 LEARNED UPDATE RULE: UNSUPERVISEDUPDATE$(\cdot;\theta)$

We wish for our update rule to generalize across architectures with different widths, depths, or even network topologies. To achieve this, we design our update rule to be *neuron-local*, so that updates are a function of pre- and post- synaptic neurons in the base model, and are defined for any base model architecture. This has the added benefit that it makes the weight updates more similar to synaptic updates in biological neurons, which depend almost exclusively on the pre- and post-synaptic neurons for each synapse (Whittington and Bogacz, 2017).

To build these updates, each neuron i in every layer l in the base model has an MLP, referred to as an update network, associated with it, with output $h_i^l(\cdot;\theta)$. All update networks share meta-parameters θ, and $h_i^l(\cdot;\theta)$ is evaluated only during unsupervised training as the update networks are part of the UnsupervisedUpdate, and not part of the base model. Evaluating the statistics of unit activation over a batch of data has proven helpful in supervised learning (Ioffe and Szegedy, 2015). It has similarly proven helpful in hand-designed unsupervised learning rules, such as sparse coding and clustering. We therefore allow $h_i^l(\cdot;\theta)$ to accumulate statistics across examples in each training minibatch.

During an unsupervised training step, the base model is first run in a standard feed-forward fashion, populating x_{ib}^l, z_{ib}^l, where b is the training minibatch index. As in supervised learning, an error signal δ_{ib}^l is then propagated backwards through the network. Unlike in supervised backprop however, this error signal is generated by the corresponding update network for each unit, $\delta_{ib}^l \leftarrow h_i^l(\cdot;\theta)$, and propogated backward using a set of learned 'backward weights' $(V^l)^T$, as opposed to the transpose of the forward weights $(W^l)^T$ as would be the case in backprop. Pictorially this can be seen in Figure 1.

Again as in supervised learning, the weight updates are a product of pre- and post-synaptic signals. Unlike in supervised learning however, these signals are generated by the per-neuron update networks: $\Delta W_{ij}^l = \sum_b c_{ib}^l d_{jb}^{l-1}$, where $\{c_{ib}^l, d_{ib}^l\} \leftarrow h_i^l(\cdot;\theta)$.

The inputs to the update network consists of unit pre- and post-activations, and backwards propagated error signal: $h_i^l\left(x_{i\cdot}^l, z_{i\cdot}^l, \left[\left(V^{l+1}\right)^T \delta^{l+1}\right]_{i\cdot}; \theta\right)$. The backward weights also are updated with the same update network. For more information on the structure of h_i^l, see Appendix F.

3.3 METAOBJECTIVE (ϕ)

The meta-objective determines the quality of the unsupervised representations. In order to meta-train via SGD, this loss must be differentiable. The meta-objective we use in this work is based on fitting a linear regression to labeled examples with a small number of data points. In order to encourage the learning of features that generalize well, we estimate the linear regression weights on one minibatch $\{x_a, y_a\}$ of K data points, and evaluate the classification performance on a second minibatch $\{x_b, y_b\}$ also with K datapoints,

$$\hat{v} = \underset{v}{\operatorname{argmin}} \left(\left\|y_a - v^T u_a\right\|^2 + \lambda \|v\|^2\right), \qquad \text{MetaObjective}(\cdot;\phi) = \operatorname{CosDist}\left(y_b, \hat{v}^T u_b\right), \tag{2}$$

where u_a, u_b are features extracted from the base model on data x_a, x_b, respectively. The target labels y_a, y_b consist of one hot encoded labels and potentially also regression targets from data augmentation (e.g. rotation angle, see Section 4.2). We found that using a cosine distance, $\operatorname{CosDist}$, rather than unnormalized squared error improved stability. Note this meta-objective is *only* used

during meta-training and *not* used when applying the learned update rule. The inner loop computation is performed without labels.

4 TRAINING THE UPDATE RULE

4.1 APPROXIMATE GRADIENT BASED TRAINING

We choose to meta-optimize via SGD as opposed to reinforcement learning or other black box method, due to the superior convergence properties of SGD in high dimensions, and the high dimensional nature of θ. Training and computing derivatives through long recurrent computation of this form is notoriously difficult (Pascanu et al., 2013). To improve stability and reduce the computational cost we approximate the gradients $\frac{\partial[\text{MetaObjective}]}{\partial\theta}$ via truncated backprop through time. Many additional design choices were also crucial to achieving stability and convergence in meta-learning, including the use of batch norm, and restricting the norm of the UnsupervisedUpdate update step. Meta-learning practitioners may find the discussion of these and other choices in Appendix C valuable.

4.2 META-TRAINING DISTRIBUTION AND GENERALIZATION

Generalization in our learned optimizer comes from both the form of the UnsupervisedUpdate (Section 3.2), and from the meta-training distribution. Our meta-training distribution is composed of both datasets and base model architectures.

We construct a set of training tasks consisting of CIFAR10 (Krizhevsky and Hinton, 2009) and multi-class classification from subsets of classes from Imagenet (Russakovsky et al., 2015) as well as from a dataset consisting of rendered fonts. We find that increased training dataset variation actually improves the meta-optimization process. To reduce computation we restrict the input data to 16x16 pixels or less during meta-training, and resize all datasets accordingly. For evaluation, we use MNIST(LeCun et al., 1998), Fashion MNIST(Xiao et al., 2017), IMDB(Maas et al., 2011), and a hold-out set of Imagenet classes. We additionally sample the base model architecture. We sample number of layers between 2-5 and the number of units per layer between 64 to 512.

As part of preprocessing, we additionally permute all inputs along the feature dimension, so that the UnsupervisedUpdate must learn a permutation invariant learning rule. Unlike other work, we focus explicitly on learning a learning algorithm as opposed to the discovery of fixed feature extractors that generalize across similar tasks. This makes the learning task *much* harder, as the UnsupervisedUpdate has to discover the relationship between pixels based solely on their joint statistics, and cannot "cheat" and memorize pixel identity. To provide further dataset variation, we additionally augment the data with shifts, rotations, and noise. We add these augmentation coefficients as additional regression targets for the meta-objective – e.g. rotate the image and predict the rotation angle as well as the image class. For additional details, see Appendix I.1.1.

4.3 DISTRIBUTED IMPLEMENTATION

We implement the above models in distributed Tensorflow (Abadi et al., 2016). Training uses 512 workers, each of which performs a sequence of partial unrolls of the inner loop UnsupervisedUpdate, and computes gradients of the meta-objective asynchronously. Training takes $\sim$8 days, and consists of $\sim$200 thousand updates to θ with minibatch size 256. Additional details are in Appendix D.

5 EXPERIMENTAL RESULTS

First, we examine limitations of existing unsupervised and meta learning methods, next we show meta-training and generalization properties of our learned optimizer. Finally we conclude by exploring how our learned optimizer functions. For details of experimental setup, see Appendix I.

5.1 OBJECTIVE FUNCTION MISMATCH AND EXISTING META-LEARNING METHODS

To illustrate the negative consequences of objective function mismatch in unsupervised learnin algorithms, we train a variational autoencoder on 16x16 CIFAR10. Over the course of training we

evaluate classification performance from few shot classification using the learned latent representations. Training curves can be seen in Figure 2. Despite continuing to improve the VAE objective throughout training (not shown here), the classification accuracy decreases sharply later in training.

To demonstrate the reduced generalization that results from learning transferable features rather than an update algorithm, we train a prototypical network with and without the input shuffling described in Section 4.2. Due to the transferable features structure of prototypical network, performance is significantly hampered by input shuffling. Results are in Figure 2.

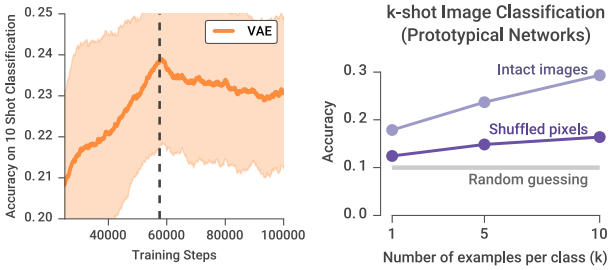

Figure 2: **Left:** Standard unsupervised learning approaches suffer from objective function missmatch. Continuing to optimize a variational auto-encoder (VAE) hurts few-shot accuracy after some number of steps (dashed line). **Right:** Prototypical networks transfer features rather than a learning algorithm, and perform poorly if tasks don't have consistent data structure. Training a prototypical network with a fully connected architecture (same as our base model) on a miniImagenet 10-way classification task with either intact inputs (light purple) or by permuting the pixels before every training and testing task (dark purple). Performance with permuted inputs is greatly reduced (gray line). Our performance is invariant to pixel permutation.

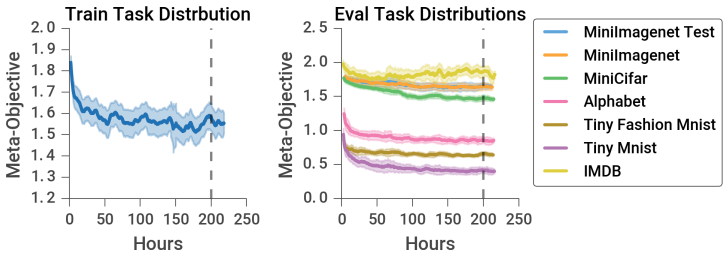

Figure 3: Training curves for the training and evaluation task distributions. Our train set consists of Mini Imagenet, Alphabet, and MiniCIFAR. Our test sets are Mini Imagenet Test, Tiny Fashion MNIST, Tiny MNIST and IMDB. Error bars denote standard deviation of evaluations with a fixed window of samples evaluated from a single model. Dashed line at 200 hours indicates model used for remaining experiments unless otherwise stated. For a bigger version, see Appendix F.

5.2 Meta-Optimization

While training, we monitor a rolling average of the meta-objective averaged across all datasets, model architectures, and the number of unrolling steps performed. In Figure 3 the training loss is continuing to decrease after 200 hours of training, which suggests that the approximate training techniques still produce effective learning. In addition to this global number, we measure performance obtained by rolling out the UnsupervisedUpdate on various training and testing datasets. We see that on held out image datasets, such as MNIST and Fashion Mnist, the evaluation loss is still decreasing. However, for datasets in a different domain, such as IMDB sentiment prediction (Maas et al., 2011), we start to see over-fitting. For all remaining experimental results, unless otherwise stated, we use meta-parameters, θ, for the UnsupervisedUpdate resulting from 200 hours of meta-training.

5.3 Generalization

The goal of this work is to learn a general purpose unsupervised representation learning algorithm. As such, this algorithm must be able to generalize across a wide range of scenarios, including tasks that are not sampled iid from the meta-training distribution. In the following sections, we explore a subset of the factors we seek to generalize over.

Generalizing over datasets and domains

In Figure 4, we compare performance on few shot classification with 10 examples per class. We evaluate test performance on holdout datasets of MNIST and Fashion MNIST at 2 resolutions: 14×14 and 28×28 (larger than any dataset experienced in meta-training). On the same base model architecture, our learned UnsupervisedUpdate leads to performance better than a variational autoencoder, supervised learning on the labeled examples, and random initialization with trained readout layer.

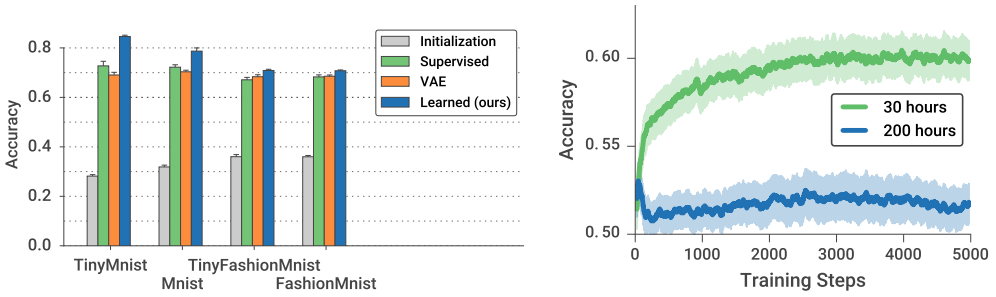

Figure 4: **Left:** The learned UnsupervisedUpdate generalizes to unseen datasets. Our learned update rule produces representations more suitable for few shot classification than those from random initialization or a variational autoecoder and outperforms fully supervised learning on the same labeled examples. Error bars show standard error. **Right:** Early in meta-training, the UnsupervisedUpdate is able to learn useful features on a 2 way text classification data set, IMDB, despite being trained only from image datasets. Later in meta-training performance drops due to the domain mismatch. Error bars show standard error across 10 runs.

To further explore generalization limits, we test our learned optimizer on data from a vastly different domain. We train on a binary text classification dataset: IMDB movie reviews (Maas et al., 2011), encoded by computing a bag of words with 1K words. We evaluate using a model 30 hours and 200 hours into meta-training (see Figure 4). Despite being trained exclusively on image datasets, the 30 hour learned optimizer improves upon the random initialization by almost 10%. When meta-training for longer, however, the learned optimizer "over-fits" to the image domain resulting in poor performance. This performance is quite low in an absolute sense. Nevertheless, we find this result very exciting as we are unaware of any work showing this kind of transfer of learned rules from images to text.

Generalizing over network architectures

We train models of varying depths and unit counts with our learned optimizer and compare results at different points in training (Figure 5). We find that despite only training on networks with 2 to 5 layers and 64 to 512 units per layer, the learned rule generalizes to 11 layers and 10,000 units per layer.

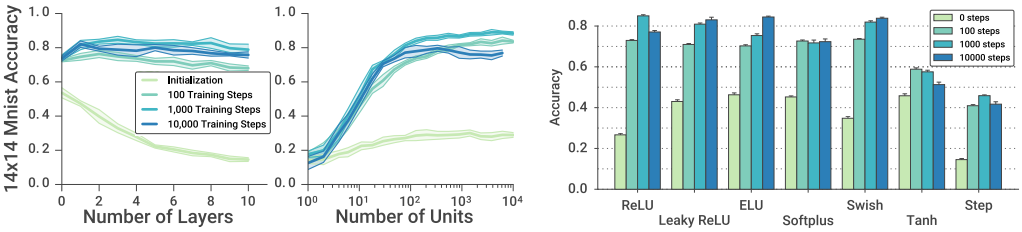

Figure 5: **Left:** The learned UnsupervisedUpdate is capable of optimizing base models with hidden sizes and depths outside the meta-training regime. As we increase the number of units per layer, the learned model can make use of this additional capacity despite never having experienced it during meta-training. **Right:** The learned UnsupervisedUpdate generalizes across many different activation functions not seen in training. We show accuracy over the course of training on 14x14 MNIST.

Next we look at generalization over different activation functions. We apply our learned optimizer on base models with a variety of different activation functions. Performance evaluated at different points in training (Figure 5). Despite training only on ReLU activations, our learned optimizer is able

to improve on random initializations in all cases. For certain activations, leaky ReLU (Maas et al., 2013) and Swish (Ramachandran et al., 2017), there is little to no decrease in performance. Another interesting case is the step activation function. These activations are traditionally challenging to train as there is no useful gradient signal. Despite this, our learned UnsupervisedUpdate is capable of optimizing as it does not use base model gradients, and achieves performance double that of random initialization.

5.4 How it Learns and How it Learns to Learn

To analyze how our learned optimizer functions, we analyze the first layer filters over the course of meta-training. Despite the permutation invariant nature of our data (enforced by shuffling input image pixels before each unsupervised training run), the base model learns features such as those shown in Figure 6, which appear template-like for MNIST, and local-feature-like for CIFAR10. Early in training, there are coarse features, and a lot of noise. As the meta-training progresses, more interesting and local features emerge.

In an effort to understand what our algorithm learns to do, we fed it data from the two moons dataset. We find that despite being a 2D dataset, dissimilar from the image datasets used in meta-training, the learned model is still capable of manipulating and partially separating the data manifold in a purely unsupervised manner (Figure 6). We also find that almost all the variance in the embedding space is dominated by a few dimensions. As a comparison, we do the same analysis on MNIST. In this setting, the explained variance is spread out over more of the principal components. This makes sense as the generative process contains many more latent dimensions – at least enough to express the 10 digits.

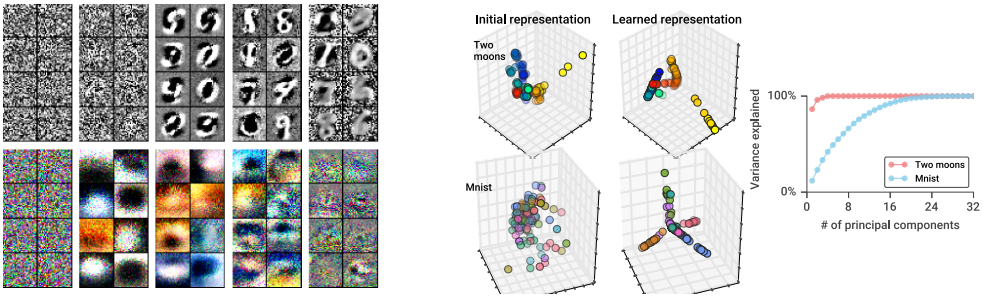

Figure 6: **Left:** From left to right we show first layer base model receptive fields produced by our learned UnsupervisedUpdate rule over the course of meta-training. Each pane consists of first layer filters extracted from ϕ after 10k applications of UnsupervisedUpdate on MNIST (top) and CIFAR10 (bottom). For MNIST, the optimizer learns image-template-like features. For CIFAR10, low frequency features evolve into higher frequency and more spatially localized features. For more filters, see Appendix E. **Center:** Visualization of learned representations before (left) and after (right) training a base model with our learned UnsupervisedUpdate for two moons (top) and MNIST (bottom). The UnsupervisedUpdate is capable of manipulating the data manifold, without access to labels, to separate the data classes. Visualization shows a projection of the 32-dimensional represnetation of the base network onto the top three principal components. **Right:** Cumulative variance explained using principal components analysis (PCA) on the learned representations. The representation for two moons data (red) is much lower dimensional than MNIST (blue), although both occupy a fraction of the full 32-dimensional space.

6 Discussion

In this work we meta-learn an unsupervised representation learning update rule. We show performance that matches or exceeds existing unsupervised learning on held out tasks. Additionally, the update rule can train models of varying widths, depths, and activation functions. More broadly, we demonstrate an application of meta-learning for learning complex optimization tasks where no objective is explicitly defined. Analogously to how increased data and compute have powered supervised learning, we believe this work is a proof of principle that the same can be done with algorithm design – replacing the hand designed techniques with architectures designed for learning and learned from data via meta-learning techniques.

REFERENCES

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

Juergen Schmidhuber. On learning how to learn learning strategies. 1995.

Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Quoc V Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S Corrado, Jeff Dean, and Andrew Y Ng. Building high-level features using large scale unsupervised learning. *arXiv preprint arXiv:1112.6209*, 2011.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. Adversarial feature learning. *arXiv preprint arXiv:1605.09782*, 2016.

Vincent Dumoulin, Ishmael Belghazi, Ben Poole, Alex Lamb, Martin Arjovsky, Olivier Mastropietro, and Aaron Courville. Adversarially learned inference. *arXiv preprint arXiv:1606.00704*, 2016.

Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In *European Conference on Computer Vision*, pages 69–84. Springer, 2016.

Ishan Misra, C Lawrence Zitnick, and Martial Hebert. Shuffle and learn: unsupervised learning using temporal order verification. In *European Conference on Computer Vision*, pages 527–544. Springer, 2016.

Pierre Sermanet, Corey Lynch, Jasmine Hsu, and Sergey Levine. Time-contrastive networks: Self-supervised learning from multi-view observation. *arXiv preprint arXiv:1704.06888*, 2017.

Adam Coates and Andrew Y Ng. Learning feature representations with k-means. In *Neural networks: Tricks of the trade*, pages 561–580. Springer, 2012.

Junyuan Xie, Ross Girshick, and Ali Farhadi. Unsupervised deep embedding for clustering analysis. In *International conference on machine learning*, pages 478–487, 2016.

Piotr Bojanowski and Armand Joulin. Unsupervised learning by predicting noise. *arXiv preprint arXiv:1704.05310*, 2017.

Jürgen Schmidhuber. Learning factorial codes by predictability minimization. *Neural Computation*, 4(6): 863–879, 1992.

Sepp Hochreiter and Jürgen Schmidhuber. Feature extraction through lococode. *Neural Computation*, 11(3): 679–714, 1999.

Bruno A Olshausen and David J Field. Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision research*, 37(23):3311–3325, 1997.

Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

Yoshua Bengio, Samy Bengio, and Jocelyn Cloutier. *Learning a synaptic learning rule*. Université de Montréal, Département d'informatique et de recherche opérationnelle, 1990.

Samy Bengio, Yoshua Bengio, Jocelyn Cloutier, and Jan Gecsei. On the optimization of a synaptic learning rule. In *Preprints Conf. Optimality in Artificial and Biological Neural Networks*, pages 6–8. Univ. of Texas, 1992.

Thomas Philip Runarsson and Magnus Thor Jonsson. Evolution and design of distributed learning rules. In *Combinations of Evolutionary Computation and Neural Networks, 2000 IEEE Symposium on*, pages 59–63. IEEE, 2000.

Oriol Vinyals, Charles Blundell, Tim Lillicrap, koray kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3630–3638. Curran Associates, Inc., 2016. URL `http://papers.nips.cc/paper/6385-matching-networks-for-one-shot-learning.pdf`.

Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. *International Conference on Learning Representations*, 2016.

Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. Meta-learning with temporal convolutions. *arXiv preprint arXiv:1707.03141*, 2017.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL `http://proceedings.mlr.press/v70/finn17a.html`.

Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*, pages 4080–4090, 2017.

Vikas K Garg and Adam Kalai. Supervising unsupervised learning. *arXiv preprint arXiv:1709.05262*, 2017.

Donald R Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of global optimization*, 21(4):345–383, 2001.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *International Conference on Learning Representations*, 2017. URL `https://arxiv.org/abs/1611.01578`.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *International Conference on Learning Representations*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando de Freitas. Learning to learn without gradient descent by gradient descent. *arXiv preprint arXiv:1611.03824*, 2016.

Ke Li and Jitendra Malik. Learning to optimize. *International Conference on Learning Representations*, 2017.

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando de Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. *International Conference on Machine Learning*, 2017.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc Le. Neural optimizer search with reinforcement learning. 2017. URL https://arxiv.org/pdf/1709.07417.pdf.

Rein Houthooft, Richard Y Chen, Phillip Isola, Bradly C Stadie, Filip Wolski, Jonathan Ho, and Pieter Abbeel. Evolved policy gradients. *arXiv preprint arXiv:1802.04821*, 2018.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/ioffe15.html.

James CR Whittington and Rafal Bogacz. An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity. *Neural computation*, 29(5):1229–1262, 2017.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P11-1015.

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

Prajit Ramachandran, Barret Zoph, and Quoc Le. Searching for activation functions. 2017.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2012.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Barak Pearlmutter. *An investigation of the gradient descent process in neural networks*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 1996.

Samuel S Schoenholz, Justin Gilmer, Surya Ganguli, and Jascha Sohl-Dickstein. Deep information propagation. *arXiv preprint arXiv:1611.01232*, 2016.

A META-LEARNING COMPARISONS TABLE

Method	Inner loop updates	Outer loop updates, meta-			Generalizes to
		parameters	objective	optimizer	
Hyper parameter optimization Jones (2001); Snoek et al. (2012); Bergstra et al. (2011); Bergstra and Bengio (2012)	many steps of optimization	optimization hyper-parameters	training or validation set loss	Baysian methods, random search, etc	nothing, or test set within fixed dataset
Neural architecture search Stanley and Miikkulainen (2002); Zoph and Le (2017); Baker et al. (2017); Zoph et al.; Real et al. (2017)	supervised SGD training using meta-learned architecture	architecture	validation set loss	RL or evolution	test loss within similar datasets
Task-specific optimizer (eg for quadratic function identification) (Hochreiter et al., 2001)	adjustment of model weights by an LSTM	LSTM weights	task loss	SGD	similar domain tasks
Learned optimizers Jones (2001); Maclaurin et al. (2015); Andrychowicz et al. (2016); Chen et al. (2016); Li and Malik (2017); Wichrowska et al. (2017); Bello et al. (2017)	many steps of optimization of a fixed loss function	parametric optimizer	average or final loss	SGD or RL	new loss functions (mixed success)
Prototypical networks Snell et al. (2017)	apply a feature extractor to a batch of data and use soft nearest neighbors to compute class probabilities	weights of the feature extractor	few shot performance	SGD	new image classes within similar dataset
MAML Finn et al. (2017)	one step of SGD on training loss starting from a meta-learned network	initial weights of neural network	reward or training loss	SGD	new goals, similar task regimes with same input domain
Evolved Policy Gradient Houthooft et al. (2018)	performing gradient descent on a learned loss	parameters of a learned loss function	reward	Evolutionary Strategies	new environment configurations, both in and not in meta-training distribution.
Few shot learning (Vinyals et al., 2016; Ravi and Larochelle, 2016; Mishra et al., 2017)	application of a recurrent model, e.g. LSTM, Wavenet.	recurrent model weights	test loss on training tasks	SGD	new image classes within similar dataset.
Meta-unsupervised learning for clustering Garg and Kalai (2017)	run clustering algorithm or evaluate binary similarity function	clustering algorithm + hyperparameters, binary similarity function	empirical risk minimization	varied	new clustering or similarity measurement tasks
Learning synaptic learning rules (Bengio et al., 1990; 1992)	run a synapse-local learning rule	parametric learning rule	supervised loss, or similarity to biologically-motivated network	gradient descent, simulated annealing, genetic algorithms	similar domain tasks
Our work — metalearning for unsupervised representation learning	**many applications of an unsupervised update rule**	**parametric update rule**	**few shot classification after unsupervised pre-training**	**SGD**	**new base models (width, depth, nonlinearity), new datasets, new data modalities**

Table 1: A comparison of published meta-learning approaches.

B MORE DETAILED SYSTEM DIAGRAM

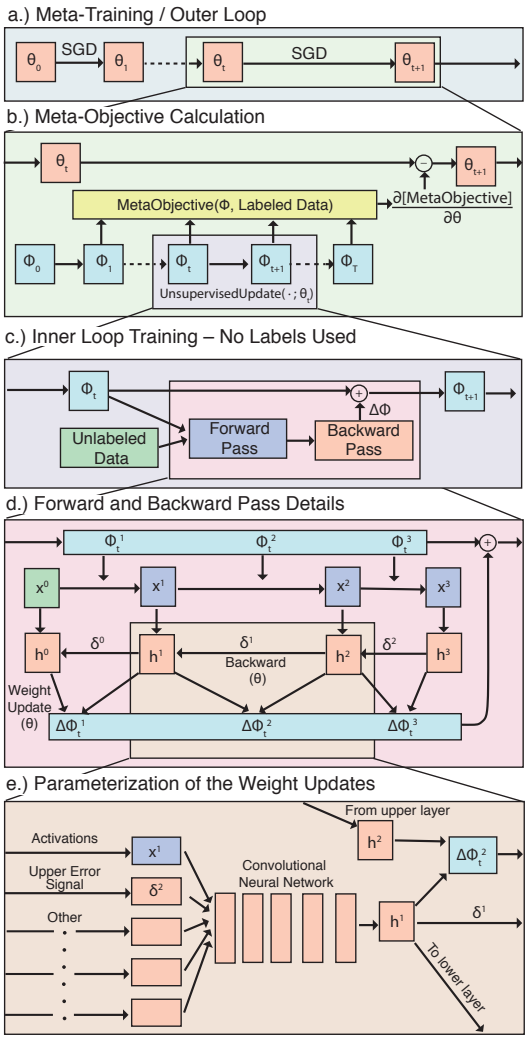

Figure App.1: Schematic for meta-learning an unsupervised learning algorithm. **a)** Meta-training, where the meta-parameters, θ, are updated via our meta-optimizer (SGD). **b)** The gradients of the MetaObjective with respect to θ are computed by backpropagation through the unrolled application of the UnsupervisedUpdate. **c)** UnsupervisedUpdate updates the base model parameters (ϕ) using a minibatch of data. **d)** Each application of UnsupervisedUpdate involves computing a forward and "backward" pass through the base model. The base model itself is a fully connected network producing hidden states x^l for each layer l. The "backward" pass through the base model uses an error signal from the layer above, δ, which is generated by a meta-learned function. **e.)** The weight updates $\Delta\phi$ are computed using a convolutional network, using δ and x from the pre- and post-synaptic neurons, along with several other terms discussed in the text.

C STABILIZING GRADIENT BASED META-LEARNING TRAINING

Training and computing derivatives through recurrent computation of this form is notoriously difficult Pascanu et al. (2013). Training parameters of recurrent systems in general can lead to chaos. We used the usual techniques such as gradient clipping (Pascanu et al., 2012), small learning rates, and adaptive learning rate methods (in our case Adam (Kingma and Ba, 2014)), but in practice this was not enough to train most UnsupervisedUpdate architectures. In this section we address other techniques needed for stable convergence.

When training with truncated backprop the problem shifts from pure optimization to something more like optimizing on a Markov Decision Process where the state space is the base-model weights, ϕ, and the 'policy' is the learned optimizer. While traversing these states, the policy is constantly meta-optimized and changing, thus changing the distribution of states the optimizer reaches. This type of non-i.i.d training has been discussed at great length with respect to on and off-policy RL training algorithms (Mnih et al., 2013). Other works cast optimizer meta-learning as RL (Li and Malik, 2017) for this very reason, at a large cost in terms of gradient variance. In this work, we partially address this issue by training a large number of workers in parallel, to always maintain a diverse set of states when computing gradients.

For similar reasons, the number of steps per truncation, and the total number of unsupervised training steps, are both sampled in an attempt to limit the bias introduced by truncation.

We found restricting the maximum inner loop step size to be crucial for stability. Pearlmutter (1996) studied the effect of learning rates with respect to the stability of optimization and showed that as the learning rate increases gradients become chaotic. This effect was later demonstrated with respect to neural network training in Maclaurin et al. (2015). If learning rates are not constrained, we found that they rapidly grew and entered this chaotic regime.

Another technique we found useful in addressing these problems is the use of batch norm in both the base model and in the UnsupervisedUpdate rule. Multi-layer perceptron training traditionally requires very precise weight initialization for learning to occur. Poorly scaled initialization can make learning impossible (Schoenholz et al., 2016). When applying a learned optimizer, especially early in meta-training of the learned optimizer, it is very easy for the learned optimizer to cause high variance weights in the base model, after which recovery is difficult. Batch norm helps solve this issues by making more of the weight space usable.

D DISTRIBUTED IMPLEMENTATION

We implement the described models in distributed Tensorflow (Abadi et al., 2016). We construct a cluster of 512 workers, each of which computes gradients of the meta-objective asynchronously. Each worker trains on one task by first sampling a dataset, architecture, and a number of training steps. Next, each worker samples k unrolling steps, does k applications of the UnsupervisedUpdate$(\cdot;\theta)$, computes the MetaObjective on each new state, computes $\frac{\partial[\text{MetaObjective}]}{\partial\theta}$ and pushes it to a parameter server. The final base-model state, ϕ, is then used as the starting point for the next unroll until the specified number of steps is reached. These gradients from different workers are batched and θ is updated with asynchronous SGD. By batching gradients as workers complete unrolls, we eliminate most gradient staleness while retaining the compute efficiency of asynchronous workers, especially given heterogeneous workloads which arise from dataset and model size variation. An overview of our training can be seen in algorithm G. Due to the small base models and the sequential nature of our compute workloads, we use multi core CPUs as opposed to GPUs. Training occurs over the course of $\sim$8 days with $\sim$200 thousand updates to θ with minibatch size 256.

E MORE FILTERS OVER META-TRAINING

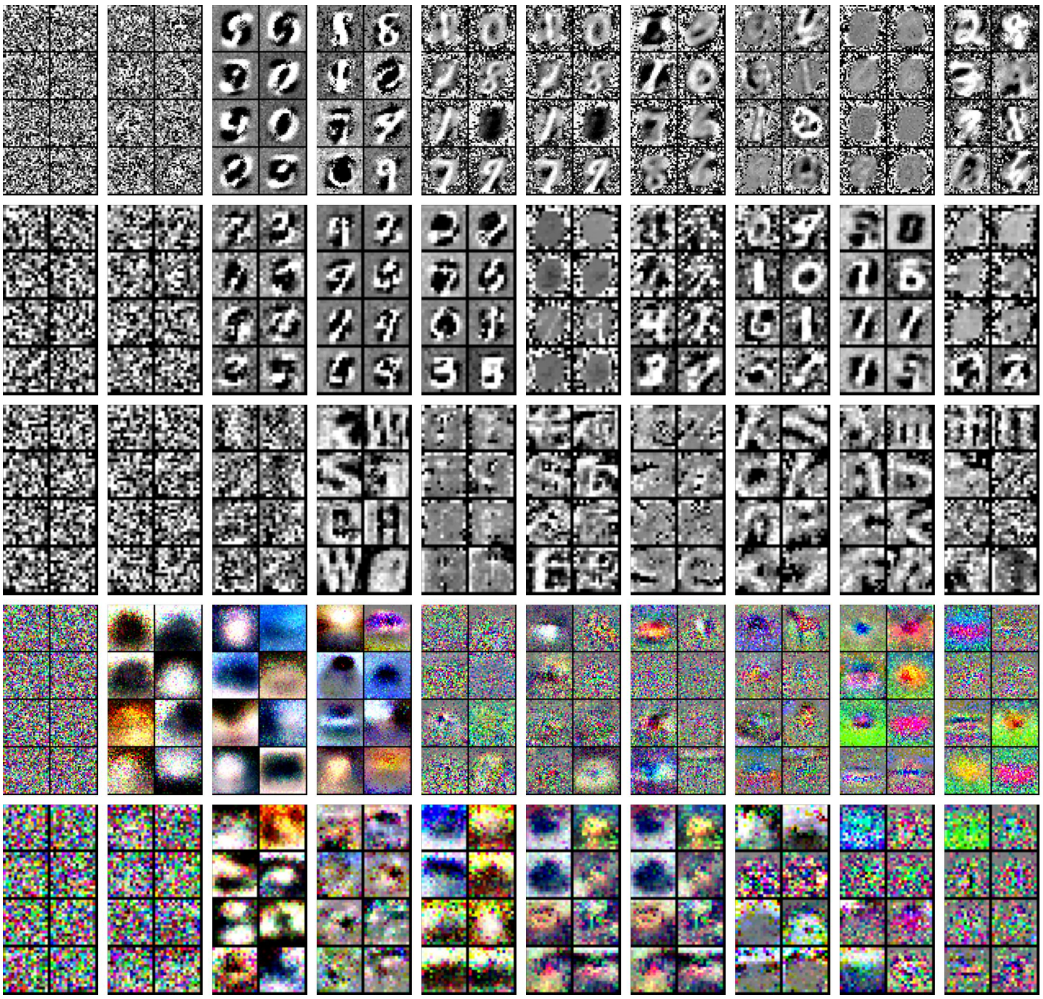

Figure App.2: More filters extracted over the course of meta-training. Note, due to implementation reasons, the columns do not represent the same point / same iteration of θ. Each filter is extracted after 10k inner loop optimization steps, at ϕ_{10k}. From top to bottom we show: MNIST, Tiny MNIST, Alphabet, CIFAR10, and Mini CIFAR10. Filters shift from noise at initialization to more local features later in meta-training.

F META-TRAINING CURVES

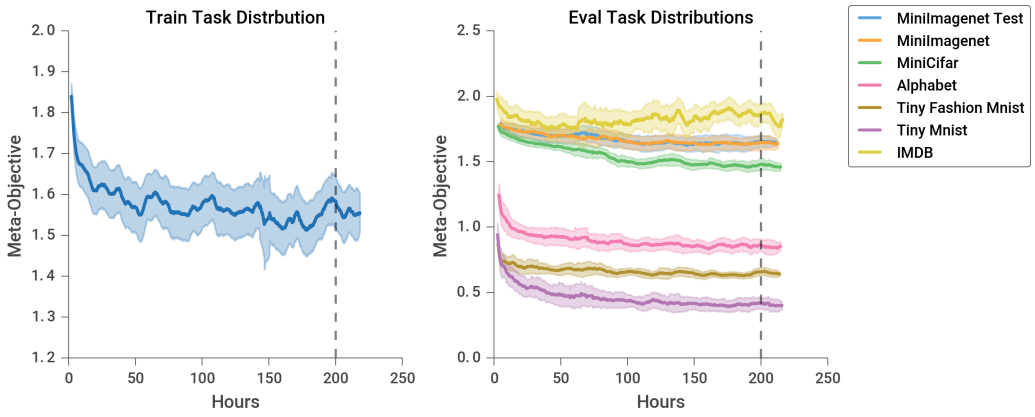

Figure App.3: Training curves for the training and evaluation task distributions. Our train set consists of Mini Imagenet, Alphabet, and MiniCIFAR10. Our test sets are Mini Imagenet Test, Tiny Fashion MNIST, Tiny MNIST and IMDB. Error bars denote standard deviation of evaluations with a fixed window of samples evaluated from a single model.

G LEARNING ALGORITHM

Initialize UnsupervisedUpdates parameters, θ_0.
Initialize meta-training step count $v \leftarrow 0$.
Initialize shared gradient state $\mathcal{S}$
Initialize V to be max meta-training steps.
while $v < V$ on 512 workers in parallel **do**
  Sample base model $f(\cdot; \phi)$ and initialize ϕ_0
  Sample supervised task, $\mathcal{D}$
  Sample K truncated steps
  Initialize learner iteration count $t \leftarrow 0$
  for $k = 1$ **to** K **do**
    Sample U unroll steps.
    for $u = 1$ **to** U **do**
      Sample data, x, from $\mathcal{D}$
      $\phi_{t+1} = \text{UnsupervisedUpdate}(x, \phi_t; \theta)$
      $t \leftarrow t + 1$
    end for
    Sample 2 batches of data, x,x', and labels, y,y' from D.
    $J = \text{MetaObjective}(x, y, x'm, y', \phi_t)$
    Compute $\frac{\partial J}{\partial \theta_v}$ for the last U steps and store in $\mathcal{S}$
    if size of $\mathcal{S} >$ mini batch size **then**
      take averaged batch of gradients G from $\mathcal{S}$.
      $\theta_{v+1} = \theta_v - ADAM(G)$
      $v \leftarrow v + 1$
    end if
  end for
end while

Algorithm 1: Distributed Training Algorithm

H MODEL SPECIFICATION

In this section we describe the details of our base model and learned optimizer. First, we describe the inner loop, and then we describe the meta-training procedure in more depth.

In the following sections, λ_{name} denotes a hyper-parameter for the learned update rule.

The design goals of this system are stated in the text body. At the time of writing, we were unaware of any similar systems, so as such the space of possible architectures was massive. Future work consists of simplifying and improving abstractions around algorithmic components.

Opensource implementation of the UnsupervisedUpdate can be found at `https://not_a_real_url.com`.

H.1 INNER LOOP

The inner loop computation consists of iterative application of the UnsupervisedUpdate on a Base Model parameterized by ϕ,

$$\phi_{t+1} = \text{UnsupervisedUpdate}(\phi_t), \tag{App.1}$$

where ϕ consists of weights and biases, W^l, b^l, parameterizing a multi-layer perceptron.

This can be broken down further as a forward pass on an unlabeled batch of data,

$$x^0 \sim \mathcal{D} \tag{App.2}$$

$$\{x^1..x^L, z^1..z^L\} = f(x; \phi_t), \tag{App.3}$$

where z^l, and x^l are the pre- and post-activations on layer l, and then a function to compute weight updates:

$$\{(\Delta_{W^{1\cdots L}})_t, (\Delta_{b^{1\cdots L}})_t\} = computeDeltaWeight(x^0 \cdots x^L, z^1 \cdots z^L, \phi_t; \theta) \tag{App.4}$$

Finally, the next set of ϕ weights and biases, W^l, b^l, are computed. We use an SGD like update but with the addition of a decay term. Equivalently, this can be seen as setting the weights and biases to an exponential moving average of the ΔW and Δb terms.

$$W^l_{t+1} = W^l_t(1 - \lambda_{\phi lr}) - \Delta_{W^l_{final}} \lambda_{\phi lr} \tag{App.5}$$

$$b^l_{t+1} = b^l_t(1 - \lambda_{\phi lr}) - \Delta_{b^l} \lambda_{\phi lr} \tag{App.6}$$

We use $\lambda_{\phi lr} = 3e - 4$ in our work.

In the following sections, we describe the functional form of the base model, f, as well as the functional form of $computeDeltaWeight\,(\cdot)$.

H.2 BASE MODEL

The base model, the model our learned update rule is training, is an L layer multi layer perception with batchnorm. We define ϕ as this model's parameters, consisting of weights and biases. We define $N^1..N^L$ to be the sizes of the layers of the base model.

$$\phi = \{W^1..W^L, b^1..b^L\}, \tag{App.7}$$

where

$$W^1 \in \mathbb{R}^{N_x, N_{inner}} \tag{App.8}$$

$$W^{2:L-1} \in \mathbb{R}^{N_{inner}, N_{inner}} \tag{App.9}$$

$$b^1 \cdots b^{L-1} \in \mathbb{R}^{N_{inner}} \tag{App.10}$$

$$W^L \in \mathbb{R}^{N_{inner}, N_{embedding}} \tag{App.11}$$

$$b^L \in \mathbb{R}^{N_{embedding}} \tag{App.12}$$

and N_{inner} is the hidden size of the network, N_x is the input size of data, and $N_{embedding}$ is the size of the output embedding. In this work, we fix this: $N_{embedding} = 32$.

The forward computation parameterized by ϕ, consumes batches of unlabeled data from a dataset $\mathcal{D}$:

$$x^0 \sim \mathcal{D}, x^0 \in \mathbb{R}^{B,N_x} \tag{App.13}$$

$$z^l = \text{BatchNorm}\left(x^{l-1}W^l\right) + b^l \tag{App.14}$$

$$x^l = \text{Relu}(\text{z}^\text{l}) \tag{App.15}$$

for $l = 1..L$.

We define $f(x;\phi)$ as a function that returns the set of internal pre- and post-activation function hidden states as well as $\hat{f}(x;\phi)$ as the function returning the final hidden state:

$$f(x,\phi) = \{z^1...z^L, x^0..x^L, \phi\} \tag{App.16}$$

$$\hat{f}(x,\phi) = x^L \tag{App.17}$$

H.3 MetaObjective

We define the MetaObjective to be a few shot linear regression. To increase stability and avoid undesirable loss landscapes, we additionally center the prediction targets, as well as normalize the predicted target before doing the loss computation. The full computation is as follows:

$$\text{MetaObjective}(x,y,x',y',\phi) :: (\mathbb{R}^{B,N^0}, \mathbb{R}^{B,N^{classes}}, \mathbb{R}^{B,N^0}, \mathbb{R}^B, \mathbb{R}^{N^{ydim}}, \Phi) \to \mathbb{R}^1 \tag{App.18}$$

First, the inputs are converted to embeddings with the base model,

$$u_a = f(x_a;\phi) \tag{App.19}$$

$$u'_a = f(x'_a;\phi) \tag{App.20}$$

Next, we center and normalize the prediction targets. We show this for y, but y' is processed identically.

$$\bar{y}_i = \mathbb{E}_a\left[y_{ai}\right] \tag{App.21}$$

$$\hat{y}_{ai} = \frac{\bar{y}_{ai}}{\sqrt{\mathbb{E}_b\left[\|\bar{y}_b\|_2^2\right]}} \tag{App.22}$$

We then solve for the linear regression weights in closed form with features: u and targets: $\hat{y}$. We account for a bias by concatenating a 1's vector to the features.

$$A = [u; \mathbb{1}] \tag{App.23}$$

$$T = (A^t A) + I\lambda_{ridge} \tag{App.24}$$

$$C = T^{-1}A^T\hat{y} \tag{App.25}$$

We then use these inferred regression weights C to make a prediction on the second batch of data, normalize the resulting prediction, and compute a final loss,

$$p = C[u'; \mathbb{1}] \tag{App.26}$$

$$\hat{p}_a = \frac{p}{\|p_a\|_2} \tag{App.27}$$

$$\text{MetaObjective}(\cdot) = \mathbb{E}_a\left[\|\hat{p}_a - \hat{y}_b\|_2^2\right]. \tag{App.28}$$

Note that due to the normalization of predictions and targets, this corresponds to a cosine distance (up to an offset and multiplicative factor).

H.4 UnsupervisedUpdate

The learned update rule is parameterized by θ. In the following section, we denote all instances of θ with a subscript to be separate named learnable parameters of the UnsupervisedUpdate. θ is

shared across all instantiations of the unsupervised learning rule. It is this weight sharing that lets us generalize across different network architectures.

The computation is split into a few main components: first, there is the forward pass, defined in $f(x;\phi)$. Next, there is a "backward" pass, which operates on the hidden states in a reverse order to propagate an error signal δ^l back down the network. In this process, a hidden state h^l is created for each layer. Weight updates to W^l and b^l are then readout from these h^l and other signals found locally.

H.4.1 BACKWARD ERROR PROPAGATION

In backprop, there exists a single scalar error signal that gets backpropagated back down the network. In our work, this error signal does not exist as there is no loss being optimized. Instead we have a learned top-down signal, d^L, at the top of the network. Because we are no longer restricted to the form of backprop, we make this quantity a vector for each unit in the network, rather than a scalar,

$$x^0 \sim \mathcal{D}, x^0 \in \mathbb{R}^{B,N_x} \tag{App.29}$$

$$\{z^1 ... z^L, x^0 .. x^L, \phi\} = f(x, \phi) \tag{App.30}$$

$$d^L = topD(x^L; \theta_{topD}) \tag{App.31}$$

$$\tag{App.32}$$

where $d^l \in \mathbb{R}^{B,N^l,\lambda deltadims}$. The architecture of $topD$ is a neural network that operates along every dimension of x^L. The specification can be found in H.7.

We structure our error propagation similarly to the structure of the backpropagated error signal in a standard MLP with contributions to the error at every layer in the network,

$$\delta^l_{ijd} = d^{l+1}_{ijd} \odot \sigma(z^{l+1}_{ijd}) + \sum_k^{N^l} \left(\theta_{errorProp\ W}\right)_{dk} h^{l+1}_{ijk} + \theta_{errorProp\ b_d} \tag{App.33}$$

where δ^l has the same shape as d^l.

In a similar way to backprop, we move this signal down the network via multiplying by a backward weight matrix. We do not use the previous weight matrix transpose as done in backprop, instead we learn a separate set of weights that are not tied to the forward weights and updated along with the forward weights as described in H.5. Additionally, we normalize the signal to have fixed second moment,

$$\tilde{d}^l_{imd} = \sum_j^{N^{l+1}} \delta^l_{ijd} (V^{l+1})_{mj} \tag{App.34}$$

$$d^l_{imd} = \hat{d}^l_{imd} \frac{\left(BN^l \lambda_{deltadims}\right)^{1/2}}{\left\|\tilde{d}^l\right\|_2} \tag{App.35}$$

The internal $h^l \in \mathbb{R}^{B,N^l,\lambda_{hsize}}$ vectors are computed via:

$$h^l = computeH\left(d^l, x^l, z^l; \theta_{computeH}\right) \tag{App.36}$$

The architecture of $computeH$ is a neural network that operates on every dimension of all the inputs. It can be found in H.8. These definitions are recursive, and are computed in order: $h^L, h^{L-1} \cdots h^1, h^0$. With these computed, weight updates can be read out.

H.5 WEIGHT UPDATES

For a given layer, l, our weight updates are a mixture of multiple low rank readouts from h^l and h^{l-1}. These terms are then added together with a learnable weight, in θ to form a final update. The final update is then normalized mixed with the previous weights. We update both the forward, W^l, and the

backward, V^l, using the same update rule parameters θ. We show the forward weight update rule here, and drop the backward for brevity.

For convenience, we define a low rank readout function lowRR that takes h^l like tensors, and outputs a single lower rank tensor.

$$\mathrm{lowRR}\left(h_a, h_b; \Theta\right) :: \tag{App.37}$$

$$\left(\mathbb{R}^{B,N_a,\lambda_{hsize}}, \mathbb{R}^{B,N_b,\lambda_{hsize}}\right) \rightarrow \mathbb{R}^{N_a,N_b} \tag{App.38}$$

Here, Θ, is a placehoder for the parameters of the given readout. lowRR is defined as:

$$\Theta = \{V^t, V^b\} \tag{App.39}$$

$$r^a_{ijp} = \sum_k h^a_{ijk} V^t_{kp} \tag{App.40}$$

$$r^b_{ijp} = \sum_k h^a_{ijk} V^t_{kp} \tag{App.41}$$

$$\mathrm{lowRR}\left(\cdot\right) = \sum_k^B \sum_i^{\lambda_{gradC}} r^b_{ijk} r^a_{ipk} \frac{1}{B\lambda_{gradC}} \tag{App.42}$$

H.5.1 Local terms

This sequence of terms allow the weight to be adjusted as a function of state in the pre- and post-synaptic neurons. They should be viewed as a basis function representation of the way in which the weight changes as a function of pre- and post-synaptic neurons, and the current weight value. We express each of these contributions to the weight update as a sequence of weight update planes, with the ith plane written $\Delta W_i \in \mathbb{R}^{N^{l-1} \times N^l}$:

$$\hat{W}^l = \frac{W^l}{\sqrt{\mathbb{E}_i\left[((W^l)_i)^2\right]}} \tag{App.43}$$

$$\Delta W^l_1 = \hat{W}^l \tag{App.44}$$

$$\Delta W^l_2 = (\hat{W}^l)^2 sign(\hat{W}^l) \tag{App.45}$$

$$\Delta W^l_3 = lowRR(h^{l-1}, h^l; \theta_{zero}) \tag{App.46}$$

$$\Delta W^l_4 = exp(-(\hat{W}^l_b)^2) \odot lowRR(h^{l-1}, h^l; \theta_{rbf}) \tag{App.47}$$

$$\Delta W^l_5 = W^l_b \odot lowRR(h^{l-1}, h^l; \theta_{first}) \tag{App.48}$$

$$\Delta W^l_6 = \mathbb{E}_b\left[\left(x^{l-1}_b - \mathbb{E}_{b'}\left[x^{l-1}_{b'}\right]\right)^T \left(x^l_b - \mathbb{E}_{b'}\left[x^l_{b'}\right]\right)^T\right] \tag{App.49}$$

H.5.2 Decorrelation terms

Additional weight update planes are designed to aid units in remaining decorrelated from each other's activity, and in decorrelating their receptive fields. Without terms like this, a common failure mode is for many units in a layer to develop near-identical representations. Here, S_i indicates a scratch matrix

associated with weight update layer i.

$$S_7 = \frac{1}{(N^{l-1})^{\frac{1}{2}}} lowRR(h^{1-1}, h^{l-1}, \theta_{linLowerSymm}) \tag{App.50}$$

$$\Delta W_7^l = \frac{1}{\sqrt{2}} \left[\left(S_7 + S_7^T \right) \odot (1 - I) \right] W^l \tag{App.51}$$

$$S_8 = \frac{1}{(N^{l-1})^{\frac{1}{2}}} lowRR(h^{1-1}, h^{l-1}, \theta_{sqrLowerSymm}) \tag{App.52}$$

$$\Delta W_8^l = \frac{1}{\sqrt{2}} \left[\left(S_8 + S_8^T \right) \odot (1 - I) \right] \left(\sqrt{1 + (W^l)^2} - 1 \right) \tag{App.53}$$

$$S_9 = \frac{1}{(N^{l})^{\frac{1}{2}}} lowRR(h^{1}, h^{l}, \theta_{linUpperSymm}) \tag{App.54}$$

$$\Delta W_9^l = \frac{1}{\sqrt{2}} W^l \left[\left(S_9 + S_9^T \right) \odot (1 - I) \right] \tag{App.55}$$

$$S_{10} = \frac{1}{(N^{l})^{\frac{1}{2}}} lowRR(h^{1-1}, h^{l-1}, \theta_{sqrUpperSymm}) \tag{App.56}$$

$$\Delta W_{10}^l = \frac{1}{\sqrt{2}} \left(\sqrt{1 + (W^l)^2} - 1 \right) \left[\left(S_{10} + S_{10}^T \right) \odot (1 - I) \right] \tag{App.57}$$

$$\tag{App.58}$$

H.6 Application of the weight terms in the optimizer

We then normalize, reweight, and merge each of these weight update planes into a single term, which will be used in the weight update step,

$$\Delta \tilde{W}_i^l = \frac{\Delta W_i^l}{\sqrt{1 + \mathbb{E}_{mn} \left[\left(\Delta W_i^l \right)_{mn}^2 \right]}} \tag{App.59}$$

$$(\Delta W_{merge}^l) = \mathbb{E}_i \left[\theta_{mergeW_i} \Delta \tilde{W}_i^l \right] \tag{App.60}$$

To prevent pathologies during training, we perform two post processing steps to prevent the learned optimizer from cheating, and increasing its effective learning rate, leading to instability. We only allow updates which do not decrease the weight matrix magnitude,

$$\Delta W_{orth}^l = \Delta W_{merge}^l - \hat{W}^l \operatorname{Relu}(\Delta W_{merge} \cdot \hat{W}^l), \tag{App.61}$$

where $\hat{W}^l$ is W^l scaled to have unit norm, and we normalize the length of the update,

$$(\Delta W_{final}) = \frac{\Delta W_{orth}^l}{\sqrt{1 + \mathbb{E}_{mn} \left[\left(\Delta W_{orth}^l \right)_{mn}^2 \right]}} \tag{App.62}$$

To compute changes in the biases, we do a readout from h^l. We put some constraints on this update to prevent the biases from pushing all units into the linear regime, and minimizing learning. We found this to be a possible pathology.

$$\Delta b_{base}^l = \sum_i \sum_k^{\lambda_{hsize}} (\theta_{B_{readout}})_k h_{ijk}^l \tag{App.63}$$

$$\Delta b_{constrained}^l = \Delta b_{base}^l - \max \left(\mathbb{E}_i \left[\left(\Delta b_{base}^l \right)_i \right] \right) \tag{App.64}$$

H.7 $topD$

This function performs various convolutions over the batch and data dimension. For ease of notation, we use m as an intermediate variable. Additionally, we drop all convolution and batchnorm

parameterizations. They are all separate elements of θ_{topD}. We define the S argument to be size of hidden units, and the K argument to be the kernel size of the 1D convolutions. Additionally, we set the second argument of $BatchNorm$ to be the axies normalized over.

$$topD\left(x^L;\theta_{topD}\right) :: \mathbb{R}^{B,N^L} \rightarrow \mathbb{R}^{B,N^L,\lambda_{deltadims}} \tag{App.65}$$

In this work, we set $\lambda_{deltadims} = 32$. First, we reshape to add another dimension to the end of X^L, in pseudocode:

$$m_0 = [x^L] \tag{App.66}$$

Next, a convolution is performed on the batch dimension with a batchnorm and a relu non linearity. This starts to pull information from around the batch into these channels.

$$m_1 = ConvBatch\left(m_0, S = \lambda_{topdeltasize}, K = 5\right) \tag{App.67}$$
$$m_2 = relu\left(BatchNorm\left(m_1, [0,1]\right)\right) \tag{App.68}$$
$$\tag{App.69}$$

We set $\lambda_{topdeltasize} = 64$. Next, a series of unit convolutions (convolutions over the last dimension) are performed. These act as compute over information composed from the nearby elements of the batch. These unit dimensions effectively rewrite the batch dimension. This restricts the model to operate on a fixed size batch.

$$m_3 = ConvUnit\left(m_2, S = B, K = 3\right) \tag{App.70}$$
$$m_4 = relu(BatchNorm\left(m_3, [0,1]\right) \tag{App.71}$$
$$m_5 = ConvUnit\left(m_4, S = B, K = 3\right) \tag{App.72}$$
$$m_6 = relu\left(BatchNorm\left(m_5, [0,1]\right)\right) \tag{App.73}$$

Next a series of 1D convolutions are performed over the batch dimension for more compute capacity.

$$m_7 = ConvBatch\left(m_6, S = \lambda_{topdeltasize}, K = 3\right) \tag{App.74}$$
$$m_8 = relu\left(BatchNorm\left(m_7, [0,1]\right)\right) \tag{App.75}$$
$$m_9 = ConvBatch\left(m_8, S = \lambda_{topdeltasize}, K = 3\right) \tag{App.76}$$
$$m_{10} = relu\left(BatchNorm\left(m_9, [0,1]\right)\right) \tag{App.77}$$

Finally, we convert the representations to the desired dimensions and output.

$$m_{11} = ConvBatch\left(m_{10}, S = \lambda_{deltadims}, K = 3\right) \tag{App.78}$$
$$topD\left(x^L;\theta_{topD}\right) = m_{11} \tag{App.79}$$

H.8 $computeH$

This is the main computation performed while transfering signals down the network and the output is directly used for weight updates. It is defined as:

$$computeH\left(d^l, x^l, z^l, w^{l-1}, w^l, b^l; \theta_{computeH}\right) ::$$
$$\left(\mathbb{R}^{B,N^l,\lambda_{deltadims}}, \mathbb{R}^{B,N^l}, \mathbb{R}^{B,N^l}, \mathbb{R}^{N^{l-1},N^l}, \mathbb{R}^{N^l,N^{l+1}}, \mathbb{R}^{N^l}\right) \rightarrow \left(\mathbb{R}^{B,N^l,\lambda_{hdims}}\right) \tag{App.80}$$

The outputs of the base model, (x^l, z^l), plus an additional positional embeddings are stacked then concatenated with d, to form a tensor in $\mathbb{R}^{B,N^l,5+\lambda_{deltadims}}$:

$$(p_0)_{ij} = sin(\frac{2i\pi}{N^l}) \tag{App.81}$$
$$(p_1)_{ij} = cos(\frac{2i\pi}{N^l}) \tag{App.82}$$
$$m_0 = [x^l, z^l, p_0, p_1] \tag{App.83}$$
$$m_1 = [m_0; d^l] \tag{App.84}$$

Statistics across the batch and unit dimensions, 0 and 1, are computed. We define a $Stats_i$ function bellow. We have 2 instances for both the zeroth and the first index, shown bellow is the zeroth index and the first is omitted.

$$Stats_0\left(w\right) :: \mathbb{R}^{K_0,K_1} \rightarrow \mathbb{R}^{K_0} \tag{App.85}$$

$$s_{l1} = \frac{1}{K_0}\sum_{i}^{K_0} abs(w_i j) \tag{App.86}$$

$$s_{l2} = \sqrt{\frac{1}{K_0}\sum_{i}^{K_0}(w_i j)^2} \tag{App.87}$$

$$s_{\mu} = \frac{1}{K_0}\sum_{i}^{K_0} w_i j \tag{App.88}$$

$$s_{\sigma} = \sqrt{\frac{1}{K_0}\sum_{i}^{K_0}((s_{\mu})_i - w_i j)^2} \tag{App.89}$$

$$Stats_0\left(w\right) = [s_{l1}, s_{l2}, s_{\mu}, s_{\sigma}] \tag{App.90}$$

We the compute statistics of the weight matrix below, and above. We tile the statistics to the appropriate dimensions and concatenate with normalized inputs as well as with the bias (also tiled appropriately).

$$(s_0)_{ijk} = (Stats_0\left(w^{l-1}, 0\right))_j \tag{App.91}$$

$$(s_1)_{ijk} = (Stats_1\left(w^{l}, 1\right))_i \tag{App.92}$$

$$m_2 = BatchNorm(m_1, [0, 1]) \tag{App.93}$$

$$\hat{b}_{ijk} = b^l_j \tag{App.94}$$

$$m_3 = [s_0; s_1; m_2; \hat{b}] \tag{App.95}$$

With the inputs prepared, we next perform a series of convolutions on batch and unit dimensions, (0, 1).

$$m_4 = ConvBatch\left(m_3, S = \lambda_{computehsize}, K = 3\right) \tag{App.96}$$

$$m_5 = relu\left(BatchNorm\left(m_4, [0, 1]\right)\right) \tag{App.97}$$

$$m_6 = ConvUnit\left(m_5, S = \lambda_{computehsize}, K = 3\right) \tag{App.98}$$

$$m_7 = relu\left(BatchNorm\left(m_6, [0, 1]\right)\right) \tag{App.99}$$

$$m_8 = ConvBatch\left(m_7, S = \lambda_{computehsize}, K = 3\right) \tag{App.100}$$

$$m_9 = relu\left(BatchNorm\left(m_8, [0, 1]\right)\right) \tag{App.101}$$

$$m_{10} = ConvUnit\left(m_9, S = \lambda_{computehsize}, K = 3\right) \tag{App.102}$$

$$m_{11} = relu\left(BatchNorm\left(m_{10}, [0, 1]\right)\right) \tag{App.103}$$

The result is then output.

$$computeH\left(\cdot; \theta_{computeH}\right) = m_{11} \tag{App.104}$$

We set $\lambda_{computehsize} = 64$.

I EXPERIMENTAL DETAILS

I.1 META TRAINING

I.1.1 TRAINING DATA DISTRIBUTION

We trained on a data distribution consisting of tasks sampled uniformly over the following datasets. Half of our training tasks where constructed off of a dataset consisting of 1000 font rendered

characters. We resized these to 14x14 black and white images. We call this the glyph dataset. "Alphabet" is an example of such a dataset consisting of alphabet characters. We used a mixture 10, 13, 14, 17, 20, and 30 way classification problems randomly sampled, as well as sampling from three 10-way classification problems sampled from specific types of images: letters of the alphabet, math symbols, and currency symbols. For half of the random sampling and all of the specialized selection we apply additional augmentation. This augmentation consists of random rotations (up to 360 degrees) and shifts up to +-5 pixels in the x and y directions. The parameters of the augmentation were inserted into the regression target of the MetaObjective as a curriculum of sorts and to provide diverse training signal.

In addition to the the glyph set, we additionally used Cifar10, resized to 16x16, as well as 10, 15, 20, and 25 way classification problems from imagenet. Once again we resized to 16x16 for compute reasons.

With a dataset selected, we apply additional augmentation with some probability consisting of the following augmentations. A per task dropout mask (fixed mask across all images in that task). A per example dropout mask (a random mask per image). A permutation sampled from a fixed number of pre created permutations per class. A per image random shift in the x direction each of image. All of these additional augmentations help with larger domain transfer.

I.2 Meta-Optimization

We employ Adam Kingma and Ba (2014) as our meta-optimizer. We use a learning rate schedule of 3e-4 for the first 100k steps, then 1e-4 for next 50k steps, then 2e-5 for remainder of meta-training. We use gradient clipping of norm 5 on minibatchs of size 256.

We compute our meta-objective by averaging 5 evaluation of the linear regression. We use a ridge penalty of 0.1 for all this work.

When computing truncated gradients, we initially sample the number of unrolled applications of the UnsupervisedUpdate in a uniform distribution of [2,4]. Over the course of 50k steps we uniformly increase this to [8,15]. This increases meta-training speed and stability as large unrolls early in training can be unstable and don't seem to provide any value.

For sampling the number of truncated steps, we use a shifted normal distribution – a normal distribution with the same mean and std. We chose this based on the expected distribution of the training step, ϕ iteration, across the cluster of workers. We initially set the STD low, 20, but slowly increased it over the course of 5000 steps to 20k steps. This slow increase also improved stability and training speed.

I.3 Experimental setup

For each experimental figure, we document the details.

I.3.1 Objective Function Mismatch

The VAE we used consists of 3 layers, size 128, with ReLU activations and batch norm between each layer. We then learn a projection to mean and log std of size 32. We sample, and use the inverse architecture to decode back to images. We use a quantized normal distribution (once again parameterized as mean and log std) as a posterior. We train with Adam with a learning rate of 1e-4. To isolate the effects of objective function mismatch and overfitting, we both train on the unlabeled training set and evaluate on the labeled training set instead of a validation set.

I.3.2 Generalization: Dataset and Domain

We use a 4 layer, size 128 unit architecture with a 32 layer embedding for all models. We select performance at 100k training steps for the VAE, and 3k for our learned optimizer.

Our supervised learning baseline consists of the same architecture for the base model but with an additional layer that outputs log probabilities. We train with cross entropy loss on a dataset consisting of only 10 examples per class (to match the other numbers). Surprisingly, the noise from batch norm acts as a regularizer, and allowing us to avoid needing a complex early stopping scheme as test set

performance simply plateaus over the course of 10k steps. We train with Adam with a learning rate of 3e-3, selected via a grid over learning rate on test set performance. In this setting, having a true validation set would dramatically lower the amount of labeled data available (only 100 labeled examples) and using the test set only aids in the this baseline's performance.

For the IMDB experiments, we tokenized and selected the top 1K words with an additional set of tokens for unknown, sentence start, and sentence end. Our encoding consisted of a 1 if the word is present, otherwise 0. We used the same 4 layer, 128 hidden units with an addition layer outputting a 32 dimensional embedding.

I.3.3 GENERALIZATION: NETWORK ARCHITECTURE

We used ReLU activations and 4 layers of size 128 with an additional layer to 32 units unless otherwise specified by the specific experiment.

I.3.4 EXISTING META LEARNING MODELS

We trained prototypical networks Snell et al. (2017) on either intact or shuffled mini-Imagenet images. For shuffled images, we generated a fixed random permutation for the inputs independently for every instantiation of the base network (or 'episode' in the meta-learning literature Vinyals et al. (2016); Ravi and Larochelle (2016)). The purpose of shuffling was to demonstrate the inductive bias of this type of meta-learning, namely that they do not generalize across data domain. Note that the base network trained was the same fully connected architecture like that used in this paper (3 layers, size 128, with ReLU activations and batch normalization between layers). Though the original paper used a convolutional architecture, here we swapped it with the fully connected architecture because the tied weights in a convolutional model do not make sense with shuffled pixels.