Stochastic Hyperparameter Optimization through Hypernetworks

Anonymous Author(s)
Affiliation
Address
email

Abstract

Machine learning models are often tuned by nesting optimization of model weights inside the optimization of hyperparameters. We give a method to collapse this nested optimization into joint stochastic optimization of weights and hyperparameters. Our process trains a neural network to output approximately optimal weights as a function of hyperparameters. We show that our technique converges to locally optimal weights and hyperparameters for sufficiently large hypernetworks. We compare this method to standard hyperparameter optimization strategies and demonstrate its effectiveness for tuning thousands of hyperparameters.

1 Introduction

Model selection and hyperparameter tuning is a significant bottleneck in designing predictive models. Hyperparameter optimization is a nested optimization: The inner optimization finds model parameters w which minimize the training loss $\mathcal{L}_{\text{Train}}$ given hyperparameters λ. The outer optimization chooses λ to reduce a validation loss $\mathcal{L}_{\text{Valid.}}$:

$$\underset{\lambda}{\operatorname{argmin}} \underset{\text{Valid.}}{\mathcal{L}} \left(\underset{\mathrm{w}}{\operatorname{argmin}} \underset{\text{Train}}{\mathcal{L}} (\mathrm{w}, \lambda) \right) \tag{1}$$

Standard practice in machine learning solves (1) by gradient-free optimization of hyperparameters, such as grid search or random search. Each set of hyperparameters is evaluated by re-initializing weights and training the model to completion. Re-training a model from scratch is wasteful if the hyperparameters change by a small amount. Some approaches, such as Hyperband (Li et al., 2016) and freeze-thaw Bayesian optimization (Swersky et al., 2014), resume model training and do not waste this effort. However, these methods often scale poorly beyond 10 to 20 dimensions.

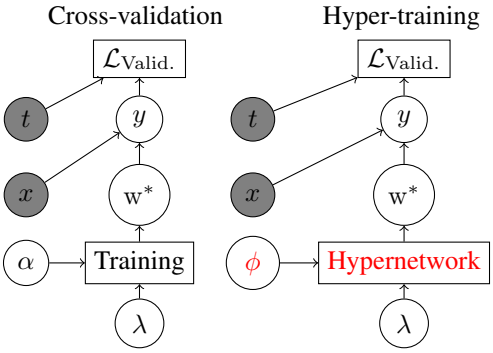

Figure 1: *Left:* A typical computational graph for cross-validation, where α are the optimizer parameters, and λ are training loss hyperparameters. It is expensive to differentiate through the entire training procedure. *Right:* The proposed computational graph with our changes in red, where ϕ are the hypernetwork parameters. We can cheaply differentiate through the hypernetwork to optimize the validation loss $\mathcal{L}_{\text{Valid.}}$ with respect to hyperparameters λ. We use x, t, and y to refer to a data point, its label, and a prediction respectively.

How can we avoid re-training from scratch each time? Note that the optimal parameters w are a deterministic function of the hyperparameters λ:

$$\mathrm{w}^*(\lambda) = \underset{\mathrm{w}}{\operatorname{argmin}} \underset{\text{Train}}{\mathcal{L}} (\mathrm{w}, \lambda) \tag{2}$$

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

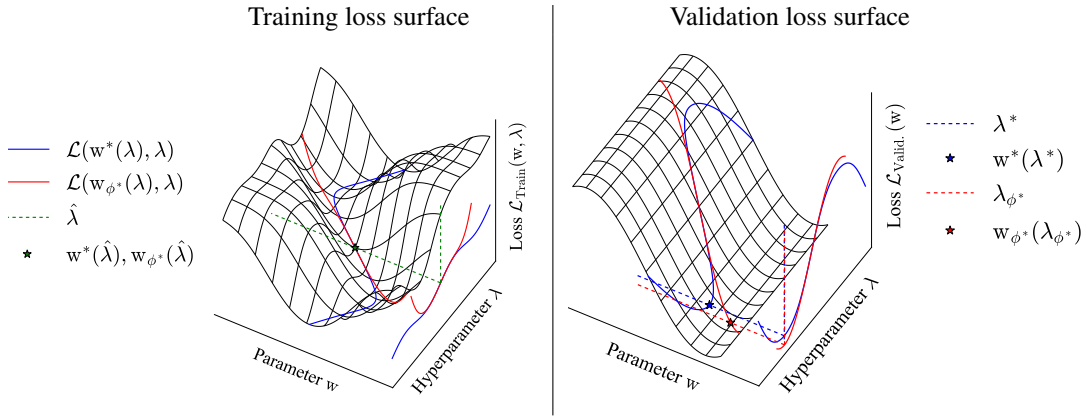

Figure 2: A visualization of exact (blue) and approximate (red) optimal weights as a function of hyperparameters. The approximately optimal weights w_{ϕ^*} are output by a linear model fit at $\hat{\lambda}$. The true optimal hyperparameter is λ^*, while the hyperparameter estimated using approximately optimal weights is nearby at λ_{ϕ^*}.

We propose to *learn this function*. Specifically, we train a neural network that takes hyperparameters as input, and outputs an approximately optimal set of weights.

This formulation provides two major benefits: First, we can train the hypernetwork to convergence using stochastic gradient descent (SGD) without training any particular model to completion. Second, differentiating through the hypernetwork allows us to optimize hyperparameters with stochastic gradient-based optimization.

2 Training a network to output optimal weights

How can we teach a *hypernetwork* (Ha et al., 2016) to output approximately optimal weights to another neural network? The basic idea is that at each iteration, we ask a hypernetwork to output a set of weights given some hyperparameters: $\mathrm{w} = \mathrm{w}_\phi(\lambda)$. Instead of updating the weights w using the training loss gradient $\partial \mathcal{L}_{\text{Train}}(\mathrm{w}) / \partial \mathrm{w}$, we update the hypernetwork weights ϕ using the chain rule: $\partial \mathcal{L}_{\text{Train}}(\mathrm{w}_\phi) / \partial \mathrm{w}_\phi \partial \mathrm{w}_\phi / \partial \phi$. This formulation allows us to optimize the hyperparameters λ with the validation loss gradient $\partial \mathcal{L}_{\text{Valid.}}(\mathrm{w}_\phi(\lambda)) / \partial \mathrm{w}_\phi(\lambda) \partial \mathrm{w}_\phi(\lambda) / \partial \lambda$. We call this method *hyper-training* and contrast it with standard training methods.

We call the function $\mathrm{w}^*(\lambda)$ that outputs optimal weights for hyperparameters a *best-response function*. At convergence, we want our hypernetwork $\mathrm{w}_\phi(\lambda)$ to match the best-response function closely.

Our method is closely related to the concurrent work of Brock et al. (2017), whose SMASH algorithm also approximates the optimal weights as a function of model architectures, to perform a gradient-free search over discrete model structures. Their work focuses on efficiently estimating the performance of a variety of model architectures, while we focus on efficiently exploring continuous spaces of models. We further extend this idea by formulating an algorithm to optimize the hypernetwork and hyperparameters jointly. Joint optimization of parameters and hyperparameters addresses one of the main weaknesses of SMASH, which is that the the hypernetwork must be very large to learn approximately optimal weights for many different settings. During joint optimization, the hypernetwork need only model approximately optimal weights for the neighborhood around the current hyperparameters, allowing us to use even linear hypernetworks.

2.1 Advantages of hypernetwork-based optimization

Hyper-training is a method to learn a mapping from hyperparameters to validation loss which is differentiable and cheap to evaluate. We can compare hyper-training to other model-based hyperparameter schemes. Bayesian optimization (e.g., Lizotte (2008); Snoek et al. (2012)) builds a model of the validation loss as a function of hyperparameters, usually using a Gaussian process (e.g., Rasmussen & Williams (2006)) to track uncertainty. This approach has several disadvantages compared to hyper-training.

First, obtaining data for standard Bayesian optimization requires optimizing models from initialization for each set of hyperparameters. In contrast, hyper-training never needs to optimize any one model fully removing choices like how many models to train and for how long.

Second, standard Bayesian optimization treats the validation loss as a black-box function: $\hat{\mathcal{L}_{\text{Valid.}}}(\lambda) = f(\lambda)$. In contrast, hyper-training takes advantage of the fact that the validation loss is a known, differentiable function: $\hat{\mathcal{L}_{\text{Valid.}}}(\lambda) = \mathcal{L}_{\text{Valid.}}(\mathrm{w}_\phi(\lambda))$. This information removes the need to learn a model of the validation loss. This function can also be evaluated stochastically by sampling points from the validation set.

Hyper-training has a benefit of learning hyperparameter to optimized weight mapping, which is substituted into the validation loss. This often has a better inductive bias for learning hyperparameter to validation loss than directly learning the loss. Also, the hypernetwork learns continuous best-responses, which may be a beneficial prior for finding weights by enforcing stability.

2.2 Limitations of hypernetwork-based optimization

We can apply this method to unconstrained continuous bi-level optimization problems with an inner loss function with inner parameters, and an outer loss function with outer parameters. What sort of parameters can be optimized by our approach? Hyperparameters typically fall into two broad categories: 1) Optimization hyperparameters, such as learning rates, which affect the choice of locally optimal point converged to, and 2) regularization or model architecture parameters which change the set of locally optimal points. Hyper-training *does not have inner optimization parameters* because there is no internal training loop, so we can not optimize these. However, we must still choose optimization parameters for the fused optimization loop. In principle, hyper-training can handle discrete hyperparameters, but does not offer particular advantages for optimization over continuous hyperparameters.

Another limitation is that our approach only proposes making local changes to the hyperparameters, and does not do uncertainty-based exploration. Uncertainty can be incorporated into the hypernetwork by using stochastic variational inference as in Blundell et al. (2015), and we leave this for future work. Finally, it is not obvious how to choose the training distribution of hyperparameters $p(\lambda)$. If we do not sample a sufficient range of hyperparameters, the implicit estimated gradient of the validation loss w.r.t. the hyperparameters may be inaccurate. We discuss several approaches to this problem in section 2.4.

A clear difficulty of this approach is that hypernetworks can require several times as many parameters as the original model. For example, training a fully-connected hypernetwork with 1 hidden layer of H units to output D parameters requires at least $D \times H$ hypernetwork parameters. To address this problem, in section 2.4, we propose an algorithm that only trains a linear model mapping hyperparameters to model weights.

2.3 Asymptotic convergence properties

Algorithm 2 trains a hypernetwork using SGD, drawing hyperparameters from a fixed distribution $p(\lambda)$. This section proves that Algorithm 2 converges to a local best-response under mild assumptions. In particular, we show that, for a sufficiently large hypernetwork, the choice of $p(\lambda)$ does not matter as long as it has sufficient support. Notation as if w_ϕ has a unique solution for ϕ or w is used for simplicity, but is not true in general.

Theorem 2.1. *Sufficiently powerful hypernetworks can learn continuous best-response functions, which minimizes the expected loss for all hyperparameter distributions with convex support.*

$$\text{There exists } \phi^*, \text{ such that for all } \lambda \in \text{support}(p(\lambda)),$$

$$\mathop{\mathcal{L}}_{\text{Train}}(\mathrm{w}_{\phi^*}(\lambda), \lambda) = \min_{\mathrm{w}} \mathop{\mathcal{L}}_{\text{Train}}(\mathrm{w}, \lambda) \quad \text{and} \quad \phi^* = \operatorname*{argmin}_{\phi} \mathop{\mathbb{E}}_{p(\lambda')} \left[\mathop{\mathcal{L}}_{\text{Train}}(\mathrm{w}_\phi(\lambda'), \lambda') \right]$$

Proof. If w_ϕ is a universal approximator (Hornik, 1991) and the best-response is continuous in λ (which allows approximation by w_ϕ), then there exists optimal hypernetwork parameters ϕ^* such that for all hyperparameters λ, $\mathrm{w}_{\phi^*}(\lambda) = \operatorname{argmin}_{\mathrm{w}} \mathcal{L}_{\text{Train}}(\mathrm{w}, \lambda)$. Thus, $\mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi^*}(\lambda), \lambda) = \min_{\mathrm{w}} \mathcal{L}_{\text{Train}}(\mathrm{w}, \lambda)$. In other words, universal approximator hypernetworks can learn continuous best-responses.

Algorithm 1 Standard cross-validation with stochastic optimization

for $i = 1, \ldots, T_{\text{outer}}$ **do**
  initialize w
  $\lambda = \Lambda\left(\lambda^{(1:i)}, \mathcal{L}_{\text{Valid.}}\left(\mathrm{w}^{(1:i)}\right)\right)$
  loop
    $\mathbf{x} \sim$ Training data
    $\mathrm{w} \mathrel{-}= \alpha \nabla_{\mathrm{w}} \mathcal{L}_{\text{Train}}(\mathrm{w}, \lambda, \mathbf{x})$
  $\lambda^i, \mathrm{w}^i = \lambda, \mathrm{w}$
$i = \underset{i}{\operatorname{argmin}}\, \mathcal{L}_{\text{Valid.}}(\mathrm{w}^{(i)}, \mathbf{x})$
Return $\lambda^{(i)}, \mathrm{w}^{(i)}$

Algorithm 2 Optimization of hypernetwork, then hyperparameters

initialize ϕ
initialize $\hat{\lambda}$
loop
  $\mathbf{x} \sim$ Training data, $\lambda \sim p(\lambda)$
  $\phi \mathrel{-}= \alpha \nabla_{\phi} \mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda, \mathbf{x})$
loop
  $\mathbf{x} \sim$ Validation data
  $\hat{\lambda} \mathrel{-}= \beta \nabla_{\hat{\lambda}} \mathcal{L}_{\text{Valid.}}(\mathrm{w}_{\phi}(\hat{\lambda}), \mathbf{x})$
Return $\hat{\lambda}, \mathrm{w}_{\phi}(\hat{\lambda})$

Algorithm 3 Joint optimization of hypernetwork and hyperparameters

initialize ϕ
initialize $\hat{\lambda}$
loop
  $\mathbf{x} \sim$ Training data, $\lambda \sim p(\lambda|\hat{\lambda})$
  $\phi \mathrel{-}= \alpha \nabla_{\phi} \mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda, \mathbf{x})$
  $\mathbf{x} \sim$ Validation data
  $\hat{\lambda} \mathrel{-}= \beta \nabla_{\hat{\lambda}} \mathcal{L}_{\text{Valid.}}(\mathrm{w}_{\phi}(\hat{\lambda}), \mathbf{x})$
Return $\hat{\lambda}, \mathrm{w}_{\phi}(\hat{\lambda})$

A comparison of standard hyperparameter optimization, our first algorithm, and our joint algorithm. Here, Λ refers to a generic hyperparameter optimization. Instead of updating weights w using the loss gradient $\partial \mathcal{L}(\mathrm{w}) / \partial \mathrm{w}$, we update hypernetwork weights ϕ and hyperparameters λ using the chain rule: $\partial \mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}) / \partial \mathrm{w}_{\phi} \, \partial \mathrm{w}_{\phi} / \partial \phi$ or $\partial \mathcal{L}_{\text{Valid.}}(\mathrm{w}_{\phi}(\lambda)) / \partial \mathrm{w}_{\phi}(\lambda) \, \partial \mathrm{w}_{\phi}(\lambda) / \partial \lambda$ respectively. This allows our method to use gradient-based hyperparameter optimization.

Substituting ϕ^* into the training loss gives $\mathbb{E}_{p(\lambda)}[\mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi^*}(\lambda), \lambda)] = \mathbb{E}_{p(\lambda)}[\min_{\phi} \mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda)]$. By Jensen's inequality, $\min_{\phi} \mathbb{E}_{p(\lambda)}[\mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda)] \geq \mathbb{E}_{p(\lambda)}[\min_{\phi} \mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda)]$. To satisfy Jensen's requirements, we have $\min_{\phi}$ as our convex function on the convex vector space of functions $\{\mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda)$ for $\lambda \in \text{support}(p(\lambda))\}$. To guarantee convexity of the vector space we require that $\text{support}(p(\lambda))$ is convex and $\mathcal{L}_{\text{Train}}(\mathrm{w}, \lambda) = \mathbb{E}_{\mathbf{x} \sim \text{Train}}[\mathcal{L}_{\text{Pred}}(\mathbf{x}, \mathrm{w})] + \mathcal{L}_{\text{Reg}}(\mathrm{w}, \lambda)$ with $\mathcal{L}_{\text{Reg}}(\mathrm{w}, \lambda) = \lambda \cdot \mathcal{L}(\mathrm{w})$. Thus, $\phi^* = \operatorname{argmin}_{\phi} \mathbb{E}_{p(\lambda)}[\mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda)]$. In other words, if the hypernetwork learns the best-response it will simultaneously minimize the loss for every point in $\text{support}(p(\lambda))$. □

Thus, having a universal approximator and a continuous best-response implies for all $\lambda \in \text{support}(p(\lambda))$, $\mathcal{L}_{\text{Valid.}}(\mathrm{w}_{\phi^*}(\lambda)) = \mathcal{L}_{\text{Valid.}}(\mathrm{w}^*(\lambda))$, because $\mathrm{w}_{\phi^*}(\lambda) = \mathrm{w}^*(\lambda)$. Thus, under mild conditions, we will learn a best-response in the support of the hyperparameter distribution. If the best-response is differentiable, then there is a neighborhood about each hyperparameter where the best-response is approximately linear. If the support of the hyperparameter distribution is this neighborhood, then we can learn the best-response locally with linear regression.

In practice, there are no guarantees about the network being a universal approximator or the finite-time convergence of optimization. The optimal hypernetwork will depend on the hyperparameter distribution $p(\lambda)$, not just the support of this distribution. We appeal to experimental results that our method is feasible in practice.

2.4 Jointly train parameters and hyperparameters

Theorem 2.1 holds for any $p(\lambda)$. In practice, we should choose a $p(\lambda)$ that puts most of its mass on promising hyperparameter values, because it may not be possible to learn a best-response for all hyperparameters due to limited hypernetwork capacity. Thus, we propose Algorithm 3, which only tries to match a best-response locally. We introduce a "current" hyperparameter $\hat{\lambda}$, which is updated each iteration. We define a conditional hyperparameter distribution, $p(\lambda|\hat{\lambda})$, which only puts mass close to $\hat{\lambda}$.

Algorithm 3 combines the two phases of Algorithm 2 into one. Instead of first learning a hypernetwork that can output weights for any hyperparameter then optimizing the hyperparameters, Algorithm 3 only samples hyperparameters close to the current $\hat{\lambda}$, and then updates $\hat{\lambda}$ using a validation batch.

Algorithm 4 Simplified joint training of hypernetwork and hyperparameters

initialize $\phi, \hat{\lambda}$
loop
  $\mathbf{x} \sim$ Training data, $\mathbf{x}' \sim$ Validation data
  $\phi \mathrel{-}= \alpha \nabla_{\phi} \mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\hat{\lambda}), \hat{\lambda}, \mathbf{x})$
  $\hat{\lambda} \mathrel{-}= \beta \nabla_{\hat{\lambda}} \mathcal{L}_{\text{Valid.}}(\mathrm{w}_{\phi}(\hat{\lambda}), \mathbf{x}')$
Return $\hat{\lambda}, \mathrm{w}_{\phi}(\hat{\lambda})$

Algorithm 4 builds on Algorithm 3 by using gradient updates on $\hat{\lambda}$ as a source of noise. This variant does not have asymptotic guarantees, but performs similarly to Algorithm 3 in practice.

perparameters near the current guess. This means the hypernetwork just has to be trained to estimate good enough weights for a small set of hyperparameters. There is an extra cost of having to re-train the hypernetwork each time we update λ. The locally-trained hypernetwork can then be used to provide gradients to update the hyperparameters based on validation set performance.

How simple can we make the hypernetwork, and still obtain useful gradients to optimize hyperparameters? Consider the case in our experiments where the hypernetwork is a linear function of the hyperparameters and the conditional hyperparameter distribution is $p(\lambda|\hat{\lambda}) = \mathcal{N}(\hat{\lambda}, \sigma\mathbb{I})$ for some small σ. This hypernetwork learns a tangent hyperplane to a best-response function and only needs to make minor adjustments at each step if the hyperparameter updates are sufficiently small. We can further restrict the capacity of a linear hypernetwork by factorizing its weights, effectively adding a bottleneck layer with a linear activation and a small number of hidden units.

3 Related work

Our work is complementary to the SMASH algorithm of Brock et al. (2017), with section 2 discussing our differences.

Model-free approaches Model-free approaches use only trial-and-error to explore the hyperparameter space. Simple model-free approaches applied to hyperparameter optimization include grid search and random search (Bergstra & Bengio, 2012). Hyperband (Li et al., 2016) combines bandit approaches with modeling the learning procedure.

Model-based approaches Model-based approaches try to build a surrogate function, which can allow gradient-based optimization or active learning. A common example is Bayesian optimization. Freeze-thaw Bayesian optimization can condition on partially-optimized model performance.

Optimization-based approaches Another line of related work attempts to directly approximate gradients of the validation loss with respect to hyperparameters. Domke (2012) proposes to differentiate through unrolled optimization to approximate best-responses in nested optimization and Maclaurin et al. (2015a) differentiate through entire unrolled learning procedures. DrMAD (Fu et al., 2016) approximates differentiating through an unrolled learning procedure to relax memory requirements for deep neural networks. HOAG (Pedregosa, 2016) finds hyperparameter gradients with implicit differentiation by deriving an implicit equation for the gradient with optimality conditions. Franceschi et al. (2017) study forward and reverse-mode differentiation for constructing hyperparameter gradients. Also, Feng & Simon (2017) establish conditions where the validation loss of best-responding weights are almost everywhere smooth, allowing gradient-based training of hyperparameters.

A closely-related procedure to our method is the $T1 - T2$ method of Luketina et al. (2016), which also provides an algorithm for stochastic gradient-based optimization of hyperparameters. The convergence of their procedure to local optima of the validation loss depends on approximating the Hessian of the training loss for parameters with the identity matrix. In contrast, the convergence of our method depends on having a suitably powerful hypernetwork.

Game theory Best-response functions are extensively studied as a solution concept in discrete and continuous multi-agent games (e.g., Fudenberg & Levine (1998)). Games where learning a best-response can be applied include adversarial training (Goodfellow et al., 2014), or Stackelberg competitions (e.g., Brückner & Scheffer (2011)). For adversarial training, the analog of our method is a discriminator who observes the generator's parameters.

4 Experiments

In our experiments, we examine the standard example of stochastic gradient-based optimization of neural networks, with a weight regularization penalty. Some gradient-based methods explicitly use the gradient of a loss, while others use the gradient of a learned surrogate loss. Hyper-training learns and substitutes a surrogate best-response function into a real loss. We may contrast our algorithm with methods learning the loss like Bayesian optimization, gradient-based methods only

handling hyperparameters that affect the training loss and gradient-based methods which can handle optimization parameters. The best comparison for hyper-training is to gradient-based methods which only handle parameters affecting the training loss because other methods apply to a more general set of problems. In this case, we write the training and validation losses as:

$$\underset{\text{Train}}{\mathcal{L}}(\mathrm{w}, \lambda) = \underset{\mathbf{x} \sim \text{Train}}{\mathbb{E}} \left[\underset{\text{Pred}}{\mathcal{L}}(\mathbf{x}, \mathrm{w}) \right] + \underset{\text{Reg}}{\mathcal{L}}(\mathrm{w}, \lambda), \qquad \underset{\text{Valid.}}{\mathcal{L}}(\mathrm{w}) = \underset{\mathbf{x} \sim \text{Valid.}}{\mathbb{E}} \left[\underset{\text{Pred}}{\mathcal{L}}(\mathbf{x}, \mathrm{w}) \right]$$

In all experiments, Algorithms 2 or 3 are used to optimize weights with a mean squared error on MNIST (LeCun et al., 1998) with $\mathcal{L}_{\text{Reg}}$ as an L_2 weight decay penalty weighted by $\exp(\lambda)$. The elementary model has $7,850$ weights. All hidden units in the hypernetwork have a ReLU activation (Nair & Hinton, 2010) unless otherwise specified. Autograd (Maclaurin et al., 2015b) was used to compute all derivatives. For each experiment, the minibatch samples 2 pairs of hyperparameters and up to $1,000$ training data points. We used Adam for training the hypernetwork and hyperparameters, with a step size of 0.0001. We ran all experiments on a CPU.

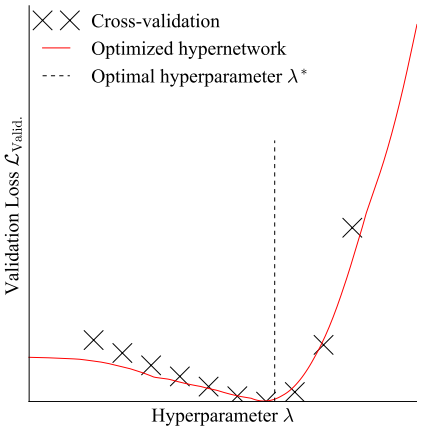

Figure 3: The validation loss of a neural net, estimated by cross-validation (crosses) or by a hypernetwork (line), which outputs $7,850$-dimensional network weights. Cross-validation requires optimizing from scratch each time. The hypernetwork can be used to evaluate the validation loss cheaply.

4.1 Learning a global best-response

Our first experiment, shown in Figure 3, demonstrates learning a global approximation to a best-response function using Algorithm 2. To make visualization of the regularization loss easier, we use 10 training data points to exacerbate overfitting. We compare the performance of weights output by the hypernetwork to those trained by standard cross-validation (Algorithm 1). Thus, elementary weights were randomly initialized for each hyperparameter choice and optimized using Adam (Kingma & Ba, 2014) for $1,000$ iterations with a step size of 0.0001.

When training the hypernetwork, hyperparameters were sampled from a broad Gaussian distribution: $p(\lambda) = \mathcal{N}(0, 1.5)$. The hypernetwork has 50 hidden units which results in $400,450$ parameters of the hypernetwork.

The minimum of the best-response in Figure 3 is near the real minimum of the validation loss, which shows a hypernetwork can satisfactorily approximate a global best-response function in small problems.

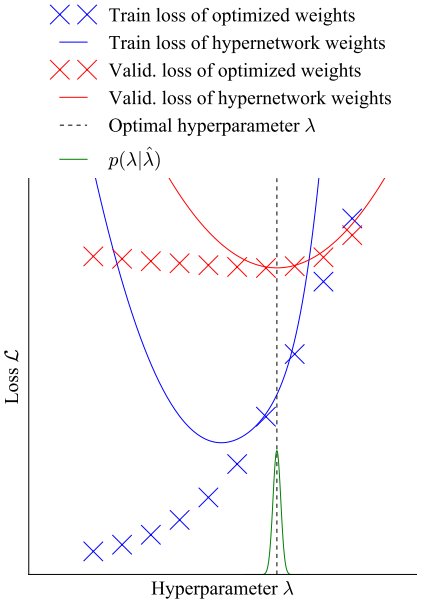

Figure 4: The losses of a neural net, estimated by cross-validation or a hypernetwork, which outputs $7,850$-dimensional network weights. A linear hypernetwork which has limited capacity making it only accurate where the hyperparameter distribution puts mass.

4.2 Learning a local best-response

Figure 4 shows the same experiment, but using the Algorithm 3. The fused updates result in finding a best-response approximation whose minimum is the actual minimum faster than the prior experiment. The conditional hyperparameter distribution is given by $p(\lambda|\hat{\lambda}) = \mathcal{N}(\hat{\lambda}, 0.00001)$. The hypernetwork is a linear model, with only $15,700$ weights. We use the same optimizer as the global best-response to update both the hypernetwork and the hyperparameters.

Again, the minimum of the best-response at the end of training minimizes the validation loss. This experiment shows that using only a locally-trained linear best-response function can give sufficient gradient information to optimize hyperparameters on

a small problem. Algorithm 3 is also less computationally expensive than Algorithms 1 or 2.

4.3 Hyper-training and unrolled optimization

To compare hyper-training with other gradient-based hyperparameter optimization methods, we train models with $7,850$ hyperparameters and a separate L_2 weight decay applied to each weight in a 1 layer (linear) model. The conditional hyperparameter distribution and optimizer for the hypernetwork and hyperparameters is the same the prior experiment. We factorize the weights for the model by selecting a hypernetwork with 10 hidden units. The factorized linear hypernetwork has 10 hidden units giving $164,860$ weights. Each hypernetwork iteration is $2 \cdot 10$ times as expensive as an iteration on just the model because there is the same number of hyperparameters as model parameters.

Figure 5, top, shows that Algorithm 3 converges more quickly than the unrolled reverse-mode optimization introduced in Maclaurin et al. (2015a) and implemented by Franceschi et al. (2017). Hyper-training reaches sub-optimal solutions because of limitations on how many hyperparameters can be sampled for each update but overfits validation data less than unrolling. Standard Bayesian optimization cannot be scaled to this many hyperparameters. Thus, this experiment shows Algorithm 3 can efficiently partially optimize thousands of hyperparameters. It may be useful to combine these methods by using a hypernetwork to output initial parameters and then unrolling several steps of optimization to differentiate through.

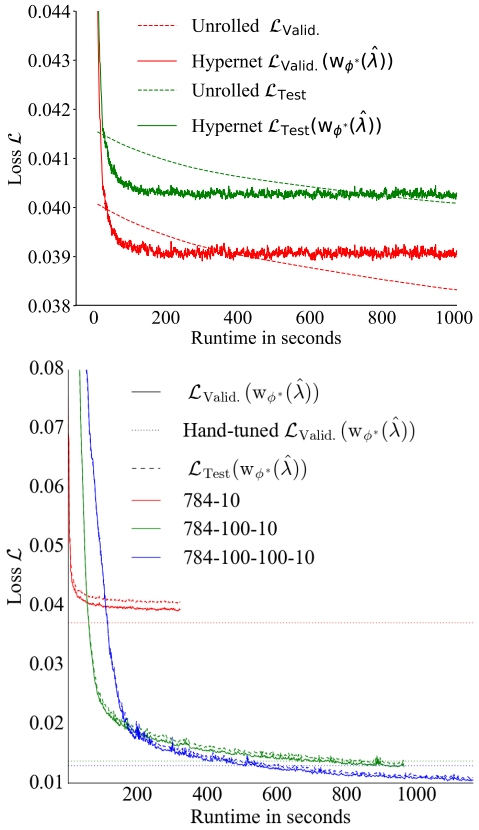

Figure 5: Validation and test losses during hyperparameter optimization with a separate L_2 weight decay applied to each weight in the model. Thus, models with more parameters have more hyperparameters. *Top:* We solve the $7,850$-dimensional hyperparameter optimization problem with a linear model and multiple algorithms. Hypernetwork-based optimization converges to a sub-optimal solution faster than unrolled optimization from Maclaurin et al. (2015a). *Bottom:* Hyper-training is applied different layer configurations in the model.

4.4 Optimizing with deeper networks

To see if we can optimize deeper networks with hyper-training we optimize models with 1, 2, and 3 layers and a separate L_2 weight decay applied to each weight. The conditional hyperparameter distribution and optimizer for the hypernetwork and hyperparameters is the same the prior experiment. We factorize the weights for each model by selecting a hypernetwork with 10 hidden units.

Figure 5, bottom, shows that Algorithm 3 can scale to networks with multiple hidden layers and outperform hand-tuned settings. As we add more layers the difference between validation loss and testing loss decreases, and the model performs better on the validation set. Future work should compare other architectures like recurrent or convolutional networks. Additionally, note that more layers perform with lesser training (not shown), validation, and test losses, instead of lower training loss and higher validation or test loss. This performance indicates that using weight decay on each weight could be a prior for generalization, or that hyper-training enforces another useful prior like the continuity of a best-response.

4.5 Estimating weights versus estimating loss

Our approach differs from Bayesian optimization which attempts to directly model the validation loss of optimized weights, where we try to learn to predict optimal weights. In this experiment, we untangle the reason for the better performance of our method: Is it because of a better inductive bias, or because our way can see more hyperparameter settings during optimization?

First, we constructed a hyper-training set: We optimized 25 sets of weights to completion, given randomly-sampled hyperparameters. We chose 25 samples since that is the regime in which we expect Gaussian process-based approaches to have the most significant advantage. We also constructed a validation set of $10,215$ (optimized weight, hyperparameter) tuples generated in the same manner. We then fit a Gaussian process (GP) regression model with an RBF kernel from sklearn on the validation loss data. A hypernetwork is fit to the same set of hyperparameters and data. Finally, we optimize another hypernetwork using Algorithm 2, for the same amount of time as building the GP training set. The two hypernetworks were linear models and trained with the same optimizer parameters as the $7,850$-dimensional hyperparameter optimization.

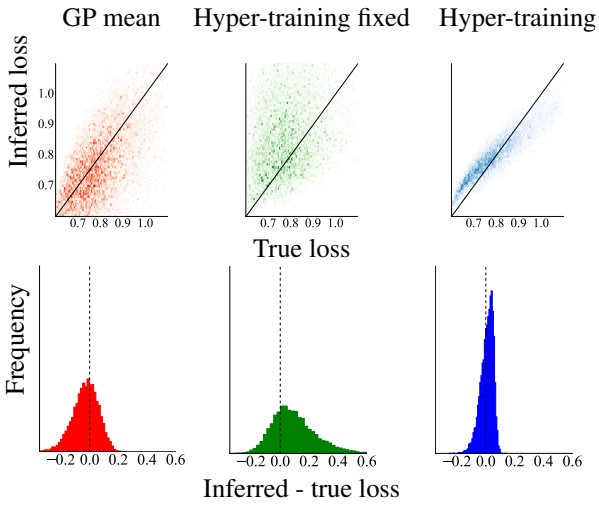

Figure 6: Comparing three approaches to inferring validation loss. *First column:* A Gaussian process, fit on 25 hyperparameters and the corresponding validation losses. *Second column:* A hypernetwork, fit on the same 25 hyperparameters and the corresponding optimized weights. *Third column:* Our proposed method, a hypernetwork trained with stochastically sampled hyperparameters. *Top row:* The distribution of inferred and true losses. The diagonal black line is where predicted loss equals true loss. *Bottom row:* The distribution of differences between inferred and true losses. The Gaussian process often under-predicts the true loss, while the hypernetwork trained on the same data tends to over-predict the true loss.

Figure 6 shows the distribution of prediction errors of these three models. We can see that the Gaussian process tends to underestimate loss. The hypernetwork trained with the same small fixed set of examples tends to overestimate loss. We conjecture that this is due to the hypernetwork producing bad weights in regions where it doesn't have enough training data. Because the hypernetwork must provide actual weights to predict the validation loss, poorly-fit regions will overestimate the validation loss. Finally, the hypernetwork trained with Algorithm 2 produces errors tightly centered around 0. The main takeaway from this experiment is a hypernetwork can learn more accurate surrogate functions than a GP for equal compute budgets because it views (noisy) evaluations of more points.

5 Conclusions and Future Work

In this paper, we addressed the question of tuning hyperparameters using gradient-based optimization, by replacing the training optimization loop with a differentiable hypernetwork. We gave a theoretical justification that sufficiently large networks will learn the best-response for all hyperparameters viewed in training. We also presented a simpler and more scalable method that jointly optimizes both hyperparameters and hypernetwork weights, allowing our method to work with manageably-sized hypernetworks.

Experimentally, we showed that hypernetworks could provide a better inductive bias for hyperparameter optimization than Gaussian processes fitting the validation loss empirically.

There are many directions to extend the proposed methods. For instance, the hypernetwork could be composed with several iterations of optimization, as an easily-differentiable fine-tuning step.

Or, hypernetworks could be incorporated into meta-learning schemes, such as MAML (Finn et al., 2017), which finds weights that perform a variety of tasks after unrolling gradient descent.

We also note that the prospect of optimizing thousands of hyperparameters raises the question of *hyper-regularization*, or regularization of hyperparameters.

References

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

Blundell, Charles, Cornebise, Julien, Kavukcuoglu, Koray, and Wierstra, Daan. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015.

Brock, Andrew, Lim, Theodore, Ritchie, JM, and Weston, Nick. Smash: One-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

Brückner, Michael and Scheffer, Tobias. Stackelberg games for adversarial prediction problems. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 547–555. ACM, 2011.

Domke, Justin. Generic methods for optimization-based modeling. In *Artificial Intelligence and Statistics*, pp. 318–326, 2012.

Feng, Jean and Simon, Noah. Gradient-based regularization parameter selection for problems with non-smooth penalty functions. *arXiv preprint arXiv:1703.09813*, 2017.

Finn, Chelsea, Abbeel, Pieter, and Levine, Sergey. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

Franceschi, Luca, Donini, Michele, Frasconi, Paolo, and Pontil, Massimiliano. Forward and reverse gradient-based hyperparameter optimization. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1165–1173. PMLR, 2017.

Fu, Jie, Luo, Hongyin, Feng, Jiashi, Low, Kian Hsiang, and Chua, Tat-Seng. Drmad: distilling reverse-mode automatic differentiation for optimizing hyperparameters of deep neural networks. *arXiv preprint arXiv:1601.00917*, 2016.

Fudenberg, Drew and Levine, David K. *The theory of learning in games*, volume 2. MIT press, 1998.

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

Ha, David, Dai, Andrew, and Le, Quoc V. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

Hornik, Kurt. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4 (2):251–257, 1991.

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

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

Li, Lisha, Jamieson, Kevin, DeSalvo, Giulia, Rostamizadeh, Afshin, and Talwalkar, Ameet. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

Lizotte, Daniel James. *Practical bayesian optimization*. University of Alberta, 2008.

Luketina, Jelena, Berglund, Mathias, Greff, Klaus, and Raiko, Tapani. Scalable gradient-based tuning of continuous regularization hyperparameters. In *International Conference on Machine Learning*, pp. 2952–2960, 2016.

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

Maclaurin, Dougal, Duvenaud, David, and Adams, Ryan P. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015b.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

Pedregosa, Fabian. Hyperparameter optimization with approximate gradient. In *International Conference on Machine Learning*, pp. 737–746, 2016.

Rasmussen, Carl Edward and Williams, Christopher KI. *Gaussian processes for machine learning*, volume 1. MIT press Cambridge, 2006.

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

Swersky, Kevin, Snoek, Jasper, and Adams, Ryan Prescott. Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

A Extra Experiments

A.1 Random Search and Bayesian Optimization Baselines

Here, optimize a model with $7,850$ and 10 hyperparameters, in which a separate L_2 weight decay is applied to the weights for each digit class in a linear regression model to assess hyper-trainings performance against other hyperparameter optimization algorithms. The conditional hyperparameter distribution and optimizer for the hypernetwork and hyperparameters is the same the prior experiments. Algorithm 3 is compared against random search and Bayesian optimization. Figure 7, right, shows that our method converges more quickly and to a better optimum than either alternative method, demonstrating that medium-sized hyperparameter optimization problems can be solved with Algorithm 3. Figure 7, left, shows that our method converges more quickly and to a better optimum than either alternative method, demonstrating that medium-sized hyperparameter optimization problems can be solved with Algorithm 3.

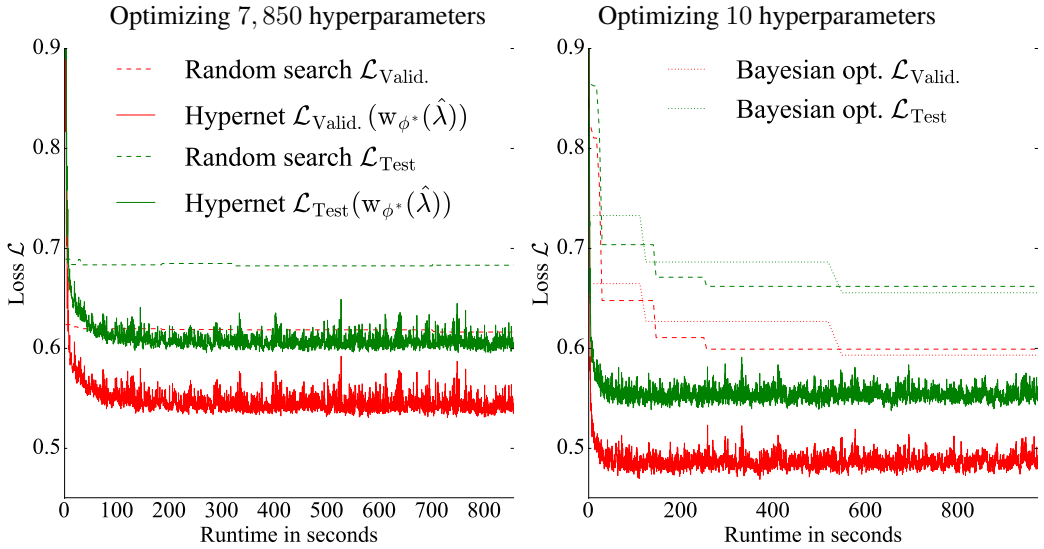

Figure 7: Validation and test losses during hyperparameter optimization. A separate L_2 weight decay is applied to the weights of each digit class, resulting in 10 hyperparameters. The weights w_{ϕ^*} are output by the hypernetwork for current hyperparameter $\hat{\lambda}$, while random losses are for the best result of a random search. hypernetwork-based optimization converges faster than random search or Bayesian optimization. We also observe significant overfitting of the hyperparameters on the validation set, which may be reduced by introducing hyperhyperparameters (parameters of the hyperparameter prior). The runtime includes the inner optimization for gradient-free approaches so that equal cumulative computational time is compared for each method.