STOCHASTIC HYPERPARAMETER OPTIMIZATION THROUGH HYPERNETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Machine learning models are usually 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 both weights and hyperparameters. Our method trains a neural network to output approximately optimal weights as a function of hyperparameters. We show that our method converges to locally optimal weights and hyperparameters for sufficiently large hypernets. 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 major bottleneck in designing predictive models. Hyperparameter optimization can be seen as a nested optimization: The inner optimization finds model parameters w which minimizes the training loss $\mathcal{L}_{\text{Train}}$ given hyperparameters λ. The outer optimization chooses λ to minimize 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, random search, or Bayesian optimization. Each set of hyperparameters is evaluated by re-initializing weights and training the model to completion. This is wasteful, since it trains the model from scratch each time, even if the hyperparameters change a small amount. Furthermore, gradient-free optimization scales poorly beyond 10 or 20 dimensions.

How can we avoid re-training from scratch each time? We usually estimate the parameters with stochastic optimization, but the true optimal parameters are a deterministic function of the hyperparameters λ:

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

We propose to *learn this function*. Specifically, we train a neural network whose inputs are the hyperparameters, and whose outputs are an approximately optimal set of weights given the hyperparameters.

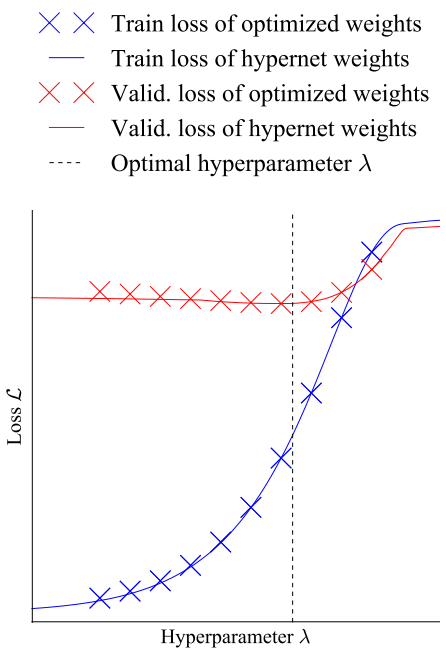

Figure 1: Training and validation loss of a neural net, estimated by cross-validation (crosses) or by a hypernet (lines), which outputs $7,850$-dimensional network weights. The training and validation loss can be cheaply evaluated at any hyperparameter value using a hypernet. Standard cross-validation requires training from scratch each time.

This approach has two major benefits: First, we can train the hypernet to convergence using stochastic gradient descent, denoted SGD, without ever training any particular model to completion. Second, differentiating through the hypernet allows us to optimize hyperparameters with gradient-based stochastic optimization.

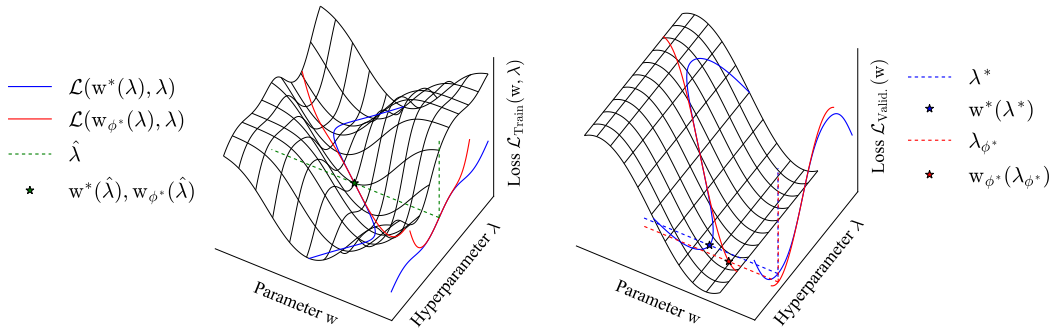

Figure 2: A visualization of exact (blue) and approximate (red) best weights as a function of given hyperparameters. *Left:* The training loss surface. *Right:* The validation loss surface. The approximately optimal weights w_{ϕ^*} are output by a linear model fit at $\hat{\lambda}$. The true optimal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

2 Training a network to output optimal weights

How can we train a neural network to output approximately optimal weights of another neural network? A neural net which outputs the weights of another neural net is called a *hypernet* (Ha et al., 2016). The basic idea is that at each iteration, we ask a hypernet to output a set of weights given the current hyperparameters: $\mathrm{w} = \mathrm{w}_\phi(\lambda)$. Instead of updating weights w using the loss gradient $\partial\mathcal{L}(\mathrm{w})/\partial\mathrm{w}$, we update the hypernet weights ϕ using the chain rule: $\frac{\partial\mathcal{L}(\mathrm{w}_\phi)}{\partial\mathrm{w}_\phi}\frac{\partial\mathrm{w}_\phi}{\partial\phi}$. We call this method *hyper-training* and contrast it with standard training methods in Figure 3.

We call the function $\mathrm{w}^*(\lambda)$ that outputs optimal weights for a given set of hyperparameters a *best-response function* (Fudenberg & Levine, 1998). At convergence, we want our hypernet $\mathrm{w}_\phi(\lambda)$ to closely match the best-response function.

2.1 Advantages of hypernet-based optimization

We can compare the hyper-training approach to other model-based hyperparameter schemes, such as Bayesian optimization. Bayesian optimization (Snoek et al., 2012) builds a model of the validation loss as a function of hyperparameters, usually using a Gaussian process (Rasmussen & Williams, 2006) to track uncertainty. This approach has several disadvantages compares 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 fully optimize any one model.

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

What sort of parameters can be optimized by our approach? Hyperparameters typically fall into two broad categories: 1) Optimization hyperparameters such as learning rates and initialization schemes, and 2) Regularization or model architecture parameters. Hyper-training *does not have inner optimization hyperparameters* because there is no inner training loop. Of course, we must still choose optimization parameters for the fused optimization loop, but this is the also case for any model-based hyperparameter optimization method.

2.2 Limitations of hypernet-based optimization

Hyper-training can handle discrete hyperparameters but does not offer any special advantage for optimizing over discrete hyperparameters. Also, our approach only proposes making local changes to

Algorithm 1: Standard cross-validation with stochastic optimization

for $i = 1, \ldots, T_{\text{outer}}$
  initialize w
  $\lambda = \text{hyperopt}\left(\ldots, \lambda^{(i)}, \mathcal{L}_{\text{Valid.}}\left(\mathrm{w}^{(i)}\right)\right)$
  for T_{inner} steps
    $\mathbf{x} \sim$ Training data
    $\mathrm{w} = \mathrm{w} - \alpha \nabla_{\mathrm{w}} \mathcal{L}_{\text{Train}}(\mathbf{x}, \mathrm{w}, \lambda)$
  $\lambda^i, \mathrm{w}^i = \lambda, \mathrm{w}$
for $i = 1, \ldots, T_{\text{outer}}$
  if $\mathcal{L}_{\text{Valid.}}\left(\mathrm{w}^{(i)}\right) < \mathcal{L}_{\text{Valid.}}(\mathrm{w})$ **then**
    $\hat{\lambda}, \mathrm{w} = \lambda^i, \mathrm{w}^i$
return $\hat{\lambda}, \mathrm{w}$

Algorithm 2: Stochastic optimization of hypernet, then hyperparameters

initialize ϕ
initialize $\hat{\lambda}$
for T_{hypernet} steps
  $\mathbf{x} \sim$ Training data, $\lambda \sim p(\lambda)$
  $\phi = \phi - \alpha \nabla_{\phi} \mathcal{L}_{\text{Train}}(\mathbf{x}, \mathrm{w}_{\phi}(\hat{\lambda}), \hat{\lambda})$

for $T_{\text{hyperparameter}}$ steps
  $\mathbf{x} \sim$ Validation data
  $\hat{\lambda} = \hat{\lambda} - \beta \nabla_{\hat{\lambda}} \mathcal{L}_{\text{Valid.}}(\mathbf{x}, \mathrm{w}_{\phi}(\hat{\lambda}))$
return $\hat{\lambda}, \mathrm{w}_{\phi}(\hat{\lambda})$

Figure 3: A comparison of standard hyperparameter optimization, and our first algorithm. Instead of updating weights w using the loss gradient $\partial\mathcal{L}(\mathrm{w})/\partial\mathrm{w}$, we update hypernet weights ϕ using the chain rule: $\frac{\partial \mathcal{L}(\mathrm{w}_{\phi})}{\partial \mathrm{w}_{\phi}} \frac{\partial \mathrm{w}_{\phi}}{\partial \phi}$. Instead of returning the best hyperparameters from a fixed set, our method uses gradient-based hyperparameter optimization.

the hyperparameters, and does not do uncertainty-based exploration. Uncertainty could conceivably be incorporated into the hypernet, but we leave this for future work. Finally, it is not obvious how to choose the distribution over hyperparameters $p(\lambda)$. We approach this problem in section 2.4.

An obvious difficulty of this approach is that training a hypernet typically requires training several times as many parameters as training a single model. For example, training a fully-connected hypernet with a single hidden layer of H units to output D parameters requires training at least $D \times H$ hypernet parameters. Again, in section 2.4 we propose an algorithm that requires training only a linear model mapping hyperparameters to model weights.

2.3 Asymptotic Convergence Properties

Algorithm 2 trains a hypernet using stochastic gradient descent, 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 hypernet, the choice of $p(\lambda)$ does not matter as long as it has sufficient support.

Theorem 2.1. *Sufficiently powerful hypernets can represent any continuous best-response function.*

$$\text{There exists } \phi^*, \text{ such that for all } \lambda \in \text{support}(p(\lambda)),$$
$$\underset{\text{Train}}{\mathcal{L}}\left(\mathrm{w}_{\phi^*}(\lambda), \lambda\right) = \min_{\mathrm{w}} \underset{\text{Train}}{\mathcal{L}}(\mathrm{w}, \lambda)$$
$$\text{and } \phi^* = \underset{\phi}{\operatorname{argmin}} \underset{p(\lambda')}{\mathbb{E}}\left[\underset{\text{Train}}{\mathcal{L}}\left(\mathrm{w}_{\phi}(\lambda'), \lambda'\right)\right]$$

Proof. If w_{ϕ} is a universal approximator (Hornik, 1991) and the best-response is continuous in λ, then there exists optimal hypernet 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 hypernets can learn continuous best-responses.

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, $\min_{\phi} \mathbb{E}_{p(\lambda)}[\mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda)] \geq \mathbb{E}_{p(\lambda)}[\min_{\phi} \mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda)]$. Thus, $\phi^* = \operatorname{argmin}_{\phi} \mathbb{E}_{p(\lambda)}[\mathcal{L}_{\text{Train}}(\mathrm{w}_{\phi}(\lambda), \lambda)]$. In other words, if the hypernet learns the best-response it will simultaneously minimize the loss for every point in the $\text{support}(p(\lambda))$. □

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.

Algorithm 2: Stochastic optimization of hypernet, then hyperparameters

1: initialize $\phi, \hat{\lambda}$
2: **for** T_{hypernet} steps
3: $\quad \mathbf{x} \sim$ Training data, $\lambda \sim p(\lambda)$
4: $\quad \phi = \phi - \alpha \nabla_\phi \mathcal{L}_{\text{Train}}(\mathbf{x}, \mathrm{w}_\phi(\hat{\lambda}), \hat{\lambda})$
5: **for** $T_{\text{hyperparameter}}$ steps
6: $\quad \mathbf{x} \sim$ Validation data
7: $\quad \hat{\lambda} = \hat{\lambda} - \beta \nabla_{\hat{\lambda}} \mathcal{L}_{\text{Valid.}}(\mathbf{x}, \mathrm{w}_\phi(\hat{\lambda}))$
8: **return** $\hat{\lambda}, \mathrm{w}_\phi(\hat{\lambda})$

Algorithm 3: Stochastic optimization of hypernet and hyperparameters jointly

1: initialize $\phi, \hat{\lambda}$
2: **for** T_{joint} steps
3: $\quad \mathbf{x} \sim$ Training data, $\lambda \sim p(\lambda|\hat{\lambda})$
4: $\quad \phi = \phi - \alpha \nabla_\phi \mathcal{L}_{\text{Train}}(\mathbf{x}, \mathrm{w}_\phi(\hat{\lambda}), \hat{\lambda})$
5:
6: $\quad \mathbf{x} \sim$ Validation data
7: $\quad \hat{\lambda} = \hat{\lambda} - \beta \nabla_{\hat{\lambda}} \mathcal{L}_{\text{Valid.}}(\mathbf{x}, \mathrm{w}_\phi(\hat{\lambda}))$
8: **return** $\hat{\lambda}, \mathrm{w}_\phi(\hat{\lambda})$

Figure 5: A side-by-side comparison of two variants of hyper-training. Algorithm 3 fuses the hypernet training and hyperparameter optimization into a single loop of stochastic gradient descent.

Theorem 2.1 holds for any $p(\lambda)$. However in practice, we have a limited-capacity hypernet, and so should choose a $p(\lambda)$ that puts most of its mass on promising hyperparameter values. This motivates the joint optimization of ϕ and $p(\lambda)$. Concretely, we can introduce a "current" hyperparameter $\hat{\lambda}$ and define a conditional hyperparameter distribution $p(\lambda|\hat{\lambda})$ which places its mass near $\hat{\lambda}$. This allows us to use a limited-capacity hypernet, at the cost of having to re-train the hypernet each time we update $\hat{\lambda}$.

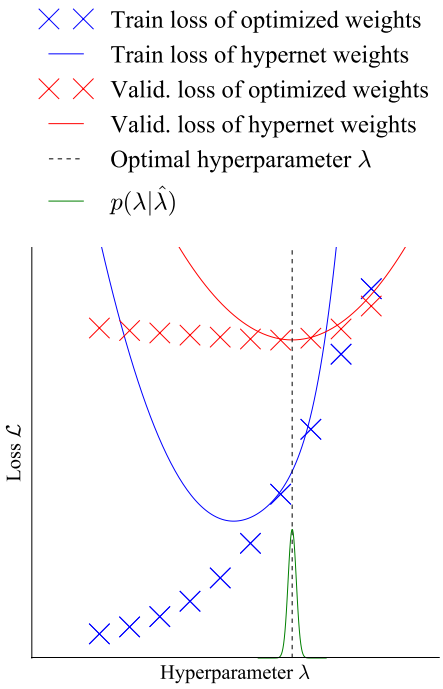

Figure 4: The training and validation losses of a neural network, estimated by cross-validation (crosses) or by a linear hypernet (lines). The limited capacity of the linear hypernet makes the approximation accurate only where hyperparameter distribution put mass.

In practice, there are no guarantees about the network being a universal approximator, or the finite-time convergence of optimization. The optimal hypernet 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 training parameters and hyperparameters

Because in practice we use a limited-capacity hypernet, it may not be possible to learn a best-response for all hyperparameters. Thus, we propose Algorithm 3, which only tries to learn 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 hypernet that can output weights for any hyperparameter then optimizing the hyperparameters, Algorithm 3 only samples hyperparameters near the current best guess. This means that the hypernet only has to be trained well enough to estimate good parameters for a small set of hyperparameters. The locally-trained hypernet can then be used to provide gradients to update the hyperparameters based on validation set performance.

How simple can we make the hypernet, and still obtain useful gradients to optimize hyperparameters? Consider the case where the hypernet is a linear function of the hyperparameters. It learns a tangent hyperplane to a best-response function if the conditional hyperparameter distribution is $p(\lambda|\hat{\lambda}) = \mathcal{N}(\hat{\lambda}, \sigma\mathbb{1})$ for some small σ. This hypernet only needs to make small adjustments at each step if the hyperparameter updates are sufficiently small. We can further

restrict the capacity of a linear hypernet by factorizing its weights, effectively adding a bottleneck layer with a linear activation with a small number of hidden units.

3 Related Work

Our work is very 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 evaluating the performance of a wide variety of discrete model architectures, while we focus on efficiently exploring continuous spaces of models.

Model-free approaches Model-free approaches only use trial-and-error to explore hyperparameter space.Simple model-free approaches applied to hyperparameter optimization include grid search and random search (Bergstra & Bengio, 2012). Model-free reinforcement learning approaches have also been applied to this problem (Huys et al., 2015). Hyperband (Li et al., 2016) combines bandit approaches with modeling the learning procedure.

Model-based approaches Model-based approaches attempt to build a surrogate function, which often facilitates gradient-based optimization or active learning. A common example is Bayesian optimization (Snoek et al., 2012). Freeze-thaw Bayesian optimization (Swersky et al., 2014) even can condition on partially-optimized model performance.

Differentiation-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 a 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. Feng & Simon (2017) establish conditions where the validation loss of best-responding weights is 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 with respect to parameters with the identity matrix. In contrast, the convergence of our method depends on having a suitably powerful hypernet.

Game theory Best-response functions are extensively studied in as a solution concept in discrete and continuous multi-agent games (Fudenberg & Levine, 1998). Games where learning a best-response can be applied include adversarial training (Goodfellow et al., 2014), or Stackelberg competitions (Brückner & Scheffer, 2011).

4 Experiments

In our experiments, we examine the standard setting of stochastic gradient-based optimization of neural networks, with a weight regularization penalty. In this case, the training and validation losses can be written as:

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

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

In all experiments, algorithms 2 or 3 are used to optimize weights of a linear regression 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 have a ReLU activation (Nair & Hinton, 2010)

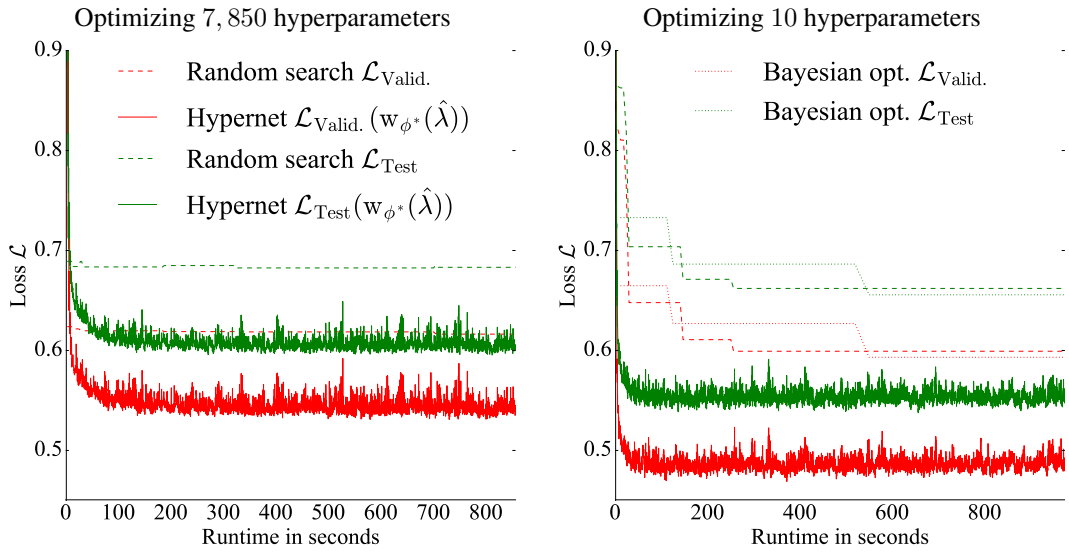

Figure 6: Validation and test losses during hyperparameter optimization. *Left:* A separate L_2 weight decay is applied to each weight in the model, resulting in $7,850$ hyperparameters. *Right:* A separate L_2 weight decay is applied to the weights each digit class, resulting in 10 hyperparameters. Hypernetwork-based optimization converges much more quickly than random search or Bayesian optimization. We also observe significant overfitting on the validation set for all methods.

unless otherwise specified. Autograd (Maclaurin et al., 2015b) was used to compute all derivatives. All experiments were run on a 2012 MacBook pro.

4.1 Learning a global best-response

Our first experiment, shown in figure 1, demonstrates learning a global approximation to a best-response function using algorithm 2. In order to make visualization of the regularization loss easier, we use only 10 training data points to exacerbate overfitting. We compare the performance of weights output by the hypernet to those trained by standard cross-validation (Algorithm 1). Thus, network weights were randomly initialized for each hyperparameter setting, 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 hypernet has 50 hidden units which results in $400,450$ parameters of the hypernetwork. Each minibatch sampled 10 pairs of hyperparameters and the entire training data. Adam was used for training the hypernet, with a step size of 0.0001.

The minimum of the best-response in Figure 1 is close to the true minimum of the validation loss. This experiment shows that on small problems, a hypernet can satisfactorily approximate a global best-response function.

4.2 Learning a local best-response

Figure 4 shows the same experiment, but using the fused updates of Algorithm 3. The conditional hyperparameter distribution is given by $p(\lambda|\hat{\lambda}) = \mathcal{N}(\hat{\lambda}, 0.00001)$. The hypernet is a linear model, with only $15,700$ weights. Each iteration samples 2 pairs of hyperparameters and the entire training data. We used SGD to train the hypernet, with a step size of 0.0001 for 10 iterations, alternated with 1 iteration of SGD on the hyperparameter with a step size of 0.1.

Again, the minimum of the best-response at the end of training is the true optimum on 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 OPTIMIZING 10 HYPERPARAMETERS

Next, we optimized a model with 10 hyperparameters, in which a separate L_2 weight decay is applied the weights for each digit class in a logistic regression model. The standard $50,000$ training data points and a mini-batch size of 100 for the validation and training sets are used. The conditional hyperparameter distribution is the same the prior experiment. A linear hypernet is used, resulting in $86,350$ hyper-weights. Each iteration samples 10 pairs of hyperparameters and a mini-batch from the training data. Adam is used for training the hypernet, with a step size of 0.0001 for 10 iterations, alternated with 1 iteration of Adam on the hyperparameter with a step size of 0.0001. Algorithm 3 is compared against random search and a standard Bayesian optimization implementation from `sklearn`.

Figure 6, 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.

4.4 OPTIMIZING $7,850$ HYPERPARAMETERS

We then optimized a model with $7,850$ hyperparameters, in which a separate L_2 weight decay is applied to each weight in a logistic regression model. If we did not factorize the weights of this linear model, it would have $61,630,350$ weights, so we select 10 hidden units to constrict the total number of weights. The factorized linear hypernet has 10 hidden units with linear activations which gives $164,860$ weights.

Figure 6, left, again shows that Algorithm 3 performs better than random optimization. Standard Bayesian optimization cannot be scaled to this many hyperparameters. This experiment shows Algorithm 3 can effectively optimize thousands of hyperparameters.

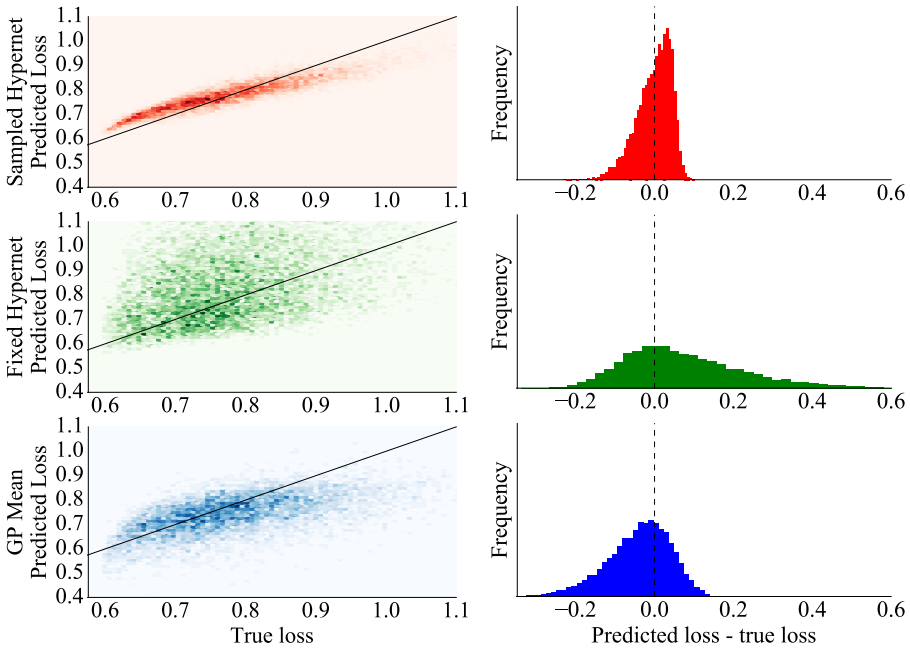

Figure 7: A comparison of a hypernet trained with stochastically sampled hyperparameters, a hypernet trained with a fixed set of hyperparameters, and a Gaussian Process fit on a fixed set of hyperparameters and optimized losses. *Left:* The distribution of predicted and true losses. The diagonal black line is where predicted loss equals true loss. *Right:* The distribution of differences between predicted and true losses. The Gaussian process often under-predicts the true loss, while the hypernet trained on the same data tends to over-predict the true loss.

Method	Evaluations of Validation Loss 10	25	100	250	1000
Gaussian process	0.90	0.67	0.60	0.60	0.62
Hypernet trained on same evaluations	0.65	**0.60**	**0.59**	**0.59**	**0.59**
Hypernet trained stochastically for equivalent time	**0.60**	0.61	**0.59**	**0.59**	**0.59**

Table 1: Actual validation loss at the best predicted hyperparameter setting, according to each model.

4.5 Estimating weights versus estimating loss

As mentioned above, our approach differs from Bayesian optimization in that we attempt to learn to predict optimal weights, while Bayesian optimization attempts to directly model the validation loss of optimized weights. In this final experiment, we attempt to untangle the reason for the better performance of our method: Is it because of a better inductive bias, or because our method can see many 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 largest advantage. We also constructed a validation set of $10,215$ (optimized weight, hyperparameter) generated in the same manner. We then fit a Gaussian process (GP) regression model with an RBF kernel on the hyper-training data. We also fit a hypernet same dataset. However, this hypernet was trained to fit optimized training weights, not optimized validation loss. Finally, we optimize a second hypernet using algorithm 2, for the same amount of time as it took to build the hyper-training set. The two hypernets were linear models, and were trained with the same optimizer parameters as the $7,850$-dimensional hyperparameter optimization.

Figure 7 shows the distribution of prediction errors of these three models. We can see that the Gaussian process tends to underestimating loss. The hypernet trained with the same small fixed set of examples tends to overestimating 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 hypernet must provide actual weights to predict the validation loss, poorly-fit regions will overestimate the validation loss. Finally, the hypernet trained with algorithm 2 produces loss errors tightly centered around 0.

Table 1 shows how varying the number of training tuples affects the hyperparameter which minimizes the predicted loss, where fixed input hyper-training uses the same fixed inputs as the Gaussian process. Algorithm 2 consistently identifies hyperparameters with a better true performance than the other two approaches.

Code for all experiments will be made available upon publication.

5 Conclusions

In this paper, we:

- Presented algorithms that efficiently learn a differentiable approximation to a best-response without nested optimization.
- Showed empirically that hypernets can provide a better inductive bias for hyperparameter optimization than Gaussian processes fit directly to the validation loss.
- Gave a theoretical justification that sufficiently large networks will learn the best-response for all hyperparameters it is trained against.

We hope that this initial exploration of stochastic hyperparameter optimization will inspire further refinements, such as hyper-regularization methods, or uncertainty-aware exploration using Bayesian hypernetworks.

REFERENCES

Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio. An actor-critic algorithm for sequence prediction. *arXiv preprint arXiv:1607.07086*, 2016.

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

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

Michael Brückner and Tobias Scheffer. 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.

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

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

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

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

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

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*, pp. 2672–2680, 2014.

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

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

Quentin JM Huys, Anthony Cruickshank, and Peggy Seriès. Reward-based learning, model-based and model-free. In *Encyclopedia of Computational Neuroscience*, pp. 2634–2641. Springer, 2015.

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

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.

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

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

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

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

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

JF Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36(1):48–49, 1950.

Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

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

David Pfau and Oriol Vinyals. Connecting generative adversarial networks and actor-critic methods. *arXiv preprint arXiv:1610.01945*, 2016.

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

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

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