SELF-TUNING NETWORKS: BILEVEL OPTIMIZATION OF HYPERPARAMETERS USING STRUCTURED BEST-RESPONSE FUNCTIONS

Anonymous authors

Paper under double-blind review

ABSTRACT

Hyperparameter optimization can be formulated as a bilevel optimization problem, where the optimal parameters on the training set depend on the hyperparameters. We aim to adapt regularization hyperparameters by fitting compact approximations to the *best-response function*, which maps hyperparameters to optimal weights and biases. We show that the best-response function for a shallow linear network with L_2-regularized Jacobian can be represented by a single network whose hidden units are gated conditionally on the regularizer. Based on this, we propose a gating-based architecture for approximating best-response functions for neural nets more broadly and which can be used as a drop-in replacement for existing deep learning modules. We fit this model online during training using a gradient-based hyperparameter optimization algorithm which alternates between approximating the best-response around the current hyperparameters and optimizing the hyperparameters using the approximate best-response function. Unlike other gradient-based approaches, we do not require differentiating the training loss with respect to the hyperparameters, allowing us to tune discrete hyperparameters, data augmentation hyperparameters, and dropout probabilities. Empirically, our approach outperforms competing hyperparameter optimization methods on large-scale deep learning problems. We call our networks, which update their own hyperparameters online during training, *Self-Tuning Networks*.

1 INTRODUCTION

Regularization hyperparameters such as weight decay, data augmentation, and dropout (Srivastava et al., 2014) are crucial to the generalization of neural networks (NNs), but difficult to tune. Popular approaches to hyperparameter optimization (HO) include grid search, random search (Bergstra & Bengio, 2012), and Bayesian optimization (Snoek et al., 2012). These approaches work well with low-dimensional hyperparameters and ample computational resources. However, they pose HO as a black-box optimization problem, ignoring structure which can be exploited for faster convergence.

We can formulate HO as a bilevel optimization problem. Let $\mathbf{w}$ and $\boldsymbol{\lambda}$ denote parameters and hyperparameters, and let $\mathcal{L}_T$ and $\mathcal{L}_V$ be functions mapping parameters and hyperparameters to training and validation losses. We aim to solve[1] :

$$\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda}} \mathcal{L}_V(\boldsymbol{\lambda}, \mathbf{w}) \text{ subject to } \mathbf{w} = \arg\min_{\mathbf{w}} \mathcal{L}_T(\boldsymbol{\lambda}, \mathbf{w}) \tag{1}$$

Substituting the *best-response function* $\mathbf{w}^*(\boldsymbol{\lambda}) = \arg\min_{\mathbf{w}} \mathcal{L}_T(\boldsymbol{\lambda}, \mathbf{w})$ gives a single-level problem:

$$\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda}} \mathcal{L}_V(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda})) \tag{2}$$

Gradient-based HO methods attempt to solve Problem 1 by approximating the best-response or its Jacobian. This can be done by differentiating through the training procedure (Maclaurin et al., 2015) or by implicitly deriving the best-response Jacobian (Larsen et al., 1996), but these methods struggle with discrete hyperparameters and are computationally expensive. Recently, Lorraine & Duvenaud

[1] The uniqueness of the $\arg\min$ is assumed.

(2018) proposed directly approximating the best-response by a parametric function $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$. They jointly optimized $\boldsymbol{\phi}$ and $\boldsymbol{\lambda}$, first $\boldsymbol{\phi}$ so $\hat{\mathbf{w}}_{\boldsymbol{\phi}} \approx \mathbf{w}^*$ in a neighborhood around the current hyperparameters, then $\boldsymbol{\lambda}$ by using $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ as a proxy for $\mathbf{w}^*$ in Eq. 2:

$$\boldsymbol{\lambda}^* \approx \arg\min_{\boldsymbol{\lambda}} \mathcal{L}_V(\boldsymbol{\lambda}, \hat{\mathbf{w}}_{\phi}(\boldsymbol{\lambda})) \tag{3}$$

This worked for small-scale problems on MNIST using L_2 regularization, but it is difficult to represent and compute $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ when $\mathbf{w}$ is high-dimensional.

We begin by constructing scalable and compact best-response approximations for NNs by analyzing the best-response structure in a Jacobian-regularized two-layer linear network. Here, the best-response scales the network's hidden units by an amount depending on the hyperparameters. Exploiting this structure, we give a memory efficient best-response approximation which scales to deep, nonlinear modern architectures.

Using this approximation gives a gradient-based HO algorithm that is (1) computationally inexpensive, (2) can optimize all regularization hyperparameters — including discrete hyperparameters — and (3) scales to large NNs. We call the resulting networks *Self-Tuning Networks* (STNs) since they update their own hyperparameters online during training. We empirically evaluate the performance of STNs for HO on large-scale deep-learning problems with the PTB and CIFAR-10 datasets. We compare STN performance to other popular HO approaches and find that STNs achieve lower validation loss in less time.

2 Bilevel Optimization

A bilevel optimization problem consists of two sub-problems called the *upper-level* and *lower-level* problems, where the upper-level problem must be solved subject to optimality of the lower-level problem. Minimax problems are an example of bilevel programs where the upper-level objective equals the negative lower-level objective. Bilevel programs were first studied in economics to model leader/follower firm dynamics (Von Stackelberg, 2010) and have since found uses in various fields (see Colson et al. (2007) for an overview). In machine learning, many problems can be formulated as bilevel programs; for example, HO, GAN training (Goodfellow et al., 2014), meta-learning, and neural architecture search (Zoph & Le, 2016).

Even if all objectives and constraints are linear, bilevel problems are strongly NP-hard (Hansen et al., 1992; Vicente et al., 1994). Due to difficulty in obtaining exact solutions, most work has focused on restricted settings, considering linear, quadratic, and convex functions. In contrast, we focus on obtaining local solutions in the nonconvex, differentiable, and unconstrained setting. Let $F, f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ denote the upper- and lower-level objectives and $\boldsymbol{\lambda} \in \mathbb{R}^n$, $\mathbf{w} \in \mathbb{R}^m$ denote the upper- and lower-level variables. We aim to solve:

$$\min_{\boldsymbol{\lambda} \in \mathbb{R}^n} F(\boldsymbol{\lambda}, \mathbf{w}) \tag{4a}$$

$$\text{subject to} \quad \mathbf{w} \in \arg\min_{\mathbf{w} \in \mathbb{R}^m} f(\boldsymbol{\lambda}, \mathbf{w}) \tag{4b}$$

It is desirable to design a gradient-based algorithm for solving Problem 4, since using gradient information provides drastic speed-ups over black-box optimization methods (Nesterov, 2013). The simplest method is simultaneous gradient descent, updating $\boldsymbol{\lambda}$ using $\partial F / \partial \boldsymbol{\lambda}$ and $\mathbf{w}$ using $\partial f / \partial \mathbf{w}$. However, simultaneous gradient descent often gives incorrect solutions since it fails to account for the dependence of $\mathbf{w}$ on $\boldsymbol{\lambda}$. Consider the relatively common situation where F doesn't depend directly on $\boldsymbol{\lambda}$, so that $\partial F / \partial \boldsymbol{\lambda} \equiv \mathbf{0}$ and hence $\boldsymbol{\lambda}$ is never updated.

2.1 Gradient Descent via the Best-Response Function

A more principled approach to solving Problem 4 is to use the *best-response function* (Gibbons, 1992). Assume the lower-level Problem 4b has a unique optimum $\mathbf{w}^*(\boldsymbol{\lambda})$ for each $\boldsymbol{\lambda}$. Substituting the best-response function $\mathbf{w}^*$ converts Problem 4 into a single-level problem:

$$\min_{\boldsymbol{\lambda} \in \mathbb{R}^n} F^*(\boldsymbol{\lambda}) := F(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda})) \tag{5}$$

If $\mathbf{w}^*$ is differentiable and has a unique optimum for Problem 4b for each $\boldsymbol{\lambda}$, we can minimize Eq. 5 using gradient descent on F^* w.r.t. $\boldsymbol{\lambda}$.

This method requires differentiability of $\mathbf{w}^*$ and a unique optimum for Problem 4b for each $\boldsymbol{\lambda}$. In general, these conditions are difficult to verify. We give sufficient conditions for them to hold in a neighborhood of a point $(\boldsymbol{\lambda}_0, \mathbf{w}_0)$ where $\mathbf{w}_0$ solves Problem 4b given $\boldsymbol{\lambda}_0$.

Lemma 1. *(Fiacco & Ishizuka, 1990) Let $\boldsymbol{w}_0$ solve Problem equation 4b for $\boldsymbol{\lambda}_0$. Suppose f is $\mathcal{C}^2$ in a neighborhood of $(\boldsymbol{\lambda}_0, \boldsymbol{w}_0)$ and the Hessian ${}^{\partial^2 f}/_{\partial \boldsymbol{w}^2}(\boldsymbol{\lambda}_0, \boldsymbol{w}_0)$ is positive definite. Then for some neighborhood U of $\boldsymbol{\lambda}_0$, there exists a continuously differentiable function $\boldsymbol{w}^* : U \to \mathbb{R}^m$ such that $\boldsymbol{w}^*(\boldsymbol{\lambda})$ is the unique solution to Problem 4b for each $\boldsymbol{\lambda} \in U$ and $\boldsymbol{w}^*(\boldsymbol{\lambda}_0) = \boldsymbol{w}_0$.*

Proof. See Appendix B.1. □

The gradient of F^* decomposes into two terms, which we term the *direct gradient* and the *response gradient*. The direct gradient captures the direct reliance of the upper-level objective on $\boldsymbol{\lambda}$, while the response gradient captures how the lower-level parameter responds to changes in the upper-level parameter:

$$\frac{\partial F^*}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}_0) = \underbrace{\frac{\partial F}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}_0, \mathbf{w}^*(\boldsymbol{\lambda}_0))}_{\text{Direct gradient}} + \underbrace{\frac{\partial F}{\partial \mathbf{w}}(\boldsymbol{\lambda}_0, \mathbf{w}^*(\boldsymbol{\lambda}_0)) \frac{\partial \mathbf{w}^*}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}_0)}_{\text{Response gradient}} \tag{6}$$

Even if ${}^{\partial F}/_{\partial \boldsymbol{\lambda}} \not\equiv 0$ and simultaneous gradient descent is possible, including the response gradient can stabilize optimization by converting the bilevel problem into a single level one, as noted by Metz et al. (2016) for GAN optimization. Conversion to a single-level problem ensures the gradient vector field is conservative, avoiding pathological issues described by Mescheder et al. (2017).

2.2 Approximating the Best-Response Function

In general, the solution to Problem 4b is a set, but assuming uniqueness of a solution and differentiability of $\mathbf{w}^*$ can yield fruitful algorithms in practice. In fact, gradient-based HO methods can often be interpreted as approximating either the best-response $\mathbf{w}^*$ or its Jacobian ${}^{\partial \mathbf{w}^*}/_{\partial \boldsymbol{\lambda}}$, as detailed in Section 5. However, these approaches can be computationally expensive. When applied to HO, they often struggle with discrete hyperparameters and stochastic hyperparameters like dropout probabilities, since they require differentiating the training loss w.r.t. hyperparameters. Promising approaches to directly approximate $\mathbf{w}^*$ were proposed by Lorraine & Duvenaud (2018), which are detailed below.

1. Global Approximation. The first algorithm proposed by Lorraine & Duvenaud (2018) approximated $\mathbf{w}^*$ as a differentiable function $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ with parameters $\boldsymbol{\phi}$. If $\mathbf{w}$ represents NN weights, then the mapping $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ is a hypernetwork (Schmidhuber, 1992; Ha et al., 2016). If the distribution $p(\boldsymbol{\lambda})$ is fixed then gradient descent w.r.t. $\boldsymbol{\phi}$ minimizes:

$$\mathbb{E}_{\boldsymbol{\lambda} \sim p(\boldsymbol{\lambda})} \left[f(\boldsymbol{\lambda}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda})) \right] \tag{7}$$

If support(p) is broad and $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ is sufficiently flexible, then $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ can be used as a proxy for $\mathbf{w}^*$ in Problem 5, resulting in the following objective $\hat{F}$:

$$\min_{\boldsymbol{\lambda} \in \mathbb{R}^n} \hat{F}(\boldsymbol{\lambda}, \boldsymbol{\phi}) := F(\boldsymbol{\lambda}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda})) \tag{8}$$

2. Local Approximation. In practice, $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ is usually insufficiently flexible to model $\mathbf{w}^*$ on support(p). The second algorithm of Lorraine & Duvenaud (2018) locally approximated $\mathbf{w}^*$ in a neighborhood around the current upper-level parameter $\boldsymbol{\lambda}$. They set $p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$ to a factorized Gaussian noise distribution with a fixed scale parameter $\boldsymbol{\sigma} \in \mathbb{R}^n_+$, and $\boldsymbol{\phi}$ is found by minimizing the objective:

$$\hat{f}(\boldsymbol{\lambda}, \boldsymbol{\phi}, \boldsymbol{\sigma}) = \mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})} \left[f(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon})) \right] \tag{9}$$

Intuitively, the upper-level parameter $\boldsymbol{\lambda}$ is perturbed by a small amount, so the lower-level parameter learns how to respond. An alternating gradient descent scheme is used, where $\boldsymbol{\phi}$ is updated to minimize equation 9 and $\boldsymbol{\lambda}$ is updated to minimize equation 8. This approach worked for problems using L_2 weight decay on MNIST (LeCun et al., 1998). However, it is unclear if the approach works with different regularizers or scales to larger problems. It requires $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$, which is a priori unwieldy for high dimensional $\mathbf{w}$. It is also unclear how to set $\boldsymbol{\sigma}$, which defines the size of the neighborhood on which $\boldsymbol{\phi}$ is trained, or if the approach can be adapted to discrete and stochastic hyperparameters.

3 SELF-TUNING NETWORKS

We first construct a best-response approximation $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ that is memory efficient and scales to large NNs. Then, we describe a method to automatically adjust the scale of the neighborhood $\boldsymbol{\phi}$ is trained on. We formally describe our algorithm and discuss how it easily handles discrete and stochastic hyperparameters. We call the resulting networks, which update their own hyperparameters online during training, self-tuning networks (STNs).

3.1 EXACT BEST-RESPONSE FOR TWO-LAYER LINEAR NETWORKS

When $\mathbf{w}$ represents the weights of a large NN, $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ must be flexible enough to model the best-response, while remaining memory-efficient and tractable. We begin by investigating the structure of the best-response in a simple setting to extract insights which can be applied to more complicated problems. Consider the regression problem with a 2-layer linear network and weights $\mathbf{w} = (\boldsymbol{Q}, \boldsymbol{s}) \in \mathbb{R}^{D\times D} \times \mathbb{R}^D$, which predict targets t from inputs $\boldsymbol{x} \in \mathbb{R}^D$:

$$\boldsymbol{a}(\boldsymbol{x}; \mathbf{w}) = \boldsymbol{Q}\boldsymbol{x}, \quad y(\boldsymbol{x}; \mathbf{w}) = \langle \boldsymbol{s}, \boldsymbol{a}(\boldsymbol{x}; \mathbf{w})\rangle \tag{10}$$

Suppose we use a squared-error loss regularized with an L_2 penalty on the Jacobian $\partial y / \partial \boldsymbol{x}$, where the penalty weight λ is mapped with $\exp$ to lie in $\mathbb{R}$ rather than $\mathbb{R}_+$:

$$\mathcal{L}_T(\lambda, \mathbf{w}) = \sum_{(\boldsymbol{x},t)\in\mathcal{D}} (y(\boldsymbol{x}; \mathbf{w}) - t)^2 + \frac{1}{|\mathcal{D}|}\exp(\lambda) \left\| \frac{\partial y}{\partial \boldsymbol{x}}(\boldsymbol{x}; \mathbf{w}) \right\|^2 \tag{11}$$

Lemma 2. *Let $\boldsymbol{w}_0 = (\boldsymbol{Q}_0, \boldsymbol{s}_0)$, where $\boldsymbol{Q}_0$ is the change-of-basis matrix to the principal components of the data matrix and $\boldsymbol{s}_0$ solves the unregularized version of Problem 11 given $\boldsymbol{Q}_0$. Then there exist $\boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^D$ such that the best-response function[2] $\boldsymbol{w}^*(\lambda) = (\boldsymbol{Q}^*(\lambda), \boldsymbol{s}^*(\lambda))$ is:*

$$\boldsymbol{Q}^*(\lambda) = \sigma(\lambda\boldsymbol{w} + \boldsymbol{b}) \odot_{\text{row}} \boldsymbol{Q}_0, \quad \boldsymbol{s}^*(\lambda) = \boldsymbol{s}_0$$

Here, $\odot_{\text{row}}$ indicates row-wise rescaling.

Proof. See Appendix B.2. □

Observe that $y(\boldsymbol{x}; \mathbf{w}^*(\lambda))$ can be viewed as sigmoidal gating of hidden units:

$$\boldsymbol{a}(\boldsymbol{x}; \mathbf{w}^*(\lambda)) = \boldsymbol{Q}^*(\lambda)\boldsymbol{x} = \sigma(\lambda\boldsymbol{w} + \boldsymbol{b}) \odot_{\text{row}} (\boldsymbol{Q}_0\boldsymbol{x}) = \sigma(\lambda\boldsymbol{w} + \boldsymbol{b}) \odot_{\text{row}} \boldsymbol{a}(\boldsymbol{x}; \mathbf{w}_0) \tag{12}$$

This architecture is shown in Figure 1. Inspired by this example, we use a similar gating of the hidden units to approximate the best-response for deep, nonlinear networks.

3.2 LINEAR BEST-RESPONSE APPROXIMATIONS

We experimented with several gating mechanisms and found gating by a linear transformation of the hyperparameters worked best. Our architecture scales and shifts the rows of weights and biases (equivalently, the network's hidden units) by amounts depending on the hyperparameters. For each layer's weight matrix $\boldsymbol{W} \in \mathbb{R}^{D_{out}\times D_{in}}$ and bias $\boldsymbol{b} \in \mathbb{R}^{D_{out}}$, we approximate the best-response as[3]:

$$\hat{\boldsymbol{W}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda}) = \boldsymbol{W}_{\text{shift}} + (\boldsymbol{U}\boldsymbol{\lambda}) \odot_{\text{row}} \boldsymbol{W}_{\text{scale}}, \quad \hat{\boldsymbol{b}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda}) = \boldsymbol{b}_{\text{shift}} + (\boldsymbol{C}\boldsymbol{\lambda}) \odot_{\text{row}} \boldsymbol{b}_{\text{scale}} \tag{13}$$

This best-response architecture is tractable to compute and memory-efficient: it requires $D_{out}(2D_{in} + n)$ parameters to represent $\hat{W}_{\boldsymbol{\phi}}$ and $D_{out}(2 + n)$ parameters to represent $\hat{\boldsymbol{b}}_{\boldsymbol{\phi}}$. In practice, the approximation can be implemented by simply replacing existing modules in deep learning libraries with "hyper" counterparts which accept an additional vector of hyperparameters[4].

There is no guarantee this architecture can represent the best-response function in general, but intuitively some hidden units should be (de-)emphasized given stronger regularization. NN generalization is closely related to the network's Jacobian norm (Novak et al., 2018), suggesting that the structure of the best-response from Problem 11 may extend to other types of regularization.

[2] This is an abuse of notation since there is not a unique solution to Problem 11 for each λ in general.

[3] We describe modifications for convolutional filters in Appendix C.

[4] We illustrate how this is done for the PyTorch library (Paszke et al., 2017) in Appendix G.

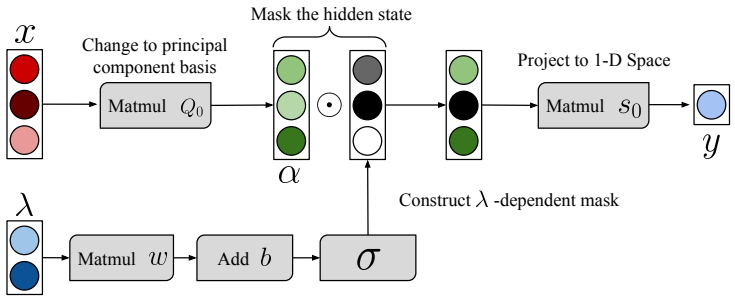


Figure 1: Best-response architecture for an L_2-Jacobian regularized two-layer linear network.

This architecture allows the use of different perturbed hyperparameters for each minibatch element since the activations of each batch element can be scaled and shifted separately, enabling $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ to quickly learn how to respond in a neighborhood around the current hyperparameter values.

To assess how accurately this approximation can model the best-response, we provide theoretical results in the quadratic setting for a slightly more general class of approximations: generic linear transformations. We show that minimizing equation 9 yields the correct optimal-response gradient, hence generic linear approximations are sufficient in this restricted setting.

Theorem 3. *Suppose f is quadratic with $\partial^2 f/\partial \mathbf{w}^2 \succ \mathbf{0}$ and $p(\boldsymbol{\epsilon}|\sigma)$ is Gaussian with mean $\mathbf{0}$ and variance $\sigma^2 \boldsymbol{I}$. Suppose $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ is linear and $\boldsymbol{\phi}^* = \arg\min_{\boldsymbol{\phi}} \hat{f}(\boldsymbol{\lambda}, \boldsymbol{\phi}, \sigma)$. Then we have $\partial \hat{\mathbf{w}}_{\boldsymbol{\phi}}/\partial \boldsymbol{\lambda}(\boldsymbol{\lambda}_0) = \partial \mathbf{w}^*/\partial \boldsymbol{\lambda}(\boldsymbol{\lambda})$.*

Proof. See Appendix B.3. □

3.3 Adapting the Hyperparameter Distribution

The entries of $\boldsymbol{\sigma}$, which control the scale of the hyperparameter distribution $\boldsymbol{\phi}$ is trained on, are crucial to the STN's performance. If the entries are too large, then $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ will not be flexible enough to capture the best-response over the samples. However, the entries must remain large enough to force $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ to capture the local shape around the current hyperparameter values. We illustrate this in Figure 2. As the smoothness of the loss landscape changes during training, it may be beneficial to vary $\boldsymbol{\sigma}$.

To address these issues we propose adjusting $\boldsymbol{\sigma}$ during training based on the sensitivity of the upper-level objective to sampled hyperparameters. We include an entropy term weighted by $\tau \in [0, 1]$ which enlarges entries of $\boldsymbol{\sigma}$. The resulting objective is:

$$\hat{F}_V(\boldsymbol{\lambda}, \boldsymbol{\phi}, \boldsymbol{\sigma}) = \mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})}[F(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon}))] - \tau \mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})] \tag{14}$$

This is similar to a variational inference (VI) objective, where the first term is analogous to the negative-log likelihood, but $\tau \neq 1$. As τ ranges from 0 to 1, our objective interpolates between variational optimization (Staines & Barber, 2012) and VI, as noted by Khan et al. (2018). Similar objectives have been used in the VI literature for better training (Blundell et al., 2015) and representation learning (Higgins et al., 2016).

Minimizing the first term on its own eventually moves all probability mass towards an optimum $\boldsymbol{\lambda}^*$, resulting in $\boldsymbol{\sigma} = \mathbf{0}$. This compels $\boldsymbol{\sigma}$ to balance between shifting towards $\mathbf{0}$ to decrease the first term while remaining sufficiently high to avoid a heavy entropy penalty. When benchmarking our algorithm's performance, we evaluate $\hat{F}$ at the deterministic current hyperparameter $\boldsymbol{\lambda}_0$. This is common practice when using stochastic operations during training, e.g., batch-norm or dropout.

3.4 Training Algorithm

We now describe the complete STN training algorithm and discuss how it can tune hyperparameters which other gradient-based algorithms cannot, such as discrete hyperparameters. We use an unconstrained parametrization $\boldsymbol{\lambda} \in \mathbb{R}^n$ of the hyperparameters. Let r denote the element-wise function which maps $\boldsymbol{\lambda}$ to the appropriate constrained space, which will involve a non-differentiable discretization for discrete hyperparameters.

Let $\mathcal{L}_T$ and $\mathcal{L}_V$ denote training and validation losses which are (possibly stochastic, e.g., if using dropout) functions of the hyperparameters and parameters. Define functions f, F by $f(\boldsymbol{\lambda}, \mathbf{w}) =$

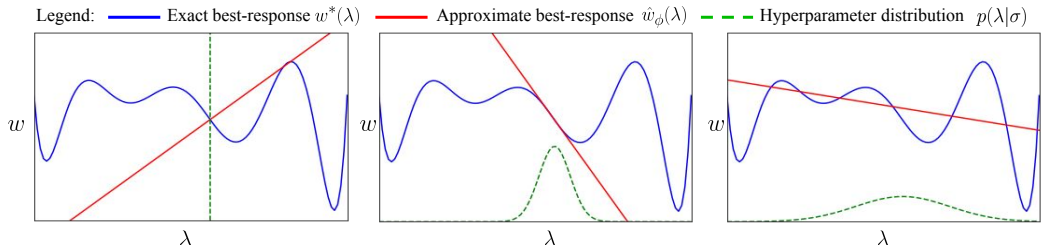


Figure 2: **Left:** If the sampled neighborhood is too small (e.g. a point mass) the approximation learned will only match the exact best-response at the current hyperparameter, with no guarantee its gradient matches that of the best-response. **Middle:** If the sampled neighborhood is not too small or wide, the approximation's gradient will match that of the best-response. **Right:** If the sampled neighborhood is too wide, the approximation will be insufficiently flexible to model the best-response, and again the gradients will not match.

$\mathcal{L}_T(r(\boldsymbol{\lambda}), \mathbf{w})$ and $F(\boldsymbol{\lambda}, \mathbf{w}) = \mathcal{L}_V(r(\boldsymbol{\lambda}), \mathbf{w})$. STNs are trained by a gradient descent scheme which alternates between updating $\boldsymbol{\phi}$ for T_{train} steps to minimize $\hat{f}(\boldsymbol{\lambda}, \boldsymbol{\phi}, \boldsymbol{\sigma})$ and updating $\boldsymbol{\lambda}$ and $\boldsymbol{\sigma}$ for T_{valid} steps to minimize $\hat{F}_V(\boldsymbol{\lambda}, \boldsymbol{\phi}, \boldsymbol{\sigma})$. We give our complete algorithm as Algorithm 1 and show how it can be implemented in code in Appendix G.

The possible non-differentiability of r due to discrete hyperparameters poses no problem. To differentiate $\hat{f}$ w.r.t. $\boldsymbol{\phi}$, we only require $\partial f / \partial \mathbf{w}$ and $\partial \hat{\mathbf{w}}_{\boldsymbol{\phi}} / \partial \boldsymbol{\phi}$. To differentiate $\hat{F}_V$ w.r.t. λ_i corresponding to a discrete hyperparameter, there are two cases we must consider:

Case 1: For most regularization schemes, $\mathcal{L}_V$ does not depend on λ_i directly and hence the only gradient is through $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$. Thus, the reparametrization gradient can be used when differentiating $\hat{F}_V$.

Case 2: If $\mathcal{L}_V$ relies explicitly on λ_i, then we can use the REINFORCE gradient estimator (Williams, 1992) to estimate $\partial \hat{F}_V / \partial \lambda_i$. The number of hidden units is an example of a hyperparameter requiring this approach since it directly affects the validation loss.

4 Experiments

Algorithm 1 STN Training Algorithm

Initialize: Approximation parameters $\boldsymbol{\phi}$, hyperparameters $\boldsymbol{\lambda}$
while not converged **do**
  for $t = 1, \ldots, T_{train}$ **do**
    $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$
    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \alpha_1 \frac{\partial}{\partial \boldsymbol{\phi}} \left[f(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon})) \right]$
  for $t = 1, \ldots, T_{valid}$ **do**
    $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$
    $\hat{F}(\boldsymbol{\lambda}, \sigma) = F(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon})) - \tau \mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})]$
    $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \alpha_2 \frac{\partial}{\partial \boldsymbol{\lambda}} \left[\hat{F}(\boldsymbol{\lambda}, \sigma) \right]$
    $\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma} - \alpha_3 \frac{\partial}{\partial \boldsymbol{\sigma}} \left[\hat{F}(\boldsymbol{\lambda}, \sigma) \right]$

We applied our methods to convolutional networks and LSTMs (Hochreiter & Schmidhuber, 1997), yielding self-tuning CNNs (ST-CNNs) and self-tuning LSTMS (ST-LSTMs). To understand the behavior of STNs, we empirically investigated their behavior in a simple setting of tuning a single hyperparameter. Next, we compared the performance of STNs to commonly used HO methods on the CIFAR-10 and PTB datasets. We did not compare with other gradient-based approaches since they cannot tune discrete hyperparameters or dropout probabilities.

4.1 Hyperparameter Schedules

When tuning a single hyperparameter, STNs do not converge to a single fixed hyperparameter. Instead, STNs discover schedules for adapting the hyperparameters online which outperform using any fixed hyperparameter in training. We examined this in detail on the PTB corpus (Marcus et al., 1993) using an ST-LSTM to tune the output dropout rate applied to hidden units.

The schedule discovered by an ST-LSTM for output dropout, shown in Figure 3, outperforms the best, fixed output dropout rate (0.68) found by a fine-grained grid search, achieving 83.25 vs 85.33

	PTB		CIFAR-10	
Method	**Val Perplexity**	**Test Perplexity**	**Val Loss**	**Test Loss**
Grid Search	97.32	94.58	.1686	.1701
Random Search	84.81	81.46	.1518	.1517
Bayesian Optimization	72.13	69.29	.1366	**.1403**
STN	**70.30**	**67.68**	**.1330**	.1411

Table 2: Final validation and test performance of each method on the PTB word-level language modeling task, and the CIFAR-10 image-classification task.

validation perplexity. We argue this is a consequence of the schedule, and not regularizing effects from sampling hyperparameters or the limited capacity of $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$.

To determine if hyperparameter stochasticity is responsible for improved generalization, we trained a standard LSTM while perturbing its dropout rate around the best value found by grid search. We perturbed using (1) random Gaussian perturbations, and (2) sinusoid perturbations, which implement a cyclic regularization schedule. STNs outperform both perturbation methods.

To determine if the limited capacity of $\hat{\mathbf{w}}_{\boldsymbol{\phi}}$ acts as a regularizer, we trained a standard LSTM from scratch using the schedule for output dropout discovered by the ST-LSTM. Again, it outperforms the best-fixed value, giving evidence that the schedule provides benefit. The final validation and test perplexities of each variant are shown in Table 1.

Next, we show that the schedules discovered by STNs are *greedy*, i.e., the hyperparameters are adapted to be well-suited for the current training phase. As shown empirically in Appendix F, lower regularization is best early in training, while higher regularization is better later. We found the STN schedule implements a curriculum by using a low dropout rate early in training, aiding optimization, and gradually increasing the dropout rate, leading to better generalization. We show the schedules the STN discovers for different initial dropout rates in Figure 3. The same schedule is followed no matter what initial rate is used.

Method	**Val**	**Test**
$p = 0.68$ (Fixed)	85.33	82.82
$p \sim \mathcal{N}(0.68, \sigma = 0.05)$	84.39	81.58
$p = 0.68 + 0.1\sin(k\pi)$	85.05	82.42
$p = 0.78$ (Converged STN)	88.12	84.94
STN (Ours)	**83.25**	**80.54**
Following STN Schedule	83.62	80.58

Table 1: Comparing fixed and time-varying methods for setting the output dropout hyperparameter for an LSTM.

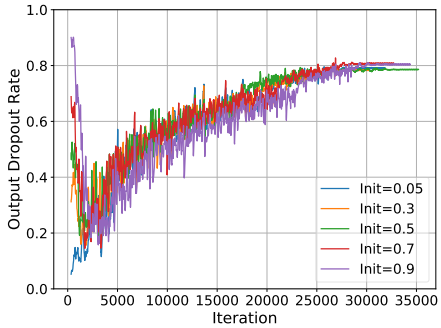
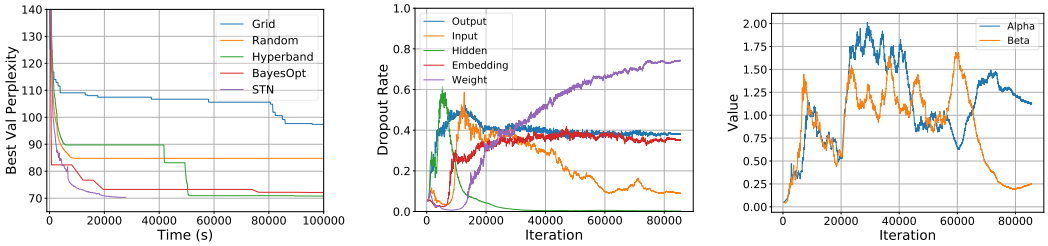


Figure 3: Output dropout schedules found by the ST-LSTM for different initial dropout rates.

4.2 Language Modelling

We evaluated an ST-LSTM on the PTB corpus (Marcus et al., 1993), which is widely used as a benchmark for RNN regularization due to its small size (Gal & Ghahramani, 2016; Merity et al., 2018; Wen et al., 2018). We used a 2-layer LSTM with 650 hidden units per layer and 650-dimensional word embeddings. We tuned 7 hyperparameters: variational dropout rates for the input, hidden state, and output; embedding dropout (i.e., setting rows of the embedding matrix to $\mathbf{0}$); DropConnect on the hidden-to-hidden weight matrix; and coefficients α and β that control the strength of activation regularization and temporal activation regularization, respectively. For LSTM tuning, we obtained the best results when using a fixed perturbation scale of 1 for the hyperparameters. Additional details about the experimental setup and the role of these hyperparameters can be found in Appendix D.

We compare STNs to grid search, random search, Bayesian optimization, and Hyperband.[5] Figure 4a shows the best validation perplexity achieved by each method over time. STNs outperform other methods, achieving lower validation perplexity more quickly. We show the schedules the STN finds for each hyperparameter in Figures 4b and 4c. Hyperband achieves validation perplexity 70.78, while STNs achieve 70.30.



(a) Time comparison (b) STN schedule for dropout (c) STN schedule for α and β

Figure 4: A comparison of the best validation perplexity achieved on PTB over time, by grid search, random search, Bayesian optimization, and STNs. STNs achieve better (lower) validation perplexity in less time. We also show the hyperparameter schedule found by the STN.

4.3 Image Classification

We evaluated ST-CNNs on the CIFAR-10 (Krizhevsky, 2009) dataset, where it is easy to overfit with high-capacity networks. We used the AlexNet architecture (Krizhevsky et al., 2012), optimizing: (1) discrete architectural hyperparameters of the number of filters in each convolutional layer; (2) continuous hyperparameters of per-layer activation dropout and input dropout, (3) discrete data augmentation hyperparameters of the length and number of cutout holes (DeVries & Taylor, 2017), and (4) continuous data augmentation hyperparameters controlling the amount of noise to apply to the hue, saturation, brightness, and contrast of an image.

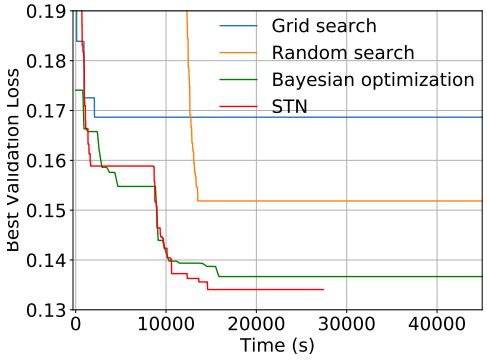


Figure 5: A comparison of the best validation loss achieved on CIFAR-10 over time, by grid search, random search, Bayesian optimization, and STNs. STNs outperform other methods for many computational budgets by finding the best validation loss.

We compared STNs to grid search, random search, and Bayesian optimization. Figure 5 shows the lowest validation loss achieved by each method over time. Details of the optimization procedures are in Appendix E. The baselines methods decay the learning rate at most 3 times by a factor of 10 each time an epoch is completed without the validation loss improving, after which we halt optimization early. Again, STNs find better hyperparameter configurations in less time than other methods. The hyperparameter schedule found by the STN is shown in Figure 6.

5 Related Work

Bilevel Optimization. Colson et al. (2007) is an overview of bilevel problems and Bard (2013) is a comprehensive textbook. When the objectives/constraints are restricted to be linear, quadratic or convex, a common approach replaces the lower-level problem with its KKT conditions added as constraints for the upper-level problem (Hansen et al., 1992; Vicente et al., 1994). In the unrestricted setting, our work loosely resembles trust-region methods (Colson et al., 2005), which repeatedly approximate the problem locally using a simpler bilevel program. In closely related work, Sinha et al. (2013) used evolutionary techniques to estimate the best-response function iteratively.

Hypernetworks. First considered by Schmidhuber (1993; 1992), hypernetworks are functions mapping to the weights of an NN. Predicting weights in CNNs has been developed in various forms

[5]For grid search, random search, and Hyperband we used the Ray Tune libraries (https://github.com/ray-project/ray/tree/master/python/ray/tune). For Bayesian optimization, we used Spearmint (https://github.com/HIPS/Spearmint).

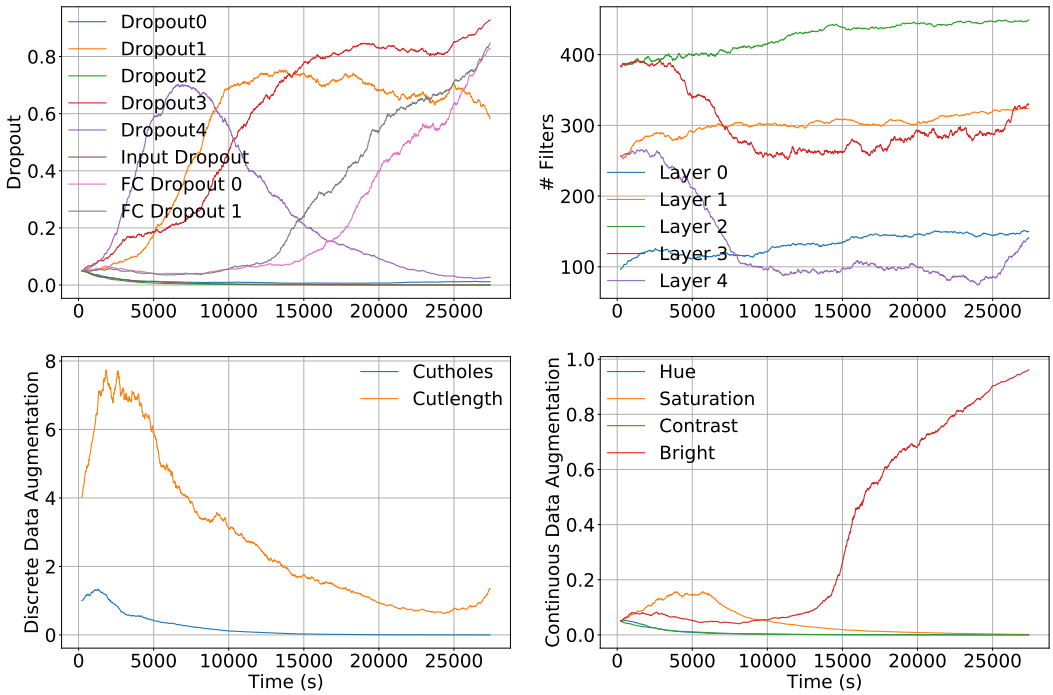


Figure 6: The hyperparameter schedule prescribed by the STN while training for image classification. The dropouts are indexed by the convolutional layer they are applied to. FC dropout is for the fully-connected layers.

(Denil et al., 2013; Yang et al., 2015). Ha et al. (2016) used hypernetworks to generate weights for modern CNNs and RNNs. Brock et al. (2017) used hypernetworks to globally approximate a best-response for architecture search. Because the architecture is not optimized during training, they require a large hypernetwork, unlike ours which locally approximates the best-response.

Gradient-Based HO. There are two main approaches. The first approach approximates $\mathbf{w}^*(\boldsymbol{\lambda}_0)$ using $\mathbf{w}_T(\boldsymbol{\lambda}_0, \mathbf{w}_0)$, the value of $\mathbf{w}$ after T steps of gradient descent on f w.r.t. $\mathbf{w}$ starting at $(\boldsymbol{\lambda}_0, \mathbf{w}_0)$. The descent steps are differentiated through to approximate $\partial \mathbf{w}^* / \partial \boldsymbol{\lambda}(\boldsymbol{\lambda}_0) \approx \partial \mathbf{w}_T / \partial \boldsymbol{\lambda}(\boldsymbol{\lambda}_0, \mathbf{w}_0)$. This approach was proposed by Domke (2012) and used by Maclaurin et al. (2015), Luketina et al. (2016) and Franceschi et al. (2018). The second approach uses the Implicit Function Theorem to derive $\partial \mathbf{w}^* / \partial \boldsymbol{\lambda}(\boldsymbol{\lambda}_0)$ under certain conditions. This was first used for HO in NNs (Larsen et al., 1996) and developed further by Pedregosa (2016). Similar approaches have been used for HO in log-linear models (Foo et al., 2008), kernel selection (Chapelle et al., 2002; Seeger, 2007), and image reconstruction (Kunisch & Pock, 2013; Calatroni et al., 2016). Both approaches struggle with certain hyperparameters, since they differentiate gradient descent or the training loss w.r.t. the hyperparameters. In addition, differentiating gradient descent becomes prohibitively expensive as the number of descent steps increases, while implicitly deriving $\partial \mathbf{w}^* / \partial \boldsymbol{\lambda}$ requires using Hessian-vector products with conjugate gradient solvers to avoid directly computing the Hessian.

Model-Based HO. A common model-based approach is Bayesian optimization, which models $p(r|\boldsymbol{\lambda}, \mathcal{D})$, the conditional probability of the performance on some metric r given hyperparameters $\boldsymbol{\lambda}$ and a dataset $\mathcal{D} = \{(\boldsymbol{\lambda}_i, r_i)\}$. We can model $p(r|\boldsymbol{h}, \mathcal{D})$ with various methods (Hutter et al., 2011; Bergstra et al., 2011; Snoek et al., 2012; 2015). $\mathcal{D}$ is constructed iteratively, where the next $\boldsymbol{\lambda}$ to train on is chosen by maximizing an acquisition function $C(\boldsymbol{\lambda}; p(r|\boldsymbol{\lambda}, \mathcal{D}))$ which balances exploration and exploitation. Training each model to completion can be avoided if assumptions are made on learning curve behavior (Swersky et al., 2014; Klein et al., 2016). These approaches require building inductive biases into $p(r|\boldsymbol{\lambda}, \mathcal{D})$ which may not hold in practice, do not take advantage of the NN structure when used for HO, and do not scale well with the number of hyperparameters. However, these approaches have consistency guarantees in the limit, unlike ours.

Model-Free HO. Model-free approaches include grid-search and random search. Bergstra & Bengio (2012) advocated using random search over grid search. Successive Halving (Jamieson & Talwalkar, 2016) and Hyperband (Li et al., 2016) extend random search by adaptively allocating resources to promising configurations using multi-armed bandit techniques. These methods ignore structure in the problem, unlike ours which uses of rich gradient information. However, it is trivial to parallelize model-free methods over computing resources.

Hyperparameter Scheduling. Population Based Training (PBT) (Jaderberg et al., 2017) considers schedules for hyperparameters. In PBT, a population of NNs are trained in parallel. The hyperparameters and weights of under-performing NNs are periodically replaced by better-performing counterparts, which are randomly perturbed before copying. Thus, a single model can experience different hyperparameters over the course of training, implementing a schedule. STNs replace the population of NNs by a single best-response approximation and use gradients to tune hyperparameters during a single training run.

6 Conclusion

We proposed Self-Tuning Networks (STNs), which efficiently approximate the best-response of parameters to hyperparameters by scaling and shifting their hidden units. This allowed us to use gradient-based optimization to tune various regularization hyperparameters, including discrete hyperparameters. We validated the approach on large-scale problems by showing STNs achieve better generalization performance than competing approaches in less time. We believe STNs offer a compelling path towards large-scale, automated hyperparameter tuning for neural networks.

References

Eugene L Allgower and Kurt Georg. *Numerical continuation methods: An introduction*, volume 13. Springer Science & Business Media, 2012.

Jonathan F Bard. *Practical bilevel optimization: Algorithms and applications*, volume 30. Springer Science & Business Media, 2013.

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

James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pp. 2546–2554. 2011.

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

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

Luca Calatroni, Chung Cao, Juan Carlos De Los Reyes, Carola-Bibiane Schönlieb, and Tuomo Valkonen. Bilevel approaches for learning of variational imaging models. 2016.

Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for Support Vector Machines. *Machine Learning*, 46(1-3):131–159, 2002.

Benoît Colson, Patrice Marcotte, and Gilles Savard. A trust-region method for nonlinear bilevel programming: Algorithm and computational experience. *Computational Optimization and Applications*, 30(3):211–227, 2005.

Benoît Colson, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Annals of Operations Research*, 153(1):235–256, 2007.

Misha Denil, Babak Shakibi, Laurent Dinh, Nando De Freitas, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pp. 2148–2156, 2013.

Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

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

John Duchi. Properties of the trace and matrix derivatives. 2007.

Anthony V Fiacco and Yo Ishizuka. Sensitivity and stability analysis for nonlinear programming. *Annals of Operations Research*, 27(1):215–235, 1990.

Chuan-sheng Foo, Chuong B Do, and Andrew Y Ng. Efficient multiple hyperparameter learning for log-linear models. In *Advances in Neural Information Processing Systems*, pp. 377–384, 2008.

Luca Franceschi, Paolo Frasconi, Saverio Salzo, and Massimilano Pontil. Bilevel programming for hyperparameter optimization and meta-learning. *arXiv preprint arXiv:1806.04910*, 2018.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pp. 1019–1027, 2016.

Robert Gibbons. *A primer in game theory*. Harvester Wheatsheaf, 1992.

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.

Pierre Hansen, Brigitte Jaumard, and Gilles Savard. New branch-and-bound rules for linear bilevel programming. *SIAM Journal on Scientific and Statistical Computing*, 13(5):1194–1217, 1992.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. Beta-VAE: Learning basic visual concepts with a constrained variational framework. 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pp. 507–523. Springer, 2011.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population-based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pp. 240–248, 2016.

Mohammad Emtiyaz Khan, Didrik Nielsen, Voot Tangkaratt, Wu Lin, Yarin Gal, and Akash Srivastava. Fast and scalable Bayesian deep learning by weight-perturbation in Adam. *arXiv preprint arXiv:1806.04854*, 2018.

Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with Bayesian neural networks. 2016.

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

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

Karl Kunisch and Thomas Pock. A bilevel optimization approach for parameter learning in variational models. *SIAM Journal on Imaging Sciences*, 6(2):938–983, 2013.

Jan Larsen, Lars Kai Hansen, Claus Svarer, and M Ohlsson. Design and regularization of neural networks: The optimal use of a validation set. In *Neural Networks for Signal Processing [1996] VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop*, pp. 62–71. IEEE, 1996.

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: Bandit-based configuration evaluation for hyperparameter optimization. 2016.

Jonathan Lorraine and David Duvenaud. Stochastic hyperparameter optimization through hypernetworks. *arXiv preprint arXiv:1802.09419*, 2018.

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, 2015.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. *International Conference on Learning Representations*, 2018.

Lars Mescheder, Sebastian Nowozin, and Andreas Geiger. The numerics of GANs. In *Advances in Neural Information Processing Systems*, pp. 1825–1835, 2017.

Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks. *arXiv preprint arXiv:1611.02163*, 2016.

Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

Roman Novak, Yasaman Bahri, Daniel A Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Sensitivity and generalization in neural networks: An empirical study. *arXiv preprint arXiv:1802.08760*, 2018.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

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

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

Jürgen Schmidhuber. A 'self-referential' weight matrix. In *ICANN'93*, pp. 446–450. Springer, 1993.

Matthias Seeger. Cross-validation optimization for large scale hierarchical classification kernel methods. In *Advances in Neural Information Processing Systems*, pp. 1233–1240, 2007.

Ankur Sinha, Pekka Malo, and Kalyanmoy Deb. Efficient evolutionary algorithm for single-objective bilevel optimization. *arXiv preprint arXiv:1303.3901*, 2013.

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.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, M Prabhat, and Ryan Adams. Scalable Bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, pp. 2171–2180, 2015.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Joe Staines and David Barber. Variational optimization. *arXiv preprint arXiv:1212.4507*, 2012.

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

Luis Vicente, Gilles Savard, and Joaquim Júdice. Descent approaches for quadratic bilevel programming. *Journal of Optimization Theory and Applications*, 81(2):379–399, 1994.

Heinrich Von Stackelberg. *Market Structure and Equilibrium*. Springer Science & Business Media, 2010.

Yeming Wen, Paul Vicol, Jimmy Ba, Dustin Tran, and Roger Grosse. Flipout: Efficient pseudo-independent weight perturbations on mini-batches. *International Conference on Learning Representations*, 2018.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1476–1483, 2015.

Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural Turing machines-revised. *arXiv preprint arXiv:1505.00521*, 2015.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

A TABLE OF NOTATION

Table 3: Table of Notation

$\boldsymbol{\lambda}, \mathbf{w}$	Hyperparameters and parameters
$\boldsymbol{\lambda}_0, \mathbf{w}_0$	Current, fixed hyperparameters and parameters
n, m	Hyperparameter and elementary parameter dimension
$f(\boldsymbol{\lambda}, \mathbf{w}), F(\boldsymbol{\lambda}, \mathbf{w})$	Lower-level & upper-level objective
r	A function which reparametrizes hyperparameters to lie in the reals
$\mathcal{L}_T(\boldsymbol{\lambda}, \mathbf{w}), \mathcal{L}_V(\boldsymbol{\lambda}, \mathbf{w})$	Training loss & validation loss - $(\mathcal{L}_T(r(\boldsymbol{\lambda}, \mathbf{w})), \mathcal{L}_V(r(\boldsymbol{\lambda}, \mathbf{w}))) = (f(\boldsymbol{\lambda}, \mathbf{w}), F(\boldsymbol{\lambda}, \mathbf{w}))$
$\mathbf{w}^*(\boldsymbol{\lambda})$	Best-response of the parameters to the hyperparameters
$F^*(\boldsymbol{\lambda})$	Single-level objective from best-response, equals $F(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda}))$
$\boldsymbol{\lambda}^*$	Optimal hyperparameters
$\hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda})$	Parametric approximation to the best-response function
$\boldsymbol{\phi}$	Approximate best-response parameters
$\boldsymbol{\sigma}$	Scale of the hyperparameter noise distribution
σ	The sigmoid function
$\boldsymbol{\epsilon}$	Sampled perturbation noise, to be added to hyperparameters
$p(\boldsymbol{\epsilon}\|\boldsymbol{\sigma}), p(\boldsymbol{\lambda}\|\boldsymbol{\sigma})$	The noise distribution and induced hyperparameter distribution
α	A learning rate
T_{train}, T_{valid}	Number of training steps on the training and validation data
$\boldsymbol{x}, t$	An input datapoint and its associated target
$\mathcal{D}$	A data matrix consisting of tuples of inputs and targets
D	The dimensionality of input data
$y(\boldsymbol{x}, \mathbf{w})$	A predicted target for a input data and elementary parameters
$\odot_{\text{row}}$	Row-wise rescaling - *not elementwise multiplication*
$\boldsymbol{Q}, \boldsymbol{s}$	A basis change matrix to the principle components of the data matrix and solution to Problem 11
$\boldsymbol{Q}_0, \boldsymbol{s}_0$	The basis change matrix and solution to the unregularized Problem 11
$\boldsymbol{Q}^*(\boldsymbol{\lambda}), \boldsymbol{s}^*(\boldsymbol{\lambda})$	The basis matrix and solution for a hyperparameter Problem 11
$\boldsymbol{a}(\boldsymbol{x}, \boldsymbol{Q})$	The data inputs which have had their bases changed by $\boldsymbol{Q}$
$\boldsymbol{W}, \boldsymbol{b}$	A layers weight matrix and bias
D_{out}, D_{in}	A layers input dimensionality and output dimensionality
$\frac{\partial \mathcal{L}_T(\boldsymbol{\lambda}, \mathbf{w})}{\partial \boldsymbol{\lambda}}$	The (validation loss) direct (hyperparameter) gradient
$\frac{\partial \mathbf{w}^*(\boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}}$	The (elementary parameter) response gradient
$\frac{\partial \mathcal{L}_T(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda}))}{\partial \mathbf{w}^*(\boldsymbol{\lambda})} \frac{\partial \mathbf{w}^*(\boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}}$	The (validation loss) response gradient
$\frac{d \mathcal{L}_T(\boldsymbol{\lambda}, \mathbf{w})}{d \boldsymbol{\lambda}}$	The hyperparameter gradient: a sum of the validation losses direct and response gradients

Table 4: Metaparameter Defaults

Metaparameter	LSTM	CNN
Train Steps / Validation Steps	2 / 1	2 / 1
Entropy Weight	0	0
Entropy Tuned	False	True
Initial Perturbation Scale	1	1

B PROOFS

B.1 LEMMA 1

Because $\mathbf{w}_0$ solves Problem 4b given $\boldsymbol{\lambda}_0$, by the first-order optimality condition, we must have:

$$\frac{\partial f}{\partial \mathbf{w}}(\boldsymbol{\lambda}_0, \mathbf{w}_0) = 0 \tag{15}$$

The Jacobian of $\partial f / \partial \mathbf{w}$ decomposes as a block matrix with subblocks given by:

$$\left[\begin{array}{c|c} \frac{\partial^2 f}{\partial \boldsymbol{\lambda} \partial \mathbf{w}} & \frac{\partial^2 f}{\partial \mathbf{w}^2} \end{array} \right] \tag{16}$$

We know f is $\mathcal{C}^2$ in some neighborhood of $(\boldsymbol{\lambda}_0, \mathbf{w}_0)$, so $\partial f / \partial \mathbf{w}$ is continuously differentiable in this neighborhood. By assumption, the Hessian $\partial^2 f / \partial \mathbf{w}^2$ is positive definite and hence invertible at $(\boldsymbol{\lambda}_0, \mathbf{w}_0)$. By the Implicit Function Theorem, there exists a neighborhood V of $\boldsymbol{\lambda}_0$ and a unique continuously differentiable function $\mathbf{w}^* : V \to \mathbb{R}^m$ such that $\partial f / \partial \mathbf{w}(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda})) = 0$ for $\boldsymbol{\lambda} \in V$ and $\mathbf{w}^*(\boldsymbol{\lambda}_0) = \mathbf{w}_0$.

Furthermore, by continuity we know there is a neighborhood $W_1 \times W_2$ of $(\boldsymbol{\lambda}_0, \mathbf{w}_0)$ such that $\partial^2 f / \partial \mathbf{w}^2$ is positive definite on this neighborhood. Setting $U = V \cap W_1 \cap (\mathbf{w}^*)^{-1}(W_2)$, we can conclude that $\partial^2 f / \partial \mathbf{w}^2(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda})) \succ 0$ for all $\boldsymbol{\lambda} \in U$. Combining this with $\partial f / \partial \mathbf{w}(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda})) = 0$ and using second-order sufficient optimality conditions, we conclude that $\mathbf{w}^*(\boldsymbol{\lambda})$ is the unique solution to Problem 4b for all $\boldsymbol{\lambda} \in U$.

B.2 LEMMA 2

This discussion mostly follows from Hastie et al. (2001). We let $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ denote the data matrix where N is the number of training examples, and D is the dimensionality of the data. We let $\boldsymbol{t} \in \mathbb{R}^N$ denote the associated targets. We can write the SVD decomposition of $\boldsymbol{X}$ as:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T \tag{17}$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are $N \times D$ and $D \times D$ orthogonal matrices and $\boldsymbol{D}$ is a diagonal matrix with entries $d_1 \geq d_2 \geq \cdots \geq d_D \geq 0$. We next simplify the function $y(\boldsymbol{x}; \mathbf{w})$ by setting $\boldsymbol{v} = \boldsymbol{s}^T \boldsymbol{Q}$, so that $y(\boldsymbol{x}; \mathbf{w}) = \boldsymbol{s}^T \boldsymbol{Q} \boldsymbol{x} = \boldsymbol{v}^T \boldsymbol{x}$. We see that the Jacobian $\partial y / \partial \boldsymbol{x} \equiv \boldsymbol{v}$ is constant, and Problem 11 simplifies to standard L_2-regularized least-squares linear regression with the following loss function:

$$\sum_{(\boldsymbol{x}, t) \in \mathcal{D}} (\boldsymbol{v}^T \boldsymbol{x} - t)^2 + \frac{1}{|\mathcal{D}|} \exp(\lambda) \|\boldsymbol{v}\|^2 \tag{18}$$

It is well-known (see Hastie et al. (2001), Chapter 3) that the optimal solution $\boldsymbol{v}^*(\lambda)$ minimizing Equation 18 is given by:

$$\boldsymbol{v}^*(\lambda) = (\boldsymbol{X}^T \boldsymbol{X} + \exp(\lambda) \boldsymbol{I})^{-1} \boldsymbol{X}^T \boldsymbol{t} = \boldsymbol{V}(\boldsymbol{D}^2 + \exp(\lambda) \boldsymbol{I})^{-1} \boldsymbol{D} \boldsymbol{U}^T \boldsymbol{t} \tag{19}$$

Furthermore, the optimal solution $\boldsymbol{v}^*$ to the unregularized version of Problem 18 is given by:

$$\boldsymbol{v}^* = \boldsymbol{V} \boldsymbol{D}^{-1} \boldsymbol{U}^T \boldsymbol{t} \tag{20}$$

Recall that we defined $\boldsymbol{Q}_0 = \boldsymbol{V}^T$, i.e. the change-of-basis matrix from the standard basis to the principal components of the data matrix, and we defined $\boldsymbol{s}_0$ to solve the unregularized regression problem given $\boldsymbol{Q}_0$. Thus, we require that $\boldsymbol{Q}_0^T \boldsymbol{s}_0 = \boldsymbol{v}^*$ which implies $\boldsymbol{s}_0 = \boldsymbol{D}^{-1}\boldsymbol{U}^T\boldsymbol{t}$.

There are not unique solutions to Problem 11, so we take any functions $\boldsymbol{Q}(\lambda), \boldsymbol{s}(\lambda)$ which satisfy $\boldsymbol{Q}(\lambda)^T\boldsymbol{s}(\lambda) = \boldsymbol{v}^*(\lambda)$ as "best-response functions". We will show that our chosen functions $\boldsymbol{Q}^*(\lambda) = \sigma(\lambda\boldsymbol{w} + \boldsymbol{b}) \odot \boldsymbol{Q}_0$ and $\boldsymbol{s}^*(\lambda) = \boldsymbol{s}_0$, where $\boldsymbol{w} = \boldsymbol{0}$ and $b_i = 2\log(d_i)$ for $i = 1, \ldots, D$, meet this criteria. We start by noticing that for any $d \in \mathbb{R}_+$, we have:

$$\sigma(\lambda + 2\log(d)) = \frac{1}{1 + \exp(-\lambda - 2\log(d))} = \frac{1}{1 + d^{-2}\exp(\lambda)} = \frac{d^2}{d^2 + \exp(\lambda)} \tag{21}$$

It follows that:

$$\begin{aligned}
\boldsymbol{Q}^*(\lambda)^T\boldsymbol{s}^*(\lambda) &= [\sigma(\lambda\boldsymbol{w} + \boldsymbol{b}) \odot \boldsymbol{Q}_0]^T\boldsymbol{s}_0 && (22)\\
&= \left[\operatorname{diag}\begin{pmatrix}\sigma(\lambda + 2\log(d_1))\\ \vdots \\ \sigma(\lambda + 2\log(d_m))\end{pmatrix}\boldsymbol{Q}_0\right]^T \boldsymbol{s}_0 && (23)\\
&= \boldsymbol{Q}_0^T\left[\operatorname{diag}\begin{pmatrix}\frac{d_1^2}{d_1^2+\exp(\lambda)}\\ \vdots \\ \frac{d_D^2}{d_D^2+\exp(\lambda)}\end{pmatrix}\right]\boldsymbol{s}_0 && (24)\\
&= \boldsymbol{V}\left[\operatorname{diag}\begin{pmatrix}\frac{d_1^2}{d_1^2+\exp(\lambda)}\\ \vdots \\ \frac{d_D^2}{d_D^2+\exp(\lambda)}\end{pmatrix}\right]\boldsymbol{D}^{-1}\boldsymbol{U}^T\boldsymbol{t} && (25)\\
&= \boldsymbol{V}\left[\operatorname{diag}\begin{pmatrix}\frac{d_1}{d_1^2+\exp(\lambda)}\\ \vdots \\ \frac{d_D}{d_D^2+\exp(\lambda)}\end{pmatrix}\right]\boldsymbol{U}^T\boldsymbol{t} && (26)\\
&= \boldsymbol{V}\left[(\boldsymbol{D}^2 + \exp(\lambda)\boldsymbol{I})^{-1}\boldsymbol{D}\right]\boldsymbol{U}^T\boldsymbol{t} && (27)\\
&= \boldsymbol{v}^*(\lambda) && (28)
\end{aligned}$$

B.3 THEOREM 3

By assumption, f is quadratic so there exist $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{B} \in \mathbb{R}^{n\times m}$, $\boldsymbol{C} \in \mathbb{R}^{m\times m}$ and $\boldsymbol{d} \in \mathbb{R}^n, \boldsymbol{e} \in \mathbb{R}^m$ such that:

$$f(\boldsymbol{\lambda}, \mathbf{w}) = \frac{1}{2}\begin{pmatrix}\boldsymbol{\lambda}^T & \mathbf{w}^T\end{pmatrix}\begin{pmatrix}\boldsymbol{A} & \boldsymbol{B}\\ \boldsymbol{B}^T & \boldsymbol{C}\end{pmatrix}\begin{pmatrix}\boldsymbol{\lambda}\\ \mathbf{w}\end{pmatrix} + \boldsymbol{d}^T\boldsymbol{\lambda} + \boldsymbol{e}^T\mathbf{w} \tag{29}$$

Furthermore, since we assume $\partial^2 f/\partial\mathbf{w}^2 \succ \boldsymbol{0}$, we must have $\boldsymbol{C} \succ \boldsymbol{0}$. One can easily compute that:

$$\frac{\partial f}{\partial \mathbf{w}}(\boldsymbol{\lambda}, \mathbf{w}) = \boldsymbol{B}^T\boldsymbol{\lambda} + \boldsymbol{C}\mathbf{w} + \boldsymbol{e} \tag{30}$$

$$\frac{\partial^2 f}{\partial \mathbf{w}^2}(\boldsymbol{\lambda}, \mathbf{w}) = \boldsymbol{C} \tag{31}$$

Setting the derivative equal to $\boldsymbol{0}$ and using second-order sufficient conditions, we have:

$$\mathbf{w}^*(\boldsymbol{\lambda}) = -\boldsymbol{C}^{-1}(\boldsymbol{e} - \boldsymbol{B}^T\boldsymbol{\lambda}) \tag{32}$$

Hence, we find:

$$\frac{\partial \mathbf{w}^*}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}) = -\boldsymbol{C}^{-1}\boldsymbol{B}^T \tag{33}$$

We can substitute $\hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda}) = \boldsymbol{U}\boldsymbol{\lambda} + \boldsymbol{b}$ and the definition of f into equation 9 to obtain:

$$\begin{aligned} e(\boldsymbol{U}, \boldsymbol{b}, \sigma) := \mathbb{E}_{\boldsymbol{\epsilon}\sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})}[&(\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon})^T \boldsymbol{A}(\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon}) + 2(\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon})^T \boldsymbol{B}(\boldsymbol{U}(\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon}) + \boldsymbol{b}) \\ &+ (\boldsymbol{U}(\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon}) + \boldsymbol{b})^T \boldsymbol{C}(\boldsymbol{U}(\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon}) + \boldsymbol{b}) \\ &+ \boldsymbol{d}^T(\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon}) + \boldsymbol{e}^T(\boldsymbol{U}(\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon}) + \boldsymbol{b})] \end{aligned} \tag{34}$$

Expanding above, we can find that equation 34 is equal to:

$$\mathbb{E}_{\boldsymbol{\epsilon}\sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})}\left[\textcircled{1} + \textcircled{2} + \textcircled{3} + \textcircled{4}\right] \tag{35}$$

where we have:

$$\textcircled{1} = \frac{1}{2}\left(\boldsymbol{\lambda}_0^T \boldsymbol{A}\boldsymbol{\lambda}_0 + 2\boldsymbol{\epsilon}^T \boldsymbol{A}\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon}^T \boldsymbol{A}\boldsymbol{\epsilon}\right) \tag{36}$$

$$\textcircled{2} = \boldsymbol{\lambda}_0^T \boldsymbol{B}\boldsymbol{U}\boldsymbol{\lambda}_0 + \boldsymbol{\lambda}_0^T \boldsymbol{B}\boldsymbol{U}\boldsymbol{\epsilon} + \boldsymbol{\lambda}_0^T \boldsymbol{B}\boldsymbol{b} + \boldsymbol{\epsilon}^T \boldsymbol{B}\boldsymbol{U}\boldsymbol{\lambda}_0 + \boldsymbol{\epsilon}^T \boldsymbol{B}\boldsymbol{U}\boldsymbol{\epsilon} + \boldsymbol{\epsilon}^T \boldsymbol{B}\boldsymbol{b} \tag{37}$$

$$\begin{aligned} \textcircled{3} = \frac{1}{2}(&\boldsymbol{\lambda}_0^T \boldsymbol{U}^T \boldsymbol{C}\boldsymbol{U}\boldsymbol{\lambda}_0 + \boldsymbol{\lambda}_0 \boldsymbol{U}^T \boldsymbol{C}\boldsymbol{U}\boldsymbol{\epsilon} + \boldsymbol{\lambda}_0 \boldsymbol{U}^T \boldsymbol{C}\boldsymbol{b} + \boldsymbol{\epsilon}^T \boldsymbol{U}^T \boldsymbol{C}\boldsymbol{U}\boldsymbol{\lambda}_0 \\ &+ \boldsymbol{\epsilon}^T \boldsymbol{U}^T \boldsymbol{C}\boldsymbol{U}\boldsymbol{\epsilon} + \boldsymbol{\epsilon}^T \boldsymbol{U}^T \boldsymbol{C}\boldsymbol{b} + \boldsymbol{b}^T \boldsymbol{C}\boldsymbol{U}\boldsymbol{\lambda}_0 + \boldsymbol{b}^T \boldsymbol{C}\boldsymbol{U}\boldsymbol{\epsilon} + \boldsymbol{b}^T \boldsymbol{C}) \end{aligned} \tag{38}$$

$$\textcircled{4} = \boldsymbol{d}^T \boldsymbol{\lambda}_0 + \boldsymbol{d}^T \boldsymbol{\epsilon} + \boldsymbol{e}^T \boldsymbol{U}\boldsymbol{\lambda}_0 + \boldsymbol{e}^T \boldsymbol{U}\boldsymbol{\epsilon} + \boldsymbol{e}^T \boldsymbol{b} \tag{39}$$

We can simplify these expressions considerably by using linearity of expectation and that $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\sigma)$ has mean $\mathbf{0}$. Doing so, then differentiating and making use of various matrix-derivative equalities (Duchi, 2007), we find:

$$\frac{\partial e}{\partial \boldsymbol{b}}(\boldsymbol{U}, \boldsymbol{b}, \sigma) = \frac{1}{2}\boldsymbol{C}^T \boldsymbol{U}\boldsymbol{\lambda}_0 + \frac{1}{2}\boldsymbol{C}\boldsymbol{U}\boldsymbol{\lambda}_0 + \boldsymbol{B}^T \boldsymbol{\lambda}_0 + \boldsymbol{e} + \boldsymbol{C}\boldsymbol{b} \tag{40}$$

$$\frac{\partial e}{\partial \boldsymbol{U}}(\boldsymbol{U}, \boldsymbol{b}, \sigma) = \boldsymbol{B}^T \boldsymbol{\lambda}_0 \boldsymbol{\lambda}_0^T + \sigma^2 \boldsymbol{B}^T + \boldsymbol{C}\boldsymbol{b}\boldsymbol{\lambda}_0^T + \boldsymbol{e}\boldsymbol{\lambda}_0^T + \boldsymbol{C}\boldsymbol{U}\boldsymbol{\lambda}_0 \boldsymbol{\lambda}_0^T + \sigma^2 \boldsymbol{C}\boldsymbol{U} \tag{41}$$

Setting the derivative ${}^{\partial e}/_{\partial \boldsymbol{b}}(\boldsymbol{U}, \boldsymbol{b}, \sigma)$ equal to $\mathbf{0}$, we have:

$$\boldsymbol{b} = -\boldsymbol{C}^{-1}(\boldsymbol{C}^T \boldsymbol{U}\boldsymbol{\lambda}_0 + \boldsymbol{B}^T \boldsymbol{\lambda}_0 + \boldsymbol{e}) \tag{42}$$

Setting the derivative for ${}^{\partial e}/_{\partial \boldsymbol{U}}(\boldsymbol{U}, \boldsymbol{b}, \sigma)$ equal to $\mathbf{0}$, we have:

$$\boldsymbol{C}\boldsymbol{U}(\boldsymbol{\lambda}_0 \boldsymbol{\lambda}_0^T + \sigma^2 \boldsymbol{C}) = -\boldsymbol{B}^T \boldsymbol{\lambda}_0 \boldsymbol{\lambda}_0^T - \sigma^2 \boldsymbol{B}^T - \boldsymbol{C}\boldsymbol{b}\boldsymbol{\lambda}_0^T - \boldsymbol{e}\boldsymbol{\lambda}_0^T \tag{43}$$

Substituting the expression for $\boldsymbol{b}$ given by equation 42 into equation 43 and simplifying gives:

$$\boldsymbol{C}\boldsymbol{U}(\boldsymbol{\lambda}_0 \boldsymbol{\lambda}_0^T + \sigma^2 \boldsymbol{I}) = -\sigma^2 \boldsymbol{B}^T + \boldsymbol{C}\boldsymbol{U}\boldsymbol{\lambda}_0 \boldsymbol{\lambda}_0^T \tag{44}$$

$$\implies \sigma^2 \boldsymbol{C}\boldsymbol{U} = -\sigma^2 \boldsymbol{B}^T \tag{45}$$

$$\implies \boldsymbol{U} = \boldsymbol{C}^{-1}\boldsymbol{B} \tag{46}$$

This is exactly the best-response Jacobian ${}^{\partial \mathbf{w}^*}/_{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda})$ as given by equation 33. Substituting $\boldsymbol{U} = \boldsymbol{C}^{-1}\boldsymbol{B}$ into the equation 42 gives:

$$\boldsymbol{b} = \boldsymbol{C}^{-1}\boldsymbol{B}^T \boldsymbol{\lambda}_0 - \boldsymbol{C}^{-1}\boldsymbol{B}^T \boldsymbol{\lambda}_0 - \boldsymbol{C}^{-1}\boldsymbol{e} \tag{47}$$

This is $\mathbf{w}^*(\boldsymbol{\lambda}) - {}^{\partial \mathbf{w}^*}/_{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda})$, as claimed.

B.4 Best-Response Gradient Lemma

Lemma 4. *Under the same conditions as Lemma 1 and using the same notation, for all $\boldsymbol{\lambda} \in U$, we have that:*

$$\frac{\partial \boldsymbol{w}^*}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}) = -\left[\frac{\partial^2 f}{\partial \boldsymbol{w}^2}(\boldsymbol{\lambda}, \boldsymbol{w}^*(\boldsymbol{\lambda}))\right]^{-1} \frac{\partial^2 f}{\partial \boldsymbol{\lambda} \partial \boldsymbol{w}}(\boldsymbol{\lambda}, \boldsymbol{w}^*(\boldsymbol{\lambda})) \tag{48}$$

Proof. Define $\iota^* : U \to \mathbb{R}^n \times \mathbb{R}^m$ by $\iota^*(\boldsymbol{\lambda}) = (\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda}))$. By first-order optimality conditions, we know:

$$\left(\frac{\partial f}{\partial \mathbf{w}} \circ \iota^*\right)(\boldsymbol{\lambda}) = 0 \quad \forall \boldsymbol{\lambda} \in U \tag{49}$$

Hence, for all $\boldsymbol{\lambda} \in U$:

$$0 = \frac{\partial}{\partial \boldsymbol{\lambda}} \left(\frac{\partial f}{\partial \mathbf{w}} \circ \iota^*\right)(\boldsymbol{\lambda}) \tag{50}$$

$$= \frac{\partial^2 f}{\partial \mathbf{w}^2}(\iota^*(\boldsymbol{\lambda})) \frac{\partial \mathbf{w}^*}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}) + \frac{\partial^2 f}{\partial \boldsymbol{\lambda} \partial \mathbf{w}}(\iota^*(\boldsymbol{\lambda})) \tag{51}$$

$$= \frac{\partial^2 f}{\partial \mathbf{w}^2}(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda})) \frac{\partial \mathbf{w}^*}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}) + \frac{\partial^2 f}{\partial \boldsymbol{\lambda} \partial \mathbf{w}}(\boldsymbol{\lambda}, \mathbf{w}^*(\lambda)) \tag{52}$$

Rearranging gives Equation 48. □

C Best-Response Approximations for Convolutional Filters

We let L denote the number of layers, C_l the number of channels in layer l's feature map, and K_l the size of the kernel in layer l. We let $\boldsymbol{W}^{l,c} \in \mathbb{R}^{C_{l-1} \times K_l \times K_l}$ and $\boldsymbol{b}^{l,c} \in \mathbb{R}$ denote the weight and bias respectively of the cth convolution kernel in layer l (so $c \in \{1, \ldots, C_l\}$). For $\boldsymbol{u}^{l,c}, \boldsymbol{a}^{l,c} \in \mathbb{R}^n$, we define best-response approximations $\hat{\boldsymbol{W}}^{l,c}$ and $\hat{\boldsymbol{b}}^{l,c}$ by:

$$\hat{\boldsymbol{W}}^{l,c}(\boldsymbol{\lambda}) = \langle \boldsymbol{\lambda}, \boldsymbol{u}^{l,c} \rangle \odot \boldsymbol{W}^{l,c}_{\text{scale}} + \boldsymbol{W}^{l,c}_{\text{shift}} \tag{53}$$

$$\hat{\boldsymbol{b}}^{l,c}(\boldsymbol{\lambda}) = \langle \boldsymbol{\lambda}, \boldsymbol{a}^{l,c} \rangle \odot \boldsymbol{b}^{l,c}_{\text{scale}} + \boldsymbol{b}^{l,c}_{\text{shift}} \tag{54}$$

Thus, the best-response parameters used for modeling $\boldsymbol{W}^{l,c}$, $\boldsymbol{b}^l$ are $\boldsymbol{\phi}^{l,c} = \{\boldsymbol{u}^{l,c}, \boldsymbol{a}^{l,c}, \boldsymbol{W}^{l,c}_{\text{scale}}, \boldsymbol{W}^{l,c}_{\text{shift}}, \boldsymbol{b}^{l,c}_{\text{scale}}, \boldsymbol{b}^{l,c}_{\text{shift}}\}$. We can compute the number of parameters used as $|\boldsymbol{\phi}^{l,c}| = 2n + 2(|\boldsymbol{W}^{l,c}| + |\boldsymbol{b}^{l,c}|)$. Summing over channels c, we find the total number of parameters is $2nC_l + 2p$, where p is the total number of parameters in the normal CNN layer. Hence, we use twice the number of parameters in a normal CNN, plus an overhead depending on the number of hyperparameters.

For an implementation in code, see Appendix G.

D Language Modelling Experiment Details

Here we present additional details on the setup of our LSTM experiments on PTB, and on the role of each hyperparameter we tune.

We tune variational dropout (re-using the same dropout mask for each step in a sequence) on the input to the LSTM, the hidden state between the LSTM layers, and the output of the LSTM. We also tune embedding dropout, which sets entire rows of the word embedding matrix to 0, effectively removing certain words from all sequences. We regularize the hidden-to-hidden weight matrix using DropConnect (zeroing out weights rather than activations). Finally, we use activation regularization (AR) and temporal activation regularization (TAR). AR penalizes large activations, and is defined as:

$$\alpha ||m \odot h_t||_2 \tag{55}$$

where m is a dropout mask and h_t is the output of the LSTM at time t. TAR is a slowness regularizer, defined as:

$$\beta ||h_t - t_{t+1}||_2 \tag{56}$$

We tune the scaling coefficients α and β.

E IMAGE CLASSIFICATION EXPERIMENT DETAILS

Here we present additional details on the CNN experiments. For all results we hold out 20% of the data for validation.

Our optimization algorithm for the elementary model in the baselines is as follows: We use SGD with a learning rate of 0.01, momentum of 0.9, mini-batches of size 128, and run for at most 200 epochs.

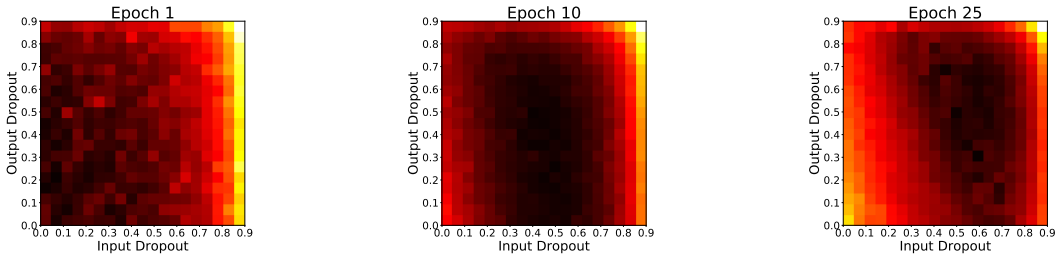
Our optimization algorithm for the ST-CNN is as follows: The ST-CNN is optimized with the same algorithm as the elementary model in the baselines. In other words, we use SGD with learning rate 0.01, momentum of 0.9, mini-batches of size 128. We use the same optimization algorithm as the ST-LSTM for the hyperparameters. This means the hyperparameters are optimized with Adam using a learning rate of 0.003. We alternate between training the best-response approximation and hyperparameters with the same schedule as the ST-LSTM - i.e. $T_{train} = 2$ steps on the training step and $T_{valid} = 1$ steps on the validation set.

Additionally, we use an entropy weight of 0 during training when optimizing the scale of the hyperparameter distributions.

F ADDITIONAL DETAILS ON HYPERPARAMETER SCHEDULES

Here, we draw connections between the hyperparameter schedules learned by STNs and curriculum learning. Curriculum learning is an instance of a family of *continuation methods* (Allgower & Georg, 2012), which optimize non-convex functions by solving a sequence of functions that are ordered by increasing difficulty. In a continuation method, one considers a family of training criteria $C_\lambda(\mathbf{w})$ with parameter λ, where $C_1(\mathbf{w})$ is the true/final objective we wish to minimize, and $C_0(\mathbf{w})$ represents the training criterion for a simpler version of the problem. One starts by optimizing $C_0(\mathbf{w})$ and then gradually increases λ from 0 to 1, while keeping $\mathbf{w}$ at a local minimum of $C_\lambda(\mathbf{w})$. When training an RL-NTM (Zaremba & Sutskever, 2015), the authors maintain a distribution over the complexity c of a problem instance (defined as the maximum length of the target output for each input), which is shifted towards harder problems as the performance of the model improves.

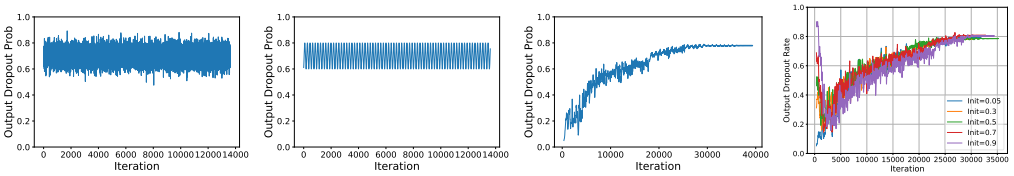
Figure 7 shows the validation perplexity for different sets of input and output dropout, at various points during training. The x- and y-axes represent the values of input and output dropout, respectively. We see that at the start of training, the best validation loss is achieved with small values of both input and output dropout. As we train for more epochs, the best validation performance is achieved with larger dropout rates.



(a) (b) (c)

Figure 7: Validation performance of a baseline LSTM given different settings of input and output dropout over time.

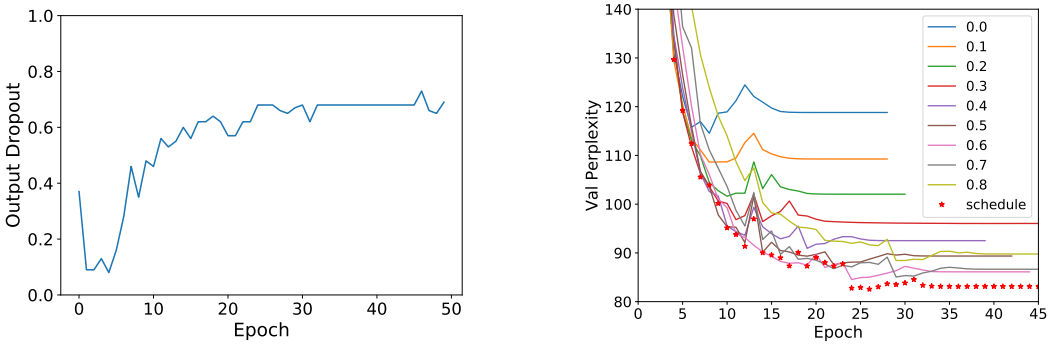
Figure 8 shows the perturbed values for output dropout we use to investigate if the improved performance yielded by STNs is due to the regularization effect, and not the schedule.



(a) $p \sim \mathcal{N}(0.68, 0.05)$ (b) $p = 0.68$ with sin noise (c) ST-LSTM (d) ST-LSTM Initial Hyperparameters

Figure 8: Output dropout rates

We can observe the benefits of greedy hyperparameter schedules by forming a schedule based on the performance in time of models from a grid search over output dropout. As shown in Figure 9, the schedule formed by taking the best output dropout value in each epoch yields better generalization than any of the fixed hyperparameter values from the initial grid search.



(a) Greedy schedule for output dropout, derived from grid search

(b) Comparison of fixed output dropout values and the dropout schedule derived from grid searches

Figure 9: Grid-search derived output dropout schedules. All experiments here use learning rate decay.

G CODE LISTINGS

In this section we provide PyTorch code listings for the approximate best-response layers used to construct ST-LSTMs and ST-CNNs: the `HyperLinear` and `HyperConv2D` classes. We also provide a stylized version of the optimization steps used on the training set and validation set.

Listing 1: HyperLinear, used as a drop-in replacement for Linear modules

```
class HyperLinear(nn.Module):

   def __init__(self, input_dim, output_dim, n_hparams):
      super(HyperLinear, self).__init__()

      self.input_dim = input_dim
      self.output_dim = output_dim
      self.n_hparams = n_hparams
      self.n_scalars = output_dim

      self.elem_w = nn.Parameter(torch.Tensor(output_dim, input_dim))
      self.elem_b = nn.Parameter(torch.Tensor(output_dim))

      self.hnet_w = nn.Parameter(torch.Tensor(output_dim, input_dim))
      self.hnet_b = nn.Parameter(torch.Tensor(output_dim))

      self.htensor_to_scalars = nn.Linear(self.n_hparams,
          self.n_scalars*2, bias=False)

      self.init_params()

   def forward(self, input, hnet_tensor):
      output = F.linear(input, self.elem_w, self.elem_b)

      if hnet_tensor is not None:
         hnet_scalars = self.htensor_to_scalars(hnet_tensor)
         hnet_wscalars = hnet_scalars[:, :self.n_scalars]
         hnet_bscalars = hnet_scalars[:, self.n_scalars:]

         hnet_out = hnet_wscalars * F.linear(input, self.hnet_w)
         hnet_out += hnet_bscalars * self.hnet_b

         output += hnet_out

      return output
```

Listing 2: HyperConv2d used as a drop-in replacement for Conv2d modules

```
class HyperConv2d(nn.Module):

   def __init__(self, in_channels, out_channels, kernel_size, padding,
       num_hparams,
      stride=1, bias=True):
      super(HyperConv2d, self).__init__()

      self.in_channels = in_channels
      self.out_channels = out_channels
      self.kernel_size = kernel_size
      self.padding = padding
      self.num_hparams = num_hparams
      self.stride = stride

      self.elem_weight = nn.Parameter(torch.Tensor(
         out_channels, in_channels, kernel_size, kernel_size))
      self.hnet_weight = nn.Parameter(torch.Tensor(
         out_channels, in_channels, kernel_size, kernel_size))
      if bias:
```

```
            self.elem_bias = nn.Parameter(torch.Tensor(out_channels))
            self.hnet_bias = nn.Parameter(torch.Tensor(out_channels))
        else:
            self.register_parameter('elem_bias', None)
            self.register_parameter('hnet_bias', None)

        self.htensor_to_scalars = nn.Linear(
            self.num_hparams, self.out_channels*2, bias=False)
        self.elem_scalar = nn.Parameter(torch.ones(1))

        self.init_params()

    def forward(self, input, htensor):
        """
        Arguments:
            input (tensor): size should be (B, C, H, W)
            htensor (tensor): size should be (B, D)
        """
        output = F.conv2d(input, self.elem_weight, self.elem_bias,
             padding=self.padding,
            stride=self.stride)
        output *= self.elem_scalar
        if htensor is not None:
            hnet_scalars = self.htensor_to_scalars(htensor)
            hnet_wscalars = hnet_scalars[:,
                :self.out_channels].unsqueeze(2).unsqueeze(2)
            hnet_bscalars = hnet_scalars[:, self.out_channels:]

            hnet_out = F.conv2d(input, self.hnet_weight,
                padding=self.padding,
               stride=self.stride)
            hnet_out *= hnet_wscalars
            if self.hnet_bias is not None:
               hnet_out += (hnet_bscalars *
                   self.hnet_bias).unsqueeze(2).unsqueeze(2)
            output += hnet_out
        return output

    def init_params(self):
        n = self.in_channels * self.kernel_size * self.kernel_size
        stdv = 1. / math.sqrt(n)
        self.elem_weight.data.uniform_(-stdv, stdv)
        self.hnet_weight.data.uniform_(-stdv, stdv)
        if self.elem_bias is not None:
            self.elem_bias.data.uniform_(-stdv, stdv)
            self.hnet_bias.data.uniform_(-stdv, stdv)

        self.htensor_to_scalars.weight.data.normal_(std=0.01)
```

Listing 3: Stylized optimization step on the training set for updating elementary parameters

```
# Perturb hyperparameters around current value in unconstrained
# parametrization.
batch_htensor = perturb(htensor, hscale)

# Apply necessary reparametrization of hyperparameters.
hparam_tensor = hparam_transform(batch_htensor)

# Sets data augmentation hyperparameters in the data loader.
dataset.set_hparams(hparam_tensor)

# Get next batch of examples and apply any input transformation
# (e.g. input dropout) as dictated by the hyperparameters.
images, labels = next_batch(dataset)
images = apply_input_transform(images, hparam_tensor)
```

```
# Run everything through the model and do gradient descent.
pred = hyper_cnn(images, batch_htensor, hparam_tensor)
xentropy_loss = F.cross_entropy(pred, labels)
xentropy_loss.backward()
cnn_optimizer.step()
```

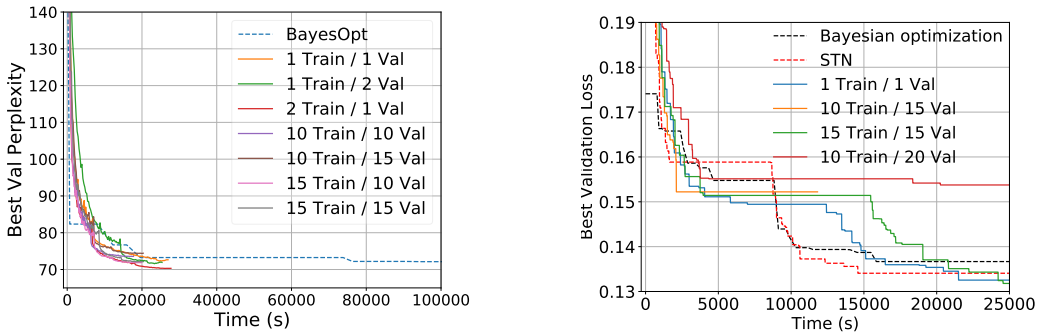
Listing 4: Stylized optimization step on the validation set for updating hyperparameters/noise scale

```
# Perturb hyperparameters around current value in unconstrained
# parametrization, so we can assess sensitivity of validation
# loss to the scale of the noise.
batch_htensor = perturb(htensor, hscale)

# Apply necessary reparametrization of hyperparameters.
hparam_tensor = hparam_transform(batch_htensor)

# Get next batch of examples and run through the model.
images, labels = next_batch(valid_dataset)
pred = hyper_cnn(images, batch_htensor, hparam_tensor)
xentropy_loss = F.cross_entropy(pred, labels)

# Add extra entropy weight term to loss.
entropy = compute_entropy(hscale)
loss = xentropy_loss - args.entropy_weight * entropy
loss.backward()

# Tune the hyperparameters.
hyper_optimizer.step()

# Tune the scale of the noise applied to hyperparameters.
scale_optimizer.step()
```

H Ablation Studies

In this section we present experiments that show how sensitive our STN models are to different meta-parameters.

In particular, we investigate the effect of using alternative schedules (Figure 10) for the number of optimization steps performed on the training and validation sets.

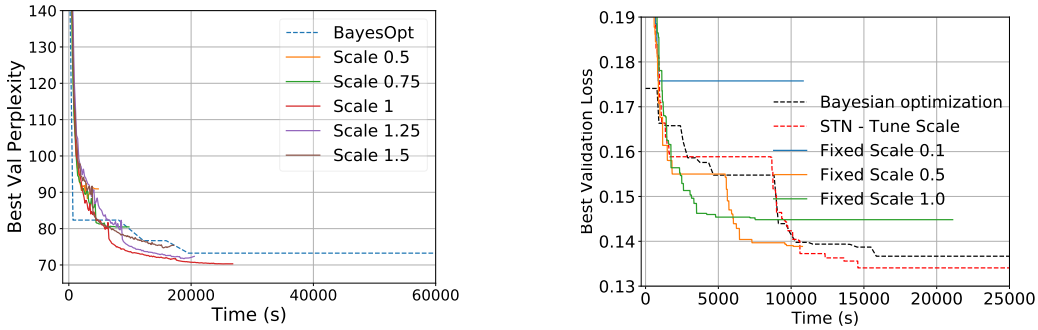


(a) LSTM Train & Valid Step Schedules

(b) CNN Train & Valid Step Schedules

Figure 10: The effect of using a different number of train/val steps. For the CNN, we include Bayesian Optimization and the reported STN parameters for comparison. During these experiments we found schedules which achieve better final loss with CNNs.

Additionally, we investigate the effect of using different initial perturbation scales for the hyperparameters, which are either fixed or tuned (Figure 11).



(a) LSTM Scales

(b) CNN Fixed & Tuned Scales

Figure 11: The effect of using a different perturbation scales. For the CNN, we include Bayesian Optimization and the reported STN parameters for comparison. For (a), the perturbation scales are fixed.

Finally, we have included experiments which allow us to assess the importance of the response gradient 12. This is done by either including or not including the response gradient. We omit the response gradient by not propagating the hyperparameter gradient through the hypernetwork.

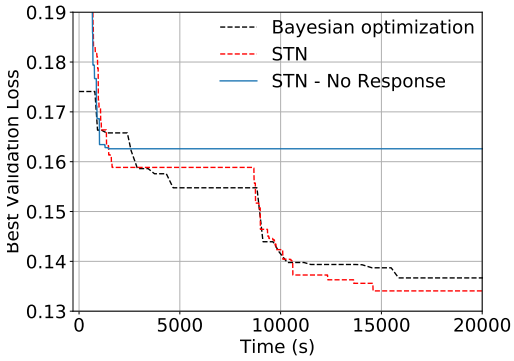


Figure 12: **CNN With & Without the Response Gradient** The effect of using the response gradient. For the CNN, we include Bayesian Optimization and the reported STN parameters for comparison.