Scalable Gradient-Based Tuning of Continuous Regularization Hyperparameters

Jelena Luketina[1] JELENA.LUKETINA@AALTO.FI
Mathias Berglund[1] MATHIAS.BERGLUND@AALTO.FI
Klaus Greff[2] KLAUS@IDSIA.CH
Tapani Raiko[1] TAPANI.RAIKO@AALTO.FI
[1]Department of Computer Science, Aalto University, Finland

[2]IDSIA, Dalle Molle Institute for Artificial Intelligence, USI-SUPSI, Manno-Lugano, Switzerland

Abstract

Hyperparameter selection generally relies on running multiple full training trials, with selection based on validation set performance. We propose a gradient-based approach for locally adjusting hyperparameters during training of the model. Hyperparameters are adjusted so as to make the model parameter gradients, and hence updates, more advantageous for the validation cost. We explore the approach for tuning regularization hyperparameters and find that in experiments on MNIST, SVHN and CIFAR-10, the resulting regularization levels are within the optimal regions. The additional computational cost depends on how frequently the hyperparameters are trained, but the tested scheme adds only 30% computational overhead regardless of the model size. Since the method is significantly less computationally demanding compared to similar gradient-based approaches to hyperparameter optimization, and consistently finds good hyperparameter values, it can be a useful tool for training neural network models.

1. Introduction

Specifying and training artificial neural networks requires several design choices that are often not trivial to make. Many of these design choices boil down to the selection of hyperparameters. The process of hyperparameter selection is in practice often based on trial-and-error and grid or random search (Bergstra and Bengio, 2012). There are also a number of automated methods (Bergstra *et al.*, 2011; Snoek *et al.*, 2012), all of which rely on multiple complete training runs with varied fixed hyperparameters, with the hyperparameter selection based on the validation set performance.

Although effective, these methods are expensive as the user needs to run multiple full training runs. In the worst case, the number of needed runs also increases exponentially with the number of hyperparameters to tune, if an extensive exploration is desired. In many practical applications such an approach is too tedious and time-consuming, and it would be useful if a method existed that could automatically find acceptable hyperparameter values in one training run even if the user did not have a strong intuition regarding good values to try for the hyperparameters.

In contrast to these methods, we treat hyperparameters similar to elementary[1] parameters during training, in that we simultaneously update both sets of parameters using stochastic gradient descent. The gradient of elementary parameters is computed as in usual training from the cost of the regularized model on the training set, while the gradient of hyperparameters (hypergradient) comes from the cost of the unregularized model on the validation set. For simplicity, we will refer to the training set as T_1 and to the validation set (or any other data set used exclusively for training the hyperparameters) as T_2. The method itself will be called $T_1 - T_2$, referring to the two simultaneous optimization processes.

Similar approaches have been proposed since the late 1990s; however, these methods either require computation of the inverse Hessian (Larsen *et al.*, 1998; Bengio, 2000; Chen and Hagan, 1999; Foo *et al.*, 2008), or propagating gradients

Proceedings of the 33^{rd} *International Conference on Machine Learning*, New York, NY, USA, 2016. JMLR: W&CP volume 48. Copyright 2016 by the author(s).

[1]Borrowing the expression from Maclaurin *et al.* (2015), we refer to the model parameters customarily trained with backpropagation as *elementary* parameters, and to all other parameters as hyperparameters.

through the entire history of parameter updates Maclaurin *et al.* (2015). Moreover, these methods make changes to the hyperparameter only once the elementary parameter training has ended. These drawbacks make them too expensive for use in modern neural networks, which often require millions of parameters and large data sets.

Elements distinguishing our approach are:

1. By making some very rough approximations, our method for modifying hyperparameters avoids using computationally expensive terms, including the computation of the Hessian or its inverse. This is because with the $T_1 - T_2$ method, hyperparameter updates are based on stochastic gradient descent, instead of Newton's method. Furthermore, any dependency of elementary parameters on hyperparameters beyond the last update is disregarded. As a result, additional computational and memory overhead therefore becomes comparable to back-propagation.

2. Hyperparameters are trained simultaneously with elementary parameters. Feedback and feedforward passes can be computed simultaneously for the training and validation set, further reducing the computational cost.

3. We add batch normalization (Ioffe and Szegedy, 2015) and adaptive learning rates (Kingma and Ba, 2015) to the process of hyperparameter training, which diminishes some of the problems of gradient-based hyperparameter optimization. Through batch normalization, we can counter internal covariate shifts. This eliminates the need for different learning rates at each layer, as well as speeding up adjustment of the elementary parameters to the changes in hyperparameters. This is particularly relevant when parametrizing each of the layers with a separate hyperparameter.

A common assumption is that the choice of hyperparameters affects the whole training trajectory, i.e. changing a hyperparameter on the fly during training has a significant effect on the training trajectory. This "hysteresis effect" implies that in order to measure how a hyperparameter combination influences the validation set performance, the hyperparameters need to be kept fixed during the whole training procedure. However, to our knowledge this has not been systematically studied. If the hysteresis effect is weak enough and the largest changes to the hyperparameter happen early on, it becomes possible to train the model while tuning the hyperparameters on the fly during training, and then use the final hyperparameter values to retrain the model if a fixed set of hyperparameters is desired. We also explore this approach.

An important design choice when training neural network models is which regularization strategy to use in order to ensure that the model generalizes to data not included in the training set. Common regularization strategies involve adding explicit terms to the model or the cost function during training, such as penalty terms on the model weights or injecting noise to inputs or neuron activations. Injecting noise is particularly relevant for denoising autoencoders and related models (Vincent *et al.*, 2010; Rasmus *et al.*, 2015), where performance strongly depends on the level of noise.

Although the proposed method could work in principle for any continuous hyperparameter, we have specifically focused on studying tuning of regularization hyperparameters. We have chosen to use Gaussian noise added to the inputs and hidden layer activations, in addition to L_2 weight penalty. A third, often used, regularization method that involves a hyperparameter choice is dropout (Srivastava *et al.*, 2014). However, we have omitted studying dropout as it is not trivial to compute a gradient on the dropout rate. Moreover, dropout can be seen as a form of multiplicative Gaussian noise (Wang and Manning, 2013). We also omit study adapting the learning rate, since we suspect that the local gradient information is not sufficient to determine optimal learning rates.

In Section 2 we present details on the proposed method. The method is tested with multiple MLP and CNN network structures and regularization schemes, detailed in Section 3. The results of the experiments are presented in Section 3.1.

2. Proposed Method

We propose a method, $T_1 - T_2$, for tuning continuous hyperparameters of a model using the gradient of the performance of the model on a separate validation set T_2. In essence, we train a neural network model on a training set T_1 as usual. However, for each update of the network weights and biases, i.e. the elementary parameters of the network, we tune the hyperparameters so as to make the direction of the weight update as beneficial as possible for the validation cost on a separate dataset T_2.

Formally, when training a neural network model, we try to minimize an objective function that depends on the training set, model weights and hyperparameters that determine the strength of possible regularization terms. When using gradient descent, we denote the optimization objective function $\tilde{C}_1(\cdot)$ and the corresponding weight update as:

$$\tilde{C}_1(\boldsymbol{\theta}|\boldsymbol{\lambda}, T_1) = C_1(\boldsymbol{\theta}|\boldsymbol{\lambda}, T_1) + \Omega(\boldsymbol{\theta}, \boldsymbol{\lambda}), \quad (1)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta_1 \nabla_{\boldsymbol{\theta}} \tilde{C}_1(\boldsymbol{\theta}_t|\boldsymbol{\lambda}_t, T_1), \quad (2)$$

where $\tilde{C}_1(\cdot)$ and $\Omega(\cdot)$ are cost and regularization penalty terms, $T_1 = \{(\mathbf{x}_i, \ \mathbf{y}_i)\}$ is the training data set, $\boldsymbol{\theta} = \{\mathbf{W}^l, \ \mathbf{b}^l\}$ a set of elementary parameters including weights and biases of each layer, $\boldsymbol{\lambda}$ denotes various hyperparameters that determine the strength of regularization, while η_1 is a learning rate. Subscript t refers to the iteration number.

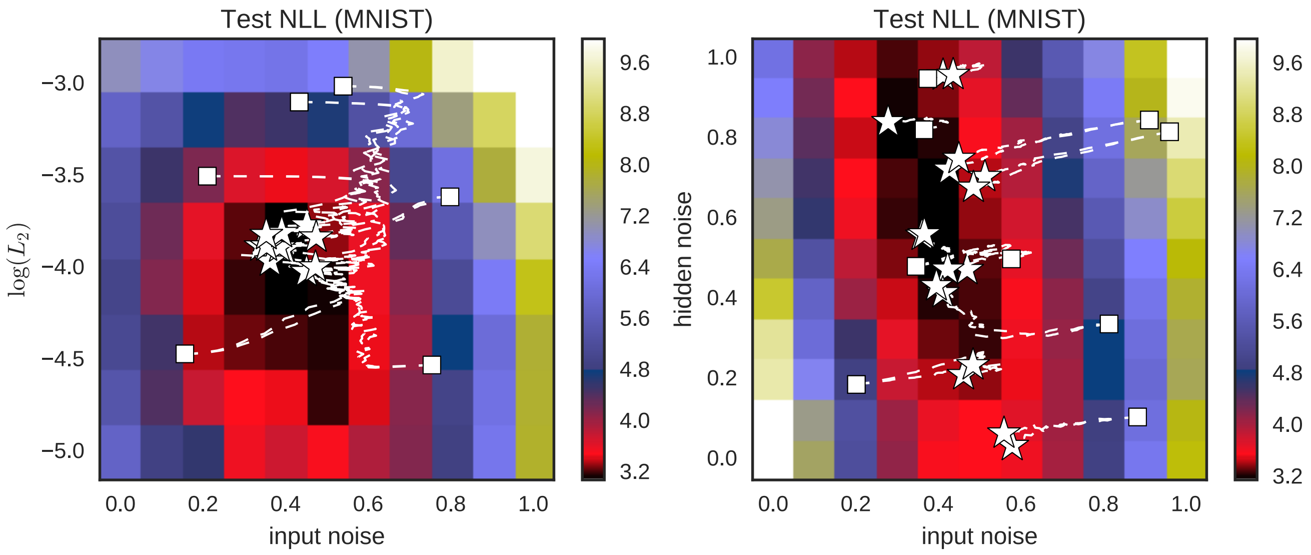

Figure 1. Left: Values of additive input noise and L_2 penalty $(n_0, log(l_2))$ during training using the $T_1 - T_2$ method for hyperparameter tuning. Trajectories are plotted over the grid search result for the same regularization pair. Initial hyperparameter values are denoted with a square, final hyperparameter values are denoted with a star. Right: Similarly constructed trajectories, on a model regularized with input and hidden layer additive noise (n_0, n_1).

Assuming $T_2 = \{(\mathbf{x}_i,\ \mathbf{y}_i)\}$ is a separate validation data set, the generalization performance of the model is measured with a validation cost $C_2(\boldsymbol{\theta}_{t+1}, T_2)$, which is usually a function of the unregularized model. Hence the value of the cost function of the actual performance of the model does not depend on the regularizer directly, but only on the elementary parameter updates. The gradient of the validation cost with respect to $\boldsymbol{\lambda}$ is:

$$\nabla_{\boldsymbol{\lambda}} C_2 = (\nabla_{\boldsymbol{\theta}} C_2)(\nabla_{\boldsymbol{\lambda}} \boldsymbol{\theta}_{t+1})$$

We only consider the influence of the regularization hyperparameter on the current elementary parameter update, $\nabla_{\boldsymbol{\lambda}} \boldsymbol{\theta}_{t+1} = \eta_1 \nabla_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{\theta}} \tilde{C}_1$ based on Eq. (2). The hyperparameter update is therefore:

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t + \eta_2 (\nabla_{\boldsymbol{\theta}} C_2)(\nabla_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{\theta}} \tilde{C}_1) \tag{3}$$

where η_2 is a learning rate.

The method is greedy in the sense that it only depends on one parameter update, and hence rests on the assumption that a good hyperparameter choice can be evaluated based on the local information within only one elementary parameter update.

2.1. Motivation and analysis

The most similar previously proposed model is the incremental gradient version of the hyperparameter update from (Chen and Hagan, 1999). However their derivation of the hypergradient assumes a Gauss-Newton update of the elementary parameters, making computation of the gradient and the hypergradient significantly more expensive.

A well justified closed form for the term $\nabla_{\boldsymbol{\lambda}} \boldsymbol{\theta}$ is available once the elementary gradient has converged (Foo *et al.*, 2008), with the update of the form (4). Comparing this expression with the $T_1 - T_2$ update, (3) can be considered as approximating (4) in the case when gradient is near convergence and the Hessian can be well approximated by identity $\nabla^2_{\boldsymbol{\theta}} \tilde{C}_1 = I$:

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t + (\nabla_{\boldsymbol{\theta}} C_2)(\nabla^2_{\boldsymbol{\theta}} \tilde{C}_1)^{-1}(\nabla_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{\theta}} \tilde{C}_1). \tag{4}$$

Another approach to hypergradient computation is given in Maclaurin *et al.* (2015). There, the term $\nabla_{\boldsymbol{\lambda}} \boldsymbol{\theta}_T$ (T denoting the final iteration number) considers effect of the hyperparameter on the entire history of updates:

$$\boldsymbol{\theta}_T = \boldsymbol{\theta}_0 + \sum_{0<k<T} \triangle \boldsymbol{\theta}_{k,k+1}(\boldsymbol{\theta}_k(\boldsymbol{\lambda}_t),\ \boldsymbol{\lambda}_t,\ \eta_k), \tag{5}$$

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t + (\nabla_{\boldsymbol{\theta}} C_2)(\nabla_{\boldsymbol{\lambda}} \boldsymbol{\theta}_T). \tag{6}$$

In the simplest case where the update is formed only with the current gradient $\triangle \boldsymbol{\theta}_{k,k+1} = -\eta_{1,k} \nabla_{\boldsymbol{\theta}} \tilde{C}_1$, i.e. not including the momentum term or adaptive learning rates, the update of a hyperparameter is formed by collecting the elements from the entire training procedure:

$$\boldsymbol{\lambda}_{t+1} = \boldsymbol{\lambda}_t + \eta_2 \nabla_{\boldsymbol{\theta}} C_2 \sum_{0<k<T} \eta_1 \nabla_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{\theta}} \tilde{C}_{1,k}. \tag{7}$$

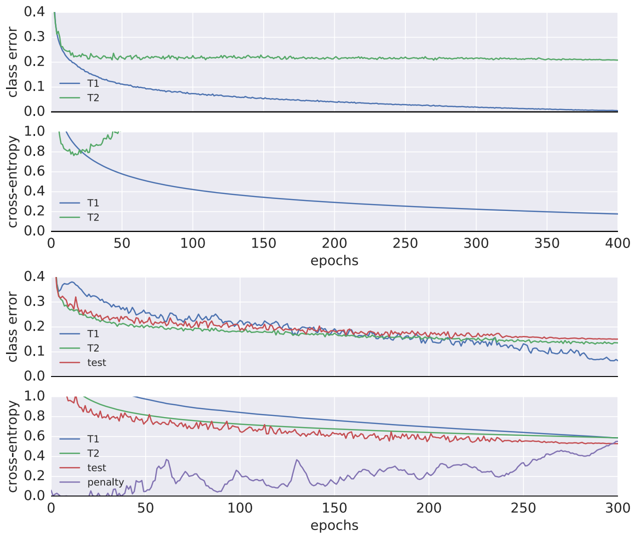

Figure 2. Evolution of classification error and cross-entropy over the course of training, for a single SVHN experiment. Top: evolution of the classification error and costs with a fixed choice of hyperparameters $(n_0, \mathbf{l_2})$. Bottom: classification error and costs during training with $T_1 - T_2$, using $(n_0, \mathbf{l_2})$ as initial hyperparameter values. $T_1 - T_2$ prevents otherwise strong overfitting.

Eq. (3) can therefore be considered as an approximation of (7), where we consider only the last update instead of backpropagating through the whole weight update history and updating the hyperparameters without resetting the weights.

In theory, approximating the Hessian with identity might cause difficulties. From Equation (3), it follows that the method converges when $(\nabla_{\boldsymbol{\theta}} C_2)(\nabla_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{\theta}} \tilde{C}_1) = \mathbf{0}$, or in other words, for all components i of the hyperparameter vector $\boldsymbol{\lambda}$, $\nabla_{\boldsymbol{\theta}} C_2$ is orthogonal to $\frac{\partial \nabla_{\boldsymbol{\theta}} \tilde{C}_1}{\partial \lambda_i}$. This is in contrast to the standard optimization processes that converge when the gradient is zero. In fact, we cannot guarantee convergence at all. Furthermore, if we replace the global (scalar) learning rate η_1 in Equation (2) with individual learning rates $\eta_{1,j}$ for each elementary-parameter $\theta_{j,t}$, the point of convergence could change.

It is clear that the identity Hessian assumption is an approximation that will not hold strictly. However, arguably, batch normalization (Ioffe and Szegedy, 2015) is eliminating part of the problem, by making the Hessian closer to identity (Vatanen *et al.*, 2013; Raiko *et al.*, 2012), making the approximation more justified. Another step towards making even closer approximation are transformations that further whiten the hidden representations (Desjardins *et al.*, 2015).

2.2. Computational cost

The most computationally expensive term of the proposed method is $(\nabla_{\boldsymbol{\theta}} C_2)(\nabla_{\boldsymbol{\lambda}} \nabla_{\boldsymbol{\theta}} \tilde{C}_1)$, where the exact complexity depends on the details of the implementation and the hyperparameter. When training $L2$ penalty regularizers $\Omega(\boldsymbol{\theta}) = \sum_j \frac{\lambda_j}{2} \theta_j^2$, the additional cost is negligible, as $\frac{\partial^2 \tilde{C}_1}{\partial \lambda_j \partial \theta_k} = \theta_j \delta_{j,k}$, where δ is the indicator function.

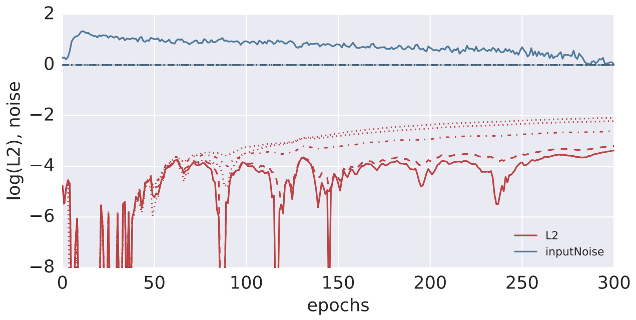

Figure 3. Evolution of the hyperparameters for the same SVHN experiment as in Figure 2 bottom. The green curve at the top shows the standard deviation of the input noise. The red curves at the bottom demonstrate values of $L2$ regularization of each layer on a log scale. The solid line refers to the input layer and least solid one to the top layer. The Figure illustrates two common patterns we observed for SVHN: firstly the noise level tends to increase in the beginning and decrease later during training, and secondly the $L2$ decay always ends up stronger for the higher layers.

The gradient of the cost with respect to a noise hyperparameter σ_l at layer l, can be computed as $\frac{\partial \tilde{C}_1}{\partial \sigma_l} = (\nabla_{\mathbf{h}_l} \tilde{C}_1) \left(\frac{\partial \mathbf{h}_l}{\partial \sigma_l}\right)^{\top}$, where $\mathbf{h}_l$ is hidden layer l activation. In case of additive Gaussian noise, where noise is added as $\mathbf{h}_l \to \mathbf{h}_l + \sigma_L \mathbf{e}$, where $\mathbf{e}$ is a random vector sampled from the standard normal distribution with the same dimensionality as $\mathbf{h}_l$, the derivative becomes $\frac{\partial \tilde{C}_1}{\partial \sigma_l} = \frac{\partial \tilde{C}_1}{\partial \mathbf{h}_l} \mathbf{e}^{\top}$. It can be shown that the cost of computing this term scales comparably to backpropagation, due to the properties of R and L-operators (Pearlmutter, 1994; Schraudolph, 2002).

For our implementation, the cost of computing the hypergradients of a model with additive noise in each layer, was around 3 times that of backpropagation. We reduced this cost further by making one hyperparameter update per each 10 elementary parameter updates. While it did not change performance of the method, it reduced the additional cost to about only 30% that of backpropagation. The cost could be possibly reduced even further by making hyperparameter updates even less frequently, though we have not explored this further.

3. Experiments

The goal of the experimental section is to address the following questions:

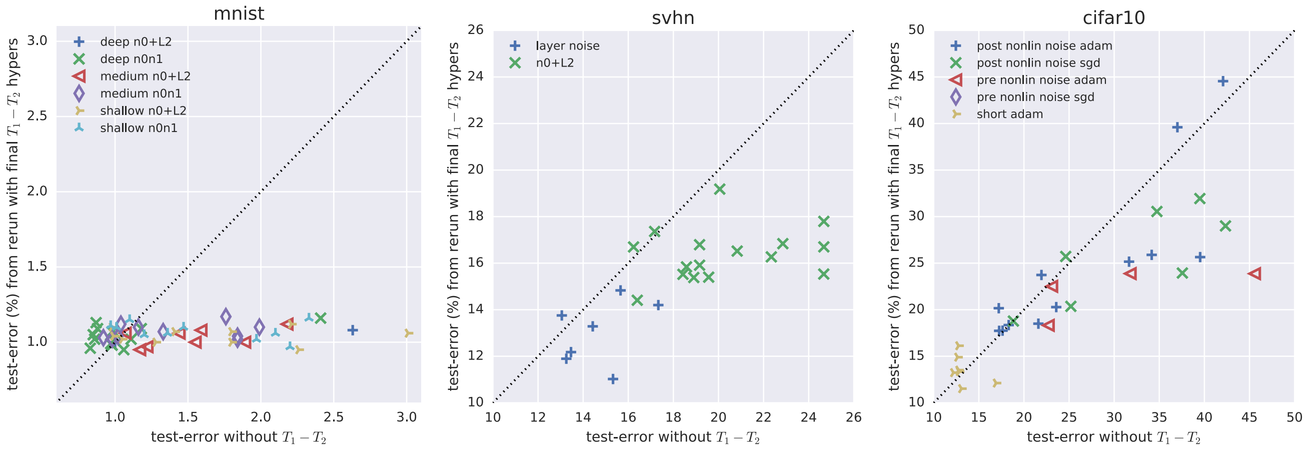

Figure 4. Comparison of test performances when training with fixed hyperparameters before and after tuning them with $T_1 - T_2$. Results for MNIST are shown on the left, SVHN in the middle, CIFAR-10 on the right. Points are generated on a variety of network configurations, where equal symbols mark equal setup.

- Will the method find new hyperparameters which improve the performance of the model, compared to the initial set of hyperparameters?
- Can we observe hysteresis effects, i.e. will the model obtained, while simultaneously modifying parameters and hyperparameters, perform the same as a model trained with a hyperparameter fixed to the final value?
- Can we observe overfitting on the validation set T_2? When hyperparameters are tuned for validation performance, is the performance on the validation set still indicative of the performance on the test set?

We test the method on various configurations of multilayer perceptrons (MLPs) with ReLU activation functions (Dahl *et al.*, 2013) trained on the MNIST (LeCun *et al.*, 1998) and SVHN (Netzer *et al.*, 2011) data set. We also test the method on two convolutional architectures (CNNs) using CIFAR-10 (Krizhevsky, 2009). The CNN architectures used were modified versions of model All-CNN-C from (Springenberg *et al.*, 2014) and a baseline model from (Rasmus *et al.*, 2015), using ReLU and leakyReLU activations (Xu *et al.*, 2015). The models were implemented with the Theano package (Team, 2016). All the code, as well as the exact configurations used in the experiments can be found in the project's Github repository[2].

For MNIST we tried various network sizes: shallow $1000 \times 1000 \times 1000$ to deep $4000 \times 2000 \times 1000 \times 500 \times 250$. Training set T_1 had 55 000 samples, and validation T_2 had 5 000 samples. The split between T_1 and T_2 was made using a different random seed in each of the experiments to avoid overfitting to a particular subset of the training set. Data preprocessing consisted of only centering each feature.

[2]https://github.com/jelennal/t1t2

In experiments with SVHN we tried $2000 \times 2000 \times 2000$ and $4000 \times 2000 \times 1000 \times 500 \times 250$ architectures. Global contrast normalization was used as the only preprocessing step. Out of 73257 training samples, we picked a random 65 000 samples for T_1 and the remaining 8 257 samples for T_2. None of the SVHN experiments used tied hyperparameters, i.e. each layer was parametrized with a separate hyperparameter, which was tuned independently.

To test on CIFAR-10 with convolutional networks, we used 45 000 samples for T_1 and 5 000 samples for T_2. The data was preprocessed using global contrast normalization and whitening.

We regularized the models with additive Gaussian noise to the input with standard deviation n_0 and each hidden layer with standard deviation n_1; or a combination of additive noise to the input layer and L2 penalty with strength multiplier l_2 for weights in each of the layers. Because L2 penalty matters less in models using batch normalization, in experiments using L2 penalty we did not use batch normalization. We tried both tied regularization levels (one hyperparameter for all hidden layers) and having separate regularization parameters for each layer. As a cost function, we use cross-entropy for both T_1 and T_2.

Each of the experiments were run with 200-300 epochs, using batch size 100 for both elementary and hyperparameter training. To speed up elementary parameter training, we use an annealed ADAM learning rate schedule (Kingma and Ba, 2015) with a step size of 10^{-3} (MLPs) or $2 \cdot 10^{-3}$ (CNNs). For tuning noise hyperparameters, we use vanilla gradient descent with a step size 10^{-1}; while for L2 hyperparameters, step sizes were significantly smaller, 10^{-4}. In experiments on larger networks we also use ADAM for tuning hyperparameters, with the step size 10^{-3} for noise and

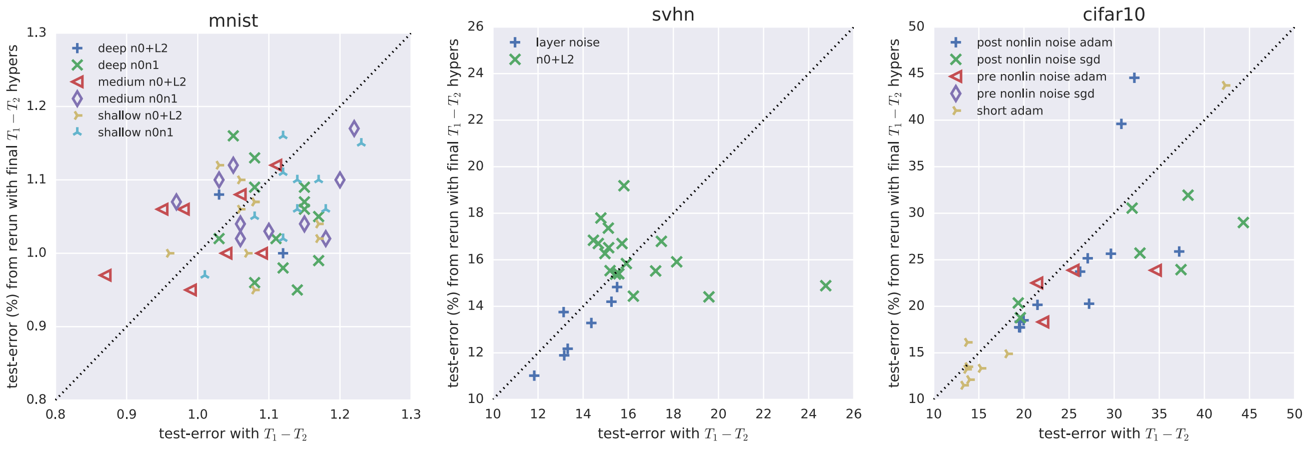

Figure 5. Test error after one run with $T_1 - T_2$ compared to a rerun where we use the final values of the hyperparameters at the end of $T_1 - T_2$ training as fixed hyperparameters for a new run (left: MNIST, middle: SVHN, right: CIFAR-10). Th correlation indicates that $T_1 - T_2$ is useful also for finding approximate hyperparameters for training without an adaptive hyperparameter method.

10^{-6} for L2. We found that while the learning rate did not significantly influence the general area of convergence for a hyperparameter, too high learning rates did cause too noisy and sudden hyperparameter changes, while too low learning rates resulted in no significant changes of hyperparameters. A rule of thumb is to use a learning rate corresponding to the expected order of magnitude of the hyperparameter. Moreover, if the hyperparameter updates are utilized less frequently, the learning rate should be higher.

In most experiments, we first measure the performance of the model trained using some fixed, random hyperparameters sampled from a reasonable interval. Next, we train the model with $T_1 - T_2$ from that random hyperparameter initialization, measuring the final performance. Finally, we rerun the training procedure with the fixed hyperparameter set to the final hyperparameter values found by $T_1 - T_2$. Note that in all the scatter plots, points with the same color indicate the same model configuration: same number of neurons and layers, learning rates, use of batch normalization, and the same types of hyperparameters tuned just with different initializations.

3.1. Results

Figure 1 illustrates resulting hyperparameters changes during $T_1 - T_2$ training. To see how the $T_1 - T_2$ method behaves, we visualized trajectories of hyperparameter values during training in the hyperparameter cost space. For each point in the two-dimensional hyperparameter space, we compute the corresponding test cost without $T_1 - T_2$. In other words, the background of the figures corresponds to grid search on the two-dimensional hyperparameter interval. The initial regularization hyperparameter value is denoted with a star, while the final value is marked with a square.

As can be seen from the figure, all runs converge to a reasonable set of hyperparameters irrespective of the starting value, gradually moving to a point of lower log-likelihood. Note that because the optimal values of learning rates for each hyperparameter direction are unknown, hyperparameters will change the most along directions corresponding to either the local gradient or the higher relative learning rate.

One way to use the proposed method is to tune the hyperparameters, and then rerun the training from the beginning using fixed values for the hyperparameters set to the final values acquired at the end of $T_1 - T_2$ training. Figure 4 illustrates how much $T_1 - T_2$ can improve initial hyperparameters. Each point in the grid corresponds to the test performance of a model fully trained with two different fixed hyperparameters: one is the initial hyperparameter before being tuned with $T_1 - T_2$ (x-axis), the other is final hyperparameter found after tuning the initial hyperparameter with $T_1 - T_2$ (y-axis). As can be seen from the plot, none of the models trained with hyperparameters found by $T_1 - T_2$ performed poorly, regardless of how poor the performance was with the initial hyperparameters.

Next we explore the strength of the hysteresis effect, i.e. how the performance of a model with a different hyperparameter history compares to the performance of a model with a fixed hyperparameter. In Figure 5 we plot the error after a run using $T_1 - T_2$, compared to the test error if the model is rerun with the hyperparameters fixed to the values at the end of $T_1 - T_2$ training. The results indicate that there is a strong correlation, with in most cases, reruns performing somewhat better. The method can be used for training models with fixed hyperparameters, or as a baseline for further hyperparameter finetuning. The hysteresis effect was stronger on CIFAR-10, where retraining produced significant improvements.

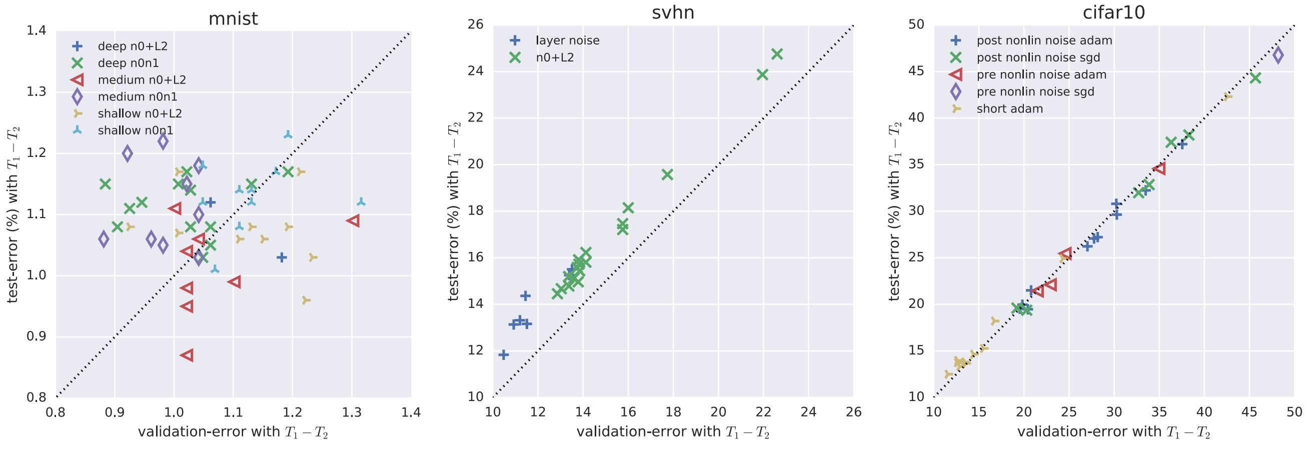

Figure 6. Classification error of validation set vs test set, at the end of $T_1 - T_2$ training for MNIST (left), SVHN (middle), and CIFAR-10 (right). For MNIST there is no apparent structure, but all the results lie in a region of low error. The results for the other two datasets correlate strongly, suggesting that validation set performance is still indicative of test set performance.

We explore the possibility of overfitting on the validation set. Figure 6 (right) shows the validation error compared to the final test error of a model trained with $T_1 - T_2$. We do not observe overfitting, with validation performance being strongly indicative of the test set performance. For MNIST all the results cluster tightly in the region of low error, hence there is no apparent structure. It should be noted though, that in these experiments we had at most 20 hyperparameters, making overfitting to validation set unlikely.

4. Discussion and Conclusion

We have proposed a method called $T_1 - T_2$ for gradient-based automatic tuning of continuous hyperparameters during training, based on the performance of the model on a separate validation set. We experimented on tuning regularization hyperparameters when training different model structures on the MNIST and SVHN datasets.The $T_1 - T_2$ model consistently managed to improve on the initial levels of additive noise and L2 weight penalty. The improvement was most pronounced when the initial guess of the regularization hyperparameter values was far from the optimal value.

Although $T_1 - T_2$ is unlikely to find the best set of hyperparameters compared to an exhaustive search where the model is trained repeatedly with a large number of hyperparameter proposals, the property that it seems to find values fairly close to the optimum is useful e.g. in situations where the user does not have a prior knowledge on good intervals for regularization selection; or the time to explore the full hyperparameter space. The method could also be used in combination with random search, redirecting the random hyperparameter initializations to better regions.

While the $T_1 - T_2$ method is helpful for minimizing the objective function on the validation set, as illustrated in Figure 7, a set of hyperparameters minimizing a continuous objective like cross-entropy, might not be optimal for the classification error. It may be worthwhile to try objective functions which approximate the classification error better, as well as trying the method on unsupervised objectives.

As a separate validation set is used for tuning of hyperparameters, it could be possible to overfit to the validation set. However, our experiments indicated that this effect is not practically significant in the settings tested in this paper, which had at most 10-20 hyperparameters.

The method could be used to tune a much larger number of hyperparameters than what was computationally feasible before. It could also be used to tune hyperparameters other than continuous regularization hyperparameters, using continuous versions of those hyperparameters. For example, consider the following implementation of a continuously parametrized number of layers: the final softmax layer takes input from all the hidden layers, however the contribution of each layer is weighted with a continuous function $a(i)$, such that one layer and its neighboring layers contribute the most, e.g. $output = softmax[\sum_{i=1..L} a(i)\tilde{W}_i\mathbf{h}_i]$, where L is the number of layers and $a(i) = e^{(i-m)^2/v}$. Which layer contributes the most to the output layer, is determined with a differentiable function, and parameters of this function, m and v in this example, could in principle be trained using the $T1 - T2$ method.

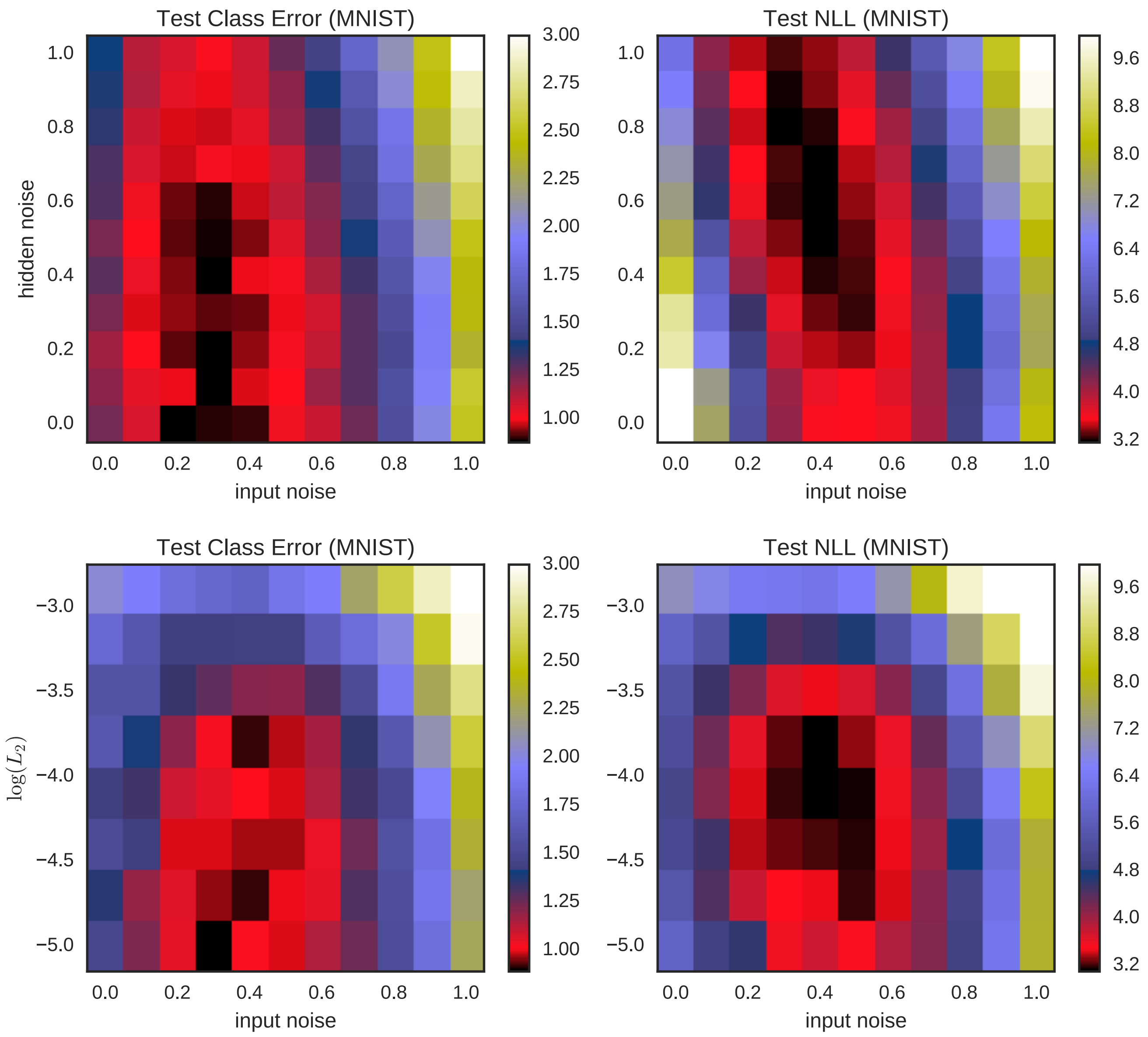

Figure 7. Grid search results on a pair of hyperparameters (no tuning with $T_1 - T_2$). Figures on the right represent the test error at the end of training as a function of hyperparameters. Figures on the left represent the test log-likelihood at the end of training as a function of hyperparameters. We can see that the set of hyperparameters minimizing test log-likelihood is different from the set of hyperparameters minimizing test classification error.

Acknowledgements

We are very thankful to many colleagues for the helpful conversations and feedback, particularly Dario Amodei and Harri Valpola. Special thanks to Antti Rasmus for the technical assistance. Also thanks to Olof Mogren and Mikael Kageback, who provided detailed comments on the paper draft. Jelena Luketina and Tapani Raiko were funded by the Academy of Finland. Mathias Berglund was funded by the HICT doctoral education network.

References

Bengio, Y. (2000). Gradient-based optimization of hyperparameters. *Neural computation*, **12**(8), 1889–1900.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, **13**, 281–305.

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Ad-*

vances in Neural Information Processing Systems 24, pages 2546–2554.

Chen, D. and Hagan, M. T. (1999). Optimal use of regularization and cross-validation in neural network modeling. In *International Joint Conference on Neural Networks*, pages 1275–1289.

Dahl, G. E., Sainath, T. N., and Hinton, G. E. (2013). Improving deep neural networks for LVCSR using rectified linear units and dropout. In *ICASSP*, pages 8609–8613.

Desjardins, G., Simonyan, K., Pascanu, R., and Kavukcuoglu, K. (2015). Natural neural networks. In *Advances in Neural Information Processing Systems*.

Foo, C.-s., Do, C. B., and Ng, A. (2008). Efficient multiple hyperparameter learning for log-linear models. In *Advances in neural information processing systems (NIPS)*, pages 377–384.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Kingma, D. and Ba, J. (2015). Adam: A method for stochastic optimization. In *the International Conference on Learning Representations (ICLR)*, San Diego. arXiv:1412.6980.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images.

Larsen, J., Svarer, C., Andersen, L. N., and Hansen, L. K. (1998). Adaptive regularization in neural network modeling. In *Neural Networks: Tricks of the Trade*, pages 113–132. Springer.

LeCun, Y., Cortes, C., and Burges, C. J. (1998). The MNIST database of handwritten digits.

Maclaurin, D., Duvenaud, D., and Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 4. Granada, Spain.

Pearlmutter, B. A. (1994). Fast Exact Multiplication by the Hessian. *Neural Computation*, pages 147–160.

Raiko, T., Valpola, H., and LeCun, Y. (2012). Deep learning made easier by linear transformations in perceptrons. In *International Conference on Artificial Intelligence and Statistics*, pages 924–932.

Rasmus, A., Valpola, H., Honkala, M., Berglund, M., and Raiko, T. (2015). Semi-supervised learning with ladder network. *Neural Information Processing Systems*.

Schraudolph, N. N. (2002). Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, **14**(7), 1723–1738.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian Optimization of Machine Learning Algorithms. *ArXiv e-prints*.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for Simplicity: The All Convolutional Net.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, **15**(1), 1929–1958.

Team, T. T. D. (2016). Theano: A Python framework for fast computation of mathematical expressions.

Vatanen, T., Raiko, T., Valpola, H., and LeCun, Y. (2013). Pushing stochastic gradient towards second-order methods–backpropagation learning with transformations in nonlinearities. In *Neural Information Processing*, pages 442–449. Springer.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and antoine Manzagol, P. (2010). Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*.

Wang, S. I. and Manning, C. D. (2013). Fast dropout training. In *In Proceedings of the 30th International Conference on Machine Learning (ICML)*.

Xu, B., Wang, N., and Li, M. (2015). Empirical evaluation of rectified activations in convolutional network. *CoRR*.