SCALABLE GRADIENT-BASED TUNING OF CONTINUOUS REGULARIZATION HYPERPARAMETERS

Jelena Luketina, Mathias Berglund, Tapani Raiko
Department of Computer Science
Aalto University, Finland
{name.lastname}@aalto.fi

ABSTRACT

Hyperparameter selection generally relies on running multiple full training trials, with hyperparameter 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 the resulting regularization levels are within the optimal regions. The method is significantly less computationally demanding compared to similar gradient-based approaches to hyperparameter optimization and consistently finds good hyperparameter values, which makes it a useful tool for training neural network models.

1 INTRODUCTION

Specifying and training neural networks models requires several design choices that are often not trivial to make. Many of these design choices boil down to selection of hyperparameters. The process of hyperparameter selection is in practice often based on trial-and-error or search with a grid or by randomly sampling from an initial guess on possible hyperparameter values (Bergstra and Bengio, 2012). There is 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 decent hyperparameter values in one training run even if the user did not have a strong intuition on good values to try for the hyperparameters.

In contrast to these methods, we consider treating hyperparameters as elementary[1] parameters during training, and 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 late 1990s; however, these methods either require computation of inverse Hessian (Larsen *et al.*, 1998; Bengio, 2000; Chen and Hagan, 1999; Foo and Ng, 2008), or propagating updates through the whole history of elementary parameter updates Maclaurin *et al.* (2015). Moreover, these methods make changes to the hyperparameter only once

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

the elementary parameter training has ended. This makes them too expensive for the 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 computation of Hessian or inverting the Hessian. This is because with $T_1 - T_2$ 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 ignored. The resulting additional computational and memory cost 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. Adding 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, hyperparameter needs 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 changes to the hyperparameter are slow enough, it would be possible to train the model 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 expect 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 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.

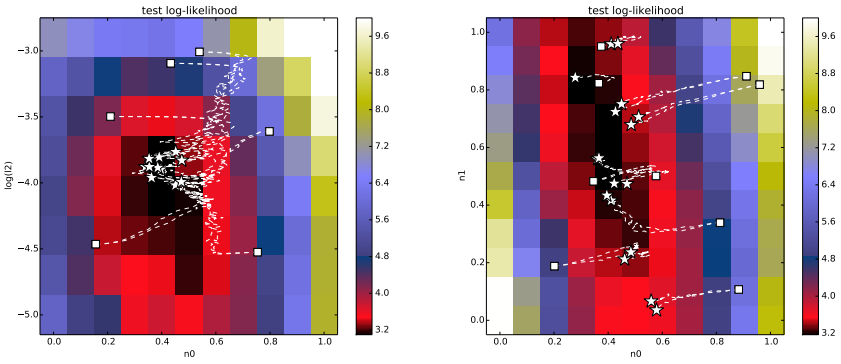

Figure 1: Left: Values of input additive noise and L_2 penalty $(n_0, log(l_2))$, during training using $T_1 - T_2$ method for hyperparameter tuning. Trajectories are plotted over the grid-search result for the same regularization pair. Initial hyperparameter values are denoted with a square, final hyperparameter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

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 $F_1(\cdot)$ and the corresponding weight update as:

$$F_1(W|\lambda, T_1) = C_1(W|\lambda, T_1) + P(W, \lambda), \tag{1}$$

$$W_{t+1} = W_t + \eta_1 \nabla_W F_1(W_t|\lambda_t, T_1), \tag{2}$$

where $C_1(\cdot)$ and $P(\cdot)$ are cost and regularization penalty terms, $T_1 = \{(x_i,\ y_i)\}$ is the training data set, $W = \{W^l,\ b^l\}$ a set of elementary parameters including weights and biases of each layer, λ denotes various hyperparameters that determine the strength of regularization, while η_1 is a learning rate. Subscript t refers to the iteration number.

Assuming $T_2 = \{(x_i,\ y_i)\}$ is a separate validation data set, the generalization performance of the model is measured with a validation cost $C_2(W_{t+1}, T_2)$, 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 λ is:

$$\nabla_\lambda C_2 = \nabla C_2 \frac{\partial W_{t+1}}{\partial \lambda_t}$$

We only consider the influence of the regularization hyperparameter on the current elementary parameter update, $\frac{\partial W_{t+1}}{\partial \lambda} = \eta_1 \frac{\partial^2 F_1}{\partial \lambda \partial W}$ based on Eq. (2). The hyperparameter update is therefore:

$$\lambda_{t+1} = \lambda_t + \eta_2 \nabla_W C_2 \frac{\partial^2 F_1}{\partial \lambda \partial W} \tag{3}$$

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 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 gradient and the hypergradient significantly more expensive.

A well justified closed form of the hypergradient is available once the elementary gradient has converged (Foo and Ng, 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 $\nabla^2_W F_1 = I$:

$$\lambda_{t+1} = \lambda_t + \nabla_W C_2 (\nabla^2_W F_1)^{-1} \frac{\partial^2 F_1}{\partial \lambda \partial W}. \tag{4}$$

Another approach to hypergradient computation is given in Maclaurin *et al.* (2015). There, the term $\frac{\partial W_T}{\partial \lambda}$ (T denoting final iteration number) considers effect of the hyperparameter on the entire history of updates $W_T = \sum_{0<k<T} \triangle W_{k,k+1}((W_k(\lambda),\ \lambda,\ T_k,\ \eta_k) + W_0$. Update of a hyperparameter is formed by collecting the elements from the entire training procedure:

$$\lambda_{t+1} = \lambda_t + \nabla_W C_2 \frac{d}{d\lambda} \sum_{0<k<T} \triangle W_{k,k+1}(W_k(\lambda_t),\ \lambda_t,\ T_k,\ \eta_k) + W_0 \tag{5}$$

$$\lambda_{t+1} = \lambda_t + \sum_{0<k<T} g'_t[\nabla^2_W F_{1,k}(\frac{\partial W_k}{\partial \lambda} + \frac{\partial \nabla_W F_{1,k}}{\partial \lambda}) + \nabla^2_W F_{1,k-1}(\frac{\partial W_{k-1}}{\partial \lambda} + \frac{\partial \nabla_W F_{1,k-1}}{\partial \lambda})]. \tag{6}$$

Eq. (3) can therefore be considered as an approximation of (6), 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 leads into difficulties. From Equation (3), it follows that the method converges when $\nabla_W C_2 \frac{\partial^2 F_1}{\partial \lambda \partial W} = 0$, or in other words, for all components i of the hyperparameter vector λ, $\nabla_W C_2$ is orthogonal to $\frac{\partial^2 F_1}{\partial \lambda_i \partial W}$. 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 $W_{j,t}$, the point of convergence could change.

It is clear that the identity Hessian assumption is an approximation that will not hold exactly. 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

In terms of computational complexity, the most expensive term is $\nabla_W C_2 \frac{\partial^2 F_1}{\partial \lambda \partial W}$, with the exact complexity depending on the details of implementation and the hyperparameters. One cheap implementation is through finite difference method, which is by Clairaut's theorem on equality of mixed partial derivatives:

$$\nabla_W C_2 \cdot \nabla_\lambda \nabla_W F_1 = \nabla_W C_2 \cdot \nabla_W \nabla_\lambda F_1 \approx \frac{\nabla_\lambda F_1(W + \epsilon \nabla_W C_2) - \nabla_\lambda F_1(W)}{\epsilon}. \tag{7}$$

Gradient of cost with respect to noise hyperparameter at layer i, σ_i, can be computed as $\frac{\partial F_1}{\partial \sigma_i} = \frac{\partial F_1}{\partial h_i} \frac{\partial h_i}{\partial \sigma_i}$, where h_i is hidden layer i activation. In case of additive Gaussian noise, where noise is added as $h_i \rightarrow h_i + \sigma_i E$, where E is a random matrix sampled from the standard normal distribution with the same dimensionality as h_i, the derivative becomes $\frac{\partial F_1}{\partial \sigma_i} = \frac{\partial F_1}{\partial h_i} E$. Hence equation (7) requires one additional feedforward and feedback pass per batch, to compute $F_1(W + \epsilon g_2)$ and $\frac{\partial F_1(W+\epsilon g_2)}{\partial h_i}$. The exact computation of the term also scales comparably to backpropagation (Pearlmutter, 1994; Schraudolph, 2002). In our experiments, cost of computing exact hypergradients in this setting was at most 3 times that of backpropagation. Furthermore, cost of computing exact hypergradients for $L2$ regularizers $P(W) = \sum_k \lambda_k w_k^2$ is negligible, since $\frac{\partial^2 F_1}{\partial \lambda_k \partial w_l} = w_k \delta_{k,l}$.

3 EXPERIMENTS

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

- 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?

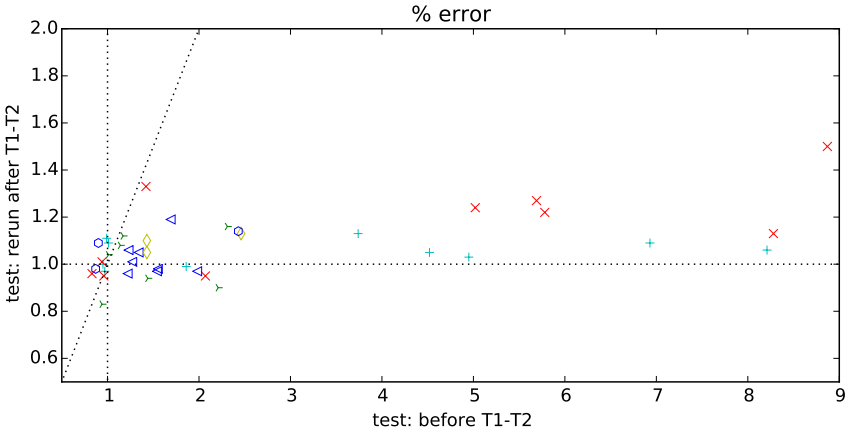

Figure 2: Comparison of test performances before and after tuning the hyperparameters with $T_1 - T_2$. Points are generated on a variety of network configurations.

We test the method on the various configurations of multilayer perceptrons (MLPs) with ReLU activation functions (Dahl *et al.*, 2013), trained on the MNIST data set. We tried networks up to the depth of 8 layers, with the layer breadth between 100 and 1000 neurons. 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 biasing results towards a particular subset of the training set. Data preprocessing consisted of only centering each feature. The model was implemented with the Theano package (Bastien *et al.*, 2012; Bergstra *et al.*, 2010).

Model complexity was further limited by 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 runs had 150 epochs, using the batch size 100 for both elementary and hyperparameter training. To speed up elementary parameter training, we use the annealed ADAM learning rate schedule (Kingma and Ba, 2015); with the basic step size of 0.001. For tuning noise hyperparameters, we use step sizes $[0.01, 0.001]$; while for L2 hyperparameters, optimal step sizes were significantly smaller, $[10^{-4}, 10^{-6}]$. 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. General rule of thumb is to use larger learning rates if there is no prior knowledge of the optimal hyperparameter values and hence starting with the hyperparameters set to zero; or if one suspects initial hyperparameters are far from the optimal values.

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 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.

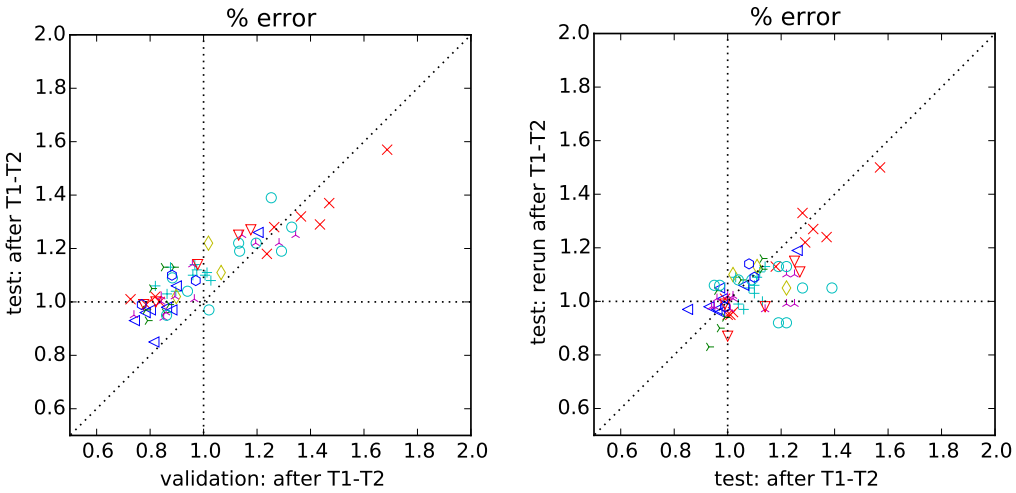

Figure 3: Left: Classification error of validation set T_2 and test set, at the end of $T_1 - T_2$ training. The results correlate strongly. While we can observe weak overfitting on T_2, it has not significantly hampered performance. Right: Test error after one run with $T_1 - T_2$ compared to a rerun 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. The results indicate that there is a strong correlation, which renders $T_1 - T_2$ useful also for finding approximate hyperparameter values for training without an adaptive hyperparameter method.

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. 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 the point with lower log-likelihood. Note that because the optimal values of learning rates for each hyperparameter direction are unknown, hyperparameter will change the most along the direction 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 start using the final values of the hyperparameters. Figure 2 illustrates how much can $T_1 - T_2$ 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 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 was the performance with the initial hyperparameters.

Method	Test error
Dropout (Srivastava *et al.*, 2014)	1.25 %
Averaged $T_1 - T_2$	1.04 % $\pm$ .11
Additive noise + L2 (Raiko *et al.*, 2012)	1.03 %
Dropout + max-norm (Srivastava *et al.*, 2014)	0.94 %
Additive noise + BN (Rasmus *et al.*, 2015)	0.80 %

Table 1: A collection of previously reported MNIST test errors in permutation-invariant setting. Averaged performance of various models (varying in number of layers, hidden units, regularization type) with regularization hyperparameters tuned using $T_1 - T_2$ is comparable.

Next we explore the strength of the hysteresis effect, i.e. how does the performance of a model with a different hyperparameter history compare to the performance of a model with a fixed hyperparameter? Is the performance somehow dependent on the regularization schedule? In Figure 3 (left) 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 therefore be used for training models with fixed hyperparameters, or as a baseline for further hyperparameter finetuning. In fact, retraining the model with the final hyperparameter fixed is likely to produce better results.

We explore the possibility of overfitting on the validation set. Figure 3 (right) shows the validation error compared to the final test error of a model trained with $T_1 - T_2$. While the results indicate some overfitting with the validation set performing mostly better, the two are still strongly correlated. Better validation performance strongly indicates better test performance. It should be noted though, that in these experiments we had at most 20 hyperparameters, making overfitting to validation set unlikely.

Finally we can compare performance of models tuned with $T_1 - T_2$, to some other purely supervised models on permutation invariant MNIST in Table 1. Without doing throughout hyperparameter search of learning rates or network configurations, just rerunning the models with the hyperparameters found by $T_1 - T_2$, on average we get comparable performance.

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 dataset.The $T_1 - T_2$ model always managed to find levels of additive noise and L2 weight penalty that yielded decent test set performance even if the initial guess of the regularization hyperparameter values was orders of magnitudes 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; it is significantly less computationally demanding and capable of quickly finding values fairly close to the optimum. This is useful 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.

While the $T_1 - T_2$ method does a decent job at minimizing the objective function of validation set, as illustrated in 4, hyperparameters minimizing a continuous objective like cross-entropy, might not be optimal for the classification error. It might be worthwhile trying objectives 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 is in theory possible to overfit to the validation set. However, our experiments indicated that this effect is not practically very significant in the settings tested in this paper, which is at most 10-20 hyperparameters.

The encouraging results open a number of fruitful avenues for further research. While $T_1 - T_2$ is computationally cheap compared to other methods used for hyperparameter selection, there is poten-

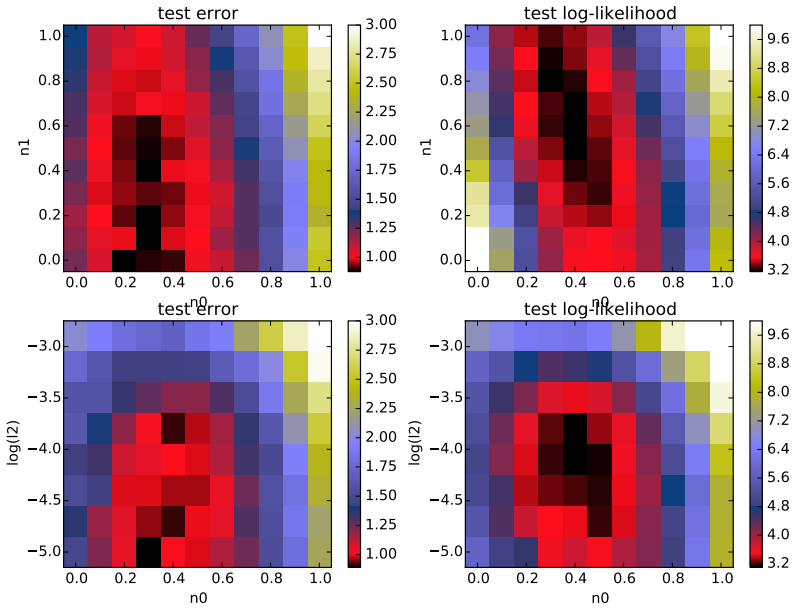

Figure 4: Grid search results on a pair of hyperparameters (no tuning with $T_1 - T_2$). Figures on the right, represent test error at the end of training as a function of hyperparameters. Figures on the left represent test log-likelihood at the end of training as a function of training. Note that the set of hyperparameters minimizing test log-likelihood is different from the set of hyperparameters minimizing test classification error.

tial to speed it up even further by updating the hyperparameters less frequently than the elementary parameters.

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 other hyperparameters than continous regularization hyperparameters: E.g. the number of layers and units could be tuned using continuous versions of those hyperparameters. In addition, $T_1 - T_2$ could be tested for other continuous hyperparameters.

REFERENCES

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., Warde-Farley, D., and Bengio, Y. (2012). Theano: new features and speed improvements. *CoRR*, **abs/1211.5590**.

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., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*.

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

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.

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

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*.

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.

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*.