Self-Tuning Networks

Matthew MacKay*, Paul Vicol*, Jon Lorraine, David Duvenaud, Roger Grosse

UNIVERSITY OF TORONTO

VECTOR INSTITUTE

Motivation

- *Regularization hyperparameters* such as weight decay, dropout, and data augmentation are crucial for neural net generalization but are *difficult to tune*

- Automatic approaches for hyperparameter optimization have the potential to:
    - *Speed up hyperparameter search* and save researcher time
    - Discover *solutions* that outperform manually-designed ones
    - Make ML more *accessible to non-experts* (e.g., chemists, biologists, physicists)

- We introduce an efficient, *gradient-based approach to adapt regularization hyperparameters during training*
    - Easy-to-implement, memory-efficient, and outperforms competing methods

Bilevel Optimization

- Hyperparameter optimization is a *bilevel optimization problem*:

$$\lambda^* = \arg\min_{\lambda} \mathcal{L}_{\text{val}}(\lambda, \mathbf{w}^*) \qquad \text{subject to} \qquad \mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}_{\text{train}}(\lambda, \mathbf{w})$$

Outer loop over hyperparameters

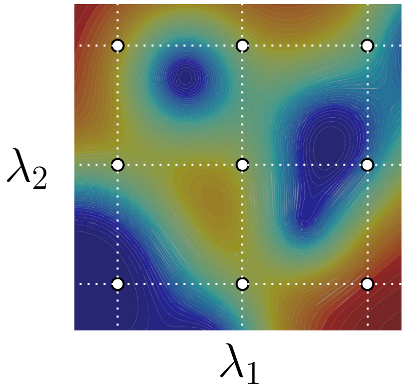
```
while True:
    hparam = get_hyperparameter_value()
    W = init_weights()

    while not converged:
        W = gradient_step(W, hparam)
```

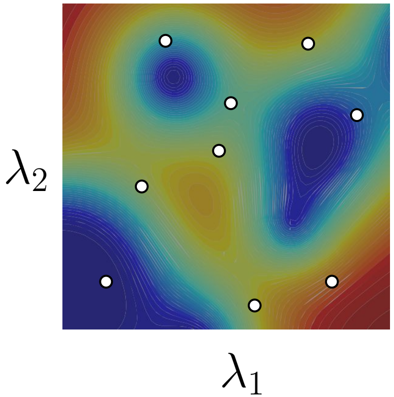
Inner loop to optimize model parameters

Grid Search, Random Search, & BayesOpt

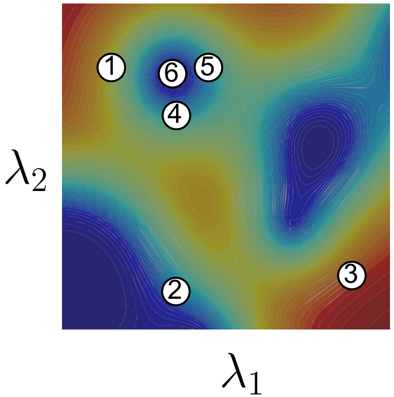
Grid Search



Random Search



Bayesian Optimization



- Many approaches treat the outer optimization over λ as a *black-box problem*
  - Ignores structure that could be used for faster convergence
- These approaches *re-train models from scratch* to evaluate each new hyperparameter
  - Wastes computation!

Approximating the *Best-Response Function*

- The *"best-response" function* maps hyperparameters to optimal weights on the training set:

$$\mathbf{w}^*(\lambda) = \arg\min_{\mathbf{w}} \mathcal{L}_{\text{train}}(\lambda, \mathbf{w})$$

- **Idea:** *Learn a parametric approximation* $\hat{\mathbf{w}}_\phi$ to the best-response function $\hat{\mathbf{w}}_\phi \approx \mathbf{w}^*$

- **Advantages:**
    - Since $\hat{\mathbf{w}}_\phi$ is differentiable, we can use *gradient-based optimization* to update the hyperparameters
    - By training $\hat{\mathbf{w}}_\phi$ we *do not need to re-train models from scratch*; the computational effort needed to fit $\hat{\mathbf{w}}_\phi$ around each hyperparameter is not wasted

Approximating the *Best-Response Function*

- Update the approximation parameters ϕ using the *chain rule*:

$$\frac{\partial \mathcal{L}_{\text{train}}(\hat{\mathbf{w}}_\phi)}{\partial \hat{\mathbf{w}}_\phi} \frac{\partial \hat{\mathbf{w}}_\phi}{\partial \phi}$$

- Update the hyperparameters using the *validation loss gradient*:

$$\frac{\partial \mathcal{L}_{\text{val}}(\hat{\mathbf{w}}_\phi(\lambda))}{\partial \hat{\mathbf{w}}_\phi(\lambda)} \frac{\partial \hat{\mathbf{w}}_\phi(\lambda)}{\partial \lambda}$$

Globally Approximating the Best-Response

Global Best-Response Approximation

initialize ϕ
initialize $\hat{\lambda}$
loop
    $x \sim \mathcal{D}_{train}$
    $\lambda \sim p(\lambda)$
    $\phi \mathrel{-}= \alpha \nabla_{\phi} \mathcal{L}_{train}(w_{\phi}(\lambda), \lambda, x)$
loop
    $x \sim \mathcal{D}_{val}$
    $\hat{\lambda} \mathrel{-}= \beta \nabla_{\hat{\lambda}} \mathcal{L}_{val}(w_{\phi}(\hat{\lambda}), x)$
Return $\hat{\lambda}, w_{\phi}(\hat{\lambda})$

Train the hypernetwork to produce good weights for any hyperparameter $\lambda \sim p(\lambda)$

Find the optimal hyperparameters via gradient descent on $\mathcal{L}_{\text{val}}$

Lorraine and Duvenaud. *Stochastic Hyperparameter Optimization through Hypernetworks*. 2018

Scalability Challenges

- *Two core challenges to scale this approach* to large networks:

1. Intractable to model $\hat{\mathbf{w}}_\phi(\lambda)$ *over the entire hyperparameter space*, e.g., the support of $p(\lambda)$

   ➡ **Solution:** Approximate the best-response *locally* in a neighborhood around the current hyperparameter value

2. Difficult to learn a mapping $\lambda \rightarrow \mathbf{w}$ when $\mathbf{w}$ are the weights of a large network

   ➡ **Solution:** STNs introduce a *compact* approximation to the best-response by *modulating activations based on the hyperparameters*

Locally Approximating the Best-Response

- *Jointly optimize* the hypernetwork parameters and the hyperparameters by *alternating gradient steps on the training and validation sets*

Local Best-Response Approximation

initialize ϕ

initialize $\hat{\lambda}$

loop

$$x \sim \mathcal{D}_{train}$$
$$\lambda \sim p(\lambda \mid \hat{\lambda})$$
$$\phi \mathrel{-}= \alpha \nabla_\phi \mathcal{L}_{train}(w_\phi(\lambda), \lambda, x)$$

Train the hypernet to produce good weights *around the current hyperparameter* $\lambda \sim p(\lambda \mid \hat{\lambda})$

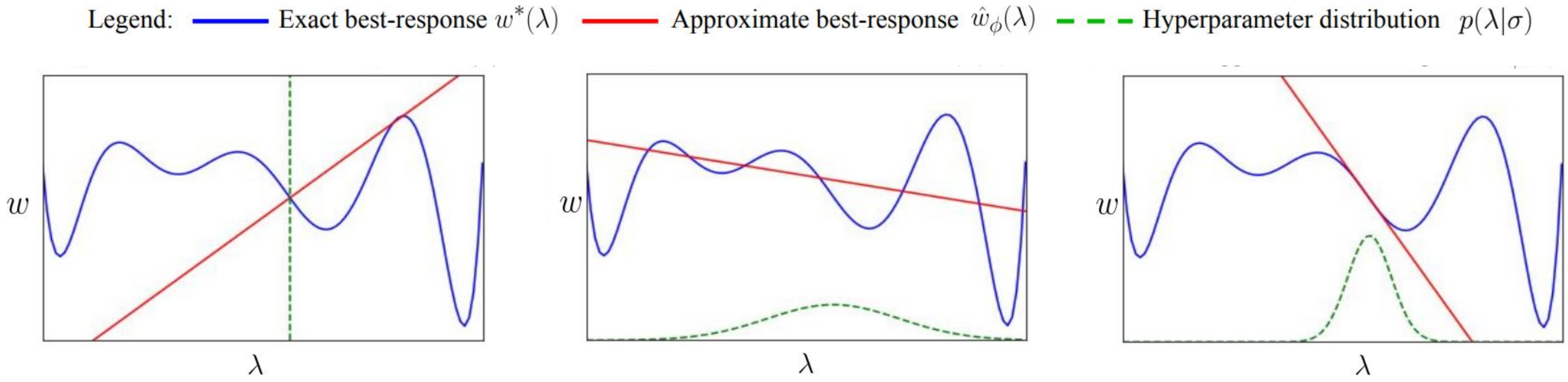
$$x \sim \mathcal{D}_{val}$$
$$\hat{\lambda} \mathrel{-}= \beta \nabla_{\hat{\lambda}} \mathcal{L}_{val}(w_\phi(\hat{\lambda}), x)$$

Update the hyperparameters using the local best-response approximation

Return $\hat{\lambda}, w_\phi(\hat{\lambda})$

Lorraine and Duvenaud. *Stochastic Hyperparameter Optimization through Hypernetworks*. 2018

Effect of the Sampling Distribution



Too small	*Too wide*	*Just right*
The hypernetwork will match the best-response at the current hyperparameter, but may not be locally correct	The hypernetwork may be insufficiently flexible to model the best-response, and the gradients will not match	The gradient of the approximation will match that of the best-response

MacKay et al. *Self-Tuning Networks*. 2019.

Adjusting the Hyperparameter Distribution

- As the smoothness of the loss landscape changes during training, it may be beneficial to *vary the scale* of the hyperparameter distribution, σ

- We adjust σ based on the sensitivity of the validation loss on the sampled hyperparameters, via an *entropy term*:

$$\mathbb{E}_{\epsilon \sim p(\epsilon|\sigma)}[\mathcal{L}_{\mathrm{val}}(\lambda + \epsilon, \hat{\mathbf{w}}_{\phi}(\lambda + \epsilon))] - \tau \mathbb{H}[p(\epsilon \mid \sigma)]$$

MacKay et al. *Self-Tuning Networks*. 2019.

Compact Best-Response Approximation

- Naively representing the mapping $\lambda \rightarrow \mathbf{w}$ is intractable when $\mathbf{w}$ is high-dimensional

- We propose an architecture that computes the usual elementary weight/bias, plus an additional weight/bias that is scaled by a linear transformation of the hyperparameters:

$$\hat{\mathbf{W}}_{\phi}(\lambda) = \mathbf{W}_{\text{elem}} + (\mathbf{V}\lambda) \odot_{\text{row}} \mathbf{W}_{\text{hyper}}$$

$$\hat{\boldsymbol{b}}_{\phi}(\lambda) = \boldsymbol{b}_{\text{elem}} + (\mathbf{C}\lambda) \odot \boldsymbol{b}_{\text{hyper}}$$

- *Memory-efficient*: roughly 2x number of parameters and scales well to high dimensions

MacKay et al. *Self-Tuning Networks*. 2019.

Compact Best-Response Approximation

- This architecture can be interpreted as *directly operating on the pre-activations of the layer*, and *adding a correction to account for the hyperparameters*:

$$\hat{\mathbf{W}}_\phi(\lambda)\boldsymbol{x} + \hat{\boldsymbol{b}}_\phi(\lambda) = \underbrace{[\mathbf{W}_{\text{elem}}\boldsymbol{x} + \boldsymbol{b}_{\text{elem}}]}_{\text{Usual computation of Linear layer}} + \underbrace{[(\mathbf{V}\lambda) \odot_{\text{row}} (\mathbf{W}_{\text{hyper}}\boldsymbol{x}) + (\mathbf{C}\lambda \odot \boldsymbol{b}_{\text{hyper}})]}_{\text{Correction term to account for the hyperparameters}}$$

- **Sample-efficient:** since the predictions can be computed by transforming pre-activations, the *hyperparameters for different examples in a mini-batch can be perturbed independently*
    - E.g., a different dropout rate for each example

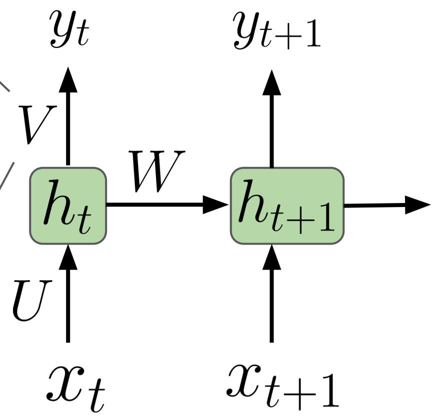
MacKay et al. *Self-Tuning Networks*. 2019.

STN Implementation

```
class HyperLinear(nn.Module):
    def __init__(self, in_dim, out_dim, n_hparams):
        super(HyperLinear, self).__init__()
        self.elem_w = nn.Parameter(torch.Tensor(out_dim, in_dim))
        self.elem_b = nn.Parameter(torch.Tensor(out_dim))
        self.hnet_w = nn.Parameter(torch.Tensor(out_dim, in_dim))
        self.hnet_b = nn.Parameter(torch.Tensor(out_dim))
        self.h_to_scalars = nn.Linear(n_hparams, out_dim*2, bias=False)

    def forward(self, input, hparam_tensor):
        output = F.linear(input, self.elem_w, self.elem_b)
        hnet_scalars = self.h_to_scalars(hparam_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
```

Use HyperLinear layer as a *drop-in replacement* for Linear layers → *build a HyperLSTM*



MacKay et al. *Self-Tuning Networks*. 2019.

STN Algorithm

Algorithm 1 STN Training Algorithm

Initialize: Best-response approximation parameters ϕ, hyperparameters $\boldsymbol{\lambda}$, learning rates $\{\alpha_i\}_{i=1}^3$

while not converged **do**

  for $t = 1, \ldots, T_{train}$ **do**

    $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$

    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \alpha_1 \frac{\partial}{\partial \boldsymbol{\phi}} f(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon}))$

  for $t = 1, \ldots, T_{valid}$ **do**

    $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$

    $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \alpha_2 \frac{\partial}{\partial \boldsymbol{\lambda}} (F(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon})) - \tau \mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})])$

    $\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma} - \alpha_3 \frac{\partial}{\partial \boldsymbol{\sigma}} (F(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon})) - \tau \mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})])$

MacKay et al. *Self-Tuning Networks*. 2019.

STN Algorithm

Optimization step on the *training set*

```
batch_htensor = perturb(htensor, hscale)
hparam_tensor = hparam_transform(batch_htensor)
images, labels = next_batch(train_dataset)
pred = hyper_model(images, batch_htensor, hparam_tensor)
loss = F.cross_entropy(pred, labels)
loss.backward()
optimizer.step()
```

Optimization step on the *validation set*

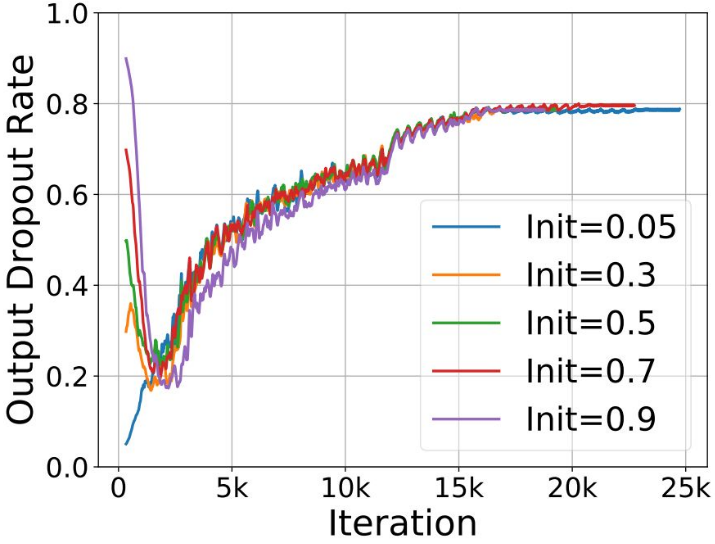
```
batch_htensor = perturb(htensor, hscale)
hparam_tensor = hparam_transform(batch_htensor)
images, labels = next_batch(val_dataset)
pred = hyper_model(images, batch_htensor, hparam_tensor)
xentropy_loss = F.cross_entropy(pred, labels)
entropy = compute_entropy(hscale)
loss = xentropy_loss - args.entropy_weight * entropy
loss.backward()
hyper_optimizer.step()
scale_optimizer.step()
```

MacKay et al. *Self-Tuning Networks*. 2019.

STN Hyperparameter Schedules

- Due to joint optimization of the hypernetwork and hyperparameters, STNs do not use fixed hyperparameter values throughout training
  - STNs discover *hyperparameter schedules which can outperform fixed hyperparameters*
- The same trajectory is followed *regardless of the initial hyperparameter value*

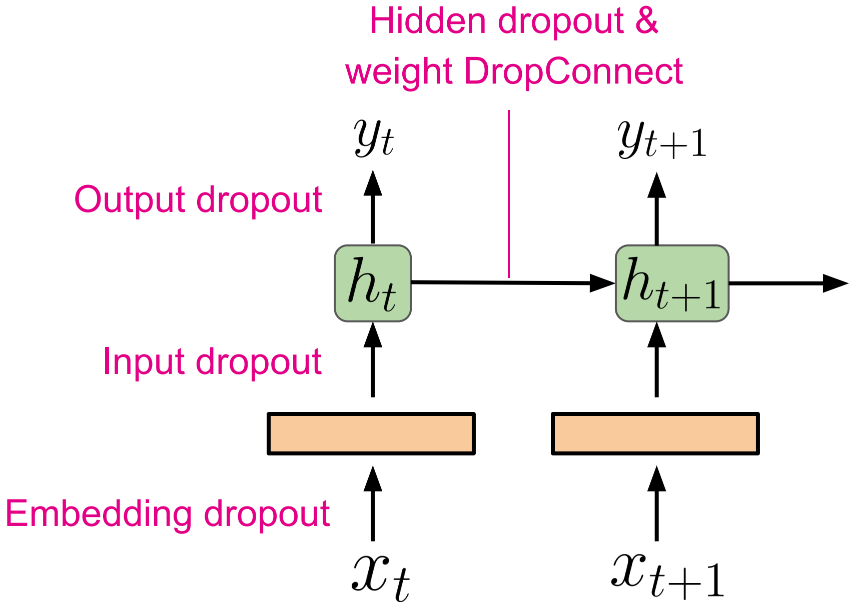
Method	Val	Test
$p = 0.68$, Fixed	85.83	83.19
$p = 0.68$ w/ Gaussian Noise	85.87	82.29
$p = 0.68$ w/ Sinusoid Noise	85.29	82.15
$p = 0.78$ (Final STN Value)	89.65	86.90
STN	**82.58**	**79.02**
LSTM w/ STN Schedule	82.87	79.93



MacKay et al. *Self-Tuning Networks*. 2019.

STN - LSTM Experiment Setup

- **Experiment:** LSTM on Penn TreeBank (a common benchmark for RNN regularization)
- 7 hyperparameters:



Activation Regularization

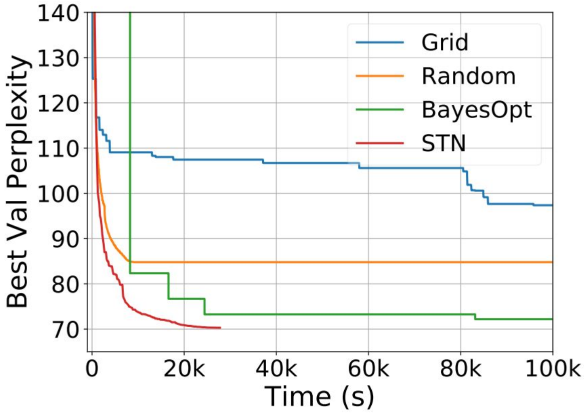
$$\alpha||m \odot h_t||_2$$

Temporal Activation Regularization

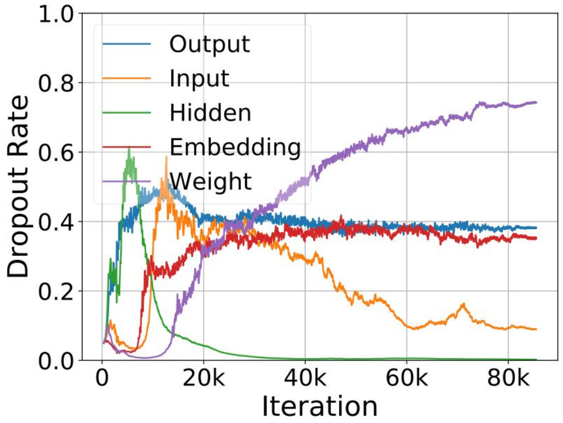
$$\beta||h_t - h_{t+1}||_2$$

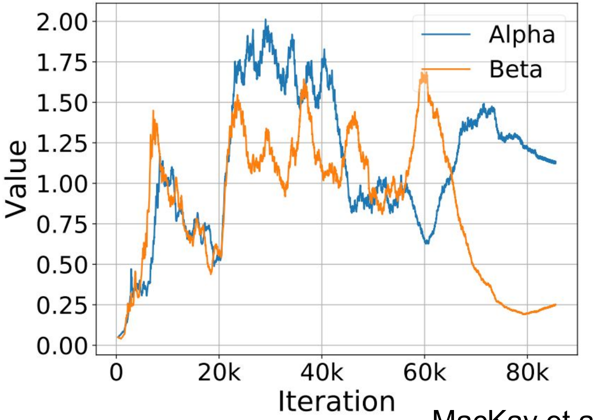
MacKay et al. *Self-Tuning Networks*. 2019.

STN - LSTM Experiment Results



PTB		
Method	**Val Perplexity**	**Test Perplexity**
Grid Search	97.32	94.58
Random Search	84.81	81.46
Bayesian Optimization	72.13	69.29
STN	**70.30**	**67.68**

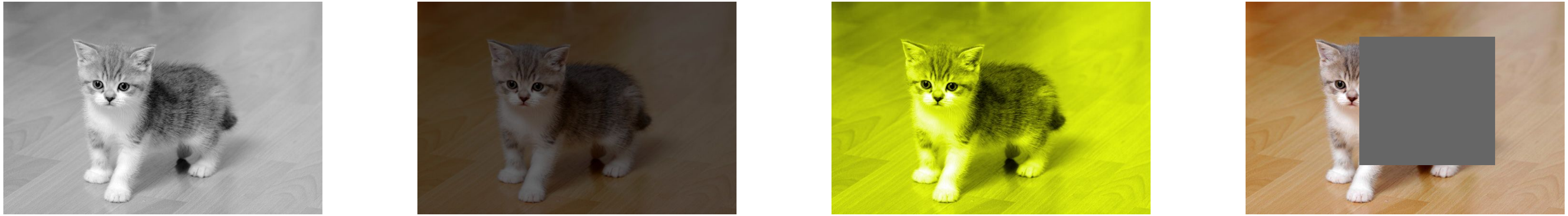




MacKay et al. *Self-Tuning Networks*. 2019.

STN - CNN Experiment Setup

- **Experiment:** AlexNet (~60 million parameters) on CIFAR-10
- 15 hyperparameters:
    - Separate dropout rates on each convolutional and fully-connected layer
    - *Data augmentation hyperparameters*



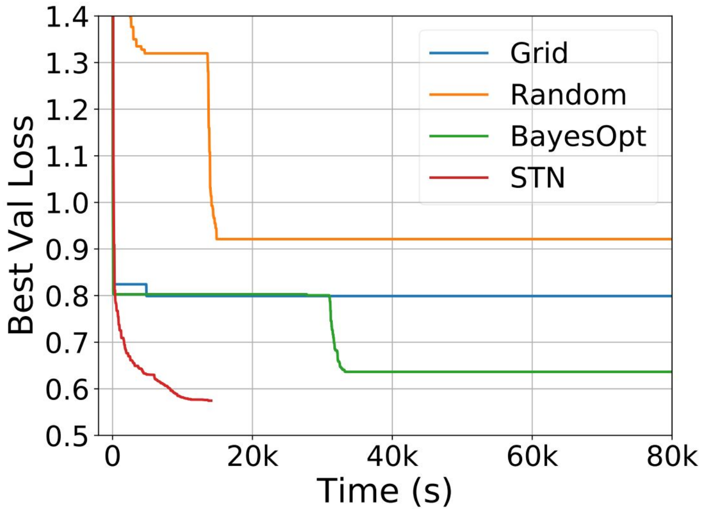
Saturation Brightness Hue Cutout

Continuous *Discrete*

MacKay et al. *Self-Tuning Networks*. 2019.

STN - CNN Experiment Results

	CIFAR-10	
Method	**Val Loss**	**Test Loss**
Grid Search	0.794	0.809
Random Search	0.921	0.752
Bayesian Optimization	0.636	0.651
STN	**0.575**	**0.576**

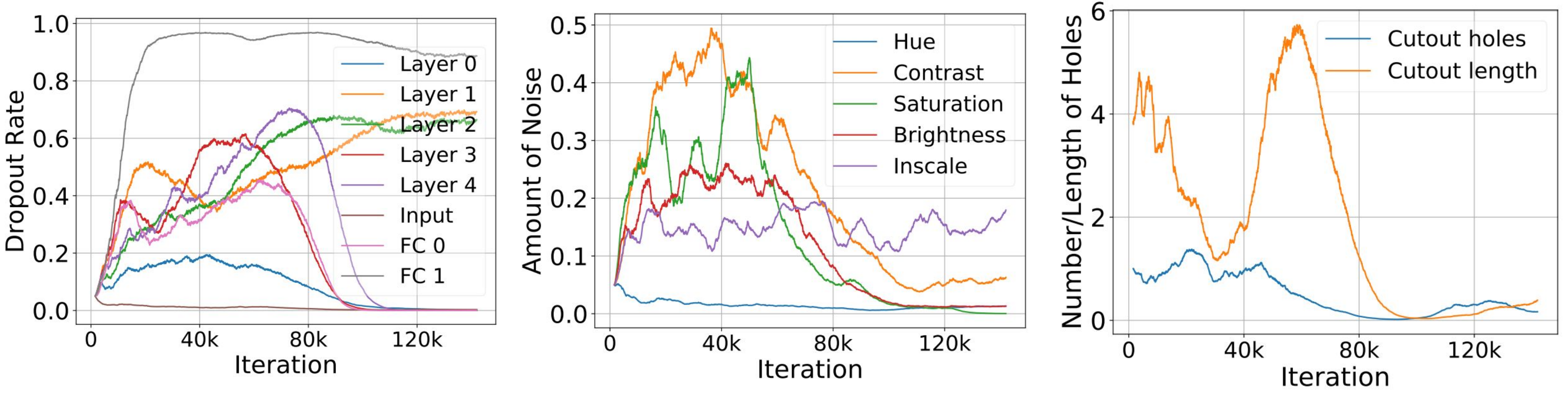


- Again, STNs substantially outperform grid/random search and BayesOpt
  - Achieve *lower validation loss than BayesOpt in < ¼ the time*

MacKay et al. *Self-Tuning Networks*. 2019.

STN - CNN Hyperparameter Schedules

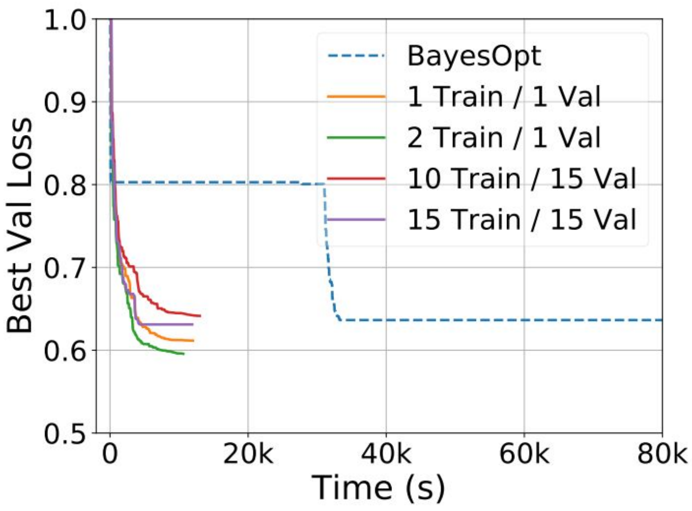
- STNs discover *nontrivial schedules for dropout and data augmentation*



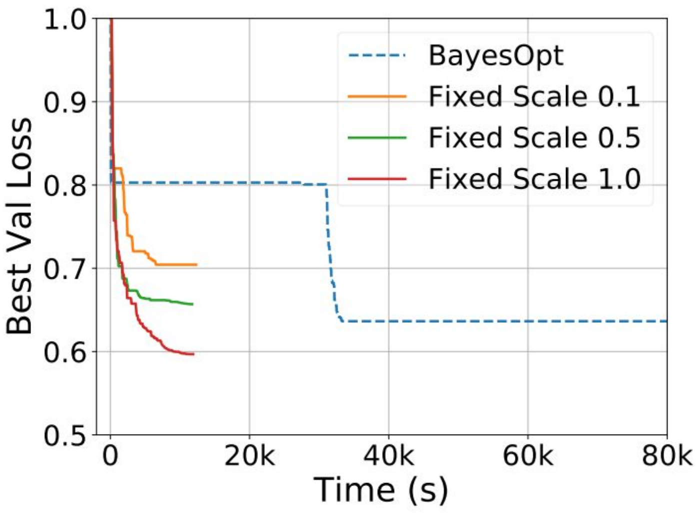
MacKay et al. *Self-Tuning Networks*. 2019.

STN - Sensitivity Analysis

- How often should we alternate between train and val steps?



- What is the effect of the variance of the hyperparameter distribution?



MacKay et al. *Self-Tuning Networks*. 2019.

What *can* we and what *can't* we tune?

What can we tune?

- STNs can tune *most regularization hyperparameters* including
  - Dropout
  - Continuous data augmentation hyperparameters (hue, saturation, contrast, etc.)
  - Discrete data augmentation hyperparameters (# and length of cutout holes)

What can't we tune?

- Because we collapsed the bilevel problem into a single-level one, *there is no inner training loop*

➡ We cannot tune inner optimization hyperparameters like *learning rates*

MacKay et al. *Self-Tuning Networks*. 2019.

Gradient-Based Approaches to HO

Implicit Differentiation

$$\lambda^* = \arg\min_{\lambda} \mathcal{L}_{val}(\lambda, \underbrace{\arg\min_{\mathbf{w}} \mathcal{L}_{train}(\lambda, \mathbf{w})}_{\frac{\partial \mathcal{L}_{train}(\lambda, \mathbf{w})}{\partial \mathbf{w}} = 0})$$

- Assuming *training has converged*, we can use the *implicit function theorem*

$$\frac{d\mathbf{w}(\lambda)}{d\lambda} = -\left(\frac{\partial^2 \mathcal{L}_{train}}{\partial \mathbf{w}^2}\right)^{-1} \frac{\partial^2 \mathcal{L}_{train}}{\partial \lambda \partial \mathbf{w}}$$

- *Expensive:* Solving the linear system with CG requires Hessian-vector products

Iterative Differentiation

$$\lambda^* = \arg\min_{\lambda} \mathcal{L}_{val}(\lambda, \underbrace{\arg\min_{\mathbf{w}} \mathcal{L}_{train}(\lambda, \mathbf{w})}_{\text{Backprop through optimization steps}})$$

- Use autodiff to *backprop through training*
- Full optimization procedure or a truncated version of it

- *Expensive* when the number of gradient steps increases

Hypernet-Based

$$\lambda^* = \arg\min_{\lambda} \mathcal{L}_{val}(\lambda, \underbrace{\arg\min_{\mathbf{w}} \mathcal{L}_{train}(\lambda, \mathbf{w})}_{\hat{\mathbf{w}}_\phi(\lambda) \approx \mathbf{w}^*(\lambda)})$$

- Learn a hypernetwork $\hat{\mathbf{w}}_\phi(\lambda) \approx \mathbf{w}^*(\lambda)$ parameterized by ϕ to *map hyperparameters to network weights*

- Does not require differentiating through optimization
- Efficient, can also optimize discrete & stochastic hyperparameters

Summary

- We propose a compact architecture for approximating neural net best-responses, that can be used as a *drop-in replacement* for existing deep learning modules.
- Our training algorithm alternates between approximating the best-response around the current hyperparameters and optimizing the hyperparameters with the approximate best-response.
    1. *Computationally inexpensive*
    2. Can optimize all *regularization hyperparameters*, including discrete hyperparameters
    3. *Scales to large NNs*
- Our approach discovers *hyperparameter schedules* that can outperform fixed hyperparameter values.

Q/A