CSC 2541: Neural Net Training Dynamics

Lecture 11 - Bilevel Optimization

Roger Grosse

University of Toronto, Winter 2022

Today

- Most of this course considered the optimization setting: minimizing a single cost function
- Last lecture considered differential games, where two or more "players" or "agents" simultaneously minimize/maximize different functions
  - Goal was to find a Nash equilibrium (no player can improve its utility by deviating from its current action, given the other players' actions)
- Now we consider bilevel optimization: minimize a cost function defined in terms of the optimal solution to another cost function
  - In game theory, this is a Stackelberg game, or leader-follower game. The difference is that one player moves first.
  - The analogous solution concept is a Stackelberg equilibrium.

Bilevel Optimization

- In bilevel (or nested) optimization, the outer (or upper) objective is defined in terms of the optimal solution to an inner (or lower) objective.

  $$\boldsymbol{\lambda}^* \in \text{``}\arg\min_{\boldsymbol{\lambda}}\text{''}\mathcal{J}_{\text{out}}(\boldsymbol{\lambda}, \mathbf{w}^*) \qquad \text{s.t.} \qquad \mathbf{w}^* \in \arg\min_{\mathbf{w}} \mathcal{J}_{\text{in}}(\boldsymbol{\lambda}, \mathbf{w}),$$

  where $\boldsymbol{\lambda}$ are the outer variables and $\mathbf{w}$ are the inner variables.
- I'll use hyperparameter optimization as a running example. Here, $\boldsymbol{\lambda}$ are the hyperparameters, and $\mathbf{w}$ are the network weights. $\mathcal{J}_{\text{out}}$ is the validation loss, and $\mathcal{J}_{\text{in}}$ is the regularized training loss.
- Here, I write $\in$ because the optimum may not be unique.
- There are scare quotes around the arg min since it's hard to write down a precise definition for the general case.

Bilevel Optimization

- For most of this lecture, I'll make the simplifying assumption that the optima are unique. In this case, we can define the best-response function, or rational reaction function,

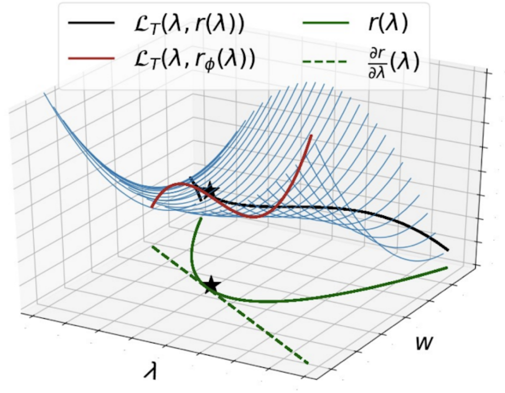
$$\mathbf{w}^* = r(\boldsymbol{\lambda}) = \arg\min_{\mathbf{w}} \mathcal{J}_{\text{in}}(\boldsymbol{\lambda}, \mathbf{w})$$

- The Implicit Function Theorem (IFT) proves existence under conditions which we won't worry about
- We can rewrite the optimization problem as:

$$\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda}} \mathcal{J}_{\text{out}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda})) \qquad \text{where} \qquad r(\boldsymbol{\lambda}) \triangleq \arg\min_{\mathbf{w}} \mathcal{J}_{\text{in}}(\boldsymbol{\lambda}, \mathbf{w}).$$

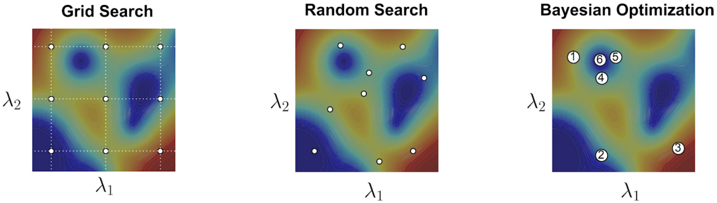
Bilevel Optimization

Example best-response function (green)



Black-Box Approaches

- One approach is to treat the outer objective as a black box: we can query function values, but not gradients, Hessians, etc.
- Each query: train the network and measure the validation loss
- The simplest algorithms are non-adaptive, like grid search and random search
- There are also adaptive algorithms which make use of information from past evaluations, like Bayesian optimization
- Drawbacks: can't use gradient information, each query is expensive
- I won't cover black-box methods since they don't raise any new NNTD issues



Hypergradient

Hypergradient

- Gradient-based optimizers are usually much more efficient than black-box ones
- To do gradient descent on $\boldsymbol{\lambda}$, we need the total gradient of $\mathcal{J}_{\text{val}}$ with respect to $\boldsymbol{\lambda}$. This is often called the hypergradient, to distinguish it from the inner gradient.

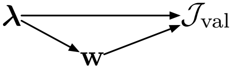
$$\overbrace{\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\lambda}}\left[\mathcal{J}_{\text{val}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))\right]}^{\text{total gradient}} = \underbrace{\frac{\partial \mathcal{J}_{\text{val}}}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))}_{\text{direct gradient}} + \underbrace{\overbrace{\left(\frac{\partial \mathbf{r}}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda})\right)}^{\text{response Jacobian}\top} \frac{\partial \mathcal{J}_{\text{val}}}{\partial \mathbf{w}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))}_{\text{response gradient}}$$

- This is just the Chain Rule. In CSC2516 backprop notation,

$$\overline{\mathcal{J}_{\text{val}}} = 1$$

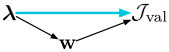
$$\overline{\mathbf{w}} = \frac{\partial \mathcal{J}_{\text{val}}}{\partial \mathbf{w}}^{\top} \overline{\mathcal{J}_{\text{val}}}$$

$$\overline{\boldsymbol{\lambda}} = \frac{\partial \mathcal{J}_{\text{val}}}{\partial \boldsymbol{\lambda}}^{\top} \overline{\mathcal{J}_{\text{val}}} + \frac{\partial \mathbf{w}}{\partial \boldsymbol{\lambda}}^{\top} \overline{\mathbf{w}}$$



Hypergradient: Direct Term

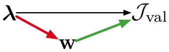
$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\lambda}}\left[\mathcal{J}_{\mathrm{val}}(\boldsymbol{\lambda},\mathbf{r}(\boldsymbol{\lambda}))\right] = \frac{\partial \mathcal{J}_{\mathrm{val}}}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda},\mathbf{r}(\boldsymbol{\lambda})) + \left(\frac{\partial \mathbf{r}}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda})\right)^{\top} \frac{\partial \mathcal{J}_{\mathrm{val}}}{\partial \mathbf{w}}(\boldsymbol{\lambda},\mathbf{r}(\boldsymbol{\lambda}))$$



- For optimizing regularization hyperparameters, the direct term is typically not very interesting
  - Regularizers like dropout or data augmentation aren't applied at validation time
  - Therefore, the direct term is **0**
- Example where we'd use a direct term: $\boldsymbol{\lambda}$ parameterizes a neural net architecture (# layers, # units, etc.), and we want to penalize the amount of memory or the number of arithmetic operations

Hypergradient: Response Term

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\lambda}}\left[\mathcal{J}_{\mathrm{val}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))\right] = \frac{\partial \mathcal{J}_{\mathrm{val}}}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda})) + \left(\frac{\partial \mathbf{r}}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda})\right)^{\top} \frac{\partial \mathcal{J}_{\mathrm{val}}}{\partial \mathbf{w}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))$$



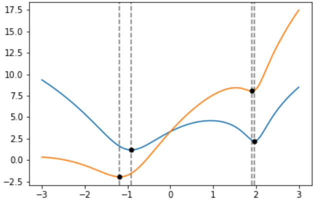
- The response term is more interesting: it says how changing $\boldsymbol{\lambda}$ influences the $\mathcal{J}_{\mathrm{val}}$ by way of changing $\mathbf{w}^*$
- The response Jacobian $\partial \mathbf{r}/\partial \boldsymbol{\lambda}$ measures how the optimal solution changes due to infinitesimal perturbations to $\boldsymbol{\lambda}$

Hypergradient: Response Term

- Formula for the response gradient (also given in Lecture 2):

$$\frac{\partial \mathbf{r}}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}) = -\underbrace{\left(\frac{\partial^2 \mathcal{J}_{\mathrm{tr}}}{\partial \mathbf{w}^2}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))\right)^{-1}}_{=\mathbf{H}^{-1}} \frac{\partial^2 \mathcal{J}_{\mathrm{tr}}}{\partial \boldsymbol{\lambda} \partial \mathbf{w}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))$$

- Sanity check from Lecture 2:



$\mathcal{J}(w; \lambda) = g(w) + \lambda w$ for $\lambda = 0$ and $\lambda = 3$

- How on earth do we wind up with $\mathbf{H}^{-1}$?

Hypergradient: Response Term

- We can derive the response Jacobian using a neat trick called implicit differentiation
- At a minimum $\mathbf{r}(\boldsymbol{\lambda})$ of the inner objective, $\partial \mathcal{J}_{\mathrm{tr}} / \partial \mathbf{w} = \mathbf{0}$
- Since this holds for *any* $\boldsymbol{\lambda}$, the total derivative w.r.t. $\boldsymbol{\lambda}$ must be $\mathbf{0}$:

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\lambda}} \left[\frac{\partial \mathcal{J}_{\mathrm{tr}}}{\partial \mathbf{w}} (\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda})) \right] = \mathbf{0}$$

- We expand this total derivative using the Chain Rule:

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\lambda}} \left[\frac{\partial \mathcal{J}_{\mathrm{tr}}}{\partial \mathbf{w}} (\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda})) \right] = \underbrace{\frac{\partial^2 \mathcal{J}_{\mathrm{tr}}}{\partial \boldsymbol{\lambda} \partial \mathbf{w}} (\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))}_{\text{direct term}} + \underbrace{\left(\frac{\partial \mathbf{r}}{\partial \boldsymbol{\lambda}} (\boldsymbol{\lambda}) \right)^{\top} \frac{\partial^2 \mathcal{J}_{\mathrm{tr}}}{\partial \mathbf{w}^2} (\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))}_{\text{response term}}$$

- Set this equal to $\mathbf{0}$ and solve for $\partial \mathbf{r} / \partial \boldsymbol{\lambda}$:

$$\frac{\partial \mathbf{r}}{\partial \boldsymbol{\lambda}} (\boldsymbol{\lambda}) = - \left(\frac{\partial^2 \mathcal{J}_{\mathrm{tr}}}{\partial \mathbf{w}^2} (\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda})) \right)^{-1} \frac{\partial^2 \mathcal{J}_{\mathrm{tr}}}{\partial \boldsymbol{\lambda} \partial \mathbf{w}} (\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))$$

Computing the Hypergradient

Computing the Hypergradient

- Want to compute:

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\lambda}}\left[\mathcal{J}_{\text{val}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))\right] = \frac{\partial \mathcal{J}_{\text{val}}}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda})) + \left(\frac{\partial \mathbf{r}}{\partial \boldsymbol{\lambda}}(\boldsymbol{\lambda})\right)^{\top} \frac{\partial \mathcal{J}_{\text{val}}}{\partial \mathbf{w}}(\boldsymbol{\lambda}, \mathbf{r}(\boldsymbol{\lambda}))$$

- The direct term and $\partial \mathcal{J}_{\text{val}} / \partial \mathbf{w}$ are easy to compute
- The hard part is multiplying by the response Jacobian, which requires the inverse Hessian:

$$\frac{\partial \mathbf{r}}{\partial \boldsymbol{\lambda}} = \underbrace{\left(\frac{\partial^2 \mathcal{J}_{\text{tr}}}{\partial \mathbf{w}^2}\right)^{-1}}_{=\mathbf{H}^{-1}} \frac{\partial^2 \mathcal{J}_{\text{tr}}}{\partial \mathbf{w} \partial \boldsymbol{\lambda}}$$

- The hypergradient is almost always estimated in one of two ways:
  1. Approximately solve the linear system using an iterative algorithm (e.g. CG), like many examples from this class
  2. Unroll the inner optimization, and backprop through it as if it were a neural net

Computation: Solving the Linear System

- **Approach 1:** Iteratively solve the linear system
- First, optimize the inner objective to convergence
    - In practice, we usually settle for approximate solutions, but the theoretical justification is unclear
- We can solve the linear system using algorithms like CG, computing the Hessian-vector products in the usual way (see Lecture 2)

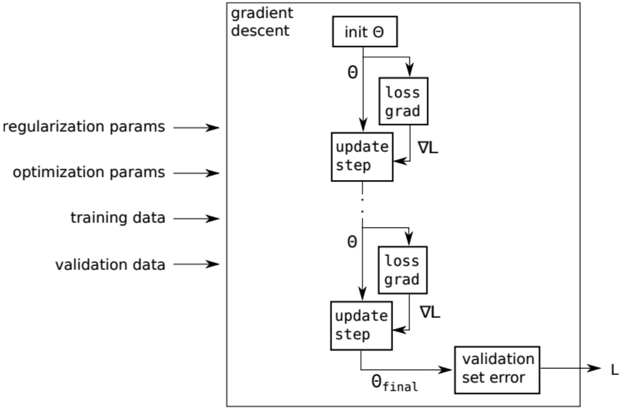
Computation: Solving the Linear System

Examples:

- Bengio (2000) used implicit differentiation to optimize ML hyperparameters (exact solution for small models)
    - Lorraine et al. (2020): optimizing millions of hyperparameters
- Influence functions (Koh and Liang, 2017) (coming up)
- Optimization layers (Amos and Kolter, 2017): neural net layers defined implicitly in terms of the solution to an optimization problem
    - generalized the IFT trick to constrained optimization
- Deep equilibrium models (Bai et al., 2019)
- Implicit MAML (Rajeswaran et al., 2019): a variant of MAML that uses implicit differentiation
    - The original MAML uses unrolling (covered next)

Computation: Unrolling

- **Approach 2:** Unroll the inner optimization
- Computation graph for gradient descent:

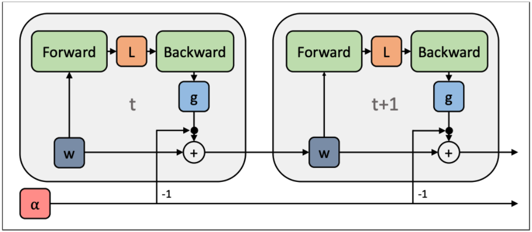


- Backprop through this graph in the usual way

Figure Credit: David Duvenaud

Computation: Unrolling

- In contrast to implicit differentiation, unrolling can be used to tune optimization hyperparameters (this is known as meta-optimization)
- Here's the computation graph for adapting the learning rate (known as meta-descent)



- Just because you can do this doesn't mean it's a good idea. We'll see in just a bit what actually happens.

Figure: Wu et al., 2018, "Understanding short-horizon bias"

Computation: Unrolling

Examples:

- Domke (2012): learning energy-based models
- Maclaurin et al. (2015): hyperparameter optimization (student presentation next week)
  - This was also the paper that introduced Autograd, the predecessor to JAX
- MAML (Finn et al., 2017): student presentation next week
- adapting learning rates (Baydin et al., 2018)
- “Learning to learn by gradient descent by gradient descent” (Andrychowicz et al., 2016): tried to use unrolling to learn an optimization algorithm (represented as an RNN)
- differentiable neural architecture search (DARTS) (Liu et al., 2019) (unrolls only one iteration???)

Implicit Differentiation vs. Unrolling

Implicit Differentiation vs. Unrolling

Which method to use?

- Lorraine et al. (2020) related the two methods to each other.
- Suppose we unroll the inner optimization and train it to covergence. Then we use truncated backprop through time, which only backprops through the last K time steps
- They showed that this method is equivalent to approximately solving the IFT system by doing gradient descent on a quadratic objective
$$\frac{1}{2}\Delta \mathbf{w}^\top \mathbf{H} \Delta \mathbf{w} + \nabla_{\boldsymbol{\lambda}} \nabla_{\mathbf{w}} \mathcal{J}_{\text{tr}}(\mathbf{w})^\top \Delta \mathbf{w}$$
- This is also equivalent to Neumann iterations, a method for solving linear systems

Implicit Differentiation vs. Unrolling

- **Educated guess:** implicit differentiation using Neumann iterations will behave similarly to unrolling gradient descent, for problems where they're both applicable
    - I'm not aware of any rigorous investigation of this
    - Implicit differentiation using CG should converge more efficiently for deterministic inner objectives
    - For stochastic inner objectives (e.g. neural net training), stochastic Neumann iterations (SGD on the quadratic) should be more efficient than (batch) CG in practice, if it's noise dominated rather than curvature dominated (Lecture 7)
        - This was the approach taken by Koh et al. (2017) for influence functions

Implicit Differentiation vs. Unrolling

- Implicit differentiation uses much less memory than unrolling
    - Unrolling requires storing the individual iterates (parameter vectors) along the optimziation trajectory
    - A big piece of Maclaurin et al. (2015)'s work on hyperparameter optimization was a scheme for cheaply storing the parameter vectors

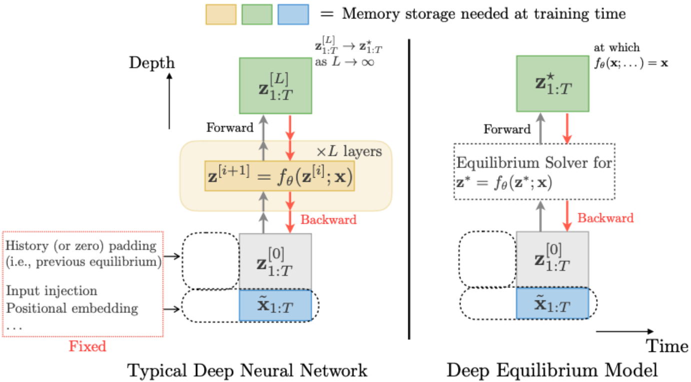

= Memory storage needed at training time
Depth
$\mathbf{z}_{1:T}^{[L]} \to \mathbf{z}_{1:T}^{\star}$ as $L \to \infty$
$\mathbf{z}_{1:T}^{[L]}$
Forward
$\times L$ layers
$\mathbf{z}^{[i+1]} = f_\theta(\mathbf{z}^{[i]}; \mathbf{x})$
Backward
History (or zero) padding (i.e., previous equilibrium)
Input injection
Positional embedding
...
Fixed
$\mathbf{z}_{1:T}^{[0]}$
$\tilde{\mathbf{x}}_{1:T}$
Typical Deep Neural Network
at which $f_\theta(\mathbf{x}; \dots) = \mathbf{x}$
$\mathbf{z}_{1:T}^{\star}$
Forward
Equilibrium Solver for $\mathbf{z}^* = f_\theta(\mathbf{z}^*; \mathbf{x})$
Backward
$\mathbf{z}_{1:T}^{[0]}$
$\tilde{\mathbf{x}}_{1:T}$
Time
Deep Equilibrium Model

Figure: Bai et al., "Deep equilibrium models"

Implicit Differentiation vs. Unrolling

To summarize:

- Unrolled SGD is closely related to implicit differentiation with Neumann iterations
- Some advantages of implicit differentiation:
  - Lower memory costs
  - Can use faster-converging algorithms for solving the linear system (e.g. CG, Broyden's method)
- Some advantages of unrolling:
  - Trivial to implement (in JAX)
  - Can adapt optimization hyperparameters
  - Still makes sense if the inner minimization is approximate
- In practice, they're largely interchangeable, in cases where they're both applicable
- Both methods have the drawback that you have to do an inner optimization for every outer update

Influence Functions

Influence Functions

Koh and Liang, 2017, "Understanding black-box predictions via influence functions."

- Goal: understand which training examples are most important by determining which ones, if removed, result in the largest change to a prediction, or to the test loss.
- Abstractly, we determine the effect of an infinitesimal change to a dataset by computing the hypergradient of the following bilevel program:

$$\boldsymbol{\eta}^* = \arg\min_{\boldsymbol{\eta}} f(\mathbf{w}^*(\boldsymbol{\eta}))$$

$$\mathbf{w}^*(\boldsymbol{\eta}) = \arg\min_{\mathbf{w}} \mathcal{J}_{\boldsymbol{\eta}}(\mathbf{w}, \mathcal{D}_{\boldsymbol{\eta}}),$$

where $\mathcal{J}_{\boldsymbol{\eta}}$ is the training objective, and $\mathcal{D}_{\boldsymbol{\eta}}$ is the training set, parameterized by $\boldsymbol{\eta}$.

Influence Functions

- If we are interested in estimating the impact of *removing* training examples, we can let $\boldsymbol{\eta}$ represent the weightings on each of the training examples:

$$\mathcal{J}_{\boldsymbol{\eta}}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \eta_i \mathcal{L}(f(\mathbf{w}, \mathbf{x}^{(i)}), t^{(i)})$$

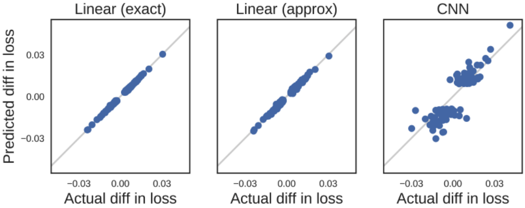
- The influence is given by the hypergradient, evaluated at $\eta_i = 1$ for all i.
- Conceptually, this is a first-order Taylor approximation to the effect of removing the training example.
- Additional error is introduced by the fact that the hypergradient can only be computed approximately. (They use LiSSA, a variant of stochastic Neumann iterations.)

Influence Functions

- Conceptually, the influence of a training example approximates the effect of training the network with that example removed.
- Why is this computationally advantageous?
  - The naïve approach requires separately re-training the network for *each* training example you want to compute the influence of.
  - By computing the hypergradient, we get the influence for *all* training examples by solving a single linear system.
  - Their method, using LiSSA, requires K Hessian-vector products on mini-batches. Each one is about as expensive as an SGD update.

Influence Functions

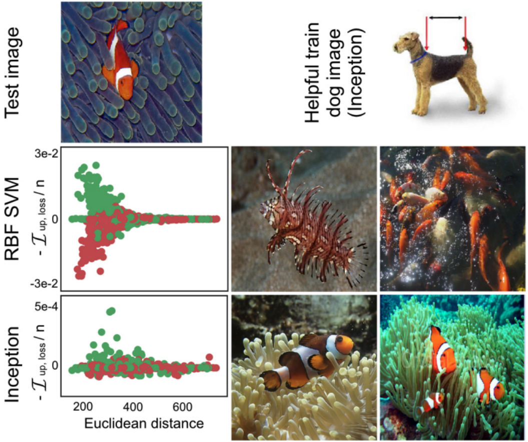
- They measure the accuracy of the approximation by comparing the influence estimate against the results of retraining the model with the data point removed.



- Caution: more recent work showed that influence estimates are often inaccurate. More work is needed to determine when you can expect accurate results. See Basu et al., "Influence functions in deep learning are fragile."

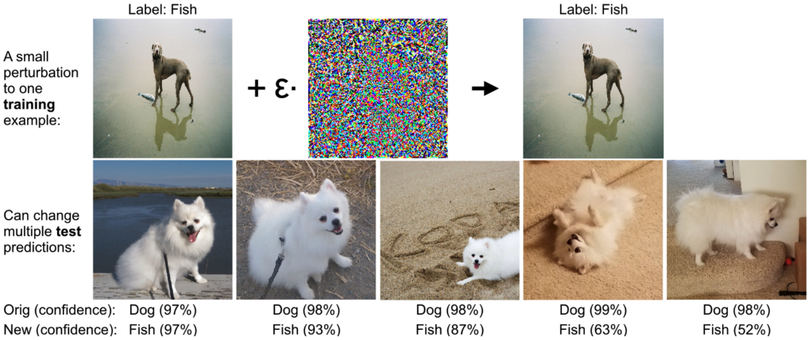
Influence Functions

- We can understand the difference between two models (in this case, an RBF model vs. Inception CNN) by seeing which training examples are the most relevant to a prediction:



Influence Functions

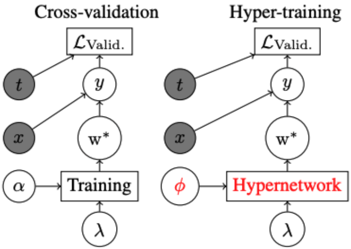
- They use the same algorithmic technique for data poisoning.
    - Goal: modify a training example in a way that makes the learned classifier perform *poorly* on a set of test images.
- Here, $\boldsymbol{\eta}$ parameterizes a training image, rather than the weightings of the training examples. Otherwise, the algorithm is identical.

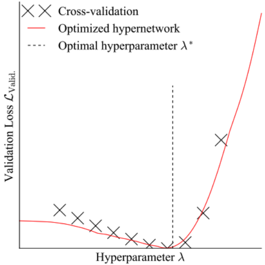

Label: Fish
A small perturbation to one **training** example:
+ ε·
Label: Fish
Can change multiple **test** predictions:
Orig (confidence): Dog (97%) Dog (98%) Dog (98%) Dog (99%) Dog (98%)
New (confidence): Fish (97%) Fish (93%) Fish (87%) Fish (63%) Fish (52%)

Hypernetworks

Hypernetworks

- Another approach we're starting to explore is to learn a hypernetwork which tries to approximate the best-response function
  - Takes in $\boldsymbol{\lambda}$, outputs $\mathbf{w}$
- At any time, we have a guess of the optimal $\boldsymbol{\lambda}$. We want the hypernet to be accurate in the vicinity of $\boldsymbol{\lambda}$

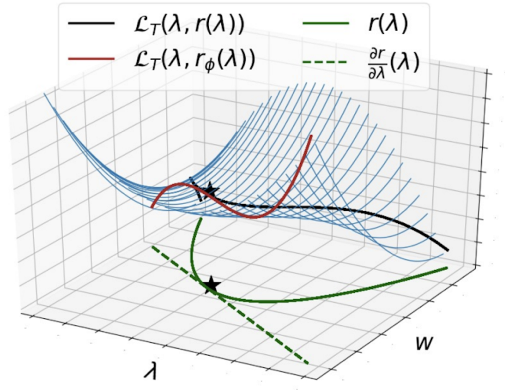




Figures: Lorraine and Duvenaud, "Stochastic hyperparameter optimization through hypernetworks"

Hypernetworks

Best-response function approximated in a parametric form (linear):



Hypernetworks

- Suppose we've somehow learned a hypernetwork $\mathbf{r}_{\boldsymbol{\phi}}$. We can compute the hypergradient by computing the derivatives of:

  $$\mathcal{J}_{\text{val}}^{\boldsymbol{\phi}}(\boldsymbol{\lambda}) = \mathcal{J}_{\text{val}}(\boldsymbol{\lambda}, \mathbf{r}_{\boldsymbol{\phi}}(\boldsymbol{\lambda}))$$

  This is just ordinary backprop. No inverse Hessian required!
- Computing the gradient requires the value and Jacobian of $\mathbf{r}_{\boldsymbol{\phi}}(\boldsymbol{\lambda})$. Our goal in training the hypernetwork is to make sure these are accurate at $\boldsymbol{\lambda}$.
- In contrast to implicit differentiation and unrolling, this lets us amortize the cost of computing the hypergradient.

Hypernetworks

- Training iteration for the hypernetwork:
  - Sample perturbed hyperparameters $\boldsymbol{\lambda}' = \boldsymbol{\lambda} + \boldsymbol{\epsilon}$, $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$
  - Sample a training batch (index i)
  - Do the gradient update:

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \alpha \nabla_{\boldsymbol{\phi}} \mathcal{J}_{\mathrm{tr}}(\boldsymbol{\lambda}', \mathbf{r}_{\boldsymbol{\phi}}(\boldsymbol{\lambda}'))$$

- Note: the perturbation scale $\boldsymbol{\Sigma}$ is important

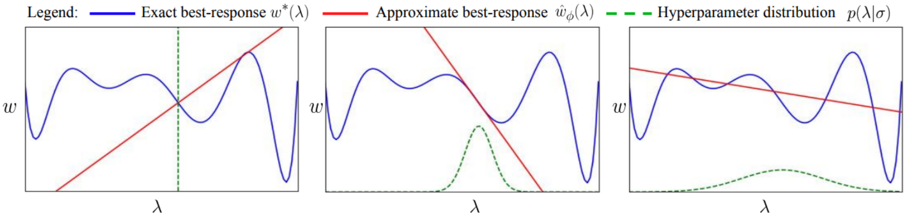


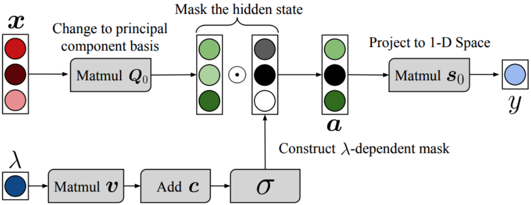
Figure: MacKay et al., “Self-tuning networks”

Self-Tuning Networks

- **Problem:** the hypernetwork is an extremely high-dimensional mapping
    - Input dimension is H (number of hyperparameters), output dimension is D (number of network parameters)
- We can linearize the network, but it's still size HD
- Self-tuning networks (STNs) (MacKay et al., 2019) are the first scalable approach to bilevel optimization using hypernetworks. The trick is a compact and efficient representation of the hypernet

Self-Tuning Networks

- **Some inspiration:** the following architecture exactly represents the global best-response function for linear regression



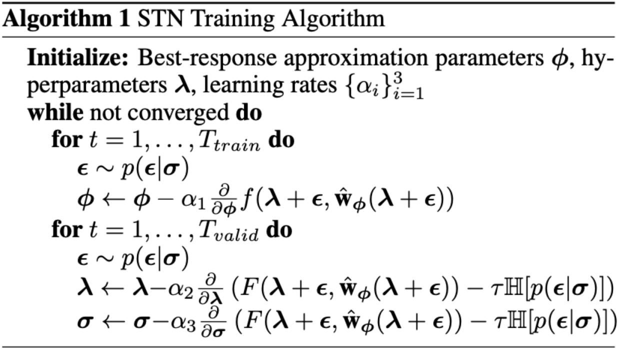
- We have an ordinary network (the base network) whose activations are modulated based on $\boldsymbol{\lambda}$
- This modulation can be equivalently interpreted as rescaling the rows of the weight matrix:

$$\mathbf{Q}(\lambda) = \sigma(\lambda \mathbf{v} + \mathbf{c}) \odot_{\text{row}} \mathbf{Q}_0$$

- This is essentially how STN layers are defined (details in the paper)

Self-Tuning Networks

- The hyperparameters $\boldsymbol{\lambda}$ and the hypernetwork parameters $\boldsymbol{\phi}$ are trained jointly
- This converts the bilevel optimization problem into a simultaneous game (as in Lecture 10)
- The perturbation scale σ is also adapted simultaneously



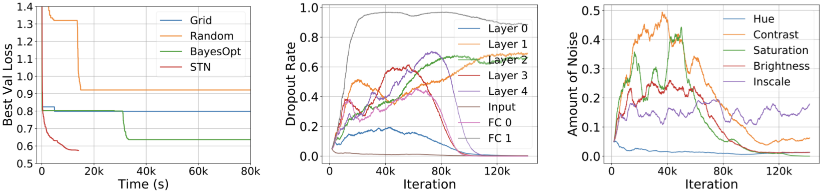
Algorithm 1 STN Training Algorithm

```
Initialize: Best-response approximation parameters $\boldsymbol{\phi}$, 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
        $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$
        $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \alpha_2 \frac{\partial}{\partial \boldsymbol{\lambda}} \left(F(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon})) - \tau \mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})]\right)$
        $\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma} - \alpha_3 \frac{\partial}{\partial \boldsymbol{\sigma}} \left(F(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon})) - \tau \mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})]\right)$
```

Self-Tuning Networks

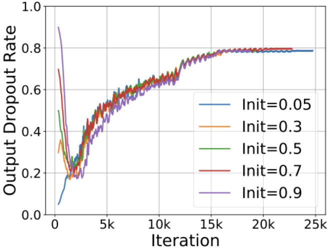
Training 15 hyperparameters of a CIFAR-10 classifier

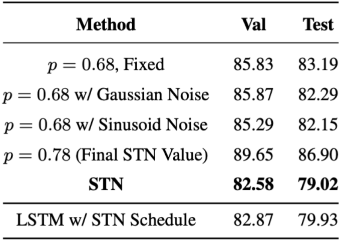
- layer-specific dropout
- input noise
- discrete data augmentation (cutout)
- continuous data augmentation (perturb hue, saturation, brightness, contrast)



Self-Tuning Networks

- Adapting the dropout rate for an LSTM PTB language model
- Yields a schedule that seems to outperform any particular hyperparameter
- Note that the initialization is unlearned quickly





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

Δ-STN

- Remember how it's a good idea to center the inputs? (Lecture 1)
- Our original STN used an uncentered parameterization of the hypernetwork:

$$\mathbf{w} = \mathbf{r}_\phi(\boldsymbol{\lambda}) = \boldsymbol{\Phi}\boldsymbol{\lambda} + \boldsymbol{\phi}_0$$

- The Δ-STN (Bae and Grosse, 2020) makes several algorithmic improvements, including a centered parameterization:

$$\mathbf{r}_\phi(\boldsymbol{\lambda}) = \boldsymbol{\Phi}(\boldsymbol{\lambda} - \boldsymbol{\lambda}_0) + \mathbf{w}_0$$

- In Lecture 1, we understood this trick in terms of conditioning and outlier eigenvalues.
  - That explanation still applies, but for hypernetworks uncentering causes an even bigger problem.

Δ-STN

- STN parameterization:

$$\mathbf{r}_{\phi}(\boldsymbol{\lambda}) = \boldsymbol{\Phi}\boldsymbol{\lambda} + \phi_0$$

- Gradient descent update for $\boldsymbol{\Phi}$:

$$\boldsymbol{\Phi} \leftarrow \boldsymbol{\Phi} - \alpha[\nabla_{\mathbf{w}} \mathcal{J}_{\mathrm{tr}}(\boldsymbol{\lambda}, \mathbf{w})]\boldsymbol{\lambda}^{\top}$$

- So early in training, approximately $\boldsymbol{\Phi} \propto -\mathbf{g}\boldsymbol{\lambda}^{\top}$, where $\mathbf{g}$ is the weight gradient.
  - The response Jacobian mistakenly thinks that adjusting $\boldsymbol{\lambda}$ in the direction $\boldsymbol{\lambda}$ will cause $\mathbf{w}^*(\boldsymbol{\lambda})$ to move opposite the gradient direction!

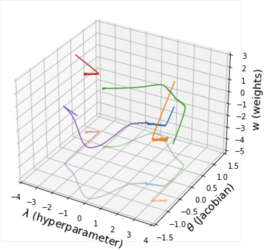
Δ-STN

Illustrative Example:

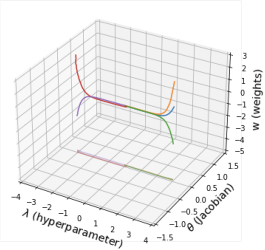
$$\mathcal{J}_{\mathrm{val}}(\lambda, w) = \frac{1}{10}\lambda^2 + w$$
$$\mathcal{J}_{\mathrm{tr}}(\lambda, w) = w^2$$

- This is an easy problem, since the inner objective doesn't depend on λ.
- The response Jacobian is 0, so λ and w can be optimized separately.
- Optimum: $\lambda = w = 0$.

Uncentered parameterization:

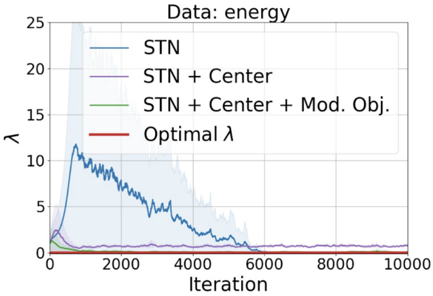


Centered parameterization:



Δ-STN

Centering eliminates some pathological choices of hyperparameters early in training


Data: energy
STN
STN + Center
STN + Center + Mod. Obj.
Optimal λ
λ
Iteration