Gradient-based Hyperparameter Optimization with Reversible Learning

Dougal Maclaurin, David Duvenaud, Ryan Adams

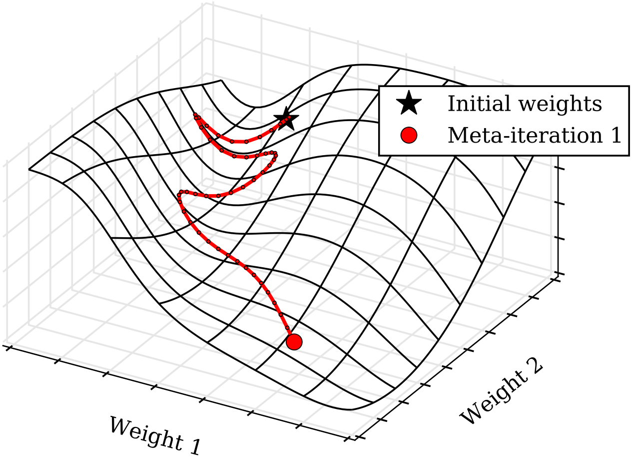
HARVARD
School of Engineering
and Applied Sciences

Motivation

- Hyperparameters are everywhere
  - sometimes hidden!
- Gradient-free optimization is hard
- Validation loss is a function of hyperparameters
- Why not take gradients?

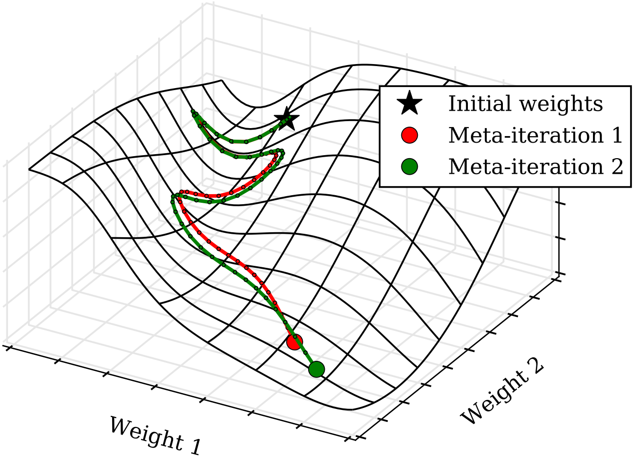
Optimizing optimization

$$\mathbf{x}_{final} = \text{SGD}\left(\mathbf{x}_{init}, \text{learn rate}, \text{momentum}, \nabla\text{Loss}(\mathbf{x}, \text{reg}, \textit{Data})\right)$$



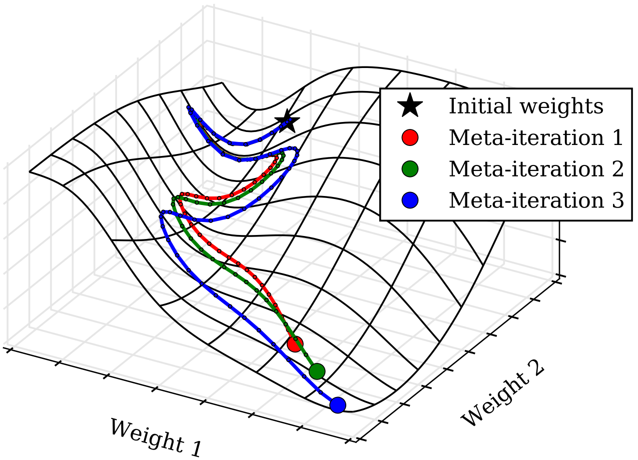
Optimizing optimization

$$\mathbf{x}_{final} = \text{SGD}\left(\mathbf{x}_{init}, \text{learn rate}, \text{momentum}, \nabla\text{Loss}(\mathbf{x}, \text{reg}, \textit{Data})\right)$$



Optimizing optimization

$$\mathbf{x}_{final} = \text{SGD}\left(\mathbf{x}_{init}, \text{learn rate}, \text{momentum}, \nabla\text{Loss}(\mathbf{x}, \text{reg}, \mathit{Data})\right)$$



A pretty scary function to differentiate

$$J = Loss\left(D_{val}, \text{SGD}\left(\mathbf{x}_{init}, \alpha, \beta, \nabla\text{Loss}(D_{train}, \mathbf{x}, \text{reg})\right)\right)$$

Stochastic Gradient Descent

1: **input:** initial $\mathbf{x}_1$, decay $\boldsymbol{\beta}$, learning rate α, regularization params θ, loss function $L(\mathbf{x}, \boldsymbol{\theta}, t)$
2: initialize $\mathbf{v}_1 = \mathbf{0}$
3: **for** $t = 1$ **to** T **do**
4: $\quad \mathbf{g}_t = \nabla_{\mathbf{x}} L(\mathbf{x}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
5: $\quad \mathbf{v}_{t+1} = \beta_t \mathbf{v}_t - (1 - \beta_t)\mathbf{g}_t$ ▷ update velocity
6: $\quad \mathbf{x}_{t+1} = \mathbf{x}_t + \alpha_t \mathbf{v}_t$ ▷ update position
7: **output** trained parameters $\mathbf{x}_T$

- Each gradient evaluation in SGD requires forward and backprop through model
- Entire learning procedure looks like a 1000-layer deep net

Autograd: Automatic Differentiation

- `github.com/HIPS/autograd`
- Works with (almost) arbitrary Python/Numpy code
- Can take gradients of gradients (of gradients...)

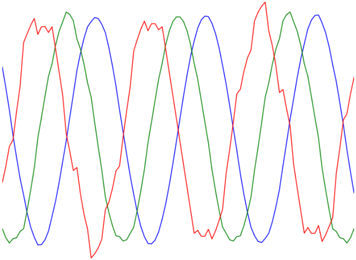
Autograd Example

```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

# Taylor approximation to sin function
def fun(x):
    curr = x
    ans = curr
    for i in xrange(1000):
        curr = - curr * x**2 / ((2*i+3)*(2*i+2))
        ans = ans + curr
        if np.abs(curr) < 0.2: break
    return ans

d_fun = grad(fun)
dd_fun = grad(d_fun)

x = np.linspace(-10, 10, 100)
plt.plot(x, map(fun, x),
         x, map(d_fun, x),
         x, map(dd_fun, x))
```



Most Numpy functions implemented

Complex & Fourier	Array	Misc	Linear Algebra	Stats
imag	atleast_1d	logsumexp	inv	std
conjugate	atleast_2d	where	norm	mean
angle	atleast_3d	einsum	det	var
real_if_close	full	sort	eigh	prod
real	repeat	partition	solve	sum
fabs	split	clip	trace	cumsum
fft	concatenate	outer	diag	
fftshift	roll	dot	tril	
fft2	transpose	tensordot	triu	
ifftn	reshape	rot90		
ifftshift	squeeze			
ifft2	ravel			
ifft	expand_dims			

Technical Challenge: Memory

- Reverse-mode differentiation needs access to entire learning trajectory
- i.e. 10^7 parameters $\times$ 10^5 training iterations
- Only need access in reverse order...
- Could we recompute the learning trajectory backwards by running reverse SGD?

SGD with momentum is reversible

Forward update rule:

$$\mathbf{x}_{t+1} \leftarrow \mathbf{x}_t + \alpha \mathbf{v}_t$$
$$\mathbf{v}_{t+1} \leftarrow \beta \mathbf{v}_t - \nabla L(\mathbf{x}_{t+1})$$

Reverse update rule:

$$\mathbf{v}_t \leftarrow (\mathbf{v}_{t+1} + \nabla L(\mathbf{x}_{t+1})) / \beta$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t+1} - \alpha \mathbf{v}_t$$

Reverse-mode differentiation of SGD

Stochastic Gradient Descent

input: initial $\mathbf{x}_1$, decays $\boldsymbol{\beta}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\mathbf{x}, \boldsymbol{\theta}, t)$

initialize $\mathbf{v}_1 = \mathbf{0}$

for $t = 1$ **to** T **do**

  $\mathbf{g}_t = \nabla_{\mathbf{x}} L(\mathbf{x}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient

  $\mathbf{v}_{t+1} = \beta_t \mathbf{v}_t - (1 - \beta_t)\mathbf{g}_t$ ▷ update velocity

  $\mathbf{x}_{t+1} = \mathbf{x}_t + \alpha_t \mathbf{v}_t$ ▷ update position

output trained parameters $\mathbf{x}_T$

Reverse-Mode Gradient of SGD

input: $\mathbf{x}_T$, $\mathbf{v}_T$, $\boldsymbol{\beta}$, $\boldsymbol{\alpha}$, train loss $L(\mathbf{x}, \boldsymbol{\theta}, t)$, loss $f(\mathbf{x})$

initialize $d\mathbf{v} = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$, $d\alpha_t = \mathbf{0}$, $d\beta = \mathbf{0}$

initialize $d\mathbf{x} = \nabla_{\mathbf{x}} f(\mathbf{x}_T)$

for $t = T$ **counting down to** 1 **do**

  $d\alpha_t = d\mathbf{x}^{\mathsf{T}} \mathbf{v}_t$

  $\mathbf{x}_{t-1} = \mathbf{x}_t - \alpha_t \mathbf{v}_t$ ▷ downdate position

  $\mathbf{g}_t = \nabla_{\mathbf{x}} L(\mathbf{x}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient

  $\mathbf{v}_{t-1} = [\mathbf{v}_t + (1 - \beta_t)\mathbf{g}_t] / \beta_t$ ▷ downdate velocity

  $d\mathbf{v} = d\mathbf{v} + \alpha_t d\mathbf{x}$

  $d\beta_t = d\mathbf{v}^{\mathsf{T}}(\mathbf{v}_t + \mathbf{g}_t)$

  $d\mathbf{x} = d\mathbf{x} - (1 - \beta_t) d\mathbf{v} \nabla_{\mathbf{x}} \nabla_{\mathbf{x}} L(\mathbf{x}_t, \boldsymbol{\theta}, t)$

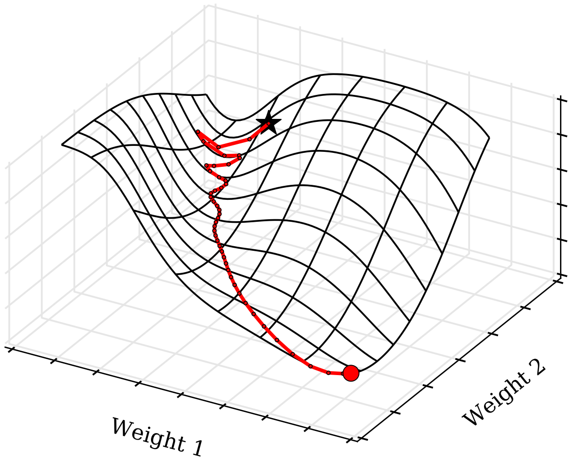
  $d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \beta_t) d\mathbf{v} \nabla_{\boldsymbol{\theta}} \nabla_{\mathbf{x}} L(\mathbf{x}_t, \boldsymbol{\theta}, t)$

  $d\mathbf{v} = \beta_t d\mathbf{v}$

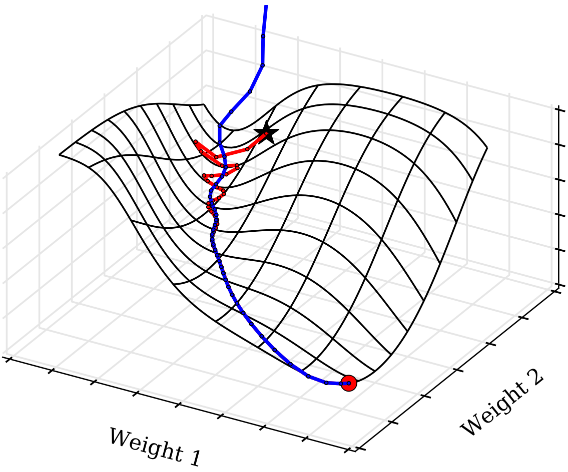
output gradient of $f(\mathbf{x}_T)$ w.r.t $\mathbf{x}_1$, $\mathbf{v}_1$, $\boldsymbol{\beta}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}$

- Outputs gradients with respect to all hypers.
- Reversing SGD avoids storing learning trajectory

Naive reversal



Naive reversal ... Fails!



A closer look at reverse SGD

Forward update rule:

$$\mathbf{x}_{t+1} \leftarrow \mathbf{x}_t + \alpha \mathbf{v}_t$$
$$\mathbf{v}_{t+1} \leftarrow \beta \mathbf{v}_t - \nabla L(\mathbf{x}_{t+1})$$

Destroys $\log_2 \beta$ bits per parameter per iteration

Reverse update rule:

$$\mathbf{v}_t \leftarrow (\mathbf{v}_{t+1} + \nabla L(\mathbf{x}_{t+1})) / \beta$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t+1} - \alpha \mathbf{v}_t$$

Needs $\log_2 \beta$ bits per parameter per iteration

How to store the lost bits?

- Switch to fixed-precision for exact addition
- Express β as a rational number
- push/pop remainders from an information buffer

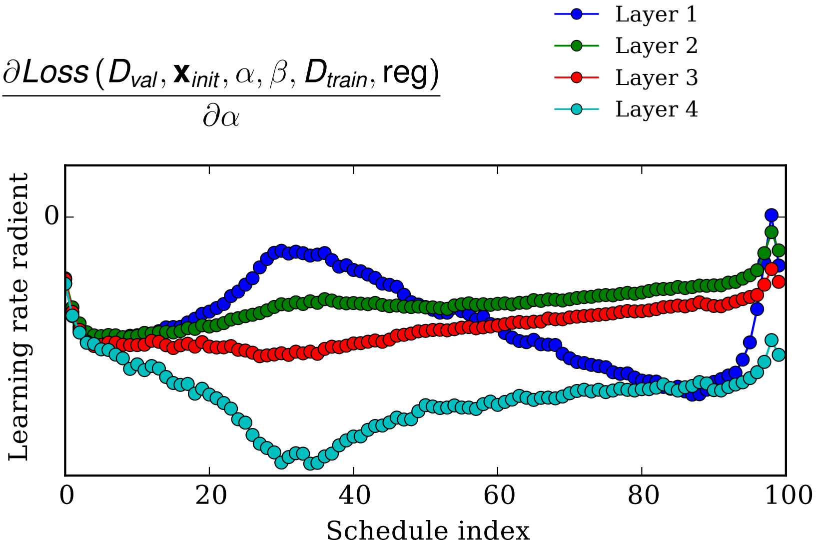
```
def rational_multiply(x, n, d, bitbuffer):
    bitbuffer.push(x % d, d)
    x /= d
    x *= n
    x += bitbuffer.pop(n)
    return x
```

- 200X memory savings when $\beta = 0.9$
- Now we have scalable gradients of hypers, only twice as slow as original!

Part 2: A Garden of Delights

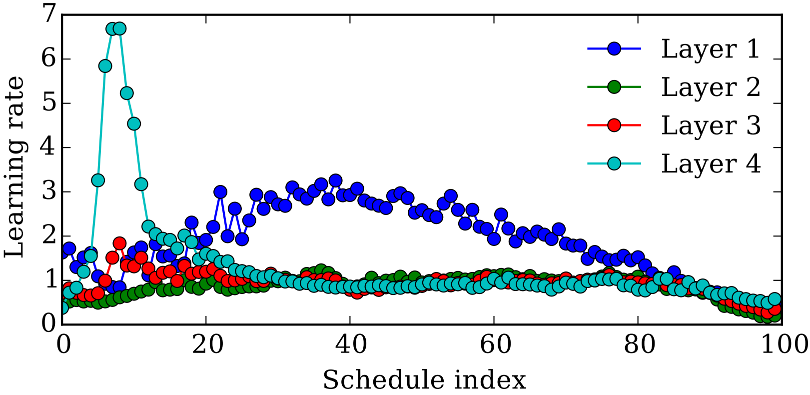
Learning rate gradients

$$\frac{\partial Loss\left(D_{val}, \mathbf{x}_{init}, \alpha, \beta, D_{train}, \text{reg}\right)}{\partial \alpha}$$



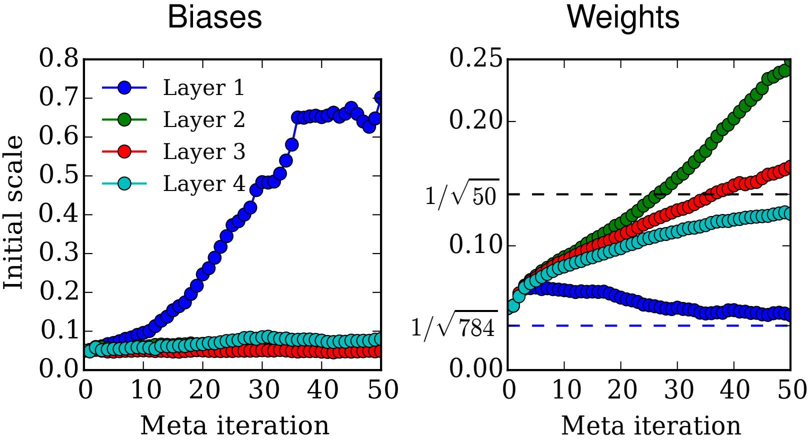
Optimized learning rates

- Used SGD to optimize SGD
- 4-layer NN on MNIST
- Top layer learns early on; slowdown at end



Optimizing initialization scales

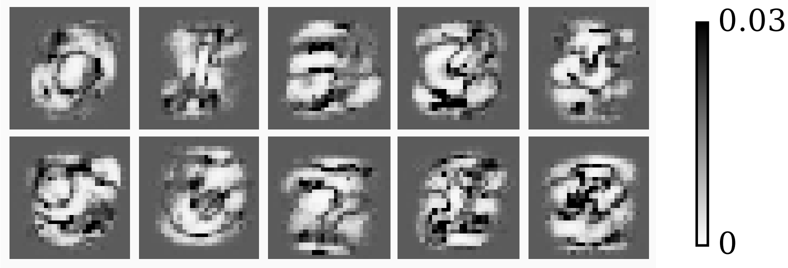
$$\frac{\partial Loss\,(D_{val}, \mathbf{x}_{init}, \alpha, \beta, D_{train}, \text{reg})}{\partial \mathbf{x}_{init}}$$



Optimizing regularization

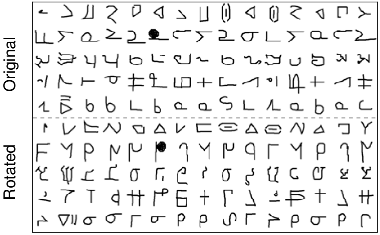
$$\frac{\partial Loss\left(D_{val}, \mathbf{x}_{init}, \alpha, \beta, D_{train}, \text{reg}\right)}{\partial \text{reg}}$$

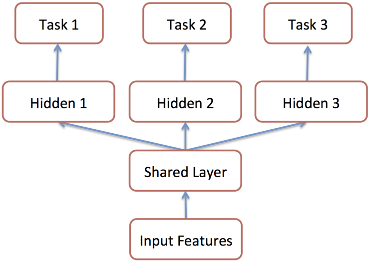
Optimized L_2 hypers for each weight in logistic regression:



Optimizing architecture

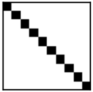
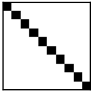
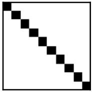
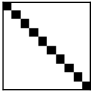
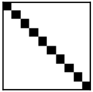
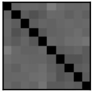
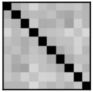
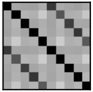
- Architecture = tying weights or setting to them zero
- i.e. convnets, recurrent nets, multi-task
- Trying be enforced by L2 regularization
- L2 regularization is differentiable





Optimizing regularization

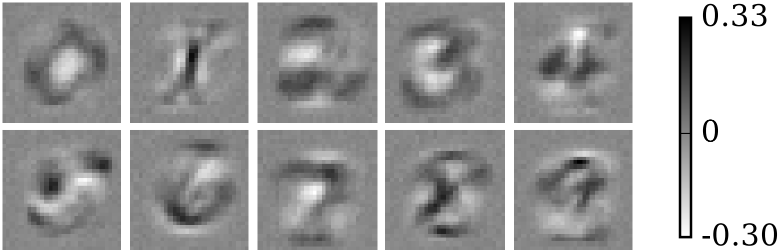
Matrices enforce weight sharing between tasks

	Input weights	Middle weights	Output weights	Train error	Test error
Separate net-works				0.61	1.34
Tied weights				0.90	1.25
Learned sharing				0.60	**1.13**

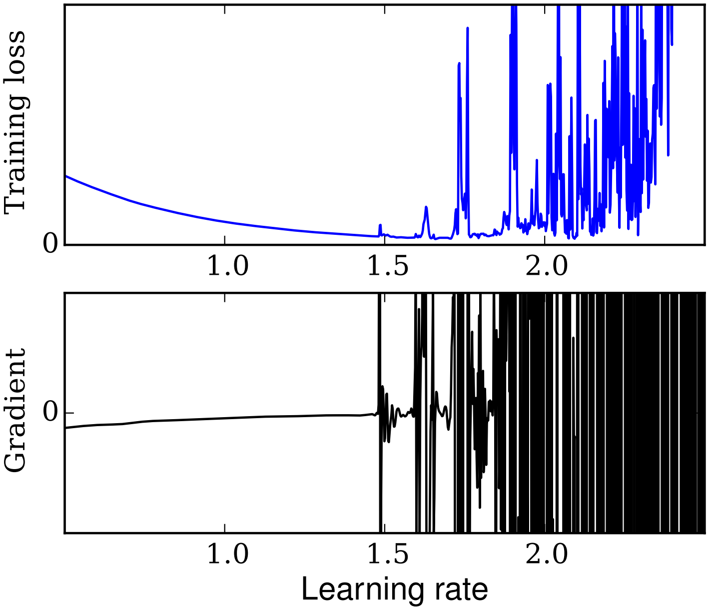
Optimizing training data

$$\frac{\partial Loss\left(D_{val}, \mathbf{x}_{init}, \alpha, \beta, D_{train}, \text{reg}\right)}{\partial D_{train}}$$

- Training set of size 10 with fixed labels on MNIST
- Started from blank images



Limitations: Chaotic learning dynamics



Summary

- Can compute gradients of learning procedures
- Reversing learning saves memory
- Can optimize thousands of hyperparameters

Summary

- Can compute gradients of learning procedures
- Reversing learning saves memory
- Can optimize thousands of hyperparameters

Thanks!