Gradient-based optimization of hyperparameters

David Duvenaud, Dougal Maclaurin, Ryan Adams

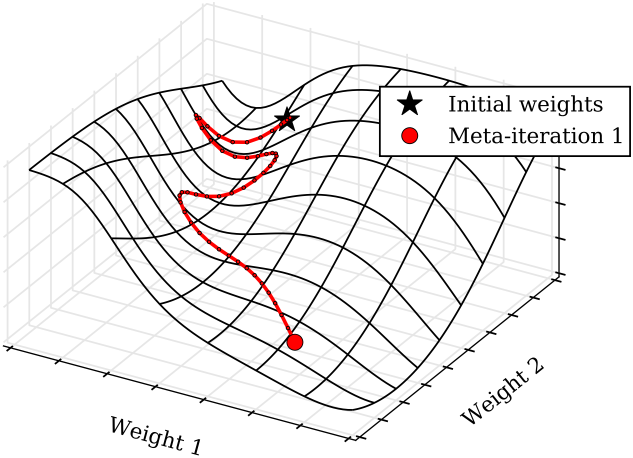
Harvard University

Motivation

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

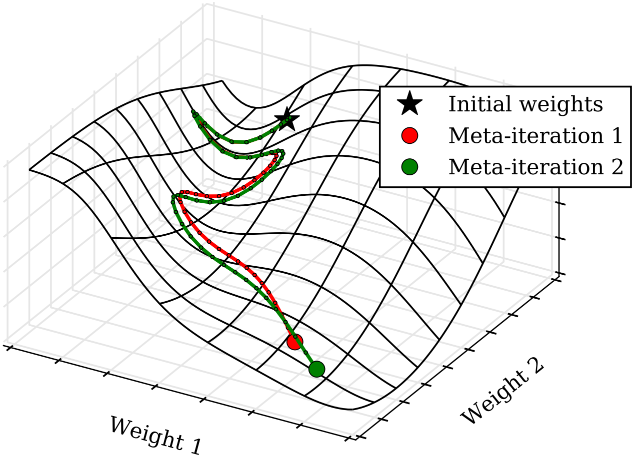
Optimizing optimization

$$\theta_{final} = \mathsf{SGD}\left(\theta_{init}, \text{learn rate}, \text{momentum}, \nabla\text{Loss}(\theta, \text{reg}, \textit{Data})\right)$$



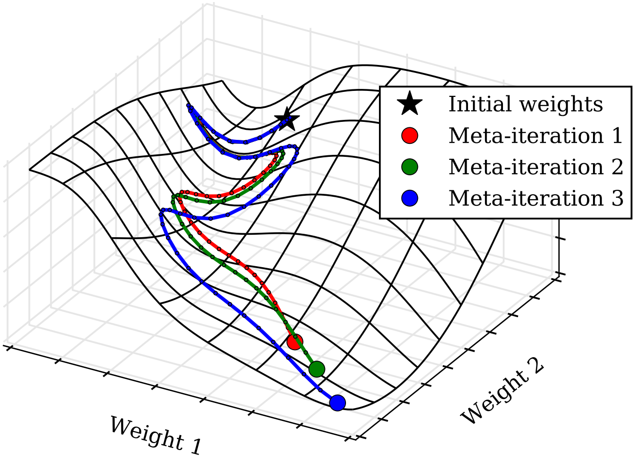
Optimizing optimization

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



Optimizing optimization

$$\theta_{final} = \mathsf{SGD}\left(\theta_{init}, \text{learn rate}, \text{momentum}, \nabla\text{Loss}(\theta, \text{reg}, \mathit{Data})\right)$$



Autograd features

github.com/HIPS/autograd

- loops, branching, recursion
- arrays, tuples, lists, dicts...
- derivatives of derivatives

Autograd features

`github.com/HIPS/autograd`

- loops, branching, recursion
- arrays, tuples, lists, dicts...
- derivatives of derivatives

used for...

- Population genetics simulations
- Inference libraries
- Protein folding simulations
- Material thermodynamics simulations
- Optimization on manifolds
- Neural Turing machines

Autograd examples

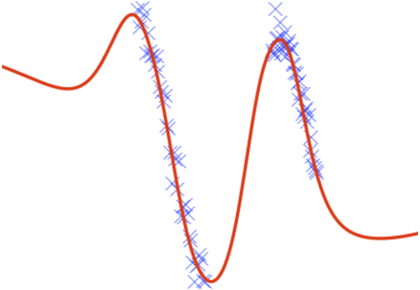
```
import autograd.numpy as np
from autograd import grad

def predict(weights, inputs):
    for W, b in weights:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs

def init_params(scale, sizes):
    return [(npr.randn(nin, out) * scale,
             npr.randn(out) * scale)
            for nin, out in zip(sizes[:-1]

def logprob_func(weights, inputs, targets)
    preds = predict(weights, inputs)
    return np.sum((preds - targets)**2)

gradient_func = grad(logprob_func)
```



Autograd examples

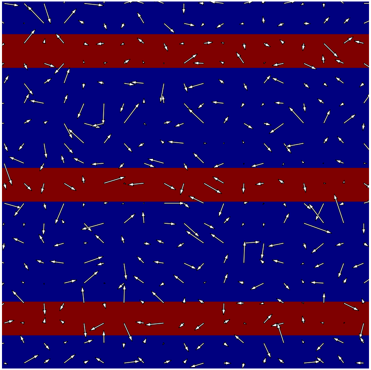
```
def project(vx, vy):
    # Project the velocity field to be approximately mass—conserving,
    # using a few iterations of Gauss—Seidel.
    p = np.zeros(vx.shape)
    h = 1.0/vx.shape[0]
    div = -0.5 * h * (np.roll(vx, -1, axis=0) - np.roll(vx, 1, axis=0)
                    + np.roll(vy, -1, axis=1) - np.roll(vy, 1, axis=1))
    for k in range(10):
        p = (div + np.roll(p, 1, axis=0) + np.roll(p, -1, axis=0)
                 + np.roll(p, 1, axis=1) + np.roll(p, -1, axis=1))/4.0
    vx -= 0.5*(np.roll(p, -1, axis=0) - np.roll(p, 1, axis=0))/h
    vy -= 0.5*(np.roll(p, -1, axis=1) - np.roll(p, 1, axis=1))/h
    return vx, vy

def advect(f, vx, vy):
    # Move field f according to x and y velocities (u and v)
    # using an implicit Euler integrator.
    rows, cols = f.shape
    cell_ys, cell_xs = np.meshgrid(np.arange(rows),
                                   np.arange(cols))
    center_xs = (cell_xs - vx).ravel()
    center_ys = (cell_ys - vy).ravel()

    # Compute indices of source cells.
    left_ix = np.floor(center_xs).astype(int)
    top_ix  = np.floor(center_ys).astype(int)
    rw = center_xs - left_ix
    bw = center_ys - top_ix
    left_ix  = np.mod(left_ix,     rows)
    right_ix = np.mod(left_ix + 1, rows)
    top_ix   = np.mod(top_ix,      cols)
    bot_ix   = np.mod(top_ix  + 1, cols)

    flat_f = (1 - rw) * ((1 - bw)*f[left_ix,  top_ix] \
                            + bw*f[left_ix,  bot_ix]) \
                 + rw * ((1 - bw)*f[right_ix, top_ix] \
                            + bw*f[right_ix, bot_ix])
    return np.reshape(flat_f, (rows, cols))

def simulate(vx, vy, smoke, num_time_steps):
    for t in range(num_time_steps):
        vx_updated = advect(vx, vx, vy)
        vy_updated = advect(vy, vx, vy)
        vx, vy = project(vx_updated, vy_updated)
        smoke = advect(smoke, vx, vy)
    return smoke, frame_list
```



Autograd examples

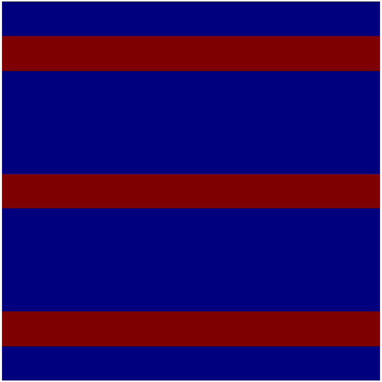
```python
def project(vx, vy):
    # Project the velocity field to be approximately mass-conserving,
    # using a few iterations of Gauss-Seidel.
    p = np.zeros(vx.shape)
    h = 1.0/vx.shape[0]
    div = -0.5 * h * (np.roll(vx, -1, axis=0) - np.roll(vx, 1, axis=0)
                    + np.roll(vy, -1, axis=1) - np.roll(vy, 1, axis=1))
    for k in range(10):
        p = (div + np.roll(p, 1, axis=0) + np.roll(p, -1, axis=0)
                 + np.roll(p, 1, axis=1) + np.roll(p, -1, axis=1))/4.0
    vx -= 0.5*(np.roll(p, -1, axis=0) - np.roll(p, 1, axis=0))/h
    vy -= 0.5*(np.roll(p, -1, axis=1) - np.roll(p, 1, axis=1))/h
    return vx, vy

def advect(f, vx, vy):
    # Move field f according to x and y velocities (u and v)
    # using an implicit Euler integrator.
    rows, cols = f.shape
    cell_ys, cell_xs = np.meshgrid(np.arange(rows),
                                   np.arange(cols))
    center_xs = (cell_xs - vx).ravel()
    center_ys = (cell_ys - vy).ravel()

    # Compute indices of source cells.
    left_ix = np.floor(center_xs).astype(int)
    top_ix  = np.floor(center_ys).astype(int)
    rw = center_xs - left_ix
    bw = center_ys - top_ix
    left_ix  = np.mod(left_ix,     rows)
    right_ix = np.mod(left_ix + 1, rows)
    top_ix   = np.mod(top_ix,      cols)
    bot_ix   = np.mod(top_ix  + 1, cols)

    flat_f = (1 - rw) * ((1 - bw)*f[left_ix,  top_ix] \
                         + bw*f[left_ix,  bot_ix]) \
             + rw * ((1 - bw)*f[right_ix, top_ix] \
                         + bw*f[right_ix, bot_ix])
    return np.reshape(flat_f, (rows, cols))

def simulate(vx, vy, smoke, num_time_steps):
    for t in range(num_time_steps):
        vx_updated = advect(vx, vx, vy)
        vy_updated = advect(vy, vx, vy)
        vx, vy = project(vx_updated, vy_updated)
        smoke = advect(smoke, vx, vy)
    return smoke, frame_list
```



Autograd examples

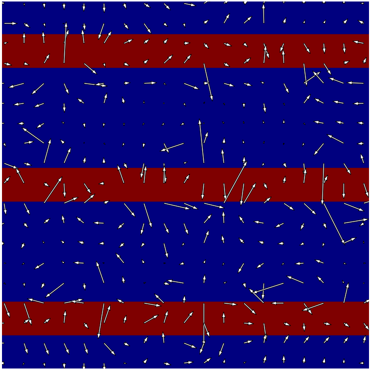
```
def project(vx, vy):
    # Project the velocity field to be approximately mass-conserving,
    # using a few iterations of Gauss-Seidel.
    p = np.zeros(vx.shape)
    h = 1.0/vx.shape[0]
    div = -0.5 * h * (np.roll(vx, -1, axis=0) - np.roll(vx, 1, axis=0)
                    + np.roll(vy, -1, axis=1) - np.roll(vy, 1, axis=1))
    for k in range(10):
        p = (div + np.roll(p, 1, axis=0) + np.roll(p, -1, axis=0)
                 + np.roll(p, 1, axis=1) + np.roll(p, -1, axis=1))/4.0
    vx -= 0.5*(np.roll(p, -1, axis=0) - np.roll(p, 1, axis=0))/h
    vy -= 0.5*(np.roll(p, -1, axis=1) - np.roll(p, 1, axis=1))/h
    return vx, vy

def advect(f, vx, vy):
    # Move field f according to x and y velocities (u and v)
    # using an implicit Euler integrator.
    rows, cols = f.shape
    cell_ys, cell_xs = np.meshgrid(np.arange(rows),
                                   np.arange(cols))
    center_xs = (cell_xs - vx).ravel()
    center_ys = (cell_ys - vy).ravel()

    # Compute indices of source cells.
    left_ix = np.floor(center_xs).astype(int)
    top_ix  = np.floor(center_ys).astype(int)
    rw = center_xs - left_ix
    bw = center_ys - top_ix
    left_ix  = np.mod(left_ix,     rows)
    right_ix = np.mod(left_ix + 1, rows)
    top_ix   = np.mod(top_ix,      cols)
    bot_ix   = np.mod(top_ix  + 1, cols)

    flat_f = (1 - rw) * ((1 - bw)*f[left_ix,  top_ix] \
                         + bw*f[left_ix,  bot_ix]) \
             + rw * ((1 - bw)*f[right_ix, top_ix] \
                         + bw*f[right_ix, bot_ix])
    return np.reshape(flat_f, (rows, cols))

def simulate(vx, vy, smoke, num_time_steps):
    for t in range(num_time_steps):
        vx_updated = advect(vx, vx, vy)
        vy_updated = advect(vy, vx, vy)
        vx, vy = project(vx_updated, vy_updated)
        smoke = advect(smoke, vx, vy)
    return smoke, frame_list
```



Autograd examples

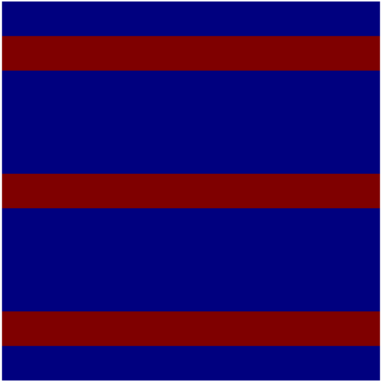
```
def project(vx, vy):
    # Project the velocity field to be approximately mass-conserving,
    # using a few iterations of Gauss-Seidel.
    p = np.zeros(vx.shape)
    h = 1.0/vx.shape[0]
    div = -0.5 * h * (np.roll(vx, -1, axis=0) - np.roll(vx, 1, axis=0)
                    + np.roll(vy, -1, axis=1) - np.roll(vy, 1, axis=1))
    for k in range(10):
        p = (div + np.roll(p, 1, axis=0) + np.roll(p, -1, axis=0)
                 + np.roll(p, 1, axis=1) + np.roll(p, -1, axis=1))/4.0
    vx -= 0.5*(np.roll(p, -1, axis=0) - np.roll(p, 1, axis=0))/h
    vy -= 0.5*(np.roll(p, -1, axis=1) - np.roll(p, 1, axis=1))/h
    return vx, vy

def advect(f, vx, vy):
    # Move field f according to x and y velocities (u and v)
    # using an implicit Euler integrator.
    rows, cols = f.shape
    cell_ys, cell_xs = np.meshgrid(np.arange(rows),
                                   np.arange(cols))
    center_xs = (cell_xs - vx).ravel()
    center_ys = (cell_ys - vy).ravel()

    # Compute indices of source cells.
    left_ix = np.floor(center_xs).astype(int)
    top_ix  = np.floor(center_ys).astype(int)
    rw = center_xs - left_ix
    bw = center_ys - top_ix
    left_ix  = np.mod(left_ix,     rows)
    right_ix = np.mod(left_ix + 1, rows)
    top_ix   = np.mod(top_ix,      cols)
    bot_ix   = np.mod(top_ix  + 1, cols)

    flat_f = (1 - rw) * ((1 - bw)*f[left_ix,  top_ix] \
                             + bw*f[left_ix,  bot_ix]) \
                 + rw * ((1 - bw)*f[right_ix, top_ix] \
                             + bw*f[right_ix, bot_ix])
    return np.reshape(flat_f, (rows, cols))

def simulate(vx, vy, smoke, num_time_steps):
    for t in range(num_time_steps):
        vx_updated = advect(vx, vx, vy)
        vy_updated = advect(vy, vx, vy)
        vx, vy = project(vx_updated, vy_updated)
        smoke = advect(smoke, vx, vy)
    return smoke, frame_list
```



Examples

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

Examples

Examples

[redacted]

[redacted]

[redacted]

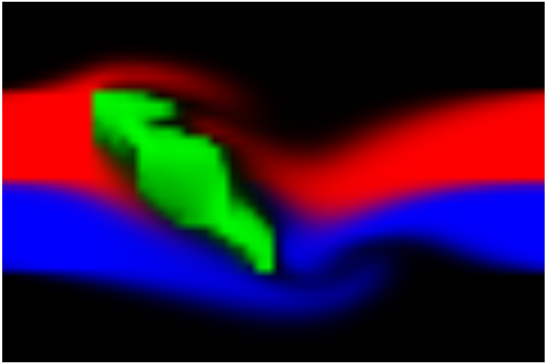
[redacted]

[redacted]

[redacted]

Examples

More fun with fluid simuliations



Can optimize any objective!

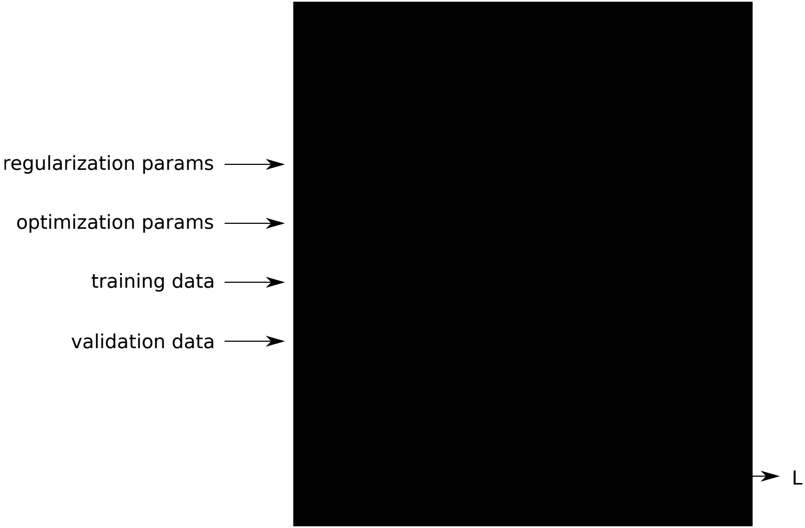
see also *Fluid control using the adjoint method*, Antoine McNamara, Adrien Treuille, Zoran Popovic, Jos Stam, 2004

Gradient-based optimization scales with dimension

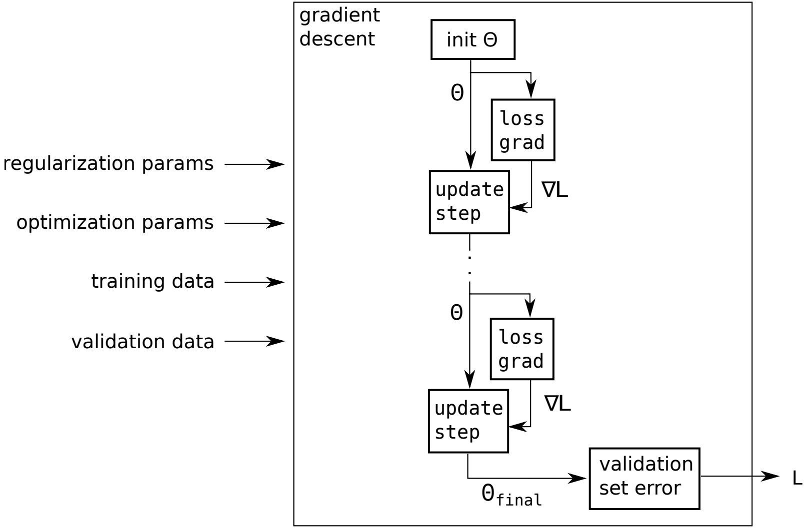
With reverse-mode differentiation (backprop):

Expression	Time cost	Scalars returned
$f(\mathbf{x})$	1	1
$\nabla f(\mathbf{x})$	~ 2	D
$\mathbf{v}^T \nabla \nabla^T f(\mathbf{x})$	~ 4	D
$\nabla \nabla^T f(\mathbf{x})$	$\sim 4D$	D^2

Can we optimize optimization itself?

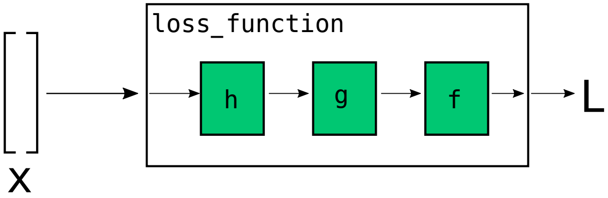


Can we optimize optimization itself?

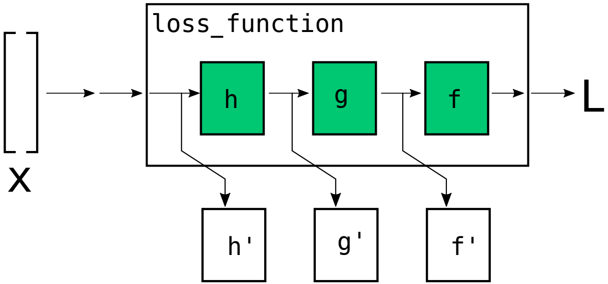


Technical challenge: memory

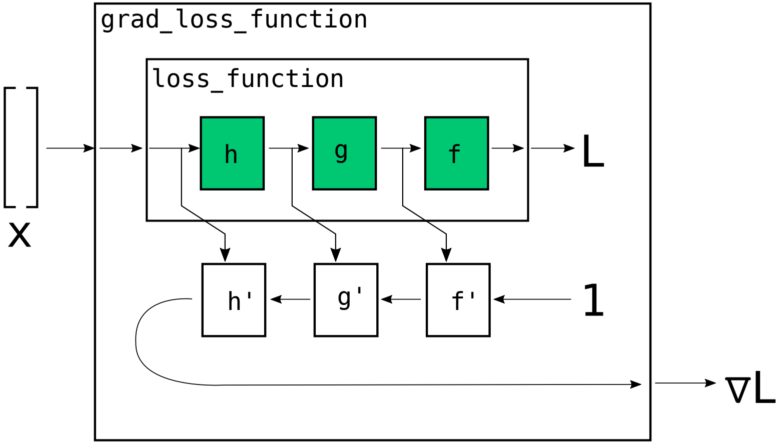
Technical challenge: memory



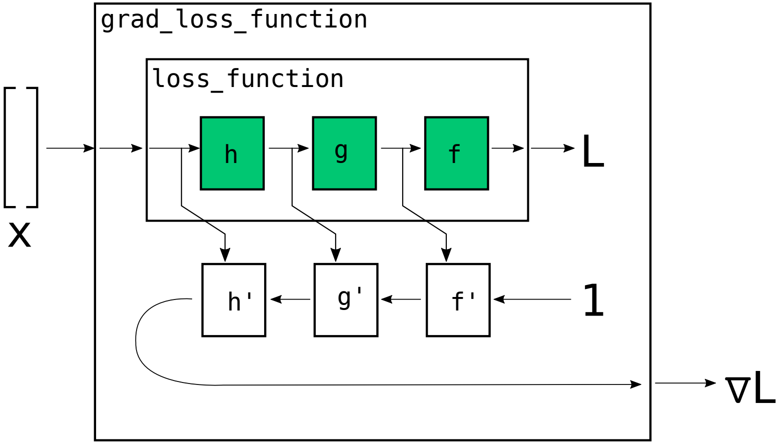
Technical challenge: memory



Technical challenge: memory

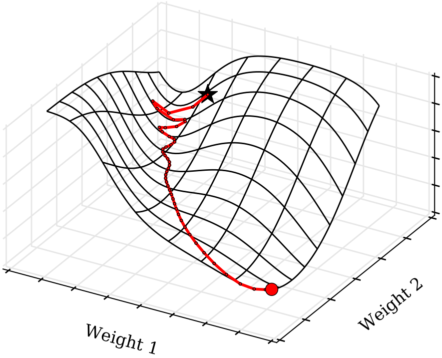


Technical challenge: memory

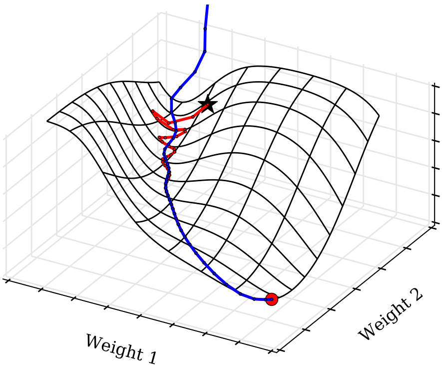


But only need LIFO access!

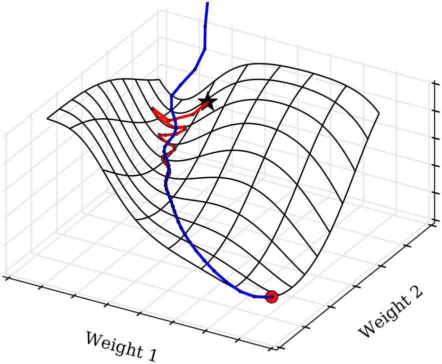
Let's run gradient *ascent*



Let's run gradient *ascent* – what happened?!



Let's run gradient *ascent* – what happened?!



- Reversed dynamics loses information

A closer look at gradient descent with momentum

Forward update rule:

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

Reverse update rule:

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

A closer look at gradient descent with momentum

Forward update rule:

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

Reverse update rule:

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

- Each iteration destroys $\log_2 \beta$ bits per parameter

A closer look at gradient descent with momentum

Forward update rule:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \mathbf{v}_t$$
$$\mathbf{v}_{t+1} \leftarrow \beta \mathbf{v}_t - \nabla L\left(\theta_{t+1}\right)$$

Reverse update rule:

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

- Each iteration destroys $\log_2 \beta$ bits per parameter
- Switch to fixed-precision for parameters
- Push lost information to buffer, restore on way back
- When $\beta = 0.9$, memory savings is 200X

A closer look at gradient descent with momentum

Forward update rule:

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

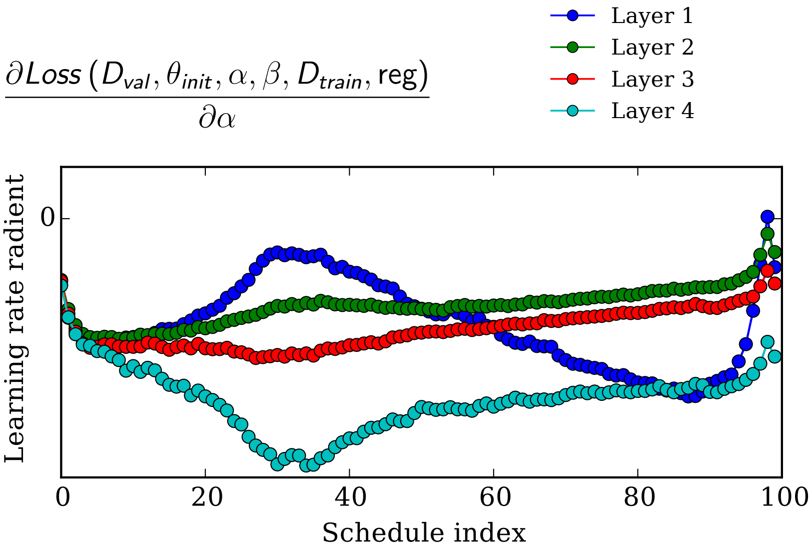
Reverse update rule:

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

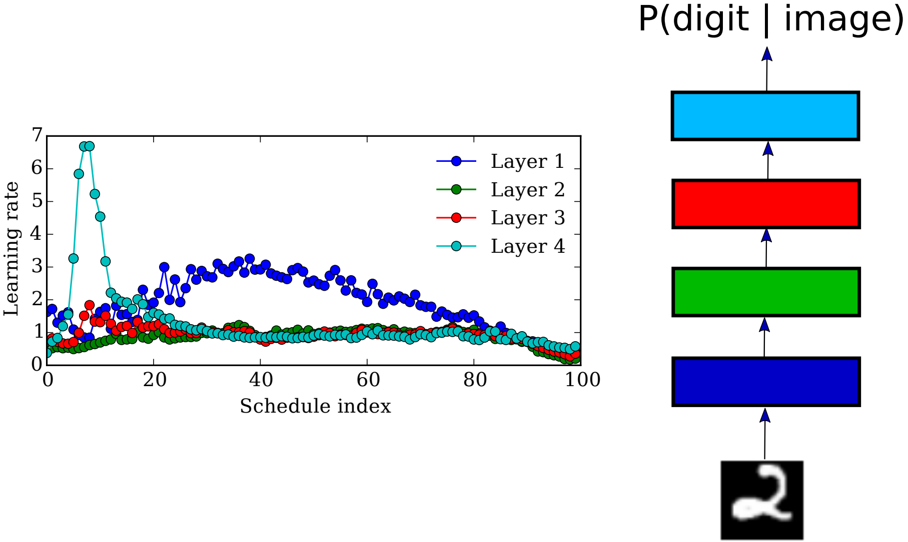
- Each iteration destroys $\log_2 \beta$ bits per parameter
- Switch to fixed-precision for parameters
- Push lost information to buffer, restore on way back
- When $\beta = 0.9$, memory savings is 200X
- More general solution: reverse model + arithmetic coding

Learning rate gradients

$$\frac{\partial Loss\left(D_{val}, \theta_{init}, \alpha, \beta, D_{train}, \mathrm{reg}\right)}{\partial \alpha}$$

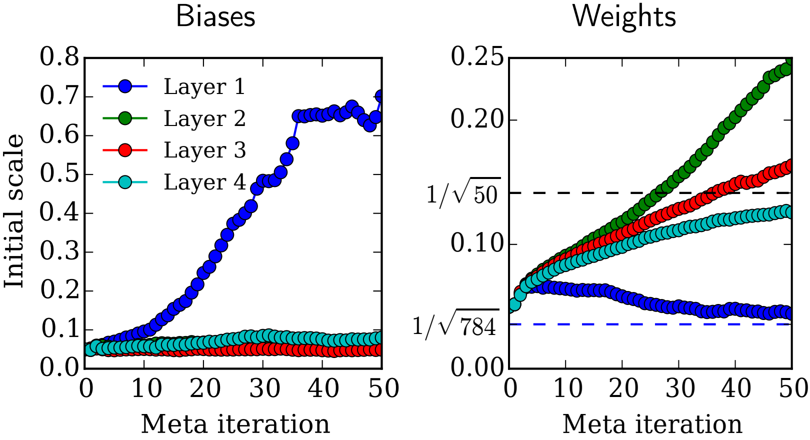


Optimized training schedules



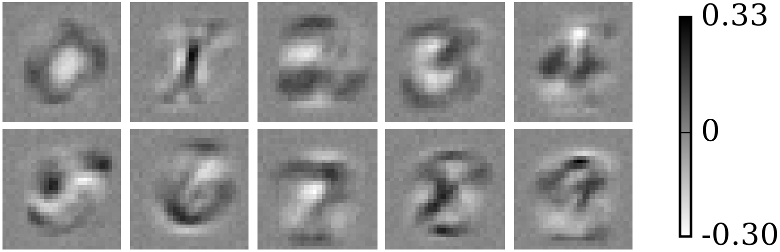
Optimizing initialization scales

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

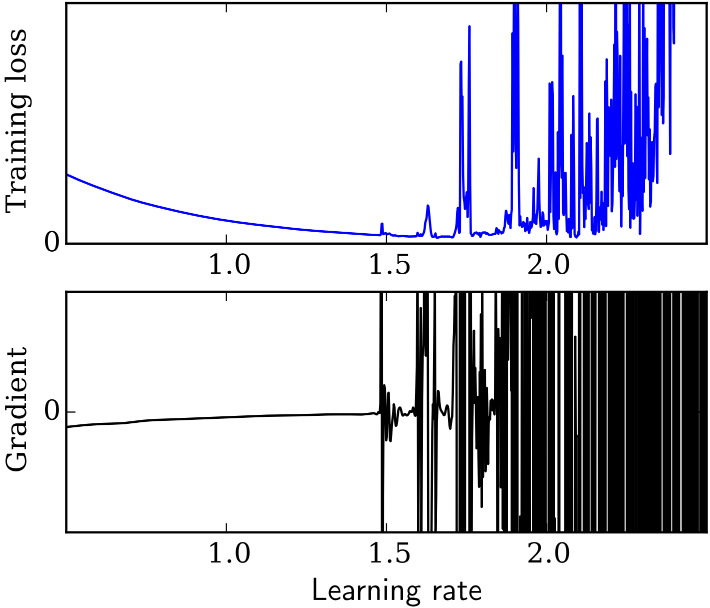


Optimizing training data

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



Limitations: Chaotic learning dynamics



A more general memory-efficient framework

Given reverse model $p(\mathsf{x}_t|\mathsf{x}_{t+1}) = f_\theta(\mathsf{x}_{t+1})$
Forward update rule:

$$\mathsf{x}_{t+1} \leftarrow f(\mathsf{x}_t)$$
$$\text{tape}_t \leftarrow \text{encode } \mathsf{x}_t \text{ using } \quad p(\mathsf{x}_t|\mathsf{x}_{t+1})$$

Reverse update rule:

$$\mathsf{x}_t \leftarrow \text{decode using } \quad p(\mathsf{x}_t|\mathsf{x}_{t+1}), \text{and tape}_t$$

A more general memory-efficient framework

Given reverse model $p(\mathbf{x}_t|\mathbf{x}_{t+1}) = f_\theta(\mathbf{x}_{t+1})$
Forward update rule:

$$\mathbf{x}_{t+1} \leftarrow f(\mathbf{x}_t)$$
$$\text{tape}_t \leftarrow \text{encode } \mathbf{x}_t \text{ using } \quad p(\mathbf{x}_t|\mathbf{x}_{t+1})$$

Reverse update rule:

$$\mathbf{x}_t \leftarrow \text{decode using } \quad p(\mathbf{x}_t|\mathbf{x}_{t+1}), \text{and tape}_t$$

- Can train model offline, or online with updates/downdates

A more general memory-efficient framework

Given reverse model $p(\mathsf{x}_t|\mathsf{x}_{t+1}) = f_\theta(\mathsf{x}_{t+1})$
Forward update rule:

$$\mathsf{x}_{t+1} \leftarrow f(\mathsf{x}_t)$$
$$\text{tape}_t \leftarrow \text{encode } \mathsf{x}_t \text{ using } \quad p(\mathsf{x}_t|\mathsf{x}_{t+1})$$

Reverse update rule:

$$\mathsf{x}_t \leftarrow \text{decode using } \quad p(\mathsf{x}_t|\mathsf{x}_{t+1}), \text{and tape}_t$$

- Can train model offline, or online with updates/downdates
- Can combine with checkpointing

A more general memory-efficient framework

Given reverse model $p(\mathbf{x}_t|\mathbf{x}_{t+1}) = f_\theta(\mathbf{x}_{t+1})$
Forward update rule:

$$\mathbf{x}_{t+1} \leftarrow f(\mathbf{x}_t)$$
$$\text{tape}_t \leftarrow \text{encode } \mathbf{x}_t \text{ using } \quad p(\mathbf{x}_t|\mathbf{x}_{t+1})$$

Reverse update rule:

$$\mathbf{x}_t \leftarrow \text{decode using } \quad p(\mathbf{x}_t|\mathbf{x}_{t+1}), \text{and tape}_t$$

- Can train model offline, or online with updates/downdates
- Can combine with checkpointing
- Memory savings depends on accuracy of reverse model

Collaborators and more ideas

Dougal Maclaurin, Ryan P. Adams

Collaborators and more ideas

Dougal Maclaurin, Ryan P. Adams

- Weather control

Collaborators and more ideas

Dougal Maclaurin, Ryan P. Adams

- Weather control
- 3D printing in a swimming pool

Collaborators and more ideas

Dougal Maclaurin, Ryan P. Adams

- Weather control
- 3D printing in a swimming pool
- Inkjet without nozzles

Collaborators and more ideas

Dougal Maclaurin, Ryan P. Adams

- Weather control
- 3D printing in a swimming pool
- Inkjet without nozzles
- Generative design

Collaborators and more ideas

Dougal Maclaurin, Ryan P. Adams

- Weather control
- 3D printing in a swimming pool
- Inkjet without nozzles
- Generative design

Thanks!

Extra Slides

How to code a Hessian-vector product?

```python
def hvp(func):
    def vector_dot_grad(arg, vector):
        return np.dot(vector, grad(func)(arg))
    return grad(vector_dot_grad)
```

- hvp(f)(x, v) returns $\mathbf{v}^T \nabla_{\mathbf{x}} \nabla_{\mathbf{x}}^T f(\mathbf{x})$
- No explicit Hessian
- Can construct higher-order operators easily

Most Numpy functions implemented

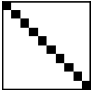
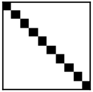
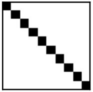
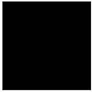
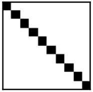
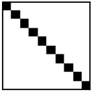
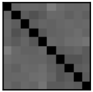
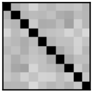
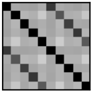
Complex & Fourier	Array	Misc	Linear Algebra	Stats
imag	atleast_1d	logsumexp	inv	std
conjugate	atleast_2d	where	norm	mean
angle	atleast_3d	einsum	det	var
real_if_close	full	sort	eigh	prod
real	repeat	partition	solve	sum
fabs	split	clip	trace	cumsum
fft	concatenate	outer	diag	norm
fftshift	roll	dot	tril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

Follow-ups

- Fu *et al.*, 2016
  - Approximate hypergradients using linear reverse path
- Luketina *et al.*, 2015
  - Approximate hypergradients using single step on validation set
- with DeepMind: memory-efficient gradients of LSTMs
- with Hugo Larochelle: training on streaming datafeeds

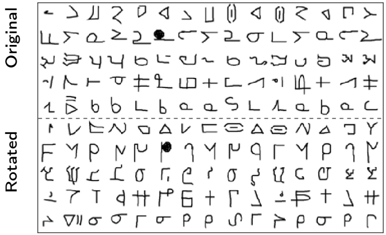
Optimizing architecture

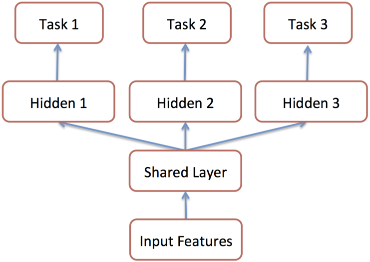
Matrices enforce weight-sharing between tasks

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

Architecture is regularization

Omniglot dataset





Reverse-mode differentiation of SGD

Stochastic Gradient Descent

input: initial $\theta_{\mathbf{1}}$, decays $\boldsymbol{\beta}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\theta, \boldsymbol{\theta}, t)$
initialize $\mathbf{v_1} = \mathbf{0}$
for $t = 1$ **to** T **do**
  $\mathbf{g}_t = \nabla_\theta L(\theta_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
  $\mathbf{v}_{t+\mathbf{1}} = \beta_t \mathbf{v}_t - \mathbf{g}_t$ ▷ update velocity
  $\theta_{t+\mathbf{1}} = \theta_t + \alpha_t \mathbf{v}_t$ ▷ update position
output trained parameters θ_T

Reverse-Mode Gradient of SGD

input: θ_T, $\mathbf{v}_T$, $\boldsymbol{\beta}$, $\boldsymbol{\alpha}$, train loss $L(\theta, \boldsymbol{\theta}, t)$, loss $f(\theta)$
initialize $d\mathbf{v} = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$, $d\alpha_t = \mathbf{0}$, $d\beta = \mathbf{0}$
initialize $d\theta = \nabla_\theta f(\theta_T)$
for $t = T$ **counting down to** 1 **do**
  $d\alpha_t = d\theta^{\mathbf{T}} \mathbf{v}_t$
  $\theta_{t-\mathbf{1}} = \theta_t - \alpha_t \mathbf{v}_t$ ▷ downdate position
  $\mathbf{g}_t = \nabla_\theta L(\theta_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
  $\mathbf{v}_{t-\mathbf{1}} = (\mathbf{v}_t + \mathbf{g}_t)/\beta_t$ ▷ downdate velocity
  $d\mathbf{v} = d\mathbf{v} + \alpha_t d\theta$
  $d\beta_t = d\mathbf{v}^{\mathbf{T}}(\mathbf{v}_t + \mathbf{g}_t)$
  $d\theta = d\theta - d\mathbf{v} \nabla_\theta \nabla_\theta L(\theta_t, \boldsymbol{\theta}, t)$
  $d\boldsymbol{\theta} = d\boldsymbol{\theta} - d\mathbf{v} \nabla_{\boldsymbol{\theta}} \nabla_\theta L(\theta_t, \boldsymbol{\theta}, t)$
  $d\mathbf{v} = \beta_t d\mathbf{v}$
output gradient of $f(\theta_T)$ w.r.t $\theta_{\mathbf{1}}$, $\mathbf{v_1}$, $\boldsymbol{\beta}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}$

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