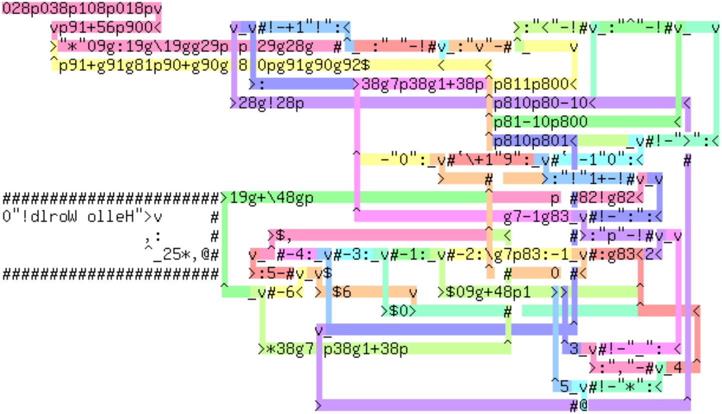
No more mini-languages: Autodiff in full-featured Python



David Duvenaud, Dougal Maclaurin, Matthew Johnson

Our awesome new world

- TensorFlow, Stan, Theano, Edward
- Only need to specify forward model
- Autodiff + inference / optimization done for you

Our awesome new world

- TensorFlow, Stan, Theano, Edward
- Only need to specify forward model
- Autodiff + inference / optimization done for you

- loops? branching? recursion? closures?

Our awesome new world

- TensorFlow, Stan, Theano, Edward
- Only need to specify forward model
- Autodiff + inference / optimization done for you

- loops? branching? recursion? closures?
- debugger?

Our awesome new world

- TensorFlow, Stan, Theano, Edward
- Only need to specify forward model
- Autodiff + inference / optimization done for you

- loops? branching? recursion? closures?
- debugger?
- a second compiler/interpreter to satisfy

Our awesome new world

- TensorFlow, Stan, Theano, Edward
- Only need to specify forward model
- Autodiff + inference / optimization done for you

- loops? branching? recursion? closures?
- debugger?
- a second compiler/interpreter to satisfy
- a new language to learn

Autograd

github.com/HIPS/autograd

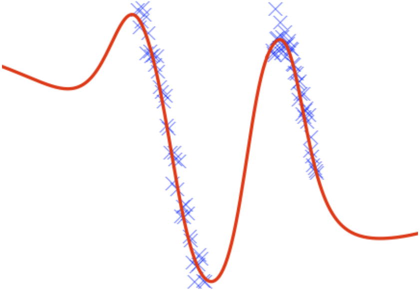
- differentiates native Python code
- handles most of Numpy + Scipy
- loops, branching, recursion, closures
- arrays, tuples, lists, dicts...
- derivatives of derivatives
- a one-function API!

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			

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], sizes[1:])]

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

gradient_func = grad(logprob_func)
```



Structured gradients

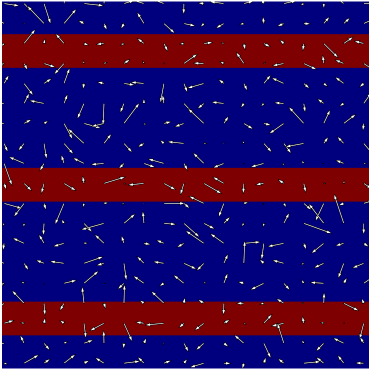
```
print(grad(logprob)(init_params, inputs, targets))

[(array([[ -5.40710861, -14.13507334, -13.94789859,  28.6188964 ]]),
   array([-17.01486765, -28.33800594, -29.77875615,  49.78987454])),
 (array([[-71.47406027, -69.1771986 ,  -7.34756845, -17.96280387],
         [ 21.90645613,  22.01415812,   2.37750145,   5.81340489],
         [-39.37357205, -38.07711948,  -4.04245488,  -9.88483908],
         [-27.00357209, -24.79890695,  -2.56954539,  -6.28235645]]),
  array([-281.99906027, -278.86794587,  -29.90316231,  -73.12033635])),
  (array([[-410.89215947],
          [ 256.31407037],
          [ -31.39182332],
          [   6.89045123]]),
  array([-1933.60342748]))]
```

How to code a Hessian-vector product?

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

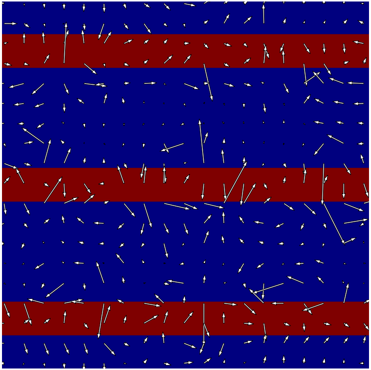
- hvp(f)(x, v) returns $\mathbf{v}^T \nabla_{\mathbf{x}} \nabla_{\mathbf{x}}^T f(\mathbf{x})$
- No explicit Hessian
- Can construct higher-order operators easily

```
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
```



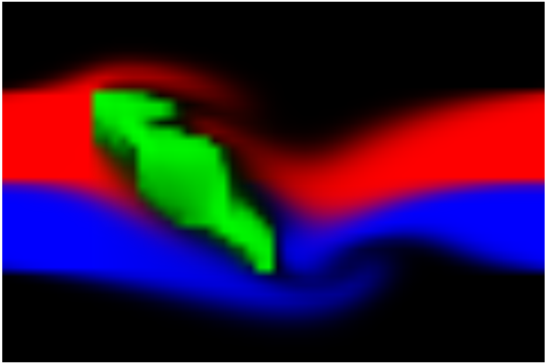
```python
def project(vx, vy):
    # Project the velocity field to be approximately mass—conserving,
    # using a few iterations of Gauss—Seidel.
    p = np.zeros(vx.shape)
    h = 1.0/vx.shape[0]
    div = -0.5 * h * (np.roll(vx, -1, axis=0) - np.roll(vx, 1, axis=0)
                    + np.roll(vy,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

```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
```



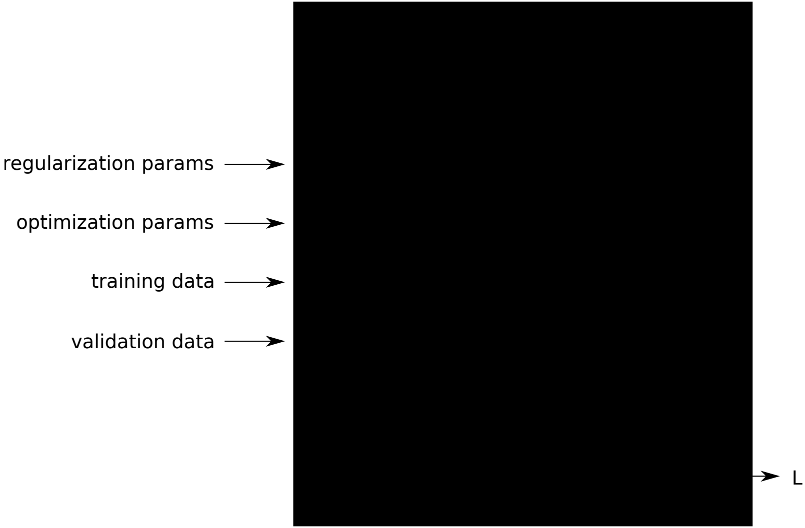
```python
def project(vx, vy):
    # Project the velocity field to be approximately mass-conserving,
    # using a few iterations of Gauss-Seidel.
    p = np.zeros(vx.shape)
    h = 1.0/vx.shape[0]
    div = -0.5 * h * (np.roll(vx, -1, axis=0) - np.roll(vx, 1, axis=0)
                    + np.roll(vy,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

More fun with fluid simulations

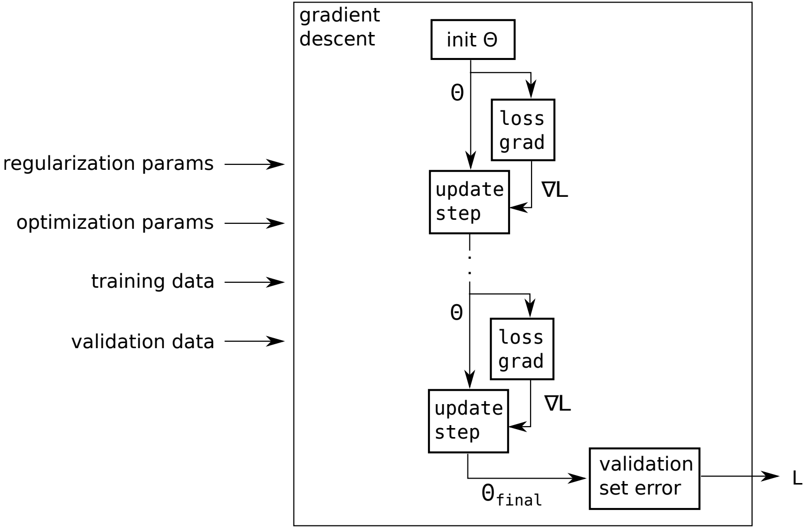


Can optimize any objective!

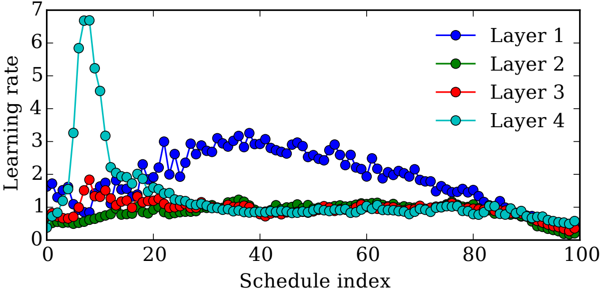
Can we optimize optimization itself?

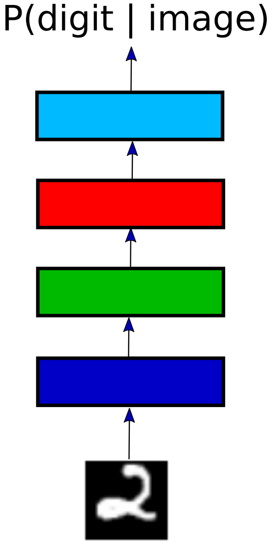


Can we optimize optimization itself?



Optimized training schedules

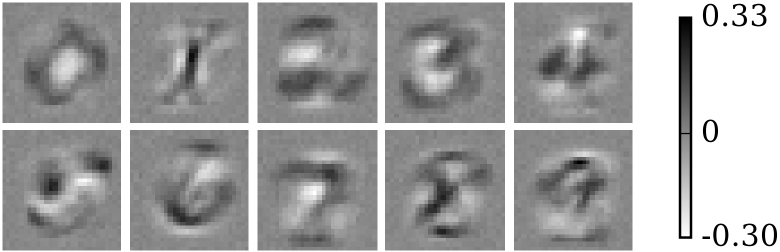




What else could we optimize?

Optimizing training data

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



Maclaurin, Duvenaud & Adams, 2015

github.com/HIPS/hypergrad

But what about inference?

Stan also provides inference routines...

But what about inference?

Stan also provides inference routines...

Ryan Adams @ryan_p_adams · 7 Nov 2015

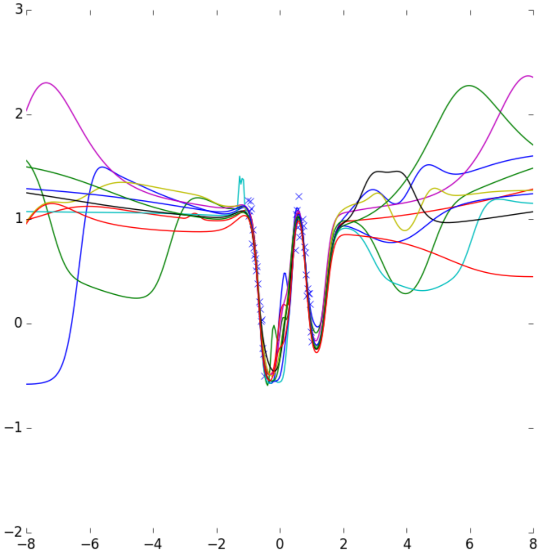
@DavidDuvenaud

```
def elbo(p, lp, D, N):
 v=exp(p[D:])
 s=randn(N,D)*sqrt(v)+p[:D]
 return mvn.entropy(0, diag(v))+mean(lp(s))
gf = grad(elbo)
```

1 7 22

... which are a tiny amount of code with autodiff!

Show Bayesian Neural Network



Collaborators

github.com/HIPS/autograd

Dougal Maclaurin

Matthew Johnson

Ryan Adams