CS207 Final Presentation

Simon Sebbagh, Dylan Randle, Adam Nitido, Paxton Maeder-York

`pip install dragongrad`

The Team

Dylan Randle

Paxton Maeder-York

Simon Sebbagh

Adam Nitido

Outline

- Background
- Implementation
- Extensions
- Summary
- Notebook Demos

QR Code to Documentation:

QR Code to Github Repository:

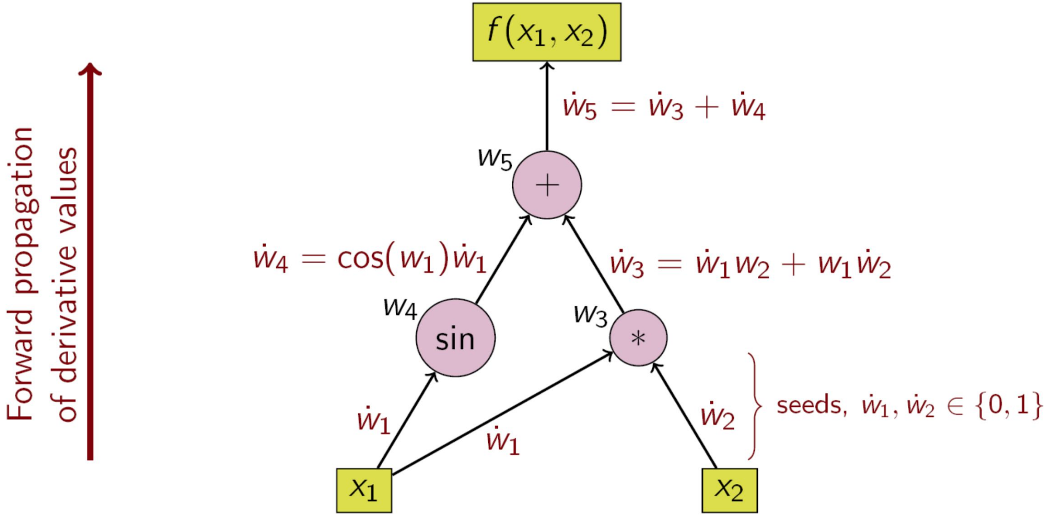
Background: Goal

Implement Automatic Differentiation

- Find Gradient
- Split complex functions
- Forward mode
- Reverse mode

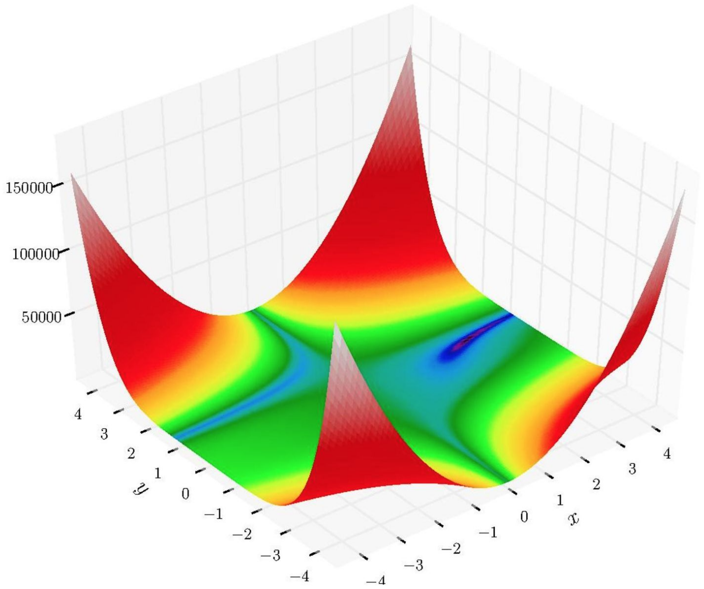
$$\mathbf{J} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \quad \cdots \quad \frac{\partial \mathbf{f}}{\partial x_n} \right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Jacobian Matrix

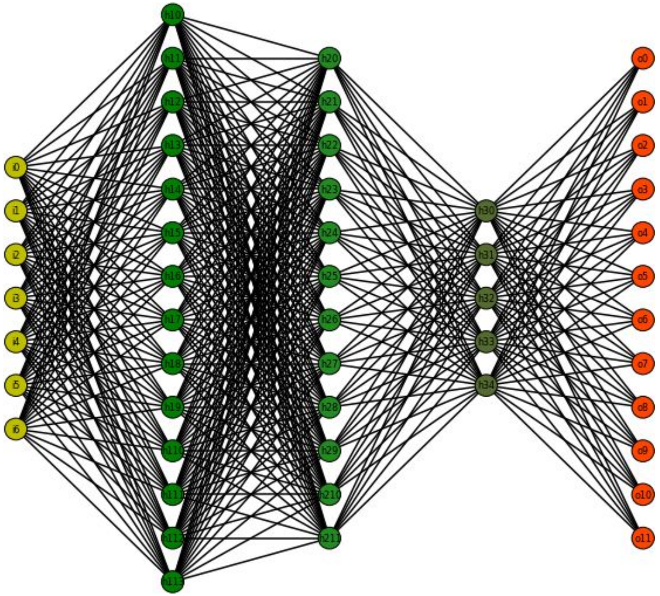


Computational Graph

Background: Use Cases



Optimization



Neural Networks

Forward Mode

Forward Mode: API Demo

```
import autograd as ad
import numpy as np
from autograd.variable import Variable

x=Variable([1,np.pi])
b1=ad.sin(x)
b2=ad.cos(x)
b3=b1+b2
b4=(b3+x)**x

b4.compute_gradients()

print(b4)
```

```
data :
[ 2.38177329 10.94058428]
grad :
[[2.76584205 0.        ]
 [0.         8.33179958]]
```

Core Abstraction: Variable

Variable	
data	-- numpy array containing the function evaluation
gradient	-- numpy array containing the gradient evalutation

- Constant : Variable with init gradient of 0

Core Abstraction: Block

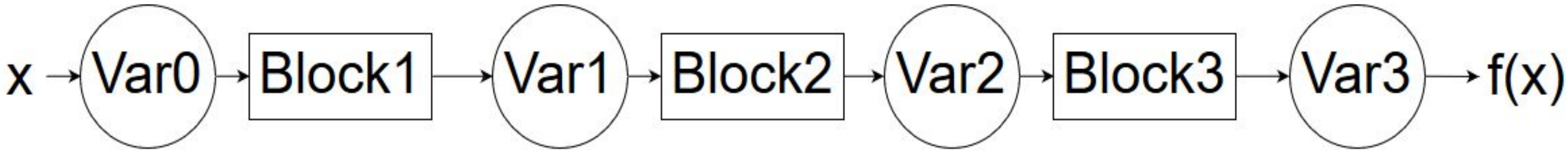
- Blocks

Block	
data_fn	- perform the actual pass on the data
get_jacobian	- return the list of jacobians of the output variable w.r. the input variables
gradient_forward	- perform the actual pass on the gradient
__call__	- return the output variable of this block, with data and gradients updated

- Not storing the whole computational graph. Gradients are computed on-the-fly

Example

- Gradient flow:



- The information is stored in Variables
- The blocks create new variables
- Each block represents a function. It is an instance of the top class Block

Multi-Variable (Naive)

```
def f(x,y,z):
    vector_variable=Variable([x,y,z])  #create the vector variable with the data of x,y and z

    #extract the relevant variables
    #the [] operator extracts both data and gradient and create a new corresponding variable
    x_var, y_var, z_var = vector_variable[0], vector_variable[1], vector_variable[2]

    output=x_var + y_var*7 - z_var

    output.compute_gradients()
    return(output)

print(f(1,3,64))
```

```
data :
[-42]
grad :
[[ 1.  7. -1.]]
```

Multi-Variable (Cleaner)

```
def f(x,y,z):
    x_var, y_var, z_var = Variable.multi_variables(x,y,z)

    output=x_var + y_var*7 - z_var
    output.compute_gradients()
    return(output)

print(f(1,3,64))
```

```
data :
[-42]
grad :
[array([[1.]]), array([[7.]]), array([[-1.]])]
```

Reverse Mode

The Node

.gradient : derivative of the output node w.r. To this node

.childrens

```
ad.set_mode('reverse')
```

```
x=Variable(2)
y=ad.sin(x)
z=x+y
```

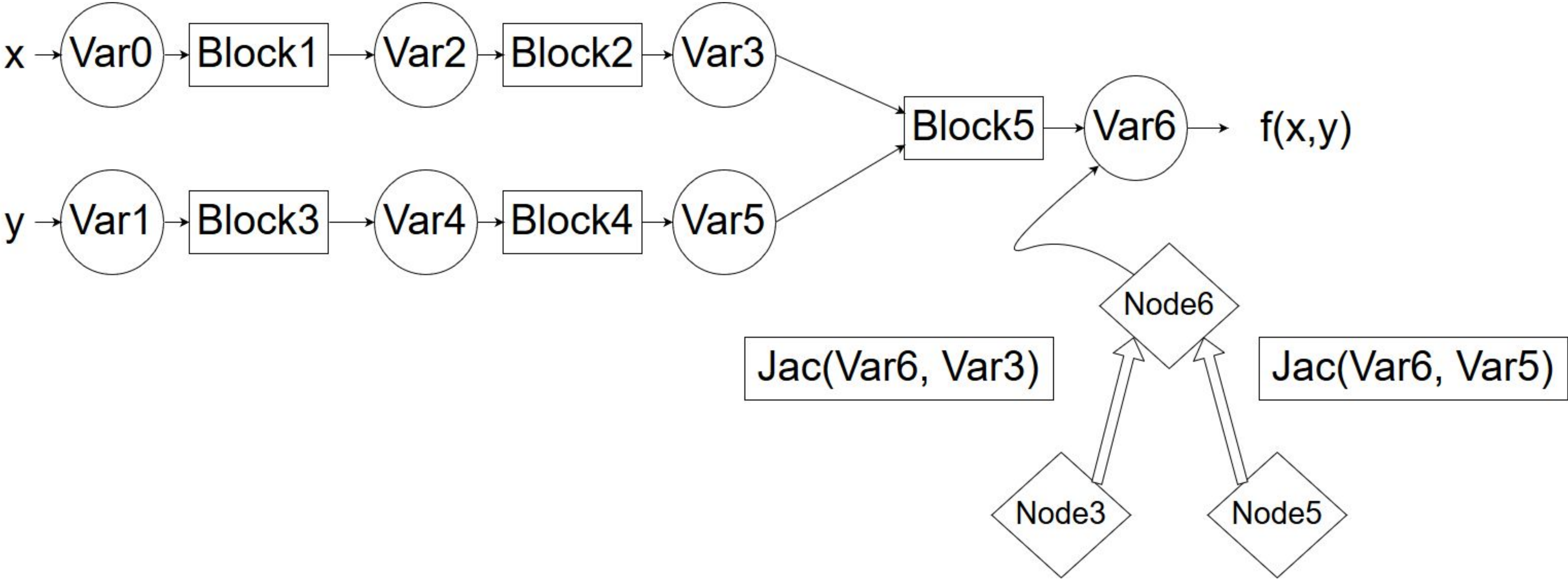
```
y.node.childrens #=[{'node':x.node, 'jacobian':cos(x.data)}]
```

```
[{'jacobian': array([[-0.41614684]]),
  'node': <autograd.node.Node at 0x19706019c18>}]
```

```
z.node.childrens #=[{'node':x.node, 'jacobian':identity}, {'node':y.node, 'jacobian':identity}]
```

```
[{'jacobian': array([[1.]]), 'node': <autograd.node.Node at 0x19706019c18>},
 {'jacobian': array([[1.]]), 'node': <autograd.node.Node at 0x19706019be0>}]
```

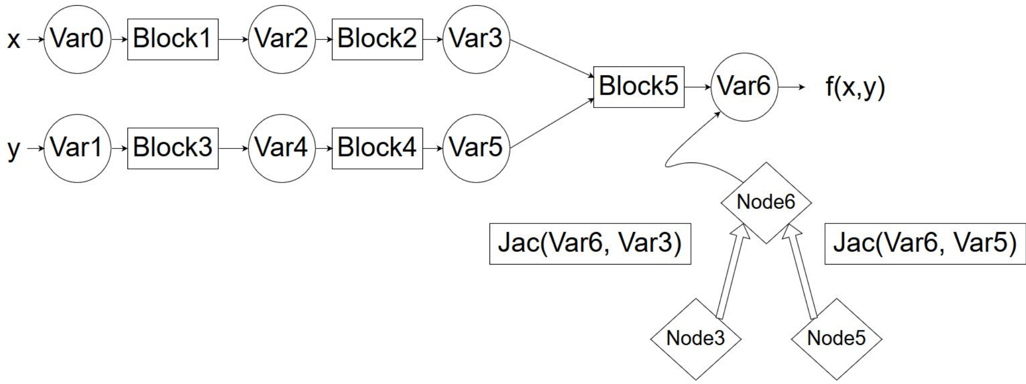
The Computational Graph



Gradient Flow

Reverse pass on the gradient

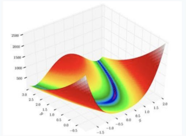
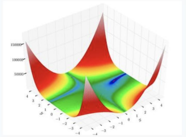
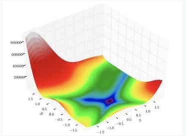
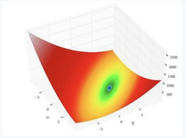
1. User calls `output_variable.compute_gradients()`
2. Function defines c_graph.output_node
3. Computational graph defines the path
4. Recursive backward() on the nodes
5. Return gradients of the input Nodes
    a. One input: Jacobian matrix
    b. Multiple inputs: List of Jacobians
6. Reset graph



Optimizers

Optimizers

- **Gradient Descent** and **Adam** optimizers
- Uses class inheritance for **extensibility**
    - Just provide a .*step*() method
- Forward and backward
- Performance analysis:
    - Across wide range of functions
    - Convergence rate
    - Wall time
    - See ***notebooks/Convergence_Results.ipynb***

Rosenbrock function		$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2\right]$
Beale function		$f(x,y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2$
Goldstein–Price function		$f(x,y) = \left[1 + (x+y+1)^2\left(19 - 14x + 3x^2 - 14y + 6xy + 3y^2\right)\right]\left[30 + (2x-3y)^2\left(18 - 32x + 12x^2 + 48y - 36xy + 27y^2\right)\right]$
Booth function		$f(x,y) = (x + 2y - 7)^2 + (2x + y - 5)^2$

Base class

```
class Optimizer():
    """
    Optimizer Base Class

    == Args ==

    loss_func (function): a function accepting a list of `params` (see below) and r
    params (array): a list of initialization parameters - these should correspond t
    lr (float): leaning rate for steps
    tol (float): tolerance for determining loss function convergence
    max_iter (int): maximumer number of steps the optimizer will run

    """
    def __init__(self, loss_func, params, lr=0.01, max_iter=100000, tol=1e-14):
        self.loss_func = loss_func
        self.params = params
        self.lr = lr
        self.max_iter = max_iter
        self.tol = tol

    def step(self):
        """
        Performs a single step of the optimizaiton
        """
        raise NotImplementedError

    def solve(self, return_steps=False):
        """
        Loop until convergence criteria is met or for max_iters
        """
        steps=[]
        count = 0
        while count < self.max_iter:

            prev_loss, prev_grad = self.loss_func(self.params)
            self.step()
            new_loss, new_grad = self.loss_func(self.params)
            if return_steps:
                steps.append(self.params)

            if abs(prev_loss - new_loss) < self.tol:
                break
            count += 1

        if return_steps:
            return (self.params, steps)
        else:
            return (self.params)
```

Adam

```python
class Adam(Optimizer):
    """
    Implements Adam Optimizer (`Adam: A Method for Stochastic Optimization`)
    """

    def __init__(self, *args, beta1=0.9, beta2=0.999, eps=1e-8, **kwargs):
        super().__init__(*args, **kwargs)
        self.beta1=beta1
        self.beta2=beta2
        self.eps=eps
        self.exp_avg=np.zeros_like(self.params)
        self.exp_avg_sq=np.zeros_like(self.params)
        self.step_count=0

    def step(self):
        # increment step count
        self.step_count += 1

        # get current loss
        loss, grad = self.loss_func(self.params)
        grad = grad[0]

        # calculate moving averages
        self.exp_avg = self.exp_avg*self.beta1 + (1-self.beta1)*grad
        self.exp_avg_sq = self.exp_avg_sq*self.beta2 + (1-self.beta2)*(grad**2)

        # perform bias correction
        bias_correction1 = self.exp_avg / (1 - self.beta1 ** self.step_count)
        bias_correction2 = self.exp_avg_sq / (1 - self.beta2 ** self.step_count)

        # calculate step size and perform step
        step_size = self.lr * bias_correction1 / (np.sqrt(bias_correction2) + self.eps)
        self.params = self.params - step_size
```

Example usage

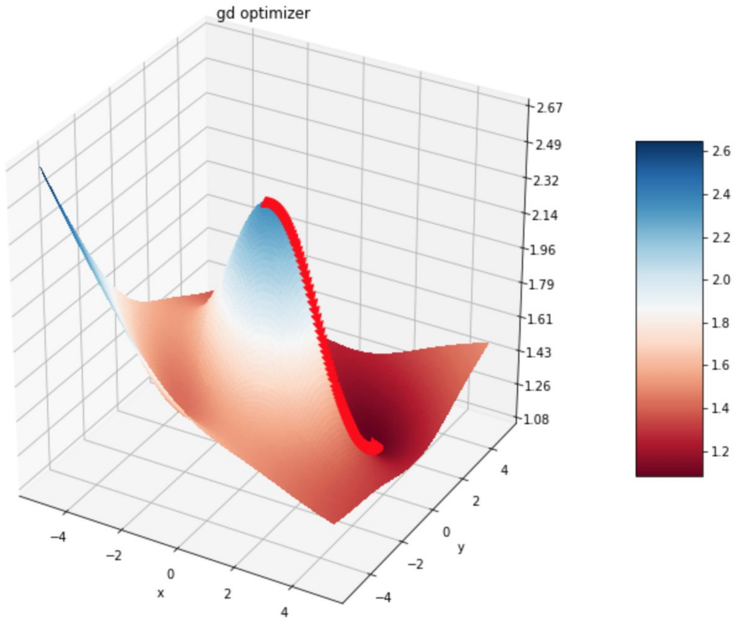
```python
def loss_function(params):
    var = av.Variable(params)
    x, y = var[0], var[1]

    b1 = ad.exp(-0.1*((x**2)+(y**2)))
    b2 = ad.cos(0.5*(x+y))
    b3 = b1*b2+0.1*(x+y)+ad.exp(0.1*(3-(x+y)))

    b3.compute_gradients()
    return (b3.data, b3.gradient)
```

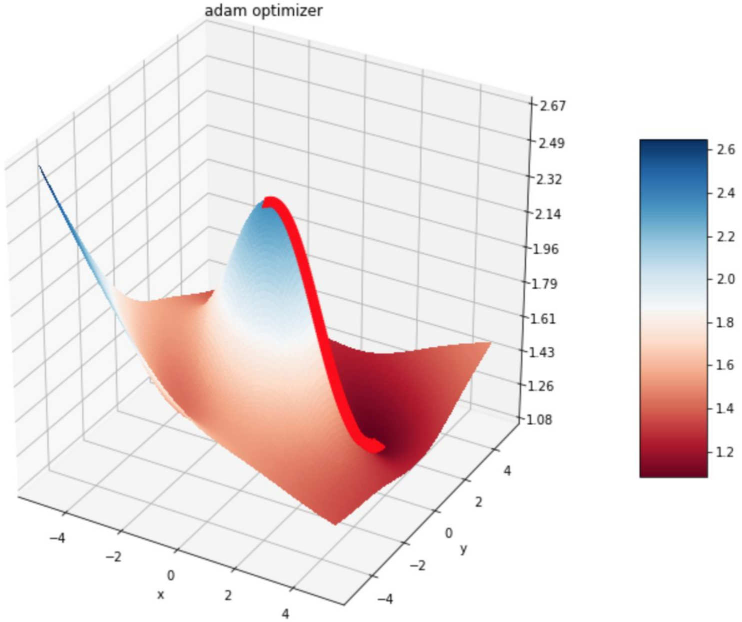
```python
optimizer = Adam(f, startingPoint, lr=lr, max_iter=iterations)
```

```python
soln, steps = optimizer.solve(return_steps=True)
```

Constructed Loss

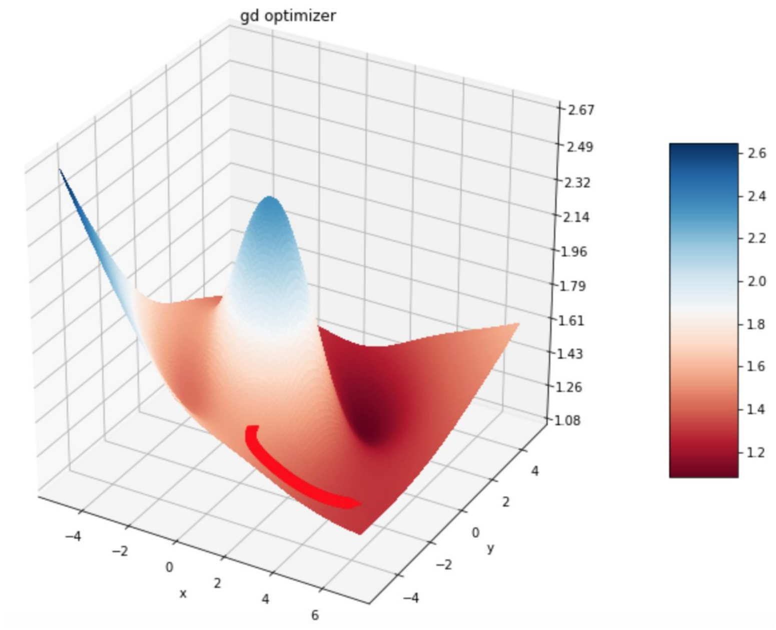
$$f(x, y) = \cos(0.5 \cdot (x + y)) \cdot e^{(-0.1 \cdot (x^2 + y^2))} + 0.1 \cdot (x + y) + e^{(0.1 \cdot (3 - (x+y))}$$

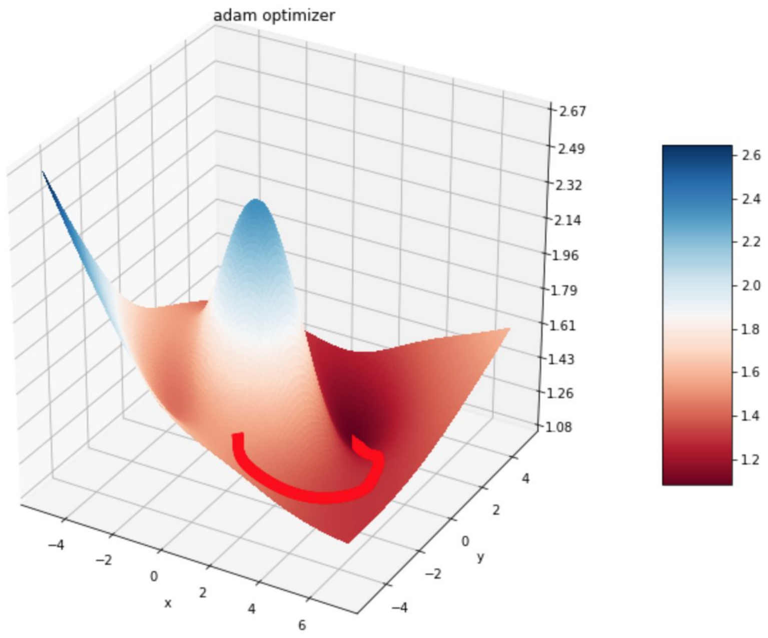




Different Starting Point

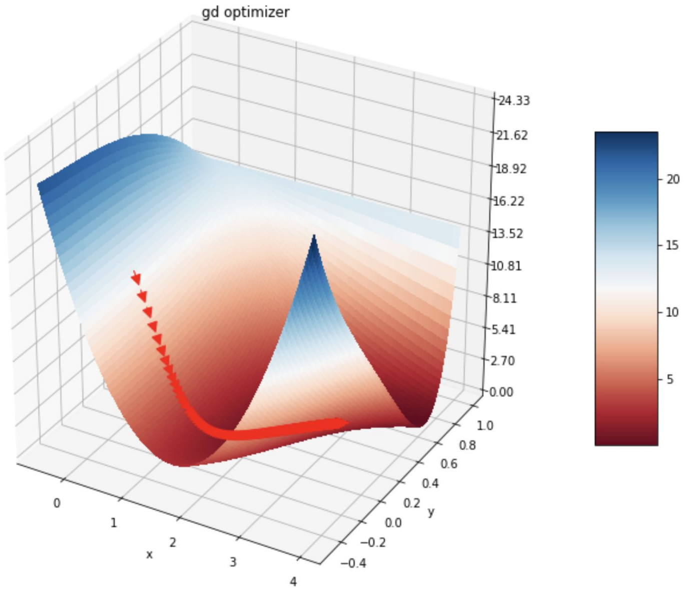
GD fails to converge, but Adam does:

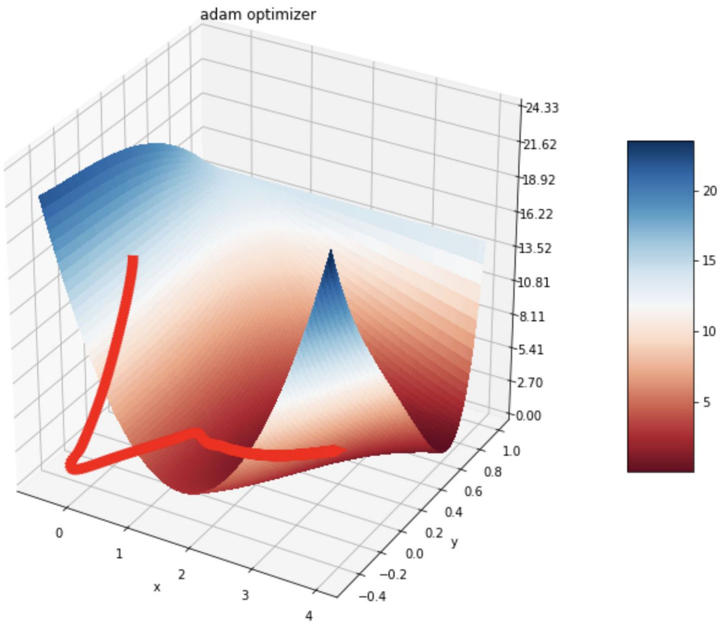




Another example: Beale Function

$$f(x,y) = (1.5 - x + x \cdot y)^2 + (2.25 - x + x \cdot y^2)^2 + (2.625 - x + x \cdot y^3)^2$$





Summary

1. Implemented Forward Mode
2. Created Optimizers
3. Built Reverse Mode

Thank you:

David Sondak

Bernard Kleynhans

Future Work

- Neural network
    - Convolutional blocks
- Medical image processing
- Help doctors discover illness in X-rays

Demos