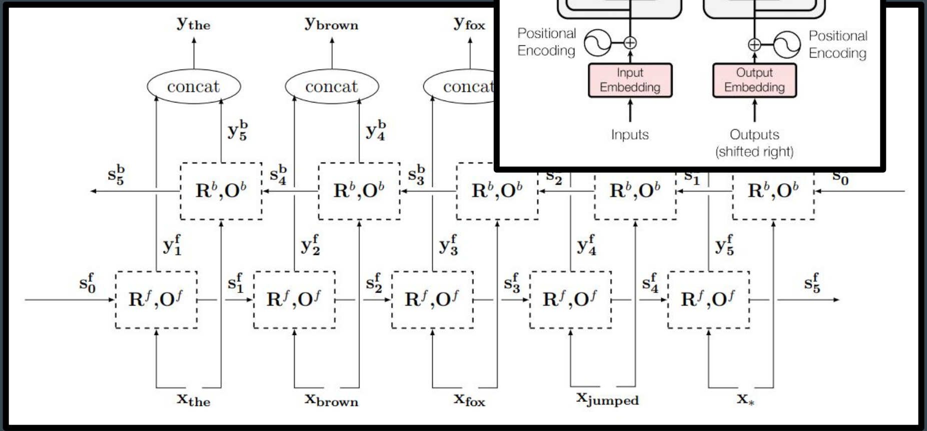
Analyzing Neural Networks with Gradient Tracing

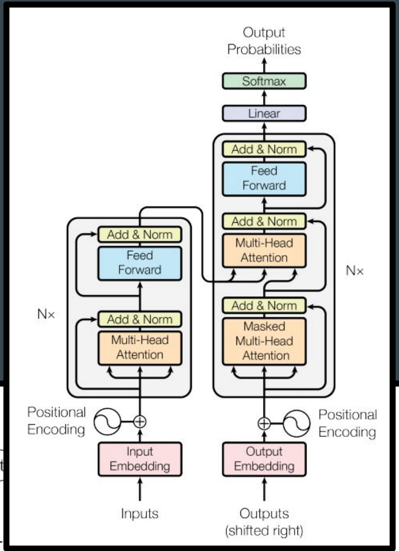
●●●

Brian DuSell

The Age of Neural

- Neural networks have proven to be powerful machine learning models in recent years
- Sweeping over multiple fields in CS, including NLP and Computer Vision
- They are notorious for being uninterpretable black boxes
- This project is focused on analyzing the flow of gradient through the network during training





Application

Neural network analysis

- Networks are organized into logical components
  - Recurrent gates, highway connections, differentiable data structures, etc.
- Let’s try to identify components that most facilitate learning

Kernel

“Gradient tracing”

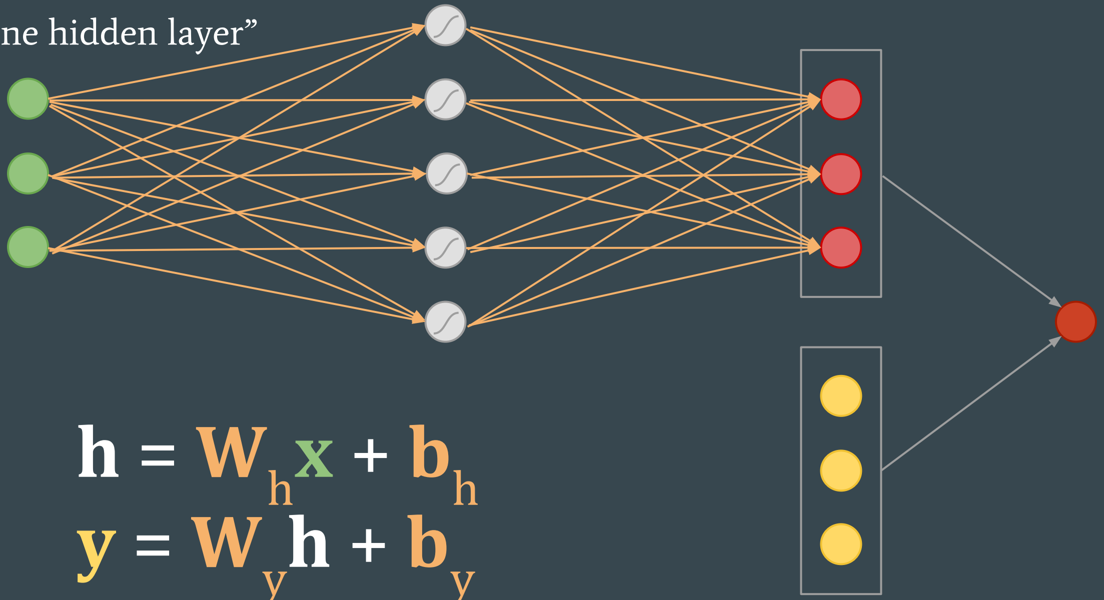
- Finds the path in the computation graph through which the most gradient propagates
- Then finds components that the path intersects with
- Algorithmically very similar to “backpropagation”

Buckle Up

This requires a *lot* of background.

Neural Network Basics

"Feed forward network with one hidden layer"



$$\mathbf{h} = \mathbf{W}_h \mathbf{x} + \mathbf{b}_h$$
$$\mathbf{y} = \mathbf{W}_y \mathbf{h} + \mathbf{b}_y$$
$$L(\mathbf{y}, \hat{\mathbf{y}})$$

Input

Target output

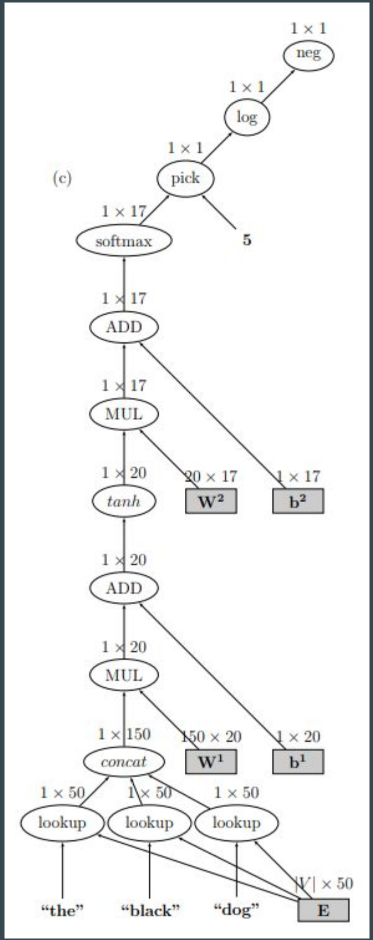
Parameters

Predicted output

Loss function

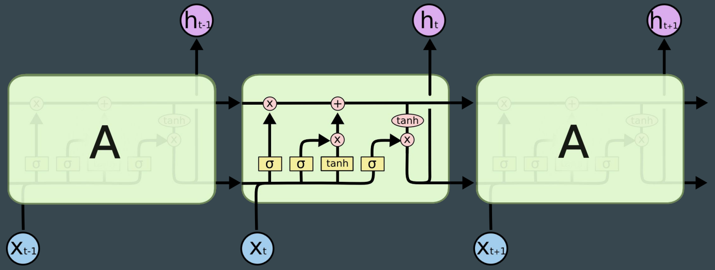
Computation Graphs

- Any neural network can be expressed as a graph of mathematical operators
- Like an abstract syntax tree
- Vertices represent operators, constants, or parameters
- Edges are directed and represent assignments to function parameters
- Always a DAG



Neural Network Components

- Additional configurations of connections and mathematical operators
- Impose an inductive bias on the model
  - e.g. attention, additive gates, etc.



Architectural diagram of LSTM

Image credit: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Why Add Components?

1. Increase the network’s modeling power
    - Stack RNNs, Queue RNNs, Neural Turing Machines
2. Make the model easier to train
    - Additive gates in LSTMs and GRUs
3. Make the model inherently more interpretable
    - In machine translation, attention aligns words in the source and target sentences

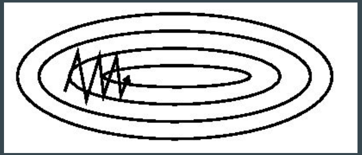
Training Neural Networks

- The behavior of the network changes depending on the values of its parameters
  - Parameters are real numbers often grouped into semantically meaningful tensors
  - Usually connection weights, but can be other things
- The parameters of the network are optimized using gradient descent
  - Involves computing the gradient of the loss function with respect to the parameters

$$\theta' = \theta - \eta \nabla_\theta L(f(\mathbf{x}; \theta), \hat{\mathbf{y}})$$

Gradient descent update rule

η : learning rate
L : loss function
f : neural network
θ : vector of all parameters



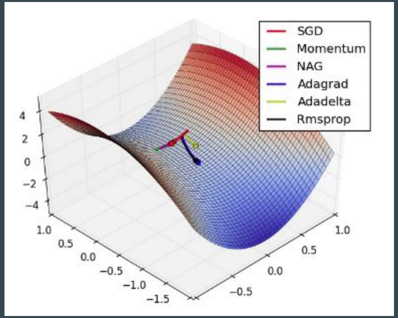
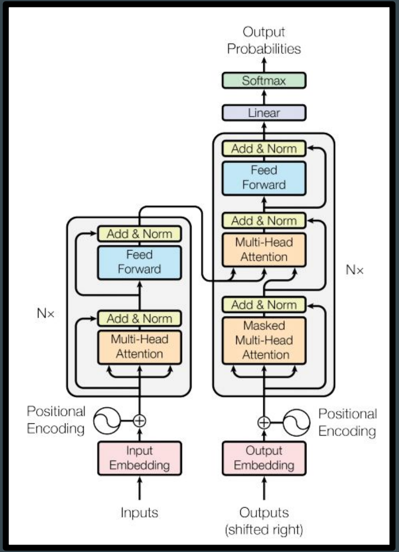


Image credit: http://ruder.io/optimizing-gradient-descent/

Computing Gradients

- Usually called "backpropagation" in the context of neural networks
  - "Gradient" propagates "backward" through the network
- Old way: compute by hand
- Modern way: automatic differentiation
  - Exploits the chain rule of calculus to compute gradients of complex functions with a fixed set of simpler functions
  - "Dynamic graph" libraries like PyTorch and DyNet implement this
  - Topological sort + forward pass + backward pass



"I can be differentiated!" :)

Chain Rule

- Allows us to compute the derivative of composite functions using derivatives of simpler functions
- Intuition
  - How sensitive is f(g(x)) to changes in x at point x?
  - Take sensitivity of g(x) to x at point x
  - Take sensitivity of f(y) to y at point y = g(x)
  - Multiply the two together
- If the same “variable” is referenced multiple times, add the gradients together
- Can be applied recursively

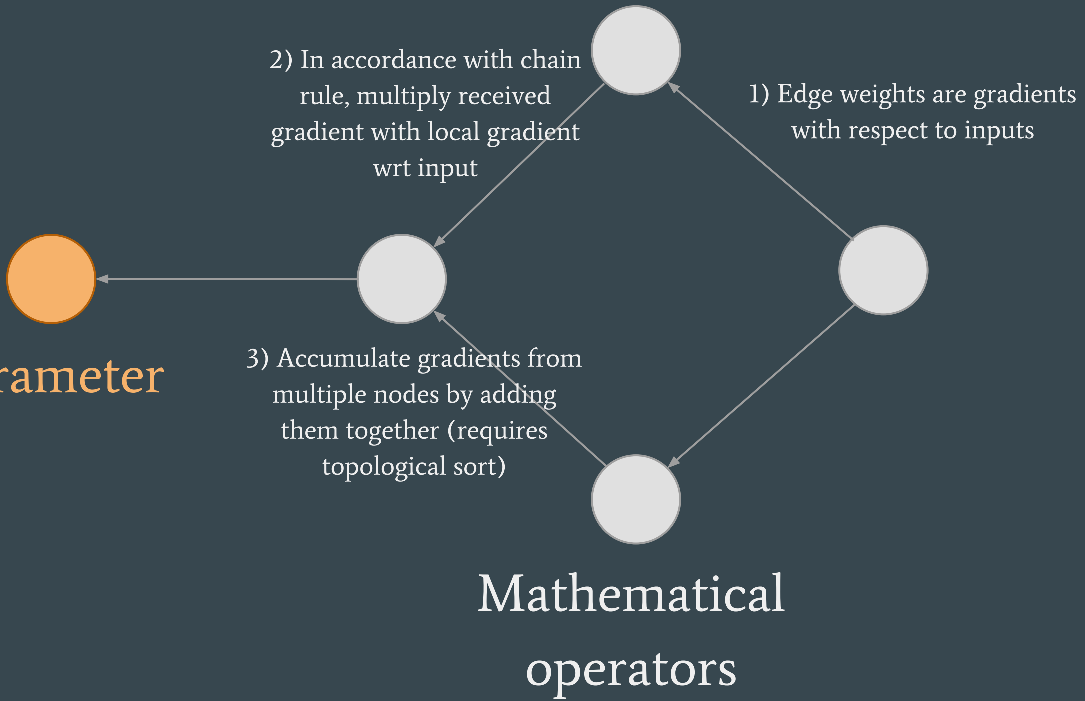
$$\frac{d}{dx} f(g(x)) = f'(g(x)) \frac{d}{dx} g(x)$$

$$\begin{aligned} \frac{\partial}{\partial x_i} f(\mathbf{g}(\mathbf{x})) &= \nabla f(\mathbf{g}(\mathbf{x})) \cdot \frac{\partial}{\partial x_i} \mathbf{g}(\mathbf{x}) \\ &= \sum_k f'_k(\mathbf{g}(\mathbf{x})) \frac{\partial}{\partial x_i} g_k(\mathbf{x}) \end{aligned}$$

Backpropagation as a Graph

Three simple rules:

- Multiply along paths
- Sum incoming edges
- Stop at parameters


2) In accordance with chain rule, multiply received gradient with local gradient wrt input
1) Edge weights are gradients with respect to inputs
Parameter
3) Accumulate gradients from multiple nodes by adding them together (requires topological sort)
Mathematical operators

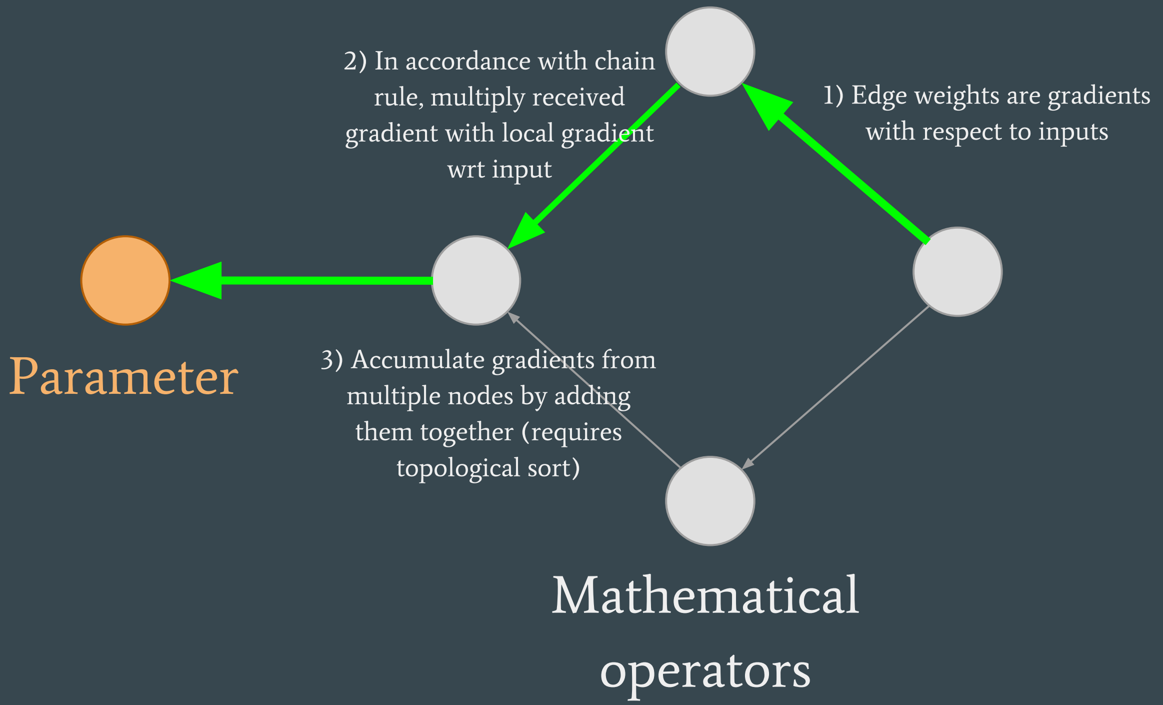
Gradient Tracing Kernel Definition

Some paths are better than others!

Remember the one that propagated the most gradient (absolute value).

Compute for all parameters at once, just like computing gradient in backprop.

Same procedure as backprop, different semiring (max instead of sum)



2) In accordance with chain rule, multiply received gradient with local gradient wrt input
1) Edge weights are gradients with respect to inputs
Parameter
3) Accumulate gradients from multiple nodes by adding them together (requires topological sort)
Mathematical operators

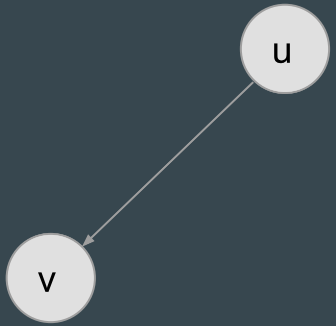
Backprop vs. Gradient Tracing

$$g_v = \nabla_U L = \sum_{(u,v)\in E} w(u,v) g_u$$

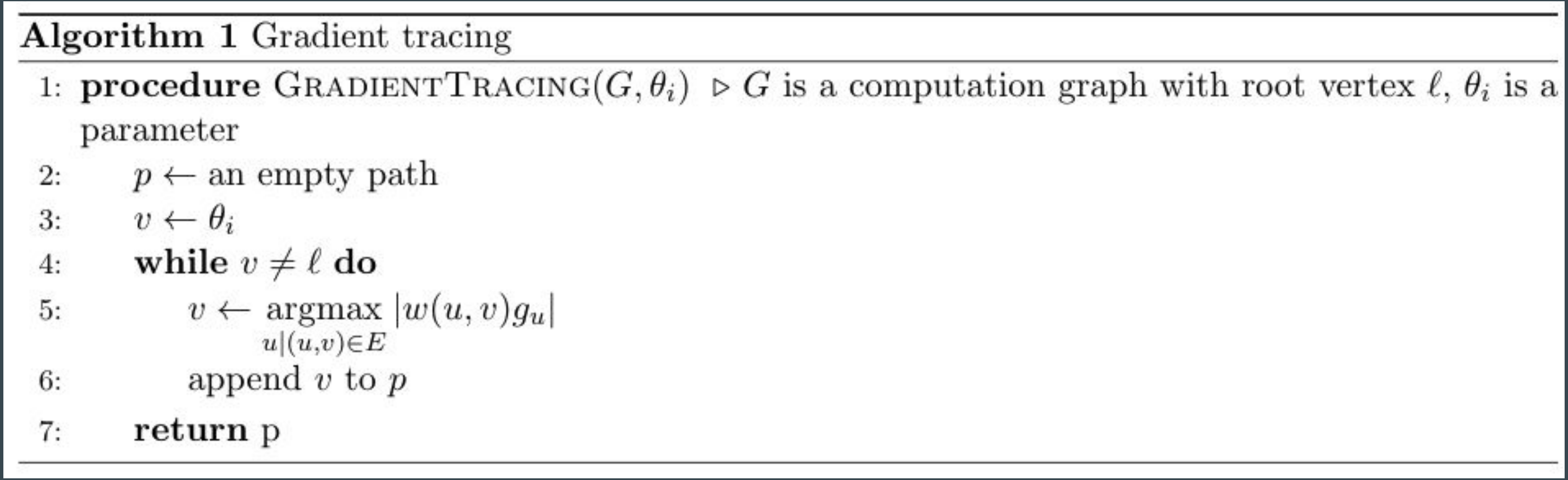
Backpropagation (computing total incoming gradient)

$$t_v = \underset{u|(u,v)\in E}{\operatorname{argmax}} |w(u,v) g_u|$$

Gradient tracing (computing total incoming gradient)



Pseudocode



Algorithm 1 Gradient tracing

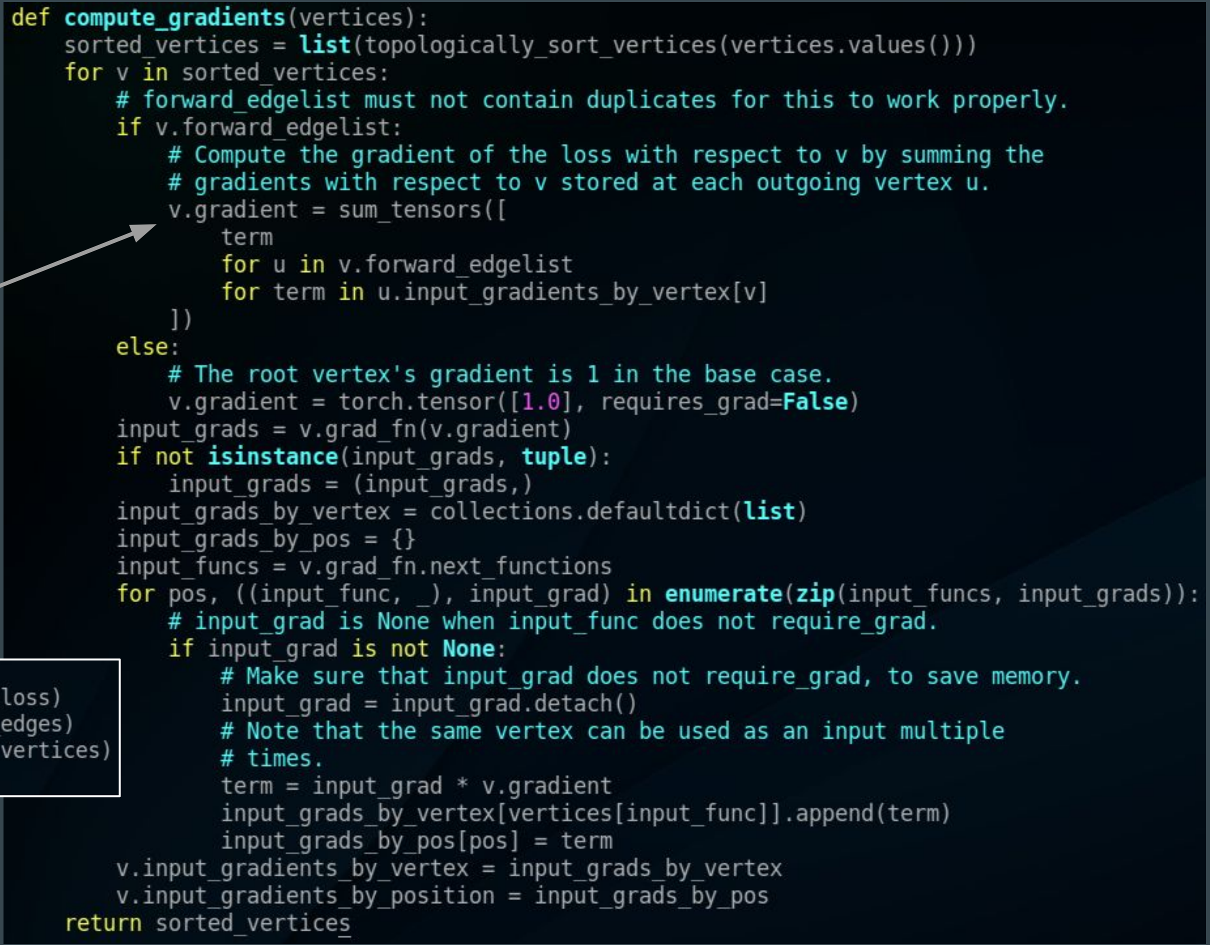
```
1: procedure GradientTracing($(G, \theta_i)$)  ▷ $G$ is a computation graph with root vertex $\ell$, $\theta_i$ is a parameter
2:     $p \leftarrow$ an empty path
3:     $v \leftarrow \theta_i$
4:     while $v \neq \ell$ do
5:         $v \leftarrow \underset{u|(u,v) \in E}{\operatorname{argmax}} |w(u,v) g_u|$
6:         append $v$ to $p$
7:     return p
```

Complexity: O(|V| + |E|)
Reason: every vertex and edge is visited at most once

Implementation

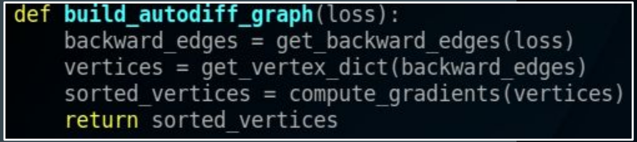
- Language: Python
- Library PyTorch
- Extracting the computation graph from PyTorch is unnecessarily painful
- Scaling results given are instead for the closely related backpropagation algorithm
  - Reason: Primitive PyTorch operations are tensor operations
  - Gradient tracing will require digging into PyTorch primitives, whereas backprop does not
  - Backprop serves as a prototype for the gradient tracing algorithm

Actual Code



```python
def compute_gradients(vertices):
    sorted_vertices = list(topologically_sort_vertices(vertices.values()))
    for v in sorted_vertices:
        # forward_edgelist must not contain duplicates for this to work properly.
        if v.forward_edgelist:
            # Compute the gradient of the loss with respect to v by summing the
            # gradients with respect to v stored at each outgoing vertex u.
            v.gradient = sum_tensors([
                term
                for u in v.forward_edgelist
                for term in u.input_gradients_by_vertex[v]
            ])
        else:
            # The root vertex's gradient is 1 in the base case.
            v.gradient = torch.tensor([1.0], requires_grad=False)
        input_grads = v.grad_fn(v.gradient)
        if not isinstance(input_grads, tuple):
            input_grads = (input_grads,)
        input_grads_by_vertex = collections.defaultdict(list)
        input_grads_by_pos = {}
        input_funcs = v.grad_fn.next_functions
        for pos, ((input_func, _), input_grad) in enumerate(zip(input_funcs, input_grads)):
            # input_grad is None when input_func does not require_grad.
            if input_grad is not None:
                # Make sure that input_grad does not require_grad, to save memory.
                input_grad = input_grad.detach()
                # Note that the same vertex can be used as an input multiple
                # times.
                term = input_grad * v.gradient
                input_grads_by_vertex[vertices[input_func]].append(term)
                input_grads_by_pos[pos] = term
        v.input_gradients_by_vertex = input_grads_by_vertex
        v.input_gradients_by_position = input_grads_by_pos
    return sorted_vertices
```

Notionally change this to a max

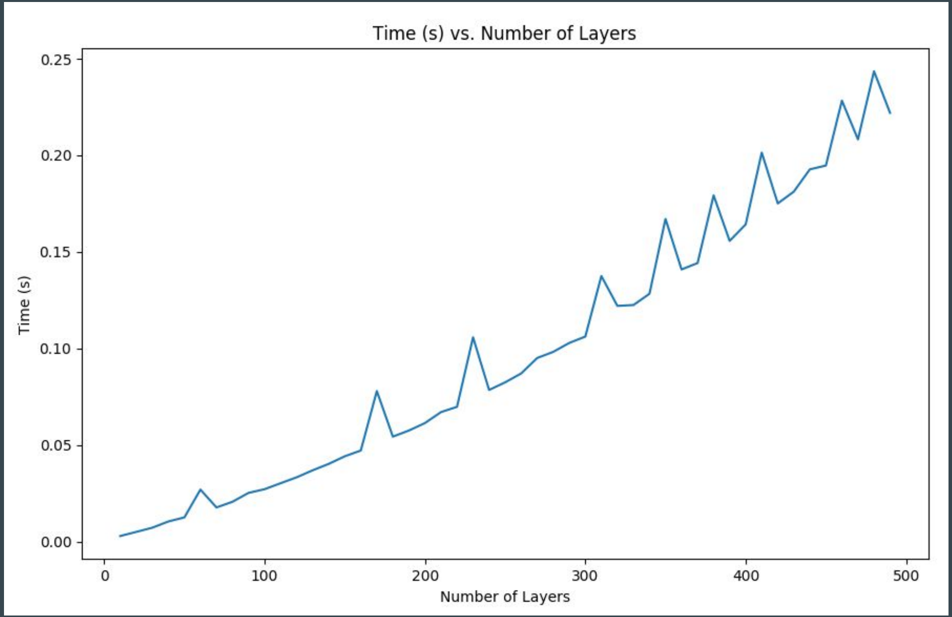


```python
def build_autodiff_graph(loss):
    backward_edges = get_backward_edges(loss)
    vertices = get_vertex_dict(backward_edges)
    sorted_vertices = compute_gradients(vertices)
    return sorted_vertices
```

Data

- Synthetically generated feed-forward neural networks
- Number of layers varies while number of units per layer is constant
  - Input, output, and hidden layers all set to size 20
- Training data is randomly generated
  - Values do not affect speed of computation

Scaling Results



- Includes time to traverse computation graph in reverse, topologically sort nodes, and compute gradients (build_autodiff_graph())
- Slightly superlinear? :(

Future Plans

- Finish implementing gradient tracing proper
- Run on more realistic neural network architectures