Automatic differentiation

Bruno Gavranović

Deep Learning course seminar

bruno.gavranovic@fer.hr

January 12, 2018
Should backpropagation be this confusing?

- Large gap between:
  - Backpropagation materials
  - Deep learning frameworks

- Every tutorial focuses on deriving specific neural network architectures
- Many other equally confusing things

- How do matrices and vectors fit into the story of derivatives?
- Do we really need so many complex rules of derivation?
Should backpropagation be this confusing?

- Large gap between:
  - Backpropagation materials
  - Deep learning frameworks
  - Every tutorial focuses on deriving specific neural network architectures
  - Many other equally confusing things
  - How do matrices and vectors fit into the story of derivatives?
  - Do we really need so many complex rules of derivation?
Should backpropagation be this confusing?

- Large gap between:
  - Backpropagation materials
  - Deep learning frameworks
Should backpropagation be this confusing?

- Large gap between:
  - Backpropagation materials
  - Deep learning frameworks
- *Every* tutorial focuses on deriving specific neural network architectures
Should backpropagation be this confusing?

- Large gap between:
  - Backpropagation materials
  - Deep learning frameworks
- Every tutorial focuses on deriving specific neural network architectures
- Many other equally confusing things
Large gap between:
- Backpropagation materials
- Deep learning frameworks

Every tutorial focuses on deriving specific neural network architectures

Many other equally confusing things
  - How do matrices and vectors fit into the story of derivatives?
Should backpropagation be this confusing?

- Large gap between:
  - Backpropagation materials
  - Deep learning frameworks
- *Every* tutorial focuses on deriving specific neural network architectures
- Many other equally confusing things
  - How do matrices and vectors fit into the story of derivatives?
  - Do we really need so many complex rules of derivation?
Key concept - computational graph

Backpropagation is a function that maps one computational graph to another.

Not connected to linear algebra.

Arbitrary tensor contraction operations can be generalized with Einstein summation.

With Einsum, calculating derivatives is elegant.
Key concept - computational graph
Key concept - computational graph
Backpropagation is a function that maps one computational graph to another
Key concept - computational graph

Backpropagation is a function that maps one computational graph to another

Not connected to linear algebra
Key concept - computational graph

Backpropagation is a function that maps one computational graph to another

Not connected to linear algebra

Arbitrary tensor contraction operations can be generalized with Einstein summation
Key concept - computational graph

Backpropagation is a function that maps one computational graph to another

Not connected to linear algebra

Arbitrary tensor contraction operations can be generalized with Einstein summation

With Einsum, calculating derivatives is elegant
Computational graphs
Computational graphs
Composition of many smaller operations

Instead of defining every neural network by hand, we define many small parts of it and we set up ways to combine them. The main idea is to build a minimal implementation of autodiff during the course of this talk.

One operation - one class
Each operation takes a Node and returns a value.
Instead of defining every neural network by hand, we define many small parts of it and we set up ways to combine them.
Composition of many smaller operations

- Instead of defining every neural network by hand, we define many small parts of it and we set up ways to combine them.
- Main idea - let’s build a minimal implementation of autodiff during the course of this talk.
Instead of defining every neural network by hand, we define many small parts of it and we set up ways to combine them.

Main idea - let's build a minimal implementation of autodiff during the course of this talk.

One operation - one class.
Composition of many smaller operations

- Instead of defining every neural network by hand, we define many small parts of it and we set up ways to combine them.
- Main idea - let’s build a minimal implementation of autodiff during the course of this talk.
- One operation - one class.
- Each operation takes a Node and returns a value.
class Variable:
    def __init__(self, value, name="Variable"):  
        self.value = value

    def _eval(self):
        return self.value
class Exp:
    def __init__(self, node, name="Exp"):  
        self.node = node

    def _eval(self):
        return np.exp(self.node._eval())
class Add:
    def __init__(self, node1, node2, name="Add"):  
        self.node1 = node1  
        self.node2 = node2

    def _eval(self):
        return node1._eval() + node2._eval()
class Sigmoid:
    def __init__(self, node, name="Sigmoid"):
        self.node = node

    def _eval(self):
        return 1 / (1 + np.exp(-self.node._eval()))
Let's abstract some common stuff

class Node:
    def __init__(self, nodes, name="Node"):
        self.nodes = nodes

    def _eval(self):
        raise NotImplementedError()
Let's abstract some common stuff

class Node:
    def __init__(self, nodes, name="Node"):  
        self.nodes = nodes

    def _eval(self): 
        raise NotImplementedError()

    def __add__(self, other):
        return Add(self, other)

    def __call__(self, *args, **kwargs):
        return self.eval()
class Node:
    def __init__(self, nodes, name="Node"):  
        self.nodes = nodes

    def _eval(self):
        raise NotImplementedError()

    def __add__(self, other):
        return Add(self, other)

    def __call__(self, *args, **kwargs):
        return self.eval()
Let’s abstract some common stuff

class Node:
    def __init__(self, nodes, name="Node"):  
        self.nodes = nodes

    def _eval(self):  
        raise NotImplementedError()

    def __add__(self, other):  
        return Add(self, other)

    def __call__(self, *args, **kwargs):  
        return self.eval()

    def eval(self):  
        if self.cached is None:
            self.cached = self._eval()

        return self.cached
Let's abstract some common stuff

class Node:
    def __init__(self, nodes, name="Node"):  
        self.nodes = nodes
        self.cached = None

    def _eval(self):
        raise NotImplementedError()

    def __add__(self, other):
        return Add(self, other)

    def __call__(self, *args, **kwargs):
        return self.eval()

    def eval(self):
        if self.cached is None:
            self.cached = self._eval()

        return self.cached
class Node:
    def __init__(self, nodes, name="Node"):  
        self.nodes = nodes  
        self.cached = None

    def _eval(self):  
        raise NotImplementedError()

    def __add__(self, other):  
        return Add(self, other)

    def __call__(self, *args, **kwargs):  
        return self.eval()

    def eval(self):  
        if self.cached is None:  
            self.cached = self._eval()
        return self.cached
class Exp(Node):
    def __init__(self, node, name="Exp"):
        super().__init__([node])

    def _eval(self):
        return np.exp(self.nodes[0]())
class Add(Node):
    def __init__(self, node1, node2, name="Add"):  
        super().__init__( [node1, node2] )

    def _eval(self):
        return self.nodes[0]() + self.nodes[1]()
class Sigmoid(Node):
    def __init__(self, node, name="Sigmoid"):  
        super().__init__([node])

    def _eval(self):
        return 1 / (1 + np.exp(-self.nodes[0]()))
What do we have so far?

We can define arbitrary computation graphs...

But how do we train them?

Where are all the derivatives?

Where is the neural network here?

Turns out, we're missing two things:

Matrix operations

Gradient calculation
What do we have so far?

- We can define arbitrary computation graphs...
What do we have so far?

- We can define arbitrary computation graphs...
- But how do we *train* them?
What do we have so far?

- We can define arbitrary computation graphs...
- But how do we *train* them?
- Where are all the derivatives?
What do we have so far?

- We can define arbitrary computation graphs...
- But how do we *train* them?
- Where are all the derivatives?
- Where is the neural network here?
What do we have so far?

- We can define arbitrary computation graphs...
- But how do we *train* them?
- Where are all the derivatives?
- Where is the neural network here?
- Turns out, we’re missing two things:
We can define arbitrary computation graphs...
But how do we *train* them?
Where are all the derivatives?
Where is the neural network here?
Turns out, we’re missing two things:
  - Matrix operations
What do we have so far?

- We can define arbitrary computation graphs...
- But how do we \textit{train} them?
- Where are all the derivatives?
- Where is the neural network here?
- Turns out, we’re missing two things:
  - Matrix operations
  - Gradient calculation
Let's quickly add matrix multiplication

class MatMul(Node):
    def __init__(self, node1, node2, name="MatMul"):
        super().__init__([node1, node2])

    def _eval(self):
        return self.nodes[0]() @ self.nodes[1]()}
Backpropagation

Diagram:
- Sigmoid node
- Add node
- Nodes labeled 'a' and 'b'

Graphical representation of a neural network with a Sigmoid function and an Add operation.
Backpropagation - we don’t need to know the types
Backpropagation
Backpropagation
Backpropagation
We need derivatives!

class Node:
    def __init__(self, nodes, name="Node"):  
        self.nodes = nodes

    # other stuff ...

    def _eval(self):
        raise NotImplementedError()
class Node:
    def __init__(self, nodes, name="Node"):  
        self.nodes = nodes

    # other stuff ...

def _eval(self):
    raise NotImplementedError()

def _partial_derivative(self, wrt, previous_grad):
    raise NotImplementedError()
class Add(Node):
    def __init__(self, node1, node2, name="Add"):  
        super().__init__([node1, node2])

    def _eval(self):
        return self.nodes[0]() + self.nodes[1]()}
class Add(Node):
    def __init__(self, node1, node2, name="Add"):  
        super().__init__(node1, node2)

    def _eval(self):
        return self.nodes[0]() + self.nodes[1]()

    def _partial_derivative(self, wrt, previous_grad):
        return previous_grad * self.nodes.count(wrt)
class Sigmoid(Node):
    def __init__(self, node, name="Sigmoid"):  
        super().__init__((node])

    def _eval(self):
        return 1 / (1 + np.exp(-self.nodes[0]())

    def _partial_derivative(self, wrt, previous_grad):
        if wrt == self.node:
            return previous_grad * self * (1 - self)
        return 0
class Sigmoid(Node):
    def __init__(self, node, name="Sigmoid"):  
        super().__init__([node])

    def _eval(self):
        return 1 / (1 + np.exp(-self.nodes[0]()))

    def _partial_derivative(self, wrt, previous_grad):
        if wrt == self.node:
            return previous_grad * self * (1 - self)
        return 0
Things to keep in mind

Constructing the graph of the gradient does not imply its evaluation! When constructing the partial derivative, by not “stepping down” from our graphs into real numbers, we get higher-order gradients for free!
Things to keep in mind

- Constructing the graph of the gradient does not imply its evaluation!
Things to keep in mind

- Constructing the graph of the gradient does not imply its evaluation!
- When constructing the partial derivative, by not “stepping down" from our graphs into real numbers, we get higher-order gradients for free!
So where is backpropagation?
def grad(top_node, wrt_list, previous_grad=None):
    if previous_grad is None:
        previous_grad = Variable(np.ones(top_node.shape), name=add_sum_name(top_node))
    dct = collections.defaultdict(list)
    dct[top_node] += [previous_grad]
    def add_partials(dct, node):
        dct[node] = Add(*dct[node], name=add_sum_name(node))
        for child in set(node.children):
            dct[child] += [node.partial_derivative(wrt=child, previous_grad=dct[node])]
        return dct
    dct = functools.reduce(add_partials, reverse_topo_sort(top_node), dct)
    return [dct[wrt] if dct[wrt] != [] else Variable(0) for wrt in wrt_list]
So where is backpropagation?

```python
def grad(top_node, wrt_list, previous_grad=None):
    if previous_grad is None:
        previous_grad = Variable(np.ones(top_node.shape),
                                  name=add_sum_name(top_node))
    dct = collections.defaultdict(list)
    dct[top_node] += [previous_grad]
    def add_partials(dct, node):
        dct[node] = Add(*dct[node], name=add_sum_name(node))
        for child in set(node.children):
            dct[child] += [node.partial_derivative(wrt=child, previous_grad=dct[node])]
        return dct
    dct = functools.reduce(add_partials, reverse_topo_sort(top_node), dct)
    return [dct[wrt] if dct[wrt] != [] else Variable(0) for wrt in wrt_list]
```
def grad(top_node, wrt_list, previous_grad=None):
    if previous_grad is None:
        previous_grad = Variable(np.ones(top_node.shape),
                                  name=add_sum_name(top_node))
    dct = collections.defaultdict(list)
    dct[top_node] += [previous_grad]

    def add_partials(dct, node):
        dct[node] = Add(*dct[node], name=add_sum_name(node))
        for child in set(node.children):
            dct[child] += [node.partial_derivative(wrt=child,
                                                    previous_grad=dct[node])]
        return dct
    dct = functools.reduce(add_partials, reverse_topo_sort(top_node), dct)
    return [dct[wrt] if dct[wrt] != [] else Variable(0) for wrt in wrt_list]
So where is backpropagation?

def grad(top_node, wrt_list, previous_grad=None):
    if previous_grad is None:
        previous_grad = Variable(np.ones(top_node.shape),
                                  name=add_sum_name(top_node))

    dct = collections.defaultdict(list)
    dct[top_node] += [previous_grad]

    def addpartials(dct, node):
        dct[node] = Add(*dct[node], name=add_sum_name(node))
        for child in set(node.children):
            dct[child] += [node.partial_derivative(wrt=child,
                                                    previous_grad=dct[node])]

    return dct

Bruno Gavranović FER

Autodiff

January 12, 2018 30 / 33
def grad(top_node, wrt_list, previous_grad=None):
    if previous_grad is None:
        previous_grad = Variable(np.ones(top_node.shape),
                                  name=add_sum_name(top_node))

    dct = collections.defaultdict(list)
    dct[top_node] += [previous_grad]

    def addpartials(dct, node):
        dct[node] = Add(*dct[node], name=add_sum_name(node))
        for child in set(node.children):
            dct[child] += [node.partial_derivative(wrt=child,
                                                   previous_grad=dct[node])]

        return dct

    dct = functools.reduce(addpartials, 
                           reverse_topo_sort(top_node),
                           dct)

    return [dct[wrt] if dct[wrt] != [] else Variable(0) for wrt in wrt_list]
def grad(top_node, wrt_list, previous_grad=None):
    if previous_grad is None:
        previous_grad = Variable(np.ones(top_node.shape),
                                  name=add_sum_name(top_node))

    dct = collections.defaultdict(list)
    dct[top_node] += [previous_grad]

    def addpartials(dct, node):
        dct[node] = Add(*dct[node], name=add_sum_name(node))
        for child in set(node.children):
            dct[child] += [node.partial_derivative(wrt=child,
                                                   previous_grad=dct[node])]

        return dct

    dct = functools.reduce(addpartials,
                           reverse_topo_sort(top_node),
                           dct)

    return [dct[wrt] if dct[wrt] != [] else Variable(0) for wrt in wrt_list]
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs w.r.t. any variable
- Support for higher-order gradients
- Extensible code - it's easy to add your own operations

What else is there?

- Support for higher-order tensors
- Numerical checks
- Checkpointing
- Visualization of the computational graph
What did we end up with?

- Dynamic creation of computational graphs
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs w.r.t. any variable
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs w.r.t. any variable
- Support for higher-order gradients
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs w.r.t. any variable
- Support for higher-order gradients
- Extensible code - it’s easy to add your own operations

What else is there?

- Support for higher-order tensors
- Numerical checks
- Checkpointing
- Visualization of the computational graph
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs w.r.t. any variable
- Support for higher-order gradients
- Extensible code - it’s easy to add your own operations
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs w.r.t. any variable
- Support for higher-order gradients
- Extensible code - it’s easy to add your own operations

What else is there?
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs \(\text{w.r.t.}\) any variable
- Support for higher-order gradients
- Extensible code - it’s easy to add your own operations

What else is there?

- Support for higher-order tensors
Dynamic creation of computational graphs
Differentiation of computational graphs w.r.t. any variable
Support for higher-order gradients
Extensible code - it’s easy to add your own operations

What else is there?

Support for higher-order tensors
Numerical checks
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs w.r.t. any variable
- Support for higher-order gradients
- Extensible code - it’s easy to add your own operations

What else is there?

- Support for higher-order tensors
- Numerical checks
- Checkpointing
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs w.r.t. any variable
- Support for higher-order gradients
- Extensible code - it’s easy to add your own operations

What else is there?

- Support for higher-order tensors
- Numerical checks
- Checkpointing
- Visualization of the computational graph
What did we end up with?

- Dynamic creation of computational graphs
- Differentiation of computational graphs w.r.t. any variable
- Support for higher-order gradients
- Extensible code - it’s easy to add your own operations

What else is there?

- Support for higher-order tensors
- Numerical checks
- Checkpointing
- Visualization of the computational graph
Future work

Difference between forward, backward and mixed mode of automatic differentiation, viewed in this context

Even more refactoring

Formal validation of these ideas

The rabbit hole of finding patterns in these abstract concepts goes incredibly deep

Backprop as a Functor
Future work

- Difference between forward, backward and mixed mode of automatic differentiation, viewed in this context
Future work

- Difference between forward, backward and mixed mode of automatic differentiation, viewed in this context
- Even more refactoring
Future work

- Difference between forward, backward and mixed mode of automatic differentiation, viewed in this context
- Even more refactoring
- Formal validation of these ideas
Future work

- Difference between forward, backward and mixed mode of automatic differentiation, viewed in this context
- Even more refactoring
- Formal validation of these ideas
- The rabbit hole of finding patterns in these abstract concepts goes incredibly deep
Future work

- Difference between forward, backward and mixed mode of automatic differentiation, viewed in this context
- Even more refactoring
- Formal validation of these ideas
- The rabbit hole of finding patterns in these abstract concepts goes incredibly deep
- *Backprop as a Functor*
Thank you!