Autograd, Computational Graphs and Vanishing Gradients

A Comprehensive Overview

Sayan Chaki

March 20, 2025

Table of Contents

What is Automatic Differentiation?

- **Automatic differentiation** (autograd) is a set of techniques to efficiently compute exact derivatives
- Essential component of modern deep learning frameworks
- Allows for training of complex neural networks via gradient-based optimization
- Different from numerical differentiation (finite differences) and symbolic differentiation

Types of Automatic Differentiation

Forward Mode

- Computes derivatives alongside function evaluation
- Efficient for functions with many outputs and few inputs

Reverse Mode

- Computes derivatives after function evaluation
- Efficient for functions with many inputs and few outputs
- Used in most deep learning frameworks

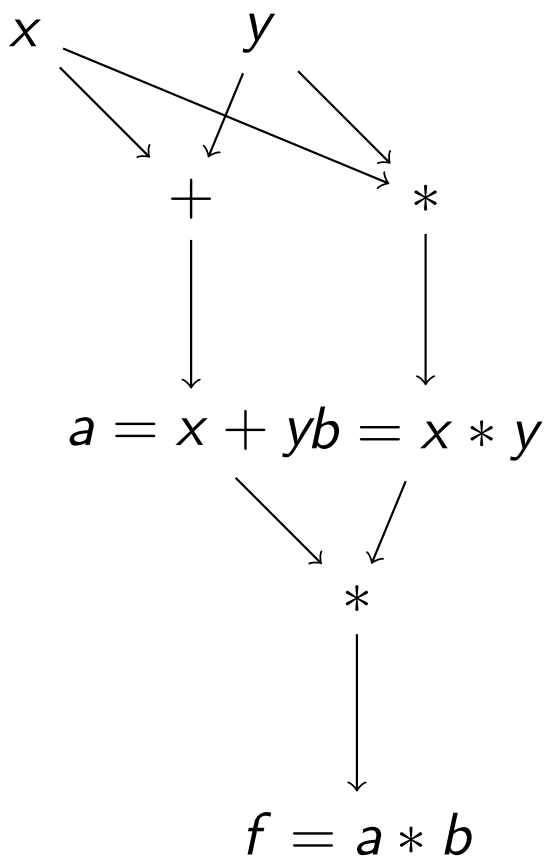
$$\text{Backpropagation} = \text{Reverse-mode autodiff} + \text{Chain rule}$$

Computational Graphs

- A computational graph is a directed graph representing mathematical operations
- Nodes: variables (inputs, parameters, intermediate values, outputs)
- Edges: operations that transform values
- Provides a visual and conceptual framework for understanding calculation flow
- Forms the basis for implementing automatic differentiation

Computational Graph Example

Consider the function: $f(x, y) = (x + y) * (x * y)$



Forward Pass vs. Backward Pass

Forward Pass

- Compute values from inputs to outputs
- Store intermediate results for backward pass
- Example:
  $f(x, y) = (x + y) * (x * y)$

$$a = x + y$$
$$b = x * y$$
$$f = a * b$$

Backward Pass

- Compute gradients from outputs to inputs
- Apply chain rule to propagate derivatives

$$\frac{\partial f}{\partial a} = b$$
$$\frac{\partial f}{\partial b} = a$$
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial x}$$
$$= b \cdot 1 + a \cdot y$$

PyTorch Autograd Basics

```python
import torch

# Create tensors with autograd enabled
x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)

# Forward pass
a = x + y
b = x * y
f = a * b

# Compute gradients
f.backward()

# Display results
print(f"f(x,y) = {f.item()}")
print(f"df/dx = {x.grad.item()}")
print(f"df/dy = {y.grad.item()}")
```

Output:

Building a Simple Autograd Engine

```python
class Value:
    def __init__(self, data, _children=(), _op=''):
        self.data = data
        self.grad = 0
        self._backward = lambda: None
        self._prev = set(_children)
        self._op = _op

    def __add__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data, (self, other), '+')

        def _backward():
            self.grad += out.grad
            other.grad += out.grad
        out._backward = _backward

        return out
```

Custom Autograd Example (continued)

```
    def __mul__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data * other.data, (self, other), '*')

        def _backward():
            self.grad += other.data * out.grad
            other.grad += self.data * out.grad
        out._backward = _backward

        return out

    def backward(self):
        # Topological sort
        topo = []
        visited = set()
        def build_topo(v):
            if v not in visited:
                visited.add(v)
                for child in v._prev:
```

The Chain Rule in Autograd

The chain rule is fundamental to autograd:

$$\frac{df}{dx} = \frac{df}{dz} \cdot \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{1}$$

For a computational graph:

$$\frac{\partial L}{\partial w_i} = \sum_j \frac{\partial L}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_i} \tag{2}$$

where:

- L is the loss function
- z_j are intermediate nodes
- w_i are parameters

The Vanishing Gradient Problem

- Problem: gradients decrease exponentially as they propagate through layers in deep networks
- Result: earlier layers learn very slowly or not at all
- Causes:
  - Activation functions with small derivatives (e.g., sigmoid, tanh)
  - Weight initialization schemes
  - Deep network architectures
- Consequence: training stalls and performance plateaus

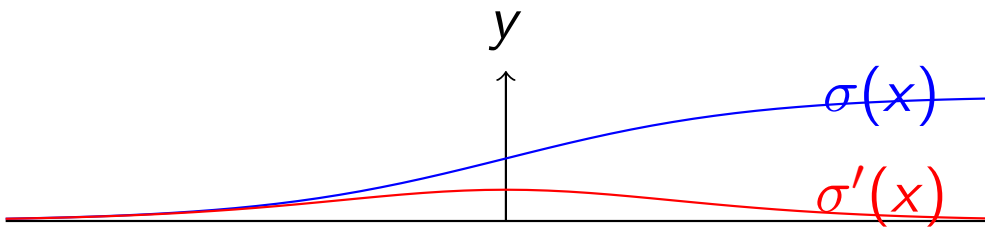
Visualizing Vanishing Gradients

Sigmoid activation function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

Its derivative:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{4}$$



Maximum value of $\sigma'(x)$ is 0.25 at $x = 0$

Gradient Flow in Deep Networks

For a deep network with L layers using sigmoid activations:

$$\left\| \frac{\partial L}{\partial w^{(1)}} \right\| \approx \prod_{l=2}^{L} \|W^{(l)}\| \cdot \prod_{l=1}^{L} \|\sigma'(z^{(l)})\| \cdot \left\| \frac{\partial L}{\partial y} \right\| \tag{5}$$

If each $\|\sigma'(z^{(l)})\| \leq 0.25$ and L is large:

$$\prod_{l=1}^{L} \|\sigma'(z^{(l)})\| \leq 0.25^{L} \tag{6}$$

For $L = 10$: $0.25^{10} \approx 10^{-6}$ (gradient virtually disappears)

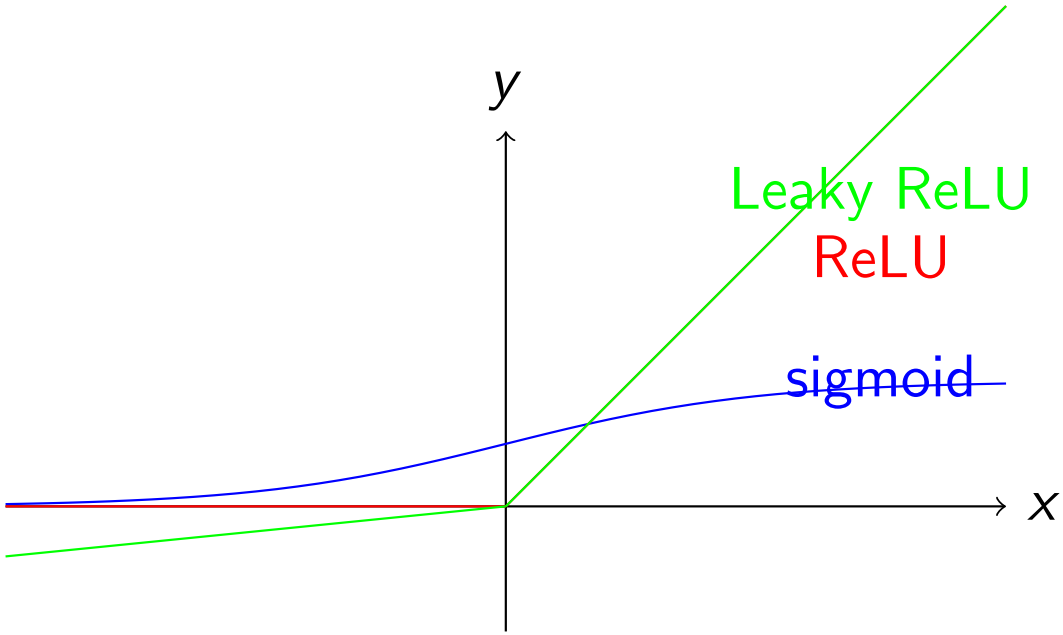
Demonstrating Vanishing Gradients with Code

```python
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

# Create a deep network with sigmoid activations
def create_deep_network(depth, activation=nn.Sigmoid()):
    layers = []
    for i in range(depth):
        layers.append(nn.Linear(1, 1))
        if i < depth - 1:
            layers.append(activation)
    return nn.Sequential(*layers)

# Track gradients through backpropagation
def check_gradients(model, depth):
    x = torch.ones(1, 1, requires_grad=True)
    y = model(x)
    y.backward()

    gradients = []
    for i, layer in enumerate(model):
```

Solutions to the Vanishing Gradient Problem

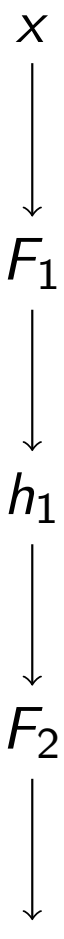
- **Better activation functions**
  - ReLU: $f(x) = \max(0, x)$
  - Leaky ReLU: $f(x) = \max(0.01x, x)$
  - ELU, SELU, Swish
- **Better initialization schemes**
  - Xavier/Glorot initialization
  - He initialization
- **Architectural innovations**
  - Skip connections (ResNet)
  - Normalization layers (BatchNorm, LayerNorm)
  - LSTM/GRU for sequence data

Activation Functions Comparison

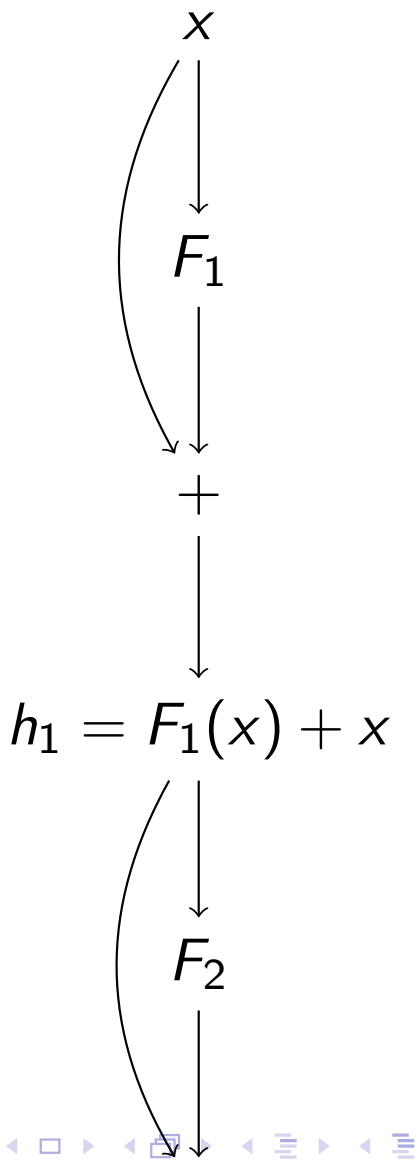


Residual Connections

Standard Network



Residual Network



Implementing Solutions in PyTorch

```python
# Better activation function
model_relu = nn.Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)

# Batch normalization
model_with_bn = nn.Sequential(
    nn.Linear(784, 256),
    nn.BatchNorm1d(256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.BatchNorm1d(128),
    nn.ReLU(),
    nn.Linear(128, 10)
)

# Residual connections
```

Training a Deep Network with PyTorch

```python
# Define a fully-connected network with ReLU activations
class DeepMLP(nn.Module):
    def __init__(self, activation=nn.ReLU()):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 512),
            activation,
            nn.Linear(512, 256),
            activation,
            nn.Linear(256, 128),
            activation,
            nn.Linear(128, 64),
            activation,
            nn.Linear(64, 10)
        )

    def forward(self, x):
        return self.model(x)

# Train with different activation functions
relu_model = DeepMLP(activation=nn.ReLU())
```

Visualizing Gradient Flow

```python
def plot_grad_flow(named_parameters):
    ave_grads = []
    layers = []
    for n, p in named_parameters:
        if(p.requires_grad) and ("bias" not in n):
            layers.append(n)
            ave_grads.append(p.grad.abs().mean().item())
    plt.figure(figsize=(10, 8))
    plt.bar(range(len(ave_grads)), ave_grads, align="center"
        )
    plt.xticks(range(len(ave_grads)), layers, rotation=90)
    plt.xlabel("Layers")
    plt.ylabel("Average gradient")
    plt.title("Gradient flow")
    plt.tight_layout()
    plt.show()

# Usage
plot_grad_flow(model.named_parameters())
```

Summary

- **Automatic differentiation** provides efficient gradient computation for neural network training
- **Computational graphs** form the conceptual and practical framework for implementing autograd
- **Vanishing gradients** represent a fundamental challenge in training deep networks
- Modern solutions include:
  - Better activation functions (ReLU family)
  - Improved weight initialization
  - Architectural innovations (skip connections, normalization)
- Understanding these concepts is crucial for designing and debugging deep neural networks

References

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. CVPR.
- Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. JMLR.
- Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. ICML.
- PyTorch documentation: `https://pytorch.org/docs/stable/autograd.html`