Auto Diff
Mark Goldstein

1 Definition of Derivative

The derivative $f'x$ of a function $f : \mathbb{R} \to \mathbb{R}$ at a point x (in the domain of f) is the slope of the tangent line to f at the point x, defined by:

$$f'x = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

When f is a scalar-valued function of a vector-valued argument, we use the notation of partial derivatives with respect to n scalar components. If $f : \mathbb{R}^n \to \mathbb{R}$, then we have a vector with entries $\partial f/\partial x_j$ for $j \in \{1, ..., n\}$ called the gradient vector and notated with ∇:

$$\nabla f = \Big(\frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_n}\Big)$$

When the codomain $\mathbb{R}^m$ is non-scalar, we have a matrix $\mathbf{J}$ (the Jacobian) with $J_{ij} = \partial f_i/\partial x_j = \partial y_i/\partial x_j$ for $i \in \{1, ..., m\}$, $j \in \{1, ..., n\}$ where each f_i is the projection of the i^{th} scalar of the result of f. Here is the notation for evaluating the Jacobian at a particular point $\mathbf{x} = \mathbf{a}$:

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}\Bigg|_{\mathbf{x}=\mathbf{a}}$$

The scalar chain rule for derivatives of compositions says:

$$\frac{d}{dx}(f(x) + g(x)) \to \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$$
$$\frac{d}{dx}(f(x)g(x)) \to \Big(\frac{d}{dx}f(x)\Big)g(x) + f(x)\Big(\frac{d}{dx}g(x)\Big)$$

The vector-valued chain rule says that when $\mathbf{x}$ is a vector, $\mathbf{g}(x)$ is a vector-valued function of a vector argument, and $f(\mathbf{u})$ is a scalar-valued function of a vector-argument:

$$\frac{\partial f \circ \mathbf{g}}{\partial x_i} = \nabla f \cdot \frac{\partial \mathbf{g}}{\partial x_i} \quad \text{or, acting on x:} \quad \frac{\partial f \circ \mathbf{g}}{\partial x_i}(\mathbf{x}) = (\nabla f)(\mathbf{g}(\mathbf{x})) \cdot \frac{\partial \mathbf{g}}{\partial x_i}(\mathbf{x})$$

When f is vector-to-vector this generalizes as:

$$\frac{\partial \mathbf{f} \circ \mathbf{g}}{\partial x_i} = \sum_{\ell} \frac{\partial \mathbf{f}}{\partial g_\ell} \frac{\partial g_\ell}{\partial x_i} \qquad \text{each term } f\text{-dimensional, such derivative for each } x_i$$

This rule expresses the fact that a change in the x_i direction may change all of $\{g_\ell\}$ and any of these changes may affect any dimensions of f. Therefore, the derivatives of composed multivariate functions are chains of Jacobian matrix multiplications. Finally, note that one can obtain the directional derivative of a scalar-valued function of f with respect to its vector-valued argument $\mathbf{x}$ along a direction $\mathbf{r}$ by performing a Jacobian-vector product (JVP) of $\mathbf{J}_f \cdot \mathbf{r}$.

2 Automatic Differentiation

Mainly following [Baydin et al 2018]. Methods for computing derivatives in computer programs can be classified into four categories:

1. manually working out derivatives and coding them (error prone)
2. numerical differentiation using finite difference approximations (highly inaccurate due to round-off and truncation errors, scales poorly for gradients with respect to millions of parameters)
3. symbolic differentiation using expression manipulation in computer algebra systems (results in complex, cryptic expressions, requires closed form expressions, i.e. no loops)
4. automatic differentiation (what this discussion is about)

Automatic Differentiation (autodiff / AD) performs non-standard interpretation of a given computer program by replacing the domain of the variables to incorporate derivative values and redefining the semantics of the operators to propagate derivatives per the chain rule of differential calculus. Some main points:

- Can be applied to regular code with minimal change, allowing branching, loops, recursion.
- Numerical computations are compositions of finite set of base operations for which derivatives are known. These are combined through the chain rule to give derivatives of the compositions. Operations include binary arithmetic operations, negation, and transcendental functions such as the exponential, logarithm, and trig. functions. Differentiation is not computable [Pour-El and Richards, 1978, 1983] for arbitrary functions, must use compositions.
- Makes use of "evaluation traces": bookkeeping includes working variables that are assigned the results of evaluating subexpressions.
- Because any numerical code will result in a numeric evaluation trace, code that loops and branches is differentiable: AD is blind with respect to any operation, including control flow statements, which do not directly alter numeric values.
- AD has two main modes. Consider differentiating a function $f : \mathbb{R}^n \to \mathbb{R}^m$
  - Forward Mode (or Forward accumulation mode, or tangent linear mode). This mode is better when $m > n$
  - Reverse Mode (or reverse accumulation mode, or adjoint linear mode or cotangent linear mode). Preferred when $n >> m$, which is often the case in machine learning, where we differentiate a *scalar* loss function with respect to *millions* of parameters.

First let's cover an evaluation trace example, cover dual numbers (important for AD) and then look at AD's two modes.

2.1 Evaluation Trace Conventions

We will use Wengert Lists [Wengert, 1964] with three-part notation used by [Griewank and Walther 2008] where a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is constructed issuing intermediate variables v_i such that:

- variables $v_{i-n} = x_i$ for $i = 1, ..., n$ are input variables
- variables v_i for $i = 1, ..., l$ are working (intermediate) variables
- variables $y_{m-i} = v_{l-i}$ for $i = m-1, ..., 0$ are the output variables.

For our working example, consider $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$. If we evaluate f at input $(2, 5)$, the evaluation trace (or in Forward-Mode AD terms, the "forward primal trace") looks as follows:

Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

2.2 Dual Numbers

Dual Numbers are used to conveniently implement one auto-differentiation approach (forward mode). We briefly review them here, following [Mathoma 2016] and [Berland 2006]

Complex Numbers $\mathbb{C}$:

- numbers of form $a + bi$ where $i^2 = -1$.
- multiplication rule: $(a + bi)(c + di) = ac - bd + (ad + bc)i$ (consequence of $i^2 = -1$)

Split Complex Numbers $\mathbb{C}_s$:

- numbers of form $a + bj$ where $j^2 = 1$
- multiplication rule: $(a + bj)(c + dj) = ac + bd + (ad + bc)j$ (consequence of $j^2 - 1$).

In both of the above: think about an imaginary unit, have it be equal to some number upon squaring, see what multiplication looks like. How about ϵ such that $\epsilon^2 = 0$?

Dual Numbers $\mathbb{D}$:

- numbers of form $a + b\epsilon$ where $\epsilon^2 = 0$
- multiplication: $(a + b\epsilon)(c + d\epsilon) = ac + (ad + bc)\epsilon$.
- Let's represent them as ordered pairs without reference to ϵ. So $a + b\epsilon$ is written as (a, b).
- multiplication rule becomes $(a, b)(c, d) = (ac, ad + bc)$.
- Addition:$(a, b) + (c, d) = (a + c, b + d)$.
- $(0, 1)(0, 1) = (0, 0)$. Two numbers not equal to 0 multiply to 0!

2.3 Using Dual Numbers for Differentiation

Consider $f(x) = x^2$, $f : \mathbb{R} \to \mathbb{R}$. Let's expand this function to operate over duals, i.e. so $f : \mathbb{D} \to \mathbb{D}$. Instead of evaluating $f(a)$ let's evaluate $f(a + 1\epsilon)$:

$$f(a + 1\epsilon) = (a + 1\epsilon)^2 = a^2 + 2a\epsilon + \epsilon^2 = a^2 + 2a\epsilon$$

where the last equality follows from $\epsilon^2 = 0$. Notice that a^2 is $f(a)$ and that $2a$ is $f'(a)$. So $f(a + 1\epsilon) = f(a) + f'(a)\epsilon$ in this case.

Now consider polynomials over dual numbers. Let $P(x) = p_0 + p_1x + p_2x^2 + ... + p_nx^n$. Extend Extend x to dual number $x + \dot{x}d$ where for now assume $\dot{x}$ has no particular meaning and is just some number. Then

$$\begin{aligned} P(x + \dot{x}\epsilon) &= p_0 + p_1(x + \dot{x}\epsilon) + ... + p_n(x + \dot{x}\epsilon)^n \\ &= p_0 + p_1x + p_2x^2 + ... + p_nx^n \\ &\quad + p_1\dot{x}\epsilon + 2p_2x\dot{x}\epsilon + ... + np_nx^{n-1}\dot{x}\epsilon \\ &= P(x) + P'(x)\dot{x}\epsilon \end{aligned}$$

If you set $\dot{x}$ to 1, then the coefficient of ϵ is the derivative of P evaluated at x. More generally, $f(a + b\epsilon) = f(a) + f'(a)b\epsilon$ (b doesn't have to be 1). Similarly, one may derive

$$\begin{aligned} \sin(x + \dot{x}\epsilon) &= \sin(x) + \cos(x)\dot{x}\epsilon \\ \cos(x + \dot{x}\epsilon) &= \cos(x) - \sin(x)\dot{x}\epsilon \\ e^{(x+\dot{x}\epsilon)} &= e^x + e^x\dot{x}\epsilon \\ \log(x + \dot{x}\epsilon) &= \log(x) + \dot{x}\frac{1}{x}\epsilon, x \neq 0 \\ \sqrt{x + \dot{x}\epsilon} &= \sqrt{x} + \dot{x}\frac{1}{2\sqrt{x}}\epsilon, x \neq 0 \end{aligned}$$

This can be extended to functions of many variables by introducing more dual components. For example, $f(x_1, x_2) = x_1x_2 \sin x_1$ extends to

$$f(x_1 + \dot{x_1}\epsilon_1, x_2 + \dot{x_2}\epsilon_2) = (x_1 + \dot{x_1}\epsilon_1)(x_2 + \dot{x_2}\epsilon_2) + \sin(x_1 + \dot{x_1}\epsilon_1) = x_1x_2 + (x_2 + \cos(x_1))\dot{x_1}\epsilon_1 + x_1\dot{x_2}\epsilon_2$$

where the coefficients of ϵ_i are the partial derivatives with respect to x_i.

2.4 Forward Mode AD

Consider again $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ whose evaluation trace is given above. For computing the derivative of f with respect to x_1, we start by associating with each intermediate variable v_i a derivative

$$\dot{v_i} = \frac{\partial v_i}{\partial x_1}$$

.

Applying chain rule to each elementary operation in the forward (primal) trace, we generate the corresponding tangent (derivative) trace (right hand side of augmented execution trace below). Evaluating the primals v_i in alternation with their corresponding tangents $\dot{v_i}$ gives us the required derivative of the final variable $\dot{v_5} = \partial y / \partial x_1$:

Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.652$	$\dot{y}$	$= \dot{v}_5$	$= \mathbf{5.5}$

This generalizes to computing the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with n independent input variables x_i and m dependent (output) variables y_j. In this case, each forward pass of AD is initialized by setting only one of the variables $\dot{x_i} = 1$ and the rest to 0. A run of the code with input $\mathbf{x} = \mathbf{a}$ computes

$$\dot{y_j} = \frac{\partial y_j}{\partial x_i}|_{\mathbf{x}=\mathbf{a}}, \quad j = 1, ..., m$$

and gives us one column of the Jacobian evaluated at the point $\mathbf{a}$. The full Jacobian can then be computed in n evaluations. This also gives us the ability to compute Jacobian-vector products in a matrix-free way, by initializing $\dot{\mathbf{x}} = \mathbf{r}$ and doing just one forward pass. When $f : \mathbb{R}^n \to \mathbb{R}$ this gives us the directional derivative along $\mathbf{r}$ (linear combination of partials) $\nabla f^\top \cdot \mathbf{r}$.

Forward mode can compute all derivatives dy_j/dx for $f : \mathbb{R} \to \mathbb{R}^m$ in just one pass. Conversely, for $f : \mathbb{R}^n \to \mathbb{R}$ we need n evaluations to get the gradient vector

$$\nabla f = \left(\frac{\partial y}{\partial x_1}, ..., \frac{\partial y}{\partial x_n} \right)$$

(a $1 \times n$ Jacobian built one column at a time with n forward evaluations)

2.5 Implementation of Forward Mode with Dual Numbers

Dual numbers are used in forward mode AD. The Dual Number addition and multiplication rules mirror differentiation rules:

$$(v + \dot{v}\epsilon) + (u + \dot{u}\epsilon) = (v + u)(\dot{v} + \dot{u})\epsilon$$
$$(v + \dot{v}\epsilon)(u + \dot{u}\epsilon) = (vu) + (v\dot{u} + \dot{v}u)\epsilon$$

If functions obey the following property:

$$\texttt{Property 1.} f(a + b\epsilon) = f(a) + f'(a)b\epsilon$$

then the chain rule works as expected on the dual number representation, which can be seen by applying the property twice:

$$f(g(v + \dot{v}\epsilon)) + f(g(v) + g'(v)\dot{v}\epsilon) = f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon$$

The coefficient on the right-hand side is exactly the derivative of the composition of f and g. This means that we can extract the derivative of a function by interpreting any non-dual number v as $(v + 0\epsilon)$ and evaluating the function in this non-standard way on an initial input with a coefficient of 1 for ϵ:

$$\frac{df(x)}{dx}|_{x=v} = \texttt{Eps}(\texttt{dual-version}(f)(v + 1\epsilon))$$

where `Eps` extracts $\epsilon's$ coefficient. In practice, a function f coded in a programming language would be fed into an AD tool and augmented with extra code (or interpreted) to handle dual operations so the function and its derivative are simultaneously computed. Note that these tools require source code transformation or operator overloading.

2.6 Reverse Mode AD

Reverse AD propagates derivatives backward from a given output. This is done by complementing each intermediate variable v_i with an adjoint

$$\bar{v_i} = \frac{\partial y_j}{\partial v_i}$$

Derivatives are computed in the second phase of a two-phase process. In the first phase, the original code is run *forward*, populating intermediate variables v_i and recording the dependencies in the computational graph through book-keeping. In the second phase, derivatives are calculated by propagating adjoints $\bar{v_i}$ in reverse, from outputs to inputs.

Recall the example $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$. Reverse Mode AD at work:

Forward Primal Trace		
$v_{-1} = x_1$		$= 2$
$v_0 = x_2$		$= 5$
$v_1 = \ln v_{-1}$		$= \ln 2$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$
$v_3 = \sin v_0$		$= \sin 5$
$v_4 = v_1 + v_2$		$= 0.693 + 10$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$
$y = v_5$		$= 11.652$

Reverse Adjoint (Derivative) Trace		
$\bar{x}_1 = \bar{v}_{-1}$		$= \mathbf{5.5}$
$\bar{x}_2 = \bar{v}_0$		$= \mathbf{1.716}$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5 = \bar{y}$	$= 1$	

Consider v_0. It only affects y through v_2 and v_3. Its contribution to the change in y is given by:

$$\frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0} \quad \text{or} \quad \bar{v_0} = \bar{v_2}\frac{\partial v_2}{\partial v_0} + \bar{v_3}\frac{\partial v_3}{\partial v_0}$$

In the table, this is computed in two steps:

$$\bar{v_0} = \bar{v_3}\frac{\partial v_3}{\partial v_0} \quad \text{and} \quad \bar{v_0} = \bar{v_0} + \bar{v_2}\frac{\partial v_2}{\partial v_0}$$

which appear in the table lined up with the lines in the forward trace from which these expressions originate. After the forward pass, we run the reverse pass of the adjoints, starting with $\bar{v_5} = \bar{y} = \frac{\partial y}{\partial y} = 1$. In the end we get derivatives $\frac{\partial y}{\partial x_1} = \bar{x_1}$ and $\frac{\partial y}{\partial x_2} = \bar{x_2}$ in just one reverse pass.

An advantage of reverse mode is that it is less costly to evaluate than forward mode for functions with many inputs. In the extreme case, $f : \mathbb{R}^n \to \mathbb{R}$, one application of reverse mode is sufficient to compute the full gradient ∇f, compared with the n passes of forward mode required. Because machine learning is focused on the gradient of scalar-valued objectives with respect to large numbers of parameters, this establishes the reverse mode as the mainstay technique.
Similar to matrix-free computation of JVP's with forward mode, reverse mode can be used for computing the transposed Jacobian-vector product $\mathbf{J}_f^\top \mathbf{r}$ by initializing the reverse phase with $\bar{\mathbf{y}} = \mathbf{r}$. Note that reverse mode has storage requirements that in the worst case grow in proportion to the number of operations in the function to be differentiated.

2.7 Forward or Backward

- As mentioned, forward is better for the case of $\mathbb{R} \to \mathbb{R}^m$ and reverse is better for the case of $\mathbb{R}^n \to \mathbb{R}$
- these are just two extremes: there is some optimal mix of approaches but it is NP-hard to find the best such approach.

3 Gradient-Based Optimization

Given an objective (loss) function $f(\mathbf{w}) : \mathbb{R}^n \to \mathbb{R}$ that one would like to minimize with respect to variable $\mathbf{w}$, one searches for (local) minimum $\mathbf{w}^\star = \arg\min_{\mathbf{w}} f(\mathbf{w})$ using a iterative steepest descent method. This is because usually f is complicated enough that one cannot simply set $\nabla f_{\mathbf{w}} = 0$ and solve for $\mathbf{w}$, a method used to analytically find minima/maxima.

Steepest Descent methods pick the descent direction $\mathbf{v}$ for which the directional derivative $\nabla f^\top \mathbf{v}$ is the most negative, subject to $||\mathbf{v}|| = 1$ for some choice of norm $||\cdot||$, and make updates to $\mathbf{w}$ by taking a step in that direction. When the norm is Euclidean, one recovers the classical gradient descent algorithm, which uses updates of the form $\Delta\mathbf{w} = -\eta\nabla_{\mathbf{w}} f$ where $\eta > 0$ is a step size.

4 How ML Libraries work

A typical machine learning library, PyTorch, works as follows. A user specifies some variables $\mathbf{w}_i$ to be optimized. Then the user implements some "forward" code that uses the $\mathbf{w}_i$ to do something like make a prediction. Following this, the user computes some loss function measuring how well (how badly) they did and saves the result in a variable called `loss`. The goal is to tune the $\mathbf{w}_i$ to minimize the value of `loss`.

During this computation, PyTorch builds a dependency graph of the expressions under the hood, where `loss` is the last vertex of the graph. A user then calls `loss.backward()` and Reverse-Mode AD uses the aforementioned adjoints to compute the derivatives of the loss with respect to all of the variables to be optimized $\mathbf{w}_i$ and saves these derivatives in a `.grad` field for each such variable.

Finally, an optimizer (usually some object) that knows about the variables to be optimized updates each one by reading the `.grad` field and applying something like gradient descent:

$$\mathbf{w}_i = \mathbf{w}_i - \eta \mathbf{w}_i\texttt{.grad}$$

perhaps with some additional tricks.

Actually, most practitioners and many researchers of machine learning exclusively use (and only know about) gradient descent, and don't know that there are more general steepest descent methods (corresponding to different choices of norms) are better for optimizing certain kinds of problems (because those methods update the variables differently from just taking a step in the negative gradient direction. Rajesh has a paper on this.

One interesting research direction is how to make an auto-differentiation + optimization system that doesn't need separate implementations of descent algorithms for each possible choice of norm. This might benefit from some form of abstraction.

5 Bayesian Probabilistic Modeling and Inference

When introducing new models, machine learning researchers have spent considerable effort on manual derivation of analytical derivatives to be used with optimization algorithms (error prone and time consuming)

[TODO]

6 Approximating Gradients of Expectations

[TODO]

7 Gradient Estimation Using Stochastic Computation Graphs

[TODO]

8 Citations

- Baydin, Pearlmutter, Radul, Siskind Automatic Differentiation in Machine Learning: a Survey 2018 https://arxiv.org/pdf/1502.05767.pdf
- The Simple Essence of Automatic Differentiation 2018 CONAL ELLIOTT, http://conal.net/papers/essence-of-ad/essence-of-ad-icfp.pdf
- Good autodiff and dual number slides Havard Berland 2006 http://www.pvv.ntnu.no/~berland/resources/autodiff-triallecture.pdf
- Marian Boykan Pour-El and Ian Richards. 1978. Differentiability properties of computable functions - A summary. Acta Cybernetica 4, 1 (1978), http://acta.bibl.u-szeged.hu/12271/1/cybernetica_004_fasc_001_123-125.pdf
- Marian Boykan Pour-El and Ian Richards. 1983. Computability and noncomputability in classical analysis. Transactions of the American Mathematical Society 275, 2 (1983)
- Mathoverflow about computability of differentiation: "John Myhill gave an example of a recursive function defined on a compact interval and having a continuous derivative that is not recursive [Michigan Math. J. 18 (1971), 97-98, MR0280373]. However, Pour-El and Richards have shown that if a recursive function defined on a compact interval has a continuous second derivative, then it has a recursive first derivative [Computability and noncomputability in classical analysis, TAMS 275 (1983), 539-560" https://mathoverflow.net/questions/35021/differentiability-of-computable-functions
- Michael Spivak. 1965. Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus. Addison-Wesley http://strangebeautiful.com/other-texts/spivak-calc-manifolds.pdf
- Gradient Estimation Using Stochastic Computation Graphs. 2016 Schulman, Heess, Weber, Abbeel https://arxiv.org/pdf/1506.05254.pdf
- Robert E. Wengert. A simple automatic derivative evaluation program. Communications of the ACM, 7:4634, 1964.
- Andreas Griewank and Andrea Walther. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Society for Industrial and Applied Mathematics, Philadelphia, 2008. doi: 10.1137/1.9780898717761.
- Mathoma (YouTube user). Dual Numbers Introduction. 2016. https://www.youtube.com/watch?v=63dt-pSKlk0