Automatic Differentiation and Backpropagation
CS701

Thomas Reps

[Based, in part, on notes taken by Naveen Neelakandan on December 1, 2015]

Abstract

This lecture discusses the relationship between *automatic differentiation* and *backpropagation.* Automatic differentiation (AD) is a technique that takes an implementation of a numerical function f (computed using floating-point numbers) and creates an implementation of f'. We explain several techniques for performing AD. For forward-mode AD, we give an explicit transformation of the program, as well as a way to implement AD using operator overloading. We then reformulate AD as a path problem over a so-called *computation graph.* The path problem can be solved in either direction: one direction corresponds to forward-mode AD; the other corresponds to reverse-mode AD.

Finally, we show how reverse-mode AD forms the basis of the backpropagation algorithm for training a neural net: in backpropagation, the weight adjustments computed for a neural net, a given input, and an expected output are found by performing reverse-mode AD on the computation graph for (i) the neural net, along with (ii) additional computation-graph elements for the sum-of-squared differences of the neural net's output with the expected output.

1 Introduction

This lecture provides another payoff for our having invested so much time on path problems. The point of the lecture is to show you how the core step used to train a neural net—namely, *backpropagation*—involves solving a path problem. Our path to presenting this concept is a bit indirect, but hopefully enlightening:

- forward-mode automatic differentiation
- reverse-mode automatic differentiation
- expressing a neural net as an expression DAG
- backpropagation

2 Forward-Mode Automatic Differentiation

This section[1] describes so-called *forward-mode automatic differentiation*, which provides a way to compute, with high accuracy, the derivative of a function defined by a program. An AD tool transforms a program that computes a numerical function $F(x)$ into a related program that computes the derivative $F'(x)$. These tools address the following issue: Suppose that you have a program `F(x)` that computes a numerical function $F(x)$. It is a very bad idea to try to compute $F'(x_0)$, the value of the derivative of F at x_0, by picking a small value `delta_x` and invoking the following

[1]This section is adapted from [17, §2].

program with the argument $\mathtt{x_0}$:[2]

```
float delta_x = ...⟨some small value⟩ ...;
float F′_naive(float x){
  return (F(x + delta_x) - F(x))/delta_x;
}
```
(1)

For a small enough value of `delta_x`, the values of $\mathtt{F(x_0+delta\_x)}$ and $\mathtt{F(x_0)}$ will usually be very close. Round-off errors in the computation of $\mathtt{F(x_0+delta\_x)}$ and $\mathtt{F(x_0)}$ are magnified by the subtraction of the two quantities, and further amplified by the division by the small quantity `delta_x`, which may cause the overall result to be useless. Automatic differentiation sidesteps this problem by computing derivatives in another fashion.

Automatic differentiation can be illustrated by means of the following example:

Example 2.1 [19]. Suppose that we have been given a collection of programs $\mathtt{f_i}$ for the functions f_i, $1 \leq i \leq k$, together with the program `Prod` shown below, which computes the function $Prod(x) = \prod_{i=1}^{k} f_i(x)$. In addition, suppose that we have also been given programs $\mathtt{f'_i}$ for the functions f'_i, $1 \leq i \leq k$. Finally, suppose that we wish to obtain a program `Prod′` that computes the function $Prod'(x)$. Column two of the table given below shows mathematical expressions for $Prod(x)$ and $Prod'(x)$. Column three shows two C++ procedures: Procedure `Prod` computes $Prod(x)$; procedure `Prod′` is the procedure that an automatic-differentiation system would create to compute $Prod'(x)$.

	Mathematical Notation	Programming Notation
Function	$Prod(x) = \prod_{i=1}^{k} f_i(x)$	`float Prod(float x){` `float ans = 1.0;` `for (int i = 1; i <= k; i++){` `ans = ans * fᵢ(x);` `}` `return ans;` `}`
Derivative	$Prod'(x) = \sum_{i=1}^{k} f'_i(x) * \prod_{j \neq i} f_j(x)$	`float Prod′(float x){` `float ans′ = 0.0;` `float ans = 1.0;` `for (int i = 1; i <= k; i++){` `ans′ = ans′ * fᵢ(x) + ans * f′ᵢ(x);` `ans = ans * fᵢ(x);` `}` `return ans′;` `}`

Notice that program `Prod′` resembles program `Prod`, as opposed to `F′_naive` (see box (1)). `Prod′` preserves accuracy in its computation of the derivative because, as illustrated below in Example 2.2, it is based on the rules for the exact computation of derivatives, rather than on the kind of computation performed by `F′_naive`. □

[2]In the remainder of this section, `Courier Font` is used to denote functions defined by programs, whereas *Italic Font* is used to denote mathematical functions. That is, $F(x)$ denotes a function (evaluated over real numbers), whereas `F(x)` denotes a program (evaluated over floating-point numbers). We adhere to this convention both in concrete examples that involve C++ code, as well as in more abstract discussions in order to distinguish between a mathematical function and a program that implements the function.

The example programs are all written in C++, although the ideas described apply to other programming languages—including functional programming languages (cf. [8, 9])—as well as to other imperative languages. To emphasize the links between mathematical concepts and their implementations in C++, we take the liberty of sometimes using ′ and/or subscripts on C++ identifiers.

The transformation illustrated above is merely one instance of a general transformation that can be applied to any program: Given a program `G` as input, the transformation produces a derivative-computing program $\texttt{G}'$. The method for constructing $\texttt{G}'$ is as follows:

- For each variable `v` of type `float` used in `G`, another `float` variable $\texttt{v}'$ is introduced.
- Each statement in `G` of the form "`v =` *exp*`;`", where *exp* is an arithmetic expression, is transformed into "$\texttt{v}'$ `=` exp'`;` `v =` *exp*`;`", where exp' is the expression for the derivative of *exp*. If *exp* involves calls to a procedure `g`, then exp' may involve calls to both `g` and $\texttt{g}'$.
- Each return statement in `G` of the form "`return v;`" is transformed into "`return` $\texttt{v}'$`;`".

In general, this transformation can be justified by appealing to the chain rule of differential calculus (see below).

Example 2.2 For Example 2.1, we can demonstrate the correctness of the transformation by symbolically executing $\texttt{Prod}'$ for a few iterations, comparing the values of $\texttt{ans}'$ and `ans` (as functions of `x`) at the start of each iteration of the for-loop:

Iteration	Value of $\texttt{ans}'$ (as a function of `x`)	Value of `ans` (as a function of `x`)
0	`0.0`	`1.0`
1	$\texttt{f}_1'(\texttt{x})$	$\texttt{f}_1(\texttt{x})$
2	$\texttt{f}_1'(\texttt{x}) * \texttt{f}_2(\texttt{x}) + \texttt{f}_1(\texttt{x}) * \texttt{f}_2'(\texttt{x})$	$\texttt{f}_1(\texttt{x}) * \texttt{f}_2(\texttt{x})$
3	$\texttt{f}_1'(\texttt{x}) * \texttt{f}_2(\texttt{x}) * \texttt{f}_3(\texttt{x})$ $+ \texttt{f}_1(\texttt{x}) * \texttt{f}_2'(\texttt{x}) * \texttt{f}_3(\texttt{x})$ $+ \texttt{f}_1(\texttt{x}) * \texttt{f}_2(\texttt{x}) * \texttt{f}_3'(\texttt{x})$	$\texttt{f}_1(\texttt{x}) * \texttt{f}_2(\texttt{x}) * \texttt{f}_3(\texttt{x})$
...	...	...
k	$\sum_{i=1}^{k} \texttt{f}_\texttt{i}'(\texttt{x}) * \prod_{j \neq i} \texttt{f}_\texttt{j}(\texttt{x})$	$\prod_{i=1}^{k} \texttt{f}_\texttt{i}(\texttt{x})$

The loop maintains the invariant that, at the start of each iteration, $\texttt{ans}'(\texttt{x}) = \frac{d}{d\texttt{x}}\texttt{ans}(\texttt{x})$.

The value of $\texttt{ans}'$ on the 3rd iteration would actually be computed with the terms grouped as follows:

$$(\texttt{f}_1'(\texttt{x})*\texttt{f}_2(\texttt{x})+\texttt{f}_1(\texttt{x})*\texttt{f}_2'(\texttt{x}))*\texttt{f}_3(\texttt{x})+(\texttt{f}_1(\texttt{x})*\texttt{f}_2(\texttt{x}))*\texttt{f}_3'(\texttt{x}). \qquad (2)$$

Terms have been expanded in the table given above to clarify how $\texttt{ans}'$ builds up a value that is equivalent—from the standpoint of evaluation in real arithmetic—to $\texttt{Prod}'(\texttt{x}) = \sum_{i=1}^{k} \texttt{f}_\texttt{i}'(\texttt{x}) * \prod_{j \neq i} \texttt{f}_\texttt{j}(\texttt{x})$. However, Eqn. (2) shows what happens operationally: for a summand of the form

$$\texttt{f}_1(\texttt{x}) * \ldots * \texttt{f}_{\texttt{i}-1}(\texttt{x}) * \texttt{f}_\texttt{i}' * \texttt{f}_{\texttt{i}+1}(\texttt{x}) * \ldots * \texttt{f}_\texttt{k}(\texttt{x}),$$

the product $\texttt{f}_1(\texttt{x}) * \ldots * \texttt{f}_{\texttt{i}-1}(\texttt{x})$ is built up in variable `ans` during the first $i-1$ iterations; then, in iteration i, it is multiplied by $\texttt{f}_\texttt{i}'$ and added to $\texttt{ans}'$; thereafter, on subsequent iterations, it resides in $\texttt{ans}'$ as $\texttt{ans}'$ is multiplied by $\texttt{f}_{\texttt{i}+1}(\texttt{x}) * \ldots * \texttt{f}_\texttt{k}(\texttt{x})$. □

For the automatic-differentiation approach, we did not really need to make the assumption that we were given programs $\texttt{f}_\texttt{i}'$ for the functions f_i', $1 \leq i \leq k$; instead, the programs $\texttt{f}_\texttt{i}'$ can be generated from the programs $\texttt{f}_\texttt{i}$ by applying the same statement-doubling transformation that was applied to `Prod`.

In languages that support operator overloading, such as C++, Ada, and Pascal-XSC, automatic differentiation can be carried out by defining a new data type that has fields for both the value and

```
enum ArgDesc { CONST, VAR };
class FloatD {
 public:
  float val′;
  float val;
  FloatD(ArgDesc,float);
};
// Constructor to convert a constant
// or a value for the independent
// variable to a FloatD
FloatD::FloatD(ArgDesc a, float v){
  switch (a) {
    case CONST:
      val′ = 0.0;
      val = v;
    break;
    case VAR:
      val′ = 1.0;
      val = v;
    break;
  }
}
FloatD operator+(FloatD a, FloatD b){
   FloatD ans;
   ans.val′ = a.val′ + b.val′;
   ans.val = a.val + b.val;
   return ans;
}
FloatD operator*(FloatD a, FloatD b){
  FloatD ans;
  ans.val′ = a.val * b.val′ + a.val′ * b.val;
  ans.val = a.val * b.val;
  return ans;
}
```

Figure 1: A differentiation-arithmetic class.

the derivative, and overloading the arithmetic operators to carry out appropriate manipulations of both fields [12, 13], along the lines of the definition of the C++ class `FloatD`, shown in Fig. 1. A class such as `FloatD` is called a *differentiation arithmetic* [14, 15, 16].

The transformation then amounts to changing the types of each procedure's formal parameters, local variables, and return value (including those of the $\mathtt{f_i}$).[3]

Example 2.3 Using class `FloatD`, the `Prod` program of Example 2.1 can be handled as follows:

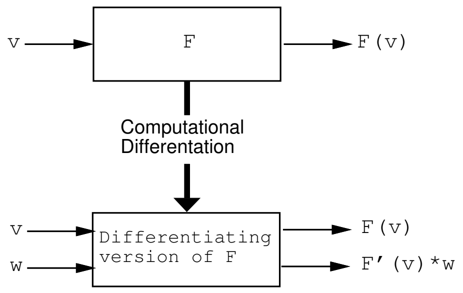
[3]We have referred to automatic differentiation as a "program transformation," which may conjure up the image of a tool that perform source-to-source rewriting fully automatically. Although source-to-source rewriting is one possible embodiment, here the term "transformation" will also include the use of C++ classes in which the arithmetic operators have been overloaded. With the latter approach, rewriting might be carried out by a preprocessor, but might also be performed by hand, because usually only light rewriting of the program source text is required.

```
float f1(float x){...}            ⇒ FloatD f1(const FloatD &x){...}
         ⋮                                 ⋮
float fk(float x){...}            ⇒ FloatD fk(const FloatD &x){...}
float Prod(float x){                FloatD Prod(const FloatD &x){
  float ans = 1.0;                    FloatD ans(CONST,1.0); // ans = 1.0
  for (int i = 1; i <= k; i++){       for (int i = 1; i <= k; i++){
    ans = ans * fi(x);            ⇒     ans = ans * fi(x);
  }                                   }
  return ans;                         return ans;
}                                   }
                                    float Prod'(float x){
                                      FloatD xD(VAR,x);
                                  ⇒   return Prod(xD).val';
                                    }
```

By changing the types of the formal parameters, local variables, and the return values of `Prod` and the $\mathtt{f_i}$ (and making a slight change to the initialization of `ans` in `Prod`), the program now carries around derivative values (in the `val′` field) in addition to performing all of the work performed by the original program. Because of the C++ overload-resolution mechanism, the $\mathtt{f_i}$ procedures invoked in the fourth line of the transformed version of `Prod` are the *transformed* versions of the $\mathtt{f_i}$ (i.e., the $\mathtt{f_i}$ of type `FloatD` → `FloatD`).

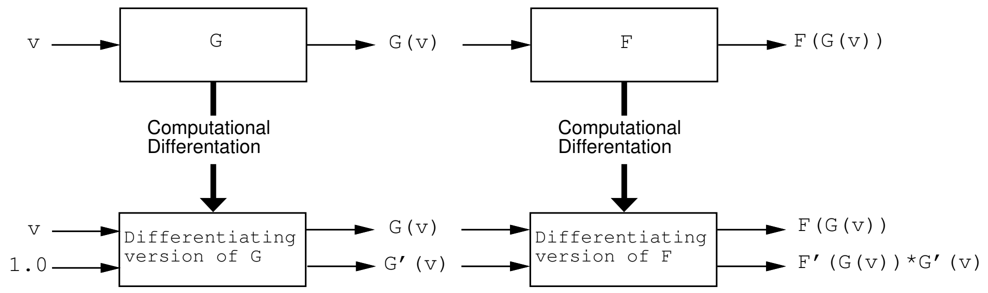
The value of `Prod`'s derivative at `v` is obtained by calling `Prod′(v)`. □

In a differentiation arithmetic, each procedure in the user's program, such as `Prod` and the $\mathtt{f_i}$ in Example 2.3, can be viewed as a box that maps two inputs to two outputs, as depicted below:



In particular, in each differentiating version of a user-defined or library procedure `F`, the lower-right-hand output produces the value `F′(v)*w`.

An input value `v` for the formal parameter is treated as a pair `(v,1.0)`. Boxes like the one shown above "snap together": when `F` is composed with `G` (and the input is `v`), the output value on the lower-right-hand side is `F′(G(v))*G′(v)`, which agrees with the usual expression for the chain rule for the first-derivative operator:



v
G
G(v)
F
F(G(v))
Computational Differentation
Computational Differentation
v
1.0
Differentiating version of G
G(v)
G′(v)
Differentiating version of F
F(G(v))
F′(G(v))*G′(v)

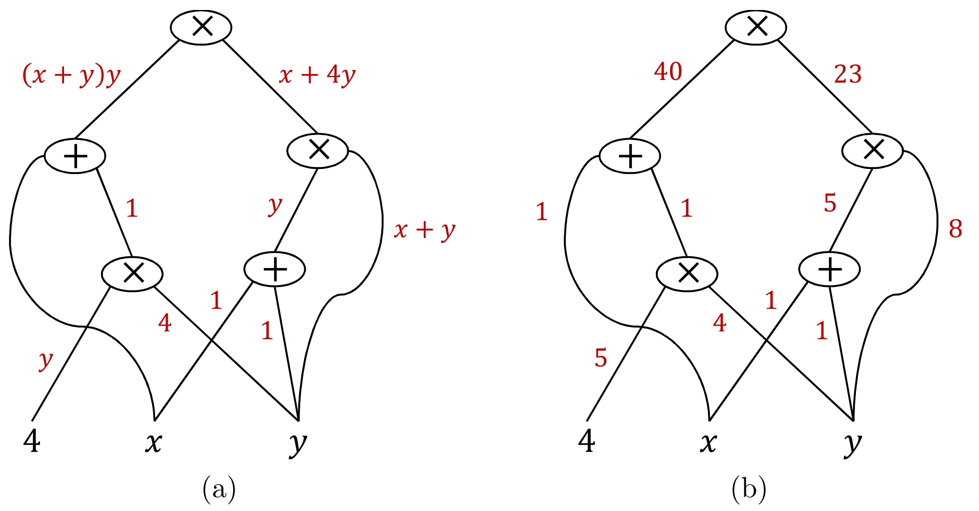


Figure 2: (a) A symbolic computation graph for the evaluation of the expression $(x + 4y) * ((x + y) * (y))$. (b) The computation graph for $(x + 4y) * ((x + y) * (y))$ when $x = 3$ and $y = 5$. (It is recommended to view the figures in this document on-line so that the colors in them can be seen.)

It should be noted that the transformation that was illustrated in Example 2.1 is not fully general in that it does not yield a procedure that can be composed with other transformed procedures. To create a composable transformed procedure, the transformation would, in essence, have to make changes that mimic all of the actions of the version created in Example 2.3 using class `FloatD`: the procedure would have to take two arguments, `x` and $\texttt{x}'$; pass these on to composable transformed versions of the $\texttt{f}_\texttt{i}$; and return a pair $\langle \texttt{ans}, \texttt{ans}' \rangle$, instead of $\texttt{ans}'$ alone.

The automatic-differentiation technique summarized above is what is known as *forward-mode* differentiation. The availability of overloading makes it possible to implement forward-mode automatic differentiation conveniently, by packaging it as a differentiation-arithmetic class, as illustrated above. The alternative to the use of overloading is to build a special-purpose preprocessor to carry out the statement-doubling transformation that was illustrated in Exs. 2.1 and 2.2. Examples of systems that use the latter approach include ADIFOR [1, 2] and ADIC [3].

When the number of independent variables is much greater than the number of dependent variables, a different technique, *reverse-mode automatic-differentiation* [10, 18, 7, 5, 6], provides theoretically better performance—a greatly reduced number of computation steps—but at the cost of the need to store or recompute intermediate values that affect the final result nonlinearly. Reverse-mode automatic differentiation is explained in §3–§5. In §6, we describe how reverse-mode automatic differentiation is the workhorse of the *backpropagation algorithm* used to train a neural network.

Forward mode and reverse mode are the endpoints of a spectrum of algorithmic techniques; in practice, automatic-differentiation tools optimize runtime and memory requirements by exploiting associativity properties of the chain rule to permit forward mode and reverse mode to be used in different parts of the computation [4].

3 Computation Graphs

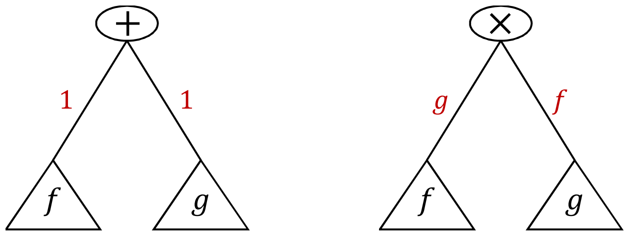
In previous lectures, we described how interprocedural dataflow-analysis problems can be formalized as path problems on graphs. Here, we will reformulate automatic differentiation—both forward-mode and reverse-mode—as a path problem. To do so, we introduce the notion of a *computation graph* [11]. A computation graph is a labeled directed-acyclic graph (DAG). We start with the—in general—multi-source/multi-sink DAG that consists of all operations performed during execution, connected by edges that represent how outputs from one operation were used as inputs to another operation. Fig. 2(a) shows an example of a computation graph for the following function:

$$h(x, y) = (x + 4y) * ((x + y) * (y)). \tag{3}$$

We will use Eqn. (3) as a running example in the rest of this document. Note that by expanding the right-hand-side expression, $h(x, y)$ equals $x^2y + 5xy^2 + 4y^3$ (i.e., it is a multivariate polynomial).

To formulate automatic differentiation as a path problem, we have to define the values that label a computation graph's edges and to define two operators, *extend* and *combine*—as we did for shortest-distance problems and dataflow-analysis problems. The path problem we need to define here is relatively straightforward: the extend operator is *multiplication* and the combine operator is *addition*. Note that unlike in the dataflow-analysis case, the combine operator is not idempotent: for instance, $1 + 1 = 2$ and not 1! However, the absense of idempotence does not create difficulties for automatic differentiation because we only need to solve path problems on a *DAG*; they can always be solved using a traversal of the DAG in topological order.

The edges are labeled according to the following scheme:



The labeling schema given above is really a *symbolic* version of the method used to construct a computation graph; Fig. 2(a) is an example of a symbolic computation graph. In an actual computation graph, we will have specific values for x and y, as well as for each of the subexpressions used as labels: $(x + y)y$, $x + 4y$, and $x + y$. Fig. 2(b) shows the computation graph for $(x + 4y) * ((x + y) * (y))$ when $x = 3$ and $y = 5$.

4 Computing Partial Derivatives Using a Computation Graph

The reason why computation graphs are interesting is because it is possible to compute partial derivatives by solving single-source/single-target path problems on a computation graph. Fig. 3 returns to the example of $h(x, y) = (x + 4y) * ((x + y) * (y))$, and illustrates how $\frac{\partial h(x,y)}{\partial x}$ and $\frac{\partial h(x,y)}{\partial y}$, when $x = 3$ and $y = 5$, can be computed via forward and backward single-source/single-target path problems.

The calculations for $\frac{\partial h}{\partial x}$ depicted in Fig. 3(a) and Fig. 3(b) involve summing over all paths from x to the root of the computation graph for h. This sum can be computed in two directions (and the same result is obtained via either method): in *forward-mode automatic differentiation*, the path-value computations proceed from the leaf x to the root of the computation graph. (Forward-mode differentiation tracks how the value at each node is affected by a change in the value of the input x.) In contrast, with *reverse-mode automatic differentiation* the path-value computations proceed from the root of the computation graph to the leaf x. (Reverse-mode differentiation tracks how the output value of the function is affected by a change in the value at a given node in the computation graph.)

For example, the green values shown in Fig. 3(a) and Fig. 3(b) indicate the intermediate path-values computed for nodes that are relevant to the final calculation during the forward and backward traversals, respectively. Because we are working with a DAG, the values can be computed in linear

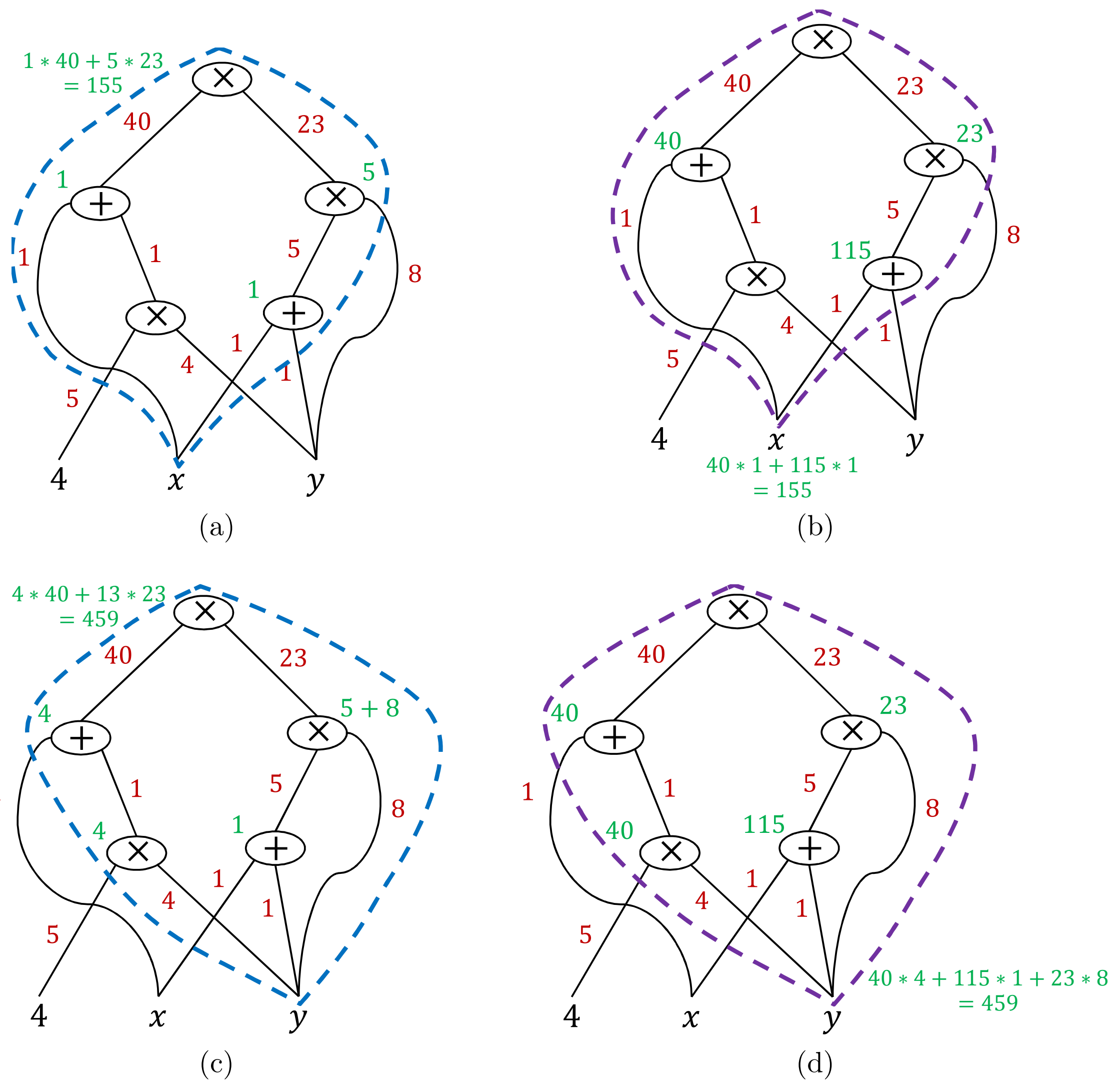
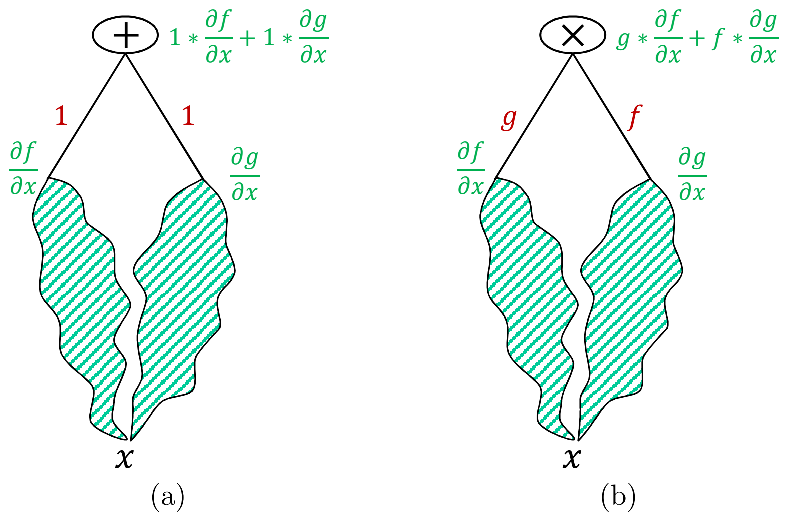


Figure 3: The computation of partial derivatives of $h(x, y)$, when $x = 3$ and $y = 5$, via forward and backward single-source/single-target path problems. (a) $\frac{\partial((x+4y)*((x+y)*(y)))}{\partial x} = 155$ via a forward path problem; (b) $\frac{\partial((x+4y)*((x+y)*(y)))}{\partial x} = 155$ via a backward path problem; (c) $\frac{\partial((x+4y)*((x+y)*(y)))}{\partial y} = 459$ via a forward path problem; (d) $\frac{\partial((x+4y)*((x+y)*(y)))}{\partial y} = 459$ via a backward path problem;

time in the size of the relevant portion of the computation graph by traversing that portion of the computation graph in topological order.

In the case of forward mode AD, the diagrams in Fig. 4 depict the main elements of the induction step in an inductive proof of correctness of the (forward) path-problem method. The induction hypothesis is that the single-source/single-target forward path problem in each of the shaded regions computes the indicated partial derivatives. The continuation of the path computations in a $(+, \times)$ path problem yield the respective indicated values $1 * \frac{\partial f}{\partial x} + 1 * \frac{\partial g}{\partial x}$ and $g * \frac{\partial f}{\partial x} + f * \frac{\partial g}{\partial x}$.

However, if we know that the forward-path-problem method is a correct method for computing partial derivatives, then the correctness of the backward-path-problem method follows immediately from the associativity and commutativity of $+$ and $\times$.



$1 * \frac{\partial f}{\partial x} + 1 * \frac{\partial g}{\partial x}$
1
1
$\frac{\partial f}{\partial x}$
$\frac{\partial g}{\partial x}$
x
(a)
$g * \frac{\partial f}{\partial x} + f * \frac{\partial g}{\partial x}$
g
f
$\frac{\partial f}{\partial x}$
$\frac{\partial g}{\partial x}$
x
(b)

Figure 4: The essence of the induction step in an inductive proof of correctness of the method for computing a partial derivative by solving a forward path problem (forward-mode AD). Diagrams (a) and (b) show how information is propagated from children to their parent in the case of a +-node and a ×-node, respectively.

Example 4.1 Let us now return to Eqn. (3) and show that the values computed in Fig. 3(a) and Fig. 3(b) are correct by computing the partial derivative by the more usual rules:

$$\begin{aligned} \left.\frac{\partial h(y,x)}{\partial x}\right|_{x=3,y=5} &= \left.\frac{\partial(x^2y + 5xy^2 + 4y^3)}{\partial x}\right|_{x=3,y=5} \\ &= 2xy + 5y^2\big|_{x=3,y=5} \\ &= 30 + 125 \\ &= 155 \end{aligned}$$

Another way to check our work is to solve the path problems from Fig. 3(a) and Fig. 3(b) symbolically, using Fig. 2(a), and show that both methods produce the answer $2xy + 5y^2$.
Forward path problem:

$$\begin{aligned} \frac{\partial h(y,x)}{\partial x} &= 1 * (x+y)y + 1 * y * (x+4y) \\ &= xy + y^2 + yx + 4y^2) \\ &= 2xy + 5y^2 \end{aligned}$$

Backward path problem:

$$\begin{aligned} \frac{\partial h(y,x)}{\partial x} &= (x+y)y * 1 + (x+4y) * y * 1 \\ &= xy + y^2 + xy + 4y^2 \\ &= 2xy + 5y^2 \end{aligned}$$

□

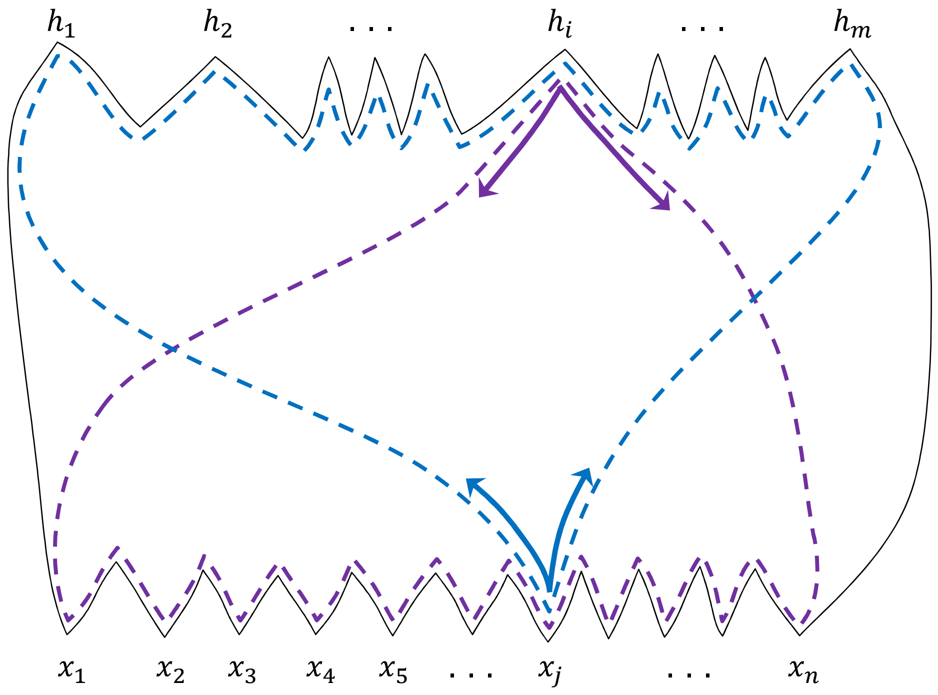


Figure 5: Depiction of the cost difference between forward-mode and reverse-mode automatic differentiation, depending on what set of partial derivatives is desired.

Forward-Mode Versus Reverse-Mode

At first glance, one may think because forward-mode and reverse-mode automatic differentiation compute the same answer, neither method is to be preferred over the other. However, the two methods may have substantially different costs, depending on the number of inputs and outputs of the function.

To understand the cost difference between forward-mode and reverse-mode automatic differentiation, consider a multivariate function $\vec{h}(\vec{x})$ that maps a vector of n inputs $\langle x_1, x_2, \ldots, x_n \rangle$ to a vector of m outputs $\langle h_1(\vec{x}), h_2(\vec{x}), \ldots, h_m(\vec{x}) \rangle$, as depicted in Fig. 5. Fig. 5 illustrates how forward-mode and reverse-mode can have much different costs, depending on what set of partial derivatives is desired:

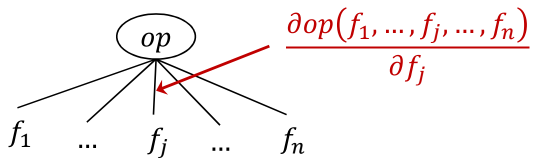
- One extreme is when we want to compute the set $\left\{ \frac{\partial h_k}{\partial x_j} \mid 1 \leq k \leq m \right\}$ of *all* partial derivatives of $\{h_k(\vec{x}) \mid 1 \leq k \leq m\}$ with respect to *some* input x_j. In effect, $n = 1$ (although we must have available the results of any sub-computation that can affect the value of any of the $h_k(\vec{x})$). In this case, the set of partial derivatives should be computed by solving a single-source/multi-target path problem, which can be performed using a single forward pass over the computation graph. This computation corresponds to forward-mode automatic differentiation, and is depicted in blue in Fig. 5.
- Another extreme is when we want to compute the set $\left\{ \frac{\partial h_i}{\partial x_k} \mid 1 \leq k \leq n \right\}$ of partial derivatives of *some* $h_i(\vec{x})$ with respect to *all* inputs $\{x_k \mid 1 \leq k \leq n\}$. In effect, $m = 1$ (and we really only need the computation graph for $h_i(\vec{x})$). In this case, the set of partial derivatives should be computed by solving a multi-source/single-target path problem, which can be performed using a single backward pass over the computation graph. This computation corresponds to reverse-mode automatic differentiation, and is depicted in purple in Fig. 5.

More generally, we might want the set $\left\{\frac{\partial h_k}{\partial x_l} \mid k \in K \subseteq \{1,\ldots,m\} \wedge l \in L \subseteq \{1,\ldots,n\}\right\}$. There is a trade-off between performing $|L|$ forward passes or $|K|$ backward passes. Each forward pass computes $\left\{\frac{\partial h_k}{\partial x_j} \mid k \in K\right\}$ for some $j \in L$. Each backward pass computes $\left\{\frac{\partial h_i}{\partial x_l} \mid l \in L\right\}$ for some $i \in K$. In this case, reverse-mode automatic differentiation is preferred when $|L| \gg |K|$. Forward-mode automatic differentiation is preferred when $|L| \ll |K|$.

As we will see in §6, when training a neural network, the number of inputs is typically large, and the number of outputs is 1. Consequently, the backpropagation algorithm for neural-net training employs reverse mode.

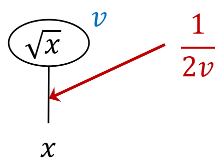
5 Derivatives for Other Operators

In this section, we describe how to construct the computation graph when the computation contains operators other than addition and multiplication. In general, for an occurrence of an n-ary operator, the label on the j^{th} argument is the value of the partial derivative $\frac{\partial op(f_1,\ldots,f_j,\ldots,f_n)}{\partial f_j}$:


op
$\frac{\partial op(f_1,\ldots,f_j,\ldots,f_n)}{\partial f_j}$
f_1
...
f_j
...
f_n

It is easy to see that the two labeling schemes for addition and multiplication depicted at the end of §3 fit the general pattern. We now consider several other examples.

Square Root. The square-root operator $\sqrt{x} \overset{\text{def}}{=} x^{\frac{1}{2}}$ is a unary operator; consequently, there is only one incoming edge to a square-root node. The value on the edge is the value of the partial derivative $\frac{\partial\sqrt{x}}{\partial x} = \frac{\partial(x^{\frac{1}{2}})}{\partial x} = \frac{1}{2}x^{-\frac{1}{2}} = \frac{1}{2\sqrt{x}}$. Consequently, for an occurrence of the square-root operator that produces the value v, the label that should be placed on the incoming edge is $\frac{1}{2v}$.



Note that the square-root operator is different from addition and multiplication in that the value on the edge is a function of the *output* of the operator.

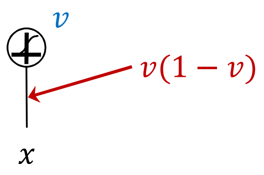
Sigmoid. One of the operations used in neural nets is the sigmoid operation, which is also unary:

$$\sigma(x) \overset{\text{def}}{=} \frac{1}{1+e^{-x}}.$$

The value on the edge is the value of the partial derivative

$$\begin{aligned}\frac{\partial\sigma(x)}{\partial x} &= \frac{\partial(1+e^{-x})^{-1}}{\partial x}\\ &= -1(1+e^{-x})^{-2}e^{-x}(-1)\\ &= \frac{e^{-x}}{(1+e^{-x})^2}\\ &= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{e^{-x}}{1+e^{-x}}\right)\\ &= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right)\\ &= \sigma(x)(1-\sigma(x))\end{aligned}$$

Consequently, for an occurrence of the sigmoid operator that produces the value v, the label that should be placed on the incoming edge is $v(1-v)$.

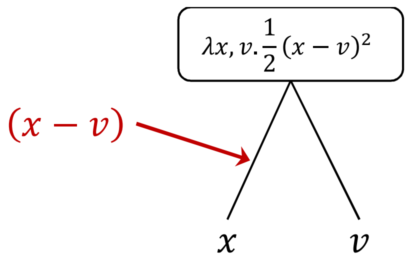


Squares of Differences. Consider the following 2-argument function used for computing one term in a sum of squares of differences:

$$\lambda x, v. \frac{1}{2}(x-v)^2.$$

In the context of training a neural net, the first argument will be the value computed by one output node of the neural net. The second argument will be the expected value of a training example, and thus is considered to be a constant. Consequently, we are only interested in the value on the left edge, which is the value of the partial derivative

$$\frac{\partial \frac{1}{2}(x-v)^2}{\partial x} = x - v.$$



6 Backpropagation for Training a Neural Net

In this section, we explain how reverse-mode automatic differentiation forms the basis of the back-propagation algorithm for training a neural net: in backpropagation, the weight adjustments computed for a neural net, a given input, and an expected output are found by performing reverse-mode automatic differentiation on the computation graph for (i) the neural net, along with (ii) additional computation-graph elements for the sum of the squared differences of the neural net's output with the expected output.

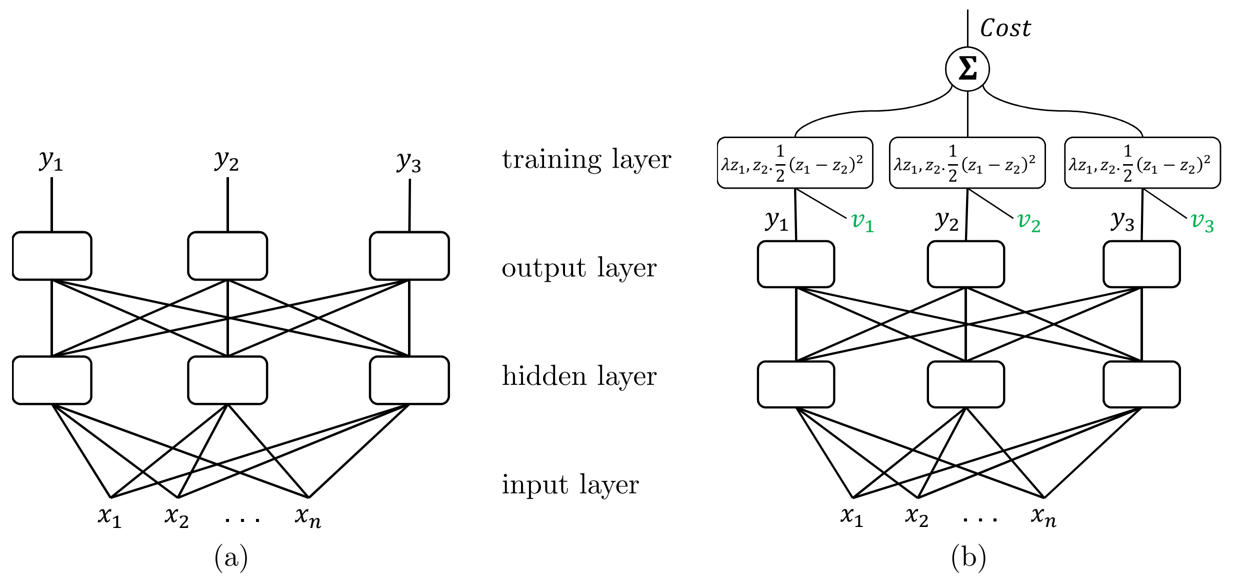


Figure 6: (a) A neural net with one hidden layer. (b) Circuit used for training a neural net.

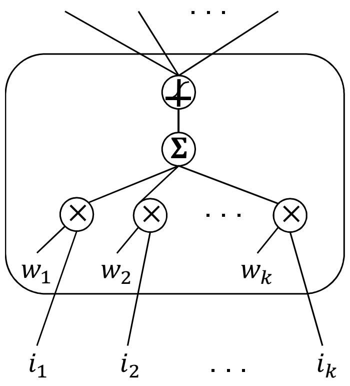


Figure 7: Structure of a node in a neural network.

A neural network has an input layer, an output layer, and at least one hidden layer in the middle, as shown in Fig. 6(a). When one looks into one of the nodes in the network, one typically has what is depicted in Fig. 7:

- a node has some number of inputs $i_1, \ldots, i_k$, which are multiplied respectively by the weights $w_1, \ldots, w_k$
- these products are then summed
- a sigmoid function is applied to the sum
- the output of the sigmoid function goes off to the next layer in the network—in general, to multiple nodes in the next layer

A neural network is trained by a "hill-climbing" procedure. One repeatedly presents it with training examples—i.e., pairs of vectors $\langle \vec{x}, \vec{v} \rangle$, where $\vec{v}$ represents the desired output for input $\vec{x}$—drawn from some suite of training data. For understanding how training works, it is useful to think of the neural net as being equipped with an additional layer that computes a cost $Cost(\vec{y}, \vec{v})$ by computing the sum of the squared differences of the network's actual output for input $\vec{x}$ (i.e.,

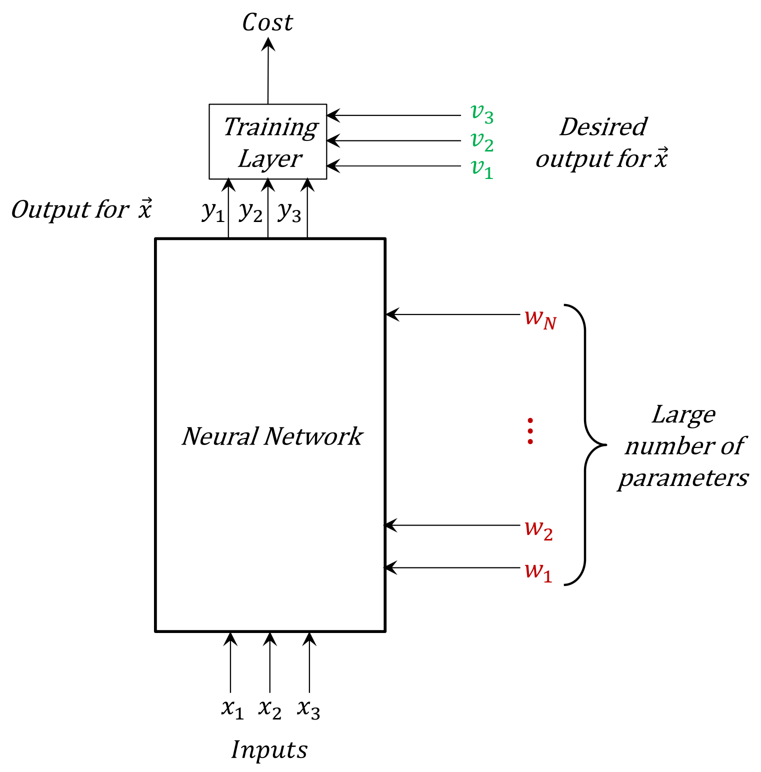


Figure 8: A neural network has three kinds of inputs: (i) the input data $\vec{x}$, (ii) the parameters $\vec{w}$, and (iii) the desired output $\vec{v}$. To perform back-propagation, one needs to compute the partial derivatives $\left\{\frac{\partial Cost}{\partial w_i}\right\}$. (Adapted from a slide in a POPL 2022 talk by Faustyna Krawiec.)

$\vec{y}$) and the expected output $\vec{v}$—see Fig. 6(b). The use of the sum-of-squared differences

$$\sum_{1 \le j \le m} \frac{1}{2}(y_j - v_j)^2$$

helps to accommodate noise in the training data. (The training data might contain mistakes due to human error, or it might consist of data observed from the environment via sensors, which will introduce measurement error.)

For each $\langle \vec{x}, \vec{v} \rangle$ pair, the weights in each of the nodes are adjusted by

1. using reverse-mode automatic differentiation to compute $\frac{\partial Cost}{\partial w_i}$, for each w_i that is a weight component of one of the hidden or output nodes in the neural network
2. making the adjustment $w_i := w_i + \eta \frac{\partial Cost}{\partial w_i}$, where η is a small constant that controls the rate of hill-climbing.

Training continues until the weight adjustments fall below some desired threshold.

Note that when training a neural network using automatic differentation, it is the set of weights $\{w_i\}$ that are considered to be the inputs of the computation graph with respect to which we want to compute the partial derivatives $\left\{\frac{\partial Cost}{\partial w_i}\right\}$ (see Fig. 8). The values from $\vec{x}$ are treated as constants for the purposes of a given round of training. Moreover, *Cost* is the only output, and thus the number of outputs is 1. Consequently, automatic differentiation is performed using reverse mode.

Fig. 9 shows a full neural-net-training example in schematic form. The figure shows the computation graph for a two-layer neural net, including the training layer. The edges of the graph

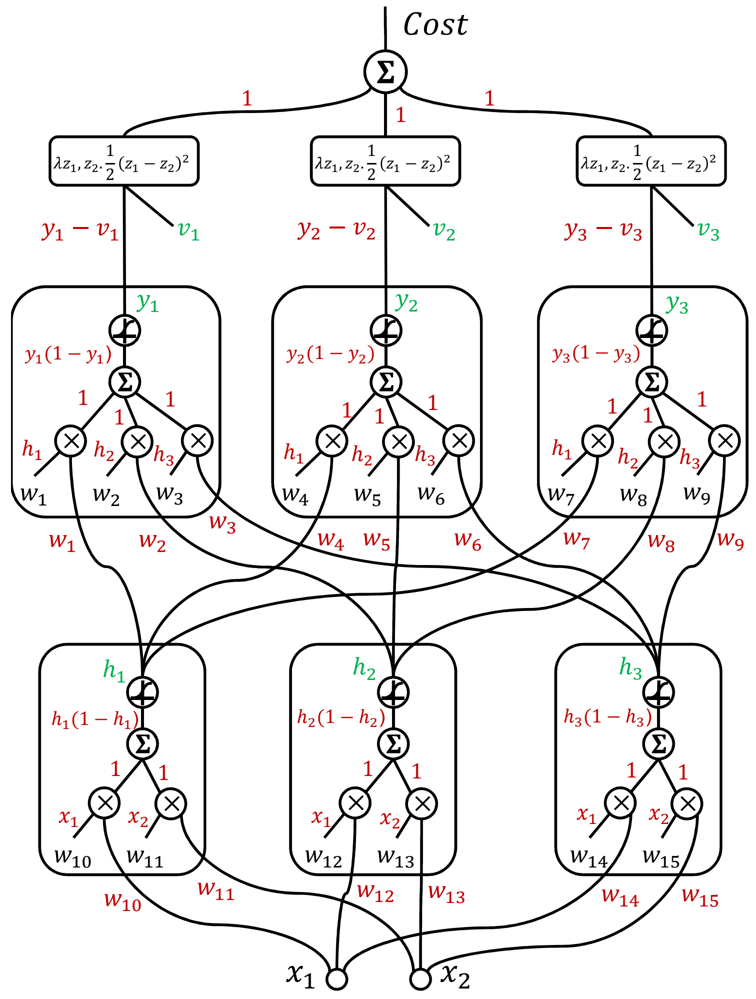


Figure 9: A back-propagation example in schematic form.

are annotated with the edge-weights used in the path problem for automatic differentiation. With reverse-mode automatic differentiation, the desired set of partial derivatives $\left\{\frac{\partial Cost}{\partial w_i}\right\}$ can be computed using a single backward pass over the computation graph.

It is instructive to consider in more detail a couple of the partial derivatives that are computed. For instance, there is only one path from *Cost* to w_4, and thus $\frac{\partial Cost}{\partial w_4}$ can be computed as follows:

$$\begin{aligned}\frac{\partial Cost}{\partial w_4} &= 1 \times (y_2 - v_2) \times y_2(1 - y_2) \times 1 \times h_1 \\ &= (y_2 - v_2)y_2(1 - y_2)h_1\end{aligned}$$

In contrast, there are multiple paths from *Cost* to w_{14}:

$$\begin{aligned}\frac{\partial Cost}{\partial w_{14}} &= \begin{pmatrix} \ 1 \times (y_1 - v_1) \times y_1(1 - y_1) \times 1 \times w_3 \times h_3(1 - h_3) \times 1 \times x_1 \\ +\ 1 \times (y_2 - v_2) \times y_2(1 - y_2) \times 1 \times w_6 \times h_3(1 - h_3) \times 1 \times x_1 \\ +\ 1 \times (y_3 - v_3) \times y_3(1 - y_3) \times 1 \times w_9 \times h_3(1 - h_3) \times 1 \times x_1 \end{pmatrix} \\ &= \begin{pmatrix} \ (y_1 - v_1)y_1(1 - y_1)w_3 \\ +\ (y_2 - v_2)y_2(1 - y_2)w_6 \\ +\ (y_3 - v_3)y_3(1 - y_3)w_9 \end{pmatrix} h_3(1 - h_3)x_1\end{aligned}$$

These results are then used to adjust the values of w_4 and w_{14} by making the adjustments

$$w_4 = w_4 + \eta \frac{\partial Cost}{\partial w_4}$$
$$w_{14} = w_{14} + \eta \frac{\partial Cost}{\partial w_{14}}$$

References

[1] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[2] C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Comp. Sci. and Eng.*, 3:18–32, 1996.

[3] C. Bischof, L. Roh, and A. Mauer. ADIC: An extensible automatic differentiation tool for ANSI-C. *Software – Practice and Experience*, 27(12):1427–1456, 1997.

[4] C.H. Bischof and M.R. Haghighat. Hierarchical approaches to automatic differentiation. In M. Berz, C. Bischof, G.F. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 83–94. Soc. for Indust. and Appl. Math., Philadelphia, PA, 1996.

[5] A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Acad., Boston, MA, 1989.

[6] A. Griewank. The chain rule revisited in scientific computing. *SIAM News*, 24, 1991.

[7] M. Iri. Simultaneous computation of functions, partial derivatives and estimates of rounding errors: Complexity and practicality. *Japan J. Appl. Math.*, 1(2):223–252, 1984.

[8] J. Karczmarczuk. Functional differentiation of computer programs. In *Int. Conf. on Func. Prog. (ICFP '98)*, pages 195–203, January 1999.

[9] J. Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symb. Comp.*, 14(1):35–57, 2001.

[10] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT*, 16(1):146–160, 1976.

[11] C. Olah. Calculus on computational graphs: Backpropagation. `http://colah.github.io/posts/2015-08-Backprop/`, 2015.

[12] L.B. Rall. Differentiation and generation of Taylor coefficients in Pascal-SC. In U.W. Kulisch and W.L. Miranker, editors, *A New Approach to Scientific Computation*, pages 291–309. Academic Press, New York, NY, 1983.

[13] L.B. Rall. Differentiation in Pascal-SC: Type GRADIENT. *ACM Trans. Math. Softw.*, 10:161–184, 1984.

[14] L.B. Rall. The arithmetic of differentiation. *Mathematics Magazine*, 59:275–282, December 1986.

[15] L.B. Rall. Differentiation arithmetics. In C. Ullrich, editor, *Computer Arithmetic and Self-Validating Numerical Methods*, pages 73–90. Academic Press, New York, NY, 1990.

[16] L.B. Rall. Point and interval differentiation arithmetics. In A. Griewank and G.F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 17–24. Soc. for Indust. and Appl. Math., Philadelphia, PA, 1992.

[17] T.W. Reps and L.B. Rall. Computational divided differencing and divided-difference arithmetics. *Higher-Order and Symb. Comp.*, 16:93–149, 2003.

[18] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms.* PhD thesis, Dept. of Comp. Sci., Univ. of Illinois, Urbana, IL, January 1980.

[19] R. Zippel. Personal communication to Thomas Reps. July 1996.