Robust gradient descent via back-propagation: A Chainer-based tutorial

Matthew J. Holland*
Osaka University

Abstract

In this tutorial, we take the reader step-by-step through an implementation of so-called "robust gradient descent" algorithms for use in training neural network models via the technique of back-propagation. Concrete examples are provided throughout using Chainer, a user-friendly framework for working with neural networks, and we have made a complete implementation available at our online repository.[1]

Contents

*Please direct correspondence to `matthew-h@ar.sanken.osaka-u.ac.jp`.
[1]Jupyter notebook and source code at https://github.com/feedbackward/chainrob.

Notation

We use upper-case letters D, K, M, N, P, Q for integer quantities of interest, and and lower-case letters i, j, k, m, n, q, t for index subscripts. The lower-case letter l is reserved for loss functions. For any positive integer K, we denote the first K integers by $[K] \coloneqq \{1, \ldots, K\}$, and use I_K to denote the $K \times K$ identity matrix. For a differentiable function $f : \mathbb{R}^D \to \mathbb{R}$, we write $\nabla_{\boldsymbol{x}} f(\boldsymbol{u})$ for the vector of partial derivatives $\partial f / \partial x_j$ evaluated at $\boldsymbol{u}$ for each $j \in [D]$.

1 Algorithmic differentiation (AD)

In most technical disciplines, one frequently has the need to compute how "sensitive" a particular quantity of interest is to changes in another quantity. Treating the former (say y) as a smooth function of the latter (say x), written $y = f(x)$, as any student of elementary calculus is familiar, the derivative $f'(\cdot)$ is a fundamental tool when investigating such sensitivities. Adding a small δ to x, the difference is well-approximated by $f'(x)\delta \approx f(x+\delta) - f(x)$. Differentiation is the process of obtaining f', and of course subsequently evaluating it. Mathematically, this can be done by a straightforward, mechanical process. In practice, since we often deal with a tremendous number of variables of interest (both on the input and output side), often with very complicated functional relations, we use computers to expedite the differentiation process. Unlike idealized mathematical objects, computers have finite precision; the numerical accuracy of the output, as well as the computational costs involved with achieving a certain precision become important issues when considering which differentiation technique to use.

Here we consider one general strategy for implementing differentiation programs, typically called *algorithmic* (or *automatic*) *differentiation*, abbreviated AD. In this section, using a few simple examples, we introduce the basic approach taken by the techniques that fall under the heading of algorithmic differentiation. A comprehensive introduction to this topic is well beyond the scope of this tutorial. Interested readers are recommended to consult the lucid introduction of Griewank and Walther [1], from whom we borrow some useful notational conventions.

1.1 A concrete example to get us started

Instead of starting at a high level of abstraction and gradually making key concepts more precise, let's begin by diving straight into a simple example. Consider differentiation of the real-valued function f defined below:

$$f(x_1, x_2) = \frac{x_1^3}{x_2} + \sin(x_1 x_2).$$

To evaluate this function, one is given x_1 and x_2, and to compute $y = f(x_1, x_2)$ one does a series of intermediate computations. For the first summand, we must calculate x_1^3, and then divide this by x_2. For the second summand, first we multiply x_1 by x_2 before passing the product to $\sin(\cdot)$. Adding these two terms gives the desired result.

Following the notation of Griewank and Walther [1], these intermediate steps can be made explicit as follows in Table 1.
This is not the only order of operations by which one could implement the evaluation of this function, but it works. This list of variables and calculations is called an *evaluation trace* by Griewank and Walther [1]. A computational graph also provides a nice visual representation of the intermediate calculations (Figure 1).

u_{-1}	$=$	x_1	$=$	1.5000
u_0	$=$	x_2	$=$	3.0000
u_1	$=$	u_{-1}^3	$=$	3.3750
u_2	$=$	$u_{-1}u_0$	$=$	4.5000
u_3	$=$	u_1/u_0	$=$	1.1250
u_4	$=$	$\sin(u_2)$	$=$	-0.9775
u_5	$=$	$u_3 + u_4$	$=$	0.1475
y	$=$	u_5	$=$	0.1475

Table 1: Evaluating the example function by hand.

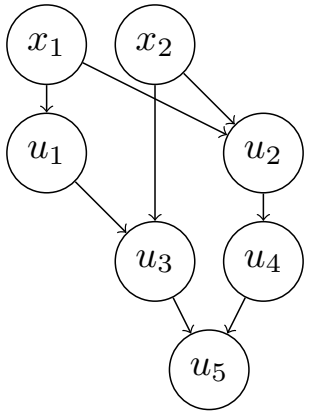

Figure 1: A computational graph for our evaluation of $f(x_1, x_2) = x_1^3/x_2 + \sin(x_1 x_2)$.

Forward mode Our interest here is with differentation. Let's consider computing the partial derivatives. Starting from input x_1, note that by taking the partial derivative of each intermediate variable (u_i with $i > 0$) with respect to $u_{-1} = x_1$, we can obtain $\partial y/\partial x_1$ via the chain rule, as follows. Write $\dot{u}_i = \partial u_i/\partial x_1$, and following the same sequence as the evaluation trace we just saw, we compute the partial derivatives in Table 2.

$\dot{u}_{-1}$	$=$	$\dot{x}_1$	$=$	1.0000
$\dot{u}_0$	$=$	$\dot{x}_2$	$=$	0.0000
$\dot{u}_1$	$=$	$3u_{-1}^2\dot{u}_{-1}$	$=$	6.7500
$\dot{u}_2$	$=$	$\dot{u}_{-1}u_0$	$=$	3.0000
$\dot{u}_3$	$=$	$\dot{u}_1\dot{u}_{-1}/u_0$	$=$	2.2500
$\dot{u}_4$	$=$	$\cos(u_2)\dot{u}_2\dot{u}_{-1}$	$=$	-0.6324
$\dot{u}_5$	$=$	$\dot{u}_3\dot{u}_{-1} + \dot{u}_4\dot{u}_{-1}$	$=$	1.6176
$\dot{y}$	$=$	$\dot{u}_5$	$=$	1.6176

Table 2: Evaluating the forward mode partial derivatives by hand.

Readers can check for themselves that $\dot{y} := \partial y/\partial x_1 = \dot{u}_5$ is actually mathematically valid; all we are doing is making repeated use of the chain rule. Most importantly, note that for any $i > 0$, as long as we have computed the intermediate function values (i.e., the $u_1, u_2, \ldots$) and the preceding variables $\dot{u}_{-1}, \dot{u}_0, \ldots, \dot{u}_{i-1}$, we always have enough information to compute $\dot{u}_i$. This approach of sequentially computing partial derivatives of intermediate variables with

respect to the input variables is called *forward mode* algorithmic differentiation.

Note that in each step computing the $\dot{u}_i$ values, compared with the corresponding step for u_i, there may be more operations (e.g., one extra multiplication in $\dot{u}_4$ compared with u_4), but this number will only be at most a small multiple of the original number of operations, and most importantly, the number of steps is the same. As such, computing $\dot{y}$ will be on the same order as computing y, namely doing function evaluation. Furthermore, the forward mode can be quite advantageous in cases where there many more outputs than inputs. In the above example, we only have one output variable y, but consider the generalized case where we have many outputs, say $\boldsymbol{y} = (y_1, \ldots, y_M)$ where each $y_j = a_j u_3 + b_j u_4$ for some scalars a_j and b_j, with $j = 1, \ldots, M$. To compute all of the $\dot{y}_1, \ldots, \dot{y}_M$ values requires simply computing

$$\dot{y}_j = a_j \dot{u}_3 \dot{u}_{-1} + b_j \dot{u}_4 \dot{u}_{-1}$$

for each $j = 1, \ldots, M$. That is, even if M is on the order of millions, as long as we compute the sequence up to $\dot{u}_4$ once, we can re-use these variables for all the outputs, making the difference between computing $\boldsymbol{y}$ itself and $\partial \boldsymbol{y}/\partial x_1$ minuscule. The exact same statements apply for differentiation with respect to the other input x_2.

Reverse mode In the "forward mode" approach described above, we fix an *input*, and consider the impact that this fixed input has on all the intermediate variables. In contrast to this, *reverse mode* algorithmic differentiation starts by fixing an *output* variable, and proceeds to compute the impact that each intermediate variable has on this fixed output. In the special case where our output is real-valued, i.e., $y \in \mathbb{R}$, then the "choice" is trivial. Using y to denote our output of interest, define $\bar{u}_i = \partial y/\partial u_i$, called the *adjoint variable*, for each step in the evaluation trace. The computation of partial derivatives in perhaps less intuitive than in the forward mode case, so let us take this step by step. First, we have

$$\begin{aligned} \bar{y} &= 1.0 \\ \bar{u}_5 &= \bar{y} \end{aligned}$$

as a seed to get the ball rolling. For $\bar{u}_4$, note that y only depends on u_4 via u_5, meaning that we can compute this as

$$\bar{u}_4 = \frac{\partial y}{\partial u_5}\frac{\partial u_5}{\partial u_4} = \bar{u}_5$$

via the chain rule and the fact that $\partial u_5/\partial u_4 = 1$. The exact same argument holds for $\bar{u}_3$, yielding

$$\bar{u}_3 = \bar{u}_5.$$

For $\bar{u}_2$ and $\bar{u}_1$, note that y only depends on these variables through u_4 and u_3 respectively, so in an analogous fashion we have

$$\begin{aligned} \bar{u}_2 &= \frac{\partial y}{\partial u_4}\frac{\partial u_4}{\partial u_2} = \bar{u}_4 \cos(u_2) \\ \bar{u}_1 &= \frac{\partial y}{\partial u_3}\frac{\partial u_3}{\partial u_1} = \bar{u}_3/u_0. \end{aligned}$$

Finally we reach the input variables. As y depends on u_0 only through u_3 and u_2, we have

$$\begin{aligned} \bar{u}_0 &= \frac{\partial y}{\partial u_3}\frac{\partial u_3}{\partial u_0} + \frac{\partial y}{\partial u_2}\frac{\partial u_2}{\partial u_0} \\ &= \bar{u}_3 \frac{(-1)u_1}{u_0^2} + \bar{u}_2 u_{-1}. \end{aligned}$$

In the same way, for u_{-1} which impacts y only through u_2 and u_1, we have

$$\begin{aligned}\bar{u}_{-1} &= \frac{\partial y}{\partial u_2}\frac{\partial u_2}{\partial u_{-1}} + \frac{\partial y}{\partial u_1}\frac{\partial u_1}{\partial u_{-1}} \\ &= \bar{u}_2 u_0 + \bar{u}_1 3u_{-1}^2.\end{aligned}$$

Summarizing these steps as before, we have partial derivatives as in Table 3.

$\bar{y}$	$=$	$\partial y/\partial y$	$=$	1.0000
$\bar{u}_5$	$=$	$\bar{y}$	$=$	1.0000
$\bar{u}_4$	$=$	$\bar{u}_5$	$=$	1.0000
$\bar{u}_3$	$=$	$\bar{u}_5$	$=$	1.0000
$\bar{u}_2$	$=$	$\bar{u}_4 \cos(u_2)$	$=$	-0.2108
$\bar{u}_1$	$=$	$\bar{u}_3/u_0$	$=$	0.3333
$\bar{u}_0$	$=$	$\bar{u}_3\frac{(-1)u_1}{u_0^2} + \bar{u}_2 u_{-1}$	$=$	-0.6912
$\bar{u}_{-1}$	$=$	$\bar{u}_2 u_0 + \bar{u}_1 3u_{-1}^2$	$=$	1.6174

Table 3: Evaluating the reverse mode partial derivatives by hand.

Just like in the forward case, we make use of the intermediate values $u_1, u_2, \ldots$, and we make use of recursive definitions. That is, starting from $\bar{u}_5$, as long as we have descended as far as $\bar{u}_{5-i}$, we always can compute one step further down, namely $\bar{u}_{5-(i+1)}$.

As seen in the forward mode approach, while we may have some extra computations at each step, the number of steps is the same as when evaluating the value of y itself. A distinct difference with the forward mode case is that running the reverse mode procedure above, we obtain *both* the partial derivatives, since $\bar{u}_0 = \partial y/\partial x_2$ and $\bar{u}_{-1} = \partial y/\partial x_1$ by definition. If there are multiple outputs, however, this procedure will need to be repeated for each. In contrast, the forward mode procedure only resulted in one partial derivative, namely $\dot{y} = \partial y/\partial x_1$, although as described, it can be very efficient when there are many more outputs than there are inputs. The strength of reverse mode algorithmic differentiation becomes salient in the case where we have many more inputs than we do outputs. For example, consider a simple modification of the function $f(x_1, x_2)$ above, where our inputs are denoted $x'_1, \ldots, x'_d, x'_{d+1}$ and the variables x_1 and x_2 become intermediate variables defined by

$$\begin{aligned}x_1 &= \prod_{j=1}^{M} x'_j \\ x_2 &= x'_{M+1}\end{aligned}$$

where we imagine that M is some very large integer. Note that even with large M, the computations from $\bar{u}_5$ down to $\bar{u}_1$ are identical with the original two-dimensional case above. The only new costs come in the final steps, namely

$$\begin{aligned}\frac{\partial y}{\partial x'_{M+1}} &= \frac{\partial y}{\partial u_0}\frac{\partial u_0}{\partial x'_{M+1}} = \bar{u}_0 \\ \frac{\partial y}{\partial x'_j} &= \frac{\partial y}{\partial u_{-1}}\frac{\partial u_{-1}}{\partial x'_j} = \bar{u}_{-1}\frac{u_{-1}}{x'_j}\end{aligned}$$

for each $j = 1, \ldots, M$. Writing $\boldsymbol{x}' = (x'_1, \ldots, x'_{M+1})$, note that the difference between computing y as a function of $\boldsymbol{x}'$ and computing the gradient $\partial y/\partial \boldsymbol{x}'$ is minuscule. Thus, when we

have real-valued functions (such as loss functions to minimize) depending on a large number of initial inputs, a reverse mode strategy is often the method of choice, as we shall see in later sections.

1.2 Layered composition and matrix multiplication

In the previous section, all the intermediate variables were scalar-valued. There are some cases, however, where it pays off to bundle up these scalars into a vector to take advantage of fast libraries for multiplying matrices, especially when some of these matrices contain many zero-valued entries.

As an example, assume we have an input $\boldsymbol{x} \in \mathbb{R}^{D_0}$ which is passed through K functions, say $f_k : \mathbb{R}^{D_{k-1}} \to \mathbb{R}^{D_k}$ for $k \in [K]$. Writing $\boldsymbol{y} = f(\boldsymbol{x})$, since the mapping takes the simple form $f = f_K \circ \cdots \circ f_1$, the intermediate variables can be simply written as

$$\begin{aligned} \boldsymbol{u}_0 &= \boldsymbol{x} \\ \boldsymbol{u}_1 &= f_1(\boldsymbol{u}_0) \\ &\vdots \\ \boldsymbol{u}_K &= f_K(\boldsymbol{u}_{K-1}) \\ \boldsymbol{y} &= \boldsymbol{u}_K. \end{aligned}$$

Based on this special structure, one can implement forward mode algorithmic differentiation in a straightforward way using a series of matrix multiplications. First, observe that using the chain rule,

$$\frac{\partial u_{k,j}}{\partial x_i} = \sum_{q=1}^{D_{k-1}} \frac{\partial u_{k,j}}{\partial u_{k-1,q}} \frac{\partial u_{k-1,q}}{\partial x_i}$$

for each $k \in [K]$, each $j \in [D_k]$, and each $i \in [D_0]$. Thus, writing $\dot{U}_k = \partial \boldsymbol{u}_k / \partial \boldsymbol{x}$ (shape $D_k \times D_0$) and defining another Jacobian matrix for the map $\boldsymbol{u}_{k-1} \mapsto \boldsymbol{u}_k$ as $J_k = \partial \boldsymbol{u}_k / \partial \boldsymbol{u}_{k-1}$ (shape $D_k \times d_{k-1}$), we obtain a convenient recursive relation as

$$\dot{U}_k = J_k \dot{U}_{k-1}.$$

Using $\dot{U}_0 = I_{D_0}$ as the base case sets the computation in motion for obtaining the desired $\partial \boldsymbol{y} / \partial \boldsymbol{x} = \dot{U}_K$.

The reverse mode approach can be done in a similar manner using the structure of this special case. Using the chain rule once again, we have

$$\frac{\partial y_i}{\partial u_{k,j}} = \sum_{q=1}^{D_{k+1}} \frac{\partial y_i}{\partial u_{k+1,q}} \frac{\partial u_{k+1,q}}{\partial u_{k,j}}$$

for each $k \in [K]$, $j \in [D_k]$, and $i \in [D_K]$. This means that writing $\bar{U}_k = \partial \boldsymbol{y} / \partial \boldsymbol{u}_k$ (shape $D_K \times D_k$), we can utilize the recursion

$$\bar{U}_k = J_{k+1} \bar{U}_{k+1}$$

for each $0 \leq k \leq K-1$. To start the computation, one uses $\bar{U}_K = I_{D_K}$ as a seed, descending to the desired $\partial \boldsymbol{y} / \partial \boldsymbol{x} = \bar{U}_0$.

Beware of naive vector bundling Recalling our original example of $f(x_1, x_2) = x_1^3/x_2 + \sin(x_1 x_2)$ from the previous section, one might be inclined to package up the intermediate scalar variables into vectors of variables that can be computed simultaneously. Considering the dependencies, we could re-write the evaluation trace as

$$\begin{aligned}
\boldsymbol{v}_0 &= (u_{-1}, u_0) \\
\boldsymbol{v}_1 &= (u_1, u_2) \\
\boldsymbol{v}_2 &= (u_3, u_4) \\
v_3 &= v_{2,1} + v_{2,2} \\
y &= v_3.
\end{aligned}$$

Indeed, it is clear that when trying to evaluate $y = f(x_1, x_2)$, evaluation in this order is perfectly legitimate, note that this does *not* have the layered structure of the previous $f = f_k \circ \cdots \circ f_1$ example, and a naive approach using matrix multiplication will lead to computational results that are mathematically incorrect. For example, note that the correct answer for say $\partial u_3/\partial x_2$ is of course

$$\frac{\partial u_3}{\partial x_2} = -\frac{u_1}{u_0^2} = -\frac{x_1^3}{x_2^2},$$

while a naive application of the recursion described above for the special layered composition case would lead to

$$\frac{\partial u_3}{\partial x_2} = \frac{\partial u_3}{\partial u_1}\frac{\partial u_1}{\partial x_2} + \frac{\partial u_3}{\partial u_2}\frac{\partial u_2}{\partial x_2} = 0 \neq -\frac{x_1^3}{x_2^2}$$

whenever $x_1 \neq 0$. This is because the "chain rule" used here is not actually valid, leading to a result which is computable, but not mathematically correct. We are simply stating the obvious here by saying that one must be careful that the chain rule be used *correctly*. The recursive relations defined above for the layered case are appealing, but cannot be used as-is in more general settings, as this simple example shows.

2 AD and machine learning

In the previous section, we gave a brief introduction to the core ideas underlying algorithmic differentiation. These techniques are, of course, extremely general-purpose, and can be used in any discipline where numerical sensitivities are of academic or practical interest. Our interest here, however, is one very specific application: learning algorithms for training machine learning models. Here we formulate a typical learning problem, and highlight the role that AD techniques can play.

2.1 A simple learning model

Consider a typical supervised learning task, where the goal is to reliably predict some response y given an input $\boldsymbol{x}$. One has a predictor $h(\boldsymbol{x}; \boldsymbol{w})$ parametrized by a vector $\boldsymbol{w} \in \mathbb{R}^D$, and access to a collection of data $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)$. Based on this data, the goal is to choose a good $\widehat{\boldsymbol{w}}$ such that the approximation $h(\boldsymbol{x}; \widehat{\boldsymbol{w}}) \approx y$ is "good enough." How do we measure this approximation, and determine what is good enough? It is typical to introduce a loss $l(\boldsymbol{w}; \boldsymbol{x}, y)$ which grows as the approximation of y made by $h(\boldsymbol{x}; \boldsymbol{w})$ worsens. The simplest example is the squared error $l(\boldsymbol{w}; \boldsymbol{x}, y) = (h(\boldsymbol{x}; \boldsymbol{w}) - y)^2$, but countless other ways of passing "feedback"

to the learning machine are possible. The canonical learning strategy is called empirical risk minimization (ERM), where one takes any $\widehat{\boldsymbol{w}}$ minimizing the sample mean of the losses, namely

$$\widehat{\boldsymbol{w}}_{\text{ERM}} \in \arg\min_{\boldsymbol{w}} \frac{1}{N} \sum_{i=1}^{N} l(\boldsymbol{w}; \boldsymbol{x}_i, y_i).$$

Of course, this minimization must be implemented in practice. Steepest descent methods are popular in the machine learning community, where one iteratively updates the parameter vector as

$$\boldsymbol{w}_{(t+1)} = \boldsymbol{w}_{(t)} - \frac{\alpha}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{w}} l(\boldsymbol{w}_{(t)}; \boldsymbol{x}_i, y_i),$$

where $\alpha > 0$ is a step-size parameter, and the update direction is the negative gradient of the ERM objective function. Linking this up to our AD introduction above, we have an objective function L depending on the data and our model parameters, written explicitly as

$$L(\boldsymbol{w}, \boldsymbol{x}_1, y_1, \ldots, \boldsymbol{x}_N, y_N) = \frac{1}{N} \sum_{i=1}^{N} l(\boldsymbol{w}; \boldsymbol{x}_i, y_i).$$

The steepest descent update given above has us taking partial derivatives with respect to some, but not all the independent variables of this function. The update direction at step t is $-\nabla_{\boldsymbol{w}} L(\boldsymbol{w}_{(t)}, y_1, \ldots, \boldsymbol{x}_N, y_N)$, a vector containing D partial derivatives, with the variables $\boldsymbol{x}_i$ and y_i for $i \in [N]$ left fixed. In the special case of a linear model $h(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{x}^T \boldsymbol{w}$, then computing the gradient is usually very easy. In the case of the squared error, then the loss gradients take the form $\nabla_{\boldsymbol{w}} l(\boldsymbol{w}; \boldsymbol{x}_i, y_i) = -2(y_i - \boldsymbol{w}^T \boldsymbol{x}_i)\boldsymbol{x}_i$, and thus manually implementing ERM by steepest descent is easy. As such, in this special case, the use of the AD techniques described above will not be necessary.

2.2 A more expressive model

In many situations, we will not be satisfied with a simple linear model. While countless non-linear models exist, as a well-known concrete exnample, here we formulate a traditional neural netwoork model, largely following the notation of Ripley [5]. This model is composed of computational units, which are fed real-valued outputs from other units, and which produce real-valued outputs themselves. More concretely, signals throughout the network are defined by

$$x_j = \sum_{i \to j} w_{ij} y_i$$
$$y_j = f_j(x_j)$$

where x_j and y_j are respectively the input and output of the jth unit. Summation "$i \to j$" means summing over all units i which feed to unit j. Each computational unit is completely specified by the activation function f_j (assumed differentiable) and weights w_{ij}. Let $\mathcal{V}$ denote the index of all units, and let e_{ij} be the indicator of {"unit i feeds unit j"}, and fix $w_{ij} = 0$ whenever $e_{ij} = 0$. To capture the inputs and outputs of the network as a whole, the network has both input units (which feed other units, but are fed by none) and output units (which feed no units, but are fed by other units).

The "layers" of a network can be captured as subsets $V_0, V_1, \ldots, V_K \subset \mathcal{V}$. Let V_0 denote the input layer, composed of only input nodes, i.e., for $j \in V_0$, $e_{ij} = 0$ for all $i \neq j$. Let V_K

denote the output layer, i.e., for $j \in V_K$, $e_{ji} = 0$ for all $i \neq j$. The inputs to the network as a whole constitute the inputs to the input layer, and the outputs of the network as a whole are the outputs of the output layer. To make things as clear as possible, we consider the special case of *feed-forward* neural networks, where units only feed their outputs to higher layers, i.e., for any k and all $j \in V_k$, we have $e_{ji} = 0$ whenever $i \in V_m$ for $m < k$. The *fully connected* feed-forward network is a special case in which for each k, all units in V_k feed to all units in V_{k+1}, and to no others. When a unit in V_k feeds a unit in layer V_{k+2} or higher, we call this a *skip-layer* connection. In what follows, we simply make the assumption that the connections are feed-forward in nature.

Training this network amounts to setting the weights w_{ij}, and using the ERM by steepest descent approach discussed above, we require partial derivatives of the objective L with respect to all of the w_{ij}. How should we go about computing these partial derivatives? Let us illustrate a natural strategy which has been re-discovered many times in the literature. First, note that for any weight w_{ij}, its impact on L is only through $x_j = \sum_{i \to j} w_{ij} y_i$. Thus, leaving all other variables fixed, we have that

$$\begin{aligned}\frac{\partial L}{\partial w_{ij}} &= \frac{\partial L}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} \\ &= \delta_j y_i\end{aligned}$$

where $\delta_j = \partial L / \partial x_j$ is for readability. Thus, if we know δ_j, and we have evaluated the function up to y_i, then we can compute the desired partial derivative with respect to w_{ij}. This holds for any of the weights. The challenge, of course, is obtaining δ_j. Noting that any unit input x_j only impacts L through y_j, namely after being passed through f_j. As such, one can write

$$\begin{aligned}\delta_j &= \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial x_j} \\ &= \frac{\partial L}{\partial y_j} f_j'(x_j).\end{aligned}$$

Since the f_j are something that we design, and x_j is computable, in order to obtain δ_j, it remains to obtain $\partial L / \partial y_j$.

Fortunately, obtaining the $\partial L / \partial y_j$ can be done in a systematic way. For units in the output layer ($j \in V_K$), then $\partial L / \partial y_j$ can be obtained directly based on knowledge of the loss $l(\boldsymbol{w}; \boldsymbol{x}, y)$ used in the definition of L. For other units, we can recover the quantity of interest in a recursive manner, as follows. Since any y_j can only impact L through the units to which it is fed, we can write

$$\begin{aligned}\frac{\partial L}{\partial y_j} &= \sum_{k:j\to k} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial y_j} \\ &= \sum_{k:j\to k} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial x_k}\frac{\partial x_k}{\partial y_j} \\ &= \sum_{k:j\to k} \delta_k w_{jk}\end{aligned}$$

where by "$k : j \to k$" we mean all nodes k that are fed by unit j. This means that for the intermediate units, we have

$$\delta_j = f_j'(x_j) \sum_{k:j\to k} \delta_k w_{jk}.$$

All that is required, then, is to "descend" the network structure, starting with the δ_j for units in $j \in V_K$, next tackling δ_j for all the units that feed the units in V_K, and so on.

The technique of re-using the partial derivatives taken with respect to inputs, namely "the deltas," was popularized as the *generalized delta rule* by Rumelhart and McClelland in the mid-1980s. See, for example, Rumelhart et al. [6, 7]. Using this approach, we can compute the gradient of the loss $\nabla_{\boldsymbol{w}} l(\boldsymbol{w}; \boldsymbol{x}_i, y_i)$ for any data point $i \in [n]$, and thus can readily implement ERM using steepest descent as introduced before. Since we start at the output layers, and work our way backwards, this technique for computing the gradients is often called *back-propagation*. Computing all the outputs y_j is typically called a *forward pass*, in contrast to a *backward pass* in which we compute the deltas and all quantities relevant to obtaining the partial derivatives.

Linking this up with AD Having seen a textbook example of back-propagation used in training a neural network, recall the simple example of section 1.1, and note that back-propagation can be readily interpreted as a special case of reverse mode AD. Observe the following correspondences:

- Function of interest: $f(x_1, x_2) \leftrightarrow L(\boldsymbol{w}; \boldsymbol{x}_1, y_1, \ldots, \boldsymbol{x}_N, y_N)$
- Input variables of interest: $(x_1, x_2) \leftrightarrow \boldsymbol{w}$.
- Intermediate variables: $u_1, \ldots, u_5 \leftrightarrow$ all unit inputs x_j.

With the above connections clear, note that the back-propagation differentiation strategy starts from the intermediate variables x_j closest to the final output value L, computing $\delta_j = \partial L / \partial x_j$ which will be re-used in a recursive procedure to eventually get to the bottom of the recursion. This is fundamentally the same as what we do in reverse mode AD, where we re-use partial derivatives with respect to the intermediate variables.

3 Using Chainer to expedite the AD process

Both in the generic setting of section 1, and the machine learning setting of section 2 where we looked at back-propagation, we saw via a handful of simple examples how the basic strategy of algorithmic differentiation works. While the core approach is clear, actually implementing such a procedure for any particular problem requires a substantial amount of effort. Especially in machine learning, where researchers want to quickly develop and test a variety of model/algorithm prototypes, spending too much time on computing gradients will stymie progress.

Fortunately, there exist high-quality open source libraries which have AD functionality out of the box. In this tutorial, we elect to use Chainer, a Python-based platform for implementing a wide variety of machine learning methods falling under the broad category of "neural networks." While alternatives such as TensorFlow, PyTorch, and Caffe are more well-known globally, Chainer is *extremely* easy to use, has a very elegant design, and the flexibility to handle all manner of models and learning algorithms included in other libraries. It also comes with support for multi-GPU hardware setups to take advantage of GPU-based acceleration. The core concept driving Chainer is a "Define-by-Run" strategy, in which the network is defined dynamically during forward pass computations.[2]

[2]Chainer documentation: https://docs.chainer.org/en/stable/.

```
import chainer.functions as cfn
import chainer as ch
import numpy as np

# Block (A): (function evaluation)
x1 = ch.Variable(np.array([1.5], dtype=np.float32))
x2 = ch.Variable(np.array([3.0], dtype=np.float32))
u1 = x1**3
u2 = x1*x2
u3 = u1/x2
u4 = cfn.sin(u2)
u5 = u3+u4

# Block (B): (reverse mode AD by hand)
ub5 = ch.Variable(np.array([1.0], dtype=np.float32))
ub4 = ub5
ub3 = ub5
ub2 = ub4*cfn.cos(u2)
ub1 = ub3/x2
xb2 = -ub3*u1/x2**2 + ub2*x1
xb1 = ub2*x2 + ub1*3*x1**2

# Block (C): (reverse mode AD using Chainer)
u5.backward(retain_grad=True)
```

Listing 1: Example code for a demonstration of Chainer's reverse mode AD faculty.

3.1 Implementing our first example

Recall our initial example from section 1.1, where we considered using AD to obtain partial derivatives of the function $f(x_1, x_2) = x_1^3/x_2 + \sin(x_1 x_2)$. Let's try using Chainer to implement this. First, let's do the function evaluation in a step-by-step manner, making the intermediate variables explicit (initialized with $x_1 = 1.5, x_2 = 3.0$), as shown in `Block (A)` of List 1. The resulting computational graph as generated by Chainer is given in Figure 2. The structure of the graph is exactly as we would expect.

Next, consider computation of partial derivatives. Following the reverse mode AD described in section 1.1, we can easily compute the $\bar{u}_i$ values using the code given in `Block (B)` of List 1. On the other hand, Chainer's AD faculty allows us to compute all of these partial gradients with *one line* of code, as shown in `Block (C)` of List 1.

As should be evident, the critical method attached to `Variable` objects is called `backward`, which recalling the terminology of section 2, does a "backward pass" over the function of interest, computing the partial derivatives with respect to all the intermediate variables, and storing them conveniently in the `grad` attribute of the various `Variable` objects defined above.[3] For example, to get $\bar{u}_2$, one need only inspect the contents of `u2.grad`. Printing out the results

[3]The option `retain_grad` asks Chainer to store the intermediate gradients. More often than not, all we'll need is `x1.grad` and `x2.grad`, in which case `retain_grad` should be `False` (the default) to save memory.

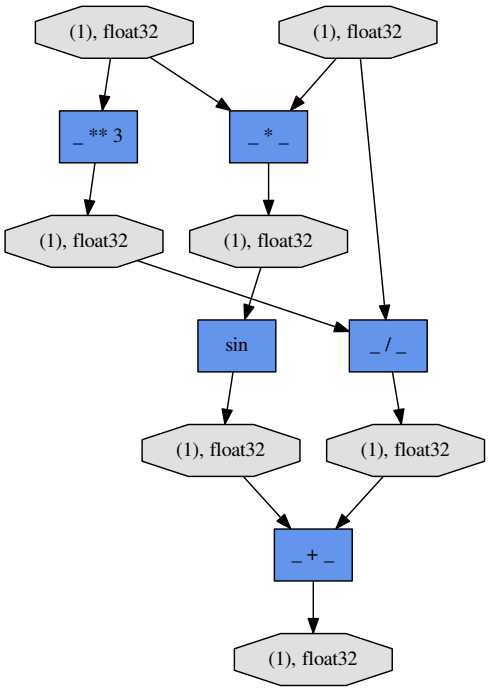

Figure 2: Computational graph generated automatically by Chainer. Compare with hand-drawn graph in Figure 1.

of our hand-implemented computations versus the Chainer-computed values, we have results as follows:

```
ub5 =  1.0000 vs. u5.grad =  1.0000
ub4 =  1.0000 vs. u4.grad =  1.0000
ub3 =  1.0000 vs. u3.grad =  1.0000
ub2 = -0.2108 vs. u2.grad = -0.2108
ub1 =  0.3333 vs. u1.grad =  0.3333
xb2 = -0.6912 vs. x2.grad = -0.6912
xb1 =  1.6176 vs. x1.grad =  1.6176
```

That is, the results are identical, strongly suggesting that Chainer is indeed computing the quantities we expect it is computing.[4] Using `Variable` objects is essentially the same as using NumPy arrays, except with the added functionality that a computational graph is managed in the background by Chainer. This becomes especially useful when it comes to obtain partial derivatives; all we have implemented is the forward pass computations, while the backward pass is handled on the backend. Even in an example as simple as this, instead of writing seven lines of code to get down to `xb1` and `xb2` (i.e., $\bar{u}_{-1}$ and $\bar{u}_0$), using Chainer it takes only *one* line. Even more importantly, the seven lines written in the hand-built case, while simple, still

[4]Note that there is a minor numerical difference between the final partial derivative and the one obtained in Table 3; the reason for this is due to rounding differences. In the computations for Table 3, we round to four digits for all computations, whereas the computations done here are done with full 32-bit precision, and the rounded intermediate results are displayed here.

require some mental effort to obtain in the first place; removing the burden of this task is critical for streamlining workflows involving differentiation.

3.2 Neural networks and aggregation

Let us now consider a Chainer-based implementation of a feed-forward neural network, as discussed in section 2. This will give us an opportunity to see some new features of Chainer, and identify the faculties that will particularly important when designing new algorithms.

As a simple example, let us consider the fully-connected case to start. For the kth layer, write $\boldsymbol{y}_k$ to denote a vector including the outputs of all the units $i \in V_k$. Write $D_k = |V_k|$ for the number of units in this layer. We can formulate the relationship between the kth layer and the layer before it as follows:

$$\boldsymbol{y}_k = f_k(W_k \boldsymbol{y}_{k-1}) = \begin{bmatrix} f_k\left(\langle \boldsymbol{w}_{k,1}, \boldsymbol{y}_{k-1} \rangle\right) \\ \vdots \\ f_k\left(\langle \boldsymbol{w}_{k,D_k}, \boldsymbol{y}_{k-1} \rangle\right) \end{bmatrix}.$$

Here W_k is a $D_k \times D_{k-1}$ matrix which controls the signals fed into the units of the kth layer. We have used $\boldsymbol{w}_{k,j}$ to denote the jth row of W_k, and when we write $f_k(W_k \boldsymbol{y}_{k-1})$, note that it is applied element-wise. We have matrix multiplication followed by an element-wise activation function, and this sort of operation is stacked to obtain $\boldsymbol{y}_0 \mapsto \boldsymbol{y}_1 \mapsto \cdots \mapsto \boldsymbol{y}_K$, the final network output. Thus, to implement a neural network, it is sufficient to implement these two operations.[5] In the remainder of this section, we look in detail at how to implement a $\boldsymbol{y}_{k-1} \mapsto W_k \boldsymbol{y}_{k-1}$; implementing f_k can be done following a perfectly analogous process.

Implementing computational nodes In our previous example, we just used simple operators such as `+`, `/`, `**`, and so on. Computation of the sensitivities of interest was done without any specification of the actual form of the intermediate partial derivatives (in contrast to the hand-built case). This was possible because Chainer handles the backward pass over these simple operations for us. What about *custom* functions? Handling matrix multiplication $\boldsymbol{y}_{k-1} \mapsto W_k \boldsymbol{y}_{k-1}$ is no problem for Chainer, but for example, the activation function f_k could certainly be something unique that is not included in the library. Fortunately, integrating customized operations with built-in operations is extremely easy, and the core role is played by `FunctionNode` objects.

A `FunctionNode` in Chainer is, quoting the documentation, a "node in the computational graph" which corresponds to "an application of a differentiable function to input variables." The basic flow is to pass an "input" in the form of a `Variable` object to a `FunctionNode` object, applying it with the method `apply()`. The first thing that applying this function does is, of course, computation of the output of the function based on the inputs given, i.e., the forward pass computations. In addition to this, it critically adds backward-reference edges on the computational graph between the input node, the function node, and the output node; this is critical for the backward pass over a series of computations, and is fortunately all handled on the back end.

To implement a `FunctionNode` child class is very easy: all we need to make explicit is the forward- and backward-pass computations. For the matrix multiplication operation of interest, we define a class `LinearFunction` which inherits `FunctionNode`. This is basically a simplified version of the same implementation in the Chainer source. Details are in the supplementary

```
def forward(self, inputs):
    if len(inputs) == 3:
        x, W, b = inputs
    else:
        (x, W), b = inputs, None
    y = x.dot(W.T)
    if b is not None:
        y += b
        self.retain_inputs((0,1,2))
    else:
        self.retain_inputs((0,1))
    return (y,)
```

Listing 2: Forward pass of a vanilla linear node.

demo notebook, but the important elements of the forward pass are given in List 2.

As should be clear, this is simply a generic map $\boldsymbol{y} = W\boldsymbol{x} + \boldsymbol{b}$, where W has dimensions $Q \times P$, and row vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_Q$. Here every variable involved in the function in included in the argument `inputs`. The `retain_inputs()` method indicates which indices of the inputs should be stored for use in the backward pass computations. The backward pass is implemented in a method called `backward`, with the key elements given in List 3.

Breaking down the backward pass There are a number of things that must be commented on and elucidated here. Let us take them one at a time. First, defining a backward pass operation in Chainer is straightforward because there are clear rules for what this function must compute, and the output it should produce. We describe some key points:

- The `indices` here corresponds to the tuple given to `retain_inputs()` in the forward pass definition.
- If the forward pass implements $\boldsymbol{y} = f(\boldsymbol{x})$, with $\boldsymbol{x} \in \mathbb{R}^P$ and $\boldsymbol{y} \in \mathbb{R}^Q$, then `backward()`, given `grad_outputs` (denoted $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_Q)$) *must* output $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_P)$, with

$$\lambda_i = \sum_{j=1}^{Q} \frac{\partial y_j}{\partial x_i} \gamma_j, \quad i \in [P].$$

- If there are multiple inputs, then the above partial gradient computations must be done for each, and stacked into a tuple to be returned.
- The shape of output $\boldsymbol{\lambda}$ must match the corresponding input. Thus if $\boldsymbol{x}$ is $P \times 1$, then $\boldsymbol{\lambda}$ must also be $P \times 1$.

The rules above are clear enough; let's make sure we understand exactly what is happening in this implementation, and how we should interpret the outputs. The `@` mark is a binary operator for matrix multiplication using `Variable` objects. The core computations are obviously being

[5] A useful introduction to defining custom functions in Chainer is available: https://docs.chainer.org/en/stable/guides/functions.html.

```
def backward(self, indices, grad_outputs):
    if len(self.get_retained_inputs()) > 2:
        x, W, b = self.get_retained_inputs()
    else:
        x, W = self.get_retained_inputs()
    gy = grad_outputs[0]
    out = []
    if 0 in indices:
        gx = gy @ W
        out.append(ch.functions.cast(gx, x.dtype))
    if 1 in indices:
        gW = gy.T @ x
        out.append(ch.functions.cast(gW, W.dtype))
    if 2 in indices:
        gb = ch.functions.sum(gy, axis=0)
        out.append(ch.functions.cast(gb, W.dtype))
    return tuple(out)
```

Listing 3: Backward pass of a vanilla linear node.

done directly after each of the "`if * in indices:`" lines. First is with respect to $\boldsymbol{x}$. Since partial derivatives are

$$\frac{\partial y_i}{\partial \boldsymbol{x}} = \boldsymbol{w}_i, \quad i \in [Q]$$

and the required output requires summing over index i with weights specified by $\boldsymbol{\gamma}$, then mathematically we should compute $\boldsymbol{\gamma}^T W$. This is precisely what is being computed in the line with `gy @ W`, noting that `gy` corresponds to $\boldsymbol{\gamma}$. Next, the case of differentiation with respect to W is essentially the same; we have

$$\frac{\partial y_i}{\partial W} = \begin{bmatrix} \mathbf{0} \\ \vdots \\ \boldsymbol{x} \\ \vdots \\ \mathbf{0} \end{bmatrix}$$

where all but the ith row are zero. Since we need to sum over the index i and multiply by the elements of $\boldsymbol{\lambda}$, mathematically the final output should be the $Q \times P$ matrix

$$\begin{bmatrix} \gamma_1 \boldsymbol{x} \\ \vdots \\ \gamma_Q \boldsymbol{x} \end{bmatrix}.$$

This is exactly what the line with `gy.T @ x` is computing. Finally, for differentiation with respect to $\boldsymbol{b}$, since we have

$$\frac{\partial y_i}{\partial \boldsymbol{b}} = (0, \ldots, 1, \ldots, 0)$$

where the ith element is 1 and the rest are non-zero. Summation over i yields a desired final output of $\boldsymbol{\lambda}$ itself. Note that in the code above, we have `sum(gy, axis=0)`. If `gy` (i.e., $\boldsymbol{\lambda}$) has shape $1 \times Q$, then since we've specified summation over `axis=0`, the function will properly return the desired λ as we expect; the reason for this summation will be discussed below.

Some readers may be wondering exactly what `gy` is, and what its shape is. Let us take each of these questions one at a time.

3.2.1 What is gy anyways?

First, regarding *what* `gy` is, note that `gy` is taken from the first element of `grad_outputs`, which is a tuple. What is `grad_outputs` supposed to contain? From the official documentation on the `backward()` method of the `FunctionNode` class, the content of `grad_outputs` is "gradients with respect to the output variables." What is being differentiated, and what are the output variables? From the perspective of just one lone node, it is just "some function," and the question of what is being differentiated does not matter because all computations involving that function are already assumed to be complete. Regarding what the output variables are, this refers to the output of *this node*. More explicitly, the content returned by `forward()`, which is in this case `(y,)`. Of course `y` is multi-variate, but since we return just one such object, this counts as one output variable. This is why `grad_outputs` has only one element in it (in our case above). Going back to the Chainer rules for the backward pass given above, things become more clear. The current node implements some $\boldsymbol{y} = f(\boldsymbol{x})$, and if F is "some function" of the outputs $\boldsymbol{y}$, then we may more naturally write $\gamma_j = \partial F / \partial y_j$, meaning that the output of the function node can be re-written

$$\begin{aligned} \lambda_i &= \sum_{j=1}^{Q} \frac{\partial F}{\partial y_j} \frac{\partial y_j}{x_i} \\ &= \frac{\partial F}{\partial x_j} \end{aligned}$$

since the inputs $\boldsymbol{x}$ only impact the function F through the outputs $\boldsymbol{y} = f(\boldsymbol{x})$, by definition. Recalling our algorithmic differentiation examples in sections 1 and 2, this should be a familiar sight: Chainer forces the `backward()` method of any function node to return the intermediate partial gradients (of some function) taken with respect to the inputs to the function node.

3.2.2 What is the shape of gy?

Next, we consider what the shape of `gy` is. First, recall that `gy` corresponds to $\boldsymbol{\gamma}$, the partial derivatives with respect to the node outputs. Following the simple rules highlighted above (summarized from the official documentation), the shape of `gy` must match the shape of `y`, which has Q columns, and one would infer from our exposition thus far that it would have 1 row. On the other hand, in our code, the summation of `gy` over `axis=0` seems to suggest that there may be situations in which `gy` has *more than one row*. What's going on here? What's going on here? The answer is very simple: the function has been designed to be able to handle *mini-batches*.[6] That is, the `x` here need not be $1 \times P$, but can rather be $N \times P$, housing a set of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ stacked along the first axis. Going back to the forward pass definition in `forward()`, since `x` will in general be treated as an $n \times d$ array, the output `y` will in correspondingly be $N \times k$ using the above code as-is. Thus instead of writing $\boldsymbol{y} = W\boldsymbol{x}$,

[6] This is described explicitly in, for example, the official documentation for `chainer.functions.linear`.

to describe the forward pass operation, it would clearer to write $Y = XW^T$, where W is as before, and X has shape $N \times P$.

With these points clear, the answer to the question regarding the size of `gy` is immediate: since the shape of `y` is $N \times Q$, and `gy` corresponds to the partial gradients with respect to `y`, naturally the shape of `gy` must also be $N \times Q$.

3.2.3 How to aggregate?

While the question of shape is clear, it raises a new and interesting question about how to *aggregate* the partial gradients over the batch. For each $i \in [N]$, one can consider the map $\boldsymbol{y}_i = W\boldsymbol{x}_i$, and obtain a partial derivative which depends on $\boldsymbol{x}_i$. Computing all of these, we will end up with N partial derivatives of shape $Q \times P$, one for each data point in the batch. However, the rules of `backward()` require that function return a partial derivative of the same shape as W, namely $Q \times P$. Thus, these N partial derivatives must be somehow aggregated into one final representative value. What aggregation method is being used in the above code? The simplest possible approach: *summation over the mini-batch*. This can be observed clearly in the example code above for both `gW` and `gb` computations. On the other hand, since `gx` will need to be of the shape $N \times P$, there is no need to do any aggregation there.

In virtually all libraries which include back-propagation functionality, the method of aggregation over mini-batches is by default set to summation, and in most cases the back-propagation sub-routines in fact *assume* this. This is of course natural, since summation works very nicely for the canonical machine learning setting we described in section 2, in which we use empirical risk minimization via steepest descent as our learning algorithm, where summation of the gradients over the data set is required.

On the other hand, Chainer is useful in that it gives the programmer the freedom to aggregate in any way he/she chooses. This means we can actually work with the individual gradients, giving us the freedom to, for example, identify outliers and either ignore them, or mitigate their impact. In contrast, naive summation gives equal weighting to all points, and thus when used in a learning algorithm to determine update direction, can easily lead the learner astray. Trying to combat this is the basic idea of recent "robust gradient descent" techniques studied in the literature [3, 2, 4]. In the remaining two sections, we introduce a standard robust gradient descent strategy, and show how it can be implemented for feed-forward neural networks to great effect when the data may be contaminated with outliers.

4 Robust gradient descent

4.1 The basic idea

In most modern machine learning problems, the usual formalism used for evaluating off-sample generalization is the risk minimization framework. The risk incurred by a particular candidate parameter is the expected loss, over the random draw of data from the true underlying distribution. Most learning problems can be reduced to selecting a parameter $\boldsymbol{w}$ (could be a vector, a set, a function, etc.) from a collection of candidates. The quality of any candidate is quantified in a pointwise fashion using a loss function $l(\boldsymbol{w}; \boldsymbol{z})$ for each candidate $\boldsymbol{w}$ and possible data value $\boldsymbol{z}$. The *risk* is defined $R(\boldsymbol{w}) = \mathbf{E}\, l(\boldsymbol{w}; \boldsymbol{z})$, where expectation is taken with respect to $\boldsymbol{z}$. We are given access to a sample $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N$, and l is known, but since the underlying distribution is unknown, the true objective $R(\cdot)$ will always be unknown. Based on this n-sized sample, the goal is to find a candidate $\widehat{\boldsymbol{w}}$ such that $R(\widehat{\boldsymbol{w}})$ is small enough, with large enough probability over the random draw of the sample.

If the risk were known, then the problem becomes an optimization problem, rather than a learning problem. For simplicity, assume the risk is differentiable, and that $\boldsymbol{w} \in \mathbb{R}^D$. With knowledge of the risk, one could iteratively tackle this problem using

$$\boldsymbol{w}^*_{(t+1)} = \boldsymbol{w}^*_{(t)} - \alpha\, \boldsymbol{g}(\boldsymbol{w}^*_{(t)}),$$

where $\boldsymbol{g}(\boldsymbol{w}) \coloneqq (\partial R(\boldsymbol{w})/\partial w_1, \ldots, \partial R(\boldsymbol{w})/\partial w_D)$. Of course, this is an idealized procedure, since we can never know the true data distribution, meaning R and thus $\boldsymbol{g}$ are always unknown. As such, in practice, we use the sample to approximate R and $\boldsymbol{g}$, subsequently feeding back this information which, albeit incomplete, is useful for minimizing the risk. One can approximate R based on location estimates using the loss values $l(\boldsymbol{w}; \boldsymbol{z}_1), \ldots, l(\boldsymbol{w}; \boldsymbol{z}_N)$, but if we want gradient information, then it may be more efficient to not bother with approximating R, and to just go straight for an approximation of $\boldsymbol{g}$. This can be done with the loss gradients $\nabla_{\boldsymbol{w}} l(\boldsymbol{w}; \boldsymbol{z}_1), \ldots, \nabla_{\boldsymbol{w}} l(\boldsymbol{w}; \boldsymbol{z}_N)$, based on which we construct an estimate $\widehat{\boldsymbol{g}} \approx \boldsymbol{g}$, which is then fed into an iterative update:

$$\widehat{\boldsymbol{w}}_{(t+1)} = \widehat{\boldsymbol{w}}_{(t)} - \alpha\, \widehat{\boldsymbol{g}}(\widehat{\boldsymbol{w}}_{(t)}),$$

As discussed in section 2, the standard approach is just to use the empirical mean of the risk gradient, namely to update using

$$\widehat{\boldsymbol{g}}(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{w}} l(\boldsymbol{w}; \boldsymbol{z}_i).$$

However, when the data may be heavy-tailed and susceptible to outliers, a few bad data points can lead the iterative update far away from the ideal path. As such, in recent years, interesting work has been done on robust gradient descent methods. One of the earliest proposals was made in a 2017 pre-print by the author [3], using a simple strategy of building per-coordinate M-estimators to construct a robust gradient estimate. That is, define

$$\widehat{\theta}_j(\boldsymbol{w}) = \operatorname*{arg\,min}_{\theta \in \mathbb{R}} \sum_{i=1}^{n} \rho\left(\frac{l'_j(\boldsymbol{w}; \boldsymbol{z}_i) - \theta}{s}\right), \quad j \in [D]$$

where $s > 0$ is a scaling parameter, $l'_j = \partial l/\partial w_j$, and ρ is a smooth, symmetric, convex function that is quadratic around its minimum (at zero), and linear in the $\pm\infty$ limit. The estimate is then just $\widehat{\boldsymbol{g}}(\boldsymbol{w}) = (\widehat{\theta}_1(\boldsymbol{w}), \ldots, \widehat{\theta}_D(\boldsymbol{w}))$. A fixed-point update can be used to compute these M-estimates quickly in an iterative fashion, for any given $\boldsymbol{w}$. A more recent proposal comes from Prasad et al. [4], who apply the "median of means" strategy to gradient descent. That is, partition the data set into K disjoint blocks as $[n] = \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_K$ and take the median of block means:

$$\widehat{\boldsymbol{g}}(\boldsymbol{w}) = \operatorname*{arg\,min}_{\boldsymbol{u}} \sum_{k=1}^{K} \|\boldsymbol{u} - \widetilde{\boldsymbol{g}}_k(\boldsymbol{w})\|$$
$$\widetilde{\boldsymbol{g}}(\boldsymbol{w}) = \frac{1}{|\mathcal{I}_k|} \sum_{i \in \mathcal{I}_k} \nabla_{\boldsymbol{w}} l(\boldsymbol{w}; \boldsymbol{z}_i).$$

Here the high-dimensional median is the so-called geometric median, minimizing the sum of the distances, a quantity which can be computed accurately using a fast iterative procedure.

4.2 An application to neural networks

Let us consider how to implement a robust gradient descent strategy in the case of neural network models. In the end, we are interested in the sensitivity of the losses $l(\boldsymbol{w}; \boldsymbol{z})$ to the weights $\boldsymbol{w}$, and the dependence of the loss on the weights is only through a finite number of intermediate computations. Writing the output of these operations as $u_1, \ldots, u_m \in \mathbb{R}$, for any individual weight w, we have that

$$\frac{\partial l(\boldsymbol{w}; \boldsymbol{z})}{\partial w} = \sum_{m=1}^{M} \frac{\partial l(\boldsymbol{w}; \boldsymbol{z})}{\partial u_m} \frac{\partial u_m}{\partial w}.$$

This partial gradient can be obtained for every data point $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$, and assuming that these are computed individually, the partial gradients can be fed into a sub-routine for robust location estimation. As a general-purpose procedure, still assuming $\boldsymbol{w} \in \mathbb{R}^D$, consider the following: at each step t in the iterative routine, select weight indices $\mathcal{I}_{(t)} \subseteq [D]$ to be estimated in a robust fashion (allows for $\mathcal{I}_{(t)} = \emptyset$). Making the update more explicit, at each step we update as

$$\widehat{\boldsymbol{w}}_{(t+1)} = \widehat{\boldsymbol{w}}_{(t)} - \alpha \, \textsc{RobGrad}\left(\widehat{\boldsymbol{w}}_{(t)}, \mathcal{I}_{(t)}\right), \tag{1}$$

where the details of RobGrad are specified in Algorithms 1. For all the specified weights, the partial derivatives are passed to a sub-routine called ROBUSTIFY. While any number of robust estimation procedures could be used here, for concreteness and clarity of this tutorial, we provide an example where we carry out soft truncation (after standardization) of the observations. We use the function

$$\psi(u) := \begin{cases} u - u^3/6, & -\sqrt{2} \leq u \leq \sqrt{2} \\ 2\sqrt{2}/3, & u > \sqrt{2} \\ -2\sqrt{2}/3, & u < -\sqrt{2} \end{cases} \tag{2}$$

for truncation, and set the scaling parameter $s > 0$ based on optimizing concentration inequalities (see Holland [2] for details), where parameter δ can be safely fixed at say $\delta = 0.005$.

Example: layer-wise robustification It remains to actually implement this procedure. While the details will differ slightly depending on the nature of the model being used, here we provide an illustrative example for fully-connected feed-forward neural networks. Recall that in section 3.2, we illustrated how the Chainer development team implemented the backward pass of a fully-connected layer in a typical neural network, i.e., the operation $\boldsymbol{x} \mapsto W\boldsymbol{x} + \boldsymbol{b}$, where we are interested in partial derivatives with respect to W and $\boldsymbol{b}$. We also mentioned how the core `FunctionNode` object was designed such that per-observation gradients are aggregated via summation, but commented on the fact that this is not the only way to do it. Since the robust gradient descent strategies highlighted above require a different method of aggregation, here we show how modifying a few lines lets us easily robustify this node. The forward-pass is identical to List 2 given in section 3.2, and thus we only show the backward pass here in List 4.

The basic structure is very similar to the vanilla linear layer; all one needs to do is replace the operations which sum over the data with a call to `self.robustifier` that takes the per-point gradients, and outputs a robust vector estimate of the proper form (this corresponds to Algorithm 2). Note that with the implementation given in List 4, we are robustifying entire layers at once. When constructing the network, simply specify which layers are to be robustified, and replace the default function node (as seen in our earlier example, Lists 2 and 3) with one that has a built-in robustification sub-routine, as in List 4.

Algorithm 1 RobGrad

inputs: candidate $\boldsymbol{w}$, weight index $\mathcal{I}$

for $j = 1, \ldots, D$ **do**

$$g_{j,i} = \sum_{m=1}^{M} \frac{\partial l(\boldsymbol{w}; \boldsymbol{z}_i)}{\partial u_m} \frac{\partial u_m}{\partial w_j}, \quad i \in [N].$$

if $j \in \mathcal{I}$ **then**

$$\widehat{g}_j = \textsc{Robustify}\left(\{g_{j,i}\}_{i=1}^{N}\right)$$

else

$$\widehat{g}_j = \frac{1}{N} \sum_{i=1}^{N} g_{j,i}$$

end if

end for

return: $(\widehat{g}_1, \ldots, \widehat{g}_D)$

Algorithm 2 Robustify: Robust mean estimation sub-routine

inputs: data $x_1, \ldots, x_n$, confidence $0 < \delta < 1$

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i, \quad \sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2, \quad s^2 = \frac{n}{2 \log(\delta^{-1})}$$

$$\widehat{x} = \frac{s}{n} \sum_{i=1}^{n} \psi\left(\frac{\widetilde{x}_i}{s}\right), \quad \text{where } \widetilde{x}_i = \frac{x_i - \bar{x}}{\sigma} \text{ for } i \in [n].$$

return: $\sigma \widehat{x} + \bar{x}$

5 A complete demo: Iris data with noisy inputs

We begin with one of the most popular and simple pattern recognition tasks, and make a few modifications as described below. We start with the Iris data set of R.A. Fisher.[7] This data (originally four features) is projected to a plane using the principal components obtained from the training data. This data is then normalized to the unit square. Finally, a small fraction of the data, selected randomly, is perturbed by the following procedure. Let $\bar{\boldsymbol{x}}$ be the empirical mean of the training set $\{\boldsymbol{x}_i\}_{i=1}^n$. Let $\sigma^2 = \mathrm{var}\{\|\boldsymbol{x}_i - \bar{\boldsymbol{x}}\|\}_{i=1}^n$. Let k be a multiplicative factor we set freely. Write $r := k\sigma / \|\boldsymbol{x} - \bar{\boldsymbol{x}}\|$. The perturbation from $\boldsymbol{x} \mapsto \widetilde{\boldsymbol{x}}$ is

$$\widetilde{\boldsymbol{x}} = (1 + r)\bar{\boldsymbol{x}} - r\boldsymbol{x}.$$

Geometrically, consider a circle centered at $\bar{\boldsymbol{x}}$ with radius $k\sigma$, and the line upon which both $\boldsymbol{x}$ and $\bar{\boldsymbol{x}}$ rest. Of the two points where this line intersects the circle, $\widetilde{\boldsymbol{x}}$ corresponds to the point farthest from $\boldsymbol{x}$. See Figure 3 for an illustrative example.

For each trial, from the 150 points in the full data set, we randomly select $n = 100$ points for training, with the remainder used as a test set. The input noise procedure above is applied to a randomly selected subset of the training data, of size equal to 2% of sample size n. The variance factor is set to $k = 10^4$ to simulate an arbitrarily large perturbation to a tiny fraction of points.

[7] https://archive.ics.uci.edu/ml/datasets/iris

```
def backward(self, indices, grad_outputs):
    if len(self.get_retained_inputs()) > 2:
        x, W, b = self.get_retained_inputs()
    else:
        x, W = self.get_retained_inputs()
    gy = grad_outputs[0]
    k, d = W.shape
    out = []
    if 0 in indices:
        gx = gy @ W
        out.append(ch.functions.cast(gx, x.dtype))
    if 1 in indices:
        for i in range(k): # Loop over output dimension.
            gradsW = x.array * np.take(gy.array, [i], 1)
            gW[i,:] = self.robustifier(x=gradsW)
        gW = ch.Variable(gW)
        out.append(ch.functions.cast(gW, W.dtype))
    if 2 in indices:
        gb = np.zeros((k,), dtype=W.dtype) # start as ndarray.
        gb = self.robustifier(x=gy.array).flatten()
        gb = ch.Variable(gb)
        out.append(ch.functions.cast(gb, W.dtype))
    return tuple(out)
```

Listing 4: Backward pass of robustified linear node. Compare with List 3.

Regarding the methods used, first we describe the model. As a typical model, we consider a feed-forward neural network with fully-connected layers. There are two hidden layers, each with 10 units. The output layer has no activation function (i.e., the signals are passed as-is), and all intermediate layers are rectified using $f(u) = \max\{u, 0\}$. In our demonstration, we compare two algorithms: vanilla gradient descent using the empirical mean of the loss gradients (denoted `deep`), and robust gradient descent using the procedure described in section 4.2 (denoted `deep-rob`), where only the weights contributing to the *first* hidden layer are estimated in a robust fashion. Both algorithms have a fixed step size of $\alpha = 0.5$, and are run for 1000 iterations.

Our demonstration here runs 20 independent trials, computing average performance over the trials for each iteration in the algorithm updates, counting the cost in gradients computed. The training and test performance of these two competing algorithms are given in Figure 4. While the robustified version takes a bit longer to kick in, the negative impact of the input noise is clearly mitigated using the robustified procedure, as we would expect.

Breakdown of demonstration code

- `demo.ipynb`: includes the source code for running the above experiments (and executing update (1) above), as well as creating the final figures.
- `config.py`: configuration file, a few non-crucial constants are set here.

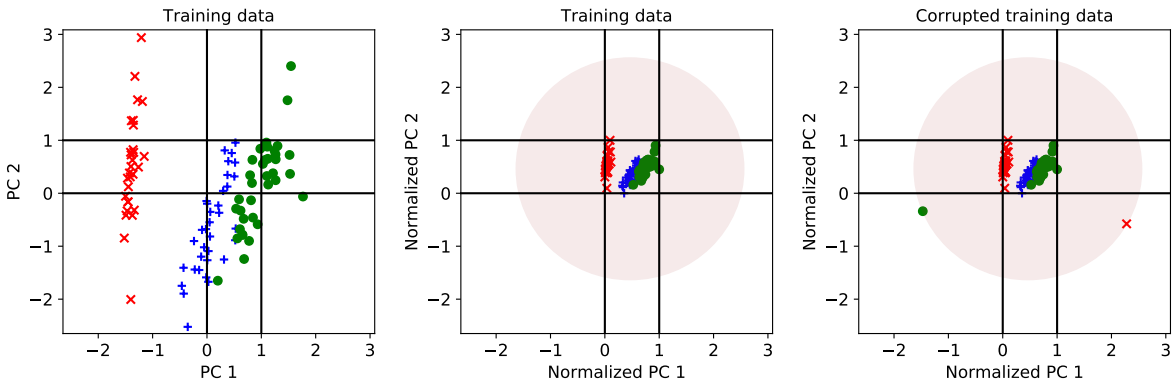

Figure 3: Illustation of the two-dimensional Iris data set with input noise set using factor $k = 10$. Left: data after projection. Center: projected data after normalization, with the ball of radius $k\sigma$ centered at the sample mean. Right: data after random perturbation.

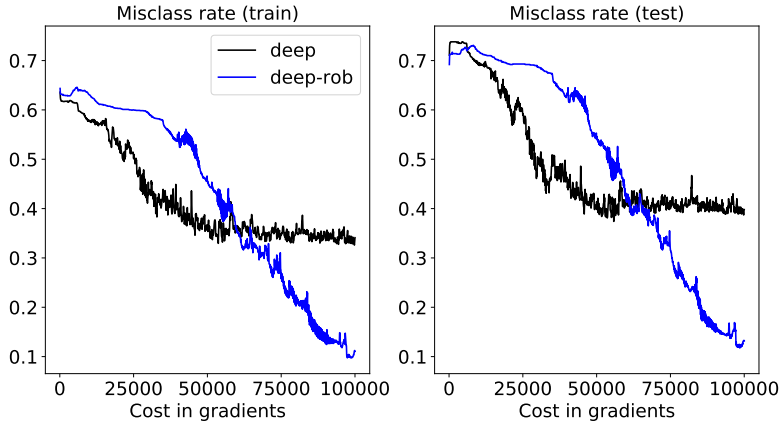

Figure 4: Results of classification under noisy inputs, averaged over 20 trials. Left: training data. Right: test data.

- `get_model.py`: definition of the model/algorithm pair used by the two competing methods of interest.
- `helpers.py`: some helper functions, including the ψ function used for soft truncation.
- `models.py`: key definitions of neural network models, starting from `FunctionNode` class definitions and creating subsequent `Link` and `Chain` objects.
- `robustify.py`: implements the sub-routine contained in Algorithm 2.

References

[1] Griewank, A. and Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2nd edition.

[2] Holland, M. J. (2019). Robust descent using smoothed multiplicative noise. In *Proceedings of Machine Learning Research (AISTATS2019, to appear)*, Okinawa, Japan.

[3] Holland, M. J. and Ikeda, K. (2017). Efficient learning with robust gradient descent. *arXiv preprint arXiv:1706.00182*.

[4] Prasad, A., Suggala, A. S., Balakrishnan, S., and Ravikumar, P. (2018). Robust estimation via robust gradient estimation. *arXiv preprint arXiv:1802.06485*.

[5] Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

[6] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

[7] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1987). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, chapter 8. MIT Press.