First-Order Automatic Differentiation in JAX

An In-Depth Tutorial

Zhihan Yang

Department of Computer Science
Cornell University
Ithaca, NY 14853
zhihany@cornell.edu

Abstract

JAX is a high-performance numerical-computing library in Python recently developed by Google. With a syntax similar to Numpy, it also features just-in-time compilation, automatic differentiation (AD), and hardware acceleration via XLA[1]. This tutorial explores two critical aspects of JAX's AD system for computing first-order derivatives[2].

Generally, we want use AD to find the derivative of each output scalar variable[3] with respect to each scalar input variable of a large acyclic computational graph composed of simpler functions with their own inputs and outputs. Most existing tutorials on AD assume that both the computation graph and its consisting functions take in one scalar/vector and outputs one scalar/vector. While this is fine for pedagogy, it is not realistic since, in practice, functions can take in and output a list[4] of tensors. Indeed, the programming abstractions of JAX's AD system were developed to compute derivatives of functions of such generality, and can hence be cryptic for those who only understand AD for scalar-to-scalar and vector-to-vector functions.

We first explain how deratives of such a computation graph can be computed via forward-mode and reverse-mode AD; important concepts such as Jaocbians, Jacobian-vector products and vector-Jacobian products are discussed in great detail. Cmoputationa graph input and output. Each function input and output. We then explain how users can supply custom AD rules to differentiate through functions that, e.g., does not include JAX code. Finally, we derive from scratch AD rules for common functions.

JAX is sophisticated codebase and we will not go to the source code. Rather, it gives the reader a mental model of how common JAX's functions work under the hood and relate to other. Prepare the fully unlease the potential JAX for machine learning research.

Contrary to machine learning, which usually just requires VJP

Table of contents

[1] According to Google video, this is how JAX got its name

[2] Also called gradients. We do not include hessian computation this in tutorial.

[3] These scalar variables may be organized in vectors, matrices or tensors.

[4] Tree is an interesting thing

1 Multivariate chain rule

The most important theorem for understanding automatic differentiation is the multivariate chain rule. In this section, we present three versions of this rule with increasing generality. In a college-level multivariate calculus class, one would learn the following version of this rule: if four scalar variables x, y, z and t follow the relationship

$$t \mapsto x \quad t \mapsto y \quad (x, y) \mapsto z,$$

then the derivative of z with respect to t can be calculated as

$$\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt}. \tag{1}$$

Its proof can be found in any standard calculus textbook.

First generalization. We can generalize this version a bit more. If vectors $\vec{x} \in \mathbb{R}^M, \vec{y} \in \mathbb{R}^N, \vec{z} \in \mathbb{R}^O$ follow the relationship

$$\vec{x} \mapsto \vec{y} \mapsto \vec{z},$$

then the derivative of $\vec{z}$ with respect to $\vec{x}$ can be calculated as the following matrix multiplication

$$\underbrace{\frac{d\vec{z}}{d\vec{x}}}_{(O,M)} = \underbrace{\frac{d\vec{z}}{d\vec{y}}}_{(O,N)} \underbrace{\frac{d\vec{y}}{d\vec{x}}}_{(N,M)}, \tag{2}$$

where the three matrices above from the left to right are the Jacobian of $\vec{z}$ with respect to $\vec{x}$, the Jacobian of $\vec{z}$ with respect to $\vec{y}$, and the Jacobian of $\vec{y}$ with respect to $\vec{x}$:

$$\begin{pmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_O}{\partial x_1} & \cdots & \frac{\partial z_O}{\partial x_M} \end{pmatrix} = \begin{pmatrix} \frac{\partial z_1}{\partial y_1} & \cdots & \frac{\partial z_1}{\partial y_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_O}{\partial y_1} & \cdots & \frac{\partial z_O}{\partial y_N} \end{pmatrix} \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_N}{\partial x_1} & \cdots & \frac{\partial y_N}{\partial x_M} \end{pmatrix}.$$

To verify Equation 2, one may start with the definition of matrix multiplication:

$$\left(\frac{d\vec{z}}{d\vec{x}}\right)_{i,j} = \sum_{k=1}^{N} \left(\frac{d\vec{z}}{d\vec{y}}\right)_{i,k} \left(\frac{\partial \vec{y}}{\partial \vec{x}}\right)_{k,j}.$$

But the left-hand side and right-hand side are, respectively, just

$$\frac{\partial z_i}{\partial x_j} = \sum_{k=1}^{N} \frac{\partial z_i}{\partial y_k} \frac{\partial y_k}{\partial x_j},$$

which is in essense no different from Equation 1! Also, note how this is the dot product of the i-th row of $d\vec{z}/d\vec{y}$ and the j-th column of $d\vec{y}/d\vec{x}$.

Second generalization. We can generalize the multivariate chain rule even further. Suppose (real) tensors, or multidimensional arrays, X, Y, Z follow the following relationship

$$X \to Y \to Z.$$

This is the most general version, since tensors include scalars and vectors. Since these tensors can each have an arbitrary number of dimensions, we will resort to an example here, where we assume that $X \in \mathbb{R}^{A \times B \times C}, Y \in \mathbb{R}^{D \times E}, Z \in \mathbb{R}^{F \times G \times H \times I}$. The derivative of Z with respect to X can be written as

$$\underbrace{\frac{dZ}{dX}}_{((F,G,H,I),(A,B,C))} = \underbrace{\frac{dZ}{dY}}_{((F,G,H,I),(D,E))} : \underbrace{\frac{dY}{dX}}_{((D,E),(A,B,C))} \tag{3}$$

where the three tensors are defined as

$$\begin{aligned} \left(\frac{dZ}{dX}\right)_{(f,g,h,i),(a,b,c)} &= \frac{\partial Z_{f,g,h,i}}{\partial X_{a,b,c}} \\ \left(\frac{dZ}{dY}\right)_{(f,g,h,i),(d,e)} &= \frac{\partial Z_{f,g,h,i}}{\partial Y_{d,e}} \\ \left(\frac{dY}{dX}\right)_{(d,e),(a,b,c)} &= \frac{\partial Y_{d,e}}{\partial X_{a,b,c}} \end{aligned}$$

and the ":" operator denotes *tensor contraction*, which is defined as

$$\left(\frac{dZ}{dX}\right)_{(f,g,h,i),(a,b,c)} = \underbrace{\left(\frac{dZ}{dY}\right)_{(f,g,h,i),(:,:)}}_{(D,E)} \cdot \underbrace{\left(\frac{dY}{dX}\right)_{(:,:),(a,b,c)}}_{(D,E)} = \frac{dZ_{f,g,h,i}}{dY} \cdot \frac{dY}{dX_{a,b,c}}.$$

The "$\cdot$" operator denotes the *tensor dot product*, which multiplies two tensors element-wise and sum up all resulting products into a single scalar. Unfortunately, tensors are high-dimensional so we can't write them out as in the vector case. But recall that the dot product was important in the vector case, too: $\partial z_i / \partial x_j$ is the dot product of the i-th row of $d\vec{z}/d\vec{y}$ and the j-th column of $d\vec{y}/d\vec{x}$.

Remark 1. Equation 2 and 3 are the same as Equation 1 except that, for Equation 2 and 3, the variables are organized in more complicated structures. Therefore, one has to rely on matrix multiplication and, more generally, tensor contraction, to express the multivariate chain rule involving these complicated structures.

$$f \;=\; ma \tag{4}$$

2 A main goal of first-order automatic differentiation

Consider a directed acyclic computation graph that takes in N tensors and spits out M tensors, with no restriction on the shape of each tensor; these tensors may be organized in pytrees[5]. This is, of course, a very general setup. We view this graph as a function and denote it by g:

$$g(X^{(1)}, \ldots, X^{(N)}) = (Y^{(1)}, \ldots, Y^{(M)}).$$

One main[6] goal of first-order AD is to obtain the following matrix of tensors:

$$\text{“Jacobian”} = \begin{pmatrix} \frac{\partial Y^{(1)}}{\partial X^{(1)}} & \cdots & \frac{\partial Y^{(1)}}{\partial X^{(N)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial Y^{(M)}}{\partial X^{(1)}} & \cdots & \frac{\partial Y^{(M)}}{\partial X^{(N)}} \end{pmatrix},$$

where (i,j)-th entry of this matrix, $\partial Y^{(i)}/\partial X^{(j)}$, is a tensor whose shape is the concatenation of the shapes of $Y^{(i)}$ and $X^{(j)}$. More specifically, if $Y^{(i)} \in \mathbb{R}^3$ and $X^{(j)} \in \mathbb{R}^2$, then $\partial Y^{(i)}/\partial X^{(j)}$ would be in 5 dimensions with[7]

$$\left(\frac{\partial Y^{(i)}}{\partial X^{(j)}}\right)_{abcde} = \frac{\partial Y^{(i)}_{abc}}{\partial X^{(j)}_{de}}.$$

We have put quotation marks around the word Jacobian because this matrix above does indeed contain all the required partial derivatives but is not the standard Jacobian of vector-to-vector functions. Whenever we use "Jacobian" later on, we are referring to this very matrix above.

Since each input and output tensor could have a different shape, the "Jacobian" is best represented as a nested tuple (i.e., a tuple of tuples) of matrices in Python. Below, let's try out `jax.jacfwd` and `jax.jacrev`, two JAX functions that compute the "Jacobian", to see if this is indeed what they do. `jax.jacfwd` and `jax.jacrev` computes the "Jacobian" via *forward-mode* (Section 3) and *reverse-mode* AD (Section 4) respectively. These two modes will be discussed at length later.

[5] JAX uses the term "pytree" to refer to "a tree-like structure built out of container-like Python objects". For example, if `A` and `B` are two JAX tensors, then `[A, {"data": B}]` would be a pytree. Pytrees have very flexible structures and, for the sake of conciseness, we will leave them out in the remainder of this tutorial.

[6] By "main", we mean that any AD system should be able to compute the "Jacobian" efficiently one way or another.

[7] We sometimes group the indices to make it more explicit where each group comes from; the following two notations are equivalent in this tutorial:

$$\left(\frac{\partial Y^{(i)}}{\partial X^{(j)}}\right)_{abcde} = \left(\frac{\partial Y^{(i)}}{\partial X^{(j)}}\right)_{(abc)(de)}.$$

```
def g(X1, X2):
    Y1 = X1 @ X2
    Y2 = X * X
    return Y1, Y2

key = jax.random.key(42)
X1_key, X2_key = jax.random.split(key, 2)
X1 = jax.random.normal(X1_key, shape=(2, 3))
X2 = jax.random.normal(X2_key, shape=(3, 4))

Y1, Y2 = g(X1, X2)
print(Y1.shape, Y2.shape)  # (2, 4) (2, 3)

jac_fwd = jax.jacfwd(g, argnums=(0, 1))(X1, X2)
# first row of the nested tuple
print(jac_fwd[0][0].shape, jac_fwd[0][1].shape)  # (2, 4, 2, 3) (2, 4, 3, 4)
# second row of the nested tuple
print(jac_fwd[1][0].shape, jac_fwd[1][1].shape)  # (2, 3, 2, 3) (2, 3, 3, 4)

jac_rev = jax.jacrev(g, argnums=(0, 1))(X1, X2)
# first row of the nested tuple
print(jac_rev[0][0].shape, jac_rev[0][1].shape)  # (2, 4, 2, 3), (2, 4, 3, 4)
# second row of the nested tuple
print(jac_rev[1][0].shape, jac_rev[1][1].shape)  # (2, 3, 2, 3), (2, 3, 3, 4)
```

Before we start later sections, there are two important things to note. Firstly, in practice, one would define the function g using some simpler functions. In this tutorial, we use f denote a function "within" the function g that takes in N_f tensors and outputs M_f tensors

$$f(X^{(f,1)},\dots,X^{(f,N_f)}) = (Y^{(f,1)},\dots,Y^{(f,M_f)}).$$

Secondly, the outputs of g given its inputs can be computed by running Algorithm 1[8]:

```
Algorithm 1
Forward pass through a computation graph

Input: computation graph, $(X^{(1)},\dots,X^{(N)})$

For $f$ in functions inside the computational graph (in a forward fashion):

    $(Y^{(f,1)},\dots,Y^{(f,M_f)}) \leftarrow f(X^{(f,1)},\dots,X^{(f,N_f)})$

Output: $(Y^{(1)},\dots,Y^{(M)})$
```

3 Forward-mode automatic differentiation

In this section, we discuss forward-mode AD for computing the "Jacobian" and a "Jacobian-vector product" ("JVP") of a computation graph.

3.1 Computing the "Jacobian"

If we apply Equation 4 to some f inside the computation graph g, we see that

$$\frac{\partial Y^{(f,i)}}{\partial X^{(j)}} = \sum_{k=1}^{N_f} \frac{\partial Y^{(f,i)}}{\partial X^{(f,k)}} : \frac{\partial X^{(f,k)}}{\partial X^{(j)}}. \tag{5}$$

[8] I must admit that the idea of looping through functions is a bit handwavy; the idea I'm trying to convey is just that we want to sequence the execution of these functions in such a way that the inputs they depend on have already been computed.

Assuming that f's own "Jacobian" is known, this relationship shows that, if we want to compute the partial derivatives of f's output variables[9] with respect to g's input variables, we must first know the partial derivatives of f's input variables with respect to g's input variables. As a result, it inspires a "forward-style" algorithm (i.e., an algorithm that first goes through variables closer to g's input variables) for computing the partial derivatives of g's output variables with respect to g's input variables:

```
Algorithm 2
Forward-mode automatic differentiaton for computing the "Jacobian"

Input: computation graph, $(X^{(1)}, \dots, X^{(N)})$
Initialize $\left(\left(\frac{\partial X^{(1)}}{\partial X^{(1)}} \leftarrow I, \frac{\partial X^{(2)}}{\partial X^{(1)}} \leftarrow 0 \dots, \frac{\partial X^{(1)}}{\partial X^{(N)}} \leftarrow 0\right), \dots, \left(\frac{\partial X^{(N)}}{\partial X^{(1)}} \leftarrow 0, \frac{\partial X^{(N)}}{\partial X^{(2)}} \leftarrow 0, \dots, \frac{\partial X^{(N)}}{\partial X^{(N)}} \leftarrow I\right)\right)$
For $f$ in functions inside the computational graph (in a forward fashion):
    $(Y^{(f,1)}, \dots, Y^{(f,M_f)}) \leftarrow f(X^{(f,1)}, \dots, X^{(f,N_f)})$   # forward pass
    $\frac{\partial Y^{(f,i)}}{\partial X^{(j)}} = \sum_{k=1}^{N_f} \frac{\partial Y^{(f,i)}}{\partial X^{(f,k)}} : \frac{\partial X^{(f,k)}}{\partial X^{(j)}}$   for all $i$ and $j$
Output: $\left(\left(\frac{\partial Y^{(1)}}{\partial X^{(1)}}, \dots, \frac{\partial Y^{(1)}}{\partial X^{(N)}}\right), \dots, \left(\frac{\partial Y^{(M)}}{\partial X^{(1)}}, \dots, \frac{\partial Y^{(M)}}{\partial X^{(N)}}\right)\right)$
```

where 0 denotes the tensor of zeros and I denotes the "identity tensor", e.g., if $X^{(1)} \in \mathbb{R}^3$, then

$$\left(\frac{\partial X^{(1)}}{\partial X^{(1)}}\right)_{(abc)(def)} = \begin{cases} 1 & \text{if } a = d, b = e, c = f \\ 0 & \text{otherwise} \end{cases}.$$

Note that we need to include a forward pass within Algorithm 2 (line 4) because we need to evaluate each $\partial Y^{(f,i)} / \partial X^{(f,k)}$ at the actual value of $X^{(f,k)}$. Despite the correctness of Algorithm 2, it doesn't represent how JAX implements `jax.jacfwd` at the code level, though it's worth emphasizing that the two ways are mathematically equivalent. In Section 3.2, we derive the algorithm for computing the "JVP" and show how it can be leveraged to compute the full "Jacobian".

3.2 Understanding `jac.jvp`

In certain scenarios, one doesn't need the full "Jacobian", which contains the partial derivative of every output variable with respect to every input variable. Instead, one might only want the partial derivatives of all output variables with respect to one specific input variable, i.e., a single scalar entry within the entire $(X^{(1)}, \dots, X^{(N)})$. Denoting this scalar input variable by $x \in \mathbb{R}$, we organize the desired partial derivatives as

$$\left(\frac{\partial Y^{(1)}}{\partial x}, \dots, \frac{\partial Y^{(M)}}{\partial x}\right), \tag{6}$$

where $\partial Y^{(i)} / \partial x$ has the same shape as $Y^{(i)}$.

We then make the important yet potentially non-trivial observation: we can obtain the quantity above using the following computation:

$$\left(\frac{\partial Y^{(1)}}{\partial X^{(1)}} : V^{(1)} + \dots + \frac{\partial Y^{(1)}}{\partial X^{(N)}} : V^{(N)}, \dots, \frac{\partial Y^{(M)}}{\partial X^{(1)}} : V^{(1)} + \dots + \frac{\partial Y^{(M)}}{\partial X^{(N)}} : V^{(N)}\right), \tag{7}$$

[9] Whenever we use the term "variable", we are referring to a scalar.

where (a) $V^{(j)}$ have the same shape as $X^{(j)}$ and (b) every entry of $(V^{(1)},\ldots,V^{(N)})$ is zero *except* the entry corresponding to x, the quantity we want to differentiate with respect to. This computation is called the "JVP"[10]. Let's verify that this is indeed what JAX's `jax.jvp` computes:

```
# let us choose x to be the (1, 2) entry of X1
# below are three ways of obtaining the same answer in jax

# first way
# indexing the result of jacfwd (the jacobian)

jac = jax.jacfwd(g, argnums=(0, 1))(X1, X2)
jvp_via_indexing = (jac[0][0][:, :, 1, 2], jac[1][0][:, :, 1, 2])

# second way
# contracting the jacobian with V1 and V2

V1, V2 = np.zeros(X1.shape), np.zeros(X2.shape)
V1 = V1.at[1, 2].set(1)

def contract(A, B):
    return np.einsum("ijkl,kl->ij", A, B)

jvp_after_jac_is_computed = (
    contract(jac[0][0], V1) + contract(jac[0][1], V2),
    contract(jac[1][0], V1) + contract(jac[1][1], V2)
)

# third way
# use jvp in jax

primals, tangents = jax.jvp(g, (X1, X2), (V1, V2))

print(np.allclose(jvp_via_indexing[0], tangents[0]))  # true
print(np.allclose(jvp_via_indexing[1], tangents[1]))  # true
print(np.allclose(jvp_after_jac_is_computed[0], tangents[0]))  # true
print(np.allclose(jvp_after_jac_is_computed[1], tangents[1]))  # true
```

While we could first compute the "Jacobian" and then contract the matrices it contains with $V^{(1)},\ldots,$ $V^{(M)}$ (the second way in the code example above), it turns out to be much more efficient to compute the "JVP" directly. Here's how this can be accomplished. Contract both sides of Equation 5 with $V^{(j)}$, obtain

$$\frac{\partial Y^{(i,f)}}{\partial X^{(j)}} : V^{(j)} = \left(\sum_{k=1}^{N_f} \frac{\partial Y^{(i,f)}}{\partial X^{(k,f)}} : \frac{\partial X^{(k,f)}}{\partial X^{(j)}}\right) : V^{(j)}.$$

[10] To see where the name "Jacobian-vector product" comes from, consider the case in which a computation graph takes a single vector input $\vec{x}$ and outputs another vector $\vec{y}$. In this case, the partial derivatives of all output variables with respect to a single input variable can be computed using

$$\frac{\partial \vec{y}}{\partial x_i} = \frac{\partial \vec{y}}{\partial \vec{x}} \vec{v} \quad \text{with} \quad \vec{v} = \vec{e}_i,$$

which is, unanimously, the product of a Jacobian and a vector. Clearly, this naming convention was kept even when engineers and researchers generalized the input and output of a computation graph to be more than one and/or higher-dimensional.

Since tensor contraction is distributive and commutative, we have

$$
\begin{aligned}
&= \sum_{k=1}^{N_f} \left(\frac{\partial Y^{(i,f)}}{\partial X^{(k,f)}} : \frac{\partial X^{(k,f)}}{\partial X^{(j)}} \right) : V^{(j)} \\
&= \sum_{k=1}^{N_f} \frac{\partial Y^{(i,f)}}{\partial X^{(k,f)}} : \left(\frac{\partial X^{(k,f)}}{\partial X^{(j)}} : V^{(j)} \right).
\end{aligned}
$$

Finally, summing across j, we have

$$
\underbrace{\frac{\partial Y^{(i,f)}}{\partial X^{(1)}} V^{(1)} + \cdots + \frac{\partial Y^{(i,f)}}{\partial X^{(N)}} V^{(N)}}_{\triangleq \dot{Y}^{(i,f)}} = \sum_{k=1}^{N_f} \frac{\partial Y^{(i,f)}}{\partial X^{(k,f)}} : \underbrace{\left(\frac{\partial X^{(k,f)}}{\partial X^{(1)}} : V^{(1)} + \cdots + \frac{\partial X^{(k,f)}}{\partial X^{(N)}} : V^{(N)} \right)}_{\triangleq \dot{X}^{(k,f)}},
$$

where we have defined two new quantities $\dot{Y}^{(i,f)}$ and $\dot{X}^{(k,f)}$. Again, using another "forward-style" algorithm (Algorithm 3), we can eventually obtain $\dot{Y}^{(f)}$, which is exactly what we wanted at the beginning of this sub-section (Expression 6 and 7).

TODO: Obviously, not strictly required to ones and zeros (?), directional derivative and so on

TODO: explain the words tangent and primals in the pushforward context

3.3 JAX uses `jac.jvp` to compute the "Jacobian"

Each time `jac.jvp` allows us to compute the partial derivatives of all output variables with respect to a single input variable. This suggests that, we can simply call `jac.jvp` once for ever input variable, and asemble the results from all the calls to the Jacobian. Indeed, this is what happens under the hood in JAX and here's how we might do it explicitly:

3.4 Defining custom JVP rules / pushforward rules via `jax.custom_jvp` and `f.defjvp`

Built-in functions of JAX certainly can be differentiated through, but what happens when incorporate code from another package into a computation graph. Can we still compute the JVP using JAX? Let's try it out

```
# previously, this was g
# def g(X1, X2):
#    Y1 = X1 @ X2
#    Y2 = X * X
#    return Y1, Y2

# suppose for some reason which we do not want to use matmul in jax but rather a
custom matmul function (which uses numpy)
# (jax.pure_callback is the way for including non-jax code in jax workflow)

def matmul(A, B):
    result_shape = jax.core.ShapedArray((A.shape[0], B.shape[1]), A.dtype)
    return jax.pure_callback(npy.matmul, result_shape, A, B)

def g2(A, B):
    C = matmul(A, B)
    D = A * A
    return C, D
```

Algorithm 3
Reverse-mode automatic differentiation for a “Jacobian-vector product”

```
Input: computation graph, primals $(X^{(1)}, \dots, X^{(N)})$, tangents $(V^{(1)}, \dots, V^{(N)})$

Initialize $(\dot{X}^{(1)} = V^{(1)}, \dots, \dot{X}^{(N)} = V^{(N)})$

For $f$ in functions inside the computational graph (in an appropriate order):

    $Y^{(f,1)}, \dots Y^{(f,M_f)} \leftarrow f(X^{(f,1)}, \dots X^{(f,N_f)})$

    For $i$ in $1, \dots M$:

        $\dot{Y}^{(f,i)} \leftarrow \sum_{k=1}^{N_f} \frac{\partial Y^{(f,i)}}{\partial X^{(f,k)}} : \dot{X}^{(f,k)}$    # line 6

Output: $\left(\dot{Y}^{(1)}\left(\frac{\partial Y^{(1)}}{\partial X^{(1)}} : V^{(1)} + \cdots + \frac{\partial Y^{(1)}}{\partial X^{(N)}} : V^{(N)}\right), \dots, \dot{Y}^{(M)}\left(\frac{\partial Y^{(M)}}{\partial X^{(1)}} : V^{(1)} + \cdots + \frac{\partial Y^{(M)}}{\partial X^{(N)}} : V^{(N)}\right)\right)$
```

If we try to naively use jax.jvp, we get into trouble

```
primals, tangents = jax.jvp(g2, (X1, X2), (V1, V2))
# ValueError: Pure callbacks do not support JVP. Please use ‘jax.custom_jvp‘ to
use callbacks while taking gradients.
```

Why is this? Notice line 6 in Algorithm 3 is called the JVP rule for f, we must know how to do this for f yet doesn't know this. Defining the custom JVP rule (derived in Section XX):

```
@jax.custom_jvp
def matmul(A, B):
    result_shape = jax.core.ShapedArray((A.shape[0], B.shape[1]), A.dtype)
    return jax.pure_callback(npy.matmul, result_shape, A, B)

@matmul.defjvp
def matmul_jvp(primals, tangents):
    A, B = primals
    A_dot, B_dot = tangents
    primal_out = matmul(A, B)
    tangent_out = matmul(A_dot, B) + matmul(A, B_dot)
    return primal_out, tangent_out
```

Now jax.jvp does work and give the correct answer

```
_, tangents_from_g = jax.jvp(g, (X1, X2), (V1, V2))
_, tangents_from_g2 = jax.jvp(g2, (X1, X2), (V1, V2))
print(np.allclose(tangents_from_g[0], tangents_from_g2[0]))  # True
print(np.allclose(tangents_from_g[1], tangents_from_g2[1]))  # True
```

Of course, there are also other reasons why we might want to define the JVP rule by ourselves. Another use case is when functions are defined implicitly. I really hope to include an example of this.

```
# TODO
```

TODO: mention how sometimes defining the JVP rule also allows you to do reverse mode (maybe give some examples after the next section?

Algorithm 4
Reverse-mode automatic differentiation for computing the "Jacobian"

```
Input: computation graph, $(X^{(1)}, \ldots, X^{(N)})$
Initialize $\left(\left(\frac{\partial Y^{(1)}}{\partial Y^{(1)}}, \ldots, \frac{\partial Y^{(1)}}{\partial Y^{(M)}}\right), \ldots, \left(\frac{\partial Y^{(M)}}{\partial Y^{(1)}}, \ldots, \frac{\partial Y^{(M)}}{\partial Y^{(M)}}\right)\right)$
Forward pass
For $f$ in functions inside the computational graph (in a backward fashion):
    $\frac{\partial Y^{(i)}}{\partial X^{(f,j)}} = \sum_{k=1}^{M_f} \frac{\partial Y^{(i)}}{\partial Y^{(f,k)}} : \frac{\partial Y^{(f,k)}}{\partial X^{(f,j)}}$ for all $i$ and $j$   # need stored values from forward pass
Output: $\left(\left(\frac{\partial Y^{(1)}}{\partial X^{(1)}}, \ldots, \frac{\partial Y^{(1)}}{\partial X^{(N)}}\right), \ldots, \left(\frac{\partial Y^{(M)}}{\partial X^{(1)}}, \ldots, \frac{\partial Y^{(M)}}{\partial X^{(N)}}\right)\right)$
```

4 Reverse-mode automatic differentiation

In this section, we discuss reverse-mode AD for computing the full "Jacobian" and a "vector-Jacobian product" ("VJP") of a computation graph.

4.1 Computing the full "Jacobian"

If we, again, apply Equation 4 to some f inside the computation graph g, we also see that

$$\frac{\partial Y^{(i)}}{\partial X^{(f,j)}} = \sum_{k=1}^{M_f} \frac{\partial Y^{(i)}}{\partial Y^{(f,k)}} : \frac{\partial Y^{(f,k)}}{\partial X^{(f,j)}} \quad \text{for all } i \text{ and } j. \tag{8}$$

This relationship shows that, if we want to compute the partial derivatives of g's output variables with respect to f's input variables, we must first know the partial derivatives of g's output variables with respsect to f's output variables. As a result, it inspires a "reverse-style" algorithm (i.e., the algorithm first goes through variables closer to g's output variables) for computing the partial derivatives of g's output variables with respect to g's input variables (Algorithm 4).

Note that we need to perform a forward pass and store all intermediate values within Algorithm 4 before everything because we need to evaluate each $\partial Y^{(f,k)} / \partial X^{(f,j)}$ at the actual value of $X^{(f,j)}$. Despite the correctness of Algorithm 4, it does not represent how JAX implements `jax.jacrev`. In Section 4.2, we derive the algorithm for computing the "VJP" and, in Section 4.3 show how it can be leveraged to compute the full "Jacobian".

4.2 Understanding `jax.vjp`

Instead of the full "Jacobian", one might only want the partial derivatives of one specific output variable (i.e., a single scalar entry within the entire $(X^{(1)}, \ldots, X^{(M)})$) with respect to all input variables. Denoting this scalar output variable by $y \in \mathbb{R}$, we organize the desired partial derivatives as

$$\left(\frac{\partial y}{\partial X^{(1)}}, \ldots, \frac{\partial y}{\partial X^{(N)}}\right) \tag{9}$$

where $\partial y / \partial X^{(i)}$ has the same shape as $X^{(i)}$. It turns out that we can obtain the quantity above using the following computation:

$$\left(W^{(1)} : \frac{\partial Y^{(1)}}{\partial X^{(1)}} + \cdots + W^{(M)} : \frac{\partial Y^{(M)}}{\partial X^{(1)}}, \ldots, W^{(1)} : \frac{\partial Y^{(1)}}{\partial X^{(N)}} + \cdots + W^{(M)} : \frac{\partial Y^{(M)}}{\partial X^{(N)}}\right), \tag{10}$$

where (a) $W^{(j)}$ have the same shape as $X^{(j)}$ and (b) every entry of $(W^{(1)}, \ldots, W^{(M)})$ is zero *except* the entry corresponding to y. This computation is called the "VJP"[11]. Let's verify that this is indeed what JAX's `jax.vjp` computes:

```
# make note of computation shapes here

# let us choose y to be the (1, 2) entry of Y1
# below are three ways of obtaining the same answer in jax

# first way
# indexing the result of jacrev (the jacobian)

jac = jax.jacrev(g, argnums=(0, 1))(X1, X2)
vjp_via_indexing = (jac[0][0][1, 2, :, :], jac[0][1][1, 2, :, :])

# second way
# contracting the jacobian with W1 and W2

W1, W2 = np.zeros(Y1.shape), np.zeros(Y2.shape)
W1 = W1.at[1, 2].set(1)

def contract(A, B):
    return np.einsum("ij,ijkl->kl", A, B)

vjp_after_jac_is_computed = (
    contract(W1, jac[0][0]) + contract(W2, jac[1][0]),
    contract(W1, jac[0][1]) + contract(W2, jac[1][1])
)

# third way
# use vjp in jax

primals, vjpfun = jax.vjp(g, X1, X2)
cotangents = vjpfun((W1, W2))

print(np.allclose(vjp_via_indexing[0], cotangents[0]))  # true
print(np.allclose(vjp_via_indexing[1], cotangents[1]))  # true
print(np.allclose(vjp_after_jac_is_computed[0], cotangents[0]))  # true
print(np.allclose(vjp_after_jac_is_computed[1], cotangents[1]))  # true
```

While we could first compute the "Jacobian" and then contract the matrices it contains with $W^{(1)}, \ldots, W^{(M)}$ (the second way in the code example above), it turns out to be much more efficient to compute the "VJP" directly. Here's how this can be accomplished. Pre-Contract both sides of Equation (should be 8 here) with $W^{(i)}$, obtain

$$W^{(i)} \colon \frac{\partial Y^{(i)}}{\partial X^{(f,j)}} = W^{(i)} \colon \left(\sum_{k=1}^{M_f} \frac{\partial Y^{(i)}}{\partial Y^{(f,k)}} \colon \frac{\partial Y^{(f,k)}}{\partial X^{(f,j)}} \right)$$

[11] To see where the name "vector-Jacobian product" comes from, consider the case in which a computation graph takes a single vector input $\vec{x}$ and outputs another vector $\vec{y}$. In this case, the partial derivatives of a single output variable with respect to all input variables can be computed using

$$\frac{\partial y_i}{\partial \vec{x}} = \vec{v}^T \frac{\partial \vec{y}}{\partial \vec{x}} \quad \text{with} \quad \vec{v} = \vec{e}_i,$$

which is, unanimously, the product of a vector (though transposed) and a Jacobian. Clearly, this naming convention was kept even when engineers and researchers generalized the input and output of a computation graph to be more sdthan one and/or higher-dimensional.

```
Algorithm 5
Reverse-mode automatic differentiation for a “vector-Jacobian product”
Input: computation graph, primals $(X^{(1)},\ldots,X^{(N)})$, cotangents $(W^{(1)},\ldots,W^{(M)})$

Initialize $(\bar{Y}^{(1)}=W^{(1)},\ldots,\bar{Y}^{(M)}=W^{(M)})$

Forward pass

For $f$ in functions inside the computational graph (in an appropriate order):

    For $i$ in $1,\ldots M$:

        $\bar{X}^{(f,i)} \leftarrow \sum_{k=1}^{M_f} \bar{Y}^{(f,k)}:\frac{\partial Y^{(f,k)}}{\partial X^{(f,j)}}$

Output: $\Big(\dot{Y}^{(1)}\Big(\frac{\partial Y^{(1)}}{\partial X^{(1)}}:V^{(1)}+\cdots+\frac{\partial Y^{(1)}}{\partial X^{(N)}}:V^{(N)}\Big),\ldots,\dot{Y}^{(M)}\Big(\frac{\partial Y^{(M)}}{\partial X^{(1)}}:V^{(1)}+\cdots+\frac{\partial Y^{(M)}}{\partial X^{(N)}}:V^{(N)}\Big)\Big)$
```

Since tensor contraction is distributive and commutative, we have

$$\begin{aligned}
&= \sum_{k=1}^{M_f} W^{(i)}:\left(\frac{\partial Y^{(i)}}{\partial Y^{(f,k)}}:\frac{\partial Y^{(f,k)}}{\partial X^{(f,j)}}\right) \\
&= \sum_{k=1}^{M_f} \left(W^{(i)}:\frac{\partial Y^{(i)}}{\partial Y^{(f,k)}}\right):\frac{\partial Y^{(f,k)}}{\partial X^{(f,j)}}
\end{aligned}$$

Finally, summing across i, we have

$$\underbrace{W^{(1)}:\frac{\partial Y^{(1)}}{\partial X^{(f,j)}}+\cdots+W^{(M)}:\frac{\partial Y^{(M)}}{\partial X^{(f,j)}}}_{\triangleq \bar{X}^{(f,i)}} = \sum_{k=1}^{M_f} \underbrace{\left(W^{(1)}:\frac{\partial Y^{(1)}}{\partial Y^{(f,k)}}+\cdots+W^{(M)}:\frac{\partial Y^{(M)}}{\partial Y^{(f,k)}}\right)}_{\triangleq \bar{Y}^{(f,k)}}:\frac{\partial Y^{(f,k)}}{\partial X^{(f,j)}},$$

where we have defined two new quantities $\bar{X}^{(f,i)}$ and $\bar{Y}^{(f,k)}$. Using another “reverse-style” algorithm (Algorithm 3), we can eventually obtain $\bar{X}^{(f)}$, which is exactly what we wanted at the beginning of this sub-section (Expression 6 and 7).

TODO: Obviously, not strictly required to ones and zeros (?), directional derivative and so on

TODO: explain the words tangent and primals in the pushforward context

4.3 Understanding `jax.jacrev`

Each time `jac.vjp` allows us to compute the partial derivatives of a single output variable with respect to all input variables. This suggests that, we can simply call `jac.vjp` once for every output variable, and asemble the results from all the calls to the Jacobian. Indeed, this is what happens under the hood in JAX and here’s how we might do it explicitly:

5 Comparing forward mode and reverse mode

6 Derivations of some JVP / pushforward rules

6.1 Scalar addition

6.2 Scalar multiplication

6.3 Scalar sine

6.4 Broadcasted function

6.5 Matrix-vector product

How can I adapt what I derived, to multiple inputs and outputs? Or when inputs and outputs are not vectors?

$$J_{\text{after } f}(\boldsymbol{x})\boldsymbol{v} = J_f(\boldsymbol{x}^{(f)})\dot{\boldsymbol{x}}^{(f)}$$

First case, multiple inputs:

$$J_{\text{after } f}(\boldsymbol{x})\boldsymbol{v} = J_f(\boldsymbol{x}^{(f)})\dot{\boldsymbol{x}}^{(f)}$$

We can simply break the Jacobian vector products into multiple pieces? Horizontal slices

Second case, matrice inputs and matrix outputs

6.6 Scalar root-finding

6.7 Matrix-matrix product

Theorem 2. *(JVP pushforward rule for matrix-matrix multiplication)*

Let function f represent matrix-matrix multiplication

$$f(A,B) = AB = C,$$

where $A \in \mathbb{R}^{n\times m}, B \in \mathbb{R}^{m\times o}, C \in \mathbb{R}^{n\times o}$

“JVP” rule:

$$\underbrace{\dot{C}}_{(N,O)} = \underbrace{\frac{\partial C}{\partial A}}_{((N,O),(N,M))} : \underbrace{\dot{A}}_{(N,M)} + \underbrace{\frac{\partial C}{\partial B}}_{((N,O),(M,O))} : \underbrace{\dot{B}}_{(M,O)}$$

$$\begin{aligned}
\dot{C}_{i,j} &= \sum_{k,l}\left(\frac{\partial C}{\partial A}\right)_{(i,j),(k,l)}\dot{A}_{k,l} + \sum_{k,l}\left(\frac{\partial C}{\partial B}\right)_{(i,j),(k,l)}\dot{B}_{k,l} \\
&= \sum_{k,l}\frac{\partial C_{i,j}}{\partial A_{k,l}}\dot{A}_{k,l} + \sum_{k,l}\frac{\partial C_{i,j}}{\partial B_{k,l}}\dot{B}_{k,l} \\
&= \sum_{k,l}\frac{\partial(\sum_m A_{i,m}B_{m,j})}{\partial A_{k,l}}\dot{A}_{k,l} + \sum_{k,l}\frac{\partial(\sum_m A_{i,m}B_{m,j})}{\partial B_{k,l}}\dot{B}_{k,l} \\
&= \sum_{k,l}\delta_{i,k}B_{l,j}\dot{A}_{k,l} + \sum_{k,l}\delta_{l,j}A_{i,k}\dot{B}_{k,l} \\
&= \sum_{l}B_{l,j}\dot{A}_{i,l} + \sum_{k}A_{i,k}\dot{B}_{k,j} \\
&= \sum_{l}\dot{A}_{i,l}B_{l,j} + \sum_{k}A_{i,k}\dot{B}_{k,j}
\end{aligned}$$

Therefore

$$\dot{C} = \dot{A}B + A\dot{B}$$

6.8 L2 loss

6.9 Linear system

Let function f represent matrix-matrix multiplication

$$f(A,b) = \{\text{solve } Ax = b \text{ for } x\} =: x$$

$A \in \mathbb{R}^{N\times N}, b \in \mathbb{R}^N, x \in \mathbb{R}^N$

$$\dot{x} = \frac{\partial x}{\partial A} : \dot{A} + \frac{\partial x}{\partial b} : \dot{b}$$

Implicit function

$$\begin{aligned}
\dot{x}_j &= \sum_{k,l} \left(\frac{\partial x}{\partial A}\right)_{j,(k,l)} \dot{A}_{k,l} + \sum_k \left(\frac{\partial x}{\partial b}\right)_{j,k} \dot{b}_k \\
&= \underbrace{\sum_{k,l} \frac{\partial x_j}{\partial A_{kl}} \dot{A}_{kl}}_{\triangleq \dot{x}_i^{[A]}} + \underbrace{\sum_j \frac{\partial x_j}{\partial b_k} \dot{b}_k}_{\triangleq \dot{x}_i^{[b]}}
\end{aligned}$$

How does a specific x element relate to a specific b element?

$$\begin{aligned}
Ax &= b \\
\sum_j A_{i,j} x_j &= b_i \\
\left(\sum_j A_{i,j} x_j\right) &= \frac{\partial}{\partial b_k}(b_i) \\
\sum_j A_{i,j} \frac{\partial x_j}{\partial b_k} &= \delta_{ik} \\
\sum_k \left(\sum_j A_{i,j} \frac{\partial x_i}{\partial b_k}\right) \dot{b}_k &= \sum_k \delta_{ik} \dot{b}_k \quad \text{(dot product)} \\
\sum_i A_{l,i} \left(\sum_j \frac{\partial x_i}{\partial b_j} \dot{b}_j\right) &= \dot{b}_k \\
\sum_i A_{l,i} \dot{x}_i^{[b]} &= \dot{b}_l \\
A\dot{x}^{[b]} &= \dot{b}
\end{aligned}$$

which is also a linear system!

6.10 Nonlinear system solve

6.11 Neural ODE

6.12 Softmax