Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 5: Automatic differentiation (forward-mode and reverse-mode)

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Automatic differentiation

Forward mode

Neural networks are compositions of blocks of the form $\mathbf{h} = f(\mathbf{x}, \mathbf{w})$, where:

- $\mathbf{x}$ is the input (possibly the output of a former block);
- $\mathbf{w}$ is a vector of trainable parameters (e.g., all elements in $\mathbf{W}$ and $\mathbf{b}$ in a fully-connected layer).

Assuming d, p, and o are the output shapes of $\mathbf{x}$, $\mathbf{w}$, and $\mathbf{h}$, to each block we can associate two Jacobians:

$$\underset{(o,d)}{\partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})} \quad \underset{(o,p)}{\partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})} \,. \tag{1}$$

We consider vectors for notational simplicity, because in the multi-dimensional case, Jacobians can have *a lot* of indexes:

```
x = tf.random.normal((3, 4))
w = tf.Variable(tf.random.normal((4, 5)))
with tf.GradientTape() as tape:
   h = x @ w
tape.jacobian(h, w).shape # Print: TensorShape([3, 5, 4, 5])
```

Given a tensor $x_{(a,b,\ldots)}$, it is **isomorphic** (equivalent) to a vector $\mathbf{x}_{(ab\ldots)}$. Because of this, our formulation is actually quite generic.

The previous code, in fact, is equivalent to:

```
x = tf.random.normal((12,))
w = tf.Variable(tf.random.normal((20,)))
with tf.GradientTape() as tape:
  h = tf.reshape(x, ((3, 4)) @ tf.reshape(w, (4, 5))
  h = tf.reshape(h, (-1,))
tape.jacobian(h, w).shape # Print: TensorShape([15, 20])
```

(This is of course not something you should do in practice.)

- **Fully-connected layers**:

$$f(\mathbf{x}, \{\mathbf{W}, \mathbf{b}\}) = \mathbf{W}\mathbf{x} + \mathbf{b}\,. \tag{2}$$

- **Elementwise non-linearity** (activation functions):

$$f(\mathbf{x}, \{\}) = \phi(\mathbf{x})\,. \tag{3}$$

- Quite a few more to come...

Our notation is inspired by *functional* frameworks such as Jax:

```
def f(W, x):
  return jnp.tanh(jnp.dot(x, W))
```

In *object-oriented* frameworks (TensorFlow, PyTorch), blocks are instead instances of classes, and parameters are specially-defined properties:

```
class F(Layer):
  def __init__(self, d, o):
    self.W = tf.Variable(tf.random.normal((d, o)))
  def __call__(self, x):
    return x @ self.W
```

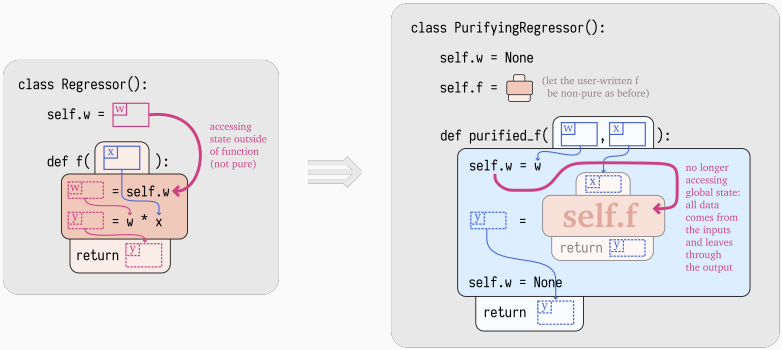

Figure 1: We can move from an OOP implementation to a functional one by a process called *purification*:
From PyTorch to JAX: towards neural net frameworks that purify stateful code.

We assume we have a variable number of blocks, but the last one is always a sum (most of the time, to aggregate the per-batch losses):

$$\begin{aligned} \mathbf{h}_1 &= f_1(\mathbf{x}, \mathbf{w}_1) \\ \mathbf{h}_2 &= f_2(\mathbf{h}_1, \mathbf{w}_2) \\ &\vdots \\ \mathbf{h}_l &= f_l(\mathbf{h}_{l-1}, \mathbf{w}_l) \\ y &= \sum \mathbf{h}_l \end{aligned}$$

We want an *efficient* algorithm to compute the parameters' gradients:

$$\{\partial_{\mathbf{w}_i} y\} \quad i = 1, \ldots, l\,. \tag{4}$$

Remember the chain rule for Jacobians:

$$\partial [f \circ g] = \partial f \ \circ \ \partial g \, . \tag{5}$$

We can interpret the chain rule as follows:

- if we have already computed g and its corresponding ∂g ...
- ... and we update our output as $f \circ g$ ...
- ... we need to update the corresponding Jacobian as $\partial f \ \circ \ \partial g$.

This is the key behind **forward-mode automatic differentiation** (forward autodiff).

Consider 3 layers as follows:

$$
\begin{aligned}
\underset{(o_1)}{\mathbf{h}_1} &= \underset{(d)}{f_1}(\mathbf{x}, \underset{(p_1)}{\mathbf{w}_1}) \\
\underset{(o_2)}{\mathbf{h}_2} &= \underset{(o_1)}{f_2}(\mathbf{h}_1, \underset{(p_2)}{\mathbf{w}_2}) \\
\underset{(o_3)}{\mathbf{h}_3} &= \underset{(o_2)}{f_3}(\mathbf{h}_2, \underset{(p_3)}{\mathbf{w}_3}) \\
y &= \langle \mathbf{h}_3, \mathbf{1} \rangle .
\end{aligned}
$$

In forward-mode autodiff, we use the chain rule to update all gradients we are interested into *after every instruction*.

More formally, for each layer in the network, forward-mode autodiff proceeds as follows:

1. Compute the new output $\mathbf{h}_i = f_i(\mathbf{h}_{i-1}, \mathbf{w}_i)$.
2. For $\mathbf{w}_i$, initialize the so-called **tangent** matrix:

$$\widehat{\mathbf{W}}_i = \partial_{\mathbf{w}_i} \mathbf{h}_i \,.$$

   Some layers might not have parameters, in which case skip this step.
3. For previous parameters, update their gradient using the chain rule:

$$\widehat{\mathbf{W}}_j = \left[\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i\right] \cdot \widehat{\mathbf{W}}_j \,, \quad j < i \,.$$

We start by computing the output of the first layer, and the corresponding gradient with respect to $\mathbf{w}_1$:

Original instruction

$$\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$$

Additional instructions

$$\widehat{\mathbf{W}}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$$

After our second operation, we need to update the gradient with respect to w_1, and initialize the one with respect to w_2:

Original instruction

$$\mathsf{h}_2 = f_2(\mathsf{h}_1, \mathsf{w}_2)\,.$$

Additional instructions

$$\underset{(o_2,p_1)}{\widehat{\mathsf{W}}_1} = \underset{(o_2,o_1)}{\left[\partial_{\mathsf{h}_1}\mathsf{h}_2\right]} \cdot \underset{(o_1,p_1)}{\widehat{\mathsf{W}}_1}\,,$$

$$\widehat{\mathsf{W}}_2 = \partial_{\mathsf{w}_2}\mathsf{h}_2\,.$$

At this point, we keep iterating our procedure:

Original instruction

$$\mathsf{h}_3 = f_3(\mathsf{h}_2, \mathsf{w}_3)\,.$$

Additional instructions

$$\widehat{\mathsf{W}}_1 = \left[\partial_{\mathsf{h}_2}\mathsf{h}_3\right] \cdot \widehat{\mathsf{W}}_1\,,$$
$$\widehat{\mathsf{W}}_2 = \left[\partial_{\mathsf{h}_2}\mathsf{h}_3\right] \cdot \widehat{\mathsf{W}}_2\,,$$
$$\widehat{\mathsf{W}}_3 = \partial_{\mathsf{w}_3}\mathsf{h}_3\,.$$

After our final operation, we have the gradients we wanted:

Original instruction

$$y = \langle \mathsf{h}_3, \mathbf{1} \rangle \,.$$

Additional instructions

$$\nabla_{\mathsf{w}_1} y = \langle \widehat{\mathsf{W}}_1, \mathbf{1} \rangle \,,$$
$$\nabla_{\mathsf{w}_2} y = \langle \widehat{\mathsf{W}}_2, \mathbf{1} \rangle \,,$$
$$\nabla_{\mathsf{w}_3} y = \langle \widehat{\mathsf{W}}_3, \mathbf{1} \rangle \,.$$

All done! That was easy… but was it efficient?

Our gradients' estimates can be easily interleaved with the main operations. In addition, the previous estimates can be discarded after each update, making it highly **memory efficient**.

On the other hand, the main operation required by forward-mode autodiff is an $(o_i, o_{i-1}) \times (o_{i-1}, p_j)$ multiplication, which scales linearly in the number of parameters. This makes it **highly** time consuming!

When running a mini-batch of n data points, all terms $o_1, o_2, \ldots$ will have a factor n inside, and the multiplication will scale quadratically in n.

In order to find a better solution, let us unroll one entire gradient computation:

$$\underset{(p_1)}{\nabla_{\mathbf{w}_1} y} = \underset{(p_1,o_1)}{\partial^\top_{\mathbf{w}_1} \mathbf{h}_1} \cdot \underset{(o_1,o_2)}{\partial^\top_{\mathbf{h}_1} \mathbf{h}_2} \cdot \underset{(o_2,o_3)}{\partial^\top_{\mathbf{h}_2} \mathbf{h}_3} \cdot \underset{(o_3)}{\mathbf{1}} \ . \tag{6}$$

Forward−mode →

← Reverse−mode

If we compute all operations in reverse (**reverse mode**), we only require matrix-vector products, which for the most part are independent of p_1 and o_1! The next algorithm implements an efficient way to do this.

1. Compute the output of all layers, storing each intermediate value. Set $\widetilde{\mathbf{h}} = \mathbf{1}$.
2. Going in reverse, $i = l, l-1, \cdots, 1$, compute the gradient of the parameters of the current layer:

$$\nabla_{\mathbf{w}_i} y = \left[\partial^{\top}_{\mathbf{w}_i} \mathbf{h}_i\right] \cdot \widetilde{\mathbf{h}}$$

3. Update the gradient of y with respect to $\mathbf{h}_{i-1}$ exploiting again the chain rule:

$$\widetilde{\mathbf{h}} = \left[\partial^{\top}_{\mathbf{h}_{i-1}} \mathbf{h}_i\right] \cdot \widetilde{\mathbf{h}}$$

Let us look at the operations required in our example:

$$\widetilde{\mathsf{h}} = 1$$

$$\nabla_{\mathsf{w}_3} y = \left[\partial_{\mathsf{w}_3}^\top \mathsf{h}_3\right] \cdot \widetilde{\mathsf{h}}\,, \quad \widetilde{\mathsf{h}} = \left[\partial_{\mathsf{h}_2}^\top \mathsf{h}_3\right] \widetilde{\mathsf{h}}\,,$$

$$\nabla_{\mathsf{w}_2} y = \left[\partial_{\mathsf{w}_2}^\top \mathsf{h}_2\right] \cdot \widetilde{\mathsf{h}}\,, \quad \widetilde{\mathsf{h}} = \left[\partial_{\mathsf{h}_1}^\top \mathsf{h}_2\right] \widetilde{\mathsf{h}}\,,$$

$$\nabla_{\mathsf{w}_1} y = \left[\partial_{\mathsf{w}_1}^\top \mathsf{h}_1\right] \cdot \widetilde{\mathsf{h}}\,.$$

This is called the **reverse (or adjoint) program**.

Reverse-mode autodiff requires *a lot* of memory, because we need to store all intermediate outputs when executing the main (primal) program.

However, the reverse program only requires matrix-vector products that scale linearly in n. Empirically, the execution of the adjoint program requires 3x-4x the time of the main one.

Reverse-mode autodiff is more or less a standard in computing gradients of deep neural networks. In this context, it is also called **backpropagation**. The primal and adjoint program are called **forward pass** and **backward pass**.

It is easy to extend our derivation beyond linear programs, to acyclic computational graphs. In particular, if a weight participates in multiple operations (**weight sharing**), its contribution is the sum of the two gradients.

Baydin, A.G., Pearlmutter, B.A., Radul, A.A. and Siskind, J.M., 2018. **Automatic differentiation in machine learning: a survey**. *Journal of Machine Learning Research*, 18.

There is a lot we are not able to cover, notably how to *implement* autodiff, acyclic graphs, etc. Below a few pointers for advanced material:

- `https://mblondel.org/teaching/autodiff-2020.pdf`
- `https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/slides/lec10.pdf`
- `https://jax.readthedocs.io/en/latest/autodidax.html`

- Wengert (1964) is credited as the first description of forward-mode AD, which became popular in the 80' mostly with the work of Griewank.
- Linnainmaa (1976) is considered the first description of modern reverse-mode AD, with the fist major implementation in Speelpenning (1980).
- Werbos (1982) is the first concrete application to NNs, before being popularized (as backpropagation) by Rumelhart et al. (1986).

https://www.math.uni-bielefeld.de/documenta/vol-ismp/52_griewank-andreas-b.pdf

Automatic differentiation

Autodiff in practice

One important consequence of the previous reasoning is that we do not truly need Jacobians, as much as the following quantities:

$$\text{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w}) \tag{7}$$

$$\text{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w}) \tag{8}$$

We call these **vector-Jacobian products**. They can be significantly easier to computer than standard Jacobians.

Remember that $\left[\mathbf{A}^\top \mathbf{v}\right]^\top = \mathbf{v}^\top \mathbf{A}$, which explains the name. Feel free to transpose everything if you prefer.

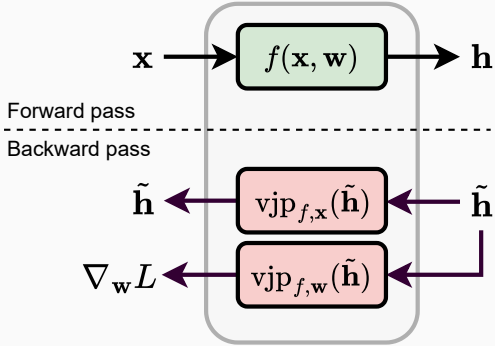

Figure 2: For performing R-AD, primitives must be augmented with two VJP operations to be able to perform a backward pass, corresponding to the input VJP (7) and the weight VJP (8). One call for each is sufficient to perform the backward pass through the primitive.

We can recover the Jacobians' computation by repeatedly calling the VJPs with the basis vectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$, to generate them one row at a time, e.g., for the input Jacobian we have:

$$\partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w}) = \begin{bmatrix} \text{vjp}_{f,\mathbf{x}}(\mathbf{e}_1) \\ \text{vjp}_{f,\mathbf{x}}(\mathbf{e}_2) \\ \vdots \\ \text{vjp}_{f,\mathbf{x}}(\mathbf{e}_n) \end{bmatrix}$$

Let us consider for example:

$$\underset{(o)}{f(\mathbf{x}, \mathbf{W})} = \underset{(o,d)(d)}{\mathbf{W}\ \mathbf{x}}\ .$$

In this case, trivially:

$$\partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\,.$$

However, the Jacobian with respect to $\mathbf{W}$ is a (o, o, d) tensor!

Can you compute it?

The VJPs are both simpler:

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \mathbf{W}^\top \mathbf{v}\,,$$
$$\text{vjp}_{\mathbf{W}}(f, \mathbf{v}) = \mathbf{x}\mathbf{v}^\top\,.$$

Practically, the core part of a framework like TensorFlow can be understood as a collection of differentiable operations f (**primitives**) together with their corresponding VJPs.

To define new primitives, one has to define both their operation and their VJP behaviour to use them in backpropagation: `https://www.tensorflow.org/guide/create_op`.

The other operation we have seen is an element-wise nonlinearity, which does not have adaptable parameters:

$$f(\mathbf{x}, \{\}) = \phi(\mathbf{x}) \,. \tag{9}$$

The Jacobian is a (d, d) diagonal matrix:

$$[\partial f(\mathbf{x}, \{\})]_{i,i} = \phi'(x_i) \,. \tag{10}$$

The JVP is instead:

$$\mathrm{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \phi'(\mathbf{x}) \odot \mathbf{v} \,. \tag{11}$$

To store all intermediate operations in TensorFlow, we can use a `tf.GradientTape` object:

```
w1 = tf.Variable(tf.random.normal((...)))
w2 = tf.random.normal((...))
with tf.GradientTape() as tape:
  # Operations inside the tape are recorded,
  # assuming at least one operand is 'watched'
  tape.watch(w2)
  ...
  y = tf.reduce_sum(h)
```

See Alice's Adventures in a Differentiable Wonderland for more details and an example of reimplementing a simple autodiff tool.

Once the forward pass is completed, we can compute gradients using the tape:

```
g = tape.gradient(y, [w1, w2])
```

The tape also allows for Jacobians, although this basically requires one backward pass for each output.

Support for forward-mode autodiff is also available inside TensorFlow with `tf.autodiff.ForwardAccumulator`.

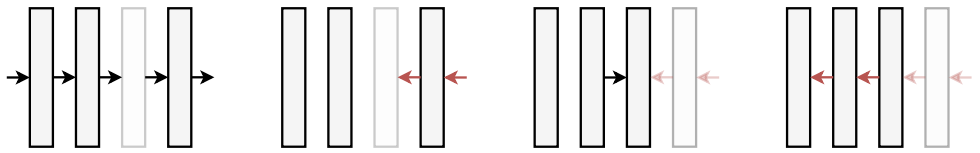

Figure 3: An example of **gradient checkpointing**. (a) We execute a forward pass, but we only store the outputs of the first, second, and fourth blocks (**checkpoints**). (b) The backward pass (red arrows) stops at the third block, whose activations are not available. (c) We run a second forward pass starting from the closest checkpoint to materialize again the activations. (d) We complete the forward pass. Compared to a standard backward pass, this requires 1.25x more computations. In general, the less checkpoints are stored, the higher the computational cost of the backward pass.

Automatic differentiation

Choosing an activation function

Understanding back-propagation gives some interesting insights into how to choose a proper activation function.

Remember that in this case:

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \phi'(\mathbf{x}) \odot \mathbf{v}\,. \tag{12}$$

Whenever a value goes through an activation function, during backpropagation we multiply by the derivative of the function.

If we have many layers, we make a lot of these multiplications. As a result:

- If $\phi'(\cdot) < 1$ always, the gradient will go to zero exponentially fast in the number of layers (**vanishing gradient**).
- If $\phi'(\cdot) > 1$ always, the gradient will explode exponentially fast in the number of layers (**exploding gradient**).

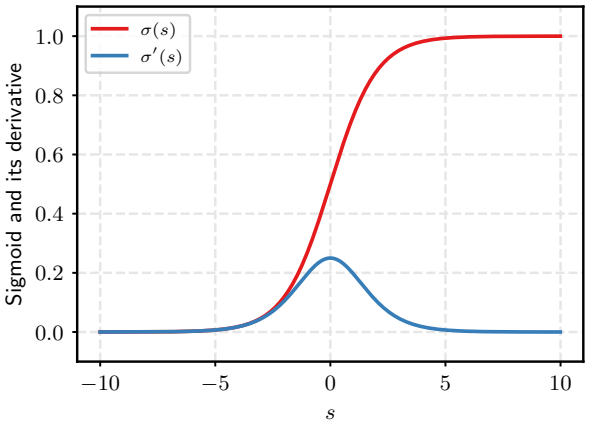

Figure 4: The sigmoid function $\sigma(\cdot)$ is a poor choice as activation function (for deep networks), because its derivative is bounded in $[0, 0.25]$.

A very common choice for deep networks is the **rectified linear unit** (ReLU), defined as:

$$\phi(s) = \max(0, s) \ . \tag{13}$$

Its derivative is either 1 (whenever $s > 0$), or 0 otherwise.

(The ReLU is not differentiable for $s = 0$, but this can be easily taken care of with the notion of **subgradients**).

ReLU is a good default choice in most applications.

The **subderivative** of a convex function $f : \mathbb{R} \to \mathbb{R}$ at a point x is a point g such that:

$$f(z) - f(x) \geq g(z - x) \ \forall z \in \mathbb{R}\,.$$

Similar extensions exist also for non-convex and vector-valued functions. For example, any point in $[0, 1]$ is a subderivative of ReLU at 0. Most frameworks use 0 by default.

Provably Correct Automatic Subdifferentiation for Qualified Programs (`https://arxiv.org/abs/1809.08530`), A mathematical model for automatic differentiation in machine learning (`https://arxiv.org/abs/2006.02080`).

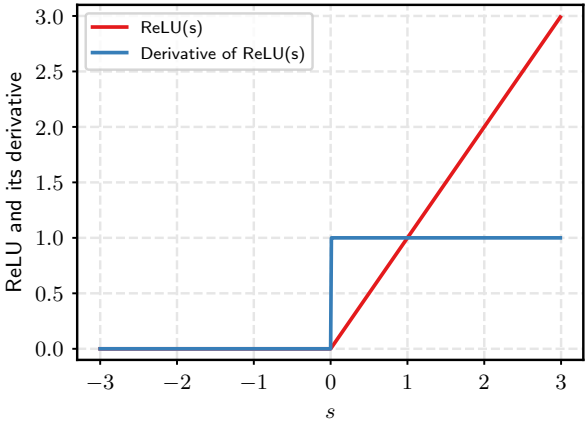

Figure 5: A plot of ReLU and its derivative.

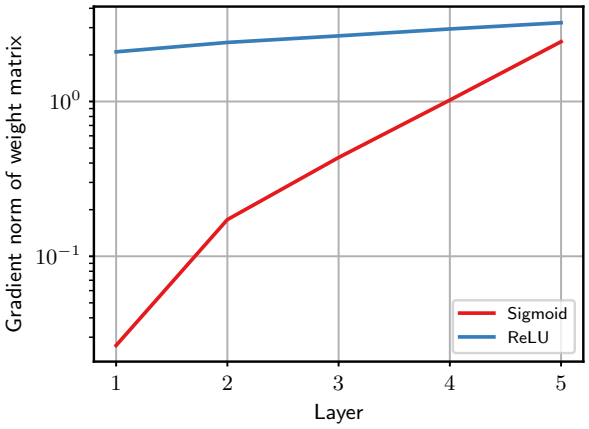

Figure 6: We initialize a NN with 5 hidden layers. Weights are sampled from the uniform distribution on $[-0.5, 0.5]$. We show the norm of a typical gradient (cross-entropy on a few examples) with sigmoid and ReLU activation functions.

- **Chapter 6** in the book.
- Baydin, A.G., Pearlmutter, B.A., Radul, A.A. and Siskind, J.M., 2018. **Automatic differentiation in machine learning: a survey**. *Journal of Machine Learning Research*, 18.
- There are many didactical libraries for learning about autodiff (e.g., MicroGrad, TinyGrad, MiniTorch, ...).