Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 3: Fully-connected neural networks and autodiff

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Feedforward neural networks

Limitations of linear models

To understand the limitations of a linear model $f(\mathbf{x})$, consider an input vector $\widehat{\mathbf{x}}$, equal to $\mathbf{x}$ but for a single feature $\widehat{x}_i = 2x_i$ (e.g., a client with double income).

The output on the two inputs is related by:

$$f(\widehat{\mathbf{x}}) = f(\mathbf{x}) + w_i x_i \,. \tag{1}$$

Effects are linearly super-imposed: it is impossible to model some form of interaction between two features (e.g., 'a client with 10k income is not trustworthy, *unless* he is very young').

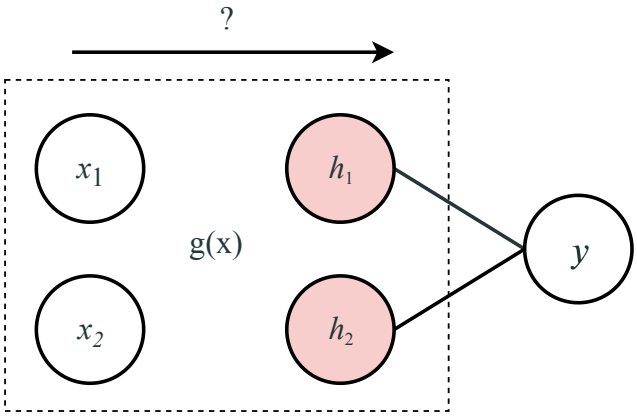


Figure 1: How can we define the intermediate computations?

Feedforward neural networks

Fully-connected layers

Can we layer two linear models?

$$\mathbf{h} = \mathbf{W}\mathbf{x}\,, \tag{2}$$
$$f(\mathbf{h}) = \mathbf{w}^\top \mathbf{h}\,. \tag{3}$$

We could, but this is equivalent to a single linear model:

$$f(\mathbf{x}) = \left(\mathbf{w}^\top \mathbf{W}\right)\mathbf{x}\,. \tag{4}$$

We need some way of separating the two linear operations.

We can avoid the 'collapse' of the two linear models by interleaving them with some element-wise nonlinearity ϕ:

$$\mathbf{h} = \phi(\mathbf{W}\mathbf{x}), \tag{5}$$

$$f(\mathbf{h}) = \mathbf{w}^\top \mathbf{h}. \tag{6}$$

This is the prototype of a **feedforward** neural network (NN), sometimes known as a **multilayer perceptron**.

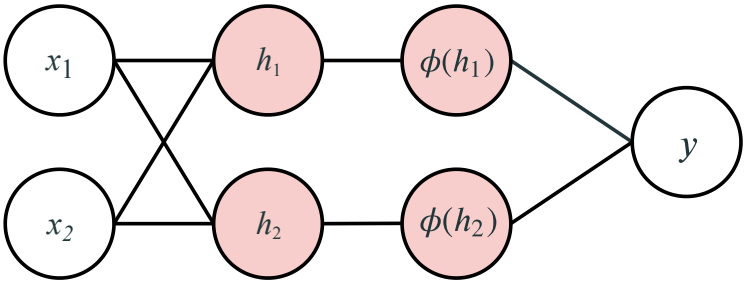


Figure 2: Visualization of a simple feedforward NN. In general, we will never show the nonlinearities explicitly in the graphs from now on.

Feedforward neural networks

Training and complexity considerations

Training of the network can be proceed similarly to a linear model. For example, given a dataset $\{\mathbf{x}_i, y_i\}_{i=1}^n$ for regression, we can minimize the LS loss:

$$\mathbf{W}^*, \mathbf{w}^* = \arg\min \frac{1}{n} \sum_{i=1}^{n} \left(y_i - f(\mathbf{x}_i)\right)^2. \tag{7}$$

For classification, we can wrap the output in a sigmoid and minimize a cross-entropy loss.

Theorem 1.1: Cybenko (1989) - Hornik (1991) - Leshno (1993)

Let $h(\mathbf{x})$ be a continuous function defined on a compact subset $S \subset \mathbb{R}^d$ and $\varepsilon > 0$. For a sufficiently large p, there exists an $f(\mathbf{x})$ as in Eq. (6)-(7) with p hidden units such that:

$$|h(\mathbf{x}) - f(\mathbf{x})| < \varepsilon, \ \forall \mathbf{x} \in S\,. \tag{8}$$

This holds for any non-constant, bounded, continuous ϕ.

This is a **universal approximation** theorem. It does not tell us about the *feasibility* for a given problem (e.g., how large should p be?).

Because the NN can be highly non-convex, its optimization problem has multiple local minimum and/or saddle points.

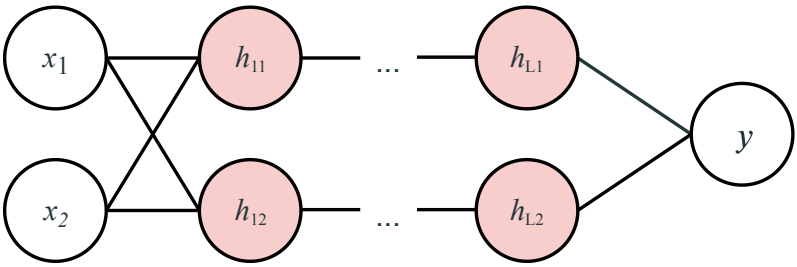
In fact, training a neural network is **NP-hard**, even for very simple architectures, and highly dependent on a good initialization.

Finding the global optimum requires running GD from almost *everywhere*: this is similar to an exhaustive search.

Blum, A. and Rivest, R.L., 1989. Training a 3-node neural network is NP-complete. In Advances in neural information processing systems (pp. 494-501).

Nothing prevents us from adding *additional* 'hidden' (intermediate) layers:

$$f(\mathbf{x}) = \mathbf{w}^T \cdot \overbrace{\phi\big(\mathbf{Z} \cdot \underbrace{\phi\,(\mathbf{W} \cdot \mathbf{x})}_{\mathbf{h}^1} + \mathbf{c}\big)}^{\mathbf{h}^2} \tag{9}$$



Differently from a linear model, a NN has several design choices that we have freedom on:

- The nonlinearity (sometimes called the **activation function**);
- The dimensionality of the hidden layers;
- Several others that we will introduce in the next lectures.

We call these **hyper-parameters** to differentiate them from parameters (weights) to be trained via GD.

Choosing the correct set of hyper-parameters is called the **model selection** / **hyper-parameter optimization** problem.

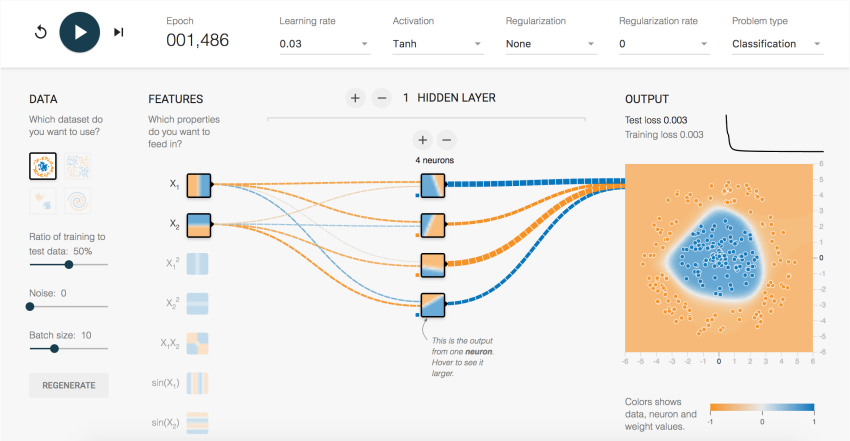


Figure 3: https://playground.tensorflow.org/.

Training of the networks

Stochastic optimization

Consider the steps needed for a single iteration of GD:

1. Computing the output of the NN $f(\mathbf{x})$ on *all* examples;
2. Computing the gradient of the cost function with respect to the weights.

Both these operations scale *linearly* in the number of examples, which is infeasible when we have 10^5 or even more examples.

In practice, to train NNs we use **stochastic** versions of GD, that approximate the real gradient from smaller amounts of data.

The key observation is that the training problem in NNs is an expectation with respect to all data ($\boldsymbol{\theta}$ is the set of weights):

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - f(\mathbf{x}_i) \right)^2 . \tag{10}$$

To limit the complexity, we use only a **mini-batch** $\mathcal{B}$ of M examples from the full dataset:

$$J(\boldsymbol{\theta}) \approx \widetilde{J}(\boldsymbol{\theta}) = \frac{1}{M} \sum_{i \in \mathcal{B}} \left(y_i - f(\mathbf{x}_i) \right)^2 . \tag{11}$$

We can use the gradient from the approximated function to perform an iteration of GD.

The previous algorithm is called **stochastic gradient descent** (SGD).

The computational complexity of an iteration of SGD is fixed with respect to M (the **batch size**) and does not depend on the size of the dataset.

Because we assumed that samples are i.i.d., we can prove SGD also converges to a stationary point in average, albeit *with noisy steps*.

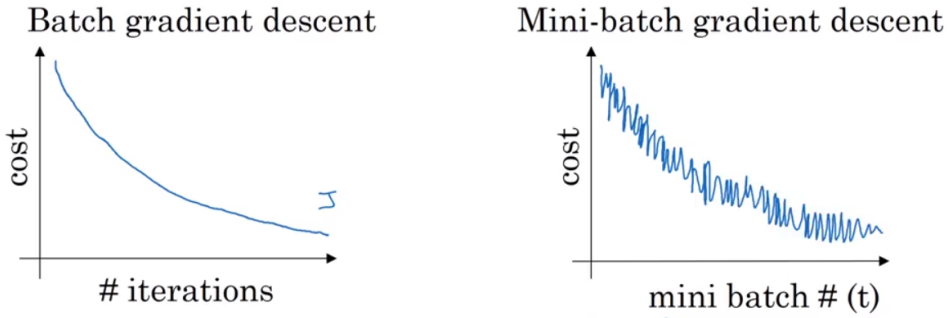


Figure 4: SGD will converge on average to a stationary point, although in an apparently noisy fashion (Source: EngMRK).

In order to extract mini-batches from our dataset, we can apply a simple procedure:

1. Shuffle the full dataset;
2. Split the dataset into blocks of M elements and process them sequentially;
3. After the last block, return to point (1) and iterate.

A full pass over the dataset is called an **epoch**.

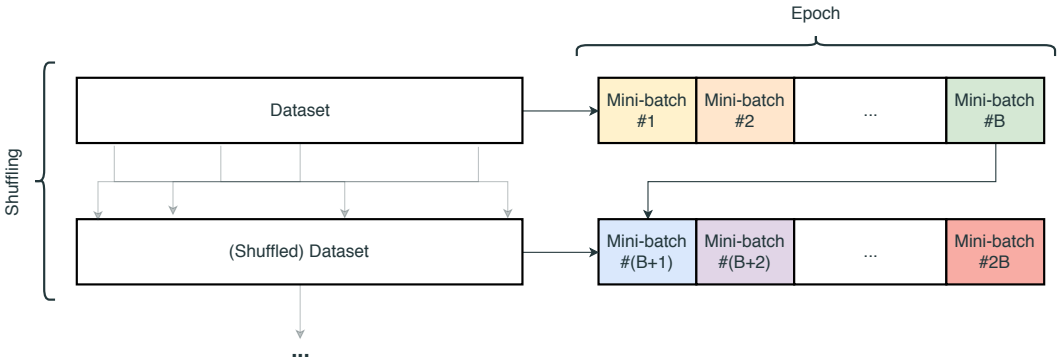


Figure 5: Dividing the optimization process into epochs is also helpful to define metrics and stopping criteria.

The last choice is the size of the mini-batch:

- Smaller mini-batches are faster, but the network might require more iterations to converge because the gradient is noisier.
- Larger mini-batches provide a more reliable estimation of the gradient, but are slower.

In general, it is typical to choose power-of-two sizes (32, 64, 128, ...) depending on the hardware configuration and the total memory available.

In the limit $M = 1$ we obtain an **online** (streaming) optimization.

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:

$$\underset{(o,d)}{\partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})} \quad \underset{(o,p)}{\partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})} \,. \tag{12}$$

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**:

$$f(\mathbf{x}, \{\mathbf{W}, \mathbf{b}\}) = \mathbf{W}\mathbf{x} + \mathbf{b}\,. \tag{13}$$

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

$$f(\mathbf{x}, \{\}) = \phi(\mathbf{x})\,. \tag{14}$$

- 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
```

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):

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

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

$$\{\partial_{\mathbf{w}_i} y\} \quad i = 1, \dots, l \,. \tag{15}$$

Remember the chain rule for Jacobians:

$$\partial [f \circ g] = \partial f \ \circ \ \partial g \,. \tag{16}$$

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).

We consider 3 layers as follows:

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

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 easily be 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!

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

$$\underset{(p_1)}{\nabla_{\mathbf{w}_1} y} = \underset{(p_1,o_1)}{\partial^\top_{\mathbf{w}_1} \mathbf{h}_1} \cdot \underset{(o_1,o_2)}{\partial^\top_{\mathbf{h}_1} \mathbf{h}_2} \cdot \underset{(o_2,o_3)}{\partial^\top_{\mathbf{h}_2} \mathbf{h}_3} \cdot \underset{(o_3)}{\mathbf{1}} \; . \tag{17}$$

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! The following 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:

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

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

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

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

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 do not scale in the number of parameters. 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`

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:

$$\begin{aligned}\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) &= \partial_{\mathbf{x}}^{\top} f(\mathbf{x}, \mathbf{w}) \cdot \mathbf{v}\,, \\ \text{vjp}_{\mathbf{w}}(f, \mathbf{v}) &= \partial_{\mathbf{w}}^{\top} f(\mathbf{x}, \mathbf{w}) \cdot \mathbf{v}\,.\end{aligned}$$

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.

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:

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

Practically, any framework defines a set of differentiable operations f of which it knows how to compute VJPs. New operations can be added by defining custom VJPs!

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

$$f(\mathbf{x}, \{\}) = \phi(\mathbf{x}) \,. \tag{18}$$

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

$$[\partial f(\mathbf{x}, \{\})]_{i,i} = \phi'(x_i) \,. \tag{19}$$

The JVP is instead:

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \phi'(\mathbf{x}) \odot \mathbf{v} \,. \tag{20}$$

To store all intermediate operations in TensorFlow, we can use a `tf.GradientTa` 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)
```

By default, a tensor is watched if it is wrapped in a `tf.Variable`, if it originates from another variable (inside the tape), or if we call `tape.watch(x)`.

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`.

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:

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \phi'(\mathbf{x}) \odot \mathbf{v}\,. \tag{21}$$

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**).

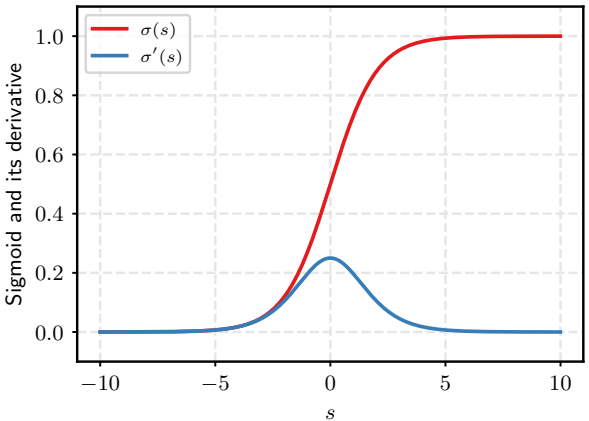


Figure 6: The sigmoid function $\sigma(\cdot)$ is a poor choice as activation function (for deep networks), because its derivative is bounded in $[0, 0.25]$.

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

$$\phi(s) = \max(0, s) \ . \tag{22}$$

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.

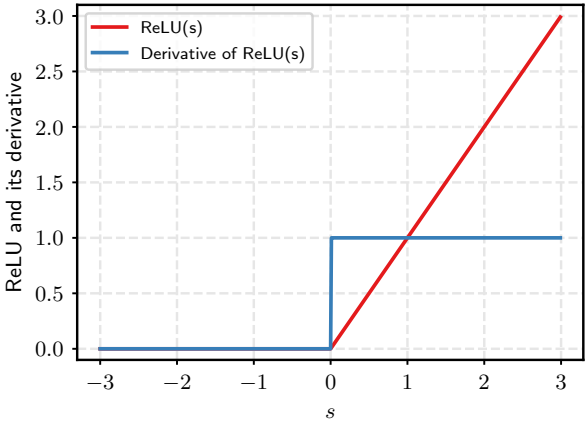


Figure 7: A plot of ReLU and its derivative.

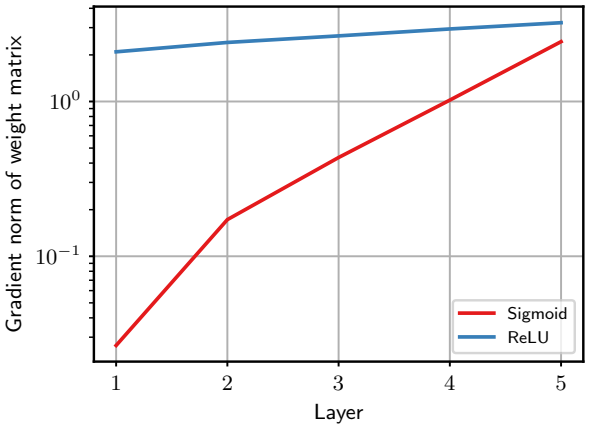


Figure 8: We initialize a NN with 5 hidden layers. Weights are sampled from the uniform distribution on $[-0.5, 0.5]$. We show the norm of a typical gradient (cross-entropy on a few examples) with sigmoid and ReLU activation functions.

- (**Mandatory**) Chapter 4 from the book.
- (**Optional**): 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.

For an interesting overview of how subderivatives interact with automatic differentiation, see:

- **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`).