



Automatic differentiation

Simone Scardapane

Neural Networks 2023/2024

October 16, 2023



SAPIENZA
UNIVERSITÀ DI ROMA

1 Automatic differentiation

Forward mode

Autodiff in practice

Choosing an activation function

Composition of parametric functions

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:

$$\begin{matrix} \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w}) & \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w}) \\ (o, d) & (o, p) \end{matrix} . \quad (1)$$

Vectors and tensors

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

```
1 x = torch.randn((3, 4))
2 w = torch.randn(((4, 5)), requires_grad=True)
3 torch.autograd.functional.jacobian(lambda w: x @ w, w).shape # Print: torch.Size([3, 5, 4, 5])
```

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

Vectors and tensors (2)

The previous code, in fact, is equivalent to:

```
1 x = torch.randn((12))
2 w = torch.randn((20), requires_grad=True)
3 torch.autograd.functional.jacobian(lambda w:
4     (x.reshape(3, 4) @ w.reshape(4, 5)).reshape(-1, ), w).shape # Print: torch.Size([15, 20])
```

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

Examples of primitives

- **Fully-connected layers:**

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

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

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

- Quite a few more to come...

A note on implementation

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

```
1 def f(W, x):  
2     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:

```
1 class F(Layer):  
2     def __init__(self, d, o):  
3         self.W = torch.Variable(torch.randn((d, o)))  
4     def call(self, x):  
5         return x @ self.W
```

Purification

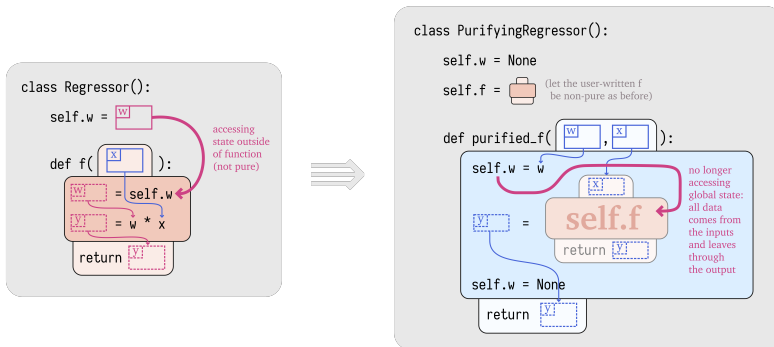


Figure 1: We can move from an OOP implementation to a functional one by a process called *purification*: From PyTorch to JAX: towards neural net frameworks that purify stateful code.

Purification in practice

In recent releases, it is possible to purify a PyTorch model with `torch.func.functional_call`:

```
1 # OOP variant
2 y = model(x)
3
4 # Functional variant
5 params = dict(model.named_parameters())
6 model_fcn = lambda w, x: torch.func.functional_call(model, w, x)
7 y = model_fcn(params, x)
```

```
1 # Gradient in the functional way
2 grad_fcn = torch.func.grad(lambda w, x: model_fcn(w, x).mean())
3 grad_fcn(params, x)
```

Composing blocks

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:

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

Chain rule for Jacobians

Remember the chain rule for Jacobians:

$$\partial [f \circ g] = \partial f \circ \partial g . \quad (5)$$

We can interpret the chain rule as follows:

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

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

A practical example

Consider 3 layers as follows:

$$\underset{(o_1)}{\mathbf{h}_1} = f_1(\underset{(d)}{\mathbf{x}}, \underset{(p_1)}{\mathbf{w}_1})$$

$$\underset{(o_2)}{\mathbf{h}_2} = f_2(\underset{(o_1)}{\mathbf{h}_1}, \underset{(p_2)}{\mathbf{w}_2})$$

$$\underset{(o_3)}{\mathbf{h}_3} = f_3(\underset{(o_2)}{\mathbf{h}_2}, \underset{(p_3)}{\mathbf{w}_3})$$

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

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

Forward-mode autodiff (general)

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:

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

Forward-mode autodiff (step 1)

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

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

Forward-mode autodiff (step 2)

After our second operation, we need to update the gradient with respect to \mathbf{w}_1 , and initialize the one with respect to \mathbf{w}_2 :

Original instruction

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

Additional instructions

$$\begin{aligned} \widehat{\mathbf{W}}_1 &= \left[\partial_{\mathbf{h}_1} \mathbf{h}_2 \right] \cdot \widehat{\mathbf{W}}_1, \\ &\quad \substack{(o_2, p_1) & (o_2, o_1) & (o_1, p_1)} \\ \widehat{\mathbf{W}}_2 &= \partial_{\mathbf{w}_2} \mathbf{h}_2. \end{aligned}$$

Forward-mode autodiff (step 3)

At this point, we keep iterating our procedure:

Original instruction

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

Additional instructions

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

$$\widehat{\mathbf{W}}_2 = \left[\partial_{\mathbf{h}_2} \mathbf{h}_3 \right] \cdot \widehat{\mathbf{W}}_2,$$

$$\widehat{\mathbf{W}}_3 = \partial_{\mathbf{w}_3} \mathbf{h}_3.$$

Forward-mode autodiff (step 4)

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

Original instruction

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

Additional instructions

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

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

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

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

Pros and cons of forward-mode autodiff



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, \dots will have a factor n inside, and the multiplication will scale quadratically in n .

An insight into a better solution

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

$$\nabla_{\mathbf{w}_1} y = \partial_{\mathbf{w}_1}^\top \mathbf{h}_1 \cdot \partial_{\mathbf{h}_1}^\top \mathbf{h}_2 \cdot \partial_{\mathbf{h}_2}^\top \mathbf{h}_3 \cdot \mathbf{1} . \quad (6)$$

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.

Reverse-mode autodiff (informal)

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

$$\nabla_{\mathbf{w}_i} y = \left[\partial_{\mathbf{w}_i}^\top \mathbf{h}_i \right] \cdot \tilde{\mathbf{h}}$$

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

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

Reverse-mode in action

Let us look at the operations required in our example:

$$\begin{aligned}\tilde{\mathbf{h}} &= \mathbf{1} \\ \nabla_{\mathbf{w}_3} y &= \left[\partial_{\mathbf{w}_3}^\top \mathbf{h}_3 \right] \cdot \tilde{\mathbf{h}}, \quad \tilde{\mathbf{h}} = \left[\partial_{\mathbf{h}_2}^\top \mathbf{h}_3 \right] \tilde{\mathbf{h}}, \\ \nabla_{\mathbf{w}_2} y &= \left[\partial_{\mathbf{w}_2}^\top \mathbf{h}_2 \right] \cdot \tilde{\mathbf{h}}, \quad \tilde{\mathbf{h}} = \left[\partial_{\mathbf{h}_1}^\top \mathbf{h}_2 \right] \tilde{\mathbf{h}}, \\ \nabla_{\mathbf{w}_1} y &= \left[\partial_{\mathbf{w}_1}^\top \mathbf{h}_1 \right] \cdot \tilde{\mathbf{h}}.\end{aligned}$$

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

Pros and cons of reverse-mode autodiff

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.

Backpropagation

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>

Some historical key moments

- 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 first 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-ismmp/52_griewank-andreas-b.pdf

1 Automatic differentiation

Forward mode

Autodiff in practice

Choosing an activation function

Vector-Jacobian products

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

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

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

We call these **vector-Jacobian products**. They can be significantly easier to compute 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.

Backpropagating through a linear projection

Let us consider for example:

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

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?

VJPs of a linear projection

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.

Element-wise operations

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

$$f(\mathbf{x}, \{\}) = \phi(\mathbf{x}) . \quad (7)$$



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

$$[\partial f(\mathbf{x}, \{\})]_{i,i} = \phi'(x_i) . \quad (8)$$

The JVP is instead:

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \phi'(\mathbf{x}) \odot \mathbf{v} . \quad (9)$$

Autodiff in TensorFlow

You can disable storage of intermediate values in PyTorch by running code in inference mode:

```
1 x = torch.randn((3, 4), requires_grad=True)
2 with torch.inference_mode():
3     y = x.sum()
4     y.grad_fn # None
```

See [Alice's Adventures in a Differentiable Wonderland](#) for more details and an example of reimplementing the core autodiff tool.

Support for forward-mode autodiff in PyTorch is in Beta: [Forward-Mode Automatic Differentiation](#)

Gradient checkpointing

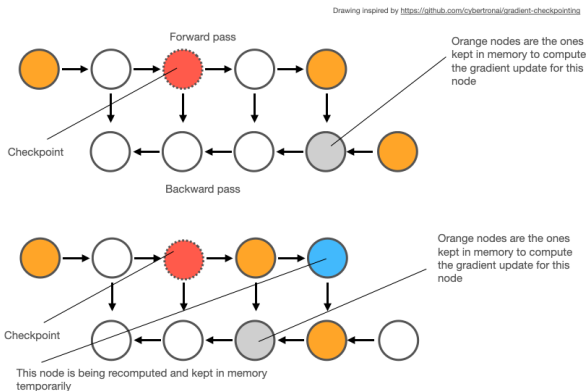


Figure 2: Gradient checkpointing improves the memory footprint of backpropagation by only storing a few elements of the computational graph, recomputing all intermediate values during the backward pass. https://github.com/rasbt/deeplearning-models/blob/master/pytorch_ipynb/mechanics/gradient-checkpointing-nin.ipynb

1 Automatic differentiation

Forward mode

Autodiff in practice

Choosing an activation function

How do we choose a nonlinearity?

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

Remember that in this case:

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \phi'(\mathbf{x}) \odot \mathbf{v} . \quad (10)$$

Vanishing and exploding gradients

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**).

Derivative of the sigmoid

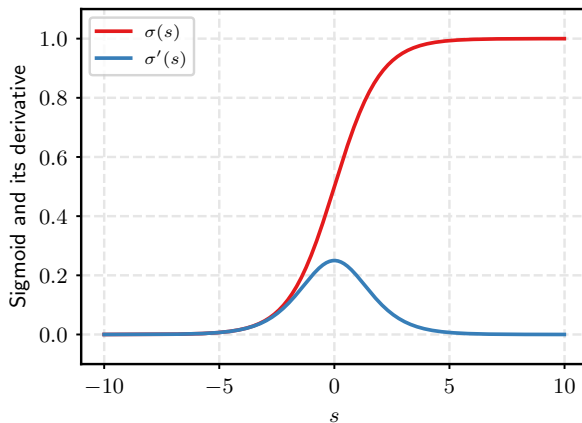


Figure 3: The sigmoid function $\sigma(\cdot)$ is a poor choice as activation function (for deep networks), because its derivative is bounded in $[0, 0.25]$.

The rectified linear unit (ReLU)

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

$$\phi(s) = \max(0, s) . \quad (11)$$

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 $f : \mathbb{R} \rightarrow \mathbb{R}$ at a point x is a point g such that:

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

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>).

ReLU and its derivative

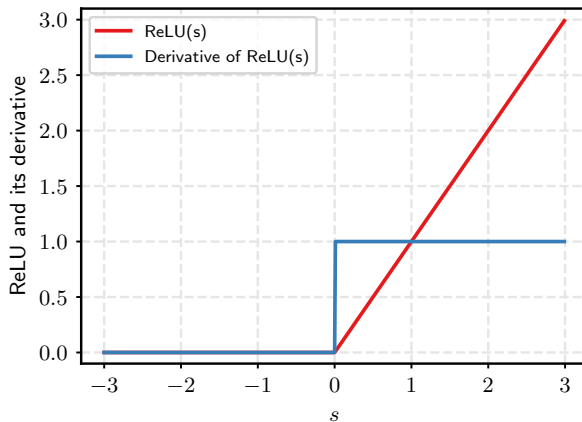


Figure 4: A plot of ReLU and its derivative.

An example of vanishing gradient

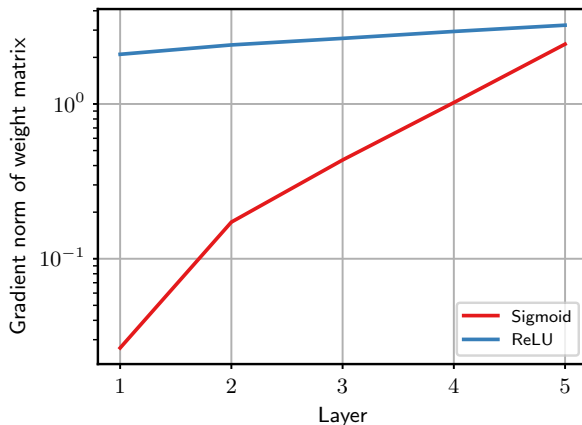


Figure 5: We initialize a NN with 5 hidden layers. Weights are sampled from the uniform distribution on $[-0.5, 0.5]$. We show the norm of a typical gradient (cross-entropy on a few examples) with sigmoid and ReLU activation functions.

2 Additional topics

Variants of the ReLU

Stochastic optimization

Variants of the ReLU

- ① **LeakyReLU** replaces the negative quadrant of ReLU with a small slope (e.g., $\alpha = 0.01$):

$$\text{LeakyReLU}(s) = \begin{cases} s & \text{if } s > 0 \\ \alpha s & \text{otherwise} \end{cases} \quad (12)$$

- ② If you want a smoother version of the ReLU, you can use the **exponential linear unit** (ELU):

$$\text{ELU}(s) = \begin{cases} s & \text{if } s > 0 \\ \alpha(\exp s - 1) & \text{otherwise} \end{cases} \quad (13)$$

Variants of the ReLU (2)

Viewing the ReLU as a kind of *gating* function $\text{ReLU}(s) = s1_{x \geq 0}$, we can obtain smoother variants as:

$$\phi(s) = s\psi(s), \quad (14)$$

where:

- If $\psi(s)$ is the standard Gaussian cumulative distribution function, we obtain the **Gaussian Exponential Linear Unit (GELU)**.
- If $\psi(s) = \sigma(s)$ we obtain the **Sigmoid linear unit (SiLU)** or **Swish**.

Comparison of activation functions

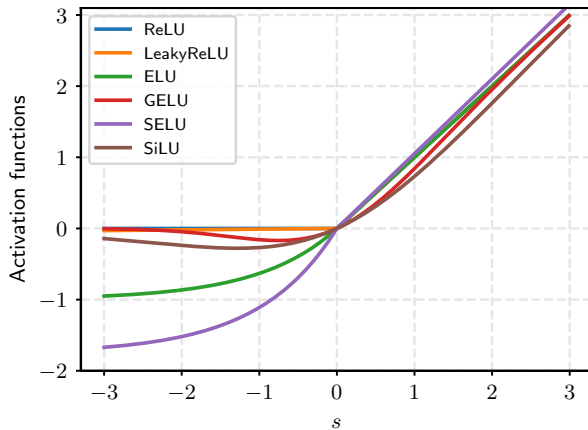


Figure 6: Common activation functions plotted side-by-side.

Trainable activation function

Activation functions can also have trainable parameters that vary for each unit. For example, the LeakyReLU with trainable α becomes the **parametric ReLU** (PReLU). Or, for a more convoluted example:

$$\text{Trainable-Swish}(s) = \sigma(as + b)(cs + d), \quad (15)$$

with 4 trainable parameters per unit. Variants of this have become popular with the recent LLaMA class of models.

Shazeer, N., 2020. **GLU variants improve transformer**. *arXiv preprint arXiv:2002.05202*.

2 Additional topics

Variants of the ReLU

Stochastic optimization

Handling large datasets

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 unfeasible 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 (\cdot is the set of weights):

$$J(\cdot) = \frac{1}{n} \sum_{i=1}^n (y_i - f(\mathbf{x}_i))^2. \quad (16)$$

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

$$J(\cdot) \approx \tilde{J}(\cdot) = \frac{1}{M} \sum_{i \in \mathcal{B}} (y_i - f(\mathbf{x}_i))^2. \quad (17)$$



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

Characteristics of stochastic 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*.

Visualization of the loss

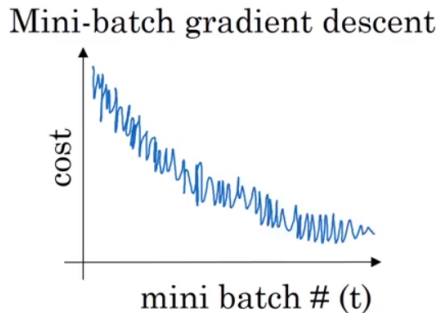
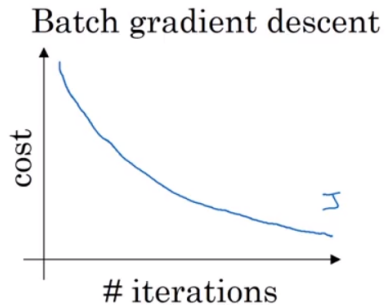


Figure 7: SGD will converge on average to a stationary point, although in an apparently noisy fashion (Source: EngMRK).

Extracting batches from the data

Instead of randomly sampling batches from the training dataset at each iteration, we generally apply the following procedure:

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

A full pass over the dataset is called an **epoch**. This is efficient because the majority of the time we deal with data stored *contiguously* in memory. In practice, even the shuffling operation can be approximated.

Extracting batches from the data

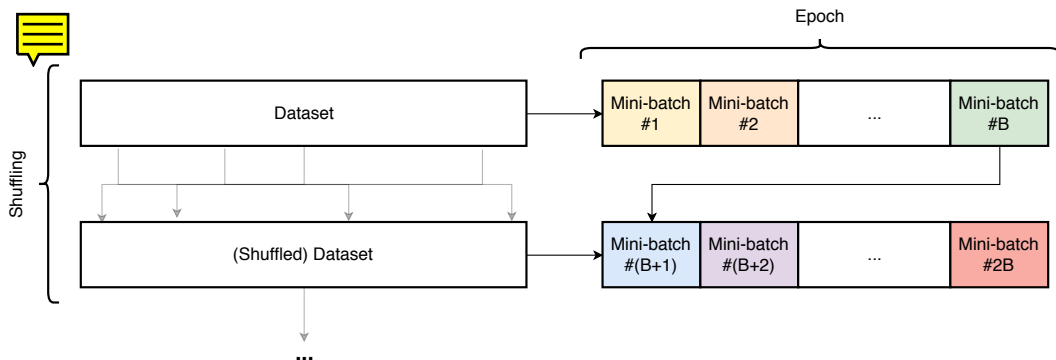


Figure 8: Dividing the optimization process into epochs is also helpful to define metrics and stopping criteria.

Choosing the batch size

Running SGD requires selecting the size of the mini-batch, which is an additional hyper-parameter:

- 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.

- **Dive into Deep Learning:** Chapter 5.
- 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, ...).