Build a simple autograd

Deep learning `from scratch`

Universitat Politècnica de Catalunya (UPC)

Author Carlos Pérez *
Tutor Ramon Sangüesa †

January 1, 2024

* carlos.perez.ruiz@estudiantat.upc.edu
† ramon.sanguesa.i@upc.edu

Contents

Contents ii

1 Introduction 1
- 1.1 Learning Process . . . 3
- 1.2 Feed Forward Networks . . . 5
- 1.3 Everything is a Computational Graph . . . 6
- 1.4 Automatic Differentiation Library . . . 9

2 Autodiff Library 11
- 2.1 Multi Layer Perceptron . . . 13
  - 2.1.1 Tensor Broadcasting . . . 15
  - 2.1.2 Output layer . . . 17
- 2.2 Model . . . 19

List of Terms 22

Introduction 1

1.1 Learning Process . . . 3
1.2 Feed Forward Networks 5
1.3 Everything is a Computational Graph 6
1.4 Automatic Differentiation Library 9

Computers have been able to defeat humans in a myriad of abstract tasks for quite a long time now. However, it is only at this moment that they are beginning to reach a level where they match or even surpass human beings in tasks such as object recognition or the processing of human speech; tasks that we humans perform effortlessly from a very early age. A person's everyday life requires an immense amount of knowledge about the world, and much of this knowledge is subjective and intuitive, and therefore difficult to articulate in a formal way. This knowledge is what we call *experience*.

One of the early ways to acquire this knowledge was to hard-code it in formal languages. A computer could reason about statements in these formal languages using logical inference rules. This is known as the **knowledge base** approach to artificial intelligence.

The difficulties faced by systems relying on hard-coded knowledge suggested that AI systems needed the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as **machine learning**.

An example of a machine learning problem is how to use a finite sample of randomly selected documents, each labeled with a topic, to accurately predict the topic of unseen documents. Clearly, the larger is the sample, the easier is the task [1]. However, the algorithm used to predict the topic of unseen documents cannot influence the way those features are represented. This means that the algorithm is tied to the representation of the data and also its performance is heavily reliant on how this data is presented and structured.

1: Each representation or information of the document is known as a **feature**, **attribute** or **covariate**.

Another way to acquire *experience* is by designing the right set of features to extract for that task, then providing these features to a simple machine learning algorithm. One solution to this problem is to use machine learning to discover not only the mapping from representation to output but also the representation itself. This approach is known as **representation learning**, and learned representations often result in much better performance than can be obtained with hand-designed representations.

When designing features or algorithms for learning features, the goal is usually to separate the factors of variation that explain the observed data. However, they may exist either as unobserved objects or unobserved forces in the physical world that affect observable quantities. A major source of difficulty in many real-world artificial intelligence

applications is that many of the factors of variation influence every single piece of data we are able to observe.

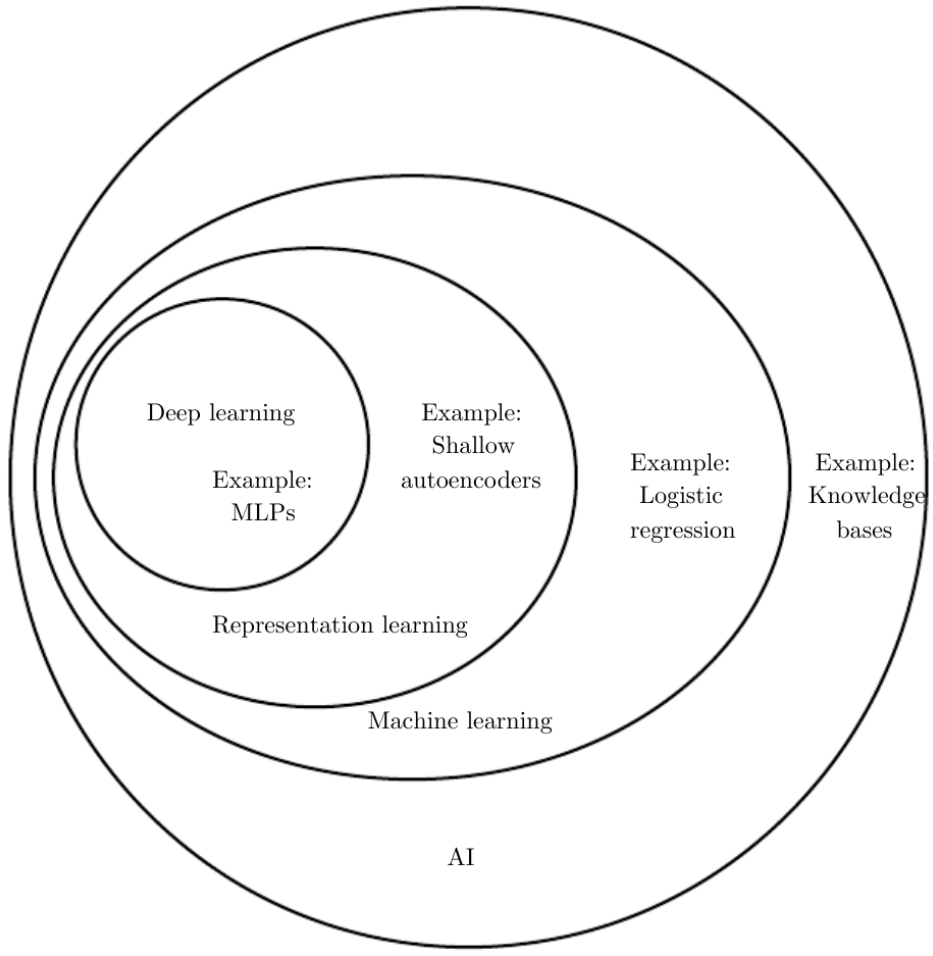


Figure 1.1: Venn diagram showing how deep learning is a kind of representation learning, which is in turn a kind of machine learning.

When it is nearly as difficult to obtain a representation as to solve the original problem, representation learning does not, at first glance, seem to help us.

Deep Learning (DL) solves this central problem in representation learning by introducing representations that are expressed in terms of other, simpler representations. DL is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones.

1.1 Learning Process

It all begins with one or several **examples** to enable learning or evaluation. These examples stem from a generator G of random data observations generated **i.i.d** from a distribution P_x. Each example is represented as an attribute vector $\mathbf{x} \in \mathcal{X}$. For instance, in the scenario of an algorithm related to climatological data, $\mathbf{x}$ might encompass variables like temperature, humidity, and so forth.

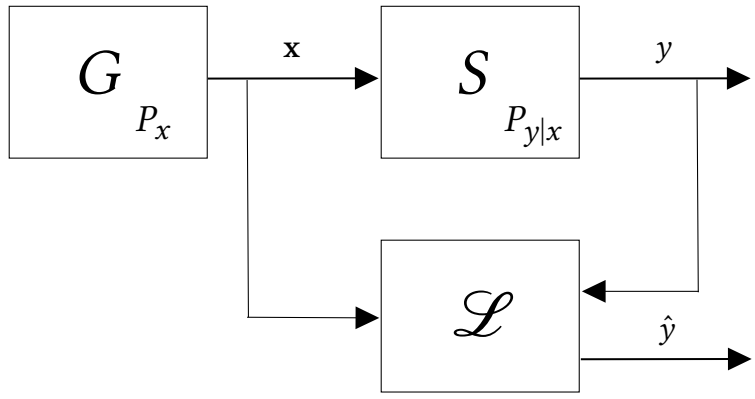


Figure 1.2: x is generated **i.i.d** from an unknown distribution P_x which is fed into the model $\mathscr{L}$ that tries to learn the distribution $P_{y|x}$ observing y from the supervisor S.

Each attribute vector is associated to a **label** $y \in \mathcal{Y}$ from a probability distribution $P_{y|x}$ defined by the supervisor S [2]. A set of samples D_N, henceforth simply referred as a **dataset**, is defined as a labeled set of input-output pairs with N examples.

$$D_N := \{ (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N) \} \tag{1.1}$$

2: The relation between $\mathcal{X}$ and $\mathcal{Y}$ is stochastic since the supervisor is not a function that always maps an input vector $\mathbf{x}$ to a particular y. For example, consider the relationship between height and weight of a person.

Where

(i) $\mathbf{x}_i \in \mathcal{X}$

(ii) $y_i \in \mathcal{Y}$

Each attribute vector $\mathbf{x}$ is directly fed into the **learning algorithm** $\mathscr{L}$, which can execute a collection of functions (**models**) $f_\theta \in \mathcal{F}$, where $\theta \in \Theta$ represents a **hyperparameter** space.

$$\mathcal{F}_i = \{ f_\theta(\mathbf{x}) := f(\mathbf{x}; \theta) \mid \mathbf{x} \in \mathcal{X}, \theta \in \Theta \} \tag{1.2}$$

The algorithm's purpose is to select the optimal function f_θ that approximates the distribution $P_{y|x}$ defined by S.

The selection of f_θ is guided with a loss/error function l, such that:

$$l \ : \ \mathcal{Y} \times \hat{\mathcal{Y}} \longrightarrow \mathbb{R}_0^+ \tag{1.3}$$

Where

(i) $l(y, \hat{y}) \geq 0$

(ii) $y = \hat{y} \rightarrow l(y, \hat{y}) = 0$

(iii) l is monotonically decreasing with $|y - \hat{y}|$

Hence, the goal is to minimize the error established by the loss function. However the loss function is not applied to the whole dataset D_N. A distinction must be made between types of samples, because each one has a different purpose in the learning process, and the loss function may be used or not.

The dataset D_N is partitioned into **training**, **validation**, and **test** subsets. The purpose of the training set is to train the algorithm (selecting f_θ), the validation set is used to fine-tune the hyperparameters θ in terms of l, and the test set is employed to evaluate the performance of the learning algorithm.

The size of each partition depends on various considerations. For instance, when the data set D_N is small, the training partition is often the largest.

The aforementioned guidelines establish the rules of the game to achieve **generalization**, as machine learning is fundamentally rooted in generalization. Generalization is the term used to describe the model's decision-making process within the function set $\mathcal{F}_i$. When dealing with a *complex* or diverse set $\mathcal{F}_i$, the algorithm might choose a function or predictor that perfectly matches the training sample, one that makes no errors on it. Conversely, in cases of a less complex function set, some errors on the training sample could be unavoidable.

Which approach yields superior generalization? How do we define the complexity of a set of hypotheses? In general, the best predictor on the training sample may not be the best overall. A predictor chosen from a very complex set $\mathcal{F}_i$ can essentially memorize the data, but generalization is distinct from the memorization of the training labels. The **trade-off** between the sample size and complexity plays a critical role in generalization. When the sample size is relatively small, choosing from a too complex $\mathcal{F}_i$ may lead to poor generalization, which is also known as **overfitting**. On the other hand, with a too simple $\mathcal{F}_i$ it may not be possible to achieve a sufficient performance, which is known as **underfitting**.

1.2 Feed Forward Networks

Feedforward Neural Networks (FNNs) represent the classic deep learning models. As a machine learning model, its goal is to approximate a function $f^* \in \mathscr{F}_i$ [3].

3: f^* means a gold function $f := f_\theta$, the best one to approximate in a specific problem.

These models are called feedforward because information flows in one direction, from the first function that evaluates $\mathbf{x}$, through the intermediate computations used to define f, and finally to the output $\hat{\mathbf{y}}$. No **feedback** connections are used, in that case we would be talking about a Recurrent Neural Network (RNN).

A FNN is typically represented by composing together similar functions, recursively defined and combined. The word architecture refers to the overall structure of the network: how many units it should has and how these units are connected to each other. Most neural networks are organized into groups of units called layers.

Multi Layer Perceptrons (MLPs) represent the canonical feedforward network architecture, which derives from the works of McCulloch & Pitts and Frank Rosenblatt. Most FNNs architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it.

The design of f is loosely guided by neuroscientific observations about the function that biological neurons compute. However, the modern neural networks are not entirely guided by biological principles, they are influenced by engineering and mathematics more than biology.

$$f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x}))) \tag{1.4}$$

These chain structures are the most commonly used structure of neural networks. In this case, f_1 is called the first layer or the input layer, the last layer is called the output layer, and every layer in between is considered a hidden layer. The length of the chain gives the depth of the model, and is from this terminology that the name *Deep Learning* arises.

To comprehend FNN, it's helpful to start by examining linear models and finding ways to overcome their limitations. Linear models, including logistic regression and linear regression, are appealing due to their efficient and reliable fitting methods, either through closed-form solutions or convex optimization. However, these models come with a notable drawback — their capacity is restricted to linear functions, which means they cannot capture interactions between any two input variables.

The common approach to represent nonlinear functions of $\mathbf{x}$ is introducing **basis functions** ϕ, so that we can apply a linear model not to $\mathbf{x}$ itself but to a transformed input $\phi(\mathbf{x})$. We can think of ϕ as providing a set of features describing $\mathbf{x}$, or as providing a new representation for $\mathbf{x}$. Now the problem is how to choose the mapping ϕ.

The strategy of deep learning is to learn ϕ. In this approach, we have a model $h = f(\mathbf{x}; \theta, \mathbf{w}) = \phi(\mathbf{x}; \theta)^\top \mathbf{w}$. We now have parameters θ and ϕ from a broad class of functions, and parameters $\mathbf{w}$ that map from $\phi(\mathbf{x})$ to the desired output. h can be understood as a FNN with ϕ defining a hidden layer. As opposed to Generalized Linear Models (GLMs), this approach gives up on the convexity of the training problem, leading to some concessions on the optimization algorithm to find θ. This approach can capture the benefit of GLMs by being highly generic — we do so by using a very broad family $\phi(\mathbf{x}; \theta)$.

One option is to use a very generic ϕ, such as an infinite-dimensional ϕ that is implicitly used by kernel machines based on the RBF kernel, like Support Vector Machines SVD. Another option is to manually engineer ϕ, but this approach requires lots of experience for each separate task, and with little transfer between domains.

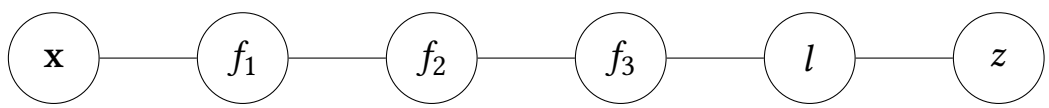
Key Takeaways

The general principle of improving models by learning features extends beyond the Feedforward Neural Networks (FNNs). FNNs are the application of this principle to learning deterministic mappings from $\mathbf{x}$ to y that lack feedback connections.

Training a FNN requires making many of the same design decisions as are necessary for a linear model: choosing the optimizer, the cost function, and the form of the output units. We must also design the architecture of the network, including how many layers the network should contain, how these layers should be connected to each other, and how many units should constitute each layer. Finally, learning in deep neural networks requires computing the gradients of complicated functions through back-propagation algorithm.

1.3 Everything is a Computational Graph

As noted in Equation (1.4), a Feedforward Neural Network (FNN) can be defined as nested functions, but a more convenient way for this lab is to visualize each definition or computation of a neural network as a Directed Acyclic Graph (DAG). To formalize an expression into a graph, we also need to introduce the idea of an operation. An operation is a simple function of one or more variables. Functions more complicated than the operations already defined can be described by composing many operations together [4].



4: Take for instance the square root, or every N-th root. These operations can be defined with exponents, e.g. $a^{1/2}, a^{1/3}, \ldots, a^{1/n}$, so there's no need to define an N-th root in a computational graph language, unless you really need so.

Suppose we define a FNN of three layers like in Equation (1.4), and we have to minimize a loss function l through gradient descent. Every partial derivative must be computed through chain rule.

$$\frac{\partial l}{\partial f_3} = l'(f(\mathbf{x}))f_3'(f_2(f_1(\mathbf{x}))) \tag{1.5}$$

$$\frac{\partial l}{\partial f_2} = l'(f(\mathbf{x}))f_3'(f_2(f_1(\mathbf{x})))f_2'(f_1(\mathbf{x})) \tag{1.6}$$

$$\frac{\partial l}{\partial f_1} = l'(f(\mathbf{x}))f_3'(f_2(f_1(\mathbf{x})))f_2'(f_1(\mathbf{x}))f_1'(\mathbf{x}) \tag{1.7}$$

To illustrate, let's define every f_i as a Multi Layer Perceptron layer.

$$\mathbf{h_1} = g\left(\mathbf{W}_1^\top \mathbf{x} + \mathbf{b}_1\right) \tag{1.8}$$

$$\mathbf{h_2} = g\left(\mathbf{W}_2^\top \mathbf{h_1} + \mathbf{b}_2\right) \tag{1.9}$$

$$\mathbf{h_3} = g\left(\mathbf{W}_3^\top \mathbf{h_2} + \mathbf{b}_3\right) \tag{1.10}$$

$$z = \frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{h}_{3i} - \hat{\mathbf{y}}_i\right)^2 \tag{1.11}$$

Where

(i) g is the activation function.

(ii) $\mathbf{W}_i$ the weight matrix of size (out_features × in_features).

(iii) $\mathbf{b}_i$ is the bias term.

(iv) l is replaced by Mean Squared Error (MSE) loss.

Computing z is called **forward propagation**. Indeed, z is a scalar cost $J(\theta)$ that corresponds to the overall misfit of the network. Though, this cost must flow backwards to update the weights accordingly, and this is what the **backward propagation** or **backprop** algorithm does.

Backprop is what computes the gradients like in Equation (1.6) to (1.7). The term back-propagation is often misunderstood as meaning the whole learning algorithm for MLP networks. Actually, back-propagation refers only to the method for computing the gradients, while another algorithm, such as Stochastic Gradient Descent (SGD), is used to update weights using these gradients.

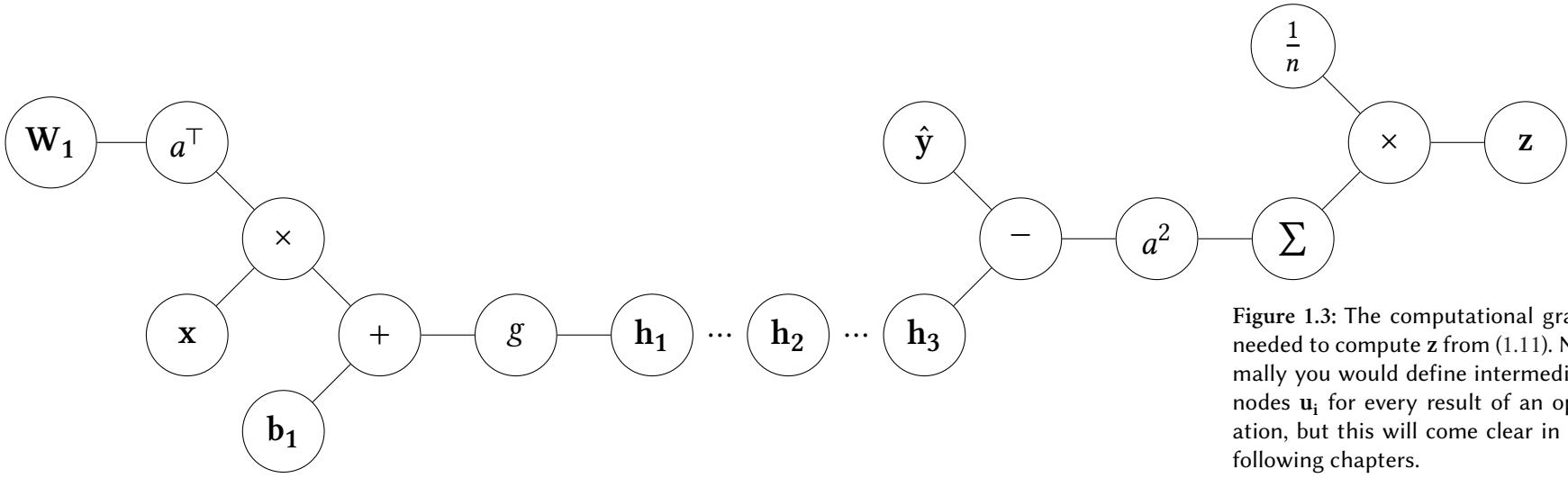


Figure 1.3: The computational graph needed to compute $\mathbf{z}$ from (1.11). Normally you would define intermediate nodes $\mathbf{u_i}$ for every result of an operation, but this will come clear in the following chapters.

The idea of a computational graph comes in handy when gradients need to be computed, because computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive.

Take for instance the gradient of $\mathbf{W_1}$ in Figure 1.3, you would need to compute all the gradients of every subsequent node [5]. As said earlier, backprop only computes the gradients, so these must be stored somewhere for the optimizer, like SGD. In addition, partial derivatives must always be computed in order from the last node to the first one. An efficient backprop algorithm should run in $O(n)$ with respect to the number of nodes [6].

5: Note that all the weights that need to be updated are leaf nodes.

6: Assuming constant time $O(1)$ to compute a partial derivative of an operation.

With those constraints, the idea of a graph is really useful, because the connections are created in the forward propagation and computing backprop is simply done with a topological sort algorithm such as a post-order traversal, and then evaluate every gradient in reverse order, starting from $\mathbf{z}$.

Yet, some data structure is required to store the gradients and also the weights and computations of each intermediate node. **Tensors** are the preferred data structure in deep learning, and are what we have been referring as nodes of the computational graph. A tensor is used to represent multi-dimensional arrays or data in a way that is compatible with the operations and computations performed by neural networks and other machine learning models.

Tensors are characterized not only by their dimensions (rank) or the fact that they store data, but also by the mathematical operations that can be performed on them within a computational graph. That is important because "what is done" with a tensor must be efficient in space and time. Libraries such as **PyTorch** or **Tensorflow** have at their core efficient implementations of tensors and GPU accelerated operations such as convolutions.

For the moment, tensors can be understood as simple matrices, because we are going to implement MLPs first, but later on the idea of multi-dimensional matrices or multi-dimensional arrays will be clarified and motivated through the implementation of batch learning, etc.

1.4 Automatic Differentiation Library

Lots of **Automatic Differentiation (AD)** or **autodiff** libraries are available open-source like MyGrad, Autograd, JAX, micrograd, etc. These libraries are not by definition Deep Learning (DL) frameworks, but through this lab you will see that 90% of what you need for neural networks is a decent AD, tensor based library. DL frameworks like PyTorch or TensorFlow are built on top of an AD library. They also trace computations dynamically at runtime, in comparison to statically typed languages like C or C++, and they provide ways to accelerate computations over the repeated iterations done when training a model, like Just-In-Time (JIT) compilation, vectorization, etc.

Efficient AD is what most of them have in common, on the contrary micrograd is a simple scalar AD library for educational purpose, not like the one you will be building, which is similar to MyGrad, but far more simplified.

All aforementioned libraries are excellent, and not because every one is efficient and fast like war horses, they are excellent because they reach their goals and fulfill the necessities of whoever use them.

Design Notes

The only third-party library needed for this lab is NumPy. Many implementations of efficient tensors are available open-source, bare in mind that every AD library usually has their own tensor class implementation, but building that from source would take too long. NumPy offers a great trade-off between compatibility and speed, but using it makes a great concession about speed, because it cannot use accelerators like GPUs, lazy evaluation is not supported and there's no JIT compiler. But these characteristics are common in a Tensor library, NumPy is an excellent all purpose n-dimensional array library, which is more than enough for what we are gonna do [7].

Now you may wonder which is the difference between a DL framework and an autograd. The main one is that the former provides a neural network specific module, in most cases named "nn" (Neural Network). Hence, DL frameworks tend to have optimized implementations of n-dimensional convolutions or computer vision procedures, which are not indispensable in an AD library.

7: If your are always worried about performance, have a GPU and do not want to put a lot of effort, substituting NumPy arrays for CuPy ones can make an improvement. But this is out of the scope of this lab.

Syntax

PyTorch has a great influence in the way everything is implemented. The idea is to use the same names and try to replicate its behavior to be able to transfer what is learned here to a real world problem, where you most probably use PyTorch. However, everything will be kept simple and only the bare minimum modules and functionalities will be implemented, so that you can train your own neural networks with quiet ease.

2 Autodiff Library

2.1 Multi Layer Perceptron 13
2.1.1 Tensor Broadcasting 15
2.1.2 Output layer 17
2.2 Model 19

The base of the autodiff module is constituted by the **Tensor** and the **Function** class.

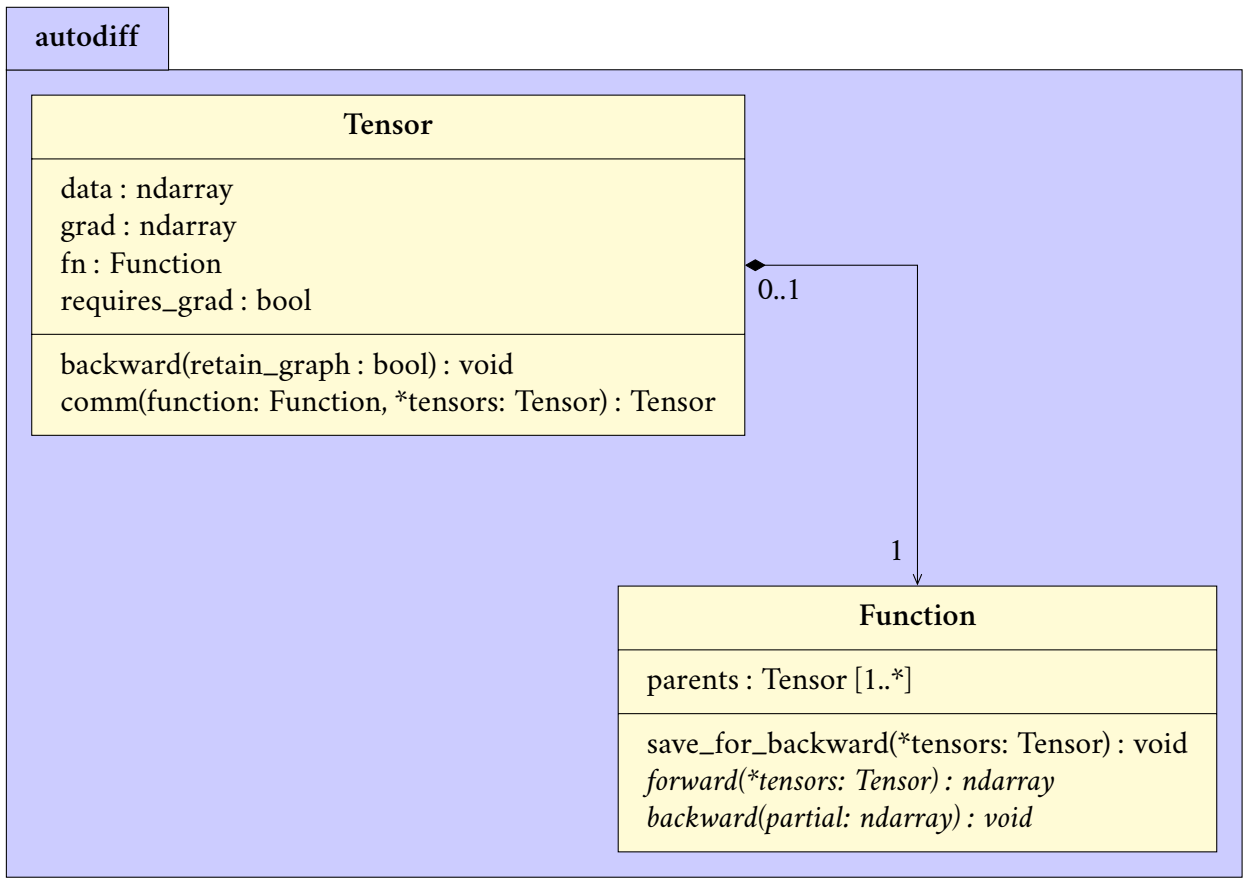


Figure 2.1: Simplified definition and relationship of **Tensor** class and **Function** class.

Tensor.data and **Tensor.grad** are NumPy arrays, but **Tensor.grad** is only initialized to *zeros* if you explicitly define that it **requires_grad**. By default every NumPy array has single-precision floating point elements, but you can change that if you want. That precision is the default one in almost all DL frameworks, and data types (**dtype**) are really important.

```
>>> import torch
>>> a = torch.tensor([2], requires_grad=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Only Tensors of floating point and complex dtype can require gradients
```

You can create an instance of the class **Tensor** directly, passing whatever *array_like* object accepted by **numpy.array**. In addition, you can create one with **Tensor.comm** classmethod[1] which is the method internally used by the **Tensor** class to create new tensors given some **Function**'s instance.

```
@classmethod
def comm(cls, function: Function, *tensors) -> Tensor:
    operands = [t if isinstance(t, Tensor) else Tensor(t) for t in tensors]
    data = function.forward(*operands)
    # NOTE: if no leaf tensor requires_grad, neither intermediate ones
    requires_grad = any(t.requires_grad for t in operands)
    return cls(data, requires_grad=requires_grad, fn=function)
```

1: A class method's purpose is to perform actions or operations that are related to the class itself rather than specific instances, often used for creating new instances of that class. `@classmethod` is used in python to denote that.

`*tensors` is an **arg** in python, similar to **kwargs**, you may find useful this link to understand, but you can assume is just an iterable of tensors. **Tensor.comm** purpose is to use the functionality of a **Function** without populating **Tensor** with a lot of code, it just gives modularity and makes your code cleaner. The **Function** class, on the other hand, is what we already know as an operator in the computational graph, and is an abstract class [2]. Take for instance the way **Add** is defined, you just need to define more operators in the operator directory and you are good to go.

2: An abstract class is a class in object-oriented programming that cannot be instantiated and must contain at least one abstract method. It can be understood as a template you must follow to make everything work correctly.

```
class Add(Function):
    def __init__(self):
        super().__init__()

    def forward(self, t1, t2) -> NDArray:
        self.save_for_backward(t1, t2)
        return t1.data + t2.data

    def backward(self, partial: NDArray):
        p1, p2 = self.parents
        if p1.requires_grad:
            p1.grad += partial

        if p2.requires_grad:
            p2.grad += partial
```

save_for_backward just saves every parent tensor in **parents**, but that behaviour is inherited with **Function**. The idea is that you save parent tensors in the **forward** pass and return the NumPy array of their data with the result of the operation you are defining. Then, in **backward**, you will be given the partial derivative with which you must update **parents**' gradients. Checking if a parent **requires_grad** is not that relevant to train Feedforward Neural Network in this toy library, but remember that the input data **x** in Figure 1.3 is also a Tensor, so you should avoid updating its gradient at all cost in a real world example.

2.1 Multi Layer Perceptron

Now you have the bare minimum classes to implement a MLP. But first let's define a simple perceptron as a graph.

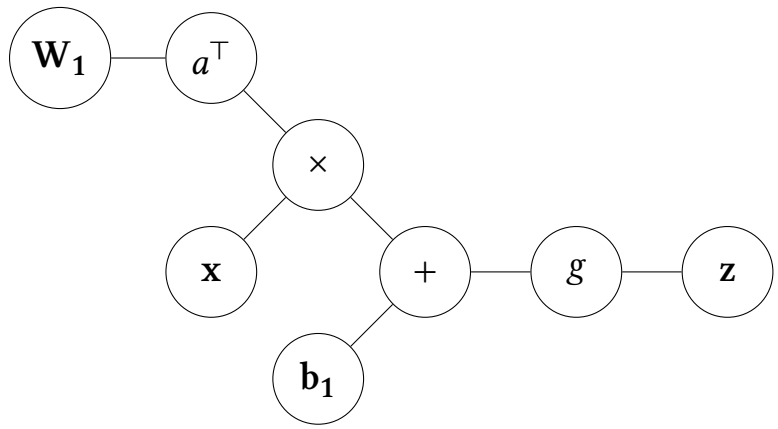


Figure 2.2: The computational graph of a perceptron.

As you may have noticed is just the same definition of a layer in Figure 1.3, speaking of which, it only needs transposition, matrix multiplication, an activation function g and adding two tensors (already implemented).

Exercise 1:

> Implement the transposition operator in `othergrad/operators/reshape.py`.

You may find useful this piece of code [3] to transpose a tensor like NumPy does, but you can define the method you want, just be consistent with your implementation.

```
@property
def T(self):
    assert self.data.ndim == 2, "Dimensions = 2 required, this is matrix \
   transposition"
    return Tensor.comm(ops.Transpose(), self)
```

And use it like this:

```
>>> from othergrad import Tensor
>>> a = Tensor([[1,2], [3,4]])
>>> a.T
```

If you implement transposition with another name, please modify lines 33 and 35 in `othergrad/nn/layers/linear.py`.

3: Using `@property` gives you the possibility to control the access of a class attribute. Acts like a normal class attribute but python internally calls a method.

Exercise 2:

> Implement matrix multiplication operator in `othergrad/operators/math.py`. After that you will be able to multiply two tensors like this:
>
> ```
> >>> a = Tensor(np.random.rand(10, 15))
> >>> b = Tensor(np.random.rand(15, 5))
> >>> a @ b
> ```

Matrix multiplication forward propagation is pretty straight forward, you just multiply two matrices, but backpropagation may seem a little tricky. Take for instance two matrices A and B, and also $AB = C$.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix} \tag{2.1}$$

$$C = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} \end{pmatrix}$$

It's easy to see that

$$\frac{\partial C}{\partial a_{11}} = b_{11} + b_{12}$$

but note that **Function.backward** receives **partial**, that is the gradient of C. If C is the last node in the computational graph then you will receive all ones, see how the gradient is initialized in **Tensor.backward**.

```
def backward(self, retain_graph=False):
    ...
    build_topo(self)

    # chain rule
    self.grad = np.ones_like(self.data) # dL/dL = 1
    for tensor in reversed(topo):
        tensor.fn.backward(tensor.grad)
        if not retain_graph:
            tensor.fn = None
    ...
```

So the gradient of C, let's say C_g

$$C_g = \begin{pmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \\ g_{31} & g_{32} \end{pmatrix}$$

must be multiplied by $\frac{\partial C}{\partial A}$ and $\frac{\partial C}{\partial B}$. Take for instance the role of b_{11} in $\frac{\partial C}{\partial a_{11}}$, this variable is part of g_{11}, so it has to be multiplied by g_{11} owing to the chain rule. That gives us the following result.

$$A_g = \begin{pmatrix} g_{11}b_{11} + g_{12}b_{12} & g_{11}b_{21} + g_{12}b_{22} & g_{11}b_{31} + g_{12}b_{32} \\ g_{21}b_{11} + g_{22}b_{12} & g_{21}b_{21} + g_{22}b_{22} & g_{21}b_{31} + g_{22}b_{32} \\ g_{31}b_{11} + g_{32}b_{12} & g_{31}b_{21} + g_{32}b_{22} & g_{31}b_{31} + g_{32}b_{32} \end{pmatrix} \tag{2.2}$$

$$B_g = \begin{pmatrix} a_{11}g_{11} + a_{21}g_{21} + a31g_{31} & a_{11}g_{12} + a_{21}g_{22} + a31g_{32} \\ a_{12}g_{11} + a_{22}g_{21} + a32g_{31} & a_{12}g_{12} + a_{22}g_{22} + a32g_{32} \\ a_{13}g_{11} + a_{23}g_{21} + a33g_{31} & a_{13}g_{12} + a_{23}g_{22} + a33g_{32} \end{pmatrix}$$

I hope you see the pattern now.

Exercise 3:

> Implement ReLU activation function in `othergrad/operators/activation.py` defined as :
>
> $$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$

2.1.1 Tensor Broadcasting

If you execute the following code:

```
>>> a = Tensor([[1,1,1], [1,1,1], [1,1,1]], requires_grad=True)
>>> b = Tensor([[0,1,2]], requires_grad=True)
>>> c = a+b
>>> c
tensor: [[1. 2. 3.]
         [1. 2. 3.]
         [1. 2. 3.]] fn: Add
>>> c.backward()
```

you will get **ValueError: non-broadcastable output operand with shape...** and that's because **b** Tensor is broadcasted over **a**. That means that every row of **a** is added to **b**.

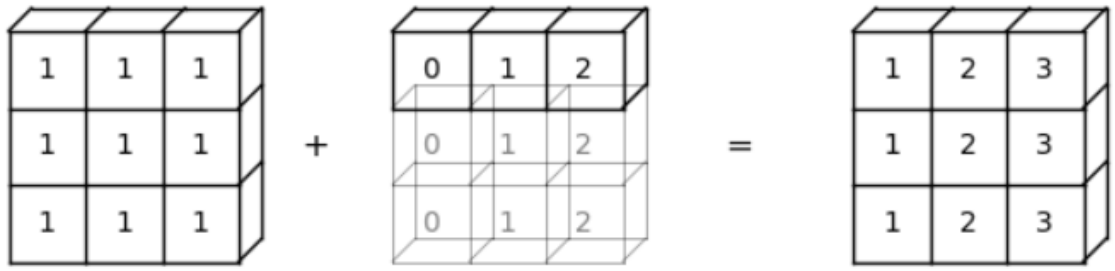


Figure 2.3: Tensor add broadcasting.

Broadcasting is one of properties that define a Tensor and is really useful to avoid having memory leakages. Take for instance training an MLP represented informally in the following image.

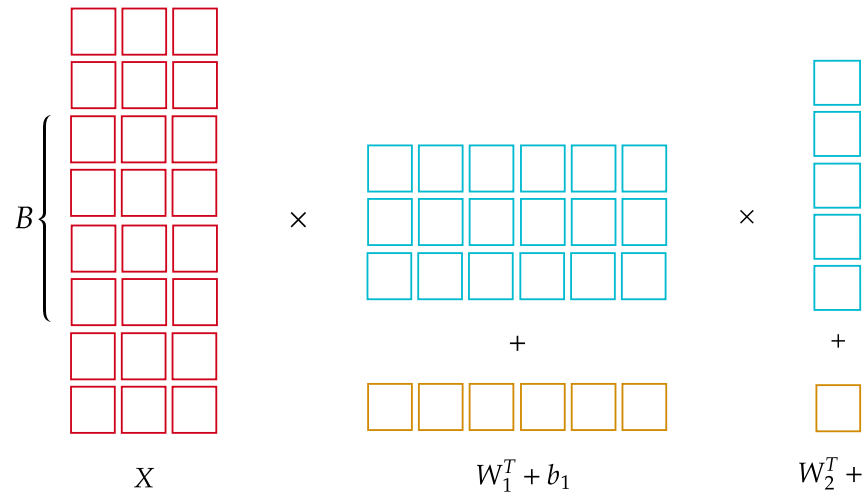


Figure 2.4: An MLP that takes $\mathbb{R}^3$ as input and outputs $\mathbb{R}$, but training data is fed into a single batch B.

After the first layer, the output tensor will be 8×6 and b_1 will be broadcasted along the rows (first dimension). When you apply backpropagation what you are trying to do is adding a 8×6 to b_1 gradient, and hence NumPy is reporting **ValueError**. From a mathematical perspective broadcasting in this scenario is applying the funtion $f(x) = x + b_1$ to every broadcastable dimension, which means that you need to add every $\frac{\partial f}{\partial b_1}$ to b_1's gradient.

Exercise 4:

Correct Add operator to take into account broadcasting. Do not assume you will always add b_1 from the right hand side, + is commutative, but you can assume that an input tensor has a maximum of two dimensions (a matrix). For that exercise you will need numpy.sum, numpy.reshape and modify **collapse** in `othergrad/operators/math.py`.

In the context of an MLP you just need to sum along the first axis, but bare in mind that libraries like PyTorch or Tensorflow are able to do it properly with any pair of compatible tensors.

Try to think what happens when you add and backpropagate tensors with more complex shapes like $(3, 5, 7, 5, 2)$ and $(5, 7, 1, 1)$, or $(5, 7, 10)$ and $(5, 7, 10, 3, 2)$.

Do you see any pattern?

```
>>> (np.ones((2,5)) + np.ones((2,5,3,2,5))).shape
(2, 5, 3, 2, 5)
>>> (np.ones((28,)) + np.ones((128,28,28))).shape
(128, 28, 28)
>>> (np.ones((3,)) + np.ones((1,1,1,1,1))).shape
(1, 1, 1, 1, 3)
```

2.1.2 Output layer

In this Lab you are trying to create your own neural network to predict digits from MNIST dataset, which means that you must implement Softmax activation function, just like you've done with ReLU. But before that, let's consider some key aspects of Softmax function.

Unlike ReLU and many other activation functions, Softmax requires a vector as input, meaning that you cannot define a generic activation function like you have done in **Exercise 3** and rely on NumPy to apply it to every value in a tensor. However you can define your own implementation with tricky tensor manipulation.

SoftMax for an element x_i in a vector $\mathbf{x}$ is defined as:

$$z_i = Softmax(\mathbf{x}_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

which means that $\frac{\partial z_i}{x_i}$ is different from $\frac{\partial z_i}{x_j}$ where $i \neq j$. Implementing Softmax's backward pass can be quite tricky, and unfortunately is not numerically stable, producing float overflow if you really want to do it with NumPy and `numpy.float32` precision (the one we are using) [4].

4: You can set all NumPy arrays to `numpy.float64`, but that is not realistic. In deep learning there are lots of problems related to memory bottlenecks, and that is why `float32` and mixed precision is used.

Though, one trick you have is to implement *"LogSoftmax"* owing to the Cross Entropy loss.

$$l(\mathbf{x}, \mathbf{y}) = -\sum_{i=1}^{N} \mathbf{y}_i \log \frac{e^{\mathbf{x}_i}}{\sum_{j=1}^{N} e^{\mathbf{x}_j}} \tag{2.3}$$

Where $\mathbf{y}$ is a one hot vector. If you are predicting class 4, then $\mathbf{y}_i = 1$ for $i = 4$ and $\mathbf{y}_i = 0$ if $i \neq 4$.

Exercise 5:

> Implement LogSoftmax activation function simplifying equation 2.3. You can define your own class in `othergrad/operators/activation.py` or just define the method in `othergrad/tensor.py` using `Tensor.sum`, `Tensor.exp` and `Tensor.log`, which are already defined.
>
> In addition, one trick you can use is substracting the maximum value of the vector $\mathbf{x}$ to every element $\mathbf{t} = \mathbf{x} - \max_{1 \leq i \leq N} \mathbf{x}_i$.
>
> $$LogSoftmax(\mathbf{x}_i) = \log \frac{e^{\mathbf{t}_i}}{\sum_{j=1}^{N} e^{\mathbf{t}_j}}$$

`Tensor.max` is already implemented, but you will need to define the `axis` and decide to set `keepdims` to `True` or `False`.

```
>>> a = Tensor(np.arange(12).reshape((4, 3)))
>>> a
tensor: [[ 0  1  2]
         [ 3  4  5]
         [ 6  7  8]
         [ 9 10 11]]
>>> a.max()
tensor: 11. fn: Max
>>> a.max(axis=0)
tensor: [ 9 10 11] fn: Max
>>> a.max(axis=0, keepdims=True)
tensor: [[ 9 10 11]] fn: Max
```

2.2 Model

The last step to create your model is defining the loss function and selecting an optimizer. For predicting multiple categories you need `CrossEntropyLoss` which is already implemented in `othergrad/nn/loss/CrossEntropyLoss.py`.

```
class CrossEntropyLoss(Module):
    def __init__(self):
        super().__init__()

    def __call__(self, pred: Tensor, target: Tensor) -> Tensor:
        one_hot = np.zeros_like(t1.data)
        one_hot[np.arange(target.size), target.data] = 1

    return (-one_hot*pred.log_softmax()).mean(axis=0).sum()
```

The loss function is also part of the computational graph. Note that you do not need to define a new class derived from `othergrad.tensor.Function`, we just reuse already implemented operations. The last step would be defining an optimizer [5] , but SGD is already implemented.

5: You can define Adam, Adagrad, etc. See the class pattern in `othergrad/optim/SGD.py`, you just need to interpret the pseudocode of torch.optim.Adam and define your `.step` method.

The aim of an optimizer is to just apply gradient descent, but everyone has its own particularity, the idea behind training a feed-forward network is to obtain gradients with backpropagation (what you have done so far) and apply an optimizer to every trainable tensor to update its weights.

One of the reasons PyTorch is so popular is its pythonic way to define neural networks. Everything inherits from `nn.Model`, which results in a tree like structure of neural network modules, where a leaf of the tree is a tensor. In othergrad `nn.Module` is defined in `othergrad/nn/loss/module.py`.

Exercise 6:

> Define your model in `mnist.py` .
>
> ```
> import othergrad.nn as nn
> import othergrad.optim as optim
>
> ...
>
> class MLP(nn.Module):
>     def __init__(self):
>         super().__init__()
>         # define the layers of your MLP
>         # self.fc1 = nn.Linear(in_features, out_features)
>         # self.fc2 = ...
>         # ...
>
>     def __call__(self, x):
>         # define the forward pass
>         # x = self.fc1(x).relu()
>         # x = self.fc2(x).relu()
>         # return self.fc10(x)
>
> model = MLP()
> criterion = nn.CrossEntropyLoss()
> optimizer = optim.SGD(
>     model.parameters(),
>     lr=0.001,
>     momentum=0.9,
>     nesterov=True
> )
>
> for _ in (prog_bar := trange(EPOCHS)):
>     for X, Y in batched_data:
>         output = model(X)
>         loss = criterion(output, Y)
>         loss.backward()
>         optimizer.step()
>         optimizer.zero_grad()
> ...
> ```
>
> Finally execute `mnist.py` and see how well your MLP performs.

Note that in **`class`** `MLP` you just need to define `__call__`, a.k.a the forward pass and the backward pass or backpropagation starts with `loss.backward()` inside the inner **`for`** loop. What you have done is implementing all the necessary steps to automate this simple line `loss.backward()`, then in every epoch, for every mini-batch, you are calling the forward pass with `model(X_train)`, applying the loss function, doing backpropagation, updating the weights with `optimizer.step()` and finally setting all gradients to zero with `optimizer.zero_grad()`, because you do not want to incrementally add gradients in every iteration or you will have exploding gradients.

Special Terms

A

AD Automatic Differentiation. 9, 10

D

DAG Directed Acyclic Graph. 6

DL Deep Learning. 2, 9–11

F

FNN Feedforward Neural Network. 5–7, 12

G

GLM Generalized Linear Model. 6

J

JIT Just-In-Time. 9, 10

M

MLP Multi Layer Perceptron. 5, 7–9, 13

MSE Mean Squared Error. 7

R

RNN Recurrent Neural Network. 5

S

SGD Stochastic Gradient Descent. 8