Goals

The goal of this practical work is to build a simple neural network and to become familiar with these models and how to train them with the *backpropagation* (or backpropagation) of the gradient.

To do this, we will begin by theoretically studying a perceptron with a hidden layer and its learning procedure. We will then implement this network with the PyTorch library first on a toy problem to check that it is working correctly, then on the MNIST dataset.

The site `http://playground.tensorflow.org` makes it possible to visualize the functioning and the learning of small neural networks. You can go there to better understand this lab.

Section 1 – Theoretical foundation

To apply a neural network to a *machine learning* problem (in supervised learning), we need 4 things adapted to the problem:

- A **supervised dataset** ;
- A **network architecture** ;
- A **loss function** that we will optimize ;
- An **optimization algorithm** to minimize the loss function.

1.1 Supervised dataset

We are interested in a supervised classification task. Thus, we have access to a supervised dataset made of N (features, targets) pairs $(x^{(i)}, y^{(i)}), i \in 1..N$, where $x^{(i)} \in \mathbb{R}^{n_x}$ is a *features* vector from which we would like to deduce the (*target, ground truth*) $y^{(i)} \in \{0,1\}^{n_y}$ such as $\|y^{(i)}\| = 1$ (*one-hot encoding*: only one component is 1, indicating the class to which the sample belongs). This supervised dataset is split into several sets: *train*, *test* and if possible *val*.

Questions

1. ★ What are the train, val and test sets used for?
2. What is the influence of the number of examples N ?

1.2 Network architcture (forward)

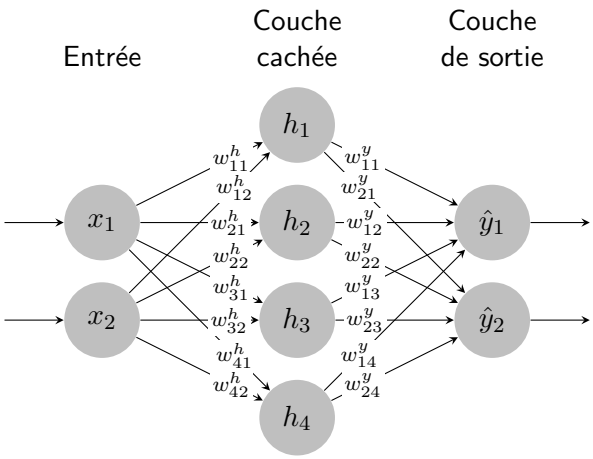

Figure 1: Neural network architecture with only one hidden layer.

A neural network f is a succession of mathematical transformations allowing to pass from the space of *features* to the space of predictions:

$$\begin{array}{rccc} f: & \mathbb{R}^{n_x} & \to & \mathbb{R}^{n_y} \\ & x & \mapsto & \hat{y} \end{array} \tag{1}$$

The computation of the output $\hat{y}$ from the input x is called the *forward* of the network.

A very simple neural network can for example consist of a simple **linear transformation**, we will then have $\hat{y} = f(x) = Wx$ with W a matrix of size $n_y \times n_x$. This transformation is the standard layer of classical neural networks. As a side note, we often use an affine transformation $f(x) = Wx + b$, where b a vector of size n_y.

Generally, each linear transformation is followed by a **activation function** which is a non-linear mathematical function whose role is twofold. First, it makes the combination of transformations (linear + activation) non-linear, thus justifying applying several transformations in a row. Then, it allows to choose an output interval different from $\mathbb{R}^{n_y}$: for instance $\tanh$ allows to have a result in the interval $[-1, 1]^{n_y}$.

The most frequent activation functions are:

- ReLU (Rectified Linear Unit) which matches negative values to 0 ($\mathrm{ReLU}(x) = \max(0, x)$) ;
- $\tanh$ which brings values into the interval $[-1, 1]$;
- σ (sigmoid) which brings values into the interval $[0, 1]$ ($\sigma(x) = 1/(1 + \exp(x))$) ;
- $\mathrm{SoftMax}$ which brings values into the interval $[0, 1]$ and such as $\sum_i \mathrm{SoftMax}(x)_i = 1$, can be analyzed as a probability distribution ($\mathrm{SoftMax}(x)_i = \exp(x_i)/\sum_j \exp(x_j)$).

As indicated previously, we will generally have in a neural network **a succession of linear layers each followed by an activation function**. Figure 1 shows a fairly simple network with a hidden layer (activation functions are not shown).

As part of this lab, **we will focus on the following architecture**:

- One **"hidden" layer** from the input of size n_x to a vector h of size n_h, made of :
  - An **affine transformation** using a **matrix of weights** W_h of size $n_h \times n_x$ and a **bias** vector b_h of size n_h ;
    We note $\tilde{h}$ the output vector of this transformation
  - The **activation function** $\tanh$;
    We note h the output vector of this transformation
- An **output layer** from the hidden features of size n_h to the output vector $\hat{y}$ of size n_y, made of :
  - An **affine transformation** using a **matrix of weights** W_y of size $n_y \times n_h$ and a **bias** vector b_y of size n_y ;
    We note $\tilde{y}$ the output vector of this transformation
  - The **activation function** $\mathrm{SoftMax}$.
    We note $\hat{y}$ the output vector of this transformation

Questions

3. Why is it important to add activation functions between linear transformations?
4. ★ What are the sizes n_x, n_h, n_y in the figure 1? In practice, how are these sizes chosen?
5. What do the vectors $\hat{y}$ and y represent? What is the difference between these two quantities?
6. Why use a $\mathrm{SoftMax}$ function as the output activation function?
7. Write the mathematical equations allowing to perform the *forward* pass of the neural network, i.e. allowing to successively produce $\tilde{h}$, h, $\tilde{y}$ and $\hat{y}$ starting at x.

1.3 Loss function

Thanks to the *forward* pass, our neural network produces for the input $x^{(i)}$ an output $\hat{y}^{(i)}$. We would like to know how much this output differs from the *target* $y^{(i)}$. For that, we use a *loss* function adapted to our problem. Most of the time, the global loss function $\mathcal{L}(X, Y)$ is the average of a unit loss function $\ell(y^{(i)}, \hat{y}^{(i)})$ between predictions and *targets*:

$$\mathcal{L}(X, Y) = \frac{1}{N} \sum_i \ell\big(y^{(i)}, f(x^{(i)})\big).$$

There are many loss functions, the two most common being:

- the **cross-entropy**, adapted to the classification tasks :

$$\ell(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i;$$

- the **mean squared error, MSE**, adapted to the regression tasks:

$$\ell(y, \hat{y}) = ||y - \hat{y}||_2^2 = \sum_i (y_i - \hat{y}_i)^2.$$

Questions

8. During training, we try to minimize the loss function. For cross entropy and squared error, how must the $\hat{y}_i$ vary to decrease the global loss function $\mathcal{L}$?
9. How are these functions better suited to classification or regression tasks?

1.4 Optimization algorithm

Thanks to the loss function, we therefore have a measure of the error of our neural network. We must now learn its parameters (the matrices and vectors $W.$ and $b.$ in our example) in order to minimize this error on the set of training examples.

Gradient descent To do this, we will use the gradient descent algorithm. It consists in calculating the derivatives of the loss with respect to a parameter w, and to take a step in the opposite direction of the gradient, that is to say to modify the value of w through :

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}(X, Y)}{\partial w}$$

where η is called the *learning rate* and controls the speed of the optimization. This step is repeated for a predefined number of iterations or until the learning converges.

The gradient $\frac{\partial \mathcal{L}(X, Y)}{\partial w}$ can be calculated on different sets, which corresponds to different variations of the gradient descent algorithm:

- **the full training set**: this is the **classic gradient descent**;
- a small **subset** (generally a few dozen) of learning examples drawn at random: this is the **stochastic gradient descent by mini-batch** (*stochastic gradient descent, SGD*);
- **a single example** $(x^{(i)}, y^{(i)})$, so we replace $\mathcal{L}$ by the unit cost ℓ: this is the **online stochastic gradient descent**.

Backpropagation To calculate the gradient of the output with respect to the weights of the network, we apply the *chain rule*:

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b} \frac{\partial b}{\partial a}.$$

In the vector case, if a, b, c are vectors, we have for the derivative of the component c_i with respect to the component a_k:

$$\frac{\partial c_i}{\partial a_k} = \sum_j \frac{\partial c_i}{\partial b_j} \frac{\partial b_j}{\partial a_k}. \tag{2}$$

Applied in a naive way, the calculation of the gradients repeats several times in each layer the same gradient computations from the *loss*. This is very inefficient and it is possible to do better by using the *chain rule* principle of *backpropagation* of the gradient, which consists in chaining the gradients in order to travel only once on the network. We schematically visualize the application of the *backpropagation* on a neural network in the figure 2. The principle is that the gradient calculated with respect to the output of a

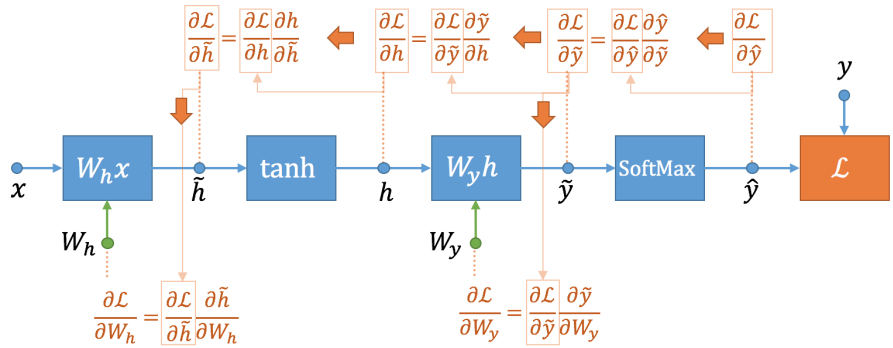

Figure 2: Schematic view of the *backpropagation* on a neural network. We start by calculating the derivative of the cost $\mathcal{L}$ compared to the output ($\hat{y}$ on the diagram) then we go back in the network by reusing the derivatives calculated previously during the calculation of the new derivatives.
Note: in the vector case, it is necessary to add sums as indicated in the equation (2).

layer can be reused to calculate the gradients with respect to the input and the parameters of this same layer: it suffices to multiply it by a simple local gradient (from the output relative to input or parameters).

The successive calculation of the gradients of the different layers is called the *backward* pass.

Questions

10. What seem to be the advantages and disadvantages of the various variants of gradient descent between the classic, mini-batch stochastic and online stochastic versions? Which one seems the most reasonable to use in the general case?
11. ★ What is the influence of the *learning rate* η on learning?
12. ★ Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the *loss* with respect to the parameters, using the naive approach and the *backprop* algorithm.
13. What criteria must the network architecture meet to allow such an optimization procedure ?
14. The function $\mathrm{SoftMax}$ and the *loss* of *cross-entropy* are often used together and their gradient is very simple. Show that the *loss* can be simplified by:

$$\ell = -\sum_i y_i \tilde{y}_i + \log\left(\sum_i e^{\tilde{y}_i}\right).$$

15. Write the gradient of the *loss* (*cross-entropy*) relative to the intermediate output $\tilde{y}$

$$\frac{\partial \ell}{\partial \tilde{y}_i} = \dots \qquad \Rightarrow \qquad \nabla_{\tilde{y}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \dots$$

16. Using the *backpropagation*, write the gradient of the *loss* with respect to the weights of the output layer $\nabla_{W_y} \ell$. Note that writing this gradient uses $\nabla_{\tilde{y}} \ell$. Do the same for $\nabla_{b_y} \ell$.

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \dots \qquad \Rightarrow \qquad \nabla_{W_y} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{W}_{y,11}} & \cdots & \frac{\partial \ell}{\partial \tilde{W}_{y,1 n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \tilde{W}_{y,n_y 1}} & \cdots & \frac{\partial \ell}{\partial \tilde{W}_{y,n_y n_h}} \end{bmatrix} = \dots$$

17. ★ Compute other gradients : $\nabla_{\tilde{h}} \ell$, $\nabla_{W_h} \ell$, $\nabla_{b_h} \ell$.

Section 2 – Implementation

We now have all the equations allowing us to make predictions (*forward*), to evaluate (*loss*) and to learn (*backward* and gradient descent) our model.

We are now going to implement this network with PyTorch. The objective of this part is to gradually familiarize yourself with this framework.

This part is inspired by the PyTorch getting started tutorial available at `http://pytorch.org/tutorials/beginner/pytorch_with_examples.html`. Do not hesitate to watch it in parallel with this lab to help you write the requested functions.

2.1 Forward and backward manuals

We will start by coding the neural network by simply using basic mathematical operations and therefore directly transcribing our mathematical equations. We will use the functions found in the `torch`: `https://pytorch.org/docs/1.2.0/index.html` package.

In torch, data is stored in objects of type `torch.Tensor`, equivalent to `numpy.array`. The basic PyTorch functions return objects of this type.

1. ★ ★ **Discuss and analyze your experiments following the implementation. Provide pertinent figures showing the evolution of the loss; effects of different batch size / learning rate, etc.**
2. Write the function `init_params(nx, nh, ny)` which initializes the weights of a network from the sizes n_x, n_h and n_y and stores them in a dictionary. All weights will be initialized according to a normal distribution of mean 0 and standard deviation 0.3.

   Hint: use the `torch.randn` and `torch.zeros` functions.
3. Write the function `forward (params, X)` which calculates the intermediate steps and the output of the network from an input batch `X` of size $n_{batch} \times n_x$ and weights stored in `params` and store them in a dictionary. We return the dictionary of intermediate steps and the output $\hat{Y}$ of the network.

   Hint: we will use `torch.mm` for matrix multiplication, and `torch.tanh`, `torch.exp`, `torch.sum`
4. Write the function `loss_accuracy (Yhat, Y)` which computes the cost function and the precision (rate of good predictions) from an output matrix $\hat{Y}$ (output of `forward`) with respect to a ground truth matrix Y of the same size, and returns the *loss* `L` and the precision `acc`.
   Note: We will use the `_, indsY = torch.max (Y, 1)` function which returns the index of the predicted class (or to be predicted) for each example.

   Hints: `torch.mean`, `torch.max`, `torch.log`, `torch.sum`
5. Write the function `backward (params, outputs, Y)` which calculates the gradients of the *loss* with respect to the parameters and stores them in a dictionary.
6. Write the function `sgd (params, grads, eta)` which applies a stochastic gradient descent by mini-batch and updates the network parameters from their gradients and the learning step .
7. Write the global learning algorithm using these functions.

load / prepare data
initialize the network
// *Iterate* N_{epoch} *times on the dataset*
for $i = 1..N_{epoch}$
    // *Iterate on the batches from the dataset*
    for $j = 1..(N/N_{batch})$
        // *Training batch, according to the gradient descent method*
        $(X_{batch}, Y_{batch}) \leftarrow$ subsets of *training* samples
        batch *forward* pass
        computing the *loss* on the batch
        batch *backward* pass
        gradient descent
    computing and showing the *loss* and the accuracy on *train* and *test*
    showing the decision boundary with `plot_data_with_grid`
showing the *loss* dynamics

2.2 Simplification of the backward pass with `torch.autograd`

PyTorch provides an automatic differentiation mechanism (*autograd*) available on all tensors. By default, on a tensor, this functionality is disabled. To activate it, you must put `requires_grad` to `True`, i.e. when creating the tensor (`torch.tensor (data, requires_grad = True)`) or define the *flag* then (`x.requires_grad = True`).

To use autograd, we do `loss.backward ()`, here on the tensor named `loss`, and autograd then computes the derivative of `loss` with respect to all tensors which produced the `loss` tensor. The gradients of all leaf tensors (those which are not the result of calculations of other tensors) are stored in the `grad` attribute of these tensors, for example `W. grad` if `W` is a leaf tensor.

Concretely, in your code, you must:

1. Activate autograd on the weights of the network:

```
# You can either specify it at initialization:
params["Wh"] = torch.randn(nh, nx, requires_grad=True)
# or specify later
params["Wh"] = torch.randn(nh, nx) ## default False
params["Wh"].requires_grad = True
```

2. To calculate the gradients, call the `backward` method on the element with respect to which we want to calculate the gradient, i.e. the *loss*:

```
L, _ = loss_accuracy(Yhat, Y)
# grads = backward(params, outputs, Y) # deleted
L.backward() # compute the gradients
```

3. Use the gradient calculated in the variables to update the weights.

```
# For instance, replace:
params['Wy'] -= eta * grads['Wy']
# by:
with torch.no_grad(): # Warning: use torch.no_grad()
```

```
    params['Wy'] -= eta * params['Wy'].grad
    params['Wy'].grad.zero_() # resets the gradient accumulator to zero
```

2.3 Simplification of the forward pass with `torch.nn` layers

PyTorch provides the standard layers of neural networks in the form of modules, making it possible to simplify and make more readable the writing of the network architecture.

1. Implement the `init_model (nx, nh, ny)` function which will declare the model architecture and the *loss* and return them.
   Example (to be adapted to the problem):

```
def init_model(nx, nh, ny):
    model = torch.nn.Sequential(
        torch.nn.Linear(nx, nh),
        torch.nn.Tanh(),
        torch.nn.Linear(nh, ny),
    )
    loss = torch.nn.CrossEntropyLoss()
    return model, loss
```

2. Replace the call to the function `forward` by a call to the model, and the computation of the *loss* in `loss_accuracy` by a call to the function of *loss* (new input parameter of the `loss_accuracy` function):

```
# Replace
Yhat, outs = forward(params, X)
# by
Yhat = model(X)

# The loss is then:
L = loss(Yhat, Y)
```

3. The `sgd` function now takes the model as input. To make a gradient descent, we iterate on its parameters and apply our gradient descent:

```
def sgd(model, eta):
    with torch.no_grad():
        for param in model.parameters():
            param -= eta * param.grad
        model.zero_grad()
```

Be careful, we can see that the `torch.nn.CrossEntropyLoss` function applies to the vector $\tilde{y}$ (therefore without $\mathrm{SoftMax}$). Remember to apply the $\mathrm{SoftMax}$ on the model output where necessary, for example on `Ygrid` by doing `torch.nn.Softmax()(Ygrid)` for the display to be correct.

2.4 Simplification of the SGD with `torch.optim`

The `torch.optim` package contains many common optimizers (different variations of the simple SGD algorithm we use here).

1. Modify `init_model` by adding the *learning rate* as an input parameter, and make sure that it also returns the desired optimizer:

```
def init_model(nx, nh, ny, eta):
    # previously
    optim = torch.optim.SGD(model.parameters(), lr=eta)
    return model, loss, optim
```

2. We replace the `sgd` function with a call to `optim.zero_grad ()` before doing the *backward*. Make a call to `optim.step ()` after the *backward*.

```
# Replace
L.backward()
model = sgd(model, 0.03)
# by
optim.zero_grad()
L.backward()
optim.step()
```

2.5 MNIST application

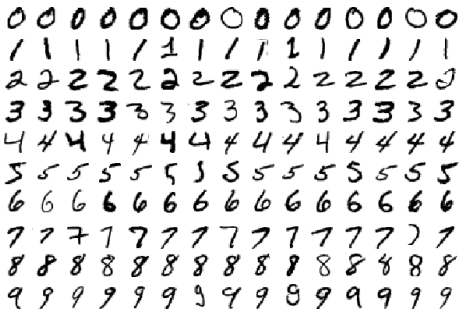

Figure 3: MNIST dataset.

Finally, you can apply your code to the MNIST dataset. MNIST is a set of handwritten digit images (see figure 3), so there are 10 classes ($n_y = 10$). The images are 28×28 pixels, thus will be represented as a vector of 784 values. To use this dataset, use the `MNISTData` class which functions like the `CirclesData` class without the dot grid and display functions.

2.6 Bonus: SVM

Try training an SVM on the Circle dataset. See the code for more details.