# Experiment 1A: a basic neural network

Github repo: `https://github.com/mariezz/visualize-nn-learning`

Marie Peeters

May 6, 2025

## Contents



## 1 Context and introduction

This document will guide you through a first experiment involving a basic neural network. Section 2 introduces the neural network along with the task it tries to solve. Section 3 gives some explanations about the notebook and how you can use it. Section 4 describes the experiment and discusses the results, and section 5 concludes this first experiment.


Figure 1: Plots of the target functions $f_1$ (left) and $f_2$ (right).

# 2 A Basic Neural Network

## 2.1 The target function $f$

The first experiment consists of programming a very small neural network and observe how it manages to learn some 1-dimensional continuous functions. We use the following target functions over the interval $[-5; 5]$:

$$f_1(x) = \exp\left(\frac{x}{5}\right)$$
$$f_2(x) = \sin(x) \cdot (-x^2 + 25)$$

Figure 1 shows the plots of these functions.

## 2.2 The dataset

In real applications, the target function is unknown and we try to let a neural network learn it by giving it examples of inputs and corresponding outputs [1]. In this toy experiment, we know the function and we "artificially" create a dataset by choosing random numbers (in the interval $[-5; 5]$) as inputs and using our target function $f$ (which in this experiment is one of $f_1$ or $f_2$) to compute the outputs. We construct a training set $\mathcal{D} = \{x^{(n)}, y^{(n)}\}_{1 \leq n \leq N}$ containing $N = 1000$ data points, and a test set of size 100.

## 2.3 The architecture

Recall that two-layer perceptrons (2LP) are universal function approximators: for each continuous function over a bounded interval, there exists a 2LP that approximates that function with the desired accuracy. But the universal approximation theorem does not say how many neurons we need or how we can learn the parameters.

So, we will use a 2-layer perceptron (that is, a fully connected feedforward neural network with one hidden layer) and vary the number $h$ of hidden neurons. The neural network must, like $f$, take one number as input and produce one output. So, it has two input neurons (one is a bias unit, whose value is set to 1) and one output neuron. We choose the linear activation function (i.e., the identity function) for the output layer (this is standard for regression tasks) and the sigmoid activation function for the

[1] And, very often, the dataset is noisy, so the example outputs we give the network are not even the ones we hope it will learn.

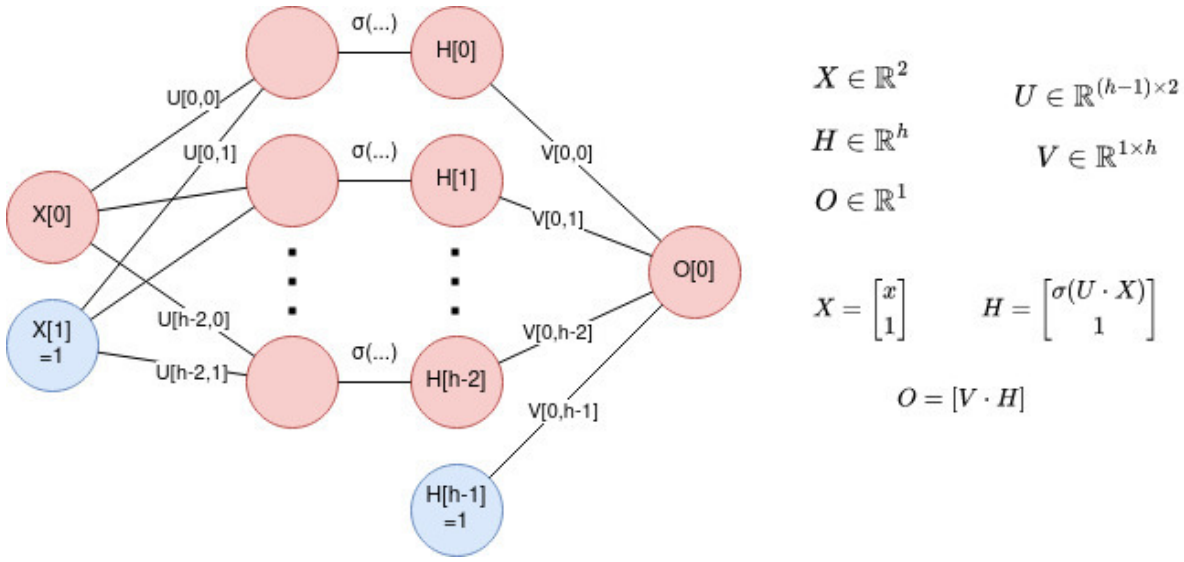



Figure 2: A representation of the basic neural network. The bias units are shown in blue.

hidden layer.

Thus, our small neural network consists of the following:

- an input layer with 2 neurons, one of which is a bias unit;
- a hidden layer with sigmoid activation and $h$ neurons, one of which is a bias unit;
- an output layer with linear activation and 1 neuron.

Figure 2 shows a schema of our basic neural network. The vectors $X \in \mathbb{R}^2, H \in \mathbb{R}^h$, and $O \in \mathbb{R}^1$ denote the contents of the input, hidden, and output layers respectively. The matrices $U \in \mathbb{R}^{(h-1)\times 2}$ and $V \in \mathbb{R}^{1\times h}$ contain the weights of the hidden and output layers respectively. From an input $x$ and the weights in $U$ and $V$, we can compute the output as follows:

$$X = \begin{bmatrix} x \\ 1 \end{bmatrix}, \quad H = \begin{bmatrix} \sigma(U \cdot X) \\ 1 \end{bmatrix}, \quad O = \begin{bmatrix} V \cdot H \end{bmatrix},$$

where $U \cdot X$ is a vector with $h-1$ elements and $\sigma(\dots)$ applies the sigmoid activation function to each of the elements in the vector.

## 2.4 The learning algorithm

The goal now is to find values of $U$ and $V$ that allow our network to estimate the target function $f$ as accurately as possible, that is, we want the value in $O$ to be close to $f(x)$.

In this first experiment, we will pursue this goal by applying the standard gradient descent algorithm (in later experiments, we will take a look at the stochastic, mini-batch, and other variants of gradient descent). We take the following steps:

- initialize $U$ and $V$ to small random values, then repeat the following two steps:
- compute the loss on the training set and the gradient of that loss with respect to $U$ and $V$,
- update the weights in $U$ and $V$ based on the computed gradients.

## 2.5 Computing the gradients

Read this section if you want to understand how the gradients are computed. Otherwise, you can skip it and come back to it later.

The loss on the training dataset $\mathcal{D} = \{x^{(n)}, y^{(n)}\}_{1 \leq n \leq N}$ is

$$L(U, V) = \frac{1}{2} \sum_{n=1}^{N} \left( y^{(n)} - o^{(n)} \right)^2,$$

where $o^{(n)}$ is the output computed for input $x^{(n)}$ using the weights in $U$ and $V$.

We will first consider the loss $l(U, V, x, y) = \frac{1}{2}(y - o)^2$ corresponding to a single training instance $(x, y)$ and computed output $o$.

Recall that $o = V \cdot H = V_0 H_0 + V_1 H_1 + \cdots + V_{h-1} H_{h-1}$, [2] so

$$\begin{aligned} \frac{\partial l}{\partial V_i} &= \frac{\partial}{\partial V_i} \left( \frac{1}{2}(y - o)^2 \right) = (y - o) \cdot \frac{\partial}{\partial V_i}(y - o) \\ &= (y - o) \cdot \left( -\frac{\partial o}{\partial V_i} \right) = (y - o) \cdot (-H_i) = (o - y) \cdot H_i \end{aligned}$$

Using vector notation we can write

$$\boxed{\frac{\partial l}{\partial V} = (o - y) \cdot H^T} \tag{1}$$

(where we transpose $H$ because it is a column vector while $V$ is a row vector).

Before we compute the gradient with respect to $U$, notice that

$$H = \begin{bmatrix} \sigma(U \cdot X) \\ 1 \end{bmatrix} = \begin{bmatrix} \sigma(U_0 \cdot X) \\ \sigma(U_1 \cdot X) \\ \vdots \\ \sigma(U_{h-1} \cdot X) \\ 1 \end{bmatrix} = \begin{bmatrix} \sigma(U_{0,0} X_0 + U_{0,1} X_1) \\ \sigma(U_{1,0} X_0 + U_{1,1} X_1) \\ \vdots \\ \sigma(U_{h-1,0} X_0 + U_{h-1,1} X_1) \\ 1 \end{bmatrix},$$

[2] We use indexes starting from 0 and not 1 for consistency with the python code, so you can better see the parallels between the formulas in this document and the corresponding lines of code in the notebook.

where $U_i$ is the $i$-th row of the matrix $U$ and $U_{i,j}$ is the element on $i$-th row and $j$-th column. So, $H_i$ is the only element of $H$ that depends on weight $U_{i,j}$, thus

$$\frac{\partial l}{\partial U_{i,j}} = \frac{\partial l}{\partial H_i} \cdot \frac{\partial H_i}{\partial U_{i,j}}$$

Similarly to the derivation above, we have

$$\frac{\partial l}{\partial H_i} = (y - o) \cdot \left( -\frac{\partial o}{\partial H_i} \right) = (y - o) \cdot (-V_i) = (o - y) \cdot V_i$$

And, since $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$, we have

$$\begin{aligned}
\frac{\partial H_i}{\partial U_{i,j}} &= \frac{\partial}{\partial U_{i,j}}(\sigma(U_i \cdot X)) \\
&= \sigma(U_i \cdot X) \cdot (1 - \sigma(U_i \cdot X)) \cdot \frac{\partial}{\partial U_{i,j}}(U_i \cdot X) \\
&= H_i \cdot (1 - H_i) \cdot \frac{\partial}{\partial U_{i,j}}(U_{i,0} X_0 + U_{i,1} X_1) \\
&= H_i \cdot (1 - H_i) \cdot X_j
\end{aligned}$$

Combining the three formulas above:

$$\frac{\partial l}{\partial U_{i,j}} = \frac{\partial l}{\partial H_i} \cdot \frac{\partial H_i}{\partial U_{i,j}} = (o - y) \cdot V_i \cdot H_i \cdot (1 - H_i) \cdot X_j$$

Using vector notation we can write

$$\boxed{\frac{\partial l}{\partial U_i} = (o - y) \cdot V_i \cdot H_i \cdot (1 - H_i) \cdot X^T} \tag{2}$$

Now that we have $\frac{\partial l}{\partial V}$ and $\frac{\partial l}{\partial U_i}$, we can compute $\frac{\partial L}{\partial V}$ and $\frac{\partial L}{\partial U_i}$:

$$\frac{\partial L}{\partial V} = \frac{\partial}{\partial V} \left( \sum_{n=1}^{N} l(U, V, x^{(n)}, y^{(n)}) \right) = \sum_{n=1}^{N} (o^{(n)} - y^{(n)}) \cdot \left( H^{(n)} \right)^T$$

$$\frac{\partial L}{\partial U_i} = \sum_{n=1}^{N} (o^{(n)} - y^{(n)}) \cdot V_i^{(n)} \cdot H_i^{(n)} \cdot \left( 1 - H_i^{(n)} \right) \left( X^{(n)} \right)^T,$$

where the superscripts $^{(n)}$ denote the values corresponding to the sample $(x^{(n)}, y^{(n)})$.

Equations 1 and 2 are used in the function `back_propagate(e)`, which takes the error $e = o - y$ on one sample as input and computes the gradients of the loss $l$. The gradients of the loss $L$ are then obtained by summing up the gradients obtained for each sample. To make the loss and the size of the updates independent of the size of the dataset, we divide the loss and the gradients by the size of the dataset. In this way, we can experiment with different sizes of the dataset while keeping the same learning rate (otherwise, multiplying the size of the dataset by 1000 would lead to very large loss and updates, which would make the learning process unstable).

# 3 The notebook

The notebook `1_basic_neural_network.ipynb` implements all elements of previous section in Python. It also contains code to visualize the dataset, the evolution of the error, the difference between the target function and the function that the network has learned, and the evolution of the weights.

**About the figures.** Notice that the plot of the evolution of the error is a semi-log plot: the iterations are put on a linear scale, while the error is put on a logarithmic scale (which corresponds to a sort of zoom on the small values). Also note that the scales of the vertical axes of all plots of the error and plots of the weights are different. To compare those plots you have to look at the values on their vertical axes.

**About the randomness.** In the following section, we will run the notebook many times and observe the effect of varying some parameters. To correctly analyze the effect of varying the learning rate (for example), we need to isolate it from other effects: we need the executions that we want to compare to have their learning rate as only difference. In the notebook, we use random numbers two times: to generate the datasets, and to generate the initial weights $U$ and $V$. These random numbers cause the results to vary a lot from one execution to another. If we want to observe the impact of other parameters, we need to use the same random numbers each time. The file `basic_neural_network_help_functions.py` contains the following functions that are then used by the notebook:

- `get_train_set(f)` and `get_test_set(f)` return the same randomly generated datasets each time.
- `get_new_random_dataset(f, N)` returns a new dataset (of size $N$) for the given function $f$.
- `get_initial_weights(h)` returns the same initial weights $U$ and $V$ each time (the results are different for different $h$, so when you vary $h$ you also vary the initial weights).
- `get_new_random_weights(h)` returns new random initial weights.

You should use the functions `get_new_random_dataset(f, N)` and `get_new_random_weights(h)` when you want to analyze the impact of the randomness of the datasets and initial weights, and you should use the other functions when you want to analyze the impact of other parameters. In the following section and unless stated otherwise, we use the functions `get_train_set(f)`, `get_test_set(f)`, and `get_initial_weights(h)` (and call their outputs the "standard datasets and standard initial weights"). Moreover, using these functions makes the experiments deterministic and reproducible. As a result, you should be able to run the experiments of next section and get exactly the same results.

**About the execution time.** All of the notebook's cells should run almost instantly, except one: the cell containing the learning loop takes about 14 seconds to run 1000 epochs if $h = 3$, and about 21 seconds if $h = 8$. Of course, the execution time of that cell is approximately proportional to the number of epochs you run and depends on your machine and many other parameters.

**Execution pitfalls.** In a notebook, you can run the Python cells in any order you want. This can result in unexpected behaviors. If you are not sure about how it works, I suggest you always run all cells in order (and only once), and restart the execution (you can do this under the "kernel" tab) each time you want to try again.

# 4 Experiments

In the following sections, we will run the notebook with different parameter values and try to interpret the results. The parameters we vary in this experiment are: the number $h$ of hidden neurons, the learning rate, the number of learning iterations (or "epochs"), and the target function $f$. To give you a better overview of the different parameter values that are tried, each tuple $(f, h, \texttt{learning\_rate}, \texttt{epochs})$ is highlighted in red. You can replicate the experiments by using these parameter values in the notebook (you should then get the same figures, except when we use new random numbers to generate the datasets and initialize the network's weights). You can also run your own experiments and try other values.

## 4.1 Experimenting with $f_1$

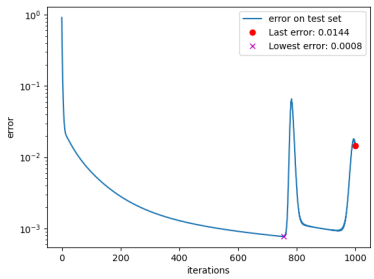
Figure 3 shows some results for $(f, h, \texttt{learning\_rate}, \texttt{epochs}) = (f_1, 3, 1, 1000)$. During the first 750 iterations (or "epochs"), the learning seems very stable (first plot of figure 3) and the network manages to learn a good approximation of the function (third plot). If we had stopped learning here, we could have thought that the learning is stable and that the following iterations would have continued to improve the network's performance. However, the learning process starts oscillating after epoch 755 and we see a shocking drop in performance (which I personally did not expect [3]). The learning oscillations are also visible in the plots of $U$ and $V$ (fourth and fifth plots). The difference between the first and last plots illustrates the impact of the randomness of the datasets and initial weights.

**A too large learning rate.** As the learning process is already unstable with a learning rate of 1 (such oscillations mean that, while doing gradient descent, we overshoot the local minima each time by taking too large steps), you can expect that bigger learning rates will lead to even less stable results. If you try $(f_1, 3, 2, 1000)$ (and use the standard datasets and initial weights), you will see that the error grows rapidly to $2,7 \cdot 10^4$. If you use $(f_1, 3, 5, 1000)$, you get an overflow error.

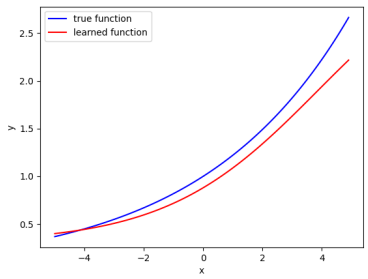
**A smaller learning rate.** We want the learning process to be more stable. So, we try a learning rate that is 10 times smaller (0.1 instead of 1) and, to compensate for the slower learning, we let the network train 10 times longer ($10^4$ epochs instead of $10^3$). Figure 4 shows the results for $(f_1, 3, 0.1, 10^4)$. You can see that the plots in figure 4 are approximately the same as the corresponding plots of figure 3, minus the instability.

**The learning rate problem.** The previous paragraphs illustrate the complexity of choosing a good learning rate: a bigger learning rate (here 1) can reach a good result much faster but has stability issues, while a smaller learning rate (here 0.1) solves the stability issues but at the cost of time performance. Moreover, the quality of a learning rate depends on the characteristics of the loss landscape (the surface of the function $L(U, V)$), and those characteristics might differ depending on the dimension (which particular weight we are considering), which region of the landscape we are in, etc. This could explain the unexpected oscillations of figure 3: it could be that, at that point, the learning algorithm enters a part of the loss landscape that is more sensitive to fluctuations of the weights (the

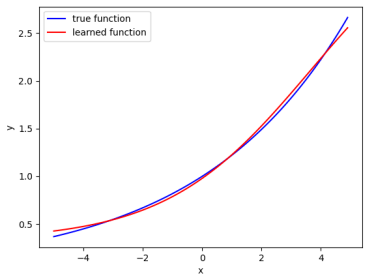
[3] Some of you may think that this is **overfitting**. It is not. You can see this by plotting the error on the training set (you can do this by uncommenting the corresponding line in the notebook), as was done in figure 9 (in the appendix). There is no overfitting here because the training set is very dense (and, in addition, our very small neural network has a limited expressivity), so a good performance on the training set almost certainly entails a good performance on the test set. We will look at overfitting in another notebook.



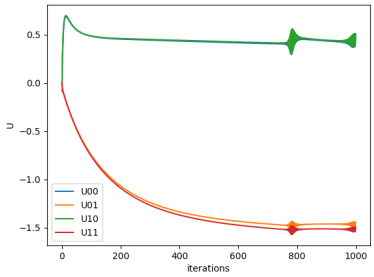

(a) Evolution of the error



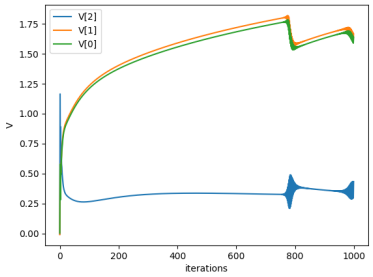

(b) Learned function (1000 iter.)



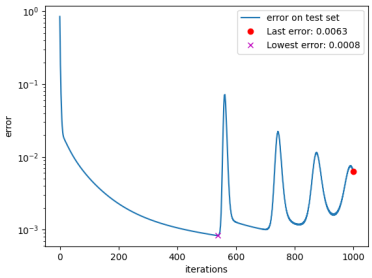

(c) Learned function (755 iter.)




(d) Evolution of $U$

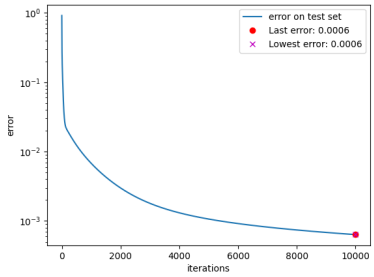



(e) Evolution of $V$

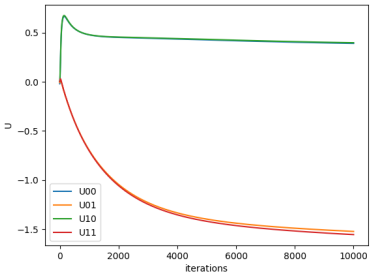



(f) Evolution of the error - new dataset and initial weights

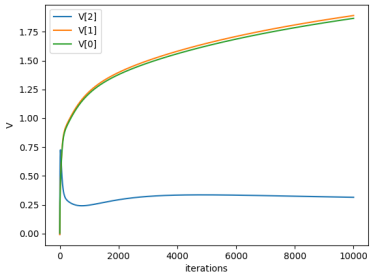
Figure 3: Results for $(f, h, \texttt{learning\_rate}, \texttt{epochs}) = (f_1, 3, 1, 1000)$. The first five plots belong to one execution: the first plot (semi-logarithmic) shows the evolution of the error during training; the second plot shows what the network has learned at the end of the execution; the third plot shows the best performance of the network during training (which was after 755 iterations); and the fourth and fifth plots show the evolution of the network's weights. The last plot (semi-logarithmic) shows the evolution of the error, when newly randomly generated datasets and initial weights were used.




(a) Evolution of the error




(b) Evolution of U




(c) Evolution of V

Figure 4: Results for $(f, h, \texttt{learning\_rate}, \texttt{epochs}) = (f_1, 3, 0.1, 10^4)$. The plot of the learned function is not shown as it is very similar to the third plot of figure 3.

curves of the loss are steeper), which leads to larger gradients, which require a smaller learning rate. To cope with this difficulty, you can use more advanced techniques that e.g. adapt the learning rate during training, or that use momentum (which accelerates learning *and* reduces the oscillations).

On figure 4, you can see that $(U_{0,0}, U_{0,1}, V_0) \approx (U_{1,0}, U_{1,1}, V_1)$ i.e., the weights coming in/out of the the first hidden neuron ($H_0$) are the same as the weights coming in/out of the second hidden neuron ($H_1$) i.e., the first and second hidden neurons are doing exactly the same thing (they have the same value and contribute the same amount to the output)! This is a strong indication that we only need one of these two neurons: we could simply remove $H_1$ and double $V_0$ (to compensate) and obtain the same output.

**A different number of neurons.** Figure 5 compares the results of $(f_1, 2, 0.1, 1000)$, $(f_1, 3, 0.1, 1000)$, and $(f_1, 5, 0.1, 1000)$. As expected, the performance does not drop while using 2 hidden neurons instead of 3, it even slightly improves. And indeed, we see that $V_0$ almost doubles to compensate for the loss of the second hidden neuron. The plots of $U_{0,0}, U_{0,1}, V_{h-1}$, while keeping their shape when going from $h = 3$ to $h = 2$, get an increased magnitude. We see the opposite trend while going from $h = 3$ to $h = 5$: the parameter plots keep the same shape (and again, all hidden neurons except the bias unit learn almost exactly the same thing!) but get a decreased magnitude. The magnitude of $V_{h-1}$ (the weight corresponding to the hidden bias unit) is also impacted, but to a lesser degree.

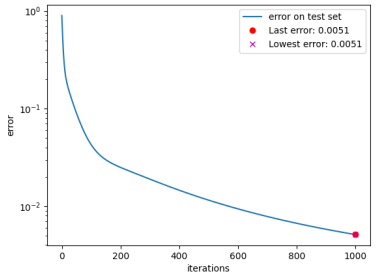
**Performance drop.** The error after 1000 iterations is 0.0051 when $h = 2$ (and $f = f_1$ and a learning rate of 0.1), 0.0068 when $h = 3$, 0.0100 when $h = 5$, and 0.0121 when $h = 8$. That is, when the model complexity is higher, the learning process needs more iterations to achieve the same results. Moreover, each iteration is more expensive, as there are more weights, hence more computations and more updates.

## 4.2 Experimenting with $f_2$

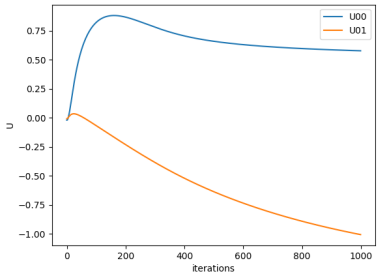
Figure 6 shows some results for $(f_2, 3, 0.1, 1000)$, $(f_2, 6, 0.1, 1000)$, and $(f_2, 8, 0.1, 1200)$. We observe the following:

- While the parameter values $(h, \texttt{learning\_rate}, \texttt{epochs}) = (3, 0.1, 1000)$ gave good results for $f_1$, they completely fail to learn $f_2$. By looking at the plot of the error (which shows quick learning in the beginning and then stagnation), we conclude that the model complexity given by 3 hidden neurons is simply too low to learn $f_2$ (underfitting). Figure 6 shows that increasing $h$ drastically improves the results.
- While a learning rate of 0.1 gave stable learning of $f_1$, here we need an even smaller learning rate.
- For $h = 8$, if you look only at the first 300 iterations of the evolution of $U$, several weights seem to converge to the same value: $U_{0,0} = U_{2,0} = U_{3,0}$ and $U_{0,1} = U_{2,1} = U_{3,1}$ (while $U_{1,0}$ and $U_{1,1}$ converge to the opposite values). So, if we had stopped learning after 300 epochs, we could have thought that, similarly to what we got with $f_1$, several hidden neurons converge to the same value and that it is not useful to keep them all. But then the values start to differ completely![4]

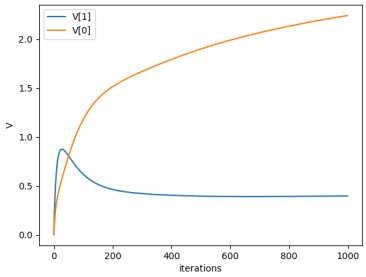
[4] As with the error plot, we learn that seeing the first hundreds or thousands of iterations does not make the following iterations predictable: a stable error can start to oscillate, and converging parameter values can start to diverge.



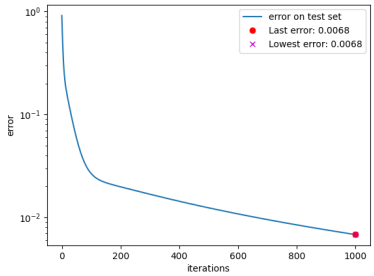

(a) Evolution of the error ($h = 2$)



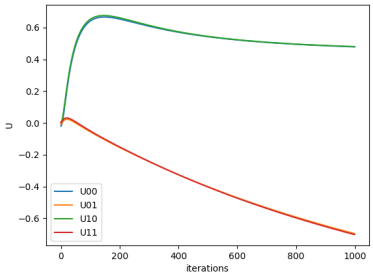

(b) Evolution of U ($h = 2$)



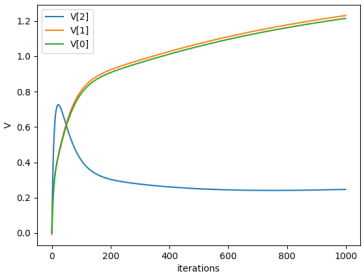

(c) Evolution of V ($h = 2$)



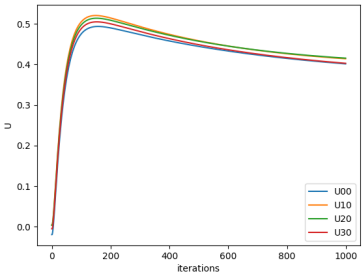

(d) Evolution of the error ($h = 3$)



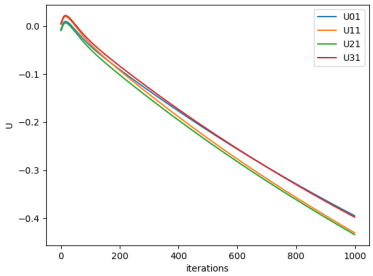

(e) Evolution of U ($h = 3$)



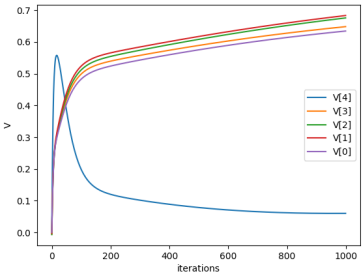

(f) Evolution of V ($h = 3$)




(g) Evolution of $U_{i,0}$ ($h = 5$)

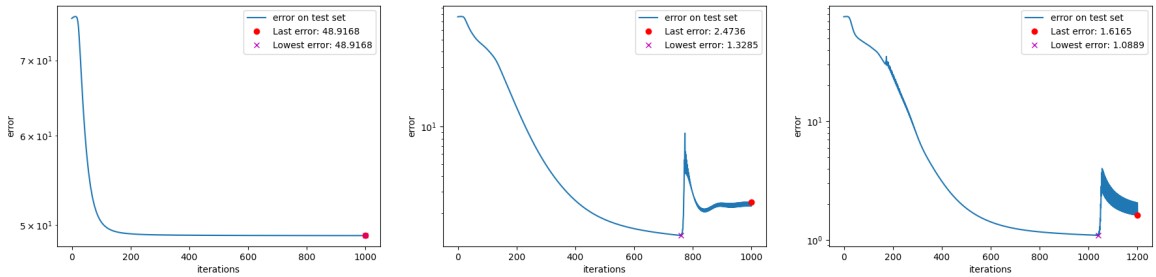



(h) Evolution of $U_{i,1}$ ($h = 5$)

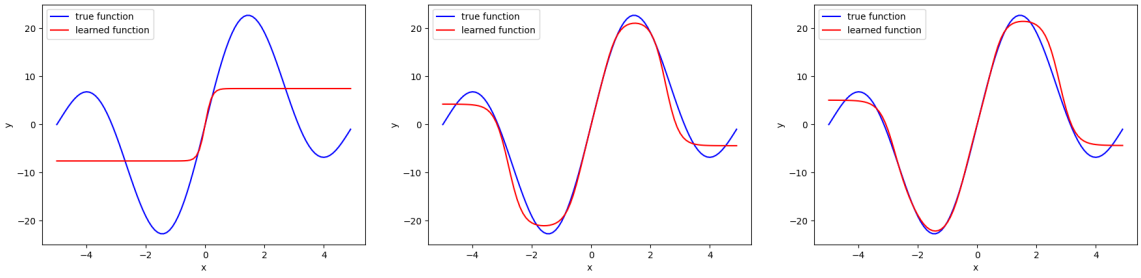



(i) Evolution of V ($h = 5$)

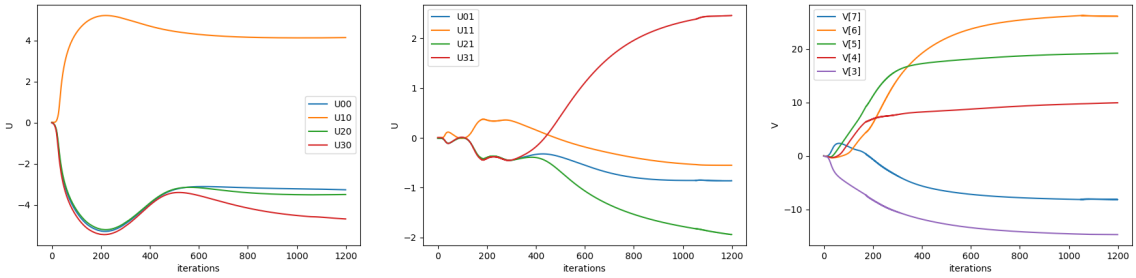
Figure 5: Comparison of the results for $(f_1, 2, 0.1, 1000)$ (top), $(f_1, 3, 0.1, 1000)$ (middle), and $(f_1, 5, 0.1, 1000)$ (bottom). For $h = 5$, we split the 8 parameters in $U$ into two plots and do not show the error plot (it is very similar to the one for $h = 3$, but with a slightly higher end error: 0.0100 instead of 0.0068). Notice that $V_{h-1}$, the parameter by which the hidden bias unit is multiplied, is $V_1$ in the third plot, $V_2$ in the sixth plot, and $V_4$ in the last plot. And notice that three middle plots show the 1000 first iterations of the corresponding plots of figure 4.




(a) Evolution of the error ($h = 3$) (b) Evolution of the error ($h = 6$) (c) Evolution of the error ($h = 8$)

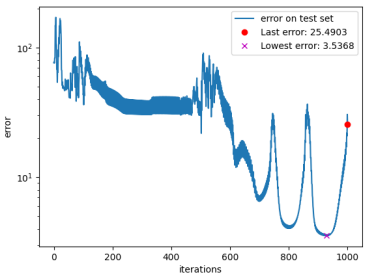



(d) Learned function ($h = 3$) (e) Learned function ($h = 6$) (f) Learned function ($h = 8$)

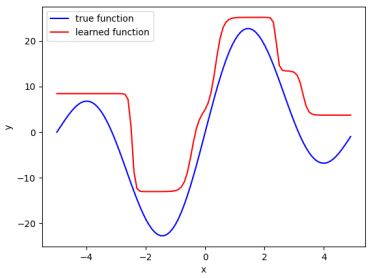



(g) Evolution of $U_{i,0}, i < 4$ ($h = 8$) (h) Evolution of $U_{i,1} i < 4$ ($h = 8$) (i) Evolution of $V_i, i > 2$ ($h = 8$)

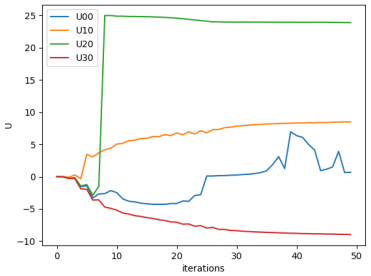
Figure 6: Results for $(f_2, 3, 0.1, 1000)$, $(f_2, 6, 0.1, 1000)$, and $(f_2, 8, 0.1, 1200)$. The evolution of the error and the learned function are shown for all three sets of parameters (pay attention to the values on the vertical axes). For the last set ($h = 8$), the evolutions of some of the network's weights (8 of the 14 parameters in $U$ and 5 of the 8 parameters in $V$) are also shown.



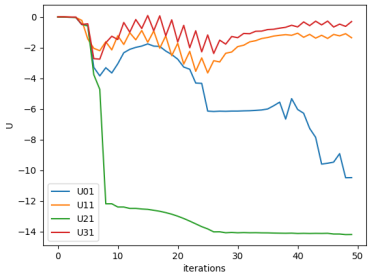

(a) Evolution of the error



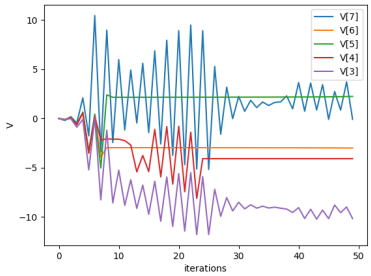

(b) Learned function (1000 iter.)




(c) Evolution of $U_{i,0}, i < 4$ $(h = 8)$




(d) Evolution of $U_{i,1} i < 4$ $(h = 8)$




(e) Evolution of $V_i, i > 2$ $(h = 8)$

Figure 7: Results for $(f_2, 8, 1, 1000)$. The plots of the evolution of the parameters show only the first 50 iterations, so you can have a closer look at the oscillations.

**A too large learning rate.** The parameters $(f_2, 8, 1.5, 1000)$ lead to divergence and an overflow error after a certain amount of iterations. Figure 7 shows the results for $(f_2, 8, 1, 1000)$ which, despite a high instability, do manage to learn something sensible.

**A smaller learning rate.** The parameters $(f_2, 8, 0.05, 1000)$ lead to the same learning as $(f_2, 8, 0.1, 1000)$ (which was shown in figure 6) except that it is stable (the curves are smooth) and 2 times slower. If we run $10^4$ epochs ($(f_2, 8, 0.05, 10^4)$), we get an error plot that is similar to the first plot of figure 6: after a consequent learning, the error stagnates (from iteration 8000 to iteration 10000, the error improves by less than 0.1%), in this case with an error of 1.0156 (which is not much better than the error achieved by $(f_2, 8, 0.1, 1200)$ before it oscillates: 1.0889). This suggests that, like $h = 3$ was, $h = 8$ is not enough to fully express $f_2$. I encourage you to try higher values for $h$ (e.g., you halve the end error by using $(f_2, 12, 0.05, 10^4)$).

**The impact of the randomness of the datasets and initial weights.** Figure 8 shows two new executions of $(f, h, \texttt{learning\_rate}) = (f_2, 8, 0.1)$ where other randomly generated datasets and initial weights are used, $(f_2, 8, 0.1, 1200, R1)$ and $(f_2, 8, 0.1, 1000, R2)$, which you can compare with the plots for $(f_2, 8, 0.1, 1200)$ in figure 6. We observe the following:

- The second and third plots of figure 8 (which show the poor performance of $(f_2, 8, 0.1, 1000, R2)$) illustrate that the results can vary a lot from one execution to another.
- The evolution of $V$ is very similar for $(f_2, 8, 0.1, 1200, R1)$ and $(f_2, 8, 0.1, 1200)$: $V_7$ and $V_5$ have similar evolutions for both, while the evolution of $V_3$ in the first case looks like the evolutions of $V_4$ and $V_6$ in the second case.
- The evolution of $U$ is very similar for $(f_2, 8, 0.1, 1200, R1)$ and $(f_2, 8, 0.1, 1000, R2)$, except that the different parameters exchanged there roles and that the sign was often switched (from negative to positive or vice-versa).

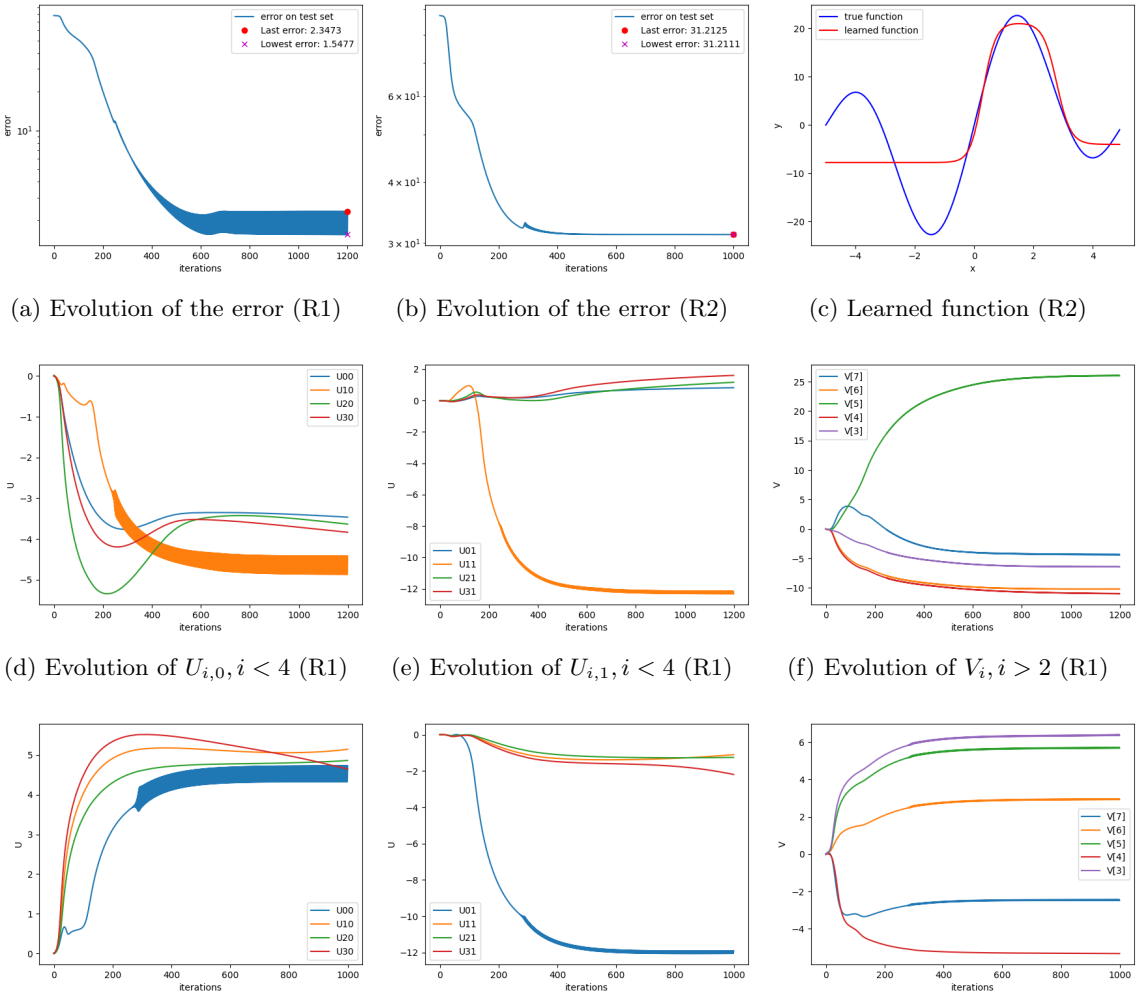
If you try to use the standard datasets but new initial weights, or new datasets with the standard initial weights, you will notice that the randomness of the datasets seems to have more impact than the randomness of the initial weights[5], and the impact is significantly higher when we use new datasets *and* new initial weights. This comforts us into thinking that we have a good initialization of the weights (as they don't seem to impact the performance that much), but highlights that we should be extra careful about the quality of the datasets.

# 5 Conclusions

In this document, we observed the following:

1. Even small neural networks can exhibit unexpected learning behaviors and the first hundreds or thousands of iterations of the learning process do not make the following iterations predictable.
2. Parameters that work well for one problem (e.g., the parameters $(h, \texttt{learning\_rate}) = (3, 0.1)$ for learning $f_1$) may not work at all for another problem (e.g., learning $f_2$), so parameters like the size of the network and the learning rate need to be adapted for each new task.

[5]Where we always initialized the weights to small random values. If we would initialize them to bigger random values, the impact of their randomness would be bigger.




(a) Evolution of the error (R1) (b) Evolution of the error (R2) (c) Learned function (R2)

(d) Evolution of $U_{i,0}, i < 4$ (R1) (e) Evolution of $U_{i,1}, i < 4$ (R1) (f) Evolution of $V_i, i > 2$ (R1)

(g) Evolution of $U_{i,0}, i < 4$ (R2) (h) Evolution of $U_{i,1}, i < 4$ (R2) (i) Evolution of $V_i, i > 2$ (R2)

Figure 8: Some results for two new executions of $(f, h, \texttt{learning\_rate}) = (f_2, 8, 0.1)$ that use new random numbers to generate the datasets and initial weights. We refer to these executions as $(f_2, 8, 0.1, 1200, R1)$ and $(f_2, 8, 0.1, 1000, R2)$ (where "R" stands for "random").

3. **Sudden instability of the learning process.** For both target functions, we see that some learning rates lead to a learning process that is stable at first, but then suddenly starts to **oscillate**. The quality of the learning rate changes when we move on the loss landscape, and the learning rate that performed well at the beginning of the learning process leads to instability later on. We need an **adaptive learning rate**.

4. By dividing the learning rate and multiplying the number of epochs accordingly, we get the same learning curves but without the instability, which improves the predictive performance but at the cost of training time.

5. **Number of neurons** and **Neuron duplicates**

   (a) For $f_1$, we see that all hidden neurons (except the bias unit) learn the same thing, so one of them (plus a bias unit) is enough. More neurons need more iterations to achieve the same performance, and those iterations are more expensive.

   (b) For $f_2$, the model complexity provided by 3 hidden neurons is not enough, and we obtain much better results by using 8 hidden neurons. In the beginning of the learning process, all neurons are (approximate) duplicates of each other, but at some point they start to differ. After some training, the learning process starts to stagnate while the loss is not that low, which indicates that model complexity offered by 8 neurons is still too low.

   (c) We conclude that few neurons can learn fast but have a limited potential (underfitting). More neurons learn slower (need more iterations to reach the same results), but can achieve much better results if given enough time. The size of the network is a trade-off whose optimal value depends on how much training time / resources we have available.

6. Randomness. The learning process varies a lot from one execution to another, and the randomness of the dataset seems to have more impact than the randomness of the initial weights (when the weights are initialized to *small* random values). However, we do see a similarity between the executions (e.g., similar evolutions of the weights).

7. **Performance.** While the network learns reasonable approximations of the target functions, the results are not really satisfactory. Although $h = 8$ leads to a reasonable estimate of $f_2$, it seems that $f_2$ requires a higher model complexity. This in turn requires a higher number of epochs to learn, and we conclude that our learning algorithm is not efficient, as it requires a significant amount of time to make a very small neural network learn a simple function.

In the next document (experiment 1B), we will have a closer look at the sudden instability of the learning process (point 3), at the duplicating/diverging behavior of neurons (point 5), and investigate whether our current network can achieve better performance by a better choice of the parameters (point 7).

In experiment 2, we will tackle the performance problem (point 7). We will try variations of gradient descent and improve the learning algorithm by using an adaptive learning rate (point 3).

In experiment 3, we will further investigate the sensitivity of the learning process to the quality of the dataset (point 6), for example by varying the number $N$ of training instances or by adding noise to the training dataset, and talk about overfitting and regularization.

# A Extra figures

Figure 9 shows both the training set error and the test set error for $(f_1, 3, 1, 1000)$.

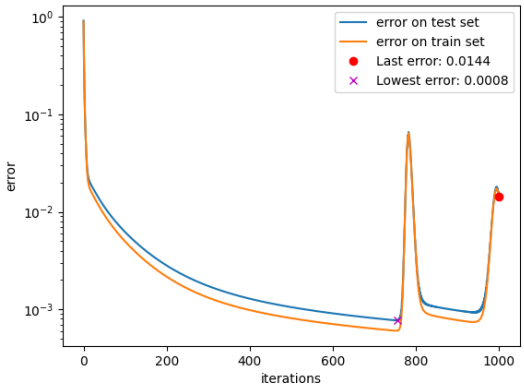



Figure 9: Same plot as the first plot in figure 3, but with the error on the training set shown as well. This figure shows that the drop in performance on the test set does not come from overfitting, since the performance on the training set follows a similar curve.