Experiment 1B: a deeper look into the basic neural network

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

Marie Peeters

May 5, 2025

Contents

1 Introduction **1**

2 Oscillations - Sudden instability of the learning process **2**

2.1 Visualizing the gradients . . . 2

2.2 Getting rid of the oscillations . . . 4

3 Number of neurons **5**

3.1 Duplicating/diverging neurons . . . 5

3.2 Improving the performance . . . 7

3.3 Encouraging neurons to differentiate . . . 8

4 Conclusion **9**

1 Introduction

In experiment 1A, we witnessed the following behaviors:

- A stable learning process suddenly starts to oscillate, and the test/training error grows rapidly.
- Reducing the learning rate seemingly removes the instability, but at the cost of training time.
- Neurons are often duplicates of each other (they learn the same weights), but then sometimes start to diverge completely.
- The performance of the network is suboptimal, seemingly due to underfitting.

In this document, we will further analyze those behaviors. Section 2 looks at the instability of the learning process, which is visualized in the second part of the notebook. Section 3 further reasons about the number of neurons and the resulting behaviors and tries to improve the final performance. Section 4 concludes this experiment.

2 Oscillations - Sudden instability of the learning process

2.1 Visualizing the gradients

In this section, we will consider $(f, h, \texttt{learning\_rate}, \texttt{epochs}) = (f_1, 3, 1, 1000)$ and $(f_2, 8, 0.1, 1200)$, both of which show a learning process that is first stable and then suddenly unstable.

The first plot of figure 1 shows the evolution of the error for $(f_1, 3, 1, 1000)$, which we already saw in the previous document. The learning process is stable and the error continuously decreases, until iteration ~ 750 where it suddenly increases. The last two plots of figure 1 show the evolution of the weights, which oscillate when the error is unstable. This behavior means that the gradient descent algorithm overshoots the local minima each time by taking too large steps. The learning rate that gave a stable learning during iterations 0-750 is now too big and leads to an unstable learning process with too large updates of the weights (remember that the updates are equal to the gradient multiplied by the learning rate). Plots (d) and (g) show iterations 730-760 of the evolution of the error.

To further visualize this instability, we look at the norm of the gradient (which is proportional to the norm of the updates) and at the angles between consecutive gradients (this angle tells us whether the gradient descent algorithm keeps going in the same direction, slowly converging to a local minima, or whether it constantly changes direction).

If the gradient $\frac{\partial l}{\partial w}$ is the vector that contains the derivatives of the loss with respect to each of the parameters in U and V, then the norm of the gradient is

$$\left\| \frac{\partial l}{\partial w} \right\| = \sqrt{\sum_{d \in \frac{\partial l}{\partial w}} d^2}.$$

In the notebook, this norm is calculated from the matrices $\frac{\partial l}{\partial U}$ and $\frac{\partial l}{\partial V}$:

$$\left\| \frac{\partial l}{\partial w} \right\| = \sqrt{\sum_{d \in \frac{\partial l}{\partial U}} d^2 + \sum_{d \in \frac{\partial l}{\partial V}} d^2}.$$

The cosine of the angle between consecutive gradients G_i and G_{i+1} (corresponding to the iterations i and $i+1$) is given by

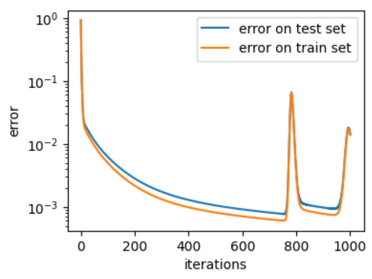
$$\alpha = \frac{G_i^T \cdot G_{i+1}}{\|G_i\| \cdot \|G_{i+1}\|}$$

(the scalar product of the gradients divided by their norms). We then compute the arccos to get the angles.

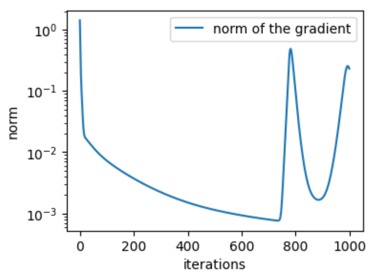
The four remaining plots in figure 1 visualize the gradients. We see the following

- first the angle grows: the updates are not aligned anymore
- then the norm of the gradient grows: the updates become bigger and bigger
- and finally the error grows: the updates lead us to zones of the loss landscape that have a higher loss.

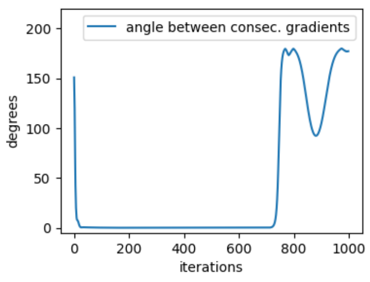
The same order holds for $(f_2, 8, 0.1, 1200)$, which leads to similar plots, despite the fact that the target function and network size are different.



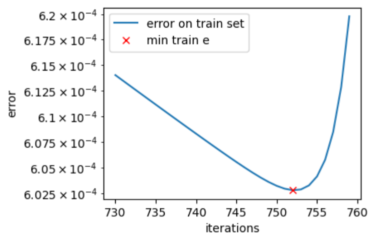
(a) Evolution of the error



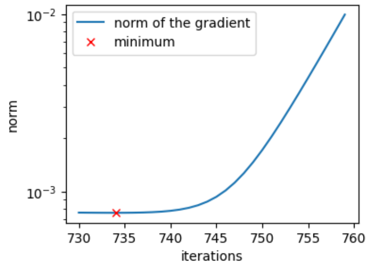
(b) Norm of the gradient



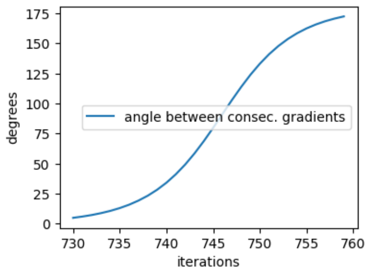
(c) Delta gradient angle



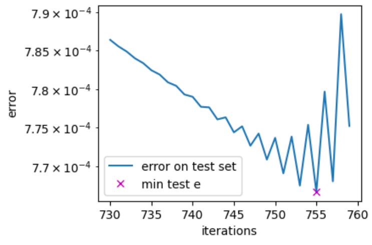
(d) Zoom on training error



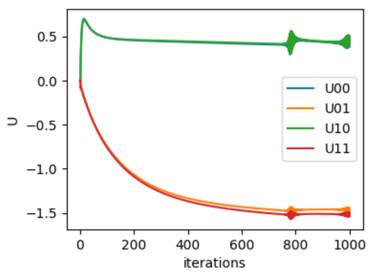
(e) Zoom on gradient norm



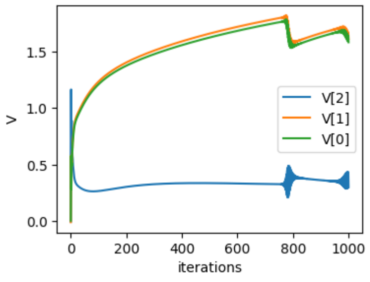
(f) Zoom on gradient angles



(g) Zoom on test error

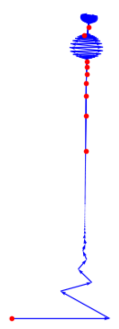


(h) Evolution of U

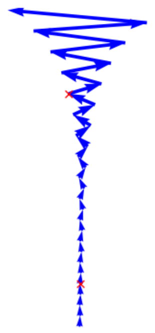


(i) Evolution of V

Figure 1: Results for = $(f_1, 3, 1, 1000)$. Plot (a) shows the evolution of the error, plot (b) shows the evolution of the gradient's norm, plot (c) shows the angles between consecutive gradients, plot (d)-(g) show zoomed versions (which show iterations 730 to 760 instead of 0 to 1000) of the first 3 plots, and plots (h) and (i) show the evolution of the weights.



(a) 2D gradient visualization



(b) Zoom on 2D gradient visualization

Figure 2: 2D visualization of the gradient's norm and angles for $(f_1, 3, 1, 1000)$. This visualization is made by plotting consecutive arrows whose lengths are determined by the norm of the gradient at each iteration, and whose directions respect the computed angles between consecutive gradients, thus producing a 2D visualization of the 7-dimensional gradient. The left plot shows iterations 0 to 1000 and contains one red dot for each 100 iterations. The right plot shows iterations 730 to 760 and the red crosses highlight where the gradient's norm and the training error are minimal.

Figure 2 shows a 2D visualization of the gradients' norms and angles together.

Run the dynamic visualization code on you computer. This will animate the plots and, at the end of the animation, you can easily zoom and have a better look at the plots. Try $(f_1, 3, 1, 1000)$ and $(f_2, 8, 0.1, 1200)$. Alternatively, you can watch the two videos that show those animations.

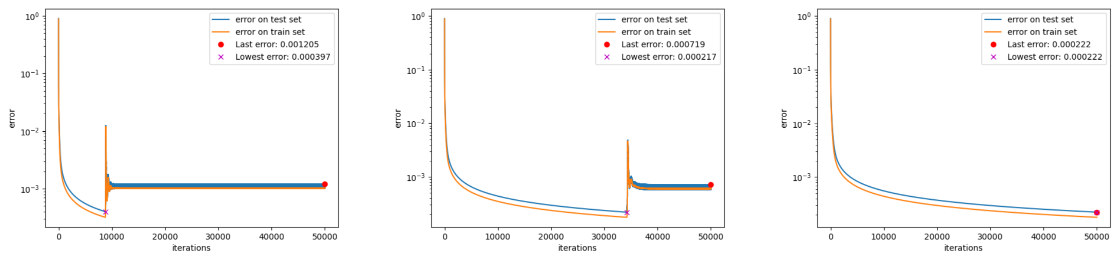
Now we reason further about figures 1 and 2. For hundreds of iterations (iterations 30 to 710), consecutive updates in the gradient descent are almost perfectly aligned, while the norm of the gradient continuously decreases. This indicates that the updates are too small. Then, the updates start to constantly change direction, the gradient's norm gets 600 times bigger, and the loss increases rapidly, which means that the updates are way too big. This shows an extreme heterogeneity of the loss landscape, despite the fact that our target function and network are very simple!

A single learning rate clearly does not fit the entire loss landscape. If we were to now consider a real-world problem and a network with millions of parameters, we could expect the million-dimensional loss landscape to be even more heterogeneous.

Common ways to cope with this are to gradually decrease the learning rate during training, use momentum, or other techniques like Adam. Another idea, which comes from the above observations (an angle growing high announces that the gradient's norm and the loss are about to increase brutally), is to increase the learning rate when the angle remains very small for a certain number of iterations and decrease it when the angle grows too much.

2.2 Getting rid of the oscillations

In experiment 1A, we compared the executions of $(f_1, 3, 1, 10^3)$ and $(f_1, 3, 0.1, 10^4)$ (and something similar for f_2) and noticed that dividing the learning rate and multiplying the number of epochs



(a) learning rate 0.4 (b) learning rate 0.3 (c) learning rate 0.2

Figure 3: Error plots for $(f, h, \texttt{learning\_rate}, \texttt{epochs}) = (f_1, 2, 0.2, 50 \cdot 10^3)$, $(f_1, 2, 0.3, 50 \cdot 10^3)$, and $(f_1, 2, 0.4, 50 \cdot 10^3)$.

accordingly leads to the same plots but without the instability, so it removes the instability at the cost of training time.

But is the instability really gone? One thing we learned during the previous experiment is that the first iterations do not make the following iterations predictable, so it is not because we don't see instability that it does not happen later during the training.

Figure 3 shows that for $(f, h) = (f_1, 2)$, reducing the learning rate from 0.4 to 0.3 moves the start of the instability from iteration ~ 9000 to iteration ~ 34000, so dividing the learning rate by 1.33 slows learning by a factor of 1.33 but multiplies the period of stability by 3.77 (and not 1.33). The last plot of figure 3 does not show any instability, but the first two plots suggest that this instability exists but much later in the later in the learning process.

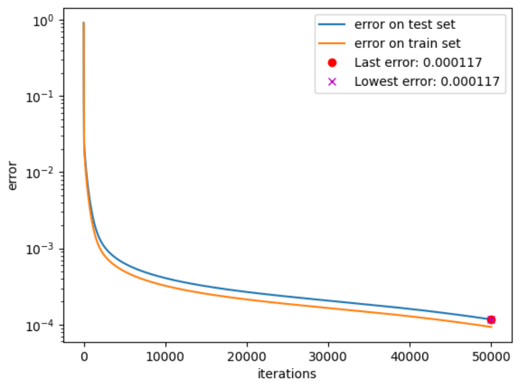
We see a similarity between the choice of the number of neurons and the choice of the learning rate: more neurons lead to slower learning but towards a better performance (if given enough time), and a smaller learning rate leads to slower learning but a much longer period of stability, which allows the learning process to achieve a better performance (if given enough time).

3 Number of neurons

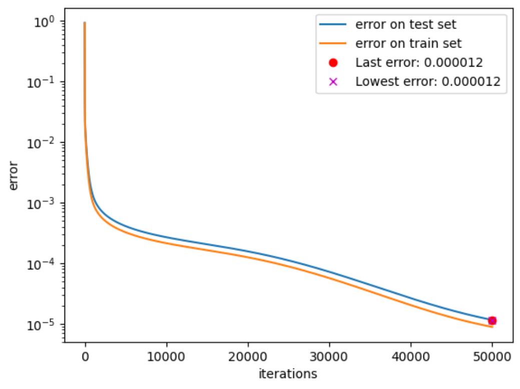
3.1 Duplicating/diverging neurons

In experiment 1A (see point 5 of the conclusion), we saw the following:

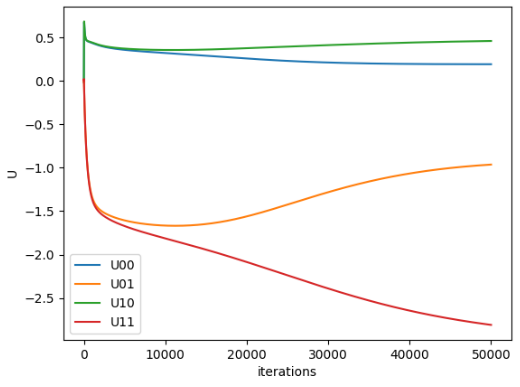
- Few neurons can learn quickly, but they have limited potential (underfitting). More neurons need more (and more expensive) iterations to reach the same result but can go further if given enough time.
- While learning f_1, all hidden neurons (except the bias unit) converge to the same value, and removing the duplicate neurons improves the network's performance.
- While learning f_2, many neurons start as duplicates of each other but then start to differ completely, and the extra complexity provided by the additional neurons is needed to approximate f_2.



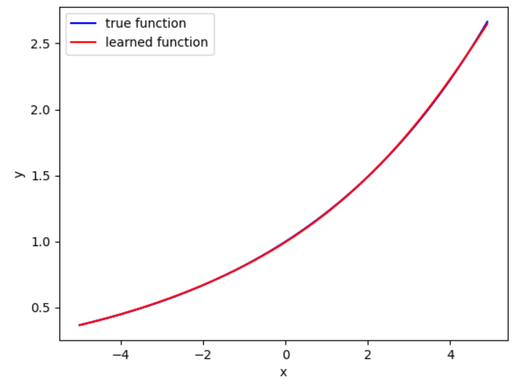
(a) Evolution of the error, learning rate 0.2



(b) Evolution of the error, learning rate 0.4



(c) Evolution of U, learning rate 0.4

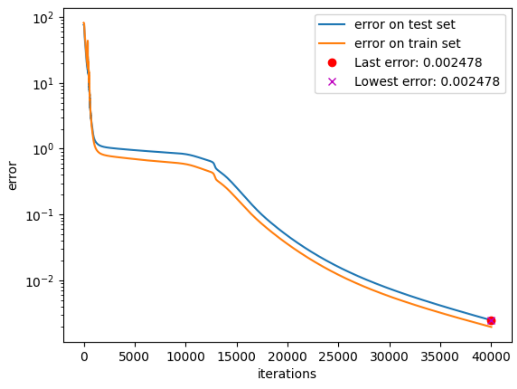


(d) Learned function, learning rate 0.4

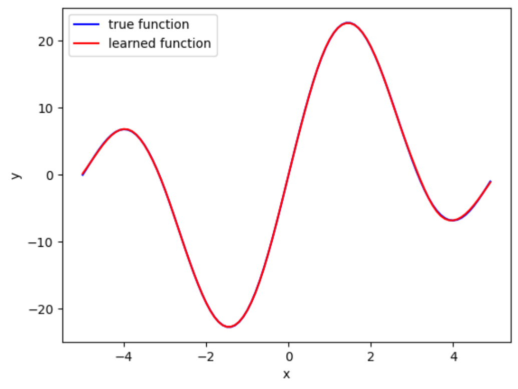
Figure 4: Plots for $(f, h, \texttt{learning\_rate}, \texttt{epochs}) = (f_1, 3, 0.2, 50 \cdot 10^3)$ and $(f_1, 3, 0.4, 50 \cdot 10^3)$.

So the question is: would duplicate neurons learning f_1 start to diverge if given much more time to learn? The best approximation of f_1 that we achieved is still far from being optimal, so maybe we need the extra neurons but have not trained long enough to see their potential.

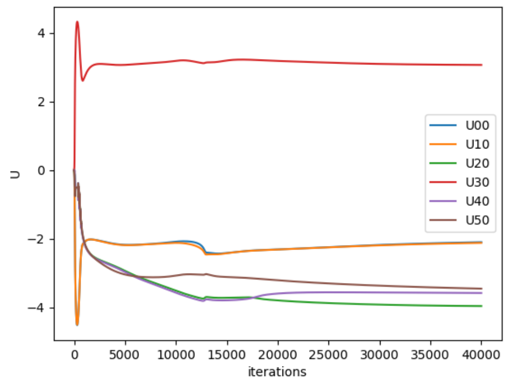
Figure 4 shows the results for $(f, h, \texttt{learning\_rate}, \texttt{epochs}) = (f_1, 3, 0.2, 50 \cdot 10^3)$ and $(f_1, 3, 0.4, 50 \cdot 10^3)$. You can compare the first plot of figure 4 (which shows that $(f_1, 3, 0.2, 50 \cdot 10^3)$ achieves a final error of $1.17 \cdot 10^{-4}$) with the last plot of figure 3 (which shows that $(f_1, 2, 0.2, 50 \cdot 10^3)$ achieves a final error of $2.22 \cdot 10^{-4}$) to see that increasing the number of neurons does improve the performance (moreover, the former is still decreasing while later stagnates). The three other plots of figure 4 show that a slightly higher learning rate leads to faster (but stable) learning and much better performance. We see that the hidden neurons indeed diverge if given enough time and we see a very satisfying approximation of the target function.



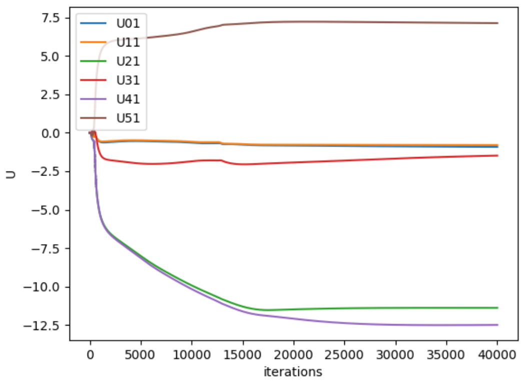
(a) Evolution of the error



(b) Learned function



(c) Evolution of U



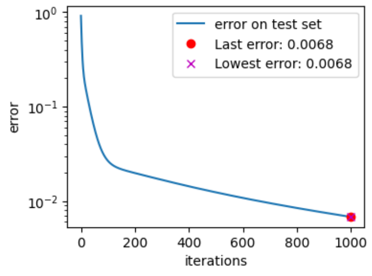
(d) Evolution of U

Figure 5: Plots for $(f_2, 20, 0.05, 40 \cdot 10^3)$.

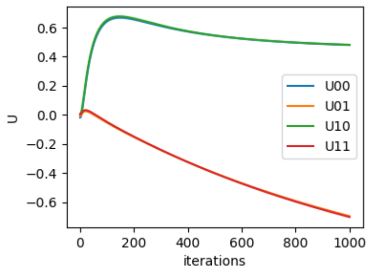
3.2 Improving the performance

We finally got a great approximation of f_1 by using one extra neuron and training for a much longer time. Can we now can a great approximation of f_2?

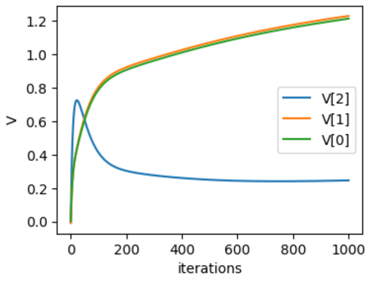
Figure 5 shows the results for $(f_2, 20, 0.05, 40 \cdot 10^3)$. We see that the error starts to decrease faster around iteration 15000, and the final approximation of f_2 is very accurate. The last two plots of figure 5 show that there are duplicate neurons (the weights $u00, u01, u10, u11$ corresponding to neurons 0 and 1 are almost the same, so the values of those two neurons are almost the same, independently of the input), so maybe we don't need that many neurons to have this performance.



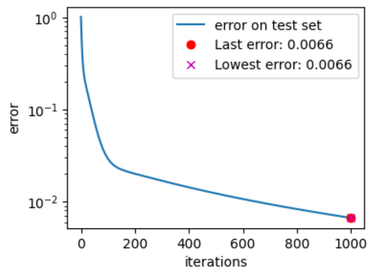
(a) Error - small initial weights



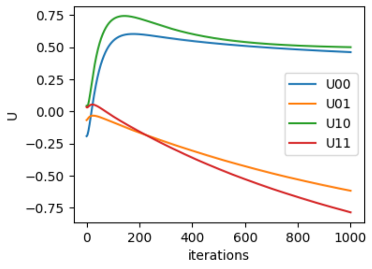
(b) U - small initial weights



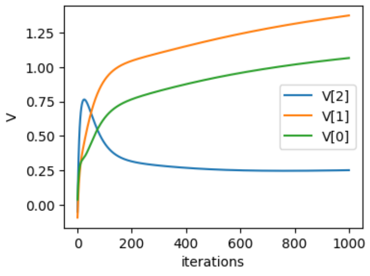
(c) V - small initial weights



(d) Error - bigger initial weights



(e) U - bigger initial weights



(f) V - bigger initial weights

Figure 6: Results for = $(f_1, 3, 0.1, 1000)$. The three top plots show the curves we are familiar with (see figures 4 and 5 in experiment 1A). The three bottom plots show the same curves, but where the initial weights were multiplied by 10, resulting in a bigger difference between the neurons.

3.3 Encouraging neurons to differentiate

In experiment 1A, we saw that having duplicate neurons seems useless, and removing the duplicate neurons slightly improves the performance. Having more neurons becomes useful when they have the time to differentiate. Can we improve the performance by making them diverge sooner? This research question deserves thorough analysis, here we will only quickly illustrate a simple idea.

Imagine that, in our toy neural network with one hidden layer, all weights corresponding to the first neuron are equal to the weights corresponding to the second neuron. The computations of both neurons and their contributions to the output (and thus to the loss) will then be the same, and the derivative of the loss with respect to a weight of the first neuron will be equal to the derivative of the loss with respect to the corresponding weight of the second neuron. So, the weights of both neurons will get the same update, and the neurons will still be equal after the update. By induction, those neurons will always have the same value, which is obviously not what we want.

Until now, we initialized the weights to very small random numbers (with mean 0 and standard deviation 0.01). Since the initial weights are approximately equal (close to 0), the reasoning above holds approximately, so it's not surprising that some weights follow a very similar learning curve in the beginning.

A simple idea is to initiate the weights to values that are more distant from each other. Here, we implement this by multiplying the initial weights by 10 (you can do this by adding the line "U *= 10; V *= 10" after the definition of U and V). Figure 6 shows an example result of this. We see that the new curves still follow a similar path, but the differences between the different neurons are much bigger. The end result is only slightly better (an error of 0.0066 instead of 0.0068).

4 Conclusion

Summary:

1. We visualized the gradients to get more insight into the unexpected oscillations.
2. We investigated whether reducing the learning rate truly removes the oscillations as it seems, or if the instability is just postponed.
3. We investigated whether the duplicating/diverging behavior of neurons witnessed while learning f_2 also happens with f_1, and concluded that this behavior happens if given enough time.
4. We found parameters that lead to much better approximations of f_1 and f_2.
5. We shortly reasoned about the reason why neurons (all in a single hidden layer) follow the same learning path and how we can try to counter it.

Points 2 and 3 above are closely related.

First, both experiments were driven by the observation that the first iterations of the learning process do not make following iterations predictable.

Second, we see similar behavior: a smaller learning rate learns slower but further (because the instability is reduced and/or happens much later), and more neurons also lead to slower but further learning (the error goes slower down but reaches lower values). The learning rate controls the oscillations and the learning time, while the number of neurons controls the expressivity and also the learning time. The available learning time then determines whether we will be able to leverage the available expressivity. More neurons need more time to achieve their potential. If not given enough time, the neurons are still duplicates of each other, so it is not really useful to have them all. If given enough time, the neurons start to differentiate, which allows to learn patterns of higher complexity.

The difference between f_1 and f_2 is that f_1 is much simpler: $h = 2$ already provides most of the model complexity that f_1 needs, but completely fails to express f_2. So, the benefit of having more neurons is much easier to see for f_2, and so we see it after a much smaller number of iterations.

The above conclusions strengthen the conclusions of experiment 1A. However, they are rather guided intuitions at this point. More experiments are needed to assert them with more certainty.