FTML practical session 10

18 mai 2025

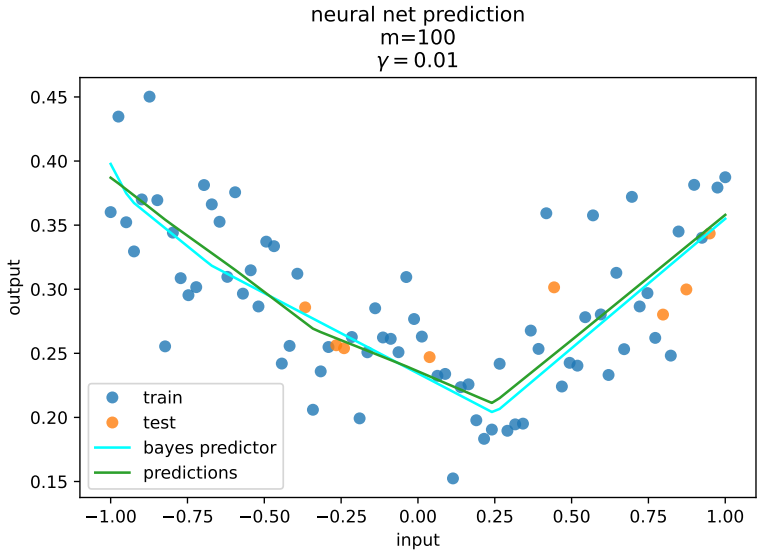

TABLE DES MATIÈRES

1 SIMPLICITY BIAS OF NEURAL NETWORKS

Introduction

This exercise assumes a basic knowledge of neural networks. We will witness that with some neural networks, it is unlikely to overfit the data and illustrate this with networks using the ReLU activation. In order to have some visual feedback, we will, use $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \mathbb{R}$ and the data to be learned are like the data displayed in 1.

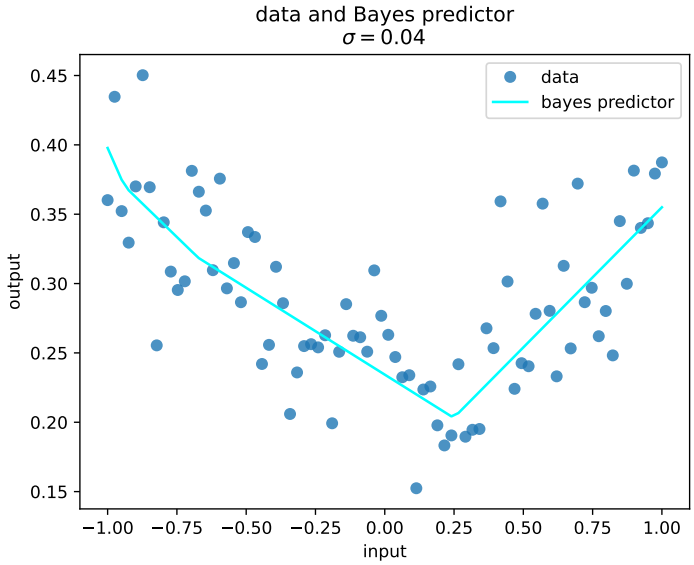

FIGURE 1 – Example target function and dataset

Let m be the number of neurons in the hidden layer, and n the number of samples in the dataset. We will see that with the specific type of networks and datasets used here, even when $m > n$ (which implies that the number of parametrers of the network is larger than n, since it is of order at least $\mathcal{O}(dm)$, d being the input dimension), no overfitting happens. This is related to the "interpolation regime", a contemporary topic in machine learning research. You can see an example of such a phenomena in figure 2.

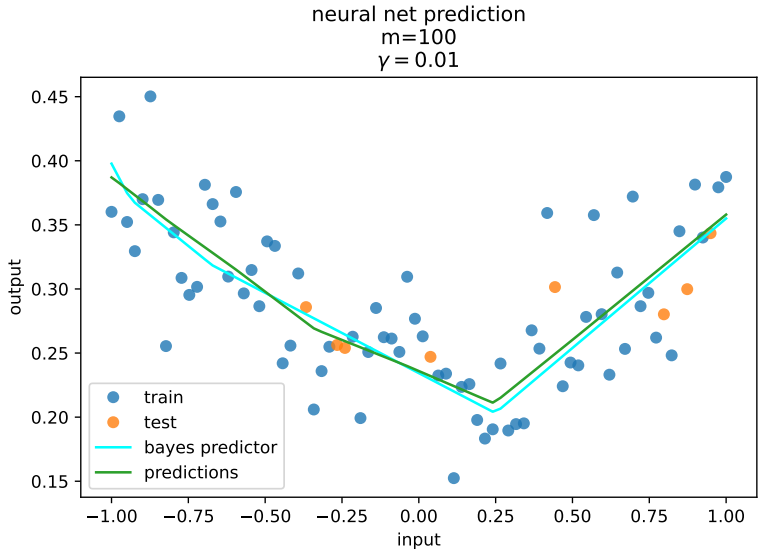

FIGURE 2 – Although the network has a high capacity (a large number of free parameters during the learning stage), it does not overfit the data.

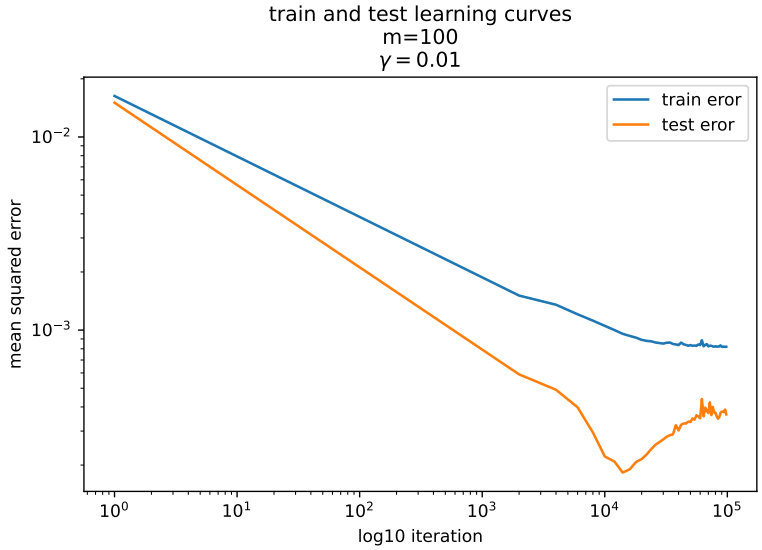

FIGURE 3 – Learning curves, same network as in figure 2

1.1 Definition of the neural network

We use the following architecture :

— $\mathcal{X} = \mathbb{R}$, $\mathcal{Y} = \mathbb{R}$, the loss is the squared loss.

— In order to add intercepts between the input layer and the hidden layer, we add a component to the inputs $x \in \mathbb{R}^d$ and build a vector $\bar{x} = (x, 1) \in \mathbb{R}^{1,d+1}$. In our case, $d = 1$. If we have n samples, we add a column of 1s to the matrix $X \in \mathbb{R}^{n,d+1}$.

— The same operation is applied to the hidden layer h, in order to add an intercept before the output. We build an "extended hidden layer" $\bar{h} = (h, 1) \in \mathbb{R}^{1,m+1}$.

— The hidden layer contains m neurons. The matrix w_h containing the weigths between the input layer and the hidden layer, is in $\mathbb{R}^{d+1,m}$ (the $d+1$ comes from the intercept)

— nonlinearity : ReLU (noted σ), applied to the hidden layer. Also applied to the output.

— pre_h is the variable obtained before applying σ, in order to compute h. The same variable pre_y is defined for y.

— The output of the network is a single real number, hence the matrix θ containing the weights between the hidden layer and the output layer is a vector in $\mathbb{R}^{m+1,1}$ (the $m+1$ again comes from the intercept).

— Finally, the output of the neural network writes :

$$\hat{y} = f(x) = \sigma(\langle \theta, \bar{h} \rangle) \in \mathbb{R} \tag{1}$$

1.1.1 *Ressources*

Useful ressources :

— https://playground.tensorflow.org/

— https://francisbach.com/quest-for-adaptivity/

— http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

— [?]

1.2 Implementation

Train neural networks with SGD (Stochastic gradient descent) with varying number m of neurons in the (unique) hidden layer, including $m > n$ (where n is the number of samples in the dataset) in order to observe the simplicity bias.

You will need to compute the gradients of the loss function with respect to the network parameters. There will be a gradient with respect to θ, noted $\nabla_\theta l(\theta, w_h)$, and a gradient with respect to w_h, noted $\nabla_{w_h} l(\theta, w_h)$. You will also need to edit the main SGD algorithm. For the implementation, you are free to either compute the gradients manually and write the computations in numpy, or to use automatic differentiation with libraries like torch of tensorflow.

1.2.1 *Python files*

The template files are more useful if you use the numpy / manual approach.

— **main.py** : main file, runs SGD on the neural network. You need to fix the algorithm.

— **utils.py** : computes the forward passes and the gradient computations. You need to fix the computations.

— **generate_data.py** : generates the data, you don't need to edit it.

1.2.2 *Initialization*

You can use the following type of initialization.

— θ is initialized uniformly in $[-\frac{1}{\sqrt{m}}, -\frac{1}{\sqrt{m}}]^{m+1}$

— Each column of w_h, that belongs to $\mathbb{R}^2$ (because $d = 1$), is initialized on the sphere of radius $\frac{1}{\sqrt{m}}$.

1.2.3 *Derivatives*

As the derivative of ReLU, you can use the heaviside function.
https://numpy.org/doc/stable/reference/generated/numpy.heaviside.html
https:/numpy.org/doc/stable/reference/generated/numpy.maximum.html

1.2.4 *Learning rate*

You will need to experiment with the hyperparameters, such as the learning rate γ in order to observe learning and the simplicity bias. As the learning algorithm and dataset generation are stochastic, you might observe different outputs.

Note that learning does not always happen, depending on the hyperparameters and dataset. In figure 4, the network has not been able to approximate the target function.

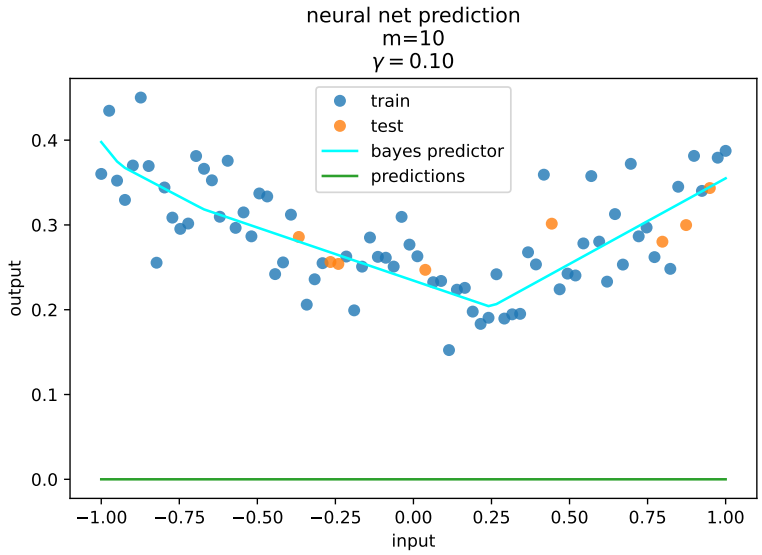

FIGURE 4 – This network has not been able to learn the data.

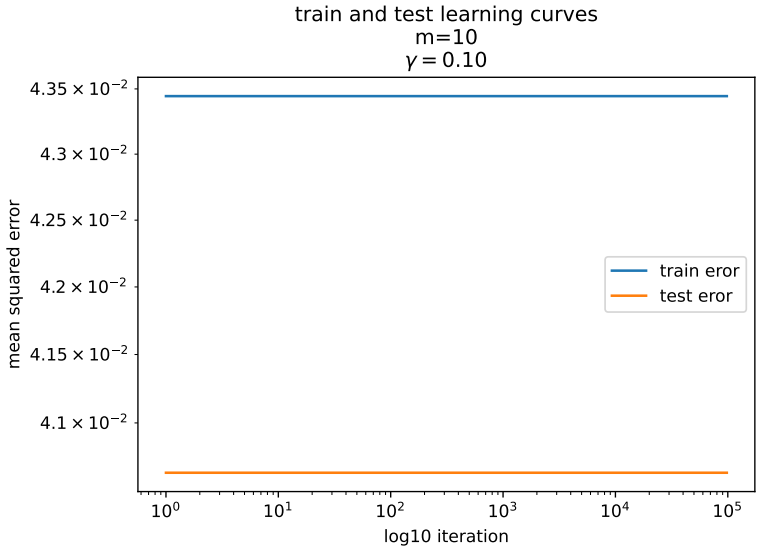

FIGURE 5 – Same network as in 4

1.3 Conclusion

These neurons tend to not overfit, although some of them have a number of parameters way larger than of the minimal space containing the target function! Note that this observation is not intuitive when we have the classical interpretation of machine learning in mind, with the famous bias-variance tradeoff, which predicts that we should observe some overfitting when the number of parameters (capacity of the model) is high compared to the number of samples.

See more in this post : https://francisbach.com/quest-for-adaptivity/