CAML - Simple NN example

Melissa Lopez

March 10, 2022

Special thanks to Armen Yeritsyan.

Environment

The neuronal network is implemented in PyCharm with a Python 3.6 interpreter, but can also be executed with jupyter. The overall programm consists of two files and uses the packages *numpy* and *matplotlib*:

- **SimpleNN.py**
  In this file the class *SimpleNN* is implemented, which contains all necessary functions for the neuronal network, like training, predicting and visualizing.
  - **init:** ***__init__(self, par_alpha, par_lambda)***
    The constructor of the class has the parameters α and λ as inputs and stores them in the class for later use.
  - **training:** ***trainNN(self, input_values, output_values, number_epochs)***
    The number of the layers and neurons is fixed (three layers with 8x3x8 neurons in total). This means that this particular implementation of a NN can only be trained with 8 different training sets, which respectively contain 8 elements (8x8 matrix). As a result the input parameters *input_values* and *output_values* must be a 8x8 matrix. The parameter *number_epochs* specifies the number of times the training sets are processed by the network.
  - **predicting:** ***predict(self, testset_x, visualize)***
    After the neuronal network is trained, the predicting function can be called. This function runs the forward propagation steps for the given inputs in *testset_x*. This input is a Nx8 matrix (N = number of test sets), which means that the number of test sets is not fixed. With the remaining *visualize* flag (True/False), it is possible to trigger the visualization function of the NN after the test sets are processed.
  - **visualization:** ***visualizeNN(...)***
    The visualization function has all values of the NN as input parameters (neurons, biases, weights) and displays them in a 2D plot. The corresponding values are coded in colors.
- **Main.py**
  In this file all training sets and test sets are specified. Those are used to train and test the NN (SimpleNN class). By default the complete training process (10000 runs) is executed for $\alpha = 0$ to 0.9 (0.1 steps) and fixed λ=0.0001. After the execution two plots are loaded and show error rates and nn-weights. The total run time is approx. 55 seconds (if the Main.py file is executed without change).

Results

Training and parameter estimation

In the present work we train our network until the error δ_3 converges to zero. Because this parameter is a vector we take the standard deviation of this vector and we plot the error of each iteration of our training. We have to iterate around 10000 times until our error converges. However, this amount of

ephocs may vary depending on how we choose our parameters α and λ. Note that we plot the standard deviation of δ_3 for each input.

Our first approach was to look for the right value of λ. We fixed our value of $\alpha = 0.6$ and varied λ in the range of [0.0000, 0.0018] since we knew that our model would only converge for small values of the regressor coefficient.

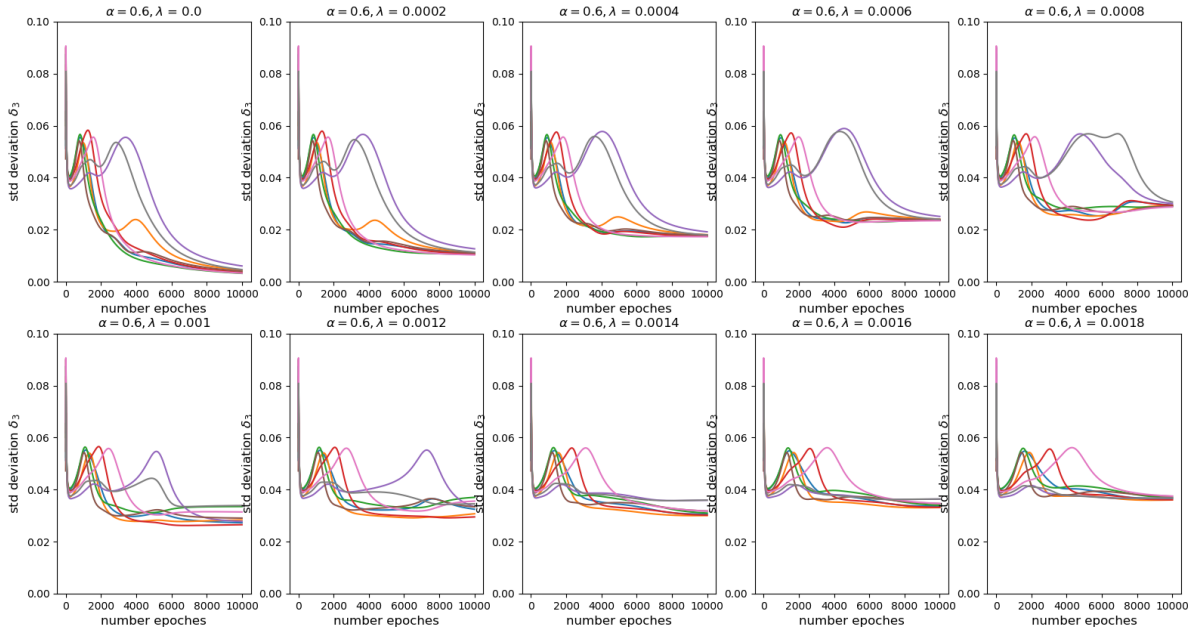

Figure 1: Convergence of the error for a fixed α and changing λ

From Fig. 1 we can see that if we increase λ our errors converge to bigger values. Then, the best value would be around λ= 0.0001. Our next step was to obtain the value of α that minimizes our errors. To this aim we fixed the value of λ=0.0001 and varied α in the range of [0, 0.9].

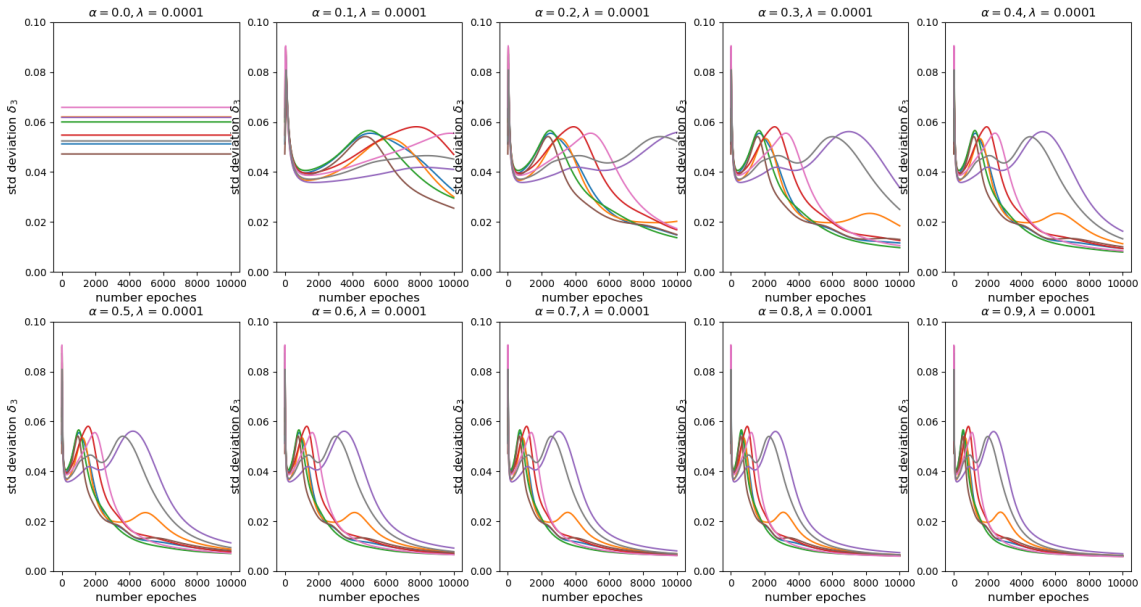

Figure 2: Convergence of the error for a fixed λ and changing α

From Fig.2 we can observe that the bigger α is the faster the error converges to zero. Despite of the

good results for large α we must take into account the fact that if we make steps too big in gradient descent we might miss the local minimum, i.e. if we make α too large our errors might not converge to zero.

To summarize, choosing a small $\lambda \approx 0.0001$, a large $\alpha \approx 0.7$ should set our convergence rate to around 9000 ephocs.

Weights and activations

In this section we will discuss the computation of the activation of each layers, the weights in between and the biases. We used the equations corresponding to the assingments' help. In Fig. 3 we can see the heat map of $a^{(l)}$ for $l = 1, 2, 3$. We define $a^{(1)}$ (left side of the figure) to be the input where the vectors are ordered at first but then they introduced randomly in our neural network. In the same way we define $a^{(3)}$ as our output that needs to be exactly the same as our input. Note that this output is only showing the probability of the one being in that position.

We initialize the weights and the biases to random value between 0 and 1 (instead of random values around 0.01) because our results were slightly improved by using this initialization. After this we can compute the activation of layer 2 and 3 in a matrix form as it follows:

$$a^{(l+1)} = f(W^{(l)}a^{(l)} + b^{(l)}) \tag{1}$$

In Fig. 3 we plotted heat maps of the activations, the weights and the biases.

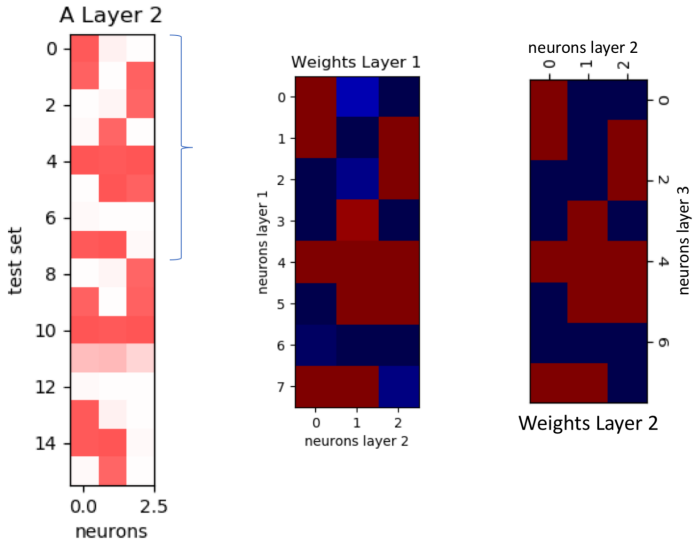

Figure 3: Heat map of activations, weights and biases.

We discovered a really interesting pattern related to $a^{(2)}$, $W^{(1)}$ and $W^{(2)}$ as we show in Fig. 4. If we invert $W^{(2)}$ and rotate it 90° we get the same pattern as $W^{(1)}$ and the part of $a^{(2)}$ that corresponds to the 'ordered' part of our input, as we have indicated with a blue curly bracket . This might mean that the weights are ordering the inputs to perform a better estimation. The estimation performed in the weights is sent to the different neurons transforming the dimensions to fit the activations.

The activation must vary between 0 and 1. We must also note that the white color of the activation means that $a^{(l)} \approx 0$ and the neuron is not active. The red color of the activation means that $a^{(l)} \approx 1$ and the neuron is active.

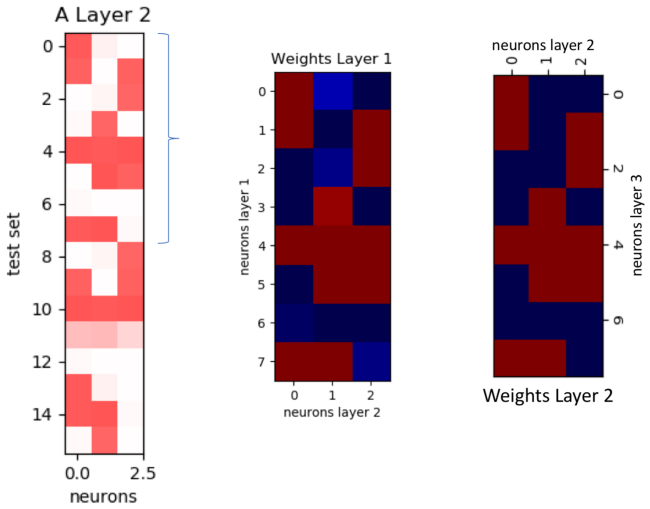

Figure 4: Pattern of $a^{(2)}$, $W^{(1)}$ and $W^{(2)}$