Experimental Report: Implementing a Neural Network from Scratch in Python

Filippo Colombi

April 22, 2025

Machine Learning for NLP

Prof. Teso Stefano and Prof. Bernardi Raffaella

1 Introduction

This work extends the experimentation from the in-class practical session and investigates a number of problems to deepen familiarity with neural network programming. This hands-on approach solidifies theoretical knowledge and extend the set of practical skills that are essential for tackling more complex NLP problems in the real-word. The experiment is particularly focused on testing the impact of different training regimes and hyperparameters tuning

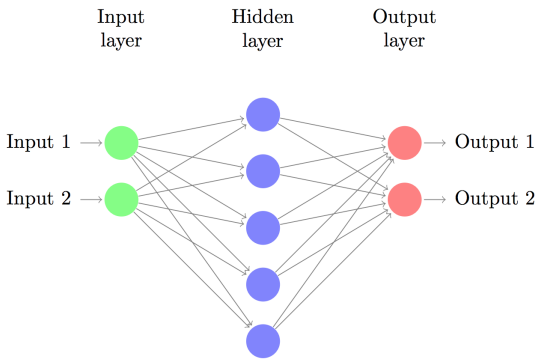

Figure 1: **Architecture of the model used:** Input Layer: Consists of 2 nodes, corresponding to the two features in the "Moons" dataset. The hidden layer contains a variable number of nodes, allowing for experimentation with different model complexities. The output layer comprises 2 nodes, one for each class in the binary classification task.

2 Data and Model

For this project, the "Moons" dataset for scikit-learn library was used. This dataset is well-suited for binary classification tasks and is often used to illustrate the behavior of different algorithms due to its clear decision boundaries and simple structure. The dataset comprises two interleaving half circles, making it an excellent choice for visualizing how well a model can classify non-linearly separable data points. The code is composed of two models:

1. A baseline logistic regression model used for comparison purposes. Logistic regression is a straightforward algorithm for binary classification tasks, making it a useful starting point for introducing classification of more complex patterns.

2. A neural network with three layers. The architecture of this neural network is shown in figure 1.

3 Experimental setup

In this section, I outline the experimental setup used to explore various aspects of a neural network implementation and optimization techniques. The experiments are based on the tasks proposed in the practical instructions (`https://dennybritz.com/posts/wildml/implementing-a-neural-network-from-scratch/`). The reason behind selecting these tasks lies in their relevance to understanding fundamental concepts in machine learning and neural networks. The implementation was based on modifying and extending the model iteratively to explore different solutions and architecture. Each task's results and discussion are carried out separately within the next paragraphs. This structured approach ensures that the insights from each task are clearly presented and understood.

3.1 Minibatch Gradient Descent

In this exercise, the objective was to implement minibatch gradient descent and compare its performance with traditional batch gradient descent in training a neural network. Initially, the script used batch gradient descent, which processes the entire dataset at once during each iteration. To ensure a fair comparison, I modified the script to incorporate minibatch gradient descent, maintaining all other hyperparameters constant.

It began with a dataset of 200 samples and a minibatch size of 32. Under these conditions, with noise set to 0.2, there was no significant difference in loss between the two methods. However, to better understand the advantages of minibatch gradient descent, especially in handling larger and noisier datasets, I increased the dataset size to 600 samples and the noise level to 0.3. The minibatch size was set at 128, a value chosen based on preliminary tests that balanced convergence speed and computational efficiency.

Moreover, to manage the increased complexity of the dataset from modifying the original setup, two neurons were added to the hidden layer, obtaining a total size of 5 neurons. This adjustment was crucial for effectively capturing the complex patterns introduced by the new chosen setup.

After running both models for 21,000 iterations, the results clearly indicated the superiority of minibatch gradient descent. It achieved a lower loss of 0.21 in just 2.51 seconds, compared to a loss of 0.29 and an execution time of 6.28 seconds for batch gradient descent. This efficiency is attributed to the fact that minibatch gradient descent leverages a more frequent update cycle and introduces variability that can help escape local minima more effectively.

This exercise highlights the practical considerations in neural network training: while minibatch gradient descent may show limited benefits on small datasets, its advantages become evident as size and complexity increase.

3.2 Learning Rate Annealing

The objective of this task was to explore the effectiveness of an annealing schedule in enhancing the training process of the neural network. An annealing schedule reduces the learning rate over time, aiming to facilitate smoother convergence and potentially improve final loss. A fixed learning rate ϵ was used for gradient descent.

Two sets of experiments to compare the outcomes with and without the annealing schedule were conducted. In both scenarios, the training was observed over 20,000 iterations with fixed loss evaluations. The results from the experiment using an annealing schedule demonstrated a significant reduction in loss over iterations, showcasing the model's ability to learn from the data. The initial loss of 0.952 rapidly decreased to 0.0305 by iteration 19,000, indicating robust learning and convergence.

On the other hand, without using an annealing schedule, while the model also exhibited a decline in loss from 0.952 to 0.031557 by iteration 19,000 the reduction was slightly less pronounced compared to the annealing schedule environment. It is not clear whether the annealing schedule actually brings any real advantages in the case of this experiment. It is challenging to evaluate solely by eye; to assess significance of these results, statistical tests such as the t-test on final loss values from multiple runs could be of help.

Overall, annealing schedule introduces complexity to the scripting process, requiring careful tuning. The implementation of an annealing schedule in neural network training introduces additional complexity that must be carefully managed. Sometimes, simpler approaches may suffice. They are easier to implement and require less adjustment. For smaller datasets or less complex models, these simpler methods might even outperform more sophisticated techniques such as annealing schedules.

3.3 Activation Function Experimentation

The network initially used a tanh activation function for the hidden layer. For this tasl, another activation was experimented with, a sigmoid function, which required changes to the backpropagation derivative.

The sigmoid outputs values between 0 and 1, making it suitable for binary classification tasks where outputs can be interpreted as probabilities. However, it is prone to the "vanishing gradeint" problem, particularly when dealing with extreme input values. This can slow down training and pose challenges in learning complex patterns. Secondly, the outputs are centered around 0.5, which may affect weight updates during backpropagation, potentially leading to delayed convergence. The Moons dataset consists of distinct clusters with non-zero centers and spans across both negative and positive values, in this regard sigmoid might not be the optimal solution.

On the other hand, tanh outputs values from -1 to 1, which helps with learning representations centered around zero. This helps weight initialization and avoids the problem that sigmoid can encounter. This symmetry can yield faster convergence and more stable learning outcomes, especially in datasets where the underlying patterns are symmetrically distributed: Moons consists of two moon-shaped clusters that are balanced and roughly mirror each other along an axis. This characteristic can be beneficial when adequately faced, *e.g.* using tanh.

By comparing these functions, I expected tanh to perform better, given its stronger gradients and zero-centered output. The results confirmed this expectation, as tanh provided more accurate training and faster convergence.

Epochs	Sigmoid Loss	Tanh Loss
0	0.526839	0.418627
1000	0.475194	0.046520
2000	0.470200	0.043340
3000	0.495330	0.041303
4000	0.272035	0.039900
5000	0.258553	0.038971
10000	0.212079	0.037463
15000	0.196228	0.037250
19000	0.191937	**0.037212**

Table 1: These results show that the tanh function converges more quickly and achieves a lower final loss compared to the sigmoid function

3.4 Extending to Three Classes

In this exercise, the network was extended from two to three classes. This involved generating an appropriate dataset for this configuration, a slight variation of the original one.

To extend the neural network from handling two to three classes, several adjustments were necessary in the implementation. Initially, the network was trained on the Moons dataset from 'sklearn.datasets', which naturally supports binary classification problems. In order to fit three classes, a suitable dataset had to be generated using the 'make_classification' function. This function was configured with rigorous hyperparameter tuning.[1]

The output layer dimensions of the neural network were updated from 2 to 3 to align with the new requirement of predicting three classes instead of two. This adjustment ensured that the network could correctly classify inputs into one of the three classes.

In adapting the model for three-class classification, logistic regression was intentionally omitted during the adaptation of the code due to conflicts encountered when attempting to integrate both methods. Moreover, logistic regression is entirely unprofitable for multiclass classification, as it is designed to handle binary classification tasks only. This decision allowed for a focused exploration of the neural network without the added complexity of incorporating multiple classifiers simultaneously.

Overall, the modifications and adjustments made to extend the neural network to three classes yielded satisfactory results. The model successfully classified the samples into the three classes (see Appendix).

[1]Hyperparameters that were adjusted: n_samples, n_features, n_informative, n_redundant, n_classes, n_clusters_per_class, class_sep, and random_state.

3.5 Four-Layer Network

In this task, the neural network was expanded to four hidden layers. This modification required detailed adjustments to both the forward propagation and backpropagation. The input and output layers remained unaltered to handle the two input features present in the dataset, and binary classification task.

In this extended configuration, as the data passed through the network, each hidden layer sequentially processed the information through its neurons. It is possible to change the number of neurons at each layer separately; for this experiment, I chose to use three neurons per layer. The neurons use weight matrices (W1 to W5) and bias vectors (b1 to b5) to compute tanh activations (a1 to a5) and finally derive output probabilities in the forward propagation step.

For backpropagation, the gradients (dW1 to dW5 and db1 to db5) were computed for each layer, allowing the adjustment of weights and biases. The regularization technique was kept as in the original code, but extended to new configuration: I retained the same regularization parameter α used in the original code, which was then applied to the three new weight matrices (W3, W4, W5). Additionally, the loss function was augmented to include W3 and W4, scaled by the same α.

Comparative analysis across different scatterplots configurations validated the model's performance consistency, this indicates robustness in learning both simple and intricate decision boundaries. Although the observed differences compared to the original architecture were minimal, the model demonstrated reliable performance.

4 Conclusion

If anything was learned from this experimentation, it is that simplicity can often be the most powerful asset we can leverage in both artificial and natural intelligence. In our challenge of understanding neural networks and their applications, we find parallels in human cognition where simpler solutions often yield optimal results. As humans, we naturally seek to maximize outcomes with minimal complexity, reflecting a fundamental principle observed across both machine learning and cognitive sciences.

Neural networks can indeed be compared to cognitive processes involved in recognizing and categorizing information in real-world scenarios. This analogy extends to language, visual perception, spatial reasoning, and beyond, where both neural networks and human cognition excel at extracting meaningful patterns from complex data. Moreover, connecting these findings to broader learning theories reveals interesting parallels. Just as different network architectures and layer complexities affect learning performance in machine learning, cognitive theories emphasize how varying cognitive strategies influence learning outcomes in humans. This can be mirrored by the principle of "simple is better" observed in some cognitive theories, where parsimonious processing mechanisms yield optimal learning outcomes. This analogy highlights the significance of investigating various neural network configurations (not necessarily the most complex) and tweaking their setups to obtain flexible and effective learning, similar to how our brain's lower-order circuits manage fundamental cognitive tasks with little computing expenses.

5 Appendix

I have written extensive code for this project. It is publicly available on GitHub (`https://github.com/fi-co/machine-learning-for-NLP`).

5.1 Additional Resources

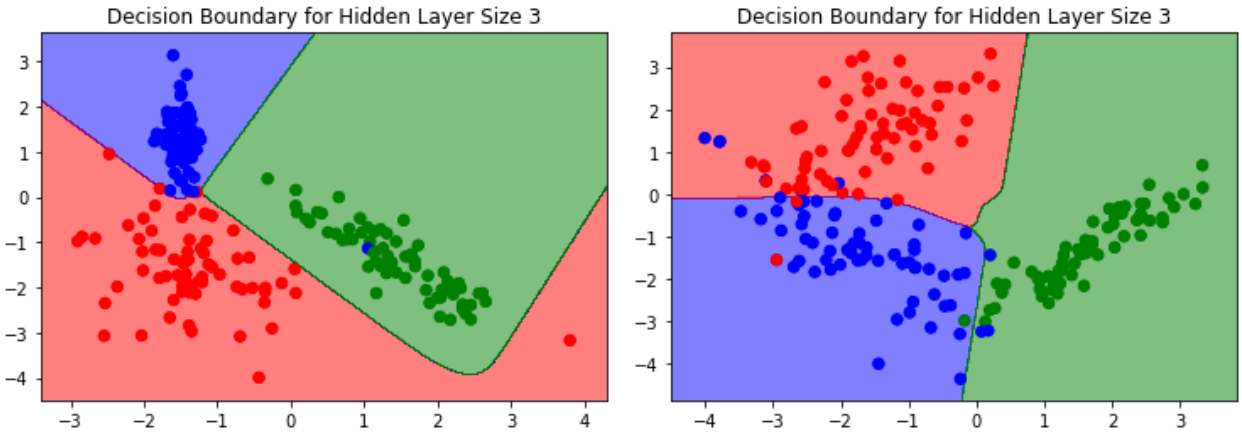

Figure 2: the three-classes neural network produced solid results, as evidenced by two generated plots. These plots clearly illustrate that the decision boundary effectively segregates the samples.

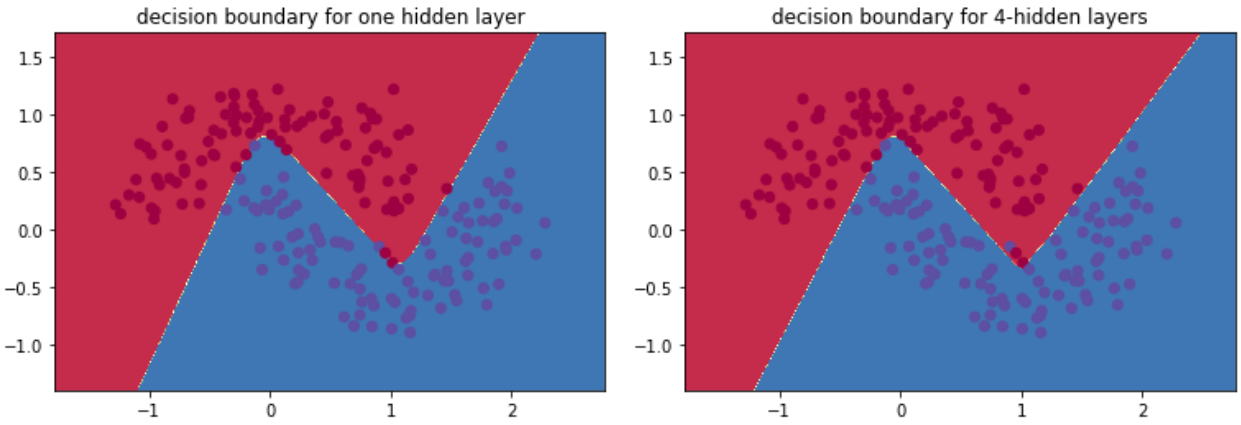

Figure 3: The decision boundary plots generated from the one hidden layer model (left) and the four hidden layers model (right) show that there is not much difference in their ability to classify the data accurately. In terms of computing resources and results trade-off, the four hidden layers model, yet accurate, is overly complex for this task. The simpler one hidden layer model provides a similar level of accuracy with significantly less engineering burden, highlighting the efficiency of simpler architectures for certain problems