Artificial Neural Networks

Marko Medved

I. Implementation of fully connected neural network for classification

A. Implementation details

We implemented a fully connected neural network with an arbitrary number of layers for a classification task. We applied the sigmoid activation function at each layer, except for the input and output layers. Since the task involves multi-class classification, we used the softmax function on the output layer (applied to the logits).

To initialize the weights we used the Xavier(Glorot) normal initialization, which works good when using the Sigmoid activation function:

$$W \sim \mathcal{N}\left(0,\, c \cdot \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

To train the network, we used an approach where gradients are computed separately for each training example and then accumulated. We used the cross-entropy loss (log loss) as the criterion. After computing the loss, we applied backpropagation to calculate the gradients and then performed standard gradient descent to update the model parameters.

To correctly perform backpropagation, the following gradients were calculated. First, the gradient of the loss function with respect to the logits was computed. This was done by combining the gradient of the loss with respect to the probabilities and the gradient of the probabilities with respect to the logits (i.e., the gradient of the softmax):

$$\frac{\partial L}{\partial z_j} = \begin{cases} p_j - 1 & \text{if } j = y_i \\ p_j & \text{otherwise} \end{cases}$$

where y_i is the true class, z_j are the logits and p_j are the probabilities of each class.

Next, the gradients for the weights and biases in the linear layer were calculated:

$$\frac{\partial L}{\partial \mathbf{W}^{(k)}} = \delta^{(k)} \cdot \left(\mathbf{a}^{(k-1)}\right)^T \quad \text{and} \quad \frac{\partial L}{\partial \mathbf{b}^{(k)}} = \delta^{(k)}$$

where $\delta^{(k)}$ is the gradient of the pre-activations in the current layer and a^{k-1} is the vector of activations in the previous layer.

Lastly, we needed to compute the gradient of the activations and consequently the pre-activations (note that in this case, k refers to the next layer, so it corresponds to $k-1$ in the previous equation):

$$\sigma'(z^{(k)}) = \sigma(z^{(k)}) \cdot \left(1 - \sigma(z^{(k)})\right)$$
$$\delta^{(k)} = \left(\delta^{(k+1)} \cdot \mathbf{W}^{(k+1)}\right) \circ \sigma'(z^{(k)})$$

where $\sigma(z^{(k)})$ represents the sigmoid function applied to the pre-activations.

B. Comparison of gradients to numerical gradients

To validate our implementation of backpropagation, we added the option to return both the numerical gradients and the gradients computed through backpropagation at a chosen layer. Our comparison procedure involved computing the gradients for all samples in the dataset (using the provided squares, doughnut, and example datasets), and then comparing the results. To compute the numerical gradients we used the defition of the derivative:

$$\frac{d}{dx} f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

and approximated it using $h = 0.0001$.

We used a network with six hidden layers, each with a different number of activations. We then checked the differences between the numerical and backpropagation-based gradients for both weights and biases. In all cases, the differences were within acceptable bounds—none of the gradient differences exceeded 1×10^{-7}.

C. Fitting the network to the squares and doughnut data

Next, we fitted our implementation to the squares and doughnut datasets to demonstrate that the network is capable of fully learning non-linear patterns.

In Figure 1, we observe that the network successfully fits the doughnut dataset. For this task, we used a network with a single hidden layer containing 3 activation units and a learning rate of 1.

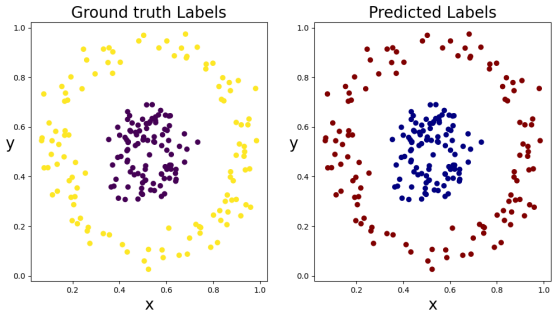

Figure 1. Network predictions fitted on the doughnut dataset.

Similarly, in Figure 2, the network also manages to perfectly fit the squares dataset. This time, we used one hidden layer with 5 activation units, keeping the learning rate at 1.

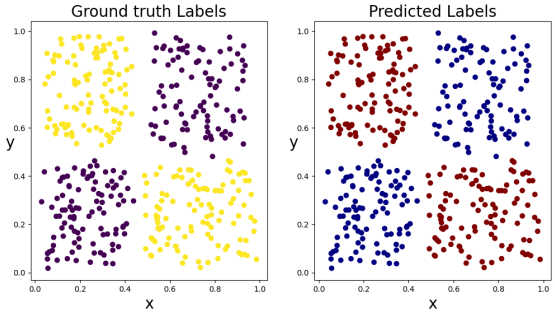

Figure 2. Network predictions fitted on the squares dataset.

II. Extending the ANN implementation

A. Adding support for regression

We also implemented the regression counterpart of the same neural network. The key differences in implementation compared to the classification network are outlined below:

- The final layer contains a single output unit instead of one per class.
- No softmax function is applied in the output layer.
- The loss function used is the mean squared error, based on the assumption of a normally distributed target variable.
- The only change in the backpropagation algorithm occurs in the final layer, where the gradient is scaled by a constant factor of 2 (in comparison to the classification network).

To test our implementation we used the provided tests.

B. Adding regularization

We also added the support for L2 regularization for both classification and regression networks. Note that the regularization was only applied to the weights, not to the biases. It was implemented by adding the correct term in gradient descent as shown by the following formula:

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} - \eta \left(\nabla_{\mathbf{W}^{(k)}} L + 2\lambda \mathbf{W}^{(k)} \right)$$

Where η is the learning rate and λ is the regularization constant.
We tested the implementation by comparing it with the PyTorch implementation in section D.

C. Adding reLU support

Next, we added support for the ReLU activation function. This required modifying the gradient computations, as the ReLU function and its derivative are defined as follows:

$$\mathrm{ReLU}(x) = \max(0, x)$$

$$\mathrm{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Additionally, we allowed users to customize the activation function used at each layer of the network. To validate our implementation, we calculated this differences in our and numerical gradients with the same procedure as for the Sigmoid. We also tested the model using ReLU activation functions on the *doughnut* and managed to completly fit the data again.

D. Comparing our implementation with PyTorch

We compared the training loss of our custom neural network implementation with an equivalent network implemented in PyTorch. Both networks used three hidden layers with sizes 5, 10, and 5, and employed sigmoid activation functions. We trained the models on the *Iris* dataset from the `scikit-learn` module for 300 epochs. To evaluate the effect of regularization, we tested four different values of the regularization coefficient. The results are shown in Figure 3. As observed, the loss curves follow similar trends between the two implementations. However, when regularization is applied, the overall scale of the losses differs, even though the general shape of the curves remains comparable.

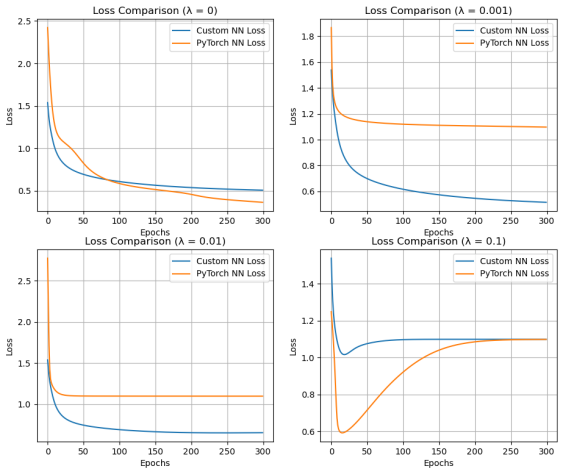

Figure 3. Comparsion of training loss of our implementation with an implementation in PyTorch.

III. Competition part

For our competition, we designed a fully connected neural network with two hidden layers, each containing 45 activations, and ReLU activation functions. We observed that adding more layers worsened the performance, so we kept the architecture simple with just two hidden layers. To add regularization, we incorporated Dropout layers into the network.

The input features consisted of spectral data at each pixel, with the positions of the pixels appended to the end of each feature vector. This combination allowed us to leverage both spectral and positional information for training.

To find the optimal configuration for the number of activations in the hidden layers, learning rate, and the number of epochs, we followed a structured approach. First, we split the dataset into 32 subsets, as illustrated in Figure 4. We then filtered these splits based on the number of annotated pixels, applying a threshold to retain only those with sufficient annotations. This process resulted in 13 valid splits.

Next, we performed cross-validation on these 13 splits and selected the model parameters that resulted in the lowest average log loss across all splits. This gave us the optimal configuration for our fully connected network.

Additionally, we experimented with 1D convolutional neural networks (CNNs) applied over the spectrum, excluding the positional data. However, we found that the performance of these CNN models did not come close to matching that of the basic fully connected networks.

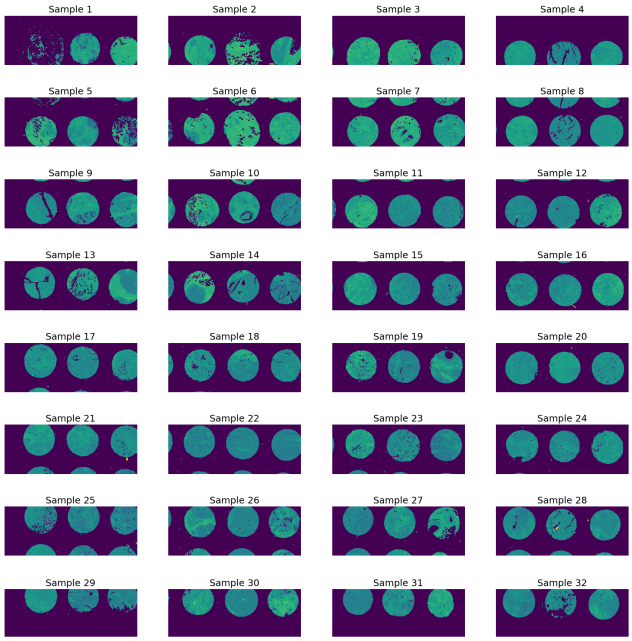

Figure 4. Cross validation splits