Artificial Neural Networks

Urša Zrimšek

Introduction

This is the report of sixth homework for class Machine Learning for Data Science 1, in which we implemented artificial neural networks (ANN).

We implemented ANN for classification and for regression, and first tested it on two datasets, that were already used in our previous assignments, `housing2r.csv` and `housing3.csv`. `housing2r.csv` dataset, has 200 rows with 5 independent and one numerical dependent column and `housing3.csv` has 500 rows with 13 independent and one categorical dependent column with 2 classes.

We separated both datasets on test and train set to find the best parameters of ANN on train set with grid search cross validation and then test them. We compared those results with results of models from previous homeworks: Logistic Regression and Suport Vector Regression.

We also used ANN for classification on bigger dataset, with 93 independent columns and one dependent with 9 classes. The dataset had 50000 rows, so it is important that we implemented an efficient network. We also did grid search cross validation on this dataset, to again find best parameters, and analyzed time needed for fitting different networks.

ANN implementation

Let's first define the architecture of the networks. We know that the size of input layer needs to be the same as the dimension of our input data. After that we can have an arbitrary number of hidden layers of any size, and the last, output layer, depends on our problem. In regression problem, we only have one layer, that outputs the target number, and in classification problem, we have as many layers as there are classes, and the output of each output neuron is the probability that the input belongs to that class.

To implement an ANN, we must choose which cost function we want to minimize and which activation function we will use. We decided to implement a network that has a possibility to use `ReLU` or `sigmoid` for hidden layer activations, and `sigmoid` or `softmax` for last layer activations of classification network. Regression network doesn't have activation on the output neuron (or we can say it has an identical one). Cost functions differ depending on the problem and last layer activation functions:

- regression problem: $\frac{1}{2}(y-t)^2$,
- classification with `softmax`: $-\log(y_t)$,
- classification with `sigmoid`: $-\log(y_t) - \sum_{i\neq t}\log(y_i)$,

where y is the output of our network and t is the true value (y_t is the output of t-th neuron, where t is the right class). This costs are defined on one datapoint, but for more points (or cost on the whole dataset), we will just average them. To prevent overfitting, we added regularization, that punishes too big weights: $\frac{1}{2}\lambda\sum\|W^l\|^2$, where W^l are the weight matrices of each layer.

We will find the optimal network by minimizing the cost. For that we will use `fmin_l_bfgs_b` from `scipy` optimization library. It has an option to numerically approximate the gradient, but since our cost function depends on many parameters (all weights and biases of the network), we need to calculate the gradients ourselves. We will do that by backpropagation, an algorithm that calculates gradients following a chain differentiation rule. For easier computation we define $\nabla_{z^l}C$ as the matrix of partial derivatives of cost over weighted sum, where columns belong to datapoints and rows to neurons in observed layer. To get derivative of biases from $\nabla_{z^l}C$ we need to average it over columns, and to get derivative of weights, we need to multiply it by activations of the previous layer and devise it by the number of datapoints, n (since we are actually calculating the average derivative by taking multiple datapoints at once), $\nabla_{W^l}C = \frac{1}{n}\nabla_{z^l}CA^T$. This calculations of gradients for biases and weights from $\nabla_{z^l}C$ is the same for all layers, so we will only describe how to get $\nabla_{z^l}C$ for the last layer and for previous ones.

Backpropagation begins with calculating the partial derivatives on the last layer of the network:

$$\nabla_{z^L}C = y^L - t,$$

where t is the correct output of the network (1 on the true class and 0 elsewhere). The derivation is the same for all three cost functions, which is also one of the reasons for the choice above. This way, our different networks can all share the same code for backpropagation. Gradients on all the previous layers are calculated using gradients on the layer after them:

$$\nabla_{z^l}C = \sigma'(z^l) \circ (W^{l+1})^T \nabla_{z^{l+1}}C,$$

where σ' is the derivative of activation function, and $\circ$ represents element-wise multiplication.

With added regularization, also the gradients by weights change. The added cost is not dependent on datapoints, and is equal to λW^l for each weight matrix. So we can just add that to the gradients. Regularization doesn't effect bias gradients.

Numerical verification of gradients

To verify if our calculation of gradients is correct, we used the definition of derivative. For each weight and bias we calculated $\frac{C(x+hw)-C(x)}{h}$ for some small h - we used $h = 10^{-6}$. Here x represent some random value of weights and biases, and w represents a vector of zeros everywhere, except for the weight or bias that we are looking at. If the difference between numerically calculated gradient and gradient that we got from backpropagation was smaller than $tol = 10^{-4}$ we considered them equal.

Housing dataset

In this section we will represent results on the housing dataset. We first separated it into train and test set (80-20), then did a GridSearchCV on the train set to find the best values for the network size and for regularization parameter. We tried $\lambda \in \{0, 0.01, 0.1, 0.5, 1\}$ and `units` $\in \{[], [10], [10, 10], [20, 20], [10, 10, 10]\}$. We tried different combinations of activation functions on hidden and last layer, and compared the performance that was achieved with the best parameters from grid search to the performance of support vector regression and logistic regression. The best parameters for methods from previous homeworks were already selected then (on the same datasets). The results can be observed in tables I and II.

Table I
Comparison of performance on `housing2r`.

	MSE
SVR	17.8
ANN with `sigmoid`	43.8
ANN with `ReLU`	10.8

The network with best performance using both activation functions was the same. It had three hidden layers with 10 neurons each, and was using regularization parameter 0.5.

Table II
Comparison of performance on `housing3`.

	MSE
logistic regression	0.23
ANN with `sigmoid`	0.6
ANN with `sigmoid` and `softmax`	0.31
ANN with `ReLU` and `sigmoid`	0.35
ANN with `ReLU` and `softmax`	0.13

Also here, the best network parameters were the same for all combinations of activation functions on hidden and last layers: two layers with 20 neurons each, and regularization parameter 0.01. When running the search for parameters, we can observe that if we don't include regularization, our network overflows, because the weights become too big. From the tables above we can see that the best networks use `ReLU` on hidden layers and `softmax` (or identity for regression) on the last layer. They outperform other two methods, but if we use other activation functions, the results are worse.

Big dataset

We also did a GridSearchCV on the big dataset, to find the best architecture and regularization parameter. As this dataset is much bigger, we only used network with `ReLU` activation on hidden layers and `softmax` on the last, as we saw that this one performs best. From grid search in which we compared $\lambda \in \{0.01, 0.05, 0.1, 0.5, 1\}$ and $units \in \{[], [10], [50], [10, 10], [20, 50], [10, 10, 10]\}$ we found out that the best architecture has two hidden layers of 20 neurons, and $\lambda = 0.01$. Here we can see that we left out the network without regularization, because as we saw before, it tends to overflow, and it takes a lot of time to fit.

As mentioned, we also compared fitting times, and found out that they are not only dependent on the size of the network, but also on the regularization parameter. We could explain this with the size of the space we are searching for the minimum in. If the regularization is bigger, that consequently means that the size of weights is more limited and that we will then quicker reach the limit of our possibilities. When comparing the times for different architectures, we see that the most important is the size of first hidden layer (if we don't set others too big), because the number of weights here multiplies with 93, the number of input neurons. The network with best parameters, mentioned above, needed 212 seconds for fitting and 0.006 seconds for predictions on the whole test set.

We did another 5–fold cross validation of the network with the best parameters on the whole dataset, to evaluate it and measure its uncertainty. It reached mean cross entropy of 0.65 on all datapoints, with standard deviation of 0.84.

Conclusion

We can see that artificial neural networks with correctly chosen activation functions outperform other methods we implemented, even when they have only few hidden layers. They need a lot of time to fit, but we can see that after that, the prediction are very fast.