Homework 6 report

Luka Utješinović, 63210495

ANN Implementation

First we need to introduce slight formalisms. Suppose our ANN has L hidden layers. Let n_l denote the number of neurons in the l-th layer. n_1 is usually equal to the number of input features, and n_L depends on the problem. For regression, $n_L = 1$ and for classification $n_L = K$ where K is the number of different classes. For classification problems, our implementation performs the softmax transformation of the last layer within the loss function rather than creating an additional output layer.

Let w_{ij}^l denote the weight that corresponds to the link between i-th neuron in l-th layer and j-th neuron in $l-1$-th layer. Let $W^l \in \mathbb{R}^{n_l \times n_{l-1}}$ represent the weight matrix between layers l and $l-1$. With b_i^l we denote the intercept of i-th neuron in l-th layer, and in similar fashion with b^l we denote the intercept vector for layer l. With f_l we denote the activation function shared by all neurons in layer l.

Explicitly writing out the function that is encompassed by an ANN is practically impossible. However, using the fact that outputs of layer l are inputs to layer $l+1$ we can easily compute the output for a given batch of N observations X following the **forward pass** algorithm:

Algorithm 1 Forward pass(X)

$a^1 \leftarrow X^T$
for $l = 2, \dots L$ **do**
  $z^l \leftarrow W^l a^{l-1} + \underbrace{[b^l \dots b^l]}_{\text{r times}}$
  $a^l \leftarrow f_l(z^l)$
  ▷ $z^l, a^l \in \mathbb{R}^{n_l \times N}$
end for
Return $(a^2)^T, \dots (a^L)^T, (z^2)^T, \dots (z^L)^T$
▷ $(z^l)^T, (a^l)^T \in \mathbb{R}^{N \times n_l}$, observations by rows and inputs/activations in columns

For convenience we introduce following abbreviations: $(a^l)^T = a'^l, (z^l)^T = z'^l$. Standard setup in supervised machine learning is **empirical risk minimization**. Assuming our network a is parametrized with weight matrices $W = \{W^2 \dots W^L\}$ and intercept vectors $b = \{b_2 \dots b_L\}$, for a training dataset $X \in \mathbb{R}^{N \times p}$ we are interested in solving the following optimization problem:

$$\begin{aligned} W^*, b^* &= \arg\min_{W,b} C(W,b) = \arg\min_{W,b} \frac{1}{N} \sum_{i=1}^{N} L(y_i, a_{W,b}(x_i)) \\ &= \arg\min_{W,b} \frac{1}{N} \sum_{i=1}^{N} L_i \end{aligned} \tag{1}$$

where L is the loss function of choice. Our implementation supports **squared error** for regression problems, and **log-loss** for classification problems, but this can easily be extended as the loss function and it's derivative is passed as a parameter as we will see shortly.

First order optimization algorithms require computing $\frac{\partial L_i}{\partial w_{kj}^l}, \frac{\partial L_i}{\partial b_k^l}$. Again, since neural networks represent extremely complex functions, we can view them as computational graphs instead. From this perspective we can derive an algorithm for computing aforementioned partial derivatives. Let $\delta_k^l = \frac{\partial L_i}{\partial z_k^l}$ represent the **error term** of k-th neuron in layer l, $\delta^l = [\delta_1^l, \dots \delta_{n_l}^l]^T$. Then the following equalities can be proven:

$$\delta^L = \nabla_{a^L} L_i \odot f_L'(z^L) \tag{2}$$

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot f_l'(z^l), l = 2, \dots L-1 \tag{3}$$

$$\frac{\partial L_i}{\partial w_{kj}^l} = a_j^{l-1} \delta_k^l \tag{4}$$

$$\frac{\partial L_i}{\partial b_k^l} = \delta_k^l \tag{5}$$

where $\odot$ represents the element-wise product operator.

These results hold for a single observation only, however considering the fact that the total loss is the average of individual losses, that is $\frac{\partial C}{\partial w_{kj}^l} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L_i}{\partial w_{kj}^l}$ and $\frac{\partial C}{\partial b_k^l} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L_i}{\partial b_k^l}$ we can easily compute $\frac{\partial C}{\partial w_{kj}^l}, \frac{\partial C}{\partial b_k^l}$. $df_l(z'l) \in \mathbb{R}^{N \times n_l}$ is the matrix obtained by applying f_l' to elements of $z'l$. Similarly we define:

$$\nabla C_{X,y} = \begin{bmatrix} \frac{\partial L_1}{\partial a_1^L} & \cdots & \frac{\partial L_1}{\partial a_{n_L}^L} \\ \vdots & \ddots & \vdots \\ \frac{\partial L_N}{\partial a_1^L} & \cdots & \frac{\partial L_N}{\partial a_{n_L}^L} \end{bmatrix} \tag{6}$$

We are interested in computing the following matrices:

$$\partial W^l = [\frac{\partial C}{\partial w_{ij}^l}]_{i=1,j=1}^{n_l, n_{l-1}} \tag{7}$$

$$\partial b^l = [\frac{\partial C}{\partial b_i^l}]_{i=1}^{n_l} \tag{8}$$

By writing C as the sum of individual losses we can obtain "vectorized" expressions for $\partial W^l, \partial b^l$:

$$\delta^L = (\nabla C_{X,y} \odot df_L(z_L'))^T \tag{9}$$

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot df_l(z_l')^T l = 2, \dots L-1 \tag{10}$$

$$\partial W^l = \delta^l a'^{l-1} l = 2, \dots L-1 \tag{11}$$

$$\partial b^l = \delta^l 1_{N \times 1} \text{ (sum of rows of matrix } \delta^l) \tag{12}$$

Even though we do not return it from the forward bass algorithm, we consider that $a'^1 = X$. Finally, we formulate the **backpropagation** algorithm which computes aforementioned

partial derivatives over a training batch X, y of size N:

Algorithm 2 Backpropagation(X, y)

$a'^2, \dots a'^L, z'^2, \dots z'^L \leftarrow$ Forward pass(X)
$\delta^L \leftarrow (\nabla C_{X,y} \odot df_L(z'^L))^T$
for $l = L-1, \dots, 2$ **do**
    $\partial W^{l+1} \leftarrow \delta^{l+1} a'^l$
    $\partial b^{l+1} \leftarrow \delta^{l+1} 1_{N\times 1}$
    $\delta^l \leftarrow ((W^{l+1})^T \delta^{l+1}) \odot df_l(z'_l)^T$
end for
$\partial W^2 \leftarrow \delta^2 X$
$\partial b^2 \leftarrow \delta^2 1_{N\times 1}$
Return $\partial W^2 \dots \partial W^L, \partial b^2 \dots \partial b^L$

This derivation does not include the regluarization term $\frac{1}{2}\sum_{l=2}^{L} ||W^l||^2$. Changes to the backpropagation algorithm with regularization are trivial.

We've implemented a generic **Network** class, and the constructor takes the following parameters:

- **units** - list of integers which describe number of neurons per hidden layer.
- **loss** - loss function for individual observations.
- **dloss** - derivative of the loss function.
- **net_type** - NetworkType enum instance. It has two possible values: R (which stands for regression) and C (which stands for classification).
- **lambda_** - L2 weight decay strength. If not specified, 0.01 is taken.
- **activations** - list of activation functions per hidden layer. If not specified, ReLU is used for every layer.
- **dactivations** - derivatives of corresponding activation functions.
- **output_activation** - activation function for the output layer. If not specified, identity is applied
- **doutput_activation** - derivative of output activation.
- **seed** - For reproducibility.

Within this class we've implemented aforementioned algorithms. As required by homework instructions, we should implement two classes. **ANNRegression** is a wrapper for **Network** with the following parameters:

- **loss** is set to square loss and **dloss** is set to **dsquare_loss**
- **net_type** is set to NetworkType.R

Similarly, **ANNClassification** is a wrapper for **Network** with the following parameters:

- **loss** is set to log loss and **dloss** is set to **dlog_loss**
- **net_type** is set to NetworkType.C

We used **scipy's L-BFGS** implementation for empirical risk minimization. Additionally, we used **Xavier weight initialization**, that is:

$$W^l, b^l \sim U\left(\frac{-1}{\sqrt{n_l}}, \frac{1}{\sqrt{n_l}}\right) \tag{13}$$

Gradient verification

In order to verify the correctness of implemented algorithms, we used numerical estimates for partial derivatives (**verify_gradient** function in the accompanying Python file). For a continuously-differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ partial derivative with respect to i-th variable in point $a = [a_1 \dots a_n]^T$ is defined as:

$$\frac{\partial f}{\partial x_i} = \lim_{h\to 0} \frac{f(a_1, \dots a_i + h, \dots a_n) - f(a)}{h}$$

We took a small step size $h = 10^{-6}$ and computed the previous quotient, which represented our numerical estimate of the true derivative. For a given network and a given dataset (generated by sampling from a normal distribution) we compared this with partial derivatives returned by backpropagation, and if the absolute difference was smaller or equal to $\varepsilon = 10^{-4}$ we considered them equal. The opposite case never occurred, and hence we consider our implementation correct.

Housing datasets

We used housing datasets and compared our implementation of ANN with well performing models implemented and trained during previous homeworks. In order to evaluate our models we used 10-fold cross validation (stratified version for classification datasets). Additionally, we standardized predictor features.

For every dataset, in order to find the best network we performed a grid search over the following hyperparameter values: $\lambda \in \{0.01, 0.1, 0.5\}$, units $\in \{[], [10], [50], [10, 20], [50, 20], [10, 20, 30], [50, 20, 30]\}$. We also compared two different activations for every layer: **ReLU** and **sigmoid**.

As a baseline for regression networks, we used our implementation of **SVR with a RBF kernel** and the following hyperparameters: $\varepsilon = 10, \lambda = 0.01, \sigma = 4.1$. As a baseline for classification networks we used our implementation of **multinomial logistic regression** (no regularization applied). After cross-validation, for different choices of activation functions we selected best performing models. The following results were obtained:

Model	Activation	λ	units	MSE
ANN	ReLU	0.5	[10, 20, 30]	26.6
ANN	Sigmoid	0.01	[10]	**25.97**
SVR	/	/	/	36.94

TABLE I
Housing regression dataset. We can see that even simple neural networks we can achieve significantly better results than with SVR.

Model	Activation	λ	units	Mean log-loss
ANN	ReLU	0.01	[50, 20]	**0.25**
ANN	Sigmoid	0.01	[]	0.28
MLR	/	/	/	0.29

TABLE II
Housing classification dataset. Unlike with regression ReLU activation gave us best results.

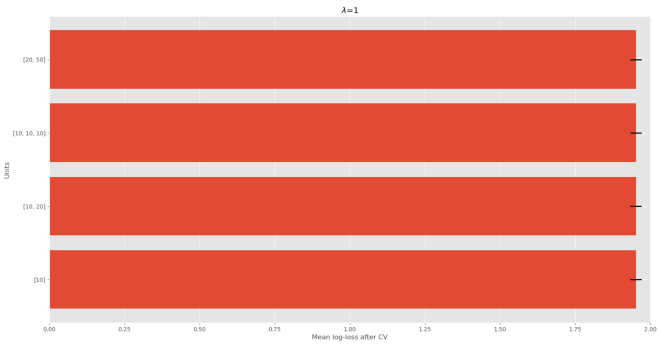
For housing classification dataset, it is interesting to observe that for sigmoid activation the best architecture did not have hidden layers (weights are only present in the output layer where identity is performed), which is equivalent to MLR

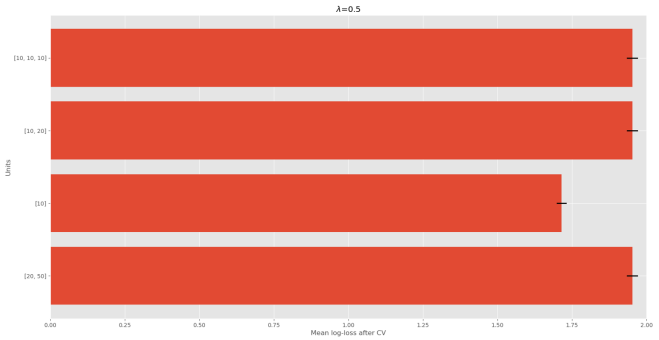
with L2 regularization weight $\lambda = 0.01$ (softmax is performed within the loss function and when making predictions). Our implementation of MLR did not support regularization, and it appears that regularization gives slightly better results, which is expected because it is a countermeasure to overfitting.

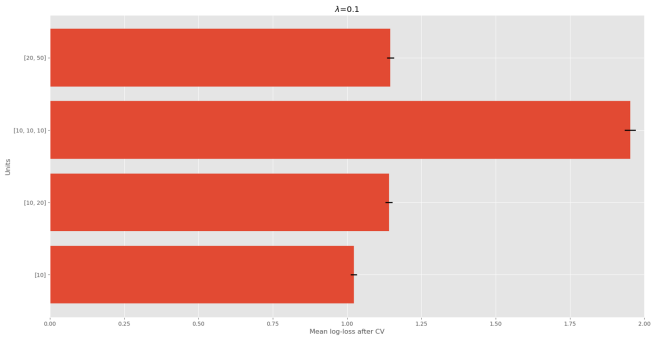
Big dataset exploration

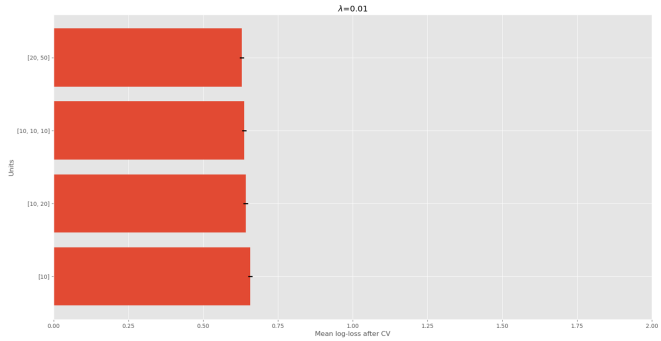
Finally we applied our ANN Implementation to the big dataset. This is a classification dataset with 5×10^4 observations, each having 95 features. One of those features (**target**) is the response, while another (**id**) just assigns unique id to each observation (which seems to be the row index). Including this feature within the learning process would probably cause problems, so we removed it. Response values {Class1, ... Class9} were mapped to $\{0, \dots 8\}$. Remaining features were standardized.

We used 5-fold stratified cross validation to evaluate our models, and we performed a grid search over the following parameters: $\lambda \in \{0.01, 0.1, 0.5, 1\}$, units $= \{[10], [10,20], [20,50], [10,10,10]\}$. ReLU performed better than sigmoid for housing classification dataset, and following this heuristic we only used ReLU as our activation. As a baseline model we used **sklearn's implementation of multinomial logistic regression**, with estimated mean log-loss of 0.64. Obtained results are shown on following figures:









Horizontal line for each plot represents a 95% confidence interval for the mean log-loss. They are pretty narrow, which is somewhat expected because our dataset is huge. Best mean log-loss ≈ 0.63 was achieved with $\lambda = 0.01$ and units $= [20,50]$, which is just slightly better than the baseline. Total time required for computation of results was ≈ 2.15 hours. Also, it seems that bigger regularization seems to lead to poorer models for this dataset.

The final model which was used to make predictions for test.csv was trained on the entire dataset using obtained hyperparameters. Total training time was ≈ 7 minutes.

To conclude, we can see that even with simple architectures neural network models outperform models we've seen previously, but we should be extremely careful as they are prone to overfitting, and they should always incorporate some kind of regularization.