REGRESSION AND CLASSIFICATION USING FEED FORWARD NEURAL NETWORKS

Ivar Stangeby

December 11, 2019

Abstract

1 Introduction

In this project we make a comparative study on feed forward neural networks in a regression and classification setting. In particular, we compare feed forward neural networks in this context with the standard methods of ordinary least squares for linear regression, and logistic regression for classification.

2 Background

2.1 Neural Networks

Construction

Neural networks has gained a widespread use in the approximation of functional relationships. In it's simplest form, a neural network consists of a set of *neurons*, being entities that take as input a set number of parameters $x_1, \dots, x_n$, and computes a linear combination of these (often with added bias):

$$z = \sum_{i=0}^{n} w_i x_i + b. \tag{1}$$

The neuron then decides on whether to "fire" or not, by evaluating a so-called scalar activation function, $a = g(z)$. In the architecture discussed herein, namely *fully connected* or *dense* neural networks, each neuron takes into account all possible parameters.

By combining a set number of neurons we obtain what is called a *layer*. A neural network typically consists of three types of layers: an *input layer*, a set number of *hidden layers*, and finally an *output layer*.

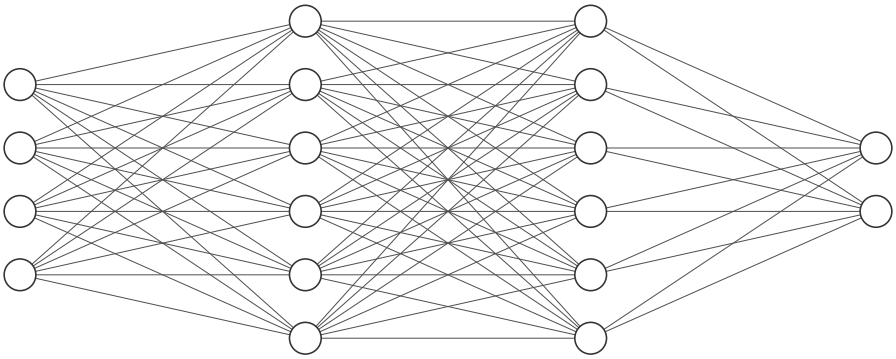

Figure 1: A simple feed forward neural network. This specific network has an input layer of four nodes. Two hidden layers consisting of six nodes each, and finally, an output layer of size two.

For a network consisting of L layers, with each layer having $n^{[\ell]}$ nodes and with activation function g, the activations in each layer are given as

$$a_k^{[\ell]} = g\left(\sum_{j=1}^{n^{[\ell]}} w_{jk}^{[\ell]} a_j^{[\ell-1]} + b_k^{[\ell]}\right) \tag{2}$$

for $k = 1, \ldots, n^{[\ell]}$ and $\ell = 1, \ldots, L$. A matrix formulation reads:

$$\mathbf{a}^{[\ell]} = g\left(\boldsymbol{W}^{[\ell]} \mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}\right). \tag{3}$$

We use the convention that the network input $\mathbf{x} = (x_1, \ldots, x_{n^{[0]}}) = \mathbf{a}^{[0]}$ and the network output $\hat{\mathbf{y}} = \mathbf{a}^{[L]}$.

Optimization

We initialize such a network by specifying the initial weights and biases for each layer. That is, the matrices $\boldsymbol{W}^{[\ell]}$ and $\mathbf{b}^{[\ell]}$ are (most commonly) drawn from some random distribution. It is our goal to iteratively optimize these weights and biases in order for the network to produce desired output. We therefore choose a *cost* function $\mathcal{C}$ which in some way incorporates our problem at hand. For example, in regression problems, our cost function is typically the mean squared error (MSE) between our target-data and our prediction, while in classification, we often use cross entropy loss.

We optimize the network by computing the gradient of the cost function $\mathcal{C}$ with respect to weights and bias' and perform gradient descent. That is

$$\begin{aligned} \boldsymbol{W}^{[\ell]} &\leftarrow \boldsymbol{W}^{[\ell]} - \lambda \frac{\partial \mathcal{C}}{\partial \boldsymbol{W}^{[\ell]}}, \\ \mathbf{b}^{[\ell]} &\leftarrow \mathbf{b}^{[\ell]} - \lambda \frac{\partial \mathcal{C}}{\partial \mathbf{b}^{[\ell]}} \end{aligned} \tag{4}$$

Here $\lambda > 0$ is the *learning rate*.

Neural Network Architecture

In this paper we study two use-cases of feed forward neural networks, namely regression, and classification. The architectures are tailored to the data-sets at hand.

Credit Card Data The credit-card data-set consists of 33 numerical features describing a users payment history. The task at hand is then to predict whether the user is prone to defaulting or not. This is therefore a binary classification problem. Our network therefore needs to have an input layer of size 33, and an output layer of size two (one probability value for each class). We—rather arbitrarily—decide on three hidden layers of size 20. We use the *rectified linear unit* as our activation function for the hidden-layers, and (since we are predicting probabilities) we use the *softmax* function as our output-activation. These are defined as follows:

$$\mathrm{RELU}(\mathbf{x}) = \max\{\mathbf{0}, \mathbf{x}\}, \tag{5}$$

$$\mathrm{Softmax}_k(\mathbf{x}) = \frac{\exp x_k}{\sum_{i=1}^{n} \exp x_i}. \tag{6}$$

Note that the softmax takes as input a vector, and spits out a scalar, and that the RELU takes the element-wise maximum. Our cost function $\mathcal{C}$ is chosen to be *binary cross entropy* (or *log-loss*) defined as

$$\mathcal{C}(\mathbf{y}, \hat{\mathbf{y}}) = -(\mathbf{y} \log(\hat{\mathbf{y}}) + (1 - \mathbf{y}) \log(1 - \hat{\mathbf{y}})). \tag{7}$$

Regression Data For our regression task, we create a synthetic data-set by noisy sampling of points from the Franke test-function, and our goal is to create a neural network that approximates the Franke function. Thus, our network should take as input a set of two values, and return a scalar. Again, we use a rectified linear unit as our activation function for the hidden layers, however, since we want an unrestrained output range, we do not add the softmax activation to the final layer, as this would squeeze the network output between zero and one. Our cost function is the mean squared error (MSE):

$$\mathcal{C}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}(\mathbf{y} - \hat{\mathbf{y}})^T(\mathbf{y} - \hat{\mathbf{y}}). \tag{8}$$

3 Implementation

While implementing backpropagation is a great exercise in programming, due to time-constraints, we have had to resort to using machine learning frameworks in the implementation.

For defining our neural networks we use the Pytorch-library, and for standard methods like linear regression and logistic regression, we resort to the scikit-learn toolkit. In addition to this, we use the Skorch-library for wrapping the Pytorch modules in a scikit-learn compatible API.

Data Wrangling

We subclass the PyTorch dataset-class for loading the credit card data and the sampled Franke data as shown in listing 1. This enables us to write lean training loops as the `DataLoader` class works as a batched iterator.

```
class CreditCardData(torch.nn.utils.data.Dataset):
    def __init__(self, *args, **kwargs):
        super().__init__()

        self.X, self.y = self.fetch_data(*args, **kwargs)
        self.num_features, self.num_items = self.X.shape

    def __getitem__(self, index):
        return self.X[index], self.y[index]

    def __len__(self):
        return self.num_items

        super().__init__()
    def fetch_data(self):
        # Handles the loading and preprocessing of data
        ...

class FrankeData(torch.nn.utils.data.Dataset):
    ...
```

Listing 1: Interfacing the data-sets with a PyTorch API, which facilitates easy iteration as shown in listing 2

```
credit_card_data = CreditCardData('path_to_dataset')
credit_card_loader = DataLoader(credit_card_data, batch_size=64)

# training loop
for (X_batch, y_batch) in credit_card_loader:
    ...
```

Listing 2: A simple training loop using PyTorch data-loaders.

Neural Network

We implement a generic neural network that will suffice for both the credit card data (the classification task) and the Franke sampled data (the regression task). By subclassing `torch.nn.Module`, we only need to register the different layers, and implement the `forward`-method. The autograd framework then takes care of the back-propagation of gradients for us. We allow for the user to specify the number of input features, the number of output features, the number of hidden layers and the the number of hidden units (which we take as constant across the hidden layers). Furthermore, we also with to be able to specify the final activation function, as this needs to be specially tailored (as much else) to the task at hand. The generic architecture is layed out in listing 3.

```
class GenericNN(torch.nn.Module):

    def __init__(
        self, num_in_features, num_out_features,
        num_hidden_layers=3,
        num_hidden_features=20,
        activation=torch.nn.Relu,
        final_activation=torch.nn.Identity
    ):
        super().__init__()
        self.input_layer = torch.nn.Linear(num_in_features,
            num_hidden_features)
        self.hidden_layers = torch.nn.ModuleList(
        [
            nn.Linear(num_hidden_features, num_hidden_features)
            for i in range(num_hidden_layers)
        ])
        self.output_layer = torch.nn.Linear(num_hidden_features,
            num_out_features)
        self.g = activation
        self.f = final_activation

    def forward(self, x):
        x = self.g(self.input_layer(x))
        for layer in self.hidden_layers:
            x = self.g(layer(x))
        x = self.f(self.output_layer(x))

        return x
```

Listing 3: A generic dense neural network realized in the PyTorch framework. By implementing the forward method, gradients are kept track of automatically, and hence no consideration is to be made for the backward propagation.

Classical Methods

For the classical methods of (regularized) ordinary least squares and logistic regression, we simply use the scikit-learn toolkit. These classifiers can be trained by calling `fit`-method. By utilizing cross validation methods, we optimize the hyper-parameters. This will be discussed further in the results-section. As an example, for the credit-card dataset, we train the classifiers shown in listing 4, performing a grid-search for the best regularization strength:

3.1 Numerical Results

In this section, we present the findings from our numerical experiments.

```
l1_params = [{'penalty':['l1'],'C':np.logspace(-4, 4, 40),\
      'solver':['liblinear']}]
l2_params = [{'penalty':['l2'],'C':np.logspace(-4, 4, 40),\
      'solver':['liblinear']}]

clf_l1 = GridSearchCV(LogisticRegression(), l1_params, cv=5,\
   verbose=True, n_jobs=-1)
clf_l2 = GridSearchCV(LogisticRegression(), l2_params, cv=5,\
   verbose=True, n_jobs=-1)

clf_l1.fit(X, y)
clf_l2.fit(X, y)
```

Listing 4: Setting up and training the logistic regressor. We use both L^1 and L^2-regularization, corresponding to Lasso and Ridge-regression, respectively. We find the optimal regularization-strength by 5-fold cross validation over the data-set. The fits are parallelized over all available CPU-cores, to speed up the process.

Credit card data

We train our sklearn-classifiers as in listing 4. We do a hyper-parameter search for the best regularization-strength for both L^1 and L^2 regularization. We note that *regularization strength* in the world of scikit-learn is the inverse of what might be the convention. Thus, a regularizatin strength of λ corresponds to the penalization coefficient $1/\lambda$. With this in mind, we display the results in fig. 2. Note how, for small values of λ, the regularization effectively sets all weights to zero. The fact that the network still predicts correctly about 78.6% of the time is due to the fact that the network now only predicts zeros, and the data-set consists of about 78.6% zeros. In the other limit, as λ increases, the regularization term is effectively set to zero, and both methods converge to non-penalized logistic regression. Incidentally, it is also this non-penalized method that performs the best with a classification accuracy of 81.6%.

The neural network classifier is trained using the `skorch`-framework, which acts as an interface to scikit-learn for PyTorch. Using this allows for simple hyper-parameter optimization as in scikit-learn, and one may test for instance network depth, width, and regularization strength. Unfortunately, due to time constraints, we were only able to perform a learning-rate optimization. However, the implementation in listing 5 allows for other sets of hyper-parameters to be tested.

The results are shown in fig. 3, and as we see, the neural network, even with its basic architecture, outperforms the classical methods by a fair bit. The optimal learning rate yields a classification accuracy of 82.54%.

A Regularized neural network

Franke function

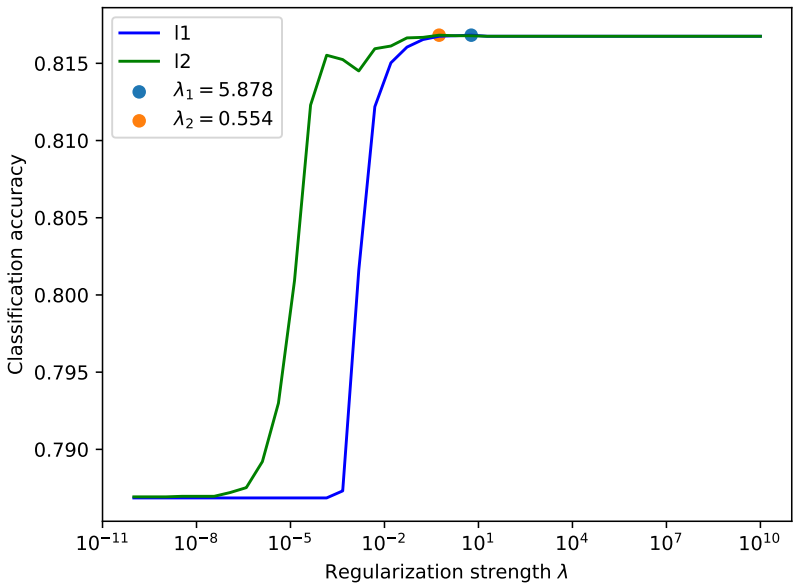

Figure 2: The results of the L^1 and L^2 regularized logistic regression. The two regularization types seems to converge to exactly the same classification accuracy as non-penalized logistic regression, with this value being 81.6%. As λ increases, the penalization term is set to zero, hence converges to non-penalized regression. Why this coincides with the best fit is not clear to me.

```
net = RegularizedNeuralNetClassifier(
    GenericNN,
    module__input_size=credit_card_data.num_features,
    module__num_hidden_layers=3,
    module__output_size=2,
    criterion=torch.nn.CrossEntropyLoss,
    train_split=CVSplit(cv=5, random_state=42),
    optimizer=torch.optim.SGD,
    max_epochs=5,
    optimizer__momentum=0.9
)

params = {
    'lr': np.logspace(-4, 0, 30),
    # these hyper-params can be trained for as well
    # 'module__num_hidden_layers' : range(0, 5),
    # 'module__hidden_layer_size' : [5, 10, 15, 20]
    # 'module__regularization = ['l2', 'none', 'l1']
    # 'module__alpha = np.logspace(-10, 10, 40) # regularization str
}
clf = GridSearchCV(
    net,
    params,
    n_jobs=12,
    cv=5
)
clf.fit(credit_card_data.X.numpy(), credit_card_data.y.numpy())
```

Listing 5: By wrapping the GenericNN in a RegularizedNeuralNetClassifier (see listing 6) we may train the PyTorch model using the GridSearchCV-functionality from scikit-learn.

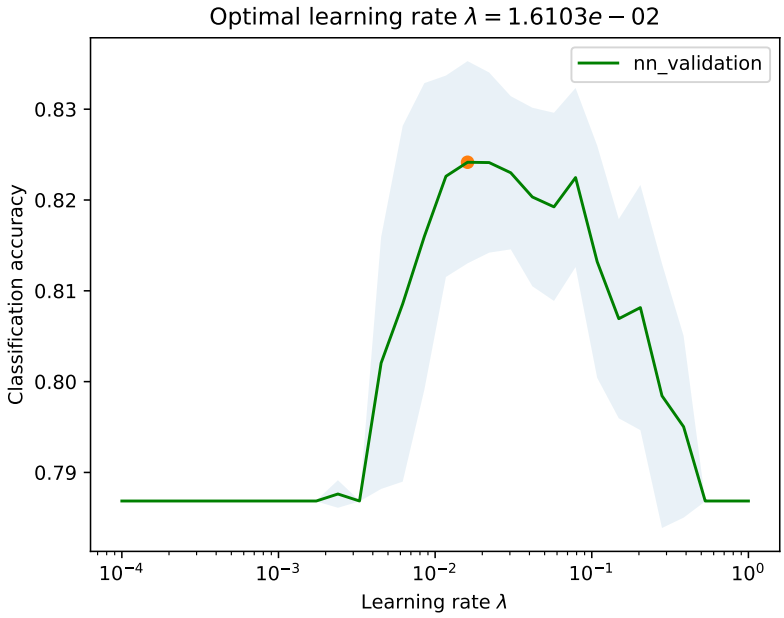

Figure 3: The classification accuracy as a function of the learning rate for the generic neural network. At the opimal learning rate, the network achieves a prediction accuracy of approximately 82.54%, which is not insignificantly better than all the classical methods. The filled blue regions represent the standard deviations as reported by `GridSearchCV`. Note how the network at performs it's worst by guessing only zeros, similarily to the overpenalized logistic regression. The network is trained over 20 epochs, using SGD with momentum and 5-fold cross validation.

```
class RegularizedNeuralNetClassifier(NeuralNetClassifier):

    def __init__(self, module, alpha=1, regularizer='none',
          criterion=torch.nn.NLLLoss,
                train_split=CVSplit(5, stratified=True),
                classes=None,
                *args,
                **kwargs):
        self.alpha = alpha
        self.regularizer = regularizer

        if 'regularizer' in kwargs.keys():
            kwargs.pop('regularizer')
        if 'alpha' in kwargs.keys():
            kwargs.pop('alpha')

        super().__init__(module, *args, criterion=criterion, \
      train_split=train_split, classes=classes, **kwargs)

    def get_loss(self, y_pred, y_true, *args, **kwargs):
        loss = super().get_loss(y_pred, y_true, *args, **kwargs)

        if self.regularizer == 'none':
            pass
        elif self.regularizer == 'l1':
            loss += self.alpha * sum([w.abs().sum()
                        for w in self.module_.parameters()])
        elif self.regularizer == 'l2':
            loss += self.alpha * sum([(w.abs() ** 2).sum()
                        for w in self.module_.parameters()])
        return loss
```

Listing 6: We implemented a regularized neural network classifier in order to be able to compare the network with both L^2 and L^1 regularization applied to the network weights. Unfortunately, we were not able to perform the required numerical tests in time, as the parameter space is rather large.