Project 2 FYS-STK4155

Reinert Eldøy

November 18, 2022

Abstract

An implementation of gradient descent and stochastic gradient descent in the context of linear regression and logistic regression is investigated. Then a feedforward neural network with backpropagation is constructed and compared with previously implemented methods.

Introduction

There is no lack of useful machine learning methods out there, from Least Squares Regression to Gradient Boosted Tress. Perhaps the brightest star of the ML constellation however is the neural network and its many variants. In the current project I will first replace the matrix inversion algorithm used in the regression subtasks of project 1 with stochastic gradient descent. This will lay some groundwork for the subsequent implementation of a standard feedforward neural network with a flexible number of hidden layers and neurons. Through the use of test functions I will evaluate the network's performance under different conditions such as varying activation functions, SGD parameters and the width and depth of the hidden layers. Finally an optimized model will be applied to a real life dataset. In the section following this introduction I will expound on methods used and the theoretical paradigms at play. The subsequent section will contain all results in the form of figures and tables, and a discussion and analysis of these soon follows. At the very end I leave a few concluding remarks, such as what worked well and what didn't, and the most important findings.

Methods & Theory

The programming language of choice in this project is Python. Here follows a brief description of Linear and Logistic Regression, Stochastic Gradient Descent and Feedforward Neural Networks.

Linear Regression

Broadly speaking, *regression* methods seek to find approximate mathematical relationships between measured quantities in the presence of noise. *Linear re-*

gression is the version where the underlying relationship is assumed linear, i.e. we might assume the data is generated by the expression

$$y = \beta X + \varepsilon$$

where X is the independent variable(e.g. resistance) and y the dependent variable(e.g. voltage). β is the linear coefficient(e.g. current) and ε is a noisy random variable(e.g. thermal fluctuations), often assumed to obey a centered normal distribution. The central task is to approximate β, and this is done by minimizing a *cost function*. The simplest case is known as Ordinary Least Squares where the cost function is the mean squared error,

$$\frac{1}{n}\sum_{i=1}^{n}(y - X\beta)^2$$

and this will hopefully result in a set of linear coefficients $\hat{\beta}$ such that $\hat{y} = \hat{\beta}X$ is a good approximation to y. The standard way of addressing this problem is to solve the *normal equations*, which result in

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

This tends to be prone to invertibility issues and thus highly varying coefficients. This can be addressed by using *regulators* such as *Ridge* and *LASSO* regression, which minimize the same cost function as above except for an additional tunable L^2 norm or L^1 norm term that restrict the allowed range of the coefficients.

Logistic Regression

This is a regression method primarily used for classification tasks. It could be as simple as a spam filter accepting or rejecting an email, a situation we might want to represent as $y \in \{0, 1\}$. This isn't continuous and in practice it turns out to be better to assign probabilities to either choice. Such a probability function could be the *sigmoid function*,

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

which looks like in Fig 1. Its smoothness is of immediate note. For purposes of machine learning we can instead consider the conditional probability

$$P(y_j = 1|x, \theta) = \frac{1}{1 + e^{-x_j^T \theta}} = \sigma(x_j^T \theta)$$

which is to be interpreted as the probability of observing that $y = 1$ *contingent on* the model parameters θ. So what's a good cost function for logistic regression? The answer is the *cross entropy*

$$C(\theta) = \sum_{i=1}^{n} -y_i \log(\sigma(x_i^T \theta)) - (1 - y_i) \log(1 - \sigma(x_i^T \theta))$$

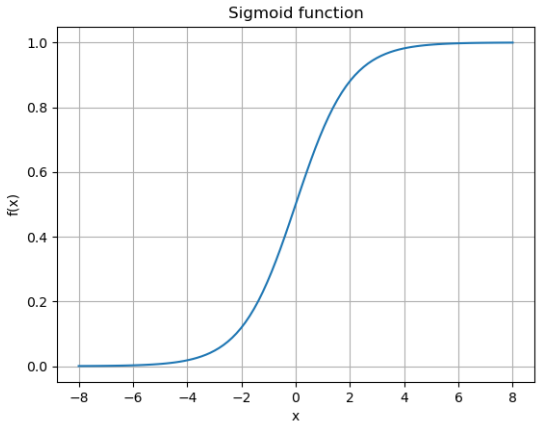

Figure 1: The sigmoid function appears like a smoothed out step function

Stochastic Gradient Descent

Rather than inverting matrices, we can minimize cost functions by incrementally approaching a minimum. The main inspiration is the Newton method which in the one-dimensional case takes the form

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

and if we were to apply it to machine learning with a vast number of regression coefficients we would have to invert a truly massive Hessian matrix, which would be prohibitively expensive computationally:

$$x_{n+1} = x_n - (Hf(x_n))^{-1}\nabla f(x_n)$$

The *gradient descent* method simply replaces it with a learning rate η. There are further problems, though, for instance that the cost function landscape may have multiple local minima. This can be addressed by introducing stochasticity to the method in the form of the *stochastic gradient descent*. This assumes that the cost function is expressible as a sum over datapoints such that the gradient can be applied to each term individually, which is less computationally taxing. In addition there is the notion of *minibatches*, namely a partition of the dataset(typically after shuffling) into k equally sized chunks and restricting the gradient to only one of these chunks at a time. All in all we get something which looks like

$$\theta_{n+1} = \theta_n - \eta \sum_{i \in B_j} \nabla C_i(\theta, x_i)$$

where θ are regression coefficients, η the learning rate, B_j a minibatch and C_i a cost function localized to a single datapoint x_i. A full cycle of all the minibatches is called an *epoch*, and with this newfound stochasticity there is a chance of avoiding or escaping local minima.

Neural Networks

The Perceptron

Artificial neural networks are inspired by the electrochemical operation of biological neural networks, or brains if you will. The fundamental unit of this system is called a *perceptron* and consists(see figure below) of an *input layer* connected to a central node by weighted edges. The inputs and weights are

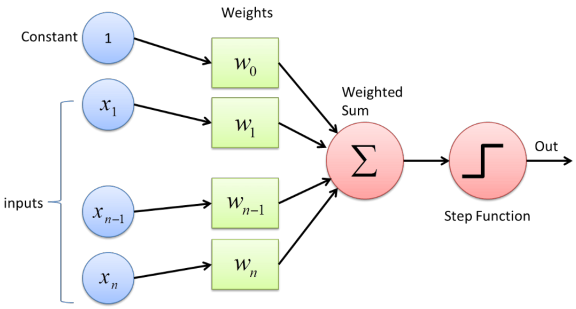

Figure 2: Taken from https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

then combined in the algebraic expression

$$z = \sum_{i=1}^{n} w_i x^i + b_i$$

where I've included an important bias term. Next up we apply a nonlinear *activation function* to the expression. In the case of the perceptron this is the Heaviside step function, but another popular choice is the *sigmoid* function,

$$\sigma(z) = \frac{1}{1 + e^{-x}}$$

or the $ReLU$ function

$$ReLU(x) = \max(x, 0)$$

Thus the output of the perceptron looks like $\sigma(\sum_{i=1}^{n} w_i x^i + b_i)$. This model is meant to mimic a single biological neuron such that the input edges might represent signal receiving dendrites and the output edge might represent the signal transmitting axon. But a biological network may consist of billions of neurons working in concert, and in the same way multiple perceptrons can tie together in large neural network architectures. This project will concern itself with one such architecture.

The Feedforward Neural Network

A feedforward neural network(henceforth "FFNN") consists as before of an input layer and an output layer, but now the output layer could contain multiple

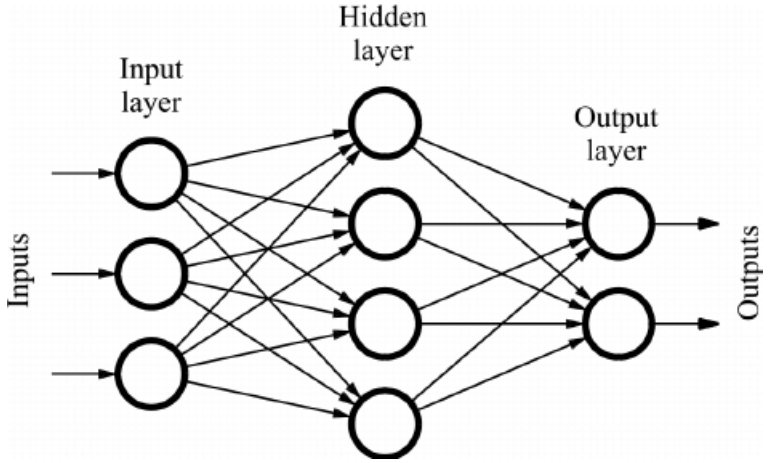

Figure 3: From https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network

nodes rather than just the one "nucleus" in the perceptron. In addition we could have a layer in between the two known as a *hidden layer*, in fact we could have multiple of these with as many hidden neurons as we please. To top it all, as we scan from left to right we see that each node in a layer to the left is connected to *every* node to the right. The final characteristic of the FFNN is that no edge loops backwards, hence "feedforward". As before each edge is weighted with a number denoted w^l_{jk} where l labels the layer it's pointing into and it points from neuron j to neuron k. Not shown in Fig. 2 is the possibility(or even necessity) of a bias term b^l_j at any of the layers. In addition to the structural freedom, we are free to pick any activation function we please at any of the neurons, and so this model seems to have a great degree of inherent flexibility.

Training the network

There are various and sundry potential applications for neural networks, from linear regression to interpreting handwriting. Given a dataset to be fed through the input layer, how can we adjust the weights so that the output at the other end holds true to this promise? The key is gradient descent. Recall that the cost function is defined on a landscape of tunable coefficients, or weights. With sufficiently large FFNNs it would be futile to compute the full gradient, so SGD is the natural choice for optimizing the model. Computing the gradient directly would still be difficult in this case, but there is a clever workaround known as *backpropagation*. This approach takes into consideration the layered structure of the model by computing the output error and using it to recursively compute the errors at each hidden layer. The algorithm goes as follows:

1. Compute the activation of each input neuron, a^1_j

2. Perform the weightings and activations for each hidden layer in turn by using the rule

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

3. Compute the output error as $\Delta_j^l = \frac{\partial E}{\partial a_{j^l}} \sigma'(z_j^l)$ item Propagate backwards by recursively computing the hidden errors:

$$\Delta_j^l = \left(\sum_k \Delta_k l + 1 w_{kj}^{l+1}\right) \sigma'(s_j^l)$$

4. Finally determine the cost function gradient with respect to the weights(and biases) as

$$\frac{\partial E}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1}$$

$$\frac{\partial E}{\partial b_j^l} = \Delta_j^l$$

Results

Discussion

Conclusion