University of Oslo

FYS-STK4155

Helle Bakke

Something catchy

November 7, 2019

Contents

Abstract

Abstract

1 Introduction

Machine learning is a fairly new concept to most people, but its early history dates back to 1943 when Warren McCulloch and Walter Pitts introduced the first neural network (Mayo et al., 2018). In the following decade, the Turing test was made known, and in 1958, Frank Rosenblatt designed the first artificial neural network called the *Perceptron*. However, it was not until the early 1980s the interest in neural networks gained new momentum. In 1986, the *back propagation* algorithm was developed, an important step in any neural network. Since then, the world has seen a large increase in both data and computing power. These factors have contributed to the rapid progress in machine learning that we see today.

The aim of this project is to study classification and regression problems by developing our own neural network and comparing it to a logistic regression code and the linear regression code developed in Project 1. We will use a credit card data set from UCI for the classification problem. This data set is well studied, which is useful when validating our models. For the linear regression problem we will revisit the Franke function from Project 1.

The structure of the report is as follows. **WRITE THIS LAST!**

2 Method

2.1 Logistic regression

Logistic regression is commonly used when dealing with a categorical dependent variable, or *target*. In classification problems, the targets take the form of discrete variables such as categories, or *features*. In many cases the target is binary, meaning that it can be either yes or no, true or false, positive or negative etc. The credit card data set we are exploring has binary targets (0,1). Rewrite??

Logistic regression is an important first step in order to understand the neural network algorithms and how supervised deep learning works. In logistic regression, we want to find β such that the cost function $C(\beta)$ is min-

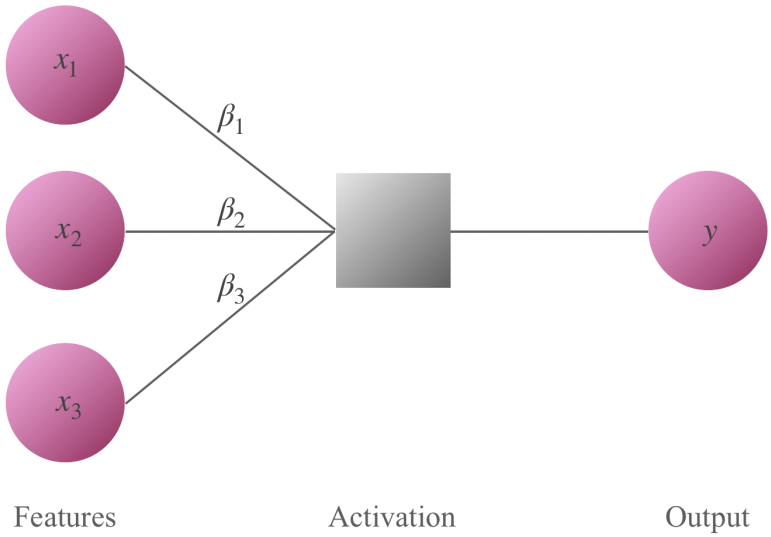

Figure 1: Simple illustration of a logistic regression model.

imised. This leads to gradient descent methods, which are important parts of nearly all machine learning algorithms. Mention that GD is an optimizer?

Figure 1 shows a simple logistic regression model with three features. The β-values of each feature are passed to an *activation function*, before the output is given. In the following, we go into the details of setting up a logistic regression model.

2.1.1 Cost function

In order to find the β-values that minimise $C(\beta)$, we need to define the cost function. The target of our classification problem is binary, hence we start by defining the probability, or likelihood, for a given event. The probability of a data point belonging to a category $y = \{0, 1\}$ is given by the Sigmoid function

$$p(\theta) = \frac{1}{1 + e^{-\theta}} = \frac{e^{\theta}}{1 + e^{\theta}} \tag{1}$$

where θ is the prediction of y (in logistic regression, this reduces to the linear equation $\tilde{\tilde{y}} = \hat{X}\beta$). Further, the probabilities of y_i being either 0 or 1 can be

expressed as

$$p(y_i = 1|\hat{X}^i, \hat{\beta}) = \frac{e^{\beta_0 + x_1^i\beta_1 + x_2^i\beta_2 + ... + x_p^i\beta_p}}{1 + e^{\beta_0 + x_1^i\beta_1 + x_2^i\beta_2 + ... + x_p^i\beta_p}}$$
$$p(y_i = 0|\hat{X}^i, \hat{\beta}) = 1 - p(y_i = 1|\hat{X}^i, \hat{\beta})$$

where p is the number of predictors. Next, we use the above probabilities to define the log-likelihood

$$C(\hat{\beta}) = -\sum_{i=1}^{n}(y_i \log p(y_i = 1|\hat{X}^i, \hat{\beta}) + (1 - y_i)\log[1 - p(y_i = 1|\hat{X}^i, \hat{\beta})]) \quad (2)$$

which is the cost function. In statistics, this equation is known as the *binary cross-entropy*, where y is the target (either 0 or 1) and $\sigma(y)$ is the predicted probability of the data point being 1. In example, we see that when $y_i = 1$, the cost reduces to

$$C(\hat{\beta}) = -\sum_{i=1}^{n} \log p(y_i = 1|\hat{X}^i, \hat{\beta}),$$

and when $y_i = 0$ the cost reduces to

$$C(\hat{\beta}) = -\sum_{i=1}^{n} \log[1 - p(y_i = 1|\hat{X}^i, \hat{\beta})].$$

2.1.2 Gradient descent

Gradient descent is an optimisation algorithm used to find the local minima of a function. The idea is to minimise $C(\hat{\beta})$, where $\hat{\beta}$ is a vector with elements for each feature value (Marsland, 2014). We want to find a sequence of new points $\hat{\beta}(i)$ that move towards a solution. This is done by taking the derivative of the cost function. However, the direction of the derivative is not arbitrary. In steepest gradient descent, we always choose to go downhill as fast as possible for each point. This leaves us with the equation

$$\hat{\beta}_{k+1} = \hat{\beta}_k - \eta \nabla_\beta C(\hat{\beta}) \quad (3)$$

where η is the leraning rate and the derivative of the cost function is

$$\nabla_\beta C(\hat{\beta}) = -\hat{X}^T(\hat{y} - \hat{p}) \quad (4)$$

Here, $\hat{X}$ is the feature matrix, $\hat{y}$ is a vector of the targets and $\hat{p}$ is a vector of the fitted probabilites (Sigmoid function). A drawback of the steepest gradient descent method is that many of the directions that $\hat{\beta}$ travel are directly towards the centre (local minima).

In general, gradient descent methods have limitations, and we can address some of the shortcomings by considering the stochastic gradient descent (SGD) method. We keep equation (3) as it is, but add stochasticity by randomly shuffle the data. Stochastic gradient descent converges faster for larger data sets, but since the method uses one example at a time, the computations can slow down (Patrikar, 2019). By adding mini-batches to the model, we can vectorise the SGD, making the computations faster. Additionally, the gradient is computed against more training samples, which in turn means that it is averaged over more training samples. This may then lead to a smoother convergence.

For each *epoch*[1], we loop over all the mini-batches and calculate $\hat{\beta}_{k+1}$ using equation (3). After we have calculated $\hat{\beta}_{k+1}$ for the training samples in all the mini-batches, we randomly shuffle the data before continuing to the next epoch. The process is repreated until we reach the end of the last epoch.

To summarise and connect the above theory to Figure 1, the logistic regression model for classification problems is as follows:

- Divide the data into features and targets.
- Split the data into training and test samples.
- Feed a randomly generated $\hat{\beta}$-value to the activation layer, where we for each epoch calculate $\hat{\beta}_{k+1}$ for the training samples in all the mini-batches. Randomly shuffle the training data.
- Calculate the prediction of $\hat{y}$.

2.2 Neural network

A neural network is a collection of neurons that can learn to recognise patterns in data. In biological terms, a neuron is a nerve cell. It is the processing units of the brain, and each neuron is a separate processor performing simple tasks (Marsland, 2014). An artificial neuron sums the incoming signals, and

[1] A single iteration over all the data.

an activation function/threshold decides whether or not an output is given. If the threshold is not overcome, the neuron has zero output (Hjorth-Jensen, 2019).

In this project, we are creating a feed-forward neural network. That means that the information moves forward through the layers. Figure 2 shows a Multi-layer Perceptron (MLP) with an input layer consisting of n neurons, one hidden layer consisting of 4 neurons and an output layer consisting of 2 neurons. An MLP is a fully-connected feed-forward neural network with three or more layers, and consists of neurons with non-linear activation functions (such as the Sigmoid function). Training the MLP consists of finding the outputs given the inputs and current weights and biases, and then updating the weigths and biases according to the output error. In the following, we will take a deeper dive into the mathematical model comprising the Multi-layer Perceptron.

2.2.1 Feed-forward pass

The feed-forward pass refers to the process of passing values from the input neurons through the network, layer by layer, until the output layer is reached. The activation values of each neuron in each layer is defined as

$$z_j^l = \sum_{k=1}^{N_{l-1}} w_{jk}^l a_k^{l-1} + b_j^l \tag{5}$$

where $\hat{w}^l$ are the weights, $\hat{a}^{l-1}$ are the forward passes/outputs from the previous layer, $\hat{b}^l$ are the biases and N_{l-1} represents the total number of neurons in layer $l-1$. With the activation values from the current layer, we can define the output $\hat{a}^l$ of the current layer as

$$\hat{a}^l = f(\hat{z}^l) \tag{6}$$

where the function f is the *activation function*. In the logistic regression model, we used the Sigmoid function in equation (1) as our activation function. This is a non-linear activation function, and we can use it in order to obtain the output of each neuron in each layer such as

$$a_j^l = f(z_j^l) = \frac{e^{z_j^l}}{1 + e^{z_j^l}} \tag{7}$$

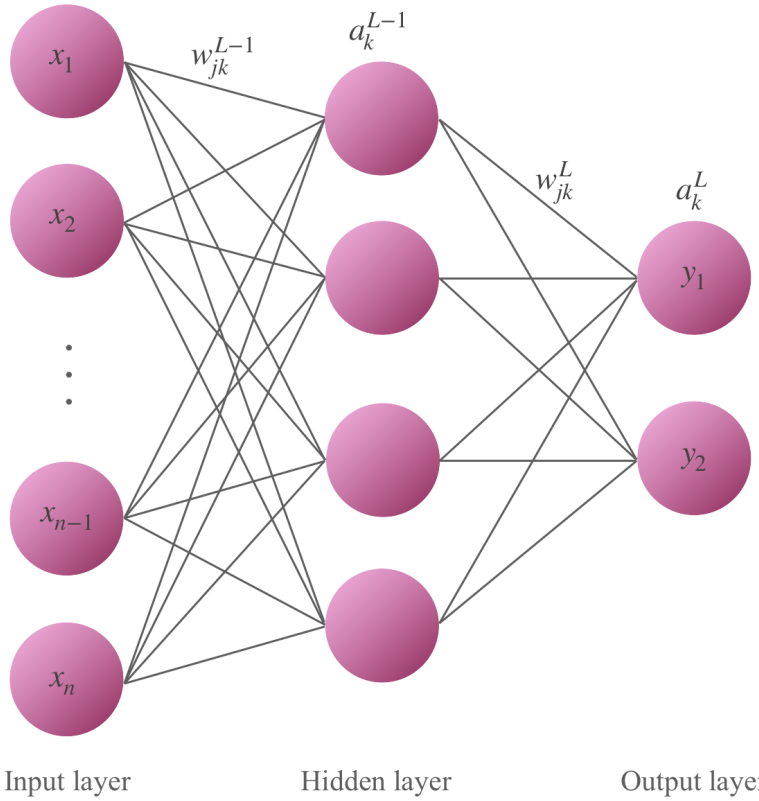

Figure 2: Simple illustration of a feed-forward neural network with one hidden layer.

In the case of linear regression, we change the activation function from a Sigmoid function to a Rectified Linear Unit (ReLU). The ReLU activation function is defined as

$$f(z_j^l) = \max(0, z_j^l), \tag{8}$$

and is linear for all positive values of z_j^l and zero for all negative values of z_j^l. More??

2.2.2 Back propagation

Back propagation refers to the algorithm used to train a feed-forward neural network. Back propagation computes the gradient of the cost function with respect to the weights of the network. Hence, it is useful to define the cost function before initiating the mathematical gymnastics of the back propagation algorithm.

For the classification problem, we use binary cross-entropy defined as

$$C(\hat{W}^L) = -\sum_{k=1}^{N_L}(y_k \log a_k^L + (1-y_k)\log[1-a_k^L]) \tag{9}$$

where y_k is the target and a_k^L is the output from neuron k in layer L (the output layer). The derivative of the cost function with respect to the weights can be written as

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \frac{\partial C(\hat{W}^L)}{\partial a_k^L}\frac{\partial a_k^L}{\partial w_{jk}^L} \tag{10}$$

where we have applied the chain rule. We can apply the chain rule to last derivative on the right-hand side as well, obtaining

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \frac{\partial C(\hat{W}^L)}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_k^L}\frac{\partial z_k^L}{\partial w_{jk}^L} \tag{11}$$

Now, we have three derivatives that we can solve. The first derivative on the

right-hand side of equation (11) is solved as

$$\begin{aligned}\frac{\partial C(\hat{W}^L)}{\partial a_k^L} &= -\frac{y_k}{a_k^L} + \left(\frac{1-y_k}{1-a_k^L}\right) \\ &= \frac{a_k^L(1-y_k)}{a_k^L(1-a_k^L)} - \frac{y_k(1-a_k^L)}{a_k^L(1-a_k^L)} \\ &= \frac{a_k^L(1-y_k) - y_k(1-a_k^L)}{a_k^L(1-a_k^L)} \\ &= \frac{a_k^L - y_k}{a_k^L(1-a_k^L)}\end{aligned}$$

The second derivative on the right-hand side of equation (11) can be solved as

$$\begin{aligned}\frac{\partial a_k^L}{\partial z_k^L} &= f'(z_k^L) \\ &= a_k^L(1-a_k^L)\end{aligned}$$

The last term on the right-hand side of equation (11) is just

$$\frac{\partial z_k^L}{\partial w_{jk}^L} = a_k^{L-1}$$

Next, we insert the separate solutions of the derivatives into equation (11) and obtain

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = a_k^{L-1}(a_k^L - y_k) \tag{12}$$

We can then define the output error

$$\delta_k^L = f'(z_k^L)\frac{\partial C(\hat{W}^L)}{\partial a_k^L} = a_k^L - y_k \tag{13}$$

This will be the starting equation of the back propagation algorithm. The error terms of the hidden layers now be defined as

$$\delta_j^l = f'(z_j^l)\frac{\partial C(\hat{W}^l)}{\partial a_j^l} = \frac{\partial a_j^L}{\partial z_j^l}\frac{\partial C(\hat{W}^l)}{\partial a_j^l} = \frac{\partial C(\hat{W}^l)}{\partial z_j^l}$$

The error can also be interpreted in in terms of biases b_k^L, such as

$$\delta_j^l = \frac{\partial C(\hat{W}^l)}{\partial b_j^l}\frac{\partial b_j^l)}{\partial z_j^l} = \frac{\partial C(\hat{W}^l)}{\partial b_j^l}$$

However, we want to express the error in terms of the equation for layer $l+1$. This can be done by employing the chain rule so that

$$\delta_j^l = \sum_k^{N_{l+1}} \frac{C(\hat{W}^l)}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k^{N_{l+1}} \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l}$$

From equation (5), we see that the derivative $\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_k^{l+1} j f'(z_j^l)$. The hidden layer error can finally be written as

$$\delta_j^l = \sum_k^{N_{l+1}} \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{14}$$

After computing the errors of the hidden layers, the weights and biases are finally updated using stochastic gradient descent as

$$w_{jk}^l \leftarrow w_{jk}^l = \eta \delta_j^l a_k^{l-1} \tag{15}$$

and

$$b_j^l \leftarrow b_j^l = \eta \delta_j^l \tag{16}$$

2.3 Model evaluation

Accuracy, mse, r2 score/cost etc, neural network testing, keras etc.

2.4 Data preprocessing

2.5 Code

Write about the setup of the code, include pseudo code if necessary. What does the design matrix look like? How is β found numerically?

3 Results

Make plot of loss versus epoch

4 Discussion

5 Conclusion

6 Appendix

Relevant programs developed to solve this project can be found at the GitHub address

https://github.com/hellmb/FYS-STK4155/tree/master/Project_2

References

Hjorth-Jensen, M.
2019. Data analysis and machine learning: Neural networks, from the simple perceptron to deep learning. https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet.html. Accessed: 2019-07-11.

Marsland, S.
2014. *Machine Learning: An Algorithmic Perspective*. Chapman and Hall/CRC.

Mayo, H., H. Punchihewa, J. Emile, and J. Morrison
2018. History of machine learning. https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html. Accessed: 2019-07-11.

Patrikar, S.
2019. Batch, mini batch & stochastic gradient descent. https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a. Accessed: 2019-07-11.