Classification and Regression: From Linear and Logistic Regression to Neural Network

Johan Mylius Kroken[1, 2] and Nanna Bryne[1, 2]

[1] Institute of Theoretical Astrophysics (ITA), University of Oslo, Norway
[2] Center for Computing in Science Education (CCSE), University of Oslo, Norway

November 10, 2022 GitHub repo link: https://github.com/Johanmkr/FYS-STK4155colab/tree/main/project2

ABSTRACT

ABSTRACT

****
Felt cute, might delete later:

for ideas (\feltcute)

rephrase this (\rephrase{...})
check if this is correct (\checkthis{...})
* *comment (\comment{...})*
add filler text (\fillertext)
for when you are lost (\wtf)
****

Nomenclature

0.1. Datasets ??

$\boldsymbol{\theta}$	Parameter vector
$\mathcal{L}$	Total loss/cost function $\mathcal{L}(\boldsymbol{\theta}; f, \mathcal{D})$ of $\boldsymbol{\theta}$ parametrised by the coordinates in the data set $\mathcal{D}$ and the function f (often written as $\mathcal{L}(\boldsymbol{\theta})$ for ease of notation)
$\mathcal{A}$	Magnitude and direction of steepest ascent in parameter space
X	Feature matrix of n row vectors $\mathbf{x}^{(i)} \in \mathbb{R}^p$, where p denotes the number of features we are considering
$\mathbf{y}$	Vector of n input targets $y^{(i)} \in \mathbb{R}$ associated with $\mathbf{x}^{(i)}$
$\mathcal{D}$	Dataset $\{X, \mathbf{y}\}$ of length $n \in \mathbb{N}$ on the form $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$
n	Number of samples in a datasets

0.2. Network components

$\mathbf{h}$	Hidden layer of a neural network ($\mathbf{h}^l \in \mathbb{R}^{N_l}$)
g	Activation function associated with a layer in a neural network, affine transformation ($g_l : \mathbb{R}^{N_l} \to \mathbb{R}^{N_l}$)
W	Matrix of weights describing the mapping from a layer to the next ($W^{l \to l+1} \in \mathbb{R}^{N_l \times N_{l+1}}$)
$\mathbf{b}$	Bias ($\mathbf{b}^l \in \mathbb{R}^{N_l}$)
	* *More!*
$\mathbf{a}$	Activation argument ($\mathbf{a}^l \in \mathbb{R}^{N_l}$)
N	Number of neurons in a layer ($N \in \mathbb{N}$)

0.3. Hyperparameter syntax

η	Learning rate
γ	Momentum factor
$\mathbf{v}$	Momentum in parameter space
L	Number of layers, not counting the input
λ	Penalty parameter in Ridge regression

0.4. Indexing and iteration variables

k	Iteration variable when optimising as **subscript**
(i)	The i^{th} example of a sample as **superscript**

0.5. Miscellaneous

$\lVert\mathbf{u}\rVert_q$	ℓ^q-norm of $\mathbf{u}$
$\nabla_\xi \varrho$	gradient of ϱ with respect to ξ

0.6. Forkortelser på engelsk

DAG	Directed acyclic graph
FFNN	Feedforward neural network
GD	Gradient descent
MSE	Mean squared error
NAG	Nesterov accelerated gradient
NN	Neural network
OLS	Ordinary least squares
ReLU	Rectified linear unit
SGD	Stochastic gradient descent

* *Should order alphabetically or logically.*

1. Introduction

2. Theory

In linear regression we have the famous assumption that

$$y_i = f(\mathbf{x}_i) + \varepsilon_i \simeq \mathbf{x}_i^\intercal \boldsymbol{\beta} + \varepsilon_i, \tag{1}$$

where f is a continous funtion of $\mathbf{x}$. If we now were to allow said function to represent discrete outputs, we would benefit from moving on to *logsitic* regression.
* *Maybe move to intro?*
add filler text smooth transition to steepest descent

2.1. Stochastic gradient descent (SGD)

Gradient descent (Hjorth-Jensen 2021)
add filler text aim is to find beta etc

The result is a flexible way to locate the minima of any cost function $\mathcal{L}(\boldsymbol{\theta})$. The ordinary least squares and Ridge schemes of linear regression that we discussed in project 1, are then implemented by using the cost functions $\mathcal{L}^{\text{OLS}}(\boldsymbol{\theta})$ and $\mathcal{L}^{\text{Ridge}}(\boldsymbol{\theta})$ respectively, given by the following expressions:

$$\mathcal{L}^{\text{OLS}}(\boldsymbol{\theta}) = \|\mathbf{f}(X;\boldsymbol{\theta}) - \mathbf{y}\|_2^2 \tag{2a}$$

$$\mathcal{L}^{\text{Ridge}}(\boldsymbol{\theta}) = \|\mathbf{f}(X;\boldsymbol{\theta}) - \mathbf{y}\|_2^2 - \lambda\|\boldsymbol{\theta}\|_2^2 \tag{2b}$$

* *Explain different parts*

2.1.1. Plain gradient descent (GD)

The most basic concept is that of steepest descent. In order to find a minimum of a function $\varrho(\boldsymbol{\xi})$ (allow multivariability of $\boldsymbol{\xi}$), we follow the steepest descent of that function, i.e. the direction of the negative gradient $-\nabla_\xi \varrho(\boldsymbol{\xi})$. We thus have an iterative scheme to find minima:

$$\boldsymbol{\xi}_{k+1} = \boldsymbol{\xi}_k - \eta_k \nabla_\xi \varrho(\boldsymbol{\xi}_k), \tag{3}$$

where the learning rate η_k goes as

$$\eta_k = \begin{cases} (1-\frac{k}{\tau})\eta_0 + \frac{k}{\tau}\eta_\tau, & 0 \le k \le \tau \\ \eta_\tau, & k > \tau \end{cases} \tag{4}$$

for an initial rate η_0 and lower limit $\eta_\tau \sim 0.01\eta_0$.
add filler text

What we would like to minimise is the cost function $\mathcal{L}(\boldsymbol{\theta})$ which is a function of the parameters $\boldsymbol{\theta}$ which we are trying to estimate. If we define $\mathcal{A}_k \equiv \nabla_\theta \mathcal{L}(\boldsymbol{\theta}_k)$ to be the direction and magnitude of the steepest ascent in parameter space, eq. (3) reads

$$\begin{aligned} \boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k + \mathbf{v}_k\,; \\ \mathbf{v}_k &= -\eta_k \mathcal{A}_k, \end{aligned} \tag{5}$$

for substitutions $\varrho \to \mathcal{L}$ and $\boldsymbol{\xi} \to \boldsymbol{\theta}$. For a sufficiently small η_k, this method will converge to a minimum of $\boldsymbol{\theta}$. (However, since we may not know the nature of $\boldsymbol{\theta}$, there is a risk that this is just a local and not a global minimum. The steepest descent method in eq. (5) is a deterministic method, which means we may get stuck in a local minimum. To avoid this we introduce stochastic gradient descent.)

2.1.2. Momentum

From eq. (5) we have that the movement in parameter space is given by $\mathbf{v}_k$ (the negative of which), which describes the direction and magnitude of the steepest ascent in parameter space. Sometimes we might want to move larger distances in one step. This can be achieved by introduction momentum: We add an addition term to $\mathbf{v}_k$ which lets us rewrite eq. (5) as

$$\begin{aligned} \boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k + \mathbf{v}_k\,; \\ \mathbf{v}_k &= \gamma \mathbf{v}_{k-1} - \eta_k \mathcal{A}_k, \end{aligned} \tag{6}$$

where γ is a momentum parameter and η_k is the same learning parameter as before. The basic idea is that this with this method we "overshoot" the descending step length in the direction of the previous step, with a magnitude that is controlled by γ. By doing this, we may reach the desired minimum with fewer iterations.

There are several modifications we can do to optimise this algorithm. In the Nesterov momentum algorithm (NAG) one applies the following adjustment to the gradient in eq. (6):

$$\begin{aligned} \mathcal{A}_k &\to \nabla_{\tilde{\theta}} \mathcal{L}(\tilde{\boldsymbol{\theta}}_k)\,; \\ \tilde{\boldsymbol{\theta}}_k &= \boldsymbol{\theta}_k + \gamma \mathbf{v}_{k-1}\ ; \end{aligned} \tag{7}$$

This is analagous to the adjustment needed to go from forward Euler to Euler-Cromer as numerical intregration method.
add filler text

2.1.3. Stochasticity

There are several weaknesses to the plain gradient descent, perhaps the largest is the computational expense of on large datasets and its sensitivity of initial conditions and learning rates. If $\mathcal{L}(\boldsymbol{\theta})$ has numerous local minima, we will find one minimum only per set of initial conditions, and we have no good way of saying whether this minimum is global or not. One way of overcoming this is by adding stochasticity to the gradient descent algorithm.

The main idea is that we have n data points, that we divide into $m = n/M$[1] minibatches (subsets), meaning that we have M data points in each minibatch, denoted $\mathcal{B}_j$ for $j \in \{1, 2, \dots, m\}$, s.t. $\bigcup_{j=1}^m \mathcal{B}_j = \bigcap_{j=1}^m \mathcal{B}_j = \mathcal{D}$. We also recognise that we may write the total cost function as a sum over all data points $\mathbf{x}^{(i)}$ for $i \in [1, n]$,

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{(\mathbf{x},y)\in\mathcal{D}} l\big(f(\mathbf{x};\,\boldsymbol{\theta}), y\big) = \sum_{i=1}^n l_i(\boldsymbol{\theta}), \tag{8}$$

where $l_i = l\big(f(\mathbf{x}^{(i)};\boldsymbol{\theta});\, y^{(i)}\big)$, and thus its gradient reads

$$\mathcal{A} = \nabla_\theta \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^n \nabla_\theta l_i(\boldsymbol{\theta}). \tag{9}$$

Now we may approximate the gradient of the cost function by only summing over the data points in a minibatch picked at random:

$$\begin{aligned} \mathcal{A}_k &= \sum_{j=1}^m \mathcal{A}_k^j\,; \\ \mathcal{A}_k \to \mathcal{A}_k^j &= \sum_{i:\mathbf{x}^{(i)}\in\mathcal{B}_j} \nabla_\theta l_i(\boldsymbol{\theta}_k). \end{aligned} \tag{10}$$

The estimate $\mathcal{A}_k^j \approx \mathcal{A}_k$ can be used in our algorithm to ensure stochasticity and relieve computational pressure.

2.1.4. Optimising the learning rate

We can mitigate parts of the struggle with hyperparameter adjustment by using an algorithm with adaptive learning rates. We present a few such schemes in short below, that is different ways of calculating $\mathbf{v}$ in eq. (5).

[1] We of course need to make sure this is an integer.
* *Explain?*

AdaGrad (Goodfellow et al. 2016, algorithm 8.4), from "adaptive gradient (algorithm)", adapts η individually to the components of $\boldsymbol{\theta}$, scaling them as $\eta \to \eta' \sim \eta/\nabla_\theta \mathcal{L}$ (mind the simplification). This method is famously a trouper in convex settings, but has the unfortunate habit of prematurely deciding on the best model in a nonconvex setting.
RMSProp (Goodfellow et al. 2016, algorithm 8.5), from "root mean squared propagation", is a modification to the AdaGrad algorithm. Where AdaGrad performs good, this one learns slow in comparison. However, RMSProp outperforms AdaGrad in nonconvex situations. This improvement introduces an additional hyperparameter ϱ
Adam (Goodfellow et al. 2016, algorithm 8.7), from "adaptive moments",
add filler text
We need two hyperparameters ϱ_1, ϱ_1 for this optimisation scheme.

All three require an original learning rate η and a small number ε for numerical stability.

2.2. Neural Network (NN)

2.2.1. Basics

A feedforward NN (FFNN) is typically built by composing together several functions into a chain of function. Associated with this model is a directed acyclic graph (DAG) describing the explicit structure. The depth of the model is determined by the length of the abovementioned chain. Each function represents a layer in the network. The final layer of an FFNN is the output layer, and the layers between the input (prior to the first) and the output layer are called hidden layers. (Goodfellow et al. 2016)

The structure of such a chain-based architecture is described by the $L-1$ hidden layers $\mathbf{h}^l \in \mathbb{R}^{N_l}$, $l = 1, 2, \dots, L-1$, given by

$$\mathbf{h}^0 = \mathbf{x}^{(i)}\,; \tag{11a}$$
$$\mathbf{h}^1 = g_1\big((W^{0\to1})^\intercal \mathbf{h}^0 + \mathbf{b}^1\big)\,; \tag{11b}$$
$$\mathbf{h}^2 = g_2\big((W^{1\to2})^\intercal \mathbf{h}^1 + \mathbf{b}^2\big)\,; \tag{11c}$$
$$\vdots$$
$$\mathbf{h}^L = g_L\big((W^{L-1\to L})^\intercal \mathbf{h}^{L-1} + \mathbf{b}^L\big)\,; \tag{11d}$$

where we defined $\mathbf{h}^0$ and $\mathbf{h}^L$ to be the input and output layer, respectively.

The matrix of weights $W^{l-1\to1} \in \mathbb{R}^{N_{l-1}\times N_l}$ applies weights and dimension corrections to the previous layer $\mathbf{h}^{l-1} \in \mathbb{R}^{N_{l-1}}$ so that the activation function $g_l : \mathbb{R}^{N_l} \to \mathbb{R}^{N_l}$ can accept the input. Rephrase? The bias $\mathbf{b}^l \in \mathbb{R}^{N_l}$ WTF

2.2.2. Activation functions

A layer $\mathbf{h}^l$ have an associated activation $\mathbf{a}^l \in \mathbb{R}^{N_l}$ containing the weights
* *Is this general?* and biases linked with its neurons. The activation is passed as argument to the activation function g_l whose job is to perform the the affine transformation from one layer to another in an NN. In eq. (11) the activation is $\mathbf{a}^l = (W^{l-1\to l})^\intercal \mathbf{h}^{l-1} + \mathbf{b}^l$, which is valid for $l = 1, 2, \dots L$. Note that the weight matrix $W^{l-1\to l}$ is associated with both the current and previous layer. We can rewrite the formula in eq. (11) as the more compact expression Felt cute, might delete later:

$$\mathbf{h}^0 = \mathbf{x}\,,\quad \mathbf{h}^l = g_l(\mathbf{a}^l)\,,\ l = 1, 2, \dots L\,;$$
$$\mathbf{a}^l = W^{l\leftarrow l-1}\mathbf{h}^{l-1} + \mathbf{b}^l\,; \tag{12}$$

where $W^{l\leftarrow l-1} \equiv (W^{l-1\to l})^\intercal$.

The output is $\hat{\mathbf{y}}^{(i)} = \mathbf{h}^L \in \mathbb{R}^{N_L}$.

$$\sigma(\boldsymbol{\xi}) = \frac{1}{1+e^{-\boldsymbol{\xi}}} = 1 - \sigma(-\boldsymbol{\xi}) \tag{13a}$$
$$\tanh(\boldsymbol{\xi}) = \frac{e^{2\boldsymbol{\xi}}-1}{e^{2\boldsymbol{\xi}}+1} = 2\sigma(2\boldsymbol{\xi}) - 1 \tag{13b}$$
$$\mathrm{ReLU}(\boldsymbol{\xi}) = \max(0, \boldsymbol{\xi}) = \begin{cases} \boldsymbol{\xi}, & \boldsymbol{\xi} > 0 \\ 0, & \boldsymbol{\xi} \le 0 \end{cases} \tag{13c}$$
$$\mathrm{ReLU}^*(\boldsymbol{\xi}) = \begin{cases} \boldsymbol{\xi}, & \boldsymbol{\xi} > 0 \\ 0.01\boldsymbol{\xi}, & \boldsymbol{\xi} \le 0 \end{cases} \tag{13d}$$
$$\zeta(\boldsymbol{\xi}) = \frac{e^{\boldsymbol{\xi}}}{\sum_{j=1}^N e^{\xi_j}} \tag{13e}$$

Assuming some $\boldsymbol{\xi} \in \mathbb{R}^K$, the set of expressions (13) show some well-known activation functions $\mathbb{R}^K \to \mathbb{R}^K$. To translate into NN-components, set $K \to N$, $\boldsymbol{\xi} \to \mathbf{a}$ and e.g. $g = \tanh$. The oldest and probably most famous is the slow-learning sigmoid function σ in eq. (13a). The hyperbolic tangent in eq. (13b) is closely related to the sigmoid, and is typically performing better (Goodfellow et al. 2016). The ReLU (eq. (13c)) or leaky ReLU (eq. (13d)) activation function provides output of the type that is easy to interpret as it resembles the linear unit. ReLU typically learns fast, but has the the unfortunate habit of killing neurons. That is to say, some neurons are deactivated for any input. The leaky ReLU can omit this issue somewhat, but there hatch is a perfomance reduction. The softmax ζ in eq. (13e)
add filler text

2.2.3. Back propagation

The information in an FFNN accepting input $\mathbf{x}$ to produce output $\hat{y}$ (check consistency!) is flowing *forward* (Goodfellow et al. 2016), hence the name. The initial information from $\mathbf{x}$ propagates through the hidden layers resulting in the production of $\hat{y}$. This information flow is called forward propagation. During training, we let forward propagation yield a cost, $\mathcal{L}(\boldsymbol{\theta})$. We may reverse this process, i.e. let $\mathcal{L}(\boldsymbol{\theta})$ provide infomation that propagates backwards through the network in order to compute the gradient, $\nabla_\theta \mathcal{L}(\boldsymbol{\theta})$, corresponding to an algorithm called back-propagation.

2.3. Classification

2.4. Logistic regression

The standard logistic function $p : \mathbb{R} \to (0, 1)$ may be written as

$$p(\xi) = \frac{1}{1+e^{-(\beta_0+\beta_1\xi)}}, \tag{14}$$

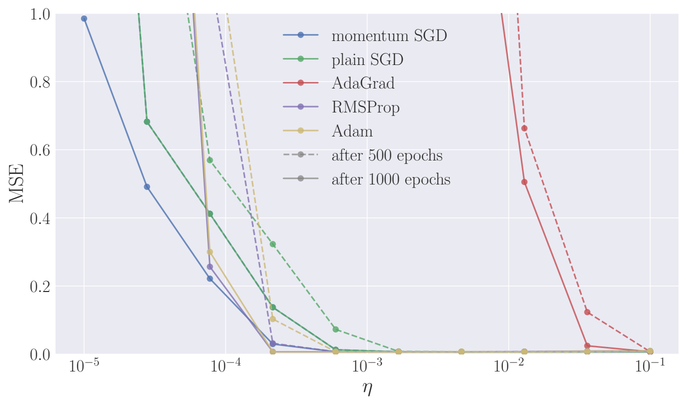

Fig. 1. This very cute figure is ...

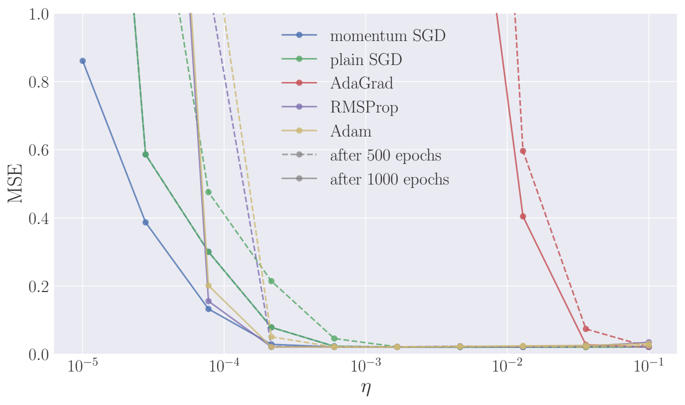

Fig. 2. This very cute figure is unnecessary! Maybe remove? Or make subplot? lambda is 0.1 i think

and is indeed the sigmoid function in eq. (13a) if we substitute $\xi \to \beta_0 + \beta_1 \xi$.

add filler text

$$\mathcal{L}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left[y^{(i)} \left(\beta_0 + \beta_1 x^{(i)} \right) - \log \left(1 + e^{\beta_0 + \beta_1 x^{(i)}} \right) \right] \tag{15}$$

3. Analysis

* *Present datasets???*
* *Maybe write something about the codes?*

OUTLINE:

1. Gradient descent
   - OLS and Ridge on regression problem
2. Building our FFNN
3. Regression problem
   - try different activation functions
4. Classification problem
   - compare with logistic regression

3.1. Gradient descent

We write a code that

add filler text

Using the SGD method, we perform an OLS regression on a dataset generated by a third order polynomial with some added noise,

$$f(x) = ax + bx^2 + cx^3 + d\mathcal{N}(0,1). \tag{16}$$

In particular, we aim to minimise the cost function in eq. (2a) for which we need to tune the learning rate η. We perform the same analysis using the Ridge cost function in eq. (2b), but here we need to tune the penalty parameter λ as well as the learning rate η.

We do not present many figures to describe this part of the analysis. We justify this by arguing that the purpose of this part is to test the SGD code, and then by extension the GD code, and give an idea of the effect of changing optimisers.

In Figure 1

add filler text

3.2. Neural network

We will build our FFNN ((i)-(iii)) and solve a supervised learning problem ((iv)-(vi)) using the steps listed below (Hjorth-Jensen 2021).

(i) Collect and prepocess data, that is we extract 80% of the dataset and reserve the rest for validation. The data is then scaled using standard score normalisation[2] with respect to the training data.

(ii) Define the model and design its architecture. In practice, this means to decide on hyperparameters of the NN such as depth (L) and activation function(s) (g).

(iii) Choose loss function and optimiser. For regression we will use the regular MSE score (ref!) as the estimator of loss, whereas the classification problem estimates the loss according to the accuracy score (ref!). please send help The choice of optimiser is between GD and SGD, but includes decisions about the exact algorithm (how we optimise the learning rate).

(iv) Train the network to find the right weights and biases.

(v) Validate model, i.e. assess model performance by applying it on the test data.

(vi) Adjust hyperparameters, and if necessary review the network architecture. That is to say, if the result is not satisfactory even after tuning the hyperparameters, return to step (ii) and start over from there.

3.3. Regression problem

Our dataset is once again fictional as it is generated by the Franke function from project 1[3] for a set of coordinates in the plane. We split and standardise the 20×20 datapoints in accordance with step (i).

3.4. Classification problem

4. Conclusion

*** Make sure not to include discussion here, Nanna! (I know you want to...)**
*** Future work!**

[2] Formula found in Hjorth-Jensen (2021), or page 6 of the project 1-report.
[3] Equation (10) in the report.

Code availability

The code is available on GitHub at `https://github.com/Johanmkr/FYS-STK4155colab/tree/main/project2`.

References

Goodfellow, I., Bengio, Y., & Courville, A. 2016, Deep Learning (MIT Press), accessed Nov. 5 2022 at `http://www.deeplearningbook.org`

Hjorth-Jensen, M. 2021, Applied Data Analysis and Machine Learning (Jupyter Book), accessed Nov. 10 at `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html`