Astronomy & Astrophysics manuscript no. project2
November 15, 2022
©ESO 2022

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 15, 2022 GitHub repo link: https://github.com/Johanmkr/FYS-STK4155colab/tree/main/project2

ABSTRACT

ABSTRACT

Notation and nomenclature

Nanna will fix this towards the end

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 $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$
- 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 Regularisation parameter (penalty parameter in Ridge regression)
- m Number of minibatcher

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

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

0.6. Acronyms

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

Linear regression assumes a linear relationship between a set of p features $\mathbf{x} \in \mathbb{R}^p$ and an observed value $y \in \mathbb{R}$. We assume there to exist a *continous* function of the input $\mathbf{x}$ giving the output $\hat{y}$. The coefficients $\boldsymbol{\theta} \in \mathbb{R}^p$ that determine said function can be estimated using a variety of methods, as discussed in project 1. In any case, the aim is to minimise some loss function $\mathcal{L}(\boldsymbol{\theta}; \hat{y}, y)$ with respect to this parameter vector ($\boldsymbol{\theta}$) describing what we sacrifice by using this exact model.

What if the function we want to fit is *discontinous*? We consider the binary situation where the observed y only takes one of two discrete values; 0 or 1. Logistic regression proposes a model where the output $\hat{y}$ is obtained from a probability distribution and subsequently a befitting total loss function that can be minimised with respect to a set of parameters $\boldsymbol{\theta}$. Now, instead of using the method-specific regression algorithms for finding the optimal $\boldsymbol{\theta}$, we can change modus operandi and focus solely on the minimisation of some objective function (e.g. a loss function such as the MSE). For this purpose, we may use the very powerful procedure of steepest descent.

Where the actual (physical) relationship between some dependent and independent variable is not of utmost importance, and the main aim is to predict the outcome given some setting, supervised learning problems may also be solved using neural networks.

* NN - no parameter vector - allow multivariable y

2.1. Stochastic gradient descent (SGD)

SGD and its subvariants are frequently used optimisation algorithms in machine learning (Goodfellow et al. 2016). The more basic algorithm known as gradient descent (GD) is technically a specific case of SGD[1] follows the gradient of some objective function J downhill in some parameter space. The effect of introducing stochasticity is discussed in section 2.1.3. The result is a flexible way to locate the minima of any J, exactly what we wished for. The ordinary least squares and Ridge schemes of linear regression that we discussed in project 1, are then implemented by using the mean squared error (MSE) function with an ℓ^2-norm regularisation term,

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2 + \frac{\lambda}{2p}\sum_{j=1}^{p}\theta_j^2. \tag{1}$$

λ is the penalty term set to zero for OLS and a small positive value for Ridge regression. The output $\hat{y}^{(i)}$ is resulting from some function evaluated at $\mathbf{x}^{(i)}$, which for our purposes reads $\hat{y}^{(i)} = [\mathbf{x}^{(i)}]^\intercal\boldsymbol{\theta}$.

add filler text

2.1.1. Plain gradient descent

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

$$\boldsymbol{\xi}_{k+1} = \boldsymbol{\xi}_k - \eta_k \nabla_\xi J(\boldsymbol{\xi}_k), \tag{2}$$

where the learning rate η_k may follow a schedule in k or stay constant. In the following, we consider a constant global[2] learning rate $\eta_k = \eta$.

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. (2) reads

$$\begin{aligned}\boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k + \mathbf{v}_k\,;\\ \mathbf{v}_k &= -\eta\mathcal{A}_k,\end{aligned} \tag{3}$$

for substitutions $J \to \mathcal{L}$ and $\boldsymbol{\xi} \to \boldsymbol{\theta}$. For a sufficiently small η, this method will converge to a minimum of $\boldsymbol{\theta}$. However, since we may not know the nature of $\mathcal{L}$ in parameter space, there is a risk that said extremum is just a local and not a global minimum. The steepest descent method in eq. (3) is a deterministic method, which means we may get stuck in a local minimum. There are several ways around this, and one such way is to include an element of randomness in the computations, as we will see in section 2.1.3.

[1] The case with number of minibatches $m = 1$.

[2] Emphasising "global" here to distinguish from the *actual* rate of learning (or step size) which may depend on the specific update rule we choose.

2.1.2. Momentum

From eq. (3) 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. (3) as

$$\begin{aligned}\boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k + \mathbf{v}_k\,;\\ \mathbf{v}_k &= \gamma\mathbf{v}_{k-1} - \eta\mathcal{A}_k,\end{aligned} \tag{4}$$

where γ is a momentum parameter and η is the same global 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.

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 with the n datapoints which we have in a dataset $\mathcal{D}$, we can create m subsets, meaning that we have n/m[3] datapoints in each *minibatch*, denoted $\mathcal{B}_j$ for $j \in \{1, 2, \ldots, m\}$, s.t. $\bigcup_{j=1}^{m}\mathcal{B}_j = \bigcap_{j=1}^{m}\mathcal{B}_j = \mathcal{D}$. We recognise that we may write the total cost function as a sum over all data points $\mathbf{x}^{(i)}$ for $i \in [1, n]$,

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{(\mathbf{x},y)\in\mathcal{D}} l\big(f(\mathbf{x};\boldsymbol{\theta}), y\big) = \sum_{i=1}^{n} l_i(\boldsymbol{\theta}), \tag{5}$$

where $l_i(\boldsymbol{\theta}) = l\big(f(\mathbf{x}^{(i)};\boldsymbol{\theta}); y^{(i)}\big)$ is the per-example loss function. Thus, its gradient is written

$$\mathcal{A} = \nabla_\theta\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n}\nabla_\theta l_i(\boldsymbol{\theta}). \tag{6}$$

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

$$\begin{aligned}\mathcal{A}_k &= \sum_{j=1}^{m}\mathcal{A}_k^j\,;\\ \mathcal{A}_k \to \mathcal{A}_k^j &= \sum_{i:\mathbf{x}^{(i)}\in\mathcal{B}_j}\nabla_\theta l_i(\boldsymbol{\theta}_k)\,;\end{aligned} \tag{7}$$

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

[3] Give or take; needs to be an integer.

2.1.4. Optimising the learning rate

There is in general no way of knowing a priori what value η or γ should take. Tuning such hyperparameters makes up a significant part of the work in a supervised learning problems. 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. (3). All require an original learning rate η and a small number ε for numerical stability.

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. By default: $\varepsilon = 10^{-7}$.
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, a decay rate controlling the length scale of the moving average. By default: $\varrho = 0.9$ and $\varepsilon = 10^{-7}$.
Adam (Goodfellow et al. 2016, algorithm 8.7), from "adaptive moments", calculates the update based on both the first-order momentum, the same as in momentum, and the second-order momentum, much like in RMSProp. We need two hyperparameters ϱ_1 and ϱ_2 for this optimisation scheme, representing the decay rate of the first and second moment, respectively. By default: $\varrho_1 = 0.9$, $\varrho_2 = 0.999$ and $\varepsilon = 10^{-8}$.

The choice of default values is inspired by (Goodfellow et al. 2016, chapter 8.5).

2.2. Neural Network (NN)

We have so far discussed regression and given a lot of attention to the coefficients $\boldsymbol{\theta}$ that we assume describe some physical relationship between a set of feature values and a target. More complex problems require more complex models, and many phenomena may not even be possible to describe with a smooth function. Deep learning models usually pay more attention to the output, in this section denoted $\hat{\mathbf{y}}$ to allow multivariable outputs[4], after training, that is.

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)

[4] For completeness. Will not be relevant to think of the output as a vector in our case.

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

$$\mathbf{h}^0 = \mathbf{x}^{(i)}\,; \tag{8a}$$
$$\mathbf{h}^1 = g_1\big((W^{0\to1})^\intercal \mathbf{h}^0 + \mathbf{b}^1\big)\,; \tag{8b}$$
$$\mathbf{h}^2 = g_2\big((W^{1\to2})^\intercal \mathbf{h}^1 + \mathbf{b}^2\big)\,; \tag{8c}$$
$$\vdots$$
$$\mathbf{h}^L = g_L\big((W^{L-1\to L})^\intercal \mathbf{h}^{L-1} + \mathbf{b}^L\big)\,; \tag{8d}$$

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

The matrix of weights $W^{l-1\to l} \in \mathbb{R}^{N_{l-1}\times N_l}$ 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. The bias $\mathbf{b}^l \in \mathbb{R}^{N_l}$ may be interpreted as a safety mechanism of the neurons to prevent their layer value to become zero, and is typically set to a small nonzero value (Goodfellow et al. 2016).

2.2.2. Activation functions

A layer $\mathbf{h}^l$ has an associated activation $\mathbf{a}^l \in \mathbb{R}^{N_l}$ which is a function of the previous layer values, $\mathbf{h}^{l-1}$, the weights, $W^{l-1\to l}$, and the biases linked with each neuron, $\mathbf{b}^l$. The activation is passed as argument to the activation function g_l whose job is to perform the affine transformation from one layer to another in a NN. In eq. (8) the activation is $\mathbf{a}^l = (W^{l-1\to l})^\intercal \mathbf{h}^{l-1} + \mathbf{b}^l$, which is valid for $l = 1, 2, \ldots L$. 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. (8) as the more compact expression:

$$\mathbf{h}^0 = \mathbf{x}\,, \quad \mathbf{h}^l = g_l(\mathbf{a}^l)\,,\ l = 1, 2, \ldots L\,;$$
$$\mathbf{a}^l = W^{l\leftarrow l-1}\mathbf{h}^{l-1} + \mathbf{b}^l\,; \tag{9}$$

where $W^{l\leftarrow l-1} \equiv (W^{l-1\to l})^\intercal$. The output is $\hat{\mathbf{y}}^{(i)} = \mathbf{h}^L \in \mathbb{R}^{N_L}$.

We present some examples of commonly used activation functions:

$$\sigma(\boldsymbol{\xi}) = \frac{1}{1+e^{-\boldsymbol{\xi}}} = 1 - \sigma(-\boldsymbol{\xi}) \tag{10a}$$
$$\tanh(\boldsymbol{\xi}) = \frac{e^{2\boldsymbol{\xi}}-1}{e^{2\boldsymbol{\xi}}+1} = 2\sigma(2\boldsymbol{\xi}) - 1 \tag{10b}$$
$$\mathrm{ReLU}(\boldsymbol{\xi}) = \max(0, \boldsymbol{\xi}) = \begin{cases} \boldsymbol{\xi}, & \boldsymbol{\xi} > 0 \\ 0, & \boldsymbol{\xi} \le 0 \end{cases} \tag{10c}$$
$$\mathrm{ReLU}^*(\boldsymbol{\xi}) = \begin{cases} \boldsymbol{\xi}, & \boldsymbol{\xi} > 0 \\ 0.01\boldsymbol{\xi}, & \boldsymbol{\xi} \le 0 \end{cases} \tag{10d}$$

Assuming some $\boldsymbol{\xi} \in \mathbb{R}^K$, the set of expressions (10) 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. (10a). The hyperbolic tangent in eq. (10b) is closely related to the sigmoid, and is typically performing better (Goodfellow et al. 2016). The ReLU (eq. (10c)) or leaky ReLU (eq. (10d)) 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 the hatch is a perfomance reduction.

2.2.3. Back propagation

The information in an FFNN accepting input $\mathbf{x}$ to produce output $\hat{\mathbf{y}}$ 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 $\hat{\mathbf{y}}$ which is the output of the final layer. This information flow is called forward propagation or forward pass. Training the network (tuning the weights and biases) consist of running forward propagation and compare the resultant output $\hat{\mathbf{y}}$ with the desired output $\mathbf{y}$, i.e. evaluate the loss function, $\mathcal{L}(\boldsymbol{\theta})$.

The art of back propagation is to reverse this process. We let $\mathcal{L}(\boldsymbol{\theta})$ provide information about the error of the output layer, that propagates backwards through the network in order to compute the gradient of the loss function for each layer, $\nabla^l_\theta \mathcal{L}(\boldsymbol{\theta})$. These gradients are used to update the weights and biases of each layer in such a way that when forward propagation is run again, the overall output loss will be lower. Over time, we propagate forwards a backwards in order to minimise the loss function, typically using stochastic gradient descent, as explained in section 2.1.3. The optimiser of choice (section 2.1.4) takes the gradients found from back propagation, (and hyperparameters) as inputs and optimises the weights and biases accordingly. A thorough walkthrough of the back propagation algorithm can be found in (Goodfellow et al. 2016, chapter 6.5).

2.3. Regression

When using a FFNN for a regression problems we opt to fit a function to a certain data set. One approach to this is to let the number of features be the dimensionality of the input data, and the number of data points be the data points in the domain we are considering. In this investigation we will use the two dimensional Franke function from project 1[5] on a 20×20 grid (more on this later).

Since we want to fit a function to data points, the obvious way of measured the error from the output layer is by considering the mean squared error (MSE). Thus, our loss function will as given in eq. (1) with a tunable hyperparameter λ.

We may use a varying number of hidden layers and neurons, depending on the data we are trying to fit. The architecture of the network is a problem dependant feature and must be tested for. The same goes for the activation functions described in eq. (10). However, the output function g_L can just to be the linear function: $g_L(\mathbf{a}^L) = \mathbf{a}^L$.

Felt cute, might delete later:

This means that the number of features is $p = 2$ input dimensions, for $n = 400$ data point. Each data point (a combination of p features) will yield one single output.

[5] Equation (10) in the paper.

2.4. Classification

For the case of classification, we typically have many features in the input data, which results in a more complex design matrix. The output of the network is structured into two main classes, binary and multivariate classification. In binary classification we have one single output node that ideally should be either 0 or 1. We achieve this by having a sigmoid output function. In multivariate classification on the other hand, we have multiple output nodes and a probability distribution between them. This probability distribution is found using the softmax function. However, the data set we will analyse is the Wisconsin Breast Cancer dataset (Pedregosa et al. 2011) which needs binary classification. Thus the main focus will be on the binary classification technique.

When evaluating the loss we use eq. (**??**)

2.5. Logistic regression

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

$$p(\xi) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 \xi)}}, \tag{11}$$

and is indeed the sigmoid function in eq. (10a) if we substitute $\xi \to \beta_0 + \beta_1 \xi$.

add filler text

* *This is rubbish*

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \left[y^{(i)} \left(\theta_0 + (X\boldsymbol{\theta})^{(i)} \right) - \log \left(1 + e^{\theta_0 + \boldsymbol{\theta}^\intercal \mathbf{x}^{(i)}} \right) \right] \tag{12}$$

In terms of weights and biases ($(\theta_0, \boldsymbol{\theta}) \to (\mathbf{b}, W)$),

$$\mathcal{L}(\mathbf{b}, W) = -\frac{1}{n} \sum_{i=1}^{n} \left[y^{(i)} \left(\mathbf{b} + W^\intercal \mathbf{h}^l \right) - \log \left(1 + e^{\theta_0 + \boldsymbol{\theta}^\intercal \mathbf{x}^{(i)}} \right) \right] \tag{13}$$

$$\mathcal{L}(l) = -\frac{1}{N_l} \left[(\mathbf{h}^l)^\intercal \mathbf{a}^l - \log \left(1 + e^{\mathbf{a}^l} \right) \right] \tag{14}$$

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

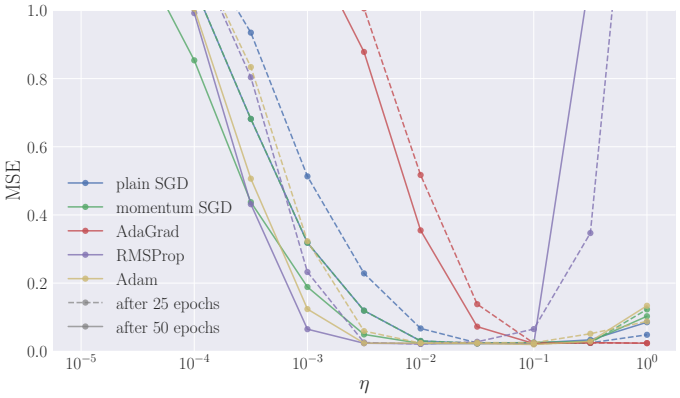

Fig. 1. The graphs show how the test MSE evolves as the global learning rate η increases for different update rules in the SGD algorithm. The penalty parameter is $\lambda = 0.1$, and we have used $m = 40$ minibatches. $\gamma = 0.5$ for the momentum SGD and the other relevant hyperparameters are set to their defaults in accordance with section 2.1.4. The dashed graphs show the MSE after 25 iterations and the solid graphs represent the MSE after 25 additional iterations. All solvers were initilised by the same random vector $\boldsymbol{\theta}_0$.

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,

$$f(x) = 2.0x + 1.7x^2 - 0.40x^3 + 0.10\mathcal{N}(0,1), \tag{15}$$

and we consider $n = 400$ datapoints. In particular, we aim to minimise the cost function in eq. (1) with $\lambda = 0$ for which we need to tune the learning rate η. We perform the same analysis using the Ridge cost function, i.e. $\lambda > 0$ in eq. (1), 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

We will study more thoroughly the dependencies on the number of epochs, the number of minibatches (m) and the penalty parameter (λ) in the NN analysis of the Franke function in section 3.2.

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[6] 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 (1) as the estimator of loss, whereas the classification problem estimates the loss according to the cross entropy (12). We will use SGD as optimiser, but we have various alternatives for the exact optimisation algorithm (see section 2.1.4).

(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[7] with an added noise of $0.1\mathcal{N}(0,1)$ for a set of coordinates in the plane. We split and standardise the 20×20 datapoints, which concludes step (i).

3.4. Classification problem

4. Conclusion

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

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

Pedregosa, F., Varoquaux, G., Gramfort, A., et al. 2011, Journal of Machine Learning Research, 12, 2825

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

[7] Equation (10) in the report.