

Neural Networks: An introduction to regression and classification

René Ask, Kaspera Skovli Gåsvær & Maria Linea Horgen

(Dated: November 3, 2020)

2 Do List Project2 FYSSTK:

RIDGE OG OLS: -Analyser med hensyn på eta, mini-batches, epochs

-Sammenlikn/diskuter resultater med proj1 ELLER Scikit

-For ridge: analyse av lambda og eta

FFNN: -Har vi: Fleksibelt antall hidden layers, vektor init med normal dist?

-Hvordan vil vi init bias? diskuter.

-Hvilken activation func for output layer, diskuter.

Sammenlikne a,b og proj1!

-Sammenlikn FFNN med OLS og Ridge fra proj1, kommenter og gjør krise analyse av resultatene.

-Sammenlikn med scikit-learn.

-Analyser regulariserings param og læringsrate for å finne optimal MSE og R2 score.

-Test Sigmoid, ReLu og Leaky ReLu og diskuter resultatene.

-Klassifisering: Bruk softmax (men koden må kunne bruke en binær aktiveringsfunksjon også).

-Se på accuracy score.

-Diskuter resultatene og gi kritisk analyse av params, inkl hyperparams som eta og lambda, forskjellige aktiveringsfunksjoner, antall hidden layers og noder.

-Sammenlikn med Scikit-learn/tensorflow.

-Logreg: Definer cost funk og design matrise.

-Studer resultatene som en funksjon av eta.

-Legg til R2 regu, sammenlikn resultatene med FFNN og scikit learn.

-Oppsummer de forskjellige algoritmene og kom med en kritisk evaluering av fordeler og ulemper hos alle sammen.

55 000 - Trening 5 000 - validering 10 000 - test

I. INTRODUCTION

dating scheme

Skriv all shiten vi skal gjøre. Skal gjøre SGD på franke function og sammenligne med lineær regresjon

$$\begin{aligned} \mathbf{v}_t &= \eta_t \nabla_{\theta} E(\theta_t), \\ \theta_{t+1} &= \theta_t - \mathbf{v}_t. \end{aligned} \tag{1}$$

Here $\nabla_{\theta} E(\theta_t)$ is the gradient of E with respect to the parameters θ in time step t and η_t is the learning rate which controls the step size we take in the direction of the gradient.

Determining the learning rate requires several aspects to be taken into consideration: choosing η_t too small is computationally expensive, i. e. the number of iterations needed to converge to the local minimum becomes huge. Depending on the convexity of our cost function $E(\theta)$, the method will either converge towards a (possible) local or global minimum. Without making any assumptions about E , the method will hence converge towards a local minimum. With a high learning rate we risk overshooting the minimum, so choosing η_t wisely is crucial for obtaining satisfactory results.

There are several variants of gradient decent methods, in this report we confine ourselves to two: stochastic gradient descent (SGD) with mini-batches and SGD with momentum. There are a few drawbacks with GD: com-

II. FORMALISM

A. Gradient Descent

Assume we are given a dataset $\mathcal{D} = (\mathbf{X}, \mathbf{z})$ of length n , where \mathbf{X} is some observed real data and \mathbf{z} is the response variable. The goal is to explain \mathbf{z} through a model $f(\mathbf{X}, \theta)$, which is a function of the parameters θ . To determine how well the model describes the observed data, we define a cost function $E(\theta) = \mathcal{C}(\mathbf{z}, f(\mathbf{X}, \theta))$. Choosing the parameters in such a way that they minimize the cost function is the core of many machine learning algorithms. In gradient descent (GD) the parameters are initialized to some value θ_0 , and then iteratively updated in the direction of the steepest descent through the up-

puting the gradient for each datapoint when n is large is, to say the least, time consuming, it lacks stochasticity and treats all directions in parameter space uniformly, i. e. the learning rate is constant.

1. Stochastic Gradient Descent with mini-batches

To incorporate randomness in GD the gradient is approximated on subsets of the observed data, called *mini-batches*, rather than being calculated in each datapoint \mathbf{x}_i . The batch size is normally significantly smaller than n , normally ranging from a ten to a couple of hundred datapoints [4].

Denoting the batch size as B , there are b_k mini-batches for $k = 1, \dots, n/B$. A full iteration over all the n/B mini-batches, or all the datapoints n , is called an *epoch*. The datapoints included in each mini-batch are randomly chosen, either with- or without replacement, and thus introduces the missing stochasticity. For the case with replacement, some datapoints might be picked several times during an epoch, while others may not be picked at all. This approach generally converges more rapidly than the one not including replacement [3].

When taking a gradient descent step, the steepest descent direction is now approximated by a mini-batch b_k , as opposed to the full dataset, meaning

$$\nabla_{\theta} E(\theta) = \sum_{i=1}^n \nabla_{\theta} E_i(\theta) \rightarrow \nabla_{\theta} E^{MB}(\theta) = \sum_{i \in B_k} \nabla_{\theta} E_i(\theta), \quad (2)$$

with $\nabla_{\theta} E^{MB}(\theta)$ now being the mini-batch approximation. This tackles the time consumption problem that arises with the regular GD method. The updating scheme becomes

$$\begin{aligned} \mathbf{v}_t &= \eta_t \nabla_{\theta} E^{MB}(\theta_t), \\ \theta_{t+1} &= \theta_t - \mathbf{v}_t. \end{aligned} \quad (3)$$

2. Adding momentum

A modification which deals with the constant learning rate η_t is to add a momentum term in the updating scheme,

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta_t \nabla_{\theta} E^{MB}(\theta_t), \\ \theta_{t+1} &= \theta_t - \mathbf{v}_t. \end{aligned} \quad (4)$$

The learning rate is still constant per se, but the momentum term serves as a memory of the previous time step by adding a fraction $\gamma \in [0, 1]$ of the update vector of the past time step to the current update vector, increasing the convergence rate.

B. SGD vs. Linear Regression

As mentioned above, in this report we will compare the results from our regression analysis in [1] for Frankes function with ordinary least squares (OLS) and Ridge regression with the ones obtained with SGD. This includes the mean square error (MSE) score, polynomial degree and the hyper-parameter λ in Ridge regression. For a thorough walk-through on how we preprocess the data, we refer to our previous work in [1].

In regression problems, the function $f(\mathbf{X}, \theta)$ can be expressed as a matrix-vector product

$$f(\mathbf{X}, \mathbf{w}) = \mathbf{X}\mathbf{w}, \quad (5)$$

with \mathbf{X} as the design matrix and $\mathbf{w} = \theta$ as the weights in the linear model. The cost function we seek to minimize has a two slightly different definitions, given by the regression type, yielding two different expressions for the gradient:

$$\begin{aligned} \nabla_{\mathbf{w}} E_{\text{OLS}} &= \frac{\partial}{\partial \mathbf{w}} \frac{1}{2} \|\mathbf{z} - \mathbf{X}\mathbf{w}\|_2^2, \\ &= -\mathbf{X}^T \mathbf{z} + \mathbf{X}^T \mathbf{X} \mathbf{w}, \end{aligned} \quad (6)$$

$$\begin{aligned} \nabla_{\mathbf{w}} E_{\text{Ridge}} &= \frac{\partial}{\partial \mathbf{w}} \left(\frac{1}{2} \|\mathbf{z} - \mathbf{X}\mathbf{w}\|_2^2 + \frac{\lambda}{2} \|\mathbf{w}\|_2^2 \right), \\ &= -\mathbf{X}^T \mathbf{z} + \mathbf{X}^T \mathbf{X} \mathbf{w} + \lambda \mathbf{w}. \end{aligned} \quad (7)$$

Both E_{OLS} and E_{Ridge} are convex functions, see section V A for a simple proof. Any local minimum of a convex function is also a global minimum, thus the solution obtained from SGD will necessarily converge towards the global minimum through an iterative scheme.

SPØR OM KODEN PÅ ONSDAG FØR MAN BEGYNNER Å SAMLE INN RESULTATER!!!

C. Logistic regression

Logistic regression is a supervised learning method used to predict discrete outputs $y \in \{0, 1\}$ given an input $x \in \mathbb{R}^p$. Problems of this kind is known as classification problems. Classification problems can consist of a single class, where the task is to predict whether x belongs to a category ($y = 1$) or not ($y = 0$). More generally, classification problems can consist of M classes such that the output $y \in \{0, 1\}^M$ belongs to a single class m known as *one-hot vector* where

$$y_i = \begin{cases} 1 & \text{if } i = m \\ 0 & \text{else} \end{cases} \quad (8)$$

The classification problem is then to predict which class m a given input x belongs to.

The classification problem may be divided further as follows:

- *Hard* classification in which the predicted values are only either 0 or 1. In this case, the model is called a hard classifier.
- *Soft* classification where the predicted values are on the real interval $[0, 1]$. The model is called a soft classifier. A soft classifier will yield a probability of whether x belongs to a class or not.

To implement an algorithm for Logistic Regression we must first look at its corresponding cost function. Suppose the output of our model $\hat{y} \in \{0, 1\}^M$ is made up of M classes. The cross-entropy for a single datapoint (x, y) with $x \in \mathbb{R}^p$ and $y \in \{0, 1\}^M$, is given by

$$E = - \sum_{m=1}^M y_m \log \hat{y}_m + (1 - y_m) \log(1 - \hat{y}_m). \quad (9)$$

The gradients for a single data point are

$$\frac{\partial E}{\partial W_{jk}} = (\sigma_j - y_j) x_k. \quad (10)$$

Similarly, for the bias term, we get

$$\frac{\partial E}{\partial b_j} = \sigma_j - y_j. \quad (11)$$

See Appendix for a more thorough derivation of the gradients.

To implement a soft-classifier, one typically use the Softmax function to compute the probabilities

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{m=1}^M e^{z_m}}, \quad (12)$$

where $z_j = \sum_i W_{ji} x_i + b_j$, for some weight matrix W and bias vector b .

D. Neural Nets

Neural nets (or neural networks) are nonlinear machine learning models for supervised learning. The basic building blocks of the model class are a set of layers, each of which has a set of neurons or nodes. These layers are usually divided into three categories:

- The *input* layer which is where a real observable is fed to the network.
- *Hidden* layers which is where the processing of the information happens.
- The *output* or *top* layer where a response or activation associated with the observable is realized.

Between any two adjacent layers, there's a set of weights connecting nodes from one layer to every node in the adjacent layer. Typically, a bias is added to each node in every layer to help facilitate activation of the node.

1. Basic mathematical formalism

Denote a layer in the neural network by l . Let there be L layers in total such that $l \in \{1, 2, \dots, L\}$. Denote the activation of neuron j in layer l by a_j^l its corresponding bias as b_j^l . Finally, let W_{jk}^l be the weights connecting neuron k in layer $l-1$ to neuron j in layer l . The activation a_j^l is modelled through the nonlinear equation

$$a_j^l = \sigma \left(\sum_k W_{jk}^l a_k^{l-1} + b_j^l \right) \equiv \sigma(z_j^l), \quad (13)$$

where $\sigma(\cdot)$ is some nonlinear function colloquially called the *activation function*. We'll come back to a discussion on common choices later on.

For $l=1$, eq. (13) specializes to

$$a_j^1 = \sigma \left(\sum_k W_{jk}^1 x_k + b_j^1 \right), \quad (14)$$

where $x \in \mathbb{R}^p$ is the input vector containing p features.

2. Training a neural network with the backpropagation algorithm

A neural network can be represented by a function of the form

$$f = f_1 \circ f_2 \circ \dots \circ f_L, \quad (15)$$

who's gradient carry a computational complexity which makes a direct calculation intractable. A clever circumvention is to create a set of equations defining a recursive relationship such that knowledge of the error in any layer $l+1$ permits us to deduce of the error in layer l . This is known as the *backpropagation algorithm*. We'll simply recite the important equations and their meaning, and refer the reader to a derivation here [4].

The backpropagation boils down to four equations which we'll describe in the following. Suppose E is some cost-function and let Δ_j^l denote the error at neuron j in layer l . The first of the four equations is the error at layer L which is defined as

$$\Delta_j^L = \frac{\partial E}{\partial z_j^L} \quad (16)$$

The second equation is a recursive equation which allows us to compute the error at layers $l = L-1, \dots, 1$ from Δ_j^L . The equation is given by

$$\Delta_j^l = \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1} \right) \sigma'(z_j^l), \quad (17)$$

where $\sigma'(\cdot)$ denotes the derivative of $\sigma(\cdot)$.

The third equation relates the errors to the gradient of the cost-function with respect to the biases which is

$$\frac{\partial E}{\partial b_j^l} = \Delta_j^l. \quad (18)$$

Finally, the fourth equation relates the errors to the gradient with respect to the weights:

$$\frac{\partial E}{\partial W_{jk}^l} = \Delta_j^l a_k^{l-1} \quad (19)$$

Before we can list the complete backpropagation algorithm, we need an explicit expression for Δ_j^L , which the next section delves into.

3. Top layer activation and choice of cost-function

The rate at which a neural network learns, is dependent on the choice of top layer activation and choice of cost-function. We will consider two cost-functions depending on the problem at hand.

Classification For classification, a natural choice for the top layer activation function is the Softmax function in eq. (12), which would make the neural network a soft classifier. A common choice for the cost-function is then the cross-entropy function, which for a neural network becomes

$$E = - \sum_m [y_m \log a_m^L + (1 - y_m) \log (1 - a_m^L)]. \quad (20)$$

Starting from eq. (16), the error at layer L can be computed as

$$\begin{aligned} \Delta_j^L &= \frac{\partial E}{\partial z_j^L} = \frac{\partial E}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \\ &= - \sum_m \left[\frac{y_m}{a_m^L} - \frac{1 - y_m}{1 - a_m^L} \right] \delta_{jm} a_j^L (1 - a_j^L) \\ &= a_j^L - y_j, \end{aligned} \quad (21)$$

meaning the error is simply given by the difference between the prediction a_j^L and the ground truth y_j .

Regression For regression, a natural choice for top layer activation is simply a linear relationship

$$a_j^L = z_j = \sum_k W_{jk}^L a_k^{L-1} + b_j^L. \quad (22)$$

In this case, choosing the cost-function (for a single data point) to be

$$E = \frac{1}{2} \sum_j (y_j - a_j^L)^2, \quad (23)$$

gives the following error at layer L

$$\Delta_j^L = \frac{\partial E}{\partial z_j^L} = \frac{\partial E}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = a_j^L - y_j, \quad (24)$$

which is exactly the same as eq. (21).

Now we're ready to list the backpropagation algorithm in 1 for a single datapoint.

Algorithm 1 Backpropagation	
$a_j^0 = x_j$ for $j = 1, \dots, p$	▷ Initialize input
Feed-forward	
for $l = 1, 2, \dots, L - 1$ do	
for $j = 1, 2, \dots, n$ do	
$a_j^l \leftarrow \sigma \left(\sum_k W_{jk} a_k^{l-1} + b_j^l \right)$	
for $j = 1, 2, \dots, m$ do	
$a_j^L \leftarrow \sigma \left(\sum_k W_{jk} a_k^{L-1} + b_j^L \right)$	
▷ m outputs	
Backward pass	
for $j = 1, 2, \dots, m$ do	
$\Delta_j^L \leftarrow a_j^L - y_j$	
$\partial E / \partial b_j^L \leftarrow \Delta_j^L$	
$\partial E / \partial W_{jk}^L \leftarrow \Delta_j^L a_k^{L-1}$	
▷ m outputs	
for $l = L - 1, \dots, 1$ do	
for $j = 1, \dots, n$ do	
$\Delta_j^l \leftarrow \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1} \right) \sigma'(z_j^l)$	
$\partial E / \partial b_j^l \leftarrow \Delta_j^l$	
$\partial E / \partial W_{jk}^l \leftarrow \Delta_j^l a_k^{l-1}$	

4. Adding L_2 regularization

A common strategy to increase the neural nets' ability to generalize is add L_2 -regularization. This can be achieved by defining the new cost-function

$$E' = E + \frac{\lambda}{2} \sum_{ij} (W_{ij}^l)^2, \quad (25)$$

where E is the un-regularized cost-function and λ is the regularization parameter. This simply adds an additional term to the update rule for the weights:

$$\frac{\partial E}{\partial W_{jk}^l} = \Delta_j^l a_k^{l-1} + \lambda W_{jk}^l. \quad (26)$$

5. Adding momentum

Adding momentum to the update rules of the neural net as described by eq. (4). This adds a new hyperparameter γ to the model, which modifies the update rules for the weights of the neural net as

$$\begin{aligned} V_{ij,t}^l &= \gamma V_{ij,t-1}^l + \eta \frac{\partial E}{\partial W_{jk}^l} \\ W_{jk}^l &\leftarrow W_{jk}^l - V_{ij,t}^l \end{aligned} \quad (27)$$

and similarly for the biases

$$\begin{aligned} v_{j,t}^l &= \gamma v_{j,t-1}^l + \eta \frac{\partial E}{\partial b_j^l} \\ b_j^l &\leftarrow b_j^l - v_{j,t}^l \end{aligned} \quad (28)$$

where V_{ij}^l and v_j^l are introduced to store the momentum of the weights and biases, respectively. The parameter t keeps track of which iteration the algorithm is at. The \leftarrow indicates assignment of what's on the right-hand side of the arrow to the variable on the left-hand side.

6. Hidden activation functions

Hidden activation functions play an important role when it comes to the neural nets' ability to learn. The Sigmoid function was a common choice for a long time, but suffers from the fact that $\sigma'(z) \approx 0$ for large z . Since $\sigma'(z)$ plays an important role in the backpropagation algorithm (see eq. (17)), this may cause a problem famously known as the *vanishing gradient* problem. Two commonly introduced activation functions can remedy this problem. The first one is the so-called *rectified linear unit* (ReLU) which is given by

$$\sigma(z) = (z)^+ = \max(0, z). \quad (29)$$

The second activation function we'll study the effects of is known as the leaky ReLU given by

$$\sigma(z) = \begin{cases} z & \text{if } z > 0, \\ 0.1z & \text{else} \end{cases}. \quad (30)$$

7. The complete neural net algorithm

The backpropagation algorithm with mini-batch size B , L_2 -regularization and momentum is as shown in algorithm 2.

III. RESULTS

A. Logistic Regression

To obtain an optimal value for learning rate η and momentum γ we ran trained our Linear Regression model using various values of the former mentioned parameters and tested it on a validation set. The resulting heatmap is presented in figure 1. We then used the optimal values of $\eta = 10^{-1.5}$ and $\gamma = 0.1$ to investigate an optimal value of regularization parameter strength λ . This was done

by training our Logistic Regression model using various values of λ and the optimal values obtained from the heatmap in figure 1. When tested on a validation set this resulted in accuracy as a function of λ and the resulting

Algorithm 2 Backpropagation of a mini-batch with L_2 -regularization and momentum γ

```

for  $b = 1, 2, \dots, B$  do                                ▷ Loop over the mini-batch
   $a_j^0 = x_j$  for  $j = 1, \dots, p$                         ▷ Initialize input
  Feed-forward
  for  $l = 1, 2, \dots, L - 1$  do
    for  $j = 1, 2, \dots, n$  do
       $a_j^l \leftarrow \sigma(\sum_k W_{jk} a_k^{l-1} + b_j^l)$ 
    for  $j = 1, 2, \dots, m$  do                                ▷  $m$  outputs
       $a_j^L \leftarrow \sigma(\sum_k W_{jk} a_k^{L-1} + b_j^L)$ 
    Backward pass
    for  $j = 1, 2, \dots, m$  do                                ▷  $m$  outputs
       $\Delta_j^L \leftarrow a_j^L - y_j$ 
       $\partial E / \partial b_j^L \leftarrow \partial E / \partial b_j^L + \Delta_j^L$ 
       $\partial E / \partial W_{jk}^L \leftarrow \partial E / \partial W_{jk}^L + \Delta_j^L a_k^{L-1}$ 
    for  $l = L - 1, \dots, 1$  do
      for  $j = 1, \dots, n$  do
         $\Delta_j^l \leftarrow (\sum_k \Delta_k^{l+1} w_{kj}^{l+1}) \sigma'(z_j^l)$ 
         $\partial E / \partial b_j^l \leftarrow \partial E / \partial b_j^l + \Delta_j^l$ 
         $\partial E / \partial W_{jk}^l \leftarrow \partial E / \partial W_{jk}^l + \Delta_j^l a_k^{l-1}$ 
  Update weights and biases
  for  $l = 1, \dots, L$  do
    for  $j = 1, \dots, N$  do
      for  $k = 1, \dots, M$  do
         $V_{ij,t}^l \leftarrow \gamma V_{ij,t-1}^l + \eta (\partial E / \partial W_{jk}^l / B + \lambda W_{jk}^l)$ 
         $W_{jk}^l \leftarrow W_{jk}^l - V_{ij,t}^l$ 
       $v_{j,t}^l \leftarrow \gamma v_{j,t-1}^l + (\eta / B) \partial E / \partial b_j^l$ 
       $b_j^l \leftarrow b_j^l - v_{j,t}^l$ 

```

values are presented in table I which can be found in the Appendix. The table shows us that the optimal value of $\lambda = 10^{-8}$.

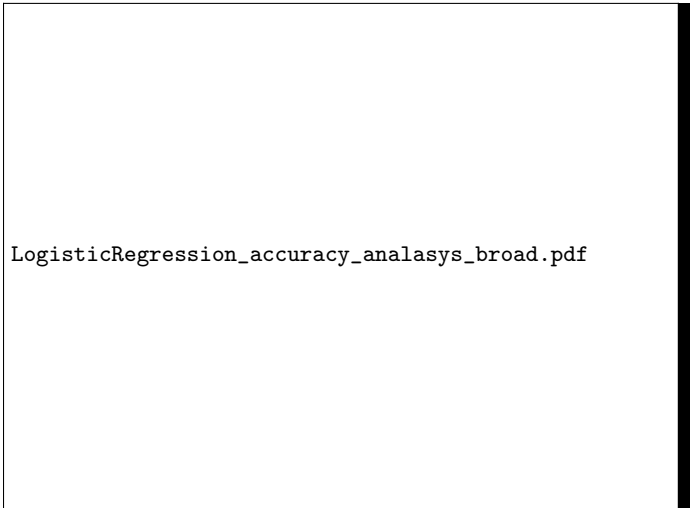
Using the obtained optimal values of η , γ and λ we train our Logistic Regression model and test it on a test set. This resulted in an accuracy of 92.28%.

B. Neural Network

IV. DISCUSSION


høy lambda = Nice plot av vektor lav lambda = Høy accuracy Aka OVERFITTING, det blir veldig strenge krav på hvordan tallene skal se ut.

Skulle gjerne ha optimalisert over alle params samtidig, men det hadde tatt 1000 timer for 10 verdier av hver parameter så noooooooooooooo.



LogisticRegression_accuracy_analasis_broad.pdf

FIG. 1. Heatmap displaying accuracy as a function of both learning rate η and momentum γ when our Logistic Regression model is tested on a validation set. We observe that we achieve highest accuracy = 92.16% when using $\eta = 10^{-1.5}$ and $\gamma = 0.1$.



figures/grid_search_learningrate_datasz.pdf

FIG. 2. The figure shows the accuracy achieved by the neural network on the MNIST dataset for as a function of training data size and learning rates. The accuracy was computed on $N = 10000$ test examples. The parameters used were $B = 10$ (batch size), 1 hidden layer, 30 neurons and 30 epochs. No regularization or momentum was added. The hidden activation function was the Sigmoid function and the top layer activation was the Softmax function.

V. APPENDIX

A. Convexity

There are several definitions for a convex function. To prove that the cost functions E_{OLS} and E_{Ridge} are two convex functions, we rely on the following definition: A function f is convex if its Hessian is positive semi-definite [2].

figures/grid_neurons_epochs.pdf

FIG. 3. The figure shows the accuracy achieved by the neural network on the MNIST dataset for as a function of hidden neurons and epochs. The accuracy was computed on $N_{\text{validation}} = 3000$ validation examples. The parameters used were $B = 10$ (batch size), 1 hidden layer, $\eta = 0.1$ (learning rate), $N_{\text{train}} = 57000$. No regularization or momentum was added. The hidden activation function was the Sigmoid function and the top layer activation was the Softmax function.

figures/grid_search_regularization_lambda.pdf

FIG. 4. The figure shows the accuracy of the neural net as a function of momentum γ and regularization strength λ computed on $N_{\text{validation}} = 3000$ validation examples. The parameters used was $B = 10$, 100 neurons, 1 hidden layer, $\eta = 0.1$, $N_{\text{train}} = 57000$. The hidden activation function was the Sigmoid function and the top layer activation was the Softmax function.

A symmetric matrix A is said to be positive semi-definite if the relation

$$u^T A u \geq 0, \quad (31)$$

is fulfilled for all $u \neq 0$. The Hessians of the cost functions in question are respectively

$$\nabla^2 E_{\text{OLS}} = \mathbf{X}^T \mathbf{X}, \quad \nabla^2 E_{\text{Ridge}} = \mathbf{X}^T \mathbf{X} + \lambda I. \quad (32)$$

Inserting these expressions into the inequality in eq. (31) yields

$$u^T (\nabla^2 E_{\text{OLS}}) u = u^T X^T X u = \|Xu\|^2 \geq 0, \quad (33)$$

$$u^T (\nabla^2 E_{\text{Ridge}}) u = u^T (X^T X + \lambda) u = \|Xu\|^2 + \lambda \|u\|^2 \geq \lambda \|u\|^2 > 0. \quad (34)$$

Given that the penalty parameter $\lambda > 0$, both of the expressions fulfils eq. (31), hence the cost functions are indeed convex.

B. Gradient of Cross-entropy

Suppose the output of our model $\hat{y} \in \{0, 1\}^M$ is made up of M classes. Then the cross-entropy for a single datapoint (x, y) with $x \in \mathbb{R}^p$ and $y \in \{0, 1\}^M$, is given by

$$E = - \sum_{m=1}^M y_m \log \hat{y}_m + (1 - y_m) \log(1 - \hat{y}_m). \quad (35)$$

Let $\hat{y}_m = \sigma(z_m) \equiv \sigma_m$, where $\sigma(z)$ is assumed to obey $\partial \sigma_m / \partial z_j = \sigma_m(1 - \sigma_m) \delta_{mj}$. Let $z_m = \sum_i W_{mi} x_i + b_m$, for some weight matrix W and bias vector b . Then the gradients for a single data point are

$$\begin{aligned} \frac{\partial E}{\partial W_{jk}} &= \frac{\partial E}{\partial \hat{y}_m} \frac{\partial \hat{y}_m}{\partial z_j} \frac{\partial z_j}{\partial W_{jk}} = \frac{\partial E}{\partial \sigma_m} \frac{\partial \sigma_m}{\partial z_j} \frac{\partial z_j}{\partial W_{jk}} \\ &= - \sum_{m=1}^M \left[\frac{y_m}{\sigma_m} - \frac{1 - y_m}{1 - \sigma_m} \right] \sigma_m(1 - \sigma_m) \delta_{jm} \delta_{ki} x_i \\ &= (\sigma_j - y_j) x_k. \end{aligned} \quad (36)$$

Similarly, for the bias term, we get

$$\frac{\partial E}{\partial b_j} = \sigma_j - y_j. \quad (37)$$

C. Logistic Regression

$\log_{10}(\lambda)$	Accuracy
-9	0.918
-8	0.9196
-7	0.9146
-6	0.9154
-5	0.9176
-4	0.9164
-3	0.9184
-2	0.9162
-1	0.8986

TABLE I. The table displays the accuracy achieved on a validation set using various values of regularizer strength λ . We observe that the highest accuracy = 91.96% is achieved when using $\lambda = 10^{-8}$.

[1] René Ask, Kaspára Gåsvær, and Maria Horgen. Regression and resampling techniques applied to franke's function and terrain data. https://github.com/reneaas/fys-stk4155/blob/master/project1/report/Linear_Regression.pdf, 2020.

- [2] Geir Dahl and Michael Floater. Optimization. <https://www.uio.no/studier/emner/matnat/math/MAT3110/h20/pensumliste/lecture12.pdf>, 2020.
- [3] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly, 2019.
- [4] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre Day, Clint Richardson, Charles Fisher, and David Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:16, 03 2018.