University of Oslo

FYS-STK4155

Applied Data Analysis and Machine Learning

Project 2: Classification and Regression, from linear and logistic regression to neural networks

Author:
Håkon Fossheim
November 19, 2022

In this project, we evaluate the performance of a fully connected feedforward neural network (FFNN) in performing regression on the Franke function as well as classification of the Wisconsin Breast Cancer Data Set. The results are then compared with linear regression and logistic regression, respectively.

For regression, the optimal FFNN model produced a mean squared error of 0.012, identical to that of linear regression. However, linear regression seems to model the characteristics of the Franke function surface better.

The FFNN seems more suited for classification, producing an accuracy of 0.931, which slightly exceeds the accuracy score given by our logistic regression at 0.929.

I. Introduction

Neural networks are pattern recognition systems inspired by how biological neurons interconnect and transmit signals.

Their usefulness stems from being able to learn unspecified[1] patterns from the data and make predictions on new samples based on which patterns they activate. This makes them very versatile problem solvers and that can perform tasks which would otherwise be practically impossible to hard code, such as image classification, self-driving and becoming the world champion in Go.

In this project, we will build our own neural network from scratch and gauge its performance on the regression task of fitting to the Franke function as well as performing classification on the Wisconsin Breast Cancer data set. The performance on the regression problem will be compared with linear regression and logistic regression will be used for comparison in classification.

Because neural networks cannot be optimized analytically, numerical solvers must be used in training them. In our case, we used stochastic gradient descent without momentum.

We also study various numerical optimization methods applied to fitting a 4th order polynomial to noisy data and compare their performance. We try to cover the theory behind the methods implemented in the project in the theory and appendix sections, and refer to project 1[1] for a discussion on linear regression and cross-validation.

[1] by the coder.

II. Theory

A. Binary logistic regression

Logistic regression aims to model classification problems. In our case, we will be dealing with two categories, making it a binary logistic problem.

We use a continuous function as our model for the classification problem. In our case, we use the Sigmoid function, given by

$$S(t) = \frac{e^t}{1+e^t} \tag{1}$$

To represent our estimate of the probability that an arbitrary x belongs to the category $y = 1$.

$$p(y=1|x,\boldsymbol{\beta}) = \frac{e^{\beta_0+\beta_1 x}}{1+e^{\beta_0+\beta_1 x}} \tag{2}$$

This is sufficient, since the probability of x belonging to category $y = 0$ is simply given by $1 - p(y = 1|x, \boldsymbol{\beta})$.

So our goal, is to somehow determine the optimal values of $\boldsymbol{\beta}$.

Unlike linear regression, the criteria for these regression coefficients aren't how well the resulting model fits the data points. This is because the points represent draws from an underlying probability distribution (the prior), and it is this PDF that we are trying to model, rather than the outcomes of these draws.

This is done by finding a model[2] that gives highest likelihood of drawing this particular set of $x_i's$ with their corresponding category y_i, i.e. our data set.

This likelihood is given by the product of the probability[3] of each of the outcomes in our experiment, i.e:

$$P(D|\mathbf{x},\boldsymbol{\beta}) = \prod_{i=1}^{n} p(y_i=1|x_i,\boldsymbol{\beta})^{y_i} p(y_i=0|x_i,\boldsymbol{\beta})^{1-y_i} \tag{3}$$

We see that the exponent of $P(y_i = 1)$ and $P(y_i = 0)$ ensures that we only calculate the probability of the observed outcome of the particular draws, x_i.

Since the logarithm is a strictly monotone function, maximizing $P(D|\mathbf{x}, \boldsymbol{\beta})$ is equivalent to maximizing $\log P(D|\mathbf{x}, \boldsymbol{\beta})$. This again is equivalent to minimizing $-\log P(D|\mathbf{x}, \boldsymbol{\beta})$, which we define as our costfunction

$$C(\boldsymbol{\beta}) = -\log P(D|\mathbf{x},\boldsymbol{\beta}) \tag{4}$$

We're after

$$\hat{\boldsymbol{\beta}} = \text{argmin}_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) \tag{5}$$

[2] By tweaking the regression coefficients $\boldsymbol{\beta}$
[3] As estimated by our model

Where

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} y_i \log\{p(y_i=1|x_i,\boldsymbol{\beta})\} + (1-y_i)\log\{p(y_i=0|x_i,\boldsymbol{\beta})\}$$

Is called the cross-entropy.

We know that $p(y_i = 0) = 1 - p(y_i = 1)$, since we only have two possible categories in which a given x_i can land. We can therefore replace $P(y_i = 0)$ with $1 - p$ where $p := p(y_i = 1)$, allowing us to reduce the clutter somewhat:

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} y_i \log\left[p(x_i,\boldsymbol{\beta})\right] + (1-y_i)\log\left[1-p(x_i,\boldsymbol{\beta})\right] \tag{6}$$

Next, we denote $t_i = \beta_0 + \beta_1 x_i$ so that

$$\begin{aligned} p(t_i) &= \frac{e^{t_i}}{1+e^{t_i}} \\ 1-p(t_i) &= \frac{1}{1+e^{t_i}} \end{aligned} \tag{7}$$

Which yields

$$\log\left[p(t_i)\right] = t_i - \log\left[1+e^{t_i}\right] \tag{8}$$

$$\log\left[1-p(t_i)\right] = -\log\left[1+e^{t_i}\right] \tag{9}$$

And thus

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} y_i\left(t_i - \log\left[1+e^{t_i}\right]\right) - (1-y_i)\log\left[1+e^{t_i}\right] \tag{10}$$

Two of these terms cancel, leaving us with the following expression for the cost function

$$\underline{C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} y_i t_i - \log\left[1+e^{t_i}\right]} \tag{11}$$

In appendix A, we show that the gradient and Hessian are given by

$$\underline{\nabla C = \frac{\partial C(\mathbf{x},\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\mathbf{X}^T(\mathbf{y}-\mathbf{p})} \tag{12}$$

$$\underline{\mathbf{H} = \frac{\partial^2 C(\mathbf{x},\boldsymbol{\beta})}{\partial \boldsymbol{\beta}\partial \boldsymbol{\beta}^T} = \mathbf{X}^T\mathbf{W}\mathbf{X}} \tag{13}$$

Where $\mathbf{y}$ and $\mathbf{p}$ are column vectors with elements y_i and p_i, respectively. We also defined the following matrices

$$\underbrace{\begin{bmatrix} p_1(1-p_1) & 0 & \dots & 0 \\ 0 & p_2(1-p_2) & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & p_n(1-p_n) \end{bmatrix}}_{\mathbf{W}}, \quad \underbrace{\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}}_{\mathbf{X}}$$

We see that this Hessian contains an additional diagonal matrix, $\mathbf{W}$, compared with the slightly simpler Hessian we obtained in linear regression: [check][reference report1]

$$\mathbf{H} = \frac{2}{n}\mathbf{X}^T\mathbf{X} \tag{14}$$

B. Optimization

When training a model ($g(\boldsymbol{\beta})$) to fit our data, we invariably employ some method for adjusting the parameters $\boldsymbol{\beta}$ in order to reduce the discrepancy between our model and this data.

This discrepancy is given by the cost function, $C(\boldsymbol{\beta})$ [4]. Its "optimal" functional form depends on the specific task at hand.

The optimal parameters, regardless of the machine learning method, are given when this discrepancy is at its minimum, i.e.

$$\hat{\boldsymbol{\beta}} = \text{argmin}_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) \tag{15}$$

Or, in the case that $C(\boldsymbol{\beta})$ is a convex function and thus contains a single (global) minimum:

$$\boldsymbol{\beta} = \hat{\boldsymbol{\beta}} \Leftrightarrow \frac{\partial C(\hat{\boldsymbol{\beta}})}{\partial \boldsymbol{\beta}} = 0 \tag{16}$$

Where we denote $\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}}|_{\boldsymbol{\beta}=\hat{\boldsymbol{\beta}}}$ as $\frac{\partial C(\hat{\boldsymbol{\beta}})}{\partial \boldsymbol{\beta}}$ for brevity.

When we cannot solve Eq.(15) analytically, we have to resort to numerical optimization methods.

These methods iterately update $\boldsymbol{\beta}$ from an initial guess with the goal of obtaining $\hat{\boldsymbol{\beta}}$;

$$\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n + \eta_n \mathbf{u}_n \tag{17}$$

Here $\mathbf{u}_n$ is the direction of step n and η_n is our step size, also called the learning rate. We can depict this visually as traversing the surface of $C(\boldsymbol{\beta})$ with the goal of reaching the global minimum. The way $\boldsymbol{\beta}$ is update at each step is what defines a given optimization method.

[4] One can formulate many expressions which encapsulate a discrepancy, some better suited than others.

1. Gradient Descent

a. *Finding the optimal step direction*

The direction in which a function decreases the most rapidly is given by the negative of the gradient of the function.

To see this, consider the rate of change in f in the direction of an arbitrary normalized vector, $\hat{\mathbf{u}}$, at the point $\mathbf{x}$. This is called the directional derivative [2] :

$$\frac{\partial}{\partial \alpha} f(\mathbf{x} + \alpha\hat{\mathbf{u}})|_{\alpha=0} = \nabla_{\mathbf{x}} f(\mathbf{x}) \cdot \hat{\mathbf{u}} = ||\nabla_{\mathbf{x}} f(\mathbf{x})||_2 ||\hat{\mathbf{u}}||_2 \cos\theta \tag{18}$$

As seen, the directional derivative is minimized when $\theta = \pi$, which implies that $\mathbf{u}_{opt}$ points in the opposite direction of the gradient;

$$\hat{\mathbf{u}}_{opt} = \frac{-\nabla_{\mathbf{x}} f(\mathbf{x})}{||\nabla_{\mathbf{x}} f(\mathbf{x})||_2} \tag{19}$$

We therefore set the $\mathbf{u}_n$ in Eq.(17) to $\mathbf{u}_n = -\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}_n)$, letting η_n absorb the normalization factor. We thus have

$$\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n - \eta_n \nabla C(\boldsymbol{\beta}_n) \tag{20}$$

b. *Finding the optimal step size*

[finding the optimal step size is not as straight-forward. A closed form solution may not be available, but we can (with certain caveats) estimate it by use of a second order Taylor expansion. For convex functions, this yields an analytic solution for the optimal step size, η_n^{opt}.] Similarly, we are also interested in finding the step size that at any iteration n will yield the greatest decrease in the cost function, i.e. we want to minimize $C(\boldsymbol{\beta}_{n+1}) - C(\boldsymbol{\beta}_n)$ with respect to η_n. We do this by performing a second order Taylor expansion around $\boldsymbol{\beta}_n$ and evaluating it at $\boldsymbol{\beta}_{n+1}$:

$$\begin{aligned} C(\boldsymbol{\beta}_{n+1}) \approx C(\boldsymbol{\beta}_n) + (\boldsymbol{\beta}_{n+1} - \boldsymbol{\beta}_n)^T \mathbf{g}_n \\ + \frac{1}{2}(\boldsymbol{\beta}_{n+1} - \boldsymbol{\beta}_n)^T \mathbf{H}_n (\boldsymbol{\beta}_{n+1} - \boldsymbol{\beta}_n) \end{aligned} \tag{21}$$

Here we denoted $\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta})|_{\boldsymbol{\beta}=\boldsymbol{\beta}_n}$ as $\mathbf{g}_n$ and likewise for the Hessian.

From Eq.(20) we have $\boldsymbol{\beta}_{n+1} - \boldsymbol{\beta}_n = -\eta_n \mathbf{g}_n$. Inserting this into Eq.(21) yields

$$C(\boldsymbol{\beta}_{n+1}) - C(\boldsymbol{\beta}_n) \approx -\eta_n \mathbf{g}_n^T \mathbf{g}_n + \frac{1}{2}\eta_n^2 \mathbf{g}_n^T \mathbf{H}_n \mathbf{g}_n \tag{22}$$

Next, we find the extrema of $\Delta C(\boldsymbol{\beta}_n) \equiv C(\boldsymbol{\beta}_{n+1}) - C(\boldsymbol{\beta}_n)$:

$$\frac{\partial \Delta C(\boldsymbol{\beta}_n)}{\partial \eta_n} \approx -\mathbf{g}_n^T \mathbf{g}_n + \eta_n \mathbf{g}_n^T \mathbf{H}_n \mathbf{g}_n = 0 \tag{23}$$

$$\Rightarrow \eta_n^{ext} \approx \left. \frac{\mathbf{g}^T\mathbf{g}}{\mathbf{g}^T\mathbf{H}\mathbf{g}} \right|_{\boldsymbol{\beta}=\boldsymbol{\beta}_n} \tag{24}$$

To determine whether this is a minimum, maximum or saddle point, we take the second derivative of Eq.(23):

$$\frac{\partial^2 \Delta C\left(\beta_n\right)}{\partial \eta_n^2} \approx \mathbf{g}_n^T \mathbf{H}_n \mathbf{g}_n \tag{25}$$

[in the case of functions with high curvature, this yields suboptimal results....]

We see that if $\mathbf{H}$ is positive definite, we are guaranteed that the η_n given by Eq.(24) corresponds to a (global?) minimum of $\Delta C\left(\boldsymbol{\beta}_n\right)$. When $C(\boldsymbol{\beta})$ is convex, this will always be the case. Furthermore, Eq(22) will be an exact equation rather than an approximation in this case, and so the we get an exact closed form solution to the optimal step size:

$$\eta_n^{opt} = \left.\frac{\mathbf{g}^T\mathbf{g}}{\mathbf{g}^T\mathbf{H}\mathbf{g}}\right|_{\boldsymbol{\beta}=\boldsymbol{\beta}_n} \tag{26}$$

However, the Hessian needs to be evaluated at each step. This can get computationally expensive, especially for methods that need to compute its inverse, such as the Newton Raphson method discussed in section B of the appendix.

One often therefore either settle for a constant learning rate η, so that

$$\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n - \eta \nabla C(\boldsymbol{\beta}_n) \tag{27}$$

Or one cooks up some alternative η_n which does not use the Hessian.

c. ***Limitiations of gradient descent***

Although conceptually elegant, the gradient descent suffers from some major drawbacks, some of which are:

- Depending on the initial position, it might get stuck in a local minimum. This becomes increasingly likely when dealing with higher dimensional cost functions, such as those of neural networks.
- It might wander into a saddle point, taking exponential time to escape.[4]
- For a fixed learning rate, we end up taking steps of ever decreasing size when approaching the minimum, as seen in Eq.(27). This can be unnecessarily computationally expensive, depending on how the tolerance is set.

C. Stochastic Gradient Descent

In machine learning, it is typically favorable to train your model on as large of a data set as possible, thereby equipping it to handle a greater variety of data that it might encounter in the real world.

This large data set does however come at the price of a more computationally expensive cost function;

$$C(\mathbf{x}, \mathbf{y}) = \frac{1}{n}\sum_{i=0}^{n-1} \underbrace{c(x_i, y_i)}_{c_i} \tag{28}$$

And therefore also a more costly gradient:

$$\mathbf{g}_{\text{GD}} = \nabla_{\boldsymbol{\beta}} C = \frac{1}{n}\nabla_{\boldsymbol{\beta}} \sum_{i=0}^{n-1} c_i = \frac{1}{n}\sum_{i=0}^{n-1} \nabla_{\boldsymbol{\beta}} c_i = \frac{1}{n}\sum_{i=0}^{n-1} \mathbf{g}_i \tag{29}$$

Where

$$\mathbf{g}_i = \sum_{k=0}^{p-1} \frac{\partial c_i}{\partial \beta_k} \hat{e}_k \tag{30}$$

With $\hat{e}_k$ being the k-th unit vector.

For a data set of size n and a model with p parameters, we therefore need to evaluate $p \times n$[check] partial derivatives for each step of our gradient descent algorithm.

Stochastic gradient descent (SGD) eases this computational burden by replacing n with M where $1 \leq M \leq n$.

At each step, it only computes the cost function on a random subset of size M of our full data sample, $\mathcal{D} = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$. This subset is referred to as a ***mini-batch*** and we will denote the mini-batch used in the k-th step as $\mathcal{D}_k$.

And so rather than comparing our model with the entire data set[5] and reducing their discrepancy by moving along $-\nabla C$, SGD compares only a subset of the data with the same subset of the model and reduces the cost function only according to the discrepancy of this subset, somewhat like repeatedly polling random groups of people and adjusting national policies each time, rather than performing elections.

But are we guaranteed that this will work? Not necessarily, but as long as we pick the data samples of our mini-batches, $\mathcal{D}_k$ in a specific way, we are guaranteed that if we repeat the SGD procedure N times, and compute the average of the SGD gradient at step k, then this average will approach the GD gradient at this step[6] as N approaches infinity, i.e.

$$\mathbb{E}\left[\widehat{\mathbf{g}_{SGD}^{(k)}}\right] := \lim_{N\to\infty} \frac{1}{N}\sum_{run=1}^{N} \mathbf{g}_{\text{SGD}}^{(k)}(run) = \mathbf{g}_{\text{GD}}^{(k)} \tag{31}$$

---

[5] Specifically the response variable, $\mathbf{y}$

[6] Since k is arbitrary, this holds for all the steps.

a. ***proof***

We can think of $\mathcal{G}_{GD}(\mathcal{D}) = \left\{\mathbf{g}_0^{(k)}, \mathbf{g}_1^{(k)}, ...\mathbf{g}_{n-1}^{(k)}\right\}$ as our population from which we draw batch k. The gradient calculated by the ordinary gradient descent algorithm is then the expected value of the population, $\mu = \mathbf{g}_{GD}^{(k)}$.

The central limit theorem tells us that if we draw M i.i.d. samples from this population, then their average,

$$z = \mathbf{g}_{\text{SGD}}^{(k)} = \frac{1}{M}\sum_{j\in\mathcal{D}_k}\mathbf{g}_j \tag{32}$$

Will be distributed as a normal distribution with expectation equal to that of the population, μ [7]. We therefore have have

$$\mathbb{E}\left[z\right] = \lim_{N\to\infty}\frac{1}{N}\sum_{run=1}^{N}\mathbf{g}_{\text{SGD}}^{(k)}(run) = \mathbf{g}_{\text{GD}}^{(k)} = \mu \tag{33}$$

And so the estimator corresponding to the SGD gradient at step k is unbiased with respect to that calculated in ordinary gradient descent.

As mentioned, we need the $\mathbf{g}_j$'s to be independent and identically distributed for this to hold. This is guaranteed if we perform the draws in the following manner:

- Draw the samples *with replacement*

If we don't draw with replacement, then the $\mathbf{g}_j$'s won't be independent. This can be readily seen in the case where $n = 2$ and $M = 2$. Drawing without replacement means the second draw is determined by the first, whilst this is not the case when doing replacement.

- Draw the samples *randomly* from a uniform distribution

If we don't draw randomly, then our draws won't be identically distributed[8]. If we were to e.g. divide the data set $\mathcal{D}$ sequentially into m groups and use these groups as our mini-batches, then the $\mathbf{g}_j$'s of different mini-batches will be distributed over different domains and therefore not identically.

Note that for mathematical brevity we constructed the mini-batches by picking M gradients, $\mathbf{g}_j$, from $\mathcal{G}$ and summing these together. Due to the linearity of the nabla operator, this is equivalent to first picking M data points from $\mathcal{D}$, and then calculating the gradient on the cost function of these data points.

[7] and variance equal to the population variance divided by M

[8] Unless repeatedly using the same subset of $\mathcal{G}$, which is equivalent to just performing regular gradient descent on a reduced version of your data.

b. ***Further benefits of SGD over GD***

In addition to being computationally faster, SGD has a few added benefits over gradient descent. For one, the random nature of the steps makes it harder to get stuck in a local minimum.

SGD also has a tendancy to "oscillate" around the minimum, not converging in the deterministic manner that ordinary gradient descent does. This is actually often beneficial, since it helps prevent overfitting.

D. SGD for OLS and Ridge regression

In order to perform stochastic gradient descent with these regression methods, we need to know $\nabla_{\boldsymbol{\beta}}C(\mathbf{y},\boldsymbol{\beta})$ in both cases.

In SGD, we compute the cost function on our mini-batch only, and use this to update our parameters. We therefore divide over the number of samples in our mini-batch, M, rather than the total number of data samples, n. In calculating the gradient, we include the regularization term from Ridge regression, since the OLS gradient can be found by simply setting $\lambda = 0$ in the result we derive.

$$\begin{aligned} C(\mathbf{y},\boldsymbol{\beta}) = MSE &= \frac{1}{M}\sum_{i\in\mathcal{D}_k}\left(y_i - \mathbf{X}_{i*}\boldsymbol{\beta}\right)^2 + \lambda\sum_{j=0}^{p-1}\beta_j^2 \\ &= \frac{1}{M}\left(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\right)^T\left(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\right) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \qquad (34) \\ &= \frac{1}{M}\left(\mathbf{y}^T - \boldsymbol{\beta}^T\mathbf{X}^T\right)\left(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\right) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \qquad (35) \end{aligned}$$

$$\Rightarrow MC(\mathbf{y},\boldsymbol{\beta}) = \mathbf{y}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} - \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + M\lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \tag{36}$$

Now, we have that $\nabla_{\boldsymbol{\beta}}C(\mathbf{y},\boldsymbol{\beta}) = \frac{\partial}{\partial\boldsymbol{\beta}}C(\mathbf{y},\boldsymbol{\beta})$. The first term vanishes under differentiation w.r.t. $\boldsymbol{\beta}$. Furthermore, since each term must be a scalar we have that

$$\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} = (\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y})^T = \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} \tag{37}$$

We are then left with

$$MC(\mathbf{y},\boldsymbol{\beta}) = -2\frac{\partial}{\partial\boldsymbol{\beta}}\mathbf{y}^T\mathbf{X}\boldsymbol{\beta} + \frac{\partial}{\partial\boldsymbol{\beta}}\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + M\lambda\frac{\partial}{\partial\boldsymbol{\beta}}\boldsymbol{\beta}^T\boldsymbol{\beta} \tag{38}$$

By use of identity (i) of the appendix on the first and last term and identity (ii) on the second term, we get

$$M\frac{\partial C}{\partial\boldsymbol{\beta}} = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + M\lambda\boldsymbol{\beta} \tag{39}$$

The batch gradient for Ridge regression is thus

$$\underline{\nabla_{\boldsymbol{\beta}}C = \frac{2}{M}\mathbf{X}^T\left(\mathbf{X}\boldsymbol{\beta}-\mathbf{y}\right) + \lambda\boldsymbol{\beta}} \tag{40}$$

And setting $\lambda = 0$ gives the batch gradient for OLS:

$$\underline{\nabla_{\boldsymbol{\beta}} C = \frac{2}{M}\mathbf{X}^T\left(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\right)} \tag{41}$$

E. Momentum gradient descent

In momentum gradient descent, the previous step is taken into account when calculating the current step. Rather than letting the step size and direction be directly proportional to the gradient, the gradient only updates the previous step through an additive term

$$\mathbf{v}_i = \gamma \mathbf{v}_{i-1} + \eta \nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}_i) \tag{42}$$

$$\boldsymbol{\beta}_{i+1} \leftarrow \boldsymbol{\beta}_i - \mathbf{v}_i \tag{43}$$

This ensures that as we approach the minimum, the step size doesn't vanish too quickly. Analogous to a ball rolling down a curved landscape, if we overshoot the minimum, the gradient term will change sign and increase in magnitude the further we get from the minimum, pulling it back in.

Momentum gradient descent also helps suppress oscillations which can occur in steep and curved landscapes.

F. Neural Networks

Our next model is the neural network. Specifically, we will be considering a type of feedforward neural network called the multilayered perceptron (MLP). We don't know of an analytical solution to this model, and so we will have to perform a numerical optimization method to find the optimal weights, $\hat{\boldsymbol{\theta}}$. To do this, we employ the stochastic gradient descent method introduced in section II C. We therefore need to find an expression for

$$\frac{\partial C(\mathbf{MLP})}{\partial \theta} \tag{44}$$

From which we can construct the gradient and perform gradient descent;

$$\vec{\theta}^{(k+1)} = \vec{\theta}^{(k)} - \eta \nabla_{\vec{\theta}^k} C(\mathbf{y} - \tilde{\mathbf{y}}) \tag{45}$$

$$\begin{bmatrix} \theta_1^{(k+1)} \\ \theta_2^{(k+1)} \\ \vdots \\ \theta_N^{(k+1)} \end{bmatrix} = \begin{bmatrix} \theta_1^{(k)} \\ \theta_2^{(k)} \\ \vdots \\ \theta_N^{(k)} \end{bmatrix} - \eta \begin{bmatrix} \partial C/\partial \theta_1^{(k)} \\ \partial C/\partial \theta_2^{(k)} \\ \vdots \\ \partial C/\partial \theta_N^{(k)} \end{bmatrix} \tag{46}$$

We see that instead of finding $\partial C/\partial \theta^{(k)}$ for every parameter, constructing the gradient and then performing a gradient descent step, we could perform mini steps by updating $\theta_u^{(k)}$ right after finding $\partial C/\partial \theta_u^{(k)}$:

$$\theta_u^{(k+1)} = \theta_u^{(k)} - \partial C/\partial \theta_u^{(k)} \tag{47}$$

Iterating throught the parameters and doing this each time would be equivalent to Eq.(46).

A neural network is a procedure for generating a very high dimensional model, inspired by how neurons fire in the brain. This is done by iteratively applying vector valued functions, $\mathbf{f}^l$, where l is called the network's *layer*, to the data, $\mathbf{x}$. The elements of $\mathbf{f}^l$ are called the *neurons/nodes/units* of the network. Now, each of these iterations "couple" the outputs of the previous function, $\mathbf{f}^{(l-1)}$ with an iteration specific set of parameters, $\Theta^{(l)}$. The way in which they are coupled, is given by $z(\Theta^l, \mathbf{f}^{l-1})$. This function couples the output of the previous layer with the parameters of the current layer and compresses this by adding them up to yield a scalar which is passed to the activation function.

The output of layer l is then given by

$$\mathbf{f}^l = \begin{bmatrix} f\Big(z\big(\Theta_{1*}^{(l)}, \mathbf{f}^{(l-1)}\big)\Big) \\ f\Big(z\big(\Theta_{2*}^{(l)}, \mathbf{f}^{(l-1)}\big)\Big) \\ \vdots \\ f\Big(z\big(\Theta_{j*}^{(l)}, \mathbf{f}^{(l-1)}\big)\Big) \\ \vdots \\ f\Big(z\big(\Theta_{n_l*}^{(l)}, \mathbf{f}^{(l-1)}\big)\Big) \end{bmatrix} \tag{48}$$

Where we left it implicit that f and z belong to layer l.

From figure 1 we then have

$$\tilde{\mathbf{y}} = \mathbf{f}^L \left[z\left(\Theta^L, \mathbf{f}^{L-1}\right) \right] \tag{49}$$

This high dimensional model is therefore generated by applying a multi-nested vector function to our data. The number of times this function is nested determines how many *layers* the neural network is said to have. This is what is referred to as *forward propagation*. Differentiating this function with respect to the parameters of the generated model is referred to as *backpropagation*.

backpropagation

Neural networks learn by iteratively adjusting their parameters, reducing the discrepancy between the network output and the training data (i.e. the cost function) after each iteration. This is done until the network performance is deemed satisfactory.

As in linear and logistic regression, a chosen optimization method is used to perform these step-wise parameter adjustments. A widely used method is the stochastic gradient descent from section II C, which we will employ in this project. Now, in order to perform these adjustments, the optimizer needs to know the direction in parameter space which at each step yields the greatest reduction in the cost function, i.e. the (negative) gradient of the neural network.

Calculating this gradient/derivative is not as straightforward as for linear- or logistic regression, and is referred to as *backpropagation*.

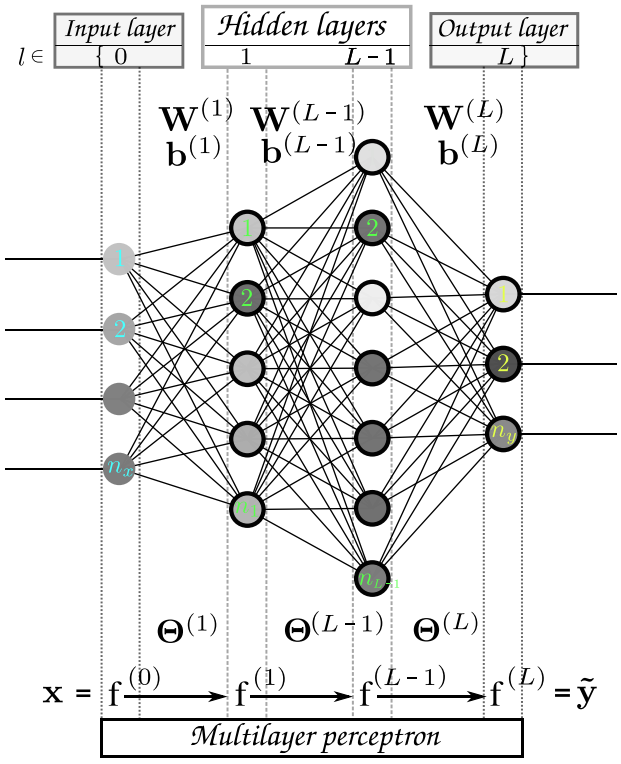

FIG. 1. Here we see a depiction of a multilayer perceptron with 2 hidden layers. The activation values of the neurons are represented by their brightness. The weights of the neural network are depicted as lines, with each line corresponding to an element of the weight matrix $\mathbf{W}^{(l)}$. A visual depiction of the biases, $\mathbf{b}^{(l)}$, has not been implemented. Furthermore, layers involved in forward propagation are coloured blue, whilst those involved in backpropagation are coloured yellow. Their overlap produces a green color which indicates the layers which feed both forwards and backwards.

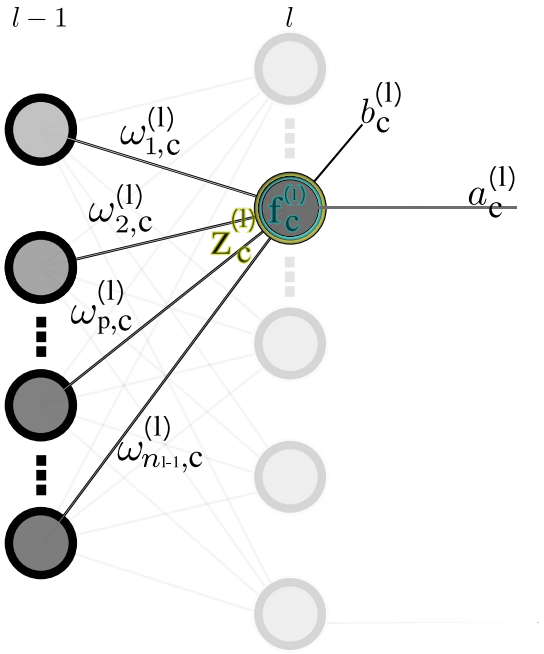

FIG. 2. Here we see what's called the perceptron model. The perceptron can itself be used for interpolation, but we will study it to only as an element of the multilayered perceptron, thereby giving us a closer look at what is happening to each neuron of the MLP. As seen, the output neuron is connected to another set of neurons, who's output values get coupled with a set of weights before they, along with a bias, get passed to z^l_j. This function outputs a scalar value, which is passed to the activation function of this particular neuron, f^l_j. This function is in turn used to produce the output of this single neuron, namely the activation value a^l_j. This is called the perceptron model, since it can alone be used as a model for interpolation.

In backpropagation, we only need to find an expression for the derivatives belonging to the outermost layer, $l = L$. Having found a general expression for the derivatives of layer l as a function of layer $l+1$, one can then propagate backwards through the neural network, finding the derivatives of the current layer with the help of those of the previous layers.

Throughout our derivation, we will use c as the index variable of the current layer under consideration, p as the index variable of the previous layer and s as the index variable of the succeeding layer.

Output layer, $l = L$

$$\frac{\partial C}{\partial \omega^L_{c,p}} = \underbrace{\frac{\partial C}{\partial a^L_c}\frac{\partial a^L_c}{\partial z^L_c}}_{\delta^L_c}\frac{\partial z^L_c}{\partial \omega^L_{c,p}} \tag{50}$$

$$\frac{\partial C}{\partial b^L_c} = \underbrace{\frac{\partial C}{\partial a^L_c}\frac{\partial a^L_c}{\partial z^L_c}}_{\delta^L_c}\frac{\partial z^L_c}{\partial b^L_c} \tag{51}$$

The choice of cost function and activation function in the final layer depends on the problem at hand.

For binary classification, one would typically use the sigmoid as the activation function in the final layer and the MSE as cost function.

For multi-class classification one would employ the softmax at the final layer and use the cross-entropy as cost function.

For regression problems, one avoids the use of activation functions in the final layer, and take the activation values as output. This allows the output to be unbounded and take on values from $-\infty$ to ∞. The mean squared error is typically used as the cost function in this case.

Regardless of the problem, z^L_c is typically given by

$$z^L_c = \sum_{p=0}^{n_{L-1}} w^L_{p,c} a^{L-1}_p + b^L_c \tag{52}$$

And so the last terms in Eq.(54) and Eq.(51) are

$$\frac{\partial z_C^L}{\partial \omega_{p,c}^L} = a_p^{L-1}, \qquad \frac{\partial z_c^L}{\partial b_c^L} = 1 \tag{53}$$

And so

$$\frac{\partial C}{\partial \omega_{c,p}^L} = \delta_c^L a_p^{L-1}, \qquad \frac{\partial C}{\partial b_c^L} = \delta_c^L \tag{54}$$

The weights and biases of the final layer are thus updated as follows:

$$\underline{\omega_{p,c}^L \leftarrow \omega_{p,c}^L - \eta \delta_c^l a_p^{L-1}} \tag{55}$$

$$\underline{b_c^L \leftarrow b_c^L - \eta \delta_c^L} \tag{56}$$

layer l as a function of layer $l+1$

Next, we want to find an expression for the derivative of the cost function w.r.t. the weights and biases of layer l in terms of the already acquired expressions from layer $l+1$, viz. the δ_s^{l+1}'s.

In finding how a single weight or bias from layer l affects the cost-function, we have to sum over all the paths the signal takes in going from a weight or bias in layer l and through layer $l+1$ to the cost-function:

$$\frac{\partial C}{\partial \omega_{p,c}^l} = \frac{\partial z_c^l}{\partial \omega_{p,c}^l} \underbrace{\frac{\partial a_c^l}{\partial z_c^l}}_{f'(z_c^l)} \sum_s \frac{\partial z_s^{l+1}}{\partial a_c^l} \underbrace{\frac{\partial a_s^{l+1}}{\partial z_s^{l+1}} \frac{\partial C}{\partial a_s^{l+1}}}_{\delta_s^{l+1}} \tag{57}$$

Using that

$$z_s^{l+1} = \sum_c \omega_{c,s}^{l+1} a_c^l + b_s^{l+1} \tag{58}$$

We get

$$\frac{\partial z_s^{l+1}}{\partial a_c^l} = \omega_{c,s}^{l+1} \tag{59}$$

Furthermore,

$$z_c^l = \sum_p \omega_{p,c}^l a_p^{l-1} + b_c^l \tag{60}$$

and so

$$\frac{\partial z_c^l}{\partial \omega_{p,c}^l} = a_p^{l-1} \tag{61}$$

and thus

$$\frac{\partial C}{\partial \omega_{p,c}^l} = a_p^{l-1} \underbrace{f'(z_c^l) \sum_s \omega_{c,s}^{l+1} \delta_s^{l+1}}_{\delta_c^l} = \delta_c^l a_p^{l-1} \tag{62}$$

$$\frac{\partial C}{\partial b_c^l} = f'(z_c^l) \sum_s \omega_{c,s}^{l+1} \delta_s^{l+1} = \delta_c^l \tag{63}$$

As mentioned, we can perform gradient descent by updating the weights and biases individually rather than constructing the entire gradient. The weights and biases are can thus updated as follows

$$\boxed{\omega_{p,c}^l \leftarrow \omega_{p,c}^l - \eta \delta_c^l a_p^{l-1}} \tag{64}$$

$$\boxed{b_c^l \leftarrow b_c^l - \eta \delta_c^l} \tag{65}$$

with

$$\boxed{\delta_c^l = f'(z_c^l) \sum_s \omega_{c,s}^{l+1} \delta_s^{l+1}} \tag{66}$$

if $l < L$ and

$$\boxed{\delta_c^l = f'(z_c^L) \frac{\partial C}{\partial a_c^L}} \tag{67}$$

if $l = L$.

Using matrix notation

Forward propagation We would like to be able to feed the network a design matrix $\mathbf{X}$ of shape $(n_{inputs}, n_{features})$ rather than having to feed the data samples one by one in the form of $1 \times n_{features}$ vectors.

We can do this in the following manner:

$$\mathbf{A}^l = \mathbf{A}^{l-1}\mathbf{W}^l + \mathbf{B}^l \tag{68}$$

The activation values of layer l are now stored in the matrix $\mathbf{A}^l$, with columns representing different neurons and rows corresponding to different inputs, i.e. rows of $\mathbf{X}$. Here $\mathbf{B}^l$ is given by

$$\mathbf{B}^l_{n_{inputs} \times n_l} = \begin{bmatrix} b_1^l & b_2^l & \dots & b_{n_l}^l \\ b_1^l & b_2^l & \dots & b_{n_l}^l \\ \vdots & \vdots & \vdots & \vdots \\ b_1^l & b_2^l & \dots & b_{n_l}^l \end{bmatrix} \tag{69}$$

The output, $\mathbf{A}_L^{n_L \times n_{input}}$, will then have columns containing the network prediction for the corresponding column in $\mathbf{X}$.

Backpropagation As seen, the weights and biases of the network are updated in the following manner:

$$\omega_{p,c}^l \leftarrow \omega_{p,c}^l - \eta \delta_c^l a_p^{l-1} \tag{70}$$

$$b_c^l \leftarrow b_c^l - \eta \delta_c^l \tag{71}$$

We can update all the weights and biases in a layer at once. Here this is done using the errors from one forward pass. Note that both $\vec{a}$ and $\vec{\delta}$ are row vectors with entries corresponding to the different neurons of that layer.

$$\mathbf{W}^l \leftarrow \mathbf{W}^l - \eta \left(\vec{a}^{l-1}\right)^T \vec{\delta}^l \tag{72}$$

$$\vec{b}^l \leftarrow \vec{b}^l - \eta \vec{\delta}^l \tag{73}$$

Or we can add the contribution from every data sample together and update the weights and biases in one go

$$\boxed{\mathbf{W}^l \leftarrow \mathbf{W}^l - \eta \left(\mathbf{A}^{l-1}\right)^T \boldsymbol{\delta}^l} \tag{74}$$

$$\boxed{\vec{b}^l \leftarrow \vec{b}^l - \eta \boldsymbol{\delta}^l \cdot I_{n_l,1}} \tag{75}$$

Where

$$\delta_c^l = f'(z_c^l) \sum_s \omega_{c,s}^{l+1} \delta_s^{l+1} = f'(z_c^l) \cdot \vec{\delta}^{l+1} \left(\mathbf{W}^{l+1}\right)_{*c}^T \tag{76}$$

The errors for layer l for an arbitrary data sample is then

$$\delta^l = f'(\vec{z}^l) \circ \vec{\delta}^{l+1} \left(\mathbf{W}^{l+1}\right)^T \tag{77}$$

Given the error matrix

$$\boldsymbol{\delta}^{l+1} = \begin{bmatrix} \vec{\delta}_0^{l+1} \\ \vec{\delta}_1^{l+1} \\ \cdots \\ \vec{\delta}_{n_{inputs}-1}^{l+1} \end{bmatrix} \tag{78}$$

For $l < L$ we then have

$$\boxed{\boldsymbol{\delta}^l = f'(\mathbf{Z}^l) \circ \boldsymbol{\delta}^{l+1} \left(\mathbf{W}^{l+1}\right)^T} \tag{79}$$

While for $l = L$ we get

$$\boxed{\boldsymbol{\delta}^L = f'(\mathbf{Z}^L) \circ \frac{\partial C}{\partial \mathbf{A}^L}} \tag{80}$$

Here we are doing element wise differentiation.

G. Regularization

To prevent overfitting, we can add a regularization term to the cost function

$$\tilde{C}(\boldsymbol{\theta}, \tilde{\mathbf{y}}) = C(\boldsymbol{\theta}, \tilde{\mathbf{y}}) + \lambda \underbrace{\sum_{i,j} \omega_{i,j}^2}_{||\boldsymbol{\omega}||_2^2} \tag{81}$$

We then get

$$\frac{\partial \tilde{C}}{\partial \omega_{p,c}^l} = \frac{\partial C}{\partial \omega_{p,c}^l} + 2\lambda \omega_{p,c}^l \tag{82}$$

We can let λ absorb the factor of 2. The weights are then updated as

$$\omega_{p,c}^l \leftarrow \omega_{p,c}^l - \eta \frac{\partial \tilde{C}}{\partial \omega_{p,c}^l} \tag{83}$$

$$\omega_{p,c}^l \leftarrow \omega_{p,c}^l - \eta \left(\delta_c^l a_p^{l-1} + \lambda \omega_{p,c}^l\right) \tag{84}$$

$$\tag{85}$$

Or in matrix form

$$\mathbf{W}^l \leftarrow \mathbf{W}^l - \eta \left(\left(\mathbf{A}^{l-1}\right)^T \boldsymbol{\delta}^l + \lambda \mathbf{W}^l\right) \tag{86}$$

And similarly for the biases:

$$\vec{b}^l \leftarrow \vec{b}^l - \eta \left(\boldsymbol{\delta}^l \cdot I_{n_l,1} + \lambda \vec{b}^l\right) \tag{87}$$

III. Results

A. Numerical Optimization

We commence by testing various optimization methods on a generating polynomial of the form

$$y = a_0 + a_1 x + a_2 x^2 + \epsilon \tag{88}$$

Where $\epsilon \sim \mathcal{N}(0, 1)$.

In order to depict how gradient descent traverses parameter space with and without momentum, we start by fitting a polynomial with two parameters;

$$\tilde{y} = \beta_0 + \beta_1 x \tag{89}$$

To the following function

$$y = 4 + 3x + \epsilon \tag{90}$$

We perform gradient descent with and without momentum, using a learning rate of $\eta = \frac{1}{\lambda_{max}} = 0.0458$ where λ_{max} is the maximum eigenvalue of the Hessian. We stop the algorithm when a step decreases the cost function by less than 10^{-5}, our tolerance.

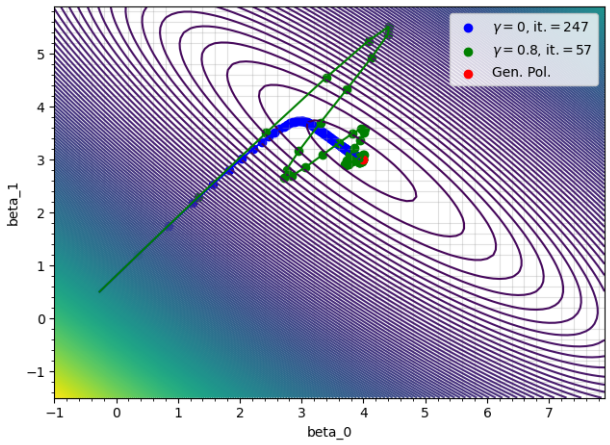

FIG. 3. Here we see the steps taken by GD (blue) and GD with momentum (green). The total number of iterations before tolerance is reached is also shown. Contour lines of the cost function have been added to compare how they traverse this surface.

As seen in Fig.3, GD with momentum reaches the minimum much faster, taking 57 iterations to reach tolerance, compared to the 247 of ordinary GD. As mentioned in II E, GD will take ever smaller steps when approaching the minimum, leading to a slow convergence. Momentum GD remedies this since it bases its next step on the size of the previous step, in addition to the gradient. The downside the added hyperparameter γ, which has to be tuned. If it is too large, it can overshoot the minimum or take too long to settle and if it is too small, we encounter the same problem as in ordinary GD.

Typically we won't know the generating function of our data. Going forward we will therefore be interpolating the 2nd order polynomial

$$y = 3 + 4x + 6x^2 + \epsilon \tag{91}$$

Using a fourth order polynomial.

For each optimization method used, an initial grid search of η and λ was performed without momentum. Using the optimal η and λ values, we then find the γ yielding the lowest MSE through e.g. Fig.5. The optimal η and λ for this γ are then found through another grid search. Finally, we plot the MSE with and without momentum as a function of iterations/epochs, using their respective optimal learning rate and regularization parameter, as seen in Fig.6. The algorithm is stopped when the difference in cost function between iterations/epochs is less than our threshold of 10^{-3}.

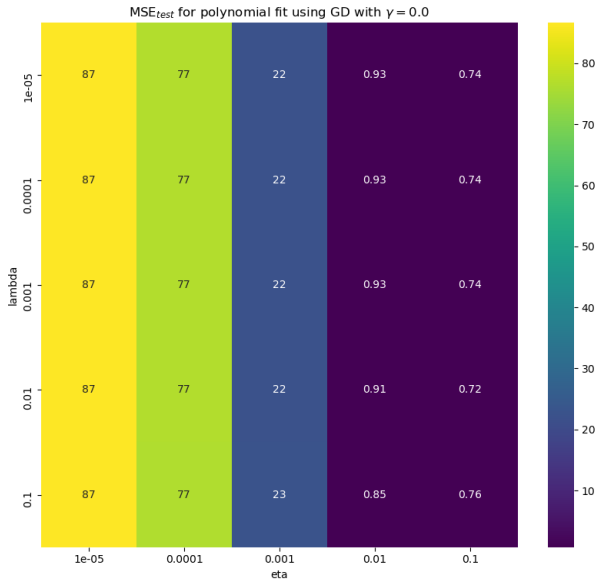

FIG. 4. Caption

In Fig.4 we see a grid search of GD without momentum. As seen, the regularization parameter has little impact on performance. The algorithm is however very quite sensitive to the learning rate.

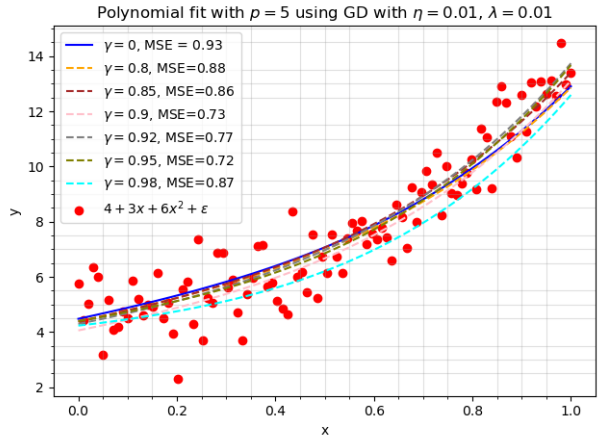

FIG. 5. Polynomial fits to our data using gradient descent for different values of the momentum term. The optimal values $\eta = 0.01, \lambda = 0.01$ from the grid search of $\gamma = 0$ have been used.

As seen in Fig.5, we achieve good fits with gradient descent both with and without momentum.

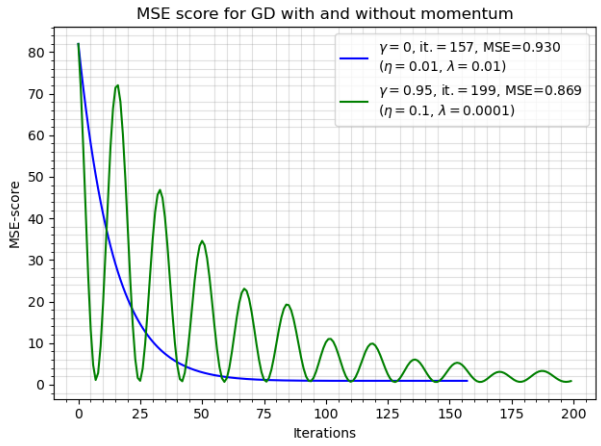

FIG. 6. MSE evaluated at each iteration of gradient descent with and without momentum.

We see in Fig.6 that GD descent reaches tolerance within 157 iterations, whereas oscillations around the minimum prevents momentum GD from converging within 200 iterations.

We perform the same grid search for SGD and find $\eta = 0.001$ and $\lambda = 0.01$ to be the optimal values without momentum. An adaptive learning rate of the form

$$\eta_i = \frac{t_1}{(\eta_{i-1} * m + i) + t_0} \tag{92}$$

was used for tuning the step sizes. We found $t_0 = 0.05$, $t_1 = 0.5$ to produce the best results.

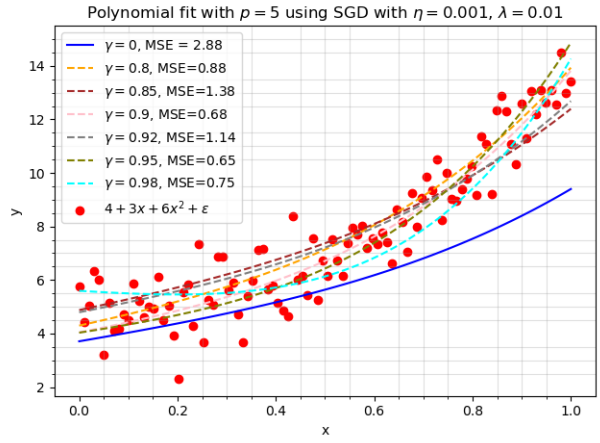

FIG. 7. Polynomial fits to our data using SGD for different values of the momentum term. We used 500 epochs and a batch size of 10. The optimal values $\eta = 0.001, \lambda = 0.01$ a the grid search of $\gamma = 0$ have been used.

Fig.7 shows that SGD fits the data rather poorly without momentum. This is likely because we use a small batch size (M=10) and so the direction of the gradient at each step will be very stochastic, jumping around the minimum but not fully converging. Momentum GD is not as affected by this stochasticity.

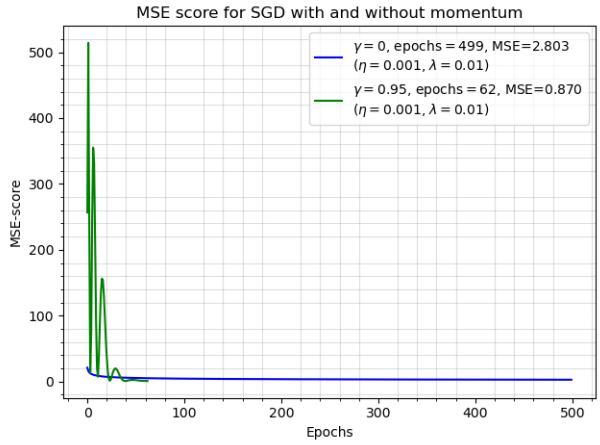

FIG. 8. MSE evaluated at each epoch of SGD with and without momentum.

We see in Fig.8 that momentum SGD reaches converges after only 62 epochs, which is faster than what was achieved using gradient descent.

Next we implemented Adagrad and RMSprop, which are algorithms tune the learning rate in a smarter manner. Neither algorithm was affected much by momentum.

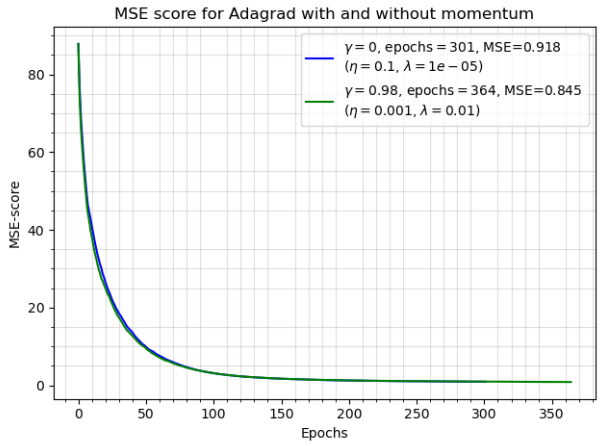

FIG. 9. MSE evaluated at each epoch for Adagrad with and without momentum.

From Fig.9, we see that Adagrad converges rather slowly. The use of momentum makes convergence slightly faster, although this may not be a consistent result as there is decent variability between runs.

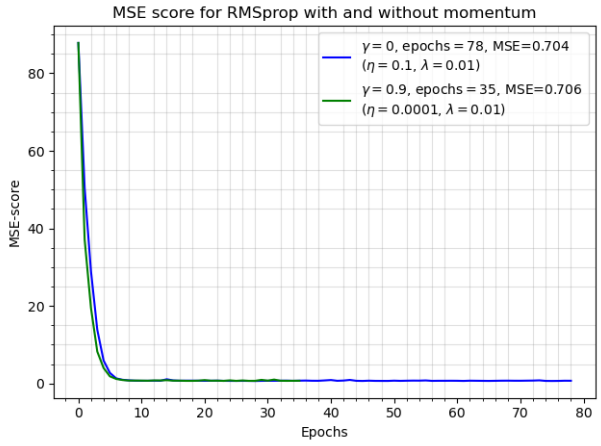

FIG. 10. MSE evaluated at each epoch for RMSprop with and without momentum.

Fig.10 shows RMSprop with momentum reaching convergence after only 35 epoch, although the algorithm performs almost as well without momentum.

Lastly, we implemented the Adam algorithm. In addition to adapting the learning rate, Adam also implements its own momentum term. We therefore set $\gamma = 0$.

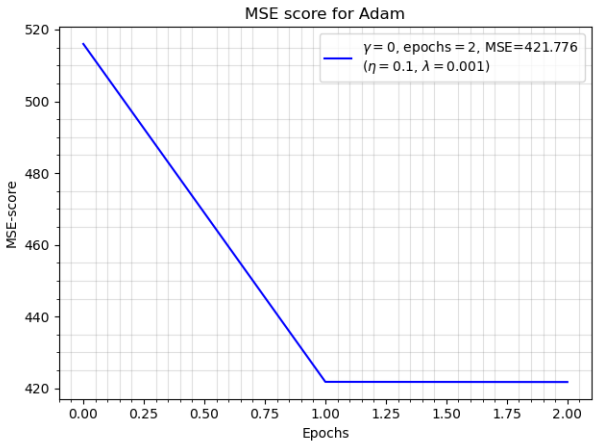

FIG. 11. MSE evaluated at each epoch for Adam.

Adam consistently flat-lines at an MSE of about 420 after a single epoch, as demonstrated in Fig.11. Setting the tolerance higher does not help. This is likely due to an error in the code, which have been unable to find.

TABLE I. Convergence rate and MSE using the optimal γ of each optimizer.

Optimizer	γ_{opt}	MSE	epochs
RMSprop	0.90	0.706	35
SGD	0.95	0.870	62
GD	0.0	0.999	157
Adagrad	0.0	0.918	301
Adam	0.0	421.8	2

B. Regression on the Franke Function

Because the gradient of the weights (and biases if the derivative of the activation function is a function of itself) of a given layer in a neural network are proportional to the the activations of the previous layer, as seen in Eq.(74), we cannot simply initialize them to zero. Instead, we initialize the weights of our neural network using the standard normal distribution. The biases are initially set to 0.01. This ensures that the neurons have some input which can be propagated in the first cycle[3].

The Franke function was constructed using $x, y \in [0, 1]$ with 20×20 data points. We added gaussian noise of standard deviation $\sigma = 0.1$. Since this is a bivariate function of x and y, our neural network will contain two input nodes and one output node. The output node contains no activation function, as we want the prediction to be unbounded.

To find the hyperparameters that optimize our neural network model, we performed grid searches over the learning rate, regularization rate, number of epochs, the mini-batch size, network architecture and activation function.

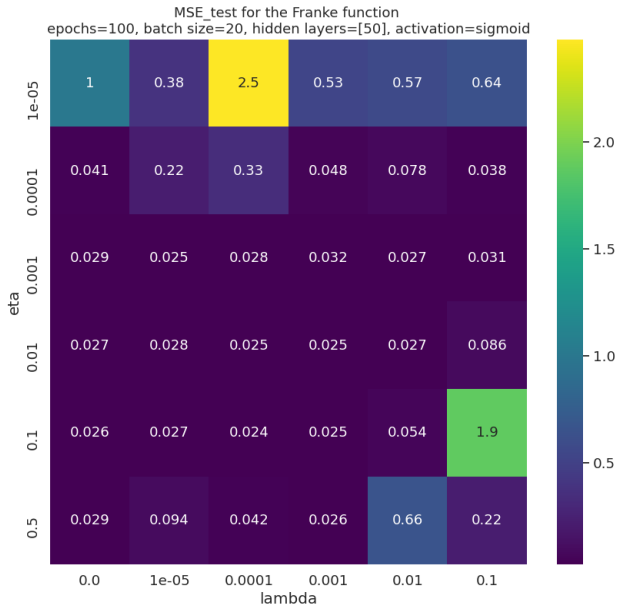

FIG. 12. MSE produced by our NN. Iterated over the learning rate η and the regularization parameter λ. The lowest MSE was found to be at $\eta = 0.1$, $\lambda = 10^{-4}$.

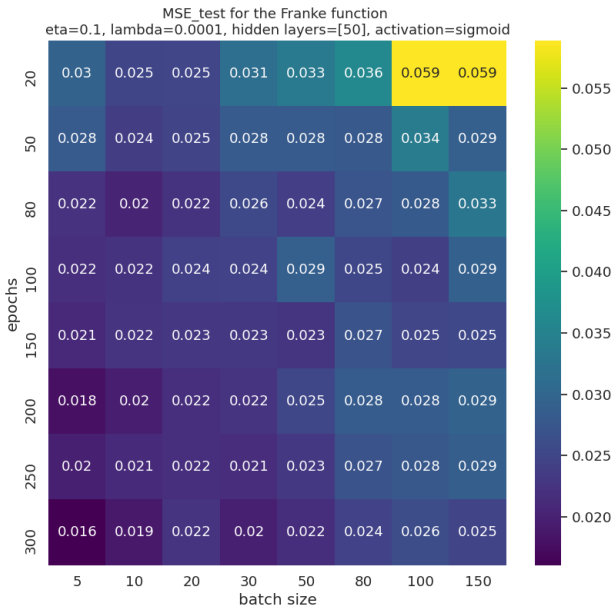

FIG. 13. MSE produced by our NN. Iterated over number of epochs and batch size. The lowest MSE was found to be at epochs=300, batch size = 5.

As seen in Fig.12 the network performs reasonably well in the range $\eta \in [0.1, 0.001], \lambda \in [0, 0.01]$. At $\eta = 10^{-5}$ we start seeing larger errors, likely due to the network being unable to reach a minimum in time due to the small learning rate.

Using these optimal values of η and λ, we perform a grid search over the number of epochs and batch sizes.

Fig.13 shows an increase performance as a function of epochs and a decrease in performance as a function of batch size.

Finally we explore various network architectures (i.e. number of hidden layers and number of nodes of each layer) and hidden layer activation functions.

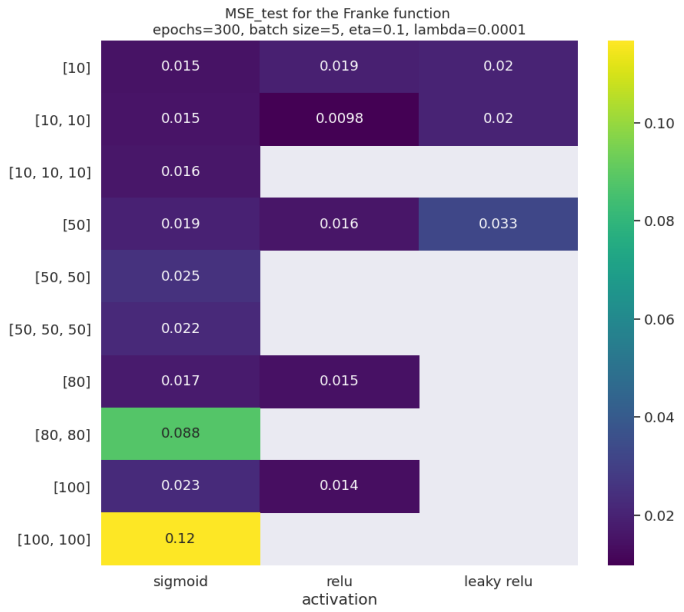

FIG. 14. MSE produced by our NN. Iterated over number of epochs and batch size. The lowest MSE produced with two hidden layers of 10 neurons each and using the ReLU activation function.

In Fig.14 we see that for the sigmoid function, the network produces the best results with hidden layers containing fewer nodes. As the number nodes increases, the MSE goes up.

The lowest MSE is given by the ReLU function with two hidden layers of 10 nodes each. However, as is the case when using leaky ReLU, for certain architectures the network outputs a NaN value, indicated by the grey areas.

Next, we compare our NN results with sklearns NN using the same hyperparameters and linear regression using a polynomial of degree 8 [9]. We calculate the $\mathrm{MSE}_{\mathrm{test}}$ and $\mathrm{R2}_{\mathrm{test}}$ values and plot the predited surfaces. For reference, Fig.16 plots the data with noise.

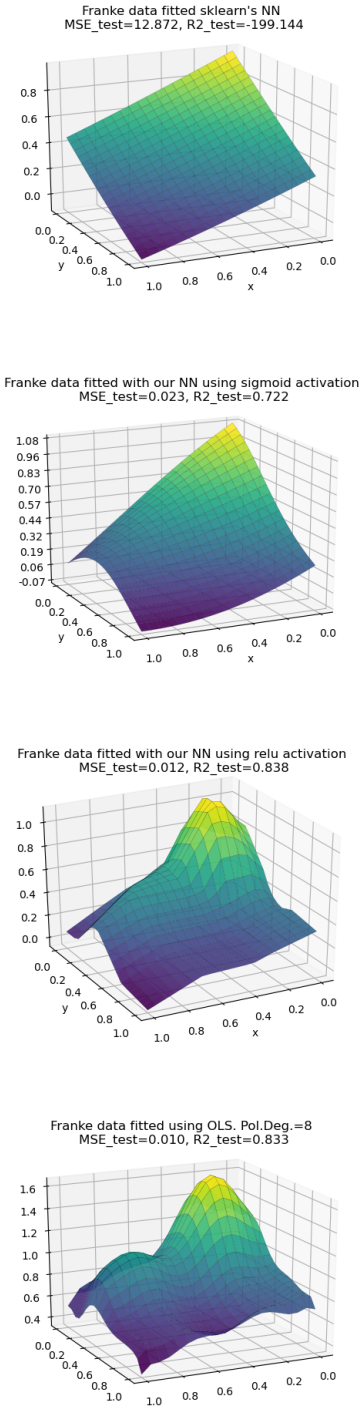

FIG. 15. Visual comparison of different models fitted to the Franke data. From top to bottom, we see sklearn's NN, our own NN using sigmoid, using ReLU and finally linear regression with a polynomial of degree 8.

[9] Which was found to be optimal in project 1[1]

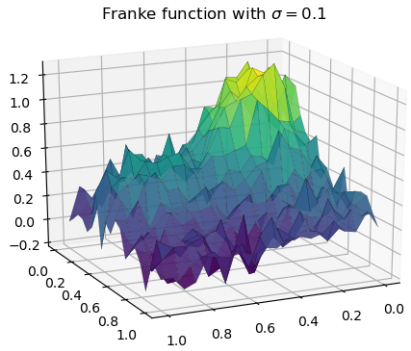

FIG. 16. Franke function with added noise gaussian noise of standard deviation $\sigma = 0.1$.

TABLE II. Performance of various models on the Franke function. Hyperparameters for the NNs: hidden layers=[10,10], Epochs=300, batch size = 5, $\eta = 0.1$ and $\lambda = 10^{-4}$.

Model	MSE_{test}	R2_{test}
Our NN w. sigmoid	0.023	0.722
Our NN w. ReLU	0.012	0.838
sklearn's MLP	12.87	-199.1
Linear Regression	0.012	0.838

As seen, our neural network using ReLU activations produced the same result as linear regression. Visually however, the fit is poorer. Although giving decent test scores, the sigmoid activation seems to produces a feature-less surface, regardless of the depth of the network. Using sklearns MLP with the same hyperparameters, we were only able to produce a plane and quite bad scores.

C. Classification of Breast Cancer Data

For classification, we will primarily focus on the Wisconsin Breast Cancer data set. It contains 569 data samples with 30 features relevant for classifying tumors as either benign or malignant.

To perform classification, we specify the activation function of the final layer to be the softmax function. This ensures that the activation of each neuron is between 0 and 1 and that they sum to 1, allowing us to interpret them as probabilities.

The cost function is specified to be the cross-entropy loss.

As in the regression case, we perform grid searches to find the optimal hyperparameters for this problem. If there are multiple maxima in our search, we typically pick the one situated in a stable[10] area.

FIG. 17. Test accuracy for our NN (top) and sklearn's NN (bottom) on breast cancer data. The highest accuracy of our network was 0.9912 at $\eta = 0.01$, $\lambda = 10^{-4}$.

Fig.17 compares the test accuracy of our network across various learning rates and regularization parameters with that of sklearn. We see that the ranges which

[10] i.e. surrounded by accuracies of the similar values.

give good accuracies differ between the two networks. We are not sure why this is. The architecture, optimization method and hyperparameters are the same as far as we know.

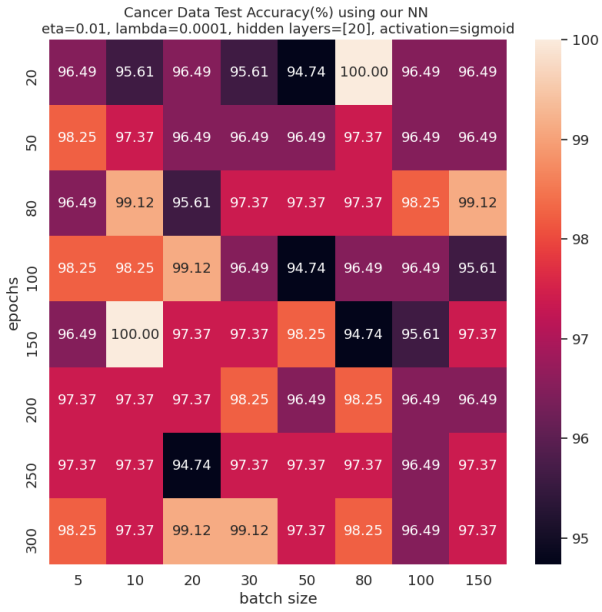

FIG. 18. Test accuracy for our NN on breast cancer data. The chosen maxima at (epochs=150, batch size=10) has an accuracy of 1.0.

We see in Fig.18 that the network performance is quite good and does not strongly depend on either the number of epochs nor the batch size for the given parameters, although accuracy seems to improve slightly with epochs.

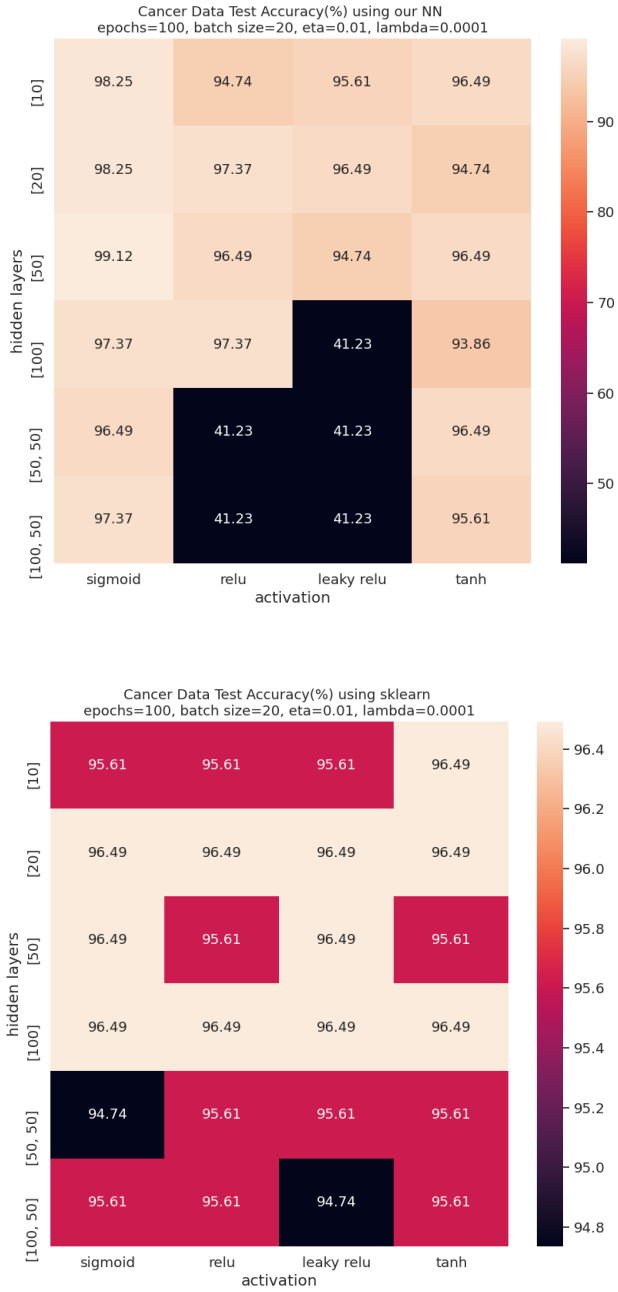

FIG. 19. Test accuracy for our NN (top) and sklearn's NN (bottom) on breast cancer data. The highest accuracy of our network was 0.9912 with (layers =[50], act=sigmoid).

As in the regression case, Fig.19 shows that our NN seems struggles when using ReLU and leaky ReLU as activations for certain architectures, although it does not explode up in the same manner.

We also see that sklearns NN produces more reliable results across activation functions, although not reaching the same accuracy as our NN for these parameters. As we saw, there is clearly some difference between the networks and so this is unsurprising given that the pa-

rameter search was done on using our NN.

TABLE III. Accuracy on cancer data using our NN that of sklearn with the optimal hyperparameters. Evaluated using cross-validation with 5 folds.

Neural network	Training acc.	Test acc.
Ours	0.9996	0.9313
sklearn's	0.9789	0.9244

To get a feel for how our network performs on another data set, one can easily alter the code to perform classification on the MNIST data set [11], this data set contains 8x8 pixel images of handwritten digits ranging from 0 to 9. Fig.20 shows decent results here as well, for a network with only one hidden layer of 20 nodes.

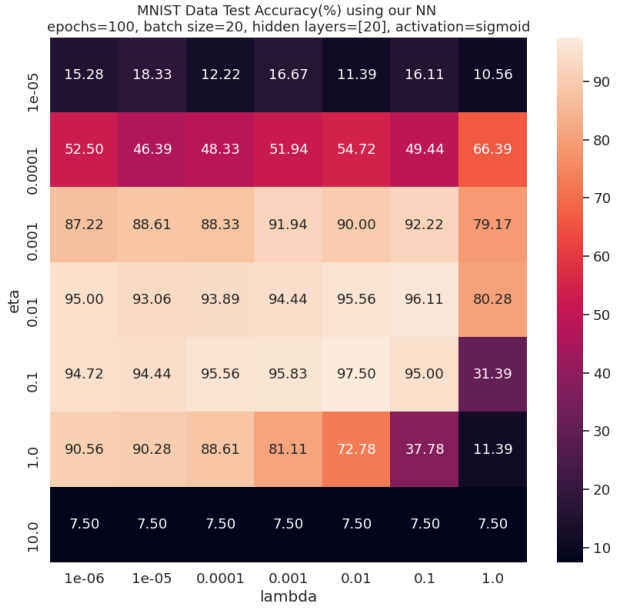

FIG. 20. Test accuracy on the MNIST data set using our NN.

Logistic Regression We now compare the tumor classification results of our NN with to that of our own logistic regression code. To be able expand our code to multi-class classification as well, we did this using softmax regression, which is discussed in the appendix A.

[11] By setting cancerBool to False in $breast_cancer.py$.

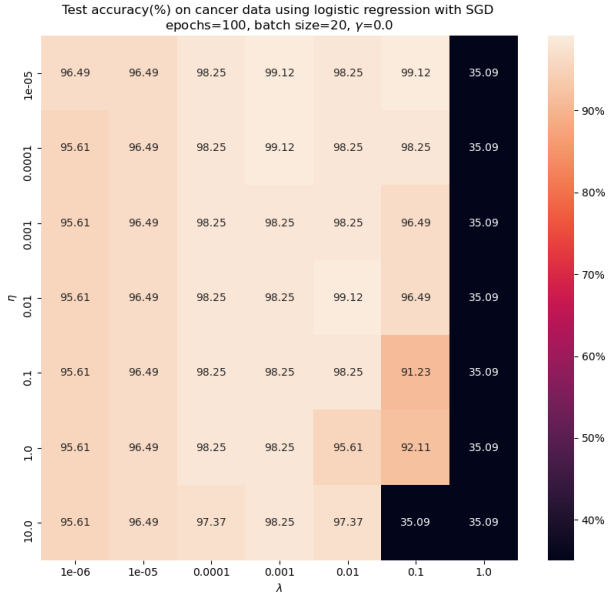

FIG. 21. Accuracy scores for our logistic regression code. SGD was used with 100 epochs, a batch size of 20 and no momentum or adaptive learning rate.

We see in Fig.21 that there are multiple maxima with accuracy $= 0.9912$. We choose $\eta = 10^{-4}$ and $\lambda = 10^{-3}$ as our optimal parameters, since this grid cell seems to lie in the most stable area, i.e. surrounded by other high accuracy grid cells.

Using these parameters, we re-calculate the test accuracy using cross-validation with 5 folds. This is also done using sklearn's logistic regressor.

TABLE IV. Test accuracy of different logistic regressors on the breast cancer data set. Calculated using cross-validation with 5 folds.

Log. reg. classifier	Test accuracy
Ours	0.9297
sklearn's	0.9332

IV. Discussion

We only implemented cross-validation on the final accuracy scores found in classification. Future improvements would include implementing resampling techniques during model evaluation in regression as well as in model selection for both tasks.

Because of this, the MSE scores obtained for the Franke function will have a greater uncertainty. Furthermore, the model selection for regression and classification may not have been optimal. Another thing that might skew the model selection process is the order in which we

perform the parameter search. As an example, the MSE values using ReLU activation in Fig.14 could likely be improved by performing a new parameter search using this as our activation function.

V. Conclusion

Neural networks possess a large number of hyperparameters, on which their performance greatly depends. Parameter tuning therefore plays an important role when applying neural networks as models.

In this project we did so for two very different tasks, only having to slightly alter the our FFNN.

The FFNN seems most suited for classification purposes, where it produced an accuracy of 0.933 on the Wisconsin Breast Cancer Data, slightly outperforming our logistic regression code which gave an accuracy of 0.929.

For regression however, it had a hard time learning the characteristics of the Franke function surface. Using the ReLU function as activation for two hidden layers of 10 neurons each produced the best result with MSE_{test}=0.012, which is the same as what was produced using linear regression. Linear regression seems the better choice in this task however, as it produced more reliable results and a better visual fit of the surface better, in addition to being computationally cheaper.

References

[1] H. Fossheim. *Project 1: Regression Analysis and Resampling Methods*. Oct. 2021. URL: https://github.com/fosheimdet/FYS-STK4155/tree/main/Project1.

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive computation and machine learning series. The MIT Press, 2017. ISBN: 978-0-262-03561-2.

[3] M. Hjorth-Jensen. *Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html.

[4] M. Hjorth-Jensen. *From Variational Monte Carlo to Boltzmann Machines and Machine Learning. Notebook 2: Boltzmann Machines*. July 2019. URL: https://github.com/CompPhysics/ComputationalPhysics2/blob/gh-pages/doc/pub/notebook2/pdf/notebook2-print.pdf.

Appendices

Throughout the appendices, we often make use of summation notation, where $a_i b_i \equiv \sum_{i=0}^{n-1} a_i b_i$. The column vector $\mathbf{a}$ and its transpose are then written as

$$\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = a_k \hat{e}_k, \quad \mathbf{a}^T = \begin{bmatrix} a_0 & a_1 & \dots & a_{n-1} \end{bmatrix} = a_k \hat{e}_k^T \tag{93}$$

Where $\hat{e}_k$ is the k-th unit basis vector. The operators $\frac{\partial}{\partial \mathbf{a}}$ and $\frac{\partial}{\partial \mathbf{a}^T}$ can be written as

$$\frac{\partial}{\partial \mathbf{a}} = \begin{bmatrix} \partial/\partial a_0 \\ \partial/\partial a_1 \\ \vdots \\ \partial/\partial a_{n-1} \end{bmatrix} = \partial_k \hat{e}_k$$

$$\frac{\partial}{\partial \mathbf{a}^T} = \begin{bmatrix} \partial/\partial a_0 & \partial/\partial a_1 & \dots & \partial/\partial a_{n-1} \end{bmatrix} = \partial_k \hat{e}_k^T$$

Where we denote $\partial/\partial a_k$ as ∂_k.

A. Logistic Regression

Multinomial Classification Assume we have a data set containing n training samples. Our explanatory variable has m features and belongs to one of C categories.

We model the probability that sample i belongs to category k using the Softmax function

$$p_{i,k} = \sigma(t_{i,k}) = \frac{e^{t_{i,k}}}{\sum_{l=1}^{C} e^{t_{i,l}}} \tag{1}$$

Where

$$t_{i,k} = \mathbf{X}_{i,*} \Theta_{*,k} = \mathbf{X}_{i,*} \vec{\theta}_k \tag{2}$$

If we include a bias in our model, we need to add a 1 to each sample so that

$$\mathbf{X}_{i,*} = [1, x_{i,1}, x_{i,2}, ...x_{i,m}], \qquad \vec{\theta}_k = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_m \end{bmatrix} \tag{3}$$

The likelihood function is the product of the probabilities (under our model) that the samples belong to their assigned categories.

If we one-hot encode the targets so that $\mathbf{y}$ becomes a $n \times C$ matrix, we can write the likelihood function as

$$L = \prod_{i=1}^{n} \prod_{l=1}^{C} (p_{i,l})^{y_{i,l}} \tag{4}$$

The likelihood is a measure of how well our model fits the data, and is what we wish to maximize. However, it is more practical to minimize the negative logarithm of this quantity, called the cross-entropy. We set this quantity as our cost function.

$$C = -\sum_{i=1}^{n} \sum_{l=1}^{C} y_{i,l} \log(p_{i,l}) \tag{5}$$

The gradient of the cost function with respect to the parameters of category k is given by

$$\frac{\partial C}{\partial \vec{\theta}_k} = -\sum_{i=1}^{n} \sum_{l=1}^{C} y_{i,l} \frac{1}{p_{i,l}} \frac{\partial p_{i,l}}{\partial t_{i,k}} \frac{\partial t_{i,k}}{\partial \vec{\theta}_k} \tag{6}$$

$$\frac{\partial p_{i,l}}{\partial t_{i,k}} = p_{i,l} \left(\delta_{l,k} - p_{i,k} \right) \tag{7}$$

$$\frac{\partial t_{i,k}}{\partial \vec{\theta}_k} = \frac{\partial \left(\mathbf{X}_{i,*} \vec{\theta}_k \right)}{\partial \vec{\theta}_k} = \left(\mathbf{X}_{i,*} \right)^T \tag{8}$$

$$\frac{\partial C}{\partial \vec{\theta}_k} = -\sum_{i=1}^{n} \sum_{l=1}^{C} y_{i,l} \frac{1}{p_{i,l}} p_{i,l} \left(\delta_{l,k} - p_{i,k} \right) \left(\mathbf{X}_{i,*} \right)^T \tag{9}$$

$$= -\sum_{i=1}^{n} \left(\mathbf{X}_{i,*} \right)^T \sum_{l=1}^{C} y_{i,l} \delta_{l,k} - y_{i,l} p_{i,k} \tag{10}$$

$$= -\sum_{i=1}^{n} \left(\mathbf{X}_{i,*} \right)^T \left(\underbrace{\sum_{l=1}^{C} y_{i,l} \delta_{l,k}}_{y_{i,k}} - p_{i,k} \underbrace{\sum_{l=1}^{C} y_{i,l}}_{1} \right) \tag{11}$$

And so for category k we have

$$\frac{\partial C}{\partial \vec{\theta}_k} = \sum_{i=1}^{n} \left(\mathbf{X}_{i,*}\right)^T \left(p_{i,k} - y_{i,k}\right) \tag{12}$$

We can calculate the gradient for every category as follows:

$$\frac{\partial C}{\partial \Theta} = \begin{bmatrix} \frac{\partial C}{\partial \vec{\theta}_1} & \frac{\partial C}{\partial \vec{\theta}_2} & \cdots & \frac{\partial C}{\partial \vec{\theta}_C} \end{bmatrix} \tag{13}$$

$$= \sum_{i=1}^{n} \left(\mathbf{X}_{i,*}\right)^T \left(\mathbf{p}_{i,*} - \mathbf{y}_{i,*}\right) \tag{14}$$

and so

$$\boxed{\frac{\partial C}{\partial \Theta} = \mathbf{X}^T \left(\mathbf{p} - \mathbf{y}\right)} \tag{15}$$

Binomial logistic regression In section II A, we found the following expression for the cost function of a binary logistic problem when using the sigmoid function to model the underlying PDF:

$$C\left(\mathbf{t}(\mathbf{x}, \boldsymbol{\beta})\right) = -\sum_{i=1}^{n} y_i t_i - \log\left[1 + e^{t_i}\right] \tag{16}$$

This can be written in summation notation as

$$C\left(\mathbf{t}(\mathbf{x}, \boldsymbol{\beta})\right) = -\left(y_i t_i - \log\left[1 + e^{t_i}\right]\right) \tag{17}$$

Where

$$t_i = \beta_0 + \beta_1 x_i \tag{18}$$

Which in summation notation is

$$t_i = \beta_k(\delta_{k0} + \delta_{k1} x_i) \tag{19}$$

In the following two subsections, we will find compact expressions for the first and second derivative of this cost function w.r.t. $\boldsymbol{\beta}$, or more specifically the gradient and the Hessian.

1. First derivative The first derivative of C w.r.t. $\boldsymbol{\beta}$, i.e. the gradient, is given by

$$\frac{\partial C(\mathbf{t})}{\partial \boldsymbol{\beta}} = \partial_k C(\mathbf{t}) \hat{e}_k = \frac{\partial C}{\partial t_i} \frac{\partial t_i}{\partial \beta_k} \hat{e}_k \tag{20}$$

The first partial derivative is given by

$$\begin{aligned} \frac{\partial C}{\partial t_i} &= -\frac{\partial}{\partial t_i}\left\{y_i t_i - \log\left[1 + e^{t_i}\right]\right\} \\ &= -\left(y_i - \frac{e^{t_i}}{1 + e^{t_i}}\right) = -(y_i - p_i) \\ \Rightarrow \underline{\frac{\partial C}{\partial t_i}} &= \underline{-(y_i - p_i)} \end{aligned} \tag{21}$$

While the second partial derivative is given by

$$\begin{aligned} \frac{\partial t_i}{\partial \beta_k} &= \partial_k\left(\beta_j(\delta_{j0} + \delta_{j1} x_i)\right) \\ &= \delta_{kj}(\delta_{j0} + \delta_{j1} x_i) = \delta_{k0} + \delta_{k1} x_i \\ \Rightarrow \underline{\frac{\partial t_i}{\partial \beta_k}} &= \underline{\delta_{k0} + \delta_{k1} x_i} \end{aligned} \tag{22}$$

Inserting Equations (21) and (22) into Eq.(20) gives

$$\begin{aligned} \frac{\partial C}{\partial \boldsymbol{\beta}} &= -(y_i - p_i)(\delta_{k0} + \delta_{k1} x_i)\hat{e}_k \\ &= -\begin{bmatrix} y_1 - p_1 + y_2 - p_2 + \ldots + y_n - p_n \\ (y_1 - p_1)x_1 + (y_2 - p_2)x_2 + \ldots + (y_n - p_n)x_n \end{bmatrix} \\ &= -\begin{bmatrix} 1 & 1 & \ldots & 1 \\ x_1 & x_2 & \ldots & x_n \end{bmatrix} (\mathbf{y} - \mathbf{p}) \end{aligned} \tag{23}$$

And thus

$$\boxed{\frac{\partial C(\mathbf{x}, \boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\mathbf{X}^T(\mathbf{y} - \mathbf{p})} \tag{24}$$

2. Second derivative We can now use Eq.(24) to find the second derivative of the cost function:

$$\begin{aligned} \frac{\partial^2 C(\mathbf{x}, \boldsymbol{\beta})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^T} &= \frac{\partial}{\partial \beta_k}\left\{-\mathbf{X}^T(\mathbf{y} - \mathbf{p})\right\} \hat{e}_k^T \\ &= \mathbf{X}^T \left\{\frac{\partial}{\partial \beta_k}\mathbf{p}\right\} \hat{e}_k^T = \mathbf{X}^T \left\{\frac{\partial}{\partial \beta_k} p_i\right\} \hat{e}_i \hat{e}_k^T \\ &= \mathbf{X}^T \left\{\frac{\partial p_i}{\partial t_i}\frac{\partial t_i}{\partial \beta_k}\right\} \hat{e}_i \hat{e}_k^T \end{aligned} \tag{25}$$

The derivative of the probability is given by

$$\frac{\partial p_i}{\partial t_i} = \frac{e^{t_i}(1 + e^{t_i}) - (e^{t_i})^2}{(1 + e^{t_i})^2} = p_i(1 - p_i) \tag{26}$$

And we already found the derivative of t_i in Eq.(22);

$$\frac{\partial t_i}{\partial \beta_k} = \delta_{k0} + \delta_{k1} x_i \tag{27}$$

We therefore get that

$$\left\{\frac{\partial p_i}{\partial t_i}\frac{\partial t_i}{\partial \beta_k}\right\}\hat{e}_i\hat{e}_k^T = \{p_i(1-p_i)(\delta_{k0}+\delta_{k1}x_i)\}\,\hat{e}_i\hat{e}_k^T$$
$$= \begin{bmatrix} p_1(1-p_1) & p_1(1-p_1)x_1 \\ p_1(1-p_1) & p_2(1-p_2)x_2 \\ \vdots & \vdots \\ p_n(1-p_n) & p_n(1-p_n)x_n \end{bmatrix}$$
$$= \underbrace{\begin{bmatrix} p_1(1-p_1) & 0 & \dots & 0 \\ 0 & p_2(1-p_2) & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & p_n(1-p_n) \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}}_{\mathbf{X}}$$

Inserting this back into Eq.(25) then gives

$$\boxed{\frac{\partial^2 C(\mathbf{x},\boldsymbol{\beta})}{\partial\boldsymbol{\beta}\partial\boldsymbol{\beta}^T} = \mathbf{X}^T\mathbf{W}\mathbf{X}} \tag{28}$$

B. The Newton-Raphson method

The Newton-Raphson method is an algorithm for finding estimates of the root of a function, $f(\mathbf{x})$.

Starting with an initial guess, $\mathbf{x}_0$, it performs a first-order Taylor expansion of f centered around $\mathbf{x}_0$.[12]

The root of this Taylor expansion, $\mathbf{x}_1$, is then found.

If the function value, $f(\mathbf{x_1})$, at this root is sufficiently[13] close to zero, it will be accepted as our estimate.

Otherwise, we perform another Taylor expansion, now centered at $\mathbf{x}_1$. This is repeated until $f(\mathbf{x}_i)$ is smaller than our tolerance.

In our case, we want to find the root of the gradient of the cost function, $\frac{\partial C}{\partial\boldsymbol{\beta}}$. To do this, let us first Taylor expand the cost function to second order around $\boldsymbol{\beta}_n$ and then differentiate so as to obtain the first order expansion of $\frac{\partial C}{\partial\boldsymbol{\beta}}$. For brevity, we leave it implicit that every instance of the gradient and Hessian are evaluated at $\boldsymbol{\beta}_n$, making independent of $\boldsymbol{\beta}$.

$$C(\boldsymbol{\beta}) \approx C(\boldsymbol{\beta}_n) + (\boldsymbol{\beta}-\boldsymbol{\beta}_n)^T\mathbf{g} + \frac{1}{2}(\boldsymbol{\beta}-\boldsymbol{\beta}_n)^T\mathbf{H}(\boldsymbol{\beta}-\boldsymbol{\beta}_n) \tag{29}$$

We then get

$$f = \frac{\partial C}{\partial\boldsymbol{\beta}} \approx \mathbf{g} + \frac{1}{2}\frac{\partial}{\partial\boldsymbol{\beta}}\left\{\boldsymbol{\beta}^T\mathbf{H}\boldsymbol{\beta} - \boldsymbol{\beta}^T\mathbf{H}\boldsymbol{\beta}_n - \boldsymbol{\beta}_n^T\mathbf{H}\boldsymbol{\beta} + \boldsymbol{\beta}_n\mathbf{H}\boldsymbol{\beta}_n\right\} \tag{30}$$

[12] This technically makes it a Maclaurin series, but it seems common to refer to this as a Taylor series instead, as will we.

[13] By this we mean that the estimated root lies withing the tolerance (which is specified by the user) of the true root.

The last term in the braces vanished since it is independent of $\boldsymbol{\beta}$. Notice so that $(\boldsymbol{\beta}_n^T\mathbf{H}\boldsymbol{\beta})^T = \boldsymbol{\beta}^T\mathbf{H}\boldsymbol{\beta}_n$, due to the Hessian being symmetric. Furthermore, since the cost function is a scalar, all the terms comprising it, including $\boldsymbol{\beta}_n^T\mathbf{H}\boldsymbol{\beta}$ must be scalars as well. The following therefore holds

$$\boldsymbol{\beta}_n^T\mathbf{H}\boldsymbol{\beta} = (\boldsymbol{\beta}_n^T\mathbf{H}\boldsymbol{\beta})^T = \boldsymbol{\beta}^T\mathbf{H}\boldsymbol{\beta}_n \tag{31}$$

Using this, Eq.(30) becomes

$$f = \frac{\partial C}{\partial\boldsymbol{\beta}} \approx \mathbf{g} + \frac{1}{2}\left(\frac{\partial\boldsymbol{\beta}^T\mathbf{H}\boldsymbol{\beta}}{\partial\boldsymbol{\beta}} - 2\frac{\partial\boldsymbol{\beta}^T\mathbf{H}\boldsymbol{\beta}_n}{\partial\boldsymbol{\beta}}\right) \tag{32}$$

By use of the identity in Eq.(38) of the appendix, we have

$$\frac{\partial\boldsymbol{\beta}^T\mathbf{H}\boldsymbol{\beta}}{\partial\boldsymbol{\beta}} = (\mathbf{H}+\mathbf{H}^T)\boldsymbol{\beta} = 2\mathbf{H}\boldsymbol{\beta} \tag{33}$$

And by applying the identity in Eq(37) of the appendix, the last term becomes

$$2\frac{\partial\boldsymbol{\beta}^T\mathbf{H}\boldsymbol{\beta}_n}{\partial\boldsymbol{\beta}} = 2\mathbf{H}\boldsymbol{\beta}_n \tag{34}$$

Which gives us the final expression for our first order Taylor expansion of $f = \frac{\partial C}{\partial\boldsymbol{\beta}}$ around $\boldsymbol{\beta}_n$:

$$\frac{\partial C}{\partial\boldsymbol{\beta}} \approx \mathbf{g} + \mathbf{H}(\boldsymbol{\beta}-\boldsymbol{\beta}_n) \tag{35}$$

The next estimate of the root of f is given by the root of this Taylor expansion:

$$f = \frac{\partial C}{\partial\boldsymbol{\beta}} \approx \mathbf{g} + \mathbf{H}(\boldsymbol{\beta}_{n+1}-\boldsymbol{\beta}_n) = 0$$
$$\Rightarrow \underline{\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n - \mathbf{H}^{-1}\mathbf{g}} \tag{36}$$

C. Vector calculus identities

In this section, we will prove the following useful identities

$$(i): \qquad \frac{\partial\left(\mathbf{b}^T\mathbf{a}\right)}{\partial\mathbf{a}} = \mathbf{b} \tag{37}$$

$$(ii): \qquad \frac{\partial\left(\mathbf{a}^T\mathbf{A}\mathbf{a}\right)}{\partial\mathbf{a}} = \left(\mathbf{A}+\mathbf{A}^T\right)\mathbf{a} \tag{38}$$

Proofs

$$\boxed{(i): \qquad \frac{\partial\mathbf{b}^T\mathbf{a}}{\partial\mathbf{a}} = \partial_k b_i a_i \hat{e}_k = \delta_{ik}b_i\hat{e}_k = b_k\hat{e_k} = \mathbf{b}} \tag{39}$$

Here we see that as long as the numerator yields a dot product of $\mathbf{a}$ and $\mathbf{b}$, the derivative w.r.t. $\mathbf{a}$ will be the same. I.e. we could replace $\mathbf{b}^T\mathbf{a}$ with $\mathbf{a}^T\mathbf{b}$ and get the same result.

$$
\begin{aligned}
(ii): \qquad \frac{\partial\left(\mathbf{a}^T\mathbf{A}\mathbf{a}\right)}{\partial\mathbf{a}} &= \partial_k\left\{a_l A_{lm} a_m\right\}\hat{e}_k = A_{lm}\partial_k\left\{a_l a_m\right\}\hat{e}_k \\
&= \left\{A_{lm}a_m\partial_k a_l + A_{lm}a_l\partial_k a_m\right\}\hat{e}_k \\
&= \left\{A_{lm}a_m\delta_{lk} + A_{lm}a_l\delta_{mk}\right\}\hat{e}_k \\
&= \left\{A_{km}a_m + A_{lk}a_l\right\}\hat{e}_k \\
&= \left\{\mathbf{A}_{k*}\mathbf{a} + \mathbf{A}^T_{k*}\mathbf{a}\right\}\hat{e}_k
\end{aligned}
$$

In the second line, we applied the product rule. In the last line we used $\mathbf{A}_{k*}$ to denote the k'th row vector of $\mathbf{A}$.

Summing over k then leaves us with

$$
\boxed{\frac{\partial\left(\mathbf{a}^T\mathbf{A}\mathbf{a}\right)}{\partial\mathbf{a}} = \left(\mathbf{A}+\mathbf{A}^T\right)\mathbf{a}} \tag{40}
$$

D. NN architechtures

Softmax classification with cross entropy cost function When dealing with multi-class classification problems, the Softmax function is typically used in the final layer. Doing so means that each output will lie between 0 and 1, and the outputs sums to 1. This allows us to interpret the output from each node of the final layer as the neural network's estimate that the input belongs to the class corresponding to the neuron.

The output of node c in the last layer is then

$$
a^L_C = \frac{e^{z^L_C}}{\sum_{c=0}^{n_L-1} e^{z^L_c}} \tag{41}
$$

We see that the activation of a given node depends on the $z's$ of all the other nodes in layer L. This means that the weight $\omega_{C,P}$ will affect not only a^L_C, but all the other activations of layer L as well. Since the cost function is a function of all these activations, we have to sum over all "paths" the weight can take to the cost function

$$
\frac{\partial C}{\partial\omega^L_{C,P}} = \frac{\partial z^L_C}{\partial\omega^L_{C,P}}\underbrace{\sum_{c=0}^{n_L-1}\frac{\partial a^L_c}{\partial z^L_C}\frac{\partial C}{\partial a^L_c}}_{\delta^L_C} \tag{42}
$$

Now, wish to evaluate how much the PMF generated by the Softmax differs from that of our one-hot encoded labels. We thus want a measure of the "distance" between these two distributions, and the function that best captures this distance is the cross entropy loss. We therefore use this as our cost function.

$$
C(\mathbf{y},\mathbf{a}^L) = -\sum_{c=0}^{n_L-1} y_c\log\left(a^L_c\right) \tag{43}
$$

The last term in equation Eq.(42) is then given by

$$
\underline{\frac{\partial C}{\partial a^L_c} = -y_c\frac{1}{a^L_c}} \tag{44}
$$

The activation of node c is given by

$$
a^L_c = \frac{e^{z^L_c}}{\sum_{i=0}^{n_L} e^{z^L_i}} \tag{45}
$$

and so

$$
\begin{aligned}
\frac{\partial a^L_c}{\partial z^L_C} &= \frac{\delta_{c,C}e^{z^L_c}\sum_{i=0}^{n_L-1}e^{z^L_i} - e^{z^L_c}e^{z^L_C}}{\left(\sum_{i=0}^{n_L-1}e^{z^L_i}\right)^2} && (46)\\
&= \frac{e^{z^L_c}}{\sum_{i=0}^{n_L-1}e^{z^L_i}}\frac{\delta_{c,C}\sum_{i=0}^{n_L-1}e^{z^L_i} - e^{z^L_C}}{\sum_{i=0}^{n_L-1}e^{z^L_i}} && (47)\\
& && (48)
\end{aligned}
$$

We therefore have

$$
\underline{\frac{\partial a^L_c}{\partial z^L_C} = a^L_c\left(\delta_{c,C} - a^L_C\right)} \tag{49}
$$

The error of node C in layer L is then

$$
\begin{aligned}
\delta^L_C &= \sum_{c=0}^{n_L-1}\frac{\partial a^L_c}{\partial z^L_C}\frac{\partial C}{\partial a^L_c} = \sum_{c=0}^{n_L-1} -y_c\frac{1}{a^L_c}a^L_c\left(\delta_{c,C} - a^L_C\right) && (50)\\
&= \sum_{c=0}^{n_L-1} y_c\left(a^L_C - \delta_{c,C}\right) = y_C a^L_C - y_C + \sum_{c\neq C} y_c a^L_C && (51)\\
&= a^L_C\underbrace{\left(y_C + \sum_{c\neq C} y_c\right)}_{1} - y_C && (52)
\end{aligned}
$$

Here we used the fact that the sum over all the elements of $\mathbf{y}$ is 1, due to it being one-hot encoded.

And so for a neural network with Softmax activations in the final layer and a cross-entropy cost function, the error in node C in this layer is

$$
\underline{\delta^L_C = a^L_C - y_C} \tag{53}
$$

Sigmoidal output with MSE cost function Our cost function is given by the mean squared error, namely

$$
C = \frac{1}{n}\sum_{i=0}^{n-1}(\tilde{y}_i - y_i)^2 = \frac{1}{2}\sum_{i=0}^{n-1}(a^L_i - y_i)^2 \tag{54}
$$

In differentiating w.r.t. a_c^L, all terms but $i = c$ vanish and we are left with

$$\underline{\frac{\partial C}{\partial a_c^L} = \frac{2}{n}\left(a_c^L - y_c\right)} \tag{55}$$

Now, if we are using the Sigmoid function

$$a(z) = \frac{1}{1+e^{-z}} \tag{56}$$

as our activation function in the out-most layer $l = L$, then we get

$$\underline{\frac{\partial a_c^L}{\partial z_c^L} = a_c^L(1 - a_c^L)} \tag{57}$$

Putting it all together, we have

$$\frac{\partial C}{\partial \omega_{j,k}^L} = \underbrace{\underbrace{\frac{2}{n}\left(a_c^L - y_c\right)}_{\frac{\partial C}{\partial a_c^L}} \underbrace{a_c^L\left(1 - a_c^L\right)}_{f'(z_c^L)}}_{\delta_c^L} a_p^{L-1} \tag{58}$$

The derivative of the cost function w.r.t. a weight in the outermost layer is then

$$\frac{\partial C}{\partial \omega_{c,p}^L} = \delta_c^L a_p^{L-1} \tag{59}$$

Very little changes when differentiating w.r.t. the bias:

$$\frac{\partial C}{\partial b_c^L} = \underbrace{\frac{\partial C}{\partial a_c^L}\frac{\partial a_c^L}{\partial z_c^L}}_{\delta_c^L}\underbrace{\frac{\partial z_c^L}{\partial b_c^L}}_{1} \tag{60}$$

$$\frac{\partial C}{\partial b_c^L} = \delta_c^L \tag{61}$$

E. Feature scaling

Some machine learning algorithms may have reduced performance unless the features are distributed according to the standard normal distribution. An example of this is the K-nearest neighbours algorithm, which counts the K nearest neighbours in the feature space and performs classification according to which of the categories is most represented. To do this requires it to calculate the euclidean norm to its neighbours, and unless the features are scaled, this yields suboptimal results. This also holds true for other algorithms using the euclidean norm, such as K-means clustering and Learning Vector Quantization (LVQ).

In principal component analysis (PCA), we are interested in the components that maximize the variance. Because the magnitude of the variance depends on the magnitude of the feature, and not its spread in units of standard deviation, we need to scale all features to make sure we are comparing apples to apples.

Furthermore, in using gradient descent to find the minimum of the cost function, it is important to scale the data. This is because if e.g. one of the features has a large magnitude compared with the others, the surface (in the case of two features) will be elongated along that feaure's axis, making gradient descent zigzag its way to the minimum rather than taking a straighter path, thereby decreases its performance.

Furthermore, scaling can avoid loss of numerical precision, since one of the features might vary drastically. Using scikit-learn's standard scaler is not recommended in this project, since it divides by the standard deviation and thus can make the features blow up in the case that this standard deviation is low.

1. Split and scale Note that we always want to scale our data, i.e. design matrix, after performing the train-test split.

Not doing so, means the test data will be included when calculating the mean of the columns of the design matrix, $\mathbf{X}$, and so will influence our training data[14]. We do not want this, since the model should be given no means of predicting the test data except through deducing the target function.

Furthermore, we have to be a bit careful in the way we scale our test-data. It should not be scaled using the mean and variance of the columns of $\mathbf{X}_{\text{test}}$, but rather using the mean and variance of the columns of $\mathbf{X}_{\text{train}}$, as was done for the training data.

To see this, consider the original unscaled problem, where our target function is approximated by

$$f(x_{tr}) \approx \tilde{y}(x_{tr}) = \beta_0 + x_{tr}\beta_1 + x_{tr}^2\beta_2 + ... + x_{tr}^p\beta_p \tag{62}$$

Where x_{tr} is just an arbitrary sample in the training set. We can write this more compactly as

$$\tilde{y}(x_{tr}) = \sum_{k=0}^{p} x_{tr}^k \beta_k \tag{63}$$

Now, if we scale our training data as follows:

$$x_{tr}^k \to x_{tr}^k - \mu_k = \bar{x}_{tr}^k \tag{64}$$

[14] since we subtracted the mean of a given column from every element in that column when performing scaling

Where μ_k is the mean of the k'th column of $\mathbf{X}_{train}$, i.e.

$$\mu_k = \sum_{tr=0}^{n_{train}} x_{tr}^k \tag{65}$$

And perform regression on this scaled training data, we get

$$\tilde{y}(x_{tr}) = \sum_{k=0}^{p} \bar{x}_{tr}^k \bar{\beta}_k \tag{66}$$

Here the regression coefficients are different, because the relationship between f and x is different due to the scaling of the training data.

We should therefore only use the regression coefficients obtained from scaled training data to perform predictions on data that has been scaled in the same manner as $\mathbf{X}_{train}$.