Project 2 - FYS-STK4155

Markus Borud Pettersen

November 13, 2020

Abstract

We derive and implement the backpropagation algorithm, and use it alongside stochastic gradient descent to train feed-forward neural networks on two tasks, a regression task and a classification task. The regression task consisted of predicting the value of the Franke function, and it was found that a single hidden layer neural network with Relu activation and a linear output performed best, achieving a test mean squared error of 0.046 and an R2 score of 0.68. This was found to match results obtained using a ordinary least squares linear regression model (MSE = 0.046, R2 = 0.63). The classification task consisted of predicting handwritten digits from the MNIST dataset, and a single hidden layer model with a Relu hidden layer activation, and softmax output activation achieved a test accuracy of 93%. A logistic regression model was also created and trained on the same dataset, achieving a test accuracy of 89 %.

Introduction

Powered by advances both in hardware and software, artificial intelligence (AI) has slowly become an integral part of everyday life. Found in everything from voice assistants on smartphones, to self driving cars [1], AI is not only enriching our lives today, but will likely continue to do so well into the future. Part of the reason why AI has become so ubiquitous in recent years, is due to the success of deep learning, a form of machine learning where the model is designed in terms of "layers" of computation, where the output of one layer is fed as input to the next. At the heart of many deep learning problems, is the backpropagation algorithm, which we will begin to explore in this project.

The backpropagation algorithm is quite simply a training procedure for deep neural networks, which works by exploiting the fact that the output of a regular deep neural network is nothing more than a multivariable function, whose goal is to minimize a cost or loss function that is unknown to us. Examples of such a loss function can be the squared distance between the network's prediction and some true value we wish to predict. As an interesting example, it was recently found [2] that deep neural networks develop the same spatial responses as real animal brains when performing a navigation task. In this case, the goal of the neural network is to predict its own actual position, and the cost function measures the actual distance to its true position.

Regardless of the exact form of the cost function, the backpropagation algorithm works by using the multivariable chain rule in order to compute the gradient of the loss with respect to the parameters of the model. By doing so, and using a minimization technique such as gradient descent, a deep learning model can actually learn to solve incredibly complex tasks, in a supervised manner.

In this project, we will as mentioned explore the backpropagation algorithm in some detail. First, however, we discuss the building blocks of deep neural networks, so-called perceptrons. We then move on to gradient descent and optimization, and outline how it can be used to train our models. We also consider possible improvements on standard gradient descent, including stochastic gradient descent (SGD) and gradient descent with momentum. In addition, we describe the explicit case of training a linear regression model. With the necessary building blocks in place, we derive the backpropagation algorithm, before giving a brief discussion on the choice of loss functions, as well as the functions to be used within our models, so-called activation functions.

After the theory is laid out, we describe how we implemented a linear regression model using stochastic gradient descent and Python, and how the same was done for a deep neural network. Finally, we outline how our deep neural net was used to perform logistic regression. We then describe the results of testing

our algorithms on two problems; one toy regression problem in which the network is tasked with predicting the function value of the so-called Franke function for a given coordinate input. The other task was a toy classification problem, in which the goal is to recognize hand-written digits from the MNIST dataset[3].

Background

The Multilayer Perceptron

Before we discuss the multilayer perceptron (MLP), we should study the single-layer perceptron. The perceptron is in fact a very simple model of a neuron, which is illustrated in Fig. 1. Note however that not all sources are so liberal as to call this process a perceptron, and would rather claim that a *true* perceptron should have a binary activation function applied to its output.

The model itself consists of a single node which receives some sort of input. Importantly, each input is weighted, mimicking the connection strength between neurons. The weighted inputs are then summed together, and an *activation function* is applied to the result. In general, this activation function may be anything, but it is most often non-linear, which means that the perceptron can be much more expressive than a corresponding linear model [4]. In addition, we can also have that the node always receives a constant input, or bias, which in some cases can be thought of as the firing threshold of the neuron [5].

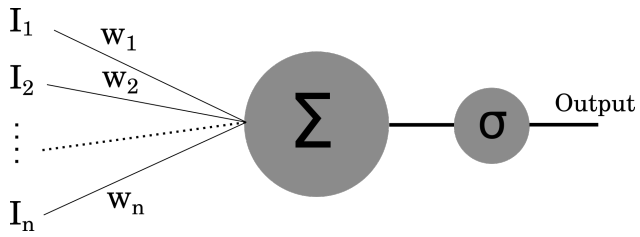

Figure 1: Illustration of single-layer perceptron model consisting of one "neuron" or node, where I indicates an input to the layer, w a weight connecting an input to the node, $\sum$ a summing operation, and σ some activation function applied to the sum.

Considering all of these operations, the output of a perceptron can be written

$$a = \sigma\left(b + \sum_j w_j I_j\right),$$

where σ is the activation function, b the bias of the node, while I and w is the set of all inputs and weights to the node. However, in the brain we can imagine that no neuron is an island, and so we might expect that the input our neuron receives is coming from other neurons, and that it is part of a larger network of connected nodes. This idea naturally leads to the concept of a multilayer perceptron, where we simply stack multiple perceptrons in sequential layers, where the output of one layer feeds into the next. As such, the output of a given perceptron takes the exact same form as before, but we now have to take into account that nodes in the previous layer are connected to nodes in the next. Considering the output of node number i, we have that

$$a_i^l = \sigma\left(b_i + \sum_j w_{ij} a_j^{l-1}\right),$$

where the superscript denotes the layer in which the node is located, while w_{ij} is the weight relating the activity of neuron j in layer $l-1$ (the previous layer) to neuron number i in layer l. However, we can see that this can be written conveniently in matrix form as

$$\mathbf{a}^l = \sigma(\mathbf{b} + W\mathbf{a}^{l-1}),$$

where $\mathbf{a}^l$ is the vector of activations of layer l (i.e. the activation of all neurons in this layer), while $\mathbf{b}$ is a vector holding all layer biases, while W is now a matrix containing all the weights connecting this layer to the previous one. Note that the use of the activation function σ is now understood to be element-wise.

Gradient Descent

We already mentioned that the models' ability to learn is due to the weights between layers; by tuning these, a seemingly simple model such as the MLP is able to perform a vast variety of tasks, and in fact, when combined with a nonlinear activation, it is capable of approximating *any* reasonably well behaved function [6]. Before approximating anything, however, we need a systematic way of determining the weights, such that the model is in some sense good. While a model may be good in many respects, we are concerned with supervised learning, i.e. tasks in which the model is presented with an input and a corresponding output or label, with the goal being to accurately predict the label. Ideally, we want a model that performs well both during training, but also during testing, when it is presented new data, which is what one would demand of a model that does some task in the real world.

The degree to which the model accurately predicts its targets, can be referred to as the loss of the model, and is quantified by the *loss function* that we choose. In this project, we will mainly use the mean squared error, or the cross-entropy, as loss functions. If we consider a model which has a set of parameters β, we want to train it in such a way as to minimize the loss function. For example, when we test our model on a dataset of unseen samples, we want the difference between the targets (the labels) and the predictions of the model to be as small as possible, for instance in terms of mean squared error. In other words, we want to find the minima of a loss function, which we will denote $C(W)$, where W is the set of all parameters of our model. One might ask why we do not consider C to be a function of its inputs (and the target values), which it in a sense is. For our purposes, however, we can imagine that we can only change our Ws, and so it makes sense to consider the inputs and targets fixed. Anyways, if our function is well behaved, a natural way of finding a minimum is by following the gradient of the function "downhill": The gradient typically tells us the direction of greatest increase, and so if we go in the opposite direction, we go in the direction of steepest descent.

If we know the gradient of the loss with respect to a given parameter, let's call it θ, we can iteratively update this parameter according to

$$\theta \to \theta - \eta \frac{\partial C}{\partial \theta},$$

where η is often referred to as the learning rate. Notice that this is nothing but taking a step in the direction of the gradient of C, with η being an indication of the size of the step. While this procedure is somewhat intuitive, it has its shortcomings. For one thing, it is sensitive to the choice of learning rate η. If the learning rate is too great, the cost might simply oscillate around a minimum, and never converge. If the learning rate is too small, convergence might be extremely slow. Other concerns include the fact that the learning rate does not discriminate between directions of the cost function "landscape". This is in contrast with higher order methods, like Newton's method, which depends also on the second derivative of the loss function. By doing so, a step with Newton's method is larger for flat directions, and smaller in steep directions. Unfortunately, higher order methods are typically prohibitively expensive in terms of computation [7].

There are ways to improve upon basic gradient descent, and in this project, we consider two possibilities, namely stochastic gradient descent, and the addition of a momentum term to the gradient. (mini-batch) Stochastic gradient descent is more or less exactly the same as regular GD, but instead of calculating the gradient for an entire dataset, we subdivide the dataset into so-called mini-batches, and approximate the gradient for each batch. In other words, if we have a dataset containing n samples, we split it into $n/m = d$ mini-batches of size m, and approximate the gradient in each iteration as

$$\nabla C \approx \frac{1}{d} \sum_{i}^{d} \nabla C_i,$$

where C_i is the loss for example i. In other words, we average the gradient over a mini-batch, not the full dataset, and use the approximated gradient to update our weights. Approximating the gradient in this way means that we do not have to compute the loss for the entire dataset in each step, but perhaps more

importantly, it introduces stochasticity to the gradient descent procedure. If we also add a momentum term by iterating

$$\begin{aligned} v &\to \omega v + \eta \nabla_\theta C \\ \theta &\to \theta - v, \end{aligned}$$

where ω is a momentum parameter, and v acts as a memory or momentum term, ensuring that the we keep moving in steep directions. Combining momentum with SGD can help us avoid getting stuck in local minima [6], but also improve generalizability.

Stochastic Gradient Descent & Linear Regression

Put simply, a linear regression model is of the form

$$\hat{\mathbf{y}} = X\beta,$$

where X is a so-called design matrix, constructed from the inputs to the model, while β is a vector of coefficients and $\hat{\mathbf{y}}$ a vector of outputs. Each row of the design matrix corresponds to one series of inputs, which we then map to a corresponding output, by choosing a suitable basis for the inputs, weighting each basis vector, and summing the result. This idea might be more explicit if we consider this equation on component form, in which

$$\hat{y}_i = \sum_j x_{ij}\beta_j,$$

i.e. the output is just a linear combination of the (transformed) inputs. However, we are free to choose a basis in which to represent the inputs to the model. For example, we can take the columns of X to be polynomials in the inputs, or sinusoidal functions. As outlined in the methods section, we will apply the former approach to Franke function data.

Now that we know how to construct a simple regression model, we need a way of determining an optimal set of coefficients, β. While it is the case for linear regression that an optimal set of weights can be found by solving the so-called normal equations, this is only possible when the design matrix is invertible/non-singular. As an alternative to this approach, we will instead perform gradient descent in order to find an optimal β. When using gradient descent, we do not need to perform any matrix inversion. As we described in the previous section, the gradient descent algorithm is quite straightforward, and we only need to determine the gradient of the loss function with respect to the parameters β in order to obtain a procedure for bettering our model. To see how this can be done, we can consider the mean squared error of a linear model:

$$C(\beta) = \frac{1}{n}\sum_i^n (y_i - \tilde{y}_i)^2,$$

where y_i is the actual sample label, and $\tilde{y}$ our model's prediction. Inserting for the prediction, we have

$$C(\beta) = \frac{1}{n}\left(\sum_i y_i - \sum_j x_{ij}\beta_j\right)^2,$$

and so the elements of the gradient of the loss with respect to the weights are

$$(\nabla C)_k = \frac{\partial C}{\partial \beta_k} = -\frac{2}{n}\sum_i (y_i - \tilde{y}_i)\sum_j x_{ij}\frac{\partial \beta_j}{\partial \beta_k},$$

by the chain rule of derivatives, since y_i is the true sample value, independent of our choice of model, and the inputs are also necessarily independent of the coefficients β. Since $\partial\beta_j/\partial\beta_k = \delta_{kj}$, with δ being the Kronecker-delta, we get

$$(\nabla C)_k = \frac{\partial C}{\partial \beta_k} = -\frac{2}{n}\sum_i (y_i - \tilde{y}_i)\, x_{ik},$$

where we recognize that the last factor is just a dot product between the columns in X and the difference between the targets and model predictions. We can therefore write this in matrix/vector form as

$$\nabla C = \frac{2}{n} X^T (X\beta - \mathbf{y}),$$

where ^{T} denotes a transpose. Since we are performing gradient descent for a given β, we get a simple update rule

$$\begin{aligned} \beta &\to \beta - \eta \nabla C \\ &= \beta - 2\frac{\eta}{n} X^T (X\beta - \mathbf{y}). \end{aligned}$$

In another popular variant of linear regression, so-called Ridge regression, an L2-penalization term on the weights is added to the loss function. Taking the mean squared error as an example again, the loss function for Ridge regression would be of the form

$$C_{Ridge} = \frac{1}{n} \sum_i (y_i - \tilde{y}_i)^2 + \lambda \sum_j \beta_j^2,$$

with λ being a regularization factor we can set. We immediately see that this modifies the gradient with respect to β only slightly:

$$\nabla C_{Ridge} = \frac{2}{n} X^T (X\beta - \mathbf{y}) + 2\lambda\beta,$$

which we can insert into the gradient descent step to obtain an update rule for Ridge regression. For a more thorough discussion on linear regression, see for example [8], or [9].

Backpropagation

Now that we have gone through some concrete examples for simple linear regression models, we will try to find an update rule for the weights of our multilayer perceptron.

As with the regression model, we start out considering the loss of the model, but this time we will not confine ourselves to selecting the mean squared error. To perform gradient descent, however, we need to determine the gradient of the loss with respect to a given parameter in the model. Somewhat surprisingly, we can actually do this quite generally without specifying a loss function, by using the chain rule. While there are certainly a few ways of doing this, we will follow the strategy of [5] going forward.

To do so, we note that the loss is a function of the model output, which is nothing but the output of the final layer. The output or activation of the final layer is a function of the layer weights, as well as its inputs. However, the inputs to the last layer is nothing but the activations of the previous layer, which is a function of *its* weights and inputs, and so on and so forth. We therefore gather that for a model with m layers, we can write the loss as

$$C = C(W^m, b^m, a^m),$$

with a^m being a function of the previous layer's activation (and implicitly a function of all other layer activations). To simplify things, we introduce the notation

$$a_i^l = \sigma(z_i^l) \quad \text{with} \quad z_i^l = \sum_{ij} w_{ij} a_j^{l-1} + b_i.$$

As promised, we will arrive at our result using the chain rule, and we start out by computing

$$\frac{\partial C}{\partial z_i^m} = \sum_k \frac{\partial C}{\partial a_k^l} \frac{\partial a_k^l}{\partial z_i^m},$$

i.e. the change in loss if we nudge the output activity slightly. We also know that

$$\frac{\partial a_k^m}{\partial z_i^m} = \frac{\partial \sigma(z_k^m)}{\partial z_i^m} = \sigma'(z_k^m)\delta_{ik},$$

as we are taking a partial derivative, and $\partial\sigma(z_k^m)/\partial z_i^m = 0$ for all $i \neq k$ *for most functions* (but not for softmax, for instance). This means that when we take the sum over all nodes, we simply get

$$\frac{\partial C}{\partial z_i^m} = \frac{\partial C}{\partial a_i^m}\sigma'(z_i^m).$$

Which is actually more useful than it might seem. While it is not obvious, we can calculate

$$\frac{\partial C}{\partial a_i^m}\sigma'(z_i^m),$$

using only the activation and activity of the output layer, as long as the derivative of the loss function is known. This is however no problem, as we simply choose the loss function to be well behaved functions with known derivatives, such as the mean squared error. To continue simplifying things, we introduce

$$\frac{\partial C}{\partial z_i^l} = \gamma_i^l,$$

so that

$$\gamma_i^m = \frac{\partial C}{\partial a_i^m}\sigma'(z_i^m).$$

While we are at it, we see that the previous expression can be written in vector form as

$$\boldsymbol{\gamma}^m = \nabla C \odot \sigma'(\mathbf{z}^m),$$

with ∇ denoting the gradient with respect to the output activations and $\odot$ elementwise multiplication. The next step is to find some way of relating γ_i^l to a known quantity. To do so, we use almost the same trick as before, by way of the chain rule:

$$\gamma_i^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_i^l}.$$

This is nothing is nothing but

$$\gamma_i^l = \sum_k \gamma_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_i^l},$$

but we know that z_k^{l+1} is a function of all its input nodes, and so we need to find

$$\frac{\partial}{\partial z_i^l}\sum_{kn} w_{kn}^{l+1}\sigma(z_n^l) + b_k,$$

and as we saw earlier, the derivative picks out $n = i$, so

$$\gamma_i^l = \sum_k \gamma_k^{l+1} w_{ki}^{l+1}\sigma'(z_i)$$

which we can restate on vector form as

$$\gamma^l = (W^{l+1})^T\gamma^{l+1} \odot \sigma'(\mathbf{z^l}). \tag{1}$$

We need a few more equations before we can determine our model's parameters. Fortunately, we are now prepared to calculate them directly. In order to perform gradient descent, we want

$$\frac{\partial C}{\partial w_{ij}^l} = \sum_k \frac{\partial C}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{ij}},$$

where we have once again used the chain rule. Massaging this further, and inserting for the activities in layer l, we get

$$\frac{\partial C}{\partial w_{ij}^l} = \sum \gamma_k^l \frac{\partial}{\partial w_{ij}^l}\left(b_k^l + \sum_n w_{kn}^l a_n^{l-1}\right) = \sum_{k,n}\gamma_k^l a_n^{l-1}\frac{\partial w_{kn}^l}{\partial w_{ij}^l},$$

but the weights do not depend on anything but themselves, and so the double sum boils down to the term where $n = j$, and $k = i$, so that

$$\frac{\partial C}{\partial w_{ij}^l} = \gamma_i^l a_j^{l-1}. \tag{2}$$

Finally, we need a way to update the biases, and we simply follow the same strategy as we did for the weights, by determining

$$\frac{\partial C}{\partial b_i^l} = \sum_k \frac{\partial C}{\partial z_k^l} \frac{\partial z_k^l}{\partial b_i^l} = \sum_k \gamma_k^l \frac{\partial b_k^l}{b_i^l},$$

as the weights do not depend on the biases in layer l, and there is no inter-dependency between biases, and once more the derivative picks out $k = i$, so

$$\frac{\partial C}{\partial b_i^l} = \gamma_i^l, \tag{3}$$

which means we are finally ready to state the update rules for *all* parameters in our model:

$$\begin{aligned} w_{ij}^l &\to w_{ij}^l - \eta \gamma_i^l a_j^{l-1} \\ b_i^l &\to b_i^l - \eta \gamma_i^l \end{aligned}, \tag{4}$$

with γ^l given as in (1). From all of this we can glean why this algorithm is called backpropagation; we first compute a *forward* pass, obtaining all activites and activations, before computing the value of the cost function, and *backpropagating* the error by way of γ^l. Once we have γ^l, we see that we can easily update the weights of layer l.

Choice of Loss & Activation Functions

As we saw in our disscussion of the backpropagation algorithm, our goal during training is to update our model's parameter according to the gradient of the loss function, so as to minimize the loss. It follows that our model's ability to learn depends directly on the gradient of the loss function, and therefore our choice of loss function. While different losses might be appropriate for different problems, we will consider two very common loss functions, the mean squared error, and the cross-entropy loss. We already saw how we could find the gradient of the mean squared error in the case of linear regression, but it is useful to also consider our multilayer perceptron, where the cost, if we assume that we have p outputs, is of the form

$$C = \sum_{i=1}^{p} (y_i - a_i^m)^2,$$

where we also assume that our model has m layers, as before. Taking the derivative with respect to the activation of a given output node, we get

$$\frac{\partial C}{\partial a_k^m} = -2 \sum_i (y_i - a_i^m) \cdot \frac{\partial a_i^m}{\partial a_k^m},$$

which is nothing but

$$\frac{\partial C}{\partial a_k^m} = -2(y_k - a_k^m),$$

as the sum picks out the term $i = k$ from the partial derivative. We see that we can also write this on vector form as

$$\nabla C = -2(\mathbf{y} - \mathbf{a}^m).$$

While the mean squared error is a tried and true quantity, it is worth investigating whether other loss functions might actually be better in some cases. As promised, we will also consider the so-called cross-entropy loss, which is given by

$$C_i = -\sum_k^p y_k \ln a_k^m,$$

with a_k^m being the k-th output of our model for the i-th sample, as before. Immediately, we see that for our loss to be positive, we require that the target values, and model outputs, must be greater than or equal to zero. In fact, cross-entropy loss is most often applied to problems where we consider the model outputs to be probabilities. Examples might be classification tasks, where we typically demand that our model outputs a number representing the probability that a sample belongs to a given class, instead of a binary yes/no. In the case where we only have two classes to choose between, so-called binary classification, we can instead use the binary cross-entropy. In this case, we simply imagine having one output node, but instead of assigning a separate probability to each class, we say that an output either belongs to one class (with probability p), or not (with probability $1 - p$).

As we mentioned briefly in our derivation of the backpropagation algorithm, the quantity γ^m, i.e. the output "error" could be separated into two factors, as long as there were no dependencies between outputs. For multiclass classification, such as categorizing handwritten digits in the MNIST dataset, we do have such troublesome dependencies, as we use the softmax as an output activation. For an input vector $\mathbf{x}$, the softmax function outputs

$$S(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}},$$

or in other words, each output is normalized, so that the outputs sum to one as if they were a probability distribution. This also happens to be exactly what we want in order to use the cross-entropy loss function, and so we only need to determine γ^m, if we wish to use our network for (multiclass) classification. In other words, we need

$$\gamma_i^m = \sum_k \frac{\partial C}{\partial a_k^m} \frac{\partial a_k^m}{\partial z_i^m},$$

where $a_k^m = S(\mathbf{z}^m)$ is now a function of all its input activities. The first factor is fortunately easy, as

$$\frac{\partial C}{\partial a_k^m} = -\frac{\partial}{\partial a_k^m} \sum_n^p y_n \ln a_p^m = -\frac{y_k}{a_k^m},$$

with p being the number of outputs, and the sum picks out $n = k$, as we have seen before. For the last factor we have

$$\frac{\partial a_k^m}{\partial z_i^m} = \frac{\partial}{\partial z_i^m} \frac{e^{z_k}}{\sum_j e^{z_j}} = S(z_j)\delta_{ik} - S(z_k)S(z_j),$$

where we have omitted some intermediary steps, and $S(z_j) = a_j^m$ is the softmax function, as before. Inserting these results, and carrying out the sum for the δ-term, we get

$$\gamma_i^m = -y_i + \sum y_i a_j^m,$$

where the a_k^ms from the loss derivative cancel with those of the softmax derivative. We then remember that $\mathbf{y}$ is a one-hot vector, as a target can only belong to one class (with 100 % probability, and zero for all other classes), and so the last sum picks out a_i^m:

$$\gamma_i^m = a_i^m - y_i, \tag{5}$$

a surprisingly simple result! We see therefore that we can circumvent the problem of misbehaving outputs during backpropagation, by specifiyng the loss and output activation function collectively, and using the above result for γ_i^m. As we will see, this is what is done in our neural network implementation.

In addition to the loss functions we choose, the activation functions applied to each layer is also of great significance. As we know, an activation function must have a defined derivative, or gradient, if we are to use it in conjunction with backpropagation. As such, there are a handful of activation functions commonly used in deep learning. In order to save some time and space, these functions, along with their gradients, are stated without derivations in Table 1, but they are readily computed.

In Fig. 2, all these activation functions and their gradients are illustrated, for a linear input x in the interval $[-5, 5]$. As we can see, the sigmoid has a relatively small gradient for all values, except for the

	Function	Gradient
Relu	$R(x) = \max(0, \mathrm{x})$	$R'(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$
Leaky Relu	$R_\alpha(x) = \begin{cases} 0.01x, & x < 0 \\ x, & x > 0 \end{cases}$	$R'_\alpha(x) = \begin{cases} 0.01, & x < 0 \\ 1, & x > 0 \end{cases}$
Sigmoid	$\sigma(x) = 1/(1 + e^{-x})$	$\sigma'(x) = \sigma(1 - \sigma)$
Linear	$I(x) = x$	$I' = 1$

Table 1: Various activation functions and their gradients.

region around $x = 0$, away from which it decays quickly. We can also take note of how it saturates for large absolute values of x. The linear function is of course familiar, and simply does nothing to its input, and the gradient is constantly equal to one. For the Relu, we see that the nonlinearity is contained in that negative values are cast to zero, while positive values are just passed through. This means that the gradient has a discontinuity at $x = 0$, and is zero for $x < 0$. The fact that the Relu has zero gradient for negative inputs, (and the function is zero itself), can cause nodes with Relu activations to stop learning, in a problem sometimes affectionately referred to as the "dying Relu" problem. To combat this, we can introduce the leaky Relu, where we see that the gradient is actually some small nonzero number, also for $x < 0$. This should hopefully lead to better performance than a standard Relu activation. Note that for the leaky Relu shown in Fig. 2, the gain for $x < 0$ was actually set to be 0.05, instead of the 0.01 listed in Table 1, but this was only done for illustration purposes (in actuality, this seems to be referred to as a parametric Relu, but that is another matter).

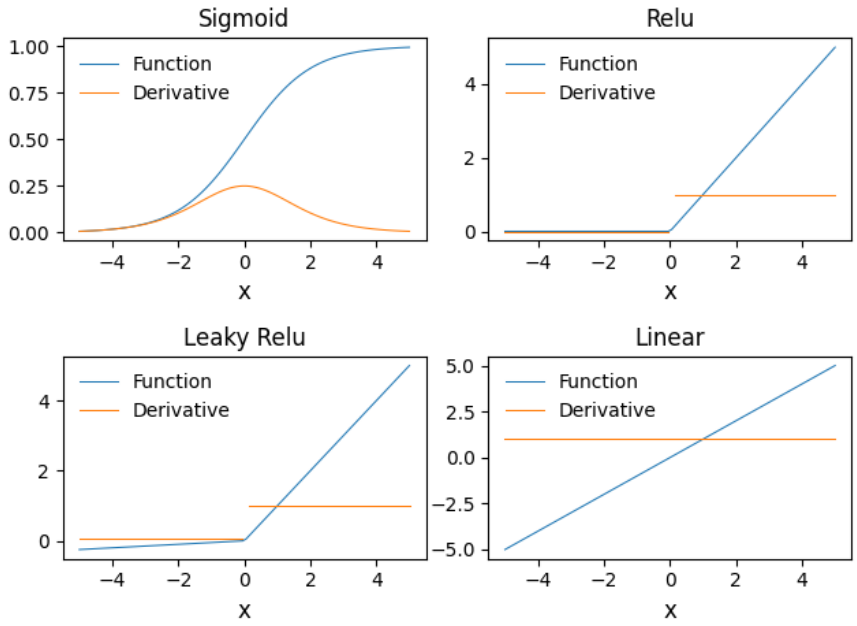

Figure 2: Various activation functions and their gradients. Note that for the leaky Relu, the gain of the negative inputs has been increased from 0.01 to 0.05, as to be visible.

Methods

In order to investigate the usage of stochastic gradient descent for optimization, an SGD optimizer was implemented in Python, and tested on a linear regression task, consisting of predicting the function value of the Franke function. The dataset consisted of 1000 samples of the Franke function, for uniformly distributed input values in the interval [0,1]. For each sample, gaussian noise drawn from a normal distribution $\mathcal{N}(0,1)$ and scaled by a factor of 0.2 was added. For more on the Franke function, see for example [8]. In all cases, 75 % of the dataset was used for training, and the rest for test purposes. For the Franke function dataset, all data were normalized by subtracting the mean of the training inputs, in all cases. For the SGD optimizer, an optional momentum term was added, but was not used further in this project.

For the linear regression, the design matrix was chosen to be polynomials in the input coordinates x and y up to degree $p = 14$, such that each row was of the form

$$X_i = [x^0y^0 \quad x_1y^0 \quad y^0x^1 \quad x^1y^1 \quad x^2y^0 \quad x^2y^2 \quad ... \quad x^py^p],$$

i.e. with entries x^iy^j for all $0 \leq i, j \leq p$. This was done both for Ridge regression, as well as the ordinary case. For both models, gradients were computed according to the expressions derived in the background section.

In order to explore the effects of variable mini-batch sizes, learning rate, and the possible addition of learning rate decay, a grid search was performed for the OLS model. The learning rates were taken to be geometrically spaced in the interval $[10^{-4}, 0.1]$, while the batch sizes were taken to be the set $[10, 50, 100, 816]$, with 816 corresponding to the length of the entire training dataset. Models were trained with and without learning rate decay, with the decaying learning rate given by

$$\eta(e) = \frac{\eta_0}{e \cdot m + 1},$$

with η_0 being the initial learning rate, e the training epoch, and m the number of mini-batches in an epoch. For reference, an OLS model was also trained on the same dataset by solving the normal equations directly.

In the Ridge regression case, a grid search was performed for the same learning rates as in the OLS case, but also for shrinkage parameters λ in the interval $[10^{-4}, 0.1]$, with a batch size of 10. In all cases, the MSE

and R2 score of the trained models were calculated using bootstrap resampling, for a total of 100 botstrap samples.

In order to investigate the use of the backpropagation algorithm, a feed forward neural network (FFNN) was implemented in Python. The network supported a flexible number of layers, each with a different number of nodes, with each layer being implemented according to the multilayer perceptron model outlined in the background section. Each model could also utilize its own activation function. The model weights were updated according to the backpropagation equations (2) and (3), by way of mini-batch stochastic gradient descent. For the hidden layer activation functions, only simple functions with no dependency between node activities were allowed. For the output layer, in order to allow for multi-class classification, a softmax activation function with corresponding crossentropy loss was implemented.

In order to explore the effects of weight initialization on training, a few different weight initialization schemes were implemented: For all layers, the kernel weights could either be initialized randomly according to a normal distribution $\mathcal{N}(0, 0.05)$, or using Glorot uniform, or Xavier initialization [10]. For the Glorot uniform initialization, weights were initialized according to a uniform distribution, in the interval $[-a, a]$, where

$$a = \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}},$$

with n_{in} being the number of inputs to a layer, and n_{out} the number of outputs. For the biases, initialization could be done either to all zero, or according to a normal distribution $\mathcal{N}(0, 0.01)$

To test the FFNN, it was applied to the same 1000-sample Franke function dataset as the linear regression models. As before, the dataset was scaled by subtracting the training set mean. Taking into account the fact that the scaled Franke function data could be negative, all FFNNs featured linear output activations. For all models, the loss was taken to be the mean squared error.

In order to test that the network was working as intended, we first trained a simple, one hidden layer model on the Franke function dataset. The hidden layer consisted of 128 nodes, and featured a Relu activation. The output consisted of a single node, with linear activation, as mentioned. The biases of the hidden layer were all initialized to zero, while the kernel was initialized according to a normal distribution $\mathcal{N}(0, 0.05)$. As a verification, an equivalent network, with equal layers and weight initialization was implementing using Tensorflow and the Keras API.

For the test, both our model and the Tensorflow version was trained using minibatch SGD, with a learning rate of 10^{-3} and no learning rate decay or momentum. The batch size was 10, and each model was trained for a total of 1000 epochs.

Following testing, a grid search was performed for the same model architecture, for the Franke function dataset. The search parameters were learning rates in the interval $[10^{-4}, 0.1]$, as well as weight regularization parameters λ in the same interval, i.e. $[10^{-4}, 0.1]$. Note that the same regularization parameter was applied to all model weights during training, and each model was trained for 100 epochs. As before, the test MSE and R2 score was calculated for all models, using bootstrap resampling.

To investigate the application of different activation functions, the same one-hidden layer model was trained on the scaled Franke function dataset, but in this case, on model was trained for each hidden layer activation, i.e. either sigmoid, Relu, leaky Relu or linear. Note that the output activation was still purely linear. Each model was trained for a total of 500 epochs, with a mini-batch size of 10. The learning rate was set to be 10^{-3} for each model.

To explore the effects of adding multiple layers to the FFNN, what we choose to call RandomNet was implemented and trained on the scaled Franke function dataset. For a given RandomNet model, the number of hidden layers is chosen at random between 1 and 10, and for each layer, the activation function was chosen randomly as either Relu, LeakyRelu, Sigmoid, or a linear activation. In addition, the weight initialization of each layer was also chosen randomly, with the kernel being initialized either using the normal distribution, or Glorot uniform distribution described earlier. The biases were initialized either to zero, or to the slightly narrower normal distribution. The goal of RandomNet was not only to verify that our algorithms functioned for deeper neural nets, but to see if a "randomly sampled" network could outperform the simpler, single hidden layer models described previously. A total of 10 RandomNet models were trained on the Franke function dataset, each for 100 epochs, with a learning rate of 10^{-3} and a batch size of 10.

In order to test our network on a classification task, a single hidden layer FFNN was trained on the MNIST dataset of handwritten digits [3], consisting of 70000 (28×28) images of grayscale images of handwritten digits

between 0 and 9. In all cases, 60000 images were used for training, and the remaining 10000 for testing. As grayscale images, each pixel takes on a value between 0 and 255, and images were therefore normalized by dividing by 255. It should be noted that the specific MNIST dataset used was obtained via the scikit-learn library, and the 28×28 images were flattened in advance, so that each input example fed to the network was a 784-element vector of pixel intensities.

The FFNN consisted of a single hidden layer, with 128 nodes, and Relu activation. As there are ten digits to predict, the output layer featured 10 nodes and a softmax activation, with the cost function taken to be the crossentropy. The output error term used in backpropagation was computed according to (5). In order to verify the softmax/crossentropy classification scheme, an equivalent model was once again implemented using Tensorflow and Keras. For both models, the kernel weights were all initialized using the Glorot uniform scheme, while biases were initialized to zero. In both cases, models were trained for 50 epochs, the learning rate was 10^{-3}, and the batch size was 50. In addition to the cross-entropy, the accuracy of the model was also computed at the end of each epoch. In this context, the accuracy was taken to be the number of times that the largest model output coincided with the target digit.

Finally, to compare the FFNN with a simpler model, we used the feed-forward network architecture to implement a simple logistic regression model and trained it on the MNIST dataset. The model itself was taken to be a simple linear combination of the input pixels, so that input to node i is

$$z_i^m = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n,$$

with x_n representing pixel number n. We can also write this on matrix form as

$$\mathbf{z}^m = XB,$$

where X is the design matrix, with rows given by the input pixel values of a given sample, i.e.

$$X_i = [1 \quad x_1 \quad x_2 \quad ... \quad x_n]$$

Note that the first column corresponds to an intercept or bias. If we n input pixels, B is a $(n \times 10)$ matrix containing all our model parameters, i.e. with columns

$$B_i^T = [\beta_{0i} \quad \beta_{1i} \quad ... \quad \beta_{ni}],$$

where we added an extra index to indicate that a given set of parameters maps to a single output node.

In order to perform multiclass classification, the output activation was taken to be the softmax function, and the loss function was the cross-entropy. Said somewhat differently, the logistic regression model was a feed forward classifier network with no hidden layers, and was trained in the same way as for the FFNN.

A logistic regression model was then trained on the MNIST dataset, with the kernel weights initialized using the Glorot uniform scheme, and the biases to zero. The addition of an $L2$ regularization factor λ to the weights was also explored, by training models with $\lambda \in [0, 0.01, 0.1, 1]$. These models were trained for 50 epochs, with a batch size of 50, and a learning rate of 10^{-3}. Finally, for comparison purposes, a logistic regression model was implemented using the scikit-learn library and trained on the MNIST dataset.

Results

Fig. 3 shows the grid search results for the OLS model trained using SGD on the Franke function dataset, with and without learning rate decay. We immediately see that the models with learning rate decay perform sligtly worse in terms of test MSE and R2 score than the corresponding constant learning rate models. For the constant LR models, the lowest MSE of 0.053 was for a learning rate of 0.01 and a batch size of 10. The corresponding MSE was 0.07 for a model with learning rate decay, for the same batch size and initial learning rate. We also find the same trend in the R2 score, the best of which (R2 = 0.55) was achieved by the same constant LR model. It is of interest to note that performance in both cases tended to decrease with increasing batch size. Note that initial testing showed that learning rates in excess of 0.1 tended to cause the error and R2 score to explode (the R2 score tended to negative infinity).

As for the Ridge regression (with constant LR) search results, shown in Fig. 4, we see a similar trend, in that the best MSE and R2 score is found for a learning rate of 0.1, at 0.055 and 0.52, respectively. There

is, however, only a marginal difference between the $\lambda = 10^{-3}$ and $\lambda = 10^{-4}$ case. On the other hand, we see that large values of λ significantly deteriorates performance.

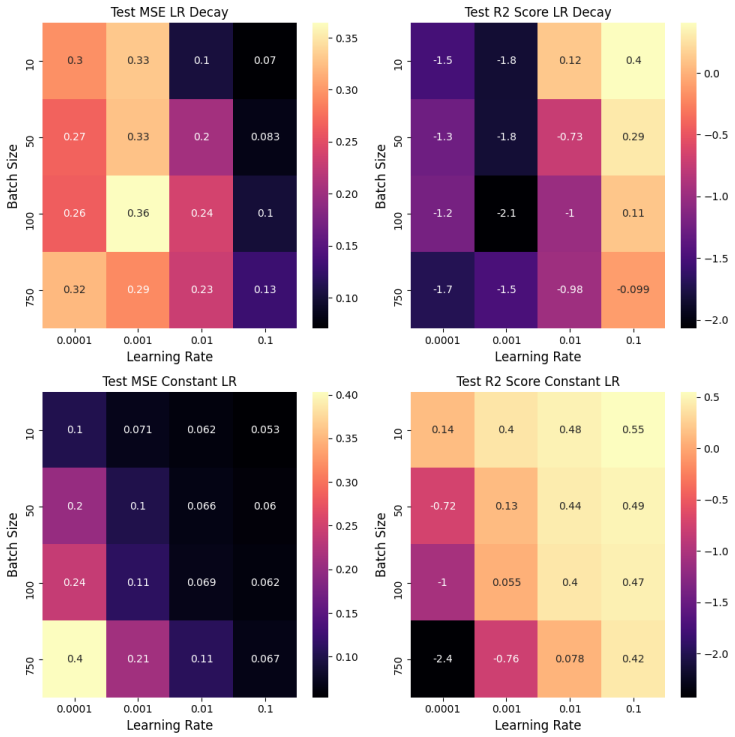

Figure 3: Grid search results for batch size and learning rate for regression models using stochastic gradient descent trained on the Franke function dataset. Shown are the test set mean squared error and R2 scores for models trained with and without learning rate decay.

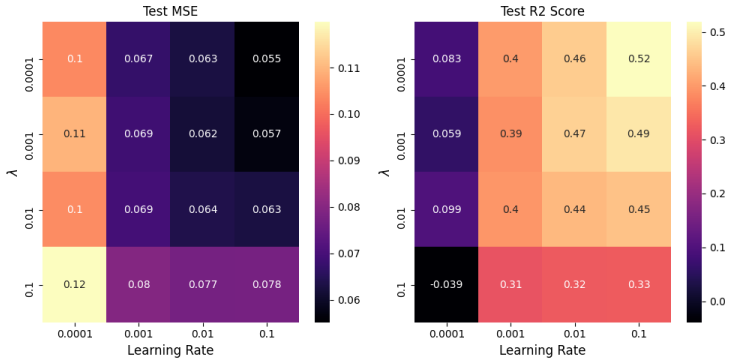

Figure 4: Grid search results for Ridge regression models trained using stochastic gradient descent on the Franke function dataset. Shown are the test set mean squared errors and R2 scores.

The reference OLS model achieved a MSE of 0.046, and an R2 score of 0.63, while the scikit-learn regression model achieved a similar result, with a MSE of 0.050, and an R2 score of 0.61.

Fig. 5 shows the training and test mean squared error of the FFNN trained on the Franke function dataset, along with the test error of a corresponding tensorflow model trained on the same data. As we can see, the test error of the two models follow each other closely, with both achieving a test MSE of approximately 0.06. We also note that the training error is slightly lower.

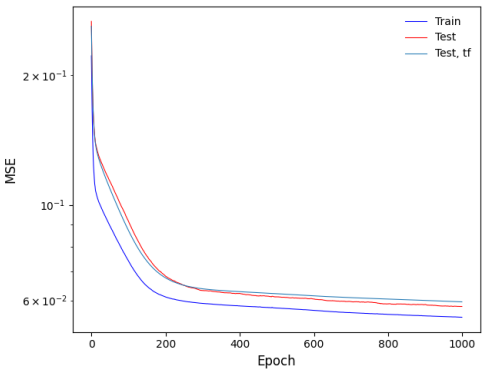

Figure 5: Mean squared error for a feed forward neural network with a single hidden layer with Relu activation, and a linear output trained on the Franke function dataset. For reference, the performance of an equivalent tensorflow model is shown for the test set.

Fig. 6 shows the search results for the one hidden layer FFNN trained on the Franke function dataset. As with the regression model, we once again observe that the lowest MSE (and highest R2 score) is achieved for a learning of 0.1. Interestingly, an MSE of 0.046 is equivalent to that of the reference OLS model, but its R2 score of 0.68 is actually slightly higher. We also observe that higher regularization values tend to decrease performance, with there being little difference between the $\lambda = 10^{-4}$ and $\lambda = 10^{-3}$ cases.

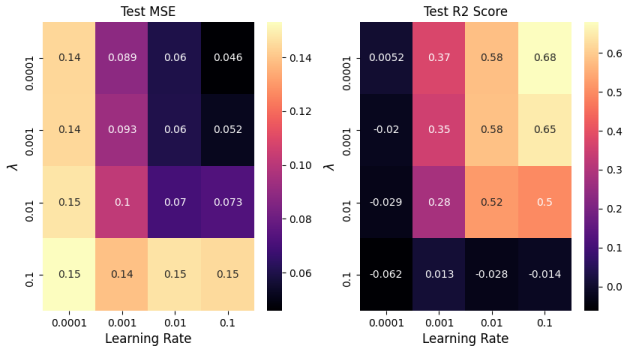

Figure 6: Grid search results for learning rates, and L2 weight regularization parameter λ, conducted for a feed forward neural network with a single hidden layer with Relu activation, and a linear output trained on the Franke function dataset.

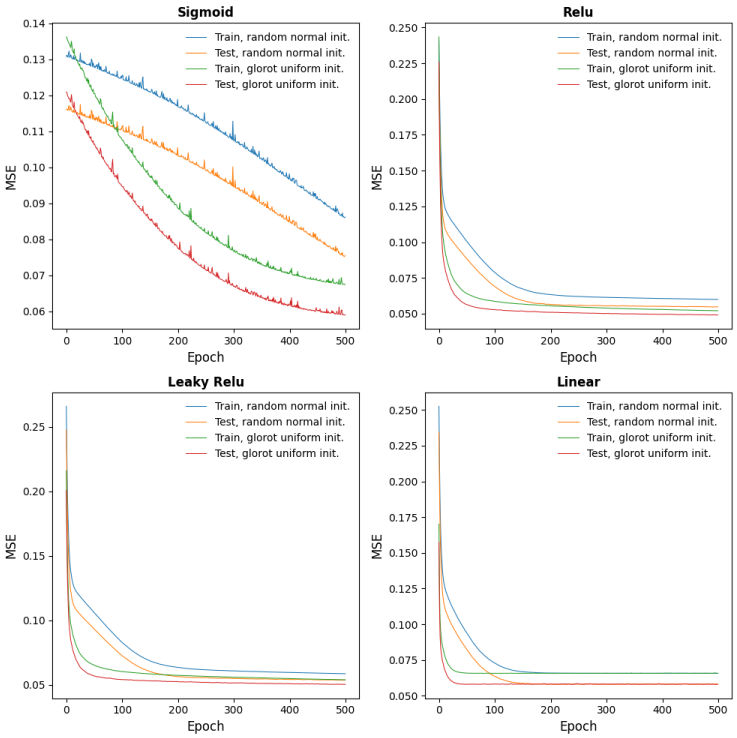

Figure 7: Mean squared error for the one hidden layer neural network trained on the Franke function dataset, for different hidden layer activation functions, and kernel weight initialization schemes. Shown are both training and test errors. Note that the output activation was linear in all cases.

When it comes to the effects of altering the hidden layer activation function and model initializations, Fig. 7 tells us, among other things, that for this model architecture, using a sigmoid function almost guarantees slow convergence. In fact the best sigmoid model, with Glorot uniform initialization, only achieved a test MSE of around 0.06 after 500 epochs. In terms of MSE, this appears to be comparable to the linear case, but for a linear activation with Glorot uniform initialization, convergence appears to take less than 50 epochs. Likewise, the Relu and Leaky Relu models exhibit fast convergence, but they also achieve a lower test MSE, of around 0.05. In fact, the best model featured regular Relu activation, and achieved a MSE of 0.049. Note that, in contrast to the best results for searches performed previously, the learning rate used for these trials was 10^{-3}. An a somewhat surprising aside, we note that the test error is actually consistently lower than the training error.

Concerning the RandomNet, out of ten trained models, the best model achieved a test set mean squared error of 0.057, and featured seven hidden layers. Interestingly, this model only featured Relu, leaky Relu, and linear hidden layer activations, i.e. no Sigmoid activations.

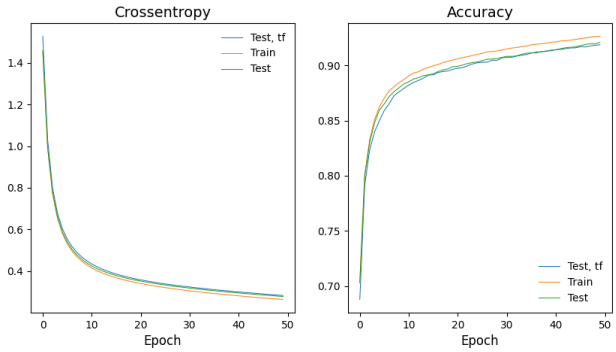

Figure 8: Cross-entropy and accuracy of a feed forward neural network predicting the digits of the MNIST dataset. The network features a single hidden layer with a Relu activation function, and a softmax output activation. Shown are both the test set and training set results. For reference, the test set performance of an equivalent tensorflow model is also indicated.

Fig. 8 shows the train and test cross-entropy loss, as well as accuracy, of the FFNN with a single hidden layer with a Relu activation, when trained to perform classification on the MNIST dataset. Also shown is the corresponding test loss of an equivalent tensorflow model trained on the same data. We can immediately see that the FFNN learns to predict MNIST digits, achieving a final test set accuracy of 0.93. As a confirmation of this result, we see that the test accuracy of the corresponding tensorflow model follows the FFNN closely. It is worth noting that the training accuracy is slightly higher than the test accuracy.

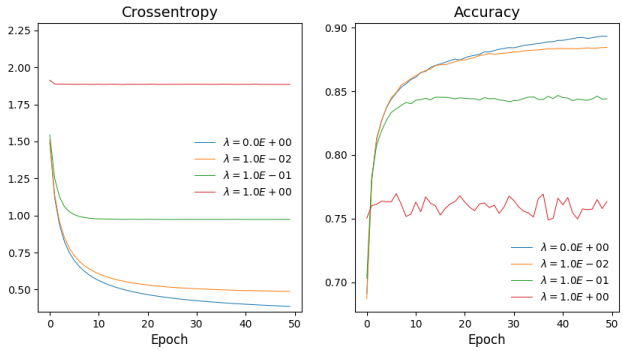

Figure 9: Test set accuracy and cross-entropy of the logistic regression model trained on the MNIST dataset, for different values of the applied L2 regularization parameter λ.

Finally, Fig. 9 showcases the test cross-entropy loss and accuracy for the logistic regression model, for different L2 regularization factors λ. Judging from the test accuracy, it would appear that regularization terms in excess of 0.1 is detrimental to model performance, as the $\lambda = 1$ model flatlines at an accuracy of around 0.76. Interestingly, there appears to be little benefit to adding weight regularization as the $\lambda = 0$ model outperforms the $\lambda = 10^{-2}$, achieving a final test accuracy of 0.89.

For reference, the scikit learn model achieved a test accuracy of 0.92, but a warning was raised that the model did not converge after the allowed 100 iterations.

Discussion

There are a multitude of possible results to address in this discussion, but the first that might be of interest, was the fact that the addition of learning rate decay did not seem to improve model performance, rather the opposite. One could object that the range of learning rate values was simply too small to see a benefit of

learning rate decay, but when values larger than 0.1 were attempted, the MSE tended to blow up, with and without learning rate decay. It would therefore be interesting to try different decay schemes, as it seemed from preliminary testing that the learning rate simply decayed too quickly, stalling learning.

In order to combat the exploding training loss, a more sensible learning rate of 10^{-3} was chosen for most models, as it seemed to give a decent compromise between convergence and stability during training. Taking into account that the models shown in Fig. 7 still converged quickly, especially when initialized using the Glorot uniform scheme, this seems to be a reasonable compromise for the Franke function data. As for the MNIST results both for the FFNN and the logistic regression model, we can tell from the slope of the accuracy curves in figures 8 and 9 that training has not quite converged, and so it could be useful to train for more epochs. However, especially for the FFNN, this was time consuming, as the MNIST dataset is somewhat large, and the python implementation of the network is quite slow. On the other hand this could be done very quickly using a faster external library, e.g. Tensorflow.

On the topic of external libraries, it appears that the Tensorflow models to a large extent verify our algorithm implementations, as they achieve comparable test set results both for the MNIST and Franke function datasets. In addition, we see that the results match up nicely at each epoch end. While figures 5 and 8 does show some difference between the two in terms of test set performance, this difference is rather tiny. Care was taken to ensure that the networks were initialized according to the same distributions, but the seed was not set explicity for each model, which might help to explain any differences.

Having established that our model probably works as intended, it is interesting that models with sigmoid activations perform rather poorly. This might be due to the fact that the gradients of the sigmoid function tend to become small, especially for deeper networks. One possible way of dealing with this is to use the binary cross-entropy loss and a sigmoid output activation, which it can be shown, should improve training significantly in some cases [5]. However, in order to do this for our regression case, we would have to scale our Franke function data to be between 0 and 1, in order to be compatible both with the cross-entropy loss, and the sigmoid function's range of outputs.

Another surprising result, is that the highly complex RandomNet regression model performed slightly worse than the corresponding single hidden layer models. On the other hand, the best RandomNet models did not feature sigmoid activations, which seems to fit nicely with the results in Fig. 7, and the previous discussion of the sigmoid.

Another interesting result, is that the single hidden layer FFNN with Relu activation achieved a test set MSE on the Franke function dataset of 0.046, equal to the best regression model, but with a slightly higher R2 score, of 0.68, compared to 0.61 for the least squares baseline model. This might hint that a FFNN might generalize better than a typical regression model, and so it would be interesting to perform more thorough grid searches, with more search values and longer training times for the results indicated in Fig. 6.

On the other hand, Fig. 7 suggests that the single hidden layer, Relu activation FFNN should converge faster than the 100 epochs used for the aformentioned search, *if* we were to change its kernel initializations to the Glorot uniform Scheme. It would therefore be interesting to perform a new grid search similar to that shown in Fig. 6, only with Glorot uniform initialization, to see if results could be improved further. Another concern is that the models in Fig. 7 actually seem to converge to a slightly higher MSE, which might suggest that the learning rate of 10^{-3} is too low, or that the models easily become stuck in local minima. As such, it would be interesting to also study the effects of adding momemtum.

For classification, we can note that the single hidden layer FFNN outperforms the scikit learn logistic regression model (and our own logistic regression model) in terms of accuracy, achieving a test accuracy of 0.93 vs. 0.92 of the scikit-learn model. This suggests that the slightly more complicated architecture of the FFNN is beneficial, compared to the very simple logistic regression model. It is nonetheless impressive that a model as simple as the logistic regression achieves such a high accuracy in a modest amount of training time.

As for the implementation of our logistic regression, there seems to be agreement between it and the scikit-learn model, in the sense that the final accuracy of 0.89 for our model is comparable to the 0.92 of the scikit model. This is especially true if we take into consideration that the scikit-model uses a higher order optimization method, and that our model did not appear to have converged fully after the allotted 50 training epochs. It would therefore be of interest to train our model again, for longer, and possibly study the effects of different learning rates. On the other hand, this can also be said of the FFNN, which also did not appear to converge in 50 epochs. Considering the fact that the FFNN is a quite simple extension of

the logistic regression model, and that it achieved a markedly better result in the same amount of training epochs, it would appear that the FFNN is the better choice of model for this classification problem, even though the logistic model performed surprisingly well given its simplicity.

Another interesting development would be to explore the addition of more layers to a FFNN used for classification. However, in the context of this project, adding more layers provides a huge amount of added model complexity, which makes the models more difficult to analyze and compare. The single hidden layer model therefore makes sense as a powerful test case. That said, it would be interesting to build a classifier using the RandomNet procedure, just to see whether a very complex model could outperform a simple one.

As a final point of concern, we see that the addition of L2 regularization only seems to degrade performance, both for the regression, and classification tasks. It is therefore pertinent to apply our models to more diverse tasks, where regularization might be more important. Alternatively, these findings suggest that something has gone awry with our regularization implementation. It might also be that the way regularization is applied, with the bias and kernel subject to the same L2 penalty factor λ for all layers, is somehow damaging to the model's predictions. Regardless of the cause, these results warrant further investigations.

Conclusion

By implementing the backpropagation algorithm from scratch in python, and using it to train feed forward neural networks, we have explored a multitude of problems in machine learning.

In order to verify the integrity of our algorithms, we trained both linear regression and feed forward neural networks models on a dataset consisting of samples drawn from the Franke function. Results showed that a neural network with a single hidden layer with Relu activation, and a linear output, achieved a test mean squared error of 0.046 and an R2 score of 0.68, matching the best linear regression model which achieved a MSE of 0.046 and R2 score of 0.63. Furthermore, equivalent networks trained using external libraries exhibited the same performance, suggesting the algorithms worked as intended.

We also explored the effects of different activation functions, learning rates, number of hidden layers and number of nodes in a layer for the same regression task. We found that for a single hidden layer, a sigmoid activation tended to be detrimental to performance, but we propose that this might be alleviated by scaling the data appropriately, and using a combined sigmoid output activation and cross-entropy loss function. For the single hidden layer model, we also found that the Glorot uniform initialization scheme leads to faster convergence and better model performance, for all tested loss functions, when compared to initialization from a normal distribution. We also attempted to train RandomNet models, models with randomly selected number of layers, nodes, and hidden layer activations, and found that the best models achieved a test MSE of 0.057, slightly higher than simpler models. Interestingly, the best RandomNet models did not feature sigmoid hidden layer activations.

Finally, we trained a feed forward neural network to classify images of handwritten images, using the MNIST dataset. This model still featured a single hidden layer with Relu activation, but with a softmax output activation, and a cross-entropy loss function. This model achieved a test set accuracy of 93% after 50 epochs. For comparison, a simple logistic regression model was also created, achieving a test accuracy of 89 %. For verification purposes, a logistic regression model was also implemented using an external library, achieving a test accuracy of 92 %. Due to its flexibility, and the high performance of the single hidden layer model, it was determined that the feed forward neural net was the slightly better choice of model for classification, even though the logistic regression model performed well given its simplicity. As a development, it would be interesting to train these networks for longer, or possibly study the effects of adding more layers to the feed forward network, and see if a higher classification accuracy could be achieved.

References

[1] *Autopilot AI*. en. Tesla, Inc. URL: https://www.tesla.com/autopilotAI (visited on 11/08/2020).

[2] Andrea Banino et al. "Vector-based navigation using grid-like representations in artificial agents". en. In: *Nature* 557.7705 (May 2018), pp. 429–433. ISSN: 1476-4687. DOI: 10.1038/s41586-018-0102-6. URL: https://www.nature.com/articles/s41586-018-0102-6 (visited on 11/13/2020).

[3] Yann Lecunn, Corinna Cortes, and Christopher Burges. *MNIST handwritten digits dataset*. URL: http://yann.lecun.com/exdb/mnist/.

[4] Trevor Hastie, Robert Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. 2nd ed. Springer series in statistics. New York, NY: Springer, 2009. ISBN: 9780387848570 9780387848587.

[5] Michael A. Nielsen. *Neural Networks and deep learning*. Chapters 1-3. Determination press, 2015. URL: http://neuralnetworksanddeeplearning.com/.

[6] Hjorth-Jensen. *Week 39: Optimization and Gradient Methods*. University of Oslo, UiO. 2020. URL: https://compphysics.github.io/MachineLearning/doc/pub/week39/html/week39.html (visited on 11/13/2020).

[7] Morten Hjorth-Jensen. *Week 40: From Stochastic Gradient Descent to Neural networks*. University of Oslo, UiO. 2020. URL: https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html (visited on 11/13/2020).

[8] Markus Borud Pettersen. "Project 1 - FYS-STK4155". Project report. Oslo: University of Oslo, 2020.

[9] Pankaj Mehta et al. "A high-bias, low-variance introduction to Machine Learning for physicists". In: *Physics Reports* 810 (May 2019). arXiv: 1803.08823, pp. 1–124. ISSN: 03701573. DOI: 10.1016/j.physrep.2019.03.001. URL: http://arxiv.org/abs/1803.08823 (visited on 10/27/2020).

[10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". en. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, Mar. 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html (visited on 11/12/2020).

Appendix

All codes developed for this project are freely available at https://github.com/markusbp/fys_stk4155/tree/master/project2