The Mathematics Behind Learning

Simon Butt
Dr M. Schmuck

A thesis presented for the degree of
Bachelor of Science

Department of Mathematics
Heriot Watt University

Contents

Abstract

The aim of this project is to take a look at the topic of machine learning from a mathematical point of view. This will involve outlining fundamental topics while by working with a basic model and problem. Then I will extend these concepts to more complicated problems and explore the mathematical demands doing this. Finally I will conclude by bringing together our findings and analysing the results on a benchmark machine learning problem.

1 Basic Concepts - Polynomial Learner

We can begin by asking the question of what characterises an algorithm to learn? A popular definition of machine learning is "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if it's performance at tasks in T, as measured by P, improves with experience E" [1]. To explore this definition, we are going to initially look at a very basic machine learning model. The polynomial learner is a small regression problem. Our task is to take an input value and using this model, attempt to predict the corresponding target output. This will be achieved by an optimisation training process using a dedicated training dataset.

1.1 The Problem

The underlying function we will initially model is:

$$y(x) = sin(2\pi x) + \frac{3}{2}rand(0, \sigma) \tag{1.1}$$

with $x \in (0, 1)$ and $rand(0, \sigma)$ denoting a normally distributed function, outputting a random number of mean average zero and standard deviation σ. The addition of the randomizer adds a level of noise to the data. Noise in a dataset is defined as a random error of variance of a measured variable [2]. The amount of noise in a dataset is of pivotal importance when teaching a model to learn a dataset. Excessive noise in training data exponentially increases the difficulty in accurately predicting the underlying function using algorithmic models and is a fundamental issue in machine learning.

Figure 1: Plot of (2.1)

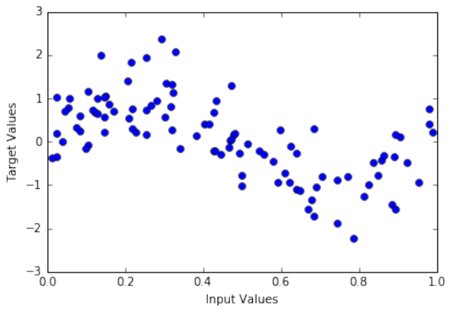


Figure 2 provides a visual demonstration of the effect of noise on a dataset. With (1.1) our underlying function, we can see that changing the standard deviation of the random gaussian input has a dramatic effect on the shape of the plot. Figure 2.1 is of standard deviation $\sigma = 0.15$. Here we can see a clearly visible sine function, with only minimal disturbance away from the trend line. By changing the standard deviation to $\sigma = 0.85$, Figure 2.2 shows a function that is heavily distorted to a point where it is impossible to visually detect our underlying function. For our $y(x)$ function (1.1) we will be using has a standard deviation $\sigma = 0.6$. as shown in Figure 1. This was chosen giving a reasonable compromise of disturbance from the true underlying function without distorting the data to a point of being unusable. See Kalapanidas et. al. [3] for more on noise sensitivity within data sets.

[1]A full code repository for all learning models and visualisations is available at: https://github.com/simonydbutt/Mathematics-Behind-Learning

Figure 2: Examples of Noise on Data

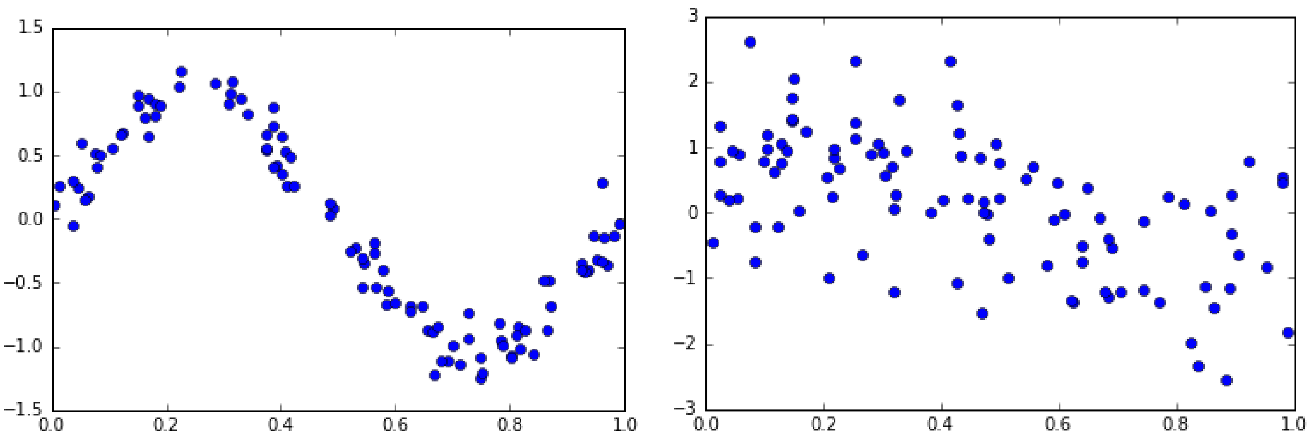
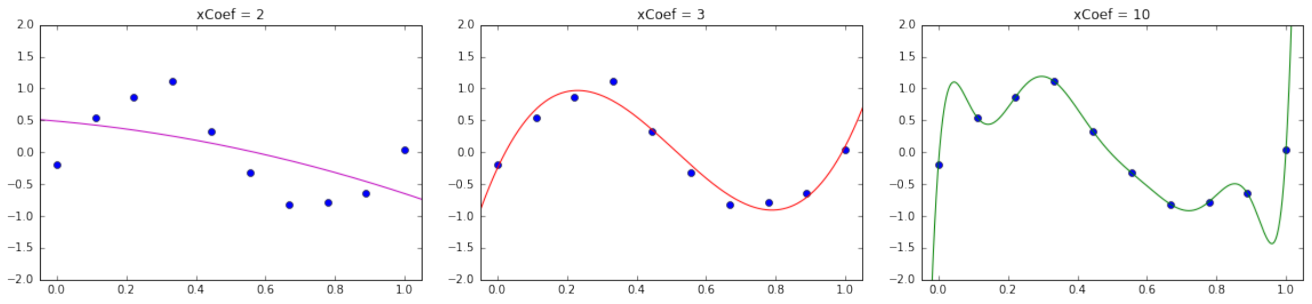


Figure 3: Changing x coefficients



1.2 Learner Function

Our learning algorithm will be a simple one dimensional polynomial regression

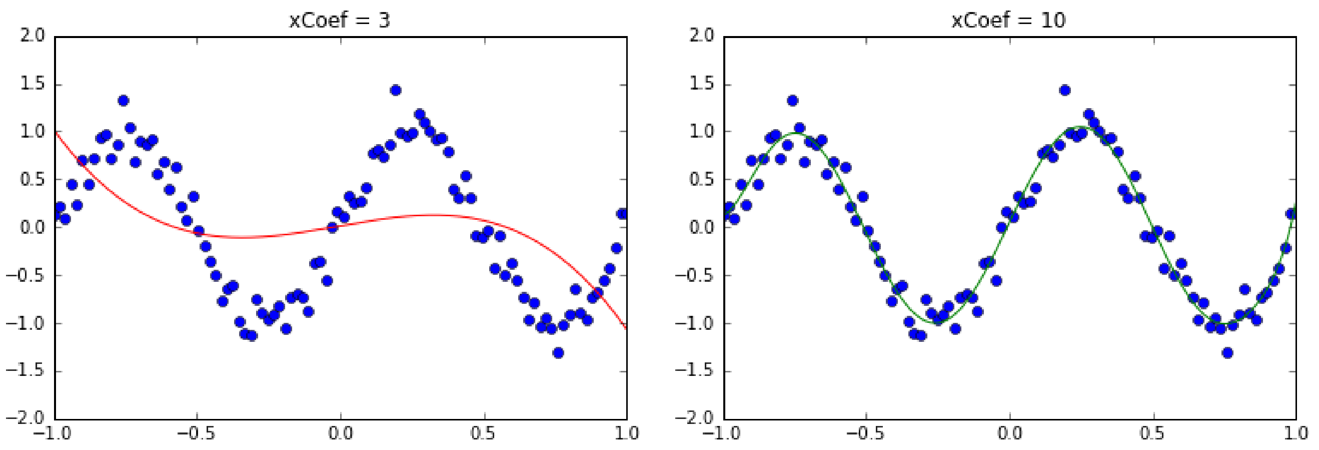
$$f(x, W) = w_0 x^0 + w_1 x^1 + \ldots + w_n x^n = \sum_{i=0}^{M} w_i x^i \tag{1.2}$$

where we will train the weights, $W = \{w_0, \ldots, w_n\}$ of the x^i coefficients to accurately predict the underlying function, $y(x)$ (1.1). The major consideration when using a polynomial regression is of which coefficient x^n to use. This shows the trade off between the optimal combination of accuracy on the training set to generalisability of the model.

Generalisation is a fundamental concept in all types of artificial intelligence and is defined as ”to take into account a large number of specific observations, then to extract and retain the important common features that characterize classes of these observations” [5]. This equates to how well a model will perform on data it has not seen before. Generalisation is therefore of vital importance in practical implementations of machine learning. Figure 3 gives a visual description of the accuracy/generalisation trade off made when choosing the maximum x coefficient. We can see that for $0 \leq x \leq 1$ our sine function can be modeled accurately using a polynomial coefficient x^3. By just reducing the maximum coefficient to x^2, the learning model (1.2) now can not hold enough information to map the desired output. This is the opposite for a maximum x^{10} coefficient. Here the model fits every point perfectly, but is fundamentally overfit and hence will generalise very poorly to previously unseen data.

The issue of choosing the maximum x coefficient becomes even more complicated when we take into account the size of our training set. Figure 4 gives a visual representation of this. Here

Figure 4: Model Fitting



the range has been extended to $x \in (-1,1)$ and the size of data increased to from the 10 data points, in Figure 4, to 100 data points. The x^3 coefficient which mapped almost perfectly to the previous range, now completely fails as the sine wave no longer resembles the cubic function. On the other hand, the increase in data size allows the x^{10} maximum coefficient polynomial to model the underlying function to a high level of accuracy. This brings us to a notable theorem that has dictated exploration within the domain of machine learning over the last twenty years. The No Free Lunch theorem states that for "any algorithm, any elevated performance over one class of problems is offset by performance over another class" [4]. See [5] or [6] to look at generalisation techniques in greater depth.

1.3 Loss Function

To determine the accuracy of our learner algorithm in correctly mapping our target function, we need a performance measure. This is the role of the loss, or cost, function in a machine learning algorithm. The loss function is a distance measurement which calculates the discrepancy between the true target value and the output the learning algorithm produces. This means that our goal in all machine learning problems is to minimise a model's loss function.

The two loss functions we will use for our polynomial regression are:

$$Loss(x) = \frac{1}{2N}\left(\sum_{n=1}^{N}\left(f(x_n,W)-y_n\right)^2\right) \tag{1.3}$$

$$LossWD(x) = \frac{1}{2N}\left(\sum_{n=1}^{N}\left(f(x_n,W)-y_n\right)^2\right) + \frac{\lambda}{2}||W||^2 \tag{1.4}$$

Both (1.3) and (1.4) are variants of mean squared error functions [7]. LossWD (1.4) differs by having the addition of a weight decay element. Weight decay [8] is a regularisation technique, used to reduce overfitting by penalising large weight coefficients. The motivation for weight decay in our problem is that the dampening of extreme weight values will allow for use of higher order x^n coefficients while mitigating the risk to generalisation. The hyperparameter, λ controls the extent of weight decay, with $\lambda = 0$ essentially turning (1.4) into (1.3). Regularisation is typically a penalty on the complexity of our learning model f, such as restrictions for smoothness or weights on the vector space norm [9]. This is an important topic in both machine learning and statistics, being one of the major techniques used to combat the accuracy/generalisation trade-off described earlier.

A second regularisation technique I will look at later in this thesis is early stopping. See Hennig et. al. [10] for more on the design of loss functions and Bickel et. al. [11] for more on regularisation techniques in computational learning.

1.4 Optimisation

The final component needed to complete our machine learning algorithm is a process to alter the weightings of each x coefficient. As our loss function $L(x, W) \to 0$, the learning algorithm $f(x_i) \to y_i$. This turns the problem of training the weights into one of optimisation:

$$\begin{aligned}\arg\min_{w\in W} L(x,W) &= \arg\min_{w\in W}\left(\frac{1}{2N}\left(\sum_{i=1}^{N}\left(f(x_n,W)-y_n\right)^2\right)+\frac{\lambda}{2}||W||^2\right)\\ &= \arg\min_{w\in W}\left(\frac{1}{2N}\left(\sum_{n=1}^{N}\left(\sum_{j=0}^{M} w_i x_n^i - y_n\right)^2\right)+\frac{\lambda}{2}||W||^2\right)\end{aligned} \tag{1.5}$$

Optimisation problems can usually be solved in one of two ways: analytically or computationally. Analytical methods [13] involve solving the minimisation equation to find the true global minimum. While this is not too difficult an exercise for our very simple polynomial regression loss function, most machine learning loss functions are far more complicated. This leads to situations where either it is implausible to find the true solution or far too computationally expensive. For these reasons our focus for the remainder of the thesis will be on iterative computation optimisation methods.

To minimise a function f, Figure 5, we need to find the point such that

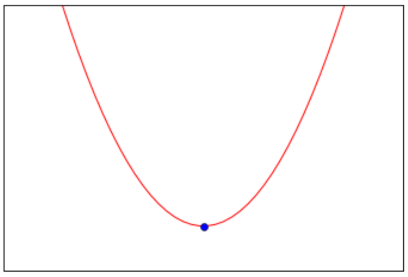
$$f(a^*) < f(a_j) \quad \text{s.t.} \quad j \in (1, n)$$

From calculus, we know that to locate a minimum a_j on the curve f we need the two following conditions to hold:

$$\frac{d}{da}f(a_j) = 0 \quad \text{and} \quad \frac{d^2}{da^2}f(a_j) > 0$$

In a convex problem, this point a_j will be equal to the global minimum a^*. This is then easily solved for using analytical methods, such as Newton Raphson or conjugate gradients [14]. Unfortunately loss functions are rarely this simple and often pose very complicated non-linear optimisation problems. In the non-linear case, our aim is to find

Figure 5: $f(a) = 2a^2$



$$a_{j^*} = \arg\min_{n\in j_1,\dots,j_m} f(a_n) \tag{1.6}$$

1.4.1 Gradient Descent

One of the most important algorithms in the development of machine learning is gradient descent [12]. This is an iterative scheme which uses the gradient of a function f at the point a^n to direct a step downwards:

$$a^{(n+1)} = a^{(n)} - \alpha\frac{d}{da}f(a^n) \tag{1.7}$$

Therefore as $n \to \infty$, $a^n \to a^*$ where a^* is the global minimum for weight a.

To show an example of gradient descent in action, lets take a simple function $f(a) = 2a^2$ (Figure 5). Here the gradient is:

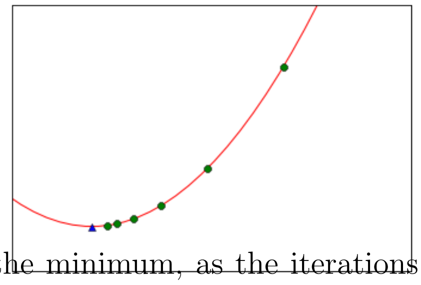
$$\frac{d}{da}2a^2 = 4a$$

giving a gradient descent update of:

$$a^{(n+1)} = a^{(n)} - \alpha 4a^{(n)} = a^{(a)}(1-4\alpha)$$

Starting at an arbitrary point $a = 1$, using a step size $\alpha = 0.1$ and running five iterations gives values (1.8) and plots to Figure 6. As we can clearly see, the algorithm gives values closer and closer to the minimum, as the iterations increase.

Figure 6: Gradient Descent



$$a^0 = 1,\ a^1 = 0.6,\ a^2 = 0.36,\ a^3 = 0.216,\ a^4 = 0.1296,\ a^5 = 0.07776 \tag{1.8}$$

With gradient descent as our optimisation algorithm, to train the weights of $f(x, W)$ (1.2), we need to find the derivative for our loss function.

We can find the derivative of our loss function (1.3) using equation manipulation and the chain rule

$$\frac{\partial}{\partial w_k}Loss(x,W) = \frac{\partial}{\partial w_k}\Big(\frac{1}{2N}\sum_{n=1}^{N}\big(f(x_n,W)-y_n\big)^2\Big)$$
$$= \frac{\partial}{\partial w_k}\Big(\frac{1}{2N}\sum_{n=1}^{N}\big(\sum_{i=0}^{M}w_ix_n^i - y_n\big)^2\Big) = \frac{1}{2N}\sum_{n=1}^{N}\Big(\frac{\partial}{\partial w_k}\big(\sum_{i=0}^{M}w_ix_n^i - y_n\big)^2\Big)$$
$$= \frac{1}{2N}\sum_{n=1}^{N}\Big(2\Big(\sum_{i=0}^{M}w_ix_n^i - y_n\Big)\frac{\partial}{\partial w_k}\Big(\sum_{i=0}^{n}w_ix_n^i - y_n\Big)\Big)$$
$$= \frac{2}{2N}\sum_{n=1}^{N}\Big(\Big(\sum_{i=0}^{M}w_ix_n^i - y_n\Big)x_n^k\Big)$$
$$= \frac{1}{N}\sum_{n=1}^{N}\Big(x_n^k\big(f(x_n,W)-y_n\big)^2\Big) \tag{1.9}$$

With this result (1.9), the gradient descent algorithm for our polynomial learner becomes:

$$w_k^{(j+1)} = w_k^{(j)} - \frac{\alpha}{N}\sum_{n=1}^{N}\Big(x_n^k\big(f(x_n,W)-y_n\big)^2\Big) \tag{1.10}$$

1.5 Training

Armed with our polynomial regression model, loss function and optimisation method; the last component is the training process of the learning algorithm. We do this by iteratively performing the gradient descent optimisation over a predetermined training dataset. A single iteration while training is a pass through the complete sample dataset.

For the initial training of our polynomial regression, we will use the model $\sum_{i=0}^{9} w_ix^i$, a learning rate $\alpha = 0.001$ and weight decay coefficient $\lambda = 10^{-5}$. We will initialise the weights w, as random numbers between -1 and 1. Below is a table showing the effect training had on the error of the model as the weights learn from the iterative gradient descent process.

Iterations	5	10	50	100	500	1000	5000	10000	50000
Error	0.3643	0.3122	0.2226	0.2142	0.2040	0.1979	0.1719	0.1594	0.1521
Noise RM	0.1724	0.1196	0.0466	0.0427	0.0324	0.0266	0.0089	0.0032	0.0056

The bottom row of the table is the error of the model at the n^{th} iteration against (1.1) but without the randomising element. For our specific underlying function (1.1), the $NoiseRM$ error rate is calculated as

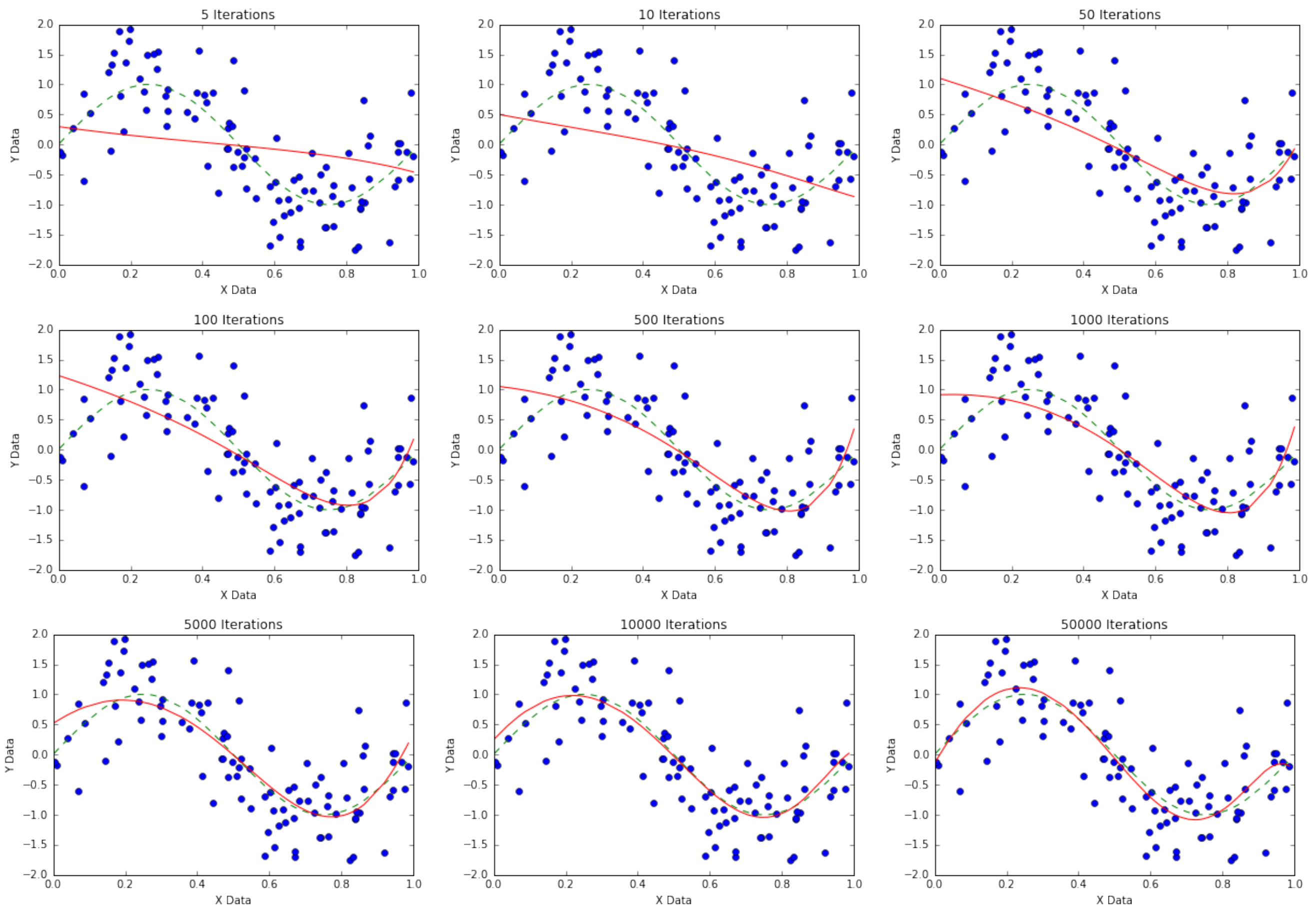
$$NoiseRM = \frac{1}{2N}\Big(\sum_{n=1}^{N}\Big(\sum_{i=0}^{9} w_i x^i - \sin(2\pi x)\Big)^2\Big) \tag{1.11}$$

As this measures the underlying function without the effect of noise on the dataset, it is essentially a measurement of the generalisation of our learning model.

From our results, we can see that at every iteration level, the loss function error rate reduces as the training iterations increase. This is the process of the weights being tuned to fit the training data. A visual representation of the training process is given by Figure 7. We can see that the initial two points show almost linear curves, with no specificity to the underlying function. At 50 iterations, we can clearly see the curve fitting between $x \in (0.8, 1)$. As the convergence for the i^{th} weight is given as $O(x^i)$, this means that the higher order weights will converge much quicker than the lower order coefficients. This manifests as the later sections of the curve fitting before the earlier sections. By 1000 iterations, the later half, $x \in (0.5, 1)$, of the curve fits almost perfectly to the underlying function. In addition, the first half is also starting to resemble the sine wave it is attempting to map. By 10000 iterations, our weight optimisation is practically complete. The normalised mean squared error is now 20% less than the initial error rate and with the effect of noise removed, the curve is within 0.56% of the true underlying curve.

The concepts of generalisation and overfitting can be shown when analysing the effect of training between 10000 and 50000 iterations. Between these two points, the loss function error decreases, but the error rate with noise removed increases. This signifies that the model's weighting is now overfit. The model has started to map to the random noise variables instead of the underlying function. We can view this by analysing the graphs at these two points. The plotted line (red), actually fits closer to the true underlying variable line (dashed green) at 10000 iterations, compared to at 50000. This signifies a loss of generalisability, manifesting itself as a poorer result when tested on an unseen dataset.

Figure 7: Training the Polynomial Regression

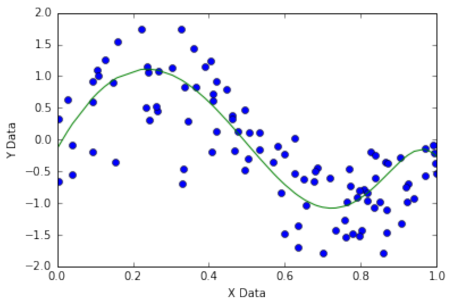


1.6 Results

Now that we have our complete trained polynomial regression algorithm, I will now analyse the performance of our learning model on a number of previously unseen datasets.

The first function analysed is a new set of points on the same initial underlying function (1.1). Using a test set the same size as our training set at 100 data entries, figure 8 visually shows the performance of our learning algorithm. The error on this data set is 0.1398, giving a more accurate mapping of the unseen test than even obtained when analysing our training set. To validate this result, I ran this same model on 1000 different variants of the same test set. This gave a mean test set error of 0.1306 with a standard deviation $\sigma = 0.01902$.

Figure 8: Plot of Test Data



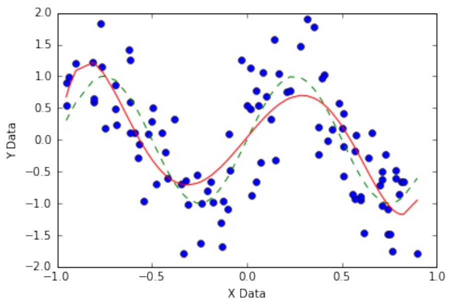
One important concept in the testing of a model's practicality is the generalisation, or out of sample, error [15] between the training and test set

$$GeneralizationError = |TestError - TrainingError| \tag{1.12}$$

The smaller the generalisation between training and test error, the more practical a model is at mapping unseen data. Here a generalisation error of 0.0262 was obtained, with standard deviation $\sigma = 0.0151$

To test the extensibility of our polynomial learner, we will now look at two test cases where we analyse a new underlying function. In the first we extend the initial equation's (1.1) boundaries to $x \in (-1, 1)$. This creates a curve closer resembling the sine wave (figure 9), in comparison to the previous cubic resemblance (figure 8). This creates an underlying function that requires a far more complicated mapping than the previous example. One observation seen when using a more complicated underlying function, is the increase in computational expensive when training. This manifested as the model requiring the full 50,000 iterations to optimise. We again ran our model on 1000 unseen test sets. This gave a mean error of 0.1656 with standard deviation $\sigma = 0.0279$ and generalisation error 0.0443 with standard deviation $\sigma = 0.0240$. We can see that as the underlying function become more complicated, the error and standard deviation rates for both the loss function and generalisation error increases. We can consider that this test case begins to show the limitations of the polynomial regression model.

Figure 9: $y(x) = sin(2\pi x)$

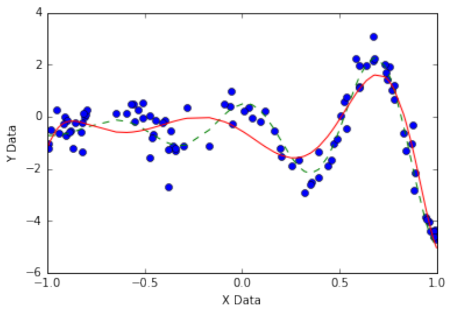


The final test case will attempt to expand on the hypothesis that our polynomial regression model shows significant limitations when attempting to map more complicated functions. Our new underlying function is

$$y(x) = e^{\frac{3}{2}x} cos(3\pi x) \tag{1.13}$$

The addition of an exponential within (1.13) creates cosine wave which expands as x increases. The complexity this adds was evident during the training process of our learner. Figure 10 shows

Figure 10: $y(x) = e^{\frac{3}{2}x} cos(3\pi x)$



our model after 100,000 iterations of gradient descent training.
We can clearly see the for $x \in (0.5, 1)$, our algorithm maps the function adequately. Between $x \in (-1, 0.5)$ though, the polynomial regression fails in modeling the cosine wave. Analysing over 1000 tests sets, we are given a mean error of 0.4710 and standard deviation $\sigma = 0.0571$. The generalisation error averaged at 0.1724 with a standard deviation $\sigma = 0.0571$. The huge increase in both loss and generalisation error gives further evidence that as the underlying function increases in complexity, the polynomial learner algorithm cannot hold the information required to successfully model it.

2 Radial Basis Function Network

2.1 Kernel Methods

In the last section, we uncovered some fundamental limitations of our polynomial learner algorithm. Now lets take another look and dive deeper into the making of our polynomial regression (1.2), with our motivation being to find a learning which can be used to model more complicated problems.

So far, we have only dealt with (1.2) as a regression model in parametric form [16]. This means that the training data is used to train the model but only the weightings are used when analysing the test set. There exists a second type of model, such that the training data is used for both the training but also in some form in the testing of the model, alongside the previously optimised weightings. The style of non-parametric schemes we will be looking at are called kernel methods.

We can introduce kernel methods by showing how (1.2) can be dual represented in a kernel method form along with it's original parametric structure. Firstly, lets recap our polynomial learner loss function (1.4):

$$LossWD = \frac{1}{2N}\Big(\sum_{n=1}^{N}\big(\sum_{i=0}^{M} w_i x_n^i - y_n\big)^2\Big) + \frac{\lambda}{2}||W||^2 \tag{1.4}$$

To put our loss function into a suitable form, we need to convert the structure from a polynomial regression to that of linear regression form. This can be done by

$$\mathrm{x}_n = \{x_n^0, x_n^1, ...,\} \tag{2.1}$$

Then

$$\mathrm{w} = \{w_0, w_1, ..., w_M\}$$

In this format, we can use the dual representation method from [9]. First, we define the activation function as a linear identity mapping

$$\mathrm{x} = \phi(\mathrm{x}) \tag{2.2}$$

With these, we can write our loss function as:

$$LossWD(\mathrm{w}) = \frac{1}{2N}\Big(\sum_{n=1}^{N}\big(\mathrm{w}^T\phi(\mathrm{x}_n) - y_n\big)^2\Big) + \frac{\lambda}{2}\mathrm{w}^T\mathrm{w} \tag{2.3}$$

Now taking the gradient of w at 0:

$$0 = \frac{1}{2N}\sum_{n=1}^{N} 2\phi(\mathrm{x}_n)\big(\mathrm{w}^T\phi(\mathrm{x}_n) - y_n\big) + \frac{\lambda}{2}(2\mathrm{w})$$

Then

$$\mathrm{w} = \frac{-1}{\lambda N}\sum_{n=1}^{N}\phi(\mathrm{x}_n)\big(\mathrm{w}^T\phi(\mathrm{x}_n) - y_n\big) \tag{2.4}$$

$$= \sum_{n=1}^{N} a_n\phi(\mathrm{x}_n) = \Phi^T\boldsymbol{a} \tag{2.5}$$

Where Φ is a design matrix, such that the n^{th} row is given by $\phi(\mathrm{x}_n)^T$ and $\boldsymbol{a}$ is an N sized vector such that:

$$a_n = -\frac{-1}{\lambda N}(\mathrm{w}^T\phi(\mathrm{x}_n) - y_n) \tag{2.6}$$

With $\boldsymbol{a}$ and Φ, we can now define our dual representation and substitute out w from our loss function

$$\begin{aligned} LossWD(a) &= \frac{1}{2N}\big(\Phi\Phi\boldsymbol{a} - \boldsymbol{y}\big)^2 + \frac{\lambda}{2}\boldsymbol{a}^T\Phi\Phi^T\boldsymbol{a} \\ &= \frac{1}{2N}\boldsymbol{a}^T\Phi\Phi^T\Phi\Phi^T\boldsymbol{a} - \boldsymbol{a}^T\Phi\Phi^T\boldsymbol{y} + \frac{1}{2}\boldsymbol{y}^T\boldsymbol{y} + \frac{\lambda}{2}\boldsymbol{a}^T\Phi\Phi^T\boldsymbol{a} \end{aligned} \tag{2.7}$$

where $\boldsymbol{y} = (y_1, ..., y_N)^T$. Now we can define the kernel matrix $\boldsymbol{K} = \Phi\Phi^T$, an NxN matrix with

$$K_{nm} = \phi(\mathrm{x}_n)^T\phi(\mathrm{x}_m) = k(\mathrm{x}_n, \mathrm{x}_m) \tag{2.8}$$

such that k is the kernel function defined as

$$k(\mathrm{x},\mathrm{x}') = \phi(\mathrm{x})\phi(\mathrm{x}') \tag{2.9}$$

Our loss function can now be written in terms of the kernel function

$$LossWD(a) = \frac{1}{2N}\boldsymbol{a}^T KK\boldsymbol{a} - \boldsymbol{a}^T K\boldsymbol{t} + \frac{1}{2}\boldsymbol{t}^T\boldsymbol{t} + \frac{\lambda}{2}\boldsymbol{a}^T K\boldsymbol{a} \tag{2.10}$$

Now using (2.6) and (2.5), we can solve for $\boldsymbol{a}$

$$\begin{aligned} a_n &= -\frac{1}{\lambda N}\left((\Phi^T\boldsymbol{a})^T\phi(x_n) - y_n\right) \\ &= -\frac{1}{\lambda N}\boldsymbol{a}^T\Phi\phi(\mathrm{x}_n) + \frac{1}{\lambda N}y_n \end{aligned} \tag{2.11}$$

Looking at the complete case as of (2.11)

$$\begin{aligned} &\boldsymbol{a} = -\frac{1}{\lambda N}\boldsymbol{a}^T\Phi\Phi^T + \frac{1}{\lambda N}\boldsymbol{y} \\ &\boldsymbol{a} + \frac{1}{\lambda N}\boldsymbol{a}^T k = \frac{1}{\lambda N}\boldsymbol{y} \\ &\lambda N\boldsymbol{a} + \boldsymbol{a}^T k = (\lambda NI + k)\boldsymbol{a} = \boldsymbol{y} \end{aligned}$$

Giving

$$\boldsymbol{a} = (\lambda N\boldsymbol{I} + k)^{-1}\boldsymbol{y} \tag{2.12}$$

Now using (2.12), we can rebuild our learning function as a kernel method

$$f(\mathrm{x}) = \mathrm{w}^T\phi\mathrm{x} = \boldsymbol{a}^T\Phi\phi(x) = k(\mathrm{x})^T(K + \lambda N\boldsymbol{I})\boldsymbol{y} \tag{2.13}$$

With our knowledge of kernel methods, we can start to look towards solving our classification problems using this style of learning algorithm.

2.2 The Iris Dataset

The classification type problem has been a fundamental problem since the early days of machine learning research [17]. We are given an input $\mathrm{x} = \{x_1, ..., x_n\}$ and using these variables, our learning algorithm places the data point into one of m different categories, or classes.

The Iris dataset [18] is a small classification problem, which we will use as the control problem in our exploration of RBF networks. Our goal is to predict the subspecies of iris flower from four four input data points: sepal length, sepal width, petal length and petal width. Using these, our algorithm will place each x value into one of three classes: Setosa, Versicolour, and Virginica. This manifests itself in data form as a four dimensional row vector input and an integer between zero and two as output. The size of the dataset is very small, at only 150 entries. These we will split into a training set of 100 entries and test set of 50. To look into classification problems in greater depth, see Ning [17] or Clancey [20].

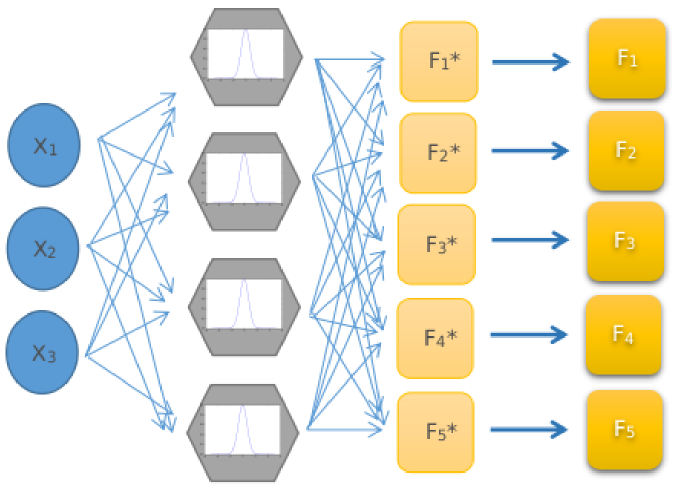
2.3 Radial Basis Function

The pivotal concept we will use in our classification examples are that of kernel based learning networks. There are two main types of networks used in learning models: neural and radial basis. Neural networks [19] are an extension of a parametric model, scaled to hold a greater amount of information than the basic regression model we previously looked at. Radial basis networks use kernel methods based on a nonlinear kernel function for prediction. It is this type that we will be focusing on for the remainder of this paper.

Figure 11 shows the structure of the network we will be using for the remainder of this thesis. The formula for the full radial basis network is

$$F(x, W) = \sum_{n=1}^{N} w_i\phi_i(x) \tag{2.14}$$

Figure 11: Radial Basis Function Network



where ϕ is our radial basis function (henceforth RBF) and N the number of RBF neurons in our network. The RBF acts as the model's kernel (or activation) function. A linear x value is inputed and returned is the radial distance from the RBF's center, such that

$$\phi_j(x) = h(||x - u_j||) \tag{2.15}$$

where ϕ is the j^{th} RBF neuron, x our input variable and center μ_j [9].

There exist a variety of RBF's we can use for the h function. Two in particular are used in the vast majority of situations. The first (Figure 12.1) is the gaussian RBF, defined as

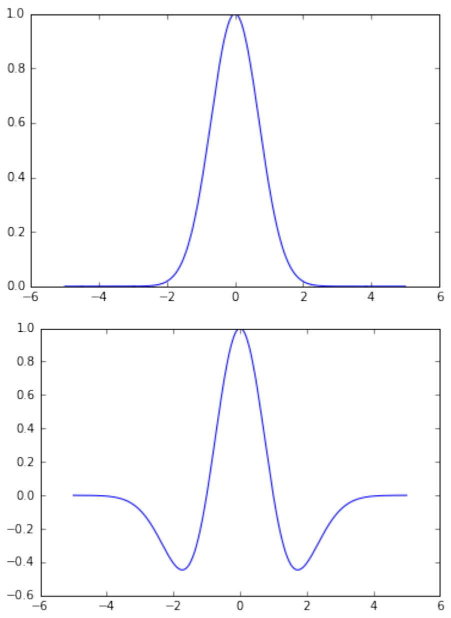
$$\phi_j^{Gauss}(x) = e^{-||x-\mu_j||^2} \tag{2.16}$$

The second commonly used activation function is the Ricker Wavelet (henceforth RW) function (Figure 12.2). This is defined as

$$\phi_j^{RW}(x) = (1 - ||x - \mu_j||^2)e^{\frac{-||x-\mu_j||^2}{2}} \tag{2.17}$$

The shape of the RW function is such that the points greatest penalised are ones approximately two standard deviations away from the center value. The motivation behind this is to allow our optimisation algorithm to quicker traverse areas of high error loss values. For more information on specific radial basis activation functions, see Blu [21].

Figure 12: Radial Basis Functions



2.4 Finding the Center Values

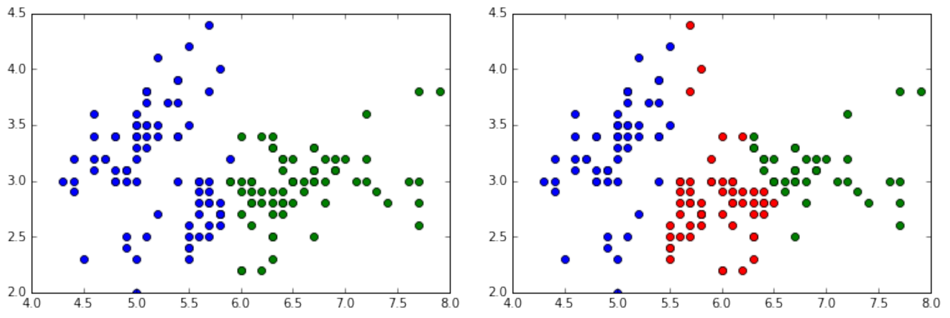
The center value μ of the RBF neuron is an important part of the radial activation function. There are two primary ways to calculate the center values [22]. The first is to randomly assign the value from either a single or a number of training set samples. The single example case is defined as

$$\mu_j = x_j^n \quad \text{for } rand(n) \in \{1, 2, ...N\} \tag{2.18}$$

while the multiple sample case gives the equation

$$\mu_j = \frac{1}{m}\sum_i^m x_j^i \quad \text{for } rand(i) \in \{1, 2, ...N\} \tag{2.19}$$

Figure 13: K Means Clustering on Iris Data



The second way to find our μ value is to calculate it using the data provided to us. Throughout the thesis so far, all problems have taken the form of being given an input data point x_i and from this, we aim to predict an output point y_i. This style of machine learning is labeled supervised learning and is defined as the "task of inferring a function from labeled training data" [23]. To find the center values of our RBF neurons, we have a surplus of input variables, but no target centers of which to use. This means that we have to train our model in an unsupervised form [24].

2.4.1 K-Means Clustering

The aim of each RBF neuron is to contain a specific amount of information which can be used to differentiate between output classes. This is done by giving all input variables which are similar to the center a high ϕ value, while input variables far away from this center point are given small values. Therefore, intuitively we can find our μ_j points by clustering our input variables into K number of clusters.

The K-Means clustering algorithm is a two step process. Firstly, we need to initialise our algorithm by randomly placing all our input variables into one of K different groups, or clusters.

$$\forall x^n \in X : x^n \rightarrow rand(C^j) \text{ s.t. } C^j \in \{C^1, C^2, ...C^K\}$$

Once all x values are initially placed randomly into a cluster, we can find the mean value of that specific cluster. To calculate our center value

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} x^i \quad \forall x^i \in C^j \tag{2.20}$$

With the initialisation step completed, we can begin to train our clustering algorithm. This involves finding the euclidean distance between each x_i value and the cluster's center values. The x_i value is then transfered into the cluster to whose center value has the smallest euclidean distance from it.

$$\{x^n \rightarrow C^j \ : \ ||x^n - \mu_j|| \leq ||x^n - \mu_i||, \ \forall i \in \{1, ..., K\}\} \tag{2.21}$$

Once we have run through all x values, we revise the center values μ_j (2.20). By iteratively performing the equations (2.21) and (2.20), we will eventually achieve clusters to which all x_i values are in the cluster most appropriate.

Figure 13 shows a example of our K-Means clustering algorithm on the Iris dataset. To visually display the algorithm, I have only used the initial two variables of the input vector, x. Figure 13.1 shows K-means with two clusters. We can clearly see a bisection of the x values at approximately the line $x = 5.7$. When performing the K-Means clustering algorithm on $K = 3$, we can clearly see that the data congregates around three visible clusters. For use in our RBF network, it is not the clusters themselves that are useful but the final center values μ_j. To extend knowledge of K-Means beyond the boundaries of this thesis, see Hartigan [25] or to expand on methods of RBF center selection see Orr [26].

2.5 RBF Loss Function

Along with our new learning function, the next component of our machine learning algorithm that needs replacing is the loss error function. The previous loss functions used for our polynomial learner case are no longer feasible for this style of problem. The intuitive classification error function is a binary loss rate, where we simply penalise every time the model predicts incorrectly. This would be defined as:

$$loss(x) = \frac{1}{N}\left(\sum_{i=1}^{N}\left(\begin{cases}0 & \text{if } f(x_i) = y_i \\ 1 & \text{else}\end{cases}\right)\right) \tag{2.22}$$

The second loss function, which is more applicable in terms of optimisation is the logLoss, or cross entropy loss [27], defined:

$$logLoss(x) = \frac{1}{N}\left(\sum_{i=1}^{N}\left(\begin{cases}log(f(x_i)) & \text{if } y_i = 1 \\ 1 - f(x_i) & \text{else}\end{cases}\right)\right) \tag{2.23}$$

The logLoss (2.23) function optimises faster the binary loss (2.31) due to not only improving upon the weights when our learning algorithm predicts wrong, but also when it correctly labels the target variable. This is done by enhancing the weights to further differentiate the correct answer from an incorrect one. While this possibly leads to overfitting if over-trained, it can be a very powerful tool when dealing with ambiguity within the input data.

We can write our conditional loss function as one function due to the binary nature of our y_i values. With a target value of either 0 or 1, we are given $log(1 - f(x_i)$ when the target value is correct, otherwise our cost value is $log(f(x_i))$. This allows you to write conditionals as one equation, making finding the gradient for optimisation far easier.

$$logLoss(x) = \frac{1}{N}\Big(\sum_{i=1}^{N} y_i log(f(x_i)) + (1 - y_i)log(1 - f(x_i)\Big) \tag{2.24}$$

For further reading on loss functions for classification problems, see Masnadi-shirazi [28].

2.6 Optimisation

When training our RBF network, our aim is to find the optimal weights for the outer layer of the network. This is completed by performing a similar gradient descent optimisation algorithm to the polynomial regression training process. To optimise for our new *logLoss* cost function we need to find the gradient of

$$\nabla_{w_i} logLoss(x) = \frac{d}{dw_i}\left(\frac{1}{N}\Big(\sum_{i=1}^{N} y_i log\big(\sum_{j=1}^{J} w_j\phi_j(x_i))\big) + (1 - y_i)log\big(1 - \sum_{j=1}^{J} w_j\phi_j(x_i)\big)\Big)\right) \tag{2.25}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(\frac{d}{dw_i} y_i log\big(\sum_{j=1}^{J} w_j\phi_j(x_i))\big) + \frac{d}{dw_i}(1 - y_i)log\big(1 - \sum_{j=1}^{J} w_j\phi_j(x_i)\big)\right) \tag{2.26}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(\frac{y_i}{\phi_j(x_i)} + \frac{1 - y_i}{1 - \phi_j(x_i)}\right) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{y_i(1 - \phi_j(x_i)) + \phi_j(x_i)(1 - y_i)}{\phi_j(x_i)(1 - \phi_j(x_i))}\right) \tag{2.27}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(\frac{y_i + \phi_j(x_i) - 2\phi_j(x_i)y_i)}{\phi_j(x_i)(1 - \phi_j(x_i))}\right) \tag{2.28}$$

Altogether we are given a gradient descent optimisation algorithm for our *logLoss* cost function (2.24) of

$$w_k^{(j+1)} = w_k^{(j)} - \frac{\alpha}{N}\sum_{n=1}^{N}\Big(\frac{y_i + \phi_j(x_i) - 2\phi_j(x_i)y_i)}{\phi_j(x_i)(1 - \phi_j(x_i))}\Big) \tag{2.29}$$

2.7 Data Preprocessing and Softmax

When we are given data from an outside source, one important initial task is data preprocessing [29]. This is the process of taking raw data and turning it into a suitable input variable to be used in our learning algorithm. In this case our preprocessing takes the form of data normalization. This problem shows an example of two of the most important data normalization techniques: feature scaling and one-to-many mapping.

The process of feature scaling [30] involves taking our base input data and converting it into an integer between two known points, usually either $x \in (0, 1)$ or $x \in (-1, 1)$, without losing the information giving by that data. In the case of our Iris dataset, we are given four input variables, all with their own scaling. The equation we use to normalise our input values is defined

$$\hat{x}_i^n = \frac{x_i^n - min(X_i)}{max(X_i) - min(X_i)} \tag{2.30}$$

where X_i is given all of the ith input variable of the complete training set. This will create a preprocessed input variable of $x_i^n \in (0, 1)\ \forall x^n \in X$ without losing any of the information given by the variables.

The second normalization technique is one-to-many mapping. This is the process of turning a variable of discrete nature into a number of variables which allow for easy processing by our learning algorithm. We will use this on our output variable, turning it from a number between zeros and two, to a vector of three dimensions with binary output, zero or one.

$$0 \rightarrow [1, 0, 0], \qquad 1 \rightarrow [0, 1, 0], \qquad 2 \rightarrow [0, 0, 1]$$

A final processing tool we use is one inside the network itself. Once we calculate our $f(x_i)$ values for the i^{th} training example, we need to covert this from a list of integers to a binary output of the same style as our target. We do this by a softmax [9] layer defined as

$$F^*(x_i) = \begin{cases} 1 & \text{if } f_j(x_i) = \arg\max F(x_i) \\ 0 & \text{else} \end{cases} \tag{2.31}$$

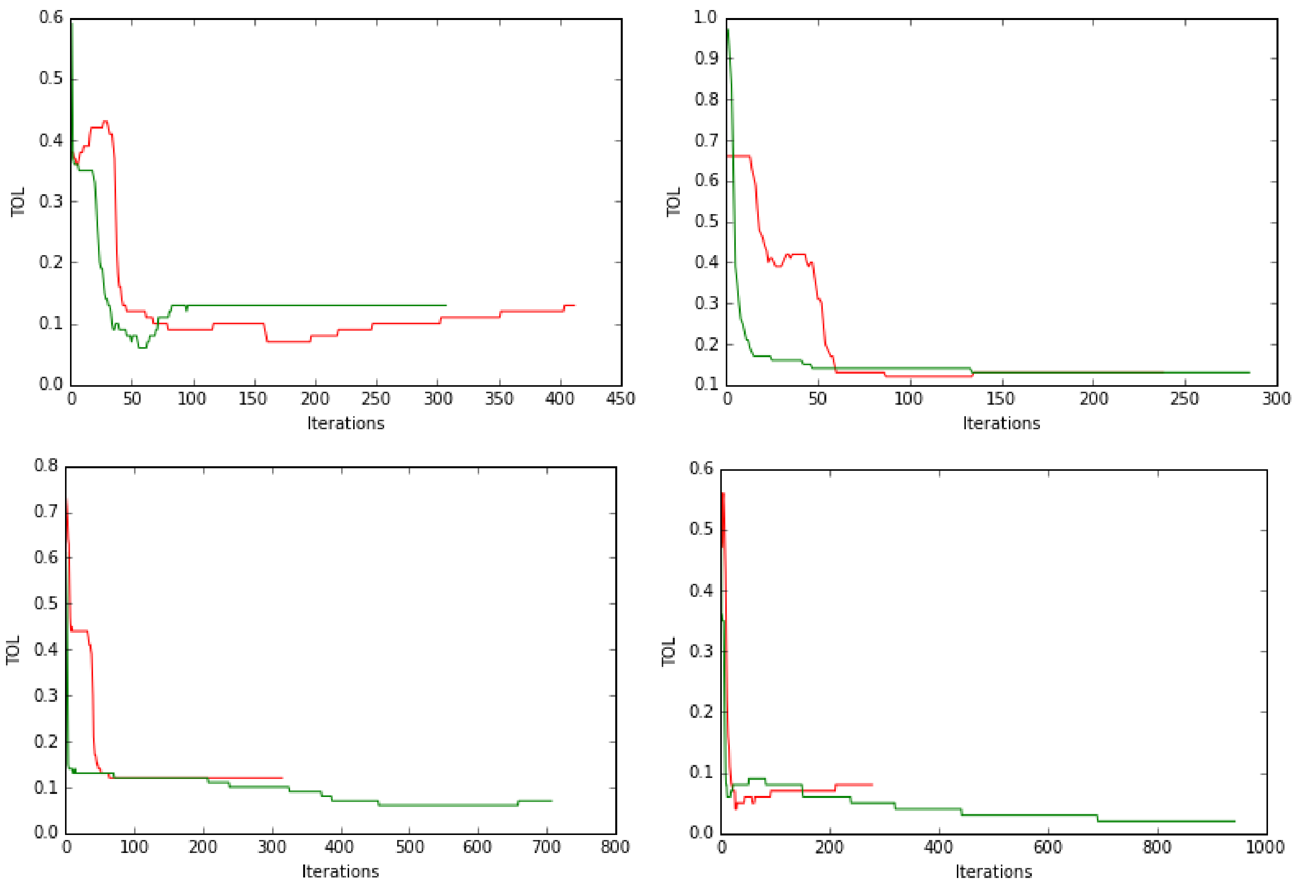
where $F(x) = \{f_1, ..., f_k\}$.

2.8 Iris Results

When training our RBF machine learning algorithm on the Iris dataset, I used a learning rate of 5x10^{-4} and maximum iteration of 1000 with an early stopping option. Early stopping [31] is a regularization technique in which the training process will stop before it reaches the maximum iterations if a specific condition is met. In the case of our RBF network, during the optimisation process, if the loss value hasn't improved within 250 iterations, our algorithm will stop and our weighting will be that of 250 iterations ago. This technique both potentially speeds up the training process and can reduce overfitting to the training set's noise. See [32] for an expansion on the early stopping technique.

I trained and analysed the results of the Iris problem on four different RBF neuron layer indexes: 3,5,8,15 neurons. Figure 14 shows the error in training against iterations for both the Gaussian and RW activation functions. Here the RW function is denoted in red, while Gaussian is plotted in green. The below tables show the results from this training on both the training set and also a test set of 50 unforeseen data points.

Figure 14: Iris Results



Gaussian Activation Function

RBF Length	Training TOL	Test TOL
3	0.87	0.84
5	0.88	0.88
8	0.88	0.94
15	0.96	0.96

Ricker Wavelet Activation Function

RBF Length	Training TOL	Test TOL
3	0.89	0.86
5	0.91	0.92
8	0.94	0.96
15	0.98	0.98

The results clearly shows three important findings. The first is that when we increase the size of the RBF layer, the accuracy of the RBF network increases. This is the case in the results of both activation functions tested. We can explain this as the model being able to 'understand' more, the larger the size of neuron layer.

This brings us to our second finding, that the greater the RBF neuron layer, the longer it takes for the outer weighted layer to train. This result is pretty intuitive in the sense that a more complex function needs a greater amount of time to optimise. It does leave us with a potential problem when it comes to extending this model to more complicated mappings. For the model to be able to hold the amount of information required to accurately map such complicated problems, we will need a far larger RBF neuron layer. This will increase the capability of the model, but at a large detriment in terms of computational expense.

The final point we can make from our results is in regards to the two activation functions. As we can see from the tables, the Ricker Wavelet activation function significantly improves the accuracy of the RBF network in comparison to the Gaussian counterpart. This is the case over all RBF lengths test, but with the exception of RBF length 3, but comes at the expense a greater time to optimisation. This leads us with an impasse in regards to which activation function to use for when approaching a more complicated problem.

3 Analysis - Stochastic Gradient Descent

Stochastic gradient descent (henceforth SGD) [33] is a variant of the base gradient descent algorithm. It was developed with the aim to combat gradient descent's two great weaknesses, slow convergence and becoming trapped in a high error point within the loss plane. It does this by using an important assumption, that the step direction given by gradient descent is only an expectation of the true downward direction [34]. This means that for us to perform an iteration, running through the whole dataset may not be necessary. If it is possible to gain the same knowledge from running through only a tenth or even a thousandth of the training data, then we will be able to iterate at a rate much quicker than previously possible. This becomes incredibly important when dealing with ever larger datasets, as training examples on modern data problems can run into the billions of data entries [35]. This causes a deterministic gradient descent optimiser to be largely inefficient and hence cause both very slow and prohibitively expensive computational run times. The equation for stochastic gradient descent is defined:

$$w_j^{n+1} = w_j^n - \alpha \nabla_{w_j} \sum_{i=1}^{\hat{M}} Loss(x^i, W) \tag{3.1}$$

for $x^i \in \hat{M} = \{a, ..., b\}$

The mini-batch $\hat{M}$, is derived by shuffling the training data and running through the data at rate alpha α until we reach the end of the set. We reshuffle our training data and repeat this process. A second way to keep the stochastic nature of this method, is to repeatedly shuffle the data at every iteration. When dealing with very small mini-batch sizes and large training sets, the process of data shuffling this can become computationally expensive. The stochastic nature of the mini-batch process, allows us to assume to gain the same knowledge from our small sample size as that of the whole training batch. For more on SGD, see Xu [36] or Marti [37].

3.1 Experiment 1. Mini-batch Size

The first experiment we will look at when analysing SGD is the effect altering the minibatch size has on both the training and mapping of a learning algorithm. To be able to evaluate this, we can no longer use iterations as our dependent variable against the loss function's error measurement. The obvious replacement here is to take the measurement temporal based instead. Measuring the time taken allows us to analyse any number of mini-batch sizes against the base full batch gradient descent algorithm.

In this experiment I used our polynomial regression algorithm, adapted to time dependent optimisation. The hyper parameters were set at learning rate $\alpha = 1$ and loss function weight decay $\lambda = 10^{-5}$. The four different batch sizes used were: 5, 10, 50, 100 along with a control of base deterministic gradient descent and analysed over two different training set sizes: 500, 5000.

I analysed the results, both in terms of TOL value produced from our loss function and as a percentage increase/decrease in comparison to the control deterministic method. I evaluated this for both a training and an unforeseen test set - over four different underlying functions. I measured the error over three different optimisation time periods: 20, 60, 180 seconds.

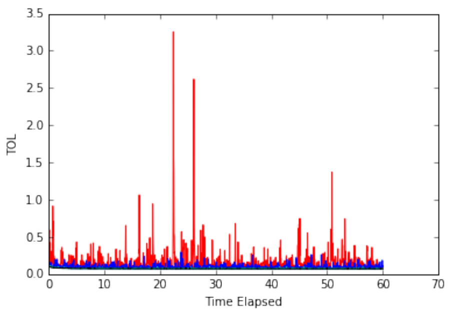
The underlying functions I used were

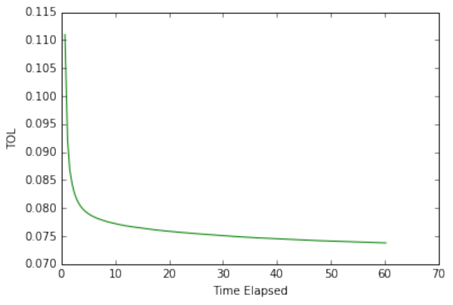
1.	$y(x) = sin(2\pi x)$	$x \in (0, 1)$
2.	$y(x) = sin(2\pi x)$	$x \in (-1, 1)$
3.	$y(x) = cos(3\pi(x-1))e^{2(x-1)}$	$x \in (0, 1)$
4.	$y(x) = cos(3\pi(x-1))e^{2(x-1)}$	$x \in (-1, 1)$

The results of these tests (Appendix A) show a number of important points. Firstly, on almost all occasions, all forms of stochastic gradient descent perform better than the comparative deterministic gradient descent control. This is due to the far greater number of iterations every SGD batch size had over the control. Intuitively, this also suggests that the smaller the batch size, the greater number of iterations and hence smallest TOL value.

In general, smaller batch sizes do result in smaller error values but seemingly only to a certain point. Between batch sizes of 10 and 50, this is clearly the case. In almost every test, there was a clear disparity between the two results. Between batch sizes of 5 and 10 on the other hand, our intuition fails. There are a number of cases where batch size 5 outperforms batch size 10 but the general rule is the opposite, with some cases of size 5 actually performing worse than the control deterministic optimisation. Where smaller batch sizes are vastly more accurate is over the 20 second optimisation. Over 180 or even 60 seconds, the larger batch sizes can run enough iterations to produce a respectable result.

Figure 15: Batch Size Results





The specific case of batch size 5 performing poorer than the control on certain occasions, brings us to figure 15.1. This plot shows a 60 second training period and subsequent TOL value measured at every second. The red line is batch size 5 while the blue is of batch size 10. All other batch sizes and control are also plotted but not visible due to the huge variance of the red plot. While the volatility in variance of SGD is one of it's most powerful features, a too small batch size can result in too large a volatility. This manifests as the optimisation algorithm very quickly traversing the loss value plane, but also missing out on the lowest points. For reference, 15.2 shows the same plot with only the deterministic gradient descent plotted.

One interesting point, consistent throughout the entire experiment but isn't easily explainable is the difference in iterations between batch sizes 5 and 10. In many cases, there difference is immaterial between the two results. The similarity between number of iterations over the same training period coupled with the reduction in volatility given by batch size 10 shows us that a batch size of 10 looks to give the ideal tradeoff between iterative speed and model volatility. Hence progressing to our classification problems our standard SGD batch size will be 10.

3.2 Experiment 2. SGD Varients

As shown in the above analysis, stochastic gradient descent algorithm is effective at speeding up the optimisation process. While the base SGD algorithm is very powerful and the volatility of using small mini-batches allow it to traverse high error points quickly, this same volatility can cause our optimiser to be too noisy a process to result in the optimal weightings. Therefore our motivation for this second experiment is to find a variant of SGD which benefits from the inherent stochastic nature but also optimises to the lowest error rates.

The first variant of SGD I will be testing contains an additional random Brownian element [38]. The motivation behind this method is for the additional non-gradient based variable to allow the weights to in effect jump out of the high error critical points. This will subsequently allow the algorithm to explore a larger amount of the loss function surface area, creating the possibility of finding a more optimum weighting. The formula for SGD with additional brownian motion is defined as:

$$w_j^{n+1} = w_j^n - \alpha \nabla_{w_j} \sum_{i=1}^{\hat{m}} Loss(x^i, W) + \sigma() \tag{3.2}$$

The second variant of SGD I will test is a SGD with additional momentum function. Momentum [39] changes the direction from being a variable of only the most recent gradients to that of a coupling of the latest gradient value plus an exponentially decaying variable of the previous gradients. The motivation behind this is twofold. Firstly, it can smooth the optimisation progression when faced with very noisy data. Secondly, the momentum variable can similarly help the algorithm move past high error local minimum areas. The formula for SGD with momentum is defined as:

$$v_i = \epsilon v_i - \frac{(1-\epsilon)}{\hat{m}} \nabla_{w_j} \sum_{j=1}^{\hat{m}} Loss(x^i, W) \tag{3.3}$$

Then

$$w_i^{n+1} = w_i^n + \alpha v_i \tag{3.4}$$

In addition to the above two variants of SGD, I will also include a SGD with high level of weight decay. The motivation is to show the effect of large regularisation in the training and modeling of a learning algorithm.

Following a similar procedure to that of the previous test, I will keep the learning rate hyper-parameter at $\alpha = 1$ along with optimal batch size found in the previous experiment of 10. The experiment will be run over the same four underlying functions, with different training set sizes: 500, 5000. The variant specific hyper parameters are weight decay constant $\lambda = 0.001$, Brownian motion $sigma = 0.1$ and momentum decay variable $\epsilon = 0.9$.

I analysed the results, both in terms of TOL value produced from our loss function and as a percentage increase/decrease in comparison to the control base SGD model. I again evaluated this over both the training and an unforeseen test set, on the four previous underlying functions. I kept the measurement times at: 20, 60, 180 seconds to record the error rates. The results are as follows.

The first major observation is in the scale of performance increase. There were clear anomalies but in general, the percentage increase in TOL value were much smaller than that of the previous test. This falls in line with general consensus that data and time to train are more important factors than the variant of optimisation algorithm [40]. In addition, another explanation to this is the exponential increase in difficulty required when getting closer to the optimum weighting and hence minimum loss value.

Variant	**Average Training %TOL**	**Average Test % TOL**
Weight Decay	-3.791254	-4.243317
Brownian	-0.286887	-0.330626
Momentum	1.498575	1.129092

The second important point can be easily shown via the above table. Averaging over the twenty-four scenarios (four different underlying functions, each tested over three time periods and two dataset sizes), only the momentum SGD variant outperformed the control base stochastic gradient descent optimisation. Before further analyse, it is prudent to state that the choice of hyperparameter could have potentially skewed these results. This is especially the case when looking at the weight decay (henceforth WD) variant. The WD variant performed especially poorly, very rarely achieving a value smaller than the control and usually resulting in a loss error rate far greater. The momentum with additional brownian noise overall performed worse than the control value but in a very different scenario than the WD variant. In this case, the variant performed on par or greater than the control in many of the first three underlying functions. It was the function $y(x) = cos(3\pi(x-1))e^{2(x-1)}$ for $x \in (-1, 1)$, where the brownian variant performed especially poorly, averaging -2.2385% over that function.

Another point which the table succinctly shows is the issue of overfitting when adding specificity to an underlying algorithm. By changing the variant of our stochastic gradient descent algorithm, we are adding our own biases into the function. This manifests itself in the results by our learning algorithm generalising poorer, on average, than the base SGD algorithm. This is the case in all variants tested, where the average training %TOL is far less than the loss value from an unseen test set.

Overall, the results show that while the base stochastic gradient descent algorithm is formidable in it's own right, the addition of a momentum component does improve the accuracy over both training and test sets. This does come with the drawback of reduced generalisation, but the advantage of an overall decrease in error rate surpasses this negative. Moving forward to our classification problems, the optimisation algorithm we will be using is stochastic gradient descent with momentum.

3.3 Conclusions

Optimiser Layer	**Training No. Identified**	**Training % Identified**	**Test No. Identified**	**Test % Identified**
Gradient Descent	87	87	45	90
SGD w. Momentum	94	94	49	98

The previous two experiments have resulted in an updated optimisation algorithm to use for our classification problems. To verify these results, I analysed our SGD with momentum at batch size 10 against our standard gradient descent optimiser on the iris dataset. I trained both models over 60 seconds, with the possibility of early stoppage.

Figure 16: Iris Dataset Results

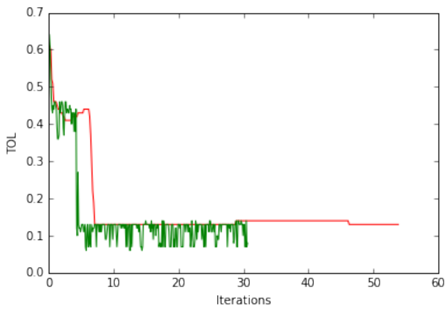


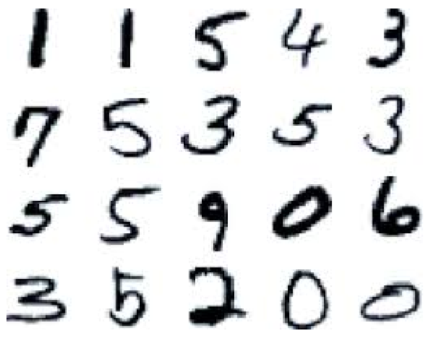
Figure 16 plots a graph of error TOL against time taken to optimisation for gradient descent (red) and SGD with momentum (green). We can see from the results that when extended to the iris dataset, the performance increase shown from the previous experiments are only heightened. Here the SGD variant outperformed gradient descent by 7% and 8% over the training and test sets. On the test set, SGD with momentum optimisation resulted in a model which only misidentified one data-point. In addition to a far superior final result, the SGD variant took 24 seconds less to optimise the weightings. Overall, this shows a increase in all possible performance measurements in the Iris problem.

4 MNIST Problem and moving forward

4.1 MNIST Dataset

We will now use the machine learning concepts discussed throughout this project to attempt to model a far more complicated benchmark problem. The problem in question will be the Mixed National Institute of Standards and Technology database (henceforth MNIST) [41]. MNIST is one of the most commonly used datasets in machine learning, especially in the testing of new models. This is due to it containing a number of features which makes it very difficult to gain high accuracy values. To score highly, learning models are required to have a significant level of generalisation, a feature very desirable in practical applications of machine learning.

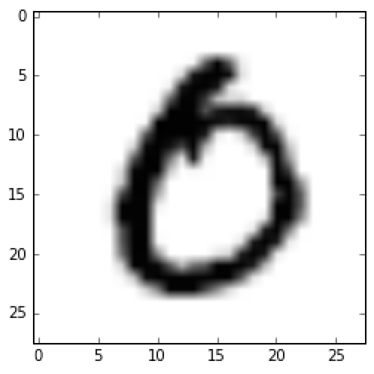
Figure 17: MNIST Problem



The data itself is a computer vision problem where we are given an image of a handwritten digit between zero and nine. Our aim is to build a model which can identify the number written down. Figure 17 gives an example of a selection of images we are given. Each image is 28x28 pixels, creating an input variable with 784 dimensions. This coupled with the training set size of 42000 data points, creates a dataset orders of magnitude larger than what we've previously seen.

The size of the data is only one of the difficulties faced when modeling this problem. The other major issue is in the data itself. The previous underlying function modeled (1.1), had a random variable added to create an amount of noise in the data. In the MNIST problem, the noise is in the nuances and habits of human handwriting. The images themselves were sourced from a mixture of American high school students and Census Bureau employees, altogether giving a diverse range of handwriting styles and henceforth input images. It doesn't help that a number of digits can look very similar to one another when not drawn with precision. Take figure 18 for example, it is easy to make arguments for this to be either a zero or a six. This creates many cases of ambiguity that our learning algorithm needs to be able to differentiate between.

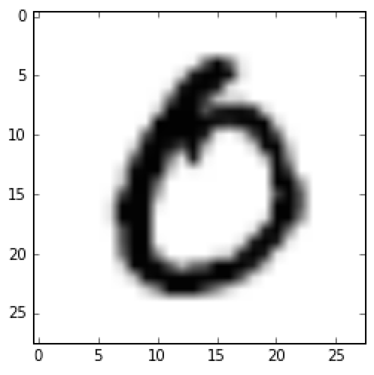
Figure 18: Ambiguous Digit



4.2 Data Preprocessing

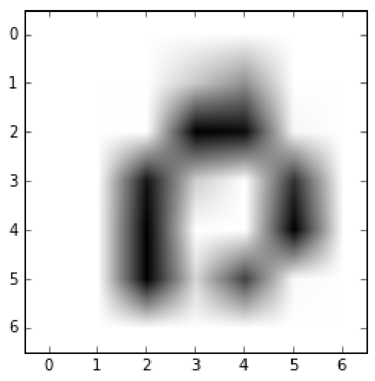
The first challenge faced when progressing from the Iris to MNIST computer vision problem is to process the input into a format suitable to our learning model. The major issue here is the increase from 4 to 784 input variables. This creates a potentially huge increase in computational expense. In addition, the input x exists between 0 and 255. The below formula shows the transformation we aim for from our preprocessing stage. $\{x_{n,m} \in (0,255) : n,m \in (1,28)\} \rightarrow \{x^*_{n^*,m^*} \in (0,1) : n^*, m^* \in (1,M)\}$ where $0 < M < 255$.

Figure 19: MNIST Dimension Reduction



To reduce our individual input value from $x \in (0, 255)$ to $x \in (0, 1)$, we simply use the data scaling algorithm 2.30 initially used in our iris dataset. To reduce the total number of input arguments per data-point, we can use a preprocessing technique called dimensionality reduction. Our aim is to reduce the number of overall input variables while losing the least amount of information. An intuitive way to do this is to divide the image into a grid of squares. We can then simply find the mean of these squares to give us our new input values. With initial dimensions of 28x28, the easiest dimension format to reduce to will be 7x7. Here each new input $x^*_{n^*,m^*}$ is the average of a 2x2 grid in $x_{n,m}$. Figures 19 and 20 gives a visual representation of the dimensionality reduction technique applied to our problem.

Figure 20: MNIST Dimension Reduction



4.3 Training and Results

When attempting to apply the same training procedure to the MNIST dataset, I very quickly discovered a problem in the computational expense. The loss function running through the entire training set at every iteration caused the whole optimisation procedure to slow to an unfeasible level. To get around this, I removed running the loss function at every iteration. In addition at the certain checkpoints during training, I ran the loss function over a random small percentage of the training data. This results in a slightly less accurate error value but removes a large computational expense.

Due to computational capabilities, I used a limited random sample of the complete MNIST dataset. I used a training set of 2000 data entries and test set of 1000. This dataset, while far less than the complete database, was large enough to train and analyse on, while small enough to keep computational expense at a reasonable level. I trained the RBF network at three neuron levels of 15, 40 and 100. The activation function used was gaussian (2.16) and the optimisation algorithm used was SGD with momentum and batch size 10.

An observation from training was that even with this vastly smaller training set, the time taken for our learning model to train was significant. On previous problems, a training time of 300 seconds would be enough to sufficiently train the learning algorithms to convergence. For this dataset, the training times required were at least an order of magnitude greater, especially when training the 100 neuron model.

Neuron Layer	**Time to Train**	**Training No. Identified**	**Training % Identified**	**Test No. Identified**	**Test % Identified**
15	600	1778	88.9	867	86.7
40	1800	1790	89.7	884	88.4
100	7200	1861	93.5	912	91.2

The results from our RBF model on the MNIST problem show a level of accuracy in the approximate 90% range, increasing as the size of the neuron layer increases. This can be explained by the greater complexity of the problem, hence requiring a greater amount of information to be stored at the neuron layer to accurately model the underlying function.

4.4 Moving Forward

In continuation of my project, to look at machine learning from a mathematics first viewpoint, there are a number of areas I would expand into. The first would be a greater expansion into kernel methods and designing a nonlinear kernel mapping function which can enhance the information retention process of the RBF neuron.

Another way we could expand this project would be to look into variable learning rates. This would be especially useful when extended to variants of SGD. By developing either a hierarchical additional brownian variable or an ϵ value in momentum which increases as the loss value decreases. This could increase the generalisation of these variants, making them more feasible optimisation algorithms for this problem.

A final way to extend this project would be by reducing the computational limitations aspect. This could be done by either looking into algorithm efficiency on a software level or by increasing the hardware capabilities. The hardware could be increased by either designing the model to run as a parallel process, for example by running one CPU to train the model and another to evaluate the loss value at certain weightings or by looking into the role of graphical processing units in the training of computational learning models.

A SGD batch size results tables

A.1 $y(x) = sin(2\pi x)$ for $x \in (0, 1)$

Dataset Size: 500

Training Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	423	0.374325	0
5	4719	0.349108	6.7366
10	4343	0.348341	6.9416
50	2547	0.355621	4.9969
100	1695	0.361050	3.5465

Test Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	423	0.372552	0
5	4719	0.356132	4.4075
10	4343	0.356444	4.3236
50	2547	0.356790	4.2307
100	1695	0.360054	3.3547

Batch Size	Iterations	TOL	% TOL
Grad Desc	1236	0.347833	0
5	14188	0.331312	4.7496
10	13168	0.329568	5.2510
50	7872	0.331864	4.5907
100	5115	0.335298	3.6035

Batch Size	Iterations	TOL	% TOL
Grad Desc	1236	0.341564	0
5	14188	0.320809	6.0762
10	13168	0.319402	6.4885
50	7872	0.322694	5.5244
100	5115	0.326619	4.3755

Batch Size	Iterations	TOL	% TOL
Grad Desc	3968	0.318240	0
5	45629	0.304405	4.3474
10	46549	0.304133	4.4328
50	27187	0.304145	4.4290
100	16296	0.304734	4.2439

Batch Size	Iterations	TOL	% TOL
Grad Desc	3968	0.343826	0
5	45629	0.340051	1.0979
10	46549	0.339608	1.2269
50	27187	0.339104	1.3734
100	16296	0.338198	1.6370

Dataset Size: 5000

Training Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	48	0.367811	0
5	586	0.348946	5.1290
10	583	0.346012	5.9267
50	539	0.346159	5.8866
100	501	0.346595	5.7682

Test Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	48	0.381297	0
5	586	0.357635	6.2056
10	583	0.355995	6.6356
50	539	0.356467	6.5121
100	501	0.357721	6.1831

Batch Size	Iterations	TOL	% TOL
Grad Desc	148	0.362981	0
5	1783	0.349822	3.6252
10	1764	0.344857	4.9932
50	1629	0.344567	5.0731
100	1486	0.345551	4.8019

Batch Size	Iterations	TOL	% TOL
Grad Desc	148	0.349098	0
5	1783	0.338264	3.1036
10	1764	0.329793	5.5300
50	1629	0.331701	4.9837
100	1486	0.332596	4.7272

Batch Size	Iterations	TOL	% TOL
Grad Desc	390	0.359000	0
5	4616	0.334949	6.6996
10	4693	0.331433	7.6791
50	4372	0.332731	7.3173
100	3943	0.333897	6.9927

Batch Size	Iterations	TOL	% TOL
Grad Desc	390	0.348379	0
5	4616	0.325815	6.4768
10	4693	0.323052	7.2699
50	4372	0.322632	7.3906
100	3943	0.324852	6.7534

A.2 $y(x) = sin(2\pi x)$ for $x \in (-1, 1)$

Dataset Size: 500

Training Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	414	0.426642	0
5	4726	0.385896	9.5504
10	4230	0.386524	9.4030
50	2537	0.396531	7.0577
100	1593	0.405040	5.0634

Test Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	414	0.324851	0
5	4726	0.303401	6.6029
10	4230	0.298098	8.2352
50	2537	0.310881	4.3002
100	1593	0.316241	2.6504

Batch Size	Iterations	TOL	% TOL
Grad Desc	1477	0.379454	0
5	16402	0.329338	13.2075
10	14998	0.328995	13.298
50	8840	0.339524	10.5229
100	5115	0.350886	7.5288

Batch Size	Iterations	TOL	% TOL
Grad Desc	1477	0.379997	0
5	16402	0.335585	11.6874
10	14998	0.339851	10.5648
50	8840	0.346376	8.8477
100	5115	0.355592	6.4223

Batch Size	Iterations	TOL	% TOL
Grad Desc	4200	0.348560	0
5	46490	0.316372	9.2345
10	42090	0.315830	9.3902
50	26825	0.316982	9.0595
100	17516	0.320339	8.0964

Batch Size	Iterations	TOL	% TOL
Grad Desc	4200	0.391976	0
5	46490	0.365238	6.8215
10	42090	0.365181	6.8359
50	26825	0.365962	6.6366
100	17516	0.368930	5.8795

Dataset Size: 5000

Training Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	47	0.498224	0
5	589	0.393040	21.115
10	575	0.393201	21.0826
50	533	0.390149	21.6951
100	494	0.390923	21.5398

Test Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	47	0.463517	0
5	589	0.368024	20.6019
10	575	0.373749	19.3668
50	533	0.365970	21.0448
100	494	0.366128	21.0109

Batch Size	Iterations	TOL	% TOL
Grad Desc	131	0.434686	0
5	1604	0.372137	14.3895
10	1664	0.372010	14.4187
50	1610	0.369073	15.0944
100	1414	0.370513	14.7631

Batch Size	Iterations	TOL	% TOL
Grad Desc	131	0.438229	0
5	1604	0.378448	13.6417
10	1664	0.378061	13.7300
50	1610	0.373994	14.6578
100	1414	0.375557	14.3012

Batch Size	Iterations	TOL	% TOL
Grad Desc	450	0.391529	0
5	5430	0.357624	8.6598
10	5342	0.353567	9.6959
50	4912	0.355481	9.2070
100	4510	0.361064	8.9849

Batch Size	Iterations	TOL	% TOL
Grad Desc	450	0.38815	0
5	5430	0.354788	8.5951
10	5342	0.352263	9.2457
50	4912	0.353272	8.9856
100	4510	0.354677	8.6238

A.3 $y(x) = cos(3\pi(x-1))e^{2(x-1)}$ for $x \in (0,1)$

Dataset Size: 500

Training Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	481	0.114491	0
5	5413	0.111972	2.2009
10	5030	0.107409	6.1861
50	3021	0.109321	4.5162
100	1928	0.110925	3.1148

Test Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	481	0.126104	0
5	5413	0.122202	3.0942
10	5030	0.118543	5.9960
50	3021	0.120369	4.5479
100	1928	0.122025	3.2345

Batch Size	Iterations	TOL	% TOL
Grad Desc	1515	0.112933	0
5	16831	0.100711	10.8227
10	15466	0.099245	12.1209
50	9095	0.100698	10.8347
100	6027	0.10316	8.6544

Batch Size	Iterations	TOL	% TOL
Grad Desc	1515	0.106044	0
5	16831	0.107981	-1.8262
10	15466	0.100254	5.4597
50	9095	0.099881	5.8116
100	6027	0.100536	5.1940

Batch Size	Iterations	TOL	% TOL
Grad Desc	4536	0.107032	0
5	50314	0.102926	3.8362
10	45853	0.102994	3.7727
50	26918	0.103331	3.4579
100	17714	0.103689	3.1234

Batch Size	Iterations	TOL	% TOL
Grad Desc	4536	0.109239	0
5	50314	0.103325	5.4142
10	45853	0.104325	4.4988
50	26918	0.104811	4.0531
100	17714	0.105203	3.6950

Dataset Size: 5000

Training Set

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	49	0.121211	0
5	595	0.112913	6.8461
10	599	0.112944	6.8207
50	548	0.112960	6.8074
100	488	0.113420	6.4275

Test Set

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	49	0.123537	0
5	595	0.115906	6.1772
10	599	0.115957	6.1361
50	548	0.115801	6.2619
100	488	0.116133	5.9935

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	149	0.118648	0
5	1855	0.111177	6.2970
10	1810	0.110697	6.7012
50	1678	0.110693	6.7044
100	1533	0.111009	6.4382

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	149	0.12627	0
5	1855	0.117469	6.9692
10	1810	0.117343	7.0690
50	1678	0.117281	7.1183
100	1533	0.117745	6.7510

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	456	0.114802	0
5	5588	0.107780	6.1172
10	5514	0.108302	5.6626
50	5090	0.108395	5.5815
100	4575	0.111076	5.2638

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	456	0.117452	0
5	5588	0.109773	6.5378
10	5514	0.110697	5.6626
50	5090	0.110631	5.8077
100	4575	0.111076	5.4285

A.4 $y(x) = cos(3\pi(x-1))e^{2(x-1)}$ for $x \in (-1, 1)$

Dataset Size: 500

Training Set

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	490	0.075062	0
5	5425	0.07334	2.2949
10	4872	0.072811	2.9998
50	2880	0.073106	2.6066
100	1961	0.07347	2.1221

Test Set

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	490	0.072949	0
5	5425	0.074275	-1.8182
10	4872	0.071964	1.3509
50	2880	0.072553	0.5434
100	488	0.072501	0.6136

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	1480	0.082201	0
5	16454	0.081359	1.0242
10	15100	0.080999	1.4618
50	8951	0.081294	1.1025
100	5810	0.081605	0.7251

Batch Size	**Iterations**	**TOL**	**% TOL**
Grad Desc	1480	0.079344	0
5	16454	0.076564	3.5028
10	15100	0.078128	1.5324
50	8951	0.078002	1.6908
100	5810	0.078249	1.3794

Batch Size	Iterations	TOL	% TOL
Grad Desc	4312	0.074469	0
5	48473	0.071685	3.7394
10	44180	0.071271	4.2952
50	25908	0.07181	3.5715
100	17210	0.072244	2.9883

Batch Size	Iterations	TOL	% TOL
Grad Desc	4312	0.070339	0
5	48473	0.070115	0.3178
10	44180	0.069322	1.4451
50	25908	0.069626	1.0142
100	17210	0.069599	1.0518

Dataset Size: 5000

Training Set

Batch Size	Iterations	SetTOL	% TOL
Grad Desc	47	0.076532	0
5	576	0.076088	0.5808
10	560	0.074921	2.1054
50	525	0.074392	2.7973
100	488	0.074366	2.8310

Test Set

Batch Size	Iterations	TOL	% TOL
Grad Desc	47	0.076126	0
5	576	0.076143	-0.0224
10	560	0.074527	2.1011
50	525	0.074014	2.7744
100	488	0.073904	2.9193

Batch Size	Iterations	TOL	% TOL
Grad Desc	144	0.075988	0
5	1724	0.072898	4.0668
10	1750	0.071950	5.3147
50	1621	0.071573	5.8113
100	1465	0.071640	5.7230

Batch Size	Iterations	TOL	% TOL
Grad Desc	144	0.079258	0
5	1724	0.076283	3.7544
10	1750	0.075329	4.9578
50	1621	0.075102	5.2433
100	1465	0.075155	5.1767

Batch Size	Iterations	TOL	% TOL
Grad Desc	433	0.073957	0
5	5254	0.073209	1.0114
10	5170	0.072285	2.2606
50	4822	0.072195	2.3814
100	4419	0.072278	2.2700

Batch Size	Iterations	TOL	% TOL
Grad Desc	433	0.077336	0
5	5254	0.076583	0.9728
10	5170	0.075843	1.9307
50	4822	0.075567	2.2866
100	4419	0.075765	2.0315

B SGD Variants results tables

B.1 $y(x) = sin(2\pi x)$ for $x \in (0, 1)$

Dataset Size: 500

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	4043	0.480686	0	0.567689	0
WD	1750	0.501295	-4.2873	0.576206	-1.5002
Brownian	3553	0.478845	0.383	0.560399	1.2842
Momentum	2547	0.355621	4.9969	0.559429	1.4551

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	11970	0.481247	0	0.509860	0
WD	5955	0.498333	-3.5503	0.524153	-2.8033
Brownian	10377	0.480402	0.1755	0.506486	0.6617
Momentum	10339	0.47978	0.3048	0.506261	0.7059

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	36960	0.489145	0	0.487579	0
WD	1981	0.518493	-4.5153	0.518493	-6.3404
Brownian	41193	0.489946	-0.1638	0.489134	-0.3190
Momentum	39799	0.488978	0.034	0.48773	-0.031

Dataset Size: 5000

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	459	0.53881	0	0.552098	0
WD	446	0.541567	-0.5115	0.552940	-0.1525
Brownian	487	0.533022	1.0743	0.547042	0.9158
Momentum	480	0.531606	1.3371	0.545198	1.2497

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	1488	0.517658	0	0.525803	0
WD	1576	0.52281	-0.9953	0.530462	-0.8861
Brownian	974	0.513448	0.8131	0.522573	0.6144
Momentum	1535	0.50837	1.7941	0.516471	1.7749

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	4503	0.527448	0	0.523169	0
WD	4675	0.537434	-1.8932	0.529532	-1.7685
Brownian	4995	0.517817	1.8260	0.509323	2.1153
Momentum	4438	0.520561	1.3059	0.514239	1.1707

B.2 $y(x) = sin(2\pi x)$ for $x \in (-1, 1)$

Dataset Size: 500

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	4474	0.579800	0	0.492069	0
WD	4456	0.609723	-5.1609	0.53431	-8.5845
Brownian	4024	0.578657	0.1971	0.50222	-2.063
Momentum	4358	0.564146	2.6999	0.499534	-1.5171

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	13352	0.470565	0	0.487199	0
WD	1307	0.500275	-6.3137	0.536795	-10.1799
Brownian	12662	0.472894	-0.4950	0.481312	1.2082
Momentum	12197	0.466016	0.9667	0.479199	1.642

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	40832	0.561578	0	0.504287	0
WD	1965	0.617452	-9.9494	0.599066	-16.8425
Brownian	39482	0.565488	-0.6963	0.517082	-0.8521
Momentum	35055	0.561025	0.0984	0.508465	-0.8285

Dataset Size: 5000

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	472	0.580602	0	0.583943	0
WD	510	0.584658	-0.6987	0.58089	0.5228
Brownian	449	0.575825	0.8227	0.582491	0.2486
Momentum	439	0.573604	1.2053	0.572118	2.0251

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	1583	0.553125	0	0.554699	0
WD	1403	0.561616	-1.5351	0.562099	-1.3341
Brownian	978	0.550659	0.4459	0.553459	0.2236
Momentum	1470	0.543579	1.7259	0.543455	2.027

Variant	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	4576	0.54566	0	0.541738	0
WD	1484	0.575414	-5.4528	0.569661	-5.1541
Brownian	4756	0.532731	2.3695	0.529022	2.3475
Momentum	4861	0.536266	1.7216	0.531849	1.8255

B.3 $y(x) = cos(3\pi(x-1))e^{2(x-1)}$ for $x \in (0,1)$

Dataset Size: 500

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	4428	0.068295	0	0.072257	0
WD	4272	0.069463	-1.71	0.076351	-5.6651
Brownian	3003	0.065687	3.8186	0.071176	1.4964
Momentum	4436	0.066595	2.4884	0.071230	1.4215

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	13441	0.064945	0	0.065820	0
WD	5835	0.07429	-14.3894	0.076073	-15.8429
Brownian	9915	0.067083	-3.2914	0.067605	-2.9470
Momentum	12709	0.064434	0.7866	0.06383	3.0241

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	39115	0.065086	0	0.066275	0
WD	3887	0.072112	-10.7938	0.070517	-6.4009
Brownian	41863	0.064605	0.7395	0.066455	-0.2715
Momentum	38349	0.064709	0.5804	0.065907	0.5554

Dataset Size: 5000

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	509	0.078566	0	0.078466	0
WD	492	0.078245	0.4084	0.078128	0.4306
Brownian	188	0.083684	-6.5137	0.08333	-6.1995
Momentum	486	0.076684	2.3961	0.076864	2.0411

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	1560	0.077386	0	0.077957	0
WD	1585	0.079463	-2.6836	0.080127	-2.7839
Brownian	1573	0.072901	4.9255	0.72901	6.4855
Momentum	1630	0.074253	4.0484	0.074691	4.1898

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	4672	0.073256	0	0.073928	0
WD	2629	0.079716	-8.818	0.080193	-8.4746
Brownian	3357	0.073172	0.1152	0.074085	-0.2116
Momentum	4797	0.069709	4.8427	0.070549	4.5712

B.4 $y(x) = cos(3\pi(x-1))e^{2(x-1)}$ for $x \in (-1,1)$

Dataset Size: 500

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	3910	0.067958	0	0.071499	0
WD	1152	0.069913	-2.8773	0.074582	-4.3101
Brownian	1592	0.070486	-3.7198	0.074682	-4.4512
Momentum	4099	0.067283	0.993	0.070632	1.2137

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	13194	0.064035	0	0.075105	0
WD	950	0.065745	-2.6716	0.077309	-2.9342
Brownian	1319	0.066336	-3.5937	0.077373	-3.0199
Momentum	12824	0.063271	1.1917	0.074694	0.5473

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	40880	0.065699	0	0.069849	0
WD	1853	0.067629	-2.9371	0.071290	-2.0639
Brownian	35976	0.065844	-0.2196	0.069856	-0.0106
Momentum	37597	0.065408	0.4433	0.069498	0.5022

Dataset Size: 5000

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	502	0.076305	0	0.077825	0
WD	470	0.075688	0.8076	0.077099	0.9338
Brownian	484	0.075245	1.3885	0.076933	1.1463
Momentum	329	0.076658	-0.4630	0.078269	-0.5699

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	518	0.074335	0	0.716448	0
WD	1421	0.074601	-0.3582	0.071748	-0.1439
Brownian	515	0.075477	-1.5364	0.73031	-1.9351
Momentum	1595	0.074370	-0.0463	0.071737	-0.1287

Batch Size	Opt Iterations	TOL	% TOL	Test TOL	Test % TOL
Base	4542	0.071868	0	0.070445	0
WD	4309	0.071942	-0.1036	0.070136	0.4388
Brownian	751	0.076000	-5.7500	0.073546	-4.4028
Momentum	3552	0.071498	0.5139	0.70264	0.2563

References

[1] Tom M Mitchell, *Machine Learning*, 1997, McGraw Hill.

[2] J. Han, M. Kamber, *Data mining: concepts and techniques*, 2000 Morgan Kaufmann Publishers.

[3] E. Kalapanidas, N. Avouris, M. Craciun, D. Neagu, *Machine learning algorithms: a study on noise sensitivity*, 2003.

[4] D. Wolpert, W. Macready, *No Free Lunch Theorems for Optimization* Evolutionary Computation (1997): 67-82.

[5] Tom M. Mitchell, *Generalization as Search*, Readings in Artificial Intelligence (1981): 517-42.

[6] Olivier Bousquet and Andre Elisseeff, *Stability and Generalization*, 2002, Journal of Machine Learning Research 2, ppt: 499-526

[7] E.L. Lehmann, G. Casella, *Theory of Point Estimation*, 1998, Springer

[8] G.E. Hinton, *Learning translation invariant recognition in massively parallel network*, PARLE: Parallel Architectures and Languages Europe. Lecture, Notes in Computer Science, ppt: 1-13, Springer-Verlag, Berlin, 1987.

[9] Bishop, C. M., *Pattern recognition and machine learning*, 2006, Springer

[10] C. Hennig, M. Kutlukaya, *Some thoughts about the design of loss functions* REVSTAT Statistical Journal, Volume 5, Number 1, March 2007, ppt: 1939.

[11] P. J. Bickel, B. Li, *Regularization in statistics* Sociedad de Estadistica e Inverstigacion Operativa, Vol. 15, No. 2, ppt: 271-344

[12] M. Nedrich, *An Introduction to Gradient Descent and Linear Regression*, Atomic Spin, 2014, https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression/

[13] P. Hansen, B. Jaumard and S.H. Lu, *An analytical approach to global optimization*, Mathematical Programming, 1991, ppt: 52-227

[14] J. Nocedal and S. Wright, *Numerical Optimisation*, 2006, Springer.

[15] V. S. Cherkassky, *Learning from data* 1998.

[16] A.C. Davidson, *Statistical Models*, 2003, Cambridge University Press.

[17] X. Ning, G. Karypis, *The Set Classification Problem and Solution Methods*, 2009, SIAM Data Mining, pp. 847-858.

[18] R.A. Fisher *The Use of Multiple Measurements in Taxonomic Problems*, 1936.

[19] David Kriesel, *A Brief Introduction to Neural Networks*, 2007, http://www.dkriesel.com

[20] William Clancey, *Classification Problem Solving*, 1984, STAN-CS-84-1018.

[21] T. Blue, M. Unser, *Wavelets Fractals and Radial Basis Functions* 2002, IEEE Transactions on Signal Processing, Vol. 50, No. 3.

[22] M. Fernandez-Redondo, J. Torres-Sospedra and C. Hernandez-Espinosa, *Training Radial Basis Functions by Gradient Descent*, The 2006 IEEE International Joint Conference on Neural Network Proceedings, 2006, pp. 756-762.

[23] Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar, *Foundations of Machine Learning*, 2012, The MIT Press.

[24] Zoubin Ghahramani, *Unsupervised Learning* 2004, http://www.gatsby.ucl.ac.uk/ zoubin.

[25] J.A. Hartigan *Clustering Algorithms* 1975, Wiley series in probability and mathematics.

[26] M.J. Orr, *Regularization in the Selection of Radial Basis Function Centers* 1995, Neural Computation, Vol. 7 Issue 3, ppt: 606-623.

[27] J. D. McCaffrey, *Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training*, 2013, https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/

[28] H. Masnadi-shirazi, N. Vasconcelos, *On the Design of Loss Functions for Classification: theory, robustness to outliers, and SavageBoost* 2008, Advances in Neural Information Processing Systems 21.

[29] S. Kotsiantis, D. Kanellopoulos, P. Pintelas, *Data Preprocessing for Supervised Learning*, 2006, International Journal of Computer Science, Vol. 1 No. 2, ppt 111117.

[30] P. Juszczak, D.M.J. Tax, R.P.W. Duin, *Feature scaling in support vector data description* 2002, Proc. ASCI 2002, 8th Annual Conf. of the Advanced School for Computing and Imaging, ppt: 95-102.

[31] Y. Yao, L. Rosasco, A. Caponnetto, *On Early Stopping in Gradient Descent Learning* 2007, A. Constr Approx (2007) 26: 289.

[32] Lutz Prechelt, *Early Stopping But When?*, 1999, Vol. 7700 of the series Lecture Notes in Computer Science ppt: 53-67.

[33] L. Bottou, *Stochastic Gradient Descent Tricks* 2012, Neural Networks: Tricks of the Trade, Volume 7700 of the series Lecture Notes in Computer Science, ppt: 421-436

[34] Ian Goodfellowith Yoshua Bengio and Aaron Courville, *Deep Learning*, 2016, MIT Press

[35] X. Zhu, C. Vondrick, C.C Fowlkes, et al. *Do We Need More Training Data?* Int J Comput Vis (2016) 119: 76. doi:10.1007/s11263-015-0812-2

[36] W. Xu, *Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent*, 2011, arXiv:1107.2490

[37] K. Marti, *Stochastic Optimization Methods, Applications in Engineering and Operations Research* 2015, Springer

[38] C.W. Gardiner *Handbook of Stochastic Methods: for Physics, Chemistry and the Natural Sciences*, 2004, Springer Series in Synergetics

[39] B.T. Polyak, *Some methods of speeding up the convergence of iteration methods*, USSR Computational Mathematics and Mathematical Physics (1964): 1-17.

[40] A. Halevy, P. Norvig, F. Pereira *The Unreasonable Effectiveness of Data* 2009, IEEE Intelligent Systems, Volume: 24, Issue: 2.

[41] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE (1998), vol. 86, no. 11, pp. 22782323.