Machine Learning Practical - Coursework 2

Luca Scimeca
s1344727

November 24, 2016

Contents

1 Introduction

This report describes the experiments run for *Coursework 2* of the course Machine Learning Practical at the University of Edinburgh. The coursework involves the exploration of three topics related to performing hand-digit image classification through Neural Networks (NN). The experiments are run on the MNIST data set containing 28x28 normalized pictures of hand-written digits. The first topic concerns the effect that the choice of a non-linear transformation has to training performance. The second set of experiments involves Batch Normalization (BN), more specifically it investigates how BN affects training and classification performance. For the last set of experiments, Convolutional Neural Networks (CNNs) are used to perform hand-written digit recognition. At first, a Convolutional Neural Network is implemented, trained and compared to previous classifiers. At last, the effect of a Batch Normalization layer in a CNN is observed.

2 Methods

This report builds upon previously run experiments [1]. The experiments are run on the MNIST dataset, containing 60000 28x28 normalized images of hand written digits. Unless otherwise stated, all experiments are run on a training set of 50000 images, using the remaining 10000 for validation purposes. Moreover, the networks are trained with batch gradient descent (batch size of 50) and perform a maximum of 100 epochs through the data. To compare models across experiments, and minimize the effects to performance that changing the learning rule has, the rule is for the most part fixed to a Momentum Learning rule with previously found stable values [1], i.e. a learning rate $\eta = 0.01$ and a moment coefficient $\alpha = 0.9$.

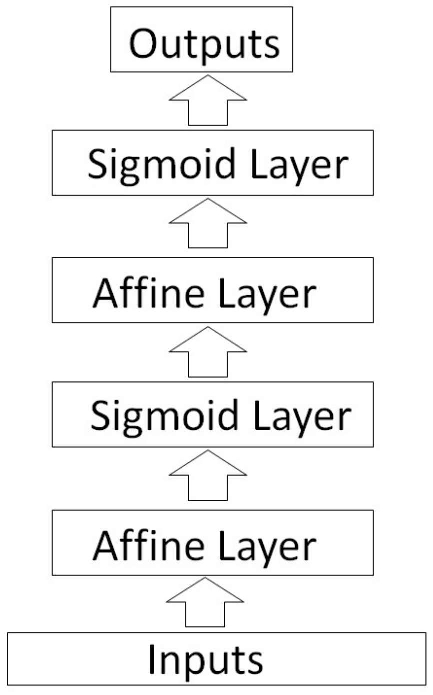
2.1 Non linearity effects

In the first experiment we test the effects of changing the non-linear transformation layer on Neural Networks; more specifically, we explore the saturation problem, in other words how some linear transformations (e.g. *Tanh* or *Sigmoid*) suffer from non accurate weight initializations. For the purpose of the experiments three Neural Networks were fit to the data, each featuring three layers: a 784 unit input layer, an 100 unit hidden layer and a 10 units output layer, ultimately doing a 1-of-10 classification of an input image. The hidden unit outputs for the three NNs were nonlinearly transformed through a *Sigmoid*, a *ReLu* and a *Tanh* function respectively. The networks were initialized with the same weight matrix, which was created by randomly sampling from a uniform distribution with an upper and lower given limit, deliberately chosen to be smaller and larger than known good initializations [1], with the objective of saturating the *Tanh* and *Sigmoid* function and comparing their behavior to the *ReLu* units. In order to avoid the learning rule to conceal the effects of the weight initialization, a simple momentum learning rule was preferred to any adaptive one. The hyper-parameters settings choice was based on previous results [1] ($\eta = 0.01$ and a moment coefficient $\alpha = 0.9$).

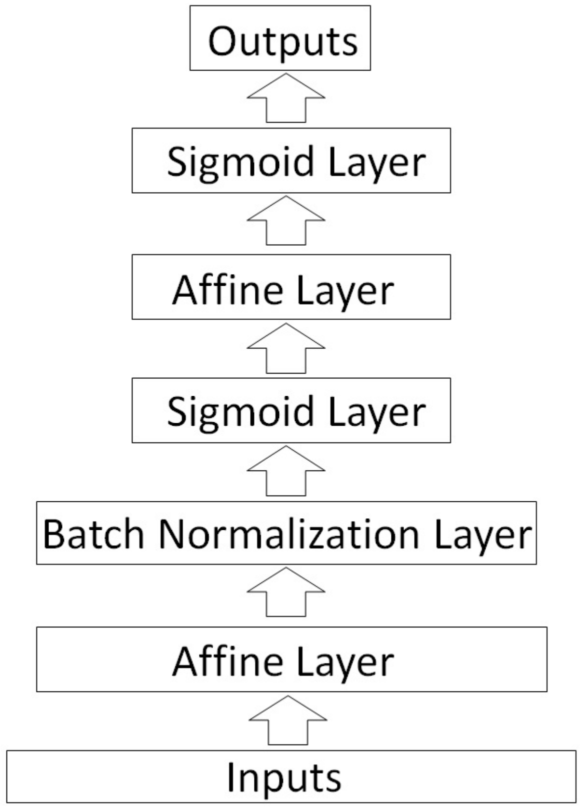
2.2 Batch normalization

The second set of experiments concerns batch normalization and its effect on Neural Network performance, more specifically, it is empirically assessed whether batch normalization improves classification accuracy on the MNIST data set, and, in tune with previous experiments, whether it can reduce the effects that weight initialization has on training networks. For the experiments, we create a new *Batch Normalization Layer* which applies batch normalization to the output of a fully connected affine layer. The new layer normalizes the outputs of the affine layer, and it scales and shits these before forwarding them to the next layer. The implementation is based on a similar version by C. Thorey [2]. After passing through the batch normal layer, the now normalized activations from the affine layer are non-linearly transformed.

To test batch normalization's robustness to weight initialization two Neural Networks are implemented, each comprising a 784 unit input layer, 2 fully connected hidden layers featuring 100 units and composed of an affine layer and a non-linear transformation, and a 10 unit output layer (see Figure 1a). The second NN additionally has a batch normalization layer prior to non-linearly transforming the output of the first affine layer (see Figure 1b). Given the purpose of the experiment, the non-linear transformation can be any between a *Tanh* and a *Sigmoid*, both with potential to saturate. A *Sigmoid* is chosen for the experiment.



(a)



(b)

Figure 1: The Figures show the architectures of the networks implemented to test the effects batch normalization has on MNIST image classification. The first NN (Figure (a)) is composed of two hidden layers with 100 units, both applying a Sigmoid transformation to their respective activations. The second NN (Figure (b)) features and additional layer prior to the first non-linear transformation, where the units are normalized, shifted and scaled through a *Batch Normalization Layer*.

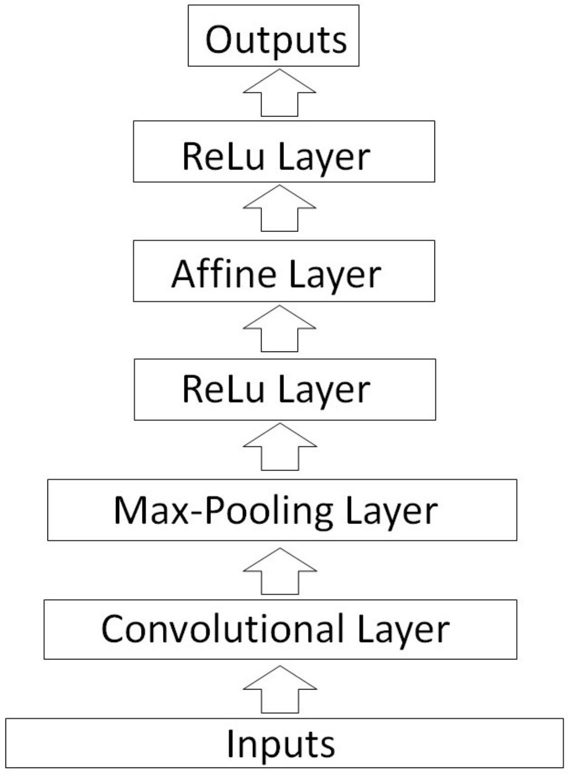
The choice of the learning rule is again a simple gradient descent with momentum, with the same hyper-parameter settings previously used (i.e. $\eta = 0.01$ and $\alpha = 0.9$). Ultimately, the weights matrices for the NN connections are initialized by randomly drawing from a Uniform distribution with a low bound of 0.1 and a high bound of 0.3, values previously found to saturate the units of a simple NN with a *Sigmoid* or *Tanh* transformation.

The second experiment in the set is meant to compare performance of networks trained with batch normalization and without. We take advantage of the ability of a network featuring a batch normalization layer to train at higher learning rates [3]. We train the NNs created in the previous experiment with a simple gradient descent learning rule with momentum and initialize all weights with a Glorot Uniform initialization. We perform an hyper-parameter search over the learning rate in the NN featuring the batch normalizer layer, finding the best settings for $\eta = 0.556$ (see Figure 5a), we fix α to 0.9 (a previously known good value for the hyper parameter) and we compare the performance over 100 epochs of the network, to the one with no batch normalization layer (also trained with the best parameter settings previously found [1], $\eta = 0.01$ and $\alpha = 0.9$).

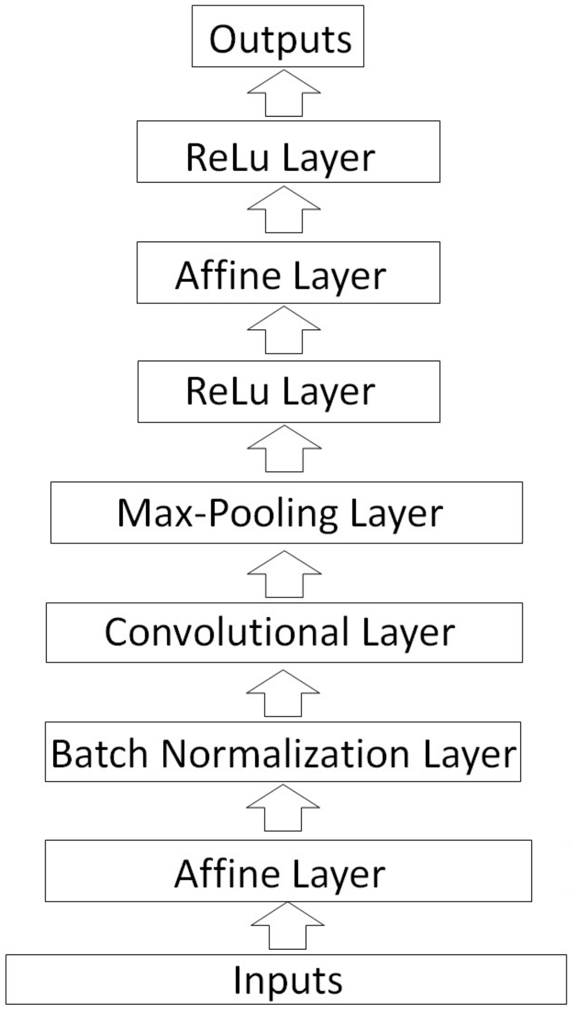
2.3 Convolutional Neural Networks

The third set of experiments explore Convolutional Neural Networks in the task of hand digit recognition. For the experiments a *Convolutional Layer* is implemented. The layer scans a set of input channels with a pre-determined sized kernel, and outputs a set of feature maps significant of the absence/presence of kernel detected features at different positions in the input channels. The *Convolutional Layer* implementation was in part based on a similar work by C. Thorey [4].

In the first experiment, we build and train a CNN and compare its performance to the performance achieved by the models in experiments run in previous coursework [1]. The CNN implemented features an input layer, duly reshaped to feed into a Convolutional Layer, followed by a *Max-Pooling* layer, a *ReLu* transformation and a final *Affine* and *ReLu* layer before doing the 1-of-k classification. In the model designed, the *Convolutional Layer* trains 24 features maps, by scanning each 28x28 image with a 5x5 kernel at strides of 1. The *Max-Pooling* layer outputs the maximum of each set of 2 input units and connects to a *ReLu* Layer. After the *ReLu* transformation, the 24 feature maps are fully connected to an Affine layer with now 6912 units, which after a final non-linear transformation connects to the output layer (see Figure 2a). To train the Network a Momentum learning rule was used, and the hyper-parameters set to values previously found to be stable (i.e. $\eta = 0.01$ and $\alpha = 0.9$).



(a)



(b)

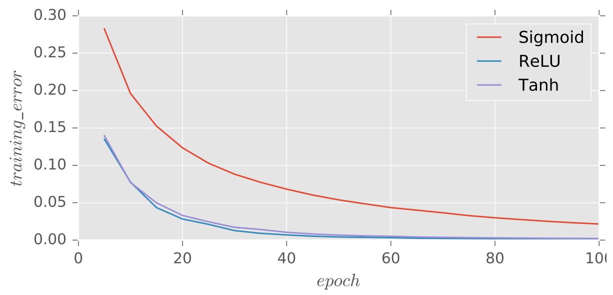
Figure 2: Figure (a) shows a CNN architecture used for MNIST image classification. In the experiments, the inputs are 784 dimensional, reshaped to feed a 28x28 dimensional channel in the *Convolutional Layer*. The layer builds 24 feature maps, feeding to a *Max-Pooling* layer which reduces the inputs to 24 24x12 dimensional channels, non-linearly transformed through a ReLu before fully connecting to an *Affine Layer*. Ultimately a final layer and transformation with 100 hidden units connects the Convolution to the outputs.
Figure (b) shows a network whose architecture is identical to the one displayed in Figure (a). In the experiments, however, the *Convolutional Layer* builds 16 feature maps, and ultimately connects to a 4608 fully connected affine layer. Moreover, a *Batch Normalization Layer* is placed between the inputs and the *Convolutional Layer*

The second experiment is meant to assess the usefulness of Batch normalization in CNNs. We run the experiments on a subset of the training a validation set containing 10000 and 2000 images respectively. The network previously designed is modified to retrieve 16 feature maps, and a second CNN model is implemented for comparison. In the second model a batch normalization layer is added between the input and the convolutional layer (see Figure 2b). The two networks are trained on the reduced training set and their performance over 50 epochs is compared.

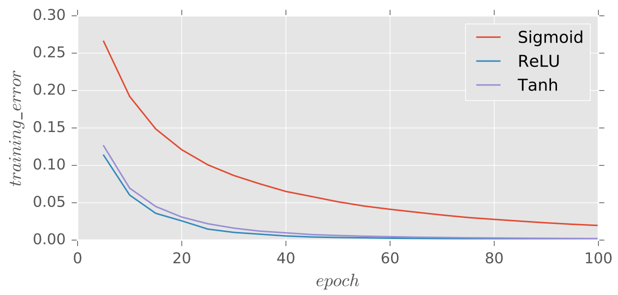
3 Results

3.1 Non linearity effects

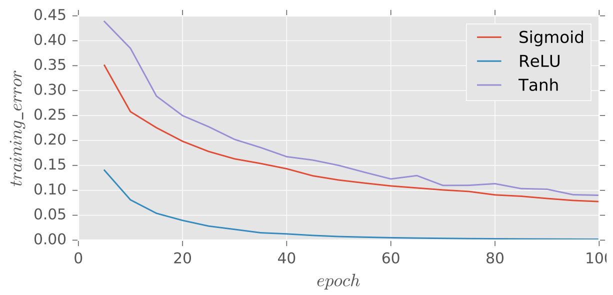
The three networks, featuring three different non-linear transformations were trained on the MNIST dataset and their error performance on both the training and validation set was subsequently drawn and compared (see Figure 3). As clear from the figures, although initially the behavior of each of the NNs seems to be sensible, as the elements in the initialized weight matrices become larger, the networks with a *Tanh* or *Sigmoid* transformation struggle to improve their performance over the epochs. The high weight initializations, in fact, saturate the units with said transformations. The saturation is here achieved by reaching values ≈ 1, where the steepness of the functions (and therefor their gradients) approaching their upper asymptotes are close to 0. Given the low gradients, it becomes unfeasible for gradient descent to move the weights towards the minimum loss. The NN featuring *ReLu* transformations, on the other hand, does not suffer saturation and although its performance is clearly worsened but non-ideal weight initialization it consistently manages to improve performance over the epochs in any one trial.



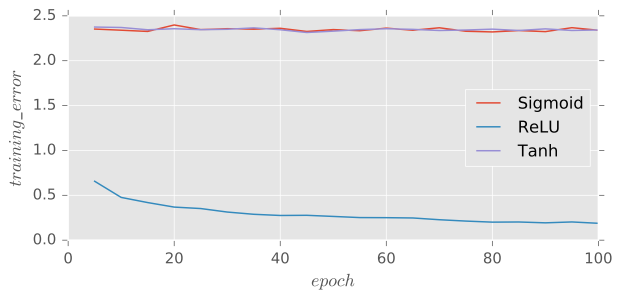
(a) $b_l = 0.000001$, and $b_h = .00001$



(b) $b_l = 0.001$, and $b_h = .01$



(c) $b_l = 0.01$, and $b_h = .1$



(d) $b_l = 0.1$, and $b_h = .3$

Figure 3: The Figures show the training and validation error over 100 epochs of three NN with a *Sigmoid*, *ReLu* and *Tanh* non-linear transformations applied to their hidden unit. Each figure shows the performance over a differently initialized weight matrix. The element of each of the matrices are drawn from a Normal distribution with different high (b_h) and low (b_l) bounds. As clear from Figure 3d, a too high weight initialization saturates the units of networks with *Sigmoid* and *Tanh* transformations, which do not manage to improve their performance on the validation and training set over the 100 epochs the algorithm runs in.

3.2 Batch normalization

We explore the effect batch normalization has on weight initialization by comparing the performance of two NNs initialized with weights previously found to saturate the units of similar networks. The two NNs, one of which featuring a batch normalization layer, were created and trained on the MNIST dataset. The training and validation performance over 100 epochs was ultimately drawn and compared (see Figure 4). As clear from Figure 4, although both networks suffer from the disproportional random initialization of the weights, the NN featuring a batch normalization layer manages to move down the gradient in weight space after $\approx$ 20 epochs, ultimately reaching its top performance within the 100 epochs of training.

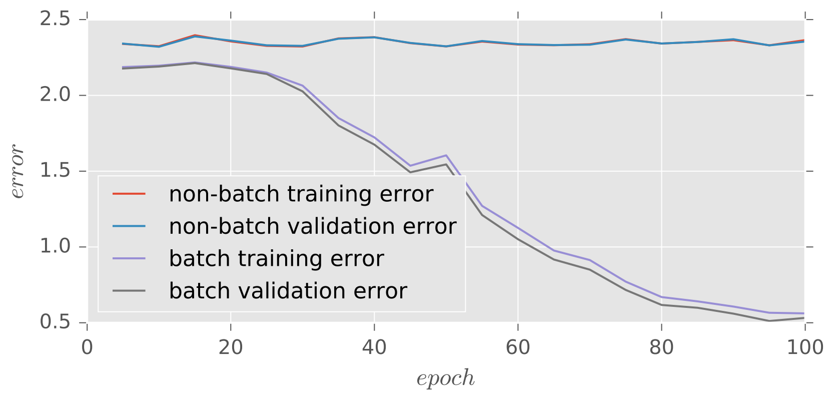
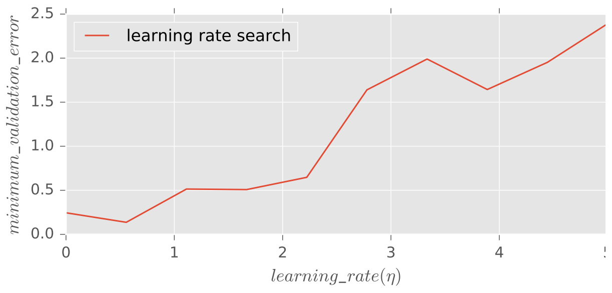
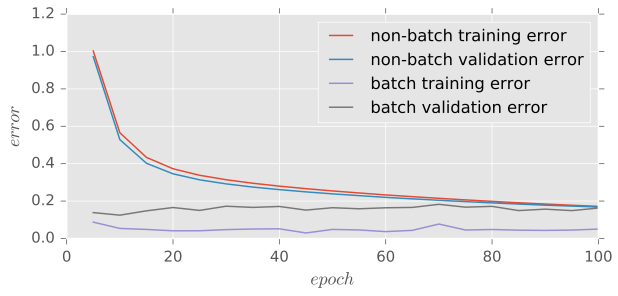


Figure 4: The Figure shows the training and validation error of two NN, identical if not for one featuring a batch normalization layer. The weights for both NN were randomly initialized from a Normal distribution with a low bound of 0.1 and a high bound of 0.3, an initialization previously found to be saturating the units of a similar Neural Network.
Batch normalization makes the network more robust against weight initialization, effectively managing to move the error down its gradient despite the disproportionate weight initialization.

The second experiment compares the performance of the two NNs when trained with the respective best hyper parameter settings. Figure 5b shows the training and validation performance of the two networks over 100 epochs. As the figure shows, by the end of the 100th epoch, the two network reach a similar performance on the validation set. The NN trained with a batch normalization layer, however, reaches its top performance after $\approx$ 10 epochs, as compared to the $\approx$ 100 epochs needed from the non-batch normalized network to reach its minimum validation error.



(a)

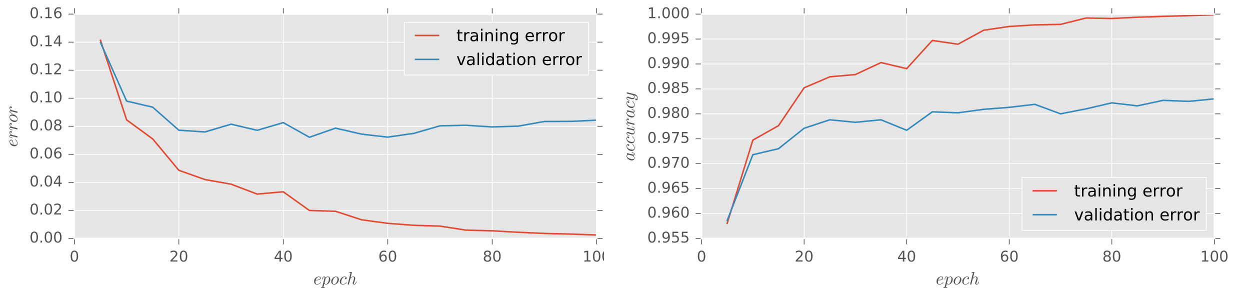


(b)

Figure 5: Figure (a) shows the minimum validation error achieved with different values of the learning rate hyper-parameter for a NN featuring a batch normalization layer. The error is minimized for $\eta = 0.556$, a value much bigger than the equivalent network with no batch normalization layer ($\eta = 0.01$).
Figure (b) shows the training and validation error, over 100 epochs, of two NNs, one of which features a batch normalization layer. Both NNs are trained with the best hyper-parameters settings, found with a search over each hyper-parameter space in turn. The NN with no batch normalization is trained with $\eta = 0.01$ and $\beta = 0.9$, while the network with the batch normalization layer is trained with $\eta = 0.556$ and $\beta = 0.9$.

3.3 Convolutional Neural Networks

The CNN designed is trained on the full training set and its performance is compared to the data gathered on models previously implemented (see Appendix I, Table 1).
Figure 6 shows the training and validation performance of the CNN over 100 epochs. The network reaches the lowest classification error on all algorithms so far implemented, both on the training ($Error_{train} = 0.00249$) and validation ($Error_{valid} = 0.07216$) set. Although both validation and training error were reduced, the increase in computation time was increase from $\approx 20min$ to $\approx 9hours$, an increase which might make the approach unusable for time-sensitive problems.



(a) (b)

Figure 6: The Figure shows the error and validation performance of a Convolutional Neural Network on the MNIST data classification The network features a convolutional layer with 24 feature maps and 5x5 kernel size, a Max-Pooling layer with a pool size of 2, a *ReLu* transformation, and a final hidden layer before the outputs. The network is trained with a moment based gradient descent rule where $\eta = 0.01$ and $\beta = 0.9$

Two CNNs, one of which featuring a batch normalization layer are trained and validated on a subset of the training and validation sets previously used, and their performance is compared. From Figure 7 it can be observed how adding a batch normalization layer to the designed model does not improve performance in any significant way. The two Networks *walk* down their respective gradients at different rates, but both reach their top performance after $\approx 40/45$ epochs over the data.

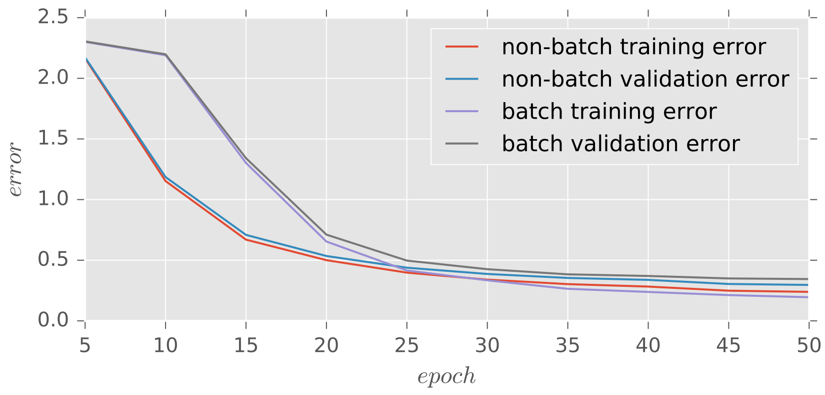


Figure 7: The Figure shows the training and validation error of two CNN, identical if not for one featuring a batch normalization layer. Both CNNs use a Momentum Learning rule with $\eta = 0.01$ and $\beta = 0.9$ and weights randomly initialized from a Glorot Normal distribution.

4 Discussion

The first set of experiments was meant to explore the effect different non-linear transformations of layers have on classification; here, the focus was on the saturation problem. As expected, *Sigmoid* and *Tanh* units saturate with large weight elements, effectively getting *stuck* on a near flat surface with gradients ≈ 0, and not improving performance on a dataset over many epochs. *ReLu* transformations do not suffer from saturation and are capable of improving performance almost independently from weight initialization. However, *ReLu* transformation might suffer from weight deactivation when unit values become ≤ 0, a characteristic which can be either positive or negative, depending on the application. Moreover, weight initialization robustness might not be the only parameter to choose a transformation by, and other non-linearities could be preferred for different advantages. Amongst others, a *Sigmoid* transformation, would have the advantage of interpretability, potentially being able to refer to the unit outputs as probabilities of features being detected; on the other hand, *Tanh* transformations give the additional advantage of negative outputs, a possibly useful feature depending on the application.

The second set of experiments was focused on batch normalization, its effects on training and its potential for reducing weight initialization problems. Training two networks, one, like in the previous set of experiments, known to saturate with a large weight initialization, and another with the same initialization but featuring a batch normalization layer. Results show how batch normalization manages, differently from its non-batch normalized version, to *pull* the network from the saturated state. The network, after relatively few epochs (≈ 20), starts improving performance on a training and validations set, ultimately reaching its top performance within the 100 epochs it runs on. Training the same two networks with their best hyper parameter settings and a known good initialization [1], shows how batch normalization can allow a NN to reach its top performance on a dataset many epochs before its non-batch normalized version (≈ 90 epochs before).
Shifting and rescaling the outputs/inputs of a NN layer turns out to be a powerful idea, and with some non-overwhelming computation complexity introduction, can improve both performance and training time.

In the last set of experiments we look at Convolutional Neural Networks and their performance on MNIST image classification. The designed CNN, featuring a Convolutional Layer with 24 feature maps improves on the previous state of the art performance on the validation set (see Appendix I, 1, *Momentum scheduling*) of ≈ 0.00804. The performance improvement, however, comes at a non-negligible increase in computational cost which, depending on the application, might not be worth it. More tangible improvements might be reached with deeper CNN, and/or increasing the number of feature maps learned by the model, both however introducing yet more complexity/computation time to the model.
Batch normalization does not improve performance on a baseline CNN. We argue how its limited usefulness in the experiment might be biased by the model and the application. More specifically, as the CNN trained featured only one Convolutional layer, the shift and rescale of the inputs might not have had any tangible effect on classification performance. Future experiments should assess whether BN can be useful in Deep Convolutional Neural Networks, where a shift and rescale of the activations in middle hidden units may turn out to be fundamental to reach minima over less epochs.

References

[1] L. Scimeca. Machine learning practical - coursework 1. 11 2016.

[2] C. Thorey. What does the gradient flowing through batch normalization looks like? `http://cthorey.github.io/backpropagation/`, 2016.

[3] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[4] C. Thorey. Note on the implementation of a convolutional neural networks. `http://cthorey.github.io./backprop_conv/`, 2016.

APPENDIX I: Tables

Table 1: The table shows the performance of the algorithms implemented in previous experiments [1].

Gradient Descent with:	**Validation Error (Minimum)**	**Accuracy (Maximum)**	**Average Run time**
Constant learning rate	0.0811	0.9775	143.3
Exponential learning rate scheduling	0.0812	0.9760	149.9
Momentum	0.1675	0.9537	147.8
Momentum scheduling	0.0802	0.9766	153.7
RMSProp	0.0893	0.9779	236.1
Adam	0.0895	0.9784	259.4